

Gestione dei Processi

Laboratorio Sistemi Operativi

Antonino Staiano
Email: antonino.staiano@uniparthenope.it

Processo

- Un programma è costituito da istruzioni e dati ed è memorizzato in un file
- Un **processo** è un programma in esecuzione

Avvio di un processo

- Chiamato da una shell o da un altro programma in esecuzione
- Quando si esegue un programma si esegue prima una routine di start-up speciale, specificata come indirizzo di partenza del programma eseguibile (impostato dal linker), che prende
 - valori passati dal kernel in `argv[]` dalla linea di comando
 - variabili d'ambiente
- Successivamente è chiamata la funzione `main`
 - Un programma C inizia l'esecuzione con una funzione chiamata `main`, il cui prototipo è:

```
int main(int argc, char *argv[])
```

- `argc` è il numero di argomenti
- `argv` è un array di puntatori agli argomenti

Terminazione di un processo

- Esistono otto modi per terminare un processo
 - Terminazione **normale**
 - Ritorno dal main
 - Chiamata di `exit`
 - Chiamata di `_exit` o `_Exit`
 - Ritorno dell'ultimo thread dalla sua routine di avvio
 - Chiamata di `pthread_exit` dall'ultimo thread
 - Terminazione **anomala**
 - Chiamata di `abort`
 - Ricezione di un segnale
 - Risposta dell'ultimo thread ad una richiesta di cancellazione
- N.B.: la routine di avvio fa in modo che quando la funzione `main` ritorna venga chiamata `exit`

[5]

Funzioni di uscita

- Sono tre le funzioni che terminano un programma normalmente
 - `_exit` (chiamata di sistema) ed `_Exit` (libreria standard) che ritornano al kernel immediatamente
 - `exit` (libreria standard) che prima esegue una procedura di "pulizia" e poi ritorna al kernel

```
#include <stdlib.h>
void exit (int status)
void _Exit(int status)

#include <unistd.h>
void _exit(int status)
```

- N.B.: la ragione per gli header differenti è dovuta al fatto che `exit` e `_Exit` sono specificate da ISO C, mentre `_exit` da POSIX.1

[6]

Funzioni di uscita

- Storicamente, la funzione `exit` esegue sempre una terminazione pulita della libreria di I/O
 - Tutti gli stream aperti sono chiusi con `fclose`
- Tutte e tre le funzioni `exit` ricevono un argomento intero (`exit status`)
- Le shell dei sistemi Unix forniscono un modo per esaminare lo stato di uscita di un processo
- Lo stato di uscita è indefinito se
 - le funzioni di uscita sono chiamate senza alcun codice di uscita
 - `main` fa un `return` senza valore di ritorno
 - il `main` non è dichiarato per restituire un intero
 - Se `main` è dichiarato per restituire un intero e si ha un ritorno implicito, allora lo stato di uscita del processo è 0

[7]

Esempio

```
#include <stdio.h>
main ()
{
    printf("Hello, World\n");
}
```

- Compilando ed eseguendo il programma, osserviamo un codice di uscita casuale
 - Compilando lo stesso programma su sistemi differenti otteniamo codici di uscita differenti a seconda del contenuto dello stack e dei registri al momento in cui la funzione `main` restituisce il controllo

```
$ gcc hello.c
$ ./a.out
Hello, World
$ echo $?
13
```

[8]

Funzioni di uscita

- Restituire un valore intero dalla funzione `main` equivale a chiamare `exit` con lo stesso valore
- Richiamare
 - `return (0);`e equivalente a richiamare
 - `exit(0);`dalla funzione `main`

Funzione atexit

- ISO C consente ad un processo di registrare almeno 32 funzioni chiamate automaticamente quando è invocata `exit`
 - Tali funzioni sono chiamate `exit handler` e sono registrate chiamando la funzione `atexit()`
- ```
#include<stdlib.h>
int atexit (void(*func)(void));
```
- Restituisce 0 se OK, <>0 in caso di errore
- Si passa l'indirizzo di una funzione come argomento
    - La funzione non riceve alcun argomento e non restituisce nulla
  - Quando si invoca la `exit` questa chiama le funzioni nell'ordine inverso rispetto a quello in cui sono state registrate
  - Con ISO C e POSIX, `exit` prima chiama gli `exit handler` e poi chiude tutti gli stream aperti

# Esempio

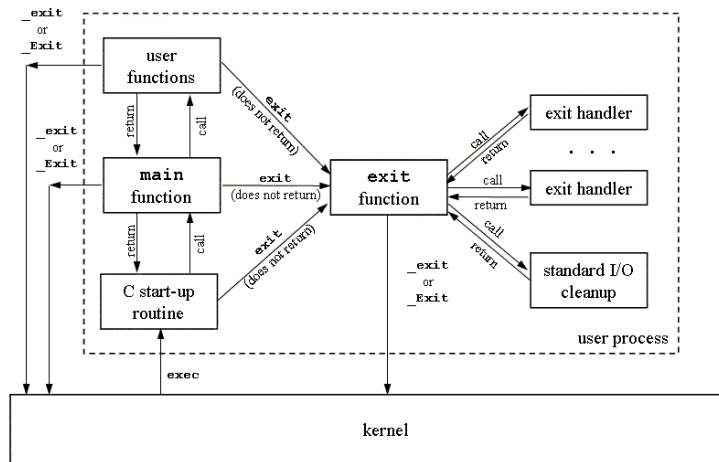
```
#include "apue.h"
static void my_exit1(void);
static void my_exit2(void);
int main (void){
 if (atexit(my_exit2)!=0)
 err_sys("Non posso registrare my_exit2");

 if (atexit(my_exit1)!=0)
 err_sys("Non posso registrare my_exit1");
 if (atexit(my_exit1)!=0)
 err_sys("Non posso registrare my_exit1");
 print("Main ha completato\n");
 return(0);
}
static void my_exit1(void){
 printf("Primo exit handler\n");
}
static void my_exit2(void){
 printf("Secondo exit handler\n");
}
```

# Esempio (cont.)

```
$./a.out
main ha completato
Primo exit handler
Primo exit handler
Secondo exit handler
```

# Inizio e fine di un programma C



[ 13 ]

# Argomenti dalla linea di comando

- Quando è eseguito un programma, il processo che esegue l'exec può passare argomenti da linea di comando al nuovo programma

```

#include "apue.h"
int main (int argc, char *argv[])
{
 int i;
 for (i=0; i<argc; i++)
 printf("argv[%d]: %s\n",i,argv[i]);
 exit(0);
}

```

```

$./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo

```

N.B.: Sia ISO C che POSIX.1 garantiscono che `argv[argc]` sia un puntatore a `NULL`. Per cui il ciclo potrebbe essere riscritto come:  
`for(i=0; argv[i]!=NULL; i++)`

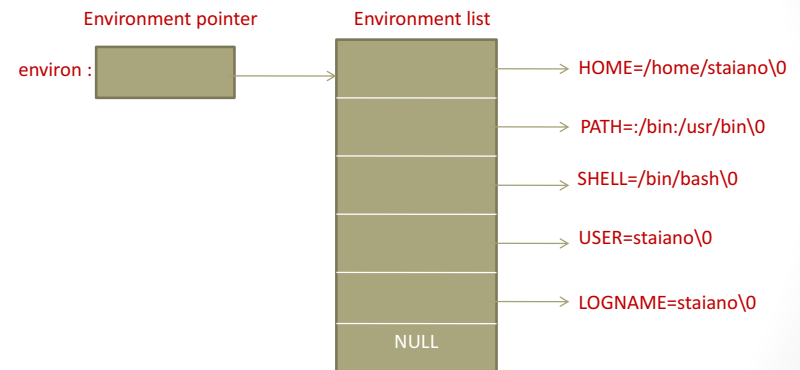
[ 14 ]

