Politechnika Świętokrzyska
Kielce University of Technology

Operating systems 1

Processes and signals in Linux

Tanmay Nandanikar | Lab report 2 | 10/16/20

dr inż. Karol Tomaszewski

INTRODUCTION

This lab primarily focused on processes in unix-based systems. Specifically, creating processes, getting information about existing processes, replacing processes using the exec() command, and communication between processes in the form of signals.

1 BRIEF OVERVIEW OF THEORY

1.1 PROCESSES IN LINUX

In Unix-based systems (including linux), processes are divided into the kernel context and the user context. A user process does not have direct access to the kernel context, which contains information about the state of this process. This area can only be modified by the kernel. Some of the values in this context can be modified from the user process level through appropriate system commands.

1.2 EXECUTION OF PROCESSES

Child processes can be created using the fork() function. To execute a new program, one of the exec() functions can be used in the child process. A wait() function can be used to check the status of child process execution. If the parent process does not perform this function, the completed child process becomes a zombie process.

1.3 SIGNALS

Signals can be considered as a simple form of communication between processes. Signals are asynchronous with respect to the process execution. They can be sent from process to process or from system kernel to process. Some signals can be ignored or blocked for a specified time while others cannot be handled, ignored or blocked

2 EXERCISES

2.1 TASK 1

Prompt:

Write a program that creates a child process. The parent process should print its pid and the pid of a child process, while the child process should print out its pid and pid of the parent process.

Source code can be found in the source section (here). Output for this task is shown below:

2.1.1 Output for Task 1

```
1. PID= 44495
2. Child PID= 44496
3. PID= 44496
4. Parent PID= 44495
```

2.2 TASK 2

Prompt:

Demonstrate how the zombie processes can appear in the system.

Source code can be found in the source section (here). Output for this task is shown below:

2.2.1 Output for Task 2

```
1. sprices@pop-os:~/Documents/operating systems$ ps -elf | grep Z F
   S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME
   TTY          TIME CMD
2. 1 Z sprices    47786   47785  0  80   0 -     0 -      04:24
   pts/0    00:00:00 [a.out] <defunct>
```

2.3 TASK 3

Prompt:

Write a program that creates a child process. The parent process should wait for the child process to execute and examine the exit status of the child process.

Source code can be found in the source section (here). Output for this task is shown below:

2.3.1 Output for Task 3

```
1. In child process
```

```
2. In parent process
3. Waiting...
4. Child exited with RC=0
5. Done
```

2.4 GDB DEBUGGER WITH PROGRAM CONTAINING A FOR LOOP

Prompt:

Write a program that will create the appropriate number of child processes that will run concurrently. The number of processes should be given by the command line argument. Each of the child processes should print 4 times (repeat) their pid, pid of their parent and a number specifying which child of a given parent the current child process is (1, 2, 3 ...). Then each process should fall asleep for as many seconds as the number indicates (the first process - 1 second, the second process - 2 seconds, the third process - 3 seconds, ...). The parent process should wait for all of its child processes to complete.

Source code can be found in the source section (here). Output for this task is shown below:

2.4.1 Output for Task 4

```
1. 2
2. In fork number 1, PID = 44617, PPID = 44616
3. In fork number 1, PID = 44617, PPID = 44616
4. In fork number 1, PID = 44617, PPID = 44616
5. In fork number 2, PID = 44618, PPID = 44616
6. In fork number 1, PID = 44617, PPID = 44616
7. In fork number 2, PID = 44618, PPID = 44616
8. In fork number 2, PID = 44618, PPID = 44616
9. In fork number 2, PID = 44618, PPID = 44616
10.  In fork number 2, PID = 44619, PPID = 44617
11.  In fork number 2, PID = 44619, PPID = 44617
12.  In fork number 2, PID = 44619, PPID = 44617
13.  In fork number 2, PID = 44619, PPID = 44617
14.  All child processes have terminated.
```

2.5 TASK 5

Prompt:

Write two programs. The first program will create a child process and then replace its program (child) with the second program.

Source code can be found in the source section (here). Output for this task is shown below

2.5.1 Output for Task 5

```
1. Task completed
2. Replacing child process PID=44659
3. sprices@pop-os:~/Documents/operating systems$ Replacement
   successful, PID=44659
```

2.6 TASK 6

Prompt:

Write a program that will send the sigalrm signal to itself and handle it.

Source code can be found in the source section (here). Output for this task is shown below:

2.6.1 Output for Task 5

```
1. sprices@pop-os:~/Documents/operating systems$ ./a.out
2. 1
3. 2
4. 3
5. 4
6. 5
7. 6
8. 7
9. 8
10.  9
11.  10
```

2.7 TASK 7

Prompt:

Write a program that will create two processes. The parent process will send the sigint signal to the child process (it can be sent „manually“ by pressing Ctrl + c on the keyboard). The child process should handle this signal with the function you wrote.

