UNIVERSITÀ DEGLI STUDI DI PARMA
Dipartimento di Ingegneria e Architettura
Corso di Laurea in Ingegneria informatica, elettronica e delle telecomunicazioni
a.a. 2021-2022

SISTEMI OPERATIVI

Esercitazione 4

1 Primitive per il controllo dei processi

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

Un processo paedre crea un processo figlio chiamando la primitiva fork(). Il processo figlio esegue lo stesso codice, dispone di una copia dell'area dati, della tabella dei file aperti e di altre informazioni ereditate dal padre, ma ovviamente ha un PID diverso da quello del padre. La fork() restituisce un valore intero. Nel processo figlio tale valore è zero, nel processo padre corrisponde al PID del figlio.

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

La funzione getpid() restituisce il PID del processo corrente. La funzione getppid() restituisce il PID del processo padre del processo corrente.

```
#include <stdlib.h>
voi exit(int status);
```

La exit chiude tutti i file aperti, per il processo che termina. Il valore ritornato viene passato al processo padre, se questo attende il processo che termina.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

La primitiva wait() attende la terminazione di un qualunque processo figlio. Se il processo figlio termina con exit(esito), l'esito, ovvero il valore di uscita, del figlio viene recuperato dal processo padre utilizzando la macro `WEXITSTATUS(status)`. Da notare anche le macro `WIFEXITED(status)` e `WIFSIGNALED(status)` che ritornano vero rispettivamente se il figlio è terminato normalmente o se è stato terminato da un segnale.

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

La primitiva waitpid (per default options=0) sospende l'esecuzione del processo chiamante finchè il processo figlio identificato da *pid* termina. Se un processo figlio è già terminato al momento della invocazione di waitpid, essa ritorna immediatamente. La primitiva ritorna il PID del processo figlio terminato. Se $pid = -1$ la waitpid attende il completamento di un qualunque processo figlio.

```
#include <unistd.h>
unsigned int sleep(unsigned int secs);
```

La funzione sleep sospende un processo per un periodo di tempo pari a *secs* secondi.

```
#include <unistd.h>
int execve(char *file_name, *argv[], *envp[]);
```

La primitiva execve non produce nuovi processi ma solo il cambiamento dell'ambiente di esecuzione del processo interessato. Il processo corrente passa ad eseguire un nuovo programma (eseguibile o script) il cui *path* è specificato dal primo argomento *file_name*. Il secondo argomento è un puntatore ad un vettore di puntatori a carattere che rappresentano gli argomenti di invocazione del programma. Il terzo parametro (puntatore ad una lista di puntatori a carattere) consente di specificare variabili d'ambiente per il nuovo programma che si aggiungono a quelle ereditate dal processo padre.

2 Esempi

> Tutti i file con il codice sorgente degli esercizi proposti (es∗.c) si trovano nel direttorio eserc4 della cartella con i file delle esercitazioni (ad es. ∼/**so-esercitazioni**).

Esercizio 0: padre e figlio

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

int main()
{
  printf("Sono il processo con PID=%d e sono figlio
       del processo con PID=%d\n", getpid(), getppid());
}
```

Si esegua il programma più volte e si notino i valori dei PID visualizzati. Come si spiegano tali valori?

Esercizio 1: esecuzione di istruzioni indifferenziate tra padre e figlio

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

```
#include <stdio.h>

int main()
{
  int pid;

  if ((pid=fork()) < 0)
    {
      perror("Errore fork");
      exit(1);
   }

   /* Entrambi i processi eseguono la printf */
   printf("Processo PID=%d - dalla fork ho ottenuto %d\n",getpid(),pid);

}
```

Provare ad eseguire il programma anche con `strace` utilizzando l'opzione `-f` per tracciare anche le system call invocate dai processi figli: *strace -f ./es1*

Esercizio 2: esecuzione di istruzioni differenziate tra padre e figlio

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

int main()
{
  int pid;

  if ((pid=fork()) < 0)
    {
      perror("Errore fork");
      exit(1);
    }
  else
   if (pid == 0)
    {/* CODICE ESEGUITO DAL FIGLIO */
      printf("Sono il processo figlio con PID=%d\n", getpid());
    }
  else
    {/* CODICE ESEGUITO DAL PADRE */
      printf("Sono il processo padre con PID=%d e ho generato un figlio
che ha PID=%d\n", getpid(),pid);
    }
}
```

Esercizio 3: come il precedente, ma con una variabile modificata nel processo figlio per dimostrare **l'indipendenza delle aree di memoria dei processi** e l'uso della sleep():

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

int main ()
{
  int pid, status;
  int myvar;


   myvar = 1 ;

   if ((pid = fork()) < 0) /* Il figlio eredita una copia dell'area del padre */
    {
      perror("Errore fork");
      exit(1);
    }
   else
   if (pid==0)
    {
      printf("Figlio: sono il processo %d e
      sono figlio di %d \n", getpid(), getppid());

      printf("Figlio: attendo 2 secondi...\n"
      sleep(2);
      myvar = 2;  /* Il figlio modifica la propria variabile myvar,
                       quindi nel proprio spazio di memoria */


      printf("Figlio: myvar=%d \n", myvar);
      exit(0);
    }
   else
    {
      wait(&status);
      printf("Padre: sono il processo %d e sono figlio di %d \n",
              getpid(), getppid());
      printf("Padre: status = %d \n", WEXITSTATUS(status));
      printf("Padre: myvar=%d \n", myvar);
      // La variabile è stata modificata dopo l'assegnamento iniziale ?
    }
}
```

Perché la variabile *myvar* nel processo padre ha ancora il valore iniziale?

Esercizio 4:

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

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

int main()
{
  pid_t pid;

  if ((pid=fork()) < 0)
    {
      perror("Errore fork");
      exit(1);
    }


  printf("ciao, pid vale %d!\n", pid);
  exit(0);
}
```

Modificare il programma aggiungendo altre fork() e verificando che il numero dei processi generati cresce esponenzialmente. Modificare il programma visualizzando il vero PID di ogni processo utilizzando la funzione getpid().

