

# Laboratorio di sistemi operativi – T4

I segnali

# Il programma

1. Introduzione a UNIX
2. Nozioni integrative del linguaggio C
3. controllo dei processi;
4. pipe e fifo;
5. code di messaggi;
6. memoria condivisa;
7. semafori;
- 8. segnali;**
9. introduzione alla programmazione bash

I segnali

# Cause che generano i segnali

- Un segnale è una **notifica a un processo** che è occorso un certo evento. Fra i tipi di eventi che causano il fatto che il kernel generi un segnale per un processo ci sono i seguenti:
  - Èoccorsa una **eccezione hardware**, l'HW ha verificato una condizione di errore che è stata notificata al kernel, il quale a propria volta ha inviato un segnale corrispondente al processo in questione.
    - per esempio, l'esecuzione di istruzioni di linguaggio macchina malformate, divisioni per 0, o riferimenti a parti di memoria inaccessibili.
  - L'utente ha digitato sul terminale dei **caratteri speciali** che generano i segnali.
    - questi caratteri includono il carattere interrupt (normalmente associato a Control-C) e il suspend carattere (Control-Z).

# Cause che generano i segnali

- Un segnale è una notifica a un processo che è occorso un certo evento. Fra i tipi di eventi che causano il fatto che il kernel generi un segnale per un processo ci sono i seguenti:
  - È occorso un **evento software**.
    - per esempio, l'input è divenuto disponibile su un descrittore di file, un timer è arrivato a 0, il tempo di processore per il processo è stato superato, o un figlio del processo è terminato.

# Nomi simbolici e codici

- I segnali sono definiti con interi unici, la cui sequenza inizia da 1. Tali interi sono definiti in `<signal.h>` (o in `<sys/signals.h>`) con nomi simbolici della forma `SIGxxxx`.
- Poiché gli effettivi numeri utilizzati per ogni segnale variano a seconda delle implementazioni, all'interno dei programmi è meglio utilizzare questi nomi.
  - per esempio, quando l'utente digita il carattere dell'interrupt, `SIGINT` (il segnale numero 2) è inviato a un processo.

No	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
4	SIGILL	create core image	illegal instruction
5	SIGTRAP	create core image	trace trap
6	SIGABRT	create core image	abort program (formerly SIGIOT)
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
15	SIGTERM	terminate process	software termination signal
16	SIGURG	discard signal	urgent condition present on socket
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
21	SIGTTIN	stop process	background read attempted from control terminal
22	SIGTTOU	stop process	background write attempted to control terminal
23	SIGIO	discard signal	I/O is possible on a descriptor (see fcntl(2))
24	SIGXCPU	terminate process	cpu time limit exceeded (see setrlimit(2))
25	SIGXFSZ	terminate process	file size limit exceeded (see setrlimit(2))
26	SIGVTALRM	terminate process	virtual time alarm (see setitimer(2))
27	SIGPROF	terminate process	profiling timer alarm (see setitimer(2))
28	SIGWINCH	discard signal	Window size change
29	SIGINFO	discard signal	status request from keyboard
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

man 7 signal

	<b>Function</b>	<b>Character</b>
<b>intr</b>	Terminates the current job.	<code>^c</code> (CTRL-c)
<b>quit</b>	Terminates the current job; makes a core file.  <code>ulimit -c unlimited</code>	<code>^\</code> (CTRL-\)
<b>susp</b>	Stops the current job (so you can put it in the background).	<code>^z</code> (CTRL-z)

# Ciclo di vita dei segnali

- Si dice che un segnale è **generato** da qualche evento.
- Dopo essere stato generato, un segnale è **inviato** (delivered: propriamente, consegnato) ad un processo, che quindi esegue una qualche azione in risposta al segnale.
- Fra il momento in cui è generato e il momento in cui è inviato al processo, il segnale è **pendente** (pending).

# Pending, delivery e block

- Di norma, un segnale pendente è inviato a un processo appena il processo è scelto per l'esecuzione, oppure immediatamente se il processo è già in esecuzione (per esempio, nel caso in cui il processo invia un segnale a se stesso).

