V. Gervasi, S. Pelagatti, S. Ruggieri, F. Scozzari, A. Sperduti

Programmazione di Sistema in Linguaggio C

Esempi ed esercizi

Corsi di Laurea e Diploma in Informatica
Università di Pisa

A.A. 2003-2004

Premessa

L'obiettivo di questa dispensa è di raccogliere in un unico volume le esercitazioni svolte nelle lezioni di *Laboratorio di Programmazione di Sistema* dei Corsi di Laurea e Diploma in Informatica dell'Università di Pisa. La dispensa non si propone come un testo autocontenuto, ma più semplicemente come una guida alla selezione ed allo studio degli argomenti presentati nei libri di testo del corso:

[**Kelley**] A. Kelley, I. Pohl. *C: Didattica e Programmazione*, Addison-Wesley, 1996.

[**Glass**] G. Glass, K. Ables. *UNIX for Programmers and Users*, Prentice Hall, 1999.

Per la trattazione di argomenti non contenuti in [**Glass**], viene invece riportata una presentazione più estesa.

La dispensa contiene una breve introduzione al linguaggio C, il cui scopo è di fornire le conoscenze necessarie sul linguaggio per permettere allo studente di realizzare brevi programmi da utilizzarsi per la programmazione di sistema, oggetto del corso. Come prerequisito, si suppone che lo studente abbia già acquisito conoscenze di programmazione in linguaggio Java.

Indice

Capitolo 1

Introduzione

Il C è un linguaggio di programmazione di uso generale, originariamente sviluppato per la scrittura del sistema operativo operativo Unix, ed oggi disponibile su tutte le maggiori piattaforme (Unix, Linux, Windows, MacOS, etc.). In queste note verrà descritto brevemente il linguaggio C come definito dall'ANSI (American National Standards Institute, organismo che si occupa della standardizzazione dei linguaggi), e più recentemente dall'ISO (International Organization for Standardization), e quindi faremo riferimento al linguaggio ANSI/ISO C.

Il C è un linguaggio del paradigma imperativo e fornisce i costrutti fondamentali per il controllo del flusso del programma (while, do, for, if-else e switch) e la definizione di funzioni. A differenza dei linguaggi orientati agli oggetti, come Java, il C non supporta le nozioni di oggetto, classe, e nessun meccanismo di ereditarietà. Una caratteristica peculiare del linguaggio C risiede nell'utilizzo dei puntatori. Un puntatore è una variabile che contiene l'indirizzo di un'altra variabile. I puntatori permettono di accedere direttamente a delle locazioni di memoria e di modificarle. Quando i puntatori sono passati come argomenti nelle chiamate di funzioni, si realizza il passaggio dei parametri per riferimento (contrapposto al passaggio per valore, standard dei linguaggi C e Java). Fondamentalmente, il C è un linguaggio semplice da utilizzare, veloce, relativamente a basso livello, e che quindi si presta particolarmente bene per la programmazione di sistema. Ricordiamo che Unix, Linux e un gran numero di applicazioni per tali sistemi sono scritti in C.

Il testo di riferimento per il linguaggio C che verrà utilizzato nel corso è:

A. Kelley, I. Pohl. *C: Didattica e Programmazione*, Addison-Wesley, 1996.

e come manuale di consultazione si consiglia:

B.W. Kernigham, D.M. Ritchie. *Linguaggio C (seconda edizione)*, Jackson, 1989.

1.1 Un programmino C

Quello che segue è un semplice programma C che stampa le cifre da 0 a 9. Per stampare viene utilizzata la funzione `printf`, che verrà discussa in seguito.

```
#include <stdio.h>
#define CIFRE 9
int main()
{
  int i;
  for (i=0; i<=CIFRE; i++)
    printf("%d \n",i);
  return(0);
}
```

La prima riga del programma indica che deve essere utilizzata la libreria standard di I/O, nella quale sono definite le principali funzioni per la gestione dell'input/output. Questa libreria è essenziale per la maggior parte dei programmi C, e permette di utilizzare le funzioni standard di I/O per leggere i caratteri da tastiera e stampare a video.

La seconda riga contiene la definizione di una macro (o costante simbolica). Le macro rappresentano un meccanismo per sostituire un nome simbolico con una stringa. Nel nostro caso, il compilatore si occupa di sostituire in tutto il testo del programma la stringa `CIFRE` con il suo valore, prima che la compilazione vera e propria abbia inizio.

La terza riga contiene la dichiarazione della funzione `main`, che costituisce il corpo principale di ogni programma C. La funzione `main` deve obbligatoriamente restituire un intero (tipicamente l'intero restituito viene utilizzato per segnalare condizioni di errore) e può prendere come argomenti eventuali dati di input al programma. Nell'esempio non ha nessun argomento, in quanto il programma non riceve nessun input. Il testo della funzione `main` è racchiuso tra parentesi graffe, inizia con la dichiarazione di una variabile di tipo intero, e prosegue con il ciclo `for`, il quale utilizza la funzione di libreria `printf` per stampare a video. La dichiarazione della funzione `printf` è contenuta nel file `stdio.h` che viene incluso nella prima linea.

Il programma termina con il comando `return(0)` che restituisce l'intero 0, per segnalare assenza di errori in fase di esecuzione.

1.2 Struttura di un generico programma C

In generale, la struttura di un programma C contiene un numero arbitrario di `#include` e di `#define`. Queste linee iniziano sempre con il simbolo `#` e sono dette *direttive al preprocessore*. Il programma prosegue con la eventuale dichiarazione di variabili e funzioni globali, ed infine

troviamo la funzione `main`. Lo scheletro di un tipico programma C è del tipo:

```
#include <filename>
      ⋮
#include filename
      ⋮
#define nome-macro testo
      ⋮
dichiarazioni di variabili globali
dichiarazioni di funzioni
int main(int argc, char *argv[])
{
   variabili locali
   corpo del programma
   return(0);
}
```

Le direttive `#include <filename>` e `#include filename` sono utilizzate per includere nel programma il file *filename*, che tipicamente è chiamato *header*. Per convenzione, i nomi degli headers hanno il suffisso `.h`. Utilizzando la direttiva `#include <filename>` il compilatore cerca il file *filename* tra gli headers standard del C (su Linux nella directory `/usr/include`). Invece, utilizzando la direttiva `#include filename` il compilatore cerca il file nella directory dove risiede il file sorgente. Questa modalità permette di suddividere un programma C in tanti file separati, che poi verranno opportunamente inclusi al momento della compilazione. Tipicamente, un header contiene le direttive al preprocessore, le definizioni di tipi e le dichiarazioni di funzioni, che sono così raggruppate in un unico file, distinto dal testo del programma. Il file che contiene il testo del programma conterrà invece la direttiva al preprocessore di includere l'header, che di solito viene posta all'inizio del programma. Ad esempio, creiamo un file `dati.h` che contiene le seguenti righe:

```
#include <stdio.h>
#define CIFRE 9
```

e poi salviamo nel file `stampacifre.c` il seguente programma:

```
#include "dati.h"
int main()
{
  int i;
  for (i=0; i<=CIFRE; i++)
    printf("%d \n",i);
  return(0);
}
```

Il compilatore si preoccuperà di includere l'header `dati.h` prima di iniziare la compilazione del programma `stampacifre.c`.

Infine, le linee che iniziano con `#define` permettono di definire delle macro. La sintassi generale di una macro è:

```
#define nome(arg1,...,argn) testo
```

La dichiarazione di una macro assomiglia ad una chiamata di funzione, ma in realtà è semplicemente un meccanismo per sostituire un nome simbolico con una stringa. Ad esempio, con la definizione:

```
#define SOMMA(X,Y) X+Y
```

si ottiene l'effetto di sostituire la linea di programma:

```
i = SOMMA(a,b);
```

con la linea:

```
i = a+b;
```

indipendentemente dal tipo delle variabili `a` e `b`. La definizione di una macro deve iniziare e terminare su una stessa linea. Alternativamente, può essere spezzata su più linee utilizzando la barra rovesciata posta alla fine delle righe da continuare. Ad esempio, nel frammento:

```
#define MACROMOLTOLUNGASBAGLIATA     printf("Questa definizione");
                                     printf("di macro e' sbagliata");

#define MACROMOLTOMOLTOLUNGACORRETTA printf("Questa definizione"); \
                                     printf("di macro e' corretta");
```

la prima definizione di macro è sbagliata e stampa solo la prima riga, e la seconda è la versione corretta, che stampa entrambe le righe. Questo uso della barra rovesciata è tipico: infatti, in molti contesti, la barra rovesciata indica che si vuole rimuovere la funzione speciale del carattere successivo. Così, la sequenza "barra rovesciata + a-capo" toglie il significato speciale di terminatore di riga al carattere "a-capo", e lo trasforma in

un semplice spazio bianco. Analogamente, all'interno di una stringa delimitata da virgolette la sequenza "barra rovesciata + virgolette" toglie alle virgolette il loro significato speciale di delimitatore di stringa, e le trasforma in un carattere ordinario: per esempio,

```
printf("Queste sono virgolette \"...");
```

stampa il testo

```
Queste sono virgolette "...
```

Le macro differiscono dalle chiamate di funzioni, oltre al fatto di non essere tipate, anche perché sono sostituite sintatticamente nel testo del programma. Per questo motivo, si possono creare degli effetti collaterali dovuti alla precedenza degli operatori. Ad esempio, la linea:

```
media = SOMMA(a,b)/2;
```

viene sostituita con:

```
media = a+b/2;
```

e, per la precedenza degli operatori, viene eseguito il calcolo `a+(b/2)`. Quindi, nell'uso delle macro, occorre prestare particolare attenzione all'uso delle parentesi.

Esercizio. Cosa stampa il seguente programma?

```
#include <stdio.h>
#define PRODOTTO(X,Y) (X*Y)
int main()
{
  int i,j = 2;
  i = PRODOTTO(j+1,3);
  printf("%d \n",i);
  return(0);
}
```

Si riscriva la macro affinché il programma stampi il risultato "intuitivamente" corretto. □

La funzione `main` deve essere dichiarata di tipo intero e può prendere, opzionalmente, due o tre argomenti, che contengono i parametri passati al programma C dalla linea di comando e le variabili d'ambiente. Questi parametri saranno trattati in dettaglio nel capitolo 5.7. La definizione della funzione `main` contiene le dichiarazioni delle variabili locali, seguite dal corpo del programma.

Il C è un linguaggio *free-format*, cioè permette di scrivere programmi utilizzando un numero arbitrario di spazi e tabulazioni. Al fine di indentare le linee del programma per una migliore leggibilità, l'editor *emacs* mette a disposizione la modalità `c-mode`. Tale modo viene attivato

automaticamente ogniqualvolta il suffisso del file è `.c`, e può essere attivato con `M-x c-mode` (il meta carattere `M` solitamente è il tasto `Esc`). Il tasto `Tab` effettua l'indentazione automatica delle linee di programma. Inoltre, il C è *case sensitive*, e quindi lettere maiuscole e minuscole sono caratteri distinti.

1.3 Compilazione di un programma C

I programmi C tipicamente iniziano con una serie di `#include` e `#define`. Al fine di includere le informazioni presenti negli headers e applicare le sostituzioni definite dalle macro, la compilazione di un programma C avviene in due passaggi. Al primo passo (detto *preprocessing*) vengono valutate le linee che iniziano con il simbolo `#`. Nell'esempio precedente, il preprocessore cerca nella directory standard delle librerie (su Linux in `/usr/include`) il file `stdio.h` e lo include nel programma. Quindi sostituisce nel testo del programma la stringa `CIFRE` con `9`, e poi, nel secondo passo, avviene la compilazione vera e propria del programma così trasformato.

Dopo aver editato e salvato il testo del programma nel file *sorgente*`.c`, per compilarlo si utilizza il comando

`gcc -Wall -g` *sorgente*`.c`

che crea l'eseguibile `a.out`[1]. L'opzione `-Wall` (che sta per "Warning: all") stampa ulteriori messaggi di *warnings*, oltre ai normali messaggi di errore del compilatore, e l'opzione `-g` include informazioni utili al debugger. Per eseguire il programma è sufficiente digitare `a.out`. Per creare il file eseguibile di nome *compilato*, si utilizza:

`gcc -Wall -g` *sorgente*`.c -o` *compilato*

Così facendo non viene creato l'eseguibile `a.out`, e l'output della compilazione viene memorizzato nel file *compilato*. Equivalentemente, può essere utilizzato il comando `make`, uno strumento che risulta particolarmente utile per automatizzare la compilazione di sistemi di grandi dimensioni. Occorre innanzitutto creare un file di nome `makefile` che conterrà le due linee seguenti:

```
CC = gcc
CFLAGS = -Wall -g
```

Il file deve risiedere nella stessa directory del file sorgente. La prima riga indica a `make` che vogliamo usare il comando "gcc" come compilatore C, mentre la seconda specifica le opzioni che vogliamo passare al comando definito con `CC`. Digitando:

[1] Per motivi storici, `a.out` è il nome di default che il compilatore assegna al risultato della compilazione quando non viene richiesto un nome specifico con l'opzione `-o` sulla riga di comando.

`make` *file*

si ottiene lo stesso effetto di `gcc -Wall -g` *file*`.c -o` *file*. Notate che il nome del file passato al comando `make` non contiene il suffisso `.c`. `make` sa già che, per ottenere un eseguibile da un file sorgente C, occorre eseguire il comando

```
${CC} ${CFLAGS} x.c -o x
```

in cui la notazione `${V}` indical il valore della variabile `V`, e x è il nome del file da compilare (`make` offre molte altre possibilità: si consulti il manuale in linea per i dettagli).

Utilizzando l'opzione `-E` il compilatore effettua il *preprocessing*, senza compilare il programma. Ad esempio, con:

`gcc -E` *sorgente*`.c`

il compilatore stampa a video il risultato della fase di *preprocessing*, che comprende l'inclusione degli headers e la sostituzione delle macro.

Il nome `gcc` è l'acronimo di GNU Compiler Collection, il compilatore free e open-source della GNU. La documentazione del compilatore e il relativo manuale sono disponibili al sito `http://www.gnu.org/software/gcc`.

1.3.1 Gli errori del compilatore

Il compilatore GNU riporta gli errori e le *warnings*. Gli errori si riferiscono ai problemi del codice per cui è impossibile procedere alla compilazione del programma. Per facilitare la ricerca dell'errore, il compilatore `gcc` mostra il nome del file sorgente e il numero della linea dove (presumibilmente) si è verificato l'errore. A volte può capitare che l'errore sia localizzato nelle righe che precedono quella indicata dal compilatore ed è quindi buona norma controllarle. Ad esempio, un programma dove manca la dichiarazione della variabile `n`, che viene poi usata nella funzione `main`, riporterà un errore del tipo:

```
mioprogramma.c:  In function ‘main’:
mioprogramma.c:13:  ‘n’ undeclared (first use in this function)
mioprogramma.c:13:  (Each undeclared identifier is reported only once
mioprogramma.c:13:  for each function it appears in.)
```

L'errore indica che nel file sorgente `mioprogramma.c`, nella funzione `main`, alla linea `13` viene usata la variabile `n` che non è stata dichiarata. Inoltre, avvisa che gli errori per variabili non dichiarate vengono riportati una sola volta per ogni funzione dove viene utilizzata la variabile, e il numero di linea si riferisce al primo uso della variabile nella funzione.

Gli errori di sintassi sono riportati dal compilatore come `parse error`. Ad esempio il seguente errore:

```
mioprogramma.c:11:  parse error before 'int'
```

riporta che nel file `mioprogramma.c` è stato trovato un errore di sintassi alla linea `11`, prima della stringa `int`.

Le *warnings* descrivono delle condizioni "inusuali" che *possono* indicare un problema, sebbene la compilazione possa procedere (e infatti, nel caso vengano riportate solo *warnings*, il codice compilato viene ugualmente prodotto). La *warning* riporta il nome del file sorgente, il numero di linea e la parola "`warning`", per distinguerla da un errore. Ad esempio, un programma dove manca l'istruzione `return` all'interno del `main` riporterà una *warning* come la seguente:

```
mioprogramma.c:  In function 'main':
mioprogramma.c:16:  warning:  control reaches end of non-void
function
```

Il messaggio informa che nel file `mioprogramma.c`, nella funzione `main` viene raggiunta la fine della funzione (la quale ricordiamo restituisce sempre un `int`, e quindi non è dichiarata come `void`) senza incontrare l'istruzione `return`. Il numero di linea `16` indica la fine del `main`, dove il compilatore si sarebbe aspettato di trovare il `return`. Tale problema viene riportato come *warning* e non come errore perché il compilatore è comunque in grado di generare il codice compilato, dove ovviamente il valore ritornato dal `main` sarà indefinito, ma ciò non rende impossibile la compilazione. Si noti infine che tale *warning* viene riportata solo nel caso in cui si utilizzi l'opzione `-Wall` nella compilazione. Si consiglia vivamente di utilizzare sempre tale opzione in tutte le compilazioni, al fine di facilitare l'individuazione di possibili sorgenti di errori.

Capitolo 2

Tipi di dati

[Kelley, Cap. 3]

2.1 Tipi primitivi

I tipi di dato scalari messi a disposizione dal linguaggio C sono:

`char`	carattere
`short`	intero corto
`int`	intero
`long`	intero lungo
`float`	numero decimale
`double`	numero decimale in doppia precisione
`long double`	numero decimale in extra precisione

Le conversioni tra i tipi di dati sono effettuate automaticamente. Per rendere esplicita una conversione, si utilizza il meccanismo di *casting*, scrivendo il tipo tra parentesi prima del valore. Ad esempio, con:

```
i = (int) j;
```

alla variabile `i` viene assegnato il numero intero derivante dall'interpretazione (come intero) della zona di memoria assegnata alla variabile `j`.

I tipi interi possono essere di lunghezza arbitraria, a seconda della piattaforma utilizzata. Utilizzando Linux su piattaforma Intel, il tipo `int` occupa 4 bytes, esattamente come il tipo `long`. Per conoscere l'occupazione in bytes di un tipo, si utilizza la funzione `sizeof`: ad esempio `sizeof(int)` restituisce 4. I tipi `char` e `int` possono essere qualificati `signed` (default) oppure `unsigned`. Le variabili di tipo `unsigned char` assumono valori compresi tra 0 e 255, mentre le varibili `signed char` utilizzano i valori da -128 a +127. Analogamente

per il tipo `int`, i numeri `unsigned int` assumono solo valori positivi o nulli. Gli interi possono essere denotati in vari modi: nella normale notazione decimale (es.: 65); in ottale, premettendo uno 0 (es.: 0101 = 65); in esadecimale, premettendo "0x" (es.: 0x41 = 65); o anche come costanti carattere delimitate da apici, secondo il codice ASCII[1] (es.: 'A' = 65).

Notate che, diversamente dal Java, non ci sono i tipi predefiniti `boolean` e `byte` ed, in generale, le variabili non inizializzate non producono nessun errore a tempo di compilazione. Inoltre, non esiste l'oggetto `String`, e le stringhe verranno realizzate come array di caratteri (si veda la sezione 2.4).

Poiché alcune caratteristiche del linguaggio C non sono completamente fissate nello standard ANSI/ISO (ad esempio l'occupazione dei vari tipi), si è reso necessario differenziare le librerie che sono completamente fissate dallo standard, da quelle che riguardano i dettagli della piattaforma specifica. Ad esempio, in `limits.h` si trovano le macro `#define CHAR_MIN (-128)` e `#define CHAR_MAX 127`. Le macro che definiscono i limiti si trovano in una posizione standard nel file system, nota al compilatore. Per esempio, nella distribuzione *RedHat* di Linux, queste ultime librerie si trovano in `/usr/lib/gcc-lib/i386-redhat-linux/*/include`, dove `*` varia a seconda della versione utilizzata. Il programmatore può quindi includere `limits.h` con `#include <limits.h>` e scrivere il programma in modo che si adatti automaticamente ai valori contenuti nelle macro (per esempio, usando `CHAR_MIN` e `CHAR_MAX` per scoprire a tempo di compilazione la gamma di valori ammissibili per un carattere).

2.2 Tipi non primitivi

I tipi di dati non primitivi sono gli array, le struct e le union. Gli array sono degli aggregati di variabili dello stesso tipo. La dichiarazione di un array è della forma:

```
char parola[10];
```

In questo modo si dichiara la variabile `parola` come un array di 10 caratteri. Il primo carattere è contenuto in `parola[0]` e l'ultimo in `parola[9]`. Il C non controlla gli accessi agli array, quindi possiamo accedere all'elemento `parola[15]` senza che il compilatore produca nessun errore. Ovviamente, questo accesso è errato ed il risultato è imprevedibile. Inoltre possiamo definire array multidimensionali come:

```
char frase[20][10];
```

che definisce una variabile frase di tipo array con 20 elementi (numerati da 0 a 19), ed ogni elemento è a sua volta un array di 10 caratteri (numerati da 0 a 9). Per accedere al quarto

[1] Il tipo di codifica è definito dalla piattaforma; i codici ASCII, ISO-8859-1 (set Europa occidentale) o ISO-8859-15 (set Europa occidentale con il simbolo dell'Euro) sono i casi pi'u comuni.

carattere del secondo elemento si scrive `frase[1][3]`. Gli array possono essere inizializzati nelle dichiarazioni, con la lista dei valori, separati da virgole, racchiusa tra parentesi graffe, ad esempio:

```
int tabella[4] = {10,23,34,45};
```

L'inizializzazione di un array multidimensionale corrisponde ad inizializzare ogni elemento dell'array con una lista. Ad esempio:

```
int a[2][3] = {
  {11, 12, 13},
  {21, 22, 23},
};
```

inizializza la prima riga dell'array con i valori `11, 12, 13` e la seconda riga con `21, 22, 23`. Equivalentemente, ma meno elegantemente, si può utilizzare:

```
int a[2][3] = {11, 12, 13, 21, 22, 23};
```

che produce lo stesso risultato. Infatti gli elementi di un array sono memorizzati per righe, in indirizzi consecutivi di memoria. Quindi, nell'esempio precedente, l'array è memorizzato con il seguente ordine:

```
a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2]
```

Oltre agli array, il C mette a disposizione il tipo `struct`, per aggregare variabili di tipo diverso. Ad esempio, con la dichiarazione:

```
struct laboratorio {
    char orario[20];
    char aula;
    int studenti;
  } lps;
```

si dichiara il tipo `struct laboratorio`, formato da 3 campi eterogenei, e contemporaneamente viene dichiarata la variabile `lps` di tipo `struct laboratorio`. Il tipo può essere riutilizzato per successive dichiarazioni di variabili, con:

```
struct laboratorio lpr;
```

Per accedere ai campi di una variabile di tipo `struct` si utilizza l'operatore '.' infisso. Ad esempio `lps.aula` è una variabile di tipo carattere. Nella sezione 5.1 vedremo un metodo alternativo per accedere ai campi di una `struct` in presenza di puntatori.

Per creare nuovi tipi si utilizza `typedef`. Ad esempio:

```
typedef  struct {
    char orario[20];
    char aula;
    int studenti;
  } laboratorio;
```

definisce il tipo laboratorio come una `struct` di 3 campi. Dichiarando la variabile `lps` di tipo `laboratorio` con:

```
laboratorio lps;
```

si ottiene lo stesso effetto della dichiarazione precedente. La sintassi generale di `typedef` è:

`typedef` *tipo variabile_di_tipo* ;

e definisce un nuovo tipo con nome *variabile_di_tipo*.

Infine, il tipo `union` è simile al tipo `struct` con la differenza che tutti i campi sono memorizzati a partire dalla stessa locazione di memoria. Ad esempio, con la dichiarazione:

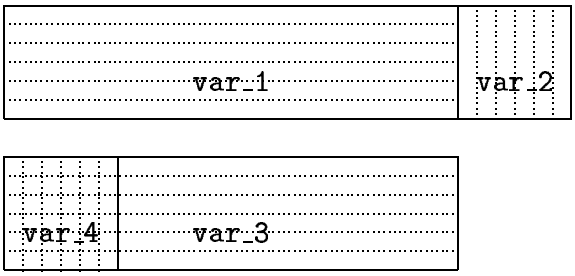
```
union atomo {
    int posizione[3];
    long velocita;
  } j;
```

si definisce la variabile `j` di tipo `atomo`. Entrambe le variabili `j.posizione` e `j.velocita` sono memorizzate a partire dalla stessa locazione di memoria. Quindi l'occupazione di un tipo `union` è data dalla massima occupazione dei suoi campi. Nel nostro esempio, il tipo `atomo` occupa 12 bytes. È compito del programmatore ricordarsi il valore correntemente utilizzato nelle variabili di tipo `union` ed usarlo correttamente. Ad esempio, le variabili `var_struct` e `var_union` dichiarate come segue:

```
struct {
  int var_1 ;
  char var_2;
} var_struct;

union {
  int var_3 ;
  char var_4;
} var_union;
```

sono memorizzate in questo modo:



Si noti che i due campi della variabile `var_union` sono sovrapposti tra loro, e può essere utilizzato solo un campo alla volta. Inoltre, non c'è modo di sapere se la `var_union` contiene, in un dato istante, un intero oppure un carattere. L'uso più tipico delle `union` prevede quindi che sia presente un campo discriminatore:

```
struct {
  int tipo;  /* discriminatore */
  union {
     int campo1;
     char campo2;
     long campo3;
  }
};
```

In questo modo, il programmatore può memorizzare nel campo `tipo` (sempre presente) un codice numerico che indichi il contenuto attuale della `union` (per esempio: 1 se l'ultimo valore memorizzato è un `int`, 2 per un `char`, ecc.).

2.3 Tipi costanti

Il qualificatore `const` indica che il valore di una variabile non è modificabile. Ad esempio, la dichiarazione:

```
const int byte = 8;
```

definisce una variabile `byte` di tipo intero costante, la quale non può essere modificata nel resto del programma. Il qualificatore `const` può essere applicato alla dichiarazione di qualsiasi variabile, ottenendo una variabile di tipo costante. Nell'esempio precedente, si definisce una variabile di tipo *intero costante*, e non una costante di tipo intero. Questa differenza sarà chiara nel caso di puntatori di tipo costante (vedi sezione 5.6). Si noti che le variabili di tipo costante non possono essere utilizzate per specificare lunghezze di array. Ad esempio, il seguente frammento produrrà un errore in fase di compilazione.

```
#include <stdio.h>
const int tre = 3;
int a[tre];             /* sbagliato!! */
```

La soluzione corretta consiste nell'utilizzare una macro:

```
#include <stdio.h>
#define TRE 3
int a[TRE];
```

Oltre ai tipi costanti, possiamo definire delle costanti enumerative, come:

```
enum settimana {LUN, MAR, MER, GIO, VEN, SAB, DOM};
```

Un'enumerazione associa ad una lista di nomi dei valori interi, a partire da 0. La dichiarazione appena vista associa al nome `LUN` il valore 0, a `MAR` il valore 1, etc. È possibile alterare la sequenza dei valori associati, assegnando esplicitamente alle variabili dei valori interi. Ad esempio, con la dichiarazione:

```
typedef enum {UNO=1, DUE, DIECI=10, UNDICI} tipo_numeri;
tipo_numeri i;
```

viene dichiarata la variabile `i` di tipo `tipo_numeri`, la quale può assumere i valori `UNO, DUE, DIECI, UNDICI`. Dopo l'assegnamento `i = UNDICI;` la variabile `i` ha valore 11.

2.4 Stringhe

Il linguaggio C non mette a disposizione nessun tipo predefinito per trattare oggetti di tipo stringa. Le stringhe sono quindi realizzate come vettori di caratteri. La fine di una stringa viene delimitata dal carattere `\0` (il carattere il cui codice ASCII è zero).

Ad esempio, la dichiarazione:

```
char stringa[6] = mango;
```

definisce la variabile `stringa` di 6 caratteri (l'ultimo carattere contiene il delimitatore `\0`). Nella dichiarazione di stringhe inizializzate, è possibile omettere il numero di caratteri. Il compilatore calcolerà automaticamente l'ampiezza dell'array per contenere la stringa. In generale, una stringa è definita nel seguente modo:

```
char nome-variabile[] = valore-iniziale-della-stringa;
```

Quindi, dichiareremo semplicemente:

```
char stringa[] = mango;
```

Essendo una stringa un array, la dichiarazione precedente è equivalente a:

```
char stringa[] = {'m','a','n','g','o','\0'};
```

Il tipo stringa (realizzato come un array di caratteri) è fondamentalmente diverso dal tipo carattere. Ad esempio, il carattere `'a'` occupa 1 byte[2] (notate l'uso degli apici singoli!). Invece, la stringa `a` (notate i doppi apici!) occupa sempre 2 bytes: il carattere `'a'` seguito dal carattere `'\0'` che delimita la fine della stringa. Si noti che, diversamente da Java, il linguaggio C utilizza un set di caratteri ASCII a 8 bit, e quindi rappresenta i caratteri con 1 byte. Quindi, le dichiarazioni:

```
char stringa[] = "m";
char carattere = 'm';
```

definiscono due variabili di tipo diverso, che occupano, rispettivamente, 2 bytes e 1 byte.

2.5 Operatori

Gli operatori aritmetici sono `+,-,*,/` e `%` che realizza il modulo (resto della divisione). Gli operatori relazionali sono `>, >=, <, <=, ==` e `!=`, mentre gli operatori logici sono `&&` (che realizza l'AND) e `||` (che realizza l'OR) e `!` (che indica il NOT). Le espressioni ottenute da operatori relazionali e logici valgono 1 se sono vere, 0 se sono false. Si noti che i valori di verità sono rappresentati con numeri interi, in mancanza di un apposito tipo booleano. Un valore 0 indica *falso*, mentre un valore diverso da 0 indica *vero*. Spesso, si usa 1 o -1 come valore tipico per *vero*. Ad esempio, dopo l'assegnamento:

```
int i = (j > 2);
```

la variabile `i` ha valore `1` se `j > 2`, `0` altrimenti. Gli operatori per la manipolazione di bit sono:

`&`	AND bit a bit
`|`	OR inclusivo bit a bit
`^`	OR esclusivo bit a bit
`<<`	shift a sinistra
`>>`	shift a destra
`~`	complemento a uno

[2]Il C utilizza la rappresentazione ASCII, la quale richiede solo 1 byte per carattere. In Java viene utilizzata la rappresentazione UNICODE, la quale utilizza due o quattro bytes per carattere.

L'operatore `=` realizza l'assegnamento, gli operatori `++` e `--` realizzano l'assegnamento con l'incremento e il decremento della variabile, e possono essere usati prefissi o postfissi. Gli assegnamenti con operatore sono realizzati da `+=`, `-=`, `*=`, `/=`, `\%=`, `&=`, `|=`, `^=`, `<<=` e `>>=`. Un assegnamento con operatore `x` *`op`*`= expr` è equivalente all'assegnamento `x = x` *`op`* `(expr)` (eccetto che per il fatto che `x` viene valutato una sola volta nella forma concisa, ma due volte in quella espansa). In C, gli assegnamenti (sia semplici che con operatori) sono considerati espressioni, e restituiscono sempre il valore calcolato a destra del segno uguale. Ad esempio, l'assegnamento `++i` restituisce come valore il successore di `i`, oltre ad incrementare la variabile `i`. Inoltre abbiamo già visto l'operatore di *casting* (ad esempio `(int)` realizza la conversione ad intero), l'operatore `sizeof` che restituisce l'occupazione in bytes di un tipo o di una variabile, l'operatore '.' per accedere ai campi delle `struct` (ad esempio `lps.aula`), e l'accesso agli elementi degli array con l'operatore `[]` (come `frase[1][3]`). L'operatore ',' infine permette di combinare una sequenza di espressioni in un'unica espressione. Ad esempio:

```
b++, c*2
```

è un'unica espressione che prima incrementa la variabile `b`, e poi calcola il valore `c*2` che viene restituito come risultato. In generale, un'espressione del tipo:

```
expr1, expr2, ..., exprn
```

viene valutata da sinistra a destra, e poi viene restituito l'ultimo valore calcolato. Ad esempio, al termine del seguente frammento di codice:

```
int b, c=0;
b = (c++, c*2);
```

la variabile `b` ha valore 2.

Esercizio. Calcolare il valore della variabile `b` al termine del codice seguente:

```
int b, c=0;
b = c++, c*2;
```

(Suggerimento: l'assegnamento ha priorità maggiore dell'operatore ',') □

Nella sezione 5.1 vedremo gli operatori relativi all'uso dei puntatori `*, &` e `->`. Si noti che, diversamente da Java, l'operatore `+` non può essere applicato a variabili di tipo "array di caratteri", e non realizza la concatenazione di stringhe. Per manipolare le stringhe, utilizzeremo delle funzioni di libreria (vedi sezione 6.3). In compenso, `+` può essere usato per operate su singoli caratteri, sfruttando l'equivalenza fra caratteri e interi. Per esempio, il frammento `char c; c='A'+2` dichiara una variabile di tipo carattere `c`, e le assegna il valore `'C'` (o, equivalentemente, 67 in decimale).

2.6 Precedenze degli operatori

Le regole di priorità e di associatività degli operatori determinano come viene valutata un'espressione. Riportiamo nel seguito le precedenze dei vari operatori, iniziando da quelli di precedenza maggiore, e mettendo in una stessa linea operatori con la stessa precedenza.

`()` `[]` `.` `->` `++`(postfisso) `--`(postfisso)
`++`(prefisso) `--`(prefisso) `!` `~` `sizeof` `+`(unario) `-`(unario) `&` `*`(deref.)
`*` `/` `%`
`+` `-`
`<<` `>>`
`<` `<=` `>` `>=`
`==` `!=`
`&`
`^`
`|`
`&&`
`||`
`=` `+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `^=` `|=`
`,`

Tutti gli operatori associano da sinistra verso destra, ad eccezione della seconda e penultima riga. Ad esempio, nell'espressione:

```
c = f(x) == 0
```

viene prima eseguito il confronto e poi l'assegnamento, cioè viene valutata come:

```
c = (f(x) == 0)
```

L'utilizzo delle parentesi, anche quando non strettamente necessarie, è caldamente consigliato, per evitare errori semantici.

