Table of contents

Premessa

In questi appunti si trovano gli strumenti per preparare e dare l'esame pratico di sistemi operativi. In aggiunta ci sarà del materiale proveniente dalle mie esperienze oppure dal manuale GNU che sarà complementare alle informazioni presenti sulle slide del corso. Sottolineo che in questi appunti **verranno saltati i**

concetti più basilari che ritengo prerequisiti, se si ha bisogno di quelli integrate con le slide su moodle.

Bash

Hashbangs

Ogni script bash inizia con una riga di questo tipo che serve per far capire al terminale con quale shell deve eseguire lo script

```
#!/usr/bin/env bash
```

Comandi basici

```
alias cmd='cmd2' # cmd diventa alias di cmd2
unalias cmd # elimina l'alias
file f.txt # da informazioni sul tipo di file
type vim # da informazioni sul percorso del file
truncate f.txt # riduce o espande la grandezza di un file
read # legge linee da un file descriptor
function nome { echo "$1" } # definizione di funzione che stampa il primo argomento
nome () { echo "$1" } # analogo a sopra
```

File descriptors e redirezionamento

I 3 file descriptor di default su linux sono: - 1: **standard input** - 2: **standard output** - 3: **standard error**

Per redirezionare si utilizzano questi operatori: - Per l'input da source a dest : `input_dest < input_source` - Per l'output da source a dest : `output_source > output_dest` (se si vuole fare append usare `>>`) - Per l'output di source all'input di dest: `output_source | input_source` - Per l'output e l'error di source all'input di dest: `output_source |& input_source`

Redirezionamenti speciali: - Da un file descriptor f1 ad un altro f2: `f1>&f2` ad esempio stderr su stdout: `2>&1` - Mutare un flusso f: `f > /dev/null`

Variabili e array

```
a = 5 # dichiarazione variabile
echo "$a"; # utilizzo

lista=("a", 1, "b") # dichiarazione lista
${lista[@]} # tutti gli elementi della lista
${lista[x]} # accesso in posizione x
${!lista[@]} # tutti gli indici della lista
${#lista[@]} # dimensione lista
```

```
lista[x]=value # assegnamento in posizione x
lista+=(value) # pushback di un elemento
${lista[@]:s:n} # subarray di lista da indice s ad n

$$ # contiene il pid del processo attuale
$? # contiene lo stato dell'ultimo comando: 0 se ok altrimenti 1 o qualsiasi valore
$1,$2 ecc.. # parametri di input
```

Variabili d'ambiente

In Linux esistono delle variabili d'ambiente, ossia variabili persistenti in ogni sessione, le principali:

- **$SHELL**: indica la shell in utilizzo
- **$PWD**: indica la cartella corrente
- **$HOME**: indica la home directory
- **$PS1**: indica come è fatto il layout della riga di terminale iniziale

Concatenazione comandi

- In sequenza: `cmd1 ; cmd2`
- Esegue cmd2 se solo se cmd1 ritorna 0: `cmd1 && cmd2`
- Se cmd1 ritorna 1 ignora cmd2 (**lazy evaluation**): `cmd1 || cmd2`

NB: In un file di script ricordo che ogni linea esegue un comando bash nello stesso terminale (a differenza del **Makefile**). Se invece si vogliono eseguire sulla stessa riga bisogna usare la concatenazione.

Subshell

```
a = $(echo "prova") # esegue il comando in una subshell e cattura SOLO lo standard output
(echo "prova") # esegue comando in una subshell
```

Espansione aritmetica

```
a = 1
(( a++ ))
(( a<1 ))
(( a=$b>$c?0:1 )) # operatore ternario
```

Confronti

```
# sintassi POSIX standard (funziona sempre):
[ $a <token> $b ] # <token>: -ge | -eq | -lt  ecc..
[ $str1 <token> $str2 ] # <token>: \> | \< | != | == | =
[ -f /tmp/prova ] # controlla se prova è un file
[ -e /tmp/prova ] # controlla se prova esiste
```

```
[ -d /tmp/prova ] # controlla se prova è una directory
# NB: posso usare ! tipo:
[ ! -e file ]

if <cond>; then
    <cmd_then>
else
    <cmd_else>
fi;
```

