Sistemi Operativi

Laboratorio

Roman Simone

Appunti di
Laboratorio

Dipartimento Informatica
Università degli studi di Trento
Italia
2022

Indice

1 Lezione 1

L'obiettivo dimestichezza nell'ideazione e realizzazione di applicazioni binarie complete o almeno singole componenti impostate con il linguaggio C utilizzabili su shell bash in un sistema Linux debian/ubuntu che implementino metodi di IPC.

1.1 Terminale - Bash

Il terminale (o terminal) è l'ambiente testuale di interazione con il sistema operativo. Tipicamente è utilizzato come applicazione all'interno dell'ambiente grafico ed è possibile avviarne più istanze, pur essendo anche disponibile direttamente all'avvio (in questo caso normalmente in più istanze accessibili con la combinazione CTRL+ALT+Fx). L'interazione avviene utilizzando un'applicazione specifica in esecuzione al suo interno comunemente detta SHELL.

1.2 I comandi

Fondamentalmente si individuano 3 canali di output:

- **input** (tipicamente la tastiera), canale 0, detto **stdin**
- **output standard** (tipicamente il video), canale 1, detto **stdout**
- **output errori** (tipicamente il video), canale 2, detto **stderr**

Solitamente un comando è identificato da una parola chiave cui possono seguire uno o più *argomenti* opzionali o obbligatori, accompagnati da un valore di riferimento o meno.
Esempio : **ls -alh /tmp**

Gli argomenti nominali sono indicati con un trattino cui segue una voce (stringa alfanumerica) e talvolta presentano una doppia modalità di riferimento: breve (tipicamente voce di un singolo carattere) e lunga (tipicamente un termine mnemonico).

È anche possibile utilizzare dei **commenti** da passare alla shell. L'unico modo formale è l'utilizzo del carattere # per cui esso e tutto ciò che segue fino al termine della riga è considerato un commento ed è sostanzialmente ignorato.

1.3 Reindirizzamento

I canali possono essere reinderizzati grazie all'attributo $<$.
Esempio :ls 1 $<$ /tmp/out.txt 2$<$/tmp/err.txt

- $<$: command $<$ file.txt invia l'input al comando (file.txt read-only): mail -s "Subject" rcpt ¡ content.txt (anziché interattivo)
- $<>$: come sopra ma file.txt è aperto in read-write (raramente usato)
- **source** $>$ **target** : command1 $>$ out.txt 2 $>$ err.txt redireziona source su target:
  - **source** può essere sottinteso (vale 1)
  - **target** può essere un canale (si indica con n, ad esempio 2)
- $>$ "$|$: si comporta come $>$ ma forza la sovrascrittura anche se bloccata nelle configurazioni
- $>>$: si comporta come $>$ ma opera un append se la destinazione esiste
- $<<$ $<<<$: permettono di utilizzare testi come here-doc

1.4 Variabili

La **shell** utilizza delle variabili per memorizzare e recuperare i valori. La shell opera in un ambiente in cui ci sono alcuni riferimenti impostabili e utilizzabili attraverso l'uso delle cosiddette "variabili d'ambiente" con cui si intendono generalmente quelle con un significato particolare per la shell stessa:

- **SHELL** : contiene il riferimento alla shell corrente (path completo)
- **PATH** : contiene i percorsi in ordine di priorità in cui sono cercati i comandi, separati da ":"
- **TERM** : contiene il tipo di terminale corrente
- **PWD** : contiene la cartella corrente
- **PS1** : contiene il prompt e si possono usare marcatori speciali
- **HOME** : contiene la cartella principale dell'utente corrente

Esempio array:

- **Definizione**: lista=("a" 1 "b" 2 "c" 3)
- **Output completo**: $lista[@]
- **Accesso singolo**: $lista[x] (0-based)
- **Lista indici**: $!lista[@]
- **Dimensione**: $#lista[@]
- **Set elemento**: lista[x]=value
- **Append**: lista+=(value)
- **Sub array**: $lista[@]:s:n (from index s, length n)

Le variabili $$ e $? non possono essere impostate manualmente (la stessa sintassi lo impedisce dato che i nomi sarebbero $ e ? non utilizzabili normalmente):

- **$$** : contiene il PID del processo attuale (*)
- **$?** : contiene il codice di ritorno dell'ultimo comando eseguito

1.5 Concateazione comandi

Concatenazione comandi È possibile concatenare più comandi in un'unica riga in vari modi con effetti differenti:

- **comando1 ; comando2** concatenazione semplice: esecuzione in sequenza
- **comando1 && comando2** concatenazione logica *and*: l'esecuzione procede solo se il comando precedente non fallisce (codice ritorno zero)
- **comando1** || **comando2** concatenazione logica "or": l'esecuzione procede solo se il comando precedente fallisce (codice ritorno NON zero)
- **comando1** | **comando2** concatenazione con piping(La concatenazione con gli operatori di piping *cattura* l'output di un comando e lo passa in input al successivo es: ls | wc -l : cattura solo stdout ls |& wc -l : cattura stdout e stderr)

1.6 Confronti logici

I costrutti fondamentali per i confronti logici sono il comando test e i raggruppamenti tra parentesi quadre singole e doppie: test . . . e **[. . .]sono built-in** equivalenti mentre **[[. . .]] è una coppia di shell-keywords** In tutti i casi il blocco di confronto genera il codice di uscita 0 in caso di successo, un valore differente (tipicamente 1) altrimenti. La differenza tra built-in e shell-keywords: i **built-in** sono sostanzialmente dei comandi il cui corpo d'esecuzione è incluso nell'applicazione shell direttamente (non sono eseguibili esterni) e quindi seguono sostanzialmente le "regole generali" dei comandi, mentre le **shell-keywords** sono gestite come marcatori speciali così che possono "attivare" regole particolari di parsing. Un caso esemplificativo sono gli operatori $<$ e $>$ che normalmente valgono come redirezionamento, ma all'interno di [[. . .]] valgono come operatori relazionali.

interi		
	`[ … ]`	`[[ … ]]`
uguale-a	`-eq`	`==`
diverso-da	`-ne`	`!=`
minore-di minore-o-uguale-a	`-lt` `-le`	`<` `<=`
maggiore-di maggiore-o-uguale-a	`-gt` `-ge`	`>` `>=`

stringhe		
	`[ … ]`	`[[ … ]]`
uguale-a	= o ==	
diverso-da	!=	
minore-di (ordine alfabetico)	`\<`	`<`
maggiore-di (ordine alfabetico)	`\>`	`>`
nota: occorre lasciare uno spazio prima e dopo i "simboli" (es. non "=" ma " = ")		

Figura 1: Confronti logici - interi e stringhe

Esistono alcuni operatori unari ad esempio per verificare se una stringa è vuota o meno oppure per controllare l'esistenza di un file o di una cartella.
Esempi:

- $[[-f/tmp/prova]]$: è un file?
- $[[-e/tmp/prova]]$: file esiste?
- $[[-d/tmp/prova]]$: è una cartella?

Con il ! si nega la condizione.

1.7 SCRIPT/BATCH

È possibile raccogliere sequenze di comandi in un file di testo che può poi essere eseguito.
Esempio 1:

```
//SCRIPT "bashpid.sh":
# bashpid.sh
echo $BASHPID
echo $( echo $BASHPID)

//CLI:
chmod +x ./bashpid.sh ; echo $BASHPID ; ./bashpid.sh
```

Costrutti:

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

//While loop:
while [[ $i < 10 ]]
do
echo $i ; (( i++ ))
```

```
done

//If condition:
if [ $1 -lt 10 ]; then
echo less than 10
elif [ $1 -gt 20 ]; then
echo greater than 10
else
echo between 10 and 20
fi
```

Esempio 2:

```
//SCRIPT "args.sh":          //CLI:
#!/usr/bin/env bash             chmod +x ./args.sh
nargs=$#                        ./args.sh uno
while [[ $1 != "" ]]; do        ./args.sh uno due tre
echo "ARG=$1"
shift
done
```

1.8 Esercizi

1. Stampa “T” (per True) o “F” (per False) a seconda che il valore rappresenti un file o cartella esistente

```
([[ -f $DATA ]] && echo T) || ([[ -d $DATA ]] && echo F)
```

2. Stampa “file”, “cartella” o “?” a seconda che il valore rappresenti un file (esistente), una cartella (esistente) o una voce non presente nel file-system.

```
([[ -f $DATA ]] && [[ -e $DATA ]] && echo file) || ([[ -d $DATA ]] && [[ -e $DATA ]] &&
    echo cartella) || echo ?
```

3. Stampa il risultato di una semplice operazione aritmetica (es: ‘1 ¡ 2’) contenuta nel file indicato dal valore di DATA, oppure “?” se il file non esiste

```
([ ! -e $DATA ] && echo "?") || echo $(( $(cat $DATA) ))
```

4. Scrivere uno script che dato un qualunque numero di argomenti li restituisca in output in ordine inverso.

```
nargs=$#
lista=()

while [[ $1 != "" ]]; do
  lista+=($1)
  shift
done

for i in ${!lista[@]}; do
  echo ${lista[$nargs-$i-1]}
done
```

5. Scrivere uno script che mostri il contenuto della cartella corrente in ordine inverso rispetto all’output generato da “ls” (che si può usare ma senza opzioni)

```
lista=($(ls))
for i in ${!lista[@]}; do
  echo ${lista[${#lista[@]}-$i-1]}
done
```

2 Lezione 2

In questa lezione vedremo Docker, GCC e Make. Questi sono utilizzati per la gestione delle varie applicazioni in C. Ognuno di questi strumenti non è indispensabile ma permette di creare un flusso di lavoro coerente e strutturato.

2.1 Docker

Tecnologia di virtualizzazione a livello del sistema operativo che consente la creazione, la gestione e l'esecuzione di applicazioni attraverso **containers**. I **containers** sono ambienti leggeri,dinamici ed isolati che vengono eseguiti sopra il kernel di Linux.

2.1.1 Docker Containers vs Virtual Machine

Docker Container	Virtual Machin
Virtualizzazione a livello OS	Virtualizzazione a livello HW
Containers condividono kernel	Ogni VM ha il suo OS
Avvio e creazione in secondi	Avvio e creazione in minuti
Leggere (KB/MB)	Leggere (GB)
Si distruggono e si rieseguono	Si trasferiscono
Utilizzo leggero di risorse	Utilizzo intenso di risorse
Minore sicurezza	Maggiore sicurezza e controllo

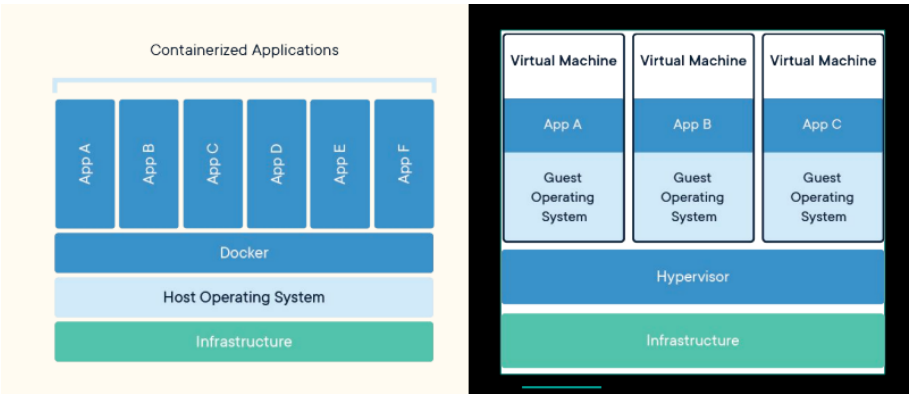

Figura 2: Docker vs VM

2.1.2 Containers e immagini

Un' immagine docker è un insieme di istruzioni per la creazione di un container. Essa consente di ragruppare varie applicazioni ed eseguirle, con una certa configurazione, in maniera rapida attraverso un container.
I container invece sono gli ambienti virtualizzati gestiti da docker, che possono essere creati, avviati, fermati ed eliminati. I container devono essere basati su un'immagine.

Gestione dei container

docker run [options] $< image >$: crea un nuovo container da un'immagine.

docker container ls [options]: : mostra i containers attivi ([-a] tutti)

docker start/stop $< container >$: avvia/ferma l'esecuzione del container.

docker exec [options] $< container > < command >$: esegue il comando all'interno del container.

docker stats : mostra le statistiche di utilizzo del containers.

docker $< command >$ $--$**help** : PER RISOLVERE TUTTI I DUBBI!!!

Parametri opzionali

- **--name** $< nome >$: assegna un nome specifico al container
- **-d** $< nome >$: detach mode, scollega il container dalla console
- **-ti** : esegue container in modalità interattiva
- **--rm** : elimina container all'uscita
- **--hostname** $< nome >$: imposta l'hostname nel container
- **--workdir** $< path >$: imposta la cartella di lavoro nel container
- **--network host** : collega il container alla rete locale
- **--privileged** : esegue il container con i privilegi dell'host

Esempi:
Esegui docker run hello-world
Esegui docker run -d -p 80:80 docker/getting-started e collegati alla pagina "localhost:80" con un qualunque browser.

Gestione delle immagini
-docker images : mostra le immagini salvate.
-docker rmi $< imageID >$: elimina un'immagine (se non in uso)
-docker search $< keyword >$: cerca un'immagine nella repository di docker
-docker commit $< container > < repository/imageName >$:crea una nuova immagine dai combiamenti nel container.

2.1.3 Dockerfile

I docker file sono dei documenti testuali che raccolgono una serie di comandi necessari alla creazione di una nuova immagine. Ogni nuova immagine sarà generata a partire da un'immagine di base, come Ubuntu o l'immagine minimale 'scrath'. La creazione a partire da un docker file viene gestita attraverso del caching che ne permette la ricompilazione rapida in caso di piccoli cambiamenti.

