$ man laboratorio

Giacomo Bertelli

Laboratorio di Sistemi Operativi, 2021-2022

Per appunti più discorsivi vedere ***Appunti di laboratorio di Giacomo Sansone - A.A.2020-2021.md***

Laboratorio 1

Lezione fatta nel corso di Reti Informatiche

Laboratorio 2

- **UID**: user id
- **GID**: group id
- `passwd`: permette di cambiare password e usa il permesso **SUID**
- `id [username]`: mostra **UID** e gruppi dell'utente corrente o di quello selezionato
- `groups [username]`: mostra gruppi dell'utente corrente o di quello selezionato
- `adduser username`: crea l'utente username
- `deluser username`: elimina l'utente username

- `su [username]`: accede al terminale di username o di root se non è specificato
- `sudo comando`: esegue `comando` con i privilegi di root
  - `-u username`: esegue `comando` con i privilegi di username

Permessi

- **SUID**: Durante l'esecuzione il processo acquisisce i privilegi del proprietario del file
- **SGID**: Durante l'esecuzione il processo acquisisce i privilegi del gruppo proprietario del file
- `r` o `4`: lettura se un file o visualizzazione della lista dei file se una cartella
- `w` o `2`: scrittura
- `x` o `1`: esecuzione se un file oppure accesso se una cartella
- `s`: **SUID** (`4`) o **SGID** (`2`)
- `6750`: **SUID** e **SGID** sono attivi (`6`=4+2), il proprietario ha tutti i permessi (`7`=4+2+1), il group owner ha permesso in lettura ed esecuzione (`5`=4+1), gli altri utenti non hanno nessun permesso(`0`)

- `ls -l`: visualizza il contenuto della cartella corrente e i privilegi
- `drwxr-xr-- 2 studenti ingegneri 4096 ott 9 13:19 subDir`
  - `d`: tipo di file `d` = directory, `-` = normale
  - `rwx`: permessi dell'owner
  - `r-x`: permessi per gli appartenenti al group owner
  - `r--`: permessi per gli altri utenti
  - `studenti`: nome dell'utente proprietario
  - `ingegneri`: nome del gruppo proprietario

- `chmod`: modifica i permessi di uno o più files
  - `-R`: in modo ricorsivo per tutto il contenuto di una cartella
  - `chmod 755 file`
  - `chmod [who][how][which] fileName`:
    - `u`: owner
    - `g`: group owner
    - `o`: others

    * `+`: aggiungere permessi
    * `-`: togliere permessi
    * `=`: assegnare permessi
    * `chmod go-rwx file`: toglie tutti i permessi agli utenti del gruppo del proprietario e a tutti gli altri

- `chown utente file`: imposta `utente` come proprietario di file, eseguibile solo da root
- `chgrp gruppo file`: imposta `gruppo` come gruppo proprietario di file, può essere eseguito solo da root o da un membro di `gruppo`

Esempi (presi dagli esercizi)

1. Avendo la cartella `visibile` con i seguenti permessi: `drwxr-xr-x 3 studenti studenti 4096 ott 11 15:58 visibile`, togliere il permesso di esecuzione (proprietario) usando la rappresentazione simbolica e ottale.
    - Simbolica: `chmod u-x visibile/`
    - Ottale: `chmod 655 visibile/`

Laboratorio 3

- `etc/passwd`: contiene informazioni pubbliche sugli utenti
    - `man 5 passwd`
    - struttura: `username:password:UID:GID:info_aggiuntive_utente:home_utente:shell`
- `etc/shadow`: contiene informazioni sensibili
    - `man passwd`
    - struttura:
      `username:<$algoritmo_di_hashing$salt$hash(salt+password):`
      `ultima_modifica:età_min:età_max:avviso:::`

Gestione dei gruppi

- `addgroup group`: crea un gruppo chiamato group, eseguibile solo da root
- `delgroup group`: rimuovo il gruppo chiamato group, eseguibile solo da root
- `gpasswd -a user group`: aggiunge l'utente user al gruppo group
- `gpasswd -d user group`: rimuove l'utente user dal gruppo group
- `gpasswd -M user1, user2,... group`: definisce i membri di group
- `gpasswd -A user1, user2,... group`: definisce gli amministratori di group
- `gpasswd group`: impostare/cambiare la password del gruppo
- `gpasswd -r group`: rimuovere la password del gruppo
    - se la password non è impostata solo i membri del gruppo possono averne i privilegi
- `newgrp group`: permette a un utente di assumere i privilegi del gruppo group se questo ha una password impostata
- per i gruppi esistono i file analoghi a quelli per gli utenti visti sopra

