PANORAMICA UNIX

•UNIX è un sistema operativo multiutente e multitasking. Più utenti possono avere vari task/processi che sono eseguiti contemporaneamente.

•UNIX è anche un ambiente di sviluppo software. È nato ed è stato progettato per questo scopo (ad esempio fornisce utilità di sistema, editor, compilatori, assemblatori, interpreti, . . . ).

•UNIX è stato appositamente progettato per essere generale e indipendente dall’hardware (facilmente portabile essendo scritto nel linguaggio ad alto livello C).

COS'HA UNIX?

•Ha dimensioni relativamente ridotte, un progetto modulare e “pulito”

•Indipendenza dall’hardware

•Il codice del SO è scritto in C anziché in uno specifico linguaggio assembler

•UNIX e le applicazioni UNIX possono essere “facilmente” portate da un sistema hardware ad un altro. Il porting usualmente consiste nel trasferire il codice sorgente e ricompilarlo

•È un ambiente produttivo per lo sviluppo del software

•Ricco insieme di tool disponibili

•Linguaggio di comandi versatile

•Serendipità

Architettura di UNIX (Metodo di funzionamento)

•Un SO può essere definito come un software (leggero, chiamato Kernel) che controlla le risorse hardware del computer e fornisce un ambiente nel quale eseguire i programmi

•L’interfaccia al kernel è uno strato di software chiamato system call (chiamate di sistema). Le chiamate di sistema coinvolgono attività che possono essere raggruppate in 4 categorie principali:

- gestione dei file

- gestione degli errori

- gestione dei processi

- comunicazione tra processi

•Le librerie di funzioni comuni sono costruite in cima all’interfaccia delle chiamate di sistema (le applicazioni, volendo, possono usare entrambe). Le funzioni di libreria forniscono servizi di utilità generale al programmatore, non sono entry point del kernel, anche se possono fare uso di system call per realizzare il proprio servizio

•La shell è un’applicazione speciale che fornisce l’interfaccia con il kernel e che consente l’esecuzione di altre applicazioni.

LINGUAGGIO C

E' un linguaggio di programmazione che:

•è molto vicino all’hardware

•Molti costrutti si traducono facilmente nel codice macchina

•I tipi di dati disponibili riflettono ciò che supporta direttamente l’hardware

•L’accesso alla memoria indiretto (via i puntatori) consente ai programmatori di accedere a tutte le parti del sistema

•Storicamente UNIX, il kernel di Linux, molti software, I driver delle periferiche sono quasi tutti scritti in C

•Il C è veloce (conseguenza del suo più basso livello di astrazione rispetto ad altri linguaggi).

•Una volta scritto un programma questo deve essere trasformato in un eseguibile e la costruzione di un programma consiste in una serie di passi. Ad ogni passo, il codice è trasformato in una forma intermedia: Preprocessed, codice assembly, codice oggetto e librerie. E’ importante comprendere ciascun passo per diversi motivi perché ognuno offre più strategie per risparmiare tempo di programmazione e per massimizzare le risorse del sistema:

- il passo di preprocessing fornisce le macro per la sostituzione ripetitiva di stringhe

- Il passo assembly o compilazione fornisce il programmatore dei nomi per le locazioni di memoria praticamente lo converte in linguaggio Assembly

- Il passo di linking fornisce un modo per il riuso di codice eseguibile esistente in programmi multipli.

•Un programma C è composto dai seguenti pezzi:

•Segmento di testo: le istruzioni macchina eseguite dalla CPU

- Condivisibile (una sola copia in memoria)

- A sola lettura (protezione)

•Segmento di dati inizializzati: contiene variabili globali e statiche inizializzate nel programma.

•Segmento di dati non inizializzati (bss, “block started by symbol”): le variabili globali e statiche sono inizializzati dal kernel a 0 o al puntatore nullo prima dell’esecuzione.

•Stack: contiene le variabili automatiche con le informazioni salvate ogniqualvolta è chiamata una funzione

- Indirizzo di ritorno, registri

- La funzione chiamata alloca spazio per le sue variabili automatiche e temporanee.

•Heap: luogo in cui avviene l’allocazione dinamica della memoria (tra il segmento dati non inizializzato e lo stack)

Come eseguire un file .c con unix?

•Utilizzando la shell di linux come nel seguente esempio:

$ gcc - c reverse.c

$ gcc - c palindroma.c

$ gcc - c usaPal.c

$ gcc reverse.o palindroma.o usaPal.o - o stampaPalindroma

$ ./stampaPalindroma anna

•Rinominando un documento come Makefile e scriverci all'interno come nell'esempio:

#Commento Makefile, per farlo nella shell, nella directory usare comando Makefile

stampaPalindroma : usaPal.o palindroma.o reverse.o

gcc usaPal.o palindroma.o reverse.o - o usaPal

usaPal.o : usaPal.c palindroma.h

gcc - c usaPal.c

palindroma.o : palindroma.c palindroma.h reverse.h

gcc - c palindroma.c

reverse.o : reverse.c reverse.h

gcc - c reverse.c

SHELL

•La shell è un programma speciale usato come interfaccia tra l’utente ed il kernel del sistema operativo UNIX/Linux tramite dei comandi digitati.

