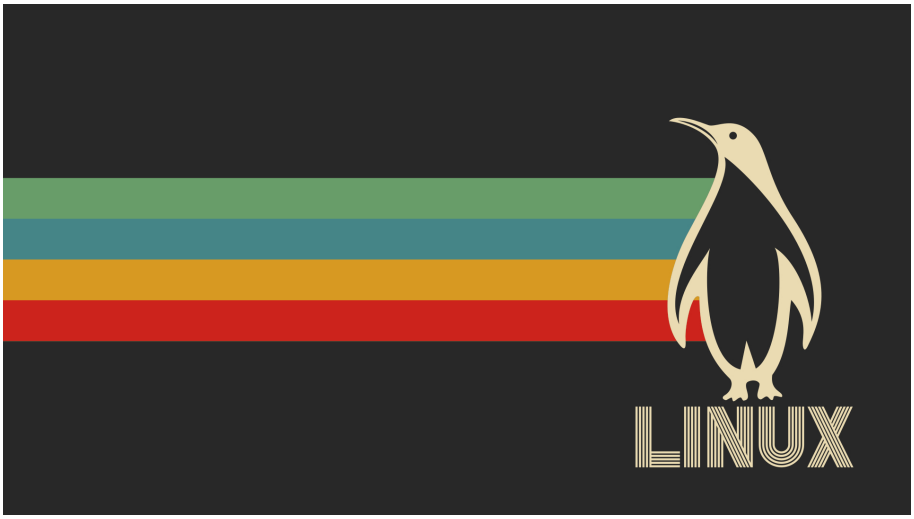
Linux per programmatori 1

Dal corso di Sistemi Operativi dell'Universita' di Pisa

Matteo Bernini

https://github.com/mattebernini/

https://t.me/mattebernini



21 settembre 2022

Indice

1 Introduzione

Questa dispensa non pensata per assoluti principianti, infatti da per scontate alcune nozioni di base di Linux come ad esempio i comandi base da terminale (cd, ls, pwd...), pensata invece per chi ha una conoscenza molto base di Linux e vuole imparare tutto ci che c' da sapere per iniziare a programmare in ambiente Linux. Per tutto ci che c' da sapere intendo: alcuni comandi da terminale pi avanzati, come usare al meglio il terminale e l'utilizzo di alcune chiamate di sistema Linux in linguaggio C per scrivere programmi in C in ambiente Linux.

1.1 Uso dei metacaratteri

I metacaratteri sono caratteri speciali usati all'interno di comandi da terminale che hanno significati particolari:

- metacarattere *: all'interno di un pathname sostituisce zero o pi caratteri e pu essere usato per indicare tutti i file che iniziano, finiscono oppure hanno ad un certo punto del path name una stringa

```
# mostra tutti i file java (che finiscono con .java)
$ ls *.java
Main.java Hello.java Program.java
# quelli che iniziano per "esempio"
$ ls esempio*
esempio.cpp esempio.py esempio.txt
# quelli che hanno "hello" nel pathname
$ ls *hello*
ciao_hello.c helloworld.cpp nohelloworld.txt

```

- metacarattere ?: all'interno di un pathname sostituisce un singolo carattere con qualsiasi carattere

```
$ ls hello?java
hello.java hello_java.txt hello_java.md

```

- metacarattere [ ]: all'interno di un pathname sostituisce un singolo carattere con un carattere nell'insieme tra parentesi quadre

```
$ ls a[b-e]c.c
abc.c adc.c aec.c acc.c

```

1.2 Redirezione I/O

I canali di input standard sono 3: stdin (input da tastiera), stdout (output su schermo), stderr (output su schermo di errore), con questi comandi possibile invece deviare l'input o l'output di un comando da terminale verso file.

```
# invia stdout a un file (scrive il risultato del comando sul file)
# se non esiste il file viene creato, se esiste sovrascritto
$ ls *.java > lista_file_java.txt
# usare 2> per stderr oppure &> per entrambi, >> in append

# si pu fare anche su pi file
$ ls *.java >> lista_file_java.txt 2> errori.txt

# da in input il contenuto del file list.txt e scrive l'output in sortedlist.txt
$ sort < list.txt > sortedlist.txt
```