---

Laboratorio 4

Find

- `find path test azioni`: cerca in path i file il cui nome rispetta i test, poi eventualmente esegue una o più azioni sui file trovati

Test

- gli elementi di un'espressione di default sono messi in AND, si possono usare `-o` per l'*OR*, `-a` per l'*AND* e `!` per il *NOT*
- `-name pattern`: ricerca basata sul nome del file che può includere anche metacaratteri:
  - `*`: uno o più caratteri qualsiasi, oppure una o più volte l'espressione precedente
  - `[abc]`: uno tra i caratteri `a`, `b` e `c`
  - I pattern devono essere racchiusi tra apici
- `-type [dfl]`: tipo di file directory, file o symbolic link
- `-size [+-]n[ckMG]`: dimensione del file maggiore (`+`) o minore (`-`) di `n` byte(`c`), kilobyte(`k`), megabyte(`M`) o gigabyte(`G`)
- `-user`: il file appartiene a user (**username** o **PID**)
- `-group`: il file appartiene a group (**group** o **GID**)
- `-perm [-/]mode`: permessi del file (modalità ottale o simbolica)
  - `mode`: i permessi devono essere quelli specificati
  - `-mode`: almeno i permessi specificati devono essere presenti
  - `/mode`: almeno uno dei permessi specificati deve essere presente

Azioni *Le azioni devono SEMPRE essere messe alla fine*

- `-delete`: i file trovati vengono eliminati, ritorna true in caso di successo
- `-exec command \;`: esegue `command` su tutti i file trovati a partire dalla directory corrente
  - Si può usare la stringa `{}` per indicare il nome del file trovato
- `execdir`: come `exec` ma a partire dalla directory in cui si trova il file

Locate

- `locate [options] file1...`: come find ma senza azioni e cerca in un database aggiornato periodicamente
  - root può forzare l'aggiornamento con `updatedb`

Ricerca nel testo

- `grep [opzioni] [-e] modello [-e modello2...] file1 [file2...]`: cerca in uno o più file di testo le righe che corrispondono ai modelli:
  - `-i`: case insensitive
  - `-v`: mostra le righe che non contengono l'espressione
  - `-n`: mostra il numero di linea
  - `-c`: mostra solo il numero di righe trovate
  - `-w`: trova solo parole intere
  - `-x`: trova solo righe intere
  - `'^'`: inizio della riga
  - `'$'`: fine della riga
  - `'.'`: qualsiasi carattere
  - `'*'`: l'espressione precedente può essere ripetuta 0 o più volte

Archiviazione

- `tar modalità[opzioni] [file1...]`: archivia/estrae una raccolta di file e cartelle

Formati

- `.tar`: nessuna compressione
- `.tar.gz`: archivio compresso con gz
- `.tar.bz2`: archivio compresso con bz2

Azioni

- `A`: aggiungi file *tar* all'archivio
- `x`: estrai file dall'archivio
- `c`: crea nuovo archivio
- `d`: trova differenze tra archivio e file system
- `--delete`: cancella file dall'archivio
- `r`: aggiungi file all'archivio
- `t`: elenca file di un archivio
- `u`: aggiungi file all'archivio ma solo se sono diversi da quelli già presenti

Opzioni

- `f`: permette di specificare il nome dell'archivio (**va messo sempre**)
- `z`: compressione con *gzip*
- `j`: compressione con *bzip2*
- `v`: verbose

Compressione

- `gzip file1 file2 ...`: I file elencati vengono compressi e salvati in file con lo stesso nome ed estensione `.gz`. I file non compressi vengono eliminati
- `gunzip file1.gz file2.gz ...`: Estrae i file compressi specificati con lo stesso nome (senza l'estensione della compressione). I file compressi vengono poi eliminati
- `bzip2` e `bunzip2`: come sopra ma con algoritmo *bzip2*

Esempi (presi dagli esercizi)