Source code can be found in the source section (here). Output for this task is shown below:

2.7.1 Output for Task 7

```
sprices@pop-os:~/Documents/operating systems$ ./a.out
In child process 44792...
In child process 44792...
In child process 44792...
^C
I AM IMMORTAL
```

2.8 TASK 8

Prompt:

Write four separate programs. Each of them should support the signal of your choice. The first process will send a signal to the second process every second, the second process after receiving the signal should print a message on the screen, and then send the signal to the third process. The third process should behave like the second, and the fourth should only print a message on the screen. The countdown of time in the first process should be handled by using sigalrm.

I could not complete this task.

2.9 TASK 9

Prompt:

Write a program that will prove that the data area is shared between the child process and parent processes until one of them modifies the data.

Source code can be found in the source section (here). Output for this task is shown below:

2.9.1 Output for Task 9

```
1. PID= 44888
2. Number= 15
3. Number now equals 17 in parent
4. PID= 44888
5. Number= 17
6. PID= 44889
7. Number 15
8. Number now equals 16 in child
9. PID= 44889
10.  Number 16
11.  PID= 44888
12.  Number= 17
```

2.10 TASK 10

Prompt:

For security reasons, it is recommended that the signal handling function only performs simple operations, such as setting a flag to inform you of a signal, and the complicated operations should be performed in a separate code. Present a scheme of this solution using the parent and child processes.

Source code can be found in the source section (here). Output for this task is shown below:

2.10.1 Output for Task 10

```
1. sprices@pop-os:~/Documents/operating systems$ ./a.out
2. ^C
3. I AM IMMORTAL!!!
4. In child process 44988...
```

2.11 TASK 11

Prompt:

Show how signals can be blocked or ignored by the process.

Source code can be found in the source section (here). Output for this task is shown below:

2.11.1 Output for Task 11

```
^C
I AM IMMORTAL!!!
In child process 45108...
(From different terminal)kill -TERM 45108
I AM IMMORTAL!!!
In child process 45108...
```

2.12 TASK 12

Prompt:

It is enough for the parent process to ignore the sigchld signal so that the child process will not become a zombie process. Write a program that will check if that is true and what does the wait() and waitpid() functions return after finishing of the child process in such case.

Source code can be found in the source section (here). Output for this task is shown below:

2.12.1 Output for Task 12

```
sprices@pop-os:~$ ps -elf | grep Z F S
UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME
TTY          TIME CMD
```

ps -elf | grep Z F S did not register a.out as a zombie process.

3 CONCLUSION

During this lab, I learnt various details about process handling in linux from a programming perspective. From what I understand, this will prove to be useful to know while creating software such as drivers where knowledge about process scheduling and efficient handling of resources might be necessary.

4 SOURCE

4.1 SOURCE CODE FOR EXERCISE ONE

```
1. #include <stdio.h>
2. #include <sys/types.h>
3. #include <unistd.h>
4. int main() {
5.    int a=fork();
6.    if(a!=0){
7.    printf("PID= %d\nChild PID= %d\n",getpid(),a);
8.    }
9.    else{
10.     printf("PID= %d\nParent PID= %d\n",getpid(),getppid());
11.     }
12.     return 0;
13.  }
```

4.2 SOURCE CODE FOR EXERCISE TWO

```
1. #include <stdlib.h>
2. #include <sys/types.h>
3. #include <unistd.h>
4.
5. int main ()
6. {
7. pid_t child_pid;
8. child_pid = fork ();
9. if (child_pid > 0) {
10.  sleep (60);
11.  }
12.  else {
13.  exit (0);
14.  }
15.  return 0;
16.  }
```

4.3 SOURCE CODE FOR EXERCISE THREE

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <sys/types.h>
4. #include <sys/wait.h>
```