Capitolo 3

Comandi e strutture di controllo

[Kelley, Cap. 4]

Abbiamo già visto il comando `=` che realizza l'assegnamento, ed alcuni operatori contratti (come `+=`) che realizzano assegnamenti composti con operazioni. Ogni comando termina con un ';' che indica la fine del comando. Si noti che ';' è un terminatore di comandi, e non un separatore. Quindi tutti i comandi devono terminare con un punto e virgola, anche se non sono seguiti da altri comandi. Le parentesi graffe vengono utilizzate per raggruppare insiemi di comandi e formare un blocco, come {`x=0; j++; printf(...)`}. Da un punto di vista sintattico, un blocco è equivalente ad un singolo comando, e quindi può essere utilizzando ovunque al posto di un comando. Si noti che, dopo la parentesi di chiusura del blocco non va posto il ';'. La sintassi generale di un blocco è:

```
{
    dichiarazioni di variabili locali al blocco
    comando_1
    ⋮
    comando_n
}
```

Si noti che la dichiarazione di variabili locali è ammessa solo all'inizio del blocco. Vediamo le strutture di controllo decisionali `if, switch` ed iterative `while, do, for`.

La sintassi del comando `if` è:

```
if (espressione)
    comando
else
    comando
```

dove la parte relativa al ramo `else` è opzionale. Il costrutto `if` valuta l'espressione, e se questa è diversa da 0, viene eseguito il primo comando. Se l'espressione è uguale a 0 ed

esiste un ramo `else`, viene eseguito il comando del ramo `else`. Quindi, il costrutto `if` è equivalente a:

```
if (espressione != 0)
    comando
else
    comando
```

Per raggruppare più scelte decisionali si utilizza il costrutto `switch`, che ha la seguente sintassi:

```
switch (espressione)
{
    case espressione-costante :  comando
    ⋮
    case espressione-costante :  comando
    default :  comando
}
```

L'ultimo caso, etichettato con `default`, è opzionale e può essere omesso. Il costrutto `switch` confronta **sequenzialmente** l'espressione iniziale con le etichette dei vari casi (che devono essere delle espressioni costanti). Quando viene raggiunta un'etichetta uguale all'espressione, vengono eseguiti sequenzialmente tutti i comandi che seguono. Ad esempio, il seguente frammento di programma:

```
int i=2;
switch (i)
  {
  case  1 : printf("uno \n");
  case  2 : printf("due \n");
  case  3 : printf("tre \n");
  default : printf("default \n");
  }
```

stampa il risultato:

```
due
tre
default
```

Per interrompere l'esecuzione sequenziale dei comandi, si utilizza il comando `break`, che permette di uscire immediatamente dal costrutto switch. Il seguente frammento:

```
int i=2;
switch (i)
  {
  case  1 : printf("uno \n"); break;
  case  2 : printf("due \n"); break;
  case  3 : printf("tre \n"); break;
  default : printf("default \n");
  }
```

stampa solamente la riga:

```
due
```

Passiamo ora ai costrutti iterativi. La sintassi del ciclo `while` è:

```
while (espressione)
    comando
```

dove *comando* può essere, al solito, un singolo comando o un blocco.

Simile al costrutto `while` è il costrutto `do-while`, il quale prima esegue il comando e poi valuta l'espressione. Se l'espressione è vera, si esegue nuovamente il comando, e così via. La sintassi del `do-while` è:

```
do
    comando
while (espressione);
```

Si noti che il `do-while` termina con un ';'. Tra i costrutti visti fino ad ora, è l'unico costrutto che necessita di un terminatore. Questo perché tutti gli altri costrutti terminano con un comando. Infine, il ciclo `for` ha la seguente sintassi:

```
for (espressione1 ; espressione2 ; espressione3 )
    comando
```

Poiché gli assegnamenti ritornano il valore dell'espressione assegnata, è molto frequente utilizzare degli assegnamenti al posto delle espressioni, come ad esempio in:

```
for (i=0; i <= 9; i++)
```

dove i due assegnamenti `i=0` e `i++` sono utilizzati come espressioni. Si noti che non si possono dichiarare variabili nelle espressioni. Quindi, non è possibile scrivere condizioni di inizializzazione di un ciclo del tipo:

```
for (int i=0; i <= 9; i++)          /* sbagliato!!!  */
```

Tutte le dichiarazioni di variabili devono essere poste all'inizio della funzione o blocco che le

contengono.

Il comando `break` permette di uscire da un ciclo (`for`, `while` o `do`) senza controllare la condizione di terminazione, esattamente come viene utilizzato per il costrutto `switch`. Un comando `break` si riferisce sempre al ciclo più interno che lo contiene. Il comando `continue` permette invece di saltare all'iterazione successiva di un cilo `for`, `while` o `do`, eseguendo immediatamente il controllo di terminazione. Nel ciclo `for` il comando `continue` permette di passare direttamente all'incremento.

Si noti che i vari costrutti per il controllo del flusso sono in generale intercambiabili, in opportune combinazioni. Per esempio, in ognuna delle coppie riportate di seguito, i due frammenti di codice sono equivalenti:

`for (inizio; condizione; passo)`
`    comando;`
	`inizio;`
	`while (condizione) {`
	`    comando;`
	`    passo;`
	`}`

`while (condizione) {`
`    comando;`
`}`
	`if (condizione)`
	`    do {`
	`        comando;`
	`    } while (condizione);`

`if (condizione)`
`    comando;`
	`while (condizione) {`
	`    comando;`
	`    break;`
	`}`

`switch (expr) {`
`  case v1: comando1; break;`
`  case v2: comando2; break;`
`  case v3: comando3; break;`
`  default: comandod;`
`}`
	`tmp=expr;`
	`if (tmp==v1)`
	`    comando1;`
	`else if (tmp==v2)`
	`    comando2;`
	`else if (tmp==v3)`
	`    comando3;`
	`else`
	`    comandod;`

Esercizi

1 Crivello di Eratostene: si costruisca un array di interi dichiarato con `int tabella[101]`, il quale abbia la seguente proprietà, per ogni $2 \leq i \leq 100$:

- se i è primo, allora $tabella[i] = 1$
- se i non è primo, allora $tabella[i] = 0$

Infine si stampino i numeri primi trovati.

2 Si definisca un'array di 20 elementi dichiarati come segue:

```
typedef struct {
    int chiave;
    char valore[20];
} elemento;
```

Dopo aver riempito l'array con valori a scelta, si stampi il secondo campo della tabella (il campo valore) ordinato secondo il campo chiave.

3 Si scriva un costrutto `for` equivalente a `while (!(a>=5)) a++`
.

4 Nel mostrare l'equivalenza fra un costrutto `switch` e una serie di `if`, abbiamo introdotto una variabile temporanea `tmp`. Perché? È possibile farne a meno? Sotto quali condizioni?

Capitolo 4

Funzioni

[Kelley, Cap. 5]

4.1 Dichiarazione e definizione

Il linguaggio C mette a disposizione due differenti meccanismi per dichiarare e definire le funzioni. Poiché in C non esistono i concetti di procedura o metodo, queste nozioni sono parzialmente rimpiazzate dal concetto di funzione, che assume quindi un ruolo fondamentale. La dichiarazione di una funzione descrive il prototipo della funzione, cioè il nome della funzione, il tipo restituito dalla funzione e i tipi di tutti gli argomenti. Ad esempio:

```
int somma(int, int);
```

dichiara una funzione di nome `somma`, che prende due interi e restituisce un intero. Si noti che nella dichiarazione di funzione non vengono assegnate variabili agli argomenti (solo i tipi) e non viene definito il corpo della funzione, ma solo il suo prototipo.

La sintassi generale per dichiarare una funzione è:

```
tipo-funzione nome-funzione (tipo-1, ..., tipo-n);
```

dove *tipo-funzione* può essere un tipo qualsiasi (anche definito con un `typedef`) ad eccezione di array e funzioni. È possibile invece restituire un puntatore ad array o funzione. Le funzioni che non ritornano nessun valore, vengono dichiarate di tipo `void`:

```
void nome-funzione (tipo-1, ..., tipo-n);
```

Una funzione di tipo `void` può quindi essere vista come una procedura. Analogamente, le funzioni che non prendono nessun argomento, vengono dichiarate con argomento `void`:

```
tipo-funzione nome-funzione (void);
```

Per definire una funzione, occorre invece descrivere il corpo della funzione, che ne individua univocamente il comportamento. Ad esempio:

```
int somma(int x, int y)
{
  return(x+y);
}
```

definisce una funzione che restituisce la somma dei suoi argomenti. Ogni definizione di funzione ha la forma:

```
tipo-funzione nome-funzione (tipo-1 var-1, ..., tipo-n var-n)
{
   variabili locali
   corpo della funzione
}
```

Si noti che le variabili locali alla funzione sono dichiarate all'inizio (come nel caso dei blocchi) e non possono essere intercalate ai comandi che costituiscono il corpo della funzione. Inoltre, le variabili dichiarate localmente alla definizione di una funzione sono accessibili solo nel corpo della funzione.

Il comando `return(`*espressione*`)` permette di terminare la funzione, restituendo un valore al chiamante. In ogni funzione con tipo diverso da `void` deve apparire il comando `return(`*espressione*`)`, dove il tipo dell'espressione restituita deve essere quello della funzione. Nelle funzioni di tipo `void` il comando `return` può essere utilizzato per uscire dalla funzione, oppure può essere omesso.

Le funzioni non posso essere annidate, ma qualsiasi funzione può richiamare (anche ricorsivamente) ogni altra funzione. Inoltre, ogni funzione deve essere dichiarata prima del suo utilizzo. Si noti che la dichiarazione di una funzione può essere contestuale alla sua definizione. Questo significa che, in mancanza di una dichiarazione esplicita, la sola definizione di funzione è sintatticamente corretta e il prototipo della funzione viene dedotto dalla definizione stessa. Ogni riferimento ad una funzione deve sempre essere preceduto dalla dichiarazione della funzione (oppure, in mancanza di questa, dalla sua definizione). La struttura di un generico programma C con dichiarazioni di funzioni è del tipo:

```
#direttive al preprocessore
dichiarazioni di variabili globali
dichiarazioni di funzioni
definizioni di funzioni
```

dove, tra le definizioni di funzioni, deve comparire la funzione `main`. Si noti che, benché le direttive al preprocessore possano apparire ovunque all'interno del file sorgente, è uso raggruppare le `#include` e `#define` all'inizio del file.

Nel linguaggio C, tutti i parametri sono passati per valore. Il passaggio per riferimento si ottiene utilizzando i puntatori (vedi sezione 5.4).

Vediamo un esempio di un semplice programma che utilizza la funzione somma. In particolare, si noti che la chiamata della funzione `somma` è preceduta dalla sua dichiarazione, ma non dalla sua definizione, che appare solamente alla fine del programma.

```
#include <stdio.h>
int somma(int, int);
int main()
{
  printf("%d \n", somma(40,2));
  return(0);
}
int somma(int x, int y)
{
  return(x+y);
}
```

La funzione `printf` usata in questo programma è una funzione di libreria, la cui dichiarazione è data in `stdio.h` (che qui viene incluso), e la cui definizione è contenuta nella libreria standard del sistema; pertanto, il programma può usarla liberamente senza che sia necessario darne una dichiarazione nè una definizione (in particolare, `printf` consente di stampare sullo schermo output formattato, e sarà discussa in maggiore dettaglio più avanti).

4.2 *Scope* delle variabili

Nel linguaggio C distinguiamo due tipi di variabili: le variabili locali (o *automatiche*), che vengono dichiarate all'interno di una funzione (compresa la funzione `main`), e le variabili globali, dichiarate fuori da tutte le funzioni.

Le variabili locali hanno due caratteristiche:

1. possono essere utilizzate solamente all'interno della funzione dove sono state dichiarate;
2. perdono il loro valore tra due chiamate successive della stessa funzione (a meno che non vengano dichiarate `static`, come si vedrà più avanti).

Al contrario, le variabili globali sono accessibili da qualsiasi funzione, e mantengono sempre il loro valore (indipendentemente da quale funzione si sta eseguendo).

Le variabili locali ad una funzione possono essere dichiarate `static`, al fine di preservare il loro valore tra due chiamate successive della stessa funzione. Le variabili locali dichiarate `static` restano sempre locali, e non sono quindi visibili all'esterno della funzione dove sono

dichiarate. Il programma che segue mostra un esempio di utilizzo di variabili statiche. Nell'esempio si utilizza la funzione di libreria `atoi` che converte (la parte iniziale di) una stringa in un intero.

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

int accu(int);

int main(int argc, char * argv[])
{
  int i;
  for(i=1; i<argc; i++)
    printf("%d \n",accu(atoi(argv[i])));
  return(0);
}

int accu(int i)
{
  static int somma = 0;
  somma += i;
  return(somma);
}
```

Capitolo 5

Puntatori

[Kelley, Cap. 6]

5.1 Dichiarazione di puntatori

Un puntatore è una variabile che contiene l'indirizzo di memoria di un'altra variabile. Con la dichiarazione:

```
int * puntatore;
```

viene dichiarata la variabile `puntatore` che conterrà l'indirizzo di memoria di un intero. Per conoscere l'indirizzo di una variabile, si utilizza l'operatore `&`. Ad esempio, se `i` è una variabile di tipo intero, allora `&i` è l'indirizzo di memoria della variabile `i`. Assegnando tale indirizzo ad una variabile di tipo puntatore ad intero, otteniamo l'effetto di poter accedere indirettamente alla variabile `i`. Per dereferenziare il puntatore (cioè accedere alla locazione di memoria puntata) si utilizza l'operatore `*`. Ad esempio, il seguente frammento di programma:

```
int i = 42;
int * puntatore = &i;
*puntatore = 1;
```

assegna alla variabile `puntatore` l'indirizzo di `i`, e poi modifica indirettamente il contenuto della variabile `i`. Al termine di questo frammento, il valore di `i` è `1`, sebbene `i` non sia stata modificata direttamente.

Supponiamo di definire un puntatore ad una struttura:

```
struct laboratorio {
  char orario[20];
  char aula;
  int studenti;
} * puntatore;
```

Per accedere ad un elemento dello struttura dobbiamo prima dereferenziare il puntatore e poi selezionare il campo. Ad esempio `(*puntatore).aula` accede al secondo campo. Lo stesso risultato si ottiene utilizzando l'operatore `->` infisso tra il nome del puntatore e il campo: `puntatore->aula` è equivalente all'espressione `(*puntatore).aula`.

5.2 Aritmetica dei puntatori

Nella dichiarazione di un puntatore, occorre dichiarare il tipo dell'oggetto puntato. Ciò permette di definire una vera e propria "aritmetica dei puntatori". Ad esempio, il seguente frammento:

```
int tabella[4] = {10,20,30,40};
int * puntatore = &tabella[0];
```

definisce un array di 4 interi ed un puntatore al primo elemento dell'array. Se incrementiamo il puntatore:

```
puntatore = puntatore + 1;
```

il risultato **non** è di incrementare di `1` il valore del puntatore, bensì di un numero di bytes equivalenti all'ampiezza di un intero, cioè `4`. In questo modo, la variabile `puntatore` punterà al secondo elemento dell'array (questo perché gli elementi di un'array sono sempre memorizzati consecutivamente). In generale, se dichiariamo un puntatore:

```
tipo-puntato * puntatore;
```

ogni incremento del puntatore `puntatore = puntatore + n` equivale a sommare `n` volte il valore restituito da `sizeof(`*`tipo-puntato`*`)`. Si noti che la somma di un puntatore con un intero è sempre di tipo puntatore, e quindi l'assegnamento effettuato è coerente con il tipo del puntatore. Analogamente, è possibile sottrarre un intero da un puntatore. Intuitivamente, questo equivale a percorre "all'indietro" gli elementi di un array. Come nel caso precedente, `puntatore = puntatore - n` equivale a sottrarre `n` volte il valore `sizeof(`*`tipo-puntatore`*`)`. Costituisce un'eccezione il tipo speciale `void`. La dichiarazione `void * puntatore` identifica un puntatore generico, il quale, prima di essere utilizzato, deve essere sottoposto ad un *casting*.

Puntatori della stesso tipo si possono anche sottrarre. La sottrazione di due puntatori conta

il numero di elementi che separano i due puntatori. Ad esempio, dopo le dichiarazioni:

```
int tabella[4] = {10,20,30,40};
int * puntatore1 = &tabella[0];
int * puntatore2 = &tabella[3];
int i = puntatore2 - puntatore1;
```

la variabile `i` ha valore 3. Ciò è coerente con il fatto che se sommiamo a `puntatore1` il valore 3, otteniamo proprio `puntatore2`. In generale, dati due puntatori:

```
tipo-puntato *p1, *p2;
```

l'espressione `p1 - p2` equivale a calcolare la differenza tra il valore di `p1` e `p2`, divisa per `sizeof(`*tipo-puntato*`)`. Inoltre, tale differenza avrà sempre tipo intero.

Abbiamo già visto come utilizzando i puntatori possiamo scorrere gli elementi di un array. Questo è reso possibile dal fatto che gli elementi di un array sono memorizzati consecutivamente. Definiamo un array di interi ed un puntatore ad interi:

```
int vettore[5];
int * p;
```

ed assegniamo al puntatore l'indirizzo del primo elemento dell'array con:

```
p = &vettore[0];
```

In questo modo possiamo scorrere tutti gli elementi dell'array utilizzando il puntatore: `p+1` è l'indirizzo del secondo elemento, cioè `&vettore[1]`, e quindi `*(p+1)` si riferisce al valore di `vettore[1]`. In generale, `p+i` è l'indirizzo dell'elemento in posizione `i`, cioè `&vettore[i]`, e `*(p+i)` è il valore `vettore[i]`. Per definizione, il valore denotato dal nome del vettore è l'indirizzo del suo primo elemento. Quindi l'assegnamento:

```
p = &vettore[0];
```

è del tutto equivalente a scrivere:

```
p = vettore;
```

Inoltre, possiamo applicare l'operatore di selezione `[]` anche ai puntatori: `p[0]` è il valore del primo elemento dell'array ed è equivalente a scrivere `*p`. In generale `p[i]` è il valore dell'elemento in posizione `i`, ed è quindi equivalente a scrivere `*(p+i)`. Analogamente, possiamo applicare l'operatore di dereferenziazione ad un array (ricordiamoci che, per definizione, un array è l'indirizzo del primo elemento). Quindi, `vettore+1` è l'indirizzo del secondo elemento, e `vettore+i` è l'indirizzo dell'elemento in posizione `i`, cioè `&vettore[i]`. In pratica, per manipolare gli elementi di un array, possiamo arbitrariamente utilizzare gli operatori degli array o dei puntatori, ottenendo gli stessi effetti. In particolare, `vettore[i],`

`*(vettore+i)`, `*(p+i)` e `p[i]` sono tutte espressioni equivalenti per riferirsi all'elemento in posizione `i` dell'array.

Infine, puntatori che puntano ad elementi dello stesso array, possono essere confrontati utilizzando i comuni operatori relazionali `>`, `>=`, `<`, `<=`, `==` e `!=`. Il risultato è di confrontare le posizioni degli elementi ai quali puntano. Ad esempio, se dichiariamo:

```
int * p1 = vettore;
int * p2 = &vettore[3];
```

allora l'espressione `p1 < p2` ha valore vero, perché `p1` punta ad un elemento che precede l'elemento puntato da `p2`. Al contrario, `p1 >= p2` ha valore falso. Quindi, gli operatori relazionali applicati agli array permettono di confrontare le posizioni relative degli elementi puntati. Un'eccezione è il confronto di un puntatore con lo zero. Infatti, lo zero è un numero convenzionale per indicare che il puntatore non sta puntando a nessuna locazione di memoria. È inoltre possibile assegnare la costante zero ad un puntatore. Per evidenziare il fatto che lo zero non è utilizzato come un intero, ma semplicemente per segnalare un puntatore che non punta a nessun dato, al posto del simbolo `0` si utilizza la costante `NULL` (che è definita in `<stdio.h>` e vale esattamente zero). Quindi scriveremo `puntatore == NULL` oppure `puntatore = NULL`.

Notare che, sebbene un array per definizione contiene l'indirizzo del primo elemento, questo non significa che un array è di fatto un puntatore. Ad esempio, l'assegnamento di un puntatore ad un array, come `vettore = p` è illecito, come pure è illecito incrementare un vettore con `vettore++`. Infatti, una differenza fondamentale tra array e vettori risiede nel fatto che, sebbene un array sia fondamentalmente un indirizzo, non è possibile associare ad un array un altro indirizzo. Tutto quello che si può fare è modificare gli elementi di un array, ma non l'indirizzo dove è memorizzato l'array. Al contrario, possiamo cambiare l'indirizzo al quale punta un puntatore, e farlo puntare ad un altro indirizzo di memoria (compatibilmente con il suo tipo), senza modificare il suo contenuto, che resta invariato, sebbene non più accessibile tramite il puntatore. Questo spiega perché `vettore++` sia concettualmente sbagliato: infatti, non stiamo modificando il vettore, ma vogliamo che il nome `vettore` punti ad un'altra locazione, e questo non è ammesso. Ovviamente, l'assegnamento di un array ad un puntatore e l'incremento di un puntatore sono invece operazioni perfettamente ammissibili. Inoltre, è lecito utilizzare i puntatori per selezionare una parte di un array. Ad esempio, se `a` è un array di `n` elementi, allora `&a[1]` è un array di `n-1` elementi.

Per concludere, possiamo sintetizzare le differenze tra puntatori e array nel seguente modo: un array è un puntatore costante, con il quale possiamo modificare i valori ma non l'indirizzo dell'array.

5.3 Esempi di dichiarazioni

Le dichiarazioni di puntatori possono sembrare molto complesse, specialmente quando sono combinati con gli array e le funzioni. In questo caso occorre prestare particolare attenzione alla precedenza degli operatori. Ad esempio, la dichiarazione:

```
int * x [10];
```

definisce un array di 10 elementi, ciasuno dei quali è un puntatore ad intero. La dichiarazione è equivalente a `(int *) x [10]`. La tabella che segue mostra alcuni tipi di dichiarazioni.

`int *a[10]`	array di 10 puntatori a `int`
`int (*p)[10]`	puntatore ad un array di 10 `int`
`int *f (int)`	funzione che prende un `int` e restituisce un puntatore a `int`
`int (*p) (int)`	puntatore a funzione che prende e restituisce un `int`
`int *a[] (int)`	array di funzioni che prendono un `int` e restituiscono un puntatore a `int`
`int (*a[]) (int)`	array di puntatori a funzioni che prendono e restituiscono un `int`
`int (*p)[] (int)`	puntatore ad un array di funzioni che prendono e restituiscono un `int`

5.4 Passaggio di parametri per riferimento

L'utilizzo di puntatori come parametri di una funzione realizza il passaggio dei parametri per riferimento. Il programma che segue:

```
#include <stdio.h>
void scambia(int *x, int *y)
{
  int temp = *x;
  *x = *y;
  *y = temp;
  return;            // puo' essere omesso
}
int main()
{
  int i=1, j=3;
  scambia(&i,&j);
  printf("i=%d, j=%d \n", i,j);
  return(0);
}
```

definisce una funzione che prende due puntatori ad intero e ne scambia i valori. Di conseguenza, il programma stampa il risultato `i=3, j=1`.

Una conseguenza del fatto che un array, per definizione, è un indirizzo, comporta che gli array siano **sempre** passati per riferimento. Se dichiariamo una funzione `f` che assegna il valore `42` al primo elemento di un array, tale modifica avrà effetto anche sull'array passato come argomento alla funzione, realizzando il passaggio di parametri per riferimento. Il seguente programma:

```
#include <stdio.h>
void f(int x[])
{
  x[0]=42;
  return;          // puo' essere omesso
}
int main()
{
  int i[2]={2,5};
  f(i);
  printf("i[0] = %d \n", i[0]);
  return(0);
}
```

stampa come risultato `i[0] = 42`. Infatti, i parametri formali di tipo array sono visti come puntatori e dichiarare un array o un puntatore è del tutto equivalente. Quindi, la funzione `f` può essere dichiarata come `void f(int *x)`. Si noti che questa equivalenza vale **solo** nelle dichiarazioni dei parametri formali.

5.5 Allocazione della memoria: `malloc`

La dichiarazione di un puntatore con `int * p` alloca l'area di memoria per contenere il puntatore stesso (cioè l'indirizzo al quale si trova l'intero puntato). Non viene invece allocata la memoria per contenere l'intero. Il seguente programma:

```
#include <stdio.h>
int * p;
int main()
{
  *p=1;                      /* sbagliato! */
  return(0);
}
```

definisce un puntatore ad intero `p`, ed assegna un valore intero alla locazione puntata da `p`. Il compilatore non genera nessun errore, ma l'esecuzione del programma termina (molto probabilmente) con un `Segmentation fault`, poiché non era stata allocata la memoria per

contenere il numero intero, ma solo per contenere il puntatore. Si verifica quindi un accesso illegale perché l'area di memoria riservata al puntatore non è inizializzata e contiene un puntatore casuale che, molto probabilmente, sarà fuori dallo spazio di indirizzamento logico del processo. La funzione di libreria `malloc` permette l'allocazione dinamica di blocchi di memoria. Per utilizzarla occorre includere l'header `<stdlib.h>`. La funzione `malloc` prende come argomento il numero di bytes da allocare e restituisce un puntatore alla memoria allocata. Se non riesce ad allocare la memoria, viene restituito `NULL`. Il prototipo della funzione `malloc` è:

```
void *malloc(size_t n);
```

dove `size_t` è una `typedef` dichiarata in `stdlib.h` come equivalente al tipo `unsigned int`. Dichiariamo un puntatore ad un array di 10 interi:

```
int *p[10];
```

Possiamo utilizzare la funzione `sizeof` per calcolare l'esatto ammontare della memoria necessaria:

```
 p = malloc(sizeof(int *[10]));
```

In questo caso vengono allocati esattamente 40 bytes. Dopo aver allocato la memoria, occorre sempre controllare che non si siano verificati errori:

```
if (p == NULL) {gestione dell'errore....}
```

Per liberare un'area di memoria precedentemente allocata con `malloc` si utilizza la funzione di libreria `free`, la quale prende come argomento il puntatore. Il prototipo della funzione `free` è:

```
void free(void *ptr);
```

Ad esempio `free(p)` libera i 40 bytes allocati precedentemente.

Si noti che, nel caso in cui un puntatore a carattere venga inizializzato contestualmente alla dichiarazione, allora viene allocata memoria per contenere il suo argomento. Ad esempio, con la dichiarazione:

```
char * i= abc;
```

vengono allocati 4 bytes per contenere la stringa `abc`. Notate che la stringa `abc` è una stringa costante e non è modificabile. Quindi, non è possibile effettuare degli assegnamenti del tipo `i[0] = 'z'`. Al contrario, una stringa dichiarata come array `char i[]= abc` può essere modificata. Infatti, quest'ultima dichiarazione è solo una abbreviazione di:

```
char i[]= {'a','b','c','\0'};
```

la quale inizializza l'array con una stringa non costante.

5.6 Puntatori costanti

Il seguente frammento di programma:

```
#include <stdio.h>
int main()
{
  const char * i = "Hello";
  i = "Ciao";
  printf("%s \n", i);
  return(0);
}
```

viene compilato senza errori e stampa `Ciao`. Questo perché una dichiarazione del tipo: `const char * i` dichiara un puntatore ad un oggetto di tipo `const char`, cioè ad una stringa costante. Questo **non** significa che `i` è una costante di tipo puntatore a stringa! Quindi, la dichiarazione sopra va letta: `i` è un puntatore ad una "stringa costante" (e non: `i` è una costante che punta ad una stringa). Per dichiarare un puntatore costante (cioè non modificabile), dobbiamo utilizzare la dichiarazione:

```
char * const i = Hello;
```

che modifica il tipo della variabile `i`.

5.7 Argomenti dalla linea di comando

Nella funzione `main` è possibile utilizzare degli argomenti, per comunicare con il sistema operativo. Nel caso più semplice vengono utilizzati due argomenti, chiamati storicamente `argc` e `argv`:

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

Il parametro `argc` contiene il numero di argomenti nella linea di comando (incluso il nome del programma). Il parametro `argv` è un array che contiene gli argomenti veri e propri. Ad esempio, se eseguiamo il programma `mioprogramma` e lo richiamiamo con:

```
mioprogramma stringa passata al programma
```

allora `argc` e `argv` conterranno i seguenti valori:

```
argc = 5
argv[0] = mioprogramma\0
argv[1] = stringa\0
argv[2] = passata\0
argv[3] = al\0
argv[4] = programma\0
argv[5] = \0
```

Infine, nelle piattaforme dove sono disponibili le variabili d'ambiente (Unix, Linux, MS-DOS) il terzo argomento di `main` viene utilizzato per accedere a queste variabili. A tal fine utilizziamo la definizione:

```
int main(int argc, char *argv[], char * env[])
```

dove il terzo argomento è un array di stringhe che contiene i valori delle variabili d'ambiente. L'ultimo elemento dell'array contiene il valore `NULL`. Il seguente programma stampa a video la lista delle variabili d'ambiente.

```
#include <stdio.h>
int main(int argc,char * argv[], char * env[])
{
  int i;
  for(i=0; env[i] != NULL; i++)
    printf("%s \n", env[i]);
  return(0);
}
```

Esercizi

1 Si scriva un programma che prende in input una stringa dalla linea di comando e la stampa al contrario (iniziando dall'ultimo carattere). Non si possono utilizzare funzioni di libreria per trattare le stringhe, inclusa `strlen`.

2 Si consideri la lista definita come segue:

```
typedef struct elemento {
  int valore;
  struct elemento * next;
} * lista;
```

Si scriva un programma che prende in input un numero intero n dalla linea di comando (convertirlo usando la funzione `atoi`) e crea una lista di n elementi di tipo `struct elemento` con valori casuali. Per creare tali valori si utilizzi la macro RAND:

```
#define RAND() 1+(int) (10.0*rand()/(RAND_MAX+1.0));
```

Infine si definiscano due funzioni: stampa, che stampa tutti gli elementi della lista, e stampa_reverse, che stampa la lista al contrario, iniziando dalla coda.

3 Si consideri il tipo albero definito come segue:

```
typedef struct nodo {
    struct nodo * figlio_sx;
    int valore;
    struct nodo * figlio_dx;
} * albero;
```

Si costruisca un albero binario di profondità n (dove n è immesso alla linea di comando) i cui valori siano interi positivi, ordinati come risultano da una visita in profondità dell'albero. Infine, si stampino i valori delle foglie.

Capitolo 6

Funzioni di libreria più comuni

6.1 Input ed output: `getchar` e `putchar`

Per leggere un input da tastiera, utilizziamo la funzione di libreria `getchar`, che ci permette di leggere un carattere alla volta. La funzione non prende nessun argomento e ritorna il codice del carattere letto, di tipo `int`, oppure il valore `EOF` (che vale `-1`) se non viene immesso nessun input. Nel linguaggio C, ogni carattere viene rappresentato con il valore intero corrispondente al suo codice ASCII: ad esempio, `'a'` ha valore 97, `'b'` 98, etc. Il prototipo della funzione è:

```
int getchar(void)
```

Si noti che il tipo del risultato di `getchar` non è semplicemente `char`, come ci si potrebbe aspettare. Infatti, `getchar` deve poter ritornare *qualunque* carattere, *più* un altro valore distinto, `EOF`. È dunque necessario che il tipo del risultato sia più grande di `char` — in questo caso, si usa infatti `int`.

La corrispondente funzione per stampare caratteri a video è `putchar`, la quale prende come argomento un intero e stampa il carattere corrispondente. La funzione restituisce il codice del carattere scritto o `EOF` se si è verificato un errore. Il suo prototipo è:

```
int putchar(int)
```

Entrambe le funzioni `getchar`, `putchar` e la macro `EOF` sono dichiarate in `<stdio.h>`. Ad esempio, il seguente frammento stampa a video i caratteri letti da tastiera (riproducendo in parte il comportamento del comando UNIX `cat` quando eseguito senza parametri):

```
#include <stdio.h>
int main()
{
  int i;
  while ( (i = getchar()) != EOF)
    putchar(i);
  return(0);
}
```

Il comportamento della funzione `getchar()` su Linux è influenzato dal fatto che l'input da tastiera viene bufferizzato. Quindi, per immettere un singolo carattere da tastiera, occorre digitare il carattere seguito da `Ctrl-D` che segnala la fine dell'input. Se si immettono più caratteri in corrispondenza di una singola chiamata `getchar()`, i caratteri avanzati rimangono disponibili per essere utilizzati da una successiva chiamata `getchar()`. Analogamente, il tasto `Enter` può essere utilizzato per segnalare che il buffer di tastiera può essere passato al programma (senza che il canale di input venga chiuso), con la differenza che viene memorizzato anch'esso, e quindi passato alla successiva chiamata `getchar()`.

6.2 Output formattato: `printf`

La funzione di libreria `printf` (definita in `<stdio.h>`) viene utilizzata per stampare a video un output formattato. La funzione `printf` prende come argomenti il formato dell'output, seguito dalla lista degli argomenti da stampare, e restituisce il numero di caratteri stampati.

`int printf(char * formato, arg1,...,argn)`

Il campo *formato* contiene i caratteri ordinari, che non necessitano conversioni, e le specifiche di conversione degli oggetti. Una specifica inizia con il simbolo `%` ed è seguita da un carattere di conversione, che varia a seconda del tipo di dato che si vuole stampare. Ad esempio, se `i` è una variabile intera, allora la seguente chiamata:

```
printf("La variabile i vale %d \n", i);}
```

stamperà la stringa `La variabile i vale` , seguita dal valore di `i`. La specifica `%d` sta ad indicare che si vuole stampare un numero decimale. Le principali specifiche di conversione sono le seguenti.

`%d`	`int`	numero decimale
`%f`	`double`	
`%c`	`int, char`	carattere
`%s`	`* char`	stampa tutti i caratteri puntati fino a '\0'

La funzione `sprintf` si comporta come `printf`, con la differenza che l'output viene memorizzato in una stringa, passata come primo argomento. Il prototipo della funzione è:

```
int sprintf(char * stringa, char * formato, arg1,...,argn)
```

Ad esempio, il seguente frammento di codice:

```
int i = 42;
char c[20];
sprintf(c,"i ha valore %d \n",i);}
```

memorizza nella variabile `c` la stringa `i ha valore 42`.

6.3 Funzioni per manipolare le stringhe

Le funzioni per manipolare le stringhe sono tutte dichiarate nell'header `<string.h>`, che quindi deve essere inclusa nel programma. La funzione `strcpy` realizza la copia di una stringa. Si noti che l'assegnamento di stringhe realizzate come array non è permesso e l'assegnamento di stringhe realizzate come puntatori ha come effetto di copiare solo l'indirizzo del puntatore. La funzione `strcpy` prende due stringhe come argomenti e copia il secondo argomento nel primo (compreso il carattere terminatore). È compito del programmatore assicurarsi che il primo argomento possa contenere la stringa da copiare. Il prototipo della funzione è:

```
char *strcpy(char *destinazione, const char *sorgente);
```

La funzione ritorna un puntatore alla stringa `destinazione`. La funzione `strncpy` permette di specificare il numero di caratteri da copiare, che deve essere fornito come ultimo argomento.

