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

Monitoring Processes

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

```
ps
```

This option list 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. To display information about your processes that are currently running.

```
ps
```

2. To display tree structure of your processes.

```
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.

Pid_t getppid()

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

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.

Graded Task 1: (2 Marks)

```
#include <stdio.h>

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

int main(void)

{

    printf("Hello World!\n");

    pid_t result = fork( );

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

    printf("\t I am process %d.\n", getpid( ));

    return 0;

}
```

Output:

Graded Task 2: (2 Marks)

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
        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
        {
                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.

Graded Task 3: (2 Marks)

```
#include <stdio.h>
#include <sys/wait.h> /* contains prototype for wait*/
#include <stdlib.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
        pid_t pid;
        int status;
        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.

Graded Task 4: (2 Marks)

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

int main()
{
        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*.**

Graded Task 5: (2 Marks)

```
#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 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");
                execv("hello.out","hello.out",NULL);
        }
        else
                printf("I am the parent process and pid is: %d.\n",getpid());
}
```

Graded Task 6: (11 +2+2+2+3=20 Marks)

You have to create a file having name **Math.c** and implement functions of the following prototypes in it.

void sum(int a , int b);
void subtract(int a , int b);
void multiply(int a , int b);
void divide(int a , int b);
void fib(int a)
void fact(int a)
void even(int a)
void odd(int a)
void prime(int a)
void square(int a)
void cube(int a)

Write driver code in it as well. Now create an executable file with name **math.out**

Create another file name parent.c parent.c received command line argument (containing relevant inputs and function name which you have created in math.c) and pass to child process, then child process load the binary math.out and execute that particular function.

Sample Output 1:

./parent.out 2 3 sum

Math.out loaded

sum=5

Sample Output 2:

./parent.out 3 prime

Math.out loaded

3 is a prime number

Graded Task 7: (2 +2+1+5=10 Marks)

The Collatz conjecture concerns what happens when we take any positive integer n and apply the following algorithm:

N = N/2 if n is even
N = 3 X N+1 if n is odd

The conjecture states that when this algorithm is continually applied, all positive integers will eventually reach 1.
For example, if n = 35, the
sequence is
35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1
Write a C program using the fork() system call that generates this sequence in the child process. The starting number will be provided from the command line. For example, if 8 is passed as a parameter on the command line, the child process will output 8, 4, 2, 1. Because the parent and child processes have their own copies of the data, it will be necessary for the child to output the sequence. Have the parents invoke the wait() call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a positive integer is passed on the command line