

# I segnali

# Segnali

- Sono ‘interruzioni’ software
  - comunicano al processo il verificarsi di un evento
  - ad ogni evento corrisponde un segnale numerato
  - un processo all’arrivo di un segnale di un certo tipo può decidere di
    - ignorarlo
    - lasciarlo gestire al kernel con l’azione di default definita per quel segnale
    - specificare una funzione (*signal handler*) che viene mandata in esecuzione appena il segnale viene rilevato

# Segnali (2)

- Da chi sono inviati i segnali?
  - da processo/thread all'altro
    - usando, **kill()** **pthread\_kill()**
  - dall'utente con particolari combinazioni di tasti (al processo in foreground)
    - Control-C corrisponde a SIGINT (ANSI)
    - Control-Z corrisponde a SIGTSTP
  - dall'utente con l'utility **kill** della shell
  - dal SO per a comunicare al processo il verificarsi di particolari eventi (es. SIGFPE, errore floating-point, SIGSEGV, segmentation fault)

# Segnali (3)

- Lo standard POSIX stabilisce un insieme di segnali riconosciuti in tutti i sistemi conformi
  - sono interi definiti come macro in `/usr/include/bits/signal.h`
  - esempi:
    - SIGKILL (9) : sintetico (e non può essere intercettata) (quit)
    - SIGALRM (14): è passato il tempo richiesto (quit)

# Segnali (4)

- SIGINT (2) Control-C
  - richiesta di interruzione (da tastiera) (quit)
- SIGTSTP (20) Control-Z
  - richiesta di sospensione (da tastiera) (suspend fino all'arrivo SIGCONT)
- SIGFPE (8)
  - si è verificato un errore Floating Point (dump)
- SIGCHLD(17)
  - si è verificato un cambiamento di stato in un processo figlio (ignore)
- SIGPIPE(13)
  - scrittura su pipe/socket senza lettori (quit)
- .....

# Segnali (5)

- SD del kernel relative ai segnali
  - *signal handler array* : descrive cosa fare quando arriva un segnale di un certo tipo
    - ignorare, trattare + puntatore al codice della funzione da eseguire (handler)
    - uno per processo (condiviso fra i thread)
  - *pending signal bitmap* : un bit per ogni tipo di segnale
    - il bit X è a 1 se c'è un segnale pendente di tipo X
    - una per thread
  - *signal mask* : un bit per ogni tipo di segnale
    - il bit X è a 1 se non voglio ricevere segnali di tipo X
    - una per thread

# Segnali (6)

- *Cosa accade quando arriva un segnale X ad un processo, con un solo thread?*
  - I segnali vengono inseriti nella *pending signal bitmap* e controllati al ritorno da SC
    - Se la *signal mask* non blocca il segnale X il processo che lo riceve viene interrotto
  - il kernel stabilisce quale comportamento adottare controllando il contenuto del *signal handler array*
    - default, SIG\_IGN, *signal\_handler*
  - se deve essere eseguito un *signal handler safun*:
    - si salva l'indirizzo di ritorno (quello dalla prossima istruzione)
    - si esegue **safun** e il processo riprende l'esecuzione dalla istruzione successiva a quella interrotta

# Segnali (7)

- *Cosa accade quando arriva un segnale X ad un processo, con più thread?*
  - Se il segnale è destinato ad un thread particolare T nel processo:
    - Se la signal mask di T non blocca il segnale X il thread viene interrotto
    - il kernel stabilisce quale comportamento adottare controllando il contenuto del *signal handler array* (globale nel processo)
    - se deve essere eseguito un signal handler si procede come già discusso



# Segnali (8)

- *Cosa accade quando arriva un segnale X ad un processo, con più thread?(segue)*
  - Se il segnale è destinato al processo: viene scelto un thread T a caso e si procede come prima

# Segnali (9)

- Tipicamente:
  - Se il segnale è dovuto ad un errore (divisione per 0, accesso erraneo alla memoria etc) viene inviato al thread che ha fatto l'errore
  - Tutti gli altri segnali vanno al processo
  - I segnali generati sinteticamente vanno al thread o al processo (dipende dall system call usata)

SC per i segnali

Sigaction etc.

