Programmazione in ambiente UNIX

Dispensa didattica consigliata per i corsi di:

Sistemi Operativi

(corsi di Laurea in Ingegneria Informatica, Elettronica e Telecomunicazioni

Università di Roma "La Sapienza")

Sistemi Operativi I e II modulo

(corso di Diploma in Ingegneria Informatica

Università di Roma "La Sapienza")

Francesco Quaglia Camil Demetrescu

30 Settembre 1999 - Versione 1.0

Contents

1 Una panoramica del sistema UNIX **3**
- 1.1 La struttura del sistema UNIX . . . 3
  - 1.1.1 Avvio del sistema . . . 3
- 1.2 Comandi di UNIX . . . 4
  - 1.2.1 Alcuni comandi fondamentali . . . 5

2 Gestione dei file **17**
- 2.1 Chiamate `creat()`, `open()` e `close()` . . . 17
- 2.2 Chiamate `read()` e `write()` . . . 20
- 2.3 Chiamata `lseek()` . . . 21
- 2.4 Chiamate `link()` e `unlink()` . . . 23
- 2.5 Chiamata `dup()` . . . 24

3 Gestione di processi **27**
- 3.1 Chiamate `fork()` e `wait()` . . . 27
- 3.2 Chiamate `execX()` . . . 29
- 3.3 Un esempio di applicazione . . . 30

4 Costrutti per la comunicazione: code di messaggi **31**
- 4.1 Chiamate `msgget()` e `msgctl()` . . . 31
- 4.2 Chiamate `msgsnd()` e `msgrcv()` . . . 33
- 4.3 Un esempio di applicazione . . . 35

5 Costrutti per la comunicazione: memoria condivisa **39**
- 5.1 Chiamate `shmget()` e `shmctl()` . . . 39
- 5.2 Chiamate `shmat()` e `shmdet()` . . . 41
- 5.3 Un esempio di applicazione . . . 43

6 Costrutti per la comunicazione: PIPE e FIFO **45**
- 6.1 Chiamata `pipe()` . . . 45
  - 6.1.1 Un esempio di applicazione . . . 46
- 6.2 Chiamata `mkfifo()` . . . 46
  - 6.2.1 Un esempio di applicazione . . . 48

7 Costrutti per la sincronizzazione: i semafori **51**
- 7.1 Creazione di un semaforo: chiamata `semget()` . . . 51
- 7.2 Operazioni su un semaforo: chiamata `semop()` . . . 52
- 7.3 Inizializzazione e rimozione di un semaforo: chiamata `semctl()` . . . 53
- 7.4 Un esempio di applicazione . . . 54

8 Gestione di eventi asincroni: i segnali **57**
8.1 Chiamata `kill()` . . . 58
8.2 Chiamata `alarm()` . . . 58
8.3 Chiamata `signal` . . . 58
8.4 Esempi di applicazione . . . 60

9 Costrutti per la comunicazione in sistemi distribuiti: socket **65**
9.1 Le chiamate `socket()` e `bind()` . . . 65
9.2 Comunicazione orientata alla connessione: chiamate accept(), `connect()` e `listen()` . . . 68
9.2.1 Un esempio di applicazione . . . 72
9.3 Comunicazione non orientata alla connessione: chiamate `sendto()` e `recvfrom()` . . . 74

Chapter 1

Una panoramica del sistema UNIX

In questo capitolo verranno forniti concetti di base sulla struttura del sistema operativo UNIX, ponendo particolare attenzione all'insieme dei comandi di base che tale sistema operativo mette a disposizione dell'utente per interfacciarsi con esso.

1.1 La struttura del sistema UNIX

UNIX è un sistema operativo time-sharing e multitasking, formato da un nucleo centrale (il kernel) e da un insieme di programmi che gestiscono le risorse del computer e forniscono servizi agli utenti. Nella maggior parte dei casi, gli utenti del sistema UNIX utillizzano un sottoinsieme di tali programmi, sviluppando una visione soggettiva, e di conseguenza non esausativa, delle potenzialità di questo sistema. Una ben più oggettiva conoscenza è richiesta ai programmatori, il cui compito è quello di produrre nuovi programmi che verranno poi messi a disposizione degli utenti.

Il kernel supporta e coordina tutti gli altri programmi in esecuzione, normalmente denominati processi, e gestisce l'insieme delle risorse fisiche (CPU, memoria, dispositivi di I/O), arbitrandone l'assegnazione ai processi stessi.

I programmi che gestiscono risorse sono denominati "utilities". Essi consentono di eseguire mansioni ausiliare, strettamente connesse alla gestione del sistema, come per esempio la duplicazione del contenuto di un file. Diversamente, programmi che forniscono servizi vengono denominati "applications". Essi permettono all'utente di risolvere problemi non necessariamente collegati al mondo dei computer (ad esempio programmi di videoscrittura o programmi per il fotoritocco), ma la cui soluzione viene in questo modo notevolmente facilitata ed accelerata.

La separazione esistente tra kernel ed utilities distingue, in termini architetturali, UNIX da molti sistemi operativi, conferendogli una maggiore flessibilita, ovvero la capacità di potersi espandere modularmente, integrando nuove funzionalita senza la necessita di modificare, e quindi ricompilare, il kernel stesso od utilities preesistenti.

1.1.1 Avvio del sistema

Quando un sistema UNIX viene avviato, una combinazione di firmware e software trasferisce il kernel in memoria centrale. Esso alloca ed inizializza una tavole dei processi (process table), i cui elementi, denominati process control blocks (PCB), vengono utilizzati per memorizzare informazioni relative ai programmi in esecuzione nel sistema. All'atto della creazione di questa tavola, l'unico programma in esecuzione è il kernel, il cui PCB ha indice zero. Il numero di PCB della tavola rappresenta il numero massimo di processi che possono essere eseguiti concorrentemente. Quando un nuovo processo viene creato, un PCB disponibile gli viene assegnato dal kernel; esso verrà rilasciato all'atto della terminazione del processo, così da poter essere riutilizzato per nuovi processi che verranno creati.

Ogni processo ha un proprio identificatore numerico, denominato *pid* (process identifier), memorizzato nel relativo PCB. Il kernel ha pid 0. Il pid dei processi via via creati, sono assegnati in maniera progressivamente

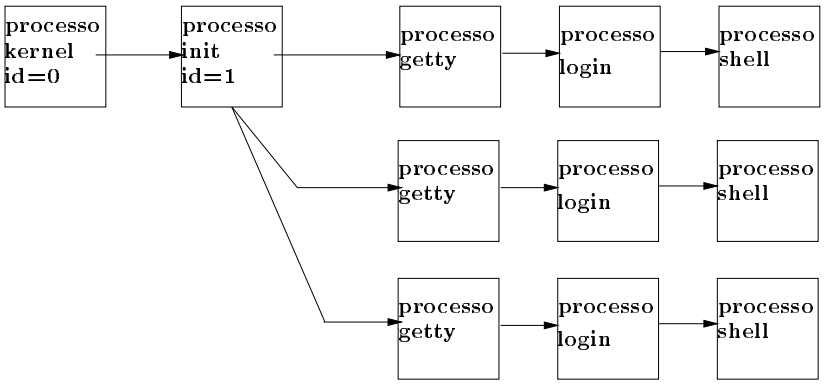

Figure 1.1: Avvio di un sistema UNIX

crescente.

Nella fase di avvio, tre utilities giocano un ruolo fondamentale per predisporre il sistema ad interagire con gli utenti. Esse sono il programma **init**, il programma **getty** ed il programma **login**. **init** viene lanciato dal kernel e gli viene assegnato pid 1. Esso resta in esecuzione fino a quando il sistema operativo viene arrestato. La funzionalità di **init** è quella di lanciare i processi necessari agli utenti per poter accedere al sistema. Come prima azione, **init** legge il file di configurazione `/etc/inittab`, contenente una lista di terminali dai quali è possibile accedere al sistema. Per ciascuno di questi terminali, **init** crea una processo istanza del programma **getty** il quale stampa sul terminale il messaggio `login:`. Quando un utente risponde al messaggio `login:` digitando una stringa di caratteri che rappresenta il proprio identificativo nell'ambito del sistema (*username*), **getty** crea un processo istanza del programma **login** e gli comunica la username, quindi termina. Il nuovo processo ha il compito di verificare se la username ricevuta corrisponde effettivamente ad un utente noto al sistema. Questo viene fatto leggento il file di configurazione `/etc/passwd`, nel quale sono memorizzate le informazioni relative a tutti gli utenti autorizzati all'accesso al sistema. Qualora la username in oggetto non corrisponda ad un utente autorizzato, **login** termina, e **init**, una volta notificato della terminazione, ricomincia la sequenza di passi creando una nuova istanza di **getty** per il terminale in oggetto. Nel caso in cui l'utente venga riconosciuto, **login** stampa il messaggio `password`: e si pone in attesa della password dell'utente, disabilitando l'eco locale in maniera tale da impedire che i caratteri digitati dall'utente appaiano sullo schermo.

Ogni utente registrato stabilisce un programma che deve essere lanciato all'atto del suo ingresso al sistema. Tale programma è specificato nel file `/etc/passwd`. Se la password digitata è corretta, **login** lancia una istanza dell suddetto programma e termina. A questo punto l'utente può interagire con il sistema tramite il programma da lui specificato. Usualmente, il programma specificato dall'utente è un programma *shell*. Una volta lanciato, questo programma permette all'utente di interagire con il sistema e di lanciare ogni altra utility o applicazione. Vi sono vari tipi di programma shell, tra i quali i più utilizzati sono: **C**, **Bourne**, **bash** e **Korn**.

La sequenza delle azioni corrispondenti all'avvio del sistema ed all'ingresso di utenti è mostrata in figura 1.1.

1.2 Comandi di UNIX

Il compito di una shell è quello di accettare e di interpretare stringhe di caratteri immesse dall'utente. Ogni stringa è la specifica di un comando la cui esecuzione viene richiesta dall'utente. I comandi si distinguono in: *comandi interni* e *comandi esterni*. L'esecuzione di un comando interno non necessita il lancio di un nuovo programma, e quindi la creazione di un nuovo processo. Esso viene eseguito direttamente dalla shell. Diversamente, l'esecuzione di un comando esterno, necessita il lancio di un programma.

Sintatticamente, la specifica di un comando è costituita come segue:

$$\texttt{nome-comando [argomento}_1\texttt{, argomento}_2\texttt{, ..., argomento}_n\texttt{]}$$

Nome	Descrizione
alias	crea alias di stringhe
apropos	cerca stringhe nel manuale in linea
cal	mostra il calendario
cat	riversa il contenuto di un file nello standard output
cd	cambia la directory di lavoro corrente della shell
chgrp	cambia il gruppo proprietario di un file
chmod	cambia i privilegi di accesso di un file
cp	copia un file
date	mostra la data e l'ora corrente
du	riporta l'occupazione di spazio su disco
export	assegna stringhe a variabili di ambiente (*bash*)
from	mostra i mittenti delle ultime mail ricevute e non ancora lette
host	richiede al DNS di default l'indirizzo IP di una macchina
id	mostra o cambia il nome dell'host su cui si è loggati
hostname	mostra l'*user id* ed il *group id* di un utente
jobs	mostra i processi a carico della shell da cui viene lanciato il comando
kill	invia segnali a processi
killall	uccide tutti i processi dell'utente
less	mostra per pagine il contenuto di un file
logname	mostra il nome dell'utente
ls	lista i file di una directory
pwd	mostra il path della directory corrente

Table 1.1: Lista dei principali comandi UNIX (a-l)

Le shell permettono di combinare più comandi a formare una riga di comandi complessa, in cui più processi vengono creati simultaneamente. I comandi possono essere concatenati mediante il carattere '|' (pipe) in modo che lo standard output dell'uno venga usato come standard input del successivo. Se le righe di comandi vengono terminate con '&' (background) la shell stessa può continuare ad accettare altri comandi durante l'esecuzione dei primi.

L'aggiunta di comandi interni necessita la modifica del programma shell; per questo motivo l'uso di tali comandi è limitato solo alle funzionalità di base che, per vincoli di natura tecnica, non possono essere relizzate tramite comandi esterni. Ad esempio, ad ogni processo, e quindi anche ad un'istanza della shell, è associata una *directory corrente* la quale contiene files che possono essere riferiti dal processo stesso tramite il solo utilizzo del loro nome, senza la necessità di specificare la loro posizione all'interno del file system. La directory corrente è memorizzata nella *variablile d'ambiente* `CDPATH`. Un tipico comando interno è il comando `cd`, che ha la funzione di modificare il valore di `CDPATH`. Questo comando non può essere implementato come comando esterno poichè, in tal caso, la sua esecuzione genererebbe un nuovo processo con proprie variabili d'ambiente, diverse da quelle della shell, ed ogni processo può accedere esclusivamente alle proprie variabili d'ambiente.

Nel paragrafo seguente vengono presentati alcuni comandi fondamentali di UNIX. Tali comandi costituiscono parte dell'interfaccia che il sistema mette a disposizione degli utenti per poter interagire con esso.

1.2.1 Alcuni comandi fondamentali

In questa sezione presentiamo una lista di comandi utili che possono essere utilizzati nella shell di comandi. La trattazione comprende la sintassi del comando, la descrizione del suo scopo e dei suoi argomenti, una lista di comandi correlati ed eventualmente uno o più esempi d'uso. Assumeremo che '>' sia il 'prompt'

presentato dalla shell quando è in attesa di comandi dall'utente. La lista completa dei comandi è presentata in Tabella 1.1 ed in Tabella 1.2.

Nome	Descrizione
man	consulta il manuale in linea relativo ad un comando
mkdir	crea una directory
mv	rinomina o sposta un file o una directory
passwd	avvia la procedura di sostituzione o inserimento della password
ps	mostra i processi in esecuzione
rehash	riaggiorna la tabella dei file eseguibili
rlogin	apre una sessione in remoto su un host
rm	rimuove un file
rmdir	rimuove una directory
setenv	assegna stringhe a variabili di ambiente (*tcsh*)
tail	mostra le linee terminali di un file di testo
telnet	comincia una sessione di telnet su una macchina
touch	aggiorna alla data corrente un file
unalias	elimina un alias
uname	mostra il nome del sistema operativo
unsetenv	elimina una variabile di ambiente
w	lista tutti gli utenti connessi sull'host ed i comandi che stanno eseguendo
which	verifica che un comando sia visibile
who	lista tutti gli utenti loggati sull'host
who am i	mostra lo *user id* con cui l'utente si è loggato
whoami	mostra lo *user id* con cui l'utente è attualmente loggato

Table 1.2: Lista dei principali comandi UNIX (m-z)

alias `stringa1 'stringa2'`	
Descrizione	Crea un alias per `stringa2`, sostituendo tutte le occorrenze future di `stringa1` (in una linea di comandi) con `stringa2`
Argomenti	stringa1: nome dell'alias 'stringa2': stringa di cui viene creato l'alias
Comandi correlati	**unalias**
Esempio	`> alias Salta 'cd /usr/src/include'` Salta è ora un sinonimo di: "`cd /usr/src/include`"

apropos `stringa`	
Descrizione	Offre una lista di tutte le voci del manuale in linea che contengano la stringa 'stringa'
Argomenti	stringa: nome della stringa da cercare nel manuale
Comandi correlati	**man**
Esempio	`> apropos man`

cal [mese] anno	
Descrizione	mostra il calendario (cal sta per "calendar")
Argomenti	vedi esempio
Comandi correlati	**date**
Esempio 1	`> cal 3 1997` mostra il calendario del mese di marzo (3) dell'anno 1997
Esempio 2	`> cal 1997` mostra il calendario di tutti i mesi dell'anno 1997

cat nomefile	
Descrizione	Riversa il contenuto di un file nello standard output. Non esegue paginazione (vedi more).
Argomenti	nomefile: nome del file da riversare nello standard output
Comandi correlati	**more**

cd [nomedir]	
Descrizione	Cambia la directory di lavoro corrente della shell (cd sta per change directory)
Argomenti	`nomedir`: (opzionale) percorso relativo o assoluto nel file system della directory che deve diventare la directory di lavoro corrente per la shell
Esempio 1	`> cd /etc` Passa dalla directory attuale alla directory "`/etc`"
Esempio 2	`> cd ..` Passa dalla directory attuale alla directory padre nell'albero delle directory
Esempio 3	`> cd ~`oppure `> cd` Passa alla home directory dell'utente corrente

chgrp gruppo nomefile	
Descrizione	Elegge il gruppo "`gruppo`" a gruppo proprietario del file "`nomefile`" (chgrp sta per "change group")
Argomenti	`gruppo`: nome del nuovo gruppo proprietario `nomefile`: nome del file di cui cambiare il gruppo proprietario
Comandi correlati	**chmod, chown**

chmod perm nomefile	
Descrizione	Cambia in base a "`perm`" i privilegi di accesso del file "`nomefile`" (chmod sta per "change mode")
Argomenti	`perm`: può essere una maschera di permessi (es. 0777) oppure una espressione regolare del tipo: `[u][g][o]{+|-}[r][w][x]` (es. go+rwx) dove `u`=*user*, `g`=*group*, `o`=*others*, `r`=*read*, `w`=*write* e `x`=*execute* `nomefile`: nome del file di cui cambiare i permessi
Comandi correlati	**chgrp, chown**

cp file1 file2	
Descrizione	Copia un file
Argomenti	`file1`: nome del file da copiare `file2`: percorso/nuovo nome del file copiato
Comandi correlati	-

date	
Descrizione	Mostra la data, l'ora, e il fuso orario (*Nota*: MET sta per Middle Europe Timezone)
Argomenti	-
Comandi correlati	**cal**

du [-k] [-s] [*]	
Descrizione	Riporta l'occupazione di spazio su disco (du sta per disk usage)
Argomenti	`-k`: l'occupazione deve essere espressa in Kbyte `-s`: l 'occupazione deve essere calcolata su tutto il sottoalbero radicato alla directory corrente
	`*`: vengono mostrate tutte le sotto-directories
Comandi correlati	-

export var value	
Descrizione	Assegna valori a variabili di ambiente Nota: funziona sulla shell *bash*; per la shell *tcsh* usare `setenv`
Argomenti	`var`: nome della variabile di ambiente da definire ed assegnare `value`: stringa da assegnare alla variabile di ambiente `var`
Comandi correlati	**unsetenv, setenv**

from	
Descrizione	Mostra i mittenti delle ultime mail ricevute e non ancora lette
Argomenti	-
Comandi correlati	-

host nome_macchina	
Descrizione	Richiede al DNS di default l'indirizzo IP di una macchina (eventualmente completa di dominio) e lo mostra sul video
Argomenti	nome_macchina: nome della macchina di cui si richiede l'indirizzo IP
Comandi correlati	-

hostname `[nuovo_nome]`	
Descrizione	mostra o cambia il nome dell'host su cui si è loggati
Argomenti	`nuovo_nome` (opzionale): nuovo nome della macchina (richiede i privilegi di root)
Comandi correlati	-

id `[username]`	
Descrizione	mostra l'*user id* ed il *group id* dell'utente `username` se specificato, altrimenti dell'utente correntemente loggato
Argomenti	`username` (opzionale): nome dell'utente di cui si vuole l'uid ed il gid
Comandi correlati	-

jobs	
Descrizione	mostra i processi a carico della shell da cui viene lanciato il comando, con il loro numero d'ordine (è il numero di job, ed è diverso dal pid)
Argomenti	-
Comandi correlati	-

kill `-segnale pid`	
Descrizione	invia un segnale ad un processo
Argomenti	segnale: numero del segnale da inviare pid: identificatore del processo a cui inviare il segnale
Comandi correlati	**killall, ps, jobs**

killall	
Descrizione	Uccide tutti i processi dell'utente che esegue il comando ad eccezione della shell da cui è emesso il comando stesso
Argomenti	-
Comandi correlati	**kill**

less `nomefile`	
Descrizione	Mostra il contenuto di un file sullo standard output paginandolo (è il more della GNU). Rispetto a more permette di scorrere avanti e indietro il testo con i tasti freccia.
Argomenti	`nomefile`: nome del file da paginare
Comandi correlati	**more, tail**

logname	
Descrizione	Mostra il nome dell'utente
Argomenti	-
Comandi correlati	**whoami, who am i**

ls [-opzioni] [nome_percorso]	
Descrizione	Lista tutti i file della directory corrente ad eccezione dei file nascosti, il cui nome comincia con un punto (ls sta per list)
Argomenti	`opzioni`: `a|l|F|1` `a` (all): lista tutti i files compresi quelli nascosti `l` (long): presenta una lista completa delle caratteristiche dei files `F` (format): presenta una lista formattata dei file `1` (one column): lista tutti i files in una sola colonna (non funziona con altri numeri) `nome_percorso`: percorso della directory da listare
Comandi correlati	-