# Pending, delivery e block

- A volte invece è necessario assicurare che un segmento di codice non sia interrotto dalla consegna di un segnale.
  - in questo caso, possiamo aggiungere un segnale alla **maschera dei segnali** del processo, cioè un insieme di segnali la cui ricezione è attualmente bloccata.
  - Se un segnale è generato mentre il processo è bloccato, il segnale rimane pendente fino a quando non viene successivamente sbloccato e rimosso dalla maschera dei segnali. Varie system call permettono ai processi di aggiungere e rimuovere segnali dalla propria maschera dei segnali.

# Delivery and process actions

- Al momento della ricezione di un segnale, un processo continua con una delle seguenti azioni di default, a seconda del segnale:
  - Il segnale è **ignorato**; in questo caso viene scartato dal kernel e non ha effetti sul processo (il processo non è neppure informato del fatto che quel segnale è occorso).
  - Il processo viene **terminato** (killed). Questa terminazione è detta anche abnormal process termination, opposta alla terminazione normale del processo, che occorre quando un processo termina usando exit().

# Delivery and process actions

- È generato un file contenente un **core dump** file, e il processo viene **terminato**.
  - Un file con core dump contiene un'immagine della memoria virtuale del processo; tale immagine può essere caricata in un debugger per ispezionare lo stato del processo al momento della terminazione.
- Il processo viene **bloccato** (stopped): l'esecuzione è in questo caso sospesa.
- L'esecuzione del processo è **ripresa** (resumed) dopo essere stata bloccata in precedenza.

# Il sistema di segnali in Unix: le trap

- Una classe di segnali sono le trap: segnali generati da **eventi prodotti da un processo e inviati al processo stesso**.
- Alcune trap sono causate da comportamenti errati del processo stesso, e immediatamente inviate al processo che normalmente reagisce terminando.
  - per esempio tentativi di divisione per zero (SIGFPE), indirizzamento errato degli array (SIGSEGV), tentativo di eseguire istruzioni privilegiate (SIGILL), etc..

# Il sistema di segnali in Unix: gli interrupt

- Gli interrupt sono segnali inviati ad un processo da un **agente esterno**: l'utente o un altro processo
- Utente:
  - CTRL-C (invia SIGINT)
  - CTRL-Z (invia SIGSTOP)
  - Comando kill: `kill -s SIGNAL PID`
- Altro processo:
  - System call kill: `kill(PID, SIGNAL)`

# Impostare l'handler del segnale

- Invece di accettare l'azione di default per un particolare segnale, un programma può modificare l'azione da intraprendere al momento della consegna (delivery) del segnale. Questo è noto come impostazione dell'handler del segnale. Un programma può impostare una dei seguenti handler del segnale:
  - L'azione di default dovrebbe essere intrapresa. Utile per cancellare precedenti modifiche della disposizione del segnale che modificavano la disposizione di default.
  - Il segnale è ignorato. Utile per un segnale la cui disposizione sarebbe quella di terminare il processo.
  - Viene eseguito un signal handler.

# Signal handlers

- Un signal handler (o gestore di segnali) è una funzione, scritta dal programmatore, che esegue azioni appropriate in risposta alla ricezione di un segnale.
  - Per esempio, la shell ha un gestore per il segnale SIGINT (generato dal carattere interrupt, Control-C) che causa il suo blocco (stop) e la restituzione del controllo al ciclo di input principale: in questo caso all'utente viene presentato il prompt della shell.
  - La notifica al kernel del fatto che deve essere invocata una funzione handler è detto installare o stabilire un signal handler.
  - Quando un handler è invocato in risposta alla ricezione di un segnale, diciamo che il segnale è stato gestito (handled) o intercettato (caught).

6	SIGABRT	create core image	abort program (formerly SIGIOT)
14	SIGALRM	terminate process	real-time timer expired

- **SIGABRT.** Un processo riceve questo segnale quando invoca la funzione `abort()`. Di default questo segnale termina il processo con un core dump. Questo produce l'effetto della chiamata `abort()`, che produce un core dump a fini di debug.
- **SIGALRM.** Il kernel genera questo segnale al momento del raggiungimento dello zero di un timer impostato da una chiamata ad `alarm()` o `setitimer()`.

