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)

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

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

Le chiamate di sistema wait e waitpid (cont.)

- Per entrambe le funzioni l'argomento statloc é 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>

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
WIFEXITED(*status*)	vero se il figlio è terminato normalmente
WEXITSTATUS(*status*)	ritorna lo stato
WIFSIGNALED(*status*)	vero se il figlio è uscito a causa di un segnale
WTERMSIG(*status*)	ritorna il segnale
WIFSTOPPED(*status*)	vero se il figlio è fermato
WSTOPSIG(*status*)	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

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));
}
```

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)

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

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

Famiglia exec

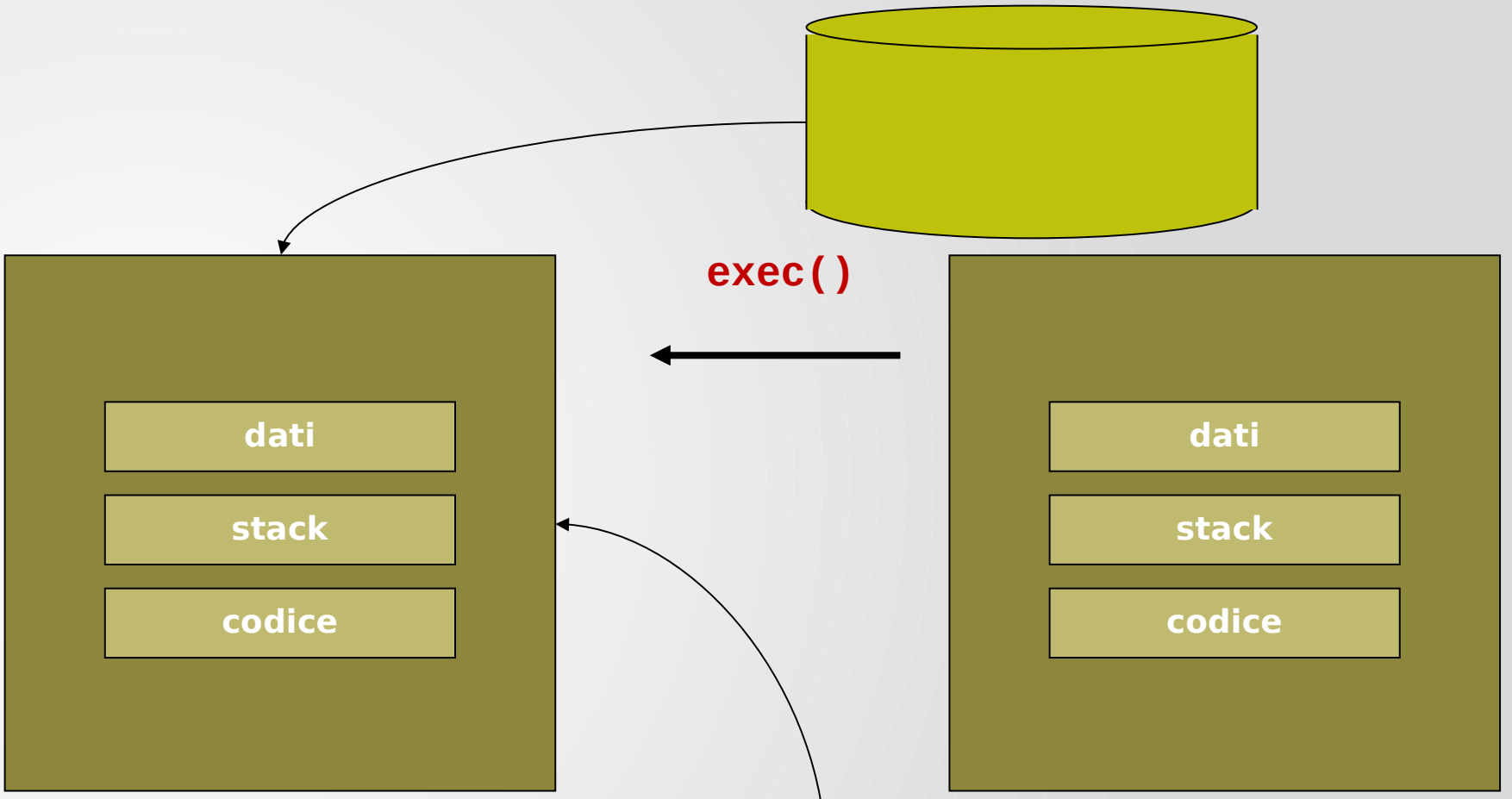
La famiglia di funzioni exec

- 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

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

exec in un processo figlio



L'immagine in memoria viene sostituita da quella di un nuovo eseguibile
Le informazioni relative al S.O. vengono conservate

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

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

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

Le funzioni exec (5)