Loops

```
for i in ${!lista[@]}; do
    echo ${lista[$i]}
done

while [[ $i < 10 ]]; do
    echo $i ; (( i++ ))
done

nargs=$#
while [[ $1 != "" ]]; do
    echo "ARG=$1"
    shift # shifta gli argomenti di input a sinistra
done
```

Makefile

I makefile sono costituiti da **target** ognuno dei quali ha dei **prerequisiti**. Invocando `make` viene eseguito il **primo target** del makefile e in caso abbia prerequisiti verranno risolti eseguendo i relativi target. Ogni target contiene una **ricetta**, ossia un insieme di istruzioni,su ogni riga **precedute da un tab**. Per ogni riga vengono eseguiti comandi SH e non BASH, perciò **la sintassi di bash non funzionerà**. **Ogni riga esegue i comandi su un nuovo terminale**, perciò se si vuole eseguire una sequenza atomica di comandi, bisognerà farlo sulla stessa riga concatenando i comandi (`&&` oppure `;`) oppure andando a capo usando per fare l'escape.

Normalmente invocando `make` viene cercato un file di nome `Makefile` o `makefile`, se si vuole dare un nome diverso allora occorrerà invocare `make -f makefilealternativo`

Le variabili in un Makefile si definiscono così: `A=1` oppure `A:=1` e si richiamano con `$(A)`

Per eseguire un determinato target del Makefile: `make target`

Per rappresentare una stringa qualsiasi in un target usare `%`. ES: ogni file con estensione `.c` si può rappresentare con `%.c`

Per silenziare un particolare comando, apporre `@` all'inizio di esso.

Target speciali

- `.SILENT` : i target specificati come prerequisito non stamperanno nulla su stderr e stdout

Funzioni speciali

- `$(eval ...)` : consente di creare nuove regole make dinamiche. ES: `$(eval VAR+=aggiunta)` concatena aggiunta a var.
- `$(shell ...)` : cattura l'output di un commando shell. ES: `PWD=$(shell pwd)`.
- `$(wildcard *)` : restituisce un elenco di file che corrispondono alla stringa specificata. ES: `OBJ_FILES:=$(wildcard *.o)`

Makefile esempio

In realtà non è altro che la mia soluzione dell'esame del 15-06-2022.

```
DEST=./

main:
    if [ -e $(DEST) ]; then \
        cd $(DEST); \
        gcc main.c -std=gnu90 -o coda; \
    else \
        echo "?ERROR" >&2; \
    fi;

cexp:
    gcc prove.c -std=gnu90 -o prove;

exp: cexp
    ./prove

.SILENT:
```

C

Librerie da includere

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

```c
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/wait.h>
#include <time.h>
#include <pthread.h>
#include <signal.h>
#include <sys/stat.h>
```

Direttive particolari

```c
#define F(A,B) A*B
#if
#else
#endif
```

Puntatori a funzione

```c
// Definizione di un puntatore a funzione fn_pointer
return_type (*fn_pointer) (type_arg1, type_arg2);

// Esempio
int (*fn_pointer) (int,int); // esempio

int f(int a, int b){
    return a*b;
}

fn_pointer = &f;

int result = (*fn_pointer)(1,2);
```

Funzioni utili

```c
// stampa dentro dest (sovrascrivendolo) usando il formato specificato
sprintf(dest, "format", src1,src2.. );
/* Tokenizza il buffer separandolo in base a delimiter(stringa).
Viene rimosso il delimitatore dal token e la stringa originale
viene modificata irreversibilmente */
char* token = strtok(buffer, delimiter_str);
char* token = strtok(NULL,delimiter_str); // continua a tokenizzare
```

Files

Streams

```c
// Dichiariamo uno stream
FILE* f = fopen("path_to_file","flags");

/*
flags can be:
    - r: read
    - w: write or overwrite (create)
    - r+: read and write (update existing)
    - w+: read and write. (truncate if exists or create)
    - a: write at end (update)
    - a+: read and write at end (create)
*/

// Read from stream
int a;
char str[10];
/* NB: la fscanf vuole le aree di memoria delle variabili,
infatti uso &a per ottenere l'indirizzo dell'intero,
mentre la stringa è un indirizzo di memoria di per sé.
*/
fscanf(f,"%d %s", &a, str);

// Print to stream
fprintf(f,"intero: %d stringa: %s", a, str);

// gets string reading maximum dim_buffer-1 from stream
fgets(buffer,dim_buffer,f);

// gets character from stream
fgetc(character,f);

// gets line from stream
int characters_read = getline(&line, &dim_line, f);

// tells if stream is ended
feof(f);

// close stream
fclose(f);
```