20	SIGCHLD	discard signal	child status has changed
19	SIGCONT	discard signal	continue after stop

- SIGCHLD. Segnale inviato dal kernel a un processo genitore quando uno dei figli termina (chiamando exit(), o ucciso da un qualche segnale). Può essere inviato a un processo quando uno dei suoi figli è bloccato o risvegliato da un segnale.
- SIGCONT. Quando viene inviato a un processo bloccato (stopped), questo segnale causa il risveglio del processo (resume), cioè che il processo venga schedulato per successivamente essere eseguito. Quando è ricevuto da un processo che non è bloccato, questo segnale è ignorato di default. Un processo può intercettare questo segnale, in modo da eseguire qualche azione particolare al momento della ripresa dell'esecuzione.

2	SIGINT	terminate process	interrupt program
9	SIGKILL	terminate process	kill program
13	SIGPIPE	terminate process	write on a pipe with no reader

- SIGINT. Quando l'utente digita il carattere di interrupt (Control-C), il terminale invia questo segnale al gruppo del processo in foreground. L'azione di default per questo segnale è terminare il processo.
- SIGKILL. È il segnale sicuro di kill. **Non può essere bloccato, ignorato, o intercettato da un handler**, e quindi termina sempre un processo.
- SIGPIPE. Segnale generato quando un processo tenta di scrivere su un pipe o un FIFO per il quale non c'è un corrispondente processo lettore. Questo normalmente occorre perché il processo lettore ha chiuso il proprio file descriptor per il canale IPC.

## 11 SIGSEGV create core image segmentation violation

- **SIGSEGV.** Segnale generato quando un programma tenta un riferimento in memoria non valido. Il riferimento può non essere valido perché la pagina riferita non esiste (per esempio, giace in un'area non mappata, fra heap e stack), oppure il processo ha tentato di modificare una locazione in read-only memory (il segmento di testo del programma o una regione di memoria marcati come disponibili in sola lettura), o il processo ha tentato di accedere a una parte della memoria del kernel durante l'esecuzione in user mode.
  - In C, questi eventi spesso derivano dalla dereferenziazione di un puntatore che contiene un 'bad address' (come un puntatore non inizializzato) o dal passaggio di un argomento non valido in una chiamata a funzione.
  - Il nome del segnale deriva da segmentation violation.

17	SIGSTOP	stop process	stop (cannot be caught or ignored)
15	SIGTERM	terminate process	software termination signal

- SIGSTOP. Segnale per il blocco (stop) sicuro. Non può essere bloccato, ignorato, o intercettato da un handler; quindi questo segnale blocca sempre un processo.
- SIGTERM. Segnale standard utilizzato per terminare un processo; segnale inviato di default dai comandi kill e killall. Gli utenti a volte inviano esplicitamente il segnale SIGKILL a un processo, usando kill –KILL or kill –9.
  - in generale, questo è un errore. Un'applicazione ben progettata deve avere un handler per SIGTERM che consenta una 'graceful' exit, che consenta di pulire i file temporanei e di rilasciare le altre risorse. L'uccisione di un processo con SIGKILL bypassa l'handler di SIGTERM, e quindi si dovrebbe sempre prima cercare di terminare un processo con SIGTERM, e tenere SIGKILL come ultima scelta per terminare i processi che non rispondono a SIGTERM.

5	SIGTRAP	create core image	trace trap
18	SIGTSTP	stop process	stop signal generated from

- SIGTRAP. Segnale utilizzato per implementare i breakpoint in fase di debugging e per la tracciatura delle system call.
  - Cercare ptrace() sul manuale per ulteriori informazioni.
- SIGTSTP. Segnale per lo stop, inviato per bloccare il gruppo di processi in foreground quando l'utente digita il carattere di sospensione (Control-Z) sulla tastiera.
  - il nome di questo segnale deriva da "terminal stop".

