Sistemi Operativi

Riassunto delle lezioni di Laboratorio A.A. 2018/2019

Matteo Franzil

29 aprile 2019

Indice

1 Comandi Bash

La seguente è una lista, non esaustiva, dei comandi visti a lezione per l'uso della shell Bash.

1.1 Comandi

- `free` → memoria libera
- `pwd` → cartella corrente
- `df` → partizioni
- `dirname` → ottiene il nome della cartella di un dato file

1.2 Piping

- `;` → esecuzione in sequenza
- `&& ||` → esecuzione in sequenza con corto circuito
- `|` → piping classico
- `>` → redirect su file classico
- `1>` → redirect di `stdout`
- `2>` → redirect di `stderr`

1.3 Intestazioni

- `#! /bin/bash` → intestazione
- `#Script "Hello World"` → intestazione nominativa

1.4 Scripting

- `var=VALORE` → assegnazione (sono sempre stringhe)
- `${var}` → stampa (con eventuale esecuzione)
- `$@` → equivalente ad `argv`
- `$#` → equivalente ad `argc`
- `$1`, `$2`, `$3`, `$n` → i primi 9 parametri passati
- `shift` → cestina il primo argomento nella lista
- `" ... "` → crea un comando / stringa interpolando variabili
- `$(( ... ))` → contengono espressioni aritmetiche: se all'interno uso una `$var`, viene sostituita come fosse una macro, se uso il singolo contenuto di `var` allora viene inserito il valore come avesse le parentesi.
- `bc` → comando che supporta il piping in entrata, per eseguire operazioni in float

- `# Commenti` → commento classico di singola riga
- `$?` → valore di ritorno globale, usato dagli script (tipo return 0 in C): ha significato booleano (**0** niente errori, arriva fino a **256**)

1.5 Booleani

- `test ...` → si aspetta un espressione booleana, e internamente modifica il registro booleano visto prima:
- `-eq, -ne, -lt, -gt` → operandi booleani utilizzati
- `[ ... ]` → sintassi di testing alternativa (gli spazi sono importanti!); attenzione che le parentesi quadrate sono considerate come ultimo comando eseguito
- `[[ ... ]]` → raggruppamento di espressioni booleani per utilizzare operatori comuni (¿, ¡ ...)-
- `-f (file) -d (directory)` → verificano l'esistenza di un dato file/cartella.

1.6 Cicli

- `if [ ... ]; then ... ; else ...; fi` → costrutto if standard
- `case $var in; a|b) ... ;; c) ... ;; esac` → costrutto switch standard
- `for (( init ; case; step )); do; ...; done` → costrutto for standard
- `until [[ ... ]] ; ...; done` → while negato
- `while [[ ... ]] ; ...; done` → while standard

1.7 Funzioni

- `func() { ... }` → accedibile come fossero degli script (`func arg1 ... argn`)

1.8 Varie

- `$( ... )` → sottoshell che esegue comandi in un processo separato
- `BASH_SOURCE[0]` → contiene il nome dello script in esecuzione
- `exit n` → uscita con codice d'errore
- `1> 2> ... n>` → redirezionamento dei diversi canali sui file, come visto prima (1 = stdout, 2 = stderr); è possibile redirezionare stderr su stdout e viceversa tramite il comando `2>&1`, oppure mettendo tutto su file: `1>output 2>&1`. L'ordine in cui vengono interpretati i redirect sono da destra a sinistra.

2 Make

2.1 Introduzione

Make è un tool utilizzato per automatizzare processi all'interno di sistemi Unix. Viene principalmente usato per automatizzare la compilazione dei file.

I Makefile sono composti da regole, composte da un **identificativo** (o nome) e da una **ricetta** (una serie di comandi indentati con una tabulazione `\t` rispetto al nome della ricetta):

```
regola:
    echo "Ciao"
```

I file vengono poi eseguiti tramite il comando `make -f nome_file.makefile`. Notare come i comandi vengono anche stampati su `stdout` oltre a essere eseguiti. Questa funzionalità può essere sfruttata per stampare a video anche i procedimenti che vengono eseguiti dal file, ma può risultare fastidioso: si può quindi inserire una chiocciola `@` che impedisce la stampa del comando.

Il nome del file è opzionale e se non presente viene cercato all'interna della cartella corrente un nome corrispondente a `Makefile`.