1. Trovare tutti i file con estensione `.txt` in `/usr/share/docutils` con dimensione superiore a 10 Kilobyte e per ogni file trovato fare in modo che venga mostrato l'output di `ls -l` eseguito dal path in cui si trovano i file.
   - `find /usr/share/docutils/ -name '*.txt' -size +10k -execdir ls -l {} \;`
2. Trovare in `/etc/passwd` le righe relative a UID da 102 a 105
   - `grep '^[a-z][-a-z0-9_]*:x:10[2-5]:' /etc/passwd`

Laboratorio 5

`pid_t` è un tipo opaco che rappresenta un *process id*

Creazione di processi

- `pid_t fork(void)`: crea un processo figlio e restituisce:
  - **Al padre**: il PID del figlio

  - **Al figlio**: `0`
  - Un valore `<0>` in caso di errore
  - Dopo la fork sia padre che figlio continuano a eseguire dall'istruzione successiva alla `fork`
- `pid_t getpid()`: restituisce il PID del processo
- `pid_t getppid()`: restituisce il PID del processo padre

Terminazione di processi

- *Involontaria*: tramite l'esecuzione di azioni illegali o per arrivo di segnali
- *Volontaria*: con la system call `exit()`

- `void exit(int status)`: termina l'esecuzione del processo e permette di comunicare al padre lo status (il padre deve usare la `wait`)
- `pid_t wait(int *status)`:
  - sospende il padre se tutti i figli sono ancora in esecuzione
  - quando uno dei figli termina (o subito se almeno uno era in stato zombie):
    * scrive in `status` lo stato di terminazione del figlio
    * restituisce il PID del figlio terminato
  - restitusce un valore negativo se non ci sono figli
  - per gestire status ci sono due macro (definite in `<sys/wait.h>`):
    * `WIFEXITED(status)`: restituisce `true` se il figlio è terminato volontariamente
    * `WEXITSTATUS(status)`: restituisce lo stato di terminazione

Sostituzione del codice

- `int execl(char *path, char *arg0, ..., char *argN, (char*)0)`:
  - La lista dei parametri può essere di lunghezza variabile ma deve terminare con un puntatore nullo
  - `path`: percorso del comando
  - `arg0`: nome del programma da eseguire
  - `arg1 ... argN`: argomenti del comando
  - In caso di successo la chiamata di `execl` non ha ritorno, in caso di fallimento viene eseguito il codice successivo

Altri comandi

- `sleep(interval)`: sospende il processo per `interval` secondi (vedere Laboratorio 6)
- `int main(int argc, char** argv)`:
  - `argc`: numero di argomenti passati
  - `argv`: array con gli argomenti passati

Laboratorio 6

- `signal.h`: contiene la lista dei segnali
- `man 7 signal`: informazioni sui segnali
  - **SIGHUP**: **1** Hang up (il terminale è stato chiuso)
  - **SIGINT**: **2** Interruzione del processo. CTRL+C da terminale.
  - **SIGQUIT**: **3** Interruzione del processo e core dump. CTRL+ da terminale.
  - **SIGKILL**: **9** Interruzione immediata. Questo segnale non può essere ignorato ed il processo che lo riceve non può eseguire operazioni di chiusura
  - **SIGTERM**: **15** Terminazione del programma.
  - **SIGUSR1**: **10** Definito dall'utente. Default: termina processo.

  - **SIGUSR2**: **12** Definito dall'utente. Default: termina processo.
  - **SIGSEGV**: **11** Errore di segmentazione
  - **SIGALRM**: **14** Il timer è scaduto
  - **SIGCHLD**: **17** Processo figlio terminato, fermato, o risvegliato. Ignorato di default.
  - **SIGSTOP**: **19** Ferma temporaneamente l'esecuzione del processo: questo segnale non può essere ignorato
  - **SIGTSTP**: **20** Sospende l'esecuzione del processo. CTRL+Z da terminale.
  - **SIGCONT**: **18** Il processo può continuare, se era stato fermato da SIGSTOP o SIGTSTP.