```
FROM ubuntu:20.04
RUN apt-get update && apt-get install build-essential nano -y
RUN mkdir /home/LabOS
CMD cd /home/labOS && bash
```

docker build -t labos/ubuntu - $<$ **dockerfile**

Docker salva i file persistenti su *bind mount* o su dei *volumi*. Sebbene i **bind mount** siano strettamente collegati con il file system dell'host OS, consentendo dunque una facile comunicazione con il containers, i **volumi** sono ormai lo standard in quanto indipendenti, facili da gestire e più in linea con la filosofia docker.

2.1.4 Sintassi dei comandi volumi

-docker volume create $< volumeName >$: crea un nuovo volume
-docker volume ls : mostra i volumi esistenti
-docker volume inspect $volumeName >$: esamina volume
-docker volume rm $volumeName >$: rimuovi volume.
-docker run -v $< volume >:< /path/in/container > < image >$: crea un nuovo container con il **volume** specificato montato nel percorso specificato
-docker run -v $< pathHost >:< /path/in/container > < image >$: crea un nuovo container con il **bind mount** specificato montato nel percorso specificato

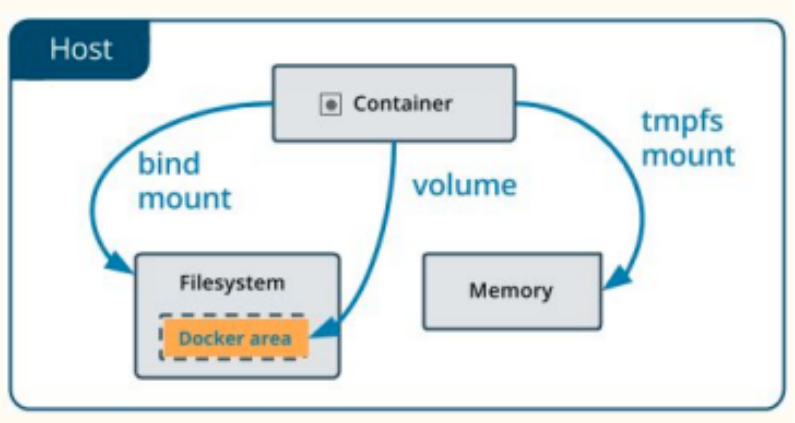

Figura 3: Volumi

2.2 gcc

GCC = Gnu Compiler Collection

Insieme di strumenti open-source che constituisce lo standard per la creazione di eseguibili su Linux. GCC supporta diversi linguaggi, tra cui C, e consente la modifica dei vari passaggi intermedi per una completa personalizzazione dell'eseguibile.

2.2.1 I comandi

Adesso vedremo la sequenza di comandi per ottenere il nostro eseguibile:

```
gcc -E <sorgente.c> -o <preProcessed.ii|.i>
gcc -S <preProcessed.i|.ii> -o <assembly.asm|.s>
gcc -c <assembly.asm|.s> -o <objectFile.obj|.o>
gcc <objectFile.obj|.o> -o <executable.out>
```

2.3 Make

Il make tool è uno strumento della collezione GNU che può essere usato per gestire la compilazione *automatica* e *selettiva* di grandi e piccoli progetti. **Make** consente di specificare delle dipendenze tra i vari file, per esempio consentendo solo la compilazione di librerie i cui sorgenti sono stati modificati. **Make** può anche essere usato per gestire il deployment di un'applicazione, assumento alcune delle capacità di uno script bash.

Make può eseguire dei makefiles i quali contengono tutte le direttive utili alla compilazione di un'applicazione, tramite il comando:

make -f makefile

In alternatica, il comando make senza argomenti processerà il file 'makefile' presente nella cartella di lavoro. Makefiles secondari possono essere inclusi nel makefile principale con delle direttive specifiche, consentendo una gestione più articolata di grandi progetti.

2.3.1 Target

Una *ricetta* è una lista di comandi che vengono eseguiti indipendentemente dal resto del makefile. I *target* sono generalmente dei files generati da uno specifico insieme di regole. Ogni target può specificare dei *prerequisiti*, ovvero degli altri file che devono esistere affinchè le regole di un target vengano eseguite. Un prerequisito può essere esso stesso un target. L'esecuzione di un makefile inizia specificando uno o più target "make -f makefile target1..." e prosegue a seconda dei vari prerequisiti.

2.3.2 Target speciali

Il target di default eseguito quando non ne viene passato alcuno è il primo disponibile.

- **.INTERMEDIATE e .SECONDARY**: hanno come prerequisiti i target "intermedi". Nel primo caso sono poi rimossi, nel secondo sono mantenuti a fine esecuzione
- **.PHONY**: ha come prerequisiti i target che non corrispondono a dei files o comunque da eseguire "sempre" senza verificare l'eventuale file omonimo.
- In un target, % sostituisce qualunque stringa. In un prerequisito corrisponde alla stringa sostituita nel target.

2.3.3 Sintassi

Un makefile è un file di testo "*plain*" in cui righe vuote e parti di testo dal carattere "#" fino alla fine della riga non in una ricetta (considerato un commento: sempre che non sia usato l'escaping con "#" o che compaia dentro una stringa con ' o ") sono ignorati. Le ricette DEVONO iniziare con un carattere di TAB (non spazi). Una ricetta che (a parte il TAB) inizia con @ non viene visualizzata in output, altrimenti i comandi sono visualizzati e poi eseguiti. Una riga con un singolo TAB è una ricetta vuota. Esistono costrutti più complessi per necessità particolari (ad esempio costrutti condizionali)

2.3.4 Variabili

Le variabili utente si definiscono con la sintassi nome:=valore o nome=valore e vengono usate con $(nome). Inoltre, possono essere sovrascritte da riga di comando con **make nome=value**. Le variabili automatiche possono essere usate all'interno delle regole per riferirsi ad elementi specifici relativi al target corrente.

2.3.5 Funzioni speciali

Ecco alcune funzioni speciali:

- $(eval ...): consente di creare nuove regole make dinamiche
- $(shell ...): cattura l'output di un commando shell
- $(wildcard *): restituisce un elenco di file che corrispondono alla stringa specificata.

2.3.6 ESEMPIO MAKE FILE

```
all: main.out
    @echo "Application compiled"
%.s: %.c
    gcc -S $< -o $@
%.out: %.s
    mkdir -p build
    gcc $< -o build/$@
clean:
    rm -rf build *.out *.s
.PHONY: clean
.SECONDARY: make.s
```

3 Lezione 3

3.1 C

Il C è un linguaggio molto usato e con molti vantaggi e svantaggi come ad esempio:

- Struttura minimale
- Poche parole chiave (con i suoi pro e contro!)
- Unix compliant, alla base di Unix, nato per scrivere Unix
- Organizzato a passi, con sorgente, file intermedi ed eseguibile finale
- Disponibilità di librerie conosciute e standard
- Nessuna struttura di alt(issim)o livello, come classi o altro
- Efficiente perché di basso livello
- Pieno controllo del programma e delle sue risorse

3.1.1 Direttive

Il compilatore, nella fase di preprocessing, elabora tutte le direttive presenti nel sorgente. Ogni direttiva viene introdotta con '#' e può essere di vari tipi:

- #include $< lib >$: copia il contenuto del file lib (cercando nelle cartelle delle librerie) nel file corrente

- #include "lib" : come sopra ma cerca prima anche nella cartella corrente

- #define VAR VAL : crea una costante VAR con il contenuto VAL, e sostituisce ogni occorrenza di VAR con VAL.

- #define MUL(A,B) A*B : dichiara una funzione con parametri A e B. Queste funzioni hanno una sintassi limitata!

- #ifdef, #ifndef, #if, else, endif : rende l'inclusione di parte di codice dipendente da una condizione.
ESEMPIO:

```c
#include <stdio.h>
#define ITER 5
#define POW(A) A*A

int main(int argc, char **argv) {
    #ifdef DEBUG
        printf("%d\n", argc);
        printf("%s\n", argv[0]);
    #endif
        int res = 1;
        for (int i = 0; i < ITER; i++){
        res *= POW(argc);
    }
    return res;
}
```

Usando i comandi:
gcc main.c -o main.out -D DEBUG=0
gcc main.c -o main.out -D DEBUG=1
Otterremmo:
./main.out 1 2 3 4.

3.1.2 Tipi e Casting

Il C è un linguaggio debolmente tipizzato che utilizza 8 tipi fondamentali. È possibile fare il casting tra tipi differenti. I tipi sono: **void** (0 byte), **char** (1 byte), **short** (2 bytes), **int** (4 bytes), **float** (4 bytes), **long** (8 bytes), **double** (8 bytes) e **long double** (8 bytes).
C supporta l'uso di stringhe che, tuttavia, corrispondono a degli array di caratteri. Gli array sono generalmente di dimensione statica e non possono essere ingranditi durante l'esecuzione del programma. Per array dinamici dovranno essere usati costrutti particolari (come malloc). Le stringhe, quando acquisite in input o dichiarate con la sintassi "stringa", terminano con il carattere '$\backslash 0$' e sono dunque di grandezza $str_len + 1$.
Esempio:

```
int nome[DIM];
long nome[] = {1,2,3,4};
char string[] = "ciao";
char string2[] = {'c','i','a','o'};
nome[0] = 22;
```

3.1.3 Puntatori

C si evolve attorno all'uso di puntatori, ovvero degli alias per zone di memorie condivise tra diverse variabili/funzioni. L'uso di puntatori è abilitato da due operatori: `'*'` ed `'&'`.
`'*'` ha significati diversi a seconda se usato in una dichiarazione o in un'assegnazione:

```
int *punt; \\Crea un puntatore ad intero\\
int valore = \\*(punt); Ottiene valore puntato\\
```

`'&'` ottiene l'indirizzo di memoria in cui è collocatauna certa variabile.

Il C consente inoltre anche di creare dei puntatori a delle funzioni: puntatori che possono contenere l'indirizzo di funzioni differenti.
Sintassi simile ma diversa!

```
\textit{float (*punt)(float,float);}
```

```
#include <stdio.h>
float xdiv(float a, float b) {
    return a/b;
}
float xmul(float a, float b) {
    return a*b;
}

void main() {
    float (*punt)(float,float);
    punt = xdiv;
    float res = punt(10,10);
    punt = &xmul;
    res = (*punt)(10,10);
    printf("%f\n", res);
}
```

3.1.4 Sizeof (operatore)

```
sizeof (type) / sizeof expression
```

Si tratta di un operatore che elabora il tipo passato come argomento (tra parentesi) o quello dell'espressione e restituisce il numero di bytes occupati in memoria. Se lo usi su un array ritornerà il numero di elementi dell'array.

```
    #include <stdio.h>
    void main() {
        int x = 10;
        printf("variable x : %lu\n", sizeof x);
        printf("expression 1/2 : %lu\n", sizeof 1/2);
        printf("int type : %lu\n", sizeof(int));
        printf("char type : %lu\n", sizeof(char));
        printf("float type : %lu\n", sizeof(float));
        printf("double type : %lu\n", sizeof(double));
    }
```

3.1.5 Main.c

A parte casi particolari (es. sviluppo moduli per kernel) l'applicazione deve avere una funzione **"main"** che è utilizzata come punto di ingresso. Il valore di ritorno è int, un intero che rappresenta il codice di uscita dell'applicazione (variabile $? in bash) ed è 0 di default se omesso. Può essere usato anche void, ma non è standard. Quando la funzione è invocata riceve normalmente in input il numero di argomenti (int argc), con incluso il nome dell'eseguibile, e la lista degli argomenti come "vettore di stringhe" (char * argv[]) [1]

Esempio di compilazione:
Compilazione: *gcc main.c -o main*
Esecuzione: *./main arg1 arg2*

```c
#include <stdio.h>
int main(int argc, char **argv) {
    printf("%d\n", argc);
    printf("%s\n", argv[0]);
    return 0;
}
```

In output si ha "3" (numero argomenti incluso il file eseguito) e "./main" (primo degli argomenti). In generale quindi argc è sempre maggiore di zero.

3.1.6 Librerie

Librerie possono essere usate attraverso la direttiva #include.
Tra le più importanti vi sono:

stdio.h : FILE, EOF, stderr, stdin, stdout, fclose(), etc...

stdlib.h : atof(), atoi(), malloc(), free(), exit(), system(), rand(), etc...

string.h : memset(), memcpy(), strncat(), strcmp(), strlen(), etc...

math.h : sin(), cos(), sqrt(), floor(), etc...

unistd.h : STDOUTFILENO, read(), write(), fork(), pipe(), etc...

fcntl.h : creat(), open(), etc... ...e ce ne sono molte altre.

3.1.7 STRUCT, UNION e ENUM

Le **struct** permettono di creare un nuovo tipo che aggrega più tipi, *esempio*:

```c
struct Books{
    char author[50];
    char title[50];
    int bookID;
} book1, book2;

struct Books book3 = {    Rowling    ,  Harry    Potter ,2};
strcpy(book1.title , Moby   Dick );
book2.bookID = 3;
```

Invece le **Unions** permettono di creare dei tipi generici che possono ospitare uno di vari tipi specificati.

```c
union Result{
    int intero;
    float decimale;
} result1, result2;

union Result result3;
result3.intero = 22;
result3.decimale = 11.5;
```

Typedef consente la definizione di nuovi tipi di variabili o funzioni.

[1] in C una stringa è in effetti un vettore di caratteri, quindi un vettore di stringhe è un vettore di vettori di caratteri, inoltre i vettori in C sono sostanzialmente puntatori (al primo elemento del vettore) → lista di argomenti spesso indicata con "char ** argv". Utile riferimento generale: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

```c
typedef unsigned int intero;
typedef struct Books{
...
} bookType;

intero var = 22; //= unsigned int var = 22;
bookType book1; //= struct Books book1;
```

Mentre **enum** lo usiamo per ridefinire un array:

```c
#include <stdio.h>
enum State {Undef = 9, Working = 1, Failed = 0};

void main() {
    enum State state=Undef;
    printf("%d\n", state); // output       9
}
```

3.1.8 Vettori e Stringhe

I **vettori** sono sequenze di elementi omogenei (tipicamente liste di dati dello stesso tipo, ad esempio liste di interi o di caratteri).
I **vettori** si realizzano attraverso un puntatore che punterà al primo elemento dell'array.
Ad esempio:

```c
int myArray[4] = {2, 0, 1}; //4 elementi da 2 bytes

char str[3] = {'c', 'i', 'a'};
str[1] = 'o'; // str[1] = *(str + 1);

//*(str+0)=*(str)=*str
```

Le **stringhe** in c sono vettori di caratteri, ossia puntatori a sequenze di bytes, la cui terminazione è definita dal valore convenzionale **0**.
Un carattere tra apici singoli. In particolare un vettore di stringhe è un vettore di vettore di caratteri e dunque:

```c
char c; #carattere
char *str; #vettore di caratteri / stringa
char **strarr; #vettore di vettore di caratteri / vettore di stringhe

//Esempio stringhe in C
#include <stdio.h>

int main(int argc, char **argv) {
    int code=0;
    if (argc<2) {
        printf("Usage: %s <carattere>\n", argv[0]);
        code=2;
    } else {
        printf("%c == %d\n", argv[1][0], argv[1][0]);
    }
    return code;
}
```

PARTE PARSING NON SO COSA SIA

3.1.9 Funzioni per le stringhe

Dato che le stringhe sono riferite con un puntatore al primo carattere non ha senso fare assegnamenti e confronti diretti, ma si devono usare delle funzioni. La libreria standard string.h ne definisce alcune come ad esempio

```c
char * strcat(char *dest, const char *src);//aggiunge src in coda a dest
char * strchr(const char *str, int c); //cerca la prima occorrenza di c in str
int strcmp(const char *str1, const char *str2); //confronta str1 con str2
size_t strlen(const char *str); //calcola la lunghezza di str
char * strcpy(char *dest, const char *src); //copia la stringa src in dst
char * strncpy(char *dest, const char *src, size_t n); //copia n caratteri dalla stringa
    src in dst
```

3.2 Files

In UNIX ci sono due modi per interagire con i file: **streams** e **file**.

- **Streams**: forniscono strumenti com e la formazione dei dati, bufferizzazione, ecc ...
- **File descriptor**: interfaccia di basso livello costituita dalle system call messe a disposizione dal kernel.

3.2.1 Streams

Utilizzando gli **streams**, un file è descritto da un puntatore a una struttura di tipo *FILE* (definita in stdio.h). I dati possono essere letti e scritti in vari modi (un carattere alla volta, una linea alla volta ecc...) ed essere interpretati di conseguenza.
Esempio:

```c
#include <stdio.h>

FILE *ptr; //Declare stream file
ptr = fopen("filename.txt","r+"); //Open
int id;
char str1[10], str2[10];

while (!feof(ptr)){ //Check end of file
    //Read int, word and word
    fscanf(ptr,"%d %s %s", &id, str1, str2);
    printf("%d %s %s\n",id,str1,str2);
}
printf("End of file\n");

fclose(ptr); //Close file
```

Modalità di accesso:

- **r**: read
- **w**: write or overwrite (create)
- **r+**: read and write
- **w+**: read and write. Create or overcreate
- **a**: write at end (create)
- **a+**: read and write at end (create)

Un altro esempio:

```c
#include <stdio.h>
#define N 10

FILE *ptr;
ptr = fopen("fileToWrite.txt","w+");
fprintf(ptr,"Content to write"); //Write content to file
rewind(ptr); // Reset pointer to begin of file
char chAr[N], inC;
fgets(chAr,N,ptr); // store the next N-1 chars from ptr in chAr
printf("   %c      %s  ",chAr[N-1], chAr);
do{
    inC = fgetc(ptr); // return next available char or EOF
    printf("%c",inC);
}while(inC != EOF); printf("\n");
fclose(ptr);
```

3.2.2 File Descriptors

Un file è descritto da un semplica **intero** (file descriptor) che punta alla respettiva entry nella file table del sistema operativo. I dati possono essere letti e scritti soltato un buffer alla volta di cui spetta al programmatore stabilire la dimentione.

Un insieme di system call permette di effettuare le operazioni di input e output mantenendo un controllo maggiore su quanto sta accadendo a prezzo di un'interfaccia meno amichevole.

Per accedere al contenuto di un file bisogna creare un canale di comunicazione con il kernel, aprendo il file con la system call open la quale localizza l'i-node del file e aggiorna la *file table* del processo.

A ogni processo è associata una tablella dei file aperti di dimensione limitata (circa 100 elementi), dove ogni elemento della tabella rappresenta un file aperto dal processo es è individuato da un indice intero(il "file descriptor").

I file descriptor 0,1 e 2 individuano normalemente standard input, output ed error .

Il kernel gestisce l'accesso ai files attraverso due strutture dati: la tabella dei files. Attivi e la tabella dei files aperti. La prima contiene una copia dell'inode di ogni file aperto (per efficienza), mentre la seconda contiene un elemento per ogni file aperto e non ancora chiuso. Questo elemento contiene:

- I/O pointer: posizione corrente nel file
- i-node pointer: Puntatore a inode corrispondente

La tabella dei file aperti può avere più elementi corrispondenti allo stesso file!

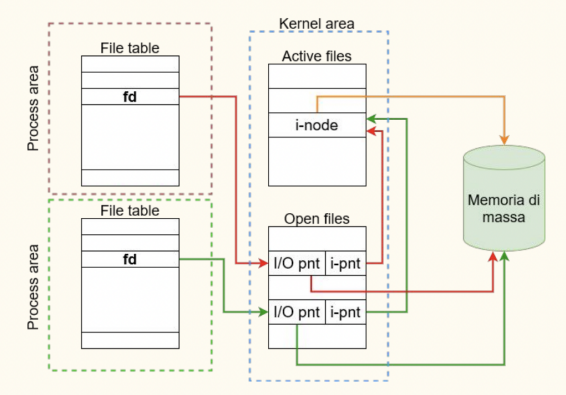

Figura 4: File Descriptor

L'input/Output Unix è basato essenzialmente su 5 funzioni: **open**, **read**, **write**,**lseek** e **close**.

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

//Open new file in Read only
int openedFile = open( filename .txt , O_RDONLY);
char content[10]; int canRead;

do{
    bytesRead = read(openedFile,content,9); //Read 9B from openedFile to buffer content
    content[bytesRead]=0;
    printf( %s ,content);
} while(bytesRead > 0);

close(openedFile);
```

```c
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

//Open file (create it with user R and W permissions)
int openFile = open( name .txt , O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);
char toWrite[] =      Professor     ;

write(openFile, "hello world\n", strlen("hello world\n")); //Write to file
lseek(openFile, 6, SEEK_SET); // riposiziona l I /O pointer
write(openFile, toWrite, strlen(toWrite)); //Write to file
close(openFile);
```

3.2.3 Open()

Per **Open()** ci sono due motalità:

- int open(const char *pathname, int flags);
- int open(const char *pathname, int flags, modet mode);

Dove **Flags** definiscono l'apertura del file con queste modalità:

- Deve contenere uno tra O_RDONLY, O_WRONLY, or O_RDWR
- O_CREAT: crea il file se non esistente
- O_APPEND: apri il file in append mode (lseek automatico con ogni write)
- O_TRUNC: cancella il contenuto del file (se aperto con W)
- O_EXCL: se usata con O_CREAT, fallisce se il file esiste già

Invece Mode definisce i privilegi da dare al file creato: S_IRUSR, S_IWUSR, S_IXUSR,S_IRWXU, S_IRGRP, . . . , S_IROTH

3.2.4 Creat() e iseek()

```
int creat(const char *pathname, mode\_t mode);
```

Alias di open(file,O_CREAT—O_WRONLY—O_TRUNC, mode)

```
off\_t lseek(int fd, off\_t offset, int whence);
```

Muove la "testina" del file di un certo offset a partire da una certa posizione:

- SEEK_SET = da inizio file,
- SEEK_CUR = dalla posizione corrente
- SEEK_END = dalla fine del file.

3.2.5 Canali standard

I canali standard (in/out/err che hanno indici 0/1/2 rispettivamente) sono rappresentati con strutture "stream" (stdin, stdout, stderr) e macro STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO). La funzione fileno restituisce l'indice di uno "stream", per cui si ha:

- fileno(stdin)=STDIN_FILENO // = 0
- fileno(stdout)=STDOUT_FILENO // = 1
- fileno(stderr)=STDERR_FILENO // = 2

isatty(stdin) == 1 (se l'esecuzione è interattiva) OPPURE 0 (altrimenti)
printf("ciao"); e fprintf(stdout, "ciao"); sono equivalenti! **Esempio**

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

void main() {
    printf("stdin: stdin ->_flags = %hd, STDIN_FILENO = %d\n",
    stdin->_flags, STDIN_FILENO
    );
    printf("stdout: stdout->_flags = %hd, STDOUT_FILENO = %d\n",
    stdout->_flags, STDOUT_FILENO
    );
    printf("stderr: stderr->_flags = %hd, STDOUT_FILENO = %d\n",
    stderr->_flags, STDERR_FILENO
    )
}
```

3.3 Funzioni/operatori

3.3.1 printf / fprintf

```
int printf(const char *format, ...)
int fprintf(FILE *stream, const char *format, ...)
```

Inviano dati sul canale stdout (printf) o su quello specificato (fprintf) secondo il formato indicato.

Il formato è una stringa contenente contenuti stampabili (testo, a capo, ...) ed eventuali segnaposto identificabili dal formato generale: %[flags][width][.precision][length]specifier Ad esempio: %d (intero con segno), %c (carattere), %s (stringa), ... Ad ogni segnaposto deve corrispondere un ulteriore argomento del tipo corretto. (rivedere esempi precedenti)

3.3.2 exit

```
void exit(int status)
```

Il processo è terminato restituendo il valore status come codice di uscita. Si ottiene lo stesso effetto se all'interno della funzione main si ha return status.

La funzione non ha un valore di ritorno proprio perché non sono eseguite ulteriori istruzioni dopo di essa.

Il processo chiamante è informato della terminazione tramite un "segnale" apposito. I segnali sono trattati più avanti nel corso

3.3.3 piping via bash

In condizioni normali l'applicazione richiamata da bash ha accesso ai canali stdin, stdout e stderr comuni (tastiera/video). Se l'applicazione è inserita via bash in un "piping"F (come in ls — wc -l) allora:

- Accede all'output del comando a sinistra da stdin
- Invia il suo output al comando di destra su stdout

```
#define MAXBUF 10
#include <stdio.h>
#include <string.h>

int main() {
    char buf[MAXBUF];
    fgets(buf, sizeof(buf), stdin); // may truncate!
    printf("%s\n", buf);
    return 0;
}
```

Esempio di una semplice applicazione che legge da stdin e stampa su stdout invertendo minuscole [a-z] con maiuscole [A-Z]

```
#include <stdio.h>

int main() {
    int c, d;
    // loop into stdin until EOF (as CTRL+D)
    while ((c = getchar()) != EOF) { // read from stdin
        d = c;
        if (c >= 'a' && c <= 'z') d -= 32;
        if (c >= 'A' && c <= 'Z') d += 32;
        putchar(d); // write to stdout
    };
    return (0);
}
```

4 Lezione 4

4.1 Kernel UNIX

Il **kernel** è l'elemento di base di un sistema Unix-like, ovvero il nucleo del sistema operativo. Il **kernel** è incaricato della gestione delle risorse essenziali: CPU, memoria, periferiche, ecc... Ad ogni boot il sistema verifica lo stato delle periferiche, monta la prima partizione (root file system) in read-only e carica il kernel in memoria.
Il **kernel** lancia il primo programma (systemd, sostituto di init) che, a seconda della configurazione voluta (target), inizializza il sistema di conseguenza. Il resto delle operazioni, tra cui l'interazione con l'utente, vengono gestite con i programmi eseguiti dal **kernel**. I programmi utilizzati dall'utente che vogliono accedere alle periferiche chiedono al **kernel** di farlo per loro. L'interazione tra programmi ed il resto del sistema viene mascherata da alcune caratteristiche intrinseche ai processori, come la gestione hardware della memoria virtuale (attraverso la MMU).
Ogni programma vede se stesso come unico possessore della CPU e non gli è dunque possibile disturbare l'azione degli altri programmi → stabilità dei sistemi Unix-like

4.2 Privilegi

Nei sistemi Unix-like ci sono due livelli di privilegi:

- **User space**: ambiente in cui vengono eseguiti i programmi.
- **Kernel space**: ambiente in cui viene eseguito il kernel.

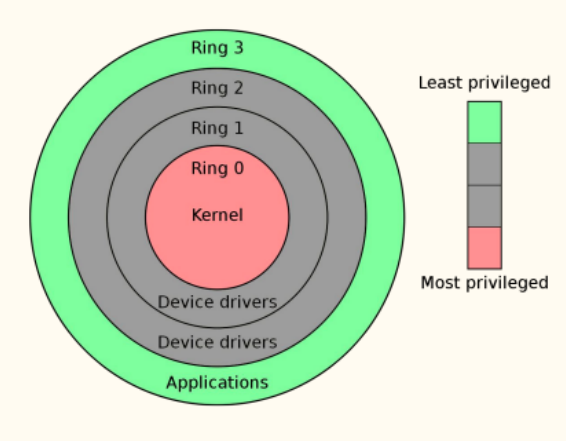

Figura 5: kernel

4.3 System calls

Le interfacce con cui i programmi accedono all'hardware si chiamano **system calls**. Letteralmente *"chiamate al sistema"* che il kernel esegue nel *kernel space*, restituendo i risultati al programma chiamante nello user space. Le chiamate restituiscono **"-1"** in caso di errore e settano la variabile golobale **errno**. Errori validi sono numeri positivi e seguono lo standard POSIX, il quale definisce degli alias.

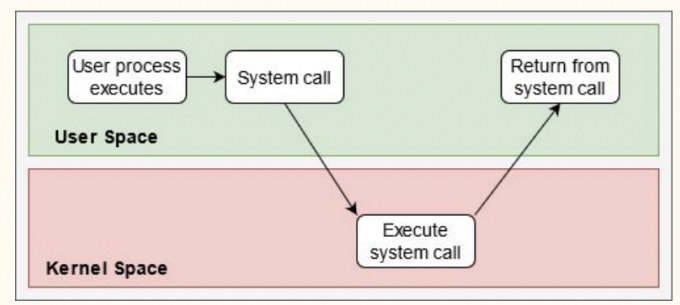

Figura 6: System calls

4.3.1 Librerie di sistema

Utilizzando il comando di shell `ldd` su di un eseguibile si possono visualizzare le librerie condivise caricate e, fra queste, vi sono tipicamente anche `ld-linux.sp` e `libc.so`

- `ls-linux.so`: quando un programma è caricato in memoria, il sistema operativo passa il controllo a **ld-linux.so** anzichè al normale punto di ingresso dell'applicazione. **ld-linux** trova e carica le librerie richieste, prepara il programma e poi gli passa il controllo.
- `libc.so`: la libreria GNU C solitamente nota come **glibc** che contiene le funzioni basilari più comuni

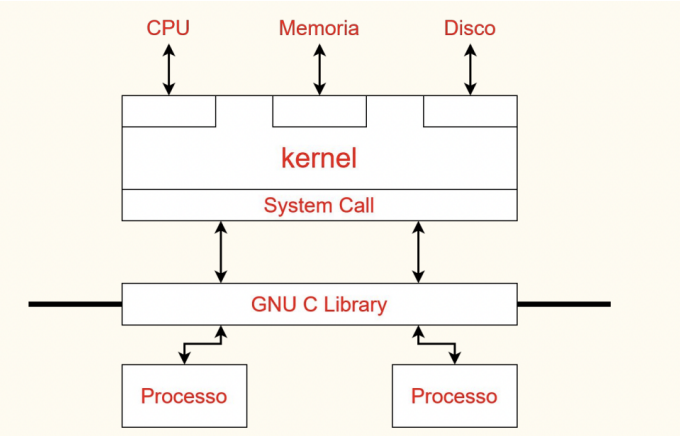

Figura 7: librerie sistema

4.3.2 Get time

Per il tempo e le date si usano le funzioni: `time` e `ctime`

```
time_t time( time_t *second )
char * ctime( const time_t *timeSeconds )
```

Esempio:

```
#include <time.h> //time.c
#include <stdio.h>

void main(){
    time_t theTime;
    time_t whatTime = time(&theTime); //seconds since 1/1/1970
    //Print date in Www Mmm dd hh:mm:ss yyyy
    printf("Current time = %s= %d\n", ctime(&whatTime),theTime);
}
```

4.3.3 Working directory

La funzione **chdir** viene utilizzata per modificare la directory di lavoro corrente del programma o del processo passando il percorso alla funzione come mostrato nella sintassi.
Mentre la funzione **getcwd()** inserisce un percorso assoluto della directory di lavoro corrente nell'array puntato da buf e restituisce buf. L'argomento size è la dimensione in byte dell'array di caratteri a cui punta l'argomento buf. Se buf è un puntatore nullo, il comportamento di **getcwd()** non è definito. Il valore restituito rappresenta la nostra directory di lavoro corrente.
Esempio:

```c
#include <unistd.h> //chdir.c
#include <stdio.h>

void main(){
    char s[100];
    getcwd(s,100); //copy path in buffer
    printf("%s\n", s); //Print current working dir
    chdir(".."); //Change working dir
    printf("%s\n", getcwd(NULL,100)); //Allocates buffer
}
```

4.3.4 Operazioni con i file

Classico visto e rivisto:

```c
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);

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

4.3.5 Duplicazione file descriptor

La funzioni **dup()** e **dup2()** creano una copia del file descriptor oldfd. La funzione **dup()** attribuisce al nuovo file descriptor, il piu' piccolo intero non usato. La funzione **dup2()** crea newfd come copia di oldfd, chiudendo prima newfd se e' necessario.

```c
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Esempio:

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

int main(void){
    char buf[51];
    int fd = open("file.txt",O_RDWR); //file exists
    int r = read(fd,buf,50); //Read 50 bytes from fd in buf
    buf[r] = 0; printf("Content: %s\n",buf);
    int cpy = dup(fd); // Create copy of file descriptor
    dup2(cpy,22); // Copy cpy to descriptor 22 (close 22 if opened)
    lseek(cpy,0,SEEK_SET); // Move I/O on all 3 file descriptors!
    write(22,"This is a fine\n",16); // Write starting from 0-pos
    close(cpy); //Close ONE file descriptor
}
```

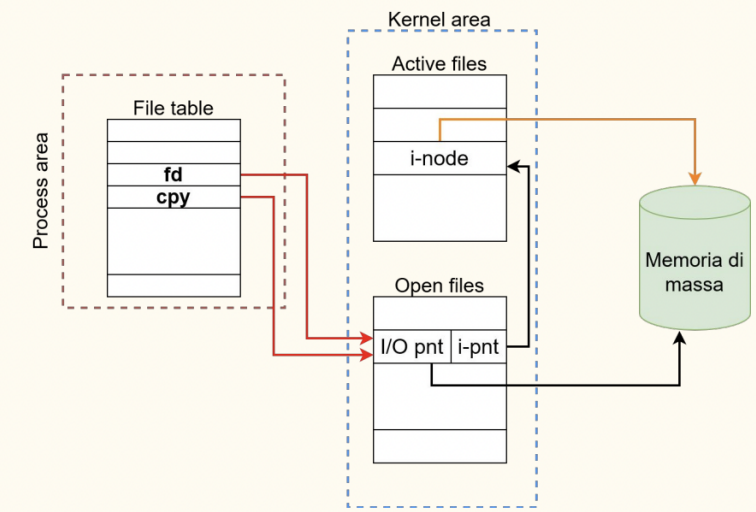

Figura 8: kernel

4.3.6 Permessi: chmod(), chown()

chmod è il comando che modifica i permessi (lettura, scrittura, esecuzione). Mentre il comando **chown** è utilizzato per cambiare l'utente proprietario e/o il gruppo assegnato ad un file o ad una directory.

