LabSO

processi e ipc

Processi / ipc

- **Per comunicare i processi necessitano di un qualche elemento condiviso**
  - a livello di “generazione” (si imposta da un antenato) es.: il “padre” PRIMA di generare i “figli” predispone delle strutture dati per il passaggio di informazioni
  - a livello di sistema (es.: file-system)

Processi / ipc

Forking e SEGNALI

Processi / ipc: forking - I

- **system call “`fork`”**
  è la base per la “creazione” di nuovi processi:
  - “clonazione”
  - gli elementi principali sono duplicati: PC (prg counter) e registri, tabella dei file e dati di processo (variabili)
  - l’esecuzione di entrambi i processi prosegue dal punto esattamente successivo all’invocazione della fork (se usata “left-side” - es.: `fork();` - dall’istruzione successiva, se usata “right-side” - es.: `fid=fork();` - dall’assegnamento)

Processi / ipc: forking - II

- i processi hanno “pari dignità”: la gestione è immediatamente passata allo scheduler (ordine di esecuzione non predicibile)
- solo in caso d’errore non è generato un nuovo processo, in questo caso al chiamante è restituito il valore `-1`
- la funzione in caso di successo “ritorna” dunque in entrambi I processi (quindi due volte), ma con valori differenti:
  - `0` al processo nuovo appena generato
  - il *pid* del processo generato al processo chiamante

Processi / ipc: forking - III

Esempio (quante righe saranno generate in output?)

```
#include  <stdio.h>
#include  <unistd.h>
int main() {
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

Processi / ipc: forking - IV

- `pid_t fork()`
  genera un nuovo processo
- `pid_t getpid()`
  restituisce il “pid” del processo
- `pid_t getppid()`
  restituisce il “pid” del padre
- `pid_t wait(int *status)`
  attende la conclusione dei figli, è possibile avere informazioni sullo stato d’uscita passando una variabile, altrimenti si può usare NULL
- `pid_t waitpid(pid_t pid, int *status, int options)`
  attende la conclusione di un figlio specifico (pid>0), di tutti (pid=-1) o di quelli appartenenti a un certo gruppo (pid<-1: appartenenti al gruppo il cui valore assoluto è pid, pid=0: appartenti allo stesso gruppo del chiamante). `Options` specifica se attendere in casi particolari.
  wait(NULL) è equivalente a waitpid(-1, NULL, 0);

Processi / ipc: forking - V

```
#include <stdio.h>      // for printf
#include <unistd.h>     // for fork, getpid, getppid()
#include <time.h>       // for time
#include <stdlib.h>     // for srand, rand, wait

void delay(double dly) {
const time_t start = time(NULL);
time_t current;
do {
time(&current);
} while (difftime(current, start) < dly);
}

int rnd(int min, int max) {
srand(time(NULL)+getpid()); // init rand's seed
int r;                  // will hold result
int steps=(max-min)+1; // how many
r = rand() % (steps);
r += min;
return (r);
}
```