In ogni ricetta, ogni riga è trattata singolarmente come un singolo processo, caricando la giusta cartella di lavoro di volta in volta. Bisogna quindi fare attenzione all'uso di `cd`.

All'inizio di ogni regola, si può inoltre specificare una o più *dipendenze* che devono essere rispettate prima di avviare la regola corrispondente. A tal scopo si usa generalmente dare il nome del file che verrà generato alla regola stessa, se la regola ne genera qualcuno:

```
file.cc: dipendenza1 dipendenza2...
    comandi per generare file.cc...
```

In tutti gli altri casi si parla di *pseudo-regole* ed il nome può essere attribuito di fantasia.

2.2 Definizione di macro

All'interno dei makefile si possono definire delle macro, che possono essere utilizzate con una sintassi analoga a bash. Ciò risulta problematico in quanto `$` è interpretato prima da Makefile e poi da Bash.

```
MACRO1=Pippo
main:
    @echo Hello $(MACRO1)
```

Le macro possono inoltre essere sovrascritte al momento dell'avvio di `MAKE` con la medesima sintassi (ma è necessario che la macro stessa sia definita all'interno del file).

Con la sintassi `$(...)` è possibile accedere a una sotto-shell, analoga a quella di Bash, per eseguire comandi all'interno del Makefile stesso. In questo sotto-comando tutti gli `\n` vengono tramutati in spazi.

3 gcc

3.1 Introduzione

`gcc` è un compilatore multi source/target per compilare i file sorgenti C. Viene utilizzato in combinazione con Make per compilare velocemente più sorgenti.

3.2 Flag

- `gcc main.c -S` → compila in codice Assembly
- `gcc main.c -E` → esegue solo il preprocessore
- `gcc main.c -c` → compila senza linkare
- `gcc main.c -o` → genera i file oggetto binari
- `gcc main.c -g` → compilazione con mapping per `gdb`
- `gcc main.c -DNOME=VAL` → compilazione con definizione di macro NOME=VAL

3.3 gdb

`gdb` è un debugger usato in combinazione con `gcc`. In questo ambiente controllato, possiamo tenere traccia dello stato dell'esecuzione del programma mappando opportunamente le variabili (in quanto i nomi vengono persi durante la compilazione).

Per iniziare, si usa il comando `run` nella shell interattiva che viene presentata da `gdb`.

Punto centrale del debugging è il **breakpoint**; si tratta di un punto, indicato manualmente con `break *`, che causerà l'interruzione dell'esecuzione del programma una volta raggiunto. A questo punto è possibile o usare `step <count>` e `next <count>` per fare uno o più step generici o principali, oppure interferire volontariamente in maniera più o meno pesante, ad esempio usando `print` per ottenere i valori delle variabili oppure `set variable vname=value` per modificare il valore. Si può infine continuare l'esecuzione con `continue`.

4 C

4.1 Introduzione

Il lingaggio C è un linguaggio di "basso livello", alla base dei sistemi Unix e dotato di una struttura minimale e ben definita, che velocizza le prestazioni omettendo funzioni di alto livello presenti invece in altri linguaggi.

Prima della compilazione vi è un *pre-processore* il cui compito è interpretare le direttive e gli alias definiti all'inizio del file (tramite l'apposito carattere `#`), generando il cosiddetto file "sorgente direttivo". Le direttive sono varie, seguono le principali:

- `#include <lib>` → inclusione di codici libreria
- `#include "file"` → inclusione di file `.h` esterni
- `#define NOME TESTO` → macro
- `#define FUNC(X,Y) (X) (Y)` → macro con funzioni
- `#if`, `#ifdef NOME`, `#ifndef NOME`, `#else`, `endif` → controlli

Il linguaggio viene quindi compilato e analizzato una sola volta: prima viene trasformato in codice intermedio (Assembly), poi in file oggetto `.o` e infine linkato ai file libreria e trasformato in linguaggio macchina `.out`.

4.2 Sintassi

- `int printf(const char *format, ...)` → stampa con formattazione: `%[flags][width][.precision][length]specifier`; alcuni specifier sono:

  `%c` → `char`

  `%d` → `int`

  `%li` → `long`

  `%s` → `char*` (ovvero una stringa)

  `%p` → locazione di memoria

- Si può ovviare alla mancanza dei booleani tramite un apposita macro: `#define BOOL char; #define FALSE 0; #define TRUE 1;`. Si noti che questa variabile può comunque, analogamente a un char, contenere valori fino a 255; è quindi cura del programmatore utilizzarla in maniera corretta e non confusionaria.