man comando	
Descrizione	Presenta le pagine del manuale in linea relative ad un comando (man sta per manual)
Argomenti	`comando`: comando di cui si vogliono le istruzioni
Comandi correlati	**apropos**

mkdir nome_directory	
Descrizione	Crea una directory (mkdir sta per sta per make directory)
Argomenti	`nome_directory`: percorso/nome della directory da creare
Comandi correlati	**rmdir**

more nomefile	
Descrizione	Mostra il contenuto di un file sullo standard output paginandolo
Argomenti	`nomefile`: nome del file da paginare
Comandi correlati	**less, tail**

mv file1 file2	
Descrizione	Rinomina o sposta un file o una directory
Argomenti	`file1`: nome del file da spostare o rinominare `file2`: se esiste una directory con questo nome, file1 viene spostato in file2, altrimenti file1 viene rinominato come file2
Comandi correlati	-

passwd	
Descrizione	Avvia la procedura di sostituzione o inserimento della password. Nota: non funziona in AFS (usare kpasswd)
Argomenti	-
Comandi correlati	-

ps [-opzioni]	
Descrizione	Mostra i processi in esecuzione figli della shell corrente
Argomenti	`opzioni`: `a`: mostra tutti i processi, non solo quelli della shell corrente `u`: mostra tutti i processi dell'utente corrente `f`: mostra i processi in "full listing" (-f), cioè con la linea di comando che li ha chiamati `aux`: mostra i processi con la loro occupazione di memoria e la percentuale di CPU in uso
Comandi correlati	-

pwd	
Descrizione	Mostra il path della directory corrente (pwd sta per print working directory). E' equivalente ad eseguire: `echo $PWD`.
Argomenti	-
Comandi correlati	-

rehash	
Descrizione	Rilegge il path, aggiornando la tabella dei file eseguibili disponibili (che diventano visibili tramite il comando which ed eseguibili tramite chiamata diretta).
Argomenti	-
Comandi correlati	**which**

rlogin host.dominio [-l nome_utente]	
Descrizione	Apre una sessione in remoto sull'host in questione (chiede prima la password).
Argomenti	`host.dominio`: macchina su cui aprire una sessione in remoto `-l nome_utente` (opzionale): utente a nome del quale aprire la sessione (se manca è l'utente corrente)
Comandi correlati	-

rm nomefile	
Descrizione	Rimuove un file
Argomenti	nomefile: nome del file da rimuovere
Comandi correlati	-

rmdir nomedir	
Descrizione	Rimuove una directory
Argomenti	nomedir: nome della directory da rimuovere (se vuota)
Comandi correlati	**mkdir**

setenv `var stringa`	
Descrizione	Assegna stringhe a variabili di ambiente (setenv sta per set environment) Nota: funziona sulla shell *tcsh*; per la shell *bash* usare `export`
Argomenti	`var`: nome della variabile di ambiente da definire ed assegnare `stringa`: stringa da assegnare alla variabile di ambiente `var`
Comandi correlati	**unsetenv, export**

tail `[{+|-}num[c]] nomefile`	
Descrizione	Mostra le linee terminali di un file di testo
Argomenti	`-num`: mostra le ultime `num` linee del file nomefile `+num`: mostra le linee terminali del file nomefile a partire dalla `num`-esima `c`: num indica numero di caratteri e non di linee `nomefile`: nome del file da mostrare
Comandi correlati	**more, less**

telnet `nomehost`	
Descrizione	Comincia una sessione di telnet su una macchina
Argomenti	`nomehost`: nome del host a cui connettersi
Comandi correlati	-

touch `nomefile`	
Descrizione	Aggiorna alla data corrente un file (se nomefile è assente, viene creato un file vuoto)
Argomenti	`nomefile`: nome del file da aggiornare
Comandi correlati	-

unalias `nomealias`	
Descrizione	Elimina un alias
Argomenti	`nomehost`: nome dell'alias da eliminare
Comandi correlati	**alias**

uname `[-a]`	
Descrizione	Mostra il nome del sistema operativo (uname sta per UNIX name)
Argomenti	`-a`: mostra il nome del sistema operativo, con tutti i dettagli relativi (versione, hostname, ...)
Comandi correlati	-

unsetenv var	
Descrizione	Elimina una variabile di ambiente (unsetenv sta per unset environment)
Argomenti	`var`: nome della variabile da eliminare
Comandi correlati	**setenv**

w	
Descrizione	Lista tutti gli utenti connessi sull'host ed i comandi che stanno eseguendo
Argomenti	-
Comandi correlati	**who**

which `comando`	
Descrizione	Verifica che un comando sia visibile e segnala il percorso della directory in cui lo ha trovato (o eventualmente a quale alias corrisponde)
Argomenti	`comando`: nome del comando da verificare
Comandi correlati	**rehash**

who `[-T]`	
Descrizione	Lista tutti gli utenti loggati sull'host
Argomenti	`-T` (opzionale): include nella lista dettagli sui loro terminali
Comandi correlati	**w, whoami, who am i**

who am i	
Descrizione	Mostra lo userid con cui l'utente si è loggato sulla macchina, la sua console e la data del suo login
Argomenti	-
Comandi correlati	**logname, w, who, whoami**

whoami	
Descrizione	Mostra lo userid con cui l'utente è attualmente loggato
Argomenti	-
Comandi correlati	**logname, w, who, who am i**

Prefazione alla seconda parte

Nel capitolo precedente è stato introdotto il concetto di "comando" quale mezzo fondamentale per l'interazione di un utente con il sistema operativo. L'invocazione di un comando da parte di un utente corrisponde alla esecuzione di un particolare programma.

Nei capitoli seguenti ci occuperemo di alcuni aspetti fondamentali inerenti il problema della progettazione e realizzazione di programmi nel sistema UNIX. In particolare, verrà trattato il problema dell'interfaccia tra il *programmatore* ed il sistema operativo. Tale problema è di fondamentale importanza poichè i programmi hanno bisogno, almeno nella maggioranza dei casi, della esecuzione di mansioni ausiliarie strettamente legate alla gestione delle risorse del sistema. Il meccanismo che permette ad una application di invocare una mansione ausiliaria è il cosiddetto meccanismo delle *system calls* (chiamate di sistema). Queste costituiscono quindi l'interfaccia tra una application ed il sistema operativo, e quindi, in maniera indiretta, l'interfaccia tra un programmatore ed il sistema.

Le chiamate di sistema sono comunemente suddivise in categorie a seconda del tipo di servizio ausiliario ed esse associato. In UNIX si hanno le seguenti tree categorie principali:

- gestione di file;
- gestione di processi;
- comunicazione e sincronizzazione tra processi.

I capitoli seguenti sono dedicati alla descrizione delle di chiamate di sistema appartenenti a queste categorie. La nostra trattazione farà riferimento al linguaggio di programmazione C. Similmente a quanto fatto per i comandi nel capitolo precedente, le chiamate di sistema verranno descritte in forma tabellare. Qualora necessario, la descrizione viene corredata da spiegazioni aggiuntive ed esempi di utilizzo.

Chapter 2

Gestione dei file

Ogni operazione di input/output (lettura/scrittura) afferisce ad uno specifico file. Nel sistema UNIX i file possono essere creati e rimossi dinamicamente, e, qualora esistenti, possono essere aperti per potervi leggere e scrivere dati.

Il kernel associa ad ogni processo una tabella dei file aperti dal processo stesso. Quando un file viene creato o, se già esistente, viene aperto, il sistema associa ad esso un numero intero non negativo, denominato *descrittore di file*, che identifica un canale di input/output. Il descrittore di file rappresenta un collegamento, tra il processo e il file; esso serve ad indicizzare la tabella dei file aperti del processo.

Quando un processo viene creato, le prime tre locazioni della sua tabella dei file vengono inizializzate come segue: il descrittore con indice 0 viene associato allo standard input ed in genere identifica il canale di input relativo alla tastiera, i descrittori con indice 1 e 2 vengono associati, rispettivamente, allo standard output ed allo standard error, che identificano normalmente canali verso il video. Ogni canale di input/output permette operazioni di input, di output o entrambe.

Ogni elemento della tabella dei file aperti contiene un puntatore di lettura/scrittura, denominato *file pointer*, relativo al file aperto. Un'operazione di input/output incrementa il puntatore di lettura/scrittura che indica il numero di byte letti o scritti in modo che la successiva operazione di lettura/scrittura sullo stesso file avverrà a partire dalla posizione identificata dal nuovo valore del puntatore.

Poichè il numero di file che un processo può tenere aperti contemporaneamente è limitato (il limite dipende dalla dimensione della tabella dei file aperti) è buona regola che ogni file al quale non si ha necessità di accedere in un immediato futuro venga chiuso.

Nelle sezioni seguenti verranno descritte le chiamate di sistema per creare, aprire, e chiudere i file, e per leggere e scrivere su di essi. Verrà anche introdotta la chiamata di sistema che serve per modificare il valore del puntatore di lettura e scrittura in modo da potersi posizionare in un qualsiasi punto del file. Infine, verranno presentate le chiamate di sistema che permettono di associare più nomi ad uno stesso file (cioè creare degli alias per quel file) e la chiamata di sistema per duplicare il descrittore di un file (tale duplicazione permette l'accesso allo stesso file tramite più di un canale di input/output).

2.1 Chiamate `creat()`, `open()` e `close()`

Per creare un file in un file system UNIX la chiamata di sistema è la `creat()` descritta come segue:

int **creat**(char *file_name, int mode)	
descrizione	invoca la creazione un file
argomenti	*file_name: puntatore alla stringa di caratteri che definisce il nome del file da creare mode: specifica i permessi di accesso al file da creare
restituzione	-1 in caso di fallimento

Il primo parametro è un puntatore ad una stringa che specifica il nome del file da creare (in realtà tale stringa specifica il pathname completo per il file espresso o in termini di path assoluto o di pth relativo).

```
void main() {
    if (creat("pippo",0666) == -1) {
        printf("Errore nella chiamata creat \n");
        exit(1);
    }
}
```

Figure 2.1: Un esempio di utilizzo della chiamata creat()

Il secondo parametro indica la specifica dei permessi di accesso al file. Qualora il file da creare già esista, l'effetto della chiamata è quello di rimuovere il contenuto del file preesistente, lasciando però inalterati i diritti di accesso del file preesistente (i nuovi diritti specificati dal parametro mode vengono ignorati). Se la chiamata va a buon fine, essa ritorna un numero intero non negativo rappresentante il descrittore di file per l'accesso al file creato. La chiamata fallisce e riporta il valore -1 se si verifica una delle seguenti condizioni:

- parte del prefisso della stringa *file_name o non è una directory o non esiste, oppure la stringa *file_name corrisponde ad una directory esistente;
- non si hanno i permessi di accesso ad una delle directory specificate in file_name. Questo accade sia nel caso in cui non si hanno permessi di scrittura nella directory in cui il file deve essere creato oppure quando la directory stessa risiede su un file system a sola lettura;
- *file_name è un puntatore nullo;
- il file da creare già esiste ed: o è attualmente utilizzato da altri processi o è negato il permesso di accesso;
- è già stato raggiunto il numero massimo di file che possono essere aperti contemporaneamente dal processo;
- *file_name punta ad un indirizzo di memoria non valido per quel processo.

In Figura 2.1 viene riportato un esempio di utilizzo della chiamata `creat()`. In questo esempio, il file che si intende creare gha il nome "pippo", ed i permessi da associare al file sono tali che tutti gli utenti possono accedere al file sia in lettura che in scrittura.

E' da notare che quando un file viene creato, esso è immediatamente disponibile per operazioni di input output poichè al processo che lo crea viene immediatamente restituito un descrittore valido per l'accesso al file. Nel caso in cui si voglia effettuare input/output da un file esistente, bisogna preventivamente aprire tale file in modo da ottenere per esso un descrittore valido. La chiamata per aprire un file già esistente è la creat() descritta come segue:

int **open**(char *file_name, int option_flags [, int mode])	
descrizione	invoca l'apertura di un file
argomenti	*file_name: puntatore alla stringa di caratteri che definisce il nome del file da aprire option_flags: specifica la modalità di apertura mode: specifica dei permessi in caso il file venga creato
restituzione	un intero positivo corrispondente al descrittore del file aperto in caso di successo; -1 in caso di fallimento

Il primo parametro della chiamata creat() è un puntatore alla stringa che definisce il nome (ovvero il pathname) del file che si intende aprire. option_flags specifica la modalità di apertura del file. Il suo valore è espresso come una combinazione, ottenuta tramite l'operatore '|', di alcuni dei seguenti valori definiti nell'header file `fnctl.h`:

O_RDONLY: apertura del file in sola lettura;

O_WRONLY: apertura del file in sola scrittura;

O_RDWR: apertura in lettura e scrittura;

O_APPEND: apertura del file con puntatore alla fine del file; ogni scrittura sul file sarà effettuata a partire dalla fine del file;

O_CREAT : crea il file con modalità d'accesso specificate da mode sole se esso stesso non esiste;

O_TRUNC : elimina il contenuto del file se esso già esistente.

O_EXCL : (exclusive) serve a garantire che il file sia stato effettivamente creato dal programma che effettua la chiamata.

Come il lettore avrà già intuito dalla spiegazione dei possibili valori da assegnare ad option_flag, la chiamata `open()` ha l'effetto di creare il file in oggetto qualora esso non esista. In tal caso, i permessi di accesso sono specificati dal valore del parametro opzionale mode. In caso di successo, la chiamata `open()` ritorna un descrittore valido per l'accesso al file. La chiamata fallisce, restituendo di conseguenza il valore -1, nei seguenti casi:

- parte del prefisso della stringa *file_name non è una directory o non si può accedere ad una delle directory specificate dalla stringa *file_name;
- O_CREAT non è presente nella specifica di option_flags e il nome del file non esiste;
- option_flags richiede un tipo di accesso che i permessi del file negano;
- è già stato raggiunto il numero massimo di file che possono essere aperti contemporaneamente dal processo;
- il file in oggetto è un file testo condiviso che è attualmente in uso;
- file_name punta ad un indirizzo di memoria non valido per il processo chiamante;
- nella definizione di option_flags sono specificate O_CREAT e O_EXCL ed il file già esiste.

Occorre notare che una chiamata del tipo `open("pippo",O_WRONLY|O_TRUNC|O_CREAT,0660)` è del tutto equivalente a `creat("pippo",0660)`, infatti se il file "pippo" esiste la chiamta `open()` lo apre ma lo tronca mettendo il puntatore all'inizio, come farebbe la chiamata `creat()`, mantenendo i precedenti diritti di accesso. Se il file non esiste entrambe lo creano con gli stessi diritti di accesso e lo aprono in scrittura.

Una volta che un file non debba più essere utilizzato per ulteriori operazioni di input/output, esso può essere chiuso un processo che tiene correntemente aperto tramite la chiamata `close()` descritta come segue:

int **close**(int ds_file)	
descrizione	invoca la chiusura di un file
argomenti	ds_file: descrittore di file associato al file da chiudere
restituzione	-1 in caso di fallimento

La chiamata fallisce, restituendo quindi -1, qualora ds_file non corrisponde ad un canale di input/output aperto. Notare che quando un processo termina, la chiusura di un file lasciato aperto viene effettuata automaticamente dal sistema operativo.

In Figura 2.2 viene mostrato un esempio di utilizzo congiunto delle chiamate `open()` e `close()`. Viene effettuata l'apertura di un file dal nome "`/home/usr/quaglia/tmp/pippo`"; in caso di successo, il file viene poi richiuso; in caso di fallimento viene invece mandato in output (sullo standard output) un messaggio di errore tramite la funzione di libreria `printf()`.

```
main() {
    int ds_file;
    ds_file = open("/home/usr/quaglia/tmp/pippo", O_RDWR);
    if (ds_file == -1) printf("Errore nella chiamata open \n");
    else               close(ds_file);
}
```

Figure 2.2: Un esempio di utilizzo delle chiamate open() e close()

2.2 Chiamate `read()` e `write()`

Nella sezione precedente abbiamo visto come poter creare, aprire e chudere file. In questa sezione ci occuperemo di chiamate per effettuare l'input/output su di essi, cioè per leggere e scrivere su file.

La chiamata di sistema per leggere da un file aperto è la `read()` descritta come segue:

int **read**(int ds_file, char *buffer_pointer, unsigned transfer_size)	
descrizione	invoca la lettura di un dato numero di caratteri (byte) da un file
argomenti	file_descriptor: descrittore valido per il file da cui si vuole leggere *buffer_pointer: puntatore all'area di memorie nella quale i caratteri letti devono essere bufferizzati transfer_size: definisce il numero di caratteri (byte) che si vogliono leggere
restituzione	un intero positivo corrispondente al numero di caratteri effettivamente letti in caso di successo; -1 in caso di fallimento

Il primo parametro corrisponde al descrittore di file dal quale si vuole leggere (la coniscenza del valore del descrittore richiede la preventiva apertura del file). Il secondo parametro è l'indirizzo dell'area di memoria ove si intende bufferizzare i caratteri, cioè i byte, letti. Il terzo parametro indica quanti byte devono essere letti dal file. In caso d successo, la chiamata `read()` restituisce il numero di byte effettivamente letti dal file. Essa fallisce, restituendo di conseguenza il valore -1, nei seguenti casi:

- ds_file non corrisponde ad un canale di input/output aperto;
- il canale identificato da ds_file non permette la lettura;
- il buffer puntato da *buffer_pointer ed avente dimensione transfer_size non è contenuto interamente all'interno dello spazio di indirizzamento del processo.

Nel caso in cui il file in oggetto contenga un numero X<transfer_size di caratteri, solo X caratteri vengono letti e trasferiti nel buffer puntato da *buffer_pointer e la chiamata, qualora non vi sia fallimento, restituisce il valore X. Nel caso in cui il numero di caratteri letti è superiore al numero massimo di caratteri che possono essere contenuti nel buffer, i caratteri eccedenti vengono persi. Inoltre, esiste il rischio che i caratteri in eccesso vengano scritti in una area di memoria esterna al buffer. Se tale area di memoria appartiene al processo la chaimata non fallisce, ma ovviamente il contenuto di locazioni di memoria proprie del processo può venire modificato erroneamente. Come già spiegato, se l'area di memoria non appartiene al processo la chiamata fallisce.

In Figura 2.3, viene mostrato un semplice esempio d'uso della chiamata `read()`.

```
#include <fcntl.h>                      /* per la definizione di option_flags     */
#include <stdio.h>

main() {
    int  ds_file;
    char buffer[1024];

    ds_file = open("pippo",O_RDONLY);   /* apertura del file in sola lettura      */
    read(ds_file,buffer,10);            /* trasferimento dei primi 10 caratteri   */
                                        /* del file "pippo" nel buffer di memoria */
    buffer[10] = '\0';
    printf("ho letto: %s\n", buffer);   /* scrittura buffer su standard output    */
    close(ds_file);                     /* chiusura del file                      */
}
```

Figure 2.3: Un esempio di uso della chiamata read()

int **write**(int file_descriptor, char *buffer_pointer, unsigned transfer_size)	
descrizione	invoca la scrittura di un dato numero di caratteri su un file
argomenti	file_descriptor: descrittore di file che identifica il canale di input/output associato al file da cui si vuole scrivere *buffer_pointer: puntatore all'area di memoria dalla quale vengono prelevati i caratteri che si vogliono scrivere transfer_size: definisce il numero di caratteri che si vogliono scrivere sul file
restituzione	un intero positivo corrispondente al numero di caratteri effettivamente scritti oppure -1 in caso di fallimento

La chiamata `write()` fallisce, restituendo il valore -1, nei seguenti casi:

- file_descriptor non corrisponde ad un canale input/output aperto;
- il canale associato a file_descriptor non permette la scrittura;
- il buffer definito dal buffer_pointer ed avente dimensione transfer_size non è contenuto interamente all'interno dello spazio degli indirizzi del processo.

Se la dimensione del buffer fosse più piccola di tranfer_size, l'effetto della chiamata sarebbe quello di scrivere sul file caratteri contenuti in aree di memoria distinte da quella relativa al buffer.

Notare che l'apertura di un file esistente e la scrittura di X caratteri su tale file ha l'effetto di sostituire i primi X caratteri del file con quelli che si intende scrivere. Tale scrittura non risulta, quindi, in un troncamento del file originario.

Vediamo in Figura 2.4 un esempio di programma che utilizza la chiamata `write()`. Il programma copia il contenuto di un file in un altro. I nomi dei file vengono passati al programma come secondo e terzo argomento della riga di comando.