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

Esempio:

```
#include <fcntl.h> //chown.c
#include <unistd.h>
#include <sys/stat.h>

void main(){
    int fd = open("file",O_RDONLY);
    fchown(fd, 1000, 1000); // Change owner to user:group 1000:1000
    chmod("file",S_IRUSR|S_IRGRP|S_IROTH); // Permission to r/r/r
}
```

4.3.7 Eseguire programmi: execve()

Le funzioni **execv** sostituiscono il processo corrente con un nuovo processo.

```
int execv(const char *path, char *const argv[])
int execvp(const char *file, char *const argv[])
int execvpe(const char *file, char *const argv[],char *const envp[])
int execl(const char *path, const char * arg0, ,argn,NULL)
int execlp(const char *file, const char * arg0, ,argn,NULL)
int execle(const char *file, const char * arg0, ,argn,
 NULL, char *const envp[])
int execve(const char *filename, char *const argv[], char
 *const envp[])
```

Esempio **execv**:

```
#include <unistd.h> //execv1.out
#include <stdio.h>

void main(){
    char * argv[] = {"par1","par2",NULL};
    execv("./execv2.out",argv); //Replace current process
    printf("This is execv1\n");
}

#include <stdio.h> //execv2.out

```

```c
void main(int argc, char ** argv){
    printf("This is execv2 with %s and %s\n",argv[0],argv[1]);
}
```

Esempio **execle**:

```c
#include <unistd.h> //execle1.out
#include <stdio.h>

void main(){
    char * env[] = {"CIAO=hello world",NULL};
    execle("./execle2.out","par1","par2",NULL,env); //Replace proc.
    printf("This is execle1\n");
}

#include <stdio.h> //execle2.out
#include <stdlib.h>

void main(int argc, char ** argv){
    printf("This is execv2 with par: %s and %s. CIAO = %s\n",argv[0],argv[1],getenv("
    CIAO"));
}
```

Esempio **dup2/exec**:

```c
#include <stdio.h> //execvpDup.c
#include <fcntl.h>
#include <unistd.h>

void main() {
    int outfile = open("/tmp/out.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR
    );
    dup2(outfile, 1); // copy outfile to FD1
    char *argv[]={"./time.out",NULL}; //time.out della slide#10
    execvp(argv[0],argv); // Replace current process
}
```

4.3.8 Chiamare la shell: system()

La system call **system()** esegue il comando specificato nella stringa string, chiamando la shell con la sintassi: /bin/sh -c string (versione UNIX)

```c
int system(const char * string);
```

Esempio:

```c
#include <stdlib.h> <stdio.h> //system.c
#include <sys/wait.h> /* For WEXITSTATUS */

void main(){
    int outcome = system("echo ciao"); // execute command in shell
    printf("Outcome = %d\n",outcome);
    outcome = system("if [[ $PWD < \"ciao\" ]]; then echo min; fi");
    printf("Outcome = %d\n",outcome);
    outcome = system("notExistingCommand");
    printf("Outcome = %d\n",WEXITSTATUS(outcome));
}
```

4.4 Forking

La syscall principale per il forking è **fork**. Il forking è la "generazione" di nuovi processi (uno alla volta) partendo da uno esistente. Un processo attivo invoca la syscall e così il kernel lo *clona* modificando però alcune informazioni e in particolare quelle che riguardano la sua collocazione nella gerarchia complessiva dei processi. Il processo che effettua la chiamata è definito **padre**, quello generato è definito **figlio**. Sono clonati gli elementi principali come il PC (Program Counter), i registri, la tabella dei file (file descriptors) e i dati di processo (variabili). Le meta-informazioni come il "pid" e il "ppid" sono aggiornate (al contrario di execve()!). L'esecuzione procede per entrambi (quando saranno schedulati!) da PC+1 (tipicamente l'istruzione seguente il **fork** o la valutazione dell'espressione in cui essa è utilizzata).

Ad ogni processo è associato un identificativo univoco per istante temporale, sono organizzati gerarchicamente (padre-figlio) e suddivisi in insiemi principali (sessioni) e secondari (gruppi). Anche gli utenti hanno un loro identificativo e ad ogni processo ne sono abbinati due: quello reale e quello effettivo (di esecuzione).

- **PID** - Process ID
- **PPID** - Parent process ID
- **SID** - Session id
- **PGID** - Process Group ID
- **UID/RUID** - User ID
- **EUID** - Effective User ID

4.4.1 GETPID(), GETPPID()

Per ottenere il **PID** del processo attivo e del processo padre si usano i comandi: **getpid()** e **getppid()**

```
pid_t getpid() // restituisce il PID del processo attivo
pid_t getppid() // restituisce il PID del processo padre
```

Esempio:[2]

```
#include <stdio.h> <unistd.h> <stdlib.h> //ppid.c

void main(){
    printf("Subshell $$ = ");
    fflush(stdout); // Forza l output di printf
    system("echo $$"); // subshell
    printf("PID: %dPPID: %d\n",getpid(),getppid());
}
```

4.4.2 Valore di ritorno

La funzione restituisce un valore che solitamente è catturato in una variabile (o usato comunque in un'espressione). Come per tutte le sys-call in generale, il valore è **-1** in caso di **errore** (in questo caso non ci sarà nessun nuovo processo, ma solo quello che ha invocato la chiamata).
Se ha **successo** entrambi i processi ricevono un valore di ritorno, ma questo è diverso nei due casi:

- Il processo padre riceve come valore il nuovo **PID** del processo figlio
- Il processo figlio riceve come valore **0**

4.4.3 Relazione tra i processi

I processi padre-figlio, conoscono reciprocamente il loro PID (ciascuno conosce il proprio tramite **getpid()**, il figlio conosce quello del padre con **getppid()**, il padre conosce quello del figlio come valore di ritorno di **fork()**).
Si possono usare altre syscall per semplici interazioni come **wait** e **waitpid**.
Eventuali variabili definite prima del fork sono valorizzate allo stesso modo in entrambi: se riferiscono risorse (ad esempio un "file descriptor" per un file su disco) fanno riferimento esattamente alla stessa risorsa.

[2](includendo <sys/types.h> e <sys/wait.h>: pid_t è un intero che rappresenta un id di processo)

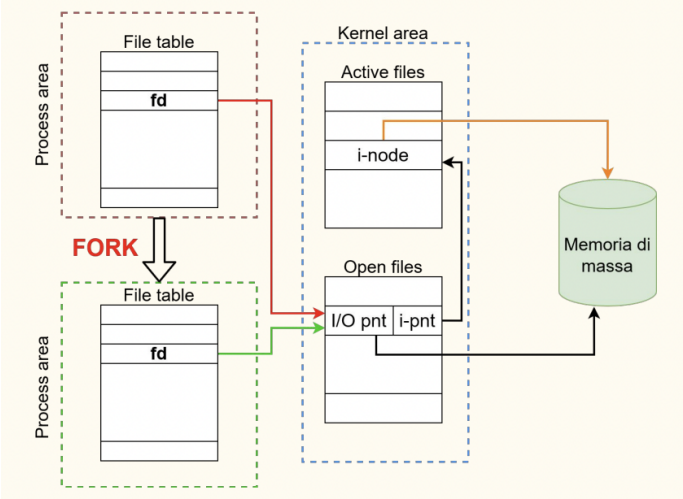

Figura 9: Confronti logici - interi e stringhe

4.4.4 Wait() e Waitpit()

pid_t wait (int *status) : attende la terminazione di UN figlio (qualunque) e ne restituisce il PID, riportando lo status nel puntatore passato come argomento (se non NULL).
pid_t waitpid(pid_t pid, int *status, int options) : analoga a wait ma consente di passare delle opzioni e si può specificare come pid:[3]

- **-n** (<-1: attende un qualunque figlio il cui "gruppo" è |-n|)
- **-1** (attende un figlio qualunque)
- **0** (attende un figlio con lo stesso "gruppo" del padre)
- **n** (>0: attende il figlio il cui pid è esattamente n)

Lo stato di ritorno è un numero che comprende più valori "composti" interpretabili con apposite macro, molte utilizzabili a mo' di funzione (altre come valore) passando lo "stato" ricevuto come risposta come ad esempio:

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

Esempio fork:

```c
#include <stdio.h> //fork1.c
#include <unistd.h>

int main() {
    fork(); fork(); fork();
    printf("hello\n");
    return 0;
}
```

[3] wait(st) corrisponde a waitpid(-1, st, 0) while(wait(NULL)¿0); attende tutti i figli

Esempio fork wait:

```c
#include <stdio.h> <stdlib.h> <unistd.h> <time.h> <sys/wait.h> //fork2.c

int main() {
    int fid=fork(), wid, st, r; // Generate child
    srand(time(NULL)); // Initialise random
    r=rand()%256; // Get random between 0 and 255
    if (fid==0) { //If it is child
        printf("Child... (%d)", r); fflush(stdout);
        sleep(3); // Pause execution for 3 seconds
        printf(" done!\n");
        exit(r); // Terminate with random signal
    } else { // If it is parent
        printf("Parent...\n");
        wid=wait(&st); // wait for ONE child to terminate
        printf("...child's id: %d==%d (st=%d)\n", fid, wid, WEXITSTATUS(st));
    }
}
```

4.4.5 Processi ”zombie” e ”orfani”

Normalmente quando un processo termina il suo stato di uscita è *catturato* dal padre: alla terminazione il sistema tiene traccia di questo insieme di informazioni (lo stato) fino a che il padre le utilizza consumandole (con wait o waitpid). Se il padre non cattura lo stato d'uscita i processi figli sono definiti **zombie** (in realtà non ci sono più, ma esiste un riferimento in sospeso nel sistema). Se un padre termina prima del figlio, quest'ultimo viene definito **orfano** e viene adottato dal processo principale (tipicamente ”init” con pid pari a 1). Un processo zombie che diventa anche orfano è poi gestito dal processo che lo adotta (che effettua periodicamente dei wait/waitpid appositamente)

5 Lezione 5

5.1 Segnali

Ci sono vari eventi che possono avvenire in maniera asincrona al normale flusso di un programma, alcuni dei quali in maniera inaspettata e non predicibile. Per esempio, durante l'esecuzione di un programma ci può essere una richiesta di terminazione o di sospensione da parte di un utente, la terminazione di un processo figlio o un errore generico. Unix prevede la gestione di questi eventi attraverso i **segnali**: quando il sistema operativo si accorge di un certo evento, genera un segnale da mandare al processo interessato il quale potrà decidere (nella maggior parte dei casi) come comportarsi.
Il numero dei segnali disponibili cambia a seconda del sistema operativo, con Linux che ne definisce 32. Ad ogni segnale corrisponde sia un valore numerico che un'etichetta mnemonica (definita nella libraria `signal.h`) nel formato **SIGXXX**.

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

Per ogni processo, all'interno della *process table*, vengono mantenute due liste:

- Pending signals: segnali emessi che il processo dovrà gestire.
- Blocked signals: segnali non comunicati al processo.

Ad ogni schedulazione del processo le due liste vengono controllate per consentire al processo di reagire nella maniera più adeguata. I segnali sono anche detti **"software interrupts"** perchè sono, a tutti gli effetti, delle interruzioni del normale flusso del processo generate dal sistema operativo (invece che dall'hardware, come per gli hardware interrupts). Come per gli interrupts, il programma può decidere come gestire l'arrivo di un segnale (presente nella lista pending):

- **Eseguendo** l'azione default.
- **Ignorandolo** (non sempre possibile) → programma prosegue normalmente.
- **Eseguendo un handler personalizzato** → programma si interrompe.

Ogni segnale ha un suo **handler** di default che tipicamente può:

- **Ignorare** il segnale
- **Terminare** il processo
- **Continuare** l'esecuzione
- **Stoppare** il processo

Ogni processo può sostituire il gestore di default con una funzione **custom** (a parte per `SIGKILL` e `SIGSTOP`) e comportarsi di conseguenza. La sostituzione avviene tramite la system call `signal()` (definita in "signal.h").