4.3 Organizzazione della memoria

Data l'immagine del processo in memoria, partendo dall'indirizzo più basso, troviamo:

- OS Kernel
- Text segment: istruzioni della CPU
- Data segment: variabili globali
- BSS: variabili non inizializzate
- Heap e stack: allocazione dinamica automatica e personalizzata

In C, non c'è garbage collection e la memoria allocata dinamicamente va esplicitamente deallocata per evitare fastidiosi memory leak. AL momento dell'allocazione in memoria, distinguiamo tra **variabili statiche**, corrispondenti alle variabili globali dichiarate all'inizio, **variabili dinamiche**, ovvero le variabili dichiarate con `new` e con puntatori e che devono essere esplicitamente deallocate, e **variabili auomatiche**, locali a una funzione.

4.4 Stringhe, puntatori, tipi

In C, una **stringa** è un array di `char` che termina con uno `\0`, detto *terminatore di sequenza*. Per poter lavorare con le stringhe è necessario utilizzare delle opportune funzioni che le trattano come array.

Un **puntatore** è una variabile che punta a una certa casella di memoria. `int* a` rappresenta un puntatore (non inizializzato) a una variabile di tipo `int`. L'asterisco viene inoltre utilizzato per accedere alla locazione `*a`; `&a` rappresenta invece la locazione di memoria puntata in esadecimale.

Importante è la notazione: infatti il simbolo `*` si lega al nome della variabile e non al tipo, per cui scrivere `int* a, b` definirà una variabile `a` di tipo puntatore a int e `b` di tipo int, e non due puntatori come si potrebbe pensare.

In C è inoltre possibile costruire tipi di dati derivati e rinominarne altri con i costrutti `struct` e `typedef`. La prima permette di creare "tipi aggregati", in maniera analoga a una classe in un linguaggio Object-Oriented; la seconda permette di ridefinire, in maniera analoga a una macro, un certo tipo di dato oppure una struct con un nuovo nome.

Esiste infine `union` che funziona in maniera simile, ma i campi sono mutualmente esclusivi e condividono lo spazio d'indirizzamento: assegnare uno causa la sovrascrittura degli altri campi.

4.5 Allocazione

All'interno della libreria `stdlib.h` abbiamo svariate primite per l'allocazione e deallocazione di memoria, tra cui:

- `sizeof(item)` che restiuisce la quantità di memoria occupata da `item`
- `malloc(size)` che alloca `size` quantità di memoria
- `calloc(num, size)` che alloca spazio per `num` oggetti di dimensione `size`
- `realloc(ptr, newsize)` che rialloca la memoria puntata da `ptr` con la nuova dimensione `newsize`
- `free(ptr)` che libera la memoria occupata da `ptr`

Un esempio di utilizzo:

```
int* a;
a = (int*) malloc(sizeof(int) * 100);
```

5 Processi

Per la teoria riguardante i processi, consultare gli appunti di teoria del corso.

La seguente è una lista, non esaustiva, di comandi utili alla gestione dei processi di sistema Unix tramite Bash.

- `command&` → lancia un processo in background, restituendo il suo PID
- `ps` → elenca i processi attivi su sistema
- `ps -o pid,ppid,pgid,sid,uid,euid` → elenca i processi attivi con informazioni extra

La seguente è una lista degli identificatori di processo:

- `PID` → attributo di identificazione univoco per istante temporale
- `PPID` → id del processo genitore
- `PGID` → id di un gruppo di processi, che possono collaborare in maniera stretta tra loro; al momento della creazione i figli entrano nel gruppo di un genitore
- `SID` → insieme di gruppi di processi; alla creazione un processo è associato alla sessione del padre
- `RUID, EUID` → rispettivamente l'id utente del creatore ed esecutore del processo

6 Architettura

6.1 Introduzione

L'architettura di sistema Unix ha come elemento centrale il kernel, che si occupa della gestione delle risorse (ulteriori informazioni nella parte di teoria del corso). Al momento dell'avvio, il kernel verifica lo stato dell'hardware, monta la partizione di sistema in modalità di sola lettura e lancia *init*, ovvero il primissimo processo (`/sbin/init`). A partire da *init* vengono fatti partire tutti i processi che si occupano dell'avvio del sistema, sia in maniera diretta con `fork` sia indiretta con `exec`.