•La shell GNU Bourne Again o Bash, è la shell di default di Linux è molto efficiente per operazioni: Sia a livello di programmazione che di uso interattivo. Inoltre, all’utente è consentito di adattare il proprio ambiente di lavoro

•E’ una delle più popolari shell usate dagli utenti UNIX e Linux ed è conforme allo standard POSIX (Il prompt di default è il $).

•Le Directory sono organizzato nel seguente modo:

- /dev contiene i driver dei dispositivi

- /bin e /usr/bin contiene i comandi Linux standard

- /lib e /usr/lib contiene le librerie standard di Linux

- /var contiene i file di configurazione ed i LOG file

- /etc contiene i file di configurazione di default

- /usr/local/bin contiene comandi che non fanno parte della distribuzione ma sono stati introdotti successivamente

- /opt contiene software commerciale

- /tmp memorizza file temporanei

- /sbin e /usr/sbin contiene i comandi di sistema dell’amministratore (/sbin sta per “safe” bin)

COMANDI SHELL

•help: manuale che spiega la sintassi del comando

$ ls - - help

#E' la guida che spiega la sintassi del comando ls

$ ls - - help | less

#consente di visualizzare la guida del comando ls su più pagine video (vedi comando less)

•man [comando]: consente di consultare il manuale in linea

•cd: consente di cambiare la directory corrente in quella indicata in <directory> (change directory)

$ cd Desktop

#se la directory corrente è home porta nella directory Desktop

$ cd ..

#porta della directory superiore (o padre)

$ cd /var

#in qualunque directory ci si trovi porta nella directory /var

$ cd ~ oppure cd

#porta della directory home dell'utente (/home/[nome utente])

$ cd -

#porta della directory precedente

•pwd: mostra la directory corrente

•ls: lista il contenuto di una directory (cartella).

I Parametri:

- a lista anche i file nascosti

- l mostra tutte le informazioni per ogni file (formato lungo)

- k dimensione dei file è in Kb (normalmente è in byte).

- F classifica i file a seconda del tipo.

- R elenco ricorsivo

- lah formato lungo

$ ls - l

#contenuto della directory corrente in formato lungo

$ ls - a

#contenuto della directory iniziale (root) compresi i file nascosti

$ ls - lah

#contenuto della directory corrente in formato lungo

$ ls - R

#elenco ricorsivo

•touch [nomefile]: si posiziona all'interno del file e permette anche di scrivere, se non esiste lo crea

•cp: copia file e director (Il parametro - r copia le directory e le sottodirectory ricorsivamente)

$ cp file1 cart1

#copia file1 dentro la directory cart1

$ cp - r cart1 cart2

#copia tutta la directory cart1 dentro a cart2

$ sudo cp - a cart1 cart2

#copia tutta la directory cart1 dentro a cart2 mantenendo gli stessi permessi e le informazioni su data e ora di creazione

$ cp - rv cart1 cart2

#copia tutta la directory cart1 dentro a cart2 visualizzando l'operazione (verbose)

•mv: muove o rinomina un file o una directory (Il parametro - i chiede la conferma)

$ mv vecchio nuovo

#rinomina il file vecchio in nuovo

$ mv file1 cart1

#sposta il file file1 dentro la directory cart1

•rm: rimuove un file o una directory (Il parametro - rf cancella ricorsivamente sia le cartelle che il loro contenuto)

$ rm file1 file2

#cancella file1 e file2

$ rm \*.\*

#cancella TUTTO il contenuto della directory corrente. Le eventuali directory presenti non saranno rimosse. Da usare con cautela !

$ rm - rf cart1

#cancella tutto il contenuto della directory cart1

•rmdir: rimuove una directory vuota, con questo comando è possibile cancellare una o più directory vuote (remove directory)

$ rmdir prova

#cancella la directory vuota prova

•mkdir: crea una directory

$ mkdir prova

#crea la directory prova

•pwd: mostra la directory corrente, con questo comando è possibile sapere in quale directory ci troviamo (print working directory)

$ pwd

#Se l'utente ECOmmunication si trova nella cartella principale visualizza /home/ecommunication

•chmod: con questo comando è possibile modificare i permessi di un file. (Il parametro - R consente di modificare ricorsivamente i permessi delle directory indicate e del loro contenuto)

$ chmod 0755 file1

#imposta lettura, scrittura ed esecuzione per il proprietario, e lettura ed esecuzione per gruppo ed altri per file1

