Experiment 2
Process Creation and Execution[1]

Objectives

- ✓ To learn how to create, terminate, and control child processes. Actually, there are three distinct operations involved: creating a new child process, causing the new process to execute a program, and coordinating the completion of the child process with the original program.

Prelab Activities

- ✓ Read the manual and try to do the experiment yourself before the lab.

General Information

A Process:

A ***process*** is basically ***a single running program***. It may be a ***``system''*** program (e.g login, update, csh) or ***program initiated by the user*** (pico, a.out or a user written one).

When UNIX runs a process it gives each process a unique number - ***a process ID, pid.***

The UNIX command ***ps*** will list all current processes running on machine and will list the ***pid***.

The C function ***int getpid( )*** will return the ***pid*** of process that called this function.
Processes are the primitive units for allocation of system resources. Each process has its own ***address space*** and (usually) one thread of control. ***A process executes a program; you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.***

Processes are organized hierarchically. Each process has a ***parent*** process which explicitly arranged to create it. The processes created by a given parent are called its ***child*** processes.
A child inherits many of its attributes from the parent process.

Every process in a UNIX system has the following attributes:

- some code
- some data
- a stack
- a unique process id number (PID)

---

[1] **Ref:** M. Akbar Badhusha, King Fahd University of Petroleum and Minerals, Saudi Arabia.

When UNIX is first started, there's only one visible process in the system. This process is called ***"init"***. The only way to create a new process in UNIX is to duplicate an existing process, so ***"init"*** is the ancestor of all subsequent processes. When a process duplicates, the parent and child processes are identical in every way except their ***PID***s; the child's code, data, and stack are a copy of the parent's, and they even continue to execute the same code. ***A child process may, however, replace its code with that of another executable file, thereby differentiating itself from its parent.*** For example, when ***"init"*** starts executing, it quickly duplicates several times. Each of the duplicate child processes then replaces its code from the executable file called ***"getty"*** which is responsible for handling user logins.

When a child process terminates, its death is communicated to its parent so that the parent may take some appropriate action.

A process that is waiting for its parent to accept its return code is called a zombie process.
If a parent dies before its child, the child (orphan process) is automatically adopted by the original "init" process.

Its very common for a parent process to suspend until one of its children terminates. For example, when a shell executes a utility in the foreground, it duplicates into two shell processes; the child shell process replaces its code with that of utility, whereas the parent shell waits for the child process to terminate. When the child process terminates, the original parent process awakens and presents the user with the next shell prompt.

Here's an illustration of the way that a shell executes a utility:

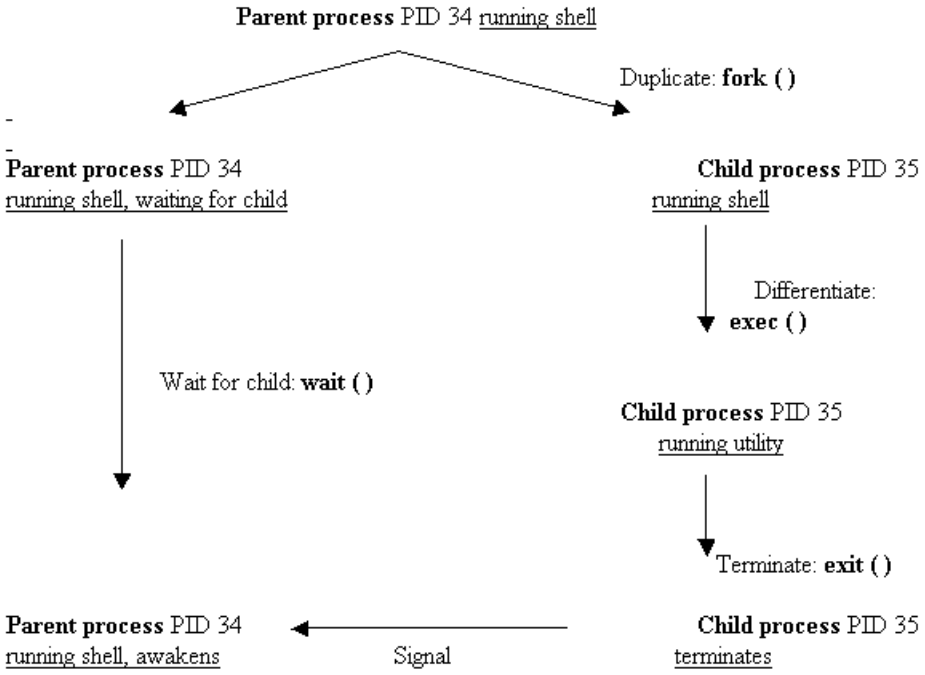

Figure 1

A program usually runs as a single process. However later we will see how we can make programs run as several separate communicating processes.

