SLIDE LABORATORIO

Lab 1 (introduzione alla bash - Rosenblatt)

La shell è un interprete di comandi: essi sono digitati dall'utente sulla riga di comando, vengono letti dalla shell, che li interpreta, e sono infine inviati al kernel che li esegue.
Sulla riga di comando troviamo il prompt, un carattere o un insieme di essi, personalizzabile dall'utente: di norma troviamo '#' per l'utente root (il superuser), mentre '$' nel caso di utenti qualsiasi.
La shell è un programma che viene eseguito quando facciamo il login e termina al logout e si trova nella directory *bin;* ve ne sono diverse tipologie e quella di default per le distribuzioni GNU/Linux è la shell bash (Bourne Again Shell).
In genere il prompt dei comandi ha la forma nome@computer~$.
Dato un generico comando, la shell dapprima separa i vari token, poi ne determina il significato ed infine esegue il comando, dopo aver effettuato le dovute preparazioni. Una linea di comandi sarà quindi formata da stringhe separate da spazi o TAB, in cui la prima parola è il comando e le restanti sono gli argomenti; è possibile che siano presenti argomenti speciali, le opzioni, che modificano il comportamento del comando tramite una certa istruzione.
È possibile usare dei caratteri speciali detti wildcard per specificare molteplici file:

- Il ? corrisponde ad un qualsiasi carattere (p?ppo = pippo, pappo, p1ppo...).
- Il carattere * qualsiasi sequenza di caratteri (*.txt = tutti i file che finiscono in .txt).
- [set] qualsiasi carattere in set (*.t[xy]t = file che finiscono in .txt e .tyt).
- [!set] qualsiasi carattere non in set (![abc].* = file che non iniziano con a, b oppure c).

I/O sotto UNIX è basato su due idee: un file I/O è una sequenza di caratteri e tutto ciò che produce o accetta dati è trattato come un file, dispositivi hardware inclusi.
Il comando **cat** serve a copiare l'input nell'output; il comando **grep** ricerca una stringa nell'input; il comando **sort** ordine le stringhe dell'input.
La **pipeline** serve a mandare l'output di un programma nell'input di un altro programma; si usa **|**.
Un processo è un programma in esecuzione e quando ne eseguiamo da shell normalmente può essercene solo uno alla volta, ma è possibile farne eseguire di più in maniera concorrente usando &.
Soltanto il processo in foreground dovrebbe avere I/O, in quanto il terminale è uno solo, ma se altri processi dovessero richiederlo allora si ha un blocco del processo nel caso si tratti di input, mentre si ha un mescolamento degli output altrimenti.

Lab 2 (Stevens, cap8: process control)

Tutti i sistemi operativi provvedono service points attraverso i quali è possibile richiedere i servizi del kernel, in particolare

Unix ha un insieme ben definito e limitato di entry points per accedere direttamente al kernel chiamati system call. Esse sono direttamente accessibili al programmatore C, a differenza di sistemi più antichi che richiedevano il linguaggio macchina, poiché nella libreria standard sono presenti funzioni con lo stesso nome in grado di richiamare l'appropriato servizio.
Dal punto di vista del programmatore, system call e funzioni di libreria appaiono come normali funzioni C, ma soltanto queste ultime possono essere riscritte.
I **primitive system data types** sono tipi di dati dipendenti dalla macchina usati dal sistema, presenti nella libreria sys/types.h, caratterizzati, in genere, dal terminare con _t.
Ogni processo ha un identificatore univoco, rappresentato come intero positivo, accessibile tramite due system calls:

- **pid_t getpid(void)** per ottenere il process ID del processo chiamante;
- **pid_t getppit(void)** per ottenere il process ID del padre del processo chiamante.

L'unico modo per creare nuovi processi è attravero una chiamata di sistema **fork** da parte di un processo già esistente che genera un processo figlio. Dalla fork si ritorna due volte: il valore restituito al processo figlio è 0, mentre il valore restituito al padre è il pid del figlio. Figlio e padre continuano ad eseguire concorrentemente le istruzioni che seguono la chiamata di fork.
Il figlio è una copia del padre, pertanto condividono dati, stack e heap, che vengono protetti dal kernel venendo settati come read-only.
Quando un processo termina, il kernel invia al padre il segnale *SIGCHLD*, che può essere ignorato dal padre (default) oppure lanciare una funzione (il signal handler); in entrambi i casi è possibile chiedere informazioni sullo stato di uscita tramite le funzioni **wait** e **waitpid**.
Se il figlio deve eseguire un programma diverso da quello del padre bisogna usare una delle funzioni di **exec**, che rimpiazza lo spazio indirizzi del figlio sostituendolo con il nuovo programma, senza intaccare il suo pid. Tutte e 6 le versioni restituiscono -1 in caso di errori, nulla altrimenti.
Le lettere finali servono a ricordare gli argomenti da passare:

- l indica una lista di argomenti, mentre v un vettore argv[] di argomenti;
- la lettera p indica che la funzione prende come argomento il nome di un file ed utilizza la variabile di ambiente PATH per cercarlo, mentre se manca va indicato il pathname completo;
- la lettera e indica che la funzione prende un envp[] array invece di usare l'environment corrente.

In alcune versioni di UNIX l'unica vera system call è **execve**, mentre le altre sono solo funzioni di libreria che successivamente chiamano execve.
I processi possono terminare involontariamente, quando si tentano azioni illegali o ci sono interruzioni mediante dei segnali,

oppure volontariamente, tramite chiamata alla funzione **exit()** o quando viene eseguita l'ultima istruzione.
Per ogni processo in esecuzione, il kernel esegue il codice del processo e determina lo stato di terminazione, che corrisponde allo stato dell'argomento exit se va tutto bene, altrimenti verrà generato uno stato che indica il motivo "anormale".
La system call exit chiude tutti i descrittori di file di un processo, dealloca le aree codice, dati e stack e poi fa terminare il processo; è sempre una chiamata senza ritorno.
Se un figlio termina prima del padre, ma questi non ne recupera lo stato, il figlio si trasforma in uno zombie e non presenterà aree codice, dati o stack allocati, ma continuerà ad avere un PCB nella Process Table. Nonostante le poche risorse richieste, la presenza di molti zombie è un problema che potrebbe costringere l'amministratore ad intervenire.
La system call **system** serve ad eseguire comandi shell dall'interno del programma.

Lab.3 (Capitolo 10 – Stevens)

Un segnale è un interrupt software che permette di gestire gli eventi asincroni e che può essere generato in qualsiasi istante sia da un processo utente che dal kernel, in caso di errori.
Ogni segnale ha un nome che comincia con SIG, ha associato una costante intera positiva diversa da 0, definita in **signal.h** e non contiene informazioni aggiuntive: in particolare, chi li riceve non può scoprire l'identità di chi li manda.
All'arrivo di un segnale, il processo può scegliere diverse risposte:

- ignorare il segnale, tranne nei casi di *SIGKILL* e *SIGSTOP*, che sono usati dal superuser per terminare o stoppare momentaneamente qualsiasi processo;
- catturare il segnale, che equivale ad associare all'occorrenza del segnale l'esecuzione di una funzione utente;
- eseguire l'azione di default associata, cioè la terminazione del processo (nella maggior parte dei segnali).

Tipi di segnali:

- *SIGABRT*, generato da una chiamata alla system call **abort**, termina il processo.
- *SIGCHLD*, che viene inviato al padre ogni volta che un processo termina o viene fermato; ignorato di default e raccoglibile tramite una delle wait.
- *SIGCONT*, inviato ad un processo fermo per farlo ripartire.
- *SIGFPE*, (floating point exception), inviato, ad esempio, quando si divide per 0.
- *SIGILL*, (illegal instruction), viene inviato quando l'hardware individua una istruzione illegale.
- *SIGINT*, (interrupt), viene usato per terminare a runtime un processo.
- *SIGALRM*, generato dalla system call **alarm**.

- *SIGQUIT*, inviato con CTRL-/, genera un core file, cioè un immagine in memoria del processo che può essere utilizzata per debugging.
- *SIGKILL*, termina il processo che lo riceve
- *SIGSEGV*, (segment violation), il processo ha fatto riferimento ad un indirizzo che non è nel suo spazio indirizzi.
- *SIGSTOP*, ferma un processo.
- *SIGSYS*, (invalid system call), il processo ha eseguito una istruzione che il kernel ha interpretato come system call senza però fornire i parametri adeguati.
- *SIGTERM*, il segnale di terminazione inviato per default dalla system call **kill**.
- *SIGBUS*, *SIGEMT*, *SIGIOT*, *SIGTRAP*, inviati quando sussistono problemi hardware.
- *SIGUSR1*, *SIGUSR2*, (user defined signals) possono essere utilizzati e definiti dall'utente, che a volte li usa impropriamente per far comunicare i processi.