# Environment List

- Ad ogni programma è passata una lista dell'ambiente
  - Array di puntatori a stringhe
  - Ogni puntatore contiene l'indirizzo di una stringa C terminata con null (\0)
  - L'indirizzo dell'array di puntatori è contenuto nella variabile globale `environ`:
    - `extern char **environ;`
- Per convenzione l'ambiente consiste delle stringhe  
 nome = valore (ad esempio, `HOME=/home/staiano\0`)

[ 15 ]

# Ambiente di 5 stringhe di caratteri in C



[ 16 ]

## Environment List

- Storicamente, molti sistemi Unix forniscono un terzo argomento alla funzione main, cioè l'indirizzo della lista dell'ambiente

```
int main (int argc, char *argv[], char *envp[]);
```

- ISO C specifica che la funzione main sia scritta con due argomenti
- POSIX.1 specifica che si debba usare `environ` anziché il terzo argomento, poiché il terzo argomento non comporta alcun vantaggio rispetto alla variabile globale `environ`

[ 17 ]

## Struttura della memoria per un programma C

- Un programma C è composto dai seguenti pezzi:
  - **Segmento di testo**: le istruzioni macchina eseguite dalla CPU
    - Condivisibile (una sola copia in memoria)
    - A sola lettura (protezione)
  - **Segmento di dati inizializzati**: contiene variabili globali e statiche iniziate nel programma (ad esempio `int maxcount = 99;`)
  - **Segmento di dati non inizializzati** (bss, "block started by symbol"): le variabili globali e statiche sono iniziate dal kernel a 0 o al puntatore nullo prima dell'esecuzione (ad esempio, `long sum[1000];`)

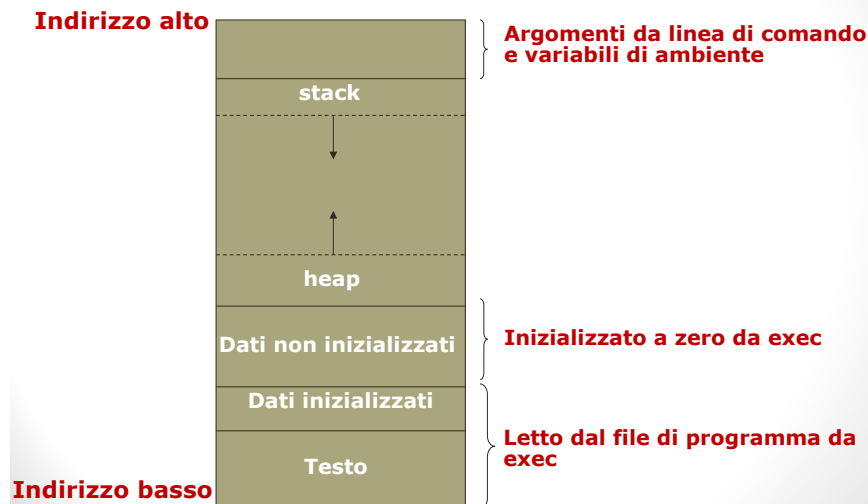
[ 18 ]

## Struttura della memoria per un programma C

- **Stack**: contiene le variabili automatiche con le informazioni salvate ogniqualvolta è chiamata una funzione
  - Indirizzo di ritorno, registri
  - La funzione chiamata alloca spazio per le sue variabili automatiche e temporanee
- **Heap**: luogo in cui avviene l'allocazione dinamica della memoria (tra il segmento dati non inizializzato e lo stack)

[ 19 ]

## Struttura della memoria per un programma C



[ 20 ]

# Struttura della memoria per un programma C

- Il comando `size` riporta la dimensione (in byte) dei segmenti di testo, dati e bss
- Ad esempio:

```
$ size /usr/bin/gcc /bin/bash
```

| <b>text</b> | <b>data</b> | <b>bss</b> | <b>dec</b> | <b>hex</b> | <b>filename</b> |
|-------------|-------------|------------|------------|------------|-----------------|
| 207716      | 3448        | 2560       | 213724     | 342dc      | /usr/bin/gcc    |
| 713988      | 37564       | 21776      | 773328     | bccd0      | /bin/bash       |

N.B.: la quarta e quinta colonna corrispondono al totale delle tre dimensioni espresse in decimale ed esadecimale, rispettivamente

[ 21 ]

# Allocazione della memoria

- ISO C specifica tre funzioni
  - `malloc`: alloca un numero specificato di byte di memoria
  - `calloc`: alloca spazio per uno specifico numero di oggetti di dimensione specificata
  - `realloc`: incrementa o decrementa la dimensione di un'area di memoria allocata in precedenza

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc (size_t nobj, size_t size);
void *realloc (void *ptr, size_t newsiz);
```