Running UNIX commands from C :
We can run commands from a C program just as if they were from the UNIX command line by using the ***system( )*** function.

int system(char *string) -- where ***string*** can be the name of a UNIX utility, an executable shell script or a user program. System returns the exit status
of the shell. System is prototyped in ***<stdlib.h>***

Example: Call ***ls*** from a program

```
#include <stdlib.h>
#include <stdio.h>
void main(){
        printf("Files in Directory are:\n");
        system("ls -l");
}
```

Program 1.

system is a call that is made up of 3 other system calls: ***execl( ), wait( ) and fork( )*** (which are prototyed in <unistd>)

Command Line Arguments:
In the UNIX environment, arguments can be passed to the ***main*** function of a C program much like parameters can be passed to other functions in C. The arguments are passed as ***strings*** to the ***main*** function. The syntax for the main function prototype in this case is as follows:

```
int  main (argc, argv)
int argc;
char *argv[ ];
```

where ***argc*** is *the number of command line arguments, including the command name*, and ***argv[i]*** is a *pointer to the i^{th} argument* which is represented as a character string.

Example:
For a program named ***prog1*** which is invoked with the command line:
`prog1 add 12 5`
the value of ***argc*** is ***4***, the value of ***argv[0]*** is ***"prog1"***, the value of ***argv[1]*** is ***"add",*** the value of ***argv[2]*** is ***"12",*** and the value of ***argv[3]*** is ***"5".***
Note that ***all arguments are represented as character strings***. If numbers are to be passed into main, they must be converted from strings inside ***main***.

Process Creation Concepts:
This section gives an overview of processes and of the steps involved in creating a process and making it run another program.

Each process is named ***by a process ID number***. A unique process ID is allocated to each process when it is created. ***The lifetime of a process ends when its termination is reported to its parent process***; at that time, all of the process resources, including its process ID, are freed.

Processes are created with the ***fork system call*** (so the operation of creating a new process is sometimes called forking a process). *The child process created by fork is a* ***copy of the original parent process, except that it has its own process ID.***
After forking a child process, both the parent and child processes continue to execute normally.

If you want your program to wait for a child process to finish executing before continuing, you must do this explicitly after the fork operation, by calling ***wait*** or ***waitpid***. These functions give you limited information about why the child terminated--for example, its exit status code.

A newly forked child process continues to execute the same program as its parent process, at the point where the fork call returns. You can use the return value from ***fork*** to tell whether the program is running in the ***parent*** process or the ***child***.

Having several processes run the same program is only occasionally useful. ***But the child can execute another program using one of the exec functions.*** The program that the process is executing is called its process image. Starting execution of a new program causes the process to forget all about its previous process image; when the new program exits, the process exits too, instead of returning to the previous process image.

Process Identification:
The ***pid_t*** data type represents process IDs. You can get the process ID of a process by calling ***getpid***. The function ***getppid*** returns the process ID of the parent of the current process (this is also known as the parent process ID). Your program should include the header files `unistd.h' and `sys/types.h' to use these functions.

```
Data Type: pid_t
```

The ***pid_t*** data type is a signed integer type which is capable of representing a process ID. In the GNU library, this is an int.

```
Function: pid_t getpid (void)
```

The ***getpid*** function returns the process ID of the current process.

```
Function: pid_t getppid (void)
```

The ***getppid*** function returns the process ID of the parent of the current process.

Creating Multiple Processes:

A special type of process important in the UNIX environment is the ***daemon***.

A ***daemon*** is a process that waits for a user to request some action, and then performs the request. As an example, there is an ftp daemon that waits until a user requests that a file be transferred and then performs the request. This implies that the ***daemon*** must always be ready to receive requests at the same time it is transferring files. To solve this problem, the ***daemon*** can create an identical process (***a child process***) to handle the file transfer while the original process (***the parent process***) continues to wait for requests. The fork system call is used by a process to spawn a process identical to it.

