# CHAPTER TWENTY THREE

# C Under Linux

[A] State True or False:

(a) We can modify the kernel of Linux OS.

Answer: True

(b) All distributions of Linux contain the same collection of applications, libraries and installation scripts.

Answer: False

(c) Basic scheduling unit in Linux is a file.

Answer: False

(d) **execl( )** library function can be used to create a new child process.

Answer: False

(e) The scheduler process is the father of all processes.

Answer: False

(f) A family of **fork( )** and **exec( )** functions are available, each doing basically the same job but with minor variations.

Answer: True

(g) **fork( )** completely duplicates the code and data of the parent process into the child process.

Answer: False

(h) **fork( )** overwrites the image (code and data) of the calling process.

Answer: False

(i) **fork( )** is called twice but returns once.

Answer: False

(j) Every zombie process is essentially an orphan process.

Answer: False

(k) Every orphan process is essentially a zombie process.

Answer: False

(l) All registered signals must have a separate signal handler.

Answer: False

(m) **Blocked** signals are ignored by a process.

Answer: **False**

(o) Only one signal can be blocked at a time.

Answer: False

(p) Blocked signals are ignored once the signals are unblocked.

Answer: False

(p) If our signal handler gets called, the default signal handler still gets called.

Answer: False

(q) **gtk_main( )** function makes uses of a loop to prevent the termination of the program.

Answer: True

(r) Multiple signals can be registered at a time using a single call to **signal( )** function.

Answer: False

(s) The **sigprocmask( )** function can block as well as unblock signals.

Answer: True

**[B]** Answer the following:

(a) If a program contains four calls to **fork( )** one after the other, how many total processes would get created?

*Answer:*

Sixteen processes would get created.

(b) What is the difference between a zombie process and an orphan process?

*Answer:*

If child process terminates before parent and parent does not query the exit code of a terminated child process, then the entry of the child process continues to exist in the Process Table. Such a child process is said to be a 'Zombie'. If the parent terminates without querying the zombie child process is treated as an 'Orphan' process. Also when the parent process terminates and child processes are still running, then such child process are called as 'Orphan' process.

(c) Write a program that prints the command-line arguments that it receives. What would be the output of the program if the command-line argument is *?

*Answer:*

```
/* To print command line arguments */
# include <unistd.h>

int main ( int argc,  char *argv [ ] )
{
    int i ;
    for ( i = 0 ; i  < argc ; i++ )
        printf ( "%s\n", argv [ i ] ) ;

    return 0 ;
}
```

If the command line argument is * then it will print all the non-hidden files in the current working directory.

(d) What purpose do the functions **getpid( )** and **getppid( )** serve?

*Answer:*

The **getpid( )** function returns the process-id of the calling process. The **getppid()** function returns the process-id of the parent of the calling process. There is no such function as **getpppid( )**!

(e) Rewrite the program in the section 'Zombies and Orphans' in this chapter by replacing the **while** loop with a call to the **sleep( )** function. Do you observe any change in the output of the program?

*Answer:*

```
/* To compare loop and sleep( ) */
# include <unistd.h>
# include <sys/types.h>

int main( )
{
    unsigned int i = 0 ;
    int pid, status ;

    pid = fork( ) ;

    if ( pid == 0 )
    {
        //while ( i < 4294967295U )
        //    i++ ;

        sleep ( 4 ) ;
        printf ( "The child is now terminating\n" ) ;
    }
    else
    {
        waitpid ( pid, &status, 0 ) ;

        if ( status )
```

```
            printf ( "Parent : Child terminated normally\n" ) ;
        else
            printf ( "Parent : Child terminated abnormally\n" ) ;
    }

    return 0 ;
}
```

(f) How does **waitpid( )** prevent creation of Zombie or Orphan processes?

*Answer:*

The **waitpid( )** function will make the parent process wait till the execution of the child process is not completed. It ensures that the child process never becomes 'Orphan'. Once the child process terminates, **waitpid( )** queries its exit-code from the process table and returns back to the parent. As a result of this querying, the entry for the child process is removed from the process table. So child process never becomes a 'zombie'.

(g) How does the Linux OS know if we have registered a signal or not?

*Answer:*

Linux operating system maintains a structure that keeps track of signals registered/blocked by a process. This structure is maintained on a per process basis. Linux operating system looks up this structure from time to time. When a new signal is registered it is entered into this structure.

(h) What happens when we register a handler for a signal?

*Answer:*

Whenever a new signal handler is registered, the operating system gets the address of the signal handler and the signal

code. Next the operating system updates the structure that keeps track of registered signals for the calling process. This structure is updated so as to make a new entry for the signal code and mark it registered.

(i) Write a program to verify whether **SIGSTOP** and **SIGKILL** signals are un-catchable signals.

*Program:*

```
/* To verify SIGSTOP and SIGKILL signals are un-catchable signals */
# include <signal.h>

void intHandler ( int signum )
{
    printf ( "SIGINT recd inside sighandler\n" ) ;
}

void killHandler ( int signum )
{
    printf ( "SIGKILL is not catchable\n" ) ;
}

void stopHandler ( int signum )
{
    printf ( "SIGSTOP is not catchable\n" ) ;
}

int main( )
{
    signal ( SIGINT, intHandler ) ;
    signal ( SIGSTOP, stopHandler ) ;
    signal ( SIGKILL, killHandler ) ;

    while ( 1 )
        printf ( "Program Running\r" ) ;

    return 0 ;
```

```
}
```

(j) Write a program to handle the **SIGINT** and **SIGTERM** signals. From inside the handler for **SIGINT** signal write an infinite loop to print the message 'Processing Signal'. Run the program and make use of Ctrl + C more than once. Run the program once again and press Ctrl + C once. Then use the **kill** command. What are your observations?

*Program:*

```
/* To handle SIGINT and SIGTERM signals */
# include <signal.h>

void intHandler ( int signum )
{
    while ( 1 )
        printf ( "Processing Signal... \n" ) ;
}

void termHandler ( int signum )
{
    printf ( "SIGTERM recd inside sighandler\n" ) ;
}

void contHandler ( int signum )
{
    printf ( "SIGCONT recd inside sighandler\n" ) ;
}

int main( )
{
    signal ( SIGINT, ( void* ) intHandler ) ;
    signal ( SIGCONT, ( void* ) contHandler ) ;
    signal ( SIGTERM, ( void* ) termHandler ) ;

    while ( 1 )
        printf ( "Program Running\n" ) ;
```

```
    return 0 ;
}
```

(k) Write a program that blocks the **SIGTERM** signal during execution of the **SIGINT** signal.

*Program:*

```
/* To block SIGTERM signal during execution of SIGINT signal */

# include <signal.h>
# include <unistd.h>
# include <sys/types.h>
# include <stdio.h>

void intHandler ( int signum )
{
    sigset_t block ;
    char buffer[ 10 ] = "\0" ;

    sigemptyset ( &block ) ;
    sigaddset ( &block , SIGTERM ) ;

    sigprocmask ( SIG_BLOCK , &block , NULL ) ;

    while ( strcmp ( buffer , "n" ) != 0 )
    {
        printf ( "Enter a string: " ) ;
        gets ( buffer ) ;
        puts ( buffer ) ;
    }

    sigprocmask ( SIG_UNBLOCK , &block , NULL ) ;
}

void termHandler ( int signum )
{
```

```
    printf ( "SIGTERM recd inside sighandler\n" ) ;
}

int main( )
{
    signal ( SIGINT, intHandler ) ;
    signal ( SIGTERM, termHandler ) ;
    while ( 1 )
        printf ( "Program Running\n" ) ;

    return 0 ;
}
```