Con le pipeline (—) si pu collegare l'output di un comando all'input del successivo:

```
# leggi poche righe da ls
$ ls -l | head
# scrivi in abc.txt le prime 10 righe di abc.md
$ cat abc.md | head > abc.txt
```

2 Gruppi

Gli utenti e i gruppi sono identificati da una stringa (nome) ed un intero (id), ad ogni gruppo possono appartenere 1 o pi utenti, ogni utente appartiene ad almeno un gruppo (quando viene creato crea un gruppo col suo nome).

```
$ passwd    # cambia password utente corrente
$ id [username]     # visualizza UID, gruppo principale e altri gruppi dell utente selezionato
$ groups [username] # visualizza i nomi dei gruppi dell utente selezionato
# con privilegi di root
$ adduser [username]
$ deluser [username]
```

2.1 Permessi di accesso al filesystem

Si pu accedere al filesystem in lettura, scrittura ed esecuzione, in lettura si intende leggerne il contenuto sia per un file che per una cartella, in scrittura si intende modificare un file mentre per la cartella si intende creare, rinominare e cancellare files, in esecuzione per la cartella vuol dire "attraversarla" mentre per il file appunto eseguirlo (sia esso uno script o un file binario). Ogni file (o directory) ha un utente proprietario ed un gruppo proprietario, perci i 3 permessi sopracitati sono specifici per 3 classi di utenti: per il proprietario, per gli utenti che appartengono al gruppo proprietario e per il resto degli utenti non facenti parte di quei 2 insiemi. Per visualizzare i permessi:

```
$ ls -l

-rw-r--r-- 1 matte matte 4096 lug 11 15:17 esempio.md
# I permessi vengono rappresentati cos
```

- Il primo carattere indica file (-) o directory (d).
- I 9 successivi indicano i permessi per owner, groupowner, others, il - significa permesso non concesso.
- Le due stringhe successive sono i nomi dell'owner e del groupowner.

E' possibile anche la rappresentazione ottale (usata nei comandi), ogni classe di utente associato ad un numero che si ottiene dalla somma di 3 numeri (uno per ogni permesso): 4 se lettura permessa, 2 se scrittura permessa, 1 per l'esecuzione. 777 garantisce ogni permesso per tutti, 776 invece impedisce l'esecuzione agli others. Per modificare i permessi:

```
1  # do tutti i permessi a tutti per esempio.txt
2  $ chmod 777 esempio.txt
3
4  # con rappresentazione simbolica:
5  # chmod [chi][cosa][quali permessi] fileName
6  # chi = u, g, o (le tre classi)
7  # cosa -> + aggiungi, - togli, = assegna
8  $ chmod ugo+rwx esempio.txt     # stesso di prima
9  $ chmod o-wx esempio.txt        # tolgo a others scrittura e
   esecuzione
```

I permessi speciali SUID e SGID permettono di acquisire i privilegi dell'owner il primo e del groupowner il secondo, questi due permessi speciali sono indicati con s al posto di x (rappresentazione letterale) mentre in ottale con un valore aggiuntivo davanti agli altri 3 cos calcolato: 4 se attivo SUID + 2 se attivo SGID.

2.2 Gestione gruppi

```
1  $ addgroup nome_gruppo
2  $ gpasswd nome_gruppo  # cambio password
3  $ gpasswd -a username nome_gruppo  # aggiungo username a nome_gruppo
4  $ gpasswd -d username nome_gruppo  # lo rimuovo
5  # solo admin possono
6  $ gpasswd -M utente1,utente2... nome_gruppo  # definisco membri
7  # solo root pu
8  $ gpasswd -A utente1,utente2... nome_gruppo  # definisco admin
9  $ delgroup nome_gruppo
10
11 $ newgrp nome_gruppo   # nome_gruppo diventa nuovo gruppo primario
   dell'utente che lancia il comando (per quella sessione)
```

