Sistemi Operativi Lab

Appunti a cura di Liccardo Giuseppe
Università degli Studi di Napoli "Parthenope"

Indice – Lab

CAPITOLO 1. Panoramica su Unix

Unix è un sistema operativo multiutente e multitasking, ovvero più utenti possono avere vari processi in esecuzione contemporaneamente. Esso è anche un ambiente di sviluppo software che fornisce editor, compilatori, assemblatori, interpreti, ecc.

1.1 Architettura di Unix

Un sistema operativo può essere definito come un software che controlla le risorse hardware del computer e fornisce un ambiente nel quale eseguire i programmi. Tale software è chiamato ***kernel***, relativamente piccolo. L'interfaccia al kernel è uno strato di software chiamato ***system call*** (*chiamate di sistema*).ù

Quando si accede ad un sistema Unix, la prima cosa da fare è l'autenticazione, ovvero fornire al sistema la nostra username e la nostra password. Una volta fatta questa operazione si può iniziare a dare comandi al sistema, utilizzando un interprete dei comandi, la ***shell***. Questo interprete accetta l'input dell'utente ed esegue i comandi. Tra le shell più utilizzate ci sono la C shell e la Bourne shell.

Quando un programma viene mandato in esecuzione, la shell apre tre descrittori, 0 per lo standard input, 1 per lo standard output, 2 per lo standard error. Di norma, questi descrittori, sono collegati con il terminale ma è, tuttavia, possibile ridirigerli tutti verso un file (es: ls > file.txt).

Il file system di Unix è un insieme di file e directory organizzato in maniera gerarchica. La radice del file system, ***root***, è una directory di nome /. Questa directory è una tabella che contiene un nome e un puntatore ad una struttura di informazioni, per ciascun file o directory in essa contenuto.
Gli attributi di questa tabella riguardano il tipo di file, la dimensione, il proprietario, i permessi, ecc.

1.2 Programmi e processi

Un ***programma*** è un file eseguibile che risiede nel file system. Viene caricato in memoria ed eseguito dal kernel quando viene eseguita una chiamata ad una delle funzioni di *exec*.
Un programma in esecuzione viene detto ***processo***, ciascuno dei quali è identificato da un intero non negativo. Ci sono tre funzioni principali per il controllo dei processi: *fork*, *exec*, *waitpid*.

1.2.1 Identificazione degli utenti

Ciascun utente appartiene ad un gruppo il cui identificativo viene detto group ID.
Più utenti possono appartenere allo stesso gruppo. Lo scopo dei gruppi è quello di raggruppare utenti per far condividere le risorse.

1.2.2 Segnali

Una tecnica per notificare ad un processo l'occorrenza di una certa situazione è quella di utilizzare i segnali.
Un processo che riceve un segnale può:

- ignorare il segnale
- lasciare che venga eseguita l'azione di default
- fornire una funzione da eseguire all'atto della ricezione del segnale

1.2.3 Chiamate di sistema

Tutti i sistemi operativi forniscono interfacce attraverso cui i programmi richiedono servizi. Tutte le implementazioni di Unix forniscono un insieme di ***chiamate di sistema***.
Queste funzioni mettono a disposizione del programmatore dei servizi che vengono poi eseguiti dal kernel.
In particolare, ogni processo che richiede l'esecuzione di una system call, trasferisce il controllo al kernel, che effettua il servizio.

Le chiamate di sistema sono una sorta di "entry point" per il kernel. Il programmatore chiama la funzione usando la sintassi usuale delle funzioni C.

Le chiamate di sistema possoo essere raggruppate in tre categorie principali:
- gestione dei file
- gestione degli errori
- gestione dei processi

1.2.4 Funzioni di libreria

Sopra il livello delle chiamate di sistema è collocata una collezione di librerie C, il cui ruolo fondamentale è quello di fornire un'interfaccia C per le chiamate di sistema.
Queste funzioni, inoltre, forniscono servizi di utilità generale al programmatore. Non sono "entry point" del kernel.

CAPITOLO 2. Introduzione alla Shell

La Shell è un interprete di comandi. I comandi digitati dal'utente vengono letti dalla shell, interpretati e inviati al kernel per essere eseguiti.

L'utente può digitare i comandi mediante riga di comando (prompt dei comandi).
Il prompt dell'utente root (*superuser*) inizia col carattere #, mentre il prompt di un utente qualsiasi con $.
Quando si digitano questi comandi bisogna rispettare delle regole, come il rispetto delle lettere maiuscole e minuscole e l'inserimento corretto degli spazi per separare i vari comandi.

Solitamente un comando è composto da:
nome del comando *opzioni (facoltative)* *argomenti (facoltativi)*

2.1 I comandi della shell

Descriviamo brevemente alcuni dei comandi basilari della shell.

2.1.1 LS

Il comando LS è utile per avere l'elenco dei file contenuti nella directory corrente.

2.1.2 CP (copy)

Il comando CP serve per copiare un file da una directory ad un'altra.
Esempio:

```
cp pluto directory_prova
```

2.1.3 MV (move)

Il comando MV serve per spostare o a rinominare un file.
Esempio:

```
mv pluto directory_prova
```

2.1.4 RM (remove)

Il comando RM serve per rimuovere dei file o directory.
Esempio:

```
rm pluto          //rimuove il file pluto
rm –r agenda      //rimuove la directory agenda con tutto il suo contenuto
```

2.1.5 SU (set user)

Il comando SU serve per selezionare un utente. Dopo aver eseguito il comando, viene richiesta la password.
Esempio:

```
su mario    //setta l'utente mario
su root     //setta il superutente
```

2.1.6 CHMOD

Questo comando permette di cambiare i permessi di accesso ad un file.

2.1.7 PIPELINE

Un operatore importante nella shell è la pipeline (|). Serve per combinare insieme comandi diversi, dove l'output del primo comando diviene l'input del secondo comando. In pratica, con questo operatore, si creano una sorta di comandi in cascata in cui l'uscita di uno è l'ingresso del successivo.
Esempio:

ls –l | less

2.2 I link

I link sono scorciatoie per accedere a file o directory. Permettono di avere più di un punto di accesso per lo stesso file o directory.
Sono implementate diversamente in Windows e in Unix.

2.2.1 Implementazione di Windows

In Windows viene creato un file che contiene un certo numero di informazioni tra cui il nome del file a cui ci si riferisce. Per questo motivo, è possibile un solo livello di collegamento.

2.2.2 Implementazione di Unix

In Unix sono possibili più livelli di collegamento, infatti esistono due tipi di link: *soft link* e *hard link*.

I ***soft link*** sono molto simili ai link utilizzati da Windows. Il nuovo collegamento contiene solo il nome del riferimento. Un soft link corrisponde ad un link che contiene il path_name del file collegato.
In questo caso, la cancellazione del soft link non comporta la modifica del file originale. Tuttavia, la cancellazione del file, rende il file inaccessibile a tutti i soft link che puntano al file stesso.

Gli ***hard link*** permettono di accedere ad un file utilizzando un nuovo *path_name*. Se utilizzo un hard link i due file fanno riferimento allo stesso i-node.
Il numero di hard link che fanno riferimento ad un i-node è memorizzato nel campo *count* dell'i-node stesso.
Quando un utente cancella un hard link al file, il campo *count* viene decrementato. Un file viene effettivamente cancellato solo quando sono stati eliminati tutti gli hard link a questo file (cioè quando il campo *count* è uguale a 0).

Esempio:
nuovo è un soft link a **vecchio**

- possono usare indifferentemente **nuovo** o **vecchio** per modificare i miei dati
- se cancello **vecchio** perdo i miei dati, **nuovo** rimane, ma nel momento in cui lo si utilizza si ottiene un errore (è un puntatore dangling, appeso)
- se cancello **nuovo** perdo solo un modo di accedere ai dati

nuovo è un hard link a **vecchio**

- posso usare indifferentemente **nuovo** o **vecchio** per modificare i miei dati
- se cancello **vecchio** posso ancora accedere ai dati tramite **nuovo**
- se cancello **nuovo** posso ancora accedere ai dati tramite **vecchio**
- se cancello **vecchio** e poi lo ricreo, ho ora due insieme di dati diversi

CAPITOLO 3. Il linguaggio C

In questo capitolo vedremo qualche cenno / ripasso delle principali operazioni riguardanti il C.

3.1 Compilazione di un programma C

Ogni versione di Unix ha un compilatore standard per il linguaggio C, generalmente chiamato **cc**.
Nel caso di GNU Linux è presente **gcc** (GNU C Compiler) conforme allo standard POSIX.

La sintassi da utilizzare nella riga di comando è la seguente:

gcc <opzioni> <file> *gcc progamma.c (1)* *gcc programma.c –o software (2)*

Il comando (1) compila il programma sorgente *programma.c* generando l'eseguibile *a.out*, mentre il comando (2) permette di scegliere il nome dell'eseguibile, *software*, grazie all'utilizzo dell'opzione –o.

3.1.1 Programma in un singolo modulo

Realizzare un programma come un singolo modulo presenta vari inconvenienti:

- ogni minima modifica richiede la ricompilazione dell'intero programma, quindi si hanno tempi di compilazione elevati
- non è facile riutilizzare le funzioni (di utilità generale) definite nel programma in altri programmi

3.1.2 Programma suddiviso in più moduli

Per le ragioni appena elencate si preferisce suddividere un programma C in più file sorgenti che vengono compilati indipendentemente e che sono collegati in un unico eseguibile.
In questo modo, le funzioni possono essere riutilizzato da diversi programmi e le eventuali modifiche da apportare al programma richiede la compilazione solo del modulo o dei moduli modificati e non dell'intero programma.

In definitiva, ogni file sorgente può essere compilato separatamente:

gcc –c modulo.c
gcc –c usa_modulo.c

In questo modo si generano i file oggetto *modulo.o* e *usa_modulo.o*. Successivamente occorre effettuare il linking dei file oggetto per creare il file eseguibile *file_exe*:

gcc modulo.o usa_modulo.o –o file_exe

Per poter utilizzare questo metodo occorre tener presente che:

- se un file utilizza un'altra funzione, non definita nello stesso file, deve contenere la dichiarazione del prototipo della funzione
- per rendere un modulo (esempio: modulo.c) facilmente riusabile si predispone un header file (esempio: modulo.h), cioè il modulo *modulo.c* include il proprio header *modulo.h*
- ogni file che utilizzare il modulo include l'header del modulo stesso

3.2 Programmazione di sistema

Per utilizzare i servizi offerti da Unix, come la creazione di file e la comunicazione tra processi, i prgrammi devono interagire col Sistema Operativo utilizzando un insieme di routine dette **system call**, che costituiscono l'interfaccia del programmatore col kernel di Unix. In pratica, le system call costituiscono un

"entry point" per il kernel ed il programmatore può richiamare queste funzioni usando la sintassi usuale delle funzioni C.

Le chiamate di sistema possono essere raggruppate in tre categorie principali:
- gestione dei file
- gestione degli errori
- gestione dei processi

3.2.1 Processi

Un processo è un programma in esecuzione, costituito da istruzioni e dati e memorizzato in un file.
Solitamente si compone di:
- introduzione
- funzione main ()
- terminazione di un processo
- argomenti da linea di comando
- environment list
- struttura della memoria di un programma C
- librerie condivise
- allocazione della memoria

INTRODUZIONE
Quando si esegue un programma si esegue prima una routine di start-up speciale che prende i valori passati dal kernel in *argv[]* dalla linea di comando e le variabili di ambiente. Successivamente viene chiamata la funzione main.

FUNZIONE MAIN ()
Un programma C inizia l'esecuzione con una funzione chiamata main, il cui prototipo è:

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

dove:
- ***argc*** è il numero di argomenti
- ***argv*** è un array di puntatori agli argomenti

TERMINAZIONE DI UN PROCESSO
Fondamentalmente un processo può terminare in due modi: *terminazione normale* e *terminazione anomala*.

Complessivamente un processo può terminare in otto modi diversi:
- ***terminazione normale***: ritorno dal main, chiamata di exit, chiamata di _exit o _Exit, chiamata di p_thread_exit, ritorno dall'ultimo thread
- ***terminazione anomala***: chiamata di abort, ricezione di un segnale, risposta dell'ultimo thread ad una richiesta di cancellazione

Come si può notare, sono tre le funzioni di uscita che terminano un programma normalmente:
- *_exit* ed *_Exit* – system call – ritornano al kernel immediatamente
- *exit* – libreria standard – prima esegue una procedura di pulizia e poi ritorna al kernel

Tutte e tre le funzioni ricevono un argomento intero (*exit status*).

