Corso di Programmazione Concorrente

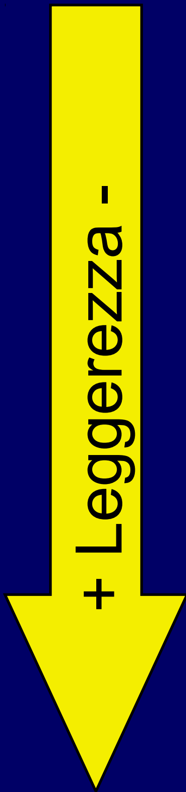
Processi

Valter Crescenzi
http://crescenzi.inf.uniroma3.it

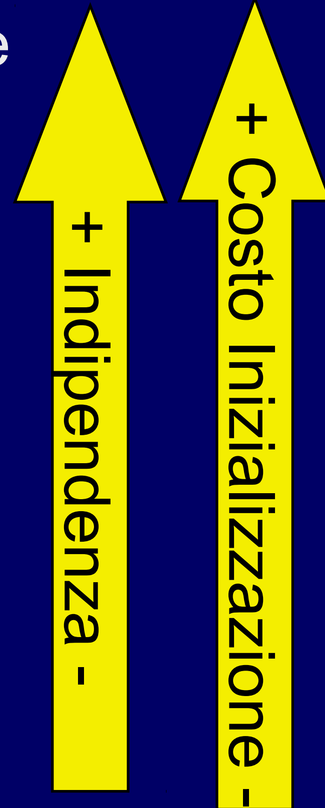
Sommario

- Processi vs Thread
- Creazione e terminazione di processi
  - chiamata di sistema `fork()`
  - chiamata di sistema `exit()`
- Implem. della primitiva astratta `join()`:
  - `wait()` e `waitpid()`
  - Processi zombie
- Chiamata di sistema `exec()`
  - Varianti di `exec()`
- Implem. della primitiva astratta `fork()`
  - Esempi: `exec() + fork()`

Processi vs Thread



- …
- Processo: unità di condivisione delle risorse (alcune possono essere inizialmente ereditate dal padre)
- …
- …
- Thread (o lightweight process): idealmente, è un flusso di esecuzione indipendente che condivide tutte le sue risorse, incluso lo spazio di indirizzamento, con altri thread
- …



Creazione di Nuovi Processi

- Si utilizza la funzione `fork()`:

```
include <unistd.h>
pid_t fork(void)
```

- Restituisce:
  - PID del figlio al padre (-1 se fallisce)
  - 0 al figlio
- Il valore di ritorno servirà ai due processi per intraprendere azioni diverse
- Il figlio eredita quasi tutte le caratteristiche (codice, descrittori dei file, segmenti di memoria condivisa 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 fallita\n");
    exit(1);
  }
  if (pid == 0) {
    printf(“Sono il figlio");
    exit(0);
  }
  if (pid > 0) {
    printf(“Sono il padre di %d\n", pid);
    exit(0);
  }
}
```

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

Eseguito solo dal figlio

Eseguito solo dal padre

`fork()`: Alcuni Dettagli

- La funzione `fork()` può fallire con errore:
  - EAGAIN: mancanza di PID liberi; raggiunto il numero max di processi
  - ENOMEM: impossibilità di allocare la memoria necessaria per mantenere 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*
  - ✔ quindi i dati *non sono condivisi*, sono soltanto inizializzati con i valori ereditati dal padre

Esercizi

```
include <unistd.h>
pid_t getpid(void)
pid_t getppid(void)
```

ritorna il PID del chiamante

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 intero inizializzata a 0.

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

Terminazione 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 primo processo *init* (PID=1)
- Un ulteriore metodo per far terminare un processo è tramite un *segnale* di `SIGKILL`

Sospensione dei Processi

- La funzione `sleep()` sospende il processo chiamante fino ad un max di s secondi

```
unsigned int sleep(unsigned int s)
```

- 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) { /* figlio */
    printf(“dormo…\n");
    sleep(5);
    printf(“…mi sveglio… ed esco\n");
  } else if (pid > 0) {
    printf(“aspetto mio figlio…\n");
    wait(NULL);
    printf(“…il bimbo si è svegliato");
  }
}
```

waitpid() (1)

- La funzione `waitpid()` sospende il processo chiamante in attesa della terminazione del processo figlio di PID specificato

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

- `pid` può essere
  - < -1: attende un qls figlio con GID pari a -PID
  - -1: attende un qls figlio esattamente come la `wait()`
  - 0: attende un qls figlio con GID pari al quello del chiamante
  - > 0: attende il figlio con il PID specificato

`waitpid()` (2)

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- `options` può essere una combinazione di
  - `WNOHANG`: evita la sospensione anche se nessun figlio è già terminato
  - `WUNTRACED`
- `status` se è stato passato non NULL, la chiamata restituisce alcune informazioni sullo stato del processo figlio che è effettivamente terminato
  - Esistono diverse macro per interrogare agevolmente tale stato

```
man 2 waitpid
```

`wait()` `waitpid()` e Processi Zombie

- Proprio per supportare queste chiamate il kernel evita di deallocare tutte le risorse precedentemente allocate per un processo ora terminato
- Bisogna eseguire esplicitamente le wait per forzare la deallocazione di tali strutture da parte del kernel
- Esiste un segnale che permette di automatizzare la chiamata di `wait()` ogni volta che un processo figlio termina

zombie.c

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

int main () {
  pid_t child_pid;
  /* Create a child process.  */
  child_pid = fork ();
  if (child_pid > 0) {// This is the parent process.
    sleep(60); //  Sleep for a minute.
  }
  else {      // This is the child process.
    exit(0); // Exit immediately.
  }
  return 0;
}
```

`zombie.c`: Esempio di esecuzione