$ sudo chmod - R 0755 dir1

#imposta lettura, scrittura ed esecuzione per il proprietario, e lettura ed esecuzione per gruppo ed altri per la directory di1 e per i file contenuti

$ chmod u+rwx file1

#imposta lettura, scrittura ed esecuzione per il proprietario lasciando inalterati gli altri permessi

$ chmod ugo+rwx file1

#imposta lettura, scrittura esecuzione per tutti (u user, g group, o other)

$ chmod ugo- x file1

#rimuove il permesso di esecuzione a tutti

•chown: modificare il proprietario di un file, con questo comando è possibile modificare il proprietario di un file. (Il parametro - R consente di modificare ricorsivamente i permessi delle directory indicate e del loro contenuto).

$ chown nome1 file1

#rende il file1 proprietario di nome1

$ chown nome1:gruppo1 file1

#rende il file1 proprietario di nome1 e del gruppo1

$ chown - R nome1:gruppo1 dir1

#rende la directory dir1 e i file contenuti proprietari di nome1 e del gruppo1

$ sudo chown - R root:root drupal

#rende la directory drupal e i file contenuti proprietari di root e del root

•cat: visualizza il contenuto di uno o più file

$ cat file1

#visualizza il contenuto del file1

$ cat file1 file2 > file3

#crea file3 con il contenuto di file1 e file2

$ cat file1 file2 >> file3

#aggiunge il contenuto di file1 e file2 al file3

$ cat file1 - n

#visualizza il contenuto del file1 numerandolo

$ tac file1

#visualizza il contenuto del file1 in ordine inverso

•tar: crea/decomprime archivi compressi. Comprimere file1 file2 in archivio.tar.gz:

$ tar - cvzf archivio.tar.gz file1 file2

#variabili c= crea, v=verbose, z=comprimi, f=nome archivio (ES archivio.tar.gz)

$ tar - xvf archivio.tar.gz

#x=estrai, v=verbose, f=nome archivio (ES archivio.tar.gz)

•who: stampa il nome di tutti gli utenti attualmente connessi al sistema. Oltre al nome degli utenti ci indica su che terminale sono connessi e quando si sono collegati.

•whoami: stampa il nome con il quale ci siamo connessi al sistema.

•ps: consente di visualizzare lo stato dei programmi presenti nel sistema. Lanciato senza parametri, visualizza i dati dei programmi associati con l'utente che l'ha lanciato. (Il paramentro - a visualizza tutti i processi del sistema, anche quelli associati ad altri terminali o non associati a nessun terminale (i cosiddetti daemon)).

$ ps - a

#stampa a video tutti i processi, si può implementare la pipe e la grid per cercare tra i processi

•pipe o "|" : il metodo secondo il cuale è possibile eseguire varie azioni senza esplicare su più linee nella shell

$ ps - ef |grep http | wc - l

#Con questo comando si chiede l’elenco di tutti i processi, quindi filtrare con quello che contiene la stringa “http” e finalmente si chiede il numero di linee. Così, in breve si conta il numero di processi http in esecuzione.

•Kill: mandare un segnale ad un processo identificato con il PID (process id).

$ kill –l

#elenca tutta la scelta dei possibili segnali in una tabella

$ kill –s stop PID\_number

#invia un segnale di stop al processo identificato con quel numero

$ kill –s cont PID\_number

#riprende il processo fermato (digitarlo da altra shell)

•ln - s: crea il soft- link, un collegamento indipendete

$ ln - s /etc/hosts ./hosts

#crea un soft link del file /etc/hosts nella directory corrente. Oppure

$ ln - s /etc ./lamiaetc

#crea un soflink chiamato lamiaetc che punta a /etc nella directory corrente.

•ls: crea l’hard- link, un collegamento stile puntatore

$ ln /var/www/pippo.html /var/www/topolino.html

#fa un hard\_link del file pippo nella stessa cartella ma rinominato con topolino

FILE I/O

•Il SO fornisce i servizi di base come: Aprire un file, leggere un file, allocare memoria

•Servizi forniti tramite delle routine del kernel (La libreria offre delle funzioni omonime che ci permettono di chiamare le system call come se fossero delle funzioni C Semplificando: system call= funzione C)

•Le librerie per la gestione dei fila I/O sono:

- printf: chiama la system call write per stampare i messaggi su video (file standard output)

- malloc: sbrk è la system call che alloca un certo numero di byte

- strcpy: non chiama nessuna system call

•//int open(const char \*path, int oflag, /\*mode\_t mode\*/...); è la function (<unistd.h>) che permette l'elaborazione del file.

