Laboratorio sistemi operativi

Un riassunto di
Paolo Alfano

Note Legali

Liberatoria, aggiornamenti, segnalazione errori:

Quest'opera viene pubblicata in formato elettronico senza alcuna garanzia di correttezza del suo contenuto. Il documento, nella sua interezza, è opera di Paolo Didier Alfano
e viene mantenuto e aggiornato dallo stesso, a cui possono essere inviate eventuali segnalazioni all'indirizzo *paul15193@hotmail.it*
Ultimo aggiornamento: 1 febbraio 2016

Contents

Introduzione

Il seguente riassunto vuole ripercorrere velocemente gli argomenti trattati nel corso di laboratorio di sistemi operativi. La struttura del testo è la seguente:

- Per ogni lezione esiste una sezione(o capitolo)
- Ad ogni file di slides presentate, corrisponde una sottosezione. Se la sottosezione non contiene niente, significa che in quel file non vi era nulla di interessante.
- Inoltre ogni capitolo contiene un'ultima sezione che indica quali funzioni utili sono state introdotte nella lezione, per fare il punto della situazione

1 Introduzione al linguaggio C

1.1 Puntatori e dintorni

In C è possibile conoscere e denotare la cella di memoria in cui è contenuta una variabile. Per farlo è sufficiente dichiarare:

```
int a = 5
int* b = &a
```

Questo è utile perché posso conoscere e modificare il valore di una variabile manipolando direttamente il puntatore a tale variabile. Ad esempio eseguendo

```
*b = *b + 4;
```

posso modificare direttamente la variabile puntata da b.
Il puntatore nullo è una costante definita in *stdio.h*(vedi sezione 2.2) denotata tramite NULL.
Ovviamente con i puntatori è possibile effettuare tutte le tipiche operazioni come creazione di *aliasing* o lo scorrimento di un array visto che gli stessi array sono raggiungibili tramite la notazione `a`$[i]$ che in realtà è equivalente alla notazione $*(a+i)$.
Ad esempio sono equivalenti i due codici seguenti che salvano nella variabile sum la somma degli elementi di un array:

```
/* versione 1 */
for(i=0;i<N;i++)
sum+ = a[i];
```

```
/* versione 2 */
for(i=0;i<N;i++)
sum+ = *(a+i);
```

Tutte queste operazioni fanno parte della cosidetta *aritmetica dei puntatori*.
In particolare ricordiamo che anche le stringhe (che in C sono array di caratteri) sono visitabili tramite puntatori ed è per questo che bisogna prestare attenzione alle porzioni di memoria che vanno a toccare e alla loro gestione. Ad esempio non porre il carattere terminatore '\0' in una stringa impedisce al programma di comprendere dove la stringa termini. Questo può portare a letture, o ancor peggio scritture, errate in memoria generando situazioni di *buffer overrun*, che nel migliore dei casi vengono segnalati tramite un segfault, ma in altri casi è possibile che vadano a danneggiare lo spazio di memoria di altre variabili. Per evitare situazioni di questo tipo è possibile utilizzare funzioni che stabiliscono la dimensione del buffer come la *strncopy*.

1.2 Puntatori a funzione e computazioni generiche

Consideriamo la funzione:
`int somma (int x, int y){return x+y;}`
in realtà *somma* è un puntatore costante alla zona di memoria contenente il codice della funzione.
Invece di chiamare la funzione potremmo dichiarare ed utilizzare:

```
int (*fun) (int,int);
fun = somma;
a = fun(3,5);
```

il puntatore *fun* è un *puntatore a funzione*. Lo scopo dei puntatori a funzione è quello di definire le *funzioni di ordine superiore* ovvero funzioni che prendono come argomento altre funzioni. Un tipico esempio di funzione di ordine superiore è la *map* che applica a tutti gli elementi di un array(x nell'esempio) una certa funzione(fun nell'esempio):

```
void map (int (*fun) (int),int x[], int l){
  for(i=0;i<l;i++) x[i] = fun(x[i]);}
```

Un modo un po' brutale per implementare il polimorfismo è quello di utilizzare un puntatore "generico" *void**. Per deferenziarlo deve prima essere castato ad un tipo noto:

```
void* c;
c = &a;
int a;
*c = 5; /*scorretto*/
*(int *)c = 5; /*corretto*/
```

Fanno uso di puntatori a void tutte le funzioni di allocazione e deallocazione della memoria come *malloc* e *free*, oppure la funzione *qsort*.

1.3 Gestione della memoria

In C la memoria può essere allocata staticamente per quel che riguarda le variabili globali e statiche, oppure automaticamente sullo stack, quando dichiariamo le variabili, o ancora dinamicamente tramite le funzioni malloc, calloc, realloc e free. Le funzioni per l'allocazione restituiscono un puntatore alla

zona di memoria allocata oppure NULL in caso di fallimento.

1.4 Struct e union

In C è possibile allocare strutture dati composte da vari campi. Nelle struct è anche possibile specificare l'ampiezza in bit di ogni campo(*bit-fields*). Questa però è un'operazione rischiosa in quanto:

- riduce la portabilità da un'architettura ad un'altra
- dobbiamo prestare molta attenzione all'endianess della macchina(si veda la sezione 10.1)[1]

Mentre le struct sono più note le union, come le struct, possono avere diversi campi di diverso tipo ma soltanto uno dei campi alla volta può essere usato. Le union forniscono un modo efficiente di utilizzare le stesse locazioni di memoria per diversi scopi.
In ogni caso condividono diverse altre proprietà con le struct in quanto è possibile fare assegnamenti o copie di intere union, e non posso confrontarle "interamente" ma solo campo per campo.

1.5 Funzioni con numero di argomenti variabili

Le funzioni con un numero variabile di argomenti sono funzioni come la *printf* di cui non sappiamo a prescindere il numero degli argomenti. Una funzione del genere lo rende chiaro inserendo tra gli argomenti tutti gli argomenti "obbligatori" seguiti dalle *ellipsis* "..." che devono per forza essere l'ultimo elemento degli argomenti. All'interno della funzione dovrà essere definita una *va_list* che indica la lista degli argomenti su cui operare, la macro *va_start* che iniziallizza la va_list di modo da puntare il primo elemento. La lista di elementi viene scorsa utilizzando il comando *va_arg* che restituisce il successivo elemento castandolo al tipo specificato come secondo argomento. Infine è necessario chiamare la *va_end* per effettuare le operazioni di cleanup subito prima che la funzione termini. Un altro tipico esempio potrebbe essere una funzione che prende un intero obbligatorio che indica il numero di argomenti interi che lo seguono di cui effettuare la somma:

[1] Consideriamo i tre bit più a sinistra di un numero. Cosa rappresentino dipende direttamente dall'endianess della macchina

```
int vasum(int count, ...) {
  va_list ap;
  va_start(ap, count); //inizializzo la lista
  int sum = 0;
  for(int i=0;i<count;i++) sum+=va_arg(ap, int); //il prossimo
  va_end(ap); // ho finito di usare la lista
  return sum;
}
```

Bisonga però fare attenzione al tipo degli argomenti variabili poiché vengono cambiati durante l'esecuzione. Ad esempio i float vengono promossi a double.

1.6 Funzioni presentate

- **strncpy(char* dest,char* src, size_t n)**: copia il contenuto della stringa src nella stringa dest, copiando non più di n bytes

- **void* malloc(size_t size)**: alloca un blocco di memoria contigua di size bytes. In realtà, spesso ne viene allocata di più per motivi di allineamento tra i dati. Non alloca la memoria all'atto della chiamata ma al momento del primo accesso. Restituisce il puntatore ala zona di memoria in caso di successo. Se il sistema operativo non riesce ad allocare lo spazio in memoria, restituisce NULL.

- **void* calloc(size_t N,size_t size)**:alloca un blocco di memoria contigua di size byte, ma come la malloc spesso ne alloca una quantità maggiore per motivi di allineamento. Alloca effettivamente al momento della chiamata e iniziallizza a zero ogni elemento che alloca.

- **void free(void *ptr)**: dealloca la memoria puntata da ptr. Nel caso in cui venga chiamata su un puntatore non allocato sullo heap oppure già deallocato, può dare comportamenti errati e imprevedibili a run time.

2 Richiami di C: funzioni e libreria I/O

2.1 Argomenti del main

Il main nel C è una funzione come le altre con qualche differenza.
Un programma C si avvia sempre dal main, dunque deve essere sempre presente. Inoltre ha due tipiche segnature:

`int main(void)`	main senza argomenti
`int main(int argc, char *argv[])`	main con argc argomenti

dove argc è il numero di argomenti del main. Ricordiamo che il main ha sempre almeno un argomento, il nome del programma. Dunque argc è sempre maggiore di 1. Invece argv è un array di stringhe,contenente gli argomenti che ha come ultima entry NULL.
Infine, il main restituisce un intero, che è zero se il programma è andato a buon fine, altrimenti non è noto. Per valutare l'esito di un programma è possibile vedere che valore ha restituito il main. Per farlo è sufficiente invocare '$?' da shell.

2.2 La libreria stdio.h

Questa libreria contiene definizioni di costanti come EOF(end of file) e le funzioni per interagire con le periferiche di input e i file nel computer. In particolare un file viene aperto tramite la funzione *fopen* che restituisce un puntatore a file. Successivamente posso lavorare sul file tramite le funzioni *fscanf*, *fgets*, *fwrite*... Inoltre stdio.h contiene le definizioni di strutture per file speciali come `FILE* stdin`(la testiera), `FILE* stdout` e `FILE* stderr` entrambi per lo schermo.
Inoltre, poiché parte delle funzioni e tutte le chiamate di sistema, in caso di errore settano una particolare variabile *errno*, nella libreria stdio.h troviamo una funzione, *perror* che converte tale valore in un messaggio che viene stampato a schermo(su stderr!). Ad esempio, definiamo il seguente codice che controlla se esista il file pippo provando ad aprirlo:

```
if ((fp=fopen("pippo","r")) == NULL) {
  perror ("Aprendo pippo");
  return -1;
}
```

Provando ad eseguire il programma in una cartella dove non esista il file pippo, otteniamo il seguente risultato:

```
Aprendo pippo:  No such file or directory
```

2.3 Funzioni: frame di chiamata

Nelle funzioni C tutti i parametri sono passati per valore. Viene fatta una copia che viene messa nello stack all'atto della chiamata che verrà rimossa quando la funzione termina. Insieme ai parametri le funzioni possiedono le proprie variabili locali ed è molto pericoloso restituire i puntatori alle variabili locali all'esterno della funzione.