```
char *strncpy(char *destinazione, const char *sorgente, size_t numero);
```

Richiamata con `strncpy(stringa1, stringa2, n)` copia (al più) `n` caratteri. Se la lunghezza di `stringa2` è minore di `n`, allora viene copiata l'intera stringa `stringa2`.

La funzione `strcat` realizza la concatenazione di stringhe. Richiamata con `strcat(s1, s2)` appende la stringa `s2` alla stringa `s1`, sovrascrivendo il carattere terminatore. La funzione `strncat` realizza la concatenazione di (al più) `n` caratteri. I prototipi delle funzioni sono:

```
char *strcat(char *destinazione, const char *sorgente);
char *strncat(char *destinazione, const char *sorgente, size_t numero);
```

La funzione `strlen`, richiamata con `strlen(s1)` restituisce il numero di caratteri nella stringa `s1`, escluso il carattere terminatore. La funzione `strcmp` applicata a due stringhe con

`strcmp(s1,s2)` restituisce zero se le due stringhe sono uguali, un numero diverso da zero altrimenti. `strncmp`, chiamata con `strncmp(s1,s2,n)` confronta solamente i primi `n` caratteri delle stringhe.

La funzione `strsep` permette di individuare all'interno di una stringa dei *token*, cioè delle sottostringhe separatate da un delimitatore fissato. La funzione `strsep`:

```
char *strsep(char **puntatore, const char *delimitatore);
```

prende un puntatore ad una stringa (ovvero, un puntatore a un puntatore a caratteri: ecco perché abbiamo il doppio *) ed una stringa costante (il delimitatore) e restituisce un puntatore al prossimo token. Il puntatore passato come primo argomento viene modificato e fatto puntare al resto della stringa da analizzare. Il programma che segue è un esempio di utilizzo della funzione `strsep` con un delimitatore '`|`'.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
  char *s;
  char *token;
  char * temp = "prima|seconda|terza|quarta";
  s = malloc(strlen(temp)+1);
  strcpy(s,temp);
  while ((token = strsep(&s, "|"))  != NULL)
    printf("%s \n",token);
  return(0);
}
```

Le funzioni per manipolare le stringhe sono descritte nel manuale in linea, alla pagina `string`. Si ricorda che il linguaggio C non effettua controlli sulle dimensioni degli array. Si consiglia quindi di utilizzare le funzioni che operano solo su una porzione fissata di array, come `strncpy`, `strncat` e `strncmp`.

Esercizio. Si scriva un programma che legga il valore della variabile d'ambiente `PATH` e che ne stampi i componenti (che nella variabile sono separati da ':'), una directory per ogni riga.

Esercizio. La funzione di libreria `getenv` consente di leggere il valore di una variabile d'ambiente. Dopo aver letto la pagina di manuale relativa, si modifichi il programma precedente in modo che usi `getenv` anziché il terzo argomento di `main`. Quale versione è più semplice?

Capitolo 7

Ancora sulla compilazione

7.1 Compilazione separata

Abbiamo già visto che l'utilizzo degli headers permette di suddividere un programma C in due (o più) parti, suddividendo i dati globali dal testo del programma. Nel caso in cui il programma C sia formato da più funzioni (oltre alla funzione `main`) è altresì possibile dividere il testo del programma in più files. Ad esempio, il seguente programma:

```
#include <stdio.h>
int somma(int, int);
int main()
{
  printf("%d \n", somma(40,2));
  return(0);
}
int somma(int x, int y)
{
  return(x+y);
}
```

può essere diviso in tre parti:

1. un header `somma.h` che contiene i dati globali:

   ```
   /* file di header somma.h */
   #include <stdio.h>
   int somma(int, int);
   ```

2. un file `mioprogramma.c` che contiene la funzione `main`:

```
/* file principale mioprogramma.c */
#include "somma.h"
int main()
{
  printf("%d \n", somma(40,2));
  return(0);
}
```

3. un file `somma.c` che contiene la definizione della funzione `somma`:

```
/* file somma.c */
#include "somma.h"
int somma(int x, int y)
{
  return(x+y);
}
```

Notare che entrambi i files sorgenti `mioprogramma.c` e `somma.c` includono l'header `somma.h`. Questo perché ogni utilizzo della funzione `somma` (esclusa, al più, la sua definizione) deve essere preceduto dalla dichiarazione della funzione. Per compilare un programma suddiviso in più files, si richiama il compilatore con i nomi di tutti i file sorgenti (**non** gli headers, che vengono inclusi automaticamente):

```
gcc -Wall -g mioprogramma.c somma.c -o mioprogramma
```

È anche possibile compilare separatamente i file sorgenti, e poi effettuare un `link` dei vari files. In questo caso occorre creare i files oggetto corrispondenti ai files sorgenti, richiamando il compilatore con l'opzione `-c`. Ad esempio:

```
gcc -Wall -g -c mioprogramma.c somma.c
```

crea i files oggetto `mioprogramma.o` e `somma.o`. Per effettuare il link dei due file oggetto, è sufficiente richiamare il compilatore:

```
gcc mioprogramma.o somma.o -o mioprogramma
```

il quale produce l'eseguibile `mioprogramma`. Si noti che gli headers vengono inclusi dal preprocessore prima di iniziare la compilazione vera e propria (sia nel caso si produca un file eseguibile, che un file oggetto). Al contrario, il linker viene utilizzato per comporre diversi files oggetto in un unico eseguibile.

7.2 Regole di visibilità per la compilazione separata

Quando il programma è suddiviso in vari file, si pone il problema di definire quali, fra le variabili globali e le funzioni definite in un file, debbano essere *visibili* (e quindi, usabili) in altri file. Per indicare dove una variabile globale o funzione è definita, e dove può essere usata, il C prevede tre modalità di dichiarazione, caratterizzate dalla presenza di determinate parole chiave prima della dichiarazione vera e propria. In particolare:

`extern` indica che la variabile è **definita in un altro file**, e che si intende usarla in questo file (variabile o funzione *importata*);

nessuna parola chiave indica che la variabile o funzione è **definita in questo file** e che **può essere usata in altri file** (a condizione che essi usino una dichiarazione `extern` per importarla); si tratta quindi di una variabile o funzione *pubblica*;

`static` indica che la variabile o funzione è **definita in questo file** e che **non può essere usata in altri file**; si tratta quindi di una variabile o funzione *privata*.

Per esempio, si consideri un programma in C diviso in tre file, con il contenuto indicato di seguito:

fileio.c

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

struct record r;
int errore=0;

extern void accu(int x);
extern int totale();

void leggi(char *file)
{
  int t;
  /* ... */
  t=read(fd,&r,sizeof(r));
  if ((t>0) && (t!=sizeof(r)))
    errore=1;
  /* ... */
  accu(r.saldo);
  /* ... */
  r=totale();
}
```

accum.c

```
static int ac=0;

void accu(int x)
{
  ac+=x;
}

int totale()
{
  return ac;
}

static void zero()
{
  ac=0;
}
```

main.c

```
#include <stdio.h>

extern int errore;

int main(int argc, char *argv[])
{
  /* ... */
  leggi("dati.bin");
  if (errore)
    exit(10);
  /* ... */
}
```

In questo programma, il file `accum.c` definisce due funzioni (che, in mancanza di altra specifica, sono pubbliche), e una variabile intera `ac` che, con la parola chiave `static`, è definita

privata, e quindi visibile esclusivamente all'interno di questo stesso file. Lo stesso vale per la funzione privata `zero`. Si noti che solo le funzioni `accu`, `totale` e `zero` possono accedere ad `ac`: si realizza così una sorta di incapsulamento, affine a quello tipico dei linguaggi orientati agli oggetti come Java. Infatti, possiamo pensare ad `accum.c` come alla definizione di un "oggetto" con due metodi pubblici e uno privato che operano su un campo privato.

Il file `fileio.c`, da parte sua, dichiara di voler accedere alle funzioni pubbliche di `accum.c` tramite le due dichiarazioni `extern`, e definisce a sua volta una funzione `leggi` che è visibile anche fuori da `fileio.c`. Inoltre, definisce due variabili globali che, in mancanza di ulteriori specifiche, sono visibili all'esterno del file: `r` e `errore`.

Vediamo che `main.c` accede effettivamente ad `errore` (che viene usato da `fileio.c` una situazione anomala), e chiama la funzione pubblica `leggi`. Tuttavia, nessuno ha bisogno di accedere alla `struct record r`: sarebbe dunque preferibile che `r` venisse dichiarata `static`, per eliminare il rischio di conflitti con altre variabili globali chiamate anch'esse `r` e dichiarate in altri file.

Si noti che la parola chiave `static` ha in C due significati completamente distinti: se usata con una variabile o funzione globale, come abbiamo visto or ora, indica che la variabile o funzione è privata al file in cui essa è definita; se invece viene applicata a una variabile locale, indica che la variabile deve mantenere il suo valore fra diverse invocazioni della funzione che la contiene.

Un'altra caratteristica interessante è che il C consente di dichiarare una variabile o funzione `extern` all'inizio di un file, anche se lo stesso file ne contiene poi la definizione. In questo caso, `extern` viene ignorata. Questa caratteristica si dimostra particolarmente utile nel caso (frequente) in cui tutte le dichiarazioni `extern` vengono raccolte in un file `.h`, che poi viene incluso da tutti i file `.c` — incluso, di volta in volta, quello che definisce la variabile o funzione in questione.

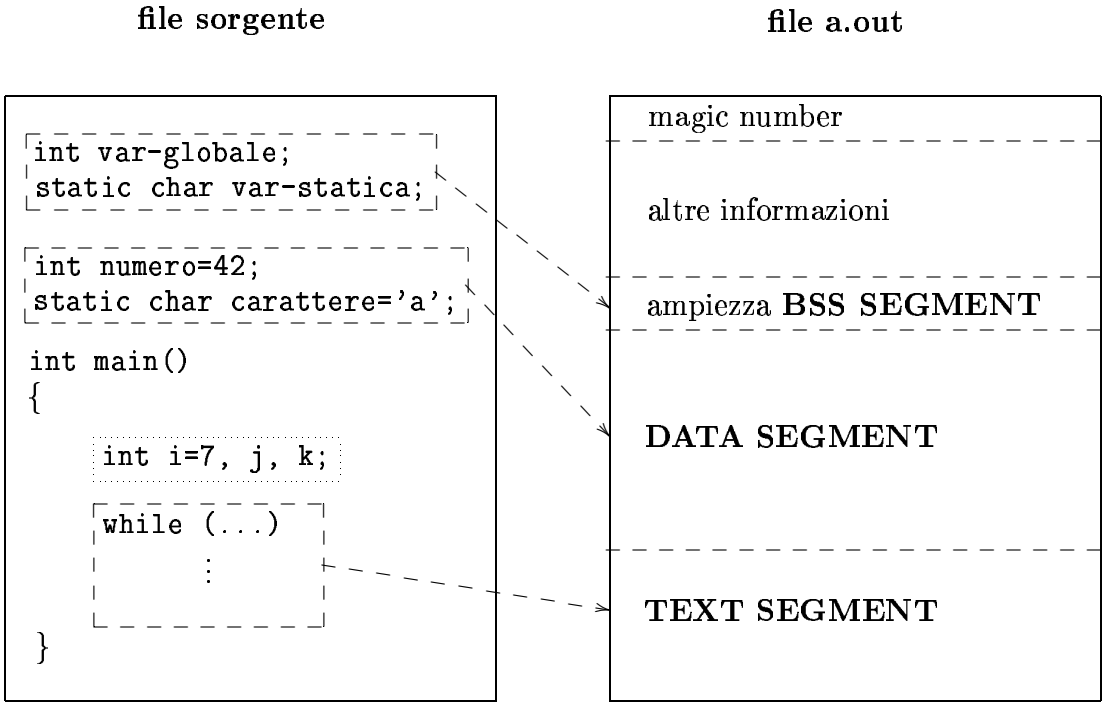
7.3 Il risultato della compilazione: l'eseguibile

Il risultato di una compilazione `gcc` *`sorgente`*`.c` è un file eseguibile di nome `a.out`. In questa sezione ci occuperemo della struttura di un file eseguibile e delle informazioni in esso contenute. Nella prossima sezione vedremo come viene eseguito.

In Linux un file eseguibile ha il formato ELF (*Executable and Linking Format*). Un file eseguibile ELF è composto da uno o più segmenti. In generale, un segmento è un'area di un file binario in cui si trovano informazioni omogenee, ad esempio il codice del programma, i dati del programma, etc. sono contenuti in dei segmenti separati. Il comando `size` mostra i segmenti di un eseguibile. Nel caso del programma sviluppato nella sezione precedente, il comando `size mioprogramma` stampa le seguenti informazioni:

```
text      data      bss      dec      hex filename
1035       232       24     1291      50b mioprogramma
```

Il primo segmento è il *text segment* che contiene il codice compilato del programma. Il secondo segmento è il *data segment* che contiene tutte le variabili globali e le variabili statiche inizializzate. Il terzo segmento *bss segment* (*Block Started by Symbol*) si riferisce alle variabili globali e alle variabili statiche non inizializzate. Poichè il valore di queste variabili non è noto, questo segmento non contiene veramente spazio per le variabili, ma solamente l'ampiezza dell'area di memoria richiesta per contenere tali dati. Vi sono poi altri due segmenti che contengono altre informazioni sul file `a.out`, come il *magic number* che contraddistingue il file come eseguibile. Si noti che le variabili automatiche (cioè locali alle funzioni) non sono presenti nel file eseguibile, ma vengono create dinamicamente a tempo di esecuzione. La figura che segue illustra come i vari componenti del file sorgente confluiscono nel file eseguibile.



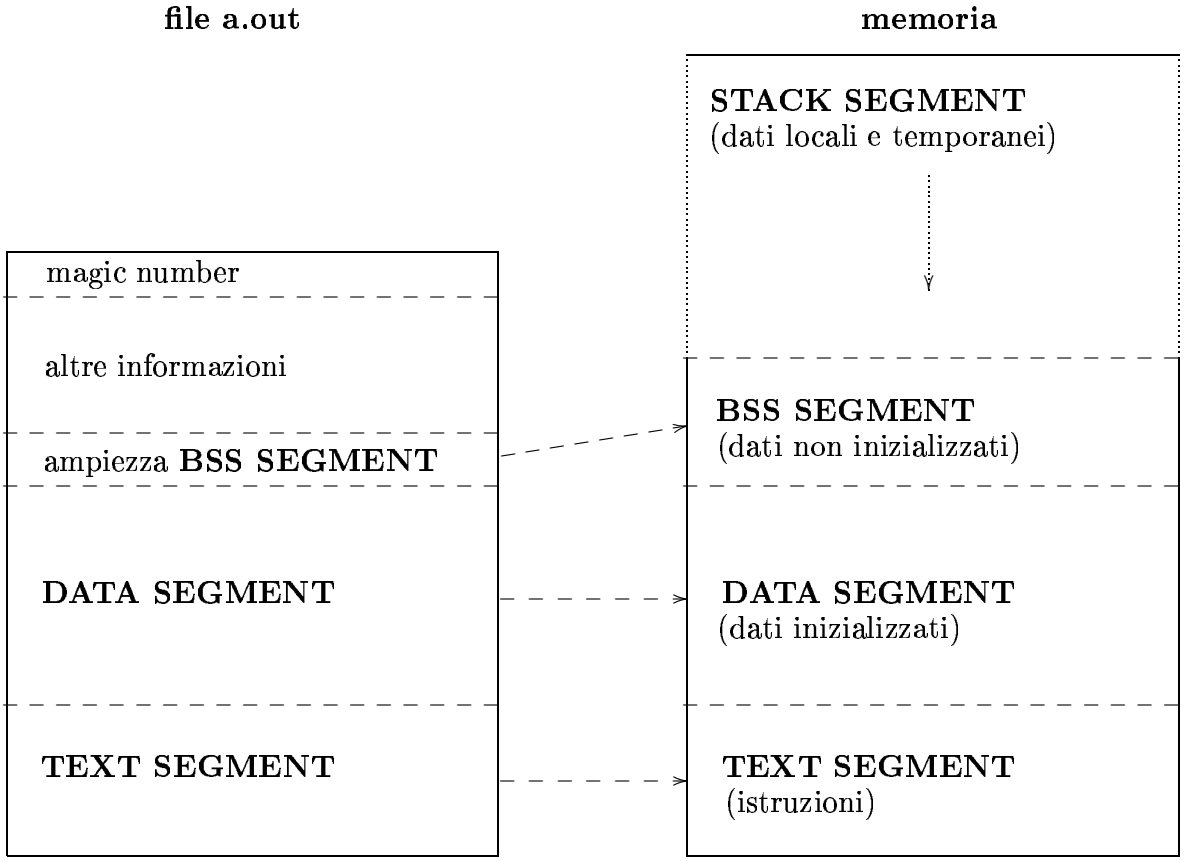
7.4 Esecuzione di un programma

Il *loader* si occupa di caricare in memoria le informazioni necessarie per eseguire un programma. La motivazione per dividere il file `a.out` in diversi segmenti risiede nel fatto che il loader può copiare direttamente i vari segmenti in memoria. I segmenti diventano quindi delle apposite aree di memoria del programma in esecuzione. Il *loader* copia direttamente in memoria il *text segment* (che contiene le istruzioni da eseguire) e il *data segment* (che con-

tiene le variabili globali e statiche inizializzate). Viene poi creato un segmento di memoria per contenere i dati globali non inizializzati, la cui ampiezza è scritta nel `bss segment` del file `a.out`. Infine vengono riservate ancora due aree di memoria:

1. lo *stack segment* che conterrà le variabili automatiche, i passaggi di parametri nelle chiamate di funzioni e in generale i valori temporanei;
2. lo *heap segment* utilizzato per allocare dinamicamente aree di memoria (con `malloc`).

La figura che segue mostra come i vari segmenti del file `a.out` sono mappati in corrispondenti aree di memoria (nello spazio di indirizzi del processo) prima che l'esecuzione abbia inizio.



Si noti che il *bss segment*, che contiene le variabili globali e statiche non inizializzate, viene opportunamente espanso al momento del caricamento in memoria. Ulteriore memoria per gestire le allocazioni dinamiche effettuate tramite `malloc` viene richiesta al sistema operativo durante l'esecuzione, al primo utlizzo della chiamata `malloc`.

Quando viene violato un segmento (ad esempio si cerca di accedere fuori del segmento) si verifica un errore di *segmentation fault*. Vediamo alcuni esempi di programmi che generano tale errore, insieme con le cause che li scaturiscono.

```
#include <stdio.h>
int main()
{
  int *puntatore = NULL;
  *puntatore = 3;
  return(0);
}
```

Questo programma cerca di dereferenziare un puntatore inizializzato ad un indirizzo che non appartiene allo spazio di indirizzamento logico del programma. La sua compilazione non genera nessun errore, ma l'esecuzione genera un *segmentation fault*, provocato dal fatto che si vuole accedere fuori dallo spazio di indirizzi logici del programma.

```
int main()
{
  int a[10];
  a[10000] = 1;
  return(0);
}
```

Questo programma produce un *segmentation fault* perché, nel tentativo di accedere ad un elemento che non appartiene all'array (provocando quindi un *buffer overflow*) si accede ad un indirizzo che non appartiene allo spazio di indirizzi logici del programma. Invece, nel programma che segue:

```
int main()
{
  int b[10000];
  int a[10];
  a[10000] = 1;

  return(0);
}
```

si verifica sempre un *buffer overflow*, ma non necessariamente un errore di *segmentation fault*, poiché in questo caso l'indirizzo di `a[10000]` potrebbe ancora appartenere allo spazio di indirizzi del programma, sebbene possibilmente non all'interno dell'array `a`, bensì dell'array `b`.

7.5 Il debugger

Lo scopo del debugger è di permettere al programmatore di controllare l'esecuzione di un programma. Per i sistemi Linux è disponibile `gdb`, il debugger simbolico della GNU (per informazioni si consulti la pagina web `http://www.gnu.org/software/gdb/`). Il `gdb` permette di eseguire un programma, fermare l'esecuzione, visualizzare i valori delle variabili, modificare tali valori e riprendere l'esecuzione. Il `gdb` può essere utilizzato dalla linea di comando con:

```
gdb programma-compilato
```

oppure può essere utilizzato un front-end grafico, come il `ddd` (Data Display Debugger, vedi `http://www.gnu.org/software/ddd/`). Il `ddd` è un software che permette di visualizzare e manipolare in forma grafica le informazioni generate dal `gdb`. Per la sua estrema facilità d'utilizzo, durante il corso utilizzeremo il `ddd`.

Supponiamo di voler analizzare il seguente programma `esempio.c`:

```
int main()
{
  int i=0, a[5];
  for(i=0;i<5;i++)
    a[i]=i;
 return(0);
}
```

Il primo passo consiste nella compilazione del programma utilizzando l'opzione `-g` la quale indica al compilatore di includere nell'eseguibile le informazioni per il debugger. Si noti che **non** è possibile utilizzare il debugger su programmi per i quali non sia stato generato il codice eseguibile. Di conseguenza, non è possibile utilizzare il debugger su programmi che presentano errori di sintassi. Dopo aver compilato il programma con:

```
gcc -Wall -g esempio.c -o esempio
```

è sufficiente richiamare il debugger con:

```
ddd esempio
```

ed apparirà una finestra con il codice sorgente ed il menu grafico con i principali comandi:

Run Inizia l'esecuzione del programma.

Interrupt Interrompe l'esecuzione del programma.

Step Esegue una linea del programma. Se la linea corrente è una chiamata di funzione, l'esecuzione passa al codice della funzione.

Next Esegue una linea del programma. Se la linea corrente è una chiamata di funzione, viene eseguito tutto il corpo della funzione senza mostrarne il codice.

Cont Continua l'esecuzione del programma.

Kill Termina il debugging.

Prima di iniziare il debugging del programma, è opportuno inserire dei breakpoint nel codice sorgente. Il breakpoint indica al debugger dove fermare l'esecuzione, per analizzare lo stato del programma. Per inserire un breakpoint è sufficiente posizionare il cursore a sinistra della linea a cui siamo interessati e premere `Break`. Per rimuovere un breakpoint occorre selezionarlo e premere `Clear`. Dopo aver inserito i breakpoint desiderati, si prema `Run` per iniziare l'esecuzione, la quale si fermerà sul primo breakpoint incontrato. Per visualizzare il valore di una variabile, selezionarne il nome e premere `Display`. In questo modo è possibile ispezionare il valore di tutte le variabili del programma. Tali valori vengono automaticamente aggiornati durante l'esecuzione. Infine, per terminare il debugging premere `Kill`.

Capitolo 8

Generalità sulle chiamate di sistema

8.1 Introduzione

[Glass, 382-384]

Un sistema di elaborazione complesso viene usualmente scomposto in diversi livelli di astrazione. A ciascun livello troviamo un linguaggio ed alcune risorse disponibili. Ad esempio, ai livelli più bassi troviamo il livello *firmware* (studiato nei corsi di *Architetture degli elaboratori*), con un linguaggio di micro-programmazione e risorse fisiche quali bus, registri e unità aritmetico-logiche. Ai livelli più alti troviamo il livello *applicazioni*, con un linguaggio di programmazione evoluto quale Java e con risorse astratte quali file, rete, e grafica. Il livello del *sistema operativo* è intermedio, e fornisce ai livelli superiori una astrazione della macchina fisica. L'astrazione del sistema operativo può così essere resa indipendente dalla particolare macchina fisica, se vengono prodotte versioni apposite del sistema operativo per una determinata macchina fisica. Le risorse rese disponibili dal sistema operativo UNIX, ad esempio, comprendono:

file management ovvero la disponibilità di diversi tipi di file e la loro organizzazione in una struttura a directory (detta *file system*),

process management ovvero la gestione di più programmi in esecuzione (detti processi) su una architettura uniprocessore o multiprocessore,

error handling ovvero la gestione di situazioni di errore.

A livello del sistema operativo non esiste un linguaggio di programmazione predefinito, ma piuttosto un insieme di "funzioni" per operare sulle risorse. Tali funzioni sono dette *chiamate di sistema*.

In generale, il compilatore o l'interprete di un linguaggio di programmazione evoluto, trasforma i costrutti del linguaggio (ad esempio, una lettura/scrittura di un file) in una o più

chiamate di sistema. Il linguaggio evoluto può così essere reso indipendente dal sistema operativo sottostante, se vengono prodotti compilatori/interpreti appositi per ciascun sistema operativo.

Lo standard ANSI C, ad esempio, definisce il linguaggio C e una libreria di funzioni standard che sono indipendenti dal sistema operativo. Un programma ANSI C dovrebbe compilare su un qualsiasi sistema operativo per cui sia disponibile un compilatore ANSI C.

Nei sistemi UNIX, in aggiunta, è possibile invocare direttamente da un programma C le funzioni offerte dal sistema operativo, ovvero le chiamate di sistema. L'insieme di tali chiamate è definito nello standard POSIX - Portable Operating System Interface. Se un compilatore C si dichiara conforme allo standard POSIX, allora sarà possibile compilare programmi contenenti chiamate alle funzioni POSIX. Sui sistemi UNIX, tali funzioni non fanno parte del codice prodotto dal compilatore, ma corrispondono a chiamate al sistema operativo.

L'obiettivo del resto di questa dispensa è quello di presentare un sottoinsieme delle chiamate POSIX, insieme ad esempi di programmi C che usano tali chiamate.

8.2 Manuali in linea

[Glass, 18-19]

I sistemi UNIX hanno un comando per la visualizzazione di informazioni sui comandi e programmi di sistema (sezione 1), chiamate di sistema POSIX (sezione 2), funzioni della libreria standard del C (sezione 3).

`man [-s section] word` ricerca `word`, nella sezione `section` se indicata. L'indicazione della sezione è necessaria se ci sono comandi o funzioni con lo stesso nome (ad esempio, si provi `man open` e `man 2 open`).

`man -k keyword` ricerca comandi o funzioni inerenti l'argomento `keyword`.

```
> man perror

PERROR(3) Library functions

NAME
        perror - print a system error message
SYNOPSIS
        #include <stdio.h>
        void perror(const char *s);
...
DESCRIPTION
```

```
...
CONFORMING TO
        ANSI C, POSIX, ...
SEE ALSO
...
```

I campi tipici di una pagina di manuale, ad esempio `man perror`, contengono:

sezione della pagina `PERROR(3) Library functions`, significa che `perror` è una funzione di libreria standard C,

nome della funzione `perror - print a system error message` descrive il nome e una breve descrizione di `perror`,

sinossi La sinossi

```
#include <stdio.h>
void perror(const char *s);
```

descrive gli include da richiamare nei programmi che usano la funzione `perror`, ed il prototipo della funzione.

descrizione è la descrizione dettagliata degli input e output della funzione,

standard sono gli standard che l'implementazione della funzione rispetta.

È inoltre disponibile una versione grafica di `man`, richiamabile con il comando `xman`.

8.3 Trattamento degli errori

[Glass, 385-386]

Tutte le chiamate di sistema ritornano un intero. Per convenzione, il valore di ritorno è `-1` nel caso in cui l'esecuzione della chiamata sia incorsa in errori (sia *fallita*). In questo caso, un codice di errore viene riportato nella variabile predefinita `errno`.

```
ERRNO(3) Library functions

NAME
        errno - numero dell'ultimo errore
SYNOPSIS
        #include <errno.h>
        extern int errno;
```

Nell'include standard `<errno.h>` sono dichiarate le macro per i vari tipi di errore, tra cui menzioniamo:

```
E2BIG    Arg list too long
EACCES   Permission denied
EAGAIN   Resource temporarily unavailable
EBADF    Bad file descriptor
...
```

Uno schema per il trattamento degli errori dopo una chiamata di sistema, ad esempio una `read(...)`, può essere il seguente.

```
if( read(...) == -1 ) {
             switch(errno) {
                  case E2BIG:    // azione
                                 break;
                  case EACCESS:  // azione
                                 break;
                  ...
             }
}
```

In realtà, la maggior parte delle volte è sufficiente stampare un messaggio di errore e terminare il programma. Per far questo, ci viene in aiuto la funzione di libreria `perror`. Una chiamata `perror(messaggio)` stampa sullo standard error il messaggio `messaggio` seguito da una descrizione in inglese del codice di errore contenuto in `errno`.

```
PERROR(3) Library functions

NAME
         perror - stampa un messaggio di errore
SYNOPSIS
         #include <stdio.h>

         void perror(const char *s);
```

Uno schema semplificato per il trattamento degli errori è quindi il seguente.

```
if( read(...) == -1 ) {
             perror("In lettura"); /* stampa msg di errore  */
             exit(errno); /* uscita dal prg con codice errno */
}
```

8.4 Macro di utilità: `sysmacro.h`

Nel seguito di questa dispensa, i programmi esempio utilizzeranno alcune macro di uso generale, che riportiamo nel file `sysmacro.h`.

```
/* File:            sysmacro.h
   Specifica:       macro per chiamate di sistema
*/

#include <stdio.h>  /* serve per la perror */
#include <stdlib.h> /* serve per la exit   */
#include <string.h> /* serve per strlen    */
#include <errno.h>  /* serve per errno     */
#include <unistd.h> /* serve per la write  */

#define IFERROR(s,m) if((s)==-1) {perror(m); exit(errno);}
#define IFERROR3(s,m,c) if((s)==-1) {perror(m); c;}

#define WRITE(m) IFERROR(write(STDOUT,m,strlen(m)), m);
#define WRITELN(m) WRITE(m);WRITE("\n");

#define STDIN  0
#define STDOUT 1
#define STDERR 2
```

La macro `IFERROR` implementa lo schema semplificato di gestione degli errori, per cui le chiamate di sistema avranno una forma del tipo:

```
...
IFERROR( read(...), "In lettura:" ); /* termina se errore */
```

Nei casi leggermente più complessi può essere utile un'azione diversa dalla terminazione del programma, ad esempio il ritorno da funzione.

```
...
IFERROR3( read(...), "In lettura:", return -1 );
   /* ritorna al chiamante se errore */
```

Le macro `WRITE` e `WRITELN` scrivono una stringa (ed un ritorno carrello nel caso di `WRITELN`) sullo standard output invocando la chiamata di sistema `write` (vedi Sezione 9.4). Infine, `STDIN`, `STDOUT` e `STDERR` denotano le costanti identificative dello standard input, output ed error rispettivamente.

8.5 Makefile generico

[Glass, 347-354]

I programmi che verranno presentati sono compilati utilizzando un `makefile` con una forma standard.

```
# File:      makefile
# Specifica: makefile generico

CC = gcc
CFLAGS = -Wall -g

# dipendenze eseguibile: oggetti

# dipendenze oggetto: header

clean:
        rm -f *~ core
cleanall:
        rm -f *.o *~ core
```

In particolare, compiliamo con le opzioni `-Wall` (warning su tutto) e `-g` (include informazioni per il debugger).

8.6 Esempi di questa dispensa

I programmi descritti in questa dispensa sono disponibili in formato sorgente all'indirizzo:

`http://www.di.unipi.it/didadoc/lps/LPS_Sources.zip`

Il file viene decompresso con il comando `unzip LPS_Sources.zip`. La decompressione crea una directory `LPS_Sources` con sottodirectory dal nome `Cap8`, `Cap9`, ecc. Nella sottodirectory `Cap8` si trova una directory per ogni esempio presentato nel Capitolo 8, e così per gli altri capitoli della dispensa. Per quanto concerne il presente capitolo, sono presenti le sottodirectory:

Perror contiene un esempio completo di uso della funzione `perror`, come descritto nella Sezione 8.3;

Macro_Perror contiene l'esempio precedente riscritto utilizzando le macro di utilità descritte nella Sezione 8.4;

File_Standard contiene l'include `sysmacro.h` e lo schema di makefile presentato nella Sezione 8.5.

Capitolo 9

Gestione dei file

9.1 Cenni sul file system di UNIX

9.1.1 Organizzazione logica e livello utente

[Glass, 23-48]

Il sistema UNIX fornisce una struttura gerarchica (ad albero) dei file detta *file system*, organizzata in directory e sottodirectory. Un file è una collezione di dati residente su memoria secondaria (floppy disk, disco rigido, compact disc, nastro, etc.). Una directory (in italiano, "cartella") è un file contenente altri file o directory[1]. I file in senso stretto, cioè non directory, sono detti file di tipo *regolare*. Ciascun file o cartella ha un nome (*pathname*) che individua esattamente la sua posizione nella gerarchia (si veda **[Glass, 24-26]**), o in modo assoluto (a partire dalla directory radice) o in modo relativo (a partire dalla directory corrente di lavoro). Dalla shell di UNIX è possibile muoversi nella gerarchia (comando `cd`), leggere/creare un file regolare (`cat`, `more`), editare un file regolare (`emacs`, `vi` o altro editor) elencare il contenuto di una directory (`ls`, `ls -a`), modificare il nome di file regolari o directory (`mv`), copiare file regolari o directory (`cp`, `cp -r`) cancellare file regolari (`rm`) e directory (`rmdir`), conoscere la directory corrente di lavoro (`pwd`), stampare un file (`lp`, `lpr`, `lpq`, `lprm`).

I file regolari possono comparire in *repliche* (o *hard link*) all'interno del file system. Due repliche sono file regolari con nomi possibilmente anche distinti, in directory possibilmente anche distinte. In realtà, però le due repliche hanno sempre lo stesso contenuto, ovvero una modifica ad una replica significa automaticamente cambiare tutte le repliche. Come si può facilmente arguire, esiste in realtà una sola copia fisica sul file system e non una per ogni replica. Per creare una replica utilizziamo il comando `ln file replica` (si veda **[Glass, 248-250]**). Cancellando una replica dal file system, non si cancellano tutte le altre. Il contenuto delle repliche viene perso solo al momento della cancellazione dell'ultima replica.

[1] In particolare, ciascuna directory contiene sempre due directory implicite dai nomi "." e "..". Esse corrispondono a "directory corrente" e "directory padre" rispettivamente.

Oltre ai file regolari ed alle directory, esistono altri tipi di file in UNIX. I file *speciali a blocchi* e *speciali a caratteri* sono astrazioni di periferiche. In altre parole, leggere/scrivere su tali file significa leggere/scrivere su una periferica[2] (terminale, floppy, disco rigido, nastro, etc.). I file di tipo *socket* sono invece utilizzati per la comunicazione client-server tra processi, anche se residenti su macchine collegate in rete. In altre parole, legggere/scrivere su tali file significa spedire/ricevere un messaggio da un altro processo. Nel corso di questa dispensa non vedremo i file speciali nè i socket.

Nel Capitolo 13 vedremo i file di tipo *pipe*, utilizzati per la comunicazione tra processi residenti sulla stessa macchina. Infine, menzioniamo i file di tipo *link simbolico*. Un link simbolico è un file che *riferisce* un altro file. È possibile creare un link simbolico con il comando `ln -s file link_simbolico`. L'accesso al link simbolico in lettura/scrittura viene automaticamente ridiretto sul file collegato. Come si può osservare esiste una analogia con le repliche. Una differenza è che i link simbolici possono esistere tra file fisicamente locati su dischi distinti, mentre le repliche no. Ancora più importante, è il fatto che se si cancella il file referenziato, il link simbolico punterà a un file inesistente, mentre una replica conterrà ancora il contenuto del file cancellato.

Per ciascun file sono definiti (si veda [**Glass, 42-48**]) un proprietario, un gruppo di utenti cui appartiene il proprietario, e dei diritti di accesso in lettura, scrittura ed esecuzione rispettivamente per il proprietario, il gruppo e tutti gli altri. È possibile cambiare il proprietario, il gruppo e i diritti di un file mediante i comandi `chown`, `chgrp`, `chmod`.

Concludiamo questa sezione vedendo come sia possibile elencare tutte le informazioni riguardo ad un file. Il comando da utilizzare è `ls -l` su una directory o su un file. Mostriamo una sessione di esempio.

