Topic to be covered

- Monitoring Processes
  - ps
  - pstree
- Process Identification:
  - getpid()
  - getppid()
- Process Creation
  - fork ()
- Process Completion
  - wait (int *)
  - exit (int)
- Orphan Process
- Zombie Process
- Process Binary Replacement
  - exec ()

Objectives

- Students are able to create new processes in linux.
- Students are able to load different programs binaries in current process
- Students are able to handle the termination of the process.

Prerequisite:

- Visual Studio Code
- GCC compiler
- Basic C Programing
- Use of man Page

Monitoring Processes

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

```
ps
```

This option lists 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) you use and the type of UNIX that you are using.

These are some of the column headings displayed by the different versions of this command.

```
PID SZ(size in Kb) TTY(controlling terminal) TIME(used by CPU) COMMAND
```

Exercise:

1. Display information about your processes that are currently running. Add Screenshot in a word file.

```
ps
```

2. Display tree structure of your processes. Add Screenshot in a word file.

```
pstree
```

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 file '**unistd.h**' and '**sys/types.h**' to use these functions.

pid_t getpid()

The `getpid()` function returns the process ID of the current process. **(man getpid)**

Pid_t getppid()

The `getppid()` function returns the process ID of the parent of the current process. **(man getppid)**

Process Creation:

The fork function creates a new process.

Duplicating a process image:

To use processes to perform more than one function at a time we can either use threads or create an entirely separate process from within program by calling fork. System call fork() is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

- If fork() returns a negative value, the creation of a child process was unsuccessful.
- fork() returns a zero to the newly created child process.
- fork() returns a positive value, the process ID of the child process, to the parent.
- The returned process ID is of type pid_t defined in sys/types.h and it is an integer value.

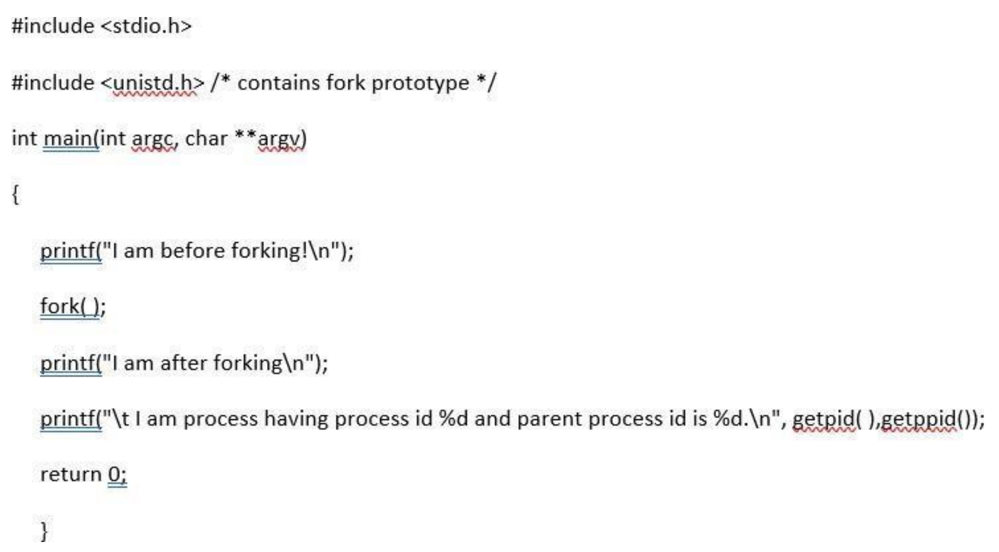
Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.

The fork function creates a new process.

pid_t fork()

- On Success
  - Return a value **0** in the child process
  - Return the **child's process ID** in the parent process.
- On Failure
  - Returns a value **-1** in the parent process and no child is created.