ARGOMENTI DALLA LINEA DI COMANDO
Quando un programma viene eseguito, da linea di comando è possibile passare al programma stesso vari argomenti che sono "gestiti" mediante *argc* e *argv*.

Esempio:

```
#include "apue.h"
int main (int argc, char *argv[])
{
  int i;
  for (i=0; i<argc; i++)
     printf("argv[%d]: %s\n",i,argv[i]);
  exit(0);
}
```

```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

ENVIRONMENT LIST

Ad ogni programma è passata una lista dell'ambiente.

STRUTTURA DELLA MEMORIA DI UN PROGRAMMA C

Un programma C è composto dai seguenti pezzi:

- *segmento di testo*
- *segmento di dati inizializzato*
- *segmento di dati non inizializzato*
- *stack*
- *heap*

LIBRERIE CONDIVISE

Rimuovono le routine delle librerie comuni dai file eseguibili, mantenendo una copia singola della routine di libreria da qualche parte in memoria a cui i processi fanno riferimento.

ALLOCAZIONE DELLA MEMORIA

ISO C specifica tre funzioni per l'allocazione della memoria: *malloc, calloc, realloc.*
Per liberare la memoria allocata invece, è messa a disposizione la funzione *free*.

CAPITOLO 4. Processi

Un processo Unix si compone fondamentalmente di tre parti:

- area codice (contiene il codice)
- area dati (contiene i dati statici del processo)
- area stack (contiene i dati dinamici o "temporanei" del processo, come la environment list, gli argomenti della command line, lo stack e l'heap)

Unix associa ad ogni processo un PCB (Process Control Block) contenente, tra le varie info, anche il PID, il PPID (Parente PID), lo stato del processo (in esecuzione, sospeso, idle, zombie, ecc), user e group ID, segnali pendenti, ecc.

4.1 Gestione dei processi

All'avvio del Sistema Operativo c'è un solo processo utente visibile chiamato ***init()*** il cui identificativo numerico è sempre 1. Quindi *init* è l'antenato comune di tutti i processi utente esistenti in un dato momento nel sistema.

Mediante una chiamata alla system call ***fork()***, in Unix, è possibile creare un nuovo processo. Per l'esattezza, con questa system call non si fa altro che duplicare un processo esistente. Quando viene creato un nuovo processo, il processo padre e il processo figlio sono identici:

- condividono lo stesso codice
- il figlio eredita l'ambiente di lavoro dal padre: file aperti, privilegi, ecc.
- i dati e lo stack del figlio sono una copia di quelli del padre

In pratica, essi differiscono solo per alcuni aspetti, come il PID e il PPID.

N.B. Un processo figlio può differenziarsi dal processo padre sostituendo il suo codice con quello di un altro file eseguibile, mediante una ***exec***.

Quando un processo figlio termina, la sua terminazione è comunicata al padre tramite un segnale.
Un processo padre si può sospendere (tramite una wait) in attesa della terminazione del figlio

4.1.1 Gestione dei processi

FORK
Quando viene invocata una fork:

- il ***processo figlio*** avrà pid = 0
- il ***processo padre*** avrà pid > 0
- se l'operazione fallisce verrà restituito valore -1

Esempio:

```
int pid;
pid = fork ()
if ( pid != 0) …    //pid diverso da 0, sono il padre
if ( pid = 0 ) …    //pid uguale a 0, sono il figlio
```

Se un padre termina prima di un suo figlio, il figlio diventa orfano e viene adottato dal processo init.

Non è possibile predire quale processo venga eseguito per primo, ciò dipende dall'algoritmo di scheduling utilizzato dal sistema.

GETPID – GETPPID
Esistono delle funzioni che consentono di ricavare il proprio PID e il PID del proprio padre, cioè il PPID.
La prima funzione è ***getpid*** che restituisce il PID del processo che la invoca.
La seconda funzione è ***getppid*** che restituisce il PID del padre.

In generale, se un processo padre vuole aspettare un proprio processo figlio per terminare, deve chiamare una delle funzioni wait.
Viceversa, se un processo figlio vuole aspettare il processo padre per terminare, dovrebbe eseguire un ciclo del tipo:

```
while ( getppid() != 1 )
        sleep (1)
```

Questo ciclo, chiamato ***polling***, spreca tempo di CPU. Per evitare il polling è necessario utilizzare i segnali o altre forme di IPC.

WAIT – WAITPID
La funzione ***wait*** sospende il processo corrente finchè un figlio termina o finchè il processo corrente riceve un segnale. Se il processo invocante ha più di un figlio, wait ritorna quando uno di questi ha terminato.

La funzione ***waitpid*** sospende il processo corrente finchè il figlio corrispondente al *pid* passato come argomento termina o finchè il processo corrente riceve un segnale.
L'argomento *pid* di waitpid può assumere questi valori:

- pid == -1 ; attende un qualsiasi figlio (come wait)
- pid > 0 ; attende il processo che il pid uguale al pid passato come argomento
- pid == 0 ; attende un qualsiasi figlio il cui group ID è uguale a quello del processo invocante
- pid < -1 ; attende un qualsiasi figlio il cui group ID è uguale a quello del valore assoluto di pid

EXIT
La funzione ***exit*** termina il processo chiamante. Inoltre ritorna il valore *status* al padre, che lo otterrà mediante wait.
Per rilevare lo stato sono utilizzate delle macro:

- WIFEXITED ; risulta vera se il figlio termina normalmente
- WEXITSTATUS ; riporta lo stato del figlio
- WIFSIGNALED ; risulta vera se il figlio è uscito a causa di un segnale
- WTERMSIG ; riporta il segnale che ha causato l'uscita del figlio
- WIFSTOPPED ; risulta vera se il figlio è fermato
- WSTOPSIG ; ritorna il segnale che ha causato la sospensione del figlio

4.1.2 Processi zombie e processi adottati

Un processo che termina non scompare dal sistema fino a che il padre non accetta il suo codice di terminazione. Un processo che sta aspettando che il padre accetti il suo codice di terminazione è chiamato ***processo zombie***.
Se un processo padre termina prima di un processo figlio, quest'ultimo processo viene detto ***orfano*** e, grazie al kernel, viene adottato da init (cioè il loro PPID sarà 1).

4.1.3 Le funzioni EXEC

Un processo figlio condivide lo stesso codice del padre. Mediante degli IF è possibile "distinguere" le parti di codice eseguite dal figlio da quelle eseguite dal padre e/o dagli altri figli.
Se un processo figlio vuole eseguire un codice diverso deve ricorrere alla funzione ***exec*** che permette di caricare un altro programma.

Quando un processo chiama una funzione della famiglia exec, la exec sostituisce il programma chiamato al processo corrente, sovrascrivendone il codice, i dati, l'heap e lo stack. Il nuovo programma, inoltre, eredita dal processo chiamante il PID.

Sono 6 le versioni della exec che possono essere usate, in realtà sono tutte derivanti dalla *execve*.

```
int execl(const char *path, const char *arg0, .../* (char *)0 */);

int execle(const char *path, const char *arg0, ... /* (char *)0, char
  *const envp[] */);

int execlp(const char *file, const char *arg0, .../* (char *)0 */);

int execv(const char *path, char *const argv[]);

int execve(const char *path, char *const argv[], char *const envp[]);

int execvp(const char *file, char *const argv[]);
```

Nel nome della funzione chiamata sono contenuti dei caratteri che indicano sia il tipo di argomenti accettati sia il tipo di comportamento della funzione stessa:

- **L** ; la funzione accetta una lista di argomenti, il primo è il nome del programma che verrà eseguito, all'ultimo argomento valido deve seguire un puntatore a NULL
- **V** ; la funzione accetta un vettore di argomenti, il primo elemento (argv[0]) deve contenere il nome del programma da invocare, l'ultimo argomento valido deve essere eseguito da un elemento contenente un puntatore a NULL
- **E** ; la funzione accetta fra i suoi parametri un array di stringhe dell'ambiente (environment) che deve essere passato al pathname da eseguire. L'array envp[] è terminato da un elemento a NULL. Le funzioni della famiglia che non contengono la E nel nome ereditano l'ambiente del processo attuale
- **P** ; in questo caso, se il pathname non contiene il carattere slash, l'argomento viene considerato come un nome di programma che viene ricercato nella lista di directory specificate da PATH. Se PATH non è specificata, la ricerca avviene per default nelle directory /bin e /usr/bin

4.2 Commento del codice FORK

4.2.1 Codice 1

```
int glob=5;
int pid=0;
pid=fork();
glob--;
pid=fork();
glob--;
if (pid!=0) {
     pid=fork();
     glob--;
}
printf("Valore di glob=%d\n",glob);
```

Dichiariamo due variabili intere *glob* e *pid*.

PASSO 1
Il padre, DOMENICO (5), esegue la fork e crea il processo figlio GIUSEPPE che eredita glob=5 dal padre. A questo punto, entrambi decrementano glob (5 → 4).
DOMENICO (4) – GIUSEPPE (4)

PASSO 2
Sia DOMENICO (4) che GIUSEPPE (4) eseguono una ulteriore fork. DOMENICO da vita a DANIELE, mentre GIUSEPPE da vita a GIOLANDA.
DANIELE e GIOLANDA ereditano glob=4.
Tutti e quattro decrementano il valore di glob (4 → 3).
DOMENICO (3) – GIUSEPPE (3) – DANIELE (3) – IOLE (3)

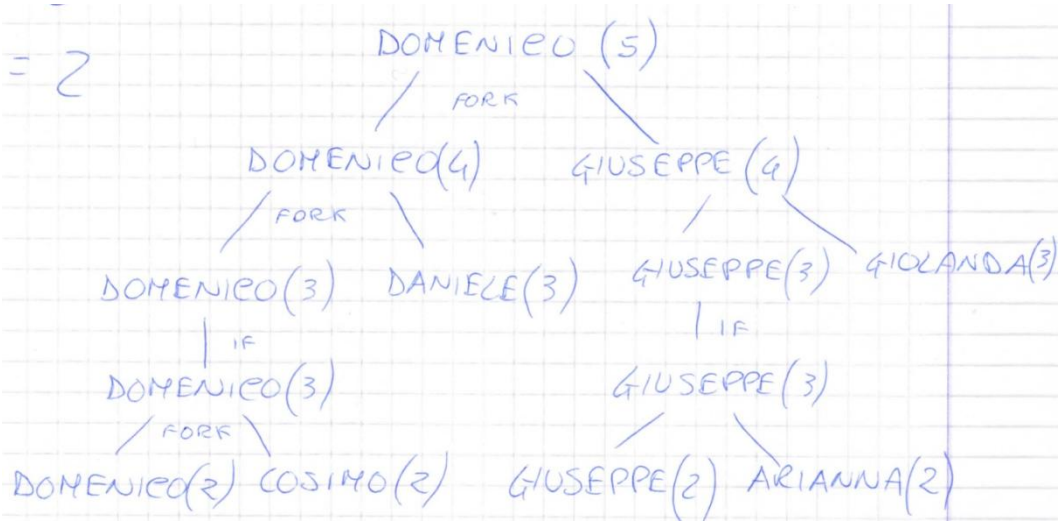
PASSO 3 (IF)
Nell'IF entrano solamente i padri perché il valore restituito dalla fork è diverso da 0.
N.B. il valore restituito dalla fork viene assegnato alla variabile pid, che per i padri sarà uguale al pid del figlio, quindi maggiore di 0, mentre per i figli sarà uguale a 0.
I padri che entrano nell'IF sono dunque DOMENICO e GIUSEPPE.
DANIELE e GIOLANDA saltano l'IF e stampano il valore di glob (3) e terminano.

DOMENICO e GIUSEPPE fanno un'altra fork, creano rispettivamente COSIMO e ARIANNA che ereditano glob=3. Tutti e quattro decrementano glob (3 →2) e vanno a stampare tale valore.

DOMENICO (2)
GIUSEPPE (2)
DANIELE (3)
COSIMO (2)
GIOLANDA (3)
ARIANNA (2)

Il funzionamento è il seguente:
- i numeri affianco ai nomi indicano il valore di glob
- le biforcazioni indicano la fork
- il segmento verticale indicano l'ingresso nell'IF

Schema di funzionamento:



4.2.2 *Codice 2*

