4.0 PROCESSI IN UNIX

Ogni processo ha un **identificatore unico** non negativo. Il primo processo è quello init che ha ID uguale a 1 ed è localizzato in */sbin/init*. Quest'ultimo è il padre di tutti i processi. Più in dettaglio viene invocato dal kernel alla fine del boot, legge il file di configurazione */etc/inittab* dove 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). Questo processo non muore mai. Ci sono delle system call su quelle che sono gli identificatori che hanno a che fare con i processi:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);  /* PROCESS ID DEL PROCESSO CHIAMANTE*/
pid_t getppid(void); /*PROCESSO 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*/
uid_t getgid(void); /*REAL GROUP ID DEL PROCESSO CHIAMANTE*/
uid_t getegid(void); /*EFFECTIVE GROUP ID DEL PROCESSO CHIAMANTE*/
```

La system call **getpid** ci restituisce il process ID del processo chiamante.

La system call **getppid** ci restituisce il process ID del padre del processo chiamante. Questa relazione padre-figlio è l'essenza fondamentale della struttura dei processi all'interno di un sistema. Ogni processo può creare un processo figlio creando un ramo.

Banalmente questo programma stampa il pid del processo corrente→

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


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

Per osservare la ramificazione dei processi in un sistema, c'è il comando **ps** che permette di vedere i processi attivi all'interno di un sistema con tutta una serie di opzioni. Senza alcuna opzione ci dice quali sono i processi che sono in stato di running nella shell attuale. Possiamo vedere che ci viene fornito il PID di ogni processo:

```
[geronimo@fujitsu ~]$ ps
  PID TTY          TIME CMD
11561 pts/0    00:00:00 bash
13487 pts/0    00:00:00 ps
```

Se vogliamo sapere qualcosa in più possiamo eseguire il seguente comando: **ps l:**

```
[geronimo@fujitsu ~]$ ps l
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
4  1000  1487  1465  20   0 157996  5660 -      Ssl+ tty2       0:00 /usr/lib/gdm-x-session --run-s
4  1000  1489  1487  20   0 1415124 175096 -     Sl+  tty2      11:22 /usr/lib/Xorg vt2 -displayfd 3
0  1000  1503  1487  20   0 257832 14028 -      Sl+  tty2       0:00 /usr/lib/gnome-session-binary
0  1000 11561 11554  20   0   8448  5160 -      Ss   pts/0      0:00 bash
4  1000 13496 11561  20   0   9652  3236 -      R+   pts/0      0:00 ps l
```

Oltre PID di ogni processo c'è pure il PPID che è colui che ha generato un certo processo. Ad esempio, possiamo notare il PPID del processo **ps l** che corrisponde a 11561, suo padre, che è la **bash**, ha proprio il suo PID uguale a 11561. Con **ps lx** lista tutti i processi associati ad un real user ID.

4.1 FORK

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

La seguente funzione non prende argomenti. Restituisce un tipo *pid_t* che possiamo semplificare ad un normale valore intero. Restituisce:

- 0 nel figlio.
- **pid** del figlio nel padre.
- -1 in caso di errore.

L'effetto di una fork è quello di creare un nuovo processo (entità attiva) che avrà un suo pid e inizia la sua esecuzione nel sistema. Un nuovo processo creato all'interno di un programma utilizzerà un certo spazio di memoria, il processo figlio diventa una copia (eredita le stesse istruzioni, dati, etc.) del processo padre, non eredita il pid in quanto ogni processo ha il proprio pid. Visto che è una copia del padre, entrambi i processi avranno lo stesso program counter che sarà l'istruzione successiva a quella della fork. Il valore di ritorno della fork è duplice nel caso in cui la funzione abbia esito positivo: il valore di ritorno della fork nel processo padre sarà il pid del figlio, mentre il valore di ritorno della fork nel processo figlio sarà 0. Entrambi i processi continueranno ad eseguire le istruzioni di un certo programma. La scelta di assegnare 0 al figlio e il pid del figlio nel padre ha un senso logico. Infatti, si potrà mantenere traccia di entrambi i processi durante l'esecuzione di un programma. All'atto della creazione di un processo figlio non avviene immediatamente una copia di tutti i dati, ma alcuni vengono condivisi. Se uno dei due processi ha necessità di modificare questi dati avverrà la copia vera e propria. In generale non si sa se il figlio è eseguito prima del padre, questo dipende dall'algoritmo di scheduling usato dal kernel.

Analizziamo la seguente porzione di codice →

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

int glob=10;      /* dati inizializzati globalmente */

int main(void)
{
   int var=100;   /* vbl locale */
   pid_t pippo;
    printf("prima della fork\n");
    pippo=fork();
     if( (pippo == 0) {glob++; var++;}
       else sleep(2);
   printf("pid=%d, glob=%d, var=%d\n",getpid(),glob,var);
   exit(0);
}
```

