

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#define MAXPAROLA 30
#define MAXRIGA 80
```

```
int main(int argc, char *argv[])
```

```
{
    int freq[MAXPAROLA]; /* vettore di contatori
    delle frequenze delle lunghezze delle parole */
    char riga[MAXRIGA];
    int i, inizio, lunghezza;
    FILE *f;
```

```
for(i=0; i<MAXPAROLA; i++)
    freq[i]=0;
```

```
if(argc != 2)
```

```
{
    fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
    exit(1);
}
```

```
f = fopen(argv[1], "r");
if(f==NULL)
```

```
{
    fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
    exit(1);
}
```

```
while( fgets( riga, MAXRIGA, f ) != NULL )
```



Processi

Segnali

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

Informazioni sul Copyright

This work is licensed under the license



Attribution-NonCommercial-NoDerivatives 4.0 International

This license requires that reusers give credit to the creator. It allows reusers to copy and distribute the material in any medium or format in unadapted form and for noncommercial purposes only.

① **BY:** Credit must be given to you, the creator.

② **NC:** Only noncommercial use of your work is permitted.

Noncommercial means not primarily intended for or directed towards commercial advantage or monetary compensation.

③ **ND:** No derivatives or adaptations of your work are permitted.

To view a copy of the license, visit:

<https://creativecommons.org/licenses/by-nc-nd/4.0/?ref=chooser-v1>

Interruzioni

❖ Interrupt

- Interruzione dell'esecuzione corrente dovuto al verificarsi di un evento straordinario

❖ Può essere provocato da

- Un dispositivo hardware che invia una richiesta di servizio alla CPU
- Un processo software che richiede l'esecuzione di una particolare operazione tramite una system call

Definizione

❖ Un **segnale** è

- Un interrupt software
- Un notifica, inviata dal kernel o da un processo a un altro processo, per notificargli che si è verificato un determinato evento

❖ I segnali

- Permettono di gestire eventi asincroni
 - Notificano il verificarsi di eventi particolari (e.g., condizioni di errore, violazioni di accesso in memoria, errori di calcolo, istruzioni illegali, etc.)
- Possono venire utilizzati per la comunicazione tra processi

Caratteristiche

- ❖ Disponibili dalle primissime versioni di UNIX
 - Originariamente gestiti in maniera poco affidabile
 - Potevano andare perduti
 - Unix Version 7: un segnale poteva essere inviato e mai ricevuto
 - Alla ricezione di ogni segnale il comportamento (per quel segnale) tornava ad essere quello di default
 - Richiedevano il **ricaricamento** del signal handler
 - Un processo non poteva ignorare la ricezione di un segnale

Caratteristiche

- ❖ Standardizzati dallo standard POSIX sono ora stabili e relativamente affidabili
- ❖ Ogni segnale ha un nome
 - I nomi incominciano per **SIG...**
 - Il file **signal.h** definisce i nomi dei segnali
 - Unix FreeBSD, Max OS X e Linux supportano 31 segnali
 - Solaris supporta 38 segnali