2.4 Funzioni presentate

- **FILE* fopen(char* path, char* mode)**: apre il file raggiungibile tramite path, in modalità mode. Le modalità previste sono diverse, lettura(r), scrittura(w), append(a)..
  Restituisce il puntatore al file in caso di successo. Altrimenti restituisce NULL e setta errno.

- **char* fgets(char* s, int size, FILE* stream)**: legge fino a size byte, o fino a quando non incontra un terminatore di riga dal file puntato da stream, salvandolo in un buffer s. Si differenzia dalla *fscanf* perché quest'ultima si ferma appena incontra uno spazio. Restituisce il puntatore al buffer in caso di successo. Restituisce NULL in caso di errore o nel caso in cui legga un EOF senza aver letto niente.

- **int fflush(FILE* ifp)**: poiché l'output delle funzioni come printf e fprintf viene bufferizzato, la fflush costringe allo svuotamento dei buffer utente relativi al file puntato da ifp. Se ifp è NULL, svuota tutti i buffer. Restituisce 0 in caso di successo. Altrimenti restituisce EOF e setta errno.

- **int fclose(FILE* fp)**: effettua un flush dei buffer relativi ad fp(tramite la fflush) e chiude il file associato a fp. Restituisce 0 in caso di successo. Altrimenti restituisce EOF e setta errno.

- **void rewind(FILE* fp)**: riporta all'inizio del file il puntatore associato al file. Non ha un valore di ritorno.

- **void perror(const char* s)**: invocata immediatamente dopo la funzione che ha dato errore converte un valore numerico in un messaggio su stderr (schermo). Non ha un valore di ritorno.

3 Richiami di C: preprocessore, compilazione e linking

3.1 Il preprocessore

Nel C la fase di *preprocessing* viene eseguita prima della compilazione andando ad effettuare delle manipolazioni testuali sul codice. Possiamo vedere il risultato della precompilazione eseguendo da terminale:
`gcc -E nomefile.c`
Le linee di codice che sono direttive al preprocessore iniziano con il simbolo '#'. Vediamone alcune:

- `#include`: direttiva di inclusione che rimpiazza la riga con il codice del file da includere. La include può essere fatta in due modi:
  `# include <include_file>` va a cercare il file nei percorsi standard contenenti molti file di libreria. Solitamente "usr/include" oppure "usr/local/include"
  Invece `#include "path_include_file"` va a cercare il file nel percorso specificato.

- `#define`: la define serve per creare dei nomi che verranno sostituiti nel codice ogni volta che appaiono. Può essere fatta in tre modi:

  1. #define nome: si definisce un nome senza associarvi un valore. Viene utilizzato nella compilazione condizionale(vedi dopo).
  2. #define nome testo: si definisce un nome a cui viene associato un valore. Viene utilizzato per parametrizzare rispetto alle costanti
  3. #define nome(parametri) testo: caso particolare in cui ad una macro associamo anche dei parametri. Un tipico esempio potrebbe essere `#define PRODOTTO(X,Y) (X)*(Y)` che può essere invocata nel codice con `a = PRODOTTO(a+1,b)`. In ogni caso quando vengono usate define con parametri è sempre bene stare attenti ad inserire le parentesi per mantenere la precedenza degli operatori e

bisogna ricordare che le define, sostituiscono alla lettera nel codice e bisogna prestarvi attenzione. Ad esempio il seguente codice:
`#define QUAD(X) (X)*(X)`
`a = QUAD(scanf("%d",&b);)`
non inserisce in a il quadrato del numero acquisito da tastiera, viene bensì espanso come:
`a = (scanf("%d",&b))*(scanf("%d",&b));`
e dunque la scanf viene eseguita due volte.

- `#if #ifdef #ifndef #endif`: ponendo una delle prime tre direttive all'inizio di una porzione di codice e ponendo alla fine di tale porzione la direttiva endif, il codice compreso tra le due direttive viene compilato se è verificata la condizione che segue la direttiva.
  Un tipico esempio potrebbe essere:

  ```
  #if espr_cond
     code
  #endif
  ```

  In questo caso il codice code viene eseguito solo se espr_cond è verificata. Questo tipo di direttiva viene solitamente utilizzato nelle situazioni di debugging.

Ricordiamo infine che è possibile compilare un file inserendovi una define "al volo" da riga di comando utilizzando `gcc -Dnome=val file.c` che va ad inserire nel codice la seguente linea `#define nome val`

3.2 Preprocessing, compilazione, esecuzione

Ogni volta che andiamo a compilare un file C, otteniamo un eseguibile che ha formato ELF(*Executable and Linking Format*). Un file di questo tipo contiene(vedi anche figura):

- un *magic number* che lo contraddistingue come file ELF
- una tabella dei simboli che contiene tutte le associazioni tra identificatori e loro significato, una tabella di rilocazione e l'indirizzo della prima istruzione(altre info)

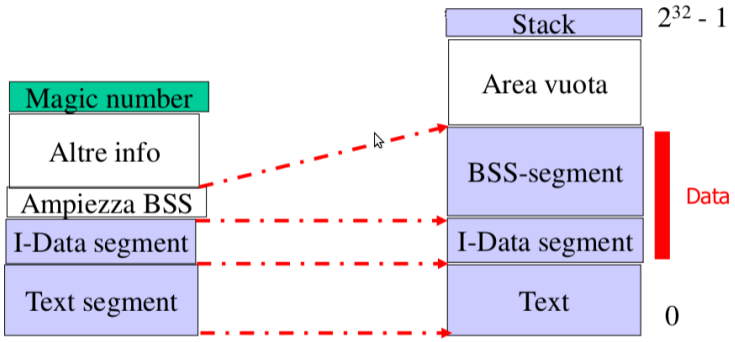

Figure 1: corrispondenza tra il file ELF e la porzione di memoria del programma

- una zona di memoria per le variabili globali non inizializzate(Ampiezza BSS)
- una zona di memoria per quelle iniziallizate(I-Data Segment)
- il codice del programma

Come si può vedere il file ELF contiene tutto quello che ci serve per andare a creare la configurazione iniziale per lo spazio di indirizzamento.
In ogni caso, un programma quando viene compilato segue tre fasi:

1. Preprocessing: vista nel paragrafo 3.1
2. Compilazione: fase in cui, a partire dal codice preprocessato vado ad ottenere il file effettivamente compilato. I file prodotti in questa fase hanno formato .o e sono detti moduli oggetto. Posso anche invocare funzioni che mi restituiscono informazioni sui moduli oggetto, ad esempio la objdump.
3. Linking: fase in cui si collegano insieme più moduli oggetto. Questa fase viene svolta quando dobbiamo unire il nostro codice a quello delle librerie per farlo funzionare. Affinché questo avvenga, dobbiamo linkare le varie parti dei moduli oggetto. In particolare, il testo e i dati dei vari moduli vengono uniti ponendo prima tutti i dati, poi tutti i testi. Mentre per quel che riguarda la tabella dei simboli, ogni oggetto possiede

un certo quantitativo di simboli che possono essere *esportati* se vengono definiti all'interno del proprio modulo ma vengono usati altrove, oppure *esterni* se non sono definiti nel proprio modulo e devono essere cercati altrove. Il compito del linker è quello di risolvere i simboli; se li trova copia il codice della funzione nell'eseguibile. Altrimenti da errore.

4 Librerie e makefile

4.1 Classi di memorizzazione

Le classi di memorizzazione definiscono le regole di visibilità delle variabili e delle funzioni quando un programma è diviso su più file. Stabiliscono dove le variabili vengono allocate e se finiscano nella tabella dei simboli.
In particolare, per le variabili abbiamo i modificatori seguenti:

- *auto*: modificatore di default per la dichiarazione delle variabili. Lo usiamo continuamente visto che le due computazioni `int x;` e `auto int x;` sono equivalenti. La visibilità di x è solo all'interno del blocco
- *register*: modificatore obsoleto che serviva per segnalare al compilatore che quella variabile veniva usata massicciamente. Oggi è un meccanismo non più utilizzato
- *static*: serve quando il valore di una variabile deve essere mantenuto da un'invocazione ad un'altra. La visibilità di variabili static è solo all'interno del blocco. Inoltre se una variabile static viene dichiarata globalmente, non finisce nella tabella dei simboli
- *extern* la variabile è dichiarata globale in questo o ina altri file, viene inserita nella tabella dei simboli e quando viene utilizzata il linker la cerca nel modulo corrente come variabile globale e negli altri moduli

Invece per quel che riguarda le funzioni abbiamo due possibili modificatori:

- *extern*: il modificatore di default delle funzioni. Il nome della funzione finisce nela tabella dei simboli e la funzione può essere chiamata da funzioni di altri file
- *static* la funzione non viene inserita nella tabella dei simboli ed è dunque utilizzabile soltanto all'interno del file. La classe static può essere utilizzata per delle implementazioni "private" all'interno di un file

4.2 Librerie statiche e dinamiche

Una libreria è un file archivio contenente codice compilato che verrà agganciato al programma nella fase di linking. Le librerie esistono fondamentalmente per ottimizzare e ne esistono di due tipi:

- *statiche*: hanno estensione .a su Unix e .lib su Windows e vengono agganciate al codice in fase di compilazione per andare a formare un blocco eseguibile monolitico.

- *dinamiche*: hanno estensione .so su Unix e .dll su Windows e la libreria è agganciata al programma in due fasi. A tempo di compilazione il linker verifica che tutti i simboli siano definiti mentre nella fase di carimento il loader carica le librerie agganciate al programma (quelle necessarie). Una differenza notevole con le librerie statiche è che quelle dinamiche non vanno a formare un blocco monolitico con l'eseguibile. Restano invece separate da quest'ultimo in un file differente. La libreria viene poi cercata a tempo di esecuzione. Su Unix la libreria viene cercata quando se ne invoca una funzione, mentre su Windows viene cercata all'inizio del programma. Se una libreria dinamica richiesta non viene trovata, il loader segnala errore.
  I vantaggi di avere una libreria dinamica sono diversi: possiamo condividerla fra diverse applicazioni, possiamo aggiornare la libreria senza aggiornare l'intero programma che la usa, infine carichiamo dinamicamente solo le funzioni che ci servono(quelle usate).

4.3 Makefile

Il makefile è un file di testo che viene utilizzato per compilare e linkare i file in modo automatico, permettendo di esprimere le dipendenze tra file.
Ad esempio se un file f.o dipende dai file f.c t.h ed r.h diciamo che:

- f.o è il *target*

- f.c t.h ed r.h sono gli elementi della sua *dependency list*

il makefile ci permette di capire cosa fare per aggiornare il target nel caso che uno degli elementi della dependency list venga modificato. La regola di aggiornamento di un target viene detta *makerule*.