```
int glob=20;
int pid=0;
int main() {
  int i=0;
  for (i=2;i<4;i++) {
      pid=fork();
      if (pid==0) {
            glob=glob*2;
            sleep(i+1);
      }
  glob=glob-1;
  printf("Valore di glob=%d\n",glob);
  }
}
```

Padre A: il padre (cioè il main) entra nel ciclo (i=2) e crea il primo figlio (chiamato B), non entra nell'IF perché il PID ≠ 0, decrementa glob da 20 a 19 e lo stampa. Risale, esso esegue un'altra iterazione nel FOR perché i=3 e crea un altro figlio (chiamato C), non entra nell'IF perché il suo PID ≠ 0 (a causa dell'istruzione pid=fork();) quindi decrementa glob da 19 a 18.
Il padre non entra più nel ciclo perché i=4.

Figlio B: quando B viene creato i=2 e glob=20. Abbiamo detto che il primo processo figlio che crea è B, entra nell'IF (perché PID = 0), assegna glob uguale a 20*2 = 40 e aspetta per 2+1=3 secondi; siccome non c'è nessuna exit dopo lo sleep, esegue le operazioni sottostanti: decrementa glob da 40 a 39, lo stampa e risale sopra. Ricomincia il FOR (perché i=3), crea un figlio (chiamato D), non va nell'IF (perché il suo PID ≠ 0), decrementa glob a 38 e lo stampa, dopodiché risale sopra. Visto che non rientra nel FOR perché i=4, termina.

Figlio C: quando C viene creato i=3 e glob=19. C è un altro figlio creato da processo iniziale A. Il figlio C entra nell'IF e assegna glob uguale a 19*2 = 38, dopodiché aspetta per 3+1=4 secondi, decrementa glob a 37 e stampa 37; dopodiché risale. Non rientra nel FOR perché i=4, quindi finisce.

Figlio D: quando D viene creato i=3 e glob=39. D è il figlio creato dal figlio B. Entra nell'IF e assegna glob uguale a 39*2 = 78, dopodiché aspetta per 3+1=4 secondi, decrementa glob=77, lo stampa e finisce. Non rientra nel FOR in quanto i=4.

CAPITOLO 5. Segnali

Un segnale è un interruzione software.
Gli eventi che possono verificarsi durante il normale funzionamento del sistema, devono essere comunicati ai processi. Questi eventi possono essere di varia natura e sono asincroni, ovvero possono verificarsi in qualsiasi momento.

Ciascun segnale ha un nome che inizia son SIG. I nomi dei segnali sono definiti in *'signal.h'* e si può far riferimento ad essi sia tramite il nome sia tramite un codice numerico associato ad ognuno di essi.

Un processo non si può limitare a controllare se un segnale si è verificato, ma deve anche comunicare al kernel cosa fare in caso di occorrenza di uno specifico segnale.
Un processo può intraprendere tre azioni quando si verifica un evento:

- ***ignorare il segnale*** ; il processo può ignorare il segnale, a meno che non si tratti di SIGKILL e SIGSTOP che sono segnali che non possono essere ignorati
- ***catturare il segnale*** ; significa scrivere un signal handler (gestore dei segnali) che viene attivato quando viene ricevuto uno specifico segnale dal processo
- ***eseguire le azioni di default*** ; ogni segnale ha un'azione di default ad esso associata, quindi se non si specifica nulla, il processo eseguirà quest'azione all'occorrenza del segnale (solitamente l'azione di default consiste nella terminazione forzata del processo)

Con il termine ***disposizione*** per un segnale si intende l'indicazione al kernel su cosa fare in seguito all'occorrenza del segnale.

5.1 Segnali di Unix

Descriviamo brevemente alcuni dei segnali più comuni di Unix.

SIGFPE: viene generato quando viene effettuata una divisione per zero. L'azione di default è la terminazione del processo e la generazione di un file core
SIGHUP: viene generato quando il terminale di controllo viene chiuso. L'azione di default è la terminazione del processo
SIGINT: viene generato quando si premono CTRL+C. L'azione di default è la terminazione del processo.
SIGKILL: è un segnale che può essere mandato dal proprietario o da root. Non può essere ignorato.
SIGSEGV: viene generato quando il processo tenta di accedere a memoria al di fuori del suo segmento. L'azione di default è la terminazione del processo e la generazione di un file core.
SIGUSR1 – SIGUSR2: Sono segnali definiti dall'utente. L'azione di default è la terminazione del processo.

5.2 Alcune system call

5.2.1 System call signal()

La system call ***signal()*** permette di stabilire cosa fare quando il processo riceve un segnale. Le azioni possibili sono tre:

1) ignorare il segnale (SIG_IGN)
2) eseguire le azioni di default (SIG_DFL)
3) passare il controllo al signal handler, che non è altro che una funzione utente

In caso di successo, la funzione signal ritorna le disposizioni precedenti per il segnale.

*void (*signal (int signum, void (*sighandler) (int))) (int);*

Questa system call ha due argomenti e ritorna un puntatore ad una funzione che non ritorna nulla (void).

signum è un intero o il nome del segnale occorso
sighandler è un puntatore ad una funzione void che prende come argomento un intero. Il valore di questo puntatore è:

- la costante SIG_IGN se si vuole ignorare il segnale
- la costante SIG_DFL se si vuole eseguire l'azione di default
- l'indirizzo di una funzione utente

N.B. E' consigliato scrivere dei signal handler ridotti all'osso, cioè meno compiti esegue il signal handler, meglio è! Si evitano incongruenze!

Esempio:

```
#include <signal.h>

int main(void)
{
   signal (SIGINT, SIG_IGN);
   while (1);
}
```

In pratica, quando si verifica il segnale SIGINT, l'azione che deve intraprendere il processo è quella di ignorare il segnale (SIG_IGN).
In particolare, SIGINT viene generato quando si preme CTRL+C, quindi in questo esempio non è possibile interrompere l'esecuzione del programma premendo CTRL+C in quanto viene ignorato.

5.2.2 *System call kill() e raise()*

Tramite la system call ***kill()*** è possibile inviare un segnale *signum* ad un processo *pid*.
Tramite la system call ***raise()*** è possibile inviare un segnale *signum* al processo invocante (cioè a se stesso).
Entrambe le funzioni restituiscono 0 se tutto è andato a buon fine, restituiscono -1 se si è verificato un errore.

int kill (pid_t pid, int signum)

pid può assumere diversi valori:

- > 0 il segnale *signum* è inviato al processo avente PID = *pid*
- = 0 il segnale *signum* è inviato ad ogni processo nel process group del processo corrente
- < 0 il segnale *signum* è inviato ad ogni processo che ha il process group uguale al valore assoluto di *pid*

5.2.3 *Alarm() – Pause() – Sleep ()*

La funzione ***alarm()*** invia al processo corrente il segnale SIGALRM dopo che siano trascorsi *num_seconds* secondi. Se *num_seconds* è 0, non viene attivato nessun allarme e viene disabilitato l'eventuale allarme precedentemente impostato.
Questa funzione ritorna il numero di secondi rimasti all'invio del segnale SIGALRM.

int alarm (int num_seconds)

La funzione ***pause()*** sospende il processo chiamante finchè non è intercettato un segnale

int pause (void)

La funzione ***sleep()*** sospende il processo chiamante fino a che trascorre la quantità di tempo specificata da *num_seconds* o finchè il processo intercetta un segnale che non sia ignorato.

int sleep (int num_seconds)

5.3 Funzioni rientranti

Quando un segnale viene gestito da un processo, la normale sequenza di esecuzione eseguita dal processo viene interrotta temporaneamente dal signal handler. Il processo, successivamente, continuerà la sua esecuzione dopo l'esecuzione delle istruzioni del signal handler.
Se il signal handler ritorna (invece di chiamare una exit, per esempio), allora continua la normale sequenza delle istruzioni che il processo stava eseguendo nel momento in cui è stato intercettato il segnale.
Nel signal handler non possiamo dire in che punto del codice si trovava il processo al momento dell'intercettazione del segnale.

Una ***funzione rientrante*** è una funzione che può essere interrotta da più di un processo in modo concorrente senza corrompere i dati. In pratica essa può essere interrotta in qualsiasi momento e riavviata in un momento successivo senza la perdita di dati. Per fare ciò, le funzioni rientranti usano *variabili locali* o *proteggono i propri dati* quando sono utilizzate le variabili globali. Inoltre non devono chiamare alcuna funzione non rientrante.

Una ***funzione non rientrante*** è una funzione che non può essere condivisa da più task a meno che non sia assicurata la mutua esclusione alla funzione usando un semaforo o disabilitando le interruzioni durante le sezioni critiche di codice.

CAPITOLO 6. Pipe

In Unix, la gestione della sincronizzazione riguarda due aspetti: scambio di dati e segnalazione di eventi.

Lo ***scambio di dati*** avviene attraverso:
- *pipe*: operazioni di I/O su code FIFO, sincronizzate dal SO
- *messaggi*: invio e ricezione di messaggi tipizzati su coda FIFO
- *memoria condivisa*: allocata e associata al processo attraverso system call

La ***segnalazione*** di azioni avviene attraverso:
- *segnali*: interrupt software
- *semafori*: una generalizzazione dei semafori classici

I processi che comunicano possono risiedere sulla stessa macchina e comunicano attraverso segnali, pipe, FIFO, socket oppure possono risiedere su macchine diverse e comunicano attraverso le socket.

Gli scopi della comunicazione sono essenzialmente la *cooperazione* per un obiettivo comune e la *sincronizzazione* dei processi per schedulare correttamente la propria attività.

6.1 Pipe anonime

Le pipe sono strumenti noti agli utenti Unix: una ***pipeline*** è il meccanismo utilizzato dalla shell per connettere l'output di un comando all'input di un altro. In questo meccanismo si memorizza automaticamente l'output dello scrittore in un buffer (solitamente di 4KB); se il buffer è pieno, lo scrittore si sospende fino a che alcuni dati non vengono letti; se il buffer è vuoto, il lettore si sospende fino a che diventano disponibili dei dati in output.

Una ***pipe anonima*** è un canale di comunicazione, mantenuto a livello kernel, che unisce due processi.
Questo canale:
- viene creato attraverso la funzione *pipe()*
- è unidirezionale
- permette la comunicazione solo tra processi con un antenato comune

Una pipe presenta due lati di accesso (in/out), ciascuno associato ad un descrittore di file.
Il lato di lettura è acceduto invocando *read()*.
Il lato di scrittura è acceduto invocando *write()*.
Quando un processo ha finito di usare un lato di una pipe chiude il descrittore con *close()*.

int pipe (int filedescrittore[2]);

La funzione *pipe()* crea una coppia di file descriptor, utilizzabili per una 'pipe'. I file descriptor vengono posti nell'array puntato da *filedescrittore*. In particolare:
- ***filedescrittore[0]*** è il descrittore utilizzabile per la lettura
- ***filedescrittore[1]*** è il descrittore utilizzabile per la scrittura

Se l'operazione va a buon fine, la funzione pipe() restituisce 0, altrimenti restituisce -1 se fallisce.

6.1.1 Pipe anonime: lettura

Se un processo legge da una pipe:
- se il lato scrittura è stato chiuso, dopo che sono stati letti tutti i dati, *read()* restituisce 0 che indica la fine dell'input

- se la pipe è vuota e il lato scrittura è ancora aperta, si sospende fino a che diventa disponibile qualche input
- se il processo tenta di leggere più byte di quelli presenti nel buffer associato, i byte disponibili vengono letti e *read()* restituisce il numero di byte effettivamente letti

6.1.2 Pipe anonime: scrittura

Se un processo scrive su di una pipe:

- se il lato di lettura è stato chiudo, *write()* fallisce (ritorna, quindi, il valore -1) ed allo scrittore è inviato un segnale SIGPIPE, la cui azione di default è la terminazione del processo
- se scrive meno byte di quelli che una pipe può contenere, *write()* viene eseguita in modo atomico (non possono avvenire intrecci dei dati scritti da processi diversi sulla stessa pipe)
- se scrive più byte di quelli che una pipe può contenere (PIPE_BUF), non c'è garanzia di atomicità

Inoltre, la *lseek()* non ha senso se applicata ad una pipe.

6.1.3 Pipe anonime: sequenza tipica

La tipica sequenza di eventi è:

- il processo crea una pipe anonima [*pipe()*]
- il processo crea un figlio [*fork()*]
- lo scrittore chiude il suo lato di lettura della pipe ed il lettore chiude il suo lato scrittura [*close()*]
- i processi comunicano usando *write()* e *read()*
- ogni processo chiude [*clode()*] il suo descrittore quando ha finito

Una comunicazione bidirezionale tra due processi si può realizzare utilizzando due pipe.

6.1.4 Pipe anonime: pipe e fork

Utilizzare una pipe in un processo singolo non ha senso. Normalmente, il processo che crea una pipe successivamente invoca una *fork()*, creando un canale di IPC dal genitore al figlio o viceversa.

Cosa accade dopo la *fork()* dipende dalla direzione del flusso di dati che vogliamo:

- per una pipe dal genitore al figlio, il genitore chiude l'estremità in lettura della pipe (*fd[0]*), ed il figlio chiude l'estremità in scrittura (*fd[1]*)
- per una pipe dal figlio al genitore, il genitore chiude l'estremità in scrittura (*fd[1]*), ed il figlio chiude l'estremità in lettura (*fd[0]*)

Esempio comunicazione padre → figlio:

```
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define SIZE 1024
int main(int argc, char*argv[]) {
  int pfd[2];
  int nread;
  int pid;
  char buf[SIZE];
  if (pipe(pfd)== -1) {
      perror("pipe() fallita)");
      exit(-1);
  }
  if((pid=fork()) < 0 ) {
      perror("fork() fallita");
      exit(-2);
  }
…
```

```
…
if(pid==0) { /* figlio */
   close(pfd[1]);
   while ( (nread=read(pfd[0], buf, SIZE)) != 0)
  {
      printf("il figlio legge: %s\n", buf);
  }
  close(pfd[0]);
} else { /* padre */
  close(pfd[0]);
  strcpy(buf,"Sono tuo padre!");
  write(pfd[1], buf, strlen(buf)+1);
  close(pfd[1]);
}
exit(0);
}
```

Esempio comunicazione figlio → padre:

```
/* talk.c */
#include <stdio.h>
#include <string.h>
#define READ 0
#define WRITE 1
char *frase = "Messaggio...";
int main() {
int fd[2], bytesRead;
char message[100];

pipe(fd);
if(fork() == 0) {
close(fd[READ]);
write(fd[WRITE],frase,strlen(frase)+1);
close(fd[WRITE]);
}
```

```
else {
close(fd[WRITE]);
bytesRead=read(fd[READ],message,
100);
printf("Letti %d byte: %s\n",
    bytesRead, message);
close(fd[READ]);
}
}


$ a.out
Letti 12 byte: Messaggio...
$
```

6.1.5 *Pipe anonime: comunicazione*

Quando un processo "scrittore" invia più messaggi di lunghezza variabile tramite una pipe, occorre fissare un protocollo di comunicazione che permetta al processo "lettore" di individuare la fine di ogni singolo messaggio. Alcune possibilità sono:

- inviare la lunghezza del messaggio (dato di dimensione fissa e nota) prima del messaggio stesso
- terminare un messaggio con un carattere speciale come '\0' o un *newline*

6.1.4 *Pipe anonime: pipe ed exec*

Supponiamo di voler implementare la pipeline:

ls | wc –w

dove:

- il padre esegue *fork()* per creare il figlio
- il padre esegue *exec()* per realizzare la *ls*
- il figlio esegue *exec()* per realizzare *wc –w*

Una *exec()* non altera la tavola dei descrittori ma si perdono le variabili locali, cioè lo spazio di memoria ereditato dal padre. Come fare per accedere alla stessa pipe se gli unici riferimenti comuni erano in variabili locali? La soluzione sta nell'utilizzo delle funzioni *dup()* e *dup2()*.

int dup (int vecchio_descrittore);
int dup2 (int vecchio_descrittore, int nuovo_descrittore);

FUNZIONI DUP() – DUP2()
Le funzioni *dup()* e *dup2()* creano una copia del descrittore *vecchio_descrittore*.
In particolare:

- la funzione *dup()* attribuisce al nuovo descrittore il più piccolo intero non usato
- la funzione *dup2()* crea *nuovo_descrittore* come copia di *vecchio_descrittore*

Dopo l'utilizzo di queste funzioni, è comunque possibile usare il vecchio e il nuovo descrittore interscambiabilmente.

In pratica, per accedere alle stesse pipe, occorre, nello specifico, utilizzare la funzione *dup2()* duplicando il descrittore che si vuole utilizzare nello standard input, nello standard output o nello standard error.

Se la situazione iniziale è la seguente:

standard input	*0*
standard output	*1*
standard errore	*2*
pfd[0]	*3*
pfd[1]	*4*

Le operazioni che dovranno essere implementate sono:

1) il padre effettua la *fork()*
2) il padre esegue *dup2 (pfd[1], 1)*, cioè duplica *pfd[1]* nello standard output; poi chiude *pfd[1]*
3) il padre realizza il comando *ls* effettuando la *exec ("ls", "ls", null)*
4) il figlio esegue *dup2 (pfd[0], 0)*, cioè duplica *pfd[0]* nello standard input; poi chiude *pfd[0]*
5) il figlio realizza il comando *wc –w* effettuando la *exec ("wc","wc", "-w", null)*

In generale, il processo padre:

- crea tante pipe quanti sono i processi figli meno uno
- crea tanti processi figli quanti sono i comandi da eseguire
- chiude tutte le estremità delle pipe
- attende la terminazione di ciascun processo figlio

Ciascun processo figlio, invece:

- chiude le estremità delle pipe che non usa
- imposta le estremità delle pipe che usa sugli opportuni canali standard
- invoca una funzione della famiglia *exec()* per eseguire il proprio comando

6.2 Pipe con nome – FIFO

Come abbiamo visto in precedenza, le pipe anonime consentono lo scambio di dati tra processi che hanno un antenato in comune. Per superare questa limitazione si può far uso delle ***pipe con nome*** o ***FIFO***.
Difatti le FIFO sono pipe che possono connettere due (o più) processi qualsiasi, agendo grossomodo come le pipe anonime.

In pratica per superare il problema dell'antenato comune lo standard POSIX ha definito dei nuovi oggetti, le FIFO, che hanno le stesse caratteristiche delle pipe anonime, ma che invece di essere strutture interne del kernel, visibili solo attraverso un file descriptor, sono accessibili attraverso un inode che risiede sul file system, così che i processi le possano usare senza dovere per forza essere in una relazione di parentela.

6.2.1 Pipe con nome: mkfifo()

Si possono creare pipe con nome dalla linea di comando o da programma.

1) Da linea di comando:
   mkfifo filename
2) Da programma, la funzione crea la FIFO *filename* con i permessi *mode*:
   *int mkfifo (const char *filename, mode_t mode)*

In entrambi i casi è comunque utilizzata la funzione *mkfifo()* che implica O_CREAT | O_EXCL.

Inoltre, il file *filename* non deve esistere sul file system, nemmeno come link simbolico altrimenti verrà restituito un errore EEXIST. Se viene restituito questo errore vuol dire che la FIFO è già presente, dunque occorre solo aprirla con *open()*.

Quindi, per creare una nuova FIFO o per aprirne una già esistente:

1) si invoca *mkfifo()*
2) si controlla un eventuale errore EEXIST, e se questo si verifica, si invoca *open()*

Esempio:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
int main(){
  int res = mkfifo("/tmp/my_fifo", 0777);
  if (res == 0)
  printf("FIFO creata\n");
   exit(0);
}
```

Il programma usa la funzione mkfifo per creare un file speciale. Sebbene impostiamo i permessi 0777 questi sono filtrati dalla maschera dell'utente (***umask***), così come abbiene nella normale creazione di file. Ad esempio, se la umask è 0022, i permessi risultanti saranno 755.

6.2.2 Pipe con nome: accedere ad una FIFO

Vediamo come si comporta una FIFO quando vi accediamo usando i programmi da linea di comando.
Una caratteristica utile delle pipe è che stando nel file system le possiamo usare in comandi dove normalmente utilizziamo nomi di file.

Esempio:
Proviamo a leggere dalla FIFO (vuota)

$ cat < /tmp/my_fifo

Ora scriviamo nella FIFO (usando un altro terminale, infatti il primo comando si sospende in attesa di dati scritti nella FIFO)

$ echo "safjandaskjfn" > /tmp/my_fifo

Si vedrà l'output apparire dal comando *cat*. Se non si invia alcun dato alla FIFO, il comando *cat* si sospenderà fino a che non lo si interrompe (CTRL+C).

In alternativa, è possibile realizzare le due operazioni ponendo il primo comando in background.

6.2.3 Pipe con nome: aprire una FIFO con open()

Vediamo il comportamento da programma quando vi accediamo in lettura e scrittura.

Ci sono varie differenze tra l'apertura di un file speciale (FIFO) e l'apertura di un file regolare.

PRIMA PARTICOLARITA'
Quando si apre una FIFO, è possibile farlo solo settando le opzioni O_RDONLY o O_WRONLY.
Infatti un programma non può aprire una FIFO per leggerla e scriverla nella modalità O_RDWR. In ogni caso, questa restrizione è relativa, in quanto usiamo le FIFO per passare i dati in una sola direzione e dunque non c'è necessità di aprirla in modalità O_RDWR.

SECONDA PARTICOLARITA'
Un'altra particolarità è che:

- un'operazione di scrittura su una FIFO aggiunge sempre i dati in coda
- un'operazione di lettura su una FIFO restituisce sempre ciò che si trova all'inizio della FIFO stessa

TERZA PARTICOLARITA'
Il secondo parametro di *open()* è l'argomento ***oflag***. Quando si apre una FIFO, questo argomento deve avere l'opzione O_NONBLOCK.

In definitiva esistono 4 combinazioni consentite di O_RDONLY, O_WRONLY e O_NONBLOCK.

```
open(const char *path, O_RDONLY);
```
in questo caso, la chiamata ad open si bloccherà; non ritorna fino a che un processo apre la stessa FIFO per scrittura

```
open(const char *path, O_RDONLY | O_NONBLOCK);
```
la chiamata ad open ha successo e ritorna immediatamente, anche se la FIFO non è stata aperta in scrittura da alcun processo

```
open(const char *path, O_WRONLY);
```
la chiamata ad open si bloccherà fino a che un processo apre la stessa FIFO in lettura

```
open(const char *path, O_WRONLY | O_NONBLOCK);
```
ritorna sempre immediatamente, ma se nessun processo ha la FIFO aperta in lettura, open ritornerà un errore, -1, e la FIFO non sarà aperta. Se un processo ha aperto la FIFO in lettura, il descrittore di file restituito può essere usato per scrivere al suo interno

6.2.4 *Pipe con nome: O_RDONLY e O_WRONLY senza O_NONBLOCK (non bloccanti)*

Un programma che ha settati i parametri in questo modo:
- consente al processo lettore di iniziare e di aspettare la chiamata ad open dello scrittore
- quando lo scrittore apre la FIFO, consente ad entrambi di continuare

6.2.5 *Pipe con nome: O_RDONLY e O_WRONLY con O_NONBLOCK (bloccanti)*

Un programma che ha settati i parametri in questo modo:
- fa sì che il processo lettore esegue la chiamata ad open e continua immediatamente, anche se non è presente alcun processo scrittore
- lo scrittore anch'esso continua immediatamente dopo la chiamata ad open() poiché la FIFO è già aperta in lettura

6.2.6 *Pipe con nome: leggere e scrivere in una FIFO bloccante*

Se la FIFO è aperta in modalità ***bloccante***:
- una *read()* su una FIFO vuota
  - aspetterà fino a che è disponibile qualche dato da leggere, se la FIFO è aperta in scrittura
  - restituisce 0 se la FIFO non è aperta in scrittura
- una *write()* su una FIFO
  - aspetterà fino a che i dati possono essere scritti, se la FIFO è aperta in lettura
  - genera una segnale SIGPIPE se la FIFO non è aperta in lettura

6.2.7 *Pipe con nome: leggere dalle FIFO non bloccanti*

L'utilizzo della modalità O_NONBLOCK influisce sul comportamento delle chiamate *read()* sulle FIFO.
Una *read()* su una FIFO vuota non bloccante:
- restituisce un errore se la FIFO è aperta in scrittura
- restituisce 0 se la FIFO non è aperta in lettura

6.2.8 *Pipe con nome: scrivere sulle FIFO non bloccanti*

L'utilizzo della modalità O_NONBLOCK influisce sul comportamento delle chiamate *write()* sulle FIFO.
Una *write()* su una FIFO non bloccante:

- genera un segnale SIGPIPE se la FIFO non è aperta in lettura
- se la FIFO è aperta in lettura
    - se il numero di byte da scrivere è <= di PIPE_BUF
        - se c'è spazio per il numero di byte specificato, sono trasferiti tutti i byte
        - se non c'è spazio, la write ritorna immediatamente con un errore
    - se il numero di byte è > di PIPE_BUF
        - se c'è spazio per almeno 1 byte nella FIFO, il kernel trasferisce tanti byte quanto spazio c'è nella FIFO e la write restituisce il numero di byte scritt
        - se la FIFO è piena, ritorna immediatamente con un errore

6.2.9 Pipe con nome: comunicazione Client-Server con FIFO

Un utilizzo delle FIFO consiste nell'inviare i dati tra un client ed un server.
Se abbiamo un server che è contattato da numerosi client, ogni client può scrivere la sua richiesta ad una FIFO che il server crea.
Poiché possono esserci multipli scrittori per la FIFO, le richieste inviate dai client al server devono essere minori di PIPE_BUF byte di dimensione

CAPITOLO 7. Thread

Oltre all'uso dei processi, è possibile usare più ***thread*** per eseguire compiti multipli nell'ambiente di un singolo processo. In questa metodologia di programmazione, tutti i thread all'interno di un singolo processo hanno accesso alle stesse componenti del processo quali, ad esempio, i descrittori di file e la memoria.