The fork function is the primitive for creating a process. It is declared in the header file `unistd.h'.

```
Function: pid_t fork (void)
```

The fork function creates a new process.

If the operation is successful, there are then both parent and child processes and both see fork return, but with different values: **it returns a value of 0 in the child process and returns the child's process ID in the parent process.**

If process creation failed, fork returns a value of -1 in the parent process and ***no child is created.***

```
#include <stdio.h>
#include <unistd.h>  /* contains fork prototype */

int main(void) {
        printf("Hello World!\n");
        fork();
        printf("I am after forking.\n");
        printf("\tI am process %d.\n\tMy parent is %d.\n",
getpid(), getppid());
}
```

Program 2.

```
Sample output:
Hello World!
I am after forking.
     I am process 2876.
     My parent is 2376.
I am after forking.
     I am process 2877.
     My parent is 2876.
```

Each process prints a message identifying itself.

```
#include <stdio.h>
#include <unistd.h>/* contains fork prototype */
int main(void)
{
        int pid;
        printf("Hello World!\n");
        printf("I am the parent process and pid is: %d \n", getpid());

        printf("Here i am before forking.\n");
        pid = fork();
        printf("Here I am after  forking.\n");
        if (0 == pid)
            printf("I am the child  process and pid is: %d\n", getpid());
        else
            printf("I am the parent process and pid is: %d\n", getpid());
}
```

Program 3.

```
Sample output:
Hello World!
I am the parent process and pid is: 2998
Here i am before forking.
Here I am after  forking.
I am the parent process and pid is: 2998
Here I am after  forking.
I am the child  process and pid is: 2999
```

Multiple forks:

```
#include <stdio.h>
#include <unistd.h>  /* contains fork prototype */
int main(void){
        printf("Here I am just before 1st forking statement\n");
        fork();
        printf("Here I am just after  1st forking statement\n");
        fork();
        printf("Here I am just after  2nd forking statement\n");
        fork();
        printf("Here I am just after  3rd forking statement\n");
        printf("\tHello World from process %d!\n", getpid());
}
```

Program 4.

Sample output:

```
Here I am just before 1st forking statement
Here I am just after  1st forking statement
Here I am just after  2nd forking statement
Here I am just after  1st forking statement
Here I am just after  2nd forking statement
Here I am just after  3rd forking statement
     Hello World from process 3192!
Here I am just after  2nd forking statement
Here I am just after  3rd forking statement
Here I am just after  3rd forking statement
     Hello World from process 3195!
Here I am just after  3rd forking statement
     Hello World from process 3194!
     Hello World from process 3193!
Here I am just after  3rd forking statement
     Hello World from process 3197!
Here I am just after  3rd forking statement
     Hello World from process 3198!
Here I am just after  2nd forking statement
Here I am just after  3rd forking statement
     Hello World from process 3196!
Here I am just after  3rd forking statement
     Hello World from process 3199!
```

Process Completion:
The functions described in this section are used to ***wait*** for a child process to terminate or stop, and determine its status. These functions are declared in the header file `sys/wait.h'.

```
Function: pid_t   wait (int  *status-ptr)
```

wait( ) will force a parent process to wait for a child process to stop or terminate. ***wait ( )*** return the ***pid*** of the ***child*** or -1 for an error. The exit status of the child is returned to ***status_ptr***.

```
Function: void exit (int status)
```

exit ( ) terminates the process which calls this function and returns the exit status value. Both UNIX and C (forked) programs can read the status.

By convention, a status of 0 means normal termination any other value indicates an error or unusual occurrence. Many standard library calls have errors defined in the sys/stat.h header file. We can easily derive our own conventions.
If the child process must be guaranteed to execute before the parent continues, the wait system call is used. A call to this function causes the parent process to wait until one of its child processes exits. The wait call returns the process id of the child process, which gives the parent the ability to wait for a particular child process to finish.

