Interprocess communication: I Segnali

- I segnali in Unix sono un meccanismo semplice per inviare degli interrupt software ai processi
- Realizzano una forma di comunicazione essenzialmente asincrona
- Solitamente sono sfruttati per gestione di situazioni d'errori o di condizioni "anomale", piuttosto che per trasmettere informazioni o dati complessi

Interprocess communication: I Segnali

- SPEDIZIONE: un processo (utente o, più solitamente, di sistema) può inviare un segnale a un processo (o a un gruppo di processi)
- RICEZIONE: i segnali possono essere ricevuti e poi gestiti o ignorati, oppure possono venir bloccati, in tal caso restano pendenti
- Il processo destinatario di un segnale può compiere diverse azioni:
  1. eseguire una azione di default (ogni segnale ne ha una predefinita)
  2. eseguire una precisa funzione per gestire il segnale: signal handling
     (non sempre è possibile)
  3. ignorare il segnale (non sempre è possibile)
- la gestione di un segnale può essere decisa dal processo stesso, tramite l'"installazione di uno handler del segnale"

Interprocess communication: I Segnali

- I segnali, definiti nel file `signal.h` sono interi positivi, riferiti tramite delle costanti simboliche
- I segnali sono un meccanismo di "basso livello". Vi possono essere differenze nel modo in cui diverse versioni di Unix e Linux implementano alcuni aspetti relativi alla gestione dei segnali (diverse versioni di kernel di linux possono non adottare le stesse soluzioni o convenzioni)
- Il comando `man 7 signal` offre molte informazioni sui segnali utilizzabili in un particolare sistema. Ecco un estratto:

```
Signal     Value     Action   Comment
------------------------------------------------------
SIGHUP        1       Term    Hangup detected
SIGINT        2       Term    Interrupt from keyboard
SIGQUIT       3       Core    Quit from keyboard
  ...
SIGALRM      14       Term    Timer signal from alarm(2)
SIGTERM      15       Term    Termination signal
SIGUSR1   30,10,16    Term    User-defined signal 1
SIGUSR2   31,12,17    Term    User-defined signal 2
SIGCHLD   20,17,18    Ign     Child stopped or terminated
  ...
```

I Segnali: azione di default

Nella maggior parte dei segnali la azione di default consiste nella terminazione del processo ricevente:

- SIGABRT, SIGBUS, SIGSEGV, SIGQUIT, SIGILL, SIGTRAP (ed altri) terminano il processo e generano un file contenente la "core-image" del processo
- segnali come SIGINT, SIGKILL, SIGUSR1 (ed altri) provocano la terminazione del processo senza generare il core
- SIGSTOP blocca il processo che lo riceve (altri segnali hanno lo stesso effetto).
  Il processo passa nello stato suspended
- SIGCONT riavvia il processo che si trova nello stato suspended (ovvero, che ha precedentemente ricevuto un segnale come SIGSTOP)
- alcuni segnali, come SIGCHLD, sono ignorati (azione di default: `Ign`)

Qualcosa di più sulla system call `wait()` (1)

Quando un processo termina, diventa zombie e il suo return value viene scritto nella process-table e il segnale SIGCHLD viene inviato al padre.

Quando un processo invoca `wait()` si eseguono questi tre passi:

1. se non vi sono processi figli restituisce un codice d'errore
2. se vi sono figli in stato zombie, uno di essi viene eliminato dalla process-table e vengono restituiti il suo PID e il suo return value
3. se esistono processi figli ma nessuno di essi è zombie, il processo chiamante viene sospeso fino alla ricezione di un qualsiasi segnale, quindi si riprende dal punto 1.

L'azione di default per SIGCHLD è "ignore". Ciò comporta che gli zombie restino tali fino alla terminazione del padre

Se invece un processo decide di ignorare SIGCHLD (imposta esplicitamente l'azione SIG_IGN) allora alla ricezione di SIGCHLD (ignorato), il kernel elimina gli zombie esistenti dalla process-table.

In tal caso, al ritorno dalla `wait()` il padre torna suspended. Se invece non ci sono più figli `wait()` restituisce un error code.

