Appunti presi da Alessandro Fossari durante l’anno accademico 23/24.

Potete contattarmi presso [a.fossari@studenti.unipi.it](mailto:a.fossari@studenti.unipi.it) in caso troviate qualsiasi errore.

UTENTI

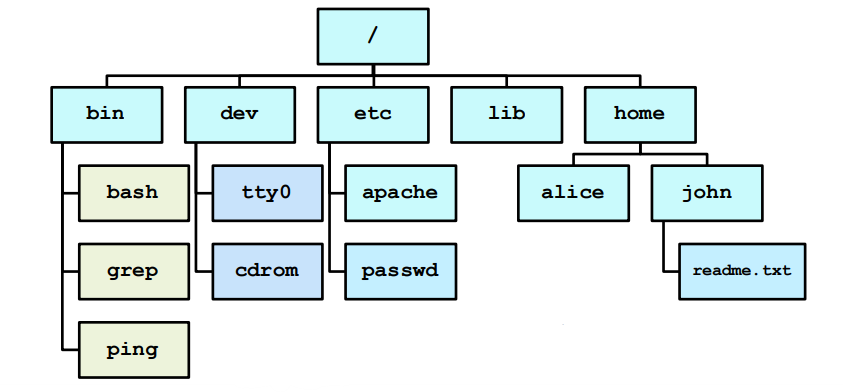
UNIX è un Sistema Operativo (SO) multiutente sul quale possono quindi operare più individui. L’utente root è l’amministratore di sistema e gli è concesso eseguire qualsiasi operazione mentre gli altri utenti mantengono privilegi limitati.

Solitamente, si crea un account utente per l’utilizzo abituale del sistema e si utilizza root solo quando necessario.

FILE SYSTEM

Il file system consente l’accesso a tutte le risorse del sistema. Path importanti sono:

1. “/”: directory principale.
2. “/home”: contiene le varie home directory degli utenti.
3. “/bin”: contiene i programmi del sistema.
4. “/etc”: contiene i file di configurazione.
5. “/media”: rende accessibili i supporti rimovibili.

Ogni documento, cartella o dispositivo è accessibile dal file system come file: Per accedere ad una risorsa del file system si utilizza il suo path, il suo percorso sull’albero. Un path può essere:

1. Assoluto: viene introdotto da ***/*** ed esprime l’intero percorso partendo dalla radice.
2. Relativo: non è introdotto da alcun carattere speciale ed esprime il percorso partendo dalla directory corrente.

Altri caratteri speciali possono essere utilizzati per dare un significato ad un path:

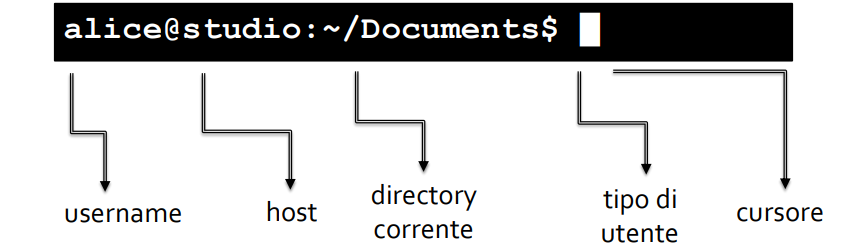
1. ***~*** : indica la nostra home directory.
2. ***.*** : indica la directory corrente.
3. ***..*** : indica la directory padre.

SHELL

Si tratta di un processo interprete di comandi che consente all’utente di interagire con il SO tramite richieste e servizi. Mentre una “shell grafica” (Graphical User Interface, GUI) è intuitiva da utilizzare una “shell testuale” (Command Line Interface, CLI) consente all’utente di esprimersi più efficacemente se si possiede una buona conoscenza dei comandi.

Una CLI, ripetutamente, mostra un promt (una certa stringa), legge un comando da tastiera e lo esegue dopo che l’utente ha premuto uno tra “invio”, “enter” o “return”. In risposta all’esecuzione, se il suo completamento non è possibile allora stampa un messaggio di errore, altrimenti stampa l’output del comando se questo lo prevede.

L’output dei comandi viene stampato di default sullo “standard output”, ovvero quello del terminale.

UNIX supporta diverse shell testuali che differiscono per aspetto e per funzionalità più avanzate ma noi ci limiteremo a bash. Il prompt bash ha aspetto seguente: Sul prompt ***$*** indica che l’utente è di tipo normale, ***~*** indica che si tratta di root.

COMANDI DI BASE

I comandi sulla shell possono essere seguiti da opzioni introdotte come ***-c arg***, dove “c” è uno dei caratteri che rappresentano un’opzione relativa al comando e “arg” il valore argomento.

Un utente può da terminale:

1. Auto completare comandi e nomi di directory con ***tab***.
2. Scorrere la cronologia dei comandi grazie alle ***frecce su/giù***.
3. Ricercare nella cronologia tramite ***Ctrl+R***.
4. Emulare terminali tramite ***Ctrl+Alt+T***.
5. Aprire terminali virtuali con ***Ctrl+Alt+Fi***, con “Fi” scelto tra F1 e F12.

I metacaratteri sono utilizzati per rappresentare più caratteri in un argomento di comando:

1. ***\**** : sostituisce una serie di zero o più caratteri.
2. ***?*** : sostituisce un singolo carattere.
3. ***[a,*** … ***b]*** e ***[a - b]*** sostituiscono un carattere con un altro nell’insieme specificato.

All’avvio è possibile selezionare uno degli utenti disponibili utilizzando le relative credenziali “username” e “password”. Una volta loggati, da terminale si può utilizzare ***logout*** o ***Ctrl+D*** per uscire dalla sessione.

Tramite il comando ***shutdown*** è possibile arrestare (***-h now***) o riavviare il sistema (***-r now***).

Il comando change directory ***cd path*** consente di passare da una directory ad un’altra esprimendone il path, esprimibile come visto in precedenza.

Il comando Print Working Directory ***pwd*** ***path*** stampa il percorso assoluto della directory corrente.

Il comando list ***ls path*** consente di elencare il contenuto della directory specificata. Possono anche essere specificati più percorsi. Nell’output, file e cartelle vengono stampati con colori diversi. Opzioni supportate dal comando sono long (***-l***), che visualizza i dettagli relativi ad ogni risultato ed all (***-a***) che mostra anche i file nascosti.

Il comando manuale ***man n comando*** consente di stampare una descrizione esaustiva del comando, tra cui sintassi, opzioni e messaggi di errore. Il carattere “n” è un numero utilizzato per risolvere ambiguità tra nomi di comandi. Non funziona solo per gli altri comandi, ma può anche essere applicato a sé stesso, alle funzionalità del kernel, alle librerie C o ai file di configurazione.

I comandi ***whatis*** e ***apropos*** servono rispettivamente a visualizzare una breve descrizione di una pagina del manuale e a ricercare parole in nomi di descrizioni.

COMANDI SU FILE

I seguenti comandi sono tutti eseguibili da tutti gli utenti.

***mkdir*** ***dir***: crea la directory dir nel path corrente.

***rmdir*** ***dir***: rimuove dir solo se questa è vuota.

***cp src dst***: se src e dest sono file allora il contenuto di src viene copiato in dst, altrimenti se dst è una directory copia src al suo interno.