30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

- SIGUSR1. Questo segnale e SIGUSR2 sono disponibili per fini specificati dal programmatore. Il kernel non genera mai questi segnali per un processo.
  - I processi possono utilizzare questi segnali per notificarsi a vicenda eventi, o per sincronizzarsi

# Invio di segnali

`kill()`, `raise()` e `alarm()`

# Inviare un segnale da linea di comando

- Si usa il comando `kill`
  - `kill -INT <PID>`
  - `kill -SIGINT <PID>`
  - `kill -2 <PID>`
- Se il segnale non è specificato, allora viene inviato il segnale SIGTERM
- **Evitate** di provare a eseguire....
  - `sudo kill -9 -1`

# La system call kill()

```
#include <signal.h>

int kill(pid_t pid, int sig) ;

//Returns 0 on success, or -1 on error
```

- L'argomento pid identifica uno o più processi a cui inviare il segnale; pid può essere interpretato in 4 modi:
  - pid > 0: il segnale è inviato al processo identificato da pid.
  - pid == 0: il segnale è inviato a ogni processo nello stesso gruppo del chiamante, chiamante incluso.
  - pid < -1: il segnale è inviato a tutti i processi nel gruppo del processo il cui ID è uguale al valore assoluto di pid. Inviare un segnale a tutti i processi nel gruppo di un processo è utile nel controllo dei job effettuato con la shell.
  - pid == -1: (broadcast signal) il segnale è inviato a tutti i processi per i quali il processo ha i permessi di inviare un segnale, eccetto init (che ha pid 1) ed il chiamante. Se l'utente non è super user, il segnale è inviato a tutti i processi con stesso uid dell'utente, escluso il processo che invia il segnale.

```
#include <signal.h>

int kill(pid_t pid, int sig) ;

//Returns 0 on success, or -1 on error
```

- Se nessun processo corrisponde al pid predefinito, kill() fallisce e setta errno a ESRCH (“No such process”)
- Se il processo esiste, ma non si hanno i permessi per inviare un segnale, allora errno = EPERM
- Verifica dell'esistenza di un processo. Se l'argomento sig è settato a 0 (detto null signal), non è inviato alcun segnale.
  - In questo caso kill() esegue unicamente un controllo degli errori per vedere se è possibile inviare segnali al processo: il null signal può essere utilizzato per testare se un processo con un certo pid esiste.
  - Se la chiamata va a buon fine, sappiamo che il processo esiste.

# La system call raise()

- Permette di inviare un segnale al processo stesso che invoca la system call

```
#include <signal.h>

int raise(int sig) ;

//Returns 0 on success, or nonzero on error
```

# La system call alarm()

- Permette di inviare un segnale SIGALRM al processo stesso che invoca la system call dopo un numero di secondi

```
#include <unistd.h>

unsigned int alarm (unsigned int sec);
```

- Valore di ritorno:
  - 0 se non ci sono precedenti invocazioni di alarm pendenti
  - il numero di secondi che mancano alla scadenza dell'allarme precedente

# Cambiare l'handler dei segnali

System call signal()