3 Gestione Files

3.1 Ricerca nel filesystem

Il comando "find" permette di ricercare files all'interno del filesystem in base al loro nome ed eseguire azioni su di essi. Oltre al nome la ricerca pu essere condotta per: owner, tipo, grandezza, timestamp e permessi. Il contenuto del file non incide sulla ricerca.

```
$ find [path1 path2 ] [espressione]
```

Possono essere specificati pi path usati come radice dell'albero per condurre la ricerca (se si vuole cercare in tutto il filesystem mettere "/" come unico path). L'espressione composta da una sequenza di elementi che sono collegati da -o (or) o da -a (and), se omesso il collegamento and di default. Gli elementi possono essere:

- Test: condizione per la ricerca

```
    $ find [path1 path2 ] -name pattern -type [dfl]    # dfl (
directory, file, link)
    $ find [path1 path2 ] -size [+-]n[ckMG]            # +5k ->
pi  grande di 5 kilobyte
    $ find [path1 path2 ] -user user -group group
    $ find [path1 path2 ] -user user -group group
    $ find [path1 path2 ] -perm [-/]quali              # -,
permessi almeno presenti, / almenu uno di quelli, niente =
esattamente quelli

```

- Azioni: da compiere sui file trovati

```
    $ find [path1 path2 ] -delete      # da scrivere dopo test
per eliminare file trovati
    $ find [path1 path2 ] -exec command ;
    $ find [path1 path2 ] -exec cat {} ;    # con {} ci si
riferisce ai file trovati
    # -execdir per eseguire il comando a partire dal path del file
 trovato

```

Esempi:

```
# applica "ls -l" ai files non csv pi  grandi di 50 Megabyte
$ find path ! -name '*.csv'  size +50M  execdir ls  l {} \;
```

Al posto di "find" si pu usare "locate", questo comando molto pi semplice, infatti attraverso un database va a ricercare semplicemente i file all'interno del filesystem.

```
$ locate [options] file1
```

Questo database per non detto sia sempre aggiornato, inoltre "find" risulta molto pi potente di "locate" ed esegue pure comandi sui file trovati a discapito della maggior complessit sintattica.

3.2 Ricerca di testo

Il comando grep (general regular expression print) permette di cercare in uno o pi file di testo le linee (righe) che corrispondono ad espressioni regolari o stringhe letterali

```
$ grep [opzioni] [-e] modello [-e modello2 ] [file1 file2 ]
```

Con "$\hat{}$ in [modello] lespressione deve trovarsi ad inizio riga. Con "$in[modello]lespressionedevetrovarsiinfon$
$z].Il.indicaqualsiasicaratterementrel'*halastessavalenzadeimetacaratteri(perindicareilcaratterepuntou$

4 Archiviazione e compressione

Il comando tar (Tape ARchive) permette di archiviare/estrarre una raccolta di file e cartelle

```
$ tar modalit [opzioni] [file1  o cartella/e]
```

....

5 Processi

I processi mantengono spazi di indirizzamento separati per dati e codice, infatti il codice pu essere condiviso se vi sono pi processi che eseguono lo stesso programma mentre lo spazio dei dati privato. Il PBC o descrittore del processo diviso in due strutture, la prima sempre in memoria la Process Structure contiene le informazioni indispensabili mentre la seconda soggetta a swap la User Structure. La prima struttura possiede un puntatore alla seconda.