Qualcosa di più sulla system call `wait()` (2)

La system call ha `wait()` il seguente prototipo:

```
pid_t wait(int *status)
```

`wait()` restituisce il PID del figlio terminato (o -1 se non esistono figli) e assegna a `*status` un valore intero da interpretare come segue:

- Se il *rightmost* byte di `*status` è 0,
  il *leftmost* byte contiene gli 8 bit meno significativi del valore restituito dal figlio con `exit()` o `return()`
- Se il *rightmost* byte di `*status` non è 0,
  i suoi 7 bit meno significativi sono il codice del segnale che ha causato la terminazione del figlio;
  l'altro bit è 1 se è stato generato un core dump

Qualcosa di più sulla system call `wait()` (3)

Un processo suspended per l'esecuzione di una `wait()` viene riattivato

- dalla ricezione di un qualsiasi segnale, oppure
- da qualsiasi cambio di stato di uno dei suoi processi figli

Ad esempio: cambi di stato si hanno quando:

- il figlio passa allo stato zombie
- il figlio riceve SIGSTOP o SIGTSTP: il figlio passa allo stato suspended
- il figlio riceve SIGCONT: il figlio lascia lo stato suspended

Processi e gruppi di processi (1)

- Ogni processo appartiene a un process group (es. la `fork()` crea un figlio nello stesso gruppo del padre (anche la `exec()` non varia il gruppo)
- Un processo può cambiare gruppo con `setpgid()`
- Ogni processo ha associato un terminal (di solito quello in cui è stato avviato)
- Ogni terminal ha associato un control process.
- Digitando `Ctrl-C` in un terminal SIGINT viene inviato a tutti i processi del gruppo del control process del terminal

Un esempio:

```
ls -al | grep .txt | more
```

genera tre processi appartenenti allo stesso process group

Processi e gruppi di processi (2)

- Quando una shell inizia l'esecuzione è il process control di un terminal
- se in una shell eseguo un `<comando>` in foreground la shell figlia cambia il suo gruppo prima di eseguire `exec(<comando>)` e prende il controllo del terminal

  Ogni segnale generato dal terminal viene recapitato alla shell figlia
- se in una shell eseguo un `<comando>` in background la shell figlia cambia il suo gruppo prima di eseguire `exec(<comando>)` ma NON prende il controllo del terminal

  Se il `<comando>` cerca di leggere dallo `stdin`, non è membro dello stesso gruppo del controllore del terminal, quindi riceve SIGTTIN (che lo sospende)

  Ogni segnale generato dal terminal viene recapitato alla shell madre

Invio di un segnale da comando bash

```
kill  -s signal pid
```

dove `pid` può essere

- *n* il segnale viene inviato al processo con PID=*n*
- 0 il segnale viene inviato a tutti i processi del gruppo corrente
- -1 il segnale viene inviato a tutti i processi con PID>1 (limitatamente a quelli dell'utente)
- *-n* (con $n > 1$) il segnale viene inviato a tutti i processi del gruppo *n*

```
kill -l
```

elenca i codici dei segnali

Invio di un segnale da programma

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

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

con argomenti simili al comando bash `kill` visto prima

Un'altra possibilità:

```
int raise(int sig);
```

Esercizio 1: scoprire cosa fa `raise()`

Esercizio 2: scoprire cosa fa `alarm()`

Esercizio 3: scoprire cosa fa `pause()`

Segnali e terminazione: un esercizio

Consultando il manuale `man 2 wait`, spiegare cosa fa il seguente programma:

```
int main(int argc, char *argv[]) {
  pid_t cpid, w;
  int status;

  cpid = fork();
  if (cpid == -1) {
      perror("fork");
      exit(EXIT_FAILURE);
  }
  if (cpid == 0) {    /* Code executed by child */
      printf("Child PID is %ld\n", (long) getpid());
      if (argc == 1)
          pause();     /* Wait for signals */
      exit(atoi(argv[1]));
```

...(continua)

Segnali e terminazione: un esercizio

...

```
} else { /* Code executed by parent */
    do { /* use "kill -s SIGSTOP pid" (or SIGKILL,SIGCONT,...)*/
       w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);
       if (w == -1) {
           perror("waitpid");
           exit(EXIT_FAILURE);
       }
       if (WIFEXITED(status)) {
           printf("exited, status=%d\n", WEXITSTATUS(status));
       } else if (WIFSIGNALED(status)) {
           printf("killed by signal %d\n", WTERMSIG(status));
       } else if (WIFSTOPPED(status)) {
           printf("stopped by signal %d\n", WSTOPSIG(status));
       } else if (WIFCONTINUED(status)) {
           printf("continued\n");
       }
   } while (!WIFEXITED(status) && !WIFSIGNALED(status));
   exit(EXIT_SUCCESS);
}}
```

Il programma crea un processo figlio e ne attende la terminazione/sospensione acquisendone il motivo

Segnali: esercizi

- Documentarsi con il `man` sulla funzione `waitpid`
- Avviare due bash
- Compilare il precedente programma ed eseguire il processo in una bash
- Durante l'esecuzione inviare un segnale al processo figlio dall'altra bash
- Verificare il comportamento del processo padre
- (Ri)-provare inviando i segnali:
  SIGSTOP, SIGCONT, SIGKILL, SIGINT, SIGUSR1
- Documentarsi tramite il `man` sul significato di questi segnali

Ignorare SIGCHLD?