7.1 Perché usare i thread?

Un tipico processo Unix è composta da un singolo thread ed esegue un singolo compito alla volta. Grazie all'uso di thread multipli è possibile sviluppare programmi in grado di eseguire più task alla volta nell'ambito di un singolo processo. In pratica, ogni thread gestisce un compito separato.
Questo approccio comporta diversi vantaggi:

- si semplifica il codice relativo alla gestione di eventi asincroni assegnando un thread differente ad ogni evento di un tipo specifico
- se si usano processi multipli è necessario usare meccanismi complessi del SO per condividere memoria e descrittori, mentre i thread hanno accesso in maniera automatica allo stesso spazio di indirizzi di memoria e di descrittori di file
- è possibile suddividere alcuni problemi in modo da migliorare il throughput complessivo del programma
- la programmazione multithread può essere adottata sia su sistemi uniprocessore che su sistemi multiprocessore

7.2 Concetto di thread

Un thread consiste di varie informazioni, utili per rappresentare un contesto:

- thread ID
- valori dei registri
- stack
- politica di scheduling e relativa priorità
- maschera per i segnali
- variabile *errno*
- dati specifici del thread

Inoltre, ogni cosa all'interno di un processo è condivisibile tra i thread di un processo:

- codice del programma eseguibile
- memoria globale del programma
- descrittori di file

7.2.1 Thread POSIX

POSIX definisce l'API ***pthread*** utilizzabile nei programmi scritti in C.
Questa libreria consiste di 60 routine che effettuano operazioni utili per la *gestione dei thread*, per la *condivisione dei dati* e per la *sincronizzazione*.

In Linux è possibile implementare la *pthread* in due modi:

- *LinuxThreads*:
    - oltre al thread iniziale e ai thread creati da programma (mediante *pthread_create*), viene creato un thread manager che gestisce la creazione e la terminazione dei thread
    - i thread di un processo non condividono lo stesso PID
    - i thread sono visibili come processi separati dal comando *ps*

- *NPTL*:
  - non viene creato un thread manager
  - tutti i thread di un processo condividono lo stesso PID

7.3 Libreria Pthread

7.3.1 Identificazione di thread

Come già detto, ogni thread ha un identificatore, il ***thread ID***. Questo ID ha senso solo nel contesto del processo a cui appartiene.
I thread ID, nelle implementazioni portabili, non possono essere trattati come interi, ma occorre rappresentarli dal tipo di dato ***pthread_t***.

OTTENERE IL PROPRIO THREAD ID
Un thread può ottenere il proprio ID invocando la funzione ***pthread_self***, il cui prototipo è il seguente:

pthread_t pthread_self (void)

Questa funzione restituisce l'IDe del thread che la invoca, ecco perché il suo tipo è pthread_t.

CONFRONTARE DUE THREAD ID
Per confrontare gli ID di due thread è necessaria la funzione ***pthread_equal***, il cui prototipo è il seguente:

int pthread_equal (pthread_t tid1, pthread_t tid2);

dove tid1 e tid2 sono i thread ID che si vogliono confrontare.
Questa funzione restituisce un valore non nullo se sono uguali, restituisce 0 se sono diversi.

7.3.2 Creazione di thread

Quando un programma inizia l'esecuzione, esso parte come un singolo processo con un singolo thread.
Con il procedere dell'esecuzione, il suo comportamento è indistinguibile dal processo tradizionale fino a che esso crea più thread.

I thread possono essere creati usando la funzione ***pthread_create***, il cui prototipo è:

*int pthread_create (pthread_t *tidp, const pthread_attr_t *attr, void *(*start_rtn) (void *), void *arg)*

Restituisce 0 se la creazione è andata a buon fine, restituisce il numero di errore in caso di problemi.

Se la funzione ritorna con successo:
- la locazione di memoria puntata da *tidp* è impostata all'ID del nuovo thread creato
- l'argomento *attr* è utilizzato per definire vari attributi del thread (attributi di default → NULL)
- il nuovo thread inizia l'esecuzione all'indirizzo della funzione *start_rtn* che:
  - prende un solo argomento, *arg*
  - nel caso in cui bisogna passare alla funzione più di un argomento, occorre memorizzare i vari argomenti in una struttura e passare alla funzione l'indirizzo della struttura in *arg*

Quando viene creato un thread, così come per i processi, non c'è alcuna garanzia riguardo il thread che viene eseguito per primo, cioè non si sa se verrà eseguito il thread appena creato oppure il thread invocante.

Il nuovo thread ha accesso allo spazio di indirizzi del processo, dunque eredita l'ambiente e la maschera dei segnali del processo.

N.B. Un differenza tra i thread ed i processi è che questi ultimi, in caso di errore, modificano la variabile *errno*, mentre i thread restituiscono un codice di errore quando falliscono.

Esempio:

```
#include "apue.h"
#include <pthread.h>

pthread_t ntid;
void printids(const char *s)
{
  pid_t     pid;
  pthread_t tid;
  pid = getpid();
  tid = pthread_self();
  printf("%s pid %u tid %u (0x%x)\n", s,
   (unsigned int) pid, (unsigned int) tid,
   (unsigned int) tid);
}
void * thr_fn(void *arg)
{
  printids("nuovo thread: ");
  return ((void *)0);
}
```

```
int main(void)
{
  int       err;
  err = pthread_create(&ntid, NULL, thr_fn, NULL);
  if (err!=0)
      err_quit("non posso creare il thread: %s\n",
                                        strerror(err));
  printids("thread principale:");
  sleep(1);
  exit(0);
}
```

Questo programma crea un thread e stampa gli ID del processo, del nuovo thread e del thread iniziale.

Nel main c'è la necessità di usare una sleep per evitare che il thread iniziale possa uscire prima (terminando l'intero processo) che il nuovo thread abbia la possibilità di iniziare l'esecuzione.

Il nuovo thread ottiene il suo ID chiamando *pthread_self* invece di leggerlo dalla memoria condivisa o riceverlo come argomento alla sua routine di inizio thread. Infatti, ricordiamo che *pthread_create* restituirà l'ID del thread del nuovo thread attraverso il primo parametro.
Nell'esempio, il thread principale memorizza questo nella variabile *ntid*.

7.3.3 Terminazione di thread

Se un qualsiasi thread in un processo chiama *exit*, *_Exit* o *_exit*, allora l'intero processo termina.
Analogamente, se viene inviato al thread un segnale la cui azione di default è la terminazione del processo, allora verrà terminato l'intero processo.

Un thread può uscire senza causare la terminazione dell'intero processo. Sono tre i possibili modi:

- il thread chiama *pthread_exit*
- il thread torna dalla routine di avvio; il valore di ritorno è il codice di uscita del thread
- il thread può essere cancellato da un altro thread nello stesso processo

TORNARE DALLA ROUTINE DI AVVIO
Il prototipo è il seguente:

*void pthread_exit (void *rval_ptr);*

dove *rval_ptr* è un puntatore senza tipo, simile all'argomento passato alla routine di avvio. Questo puntatore è disponibile agli altri thread del processo chiamando la funzione *pthread_join*.

Con questo metodo, il thread chiamante si bloccherà fino a che il thread specificato chiama *pthread_exit*, ritorna alla sua routine di avvio, o è cancellato:

- se il thread ritorna dalla sua routine di avvio, *rval_ptr* conterrà il codice di ritorno
- se il thread è cancellato, la locazione di memoria specificata da *rval_ptr* è impostata a *PTHREAD_CANCELED*
- chiamando *pthread_join*, poniamo automaticamente il thread con cui facciamo il join nello stato *detached* in modo che le sue risorse possono essere recuperate

Se non si è interessati al valore di ritorno del thread, è possibile impostare il valore di *rval_ptr* a NULL. in questo caso, la chiamata a *pthread_join* ci consente di aspettare il thread specifico, ma non recupera lo stato di terminazione del thread.

CANCELLAZIONE DA PARTE DI UN ALTRO THREAD DELLO STESSO PROCESSO

Un thread può richiedere che un altro thread nello stesso processo sia cancellato mediante la funzione *pthread_cancel*, il cui prototipo è:

int pthread_cancel (pthread_t tid);

Questa funzione restituisce 0 se le operazioni si concludono senza errori; restituisce il numero di errore in caso di problemi.

Per default, *pthread_cancel* fa sì che il thread specificato da *tid* si comporti come se avesse chiamato *pthread_exit*, con l'argomento di *PTHREAD_CANCELED*.

N.B. LA funzione *pthread_cancel* non attende che il thread termini, ma effettua solamente la richiesta.

7.3.4 Thread VS Processi

Osserviamo le similitudini tra le funzioni relative ai thread e quelle relative ai processi.

Primitiva di processo	Primitiva di thread	Descrizione
fork	pthread_create	Crea un nuovo flusso di controllo
exit	pthread_exit	Esce da un flusso di controllo esistente
waitpid	pthread_join	Acquisisce lo stato di uscita dal flusso di controllo
getpid	pthread_self	Determina l'id del flusso di controllo
abort	pthread_cancel	Richiede la terminazione anomala del flusso di controllo

7.3.5 Attributi dei thread

Fino ad ora abbiamo assunto che i thread abbiano avuto attributi di default, cioè è stato passato alla funzione *pthread_create* l'argomento NULL invece che un puntatore ad una struttura *pthread_attr_t*.

Questa struttura si può usare per modificare gli attributi di default.
Si usa la funzione *pthread_attr_init* per inizializzare la struttura *pthread_attr_t*. Nel momento in cui questa struttura viene inizializzata, contiene i valori di default per tutti gli attributi.
Per liberare la memoria della struttura si usa la funzione *pthread_destroy*.
Di seguito riportiamo i due prototipi:

*int pthread_attr_init (pthread_attr_t *attr);*
*int pthread_attr_destroy (pthread_attr_t *attr);*

POSIX definisce funzioni separate per interrogare ed impostare ciascun attributo.

Riportiamo a sinistra i valori di default degli attributi, a destra i valori assegnabili:

Attributo	Valore	Significato del valore di default
int scope	PTHREAD_SCOPE_PROCESS	competizione sulle risorse all'interno di un processo
int detachstate	PTHREAD_CREATE_JOINABLE	joinable con altri threads
void *stackaddr	NULL	allocato dal sistema
size_t *stacksize	NULL	1 megabyte
priority	NULL	priorità del thread padre
int schedpolicy	SCHED_OTHER	determinata dal sistema
ineritsched	PTHREAD_EXPLICIT_SCHED	attributi di scheduling stabiliti esplicitamente, es. policy

Attributo	Valore
scope	PTHREAD_SCOPE_PROCESS PTHREAD_SCOPE_SYSTEM
detachstate	PTHREAD_CREATE_JOINABLE PTHREAD_CREATE_DETACHED
stackaddr	NULL,
stacksize	NULL, PTHREAD_STACK_MIN
priority	NULL
schedpolicy	SCHED_OTHER, SCHED_FIFO, SCHED_RR
ineritsched	PTHREAD_EXPLICIT_SCHED, PTHREAD_EXPLICIT_SCHED

Tra i vari attributi quello di maggior rilievo è il concetto di thread distaccato: se non siamo interessati allo stato di terminazione di un thread esistente, possiamo usare *pthread_detach* per consentire al sistema operativo di reclamare le risorse del thread quando il thread esce. Se sappiamo di non aver bisogno dello stato di terminazione del thread nel momento in cui lo creiamo, è possibile avviare il thread nello stato distaccato modificando l'attributo *detachstate*.

7.3.6 Thread e segnali

Ogni thread ha una propria maschera dei segnali.

I thread possono, individualmente, bloccare i segnali, ma quando un thread modifica l'azione associata ad un dato segnale, tutti i thread condividono l'azione. Cioè se un thread decide di ignorare un certo segnale, tutti gli altri lo ignoreranno, ma ognuno di essi ha la possibilità di annullare quella scelta, cambiando la disposizione da adottare: eseguire l'azione di default o installare un signal handler.