whereis signal.h
→
/usr/include/signal.h

Il comando **kill -l** fornisce
l'elenco dei segnali

Segnali principali

Nome	Descrizione
SIGABRT	Process abort, generato chiamando la funzione abort
SIGALRM	Alarm clock, generato dalla funzione alarm a seguito di una system call sleep(t); per default fa ripartire il processo
SIGFPE	Floating-Point exception
SIGINT	Inviato al padre di un processo che riceve un ctrl-C da tastiera; per default si termina il processo
SIGKILL	Kill (non mascherabile)
SIGPIPE	Write on a pipe with no reader
SIGSEGV	Invalid memory segment access
SIGCHLD	Inviato al padre di un processo figlio che è terminato; l'azione di default è ignorare il segnale
SIGUSR1 SIGUSR2	User-defined signal 1/2 Comportamento di default: terminazione Disponibile per utilizzo in applicazioni utente

Segnali inviati dall'exception handler

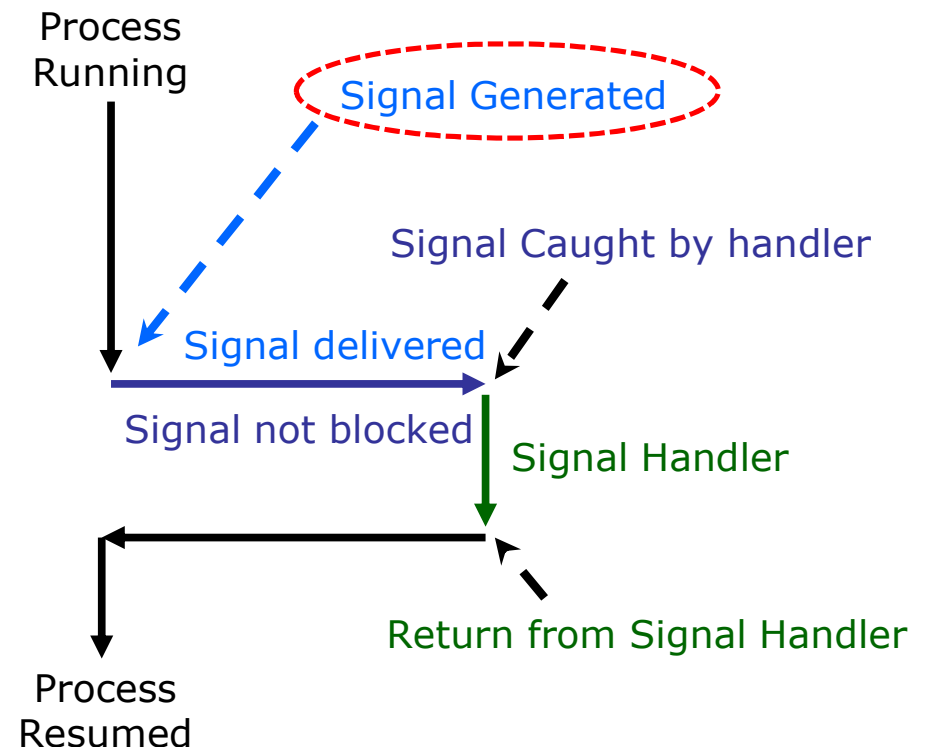
Nome	Eccezione	Descrizione
SIGFPE	divide_erro()	Divide error
SIGTRAP	debug()	Debug
SIGTRAP	int3()	Breakpoint
SIGSEGV	overflow()	Overflow
SIGSEGV	bounds()	Bounds check
SIGILL	invalid_op()	Invalid opcode
SIGBUS	segment_not_present()	Segment not present
SIGBUS	stack_segment()	Stack segment fault
SIGSEGV	general_protection()	General protection
SIGSEGV	page_fault()	Page fault
None	none	Interval reserved
SIGFPE	coprocessor_erro()	Floating point error

Gestione dei segnali

❖ La gestione di un segnale implica tre fasi

1. Generazione del segnale

- Avviene quando il kernel o il processo sorgente effettua l'evento necessario



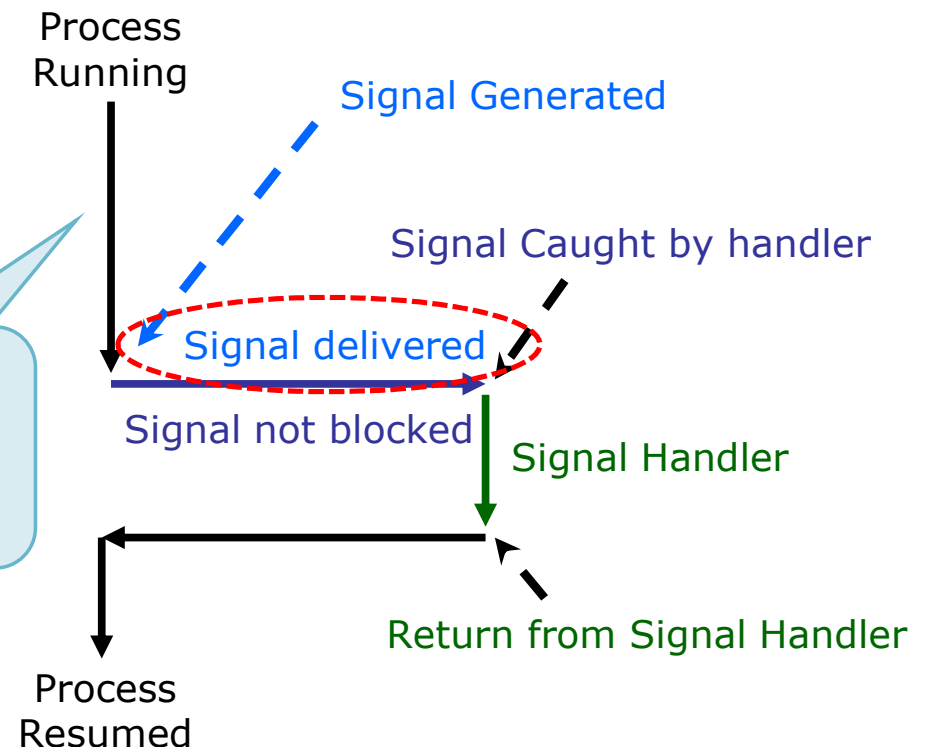
Gestione dei segnali

2. Consegna del segnale

- Un segnale non consegnato risulta pendente
- Un segnale ha un tempo di vita che va dalla sua generazione alla sua consegna
- Alla consegna il processo destinatario assume le azioni richieste dal segnale

Non esiste una coda di segnali; il kernel setta un flag nella process table

Flusso esecuzione standard iniziale



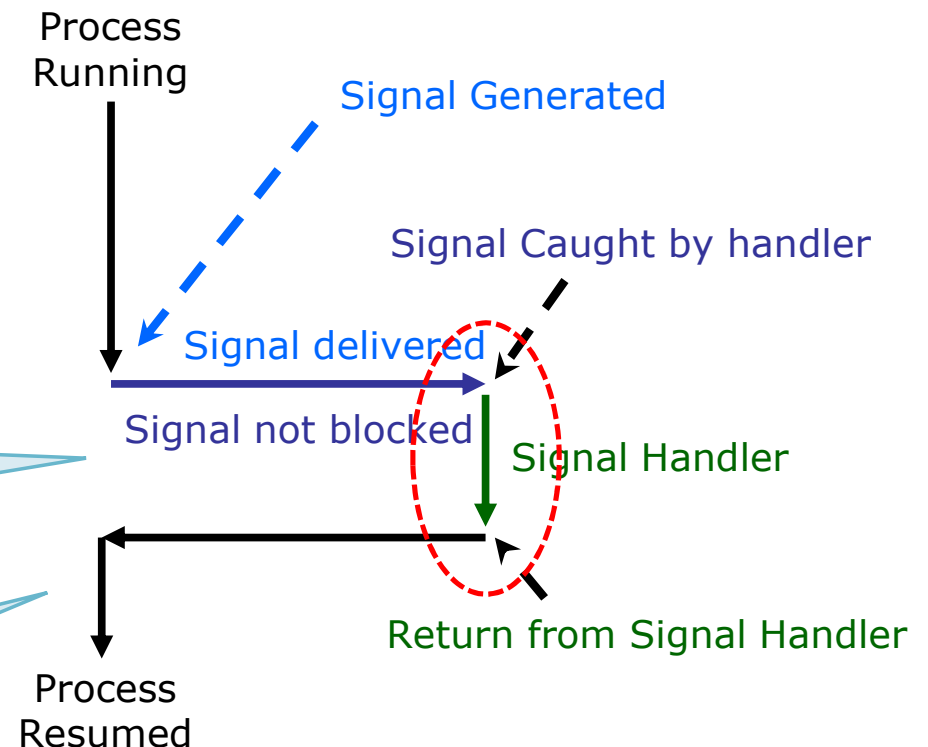
Gestione dei segnali

3. Gestione del segnale

- Per gestire un segnale un processo deve dire al kernel che cosa intende fare nel caso riceva tale segnale
- Tipicamente è possibile impostare **3** comportamenti
 - Utilizzare il comportamento di default
 - Ignorare il segnale
 - Gestire (catturare, "catch") il segnale

Cattura di un segnale attraverso un **"signal handler"**

Riprende flusso esecuzione standard



Gestione dei segnali

- ❖ La gestione di un segnale può venire effettuata con cinque system call principali

System call	Descrizione
signal	Instanzia un gestore di segnali allo scopo di gestire uno specifico segnale in maniera opportuna
kill	Invia uno specifico segnale a un processo definito
raise	Invia un segnale al chiamante stesso
pause	Sospende un processo per una durata di tempo indefinita, sino all'arrivo di un segnale
alarm	Attiva un timer (ovvero un count-down)

I termini **signal** e **kill** sono relativamente inappropriati

System call signal

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

