## Segnali

Sincronizzazione mediante segnali

Vi sono spesso eventi importanti da notificare ai processi:

- tasti speciali sul terminale (es. ^C)
- eccezioni hardware (es. divisione per 0)
- primitiva/comando kill (es. `kill -9 1221`)
- condizioni software (es. scadenza di un timer)

L'arrivo di tali **eventi asincroni** può richiedere un'immediata gestione da parte del processo (analogamente alle interruzioni hardware)

⇒ **i segnali sono anche detti software interrupts**



## Segnali

Quali sono le possibilità di gestione di un segnale per un processo ?

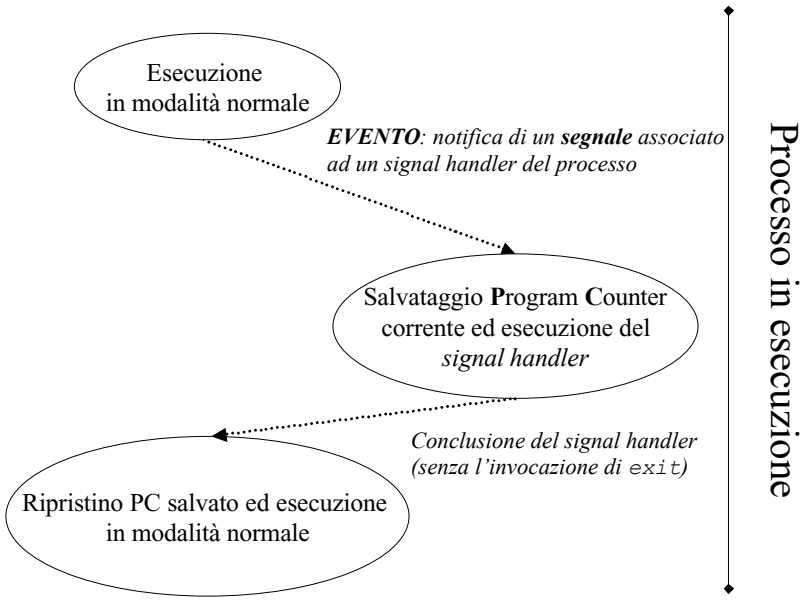
1. può decidere di **ignorarlo** (possibile solo per alcuni tipi di segnale)
2. può contare su un'azione di **default**
3. può far eseguire un'**azione** specificata dall'utente (gestore del segnale - signal handler )

Per tutta la durata dell'esecuzione del gestore del segnale, l'esecuzione del programma interrotto rimane bloccata:

⇒ al processo UNIX è associato un solo flusso di controllo (un solo Program Counter)



## Segnali






## Segnali

Elenco dei segnali Linux (/usr/include/asm/signal.h )

```
#define  SIGHUP      1          /* Hangup (POSIX).  */
#define  SIGINT      2          /* Interrupt (ANSI).  */
#define  SIGQUIT     3          /* Quit (POSIX).  */
#define  SIGILL      4          /* Illegal instruction (ANSI).  */
#define  SIGTRAP     5          /* Trace trap (POSIX).  */
#define  SIGABRT     6          /* Abort (ANSI).  */
#define  SIGIOT      6          /* IOT trap (4.2 BSD).  */
#define  SIGBUS      7          /* BUS error (4.2 BSD).  */
#define  SIGFPE      8          /* Floating-point exception (ANSI).  */
#define  SIGKILL     9          /* Kill, unblockable (POSIX).  */
#define  SIGUSR1     10         /* User-defined signal 1 (POSIX).  */
#define  SIGSEGV     11         /* Segmentation violation (ANSI).  */
#define  SIGUSR2     12         /* User-defined signal 2 (POSIX).  */
#define  SIGPIPE     13         /* Broken pipe (POSIX).  */
#define  SIGALRM     14         /* Alarm clock (POSIX).  */
#define  SIGTERM     15         /* Termination (ANSI).  */
#define  SIGSTKFLT   16         /* Stack fault.  */
#define  SIGCLD      SIGCHLD    /* Same as SIGCHLD (System V).  */
```

Per Solaris 5.x cfr. /usr/include/sys/signal.h : alcuni segnali hanno valori diversi