*Void(*signal(int signo, void (*func)(int)))(int);*
prende in input il nome del segnale signo ed il puntatore alla funzione func da eseguire come azione da associare all'arrivo di signo. Restituisce il puntatore ad una funzione che prende come argomento un intero e non restituisce nulla, che rappresenta il puntatore al precedente signal handler. In caso di errore restituisce *SIG_ERR*, cioè -1. Il valore di func può essere *SIG_IGN* per ignorare il segnale, *SIG_DFL*, per l'azione di default, oppure l'indirizzo di una funzione da eseguire.
L'azione del padre su un segnale viene ereditata dai figli.

Int kill(pid_t pid, int signo)
int raise(int signo)
kill manda un segnale ad un processo o ad un gruppo di processi specificato da pid, mentre **raise** consente ad un processo di inviarsi un segnale. Se il valore pid è maggiore di 0, il segnale è inviato al processo di PID pid, se è 0 viene inviato a tutti i processi il cui process group ID è uguale a quello del mittente e infine se è minore di 0 viene inviato a tutti i processi il cui process group ID è uguale al valroe assoluto di pid.

A differenza del superuser, i processi non possono inviare segnali a qualsiasi processo, ma hanno bisogno di appositi permessi; inoltre i segnali non sono consigliabili come forma di comunicazione, in quanto possono essere persi e possono causare problemi in quanto interrompono l'esecuzione.

Unsigned int sleep(unsigned int secs)
Il processo che chiama **sleep** dorme finché il numero di secondi indicati è trascorso oppure interviene un segnale ed il signal handler ritorna: nel primo caso ritorna 0, altrimenti il numero di secondi mancanti.

Void abort(void)

non ritorna mai ed invia il segnale *SIGABRT* al processo che la invoca.

Unsigned int alarm(unsigned int secs)
Fa partire un conto alla rovescia di secs secondi, la cui durata può aumentare per ritardi di schedulazione, al cui termine viene inviato *SIGALRM*.

Int pause(void).
Sospende l'esecuzione finché non interviene un segnale.

Lab.4 (Capitolo 3 – Stevens)

Un file descriptor (fd) è un intero non negativo associato ad un file che serve a renderlo identificabile al kernel ed alle funzioni di I/O. Il kernel si occupa di restituire al processo che crea oppure apre un file il file descriptor, che sarà usato da lì in poi. Un file descriptor è compreso tra 0 e *OPEN_MAX*, una costante definita in **<limits.h>**.
Ogni nuovo processo apre 3 file standard: input, a cui si fa riferimento tramite il fd 0 (*STDIN_FILENO*), output, fd 1 (*STDOUT_FILENO*) ed error, fd 2 (*STDERR_FILENO*).

*Int open(const char **pathname, int oflag, ...,**/* mode_t mode*/)*
l'argomento oflag indica diverse opzioni tramite una o più delle costanti simboliche definite in **<fnctl.h>**:

- Una sola tra *O_RDONLY* (sola lettura), *O_WRONLY* (solo scrittura) oppure *O_RDWR* (apertura e scrittura).
- *O_APPEND* è opzionale e tutte le write avvengono alla fine del file.
- *O_CREAT* è opzionale e crea il terzo file se non esiste; richiede il terzo argomento mode.
- *O_EXCL* è opzionale e genera errore se *O_CREAT* è specificato ed il file esiste già.
- *O_TRUNC* è opzionale e tronca la lunghezza del file a 0 se già esiste.

L'argomento mode viene usato quando si crea un nuovo file per specificarne i permessi di accesso.

*ssize_t read(int filedes, void *buff, size_t nbytes)*
Legge dal file con file descriptor filedes un numero di byte che è al più nbytes e il mette in buff. La lettura parte dal current offset, che viene incrementato del numero di bytes letti. Se nbytes è 0, viene restituito 0 e non succede nulla, mentre se il current offset è alla fine del file o dopo, viene restituito 0 e non c'è alcuna lettura, infine se ci sono bytes in cui non è stato scritto, vengono letti byte con valore 0.

*ssize_t write(int filedes, const void *buff, size_t nbytes)*
La posizione da cui si inizia a scrivere è il current offset, che viene incrementato di nbytes, così come viene incrementata la lunghezza del file. È possibile che venga restituito un valore inferiore ad nbytes, nel caso in cui non ci sia abbastanza spazio

per scrivere tutto. Se filedes è stato aperto con O_APPEND, allora current offset è settato alla fine di ogni write.

