

Riferimenti e Crediti

- <http://jan.netcomp.monash.edu.au/ProgrammingUnix/>
- <http://dia.uniroma3.it/~cenciott>
- <http://www-users.cs.umn.edu/~bentlema/unix/>
- **An Introductory 4.4BSD
Interprocess Communication Tutorial**
- **An Advanced 4.4BSD Interprocess
Communication Tutorial**

Processi

- Programma: descrizione statica
- Processo: concetto dinamico
- Processi concorrenti (contrapposti a quelli sequenziali), vengono implementati tramite:
 - pseudo-parallelismo
 - reale parallelismo se più CPU sono disponibili
- Terminologia e granularità
 - task: indipendente, non condivide nulla con il suo creatore
 - process: eredita o alcune informazioni dal padre
 - thread (o lightweight process): eredita fin quanto possibile le proprie informazioni dal suo creatore. Il più economico da inizializzare.

Creazione di nuovi processi

- Si utilizza la funzione `fork()` :

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

```
pid_t fork(void)
```

- Restituisce:
 - PID del figlio al padre
 - 0 al figlio

(-1 se fallisce)

- Il valore di ritorno servirà ai due processi per intraprendere azioni diverse
- Il figlio eredita quasi tutte le caratteristiche (codice, dati, descrittori dei file ecc. ecc.)

fork.c

```
#include <stdio.h>
#include <sys/types.h>
int main(void) {
    pid_t pid;
    pid = fork();
    if (pid == -1) {
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    if (pid == 0) {
        printf("I'm the child\n");
        exit(0);
    }
    if (pid > 0) {
        printf("I'm the parent with child %d\n", pid);
        exit(0);
    }
}
```

```
$ gcc fork.c -o fork
$ ./fork
```

Eseguito solo dal figlio

Eseguito solo dal padre

Creazione di nuovi processi (2)

- La funzione `fork()` può fallire se:
 - mancanza di PID liberi
 - raggiunto il numero max di processi
 - impossibilità di allocare la memoria per le strutture del processo figlio
- L'area dati del padre non viene immediatamente copiata in quella del figlio ma segue una politica *copy-on-write*

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

```
pid_t getpid(void)    ritorna il PID del chiamante
```

```
pid_t getppid(void)   ritorna il PID del padre del  
                      chiamante
```

Esercizio: scrivere un programma in cui un processo genera un altro processo. Il primo stamperà il proprio PID e quello del figlio, l'altro stamperà il proprio PID e quello del padre.

Esercizio: scrivere un programma che dichiari una variabile locale di tipo int inizializzata a 0.

Di seguito si genera un altro processo che la incrementa. Si stampi il suo contenuto da entrambi i processi.

Eliminazione di Processi

- Si utilizza la funzione `exit()` :
`include <unistd.h>`
`void exit(int code)`
- riceve un intero che risulta utile per il debugging in quanto restituito alla shell chiamante
- Se un processo termina prima dei suoi figli, questi vengono “adottati” dal processo primo processo init (di PID pari a 1)
- Vedremo che un ulteriore metodo per far terminare un processo è tramite un *segnale* di **SIGKILL**

Messaggi di Errore

- Se una chiamata di una funzione del kernel fallisce, ritorna dei valori specifici documentati nella *man page* della funzione e setta la variabile `errno`
- `errno` va dichiarata come `extern int`
- per ottenere una stringa che descriva l'errore
`include <string.h>`
`char * strerror(int errnum)`
- la stampa di un msg di errore si può effettuare con:
`include <stdio.h>`
`void perror(const char *s)`

Esercizio: determinare il numero massimo di processi eseguibili in loop eterno. Stampare il relativo messaggio di errore.