***cp src1 … scrN dir***: copia una serie di file o directory src in una directory dst.

***mv src dst***: rinomina un file o una directory.

***mv src1 … scrN dir***: sposta una serie di file o directory src in una directory dst.

***touch file***: aggiorna il timestamp di accesso a file. Se questo non esiste viene creato.

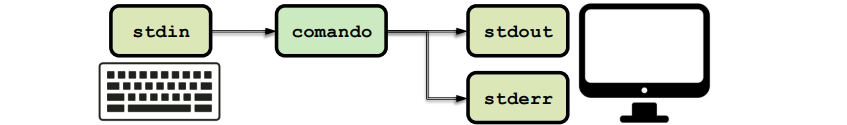
***cat file1 … fileN***: concatena il contenuto di una serie di file e stampa il tutto in standard output.

***rm file1 … fileN***: rimuove una serie di file e directory solo se seguito dall’opzione ***-r***.

***less file***: utilizzato per visualizzare file poco alla volta.

***head file*** e ***tail file*** vengono utilizzati per visualizzare rispettivamente la prima e l’ultima parte di un file. Il numero dei byte da mostrare viene specificato dall’opzione ***-c*** o quello delle righe da ***-n*** (di default 10).

REDIREZIONE I/O

I processi possiedono uno standard input stdin (input da tastiera), uno standard output stdout (output su schermo) e uno standard error stderr (messaggi su schermo). 

Attraverso comandi è possibile deviare l’output di un comando verso un file o acquisire l’input da un file.

Attraverso ***>***, ***2>*** e ***&>*** inviamo rispettivamente lo stdout, lo stderr ed entrambi ad un file in maniera distruttiva (sovrascrivendo il file). Possiamo fare lo stesso in append tramite ***>>***, ***2>>*** e ***&>>***.

Si possono inviare i due output su file diversi concatenando la sintassi vista sopra.

Attraverso ***<*** è possibile recuperare l’output da un file. Può essere utilizzato con ***>***.

Possiamo creare un pipe che colleghi l’output di un comando all’input di un altro attraverso ***|*** (non è uno slash giuro è che è in corsivoe).

SUDO

Il comando switch user ***su utente*** serve per accedere al terminale di un altro utente. Se non si specifica l’utente si accede a root. Viene ovviamente richiesta la password dell’utente a cui si intende accedere.

Il comando ***sudo -u utente comando*** serve a lanciare un comando come un altro user “utente”. Se non si specifica nessun attributo e nessun utente si usa root. Viene richiesta la password dell’utente e il chiamante deve far parte del gruppo dei “sudoers”.

UTENTI E GRUPPI

Un utente è identificato in UNIX da uno username e da uno User ID (UID) numerico.

Un gruppo è identificato in UNIX da un group name e da un Group ID (GID) numerico.

Ogni utente deve appartenere almeno ad un gruppo (primary group).

Possiamo modificare utenti e gruppi attraverso l’uso di comandi diversi.

1. ***passwd***: consente di cambiare la password di un utente (sfrutta il permesso SUID).
2. ***id nome:*** stampa informazioni quali UID, primary group ed altri gruppi di un utente “nome”.
3. ***groups nome***: visualizza i gruppi dell’utente “nome” selezionato o di quello corrente se non viene specificato.
4. ***$ adduser nome***: crea un nuovo utente “nome”.
5. ***$ deluser nome***: rimuove un utente “nome”.
6. ***su*** e ***sudo***: come visti in precedenza.
7. ***$ addgroup grp***: crea un nuovo gruppo “grp”.
8. ***$ delgroup grp***: rimuove un gruppo “grp”.
9. ***gpasswd***: viene utilizzato per aggiungere un utente utilizzando l’opzione ***-a nome grp***, per rimuoverlo con ***-d nome grp***, per definire tutti gli utenti come ***-M nome1*** … ***nomeN grp*** o per definirne tutti gli amministratori con ***-A nome1*** … ***nomeN grp***. Solo root può modificare gli amministratori, ma questi ultimi possono anche gestire gli utenti che ne fanno parte. Il comando può essere infine utilizzato per settare la password di un gruppo senza l’utilizzo di opzioni (quindi con ***gpasswd grp***) o per rimuoverla con ***-r grp***. Solo i membri del gruppo possono gestire la password se questa non è impostata.
10. ***newgrp grp***: viene utilizzato dagli utenti per specificare il proprio gruppo primario per la sessione di login corrente. Può essere utilizzato da utenti non facenti parte di “grp” per ottenerne i privilegi quando la password è impostata. Le password dei gruppi sono generalmente poco sicure in quanto condivise tra più utenti.

I file di configurazione riguardo gli utenti si trovano all’interno di “/etc/passwd” e le informazioni per loro sensibili sono invece mantenute in “etc/shadow”. I gruppi, a loro volta, mantengono queste informazoni all’interno di “/etc/group” e di “/etc/gshadow”.

Il file /passwd è apribile tramite il comando ***vipw*** ed al suo interno troviamo su ogni riga le informazioni pubbliche di ciascun utente, mostrate come segue: Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamenteTroviamo in ordine username, password cifrata (appare “x” se si trova nel file /shadow), UID, GID, dei dati personali, il path assoluto della cartella home e lo shell da utilizzare. Possiamo impedire ad un utente di loggarsi settando la shell a valori come “/bin/nologin” o “/bin/false”.

Il file /shadow può essere aperto tramite comando ***vipw -s*** ed al suo interno troviamo righe del genere: Immagine che contiene testo, Carattere, linea, schermata

Descrizione generata automaticamenteIn ordine troviamo l’username, un numero per rappresentare una funzione di hashing (hash), un numero casuale (salt), la password utente criptata come hash(salt + password), informazioni riguardo ultima modifica ed intervallo di validità della password (espresso come un’età minima e massima), il periodo di avviso in giorni previo la scadenza della password, un periodo di inattività nel quale la password viene accettata anche dopo la scadenza ed un campo riservato per utilizzi futuri.

Il file “/group” può essere aperto tramite comando ***vigr***. Al suo interno troveremo righe come la seguente: Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamenteIn ordine troviamo nome del gruppo, password gestita alla stessa maniera degli utenti, GID ed una lista dei partecipanti. In questa lista non sono presenti gli utenti per i quali il gruppo è primario, in quanto già memorizzato in “/passwd”.

Il file “/gshadow” può essere aperto tramite ***vigr -s***. Le righe che contiene sono come le seguenti: Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamenteIn ordine abbiamo il nome del gruppo, la pasword criptata (***\**** oppure ***!*** se non impostata), il GID, una lista degli amministratori ed una degli utenti (amministratori compresi).

PERMESSI DI ACCESSO

Grazie al meccanismo dei permessi possiamo gestire l’accesso da parte degli utenti al file system.

Per ogni file o directory vengono infatti definiti un utente proprietario (owner) ed un gruppo proprietario (group owner), il che implica l’esistenza di tre classi di utenti: l’utente proprietario del file, gli utenti del gruppo proprietario e tutti gli altri utenti.

A ciascuna delle categorie dette sopra, possiamo associare dei permessi diversi.

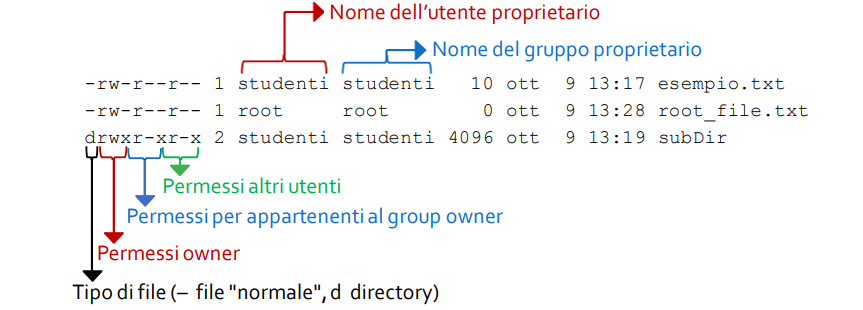
I permessi di accesso possono essere in lettura (r), in scrittura (w) o in esecuzione (x).