Example Task 1: (Add output in word file)



```
#include <stdio.h>

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

int main(int argc, char **argv)

{

    printf("I am before forking!\n");

    fork( );

    printf("I am after forking\n");

    printf("\t I am process having process id %d and parent process id is %d.\n", getpid( ),getppid());

    return 0;

    }
```

Output:

Example Task 2: (Add output in word file)

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(int argc, char**argv)
{
        pid_t pid;
        printf("Hello World!\n");
        printf("I'm the parent process & 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 if(pid>0)
        {
                printf("I am the parent process and pid is: %d .\n",getpid());
        }
        else if(pid<0)
        {
                printf("Error fork failed\n");
        }
        return 0;
}
```

Output:

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**".

pid_t wait (int * status)

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**.

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.

Example Task 3: (Add output in word file)

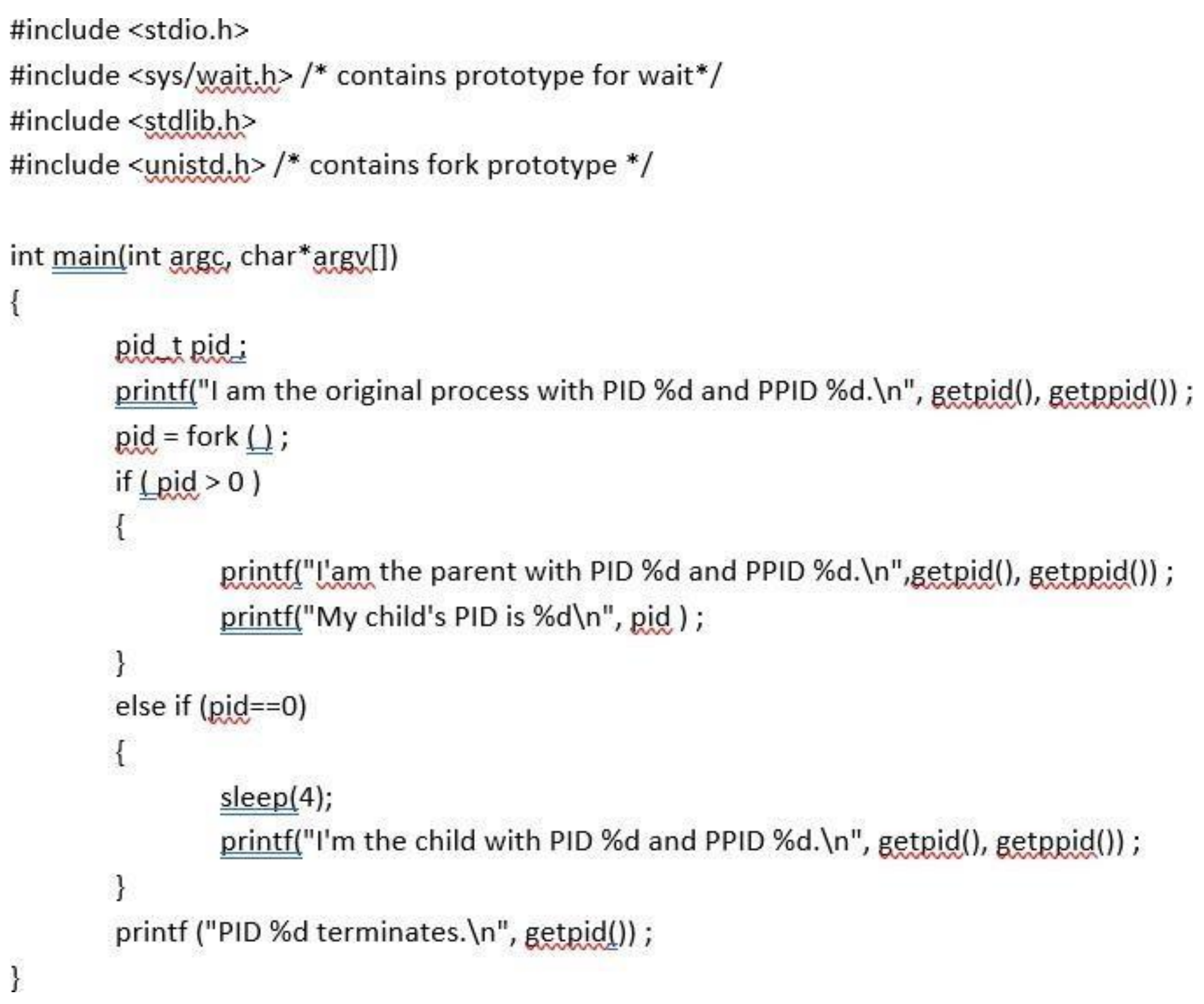
```
#include <stdio.h>
#include <sys/wait.h> /* contains prototype for wait*/
#include <stdlib.h>
#include <unistd.h> /* contains fork prototype */
int main(int argc , char * argv [])
{
        pid_t pid;
        int status=0;
        printf("Hello World!\n");
        pid = fork( );
        if (pid <0) /* 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");
        }
}
```

Output:

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 Task 4: (Add output in word file)