```
void free (void *ptr);
```

N.B.: tutte restituiscono un puntatore non nullo se tutto va a buon fine, `NULL` in caso di errore

[ 22 ]

# Allocazione della memoria

- Le routine di allocazione sono implementate con la system call `sbrk`
  - Espande o contrae l' heap del processo
- Molte implementazioni allocano un poco più spazio di quanto richiesto per memorizzare varie informazioni, tra cui:
  - Dimensione del blocco allocato
  - Puntatore al successivo blocco allocato

[ 23 ]

# Allocazione della memoria

- Sorgenti di errore
  - Scrivere prima della fine di un'area allocata può sovrascrivere le informazioni relative ad un blocco successivo
  - Liberare un blocco già liberato da una chiamata a `free`
  - Chiamare `free` con un puntatore non ottenuto da una delle tre funzioni di allocazione
  - Se un processo chiama `malloc` e dimentica di chiamare `free` l'uso di memoria continua a crescere

[ 24 ]

# Controllo dei Processi

## Unix e i processi

- Unix è una famiglia di sistemi multiprogrammati basati su processi
- Un processo consiste nell'insieme di eventi che scaturiscono durante l'esecuzione di un programma
  - E' un'entità dinamica a cui è associato un insieme di informazioni necessarie per la corretta esecuzione e gestione del processo da parte del sistema operativo
- Il processo Unix mantiene spazi di indirizzamento separati per i dati e per il codice
  - Ogni processo ha uno spazio di indirizzamento dei dati privato
    - Non è possibile condividere variabili tra processi diversi!
    - E' necessaria un'interazione basata su scambi di messaggi
  - A differenza dello spazio di indirizzamento dati, lo spazio di indirizzamento del codice è condivisibile
    - Più processi possono eseguire lo stesso programma facendo riferimento alla stessa area di codice nella memoria centrale

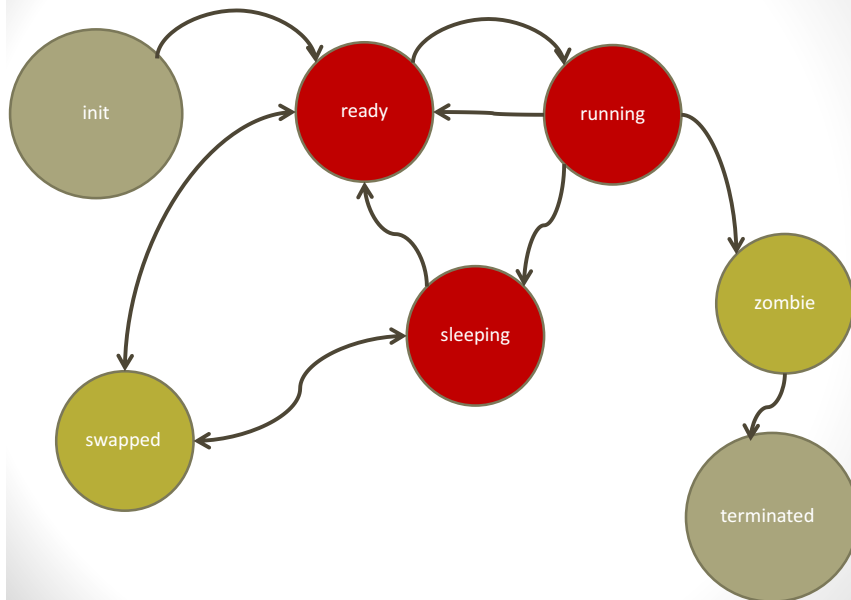
## Caratteristiche del processo Unix

- Processo pesante con codice rientrante
  - Dati non condivisi
  - Codice condivisibile con altri processi
- Funzionamento in doppia modalità
  - Processi utente (modalità utente)
  - Processi di sistema (modalità kernel)

## Stati di un processo Unix

- Come nel caso generale
  - **Init**: caricamento in memoria del processo e inizializzazione delle strutture del SO
  - **Ready**: processo pronto
  - **Running**: il processo usa la CPU
  - **Sleeping**: il processo è sospeso in attesa di un evento
  - **Terminated**: deallocazione del processo dalla memoria
- Inoltre
  - **Zombie**: il processo è terminato ma è in attesa che il padre ne rilevi lo stato di terminazione
  - **Swapped**: il processo (o parte di esso) è temporaneamente trasferito in memoria secondaria

## Stati di un processo Unix (2)



## Processi swapped

- Lo scheduler a medio termine (**swapper**) gestisce i trasferimenti dei processi
  - Da memoria centrale a secondaria (**swap out**)
    - Si applica, preferibilmente, ai processi bloccati (sleeping) prendendo in considerazione tempo di attesa, di permanenza in memoria e dimensione del processo (preferibilmente i processi più lunghi)
  - Da memoria secondaria a centrale (**swap in**)
    - Si applica preferibilmente ai processi più corti

## Rappresentazione dei processi

- Il codice dei processi è rientrante, vale a dire, più processi possono condividere lo stesso codice (segmento di testo)
  - Codice e dati sono separati
  - Il SO gestisce una struttura dati globale in cui sono contenuti i puntatori ai segmenti di testo (eventualmente condivisi) dai processi
    - **Text table**
  - L'elemento della text table si chiama **text structure** e contiene tra gli altri
    - Puntatore al segmento di testo (se il processo è in stato di swap, il riferimento alla memoria secondaria)
    - Numero dei processi che lo condividono

## Rappresentazione dei processi

- Il Process Control Block (**PCB**) è rappresentato da due strutture dati
  - **Process structure**: informazioni necessarie al sistema per la gestione del processo (a prescindere dal suo stato)
  - **User structure (u-area)**: informazioni necessarie solo se il processo è residente in memoria centrale



# Process e User Structure

| Process structure                                                                                                                                                                                                                                                                                                                                           | User Structure                                                                                                                                                                                                                                                                                                                             |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>• PID</li> <li>• Stato del processo</li> <li>• Riferimento ad aree dati e stack</li> <li>• Riferimento indiretto al codice</li> <li>• PID del processo padre</li> <li>• Priorità del processo</li> <li>• Riferimento al prossimo processo in coda</li> <li>• Puntatore alla User structure</li> <li>• ...</li> </ul> | <ul style="list-style-type: none"> <li>• Una copia dei registri di CPU</li> <li>• Informazioni sulle risorse allocate (file aperti)</li> <li>• Informazioni sulla gestione di eventi asincroni (segnali)</li> <li>• Directory corrente</li> <li>• Proprietario</li> <li>• Gruppo</li> <li>• Argc/argv, PATH, ...</li> <li>• ...</li> </ul> |

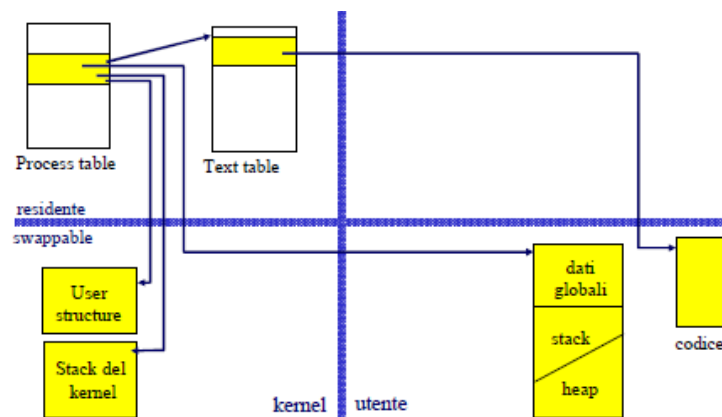
[ 33 ]

# Immagine di un processo

- L'immagine di un processo è l'insieme delle aree di memoria e delle strutture dati associate al processo
- Non tutta l'immagine è accessibile in modo user
  - Parte di kernel
  - Parte di utente
- Ogni processo può essere soggetto a swapping
  - Non tutta l'immagine può essere trasferita in memoria
    - Parte swappable
    - Parte residente o non swappable

[ 34 ]

# Immagine di un processo (2)



[ 35 ]

# Immagine di un processo Unix

- **Process structure** (kernel, residente)
  - è l'elemento della process table associato al processo
- **Text structure** (kernel, residente)
  - elemento della text table associato al codice del processo
- **Area dati globali utente** (user, swappable)
  - Segmento dati inizializzati
  - Segmento dati non inizializzati
- **Stack, heap utente** (user, swappable)
  - aree dinamiche associate al programma eseguito
- **Stack del kernel** (kernel, swappable)
  - stack di sistema associato al processo per le chiamate a system call
- **U-area** (kernel, swappable)
  - struttura dati contenente i dati necessari al kernel per la gestione del processo quando è residente

[ 36 ]

# Processi

- All'avvio del SO c'è un solo processo utente visibile chiamato `init()`, il cui identificativo numerico unico è sempre 1 (`init()` è invocato dal kernel alla fine della procedura di bootstrap)
  - Nelle precedenti versioni di Unix il file si trovava in `/etc`, nelle più recenti si trova in `/sbin`
- Quindi `init()` è l'antenato comune di tutti i processi utenti esistenti in un dato momento nel sistema
- Esempio:
  - `init()` crea i processi `getty()` responsabili di gestire i login degli utenti

[ 37 ]

# Processi (cont.)

- Il processo con ID 0 è lo scheduler noto anche come `swapper`
  - A tale processo non corrisponde alcun programma su disco poiché è parte del kernel ed è, dunque, un `processo di sistema`
- Ogni implementazione di Unix ha i propri processi kernel che forniscono i servizi del sistema operativo
  - Ad esempio, il processo con ID 2 è il `pagedaemon` che è responsabile della paginazione del sistema della memoria virtuale

[ 38 ]

# La chiamata di sistema `fork()`

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

- Esempio: `esito = fork()`
  - Crea una copia del processo che esegue la `fork`
  - L'area dati viene duplicata, l'area codice viene condivisa
  - Il processo creato (figlio) riceve `esito = 0`
  - Il processo creante (padre) riceve `esito > 0` che corrisponde all'identificatore (PID) di processo del processo creato
  - Se l'operazione fallisce
    - `esito = -1`
    - alla variabile `errno` viene assegnato il codice relativo all'errore. La `fork()` può fallire se, per esempio, la tabella dei processi è piena e non c'è più spazio per allocare nuovi descrittori di processo
  - N.B.: `fork()` è invocata da un processo ma restituisce il controllo a due processi

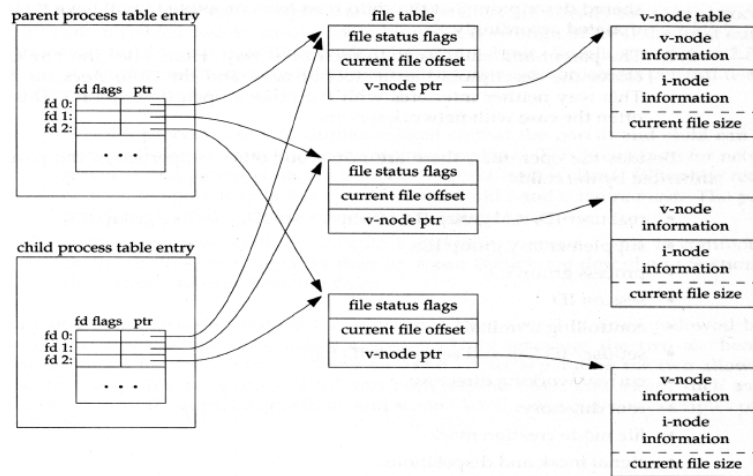
[ 39 ]

# La chiamata di sistema `fork()` (cont.)

- Il processo figlio è una copia del genitore (spazio dei dati, heap e stack), cioè essi non condividono parti di memoria
- PID e PPID nei processi padre e figlio sono differenti
- Una volta invocata una `fork` non si può sapere se il figlio andrà in esecuzione prima del genitore o dopo
- Tutti i descrittori aperti nel genitore sono duplicati nel figlio. Nella tabella dei file, il padre ed il figlio condividono lo stesso elemento per ogni descrittore aperto, condividono cioè lo stesso offset

[ 40 ]

## Descrittori file aperti padre duplicati nel figlio



[ 41 ]

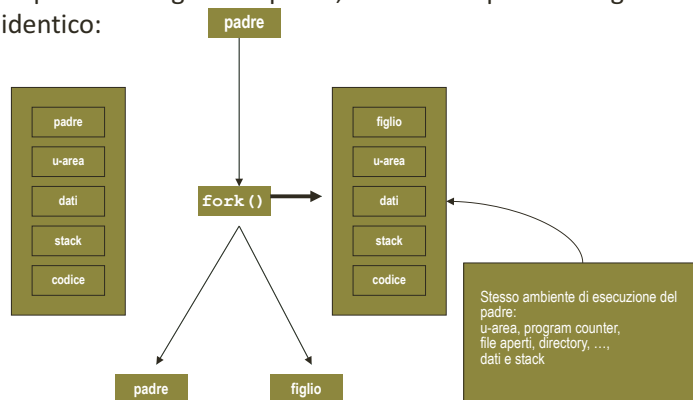
## Creazione di processi

- Quando un processo è duplicato, il processo padre ed il processo figlio sono virtualmente identici
  - il codice, i dati e lo stack del figlio sono una copia di quelli del padre ed il processo figlio continua ad eseguire lo stesso codice del padre
  - differiscono per alcuni aspetti quali PID, PPID e risorse a run-time (es. segnali pendenti)
- Quando un processo figlio termina (tramite una `exit()`), la sua terminazione è comunicata al padre (tramite un segnale) e questi si comporta di conseguenza

[ 42 ]

## Creazione di un processo figlio: `fork()`

- Duplica l'immagine del padre, creando un processo figlio identico:



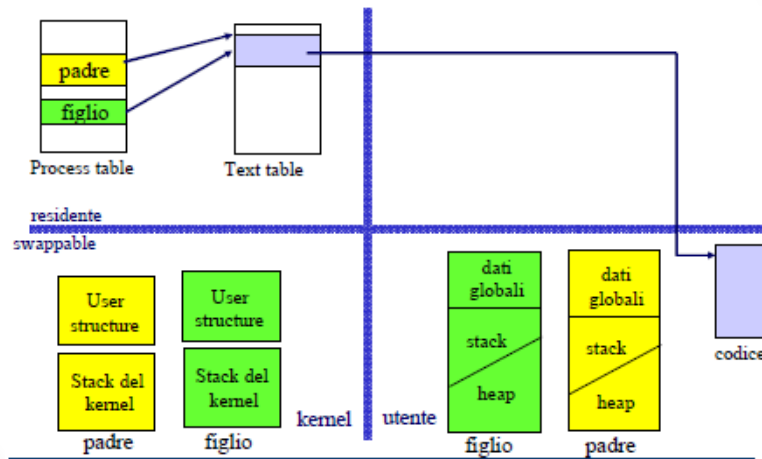
[ 43 ]

## Effetti della `fork()`

- Allocazione di una **nuova process structure** nella process table associata al processo figlio e sua inizializzazione
- Allocazione di una **nuova user structure** nella quale viene copiata la user structure del padre
- Allocazione dei **segmenti di dati e stack** del figlio nei quali vengono copiati dati e stack del padre
- Aggiornamento della **text structure** del codice eseguito (condiviso col padre): incremento del contatore dei processi, etc.

[ 44 ]

## Effetti della fork()



[ 45 ]

## Ottenere il PID: getpid() e getppid()

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

- getpid() restituisce il PID del processo invocante
- getppid() restituisce il PPID (cioè il PID del padre) del processo invocante
- Hanno sempre successo
- Il PPID di init (il processo con PID 1) è ancora 1

[ 46 ]

## Esempio: padre e figlio eseguono l'assegnamento x = 1 dopo il ritorno dalla fork

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
 int x;
 x = 0;
 fork();
 x = 1;
 printf("process %d, x = %d\n", getpid(), x);
 return 0;
}
```