# Personalizzare la gestione: sigaction

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

```
int sigaction(
```

```
    int signum,          /* segnale */
```

```
    const struct sigaction* act, /* nuova gest */
```

```
    struct sigaction* oldact /* vecchia gest */
```

```
);
```

```
/* (0) successo (-1) errore (sets errno) */
```

- **act** : struttura che definisce il nuovo trattamento del segnale signum;
- **oldact** : ritorna il contenuto precedente del signal handler array (può servire per ristabilire il comportamento precedente)
- con **act** a NULL si prende solo il vecchio gestore

# Personalizzare : sigaction (2)

```
struct sigaction {  
    void (*sa_handler) (int);  
    /*SIG_DFL, SIG_IGN o puntatore a funzione*/  
    sigset_t sa_mask; /* segnali da bloccare  
    (oltre a quello che stiamo registrando) */  
    int sa_flags; /* opzioni*/ ... /* altri c */  
};
```

**sa\_handler**: indica come gestire il segnale:

- **SIG\_IGN** ignora il segnale, **SIG\_DFL** usare la funzione di gestione di default altrimenti ho il puntatore alla funzione da invocare all'arrivo del segnale

# Personalizzare : sigaction (3)

```
struct sigaction {  
    void (*sa_handler) (int);  
    /*SIG_DFL, SIG_IGN o puntatore a funzione*/  
    sigset_t sa_mask; /* segnali da bloccare  
    (oltre a quello che stiamo registrando) */  
    int sa_flags; /* opzioni*/ ... /* altri c */  
};
```

**sa\_mask**: segnali da mascherare durante l'esecuzione del gestore:

- **sigaction** blocca automaticamente il segnale per cui stiamo registrando il gestore, ma se uso lo stesso gestore per due o più segnali devo bloccarli tutti esplicitamente per evitare stati inconsistenti (il gestore viene rieseguito prima di

# Personalizzare : sigaction (4)

```
struct sigaction {  
    void (*sa_handler) (int);  
    /*SIG_DFL, SIG_IGN o puntatore a funzione*/  
    sigset_t sa_mask; /* segnali da bloccare  
    (oltre a quello che stiamo registrando) */  
    int sa_flags; /* opzioni*/  
    ... /* altri campi */  
};
```

`sa_flags`: opzioni, ne vedremo esempi in seguito

possono esserci altri campi (system dependent)

# Esempio: gestire SIGINT

```
/* un gestore piuttosto semplice */  
static void gestore (int signum) {  
    printf("Ricevuto segnale %d\n", signum);  
    exit(EXIT_FAILURE);  
}
```



# Esempio: gestire SIGINT (2)

```
/* genero una sequenza infinita di interi */
int main (void) {
    struct sigaction s; int i;
    /* inizializzo s a 0*/
    bzero( &s, sizeof(s) );
    s.sa_handler=gestore; /* registro gestore */
    /* installo nuovo gestore s */
    err_meno1( sigaction(SIGINT,&s,NULL) );
    for (i=1; ;i++) { /* ciclo infinito */
        sleep(1);
        printf(“%d \n”,i);
    }
    exit(EXIT_SUCCESS); } /* mai eseguita */
```

# Esempio: gestire SIGINT (3)

```
/* compilazione ed esecuzione */
```

```
bash:~$ gcc -Wall -pedantic testsignum.c
```

```
bash:~$ ./a.out
```

```
1
```

```
2
```

```
3
```

# Esempio: gestire SIGINT (4)

*/\* compilazione ed esecuzione \*/*

```
bash:~$ gcc -Wall -pedantic testsignum.c
```

```
bash:~$ ./a.out
```

```
1
```

```
2
```

```
3
```

*CTRL-C*

```
Ricevuto segnale 2
```

```
bash:~$
```

# Esempio: ignorare SIGINT

```
/* genero una sequenza infinita di interi */
int main (void) {
    struct sigaction s; int i;
    /* inizializzo s a 0*/
    bzero( &s, sizeof(s) );
    s.sa_handler=SIG_IGN; /* registro gestore */
    /* installo nuovo gestore s */
    err_meno1( sigaction(SIGINT,&s,NULL) );
    for (i=1; ;i++) { /* ciclo infinito */
        sleep(1);
        printf( "%d \n", i );
    }
    exit(EXIT_SUCCESS); } /* mai eseguita */
```

# Esempio: ignorare SIGINT (2)

*/\* compilazione ed esecuzione \*/*

```
bash:~$ gcc -Wall -pedantic testsignum2.c
```

```
bash:~$ ./a.out
```