Sospensione dei Processi

- La funzione `sleep()` sospende il processo chiamante fino ad un max di `s` secondi
- La funzione `wait()` attende che un processo figlio termini e quindi restituisce il suo PID

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

```
#include <sys/wait.h>
```

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

Wait() e Sleep()

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/types.h>
int main(int argc, char *argv[]) {
    pid_t pid;
    if ((pid = fork()) == 0) { /* child */
        printf("sleeping...\n");
        sleep(5);
        printf("waking ... and exiting\n");
    } else if (pid > 0) {
        printf("waiting for child\n");
        wait(NULL);
        printf("child woke up\n");
    }
}
```

Nuovo Codice Eseguitibile per un Processo

- Una volta creato un processo, si può rimpiazzare il suo codice con quello di un diverso *programma* eseguendo la funzione `exec()`.
- Non viene creato un nuovo processo, ma viene solo “installato” un nuovo codice da eseguire
- Vengono mantenuti PID, PPID, ambiente, process group identifier, terminale di controllo e permessi sui file

exec()

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

```
int execlp(char *file, Nome dell'eseguibile
```

```
char *arg0,
```

```
char *arg1,
```

```
...
```

```
...
```

```
char *argn,
```

```
(char *) 0); L'ultimo argomento è NULL
```

arg0 spesso
coincide con
file ed è il
nome riportato
dal comando ps

Argomenti della
chiamata al programma

```
if (fork() == 0) {  
    if (execlp("lpr",  
               "lpr",  
               "myfile",  
               (char *) 0) == -1)  
        fprintf(stderr, "exec failed\n");  
} else { /* ...parent... */ }
```

Se fallisce
restituisce -1

Varie Versioni di exec()

		\$PATH	env
Exec1	const char *path, const char *arg0,...	No	No
Execv	const char *file, const char *argv[]	No	No
Exec1p	const char *file, const char *arg0,...	Yes	No
Exec1e	const char *path, const char *arg0,..., const char *envp[]	No	Yes
Execvp	const char *file, const char *argv[]	Yes	No
Execve	const char *file, const char *argv[], const char *envp[]	No	Yes

Alcuni Dettagli sulle Varianti di `exec()`

- In genere va specificato il percorso assoluto di un eseguibile
- `exec1p` ed `execvp` usano la variabile d'ambiente `PATH` per la ricerca dell'eseguibile
- Nelle `exec1e` ed `execve` e' possibile specificare un ambiente di esecuzione
- Gli argomenti del programma possono essere specificati separatamente o in un array di stringhe (`execv`, `execvp`, `execve`)

Un Esempio fork()+exec()

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/types.h>
int main(int argc, char *argv[]) {
    pid_t pid;
    if ((pid = fork()) < 0) {
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    if (pid == 0) {
        execlp("echo", "echo", "Hello from",
              "the child", (char *) NULL);
        fprintf(stderr, "execl failed\n");
        exit(2);
    }
    printf("parent carries on\n");
    exit(0);
}
```

Un Altro Esempio di fork()+exec()

```
...
if ((pid = fork()) == 0)
    if (execlp("lpr", "lpr", "myfile",
               (char *) 0) == -1)
        fprintf(stderr, "exec failed\n");
    else {
        /* do lots of things while
           the printing takes place
        */
        /* now remove the file */
        while (wait(NULL) != pid);
        unlink("myfile");
        /* and do lots more things */
    }
}
```

Comunicazione tra Processi: i Segnali

- Permettono una comunicazione *asincrona*
- Vengono lanciati attraverso la chiamata

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

#include <signal.h>

int kill(pid_t pid, int sig);
```
- `pid` indica il numero del processo al quale si vuole inviare un determinato segnale
 - `pid= 0` per tutti i processi del gruppo di quello corrente
 - `pid=-1` per tutti i processi tranne il primo nella tavola dei processi
 - `pid<-1` il segnale e' spedito a tutti i processi del gruppo `-pid`
- `sig` e' il numero del segnale che si vuole inviare
- Per poter gestire i segnali si utilizzano apposite funzioni o procedure dedicate (*handlers*)

Invio di un Segnale

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    pid_t pid;
    if ((pid = fork()) == 0)
        while (1);
    else {
        /* let the child run for 10 seconds max */
        sleep(10);
        kill(pid, SIGINT); // ... and then kill it!
    }
}
```

I segnali possono essere anche lanciati da shell:

bash\$ myexe &

[1] 19688

bash\$ kill -SIGUSR1 19688

Esercizio: scrivere un programma per permettere all'utente di generare processi che eseguano cicli eterni (fino ad un massimo di 20), o di uccidere quelli già creati.