2.3 Chiamata `lseek()`

Come già accennato, il sistema operativo assicia a ciascun canale di input/output un intero, chiamato anche puntatore di lettura/scrittura (file pointer), indicare la posizione dove avverrà la prossima lettura o scrittura all'interno del file. E' possibile modificare il valore del file pointer, e quindi posizionarsi in punti particolari del file in oggetto tramite la seguente system call:

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

#define BUFSIZE 1024

int main(int argc, char *argv[]) {
    int sd, dd, size, result;
    char buffer[BUFSIZE];

    /* Controllo sul numero di argomenti */
    if (argc != 3) {
        printf("usage: copia source target\n");
        exit(1);
    }

    /* Apertura del file da copiare in sola lettura */
    sd=open(argv[1],O_RDONLY);
    if (sd == -1) {
        perror(argv[1]);
        exit(1);
    }

    /* Creazione del file destinazione */
    dd=open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,0660);
    if (dd == -1) {
        perror(argv[2]);
        exit(1);
    }

    /* Qui iniziano le operazioni di copia */
    do {
        /* Lettura fino ad un massimo di BUFSIZE caratteri */
        size=read(sd,buffer,BUFSIZE);
        if (size == -1) {
            perror(argv[1]);
            exit(1);
        }

        result = write(dd,buffer,size);
        if (result == -1) {
            perror(argv[2]);
            exit(1);
        }

    } while(size > 0);

     close(sd);
     close(dd);
}
```

Figure 2.4: Un esempio di uso della chiamata write()

int **lseek**(int file_descriptor, long offset, int option)	
descrizione	invoca la modifica del file pointer
argomenti	file_descriptor: descrittore di file che identifica il canale di input/output associato al file del quale si vuole modificare il file pointer offset: numero di caratteri di cui viene spostato il file pointer option: tipo di spostamento da effettuare
restituzione	nuovo valore del file pointer, valutato in termini di caratteri dall'inizio del file, oppure -1 in caso di fallimento

option può assumere i seguenti valori: 0 (spostamento a partire da inizio file) 1 (spostamento a partire dal valore corrente del file pointer) e 2 (spostamento a partire dalla fine del file).

Il fallimento della chiamata lseek() comporta che il file pointer non venga modificato. Tale fallimento si verifica nei seguenti casi:

- file_descriptor non corrisponde a nessun canale di input/output aperto;
- option ha un valore diverso da 0,1 o 2;
- il nuovo file_pointer verrebbe avere un valore negativo.

Vediamo alcuni esempi:

```
lseek(fd, 10, 0);  /* Spostamento di 10 byte dall'inizio di fd                    */
lseek(fd, 20, 1);  /* Spostamento di 20 byte in avanti dalla posizione corrente   */
lseek(fd, -10, 1); /* Spostamento di 10 byte all'indietro dalla posizione corrente */
lseek(fd, -10, 2); /* Spostamento di 10 byte all'indietro dalla fine del file     */
lseek(fd, -10, 0); /* Fallisce e il valore del file pointer resta uguale          */
```

2.4 Chiamate `link()` e `unlink()`

Il sistema UNIX permette di dare più nomi ad uno stesso file (o directory) tale operazione è detta aliasing. Essa è realizzata tramite la seguente chiamata di sistema:

int **link**(char *path_name, char *alias_name)	
descrizione	invoca la creazione di un alias
argomenti	*path_name: puntatore alla stringa di caratteri che definisce il nome del file (o directory) di cui si vuole creare un alias *alias_name: puntatore alla stringa di carateri che definisce il nome dell'alias
restituzione	0 in caso di successo, -1 in caso di fallimento

La chiamata fallisce, e di conseguenza nessun alias viene creato se si verifica una delle seguenti condizioni:

- un componente del path name o non è una directory, o non esiste, o supera i limiti;
- *alias_name punta ad una stringa di caratteri che identifica un alias già esistente; non esiste;
- *path_name punta ad una stringa di caratteri ed il processo non è un processo del superutente (questo implica che alias di directory possino essere creati esclusivamente dal superutente);
- la directory di destinazione per l'alias non è accessibile in scrittura.

E' da notare che molte implementazioni richiedono che che sia path_name che l'alias risiedano sul medesimo file system.

Così come vengono creati, gli alias possono essere anche rimossi, questo avviene tramite la seguente chiamata di sistema:

```
#include <stdio.h>

main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr,"uso: %s vecchionome nuovonome \n",argv[0]);
                                /* La chiamata giusta e' solo quella con la sequenza:  */
        exit(1);                /* nome_eseguibile vecchio_file nuovo_file             */
    }
    if (link(argv[1],argv[2]) == -1) {
        perror("link");         /* Se la chiamata link fallisce allora esegue          */
        exit(1);                /* perror ed esce, altrimenti crea il link             */
    }
    if (unlink(argv[1]) == -1) {
        perror("unlink");       /* Se la chiamata unlink fallisce allora esegue perror */
        exit(1);                /* ed esce, altrimenti il vecchio link viene eliminato */
    }
}
```

Figure 2.5: Esempio di uso di link() e unlink() per la rinominazione di un file

int **unlink**(char *alias_name)	
descrizione	invoca la rimozione di un alias
argomenti	*alias_name: puntatore alla stringa di caratteri che definisce il nome dell'alias che si vuole rimuovere
restituzione	0 in caso di successo, -1 in caso di fallimento

Tale chiamata ha l'effetto di rimuovere l'alias, e quindi la corrispondente directory entry, ed anche di decrementare il contatore di riferimenti al corrispondente file (cioè il numero di nomi che lo referenziano). Se il numero di riferimenti raggiunge il valore 0, il corrisponente file viene rimosso dal file system.

La chiamata fallisce se si verifica una delle seguenti condizioni:

- *alias_name identifica un alias non esistente;
- l'alias risiede in un file system a sola lettura;

Vediamo, adesso, un esempio d'uso congiunto delle chiamate link() e unlink() in un programma che esegue la rinominazione di un file (Figura 2.5). Il programma prima crea un alias di un file e poi elimina l'alias originale.

Il risultato finale non è altro che una rinominazione del fine in oggetto (il suo nome viene cambiato un quello dell'alias creato). La riga di comando per l'esecuzione del programma sarà: rename vecchio_nome nuovo_nome.

2.5 Chiamata dup()

In UNIX è possibile duplicare un descrittore di file esistente. Il nuovo descrittore file ha le seguenti caratteristiche:

- afferisce allo stesso file associato al descrittore originario;
- eredita lo stesso puntatore di lettura/scrittura del canale originario;
- eredita la stessa modalità di accesso al file del canale associato al descrittore originario;
- il descrittore di file restituito dalla chiamata `dup()` è sempre il valore intero corrispondete al più piccolo indice della tavola dei file aperti associato ad una entry libera.

La duplicazione avviene tramite la seguente chiamata di sistema:

```
#define FNAME "info.txt"
#define STDIN 0

int main() {
    int fd;
    fd = open(FNAME,O_RDONLY);  /* Apro il file in lettura        */
    close(STDIN);               /* Chiudo lo standard input       */
    dup(fd);                    /* Duplico il descrittore di file */
    execlp("more","more",0);    /* Eseguo 'more'                  */
}
```

Figure 2.6: Redirezione del canale di input mediante le chiamata close() e dup()

int **dup**(int file_descriptor)	
descrizione	invoca la duplicazione di un descrittore di file
argomenti	file_descriptor: descrittore di file che si vuole duplicare
restituzione	un intero positivo corrispondente al nuovo dexcrittore di file oppure -1 in caso di fallimento

Notare che la duplicazione del file_descriptor equivale alla riapertura del file associato al canale di input/output identificato da file_descriptor stesso. La chiamata fallisce se si verifica una delle seguenti condizioni:

- file_descriptor non identifica un canale di I/O aperto;
- è già stato raggiunto il numero massimo di file che possono essere aperti contemporaneamente dal processo.

La chiamata `dup()` può essere usata per effettuare la redirezione dell'input di un programma. Vediamo un esempio. Il file info.txt contiene delle informazioni che vogliamo visualizzare, una schermata alla volta. Questo può essere effettuato sia tramite la shell, con la seguente linea di comando: `more < info.txt`, oppure tramite il programma mostrato in Figura 2.6.

Chapter 3

Gestione di processi

Come già accenato nei capitoli precedenti, un processo è qualche cosa in più del codice di un programma. Esso comprende un'attività correntemente svolta, rappresentata da una serie di parametri quali, per esempio: il valore del contatore di programma, il contenuto dei registri del processore ed il contenuto dello stack. Quest'ultimo contiene dati temporanei associati all'esecuzione del processo quali i parametri di procedura, le variabili locali di procedura e gli indirizzi di ritorno dalla chiamata a procedura. Notare che, sebbene più processi possano essere associati allo stesso programma, essi sono da considerare entità totalmente distinte.

Il kernel gestisce una tabella contenete informazioni su tutti i processi attivi nel sistema ed ogni processo attivo è univocamente identificato tramite un identificatore di processo (PID), corrispondente ad un valore intero positivo che viene assegnato dal kernel stesso all'atto della creazione del processo. La suddetta tabella contiene, per ogni processo una struttura nota con il nome Process Control Block (PCB) contenente le informazioni relative al processo, quali per esempio: il PID del processo e lo stato del processo (pronto per l'esecuzione, in esecuzione, in attesa etc.).

In questo capitolo esamineremo le chiamate di sistema di base per la gestione e manipolazione di processi in ambiente UNIX.

3.1 Chiamate `fork()` e `wait()`

In ambiente UNIX, *l'unica possibilità che un processo in esecuzione ha per generare un altro processo* è quella di eseguire la chiamata di sistema `fork()` descritta dalla seguente tabella (notare che il processo che effettua la chiamata viene denominato "processo padre" mentre il nuovo processo originato viene denominato "processo figlio"):

int **fork**()	
descrizione	invoca la duplicazione del processo chiamante (creazione di un figlio)
restituzione	nel chiamante: identificatore di processo del figlio generato, -1 in caso di fallimento; 0 nel figlio

L'effetto della chiamata `fork()` è quello di originare un processo che è l'esatta copia (in termini di istruzioni, dati memorizzati nello stack etc.) del processo che esegue la chiamata. Come è facile intuire dalla spiegazione in forma tabellare, la chiamata fork() ritorna sia nel processo padre che nel processo figlio (notare che invece tutte le istruzioni precedenti alla chiamata `fork()` di fatto nel figlio non vengono eseguite). Nel padre la `fork()`, qualora vada a buon fine, restituisce il PID del nuovo processo originato (ovvero il figlio). Viene riportato in Figura 3.1 un semplice esempio di utilizzo della chiamata `fork()`: il processo padre genera un processo figlio; entrambi i processo testano in valore di ritorno della chiamata `fork()` per rendersi conto di chi di loro è il padre e chi il figlio; il padre manda in output sullo standard output la stringa "Sono il padre", il figlio manda sullo standard output la stringa "Sono il figlio". Notare che il processo figlio, essendo una copia perfetta del processo padre eredita da lui tutti i descrittori di file, inclusi quelli afferenti allo standar input ed allo standard output (questo implica che gli standard output di padre e figlio coincidono).

```
#include <stdio.h>

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

    pid = fork();

    if (pid == 0) printf("Sono il processo figlio\n");
    else          printf("Sono il processo padre\n")
}
```

Figure 3.1: Esempio di uso della chiamata fork().

```
#include <stdio.h>

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

    pid = fork();

    if (pid == 0) printf("Sono il processo figlio\n");
    else {
        wait(&status);
        printf("Sono il processo padre\n")
    }
}
```

Figure 3.2: Esempio di uso combinato delle chiamate fork() e wait().

Con riferimento all'esempio in Figura 3.1, è possibile che una qualsiasi delle due stringhe venga trasferita sullo standard output per prima. *Non è quindi possibile affermare con certezza che la stringa assiciata al padre venga visualizzata proma di quella associata al figlio e viceversa.* Questo fenomeno è dovuto al fatto che non sono specificati meccanismi per la sincronizzazione delle azioni dei due processi all'interno del codice mostrato e l'ordine relativo di esecuzione dei processi dipende dalle politiche di scheduling della CPU proprie del sistema operativo. In UNIX esiste un semplicissimo meccanismo di sincronizzazione che permette ad un processo padre di attendere la terminazione di un suo processo figlio. Questo meccanismo è realizzato tramite la chiamata di sistema `wait()` descritta dalla seguente tabella:

int **wait**(int *status)	
descrizione	invoca l'attesa della terminazione di almeno un figlio
argomenti	*status: puntatore ad un intero dove viene registrato lo stato di terminazione del figlio che termina
restituzione	l'identificatore di processo del figlio che termina; -1 in caso non esistano figli

L'effetto della chiamata `wait()` è quello di sospendere l'esecuzione del processo padre fino alla terminazione di almeno un processo figlio. Tramite questa chiamata di sistema è possibile modificare il codice mostrato in Figura 3.1 in modo tale da essere sicuri che la stringa associata al padre vada sullo standard output dopo di quella associata al figlio. Per realizzare questo tipo di sincronizzazione semplice basta inserire la chiamata `wait()` nel codice associato al padre prima che esso immetta sullo standard output la sua stringa. In tal modo, questa stringa andrà sullo standard output solo dopo che il processo figlio ha terminato la sua esecuzione. Il codice risultante è mostrato in Figura 3.2.

3.2 Chiamate execX()

Come mostrato dalla sezione precedente, tramite la chiamata **fork()** è possibile originare nuovo processi che sono copie esatte del processo che esegue la chiamata **fork()**. In UNIX è anche possibile che un processo mandi in esecuzione un comando (cioè un qualsiasi eseguibile memorizzato all'interno del file system). Per far questo, esiste una famiglia di chamate di sistema dette **execX()** (dove X sta per: l, lp, v, vp). Esse sono descritte dalle seguenti tabelle:

int **execl**(char *file_name, [*arg0, ... , *argN,] 0)	
descrizione	invoca l'esecuzione di un comando
argomenti	*file_name: puntatore alla stringa che specifica il path-name del comando da eseguire [*arg0, ... , *argN,]: lista di argomenti che definiscono la linea di comando del comando da eseguire
restituzione	-1 in caso di fallimento

int **execlp**(char *file_name, [*arg0, ... , *argN,] 0)	
descrizione	invoca l'esecuzione di un comando
argomenti	*file_name: puntatore alla string che specifica il nome del comando da eseguire [*arg0, ... , *argN,]: lista di argomenti che definiscono la linea di comando del comando da eseguire
restituzione	-1 in caso di fallimento

int **execv**(char *file_name, *argv[])	
descrizione	invoca l'esecuzione di un comando
argomenti	*file_name: puntatore alla stringa che specifica il path-name del comando da eseguire *argv[]: lista di argomenti che definiscono la linea di comando del comando da eseguire
restituzione	-1 in caso di fallimento

int **execvp**(char *file_name, *argv[])	
descrizione	invoca l'esecuzione di un comando
argomenti	*file_name: puntatore alla stringa che specifica il nome del comando da eseguire *argv[]: lista di argomenti che definiscono la linea di comando del comando da eseguire
restituzione	-1 in caso di fallimento

Qui di seguito analizzeremo in dettaglio la chiamata **execlp()**, le considerazioni che verranno effettuate valgono allo stesso modo per tutte le altre chamate di questa famiglia. Prima di entrare nella discussione, è da notare che la differenza di base tra le prime due chiamate (**execl()** ed **execlp()**) e le ultime due (**execv()** ed **execvp()**) risiede nel fatto che nelle prime due il numero di parametri passati alla chiamata e mappati negli argomenti del comando invocato è determinata a tempo di compilazione, mentre nelle seconde due, esso può variare a tempo di esecuzione.

Tornando alla chiamata **execlp()**, abbiamo che se un processo invoca tale chamata, l'effetto della sua esecuzione (in caso di non fallimento) è quello di sostituire il codice eseguibile del processo chiamante con il codice eseguibile del comando che si vuole lanciare tramite la chiamata. Notare che la sostituzione di codice non implica la creazione di un nuovo processo, al contrario, il comando che va in esecuzione resta identificato dal sistema con lo stesso PID del processo che invoca la execlp(). Quindi l'effetto globale è quello di sostituire il codice eseguibile di un processo senza alterare l'identità del processo stesso. Notare che la chiamate **execlp()** ha l'effetto di chiudere tutti i file aperti del processo chiamante (eccetto i tre standard).

Come esempio di applicazione, riportiamo in Figura 3.3 un codice in cui un processo richiede l'esecuzione del comando "ls" tramite la chiamata **execlp()**. E' da notare che se la chiamata fallisce, il codice del processo chiamate non viene sostituito e quindi esso manda sullo standard output un messaggio d'errore. Se la chiamata non fallisce, il codice del processo chiamante viene sostituito con il codice del comando "ls", quindi l'istruzione **printf("...")** per mandare in output il messaggio di errore non verrà mai eseguita.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    execlp("ls","ls",0);
    printf("La chiamata execlp() ha fallito\n")
}
```

Figure 3.3: Esempio di uso della chiamata execlp().

3.3 Un esempio di applicazione

In questa sezione verrà descritto un esempio di applicazione delle chiamate di sistema presentate in questo capitolo. In particolare, verrà implementata una semplicissima shell di comandi che accetta linee di comando semplici costituite dal solo nome del comando che si viole eseguire (non c'è possibilità di richiedere opzioni oppure di passare argomenti sulla linea di comando). Per chiarezza ricordiamo che una shell è un programma che interagisce con un utente attendendo che egli digiti appunto una riga di comando che la shell deve poi mandare in esecuzione. Al termine della esecuzione, la shell si rimette in attesa della successiva linea di comando specificata dall'utente.

L'architettura del software è strutturata al seguente modo: esiste una shell padre che attende sullo standard input il comando digitato dall'utente; una volta letto il comando, la shell padre genera una shell figlio (tramite una chiamata `fork()`) e si mette in attesa che essa termini (tramite una chamata `wait()`); la shell figlio esegue una `execlp()` invocando l'esecuzione del comando specificato dall'utente. Il codice viene riportato in Figura 3.4. Se il comando specificato dall'utente è "exit", la shell padre termina. E' da notare che la chiamata `fork()` è necessaria poichè se la shell padre eseguisse lei stessa la chamata `execlp()` il suo codice verrebbe sostuituito con il codice del comando specificato dall'utente ed al termine dell'esecusione di questo la shell padre non esisterebbe più e non potrebbe quindi accettare nuovi comandi dall'utente.

```
#include <stdio.h>
void main() {
    char comando[256];
    int  pid, status;

    while(1) {
        printf("Digitare un comando ('quit' per terminare): ");
        scanf("%s",comando);
        if ( strcmp(comando,"quit") == 0 ) break;

        pid = fork();

        if ( pid == -1 ) {
            printf("Errore nella fork.\n");
            exit(1);
        }
        if ( pid == 0 ) execlp(comando,comando,0);
        else wait(&status);
    }
}
```

Figure 3.4: Implementazione di una semplice shell di comandi

Chapter 4

Costrutti per la comunicazione: code di messaggi

Uno dei costrutti che UNIX mette a disposizione per la comunicazione fra processi è la coda di messaggi. Una coda di messaggi è una struttura che mantiene memorizzato un messaggo inviati da un processo fino a quando un altro processo (o il mittente stesso) non decida di estrarre il messaggio dalla coda. Ogni coda ha un nome unico nell'ambito del sistema, esattamente come un qualsiasi file, però, a differenza dei file, il nome di una coda non e espresso come una stringa (cioè un pathname) bensì come un numero intero positivo denominato "chiave". La chiave è ciò che i processi utilizzano per accedere alla coda di messaggi (cioè depositare od estrarre messaggi). E' da notare che un processo che vuole utilizzare una coda con una data chiave deve preventivamente aprire tale coda (così come l'accesso ad un file vuole l'apertura preventiva del file). Inoltre, esattamente come i file, le code di messaggi sono strutture permanenti per cui l'eliminazione di una coda richiede l'invocazione di una chiamata di sistema apposita. La proprietà di essere permanente è valida anche per tutti i messaggi depositati in una coda e non ancora estratti. Un messaggio in UNIX è un'unità di informazione di dimensione variabile, non caratterizzato da alcun formato predefinito.

In questo capitolo analizzeremo le chiamate di sistema per creare, aprire e distruggere una coda di messaggi e per depositare od estrarre messaggi su di essa.