5.1.1 System call

```c
sighandler_t signal(int signum, sighandler_t handler);
typedef void (*sighandler_t)(int);

#include <signal.h> <stdio.h> <stdlib.h>

void myHandler(int sigNum){
    printf("CTRL+Z\n");
}
void main(){
    signal(SIGINT,SIG_IGN); //Ignore signal
    signal(SIGCHLD,SIG_DFL); //Use default handler
    signal(SIGTSTP,myHandler); //Use myHandler
}
```

Esempio:

```c
#include <signal.h> //sigCST.c
#include <stdio.h>

void myHandler(int sigNum){
    printf("CTRL+Z\n");
    exit(2);
}
int main(void){
    signal(SIGTSTP,myHandler);
    while(1);
}

#include <signal.h> //sigDFL.c
int main(){
    signal(SIGTSTP,SIG_DFL);
    while(1);
}

#include <signal.h> //sigIGN.c
int main(){
    signal(SIGTSTP,SIG_IGN);
    while(1);
}
```

5.1.2 Custom handler

Un **handler** personalizzato deve essere una funzione di tipo **void** che accetta come argomento un intero, il quale rappresenta il segnale catturato. Questo consente l'utilizzo di un solo **handler** per segnali differenti.

```c
#include <signal.h> <stdio.h> //param.c

void myHandler(int sigNum){
    if(sigNum == SIGINT) printf("CTRL+C\n");
    else if(sigNum == SIGTSTP) printf("CTRL+Z\n");
}

signal(SIGINT,myHandler);
signal(SIGTSTP,myHandler);
```

5.1.3 Signal() return

`signal()` restituisce un riferimento all'handler che era precedentemente assegnato al segnale:

- `NULL`: handler precedente era l'handler di default
- `1`: l'handler precedente era SIG_IGN
- `address`: l'handler precedente era *(address)

```c
#include <signal.h> <stdio.h> //return.c

void myHandler(int sigNum){}
int main(){
    printf("DFL: %p\n",signal(SIGINT,SIG_IGN));
    printf("IGN: %p\n",signal(SIGINT,myHandler));
    printf("Custom: %p == %p\n",signal(SIGINT,SIG_DFL),myHandler);
}
```

Alcuni segnali:

SIGXXX	description	default
SIGALRM	(alarm clock)	quit
SIGCHLD	child terminated	ignore
SIGCONT	continue if stopped	ignore
SIGINT	terminal interrupt, CTRL+C	quit
SIGKILL	kill process	quit
SIGSYS	bad argument to syscall	quit whith dump
SIGTERM	usoftware termination	quit
SIGUSR1/2	user signal 1/2	quit
SIGSTOP	stopped	quit
SIGTSTP	terminal stop CTRL+Z	quit

Esempio:

```c
#include <signal.h> <stdio.h> <unistd.h> <sys/wait.h> //child.c

void myHandler(int sigNum){
    printf("Child terminated! Received %d\n",sigNum);
}
int main(){
    signal(SIGCHLD,myHandler);
    int child = fork();
    if(!child){
        return 0; //terminate child
    }
    while(wait(NULL)>0);
}
```

5.1.4 Inviare i segnali: kill()

```c
int kill(pid_t pid, int sig);
```

Invia un segnale ad uno o più processi a seconda dell'argomento **pid**:

- **pid > 0**: segnale al processo con PID=pid
- **pid = 0**: segnale ad ogni processo dello stesso gruppo
- **pid = -1**: segnale ad ogni processo possibile (stesso UID/RUID)
- **pid < -1**: segnale ad ogni processo del gruppo |pid|

Restituisce **0** se il segnale viene inviato, **-1** in caso di errore. Ogni tipo di segnale può essere inviato, non deve essere necessariamente un segnale corrispondente ad un evento effettivamente avvenuto!
Esempio:

```c
#include <signal.h> <stdio.h> <stdlib.h> <sys/wait.h> <unistd.h>
//kill.c

void myHandler(
    int sigNum){printf("[%d]ALARM!\n",getpid());
}

int main(void){
    signal(SIGALRM,myHandler);
    int child = fork();
    if (!child) while(1); // block the child
    printf("[%d]sending alarm to %d in 3 s\n",getpid(),child);
    sleep(3);
    kill(child,SIGALRM); // send ALARM, childs handler reacts
    printf("[%d]sending SIGTERM to %d in 3 s\n",getpid(),child);
    sleep(3);
    kill(child,SIGTERM); // send TERM: default is to terminate
    while(wait(NULL)>0);
}
```

5.1.5 Kill da bash

kill è anche un programma in **bash** che accetta come primo argomento il **tipo** di segnale (kill -l per la lista) e come secondo argomento il **PID** del processo.

```c
#include <signal.h> <stdio.h> <stdlib.h> <unistd.h> //bash.c

void myHandler(int sigNum){
    printf("[%d]ALARM!\n",getpid());
    exit(0);
}

int main(){
    signal(SIGALRM,myHandler);
    printf("I am %d\n",getpid());
    while(1);
}
```

Da bash scriveremo:

```
$ gcc bash.c -o bash.out
$ ./bash.out
# On new window/terminal
$ kill -14 <PID>
```

5.1.6 Programmare un alarm: alarm()

alarm() - imposta un allarme temporizzato per l'invio di un certo segnale.

```c
unsigned int alarm(unsigned int seconds);
```

Esempio:

```c
#include <signal.h> <stdio.h> <stdlib.h> <unistd.h> //alarm.c

short cnt = 0;
void myHandler(int sigNum){printf("ALARM!\n"); cnt++;}

int main(){
    signal(SIGALRM,myHandler);
    alarm(5); //Set alarm in 5 seconds
    //Set new alarm (cancelling previous one)
    printf("Seconds remaining to previous alarm %d\n",alarm(2));
    while(cnt<1);
}
```

5.1.7 Mettere in pausa: pause()

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

//pause.c
void myHandler(int sigNum){
    printf("Continue!\n");
}
int main(){
    signal(SIGCONT,myHandler);
    signal(SIGUSR1,myHandler);
    pause();
}
```

In **bash** scriveremo:

```
$ gcc pause.c -o pause.out
$ ./pause.out
# On new window/terminal
$ kill -18/-10 <PID>
```

5.1.8 Bloccare i segnali

Oltre alla lista dei "pending signal" esiste la lista dei **"blocked signals"**, ovvero dei segnali ricevuti dal processo ma volutamente non gestiti. Mentre i segnali ignorati non saranno mai gestiti, i segnali bloccati sono solo temporaneamente non gestiti. Un segnale bloccato rimane nello stato pending

fino a quando esso non viene gestito oppure il suo handler tramutato in ignore. L'insieme dei segnali che vanno bloccati è detto **"signal mask"**, una maschera dei segnali che è modificabile attraverso la system call **sigprocmask()**.

5.1.9 Bloccare i segnali: sigset_t

Una signal mask può essere gestita con un **sigset_t**, ovvero una lista di segnali modificabile con alcune funzioni. Queste funzioni modificano il **sigset_t**, non la maschera dei segnali del processo!

```
int sigemptyset(sigset_t *set); //Svuota
int sigfillset(sigset_t *set); //Riempie
int sigaddset(sigset_t *set, int signo); //Aggiunge singolo
int sigdelset(sigset_t *set, int signo); //Rimuove singolo
int sigismember(const sigset_t *set, int signo); //Interpella
```

5.1.10 Bloccare i segnali: sigprocmask()

```
int sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict oldset);
```

A seconda del valore di **how** e di **set**, la maschera dei segnali del processo viene cambiata. Nello specifico:

- **how = SIG_BLOCK**: i segnali in set sono aggiunti alla maschera;
- **how = SIG_UNBLOCK**: i segnali in set sono rimossi dalla maschera;
- = SIG_SETMASK: set diventa la maschera.

Se **oldset** non è nullo, in esso verrà salvata la vecchia maschera (anche se **set** è nullo
Esempio 1:

```
#include <signal.h>

int main(){
    sigset_t mod,old;
    sigfillset(&mod); // Add all signals to the blocked list
    sigemptyset(&mod); // Remove all signals from blocked list
    sigaddset(&mod,SIGALRM); // Add SIGALRM to blocked list
    sigismember(&mod,SIGALRM); // is SIGALRM in blocked list?
    sigdelset(&mod,SIGALRM); // Remove SIGALRM from blocked list

    // Update the current mask with the signals in    mod    sigprocmask(SIG_BLOCK,&mod
    ,&old);
}
```

Esempio 2:

```
#include <signal.h> <unistd.h> <stdio.h> //sigprocmask.c
sigset_t mod, old;
int i = 0;

void myHandler(int signo){
    printf("signal received\n");
    i++;
}

int main(){
    printf("my id = %d\n",getpid());
    signal(SIGUSR1,myHandler);
    sigemptyset(&mod); //Initialise set
    sigaddset(&mod,SIGUSR1);
    while(1) if(i==1) sigprocmask(SIG_BLOCK,&mod,&old);
}
```

5.1.11 Verificare pending signals: sigpendin()

La funzione **sigpending()** restituisce segnali che sono bloccati dalla consegna e in sospeso per il thread chiamante o per il processo.

```
int sigpending(sigset_t *set);
```

Esempio:

```
//sigpending.c
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

sigset_t mod,pen;
void handler(int signo){
    printf("SIGUSR1 received\n");
    sigpending(&pen);
    if(!sigismember(&pen,SIGUSR1))
     printf("SIGUSR1 not pending\n");
    exit(0);
}

int main(){
    signal(SIGUSR1,handler);
    sigemptyset(&mod);
    sigaddset(&mod,SIGUSR1);
    sigprocmask(SIG_BLOCK,&mod,NULL);
    kill(getpid(),SIGUSR1);
    // sent but its blocked...
    sigpending(&pen);
    if(sigismember(&pen,SIGUSR1))
        printf("SIGUSR1 pending\n");
    sigprocmask(SIG_UNBLOCK,&mod,NULL);
    while(1);
}
```

5.1.12 sigaction()

La chiamata di sistema **sigaction()** viene utilizzata per modificare l'azione intrapresa da un processo alla ricezione di un segnale specifico

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


struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask; //Signals blocked during handler
    int sa_flags; //modify behaviour of signal
    void (*sa_restorer)(void); //Deprecated
};
```

Esempio 1:

```
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h> //sigaction.c
void handler(int signo){
    printf("signal received\n");
}

int main(){
    struct sigaction sa; //Define sigaction struct
    sa.sa_handler = handler; //Assign handler to struct field
    sigemptyset(&sa.sa_mask); //Define an empty mask
    sigaction(SIGUSR1,&sa,NULL);
    kill(getpid(),SIGUSR1);
}
```

Esempio: blocking signal:

```
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h> //sigaction2.c
void handler(int signo){
    printf("signal %d received\n",getpid());
```

```
    sleep(2);
    printf("Signal done\n");
}

int main(){
    printf("Process id: %d\n",getpid());
    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask); //Use an empty mask     block no signal
    sigaction(SIGUSR1,&sa,NULL);
    while(1);
}

\\BASH
$ kill -10 <PID> ; sleep 1
&& kill -12 <PID>
```

Esempio: blocking signal

```
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h> //sigaction3.c

void handler(int signo){
    printf("signal %d received\n",getpid());
    sleep(2);
    printf("Signal done\n");
}

int main(){
    printf("Process id: %d\n",getpid());
    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask,SIGUSR2); // Block SIGUSR2 in handler
    sigaction(SIGUSR1,&sa,NULL);
    while(1);
}

\\BASH
$ kill -10 <PID> ; sleep 1
 && kill -12 <PID>
```

Esempio: sa_sigaction:

```
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h> //sigaction4.c

void handler(int signo, siginfo_t * info, void * empty){
    //print id of process issuing the signal
    printf("Signal received from %d\n",info->si_pid);
}
int main(){
    struct sigaction sa;
    sa.sa_sigaction = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags |= SA_SIGINFO; // Use sa_sigaction
    sa.sa_flags |= SA_RESETHAND; // Restore def handler afterward
    sigaction(SIGUSR1,&sa,NULL);
    while(1);
}

\\BASH
$ echo $$ ; kill -10 <PID> # custom
$ kill -10 <PID> # default
```

6 Lezione 6

6.0.1 Process groups

All'interno di Unix i processi vengono raggruppati secondi vari criteri, dando vita a sessioni, gruppi e threads.

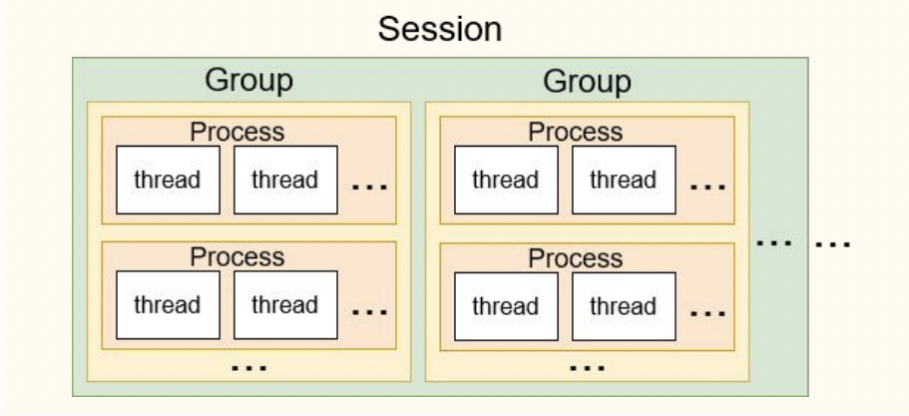

Figura 10: Sessioni

I **process groups** consentono una migliore gestione dei segnali e della comunicazione tra i processi.
Un processo, per l'appunto, può:

- **Aspettare** che tutti i processi figli appartenenti ad un determinato gruppo terminino;
- **Mandare un segnale** a tutti i processi appartenenti ad un determinato gruppo.

```
waitpid(-33,NULL,0); // Wait for a children in group 33 (|-33|)
kill(-45,SIGTERM); // Send SIGTERM to all children in group 45
```

6.0.2 Gruppi in Unix

Mentre, generalmente, una sessione è collegata ad un terminale, i processi vengono raggruppati nel seguente modo:

- In bash, processi concatenati tramite pipes appartengono allo stesso gruppo: cat /tmp/-ciao.txt — wc -l — grep '2'
- Alla loro creazione, i figli di un processo ereditano il gruppo del padre
- Inizialmente, tutti i processi appartengono al gruppo di 'init', ed ogni processo può cambiare il suo gruppo in qualunque momento.

Il processo il cui **PID** è uguale al proprio **GID** è detto **process group leader**.