•Il risultato intero è dato dalla posizione che il file aperto nel registro dei dispositivi I/O del sistema Per convenzione DA RICORDARE che

- il descrittore 0 viene associato allo standard input

- Il descrittore 1 allo standard output

- Il descrittore 2 allo standard error

- quindi (eccetto modifiche di sistema come il cambio delle variabili standard) il risultato sarà >2 se è stato aperto con successo, altrimenti - 1 se non è riuscita l'apertura QUESTO VALORE E' QUELLO CHE ANDREMO A PASSARE IN OGNI CAMPO FIELDS.

•La variabile path è dove riesede il file su cui operare

•La variabile oflag (<fcntl.h>) è settata come, si possono richiamare più di uno utilizzando la pipe "|":

- O\_RDONLY apri solo in lettura

- O\_WRONLY apri solo in scrittura

- O\_RDWR apri in lettura e scrittura

- O\_APPEND esegue un’aggiunta alla fine del file per ciascuna scrittura

- O\_CREAT crea il file se non esiste

- O\_EXCL non utilizzare insieme a O\_CREAT

- O\_TRUNC se il file esiste, ed è aperto con successo per sola scrittura o per lettura- scrittura, lo tronca a lunghezza zero

- O\_NOCTTY se path è un terminal device, non lo rende il terminale di controllo del processo

- O\_NONBLOCK se path è una FIFO, un file a blocchi o a caratteri, apre in maniera non bloccante, sia in lettura sia in scrittura

•La variabile mode definisce i bit di permesso di accesso ai file, in particolare:

- S\_IRUSR Lettura utente

- S\_IWUSR Scrittura utente

- S\_IXUSR Esecuzione utente

- S\_IRGRP Lettura gruppo

- S\_IWGRP Scrittura gruppo

- S\_IXGRP Esecuzione gruppo

- S\_IROTH Lettura altri

- S\_IWOTH Scrittura altri

- S\_IXOTH Esecuzione altri

•//int close(int fildes); è la SystemCall (<unistd.h>) che permette di chiudere il file. Ritorna - 1 se l'operazione non è andata a buon fine

•//off\_t lseek (int fildes, off\_t offset, int whence); è la SystemCall (<unistd.h>) della lseek, è associato un valore intero non negativo, detto offset corrente del file, che misura il numero di byte dall'inizio del file, altrimenti - 1 se occorre un errore.

•Il campo offset è la distanza (in byte) da cui iniziare ad operare sul file in particolare da whence+offset

•Il campo whence è definito dalle seguenti variabili:

- SEEK\_SET l'offset viene posto a offset byte dall'inizio del file

- SEEK\_CUR viene aggiunto offset all'offset corrente

- SEEK\_END l'offset viene posto alla fine del file, più offset

•//ssize\_t read(int fildes, void \*buf, size\_t nbytes); è la SystemCall (<unistd.h>) che permette la lettura su file, memorizza gli nbytes dal seek e li salva nella varibile buf. L'output è il numero dei caratteri effettivamente letti (al più nbytes), altrimenti - 1 se l'operazione ha dato errore. Terminata la lettura aggiorna il seek all'ultimo+1 carattere letto.

•//ssize\_t write(int fildes, const void \*buf, size\_t nbytes); è la SystemCall (<unistd.h>) che permette la scrittura su file dalla variabile buf (di almeno nbytes/8 caratteri) gli nbytes dal seek. L'output è il numero dei caratteri effettivamente scritti (al più nbytes), altrimenti - 1 se l'operazione ha dato errore. Terminata la lettura aggiorna il seek all'ultimo+1 carattere letto.

•Il "BestSize" degli nbytes nella lettura è di 4096 o 8192 bytes

•Per gestire gli errori originati dalle system call, i due principali ingredienti da utilizzare sono:

- errno variabile globale che contiene il codice numerico dell'ultimo errore generato da una system call

- perror() subroutine che mostra una descrizione “testuale” dell'ultimo errore generato dall'invocazione di una system call

FILES E DIRECTORY

•La maggior parte dei file in Unix sono di due tipi: regolari e directory. Esistono anche tipi di file aggiuntivi, I tipi di file sono:

- Regolari (il tipo più comune di file contenente dati in una qualche forma)

- Per il kernel non c’è distinzione tra dati testo o binari

- Directory (contiene nomi di altri file e puntatori alle informazioni su tali file)

- File speciali a blocco

- sono usati per rappresentare dispositivi che consistono in un insieme di blocchi a indirizzamento casuale (dischi)

- File speciali a caratteri

- sono usati per rappresentare dispositivi che costituiscono flussi di caratteri (terminali, stampanti e interfacce di rete)

- FIFO

- usato per la comunicazione tra processi

- Socket

- tipo di file usato per la comunicazione su rete tra processi

- Link simbolici

•//int stat (const char \*path, struct stat \*buf); è la SystemCall (<sys/stat.h>) che permette, dato (avendo i permessi per accedere) un percorso file: ritornano informazioni sul file. Si può trovare il file stat secondo altre due funcion:

- //int fstat (int filedes, struct stat \*buf); informazioni sul file aperto sul descrittore I/O

- //int lstat (const char \*path, struct stat \*buf); informazioni sul link simbolico, non sul file puntato da esso.