Di norma la più semplice regola per un makefile è strutturata come segue[2]:

```
target list :  dependency list
      command 1
      ...
      command n
```

Notiamo che l'ordine delle regole è importante perché l'albero delle dipendenze viene scritto a partire da tale ordine.
Il target della prima regola trovata rappresenta la radice dell'albero e gli elementi della sua dependency list vengono appesi come figli. Successivamente viene controllato se esistono delle regole che hanno come target i figli appesi alla radice. Questo procedimento viene iterato fino a quando non esistono più regole che hanno come target una foglia. Quando succede, l'albero delle dipendenze è completo. L'albero delle dipendenze serve perché ogni volta che viene invocato il makefile controlla i tempi di ultima modifica dei vari file. Se uno dei file da cui dipende il target è stato modificato, ne viene invocata la regola.
Per eseguire il makefile, se il file si chiama "Makefile" basta eseguire il comando *make* da terminale. Altrimenti basta invocare `make -f nomefile`.
Possiamo anche utilizzare dei nomi di variabile per semplificare la struttura del makefile e specificare prima il compilatore e i flags da usare. Ad esempio proviamo a riscrivere la seguente regola:

```
exe:  f.o r.o
      gcc f.o r.o -o exe
```

che viene riscritta come segue

```
CC = gcc
CFLAGS = -Wall -pedantic
objects = r.o f.o
exe:  $(objects)
      $(CC) $(objects) -o exe
```

[2]Affinché i comandi funzionino, ogni riga contenente un comando deve iniziare con un TAB, fra una regola e un'altra deve esserci una riga vuota e il file deve terminare con un newline

Inoltre esistono nel makefile delle convenzioni che possiamo utilizzare, sono indicate come *regole implicite*. Ad esempio nel caso del C il makefile sa che per aggiornare un file .o deve compilare il corrispondente file .c dunque:

```
f.o :  f.c t.h r.h
      $(CC) $(CFLAGS) -c f.c
```

può essere riscritta come

```
f.o:  f.c t.h r.h
```

Esistono anche altri simboli predefiniti, i più usati sono: $@, il nome del target, $ ^ la dependency list e $< il primo nome nella dependency list.
Un'altra funzionalità prevista dal makefile è l'utilizzo di regole il cui target non è legato ad un particolare file ma che servono solo ad eseguire una certa sequenza di azioni. Un esempio è:

```
cleanall:
      @echo "Removing garbage"
      -rm -f *.o core *.~
```

Per utilizzare tale regola basta invocare `make cleanall`.
L'unica cosa a cui stare attenti è che tali regole funzionano fintanto che non esiste un file con il nome della regola nella cartella del makefile. In tal caso avrei una dependency list sempre aggiornata(poiché vuota!) impedendomi di eseguire la command list della regola.
Introdurre regole "fittizie" il cui scopo è solo quello di eseguire azioni è una pratica comune nella programmazione di makefile, infatti i target "falsi" hanno anche un loro nome, ovvero *phony*.
Una buona norma consiste nel dichiarare sempre i target phony esplicitamente come segue:

```
.PHONY : cleanall
cleanall:
      @echo "Removing garbage"
      -rm -f *.o core *.~
```

5 Bash

5.1 Bash, parte 1

La shell è un normale programma a livello utente(non kernel) che interpreta comandi in modo interattivo o non interattivo.
La shell è interattiva quando il comando proviene da un utente che digita da tastiera, mentre è non interattiva quando legge il successivo comando da un file. La shell interattiva(non interattiva) segue sempre uno stesso ciclo in tre fasi:

1. Iniziallizzazione
2. Ciclo principale in cui si richiede un comando da tastiera(si legge un comando da file), lo si interpreta e lo si esegue
3. Terminazione

Noi usiamo i comandi della shell perché ci consentono di capire meglio cosa avviene "sotto" e perché una volta imparati sono rapidi e flessibili.
Notiamo inoltre che i nostri programmi interagiscono sempre con la shell tramite tre canali che ogni programma possiede, stdin, stdout e stderr.
Il tipico comando base in Unix è:

`nome <opzioni> <argomenti>`

dove "opzioni" sono i possibili modi in cui un comando si articola e possono essere *corte* se sono del tipo -a -f -l oppure *lunghe* se sono del tipo –help –version –all.
Nella shell esistono diverse funzionalità che possiamo utilizzare per muoverci più velocemente. Le combinazioni di tasti:

combinazione	risultato
CTRL-a	va a inizio riga
CTRL-e	va a fine riga
CTRL-k	cancella fino a fine linea
CTRL-y	reinserisce la stringa cancellata
CTRL-d	cancella il carattere sul cursore

inoltre, come noto, possiamo autocompletare (quando possibile) il comando che stiamo inserendo premendo il tasto TAB.

In Unix abbiamo anche diversi tipi di file che sono distinguibili tramite un carattere. Infatti esistono diversi tipi di file:

- regular(-): un normale file
- directory(d): una directory
- pipe(p): file associato ad una pipe
- socket(s): file associato ad una socket
- a blocchi(b): una periferica o un dispositivo virtuale su cui posso effettuare operazioni di input/output di quantità di byte predeterminate, tipicamente dispositivi di memoria di massa come i dischi rigidi
- a caratteri(c): una periferica o un dispositivo virtuale su cui è possibile effettuare operazione di input/output anche per singoli byte o comunque per quantità non rigidamente determinate.
- link(l): file contenente un symbolic link

oltre a questo ogni file possiede certi permessi. I permessi possono essere di *lettura* (r), di *scrittura* (w) o di *esecuzione* (x).
Ovviamente questi permessi hanno significato diverso a seconda del file che consideriamo

	Su file	Su directory	Su file speciali
read(r)	lettura	accesso e lettura degli elementi della directory	posso eseguirvi la SC read
write(w)	scrittura	modifica degli elementi nella directory	posso eseguirvi la SC write
exec(x)	esecuzione	possibilità di utilizzo nei path	/

Tutte queste informazioni si ritrovano nell'esecuzione del comando `ls -l`. Ad esempio:

```
ls -l mtrace.txt
-rw-rw-r-- 1 paolo paolo 505 Mar 10 15:10 mtrace.txt
```

Il risultato ottenuto si interpreta come segue. Il primo carattere (-) indica il

tipo del file, regular. I successivi tre caratteri(rw-) indicano i diritti del proprietario del file. I successivi tre caratteri(rw-) indicano i diritti del gruppo. Gli ultimi tre (r–) indicano i diritti di tutti gli altri.
Il numero "1" indica il numero di hard link, i due successivi nomi sono il nome dell'owner e del gruppo. A seguire abbiamo la lunghezza in byte del file(505), la data di ultima modifica(Mar 10 15:10) e infine il nome del file.
Un'altra utile funzionalità nella shell è quella di poter utilizzare metacaratteri. I due principali metacaratteri sono

metacarattere	effetto
?	ogni carattere
*	ogni stringa

chiaramente ne esistono molti altri e metacaratteri possono essere usati in molte situazioni. Ad esempio supponiamo di voler cercare tutti i programmi C in una directory:

```
bash:~$ls *.c
g.c h.c uno.c
```

Se invece avessimo usato l'altro metacarattere:

```
bash:~$ls ?.c
g.c h.c
```

nel caso in cui un comando con metacarattere non trovi alcun file allora restituisce il pattern inserito. Ad esempio:

```
bash:~$echo *.f
*.f
```

Vediamo adesso due ulteriori funzionalità che rendono estremamente flessibile la shell. La prima è la *redirezione*.
Come abbiamo visto, ogni processo ha almeno tre canali predefiniti di comunicazione, stdin, stdout e stderr.
A tali canali sono associati dei descrittori standard:

canale	descrittore
stdin	0
stdout	1
stderr	2

possiamo redirigere tutti e tre i canali come segue

- stdin: viene rediretto così
  `command [n] <filename`

dove n è un descrittore. Se n è assente, la redirezione è automatica su stdin. Un esempio d'uso è il seguente, se invece che acquisire da tastiera volessimo prendere gli input da un file, ci basta digitare:
`bash:~ $sort 0< lista.utenti`

- stdout: viene rediretto così
  `command [n] >filename`
  se n è assente la redirezione è automatica su stdout. Se ad esempio volessimo inviare il risultato di un `sort` su un file ci basterebbe digitare:
  `bash:~ $ ls > dir.txt`

- stderr: analogo allo stdout

L'altra utile funzionalità è l'utilizzo delle pipeline. Digitando il comando `<cmd1> | <cmd2> | ...| <cmdN>` abbiamo in realtà digitato una serie di comandi messi in comunicazione tramite pipe. Ad esempio lo stdout di cmd1 è rediretto sullo stdin di cmd2.
Ad esempio eseguire:
`ps aux | grep ciccio`
restituisce su stdout i processi nel cui nome sia contenuta la stringa ciccio.

Per terminare questa prima parte, notiamo che fino ad ora per ogni comando eseguito veniva creato un processo in *foreground*. Vediamo adesso come creare un processo in background. Per farlo basta eseguire:
`command &`
a questo punto command verrà eseguito in background. Possiamo riportare in foreground l'ultimo comando mandato in background eseguendo il comando *fg*. Inoltre i processi in background possono essere visualizzati tramite il comando *jobs*.
Inoltre un processo viene mandato in background e disattivato se riceve un segnale(capitolo 11) SIGSTOP. Può essere riattivato tramite il comando *bg*.
Per quanto riguarda i segnali, esiste la possibilità di specificare un gestore interno alla shell con l'utilizzo del comando *trap*.
Solitamente si decide di eseguire un processo in background quando il processo da eseguire è pesante e ha scarsa interazione con l'utente. Per questo durante l'esecuzione di un processo in background è possibile eseguire altri comandi. Un esempio potrebbe essere:
`sort <file_enorme >file_enorme.ord && echo Sort terminato!  &`

5.2 Bash, parte 2

Andiamo adesso ad approfondire l'utilizzo della shell non interattiva. I file letti dalla shell sono detti *script bash*. L'esecuzione di uno script bash avviene invocandolo da terminale.
Quando lanciamo uno script, possiamo anche passare degli argomenti ai quali sarà possibile accedere tramite $1, $2 ...
Ricordiamo che $0 rappresenta il nome dello script.
Inoltre è possibile specificare(ed è buona norma) quale shell deve eseguire lo script. Solitamente si inserisce all'inizio del file di script il comando
`#!/bin/bash`
fatto ciò per eseguire il contenuto dello script è sufficiente eseguire
`bash:~$ nomescript arg1 arg2 ...`
i comandi contenuti in uno script possono essere comandi *built-in* come ls, cd oppure *file eseguibili*. Nel secondo caso viene creato un nuovo processo shell che esegue il programma. La shell padre aspetta che il programma abbia terminato per poter proseguire.