4.1 Chiamate `msgget()` e `msgctl()`

La chiamata di sistema per creare una coda di messaggi è la `msgget()`, descritta in forma tabellare come segue:

int **msgget**(key_t key, int flag)	
descrizione	invoca la creazione una Message Queue
argomenti	key: chiave per identificare la coda di messaggi in maniera univoca nel sistema flag: intero per la specifica della modalità di creazione
restituzione	identificatore numerico per l'accesso alla coda in caso di successo (descrittore di coda); -1 in caso di fallimento

Come descritto dalla tabella, la chiamata `msgget()` non solo permette di specificare la chiave da associare alla coda stessa come identificatore unico nell'ambito del sistema, ma anche i permessi di accesso alla coda stessa (questo evidenzia molte analogie con la chiamata `creat()` per la creazione di un nuovo file, nella quale il secondo parametro specifica appunto i permessi di accesso al file stesso). In particolare, il parametro flag può essere specificato come una combinazione, ottenuta tramite l'operatore "—" dei valori IPC_CREAT, IPC_EXCL (definiti negli header file <sys/ipc.h> e <sys/msg.h>) e della modalità per l'accesso alla coda specificata, come nel caso dei file, tramite una codifica ottale. Il valore IPC_CREAT specifica appunto che si vuole creare una coda di messaggi con il nome designato dal parametro key (è da notare che questo parametro è di tipo key_t che non è altro che una ridefinizione del tipo long). Nel caso la coda con quella data chiave

non esista, essa viene creata ed automaticamente aperta per il processo chiamante al quale viene restituito il descrittore di coda. Nel caso la coda con quella data chiave già esista, l'unico effetto della chiamata è quello di restituire al processo un descrittore della coda gia esistente per l'accesso ad essa. In Figura 4.1 viene riportato un semplicissimo esempio di utilizzo della chiamata `msgget()` con specifica del parametro IPC_CREAT e dei parmessi di accesso alla coda. In questo esempio, qualsiasi processo ha accesso alla coda sia per quel che riguarda la capacità di depositare che di estrarre messaggi. E' da notare che al termine dell'esecuzione del codice, se non si verifica fallimento nella chiamata `msgget()`, allora la coda viene creata e resta permanente (cioè non viene imossa alla terminazione dell'esecuzione).

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
void main() {

    int ds_coda;
    key_t chiave = 30;

    ds_coda = msgget(chiave, IPC_CREAT|0666);
    if ( ds_coda == -1 )  printf("\n errore nella chiamata msgget \n");
}
```

Figure 4.1: Un esempio di utilizzo della chiamata msgget()

Il valore IPC_EXCL si può usare solo in combinazione con il valore IPC_CREAT e fa si che il sistema operativo ritorni un errore nella chiamata `msgget()` qualora la coda di messaggi con quella data chiave già esista (fondamentalmente, qualora specificato nella definizione di flag, il valore IPC_EXCL sta ad indicare una creazione di tipo esclusivo da parte del processo).

Una volta creata, una coda rimarrà quindi presente nel sistema fino a quando non venga richiesta esplicictamente la sua rimozione. La chiamata di sistema per rimuovere una coda o anche per modificare i permessi di accesso o per acquisire statistiche su di essa è la `msgctl()` descritta come segue:

int **msgctl**(int ds_coda, int cmd, struct msquid_ds *buff)	
descrizione	invoca un comando su una Message Queue
argomenti	ds_coda: descrittore della coda sulla quale si invoca il comando cmd: specifica il comando invocato *buff: puntatore ad una struttura dati nella quale memorizzare informazioni relative allo stato della Message Queue
restituzione	identificatore numerico per l'accesso alla coda in caso di successo; -1 in caso di fallimento

Come evidenziato dalla tabella, esiste la possibilità di specificare comandi differenti tramite l'assegnazione del valore del parametro cmd. Ad esso si può assegnare uno dei valori tra IPC_RMID, IPC_STAT e IPC_SET. Il valore IPC_RMID specifica al sistema operativo di rimuovere la coda associata al descrittore ds_coda. Il valore IPC_STAT specifica al sistema che il processo vuole statistiche afferenti alla coda associata al descrittore ds_coda (le statistiche verranno restituite dal sistema nel buffer strutturato di tipo struct msqid_ds puntato dal terzo parametro della chiamata `msgctl()`). Il valore IPC_SET specifica al sistema che il processo vuole cambiare i permessi di accesso alla coda associata al descrittore ds_coda (la specifica dei nuovi permessi è contenuta nel buffer strutturato di tipo struct msqid_ds puntato dal terzo parametro della chiamata `msgctl()`). Riportiamo in Figura 4.2 un esempio di codice in cui viene chiesta la creazione esclusiva di una coda con chiave 40 e la sua rimozione prima della terminazione dell'esecuzione invocata tramite la chiamata `msgctl()`.

E' da notare che nel caso specifico dell'esempio in Figura 4.2, il terzo parametro della chiamata `msgctl()` è il puntatore nullo perchè il comando di rimozione IPC_RMID non richiede un buffer allocato ove debbano

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

void main() {
    int ds_coda;
    key_t chiave = 40;

    ds_coda = msgget(chiave, IPC_CREAT|IPC_EXCL|0666);
    if ( ds_coda == -1 ) {
        printf("Errore nella chiamata msgget \n");
        exit(1);
    }
    sleep(20);
    msgctl(ds_coda, IPC_RMID, NULL);
}
```

Figure 4.2: Un esempio di utilizzo della chiamata msgctl()

essere restituite statistiche oppure dove sia memorizzata la specifica di nuovi permessi per l'accesso alla coda.

4.2 Chiamate `msgsnd()` e `msgrcv()`

Nella sezione precedente abbiamo visto come sia possibile creare e distruggere code di messaggi in UNIX. In questa sezione verranno descritte le chiamate di sistema per depositare od estrarre messaggi.

La chiamata di sistema per depositare un messaggio in una coda di messaggi è la **`msgsnd()`** descritta in forma tabellare come segue:

int **msgsnd**(int ds_coda, const void *ptr, size_t nbyte, int flag)	
descrizione	invoca la spedizione di un messaggio su una Message Queue
argomenti	ds_coda: descrittore della coda di messaggi su cui si vuole inviare il messaggio *ptr: puntatore al buffer contenente il messaggio da inviare nbyte: numero di byte da prelevare dal buffer e spedire flag: specifica se la spedizione è bloccante o meno
restituzione	-1 in caso di fallimento

Il primo parametro associato alla specifica di **`msgsnd()`** è il descrittore della coda dimessaggi sulla quale si vuole depositare il messaggio. *ptr è il puntatore al buffer contenente i byte associati al messaggio da depositare. nbyte indica quanti dei byte memorizzati nel buffer puntato da *ptr effettivamente costituiscono il corpo del messaggio ed andranno quindi trasferiti sulla coda. flag specifica la modalità di spedizione del messaggio; il valore di default è lo zero, in tal caso si otterà per la spedizione il seguente comportamento: se la coda di messaggi è satura (cioè è stato raggiunto il numero massimo di messaggi che essa può memorizzare) allora il processo che richiede la spedizione verrà sospeso fino a che almeno un messaggio verrà rimosso dalla coda (in tal caso verrà rilasciato spazio per poter depositare il messaggio in oggetto). Esiste comunque la possibilità di evitare che il processo che invoca la spedizione si sospenda anche nel caso in cui la coda sia piena; per ottenere questo comportamento, andrà specificato per il parametro flag il valore IPC_NOWAIT. Se la coda è effettivamente piena e per flag viene specificato tale valore, allora si otterrà che il processo che invoca la spedizione non verrà sospeso, ma la spedizione stessa verrà abortita (con il conseguente ritorno del valore -1 da parte della chiamata a **`msgsnd()`**). Notare che la chiamata msgsnd() ritorna il valore -1 anche in ogni altro caso di fallimento.

E' importantissomo notare che esiste un vincolo sulla struttura del buffer puntato da *ptr e contenente i byte del messaggio da trasferire. Esso deve contenere nei primi quattro byte un valore di tipo long positivo o

nullo che specifica il tipo di messaggio contenuto nel buffer, quindi ad ogni intero positivo o nullo corrisponde un ben determinato tipo di messaggio. Il resto dei byte nel buffer puntato da *ptr contengono dati facenti effetivamente parte del testo del messaggio. Notare che nbyte specifica esclusivamente la taglia della parte testo (quindi non include i quattro byte della specifica del tipo di messaggio).

In Figura 4.3 viene riportato un esempio di utilizzo della chiamata `msgsnd()` per depositare un messaggio di tipo 1 contenente la stringa "saluti" in una coda di messaggi. Prima di depositare il messaggio la coda viene creata. Prima della terminazione dell'esecuzione, la coda vene rimossa, e così anche il messaggio precedenetemente depositato.

```
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

typedef struct {
    long mtype;
    char mtext[TAGLIA];
} msg;

void main() {
    msg   messaggio;
    int   ds_coda, ris;
    key_t chiave = 40;

    ds_coda = msgget(chiave, IPC_CREAT|IPC_EXCL|0666);

    if ( ds_coda == -1 ) {
        printf("Errore nella chiamata msgget\n");
        exit(1);
    }

    messaggio.mtype = 1;
    strcopy(messaggio.mtext,"saluti");

    ris = msgsnd(ds_coda, &messaggio, 7, IPC_NOWAIT);

    if ( ris == -1 ) {
        printf("Errore nella chiamata msgsnd\n");
        exit(1);
    }

    ris = msgctl(ds_coda, IPC_RMID, NULL)
    if ( ris == -1 ) {
        printf("Errore nella chiamata msgctl\n");
        exit(1);
    }
}
```

Figure 4.3: Un esempio di utilizzo della chiamata msgsnd()

L'estrazione di un messaggio da una coda di messaggi ha luogo tramite a chiamata `msgrcv()` descritta dalla seguente tabella:

int **msgrcv**(int ds_coda, void *ptr, size_t nbyte, long type, int flag)	
descrizione	invoca la ricezione di un messaggio da una coda di messaggi
argomenti	ds_coda: identificatore numerico della coda di messaggi da cui si vuole ricevere il messaggio *ptr: puntatore al buffer dove registrare il messaggio da ricevere nbyte: massimo numero di byte del messaggio da ricevere type: tipo di messaggio da ricevere flag: specifica se la ricezione è bloccante o meno
restituzione	numero di byte ricevuti; -1 in caso di fallimento

Per quanto riguarda i parametri, il significato di ds_coda è immediato. *ptr è il puntatore al buffer ove memorizzare il messaggio estratto dalla coda. nbyte è il numero massimo di byte del messaggio estratto dalla coda che devono essere memorizzati nel buffer puntato da *ptr (notare che anche in questo caso i quattro byte che indicano il tipo di messaggio non vengono inclusi nel valore assegnato a nbyte). type è il tipo di messaggio che si intende ricevere; se il valore di type è settato zero, allora non vi è specifica di alcun tipo particolare di messaggio da voler ricevere, cioè si desidera ricevere il più vecchio messaggio attualnmente presente nella coda, indipendentemente dal valore del suo tipo. Se il valore di type è un intero positivo x allora si desidera estrarre dalla coda solo ed esclusivamente un messaggio di tipo x. Se invece il valore di type è un intero negativo $-x$, allora si desidera estrarre dalla coda un messaggio di tipo x se presente, oppure un messaggio di tipo $x-1$ se nessun messaggio di tipo x è presente e così via fino al tipo 1. Il valore di flag specifica la modalità di ricezione. Il default zero indica una ricezione di tipo bloccante che sospende il process fino a che almeno un messaggio conforme alla specifica del valore di type sia effettivamnete presente nella coda. La ricezione si può rendere non bloccante specificando per flag il valore IPC_NOWAIT. In tal caso, la chiamata `msgrcv()` ritorna immediatamente. E' da notare che se per flag è specificao il valore IPC_NOWAIT e nessun messaggio conforme a type è presente nella coda, allora la chiamata `msgrcv()` ritorna il valore -1, così come in ogni altro caso di fallimento.

In Figura 4.4 viene riportato un esempio di utilizzo congiunto delle chiamate `msgsnd()` e `msgrcv()`. In questo esempio, viene depositato un messaggio contenente la stringa "saluti" nella coda di messaggi con chiave 40; subito dopo viene riestratto ed il suo contenuto (cioè la parte testo) viene mandata sullo standard output.

4.3 Un esempio di applicazione

In questa sezione verrà descritto un esempio di applicazione delle chiamate di sistema presentate in questo capitolo. In particolare, verrà implementato un trasferimento di stringhe tra due processi realizzato tramite l'uso di una coda di messaggi. L'architettura del software, descritta in Figura 4.5, è strutturata al seguente modo: esiste un processo padre che genera la coda di messaggi e da poi luogo a due figli; il primo figlio prende stringhe in ingresso sullo standard input e le deposita nella coda di messaggi tramite, appunto la spedizione di un messaggio per ciascuna stringa; l'altro figlio estrae i messaggi e rivisualizza le corrispondenti stringhe sullo standard output. Il trasferimento termina quando viene digitata la stringa "quit". Al termine del trasferimento il padre rimuove la coda di messaggi. E' da notare che il padre attende la terminazione di entrambi i figli prima di rimuovere la coda e terminare.

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define TAGLIA 1024

typedef struct {
    long mtype;
    char mtext[TAGLIA];
} msg;

void main() {
    msg   messaggio;
    int   ds_coda, ris;
    key_t chiave = 40;

    ds_coda = msgget(chiave, IPC_CREAT|IPC_EXCL|0666);

    if ( ds_coda == -1 ) {
        puts("Errore nella chiamata msgget");
        exit(1);
    }

    messaggio.mtype = 1;
    strcpy(messaggio.mtext,"saluti");

    ris = msgsnd(ds_coda, &messaggio, strlen(messaggio.mtext)+1, IPC_NOWAIT);

    if ( ris == -1 ) puts("Errore nella chiamata msgsnd");

    ris = msgrcv(ds_coda, &messaggio, TAGLIA, 1, 0);

    if ( ris == -1) puts("Errore nella chiamata msgrcv");
    else            printf("Messaggio: %s\n", messaggio.mtext);

    ris = msgctl(ds_coda, IPC_RMID, NULL);

    if ( ris == -1 ) {
        puts("Errore nella chiamata msgctl");
        exit(1);
    }
}
```

Figure 4.4: Un esempio di utilizzo della chiamata msgsnd()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

#define TAGLIA 128

typedef struct{
    long mtype;
    char mtext[TAGLIA];
} msg;

#define Errore_(x) { puts(x); exit(1); }

int ris;

/* lettura delle stringhe e spedizione sulla coda */
void produttore(int ds_coda) {
    msg messaggio;
    puts("digitare le stringhe da trasferire (quit per terminare):");
    do {
        scanf("%s",messaggio.mtext);
        messaggio.mtype = 1;

        ris = msgsnd(d_coda, &messaggio, TAGLIA, IPC_NOWAIT);
        if ( ris == -1 ) Errore_("Errore nella chiamata msgsnd");

        printf("Inviato messaggio: %s\n", messaggio.mtext);
    } while( (strcmp(messaggio.mtext,"quit") != 0));
    exit(0);
}

/* ricezione delle stringhe e visualizzazione sullo standard output */
void consumatore(int ds_coda) {
    msg messaggio;
    do {
        ris = msgrcv(d_coda, &messaggio, TAGLIA, 1, 0);
        if ( ris == -1 ) Errore_("Errore nella chiamata msgrcv");

        printf("ricevuto messaggio: %s\n", messaggio.mtext);
    } while( (strcmp(messaggio.mtext,"quit") != 0));
    exit(0);
}

int main(int argc, char *argv[]) {
    int  des_coda, status;
    long chiave = 40;

    des_coda = msgget(chiave, IPC_CREAT|IPC_EXCL|0666);
    if ( des_coda == -1 ) Errore_("Errore nella chiamata msgget");

    if ( fork()!=0 ) {
        if ( fork()!=0 ) {
            wait(&status);
            wait(&status);
        }
        else produttore(des_coda);
    }
    else consumatore(des_coda);

    ris = msgctl(des_coda, IPC_RMID, NULL);
    if ( ris == -1 ) Errore_("Errore nella chiamata msgctl");
}
```

Figure 4.5: Trasferimento di stringhe tra due processi tramite la coda di messaggi

Chapter 5

Costrutti per la comunicazione: memoria condivisa

Un altro costrutto che UNIX mette a disposizione per la comunicazione fra processi è la memoria condivisa. Una memoria condivisa è una porzione di memoria accessibile da più processi. Su di essa un processo può scrivere dei dati che possono essere letti da un qualsiasi processo abbia i permessi per accedere alla memoria condivisa. Ogni memoria condivisa ha un nome unico nell'ambito del sistema che, analogamente ad una coda di messaggi, è espresso come un numero intero positivo denominato "chiave". La chiave è ciò che i processi utilizzano per accedere alla memoria condivisa (cioè scrivere o leggere dati). E' da notare che un processo che vuole utilizzare una memoria condivisa con una data chiave deve preventivamente aprirla (così come l'accesso ad una coda di messaggi vuole l'apertura preventiva della coda di messaggi) ed "attaccarla" al proprio spazio di indirizzamento . Qualora un processo non desideri più utilizzare una memoria condivisa, esso può "staccarla" dal suo spazio di indirizzamento. Notare che questo non implica che la memoria condivisa venga eliminata dal sistema poichè, esattamnente come le code di messaggi ed i file, le memorie condivise sono strutture permanenti. L'eliminazione di una memoria condivisa richiede l'invocazione di una chiamata di sistema apposita differente dalla chiamata di sistema per staccarla dallo spazio di indirizzamento. La proprietà di essere permanente è valida anche per tutti i dati memorizzati nella memoria condivisa.

In questo capitolo analizzeremo le chiamate di sistema per creare, aprire e distruggere una memoria condivisa e per attaccarla e staccarla dallo spazio di indirizzamento. Letture e scritture su di una memoria condivisa non hanno necessità di particolari chiamate di sistema. Una volta attaccata allo spazio di indirizzamento, la memoria condivisa può essere scritta e/o letta come una qualsiasi variabile facente parte dello spazio di indirizzamento del processo.

5.1 Chiamate `shmget()` e `shmctl()`

La chiamata di sistema per creare una memoria condivisa è la shmget(), descritta in forma tabellare come segue:

int **shmget**(key_t key, int size, int flag)	
descrizione	invoca la creazione una Memoria Condivisa
argomenti	key: chiave per identificare la Memoria Condivisa in maniera univoca nel sistema size: taglia della Memoria Condivisa flag: intero per la specifica della modalità di creazione
restituzione	identificatore numerico intero non negativo (descrittore di coda) per l'accesso alla coda in caso di successo; -1 in caso di fallimento

Come descritto dalla tabella, la chiamata `shmget()` non solo permette di specificare la chiave da associare alla memoria condivisa come identificatore unico nell'ambito del sistema, ma anche i permessi di accesso alla

coda stessa (questo evidenzia molte similitudini con la chiamata `msgget()` per creare una coda di messaggi descritta nel Capitolo 4). Il parametro size indica la taglia, espressa in termini di quantità di byte, della memoria condivisa che si vuole creare. Il parametro flag può essere specificato come una combinazione, ottenuta tramite l'operatore "|" dei valori IPC_CREAT, IPC_EXCL (definiti negli header file <sys/ipc.h> e <sys/msg.h>) e della modalità per l'accesso alla memoria condivisa specificata, come nel caso dei file e delle code di messaggi, tramite una codifica ottale. Il valore IPC_CREAT specifica che si vuole creare una memoria condivisa con il nome designato dal parametro key (è da notare che questo parametro è di tipo key_t che come già spiegato nel Capitolo 4 una ridefinizione del tipo long). Nel caso la memoria condivisa con quella data chiave non esista, essa viene creata ed al processo chiamante viene restituito il descrittore di coda. Nel caso la memoria condivisa con quella data chiave già esista, l'unico effetto della chiamata è quello di restituire al processo un descrittore della memoria già esistente (in tal caso, il valore del parametro taglia ed in permessi specificati da flag vengono ignorati). In Figura 5.1 viene riportato un semplicissimo esempio di utilizzo della chiamata `shmget()` con specifica del parametro IPC_CREAT, della taglia (1024 byte) e dei parmessi di accesso alla memoria condivisa. In questo esempio i permessi specificano che qualsiasi processo ha accesso alla memoria condivisa. E' da notare che al termine dell'esecuzione del codice, se non si verifica fallimento nella chiamata `shmget()`, allora la coda viene creata e resta permanente (cioè non viene rimossa alla terminazione dell'esecuzione).

```
#include <stdio.h>
#include <sys/ipc.h>

#define Errore_(x) { puts(x); exit(1); }

void main() {
    int   ds_shm, ris;
    key_t chiave = 40;

    ds_shm = shmget(chiave, 1024, IPC_CREAT|IPC_EXCL|0666);
    if ( ds_shm == -1 ) Errore_("Errore nella chiamata shmget");
}
```

Figure 5.1: Un esempio di utilizzo della chiamata shmget()

Una volta creata, una memoria condivisa rimarrà quindi presente nel sistema fino a quando non venga richiesta esplicitamente la sua rimozione. La chiamata di sistema per rimuovere una memoria condivisa o anche per modificarene ipermessi di accesso o per acquisire statistiche su di essa è la `shmctl()` descritta come segue:

int **shmctl**(int ds_shm, int cmd, struct shmid_ds *buff)	
descrizione	invoca un comando su una memoria condivisa
argomenti	ds_shm: descrittore della memoria condivisa su cui eseguire il comando cmd: specifica il comando da eseguire *buff: puntatore ad una struttura dati nella quale memorizzare informazioni relative al comando da eseguire
restituzione	-1 in caso di fallimento

Come per il caso delle code di messaggi, esiste la possibilità di specificare comandi differenti tramite l'assegnazione del valore del parametro cmd. Ad esso si può assegnare uno dei valori tra IPC_RMID, IPC_STAT e IPC_SET. Il valore IPC_RMID specifica al sistema operativo di rimuovere la memoria condivisa associata al descrittore ds_shm. Il valore IPC_STAT specifica al sistema che il processo vuole statistiche afferenti alla memoria condivisa associata al descrittore ds_shm (le statistiche verranno restituite dal sistema nel buffer strutturato di tipo struct shmid_ds puntato dal terzo parametro della chiamata `shmctl()`). Il valore IPC_SET specifica al sistema che il processo vuole cambiare i permessi di accesso alla memoria condivisa associata al descrittore ds_shm (la specifica dei nuovi permessi è contenuta nel buffer strutturato di tipo struct

shmid_ds puntato dal terzo parametro della chiamata `shmctl()`). Riportiamo in Figura 5.2 un esempio di codice in cui viene chiesta la creazione esclusiva di una coda con chiave 40 e la sua rimozione prima della terminazione dell'esecuzione invocata tramite la chiamata `shmctl()`.

```
#include <stdio.h>
#include <sys/ipc.h>

#define Errore_(x) { puts(x); exit(1); }

void main() {
    int ds_shm, ris;
    key_t chiave = 40;

    ds_shm = shmget(chiave, 1024, IPC_CREAT|IPC_EXCL|0666);
    if ( ds_shm == -1 ) Errore_("Errore nella chiamata shmget");

    sleep(20);
    ris = shmctl(ds_shm, IPC_RMID, NULL);
    if ( ris == -1 ) Errore_("Errore nella chiamata shmctl");
}
```

Figure 5.2: Un esempio di utilizzo della chiamata shmctl()

E' da notare che nel caso specifico dell'esempio in Figura 5.2, il terzo parametro della chiamata `shmctl()` è il puntatore nullo perchè il comando di rimozione IPC_RMID non richiede un buffer allocato ove debbano essere restituite statistiche oppure dove sia memorizzata la specifica di nuovi permessi per l'accesso alla memoria condivisa.