1

2

3

*CTRL-C -- ignorato*

4

5

# Esempio: ignorare SIGINT (3)

*/\* compilazione ed esecuzione \*/*

```
bash:~$ gcc -Wall -pedantic testsignum2.c
```

```
bash:~$ ./a.out
```

1

2

3

*CTRL-C*

4

5

*CTRL-\ -- invio SIGQUIT*

*Quit -- stampato dal gestore di default*

```
bash:~$
```

# Personalizzare la gestione (3)

- **SIGKILL** e **SIGSTOP**: non possono essere gestiti se non con la procedura di default
- **SIGKILL** uccide tutto il processo (non solo un thread)
- **SIGSTOP** blocca tutto il processo (non solo un thread)
- la gestione di default si applica sempre a tutti i thread del processo
- se il segnale è gestito, l'handler è eseguito solo da un thread
- i segnali **SIGCHLD** sono gli unici ad essere accumulati (*stacked*) negli altri casi se arriva un segnale dello stesso tipo di uno già arrivato viene perso

# Cosa mettere nell'handler

- Durante l'esecuzione dell'handler possono arrivare altri segnali: questo può generare situazioni inconsistenti
- Quindi:
  - l'handler deve essere breve, semplicemente aggiornare lo stato interno e/o terminare l'applicazione
  - non tutte le funzioni di libreria possono essere chiamate nell'handler con la garanzia che non succeda niente di strano (sul libro di testo p 616 o in rete (man signal)trovate una lista)
  - in particolare *non è safe* chiamare tipiche funzioni della libreria standard, come la printf(), la scanf() o altre funzioni definite all'interno del programma



# Esempio: gestire SIGINT (3)

```
/* un gestore corretto che usa solo funzioni  
signal safe */  
static void gestore (int signum) {  
    write(1, "Ricevuto SIGINT\n", 18);  
    _exit(EXIT_FAILURE);  
}  
/* ne printf() ne exit() sono garantite signal  
safe */
```

# Cosa mettere nell'handler (2)

- Inoltre non è nemmeno garantito l'accesso safe a variabili globali, a meno di non averle definite di tipo `volatile sig_atomic_t`

Quindi: usare i segnali il meno possibile, essenzialmente solo per situazioni standard

- gestire SIGINT, SIGTERM e simili, per ripulire l'ambiente in caso si richieda la terminazione dell'applicazione
- gestire SIGSEGV e simili, per evitare il display diretto di errori brutti tipo Segmentation fault, Bus error etc ... (dopo segnali di questo tipo bisogna però SEMPRE uscire, la memoria non è più garantita essere consistente)
- ignorare SIGPIPE (ad esempio in modo da non far terminare il server se un client ha riattaccato)

# fork , exec , pthread\_create

- Funzioni di gestione (handler):
  - con la `fork()` il figlio eredita la gestione dei segnali dal padre
  - dopo la `exec()` le gestioni ritornano quelle di default (ma i segnali ignorati continuano ad essere ignorati)
    - infatti il codice del gestore non esiste più
  - siccome le funzioni di gestione interessano tutto il processo non vengono alterate da `pthread_create()`

# fork , exec , pthread\_create (2)

- Signal mask:
  - con la `fork()` il figlio eredita la signal mask dal padre
  - dopo la `exec()` la signal mask rimane la stessa
  - la signal mask del nuovo thread viene ereditata dal thread che invoca la `pthread_create()`

# **fork , exec , pthread\_create (3)**

- Maschera dei segnali pendenti:
  - con la **fork()** viene messa a 0 (nessun segnale pendente)
  - rimane la stessa del thread che ha invocato la **exec()**
  - viene azzerata dalla **pthread\_create()**

# Mascherare: pthread\_sigmask

```
#include <pthread.h>
#include <signal.h>
int pthread_sigmask(
    int how,          /* come si deve cambiare */
    const sigset_t set, /* insieme di segnali
                        (bitmap) */
    sigset_t* oldset /* vecchia maschera */
);
/* (0) successo (codice di errore) se errore
(NON setta errno) */
```

- **set** : una maschera di bit che serve per modificare la signal mask (vedi poi);

# Mascherare : pthread\_sigmask (2)

- **how**: come vogliamo cambiare la signal mask in base ai segnali specificati da **set**
  - **SIG\_BLOCK**: la nuova *signal mask* diventa l'or di **set** e della vecchia signal mask (i segnali in set sono aggiunti alla maschera)
  - **SIG\_SETMASK**: la nuova *signal mask* diventa **set** , indipendentemente dal valore della vecchia
  - **SIG\_UNBLOCK**: la nuova *signal mask* rimuove i segnali presenti in set dalla vecchia signal mask
- **oldset**: se oldset non è nullo restituisce il valore della vecchia signal mask, prima di effettuare la modifica

