Lab 3 – PROCESSES

OBJECTIVE

Learn to create processes using fork()system call.

TIME REQUIRED	:	3 hrs
PROGRAMMING LANGUAGE	:	C/C++
SOFTWARE REQUIRED	:	Ubuntu/Fedora, gcc/gc, Text Editor, Terminal, Windows, Dev
HARDWARE REQUIRED	:	Core i5 in Computer Labs

PROCESS CREATION

A process is a program in execution. The process which creates another process is called parent process. The process which is created is called child process. We can identify process by their unique key called process identifier or pid (integer number). In Linux we can use fork() system call to create processes. By this system call new process is created containing copy of parent process. Both process (parent and child) continue executing instructions after fork(). The return number of fork() for new (child) process will be 0 (zero), whereas for parent process value will be nonzero positive process identifier. If fork() fails, it return a negative number. In this section, we create a simple program using fork() to create child process

TASK 3.1

Using getpid(): This function returns the pid of the current program.

Use the following code and write the output.

```
 # include <stdio.h>
# include <unistd.h>
 # include <stdlib.h>
 int main()
{
        int pid;
        pid = getpid();
        printf("Process ID is %d\n", pid);
        return 0;
    }
```

Now compile the code written. This can be accomplished by using the following appended command. `$ gcc code1.c –o code1`

Whereas –o is used to open the save file in the command. Then after –o, we write the filename.

After compilation, run the command.

```
$ ./code1
```

$ GCC –o code1 in this examp in this example, both the PID and PPID are used.le, both the PID and PPID are used.code1.cz

TASK 3.2

In this example, both the PID and PPID are used. Consider a file that contains two variables in the main program that are assigned by the process IDs. One is of the current process, and the other one is of the parent process. Then similar to the first example, print both the ids through their variables.

```
Int pid_t =getpid();
Int ppid_t =getppid();
```