I programmi utente, per accedere all'hardware, devono per forza passare per il kernel (vedi parte del *context switch* nella parte di teoria). Il kernel rimane l'unica risorsa di sistema ad avere accesso privilegiato a tutte le componenti dell'hardware.

Ogni programma è all'oscuro delle astrazioni fornite dal sistema operativo e crede di avere la CPU tutta per sè, e non gli viene permesso di disturbare le azioni degli altri processi. Grazie a ciò i processi Unix sono considerati particolarmente stabili.

6.2 File system

Il sistema Unix fornisce due interfacce per l'accesso ai file da parte dei processi: gli *stream* e i *file descriptor*.

6.2.1 Stream

Negli *stream*, un file è descritto come puntatore a un tipo file (`File*`): una volta creato un oggetto di questo tipo possiamo usare strumenti per la lettura, la scrittura, con relativa formattazione e buffering, che forniscono un livello di astrazione in più rispetto a quanto fornito dal kernel.

6.2.2 File descriptor

Nei *file descriptor*, ogni file è associato a un intero (`int`) che punta alla rispettiva locazione della tabella di file. In questo caso vi è un buffer specifico per la lettura e la scrittura, di dimensione arbitraria ma fissata. Per accedere al buffer si fa uso di apposite system call, ovvero `open, read, write, lseek, close`. Queste permettono un controllo maggiore sul file ma complicano la scrittura di codice.

Da notare come ogni processo Unix possiede, alla nascita, tre file descriptor `0, 1, 2` che corrispondono a `stdin`, `stdout` e `stderr`. Tutti i file che vengono apertitramite file descriptor useranno interi successivi a questi (con un limite di circa 100).

Al momento dell'apertura di un file con `open`, la system call localizza l'i-node del file (ovvero la struttura dati appartenente al file system che contiene informazioni sul file come dimensioni, permessi, etc...) e ne inserisce un puntatore nella *tabella dei file aperti dal processo* assegnandogli un intero, in modo da potervi rapidamente accedere.

A livello kernel, inoltre, vi sono inoltre due tabelle globali. Una è la *tabella dei file attivi* che contiene una copia dell'i-node di tutti i file che sono stati aperti da tutti i processi, l'altra è la *tabella dei file aperti del sistema* che invece contiene riferimenti a tutti i file che non hanno ancora terminato operazioni di lettura o scrittura. Ogni entry contiene il puntatore I/O della posizione e un puntatore all'i-node della tabella dei file attivi. Se un file viene aperto da pià processi, avrà elementi distinti in questa tabella.

7 System Call

Come visto, i programmi fanno uso di system call per accedere all'hardware. Queste system call sono disponibili sulle librerie condivise (visualizzabili con `ldd`) che vengono automaticamente caricate in compilazione e al momento di caricamento. Due di queste librerie sono `ld-linux.so`, che carica eventuali librerie mancati per conto del sistema operativo, e `libc.so`, che contiene le principali funzioni GNU.

7.1 Varie

Alcune di queste system call sono `time()`, `ctime()`, `exit()` che rispettivamente

- `time()` fornisce il tempo in secondi dal sistema (misurato a partire dall'epoca Unix)
- `ctime()` rende un tempo `time_t`, passato come puntatore, leggibile (la cosìddetta human-readable form)
- `exit(int status)` chiude il processo e tutti i file descriptor associati ad esso. La procedura di exit passa al processo padre l'intero utilizzato come parametro, che fornisce indicazioni sul fallimento o sul successo del programma.

7.2 File

7.2.1 File descriptor

Per quanto riguarda l'accesso ai file, utilizziamo le seguenti system call in modalità file descriptor:

```
#include <fcntl.h>

int open (char *filename, int access, int permission);
int read (int handle, void *buffer, int nbyte);
int write(int handle, void *buffer, int nbyte);
int close(int handle);
```

`access` e `permission` sono degli *enum* che possono assumere i seguenti valori, a seconda dei permessi che si desidera assegnare:

```
Access:     O_RDONLY O_APPEND O_WRONLY O_RDWR O_BINARY O_TEXT
Permission: S_IWRITE S_IREADS_IWRITE | S_IREAD
```

`open` restituisce un handle, generalmente > 2, che rappresenta l'intero descrittore del file che andrà usato in seguito.