6.0.3 Group system calls

`getpgrp (void)`, per recuperare il **PGID** del processo chiamante; e `setpgid()`, per impostare il **PGID** di un processo.

```
int setpgid(pid_t pid, pid_t pgid); //set GID of proc. (0=self)
pid_t getpgid(pid_t pid); // get GID of process (0=self)
```

Esempio:

```
#include <stdio.h> <unistd.h> <sys/wait.h> //setpgid.c
int main(void){
    int isChild = !fork(); //new child
    printf("PID %d PPID: %d GID %d\n",getpid(),getppid(),getpgid(0));
    if(isChild){
```

```
        isChild = !fork(); //new child
        if(!isChild) setpgid(0,getpid()); // Become group leader
        sleep(1);
        fork(); //new child
        printf("PID %d PPID: %d GID %d\n",getpid(),getppid(),getpgid(0));
    }; while(wait(NULL)>0);
}
```

6.0.4 Mandare segnali ai gruppi

Nel prossimo esempio:

1. Processo 'ancestor' crea un figlio
   (a) Il figlio cambia il proprio gruppo e genera 3 figli (Gruppo1)
   (b) I 4 processi aspettano fino all'arrivo di un segnale
2. Processo 'ancestor' crea un secondo figlio
   (a) Il figlio cambia il proprio gruppo e genera 3 figli (Gruppo2)
   (b) I 4 processi aspettano fino all'arrivo di un segnale
3. Processo 'ancestor' manda due segnali diversi ai due gruppi

```
#include <stdio.h><unistd.h><sys/wait.h><signal.h><stdlib.h>//gsignal.c

void handler(int signo){
    printf("[%d,%d] sig%d received\n",getpid(),getpgid(0),signo);
    sleep(1); exit(0);
}

int main(void){
    signal(SIGUSR1,handler);
    signal(SIGUSR2,handler);
    int ancestor = getpid(); int group1 = fork(); int group2;
    if(getpid()!=ancestor ){ // First child
        setpgid(0,getpid()); // Become group leader
        fork();fork(); //Generated 3 children in new group
    }
    else{
        group2 = fork();
        if(getpid()!=ancestor){ // Second child
            setpgid(0,getpid()); // Become group leader
            fork();fork();} //Generated 3 children in new group
    if(getpid()==ancestor){
        printf("[%d]Ancestor and I'll send signals\n",getpid());
        sleep(1);
        kill(-group2,SIGUSR2); //Send SIGUSR2 to group2
        kill(-group1,SIGUSR1); //Send SIGUSR1 to group1
    }else{
        printf("[%d,%d]chld waiting signal\n", getpid(),getpgid(0));
        while(1);
    }
    while(wait(NULL)>0);
    printf("All children terminated\n");
}
```

6.0.5 Wait figli in un gruppo

Nel prossimo esempio:

1. Processo ancestor crea un figlio
   (a) Il figlio cambia il proprio gruppo e genera 3 figli (Gruppo1)
   (b) I 4 processi aspettano 2 secondi e terminano
2. Processo 'ancestor' crea un secondo figlio
   (a) Il figlio cambia il proprio gruppo e genera 3 figli (Gruppo2)

(b) I 4 processi aspettano 4 secondi e terminano

3. Processo ‘ancestor’ aspetta la terminazione dei figli del gruppo1
4. Processo ‘ancestor’ aspetta la terminazione dei figli del gruppo2

```c
#include <stdio.h><unistd.h><sys/wait.h> //waitgroup.c

int main(void){
    int group1 = fork(); int group2;
    if(group1 == 0){ // First child
        setpgid(0,getpid()); // Become group leader
        fork();fork(); //Generated 4 children in new group
        sleep(2); return; //Wait 2 sec and exit
    }else{
        group2 = fork();
        if(group2 == 0){
            setpgid(0,getpid()); // Become group leader
            fork();fork(); //Generated 4 children
            sleep(4); return; //Wait 4 sec and exit
        }
    }
    sleep(1); //make sure the children changed their group
    while(waitpid(-group1,NULL,0)>0);
    printf("Children in %d terminated\n",group1);
    while(waitpid(-group2,NULL,0)>0);
    printf("Children in %d terminated\n",group2);
}
```

7 Lezione 7

7.1 Errors in C

Durante l'esecuzione di un programma ci possono essere diversi tipi di errori: **system calls** che **falliscono**, divisioni per zero, problemi di memoria etc...
Alcuni di questi errori non fatali, come una system call che fallisce, possono essere indagati attraverso la variabile `errno`. Questa variabile globale contiene l'ultimo codice di errore generato dal sistema.
Per convertire il codice di errore in una stringa comprensibile si può usare la funzione `char *strerror(int errnum)`.
In alternativa, la funzione `void perror(const char *str)` che stampa su stderr la stringa passatagli come argomento concatenata, tramite **":"**, con `strerror(errno)`.
Esempio: errore apertura file:

```c
#include <stdio.h> <errno.h> <string.h> //errFile.c
extern int errno; // declare external global variable

int main(void){
    FILE * pf;
    pf = fopen ("nonExistingFile.boh", "rb"); //Try to open file
    if (pf == NULL) { //something went wrong!
        fprintf(stderr, "errno = %d\n", errno);
        perror("Error printed by perror");
        fprintf(stderr,"Strerror: %s\n", strerror(errno));
    } else {
        fclose (pf);
    }
}
```

Esempio: errore processo non esistente:

```c
#include <stdio.h> <errno.h> <string.h> <signal.h> //errSig.c
extern int errno; // declare external global variable

int main(void){
    int sys = kill(3443,SIGUSR1); //Send signal to non existing proc
    if (sys == -1) { //something went wrong!
        fprintf(stderr, "errno = %d\n", errno);
        perror("Error printed by perror");
        fprintf(stderr,"Strerror: %s\n", strerror(errno));
    } else {
        printf("Signal sent\n");
    }
}
```

7.2 Pipe anonime

Il **piping** connette l'output (**stdout** e **stderr**) di un comando all'input (**stdin**) di un altro comando, consentendo dunque la comunicazione tra i due.
Esempio:

```
ls . | sort -R #stout -> stdin
ls nonExistingDir |& wc #stdout e stderr -> stdin
cat /etc/passwd | wc | less #out -> in, out-> in
```

I processi sono eseguiti in concorrenza utilizzando un buffer:

- Se **pieno** lo scrittore (left) si sospende fino ad avere spazio libero
- Se **vuoto** il lettore si sospende fino ad avere i dati

Esempio:

```c
// output.out
#include <stdio.h>
#include <unistd.h>
int main(){
    for (int i = 0; i<3; i++) {
        sleep(2);
```

```
        fprintf(stdout, "Written in buffer");
        fflush(stdout);
    };
};

// input.out
#include <stdio.h>
#include <unistd.h>
int main() {
    char msg[50]; int n=3;
    while((n--)>0){
        int c = read(0,msg,50);
        if (c>0) {
            msg[c]=0;
            fprintf(stdout, "Read: '%s' (%d)\n",msg,c);
        };
    };
};

//BASH

$ ./output.out | ./input.out
```

Le **pipe anonime**, come quelle usate su **shell**, 'uniscono' due processi aventi un antenato comune (oppure tra padre-figlio). Il collegamento è unidirezionale ed avviene utilizzando un buffer di dimensione finita. Per interagire con il buffer (la pipe) si usanono due **file descriptors**: uno per il lato in scrittura ed uno per il lato in lettura.
Visto che i processi figli ereditano i file descriptors, questo consente la comunicazione tra i processi (ma serve l'antenato comune).

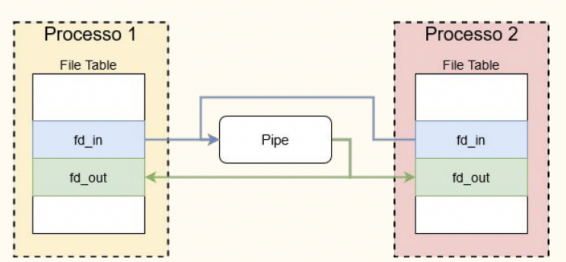

Figura 11: Caption

7.2.1 Creazione pipe

Per la creazione di un pipe si usa il comando **pipe**.

```
int pipe(int pipefd[2]); //fd[0] lettura, fd[1] scrittura

#include <stdio.h> //pipe.c
#include <unistd.h>
int main(){
    int fd[2], cnt = 0;
    while(pipe(fd) == 0){ //Create unnamed pipe using 2 file descriptors
        cnt++;
        printf("%d,%d,",fd[0],fd[1]);
            }
    printf("\n Opened %d pipes, then error\n",cnt);
    int op = open("/tmp/tmp.txt",O_CREAT|O_RDWR,S_IRUSR|S_IWUSR);
    printf("File opened with fd %d\n",op);
}
```

7.2.2 Lettura pipe

La **lettura** della pipe tramite il comando **read** restituisce valori differenti a seconda della situazione:

- In caso di **successo**, **read()** restituisce il numero di bytes effettivamente letti

- Se il lato di scrittura è stato **chiuso** (da ogni processo), con il buffer vuoto restituisce 0, altrimenti restituisce il numero di bytes letti.
- Se il buffer è **vuoto** ma il lato di scrittura è ancora aperto (in qualche processo) il processo si sospende fino alla disponibilità dei dati o alla chiusura
- Se si provano a leggere **più bytes** (num) di quelli disponibili, vengono recuperati solo quelli presenti

```
int read(int fd[0],char * data, int num)
```

```
#include <stdio.h> //readPipe.c
#include <unistd.h>

int main(void){
    int fd[2]; char buf[50];
    int esito = pipe(fd); //Create unnamed pipe
    if(esito == 0){
        write(fd[1],"writing",8); // Writes to pipe
        int r = read(fd[0],&buf,50); //Read from pipe
        printf("Last read %d. Received:    %s  \n",r,buf);
        // close(fd[1]); // hangs when commented
        r = read(fd[0],&buf,50); //Read from pipe
        printf("Last read %d. Received:    %s  \n",r,buf);
    }
}
```

7.2.3 Scrittura pipe

La **scrittura** della pipe tramite il comando **write** restituisce il numero di bytes scritti. Tuttavia, se il lato in lettura è stato chiuso viene inviato un segnale **SIGPIPE** allo scrittore (default handler quit). In caso di scrittura, se vengono scritti meno bytes di quelli che ci possono stare **(PIPE_BUF)** la scrittura è ”atomica” (tutto assieme), in caso contrario non c'è garanzia di atomicità e la scrittura sarà bloccata (in attesa che il buffer venga svuotato) o fallirà se il flag **O_NONBLOCK** viene usato.

```
int fcntl(int fd, F_SETFL, O_NONBLOCK);
```

```
#include <unistd.h> <stdio.h> <signal.h> <errno.h> <stdlib.h>
extern int errno; //writePipe.c

void handler(int signo){
    printf("SIGPIPE received\n"); perror("Error"); exit(errno);
}

int main(void){
    signal(SIGPIPE,handler);
    int fd[2]; char buf[50];
    int esito = pipe(fd); //Create unnamed pipe
    close(fd[0]); //Close read side
    printf("Attempting write\n");
    write(fd[1],"writing",8);
    printf("I've written something\n");
}
```

7.2.4 Esempio comunicazione unidirezionale

Un tipico esempio di **comunicazione unidirezionale** tra un processo scrittore P1 ed un processo lettore P2 è il seguente:

- P1 crea una **pipe()**
- P1 esegue un **fork()** e crea P2
- P1 chiude il lato lettura: **close(fd[0])**
- P2 chiude il lato scrittura: **close(fd[1])**

- P1 e P2 chiudono l'altro fd appena finiscono di comunicare

```c
#include <stdio.h> <unistd.h> <sys/wait.h> //uni.c
int main(){
    int fd[2]; char buf[50];
    pipe(fd); //Create unnamed pipe
    int p2 = !fork();
    if(p2){
        close(fd[1]);
        int r = read(fd[0],&buf,50); //Read from pipe
        close(fd[0]); printf("Buf:    %s  \n",buf);
    }else{
        close(fd[0]);
        write(fd[1],"writing",8); // Write to pipe
        close(fd[1]);
    }
    while(wait(NULL)>0);
}
```

7.2.5 Esempio comunicazione bidirezionale

Un tipico esempio di **comunicazione bidirezionale** tra un processo scrittore P1 ed un processo lettore P2 è il seguente:

- P1 **crea** due `pipe()`, pipe1 e pipe2
- P1 **esegue** un `fork()` e crea P2
- P1 **chiude** il lato lettura di pipe1 ed il lato scrittura di pipe2
- P2 **chiude** il lato scrittura di pipe1 ed il lato lettura di pipe2
- P1 e P2 **chiudono** gli altri fd appena finiscono di comunicare

```c
#include <stdio.h> <unistd.h> <sys/wait.h> #define READ 0 #define WRITE 1 //bi.c
int main(){
    int pipe1[2], pipe2[2]; char buf[50];
    pipe(pipe1); pipe(pipe2); //Create two unnamed pipe
    int p2 = !fork();
    if(p2){
        close(pipe1[WRITE]); close(pipe2[READ]);
        int r = read(pipe1[READ],&buf,50); //Read from pipe
        close(pipe1[READ]); printf("P2 received:    %s  \n",buf);
        write(pipe2[WRITE],"Msg from p2",12); // Writes to pipe
        close(pipe2[WRITE]);
    }else{
        close(pipe1[READ]); close(pipe2[1]);
        write(pipe1[WRITE],"Msg from p1",12); // Writes to pipe
        close(pipe1[WRITE]);
        int r = read(pipe2[READ],&buf,50); //Read from pipe
        close(pipe2[READ]); printf("P1 received:    %s  \n",buf);
    }
    while(wait(NULL)>0);
}
```

7.2.6 Gestire la comunicazione

Per **gestire** comunicazioni complesse c'è bisogno di definire un "protocollo".
Esempio:

- Messaggi di lunghezza fissa (magari inviata prima del messaggio)
- Marcatore di fine messaggio (per esempio con carattere NULL o newline)

Più in generale occorre definire la sequenza di messaggi attesi.

```c
#include <stdio.h> <unistd.h> #define READ 0 #define WRITE 1 //redirect.c
int main (int argc, char *argv[]) {
    int fd[2];
    pipe(fd); // Create an unnamed pipe
```

```
    if (fork() != 0) { // Parent, writer
        close(fd[READ]); // Close unused end
        dup2(fd[WRITE], 1); // Duplicate used end to stdout
        close(fd[WRITE]); // Close original used end
        execlp(argv[1], argv[1], NULL); // Execute writer program
        perror("connect"); // Should never execute
    } else { // Child, reader
        close(fd[WRITE]); // Close unused end
        dup2(fd[READ], 0); // Duplicate used end to stdin
        close(fd[READ]); // Close original used end
        execlp(argv[2], argv[2], NULL); // Execute reader program
        perror("connect"); // Should never execute
    }
}
```

7.3 pipe con nome/FIFO

Le **pipe** con nome, o **FIFO**, sono delle pipe che corrispondono a dei file speciali nel filesystem grazie ai quali i processi, senza vincoli di gerarchia, possono comunicare. Un processo può accedere ad una di queste pipe se ha i permessi sul file corrispondente ed è vincolato, ovviamente, dall'esistenza del file stesso. Le **FIFO** sono interpretate come dei file, perciò si possono usare le funzioni di scrittura/lettura dei file viste nelle scorse lezioni per interagirvi. Restano però delle pipe, con i loro vincoli e le loro capacità.
NB: Non sono dei file: `lseek` non funziona ed il loro contenuto è sempre vuoto! Normalmente, aprire una **FIFO** blocca il processo finché anche l'altro lato non è stato aperto. Le differenze tra **pipe anonime** e **FIFO** sono solo nella loro creazione e gestione.