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

 # include <stdlib.h>



int main(void)
{
//getting process id and storing in a variable
int pid-a=getpid();
//getting parent function process id and storing in a variable
int ppid_x=getppid(0);
//printing the ids of the processes
printf("process id is : %d\n" , pid_a);
printf("parent process id is : %d\n" ,ppid_x);
return 0;
}
```

Save the file with code1.c and again, compile the code and then run it. `./code1`

Paste screenshots here

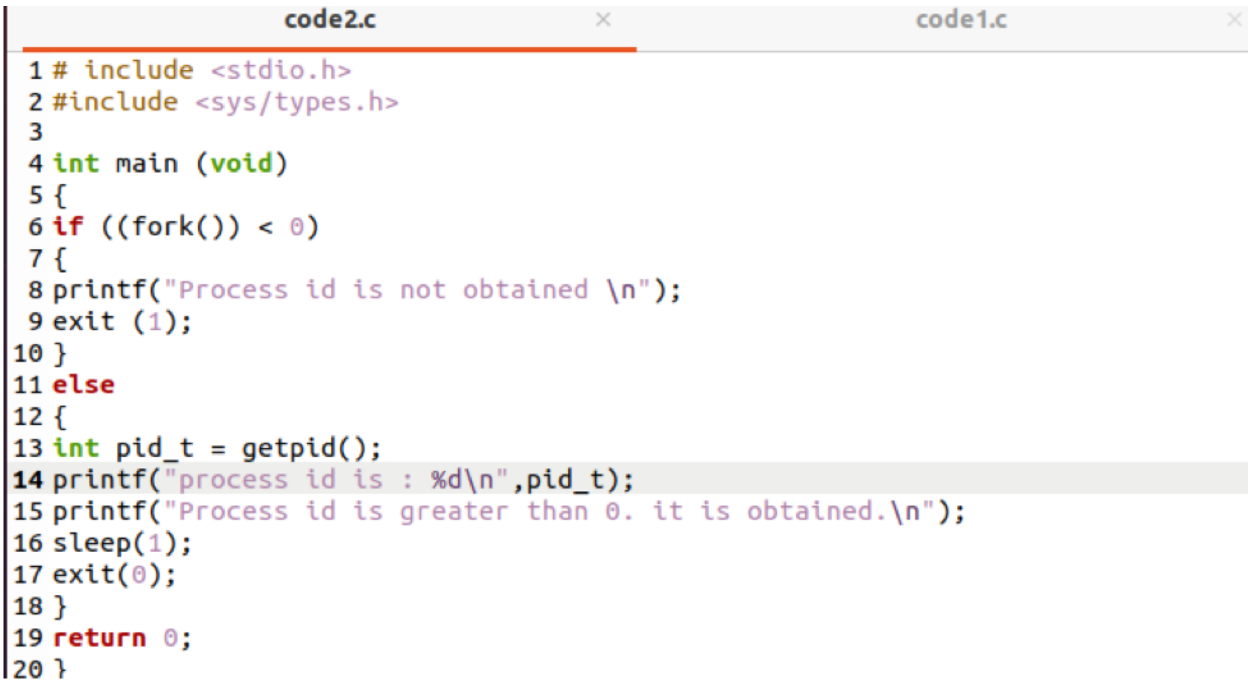
TASK 3.3 Using Fork() system calls

Consider a new file having two libraries in the respective header. Here a condition is used in which we have utilized the "if-else" statement. In the main program, it is stated that if the fork value is in –ive value, it will display a message that the process's id is failed and will not be obtained. If the situation is false, then the compiler will move to the else part of the condition. In this part, the process id is obtained, then we will display this process ID and display a message that the process ID is obtained. Here we will quote the if-else statement of the source code.

Now again, compile the code and then run it.

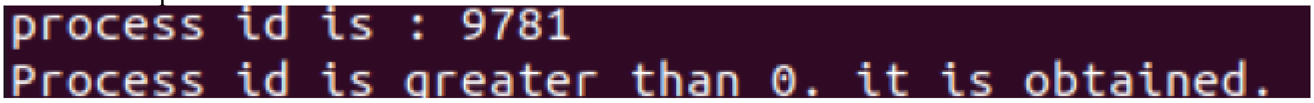
./code2

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



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

int main (void)
{
if ((fork()) < 0)
{
printf("Process id is not obtained \n");
exit (1);
}
else
{
int pid_t = getpid();
printf("process id is : %d\n",pid_t);
printf("Process id is greater than 0. it is obtained.\n");
sleep(1);
exit(0);
}
return 0;
}
```

Now again, compile the code and then run it.
./code2
This is nan output

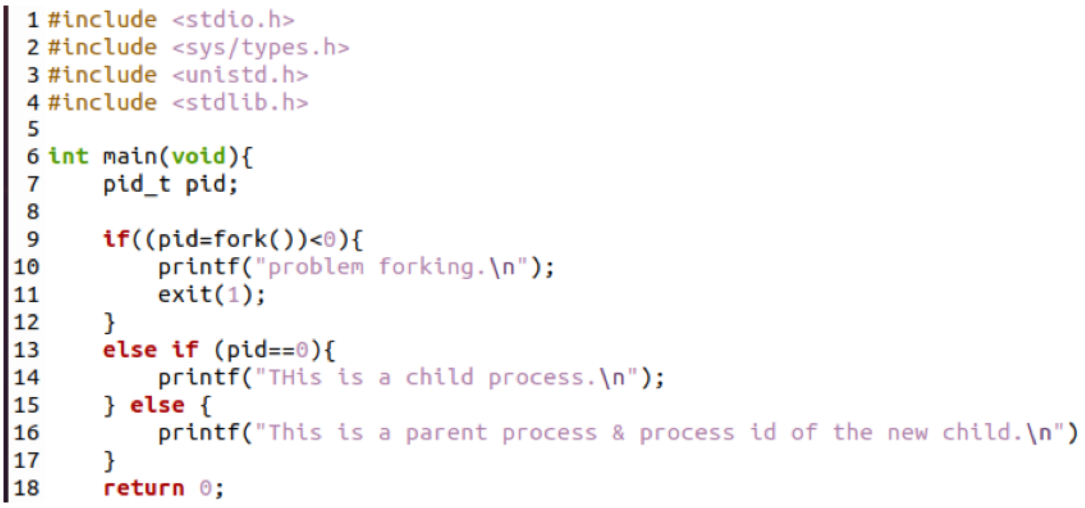


```
process id is : 9781
Process id is greater than 0. it is obtained.
```

The output shows that the else part has been executed and will print the process id and then display a PID message.