5.1 Primitive

5.1.1 fork()

La primitiva fork() permette di clonare il processo che la chiama in un processo figlio identico in tutto e per tutto al padre.

```
pid_t pid;
pid = fork();
// il codice procede su due binari paralleli
if(pid==0)
    // processo figlio
if(pid > 0)
    // processo padre

pid_t getpid();     // pid processo corrente
pid_t getppid();    // pid processo padre del processo corrente
```

5.1.2 exit() e wait()

Un processo pu terminare volontariamente usando la funzione exit(int status) oppure involontariamente per eventi esterni o per azioni illegali commesse dal programma. La exit() una funzione senza ritorno che permette di comunicare al padre lo stato di terminazione tramite il parametro "status". Chiamando la wait(int *status) il padre avr come risultato il pid del primo figlio che termina (alla chiamata si blocca in attesa della terminazione) o -1 in caso il processo non abbia figli, come argomento a wait(&var) va passato il riferimento alla variabile in cui mettere lo stato della terminazione del processo figlio.

```
// proc padre
pid_t pid_figlio;
int stato_terminazione_figlio;

pid_figlio = wait(&stato_terminazione_figlio);
```

```
    // in pid_figlio ho il pid del primo figlio terminato (se avevo un figlio)
    // in stato_terminazione_figlio ho un valore che posso leggere solo tramite delle macro
    WIFEXITED(stato_terminazione_figlio); // vero se terminato volontariamente
    WEXITSTATUS(stato_terminazione_figlio); // ritorna lo stato di terminazione
```

5.1.3 exec()

Con questa primitiva posso sostituire nel processo codice e dati del programma che sto eseguendo con un altro programma, quello del comando passato per argomento. exec() in realt una famiglia di primitive: execl(), execvp(), execle(), execclp(), execv(), execve().

```
    // int execl(char* path, char* arg0,    , char* argN, (char*)0)
    execl("/bin/ls", "ls", "-l", NULL, NULL);
    // da ora in poi il programma   : ls -l
    // tutto quello che c'   qui sotto non conta pi
    // il processo termina quando finisce ls -l
    ...
```

5.2 Interazione tra processi

I processi hanno spazio di indirizzamento privato e non condividono variabili, l'unico modo per farli comunicare quindi a livello kernel tramite sincronizzazione o scambio di messaggi usando le primitive.

5.2.1 Segnali

I segnali sono il modo che i processi hanno per sincronizzarsi, infatti questo meccanismo permette la notifica di eventi asincroni ad uno o pi processi e possono essere utilizzati dal sistema operativo stesso. I segnali sono vere e proprie interruzioni software. Ricevere un segnale porta a: esecuzione di un handler (funzione di gestione) definito dal programmatore o un default handler. Il segnale pu anche essere ignorato, nel caso dell'handler invece questo viene eseguito interrompendo il processo e facendolo ripartire dopo la fine dell'handler. Sono definiti 32 segnali nel file "signal.h" identificati da un intero ed un nome simbolico. Tramite la primitiva signal() si pu definire la funzione che dovr gestire il segnale. Tramite la kill(), alarm() e sleep() invece possibile inviare segnali (implicitamente per le ultime due).

```
    void handler(int sig)
    {
        /* corpo handler */
    }
    int main()
    {
        sighandler_t old;
        // associo al segnale SIGUSR1 l'handler "handler"
        // metto in old il puntatore al precedente handler
        old = signal(SIGUSR1, handler);
```

```
11
12          // termino immediatamente il processo pid
13          kill(pid, SIGKILL);
14          // pid -> 0 ai processi del process group
15          // pid -> -1 a tutti i processi a cui posso inviare segnali
16          // pid < -1 a tutti i processi il cui process group    -pid
17
18          // blocca il processo e poi lo risveglia dopo 30 secondi
19          sleep(30);
20
21          // provoca la ricezione di un segnale SIGALARM dopo 30 secondi
22          alarm(30);
23      }
```

5.2.2 Pipe

Una pipe come una mailbox nella quale si possono accodare messaggi in modo FIFO. E' un canale monodirezionale con due estremi: scrittura e lettura, a ciascuno associato un file descriptor su cui si fa read() e write(). I figli ereditano le pipe dal padre, per comunicare fuori dalla gerarchia utilizzare i socket.

```
1       int main()
2       {
3           int fd[2];
4           ...
5
6           pipe(fd);
7
8           pid = fork()
9
10          if(pid == 0)
11          {
12              // figlio
13              read(fd[0], buff, lmsg);
14          }
15          else
16          {
17              // padre
18              close(fd[0]);
19              write(fd[1], msg, lmsg);
20          }
21      }
```