```
> ls -l                          ... directory vuota
total 0
> cat > a                        ... creo un file di esempio
Laboratorio IV
> mkdir b                        ... creo una directory
> ln a c                         ... creo una replica di a
> ln -s a d                      ... creo un link simbolico di a
> ls -l
total 3
   1 -rw-r--r--    2 ruggieri personal       15 Feb  5 19:32 a
   1 drwxr-xr-x    2 ruggieri personal     1024 Feb  5 19:32 b/
   1 -rw-r--r--    2 ruggieri personal       15 Feb  5 19:32 c
   0 lrwxrwxrwx    1 ruggieri personal        1 Feb  5 19:33 d -> a
> cat >> c                       ... modifico la replica
A.A. 2000-2001   ... concludo l'input con Ctrl-D
```

[2]Si provi la seguente sequenza di comandi: `tty` produce il nome del file speciale corrispondente al terminale su cui si è collegati. Supponiamo sia qualcosa del tipo `/dev/pts/0`. Si provi quindi ad eseguire il comando `ls > /dev/pts/0`. Dove abbiamo ridiretto l'output del comando `ls`?

```
> ls -l
total 3
   1 -rw-r--r--    2 ruggieri personal       30 Feb  5 19:33 a
   1 drwxr-xr-x    2 ruggieri personal     1024 Feb  5 19:32 b/
   1 -rw-r--r--    2 ruggieri personal       30 Feb  5 19:33 c
   0 lrwxrwxrwx    1 ruggieri personal        1 Feb  5 19:33 d -> a
> cat a                           ... la modifica vale anche per a
Laboratorio IV
A.A. 2000-2001
> cat d                           ... la modifica vale anche per d
Laboratorio IV
A.A. 2000-2001
> rm a                            ... rimuovo a
rm: remove ‘a’? y
> cat c                           ... la replica esiste ancora
Laboratorio IV
A.A. 2000-2001
> cat d                           ... il link simbolico e’ perso
cat: d: No such file or directory
>
```

Si noti come una riga prodotta da `ls -l`, quale

```
   1 -rw-r--r--    2 ruggieri personal       30 Feb  5 19:33 a
```

produce le seguenti informazioni:

- `1` il numero di unità di memorizzazione (*blocchi*) che il file occupa su disco;
- `-` il tipo del file (in questo caso, `-` significa regolare);
- `rw-r--r--` i diritti di accesso in lettura/scrittura/esecuzione per proprietario, gruppo e altri;
- `2` il numero di repliche esistenti (compreso il file stesso);
- `ruggieri` il proprietario,
- `personal` il gruppo del proprietario,
- `30` la lunghezza in bytes del file,
- `Feb 5 19:33` data e ora dell'ultima modifica al file,
- `a` il nome del file.

9.1.2 Organizzazione fisica e implementazione in UNIX

Rimandiamo al libro di testo [**Glass, 521-532**] la presentazione dell'organizzazione fisica di un disco (piatto, traccia, settore, blocco, etc.), la modalità di memorizzazione di un file su disco e l'implementazione del file system attraverso gli *i-nodi* (superblocco, matrice di allocazione dei blocchi, i-nodi e blocchi dati, implementazione delle repliche, implementazione delle directory, trasformazione di un pathname nel relativo i-nodo).

9.2 Apertura di un file: `open`

[Glass, 386-397]

L'accesso ai file richiede una operazione preliminare, detta di "apertura" del file. L'apertura avviene mediante una chiamata

```
int open(const char *pathname, int flags);
```

che richiede di specificare:

`pathname` il nome del file che si intende aprire. Il nome può essere un pathname relativo (dati.txt, ../dati.txt) o assoluto (/tmp/dati.txt);

`flags` le modalità di accesso al file. Deve essere specificato esattamente una tra le seguenti macro:

- `O_RDONLY` apertura per sola lettura,
- `O_WRONLY` apertura per sola scrittura,
- `O_RDWR` apertura per lettura e scrittura,

eventualmente in or bit a bit con uno o piu' tra:

- `O_APPEND` scrittura in coda al file,
- `O_CREAT` se il file non esiste viene creato,
- `O_TRUNC` in fase di creazione, se il file esiste già viene troncato,
- `O_EXCL` in fase di creazione, se il file esiste già viene segnalato un errore.

La chiamata `open` ritorna `-1` in caso di errore (impostando `errno` con il codice di errore corrispondente). In caso di successo, ritorna un intero non negativo detto *descrittore del file*, il quale viene utilizzato nei successivi accessi in scrittura, lettura e chiusura per identificare il file aperto[3]. Intuitivamente, un descrittore è l'indice di un array (*tabella dei descrittori*

[3]Si noti che il modo di indentificare i file aperti mediante descrittori è diverso dalle funzioni di libreria ANSI C per la gestione dei file, le quali usano gli *stream*, ovvero puntatori ad una struttura di tipo `FILE`. Mentre gli stream e le operazioni su di essi sono standard del linguaggio C, le chiamate descritte in questo capitolo sono standard POSIX, quindi disponibili solo su compilatori che rispettano questo standard.

dei file) i cui elementi mantengono le informazioni necessarie per accedere ai file aperti. Un modo standard di aprire un file in lettura, sarà quindi una istruzione del tipo:

```
int fd; /* descrittore del file */
...
fd = open("dati.txt", O_READ);
if( fd == -1 ) {
            perror("Aprendo dati.txt");
            exit(errno); /* oppure altra azione */
}
```

che può essere riscritto più succintamente utilizzando le macro definite in `sysmacro.h`

```
int fd; /* descrittore del file */
...
IFERROR( fd = open("dati.txt", O_RDONLY), "Aprendo dati.txt");
```

Nel caso di scrittura, la modalità di apertura sarà `O_WRONLY | O_CREAT | O_TRUNC`, ovvero apertura in scrittura (`O_WRONLY`) con eventuale creazione del file (`O_CREAT`) o eventuale troncamento se esistente (`O_TRUNC`). La mancata indicazione di `O_CREAT` fà fallire la chiamata se il file non esiste già. La mancata indicazione di `O_TRUNC` lascia *intatto* il file nel caso esista già: questo significa che se sovrascriviamo i primi 100 bytes di un file di 1000 bytes, allora i restanti 900 rimarranno intatti.

L'apertura mediante la chiamata

```
int open(const char *pathname, int flags, mode_t mode );
```

viene utilizzata quando si vogliono specificare i diritti con cui il file viene eventualmente creato (quindi ha senso usarla solo se `flags` include `O_CREAT`). Ad esempio,

```
int fd; /* descrittore del file */
...
IFERROR( fd = open("dati.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666), "Creando dati.txt");
```

apre il file `dati.txt` eventualmente creandolo con diritti espressi in ottale `0666`, o mediante or di macro (si veda `man 2 open`). In realtà i diritti indicati sono messi in "and bit a bit" la "negazione bit a bit" di una maschera dei diritti detta `umask`, con cui l'utente può indicare quali diritti vuole che i nuovi file **non** abbiano, indipendentemente dai diritti richiesti dal programma che li crea . Se la maschera vale, ad esempio, `0022`, allora i diritti assegnati saranno `0644`. La maschera dei diritti può essere acceduta/cambiata dalla shell con il comando `umask [nuova_maschera]`.

```
OPEN(2)

NAME
       open - apertura, ed eventualmente creazione di un file

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int open(const char *pathname, int flags);
       int open(const char *pathname, int flags, mode_t mode);
```

Alla partenza di ciascun programma, sono implicitamente aperti i descrittori dello standard input (descrittore 0), standard output (descrittore 1) e standard error (descrittore 2). In `sysmacro.h` abbiamo definito opportune macro per questi valori (`STDIN`, `STDOUT` e `STDERR`).

9.3 Chiusura di un file: `close`

[Glass, 401]

Una volta terminate le operazioni su un file aperto, occorre eseguire una operazione finale di "chiusura", sia per liberare il descrittore di file (che può quindi essere riassegnato in una successiva apertura) sia per garantire che eventuali buffer del sistema operativo siano riversati su disco.

```
CLOSE(2)

NAME
       close - chiusura di un descrittore di file

SYNOPSIS
       #include <unistd.h>

       int close(int fd);
```

9.4 Lettura e scrittura di un file: `read` e `write`

[Glass, 397-398]

La lettura da un file precedentemente aperto in lettura avviene mediante una chiamata

```
ssize_t read(int fd, void *buf, size_t count);
```

la quale richiede che sia indicato un descrittore (`fd`), l'indirizzo di un'area di memoria o buffer (`buf`) in cui scrivere ed il numero di bytes da leggere (`count` di tipo `size_t`[4]). La lettura avviene a partire dalla posizione corrente di lettura (dopo la `open`, l'inizio del file) e aggiorna tale posizione. Se dalla posizione di lettura alla fine del file vi sono meno di `count` bytes, vengono letti solo quelli esistenti. La `read` ritorna un intero (`ssize_t` è sostanzialmente `int`):

`-1` in caso di errore,

`0` se la posizione di lettura è già a fine file,

`n` con `n` $\leq$ `count` se la posizione di lettura non è a fine file e sono stati letti `n` bytes.

Pertanto, per scorrere un intero file si scrive di solito del codice della forma:

```
int fd, letti;
char dati[MAX+1]; /* allocazione statica */

IFERROR(fd = open("dati.txt", O_RDONLY), "In apertura");
while( (letti=read(fd,dati,MAX)) > 0 ) {
    ...
}
IFERROR( letti, "In lettura");
```

Ad ogni passo del ciclo `while` si tenta di leggere `MAX` bytes. In tutti i passi tranne l'ultimo si leggono effettivamente `MAX` bytes, mentre nell'ultimo si leggono *al massimo* `MAX` bytes. All'uscita dal ciclo, o si è raggiunta la fine del file (`letti` vale 0) oppure c'è stato un errore (`letti` vale -1). L'`IFERROR` finale controlla che non ci sia stato un errore.

```
READ(2)

NAME
       read - lettura da un descrittore di file

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);
```

[4] `size_t` è il tipo usato per esprimere una misura, ed è definito come `typedef unsigned int size_t;`

L'area di buffer può anche essere allocata dinamicamente (per esempio perché non si conosce a priori la dimensione del buffer). Attenzione, però a non dimenticare di allocare la memoria prima di leggere.

```
int fd, letti;
char *dati;

dati = (char *) malloc( MAX ); /* allocazione dinamica */
IFERROR(fd = open("dati.txt", O_RDONLY), "In apertura");
while( (letti=read(fd,dati,MAX)) > 0 ) {
    ...
}
IFERROR( letti, "In lettura");
```

Si noti che la `read` da standard input, a differenza delle funzioni di input ANSI C, non scrive automaticamente lo `\0` finale dopo la lettura di una stringa. Dopo aver digitato una stringa, l'operatore può premere [Control-D] (fine file) per indicare la fine dell'input, oppure [Invio] per indicare la fine di una riga. Dal momento, però che lo standard input si comporta come un *pipe* (concetto che studieremo nel Capitolo 13), la pressione di [Invio] rende subito disponibili i caratteri digitati alla `read`: in `buf` viene scritto quanto digitato, compreso lo `\n` finale. Per leggere una stringa terminata da [Control-D] o da [Invio] si userà quindi un frammento di programma del tipo:

```
char buf[MAX];
int  letti;

WRITE("Scrivi una stringa e poi [Control-D]: ");
IFERROR( letti = read(STDIN, buf, MAX), "Leggendo da stdin");
buf[letti] = '\0'; /* determino la fine della stringa */

WRITE("Scrivi una stringa e poi [INVIO]: ");
IFERROR( letti = read(STDIN, buf, MAX), "Leggendo da stdin");
buf[letti-1] = '\0'; /* in buf[letti-1] c'e' il carattere '\n' */
```

Analogamente, la scrittura su un file precedentemente aperto in scrittura avviene mediante una chiamata

```
ssize_t write(int fd, const void *buf, size_t count);
```

la quale richiede che sia indicato un descrittore (`fd`), l'indirizzo di un'area di memoria o buffer (`buf`) da cui leggere ed il numero di *bytes* da scrivere (`count`). La scrittura avviene a

partire dalla posizione corrente di scrittura (dopo la `open`, l'inizio del file o la fine del file, a seconda che si sia o meno aperto con modalità `O_APPEND`).

Ad esempio, per scrivere sullo standard output (implicitamente aperto in scrittura per ogni programma), si usa una chiamata:

```
write(1, Corso di laboratorio, 20);
```

dove `1` è il descrittore dello standard output e `20` è il numero di caratteri della stringa `Corso di laboratorio` che vogliamo stampare (si noti che la stringa contiene in realtà 21 caratteri, l'ultimo dei quali è lo `\0` finale che non vogliamo stampare). Grazie alle macro definite in `sysmacro.h`, possiamo riscrivere l'istruzione come:

```
WRITE(Corso di laboratorio);
```

La write restituisce il numero di byte scritti, in caso di successo, e -1 in caso di errore (impostando `errno` con il codice di errore corrispondente).

Si noti anche che ogni singola operazione di scrittura è atomica, nel senso che se due o più programmi tentano di scrivere su uno stesso file, scrivono uno alla volta (accesso in mutua esclusione) secondo una schedulazione del sistema operativo non predicibile a priori. Si noti infine che la `write`, a differenza della funzione di libreria `printf`[5], non effettua alcuna bufferizzazione dell'output. Quindi una `write` sullo standard output produce immediatamente il risultato a video (a meno che lo standard output non sia stato rediretto).

```
WRITE(2)

NAME
       write - scrive in un descrittore di file

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);
```

Riprendendo l'esempio della `read`, per scorrere un intero file e scriverlo sullo standard output si scrive del codice della forma:

```
    int fd, letti;
```

[5]Ricordiamo che le funzioni di libreria ANSI C sono standard del C ed indipendenti dal sistema operativo, mentre le chiamate di sistema POSIX sono appunto chiamate al sistema operativo UNIX/LINUX. Ovviamente, sui sistemi UNIX/LINUX la definizione della funzione `printf` utilizzerà le chiamate di sistema (quindi la `write`).

```
    char dati[MAX]; /* allocazione statica */

    IFERROR(fd = open("dati.txt", O_RDONLY), "In apertura");
    while( (letti=read(fd,dati,MAX)) > 0 ) {
        IFERROR( write(STDOUT, dati, letti), "In scrittura");
    }
    IFERROR( letti, "In lettura");
    IFERROR( close(fd), "In chiusura");
```

Ad ogni passo del ciclo tranne l'ultimo si leggono `MAX` caratteri dal file `dati.txt` e si scrivono sullo standard output. Nell'ultimo passo si leggono al massimo `MAX` caratteri e si scrivono sullo standard output. Un eventuale errore in lettura causa l'uscita dal ciclo: occorre quindi, subito dopo, controllare il valore di `letti`.

9.5 Esempi ed esercizi

9.5.1 Esempio: `mycat`

Riportiamo di seguito il programma `mycat`, il quale scrive sullo standard output il contenuto di un file. In altre parole, `mycat` implementa il comando di shell `cat filename`. Il corpo principale del programma è il ciclo di lettura/scrittura visto nella sezione precedente.

```
/* File:            mycat.c
   Specifica:       implementazione del comando > cat filename
*/

/* include per chiamate sui file */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "sysmacro.h" /* macro di utilita' */

#define MAXBUFFER 80  /* dimensione del buffer di I/O */

int main(int argc, char * argv[])
{
  int fd, letti;
  char buffer[MAXBUFFER];

  if( argc != 2 ) {
    WRITELN("Usage:\nmycat filename");
    exit(0);
```

```
  }

  /* apertura del file */
  IFERROR(fd = open(argv[1],O_RDONLY),argv[1]);

  /* ciclo di lettura/scrittura */
  while ( (letti = read(fd,buffer,MAXBUFFER)) > 0 )
    IFERROR(write(STDOUT,buffer,letti), "Standard output");

  /* controllo all'uscita dell'ultima lettura*/
  IFERROR(letti, argv[1]);

  /* chiusura del file */
  IFERROR(close(fd),argv[1]);

  return(0);
}
```

Esercizi

1 Modificare `mycat` in modo che prenda un ulteriore parametro corrispondente alla dimensione del buffer di I/O.

2 Eseguire `mycat` (o il programma al punto `1`) con diverse dimensioni per il buffer (ad esempio, 1, 100, 10000 caratteri) su un file abbastanza grande (>10Kb) e calcolare i tempi di esecuzione[6]. Commentare il risultato, ed in particolare il fatto che siano tempi molto differenti?

3 Modificare `mycat` in modo che quando invocato senza parametri legga dallo standard input.

4 Modificare `mycat` in modo che quando invocato con uno o più parametri legga i file corrispondenti uno alla volta scrivendoli sullo standard output.

5 Si provi ad eseguire il comando `strace mycat mycat.c > output`. Su `output` abbiamo ridiretto il risultato di `mycat mycat.c`. Sullo standard error, invece, compare l'elenco di tutte le chiamate di sistema che il programma `mycat` ha effettuato.

[6] Il comando `time comando parametri` esegue il comando `comando` con parametri `parametri` e ne riporta i tempi di esecuzione.

9.5.2 Esempio: `mycopy`

Riportiamo di seguito il programma `mycopy`, il quale copia il contenuto di un file su un altro file, creandolo se necessario. In altre parole, `mycopy` implementa il comando di shell `cp file1 file2`. Il corpo principale del programma è ancora un ciclo di lettura/scrittura.

```
/* File:            mycopy.c
   Specifica:       implementazione del comando > cp file1 file2
*/

/* include per chiamate sui file */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "sysmacro.h" /* macro di utilita' */

#define MAXBUFFER 8192  /* dimensione del buffer di I/O */

int main(int argc, char * argv[])
{
  int fd1, fd2, letti;
  char buffer[MAXBUFFER];

  if( argc != 3 ) {
    WRITELN("Usage:\nmycopy file1 file2");
    exit(0);
  }

  /* apertura del file1 */
  IFERROR(fd1 = open(argv[1],O_RDONLY),argv[1]);

  /* apertura/creazione del file2 */
  IFERROR(fd2 = open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,0644),argv[2]);

  /* ciclo di lettura/scrittura */
  while ( (letti = read(fd1,buffer,MAXBUFFER)) > 0 )
    IFERROR(write(fd2,buffer,letti), argv[2]);

  /* controllo all'uscita dell'ultima lettura*/
  IFERROR(letti, argv[1]);

  /* chiusura dei file */
  IFERROR(close(fd1),argv[1]);
  IFERROR(close(fd2),argv[2]);
```

```
  return(0);
}
```

Esercizi

1 Modificare `mycopy` in modo che quando il file da scrivere esiste chieda conferma.

2 Quanti file è possibile tenere aperti contemporaneamente? Scrivere un programma che in un ciclo apre continuamente un file (senza richiuderlo), per capire a quale iterazione la `open` non è più in grado di ritornare un descrittore.

3 Modificare `mycopy` in modo che quando sia passato un solo parametro, legga dallo standard input.

4 Modificare `mycopy` in modo che accetti n $>$ 1 parametri, Il comando `mycopy file-1 ... file-n` copia uno di seguito all'altro il contenuto di `file-1`, ..., `file-(n-1)` nel file `file-n`.

5 Scrivere il comando `mycopydir`, il quale invocato con `mycopydir nome-1 ... nome-n dirdest` (con n $>$ 0) copia i file `nome-1`, ..., `nome-n` nella directory `dirdest`. Suggerimento: si scriva una funzione `void copyfile(char * namesrc, char * namedest);` che copia il file `namesrc` sul file `namedest`. Esempio di sessione:

```
> ls
total 28
   2 a.c    26 mycopydir*
> mkdir DIR
> mycopydir
Usage:
mycopydir file1 ... filen directory
> mycopydir pippo a.c DIR
pippo: No such file or directory
> ls DIR
total 2
   2 a.c
```

6 Supponiamo di avere una qualche struttura `struct my_struct data` e di effettuare una scrittura `write(fd, &data, sizeof(data))`. Cosa abbiamo salvato? Scrivere due programmi, uno che effettua la `write` precedente, ed uno che effettua una `read(fd, &data, sizeof(data))`. Che valori trovate nei campi di `data` dopo la lettura?

9.6 Posizionamento: `lseek`

[Glass, 399-400]

I programmi di esempio della sezione precedente accedono ai file in modalità sequenziale, ossia partendo dall'inizio e scorrendo fino alla fine l'intero file. Ad ogni lettura/scrittura, la posizione di lettura/scrittura viene automaticamente aggiornata dalla `read`/`write`.

È anche possibile accedere ad un file in modalità *random*, ovvero spostando la posizione di lettura/scrittura in qualsiasi punto del file. La chiamata di sistema:

```
off_t lseek(int fd, off_t offset, int whence)
```

permette di spostare la posizione di lettura/scrittura del descrittore `fd` di un certo numero di bytes `offset` (positivo o negativo) rispetto ad un riferimento `whence`, il quale può essere una delle seguenti macro:

`SEEK_SET` indica che lo spostamento è relativo all'inizio del file,

`SEEK_CUR` indica che lo spostamento è relativo alla posizione corrente,

`SEEK_END` indica che lo spostamento è relativo alla fine del file.

Il tipo `off_t` è un `long`. Ad esempio, `lseek(fd, 0, SEEK_SET)` ci posiziona all'inizio del file; `lseek(fd,-1,SEEK_END)` ci posiziona all'ultimo byte; `lseek(fd, 1,SEEK_CUR)` avanza di un byte rispetto alla posizione corrente. La chiamata `lseek` ritorna la posizione corrente (in bytes) rispetto all'inizio del file in caso di successo e `-1` in caso di errore (impostando `errno` con il codice di errore corrispondente).

```
LSEEK(2)

NAME
       lseek - posizionamento all'interno di un file

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       off_t lseek(int fd, off_t offset, int whence);
```

Un uso tipico di lseek consiste nella implementazione di accesso random a porzioni di dimensione fissa di un file (*record*). Nel frammento di programma di seguito, viene acceduto in lettura/scrittura l'`i`-esimo record con il contenuto di una variabile di tipo `struct record`. Il codice (in cui, per semplicità, abbiamo omesso il controllo degli errori) funziona indipendentemente da come è definita la struttura `struct record`, dal momento che usa `sizeof` per determinarne la lunghezza in bytes.

```
struct record  a;
...
fd = open("filedati", O_RDWR | O_CREAT);
...
   /* posiziona sull'i-esimo record */
lseek(fd, sizeof(struct record) * i, SEEK_SET);
   /* legge l'i-esimo record nella variabile a */
read(fd, &a, sizeof(struct record));
...
  /* eventuale modifica dei campi di a */
...
   /* posiziona sull'i-esimo record */
lseek(fd, sizeof(struct record) * i, SEEK_SET);
    /* scrive la variabile a sull'i-esimo record */
write(fd, &a, sizeof(struct record));
```

È importante sottolineare come, accedendo ad un file in scrittura con modalità random l'apertura del file debba essere del tipo `O_RDWR | O_CREAT` oppure del tipo `O_WRITE | O_CREAT`, ovvero *non* deve contenere l'opzione `O_TRUNC` – altrimenti si cancellerebbe il contenuto del file.

9.7 Esempi ed esercizi

9.7.1 Esempio: `seekn`

Il seguente programma implementa un comando `seekn filename n start`. Se `start` è un intero positivo, il programma stampa `n` bytes a partire da `start` bytes dall'inizio del file `filename`. Se `start` è un intero negativo, il programma stampa `n` bytes a partire da `start` bytes dalla fine[7] del file `filename`.

La lettura avviene con una sola chiamata, allocando dinamicamente un `buffer` della dimensione opportuna. La trasformazione dei parametri da stringhe a interi avviene mediante la funzione di libreria `int atoi(const char *nptr)`. Ad esempio, `atoi(42)` ritorna l'intero 42.

```
/* File:            seekn.c
   Specifica:       seek file n start - scrive sullo stdout (al piu') gli
                    n bytes di file dopo i primi start bytes. Se start e'
                    negativo, legge a partire dalla fine del file.
*/
```

[7] Si noti come, dal momento che `start` è negativo, questo significa `-start` bytes prima della fine del file.

```
/* include per chiamate sui file */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "sysmacro.h" /* macro di utilita' */

int main(int argc, char *argv[])
{
  int fd, start, n, letti;
  char *buffer;

  /* controlla argomenti */

  if( argc != 4){
      WRITELN("Usage: seekn filename n start");
      exit(0);
  }

  n = atoi(argv[2]);
  start = atoi(argv[3]);

  if( n <= 0 ) {
    WRITELN("n  must be positive");
    exit(1);
  }

  /* apertura del file */
  IFERROR(fd = open(argv[1],O_RDONLY), argv[1]);

  /* posizionamento */
  if( start >= 0 ) {
    IFERROR(lseek(fd,start,SEEK_SET),argv[1]);
  } else {
    IFERROR(lseek(fd,start,SEEK_END),argv[1]);
  }

  /* alloca memoria */
  if( (buffer = (char*) malloc( n )) == NULL ) {
    WRITELN("No more memory");
    exit(-1);
  }

  /* lettura */
  IFERROR(letti = read(fd,buffer,n),argv[1]);
```

```
  /* stampa in output */
  IFERROR(write(STDOUT,buffer,letti),"stdout");

  /* chiusura file */
  IFERROR(close(fd),argv[1]);

  return(0);
}
```

Esercizi

1 Modificare `seekn` utilizzando un buffer di dimensione fissa. Questo comporta di dover sostituire l'unica lettura con un ciclo di una o più.

2 Scrivere un programma che crea un file con descrittore `fd`, quindi effettua una `lseek(fd, 100, SEEK_SET)` (posizionandosi oltre la fine del file), quindi effettua una qualche scrittura con `write`. Qual'è lo stato del file creato al termine del programma. Cosa c'è nei primi 100 bytes? Per leggere un file contenente anche caratteri non stampabili (file *binario*) si usi il comando `od -cAd nomefile | more`.

3 Si scriva un programma che implementa il comando `writear`, il quale invocato con `write nomefile` chiede all'operatore un nome. Se il nome è uguale a fine, il comando termina, altrimenti chiede l'età e memorizza nome ed età in una variabile `rec` di tipo RECORD.

```
#define MAXNAME 128

typedef struct record {
  char name[MAXNAME];
  int age;
} RECORD;
```

Quindi scrive sul file `nomefile` l'area di memoria occupata dalla variabile `rec`, e torna a chiedere un altro nome. Mostriamo una sessione di esempio.

```
> writear archivio
Name (o fine): andrea
Age: 23
Name (o fine): marco
Age: 40
Name (o fine): luigi
Age: 12
Name (o fine): fine
> od -Ad -c archivio
0000000   a   n   d   r   e   a  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
```

```
0000016  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0000128 027  \0  \0  \0   m   a   r   c   o  \0  \0  \0  \0  \0  \0  \0
0000144  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0000256  \0  \0  \0  \0   (  \0  \0  \0   l   u   i   g   i  \0  \0  \0
0000272  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
```

4 Si scriva un programma che implementi il comando `readar`, il quale invocato con `readar nomefile` chiede all'operatore una posizione `n` di record, quindi legge l'`n`-esimo nome ed età memorizzati in `nomefile` mediante il comando `writear`. Mostriamo una sessione di esempio.

```
> readar archivio
Posizione (o fine): 1
rec.name=marco  rec.age=40
Posizione (o fine): 2
rec.name=luigi  rec.age=12
Posizione (o fine): 0
rec.name=andrea rec.age=23
Posizione (o fine): 3
posizione non esistente
Posizione (o fine): fine
```

5 Quali cambiamenti occorrerebbe effettuare nei programmi dei due esercizi precedenti se il campo `name` di `RECORD` fosse di tipo `char *`, ovvero una stringa allocata dinamicamente?

6 Si scriva un programma che implementi il comando `tailn`, il quale invocato con `tailn filename n` stampa le ultime `n` linee del file `filename`.

9.8 Informazioni sui file: `stat`

[Glass, 411-412]

Nella sezione 9.1.1 abbiamo richiamato i diversi attributi di un file: tipo (regolare, directory, etc.), diritti di accesso (proprietario, gruppo e altri), numero di hard link, lunghezza in bytes, ecc. È possibile accedere a tutte queste informazioni mediante la chiamata di sistema:

```
int stat(const char *file_name, struct stat *buf);
```

la quale richiede di specificare:

`file_name` il nome del file (assoluto o relativo),

`buf` un puntatore ad una struttura di tipo `struct stat` in cui la funzione scrive le informazioni sul file richiesto.

La chiamata `int fstat(int filedes, struct stat *buf);` differisce da `stat` solo in quanto richiede un descrittore di file (aperto) invece del nome del file. Entrambe ritornano `0` in caso di successo e `-1` in caso di errore.

La struttura `struct stat` contiene i seguenti campi:

```
struct stat
{
...
 ino_t         st_ino;      /* inodo */
 mode_t        st_mode;     /* diritti di protezione */
 nlink_t       st_nlink;    /* numero di hard links */
 uid_t         st_uid;      /* ID utente del proprietario */
 off_t         st_size;     /* lunghezza totale, in bytes */
 unsigned long st_blksize;  /* dimensione blocco del filesystem */
 unsigned long st_blocks;   /* numero di blocchi da 512 bytes
                               occupati dal file */
 time_t        st_atime;    /* ultimo accesso */
 time_t        st_mtime;    /* ultima modifica file */
 time_t        st_ctime;    /* ultimo cambiamento dati */
...
};
```

i quali assumono il seguente significato:

`st_ino` è il numero di inodo del file richiesto (vedi sezione 9.1.2 per il concetto di inodo),

`st_mode` è un intero che codifica il tipo di file ed i diritti di accesso. È possibile testare tali informazioni con delle apposite macro:

```
struct stat info;

IFERROR(stat("dati.txt", &info), "Nella stat");

if( S_ISLNK(info.st_mode) ) { /* link simbolico */ }
if( S_ISREG(info.st_mode) ) { /* file regolare  */ }
if( S_ISDIR(info.st_mode) ) { /* directory  */ }
if( S_ISCHR(info.st_mode) ) { /* speciale a caratteri  */ }
if( S_ISBLK(info.st_mode) ) { /* speciale a blocchi  */ }

if( info.st_mode & S_IRUSR ) { /* diritto r per l'utente  */ }
if( info.st_mode & S_IWUSR ) { /* diritto w per l'utente  */ }
if( info.st_mode & S_IXUSR ) { /* diritto x per l'utente  */ }
```

`st_n_link` è il numero di hard link (repliche) del file,

`st_uid` è lo *user id* del proprietario, ovvero un intero che individua univocamente l'account proprietario del file,

`st_size` è la lunghezza in bytes del file,

`st_blksize` è la dimensione in bytes di un blocco sul disco (valore stabilito in fase di formattazione),

`st_blocks` [8] è il numero di blocchi da 512 bytes[9] che il file occupa sul disco. Dal momento che l'ultimo blocco può essere occupato solo parzialmente, in generale abbiamo `st_size` $\leq$ `st_blocks` $*512$.

`st_atime`, `st_mtime`, `st_ctime` codificano la data/ora di ultimo accesso, modifica del file e cambiamento (modifica o variazioni di proprietario o numero hard links) del file. In realtà, non sempre i sistemi Linux/Unix implementano tutte e tre le date, ma si limitano ad implementarne una sola e a ritornare solo quella nei campi `st_atime`, `st_mtime` e `st_ctime`. Il tempo è codificato (nel tipo `time_t`) in numero secondi trascorsi dal primo gennaio 1970. Per convertirlo in una stringa del tipo `Tue Mar 14 17:12:32 2000` si può usare la routine di libreria `char *ctime(const time_t *timep);` come nel seguente esempio:

```
#include <time.h>

...
struct stat info;

IFERROR( stat("dati.txt", &info), "Eseguendo la stat()");
WRITE( ctime( &info.st_atime ) );
...
```

```
STAT(2)

NAME
       stat, fstat - informazioni sui file

SYNOPSIS
       #include <sys/stat.h>
       #include <unistd.h>

       int stat(const char *file_name, struct stat *buf);
       int fstat(int filedes, struct stat *buf);
```

---

[8] Si noti anche che `st_blocks` e `st_blksize` non sono previsti dallo standard POSIX, per cui la loro interpretazione può variare da sistema a sistema

[9] Si noti che il numero di blocchi riportato da `ls -la` si riferisce normalmente a blocchi di 1024 bytes.

9.9 Creazione e cancellazione di file: `creat` e `unlink`

Abbiamo visto in precedenza come la `open`, con opportuni parametri, possa anche creare un nuovo file. A questo scopo è però disponibile anche una interfaccia più diretta, costituita dalla funzione `creat`:

```
CREAT(2)

NAME
       creat - create a file

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int creat(const char *pathname, mode_t mode);
```

I parametri di `creat` sono identici a quelli di `open`, eccezion fatta per i flag, che in questo caso non sono presenti (l'operazione richiesta è sempre `O_CREAT`). Anche il valore di ritorno è analogo a quello di `open`: `creat` restituisce il descrittore del file appena creato, oppure -1 in caso di errore.

È interessante notare che la `creat` (come già la `write`) è un'operazione *atomica*: non è possibile che due processi distinti creino lo stesso file nello stesso momento. Per questa sua caratteristica, la `creat` è a volte usata per implementare dei *semafori* con cui sincronizzare processi distinti.

L'operazione duale — la cancellazione di un file esistente — è svolta dalla funzione `unlink`:

```
UNLINK(2)

NAME
       unlink - delete a name and possibly the file it refers to

SYNOPSIS
       #include <unistd.h>

       int unlink(const char *pathname);
```

La funzione richiede che venga passata una stringa, contenente il pathname del file da cancellare. Se il file esiste, viene cancellato dalla directory in cui risiede. Se il file aveva un

solo link, e al momento della `unlink` non era aperto dallo stesso o da un altro programma, viene anche cancellato il contenuto, e liberato lo spazio disco corrispondente. Altrimenti, a seconda dei casi, viene decrementato il numero dei link, oppure la cancellazione del contenuto viene rimandata a quando l'ultimo programma ad accedere al file avrà chiuso il descrittore corrispondente.

9.10 Esempi ed esercizi

9.10.1 Esempio: `filetype`

Il seguente programma `filetype`, invocato con `filetype file1 .. filen` scrive sullo standard output il tipo ed i diritti di ciascun file `file1`, ..., `filen`. Il corpo del programma è un ciclo `for` che, per ciascun argomento, chiama una funzione per la stampa del nome e del tipo di file. Si noti come il caso di fallimento della `stat` sia trattato con `IFERROR3` al fine di stampare un messaggio di errore e quindi, invece di terminare il programma, ritornare al ciclo `for` per trattare il successivo argomento.

