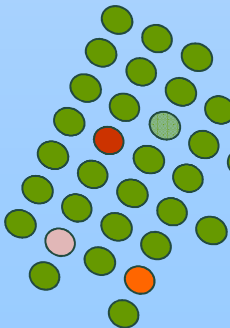
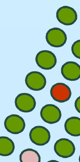
Processi in Unix

Capitolo 8 -- Stevens



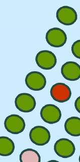
Controllo dei processi

- Creazione di nuovi processi
- Esecuzione di programmi
- Processo di terminazione
- Altro…



Identificatori di processi

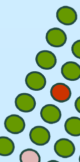
- Ogni processo ha un identificatore unico non negativo
- Process Id = 1 → il cui file di programma è contenuto in **/sbin/init**
  - invocato dal kernel alla fine del boot, legge il file di configurazione **/etc/inittab** dove ci sono elencati i file di inizializzazione del sistema (rc files) e dopo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



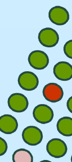
Identificatori di processi

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

#include <unistd.h>
```

pid_t getpid (void);	process ID del processo chiamante
pid_t getppid (void);	process ID del padre del processo chiamante
uid_t getuid (void);	real user ID del processo chiamante
uid_t geteuid (void);	effective user ID del processo chiamante
gid_t getgid (void);	real group ID del processo chiamante
gid_t getegid (void);	effective group ID del processo chiamante



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

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



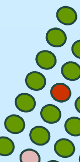
Creazione di nuovi processi

- L'unico modo per creare nuovi processi è attraverso la chiamata della funzione fork da parte di un processo già esistente

- quando viene chiamata, la fork genera un nuovo processo detto figlio

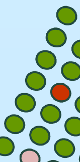
- dalla fork si ritorna **due** volte
  - il valore restituito al processo figlio è 0
  - il valore restituito al padre è il **pid** del figlio
    - un processo può avere più figli e non c’e’ nessuna funzione che puo’ dare al padre il pid dei suoi figli

- figlio e padre continuano ad eseguire le istruzioni che seguono la chiamata di fork



Creazione di nuovi processi

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



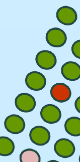
Funzione fork

```
#include <sys/types>

#include <unistd.h>


pid_t fork(void);
```

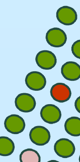
Restituisce: 0 nel figlio,

pid del figlio nel padre

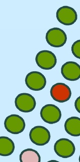
-1 in caso di errore



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

int glob=10; /* dati inizializzati */

char buf [ ]=“Scritta su stdout\n”;

int main(void)

{

    int var=100;   /* vbl sullostack */

    pid_t pippo;

      write(STDOUT_FILENO, buf, sizeof(buf)-1)

      printf(“prima della fork\n”);

       pippo=fork();

      if( (pippo == 0) {glob++; var++;}

        else sleep(2);

    printf(“pid=%d, glob=%d, var=%d\n”,getpid(),glob,var);

   exit(0);

}
```

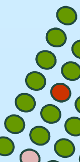


```
$ a.out

Scritta su stdout

prima della fork

pid=227, glob=11, var=101

pid=226, glob=10, var=100



$ a.out>temp.txt

$ cat  temp.txt
       Scritta su stdout
       prima della fork
       pid=229, glob=11, var=101
       prima della fork
       pid=228, glob=10, var=100
$
```



Condivisione file

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



Condivisione files tra padre/figlio

parent process table entry

fd flags ptr
fd 0:
fd 1:
fd 2:
. . .

child process table entry

fd flags ptr
fd 0:
fd 1:
fd 2:
. . .

file table

file status flags
current file offset
v-node ptr

file status flags
current file offset
v-node ptr

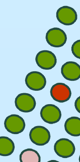
file status flags
current file offset
v-node ptr

v-node table

v-node information
i-node information
current file size

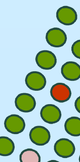
v-node information
i-node information
current file size

v-node information
i-node information
current file size



Problema della sincronizzazione

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



Esercizi

a) Si supponga di mandare in esecuzione il seguente programma:

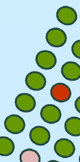
```
int main(void)
{
pid_t pid1, pid2;
pid1 = fork();
pid2 = fork();
exit(0);
}
```

Dire quanti processi vengono generati. Giusticare la risposta.

b) Si supponga di mandare in esecuzione il seguente programma:

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

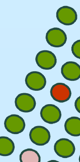
Dire quanti processi vengono generati. Giusticare la risposta



conclusioni: quando si usa **fork**?

