# Lezione 6 - 19/4/21 (Segnali, Groups, Group system calls)

## Segnali

### Gestione dei segnali

I segnali detti anche "*software interrupts*" perché sono a tutti gli effetti delle interruzioni del normale flusso del processo generato 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 di default
- Ignorandolo (non sempre possibile) → programma prosegue normalmente
- Eseguendo un handler personalizzato → programma si interrompe

### Default handler

Ogni segnale ha il suo handler di default che può:

- Ignorare il segnale
- Terminare il processo
- Continuare l'esecuzione (se il processo era stato stoppato)
- Stoppare il processo

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

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

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

## 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 uno steso handler per segnali differenti.

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

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

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

## Alcuni segnali

## 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 with dump |
| SIGTERM | (software termination) | quit |
| SIGUSR1/2 | (user signal 1/2) | quit |
| SIGSTOP | (stopped) | quit |
| SIGTSTP | (terminal stop, CTRL + Z) | quit |

## Inviare segnali: kill()

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

Invia un segnale ad uno o più processi a seconda di "*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 è stato inviato e -1 nel caso di errore.

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

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

```
int main(){
  signal(SIGALRM, myHandler);
  int child = fork();
  if(!child){
    while(1){

    }
  }
  printf("[%d]: sending alarm to %d in 1 s\n",getpid(),child);
  sleep(1)
  kill(child,SIGALRM);
  printf("[%d]: sending SIGTERM to %d in 1s\n",getpid(),child);
  sleep(1);
  kill(child, SIGTERM);
  while(wait(NULL)>0);
}
```

## Mettere in pausa: pause()

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

void myHandler(int sigNum){
  printf("Continue!\n");
}
int main(int argc, char **argv){
  signal(SIGCONT,myHandler);
  signal(SIGUSR1,myHandler);
  pause();
}
```

## Bloccare segnali

Oltre la lista dei **pending signal** esiste la lista dei **blocked signal** che sono i 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 viene tramutato in **ignore**.

L'insieme dei segnali bloccato è detto **signal mask,** una maschera dei segnali che è modificabile

attraverso la system call `sigprocmask()`**.**

## 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
```

## Bloccare i segnali: sigprocmask()

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

A seconda del valore id **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
- **how = SIG_SETMASK: set** diventa la maschera

Se **oldset** non è nullo, in esso verrà salvata la vecchia maschera. **oldset** viene riempito anche se **set** è nullo.

```
//sigSet.c
#include <signal.h>

int main(){
  sigset_t mod,old;
  sigfillset(&mod); //Add al 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);
}
```

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

```
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);
  sigaddset(&mod,SIGUSR1);
  while(1) if(i==1) sigprocmask(SIG_BLOCK,&mod,&old);
}
/*
./sigprocmask.out
kill -10 <PID> # ok
kill -10 <PID> # blocked
*/
```

## sigpending()

```
#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 it’s blocked...
    sigpending(&pen);
    if(sigismember(&pen,SIGUSR1))
        printf("SIGUSR1 pending\n");
    sigprocmask(SIG_UNBLOCK,&mod,NULL);
    while(1);
}
```

### Sigaction()

```
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;
  int sa_flags;
};
```

## Esempio

```
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>
void handler(int signo){
  printf("signal recived\n");
}
int main(){
  struct sigaction sa;
  sa.sa_handler=handler;
  sigemptyset(&sa.sa_mask);
  sigaction(SIGUSR1,&sa,NULL);
  kill(getpid(),SIGUSR1);
}
```