Possiamo inoltre definire degli *alias*, che servono per eseguire in un colpo solo un'insieme complesso di istruzioni o per rendere più sicure le chiamate di comandi "pericolosi", ad esempio:
`bash:~$ alias rm="rm -i"`
che va a modificare il comando rm di modo che chieda sempre conferma di una rimozione.
Possiamo anche creare alias di alias e così via, ma quando si incontra un alias già espanso ci fermiamo per evitare cicli. Per generalizzare gli alias si utilizzano le funzioni(vedi poi).

Nella shell possiamo anche definire le variabili che si comportano come le variabili del C. Infatti se la variabile non esiste viene creata, altrimenti viene sovrascritta. Le variabili create sono sempre globali ammeno che non vengano dichiarate *local*
Esistono anche dei metodi predefiniti per accedere agli argomenti dello script. Ad esempio:

- `$*`: insieme di tutti i parametri. Se inserito tra doppi apici restituisce tutti gli argomenti passati come unico nome
- `$@`: insieme di tutti i parametri. Se inserito tra doppi apici restituisce

tutti gli argomenti come una lista di elementi. Dunque `"$@"` equivale a `"$1" "$2" "$3"`...

- `$$`: il PID della shell

Come in ogni altro linguaggio, possiamo poi definire le operazioni per il controllo del flusso(if, while, case..)
Possiamo inoltre introdurre funzioni che a differenza degli script vengono eseguiti nella shell corrente.

Inoltre abbiamo degli operatori di controllo che ci permettono di fare cose particolari.

Operatore	Effetto
&&	Posto tra due comandi, esegue il secondo solo se il primo ha avuto successo
||	Posto tra due comandi, esegue il secondo solo se il primo non ha avuto successo
{<list>;}	Esegue la lista di comandi nella shell corrente raggruppandoli in un unico blocco
(<list>)	Esegue la lista di comandi in una sottoshell, quindi le operazioni eseguite non lasciano traccia dopo l'esecuzione

Per concludere la panoramica sulla shell, vediamo velocemente le fasi di espansione cui va incontro uno script quando viene eseguito:

1. Espansione degli alias e dell'history: se un comando è presente nella lista degli alias viene espanso. Se un comando inizia con ! (ad esempio !56) viene cercato il comando corrispondente nell'history

2. Espansione delle parentesi graffe: generazione di stringhe arbitrarie. Ad esempio eseguendo `bash:~$ mkdir m{i,ia}o` vengono generate le cartelle mio e miao.

3. Espansione della tilde: al posto del simbolo tilde viene inserito il nome della home directory

4. Espansione delle variabili: sostituzione del nome della variabile con il suo valore

5. Sostituzione dei comandi: espande un comando con il suo output
6. Espansione delle espressioni aritmetiche: vengono risolte le espressioni aritmetiche
7. Suddivisione in parole:
8. Espansione di percorso o globbing: se una parola contiene uno dei simboli speciali come "*" oppure "?" viene opportunamente espanso

5.3 Comandi shell introdotti

Dei primi comandi non abbiamo inserito le opzioni possibili. Per maggiori dettagli consultare un manuale in linea.

- **ls**: mostra tutti i file e le directory nel percorso corrente
- **more, less, cat**: mostrano il contenuto dei file
- **cp, mv**: copia o spostamento di un file
- **mkdir**: crea una directory
- **rm, rmdir**: rimuovere un file o una cartella
- **head, tail**: selezionare le linee iniziali o finali di un file
- **file**: restituisce informazioni sul tipo del file
- **wc**: restituisce rispettivamente il numero di linee, parole e caratteri in un file
- **history**: visualizza gli ultimi comandi eseguiti
- **chmod <num> file** cambia i permessi di "file". Notiamo che "num" è un valore ottale, quindi ad esempio per cambiare il permesso in rw-rw-r– (ovvero 110110100) num deve essere 664
- **locate <pattern>**: cerca i file utilizzando un database periodicamente aggiornato, stampando su stdout tutti i percorsi assoluti contenenti l'elemento cercato.

- **grep** <**opt**> <**pattern**> [**file(s)** ...]: cerca nei file specificati le linee che contengono il pattern specificato e le stampa sullo standard output. Ad esempio:
  ```
  bash:∼ $grep MAX *.c *.h
  mymacro.h:  #define MAX 200
  rand.h:  #define MAX_MIN 4
  ```
  tra le opzioni che ammette abbiamo `-i` che esegue la ricerca con case insensitive, `-v` che stampa tutte le linee che non contengono pattern.

- **alias** <**n**>=<**c**>: crea un alias della lista di comandi c chiamato n.

- **unalias** <**n**>: distrugge l'alias avente nome n

- **vname**=[<**value**>]: viene creata la variabile vname avente valore value.

- **unset vname**: elimina l'associazione tra la variabile vname e il suo valore

- **if** <**condition**>**; then**
  <**command-list**>
  [**else**
  <**command-list**>]
  fi
  comando condizionale. visto che restituisce zero se è andato tutto bene, allora solitamente i comandi da controllare vengono eseguiti nella guardia del comando if e se sono andati male ce ne accorgiamo

- **test** <**condition**>: generalizza l'utilizzo del comando if andando ad analizzare proprietà dei file e altre cose

- **for** <**var**> [**in** <**list**>]**; do**
  <**command-list**>
  done
  costrutto for

- **case** <**expr**> **in**
  (<**pattern**>)
  <**command-list**> **;;**
  (<**pattern**>)

<**command-list**> ;;
...
esac
costrutto case

- **select** <**var**> [**in** <**list**>]; **do**
<**command-list**>
done costrutto select

- **while** <**condition**>; **do**
<**command-list**>
done
costrutto while

- <**nome**> () {
<**lista di comandi**>}
dichiarazione di funzione

- **unset -f name** : elimina la funzione name

- <**command1**> && <**command2**>: esegue command1 e se il suo exit value è zero(true) esegue command2

- <**command1**> || <**command2**>: esegue command1 e se il suo exit value è diverso da zero(false) esegue command2

6 Ancora su bash

6.1 Comandi shell introdotti

In questa lezione non abbiamo introdotto nuovi concetti teorici ma sono state soltanto illustrate alcune funzionalità della shell per operare sulle stringhe. Le elenchiamo di seguito:

- **$**<**var**>: restituisce il valore della variabile var

- **${**<**var**>**:-**<**val**>**}**: restituisce il valore di var se esiste ed è non vuoto, altrimenti restituisce val

- **${**<**var**>**:=**<**val**>**}**: se var esiste non vuoto lo restituisce. Altrimenti gli assegna val e lo restituisce.

- **${<var>:?<message>}**: se var esiste non vuoto lo restituisce. Altrimenti stampa message su stderr
- **${<var>:+<val>}**: se var esiste non vuoto, restitusice val
- **${<var>:<offset>}**: restituisce var a partire dalla posizione offset
- **${<var>:<offset>:<length>}**: restituisce length caratteri della stringa var a partire dalla posizione offset
- **${#<var>}**: restituisce la lunghezza di var
- **${<var>//<pattern>/<string>}**: tutte le occorrenze di pattern in var vengono sostituite con string. Nel caso vi sia solo uno slash fra var e pattern, allora viene sostituita solo la prima

Ricordiamo infine che è possibile trattare le stringhe come interi lavorandoci tramite i seguenti operatori

operatore	effetto
lt	minore
le	minore uguale
eq	uguale
ge	maggiore uguale
gt	maggiore
ne	diverso

7 Chiamate di sistema

7.1 Principali chiamate e gestione degli errori

Prima di introdurre le chiamate di sistema, ricordiamo che sotto Unix ogni file viene rappresentato tramite un i-node. Gli i-node contengono molte informazioni, ovvero tutte quelle visualizzabili eseguendo da shell il comando `ls -l`. Informazioni usuali sono: il tipo di file(-, d, p...), i bit dei permessi(rw-rw-r–), identificativo dell'utente e del gruppo, la dimensione, il tempo di creazione il contatore di hard link...
Inoltre se il file è regolare o directory contiene gli indirizzi dei primi 10 blocchi su disco contenenti i dati del file e l'indirizzo di uno o più blocchi indiretti.

Se invece è un file associato ad un dispositivo(file di questo tipo sono segnalati con la lettera c, e sono contenuti nella cartella dev), contiene il major number e minor number ovvero gli identificatori del driver e del dispositivo.

Ricordiamo che le chiamate di sistema sono il meccanismo tramite cui vengono effettuate delle richieste dal livello utente al livello kernel. Questo meccanismo viene solitamente implementato tramite delle apposite funzioni. Notiamo che le chiamate di sistema possono fallire. Se falliscono restituiscono un valore diverso da zero che può essere -1, NULL, e altri ancora. Ci sono tante ragioni per cui una system call può fallire, e solitamente quando fallisce va a settare la variabile errno. La soluzione consiste nel controllare sistematicamente il valore restituito dalla chiamata e in caso di fallimento controllare errno e utilizzare la funzione perror.
In particolare per le chiamate di sistema di I/O diventa importante esplicitare alcune caratteristiche. Infatti le system call su I/O possono essere:

- Sincronizzate(dove): quelle chiamate di sistema che terminano solo quando i dati vengono effettivamente scritti su disco. Le chiamate di sistema non sincronizzate terminano quando lasciano i dati nei buffer
- Sincrone(quando):quelle chiamate di sistema che terminano quando il lavoro da fare è finito. Le chiamate di sistema non sincrone terminano appena avviate lasciando controllare in seguito il loro effettivo completamento

In generale le chiamate di sistema su I/O in UNIX sono **non** sincronizzate e sincrone. Ad esempio la write scrive su dei buffer e non direttamente sul file da scrivere(dunque è non sincronizzata) ma per procedere dobbiamo aspettare che completi tale scrittura(dunque è sincrona).
Notiamo che le chiamate di sistema introdotte(open, read, write..) fanno parte dello standard POSIX e sono effettivamente system call. Ad esse corrispondono una serie di funzioni facenti parte dello standard ANSI(fopen, fread, fwrite..)
Quali utilizzare? solitamente preferiamo usare quelle dello standard ANSI perché quelle dello standard POSIX non bufferizzano nello spazio utente, e quindi ogni chiamata richiede una commutazione di contesto utente-kernel e un'altra commutazione kernel-utente. Dunque quelle dello standard ANSI sono solitamente più rapide. In ogni caso, l'unica prassi assolutamente da evitare è di usare sia system call che chiamate di libreria. Bisogna scegliere,

o le une, o le altre.

7.2 System call su processi

Abbiamo introdotto le system call relative alla creazione e alla gestione dei processi.