# Mascherare : pthread\_sigmask (3)

Funzioni per la maschera di bit da passare come set,

- `int sigemptyset(sigset_t* pset);` azzera la maschera puntata da `pset`
- `int sigfillset(sigset_t* pset);` mette a uno tutte le posizioni della maschera puntata da `pset`
- `int sigaddset(sigset_t* pset, int signum);` mette a 1 la posizione del segnale `signum` in `pset`
- `int sigdelset(sigset_t* pset, int signum);` mette a 0 la posizione relativa a `signum` in `pset`
- `int sigismember(const sigset_t* pset, int signum);` restituisce 1 se `signum` è membro della maschera `pset`, 0 se non lo è e -1 in caso di errore



# Mascherare : pthread\_sigmask (4)

– Vediamo degli esempi di maschere

```
sigset_t set;
```

*-- crea una maschera per tutti i segnali*

```
ec_meno1( sigfillset(&set) )
```

*-- toglie dalla maschera SIGINT*

```
ec_meno1( sigdelset(&set, SIGINT) )
```

# Mascherare : pthread\_sigmask (5)

- Quando è utile mascherare i segnali?
  - Se un segnale deve essere ignorato per tutta la durata dell'applicazione meglio gestirlo con SIG\_IGN
  - è utile invece per mascherare i segnali per non essere interrotti durante l'esecuzione di un gestore
    - il segnale per cui è registrato il gestire è automaticamente mascherato durante la gestione (se usiamo sigaction) ma gli altri no
  - nello startup del programma quando ancora non abbiamo registrato tutte le gestione con **sigaction**

# Mascherare : pthread\_sigmask (6)

- Esiste anche la **sigprocmask**
  - era la vecchia SC per processi con un solo thread
  - funziona come l'altra, solo ritorna -1 in caso di errore settando errno

# Esempio: gestione minimale dei segnali

```
static int handle_signals(void) {
    sigset_t set;
    struct sigaction sa;
    /* maschero tutti i segnali finchè i gestori
    permanenti non sono installati */
    ec_meno1( sigfillset(&set) )
    ec_meno1( sigprocmask(SIG_SETMASK, &set, NULL) )
    memset(&sa, sizeof(sa), 0);
    sa.sa_handler = SIG_IGN;
    ec_meno1(sigaction(SIGINT, &sa, NULL))
    ec_meno1(sigaction(SIGQUIT, &sa, NULL))
    ec_meno1(sigaction(SIGPIPE, &sa, NULL))
```

# Esempio: gestione minimale ... (2)

```
sa.sa_handler = gestoreTERM;
ec_meno1(sigaction(SIGTERM, &sa, NULL)
sa.sa_handler = gestoreSEGV;
ec_meno1(sigaction(SIGSEGV, &sa, NULL)
/*altri gestori */
...
/*tolgo la maschera */
ec_meno1( sigemptyset(&set) )
ec_meno1( sigprocmask(SIG_SETMASK, &set, NULL) )
return 0;
}
```

# Inviare un segnale a processo: kill

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

```
int kill(
```

```
    pid_t pid,          /* pid processo */
```

```
    int signum         /* segnale da inviare */
```

```
);
```

```
/* (0) successo (-1) se errore (setta errno) */
```

– genera un segnale sintetico di tipo `signum` e lo invia a uno o più processi (dipende da `pid`)

- `pid>0` : il processo di pid `pid`
- `pid=0` : i processi dello stesso gruppo di chi ha invocato `kill`
- `pid<0` : i processi dello gruppo `-pid`
- `pid=-1`: tutti i processi per cui l'utente ha il permesso

# Inviare un segnale a processo: kill (2)

- il segnale viene inviato solo se
  - il processo che invia il segnale e chi lo riceve hanno lo stesso owner
  - il processo che invia il segnale è owned dal superutente (**root**)

# Inviare segnale a thread: `pthread_kill`

```
#include <pthread.h>
```

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

```
int pthread_kill(  
    pthread_t tid,          /* tid thread */  
    int signum             /* segnale da inviare */  
);  
/* (0) successo (err number) se errore (NON  
   setta errno) */
```

- genera un segnale sintetico di tipo `signum` e lo invia al thread `tid`, che deve appartenere allo stesso processo



## ...segnale a thread: `pthread_kill` (2)

- attenzione alla `pthread_kill`: i segnali che di default terminano o uccidono lo fanno per tutto il processo
  - es: `pthread_kill(tid, SIGKILL)` uccide sempre tutti i thread del processo
- quindi conviene usarla solo per inviare segnali per cui è stato installato un gestore!