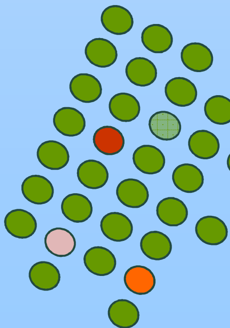
1. un processo attende richieste (p.e. da parte di client nelle reti) allora si duplica
   - il figlio tratta (handle) la richiesta
   - il padre si mette in attesa di nuove richieste

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



Ambiente di un processo

Capitolo 7 -- Stevens

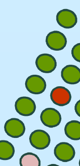


Avvio di un processo

Quando parte un processo:

- si esegue prima una routine di start-up speciale che prende
  - valori passati dal kernel dalla linea di comando
    - in *argv*[ ] se il processo si riferisce ad un programma C
  - variabili d'**ambiente**

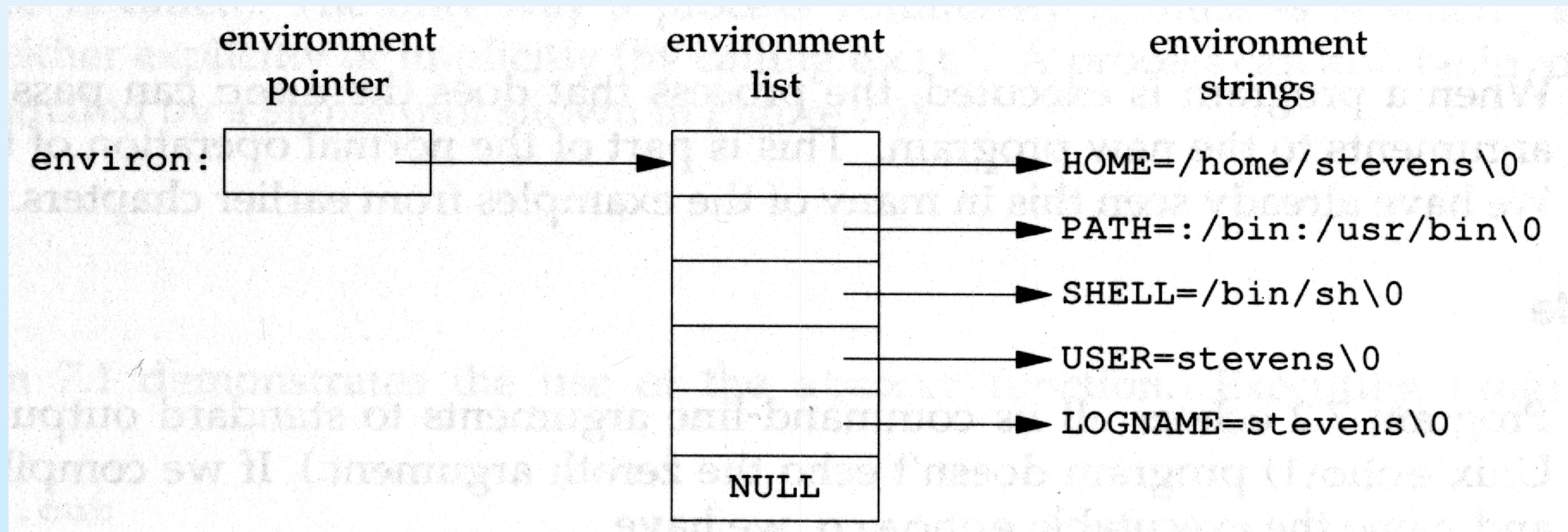
- successivamente viene chiamata la funzione principale da eseguire
  - int main(int *argc*, char **argv* [ ]);