•//struct stat

{

dev\_t st\_dev; /\* device \*/

ino\_t st\_ino; /\* inode \*/

mode\_t st\_mode; /\* file type & protection \*/

nlink\_t st\_nlink; /\* number of hard links \*/

uid\_t st\_uid; /\* user ID of owner \*/

gid\_t st\_gid; /\* group ID of owner \*/

dev\_t st\_rdev; /\* device type (if inode device) \*/

off\_t st\_size; /\* total size, in bytes \*/

unsigned long st\_blksize; /\* blocksize for filesystem I/O \*/

unsigned long st\_blocks; /\* number of blocks allocated \*/

time\_t st\_atime; /\* time of last access \*/

time\_t st\_mtime; /\* time of last modification \*/

time\_t st\_ctime; /\* time of last change \*/

}; è la variabile/struttura data in output che indica il contenuto stat, (cosa possiamo fare con essa a secondo dei campi) \*/

•Per determinare il tipo di file si utilizzano le seguenti macro: con input il campo della struct stat buffer mode\_t st\_mode:

- S\_ISLNK(st\_mode) symbolic link

- S\_ISREG(st\_mode) regular file

- S\_ISDIR(st\_mode) directory

- S\_ISCHR(st\_mode) character device

- S\_ISBLK(st\_mode) block device

- S\_ISFIFO(st\_mode) FIFO

- S\_ISSOCK(st\_mode) socket

•Il campo st\_size di stat contiene la dimensione in byte del file. Ha senso solo per file regolari, directory e link simbolici.

•Se presenti, st\_blksize e st\_blocks si riferiscono, rispettivamente, al migliore fattore di blocco per eseguire operazioni di I/O sul file e al numero di blocchi da 512 byte allocati per il file.

•A ciascun processo vengono associati i seguenti identificativi:

- real user ID e real group ID, identificano l'utente

- effective user ID, effective group ID e supplementary group ID determinano i permessi di accesso ai file.

A seconda dalla chiamata al file posso avere o no determinati permessi lettura/scritture/esecuzione, quindi anche l'accesso allo stesso.

•//int access (const char \*pathname, int mode); è la SystemCall (<unistd.h>) che permette di effettuare il test di accessibilità di un file sulla base del

real user ID e del real group ID. Essa restituisce 0 se Ok, altrimenti - 1.

•//mode\_t umask(mode\_t mask); è la SystemCall (<sys/stat.h>) che viene utilizzata per assegnare ad un processo la modalità di creazione di un file. QUINDI una volta instanziato l'unmask all'interno del programma, tutto il programma creerà i file con i permessi di accesso dati dall'unmask stessa. La funzione ritorna il valore precedente della maschera di creazione dei file

•L’argomento mode\_t mask è formato da un OR bit a bit delle nove costanti di permesso di accesso ai file (quindi si possono selezionare più di una separata dalla pipe "|"), in particolare:

- S\_IRUSR 0400 owner read

- S\_IWUSR 0200 owner write

- S\_IXUSR 0100 owner execute

- S\_IRGRP 0040 group read

- S\_IWGRP 0020 group write

- S\_IXGRP 0010 group execute

- S\_IROTH 0004 others read

- S\_IWOTH 0002 others write

- S\_IXOTH 0001 others execute

•//int chmod (const char \*path, mode\_t mode); è la SystemCall (<sys/stat.h>) che perme di cambiare i permessi di accesso ad un file. Per cambiare i bit di permesso di un file, l’effective user ID del processo deve essere uguale all’ ID del proprietario, oppure il processo deve avere i diritti del superutente. Restituiscono 0 se OK, - 1 in caso di errore. essa si può presentare anche come:

- int fchmod (int fildes, mode\_t mode); che in input è il valore del file aperto nel registro I/O

•//int chown (const char \*path, uid\_t owner, gid\_t group); è la SystemCall (<unistd.h>) che permette di cambiare lo user ID ed il group ID di un file. (Solo un processo superutente può modificarne il proprietario. Un processo non superutente, proprietario del file può solo modificarne il gruppo con uno tra quelli supplementari a cui appartiene). Ritornano 0, se OK, - 1 in caso di errore. Tali funzioni possono trovarsi anche come:

- int fchown (int fd, uid\_t owner, gid\_t group); che in input è il valore del file aperto nel registro I/O

- int lchown (const char \*path, uid\_t owner, gid\_t group);che in input è il path del link.

