Operating Systems

LAB 5

Process Creation and Execution

Objective:
This lab describes how a program can create, terminate, and control child processes. Actually, there are a few distinct operations involved: **creating a new child process**, and **coordinating the completion of the child process with the original program**.

****

****

What is 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.exe 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 your 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)**

When UNIX is first started, there's only one visible process in the system. This process is called "**init**", and its **PID** is **1**. 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

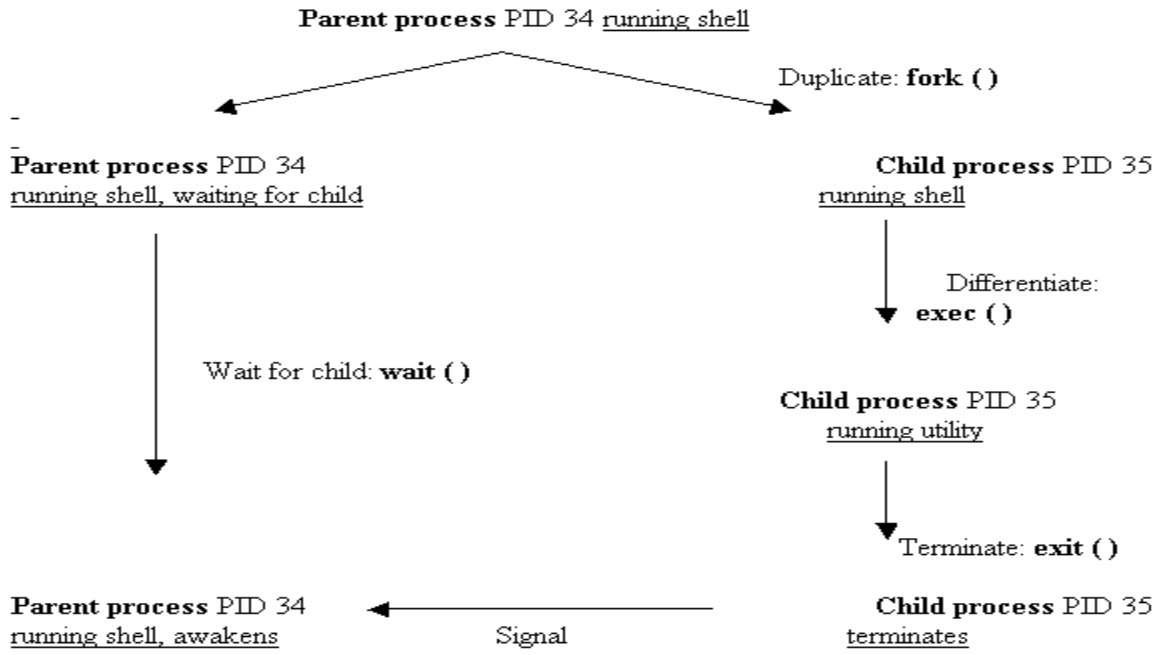
a process duplicates, the parent and child processes are identical in every way except their **PIDs**; the child's code, data, and stack are a copy of the parent's, and they even continue to execute the same code. **A child process may, however, replace its code with that of another executable file, 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 whose PID is 1.

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:



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

File Lab5_0.c :

```
main( )
{    printf(``Files in Directory are:n'');
     system(``ls -l'');
}
```

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

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**. This

function gives 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**.

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**.

The specific attributes of the child process that differ from the parent process are:

The child process has its own unique process ID.
The parent process ID of the child process is the process ID of its parent process. The child process gets its own copies of the parent process's open file descriptors.
Subsequently changing attributes of the file descriptors in the parent process won't affect the file descriptors in the child, and vice versa. However, the file position associated with each descriptor is shared by both processes. The elapsed processor times for the child process are set to zero.
The child doesn't inherit file locks set by the parent process.
The child doesn't inherit alarms set by the parent process.
The set of pending signals for the child process is cleared.