## Segnali

Elenco dei segnali (cont.)

```
#define   SIGCHLD       17      /* Child status has changed (POSIX).  */
#define   SIGCONT       18      /* Continue (POSIX).  */
#define   SIGSTOP       19      /* Stop, unblockable (POSIX).  */
#define   SIGTSTP       20      /* Keyboard stop (POSIX).  */
#define   SIGTTIN       21      /* Background read from tty (POSIX).  */
#define   SIGTTOU       22      /* Background write to tty (POSIX).  */
#define   SIGURG        23      /* Urgent condition on socket (4.2 BSD).  */
#define   SIGXCPU       24      /* CPU limit exceeded (4.2 BSD).  */
#define   SIGXFSZ       25      /* File size limit exceeded (4.2 BSD).  */
#define   SIGVTALRM     26      /* Virtual alarm clock (4.2 BSD).  */
#define   SIGPROF       27      /* Profiling alarm clock (4.2 BSD).  */
#define   SIGWINCH      28      /* Window size change (4.3 BSD, Sun).  */
#define   SIGPOLL       SIGIO   /* Pollable event occurred (System V).  */
#define   SIGIO         29      /* I/O now possible (4.2 BSD).  */
#define   SIGPWR        30      /* Power failure restart (System V).  */
#define SIGSYS          31      /* Bad system call.  */
#define SIGUNUSED       31
```



## Segnali

L'interfaccia **signal** per la gestione dei segnali

```
#include <signal.h>
void (*signal( int signo, void (*func)(int))) (int) ;
```

Si specifica quale segnale (`signo`) e come deve essere trattato (`func`)

Valori ammessi per il parametro `func` :

1. **SIG_IGN** (ignora il segnale - solo per alcuni segnali)
2. **SIG_DFL** (azione di default per quel segnale)
3. indirizzo (ovvero in C il nome) di una funzione del programma (signal handler o gestore del segnale)



## Segnali

`signal` fornisce un'interfaccia semplificata presente in ANSI C

- in SVR4 si ottengono segnali inaffidabili :
  - alla notifica del segnale, dopo l'esecuzione del gestore, l'azione torna ad essere quella di default (il gestore deve quindi installare di nuovo se stesso richiamando la `signal`)
  - segnali inviati possono andare perduti

  ⇒ deve essere usata la `sigaction` (più avanti)

- in BSD4.3+ e in LINUX si ottengono invece segnali affidabili perchè la `signal` è implementata attraverso la `sigaction` (più avanti)



## Segnali

Uso della `signal`

```
...
/* Il processo richiede che una eventuale notifica
di un segnale SIGINT venga gestita dalla propria
funzione sigint_handler */

signal(SIGINT, sigint_handler);
...

void sigint_handler(int signo)
{
/* In SVR4 e' necessario installare di nuovo il gestore
per la successiva notifica di SIGINT  */
signal(SIGINT, sigint_handler);
...
}
```

Un gestore di segnale può decidere di far continuare il processo oppure di farlo terminare (exit)

## Segnali

Esempio di gestione di segnale con `signal`

*Per terminare l'esecuzione di questo processo: ^\ oppure `kill -9 pidprocesso` dallo shell*

```
#include <signal.h>

void catchint(int);

main()
{
int i;

/* La notifica di un segnale SIGINT deve avviare il gestore
catchint: si dice comunemente che il processo "intercetta" o
"aggancia" il segnale */
signal(SIGINT, catchint);

while(1)  /* Ciclo senza fine */
       for(i=0 ; i< 10000 ; i++)
              printf("i vale %d\n", i);
}

void catchint(int signo)
{
signal(SIGINT,catchint);printf("catchint: signo=%d\n", signo);
/* Non viene invocata la exit: ritorno al segnalato */
}
```



## Segnali

Altre primitive per i segnali

```
#include <sys/types.h>
#include <signal.h>

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

Invia il segnale `sig` al processo `pid`

- il processo mittente e quello destinatario del segnale devono appartenere allo stesso utente
- solo root può inviare segnali a processi di altri utenti

La primitiva `kill` è asincrona (non bloccante):

⇒ il processo mittente prosegue immediatamente mentre al destinatario viene notificato il segnale



## Segnali

Invio temporizzato di segnali

```
#include <unistd.h>

unsigned int alarm(unsigned int  nseconds);
```

Dopo *`nseconds`* secondi il processo chiamante riceve un segnale `SIGALRM` inviato dal S.O.

Attesa di un segnale

```
#include <unistd.h>

int pause(void);
```

Il processo chiamante rimane bloccato fino a quando non viene eseguito un gestore di segnale per l' arrivo di un qualunque segnale
La `pause` ritorna sempre con -1 e con errno che vale EINTR



## Segnali

Sospensione temporizzata