Files Descriptors

```
// Open a file descriptor
int f = open("filename", flags, mode);
/*
Flags can be ( per unirli usare | ):
    - Deve contenere uno tra O_RDONLY, O_WRONLY, o O_RDWR
    - O_CREAT: crea il file se non esistente
    - O_APPEND: apri il file in append mode
    - O_TRUNC: cancella il contenuto del file (se aperto con W)
    - O_EXCL: se usata con O_CREAT, fallisce se il file esiste già

Mode ( opzionale ) specifica i permessi da dare al file
( fare man open per vederli tutti, oppure 0777 per darli tutti ).
*/

// Read from file descriptor (remember to add '\0' to strings)
int bytesRead = read(f,buffer,bytes_to_read);

// Create a file
int esito = creat(path_to_file, mode); // se va male ritorna -1

// Move the pointer in the stream counting from whence
lseek(f, offset_bytes, whence);

/*
Whence can be:
    - SEEK_SET = da inizio file,
    - SEEK_CUR = dalla posizione corrente
    - SEEK_END = dalla fine del file.
*/

// Close file descriptor
close(f);
```

NB: gli stream principali sono **stdin**,**stderr**,**stdout** ; i loro file descriptor si ricavano chiamando **fileno(stream)** oppure usando le costanti **STDIN_FILENO**, **STDERR_FILENO**, **STDOUT_FILENO**.

Duplicazione dei file descriptors

```
/* The dup() system call creates a copy of the file descriptor oldfd,
using the lowest-numbered unused file descriptor for the new descriptor. */
int f = dup(oldfd);

/* The dup2() system call performs the same task as dup(), but instead of
using the lowest-numbered unused file descriptor,it uses the file
```

```
descriptor number specified in newfd. If the file descriptor newfd was
previously open, it is silently closed before being reused. */
int f = dup2(int oldfd, int newfd);
```

Forking e system calls

Exec

```
/*
Queste chiamate sostituiscono il nostro programma eseguendo il programma specificato:
    - path: full path to file
    - file: filename considering the folder or $PATH
    - argv: pgm arguments (NULL terminated)
    - env: variables to pass. eg: "A=3,B=prova"
*/
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],char *const envp[]);
int execl(const char *path, const char * arg0, ...,argn,NULL);
int execlp(const char *file, const char * arg0, ...,argn,NULL);
int execle(const char *file, const char * arg0, ...,argn,
NULL, char *const envp[]);
int execve(const char *filename, char *const argv[], char
*const envp[]);

//Executes a command in an SH shell ( BASH SINTAX DOES NOT WORK ! )
int system(const char * string);
```

Forking

```
/* Il padre ottiene il pid del figlio mentre il figlio ottiene 0.
-1 in caso di errore */
int pid = fork();
pid_t getpid(); // restituisce il PID del processo attivo
pid_t getppid(); // restituisce il PID del processo padre

/* Attende la conclusione di UN figlio e ne restituisce il PID,
riportando lo status nel puntatore passato come argomento se non NULL */
pid_t wait (int *status);

pid_t waitpid(pid_t pid, int *status, int options)
/*
analoga a wait ma consente di passare delle opzioni e si può specificare come pid:
    - -n (<-1: attende un qualunque figlio il cui "gruppo" è |-n|)
    - -1 (attende un figlio qualunque)
    - 0 (attende un figlio con lo stesso "gruppo" del padre)
    - n (n>0: attende il figlio il cui pid è esattamente n)
```

```
Lo stato di ritorno è un numero che comprende più valori "composti",
interpretabili con apposite macro utilizzabili come funzione, (altre come
valore) passando lo "stato" ricevuto come risposta
*/

// restituisce lo stato vero e proprio (ad esempio il valore usato nella "exit")
WEXITSTATUS(sts);
WIFCONTINUED(sts); // true se il figlio ha ricevuto un segnale SIGCONT
WIFEXITED(sts); // true se il figlio è terminato normalmente
WIFSIGNALED(sts); // true se il figlio è terminato a causa di un segnale non gestito
WIFSTOPPED(sts); // true se il figlio è attualmente in stato di "stop"
WSTOPSIG(sts); // numero del segnale che ha causato lo "stop" del figlio
WTERMSIG(sts); // numero del segnale che ha causato la terminazione del figlio
```