[ 47 ]

## Esempio: creazione di una catena di n processi

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (int argc, char *argv[]) {
 pid_t childpid = 0;
 int i, n;
 if (argc != 2) { /* controllo argomenti */
 fprintf(stderr, "Uso: %s processi\n", argv[0]);
 return 1;
 }
 n = atoi(argv[1]);
 for (i = 1; i < n; i++)
 if (childpid = fork())
 break;
 printf("i:%d processo ID:%d padre ID:%d figlio ID:%d\n", i,
 getpid(), getppid(), childpid);
 exit(0);
}
```



[ 48 ]

# Identificativi di Processo

```
#include <unistd.h>
#include <sys/types.h>
pid_t getpid(void) process ID
pid_t getppid(void) parent process ID
uid_t getuid(void) real user ID
uid_t geteuid(void) effective user ID
gid_t getgid(void) real group ID
gid_t getegid(void) effective group ID
```

- Questi identificativi sono interi non negativi

[ 49 ]

# Esempio

```
/* Stampa vari user e group ID per un processo */

#include <stdio.h>
#include <unistd.h>
int main(void) {
 printf("Mio real user ID: %5d\n", (uid_t)getuid());
 printf("Mio effective user ID:%5d\n", (uid_t)geteuid());
 printf("Mio real group ID:%5d\n", (gid_t)getgid());
 printf("Mio effective group ID:%5d\n", (gid_t)getegid());
 return 0;
}
```

[ 50 ]

# Esempio: myfork.c

```
/* Un programma che si sdoppia e mostra il PID e PPID dei due processi componenti */
#include <stdio.h>
int main (int argc, char *argv[])
{
 int pid;

 printf ("Sono il processo di partenza con PID %d e PPID %d.\n",getpid(),getppid());

 pid = fork (); /* Duplicazione. Figlio e genitore continuano da qui */

 if (pid != 0) { /* pid diverso da 0, sono il padre */
 printf ("Sono il processo padre con PID %d e PPID %d.\n",getpid(),getppid());
 printf ("Il PID di mio figlio e' %d.\n", pid); /* non aspetta con wait(); */
 }
 else { /* il pid è 0, quindi sono il figlio */
 printf ("Sono il processo figlio con PID %d e PPID %d.\n",getpid(),getppid());
 }

 printf ("PID %d termina.\n",getpid());
 /* Entrambi i processi eseguono questa parte */
 return 0;
}
```

[ 51 ]

# Esempio: myfork.c