# Attendere un segnale: pause

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

```
int pause (void);
```

```
/* (-1) se errore (setta errno) */
```

- funzione di attesa generica ...
- attende finchè non viene interrotta da un segnale, in questo caso se il segnale è gestito e il gestore ritorna, la pause ritorna -1 con errore EINTR

# Attendere un segnale: sigwait

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

```
int sigwait (  
    const sigset_t *set, /* segnali da attendere */  
    int * signum        /* segnale arrivato */  
);  
/* (0) successo (errno) se errore (NON setta  
   errno) */
```

- **set** a permette di specificare i segnali da attendere con una maschera (come per la signal mask)
- quando ritorna **signum** contiene il segnale effettivamente ricevuto

# Esempio : l'attesa una di sveglia

- vogliamo metterci in attesa per 5 secondi ed essere svegliati all'arrivo di un segnale :
  - SIGALRM è il segnale di sveglia, vediamo come generarlo con `alarm ...`

# Settare un timer: alarm

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

```
unsigned alarm(
```

```
    unsigned secs,      /* secondi da aspettare */
```

```
);
```

```
/* (x>0) secondi ancora da passare da un allarme  
   settato precedentemente (0) se non ce ne sono  
   (NON ritorna errori) */
```

- genera un segnale **SIGALRM** e lo invia al processo che l'ha invocata dopo **secs** secondi
- se **secs** è 0 non setta nessun allarme
- le richieste di allarme precedenti sono cancellate
- **ATTENZIONE**: non può essere usata insieme a **sleep()**

# Esempio: timer a 3 sec

```
int main (void) {  
    alarm(3);          /* SIGALRM fra 3 secondi */  
    printf("Inizio il ciclo infinito ..\n") ;  
    while (1) ;       /* ciclo infinito */  
    printf("Pippo\n") ; /* mai eseguita */  
    return 0 ;  
}
```

# Esempio: timer a 3 sec (2)

se eseguiamo il codice dell'esempio :

```
bash:~$ ./a.out
```

```
Inizio il ciclo infinito ...
```

```
-- per (circa) 3 secondi non accade niente
```

# Esempio: timer a 3 sec (3)

se eseguiamo il codice dell'esempio :

```
bash:~$ ./a.out
```

```
Inizio il ciclo infinito ...
```

```
Alarm clock          -- arriva il segnale
```

```
-- processo terminato (gestione di default)
```

```
bash:~$
```



# Esempio: timer a 3 sec (4)

```
/* per avere un vero timer dobbiamo
personalizzare SIGALRM */
void gestore (int sig) {
    write(1,"SIGALRM catturato\n",18) ;
}
```

# Esempio: timer a 3 sec (5)

```
int main (void) {
struct sigaction s;
ec_meno1(sigaction(SIGALRM, NULL, &s) )
s.sa_handler=gestore;
ec_meno1(sigaction(SIGALRM, &s, NULL) )
alarm(3);          /* SIGALRM fra 3 secondi */
printf("Ciclo infinito ...\n") ;
while (1) ;       /* ciclo infinito */
printf("Pippo\n") ; /* mai eseguita */
return 0 ;
}
```

# Esempio : timer a 3 sec (6)

cosa accade se eseguiamo il codice dell'esempio :

```
bash: ~$ ./a.out
```

```
Ciclo infinito ...
```

*-- per (circa) 3 secondi non accade niente*

# Esempio : timer a 3 sec (7)

cosa accade se eseguiamo il codice dell'esempio :

```
bash: ~$ ./a.out
```

```
Ciclo infinito ...
```

```
SIGALRM catturato    -- arriva il segnale  
                    -- il processo cicla indefinitamente ...
```

# Esempio: commenti ...

- L'attesa attiva è costosa...