# signal() e sigaction()

- I sistemi UNIX forniscono due modi per cambiare l'handler di un segnale: signal() e sigaction().
  - La system call signal() è l'API originale per assegnare l'handler di un signal, e fornisce un'interfaccia più semplice di sigaction().
- NB: vi sono differenze nel comportamento di signal() fra le varie implementazioni di UNIX. → da evitare!

```
SIGNAL(2)                               Linux Programmer's Manual      SIGNAL(2)

NAME
    signal - ANSI C signal handling

SYNOPSIS
    #include <signal.h>

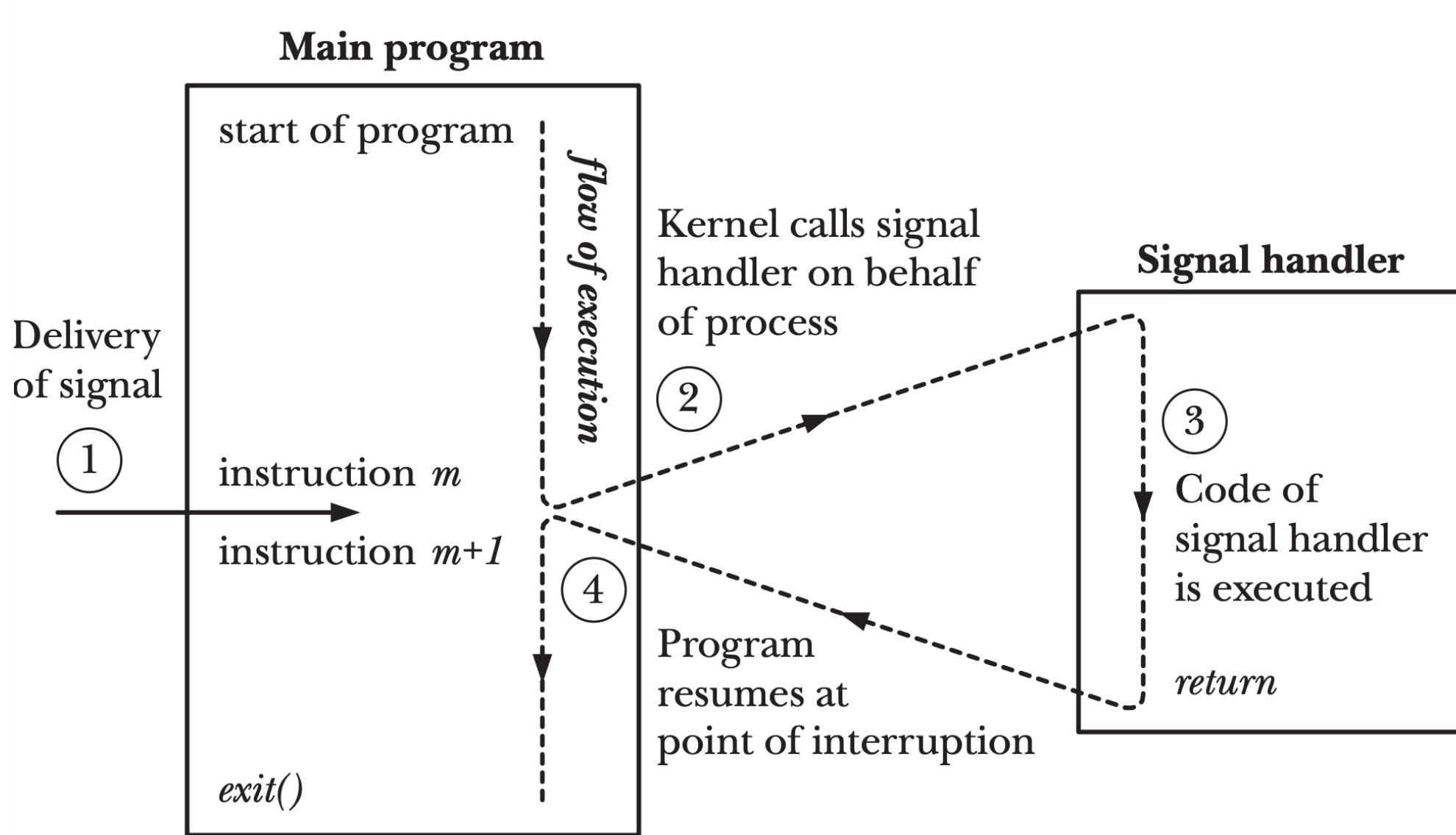
    typedef void (*sighandler_t)(int);

    sighandler_t signal(int signum, sighandler_t handler);

DESCRIPTION
WARNING:
    the behavior of signal() varies across UNIX versions, and has also varied historically across
    different versions of Linux. Avoid its use: use sigaction(2) instead. See Portability below.
```

# Signal handlers

- Un signal handler è una funzione chiamata quando un processo riceve uno specifico segnale.
- L'invocazione di un handler può interrompere il flusso principale del programma in qualsiasi momento;
  - il kernel chiama l'handler da parte del processo, e
  - quando l'handler restituisce, l'esecuzione del programma riprende dal punto in cui l'handler lo aveva interrotto.



sigaction

# Gestione dei segnali con sigaction

- Gestione dei segnali più completa e robusta rispetto all'uso di signal () (che tra l'altro non va usata)
- Parametri
  - signum: segnale che deve essere gestito
  - act: nuovo handler per il segnale (se NULL, nessun cambiamento)
  - oldact: puntatore al vecchio handler

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

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

# Gestione dei segnali con sigaction

```
struct sigaction new, old;  
  
// set new handler to new  
sigaction(signum, new, NULL);  
  
// current handler in old  
sigaction(signum, NULL, old);  
  
// do both  
sigaction(signum, new, old);
```

# Sigaction structure

- Principali elementi
  - sa\_handler: puntatore alla funzione di handling
  - sa\_mask: maschera per indicare i segnali bloccati **durante l'esecuzione dell'handler**
  - sa\_flags: maschera bitwise (or-style) che modifica il comportamento del gestore del segnale