1. r: consente di leggere il contenuto di un file o di una directory. Se disabilitato su una directory allora non consente l’esecuzione di ***ls***.
2. w: consente di modificare il contenuto di un file e di crearne, rinominarle o cancellarne altri in una directory. Avendo i permessi di scrittura su un file non sarà possibile cancellarlo ma saranno necessari quelli di directory.
3. x: consente di eseguire un file o di attraversare una directory. Se disabilitato su una directory allora non consente l’esecuzione di ***cd***.

I permessi vengono rappresentati come stringhe di caratteri ***rwx***per ogni livello inserendo i soli caratteri permessi oppure attraverso tre cifre ottali (una per ogni classe di permessi):

1. Per concedere “r” la cifra vale 4.
2. Per concedere “w” la cifra vale 2.
3. Per concedere “x” la cifra vale 1.

Per concedere permessi diversi sommiamo le cifre.

I permessi per ciascun file o directory sono visibili con ***ls -l***. In output avremo qualcosa del genere: Il carattere “d” che figura sopra nei permessi sta a significare che ci si sta riferendo ad una directory.

Il comando ***chmod nnn file*** consente di dare come permessi a “file” le tre cifre ottali o la stringa di caratteri “nnn”. Attraverso l’opzione ***-R*** se il target è una directory è poi possibile applicare i permessi ricorsivamente a file e directory figlie. La sintassi rispetta la forma ***chmod who how which***:

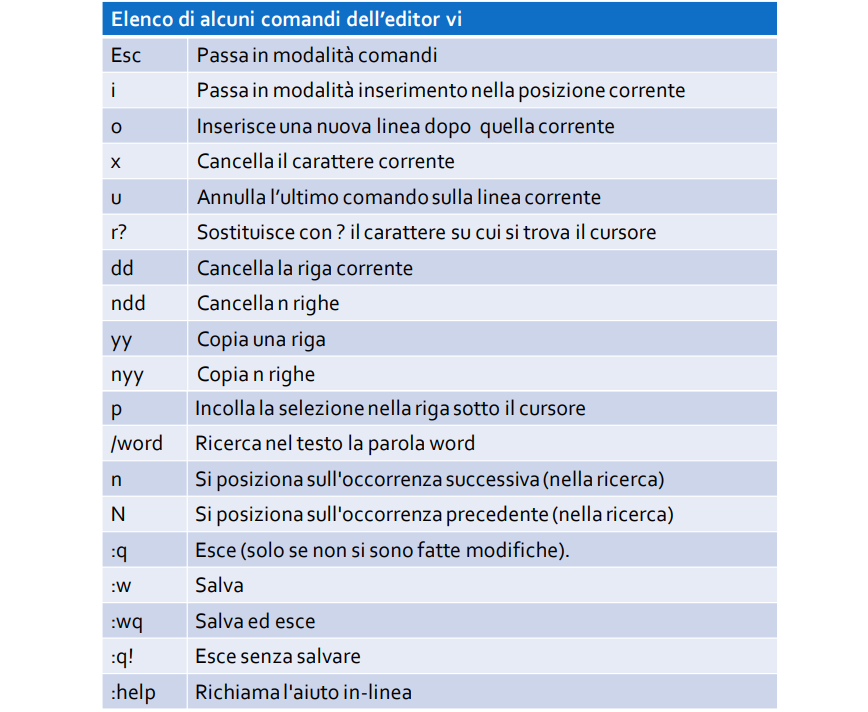
1. ***who*** indica la classe di utenti a cui modificare i permessi: ***u***, ***g***, ***o*** per owner, gruppo o altri.
2. ***how*** indica l’azione di modifica dei permessi: ***+***, ***-***, ***=*** per aggiungerne, toglierne o per assegnarne.
3. ***which*** indica il file target.

Permessi aggiuntivi sono ***SUID*** e ***SGID*** che consentono ad un processo in esecuzione di acquisire i privilegi rispettivamente del proprietario e del group owner del file, quando di default sarebbero quelli di chi lo esegue. Il ***SUID*** e ***SGID*** vengono rappresentati in simboli dalla lettera ***s*** andando a sostituire la ***x*** in corrispondenza dei privilegi del file owner e group owner. Per la rappresentazione ottale utilizziamo un’ulteriore cifra nella quale SUID = 4 e SGID = 2.

Possiamo modificare owner e group owner di “file” con i comandi ***$*** ***chown nome file*** e ***chgrp grp file***. Mentre per il primo sono sempre necessarie le autorizzazioni di root, per il secondo basta essere all’interno del gruppo “grp”.

EDITOR

Il sistema ci offre diversi editor di testo quali vi, emacs, vim o nano, ma anche di grafici come gedit.

I comandi vim sono i seguenti: 

RICERCA

Il comando ***find*** è uno strumento molto potente per la ricerca di file applicando una sintassi abbastanza complessa.

Facendola breve, consente di effettuare ricerche combinando test sulle proprietà dei file, ovvero nome, tipo, owner e group owner, permessi e timestamp.

È possibile seguire dei comandi, dette actions, sui risultati della ricerca.

Possiamo utilizzare il comando come ***find [path1*** … ***pathN] [expr]*** dove il primo argomento è una lista di path nei quali cercare, mentre “expr” è un’espressione che descrive come effettuare la ricerca e quali azioni effettuare.

Le espressioni sono composte da una sequenza di elementi:

1. Test: valuta la proprietà di un file e ritorna true o false.
2. Azioni: da effettuare sui file trovati, come l’esecuzione di un comando. Restituisce true se ha successo.
3. Opzioni globali: influenzano esecuzione di test ed azioni. Ritornano sempre true.
4. Opzioni posizionali: influenzano solo test ed azioni che le seguono. Ritornano sempre true.

Gli elementi di un’espressione sono collegati da operatori come opzioni: ***-o*** indica un OR, ***-a*** un AND e ***!*** un NOT. L’AND viene utilizzato di default se nessuno è specificato.

Il test può essere effettuato tramite opzioni seguenti:

1. ***-name pattern***: ricerca basata sui nomi. In “pattern” possono essere utilizzati i metacaratteri, scrivendoli tra apici per evitare che si espandano.
2. ***-type [dfl]***: ricerca per tipo. I tre caratteri stanno per “directory”, “file” e “link”.
3. ***-size [+-] n [ckMG]***: ricerca per dimensione. I segni rappresentano se la dimensione deve essere maggiore o minore di quanto specificato, “n” rappresenta il quantitativo di unità e uno tra gli ultimi caratteri indica l’unità di misura (byte, Kb, Mb, Gb).
4. ***-user nome***: ricerca per proprietario, specificato come nome o UID.
5. ***-group grp***: ricerca per group owner, specificato tramite nome o GID.
6. ***-perm [-/] mode***: ricerca in funzione dei permessi, espressi in maniera ottale o simbolica. Con ***–*** chiediamo che almeno i permessi specificati debbano essere presenti, con ***/*** che almeno uno dei permessi specificati debba essere presente e omettendo il campo chiediamo che i permessi siano esattamente quelli specificati.