```
#include <stdio.h>
#include <sys/wait.h> /* contains prototype for wait*/
#include <stdlib.h>
#include <unistd.h> /* contains fork prototype */

int main(int argc, char*argv[])
{
        pid_t pid ;
        printf("I am the original process with PID %d and PPID %d.\n", getpid(), getppid()) ;
        pid = fork ( ) ;
        if ( pid > 0 )
        {
                printf("I'am the parent with PID %d and PPID %d.\n",getpid(), getppid()) ;
                printf("My child's PID is %d\n", pid ) ;
        }
        else if (pid==0)
        {
                sleep(4);
                printf("I'm the child with PID %d and PPID %d.\n", getpid(), getppid()) ;
        }
        printf ("PID %d terminates.\n", getpid()) ;
}
```

Output:

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*.**

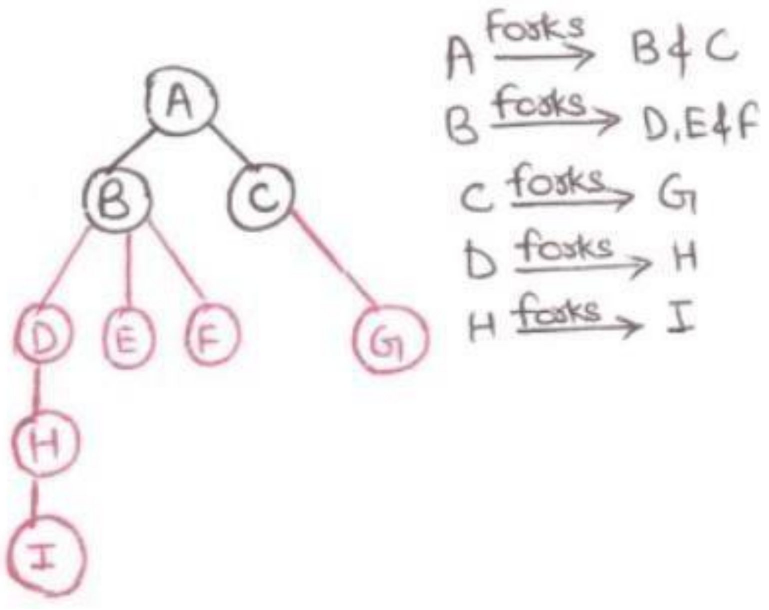
Example Task 5: (Add output in word file)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> /* contains fork prototype */
int main ( )
{
        pid_t pid ;
        pid = fork();
        if ( pid > 0 )      /* pid is non-zero, so I must be the parent */
        {
                While (1){
                        sleep(100);
                }
        }
        else if(pid==0)
        {
                exit (0) ;
        }
}
```

Output:

CP
Task 1

Write a C Program using fork system call to simulate the following scenario.



Task05.1: Run the above program and understand the output. See the status of both processes using “ps -a”. How the child can be terminated?