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

struct sigaction {
    void (*sa_handler ) (int signum ) ;
    sigset_t sa_mask ;
    int sa_flags ;
    // plus others (for advanced users )
};
```

# Gestire più segnali con un unico handler

- Il costrutto switch può essere utilizzato per discriminare il tipo di segnale che l'handler ottiene come argomento

```
void handle_signal(int signum) {
    /* signal signum triggered the handler */
    switch (signum) {
        case SIGINT:
            /* handle SIGINT */
            break;
        case SIGALRM:
            /* handle SIGALRM */
            break;
        /* other signals */
    }
}
```

# Handler definiti dall’utente

- Le funzioni di gestione degli handler vengono ereditate dai processi figli dopo una fork
- Le variabili globali definite nel programma sono visibili sia dalle funzioni handler che dal resto del programma
  - Questo può essere utile per modificare il valore di una variabile globale simulando un cambio di stato del programma che verrà poi utilizzato durante l’esecuzione “normale” del codice
  - Ma....

# Handler definiti dall'utente e variabili globali

- Le variabili globali possono essere utili, ma cosa succede quando un handler modifica il valore di una variabile globale il cui valore non dovrebbe essere modificato?
- Alcune system call utilizzano strutture dati globali e quindi il loro utilizzo all'interno di un handler può generare problemi
  - Esempio: printf
    - Cosa succede se un segnale interrompe l'esecuzione di una printf, che poi viene anche utilizzata all'interno dell'handler?
    - man signal-safety

# Handler definiti dall'utente e variabili globali

- Fortunatamente ci sono funzioni che sono definite in libc **"AS-Safe"** (Asynchronous Signal-Safe)
  - Esempio: `write()`
- Consiglio: ogni volta che si vuole utilizzare una system call in un hadler, controllare se è AS-safe nella documentazione
- `errno` è una variabile globale e il suo valore potrebbe essere sovrascritto durante l'esecuzione di un handler
  - Consiglio: salvare e ripristinare il valore di `errno` nell'handler

# sigaction structure: signal mask

- Quando un segnale signum è consegnato ad un processo, durante l'esecuzione del suo handler, il segnale signum è bloccato
  - A meno che il flag SA\_NODEFER è settato nella sigaction

```
struct sigaction sa;
sigset_t my_mask;

sa.sa_handler = handle_signal;
sa.sa_flags = SA_NODEFER; // allow nested invocations
sigemptyset(&my_mask); // do not mask any signal
sa.sa_mask = my_mask;
sigaction(SIGUSR1, &sa, NULL); // set the handler
```

# Signal mask

man sigsetops

- La maschera è la collezione di segnali attualmente blocked
  - Ogni processo ha la propria maschera di segnali: un nuovo processo eredita la maschera del genitore
- Le maschere sono di tipo `sigset_t`
  - Le funzioni per la manipolazione dei set sono (man `sigsetops` per dettagli):

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

# Impostare la maschera dei segnali di un processo

```
#include <signal.h>
int sigprocmask(int how,
                 const sigset_t *set,
                 sigset_t *oldset);
```

Returns 0 if successful; otherwise the value -1 is returned and the global variable errno is set.

- per impostare la maschera di segnali bloccati durante l'esecuzione dei processi si usa la system call `sigprocmask()`
- l'argomento `how` può assumere i seguenti valori:
  - `SIG_BLOCK`: i segnali nel set sono aggiunti a quelli bloccati;
  - `SIG_UNBLOCK`: i segnali nel set sono rimossi dalla maschera esistente;
  - `SIG_SETMASK`: il set diventa la nuova maschera del segnale.
- `oldset` è la vecchia maschera