```
unsigned int sleep(unsigned int nseconds);
```

Sospende il processo chiamante per il tempo specificato

L'arrivo di un segnale mentre il processo è sospeso interrompe l' attesa:

- in questo caso `sleep` ritorna con -1 e con errno che vale EINTR

E' preferibile l'utilizzo della `nanosleep` che in caso di interruzione restituisce l'intervallo di attesa residuo

Alcune implementazioni della `sleep` usano `SIGALRM` per cui non è possibile utilizzare contemporaneamente `sleep` e `alarm`

## Segnali
Esempio di interazione tra processi mediante segnali inaffidabili

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

void catcher(int signo)
{
  static int ntimes = 0;
  printf("Processo %d ricevuto #%d volte\n", getpid(), ++ntimes);
}

int main()
{
int pid, ppid;

signal(SIGUSR1, catcher); /* il figlio la ereditera' */

if ((pid=fork()) < 0)
  {
  perror("fork error");
  exit(1);
  }
else
  if (pid == 0)
    {
       /* Processo figlio - continua … */
```



## Segnali

```
    {/* … continua processo figlio */
    ppid = getppid();
    printf("figlio: mio padre e' %d\n", ppid);
    for (;;)
        {

        sleep(1);
        kill(ppid, SIGUSR1);
        pause();
        }
    }
  else
    {
      /* Processo padre */
      printf("padre: mio figlio e' %d\n", pid);
      for (;;)
        {
          pause();
          sleep(1);
          kill(pid, SIGUSR1);
        }
    }
}
```



## Segnali
Problemi con i segnali inaffidabili (unreliable)

1) Reset del gestore dopo una notifica del segnale (SVR4)

```
...
signal(SIGINT, sig_int);
...

void sig_int()
{
      <= = = ====================
signal(SIGINT, sig_int);
...

}
```

⇒Esiste una finestra temporale in cui la notifica di un secondo segnale fa terminare il processo (in questo esempio):
prima della nuova signal è attiva la gestione di default!)



## Segnali
Problemi con i segnali inaffidabili (unreliable)

2) Attesa di un segnale

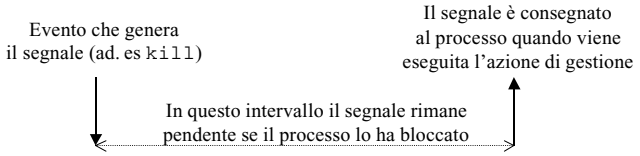
```
...
int segnale_arrivato=0;
...
signal(SIGINT, sig_int);
...
if (! segnale_arrivato)
              <= = = ========================
      pause(); /*Il processo attenderà un ulteriore
                  segnale */
...

void sig_int()
{
signal(SIGINT, sig_int);
segnale_arrivato= 1;
}
```

⇒ Esiste una finestra temporale in cui la notifica del segnale può provocare un deadlock

## Segnali

La gestione affidabile dei segnali

Un processo può bloccare temporaneamente la consegna di un segnale (non il `SIGKILL`) che rimane pendente




Il segnale rimane pendente fino a che:

- il processo lo sblocca

oppure

- il processo sceglie di ignorare quel segnale

⇒ un processo può cambiare l'azione da eseguire prima della consegna del segnale



## Segnali

sigset_t è un nuovo tipo di dato che rappresenta un insieme di segnali (signal set)

Funzioni di utilità per la manipolazione dei signal set

```
#include <signal.h>

int sigemptyset (sigset_t *set);
```
azzera / rende vuoto un sigset

```
int sigfillset (sigset_t *set);
```
mette tutti i segnali nel sigset

```
int sigaddset (sigset_t *set, int signo);
```
aggiunge un segnale al sigset

```
int sigdelset (sigset_t *set, int signo);
```
rimuove un segnale dal sigset

```
int sigismember (sigset_t *set, int signo);
```
valuta se quel segnale è presente nel sigset



## Segnali

Un processo può esaminare e/o modificare la propria signal mask che è l'insieme dei segnali che sta bloccando

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

dove *how* può valere:

- **SIG_BLOCK** ⇒ la nuova signal mask diventa l'OR binario di quella corrente con quella specificata da `set`
- **SIG_UNBLOCK** ⇒ i segnali indicati da `set` sono rimossi dalla signal mask (AND NOT mask)
- **SIG_SETMASK** ⇒ la nuova signal mask diventa quella specificata da `set`

se `oset` è diverso da `NULL` la signal mask precedente viene restituita in `oset`

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

restituisce il sottoinsieme dei segnali bloccati che sono attualmente pendenti



## Segnali

La primitiva fondamentale per la gestione dei segnali affidabili

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *act,
const struct sigaction *oact);
```

Permette di esaminare e/o modificare l'azione associata ad un segnale

- `signo` identifica il segnale del quale si vuole esaminare e/o modificare l'azione
- se *act* non è NULL si modifica l'azione
- se *oact* non è NULL viene restituita la precedente azione

## Segnali

Definizione della struttura sigaction

