Università degli Studi di di Verona

FACOLTÀ DI SCIENZE E INGEGNERIA
Corso di Laurea in Informatica

Sistemi Operativi - Laboratorio

Riassunto dei principali argomenti

Autore:
Davide Bianchi

Anno Accademico 2016/2017

Indice

1 Introduzione

Questa dispensa è scritta sulla base del programma di laboratorio del corso di Sistemi Operativi dell'anno accademico 2015/2016 per il corso di Laurea in Informatica, Università di Verona.

È stata omessa una sezione sulle thread.

Il codice della dispensa, scritta in LaTeX, è reperibile seguendo il link `https://github.com/alx79/dispense-univr.git`.

Chiunque volesse contribuire, segnalare errori (sicuramente presenti) o modificare il testo è ben accetto, è sufficiente scrivere ai gestori del repository GitHub (vedi fine documento).

2 Shell UNIX

2.1 Introduzione

La shell è una sezione del sistema operativo che permette all'utente di interfacciarsi direttamente con il kernel del sistema, che controlla direttamente l'hardware della macchina. Per "dialogare" con il resto del sistema utilizza un insieme di librerie e funzioni che semplificano il lavoro dell'utente, automatizzando gruppi di operazioni che altrimenti andrebbero svolte singolarmente.

2.2 Comandi utente e manuale

La sintassi generale di un comando da shell è la seguente:

```
comando [opzioni] [argomenti]
```

I comandi lunghi possono essere continuati sulla riga successiva con un carattere "\" posto alla fine della riga, mentre se si vogliono dare molteplici comandi in una sola volta basta separarli sulla stessa riga con ";". Per determinare l'utente corrente (ogni user è identificato da un *user-id* e/o *group-id*) si possono dare i seguenti comandi:

- `whoami`
- `who`
- `finger`
- `id`

Tutti i comandi sono inoltre documentati (con tutte le opzioni disponibili e gli argomenti che possono essere passati). La documentazione di un comando è reperibile tramite il comando

```
man [comando]
```

2.3 Struttura e comandi per il file system

2.3.1 Path assoluti e relativi

Ogni directory contiene due directory:

- `.` punta alla cartella corrente
- `..` punta alla cartella padre

Ricordare che i file che iniziano con "." sono nascosti. Per la navigazione nel filesystem si usano i *path*, ovvero i percorsi di directory e file. Un path può essere di due tipologie:

- **assoluto:** ha come radice la radice del filesystem (cartella root "\")
- **relativo:** ha origine dalla cartella corrente

2.3.2 File

I file a livello fisico sono semplicemente visti come flusso di byte (*byte stream*), mentre a livello logico sono di 4 tipi:

- Directory: contengono nomi e indirizzi di altri file
- Special file: costituiscono l'entry point per i device (puntano al relativo device driver)
- Link:sono collegamenti ad altri file (hard e soft link)
- File ordinario: tutti gli altri file

Special file Sono file particolari che semplificano le operazioni di lettura/scrittura, rendendo indipendenti l'I/O dal tipo di dispositivo.

Link Sono generalmente collegamenti a file. Ne esistono di due tipi:

- Soft link: contiene semplicemente il nome di un altro file
- Hard link:Nome in una directory che punta ad un i-node (che può essere puntato anche da altri)

N.B.: non si può fare un hard link di directory nè a file su altri file system. Inoltre un file viene rimosso solo quando tutti i suoi hard link vengono cancellati.

Contenuto di directory Il comando `ls` elenca file e directory in una data cartella
Sintassi: `ls [-opzioni] [args]`

Opzioni:

- `-a` mostra file e directory nascosti
- `-l` mostra l'output in formato esteso

- `-g` include/sopprime il nome del proprietario
- `-r` ordine inverso (alfabetico/temporale)
- `-F` (obsoleta) indica file particolari aggiungendo un carattere
- `-R` elenca i file nelle sottodirectory

Spazio su disco Il comando `df` mostra lo spazio occupato nel disco
Sintassi: `df [-opts]`

Opzioni:

- `-k` mostra l'occupazione in byte
- `-h` mostra l'occ. in formato più facilmente leggibile

Occupazione delle directory Il comando `du`mostra l'occupazione del contenuto di una directory
Sintassi: `du [-opts] [dir]`

Opzioni:

- `-a` mostra l'occ. di ciascun file
- `-s` mostra solo il totale complessivo
- `-h` come in `df`

Visualizzazione di file I comandi descritti più avanti sono usati per visualizzare il contentuto di file di testo. `cat` concatena i file passati su stdout, `head` e `tail` visualizzano le prime/ultime righe di un file
Sintassi:

```
cat [files]
head [-opts] [files]
tail [-/+ opts] [files]
```

Opzioni: (`-n` per `head` e `tail`, `-r` e `-f` solo per `tail`.

- `-r` visualizza in ordine inverso
- `-f` rilegge continuamente il file (cazzo vuol dire?)
- `-n` visualizza o salta (con il `+n`) le ultime (prime) n righe

Visualizzazione interattiva di file I seguenti comandi visualizzano per pagine in modo interattivo un file di testo
Sintassi:

```
pg [files]
more [files]
less [files]
```

Non esistono delle vere proprie opzioni, ma si possono dare comandi interattivi utili per la navigazione:

- spazio:pagina successiva
- CR:riga successiva
- b:pagina precedente
- /*pattern*:prossima pagina con *pattern*
- ?*pattern*:pagina prec. con *pattern*
- q:termina programma
- v:edita file corrente

Per la navigazione si possono usare anche n o p.

Copia, spostamento e cancellazione di file Comandi usati per la manipolazione di file. `cp` copia un file, `rm` rimuove file e `mv` li sposta.
Sintassi:

```
cp [-opts] [srcs] [dest]
rm [-opts] [files]
mv [-opts] [files] [dest]
```

Opzioni: (`-r` solo per `cp` e `rm`)

- `-f` non chiede conferma
- `-i` chiede conferma per ogni file
- `-r` opera ricorsivamente nelle sottodir.

Manipolazione di directory Questi sono i comandi che manipolano delle directory.`cd` consente di muoversi nel fs, `pwd` stampa il path della directory corrente, `mkdir` e `rmdir` creano/rimuovono la cartella specificata.
Sintassi:

```
cd [dir]
pwd
mkdir [dir]
rmdir [dir]
```

Non ci sono opzioni.

Gruppi e proprietari I seguenti comandi cambiano gruppo/proprietario del file.
Sintassi:

```
chgrp [-opt] [group] [file]
chown [-opt] [utente:gruppo] [file]
```

Opzioni:

- `-R` opera su sottodirectory in entrambi i casi.

