Chiamate di sistema per la Gestione dei processi in POSIX

E.Mumolo, DEEI

mumolo@units.it

Process id ed altri identificatori

- **pid_t getpid();** // Process id del processo chiamante
- **pid_t getppid();** // Process id del processo padre
- **uid_t getuid();** // Real user id
- **uid_t geteuid();** // Effective user id
- **gid_t getgid();** // Real group id
- **gid_t getegid();** // Effective group id

Process id

- **System call:**

```
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
```

  - Cambiano real uid/gid e effective uid/gid
  - Esistono delle regole per cambiare questi id:
    - Se il processo ha privilegi da superutente, la funzione **setuid** cambia real uid/ effective uid / saved-set-uid con uid
    - Se il processo non ha privilegi da superutente e uid è uguale a real uid o a saved-set-uid, la funzione **setuid** cambia effective uid
    - Se nessuna di queste condizioni è vera, **errno** è settato uguale a EPERM e viene ritornato un errore
- Per quanto riguarda group id, il discorso è identico
- Solo un processo eseguito da root può cambiare il real uid; normalmente, esso viene settato da login e non cambia più

Controllo processi

- **System call:**

```
#include <unistd.h>

pid_t fork(void);
```

  - crea un nuovo processo child, copiando completamente l'immagine di memoria del processo parent
  - data, heap, stack vengono copiati
  - il codice viene spesso condiviso
  - in alcuni casi, si esegue copy-on-write
  - sia il processo child che il processo parent continuano ad eseguire l'istruzione successiva alla **fork**
- **fork** viene chiamata una volta, ma ritorna due volte
  - processo child: ritorna 0 (E' possibile accedere al pid del parent tramite la system call **getppid**)
  - processo parente: ritorna il process id del processo child
  - errore: ritorna valore negativo

Esercizio: cosa viene stampato?

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

int             glob = 6;               /* variabile globale */

Int main(void)
{
   int          var;
   pid_t        pid;

   var = 88;

   printf("prima di fork\n");

   if ( (pid = fork()) < 0)
        err_sys("errore di fork ");
   else if (pid == 0) {                 /* processo figlio */
        glob++;                         /* modifica la variabile globale */
        var++;
   } else
        sleep(2);                               /* processo padre */

   printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
   exit(0);
}
```

Esempio: creazione multipla

```
#include        <sys/types.h>
#include        <sys/wait.h>
#include        "ourhdr.h"

Int main(void)
{
   pid_tpid;

   if ( (pid = fork()) < 0)     printf("errore di fork");
   else if (pid == 0) { /* primo figlio */
        if ( (pid = fork()) < 0) printf("fork error");
        else if (pid > 0)
                exit(0);        /* il primo figlio genera un processo */

        /* qui sono il secondo figlio (processo generato) */

        sleep(2);
        printf("Sono il secondo figlio, sono stato creato da = %d\n", getppid());
        exit(0);
   }

   if (waitpid(pid, NULL, 0) != pid)  // aspetta la terminazione del primo figlio
        printf("waitpid error");