•//int truncate (const char \*path, off\_t length); è la SystemCall (<unistd.h>) che tronca un file esistente a off\_t length byte. Se il file ha una dimensione maggiore di length, i dati oltre length non sono più accessibili. Se il file ha meno di length byte, il comportamento della funzione dipende dall'implementazione. Ritornano 0 se OK, - 1 in caso di errore. Si può trovare anche\_

- int ftruncate (int fd, off\_t length); che in input è il valore del file aperto nel registro I/O.

•//int link (const char \*oldpath, const char \*newpath); è la SystemCall (<unistd.h>) che fa puntare più directory all'i- node di un file, praticamente esegue un hard- link. Se newpath già esiste è ritornato un errore. Ritorna 0 se OK, - 1 in caso di errore (Il più delle implementazioni richiedono che oldpath e newpath risiedano nello stesso filesystem).

include <unistd.h>

•//int symlink (const char \*oldpath, const char \*newpath); è la SystemCall (<unistd.h>) che permette di crere un link simbolico soft- link. Restituisce 0 se OK, - 1 in caso di errore.

•//int readlink (const char \*path, char \*buf, size\_t bufsiz); è la SystemCall (<unistd.h>) che permette di legge i link. Il contenuto del link è posto in buf, senza carattere di terminazione. Restituisce il numero di byte letti se OK, - 1 in caso di errore.

•//int unlink (const char \*pathname); è la SystemCall (<unistd.h>) e la si usa per rimuovere un elemento dalla tabella della directory dell'i- node a essa associata. La funzione decrementa il numero di link (nella variabile tipo nlink\_t st\_nlink nella struct stat) del file puntato da pathname. Il file risulta ancora accessibile, se il numero di link è non nullo. Ritorna 0 se OK, - 1 in caso di errore. ATTENZIONE quando il file viene chiuso, il kernel conta il numero di processi che hanno aperto il file: se questo è zero ed il numero di link del file è zero, allora il file è cancellato.

RIGUARDANDO LE DIRECROTY

•//int mkdir (const char \*pathname, mode\_t mode); è la SystemCall (<sys/stat.h>) che permette la creazione di una cartella. Ritorna 0 se OK, - 1 in caso di errore;

•//int rmdir (const char \*pathname); è la SystemCall (<unistd.h>) che permette l'eliminazione di una cartella. Ritorna 0 se OK, - 1 in caso di errore;

•//DIR \*opendir(const char \*name); è la SystemCall (<dirent.h>) che permette di essere aperta (da chiunque abbia i permessi di accesso). Ritorna un puntatore se OK, NULL in caso di errore.

•//struct dirent \*readdir(DIR \*dir); è la SystemCall (<dirent.h>) che permette di leggere la directory specificata in input, da qualunque processo abbia le autorizzazioni per farlo. Ritorna un puntatore se OK, NULL alla fine o in caso di errore

•//struct dirent

{

ino\_t d\_ino;

char d\_name[NAME\_MAX + 1];

}; è la variabile/struttura data in output dalla function readdir.

•//void rewinddir(DIR \*dir); è la SystemCall (<dirent.h>) che permette di riavvolgere la directory in lettura.

•//int closedir(DIR \*dir); è la SystemCall (<dirent.h>) che chiude la directory aperta. Restituisce Il puntatore ad una struttura di tipo DIR ritornato da opendir è utilizzato dalle altre funzioni.

•//int chdir (const char \*path); è la SystemCall (<unistd.h>) che cambia la directory di lavoro del processo. Ritorna 0 se OK, - 1 in caso di errore

•//char \*getcwd (char \*buf, size\_t size); è la SystemCall (<unistd.h>) che ritorna il path della directory di lavoro corrente. Ritorna puntatore a buf se OK, NULL in caso di errore.

PROCESSI

•Un processo è un programma in esecuzione e un programma è costituito da istruzioni e dati ed è memorizzato in un file. Un processo consiste nell’insieme di eventi che scaturiscono durante l’esecuzione di un programma. E’ un’entità dinamica a cui è associato un insieme di informazioni necessarie per la corretta esecuzione e gestione del processo (memorizzate nella Process Structure, nel caso il processo è nella RAM anche l'User-structure, entrambe risiedono nel PCB) da parte del sistema operativo. Il processo Unix mantiene spazi di indirizzamento separati per i dati e per il codice del processo. Ogni processo ha uno spazio di indirizzamento dei dati privato (per le variabili del processo) e uno spazio di indirizzamento del codice condiviso. Vengono salvati nella Text Table i vari text structure che contengono il puntatore al segmento di codice e i vari processi a cui fanno rifermento. La Text Table risede nell'area di memoria della Process Table (che memorizza anche altri parametri) ove sono memorizzati tutti i processi attivi.