5.3 Gruppi di processi

Il primo processo ad andare in esecuzione detto "init system" ed anche il processo da cui discendono tutti gli altri. In Linux Ubuntu esso systemmd. I processi sono organizzati in gruppi, i figli hanno lo stesso gruppo del padre mentre i processi mandati in esecuzione hanno un nuovo process group. Ogni processo ha una serie di identificatori e permessi, oltre al PID, PPID e PGID (id gruppo processo) sono presenti anche id per l'user e il groupuser che si dividono in reali ed effettivi (in base ai bit SUID o SGID): RUID, RGID, EUID, EGID.

I processi normali presentano un ulteriore campo di priorit denominato "niceness", pi basso pi priorit il processo avr, l'intervallo [-20, 19], solo root con la syscall nice() pu abbassare la niceness, l'utente pu solo alzarla.

5.4 Da terminale

```
# kill [options] pid [pid2 ]
# posso inviare segnali solo ai miei processi (a tutti se sono root)
$ kill -SEGNALE pid

# visualizzare processi di tutti:
ps u a x

# albero dei processi
pstree

# eseguire comandi in background
$ comando &

# CTRL+Z per fermare un comando
# CTRL+C per ucciderlo

# deviare l'output in nohup.out
nohoup comando

# specificare la nice a 3 mandando in esecuzione run
nice -n 3 ./run
```

6 Thread

Un thread un flusso di esecuzione indipendente all'interno di un processo, ad un singolo processo possono essere associati pi thread, questi condividono le risorse e lo spazio di indirizzamento con gli altri thread. Sono detti anche "processi leggeri" visto che creazione, distruzione e cambio di contesto sono molto meno onerosi rispetto ad un processo normale (non c' da fare cambi contesto etc...). La "leggerezza" dei thread un bel vantaggio in quanto ci consente di scrivere applicazioni multithread molto pi efficenti di una identica multiprocesso, questo per comporta delle piccole attenzioni che dobbiamo tenere: il codice infatti deve tenere conto che un altro thread pu involontariamente interagire con un altro e che le risorse condivise devono essere accedute in mutua esclusione, per evitare inconsistenze in quest'ultime. Linux supporta nativamente, a livello di kernel, il concetto di thread, infatti anche gli stessi processi sono visti come "thread che non condividono risorse" e inoltre proprio il thread l'unit di scheduling.

6.1 Standard POSIX

6.1.1 Creazione

```c
#include <pthread.h>

void* tr_code(void* arg)
{
    printf("arg = %d\n", *(int*)arg);
    free(arg);
    pthread_exit(NULL); // termina il thread
}

int main()
{
    pthread_t tr;
    int* arg1 = (int*)malloc(sizeof(int));
    *arg1 = 1;
    // siamo nel thread main

    // creo un thread tr che esegue il codice di tr_code
    // e gli passa come unico argomento arg1
    pthread_create(&tr, NULL, tr_code, arg1);
}
```

6.1.2 Mutua esclusione

```
#include <pthread.h>

// definizione variabile di mutua esclusione
pthread_mutex_t M;
// risorsa condivisa
int risorsa = 0;

void* tr_code(void* arg)
{
    printf("arg = %d\n", *(int*)arg);
    free(arg);
    pthread_mutex_lock(&M);
    // accedo a risorsa solo in mutua esclusione perch     condivisa
    risorsa++;
    pthread_mutex_unlock(&M);
    pthread_exit(NULL); // termina il thread
}

int main()
{
    pthread_t tr;
    int* arg1 = (int*)malloc(sizeof(int));
    *arg1 = 1;

    // inizializzo variabile mutua esclusione
    pthread_mutex_init(&M, NULL);

    pthread_create(&tr, NULL, tr_code, arg1);
}
```