Guarantees the child process will print message before the parent process.

```
#include <stdio.h>
#include <unistd.h>   /* contains fork prototype */
#include <sys/wait.h> /* contains prototype for wait */
int main(void){
        int pid, status;
        pid = fork();
        if (-1 == pid){ /* check for error in fork */
                perror("bad fork");
                exit(1);
        }
        if (0 == pid)
                printf("\tI am the child process.\n");
        else{
                wait(&status); /*parent waits for child to finish */
                printf("I am the parent process.\n");
        }
}
```

Program 5.

```
Sample Output:
     I am the child process.
I am the parent process.
```

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

void main(){
    int forkresult ;
    printf ("%d:  I am the parent. Remember my number!\n", getpid()) ;
    printf ("%d:  I am now going to fork ... \n", getpid()) ;

    forkresult = fork () ;
    if (0 != forkresult){    /* the parent will execute this code */
        printf ("%d:  My child's pid is %d\n", getpid(), forkresult);
    }
    else{   /* the child will execute this code */
        printf ("%d:  Hi!  I am the child.\n", getpid()) ;
    }
    printf ("%d:  like father like son. \n", getpid()) ;
}
```

Program 6.

Sample Output:

```
3584:  I am the parent. Remember my number!
3584:  I am now going to fork ...
3584:  My child's pid is 3585
3584:  like father like son.
3585:  Hi!  I am the child.
3585:  like father like son.
```

sleep: A process may suspend for a period of time using the ***sleep*** command

```
Function: unsigned int  sleep (seconds)
```

Orphan processes:

When a parent dies before its child, the child is automatically adopted by the original "init" process. To illustrate this, insert a sleep statement into the child's code. This ensured that the parent process terminated before its child.

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

void main()
{
      printf ("I'am the original process with PID %d and PPID %d.\n",
getpid(), getppid ()) ;

    int pid = fork();   /*Duplicate. Child and parent continue from here*/
    if (0 != pid){      /*pid is non-zero, so I must be the parent*/
        printf("I'am the parent process with PID %d and PPID %d.\n",
getpid(), getppid()) ;
        printf("My child's PID is %d\n", pid);
    }
    else{               /*pid is zero, so I must be the child*/
        sleep (4) ;     /*make sure that the parent terminates first*/
        printf ("I'am the child process with PID %d and PPID
%d.\n",getpid(), getppid ()) ;
      }
      printf ("PID %d terminates.\n", getpid()) ;
}
```

Program 7.

The output is:

```
I'am the original process with PID 3714 and PPID 2376.
I'am the parent process with PID 3714 and PPID 2376.
My child's PID is 3715
PID 3714 terminates. /* Parent dies */
I'am the child process with PID 3715 and PPID 8080.
/*Orphaned, whose parent process is "init"*/
PID 3715 terminates.
```

Zombie processes:

A process that terminates cannot leave the system until its parent accepts its return code. If its parent process is already dead, it'll already have been adopted by the "init" process, which always accepts its children's return codes. However, ***if a process's parent is alive but never executes a*** wait( ), ***the process's return code will never be accepted and the process will remain a*** zombie.

The following program created a zombie process, which was indicated in the output from the ps utility. When the parent process is killed, the child was adopted by "init" and allowed to rest in peace.

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

void main(){
     int pid=fork(); /*Child and parent continue from here*/
     if (0 != pid){  /*pid is non-zero, so I must be the parent*/
          while(1)   /*Never terminate and never execute a wait()*/
               sleep (100);/*stop executing for 100 seconds*/
     }
     else{           /*pid is zero, so I must be the child*/
          exit (42); /*exit with any number*/
     }
}
```

Program 8.

The output is:

```
$ a.out& execute the program in the background
[1] 5186

$ ps -u obtain process status (sth. similar to following)
  PID TT STAT   TIME COMMAND
 5187 p0 Z      0:00 <exiting> the zombie child process
 5149 p0 S      0:01 -csh (csh)       the shell
 5186 p0 S      0:00 a.out            the parent process
 5188 p0 R      0:00 ps

$ kill 5186      kill the parent process
[1]    Terminated           a.out

$ ps -u notice that the zombie is gone now
  PID TT STAT   TIME COMMAND
 5149 p0 S      0:01 -csh (csh)
 5189 p0 R      0:00 ps
```

Executing a file:
This section describes the ***exec*** family of functions, for executing a file as a process image. You can use these functions to make a child process execute a new program after it has been forked.

The functions in this family differ in how you specify the arguments, but otherwise they all do the same thing. They are declared in the header file `unistd.h'.