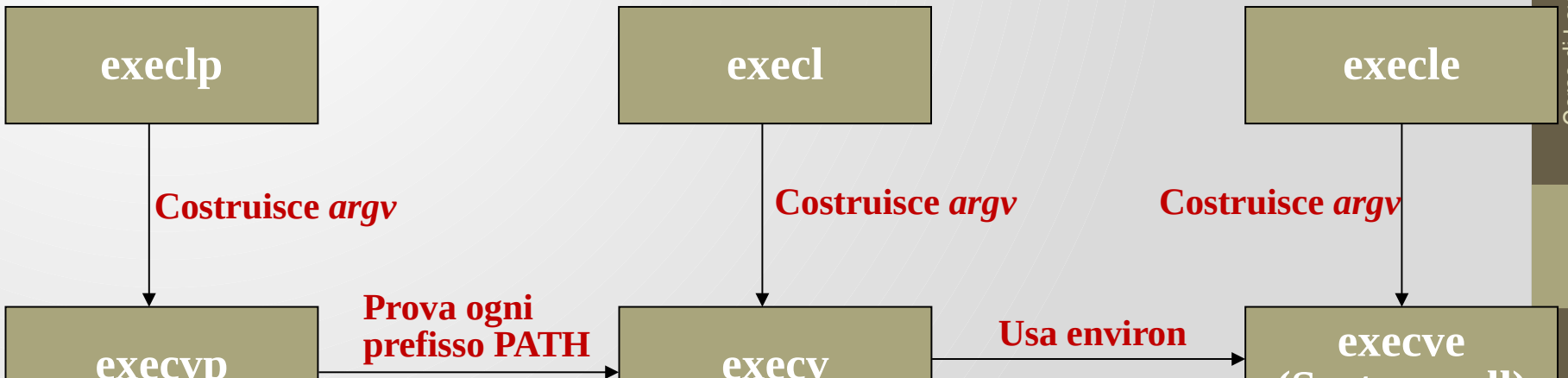
- La terza differenza è come viene passata la lista delle variabili di ambiente
  - Con il suffisso e vengono indicate quelle funzioni che necessitano di un vettore di parametri envp[] analogo a quello usato per gli argomenti a riga di comando
  - Le altre usano il valore della variabile environ del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

Caratteristiche funzioni exec

Caratteristiche	Funzioni					
	`execl`	`execlp`	`execle`	`execv`	`execvp`	`execve`
Argomenti a lista	●	●	●			
Argomenti a vettore				●	●	●
Filename completo	●		●	●		●
Ricerca su PATH		●			●	
Ambiente a vettore			●			●
Uso di environ	●	●		●	●	

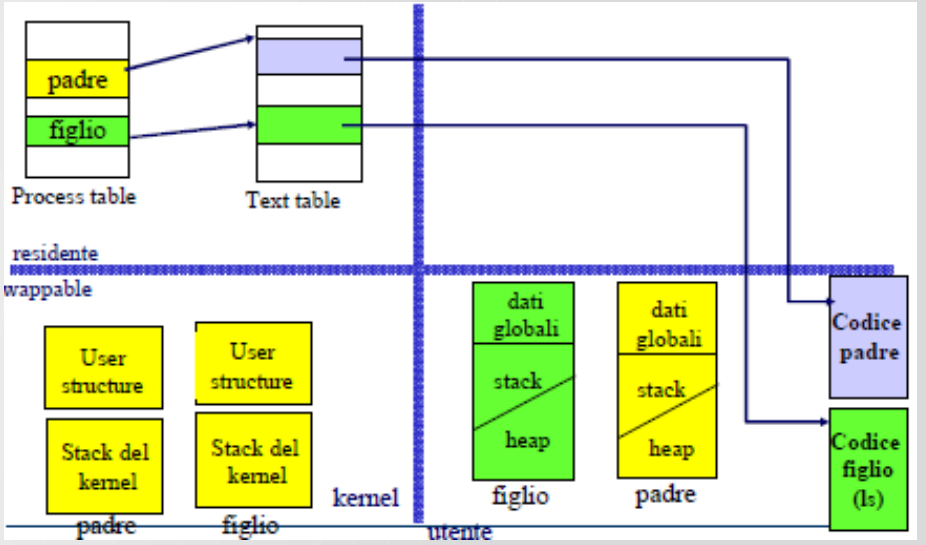
Relazione tra le funzioni della famiglia exec

- Relazione tra le funzioni exec
  - solo la funzione execve è una system call del kernel
  - le altre sono librerie che invocano la system call



Effetti della exec: visione d'insieme

- Il processo dopo l'exec:
  - mantiene la stessa **process structure** (salvo le informazioni relative al codice):
    - stesso pid
    - stesso ppid
    - ...
  - ha codice, dati globali, stack e heap nuovi
  - referenzia una nuova text structure
  - mantiene **user area** (**a parte PC e informazioni legate al codice**) e stack del kernel:
    - mantiene le stesse risorse (es: file aperti)
    - mantiene lo stesso environment (a meno che non sia execle o execve)



Schema di chiamate a fork, exec e wait (waitpid)