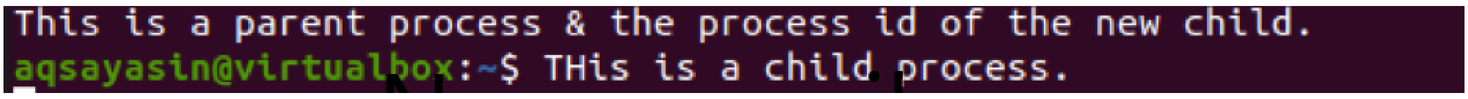
TASK 3.4

This is another example of explaining the same concept. Fork() function returns two different values. In the case of a child process, the value is 0, which is to be returned. At the same time, the value in the case of the parent process is the process ID of the new child.



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

int main(void){
    pid_t pid;

    if((pid=fork())<0){
        printf("problem forking.\n");
        exit(1);
    }
    else if (pid==0){
        printf("THis is a child process.\n");
    } else {
        printf("This is a parent process & process id of the new child.\n")
    }
    return 0;
```

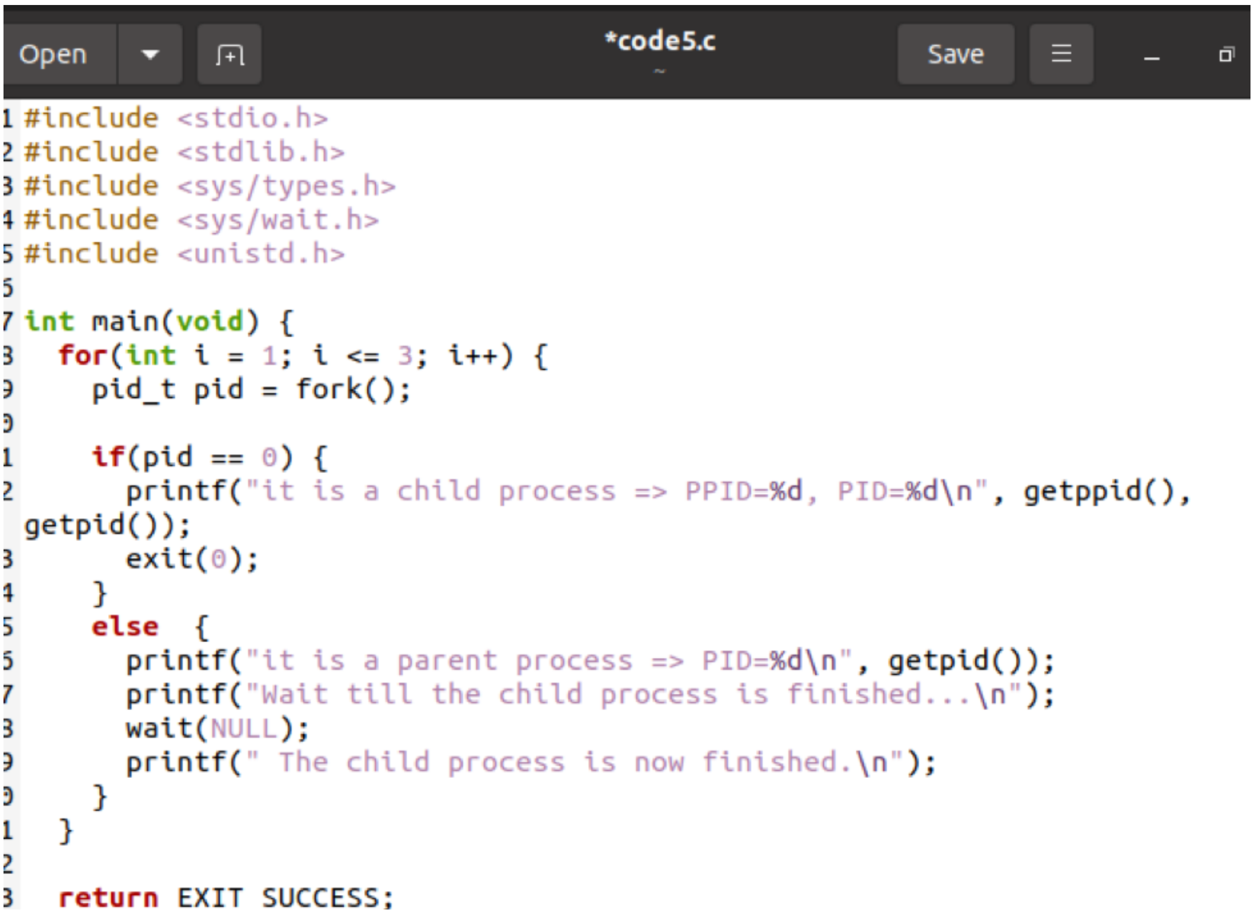
Save thid file with code3. c. Now compile and run the code.
$ gcc –o code3 code3.c
$./code3
This is output