5.2 Chiamate `shmat()` e `shmdet()`

Nella sezione precedente abbiamo visto come sia possibile creare e distruggere una memoria condivisa. In questa sezione verranno descritte le chiamate di sistema per "attaccare" e "staccare" la memoria condivisa dalla spazio di indirizzamento.

La chiamata di sistema per attaccare una memoria condivisa allo spazio di indirizzamento di un processo è la `shmat()` descritta in forma tabellare come segue:

void ***shmat**(int ds_shm, void *addr, int flag)	
descrizione	invoca il collegamento di una memoria condivisa allo spazio di indirizzamento del processo
argomenti	ds_shm: identificatore numerico della memoria condivisa da collegare *addr: indirizzo di memoria dove collegare la memoria condivisa flag: specifica le modalità di accesso alla memoria condivisa
restituzione	indirizzo effettivo dove la memoria condivisa è stata collegata in caso di successo; -1 in caso di fallimento

Il primo parametro associato alla specifica di `shmat()` è il descrittore della memoria condivisa che si vuole collegare allo spazio di indirizzamento del processo. *addr è l'indirizzo ove il processo vorrebbe collegare tale memoria (se si specifica 0 per *addr allora si dà libera scelta al sistema operativo per l'assegnazione dell'indirizzo ove collegare la memoria condivisa). flag specifica la modalità di accesso alla memoria condivisa. I possibili valori, specificati nell'header file <sys/shm.h>, sono: SHM_R, SHM_W e SHM_RW. Se per flag viene selezionato il valore SHM_R allora l'accesso ad essa è richiesto in sola lettura. SHM_W indica la richiesta di accesso in sola scrittura. SHM_RW indica invece accesso sia in lettura che in scrittura. In caso

di successo, la chiamata **shmat()** ritorna l'indirizzo effettivo dove la memoria condivisa è stata collegata. In caso di fallimento viene restituito il valore -1.

In Figura 5.3 viene riportato un esempio di utilizzo della chiamata **shmat()** per collegare la memoria condivisa avente chiave 40 allo spazio di indirizzamento. Dopo il collegamento, viene scritta la stringa "saluti" a partire dal primo byte della memoria condivisa. Successivamente tale stringa viene visualizzata sullo standard output. Prima della terminazione dell'esecuzione, la memoria condivisa vene rimossa, e così anche la stringa precedenetemente scritta.

```
#include <stdio.h>
#include <sys/ipc.h>

#define Errore_(x) { puts(x); exit(1); }

void main() {
    int   ds_shm, ris;
    key_t chiave = 40;
    void *addr;

    /* Creazione/accesso segmento memoria condivisa */
    ds_shm = shmget(chiave, 1024, IPC_CREAT|IPC_EXCL|0666);
    if ( ds_shm == -1 ) Errore_("Errore nella chiamata shmget");

    /* Il segmento viene portato nello spazio di indirizzi del processo */
    addr = shmat(ds_shm, 0, SHM_RW);
    if ( addr == (void*) -1 ) Errore_("Errore nella chiamata shmat");

    /* Scrittura nel segmento di memoria condivisa */
    strcpy(addr,"saluti");
    printf("Stringa copiata in memoria condivisa: %s\n",addr);

    /* Rimozione segmento di memoria condivisa */
    ris = shmctl(ds_shm, IPC_RMID, NULL);
    if ( ris == -1 ) Errore_("Errore nella chiamata shmctl");
}
```

Figure 5.3: Un esempio di utilizzo della chiamata shmat()

Come ultima chiamata di sistema per la gestione della memoria condivisa abbiamo la **shmdet()**, utilizzata per scollegare una memoria condivisa dallo spazio di indirizzamento di un processo. essa è descritta in forma tabellare come segue:

int **shmdt**(const void *addr)	
descrizione	invoca lo scollegamento di una memoria condivisa dallo spazio di indirizzamento del processo
argomenti	*addr: indirizzo della memoria condivisa da scollegare dallo spazio di indirizzamento
restituzione	-1 in caso di fallimento

L'unico parametro della chiamata è l'indirizzo ove la memoria condivisa da scollegare è attualmente collegata. Dopo l'esecuzione della chimata **shmdet()**, la memoria condivisa precedentemente collegata a quel indirizzo non risulta più accessobile al processo. E' da notare che ae un processo cha ha una memoria condivisa collegata al suo spazio di indirizzamento termina la sua esecuzione, allora la memoria condivisa viene automaticamente scollegata poichè lo spazio di indirizzamento del processo viene completamente eliminato. In Figura 5.4 viene riportato lo stesso codice di Figura 5.3 con in più la chiamata **shmdet()** per scollegare la memoria condivisa dallo spazio di indirizzamento prima della sua rimozione dal sistema.

```
#include <stdio.h>
#include <sys/ipc.h>

#define Errore_(x) { puts(x); exit(1); }

void main() {
    int   ds_shm, ris;
    key_t chiave = 40;
    void *addr;

    /* Creazione/accesso segmento memoria condivisa */
    ds_shm = shmget(chiave, 1024, IPC_CREAT|IPC_EXCL|0666);
    if ( ds_shm == -1 ) Errore_("Errore nella chiamata shmget");

    /* Il segmento viene portato nello spazio di indirizzi del processo */
    addr = shmat(ds_shm, 0, SHM_RW);
    if ( addr == (void*) -1 ) Errore_("Errore nella chiamata shmat");

    /* Scrittura nel segmento di memoria condivisa */
    strcpy(addr,"saluti");
    printf("Stringa copiata in memoria condivisa: %s\n",addr);

    /* Il segmento viene tolto dallo spazio di indirizzi del processo */
    ris = shmdet(addr);
    if ( ris == -1 ) Errore_("Errore nella chiamata shmdet");

    /* Rimozione segmento di memoria condivisa */
    ris = shmctl(ds_shm, IPC_RMID, NULL);
    if ( ris == -1 ) Errore_("Errore nella chiamata shmctl");
}
```

Figure 5.4: Un esempio di utilizzo della chiamata shmdet()

5.3 Un esempio di applicazione

In questa sezione verrà descritto un esempio di applicazione delle chiamate di sistema presentate in questo capitolo. In particolare, verrà implementato un trasferimento di stringhe tra due processi realizzato tramite l'uso di una memoria condivisa. Questa applicazione è analoga a quella descritta nel Capitolo 4 riguardante l'uso della coda di messaggi.

Il programma, mostrato in Figura 5.5, è strutturato nel seguente modo. Esiste un codice padre che crea la memoria condivisa e dà poi luogo ad un primo figlio (scrittore) che prende stringhe in ingresso sullo standard input e le trasferisce nella memoria condivisa. Si suppone che le stringhe non siano più lunghe di 15 caratteri. Il trasferimento termina quando viene digitata la stringa "quit". Si suppone anche che le stringhe trasferite non saturino la mamoria condivisa. Notare che ogni stringa wiene scritta spiazzandosi all'interno della memoria condivisa di 20 byte dall' inizio della stringa precedente (qualora quiesta esista). Al termine del trasferimento, viene attivato un secondo figlio (lettore) il quale rivisualizza sullo standard output le stringhe precedentemente immesse nella memoria condivisa. Al termine della rivisualizzazione il padre rimuove la memoria condivisa. Notare a differenza di quanto accadeva con la coda di messaggi, in questo caso i due figli non vengono creati contemporaneamente, bensì, il figlio lettore viene creato solo dopo che l'altro figlio ha terminato (questo tipo di sincronizzazione è realizzata tramite le azioni del padre associate alla chiamata di istema `wait()`).

Il lettore e lo scrittore non possono essere lanciati contemporaneamente poichè il lettore deve accedere alla memoria condivisa solo dopo che vi sia certezza che la stringa da leggere sia già stata scritta dallo scrittore. Vedremo nel Capitolo 7 come un differente tipo di sincronizzazione che non richiede intervento da parte del padre possa essere implementata tramite l'utilizzo del costrutto di sincronizzazione semaforo.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define DISP_ 20
#define Errore_(x) { puts(x); exit(1); }

char messaggio[256];

void scrittore(int ds) {
    char *p;

    p = shmat(ds, 0 , SHM_W);
    if ( p == (char*) -1 ) Errore_("Errore nella chiamata shmat");

    puts("Digitare le parole da trasferire in memoria condivisa ('quit' per terminare):");
    do {
        scanf("%s", messaggio);
        strncpy(p, messaggio, DISP_);
        p += DISP_;
    } while( (strcmp(messaggio,"quit") != 0));

    exit(0);
}

void lettore(int ds) {
    char *p;

    p = shmat(ds, 0 , SHM_R);
    if ( p == (char*) -1 ) Errore_("Errore nella chiamata shmat");

    printf("Contenuto memoria condivisa: \n");
    while( (strcmp(p,"quit") != 0)) {
        printf("%s\n", p);
        p += DISP_;
    }

    exit(0);
}

int main(int argc, char *argv[]) {
    int  ds_shm, rit, status;
    long chiave=30;

    ds_shm = shmget(chiave, 1024, IPC_CREAT|0666);
    if ( ds_shm == -1 ) Errore_("Errore nella chiamata shmget");

    if (fork()) wait(&status);
    else scrittore(ds_shm);

    if (fork()) wait(&status);
    else lettore(ds_shm);

    ris = shmctl(id_shm, IPC_RMID, NULL);
    if ( ris == -1 ) Errore_("Errore nella chiamata shmctl");
}
```

Figure 5.5: Trasferimento di stringhe tramite memoria condivisa

Chapter 6

Costrutti per la comunicazione: PIPE e FIFO

In questo capitolo vedremo due costrutti di comunicazione concettualmente simili, *PIPE* e *FIFO*, che permettono a più processi di comunicare come se stessero accedendo a dei file sequenziali.

I PIPE sono tra i primi costrutti di comunicazione offerti da UNIX sin dagli albori di questo sistema operativo. Tuttavia, essi presentano pesanti limiti, come il fatto che la comunicazione può essere solo monodirezionale, e che i processi che usano un PIPE devono avere un antenato in comune nell'albero dei processi che lo abbia predisposto. I FIFO, noti anche come *PIPE con nome* (dall'inglese 'named pipe'), evitano questo inconveniente.

6.1 Chiamata `pipe()`

Il termine 'pipe' significa tubo in Inglese. I PIPE consentono una comunicazione monodirezionale che sfrutta, come vedremo, il concetto di *file descriptor condiviso*.

Ad ogni PIPE sono associati due descrittori: uno serve esclusivamente per la scrittura, mentre l'altro solo per la lettura. Una volta lette, le informazioni 'spariscono' dal PIPE e non possono più ripresentarsi, a meno che non vengano riscritte all'altra estremità del 'tubo'.

A livello di sistema operativo, i PIPE non sono altro che buffer di dimensione più o meno grande (solitamente 4096 byte), e quindi è possibilissimo che un processo venga bloccato se tenta di scrivere su un PIPE già pieno, come è anche possibile che un processo venga bloccato li lettura su un PIPE vuoto.

Si noti che non è possibile compiere operazioni di seek, per spostarsi avanti e indietro nel flusso di dati presente nel PIPE e questo li pone in perfetta analogia con i file ad accesso sequenziale.

I PIPE sono anche usabili per la comunicazione di più processi, senza limitarsi a due: è possibile avere un processo scrittore e molti lettori, molti scrittori ed un lettore, molti scrittori e molti lettori, anche se in questi casi si presentano problemi di sicronizzazione.

Ogni kernel UNIX associa ad un processo una tavola dei suoi file aperti, indicizzata dai descrittori di file. Quando si esegue una `fork()` per creare un nuovo processo, la tavola dei file aperti viene duplicata assieme ad altre infomazioni (stack, tabella dei segnali, area dati). Ciò permette di condividere dei descrittori di file tra un processo ed i suoi figli, rendendo quindi possibile, come vedremo, l'uso di uno o più PIPE per farli comunicare tra loro.

La chiamata di sistema per creare un PIPE è descritta nella seguente tabella:

`int pipe(int fd[2])`	
inclusioni	`#include <unistd.h>`
descrizione	invoca la creazione una PIPE
argomenti	*fd: puntatore ad un buffer di due interi (in fd[0] viene restitutito il descrittore di lettura dalla pipe, in fd[1] viene restituito il descrittore di scrittura sulla pipe)
restituzione	-1 in caso di fallimento

La chiamata `pipe()` crea una coppia di descrittori (tecnicamente, inizializza un array di due descrittori) associati ad un PIPE con la seguente semantica:

- `fd[0]` è un canale aperto in lettura che consente ad un processo di leggere dati da un PIPE;
- `fd[1]` è un canale aperto in scrittura che consente ad un processo di immettere dati sul PIPE;

`fd[0]` e `fd[1]` possono essere usati come se fossero normali descrittori di file tramite le chiamate `read()` e `write()`.

I PIPE non sono dispositivi fisici, ma logici, pertanto viene spontaneo chiedersi come un processo sia in grado di vedere la fine di un file su un PIPE. Per convenzione, ciò avviene quando tutti i processi scrittori che condividevano il descrittore `fd[1]` lo hanno chiuso. Tecnicamente, in questo caso la chiamata `read()` effettuata da un lettore restituisce zero come notifica dell'evento che tutti gli scrittori hanno terminato il loro lavoro. Allo stesso modo, un processo scrittore che tenti di scrivere sul descrittore fd[1] quando tutte le copie del descrittore fd[0] siano state chiuse (non ci sono lettori sulla PIPE), riceve il segnale `SIGPIPE`, altrimenti detto *Broken pipe*.

Per fare in modo che tutto funzioni correttamente e non si verifichino situazioni di deadlock è necessario che tutti i processi chiudano i descrittori che non gli servono, usando una normale `close()`.

Si noti che ogni processo lettore che erediti la coppia (`fd[0]`,`fd[1]`) deve necessariamente chiudere la propria copia di `fd[1]` prima di mettersi a leggere da `fd[0]` dichiarando così di non essere uno scrittore. Se così non facesse, l'evento "tutti gli scrittori hanno terminato" non potrebbe mai avvenire se il lettore è impegnato a leggere, e si avrebbe un deadlock.

6.1.1 Un esempio di applicazione

In figura 6.1 viene riportato un esempio di utilizzo della PIPE in UNIX per realizzare un trasferimento di informazione tra due processi. Si ha un processo padre che genera un PIPE e poi un figlio: il padre prende stringhe in ingresso sullo standard input e tramite il PIPE le trasferisce al figlio che le invia a sua volta sullo standard output. Il trasferimento termina quando viene digitata la stringa "quit".

Si noti che sia il processo padre che il processo figlio chiudono i descrittori che non usano subito dopo la chiamata `fork()`.

6.2 Chiamata `mkfifo()`

In questa sezione vediamo come è possibile creare un FIFO per mettere in comunicazione due processi non correlati, usando modalità simili a qualle viste fino ad ora per i PIPE classici. Diversamente dai PIPE, che non possono esistere indipendentemente dai processi che li creano, i FIFO risiedono nel file system e sono quindi individuabili da un pathname.

Questo permette a due processi, uno che scrive ed uno che legge dallo stesso FIFO, di comunicare come se stessero interagendo (apertura, lettura/scrittura, chiusura) con un normalissimo file ad accesso sequenziale. Diversamente da questo però, i dati letti non si possono più incontrare ad una seconda lettura poichè vengono rimossi durante la prima.

La seguente tabella illustra la chiamata utilizzata per creare un FIFO nel file system:

```
#include <stdio.h>

#define Errore_(x) { puts(x); exit(1); }

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

    char messaggio[30];
    int  pid, status, fd[2];

    ret = pipe(fd); /* crea un PIPE */
    if ( ret == -1 ) Errore_("Errore nella chiamata pipe");

    pid = fork();   /* crea un processo figlio */
    if ( pid == -1 ) Errore_("Errore nella fork");

    /* processo figlio: lettore */
    if ( pid == 0 ) {

        /* il lettore chiude fd[1]: importante! */
        close(fd[1]);

        while( read(fd[0], messaggio, 30) > 0 )
            printf("letto messaggio: %s", messaggio);

        close(fd[0]);
    }

    /* processo padre: scrittore */
    else {

        close(fd[0]);

        puts("digitare testo da trasferire (quit per terminare):");

        do {
            fgets(messaggio,30,stdin);
            write(fd[1], messaggio, 30);
            printf("scritto messaggio: %s", messaggio);
        } while( strcmp(messaggio,"quit\n") != 0 );

        close(fd[1]);

        wait(&status);
    }

}
```

Figure 6.1: Trasferimento di stringhe tramite pipe

int **mkfifo**(char *fifo_name, int mode)	
inclusioni	`#include <unistd.h>`
descrizione	invoca la creazione un FIFO
argomenti	*fifo_name: puntatore ad una stringa che identifica il nome del FIFO da creare mode: intero che specifica modalità di creazione e permessi di accesso al FIFO
restituzione	-1 in caso di fallimento

La rimozione di un FIFO dal file system avviene esattamente come per i file normali mediate la chiamata di sistema `unlink()`, la funzione di libreria ANSI C `remove()` o il comando `rm` da shell.

Normalmente, la lettura da un FIFO è bloccante, nel senso che il processo che tenti di aprirla il lettura (scrittura) viene bloccato fino a quando un altro processo non la apre in scrittura (lettura). Se si vuole inibire questo comportamento è possibile aggiungere la flag `O_NONBLOCK` al valore `mode` passato alla `open()`.

Ogni FIFO deve avere sia un lettore che uno scrittore: se un processo tenta di scrivere su un FIFO che non ha un lettore esso riceve il segnale `SIGPIPE` da parte del kernel.

6.2.1 Un esempio di applicazione

In questa sezione viene riportato un esempio di utilizzo delle FIFO in UNIX per realizzare un'architettura software di tipo cliente/servente. Il servente (Figura 6.2.1) accetta richieste su di una FIFO ben nota dal nome *serv*, e risponde ai client (Figura 6.3) su FIFO specificate da essi con nomi rappresentati da caratteri alfabetici minuscoli. Il server è concorrente. Notare che in questo esempio non viene affrontato il problema dei processi *zombie* originati dalla concorrenza del server. Per un esempio di server con trattazione degli zombie, si veda il capitolo 8.

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

typedef struct {
    long type;
    char fifo_response[20];
} request;

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

    char  *response = "fatto";
    int pid, fd, fdc, ret;
    request r;

    ret = mkfifo("serv", O_CREAT|0666);
    if ( ret == -1 ) {
        printf("Errore nella chiamata mkfifo\n");
        exit(1);
    }

    fd = open("serv",O_RDWR);

    while(1) {
        ret = read(fd, &r, sizeof(request));
        if (ret != 0) {

         pid = fork();
            if (pid == 0) {
                printf("Richiesto un servizio (fifo di restituzione = %s)\n", r.fifo_response);
                /* switch sul tipo di messaggio: da implementare */
                sleep(10); /* emulazione di ritardo per il servizio */
                fdc = open(r.fifo_response,O_WRONLY);
                ret = write(fdc, response, 20);
                ret = close(fdc);
                exit(0);
            }
        }
    }
}
```

