Week 3 – Assignments

1. Write a C program to demonstrate zombie process (A/P)

Program

```
2.
3.
4.  //zombie.c
5.  #include<stdio.h>
6.  #include<unistd.h>
7.  int main()
8.  {
9.     pid_t t;
10.    t=fork();
11.    if(t==0)
12.    {
13.        printf("Child having id %d\n",getpid());
14.     }
15.     else
16.     {
17.         printf("Parent having id %d\n",getpid());
18.         sleep(15); // Parent sleeps. Run the ps command during
    this time
19.     }
20. }
21.
```

22. **Output:**
23. $gcc zombie.c
24. $./a.out &
25. 

2. Write a C program to demonstrate orphan process (A/P)

```
// A C program to demonstrate Orphan Process.
// Parent process finishes execution while the
// child process is running. The child process
// becomes orphan.
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
	// Create a child process
	int pid = fork();

	if (pid > 0)
		printf("in parent process");

	// Note that pid is 0 in child process
	// and negative if fork() fails
	else if (pid == 0)
	{
		sleep(30);
		printf("in child process");
	}
return 0;
}
```

3. Write a C program to display process id and group id (A/P)

/*C program to get Process Id and Parent Process Id in Linux.*/

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    int p_id, p_pid;

    p_id = getpid(); /*process id*/
    p_pid = getpid(); /*parent process id*/

    printf("Process ID: %d\n", p_id);
    printf("Parent Process ID: %d\n", p_pid);
    return 0;
}
```

4. Write a C program to display user id, group id, parent id, process id

```
#include<stdio.h>
#include<unistd.h>
int main()
{

  printf("Process ID: %d\n", getpid() );
  printf("Parent Process ID: %d\n", getppid() );

  return 0;
}
```

5. Write a C program to display process statements before and after forking (A/P)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    /* fork a process */
    fork();
    /* the child and parent will execute every line of code
after the fork (each separately)*/
    printf("Hello world!\n");
    return 0;
}
```

The output will be:

```
Hello world!
Hello world!
```

6. Write a C program to display child process-id, parent process-id, process id before and after forking (A/P)

```
#include<stdio.h>



int main()
{
        for(int i=0;i<5;i++) // loop will run n times (n=5)
        {
                if(fork() == 0)
                {
                        printf("[son] pid %d from [parent] pid %d\n",getpid(),getppid());
                        exit(0);
                }
        }
```

```
        for(int i=0;i<5;i++) // loop will run n times (n=5)
        wait(NULL);

}
```

7. Write a C program to explain waiting chain of process (A/P)

```
// C program to demonstrate working of wait()
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
        if (fork()== 0)
                printf("HC: hello from child\n");
        else
        {
                printf("HP: hello from parent\n");
                wait(NULL);
                printf("CT: child has terminated\n");
        }

        printf("Bye\n");
        return 0;
}
```

8. Write a C program to run two processes and using sleep (A/P)

```
// C++ program to demonstrate creating processes using fork()
#include <unistd.h>
#include <stdio.h>

int main()
{
	// Creating first child
	int n1 = fork();

	// Creating second child. First child
	// also executes this line and creates
	// grandchild.
	int n2 = fork();

	if (n1 > 0 && n2 > 0) {
		printf("parent\n");
		printf("%d %d \n", n1, n2);
		printf(" my id is %d \n", getpid());
	}
	else if (n1 == 0 && n2 > 0)
	{
		printf("First child\n");
		printf("%d %d \n", n1, n2);
		printf("my id is %d \n", getpid());
	}
```

```
        else if (n1 > 0 && n2 == 0)
        {
                printf("Second child\n");
                printf("%d %d \n", n1, n2);
                printf("my id is %d \n", getpid());
        }
        else {
                printf("third child\n");
                printf("%d %d \n", n1, n2);
                printf(" my id is %d \n", getpid());
        }
        return 0;
}
```