## Esempio: blocking signal

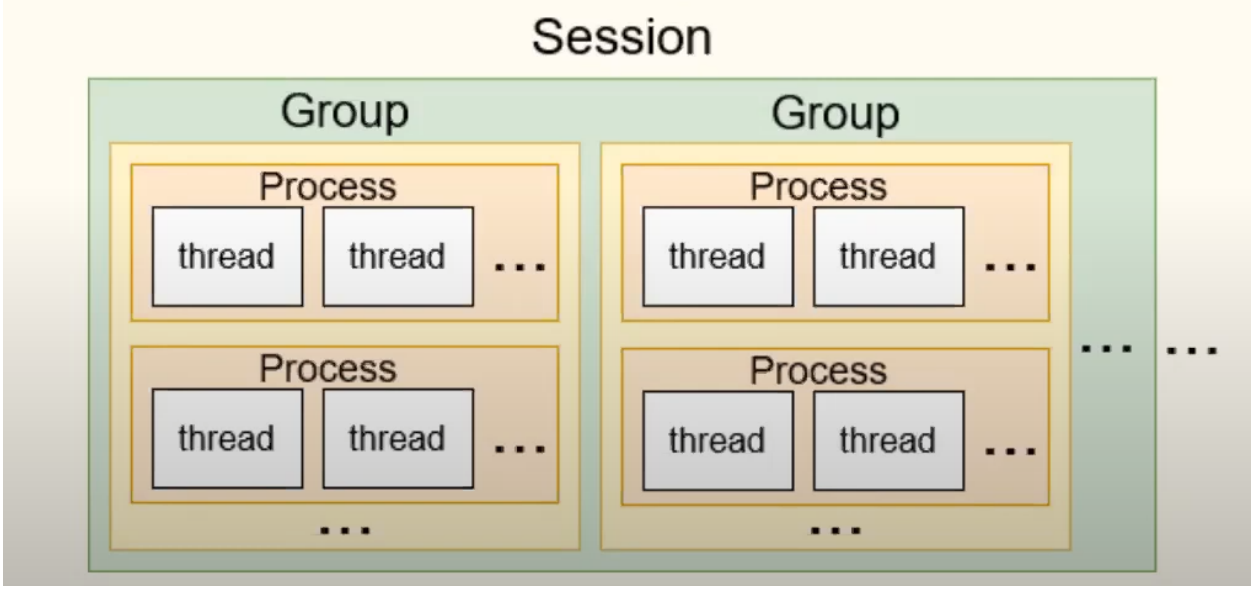
```
//sigaction3.c
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

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

int main(){
  printf("Process id: %d\n",gepid());
  struct sigaction sa;
  sa.sa_handler=handler;
  sigemptyset(&sa.sa_mask);
  sigaddset(&sa.sa_mask,SIGUSR2);
  sigaction(SIGUSR1,&sa,NULL);
  while(1);
}
/*
kill -10 <pid> ; sleep 1 && kill -12 <pid>
*/
```

### Esempio: sa_sigaction

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

void handler(int signo, siginfo_t * info, void * empty){
    printf("Signal recived from %d\n",(info)->si_pid);
}

int main(){
    struct sigaction sa;
    sa.sa_sigaction=handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags |= SA_SIGINFO;  // |= utilizzato per aggiungere
    sa.sa_flags |= SA_RESETHAND;

    sigaction(SIGUSR1, &sa, NULL);
    while(1);
}
```

# Process group

## Gestione dei processi in Unix




Vanataggi:

- Migliore gestione dei segnali
- Migliore gestione comunicazione tra processi

Un processo può:

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

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

**Sessione:** generalmente associata ad un terminale.

**Gruppo:** Se eseguo singoli comandi su bash sono di gruppi diversi se invece faccio con il piping allora sono dello stesso gruppo, i figli di un processo sono dello stesso gruppo del padre

I **gruppi** non possono essere creati cambiando il GID con uno non già utilizzato ma posso solo cambiare tra i gruppi esistenti. Ciò perché ho bisogno di un Processo Group Leader.

**N.B.**

Se PID==GID allora si parla di processo padre.

## Group system calls

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

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

Esempio:

```
//waitgroup.c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
void main(){
    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);
}
```

```
setpgid(pid_t pid, pid_t pgid);
```

il primo parametro **pid** può valere **0** se devo cambiare il process group id del programma corrente, o può equivalere a un **pid** se ho i permessi di cambiarlo ad altri, e il secondo parametro è il nuovo process group id.