```
$ gcc zombie.c -o zombie
$ ./zombie &
[1] 19238
$ ps -o pid,ppid,command,state
  PID  PPID COMMAND             S
 2627  2625 /bin/bash           S
19238  2627 ./zombie            S
19239 19238 [zombie]   <defu Z
19240  2627 ps -o pid,ppid,c R
$
```

Nuovo Eseguibile 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, GID, terminale di controllo e permessi sui file

exec()

```
include <unistd.h>
int execlp(char *file,
           char *arg0,
           char *arg1,
           …
           …
           char *argn,
          (char *) 0);
```

Nome dell'eseguibile

Argomenti della chiamata al programma

L'ultimo argomento è NULL

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

```
if (fork() == 0) {
    execlp("lpr",
           "lpr",
           "myfile",
           (char *) 0);
    fprintf(stderr, "exec fallita\n");
} else { /* …padre… */ }
```

Se fallisce restituisce -1

Varianti di `exec()`

		$PATH	env
exec**l**	const char *path, const char *arg0,…	No	No
exec**v**	const char *file, const char *argv[]	No	No
exec**lp**	const char *file, const char *arg0,…	Yes	No
exec**le**	const char *path, const char *arg0,…, const char *envp[]	No	Yes
exec**vp**	const char *file, const char *argv[]	Yes	No
exec**ve**	const char *file, const char *argv[], const char *envp[]	No	Yes

`exec()`: Dettagli sulle Varianti (1)

- `execlp` ed `execvp` usano la variabile d'ambiente `PATH` per la ricerca dell'eseguibile
- Nelle `execle` ed `execve` è possibile specificare un ambiente di esecuzione
- Gli argomenti del programma possono essere specificati separatamente o in un array di stringhe (`execv`, `execvp`, `execve`)

`exec()`: Dettagli sulle Varianti (2)

- Le varianti che contengono una ‘`p`’ nel nome
  - ricercano l’eseguibile nel PATH degli eseguibili
- Le varianti che contengono una ‘`v`’ nel nome
  - accettano gli argomenti dell’eseguibile come un array di puntatori a stringhe che finisce per NULL
- Le varianti che contengono una ‘`l`’ nel nome
  - accettano gli argomenti dell’eseguibile con il meccanismo delle funzioni con numero varabile di argomenti
- Le varianti che contengono una ‘`e`’ nel nome
  - accettano un array di variabili d’ambiente come argomento aggiuntivo; anche in questo caso di tratta di un array di puntatore a stringhe che finisce per NULL. La singola stringa deve essere nella forma ‘NAME=value’

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 fallita\n");
    exit(1);
  }
  if (pid == 0) {
    execlp("echo","echo",“Ciao dal",
           “figlio",(char *) NULL);
    fprintf(stderr,"execl fallita\n");
    exit(2);
  }
  printf(“il padre prosegue\n");
  exit(0);
}
```

Altro Esempio di `fork()`+ `exec()`

```
…
if ((pid = fork()) == 0)
  if (execlp("lpr","lpr","myfile",
              (char *) 0) == -1)
    fprintf(stderr,"exec fallita\n");
  else {
   /*  fai molto altro mentre la stampa è in corso  */
   /* adesso rimuovi il file */
   while (wait(NULL) != pid);
   unlink("myfile");
  /*  e fai molto altro ancora  */
 }
```

Ultimo Esempio di `fork()+ exec()` (1)

…include omessi…

```
int spawn (char* program, char** arg_list) {
  pid_t child_pid;
  child_pid = fork (); // Duplicate this process
  if (child_pid != 0)
    /* This is the parent process.  */
    return child_pid;
  else {
    // Now execute PROGRAM, searching for it in the path.

    execvp (program, arg_list);
    // The execvp function returns only for errors
    fprintf (stderr, "an error occurred in execvp\n");
    abort ();
  }
}
int main() {…}
```

Ultimo Esempio di `fork()`+ `exec()` (2)

```
int spawn() { … }
int main () {
  // The argument list to pass to the "ls" command
  char* arg_list[] = {
    "ls", // argv[0], the name of the program.
    "-l",
    "/",
    NULL  // The argument list must end with a NULL
  };

  /* Spawn a child process running the "ls" command.
     Ignore the returned child process id.  */
  spawn ("ls", arg_list);
  printf ("done with main program\n");
  return 0;
}
```

Esercizi

Esercizio: quanti processi crea questo frammento di codice?

```
…
int main () {
  int i=0;
  for(int i=0; i<N; i++) {
     pid_t child_pid;
     child_pid = fork ();
  }
  return 0;
}
```

Esercizi

Esercizio: un processo padre crea un numero di processi figli specificato dall'utente. Ciascuno di questi effettua una pausa casuale e termina. Il padre termina solo quando tutti i suoi figli sono già terminati.

Esercizio: risolvere l'esercizio di sopra con **due** programmi distinti, `padre.c` e `figlio.c`.

Esercizio: programmare una mini-shell che iterativamente legge una riga di testo ed esegue il suo contenuto in un processo dedicato. La shell termina con il comando '`stop`'.