   /* Ora sono il processo originale */

   exit(0);
}
```

Attesa terminazione figlio

- **System call:**

  #include <sys/wait.h>

  pid_t wait(int *status);

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

  - Ottengono lo stato di uno dei processi figlio
  - **wait** può bloccare il chiamante fino a quando un qualsiasi processo generato non termina
  - **waitpid** può mettersi in attesa di uno specifico processo
  - **status** è un puntatore ad un intero; se diverso da **NULL**, lo stato della terminazione viene messo in questa locazione
  - Le chiamate ritornano l'ID del processo che è terminato
  - **waitpid** ha delle opzioni per evitare di bloccarsi

Tabelle dei file

- **Tabelle coinvolte: User File Descriptor Table, File Table, Inode (o Vnode) table**
- Esempio: un processo che apre due file distinti

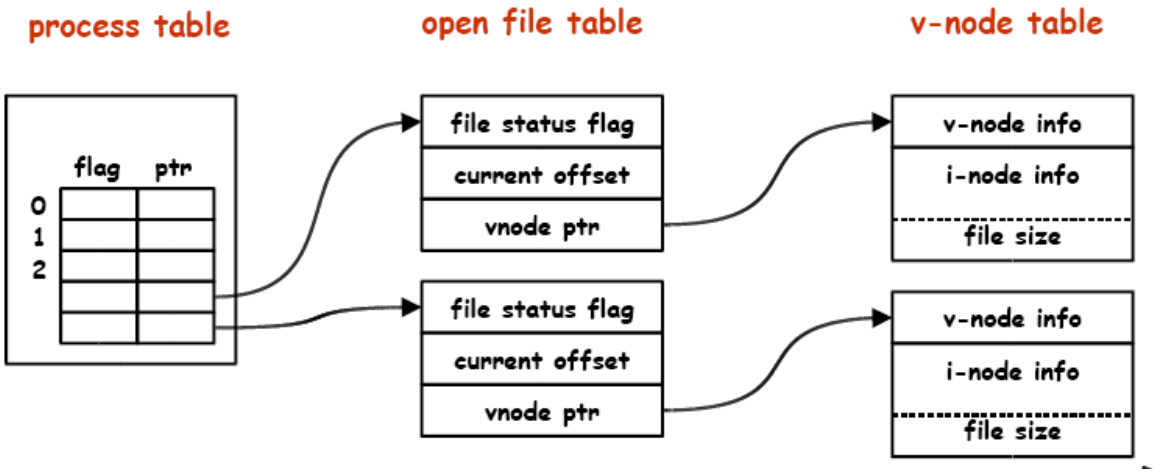
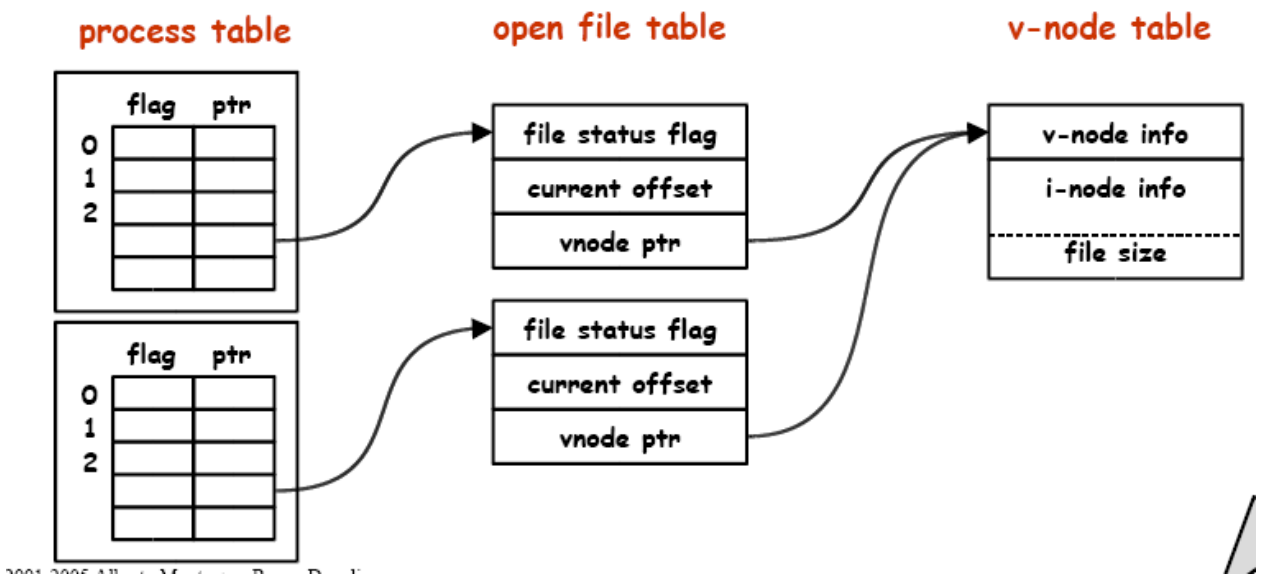


Tabelle dei file

- Esempio: due processi che aprono lo stesso file



Redirezione

- Una caratteristica della chiamata **fork** è che tutti i descrittori che sono aperti nel processo parent sono duplicati nel processo child
- La redirezione viene ottenuta con la ***Duplicazione descrittori:***