Esercizio: scrivere un videogame in cui, ad intervalli di tempo casuali, compaiono icone aventi come titolo lettere casuali. Un'icona verrà eliminata battendo la corrispondente lettera sulla tastiera. Il gioco termina se sullo schermo compaiono 10 icone.

```
include <stdlib.h>
```

```
long int random(void) restituisce un valore casuale da 0 a RAND_MAX
```

```
xterm -iconic    attiva xterm in forma iconizzata
```

```
    -n <name>    definisce il titolo della finestra xterm
```

Signal()

- Per intercettare un segnale all'interno di un processo si utilizza una funzione routine `signal()` che permette di "installare" una funzione da eseguire (handler) ogni volta che si riceve il segnale indicato:

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

Eseguire: `man signal`

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

- Ritorna il puntatore alla funzione handler precedente o -1 in caso di errore
- `sig` è il segnale a cui associare il nuovo handler
- `handler()` può essere:
 - `SIG_DFL`, a pointer to `SIG_DFL()` (fornita dalla libreria)
 - `SIG_IGN`, a pointer to `SIG_IGN()` (fornita dalla libreria)
 - il puntatore ad una funzione definita dall'utente

Handlers

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

```
void int_handler(int sig) {
    printf("Ouch - shot in the ...\n");
    exit(2);
}
```

```
int main(int argc, char *argv[]) {
    pid_t pid;
    if ((pid = fork()) < 0) {
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    if (pid == 0) {
        signal(SIGINT, int_handler);
        while (1) printf("Running amok!!!\n");
    }
    sleep(3);
    kill(pid, SIGINT);
    exit(0);
}
```

Tavola dei Segnali di Linux (1)

Nome segnale	Valore	Azioni	Commenti
SIGHUP	1	A	Hangup detected
SIGINT	2	A	Interrupt from keyboard
SIGQUIT	3	A	Quit from keyboard
SIGILL	4	A	Illegal Instruction
SIGTRAP	5	CG	Trace/breakpoint trap
SIGABRT	6	C	Abort
SIGUNUSED	7	AG	Unused signal
SIGFPE	8	C	Floating point exception
SIGKILL	9	AEF	Terminal signal
SIGUSR1	10	A	User defined signal 1
SIGSEGV	11	C	Invalid memory reference
SIGUSR2	12	A	User defined signal 2
SIGPIPE	13	A	Write to pipe with no readers

Tavola dei Segnali di Linux (2)

Nome segnale	Valore	Azioni	Commenti
SIGALRM	14	A	Timer signal from alarm(1)
SIGTERM	15	A	Termination signal
SIGSTKFLT	16	AG	Stack fault on coprocessor
SIGCHILD	17	B	Child terminated
SIGCONT	18		Continue if stopped
SIGSTOP	19	DEF	Stop process
SIGTSTP	20	D	Stop typed at tty
SIGTTIN	21	D	tty input for background p.
SIGTTOU	22	D	tty output for backg. process

Tavola dei Segnali di Linux (3)

Nome segnale	Valore	Azioni	Commenti
SIGIO	23	AG	I/O error
SIGXCPU	24	AG	CPU time limit exceeded
SIGXFSZ	25	AG	File size limit exceeded
SIGVTALRM	26	AG	Virtual time alarm
SIGPROF	27	AG	Profile signal
SIGWINCH	29	BG	Window resize signal

A: per default l'azione e' la terminazione del processo

B: per default il segnale viene ignorato

C: per default viene creato un coredump

D: per default il processo viene fermato

E: il segnale non può essere intercettato

F: il segnale non può essere ignorato

G: il segnale non e' nello standard POSIX.1

Segnali da Terminale

- $\text{^C} = \text{CTRL+C} \longrightarrow \text{SIGINT}$

la shell invia al processo il segnale

- $\text{^Z} = \text{CTRL+Z} \longrightarrow \text{SIGTSTP}$

la shell sospende il processo

- $\text{^\backslash} = \text{CTRL+\backslash} \longrightarrow \text{SIGQUIT}$

la shell ferma il processo

ed effettua un core dump

terminal.c (segue...)