Esercizio 5: Il processo figlio crea un nuovo file (il cui nome deve essere specificato come argomento), il processo padre attende il completamento del figlio e successivamente legge il contenuto del file.

```
#include <fcntl.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define N 256

int main (int argc, char **argv)
{
  int nread, nwrite = 0, atteso, status, fileh, pid;
  char st1[N];
  char st2[N];


  if (argc != 2)
    {
    fprintf(stderr,"Uso: %s nomefile\n'',argv[0]);
    exit(1);
    }
```

```
  /* APERTURA IN LETTURA/SCRITTURA */
  fileh = open(argv[1], O_CREAT|O_RDWR|O_TRUNC, 0644);
  if (fileh == -1) {
    perror("Errore open");
exit(2);
}

  if((pid=fork()) < 0)
    {
      perror("Errore fork");
      close(fileh);
      exit(3);
    }
  else if (pid==0)
    {
      /* FIGLIO: legge una stringa che l'utente immette da tastiera */
      printf("Scrivere una stringa (senza spazi) e premere Invio\n");
      scanf("%s",st1);
      nwrite = write(fileh, st1, strlen(st1)+1);
      if (nwrite == -1) {
perror("Errore write");
exit(4);
}
      exit(0);
    }
  else
    {
      atteso=wait(&status); /* ATTESA DEL FIGLIO */
      printf("Il figlio con PID=%d e' terminato con
       stato=%d\n",atteso,WEXITSTATUS(status));


      // Riposizionamento del file offset all'inizio del file
      lseek(fileh, 0, SEEK_SET);

      nread = read(fileh, st2, N);
      if (nread == -1)
        perror("Errore read");

      printf("nread=%d\n",nread);
      printf("Il figlio ha scritto la stringa %s\n", st2);
      close(fileh);
      return(0);
    }

  exit(0);
}
```

Esercizio 6: il padre crea N figli e attende la fine dalla loro esecuzione

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

#include <stdio.h>
#include <stdlib.h>


#define N 8

int main()
{
  int status, i;
  pid_t pid;

  /* IL PADRE CREA N PROCESSI FIGLI */
  for (i=0; i<N ; i++)
    if ((pid=fork())==0)
    {
      sleep(1);
      exit(10+i);
    }

  /* IL PADRE ATTENDE I FIGLI */
  while ((pid=waitpid(-1, &status, 0)) > 0)
  {
    if (WIFEXITED(status)) /* ritorna 1 se il figlio è terminato correttamente */
      printf("Il figlio %d è terminato correttamente con exit status=%d\n",
              pid, WEXITSTATUS(status));
    else
      printf("Il figlio %d non è terminato correttamente\n",pid);
  }

  exit(0);
}
```

Modificare il programma in modo tale che il processo padre attenda il completamento dei processi figli nello stesso ordine in cui sono stati creati utilizzando sempre la primitiva `waitpid`.

Esercizio 7: utilizzo di execve()

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

```
int main()
{
  int status;
  pid_t pid;
  char* arg[]= {"ls", "-l", "/usr/include", (char *)0};
  char* env[]= { (char *)0};

  if ((pid=fork())==0)
  {
    /* CODICE ESEGUITO DAL FIGLIO */
    execve("/bin/ls", arg , env);
    /* Si torna qui solo in caso di errore */
    perror("Errore exec");
    exit(1);
  }
  else
  {
    /* CODICE ESEGUITO DAL PADRE */
    wait(&status);
    printf("exit di %d con %d\n", pid, status);
  }

  exit(0);
}
```

Esercizio 8: Il programma richiede la presenza nel direttorio corrente di due file di testo *f1* e *f2*.

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

int main()
{
  int status;
  pid_t pid;
  char *env[] = {"TERM=vt100", "PATH=/bin:/usr/bin", (char *) 0 };
  char *args[] = {"cat", "f1", "f2", (char *) 0};

  if ((pid=fork())==0)
  {
    /* CODICE ESEGUITO DAL FIGLIO */
    execve("/bin/cat", args, env);
    /* Si torna solo in caso di errore */
    perror("Errore execve");
```

```
    exit(1);
  }
  else
  {
    /* CODICE ESEGUITO DAL PADRE */
    wait(&status);
    printf("exit di %d con %d\n", pid, WEXITSTATUS(status));
  }

  exit(0);
}
```

Si modifichi il programma in modo che recuperi dai propri argomenti di invocazione i nomi dei due file di testo da far visualizzare al programma *cat*.

Esercizio proposto:

Realizzare un programma C con le seguenti caratteristiche:

1. deve creare un processo figlio ;
2. il processo figlio deve eseguire il comando passato come argomento al programma:
   esempio 1: *./run cp file1.txt file2.txt*
   esempio 2: *./run rm file1.txt*
   [nei precedenti esempi, *run* è l'eseguibile che dovete creare]
3. il processo padre deve attendere il completamento del figlio.

   Suggerimento: in cosa differiscono il vettore degli argomenti ricevuto dal programma *run* da quello necessario per l'esecuzione del comando che gli viene passato come argomento?