```
void (*signal (int sig,  
               void (*func) (int))) (int);
```

Parametro
del signal
handler
ricevuto

Parametro
del signal
handler
ritornato

- ❖ Consente di istanziare un gestore di segnali
- ❖ Ovvero specifica
 - Il segnale da gestire (**sig**) la cui ricezione attiverà
 - La funzione per gestirlo (**func**), i.e., il **signal handler**

System call signal

❖ Parametri

- **sig** indica il segnale da intercettare
 - SIGCHLD, SIGUSR1, etc.
- **func** specifica l'indirizzo (i.e., puntatore) della funzione da invocare quando si presenta il segnale
 - Tale funzione ha un solo parametro di tipo int
 - Esso indica a sua volta il segnale da gestire
 - Ritorna un void *
 - In puntatore a una funzione

Parametro
del signal
handler
ricevuto

Parametro
del signal
handler
ritornato

```
void (*signal (int sig, void (*func) (int))) (int) ;
```

System call signal

❖ Valore di ritorno

- In caso di successo il puntatore al signal handler che gestiva il segnale in precedenza e che
 - Ha ricevuto come parametro un intero (il codice intero dell'errore)
 - Ritorna un void *
- **SIG_ERR** in caso di errore
 - #define SIG_ERR ((void (*)(int)) -1)

Cast unknown name into
pointer to function (int)
returning void

```
void (*signal (int sig, void (*func)(int)))(int);
```


System call signal

- ❖ La system call **signal** permette di impostare i tre comportamenti previsti in precedenza
 - Lasciare che si verifichi, per ciascun possibile segnale, il comportamento di default
 - signal (SIG..., **SIG_DFL**)
 - Dove **SIG_DFL** è una costante definita in signal.h
 - #define SIG_DFL ((void (*)(int)) 0)
 - e utilizzata al posto del puntatore al signal handler
 - Ogni segnale ha un comportamento di default definito dal sistema
 - La maggior parte dei comportamenti di default consistono nel terminare il processo

System call signal

➤ Ignorare esplicitamente il segnale

- signal (SIG..., **SIG_IGN**)
- Dove **SIG_IGN** è una costante definita in signal.h
 - #define SIG_IGN ((void (*)(int)) 1)
- Applicabile alla maggior parte dei segnali
 - Ignorare un segnale spesso comporta avere un comportamento indefinito
- Alcuni segnali non possono essere ignorati
 - SIGKILL e SIGSTOP non possono essere ignorati altrimenti non si permetterebbe al kernel (o allo superuser) di avere controllo su tutti i processi
 - In altri casi il comportamento del processo sarebbe indefinito se il segnale venisse ignorato, ad esempio nel caso si riceva un segnale per un accesso illegale in memoria SIGTSTP

System call signal

➤ Catturare (catch) il segnale

- signal (SIG..., **signalHandlerFunction**)

- Dove

- **SIG...** indica il segnale che si vuole gestire
- **signalHandlerFunction** è la **funzione utente** di gestione del segnale, richiamata nel caso si presenti il segnale

- La funzione di gestione

- Può intraprendere le azioni ritenute corrette per la gestione del segnale
- È richiamata in maniera asincrona una volta ricevuto il segnale associato
- Quando ritorna, il processo continua come se non fosse mai stato interrotto

Occorre una funzione di gestione **per ogni** segnale che si vuole gestire

Esempio 1

Gestione esplicita
di **un** segnale

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

void manager (int sig) {
    printf ("Ricevuto il segnale %d\n", sig);
    // (void) signal (SIGINT, manager);
    return;
}

int main() {
    signal (SIGINT, manager);
    while (1) {
        printf ("main: Hello!\n");
        sleep (1);
    }
}
```

Signal handler per il
segnale SIGINT

Versioni obsolete:
re-istanziamento del
signal handler

Istanza il signal
handler

Esempio 2

Gestione esplicita
di **più** segnali

Signal handler che
gestisce più segnali

```
...  
void manager (int sig) {  
    if (sig==SIGUSR1)  
        printf ("Ricevuto SIGUSR1\n");  
    else if (sig==SIGUSR2)  
        printf ("Ricevuto SIGUSR2\n");  
    else printf ("Ricevuto %d\n", sig);  
    return;  
}  
...  
int main () {  
    ...  
    signal (SIGUSR1, manager);  
    signal (SIGUSR2, manager);  
    ...  
}
```

Il signal handler deve
essere istanziato
per tutti i segnali in
maniera esplicita

Esempio 3-A

Gestione **sincrona**
di SIGCHLD (con
wait)