Environment List

ad ogni programma è passato anche una lista di variabili di ambiente individuata dalla variabile

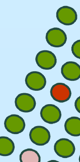
extern char **environ



Terminazione di un processo

- Terminazione normale
  - ritorno dal main
  - chiamata a exit
  - chiamata a _exit

- Terminazione anormale
  - chiamata abort
  - arrivo di un segnale



Funzioni exit

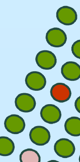
```
#include <stdlib.h>

 void exit (int status);
```

Descrizione: restituisce *status* al processo che chiama il programma includente exit ; effettua prima una *pulizia* e poi ritorna al kernel

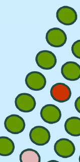
- effettua lo shoutdown delle funzioni di libreria standard di I/O (fclose di tutti gli stream lasciati aperti) => tutto l'output è flushed

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

Descrizione: ritorna immediatamente al kernel



```
#include <stdio.h>

int main(void)
{
    printf("Ciao a tutti");
    exit(0);
    _exit(0);
}
```



Exit handler

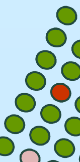
```
#include <stdlib.h>

int atexit (void (*funzione) (void));
```

Restituisce: 0 se O.K.

diverso da 0 su errore

funzione = punta ad una funzione che e' chiamata per cleanup il processo alla sua normale terminazione.

Il numero di exit handlers che possono essere specificate con atexit e' limitato dalla quantita' di memoria virtuale.

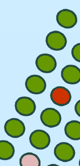


```
int main(void)
{
   atexit(my_exit2);      atexit(my_exit1);
   printf("ho finito il main\n");
   return(0);
}

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

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

Sostituiamo return(0) con _exit(0).

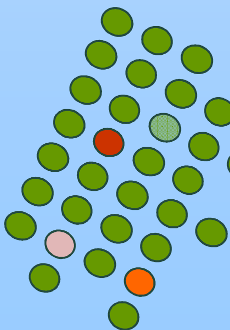
Che cosa succede mandando in esecuzione

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



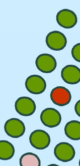
Segnali: Interrupt software per la gestione di eventi asincroni

Capitolo 10 - Stevens



Concetto di segnale

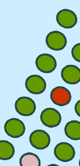
- Un segnale è un interrupt software
- Un segnale può essere generato da un processo utente o dal kernel a seguito di un errore software o hardware
- Ogni segnale ha un nome che comincia con SIG (ex. SIGABRT, SIGALARM) a cui viene associato una costante intera ($\neq 0$) positiva definita in **signal.h**
- Il segnale e' un evento asincrono; esso puo' arrivare in un momento qualunque ad un processo ed il processo puo' limitarsi a verificare, per esempio, il valore di una variabile, o può fare cose piu' specifiche



Azione associata ad un segnale

Le azioni associate ad un segnale sono le seguenti:

- **Ignorare il segnale** (tranne che per SIGKILL e SIGSTOP)
- **Catturare il segnale** (equivale ad associare una funzione utente quando il segnale occorre; ex. se il segnale SIGTERM e' catturato possiamo voler ripulire tutti i file temporanei generati dal processo)
- **Eseguire l'azione di default associata** (terminazione del processo per la maggior parte dei segnali)



Segnali in un sistema Unix

Name	Description	ANSI C	POSIX.1	SVR4	4.3+BSD	Default action
SIGABRT	abnormal termination (abort)	•	•	•	•	terminate w/core
SIGALRM	time out (alarm)		•	•	•	terminate
SIGBUS	hardware fault			•	•	terminate w/core
SIGCHLD	change in status of child		job	•	•	ignore
SIGCONT	continue stopped process		job	•	•	continue/ignore
SIGEMT	hardware fault			•	•	terminate w/core
SIGFPE	arithmetic exception	•	•	•	•	terminate w/core
SIGHUP	hangup		•	•	•	terminate
SIGILL	illegal hardware instruction	•	•	•	•	terminate w/core
SIGINFO	status request from keyboard				•	ignore
SIGINT	terminal interrupt character	•	•	•	•	terminate
SIGIO	asynchronous I/O			•	•	terminate/ignore
SIGIOT	hardware fault			•	•	terminate w/core
SIGKILL	termination		•	•	•	terminate
SIGPIPE	write to pipe with no readers		•	•	•	terminate
SIGPOLL	pollable event (poll)			•		terminate
SIGPROF	profiling time alarm (setitimer)			•	•	terminate
SIGPWR	power fail/restart			•		ignore
SIGQUIT	terminal quit character		•	•	•	terminate w/core
SIGSEGV	invalid memory reference	•	•	•	•	terminate w/core
SIGSTOP	stop		job	•	•	stop process
SIGSYS	invalid system call			•	•	terminate w/core
SIGTERM	termination	•	•	•	•	terminate
SIGTRAP	hardware fault			•	•	terminate w/core
SIGTSTP	terminal stop character		job	•	•	stop process
SIGTTIN	background read from control tty		job	•	•	stop process
SIGTTOU	background write to control tty		job	•	•	stop process
SIGURG	urgent condition			•	•	ignore
SIGUSR1	user-defined signal		•	•	•	terminate
SIGUSR2	user-defined signal		•	•	•	terminate
SIGVTALRM	virtual time alarm (setitimer)			•	•	terminate
SIGWINCH	terminal window size change			•	•	ignore
SIGXCPU	CPU limit exceeded (setrlimit)			•	•	terminate w/core
SIGXFSZ	file size limit exceeded (setrlimit)			•	•	terminate w/core

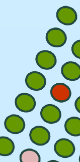
Funzione signal