Per gestire un segnale un processo può optare per due particolari azioni:

SIG_IGN e SIG_DFL

Vi è differenza tra affidarsi all'azione di default (ignore) e impostare esplicitamente l'azione SIG_IGN:

- l'azione di default causa la permanenza del figlio nello stato zombie fino alla esecuzione della `wait()`
- l'azione SIG_IGN causa la rimozione del processo zombie
- effetto simile a SIG_IGN si ha impostando il flag SA_NOCLDWAIT tramite la system call `sigaction()` (vedi dopo)

Tuttavia affidarsi a queste possibilità può ridurre la portabilità del codice (non tutti i sistemi/kernel le implementano nello stesso modo)
L'unico modo sicuro per gestire gli zombie è installare un handler per SIGCHLD che esegua `wait()`

Gestione dei Segnali

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act,
                                 struct sigaction *oact);
```

- `signo` è il segnale che si vuole gestire
- `act` specifica la gestione del segnale; la struct a cui punta `act` è:

```
struct sigaction {
  void (*sa_handler)(int); /* azione da compiere */
  sigset_t sa_mask;   /* altri segnali da bloccare */
  int sa_flags;       /* flag che influiscono
                       *   sull'effetto del segnale */
  void (*sa_sigaction)(int, siginfo_t *, void *);
              /* handler usato in alternativa
               * a sa_handler se flags vale SA_SIGINFO */
  }
```

- in `oact` vengono salvati i valori correnti (per, eventualmente, ripristinarli).

dove...

Gestione dei Segnali

- `sa_handler` è una funzione C (solitamente definita dall'utente) con prototipo, ad esempio, come questo:

```
void miagestione(int signo) {
                ...
}
```

  Ha come unico parametro il numero del segnale ricevuto.
- La *registrazione* dell'handler avviene così:

```
act.sa_handler = miagestione;
```

- Per mascherare altri segnali si può usare:

```
sigfillset(&(act.sa_mask));
```

  che maschera tutti i segnali (DURANTE l'esecuzione di `miagestione`).
- La funzione `miagestione` può essere sostituita da una costante SIG_DFL o SIG_IGN (si ricordi che non tutti i segnali possono essere gestito o ignorati)

Gestione dei Segnali

- Un segnale bloccato resta pending
- I segnali mascherati durante l'esecuzione di un handler vengono bloccati
- Se vi sono più segnali pending dello stesso tipo, solo uno viene mantenuto (fa eccezione SIGCHLD)
- Due particolari flag possono essere usati nell'installare un handler con la chiamata a `sigaction()`:

  SA_NOCLDWAIT e SA_NOCLDSTOP

  per modificare il comportamento della `wait()` relativamente alle notifiche dei cambi di stato dei processi figli

ATTENZIONE:

Non tutti i segnali possono essere ignorati, gestiti, bloccati

Ad esempio SIGKILL e SIGCONT

Esiste una altra funzione simile a `sigaction()`

```
sighandler_t signal(int signum, sighandler_t handler)
```

Meglio non utilizzarla: può avere comportamenti inattesi e la sua implementazione non è completamente specificata (vedi il `man`)

Esercizio

- Scrivere un programma in cui il processo imposta un handler per il segnale SIGINT
- Altri segnali sono bloccati durante la gestione di SIGINT
- Scoprire che significato ha il segnale SIGINT e come spedirlo ad un processo
- L'handler intercetta il segnale e stampa un messaggio
- Il processo continua l'esecuzione per un tempo sufficiente all'utente per poter inviare il segnale da un'altra bash (con che comando?)
- Se per far "perdere tempo" al processo si usa una lunga `sleep()`, cercare di capire cosa succede se si invia il segnale SIGINT durante tale sleep
- Si scopra con `man 2 alarm` cosa fa `alarm()`
- Si scopra con `man 3 sleep` come `sleep()` potrebbe interferire con `alarm()`