- per vederlo bene eseguite

```
bash:~$ gcc -Wall -pedantic -O3 alrm3.c
```

```
-- compilazione ottimizzata
```

```
bash:~$ time ./a.out
```

```
Ciclo infinito ...
```

```
SIGALRM catturato
```

```
CTRL-C
```

```
real 0m4.017s
```

```
user 0m3.941s
```

```
sys 0m0.005s
```

```
bash:~$
```

## Esempio: commenti ... (2)

- Abbiamo sprecato tre secondi di test continui del ciclo `while`!
  - Vediamo adesso come sospenderci nell'attesa del segnale `SIGALRM` senza sprecare CPU time con **pause** e **sigwait**
  - usando **pause**, abbiamo bisogno di un signal handler e di discriminare il segnale arrivato
  - **sigwait** permette di attendere senza installare un gestore e quando il segnale è arrivato possiamo gestirlo senza restrizioni perché non siamo in un signal handler ma stiamo eseguendo normale codice C

# Esempio: pause

```
/* indica se è arrivato SIGALARM (=1) o no (=0) */
```

```
volatile sig_atomic_t int sigalarm_flag = 0;
```

```
void gestore (int sig)
    sigalarm_flag = 1;
}
```

```
int main (void) {
    struct sigaction s ;
    .....
}
```

## Esempio: pause (2)

```
int main (void) {
    struct sigaction s;
    ec_meno1(sigaction(SIGALRM, NULL, &s))
    s.sa_handler=gestore;
    ec_meno1(sigaction(SIGALRM, &s, NULL))
    alarm(3);          /* SIGALRM fra 3 secondi */
    printf("Ciclo infinito ...\n") ;
    while (sigalarm_flag != 1)
        pause(); /* ciclo fino a SIGALRM */
/* serve a mettersi in pausa solo se sigalarm
NON è ancora arrivato */
    printf("Pippo\n") ;    return 0 ;
}
```



# Esempio : pause (3)

cosa accade se eseguiamo il codice dell'esempio :

```
bash: ~$ ./a.out
```

```
Ciclo infinito ...
```

```
-- per (circa) 3 secondi non accade niente
```

# Esempio : pause (4)

cosa accade se eseguiamo il codice dell'esempio :

```
bash:~$ ./a.out
```

```
Ciclo infinito ...
```

```
-- per (circa) 3 secondi non accade niente
```

```
Pippo
```

```
-- arriva il segnale
```

```
-- processo terminato
```

```
bash:~$
```

# Esempio : pause (5)

ed il tempo di CPU è salvo :

```
bash:~$ time ./a.out
```

```
Ciclo infinito ...
```

*-- per (circa) 3 secondi non accade niente*

```
Pippo
```

*-- arriva il segnale*

*-- processo terminato*

```
real 0m3.002s
```

```
user 0m0.001s
```

```
sys 0m0.001s
```

```
bash:~$
```

# Esempio : pause (6)

Però non siamo ancora del tutto soddisfatti:

- se SIGALARM arriva dopo il test del while ma prima della pause il processo non si sveglia più

```
.....  
printf("Ciclo infinito ...\n") ;  
while (sigalarm_flag != 1)  
    pause(); /* ciclo fino a SIGALRM */  
/* serve a mettersi in pausa solo se sigalarm  
NON è ancora arrivato (possible deadlock)*/  
printf("Pippo\n") ; return 0 ;  
...
```

# Esempio : pause (7)

Però non siamo ancora del tutto soddisfatti (cont):

- possiamo usare **sigsuspend** invece di pause:
  - questo permette di mascherare SIGALRM fino all'attesa
  - e sboccarlo atomicamente con la messa in attesa

.....

```
/* set è la maschera corrente */  
ec_meno1( sigdelset(&set, SIGALRM) );  
while (sigalarm_flag != 1)  
    sigsuspend(&set); /* ciclo fino a SIGALRM */  
/* si può registrare di nuovo la vecchia  
maschera */
```

...