Description:

Example-06: ***Describes what happens when fork() is called multiple times***

```
#include<stdio.h>
int main(){
 fork();
 fork();
 fork();
 printf("Hello fork...\n");
 return 0;
}
```

Task-06: Compile and run above program and discuss the output with tree diagram:

Output:

Description With Tree Diagram:

Example-07: ***Describes what happens when fork() is called multiple times using &&.***

```
#include<stdio.h>
int main(){
 fork() && fork();
 printf("UCP\n");
 return 0;
}
```

Task-07: Compile and run above program and discuss the output with tree diagram:
Output:

Description With Tree Diagram:

Example-08: ***Describes what happens when fork() is called multiple times using ||.***

```
#include<stdio.h>
int main(){
 fork() || fork();
 printf("UCP\n");
 return 0;
}
```

Task-08: Compile and run above program and discuss the output with tree diagram:
Output:

Description With Tree Diagram:

Replacing a process code with binary (executable):
The exec system call is used to execute a file which is residing in an active process. When exec is called the previous executable file is replaced and new file is executed.
More precisely, we can say that using exec system call will replace the old file or program from the process with a new file or program. The entire content of the process is replaced with a new program. The user data segment which executes the exec() system call is replaced with the data file whose name is provided in the argument while calling exec(). The new program is loaded into the same process space. The current process is just turned into a new process and hence the process id PID is not changed, this is because we are not creating a new process we are just replacing a process with another process in exec. If the currently running process contains more than one thread then all the threads will be terminated and the new process image will be loaded and then executed. There are no destructor functions that terminate threads of current process. PID of the process is not changed but the data, code, stack, heap, etc. of the process

are changed and are replaced with those of newly loaded process. The new process is executed from the entry point.
Exec system call is a collection of functions and in C programming language, the standard names for these functions are as follows:

1. **execl**
2. **execle**
3. **execlp**
4. **execv**
5. **execve**
6. **execvp**

It should be noted here that these functions have the same base *exec* followed by one or more letters. These are explained below:

Why exec is used?

exec is used when the user wants to launch a new file or program in the same process.

Inner Working of exec

Consider the following points to understand the working of exec:

1. Current process image is overwritten with a new process image.
2. New process image is the one you passed as exec argument
3. The currently running process is ended
4. New process image has same process ID, same environment, and same file descriptor (because process is not replaced process image is replaced)
5. The CPU stat and virtual memory is affected. Virtual memory mapping of the current process image is replaced by virtual memory of new process image.

Syntaxes of exec family functions:

The following are the syntaxes for each function of exec:

```
int execl(const char* path, const char* arg, …)
int execlp(const char* file, const char* arg, …)
int execle(const char* path, const char* arg, …, char* const envp[])
int execv(const char* path, const char* argv[])
int execvp(const char* file, const char* argv[])
int execvpe(const char* file, const char* argv[], char *const envp[])
```

Example-09: Use of exec() family system calls.

```
#include<stdio.h>
int main(){
 int cpid = fork();
 if (cpid == 0){
 execl("/bin/ls", "myls", "-l", "/home/", NULL);
 printf("This line will not be printed\n");
 }
 else{
	wait(NULL);
	printf("Hello I m Parent.\n");
 }
 return 0;
}
```

Output: Understand the output and paste here.

Description:

Example-10: Running calculator program using exec in child.

```
int main()
{
 int cpid = fork();
 if (cpid == 0){
 execl("/usr/bin/gnome-calculator", "mycalc",NULL);
 printf("This line will not be printed\n");
 }
 else{
	wait(NULL);
	printf("Hello I m Parent.\n");
 }
 return 0;
}
```

Output: Understand the output and paste here.

Description:

Task-11: Write a program that adds two numbers . Compile and run it to test. Now write another program as given in Example-10. Now instead of passing gnome-calculator, pass it executable of your addition program. Does this work?

Task-12: Run the   sample programs to get proof of concept about sperate copy of variables in child and parent, vfork, etc. See the attached list of programs.