Modifica dei permessi Il comando `chmod` cambia i permessi di un dato file.
Sintassi: `chmod [-opt] [prot] [file]`

Opzioni:

- `-R` opera ricorsivamente sulle sottodirectory

I permessi vengono modificati con la codifica ottale. Si indicano 4,2,1 per i permessi di lettura/scrittura/esecuzione per ogni tipo di utente (owner, gruppo, altri). Sommando la codifica ottenuta si ottengono i permessi da aggiungere/modificare. Esempio: `chmod 755 file.txt` setta i permessi rwx per il proprietario $(4+2+1=7)$, r-x per il gruppo e gli altri utenti $(4+0+1=5)$.

Sticky bit bit impostato solo sulle directory che, a seconda di come è impostato, consente a tutti oppure solo al proprietario e all'amministratore di cancellare, rinominare o modificare il contenuto di una directory.

Campi setuid/setgid: campi modificabili tramite comando `chmod` che consentono di diventare temporaneamente padrone del file o di appartenere allo stesso gruppo del padrone del file.

Ricerca di file e directory Il comando `find` visita il sottoalbero sotto la directory specificata e ritorna i file che rendono vera l'espressione passata.
Sintassi: `find [dir] [expr]`

Espressioni di ricerca possibili:

- `name [pattern]`
- `type [tipo]`
- `group [gruppo]`
- `newer file`
- `atime/mtime/ctime [+/- giorni]`
- `print`
- `size [+/- blocchi]`

Comando diff ritorna le differenze tra due file o directory. Mostra le righe diverse, indicando quelle in più (`a`), quelle cancellate(`d`) e quelle cambiate (`c`).
Sintassi:

```
diff [-opts] [file1] [file2]
diff [-opts] [dir1] [dir2]
```

Opzioni:

- `-b` ignora gli spazi a fine riga, collassando gli altri
- `-i` ignora maiuscolo/minuscolo
- `-w` ignora la spaziatura

Attributi di file modifica gli attributi di un file. Se la data non è specificata la imposta a quella corrente, se il file non esiste lo crea vuoto.
Sintassi: `touch [-opts] [data] [file]`

Opzioni:

- `-a` modifica accesso
- `-m` cambia ultima modifica

2.4 Processi e relativi comandi

Un processo è definito come un programma in esecuzione, ovvero una sequenza di byte interpretata dalla cpu come un insieme di istruzioni e dati. I processi hanno 3 caratteristiche di base:

- sono organizzati in maniera gerarchica
- hanno un PID univoco (assegnato dal sistema)
- hanno una priorità (asseganta dal sistema)

Evolvono attraverso un certo numero di stati (generici):

- In esecuzione in user mode
- In esecuzione in kernel mode
- In attesa
- Pronto

In UNIX i processi eseguono normalmente in foreground hanno tre canali standard connessi al terminale (*stdin*, *stdout*, *stderr*). I processi che sono attivati con la "&" sono eseguiti in background e sono privi di stdin. I normali processi in fg possono essere sospesi con "^Z". I processi sono eseguiti in entrambe le modalità e possono essere scambiati (da bg a fg e viceversa).

2.4.1 Dati relativi ai processi

Consente di analizzare dati relativi ai processi attivi. Gli stati dei processi UNIX sono i seguenti:

- **R** In esecuzione/eseguibile
- **T** Stoppato
- **S** Addormentato
- **Z** Zombie
- **D** In attesa con I/O non interrompibile.

Ci sono particolari processi detti **daemon** che sono vitali e girano sempre in bg, come il processo di gestione delle risorse.
Sintassi: `ps [-opts]`

Opzioni:

- `-a` visualizza processi di tutti gli utenti
- `-x` visualizza anche i processi in background
- `-u` visualizza informazioni relative all'utente

2.4.2 Processi in back/foreground

`fg` e `bg` mettono i processi in bg e fg e `jobs` elenca i processi in background/sospesi.
Sintassi:

```
jobs [-l]
bg [job-id]
fg [job-id]
```

2.4.3 Segnali

manda un segnale al job indicato.
Sintassi:

```
kill [-sig] [pid]
kill [-sig] [job-id]
```

I segnali più usati sono `-9` (kill) e `-1` (hup), ma con il comando `kill -l` si possono visualizzare tutti i segnali disponibili.

2.5 Programmazione della shell

Offre vie di comunicazione rapide con il sistema operativo, tramite script o interattività. Funzionalià avanzate della shell permettono di ridirezionare l'I/O su file e di strutturare pipe tra comandi (usare lo stdout del primo come stdin del secondo). Ogni script di bash inizia con la riga `#!bin/bash`, che specifica il percorso della shell usata.

2.5.1 Assegnamenti e accesso alle variabili

Gli assegnamenti in bash si svolgono alla classica maniera `var1=10` (senza spazi vicino al simbolo =), mentre l'accesso a valori si effettua con il simbolo "$" (`var1=$var2`). Ricordare che i valori delle variabili **sono sempre stringhe**. Per effettuare valutazioni aritmetiche si usa quindi una sintassi del tipo `$(())`.

2.5.2 Cronologia bash

La bash mantiene una cronologia dei comandi dati nel file di default `.bash_history`, modificabile cambiando il valore della variabile d'ambiente `HISTFILE`. Ai comandi passati si può accedere nei seguenti modi:

- `!n` esegue il comando n del buffer, che potrebbe anche non esistere
- `!!` esegue l'ultimo comando
- `!-n!` esegue l'*n-ultimo* comando
- `!^` primo parametro del comando precedente

- `!$` l'ultimo parametro del comando precedente
- `!*` tutti i parametri del comando precedente
- `![str]` l'ultimo comando che inizia con stringa
- `^[str1]^[str2]` nell'ultimo comando sostituisce `str1` con `str2`.

2.5.3 Wildcard

Sono caratteri speciali che hanno funzioni particolari:

- `/` separa i nomi delle dir in un path
- `?` un qualunque carattere
- `*` una qualunque sequenza di caratteri
- `~` la directory di login (*home*)
- `~[user]` la home dell'utente specificato
- `[]` un carattere tra quelli in parentesi
- `{,}` una parola tra quelle in parentesi

2.5.4 Aliasing

Il comando `alias` consente di dare nomi diversi a comandi noti. Tipicamente i nomi dati sono più semplici da memorizzare. `alias` senza parametri elenca gli alias attuali, `alias [nome]='[valore]'` consente di assegnare un alias, `unalias [nome]` cancella l'alias specificato.

Sintassi:

```
alias [nome]='[valore]'
alias
unalias [nome]
```

2.5.5 Ambiente e variabili d'ambiente

Le variabili sono normalmente locali alla bash in cui vengono eseguite. Tuttavia, in casi particolari come quello di una bash padre e una bash figlio è possibile trasmettere le modifiche eseguite, tramite i comandi `export [var]` (da padre a figlio) e `source` (da figlio a padre).

2.5.6 Ridirezione dell'I/O

L'output o l'input di un comando (presi tramite canali stdin e stdout) possono essere ridiretti (*pipe*) su file o in un altro comando.

- `comando [params] < [file]` esegue lo stdin da file
- `[comando] > [file]` stdout su file (se il file non è vuoto viene cancellato quello che c'è dentro

- `[comando] >> [file]` stdout aggiunto in coda al file
- `[comando] >& [file]` stderr e stdout in file
- `[comando1] |[comando2]` pipe tra `comando1` e `comando2`.