Segnali

Su linux sono 32, ecco i principali:

- SIGALRM (alarm clock)
- SIGCHLD (child terminated)
- SIGCONT (continue, if stopped)
- SIGINT (terminal interrupt, CTRL + C)
- SIGKILL (kill process)
- SIGQUIT (terminal quit)
- SIGSTOP (stop)
- SIGTERM (termination)
- SIGUSR1 (user signal)
- SIGUSR2 (user signal)

```
/* Specifica come gestire un segnale. Con handler = SIG_IGN ignora il
segnale, con SIG_DFL usa quello di default. Signal funziona per un singolo
segnale inviato, poi viene ripristinato il comportamento di default
*/
sighandler_t signal(int signum, sighandler_t handler);


void myHandler(int sigNum){ ... } // Handler personalizzato


int kill(pid_t pid, int sig);
/*
Invia un segnale ad uno o più processi a seconda dell'argomento pid:
    - pid > 0: segnale al processo con PID=pid
    - pid = 0: segnale ad ogni processo dello stesso gruppo di chi esegue "kill"
    - pid = -1: segnale ad ogni processo possibile (stesso UID/RUID)
    - pid < -1: segnale ad ogni processo del gruppo |pid|
```

```c
*/


sigset_t mod,old;
int sigemptyset(sigset_t *set); // Svuota
int sigfillset(sigset_t *set); // Riempie
int sigaddset(sigset_t *set, int signo); // Aggiunge singolo
int sigdelset(sigset_t *set, int signo); // Rimuove singolo
int sigismember(const sigset_t *set, int signo); // Interpella
// Update the current mask with the signals in 'mod'
int sigprocmask(int how, const sigset_t *restrict set,sigset_t *restrict oldset);

/*
A seconda del valore di how e di set, la maschera dei segnali del processo viene cambiata.
Nello specifico:
    - how = SIG_BLOCK: i segnali in set sono aggiunti alla maschera;
    - how = SIG_UNBLOCK: i segnali in set sono rimossi dalla maschera;
    - how = SIG_SETMASK: set diventa la maschera.
Se oldset non è nullo, in esso verrà salvata la vecchia maschera (anche se set è nullo).
*/

int sigpending(sigset_t *set); // Riempie il set con i segnali pending

// Per gestire i segnali in maniera sofisticata.

struct sigaction {
    void (*sa_handler)(int); // handler semplice
    /* handler complesso con parametri aggiuntivi.
    Con info->si_pid si ottiene il pid del mandante */
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask; //Signals blocked during handler
    int sa_flags;
    /* modify behaviour of signal:
        - SA_SIGINFO ( utilizza l'handler complesso sa_sigaction )
        - SA_RESETHAND (resetta l'handler di default dopo un primo handling)
    */
    void (*sa_restorer)(void); //Deprecated, not POSIX
};

//Applica la sigaction
int sigaction(int signum, const struct sigaction *restrict act,
              struct sigaction *restrict oldact);
```

Errors in C

```c
/* Dichiaro la variabile errno come esterna,
essa contiene l'ultimo codice di errore */
export int errno;
strerror(errno); // da un messaggio verboso sull'errore
```

Pipes

Pipe anonime

```c
int fd[2];
pipe(fd); // ritorna 0 se tutto ok

/* Lettura da pipe bloccante che restituisce il numero di byte letti,
a meno che la write sia stata chiusa. */
int read(int fd[0],char * data, int num);

// scrittura su pipe
int write(int fd[1],char * data, int num);

/* Si chiude sempre la parte che non ci interessa,
per evitare che la read sia bloccante all'infinito. */
```

Pipe named

```c
// crea una pipe(file) con nome , ritorna -1 se da errore altrimenti 0
int mkfifo(const char *pathname, mode_t mode);

/* Per usare la coda entrambi i processi devono averla aperta
contemporaneamente usando la open con il nome del file che è la pipe */
```