Fuori dal main c'è la dichiarazione di una variabile globale *glob* inizializzata a 10. All'interno viene inizializzata una variabile locale *var* uguale a 100. Viene istanziato *pippo* che è un pid. C'è una printf in cui viene stampata una sequenza di caratteri e si presume avvenga una sola volta. Alla variabile *pippo* viene eseguita la fork. Entrambi, padre e figlio, continueranno dall'istruzione successiva. Il valore di *pippo* varrà 0 per il figlio mentre per il padre sarà il pid del figlio. Tenendo a mente questa informazione andiamo dentro l'if, che viene affrontato da entrambi i processi. Per uno dei due il risultato dell'if sarà vero mentre per l'altro sarà falso. In particolare, il figlio avrà valore vero nella condizione e dunque siccome sta andando a modificare i valori delle variabili *glob* e *var* ci sarà una copia dei due valori che verranno anche incrementati. Il padre invece fallirà il controllo, dunque il processo padre rimarrà fermo per 2 secondi. In questo modo possiamo essere abbastanza sicuri che il figlio verrà eseguito prima del padre. Terminato l'if-else, il figlio stamperà immediatamente tramite printf i suoi valori pid, glob e var. Dopo circa 2 secondi ci sarà la printf fatta dal processo padre che stamperà il suo valore pid, glob e var.

Poiché il processo figlio ha effettuato una modifica alle variabili *glob* e *var*, ha effettuato una copia delle variabili del processo padre e le ha modificate per lui (anche le variabili globali), dunque l'esecuzione di questo processo ci fornirà il seguente output:

```
$ a.out
prima della fork
pid=227, glob=11, var=101
pid=226, glob=10, var=100
```

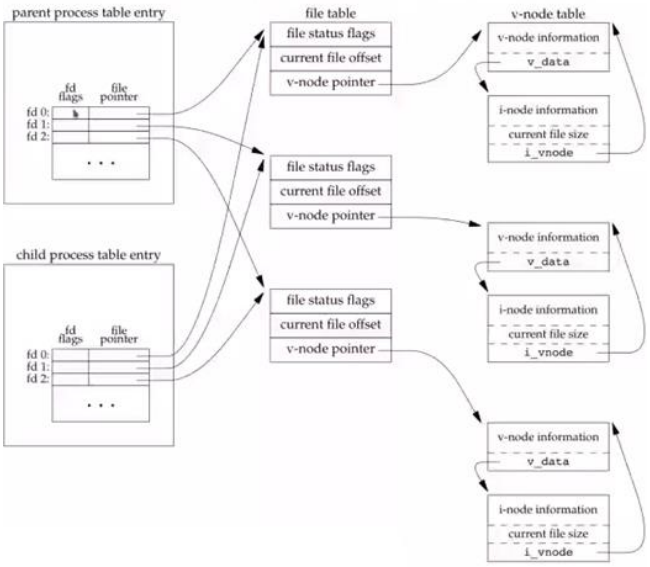
Vediamo ora questa porzione di codice che **si differenzia dalla precedente nella printf in quanto qui non c'è lo \n** →

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

int glob=10;       /* dati inizializzati globalmente */

int main(void)
{
   int var=100;   /* vbl locale */
   pid_t pippo;
    printf("prima della fork");
    pippo=fork();
     if( (pippo == 0) {glob++; var++;}
       else sleep(2);
   printf("pid=%d, glob=%d, var=%d\n",getpid(),glob,var);
  exit(0);
}
```

Notiamo che viene eseguita la *printf("prima della fork")*, in questo caso sappiamo che non viene stampato nulla su standard output (perché non c'è il \n che svuota il buffer) e che l'informazione da stampare viene salvata nel buffer.
Successivamente viene eseguita la fork e sappiamo già come viene salvata in pippo. Il figlio incrementerà le variabili, mentre il padre aspetterà 2 secondi. Adesso il figlio effettua la printf (si consideri che c'è un \n che svuota il buffer), però si consideri che al momento della fork è stata effettuata una copia anche del buffer e dunque anche il processo figlio stamperà "prima della fork". L'output di questo programma sarà dunque:

```
$ a.out
prima della forkpid=227, glob=11, var=101
prima della forkpid=226, glob=10, var=100
```

Dobbiamo tenere conto di come sono gestiti i file aperti all'interno del processo. Sappiamo che esiste la tabella dei file aperti da un processo e ad ogni file è associato un puntatore al cosiddetto file table in cui ci sono tutte le informazioni del file. Attraverso la file table, al suo interno c'è un campo *v-node pointer* che punta al file fisico.
2 processi distinti hanno puntatori uguali (tranne 0 1 2 che sono gli standard che sono sempre gli stessi per tutti i processi), dunque punteranno alla stessa file table (condividono anche l'offset dunque). Se il padre prima della creazione del figlio aveva aperto un file e aveva eseguito una certa operazione, il figlio, quando verrà creato erediterà questa informazione.
Da questo capiamo che nella porzione di codice precedenti, lo standard output del figlio è copiato da quello del padre. Tutti i file aperti del padre vengono condivisi al figlio.
Quest'ultima affermazione può creare problemi di sincronizzazione: chi scrive per prima? Se non si sincronizzasse potremmo avere un output intermixed tra i due processi. Nel codice precedente abbiamo messo una pezza con una *sleep(2)*.
C'è una regola: il padre aspetta che il figlio termini o viceversa. In generale è più naturale il primo caso, infatti, se un processo decide di creare un figlio, si aspetta che il figlio faccia qualcosa e che dunque termini. Altrimenti il processo padre che motivo aveva per crearlo?



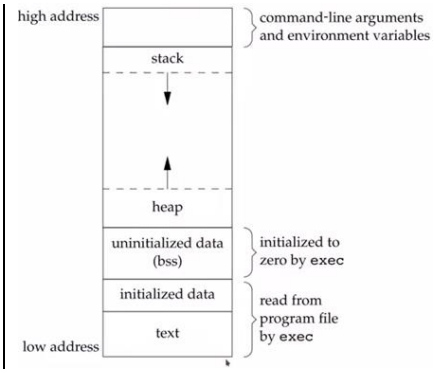

```
int main(void)
{
pid_t pid1, pid2;
pid1 = fork();
pid2 = fork();
exit(0);
}
```

In questa porzione di codice, vengono dichiarate due variabili di tipo *pid_t* che sono *pid1* e *pid2*. Viene eseguita una prima fork e il risultato viene immagazzinato all'interno di *pid1*. In questo momento abbiamo quindi 2 processi.
Questi 2 processi continuano l'esecuzione del programma, dall'istruzione successiva che è un'altra fork che sarà eseguita da entrambi. Dunque, il numero di processi che vengono generati in totale sono 4.

La fork in generale si usa quando un processo attende richieste (per esempio da parte di client), dunque decide di duplicarsi: il figlio gestisce la richiesta e il padre si mette in attesa di nuove richieste. Quando viene avviato un processo:

- Si esegue prima una routine di start-up speciale che prende i valori passati dal kernel alla linea di comando (in argv[] se il processo si riferisce ad un programma C) e le variabili d'ambiente.
- Successivamente viene chiamata la funzione principale da eseguire (il main)

Questa è una tipica occupazione di memoria di un processo. In alto c'è la zona per le variabili d'ambiente e variabili da linea di comando. Poi c'è un area condivisa da heap stack. Lo stack va verso il basso e l'heap verso l'alto in modo da avere lo stesso spazio di memoria a disposizione. Ci sono poi i dati non inizializzati, quelli inizializzati e le istruzioni di un programma.
Le variabili d'ambiente sono variabili fondamentali per il nostro processo (SHELL, PATH, USER, etc.)



4.2 EXIT

La terminazione di un processo può essere: **normale** o **anormale**. Con terminazione normale s'intende un **return** dal main, una chiamata a **exit** o **_exit**. Con terminazione anormale s'intende una chiamata **abort** oppure l'arrivo di un segnale (per esempio mandare un CTRL+C).

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

La **exit** restituisce *status* al processo che chiama il programma includente exit. Effettua prima una pulizia e poi ritorna al kernel. Più in particolare viene effettuato uno shoutdown delle funzioni di libreria standard di I/O (fclose di tutti gli stream lasciati aperti) e tutto l'output è flushed.
La **_exit**, invece, si ritorna direttamente al kernel, dunque tutte le printf correttamente eseguite, ma senza \n, non verranno stampate.

Con questa porzione di codice, dunque, non vedremo mai alcuna stampa su standard output →