```
/* File:   filetype.c
   Specifica: stampa il tipo dei nomi di file passati come parametro
*/

/* include per stat */
#include <sys/stat.h>

#include "sysmacro.h"

void printtype(char *);

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

  /* per ciascun argomento */
  for (i=1; i<argc; i++)
    printtype(argv[i]); // stampa nome e tipo

  return(0);
}


void printtype(char * filename)
{
```

```
  struct stat info;

  /* se non riesco ad aprile il file, scrivo un
     messaggio e ritorno al chiamante */
  IFERROR3(stat(filename, &info), filename, return);

  WRITE(filename);
  WRITE(": ");
  if( S_ISLNK(info.st_mode) ) { WRITE("symbolic link\n"); }
  else if( S_ISREG(info.st_mode) ) { WRITE("regular\n"); }
  else if( S_ISDIR(info.st_mode) ) { WRITE("directory\n"); }
  else if( S_ISCHR(info.st_mode) ) { WRITE("special character\n"); }
  else if( S_ISBLK(info.st_mode) ) { WRITE("special block\n"); }
  else { WRITE("unknown type\n"); }
}
```

Esercizi

1 Si scriva un programma C, che implementi il comando `infostat`, il quale invocato con `infostat numero nome-file` fornisce informazioni sul file `nome-file`, e quindi ogni 5 secondi (fino a un massimo di `numero` volte) verifica se la data-orario di ultima modifica sono cambiate e, se lo sono, presenta le informazioni aggiornate.

Suggerimento: per attendere 5 secondi si usi la routine di libreria `sleep` (vedi `man 3 sleep`). `sleep(5)` sospende un programma in esecuzione riattivandolo dopo `5` secondi.

Il seguente esempio fa vedere l'esecuzione di `infostat 10 pippo` inizialmente di tipo regolare ed eseguibile, che nel frattempo (cioé in un'altra finestra) viene cancellato, poi ricreato come regolare non eseguibile, poi ricancellato, e infine ricreato come directory.

```
> infostat 10 pippo
Info on file pippo:
        type:            regular
        executable:      Yes
        inode:           675877
        hard links:      1
        size:            269
        st_blksize:      4096
        st_blocks x 512: 1024
        access time:     Tue Feb 20 19:48:10 2001
        modific time:    Tue Feb 20 19:48:10 2001
        change time:     Tue Feb 20 19:48:10 2001
        Diritti:         rwx------
pippo: No such file or directory
Info on file pippo:
```

```
        type:            regular
        executable:      No
        inode:           167969
        hard links:      1
        size:            114
        st_blksize:      4096
        st_blocks x 512: 1024
        access time:     Tue Feb 20 19:49:29 2001
        modific time:    Tue Feb 20 19:49:29 2001
        change time:     Tue Feb 20 19:49:29 2001
        Diritti:         rw-r--r--
pippo: No such file or directory
Info on file pippo:
        type:            directory
        executable:      Yes
        inode:           874534
        hard links:      2
        size:            1024
        st_blksize:      4096
        st_blocks x 512: 1024
        access time:     Tue Feb 20 19:49:38 2001
        modific time:    Tue Feb 20 19:49:38 2001
        change time:     Tue Feb 20 19:49:38 2001
        Diritti:         rwxr-xr-x
```

2 Scrivere una funzione `int isdirectory(const char *filename);` che ritorni `1` se `filename` è una directory e `0` in tutti gli altri casi (cioè non è una directory, è un altro tipo di file o non è un file, o non esiste).

3 Si scriva un programma che crei un file di nome `lock` in `/tmp`, attenda per 1 secondo, lo cancelli, attenda 5 secondi, e torni a crearlo, ripetendo il processo finche' non si verifica un errore — in questo caso, il programma deve stampare l'errore e terminare.

Si eseguano quindi in contemporanea due o più istanze di questo programma, usando l'operatore `&` della shell. Cosa accade?

9.11 Mappaggio dei file in memoria

Una modalità alternativa all'uso di `read` e `write` per leggere e scrivere su file consiste nel *mappare* un file in memoria, rendendo disponibile il contenuto del file come se si trattasse di strutture dati in memoria.

In questa sezione spieghiamo brevemente in cosa consiste il mappaggio dei file in memoria e descriviamo le chiamate di sistema che implementano questa operazione: (`mmap()` e `munmap()`).

9.11.1 Mappaggio dei file in memoria

Ogni processo Unix ha uno spazio degli indirizzi formato da tre segmenti: il testo, i dati (inizialmente distinti in dati inizializzati e dati non inizializzati, ma poi fusi durante l'esecuzione) e lo stack (vedi Capitolo 7). La Figura 9.1 mostra un esempio di come i tre segmenti relativi a due processi attivati a partire dallo stesso esguibile vengono allocati nella memoria fisica disponobile. Inoltre, la gestione della memoria si basa sulla paginazione, cosicché tutti i segmenti logici vengono in realtà paginati e caricati in aree diverse delle memoria fisica. I segmenti di testo non possono essere modificati e le aree di memoria fisica che li contengono vengono condivise fra i vari processi attivati utilizzando il medesimo eseguibile. I segmenti dati e stack sono invece modificabili, e quindi ogni processo ne possiede una copia privata allocata in un'area distinta della memoria fisica.

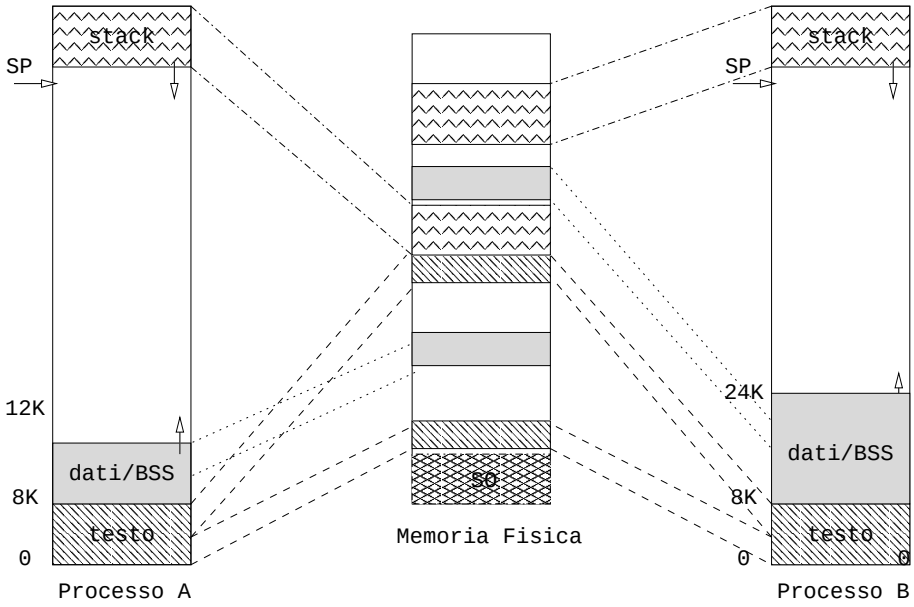


Figura 9.1: Spazio di indirizzamento di due processi e loro allocazione nella memoria fisica.

Lo standard POSIX prevede chiamate di sistema che realizzano i file mappati in memoria. In questo caso, viene creata una corrispondenza fra una porzione dello spazio di indirizzamento di un processo (Fig. 9.2) ed il contenuto del file, in modo da poter leggere e scrivere i byte del file come se fossero un array di byte in memoria, senza bisogno di usare `read()` o `write()`. Come illustrato dalla Figura 9.2, due processi diversi possono mappare lo stesso file in memoria a partire da indirizzi logici diversi. In questo modo, i due processi condividono le pagine fisiche su cui è stato allocato il file. Il file è condiviso a tutti gli effetti e le scritture di un processo sul file sono immediatamente visibili dall'altro. In effetti, mappando un file temporaneo nella memoria di due o più processi è possibile condividere parte dello spazio di indirizzamento durante l'esecuzione.

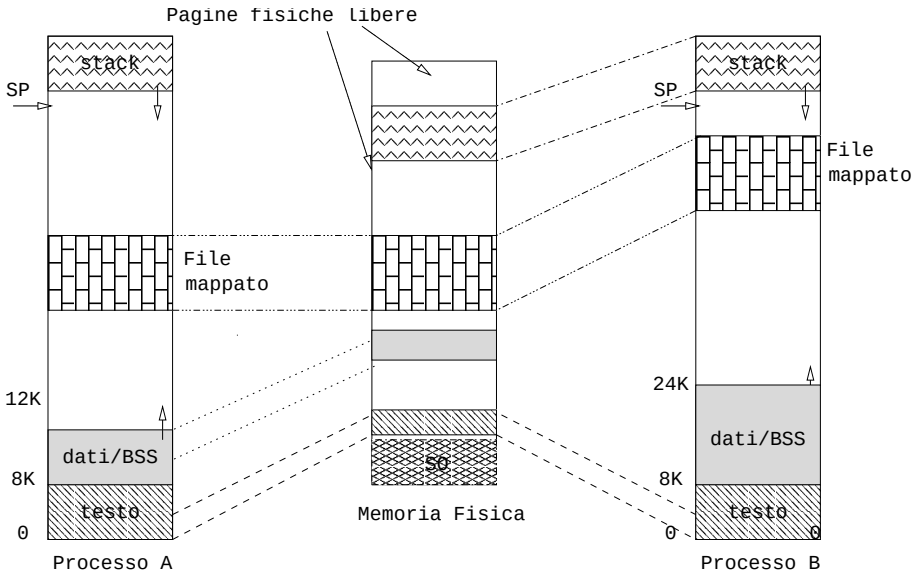


Figura 9.2: Due processi condividono lo stesso file mappato in memoria.

9.11.2 Mappare un file in memoria: `mmap`, `munmap`

```
MMAP(2)

NAME
       mmap, munmap - map or unmap files or devices into memory

SYNOPSIS
      #include <unistd.h>
      #include <sys/mman.h>

      #ifdef _POSIX_MAPPED_FILES
      void * mmap(void *start, size_t length, int prot , int flags, \
                int fd, off_t offset);
      int munmap(void *start, size_t length);
      #endif
```

La funzione `mmap` chiede di mappare `lenght` byte a partire dall'offset `offset` nel file specificato dal descrittore `fd` nello spazio di indirizzamento del processo, preferibilemente a partire dall'indirizzo `start`. L'indirizzo `start` è un suggerimento da parte del programmatore e tipicamente non viene specificato (ovvero, viene passato `NULL`). L'indirizzo logico in cui il file viene realmente mappato è il valore di ritorno della `mmap`. La `mmap()` ritorna `MAP_FAILED` nel caso non riesca a mappare il file e setta il valore di `errno` opportunamente.

`prot` è una maschera di bit che descrive il tipo di protezione dell'area mappata (la prote-

zione richiesta deve essere consistente con quella richiesta quando il file è stato aperto). La protezione desiderata si può specificare mettendo in OR dei flag predefiniti (per esempio, `PROT_WRITE` specifica il permesso di scrittura, `PROT_READ` specifica il permesso di lettura, etc.). Il parametro `flags` invece è una maschera di bit che specifica se l'area di memoria su cui è mappato il file può essere condivisa, e con quali modalità. Anche in questo caso il tipo di comportamento desiderato può essere definito mettendo in OR alcune maschere predefinite. Ad esempio, `MAP_SHARED` permette di condividere il mapping con tutti gli altri processi che mappano lo steso oggetto. `MAP_PRIVATE` crea una copia privata del processo, e le scritture non vanno a modificare il file originale.

Con la `mmap()` si possono mappare solo parti di un file che sono multipli dell'ampiezza di pagina. Per conoscere l'ampiezza della pagina nel sistema è possibile chiamare la funzione `getpagesize()` che restituisce l'ampiezza di pagina in byte. Sia `length` che `offset` devono essere allineati a questa ampiezza.

```
GETPAGESIZE(2)

NAME
       getpagesize - get memory page size

SYNOPSIS
       #include <unistd.h>
       int getpagesize(void);
```

La `munmap()` elimina il mapping precedentemente stabilito per l'indirizzo logico `start` per un'ampiezza di `length` byte (anche questo deve essere un multiplo dell'ampiezza di pagina). La `munmap()` ritorna 0 in caso di successo e -1 in caso di fallimento, settando il valore di `errno` opportunamente. Tutti i mapping definiti vengono automaticamente eliminati quando un processo termina. Invece la chiusura di un file non elimina il suo eventuale mappaggio in memoria che deve essere esplicitamente cancellato con una `munmap()`.

Il seguente programma, mappa in memoria il proprio sorgente C e lo stampa sullo standard output dopo aver modificato il commento iniziale.

```
/*
   File: es-mmap.c
   Specifica: esempio di uso di mmap, munmap
*/

/* include per la mmap, munmap*/
#include <sys/mman.h>

/* include per le chiamate sui file */
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>

#include "sysmacro.h"

int main (void) {
    int fd;
    int i;
    char * file;
    int bytes_in_page;

    /* ottengo l'ampiezza di pagina */
    bytes_in_page = getpagesize();
    printf("L'ampiezza di pagina e' %d bytes\n", bytes_in_page);

    /* apertura del file da mappare */
    IFERROR(fd = open("./es-mmap.c",O_RDWR),"aprendo es-mmap.c");

    /* mapping del file */
    IFERROR((file = mmap (NULL,bytes_in_page,PROT_READ | PROT_WRITE, \
                  MAP_SHARED,fd,0))  ==MAP_FAILED,"mappando es-mmap.c");

    /* chiudo il file */
    IFERROR(close(fd),"chiudendo es-mmap.c");

    /* modifico il commento ad inizio file */
    strcpy(file+4,"$$ modifica     $$");

    /* stampo il file sullo standard output */
    for(i=0; (i< bytes_in_page) && (file[i] != EOF); i++)
        putchar(file[i]);

    putchar('\n');

    /* elimino il mappaggio del file */
    IFERROR(munmap (file,bytes_in_page), "eliminando il mapping es-map.c");

    return 0;
}
```

Capitolo 10

Gestione delle directory

Nello standard POSIX, la chiamata di sistema per l'accesso all'elenco dei file contenuti in una directory è la `getdents`, come descritta in [**Glass, 412-413**]. I sistemi Linux offrono tale chiamata ma con qualche deviazione dallo standard POSIX. Di seguito presentiamo non `getdents`, ma piuttosto delle routine di libreria[1] conformi allo standard POSIX. Essendo routine e non chiamate, sono descritte nella sezione 3 del manuale in linea.

10.1 Funzioni di utilità e librerie

10.1.1 Funzioni di utilità

Prima di presentare le chiamate per l'accesso alle directory, mostriamo alcune funzioni di uso comune, che ci serviranno nel seguito del capitolo.

```
/* File:   util.h
   Specifica: prototipi funzioni di utilita'
*/

int isdirectory(const char *);
char * concatena(const char *, const char *);
char * concatena3(const char *, const char *, const char *);
char ** split_arg(char *, char *, int *);
void free_arg(char **);
```

Il file `util.h` contiene i prototipi delle funzioni:

[1] Più precisamente sono funzioni di libreria (sezione 3 del manuale) `opendir`, `closedir`, `readdir`, `rewinddir`, e `getcwd`. Sono chiamate di sistema (sezione 2 del manuale) la `chdir` e la `fchdir`. ATTENZIONE: esiste anche una chiamata di sistema (quindi nella sezione 2 del manuale) `readdir`!

`int isdirectory(const char *);` ritorna `1` se il pathname passato come parametro è una directory, e `0` altrimenti;

`char * concatena(const char *, const char *);` ritorna una stringa (allocando lo spazio necessario) ottenuta concatenando le due stringhe passate come parametri; ad esempio, `concatena(Lab, oratorio)` ritorna un puntatore ad una stringa allocata dinamicamente `Laboratorio`;

`char * concatena3(const char *, const char *, const char *);` ritorna una stringa (allocando lo spazio necessario) ottenuta concatenando le tre stringhe passate come parametri; ad esempio, `concatena(Lab, oratorio, IV)` ritorna un puntatore ad una stringa allocata dinamicamente `Laboratorio IV`;

`char ** split_arg(char *, char *, int *);` questa funzione invocata ad esempio con:

```
char **argv;

argv = split_arg("ls -l -a *", " ", &argc)
```

estrae i token della stringa `ls -l -a *` divisi da un qualsiasi carattere in (qui, solo spazio) ritornando un puntatore ad un vettore di stringhe allocate dinamicamente. `argv[0]` punterà a `ls`, `argv[1]` a `-l`, `argv[2]` a `-a`, `argv[3]` a `*` e `argv[4]` a `NULL`. Infine, `split_arg` scriverà in `argc` il numero di token trovati, ovvero `4`.

`void free_arg(char **);` disalloca il vettore di stringhe denotato dall'argomento (che deve essere un puntatore ritornato da `split_arg`). Ogni `split_arg` deve avere una corrispondente `free_arg`, a meno che non si desideri lasciare allocata la memoria per i token fino alla terminazione del programma.

La definizione delle funzioni è contenuta nel file `util.c` (disponibile nei file sorgenti allegati a questa dispensa – vedi Sezione 8.6). Si consiglia al lettore di provare a scrivere la definizione delle funzioni descritte sopra, confrontando poi il risultato con il file `util.c`

10.1.2 Librerie e make

[Glass, 354-358]

Nei sorgenti allegati (vedi Sezione 8.6), è proposto un esempio d'uso delle funzioni di utilità nella directory `Cap10/Util`. Il makefile dell'esempio crea una *libreria* contenente le funzioni di utilità.

La compilazione separata di vari file C componenti un programma evita di dover ricompilare se non è cambiato. Per quei file contenenti funzioni (in genere di utilità) usate in più programmi distinti, è utile, invece di replicare i file in ciascun programma, raccogliere i file (o meglio

i file oggetto) in un archivio, detto *libreria*. Le librerie hanno, in genere, estensione `.a`. Il comando `ar r nomelibreria.a fileoggetto1 ... fileoggetton` inserisce/aggiorna i file oggetto `fileoggetto1 ... fileoggetton` nella libreria `nomelibreria.a`, eventualmente creandola. Ad esempio:

```
> gcc -Wall -g -c util.c -o util.o
> ar r libutil.a util.o
> mv libutil.a ~/lib
```

compila `util.c` nel file oggetto `util.o`, ed inserisce questo nella libreria `libutil.a`, quindi sposta la libreria nella directory $\sim$`/lib`. Le fasi di compilazione e creazione della libreria sono automatizzate con il comando `make -f makefilelib install`, dove il file `makefilelib` è riportato di seguito.

```
# makefile per la creazione di librerie
# invocare con > make -f makefilelib

CC = gcc
CFLAGS = -Wall -g

LIBHOME = $(HOME)/lib/        # directory con le nostre librerie
LIBNAME = libutil.a           # librerie

# dipendenze per la libreria
LIBOBJS = util.o
$(LIBNAME):  $(LIBOBJS)
        ar r $(LIBNAME) $(LIBOBJS)

# dipendenze oggetto: header

util.o: util.h

# installa la libreria in $(LIBHOME)
install:        $(LIBNAME)
        mv $(LIBNAME) $(LIBHOME)
```

Per vedere l'indice dei simboli definiti nella libreria `libutil.a` si può usare il comando `nm libutil.a`. Ad esempio:

```
> nm ~/lib/libutil.a

util.o:
000000c8 T concatena
00000040 T concatena3
```

```
00000000 T isdirectory
         U malloc
         U stat
         U strcat
         U strcpy
         U strlen
```

Nell'output di `nm` i simboli etichettati da `T` sono le funzioni definite in `util.o` (`T` sta per "text", ovvero simboli definiti nel segmento di codice del file oggetto), mentre i simboli etichettati da `U` sono usati ma definiti altrove (`U` sta per "undefined", ovvero non definito nel file oggetto).

Dal momento che una libreria è essenzialmente un archivio di file oggetto, è possibile utilizzarla nella compilazione dei programmi. Ad esempio:

```
> gcc -Wall -g -c esempio.c -o esempio.o
> gcc -L~/lib -lutil esempio.o % ~/libutil.a
```

compila `esempio.c` nel file oggetto `esempio.o`, quindi effettua il linking con `~/libutil.a`. L'opzione `-L~/lib` specifica in quale directory cercare la libreria, mentre l'opzione `-lutil` specifica che la libreria si chiama `libutil.a` – si noti come il prefisso `lib` e il suffisso `.a` siano impliciti. Possiamo automatizzare il tutto definendo il `makefile` come segue.

```
# makefile con uso di librerie

CC = gcc
CFLAGS = -Wall -g

LIBHOME = $(HOME)/lib/        # directory con le nostre librerie
LIBNAME = libutil.a           # librerie
LDFLAGS = -L$(LIBHOME) -lutil # opzioni per il linker

# dipendenze eseguibile: oggetti e/o librerie

esempio: esempio.o $(LIBHOME)$(LIBNAME)

# dipendenze oggetto: header

esempio.o: sysmacro.h

clean:
        rm -f *~ core

cleanall:
        rm -f *.o *.a *~ core
```

10.2 Apertura e chiusura: `opendir` e `closedir`

Nelle routine di gestione delle directory, una directory è individuata da un puntatore (handle) ad una struttura di tipo `DIR` (analogamente alle routine ANSI C per la gestione dei file che usano un puntatore ad una struttura di tipo FILE). La chiamata:

```
DIR *opendir(const char *name);
```

richiede di specificare il pathname (assoluto o relativo) `name` di una directory a cui si intende accedere. L'accesso alla directory aperta avviene solo in modalità di lettura. Il valore di ritorno è un puntatore ad una struttura `DIR`, utilizzata poi nelle chiamate di lettura e chiusura. In caso di errore, il puntatore restituito è `NULL`. Quindi uno schema tipico di apertura di una directory è il seguente.

```
DIR * dir;
...
if( (dir = opendir("nomedirectory")) == NULL) {
  perror("messaggio");
  exit(errno); // o altra azione
}
```

```
OPENDIR(3)

NAME
       opendir - apertura di una directory

SYNOPSIS
       #include <sys/types.h>
       #include <dirent.h>

       DIR *opendir(const char *name);
```

La chiamata:

```
int closedir(DIR *dir);
```

chiude la directory aperta precedentemente ed individuata dall'handle `dir`. La funzione `closedir` ritorna `-1` in caso di errore (impostando `errno` con il codice di errore corrispondente). Per il controllo del risultato di `closedir` è quindi possibile utilizzare la macro `IFERROR` – differentemente da `opendir`.

```
CLOSEDIR(3)

NAME
       closedir - chiusura di una directory

SYNOPSIS

       #include <sys/types.h>
       #include <dirent.h>

       int closedir(DIR *dir);
```

10.3 Lettura: `readdir`

Leggere il contenuto di una directory significa leggere l'elenco dei file contenuti in essa. La lettura dell'elenco avviene un file alla volta. Alla prima lettura vengono restituite informazioni sul primo file, alla seconda sul secondo file, e così via. Ad ogni lettura viene restituito un puntatore ad una struttura contenente informazioni sul file corrente. Alla lettura successiva all'ultimo file, viene restituito un puntatore a `NULL`, il quale codifica che abbiamo terminato di scorrere l'elenco. La chiamata:

```
struct dirent *readdir(DIR *dir);
```

richiede un handle `dir` e ritorna un puntatore ad una struttura `struct dirent` nella quale sono disponibili le informazioni sul file corrente:

```
struct dirent
{
  ...
  long d_ino;                 /* numero di i-nodo */
  unsigned short d_reclen;    /* lunghezza di d_name */
  char d_name [NAME_MAX+1];   /* nome del file */
}
```

L'uso tipico della `readdir` è quindi il seguente:

```
struct dirent * fileinfo;
...
while((fileinfo = readdir(dir))!=NULL) {
  // uso di fileinfo->d_name
  ...
}
```

```
READDIR(3) <- attenzione esiste anche READDIR(2)

NAME
       readdir - legge i nomi dei file in una directory

SYNOPSIS

       #include <sys/types.h>
       #include <dirent.h>

       struct dirent *readdir(DIR *dir);
```

10.4 Riposizionamento: `rewinddir`

Alla lettura successiva all'ultimo file, la `readdir` restituisce un puntatore a `NULL`. Come fare nel caso si voglia scorrere nuovamente l'elenco? La chiamata:

```
void rewinddir(DIR *dir);
```

riporta la posizione corrente di lettura all'inizio dell'elenco. Si noti che la chiamata non ha alcun valore di ritorno.

```
REWINDDIR(3)

NAME
       rewinddir - resetta la posizione di lettura

SYNOPSIS

       #include <sys/types.h>
       #include <dirent.h>

       void rewinddir(DIR *dir);
```

10.5 Directory corrente: `chdir`, `fchdir`, `getcwd`

Ciascun programma in esecuzione (*processo*) ha associata una *directory corrente*. All'inizio, la directory corrente è quella in cui ci si trovava al momento di invocare il programma. Tutti i pathname relativi usati nelle chiamate di sistema sono relativi, appunto, alla directory corrente. È possibile cambiare la directory corrente mediante la chiamata:

```
int chdir(const char *path);
```

la quale accetta il pathname (relativo o assoluto) della nuova directory corrente. Ad esempio, `chdir(..)` assume come nuova directory corrente la directory padre della directory corrente attuale. La chiamata `chdir` ritorna `-1` in caso di errore (impostando `errno` con il codice di errore corrispondente). Una analoga chiamata:

```
int fchdir(int fd);
```

cambia la directory corrente in quella specificata da un descrittore di file aperto (con `open`).

```
CHDIR(2)

NAME
       chdir, fchdir  - cambia la directory corrente

SYNOPSIS

       #include <unistd.h>

       int chdir(const char *path);
       int fchdir(int fd);
```

Per conoscere la directory corrente, è disponibile la routine di libreria:

```
char *getcwd(char *buf, size_t size);
```

la quale scrive in `buf` la stringa con il nome della directory corrente. Il parametro `size` deve contenere la dimensione massima della stringa scrivibile in `buf`. Se la stringa con la directory corrente eccede tale lunghezza, `getcwd()` ritorna `NULL` altrimenti `buf`. Per stampare la directory corrente, si può usare il seguente frammento di programma:

```
char stringa[MAX];

if( getcwd(stringa, MAX) != NULL ) {
       WRITE("La directory corrente e': ");
       WRITE(stringa);
}
```

```
GETCWD(3)

NAME
```

```
       getcwd - nome della directory corrente

SYNOPSIS
       #include <unistd.h>

       char *getcwd(char *buf, size_t size);
```

10.6 Esempi ed esercizi

10.6.1 Esempio: `lsdir`

Riportiamo di seguito il programma `lsdir`, il quale implementa il comando `ls` senza argomenti, ovvero produce la lista dei file contenuti nella directory corrente. Per far questo, apre la directory corrente `.` e scorre la lista stampando per ciascun file il nome. Per le directory stampa anche il suffisso `/`. Infine, chiude la directory.

```
/* File:   lsdir.c
   Specifica: implementazione del comando -> ls senza argomenti
*/

/* include per routine su directory */
#include <sys/types.h>
#include <dirent.h>

/* include per chiamata stat */
#include <sys/stat.h>

#include "sysmacro.h"
#include "util.h"

int main()
{
  DIR *dir;
  struct dirent *file;

  if( (dir = opendir(".")) == NULL) {
    perror("directory corrente");
    exit(errno);
  }

  while((file = readdir(dir))!=NULL) {
    WRITE(file->d_name);
    if( isdirectory(file->d_name) )
```

```
      WRITE("/"); /* scrive "/" solo se e' una directory */
    WRITE("\t");
  }

  WRITE("\n");

  IFERROR(closedir(dir), "directory corrente");
  return(0);
}
```

Esercizi

1 Si modifichi `lsdir` in modo che stampi anche la lunghezza in bytes di ciascun file contenuto nella directory.

2 Si modifichi `lsdir` in modo che stampi insieme i file dello stesso tipo (es., prima tutti i file regolari, poi tutte le directory, ecc.).

3 Si scriva un programma `findfile.c` che quando invocato con `findfile ext directory` trovi tutti i file contenuti in `directory` che hanno come estensione `ext`.

4 Si scriva un programma `rdir.c` che quando invocato con `> rdir nome-1 nome-2 nome-3 ...` elenca i file contenuti nelle directory `nome-1`, `nome-2`, `nome-3` ... ed in tutte le sotto-directory in esse contenute.

Suggerimento 1: si definisca una funzione ricorsiva `void visit(const char *filename)` che effettua la visita della directory `filename`, facendo attenzione a non richiamarla sulle directory . (directory corrente) e .. (directory padre), pena un ciclo infinito.

Suggerimento 2: si sfrutti la funzione di utilità `concatena3` per concatenare pathnameattuale con / e con nomedirectory al fine di ottenere il nuovo pathname pathnameattuale/nomedirectory.

Suggerimento 3: in alternativa, si usi `chdir` per spostarsi fra le directory nel corso della visita. Quale versione è più semplice?

Capitolo 11

Gestione dei processi

11.1 Introduzione ai processi

[Glass, 422-436]

Un processo è definibile come una esecuzione di un programma. In particolare, in UNIX un processo è caratterizzato da quattro sezioni logiche:

- codice, in cui è contenuto il codice del processo;
- dati, in cui sono contenuti i dati statici;
- heap, in cui sono contenuti i dati allocati dinamicamente;
- stack, in cui sono contenuti i dati locali alle chiamate di funzioni.

Un processo è inoltre caratterizzato dai seguenti ID (numeri interi positivi) assegnati dal kernel (il "cuore" del sistema operativo) al processo all'atto della sua creazione:

- Process-ID (`PID`), identificativo unico del processo;
- Process-ID del processo padre (`PPID`), identificativo unico del processo padre;
- User-ID reale (real `UID`);
- User-ID effettivo;
- Process Group ID (`PGID`);
- User Group-ID reale (real `UGID`);
- User Group-ID effettivo.

Quando UNIX è avviato, opportuni meccanismi software e firmware caricano una copia del kernel in memoria centrale. Una volta in esecuzione il kernel provvede ad inizializzare le proprie strutture dati, ad effettuare alcuni controlli di sistema, e quindi crea il processo `init` che costituisce il processo "padre" del sistema in quanto tutti i processi successivi sono da esso generati attraverso un meccanismo di duplicazione che sarà discusso di seguito.

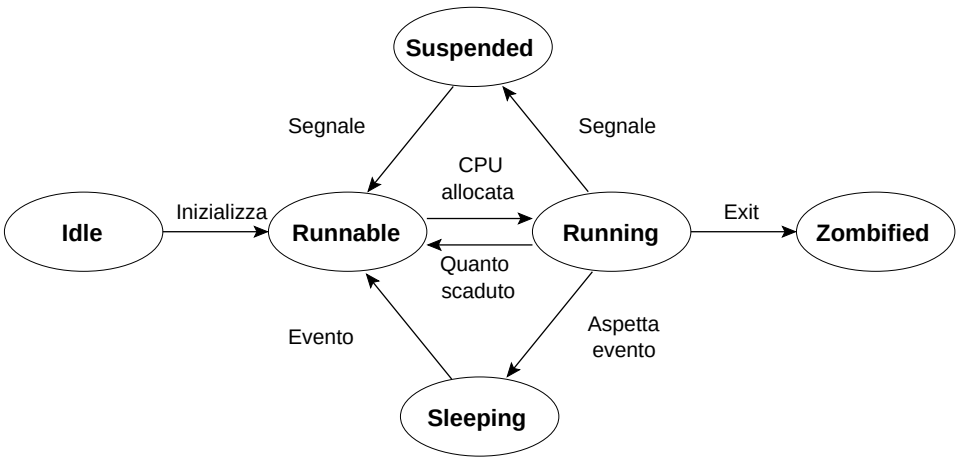
Le informazioni relative ai processi sono mantenute in una struttura dati del kernel detta *Tabella dei Processi*. Questa è costituita da un numero predefinito di locazioni (e tale numero determina il numero massimo di processi che possono essere simultaneamente in esecuzione in un dato istante). Quando un processo è creato, il kernel lo alloca in una locazione della *Tabella dei Processi* e lo dealloca quando il processo è distrutto.

Poiché il kernel è esso stesso un processo, la locazione 0 della *Tabella dei Processi* è ad esso riservata, mentre il processo `init` occuperà la locazione 1. L'identificativo della locazione è utilizzata quale identificatore (`PID`) del processo stesso.

Un processo può trovarsi in vari stati:

- `idle`, stato iniziale, tipico di un processo appena creato tramite una `fork()`;
- `runnable`, pronto per l'esecuzione e in attesa che la CPU sia disponibile;
- `running`, in esecuzione (occupa la CPU);
- `sleeping`, in attesa di un evento per riattivarsi, ad esempio se un processo esegue una `read()`, si addormenta fino a quando la richiesta di I/O non è completata;
- `suspended`, il processo è stato "congelato" (`frozen`) da un segnale, come ad esempio `SIGSTOP`; il processo è "scongelato" solo quando riceve il segnale `SIGCONT`;
- `zombified`, risorse rilasciate ma ancora presente nella *Tabella dei Processi*;

Di seguito sono riportate le possibili transizioni da uno stato all'altro



Lo stato di un processo può essere visualizzato tramite il comando `ps`. Ad esempio, utilizzando la flag `w` si ottiene il seguente risultato:

```
well20 ~> ps w
  PID TTY      STAT   TIME COMMAND
30142 pts/6    S      0:00 -bin/tcsh
30179 pts/6    R      0:00 ps w
```

dove lo stato è riportato sotto la colonna con titolo `STAT`. In questo caso sono visualizzati due processi appartenenti all'utente che ha eseguito il comando `ps w`: la shell, che si trova in stato `S` (*sleeping*), e il processo che esegue il comando stesso, che si trova in stato `R` (*running*). Per avere più infomazioni su come il comando `ps` visualizza le informazioni relative ad un processo si rimanda al manuale in linea (`man ps`).

Come anticipato, la creazione di un nuovo processo avviene attraverso un meccanismo di "duplicazione". Tale meccanismo è invocato da una chiamata di sistema (`fork()`) che causa la duplicazione del processo chiamante. Il processo che invoca la `fork()` è detto processo padre, mentre il processo generato (duplicato) è chiamato figlio. A parte alcune informazioni, fra cui il `PID` e `PPID`, il processo figlio è del tutto identico al processo padre: il codice, dati, heap e stack sono copiati dal padre. Inoltre, il processo figlio, continua ad eseguire il codice "ereditato" dal padre a partire dalla istruzione seguente alla chiamata della `fork()`.

Un processo può cambiare il codice che esegue attraverso la chiamata ad una delle procedure di sistema della famiglia `exec()`. Tale meccanismo è tipicamente utilizzato da un figlio per differenziarsi dal padre. Quando il processo figlio termina (ad esempio, in maniera volontaria tramite la chiamata `exit()`), attraverso il meccanismo delle interruzioni software (segnali), la sua terminazione viene comunicata al padre. Il padre, che solitamente si sospende (tramite la chiamata della `wait()`) in attesa della terminazione di uno dei suoi figli, quando riceve la segnalazione della terminazione del figlio si risveglia (permettendo la deallocazione del processo figlio dalla *tabella dei processi*).

11.2 Identificativo di processo: `getpid` e `getppid`

Un processo può conoscere il suo `PID` e quello del padre (`PPID`) attraverso le seguenti chiamate

---

```
GETPID(2)

NAME
       getpid, getppid - ottiene l'identificatore di un processo

SYNOPSIS
       #include <unistd.h>

       pid_t getpid(void);
       pid_t getppid(void);
```

`getpid()` restituisce il `PID` del processo chiamante, mentre `getppid()` restituisce il `PID` del processo padre (cioè il `PPID`). Il `PPID` del processo 1 (`init`) è 1. La chiamata ha sempre successo e `pid_t` è il tipo `int`.

11.3 Duplicazione di un processo: `fork`

Abbiamo già visto che un nuovo processo viene creato attraverso un meccanismo di "duplicazione" realizzato dalla chiamata di sistema `fork`. Ecco di seguito una sua descrizione più dettagliata

---

```
FORK(2)

NAME
       fork - crea un processo figlio

SYNOPSIS
       #include <unistd.h>

       pid_t fork(void);
```

`fork()` causa la duplicazione di un processo. Il processo figlio è una copia quasi identica al processo chiamante (padre). Esso eredita le sezioni codice, dati, heap e stack del padre. Inoltre eredita i descrittori di file aperti, e la tabella dei segnali (segnali in attesa). Tuttavia, il padre ed il figlio hanno `PID` e `PPID` diversi. Se la chiamata ha successo, `fork()` restituisce il `PID` del figlio al padre e `0` al figlio. Se fallisce, restituisce valore `-1` al padre e nessun processo figlio è creato. Notare che le variabili e puntatori del padre sono *duplicati e non condivisi* da padre a figlio. Al contrario, i file aperti del padre al momento della chiamata sono *condivisi*. In particolare,

- sono condivisi anche i puntatori ai file usati per `I/O` (ma vengono mantenute copie distinte per ogni processo);
- l' `I/O pointer` è modificato per entrambi i processi in seguito ad operazioni di lettura/scrittura da parte degli stessi.

Esempio di duplicazione di un processo

```
/*  File:   crea.c
    Specifica: esempio di creazione di un processo mediante fork()
*/

/* include per chiamate sui processi */
```

```
#include <unistd.h>

#include "sysmacro.h" /* macro di utilita' */

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

  IFERROR(pid = fork(), "generando il figlio");
  if( pid ) {
    /* siamo nel padre */
    printf("Processo padre(pid=%d): ho generato figlio (pid=%d)\n",
           getpid(),pid);
    sleep(2); /* il processo si sospende per 2 secondi */
  }
  else  {
    /* siamo nel figlio */
    printf("Processo figlio(pid=%d): sono generato dal processo (pid=%d)\n",
           getpid(),getppid());
  }

  /* padre e figlio */
  return(0);
}
```

Si noti come il codice sia lo stesso sia per il padre che per il figlio, tuttavia, a seconda del contenuto della variabile `pid`, istanziata in modo diverso dalla `fork()` a seconda che si tratti del processo figlio o del processo padre, i due processi eseguono parti diverse del codice: il padre segue il ramo `then`, mentre il figlio il ramo `else`, del test `if( pid )`.

Altra particolarità del codice è la auto-sospensione per due secondi del padre attraverso la chiamata `sleep(2)`. Al momento del suo risveglio, il processo figlio sarà terminato. Vedremo di seguito che il modo corretto per il padre di aspettare la terminazione del figlio è quello di utilizzare la chiamata di sistema `wait()`.

11.4 Terminazione esplicita di un processo: `exit`

Un processo può terminare in ogni momento attraverso la funzione di libreria `exit()`

```
EXIT(3)

NAME
       exit - causa la terminazione normale di un programma
```

```
SYNOPSIS
       #include <stdlib.h>

       void exit(int status);
```

`exit()` chiude tutti i descrittori di file del processo chiamante; dealloca il suo codice, dati, heap e stack, ed infine termina il processo chiamante. Inoltre invia il segnale di terminazione `SIGCHLD` al processo padre del processo chiamante ed attende che il codice di stato di terminazione (`status`) sia accettato. Solo gli 8 bit meno significativi di `status` sono utilizzati, quindi lo stato di terminazione è limitato ai valori nel range 0-255.

Nel caso in cui il processo padre sia già terminato, il kernel assicura che tutti i processi figlio diventino "orfani" e siano adottati da `init`, provvedendo a settare i `PPID` dei processi figli al valore `1` (cioè il `PID` di `init`). Ovviamente, `exit()` non ritorna alcun valore. Se eseguita in `main` è equivalente ad una `return()`.

11.5 Esempi

11.5.1 Condivisione dell'I/O

Abbiamo detto che i puntatori ai file usati per `I/O` sono condivisi dal padre e dal figlio. Nel seguente esempio, il padre apre un file (il suo stesso sorgente), quindi crea un figlio. Il figlio legge 100 bytes dal file (condiviso), quindi il padre legge anche lui 100 bytes. Dal momento che i puntatori al file sono condivisi, il figlio leggerà i primi 100 bytes del file ed il padre i secondi 100 bytes.

```
/*  File:   crea_io.c
    Specifica: esempio sulla condivisione dell'i/o dopo la fork()
*/

/* include per chiamate sui processi */
#include <unistd.h>

/* include per chiamate sui file */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#include "sysmacro.h" /* macro di utilita' */

int main(int argc, char * argv[])
{
  int fd;
  int pid, n;
```

```
  char buf[1000];

  IFERROR( fd = open("crea_io.c", O_RDONLY), "aprendo il file");
  IFERROR(pid = fork(), "generando il figlio");
  if( pid ) {
    /* siamo nel padre */
    sleep(2); /* il processo si sospende per 2 secondi */
    IFERROR( n = read(fd, buf, 100), "leggendo");
    IFERROR( write(1, buf, n), "scrivendo");
  }
  else  {

    /* siamo nel figlio */
    IFERROR( n = read(fd, buf, 100), "leggendo");
    IFERROR( write(1, buf, n), "scrivendo");
  }

  /* padre e figlio */
  return(0);
}
}
```

11.5.2 Adozione

Se il padre termina prima del figlio, senza aspettare la sua terminazione (usando la chiamata `wait`, che vedremo nel seguito di questo capitolo), il figlio rimane "orfano" e viene adottato dal processo `init` (che ha `PID 1`).

Tale adozione viene resa necessaria a causa del meccanismo di terminazione dei processi: il processo figlio non può essere distrutto completamente, cioè deallocato definitivamente dalla *Tabella dei Processi*, fino a quando il padre non ha ricevuto e servito il segnale (segnale `SIGCHLD`) che comunica la terminazione del figlio (si rimanda al Capitolo 12 per il trattamento dei segnali). Quindi, nel caso in cui la terminazione del padre avvenga prima della terminazione del figlio, ovviamente non è possibile per il padre ricevere e servire il segnale `SIGCHLD`. A questa situazione si pone rimedio facendo adottare il figlio dal processo "padre di tutti i processi" `init`. Il processo `init` raccoglie e serve tutti i segnali di tipo `SIGCHLD` dei processi adottati, permettendone la definitiva distruzione (deallocazione dalla *Tabella dei Processi*).

Vediamo di seguito un esempio di adozione.

```
/*  File:   orfano.c
    Specifica: esempio di creazione di un processo mediante fork(), con
           terminazione del padre prima del figlio
*/

/* include per chiamate sui processi */
```

```
#include <unistd.h>

#include "sysmacro.h" /* macro di utilita' */

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

  IFERROR(pid = fork(), "generando il figlio");
  if( pid ) {
    /* siamo nel padre */
    printf("Processo padre(pid=%d): ho generato figlio (pid=%d)\n",
           getpid(),pid);
  }
  else {
    /* siamo nel figlio */
    sleep(2); /* aspettiamo un po' ...
                 il padre dovrebbe nel frattempo terminare */
    printf("Processo figlio(pid=%d):  adottato dal processo (pid=%d)\n",
           getpid(),getppid()); /* a questo punto il figlio e'
                                   stato adottato da init */
  }

  /* padre e figlio */
  return(0);
}
```

11.5.3 Zombie

Uno stato particolare in cui un processo può venire a trovarsi è quello di `zombie`. Un processo entra in tale stato se termina, ma suo padre non serve il segnale `SIGCHLD` ad esso relativo. Si noti che questa situazione non avviene mai se il processo figlio è adottato in quanto il processo `init` accetta automaticamente il segnale. Tale situazione, invece, si determina quando il padre è comunque vivo, ma non esegue una chiamata di sistema `wait()` la quale è responsabile per la ricezione e il servizio del segnale `SIGCHLD`.

Un processo `zombie` non occupa nessuna risorsa del sistema tranne la propria locazione della *Tabella dei Processi*.

Vediamo di seguito un esempio di processo zombie.

```
/*  File:   zombie.c
    Specifica: esempio di creazione di un processo mediante fork(), con
           terminazione del figlio senza wait da parte del padre
*/
```

```
/* include per chiamate sui processi */
#include <unistd.h>

#include "sysmacro.h" /* macro di utilita' */

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

  IFERROR(pid = fork(), "generando il figlio");
  if( pid ) {
    /* siamo nel padre */
    printf("Processo padre(pid=%d): ho generato figlio (pid=%d)\n",
           getpid(),pid);
    sleep(1); /* aspettiamo un po' ...
                 il figlio dovrebbe nel frattempo terminare */

    IFERROR( system("ps -l"), "eseguendo ps");

  }
  else {
    /* siamo nel figlio */
    printf("Processo figlio(pid=%d): sono generato dal processo (pid=%d)\n",
           getpid(),getppid());
  }

  /* padre e figlio */
  return(0);
}
```

Si verifichi che l'output generato dal comando `ps -l`, invocato tramite la utility `system()` che sarà spiegata di seguito, identifichi lo stato del processo figlio attraverso la lettera `Z`, che sta ad indicare lo stato di `zombie`.

11.6 Attesa di terminazione: `wait` e `waitpid`

Vediamo adesso il modo corretto per il padre di attendere la terminazione di un figlio. L'attesa viene realizzata in modo passivo (attesa passiva) dalla chiamata di sistema `wait()` (e `waitpid()`)

```
WAIT(2)

NAME
       wait, waitpid - apetta la terminazione di un processo
```

```
SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>

       pid_t wait(int *status)
       pid_t waitpid(pid_t pid, int *status, int options);
```

`wait()` sospende un processo fino alla terminazione di uno qualunque dei suoi processi figli. In particolare, la chiamata attende la terminazione di un processo figlio e ne ritorna il `PID`, mentre nel parametro di uscita `status` vengono restituiti, attraverso una codifica "a byte", il motivo della terminazione e lo stato di uscita del processo che termina. Se al momento della chiamata esiste un processo figlio `zombie`, la chiamata serve immediatamente il corrispondente segnale e termina. La variabile `status` può essere valutata in modo semplice attraverso delle macro di cui riportiamo quelle più utilizzate (per maggiori dettagli sulle altre macro digitare da shell il comando `man 2 wait`):

- `WIFEXITED(status)`, viene valutata ad un valore diverso da `0` se il figlio è terminato normalmente;
- `WEXITSTATUS(status)`, restituisce il codice di ritorno codificato negli 8 bit meno significativi di `status`. Può essere utilizzata solo se `WIFEXITED(status)` ha restituito un valore diverso da `0`.

In caso di errore, invece del `PID`, viene ritornato il valore `-1`. Se il processo chiamante non ha figli, la chiamata ritorna immediatamente con valore `-1`.

`waitpid()` si comporta in modo simile a `wait()`, però permette di specializzare l'attesa. In particolare, il processo chiamante viene posto in attesa della terminazione del figlio con `PID pid`. Se il processo indicato da `pid` è uno `zombie`, la chiamata serve immediatamente il corrispondente segnale e termina. Il valore di `pid` può assumere i seguenti range:

$< $ `-1` prescrive l'attesa della terminazione di un qualunque processo figlio il cui `PGID` (identificativo di process group) è uguale al valore assoluto di `pid`;

`-1` prescrive l'attesa della terminazione di un qualunque processo figlio; il comportamento è quindi identico a quello della `wait()`;

`0` prescrive l'attesa della terminazione di un qualunque processo figlio il cui `PGID` (identificativo di process group) è uguale a quello del processo chiamante;

$>$ `0` prescrive l'attesa della terminazione del processo figlio il cui `PID` è uguale al valore di `pid`.

Il valore di `options` è il risultato dell'OR di zero o più delle seguenti costanti

- `WNOHANG` prescrive alla chiamata di ritornare immediatamente se nessun processo figlio è terminato (`wait()` non bloccante);

- `WUNTRACED` prescrive alla chiamata di ritornare anche nel caso in cui i processi figli sono in stato di stop, e il cui stato non è stato riportato.

`waitpid()` ritorna valore `-1` negli stessi casi di `wait()`. Inoltre, se c'è l'opzione `WNOHANG` e non ci sono figli terminati, ritorna `0`.

Esempio di attesa di terminazione

```
/*  File:   creaattesa.c
    Specifica: esempio di creazione di un processo e attesa terminazione
*/

/* include per chiamate sui processi */
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "sysmacro.h"

int main(int argc, char * argv[])
{
  int pid, pidw, status;

  IFERROR(pid = fork(), "generando il figlio");
  if( pid ) { /* siamo nel padre */
    printf("Processo padre(pid=%d): ho generato figlio (pid=%d)\n",
           getpid(),pid);
    pidw = wait(&status); /* attendo la terminazione del figlio */
    if(WIFEXITED(status))
      printf("Processo padre(pid=%d): processo %d terminato con exit(%d)\n",
             getpid(), pidw, WEXITSTATUS(status));
    else
      printf("Processo padre(pid=%d): processo %d terminato con segnale\n",
             getpid(),pidw);
  }
  else {
    /* siamo nel figlio */
    printf("Processo figlio(pid=%d): sono generato dal processo (pid=%d)\n",
           getpid(),getppid());
    sleep(1);
  }

  /* padre e figlio */
  return(0);
}
```

Si noti che nel caso in cui `WIFEXITED(status)` sia uguale a `0`, ciò significa che il processo figlio è terminato in modo anomalo (molto probabilmente a causa di un segnale che ne ha determinato la terminazione prematura) e quindi non ha avuto modo di riportare il proprio stato.

11.7 Esempi

Vediamo di seguito altri esempi di utilizzo di `wait()` e `waitpid()`.

11.7.1 Creazione di `n` processi

Vediamo prima un esempio dove il padre attende solo la terminazione di uno qualsiasi degli `n` processi figli.

```
/*  File:   crean1.c
    Specifica: esempio di creazione di n processi con attesa della
           terminazione di uno qualsiasi
*/

/* include per chiamate sui processi */
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "sysmacro.h"

int main(int argc, char * argv[])
{
  int pid, pidw, status, i, n;

  if(argc == 2)
    n = atoi(argv[1]);
  else
    n = 4; /* valore di default */

  /* genero n figli */
  for(i=0;i<n;i++) {
    IFERROR(pid = fork(), "generando un figlio");
    if( pid ) {
      /* siamo nel padre */
      printf("Processo padre(pid=%d): i=%d, processo figlio (pid=%d)\n",
             getpid(),i, pid);
    }
    else {
```

```
      /* siamo nel figlio */
      printf("Processo figlio(pid=%d): generato dal proc. (pid=%d) con i=%d\n",
             getpid(),getppid(),i);
      sleep(5 + i);
      exit(i);
    }
  }

  /* aspetto la terminazione di uno qualsiasi */

  pidw = wait(&status);
  if(WIFEXITED(status))
    printf("Processo %d terminato con exit(%d)\n",
           pidw,WEXITSTATUS(status));
  else
    printf("Processo %d terminato con segnale.\n",
           pidw);

  return(0);
}
```

Di seguito viene mostrata una versione dell'esempio precedente dove il padre attende la terminazione di tutti i suoi figli.

```
/*  File:   crean2.c
    Specifica: esempio di creazione di n processi con attesa della
           loro terminazione
*/

/* include per chiamate sui processi */
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "sysmacro.h"

int main(int argc, char * argv[])
{
  int pid, pidw, status, i, n;

  if(argc == 2)
    n = atoi(argv[1]);
  else
    n = 4; /* valore di default */
```

```
  /* genero n figli */
  for(i=0;i<n;i++) {
    IFERROR(pid = fork(), "generando un figlio");
    if( pid ) {
      /* siamo nel padre */
      printf("Processo padre(pid=%d): i=%d, processo figlio (pid=%d)\n",
             getpid(),i, pid);
    }
    else {
      /* siamo nel figlio */
      printf("Processo figlio(pid=%d): generato dal proc. (pid=%d) con i=%d\n",
             getpid(),getppid(),i);
      sleep(n - i);
      exit(i);
    }
  }

  /* aspetto la terminazione dei figli */
  for(i=0;i<n;i++) {
    pidw = wait(&status);
    if(WIFEXITED(status))
      printf("Processo %d terminato con exit(%d)\n",
             pidw,WEXITSTATUS(status));
    else
      printf("Processo %d terminato con segnale.\n",
             pidw);
  }

  return(0);
}
```

Infine, viene considerato il caso in cui il padre attende solo la terminazione dell'ultimo figlio generato.

```
/*  File:   crean3.c
    Specifica: esempio di creazione di n processi con attesa della
           terminazione solo dell'ultimo
*/

/* include per chiamate sui processi */
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "sysmacro.h"
```

```
int main(int argc, char * argv[])
{
  int pid, pidw, status, i, n, ultimo;

  if(argc == 2)
    n = atoi(argv[1]);
  else
    n = 4; /* valore di default */

  /* genero n figli */
  for(i=0;i<n;i++) {
    IFERROR(pid = fork(), "generando un figlio");
    if( pid ) {
      /* siamo nel padre */
      printf("Processo padre(pid=%d): i=%d, processo figlio (pid=%d)\n",
             getpid(),i, pid);
      ultimo = pid;
    }
    else {
      /* siamo nel figlio */
      printf("Processo figlio(pid=%d): generato dal proc. (pid=%d) con i=%d\n",
             getpid(),getppid(),i);
      sleep(5 + i);
      exit(i);
    }
  }

  /* aspetto la terminazione dell'ultimo dei figli */

  pidw = waitpid(ultimo, &status, 0);
  if(WIFEXITED(status))
    printf("Processo %d terminato con exit(%d)\n",
           pidw,WEXITSTATUS(status));
  else
    printf("Processo %d terminato con segnale.\n",
           pidw);

  return(0);
}
```

11.7.2 Recupero dello stato di terminazione

Di seguito si esemplifica l'uso delle macro per la `waitpid` e la gestione dello stato di terminazione di un processo.

```
/*  File:   creawaitpid.c
    Specifica: esempio di creazione di un processo e riporto dello stato
           (terminato o running)
*/

/* include per chiamate sui processi */
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "sysmacro.h"

int main(int argc, char * argv[])
{
  int pid, pidw, status;

  IFERROR(pid = fork(), "generando il figlio");
  if( pid ) {
    /* siamo nel padre */
    printf("Processo padre(pid=%d): ho generato figlio (pid=%d)\n",
           getpid(),pid);

    do {
      sleep(1);
      /* verifico lo stato del figlio */
      IFERROR(pidw = waitpid(pid, &status, WNOHANG),"waitpid");
      if( pidw > 0) {
        if(WIFEXITED(status))
          printf("Processo %d terminato con exit(%d)\n",
                 pidw,WEXITSTATUS(status));
        else
          printf("Processo %d terminato con segnale.\n",
                 pidw);
      }
      else
        printf("Processo %d in esecuzione.\n", pid);
    } while( pidw == 0 );
  }
  else {
    /* siamo nel figlio */
    printf("Processo figlio(pid=%d): sono generato dal processo (pid=%d)\n",
           getpid(),getppid());
    sleep(5);
    printf("Processo figlio(pid=%d): ho terminato!\n", getpid());
  }
```

```
  return(0);
}
```

11.8 Esecuzione esterna: `exec`, `system`

Un processo può decidere di cambiare il codice che sta eseguendo. In particolare, esso può rimpiazzare il suo codice, dati, heap e stack, con quelli di un eseguibile attraverso la famiglia di chiamate di sistema `exec()`

```
EXEC(3)

NAME
       execl, execlp, execle, execv, execvp - esegue un file eseguibile

SYNOPSIS
       #include <unistd.h>

       extern char **environ;

       int execl( const char *path, const char *arg0, const char *arg1, ...,
                  const char *argn, NULL);
       int execlp( const char *file, const char *arg0, const char *arg1, ...,
                   const char *argn, NULL);
       int execle( const char *path, const char *arg0, const char *arg1, ...,
                   const char *argn, NULL, char * const envp[]);
       int execv( const char *path, char *const argv[]);
       int execvp( const char *file, char *const argv[]);
```

I caratteri che seguono la stringa `exec` nei nomi delle chiamate di sistema individuano le funzionalità peculiari delle stesse. In particolare,

`p` indica una chiamata che prende un nome di file come argomento e lo cerca nelle directory specificate nella variabile di ambiente `PATH`;

`l` indica una chiamata che riceve una lista di argomenti terminata da un puntatore `NULL`;

`v` indica una chiamata che riceve un vettore nello stesso formato di `argv[]`;

`e` indica una chiamata che riceve anche un vettore di ambiente `envp[]` invece di utilizzare l'ambiente corrente.

Queste primitive consentono ad un processo figlio di svincolarsi dalla condivisione del codice con il padre, caricando un nuovo programma da un file eseguibile la cui locazione è specificata in `path`.

Pur perdendo la condivisione della regione del codice col processo padre, oltre al cambiamento delle regioni dei dati, heap e di stack, il processo figlio mantiene la condivisione dei file aperti. Se il file eseguibile non è trovato, la chiamata di sistema ritorna valore `-1`; altrimenti, il processo chiamante sostituisce il suo codice, dati, heap e stack con quelli dell'eseguibile e parte con l'esecuzione del nuovo codice. Una chiamata che ha successo non ritorna mai in quanto il codice chiamante è completamente sostituito dal nuovo codice.

Nelle chiamate che lo prevedono `arg0` deve essere il nome del file eseguibile, e la lista degli argomenti (`arg1, ..., argn`) deve essere terminata dal puntatore `NULL`. Similmente, per le chiamate che lo prevedono, `argv[0]` deve essere il nome del file eseguibile, `argv[i]` con `i=1,...,n` gli argomenti, e `argv[n+1]` deve essere il puntatore `NULL`. Il vettore `envp[]` è un array di puntatori a stringhe terminate dal `NULL` che rappresentano legami di ambiente, ed è egli stesso terminato dal puntatore `NULL` (ultimo elemento).

In realtà tutte le chiamate della famiglia `exec()` sono realizzate come funzioni di libreria che invocano la `execve()` che è l'unica vera chiamata di sistema.

> `ATTENZIONE:` una `exec()` **non prevede di tornare** al programma chiamante, e **non produce nuovi processi**.

Ci sono alcuni attributi che il processo che esegue la `exec()` mantiene, come ad esempio

- il `PID`;
- il `PPID`;
- il `PGID`;
- il real `UID`;
- il real `UGID`;
- i descrittori dei file che sono aperti al momento della chiamata;
- la directory corrente;
- la maschera di creazione dei diritti dei file;
- la maschera dei segnali;
- il terminale di controllo (vedi segnali);

Vediamo di seguito degli esempi di chiamata dei comandi della famiglia `exec()`. Iniziamo con l'esecuzione del comando `ls -l` attraverso la `execl()`:

```
/*  File:   dir01.c
    Specifica: esegue un 'ls -l'
*/
```

```
/* include per chiamate sui processi */
#include <unistd.h>

#include "sysmacro.h"

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

  if(argc != 2) {
    WRITE("Usage: dir file\n");
    return(0);
  }

  /* versione con execl */
  execl("/bin/ls", "/bin/ls", "-l", argv[1], NULL);
  WRITE("** Errore: exec non eseguita **\n");
  return(0);
}
```

Se l'eseguibile che si vuole eseguire si trova in una delle directory riferite dalla variabile di ambiente `PATH`, allora, invece del cammino assoluto del file eseguibile (in questo caso `/bin/ls`), si può più semplicemente utilizzare il nome dell'eseguibile in congiunzione con la `execlp()`:

```
/*  File:   dir02.c
    Specifica: esegue un 'ls -l'
*/

/* include per chiamate sui processi */
#include <unistd.h>

#include "sysmacro.h"

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

  if(argc != 2) {
    WRITE("Usage: dir file\n");
    return(0);
  }

  /* versione con execlp */
  execlp("ls", "ls", "-l", argv[1], NULL);
  WRITE("** Errore: exec non eseguita **\n");
  return(0);
}
```

Si noti che la `execlp()` richiede la conoscenza esatta da parte del programmatore del numero di argomenti da passare all'eseguibile. Questi, infatti, vengono riferiti esplicitamente nel codice C che fa uso della chiamata di sistema. Nel caso in cui tali informazioni non sia disponibile in anticipo, risulta utile utilizzare la `execvp()`:

```
/*  File:   dir03.c
    Specifica: esegue un 'ls -l'
*/

/* include per chiamate sui processi */
#include <unistd.h>

#include "sysmacro.h"

int main(int argc, char * argv[])
{
  char *argomenti[] = {"ls", "-l", argv[1], NULL};

  if(argc != 2) {
    WRITE("Usage: dir file\n");
    return(0);
  }

  /* versione con execvp */
  execvp("ls", argomenti);
  WRITE("** Errore: exec non eseguita **\n");
  return(0);
}
```

Un modo alternativo alla famiglia `exec()` per eseguire un comando di shell è quello di utilizzare

---

```
SYSTEM(3)

NAME
       system - esegue un comando di shell

SYNOPSIS
       #include <stdlib.h>

       int system (const char * string);
```

`system()` esegue il comando specificato in `string` chiamando `/bin/sh -c string`, e ritorna dopo che il comando è stato completato.

Utilizzando la `system()`, l'esempio precedente diventa:

```
/*  File:   dir04.c
    Specifica: esegue un 'ls -l'
*/

/* include per chiamate sui processi */
#include <unistd.h>

#include "sysmacro.h"

int main(int argc, char * argv[])
{
  char comando[200];

  if(argc != 2) {
    WRITE("Usage: dir file\n");
    return(0);
  }

  sprintf(comando, "ls %s", argv[1]);

  /* versione con system */
  system(comando);

  /* notare che la system ritorna! */
  return(0);
}
```

11.9 Esempi

11.9.1 Uso combinato di `execvp` e `wait`

Vediamo come si può eseguire un comando attraverso la generazione di un processo figlio.

```
/*  File:   comando.c
    Specifica: esegue un comando mediante execvp, e attende la terminazione
*/

/* include per chiamate sui processi */
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "sysmacro.h"
```

```
int main(int argc, char * argv[])
{
  int pid, status;

  if(argc < 2) {
    WRITE("Usage: comando cmd arg ...\n");
    return(0);
  }

  IFERROR(pid = fork(), "generando il figlio");
  if(pid == 0) {
    /* siamo nel figlio */
    execvp(argv[1],&argv[1]);
    printf("Non ho potuto eseguire %s\n", argv[1]);
    exit(-1);
  }
  else {
    /* siamo nel padre */
    wait(&status);
    WRITE("Comando terminato.\n");
  }
  return(0);
}
```

11.9.2 Esecuzione di una sequenza di comandi

Esemplificazione di come si possa eseguire una sequenza di comandi attraverso la `execvp`.

```
/*  File:   sequenza.c
    Specifica: esegue una sequenza di comandi mediante execvp
*/

/* include per chiamate sui processi */
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "sysmacro.h"
#include "util.h" /* funzioni di utilita' */

int main(int argc, char * argv[])
{
  int pid, status, argn, i;
  char ** newargv;
```

```
  if(argc < 2) {
    WRITE("Usage: sequenza cmd ...\n");
    return(0);
  }

  for( i = 1; i < argc; i++) {
    IFERROR(pid = fork(), "generando il figlio");
    if(pid == 0) {
      /* siamo nel figlio */
      newargv = split_arg(argv[i], " ", &argn);
      execvp(newargv[0], newargv);
      printf("Non ho potuto eseguire %s\n", argv[1]);
      exit(-1);
    }
    else {
      /* siamo nel padre */
      wait(&status);
      WRITE("Comando terminato.\n");
    }
  }
  return(0);
}
```

Ad esempio, `sequenza ls -l date`.

11.10 Realizzazione di processi in background

Combinando le chiamate di sistema `fork` ed `exec`, è possibile realizzare processi che eseguono attività in background. Ecco un esempio:

```
/*  File:   background.c
    Specifica: esegue un comando in background mediante execvp, e notifica
           la terminazione
*/

/* include per chiamate sui processi */
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "sysmacro.h"

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

```
  int pid, status;

  if(argc < 2) {
    WRITE("Usage: comando cmd arg ...\n");
    return(0);
  }

  IFERROR(pid = fork(), "generando il figlio");
  if(pid == 0) {
    /* siamo nel figlio */

    IFERROR(pid = fork(), "generando il nipote");
    if(pid == 0) {
      /* siamo nel nipote, il quale esegue il comando */
      execvp(argv[1],&argv[1]);
      printf("Non ho potuto eseguire %s\n", argv[1]);
      exit(-1);
    } else {
      /* siamo nel figlio, il quale attende la terminazione
         del nipote */
      wait(&status);
      WRITE("Comando terminato.\n");
    }
  }
  /* il padre termina subito */

  return(0);
}
```

11.11 Ridirezione: dup e dup2

[Glass, 415-416,438-439]

Vediamo come attraverso l'utilizzo della `fork` e di chiamate di sistema per la duplicazione di descrittori di file si riesca a realizzare la ridirezione dello standard input o output di un processo.

La duplicazione di descrittori di file si può ottenere tramite le seguenti chiamate di sistema

---

```
DUP(2)

NAME
       dup, dup2 - duplica un descrittore di file

SYNOPSIS
       #include <unistd.h>
```

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

`dup()` e `dup2()` creano una copia del descrittore di file `oldfd`. Se le chiamate di sistema ritornano con successo, il vecchio e nuovo descrittore possono essere usati intercambiabilmente. Essi condividono i `lock` (accesso in mutua esclusione), i puntatori alla posizione nel file, e (quasi) tutte le flag. `dup()` cerca la prima (cioè con indice più basso) locazione vuota (libera) nella tabella dei descrittori di file e vi ricopia il contenuto della locazione occupata da `oldfd`. `dup2()` chiude `newfd` se questo è attivo e quindi vi ricopia il contenuto della locazione occupata da `oldfd`. Se hanno successo, entrambe le chiamate restituiscono l'indice della locazione del nuovo descrittore di file, altrimenti restituiscono il valore `-1`.

Esempio di uso di `dup` e `dup2`

```
/* File: es-dup.c
   Specifica: esempio di uso di dup e dup2
 */

/* include per le chiamate a dup e dup2 */
#include <unistd.h>

/* include per chiamate sui file */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "sysmacro.h"

int main(int argc, char * argv[])
{
  int fd1, fd2, fd3;

  IFERROR(fd1 = open("test.txt", O_RDWR | O_TRUNC | O_CREAT), "aprendo test.txt");
  printf("fd1 = %d\n", fd1);
  IFERROR(write(fd1,"testo ",6),"scrivendo su test.txt");
  IFERROR(fd2 = dup(fd1),"duplicando test.txt");               /* copia di  fd1 */
  printf("fd2 = %d\n", fd2);
  IFERROR(write(fd2,"composto ",9),"scrivendo su test.txt con fd2");
  IFERROR(close(0), "chiudendo standard input");     /* chiusura standard input */
  IFERROR(fd3 = dup(fd1),"duplicando test.txt");         /* altra copia di  fd1 */
  printf("fd3 = %d\n", fd3);
  IFERROR(write(fd3,"a piu' ",7),"scrivendo su test.txt con fd3");
  IFERROR(dup2(3,2),"duplicando canale 3 nel 2");  /* duplica il canale 3 nel 2 */
```

```
  IFERROR(write(2,"mani!!\n",7),"scrivendo su test.txt");
  return(0);
}
```

Tornando alla ridirezione, consideriamo ad esempio il comando di shell `ls > ls.out` Per eseguire la ridirezione, la shell esegue la seguente serie di passi:

- il processo shell padre si duplica tramite una `fork` ed attende (tramite una `wait`) la terminazione del figlio;
- il processo shell figlio apre in scrittura il file `ls.out`, creandolo o troncandolo a seconda dei casi;
- quindi il processo shell figlio duplica il descrittore di file di `ls.out` (attraverso una `dup2`) nel descrittore di standard output (numero `1`) e poi chiude il descrittore di file originario di `ls.out`. Di conseguenza, tutto lo standard output è ridiretto verso `ls.out`;
- di seguito, il processo shell figlio, tramite una `exec`, provvede ad eseguire il codice associato al comando `ls`. Poichè i descrittori di file sono conservati dalla `exec`, tutto lo standard output generato da `ls` viene ridiretto verso `ls.out`;
- quando il processo shell figlio termina, il processo shell padre si risveglia. In particolare, i descrittori di file del padre non sono modificati dalle attività del figlio, poiché ogni processo mantiene la sua tabella privata dei descrittori di file.