AVVIO DI UN PROCESSO (in c):

•Si ha con l'invocazione da una shell o da un altro programma in esecuzione. Quando si esegue un programma si esegue prima una routine di start-up speciale, specificata come indirizzo di partenza del programma eseguibile (impostato dal linker), che prende:

- valori passati dal kernel in argv[]dalla linea di comando

- variabili d’ambiente

e poi segue successivamente è la funzione main (in c).

•//int main (int argc, char \*argv[], char \*envp[]); argc è il numero di argomenti, argv è un array di puntatori agli argomenti e anvp che sono i puntatori alle variabili di ambiente da utilizzare.

TERMINAZIONE E COMUNICAZIONE DI TERMINE DI UN PROCESSO (in c):

•La terminazione di un processo consiste in una serie di operazioni che lasciano il sistema in stato coerente:

- chiusura dei file aperti

- rimozione dell’immagine dalla memoria

- eventuale segnalazione al processo padre

•Per gestire quest’ultimo aspetto Unix impiega le system call exit e wait (o waitpid) in modo coordinato:

- terminazione dell’esecuzione di un processo (exit)

- attesa della terminazione di un processo da parte del processo che lo ha creato (wait).

•Esistono diversi tipi di terminazioni di processo:

- Terminazione normale

- Ritorno dal main

- Chiamata di exit

- Chiamata di \_exit o \_Exit

- Ritorno dell’ultimo thread dalla sua routine di avvio

- Chiamata di pthread\_exit dall’ultimo thread

- Terminazione anomala

- Chiamata di abort

- Ricezione di un segnale

- Risposta dell’ultimo thread ad una richiesta di cancellazione

Ricordiamo che le funzioni di uscita sono chiamate senza alcun codice di uscita.

•Sono tre le funzioni che terminano (date come l'argomento) un programma normalmente:

•// void \_exit(int status); situata in (<unistd.h> ) (chiamata di sistema) che ritorna al kernel immediatamente;

•// void \_Exit(int status); situata in (#include <stdlib.h>) (libreria standard) che ritornano al kernel immediatamente;

•// void exit (int status); situata in (#include <stdlib.h>) (libreria standard) che prima esegue una procedura di “pulizia” e poi ritorna al kernel;

•// int atexit (void(\*func)(void)); situata in (#include <stdlib.h>) (libreria standard) registrare almeno 32 funzioni chiamate automaticamente quando è invocata exit. Resistuisce 0 se OK, < o > 0 in caso di errore. Si passa l’indirizzo di una funzione come argomento (non riceve alcun argomento e non restituisce nulla). (Quando si invoca la exit questa chiama le funzioni nell’ordine inverso rispetto a quello in cui sono state registrate).

•//pid\_t wait(int \*statloc); situata in (<sys/types.h> e<sys/wait.h>). sospende il processo invocante finché: un figlio ha terminato la propria esecuzione, oppure riceve un segnale. Ritorna un errore (- 1) se il processo non ha figli, altrimenti è restituito il pid del processo figlio terminato. Se il processo invocante ha più di un figlio, wait ritorna quando uno qualsiasi di essi ha terminato.

•L'argomento int \*statloc é un puntatore ad un intero che serve per memorizzare lo stato di uscita ed altri per indicare il segnale che ha causato la terminazione ove:

- se il byte meno significativo di statloc è 0, il byte più significativo rappresenta lo stato di terminazione.

- in caso contrario, il byte meno significativo di statloc descrive il segnale che ha terminato il figlio (terminazione involontaria).

•//pid\_t waitpid(pid\_t pid, int \*statloc, int options); situata in (<sys/types.h> e <sys/wait.h>). sospende il processo invocante finché: il processo pid ha terminato la propria esecuzione, oppure riceve un segnale. Ritorna un errore (-1) se il processo non ha figli, altrimenti è restituito il pid del processo figlio terminato.

•L'argomento pid\_t pid di waitpid ha la seguente interpretazione:

- pid ==-1 attende un qualsiasi figlio (uguale a wait)

- pid > 0 attende il processo che ha il process ID uguale a pid

- pid == 0 attende un qualsiasi figlio il cui process group ID è uguale a quello del processo chiamante

- pid < - 1 attende un qualsiasi figlio il cui process group ID è uguale a quello del valore assoluto di pid

•L'argomento int options può essere:

- 0: nullo

- WNOHUNG: non bloccherà il processo invocante se il pid del figlio non è immediatamente disponibile (ritorna 0)

- WUNTRACED: ritorna lo stato di un figlio sospeso

- WIFEXITED: vero se il figlio è terminato normalmente

- WEXITSTATUS: ritorna lo stato

- WIFSIGNALED: vero se il figlio è uscito a causa di un segnale

- WTERMSIG: ritorna il segnale

- WIFSTOPPED: vero se il figlio è fermato

- WSTOPSIG: ritorna il segnale

•Il kernel notifica al genitore la terminazione di un processo figlio mediante il segnale SIGCHLD