Le azioni possibili sono:

1. ***-delete***: vengono eliminati i file trovati. Ritorna true in caso di successo. Bisogna prestare attenzione alla posizione dell’opzione in quanto se fosse posta prima dei test eliminerebbe tutti i file.
2. ***-exec comando ;*** : esegue un comando sui risultati della ricerca. Tutti gli argomenti specificati tra “comando” e “ ; “ vengono considerati come fossero del comando stesso. La stringa ***{}*** è utilizzata per indicare il nome del file attualmente processato. Per evitare che la shell fraintenda “ ***;*** “ va utilizzata come “ ***\;*** “. Per svolgere il comando dal percorso dei file processati si utilizza ***-execdir***.

In alternativa a ***find*** abbiamo il comando ***locate [opzioni] file1*** … ***fileN***. A differenza del primo, questo comando sfrutta un database aggiornato periodicamente dal sistema (l’aggiornamento può essere forzato tramite ***$ updatedb***).

La ***find*** è migliore di ***locate*** anche se meno semplice da utilizzare perché si tratta di un comando standard UNIX, dà sempre risultati aggiornati e permette la definizione di test ed azioni.

Per effettuare ricerche interne a file utilizziamo invece ***grep*** (General Regular Expression Print). Il comando consente di cercare in uno o più file di testo le righe che corrispondono ad espressioni regolari o stringhe letterali come ***grep [opzioni] [-e] modello [-e modello2*** … ***-e modelloN] [file1*** … ***fileN]***, dove con “modello” si intende una stringa o un’espressione regolare. Volendo utilizzare più di un modello bisogna utilizzare ***-e*** prima di ciascun modello, primo incluso. Altre opzioni sono:

1. ***-i***: non case sensitive.
2. ***-v***: mostra le linee che non contengono il modello.
3. ***-n***: mostra l’indice di linea.
4. ***-c***: riporta solo il conteggio delle linee trovate.
5. ***-w***. trova solo parole intere.
6. ***-x***. trova solo linee intere.

Il comando ***grep*** supporta modelli di stringhe ed espressioni regolari:

1. ***^str***: indica le righe che iniziano per “str”.
2. ***str$***: indica le righe che finiscono per “str”.
3. ***[az]***: esprime un set di valori che un carattere può assumere, nel caso riportato o “a” o “z”. Se tra i due caratteri vi è un ***-*** allora si intendono tutti i valori tra i due caratteri specificati quindi nel nostro caso tutto l’alfabeto minuscolo. Più intervalli possono essere specificati.
4. ***.*** : indica un qualsiasi carattere.
5. ***expr\**** : indica che l’espressione “expr” può essere ripetuta zero o più volte.
6. ***\*** : carattere di escape. Viene utilizzato per riconoscere caratteri che avrebbero altrimenti un significato particolare (come “ . “, “[]” o “\*”).

I simboli ***^*** e ***$*** possono essere combinati: ***^str$*** troverà le righe che contengono solo “str”, mentre ***^$*** tutte le righe vuote.

ARCHIVIAZIONE E COMPRESSIONE

Il comando Tape Archive ***tar*** permette di archiviare ed estrarre raccolte di file e cartelle. Viene utilizzato come ***tar mode [opzioni] [file1*** … ***fileN]***, dove “mode” indica la modalità con la quale opera il comando (creare un archivio o estrarne uno).

Il formato del file creato dipende dalla compressione utilizzata: “.tar” per nessuna compressione e “.tar.gz” o “.tar.bz2” se compresso tramite gzip o bzip2.

Le modalità previste da tar sono le seguenti e possono essere specificate come stringhe dei caratteri a loro corrispondenti:

1. ***A***: aggiungi file tar all’archivio.
2. ***c***: crea un nuovo archivio.
3. ***d***: trova le differenze tra l’archivio ed il file system.
4. ***--delete***: cancella file dall’archivio.
5. ***r***: aggiunge file all’archivio.
6. ***t***: elenca i file di un archivio.
7. ***u:*** aggiungi file all’archivio ma solo se differiscono da una copia già presente.
8. ***x***: estrai file dall’archivio.
9. ***v***: verbose.
10. ***z***: compressione tramite gzip.
11. ***j:*** compressione tramite bzip2.
12. ***f***: consente di specificare il nome dell’archivio.

Se si devono comprimere file o archivi creati con tar è possibile utilizzare ***gzip file1*** … ***fileN*** per comprimerli in file che portano lo stesso nome ma estensione “.gz” (i file non compressi vengono eliminati). Possiamo invece estrarre i file tramite ***gunzip file1.gz*** … ***fileN.gz***. Allo stesso modo fanno ***bzip2*** e ***bunzip2*** ma applicando algoritmo bzip2.

PROCESSI IN UNIX

I processi UNIX mantengono spazi di indirizzamento separati per dati e codice separati, in modo che il primo possa essere privato ed il secondo condivisibile.

L’assegnamento della CPU ai processi avviene in funzione di divisione di tempo (tipo RR) e per questo motivo attraversano vari stati:Immagine che contiene testo, cerchio, schermata, diagramma

Descrizione generata automaticamenteUno stato mai visto è lo “zombie”: questo stato viene assunto da processi che sono pronti a terminare ma, essendo monitorati da altri processi, sono tenuti in vita in modo che possano dare loro dei valori di ritorno.

I descrittori di processo in UNIX sono divisi in due strutture dati distinte: la Process Structure (PS) che contiene le informazioni indispensabili e la user Structure (US) che contiene quelle necessarie quando il processo si trova in memoria principale. La PS contiene un campo che punti alla relativa US. Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Si possono visualizzare i processi in esecuzione grazie al comando ***ps*** che scatta una foto all’esecuzione in quell’istante. Le sue principali opzioni servono a visualizzare:

1. ***-u nome***: i processi di “nome”.
2. ***u***: un formato di informazioni utile all’analisi di utilizzo delle risorse.
3. ***a***: i processi di tutti gli utenti.
4. ***x***: anche i processi non generati da terminale.
5. ***o campo1 … campoN***: solo i campi specificati.
6. ***-O campo1 … campoN***: i campi specificati più altri di default.

In output a video appariranno anche gli stati dei processi, principalmente S(sleep), T(bloccato), R(running) e Z(zombie).

SYSTEM CALL PER PPROCESSI

Ogni processo in UNIX è in grado di generare dei processi figli attraverso una chiamata di sistema, la ***fork***. Il processo creato avrà uno spazio dati separato ma continuerà a condividere il codice del padre.

***pid\_t fork();***

La funzione non richiede parametri e restituisce coma vaore un tipo ***pid\_t***, un tipo standard per rappresentare un identidicativo di processo (di fatto un intero). Restituisce al processo padre l’id del figlio e al figlio il valore “0”. Il figlio continuerà a condividere il codice con il padre ed erediterà da quest’ultimo una copia di tutte le sue aree dati globali (spazio di indirizzamento dei dati, stack, heap US) e persino del Program Counter (il figlio riparte la sua esecuzione da dove era rimasto il padre). Per questo motivo padre e figlio sono praticamente indistinguibili se non fosse per il valore di ritorno della ***fork***.

I pid dei processi possono essere estrapolati grazie alle funzioni ***pid\_t getpid()*** e ***pid\_t getppid()*** che ritornano rispettivamente il pid del processo e del genitore.