2.5.7 Variabili speciali

La bash memorizza gli argomenti della linea di comando in una serie di variabili `$1,$2,...$n`. Vi sono anche variabili particolari come :

- `$$` PID del processo shell
- `$0` nome dello script eseguito
- `$#` numero di argomenti
- `$?` exit-code dell'ultimo programma eseguito in foreground
- `$!` PID dell'ultimo programma eseguito in bg
- `$*` tutti gli argomenti (`"$1 $2...$n"`)
- `$@` tutti gli argomenti (`"$1" "$2"..."$n"`)

2.5.8 Vettori

Le variabili vettore hanno in bash la stessa struttura che assumono gli array in C. Si definiscono enumerando i valori tra parentesi tonde. (`v=(1 2 3)`), e si accede ai campi tramite parentesi quadre (`${v[1]}`). Ricordare che **gli indici partono da 0**.

2.5.9 Input/Output

Per stampare un valore su stdout si usa la keyword `echo [valore]`, mentre per acquisire un valore da stdin si usa `read [var]`.

2.5.10 Condizionale if

Come in altri linguaggi di programmazione, la struttura condizionale `if-else` si usa per imporre l'esecuzione di codice al verificarsi di determinate condizioni. La sintassi più classica è la seguente:

```
if [ cond ];
    then [azioni];
fi
```

oppure, usando `else`/`else if`:

```
if [ cond ];
    then (istr);
elif [ cond ];
    then (istr);
...
else
    (istr);
fi
```

Ricordare che tra le parentesi quadre e la condizione ci va uno **spazio** sia a dx che a sx. **Precisazioni sulle parentesi:** nella struttura `if-else` si possono usare tutti i tipi di parentesi:

- `[...]` la versione classica (comando test)
- `[[...]]` una versione che non considera l'espansione dei pathname
- `((...))` per usare l'ambiente aritmetico (per testare il risultato di operazioni)

Le parentesi tonde singole sono invece usate per modificare le regole di precedenza nello svolgimento di operazioni.

Per specificare una condizione all'interno di una struttura condizionale si usa la seguente sintassi:

- `-n` verifica se l'operando ha lunghezza diversa da 0
- `-z` verifica se l'operando ha lunghezza 0
- `-d` esiste una dir. con nome uguale all'operando
- `-f` esiste un file regolare con nome uguale all'operando
- `-e` esiste un file con nome uguale all'operando
- `-r,-w,-x` esiste un file leggibile/scrivibile/eseguibile
- `-eq,-ne` gli operandi sono interi e uguali/diversi
- `=,!=` gli operandi sono stringhe uguali/diverse
- `-lt,-gt` operando maggiore/minore di un altro
- `-ge,-le` operando maggiore o uguale/minore o uguale all'altro

Possono anche essere utilizzati i simboli classici (`&&`, `!=`, ...) quando si usa l'ambiente aritmetico.

2.5.11 Struttura switch/case

Lo switch case funziona in maniera simile al C. Ricordare che la sintassi `*)` indica il `default` del C. Inoltre le guardie dei casi sono considerate stringhe.

```
case [sel] in
case1) [istr];;
case2) [istr];;
...
*) [istr];;
esac
```

2.5.12 Ciclo for

Il ciclo for è strutturato su una sintassi particolare per la bash, anche se è previsto un for con sintassi come quella del C/Java.

```
for [arg] in [lista]
    [istr]
done
```

La variabile lista può assumere i seguenti valori:

- un elenco di valori
- una variabile (una lista di valori)
- un meta-carattere che si può espandere in una lista di valori (??)

Se non viene messa la clausola `in` il ciclo itera sugli argomenti da linea di comando.

2.5.13 Ciclo while

Come per il `for`, anche il ciclo while ha due sintassi:

```
while [ cond ]
do
    [istr]
done
```

e una come in C/Java. La parte tra `[ ]` funziona come in un `if`.

2.5.14 Ciclo until

Il ciclo until non ha corrispondenti in C/Java, e ha la particolarità di reiterare finchè la condizione specificata è **falsa**.

```
until [ cond ]
do
    [istr]
done
```

2.5.15 Funzioni

Come in altri linguaggi di programmazione in bash si possono scrivere funzioni, da richiamare nel corpo dello script. La sintassi per definire una funzione è:

```
function [nome] {
    [istr]
}
```

La funzione vede come parametri `$1, $2,...` come se si invocasse uno script indipendente. Può ritornare un valore con la keyword `return [valore]`.

2.5.16 Filtrare l'output

Il comando `grep` fa parte dei filtri (come i seguenti), comandi che prendono un input, lo filtrano secondo alcune regole passate come parametro e producono un output. Possono rivelarsi molto utili se ridirezionati su file o in pipe. `grep` cerca se una stringa compare all'interno di un file.
Sintassi : `grep [-opts] [pattern] [file]`

Opzioni:

- `-c` conta le righe contenenti *pattern*
- `-i` ignora il upper/lower case
- `-l` elenca i nomi dei file contenenti *pattern*
- `-n` indica il numero d'ordine delle righe
- `-v` considera le righe non contenenti *pattern*
- `-x` controlla che l'intera linea corrisponda completamente
- `-w` controlla che l'intera parola corrisponda completamente

Le espressioni per `pattern` possono essere normali stringhe oppure:

- `.` un carattere qualunque
- `^` inizio riga
- `$` fine riga
- `*` ripetizione (zero o più)
- `+` ripetizione (una o più)
- `[ ]` un carattere tra quelli in parentesi
- `[^]` un carattere tranne quelli in parentesi
- `\<` inizio parola
- `\>` fine parola

Esistono anche due varianti di `grep`:

- `fgrep [-opts] [str] [file]` i pattern sono stringhe (più veloce e compatto)
- `egrep [-opts] [str] [file]` i pattern sono espressioni regolari estese (più lento e consuma più memoria)

2.5.17 Ordinamento

Il comando `sort` si occupa di ordinare dei dati secondo criterio.
Sintassi: `sort [-opts] [file]`

Opzioni:

- `-b` ignora gli spazi iniziali
- `-d` modo alfabetico
- `-f` ignora upper/lower case
- `-n` modo numerico
- `-o [file]` scrive i dati da ordinare in `file`
- `-r` ordinamento inverso
- `-t [char]` usa `char` come separatore per i campi
- `-k [field]` ordina sul campo/i specificati

2.5.18 Selezione di campi

Il comando `cut` seleziona campi in una riga di testo
Sintassi:

```
cut [-opt+list] [file]
cut [-opt+list] [-opt+char] [-opt] [file]
```

Opzioni:

- `-c` selezione per caratteri
- `-f` seleziona per campi
- `-d [char]` usa `char` come separatore
- `-s` considera solo le linee che contengono il separatore

2.5.19 Contatori di riga

Il comando `wc` può contare numero di righe, parole, caratteri in un file.
Sintassi:`wc [-opts] [file]`

Opzioni:

- `-c` conta solo i caratteri
- `-l` conta solo le righe
- `-w` conta solo le parole

2.5.20 Soppressione dei doppioni

Il comando `uniq` trasferisce l'input sull'output sopprimendo le righe doppie contigue.
Sintassi:`uniq [-opts] [file]`

Opzioni:

- `-u` visualizza solo le righe non ripetute
- `-c` visualizza anche il contatore del numero di righe

2.5.21 Uso dell'output di un comando

È possibile usare l'output di un comando come dati per un comando successivo, tramite l'operatore `"‘[comando]‘"` oppure con `$([comando])`.