I segnali sono consegnati ad un singolo thread nel processo. In particolare:

- se il segnale è relativo a un errore hardware, allora il segnale è inviato al thread che ha causato l'evento
- gli altri segnali sono consegnati ad un thread arbitrario

Per inviare un segnale ad un processo, invochiamo *kill*.
Per inviare un segnale ad un thread, invochiamo *pthread_kill*.

CAPITOLO 8. Mutex e variabili di condizione

Quando più thread condividono la stessa memoria è necessario che ognuno di essi mantenga la coerenza dei dati. Non c'è alcun problema di consistenza se ciascun thread usa variabili che altri thread non leggono o modificano. Quando, invece, un thread può modificare una variabile che altri thread possono leggere o modificare, occorre sincronizzare i thread per assicurare che questi non utilizzino un valore non valido quando accedono al contenuto di memoria della variabile.

In pratica, quando si utilizzano variabili condivise, occorre sincronizzare i thread per l'accesso alle variabili.
Per poter raggiungere questo obiettivo i thread devono usare un *"lock"* che consente ad un solo thread alla volta di accedere alla variabile.

Un altro caso in cui è necessaria la sincronizzazione dei thread è quando due thread (o più di due) cercano di modificare la stessa variabile nello stesso tempo. Se, ad esempio, due thread cercano di incrementare la stessa variabile nello stesso (quasi) momento senza sincronizzarsi, il risultato può essere inconsistente.
Se la modifica della variabile fosse stata atomica, non ci sarebbe stata alcuna *race condition*.

POSIX.1c mette a disposizione due primitive per la sincronizzazione dei thread:
- ***mutex***
- ***variabili di condizione***

POSIX.1b permette di sincronizzare i thread mediante l'uso di ***semafori***.

8.1 Mutex

Un mutex (MUTual EXclusion – mutua esclusione) è un oggetto che permette a processi o thread concorrenti di sincronizzare l'accesso a dati condivisi.

Un mutex possiede due stati: ***bloccato*** *(1)* e ***non bloccato*** *(0).*
Quando un mutex è bloccato, cioè ha valore 1, gli altri thread che tentano di bloccarlo restano in attesa.
Quando il thread che blocca rilascia il mutex, cioè lo imposta a 0, uno dei thread in attesa lo acquisisce.

Il funzionamento della sincronizzazione è il seguente: ogni volta che un processo o thread ha bisogno di accedere ai dati condivisi, acquisisce il mutex; quando termina le sue operazioni, il mutex viene rilasciato, permettendo ad un altro processo o thread di acquisirlo per eseguire le sue operazioni.
Un mutex è, quindi, utilizzato per proteggere una sezione critica assicurando che solo un thread per volta esegua il codice nella regione critica.

Il codice normalmente è del tipo:

```
lock_the_mutex (…);
regione critica
unlock_the_mutex (…);
```

Un mutex è una variabile rappresentata dal tipo di dato *pthread_mutex_t*.
Prima di usare un mutex è necessario inizializzarlo. Esistono due tipi di inizializzazione:
- ***statica*** si imposta il valore della costante *PTHREAD_MUTEX_INITIALIZER*
- ***dinamica*** si invoca *pthread_mutex_init()*

Nel secondo caso, è necessario invocare *pthread_mutex_destroy()* per liberare la memoria
Riportiamo di seguito i prototipi delle funzioni necessarie per l'approccio dinamico:

```
int pthread_mutex_init ( pthread_mutex_t *mutex, const pthread_mutexattr_t *attr )
int pthread_mutex_destroy ( pthread_mutex_t *mutex )
```

8.1.1 Acquisizione e rilascio dei mutex

Un thread per bloccare un mutex usa *pthread_mutex_lock()*
- la funzione ritorna quando il mutex è stato bloccato dal thread chiamante
- il mutex resta bloccato fino a quando non è sbloccato dal thread chiamante

Per sbloccare un mutex si usa *pthread_mutex_unlock()*
- se vi sono più thread in attesa di acquisire il mutex, la politica di scheduling dei thread stabilisce chi lo acquisisce

Di seguito riportiamo i prototipi di queste due funzioni:

```
int pthread_mutex_lock ( pthread_mutex_t *mutex );
int pthread_mutex_unlock ( pthread_mutex_t *mutex );
```

Restituiscono 0 se tutto va a buon fine, altrimenti restituiscono il numero di errore.

8.1.2 Attributi dei mutex

Per default (attributo settato a *PTHREAD_PROCESS_PRIVATE*), un mutex può essere usato solo da thread che appartengono allo stesso processo.
Però utilizzando l'attributo *PTHREAD_PROCESS_SHARED* si permette a thread di altri processi di utilizzare il mutex.

Per allocare dinamicamente gli attributi di un mutex si usa:

```
int pthread_mutexattr_init ( pthread_mutexattr_t *attr );
```

Per deallocare dinamicamente gli attributi di un mutex si usa:

```
int pthread_mutexattr_destroy ( pthread_mutexattr_t *attr );
```

8.2 Variabili di condizione

Le variabili di condizione costituiscono un ulteriore meccanismo di sincronizzazione per i thread.
Mentre i mutex implementano la sincronizzazione controllando l'accesso dei thread ai dati usando il polling (ciclo in cui si spreca tempo di CPU), le variabili di condizione permettono di sincronizzare i thread sulla base dell'attuale valore dei dati, evitando il polling.
Quando un altro thread causerà l'occorrenza di tale evento, uno o più thread in attesa riceveranno un segnale e si risveglieranno.

Una variabile di condizione è sempre associata ad un mutex lock.

8.2.1 Caratteristiche e inizializzazione

Una variabile di condizione è del tipo *pthread_cond_t* e prima di usarla occorre inizializzarla.
Sono due i modi per inizializzare una variabile di condizione:
- ***inizializzazione statica***: mediante la costante *PTHREAD_COND_INIZIALIZER*
- ***inizializzazione dinamica***: mediante *pthread_cond_init()*

In quest'ultimo caso, occorre richiamare *pthread_cond_destroy()* per deallocare una variabile di condizione prima di liberare la memoria.

8.2.2 *Attendere e segnalare una condizione*

Un thread può attendere su una variabile condizione:

- per un tempo specificato – *thread_cond_timedwait()*
- per un tempo indefinito – *thread_cond_wait()*

Quando la condizione si verifica, si può risvegliare:

- almeno un thread in attesa – *pthread_cond_signal (condition)*
- tutti i thread in attesa – *pthread_cond_broadcast (condition)*

Vediamo qui di seguito i prototipi di queste funzioni:

*int pthread_cond_wait (pthread_cond_t *cptr, pthread_mutex_t *mptr);*
*int pthread_cond_signal (pthread_cond_t *cptr);*
*int pthread_cont_broadcast (pthread_cond_t *cptr);*
*int pthread_cond_timedwait (pthread_cond_t *cptr, pthread_mutex_t *mptr, const struct timespec *abstime);*

Il mutex passato a *pthread_cond_wait()* protegge la condizione

- il chiamante lo passa bloccato alla funzione
- la funzione pone il thread chiamante nella lista dei thread in attesa della condizione e sblocca il mutex (tutto in modo atomico)
- quando *pthread_cond_wait()* ritorna, il mutex viene dinuovo bloccato

<table>
<tr><td colspan="2">Thread principale
•Dichiara ed inizializza dati/variabili globali che richiedono sincronizzazione
•Dichiara ed inizializza una variabile di condizione
•Dichiara ed inizializza un mutex associato
•Crea thread A e B</td></tr>
<tr><td>Thread A
•Esegue fino al punto in cui una certa condizione deve verificarsi
•Lock il mutex associato e controlla il valore di una variabile globale
•Chiama pthread_cond_wait() per effettuare una wait bloccante in attesa del risveglio da parte del thread B (automaticamente e atomicamente corrisponde ad un unlock del mutex associato in modo tale che possa essere usato dal thread B).
•Quando risvegliato, lock il mutex in modo automatico e atomico
•Unlock il mutex in modo esplicito
•Continua</td><td>Thread B
Lavora
Lock il mutex associato
Modifica il valore della variabile globale su cui il thread A è in attesa
Controlla il valore della variabile globale di attesa del thread A. Se si verifica la condizione desiderata, risveglia il thread A invocando pthread_cond_signal()
Unlock il mutex
Continua</td></tr>
<tr><td colspan="2">Thread principale
Join / Continua</td></tr>
</table>

CAPITOLO 9. Semafori POSIX

Un semaforo è una primitiva usata per fornire un meccanismo di sincronizzazione tra vari processi.
Possiamo considerare tre tipologie di semafori:

- i ***semafori POSIX***, che non sono mantenuti nel kernel e che possono essere di due tipi:
  - *basati su nome* ed identificati da nomi POSIX per IPC
  - *basati su memoria* e memorizzati in memoria condivisa
- i ***semafori nella versione System V***, che sono mantenuti nel kernel

9.1 Introduzione

9.1.1 Operazioni sui semafori

Un processo può eseguire 3 operazioni sui semafori:

- ***creazione***: richiede di inizializzare anche il valore di partenza (0 o 1 per i semafori binari)
- ***wait*** (o anche ***P***): testa il valore del semaforo
  - si blocca se il valore è ≤ 0
  - decrementa il valore del semaforo se è > 0

  N.B. Le operazioni di test e di decremento sono fatte atomicamente
- ***signal*** (o anche ***V*** o ***post***): incrementa il valore del semaforo
  - se un processo è in attesa che il valore del semaforo sia > 0, allora tale processo può essere risvegliato

9.1.2 Tipi di semaforo

Esistono vari tipi di semaforo.

SEMAFORO CONTATORE
Il concetto di semaforo contatore generalizza quello di semaforo binario.

Esso è un semaforo il cui valore varia tra 0 e qualche valore limite. Solitamente viene utilizzato per contare le istanze disponibili di una qualche risorsa, dunque il suo valore indica il numero di risorse disponibili.

L'operazione *wait* aspetta che il valore del semaforo sia maggiore di 0 e poi decrementa tale valore.
L'operazione *signal* incrementa il valore del semaforo e risveglia un processo in attesa che il valore del semaforo sia > 0.

SEMAFORO BINARIO
Un semaforo binario è un semaforo che può assumere solo i valori 0 e 1 dove:

- 0 sta per bloccato
- 1 sta per sbloccato

Esso può essere usato per la mutua esclusione come un mutex:

```
inizializza il mutex;              inizializza il semaforo a 1;

pthread_mutex_lock(&mutex);        sem_wait(&sem);
    regione critica;                   regione critica;
pthread_mutex_unlock(&mutex);      sem_post(&sem);
```

Viene inizializzato il semaforo ad 1.

La chiamata a *sem_wait()* attende che il valore sia > 0 e poi decrementa il valore.
La chiamata a *sem_post()* incrementa il valore (da 0 a 1) e risveglia un thread o processo bloccato da una *sem_wait()* sullo stesso semaforo.

C'è una differenza tra l'uso dei mutex e l'uso dei semafori binari usati come mutex.
I semafori hanno una caratteristica che i mutex non hanno: un mutex deve sempre essere sbloccato dal thread che lo ha bloccato, mentre un'operazione *signal* su un semaforo non deve essere necessariamente effettuata dallo stesso thread che ha effettuato la *wait* sul semaforo.

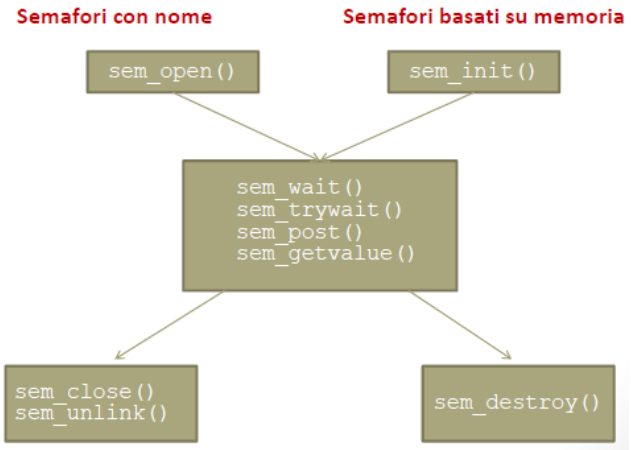
9.1.3 Differenze tra meccanismi di sincronizzazione

Possiamo elencare tre differenze tra semafori, mutex e variabili di condizione:

1) un mutex deve sempre essere sbloccato dal thread che lo ha bloccato, mentre un'operazione *signal* su un semaforo non deve necessariamente essere fatta dallo stesso thread che ha invocato l'operazione *wait* sul semaforo
2) un mutex può essere nello stato bloccato o sbloccato (uno stato binario simile al semaforo binario)
3) poiché un semaforo ha uno stato associato con esso (il contatore), un'operazione *signal* viene sempre ricordata; quando una variabile condizione è segnalata, se nessun thread è in attesa per essa, il segnale viene perso

9.2 Semafori basati su nome

POSIX fornisce due tipi di semafori: con nome e basati su memoria.