System call per i sengali

- `signal`: permette di definire la funzione che deve gestire il segnale
  - `sighandler_t signal(int sig, sighandler_t handler)`
    - `handler` può valere anche **SIG_IGN** (ignora il segnale) o **SIG_DFL** (ripristina azione di default)
    - Restituisce un puntatore al precedente handler del segnale, **SIG_ERR** in caso di errore
  - Il figlio eredita dal padre le informazioni relative alla gestione dei segnali
- `int kill(pid_t pid, int sig)`: invia il segnale `sig` al processo `pid`
  - `pid > 0`: il segnale viene inviato a `pid`
  - `pid == 0`: il segnale viene inviato a tutti i processi nello stesso process group del chiamante
  - `pid == -1`: il segnale viene inviato a tutti i processi a cui il chiamante può inviare sengali
  - `pid < 1`: il segnale viene inviato ai processi il cui process group è `-pid`
  - ritorna `0` in caso di successo
- `unsigned int sleep(unsigned int seconds)`: mette il processo nello stato sleep per seconds secondi o finché non arriva un sengnale non ignorato
  - passato il tempo viene inviato il sengnale **SIGALARM**
- `unsigned int alarm(unsigned int seconds)`: Provoca la ricezione di un segnale **SIGALARM** dopo seconds secondi
  - Un eventuale "allarme" invocato precedentemente viene cancellato
  - Se "seconds" è zero, viene eliminato un eventuale "allarme" invocato precedentemente

Gestione processi da terminale

- `kill -SEGNALE pid`: invia SEGNALE al processo pid
  - Il segnale di default è **SIGTERM**
  - `-l`: mostra l'elenco dei segnali disponibili
  - Un processo utente può inviare sengali solo ai processi di cui è proprietario
- `ps`: mostra i processi in esecuzione
  - `-u utente`: visualizza i processi di utente
  - `u`: formato output utile all'analisi dell'utilizzo delle risorse
  - `a`: processi di tutti gli utenti
  - `x`: anche processi non generati da terminali
  - `o`: solo campi specificati in seguito
  - `-O`: mostra i campi specificati di seguito, oltre ad alcuni campi di default
  - **Stati**:
    - `S`: sleep
    - `T`: bloccato
    - `R`: running
    - `Z`: zombie

Laboratorio 7

- `pstree`: visualizza l'albero dei processi
- **process group id**: un processo ha un process group id, se genera dei figli anche loro hanno lo stesso process group id
- **PID**: id univoco del processo
- **PPID**: id del processo padre
- **PGID**: id del process group del processo
- **RUID**: real user id, id dell'utente che ha mandato in esecuzione il processo
- **EUID**: effective user id, diverso da **RUID** se il comando eseguito ha il bit **SUID** attivo
- **RGID**: real group id, id dell'utente che ha mandato in esecuzione il processo
- **EGID**: effective group id, diverso da **RGID** se il comando eseguito ha il bit **SGID**

Niceness

- Più è alto il valore di **nice** meno priorità ha il processo, di default è 0, un utente normale non può diminuire il valore di niceness
  - `nice -n valore_nice comando`: `valore_nice` != 0, solo root può assegnare valori < 0. Mette in esecuzione comando con niceness = `valore_nice`
  - `renice valore_nice PID`: cambia la niceness di PID, solo root la può abbassare
- `jobs`: visualizza la tabella dei jobs, processi padre e figlio o una pipeline di comandi hanno lo stesso job id
- `comando &`: mette in esecuzione `comando` in background
- **CTRL-Z**: mette in pausa il job in foreground
- `fg JOB_ID` e `bg JOB_ID`: fanno partire, rispettivamente in foreground e background, il job JOB_ID, se non metto JOB_ID metto in esecuzione l'ultimo fermato
- `kill JOB_ID`: invia `SIG_TERM` a `JOB_ID`
- `nohup` e `disown`: eliminano la connessione tra il job e il terminale, se chiudo il terminale non termino il processo in background
  - `nohup comando`: il job è immune a **SIGHUP**, non ha accesso a stdin, stdout è rediretto su `nohup.out`
  - `disown %JOB_ID`: il job è immune a **SIGHUP**, vanno gestiti a mano gli accessi a *stdin* e *stdout* con `>` e `<`
- `top`: visualizza e permette di interagire con i processi in modo interattivo, il campo `NI` è la niceness
  - `d`: intervallo di aggiornamento
  - `k`: invio di un sengale
  - `n`: numero di processi da visualizzare
  - `r`: renice
  - `u`: utente da visualizzare
  - `q`: quit
- `time comando`: esegue `comando` e poi mostra il tempo impiegato ad eseguirlo

---

Laboratorio 8

- Includere la libreria `#include <pthread.h>`
- Compilare con il seguente comando: `gcc <opzioni> file.c -lpthread -std=c99`