3 System call

3.1 Introduzione

Le system call sono delle funzioni del sistema operativo che permettono di interfacciarsi con il kernel. Dal punto di vista della loro invocazione funzionano esattamente come un'invocazione di una funzione in C. In realtà funzionano in maniera leggermente più complicata.

- Esiste una *system call library* che contiene funzioni con lo stesso nome della syscall.
- Le funzioni di libreria eseguono lo switch da user a kernel mode, per eseguire il codice della funzione chiamata.
- La funzione di libreria passa un ID unico al kernel che identifica la syscall chiamata.

Di norma l'utilizzo delle syscall è generalmente meno efficiente delle corrispondenti chiamate alla libreria C. In caso di errore le syscall ritornano un valore (tipicamente -1).

3.2 System call per il file system

3.2.1 Introduzione

Ricordare che in UNIX esistono i seguenti tipi di file :

- Link
- File regolari
- Directory
- Pipe
- Special file

N.B.: gli special file sono file particolari che non contengono dati, ma solo un puntatore al device driver del corrispondente device.

3.2.2 I/O bufferizzato

Le funzioni del sistema come `fopen(), fread(), fwrite(), fclose(), printf()` sono bufferizzate, mentre quelle di POSIX non lo sono. In caso di I/O si parla di *canali* e non di file.

3.2.3 Apertura di file

L'apertura di un file si effettua con una chiamata della funzione `fopen`:

```
int open ( char * name , int access , mode_t mode );
```

I valori che il parametro `access` può assumere sono i seguenti:

- Uno fra `O_RDONLY`, `O_WRONLY` o `O_RDWR`
- Uno o più fra `O_APPEND`, `O_CREAT`, `O_EXCL`, `O_SYNC`, `O_TRUNC`.

Ricordare che **vanno messi in OR**. I valori del parametro `mode` possono essere uno o più fra i seguenti (sempre messi in OR):

```
S_IRUSR S_IWUSR S_IXUSR S_IRGRP S_IWGRP S_IXGRP
S_IROTH S_IWOTH S_IXOTH S_IRWXU S_IRWXG S_IRWXO
```

Le costanti possono essere rimpiazzate usando gli ottali che si usano per i permessi dei file modificati con `chmod`, ma per garantire un minimo di portabilità è meglio usare le costanti. La funzione `open` mette a disposizione anche modi speciali:

- `O_EXCL`: apertura in modo esclusivo (nessun altro processo può aprire/creare)
- `O_SYNC`: apertura in modo sincronizzato (file tipo lock, prima terminano eventuali operazioni di I/O in atto)
- `O_TRUNC`: apertura di file esistente implica cancellazione del suo contenuto.

3.2.4 Creazione di file

La funzione `creat()` permette di creare un file (precisamente un i-node) e di aprirlo **in lettura**.

```
int creat ( char * name , int mode );
```

La funzione prende come parametro `int mode` che funziona allo stesso modo del parametro `access` della funzione `fopen`.

N.B.: nonostante anche la open possa creare un file, è preferibile usare la creat, e usare la open solo per aprire un file già esistente.Inoltre sia la open che la creat devono includere <fcntl.h>

3.2.5 Creazione di directory

Per creare una directory si usa la funzione `mknod`, con la seguente sintassi:

```
int mknod ( char * path , mode_t mode , dev_t dev );
```

Prende come parametro un path, una modalità (vedi tabella) e il parametro `dev`, in cui viene specificato il major/minor number se si tratta di uno special file, viene ginorato in caso contrario. Ricordarsi di includere :

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

La creazione di una directory con creat NON genera le directory . e .., necessarie per la navigazione tra le directory create.

È preferibile usare le funzioni di libreria `mkdir` e `rmdir`.

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

int mkdir ( const char * path , mode_t mode );
int rmdir ( const char * path );
```

Con gli stessi parametri della funzione `creat`.

3.2.6 Manipolazione diretta di file

Le funzioni di read/write consentono di leggere/scrivere dati da file.

```
ssize_t read (int fildes, void * buf, size_t n);
ssize_t write (int fildes, void * buf, size_t n);
```

Queste funzioni necessitano dell'header `unistd .h` per funzionare e ritornano in entrambi i casi il numero di byte letti. Il parametro `size_t` indica la dimensione dei blocchi da leggere (la massima efficienza si ha quando $n = 512$ byte o $n = 1$ kb.

Terminata la lettura del file si può invocare la funzione `close` con i seguenti parametri:

```
int close ( int fildes );
```

dove `fildes` è il file descriptor del file sul quale sono state eseguite le operazioni.

3.2.7 Duplicazione di canali

Duplica un file descriptor che ha in comune con quello precedente il riferimento al file, il puntatore per l'accesso casuale e il modo di accesso. Ritorna il primo descrittore libero partendo da 0.

3.2.8 Accesso alle directory

Per accedere/manipolare alle directory si possono utilizzare le seguenti funzioni:

```
DIR *opendir(char *dirname)
struct dirent *readdir (DIR *dirp)
void rewinddir(DIR *dirpath)
int closedir(DIR *dirpath)
```

Rispettivamente:

- Apertura di una cartella
- Puntatore alla prossima entry nella directory `dirp`
- Resetta la posizione del puntatore all'inizio
- Chiusura della directory

È consigliabile non usare la `open` per creare/manipolare directory, è consigliato l'uso di funzioni C, non system call.

Per navigare all'interno del filesystem, attraversando directory, si può usare la `chdir` con la seguente sintassi: `int chdir(char *dirname)`. È necessario che la directory abbia permessi di esecuzione.

3.2.9 Gestione dei link

Per la creazione/rimozione di link si usano:

```
int link(char *orig_name, char *new_name);
int unlink(char *file_name);
```

Con la prima viene creato un hard link a `orig_name`, con `unlink`, si cancella un file cancellando l'*i-number* nella dir. entry, oppure sottrae 1 al link count dell'inode passato (se il link count è 0, il file viene cancellato).

3.2.10 Privilegi e controllo dell'accesso

La funzione `access` verifica i permessi così passati: `int access (char *file_name, int access_mode);` I permessi sono una combinazione bitwise (*and, or...*) dei valori `R_OK`, `W_OK` e `X_OK`. Viene ritornato 0 se la verifica ha avuto successo. Per *modificare* i permessi si usano le funzioni:

```
int chmod (char *file_name, int mode);
int fchmod(int filedes, int mode);
```

Qui di seguito un riassunto con i permessi utilizzabili:

Per cambiare il proprietario di un file si usa: `int chown (char *file_name, int owner, int group);` Con `owner` e `group` si intendono `UID` e `GID`. `chown` è utilizzabile sono con permessi di super-user.

3.2.11 Informazioni sullo stato di un file

Le seguenti funzioni ritornano informazioni contenute nell'i-node di un file. (All'interno di `stat_buf`.)