La terminazione dei processi può poi essere involontaria, se a causa di azioni illegali individuate dalle eccezioni o a causa della ricezione di segnali, o volontaria, se conseguente al raggiungimento della fine del codice o alla chiamata della system call ***exit***.

***void exit(int status);***

Questa system call viene utilizzata per permettere di comunicare al processo padre lo stato di terminazione del processo. Non prevede valori di ritorno nel figlio che termina la sua esecuzione.

Lo stato di terminazione di un processo figlio può essere letto dal padre tramite la funzione ***wait***.

***pid\_t wait(int \*status);***

La system call forza il chiamante a sospendersi in attesa che uno qualsiasi dei figli termini (sono questi i processi zombie) in modo da ricevere il pid di quest’ultimo e le relative informazioni di terminazione all’indirizzo della variabile “status”. Se il processo non possiede figli allora la funzione ritorna un valore negativo, interpretato come un errore.

Il valore inserito in “status” dalla ***wait*** codifica diverse informazioni di terminazione. In particolare, se il suo byte meno significativo è “0” allora la terminazione è stata volontaria e nel byte più significativo potrà essere letto lo stato di terminazione. Per gestire lo stato di ritorno, la libreria ***sys/wait.h*** mette a disposizione le macro ***WIFEXITED(status)***, che ritorna true se la terminazione è stata volontaria, e ***WEXITSTATUS(status)***, che ritorna lo stato di terminazione.

Abbiamo detto in precedenza come processi padre e figlio condividano il codice. Un processo può sostituire il programma che sta eseguendo utilizzando una system call della famiglia di funzioni ***exec***.

***int execl (char\* path, char\* arg0,*** … ***char\* argN, (char\*)0);***

La funzione ***execl*** nello specifico esegue un comando memorizzato al percorso “path” e con programma di nome “arg0” e considera i restanti argomenti come argomenti del comando stesso. La lista di argomenti termina con un puntatore nullo. La chiamata non ritorna nulla quando ha successo ed il codice successivo alla chiamata viene eseguito solo in caso di fallimento.

INTERAZIONE TRA PROCESSI

Come già detto i processi UNIX possiedono uno spazio di indirizzamento privato che non consente la condivisione di variabili. L’unica forma di interazione tra processi diviene la cooperazione in forma di scambio di segnali temporali (sincronizzazione) o di messaggi (comunicazione) in ogni caso consentite dal kernel tramite system call.

I segnali sono il meccanismo messo a disposizione da UNIX per effettuare sincronizzazione tra processi in maniera asincrona. Sono detti “interrupt software”.

Alla ricezione di un segnale:

1. Viene eseguita una routine di gestione dell’errore (handler) definito dal programmatore.
2. Viene eseguito un handler definito dal SO (default handler).
3. Il processo lo ignora.

Nei primi due casi l’esecuzione dell’handler è asincrona rispetto a quella del processo, che viene interrotto perché l’handler possa eseguire. Se il processo non termina, quando tornerà in esecuzione riprenderà dall’istruzione subito successiva al segnale.

I segnali definiti su UNIX dipendono dalla versione e sono contenuti all’interno di ***signal.h***. Ciascun segnale è identificato da un intero e da un nome simbolico. Esempi di segnali sono i seguenti: Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

Per definire un handler di segnale definiamo prima il tipo ***sighandler\_t*** come puntatore ad una funzione che prenda in ingresso un tipo ***int*** (conterrà alla chiamata il codice del segnale ricevuto) e non ritorni nulla:

***typedef void (\*sighandler\_t) (int);***

Possiamo poi associare al segnale di codice “sig” la funzione “handler” che andremo a definire per la gestione del segnale attraverso la funzione ***signal***:

***sighandler\_t signal(int sig, sighandler\_t handler);***

La funzione signal restituisce un puntatore al vecchio handler di gestione (se ne era già stato definito uno) e ***SIG\_ERR*** in caso di errore. Il parametro “handler”, oltre che la funzione associata al segnale, può anche essere settato a ***SIG\_IGN*** per ignorare il segnale o a ***SIG\_DFL*** per ripristinare l’azione default.

I processi figli ereditano dal padre le informazioni relative alle gestioni dei segnali ma la ***exec*** non in grado di mantenere le associazioni della ***signal*** anche se i segnali ignorati continueranno ad esserlo.

Perché un processo possa inviare un segnale ad un altro si utilizza la funzione ***kill***:

***int kill(pit\_t pid, int sig);***

La system call ritorna “0” in caso di successo ed invia il segnale di codice “sig” a:

1. Se “pid” > 0 al processo con pid == “pid”.
2. Se “pid” == 0 ai processi nello stesso gruppo del chiamante.
3. Se “pid” == -1 a tutti i processi ai quali il chiamante può inviare segnali.
4. Se “pid” < -1 ai processi il cui gruppo è “-pid”.

Altre funzioni relative ai segnali sono:

1. ***int pause()***: il processo si blocca fino alla ricezione di un segnale e ritorna “-1” se il gestore non termina l’esecuzione del processo.
2. ***unsigned int sleep(unsigned int sec)***: il processo si blocca per “sec” secondi (si riceve il segnale ***SIGALARM***) o fin quando non arriva un segnale non ignorabile. Ritorna il tempo residuo una volta svegliato (quindi “0” se passa tutto il tempo).
3. ***unsigned int alarm(unsigned int sec)***: il processo riceverà un segnale ***SIGALARM*** dopo “sec” secondi. Eventuali allarmi invocati precedentemente vengono cancellati e se “sec” == 0 viene semplicemente eliminato l’allarme precedente. Ritorna “0” se non esisteva alcun allarme in programma altrimenti il suo tempo residuo.

L’invio di segnali verso i processi può anche essere fatto da terminale attraverso ***kill [opzioni] pid1*** … ***pidN***, il quale invia di default il segnale ***SIGTERM*** ai processi specificati, altrimenti rispetta le opzioni:

1. ***-l***: mostra la lista dei segnali disponibili.
2. ***-segnale***: invia “segnale” ai processi specificati.

L’utente root può inviare segnali a tutti i processi mentre un normale utente solo a quelli di cui è owner.

GERARCHIE DI PROCESSI

I sistemi UNIX prevedono un init system: si parla di un processo mandato in esecuzione dal kernel durante il boot. E’ il primo processo messo in esecuzione e ha pid = 1. Tutti gli altri processi discendono da esso. Su UNIX tale processo ha nome ***systemd***. L’albero dei processi che da lui si origina può essere visualizzato tramite comando ***pstree***.

IDENTIFICATORI DI PROCESSI

Oltre che pid e ppid visti in precedenza (identificatore di processo e del padre) in relazione ad un processo esistono altri identificatori che determinano i permessi di un processo:

1. PGID (Process Group ID): relativo ad un gruppo di processi. Ottenibile da ***pid\_t getpgrp()***.
2. RUID (Real User ID): identificatore relativo all’utente che ha mandato il processo in esecuzione. Ottenibile da ***uid\_t getuid()***.
3. RGID (Real Group ID): relativo al gruppo primario dell’utente che ha mandato il processo in esecuzione. Ottenibile da ***uid\_t getgid()***.
4. EUID (Effective User ID): ottenibile da ***uid\_t geteuid()***.
5. EGID (Effective Group ID): ottenibile da ***uid\_t getegid()***.

EUID e EGID possono differire da RUID e RGID se il comando eseguito ha bit ***SUID*** o ***SGID*** attivo. Un utente può inviare segnali ad un altro processo solo se il suo EUID o il suo RUID coincide con il RUID del processo destinatario.