```
#include <stdio.h>


int main(void)

{

    printf("Ciao a tutti");


   _exit(0);
    exit(0);
}
```

4.3 EXIT HANDLER

```
#include <stdlib.h>
int atexit (void (*funzione) (void));
```

Questa system call prende come parametro una funzione di tipo void. Restituisce 0 se OK, diverso da 0 se errore.
Questa system call fa in modo che la funzione che gli è stata passata come parametro, venga eseguita solo all'atto di una exit. Quindi significa che si mettono da parte delle funzioni che vengono eseguite esattamente quando viene chiamata una exit, solo in quel momento. Come se programmassi un'uscita fatta in un certo modo. Si possono dichiarare più exit handlers, che vengono messi in uno stack.

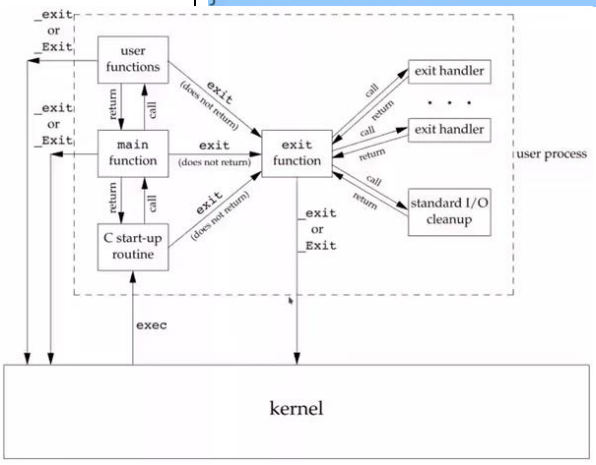
In questa porzione di codice che una prima atexit che prende come parametro la funzione *my_exit2*. Successivamente ci sta un'altra atexit che prende come parametro la funzione *my_exit1*. C'è una printf che stampa una sequenza di caratteri. Al momento in cui c'è il return viene prima eseguito *my_exit1* e poi *my_exit2*.

```
$ a.out
ho finito il main
sono il primo handler
sono il secondo handler
```

Se invece di return(0) avessimo messo _exit(0) l'output sarebbe stampato solo "ho finito il main\n" (si consideri che viene stampato perché ci sta \n che svuota il buffer).

```
int main(void)
{
  atexit(my_exit2);     atexit(my_exit1);
  printf("ho finito il main\n");
  return(0);
}