Quando un figlio
(PID=3057) muore al
padre viene inviato un
SIGCHLD

```
if (fork() == 0) {  
    // child  
    i = 2;  
    sleep (1);  
    printf ("i=%d PID=%d\n", i, getpid());  
    exit (i);  
} else {  
    // father  
    sleep (5);  
    pid = wait (&code);  
    printf ("Wait: ret=%d code=%x\n", pid, code);  
}
```

Wait: ret = 3057 code = 200

Esempio 3-B

Comportamento
mascherato
di una **wait**

Disabilito (SIG_IGN) la gestione del segnale SIGCHLD, ricevuto alla terminazione di un figlio

```
signal (SIGCHLD, SIG_IGN) ;
```

```
if (fork() == 0) {  
    // child  
    i = 2;  
    sleep (1);  
    printf ("i=%d PID=%d\n", i, getpid());  
    exit (i);  
} else {  
    // father  
    sleep (5);  
    pid = wait (&code);  
    printf ("Wait: ret=%d code=%x\n", pid, code);  
}
```

PID=3057

Nessuna attesa:
Wait: ret = -1 code = 7FFF

L'esecuzione di una **signal(SIGCHLD,SIG_IGN)** evita che i figli diventino degli zombie mentre una **signal(SIGCHLD,SIG_DFL)** non è sufficiente a tale scopo (anche se SIGCHLD viene ignorato)

Esempio 3-C

Gestione
asincrona del
segnale SIGCHLD

```
static void sigChld (int signo) {
    if (signo == SIGCHLD)
        printf("Received SIGCHLD\n");
    return;
}

...
signal(SIGCHLD, sigChld);
if (fork() == 0) {
    // child
    ...
    exit (i);
} else {
    // father
    ...
}
```

System call kill

```
#include <signal.h>

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

- ❖ Invia il segnale (**sig**) a un processo o a un gruppo di processi (**pid**)
- ❖ Per inviare un segnale a un processo occorre averne diritto
 - Un processo utente può spedire segnali solo a processi con lo stesso UID
 - Il superuser può inviare segnali a tutti i processi

System call kill

❖ Valore di ritorno

- Il valore 0, in caso di successo
- Il valore -1, in caso di errore

❖ Parametri

Se sig=0 si invia un segnale NULL (i.e., non si invia alcun segnale). Spesso usato per capire se un processo esiste: se si riceve -1 dalla kill il processo non esiste

Se pid è	Si invia il segnale sig ...
>0	al processo di PID uguale a pid
==0	a tutti i processi con group id uguale al suo (a cui lo può inviare)
<0	a tutti i processi di group id uguale al valore assoluto di pid (a cui lo può inviare)
==-1	a tutti i processi del sistema (a cui lo può inviare)

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

System call raise

```
#include <signal.h>

int raise (int sig);
```

❖ La system call raise permette a un processo di inviare un segnale a se stesso

➤ La chiamata

- `raise (sig)`

equivale a

- `kill (getpid(), sig);`

System call pause

```
#include <unistd.h>

int pause (void);
```

- ❖ Sospende il processo chiamante sino all'arrivo di un segnale
- ❖ Ritorna solo quando viene eseguito un gestore di segnali e questo termina la sua esecuzione
 - In questo caso la funzione restituisce valore -1

System call alarm

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

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

- ❖ Attiva un timer (ovvero un count-down)
 - Il parametro **seconds** specifica il valore del conteggio in numero di secondi
 - Al termine del count-down viene generato il segnale **SIGALRM**
 - Se SIGALRM non viene catturato o ignorato, l'azione di default è la terminazione del processo

System call alarm

- ❖ Se la system call viene eseguita **prima** della terminazione della chiamata precedente il count-down ricomincia dal nuovo valore
 - In particolare se **seconds** è uguale a 0 (secondi) si disattiva il precedente allarme
- ❖ Valore di ritorno
 - Il valore rimanente del conteggio, nel caso di una ulteriore chiamata
 - Il valore zero, nel caso di terminazione del conteggio

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


System call alarm

❖ Occorre osservare quanto segue

➤ L'allarme è generato dal kernel

- È possibile passi altro tempo prima che il processo riacquisisca il controllo
- Tale tempo è funzione del tempo di "reazione" dello scheduler

➤ Esiste un unico contatore per ciascun processo, inoltre le system call **sleep** e **alarm** utilizzano lo stesso timer

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

Esempio

- ❖ Implementare la system call **sleep** (mySleep) utilizzando le system call **alarm** e **pause**

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

```
static void myAlarm (int signo) {
    return;
}
```

```
unsigned int mySleep (unsigned int nsecs) {
    if (signal(SIGALRM, myAlarm) == SIG_ERR)
        return (nsecs);
```

```
    alarm (nsecs);
    pause ();
    return (alarm(0));
}
```

Il gestore va
istanziato **prima**
di settare l'allarme

Si imposta un allarme
e si va in pausa

Si disattiva il timer e si
ritorna il tempo rimanente

Esempio

- ❖ Implementare la system call **alarm** utilizzando le system call **fork**, **signal**, **kill** e **sleep**

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

void myAlarm (int sig) {
    if (sig==SIGALRM)
        printf ("Alarm on ...\n");
    return;
}
```

Esempio

```
int main (void) {
    pid_t pid;
    signal (SIGALRM, myAlarm);
    pid = fork();
    switch (pid) {
        case -1: /* error */
            printf ("fork failed");
            exit (1);
        case 0: /* child */
            sleep (5);
            kill (getppid(), SIGALRM);
            exit (0);
    }
    /* father */
    ...
    return (0);
}
```

Il figlio attende. Quindi
invia il segnale
SIGALRM

Il padre procede come deve
e riceverà il segnale
SIGALRM dal figlio

Limiti dei segnali

❖ I limiti dei segnali sono di natura variegata

➤ Limiti di memoria

- Nessun tipo di dato viene consegnato con i segnali tradizionali
- La **memoria** dei segnali "pending" è **limitata**