7.2.2 Stream

Per accedere ai file in modalità stream, invece:

```
#include <stdio.h>

FILE *fopen(const char *filename, const char *mode)
int fclose(FILE *stream)
int fgetc(FILE *stream)
int feof(FILE *stream)
```

7.3 Permessi

Le seguenti system call sono analoghe al sistema di gestione permessi per Unix. `uid_t, gid_t, mode_t` sono interi che rappresentano le modalità, l'utente e il gruppo che si desidera assegnare.

```
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);
int chown(const char *pathname, uid_t owner, gid_t group);
```

7.4 Processi

Come visto in precedenza, ogni processo ha un unico process ID. Viene usato `pid_t` che è un intero.

```
#include <stdlib.h>

int getpid();
int getppid();
int getuid();
```

Tramite `int system(char* cmd)` infine, il processo corrente crea un processo figlio che esegue la shell di sistema /bin/sh e il comando impartito. Una volta terminato il processo padre continua da dove aveva interrotto.

8 Inter-Process Communication

8.1 Fork

Affinché un processo in esecuzione, dotato di un immagine in memoria, possa generare un nuovo processo figlio, deve fare uso della system call `fork` per crearne uno. Tramite questa, il processo corrente viene clonato, duplicando Program Counter, registri e variabili. A questo punto i due processi coesistono con PID diversi, e vanno competere tra loro per il tempo CPU: verranno trattati infatti allo stesso modo dallo scheduler. La funzione infine restituisce un intero, che sarà `0` per il codice eseguito nel processo figlio e il PID del figlio al codice eseguito nel padre.

Segue una lista di system call utilizzabili nell'ambito del forking.

- `pid_t fork()` → genera un nuovo processo
- `pid_t getpid()` → restituisce il "pid" del processo
- `pid_t getppid()` → restituisce il "pid" del padre
- `pid_t wait(int *status)` → attende la conclusione dei figli, è possibile avere informazioni sullo stato d'uscita passando una variabile, altrimenti si può usare NULL
- `pid_t waitpid(pid_t pid, int *status, int options)` → attende la conclusione di un figlio specifico (pid¿0), di tutti (pid=-1) o di quelli appartenenti a un certo gruppo (pid¡-1: appartenenti al gruppo il cui valore assoluto è pid, pid=0: appartenti allo stesso gruppo del chiamante). Options specifica se attendere in casi particolari. wait(NULL) è equivalente a waitpid(-1, NULL, 0);

8.2 Exec

A differenza di `fork`, `exec` sostituisce l'immagine del processo corrente con un'altro. Il comando utilizzato per fare ciò è il seguente, dove `file` è il processo da caricare.

```
int execvp (const char* file, char* const argv[])
```

Esistono comunque diverse famiglie di comandi exec che variano a seconda dei parametri passati: dato `exec`, è possibile avere:

- Una `l` finale significa che gli argomenti vengono presi in numero variabile.
- Una `v` finale significa che viene preso come parametro un vettore di stringhe
- Una `e` finale permette di passare un ulteriore vettore di stringhe del tipo chiave-valore passate come ambiente
- Una `p` finale prende un nome del file da cercare del path di sistema per l'immagine.

8.3 Segnali

I segnali sono interruzioni a livello software, identificate con un intero, alla quale viene associato un valore enum mnemonico che ne descrive brevemente il funzionamento.

Esempi di segnali sono `SIGALRM` (usato come sveglia), `SIGCHLD` (figlio ucciso), `SIGCONT` (continua, se bloccato precedentemente), `SIGINT` (interrupt utente, tipo CTRL+C), `SIGKILL` e `SIGSTOP` (interrupt forzato di sistema), `SIGQUIT` (uscita dal terminale), `SIGTERM` (terminazione del processo).

Per ogni processo è tenuta una lista dei segnali da catturare e di quelli bloccati. Ogni volta che il processo viene schedulato, se la lista dei segnali da catturare non è vuota viene gestita a seconda dei segnali presenti.

Ogni segnale ha un suo handler, che ha il compito di continuare l'esecuzione, terminare o stoppare il processo, oppure ignorarlo. Ogni processo è libero di sostituire l'handler con una funzione a suo piacere, a parte per `SIGKILL` e `SIGSTOP` che non possono essere bloccati.