GRUPPI DI PROCESSI

I processi sono organizzati in gruppi. Quando mandato in esecuzione, ad un processo viene assegnato un process group e se questo genera dei figli loro ne prenderanno parte. La ***exec*** preserva il gruppo.

I gruppi consentono di mandare segnali ad una gerarchia di processi e sono alla base del job-control offerto dalla shell.

NICENESS

Lo scheduler UNIX, come già detto, gestisce i processi secondo divisione di tempo. In realtà ciò avviene solo sulle stesse code di processo: ogni processo possiede una certa priorità ed il sistema possiede più code di processi sulle quali avviene la divisione di tempo. Ogni coda ha priorità diversa.

La priorità dei processi non in tempo reale può essere controllata in parte mediante il concetto di “niceness” attraverso la system call ***nice***.

Ad ogni processo viene associato un valore di niceness scelto in [-20, 19] e ad una niceness maggiore corrisponde una priorità minore (un processo “buono” tende a lasciare gli altri in esecuzione).

La niceness può solo essere controllata da root.

Da terminale è possibile utilizzare il comando ***nice -n valore comando*** per eseguire “comando” con un dato valore di niceness. Se questo è già in esecuzione si può utilizzare ***renice valore pid***.

JOB CONTROL

Con questo termine si intende la possibilità di sospendere e riattivare gruppi di processi (job) attraverso la shell mediante comandi. Un job ID distinto viene associato a singoli comandi anche se questi sono messi in piepeline. Attraverso il comando ***jobs*** è possibile visualizzare mediante una tabella specifica tutti i job.

I job possono essere posti in “foreground” o in “background”: nel primo caso il processo ha di fatto il controllo del terminale, dello stdin e dello stdout per poi restituirli al termine (è la modalità con la quale i comandi sono normalmente eseguiti). Un job può essere eseguito in background digidando di seguito al comando il carattere ***&***. Non concedendo la shell al processo il programmatore può continuare ad utilizzarla anche se questo non viene completato.

Un processo in foreground può essere fermato inviando il segnale ***SIGSTP*** (***Ctrl+Z***). Sui job fermati è possibile intervenire tramite ***fg jobid*** e ***bg jobid*** che permettono di riprendere dei job di ID “jobid” rispettivamente in foreground ed i background.

Il comando ***kill*** può anche essere utilizzato sui job sostituendo a ***pid*** il valore ***%jobid***.

Quando il terminale viene chiuso i job in esecuzione ricevono il segnale ***SIGUP*** e vengono di default terminati. Per fare in modo che ciò non accada utilizziamo due tecniche:

1. ***nohup comando***: il job diviene immune a ***SIGHUP***, perde l’accesso allo stdin (in caso di lettura riceve EOF) e lo stdout viene rediretto in un file ***nohup.out***.
2. ***disown %jobid***: il job, già in esecuzione, diviene immune a ***SIGHUP*** in quanto la shell non lo invierà a questo processo alla chiusura. Per evitare errori durante l’esecuzione è opportuno che il job non legga lo stdin e che lo stdout sia rediretto altrove.

MONITOR DI SISTEMA

Il comando ***top*** permette di visualizzare i processi e di effettuare delle operazioni su di essi. Una volta eseguito, il comando metterà in output una lista di processi ordinati per utilizzo di CPU con delle relative informazioni. Sarà possibile inviare segnali e cambiare niceness dei processi lì visibili.

Tra i possibili comandi interattivi abbiamo quelli seguenti, richiamabili attraverso la pressione dei relativi caratteri mentre ***top*** è attivo:

1. ***h***: help.
2. ***d intervallo***: specifica l’intervallo con il quale aggiornare i risultati in ***top***.
3. ***k pid***: invia segnali al processo “pid”.
4. ***n numero***: specifica il numero di processi da visualizzare in ***top***.
5. ***r pid***: renice del processo “pid”.
6. ***u nome***: visualizza i processi dell’utente “nome”.
7. ***q***: quit.

THREAD

Un thread è un flusso di esecuzione indipendente all’interno di un processo. Ad un singolo processo possono essere associati più thread, i quali dovranno condividere risorse e spazio di indirizzi tra di loro.

I thread vengono anche detti processi leggeri perché creazione, distruzione e cambio di contesto tra thread dello stesso processo sono operazioni meno onerose rispetto a quelle nei processi o al cambio di contesto tra thread di processi diversi.

Il difetto dell’uso dei thread sta nella concorrenza fra thread: il codice deve essere scritto in modo da garantire il corretto funzionamento del programma accedendo alle risorse condivise in mutua esclusione.

Linux supporta il concetto di thread come “unità di scheduling” e i processi tradizionali sono visti come thread che non condividono risorse.

Per utilizzare i thread ci viene fornita la librearia ***pthread.h*** dallo standard POSIX, da includere al momento della compilazione con ***gcc [opzioni] nomefile.c -lpthread -std=c99***.

I thread possiedono un ID d tipo ***pthread\_t***, ottenibile al momento dell’esecuzione come ***pthread\_t pthread\_self ()***. Il tipo è “opaco”: non ha senso stamparlo perché utilizzabile solo mediante apposite funzioni (ad esempio, per confrontare due ID è necessaria la ***pthread\_equal(tid1, tid2)***). Nello specifico, Linux possiede una funzione ***pthread\_t gettid()*** per ottenere un ID. Se un processo ha un solo thread allora gli ID dei due coincidono.

L’esecuzione di un programma determina la creazione del primo thread che esegue il codice del main. A sua volta, il thread principale potrà generarne altri come in una gerarchia grazie alla funzione ***create***:

***int pthread\_create(pthread\_t\* t, const pthread\_attr\_t\* attr, void\* (\*start\_routine) (void\*), void\* arg);***

1. t: puntatore ad identificatore di thread nel quale ne verrà scritto l’ID.
2. attr: attributi del thread, nullptr per valori di default.
3. (\*start\_routine)(void\*): puntatore alla funzione che contiene il codice del nuovo thread.
4. arg: puntatore passato come argomento a “start routine”.

La funzione ritorna “0” in assenza di errore, un altro valore altrimenti.

La terminazione di un thread può avvenire in seguito ad una ***exit***:

***void pthread\_exit(void\* retval);***

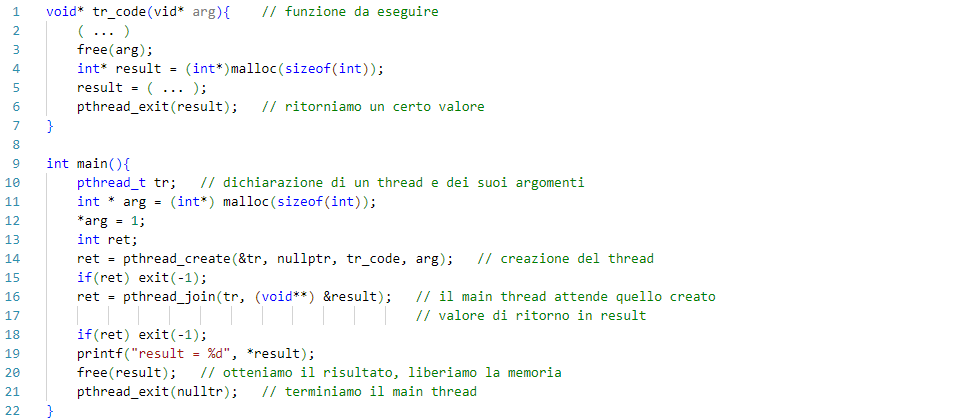
L’esecuzione della funzione inserisce all’indirizzo di “retval” il valore di ritorno del thread terminato, lo termina e libera le risorse in suo possesso. Se un thread possiede dei thread figli se termina a causa di una ***exit*** non li farà terminare, ma se termina per altri motivi allora ciò accade.