```
#include <unistd.h>
int dup(int fildes);
int dup2(int fildes, int fildes2);
```

- **Funzione dup**
  - Seleziona il più basso file descriptor libero della tabella dei file descriptor
  - Assegna la nuova file descriptor entry al file descriptor selezionato
  - Ritorna il file descriptor selezionato
- **Funzione dup2**
- Con **dup2**, specifichiamo il valore del nuovo descrittore come argomento **filedes2**
- Se **filedes2** è già aperto, viene chiuso e sostituito con il descrittore duplicato
- Ritorna il file descriptor selezionato

Redirezione

- **Proprietà che il processo child eredita dal processo parent:**
  - real uid, real gid, effective uid, effective gid
  - gids supplementari
  - id del gruppo di processi
  - session ID
  - terminale di controllo
  - set-user-ID flag e set-group-ID flag
  - directory corrente
  - directory root
  - maschera di creazione file (umask)
  - maschera dei segnali
  - flag close-on-exec per tutti i descrittori aperti
  - environment
- **Proprietà che il processo child non eredita dal processo parent:**
  - valore di ritorno di **fork**
  - process ID
  - process ID del processo parent
  - file locks
  - l'insieme dei segnali in attesa viene svuotato

Terminazione di un processo

- **Esistono tre modi per terminare normalmente:**
  - eseguire un return da main; equivalente a chiamare **exit**()
  - chiamare la funzione **exit**:
    - invocazione di tutti gli exit handlers che sono stati registrati
    - chiusura di tutti gli I/O stream standard
    - specificata in ANSI C; non completa per POSIX
  - chiamare la system call **_exit**
    - si occupa dei dettagli POSIX-specific;
    - chiamata da **exit**

Terminazione

- **Esistono due modi per terminare in modo anormale:**
  - Ricevere segnali
    - Generati dal processo stesso
    - Generati da altri processi
    - Generati dal kernel
  - Chiamando abort() → genera il segnale SIGABRT
- In ogni caso, l'azione del kernel e' lo stessa:
  - Rimozione della memoria utilizzata dal processo
  - Chiusura dei descrittori aperti

Terminazione

- **Cosa succede se il parent termina prima del child?**
  - si vuole evitare che un processo divenga "orfano"
  - il processo child viene "adottato" dal processo **init** (pid 1)
  - meccanismo: quando un processo termina, si esamina la tabella dei processi per vedere se aveva figli; in caso affermativo, il parent pid viene posto uguale a 1
- **Cosa succede se il child termina prima del parent?**
  - se il child termina, il parent non avrebbe più modo di ottenere informazioni sul termination/exit status del child
  - per questo motivo, alcune informazioni sul child vengono mantenute in memoria e il processo diventa uno *zombie*

Terminazione

- **Stato ‘zombie’**
  - vengono mantenute le informazioni che potrebbero essere richieste dal processo parent tramite **wait** e **waitpid**
    - Process ID
    - Termination status
    - Accounting information (tempo impiegato dal processo)
  - il processo resterà uno zombie fino a quando il parent non eseguirà una delle system call **wait**
- **Processi figli del processo init:**
  - non possono diventare zombie
  - tutte le volte che un child di **init** termina, **init** esegue una chiamata **wait** e raccoglie eventuali informazioni
  - in questo modo gli zombie vengono eliminati

Esecuzione dei processi

- ***System Call EXEC***
- **Quando un processo chiama una delle system call exec**
  - il processo viene rimpiazzato completamente da un nuovo programma (text, data, heap, stack vengono sostituiti)
  - il nuovo programma inizia a partire dalla sua funzione main
  - il process id non cambia
- **Esistono sei versioni di exec:**
  - con/senza gestione della variabile di ambiente **PATH**
  - Se viene gestita la variabile d'ambiente, un comando corrispondente ad un singolo filename verrà cercato nel **PATH**
  - con variabili di ambiente ereditate / con variabili di ambiente specificate
  - con array di argomenti / con argomenti nella chiamata (**NULL** terminated)

Esecuzione dei processi

- **System call:**