➤ Richiedono **funzioni rientranti** (reentrant function)

➤ Producono **corse critiche** (race conditions)

Limite 1: Memoria

❖ La **memoria** dei segnali "pending" è **limitata**

- Si ha al massimo un segnale "pending" (inviato ma non consegnato) per ciascun tipo di segnale
 - Segnali successivi (dello stesso tipo) sono perduti
- I segnali possono essere bloccati, i.e., si può evitare di riceverli

Most UNIX systems do not queue signals unless support a real-time extension of POSIX

Limite 1: Memoria

...

```
static void sigUshr1 (int signo) {  
    if (signo == SIGUSR1)  
        printf("Received SIGUSR1\n");  
    else  
        printf("Received wrong SIGNAL\n");  
    sleep (5);  
    return;  
}
```

```
static void sigUshr2 (int signo) {  
    if (signo == SIGUSR2)  
        printf("Received SIGUSR2\n");  
    else  
        printf("Received wrong SIGNAL\n");  
    sleep (5);  
    return;  
}
```

Programma con 2
gestori di segnale:
sigUshr1 e **sigUshr2**

Limite 1: Memoria

```
int main (void) {  
    if (signal(SIGUSR1, sigUshr1) == SIG_ERR) {  
        fprintf (stderr, "Signal Handler Error.\n");  
        return (1);  
    }  
    if (signal(SIGUSR2, sigUshr2) == SIG_ERR) {  
        fprintf (stderr, "Signal Handler Error.\n");  
        return (1);  
    }  
    while (1) {  
        fprintf (stdout, "Before pause.\n");  
        pause ();  
        fprintf (stdout, "After pause.\n");  
    }  
    return (0);  
}
```

Il main istanzia i gestori di segnali e itera in attesa di segnali (da shell)

Limite 1: Memoria

Comandi da shell

```
> ./pgrm &  
[3] 2636  
> Before pause.  
> kill -USR1 2636  
> Received SIGUSR1  
After pause.  
Before pause.  
> kill -USR2 2636  
> Received SIGUSR2  
After pause.  
Before pause.
```

Ricezione corretta di un segnale USR1 (-SIGUSR1)

Ricezione corretta di un segnale USR2 (-SIGUSR2)

Osservazione
il comando di shell **kill** invia un segnale a un processo di PID dato

Limite 1: Memoria

L'ordine di consegna dei due segnali è ignoto: USR2 viene consegnato prima

```
> kill -USR1 2636 ; kill -USR2 2636  
> Received SIGUSR2  
Received SIGUSR1  
After pause.  
Before pause.
```

Invio di due segnali contemporanei: USR1 e USR2
Entrambi sono ricevuti

```
> kill -USR1 2636 ; kill -USR2 2636 ; kill -USR1 2636  
> Received SIGUSR1  
Received SIGUSR2  
After pause.  
Before pause.
```

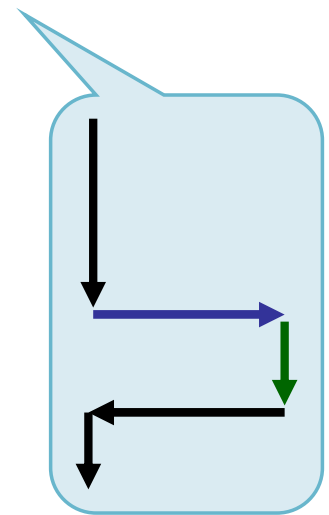
Invio di tre segnali contemporanei: due USR1 e un USR2
Uno dei segnali USR1 viene perduto

```
> kill -9 2636  
[3]+ Killed ./pgrm
```

-9 = SIGKILL = Kill
Kill a process

Limite 2: Funzioni rientranti

- ❖ Il comportamento in presenza di un segnale prevede
 - L'interruzione del flusso di istruzioni corrente
 - L'esecuzione del signal handler
 - Il ritorno al flusso standard alla terminazione del signal handler
- ❖ Quindi
 - Il kernel **sa** da dove riprendere il flusso di istruzioni precedente
 - Il processo **non sa** che cosa il signal handler ha interrotto



Limite 2: Funzioni rientranti

- ❖ Una funzione si dice rientrante (reentrant function) se
 - La sua esecuzione può essere interrotta in qualsiasi punto e la funzione rieseguita senza che questo produca effetti indesiderati
 - In altre parole i risultati e il comportamento della funzione non devono differire rispetto a caso in cui la funzione non venga mai interrotta
- ❖ La rientranza è un concetto importante nella programmazione single-threaded (sequenziale)
 - In particolare, è un concetto rilevante per la corretta implementazione di signal handlers

Limite 2: Funzioni rientranti

- ❖ Non sono rientranti quelle funzioni che
 - Modificano variabili locali statiche (stato interno)
 - Modificano variabili globali
 - Chiamano funzioni non rientranti

- ❖ Esempi

La funzione **myf** produce un output **inatteso** se viene interrotta (da se stessa) dopo l'istruzione **x+=v**

```
int x=0;  
  
int myf (int v) {  
    x+=v;  
    return (x);  
}
```

- Una **malloc** è interrotta da un'altra malloc?
 - La funzione malloc mantiene una lista delle aree di memoria rilasciate e la lista può essere corrotta

Limite 2: Funzioni rientranti

- ❖ La "Single UNIX Specification" definisce le funzioni rientranti (reentrant), ovvero che possono essere interrotte senza problemi
 - `read`, `write`, `sleep`, `wait`
- ❖ Originariamente molte funzioni C non erano rientranti
 - `printf`, `scanf`, etc.
- ❖ Molte di esse hanno però oggi una corrispondente versione rientrante
 - `strtok` → `strtok_r`, `rand` → `rand_r`, etc.