static void my_exit1(void)
{
  printf("sono il primo handler\n");
}

static void my_exit2(void)
{
  printf("sono il secondo handler\n");
}
```

Questa figura mostra realmente quello che accade quando si sta avvia, esegue e termina un processo. Si inizia il main, al suo interno possono esserci delle funzioni utente a cui si passa il controllo e successivamente si ritorna al main. Alla fine del main c'è poi una exit che chiama tutti gli exit handler che ci sono eventualmente, poi fa il cleanup di I/O. Alla fine verrà eseguita una _exit che passerà il controllo al kernel.



4.4 WAIT E WAITPID

Il kernel all'atto della terminazione del processo figlio invia al padre un segnale *SIGCHLD*. Il padre con questo segnale può: ignorarlo (default) o lanciare una funzione (signal handler). In più, il processo padre può ottenere una serie di informazioni sullo stato di uscita del figlio, utilizzando due funzioni:

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

La **wait** restituisce il pid del processo che ha terminato la sua esecuzione e gli si può passare come parametro un puntatore ad un intero. Ogni volta che viene invocata tale funzione il processo padre si blocca in attesa che un processo figlio termini la sua esecuzione, una volta terminato, la **wait** restituisce il PID se OK, -1 in caso di errore. All'interno della variabile *statloc* è memorizzato il cosiddetto **stato di terminazione del figlio**.
La **waitpid** oltre a contenere la *statloc*, contiene altri 2 parametri che sono un *pid* e *options*. Una volta che la **waitpid** viene chiamata dal processo padre che chiede lo stato di terminazione all'interno della variabile *statloc* del figlio specificato dal valore *pid*. Il processo padre si blocca in attesa o meno a seconda del contenuto di *options*. Il valore di ritorno è il PID del processo di cui si sta aspettando la terminazione, 0 oppure -1 in caso di errore.
Vediamo quali sono i valori attribuibili a ***pid***:

- *pid > 0*: si vuole aspettare la terminazione del figlio che abbiamo process ID uguale al valore di pid.
- *pid* == -1: qualsiasi figlio. La rende simile a **wait.**
- *pid == 0*: si vuole aspettare la terminazione di un figlio che abbia GroupID uguale al padre.
- *pid < 0*: si vuole aspettare la terminazione di un figlio che abbia GroupID uguale a **abs(pid)** (il valore assoluto di pid).

Il valori attribuibili a ***options*** sono:

- 0: non accade nulla, come nella **wait**.
- **WNOHANG**: il processo che ha chiamato la **waitpid**, non blocca se il figlio indicato non è disponibile.

La differenza tra le due funzioni è che quando si usa la **wait** l'idea è che il processo si blocca e aspetta che un figlio termini la sua esecuzione. Se qualche figlio ha terminato la propria esecuzione quando si lancia la **wait**, viene ritornato lo stato di un figlio. Nel caso in cui si lancia la **wait** e un processo non ha figli viene ritornato immediatamente un errore. Se un processo padre ha più figli, con la **waitpid** posso scegliere quale figlio aspettare (1° argomento). Attraverso il 3° parametro posso evitare che il processo padre che sta aspettando la terminazione del figlio, si blocchi.

TERMINAZIONE DI UN PROCESSO:

Quando un processo termina il kernel che sta gestendo l'esecuzione del processo determina lo stato di terminazione ***normale*** o ***anormale***. Se è ***normale***, lo stato di terminazione è l'argomento della **exit**, **return** oppure **_exit**. Se è ***anormale***, il kernel genera uno stato di terminazione che indica il motivo anormale. In entrambi i casi il padre del processo ottiene questo stato da **wait** o **waitpid** dalla variabile *statloc*.
Per leggere lo stato di terminazione si usano delle MACRO, perché lo stato di terminazione è l'unione di due informazioni:

(terminazione normale/anormale) + (exit status negli 8 bit meno significativi se la terminazione è normale ||).

La MACRO **WIFEXITED**(*status*) darà valore vero o falso se il figlio ha terminato come effetto della exit. Nel caso restituisca vero, posso usare la seconda MACRO **WEXITSTATUS**(*status*) che mi restituirà lo stato di exit (il valore usato nella exit).

Nel caso **WIFEXITED**(*status*) dia valore falso, posso tentare con la macro **WIFSIGNALED**(*status*) che darà valore vero se il processo figlio ha terminato la propria esecuzione per l'arrivo di un segnale, oppure falso altrimenti. Nel caso sia vero, posso usare **WTERMSING**(*status*) che restituirà il numero del segnale che ha causato l'interruzione.
Se il processo figlio è stato stoppato posso controllarlo con la macro **WIFSTOPPED**(*status*), nel caso sia vero posso vedere quale dei due segnali di stop ha stoppato il processo figlio attraverso la **WSTOPSIG**(*status*).
Le macro **WIFEXITED, WIFSIGNALED, WIFSTOPPED,** sono quelle che ci riconoscono se è successo una terminazione normale, anormale o stop. Le altre dicono quale segnale o valore di exit è stato mandato.
In una situazione normale, un processo padre crea un figlio, fa la **wait**, quando il processo figlio termina il processo padre continua la sua esecuzione.
In una situazione anormale, se un processo figlio termina la sua esecuzione e il padre non ha ancora invocato la **wait**, si dice che viene creato uno **zombie**: un processo che ha terminato la sua esecuzione però ci sono ancora delle informazioni nel sistema che aspettano di essere restituite al padre (come lo stato di terminazione). Questo significa che nel momento in cui il processo termina, viene associato a quel processo lo status di terminazione.
Se il padre termina senza invocare la **wait** e il figlio è ancora in stato di running, il processo diventa un **orfano.**

In questa porzione di codice, si effettua una fork (creazione di un figlio). Entrambi i processi proseguiranno dalla riga successiva. In particolare, solo per il figlio la condizione dell'if sarà vera ed effettuerà una **exit**(128).
A questo punto, questo secondo if in linea di principio sarà eseguita da entrambi i processi, ma di fatto il figlio è uscito precedentemente. Con il comando **wait(&status)** il processo padre si mette in attesa che il processo figlio termini la sua esecuzione (con la exit precedente). Una volta sbloccata la **wait**, nel caso in cui il valore di ritorno della **wait** è uguale alla variabile *pid* allora verrà stampato che la terminazione è avvenuta in maniera normale. In linea di principio non sappiamo come il figlio abbia terminato la sua esecuzione, per stampare un valore corretto di *status* si dovrebbero usare le macro precedenti.