Un handler è una funzione che restituisce `void` e accetta un intero come argomento, corrispondente al segnale da gestire. All'interno della funzione è necessario utilizzare uno `switch` per discriminare il segnale corretto e gestirlo a modo. Per binare la funzione appena scritta al rispettivo segnale, bisogna usare la funzione `signal` che prende in ingresso il codice e la funzione (passata come puntatore).

Per mandare un segnale è necessario usare la funzione `kill` che prende come parametri il pid del processo e il codice del segnale da inviare.

9 Errori

Per gestire gli errori usiamo:

- `errno`, una variabile globale che contiene il codice dell'ultimo errore di sistema
- `strerror`, una funzione che restituisce il messaggio d'errore di cui sopra
- `perror`, una funzione che stampa una stringa passata come parametro e ne accoda il messaggio d'errore di cui sopra

10 File descriptor

Utilizziamo le seguenti funzioni per gestire i file tramite file descriptor:

- `open`: apre un file, prendendo come parametri un riferimento al file, una o più flag che specificano la modalità d'apertura e una o più flag che specificano i permessi. Le flag possono essere combinate tra loro tramite "or" bitwise.

  Le flag utilizzabili sono `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT` e `O_EXCL`. Le ultime due rispettivamente creano se non esiste il file ed evita di sovrascrivere se esiste.

  Per i permessi, è possibile utilizzare le flag apposite oppure utilizzare la sintassi Unix classica.

  ```
  #include <fcntl.h>
  #include <sys/types.h>
  #include <sys/stat.h>

  int open (const char* Path, int flags, int mode);
  ```

- `close`: chiude un file descriptor già aperto.

- `read`: legge al file puntato da un file descriptor mettendo `cnt` byte in `buf` e restituisce il numero di byte effettivamente letti, 0 se ha finito oppure -1.

  ```
  #include <fcntl.h>
  size_t read (int fd, void* buf, size_t cnt);
  ```

- `write`: scrive su unfile puntato da un file descriptor mettendo `cnt` byte presenti in `buf` e restituisce il numero di byte effettivamente scritti, 0 se ha finito oppure -1.

  ```
  #include <fcntl.h>
  size_t write (int fd, void* buf, size_t cnt);
  ```

- `lseek`: sposta il cursore alla posizione `offset`contata da `whence`. Quest'ultimo parametro può assumere i valori `SEEK_SET` (conta dall'inizio), `SEEK_CUR` (conta dalla posizione corrente), `SEEK_END` (conta dalla fine).

  In ogni caso restituisce la posizione del cursore.

  ```
  #include <sys/types.h>
  #include <unistd.h>
  off_t lseek(int fd, off_t offset, int whence);
  ```

- `dup/dup2`: creano una copia del file descriptor con indice più piccolo possibile (`dup`) o indice dato (`dup2`).

  ```
  #include <unistd.h>
  int dup(int oldfd);
  int dup2(int oldfd, int newfd);
  ```

11 Pipe

Il piping connette l'output di un comando all'input di un altro. I due processi vengono eseguiti in maniera concorrente e fanno uso di un buffer comune.

Esistono due tipi di pipe: quelle anonime (usate dalla shell) e quelle nominative (usate su file).

11.1 Pipe anonime

Le pipe anonime uniscono due processi con antenato comune in maniera unidirezionale. Viene usato un buffer comune con due lati d'accesso, uno in sola scrittura e uno in sola scrittura.

La syscall è `int pipe (int fd[2])` che imposta due file descriptor, `fd[0]` per il lato di lettura e `fd[1]` per il lato di scrittura. A questo punto è possibile utilizzare le syscall per i file descriptor viste in precedenza. Importante notare come il buffer, avendo capacità limitata, si limiterà a restituire tutti i dati disponibili in lettura anche se ne vengono richiesti di più. Inoltre, se è vuoto il processo si sospende fino all'arrivo di nuovi dati o alla chiusura.

Il lato di scrittura, invece, invia un `SIGPIPE` al processo se il lato di lettura è stato chiuso; generalmente, prova ad effettuare scritture atomiche se e solo se il numero di byte scritti è minore dello spazio disponibile.

Generalmente, una volta aperta una pipe, viene eseguito un fork e i due processi P_1, P_2 chiudono il lato che a loro non interessa.

11.2 Pipe con nome

Le pipe con nome vengono gestiti con file speciale, non hanno vincoli di gerarchia e sono persistenti. In ogni caso sono trattati come file dal sistema e le syscall viste in precedenza sono perfettamente funzionanti.

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```