```
int stat (char *file_name ,struct stat *stat_buf);
int fstat (int fd ,struct stat *stat_buf);
```

Vedere sulle diapositive i possobili valori di `stat`.

3.2.12 Variabili d'ambiente

La funzione `getenv` ritorna il valore di una variabile d'ambiente, `NULL` se la variabile non è definita.

```
char *getenv (char *env_var);
```

Si può inoltre accedere alla variabile `char **environ`, e si possono analizzare le variabili d'ambiente con il terzo argomento del main (`char *env[]`), da evitare se si vuole ottenere un programma facilmente portabile.

3.3 System call per i processi

3.3.1 Introduzione

La struttura del programma eseguibile è la seguente:

- HEADER: (definito in `usr/include/linux/a.out.h`), contiene la dimensione delle altre sezioni, l'entry point dell'esecuzione e il *magic number*, ovvero un codice speciale per la trasformazione in processo.
- TEXT: sono le istruzioni del programma
- DATA: i dati inizializzati (`static`, `extern`)
- BSS: dati non inizializzati
- RELOCATION: come il loader carica il programma
- SYMBOL TABLE: contiene la locazione, il tipo e lo scope di variabili e funzioni.

La struttura del processo in memoria (processo = programma in esecuzione) è la seguente:

- TEXT/DATA (heap)/BSS: come l'omonima sezione dell'eseguibile
- STACK: creato nella costruzione del processo. Contiene variabili automatiche, parametri delle procedure, argomenti del programma.
- USER BLOCK: sottoinsieme delle informazioni mantenute dal sistema sul processo

3.3.2 Fork

Crea un nuovo processo figlio di quello corrente. Il figlio eredita dal padre i file/directory usati dal padre e le variabili d'ambiente. Nel processo figlio viene replicata solo la thread corrente. Ritorna 0 al figlio, il PID del figlio al padre (-1 in caso di errore). Il codice è semplice:

```
fork();
```

3.3.3 Esecuzione di un programma

Sostituisce all'immagine attuale l'immagine specificata da `*file`. Qui i prototipi:

```
int execl (char* file, char* arg0, char* arg1,(char*) NULL);
int execlp (char* file, char* arg0, char* arg1,(char*) NULL);
int execle (char* file, char* arg0, char* arg1,(char*) NULL,
  char* envp[]);
int execv (char* file , char* argv[]);
int execvp (char* file, char* argv[]);
int execve (char* file, char* argv[], char* envp[]);
```

Ricordare che il codice inserito dopo una exec non viene mai eseguito, in quanto viene caricata una nuova immagine di programma e non ritorna nulla se il programma viene eseguito.

- `execl` utile quando si conoscono in anticipo il numero di args e gli argomenti (altrimenti usare `execv`).
- `execle` e `execve` ricevono come parametro anche la lista delle variabili d'ambiente
- `execlp` e `execvp` usano la var. d'ambiente PATH per cercare il comando del file

Alla fine di ogni chiamata, `(char*) NULL)` viene inserito come terminatore. Devono essere terminati così anche gli array `argv` e `envp`.

3.3.4 Sincronizzazione padre/figlio

Per sincronizzare padre e figlio si usano le funzioni:

```
void exit(int status);
void _exit(int status);
pid_t wait(int *status);
```

La `exit` è usata come wrapper per la syscall `_exit`. La funzione `wait` sospende l'esecuzione di un processo fino a quando uno dei suoi figlio termina. Ritorna il PID e lo stato di terminazione del figlio al padre, -1 se il processo non ha figli.

Un figlio rimane zombie dalla terminazione a quando il padre ne legge lo stato con la wait.

Se il figlio è terminato con `exit` allora la `wait` ritorna:

- Byte 0: (tutti 0)
- Byte 1: argomento della `exit`

Se il figlio è terminato con un segnale allora `wait` ritorna:

- Byte 0: valore del segnale
- Byte 1: tutti 0

Per testare lo stato si possono usare le seguenti macro:

- `WIFEXITED(status)`
- `WEXITSTATUS(status)`
- `WIFSIGNALED(status)`
- `WTERMSIG(status)`
- `WIFSTOPPED(status)`
- `WSTOPSIG(status)`

3.3.5 Famiglia di wait

Oltre alla classica `wait` esistono anche altre due funzioni che offrono funzionalità aggiuntive:

```
pid_t waitpid(pid_t pid, int *status, int options);
pid_t wait3(int *status, int options, struct rusage *rusage);
```

La funzione `waitpid` attende la terminazione di un particolare processo:

- `pid=-1`: tutti i figli del processo corrente
- `pid=0`: tutti i figli con lo stesso GID del processo chiamante
- `pid<-1`: tutti i figli con GID=`-pid`
- `pid>0`: il processo con PID=`pid`

La fuzione `wait3` è simile alla precedente, ma ritorna informazioni sull'uso delle risorse nella struct `rusage`. Vedere il manuale per dettagli (`man rusage`).

3.3.6 Informazioni sui processi

Per ottenere il pid di un processo si utilizzano le funzioni:

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

`getpid()` ritorna il PID del processo corrente, mentre `getppid()` ritorna il pid del padre del processo corrente.

Esistono funzioni che permettono di verificare quale utente o gruppo ha generato un certo processo. Per i *real* UID e GID si usano le funzioni:

```
uid_t getuid();
uid_t getgid();
```

Per gli *effective* UID e GID si usano le funzioni:

```
uid_t geteuid();
uid_t getegid();
```

La differenza tra *real* ed *effective* UID/GID sta nel fatto che con real ci si riferisce sempre ai dati reali dell'utente che lancia il processo, con effective ai dati che possono essere modificati con i comandi `suid` e `sgid`.

3.3.7 Segnalazioni tra processi

I segnali tra processi sono spediti asincronamente (sono quindi sconsigliati per sincronizzare processi, in quanto non garantiscono al 100% la corretta esecuzione del codice).