STATO E CARATTERISTICHE DEI PROCESSI

•Le caratteristiche di un processo può essere visto come:

•Processo pesante con codice rientrante

- Dati non condivisi

- Codice condivisibile con altri processi

•Processo con funzionamento in doppia modalità

- Processi utente (modalità utente)

- Processi di sistema (modalità kernel)

•Lo stato in cui il processo può trovarsi sono:

- Init: caricamento in memoria del processo e inizializzazione delle strutture del SO

- Ready: processo pronto

- Running: il processo usa la CPU

- Sleeping: il processo è sospeso in attesa di un evento

- Terminated: deallocazione del processo dalla memoria

Inoltre

- Zombie: il processo è terminato ma è in attesa che il padre ne rilevi lo stato di terminazione

- Swapped: il processo (o parte di esso) è temporaneamente trasferito in memoria secondaria

•Lo scheduler a medio termine (swapper) gestisce i trasferimenti dei processi

- Da memoria centrale a secondaria (swap out). Si applica, preferibilmente, ai processi bloccati (sleeping) prendendo in considerazione tempo di attesa, di permanenza in memoria e dimensione del processo (preferibilmente i processi più lunghi)

- Da memoria secondaria a centrale (swap in).Si applica preferibilmente ai processi più corti.

•Elementi della Process Structure:

- PID

- Stato del processo

- Riferimento ad aree dati e stack

- Riferimento indiretto al codice

- PID del processo padre

- Priorità del processo

- Riferimento al prossimo processo in coda

- Puntatore alla User structure

- ...

•Elementi della User structure:

- Una copia dei registri di CPU

- Informazioni sulle risorse allocate (file aperti)

- Informazioni sulla gestione di eventi asincroni (segnali)

- Directory corrente

- Proprietario

- Gruppo

- Argc/argv, PATH, ENVIROMENT

- ...

•la Process structure e la User structure sono immagine di un processo. Non tutta l'immagine è accessibile in modo user, poichè servirebbe i permessi Kernel. anche perchè la parte residente come la Process Table e la Tect table sono una sorta di registri non swappabili di proprietà del S.O. e quello che potremmo vedere sono:

- Process structure (kernel, residente): è l’elemento della process table associato al processo

- Text structure (kernel, residente): elemento della text table associato al codice del processo

- Area dati globali utente (user, swappable): Segmento dati inizializzati e non inizializzati

- Stack, heap utente (user, swappable): aree dinamiche associate al programma eseguito

- Stack del kernel(kernel, swappable): stack di sistema associato al processo per le chiamate a system call

- U- area(kernel, swappable): struttura dati contenente i dati necessari al kernel per la gestione del processo quando è residente

•//pid\_t fork (void); residente in (<unistd.h>). Crea una copia del processo che esegue la fork: L’area dati viene duplicata, PID e PPID nei processi padre e figlio sono differenti, l’area codice viene condivisa:

- Il processo creato (figlio) riceve esito = 0

- Il processo creante (padre) riceve esito > 0 che corrisponde all’identificatore (PID) di processo del processo creato

Se l’operazione fallisce genera esito = - 1.

•//pid\_t vfork (void); residente in (<unistd.h>). Crea un nuovo processo figlio, esattamente come fork, ma senza copiare lo spazio di indirizzamento. Fino a che il figlio non esegue una exec o exit, esso viene eseguito nello spazio di indirizzamento del genitore (tecnica Copy on Write). Assicura che il figlio venga eseguito per primo, fino a quando questi non chiama exec o exit. La funzione vfork viene utilizzata quando il processo generato ha lo scopo di eseguire (exec) un nuovo programma.

•Avere due processi che eseguono esattamente lo stesso codice non è molto utile, allora nella pratica accade che si genera un secondo processo per affidargli l’esecuzione di un compito specifico (ad esempio, la gestione di una connessione dopo che questa è stata stabilita) e gli si fa eseguire un altro programma. Per questo si fa uso della exec. le funzioni della famiglia exec permettono di caricare un altro programma da disco sostituendo quest’ultimo all’immagine corrente; l’immagine precedente viene completamente cancellata. Exec è una famiglia di primitive ove il primo parametro specifica la residenza del file:

- //int execl (const char \*path, const char \*arg0, ... /\* (char \*)0 \*/ ); prende in input una lista

- //int execle (const char \*path, const char \*arg0, ... /\* (char \*)0, char \*const envp[] \*/ ); prende in input una lista e delle variabili di ambiente da passare al programma come argv[0], argv[1], … , argv[n].

- //int execlp (const char \*file, const char \*arg0, ... /\* (char \*)0 \*/); prende in input una lista e un riferimento ad un $path che sarà utilizzato dal programma specificato come primo parametro.

•

•

•

•

•

•

•

•

•

•