*Int creat(const char *pathname, mode_t mode)*
Crea un file di nome pathname con i permessi descritti in mode. Restituisce il file descriptor del file aperto come write-only: equivale a *open(pathname, O_WRONLY| O_CREAT | O_TRUNC, mode)*.

Int close(int filedes)
Quando un processo termina, tutti i file aperti vengono chiusi automaticamente dal kernel.

Ogni file aperto ha assegnato un current offset, intero positivo, che misura in numero di byte la posizione raggiunta. **Open e Creat** settano l'offset all'inizio del file se O_APPEND non è specificato, mentre **Read e Write** partono dall'offset e lo incrementano.

off_t lseek(int filedes, off_t offset, int whence)
Restituisce il nuovo offset se OK. L'argomento whence può assumere diversi valori: *SEEK_SET*, ci si sposta del valore di offset a partire dall'inizio; *SEEK_CUR*, ci si sposta del valore di offset a partire dalla posizione corrente; *SEEK_END*, ci si sposta del valore di offset a partire dalla fine. Permette di settare il current offset oltre la fine dei dati esistenti nel file senza aumentarne la taglia. Nel caso di fallimento, viene restituito -1 e il valore dell'offset rimane inalterato.

Lab.5 (I/O, poi Cap.4 Stevens: Files and Directories)

int dup(int filedes)
int dup2(int filedes, int filedes2)
Assegnano un altro fd ad un file che ne possedeva uno. Il primo restituisce il più piccolo fd disponibile. Il secondo è un'operazione atomica che specifica il valore del nuovo file descriptor: assegna al file avente fd filedes anche il file descriptor filedes2, se questi è aperto, verrà chiuso, mentre se è uguale a filedes viene restituito direttamente filedes2 senza chiuderlo. Dopo la chiamata si ottengono due fd che si riferiscono allo stesso puntatore di file e il vantaggio di condividere il puntatore è quello di poter far comunicare dei processi.

In UNIX vi sono diversi tipi di file:

- Files regolari, i più comuni, che contengono dati. Il kernel non fa differenza tra testo e dati in forma binaria, quindi spetta all'utente l'interpretazione.
- Directories, file che contengono il nome di altri file e puntatori alle informazioni su di essi. Solo il kernel può scrivere, mentre i processi possono solo leggere se hanno i permessi.
- Characters-Blocks Special Files, file usati per rappresentare dispositivi di I/O, in genere.
- Pipe e FIFO, tipi di file usati per comunicare tra processi.

- Sockets, tipo di file usato per la comunicazione tra processi su una rete.
- Link simbolici, tipo di file che punta ad un altro file.

*Int stat(const char *pathname, struct stat *buf)*
*int fstat(int fd, struct stat *buf)*
*int lstat(const char *pathname, struct stat *buf)*
Dato un pathname, stat fornisce una struttura di informazioni relative al file indicato nel primo argomento. Il secondo argomento è il nome della struttura che verrà restituita, la cui definizione è system dependent. Fstat ottiene informazioni su un file già aperto tramite l'fd. Lstat restituisce informazioni sul link simbolico e non sul file referenziato.

Ogni system call di I/O può essere usata su una directory, che generalmente viene aperta per essere letta, in quanto i contenuti non si dovrebbero modificare. Per interpretarne il contenuto esiste uno standard header file **<dirent.h>**, che definisce una struct dirent che descrive una entry nella directory.

*DIR *opendir(const char *pathname)*
ritorna NULL se errore

*struct dirent *readdir(DIR *dp)*
ritorna NULL se non ci sono più elementi

*int closedir(DIR *dp)*

```
struct dirent {
     ino_t d_ino //inode number
     char d_name[256] //filename
}
```

*int mkdir(const char *pathname, mode_t mode)*
Crea una directory i cui permessi di accesso vengono determinati da mode e dalla mode creation mask del processo. La directory ha come owner ID l'effective ID del processo, mentre ha come group ID il group ID della directory padre.

*Int rmdir(const char *pathname)*
Viene decrementato il numero di link al suo i-node; se è uguale a 0 si libera la memoria solo se nessun processo ha quella directory aperta.

*Int chdir(const char *pathname)*
Cambiano la cwd del processo chiamante a quella specificata come argomento.

*Char *getcwd(char *buf, size_t size)*
Ottiene in buf il path assoluto della cwd.