```
Function: int execv(const char *filename,char *const argv[ ] )
```

The ***execv*** function executes the file named by filename as a new process image.

The ***argv*** argument is an array of null-terminated strings that is used to provide a value for the argv argument to the main function of the program to be executed. The last element of this array must be a null pointer. By convention, the first element of this array is the file name of the program sans directory names.

The environment for the new process image is taken from the environ variable of the current process image.

```
Function: int execl(const char *filename,const char *arg0, ...)
```

This is similar to ***execv***, but the ***argv*** strings are specified individually instead of as an array. A null pointer must be passed as the last such argument.

```
Function: int   execvp (const char *filename,  char *const argv[ ] )
```

The ***execvp*** function is similar to execv, except that it searches the directories listed in the PATH environment variable to find the full file name of a file from filename if filename does not contain a slash.

This function is useful for executing system utility programs, because it looks for them in the places that the user has chosen. Shells use it to run the commands that users type.

```
Function: int   execlp (const char *filename, const char *arg0, ...)
```

This function is like execl, except that it performs the same file name searching as the execvp function.

These functions normally don't return, since execution of a new program causes the currently executing program to go away completely. A value of -1 is returned in the event of a failure.

If execution of the new file succeeds, it updates the access time field of the file as if the file had been read.

Executing a new process image completely changes the contents of memory, copying only the argument and environment strings to new locations. But many other attributes of the process are unchanged:

The process ID and the parent process ID.
Session and process group membership.
Real user ID and group ID, and supplementary group IDs.
Current working directory and root directory. In the GNU system, the root directory is not copied when executing a setuid program; instead the system
default root directory is used for the new program.
File mode creation mask.
Process signal mask.
Pending signals.
Elapsed processor time associated with the process; see section Processor Time.

The following programs execs the commands "ls -l -a" and "echo hello there" using the 4 most-used forms of exec. Enter each, compile, and run.

Using execl( ) : The version will not search the path, so the full name of the executable file must be given. Parameters to main() are listed as arguments to execl().

```
#include <stdio.h>
#include <unistd.h>
void main(){
        execl ("/bin/ls",       /* program to run - give full path */
                "ls",           /* name of program sent to argv[0] */
                "-l",           /* first parameter (argv[1])*/
                "-a",           /* second parameter (argv[2]) */
                NULL) ;         /* terminate arg list */

        printf ("EXEC Failed\n") ;
 /*This above line will be printed only on error and not otherwise*/
}
```

Program 9.

Using execlp( ) : The version searches the PATH, so the full name of the executable file need not be given (if it is on the path). Parameters to main() are listed as arguments to execl()

```
#include <stdio.h>
#include <unistd.h>
void main(){
        execlp ("ls",           /* program to run - PATH Searched */
                "ls",           /* name of program sent to argv[0] */
                "-l",           /* first parameter (argv[1])*/
                "-a",           /* second parameter (argv[2]) */
                NULL) ;         /* terminate arg list */

        printf ("EXEC Failed\n") ;
}
```

Program 10.

Using execv(): The version will not search the path, so the full name of the executable file must be given. Parameters to main() are passed in a single array of character pointers.

```
#include <stdio.h>
#include <unistd.h>
void main (argc, argv )
int argc ;
char *argv[ ] ;
{
     execv ("/bin/echo",
                &argv[0]) ; /*program to load - full path only*/

     printf ("EXEC Failed\n") ;
}
```

Program 11. Run this program like this: **./a.out dummytext**

Using execvp( ) : The version searches the path, so the full name of the executable need not be given. Parameters to main() are passed in a single array of character pointers. This is the form used inside a shell!