I semafori basti su nome sono dei semafori identificati da un nome che referenzia un file nel file system.

9.2.1 Creazione di un semaforo

La funzione usata per creare un semaforo è *sem_open()* che crea un nuovo semaforo con nome oppure apre un semaforo già esistente.
Un semaforo con nome può sempre essere usato per sincronizzare sia i thread che i processi.
Il prototipo è il seguente:

*sem_t *sem_open (const char *name, into flag, mode_t mode, int value);*

Il primo argomento è un nume che può essere o meno un pathname reale nel file system.
Il secondo argomento è *oflag*, che può essere:

- 0
- O_CREAT, che può essere specificato sia se il semaforo già esiste sia se non esiste
- O_CREAT | O_EXCL, che può essere specificato solo se il semaforo non esiste

Se viene specificato O_CREAT allora sono richiesti anche il terzo e il quarto argomento:

- il terzo argomento consiste nei bit di permesso
- il quarto argomento consiste nel valore iniziale del semaforo che non può superare *SEM_VALUE_MAX*; i semafori binari hanno solitamente valore iniziale 1

La funzione *sem_open()* ritorna un puntatore al tipo *sem_t* che viene utilizzato come argomento nelle altre funzioni *sem_close()*, *sem_wait()*, *sem_trywait()*, *sem_post()* e *sem_getvalue()*.

9.2.2 Chiusura di un semaforo

La funzione utilizzata è la *sem_close()* a cui viene passato il semaforo da chiudere. Restituisce 0 se la chiusura va a buon fine, altrimenti restituisce -1.
Il prototipo è il seguente:

```
inte sem_close ( sem_t *sem );
```

La chiusura di un semaforo non lo rimuove dal sistema. La sua persistenza è almeno a livello kernel: ciò vuol dire che esiste fino al reboot o fino a che l'oggetto è cancellato esplicitamente (*semafori System V*).
In ogni caso, per rimuovere un semaforo si usa la funzione *sem_unlink()*.

9.2.3 Funzioni sem_wait() e sem_trywait()

La funzione *sem_wait()* testa il valore del semaforo specificato.
Se il valore è > 0, il valore è decrementato e la funzione ritorna immediatamente.
Se il valore è 0, il thread chiamante è messo in attesa (sleep) fino a che il valore del semaforo diventa > 0. Non appena il valore diventa maggiore di 0, il semaforo sarà decrementato e la funzione ritorna.

La differenza tra *sem_wait()* e *sem_trywait()* è che quest'ultima non pone il thread in attesa nel caso in cui il valore del semaforo sia 0, ma viene restituito un errore.

9.2.4 Funzioni sem_post() e sem_getvalue()

La funzione *sem_post()* incrementa il valore del semaforo di 1 e risveglia qualsiasi thread in attesa che il suo valore diventi positivo.

La funzione *sem_getvalue()* restituisce il valore corrente del semaforo. Se il semaforo è bloccato, restituisce il valore 0.

9.3 Semafori basati su memoria

Nei semafori basati su memoria, l'applicazione alloca la memoria per il semaforo che successivamente il sistema provvede ad inizializzare.

9.3.1 Funzioni sem_init() e sem_destroy()

Un semaforo basato su memoria è inizializzato con *sem_init()*, poi una volta finite le operazioni, il semaforo è deallocato con *sem_destroy()*.

La differenza fondamentale tra *sem_open()* e *sem_init()* è che la prima restituisce un puntatore ad una variabile sem_t che la funzione ha allocato ed inizializzato. La seconda, invece, ha come primo argomento un puntatore ad una variabile sem_t che il chiamante deve allocare e la funzione deve inizializzare.

CAPITOLO 10. Semafori System V

I semafori System V aggiungono un ulteriore livello di dettaglio ai semafori contatori POSIX, dando la possibilità di definire un insieme di semafori contatori, ognuno dei quali è un semaforo contatore; in Linux è possibile creare insiemi di massimo 250 semafori.
Quando si parla di "semafori System V" ci riferiamo ad un insieme di semafori contatori.
Quando si parla di "semafori POSIX" ci riferiamo ad un singolo semaforo contatore.

10.1 Introduzione

Il kernel associa a ciascun insieme di semafori la seguente struttura:

```
struct semid_ds {
struct ipc_perm sem_perm;  /* info su proprietari, permessi, …  */
struct sem *sem_base;          /* ptr a primo semaforo */
ushort sem_nsems;              /* numero semafori */
time_t sem_otime;              /* tempo ultima semop() */
time_t sem_ctime;              /* tempo ultima modifica */
};
```

Il campo *sem_base* è il puntatore alle strutture *sem*:

```
struct sem {
ushort semval;          /* valore del semaforo (>= 0) */
pid_t sempid;           /* pid ultima operazione */
ushort semncnt;    /* #processi in attesa di semval > current_value */
ushort semzcnt;         /* #processi in attesa semval=0 */
};
```

La struttura *ipc_perm* contiene informazioni per ogni oggetto di IPC che il kernel memorizza:

```
struct ipc_perm {
  uid_t       uid;    /* user id proprietario */
  gid_t       gid;    /* group id proprietario */
  uid_t       cuid;   /* user id creatore */
  gid_t       cgid;   /* group id creatore */
  mode_t      mode;   /* permessi lettura-scrittura */
  ulong_t     seq;    /* numero sequenza */
  key_t       key;    /* chiave IPC */
};
```

10.1.1 Utilizzo dei semafori

I semafori possono essere utilizzati sia singolarmente sia come insiemi.
Un insieme di semafori:

- viene inizializzato con *semget()*
- le operazioni vengono eseguite con *semop()*
- le operazioni di controllo sono eseguite con *semctl()*

Quando si crea un semaforo, deve essere specificata una chiave *key* che deve essere nota a tutti i processi che vogliono utilizzare la struttura. Le chiavi possono essere ottenute con *ftok()*.

CAPITOLO 11. Socket

Nei moderni SO i servizi disponibili in rete si basano principalmente sul modello client/server. Questa architettura consente ai sistemi di condividere risorse e cooperare per il raggiungimento di un obiettivo.

Per implementare programmi client/server esiste l'interfaccia delle socket.

Questa interfaccia prevede un server e molti client.
Il server è sempre attivo, ha un indirizzo IP fisso e il suo compito è quello di rispondere alle richieste di servizi da parte dei client. Quindi un ***processo server*** è il processo che aspetta di essere contattato.
I client comunicano con il server in qualsiasi momento, possono avere indirizzi IP dinamici e non comunicano direttamente con gli altri client. Quindi un ***processo client*** è un processo che inizia la comunicazione col processo server.

11.1 Operazioni sulle socket

Due processi, il client e il server, che sono in esecuzione su host differenti comunicano scambiandosi messaggi. Lo scambio dei messaggi avviene attraverso l'utilizzo di ***socket***, che più o meno sono come delle porte. Quindi processo client e processo server comunicano tramite socket.
La comunicazione tra due host può sfruttare il protocollo TCP (affidabile, cioè garantisce la riuscita del trasferimento) o il protocollo UDP (non affidabile, cioè non garantisce la riuscita del trasferimento).

11.1.1 Programmazione delle socket con TCP

Il client deve contattare il server, che ovviamente deve essere in esecuzione.
I due processi stabiliscono la connessione con la triplice stretta di mano.

Il client contatta il server:
- creando una socket TCP
- specificando l'indirizzo IP e il numero di porta del server

Il server viene contattato dai client:
- per comunicare con ciascuno di essi crea una nuova socket
- usa i numeri di porta per distinguere i client

11.1.2 Programmazione delle socket con UDP

I due processi non stabiliscono una connessione affidabile in quanto non effettuano la triplice stretta di mano. Quindi i dati trasmessi possono perdersi e, inoltre, possono arrivare a destinazione in un ordine diverso da quello di invio.

Il mittente allega esplicitamente ad ogni pacchetto l'indirizzo IP e la porta di destinazione.
Il server deve estrarre l'indirizzo IP e la porta del mittente dal pacchetto ricevuto.

11.1.3 Caratteristiche socket

Una socket è un meccanismo di comunicazione che consente di sviluppare sistemi client/server sia localmente, su una macchina locale, o attraverso una rete. La differenza con le pipe è che le socket distinguono il client e il server.

Visti i due protocolli, abbiamo due tipi di socket:

- ***stream socket***, che forniscono stream di dati affidabili ed ordinati
- ***socket a datagrammi***, che trasferiscono messaggi di dimensione variabile (preservando i confini) ma senza garantire ordine o arrivo dei pacchetti

11.1.4 Connessioni socket (Protocollo TCP)

LATO SERVER

1. Per primo, il server crea una socket mediante la system call *socket()*.
2. Il processo server da un nome alla socket attraverso la system call *bind()*. Questo nome sarà un identificatore (numero di porta / punto di accesso) che consente al SO di instradare le connessioni in arrivo dai client.
3. Dopo aver effettuato la bind, il server si mette in attesa che un client si connetta alla socket a cui è stato dato il nome. Questa attesa è gestita mediante la system call *listen()* che crea una coda di connessioni in arrivo.
4. Le richieste che arrivano dai client vengono accettate dal server mediante la system call *accept()*. Quando il server invoca questa funzione, viene creata una nuova socket distinta da quella dotata di nome (triplice stretta di mano) che viene usata per comunicare con il client specifico, mentre la socket dotata di nome resta a disposizione per ulteriori connessioni con altri client

LATO CLIENT

1. Il client crea una socket con la system call *socket()*.
2. Successivamente invoca *connect()* per stabilire una connessione con il server usando la socket del server come indirizzo.

11.1.3 Attributi delle socket

Le socket sono caratterizzate da tre attributi: *dominio*, *tipo* e *protocollo*.

DOMINI DELLE SOCKET

I domini specificano il mezzo della rete che la comunicazione socket userà. Il più comune è AF_INET (altrimenti ci sono AF_UNIX per il client-server locale, AF_ISO o AF_XNS).

Possono esserci numerosi servizi su un server. I server aspettano le connessioni su porte specifiche.
Nel sistema una porta è identificata internamente da un intero ed esternamente dalla combinazione dell'indirizzo IP e del numero di porta.
Solitamente, per i servizi noti, sono allocati numeri di porta standard, ad esempio http ha la porta 80.

TIPI DI SOCKET

Come già detto, i protocolli internet forniscono due livelli distinti di servizio: *stream* e *datagram*.

Le ***socket stream*** forniscono una connessione che è un flusso di byte affidabile e sequenziato (cioè rispetta l'ordine di invio). E' garantito che i dati non siano persi, duplicati o riordinati. I messaggi più grandi sono frammentati, trasmessi e riassemblati.

Le socket stream sono specificate dal tipo *SOCK_STREAM* e sono implementate nel dominio *AF_INET*.

Le ***socket datagram*** non stabiliscono una connessione, dunque i dati possono andare persi. Inoltre non viene garantita la sequenzialità, cioè i dati possono arrivare in ordine diverso da quello di invio.

Le socket datagram sono specificate dal tipo *SOCK_DGRAM* e sono implementare nel dominio *AF_INET*.

11.1.5 Creare una socket

La system call *socket()* crea una socket e restituisce un descrittore che può essere usato per accedere alla socket stessa. Il prototipo è il seguente:

int socket (int dominio, int tipo, int protocollo);

La socket creata è un estremo del canale di comunicazione:

- il parametro *dominio* specifica la famiglia di indirizzi (possibili valori: *AF_INET* e *AF_UNIX*)
- il parametro *tipo* specifica il tipo di comunicazione da usare con questa socket (possibili valori: *SOCK_STREAM* e *SOCK_DGRAM*)
- *protocollo* specifica il protocollo da impiegare, solitamente è determinato dal tipo di socket e dal dominio (0 seleziona il protocollo di default)

Ciascun dominio richiede il proprio formato di indirizzo. Per indicare una socket nel dominio AF_INEt, l'indirizzo è specificato usando una struttura *sockaddr_in* contenente, tra le altre info, il numero di porta e l'indirizzo IP.

11.1.6 Assegnare un nome alle socket

Dopo aver creato una socket, il server deve assegnare un nome alla stessa, mediante *bind()*.