- `pthread_t`: tipo analogo a `pid_t`
- `pthread_t pthread_self(void)`: restituisce il `pthread` del thread corrente
- `pthread_equals(tid1, tid2)`: restituisce `true` se i tid sono uguali

Creazione di un thread

```
int pthread_create(pthread_t* thread,
  const pthread_attr_t* attr,
  void* (*start_routine)(void *),
  void* arg );
```

- `pthread_t* thread`: verrà sovrascritto con il *TID* del thread creato
- `const pthread_attr_t* attr`: Attributi del thread, `NULL` per valori di default
- `void* (*start_routine)(void *)`: puntatore alla funzione col codice del nuovo thread
- `void * arg`: puntatore passato come argomento a `start_routine`
- In caso di successo la funzione restituisce `0`

Esempi

1. `main`:

```
int main () {
  pthread_t tr1, tr2;
  int* arg1 = (int*)malloc(sizeof(int));
  int* arg2 = (int*)malloc(sizeof(int));
  *arg1 = 1;
  *arg2 = 2;
  int ret;
  ret = pthread_create(&tr1, NULL, tr_code, arg1);
  if (ret){
    printf("Error: return code from pthread_create is %d\n", ret);
    exit(-1);
  }
}
```

2. Corpo del thread:

```
void* tr_code(void* arg) {
  printf("Hello World! My arg is %d\n", *(int*)arg);
  free(arg);
  pthread_exit(NULL);
}
```

Terminazione e join

- `void pthread_exit(void* retval)`: termina l'esecuzione di un thread
  - Il sistema libera le risorse allocate
  - Permette ai figli di continuare l'esecuzione. **Se un thread termina senza chiamare la `pthread_exit` anche i figli vengono terminati**
  - `void *retval`: Valore di ritorno del thread (exit status) consultabile da altri thread che utilizzano la pthread_join

- `int pthread_join(pthread_t thread, void** retval)`: il processo che la invoca si blocca in attesa della terminazione di un processo specifico
  - `pthread_t thread`: *TID* di cui attendere la terminazione
  - `void **retval`: Puntatore al puntatore dove verrà salvato l'indirizzo restituito dal thread con la pthread_exit. Può essere impostato a `NULL` (in questo caso viene ignorato)
  - In caso di successo la funzione restituisce `0`, sennò un codice di errore

Mutua esclusione

- `pthread_mutex_t`: semaforo binario (libero/occupato) con una lista di thread in attesa che si liberi
- `int pthread_mutex_init(pthread_mutex_t* M, const pthread_mutexattr_t* mattr)`: inizializza il mutex puntato da `M` con gli attributi `mattr`. Se `mattr == NULL` inizializza un mutex libero
- `int pthread_mutex_lock(pthread_mutex_t* M)`: equivalente della *wait*. Ritorna `0` in caso di successo
- `int pthread_mutex_unlock(pthread_mutex_t* M)`: equivalente della *signal*. Ritorna `0` in caso di successo

---

Laboratorio 9

Variabili condizione

- `pthread_cond_t`: tipo opaco che rappresenta una variabile condizione
- `int pthread_cond_init(pthread_cond_t* C, pthread_cond_attr_t* attr)`: inizializzazione di una vairabile condizione
  - `pthread_cond_t* C`: Puntatore alla variabile condizione da inizializzare
  - `pthread_cond_attr_t* attr`: Attributi per la condizione, `NULL` per attributi di default

Wait

- `int pthread_cond_wait(pthread_cond_t* C,pthread_mutex_t* M)`:
  - `pthread_cond_t* c`: variabile condizione su cui sospendersi
  - `pthread_mutex_m_t* m`: Mutex associato alla variabile condizione: viene liberato automaticamente quando il thread si sospende; viene eseguito un nuovo lock quando il thread viene risvegliato.

Signal

- `int pthread_cond_signal(pthread_cond_t* C)`:
  - Se esistono thread in coda sulla condition, (almeno) uno viene risvegliato
  - Se non ci sono thread sospesi non succede nulla
  - La politica della *signal* della libreria pthread è di tipo **signal & continue**

Broadcast

- `int pthread_cond_broadcast(pthread_cond_t* C)`:
  - Come la signal ma risveglia **tutti** i thread in coda sulla condition `C`

Esempi

1. **Produttori e consumatori**

Più thread accedono a un buffer circolare condiviso con capacità limitata.