```
int kill(pid_t pid, int sig);
```

I valori possibili di `pid` sono i seguenti:

- `pid`>0: inviato al processo con PID=`pid`

- `pid`=0: a tutti i processi dello stesso gruppo del processo chiamante
- `pid`=-1: a tutti i processi (tranne quelli di sistema)
- `pid`<-1 a tutti i processi del gruppo `-pid`

Nota: con *guppo di processi* ci si riferisce a tutti i processi aventi un antenato in comune.

Il processo che riceve il segnale può avere una routine che avvia alla sua ricezione:

```
typedef void (*sighandler_t)(int);
sighandler_t signal (int signum, sighandler_t funct);
```

Nel codice sopra la funzione `funct` è la routine da eseguire quando il segnale viene registrato. Può anche essere definita dall'utente oppure `SIG_DFL` per specificare un comportamento di default o `SIG_IGN` per ignorare il segnale ricevuto.

I segnali `SIGKILL` e `SIGSTOP` non possono essere intercettati per ovvi motivi, inoltre il segnale `SIGCLD` è usato quando un figlio segnala al padre la propria terminazione. In questo caso l'azione di default è di ignorare il segnale(può comunque essere intercettato per modificare l'azione da svolgere). Tenere presente che il segnale `SIGCLD` causa lo sblocco della `wait`.

3.3.8 Timeout e sospensione

Tramite appropriata syscall è possibile implementare dei timeout, utili per il controllo di risorse utilizzate da più processi.

```
unsigned int alarm(unsigned seconds);
```

Invia un segnale di allarme (`SIGALRM`) al processo chiamante dopo `seconds` secondi. Ritorna 0 nel caso normale; se esistevano delle `alarm` con tempo residuo, viene ritornato il numero di secondi che mancavano all'allarme. Per eliminare eventuali allarmi sospesi si passa il parametro 0.

La funzione

```
int pause();
```

sospende un processo fino a quando non riceve un *qualsiasi* segnale. Ritorna sempre -1.

Nota: se si usa `alarm` per chiudere una `pause`, inserire dopo la pause un `alarm(0)` per eliminare l'allarme, per evitare che l'allarme scatti nel caso che la `pause` sia stata disinserita a causa di un altro segnale.

3.4 System call per la comunicazione tra processi (IPC)

3.4.1 Introduzione

Le IPC sono syscall che consentono di effettuare comunicazioni tra processi tramite l'utilizzo di strutture dati come pipe, FIFO, semafori e code di messaggi.

Tramite il comando `ipcs [-opts]` si possono ottenere informazioni riguardanti le risorse allocate:

- `-s` per i semafori

- `-m` per la memoria condivisa
- `-q` per le code di messaggi

Tramite il comando `ipcrm` si possono eliminare le risorse (se permesso dal sistema). Si usa con le stesse opzioni di `ipcs` e va specificato un ID di risorsa (ritornato da `ipcs`).

3.4.2 Pipe

Le pipe sono dei canali di comunicazione **unidirezionali e sequenziali** in memoria. Per creare una pipe si usa la funzione:

```
int pipe(int filedes[2]);
```

Ritorna 0 in caso di successo, -1 in caso di errore. La funzione richiede come parametro due file descriptor, il primo per leggere, il secondo per scrivere. Le pipe hanno alcune caratteristiche particolari, quali:

- Non hanno accesso random
- La dimensione fisica di una pipe è limitata (variabile a seconda del sistema)
- L'operazione di scrittura su una pipe è atomica
- La lettura si blocca su una pipe vuota e si sblocca appena è disponibile un byte da leggere

Ogni estremo della pipe è indipendente dall'altro e può essere chiuso (volontariamente tramite funzione o prematuramente). Un estremo di pipe si chiude prematuramente quando:

- Le `read` ritornano 0
- i processi in scrittura ricevono il segnale (`SIGPIPE`)

È possibile aggirare il problema del bloccaggio delle funzioni `read` e `write` con la funzione:

```
fcntl(int fd, F_SETFL, O_NONBLOCK);
```

Il parametro `fd` è uno dei file descriptor su cui è stata modellata la pipe. Impostando il flag `O_NONBLOCK`, le `write` ritornano 0 se la pipe è piena (è impossibile scriverci dentro) e le `read` ritornano subito 0 se la pipe è vuota (nulla da leggere). Questo meccanismo è utile per implementare controlli tipo polling.

Limitazioni delle pipe:

- Possono essere create solo tra processi imparentati
- Non sono permanenti ma vengono distrutte quando il processo che le crea termina

3.4.3 Named pipe (FIFO)

Sono pipe particolari che esistono fisicamente su disco e consentono di aggirare i limiti imposti dalle pipe (vanno rimosse con `unlink`). Il riferimento avviene attraverso un nome, non con i file descriptor. Le FIFO si creano con la seguente funzione:

```
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

Ricordare che l'argomento `dev` è ignorato (passare `0`).
Argomenti per `mode_t mode`:

- `S_IFIFO`
- `S_IRUSR`
- `S_IWUSR`

Questa funzione ritorna 0 in caso di successo, -1 in caso di errore. Apertura, chiusura e operazioni di I/O avvengono come per un file normale (quindi sono ereditate dai figli), inoltre l'I/O è sempre atomico.
L'I/O normalmente è bloccante, ma tramite la `open` e il flag `O_NONBLOCK` la FIFO viene aperta non bloccante (`read` e `write` funzionano quindi come per una pipe non bloccante). In alternativa alla `mknod` si può usare la funzione di libreria `mkfifo`, con gli stessi parametri e valori di ritorno della prima.

3.5 System call per meccanismi di IPC avanzati

Gli IPC avanzati sono principalmente di tre categorie:

- Code di messaggi
- Memoria condivisa
- Semafori

Tutti questi costituenti sono dotati di due primitive:

- `get` per creare nuove entry e recuperare entry esistenti
- `ctl` per verificare lo stato di entry, cambiare lo stato di quelle esistenti e rimuovere entry già presenti.

3.5.1 Primitiva get

La primitiva `get` richiede una *chiave*, usata per la creazione o il recupero dell'oggetto. Richiede inoltre dei flag di utilizzo, ovvero:

- permessi relativi all'accesso
- `IPC_CREAT`: si crea una nuova entry se la chiave non esiste
- `IPC_CREAT + IPC_EXCL`: si crea una nuova entry ad uso esclusivo da parte del processo

L'identificatore ritornato dalla `get` (se il valore è diverso da -1) è un descrittore utilizzabile anche dalle altre syscall. La creazione di un oggetto IPC causa anche l'inizializzazione di:

- una struttura dati, contenente informazioni su:
  - UID, GID
  - PID dell'ultimo processo che ha apportato qualche modifica
  - tempi di ultimo accesso/modifica
- una struttura di permessi `ipc_perm`

La struttura `ipc_perm` è costruita come segue:

```
struct ipc_perm {
    key_t key;
    uid_t uid;
    gid_t gid;
    uid_t cuid;
    gid_t cgid;
    unsigned short int mode;
}
```

e contiene ripsettivamente:

- chiave
- UID del proprietario
- GID del proprietario
- UID del creatore
- GID del creatore
- permessi di read/write

3.5.2 Primitiva ctl

La primitiva `ctl` richiede la specifica di informazioni diverse in base all'oggetto di sistema. In tutti i casi, richiede:

- un descrittore, usato per accedere all'oggetto di sistema (valore intero ritornato dalla `get`)
- comandi di utilizzo (cancellazione, modifica, lettura di informazioni relative agli oggetti)

3.5.3 Funzione ftok

La funzione `ftok()` è usata per ottenere chiavi probabilmente non in uso nel sistema. Si usa con la seguente sintassi: `key_t ftok(const *char pathname, int proj_id);` Come parametri richiede un path ad un file esistente ed accessibile, e una costante espressa su *un byte*. In caso di errore ritorna -1, o una chiave univoca a parità di path/ID.