```
pid_t pid;
int status;

  pid=fork();
  if (pid==0)  /* figlio */
    exit(128);   /* qualsiasi numero */
  if (wait(&status) == pid)
     printf("terminazione normale\n: %d",
  status);
```

In questo caso si effettua una fork e nel figlio avviene una **abort()** che non fa nient'altro che generare (il kernel invia al processo chiamante) un segnale di **SIGABRT**. L'effetto di questo segnale e di far terminare immediatamente in maniera anormale l'esecuzione di un process. Questo processo figlio che era stato creato, il suo stato di terminazione manderà queste 2 informazioni: terminato in maniera anormale e negli 8 bit meno significativi ci sarà il valore del segnale che ha provocato la terminazione (il numero associato a **SIGABRT**). Anche in questo caso è necessario l'uso delle macro precedenti per sapere con precisione quale sia stata la terminazione del processo figlio.

```
pid_t pid;
int status;

  pid = fork();
 if (pid==0)   /* figlio  */
     abort();   /* genera il segnale SIGABRT */
 if (wait(&status) == pid)
    printf("terminazione anormale con abort\n: %d"status);
```

Si effettua una fork. Successivamente nel processo figlio si effettua una banale printf in cui viene stampato il pid del processo figlio. Successivamente una **exit(0)**, dunque terminazione del processo figlio. Il processo padre continuerà da solo l'esecuzione del codice, e farà una **sleep(2)** che ci garantisce che il figlio riesca a terminare la sua esecuzione prima che il padre si risvegli dalla **sleep**. Subito dopo viene usata una system call che permette di eseguire un comando di sistema **ps -T** che mostra tutti i processi attivi all'interno della shell. L'opzione **-T** mostra anche lo status attuale di tutti i processi visibili all'interno della shell. In particolare, se si va a vedere che il processo figlio sarà ancora visualizzato pur avendo terminato la sua esecuzione e nella colonna di stato comparirà una **STAT Z**. Quindi si vedrà che è un processo zombie. All'atto dell **exit(0)**, il processo figlio diventerà figlio di **init** che effettuerà una chiamata **wait**.

```
int main()
{ pid_t pid;

   if ((pid=fork()) < 0)
     printf("fork error");
   else if (pid==0){            /*  figlio */
        printf("pid figlio= %d",getpid());
      exit(0);
      }
   sleep(2);                    /*  padre */
   system("ps -T");  /* dice che il figlio è
  zombie... STAT Z*/
   exit(0);
}
```

Ogni volta che dei processi tentano di fare qualcosa con dati condivisi e il risultato finale dipende dall'ordine in cui i processi sono eseguiti sorgono delle **race conditions**.

La funzione **charatatime** si comporta nel seguente modo: innanzitutto si setta lo standard output ad **unbuffered**, dunque, non c'è più buffering associato allo standard output. A questo punto c'è un ciclo banale che scandisce ogni lettera dalla stringa passata in output e la stampa.
Il programma, banalmente effettua una fork. Per il processo figlio viene chiamata **charatatime** con una certa stringa, mentre per il padre un'altra stringa.
In questo caso l'output in linea di principio non si sa, potremo avere un mix qualunque di caratteri. La risorsa condivisa in questo caso è lo standard output, e infatti è capitata una race conditions. Questa è una problematica che si affronta con i semafori. Nel caso di padre-figlio, la gestione della race conditions avviene attraverso uno standard.

