

## CORSO DI SISTEMI OPERATIVI A - ESERCITAZIONE 4

### 1 Primitive per il controllo dei processi

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

Un processo padre crea un processo figlio chiamando la funzione `fork()`. Il processo figlio esegue lo stesso codice, dispone di una copia dell'area dati e della tabella dei file aperti del padre, ma possiede un diverso PID. La funzione `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.

```
void 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/wait.h>
pid_t wait(int *status);
```

La primitiva `wait()` attende la terminazione di un qualunque processo figlio. Se il processo figlio termina con una `exit()` il secondo byte meno significativo di `status` è pari all'argomento passato alla `exit()` e il byte meno significativo è zero.

```
#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.

```
int execl(char *file_name, *argv[]);
```

La primitiva `execl` non produce nuovi processi ma solo il cambiamento dell'ambiente di esecuzione del processo interessato. Il processo corrente passa ad eseguire un nuovo comando 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.

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

La primitiva `execve` ha lo stesso comportamento di `execl`. L'unica differenza è la presenza di un terzo parametro (puntatore ad una lista di puntatori a carattere) che consente di specificare le variabili d'ambiente del nuovo programma.

## 2 Esempi

Tutti i file con il codice sorgente degli esercizi proposti (*es\*.c*) si trovano nel direttorio:

```
/temi_esame/soa/eserc4
```

Esercizio 1: padre e figlio

```
#include <unistd.h>

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

Esercizio 2: esecuzione di istruzioni differenziate tra padre e figlio

```
#include <unistd.h>

int main()
{
    int pid;

    if ((pid=fork()) < 0)
    {
        perror("fork error");
        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 qualche variabile in più e l'uso della `sleep()`

```

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

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

    myvar = 1 ;

    if ((pid = fork()) < 0) /* Il figlio eredita una copia dell'area del padre */
    {
        perror("fork error");
        exit(1);
    }
    else
    if (pid==0)
    {
        sleep(2);
        myvar = 2; /* Il figlio modifica la propria variabile myvar */
        printf("Figlio: sono il processo %d e sono figlio di %d \n",
            getpid(), getppid());
        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);
    }
}

```

Esercizio 4:

```

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

int main()
{
    pid_t pid;

    if ((pid=fork()) < 0)
    {
        perror("fork error\n");
        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>

#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("open error");

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

    lseek(fileh, 0, SEEK_SET);
    nread = read(fileh, st2, N);
    if (nread == -1)
        perror("read error");
    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 aspetta la fine dalla loro esecuzione

```

#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.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 ha terminato correttamente */
        printf("Il figlio %d ha terminato correttamente con exit status=%d\n",
            pid, WEXITSTATUS(status));
    else
        printf("Il figlio %d non ha 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.

*Esercizio 7*: utilizzo di `execv()`

```

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

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

    if ((pid=fork())==0)
    {
        /* CODICE ESEGUITO DAL FIGLIO */
        execv("/bin/ls", arg);
        /* Si torna solo in caso di errore */
        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/types.h>
#include <sys/wait.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 */
        exit(-1);
    }
    else
    {
        /* CODICE ESEGUITO DAL PADRE */
        wait(&status);
        printf("exit di %d con %d\n", pid, WEXITSTATUS(status));
    }

    exit(0);
}
```

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: `./run cp file1.txt file2.txt`  
esempio: `./run rm file1.txt`  
[nei precedenti esempi, *run* è l'eseguibile che dovete creare]
3. il processo padre attende il completamento del figlio.

### Assegnamento(opzionale):

Si realizzi un programma C che realizzi un *mini-shell* con le seguenti caratteristiche :

1. presenti all'utente un prompt, ad es. \$
2. recuperi dalla linea digitata dall'utente il comando e i suoi eventuali argomenti
3. sia in grado di gestire direttamente due comandi interni :
  - *printenv* : visualizza tutte le variabili d'ambiente
  - *exit* : fa terminare il *mini-shell*
4. esegua in foreground altri comandi mediante l'utilizzo di una primitiva della famiglia `exec`

Si suggerisce l'utilizzo della funzione C `strtok` per estrarre comando e argomenti dalla linea digitata dall'utente.

### **Valutazione e regole dell'assegnamento**

**Una valutazione positiva dell'assegnamento permetterà di migliorare fino a un massimo di due punti il risultato conseguito nella prova UNIX in un qualsiasi appello di questo A.A.**

La soluzione all'assegnamento dovrà essere personale (ovvero non sono ammesse soluzioni presentate da più di uno studente) e originale (verrà quindi verificata l'assenza di plagio). Per la valutazione il sorgente del programma (denominato con la propria matricola e contenente in un commento matricola, nome e cognome) dovrà essere caricato nella sezione Elaborati del sito del corso ([corsi.unipr.it](http://corsi.unipr.it)) entro la mezzanotte di domenica 30 maggio.