# Esempio: sigwait

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

int main (void) {
    sigset_t set; int sig;
    /* costruisco la maschera con solo SIGALRM */
    sigemptyset (&set);
    sigaddset (&set, SIGALRM);
    /* blocco SIGALRM */
    pthread_sigmask (SIG_SETMASK, &set, NULL);
    alarm (3); /* SIGALRM fra 3 secondi */
    printf ("Inizio attesa ...\n");
    sigwait (&set, &sig);
    printf ("Pippo, %d\n", sig); return 0;}

```

# Esempio : sigwait (2)

cosa accade se eseguiamo il codice dell'esempio :

```
bash:~$ gcc -Wall -pedantic a1sigw.c -lpthread
```

```
bash:~$ ./a.out
```

```
Inizio attesa ...
```

*-- per (circa) 3 secondi non accade niente*

# Esempio : sigwait (3)

cosa accade se eseguiamo il codice dell'esempio :

```
bash: ~$ ./a.out
```

```
Inizio attesa ...
```

```
-- per (circa) 3 secondi non accade niente
```

```
Pippo 14
```

```
-- arrivato SIGALRM
```

```
-- processo terminato
```

```
bash: ~$
```



# Gestione di segnali con sigwait

- È possibile usare `sigwait` in ambiente multithreaded per gestire i segnali destinati al processo senza installare gestori:
  - usiamo un thread *handler* come gestore dei segnali
  - tutti i thread mascherano tutti i segnali all'inizio
  - handler fa continuamente delle `sigwait`, e ogni volta che ritorna gestisce il segnale corrispondente nel codice normale del thread
  - non funziona se il segnale è destinato direttamente a un thread

# Interrupted System Call

- Un segnale non mascherato può interrompere l'esecuzione di una system call:
  - es. la **pause** appena vista, una **read** in attesa
  - in questo caso *se la gestione di default o il gestore terminano il processo non succede niente di strano*
  - se invece il gestore ritorna, la system call non riprende a fare quello che stava facendo (es. attendere una scrittura su stdin) ma fallisce con codice di errore EINTR (Interrupted System Call)
  - Vediamo prima un esempio di read interrotta

# Esempio: Interrupted SC

```
/* lettura da stdin interrotta ... */  
  
/* un gestore che ritorna senza terminare il  
processo ... */  
void gestore (int sig)  
    write(1, "SIGALRM catturato\n",18);  
}  
  
int main (void) {  
    struct sigaction s ;  
    .....  
}
```

# Esempio: Interrupted SC (2)

```
int main (void) {
    struct sigaction s; char buf[100];
    ec_meno1(sigaction(SIGALRM, NULL, &s))
    s.sa_handler=gestore;
    ec_meno1(sigaction(SIGALRM, &s, NULL))
    alarm(3);          /* SIGALRM fra 3 secondi */
    printf("Attendo la read ...\n") ;
    if (read(0, buf, 100) == -1) {
        perror("read") ;
    }
    return 0 ;
}
```

# Esempio : Interrupted SC (3)

cosa accade se eseguiamo il codice dell'esempio :

```
bash:~$ gcc -Wall -pedantic eseISC.c
```

```
bash:~$ ./a.out
```

```
Attendo la read ...
```

*-- per (circa) 3 secondi non accade niente*

# Esempio : Interrupted SC (4)

cosa accade se eseguiamo il codice dell'esempio :

```
bash: ~$ ./a.out
```

```
Attendo la read ...
```

```
-- per (circa) 3 secondi non accade niente
```

```
SIGALRM catturato -- scritto dal gestore
```

```
read: Interrupted system call -- perror
```

```
bash: ~$
```

# Interrupted System Call (2)

- Ma quali system call possono essere interrotte?
  - Tipicamente le SC che si possono bloccare in attesa di un evento (stato di *blocked*),
    - es: `read`, `open`, `pause`, ma anche `pthread_cond_wait()`.
  - Però non tutte le SC che possono bloccarsi possono essere interrotte:
    - es `pthread_mutex_lock()`
  - Per sapere esattamente quali sono interrompibili e quali no, leggere attentamente la documentazione

# Interrupted System Call (3)

- Come ci dobbiamo comportare quindi?
  - ogni volta che registriamo un gestore di segnale che ritorna *dobbiamo considerare con estrema cura le possibili interazioni con SC interrompibili*
  - a volte essere interrotti è esattamente il comportamento desiderato,
    - ad esempio vogliamo dare un timeout al verificarsi di un certo evento (es l'arrivo dei dati della `read`) e quindi *non vogliamo riprendere quello che stavamo facendo*
  - se questo non va bene (cioè se possono verificarsi problemi nell'algoritmo) bisogna trattare il caso esplicitamente



# Interrupted System Call (4)

- Come si può trattare l'interruzione indesiderata?
  - se non vogliamo MAI essere interrotti si può chiedere all'atto della registrazione del gestore che ritorna (flag **SA\_RESTART** ) o invocando **siginterrupt**
  - altrimenti si può gestire esplicitamente riattivando la SC interrotta all'interno di un ciclo
    - abbiamo già visto degli esempi con le wait su condition variable
  - vediamo le due alternative nel nostro esempio

# Esempio: Restarted SC