7.3.1 Creazione FIFO

Per la **creazione** si utilizza il comando `mkfifo`

```
int mkfifo(const char *pathname, mode_t mode);
```

```
#include <sys/stat.h><sys/types.h><unistd.h><fcntl.h><stdio.h> //fifo.c
int main(void){
    char * fifoName = "/tmp/fifo1";
    mkfifo(fifoName,S_IRUSR|S_IWUSR); //Create pipe if it doesnt exist
    perror("Created?");
    if (fork() == 0){ //Child
        open(fifoName,O_RDONLY); //Open READ side of pipe...stuck!
        printf("Open read\n");
    }else{
        sleep(1);
        open(fifoName,O_WRONLY); //Open WRITE side of pipe
        printf("Open write\n");
    }
}
```

Esempio comunicazione: writer:

```
#include<sys/stat.h><sys/types.h><unistd.h><fcntl.h><stdio.h><string.h>
//fifoWriter.c
int main (int argc, char *argv[]) {
    int fd; char * fifoName = "/tmp/fifo1";
    char str1[80], * str2 = "I m a writer";
    mkfifo(fifoName,S_IRUSR|S_IWUSR); //Create pipe if it doesnt exist
    fd = open(fifoName, O_WRONLY); // Open FIFO for write only
    write(fd, str2, strlen(str2)+1); // write and close
    close(fd);
    fd = open(fifoName, O_RDONLY); // Open FIFO for Read only
    read(fd, str1, sizeof(str1)); // Read from FIFO
    printf("Reader is writing: %s\n", str1);
    close(fd);
}
```

Esempio comunicazione: reader:

```
#include<sys/stat.h><sys/types.h><unistd.h><fcntl.h><stdio.h><string.h>
//fifoReader.c
int main (int argc, char *argv[]) {
    int fd; char * fifoName = "/tmp/fifo1";
```

```
5     mkfifo(fifoName,S_IRUSR|S_IWUSR); //Create pipe if doesnt exist
6     char str1[80], * str2 = "I'm a reader";
7     fd = open(fifoName , O_RDONLY); // Open FIFO for read only
8     read(fd, str1, 80); // read from FIFO and close it
9     close(fd);
10    printf("Writer is writing: %s\n", str1);
11    fd = open(fifoName , O_WRONLY); // Open FIFO for write only
12    write(fd, str2, strlen(str2)+1); // Write and close
13    close(fd);
14 }
```

7.4 Pipe anonime vs FIFO

	pipe	FIFO
Rappresentazione	Buffer	File
Accesso	2 File descriptors	1 File descriptor
Persistenza	Eliminata alla terminazione di tutti i processi	Esiste finchè esiste il file
Vincoli accesso	Antenato comune	Permessi sul file
Creazione	pipe()	mkfifo()
Max bytes per atomicità	PIPE_BUF = 4096 on Linux, minimo 512 Bytes POSIX	

8 Lezione 8

8.1 Message queues

Una coda di messaggi, *message queue*, è una lista concatenata memorizzata all'interno del kernel ed identificata con un id (un intero univoco), chiamato **queue identifier**. Questo ID viene condiviso tra i processi interessati, e viene generato attraverso una chiave univoca.
Una coda deve essere innanzitutto generata in maniera analoga ad una **FIFO**, impostando dei permessi. Ad una coda esistente si possono aggiungere o recuperare messaggi tipicamente in modalità "autosincrona": la lettura attende la presenza di un messaggio, la scrittura attende che sia spazio disponibile. Questi comportamenti possono però essere configurati.
Quanto trattiamo di **message queue**, abbiamo due identificativi:

- **Key**: intero che identifica un insieme di risorse condivisibili nel kernel, come semafori, memoria condivisa e code. Questa chiave univoca deve essere nota a più processi, e viene usata per ottenere il **queue identifier**.
- **Queue identifier**: id univoco della coda, generato dal kernel ed associato ad una specifica *key*. Questo ID viene usato per interagire con la coda.

8.1.1 creazione coda

Per la creazione si usa il comando:

```
\textit{int msgget(key_t key, int msgflg)}
```

Restituisce l'identificativo di una coda basandosi sulla chiave **"key"** e sui **flags**:

- **IPC_CREAT**: crea una coda se non esiste già, altrimenti restituisce l'identificativo di quella già esistente;
- **IPC_EXCL**: (da usare con il precedente) fallisce se coda già esistente;
- **0xxx**: permessi per accedere alla coda, analogo a quello che si può usare nel file system. In alternativa si possono usare S_IRUSR etc.

```
    #include <sys/types.h> <sys/ipc.h> <sys/msg.h> //msgget.c
    key_t queueKey = 56; //Unique key
    int queueId = msgget(queueKey, 0777 | IPC_CREAT | IPC_EXCL);
```

Invece usiamo questo comando **ftok** per ottenere la chiave univoca:

```
    key_t ftok(const char *path, int id)
```

Restituisce una chiave basandosi sul **path** (una cartella o un file), esistente ed accessibile nel file-system, e sull'id numerico. La chiave dovrebbe essere univoca e sempre la stessa per ogni coppia $< path, id >$ in ogni istante sullo stesso sistema.
Un metodo d'uso, per evitare possibili conflitti, potrebbe essere generare un **path** (es. un file) temporaneo univoco, usarlo, eventualmente rimuoverlo, ed usare l'id per per rappresentare diverse "categorie" di code, a mo' di indice.

```
#include <sys/ipc.h> //ftok.c
key_t queue1Key = ftok("/tmp/unique", 1);
key_t queue2Key = ftok("/tmp/unique", 2); ...
```

Esempio creazione:

```
#include <sys/types.h> <sys/ipc.h> <sys/msg.h><stdio.h> //ipcCreation.c
void main(){
    remove("/tmp/unique"); //Remove file
    key_t queue1Key = ftok("/tmp/unique", 1); //Get unique key      fail
    creat("/tmp/unique", 0777); //Create file
    queue1Key = ftok("/tmp/unique", 1); //Get unique key      ok
    int queueId = msgget(queue1Key ,0777 | IPC_CREAT); //Create queue      ok
    queueId = msgget(queue1Key , 0777); //Get queue      ok
    msgctl(queue1Key,IPC_RMID,NULL); //Remove non existing queue      fail
    msgctl(queueId,IPC_RMID,NULL); //Remove queue      ok
    queueId = msgget(queue1Key , 0777); //Get non existing queue      fail
```

```
    queueId = msgget(queue1Key , 0777 | IPC_CREAT); //Create queue      ok
    queueId = msgget(queue1Key , 0777 | IPC_CREAT); //Get queue      ok
    queueId = msgget(queue1Key , 0777 | IPC_CREAT | IPC_EXCL); /* Create
    already existing queue -> fail */
}
```

```
#include <sys/ipc.h> <stdio.h> <sys/msg.h> //persistent.c

void main(){
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , 0777 | IPC_CREAT | IPC_EXCL);
    perror("Error:");
}
```

Se eseguiamo questo programma dopo aver eseguito il precedente **ipcCreation.c** verrà generato un errore dato che la coda esiste già ed abbiamo usato il flag **IPC_EXCL**!

8.1.2 Comunicazione

Ogni messaggio nella coda ha:

- Un tipo, categoria, etc... (intero ¿ 0)
- Una grandezza non negativa
- Un payload, un insieme di dati di lunghezza corretta

Esempio:

```
struct msg_buffer{
    long mtype;
    char mtext[100];
} message;
```

Al contrario delle **FIFO**, i messaggi in una coda possono essere recuperati anche sulla base del tipo e non solo del loro ordine "assoluto" di arrivo. Così come i files, le code sono delle strutture persistenti che continuano ad esistere, assieme ai messaggi in esse salvati, anche alla terminazione del processo che le ha create. L'eliminazione deve essere esplicita.
Il **payload** del messaggio non deve essere necessariamente un campo testuale: può essere una qualsiasi struttura dati. Infatti, un messaggio può anche essere senza **payload**.

```
typedef struct book{
    char title[10];
    char description[100];
    unsigned short chapters;
} Book;

struct msg_buffer{
    long mtype;
    Book mtext;
} message;

struct msg_empty{
    long mtype;
} message_empty;
```

8.1.3 Inviare/Ricevere messaggi

Per inviare i messaggi usiamo il comando:

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Aggiunge una copia del messaggio puntato da msgp, con un **payload** di dimensione **msgsz**[4], alla coda identificata da **msquid**. Il messaggio viene inserito immediatamente se c'è abbastanza spazio disponibile, altrimenti la chiamata si blocca fino a che abbastanza spazio diventa disponibile. Se **msgflg** è IPC_NOWAIT allora la chiamata fallisce in assenza di spazio.
Per ricevere messaggi invece usiamo:

[4] msgsz è la grandezza del payload del messaggio, non del messaggio intero (che contiene anche il tipo)! Per esempio, **sizeof(msgp.mtext)**

```
ssize_t msgrcv(int msqid,void *msgp,size_t msgsz,long msgtyp,int msgflg)
```

Rimuove un messaggio dalla coda **msqid** e lo salva nel buffer **msgp**.
msgsz specifica la lunghezza massima del payload del messaggio (per esempio **mtext** della struttura msgp). Se il payload ha una lunghezza maggiore e **msgflg** è **MSG_NOERROR** allora il payload viene troncato (viene persa la parte in eccesso), se **MSG_NOERROR** non è specificato allora il payload non viene eliminato e la chiamata fallisce. Se non sono presenti messaggi, la chiamata si blocca in loro attesa. Il flag **IPC_NOWAIT** fa fallire la syscall se non sono presenti messaggi.
A seconda di **msgtyp** viene recuperato il messaggio:

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

Esempio comunicazione 1:

```c
#include <sys/types.h><sys/ipc.h><sys/msg.h><string.h><stdio.h> //ipc.c

struct msg_buffer{
    long mtype;
    char mtext[100];
} msgp, msgp2; //Two different message buffers

int main(void){
    msgp.mtype = 20;
    strcpy(msgp.mtext,"This is a message");
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , 0777 | IPC_CREAT | IPC_EXCL);
    int esito = msgsnd(queueId , &msgp, sizeof(msgp.mtext),0);
    esito = msgrcv(queueId , &msgp2, sizeof(msgp2.mtext),20,0);
    printf("Received %s\n",msgp2.mtext);
}
```

Esempio comunicazione 2:

```c
#include <sys/types.h> <sys/ipc.h> <sys/msg.h> <string.h> //ipcBook.c

typedef struct book{
    char title[10];
    char description[200];
    short chapters;
} Book;

struct msg_buffer{
    long mtype;
    Book mtext;
} msgp_snd, msgp_rcv; //Two different message buffers

int main(void){
    msgp_snd.mtype = 20;
    strcpy(msgp_snd.mtext.title ,"Title");
    strcpy(msgp_snd.mtext.description ,"This is a description");
    msgp_snd.mtext.chapters = 17;
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , 0777 | IPC_CREAT);
    int esito = msgsnd(queueId , &msgp_snd, sizeof(msgp.mtext),0);
    esito = msgrcv(queueId , &msgp_rcv, sizeof(msgp_rcv.mtext),20,0);
    printf("Received: %s %s %d\n",msgp_rcv.mtext.title ,
    msgp_rcv.mtext.description , msgp_rcv.mtext.chapters);
}
```

Esempio comunicazione 2:

```c
#include <sys/types.h> <sys/ipc.h> <sys/msg.h> <string.h> //ipcType.c

struct msg_buffer{
    long mtype;
    char mtext[100];
} mspg_snd,msgp_rcv; //Two different message buffers

int main(int argc, char ** argv){
```

```
    int to_fetch = atoi(argv[0]); //Input to decide which msg to get
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , 0777 | IPC_CREAT);
    msgp_snd.mtype = 20;
    strcpy(msgp_snd.mtext,"A message of type 20");
    int esito = msgsnd(queueId , &msgp_snd, sizeof(msgp.mtext),0);
    msgp_snd.mtype = 10; //Re-use the same message
    strcpy(msgp_snd.mtext,"Another message of type 10");
    esito = msgsnd(queueId , &msgp_snd, sizeof(msgp.mtext),0);
    esito = msgrcv(queueId , &msgp_rcv, sizeof(msgp_rcv.mtext), to_fetch,0);
    printf("Received: %s %s %d\n",msgp_rcv.mtext.title , msgp_rcv.mtext.description,
    msgp_rcv.mtext.chapters);
}
```

8.1.4 Modificare la coda

Per modificare la coda usiamo:

```
int msgctl(int msqid, int cmd, struct msqid$_$ds *buf);
\end{\textbf{center}}

Modifica la coda identificata da msqid secondo i comandi cmd, riempiendo buf con
    informazioni sulla coda (ad esempio tempo di ultima scrittura, di ultima lettura,
numero messaggi nella coda, etc ).

Values for cmd are
\begin{itemize}
    \item \textbf{IPC$_$STAT}: recupera informazioni da kernel
    \item \textbf{IPC$_$SET}: imposta alcuni parametri a seconda di buf
    \item \textbf{IPC$_$RMID}: rimuove immediatamente la coda
    \item \textbf{IPC$_$INFO}: recupera informazioni generali sui limiti delle code nel
    sistema
    \item \textbf{MSG$_$INFO}: come IPC$_$INFO ma con informazioni differenti
    \item \g{MSG\_STAT}: come IPC$_$STAT ma con informazioni differenti
\end{itemize}

\textbf{msquid\_ds structure}
\begin{lstlisting}[language=c]
    struct msqid_ds {
        struct ipc_perm msg_perm; /* Ownership and permissions */
        time_t msg_stime; /* Time of last msgsnd(2) */
        time_t msg_rtime; /* Time of last msgrcv(2) */
        time_t msg_ctime; //Time of creation or last modification by msgctl
        unsigned long msg_cbytes; /* # of bytes in queue*/
        msgqnum_t msg_qnum; /* # of messages in queue */
        msglen_t msg_qbytes; /* Maximum # of bytes in queue */
        pid_t msg_lspid; /* PID of last msgsnd(2) */
        pid_t msg_lrpid; /* PID of last msgrcv(2) */
    };
```

ipc_perm structure

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

esempio

```
int main(void){
    struct msqid_ds mod;
    int esito = open("/tmp/unique",O_CREAT,0777);
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , IPC_CREAT | S_IRWXU );
    msgctl(queueId,IPC_RMID,NULL);
    queueId = msgget(queue1Key , IPC_CREAT | S_IRWXU );
    esito = msgctl(queueId,IPC_STAT,&mod); //Get info on queue
    printf("Current permission on queue: %d\n",mod.msg_perm.mode);
    mod.msg_perm.mode= 0000;
    esito = msgctl(queueId,IPC_SET,&mod); //Modify queue
    printf("Current permission on queue: %d\n\n",mod.msg_perm.mode);
```

```
}
```

8.2 ESEMPIO FINALE

```
#include<sys/types.h><sys/ipc.h><sys/msg.h><string.h><stdio.h><unistd.h><wait.h>
//ipc4.c
struct msg_buffer{
    long mtype;
    char mtext[100];
} msgpSND,msgpRCV;
void main(){
    struct msqid_ds mod;
    msgpSND.mtype = 1;
    strcpy(msgpSND.mtext,"This is a message from sender");
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , 0777 | IPC_CREAT);
    msgctl(queueId,IPC_RMID,NULL); //Remove queue if exists
    queueId = msgget(queue1Key , 0777 | IPC_CREAT); //Create queue
    msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),0); //Send msg
    msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),0); //Send msg
    msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),0); //Send msg
    msgctl(queueId,IPC_STAT,&mod); //Modify queue
    printf("Msg in queue: %ld\nCurrent max bytes in queue: %ld\n\n",
           mod.msg_qnum, mod.msg_qbytes);
    mod.msg_qbytes = 200; //Change buf to modify queue bytes
    msgctl(queueId,IPC_SET,&mod); //Apply modification
    printf("Msg in queue: %ld --> same number\nCurrent max bytes in
    queue: %ld\n\n",mod.msg_qnum, mod.msg_qbytes);
    if( fork() != 0 ){ //Parent keep on writing on the queue
        printf("[SND] Sending 4th message with a full queue...\n");
        msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),0); //Send msg
        printf("[SND] msg sent\n");
        printf("[SND] Sending 5th message with IPC_NOWAIT\n");
        if(msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),IPC_NOWAIT )
           == -1){ //Send msg
            perror("Queue is full --> Error");
        }
    } else { // Child keeps reading the queue every 3 seconds
        sleep(3); msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,0);
        printf("[Reader] Received msg 1 with msg '%s'\n",msgpRCV.mtext);
        sleep(3); msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,0);
        printf("[Reader] Received msg 2 with msg '%s'\n",msgpRCV.mtext);
        sleep(3); msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,0);
        printf("[Reader] Received msg 3 with msg '%s'\n",msgpRCV.mtext);
        sleep(3); msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,0);
        printf("[Reader] Received msg 4 with msg '%s'\n",msgpRCV.mtext);
        sleep(3);
        if(msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,IPC_NOWAIT)
           == -1){
            perror("Queue is empty --> Error");
            }else{
            printf("[Reader] Received msg 5 with msg '%s'\n",
                   msgpRCV.mtext);
        }
    }
    while(wait(NULL)>0);
}
```

9 Lezione 9

9.1 Threads

I **thread** sono singole sequenze di esecuzione all'interno di un processo, aventi alcune delle proprietà dei processi. I **threads** non sono indipendenti tra loro e condividono il codice, i dati e le risorse del sistema assegnate al processo di appartenenza. Come ogni singolo processo, i **threads** hanno alcuni elementi indipendenti, come lo *stack*, il *PC* ed i *registri del sistema*. La creazione di threads consente un parallelismo delle operazioni in maniera rapida e semplificata. Context switch tra **threads** è rapido, così come la loro creazione e terminazione. Inoltre, la comunicazione tra **threads** è molto veloce.