```
#include <signal.h>
```

```
void (*signal(int signo, void (*func)(int)))(int);
```

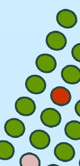
Restituisce: SIG_ERR in caso di errore e

il puntatore al precedente gestore del segnale se OK



Funzione signal

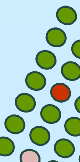
- prende due argomenti: il nome del segnale *signo* ed il puntatore alla funzione *func* da eseguire come azione da associare all'arrivo di *signo* (*signal handler*) e
- restituisce il puntatore ad una funzione che prende un intero e non restituisce niente che rappresenta il puntatore al precedente *signal handler*



Funzione signal

Il valore di *func* può essere:

- SIG_IGN per ignorare il segnale (tranne che per SIGKILL e SIGSTOP)
- SIG_DFL per settare l’azione associata al suo default
- L’indirizzo di una funzione che sarà eseguita quando il segnale occorre

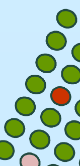


Funzioni kill e raise

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

#include <signal.h>



int kill (pid_t pid, int signo);

int raise (int signo);
```

Descrizione: mandano il segnale *signo* specificato come argomento

Restituiscono: 0 se OK,

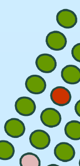


Funzioni kill e raise

kill manda un segnale ad un processo o ad un gruppo di processi specificato da *pid*

raise consente ad un processo di mandare un segnale a se stesso

- *pid* > 0 invia al processo *pid*
- *pid* == 0 invia ai processi con lo stesso *gid* del processo sender
- *pid* < *0* invia ai processi con *gid* uguale a |*pid*|



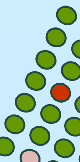
Funzione pause

#include <unistd.h>

int pause(void);

Descrizione: sospende il processo finché non arriva un segnale ed il corrispondente signal handler è eseguito ed esce

Restituisce: –1



```
#include <signal.h>
void sig_usr(int);
int main (void) {
 signal(SIGUSR1, sig_usr);
 signal(SIGUSR2, sig_usr);
 for(; ;)  pause();
 }
void sig_usr(int signo) {
 if(signo == SIGUSR1) printf(“SIGUSR1\n”);
 else if (signo == SIGUSR2) printf(“SIGUSR2\n”);
       else printf(“Segnale %d\n”,signo);
}
```

uso dell'esempio

- lanciare in background (&) il programma precedente e vedere con che pid gira
- scrivere un programma che manda il segnale a questo processo con

  kill(pid, SIGUSR1) ;