Message Queues

```c
int creat(const char *pathname, mode_t mode); // to create the file for the ftok

// to create the key to get and make the queue
key_t ftok(const char *pathname, int proj_id);

/*
Crea o ottiene l'id della coda.
msgflag: 0777 ( da mettere per ottenere tutti i permessi sulla coda ),
IPC_CREAT (crea se non esiste),
IPC_EXCL(fallisce se la coda esiste).
Fallisce con -1.
*/
int msgget(key_t key, int msgflg);
```

```c
/* Per inviare messaggi posso definire una struttura ad hoc
e usare mtype per definire il tipo di messaggio che andrà nella coda */
struct msg_buffer{
    long mtype;
    char mtext[100];
} message;

// Flags: IPC_NOWAIT( la chiamata fallisce in assenza di spazio)
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

/* Per ricevere messaggi.
Flags:
- IPC_NOWAIT ( fallisce se non ci sono messaggi ),
- MSG_NOERROR ( tronca il messaggio se è più grande del buffer,
in assenza fallirebbe ) */
ssize_t msgrcv(int msqid,void *msgp,size_t msgsz,long msgtyp,int msgflg)


/*
A seconda di msgtyp viene recuperato il messaggio:
    - msgtyp = 0: primo messaggio della coda (FIFO)
    - msgtyp > 0: primo messaggio di tipo msgtyp, o primo messaggio di tipo
    diverso da msgtyp se MSG_EXCEPT è impostato come flag
    - msgtyp < 0: primo messaggio il cui tipo T è min(T <= |msgtyp|)
*/

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
/*
Modifica la coda identificata da msqid secondo i comandi cmd,
riempiendo buf con informazioni sulla coda (ad esempio:
tempo di ultima scrittura, di ultima lettura,numero messaggi nella coda, etc...).
Valori di cmd possono essere:
- IPC_STAT: recupera informazioni da kernel
- IPC_SET: imposta alcuni parametri a seconda di buf
- IPC_RMID: rimuove immediatamente la coda
- IPC_INFO: recupera informazioni generali sui limiti delle code nel sistema
- MSG_INFO: come IPC_INFO ma con informazioni differenti
- MSG_STAT: come IPC_STAT ma con informazioni differenti
*/

struct msqid_ds {
    struct ipc_perm msg_perm; /* Ownership and permissions */
    time_t msg_stime; /* Time of last msgsnd(2) */
    time_t msg_rtime; /* Time of last msgrcv(2) */
    time_t msg_ctime; //Time of creation or last modification by msgctl
```

```
    unsigned long msg_cbytes; /* # of bytes in queue */
    msgqnum_t msg_qnum; /* # of messages in queue */
    msglen_t msg_qbytes; /* Maximum # of bytes in queue */
    pid_t msg_lspid; /* PID of last msgsnd(2) */
    pid_t msg_lrpid; /* PID of last msgrcv(2) */
};

struct ipc_perm {
    key_t __key; /* Key supplied to msgget(2) */
    uid_t uid; /* Effective UID of owner */
    gid_t gid; /* Effective GID of owner */
    uid_t cuid; /* Effective UID of creator */
    gid_t cgid; /* Effective GID of creator */
    unsigned short mode; /* Permissions */
    unsigned short __seq; /* Sequence number */
};
```

Threads

```
int pthread_create(
    pthread_t *restrict thread, /* Thread ID */
    const pthread_attr_t *restrict attr, /* Attributes */
    void *(*start_routine)(void *), /* Function to be executed */
    void *restrict arg /* Parameters to above function */
);

int pthread_cancel(pthread_t thread); // Invia una richiesta di cancellazione

/* Con state = PTHREAD_CANCEL_DISABLE o PTHREAD_CANCEL_ENABLE
( abilita o disabilita la possibilità di essere cancellati) */
int pthread_setcancelstate(int state, int *oldstate);

/* Con type = PTHREAD_CANCEL_DEFERRED( attende un cancellation point )
o PTHREAD_CANCEL_ASYNCHRONOUS ( cancellabile in qualsiasi momento ) */
int pthread_setcanceltype(int type, int *oldtype);

int pthread_join(pthread_t thread, void **retval); // attende un thread joinabile

// setta un certo parametro del thread
int pthread_attr_setxxxx(pthread_attr_t *attr, params);
// ottiene un certo parametro del thread
int pthread_attr_getxxxx(const pthread_attr_t *attr, params);

/*
I più importanti sono er la set/get_detachstate per valutare se un thread è joinabile:
    - PTHREAD_CREATE_DETACHED: non può essere aspettato
```

```
    - PTHREAD_CREATE_JOINABLE: default, può essere aspettato
*/
```