Per la compilazione è necessario aggiungere il flag `-pthread`, ad **esempio**:
gcc -o program main.c -pthread

9.1.1 Creazione

In C i thread corrispondono a delle funzioni eseguite in parallelo al codice principale. Ogni thread è identificato da un ID e può essere gestito come un processo figlio, con funzioni che attendono la sua terminazione.

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

Esempio creazione:

```
#include <stdio.h> <pthread.h> <unistd.h> //threadCreate.c

void * my_fun(void * param){
    printf("This is a thread that received %d\n", *(int *)param);
    return (void *)3;
}
void main(){
    pthread_t t_id;
    int arg=10;
    pthread_create(&t_id, NULL, my_fun, (void *)&arg);
    printf("Executed thread with id %ld\n",t_id);
    sleep(3);
}
```

Nel kernel...[5]

```
$ ./threadList.out & ps -eLf

#include <pthread.h> //threadList.c

void * my_fun(void * param){
    while(1);
}
void main(){
    pthread_t t_id;
    pthread_create(&t_id, NULL, my_fun, NULL);
    while(1);
}
```

9.1.2 Terminazione

Un nuovo thread termina in uno dei seguenti modi:

- Chiamando la funzione `noreturn void pthread_exit(void * retval);` specificando un puntatore di ritorno.
- Ritorna dalla funziona associata al thread specificando un valore di ritorno.

[5] Light-Weight Process: LWP: identificativo Thread / NLWP: numero di Threads nel processo

- Viene cancellato con **int pthread_cancel(pthread_t thread)**.
- Qualche thread chiama **exit()**, o il thread che esegue **main()** ritorna dallo stesso, terminando così tutti i threads.

9.1.3 Cancellazione thread

```
int pthread_cancel(pthread_t thread);
```

Invia una **richiesta** di cancellazione al thread specificato, il quale reagirà (come e quando) a seconda di due suoi attributi: **state** e **type**.

- **State** può essere enabled (default) o disabled: se disabled la richiesta rimarrà in attesa fino a che state diventa enabled, se enabled la cancellazione avverrà a seconda di type.
- **Type** può essere deferred (default) o asynchronous: il primo attende la chiamata di un cancellation point, il secondo termina in qualsiasi momento. I cancellation points sono funzioni definite nella libreria pthread.h (lista).

State e **type** di un thread possono essere modificati solo dal thread stesso con le seguenti funzioni:

```
int pthread_setcancelstate(int state, int *oldstate);
/*con state = PTHREAD_CANCEL_DISABLE o PTHREAD_CANCEL_ENABLE*/


int pthread_setcanceltype(int type, int *oldtype);
/*Con type = PTHREAD_CANCEL_DEFERRED o PTHREAD_CANCEL_ASYNCHRONOUS*/
```

Esempio cancellazione 1:

```
#include <stdio.h> <pthread.h> <unistd.h> //thCancel.c

int i = 1;
void * my_fun(void * param){
    if(i--)
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,NULL); //Change mode
    printf("Thread %ld started\n",*(pthread_t *)param); sleep(3);
    printf("Thread %ld finished\n",*(pthread_t *)param);
}

int main(void){
    pthread_t t_id1, t_id2;
    pthread_create(&t_id1, NULL, my_fun, (void *)&t_id1); sleep(1); //Create
    pthread_cancel(t_id1); //Cancel
    printf("Sent cancellation request for thread %ld\n",t_id1);
    pthread_create(&t_id2, NULL, my_fun, (void *)&t_id2); sleep(1); //Create
    pthread_cancel(t_id2); //Cancel
    printf("Sent cancellation request for thread %ld\n",t_id2);
    sleep(5); printf("Terminating program\n");
}
```

Esempio cancellazione 2:

```
#include <stdio.h> <pthread.h> <unistd.h> <string.h> //thCancel2.c

int tmp = 0;
    void * my_fun(void * param){
    pthread_setcanceltype(*(int *)param,NULL); // Change type
    for (long unsigned i = 0; i < (0x9FFF0000); i++); //just wait
    tmp++;
    open("/tmp/tmp",O_RDONLY); //Cancellation point!
}

int main(int argc, char ** argv){ //call program with 'async' or 'defer'
    pthread_t t_id1; int arg;
    if(!strcmp(argv[1],"async")) arg = PTHREAD_CANCEL_ASYNCHRONOUS;
    else if(!strcmp(argv[1],"defer")) arg = PTHREAD_CANCEL_DEFERRED;
    pthread_create(&t_id1, NULL, my_fun, (void *)&arg); sleep(1); //Create
    pthread_cancel(t_id1); sleep(5); //Cancel
    printf("Tmp %d\n",tmp);
}
```

9.1.4 Aspettare un thread

Un processo (thread) che avvia un nuovo thread può aspettare la sua terminazione mediante la funzione:

```
int pthread_join(pthread_t thread, void ** retval);
```

Questa funzione ritorna quando il thread identificato da **thread** termina, o subito se il thread è già terminato. Se il valore di ritorno del thread non è nullo (parametro di `pthread_exit` o di `return`), esso viene salvato nella variabile puntata da `retval`. Se il thread era stato cancellato, `retval` è riempito con `PTHREAD_CANCELED`. Solo se il thread è joinable può essere aspettato! Un thread può essere aspettato da al massimo un thread!
Esempio join 1:

```
#include <stdio.h> <pthread.h> <unistd.h> //thJoin.c

void * my_fun(void * param){
    printf("Thread %ld started\n",*(pthread_t *)param); sleep(3);
    char * str = "Returned string";
    pthread_exit((void *)str); //or    return    (void *) str;
}
void main(){
    pthread_t t_id;
    void * retFromThread; //This must be a pointer to void!
    pthread_create(&t_id, NULL, my_fun, (void *)&t_id); //Create
    pthread_join(t_id,&retFromThread); // wait thread
    // We must cast the returned value!
    printf("Thread %ld returned '%s'\n",t_id,(char *)retFromThread);
}
```

Esempio join 2:

```
#include <stdio.h> <pthread.h> <unistd.h> //threadJoin.c

void * my_fun(void *param){
    printf("This is a thread that received %d\n", *(int *)param);
    return (void *)3;
}
void main(){
    pthread_t t_id;
    int arg=10, retval;
    pthread_create(&t_id, NULL, my_fun, (void *)&arg);
    printf("Executed thread with id %ld\n",t_id);
    sleep(3);
    pthread_join(t_id, (void **)&retval); //A pointer to a void pointer
    printf("retval=%d\n", retval);
}
```

9.1.5 Attributi di un thread

Ogni thread viene creato con degli attributi specificati nella struttura `pthread_attr_t`. Questa struttura è usata solo alla creazione di un thread, ed è poi indipendente dallo stesso (se cambia, gli attributi del thread non cambiano). La struttura va inizializzata e, a fine utilizzo, distrutta. Una volta inizializzati, i vari attributi della struct possono, e devono, essere modificati singolarmente delle funzioni:

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_setxxxx(pthread_attr_t *attr, params);
int pthread_attr_getxxxx(const pthread_attr_t *attr, params);
```

Lista attributi:

- detachstate(pthread_attr_t *attr, int detachstate)
  - PTHREAD_CREATE_DETACHED → non può essere aspettato
  - PTHREAD_CREATE_JOINABLE → default, può essere aspettato
  - Può essere cambiato durante l'esecuzione con int pthread_detach(pthread_t thread);
- ...sigmask_np(pthread_attr_t *attr,const sigset_t *sigmask);

- ...affinity_np(...)
- ...setguardsize(...)
- ...inheritsched(...)
- ...schedparam(...)
- ...schedpolicy(...)

I threads vengono creati di default nello stato joinable, il che consente ad un altro thread di attendere la loro terminazione attraverso il comando `pthread_join`. I thread joinable rilasciano le proprie risorse non alla terminazione ma quando un thread fa il join con loro (salvando lo stato di uscita) (similmente ai sottoprocessi), oppure alla terminazione del processo. Contrariamente, i thread in stato detached liberano le loro risorse immediatamente una volta terminati, ma non consentono ad altri processi di fare il "join". NB: un thread detached non può diventare joinable durante la sua esecuzione, mentre il contrario è possibile.
Esempio attributi:

```c
#include <stdio.h> <pthread.h> <unistd.h> //threadAttr.c

void *my_fun(void *param){
    printf("This is a thread that received %d\n", *(int *)param);return (void*)3;
}

void main(){
    pthread_t t_id; pthread_attr_t attr;
    int arg=10, detachSTate;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED); //Set detached
    pthread_attr_getdetachstate(&attr,&detachSTate); //Get detach state
    if(detachSTate == PTHREAD_CREATE_DETACHED) printf("Detached\n");
    pthread_create(&t_id, &attr, my_fun, (void *)&arg);
    printf("Executed thread with id %ld\n",t_id);
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE); //Ineffective
    sleep(3); pthread_attr_destroy(&attr);
    int esito = pthread_join(t_id, (void **)&detachSTate);
    printf("Esito '%d' is different 0\n", esito);
}
```

9.2 Mutex

Quando eseguiamo un programma con più thread essi condividono alcune risorse, tra le quali le variabili globali. Se entrambi i thread accedono ad una sezione di codice condivisa ed hanno la necessità di accedervi in maniera esclusiva allora dobbiamo instaurare una sincronizzazione. I risultati, altrimenti, potrebbero essere inaspettati.
Esempio:

```c
#include <pthread.h> <stdlib.h> <unistd.h><stdio.h> //syncProblem.c

pthread_t tid[2];
int counter = 0;

void *thr1(void *arg){
    counter = 1;
    printf("Thread 1 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0x00FF0000); i++); //wait some cycles
    counter += 1;
    printf("Thread 1 expects 2 and has: %d\n", counter);
}

void *thr2(void *arg){
    counter = 10;
    printf("Thread 2 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0xFFF0000); i++); //wait some cycles
    counter += 1;
    printf("Thread 2 expects 11 and has: %d\n", counter);
}

void main(void){
    pthread_create(&(tid[0]), NULL, thr1,NULL);
```

```
    pthread_create(&(tid[1]), NULL, thr2,NULL);
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
}
```

I **mutex** sono dei semafori imposti ai thread. Essi possono proteggere una determinata sezione di codice, consentendo ad un thread di accedervi in maniera esclusiva fino allo sblocco del semaforo. Ogni thread che vorrà accedere alla stessa sezione di codice dovrà aspettare che il semaforo sia sbloccato, andando in sleep fino alla sua prossima schedulazione. I **mutex** vanno inizializzati e poi assegnati ad una determinata sezione di codice.

9.2.1 Creare e distruggere un mutex

Per la creazione e distruzione di un mutex usiamo i comandi:

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *
    restrict attr)

int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

```
#include <pthread.h> //createMutex.c
pthread_mutex_t lock;

int main(void){
    pthread_mutex_init(&lock, NULL); // Create mutex with default attrs
    pthread_mutex_destroy(&lock); // Destroy mutex (it can be re-init)
}
```

9.2.2 Bloccare e sbloccare un mutex

Per bloccare e sbloccare un mutex usiamo:

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Esempio:

```
#include <pthread.h> <stdlib.h> <unistd.h><stdio.h> //mutex.c

pthread_mutex_t lock;
pthread_t tid[2];
int counter = 0;

void* thr1(void* arg){
    pthread_mutex_lock(&lock);
    counter = 1;
    printf("Thread 1 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0x00FF0000); i++);
    counter += 1;
    pthread_mutex_unlock(&lock);
    printf("Thread 1 expects 2 and has: %d\n", counter);
}

void* thr2(void* arg){
    pthread_mutex_lock(&lock);
    counter = 10;
    printf("Thread 2 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0xFFF0000); i++);
    counter += 1;
    pthread_mutex_unlock(&lock);
    printf("Thread 2 expects 11 and has: %d\n", counter);
}

int main(void){
    pthread_mutex_init(&lock, NULL);
    pthread_create(&(tid[0]), NULL, thr1,NULL);
    pthread_create(&(tid[1]), NULL, thr2,NULL);
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
}
```

9.2.3 Tipi di mutex

i tipi di mutex sono:

- PTHREAD_MUTEX_NORMAL: no deadlock detection
  - Ribloccare quando bloccato → deadlock
  - Sbloccare quando bloccato da altri → undefined
  - Sbloccare quando sbloccato → undefined
- PTHREAD_MUTEX_ERRORCHECK: error checking
  - Ribloccare quando bloccato → error
  - Sbloccare quando bloccato da altri → error
  - Sbloccare quando sbloccato → error
- PTHREAD_MUTEX_RECURSIVE: multiple locks
  - Ribloccare quando bloccato → increase lock count → richiede stesso numero di sbloccaggi
  - Sbloccare quando bloccato da altri → error
  - Sbloccare quando sbloccato → error
- PTHREAD_MUTEX_DEFAULT:
  - Ribloccare quando bloccato → undefined
  - Sbloccare quando bloccato da altri → undefined
  - Sbloccare quando sbloccato → undefined

Esempio:

```c
#include <pthread.h> <stdlib.h> <unistd.h><stdio.h> //recursive.c

pthread_mutex_t lock;
pthread_t tid[2];
int counter = 0;

void* thr1(void* arg){
    pthread_mutex_lock(&lock);
    pthread_mutex_lock(&lock);
    counter = 1;
    printf("Thread 1 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0x00FF0000); i++);
    counter += 1;
    pthread_mutex_unlock(&lock);
    printf("Thread 1 expects 2 and has: %d\n", counter);
    pthread_mutex_unlock(&lock);
}

void* thr2(void* arg){
    pthread_mutex_lock(&lock); pthread_mutex_lock(&lock);
    counter = 10;
    printf("Thread 2 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0xFFF0000); i++);
    counter += 1;
    pthread_mutex_unlock(&lock); pthread_mutex_unlock(&lock);
    printf("Thread 2 expects 11 and has: %d\n", counter);
}

void main(){
    pthread_mutexattr_t attr;
    pthread_mutexattr_settype(&attr,PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&lock, &attr);
    pthread_create(&(tid[0]), NULL, thr1,NULL);
    pthread_create(&(tid[1]), NULL, thr2,NULL);
    pthread_join(tid[0], NULL); pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
}
```