```
#include <unistd.h>

int execl(char *pathname, char *arg0, …);

int execv(char *pathname, char *argv[]);

int execle(char *pathname, char *arg0, …, char* envp[]);

int execve(char *pathname, char *argv[] , char* envp[]);

int execlp(char *filename, char *arg0, …);

int execvp(char *filename, char *argv[]);
```

- **Nota:**
  - Normalmente una sola di queste è una system call, le altre sono chiamate di libreria

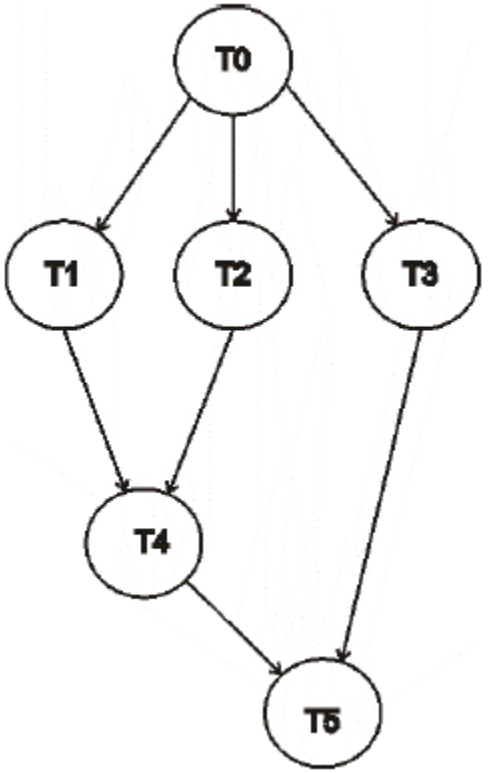
Esecuzione dei processi

Funzione	pathname	filename	arg list	argv[ ]	environ	envp[ ]
`execl`	●		●		●	
`execlp`		●	●		●	
`execle`	●		●			●
`execv`	●			●	●	
`execvp`		●		●	●	
`execve`	●			●		●
lettere		p	l	v		e

Esecuzione dei processi

- **Cosa viene ereditato da exec?**
  - process ID e parent process ID
  - real uid e real gid
  - supplementary gid
  - process group ID
  - session ID
  - terminale di controllo
  - current working directory
  - root directory
  - maschera creazione file (umask)
  - file locks
  - maschera dei segnali
  - segnali in attesa
- **Cosa non viene ereditato da exec?**
  - effective user id e effective group id
  - vengono settati in base ai valori dei bit di protezione

Implementiamo il seguente grafo di processi concorrenti



Codice con fork/waitpid

```
#include <sys/wait.h> #include <stdlib.h> #include <unistd.h> #include <stdio.h>
int main(int argc, char *argv[])
{
    pid_t cpid, w, t1,t2,t3,t4,t5;
    int status;
    t1 = fork();
    if (t1 == 0) {            /* codice del figlio */
        printf("processo T1; PID %ld\n", (long) getpid()); sleep(1);  _exit(0);
    } else {                  /* codice del padre */
        t2 = fork();
        if (t2 == 0) {            /* figlio */
          printf("processo T2; PID %ld\n", (long)getpid()); sleep(1); _exit(0);
        } else {
            t3 = fork();
            if (t3 == 0) {            /* figlio */
               printf("processo T3; PID %ld\n", (long)getpid()); sleep(1);  _exit(0);
         } else {
```

Codice con fork/waitpid (cont.)

```
        waitpid(t1, &status,0); printf("sono il padre, ho aspettato t1\n");
        waitpid(t2, &status,0); printf("sono il padre, ho aspettato t2\n");
          t4 = fork();
         if (t4 == 0) {            /* figlio */
          printf("processo T4; PID %ld\n", (long)getpid()); sleep(1); _exit(0);
           } else {
              waitpid(t4, &status,0); printf("sono il padre, ho aspettato t4\n");
              waitpid(t3, &status,0);  printf("sono il padre, ho aspettato t3\n");
              t5 = fork();
              if (t5 == 0) {            /* figlio */
                   printf("proc. T5; PID %ld\n",(long)getpid()); sleep(1); _exit(0);
              } else {                   /* padre */
                   waitpid(t5, &status,0); printf("sono il padre, ho atteso t5\n");
                }
           }
             }
       }
    }
}
```