```
int main() {
pid_t fid;
int r;
fid=fork();
if (fid == -1) {
printf("?Error. Forking failed!\n");
return (1);
};
r=rnd(1,3);
delay(r);
if (fid > 0) {
delay(rnd(1,3));
printf("(waited for %d secs) Parent, My pid is %d.
Generated child has pid=%d\n", r, getpid(), fid);
} else {
delay(rnd(1,3));
printf("(waited for %d secs) Child.  My pid is %d.
Parent of me has pid=%d\n", r, getpid(), getppid());
};
wait(NULL); // wait for all children (parent has one,
child has none)
return (0);
}
```

Processi / ipc: exec - I

- **Le chiamate della famiglia “exec” (`#include unistd.h`) sostituiscono l’immagine del processo corrente con un altro**
  - `int execvp (const char *file, char *const argv[]);`

    es.:

    ```
    char *args[]={"./other",NULL};
    execvp(args[0],args);
    ```

(passo anche l’eseguibile come argomento “0”)

Processi / ipc: exec - II

- **`execl / execle / execlp / execlpe`**
- **`execv / execve / execvp / execvpe`**
  - Il primo gruppo prende gli argomenti in numero variabile anzichè come vettore di stringhe come nel secondo gruppo
  - le varianti con le “e” finale permettono di passare un ulteriore vettore di stringhe del tipo “*key*=*value*” passate come “ambiente”
  - le varianti con le “p” prendono un nome di file e lo cercano nel path di sistema, altrimenti (le altre varianti) si deve passare il path completo (assoluto o relativo)

Processi / ipc: exec - esercizio

- **Realizzare un programma che accetta come argomento un intero “n” e attende n secondi prima di terminare (stampando un feedback testuale e il suo pid all’avvio e al termine): ad ogni secondo di attesa può richiamare “se stesso” con una probabilità del 50% e argomento n/2 arrotondato per difetto solo per n>1**

Processi / ipc: segnali - I

- I segnali sono interruzioni a livello software identificabili con un valore numerico cui è associata un'etichetta mnemonica. Solitamente possono essere ignorati, salvo un paio di eccezioni

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

tra cui:

- `SIGALRM` (alarm clock)
- `SIGCHLD` (child terminated)
- `SIGCONT` (continue, if stopped)
- `SIGINT` (terminal interrupt, e.g. CTRL+C)
- `SIGKILL` (cannot be caught or ignored)
- `SIGQUIT` (terminal quit)
- `SIGSTOP` (cannot be caught or ignored)
- `SIGTERM` (termination)
- `SIGUSR1` / `SIGUSR1` (user's signals)

Processi / ipc: segnali - II

- **Per ogni processo è tenuta una lista dei segnali pendenti (emessi, da gestire, cioè da “catturare” da parte del processo) e di quelli bloccati (cioè che non sono comunicati al processo): un bit per ogni segnale in ogni lista**
- **Ad ogni schedulazione di un processo le due liste sono controllate:**
  - se non vi sono segnali pendenti o ce ne sono ma sono tutti bloccati il processo prosegue normalmente
  - se vi sono segnali pendenti non bloccati il corrispondente “gestore” (*handler*) - una funzione dedicata - è eseguito anzichè proseguire con l’esecuzione normale del processo.

Processi / ipc: segnali - III

- **Ogni segnale ha un suo “handler” di default che tipicamente può:**
  - ignorare il segnale (non fa nulla: l’effetto che si ha è che il processo prosegue normalmente)
  - terminare il processo
  - continuare l’esecuzione (se il processo era in “stop”)
  - stoppare il processo
- **Ogni processo può sostituire il gestore di default con una funzione “custom” (a parte SIGKILL/SIGSTOP) e comportarsi di conseguenza**

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

Processi / ipc: segnali - IV

- **Un gestore “custom” è una funzione detta “handler” che deve restituire “void” e accettare un intero come argomento: sarà il codice del segnale gestito (si può quindi usare lo stesso handler per più segnali e discriminare sul valore dell’argomento)**

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

void sighandle_int(int sig) {
  printf("\n[SIGINT]\n");
}
```

```
int main() {
  signal(SIGINT, sighandle_int);
  while (1) {
    printf(".");
    sleep(1);
    fflush(stdout); // force stdout
  };
  return 0;
}
```

Processi / ipc: segnali - V

- **Il secondo argomento di `signal` può essere:**
  - SIG_DFL : attiva/ripristina il default
  - SIG_IGN : attiva un gestore che ignore il segnale
- **la funzione restituisce un riferimento al gestore attivo prima della variazione: può essere usato (ad esempio dentro il corpo dell'handler) per chiamarlo successivamente, creando di fatto un "wedge"**

Processi / ipc: segnali - VI

- **Per “inviare” un segnale si può usare `kill`:**

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

  - `pid`: pid del processo a cui inviare il segnale
  - `signal`: il (codice del) segnale da inviare
  - restituisce 0 se il segnale è emesso con successo

Processi / ipc: segnali - esercizio

- **Fare un programma che genera 4 figli e invia a ciascuno un segnale SIGUSR1 o SIGUSR2 con una probabilità del 50%. Il figlio deve “rispondere” al segnale stampando su stdout il proprio pid e il nome del segnale ricevuto e poi chiudersi. Alla chiusura di tutti I figli anche il padre può terminare.**

Processi / ipc: VII

- **Allarme (SIGALRM): la funzione**

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

scatena un segnale SIGALRM sullo stesso processo d’esecuzione dopo il numero di secondi indicato e restituisce il numero di secondi restanti dovuti alla chiamata precedente se esistente, 0 altrimenti

Processi / ipc: VIII

- **Pausa: la funzione**

```
int pause (void)
```

sospende il processo attivo fino all’arrivo di un segnale gestito o di terminazione