Example Lab5_1.c :

```
 tiger> gedit Lab5_1.c
#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", getpid( ));
}
```

Sample output :

```
Hello World!
I am after forking
        I am process 23848.
I am after forking
        I am process 23847.
```

When this program is executed, it first prints Hello World! . When the fork is executed, an identical process called the child is created. Then both the parent and the child process begin execution at the next statement. Note the following:

When a fork is executed, everything in the parent process is copied to the child process. This includes variable values, code, and file descriptors.
Following the fork, the child and parent processes are completely

independent.
There is no guarantee which process will print I am a process first.
The child process begins execution at the statement immediately after the fork, not at the beginning of the program.
A parent process can be distinguished from the child process by examining the return value of the fork call. Fork returns a zero to the child process and the process id of the child process to the parent.
A process can execute as many forks as desired. However, be wary of infinite loops of forks (there is a maximum number of processes allowed for a single user).

Example Lab5_2.c :

Each process prints a message identifying itself.

```
 tiger> gedit Lab5_2.c
#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 use of forking\n");
   pid = fork( );
   printf("Here I am just after forking\n");
   if (pid == 0)
      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());
}
```

Sample Output :

```
Hello World!
I am the parent process and pid is : 23951 .
Here i am before use of forking
Here I am just after forking
I am the child process and pid is :23952.
Here I am just after forking
I am the parent process and pid is: 23951 .
```

Example Lab5_3.c :

Multiple forks:

```
   tiger> gedit Lab5_3.c
#include <stdio.h>
#include <unistd.h>  /* contains fork prototype */
int main(void)
```

```
{
   printf("Here I am just before first forking statement\n");
   fork( );
   printf("Here I am just after first forking statement\n");
   fork( );
   printf("Here I am just after second forking statement\n");
   fork( );
   printf("Here I am just after third  forking statement\n");
   printf("    Hello World from process %d!\n", getpid( ));
}

Sample Output :

Here I am just before first forking statement
Here I am just after first forking statement
Here I am just after second forking statement
Here I am just after third  forking statement
   Hello World from process 24120!
Here I am just after first forking statement
Here I am just after second forking statement
Here I am just after third  forking statement
    Hello World from process 24119!
Here I am just after second forking statement
Here I am just after third  forking statement
    Hello World from process 24122!
Here I am just after third  forking statement
    Hello World from process 24123!
Here I am just after second forking statement
Here I am just after third  forking statement
    Hello World from process 24118!
Here I am just after third  forking statement
    Hello World from process 24121!
Here I am just after third  forking statement
    Hello World from process 24124!
Here I am just after third  forking statement
    Hello World from process 24117!
```

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 value.

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.

sleep

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

Function: unsigned int sleep (seconds)

Example Lab5_4.c :

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

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

    }

     printf ("%d:  like father like son. \n", getpid ( ) ) ;
}
```

Sample Output :

```
28715:  I am the parent. Remember my number!
28715:  I am now going to fork ...
28716:  Hi!  I am the child.
28716:  like father like son.
28715:  My child's pid is 28716
28715:  like father like son.
```

Orphan processes :

When a **parent dies before its child**, the child is automatically adopted by the original "**init**" process whose **PID** is **1**. To, illustrate this insert a **sleep** statement into the child's code. This ensured that the parent process terminated before its child.

Example Lab5_5.c :

```
#include <stdio.h>
main ( )
{
   int   pid ;

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

   pid = fork ( ) ;       /* Duplicate. Child and parent continue from here */
   if ( pid != 0 )        /* 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 ( ) ) ;
}
```

Sample Output:

```
I'am the original process with PID 5100 and PPID 5011.
I'am the parent process with PID 5100 and PPID 5011.
My child's PID is 5101
PID 5100 terminates.    /* Parent dies */
I'am the child process with PID 5101 and PPID 1.
            /* Orphaned, whose parent process is "init" with pid 1 */
PID 5101 terminates.
```

Important points to note:

1. **The Shell acts as the parent process**. All the processes started by the user are treated as the children of **shell**.

2. The **status of a UNIX process** is shown as the **second column** of the process table when viewed by the execution of the **ps** command. Some of the states are:
R: ***running***, **O**: ***orphan***, **S**: ***sleeping***, **Z**: ***zombie***.

3. **The child process is given the time slice before the parent process**. This is quite logical. For example, we do not want the process started by us to wait until its parent, which is the UNIX shell finishes. This will explain the order in which the print statement is executed by the parent and the children.

TASKS:

Execute the C programs given in the following problems. ***Observe*** and **Interpret** the results. You will learn about ***child*** and ***parent*** processes, and much more about UNIX processes in general by performing the suggested experiments. UNIX Calls used in the following problems:
getpid( ), getppid( ), sleep( ) and ***fork( )***.

1) Run the following program twice. Both times as a background process, i.e., suffix it with an ampersand "**&**". Once both processes are running as background processes, view the *process table* using **ps -l** UNIX command. Observe the *process state*, *PID (process ID)* etc. Repeat this experiment to observe the changes, if any. Write your observation about the Process ID and state of the process.