7.3 Funzioni presentate

- **int open(char* path,int flags,mode_p perm)**: apre il file del path. In quale modalità lo apre dipende dai flags che possono essere: lettura(O_RDONLY), scrittura(O_WRONLY), lettura e scrittura (O_RDWR). La modalità di apertura può anche essere messa in OR(tramite il simbolo |) con le seguenti opzioni: O_APPEND, scrive in coda al file. O_CREAT, crea il file se non esiste. O_TRUNC, se il file esiste lo sovrascrive[3]. O_EXCL, se il file esiste da errore. Poiché il comportamento della open con flag O_EXCL è "crea il file se non esiste, altrimenti fallisci", un'interessante applicazione è quella di usare i file come delle lock. Infatti se diversi thread cercano di creare uno stesso file(è sufficiente accordarsi sul nome), solo uno di essi riceve un responso positivo invocando la open con O_EXCL, consentendogli di avere accesso esclusivo ad una risorsa condivisa.
  I permessi devono essere specifiati solo se stiamo creando il file. In particolare vengono calcolati mettendo in AND bit a bit il valore di perm con il valore negato della *file mode creation mask*, detta anche *umask*. Ricordiamo che umask viene ereditata dal padre. Ad esempio, supponiamo che umask valga 0022 e perm 0666:
  `110110110` perm
  `111101101` ¬umask
  `110100100` perm ∧ ¬umask
  `rw-r--r--`
  Se ha successo restituisce il file descriptor del file aperto inserendolo nella tabella del file descriptor. Altrimenti restituisce -1

[3]In realtà la combinazione O_WRONLY | O_CREAT | O_TRUNC è talmente comune che esiste una system call dedicata:
int creat(char* path, mode_t perms)
che crea o sovrascrive il file del percorso path

- **int close(int fd)**: libera il file descriptor fd(che può essere riutilizzato), la memoria impiegata nelle tabelle ed eventualmente l'i-node. A differenza della fclose, non effettua il flush dei buffer nel kernel. Restituisce 0 in caso di successo, -1 altrimenti

- **int read(int fd, void* buf, size_t n)**: legge al più n bytes dal file legato al file descriptor fd salvandoli nel buffer buf.
  Restituisce il numero di byte letti in caso di successo(se restituisce 0 vuol dire che siamo arrivati in fondo al file), altrimenti restituisce -1.

- **int write(int fd, void* buf, size_t n)**: scrive al più n bytes sul file legato al file descriptor fd prelevandoli dal buffer buf.
  Restituisce il numero di byte scritti, o -1 in caso di errore.

- **int unlink (char* path)**: elimina un link al file nel percorso path. Se il contatore di riferimenti al file va a zero, viene eliminato il file. Ma nel caso in cui alcuni processi stiano ancora lavorando sul file, la unlink aspetta che tutti abbiano invocato la close.
  Restituisce 0 in caso di successo o -1 in caso di fallimento.

- **off_t lseek(int fd, off_t pos, size_t w)**: sposta il punto di lettura del file avente file descriptor fd. Tramite w decido l'offset di partenza che può essere:
  SEEK_SET: in posizione pos
  SEEK_CUR: a partire dall'offset corrente a cui devo aggiungere il valore pos
  SEEK_END: a partire dalla fine del file a cui devo aggiungere il valore pos
  Notiamo che il valore di pos può anche essere negativo.
  Inoltre la posizione dell'offset finale può anche essere dopo la fine del file. In tal caso la successiva write estende il file della lunghezza necessaria, riempiendo l'intervallo con byte pari a 0.
  Restituisce il nuovo offset in caso di successo o -1 in caso contrario

- **int stat(char* path, struct stat* buf)**: salva diverse informazioni riguardo al file contenuto in path in un buffer strutturato come una struct avente campi che indicano i-node, diritti, ultima modifica, ultimo accesso..
  Restituisce 0 in caso di successo, -1 altrimenti.

- **DIR* opendir(char* path)**: funziona come una fopen, solo che viene fatta su cartelle. Infatti il puntatore restituito viene utilizzato dalle altre funzioni.
  Restituisce il puntatore in caso di successo, NULL se fallisce

- **int closedir(DIR* dirp)**: funziona come una fclose, solo che viene fatta su cartelle.
  Restituisce 0 in caso di successo, -1 in caso di fallimento

- **struct dirent* readdir(DIR* dirp)**[4] Restituisce un puntatore a dirent che rappresenta la successiva directory nella lista delle directory puntata da dirp. Fondamentalmente se la readdir viene chiamata dentro ad una cartella A che contiene altre cartelle B,C.. allora restituisce il puntatore alla prima cartella contenuta in A. Altrimenti restituisce NULL.
  Notiamo però che la readdir può restituire NULL in due casi diversi. Il primo è se ha fallito, il secondo è se non ci sono più cartelle da visitare. Possiamo distinguere fra i due casi perché il primo setta errno, il secondo ovviamente no.

- **char* getcwd(char* buf, size_t bufsize)**: salva in buf il percorso corrente. In caso di successo restituisce il puntatore al buffer. Se il buffer non è abbastanza grande, restituisce NULL con errore ERANGE

Introduciamo ora le system call che operano su processi:

- **pid_t getpid(void)**: restituisce il pid del processo

- **pid_t getppid(void)**: restituisce il pid del processo genitore

- **pid_t fork(void)**: crea un nuovo processo. Lo spazio di indirizzamento del figlio come la tabella dei descrittori di file sono copie distinte mentre la tabella dei file aperti(e dunque tutti i puntatori ai file) sono condivisi.[5]

[4]Sotto alcuni sistemi operativi come Solaris è possibile leggere una cartella con la SC read. Non essendo però standard POSIX, tralasciamo questo punto

[5]Ricordiamo che ogni processo ha la sua tabella dei file descriptor. Quando viene invocata la fork il figlio possiede nel suo spazio di indirizzamento una copia identica ma distinta della tabella dei descrittori. Resta invece comune la tabella dei file aperti a cui entrambi i processi puntano(non è contenuta nel processo ma a livello kernel)

Notiamo che creare una copia è costoso e che spesso il processo figlio modifica molte cose avviando una exec. Per questi motivi la fork è implementata tramite meccanismo *copy-on-write*. Restituisce -1 in caso di errore[6]

- **int execl(char* path, char* arg0, ..., (char *) NULL)**: serve per modificare i dati di un processo(spazio di indirizzamento..) avviando un altro eseguibile.
  Solitamente viene utilizzata in coppia con la fork, ma in certi(rari) casi viene utilizzata anche da sola. Ad esempio nel caso in cui un programma passi attraverso diverse "fasi". Queste fasi devono essere tendenzialmente indipendenti tra loro visto che la exec va a riscrivere lo spazio di indirizzamento. Altre volte la exec viene utilizzata da sola per invocare un comando dopo che sono stati eseguiti alcuni lavori preliminari.
  La funzione execl è in realtà parte di una famiglia di sei funzioni. In generale, ogni funzione appartenente a tale famiglia è nella forma execAB, dove:

  – A può essere *l* oppure *v*. Abbiamo "l" nel caso in cui gli argomenti da passare siano passati tramite una lista. Abbiamo "v" nel caso in cui gli argomenti da passare siano contenuti in un array(vettore). Scegliere di passare gli argomenti tramite vettore è d'obbligo se a tempo di compilazione non sappiamo quanti argomenti dovremo passare al nuovo programma

  – B può essere del tutto assente, oppure *p* oppure *e*. Abbiamo "p" se vogliamo eseguire il nuovo programma passandogli le variabili d'ambiente del processo padre. Abbiamo "e" se vogliamo eseguire il nuovo programma passandogli delle variabili d'ambiente da noi specificate tramite il campo env del main:
    int main(int argc, char* argv[ ], char* env[ ])

  Nel complesso abbiamo 6 diverse funzioni appartenenti alla famiglia exec: execl, execlp, execle, execv, execvp e execvpe. Se ha successo non restituisce in quanto il suo indirizzo di ritorno non è quello dove era

[6] Anche se vale la pena notare che la fork, non avendo argomenti, può restituire -1 solo in caso di esaurimento di risorse

stata invocata la funzione, bensì l'inizio del codice del nuovo processo. Se fallisce restituisce -1

- **void exit(int status)**: termina il processo, chiude i file descriptor, libera lo spazio di indirizzamento, invia un SIGCHLD al padre e restituisce status a quest'ultimo. Inoltre chiama la funzione associata ad atexit(se c'è) ed esegue il flush dei buffer di I/O

- **int atexit(void* function(void))**: registra la funzione function di modo che quando il programma termina con una exit o una return venga eseguita function. La atexit viene solitamente utilizzata per lanciare del codice di pulizia a fine programma. Può lavorare soltanto su variabili globali e se viene registrata più di una funzione, le funzioni registrate verranno chiamate in ordine inverso

- **int waitpid(pid_t pid, int* statusp, int options)**: attende la terminazione di qualcosa. Nel caso in cui:

  - pid > 0: aspetta il figlio avente pid corrispondente
  - pid = -1: aspetta un figlio qualsiasi
  - pid = 0: aspetta un figlio qualsiasi nello stesso process group
  - pid < -1: aspetta un figlio qualsiasi del gruppo -pid

  Status prende il codice di ritorno del processo figlio e altre informazioni. Options è un insieme di flags in OR che forniscono informazioni aggiuntive.
  Se ha successo restituisce 0 o pid, altrimenti restituisce -1

8 Pipe

8.1 System call per interprocess communication(IPC)

Le *pipe* sono file speciali che vengono utilizzati per la comunicazione tra processi. Ad esempio usiamo delle pipe ogni volta che nella shell concateniamo più operazioni separate dal simbolo di pipeline.
Possiamo anche avere delle pipe bidirezionali e che connettono più di due processi. Tali funzioni avanzate non sono implementate di default nella shell ma devono essere programmate tramite le opportune system call.
Le pipe sono di due tipi:

- Pipe senza nome(unnamed): utilizzano un file senza nome visibile e servono per la comunicazione tra processi appartenenti ad una stessa gerarchia familiare(figli, nipoti..) originata dalla chiamata di fork
- Pipe con nome (named): utilizzano un file speciale visibile a tutti i processi sulla macchina