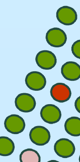
  kill(pid, SIGUSR2) ;

- oppure usare kill(1) da linea di comando

  kill –USR1 pid //si otterra' SIGUSR1

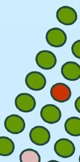
  kill –USR2 pid //si otterra' SIGUSR2

  kill pid //si sta mandando SIGTERM e con esso il processo termina perche' tale segnale non e' catturato e di default termina



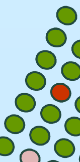
esercizio

- scrivere un programma che intercetta il ctrl-C da tastiera (SIGINT) e gli fa stampare un messaggio.



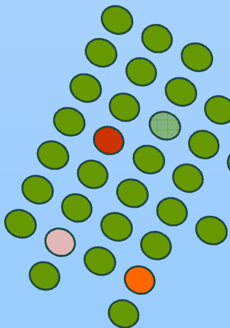
Funzione sleep

```
#include <unistd.h>

unsigned int sleep (unsigned int secs);
```



Controllo dei processi in UNIX

Capitolo 8 -- Stevens

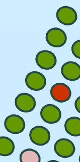


Funzioni wait e waitpid

- quando un processo termina il kernel manda al padre il segnale SIGCHLD

- il padre può ignorare il segnale (default) oppure lanciare una funzione (signal handler)

- in ogni caso il padre può chiedere informazioni sullo stato di uscita del figlio; questo è fatto chiamando le funzioni wait e waitpid.



Funzione wait

#include <sys/types.h>

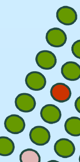
#include <sys/wait.h>

pid_t wait (int *statloc*);

Descrizione: chiamata da un processo padre ottiene in *statloc* lo stato di terminazione di un figlio

Restituisce: PID se OK,

-1 in caso di errore



Funzione waitpid

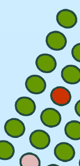
```
#include <sys/types.h>

#include <sys/wait.h>
```

```
pid_t waitpid (pid_t pid, int *statloc, int options);
```

Descrizione: chiamata da un processo padre chiede lo stato di terminazione in *statloc* del figlio specificato dal *pid* 1° argomento; tale processo padre si blocca in attesa o meno secondo il contenuto di *options*

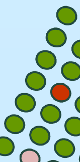
Restituisce: PID se OK,

0 oppure -1 in caso di errore



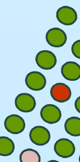
Funzione **waitpid**

- *pid* == -1 (qualsiasi figlio...la rende simile a wait)
- *pid* > 0 (pid del figlio che si vuole aspettare)
- *options* =
  - 0 (niente... come wait)
  - WNOHANG (non blocca se il figlio indicato non è disponibile)



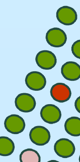
differenze

- in generale con la funzione wait
  - il processo si blocca in attesa (se tutti i figli stanno girando)
  - ritorna immediatamente con lo stato di un figlio
  - ritorna immediatamente con un errore (se non ha figli)
- un processo puo’ chiamare wait quando riceve SIGCHLD, in questo caso ritorna immediatamente con lo stato del figlio appena terminato
- waitpid può scegliere quale figlio aspettare (1° argomento)
- wait può bloccare il processo chiamante (se non ha figli che hanno terminato), mentre waitpid ha una opzione (WNOHANG) per non farlo bloccare e ritornare immediatamente



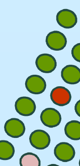
Terminazione

- in ogni caso il kernel esegue il codice del processo e determina lo *stato di terminazione*
  - se normale, lo stato è l'argomento di
    - exit, return oppure _exit
  - altrimenti 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



Cosa succede quando un processo termina?