```
#include <stdio.h>
#include <unistd.h>
void main (argc, argv )
int argc ;
char *argv[] ;
{
        execvp ("echo",        /*program to load - PATH searched*/
                &argv[0] ) ;

        printf ("EXEC Failed\n") ;
}
```

Program 12. Run this program like this: **./a.out dummytext**

Write a program where a child is created to execute a command.

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

void main ( ){
        printf ("%d:  I am the parent. Remember my number!\n", getpid());
        printf ("%d:  I am now going to fork ... \n", getpid());

        int pid = fork ();
        if (0 != pid){    /* the parent will execute this section */
                printf("%d:  My child's pid is %d\n", getpid(), pid);
        }
        else{     /* the child will execute this section */
                printf ("%d:  Hi!  I am the child.\n", getpid());
                printf ("%d:  I'm now going to exec ls!\n\n\n", getpid());
                execlp ("ls", "ls", NULL);
                printf("%d:  AAAAH ! ! My EXEC failed !!! !\n", getpid());
                exit (1) ;
        }
        printf ("%d:  like father like son. \n", getpid());
}
```

Program 13.

Run this program several times. You should be able to get different ordering of the output lines (sometimes the parent finished before the child, or vice versa). This means that after the fork, the two processes are no longer synchronized.

Process Completion Status: If the exit status value of the child process is zero, then the status value reported by wait is also zero. You can test for other kinds of information encoded in the returned status value using the following macros. These macros are defined in the header file `sys/wait.h'.

Macro: ***int WIFEXITED (int status)***
This macro returns a nonzero value if the child process terminated normally with exit.

Macro***: int WEXITSTATUS (int status)***
If WIFEXITED is true of status, this macro returns the low-order 8 bits of the exit status value from the child process.

Macro: ***int WIFSIGNALED (int status)***
This macro returns a nonzero value if the child process terminated because it received a signal that was not handled.

Macro: ***int WTERMSIG (int status)***
If WIFSIGNALED is true of status, this macro returns the signal number of the signal that terminated the child process.

Macro: ***int WCOREDUMP (int status)***
This macro returns a nonzero value if the child process terminated and produced a core dump.

Macro: ***int WIFSTOPPED (int status)***
This macro returns a nonzero value if the child process is stopped.

Macro: ***int WSTOPSIG (int status)***
If WIFSTOPPED is true of status, this macro returns the signal number of the signal that caused the child process to stop.

Here's a program which forks. The parent waits for the child. The child asks the user to type in a number from 0 to 255 then exits, returning that number as status.

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

void main()
{
    int number=0, statval;
    printf ("%d: I'm the parent !\n", getpid());
    printf ("%d: number = %d\n", getpid(), number);
    printf ("%d: forking ! \n", getpid());

    if (0 == fork())
    {
            printf ("%d: I'm the child !\n", getpid());
            printf ("%d: number = %d\n", getpid(), number);
            printf ("%d: Enter a number :  ", getpid());
            scanf ("%d", &number) ;
            printf ("%d: number = %d\n", getpid(), number);
            printf ("%d: exiting with value %d\n", getpid() , number);
            exit(number);
    }

    printf ("%d: number = %d\n", getpid(), number);
    printf ("%d: waiting for my kid !\n", getpid());
    wait (&statval) ;

    if(WIFEXITED(statval))
            printf ("%d: my kid exited with status %d\n",
                        getpid(), WEXITSTATUS(statval));
    else
            printf ("%d: My kid was killed off ! ! \n", getpid());
}
```

Program 14.

Here's an example call to wait(): a program spawns two children, then waits for their completion and behaves differently according to which one is finished. Try to compile and execute it.

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

void main(){
      pid_t  whichone, first, second;
      int  howmany, status;
      if (0 == (first = fork()) ){      /* Parent spawns 1st child */
            printf("Hi, I am the first child, and my ID is %d\n", getpid());
            sleep(10);
            exit(0);
      }
      else if (first == -1){
            perror ("1st fork: something went wrong\n");
            exit (1);
      }
      else if (0 == (second = fork())){   /* Parent spawns 2nd child */
            printf("Hi, I am the second child, and my ID is
%d\n",getpid());
            sleep(15);
            exit(0);
      }
      else if (second == -1){
            perror("2nd fork: something went wrong\n") ;
            exit(1) ;
      }
      printf("This is parent\n") ;

      howmany = 0;
      while (howmany < 2){                  /* Wait Twice */
            whichone = wait(&status);
            howmany++;
            if (whichone == first)
                  printf("First child exited\n");
            else
                  printf("Second child exited\n");

            if ((status & 0xffff) == 0 )
                  printf("correctly\n");
            else
                  printf("uncorrectly\n");
   }
}
```

Program 15.