```
$./myfork
Sono il processo di partenza con PID 724 e PPID 572.
Sono il processo padre con PID 724 e PPID 572.
Sono il processo figlio con PID 725 e PPID 724.
PID 725 termina.
IL PID di mio figlio è 725
PID 724 termina.
$
```

- Nell'esempio, il padre non aspetta la terminazione del figlio per terminare a sua volta
- Se un padre termina prima di un suo figlio, il figlio diventa orfano e viene automaticamente adottato dal processo init()

[ 52 ]

## Esempio: le modifiche alle variabili del processo figlio non si estendono ai valori delle variabili del processo padre

```
#include "apue.h"
int glob=6; /* variabile esterna (blocco dati inizializzati) */
char buf[] = "una write sullo stdout\n";
int main(void)
{
 int var; /* variabile automatica sullo stack */
 pid_t pid;
 var = 88;
 if (write(STDOUT_FILENO,buf, sizeof(buf)-1) != sizeof(buf)-1)
 err_sys("errore della write");
 printf("prima della fork\n");

 if ((pid = fork())<0) {
 err_sys("errore della fork");
 } else if (pid ==0){ /* figlio */
 glob++; /* modifica le variabili */
 var++;
 } else {
 sleep(2); /* padre */
 }
 printf("pid = %d, glob = %d, var = %d\n",getpid(),glob,var);
 exit(0);
}
```

53

## Esempio (cont.)

```
$./a.out
una write sullo stdout
prima della fork
pid = 430, glob = 7, var = 89
 le variabili del figlio sono cambiate
pid = 429, glob = 6, var = 88
 la copia del padre non è cambiata
$./a.out > temp.out
$ cat temp.out
una write sullo stdout
prima della fork
pid = 432, glob = 7, var = 89
prima della fork
pid = 431, glob = 6, var = 88
```

54

## Esempio (cont.)

- La **write** non è bufferizzata, i dati sono scritti un'unica volta
  - È invocata prima della **fork**
- La libreria standard di I/O è bufferizzata
  - In particolare, lo standard output è bufferizzato per linea se è connesso ad un terminale, altrimenti è totalmente bufferizzato
- Esecuzione interattiva
  - Solo una copia della linea della **printf**
    - Il buffer è scaricato (flushed) dal newline
- Esecuzione rediretta
  - Due copie della riga della **printf**
    - La **printf** prima della **fork** è chiamata una sola volta, ma la riga resta nel buffer quando è chiamata la **fork**
    - Il buffer è copiato nel figlio
    - Padre e figlio hanno il buffer riempito con questa linea
    - La seconda **printf**, prima della **exit**, aggiunge i suoi dati al buffer esistente
    - Infine, quando termina ciascun processo, le copie dei buffer sono scaricate

55

## Ulteriori informazioni sulla **fork()**

- Il segmento di testo (e solo esso!) dei processi padre e figlio è condiviso e tenuto in modalità a sola lettura per il padre ed i suoi figli
- Per gli altri segmenti, Linux utilizza la tecnica del copy on write: viene effettivamente copiata una pagina di memoria per il nuovo processo solo quando ci viene effettuata sopra una scrittura
- Il meccanismo di creazione di un nuovo processo è molto più efficiente, non essendo necessaria la copia di tutto lo spazio degli indirizzi virtuali del padre, ma solo delle pagine di memoria che sono state modificate e solo al momento della modifica stessa

56

# La chiamata di sistema vfork()

```
#include <unistd.h>
pid_t vfork(void);
```

- **vfork** crea un nuovo processo, esattamente come **fork**, ma senza copiare lo spazio di indirizzamento. Fino a che il figlio non esegue una **exec** o **exit**, esso viene eseguito nello spazio di indirizzamento del genitore
- La **vfork** assicura che il figlio venga eseguito per primo, fino a quando questi non chiama **exec** o **exit**. La funzione **vfork** viene utilizzata quando il processo generato ha lo scopo di eseguire (**exec**) un nuovo programma

[ 57 ]

# Esempio: vfork()

```
#include "apue.h"
int glob=6; /* variabile esterna (blocco dati inizializzati) */
int main(void)
{
 int var; /* variabile automatica sullo stack */
 pid_t pid;
 var = 88;
 printf("prima della fork\n");

 if ((pid = vfork())<0) {
 err_sys("errore della vfork");
 } else if (pid ==0) { /* figlio */
 glob++; /* modifica le variabili */
 var++;
 _exit(0); /* il figlio finisce */
 }
 /* il padre continua qui */
 printf("pid = %d, glob = %d, var = %d\n",getpid(),glob,var);
 exit(0);
}
```

[ 58 ]

# Esempio: vfork (cont.)