*int bind (int sockfd, struct sockaddr *address, size_t len);*

Questa system call assegna l'indirizzo specificato da *address* alla socket senza nome associata al descrittore di file *sockfd*. La lunghezza della struttura è passata come *len*.

11.1.7 Creare una coda per la socket

Per accettare le connessioni in arrivo su una socket, un server deve creare una coda per memorizzare le richieste pendenti. Tutto questo tramite *listen()*.

int list (int sockfd, int backlog);

Solitamente questa system call è eseguita dopo le chiamate di sistema *socket()* e *bind()* ma prima della chiamata *accept()*. Essa imposta la lunghezza della coda a *backlog*.

11.1.8 Accettare le connessioni

Dopo che un server ha creato e dato un nome ad una socket, pul aspettare le connessioni usando la system call *accept()*.

*int accept (int sockfd, struct sockaddr *address, size_t *len);*

Questa system call ritorna quando un client (presenta nella coda della socket) tenta di connettersi alla socket specificata da *sockfd*. Essa crea una nuova socket per comunicare con il client e restituisce il suo descrittore. La nuova socket avrà lo stesso tipo della socket in ascolto del server.

L'indirizzo del client sarà posto nella struttura *sockaddr* puntata da *address*.

Il parametro *len* specifica la lunghezza della struttura del client.

Se non ci sono connessioni pendenti sulla coda della socket, *accept()* blocca il processo fino a che un client effettua una connessione.

11.1.9 Richiedere connessioni

I client si connettono ai server stabilendo una connessione tra una socket senza nome e la socket del server in ascolto. Questo è fatto invocando *connect()*.

*int connect (int sockfd, struct sockaddr *address, size_t len);*

La socket specificata da *sockfd* è conessa alla socket del server specificata da *address*, che è di lunghezza *len*.

Se la connessione non può essere impostata immediatamente, *connect()* bloccherà il processo per un periodo di tempo (timeout) non specificato. Una volta trascorsa il timeout, la connessione sarà annullata e *connect()* fallisce.

11.1.10 Chiudere una socket

Si può terminare una connessione socket al server e al client invocando *close()*.
E' necessario sempre chiudere la socket in ambo i lati.

11.1.11 Scambio di dati

Una volta stabilita la connessione tra client e server, si possono usare le system call *send()* e *recv()* per trasmettere e ricevere dati attraverso la socket.

11.1.12 Trasferimento UDP

Se si usa il protocollo UDP, il ***server*** dopo aver creato una socket locale e dopo aver eseguito la *bind()* non deve mettersi in ascolto, in quanto il client farà la maggior parte del lavoro.
Il ***client*** crea una propria socket locale, chiama la *bind()* e per inviare e ricevere i messaggi al/dal server utilizza le chiamate *recvfrom()* e *sendto()*.

11.2 Ordinamento dei byte

Se si eseguono programmi server e client è possibile vedere le connessioni di rete usando il comando *netstat* che mostra le connessioni client/server in attesa di chiusura.
Si possono osservare i numeri di porta che sono stati assegnati alla connessione tra il server ed il client:

- il local address mostra il server
- il foreign address è il client remoto

Inoltre, per assicurare che tutte le socket siano distinte, queste porte client sono tipicamente differenti dalle socket del server in ascolto e uniche.

Visto che in rete ci sono computer con differenti architetture, possono crearsi delle incongruenze circa gli ordinamenti degli interi. Infatti alcuni processori sono di tipo ***little endian*** mentre altri processori sono di tipo ***big endian***. Se la memoria usata per gli interi è copiata byte per byte, due computer con ordinamenti differenti, assegnano valori diversi all'intero.

Per consentire a computer di tipo differente di avere rappresentazioni degli interi coerenti, è necessario che i client e i server convertano le rispettive rappresentazioni interne prima di effettuare la trasmissione. Ci sono funzioni che effettuano queste operazioni come la *htonl*, la *htons*, la *ntohl* e la *ntohs*. Esse si occupano di convertire gli interi tra il formato host nativo e il formato scelto come standard nella connessione di rete.

Ciò comporta che i codici client e server devono essere modificati opportunamente. Ad esempio:

- Server

```
server_address.sin_addr.s_addr = htonl(INADDR_ANY);
server_address.sin_port = htons(9734);
```

- Client

```
address.sin_port = htons(9734);
```

CAPITOLO 12. File I/O

I programmi, per utilizzare i servizi offerti da Unix, devono interagire col sistema operativo mediante delle ***system call***, che costituiscono l'interfaccia tra il programmatore e il kernel, in pratica sono delle "entry point" per il kernel.
Le funzioni di libreria invece non sono delle "entry point" per il kernel, ma consentono comunque di chiamare le system call come se fossero delle funzioni C. In definitiva possiamo semplificare il concetto dicendo che system call = funzione C.

12.1 Gestione dei file

Le system call per la gestione dei file permettono di manipolare file regolari, directory e file speciali (link simbolici, dispositivi e meccanismi di IPC come pipe e socket).

I file aperti sono gestiti dal kernel mediante dei ***descrittori di file*** (interi non negativi) che possono variare da 0 a *OPEN_MAX*. Quando si apre un file esistente o se ne crea uno nuovo, il kernel ritorna un descrittore di file al processo chiamante.

Per convenzione:
- il descrittore 0 viene associato allo standard input [*STDIN_FILENO*]
- il descrittore 1 viene associato allo standard output [*STDOUT_FILENO*]
- il descrittore 2 viene associato allo standard error [*STDERR_FILENO*]

Le system call per le operazioni di base per la gestione dei file sono:
- *open()*
- *read()*
- *write()*
- *lseek()*
- *close()*

12.1.1 Chiamata open()

Questa funzione è usata per aprire o creare file

*int open (const *path, int oflag, mode_t mode);*

dove:
- *path* è il nome del file da creare o aprire
- *oflag* può assumere diversi valori: *O_RDONLY*, *O_WRONLY* o *O_RDWR*
- il terzo argomento viene usato solo quando si crea un file

Ci sono altre opzioni del secondo argomento che possono essere usate in questa system call come *O_TRUNC, O_NONBLOCK, O_CREAT, O_EXCL, O_APPEND*.

12.1.2 Chiamata close()

Questa funzione permette di chiudere un file.

12.1.3 Chiamata lseek()

Ad ogni file aperto è associato un valore intero non negativo, detto ***offset corrente del file***, che misura il numero di byte dall'inizio del file.

off_t lseek (int filedes, off_t offset, int whence);

Questa funzione, in caso di successo, ritorna il nuovo offset.

L'argomento *whence* può assumere i seguenti valori:
- *SEEK_SET*, l'offset viene posto a offset byte dall'inizio del file
- *SEEK_CUR*, viene aggiunto offset all'offset corrente
- *SEEK_END*, l'offset viene posto alla fine del file, più offset

Poiché una chiamata a *lseek* restituisce il nuovo offset del file, per determinare l'offset corrente si cerano zero byte dalla posizione corrente:

posizione_corrente = lssek (fd, 0, SEEK_CUR);

L'offest di un file può essere più grande della dimensione corrente del file:
- la write successiva estende il file
- crea un buco
- qualsiasi byte nel file che non è stato scritto è letto come 0
- non è richiesto che ai buchi sia allocato un blocco su disco

12.1.4 Chiamata read()

Questa funzione legge dal file *filedes*, *nbytes* byte in *buf*, a partire dalla posizione corrente.

*ssize_t read (int filedes, void *buf, size_t nbytes);*

Ritorna il numero di byte effettivamente letti se ha successo, ritorna 0 se alla fine del file.
Inoltre, aggiorna la posizione corrente.

12.1.4 Chiamata write()

Questa funzione scrive nel *filedes*, *nbytes* byte da *buf*, a partire dalla posizione corrente.

*ssize_t write (int filedes, const void *buf, size_t nbytes);*

Restituisce il numero di byte effettivamente scritti se ha successo.
Inoltre, aggiorna la posizione corrente.

12.2 Gestione degli errori: errno e perror()

Una system call ritorna -1 se fallisce.
Per gestire gli errori originati dalle system call, si possono utilizzare:
- ***errno***, è una variabile globale che contiene il codice numerico dell'ultimo errore generato da una system call
- ***perror()***, è una subroutine che mostra una descrizione "testuale" dell'ultimo errore generato dall'invocazione di una system call

CAPITOLO 13. File e directory

La maggior parte dei file in Unix sono di due tipi: regolari e directory. Le informazioni sui file possono essere ottenuto usando le funzioni *stat()*, *fstat()* e *lstat()*.
La *stat()* ritorna informazioni sul file specificato.
La *fstat()* ritorna informazioni sul file aperto sul descrittore *filedes*.
La *lstat()* ritorna informazioni sul link simbolico e non sul file puntato da esso.

12.1 Set UID e Set GID

A ciascun processo vengono associati i seguenti identificativi:

- *real user ID* e *real group ID* identificano l'utente
- *effective user ID*, *effective group ID* e *supplementary group ID* determinano i permessi di accesso ai file

Solitamente l'*effective user ID* coincide con il *real user ID* e l'*effective group ID* coincide con il *real group ID*.

Ogni file ha un proprietario ed un gruppo che lo possiede. Quando si lancia un programma, questo viene eseguito con i permessi di chi lo manda in esecuzione, non di chi lo possiede.

Esempio:

Sia exe il nome di un file eseguibile. Siano utente1 e gruppo1 i valori di UID e GID dell'utente che lancia exe, producendo un processo P, che nel corso del proprio operato cercherà di accedere ad un file che chiameremo info

- Il real user ID di P è l' UID dell'utente che lo ha generato (**utente1**). Analogamente, il real group ID di P è il GID dell'utente che lo ha generato (**gruppo1**)
- L'effective user ID e l'effective group ID di P dipendono dai due bit speciali, set user id e set group id, associati all'eseguibile exe
- Se set user id è attivo, l'effective user id di P sarà uguale all'**UID del proprietario** del file eseguibile; in caso contrario l'effective user id di sarà uguale al suo **real user id** (cioè utente 1). Stesso discorso vale per set group id
- Se l'effective user id di P coincide con il proprietario di info, il processo acquisisce i **diritti di accesso del proprietario** di info
- Altrimenti, se l'effective group id di P e il gruppo di info coincidono, P acquisisce i **diritti di accesso del gruppo** di utenti associati ad info
- Se nessuna delle due precedenti condizioni è valida, valgono le normali triple di diritti di accesso: l'accesso sarà consentito o meno a seconda della categoria di utenti nella quale ricadono real user id e real group id del processo P

12.1.1 Permessi di accesso ai file

Complessivamente, sono dedicati 12 bit per i permessi.
I primi 3 per il proprietario
I secondi 3 per il gruppo
I terzi 3 per gli altri
Gli ultimi 3 sono rispettivamente il *set user-id*, il *set group-id* e lo *sticky bit*.

12.1.2 Sticky bit

Lo sticky bit permette di richiedere al kernel che l'immagine di un processo resti allocata nell'area di swap anche dopo la sua terminazione. Risulta utile per programmi frequentemente utilizzati.

12.1.3 System call umask()

Questa system call viene utilizzata per asseganre ad un processo la modalità di creazione di un file.

mode_t umask (mode_t mask);

L'argomento mask è format da un OR bit a bit delle nove costanti di permesso di accesso di un file. La funzione ritorna il valore precedente della maschera di creazione dei file.

La maschera è usata ogni qualvolta il processo crea un nuovo file o una nuova directory.
Dato un valore della maschera, i permessi di un file creato sono calcolati usando la seguente operazione bit a bit: ***AND bit a bit tra il complemento di mask e la modalità di accesso specificata in open()***

Esempio:
Se il valore di default di umask viene inizializzato a 022, un nuovo file creato con permessi 666 avrà:

666 & ~022 = 644 = r w – r – – r – –

12.2 Directory

Una directory può essere letta da chiunque abbia i permessi di accesso per lettura.
I permessi di accesso e scrittura per una directory determinano se si possono creare nuovi file nella directory e se si possono cancellare.

Esempio: elencare i file di una directory

```
#include <sys/types.h>
#include <dirent.h>
int main(int argc, char *argv[ ])
{
  DIR *dp;
  struct dirent *dirp;
  if (argc != 2){
       printf("a single argument (the directory name) is required");
       exit(-1);
     }
  if ( (dp = opendir(argv[1])) == NULL){
       printf("can't open %s", argv[1]);
       exit(-1);}
  while ( (dirp = readdir(dp)) != NULL)
       printf("%s\n", dirp->d_name);
   closedir(dp);
  exit(0);
}
```