Figure 6.2: Esempio di servente implementato con FIFO

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

typedef struct {
    long type;
    char fifo_response[20];
} request;

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

    int pid, fd, fdc, ret;  request r;
    char response[20];

    printf("Selezionare un carattere alfabetico minuscolo: ");
    scanf("%s",r.fifo_response);

    if (r.fifo_response[0] > 'z' |r.fifo_response[0] < 'a' ) {
        printf("carattere selezionato non valido, ricominciare operazione\n");
        exit(1);
    }
    r.fifo_response[1] = '\0';

    ret = mkfifo(r.fifo_response, O_CREAT|0666);
    if ( ret == -1 ) {
        printf("\n servente sovraccarico - riprovare \n");
        exit(1);
    }

    fd = open("serv",1);
    if ( fd == -1 ) {
        printf("\n servizio non disponibile \n");
        ret = unlink(r.fifo_response);
        exit(1);
    }

    ret = write(fd, &r, sizeof(request));
    ret = close(fd);

    fdc = open(r.fifo_response,O_RDWR);
    ret = read(fdc, response, 20);
    printf("risposta = %s\n", response);

    ret = close(fdc);
    ret = unlink(r.fifo_response);
}
```

Figure 6.3: Esempio di cliente implementato con FIFO

Chapter 7

Costrutti per la sincronizzazione: i semafori

Un semaforo è una primitiva di sincronizzazione che fornisce e protegge il sistema operativo, e ciò garantisce l'atomicità delle operazioni fatte su di esso. Un semaforo, nella sua definizione classica, è una variabile contente un valore non negativo.

Esso può essere meglio descritto come un contatore usato per moderare l'accesso a risorse condivise a più processi. I semafori sono molto spesso usati come semplice meccanismo di mutua esclusione per evitare che una risorsa attualmente detenuta da un processo venga acceduta da altri processi.

Ad un semaforo si accede mediante due primitive atomiche note come *signal* e *wait* che servono rispettivamente per incrementarne e decrementarne il valore. Nel caso della wait, il decremento avviene solo se il volore del semaforo è maggiore di zero, altrimenti il processo che invoca la wait viene sospeso fino a quando un altro processo non lo incrementa e quindi il decremento diviene possibile.

I semafori sono stati apparsi per la prima volta nella versione System V di UNIX. In questa implementazione, essi non detengono necessariamente un unico valore, ma piuttosto un array di valori. Non ci sono le classiche operazioni di wait e signal, ma bisogna 'costruirsele' avendo a disposizione una serie di chiamate a sistema molto potenti, anche se piuttosto macchinose, che vedremo in questo capitolo.

I passi necessari all'uso di un semaforo sono:

- creare un semaforo, oppure ottenere il descrittore di un semaforo già creato;
- inizializzare il semaforo, se è stato creato da noi;
- operare sul semaforo con funzioni equivalenti alla wait e alla signal;
- distruggere il semaforo quando non è più necessario.

In seguito dettagliamo ciascuno di questi punti, spiegando le chiamate a sistema che devono intervenire. Le inclusioni necessarie per l'uso dei semafori in UNIX sono:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

7.1 Creazione di un semaforo: chiamata `semget()`

In UNIX System V vengono creati un array di semafori, piuttosto che un solo semaforo. Ciascun array di semafori risiede nello spazio di indirizzi di memoria associati al kernel e viene identificato da un numero intero unico nel sistema noto come 'IPC descriptor', esattamente come avviene per code di messaggi e segmenti di memoria condivisa. Questo numero viene restituito dalla chiamata `semget()` che usa una chiave numerica

```
int ApriArraySemafori(key_t chiave, int numero){
    if (numero<1) return -1;
    return semget(chiave, numero, IPC_CREAT | 0660);
}
```

Figure 7.1: Una routine per aprire un array di semafori

unica nel sistema decisa dal programmatore per accedere all'array di semafori. L'IPC descriptor viene poi usato da tutte le chiamate di manipolazione di semafori.

La seguente tabella descrive la chiamata a sistema `semget()` che viene usata per creare un array di un certo numero di semafori (o per accedere ad un semaforo già esistente) a partire da una chiave unica:

int **semget**(key_t key, int number, int flag)	
descrizione	invoca la creazione di un insieme di semafori
argomenti	key: identificatore dell'insieme di semafori da creare number: numero di semafori dell'insieme flag: specifica la modalità di creazione dell'insieme di semafori
restituzione	identificatore numerico intero non negativo per l'accesso al semaforo; -1 in caso di fallimento

Le possibili modalità di creazione di un array di semafori sono le seguenti:

- `IPC_CREAT`: crea il semaforo se non esiste già nel kernel;
- `IPC_EXCL`: se usato assieme a `IPC_CREAT`, provoca il fallimento della chiamata `semget()` qualora un array di semafori avente chiave `key` esista già;

Come per gli altri tipi di IPC UNIX (code di messaggi, ecc.), una maschera di permessi puo essere messa in 'or' con il valore passato per `flag`.

Si noti che l'argomento `number` viene ignorato se si sta accedendo ad un array di semafori già esistente.

In figura 7.1 viene mostrato il codice di una funzione universale che può essere utilizzata per creare/accedere ad un array di semafori identificati da una data chiave.

7.2 Operazioni su un semaforo: chiamata `semop()`

La chiamata a sistema `semop()` può essere utilizzata per incrementare o decrementare i valori assunti da uno o più semafori in un array di semafori. Essa può quindi essere usata per implementare sia l'operazione di signal che di wait. Nella seguente tabella viene mostrato il prototipo della `semop()`:

int **semop**(int sem_des, struct sembuf oper[], int number)	
descrizione	invoca una operazione su un insieme di semafori
argomenti	sem_des: identificatore numerico dell'insieme di semafori su cui operare *oper: puntatore ad un buffer contenente la specifica dell'operazione da eseguire number: numero di elementi del buffer puntato da *oper
restituzione	-1 in caso di errore

La chiamata `semop()` permette il un colpo solo di effettuare `number` operazioni su un array di semafori avente descrittore IPC `sem_des`; le operazioni vengono specificate da un array contente `number` record aventi ciascuno la seguente struttura:

```
int wait(int sem_des, int numero_semaforo){
    struct sembuf operazioni[1] = { { numero_semaforo, -1, 0 } };
    return semop(sem_des, operazioni, 1);
}

int signal(int sem_des, int numero_semaforo){
    struct sembuf operazioni[1] = { { numero_semaforo, +1, 0 } };
    return semop(sem_des, operazioni, 1);
}
```

Figure 7.2: Implementazione delle primitive *wait* e *signal* utilizzando la chiamata `semop()`

```
struct sembuf {
    ushort sem_num;
    short  sem_op;
    short  sem_flg;
};
```

Ciascun record contiene:

- `sem_num` è un valore compreso tra zero ed il numero di semafori nell'array di semafori meno uno ed indica il semaforo su cui operare;
- `sem_flg` indica particolari opzioni con cui deve essere effettuata l'operazione (se non ci sono opzioni usare zero): in particolare, se vale `IPC_NOWAIT` allora l'operazione sul semaforo non è mai bloccante, cioè il processo che invoca la `semop()` non viene mai sospeso in attesa che qualche altro processo effettui una modifica del semaforo;
- `sem_op` è un valore positivo, negativo o zero, che si vuole venga sommato algebricamente al valore corrente del semaforo su sui si agisce con la seguente semantica (assumiamo che x sia il valore corrente del semaforo `sem_num`):
  - `sem_op` è un valore negativo: se $x+$`sem_op` < 0 il processo che ha invocato la `semop()` viene sospeso (o la `semop()` fallisce se `IPC_NOWAIT` è specificato nel campo `sem_flg`) fino a che $x +$ `sem_op` ≥ 0 e quindi l'operazione viene effettuata decrementando x di `-sem_op`; questa modalità permette di realizzare la primitiva *wait*;
  - `sem_op` è un valore positivo: l'operazione viene effettuata incrementando x di `sem_op`; questa modalità permette di realizzare la *signal*;
  - `sem_op` è zero: il processo viene sospeso fino a che x non diventa zero;

In figura 7.2 è mostrata l'implementazione delle primitive *wait* e *signal* utilizzando la chiamata `semop()`.

7.3 Inizializzazione e rimozione di un semaforo: chiamata `semctl()`

La `semctl()` è la più generale e complessa delle chiamate di sistema relative ai semafori. Essa consente di effettuare diverse operazioni (comandi) su un semaforo o su tutti i semafori di un array. Nella seguente tabella è dato il prototipo della `semctl()`:

int **semctl**(int sem_des, int sem_num, int cmd, union senum arg)	
descrizione	invoca una operazione su un insieme di semafori
argomenti	sem_des: identificatore numerico dell'insieme di semafori su cui operare sem_num: numero del semaforo su cui eseguire il comando cmd: specifica del comando da eseguire arg: argomenti del comando da eseguire
restituzione	-1 in caso di errore

La seguente struttura definisce il formato dei dati che vengono passati alla chiamata `semctl()` (omettiamo alcuni campi meno usati):

```
struct semun {
    int     val;      /* usato se cmd == SETVAL          */
    ushort *array;    /* usato se cmd == GETALL o SETALL */
};
```

Il parametro chiave che specifica l'azione della chiamata `semctl()` è `cmd` che può assumere uno di questi valori (elenchiamo soltanto i più frequentemente utilizzati):

- `IPC_RMID`: rimuove dal kernel l'array di semafori identificato da `sem_des`; gli altri parametri vengono ignorati;
- `GETALL`: il valore di tutti i semafori dell'array identificato da `sem_des` viene copiato in un buffer precedentemente allocato dal programmatore a puntato dal campo `array` del record `arg`;
- `SETALL` il valore di tutti i semafori dell'array identificato da `sem_des` viene inizializzato con i valori memorizzati in un buffer precedentemente predisposto dal programmatore a puntato dal campo `array` del record `arg`; il semaforo con indice 0 prenderà il valore arg.array[0], quello con indice 1 il valore arg.array[1] ecc.
- `GETVAL` la `semctl()` restituisce il valore del semaforo avente indice `sem_num` nell'array identificato da `sem_des`; gli altri parametri vengono ignorati;
- `SETVAL` il valore memorizzato in `arg.val` viene assegnato al semaforo avente indice `sem_num` nell'array di semafori identificato da `sem_des`;

Nella sezione seguente mostriamo un esempio di applicazione delle chiamate a sistema descritte in questo capitolo.

7.4 Un esempio di applicazione

In Figura 7.4 ed in Figura 7.4 viene riportato un esempio di utilizzo di un semaforo in UNIX per arbitrare l'accesso ad un segmento di memoria condivisa. Si ha un processo padre che crea il semaforo (inizialmente settato ad 1) e la memoria condivisa e da poi luogo a due figli. Uno di essi prende stringhe in ingresso sullo standard input e le trasferisce nella memoria condivisa. Il trasferimento termina quando viene digitata la stringa "quit". Al termine del trasferimento, il figlio accede al semaforo e lo pone a 0. Il secondo figlio era in attesa che il valore di tale semaforo divenisse 0. Quando questo accade, esso rivisualizza sullo standard output le stringhe precedentemente immesse nella memoria condivisa. Al termine della rivisualizzazione il padre rimuove la memoria condivisa ed il semaforo.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>

#define DISP_ 20
#define Errore_(x) { puts(x); exit(1); }

char messaggio[256];

void produttore(int id, int  sem_id) {
    char *p;
    int ret;
    struct sembuf oper;

    p = shmat(id, 0 , SHM_W);
    if ( p == (char*) -1 ) Errore_("Errore nella chiamata shmat");

    puts("Digitare le parole da trasferire in memoria condivisa (quit per terminare):");
    do {
        scanf("%s", messaggio);
        strncpy(p, messaggio, DISP_);
        p += DISP_;
    } while( (strcmp(messaggio,"quit") != 0));

    oper.sem_num =  0;
    oper.sem_op  = -1;
    oper.sem_flg =  0;

    ret = semop(sem_id, &oper, 1);
    if ( ret == -1 ) Errore_("Errore nella chiamata semop");

    exit(0);
}

void consumatore(int id, int  sem_id) {
    char *p;
    int ret;
    struct sembuf oper;

    p = shmat(id, 0 , SHM_R);
    if ( p == (char*) -1 ) Errore_("Errore nella chiamata shmat");

    oper.sem_num = 0;
    oper.sem_op  = 0;
    oper.sem_flg = 0;

    ret = semop(sem_id, &oper, 1);
    if ( ret == -1 ) Errore_("Errore nella chiamata semop");

    puts("Contenuto memoria condivisa:");
    while( (strcmp(p,"quit") != 0)){
        printf("%s\n", p);
        p += DISP_;
    }

    exit(0);
}
```

Figure 7.3: Un esempio di comunicazione tra processi usando semafori e memoria condivisa: produttore e consumatore

```
int main(int argc, char *argv[]) {
    long id_shm, id_sem;
    int  ret, status;
    long chiave_shm=30, chiave_sem = 50;

    id_shm = shmget(chiave_shm, 1024, IPC_CREAT|0666);
    if ( id_shm == -1 ) Errore_("Errore nella chiamata shmget");

    id_sem = semget(chiave_sem, 1, IPC_CREAT|IPC_EXCL|0666);
    if ( id_sem == -1 ) Errore_("Errore nella chiamata semget");

    ret = semctl(id_sem, 0, SETVAL, 1);
    if ( ret == -1 ) Errore_("Errore nella chiamata semctl");

    if ( fork()!=0 ){
        if ( fork()!=0 ){
            wait(&STATUS);
            wait(&STATUS);
        }
        else consumatore(id_shm, id_sem);
    }
    else produttore(id_shm, id_sem);

    ret = shmctl(id_shm, IPC_RMID, NULL);
    if ( ret == -1 ) Errore_("Errore nella chiamata shmctl");

    ret = semctl(id_sem, 0, IPC_RMID , 1);
    if ( ret == -1 ) Errore_("Errore nella chiamata semctl");
}
```

Figure 7.4: Un esempio di comunicazione tra processi usando semafori e memoria condivisa: la funzione `main`

Chapter 8

Gestione di eventi asincroni: i segnali

I segnali sono un semplice mezzo tramite cui un processo può essere notificato di un evento asincrono. I segnali sono talvolta classificati come semplicissimi messaggi, giacchè sono identificati da numeri che permettono di distinguerli. Tuttavia, essi differiscono dai messaggi in diversi modi:

- un segnale può essere inviato in qualsiasi momento, occasionalmente da un altro processo, ma molto più spesso dal kernel come risultato di un evento eccezionale (come ad esempio un errore di calcolo in virgola mobile);
- un segnale non viene necessariamente ricevuto e processato. Implicitamente la maggior parte dei segnali provocano la terminazione del processo a cui sono inviati. Alternativamente, un processo può decidere di ignorare segnali di un certo tipo;
- i segnali non hanno nessun contenuto informativo. In particolare, il ricevente non può conoscere l'identità del mittente;
- i segnali possono essere inviati solo a processi, e non ad una coda di messaggi. Per questo i segnali non possono essere utilizzati come semafori.

Nondimeno, trattare i segnali è molto importante per i programmatori UNIX perchè il kernel invia segnali indipendentemente dalla nostra volontà: se come vedremo non decidiamo di ignorarli esplicitamente o di "catturarli" il nostro processo viene automaticamente terminato.

I segnali dovrebbero essere utilizzati solo per notificare eventi eccezionali, e non come strumento di comunicazione.

Vediamo ora un elenco dei segnali più frequentemente usati, mostrando gli identificatori di costante associati ad essi ed il loro valore numerico:

- `SIGHUP` (1): *Hangup.* Il processo riceve questo segnale quando il terminale a cui era associato viene chiuso (ed esempio nel caso di un xterm) oppure scollegato (ad esempio nel caso di una connessione via modem o via telnet). Spesso molti server rilegono i propri file di configurazione quando ricevono questo segnale.
- `SIGINT` (2): *Interrupt.* Ricevuto da un processo quando l'utente preme la combinazione di tasti di interrupt (solitamente Control+C).
- `SIGQUIT` (3): *Quit.* Simile a `SIGINT`, ma in più il processo genera un *core dump*, ovvero un file che contiene lo stato della memoria al momento in cui il segnale `SIGQUIT` è stato ricevuto. Solitamente `SIGQUIT` viene generato premendo i tasti Control+\.
- `SIGILL` (4): *Illegal Instruction.* Il processo ha tentato di eseguire un'istruzione proibita (o inesistente).
- `SIGKILL` (9): *Kill.* Questo segnale non può essere intercettato in nessun modo dal processo ricevente, che non può fare altro che terminare. E' il modo più sicuro e brutale per "uccidere" un processo.

- `SIGSEGV` (11): *Segmentation violation.* Generato quando il processo tenta di accedere ad un indirizzo di memoria al di fuori del proprio spazio.
- `SIGTERM` (15): *Termination.* Inviato ad un processo come richiesta non forzata di terminazione (cfr. `SIGKILL`). Questo segnale può essere ignorato.
- `SIGUSR1, SIGUSR2` (10, 12): *User defined.* Non hanno un significato preciso, e possono essere utilizzati dai processi utente per implementare un rudimentale protocollo di comunicazione.
- `SIGCHLD` (17): *Child death.* Inviato ad un processo quando uno dei suoi figli termina.

8.1 Chiamata `kill()`

La chiamata di sistema per inviare un segnale ad un processo è descritta nella seguente tabella:

int **kill**(int pid, int segnale)	
descrizione	invia un segnale
argomenti	pid: identificatore di processo destinatario del segnale segnale: specifica del segnale da inviare
restituzione	-1 in caso di fallimento

La chiamata `kill` provoca l'invio del segnale `segnale` al processo con identificativo `pid`. Si ricordi che esiste anche un comando di shell omonimo per inviare segnali ai processi (si veda il capitolo 1.2.1).

8.2 Chiamata `alarm()`

La chiamata `alarm()` può essere invocata da un processo per programmare il kernel ad inviargli il segnale `SIGALRM` dopo lo scadere di un certo tempo. Ciò può essere molto utile per realizzare con facilità compiti periodici o implementare meccanismi di "timeout". La seguente tabella mostra il prototipo della chiamata `alarm()`.

unsigned **alarm**(unsigned time)	
descrizione	invoca l'invio del segnale SIGALRM a se stessi
argomenti	time: tempo allo scadere del quale il segnale SIGALRM deve essere inviato
restituzione	tempo restante prima che un segnale SIGALRM invocato da una chiamata precedente arrivi

Il parametro `time` specifica il tempo in secondi allo scadere del quale il segnale deve essere inviato. Se questo valore viene impostato a zero, allarmi pendenti dovuti a precendenti chiamate ad `alarm()` vengono eliminati. Il valore restituito nel suo nome dalla chiamata `alarm()` è il tempo restante all'invio del segnale `SIGALRM` relativo ad una chiamata precedente di `alarm()`. Se questo valore è zero, non c'è nessun allarme pendente.

Si noti che le impostazioni di `alarm()` vengono ereditate dopo una `fork()` anche da un processo figlio. Tuttavia modifiche successive delle impostazioni di allarme effettuate da padre e figlio sono del tutto indipendenti le une dalle altre.

8.3 Chiamata `signal`

Questa chiamata serve per stabilire cosa deve essere fatto al momento della ricezione di un segnale. Tipicamente, serve per 'armare' un segnale, cioè per associargli una funzione di gestione da eseguire al momento della sua ricezione. Tuttavia, `signal()` può essere usata anche per fare in modo che il processo ignori un segnale oppure segua un comportamento predefinito. La chiamata `signal()` è descritta nella seguente tabella:

void (*__signal__(int sig, void (*ptr)(int)))(int)	
descrizione	specifica il comportamento di ricezione di un segnale
argomenti	sig: specifica del segnale da trattare ptr: puntatore alla funzione di gestione del segnale o `SIG_DFL` o `SIG_IGN`
restituzione	-1 in caso di errore

Il valore restituito da `signal` nel suo nome è il valore precedente del gestore del segnale che viene sovrascritto con `ptr`, oppure -1 in caso di errore. Il primo argomento, `sig`, è il numero del segnale da trattare. Il secondo, `ptr` può essere una delle seguenti cose:

- `SIG_DFL`: setta un'azione di default associata alla ricezione del segnale `sig`. `SIGCHLD` viene ignorato; tutti gli altri provocano la terminazione del processo che li riceve. Un genitore che effettui una chiamata `wait()` verrà notificato nello stato di terminazione che un suo figlio è stato terminato da un segnale, piuttosto che uscire normalmente tramite una chiamata `exit()`.
- `SIG_IGN`: stabilisce che il segnale `sig` deve essere ignorato. In altre parole il processo che invoca la `signal()` diviene immune al segnale `sig`. Il segnale `SIGKILL` non può essere ignorato. In genere, solamente `SIGHUP`, `SIGINT` e `SIGQUIT` dovrebbero essere sempre permanentemente ignorati. La ricezione degli altri segnali dovrebbe essere almeno tracciata in modo da indicare che qualcosa di eccezionale è accaduto. `SIG_IGN` ha una semantica particolare per il segnale `SIGCHLD` come vedremo tra breve.
- *puntatore a funzione*: stabilisce che la funzione puntata da `ptr` deve essere invocata quando il segnale `sig` viene ricevuto. Questo comportamento viene definito in gergo come "cattura di un segnale". Qualunque segnale può essere catturato, tranne `SIGKILL`. La funzione di gestione del segnale deve avere il prototipo: `void gestoreSegnale(int)`.