```
struct sigaction {
void  (*sa_handler)(); /* indirizzo del gestore o SIG_IGN o
                          SIG_DFL */
void  (*sa_sigaction)(int, siginfo_t *, void *);
/*  indirizzo del gestore che riceve informazioni addizionali
sul segnale ricevuto (utilizzato al posto di sa_handler se
sa_flags contiene SA_SIGINFO ) */

sigset_t    sa_mask;      /* segnali addizionali da
     bloccare prima dell'esecuzione del gestore */

int  sa_flags;            /* opzioni addizionali */
};
```

Notifiche (eventualmente multiple) dello stesso segnale durante l'esecuzione dell'azione sono bloccate fino al termine del gestore (a meno che `sa_flags` valga `SA_NODEFER`) quando ne viene notificata comunque una sola

Con la `sigaction` l'azione rimane permanentemente installata fino a quando non viene modificata



## Segnali

Attesa di un segnale con la gestione affidabile

```
int sigsuspend(const sigset_t *sigmask)
```

permette **l'**attivazione della signal mask specificata (sbloccando alcuni o tutti gli eventuali segnali pendenti) e **l'**attesa in modo **atomico**:

- se il segnale è pendente viene immediatamente eseguita l'azione corrispondente: successivamente, se non viene invocata la `exit`, la `sigsuspend` ritorna ;
- se il segnale non è pendente la `sigsuspend` attende fino alla notifica di un qualunque segnale che causa l'immediata esecuzione dell'azione corrispondente: se non viene invocata la `exit` la `sigsuspend` ritorna ;

`sigsuspend` ritorna sempre -1 con errno che vale EINTR



## Segnali

Uso di `sigsuspend`

```
...
sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask,SIGINT);
sigprocmask(SIG_BLOCK,&newmask, &oldmask);
/* Da qui il segnale SIGINT e' bloccato per il processo */
...

/* Sblocco di tutti i segnali pendenti
e attesa di un qualunque segnale */
sigsuspend(&zeromask);

sigprocmask(SIG_SETMASK,&oldmask, NULL);
/* sigsuspend rimette la maschera attiva prima
della sua chiamata: se necessario va rimessa
quella ancora precedente (oldmask) */
...
```



## Segnali

Esempio di interazione tra processi mediante segnali affidabili

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

void catcher(int signo)
{
static int ntimes = 0;
printf("Processo %d: SIGUSR1 ricevuto #%d volte\n",getpid(),++ntimes);
}

int main()
{
int pid, ppid;
struct sigaction sig, osig;
sigset_t sigmask, oldmask;

sig.sa_handler= catcher;
sigemptyset( &sig.sa_mask);
sig.sa_flags= 0;

sigemptyset( &sigmask);
sigaddset(&sigmask, SIGUSR1);
sigprocmask(SIG_BLOCK, &sigmask, &oldmask);

sigaction(SIGUSR1, &sig, &osig); /* il figlio la ereditera' */

/* Continua … */
```

## Segnali

```
/* ...continua */
if ((pid=fork()) < 0) {
  perror("fork error");
  exit(1);
  }
else
  if (pid == 0) {
    /* Processo figlio */
    ppid = getppid();
    printf("figlio: mio padre e' %d\n", ppid);
    while(1) {
        sleep(1);
        kill(ppid, SIGUSR1);
        /* Sblocca il segnale SIGUSR1 e lo attende */
        sigsuspend(&zeromask);
        }
    }
  else {
      /* Processo padre */
      printf("padre: mio figlio e' %d\n", pid);
      while(1) {
          /* Sblocca il segnale SIGUSR1 e lo attende */
          sigsuspend(&zeromask);
          sleep(1);
          kill(pid, SIGUSR1);
        }
    }
}
```



## Segnali

### Implementazione della `signal` mediante la `sigaction`

```
#include <signal.h>

typedef void Sigfunc(int);

Sigfunc *signal(int signo, Sigfunc *func)
{
struct sigaction      act,oact;

act.sa_handler = func;
sigemptyset(&act.sa_mask);
act.sa_flags= 0;

/* Il segnale SIGALARM viene normalmente utilizzato per avere un
timeout sulle primive bloccanti (read, connect, accept, ...) */
if(signo == SIGALRM)
#ifdef SA_INTERRUPT /* in SunOS si avrebbe di default il RESTART */
        act.sa_flags  |= SA_INTERRUPT;
#endif
} else {
#ifdef SA_RESTART /* in SVR4/4.3+BSD di default niente RESTART */
        act.sa_flags  |= SA_RESTART;
#endif
}

if(sigaction(signo, &act, &oact) < 0 ) return (SIG_ERR);
return (oact.sa_handler);
}
```