Una chiamata a `printf` può essere interrotta e dare risultati inaspettati

Limite 3: Corse Critiche

- ❖ Nei sistemi concorrenti una corsa critica (o semplicemente corsa) avviene quando
 - Il comportamento (e il risultato finale) di più processi dipende dalla loro temporizzazione e dall'ordine con cui vengono eseguiti
- ❖ La programmazione concorrente è prona a race conditions
- ❖ In particolare l'utilizzo di segnali tra processi può generare race conditions che portano il programma a non funzionare nel modo desiderato

Limite 3: Corse Critiche

- ❖ Si supponga un processo voglia svegliarsi dopo nSec secondi usando alarm e pause
- ❖ Purtroppo
 - Non si possono fare previsioni sull'arrivo di un segnale, e.g., il segnale può arrivare prima che il processo entri in pausa se il sistema è molto carico

```
static void  
myHandler (int signo) {  
    ...  
}  
...  
signal (SIGALARM, myHandler)  
alarm (nSec);  
pause ();
```

Il segnale **SIGALRM** può arrivare prima della pause

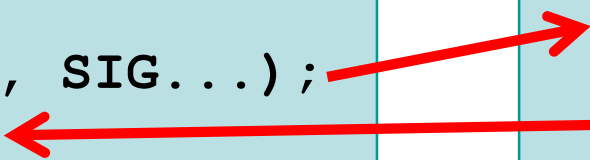
La **pause** blocca il processo per sempre visto che il segnale di allarme è stato perduto

Limite 3: Corse Critiche

- ❖ Si supponga due processi P_1 e P_2 vogliano sincronizzarsi mediante l'utilizzo di segnali
- ❖ Purtroppo
 - Se il segnale di P_1 (P_2) arriva prima che P_2 (P_1) sia entrato in pause
 - Il processo P_2 (P_1) si blocca indefinitamente in attesa di un segnale

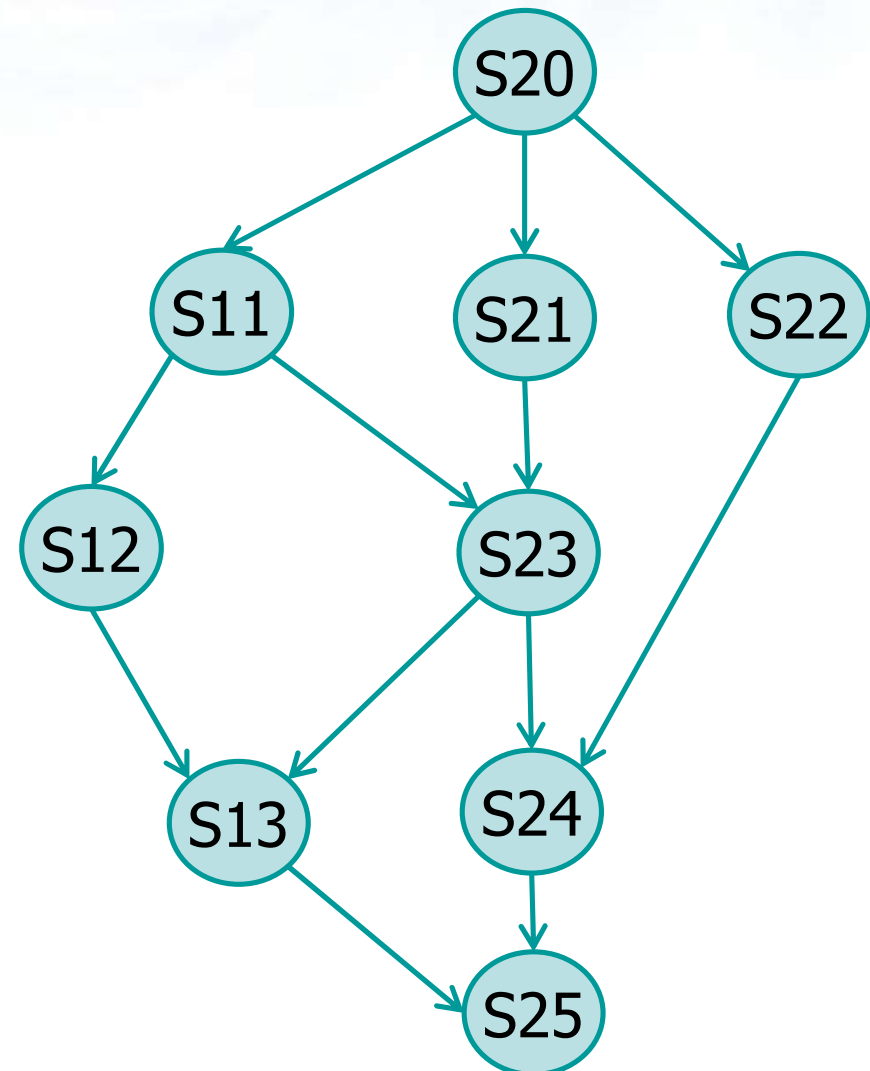
```
P1
while (1) {
    ...
    kill (pidP2, SIG...);
    pause ();
}
```

```
P2
while (1) {
    pause ();
    ...
    kill (pidP1, SIG...);
}
```



Esercizio

- ❖ Nonostante i loro difetti i segnali possono fornire un rozzo meccanismo di sincronizzazione
- ❖ **Ipotizzando di trascurare le corse critiche** (e utilizzando fork, wait, signal, kill, e pause) realizzare il seguente grafo di precedenza