```
$./a.out
Prima della vfork
Pid = 29039, glob = 7, var = 89
```

- L'incremento delle variabili fatte dal figlio modifica i valori nel genitore

[ 59 ]

# Ulteriori informazioni sulla vfork()

- Non viene creata la tabella delle pagine né la struttura dei task per il nuovo processo. Il processo padre è posto in attesa fintanto che il figlio non ha eseguito una **execve** o non è uscito con una **\_exit**
- Il figlio condivide la memoria del padre (le modifiche della stessa possono avere effetti imprevedibili) e non deve ritornare o uscire con una **exit** ma usare esplicitamente **\_exit**
  - Se il figlio invoca **exit** gli stream I/O sono scaricati ed eventualmente chiusi (dipende dall'implementazione), la **printf** eseguita dal processo padre restituisce (eventualmente) un errore poiché la memoria che rappresenta l'oggetto FILE per lo standard output sarà cancellato (eventualmente)
- Introdotta in BSD per migliorare le prestazioni poiché **fork** comportava la copia completa del segmento dati del processo padre
  - inutile appesantimento nei casi in cui la **fork** è chiamata solo per creare un figlio che esegue una **exec**
- Poiché Linux supporta la copy-on-write la perdita di prestazioni è assolutamente trascurabile e l'uso di questa funzione è deprecato

[ 60 ]



# Terminazione di processi in Unix

- La terminazione di un processo consiste in una serie di operazioni che lasciano il sistema in stato coerente
  - chiusura dei file aperti
  - rimozione dell'immagine dalla memoria
  - eventuale segnalazione al processo padre
- Per gestire quest'ultimo aspetto Unix impiega le system call **exit** e **wait** (o **waitpid**) in modo coordinato
  - terminazione dell'esecuzione di un processo (**exit**)
  - attesa della terminazione di un processo da parte del processo che lo ha creato (**wait**)

[ 61 ]

CoD in Informatica - Laboratorio di SO - A.A. 2016/2017 - Prof. Antonino Staiano

# Le chiamate di sistema wait e waitpid

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

- La funzione **wait** sospende il processo invocante finché:
  - un figlio ha terminato la propria esecuzione, oppure
  - riceve un segnale
- La funzione **waitpid** sospende il processo invocante finché:
  - il processo **pid** ha terminato la propria esecuzione, oppure
  - riceve un segnale
- Se il processo è uno zombie, le funzioni ritornano subito
- Entrambe ritornano con un errore (-1) se il processo non ha figli, altrimenti è restituito il **pid** del processo figlio terminato

[ 62 ]

CoD in Informatica - Laboratorio di SO - A.A. 2016/2017 - Prof. Antonino Staiano

# Le chiamate di sistema wait e waitpid (cont.)

- Il kernel notifica al genitore la terminazione di un processo figlio mediante il segnale **SIGCHLD**
- Le differenze tra le due funzioni sono:
  - se il processo invocante ha più di un figlio, **wait** ritorna quando uno qualsiasi di essi ha terminato; **waitpid** permette di controllare quale figlio aspettare
  - **wait** blocca il processo chiamante fino a quando il figlio non è terminato, mentre **waitpid** può non farlo

[ 63 ]

CoD in Informatica - Laboratorio di SO - A.A. 2016/2017 - Prof. Antonino Staiano

# Le chiamate di sistema wait e waitpid (cont.)

- Per entrambe le funzioni l'argomento **statloc** è un puntatore ad un intero
- Se l'argomento non è **NULL**, lo stato di terminazione è conservato nella locazione puntata dall'argomento
  - Il valore puntato dipende dall'implementazione e tradizionalmente alcuni bit (in genere 8) sono riservati per memorizzare lo stato di uscita ed altri per indicare il segnale che ha causato la terminazione (in caso di terminazione anomala)
- Lo stato di terminazione può essere rilevato utilizzando le macro definite in **<sys/wait.h>**

[ 64 ]

CoD in Informatica - Laboratorio di SO - A.A. 2016/2017 - Prof. Antonino Staiano



## Rilevazione dello stato

- se il byte **meno significativo** di `statloc` è 0, il byte **più significativo** rappresenta lo stato di terminazione (terminazione volontaria, ad esempio con `exit`)
- in caso contrario, il byte meno significativo di `statloc` descrive il segnale che ha terminato il figlio (terminazione involontaria)

## Argomenti di `waitpid`

- L'argomento `pid` di `waitpid` ha la seguente interpretazione:
  - `pid == -1` attende un qualsiasi figlio (uguale a `wait`)
  - `pid > 0` attende il processo che ha il process ID uguale a `pid`
  - `pid == 0` attende un qualsiasi figlio il cui process group ID è uguale a quello del processo chiamante
  - `pid < -1` attende un qualsiasi figlio il cui process group ID è uguale a quello del valore assoluto di `pid`

## Argomenti di `waitpid` (cont.)

- L'argomento `options` di `waitpid` è 0, oppure una combinazione in OR delle costanti:
  - **WNOHUNG** non bloccherà il processo invocante se il pid del figlio non è immediatamente disponibile (ritorna 0)
  - **WUNTRACED** ritorna lo stato di un figlio sospeso

## Rilevazione dello stato (cont.)

| Macro                                                              | Descrizione                                                            |
|--------------------------------------------------------------------|------------------------------------------------------------------------|
| <code>WIFEXITED(status)</code><br><code>WEXITSTATUS(status)</code> | vero se il figlio è terminato normalmente<br>ritorna lo stato          |
| <code>WIFSIGNALED(status)</code><br><code>WTERMSIG(status)</code>  | vero se il figlio è uscito a causa di un segnale<br>ritorna il segnale |
| <code>WIFSTOPPED(status)</code><br><code>WSTOPSIG(status)</code>   | vero se il figlio è fermato<br>ritorna il segnale                      |

Queste macro hanno per argomento un intero, non un puntatore!

# La chiamata exit

- `void exit(int status);`  
termina il processo chiamante
- rende disponibile il valore di `status` al processo padre (che lo otterrà tramite `wait`)
  - Nel caso di conclusione normale lo stato di uscita del processo viene caratterizzato tramite il valore `exit status` (stato di uscita), cioè il valore passato alle funzioni `exit` o `_exit` (o dal valore di ritorno del `main`)
  - Se il processo viene concluso in maniera anomala il programma non può specificare nessun `exit status`, ed è il kernel che deve generare autonomamente il `termination status` per indicare la ragione della conclusione anomala
  - Si noti la distinzione fra `exit status` e `termination status`: quello che contraddistingue lo stato di chiusura del processo e viene riportato attraverso le funzioni `wait` o `waitpid` è sempre quest'ultimo; in caso di conclusione normale il kernel usa il primo (nel codice eseguito da `_exit`) per produrre il secondo

[ 69 ]

# Esempio

```
#include "apue.h"
#include <sys/wait.h>
void pr_exit (int status)
{
 if (WIFEXITED(status))
 printf("term. normale, exit status =%d\n", WEXITSTATUS(status));
 else if (WIFSIGNALED(status))
 printf("term. anomala", num. segnale =%d\n", WTERMSIG(status));
 else if (WIFSTOPPED(status))
 printf("figlio fermato, num. segnale %d\n", WSTOPSIG(status));
}
```

[ 70 ]

# Processi zombie

- Un processo che termina non scompare dal sistema fino a che il padre non accetta il suo codice di terminazione
  - Un processo che sta aspettando che il padre accetti il suo codice di terminazione è chiamato processo **zombie** (`<defunct>` in `ps`)
- Se il padre non termina e non esegue mai una `wait()`, il codice di terminazione non sarà mai accettato ed il processo resterà sempre uno zombie
- Uno zombie non ha aree codice, dati o pila allocate, quindi non usa molte risorse di sistema ma continua ad avere un PCB nella Process Table (di grandezza fissa)

[ 71 ]

# Processi adottati

- Se un processo padre termina prima di un figlio, quest'ultimo processo viene detto **orfano**
- Il kernel assicura che tutti i processi orfani siano adottati da `init()` ed assegna loro **PPID 1**
- Cosa accade quando un processo adottato da `init` finisce?
  - Il processo adottato non diventa zombie, poiché `init` è scritto in modo tale che se un qualsiasi suo processo figlio termina, venga chiamata una delle funzioni `wait` per determinare lo stato di uscita
    - `init` previene la proliferazione di zombie
  - Osserviamo che per processi di `init`, intendiamo sia i processi generati direttamente da `init` che quelli rimasti orfani e da esso adottati successivamente

[ 72 ]

# Race Conditions

- Si verificano quando più processi cercano di operare con dati condivisi
  - Il risultato finale dipende dall'ordine in cui i processi sono eseguiti
- In generale, non è possibile predire quale processo venga eseguito per primo
  - Anche se lo sapessimo, ciò che accade dopo che il processo inizia l'esecuzione dipende dal carico del sistema e dall'algoritmo di scheduling del kernel
- Per evitare le **race condition** è necessaria qualche forma di segnalazione tra i vari processi coinvolti o varie forme di IPC

[ 73 ]

# Famiglia exec

[ 74 ]

## La famiglia di funzioni exec

- Avere due processi che eseguono esattamente lo stesso codice non è molto utile, allora nella pratica accade che:
  - si genera un secondo processo per affidargli l'esecuzione di un compito specifico (ad esempio, la gestione di una connessione dopo che questa è stata stabilita)
  - gli si fa eseguire un altro programma (come fa la **shell**)
- Per quest'ultimo caso si usa la terza funzione fondamentale per la programmazione con i processi che è la **exec**

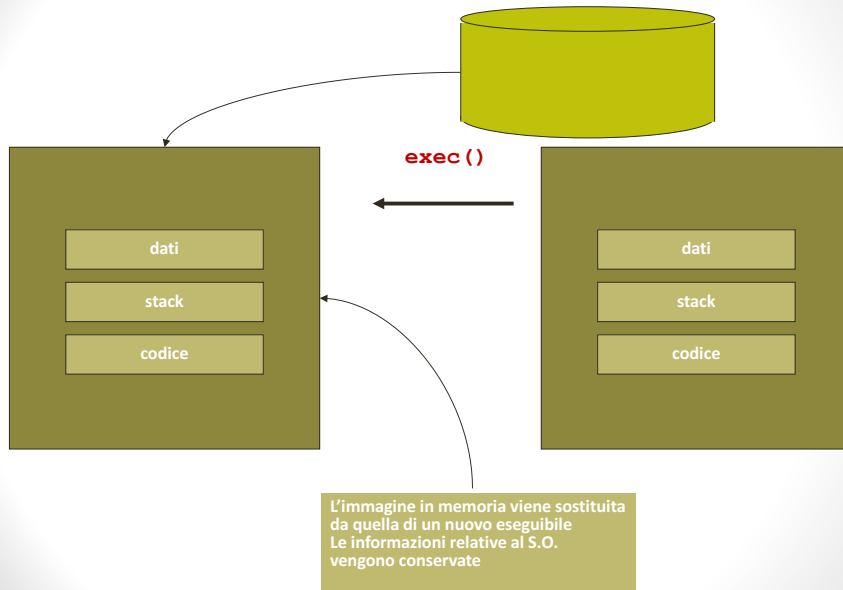
[ 75 ]

## Le funzioni exec

- Il programma che un processo sta eseguendo si chiama immagine del processo
  - le funzioni della famiglia **exec** permettono di caricare un altro programma da disco sostituendo quest'ultimo all'immagine corrente; l'immagine precedente viene completamente cancellata
- Quando il nuovo programma termina anche il processo termina e non si può tornare alla precedente immagine

[ 76 ]

## exec in un processo figlio



## La famiglia exec

- `exec` è una famiglia di primitive:

```
int execl(const char *path, const char *arg0,
.../* (char *)0 */);
```

```
int execle(const char *path, const char *arg0, ...
/* (char *)0, char *const envp[] */);
```

```
int execlp(const char *file, const char *arg0,
.../* (char *)0 */);
```

...l (list) ...e (environment)

...p (path) fa riferimento alla variabile di shell \$PATH

## La famiglia exec (2)

...v (vector) :

```
int execv(const char *path, char *const argv[]);
```

```
int execve(const char *path, char *const argv[],
char *const envp[]);
```

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

...e (environment)

...p (path) fa riferimento alla variabile di shell

- Tutte le sei funzioni della famiglia `exec` restituiscono -1 in caso di errore, altrimenti, in caso di successo, non ritornano

## Le funzioni exec

- Quando un processo chiama una funzione della famiglia `exec` esso viene completamente sostituito dal nuovo programma
  - il `pid` del processo non cambia, dato che non viene creato un nuovo processo
  - la funzione rimpiazza semplicemente lo `stack`, lo `heap`, i `dati` e il `testo` del processo corrente con un nuovo programma letto da disco
- Come abbiamo già visto, ci sono 6 versioni delle `exec` che possono essere usate per questo compito, in realtà sono tutte un front-end a `execve`

```
int execve(char *filename, char *argv[], char *envp[])
```

## Le funzioni exec (2)

- La funzione `execve` esegue il file o lo script indicato da `filename`, passandogli la lista di argomenti indicata da `argv` e come ambiente la lista di stringhe indicata da `envp`
- Entrambe le liste devono essere terminate da un puntatore nullo

( 81 )

## Le funzioni exec (3)

- Le differenze delle funzioni della famiglia exec sono riassunte dai suffissi `v` ed `l` che stanno per `vector` e `list`. Nel primo caso gli argomenti sono passati tramite il vettore di puntatori `argv[]` a stringhe terminate con zero. Questo vettore deve essere terminato con un puntatore nullo
- Nel secondo caso le stringhe degli argomenti sono passate alla funzione come lista di puntatori, nella forma: `char *arg0, char *arg1, ..., char *argn, NULL` che deve essere terminata da un puntatore NULL

( 82 )

## Le funzioni exec (4)

- La seconda differenza fra le funzioni riguarda la modalità con cui si specifica il programma che si vuole eseguire
  - Con il suffisso `p` si indicano le due funzioni che replicano il comportamento della shell nello specificare il comando da eseguire
  - Quando l'argomento file non contiene `"/` esso viene considerato come un nome di programma e viene eseguita automaticamente una ricerca fra i file presenti nella lista di directory specificate dalla variabile di ambiente `PATH`
  - Le altre quattro funzioni si limitano invece a cercare di eseguire il file indicato dall'argomento `path`, che viene interpretato come il pathname del programma

( 83 )

## Le funzioni exec (5)

- La terza differenza è come viene passata la lista delle variabili di ambiente
  - Con il suffisso `e` vengono indicate quelle funzioni che necessitano di un vettore di parametri `envp[]` analogo a quello usato per gli argomenti a riga di comando
  - Le altre usano il valore della variabile `environ` del processo di partenza per costruire l'ambiente

( 84 )



## Esempio: myexec.c

- Il programma **myexec.c** mostra un messaggio e quindi rimpiazza il suo codice con quello dell'eseguibile **ls** (con opzione **-l**). Si noti che non viene invocata nessuna **fork()**
- La **execl()**, eseguita con successo, non restituisce il controllo all'invocante (quindi la seconda **printf()** non è mai eseguita)

```
#include <stdio.h>
int main (void) {
 printf("Sono il processo %d, eseguo una ls -l\n",getpid());
 execl("/bin/ls", "ls", "-l", NULL); /* Esegue ls -l */
 printf("Questa riga non dovrebbe essere eseguita\n");
}
```

[ 89 ]

## Myexec in esecuzione

```
$ myexec
Sono il processo 797 e sto per eseguire una ls -l
total 3324
drwxr-xr-x 3 lagrene bireli 4096 May 16 19:14 Glass
-rwxr-xr-x 1 lagrene bireli 22199 Mar 17 18:34 lez1.sxi
-rwxr-xr-x 1 lagrene bireli 25999 May 21 17:14 lez20.sxi
$
```

[ 90 ]

## Esempio: myexec1.c

```
int main()
{
 int pid, status;
 pid=fork();
 if (pid==0)
 {execl("/bin/ls", "ls", "-l", "pippo", (char *)0);
 printf("exec fallita!\n");
 exit(1);
 }
 else if (pid >0)
 {pid=wait(&status);
 /* gestione dello stato.. */
 exit(0);
 }
 else printf("fork fallita!");
 exit(2);
}
```

[ 91 ]

## Esempio: esecuzione di un comando

- Programma **execute cmd args**
  - Usa **fork()** ed **execvp()** per eseguire il comando **cmd** con i suoi argomenti
- La lista degli argomenti è passata ad **execvp()** tramite il secondo argomento **&argv[1]**. Si ricordi che **execvp()** permette di usare la variabile **PATH** per trovare l'eseguibile

```
#include <stdio.h>
int main (int argc, char *argv[]) {
 if (fork () == 0) { /* Figlio */
 execvp (argv[1], &argv[1]); /* Esegue un altro programma */
 fprintf (stderr, "Non ho potuto eseguire %s\n", argv[1]);
 }
}

La lista degli argomenti è chiusa, come necessario, da NULL
poiché vale sempre argv[argc]=NULL
```

[ 92 ]

# Esecuzione di un comando

```
$ execute sleep 5
$ ps
PID TTY TIME CMD
822 pts/0 00:00:00 bash
925 pts/0 00:00:01 vi
965 pts/0 00:00:00 sleep
969 pts/0 00:00:00 ps
$
```

# Esempio: uso delle exec per visualizzare argomenti e variabili di ambiente

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

char *env_init[]={ "USER=sconosciuto", "PATH=/tmp", NULL};
int main(void){
 pid_t pid;
 if ((pid = fork())<0){
 perror("Errore fork");
 exit(-1);
 } else if (pid == 0) {
 if (execle("/home/lagrene/bin/echoall", "echoall",
"mioarg1", "MIO ARG2", (char *)0, env_init) <0){
 perror("Errore execle");
 exit(-1);}
 }
}
```