```
This is a parent process & the process id of the new child.
aqsayasin@virtualbox:~$ THis is a child process.
```

Task 3.5

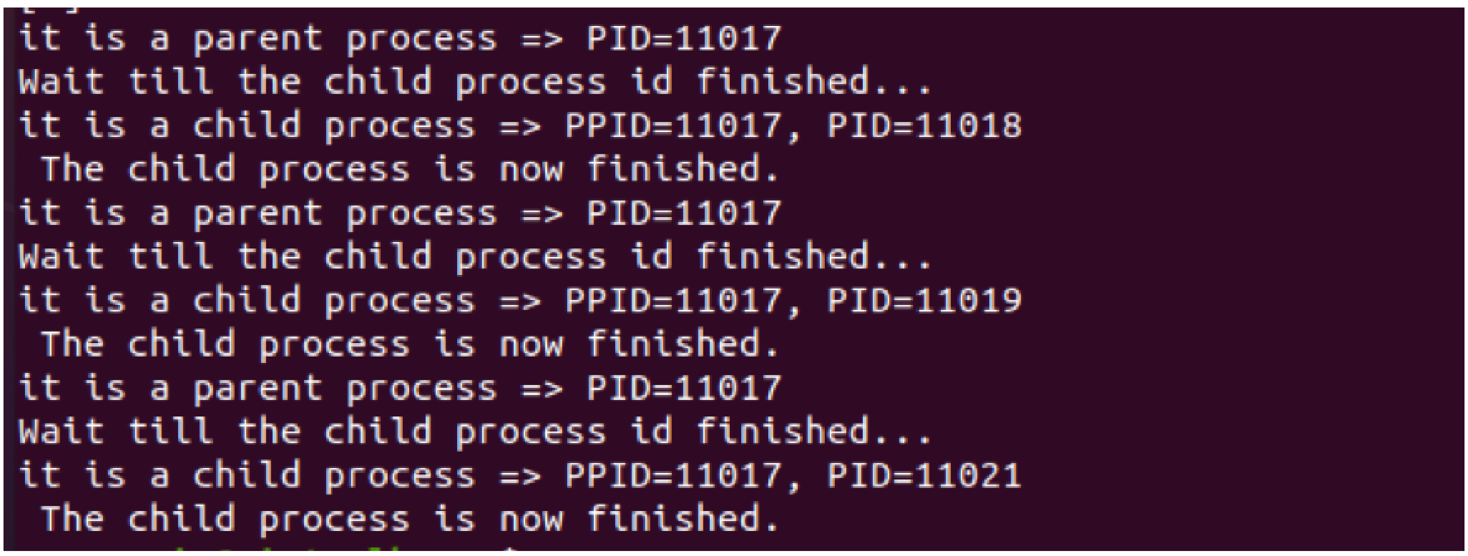
We can also use loops with the fork () functions to use the getpid() functions. We can use loops to create many child processes. Here we have to use the value of 3 in the loop.



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

int main(void) {
  for(int i = 1; i <= 3; i++) {
    pid_t pid = fork();

    if(pid == 0) {
      printf("it is a child process => PPID=%d, PID=%d\n", getppid(),
getpid());
      exit(0);
    }
    else  {
      printf("it is a parent process => PID=%d\n", getpid());
      printf("Wait till the child process is finished...\n");
      wait(NULL);
      printf(" The child process is now finished.\n");
    }
  }

  return EXIT SUCCESS;
```