Se si fosse voluto redirigere anche lo standard error, sarebbe bastato che il processo shell figlio avesse duplicato il descrittore di file di `ls.out` anche sullo standard error (numero `2`).

11.12 Esempi ed esercizi

11.12.1 Ridirezione dello standard output

Vediamo come si può ridirigere lo standard output tramite la `dup2`

```
/*  File:   redir1.c
    Specifica: redirige ">" un comando su un file, versione con dup2
*/

/* include per le chiamate a dup e dup2 */
#include <unistd.h>

/* include per chiamate sui file */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
#include "sysmacro.h"

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

  if(argc < 3) {
    WRITE("Usage: redir1 file cmd arg ...\n");
    return(0);
  }

  IFERROR(fd = open(argv[1],O_WRONLY|O_CREAT|O_TRUNC, 0644), "aprendo il file");
  dup2(fd, STDOUT); /* duplica fd sullo STDOUT */
  close(fd); /* fd non serve piu' */
  execvp(argv[2],&argv[2]); /* lo STDOUT viene ereditato da argv[2] */
  printf("Non ho potuto eseguire %s\n", argv[1]);

  return(-1);
}
```

Segue una variante, che utilizza `dup`, dove lo standard output è rediretto in modalità append.

```
/*  File:   redir2.c
    Specifica: redirige ">>" un comando su un file, versione con dup
*/

/* include per le chiamate a dup e dup2 */
#include <unistd.h>

/* include per chiamate sui file */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "sysmacro.h"

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

  if(argc < 3) {
    WRITE("Usage: redir2 file cmd arg ...\n");
    return(0);
  }
```

```
  IFERROR(fd = open(argv[1],O_WRONLY|O_CREAT|O_APPEND, 0644), "aprendo il file");
  close(STDOUT); /* chiude STDOUT */
  dup(fd); /* duplica fd sul descrittore piu' basso, cioe' STDOUT */
  close(fd); /* fd non serve piu' */
  execvp(argv[2],&argv[2]); /* lo STDOUT e' ereditato da argv[2] */
  printf("Non ho potuto eseguire %s\n", argv[1]);

  return(-1);
}
```

Esercizi

1 Si scriva un programma C, che implementi il comando listaproc, il quale invocato con

`listaproc` numero

genera un processo, stampa 1 e termina. Il processo generato deve generare un altro processo, stampare 2 e terminare. Il processo generato deve generare un altro processo, stampare 3 e terminare. E cosi via, fino a quando si giunge al processo che deve stampare numero, il quale non genera alcun processo.

2 Si realizzi un programma C che accetta come parametro un numero (intero) n e crea esattamente n processi. Ognuno dei processi viene attivato ed esegue k+2 iterazioni dove k è il numero progressivo di processo generato. Ad ogni iterazione il processo stampa il proprio `PID` e il proprio numero di iterazione e attende un secondo con una `sleep(1)`. Ogni processo termina passando alla `exit` (o `return`) il proprio pid modulo 256. Il processo padre attende la terminazione di tutti i figli e ne stampa il codice di uscita.

3 Si scriva un programma C che esegue il comando il cui nome e i cui parametri vengono passati come parametri della riga di comando. L'esecuzione del programma deve avvenire lanciando un nuovo processo. Alla terminazione del processo lanciato, vogliamo conoscere (stampare a terminale) lo stato di uscita del processo che ha eseguito il comando. In altre parole, se digitiamo il comando:

`a.out ls -l ciccio`

e `ciccio` non esiste nella directory corrente, vorremmo vedere qualcosa come:

`Il processo figlio e' terminato con una exit(1)`

Aiuto: Si esegua una `fork`. Nel ramo relativo al processo figlio si esegua una `execvp` utilizzando parte del vettore `argv` come vettore dei parametri passati alla `exec`. Nel ramo relativo al processo padre, si esegua una `wait`. Al ritorno della chiamata si stampi il codice di uscita del processo figlio con la macro `WEXITSTATUS`.

4 Si scriva un programma che invocato con copia file_1 ... file_n dir effettui la copia dei files file_1, ..., file_n nella directory dir in parallelo. Il processo padre deve creare n processi figli, e ogni processo figlio deve occuparsi della copia di un singolo file.

5 Si scriva un programma C, che implementi il comando rcom, il quale invocato con

```
> rcom nome-comando nome-1 nome-2 nome-3 ...
```

applica il comando nome-comando a tutti i file non directory contenuti nelle directory nome-1, nome-2 nome-3 ... ed in tutte le sotto-directory in esse contenute. L'applicazione del comando avviene mediante la generazione di un processo figlio il quale effettua una chiamata della famiglia exec o una chiamata system. Il comando rcom termina solo dopo che tutti i processi generati sono terminati. Alla generazione di un processo viene stampato il comando che il processo eseguirà ed il suo pid. Al termine di un processo viene stampato il pid, ed un codice di uscita (OK/Codice di Errore).

```
> ls A
total 2
   1 B/     1 testo.txt
> ls A/B
total 1
   1 dati.dat
> rcom ls A pippo

ls A/testo.txt ...[20453]
ls A/B/dati.dat ...[20455]
ls pippo ...[20456]
A/B/dati.dat
[20455] OK
A/testo.txt
[20453] OK
ls: pippo: No such file or directory
[20456] Error:1
```

Capitolo 12

Gestione dei segnali

12.1 Introduzione

[Glass, 439-452]

I *segnali* sono una forma di comunicazione asincrona fra processi (e fra il kernel e i processi). Essi servono principalmente per gestire eventi imprevisti o che avvengono in istanti imprevedibili, come

- Errori nei calcoli (Floating Point Error);
- Mancanza di alimentazione;
- Scadenza di un timer (segnale di allarme);
- Terminazione di un processo figlio;
- Richiesta di terminazione da tastiera (Control-C);
- Richiesta di sospensione da tastiera (Control-Z).

Tali eventi sono chiamati *interruzioni* in quanto interrompono il flusso di calcolo. Quando si verifica una interruzione, Unix/Linux invia al processo coinvolto nella interruzione un segnale.

Ad ogni evento corrisponde un unico segnale (numerato). Ad esempio, il segnale 8 corrisponde ad un floating point error.

Un segnale può essere inviato anche da un generico processo, così come un processo, dato un segnale particolare, può decidere di:

- ignorarlo;
- lasciare la gestione del segnale al kernel;
- gestirlo direttamente tramite una procedura di gestione propria (*signal handler*).

Quando arriva un segnale il flusso di calcolo è interrotto e, salvato il contesto relativo alla esecuzione del processo interrotto, viene mandata in esecuzione la procedura di gestione del segnale (*signal handler*). Terminata la procedura di gestione del segnale si riprende l'esecuzione sospesa in precedenza (dopo aver ripristinato il contesto precedentemente salvato). La procedura di gestione del segnale è tipicamente quella definita di default dal kernel, a meno che il programmatore non abbia previsto esplicitamente la gestione della interruzione tramite una (o più) procedure da lui stesso definite ed attivate tramite un opportuno meccanismo che sarà spiegato di seguito.

I segnali sono definiti nel file `/usr/include/sys/signal.h` (oppure `/usr/include/signal.h`). La corrispondenza fra tipo di segnale e numero ad esso associato dipende dal sistema (Linux, System V, BDS, ...). Quando si vuol riferire un segnale, è comunque conveniente utilizzare la macro simbolica corrispondente (definita in `/usr/include/bits/signum.h`):

```
/* Esempio: Signals.  */
#define SIGHUP          1       /* Hangup (POSIX).  */
#define SIGINT          2       /* Interrupt (ANSI).   Control-C */
#define SIGQUIT         3       /* Quit (POSIX).  */
#define SIGILL          4       /* Illegal instruction (ANSI).  */
#define SIGTRAP         5       /* Trace trap (POSIX).  */
#define SIGABRT         6       /* Abort (ANSI).  */
#define SIGIOT          6       /* IOT trap (4.2 BSD).  */
#define SIGBUS          7       /* BUS error (4.2 BSD).  */
#define SIGFPE          8       /* Floating-point exception (ANSI).  */
#define SIGKILL         9       /* Kill, unblockable (POSIX).  */
#define SIGUSR1         10      /* User-defined signal 1 (POSIX).  */
#define SIGSEGV         11      /* Segmentation violation (ANSI).  */
#define SIGUSR2         12      /* User-defined signal 2 (POSIX).  */
#define SIGPIPE         13      /* Broken pipe (POSIX).  */
#define SIGALRM         14      /* Alarm clock (POSIX).  */
#define SIGTERM         15      /* Termination (ANSI).  */
#define SIGSTKFLT       16      /* Stack fault.  */
#define SIGCLD          SIGCHLD /* Same as SIGCHLD (System V).  */
#define SIGCHLD         17      /* Child status has changed (POSIX).  */
#define SIGCONT         18      /* Continue (POSIX).  */
#define SIGSTOP         19      /* Stop, unblockable (POSIX).  */
#define SIGTSTP         20      /* Keyboard stop (POSIX).  Control-Z */
#define SIGTTIN         21      /* Background read from tty (POSIX).  */
#define SIGTTOU         22      /* Background write to tty (POSIX).  */
#define SIGURG          23      /* Urgent condition on socket (4.2 BSD).  */
#define SIGXCPU         24      /* CPU limit exceeded (4.2 BSD).  */
#define SIGXFSZ         25      /* File size limit exceeded (4.2 BSD).  */
#define SIGVTALRM       26      /* Virtual alarm clock (4.2 BSD).  */
#define SIGPROF         27      /* Profiling alarm clock (4.2 BSD).  */
#define SIGWINCH        28      /* Window size change (4.3 BSD, Sun).  */
#define SIGPOLL         SIGIO   /* Pollable event occurred (System V).  */
#define SIGIO           29      /* I/O now possible (4.2 BSD).  */
```

```
#define SIGPWR          30      /* Power failure restart (System V).  */
#define SIGUNUSED       32

#define _NSIG           64      /* Biggest signal number + 1
                                   (including real-time signals).  */
```

Il gestore di default dei segnali esegue una delle seguenti azioni:

- termina il processo e genera un file core (`dump`);
- termina il processo senza generare il file core (`quit`);
- ignora e rimuove [dalla coda di attesa di servizio] il segnale (`ignore`);
- sospende il processo (`suspend`);
- riattiva il processo (`resume`).

Un esempio di chiamata di sistema che invia un segnale è data dalla chiamata ad `alarm`.

```
ALARM(2)

NAME
       alarm - predispone una sveglia (alarm) per l'invio di un segnale

SYNOPSIS
       #include <unistd.h>

       unsigned int  alarm(unsigned int count)
```

`alarm()` predispone il kernel ad inviare un segnale `SIGALRM` al processo chiamante dopo `count` secondi. Se è già previsto un allarme, questo viene sovrascritto. Se `count` è `0`, tutte le richieste di allarme sono cancellate. Ecco un esempio tipico di utilizzo di `alarm`:

```
/*
   File: alarm.c
   Specifica: esempio di utilizzo di alarm
 */


/* include per chiamate sui segnali */
#include <unistd.h>

#include "sysmacro.h"
```

```
int main()
{
  alarm(3); /* predispone l'invio di un segnale di allarme */
            /* che sara' inviato dopo 3 secondi            */
  printf("Ciclo infinito...\n");
  while(1);
  printf("Mai eseguita");
  return(0);
}
```

12.2 Gestione personalizzata del segnale

Tramite la chiamata `sigaction` è possibile gestire direttamente i segnali.

```
SIGACTION(2)

NAME
       sigaction - POSIX signal handling function.

SYNOPSIS
       #include <signal.h>

       int sigaction(int signum, const struct sigaction  *act,
                     struct sigaction *oldact);
```

`sigaction()` permette ad un processo di specificare le azioni da intraprendere quando arriva un particolare segnale. Il parametro `signum` specifica il numero del segnale che deve essere gestito. La `struct sigaction` contiene vari campi, tra cui `void (sa_handler) (int)`. Questo è l'unico campo a cui siamo interessati, nel quale viene specificato il comportamento del processo alla ricezione del segnale. Dobbiamo quindi definire una funzione `sa_handler` che prende un intero e non ritorna alcun valore. Tale funzione può assumere uno dei seguenti valori:

- `SIG_IGN`, che indica di ignorare e rimuovere il segnale;
- `SIG_DFL`, che indica di utilizzare la procedura di gestione di default associata al segnale da gestire;
- l'indirizzo di una funzione definita dall'utente da usare per la gestione del segnale.

Il campo `oldact` punta ad una struttura dello stesso tipo della precedente, la quale, al ritorno della chiamata, conterrà nel campo `sa_handler` l'indirizzo del precedente gestore del signale, che è stato sostituito in caso di successo della chiamata. Se non siamo interessati a memorizzare il precedente

gestore, possiamo passare `NULL` come terzo parametro. Infine, la chiamata `sigaction()` ritorna un intero uguale a 0 in caso di successo, e -1 in caso di errore, riportando il codice di errore nella variabile `errno`.

I segnali `SIGKILL` e `SIGSTP` non possono essere gestiti in modo diverso dal default (questo garantisce di avere sempre un modo di terminare forzosamente un processo). Un processo figlio eredita il settaggio dei segnali dal padre al momento della `fork()`. Dopo una `exec()` i segnali ignorati in precedenza rimangono ignorati, ma le procedure di gestione dei segnali tornano ad essere quelle di default. Con l'eccezione di `SIGCHLD`, i segnali non sono accumulati (stacked), cioè se arrivano nuovi segnali quando il primo segnale non è stato ancora servito, questi ultimi vengono persi.

Un processo si può sospendere in attesa di un segnale tramite la chiamata `pause()`

```
PAUSE(2)

NAME
       pause - attende un segnale

SYNOPSIS
       #include <unistd.h>

       int pause(void);
```

`pause` sospende il processo che lo ha invocato e ritorna quando il processo riceve un segnale. Ritorna sempre il valore `-1`. Viene utilizzato per implementare la attesa passiva di un evento (tipicamente un segnale di allarme).

Ecco di seguito un tipico esempio di chiamata della `sigaction`:

```
/*
   File: handler.c
   Specifica: invio di un segnale di allarme con gestione personalizzata
              della routine di gestione del segnale
*/

/* include per chiamate sui segnali */
#include <unistd.h>
#include <signal.h>

#include "sysmacro.h"

int alarmFlag = 0;
struct sigaction azione;
void gestore_segnale(int);
```

```
int main(int argc, char *argv[])
{
  IFERROR(sigaction(SIGALRM,NULL,&azione),"eseguendo la sigaction");
  azione.sa_handler=gestore_segnale;
  IFERROR(sigaction(SIGALRM,&azione,NULL),"eseguendo la sigaction");
                /* istallazione routine gestione segnale */
  alarm(3);       /* predispone l'invio di un segnale di allarme */
  /* che sara' inviato dopo 3 secondi            */
  printf("Ciclo infinito...\n");
  while(!alarmFlag)
    pause(); /* sospende il processo chiamante fino
                alla ricezione di un segnale */
  printf("Terminazione ciclo a causa del segnale di allarme\n");
  return(0);
}

void gestore_segnale(int sig)
{
  alarmFlag = 1;
}
```

12.2.1 Invio di un segnale

Un processo può inviare un segnale ad un altro processo tramite la chiamata `kill`, che deve il suo nome al fatto che tipicamente il segnale inviato provoca la terminazione del processo ricevente. Si ricordi, tuttavia, che il segnale inviato può essere qualunque.

```
KILL(2)

NAME
       kill - invia un segnale ad un processo

SYNOPSIS
       #include <sys/types.h>
       #include <signal.h>

       int kill(int pid, int sigCode)
```

`kill()` invia il segnale con valore `sigCode` al processo con PID `pid`. Il segnale è inviato se almeno una delle seguenti condizioni è vera:

- il processo che invia il segnale e quello che lo riceve hanno lo stesso proprietario (owner);

- il processo che invia il segnale è posseduto da un super-utente.

Se `pid` è `0`, il segnale è inviato a tutti i processi appartenenti al gruppo di processo del mittente; Se `pid` è `-1` e il processo mittente è posseduto da un super-utente, il segnale è inviato a tutti i processi, incluso il mittente; Se `pid` è `-1` e il processo mittente non è posseduto da un super-utente, il segnale è inviato a tutti i processi posseduti dal proprietario del processo mittente, escluso il mittente stesso; Se `pid` è negativo e diverso da `-1`, il segnale è inviato a tutti i processi con lo stesso gruppo di processo (vedere Sezione 12.3) del valore assoluto di `pid`. Ritorna `0` in caso di successo e `-1` in caso di fallimento.

Un esempio di utilizzo della `kill` è quello in cui il processo padre provoca la terminazione dei figli inviando un segnale:

```
/*
    File:   timeout.c
    Specifica: esegue un comando per n secondi
*/

/* include per chiamate sui segnali */
#include <sys/types.h>
#include <signal.h>

/* include per chiamate sui processi */
#include <sys/wait.h>

#include "sysmacro.h"

struct sigaction azione;
void gestore_chld(int);

int main(int argc, char * argv[])
{
  int pid, secs, resto=0;

  if( argc < 3 ) {
    WRITE("Usage: timeout secs cmd arg ...\n");
    return(0);
  }

  secs = atoi(argv[1]);
  IFERROR(pid = fork(), "generando il figlio");
  if(pid == 0)     /* siamo nel figlio */
    IFERROR( execvp(argv[2],&argv[2]), argv[2]);

  /* siamo nel padre */
  IFERROR(sigaction(SIGCHLD,NULL,&azione),"eseguendo la sigaction");
```

```
  azione.sa_handler=gestore_chld;
  IFERROR(sigaction(SIGCHLD,&azione,NULL),"eseguendo la sigaction");
  resto=sleep(secs);     /* la sleep viene interrotta dal segnale */
  if (resto==0)
    IFERROR(kill(pid, SIGKILL),"inviando il segnale");
                         /* uso SIGKILL perche' non ignorabile */
  sleep(1);              /* attendo l'arrivo del segnale */
  return(0);
}

void gestore_chld(int sig)
{
  int pid, stato;

  pid = wait(&stato);
  if(WIFEXITED(stato))  /* terminazione mediante exit() */
    printf("Processo %d terminato con una exit(%d)\n",
           pid,WEXITSTATUS(stato));

  if(WIFSIGNALED(stato))  /* terminazione mediante segnale */
    printf("Processo %d terminato per una kill(%d)\n",
           pid,WTERMSIG(stato));
  exit(0);
}
```

12.3 Process Group

Ogni processo è membro di un *process group*. I figli di un processo hanno hanno lo stesso process group del padre. Il process group rimane lo stesso anche dopo la esecuzione di una `exec()`. Il process group di un processo può essere ottenuto utilizzando la chiamata di sistema `getpgrp()` e può essere modificato con la chiamata `setpgid()`.

```
SETPGID(2)

NAME
       getpgrp, setpgid - ottiene/setta l'identificatore di gruppo

SYNOPSIS
       #include <unistd.h>

       pid_t getpgrp(void);
       int setpgid(pid_t pid, pid_t pgid);
```

`getpgrp()` restituisce l'identificatore di process group associato al processo chiamante.

Un processo può cambiare il suo process group usando la chiamata di sistema `setpgid()`. `setpgid()` assegna all'identificatore di process group del processo con `PID pid` il valore `pgid`. Se `pid` è `0`, all'identificatore di process group del processo chiamante viene assegnato il valore `pgid`. Se `pgid` è `0`, `pid` viene usato come process group. Almeno una delle seguenti condizioni deve essere verificata:

- il processo chiamante e quello con `PID pid` devono avere lo stesso proprietario;
- il processo chiamante deve essere posseduto da un super-utente.

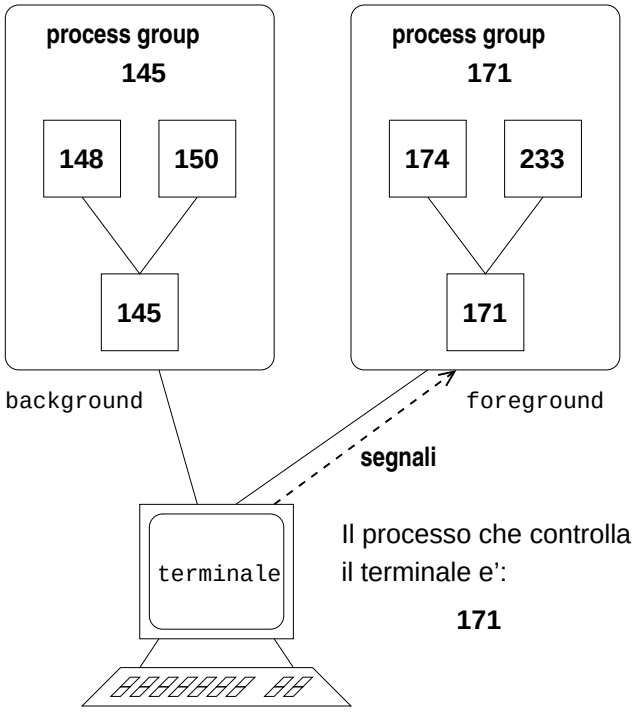
Il processo chiamante può porsi in un nuovo gruppo semplicemente facendo coincidere `pgid` con il suo `PID`. Se `setpgid()` fallisce, ritorna valore `-1`.

12.4 Terminale

Il concetto di process group è utile per la gestione di segnali provenienti da tastiera. Ogni processo può avere associato un terminale di controllo. Il terminale di controllo è ereditato dai figli (attraverso la `fork()`) e rimane invariato anche se viene eseguita una `exec()`.

Ogni terminale può essere associato ad un singolo processo di controllo. Quando si rileva da tastiera un metacarattere (ad esempio Control-C) il terminale invia il segnale corrispondente a tutti i processi che appartengono al process group del processo di controllo.

Se un processo cerca di leggere dal suo terminale di controllo e non è membro dello stesso process group del processo di controllo del terminale, gli viene inviato un segnale `SIGTTIN` che normalmente lo sospende. Per maggiori dettagli si rimanda a [**Glass, 448-452**].



12.5 Interruzione chiamate di sistema: `siginterrupt`

In Linux se una chiamata di sistema è interrotta da un segnale, questa viene fatta ripartire automaticamente. Si può cambiare tale comportamento da parte del sistema utilizzando

---

```
SIGINTERRUPT(3)

NAME
       siginterrupt - permette ai segnali di interrompere le chiamate di sistema

SYNOPSIS
       #include <signal.h>

       int siginterrupt(int sig, int flag);
```

`siginterrupt()` cambia il comportamento di riattivazione quando una chiamata di sistem è interrotta dal segnale la cui identità è specificata in `sig`. Se la `flag` è falsa (`0`), allora le chiamate di sistema sono fatte ripartire se interrotte dal segnale `sig` (comportamento di default in Linux). Tuttavia, se si è installato una nuova procedura di gestione del segnale tramite una `sigaction`, allora la chiamata di sistema è interrotta di default. Se la `flag` è vera (`1`) e nessun dato è stato trasferito (ad esempio, tramite una `write`), allora la chiamata di sistema viene interrotta dal segnale

`sig` ed essa stessa ritorna il valore `-1`. Se invece la `flag` è vera (`1`) e sono stati trasferiti dei dati, allora la chiamata di sistema viene interrotta dal segnale `sig` ed essa stessa ritorna la quantità di dati trasferiti. `siginterrupt()` restituisce `0` se ha successo e `-1` se il numero di segnale `sig` non è valido.

12.6 Esempi ed esercizi

12.6.1 Protezione di codice critico

La protezione da interruzione di codice critico può essere implementata attraverso una opportuna chiamata di `signal` che prescrive di ignorare il segnale generato dal `Control-C`.

```
/*
   File: critical.c
   Specifica: esempio di protezione di codice critico
 */

/* include per chiamate sui segnali */
#include <signal.h>

#include "sysmacro.h"

int main(int argc, char *argv[])
{
  struct sigaction oldHandler; /* per memorizzare il vecchio handler */
  struct sigaction newHandler; /* per memorizzare il nuovo handler */

  printf("Posso essere interrotto\n");
  sleep(3);
  newHandler.sa_handler=SIG_IGN;
  IFERROR(sigaction(SIGINT,&newHandler,&oldHandler),"eseguendo la sigaction");
                          /* ignora Control-C */
  printf("Ora non posso essere interrotto\n");
  sleep(3);
  IFERROR(sigaction(SIGINT,&oldHandler,NULL),"eseguendo la sigaction");
                          /* ripristina vecchio handler */
  printf("Posso essere interrotto di nuovo!\n");
  sleep(3);
  printf("Ciao!!!\n");
  return(0);
}
```

12.6.2 Sospensione e riattivazione processi

Vediamo di seguito un esempio di utilizzo della `kill` per realizzare la sospensione e successiva riattivazione di processi.

```
/*
   File: pulse.c
   Specifica: esempio di sospensione e riattivazione di processi
 */

/* include per chiamate sui segnali */
#include <sys/types.h>
#include <signal.h>

/* include per chiamate sui processi */
#include <unistd.h>

#include "sysmacro.h"

int main(int argc, char *argv[])
{
  int pid1, pid2;

  IFERROR(pid1 = fork(), "generando il figlio 1");
  if(pid1==0)
    {
      while(1)
        {
          printf("Figlio 1 (%d) e' vivo\n",getpid());
          sleep(1);
        }
    }
  IFERROR(pid2 = fork(), "generando il figlio 2");
  if(pid2==0)
    {
      while(1)
        {
          printf("Figlio 2 (%d) e' vivo\n",getpid());
          sleep(1);
        }
    }
  sleep(3);
  kill(pid1, SIGSTOP);
  system("ps f | grep pulse | grep -v sh");
  sleep(3);
  kill(pid1, SIGCONT);
```

```
  sleep(3);
  kill(pid1, SIGINT);
  kill(pid2, SIGINT);
  return(0);
}
```

12.6.3 Esempio di uso di signal

```
/*
   File: es-sigint.c
   Specifica: esempio di gestione di SIGINT
*/

/* include per chiamate sui segnali */
#include <sys/types.h>
#include <signal.h>

/* include per chiamate sui processi */
#include <unistd.h>

#include "sysmacro.h"

struct sigaction azione; /* per memorizzare il nuovo handler */
void gestore_sigint(int);

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

  IFERROR(sigaction(SIGINT,NULL,&azione),"eseguendo la sigaction");
  azione.sa_handler=gestore_sigint;
  IFERROR(sigaction(SIGINT,&azione,NULL),"eseguendo la sigaction");
                          /* installo gestore Ctrl-C */

  IFERROR(pid = fork(), "generando il figlio");

  if(pid == 0)
      printf("Figlio PID %d PGRP %d aspetta\n",getpid(),getpgrp());
  else
      printf("Padre PID %d PGRP %d aspetta\n",getpid(),getpgrp());
  pause();
  return(0);
}

void gestore_sigint(int sig)
```

```
{
  printf("Processo %d ha ricevuto un SIGINT\n",getpid());
  exit(1);
}
```

12.6.4 Uso di setpgid

```
/*
   File: es-setpgid.c
   Specifica: esempio di cambio di identificatore di gruppo
*/

/* include per chiamate sui segnali */
#include <sys/types.h>
#include <signal.h>

/* include per chiamate sui processi */
#include <unistd.h>

#include "sysmacro.h"

struct sigaction azione; /* per memorizzare il nuovo handler */
void gestore_sigint(int);

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

  IFERROR(sigaction(SIGINT,NULL,&azione),"eseguendo la sigaction");
  azione.sa_handler=gestore_sigint;
  IFERROR(sigaction(SIGINT,&azione,NULL),"eseguendo la sigaction");
                          /* installo gestore Ctrl-C */
  IFERROR(pid = fork(), "generando il figlio");
  if(pid == 0)
     IFERROR(setpgid(0,getpid()),"eseguendo la setpgid");
  printf("Processo PID %d PGRP %d in attesa\n",getpid(),getpgrp());
  for(i=1;i<=3;i++)  {
    printf("Processo %d e' vivo\n",getpid());
    sleep(1);
  }
  return(0);
}
```

```
void gestore_sigint(int sig)
{
  printf("Processo %d ha ricevuto un SIGINT\n",getpid());
  exit(1);
}
```

12.6.5 Esempio di intercettazione di SIGTTIN

Se un processo tenta di leggere dal terminale di controllo dopo essersi dissociato dal gruppo del terminale, il sistema gli invia un segnale SIGTTIN. Di seguito vediamo un esempio dove il segnale SIGTTIN è catturato.

```
/*
   File: es-sigttin.c
   Specifica: esempio di intercettazione di SIGTTIN
*/

/* include per chiamate sui segnali */
#include <sys/types.h>
#include <signal.h>

/* include per chiamate sui processi */
#include <unistd.h>
#include <sys/wait.h>

#include "sysmacro.h"

struct sigaction azione; /* per memorizzare il nuovo handler */
void gestore_sigttin(int);

int main(int argc, char *argv[])
{
  int status, pid;
  char carattere;

  IFERROR(pid = fork(), "generando il figlio");
  if(pid == 0)
    {
      IFERROR(sigaction(SIGTTIN,NULL,&azione),"eseguendo la sigaction");
      azione.sa_handler=gestore_sigttin;
      IFERROR(sigaction(SIGTTIN,&azione,NULL),"eseguendo la sigaction");
                          /* installo gestore SIGTTIN*/

      IFERROR(setpgid(0,getpid()),"eseguendo la setpgid");
```

```
      printf("Inserisci un carattere: ");
      carattere=getchar();
      printf("Hai inserito %c\n",carattere);
    }
  else
    {
      wait(&status);
    }
  return(0);
}

void gestore_sigttin(int sig)
{
  printf("Tentata lettura inappropriata dal terminale di controllo\n");
  exit(1);
}
```

12.6.6 Esempio di intercettazione di SIGSEGV

Il segnale `SIGSEGV` (*SEGmentation Violation*) viene inviato dal kernel al processo che ha tentato di accedere al di fuori dei segmenti di memoria assegnati. Un caso tipico è quando si accede oltre i limiti di un array.

```
/*
    File:   sig.c
    Specifica: esempio di installazione gestore SIGSEGV
*/

/* include per chiamate sui segnali */
#include <signal.h>

#include "sysmacro.h"

void gestore_SIGSEGV(int);
struct sigaction azione;

int main(int argc, char * argv[])
{
  char stringa[10] ;

  IFERROR(sigaction(SIGSEGV,NULL,&azione),"eseguendo la sigaction");
  azione.sa_handler=gestore_SIGSEGV;
  IFERROR(sigaction(SIGSEGV,&azione,NULL),"eseguendo la sigaction");
                       /* installo gestore per SIGSEGV */
  stringa[100000]=1;   /* scrivo in area non assegnata */
```

```
  return(0);
}

void gestore_SIGSEGV(int sig)
{
  printf("ricevuto segnale numero %d\n",sig);
  exit(-1);
}
```

Esercizi

1 Si scriva un programma C `sveglia.c` che chiamato con sveglia num-secondi cmd lista-opzioni esegue in background il comando cmd con opzioni lista-opzioni. Se non termina dopo num-secondi secondi, viene terminato con un signale `SIGINT`. Si utilizzi un solo gestore per i segnali `SIGALRM` e `SIGCHLD`.