- *Risorsa*

```
typedef struct {
  int buffer[BUFFER_SIZE];
  int readInd, writeInd;  // indici di read/write nel buffer
  int cont;               // elementi nel buffer
  pthread_mutex_t M;      // per garantire accesso esclusivo alle risorse
  pthread_cond_t FULL;    // condition var. buffer pieno
  pthread_cond_t EMPTY;   // condition var. buffer vuoto
} risorsa;
```

- *Inizializzazione risorsa*

```
risorsa r; // variabile globale, condivisa da tutti i thread
int main() {
  pthread_mutex_init(&r.M, NULL);
  pthread_cond_init(&r.FULL, NULL);
  pthread_cond_init(&r.EMPTY, NULL);
  r.readInd = r.writeInd = r.cont = 0;
  ...
}
```

- *Consumatore*

Il consumatore deve controllare che il buffer non sia vuoto prima di prelevare e risvegliare un eventuale produttore dopo averlo prelevato

```
...
int val; // per il dato che verrà prelevato dal buffer

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

// Preleva un dato e aggiorna lo stato del ring buffer
val = r.buffer[r.readInd];
r.cont--;
r.readInd = (r.readInd+1) % BUFFER_SIZE; // gestione circolare

// Risveglia un eventuale thread produttore
pthread_cond_signal(&r.FULL);
pthread_mutex_unlock(&r.M);
...
```

- *Produttore*

Il produttore deve controllare che il buffer non sia pieno prima di inserire un dato e risvegliare l'eventuale consumatore sospeso

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

// Inserisci un dato e aggiorna lo stato del ring buffer
```

```
r.buffer[r.writeInd] = val;
r.cont++;
r.writeInd = (r.writeInd+1) % BUFFER_SIZE; // gestione circolare

// Risveglia un eventuale thread consumatore
pthread_cond_signal(r.EMPTY);
pthread_mutex_unlock(&r.M);
...
```

2. **Accesso limitato a una risorsa**

Una risorsa può essere usata contemporaneamente da un massimo di `MAX_T` threads

- *Variabili globali*

```
#DEFINE MAX_T 10

int n_users = 0;       // numero di thread che stanno usando la risorsa
pthread_cond_t FULL;   // condition var. limite di utilizzo raggiunto
pthread_mutex_t M;     // Mutex per l'accesso esclusivo a n_users
```

- *Fase di ingresso*

```
...

pthread_mutex_lock(&M);
while (n_users == MAX_T)           // massimo numero di users raggiunto
  pthread_cond_wait(&FULL, &M);    // attendi...

n_users++;
pthread_mutex_unlock(&M);          // rilascio il lock

...
// Uso della risorsa
...
```

- *Fase di uscita*

```
...
// Uso della risorsa
...

pthread_mutex_lock(&M);
n_users--;
pthread_cond_signal(&FULL);
pthread_mutex_unlock(&M);
```

Laboratorio 10

Accesso ai file

- `int open(const char* path, int flags)`: apertura di un file
  - `const char *path`: path del file da aprire
  - `int flags`: indicano la modalità con cui si vuole accedere al file, ci sono delle macro definite in `<fcntl.h>` che possono essere messe in `OR`:
  - `O_RDONLY`

  - `O_WRONLY`
  - `O_RDRW`
  - `O_APPEND`
  - vedere `man 2 open` per altri dettagli
- `ssize_t read(int fd, void* buf, size_t count)`: lettura di un file
  - `int fd`: descrittore del file da cui leggere
  - `void *buf`: puntatore al buffer in cui scrivere i dati letti
  - `size_t count`: numero di byte da leggere (`> 0`)
  - restituisce il numero di byte letti o un valore negativo in caso di errore
- `ssize_t write(int fd, const void* buf, size_t count)`: scrittura su file
  - `int fd`: descrittore del file su cui scrivere
  - `void *buf`: buffer da cui leggere i dati da scrivere
  - ‘size_t

Pipe

- `int pipe(int fd[2])`: creazione di un pipe
  - `int fd[2]`: contiene i due descrittori del pipe
    - `df[0]`: descrittore per la lettura
    - `df[1]`: descrittore per la scrittura
  - restituisce `0` in caso di successo e `-1` altrimenti