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.

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:

Process Binary Replacement:

Forking provides a way for an existing process to start a new one, but what about the case where the new process is not part of the same program as the parent process? This is the case in the shell; when a user starts a command it needs to run in a new process, but it is unrelated to the shell.

This is where the **exec** system call comes into play exec will replace the contents of the currently running process with the information from a program binary. The following code replaces the child process with the binary created for hello.c

Step 1: Create a file “hello.c” and type following source code

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

Step 2: Compile and make a binary file named hello.out

Step3: Create another file “parent.c”. Type following code.

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
        pid_t pid;
        printf("I am the parent process and pid is : %d.\n",getpid());
        printf("Here I am before use of forking\n");
        pid = fork(); //new process is created
        printf("Here I am just after forking\n"); //Child process will start execution from this line
        if (pid == 0)
        {
                printf("I am the child process and pid is :%d.\n",getpid());
                printf(“I am loading „hello“ process\n”);
                execl(“hello.out”,”hello.out”,NULL);
        }
        else
                printf("I am the parent process and pid is: %d.\n",getpid());
}
```

Task 1

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