Il comportamento associato alla ricezione di un segnale viene ereditata da processi padre a processi figlio. Per quanto riguarda l'`exec()`, solo le impostazioni di `SIG_IGN` e `SIG_DFL` vengono mantenute, mentre per ogni segnale armato con un gestore viene automaticamente settato il comportamento di default (infatti il codice del gestore potrebbe non essere più presente dopo la `exec()`). Ovviamente, il nuovo programma può settare una sua nuova gestione dei segnali.

Con l'unica eccezione del segnale `SIGCHLD`, i segnali non vengono mai accodati: essi vengono ignorati, terminano il processo, oppure vengono catturati. Per questa ragione i segnali sono inappropriati come strumento di comunicazione tra processi: se al momento della ricezione un certo tipo di segnale è temporaneamente ignorato, esso viene perso. Un altro problema dei segnali è che interrompono qualunque cosa il processo stesse facendo, e come vedremo questo può creare problemi.

Usare `SIG_IGN` o `SIG_DFL` è facile: i problemi sorgono quando bisogna installare un gestore per il segnale, cioè bisogna armare il segnale.

Quando un segnale che deve essere catturato arriva, la computazione del processo che lo riceve viene interrotta ed accadono (nell'ordine) le seguenti due cose:

1. il segnale è resettato al suo valore di default, che solitamente è la terminazione. In altre parole il segnale viene disarmato. Si noti che `SIGILL` fa eccezione.
2. la funzione di gestione del segnale, precedentemente assegnata con una chiamata `signal()`, viene invocata e riceve come unico parametro il numero del segnale che è stato catturato. Se e quando il gestore termina normalmente (tramite `return` o perchè il corpo della funzione è terminato), la computazione interrotta riprende nel seguente modo:
   - la computazione interrotta era l'esecuzione di una generica istruzione: essa riprende da dove era stata interrotta;
   - la computazione interrotta era una chiamata a sistema su cui il processo era bloccato[1]: la chiamata restituisce -1 e la variabile globale `errno` viene settata al valore `EINTR` (si veda la header `errno.h`). Si noti che la chiamata a sistema non viene ripristinata automaticamente.

[1] Possono essere bloccanti chiamate come `read()`, `write()`, `wait()`, `semop`, e `msgrec()`. Si noti che l'I/O su file, essendo non bloccante, è immune da interruzione.

Si noti che chiamate a sistema che non bloccano il processo sono eseguite in modo atomico e non vengono mai interrotte da alcun segnale. Se un segnale armato arriva, esso viene catturato soltanto al termine dell'esecuzione della chiamata di sistema.

Poichè una chiamata a sistema bloccante interrotta da un segnale non viene ripristinata automaticamente, il modo corretto di invocarla è:

```
while( chiamataSistema() == -1 )
    if ( errno != EINTR) {
        perror("Errore");
        exit(1);
    }
```

Ciò garantisce che se essa è stata abortita a causa di un segnale (ed un eventuale gestore ritorna normalmente), viene rieseguita. Errori di altra natura (`errno` è diverso da `EINTR`) vengono riconosciuti correttamente e la chiamata non viene rieseguita.

Se il programma non prevede la gestione dei segnali, le chiamate a sistema possono essere invece normalmente invocate come segue:

```
if ( chiamataSistema() == -1 ) {
    perror("Errore");
    exit(1);
}
```

Vediamo ora come dovrebbe essere scritto un gestore di un segnale. Le regole generali sono queste:

- la funzione dovrebbe essere il più semplice possibile
- la funzione non dovrebbe invocare chiamate a sistema bloccanti

Una buona soluzione può essere quella di settare il valore di una variabile globale all'interno del gestore del segnale: in questo modo il programma interrotto può accorgersi che un segnale è stato catturato ed eseguire eventualmente istruzioni per la sua gestione effettiva. Ove il segnale debba essere riarmato, questo può essere fatto all'interno del gestore stesso.

Per concludere questa sezione, ritorniamo sull'effetto di una chiamata `signal(SIGCHLD,SIG_IGN)`. Tramite questa chiamata si chiede di ignorare esplicitamente il segnale `SIGCHLD`, corrispondente all'evento di terminazione di un generico processo figlio. Normalmente, lo stato di terminazione di un processo, che viene settato mediante la chiamata `exit()` da parte del processo stesso, viene mantenuto dal sistema operativo nel process control block poichè il processo padre potrebbe farne eventualmente richiesta tramite la chiamata `wait()`.

8.4 Esempi di applicazione

In questo documento sono descritti due esempi di utilizzo dei segnali in UNIX.

Il primo esempio mostrato in Figura 8.1, riporta il codice associato ad un processo che invia se stesso un segnale di allarme `SIGALRM` e lo cattura. Ad ogni cattura corrisponde una scrittura di un messaggio sullo standard output.

Il secondo esempio (vedi Figura 8.3, Figura 8.4 e Figura 8.2) e' lo stesso cliente/servente visto in precedenza (vedere documento sulle FIFO) in cui il servente originale (il padre) specifica in maniera esplicita di voler ignorare il segnale `SIGCHLD`. Il servente cattura anche il segnale di terminazione `SIGTERM`. In tal caso effettua lo shutdown (rimuove la FIFO "serv" e termina). Il cliente, a sua volta, e' tale da inviare a se stesso un segnale `SIGALRM` come timeout per decidere se attendere ancora il servizio dal servente.

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

char c;

void gestione_timeout() {
    printf("Sveglia!\n");
    signal(SIGALRM, gestione_timeout);
    alarm(5);
}

int main(int argc, char *argv[]) {
    alarm(5);
    signal(SIGALRM, gestione_timeout);

    while(1) read(0, &c, 1);
}
```

Figure 8.1: Esempio di uso della chiamata alarm()

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

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

    int pid_server;
    printf("digitare il PID del server: ");
    scanf("%d", &pid_server);
    kill(pid_server, SIGTERM);
}
```

Figure 8.2: Software per lo shutdown del server (invio di SIGTERM)

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

void gestione_segnale_terminazione() {
    printf("SHUTDOWN del server in corso (eliminazione della FIFO)\n");
    unlink("serv");
    printf("SHUTDOWN del server completato\n");
    exit(0);
}

typedef struct {
    long type;
    char fifo_response[20];
} request;

int main(int argc, char *argv[]) {
    char *nome, *response = "fatto";
    int pid, status, fd, fdc, i, ret, my_pid;
    request r;

    my_pid = getpid();

    ret = mkfifo("serv", O_CREAT|0666);
    if ( ret == -1 ) {
        printf("Errore nella chiamata mkfifo\n");
        exit(1);
    }

    fd = open("serv",O_RDWR);
    signal(SIGCHLD, SIG_IGN);
    signal(SIGTERM, gestione_segnale_terminazione);

    while(1) {
        ret = read(fd, &r, sizeof(request));
        if (ret > 0) {
            pid = fork();
            if (pid == 0) {
                printf("richiesto un servizio (fifo di restituzione = %s)\n",
                       r.fifo_response);

                sleep(10);     /* emulazione di un ritardo per il servizio */
                fdc = open(r.fifo_response,O_WRONLY);
                ret = write(fdc, response, 20);
                ret = close(fdc);
                exit(0);
         }
     }
    }
}
```

Figure 8.3: Esempio di servente generico con uso di segnali

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

typedef struct {
    long type;
    char fifo_response[20];
} request;

request r;

int fdc, ret;
char response[20];

void gestione_timeout() {
    char c[2];

    printf("Service timeout (il server non risponde al servizio).\n"
           "Attendere ancora (y/n): ");
    scanf("%s", c);

    if (c[0] == 'n') {
        unlink(r.fifo_response);
        exit(0);
    }
    else {
        alarm(5);
        signal(SIGALRM, gestione_timeout);
        ret = read(fdc, response, 20);
    }
}

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

    int fd;

    alarm(5);
    signal(SIGALRM, gestione_timeout);

    printf("Selezionare un carattere alfabetico minuscolo: ");
    scanf("%s",r.fifo_response);

    if (r.fifo_response[0] > 'z' |r.fifo_response[0] < 'a' ) {
        printf("Selezionato non valido, ricominciare operazione\n");
        exit(1);
    }
    r.fifo_response[1] = '\0';

    ret = mkfifo(r.fifo_response, O_CREAT|0666);
    if ( ret == -1 ) {
        printf("Servente sovraccarico - riprovare \n");
        exit(1);
    }

    fd = open("serv",1);
    if ( fd == -1 ) {
        printf("Servizio non disponibile \n");
        ret = unlink(r.fifo_response);
        exit(1);
    }

    ret = write(fd, &r,24);
    ret = close(fd);
    fdc = open(r.fifo_response,O_RDWR);
    ret = read(fdc, response, 20);
    printf("risposta = %s\n", response);
    ret = close(fdc);

 ret = unlink(r.fifo_response);
}
```

Figure 8.4: Esempio di cliente generico con uso di segnali

Chapter 9

Costrutti per la comunicazione in sistemi distribuiti: socket

Nei capitoli dal 4 al 8 sono stati intodotti costrutti per la comunicazione in ambiente UNIX. Il limite di questi costrutti risiede nel fatto che la comunicazione è limitata a processi gestiti nell'ambito dello stesso sistema UNIX. Per poter far comunicare processi gestiti da sistemi UNIX distinti, uno dei costrutti di comunicazione messi a disposizione da UNIX è noto come *socket*. La comunicazione nell'ambito di sistemi distinti verrà riferita come comunicazione nell'ambito di sistemi distribuiti (notare che tale tipo comunicazione prevede necessariamente la presenza di una rete di comunicazione tra i sistemi).

In questo capitolo analizzeremo le chiamate di sistema per creare e distruggere dei socket e per spedire e ricevere dati tramite essi. E' da notare che le informazioni riportate in questo capitolo non sono esaustive di tutte le problematiche e le potenzialità a riguardo dell'utilizzo dei socket. Per il lettore interessato ad approfondire la comunicazione tramite socket si consiglia la lettura di testi specialistici quali [1] e [2].

9.1 Le chiamate `socket()` e `bind()`

Ogni processo che voglia comunicare nell'ambito di un sistema distribuito deve creare preventivamente un socket. La chiamata di sistema per effettuare tale creazione è la `socket()`, descritta nella seguente tabella:

int **socket**(int domain, int type, int protocol)	
descrizione	invoca la creazione di un socket
argomenti	domain: specifica del dominio di comunicazione relativamente al quale può operare il socket type: specifica la semantica della comunicazione associata alla socket protocol: specifica il particolare protocollo di comunicazione per il socket
restituzione	un intero positivo (descrittore di socket) in caso di successo; -1 in caso di fallimento

Il parametro domain specifica qual è il dominio di indirizzi sul quale si vuole utilizzare il socket. Ad ogni dominio corrisponde una codifica ben precisa degli indirizzi, che richiede uno specifico numero di byte. Attualmente, sono supportate quattro codifiche, corrispondenti ai seguenti quattro valori che si possono assegnare al parametro domain: AF_INET, AF_UNIX, AF_NS e AF_IMPLINK. Il dominio di indirizzi di interesse per la nostra trattazione è AF_INET, che corrisponde al dominio di indirizzi attualmente utilizzati per la comunicazione in Internet. Tali indirizzi vengono anche denominati come indirizzi IP. Vedremo tra breve come un indirizzo IP è strutturato.

Il parametro type specifica qual è la modalità di comunicazione associata al socket. Per questo parametro è possibile scegliere nell'ambito di un certo insieme di valori. Tra questi valori, quelli di interesse nella nostra trattazione sono: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW. E' da notare che una volta fissato il

dominio di indirizzi, non tutte le modalità di comunicazione possono essere utilizzate. Per quanto riguarda il dominio AF_INET, tutte e tre le precedenti modalità possono essere utilizzate.

Alcune volte, fissata la coppia (domain,type), è possibile scegliere tra più protocolli di comunicazione (altre volte invece fissata tale coppia esiste un solo protocollo di comunicazione). Il parametro protocol specifica quale protocollo si vuole effettivamente usare una volta fissata tale coppia qualora esista una possibilità di scelta. Il valore 0 per il parametro protocol indica che si vuole utilizzare il protocollo di default, o eventualmente l'unico disponibile per quella coppia (domain,type).

Per la coppia (AF_INET,SOCK_STREAM) esiste un unico protocollo, noto con il nome *TCP* (Transmission Control Protocol). Per la coppia (AF_INET,SOCK_STREAM) esiste un unico protocollo, noto con il nome *UDP* (User Datagram Protocol). Infine, per la coppia (AF_INET,SOCK_RAW) esiste un unico protocollo, noto con il nome IP. Il protocollo TCP fornisce una modalità di comunicazione orientata alla connessione ed affidabile. Il protocollo UDP fornisce una modalità di comunicazione non orientata alla connessione e non affidabile. Il protocollo IP fornisce la stessa modalità fornita da UDP, però utilizza modi diversi di impacchettamento dei dati da trasferire.

Notare che una comunicazione orientata alla connessione richiede l'instaurazione preventiva di una connessione prima del trasferimento mentre una comunicazione non orientata alla connessione non la richiede (vedremo poi come le connessioni tra socket possono essere instaurate). Inoltre, una comunicazione affidabile garantisce che i dati arrivino a destinazione, mentre una non affidabile non dà questa garanzia.

Ciò che la chiamata `socket()` restituisce in caso di successo è un descrittore di socket (cioè un intero positivo) per le future operazioni sul socket stesso. In caso di fallimento, la chiamata restituisce il valore -1.

Per poter utilizzare un socket per ricevere dati, al socket stesso deve essere assegnato un indirizzo appartenente al dominio di indirizzi specificati nella creazione del socket. Questa operazione è del tutto analoga all'assegnazione di un numero telefonico ad un utente che ha già l'apparecchio per poter comunicare ma che ancora non può essere contattato poichè a quall'apparecchio non corrisponde alcun numero telefonico. La chiamata di sistema per poter assegnare un indirizzo ad un socket è la `bind()`, descritta come segue:

int **bind**(int ds_sock, struct sockaddr *my_addr, int addrlen)	
descrizione	invoca l'assegnazione di un indirizzo alla socket
argomenti	ds_sock: descrittore di socket *my_addr: puntatore al buffer che specifica l'indirizzo addrlen: lunghezza (in byte) dell'indirizzo
restituzione	-1 in caso di fallimento

Il primo parametro, cioè ds_sock, corrisponde al descrittore del socket al quale si vuole assegnare l'indirizzo. Il parametro *my_addr è un puntatore al buffer strutturato di tipo sockaddr, che contiene la specifica dell'indirizzo da assegnare al socket. Il parametro addrlen specifica quanti byte di quel buffer realmente costituiscono l'indirizzo da assegnare. Il terzo parametro è stato introdotto nella specifica del prototipo della chiamata bind() poichè, come accennato precedentemente, è possibile assegnare indirizzi appartenenti a domini differenti i quali necessitano di differenti quantità di byte per la specifica (ovviamente il buffer strutturato di tipo sockaddr è dimensionato in modo da poter contenere indirizzi appartenenti al dominio per cui la loro specifica richiede il massimo numer di byte).

Per il dominio di iteresse nella nostra trattazione, cioè AF_INET, l'indirizzo è strutturato come mostrato in Figura 9.1. Il primo campo (2 byte) specifica ancora che esso è un indirizzo di un certo dominio. Il secondo campo (2 byte) è un numero di porta. Il terzo campo (4 byte) specifica l'identificatore dell'host dove il socket risiede e quello della rete di cui quell'host fa parte. Tale campo viene anche denominato numero IP. La coppia (numero-porta,numero-IP) costituisce di fatto l'indirizzo nel dominio AF_INET. Essa viene anche denominata come indirizzo IP. E da notare che struct sockaddr è una struttura del tutto generale; la sua specifica per quanto riguarda indirizzi IP è una struttura dal nome struct sockaddr_in già organizzata nei campi di cui prima. Tale struttura è mostrata in Figura 9.1.

Quando si vuole assegnare un indirizzo ad un socket basta riempire tali campi e passare il puntatore al buffer alla chiamata bind(). In Figura 9.2 viene mostrato un esempio di utilizzo delle chiamate sock() a bind(). E' da notare che nel preparare il contenuto del buffer my_addr di tipo sockaddr_in, ciò che va specificato nel campo my_addr.in_addr.sin_addr sono i numeri IP degli host per i quali è possibile contattare questo socket (il valore INADDR_ANY specificato nell'header file `netinet/in.h`, sta ad indicare che un

```
struct sockaddr_in {
    short          sin_family;   /* dominio (es. AF_INET)    */
    u_short        sin_port;     /* numero di porta (2 byte) */
    struct in_addr sin_addr;     /* numero IP (4 byte);      */
    char           sin_zero[8];  /* byte non utilizzati      */
}
```

Figure 9.1: La struttura del buffer di tipo sockaddr_in

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

void main() {
    int ds_sock;
    struct sockaddr_in my_addr;

    ds_sock = socket(AF_INET, SOCK_STREAM, 0);

    my_addr.sin_family      = AF_INET;
    my_addr.sin_port        = 1999;
    my_addr.sin_addr.s_addr = INADDR_ANY;

    bind(ds_sock, &my_addr, sizeof(my_addr));
}
```

Figure 9.2: Un esempio di utilizzo delle chiamate socket() e bind()

host con un qualunque numero IP, quindi un qualunque host, può contattare questo socket). Non c'è necessità di specificare il numero IP dell'host che ospita il socket poichè il sistema operativo già lo conosce e lo userà automaticamente per assegnare l'indirizzo al socket. Nel nostro esempio, il numero di porta chiesto per definire l'indirizzo del socket è 1999. Notare che se esiste già un socket attivo con tale numero di porta allora la chiamata bind() restituirà un errore (ritornando il valore -1) e nessun indirizzo verrà assegnato al socket in questione. Nel nostro esempio il socket creato tramite la chiamata socket() è adibito per comunicazione orientata alla connessione ed affidabile poichè, come spiegato precedentemente, la coppia (AF_INET,SOCK_STREAM) specifica TCP come protocollo di comunicazione. Se avessimo voluto un socket per comunicazione non orientata alla connessione e non affidabile allora la chiamata socket() avrebbe dovuto avere i seguenti parametri: `socket(AF_INET, SOCK_DGRAM, 0)`.

Infine, quando un processo non ha più bisogno di un socket per la comunicazione può chiudere tale socket tramite la chiamata `close()`, già analizzata nel Capitolo 2 a riguardo della gestione del file system, passando come parametro della chiamata il descrittore del socket che si vuole chiudere. E' da notare che quando un processo chiude un socket, il socket stesso viene rimosso solo qualora non vi sia alcun altro processo che possieda un descrittore valido per quel socket. Se si considera l'esempio mostrato in Figura 9.3, abbiamo che dopo la creazione del socket, viene effettuata una `fork()`, con conseguente generazione di un figlio il quale eredita dal padre il descrittore del socket. Il padre effettua la chiamata close() passando il descrittore del socket, però il socket non viene rimosso poichè il figlio possiede un descrittore valido per esso. Tale socket viene rimosso solo alla terminazione del figlio, che avverrà dopo che il figlio stesso avrà letto dallo standard input almeno un carattere.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

void main() {
    int ds_sock;
    struct sockaddr_in my_addr;
    char c;

    ds_sock = socket(AF_INET, SOCK_STREAM, 0);

    my_addr.sin_family      = AF_INET;
    my_addr.sin_port        = 1999;
    my_addr.sin_addr.s_addr = INADDR_ANY;

    bind(ds_sock, &my_addr, sizeof(my_addr));

    if ( fork()!=0 ) close(ds_sock)
    else {
        while ( read(0,&c,1) == -1 );
        close(ds_sock)
    }
}
```

Figure 9.3: Un esempio di utilizzo della chiamata close() su un socket

9.2 Comunicazione orientata alla connessione: chiamate accept(), `connect()` e `listen()`

Nella sezione precedente abbiamo visto come creare un socket, assegnargli un indirizzo (in modo tale da poter essere contattati e ricevere dati) e chiudere il socket stesso. In questa sezione vedremo come utilizzare effettivamente i socket per instaurare ed abbattere una connessione nel protocollo orientato alla connessione TCP.

