Programming Assignment # 1

Process Creation and Execution

Objective:

This lab describes how a program can 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.

Process Creation Concepts

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.

A process ID number names each process. 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 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 process.

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

Monitoring Processes

To monitor the state of your processes under Unix use the ps command.

```
ps [-option]
```

Used without options this produces a list of all the processes owned by you and associated with your terminal. The information displayed by the **ps** command varies according to which command option(s). if you use Cygwin Terminal Then the following column headings displayed on the screen.

```
Alaa@Alaa-PC /
$ ps
      PID    PPID    PGID    WINPID   TTY      UID    STIME COMMAND
     4652    4944    4652      1816   pty0    1000 03:11:44 /usr/bin/bash
     4944       1    4944      4944   ?       1000 03:11:44 /usr/bin/mintty
     5812    4652    5812      5456   pty0    1000 04:20:48 /usr/bin/ps
```

Where PID represents Process ID
PPID = Parent Process ID
PGID= Process Group ID
TTY= Controlling Terminal
UID = User ID
WINPID= Windows Process ID

As an Example:
To display information about all your processes
```
$ ps –u mohammed
```

Process Identification:

The **pid_t** data type represents process IDs which is basically a signed integer type (**int**). 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.

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

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.

Lab1.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( ));
}
```

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 careful of infinite loops of forks (there is a maximum number of processes allowed for a single user).

Lab2.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());
}
```
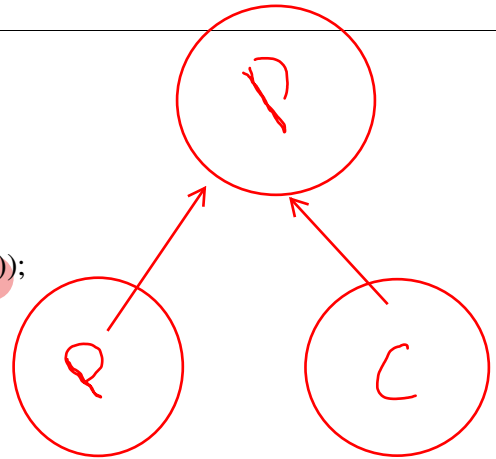
Lab3.c (Multiple forks):

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
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");
printf("\t\tHello World from process %d!\n", getpid());
}
```
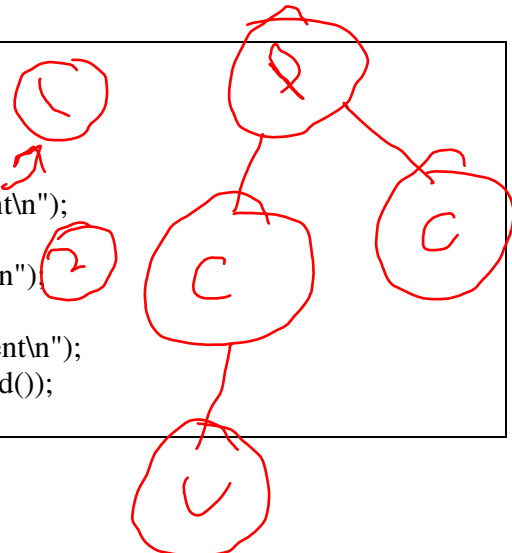
Process Completion

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 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.

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.

Sleep

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

Function: unsigned int sleep (seconds)

Examples

Lab4.c: Guarantees the child process will print its message before the parent process.

```
#include <stdio.h>
#include <sys/wait.h> /* contains prototype for wait */
int main(void)
{
int pid;
int status;
printf("Hello World!\n");
pid = fork( );
if (pid == -1) /* check for error in fork */
{
perror("bad fork");


}
```

```
exit(1);
}
if (pid == 0)
printf("I am the child process.\n");
else
{
wait(&status); /* parent waits for child to finish */
printf("I am the parent process.\n");
}
}
```

Lab5.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());
}
```

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
Lab6.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 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'm the child with PID %d and PPID %d.\n",
getpid(), getppid()) ;
}
printf ("PID %d terminates.\n", getpid()) ;
}
```

The output is:
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.