Now save the file and execute it. There is another simple method to compile and execute the code only in a single command. That is.

```
$ gcc codes5.c –o s & ./code5
```



```
it is a parent process => PID=11017
Wait till the child process id finished...
it is a child process => PPID=11017, PID=11018
 The child process is now finished.
it is a parent process => PID=11017
Wait till the child process id finished...
it is a child process => PPID=11017, PID=11019
 The child process is now finished.
it is a parent process => PID=11017
Wait till the child process id finished...
it is a child process => PPID=11017, PID=11021
 The child process is now finished.
```

TASK 3.6

What is the outcome of the following program?

```
int main(){
        int pid;
        pid = fork();
        if(pid==0){
                printf("I am child, my process ID is %d\n", getpid())printf("The   parent
                process ID is                                    %d\n", getppid());
        }
        else{
                printf("I am parent, my process ID is             %d\n", getpid())printf("The
                parent process ID is                              %d\n", getppid());
        }
        return 0;
}
```

TASK 3.6

To see if the pid is same as shown in the system, Open System Monitor. Check to see if the pid is same. Use the following code

```
int main(){
        int pid,i; pid =
        fork();if(pid==0){
                printf("I am child, my process ID is                    %d\n", getpid());
                printf("The   parent   process ID is                  %d\n", getppid());
        }
        else{
                printf("I am parent, my process ID is             %d\n", getpid());printf("The
                parent process ID is                              %d\n", getppid());
        }
        scanf("%d",&i);             //so that program halts for user inputreturn 0;
}
```

Show screenshots here: -

TASK 3.7

What is the outcome of this program?

```
/**
 *   This program forks a separate process using the fork()/exec()system calls.
 *   * Figure 3.10*
 *   @author Gagne, Galvin, Silberschatz Operating System Concepts -Seventh Edition
 *   Copyright John Wiley & Sons - 2005. *#include
<stdio.h>
#include <unistd.h> #include
<sys/types.h>int main(){
pid_t pid;
        /* fork a child process */pid = fork();
        if (pid < 0) { /* error occurred */ fprintf(stderr, "Fork
                Failed\n");exit(-1);
        }
        else if (pid == 0) { /* child process */printf("I am the
                child %d\n",pid); execlp("/bin/ls","ls",NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */printf("I am the parent
                %d\n",pid);
                wait(NULL);

                printf("Child Complete\n");exit(0);
        }
```

TASK 3.8

Using Exec: The *fork* system call creates a new process but that process contains, and is executing, exactly the same code that the parent process has. More often than not, we'd like to run a new program.

Example of Execve(): The *execve* system call replaces the current process with a new program. Type the following command in Terminal and show the output.

```
ls -aF /
```

Now execute the following code

```
/* execve: run a program */
#include <stdlib.h> /* needed to define exit() */ #include <unistd.h>  /*
needed to define getpid() *#include <stdio.h>  /* needed for printf() */
int main(int argc, char **argv) {
            char *args[] = {"ls", "-aF", "/", 0};
                    /* each element represents a command line argument */char *env[] = { 0 };
                    /* leave the environment list null */
```

```
                printf("About to run /bin/ls\n");execve("/bin/ls",
                args, env);
                perror("execve"); /* if we get here, execve failed */exit(1);
        }
```

What is the outcome?