Per poter creare una connessione, deve esistere un processo che crea un socket specificando la coppia (AF_INET,SOCK_STREAM); assegna un indirizzo al socket e si mette poi in attesa di connessioni in ingresso su quel socket. La chiamata di sistema per attendere connessioni su un socket è la `accept()` descritta come segue:

int **accept**(int ds_sock, struct sockaddr *addr, int *addrlen)	
descrizione	invoca l'accettazione di una connesione su una socket
argomenti	ds_sock: descrittore di socket *addr: puntatore al buffer su cui si copierà l'indirizzo del chiamante *addrlen: puntatore al buffer su cuo si scriverà la taglia dell'indirizzo del chiamante
restituzione	un intero positivo indicante il descrittore di una nuova socket in caso di successo; -1 in caso di errore

Il primo parametro indica il descrittore del socket sul quale si desidera attendere connessioni. Il secondo parametro è un puntatore al buffer nel quale verrà scritto l'indirizzo di del socket dal quale è stata lanciata la connessione verso il socket in oggetto. Il terzo parametro è un puntatore al buffer ove verrà restituita la taglia di tale indirizzo. E' fondamentale notare la restituzione della chiamata `accept()`. In caso di fallimento essa è il valore -1; in caso di successo, quello che si ha è che viene creato un nuovo socket del tutto analogo (per quanto riguarda la modalità di comunicazione che può essere supportata) al socket associato al descrittore

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

void main() {
    int ds_sock, ds_sock_acc;
    struct sockaddr_in my_addr;
    struct sockaddr addr;
    int addrlen;

    ds_sock = socket(AF_INET, SOCK_STREAM, 0);

    my_addr.sin_family      = AF_INET;
    my_addr.sin_port        = 1999;
    my_addr.sin_addr.s_addr = INADDR_ANY;

    bind(ds_sock, &my_addr, sizeof(my_addr));
    ds_sock_acc = accept(ds_sock, &addr, &addrlen);

    close(ds_sock);
    close(ds_sock_acc);
}
```

Figure 9.4: Un esempio di utilizzo della chiamata accept()

ds_sock e ne viene restituito il descrittore. L'unica differenza con il socket originale è che il nuovo socket non ha un indirizzo ad esso associato. Questo nuovo socket è realmente connesso con il socket dal quale è stata lanciata la connessione. In Figura 9.4 viene mostrato un esempio in cui il processo rimane in attesa di una connessione su di un socket. Quando la connessione viene instaurata a partire da un altro socket, il processo si sblocca dalla chiamata di sistema `connect()` e gli viene restituito il descrittore del nuovo socket generato. Il processo poi chiude sia il socket originale che quello duplicato. Notare che la chiusura del socket duplicato implica che la connessione che ad esso afferisce viene abbattuta.

Vediamo ora come un processo possa stabilire da un socket una connessione verso un altro socket avente un ben specificato indirizzo e per il quale ci sia un processo che abbia effettuato una chiamata `accept()` per attendere appunto connessioni entranti verso tale socket. La chiamata per lanciare una connessione è la `connect()` descritta come segue:

int **connect**(int ds_socks, struct sockaddr *addr, int addrlen)	
descrizione	invoca la connessione su una socket
argomenti	ds_sock: descrittore del socket dal quale si vuole lanciare la connessione *addr: puntatore al buffer contenente l'indirizzo del socket al quale ci si viole connettere addrlen: la taglia dell'indirizzo del socket al quale ci si vuole connettere
restituzione	-1 in caso di errore

Il primo parametro, cioè ds_sock, identifica il descrittore del socket dal quale si vuole lanciare la connessione (notare che la connessione viene lanciata da un socket verso un altro socket). Il secondo parametro è il puntatore al buffer che contiene la specifica dell'indirizzo del socket verso il quale si vuole lanciare la connessione. Il terzo parametro contiene la specifica della taglia (numero di byte) che costituiscono tale indirizzo. Ricordiamo a tal proposito che gli indirizzi del dominio AF_INET sono specificati da un numero di porta (2 byte) e da un numeri IP (4 byte). essi vanno però scritti in un apposito buffer di tipo sockaddr_in (della stessa taglia di quello di tipo sockaddr) che è strutturato come già spiegato nella sezione precedente. Notare che se si vuole utilizzare un socket solo per stabilire connessioni verso altri socket ma

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

void main() {
    int ds_sock, length, ret;

    struct sockaddr_in addr;
    struct hostent *hp; /* utilizzato per la restituzione della chiamata */
                        /* gethostbyname() commentata nel testo          */

    ds_sock = socket(AF_INET, SOCK_STREAM, 0);

    addr.sin_family = AF_INET;
    addr.sin_port   = 1999;
    hp = gethostbyname("conde.dis.uniroma1.it");
    memcpy(&addr.sin_addr, hp->h_addr, 4);

    ret = connect(ds_sock, &addr, sizeof(addr);
    if ( ret == -1 ) Errore_("Errore nella chiamata connect");

    close(ds_sock);
}
```

Figure 9.5: Un esempio di utilizzo della chiamata connect()

```
#define h_addr h_addr_list[0]; /* indirizzo del buffer di specifica del numero IP */
struct hostent {
    char   *h_name;            /* nome ufficiale dell'host          */
    char   **h_aliases;        /* lista degli alias                 */
    int    h_addrtype;         /* tipo di indirizzo dell'host       */
    int    h_length;           /* lunghezza (in byte) dell'indirizzo */
    char   **h_addr_list;      /* lista di indirizzi dal name server */
}
```

Figure 9.6: Struttura del buffer di tipo hostent

non per accettare connessioni entranti allora non è necessario assegnare un indirizzo a tale socket (cioè non vi è necessità di effettuare la chiamata bind()). Questo è il caso dell'esempio descritto in Figura 9.5, dove viene presentato un codice in cui viene creato un socket e tramite esso si stabilisce una connessione verso un socket con numero di porta 1999 ed afferente ad un host dal nome `conde.dis.uniroma1.it` avente un determinato numero IP. Per poter risalire al numero IP di tale host, è stata utilizzata la particolare chiamata `gethostbyname("conde.dis.uniroma1.it")` che restutuisce un puntatore ad un buffer di tipo hostent strutturato come mostrato in Figura 9.6 (tale buffer viene automaticamnete allocato dalla chiamata stessa).

Il campo h_addr del buffer di tipo hostent è un puntatore ad un buffer di quattro byte dove viene scritto il numero IP associato all'host in questione. Una volta eseguita questa chiamata, il numero IP è disponibile in tale buffer. Esso viene poi copiato nel campo opportuno del buffer addr di tipo sockaddr_in in modo da poter poi effettuare la chiamata connect() passando come secondo parametro il puntatore a tale buffer. Dopo che la connessione è stata instaurata, il socket viene chiuso (abbattendo quindi la connessione stessa). E' da notare che se l'eseguibile corrispondente al codice mostrato in Figura 9.4 venisse lanciato sull'host `conde.dis.uniroma1.it`, allora la connessione lanciata tramite la connect in Figura 9.5 afferirebbe esattamente al socket associato all'esecuzione del codice in Figura 9.4.

Come ultima cosa, il lettore si può domandare cosa può accadere se un processo cerca di effettuare una

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define MAX_BACKLOG 10

void main() {
    int ds_sock, ds_sock_acc, addrlen;
    struct sockaddr_in my_addr;
    struct sockaddr addr;

    ds_sock = socket(AF_INET, SOCK_STREAM, 0);

    my_addr.sin_family      = AF_INET;
    my_addr.sin_port        = 1999;
    my_addr.sin_addr.s_addr = INADDR_ANY;

    bind(ds_sock, &my_addr, sizeof(my_addr));
    listen(ds_sock, MAX_BACKLOG)

    while(1) {
        ds_sock_acc = accept(ds_sock, &addr, &addrlen);
        close(ds_sock_acc);
    }
}
```

Figure 9.7: Un esempio di utilizzo della chiamata listen()

chiamata `connect()` verso un socket per il quale non vi è alcun processo che attualmente sta attendendo connessioni in ingresso tramite la chiamata `accept()`. Quello che succede è che la chiamata connect() fallirà (restutuendo in valore -1) e nessuna connessione verrà instaurata. E' possibile però che il processo che gestice il socket che accetta connessioni in ingresso faccia si che non vi sia fallimento nella chiamata `connect()` effettuata da chi si vuole connettere. Per far questo è possibile specificare quante connessioni entranti si vogliono mantenere momentaneamente sospese (senza provocarne il fallimento) fino a che il processo stesso non esegua una chiamata `accept()` per accettare realmente la connessione. La chiamata di sistema per effettuare questo è la `listen()` descritta come segue:

int **listen**(int ds_sock, int backlog)	
descrizione	invoca l'impostazione del massimo numero di connessioni pendenti
argomenti	sock_ds: descrittore di socket backlog: numero massimo di connessioni da mantenere sospese
restituzione	-1 in caso di errore

Il primo parametro è il descrittore del socket per il quale si vogliono mantenere connessioni pendenti. Il secondo è il numero massimo di connessioni pendenti che si vogliono mantenere. In Figura 9.7 viene riportato un codice simile a quello di Figura 9.4, ma con in più la specifica del numero massimo di connesioni pendenti da mantenere per il socket associato al descrittore ds_sock restituito dalla chiamata `socket()`. Un'ulteriore differenza sta nel fatto che le connessioni sul socket vengono accettate ciclicamente ed inoltre il socket che viene chiuso prima di effettuare un nuovo `accept()` è solo quello duplicato per effetto dell'`accept()` stessa ed avente descrittore ds_sock1. Se si chiudesse anche il socket originale non si potrebbero più accettare connessioni in ungresso su quel socket poiche verrebbe rimosso quindi non avrebbe neanche senso specificare un massimo numero di connessioni pendenti per quel socket poichè queste non verrebbero mai accettate.

Fino ad ora abbiamo visto come creare socket e stabilire connessioni tra di essi. Quando una connessione

tra due socket è stabilita, essa di fatto costituisce un canale di comunicazione bidirezionale a tutti gli effetti. Il modo più semplice per trasferire dati su tale connessione è utilizzare le chiamata `read()` e `write()` già analizzate nell'ambito della gestione del file system passando come primo parametro il descrittore sul socket da cui si viole leggere o su cui si vuole scrivere. Nella seguente sottosezione viene riportato un esempio di applicazione che utilizza appunto le chiamate `read()` e `write()` per gestire il trasferimento di dati.

9.2.1 Un esempio di applicazione

L'esempio riportato riguarda il trasferimento di dati tra due processi realizzato tramite socket utilizzando il protocollo di comunicazione di tipo connesso TCP. Esiste un processo server che accetta connessioni su di un socket avente un indirizzo con numero di porta 1999. Il suo codice è mostrato in Figura 9.8. Quando un cliente instaura una connessione su tale socket, il server accetta stringhe dalla connessione (leggendole tramite la chiamata `read()` con opportuno descrittore di socket passato come parametro) e le visualizza sullo standard output. Questo avviene fino a quando non viene ricevuta la stringa "quit". Tali stringe vengono trasferite sulla connessione dal client, il cui codice è in Figura 9.9 tramite la chiamata `write()` (il client legge tali stringhe dal suo standard input). Al termine il server risponde al client con la stringa "fatto". Il server è concorrente (ogni connessione sul socket duplicato viene gestita da un figlio specifico del server mentre il padre torna ad accettare connessioni entranti sul socket originale tramite la chiamata `accept()`). Come già accennato, il protocollo di comunicazione è TCP, quindi il dominio di indirizzi per il socket è AF_INET ed il tipo per la comunicazione è SOCK_STREAM. Il server risiede sull'host con nome simbolico `conde.dis.uniroma1.it`. Il massimo numero di connessioni sospese in ingresso al server è fissato a 3 tramite la chiamata `listen()`.

Software per il server

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/signal.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define MAX_DIM 1024
#define MAX_CODA 3

void main() {
    int ds_sock, ds_sock_a, rval;
    struct sockaddr_in server;
    struct sockaddr client;
    char buff[MAX_DIM];

    sock = socket(AF_INET, SOCK_STREAM, 0);

    server.sin_family      = AF_INET;
    server.sin_port        = 1999;
    server.sin_addr.s_addr = INADDR_ANY;

    bind(ds_sock, &server, sizeof(server));
    listen(ds_sock, MAX_CODA);

    length = sizeof(client);
    signal(SIGCHLD, SIG_IGN);

    while(1) {

        while( (ds_sock_a = accept(ds_sock, &client, &length)) == -1);

        if (fork()==0) {
            do {
                read(ds_sock, buff, MAX_DIM);
                printf("messaggio del client = %s\n", buff);
            } while(strcmp(buff,"quit") != 0);
            write(ds_sock, "fatto", 10);
            close(ds_sock_a);
            exit(0);
        }
        else close(ds_sock_a);
    }
}
```

Figure 9.8: Il server che gestisce acquisizione di stringhe

Software per il client

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define MAX_DIM 1024
#define MAX_CODA 3

void main() {
int ds_sock, length, res;
struct sockaddr_in client;
struct hostent *hp;
char buff[MAX_DIM];

ds_sock = socket(AF_INET, SOCK_STREAM, 0);

client.sin_family = AF_INET;
client.sin_port   = 1999;

hp = gethostbyname("conde.dis.uniroma1.it");
bcopy(hp->h_addr, &client.sin_addr, hp->h_length);

res = connect(ds_sock, &client, sizeof(client));
if ( res == -1 ) {
printf("Errore nella connect \n");
exit(1);
}

printf("Digitare le stringhe da trasferire (quit per terminare): ");
do {
            scanf("%s", buff);
            write(ds_sock, buff, MAX_DIM);
        } while(strcmp(buff,"quit") != 0);

        read(ds_sock, buff, MAX_DIM);
        printf("Messaggo del server: %s\n", buff);

        close(ds_sock);
}
```

Figure 9.9: Il client che gestice l'invio di stringhe

9.3 Comunicazione non orientata alla connessione: chiamate `sendto()` e `recvfrom()`

Nella sezione precedente abbiamo visto come creare una connession tra socket e gestire il trasferimento di dati. In questa sezione vedremo come utilizzare i socket per realizzare una comunicazione di tipo non orientato alla connessione. Ricordiamo che i protocolli di comunicazione per poter realizzare questo tipo di comunicazione sono due, UDP ed IP. In quanto segue ci concentreremo sul protocollo UDP.

Il modo piú semplice per poter comunicare tramite il protocollo UDP é il seguente. Deve esistere un processo che crea un socket specificando la coppia (AF_INET,SOCK_DGRAM); assegna un indirizzo al socket (tramite la chiamata `bind()` descritta precedentemente) e si mette poi in attesa di dati in arrivo su di esso (non di connessioni in ingresso su quel socket, come accadeva nel protocollo TCP. Una delle chiamate di sistema per attendere dati su di un socket è la `recvfrom()` descritta come segue:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

void main() {
    int ds_sock;
    struct sockaddr_in my_addr;
    struct sockaddr addr;
    int addrlen;
    char buff[1024];

    ds_sock = socket(AF_INET, SOCK_DGRAM, 0);

    my_addr.sin_family      = AF_INET;
    my_addr.sin_port        = 1999;
    my_addr.sin_addr.s_addr = INADDR_ANY;

    bind(ds_sock, &my_addr, sizeof(my_addr));
    if (recvfrom(ds_sock, buff, 1024, &addr, &addrlen)==-1) {

        printf("Errore nella recvfrom \n");
        exit(1);
    }

    close(ds_sock);
}
```

Figure 9.10: Un esempio di utilizzo della chiamata recvfrom()

int **recvfrom**(int sock_ds, void *buff, int size, int flag, struct sockaddr *addr, int *addrlen)	
descrizione	invoca spedizione di un messaggio ad una socket
argomenti	sock_ds: descrittore di socket locale *buff: puntatore al buffer dove scaricare il messaggio size: taglia massima del messaggio da ricevere flag: specifica delle opzioni di ricezione *addr: puntatore al buffer che specifica l'indirizzo della socket remota *addrlen: puntatore al buffer che specifica la taglia dell'indirizzo della socket remota
restituzione	-1 in caso di errore

Il primo parametro indica il descrittore del socket sul quale si desidera attendere dati. Il secondo parametro è un puntatore al buffer nel quale si desiderano ricevere i dati. Il terzo parametro e' la taglia massima, in byte del pacchetto di dati in arrivo che si desidera ricevere. Il quarto parametro è il puntatore ad un buffer nel quale verrà scritto l'indirizzo del socket dal quale sono stati spediti i dati verso il socket in oggetto. Il quinto parametro è un puntatore al buffer ove verrà restituita la taglia di tale indirizzo. E' importante notare la restituzione della chiamata `recvfrom()`. In caso di fallimento essa è il valore -1; in caso di successo, essa restituisce il numero di byte effettivamente ricevuti. In Figura 9.10 viene mostrato un esempio in cui il processo rimane in attesa di dati in ingresso su di un socket. All'arrivo dei dati, o in caso di fallimento della chiamata `recvfrom()`, il processo si sblocca. In caso di fallimento viene mandato un messaggio di errore sullo standard output. In caso contrario, il processo chiude il socket con la chiamata `close()`.

Vediamo ora come un processo possa apedire dati tramite il protocollo UDP da un socket una verso un altro socket avente un ben specificato indirizzo e per il quale ci sia un processo che abbia effettuato una chiamata `recvfrom()` per attendere appunto dati in ingresso verso tale socket. Una delle chiamate per

spedire dati è la `sendto()` descritta come segue:

int **sendto**(int sock_ds, const void *buff, int size, int flag, struct sockaddr *addr, int addrlen)	
descrizione	invoca spedizione di un messaggio ad una socket
argomenti	sock_ds: descrittore di socket locale *buff: puntatore al buffer contenete il messaggio size: taglia del messaggio flag: specifica delle opzioni di spedizione *addr: puntatore al buffer che specifica l'indirizzo della socket remota addrlen: taglia dell'indirizzo della socket remota
restituzione	-1 in caso di errore

Il primo parametro, cioè ds_sock, identifica il descrittore del socket tramite il quale si vogliono spedire i dati. Il secondo parametro è un puntatore al buffer che contiene i dati da spedire. Il terzo parametro specifica quanti byte di tale baffer devono essere effettivamente spediti. Il quarto parametro è il puntatore al buffer che contiene la specifica dell'indirizzo del socket verso il quale si vogliono spedire i dati. Il terzo parametro contiene la specifica della taglia (numero di byte) che costituiscono tale indirizzo.

Analogamente al caso di comunicazione connessa (protocollo TCP) è da notare che se si vuole utilizzare un socket solo per spedire dati verso altri socket ma non per ricevere dati in ingresso, allora non è necessario assegnare un indirizzo a tale socket (cioè non vi è necessità di effettuare la chiamata bind()). Questo è il caso dell'esempio descritto in Figura 9.11, dove viene presentato un codice in cui viene creato un socket e tramite esso vengono spediti 1024 byte verso un socket con numero di porta 1999 ed afferente ad un host dal nome `conde.dis.uniroma1.it` avente un determinato numero IP. Come in esempi precedenti, per poter risalire al numero IP di tale host, è stata utilizzata la particolare chiamata `gethostbyname("conde.dis.uniroma1.it")`. Viene lasciato al lettore il semlice computi di implemetare tramite il protocollo UDP (cmunicazione non connessa) il trasferimento di stringhe tra un processo client ed un processo server analogo a quello descritto nella Sezione 9.2.1 per il caso di comunicazione connessa.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

void main() {
    int ds_sock, ret;

    struct sockaddr_in addr;
    struct hostent *hp; /* utilizzato per la restituzione della chiamata */
                        /* gethostbyname() commentata nel testo          */
    char buff[1024];

    ds_sock = socket(AF_INET, SOCK_DGRAM, 0);

    addr.sin_family = AF_INET;
    addr.sin_port   = 1999;
    hp = gethostbyname("conde.dis.uniroma1.it");
    memcpy(&addr.sin_addr, hp->h_addr, 4);

    ret = sendto(ds_sock, buff, 1024, &addr, sizeof(addr);
    if ( ret == -1 ) Errore_("Errore nella chiamata sendto\n");

    close(ds_sock);
}
```

Figure 9.11: Un esempio di utilizzo della chiamata sendto()

Bibliography

[1] Rago, S.: *UNIX System V Network Programming*, Addison-Wesley, 1993.

[2] Stevens, *UNIX Network Programming*, , 1998.