Soluzione

Definizione del gestore di segnali

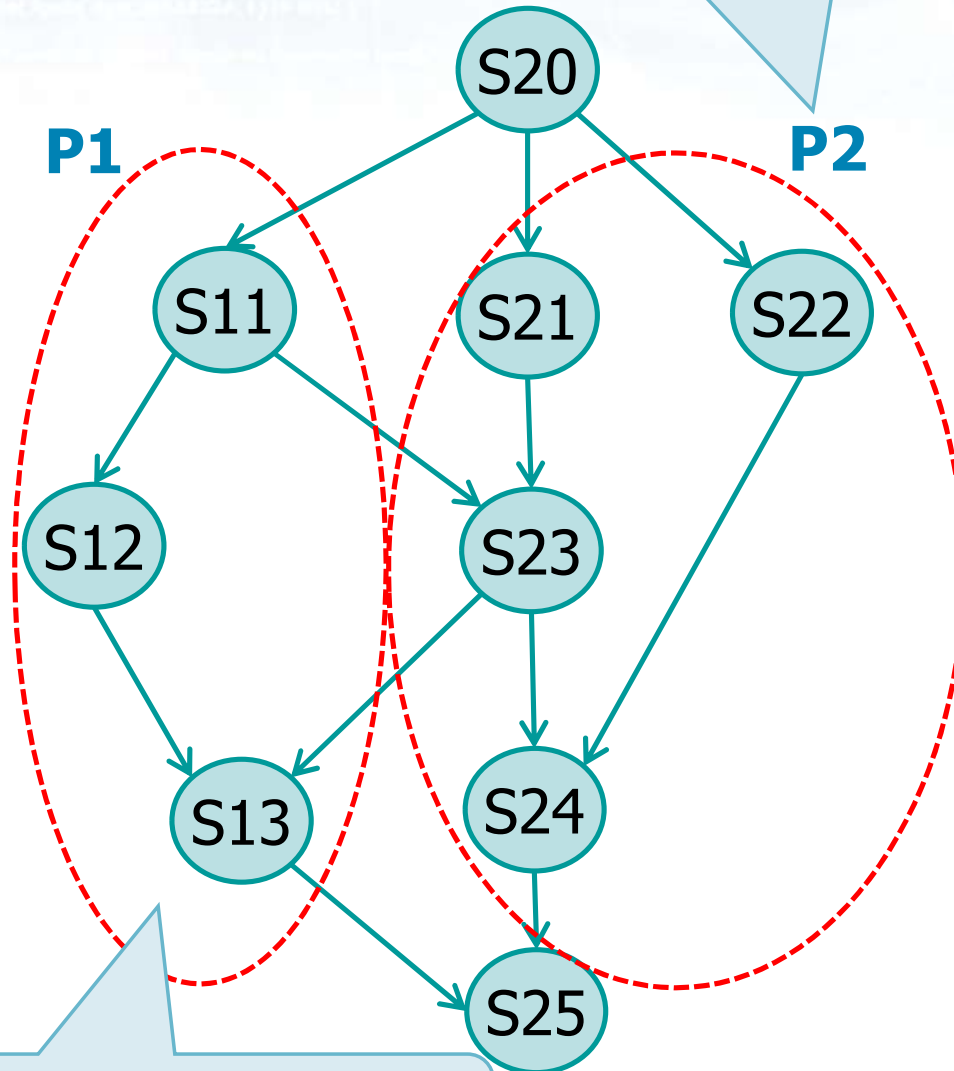
```
static void sigUsr (int signo) {  
    if (signo==SIGUSR1)  
        printf ("SIGUSR1\n");  
    else if (signo==SIGUSR2)  
        printf ("SIGUSR2\n");  
    else  
        printf ("Signal %d\n", signo);  
    return;  
}
```

Instanziazione del gestore di segnali per i segnali SIGUSR1 e SIGUSR2

```
int main (void) {  
    pid_t pid;  
    if (signal(SIGUSR1, sigUsr) == SIG_ERR ||  
        signal(SIGUSR2, sigUsr) == SIG_ERR) {  
        printf ("Signal Handler Error.\n");  
        return (1);  
    }  
}
```

Soluzione

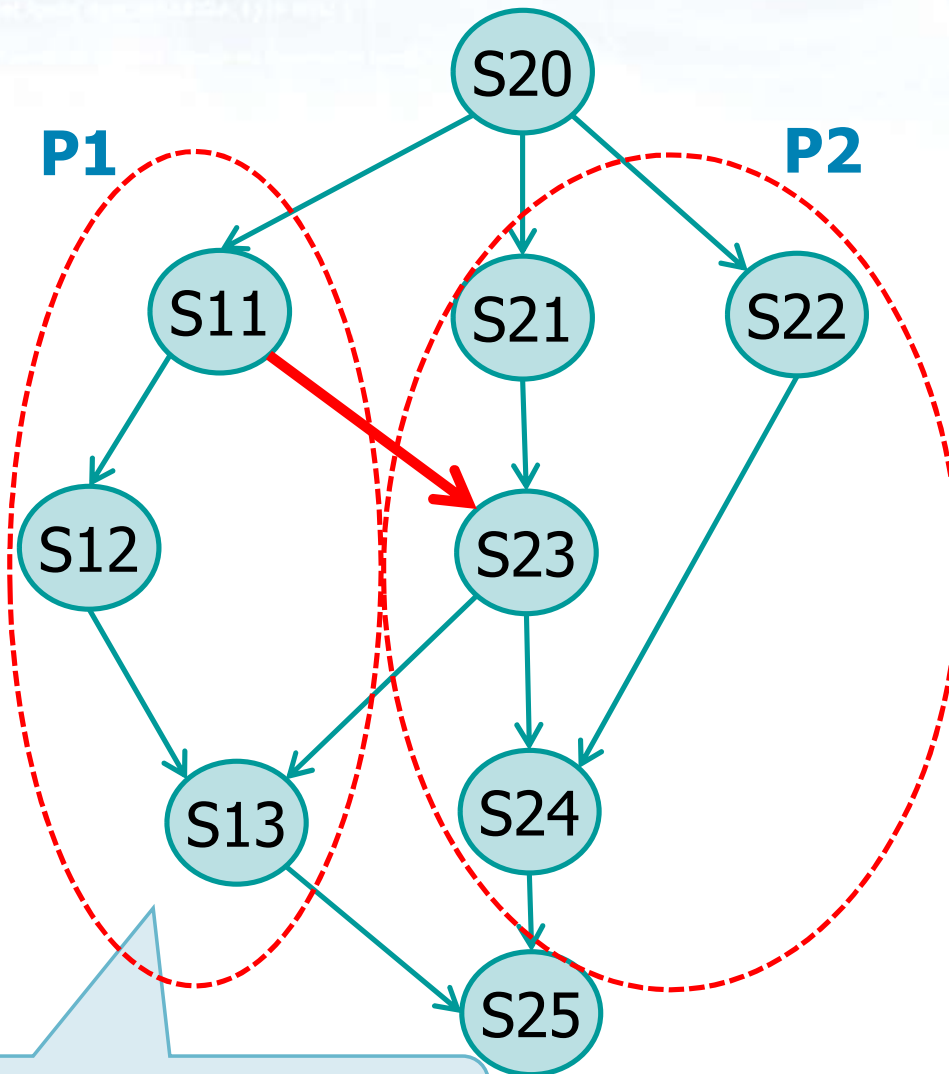
P2 è il figlio può ottenere il pid del padre con getppid