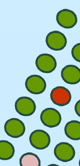
- Se un figlio termina prima del padre, allora il padre ottiene lo status del figlio con wait
- Se un padre termina prima del figlio, allora il processo init diventa il nuovo padre
- Se un figlio termina prima del padre, ma il padre non utilizza wait, allora il figlio diventa uno zombie



esempio: terminazione normale

```
pid_t pid;

int status;



  pid=fork();

  if (pid==0)   /* figlio */

     exit(128);   /* qualsiasi numero */

  if (wait(&status) == pid)

      printf("terminazione normale\n");
```

vedi fig. 8.2 per le macro per la verifica di status e per stampare lo stato di terminazione



esempio: terminazione con abort

```
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");
```

zombie.c

```
int main()
{ pid_t pid;


   if ((pid=fork()) < 0)
     err_sys("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);
}
```

Race Conditions

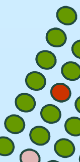
- ogni volta che dei processi tentano di fare qualcosa con dati condivisi e il risultato finale dipende dall'ordine in cui i processi sono eseguiti sorgono delle race conditions

1. se si vuole che un figlio aspetti che il padre termini si può usare:

```
while ( getppid() != 1)
    sleep(1);
```

2. se un processo vuole aspettare che un figlio termini deve usare una delle wait

- la prima soluzione spreca molta CPU, per evitare ciò si devono usare i segnali oppure qualche forma di IPC (interprocess communication).

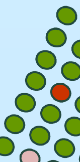


esempio di race conditions

```
int main(void){
    pid_tpid;

    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);

}
```

output dell'esempio

```
/home/studente > a.out

output from child

output from parent



/home/studente > a.out

oouuttppuutt ffrroomm cphairlednt

...
```



figlio esegue prima del padre (esempio)

```
pid = fork();

if (!pid){                         /* figlio  */

   /* il figlio fa quello che deve fare */

     }

else{            /* padre */

   wait();

       /* il padre fa quello che deve fare */

}
```

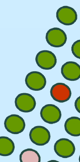
padre esegue prima del figlio (esempio)

```
#include <signal.h>

void catch(int);

int main (void) {

pid = fork();

if (!pid){                              /* figlio  */

    signal(SIGALRM,catch);

    pause();

      /* il figlio fa quello che deve fare */

} else{                         /* padre */

        /* il padre fa quello che deve fare */

        kill(pid,SIGALRM);

}


void catch(int signo) {

printf("parte il figlio");

}
```

primitive di controllo

- con la exec è chiuso il ciclo delle primitive di controllo dei processi UNIX

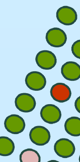
1. fork → creazione nuovi processi
2. exec → esecuzione nuovi programmi
3. exit → trattamento fine processo
4. wait/waitpid→trattamento attesa fine processo



Funzioni exec

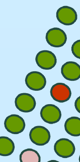
- fork di solito è usata per creare un nuovo processo (il figlio) che a sua volta esegue un programma chiamando la funzione exec.

- in questo caso il figlio è completamente rimpiazzato dal nuovo programma e questo inizia l'esecuzione con la sua funzione main
  - non è cambiato il pid... l'address space è sostituito da un nuovo programma che risiedeva sul disco



Funzioni exec

- L'unico modo per creare un processo è attraverso la fork
- L'unico modo per eseguire un eseguibile (o comando) è attraverso la exec
- La chiamata ad exec reinizializza un processo: il segmento istruzioni ed il segmento dati utente cambiano (viene eseguito un nuovo programma) mentre il segmento dati di sistema rimane invariato

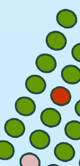


Funzioni exec

```c
#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[ ]);
```