TASK 3.9

Using Execlp

execlp, which allows you to specify all the arguments as parameters to the function. Note that the first parameter is the command. The second parameter is the first argument in the argument list that is passed to the program (argv[0]). These are often the same but don't have to be. The last parameter must be a null pointer.

```
        /* execlp: run a program using execlp */
        #include <stdlib.h> /* needed to define exit() */ #include <unistd.h>  /*
        needed to define getpid() */#include <stdio.h>  /* needed for printf() */
        int main(int argc, char **argv) {printf("About to
                run ls\n");
                execlp("ls", "ls", "-aF", "/", (char*)0); perror("execlp");        /* if we get here,
                execlp failed */exit(1);
        }
```

What is the output?

TASK 3.10

Using fork() and exec(): The fork system call creates a new process. The execve system call overwrites a process with a new program. A process forks itself and the child process execs a new program, which overlays the one in the current process.

```
/* forkexec: create a new process. */
/*   The child runs "ls -aF /". The parent wakes up after 5 seconds
*/
#include <stdlib.h> /* needed to define exit() */ #include <unistd.h>  /*
needed for fork() and getpid() *#include <stdio.h>  /* needed for printf() */
int main(int argc, char **argv) {void
        runit(void);
        int pid; /* process ID */switch (pid =
        fork()) {
        case 0:                    /* a fork returns 0 to the child */runit();
                break;
        default: /* a fork returns a pid to the parent */sleep(5); /* sleep for 5
                seconds */ printf("I'm still here!\n");
                break;
        case -1: /* something went wrong */
                perror("fork");
                exit(1);
        }
        exit(0);
}
void runit(void) {
        printf("About to run ls\n");
        execlp("ls", "ls", "-aF", "/", (char*)0); perror("execlp");        /* if we get here,
        execlp failed */exit(1);
}
```

What is the outcome of the program?

EXERCISE 3.2

[2]

What do you understand from last code?

TASK 3.11

Creating a Process in Windows

```
        /**
          *   This program creates a separate process using the
        CreateProcess() system call. Figure 3.12
          *   @author Gagne, Galvin, Silberschatz Operating System Concepts -
        Seventh Edition
          *   Copyright John Wiley & Sons - 2005.
          */
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

int main( int argc, TCHAR *argv[] )
{
   STARTUPINFO si;
   PROCESS_INFORMATION pi;
```

```
ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);
ZeroMemory( &pi, sizeof(pi) );

if( argc != 2 )
{
    printf("Usage: %s [cmdline]\n", argv[0]);
   return 1;
}

// Start the child process.
if( !CreateProcess( NULL,   // No module name (use command line)
    argv[1],        // Command line
    NULL,           // Process handle not inheritable
    NULL,           // Thread handle not inheritable
    FALSE,          // Set handle inheritance to FALSE
    0,              // No creation flags
    NULL,           // Use parent's environment block
    NULL,           // Use parent's starting directory
    &si,            // Pointer to STARTUPINFO structure
    &pi )           // Pointer to PROCESS_INFORMATION structure
)
{
    printf( "CreateProcess failed (%d).\n", GetLastError() );
    return 1;
}

// Wait until child process exits.
WaitForSingleObject( pi.hProcess, INFINITE );

// Close process and thread handles.
CloseHandle( pi.hProcess );
```

```
    CloseHandle( pi.hThread );
}
```

What is the Outcome?

EXERCISE 3.3 **[6]**

Which OS gives you a better interface for creating and executing child programs and why?

RESOURCES

http://manpages.ubuntu.com/manpages/lucid/man2/fork.2.html

http://www.cs.rutgers.edu/~pxk/416/notes/c-tutorials/exec.html

http://www.cs.rutgers.edu/~pxk/416/notes/c-tutorials/forkexec.html

http://www.advancedlinuxprogramming.com/alp-folder/alpch03-processes.pdf