9. Write a C program to run two processes closing of write end of parent process and read end of child process (A/P)

```
// C program to demonstrate use of fork() and pipe()
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
        // We use two pipes
```

```
// First pipe to send input string from parent
// Second pipe to send concatenated string from child

int fd1[2]; // Used to store two ends of first pipe
int fd2[2]; // Used to store two ends of second pipe

char fixed_str[] = "forgeeks.org";
char input_str[100];
pid_t p;

if (pipe(fd1) == -1) {
        fprintf(stderr, "Pipe Failed");
        return 1;
}
if (pipe(fd2) == -1) {
        fprintf(stderr, "Pipe Failed");
        return 1;
}

scanf("%s", input_str);
p = fork();

if (p < 0) {
        fprintf(stderr, "fork Failed");
        return 1;
}
```

```
// Parent process
else if (p > 0) {
        char concat_str[100];

        close(fd1[0]); // Close reading end of first pipe

        // Write input string and close writing end of first
        // pipe.
        write(fd1[1], input_str, strlen(input_str) + 1);
        close(fd1[1]);

        // Wait for child to send a string
        wait(NULL);

        close(fd2[1]); // Close writing end of second pipe

        // Read string from child, print it and close
        // reading end.
        read(fd2[0], concat_str, 100);
        printf("Concatenated string %s\n", concat_str);
        close(fd2[0]);
}

// child process
else {
```

```
        close(fd1[1]); // Close writing end of first pipe

        // Read a string using first pipe
        char concat_str[100];
        read(fd1[0], concat_str, 100);

        // Concatenate a fixed string with it
        int k = strlen(concat_str);
        int i;
        for (i = 0; i < strlen(fixed_str); i++)
                concat_str[k++] = fixed_str[i];

        concat_str[k] = '\0'; // string ends with '\0'

        // Close both reading ends
        close(fd1[0]);
        close(fd2[0]);

        // Write concatenated string and close writing end
        write(fd2[1], concat_str, strlen(concat_str) + 1);
        close(fd2[1]);

        exit(0);
    }
}
```

10. Write a C program to implement pipe system call (A/P)

```
// C program to illustrate
// pipe system call in C
#include <stdio.h>
#include <unistd.h>
#define MSGSIZE 16
char* msg1 = "hello, world #1";
char* msg2 = "hello, world #2";
char* msg3 = "hello, world #3";

int main()
{
        char inbuf[MSGSIZE];
        int p[2], i;

        if (pipe(p) < 0)
                exit(1);

        /* continued */
        /* write pipe */

        write(p[1], msg1, MSGSIZE);
        write(p[1], msg2, MSGSIZE);
        write(p[1], msg3, MSGSIZE);
```

```
        for (i = 0; i < 3; i++) {
                /* read pipe */
                read(p[0], inbuf, MSGSIZE);
                printf("% s\n", inbuf);
        }
        return 0;
}
```

11. Write a C program to create related process (parent/child through pipes) (A/P)

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

int main() {
  int pipefds[2];
  int returnstatus;
  char writemessages[2][20]={"Hi", "Hello"};
  char readmessage[20];
  returnstatus = pipe(pipefds);

  if (returnstatus == -1) {
    printf("Unable to create pipe\n");
    return 1;
  }

  printf("Writing to pipe - Message 1 is %s\n", writemessages[0]);
  write(pipefds[1], writemessages[0], sizeof(writemessages[0]));
  read(pipefds[0], readmessage, sizeof(readmessage));
  printf("Reading from pipe – Message 1 is %s\n", readmessage);
  printf("Writing to pipe - Message 2 is %s\n", writemessages[0]);
  write(pipefds[1], writemessages[1], sizeof(writemessages[0]));
  read(pipefds[0], readmessage, sizeof(readmessage));
  printf("Reading from pipe – Message 2 is %s\n", readmessage);
  return 0;
}
```

12. Write a C program to create a pipe by calling pipe function. (A/P)

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

int main(void) {
  int pipefds[2];

  if(pipe(pipefds) == -1) {
    perror("pipe");
    exit(EXIT_FAILURE);
  }

  printf("Read File Descriptor Value: %d\n", pipefds[0]);
  printf("Write File Descriptor Value: %d\n", pipefds[1]);

  return EXIT_SUCCESS;
}
```

13. Write a C program to show data is the pipe flow through the kernel? (A/P)

```c
// C program to implement pipe in Linux

#include <errno.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/wait.h>

#include <unistd.h>


int main()

{

        // array of 2 size a[0] is for reading

        // and a[1] is for writing over a pipe

        int a[2];


        // opening of pipe using pipe(a)

        char buff[10];
```

```
        if (pipe(a) == -1)
        {
                perror("pipe"); // error in pipe
                exit(1); // exit from the program
        }

        // writing a string "code" in pipe
        write(a[1], "code", 5);
        printf("\n");

        // reading pipe now buff is equal to "code"
        read(a[0], buff, 5);

        // it will print "code"
        printf("%s", buff);
}
```

14. What do you mean by process and different states of process?

The process executes when it changes the state. **The state of a process is defined by the current activity of the process**. Each process may be in any one of the following states − New − The process is being created. Running − In this state the instructions are being executed.

15. Differentiate Process Vs Threads in Operating System

A process is a program under execution i.e an active program. A thread is a lightweight process that can be managed independently by a scheduler. Processes require more time for context switching as they are more heavy. Threads require less time for context switching as they are lighter than processes.

16. Explain in detail about Kernel and its Functions in Operating System

The kernel is the central manager of these processes. It knows which hardware resources are available and which processes need them. It then allocates time for each process to use those resources. The kernel is critical to a computer's operation, and it requires careful protection within the system's memory.