```
static int segnale_ricevuto = 0;
int main (int argc, char *argv[]) {
    sigset_t orig_mask;
    sigset_t mask;
    struct sigaction act;

    printf("process pid: %d\n", getpid());
    memset (&act, 0, sizeof(act));
    act.sa_handler = mio_handler;

    if (sigaction(SIGTERM, &act, 0))
        ; // gestione errore
    sigemptyset (&mask);
    sigaddset (&mask, SIGTERM);

    if (sigprocmask(SIG_BLOCK, &mask, &orig_mask) < 0)
        ; // gestione errore
    sleep (20);
    if (sigprocmask(SIG_SETMASK, &orig_mask, NULL) < 0)
        ; // gestione errore
    sleep (1);
    if (segnale_ricevuto) puts ("signal ricevuto!!!!");
    return 0;
}
```

# Signal mask during a handler

- L'handler può comunque essere interrotto da altri segnali (o dallo stesso segnale, nel caso `SA_NODEFER` sia settato)
  - quando l'handler termina, l'insieme di segnali bloccati è reimpostato al suo valore precedente la sua esecuzione, indipendentemente dalle possibili manipolazioni dei segnali bloccati eventualmente presenti nell'handler

# Signal mask during a handler

- L'handler può comunque essere interrotto da altri segnali (o dallo stesso segnale, nel caso `SA_NODEFER` sia settato)

```
struct sigaction sa;
sigset_t my_mask;

sa.sa_handler = &handle_signal; // handler
sa.sa_flags = 0; // No special behaviour

// Create an empty mask
sigemptyset(&my_mask); // Do not mask any signal
sigaddset(&my_mask, signal_to_mask_in_handler);
sa.sa_mask = my_mask;

sigaction(SIGINT, &sa, NULL); // Set the handler
```

# Merged signals

- se un segnale viene generato mentre un altro segnale (stesso segnale) è in stato pending, il nuovo segnale e quello pending sono accorpati in uno;
- la presenza di un segnale pending è gestita solo con un flag, non da un numero (di segnali in stato pending)
  - un gestore di segnali non può essere utilizzato per contare il numero di segnali ricevuti

# Delivery di segnali a processi sospesi

- pause(), sleep() e nanosleep()

```
#include <unistd.h>
int pause (void);
```

```
#include <unistd.h>
unsigned int sleep (unsigned int seconds);
```

```
#include <time.h>
struct timespec my_time;
my_time.tv_sec = 1;
my_time.tv_nsec = 234567000;
nanosleep (&my_time, NULL);
```

# Delivery di segnali a processi sospesi

- all'arrivo (asincrono) di un segnale
  1. Lo stato del processo è salvato (registers, etc.)
  2. La funzione di handler è eseguita
  3. Lo stato iniziale del processo è ripristinato
- Quando un processo è in attesa su `wait()` (o altre system call), o sospeso con `pause()` o `sleep()`:
  1. il processo non è in esecuzione (è sospeso su qualche system call)
  2. la funzione dell'handler è eseguita normalmente
  3. quando l'handler ritorna:
    - A. la syscall restituisce un errore, con `errno` settato a `EINTR`; o
    - B. la syscall viene automaticamente ripresa.
- A o B? dipende dal sistema operativo, e dal flag `SA_RESTART` nella syscall `sigaction()`.
  - Il comportamento di default è A (aborting)
  - Se il flag `SA_RESTART` è settato in `sa_flags`, si ha invece un restart della system call che aveva generato l'attesa
    - Il comportamento dipende dalla system call, purtroppo
    - Nella sezione «*Interruption of system calls and library functions by signal handlers*» di «`man 7 signal`» ci sono le indicazioni puntuali

`man 7 signal`

# Gestione sincrona dei segnali

- Non utilizzata durante il corso
- La gestione sincrona dei segnali è possibile
  - Un processo può attendere la ricezione di un particolare segnale
- se interessati, approfondire le system call
  - `sigwaitinfo()`, `sigtimedwait()`, `sigwait()`

# Esercizio

- Scrivere un programma che realizzi un semplice gioco. Il programma seleziona un numero intero casuale tra 0 e argv[1] (il primo argomento passato a riga di comando), e l'utente deve indovinare questo numero. Per fare questo, viene realizzato un ciclo in cui il programma legge da tastiera un numero inserito dall'utente:
  - se il numero è stato indovinato, il gioco finisce;
  - se il numero è maggiore o minore di quello estratto casualmente, viene stampato a video la scritta “maggiore” o “minore”, rispettivamente.
- Se il giocatore non indovina entro argv[2] secondi (da realizzare con alarm e gestendo il segnale SIGALRM), il programma stampa a video “tempo scaduto”, ed esce.