# Esempio (cont.)

```
if (waitpid(pid,NULL,0) <0){
 perror("Errore wait");
 exit(-1);}
if ((pid = fork()) <0) {
 perror("Errore fork");
 exit(-1);
} else if (pid == 0) {
 if (execlp("echoall","echoall","solo 1 arg", (char *)0)<0){
 perror("errore execlp");
 exit(-1);
 }
}
exit(0);
}
```

# Esempio (cont.)

```
/* echoall.c */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[]){
 int i;
 char **ptr;
 extern char **environ;

 for (i=0; i < argc; i++) // echo di tutti gli arg da riga di cmd
 printf("argv[%d]: %s\n", i, argv[i]);

 for (ptr = environ; *ptr !=0; ptr++) /* echo stringhe di env */
 printf("%s\n", *ptr);
 exit(0);
}
```



## Esempio in esecuzione

```
$./a.out
argv[0]: echoall
argv[1]: mioarg1
argv[2]: MIO ARG2
USER=sconosciuto
PATH=/temp
$ argv[0]: echoall /* Prompt prima di argv[0]
 perché il padre non attende la terminazione del
 figlio */
argv[1]: solo 1 arg
USER=lagrene
LOGNAME=lagrene
SHELL=/bin/bash
...
Etc. etc.
```

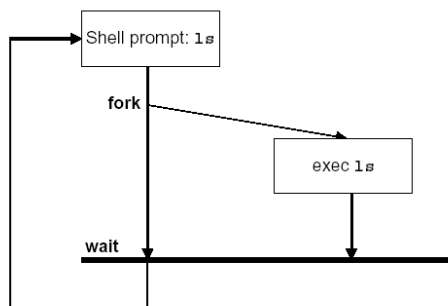
[ 97 ]

## Esecuzione dei comandi nella shell

- L'esecuzione dei comandi corrisponde ad una sequenza di stringhe separate da spazi. La prima di queste stringhe corrisponde al comando da eseguire mentre le restanti identificano i parametri passati al comando stesso
  - Esempio: `ls -la /usr`
- Ogni comando termina fornendo un **exit status** che rappresenta l'esito della computazione del comando stesso. Mediante l'exit status è possibile controllare la buona riuscita del comando
  - L'exit status è un intero:
    - 0 : esecuzione riuscita con successo
    - $n > 0$  : esecuzione fallita
    - $128 + n$  : esecuzione terminata in seguito al segnale numero  $n$
- L'esecuzione di un comando sospende temporaneamente la shell fino alla terminazione del comando stesso

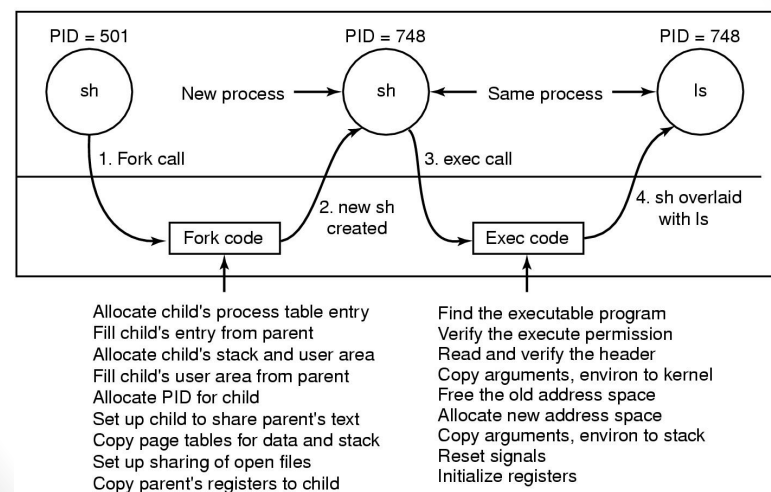
[ 98 ]

## Esecuzione dei comandi nella shell (cont.)



[ 99 ]

## fork+exec per una shell

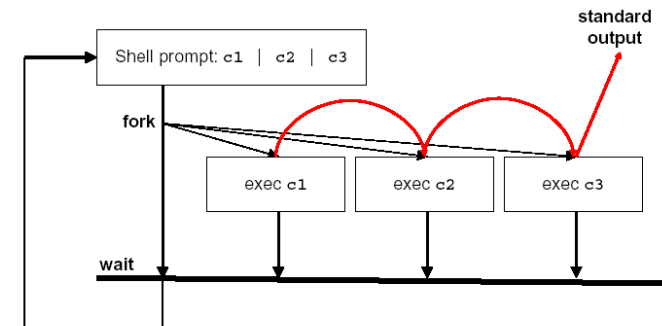


[ 100 ]

# Esecuzione di una pipeline da shell

- L'operatore `pipe` | consente alla shell di connettere tra loro due o più comandi
  - `comando1 | comando2 | ... | comando n`
- In questo modo, si redirige lo standard output del primo comando verso lo standard input del successivo e così via

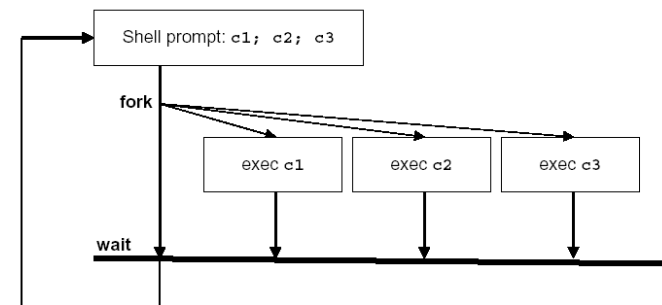
# Esecuzione di una pipeline da shell (cont.)



# Liste di comandi

- Le liste di comandi consentono di eseguire in maniera sequenziali più comandi come se costituissero un unico comando
- La sintassi per specificare liste di comandi è:  
`command1 [; command2...] [;]`
  - `command1` e `command2` possono essere delle pipeline
- L'`exit status` corrisponde all'`exit status` dell'ultimo comando della lista
- La shell attende che tutti i comandi nella lista abbiano terminato la loro esecuzione prima di restituire il prompt
- A differenza delle pipeline non c'è nessun collegamento tra l'input e l'output dei vari comandi nella lista

# Liste di comandi (cont.)



## Esempio: modello semplificato di shell

```
while (TRUE){
 read_command(command, parameters);
 if (fork()!=0) {
 waitpid(-1, &status, 0);
 } else {
 execve(command, parameters, env);
 }
}
```

Esempio: `cp file1 file2`

- Il meccanismo è identico al passaggio di parametri da linea di comando in C

[ 105 ]

## Esercizio

```
int glob=20;
int pid=0;
int main() {
 int i=0;
 for (i=2;i<4;i++) {
 pid=fork();
 if (pid==0) {
 glob=glob*2;
 sleep(i+1);
 }
 glob=glob-1;
 printf("Valore di glob=%d\n",glob);
 }
}
```

[ 106 ]

## Esempio: creazione e attesa processi

```
...
int main (int argc, char *argv[]) {
 pid_t childpid;
 int i, n;
 pid_t waitreturn;
 if (argc != 2){ /* controllo argomenti */
 fprintf(stderr, "Uso: %s processi\n", argv[0]);
 return 1;
 }
 n = atoi(argv[1]);
 for (i = 1; i < n; i++)
 if (childpid = fork())
 break;
 while(childpid != (waitreturn = wait(NULL)))
 if ((waitreturn == -1)
 break;
 fprintf(stderr, "processo %d, padre %d\n",getpid(),getppid());
 return 0;
}
```

[ 107 ]

## Esercizio

- Scrivere un programma che crea uno zombie e poi esegue il comando **ps** per verificare che il processo è uno zombie

[ 108 ]