Le pipe senza nome vengono create tramite la funzione *pipe* e vengono rappresentate da due descrittori di file contenuti in un array, uno per la lettura(posizione 0 dell'array), uno per la scrittura(posizione 1 dell'array).
Una volta creata è possibile vederne la capacità minima andando a leggere la variabile Posix `_POSIX_PIPE_BUF`. Questo valore rappresenta però la dimensione **minima** della pipe, mentre per la dimensione effettiva devo utilizzare la funzione *fpathconf*.
Per scrivere e leggere sulle pipe usiamo le chiamate già viste, ovvero la *write* e la *read*. Il primo argomento di queste due funzioni è un file descriptor. Ovviamente per leggere e scrivere dovremo usare i file descriptor contenuti nell'array di due posizioni che avevamo già utilizzato nella funzione pipe.
Per gestire la chiusura di una pipe è necessario chiamare la *close* sui descrittori contenuti nell'array che stato utilizzato durante la comunicazione.
Facciamo un esempio: supponiamo ora di avere un processo padre che voglia fare da server e un processo figlio che voglia fare da client. I passi che solitamente vengono fatti per utilizzare una pipe senza nome tra padre e figlio sono i seguenti:

1. Il padre crea la pipe
2. Il padre si duplica con fork
3. Il padre(server) chiude pfd[1](scrittura) mentre il figlio(client) chiude pfd[0](lettura)
4. I processi comunicano con read/write
5. La comunicazione è finita. Il padre chiude anche pfd[0] mentre il figlio chiude anche pfd[1]

Notiamo che i passi sopraelencati sono obbligati e che il funzionamento della pipe è dovuto alla natura della SC fork. Infatti, come detto nel capitolo precedente, quando viene invocata la fork il nuovo processo ottiene una

copia identica e distinta della tabella dei file descriptor del padre. In questo modo il nuovo processo conosce l'effettivo file descriptor su(da) cui può scrivere(leggere).[7] In tal senso diventa chiaro che le pipe possono essere create solo tra processi che hanno un grado di parentela.
Per scrivere e leggere messaggi di lunghezza variabile sulla pipe è necessario implementare un protocollo. Il più semplice è quello di scrivere prima la lunghezza del messaggio che verrà scritto subito dopo.
Possiamo inoltre duplicare i file descriptor tramite la chiamata delle funzioni *dup* e *dup2*. Queste due funzioni vengono solitamente utilizzate per effettuare operazioni di redirezione. Ad esempio, posso utilizzare la dup2 per redirigere lo stdout su un file e fare in modo che ogni stampa vada su tale file.
Infine, per fare in modo che un processo A possa inviare e ricevere dati ad un processo B, il modo migliore è utilizzare due pipe unidirezionali. Sulla prima pipe A scrive e B legge, viceversa sull'altra.[8]

Le pipe con nome vengono invece create tramite la funzione *mkfifo*. Poiché invocando la mkfifo viene creato un nuovo file di tipo "p", nelle pipe con nome viene meno quanto dicevamo riguardo al funzionamento dovuto alla SC fork. Un'altra differenza è che prima di iniziare la lettura o la scrittura la pipe con nome deve essere aperta da due processi. L'apertura viene fatta tramite la open, e se un processo scrittore(lettore) invoca la open aspetta che un altro processo lettore(scrittore) invochi una open.
Se `O_NONBLOCK`, oppure `O_NDELAY` sono settate e non ci sono lettori, la open fallisce.
Un'applicazione tipica si ritrova nel sistema server singolo-client multipli. Il server apre la pipe in lettura e scrittura(se la aprisse solo in lettura alla chiusura dell'ultimo processo leggerebbe un EOF mentre noi vogliamo che il server resti sempre attivo). I client scrivono su una stessa pipe e ricevono risposta su pipe private. Affinché non venga terminato, il server deve ignorare i SIGPIPE che riceve nel momento in cui prova a scrivere su una pipe senza lettori.

[7] Inoltre questa è una delle motivazioni per cui le SC fork ed exec siano separate. Infatti tra l'una e l'altra abbiamo del lavoro extra da fare(ad esempio ogni processo deve chiudere uno dei file descriptor). Potremmo anche chiederci come mai non proporre una versione di fork+exec con argomenti speciali che denotino se la circostanza richieda di effettuare tali lavori extra. La risposta è che i creatori del C volevano limitare il numero di SC

[8] In realtà esistono anche delle pipe bidirezionali ma non sono standard, quindi non sono portabili, non ce ne occupiamo

8.2 Funzioni presentate

- **int pipe(int pfd[2])**: crea la pipe. Da questo momento è possibile leggere sulla pipe utilizzando pfd[0] o scriverci utilizzando pfd[1]. Se ha successo restituisce 0, altrimenti -1

- **long fpathconf(int fd, int name)**: restituisce informazioni sul file descriptor fd. Può essere usata per ottenere la capacità della pipe settando come opzione name la macro `_PC_PIPE_BUF`

- **int write(int fd, void* buf, size_t n)**: cerca di scrivere n byte. Se n è minore della dimensione della pipe, allora la scrittura è atomica. Se non c'è abbastanza spazio la write aspetta che dei lettori l'abbiano svuotata. Se invece il messaggio da scrivere è più grande della dimensione effettiva del buffer della pipe, allora possono verificarsi scritture parziali(non atomiche).
  Nel caso `O_NONBLOCK` sia settato[9] la write cerca di scrivere tutto in una volta. Se non vi riesce, restituisce -1.
  Infine, se un processo cerca di scrivere su una pipe senza lettori, riceve un SIGPIPE. Questa è una situazione a cui stare attenti perché SIGPIPE, di default, genera la terminazione del processo. Se SIGPIPE viene ignorato, la scrittura su una pipe senza lettori restituisce -1 con errore EPIPE.

- **int read(int fd, void* buf, size_t n)**: legge dalla pipe fino a n byte. Se `O_NONBLOCK` non è settato, la pipe è vuota e i descrittori di scrittura non sono stati chiusi, allora pone il processo in attesa che qualcuno vi scriva. Invece se i descrittori di scrittura sono stati chiusi restituisce 0.Se invece `O_NONBLOCK` è settato, se la pipe è vuota restituisce -1

- **int close(int fd)**: libera il descrittore fd. Se era l'ultimo associato alla pipe, da adesso in poi chiunque cerchi di fare una read, riceverà 0 come risultato. Se un processo prova a fare una write su una pipe avente tutti i descrittori chiusi, riceve un SIGPIPE

- **int dup(int fd)**: duplica il file descriptor fd e restituisce la posizione della tabella dei descrittori in cui ha fatto la copia. L'intero restituito rappresenta la prima posizione libera di tale tabella. Restituisce -1 altrimenti

[9] `O_NONBLOCK può essere settato tramite la funzione fcntl`

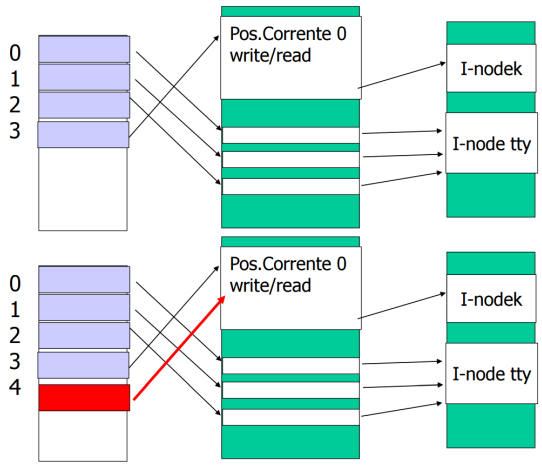

Figure 2: in basso, come viene modificata la tabella dei descrittori dopo la chiamata di dup(3)

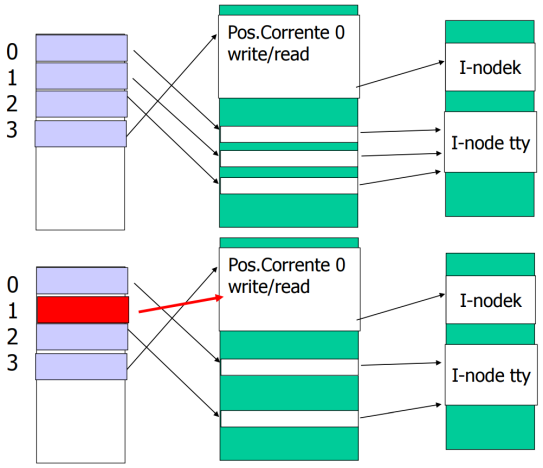

Figure 3: in basso, come viene modificata la tabella dei descrittori dopo la chiamata di dup2(3,1)

- **int dup2(int fd, int fd2)**: duplica il file descriptor fd. La copia viene inserita in posizione fd2 nella tabella dei descrittori. Se fd2 era in uso, viene chiuso prima della copia. La dup2 è atomica, se qualcosa va male quindi, fd2 non viene chiuso. Inoltre se fd=fd2, la funzione non fa niente. Restituisce fd2 in caso di successo, -1 altrimenti

- **int mkfifo(const char* path, mode_t perms)**: apre una pipe con nome path, avente permessi perms. I permessi sono scritti come nella open(utilizzo di umask). Se ha successo restituisce 0, altrimenti -1

9 Thread

9.1 System call su thread

Fino ad ora i nostri processi avevano un unico flusso di controllo, o *thread*. Da adesso è possibile far lavorare e condividere dati a più flussi di controllo, ognuno con il proprio program counter e stack.
Una domanda che potrebbe sorgere spontanea è quando usare i thread e quando usare i processi. Tendenzialmente i thread vengono usati quando dobbiamo lavorare in modo concorrente su una stessa struttura dati. Infatti potrei voler mantenere attiva l'interfaccia utente mentre in background effettuiamo dei calcoli. Oppure potrei voler semplicemente sfruttare le potenzialità di una CPU multicore, visto che lo scheduler UNIX assegna ad ogni core un thread. In generale i thread vengono utilizzati quando vogliamo un grado di concorrenza più "fine", visto che inoltre, i processi sono più pesanti da creare, schedulare...
I nuovi thread vengono creati tramite la *pthread_create*. Possiamo attendere che un thread finisca di fare ciò che deve fare(ovvero termini la funzione da cui si avvia), tramite la *pthread_join*. Chi chiama la join si ferma aspettando che un certo thread termini.
Un thread può terminare in due modi. Normalmente tramite una return o una *pthread_exit* o se viene cancellato.