2 Si scriva un programma C `schedulatore.c` che chiamato con schedulatore comando lista-opzioni esegue il comando cmd con opzioni lista-opzioni per 5 secondi, poi lo sospende (con un signale `SIGSTOP`) e dopo 5 secondi lo riavvia con `SIGCONT`, e cosi' via fino alla terminazione del processo. Si testi il programma sul programma

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

int main()
{
  int i=0;
  for(;;i++) {
    sleep(1);
    printf("%d\n",i);
  }
  return(0);
}
```

3 Si scriva un programma C `output.c` che richiamato con output file comando lista-opzioni rediriga lo standard output del comando sul file passato come primo argomento (il comportamento deve mimare la ridirezione della shell comando lista-opzioni > file).

4 Si scriva un programma C `red.c` che richiamato con output file1 file2 comando lista-opzioni rediriga lo standard input del comando nel file file1 e lo standard output sul file file2 (il comportamento deve mimare la ridirezione della shell comando lista-opzioni < file1 > file2).

5 Si scriva un programma C, che implementi il comando alter, il quale invocato con

```
> alter cmd arg ...
```

esegue il comando `cmd arg ...` per un secondo, quindi lo sospende per un altro secondo, quindi lo riattiva per un secondo, e così via fino al termine del comando (si provi, ad esempio, `alter emacs`). Al termine del comando, si stampi il codice di uscita.

Capitolo 13

Gestione dei pipe

13.1 Introduction

[Glass, 453-459]

I *pipe* costituiscono meccanismi di comunicazione tra processi. Come si intuisce dal significato letterale del termine inglese (in italiano, "tubo" o "condotto"), un pipe permette di collegare l'output di un processo direttamente in ingresso ad un altro processo. Un esempio di pipe, è il seguente comando di shell

```
who | wc -l
```

che conta quanti utenti sono presenti sul sistema. Infatti, l'output del comando `who`, che restituisce sullo standard output la lista degli utenti correntemente connessi, uno per ogni linea, viene "incanalato" attraverso il pipe (sintatticamente indicato dal carattere "|") e fornito direttamente sullo standard input del comando `wc` che, come prescritto della opzione `-l`, restituisce sullo standard output il numero di linee ricevute sullo standard input.

Nell'esempio appena visto, bisogna puntualizzare che i processi che realizzano i due comandi `who` e `wc` sono di fatto concorrenti. Questi cioè non sono attivati uno di seguito all'altro, ma "simultaneamente". In particolare, il processo che realizza `who`, man mano che genera dati di output, li scrive nel pipe, il quale automaticamente li bufferizza in una opportuna area di memoria riservata. Nel caso in cui tale area si riempia (perchè il processo collegato all'altra estremita del pipe non provvede a leggerli, e quindi di fatto a rimuoverli da tale area), il processo che esegue la scrittura (nel nostro caso, quello che realizza il comando `who`) viene sospeso. Non appena alcuni dati vengono letti (e quindi rimossi) dal pipe, il processo che scrive viene riattivato. In modo complementare, se il processo lettore (nel nostro caso `wc`) tenta di leggere da un pipe che è "vuoto", allora viene sospeso fino all'arrivo di qualche dato nel pipe.

In UNIX esistono due tipi di pipe. Il primo viene detto "unnamed", in quanto non ha associato un nome "visibile" da altri processi, ma viene utilizzato esclusivamente per realizzare la comunicazione

fra processi parenti. Il secondo è invece detto "named" in quanto è identificato da un nome visibile, e quindi tipicamente utilizzato per la comunicazione fra processi che non sono in relazione diretta di parentela.

13.2 Pipe senza nome: `pipe`

Un pipe senza nome costituisce un canale unidirezionale bufferizzato (su Linux, la dimensione del buffer è di 4Kb), dove si possono scrivere dati usando la `write` e leggerli usando la `read`. Un pipe senza nome viene creato attraverso la seguente chiamata di sistema

---

```
PIPE(2)

NAME
       pipe - crea un pipe

SYNOPSIS
       #include <unistd.h>

       int pipe(int filedes[2]);
```

`pipe()`, dopo aver creato un pipe senza nome, restituisce due descrittori di file: l'ingresso del pipe (utilizzato dalla `write`) è identificato dal descrittore di file memorizzato in `filedes[1]`, mentre l'uscita del pipe (utilizzata dalla `read`) dal descrittore di file memorizzato in `filedes[0]`. Il processo che scrive nel pipe sottostà alle seguenti regole:

- se un processo scrive su un pipe il cui descrittore in lettura (`filedes[0]`) è stato chiuso, la `write` fallisce ed un segnale SIGPIPE viene inviato al processo scrittore. La azione di default del segnale è quello di terminare il processo che lo riceve;
- se il processo scrittore scrive una quantità di byte che possono essere ricevuti dal pipe (cioè c'è spazio nell'area di memoria riservata al pipe), la `write` è garantita essere "atomica" (scrittura in mutua esclusione). Tale atomicità, tuttavia, non è garantita nel caso in cui la `write` tenti di scrivere più byte di quanti il pipe possa ricevere.

Per il processo lettore, invece, si applicano le seguenti regole:

- se un processo legge da un pipe il cui descrittore in scrittura (`filedes[1]`) è stato chiuso, la `read` ritorna restituendo un valore nullo, che indica la fine dell'input;
- se un processo legge da un pipe vuoto il cui descrittore in scrittura è ancora aperto, il lettore è sospeso fino all'arrivo di dati nel pipe;
- se un processo tenta di leggere da un pipe più byte di quelli effettivamente presenti, tutti i dati disponibili sono letti e la `read` ritorna restituendo il numero di byte effettivamente letti.

La chiamata di sistema `lseek` non ha alcun senso quando chiamata su un pipe. Infine, se il kernel non riesce ad allocare abbastanza spazio per un pipe, la `pipe` restituisce il valore `-1`, altrimenti il valore `0`.

Bisogna notare che il fatto di accedere ad un pipe senza nome attraverso i descrittori di file, tipicamente limita l'utilizzo di tali pipe fra processi in relazione di discendenza, anche se attraverso tecniche più sofisticate si possono superare tali limitazioni.

Una sequenza tipica di utilizzo della `pipe` è la seguente:

1. il processo padre crea un pipe senza nome attraverso la chiamata `pipe`;
2. il processo padre si duplica attraverso la chiamata `fork`;
3. il processo scrittore (che può essere il padre o il figlio) provvede a chiudere il proprio descrittore in lettura del pipe, mentre il lettore chiude il proprio descrittore in scrittura[1];
4. i processi comunicano attraverso opportune `write` e `read`;
5. ogni processo chiude il proprio descrittore rimasto attivo non appena ha terminato la propria attività relativa all'uso del pipe.

Quanto detto in precedenza implica che l'uso bidirezionale del pipe non è ammesso. La comunicazione bidirezionale si può realizzare attraverso l'uso di due pipe.

Vediamo di seguito un esempio tipico di `pipe`.

```
/*  File:   pipeuni.c
    Specifica: comunicazione unidirezionale dal figlio al padre via pipe
           senza nome. codifica "a lunghezza fissa" del messaggio
*/

/* include per chiamate sui pipe */
#include <unistd.h>

#include "sysmacro.h"

#define MAXS 256

int main(int argc, char * argv[])
{
  int pid, fd[2];
  char messaggio[MAXS];

  IFERROR(pipe(fd), "creando il pipe senza nome");
```

[1]Si ricordi che con la duplicazione di un processo si duplicano anche i descrittori di file aperti al momento della duplicazione.

```
  IFERROR(pid = fork(), "generando il figlio");
  if( pid == 0 ) {
    int i;
    /* siamo nel figlio */

    close(fd[0]);  /* chiudo lettura sul pipe */

    sprintf(messaggio, "Dal figlio (%d) al padre (%d)\n",
            getpid(), getppid());

    /* provare con MAXS = 4097

    for(i=0; i< MAXS -1; i++)
      messaggio[i]='a';
    messaggio[MAXS-1] = '\0';
    */

    write(fd[1], messaggio, MAXS); /* scrive l'intero array */
    close(fd[1]);

  } else {

    /* siamo nel padre */

    close(fd[1]); /* chiudo scrittura sul pipe */

    read(fd[0], messaggio, MAXS); /* leggo per MAXS bytes */
    WRITE(messaggio);

    close(fd[0]);
  }
  return(0);
}
```

In questo esempio, il processo padre ed il processo figlio "concordano" una modalità di scambio del messaggio, stabilendo che esso ha una lunghezza fissa pari a `MAXS`. Questa convenzione si rende necessaria affinché il processo lettore sappia determinare con certezza la fine del messaggio. La codifica "a lunghezza fissa" ha, però lo svantaggio di sprecare banda di comunicazione (solo una parte dei bytes comunicati contiene il messaggio). Se il messaggio fosse unico, una codifica alternativa potrebbe considerare la fine file (ovvero la chiusura del pipe da parte dello scrittore) come terminatore del messaggio. Una seconda alternativa, di uso più generale, è quella di spedire prima la lunghezza del messaggio (codificata a sua volta "a lunghezza fissa", ovvero spedendo la zona di memoria che contiene la lunghezza) e poi il messaggio stesso.

```
/*  File:   pipeunivar.c
```

```
    Specifica: comunicazione unidirezionale dal figlio al padre via pipe
           senza nome. codifica "a lunghezza variabile" dei messaggi
*/

/* include per chiamate sui pipe */
#include <unistd.h>

#include "sysmacro.h"

#define MAXS 256

int main(int argc, char * argv[])
{
  int pid, intero, fd[2];
  char messaggio[MAXS];

  IFERROR(pipe(fd), "creando il pipe senza nome");

  IFERROR(pid = fork(), "generando il figlio");
  if( pid == 0 ) {
    /* siamo nel figlio */

    close(fd[0]);  /* chiudo lettura sul pipe */

    sprintf(messaggio, "Dal figlio (%d) al padre (%d)\n",
            getpid(), getppid());

    intero = strlen(messaggio)+1;
    write(fd[1], &intero, sizeof( int ) ); /* scrive la zona di memoria
                                              occupata da intero */
    write(fd[1], messaggio, intero); /* scrive il messaggio,
                                        incluso lo '\0' finale */
    close(fd[1]);

  } else {

    /* siamo nel padre */

    close(fd[1]); /* chiudo scrittura sul pipe */

    read(fd[0], &intero, sizeof( int )); /* leggo la lunghezza del messaggio */
    sprintf(messaggio, "Sto per ricevere %d bytes\n", intero);
    WRITE(messaggio);
    read(fd[0], messaggio, intero); /* leggo il messaggio */
    WRITE(messaggio);
```

```
    close(fd[0]);
  }
  return(0);
}
```

13.3 Pipe con nome: `mkfifo`

Una alternativa ai pipe senza nome è costituita dai pipe con nome (conosciuti anche come FIFO, "first in, first out"). Rispetto ai primi, questi hanno i seguenti vantaggi:

- hanno un nome che esiste nel file system;
- possono essere utilizzati da processi che non sono in relazione di parentela;
- rimangono in vita fino a quando non sono rimossi esplicitamente dal file system.

Da un punto di vista operativo, i pipe con nome si comportano esattamente come quelli senza nome, a parte il fatto che sono in grado di bufferizzare molta più informazione (tipicamente 40K).

I pipe senza nome sono file speciali del file system e possono essere creati in uno dei seguenti modi:

- attraverso la utility `mknod`;
- usando la chiamata di sistema `mknod()`;
- usando la routine di libreria `mkfifo()`.

Si noti che tutte le possibilità fanno riferimento agli strumenti per la creazione generica di un file speciale. Quindi un pipe con nome è visto come una particolare istanza di file speciale. In ogni caso, poiché la creazione di un file speciale richiede diritti particolari, si consiglia di creare il pipe con nome sotto la directory temporanea `/tmp`.

Vediamo di seguito come creare un pipe con nome utilizzando la utility `mknod`:

```
well20 ~> mknod /tmp/pippo p
well20 ~> chmod ug+rw /tmp/pippo
well20 ~> ls -l /tmp/pippo
   0 prw-rw-r--    1 sperduti    sperduti           0 Feb 26 11:17 /tmp/pippo|
```

Si noti che `mknod` deve essere invocata con opzione `p`, che indica la direttiva di creare un pipe con nome. Una volta creato, il pipe deve essere reso accessibile ad altri processi. Per questo motivo si utilizza il comando `chmod`, come visto sopra. Infine, se si prova a listare le caratteristiche del pipe attraverso il comando `ls -l`, si può notare che il pipe è caratterizzato dal tipo 'p'.

Lo stesso effetto si può ottenere attraverso le seguenti chiamate in C:

```
mknod("/tmp/pippo", S_IFIFO, 0);  /* crea un pipe con nome ‘‘pippo’’ */
chmod("/tmp/pippo", 0664);         /* modifica i permessi, abilitando lettura
                                        e scrittura da parte di altri processi */
```

L'ultimo modo per creare un pipe con nome consiste nell'utilizzare la seguente routine.

```
MKFIFO(3)

NAME
       mkfifo - crea un file speciale FIFO (pipe con nome)

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>

       int mkfifo ( const char *pathname, mode_t mode );
```

`mkfifo()` crea un file speciale FIFO con nome `pathname`. I diritti del file sono specificati da `mode`. Questi sono modificati nel modo usuale dalla `umask` del processo: i diritti sono settati al risultato della operazione `(mode & ~umask)`.

Ecco un esempio di codice C per creare un pipe con nome con la `mkfifo()`:

```
char *nome_pipe = "/tmp/pippo";

mkfifo(nome_pipe,0644)
```

Una volta creato un file speciale FIFO, ogni processo lo può aprire in lettura o scrittura, come con un file regolare. Tuttavia, prima che il pipe con nome si possa utilizzare, questi deve essere aperto da tutti e due gli estremi: da una parte in lettura e dall'altra in scrittura. In particolare:

- se un processo tenta di aprire un pipe con nome in sola lettura e nessun processo lo ha ancora aperto in scrittura, il processo lettore si sospende fino a quando un altro processo non apre il pipe in scrittura, a meno che non siano utilizzati i flag `O_NONBLOCK` o `O_NDELAY` con la `open`, nel qual caso la `open` ha successo immediatamente;
- se un processo tenta di aprire un pipe con nome in sola scrittura e nessun processo lo ha ancora aperto in lettura, il processo scrittore si sospende fino a quando un altro processo non apre il pipe in lettura, a meno che non siano utilizzati i flag `O_NONBLOCK` o `O_NDELAY` con la `open`, nel qual caso la `open` fallisce immediatamente;
- i pipe con nome non funzionano attraverso la rete (i processi in comunicazione, devono essere in esecuzione sulla stessa macchina).

In genere i pipe si usano con

- un lettore (o server di comandi);
- uno o più scrittori (o client di comandi).

Questo implica che tipicamente è responsabilità del lettore quella di creare il pipe con nome se questo non esiste.

13.4 Esempi ed esercizi

13.4.1 Pipe fra due comandi

Ecco un esempio di come si può realizzare un collegamento di tipo pipe fra due comandi.

```
/*  File:   pipe.c
    Specifica: pipe di due comandi
*/

/* include per chiamate sui pipe */
#include <unistd.h>

#include "sysmacro.h"

int main(int argc, char * argv[])
{
  int pid, pos, fd[2];

  if( argc < 4 ) {
    WRITE("Usage: pipe cmd1 par ... InPipeCon cmd2 par ...\n");
    return(0);
  }

  pos = 1;
  while( (pos < argc) && strcmp(argv[pos],"InPipeCon") )
    pos++;

  /* argv[pos] = "InPipeCon" */
  argv[pos] = NULL;

  IFERROR(pipe(fd), "creando il pipe senza nome");

  IFERROR(pid = fork(), "generando il figlio");
  if( pid == 0 ) {
```

```
    /* siamo nel figlio */

    close(fd[0]);  /* chiudo lettura sul pipe */
    dup2(fd[1], STDOUT); /* redirigo STDOUT su fd[1]  */
    close(fd[1]);

    IFERROR(execvp(argv[1], &argv[1]), argv[1]);

  } else {

    /* siamo nel padre */

    close(fd[1]); /* chiudo scrittura sul pipe */
    dup2(fd[0], STDIN); /* redirigo STDIN su fd[0]  */
    close(fd[0]);

    IFERROR(execvp(argv[pos+1], &argv[pos+1]), argv[pos+1]);
  }

  return(0);
}
```

13.4.2 Utilizzo di pipe con nome

Vediamo un utilizzo del pipe con nome. L'esempio considera un processo lettore ed uno scrittore, non in relazione di parentela fra loro, che comunicano attraverso un pipe con nome (di cui conoscono il nome). La codifica del messaggio adottata dai due processi è quella "a lunghezza fissa".

```
/*  File:   lettore.c
    Specifica: lettore da pipe con nome, creandolo se non esistente
*/

/* include per chiamate sui file */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/* include per chiamate sui pipe */
#include <unistd.h>

#include "sysmacro.h"

#define MAXS 100
```

```
int ispipe(const char *filename);

int main(int argc, char * argv[])
{
  int n, fd;
  char messaggio[MAXS];

  if( argc != 2 ) {
    WRITE("Usage: lettore pipeconnome\n");
    return(0);
  }

  if( ! ispipe(argv[1]) ) {
    IFERROR(mkfifo(argv[1],0644), argv[1]);
  }

  IFERROR(fd = open(argv[1], O_RDONLY), argv[1]);

  while( read(fd, messaggio, MAXS) > 0 )
    WRITE(messaggio);

  IFERROR(n, argv[1]);
  IFERROR(close(fd), argv[1]);

  return 0;
}


int ispipe(const char *filename)
{
  struct stat info;

  return (stat(filename, &info) != -1) && S_ISFIFO(info.st_mode);
}

/*  File:   scrittore.c
    Specifica: scrive su un file (non necessariamente un pipe), aspettando
           se non esistente
*/

/* include per chiamate sui file */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
/* include per chiamate sui pipe */
#include <unistd.h>

#include "sysmacro.h"

#define MAXS 100

int main(int argc, char * argv[])
{
  int fd;
  char messaggio[MAXS];

  if( argc != 2 ) {
    WRITE("Usage: scrittore pipeconnome\n");
    return(0);
  }

  /* ciclo di apertura */
  while( (fd = open(argv[1], O_WRONLY)) == -1 )
    sleep(1);

  sleep(2); /* primo messaggio */
  sprintf(messaggio,"Sono il processo con pid: %d\n",getpid());
  write(fd, messaggio, MAXS);

  sleep(2); /* secondo messaggio */
  sprintf(messaggio,"In esecuzione sull'host: %s\n",getenv("HOSTNAME"));
  write(fd, messaggio, MAXS);

  IFERROR(close(fd), argv[1]);

  return 0;
}
```

Esercizi

1 Si scriva un programma C, che implementi il comando pipebid, il quale invocato con

```
> pipebid numero
```

crea un processo figlio. Attraverso un pipe senza nome, il padre spedisce gli interi da 1 a numero al figlio. Attraverso un altro pipe senza nome, il figlio, ricevuto un intero, spedisce al padre il quadrato di tale intero.

```
> pipebid 4
Padre. Spedito: 1
```

```
Padre. Spedito: 2
Padre. Spedito: 3
Padre. Spedito: 4
Padre. Ricevuto: 1
Padre. Ricevuto: 4
Padre. Ricevuto: 9
Padre. Ricevuto: 16
```

Suggerimento. La soluzione ovvia di questo esercizio è a rischio di deadlock. Si analizzi accuratamente cosa avviene se `numero` è tanto grande da far sì che i messaggi con gli interi da 1 a `numero` (e quelli con i relativi quadrati) riempano completamente il buffer della pipe...

2 Si scriva un programma C `pipe.c` che richiamato con pipe comando1 lista-opzioni1 PIPE comando2 lista-opzioni2 si comporti come il comando dato dalla shell: comando1 lista-opzioni1 | comando2 lista-opzioni2.

3 Si scrivano due programmi C, che implementino i comandi `lettorebid` e `scrittorebid`, i quali invocati con

```
> lettorebid nomefile & scrittorebid nomefile &
```

si comportano come segue. Scrittorebid scrive sul pipe nomefile un nome di file, diciamo piperitorno, e crea un pipe con tale nome. Lettorebid legge dal pipe nomefile il nome del file, quindi scrive su di esso un messaggio, diciamo Un messaggio dal lettore allo scrittore.

```
> lettorebid file & scrittorebid file &
[2] 13457
[3] 13458
Un messaggio dal lettore allo scrittore
[3]    Done                          scrittorebid file
[2]  - Done                          lettorebid file

> ls -la file piperitorno
   0 prw-r--r--   1 ruggieri personal        0 Apr 28 18:21 file|
   0 prw-r--r--   1 ruggieri personal        0 Apr 28 18:21 piperitorno|
```

4 Si scriva un programma che crea due processi figli e comunica ai processi, attraverso un pipe senza nome, i numeri interi da 1 a argv[1] (immesso dalla riga di comando). I processi figli leggono i numeri dal pipe ed alla fine stampano la somma di tutti i numeri letti.

5 Si scriva un programma `bff` che, invocato con

```
bff comando parametri
```

esegua `comando parametri`, intercetti il suo standard output, rimuova tutte le vocali, e invii il risultato sul proprio standard output. Per esempio:

```
> bff ls -la file piperitorno
   0 prw-r--r--   1 rggr prsnl        0 pr 28 18:21 fl|
   0 prw-r--r--   1 rggr prsnl        0 pr 28 18:21 pprtrn|
```

Capitolo 14

Esempi di progetto di una shell

In questo capitolo verranno progettati degli interpreti di comandi (*shell*) di complessità crescente. Si inizia da una shell che semplicemente invoca la chiamata `system` su ogni comando ricevuto, fino ad arrivare ad una shell in grado di riconoscere comandi interni (`cd`, `ls`, ecc.) ed esterni, sequenze di comandi, lancio di comandi in background (`&`), ridirezione e pipeline.

Le shell presentate sono disponibili nel formato sorgente in allegato a questa dispensa (vedi Sezione 8.6). Per motivi di spazio, nel seguito vengono presentate solo le parti più rappresentative del codice sorgente.

14.1 Versioni 0.x

14.1.1 Versione 0.0: interpretazione mediante `system`

Una shell può essere rapidamente sviluppata utilizzando la chiamata di sistema `system` per l'interpretazione dei comandi. In questo caso, la shell consiste di un semplice ciclo di lettura comando da standard input e chiamata alla `system`.

```
/*
  File:   shell00.c
  Specifica: shell: interprete di comandi V0.0
*/

/* include per chiamate sui file */
#include <sys/types.h>
#include <sys/wait.h>

#include "sysmacro.h"

#define MAXLINE 256 // lunghezza massima comando
```

```
int main()
{
 char line[MAXLINE];
 int ncommand = 0, // ncommand e' il numero del comando attuale
   n, stato = 0;   // stato e' il codice di ritorno dell'ultimo comando

 do {
   ncommand++;

   /* prompt = stato ultimo comando, numero comando */
   sprintf(line, "S=%d C=%d> ",stato, ncommand);
   WRITE(line);

   /* legge un comando */
   IFERROR(n = read(STDIN, line, MAXLINE),"stdin");
   line[n-1] = 0;

   /* se comando = "exit" esce dal ciclo */
   if( strcmp(line, "exit") == 0 )
     break;

   /* interpretazione comando per mezzo della system() */
   stato = system(line);

   /* calcolo stato di ritorno */
   if( WIFEXITED(stato) )
     stato = WEXITSTATUS(stato);
   else
     stato = WTERMSIG(stato);

 } while( 1 );

 WRITELN("Bye");
 return(0);
}
```

Dal momento che la chiamata `system` richiama la `tcshell` (o quella predefinita dall'utente), il vantaggio di questa soluzione è che sono disponibili molte delle funzionalità della `tcsh`, quali espansione dei metacaratteri nei nomi dei file, ridirezione, pipelining, lancio di comandi in background, sequenze di comandi. Per lo stesso motivo, però il comando `cd` ha l'effetto di cambiare la directory corrente del processo `tcshell` richiamato dalla `system` e non, come si vorrebbe, del processo `shell00`.

```
> shell00
S=0 C=1> ls *
```

```
makefile  shell00  shell00.c  sysmacro.h
S=0 C=2> ls > listafile
S=0 C=3> ls | wc
      5       5      48
S=0 C=4> ls ; wc listafile
listafile  makefile  shell00  shell00.c  sysmacro.h
      5       5      48 listafile
S=0 C=5> ls &
listafile  makefile  shell00  shell00.c  sysmacro.h
S=0 C=6> pwd
/home/ldb/ruggieri/LabIV/SYSCALL/SHELL/V0.0
S=0 C=7> cd ..
S=0 C=8> pwd
/home/ldb/ruggieri/LabIV/SYSCALL/SHELL/V0.0
S=0 C=9> exit
Bye
```

Esercizi

1 Perché il completamento dei nomi dei file (premendo il tasto `[Tab]`) non funziona sulla `shell00`?

14.1.2 Versione 0.1: interpretazione mediante `execvp`

Una prima variante consiste nel chiamare direttamente il programma richiesto dall'utente utilizzando una delle funzioni `exec..()`. Da un lato, questo *non* ci permetterà di sfruttare le funzionalità della shell (ridirezione, pipelining, ecc.). Dall'altro, però, è un passo avanti verso un maggiore controllo delle funzionalità della *nostra* shell.

Un problema da affrontare è che le `exec..()` richiedono che il nome del programma e gli argomenti siano passati separatamente, e non come una singola stringa. A tal fine, ricordiamo che nei file di utilità `util.c` e `util.h` ci sono la definizione e la dichiarazione della funzione:

```
char ** split_arg(char *line, char *sep, int *argc);
```

la quale invocata ad esempio con:

```
char **argv;

argv = split_arg("ls -l -a *", " ", &argc)
execvp(argv[0], argv);
```

estrae i token della stringa `ls -l -a *` divisi da un qualsiasi carattere in (qui, solo spazio) ritornando un puntatore ad un vettore di stringhe. `argv[0]` punterà a `ls`, `argv[1]` a `-l`, `argv[2]` a `-a`, `argv[3]` a `*` e `argv[4]` a `NULL`. Infine, `split_arg` scriverà in `argc` il numero di token trovati, ovvero 4.

Il modo in cui `split_arg` costruisce il vettore di stringhe (ovvero con l'ultimo elemento del vettore che punta a `NULL`) è esattamente quanto richiesto dalla `execvp`, che quindi è la naturale candidata tra le funzioni `exec..()` ad essere utilizzata.

Riportiamo di seguito una sessione d'uso. Si noti come non siano più disponibili l'espansione dei metacaratteri nei nomi dei file, la ridirezione, e la sequenza di comandi. I parametri dei comandi, invece, vengono correttamente passati in fase di chiamata.

```
> shell01
S=0 C=1> ls *
ls: *: No such file or directory
S=1 C=2> ls
makefile  shell01.c   shell01.o   util.c  util.o
shell01   shell01.c~  sysmacro.h  util.h
S=0 C=3> ls -l makefile
-rw-r--r--    1 ruggieri personal      270 Feb 22 17:25 makefile
S=0 C=4> ls ; ls
ls: ;: No such file or directory
ls: ls: No such file or directory
S=1 C=5> ls > a
ls: >: No such file or directory
ls: a: No such file or directory
S=1 C=6>
```

Esercizi

1 Si scriva il codice della funzione `split_arg` utilizzando la funzione `strsep` (si veda il manuale in linea per `strsep`).

14.2 Versioni 1.x

14.2.1 Versione 1.0: inter. comandi interni ed esterni

In questa versione della shell vengono riconosciuti ed implementati alcuni comandi interni: `cd`, `pwd`, `help`, `ls`, `copy`, `echo`. Tutti gli altri comandi sono considerati esterni, e vengono quindi implementati con una chiamata `execvp` come nella versione 0.1.

Ogni comando interno è implementato da una funzione della forma

```
int nomecomando(int argc, char **argv);
```

in cui `argc` è il numero di parametri e `argv` è il vettore dei parametri, ed il tipo ritornato è un `int`. Si noti che questo prototipo è esattamente quello del `main` di un programma C. Infatti, molti dei comandi interni sono ottenuti semplicemente rinominando il `main` di un qualche programma C. Ad

esempio, la funzione per il comando `ls` è ottenuta a partire dal programma `lsdir` presentato nella Sezione 10.6.1. L'implementazione dei comandi è riportata nel file `comandi.c`, mentre il ciclo di lettura comando / richiamo della funzione è riportato nel file `shell10.c`.

I nomi dei comandi sono memorizzati in un vettore di stringhe:

```
char *nomecomandi[] = {cd, pwd, help, ls, copy, echo, NULL};
```

con, in particolare, l'ultimo elemento pari a `NULL`. Gli indirizzi delle funzioni che implementano i comandi sono memorizzati in un *vettore di puntatori a funzioni con argomenti un intero e un vettore di stringhe e con risultato un intero* (questo è possibile perché tutte le funzioni hanno gli stessi tipi di parametri e di risultato)

```
int (*nomefunzioni[])(int, char **) = {cd, pwd, help, ls, copy, echo, esterno};
```

con, in particolare, l'ultimo elemento pari alla funzione che implementa i comandi esterni.

Con queste codifica, il comando immesso viene prima diviso in token utilizzando la funzione `split_arg`. Quindi viene richiamata una funzione `interpreta`, la quale ricerca il primo token (ovvero il nome del comando) nel vettore `nomecomandi`. Quindi viene eseguita la funzione corrispondente al nome del comando, ovvero se il comando è l'`i`-esimo in `nomecomandi`, viene chiamata la funzione

```
nomefunzioni[i](argc, argv);
```

dove `argc` e `argv` sono il risultato della `split_arg`.

Riportiamo di seguito una sessione d'uso. Dal momento che `cd` e `pwd` sono comandi interni, è ora possibile navigare tra le directory del file system.

```
> shell10
S=0 C=1> pwd
/l/disc2/home/ruggieri/LabIV/SYSCALL/SHELL/V1.0
S=0 C=2> cd ..
S=0 C=3> pwd
/l/disc2/home/ruggieri/LabIV/SYSCALL/SHELL
S=0 C=4> help
Universita' di Pisa
Shell del Corso di Laboratorio IV
S=0 C=5> echo Laboratorio IV
Laboratorio IV
S=0 C=6> cd V1.0
S=0 C=7> ls
./      ../     makefile        util.c  util.h  sysmacro.h      shell10.o
comandi.c        shell10.c       comandi.o       util.o  shell10
S=0 C=8> cp makefile makefile1
S=0 C=9> copy makefile makefile2
S=0 C=10> diff makefile1 makefile2
S=0 C=11>
```

Esercizi

1 Si aggiunga il comando interno `head n filename`, il quale stampa le prime `n` linee del file `filename`.

14.2.2 Versione 1.1: inter. comandi interni ed esterni, sequenze di comandi.

In questa versione vogliamo aggiungere alla nostra shell la funzionalità di eseguire *sequenze* di comandi. Le modifiche alla versione 1.0 sono in realtà circoscritte. Non abbiamo, infatti, necessità di modificare l'interpretazione dei comandi (il file `comandi.c` rimane lo stesso della versione 1.0).

Invece, quello che vogliamo modificare è il modo in cui tali comandi sono richiamati. Mentre nella versione 1.0 il comando veniva suddiviso in token corrispondenti al nome del comando ed ai suoi parametri, in questa nuova versione il comando viene anzitutto suddiviso in token separati da `;`: *ognuno di tali token è un comando*! Per ciascuno dei comandi così individuati verrà richiamata la funzione di interpretazione dei comandi.

Riportiamo di seguito una sessione d'uso con una sequenza di comandi interni / esterni.

```
> shell11
S=0 C=1> echo Universita' di Pisa; ps ; ls
Universita' di Pisa
  PID TTY          TIME CMD
31239 pts/2    00:00:00 csh
31375 pts/2    00:00:03 emacs
31505 pts/2    00:00:00 shell11
31506 pts/2    00:00:00 ps
./      ../     makefile        util.c  sysmacro.h      util.h
comandi.c      shell11.c        shell11
S=0 C=2>
```

Esercizi

1 Si aggiunga l'interpretazione del metacarattere `*`, in modo che `comando *` venga interpretato come `comando file1 ... filen` dove `file1`, ...`filen` sono i file della directory corrente.

14.2.3 Versione 1.2: inter. comandi interni ed esterni, sequenze di comandi, comandi in background

Un ulteriore funzionalità consiste nell'esecuzione in background di un comando. Anche in questo caso non abbiamo la necessità di modificare l'interpretazione dei comandi (il file `comandi.c` rimane lo stesso della versione 1.0), ma solo il modo in cui tali comandi sono richiamati.

Come nella versione 1.1 il comando immesso viene suddiviso in token separati da `;`, ovvero in sequenze di comandi. Per ogni comando, prima di richiamare la funzione di interpretazione, viene verificato se questo termina con il token `&`. In caso affermativo, la shell effettua una `fork`. Il processo figlio procede con l'interpretazione del comando (escluso il token finale `&`) e quindi termina. Il processo padre ritorna ad accettare il prossimo comando (o a eseguire il prossimo comando nella sequenza). Il riconoscimento della terminazione del figlio avviene in modo asincrono: ad ogni ciclo di lettura comando viene richiamata una funzione `check_fine_proc` che verifica l'eventuale terminazione di uno dei processi in background.

Riportiamo di seguito una sessione d'uso con una sequenza di comandi interni / esterni lanciati in background.

```
> ./shell12
S=0 C=1> sleep 10 & ; ps
>31618<
  PID TTY          TIME CMD
31239 pts/2    00:00:00 csh
31605 pts/2    00:00:01 emacs
31617 pts/2    00:00:00 shell12  ... shell padre
31618 pts/2    00:00:00 shell12  ... shell figlio
31619 pts/2    00:00:00 ps
31620 pts/2    00:00:00 sleep    ... comando esterno in esecuzione
S=0 C=2>                         ... premo [INVIO]
>31618 terminato con stato 0<    ... riconosce terminazione shell figlio
S=0 C=3> > ps
  PID TTY          TIME CMD
31239 pts/2    00:00:00 csh
31605 pts/2    00:00:01 emacs
31617 pts/2    00:00:00 shell12
31626 pts/2    00:00:00 ps
S=0 C=4>
```

Esercizi

1 La priorità degli operatori `;` e `&` nella nostra shell è diversa da quella della `tcshell`. Nella nostra shell, il comando `ls ; ps &` viene interpretato come `ls ; (ps &)`, mentre nella `tcshell` viene interpretato come `(ls ; ps) &`. Modificare `shell12.c` in modo che rispetti la priorità secondo la convenzione della `tcshell`.

2 Modificare la shell in modo che riconosca la terminazione dei processi figli in modo sincrono utilizzando le chiamate di gestione dei segnali.

14.2.4 Versione 1.3: inter. comandi interni ed esterni, sequenze di comandi, comandi in background, ridirezione

L'ultima aggiunta alle funzionalità della nostra shell consiste nella possibilità di ridirigere lo standard output di un comando verso un file. Anche in questo caso non abbiamo la necessità di modificare l'interpretazione dei comandi (il file `comandi.c` rimane lo stesso della versione 1.0), ma solo il modo in cui tali comandi sono richiamati.

Come nella versione 1.1 il comando immesso viene suddiviso in token separati da `;`, ovvero in sequenze di comandi. Per ogni comando, prima di richiamare la funzione di interpretazione, viene verificato se questo termina con il token `&` e vengono prese le azioni descritte nella versione 1.2. La modifica per introdurre la ridirezione avviene quindi nella funzione di interpretazione dei comandi, la quale ora considera la possibilità che gli ultimi due token siano» e un nome di file su cui redirigere l'output del programma. In questo caso, il comando viene interpretato con lo standard output ridiretto opportunamente. Al termine viene ripristinato lo standard output originario.

Riportiamo di seguito una sessione d'uso con una sequenza di comandi interni / esterni ridiretti su file.

```
> shell13
S=0 C=1> cat > a
Laboratorio IV
S=0 C=2> cat a
Laboratorio IV
S=0 C=3> ls > a
S=0 C=4> cat a
./      ../     makefile        util.c  sysmacro.h      util.h  comandi.c
shell13.c        shell13.o       makefile~       shell13.c~      comandi.o
util.o   shell13 b       a
S=0 C=5>
```

Esercizi

1 Modificare `shell13.c` in modo che ridiriga lo standard input.

2 Modificare `shell13.c` in modo che ridiriga lo standard output accodandolo alla fine di un file.

3 Modificare `shell13.c` in modo che interpreti il pipelining di due comandi, come, ad esempio, in `> ls | wc`.

Indice analitico