```
esito = fork(); /* crea un processo figlio */
if (esito < 0) /* creazione OK? */
{ error("fork() non eseguita");
...
}
if (esito == 0) /* è il processo figlio ? */
{exec("p",...); /* si esegue il programma “p” */
...    /* ...a meno di errori */
}
id = wait(&stato); /* il processo padre attende la  fine
  */
if (stato == ...) /* del figlio e ne analizza  l’esito */
{
...
}
```

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");
}
```

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
$
```

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);
}
```

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

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/giusal/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);
}
```

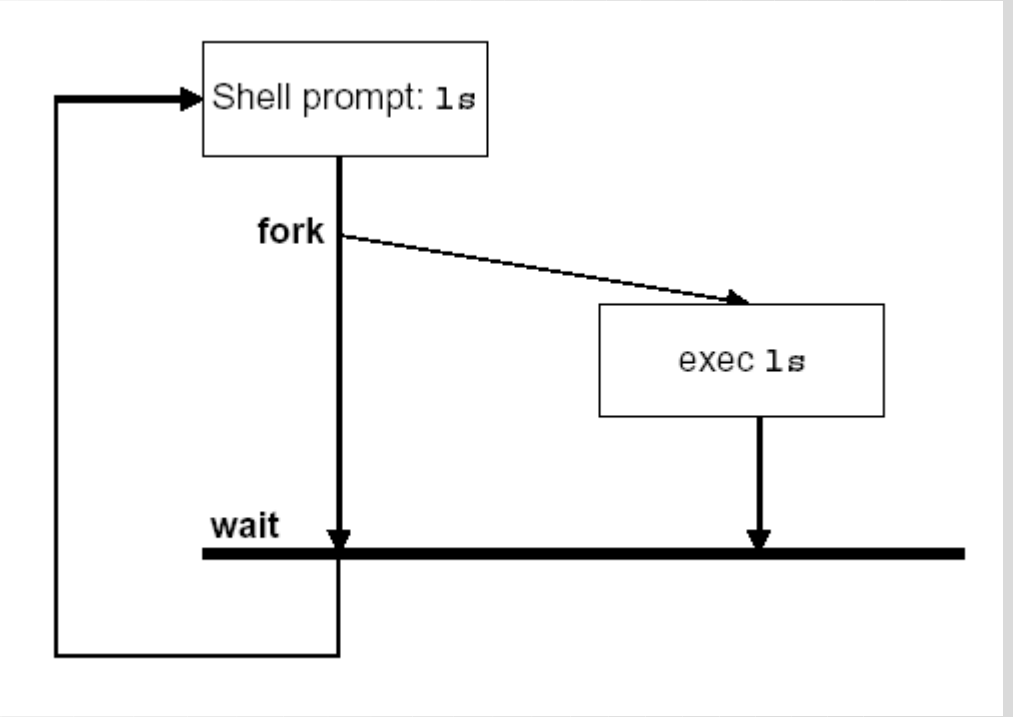
Esempio in esecuzione