Il principale problema che si incontra nell'utilizzo dei thread è nella *concorrenza*. Capita infatti molto spesso che più thread debbano accedere ad una risorsa condivisa(una variabile globale, un buffer..) e se lo fanno in modo disordinato possono generare danni disastrosi. Per evitare situazioni di questo tipo si utilizzano le funzioni che garantiscono la *mutua esclusione*, funzioni che consentono l'accesso esclusivo ad una risorsa.
La mutua esclusione viene garantita tramite i *mutex*, dei semafori binari che permettono di bloccare(lock) le risorse.
I mutex vengono creati in due modi: se il mutex è globale viene creato tramite la macro `PTHREAD_MUTEX_INITIALIZER`. Se è non globale, invocando

la *pthread_mutex_init*.
I mutex vengono gestiti tramite due funzioni principali, la *pthread_mutex_lock* e la *pthread_mutex_unlock* che rispettivamente acquisiscono e rilasciano il mutex.
Un'efficiente uso delle lock consiste nell'acquisire una lock subito prima di dover accedere ad una risorsa condivisa e nel rilasciare la lock subito dopo aver lavorato sulla risorsa.
Può anche succedere che si formi una situazione di *deadlock*, ovvero quando i thread si bloccano perché non possono prelevare una lock.
Strettamente legate alle lock, abbiamo le *condition variables*, che ci permettono di avvisare tempestivamente un altro thread che un evento è occorso. Infatti noi associamo (concettualmente) ad ogni variabile di condizione un evento. Quando quell'evento si verifica chiamiamo una funzione apposita che segnali ai thread che quell'evento si è verificato.
Come per le lock, le condition variables sono create in due modi.
Tramite la macro `PTHREAD_COND_INITIALIZER` se sono globali, oppure con la *pthread_cond_init*.
Un thread può anche essere cancellato invocando la funzione *pthread_cancel*. La cancel fa terminare il thread appena raggiunge un punto di cancellazione. Di norma, tale punto coincide con la chiamata di una funzione safe stabilita.
Per gestire meglio la terminazione di un thread si utilizzano dei *cleanup handlers*, delle funzioni che vengono invocate quando il thread termina. Devono il loro nome al loro principale utilizzo: "pulire" quando il thread ha finito. Queste funzioni vengono chiamate nell'ordine inverso col quale sono state registrate.

9.2 Funzioni presentate

- **int pthread_create(pthread_t* id, pthread_attr_t* attr, void* *start_fcn(void *), void* arg)**: crea un nuovo thread e ne salva il thread ID in id. Il thread, una volta creato esegue la funzione start_fcn, passandogli l'argomento arg. Poiché useremo gli attributi di default, attr sarà NULL. Restituisce 0 in caso di successo, o il numero dell'errore in caso di fallimento.

- **int pthread_join(pthread_t tid, void** status_ptr)**: attende il thread tid(dove tid è il thread id del thread). Se status_ptr è diverso da NULL, viene salvato lo stato di terminazione del thread. Notiamo

inoltre che quando un thread termina, la sua memoria non viene liberata se qualcuno non chiama una join su tale thread. Dunque non chiamare la join genera memory leak![10] Restituisce 0 in caso di successo, il numero di errore altrimenti.

- **void pthread_exit(void* retval)**: il thread che la chiama termina salvando il suo stato di ritorno retval di modo che tale valore possa essere letto da una join

- **int pthread_mutex_lock(pthread_mutex_t* mutex)**: acquisisce la lock mutex. Se mutex è già stato acquisito da un altro thread, chi ha chiamato la lock si pone in uno stato di attesa attiva fino a quando non ottiene mutex. Restituisce 0 in caso di successo o il numero di errore altrimenti

- **int pthread_mutex_unlock(pthread_mutex_t* mutex)**: rilascia la lock mutex. Restituisce 0 in caso di successo o il numero di errore altrimenti

- **int pthread_cond_signal(pthread_cond_t* cond)**: segnala che l'evento associato a cond si è verificato. Se nessuno aspettava tale evento, la signal va persa. Restituisce 0 in caso di successo, il numero dell'errore altrimenti

- **int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mtx)**: blocca il processo chiamante in attesa che qualcuno chiami una signal o una broadcast su cond. Nel farlo, rilascia la lock mtx. Quando il thread viene svegliato acquisice nuovamente mtx. Restituisce 0 in caso di successo, il numero dell'errore altrimenti

- **void pthread_cleanup_push(void (*handler) (void*), void* arg)**: registra la funzione handler(ponendola in cima alla pila delle funzioni registrate) passandogli gli argomenti arg.

- **void pthread_cleanup_pop(int execute)**: rimuove la funzione in cima alla pila delle funzioni registrate. La esegue se execute è un valore diverso da zero

[10] Oppure possiamo invocare la pthread_detach avente segnatura:
int pthread_detach(pthread_t tid)
In tal caso le risorse utilizzate dal thread vengono liberate e su quel thread non potrò invocare una join

10 Socket

10.1 System call per interprocess communication

Consideriamo adesso un'altro sistema per avere connessione tra più processi, le socket.[11]
Una socket è un file speciale per connettere due o più processi tramite un canale di comunicazione. La prima differenza con le pipe, è che i processi possono anche risiedere in macchine differenti. Un'altra differenza è che il processo di creazione della connessione è asimmetrico per client e server. Vediamone i passi fondamentali:

- Per il server abbiamo le seguenti fasi
    1. creazione del file della socket tramite la SC *socket*
    2. associare il file ad un indirizzo tramite la SC *bind*
    3. specificare che sulla socket creata, il server è disposto ad accettare le connessioni, operazione effettuata tramite la *listen*
    4. porsi in attesa di richieste di connessioni invocando la *accept*
- Per il client invece
    1. creazione del file della socket tramite la SC *socket*
    2. collegarsi al server tramite la SC *connect*

una volta che c'è una corrispondenza tra la accept del server e la connect di un client, viene creata effettivamente la socket e il server può porsi nuovamente in attesa di altre connect.

Uno dei principali problemi da risolvere utilizzando le socket risiede nel fatto che noi vorremmo un server capace di gestire connessioni con diversi client.Per risolvere tale problema ci sono due soluzioni tipiche.
La prima consiste nell'utilizzare un server multithread in cui un dispatcher si pone in attesa di connessioni sulla accept mentre ogni thread worker gestisce il trasferimento di informazioni con un altro processo(ponendosi in attesa sulla read).

[11] In realtà esistono diversi meccanismi di IPC che non vedremo. Li citiamo per completezza: le message queues, i semafori e la memoria condivisa

La seconda soluzione consiste nell'avere un server sequenziale che utilizza la funzione *select*. Infatti tramite la select è possibile stabilire quale file descriptor sia pronto ad operare(sia che operi in lettura o in scrittura).

Tramite le socket possiamo anche utilizzare gli indirizzi `AF_INET` che ci consentono di fare trasferimenti tra macchine diverse. Il primo problema che si pone in questo caso è la rappresentazione dei dati. Esistono infatti due modi di memorizzare un numero:

- Little endian: parte dal byte meno significativo, utilizzato nei processori intel.
- Big endian: parte dal byte più significativo, utilizzato nei protocolli internet, per questo viene anche detto *Network Byte Order*

Il tipo di memorizzazione della macchina specifica viene detto *Host Byte Order*. Allora è necessario che ogni macchina prima dell'invio converta la propria memorizzazione da *Host* a *Network* e che il ricevente faccia l'inverso.
Gli indirizzi `AF_INET` sono valori formati da due parti.
La prima, di 32 bit identifica la macchina(IPv4[12]), e viene scritta in *dotted notation*(192.168.1.209)
La seconda, composta di 16 bit è un identificativo della porta. La porta serve per identificare il servizio che la macchina sta usando.[13]
Conoscendo l'indirizzo del server a cui chiedere è possibile reperire informazioni sui siti e altro. Se si conosce il nome del server è anche possibile ricavarne l'indirizzo tramite un database distribuito invocando la funzione getaddrinfo.

Infine, scriviamo per completezza delle *connectionless sockets* nell'ambito della comunicazione internet. Questo protocollo per lo scambio di dati non usa una connessione dunque la listen e la accept non sono chiamate, chi vuole inviare specifica un indirizzo a cui inviare e chi riceve specifica a quale indirizzo vuole ricevere. Per farlo chiama la bind con cui ottiene un nome a cui potrà essere raggiunto dall'esterno. Solitamente i dati inviati con questo metodo vengono detti *datagrams*, ovvero dei pacchetti di dati indipendenti che possono arrivare in ordine sparso e raramente non arrivare proprio.

[12]Ne esiste anche una versione più recente, chiamata IPv6 che grazie ai suoi 128 bit copre fino a $2^{128} \simeq 3,4 * 10^{38}$ siti internet

[13]Ad esempio la porta 80 è quella dei server http, la porta 4662 è quella usata da eMule

10.2 Funzioni presentate

- **int socket(int domain, int type, int protocol)**: crea il file relativo al socket. Questa funzione viene chiamata sia dal server che dal client. Il dominio domain è `AF_UNIX` per processi su una stessa macchina, ma può anche essere `AF_INET` per connessioni internet. Il dominio deve però coincidere con quello specificato nella bind. Il tipo di comunicazione dipende anche dal tipo di dominio scelto. Per noi che spesso abbiamo usato `AF_UNIX`, un tipo comune è `SOCK_STREAM`. Anche il protocollo dipende dal dominio, quello di default è 0. Il valore restituito è il file descriptor in caso di successo, o -1 in caso di fallimento

- **int bind(int sock_fd, struct sockaddr* sa, socklen_t sa_len)**: assegna un indirizzo a una socket. Restituisce 0 in caso di successo, -1 altrimenti

- **int listen(int fd, int backlog)**: serve per segnalare che la socket con file descriptor fd accetta connessioni. Il parametro backlog indica la lunghezza massima della coda delle connessioni. Tale valore è solitamente settato con la macro `SOMAXCONN` che è il valore massimo supportato. Restituisce 0 in caso di successo, -1 altrimenti

- **int accept(int sock_fd, struct sockaddr* sa, socklen_t sa_len)**: accetta le connesioni provenienti dagli altri processi. Se il secondo parametro è diverso da NULL, al termine della funzione contiene il file descriptor della socket che ha accettato di connettersi. In tal caso, sa_len rappresenta la dimensione di sa. Se sa è uguale a NULL, sa_len non è utilizzato. Se non ci sono richieste, si blocca, altrimenti crea effettivamente la nuova socket restituendone il file descriptor. Se fallisce, restituisce -1

- **int connect(int sock_fd, struct sockaddr* sa, socklen_t sa_len)**: serve per richiedere una connessione, la connect viene infatti chiamata dal client. L'indirizzo sa deve essere lo stesso utilizzato nella bind. Restituisce 0 in caso di successo, -1 altrimenti

- **int select(int nfds, fd_set* rd, fd_set* wr, fd_set* err, struct timeval* timeout)**: la funzione select pone il chiamante in stato di sleep fintanto che almeno uno dei file descriptor degli insiemi specificati (rd, wr e err), non diventa attivo, per un tempo massimo specificato

da timeout.
Notiamo che rd, wr, err non sono dei file descriptor, bensì **insiemi** di file descriptor! Non a caso il loro tipo è *fd_set*. Questi insiemi di file descriptor possono essere gestiti tramite un kit di quattro funzioni:

- FD_ZERO(fd_set* set):inizializza l'insieme set(vuoto).
- FD_SET(int fd, fd_set* set): inserisce il file descriptor fd nell'insieme set.
- FD_CLR(int fd, fd_set* set): rimuove il file descriptor fd dall'insieme set.
- FD_ISSET(int fd, fd_set* set): controlla se il file descriptor fd è nell'insieme.

l'idea è di inserire i file descriptor dedicati alla lettura nel fd_set rd, i file descriptor dedicati alla scrittura nel fd_set wr, e i file descriptor dedicati agli errori nel fd_set err.
Questi inserimenti vengono fatti utilizzando la FD_SET dopo che i vari insiemi sono stati azzerati tramite la FD_ZERO.
Una volta che è stata invocata la select ciascun insieme viene modificato e resta settato solo il bit di quei file descriptor che hanno fatto una richiesta. Per capire quale file descrptor sia attivo dopo la chiamata di select devo controllarli tutti con FD_ISSET. Quindi per motivi di efficienza è meglio sapere quali file descriptor sono in uso e testare solo quelli piuttosto che testare tutti i file descrptor nell'intervallo: [0, MAX$\{fd_i\}$]
Il fatto che l'insieme di file descriptor venga modificato, implica che l'insieme vada reimpostato dopo ogni chiamata di select.
Se la chiamata di FD_ISSET restituisce true su un certo file descriptor dopo che è stata invocata la select, allora posso effettuare le operazioni necessarie su quel file descriptor, siano esse di lettura o scrittura.
In caso di successo restituisce il numero di file descriptor settati, -1 altrimenti

11 Segnali

11.1 I segnali

I segnali sono delle "interruzioni" software che segnalano il verificarsi di un evento. Vi è un segnale per ogni evento particolare(ad ogni segnale corrisponde un numero), e quando il segnale arriva può: essere ignorato, essere gestito dalle funzioni di default, essere gestito come più ci aggrada.
I segnali possono essere inviati in molte situazioni: da un processo/thread ad un altro, dal sistema operativo, dall'utente tramite shell o da tastiera con particolari combinazioni di tasti.

Per gestire i segnali abbiamo tre strutture fondamentali:
pending signal bitmap, contiene un bit per ogni tipo di segnale e indica se c'è un segnale pendente per quel thread. Nel caso vi sia, il bit settato mi dice qual'è.
Signal mask, contiene un bit per ogni tipo di segnale. Se il bit per un certo segnale è settato allora quel segnale verrà ignorato quando viene ricevuto.
Signal handler array, che contiene le informazioni su come gestire il segnale una volta che viene intercettato.
Quando un programma single thread riceve un segnale, viene inserito nella pending signal bitmap, se il bit relativo della signal mask non è settato allora il processo signle thread viene interrotto, altrimenti viene gestito da un eventuale gestore personalizzato.
Se il programma che riceve il segnale è multi-thread, allora ci sono due situazioni possibili: se il segnale è indirizzato ad un thread preciso, si procede come se il thread che lo riceve fosse un programma single-thread. Se invece il segnale era destinato all'intero processo, viene passato ad un thread a caso.
Poiché gli unici casi in cui un segnale è diretto ad un thread specifico è quello in cui sia stato proprio quel thread a generare la situazione anomala(ad esempio il thread ha tentato una divisione per zero), in tutti gli altri casi i segnali devono essere gestiti con attenzione.

Per personalizzare la gestione si utilizza la funzione *sigaction*, che definisce le operazioni da eseguire quando un certo segnale viene ricevuto.
Ricordiamo però che alcuni segnali, come SIGKILL e SIGSTOP, non possono essere gestiti in modo personalizzato. SIGKILL uccide l'intero processo, mentre SIGSTOP sospende l'intero processo.

I segnali SIGCHLD sono gli unici ad essere accumulati. Negli altri casi se arriva un segnale dello stesso tipo di uno già arrivato, viene perso.
Visto che durante l'esecuzione del gestore possono arrivare altri segnali, il gestore deve essere il più breve e semplice possibile. Inoltre solo certe funzioni possono essere chiamate nel gestore in modo sicuro. Ad esempio le chiamate di printf, scanf, non sono *safe*.
Di norma allora, si preferisce gestire SIGINT, SIGTERM, SIGSEGV e ignorare SIGPIPE(quest'ultimo viene ignorato perché anche la chiusura improvvisa di un client genera un SIGPIPE nel server, e noi non vogliamo che il server si fermi solo per questo motivo).
Quando gestiamo il segnale in modo personalizzato, dobbiamo sempre pensare a due cose:

1. cosa fa il nostro gestore per cambiare lo stato del sistema una volta che il segnale è occorso

2. dove andare una volta che il gestore ha terminato. La scelta più comune è di ritornare dove eravamo, terminare il programma o fare un global jump[14]

Ricordiamo comunque che quando viene invocata una fork o una pthread_create, la maschera dei segnali pendenti viene resettata. Rimane intatta se viene invocata una exec.
La signal mask invece viene sempre ereditata senza alterazioni(sia che venga chiamata fork, exec o pthread_create) Il signal handler array invece viene ereditato quando viene chiamata una fork, ma che dopo la chiamata della exec, la gestione ritorna quella di default. La pthread_create invece mantiene intatto il gestore.

Per evitare che alcuni thread gestiscano i segnali, quei thread li mascherano usando la *pthread_sigmask*. Mascherare i segnali può essere utile quando vogliamo che solo certi thread gestiscano i segnali, oppure in attesa che il programma finisca di registrare i segnali da ignorare.

Come dicevamo, uno dei modi per inviare segnali consiste nel far mandare a un processo/thread un segnale ad un altro. Le funzioni che effettuano queste

[14]Tuttavia l'ultima opzione è sconsigliata poiché un programma che fa uso di goto o funzioni di jump è difficile da capire e mantenere

operazioni sono la *kill* e la *pthread_kill*
Possiamo anche porre i processi in attesa di ricevere un segnale, tramite le funzioni *pause* e *sigwait*. Questa strategia è chiaramente da preferire ad un'attesa attiva che potremmo generare tramite un ciclo while.

Un altro aspetto che ci costringe ad attuare una gestione dei segnali corretta è il fatto che certi segnali possono interrompere alcune chiamate di sistema. Infatti se il proesso stava eseguendo una system call, e riceve un segnale, se il segnale causa la terminazione allora non succede nulla di strano ma se l'esecuzione procede, la chiamata di sistema fallisce con codice di errore EINTR (Interrupted System Call). Questo ci spinge ad analizzare con cura le interazioni tra chiamate di sistema e gestione dei segnali.
Il comportamento da tenere in generale è il seguente: se non vogliamo mai essere interrotti, possiamo specificarlo settando il flag SA_RESTART subito dopo la registrazione del gestore. In alternativa possiamo riattivare esplicitamente una chiamata di sistema interrotta. Questo viene fatto controllando se, subito dopo l'esecuzione della chiamata, errno = EINTR. In tal caso basta chiamare nuovamente la system call interrotta.

11.2 Funzioni presentate

- **int sigaction(int num, struct sigaction* a, struct sigaction* o)**: fa gestire ad a, il segnale avente numero num. Se o è specificato, dopo la chiamata contiene le informazioni del precedente signal handler array. Restituisce 0 in caso di successo, -1 altrimenti. Riportiamo anche la composizione del tipo composto *struct sigaction*:
  ```
  struct sigaction {
     void (*sa_handler) (int);
     sigset_t sa_mask;
     int sa_flags;
  }
  ```
  dove sa_handler specifica come gestire il segnale, ovvero con `SIG_IGN` per ignorarlo, oppure `SIG_DFL` per gestirlo di default, oppure se la gestione è personalizzata, il puntatore alla funzione che gestisce il segnale. Con sa_mask determino quali altri segnali devono essere gestiti oltre a quello che stiamo registrando con la sigaction. Infine sa_flags serve per le eventuali opzioni.

- **int pthread_sigmask(int how, const sigset_t set, sigset_t* old)**: modifica la signal mask del processo/thread che la invoca secondo quanto specificato da how. Infatti how può essere:
  `SIG_SETMASK`: la nuova signal mask è set
  `SIG_BLOCK`: la nuova signal mask diventa l'OR di set e della vecchia signal mask
  `SIG_UNBLOCK`:la nuova signal mask rimuove i segnali presenti in set dalla vecchia signal mask
  Nel caso in cui old non sia NULL, dopo la chiamata contiene la maschera prima che venisse modificata. Restituisce 0 in caso di successo, il codice di errore altrimenti.

- **int sigfillset(sigset_t* pset)**: mette a 1 tutti i bit della maschera puntata da pset. Restituisce 0 in caso di successo, -1 altrimenti.

- **int sigemptyset(sigset_t* pset)**: azzera tutta la maschera puntata da pset. Restituisce 0 in caso di successo, -1 altrimenti.

- **int sigaddset(sigset_t* pset, int signum)**: mette a 1 la posizione del segnale signum in pset. Restituisce 0 in caso di successo, -1 altrimenti.

- **int sigdelset(sigset_t* pset, int signum)**: mette a 0 la posizione relativa a signum in pset. Restituisce 0 in caso di successo, -1 altrimenti.

- **int sigismember(const sigset_t* pset, int signum)**: restituisce 1 se signum è membro della maschera pset, 0 se non lo è e -1 in caso di errore.

- **int kill(pid_t pid, int signum)**: genera il segnale signum inviandolo a uno o più processi. Infatti se:
  pid>0: il processo di pid pid
  pid=0: i processi dello stesso gruppo di chi ha invocato kill
  pid<0: i processi del gruppo -pid
  pid=-1: tutti i processi per cui l'utente ha il permesso
  Il segnale viene mandato solo se i due processi hanno lo stesso owner e se il processo che invia il segnale ha come owner il superutente(root). Restituisce 0 in caso di successo, -1 altrimenti.

- **int pthread_kill(pthread_t tid, int signum)**: genera il segnale signum e lo invia al thread avente tid "tid", che deve appartenere allo stesso processo. Bisogna fare attenzione poiché i segnali che di default terminano o uccidono lo fanno per tutto il processo, quindi è bene usare la pthread_kill solo per segnali opportunamente gestiti. Restituisce 0 in caso di successo, il numero di errore altrimenti.

- **int pause (void)**: attende finchè non viene interrotta da un segnale, in questo caso se il segnale è gestito e il gestore ritorna, la pause ritorna -1 con errore EINTR.

- **int sigwait (const sigset_t *set, int* signum)**: set permette di specificare i segnali da attendere con una maschera (come per la signal mask). Dopo la chiamata, signum contiene il segnale effetivamente ricevuto.