6.1.3 Sincronizzazione

```
typedef struct {
    int buffer[DIM_BUFF];
    int readInd, writeInd;
    int cont;

    // per accedere alla risorsa in mutua esclusione
    pthread_mutex_t M;

    // condition var per la sincronizzazione
    pthread_cond_t FULL;
    pthread_cond_t EMPTY;
} risorsa;

risorsa r; // variabile globale

void* produttore(void* arg)
{
    int val = *(int*)arg;

    // mutua esclusione su risorsa
    pthread_mutex_lock(&r.M);

    // buffer pieno? attendi
    while (r.cont == DIM_BUFF)
        pthread_cond_wait(&r.FULL, &r.M);

    // utilizzo risorsa
    r.buffer[r.writeInd] = val;
    r.cont++;
    r.writeInd = (r.writeInd+1) % DIM_BUFF;

    // Risveglia un eventuale thread consumatore
    pthread_cond_signal(&r.EMPTY);
    // rilascio la mutua esclusione
    pthread_mutex_unlock(&r.M);
}
void* consumatore(void* arg)
{
    int val;

    // mutua esclusione su risorsa
    pthread_mutex_lock(&r.M);

    // buffer vuoto? attendi
    while (r.cont == 0)
        pthread_cond_wait(&r.EMPTY, &r.M);

    // utilizzo risorsa
    val = r.buffer[r.readInd];
    r.cont--;
    r.readInd = (r.readInd+1) % DIM_BUFF;

```

```
53          // Risveglia un eventuale thread produttore
54          pthread_cond_signal(&r.FULL);
55          // rilascio la mutua esclusione
56          pthread_mutex_unlock(&r.M);
57      }
58
59
60      int main()
61      {
62          pthread_t tr[NTHREADS];
63          int* args[NTHREADS];
64
65          pthread_mutex_init(&r.M, NULL);
66          pthread_cond_init(&r.FULL, NULL);
67          pthread_cond_init(&r.EMPTY, NULL);
68          r.readInd = r.writeInd = r.cont = 0;
69
70          for (int i = 0, j; i < 10; i++)
71          {
72              j = i+5;
73              args[i] = (int*)malloc(sizeof(int));
74              *args[i] = i+1;
75              pthread_create(&tr[i], NULL, produttore, args[i]);
76              pthread_create(&tr[j], NULL, consumatore, NULL);
77          }
78          return 0;
79      }
```

7 File

Produttore-Consumatore Ad ogni file aperto viene associato un I/O pointer che scorre il file sequenzialmente e ci permette di leggerlo o di scriverci. Linux tiene in memoria due tipi di tabelle: una con i file aperti in tutto il sistema, una tabella per ogni processo con i file aperti da quest'ultimo. All'avvio vengono generati automaticamente 3 descrittori: STDIN, STDOUT, STDERR. Il processo figlio essendo identico al padre eredita anche i suoi file descriptor.

```
1       #include<fcntl.h>
2       #include<stdlib.h>
3       #include<stdio.h>
4
5       #define BUF_SIZE 64
6
7       int main(int argc, char** argv)
8       {
9           if (argc < 2) {
10              printf("Usage: %s FILENAME\n", argv[0]);
11              exit(-1);
12          }
13
14          // LETTURA
```

```
15          int fd = open(argv[1], O_RDONLY);
16          if (fd < 0) {
17              perror("Errore nella open\n");
18              exit(-1);
19          }
20
21          // SCRITTURA
22          char buffer[BUF_SIZE];
23          ssize_t nread;
24          while((nread=read(fd, buffer,BUF_SIZE-1)) > 0){
25              buffer[nread] = '\0';
26              printf("%s", buffer);
27          }
28          close(fd);
29          if (nread < 0) {
30              perror("Errore nella read\n"); exit(1);
31          }
32          exit(0);
33      }
```

Riferimenti bibliografici

[1] Lezioni e slide del Ing. Domenico Minici del corso di Sistemi Operativi, Universit di Pisa, Corso di Laurea in Ingegneria Informatica.

[2] Alcuni pezzi di codice d'esempio sono presi per intero dalle slide sopracitate.