```
int main(void){
  pid_t     pid;

  pid = fork();
  if (pid==0) {charatatime("output dal figlio\n"); }
            else { charatatime("output dal padre\n"); }
  exit(0);

}

static void charatatime(char *str)
{char *ptr;   int c;
  setbuf(stdout, NULL);       /* set unbuffered */
  for (ptr = str; c = *ptr++; )
      putc(c, stdout);
```

Volendo sincronizzare un processo padre ed un processo figlio possiamo tentare di utilizzare strumenti che abbiamo già introdotto: se un processo padre vuole aspettare che un figlio termini deve usare una delle **wait**.
Lo schema generale di quanto appena detto è il seguente. Si nota come la prima cosa che il padre fa è la **wait**. Si nota come in questa **wait** non viene passato alcun parametro al suo interno.

```
pid = fork();

if (!pid){                    /* figlio  */

   /* il figlio fa quello che deve fare */

     }

else{   /* padre */

   wait();

       /* il padre fa quello che deve fare */

}
```

Se un processo figlio vuole aspettare che il padre termini si può tentare di utilizzare i segnali.

Con questa porzione di codice, innanzitutto viene creato un processo figlio attraverso la fork. Se siamo nel figlio, ci sarà un ciclo while. Fintantoché il padre (getppid restituisce il pid del padre) è diverso da 1 allora cicla inutilmente. Per quanto riguarda il padre, continuerà la sua esecuzione e prima o poi troverà una exit. All'atto della terminazione, il processo figlio diventerà il figlio di init, dunque gettpid diventerà uguale a 1, fallirà il while e il figlio potrà continuare il suo codice.

```
#include <signal.h>
void catch(int);
int main (void) {
pid = fork();
if (!pid){      /* figlio  */
      while ( getppid() != 1) ;
     /* il figlio fa quello che deve fare */
} else{       /* padre */
     /* il padre fa quello che deve fare */

}
```

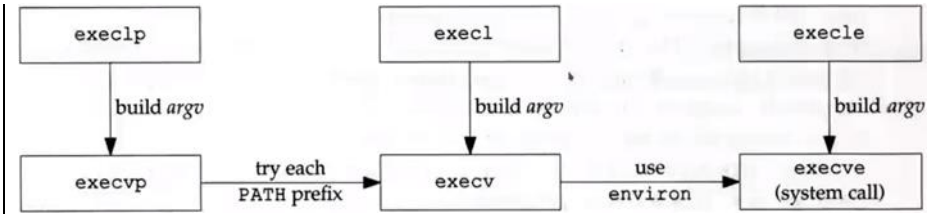
4.5 EXEC

La fork di solito è usata per creare un nuovo processo (figlio). Di norma, questo processo, una volta creato, esegue un nuovo programma che può eseguire attraverso la funzione **exec**, l'effetto è che il processo figlio è rimpiazzato dal nuovo programma ed inizia l'esecuzione col suo **main**.
L'unico modo per creare un processo è attraverso la **fork**, mentre l'unico modo per eseguire un eseguibile (o comando) è attraverso la **exec**. La chiamata ad **exec** reinizializza un processo: il segmento di istruzioni ed il segmento dati utente cambiano (viene eseguito un nuovo programma) mentre il segmento dei dati di sistema rimane invariato.