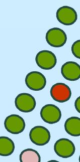
Restituiscono: -1 in caso di errore

non ritornano se OK

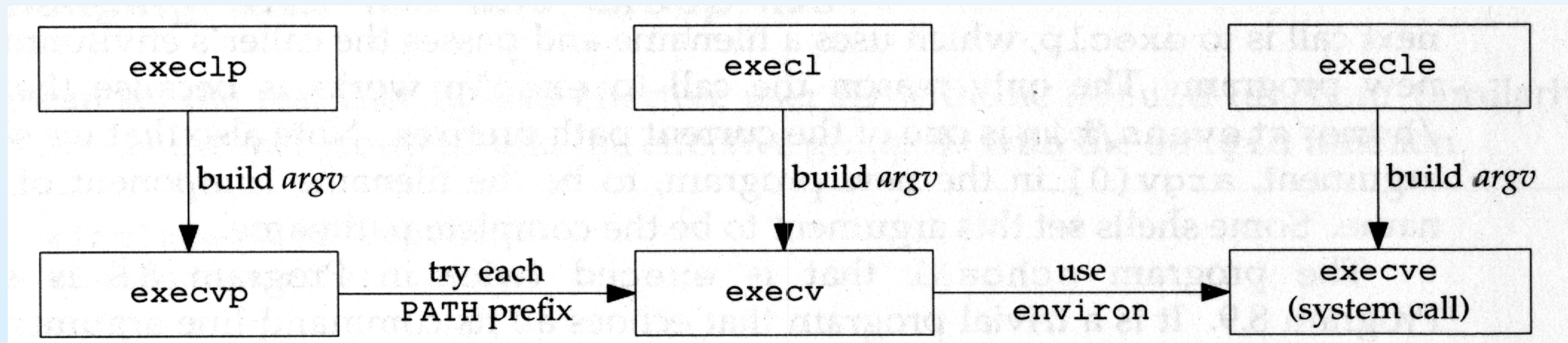


Funzioni exec - *differenze*

- Nel nome delle exec l sta per list mentre v sta per vector
  - execl, execlp,execle prendono come parametro la lista degli argomenti da passare al *file* da eseguire
  - execv, execvp,execve prendono come parametro l'array di puntatori agli argomenti da passare al *file* da eseguire
- execlp ed execvp prendono come primo argomento un *file* e non un *pathname*, questo significa che il file da eseguire e' ricercato in una delle directory specificate in PATH
- execle ed execve passano al *file* da eseguire la enviroment list; un processo che invece chiama le altre exec copia la sua variabile environ per il nuovo *file* (programma)



Relazione tra le funzioni exec

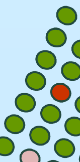


```
……

……

printf("Sopra la panca \n");

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

……
```

```
$a.out
Sopra la panca
la capra campa
$
$a.out>temp.txt
$cat temp.txt
la capra campa
```

esempio: echoenv.c

```
#include <stdlib.h>

extern **char environ;


int main(){

   int i = 0;


   while (environ[i])

        printf("%s\n", environ[i++]);


   return (0);

}
```



esempio di execl[ep]

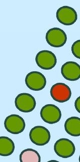
```
#include<sys/types.h>
#include<sys/wait.h>
#include"ourhdr.h"
char    *env_init[ ]={"USER=studente", "PATH=/tmp", NULL };
int main(void){
  pid_t pid;
  pid = fork();
  if (!pid) {   /* figlio */
   execle("/home/studente/echoenv","echoenv",(char*) 0,env_init);
   }
   waitpid(pid, NULL, 0);
   printf("sono qui \n\n\n");
   pid = fork();
   if (pid == 0){/* specify filename, inherit environment */
       execlp("echoenv", "echoenv", (char *) 0);
   }
   exit(0);
}
```

Start-up di un programma

L’esecuzione di un programma tramite fork+exec ha le seguenti caratteristiche

- Se un segnale è **ignorato** nel processo padre viene ignorato anche nel processo figlio

- Se un segnale è **catturato** nel processo padre viene assegnata l’azione di default nel processo figlio

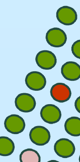


Funzione system

```
#include <stdlib.h>
```

```
int system (const char *cmdstring);
```

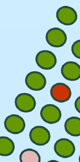
- Serve ad eseguire un comando shell dall'interno di un programma
- esempio: system("date > file");
- essa e' implementata attraverso la chiamata di fork, exec e waitpid



Esercizio 1

1. Scrivere un programma che crei un processo zombie.

2. Fare in modo che un processo figlio diventi figlio del processo ***init***.



Esercizio 3

Scrivere un programma che effettui la copia di un file utilizzando 2 figli:

- uno specializzato nella copia delle vocali ed
- uno nella copia delle consonanti.

Per la sincronizzazione tra i processi figli utilizzare un semaforo implementato tramite file.