3.5.4 Code di messaggi

Introduzione Un messaggio è un'unità di informazione di dimensione variabile, mentre una coda è un vero e proprio oggetto di sistema.

Gestione di una coda Con la funzione:

```
int msgget(key_t key, int flag);
```

viene ritornato l'identificatore di una cosa di messaggi se si trova una corrispondenza, altrimenti viene ritornato un errore. È inoltre usata per creare una coda di messaggi data la chiave `key` nel caso in cui: `key==IPC_PRIVATE`, oppure `key!=IPC_PRIVATE` e il flag `IPC_CREAT` è settato a true.

La funzione:

```
int msgctl (int id, int cmd, struct msqid_ds *buf);
```

permette di accedere ai campi della `struct msqid_ds` mantenuta all'indirizzo `buf` per la coda specificata dalla chiave `id`, ovvero il costruttore ritornato da `msgget`. Il comportamento della funzione dipende dal valore dell'argomento `cmd`:

- `IPC_RMID` cancella la coda
- `IPC_STAT` ritorna informazioni relative alla struttura puntata da `buf` (UID, GID, stato della coda)
- `IPC_SET` modifica un sottoinsieme dei campi contenuti nella struct.

La coda può essere cancellata solo con la funzione `msgctl` o riavviando la macchina; una chiusura prematura del programma causata da un segnale del sistema operativo o da un interrupt da tastiera non la rimuove.

La struttura puntata da `buf` è definita in `sys/msg.h` ed è cos' costituita:

```
struct msqid_ds {
    struct ipc_perm msg_perm;
    __time_t msg_time;
    __time_t msg_rtime;
    __time_t msg_ctime;
    unsigned long int __msg_cbytes;
    msgqnum_t msg_qnum;
    msglen_t msg_qbytes;
    __pid_t msg_lspid;
    __pid_t msg_lrpid;
}
```

I campi della struttura contengono rispettivamente:

- permessi (`struct ipc_perm`, vedi sopra)
- ora dell'ultimo comando `msgsnd`
- ora dell'ultimo comando `msgrcv`
- ora dell'ultima modifica

- numero di byte nella coda
- numero corrente di messaggi nella coda
- numero massimo di byte che la coda può contenere
- pid dell'ultimo `msgsnd()`
- pid dell'ultimo `msgrcv()`

Scambio delle informazioni Per mandare dei messaggi su una coda si usa la funzione:

```
int msgsnd(int id, struct msgbuf *msg, size_t size, int flag);
```

dove `id` è l'identificatore della coda , `*msg` un buffer dove il messaggio è salvato, `size` la lunghezza del messaggio. Viene ritornato 0 in caso di successo, -1 in caso di errore. La struttura `msgbuf` è così definita:

```
struct msgbuf {
    long mtype;
    char mtext[1];
}
```

dove `mtype` è il tipo di messaggio e `mtext` è il testo. Questo è solo un template dei messaggi, scrivendo il codice si usa una struct definita dall'utente.

La funzione per ricevere messaggi è la seguente:

```
int msgrcv(int id, struct msgbuf *msg, size_t size, long type, int flag);
```

La funzione legge un messaggio dalla coda `id`, lo scrive nella struttura puntata da `msg` e ritorna il numero di byte letti. Una volta estratto, il messaggio viene rimosso dalla coda. L'argomento `size` indica la lunghezza *massima* del testo del messaggio: se il messaggio ha lunghezza superiore a `size`, il messaggio non viene estratto e la funzione ritorna con un errore.

I flag di `msgsnd` e `msgrcv` sono `IPC_NOWAIT` (non si blocca se non ci sono messaggi da leggere) e `MSG_NOERROR` (tronca il messaggio a `size` byte senza ritornare l'errore), quest'ultima solo per `msgrcv`.

Il valore di `type` indica il messaggio da prelevare:

- `type=0` il primo messaggio (indip. dal tipo)
- `type>0` il primo messaggio di tipo `type`
- `type<0` il messaggio con tipo più vicino al modulo di `type`.

3.5.5 Memoria condivisa

Introduzione Il metodo della memoria condivisa è un altro dei modi tramite i quali i processi possono comunicare. Funziona attraverso la condivisione di una parte dello spazio di indirizzamento proprio di ogni processo e la comunicazione avviene leggendo e scrivendo in questa parte di memoria.

Allocazione La funzione `shmget` ritorna un puntatore alla zona di memoria allocata. Si usa con la seguente sintassi:

```
int shmget (key_t key, size_t size, int flags);
```

dove i parametri hanno lo stesso significato di quelli della funzione `msgget`.

Una volta creata, l'area di memoria non è immediatamente disponibile ai processi, ma deve essere collegata all'area dati dei processi che la vogliono usare, tramite la funzione:

```
void *shmat (int shmid, void *shmaddr, int flag)
```

Nota: l'area di memoria è ereditata dei processi creati con `fork`, ma non con quelli creati da `exec`.

Il parametro `*shmaddr` indica l'indirizzo virtuale dove il processo vuole collegare il segmento di memoria condivisa (tipicamente è `NULL`), mentre il parametro `shmid` è un semplice identificatore del segmento di memoria utilizzato.

In base ai valori di `flag` e `shmaddr` si determina il punto di attacco del segmento:

- se `shmaddr == NULL` la memoria viene posizionata in un punto a scelta del sistema operativo
- se `shmaddr != NULL` e `SHM_RND` è nei flag la memoria viene attaccata al multiplo di `SHMLBA`(**??**) più vicino a `shmaddr` (comunque non oltre `shmaddr`)
- se non è valida nessuna delle condizioni precedenti, `shmaddr` deve essere allineato ad una pagina di memoria (**??**).

Il segmento è attaccato in lettura se il flag `SHM_RDONLY` è presente, altrimenti è attaccato sia in lettura che scrittura.

Altri flag possono essere (sono tutti specifici per Linux):

- `SHM_REMAP`: si vuole rimpiazzare un segmento esistente rimappandolo nell'intervallo [`shmaddr`; `shmaddr + size`].
- `SHM_EXEC`: consente di eseguire il contenuto del segmento (necessari permessi di esecuzione)

Il valore di ritorno della `shmat` è un puntatore alla zona di memoria attaccata, in caso di errore viene ritornato `(void *) -1` e viene settata la variabile `errno` ai seguenti valori:

- `EACCESS`: il processo non ha i permessi necessari per attaccare la memoria
- `EIDRM`: `shmid` punta a un identificatore rimosso

- `EINVAL`: i valori di `shmid` o `shmaddr` non sono validi o non è stato possibile attaccare il segmento all'indirizzo specificato oppure `SHM_REMAP` è stato specificato e `shmaddr` punta a `null`.