```
main ( ) {
   printf ("Process ID is: %d\n",
getpid( ) ) ;
   printf ("Parent process ID is:
%d\n", getppid( ) ) ;
   sleep (60) ;
   printf ("I am awake. \n");
}
```

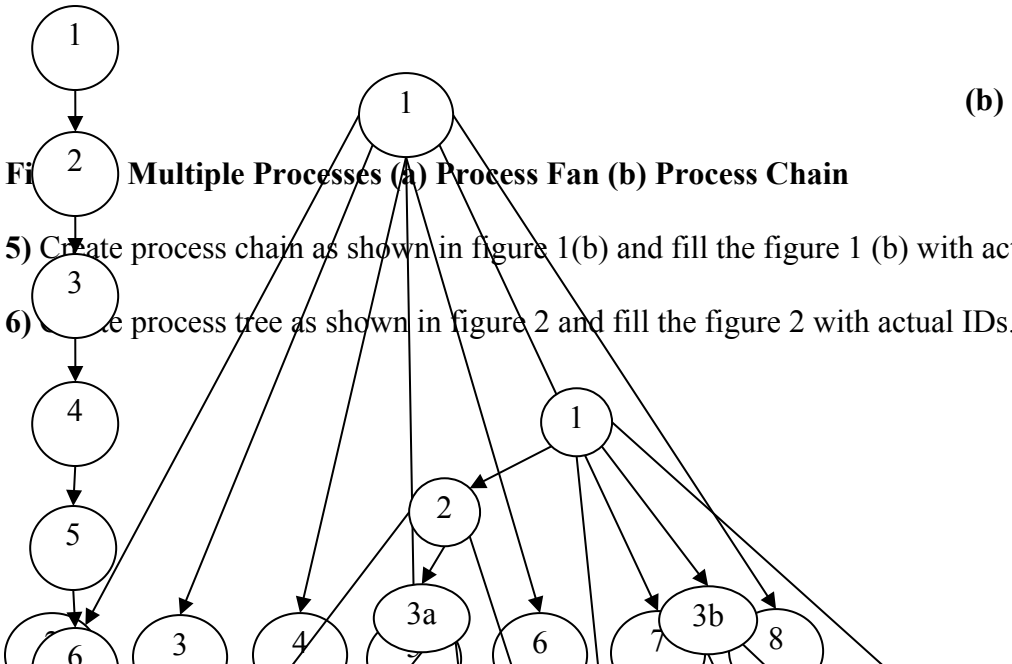
2) Run the following program and observe the *number of times* and the *order* in which the print statement is executed. The **fork( )** creates a child that is a duplicate of the parent process. The child process begins from the **fork( )**. All the statements after the call to **fork ( )** are executed by the parent process and also by the child process. Draw a family tree of processes and explain the results you observed.

```
main ( ) {
   fork ( ) ;
   fork ( ) ;
   printf ("Parent Process ID is
%d\n", getppid ( ) ) ;
}
```

3) Run the following program and observe the result of **time slicing** used by UNIX.

```
main ( )  {
   int i=0, j=0, pid, k, x ;
   pid = fork ( );
   if ( pid == 0 ) {
      for ( i = 0; i < 20; i++ ) {
         for (k = 0; k < 10000; k++ );
          printf ("Child: %d\n", i) ;
      }
   }
   else {
      for ( j = 0; j < 20; j++ ){
           for (x = 0; x < 10000; x++ );
           printf ("Parent: %d\n", j) ;
      }
   }
}
```

4) Create process fan as shown in figure 1 (a) and fill the figure 1 (a) with actual IDs.



(b)

Fi Multiple Processes (a) Process Fan (b) Process Chain

5) Create process chain as shown in figure 1(b) and fill the figure 1 (b) with actual IDs.

6) e process tree as shown in figure 2 and fill the figure 2 with actual IDs.