```
#include <unistd.h>
int execl (const char *path, const char *arg0, ../* (char *) 0 */);
int execv (const char *path, char *const argv[ ]);
int execle (const char *path, const char *arg0, ../*(char *) 0, char *const envp[ ] */);
int execve (const char *path, char *const argv[ ], char *const envp[ ]);
int execlp (const char *file, const char *arg0, ../*(char *)0 */);
int execvp (const char *file, char *const argv[ ]);
```

Tutte queste funzioni cercano di eseguire all'interno di un processo corrente un nuovo programma. Nota: **l** sta per list mentre **v** sta per vector.
La funzione **execl**, come argomenti ha un *pathname* dell'eseguibile e successivamente un numero variabile di argomenti. Eseguire la **execl** significa all'interno del processo corrente, eseguire il comando che ha *pathname* come primo argomento, seguito dagli argomenti che sono indicati come parametri successivi. Questi argomenti sono essenzialmente quelli che ritroviamo nel nostro comune **argv[]**. Fondamentalmente gli argomenti della **execl** sono l'equivalente della linea di comando che eseguiremmo normalmente. La funzione **execv** segue la stessa linea di principio.
La funzione **execle** si basa sul seguente principio: se oltre al nome dell'eseguibile e agli argomenti bisogna anche passare l'environment list (variabili d'ambiente), allora si chiama questa funzione. Stesso ragionamento per **execve**.
La funzione **execlp** il primo argomento è un *filename* dell'eseguibile. In questo caso, quando si passa un nome di file di un eseguibile, quest'eseguibile sarà cercato all'interno delle variabili directory contenute nella variabile *path*. Si usa quando il comando da eseguire è un comando di sistema
Tutte quante restituiscono -1 in caso di errore, non ritornano se OK. Dal punto di vista pratico significa che quando viene chiamata una **exec** il codice chiamante non c'è più, e per questo che non ritornano niente in caso di funzione positiva.

In questa figura viene mostrato che di fatto che la vera system call che è utilizzata è la **execve**. Tutte le altre sono essenzialmente dei wrapper.
Logicamente, supponiamo di scrivere sulla shell **cat FILE1 FILE2**. Quando premiamo invio, tenendo conto che la shell è un programma, verrà individuato i pezzi del comando.



Viene effettuata una fork. Dopo la fork sarà costruita una **exec**. Se il primo argomento non inizia con un punto o con /, probabilmente sarà costruita una **execlp**("cat", "cat", "FILE1", "FILE2", *0); a partire dalla **execlp**, una volta organizzati gli argomenti di **argv**, viene la **execvp**("cat", argv) dove argv non è nient'altro che l'organizzazione dei parametri di **execlp**. Successivamente di questa **execvp**, il kernel va a cercare all'interno della variabile *path* in quale directory si trova il comando da eseguire. Supponiamo sia "\usr\bin\cat". A questo punto viene generata la **execv**("\usr\bin\cat", argv).
Visto che si deve usare l'environment attuale, viene invocata la **execve**("\usr\bin\cat", argv, envp).
Con la **exec** si chiude il ciclo delle primitive di controllo dei processi UNIX:

- **fork**: creazione nuovi processi.
- **exec**: esecuzione nuovi programmi.
- **exit**: trattamento fine processo.
- **wait/waitpid**: trattamento attesa fine processo.

Questo è un esempio estremamente atipico di una **exec.** Con la **execl** viene sostituito il codice visibile in figura e ci sarà il codice di **echo** che verrà caricato al posto di quello in figura. In più, questo programma avrà la sua lista degli argomenti in cui c'è argv[0]=echo, argv[1]="la" e così via…
echo permette di stampare in sequenza i vettori che gli si passano come argomento.
Alla fine, c'è un exit(0) che non esiste perché tutto quello che c'è dopo l'**exec** non verrà mai eseguito (in caso positivo della exec ovviamente).
L'output di questo programma sarà:

```
Sopra la panca
la capra campa
```

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

int main (void) {
printf("Sopra la panca \n");

execl("/bin/echo","echo","la","capra","campa",NULL);

exit(0);

}
```

In questo programma l'output dipende dalla positività della **exec**. Se viene eseguita correttamente l'output sarà uguale a quello precedente.
Se la **exec** fallisce l'output sarà quello precedente con l'aggiunta di "sotto la panca crepa"

```
Sopra la panca
la capra campa
```

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

int main (void) {
printf("Sopra la panca \n");
execl("/bin/echo","echo","la","capra","campa",NULL);
printf("sotto la panca crepa \n");
exit(0);
}
```

```
$a.out
Sopra la panca
la capra campa
sotto la panca crepa
$
```

```
$a.out
Sopra la panca
sotto la panca crepa
la capra campa
$
```

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

int main (void) {
printf("Sopra la panca \n");
pid = fork();
if (pid==0){
   execl("/bin/echo","echo","la","capra","campa",NULL);}
printf("sotto la panca crepa \n");
exit(0);
}
```

In questo caso viene eliminata la race conditions che poteva esserci precedentemente.

```
$a.out
Sopra la panca
la capra campa
sotto la panca crepa
$
```

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

int main (void) {
printf("Sopra la panca \n");
pid = fork();
if (!pid){
   execl("/bin/echo","echo","la","capra","campa",NULL);}
wait( );
printf("sotto la panca crepa \n");
exit(0);
}
```