P1 è il padre deve "ricordarsi" il pid del figlio

```
printf ("S20\n");
pid = fork ();
if (pid > (pid_t) 0) {
    P1 (pid);
    wait ((int *) 0);
} else {
    P2 ();
    exit (0);
}
printf ("S25\n");
return (0);
}
```

Soluzione



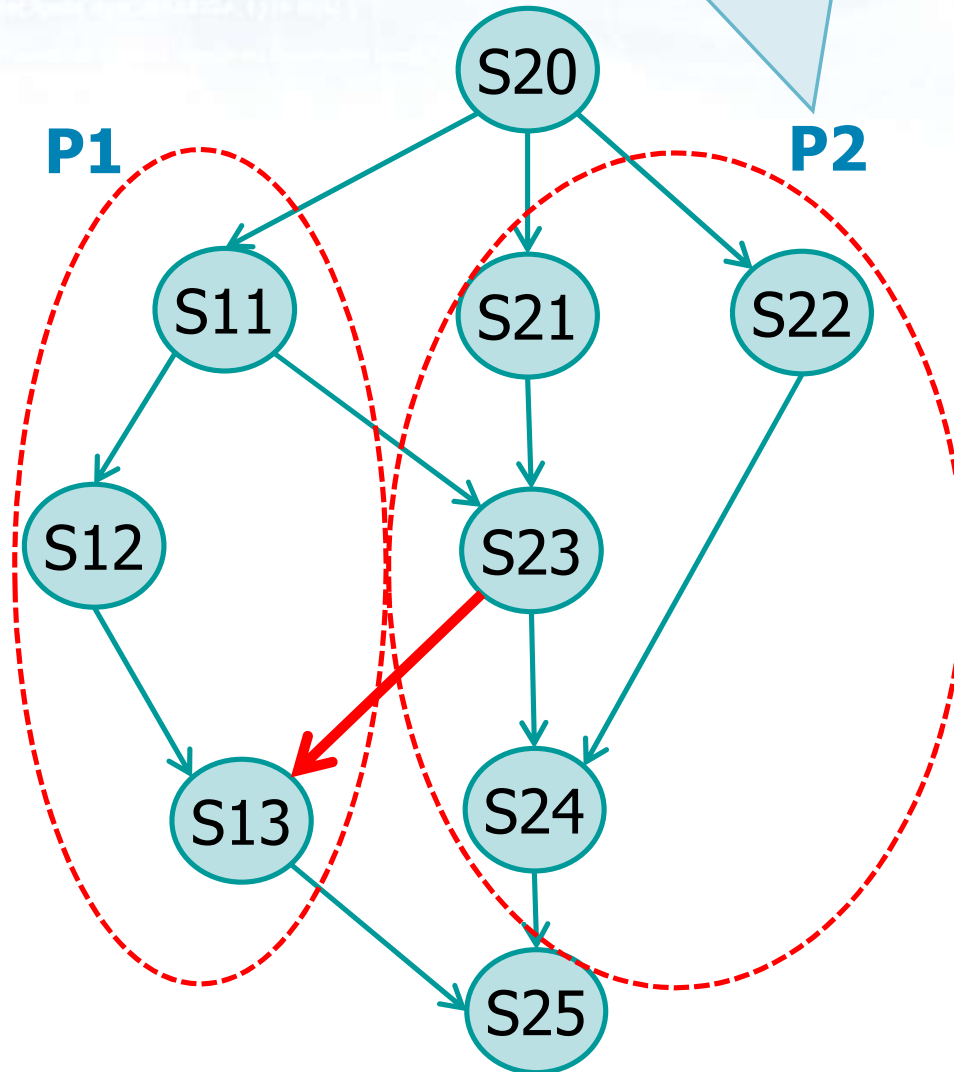
P1 è il padre deve "ricordarsi" il pid del figlio

```
void P1 (
    pid_t cpid
) {
    printf ("S11\n");
    //sleep (1);
    kill (cpid, SIGUSR1);
    printf ("S12\n");
    pause ();
    printf ("S13\n");

    return;
}
```

Soluzione

P2 è il figlio può ottenere il pid del padre con getppid



```
void P2 ( ) {
    if (fork () > 0) {
        printf ("S21\n");
        pause ();
        printf ("S23\n");
        kill (getppid (),
              SIGUSR2);

        wait ((int *) 0);
    } else {
        printf ("S22\n");
        exit (0);
    }
    printf ("S24\n");
    return;
}
```