```
$ ./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=giusal
LOGNAME=giusal
SHELL=/bin/bash
….
Etc. etc.
```

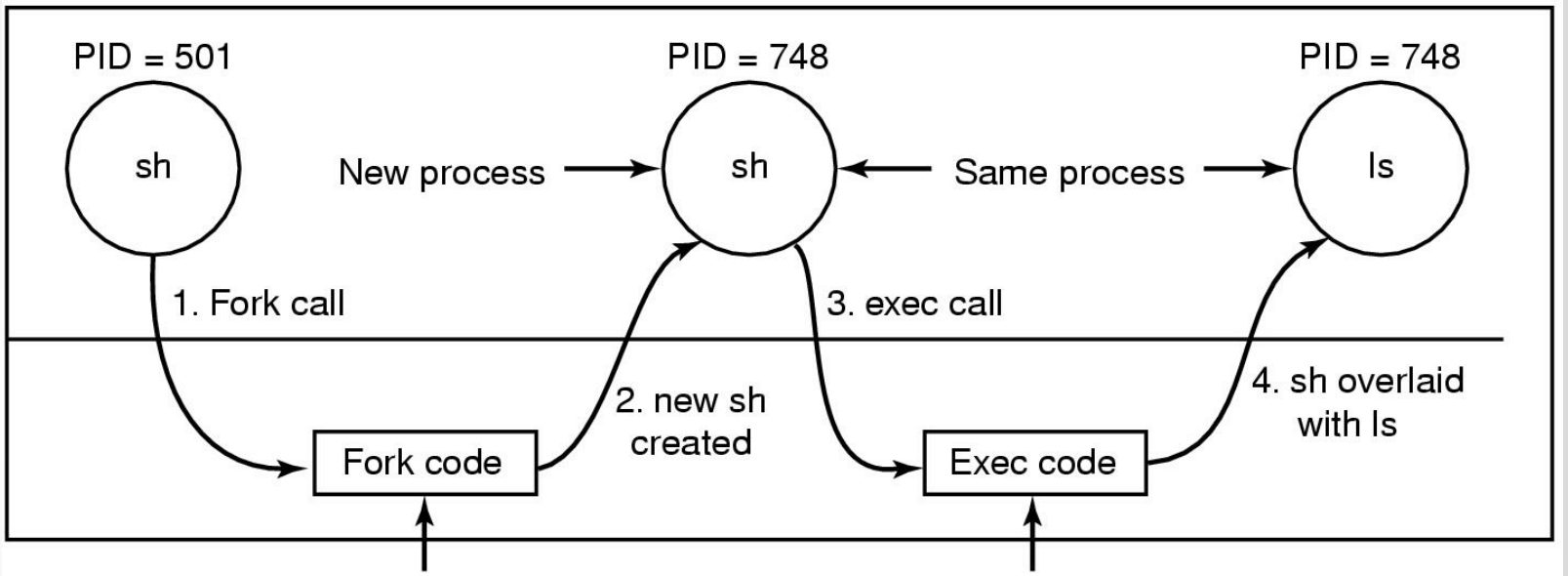
Esecuzione dei comandi nella shell

- 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

Esecuzione dei comandi nella shell (cont.)



fork+exec per una shell



Allocate child's process table entry
Fill child's entry from parent
Allocate child's stack and user area
Fill child's user area from parent
Allocate PID for child
Set up child to share parent's text
Copy page tables for data and stack
Set up sharing of open files