```
/* esempio lettura da stdin interrotta rivisto  
per evitare l'interruzione ... */
```

```
/* un gestore che ritorna senza terminare il  
processo ... */
```

```
void gestore (int sig)
```

```
    write(1, "SIGALRM catturato\n", 18);
```

```
}
```

```
int main (void) {
```

```
    struct sigaction s ;
```

```
    .....
```

```
}
```

# Esempio: Restarted SC (2)

```
int main (void) {
    struct sigaction s; char buf[100];
    ec_meno1(sigaction(SIGALRM, NULL, &s))
    s.sa_handler=gestore;
    s.sa_flags=SA_RESTART;
    ec_meno1(sigaction(SIGALRM, &s, NULL))
    alarm(3);          /* SIGALRM fra 3 secondi */
    printf("Attendo la read ...\n") ;
    if (read(0, buf, 100) == -1) {
        perror("read") ;
    }
    return 0 ;
}
```

# Esempio : Restarted SC (3)

cosa accade se eseguiamo il codice dell'esempio :

```
bash:~$ gcc -Wall -pedantic eseREST.c
```

```
bash:~$ ./a.out
```

```
Attendo la read ...
```

*-- per (circa) 3 secondi non accade niente*

# Esempio : Restarted SC (4)

cosa accade se eseguiamo il codice dell'esempio :

```
bash: ~$ ./a.out
```

```
Attendo la read ...
```

*-- per (circa) 3 secondi non accade niente*

```
SIGALRM catturato          -- scritto dal gestore
```

*-- la read viene fatta ripartire e si  
rimette in attesa di input*

# Esempio: Restarted SC (5)

```
int main (void) {
    struct sigaction s; char buf[100]; int l;
    ec_meno1(sigaction(SIGALRM, NULL, &s))
    s.sa_handler=gestore;
    ec_meno1(sigaction(SIGALRM, &s, NULL))
    alarm(3);
    printf("Attendo la read ... \n") ;
    while( (l=read(0, buf, 100) == -1) && (errno==EINTR) )
        ; /* se sono stato interrotto riattivo read */
    if (l<0) perror("read") ;
    else write(1, buf, l);
    return 0 ;
}
```

# Esempio : Restarted SC (6)

cosa accade se eseguiamo il codice dell'esempio :

```
bash:~$ gcc -Wall -pedantic eseRESTwhile.c
```

```
bash:~$ ./a.out
```

```
Attendo la read ...
```

*-- per (circa) 3 secondi non accade niente*

# Esempio : Restarted SC (7)

cosa accade se eseguiamo il codice dell'esempio :

```
bash: ~$ ./a.out
```

```
Attendo la read ...
```

*-- per (circa) 3 secondi non accade niente*

```
SIGALRM catturato          -- scritto dal gestore
```

*-- la read viene fatta ripartire e si  
rimette in attesa di input*



# Esempio : Restarted SC (8)

cosa accade se eseguiamo il codice dell'esempio :

```
bash: ~$ ./a.out
```

```
Attendo la read ...
```

*-- per (circa) 3 secondi non accade niente*

```
SIGALRM catturato          -- scritto dal gestore
```

*-- la read viene fatta ripartire e si  
rimette in attesa di input*

```
ciccio    -- eco caratteri digitati
```

```
ciccio    -- output write
```

```
bash: ~$
```

# Segnali: system call obsolete

- `signal()`:
  - permette di registrare direttamente un gestore (senza usare `struct sigaction` o `sigaction`).
  - Questa è una SC estremamente pericolosa:
    - il comportamento con più thread non è PREDICIBILE (vedi man)
    - appena il segnale viene catturato ed il gestore invocato la ritorna il gestore di default (quindi il gestore va subito registrato di nuovo altrimenti parte la gestione di default)
    - il segnale appena arrivato NON viene mascherato di default, quindi se arriva di nuovo durante l'esecuzione del gestore il gestore viene interrotto con la gestione di default (eventualmente terminando il processo!)

# Segnali: system call obsolete (2)

- **sigset, sigrelse, sigpause, sighold, sigignore:**
  - sono modi equivalenti di fare cose permesse dalle system call già viste
  - spesso le interazioni sono sottilmente differenti
  - non sono definite in ambiente multithreading