Per staccare segmenti si usa la funzione `shmdt` con la seguente sintassi:

```
int shmdt(void *shmaddr)
```

Il parametro da passare è di immediata intuizione.

La funzione `shmdt` ritorna 0 in caso di successo, -1 se qualcosa va storto; in tal caso `errno` viene settata a `EINVAL` per indicare che non è stato rintracciato nessun segmento di memoria.

Gestione La funzione

```
int shmctl (int shmid, int cmd, struct shmid_ds *buffer)
```

esegue le operazioni di controllo specificate in `cmd`, quali:

- `IPC_RMID`: cancella il segmento
- `IPC_STAT`: ritorna informazioni sulla zona di memoria nella struct `*buffer`
- `IPC_SET`: modifica un sottoinsieme dei campi della struct (UID, GID, permessi)
- `SHM_LOCK`: impedisce che il segmento venga swappato o paginato
- `SHM_UNLOCK`: ripristina il normale utilizzo della zona di memoria

Ricordare che `shmid` è il valore ritornato dalla funzione `shmget`.

La struct `shmid_ds` è così composta:

```
struct shmid_ds {
    struct ipc_perm shm_perm;
    size_t shm_segsz;
    __time_t shm_atime;
    __time_t shm_dtime;
    __time_t shm_ctime;
    __pid_t shm_cpid;
    __pid_t shm_lpid;
    shmatt_t shm_nattch;
}
```

informazioni che rispettivamente indicano:

- le operazioni permesse
- misura del segmento in byte
- orario dell'ultima `shmat()`
- orario dell'ultima `shmdt()`
- orario dell'ultima modifica da parte di `shmctl()`
- PID del processo che ha creato il segmento
- PID dell'ultima `shmop`
- numero corrente di collegamenti

3.5.6 Semafori e sincronizzazione

Introduzione ai semafori I semafori sono un meccanismo di IPC avanzato che consente di sincronizzare l'accesso ad una zona di memoria condivisa tra due o più processi. Possono essere di due tipologie:

- Binari*(mutex)*: valgono solo 1 o 0.
- Interi: assumono qualsiasi valore intero e tornano particolarmente utili quando si vuole limitare il numero di accessi ad una risorsa.

Funzionamento Che il semaforo sia biario o intero il protocollo usato per accedere alle risorse condivise è lo stesso:

- `sem>0`: la risorsa è utilizzabile (si decrementa il valore del semaforo.
- `sem==0`: il processo entra nello stato di *sleep* fino a quando il semaforo non torna a valori positivi.

Quando l'uso della risorsa è terminato, il valore del semaforo viene incrementato.

La verifica dei valori di un semaforo **deve** essere atomica, altrimenti non si potrà mai garantire un corretto svolgimento del meccanismo di sincronizzazione implementato. Per questo motivo la struttura dei semafori è implementata nel kernel. I semafori sono inizializzati a valori sempre positivi. Sebbene il Linux si possano inizializzare a 0, è sconsigliato per motivi di portabilità.

Ricordare che non si può allocare un singolo semaforo ma se ne deve allocare un array.

Funzioni relative La funzione `semget` consente di ottenere il set di semafori indentificato con il parametro relativo:

```
int semget(key_t key, int nsems, int semflg);
```

I parametri `key` e `semflg` assumono gli stessi valori dei corrispondenti parametri delle funzioni relative alle code di messaggi e alla memoria condivisa, mentre `nsems` è il numero di semafori identificati da `semid`, ovvero il numero di semafori contenuti nel vettore.

La funzione:

```
int semctl(int semid, int semnum, int cmd, ... );
```

esegue operazioni di controllo (specificate in `cmd`), sull'insieme di semafori specificato da `semid` o sul `semnum`-esimo semaforo dell'insieme. Questa funzione può accettare opzionalmente il quarto parametro `semun (args)` così costruito:

```
union semun {
    int val;
    struct semid_ds* buffer;
    unsigned short *array;
    struct seminfo *__buf;
};
```

Le operazioni eseguibili tramite `cmd` sono:

- `IPC_RMID`: rimuove il set di semafori

- `IPC_SET`: modifica il set
- `IPC_STAT`: statistiche sul set
- `GETVAL`: legge il valore del semaforo `semnum` in `args.val`
- `GETALL`: legge tutti i valori in `args.array`
- `SETVAL`: assegna il valore del semaforo `semnum` in `args.val`
- `SETALL`: assegna tutti i semafori con valori in `args.array`
- `GETPID`: valore del PID dell'ultimo processo che ha eseguito operazioni
- `GETNCNT`: numero di processi in attesa che un semaforo aumenti
- `GETZCNT`: numero di processi in attesa che un semaforo diventi 0

Il parametro `buffer` è un puntatore ad una struttura `semid_ds` così composta:

```
struct semid_ds {
    struct ipc_perm sem_perm;
    time_t sem_otime;
    time_t sem_ctime;
    unsigned short sem_nsems;
};
```

e contiene rispettivamente:

- operazioni permesse
- ora dell'ultimo `semop`
- orario dell'ultima modifica
- numero di semafori

Ricordare che la struttura `semun` va definita nel codice del processo che chiama la `semctl()`.

Per eseguire operazioni di incremento/decremento del valore di un semaforo si usa la funzione:

```
int semop(int semid, struct sembuf* sops, unsigned nsops);
```

applica l'insieme `sops` di operazioni all'insieme di semafori `semid`. Il parametro `nsops` contiene il numero di operazioni eseguite.

Le operazioni (contenute in un array opportunamente allocato) sono descritte nella `struct sembuf`:

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

La struttura contiene rispettivamente:

- semaforo su cui l'operazione viene eseguita
- l'operazione da eseguire
- modalità di esecuzione dell'operazione

I possibili valori si `sem_op` sono:

- `sem_op` < 0: equivale a **P**. Si blocca se il risultato finale di `sem_val` è minore di 0
- `sem_op` $= 0$: rimane in attesa che il semaforo diventi 0
- `sem_op` > 0: equivale a **V**. Incrementa il semaforo della quantità `sem_op`

I valori di `sem_flg` sono soltanto 2: `IPC_NOWAIT` per realizzare **P** e **V** non bloccanti (utili per meccanismi di polling), `SEM_UNDO` per ripristinare un valore precedente del semaforo (nel caso di terminazioni impreviste).

Ricordare che le operazioni inserite in una chiamata di `semop()` vengono eseguite in modo **atomico**: se una delle operazioni specificate non può essere eseguita, il comportamento della syscall dipende dal flag `IPC_NOWAIT`:

- se è settato, `semop` fallisce e ritorna -1
- se non è settato, il processo viene bloccato

4 Credits

Davide Bianchi (mail: davideb1912@gmail.com)
Matteo Danzi