Cattura dei Segnali Lanciati da Terminale

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

/* SIG_IGN, SIG_DFL are pointers to
   functions returning integers */
int count = 0;
main (int argc, char* argv[]) {
    int i;
    void gotCntrlBSlash(), gotCntrlC();
    signal(SIGQUIT, gotCntrlBSlash); // message for ^\
    signal(SIGINT, gotCntrlC);       // message for ^C,
                                    // reset after 3 times

    signal(SIGTSTP, SIG_IGN);        // ignore ^Z
    for (i=1; i<100; i++) {
        sleep(1);
        fprintf(stderr, (i%70==0)?".\n": ".");
    }
} /* end of main */
```

terminal.c (...continua)

```
void gotCntrlC() {
    signal(SIGINT, gotCntrlC);
    switch (++count) {
        case 1: fprintf(stderr, "(First ^C)");
                break;
        case 2: fprintf(stderr, "(Second ^C)");
                break;
        case 3: fprintf(stderr, "(Third ^C)");
                signal(SIGINT, SIG_DFL);
                break;
        default: break;
    }
} /* end of gotCntrlC */
```

```
void gotCntrlBSlash () {
    signal(SIGQUIT, gotCntrlBSlash);
    fprintf(stderr, "(Got a ^\\)");
}
```

Attesa di un Segnale

- La funzione `pause()` sospende un processo fino alla ricezione di un segnale

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

Esercizio: scrivere un programma che crei due processi. I due processi hanno un nome distinto che stampano indefinitivamente in maniera alternata:

```
<nome processo 1>
<nome processo 2>
<nome processo 1>
...
```

Esercizio: ripetere l'esercizio di sopra ma con **due** programmi diversi.

Segnali Temporizzati

- La funzione **alarm()** permette di lanciare un segnale di tipo SIGALRM ad un determinato processo dopo un certo numero di secondi

```
#include <unistd.h>
#include <signal.h>
unsigned int alarm (unsigned int secs);
```
- ritorna il numero di secondi mancanti all'invio del segnale

Esercizio: scrivere un programma che utilizzando almeno due processi richiami ad intervalli di tempo regolari il comando eseguibile `date`.

alarm.c

```
#include <stdio.h>
#include <signal.h>
int main (int argc, char* argv[]) {
    void announce();
    if (argc!=2) {
        fprintf(stderr, "Usage: %s seconds\n", argv[0]);
        exit(1);
    }
    signal(SIGALRM, announce);
    alarm((unsigned)atoi(argv[1]));
    pause(); /* wait for a signal */
} /* end main */

void announce() {
    fprintf(stdout, "Wake up! \n");
    exit (0);
}
```

Esercizio: scrivere un programma che genera un xterm, ed ogni volta che l'utente prova ad eliminarne uno, ne vengono generati altri due fino ad un massimo di 8 processi attivi contemporaneamente.

Esercizio: scrivere un programma che crei una catena di n processi: il primo genera ad intervalli di tempo regolari un segnale per il secondo, il quale attende un tempo casuale e ne genera un altro per il terzo; quest'ultimo attende un intervallo di tempo casuale e genera un segnale per il quarto, e così di seguito fino all'ennesimo processo che non genera nuovi segnali. Ciascun segnale ricevuto genera un corrispondente messaggio che specifica il processo ricevente.

Esercizio: scrivere un programma che simuli una coda di attesa con tre processi rappresentanti, rispettivamente, gli utenti che arrivano, la coda di attesa e l'impiegato che riceve gli utenti. Il primo invia al secondo un segnale che genera ad intervalli di tempo casuali per indicare un nuovo arrivo nella coda. Quando la coda non è vuota, il terzo processo impiega un tempo casuale prima di notificare al secondo che un utente è stato servito. Il processo che modella la coda stampa informazioni che descrivono la sequenza nella quale accadono gli eventi riportando di volta in volta anche il numero di utenti in coda.