Copy parent's registers to child

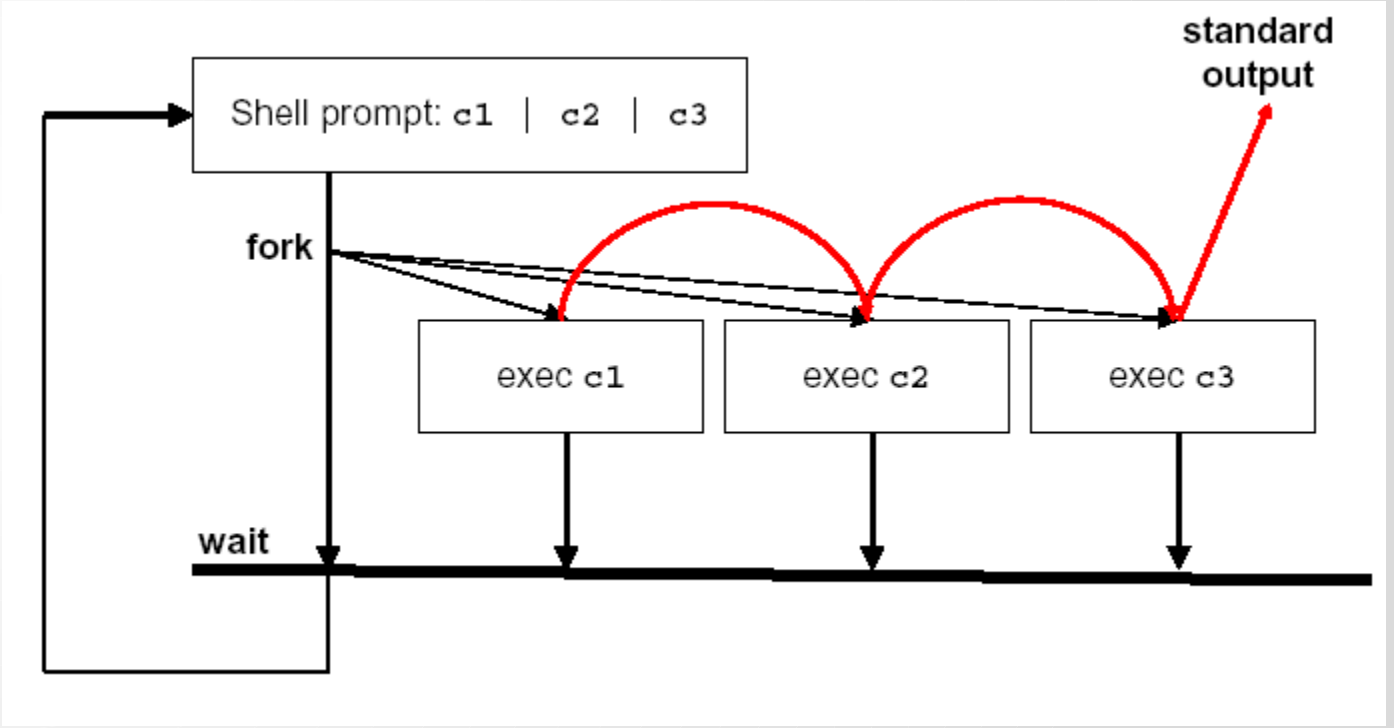
Find the executable program
Verify the execute permission
Read and verify the header
Copy arguments, environ to kernel
Free the old address space
Allocate new address space
Copy arguments, environ to stack
Reset signals
Initialize registers

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.)



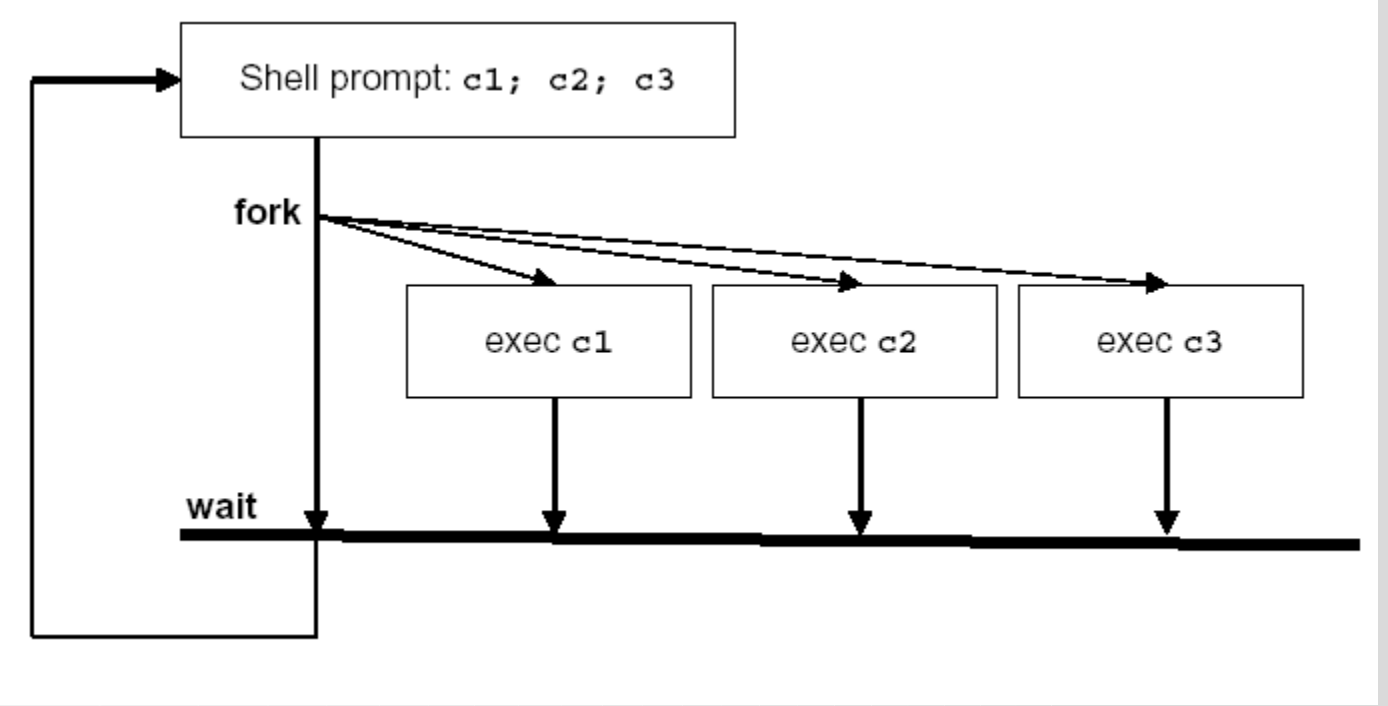
Liste di comandi

- 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

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);
 }
}
```

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;
}
```