

Laboratorio di Sistemi Operativi

Marzo-Giugno 2008
matricole congrue 0 mod 3

Controllo dei processi - I

Controllo dei processi

- ▶▶ Creazione di nuovi processi
- ▶▶ Esecuzione di programmi
- ▶▶ Processo di terminazione
- ▶▶ Altro...

Identificatori di processi

- ▶▶ Ogni processo ha un identificatore unico non negativo
- ▶▶ Process Id = 1 → il cui file di programma è contenuto in **/sbin/init**
 - ▶ invocato dal kernel alla fine del boot, legge il file di configurazione **/etc/inittab** dove ci sono elencati i file di inizializzazione del sistema (rc files) e dopo legge questi rc file portando il sistema in uno stato predefinito (multi user)
 - ▶ non muore mai.
 - ▶ è un processo utente (cioè non fa parte del kernel) di proprietà di root e ha quindi i privilegi del superuser

Identificatori di processi

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

```
pid_t getpid (void);      process ID del processo chiamante
pid_t getppid (void);    process ID del padre del processo chiamante
```

```
uid_t getuid (void);     real user ID del processo chiamante
uid_t geteuid (void);    effective user ID del processo chiamante
```

```
gid_t getgid (void);     real group ID del processo chiamante
gid_t getegid (void);    effective group ID del processo chiamante
```

```

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

int main(void)
{
    printf("pid del processo = %d\n", getpid() );
    return (0);
}

```

Creazione di nuovi processi

- ▶ L'unico modo per creare nuovi processi è attraverso la chiamata della funzione **fork** da parte di un processo già esistente
- ▶ quando viene chiamata, la **fork** genera un nuovo processo detto **figlio**
- ▶ dalla **fork** si ritorna **due** volte
 - ▶ il valore restituito al processo **figlio** è 0
 - ▶ il valore restituito al **padre** è il **pid** del **figlio**
 - un processo può avere più figli e non c'è nessuna funzione che può dare al padre il pid dei suoi figli
- ▶ **figlio** e **padre** continuano ad eseguire le istruzioni che seguono la chiamata di **fork**

Creazione di nuovi processi

- ▶ il figlio è una copia del padre
- ▶ condividono *dati*, *stack* e *heap*
 - ▶ il kernel li protegge settando i permessi *read-only*
- ▶ solo se uno dei processi tenta di modificare una di queste regioni, allora essa viene copiata (*Copy On Write*)
- ▶ in generale non si sa se il figlio è eseguito prima del padre, questo dipende dall'algoritmo di scheduling usato dal kernel

Laboratorio di Sistemi Operativi

7

Funzione fork

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

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

```
pid_t fork(void);
```

Restituisce: 0 nel figlio,
pid del figlio nel padre
-1 in caso di errore

Laboratorio di Sistemi Operativi

8

```

#include <sys/types.h>
int glob=10; /* dati inizializzati */
char buf [ ]="Scritta su stdout\n";
int main(void)
{
    int var=100; /* vbl sullostack */
    pid_t pippo;
    write(STDOUT_FILENO, buf, sizeof(buf)-1);
    printf("prima della fork\n");
    if( (pippo=fork()) == 0) {glob++; var++;}
    else sleep(2);
    printf("pid=%d, glob=%d, var=%d\n",getpid(),glob,var);
    exit(0);
}

```

```

$ a.out
Scritta su stdout
prima della fork
pid=227, glob=11, var=101
pid=226, glob=10, var=100
$
$ a.out>temp.txt
$ cat temp.txt
    Scritta su stdout
    prima della fork
    pid=229, glob=11, var=101
    prima della fork
    pid=228, glob=10, var=100
$

```

Per avere il pid del padre:

Aggiungiamo la riga:

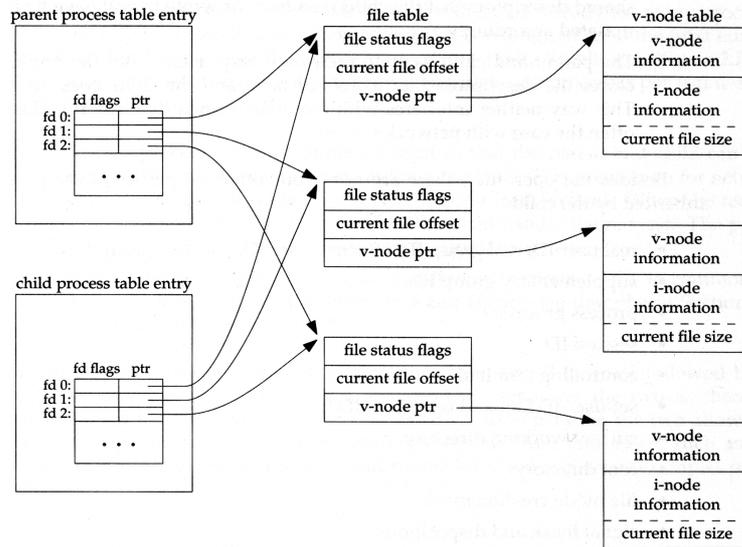
```
▶▶ printf("pid padre = %d\n", getppid());
```

Cosa possiamo aspettarci che succeda?

Condivisione file

- ▶▶ nel programma precedente anche lo standard output del **figlio** è ridiretto...infatti
 - ▶ **tutti** i file descriptor aperti del padre sono duplicati nei figli
- ▶▶ **padre** e **figlio** condividono una entry nella tavola dei file per ogni file descriptor aperto
- ▶▶ problema della sincronizzazione:
 - ▶ chi scrive per prima? oppure è intermixed
 - ▶ nel programma abbiamo messo *sleep(2)*

Condivisione files tra padre/figlio



Problema della sincronizzazione

- ▶▶ il padre aspetta che il figlio termini
 - ▶ I current offset dei file condivisi vengono eventualmente aggiornati dal figlio ed il padre si adegua
- ▶▶ o viceversa
- ▶▶ tutti e due chiudono i file descriptor dei file che non gli servono, così non si danno fastidio
- ▶▶ ci sono altre proprietà ereditate da un figlio:
 - ▶ uid, gid, euid, egid
 - ▶ suid, sgid
 - ▶ cwd
 - ▶ file mode creation mask
 - ▶ ...

conclusioni: quando si usa **fork**?

- un processo attende richieste (p.e. da parte di client nelle reti) allora si duplica
 - ▶ il figlio tratta (handle) la richiesta
 - ▶ il padre si mette in attesa di nuove richieste

- un processo vuole eseguire un nuovo programma (p.e. la shell si comporta così) allora si duplica e il figlio lo esegue