```c
#include <unistd.h>
int main() {
   pid_t pid = fork();
   int   chld_state;
   int a=(int)pid;
    if(pid!=0){
    printf("In child process\n");
    sleep(3);
    exit(0);
    }

    printf("In parent process\n");
    printf("Waiting...\n");
    wait(NULL);
    if (WIFEXITED(chld_state)) {
     printf("Child exited with
RC=%d\n",WEXITSTATUS(chld_state));
    }
    printf("Done\n");

    return 0;
 }
```

4.4 Source code for Exercise Four

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main() {
   int n;
   pid_t childPid, wpid;
   int status=0;
    scanf("%d",&n);
    int myvar = 0;
    for(int i=0;i<n;i++){
    if(childPid=fork()==0)
         myvar = i+1;
    }

    if(myvar!=0){
    printf("In fork number %d, PID = %d, PPID =
%d\n",myvar,(int)getpid(),(int)getppid());
    printf("In fork number %d, PID = %d, PPID =
%d\n",myvar,(int)getpid(),(int)getppid());
    printf("In fork number %d, PID = %d, PPID =
%d\n",myvar,(int)getpid(),(int)getppid());
```

```
21.     printf("In fork number %d, PID = %d, PPID =
   %d\n",myvar,(int)getpid(),(int)getppid());
22.     sleep(myvar);
23.     exit(0);
24.     }
25.     else{
26.     while((wpid=wait(&status))>0);
27.     printf("All child processes have terminated.\n");
28.     }
29.
30.     return 0;
31.  }
```

4.5 Source code for Exercise Five

```
1. 1
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <sys/types.h>
5. #include <sys/wait.h>
6. #include <unistd.h>
7. int main() {
8.    int a=fork();
9.    if(a==0){
10.     printf("Replacing child process PID=%d\n",getpid());
11.     char *args[]={"./ex5_out2", NULL};
12.     execvp(args[0],args);
13.     }
14.     if(a>0){
15.     printf("Task completed\n");
16.     }
17.     return 0;
18.  }
```

```
1.  2
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <sys/types.h>
5. #include <sys/wait.h>
6. #include <unistd.h>
7. int main() {
8.    printf("Replacement successful, PID=%d\n",getpid());
9.
10.     return 0;
11.  }
12.
```

4.6 Source code for Exercise Six

```
#include <signal.h>
#include <stdio.h>
#include <stdbool.h>
#include <unistd.h>

bool print_flag = false;

void handle_alarm( int sig ) {
    print_flag = true;
}

int main() {
    signal( SIGALRM, handle_alarm );
    alarm( 1 );
    int i=1;
    for (;;) {
        sleep( 5 );
        if ( print_flag ) {
            printf( "%d\n", i);
            i++;
            print_flag = false;
            alarm( 1 );
            if(i>10){
            return 0;
            }
        }
    }
}
```

4.7 Source code for Exercise Seven

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

void handler(int num){
   write(STDOUT_FILENO, "I AM IMMORTAL!!!\n",13);
}

 int main() {
    signal(SIGINT,handler);
    int a=fork();
    if(a!=0){
    while(1){
    printf("In child process %d...\n",getpid());
    sleep(2);
    }
    }

    return 0;
 }
```

4.9 Source code for Exercise Nine

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
   int n=15;
   int a=fork();
   if(a!=0){
     printf("PID= %d\nNumber= %d\n",getpid(),n);
     n=17;
     printf("Number now equals %d in parent\n",n);
     }
     else{
     printf("PID= %d\nNumber %d\n",getpid(),n);
     n=16;
     printf("Number now equals %d in child\n",n);
     }

```

```
    if(a!=0){
    printf("PID= %d\nNumber= %d\n",getpid(),n);
    sleep(1);
    printf("PID= %d\nNumber= %d\n",getpid(),n);
    }
    else{
    printf("PID= %d\nNumber %d\n",getpid(),n);
    }
    return 0;
}
```

4.10 Source code for Exercise Ten

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

bool immortal_flag=false;

void handler(int num){
    immortal_flag=true;
}

int main() {

    signal(SIGINT,handler);
    int a=fork();
    if(a!=0){
    while(1){
    if ( immortal_flag ) {
            printf( "\nI AM IMMORTAL!!!\n");
            immortal_flag = false;
    printf("In child process %d...\n",getpid());
    sleep(2);
    }
    }
    }
    return 0;
}
```

4.11 Source code for Exercise Eleven

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

bool immortal_flag=false;
void handler(int num){
   immortal_flag=true;
 }
 int main() {
    signal(SIGINT,handler);
    signal(SIGTERM,handler);
    int a=fork();
    if(a!=0){
    while(1){
    if ( immortal_flag ) {
             printf( "\nI AM IMMORTAL!!!\n");
             immortal_flag = false;
    printf("In child process %d...\n",getpid());
    sleep(2);
    }
    }
    }
    return 0;
 }
```

4.12 Source code for Exercise Twelve

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


void ignore(int signum){
 wait(NULL);
 }

 int main ()
 {
 pid_t child_pid;
 child_pid = fork ();
```

```
17.  if (child_pid == 0) {
18.  sleep (60);
19.  }
20.  else {
21.  signal(SIGCHLD,ignore);
22.  printf("This is the parent\n");
23.  while(1);
24.  }
25.  return 0;
26.  }
```