Un thread può bloccarsi in attesa della terminazione di un altro attraverso la funzione ***join***:

***int pthread\_join(pthread\_t t, void\*\* retval);***

1. t: thread di cui attendere la terminazione.
2. retval: puntatore al puntatore dove viene salvato l’indirizzo restituito dal thread dopo ***exit***. Se impostato a nullptr viene ignorato.

La funzione ritorna “0” nel caso di successo, altrimenti un codice errore, come nel caso di rischio di deadlock.

Applichiamo quanto visto con del codice nel quale un thread ne crea un secondo e ne legge il risultato: 

MUTUA ESCLUSIONE

Per risolvere i problemi di mutua esclusione ***pthread.h*** ci mette a disposizione l’astrazione come variabile dei mutex. I mutex ci consentono di proteggere l’accesso a variabili condivise fra thread. Il tipo ***pthread\_mutex\_t*** che li rappresenta non è che una struttura dati contenente una variabile relativa allo stato del semaforo e una coda di spospensione dei processi. Un mutex, per definizione, assume come stato “libero” o “occupato”.

Per inizializzare un mutex utilizziamo la ***init***:

***int pthread\_mutex\_init(pthread\_mutex\_t\* M, const pthread\_mutexattr\_t\* mattr);***

1. M: è il puntatore al mutex da inizializzare.
2. mattr: è un puntatore ad una struttura contenente degli attributi di inizializzazione per il mutex. Può essere ignorata se settata a nullptr ed inizializzando il mutex come libero.

Wait e signal sono implementate grazie alle funzioni seguenti:

***int pthread\_mutex\_lock(pthread\_mutex\_t\* M);***

***init pthread\_mutex\_unlock(pthread\_mutex\_t\* M);***

Entrambe ritornano “0” in caso di successo, un codice errore altrimenti.

Se più thread provano ad accedere ad una variabile condivisa tramite una ***lock*** solo il primo che arriva non si bloccherà mentre gli altri dovranno attendere sulla coda del mutex per essere poi sbloccati da una ***unlock***.

Per politiche di accesso più avanzate possiamo utilizzare le variabili di condizione, un metodo di sincronizzazione “diretta” sulle quali i processi si mettono in attesa del verificarsi di una determinata condizione.

Una variabile di condizione viene definita con tipo ***pthread\_cond\_t*** e può essere definita attraverso la funzione ***init***:

***int pthread\_cond\_init(pthread\_cond\_t\* C, pthread\_cond\_attr\_t\* attr);***

1. C: puntatore alla variabile di condizionamento.
2. attr: struttura relativa agli attributi della condizione. Può essere inizializzata a nullptr per utilizzare gli attributi di default.

Un thread può effettuare operazioni di ***wait*** e di ***signal*** su una variabile di condizionamento.

***int pthread\_cond\_wait(pthread\_cond\_t\* C, pthread\_mutex\_t\* M);***

1. C. puntatore alla variabile su cui sospendersi.
2. M: mutex associato alla variabile. Va liberato quando il thread si sospende per liberare le risorse perché altri le prendano e va richiesta nuovamente una ***lock*** dopo lo sblocco.

La ***wait*** si effettua dopo l’aver verificato la veridicità di una certa condizione logica all’interno di un costrutto ***while***: un ***if*** non sarebbe sufficiente in quanto, una volta sbloccato, il thread non andrebbe istantaneamente in esecuzione e non vi sarebbe sicurezza che la condizione continui ad essere veritiera. La chiamata della funzione sospende il thread nella coda di “C” e libera il mutex “M”.

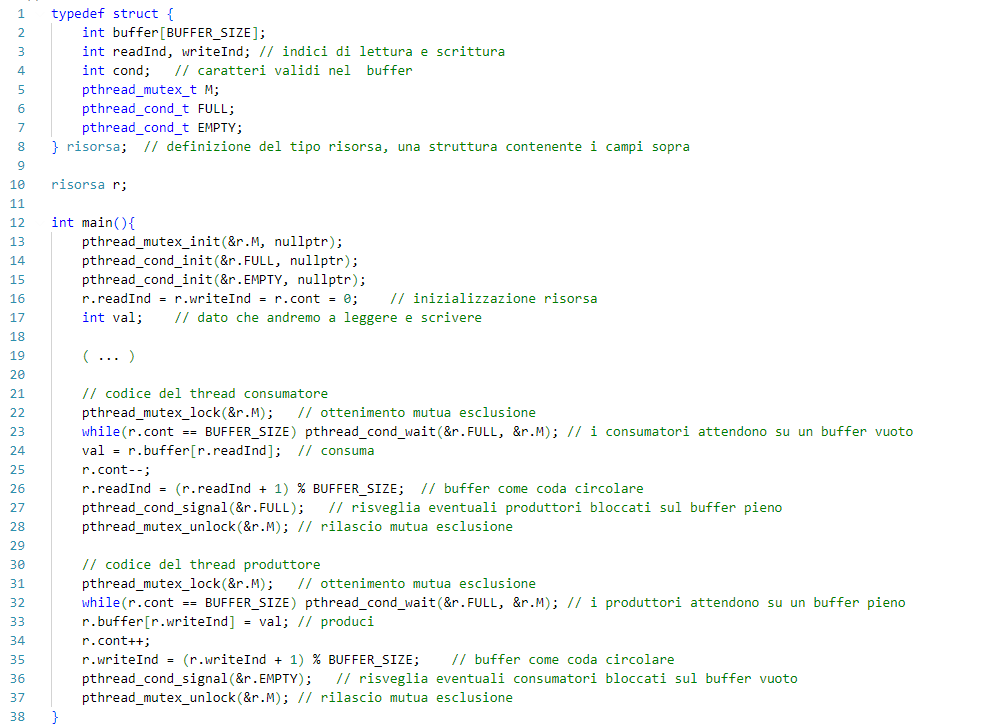
***int pthread\_cond\_signal(pthread\_cond\_t\* C);***

La ***signal***, alla chiamata, sblocca almeno uno dei thread in attesa e non ha alcun effetto se nessuno lo è.

La libreria ***pthread.h*** segue una politica signal&continue sulla ***signal***: il chiamante continua la sua esecuzione e mantiene il controllo del mutex fino al suo eplicito rilascio mentre lo sbloccato deve verificare nuovamente la condizione.

Per risvegliare tutti i thread in attesa su una condizione si può utilizzare la funzione di ***broadcast***:

***int pthread\_cond\_broadcast(pthread\_cond\_t\* C);***

Applichiamo quanto visto su un problema di produttori e consumatori: 

FILESYSTEM

In UNIX ogni risorsa è vista in maniera omogenea, ovvero come un file. Se ne distinguono tre tipi:

1. file ordinario: dati in memoria di massa.
2. file speciale: serve per riferirsi ad un dispositivo fisico come le periferiche.
3. directory: contenitori di file ed altre directory.

Una parte della memoria secondarie è dedicata alla i-List, la lista di tutti i descrittori di file, gli i-node, ognuno dei quali è riferito da un i-number.

Il contenuto di una directory è costituito da una serie di record che ne desrivono il contenuto: ***<nomefile, i‑number>***.

Un i-node descrive per ogni file:

1. tipo: ordinario, speciale, directory.
2. permessi: owner, group owner, altri.
3. dimensioni.
4. link: il numero di nomi che si riferiscono al file (hard links).
5. vettori di indirizzamento ed altre informazioni.

Alla creazione di un nuovo file viene creato il suo i-node ed aggiunto alla i-List in una certa posizione dettata dall’i-number. Viene quindi creato un nuovo record nel contenuto della directory. Dopo la creazione, il file possiede un solo nome, quello riportato nel record della directory. Tale nome rivestirà il ruolo di hard link del file.

Il filesystem consente di definire più hard link verso lo stesso i-node grazie al supporto di nomi alternativi per lo stesso file. In più, è possibile definire dei collegamenti simbolici per fare riferimento agli hard link, ovvero i soft link.

La differenza tra hard e soft link sta nel fatto che i primi puntano direttamente allo stesso i-node, mentre gli atri puntano ad altri hard link. Eliminare hard link non ha effetto sugli altri e l’i-node verrà rimosso solo se il loro numero diviene nullo. Anche l’eliminazione di un soft link non ha effetto sugli hard, ma non vale il contrario.

Per creare nuovi hard link con il comando link, utilizzato come ***ln target nomelink***. Possiamo creare dei soft link aggiungendo l’opzione ***-s*** prima di specificare “target”.

Il meccanismo per l’accesso ai file è di tipo sequenziale: per ogni file aperto da un processo viene associato un I/O pointer, un riferimento utilizzato per le letture e le scritture sequenziali e che viene avanzato dopo ogni operazione.

Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamente

Le strutture dati che garantiscono l’accesso su file sono gestite dal kernel e sono:

1. Tabella dei file aperti di sistema: una struttura dati globale nella quale vengono memorizzati tutti gli accessi dei processi ai file (riconosciuti dai loro i-node) attraverso record ***<I/O pointer, i-node>***. Esiste un’entrata per ogni file aperto da ogni processo (se n processi aprono lo stesso file allora avremo n entrate tutte con stesso i-node).
2. Tabella dei file attivi: struttura dati globale che contiene la i-List.
3. Tabella dei file aperti dal processo: ve ne è una per ogni processo e viene contenuta nella User Structure. Ogni suo elemento è detto “file descriptor” e rappresenta un riferimento ad un’entrata della tabella dei file aperti di sistema.

Gli stream sono visti come file a loro volta: i loro descrittori sono detti “di default” perché generati automaticamente dal sistema al momento dell’esecuzione del programma. Sono individuati dai valori ***STDIN***, ***STDOUT*** e ***STDERR***.

Quando un processo genera un figlio, quest’ultimo eredita dal padre una copia della User Structure e quindi anche dei descrittori di file, il che significa che entrambi i processi punteranno alle stesse entrate della tabella dei file aperti di sistema e condivideranno gli stessi I/O pointer.

ACCESSO AI FILE

Per effettuare un accesso ai file sono disponibili delle primitive di sistema.

Per l’apertura utilizziamo la ***open***:

***int open(const char\* path, int flags);***

1. path: è il percorso del file system nel quale è memorizzato il file.
2. flags: indica le modalità di accesso al file, gestite tramite macro. E’ possibile combinare tali modalità tramite un OR tra macro.

La funzione ritorna il descrittore di file aperto (un intero).

Tra le modalità di accesso vi sono ***O\_RDONLY***, ***O\_WRONLY*** o ***O\_RDWR***. Dopo l’apertura l’I/O pointer viene posizionato all’inizio del file, a meno che non si specifichi la modalità ***O\_APPEND***, che lo pone alla fine.

Per esprimere delle dimensioni si utilizza il tipo standard ***ssize\_t*** (un ***int***, necessario a rappresentare lunghezze positive ma anche degli errori se negativo).

Una volta paerto un file sarà possibile leggere o scrivere.

Effettuiamo la lettura attraverso la system call ***read***:

***ssize\_t read(int fd, void\* buf, size\_t count);***

1. fd: descrittore del file.
2. buf: puntatore ad un array nel quale andare ad inserire i dati letti.
3. count: numero di byte da leggere dal file (un ***unsigned int***).

La funzione ritorna il numero di byte effettivamente letti (che può anche essere minore di “count” in caso di errori o di termine del file) o un valore negativo in caso di errore.

Effettuiamo la scrittura attraverso la system call ***write***:

***ssize\_t write(int fd, void\* buf, size\_t count);***

1. fd: descrittore del file.
2. buf: puntatore ad un array nel quale andare a prelevare i dati da scrivere.
3. count: numero di byte da scrivere sul file (un ***unsigned int***).

La funzione ritorna il numero di byte effettivamente scritti (che può anche essere minore di “count” in caso di errori) o un valore negativo in caso di errore.

Per terminare l’utilizzo di un file utilizziamo la funzione ***close(int fd)*** che andarà a deallocare le strutture dati utilizzate.

Vediamo del codice per creare un ipotetico comando che legga da un file e stampi a video il contenuto: Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente

La libreria ***stdlib.h*** ci mette a disposizione le funzioni ***fopen***, ***fwrite***, ***fread***, ***fclose*** ma anche funzioni per leggere dagli stream come ***fscanf*** o ***fprintf***.

COMUNICAZIONE TRA PROCESSI

Il sistema mette a disposizione dei processi il meccanismo delle pipe per poter comunicare tra loro.

Una pipe è realizzata come una coda FIFO nella quale accodare i messaggi. Si tratta di un canale monodirezionale: viene realizzata tramite le sue due estremità viste come descrittori di file, uno dei quali di scrittura e l’altro di lettura. I problemi di sincronizzazione sono risolti dalle primitive di ***read*** e ***write***: il lettore si bloccherà su una pipe vuota e lo scrittore farà lo stesso su una piena.

Le pipe possono essere utilizzate nella comunicazione tra processi padre e figlio: se il padre inizializza una pipe allora anche il figlio ne erediterà i descrittori (bisogna però chiudere uno dei due descrittori, quello di lettura su un processo e quello di scrittura sull’altro in quanto non necessari). Per processi non nella stessa gerarchia si preferisce utilizzare altri strumenti come i socket.

Per creare una pipe utilizziamo la funzione di inizializzazione:

***int pipe(int fd[2]);***

L’argomento della funzione, “fd[2]”, è un vettore di descrittori di file di dimensione due che, dopo la chiamata verrà inizializzato come il file di lettura in fd[0] e di scrittura in fd[1].

La funzione ritorna “0” se ha successo o “-1” in caso di errore.

PILOTAGGIO DI APPLICAZIONI

Come già detto i descrittori di default per gli stream sono ***STDIN***, ***STDOUT*** e ***STDERR***. e vengono generati automaticamente all’esecuzione.

All’interno della libreria ***unistd.h*** sono definite delle macro per rappresentare i descrittori di file degli stream standard: ***STDIN\_FILENO***, ***STDOUT\_FILENO***, ***STDERR\_FILENO****.*

Per duplicare un file descriptor possiamo utilizzare la system call seguente:

***int dup2(int target, int newfd);***

1. target: è il file descriptor da duplicare.
2. newfd: il file descriptor nel quale andrà messa la copia di “target”.

Prima di duplicare il descrittore, la funzione chiude “newfd” se questo era precedentemente aperto.

Possiamo combinare pipe e duplicazione per redirigere il flusso di dati dagli standard stream verso altri descrittori di file: 