CHAPTER 4
PROCESS AND ITS MANAGEMENT

Learning Objectives

1. Students can perform simple Linux process management tasks
2. Students can demonstrate the use of some system calls for process management

Introduction

Process is a program in execution [1]. Please note that program and process are different. Program is a passive entity that resides on a disk, while process is a program that is running. A single program can create many processes when run multiple times. In general, a process has the following attributes in the process control block.

1. Process state: the state of the process.
2. Program counter: the location of the next instruction to execute.
3. CPU registers: the contents of all process-centric registers.
4. CPU scheduling information: information about the priorities and scheduling queue pointers.
5. Memory management information: the memory allocated to the process.
6. Accounting information: information about CPU used, current duration of the process, and time limits.
7. I/O status information: information about I/O devices allocated to process and list of open files.

During its execution, a process could change into one state to another. In Linux, these states are as follows:

1. **Running or Runnable State (R)** is the state where a new process is started. In the running state, the process takes up a CPU core to execute its code. Processes that in this state could be forced to be placed into a run queue to ensure each process have a fair share of CPU resources. If a process is in this queue, then its state now becomes runnable state, waiting for its turn to execute. Although running and runnable states are distinct, they are grouped in Linux into a single state denoted by R.

2. **Sleeping state: Interruptible (S) and Uninterruptible (D)** represents state where a process is waiting for external resources. These could be an I/O based task, such as reading from a file from disk or make a network request. Because a process cannot proceed without the requested resources, it will give up its CPU cycles to other tasks that are ready and change into a sleeping state. The interruptible sleeping state (S) will react to both signals and the availability of the requested resources. On the other hand, the uninterruptible sleeping state will only wait for the resources is available and does not react to any signals.
3. **Stopped state (T)** represents state where a process in a running or runnable states is stopped using the SIGSTOP or SIGSTP signals. The SIGSTOP signal can be performed by running "**kill STOP [pid]**". This signal cannot be ignored by a process and the process will change to the stopped state. Meanwhile, the SIGSTP signal can be performed by sending CTRL+Z using the keyboard. Unlike SIGSTOP, the SIGSTP signal can be ignored by a process and, if this happens, the process will continue to execute. A process in this state could be brought back into running or runnable states by sending the SIGCONT signal.
4. **Zombie state (Z)** represents a state where a child process completed its execution or terminated. This child process will signal SIGCHLD to the parent process and change into the zombie state. Processes in this state will remain in this state until the parent process clears them from the process table. The clear them off the table, parent process must read the exit value of the child using the wait() or waitpid() system calls.

If a process has completed its execution, the process will be terminated and not listed in the process table anymore. Figure 1 shows the states of a Linux process and the transitions between them.

There are also two types of processes [3], **foreground processes** and **background processes**. **Foreground processes** are interactive processes meaning that they need to be initiated by users. These processes take input from users and return the output. On the other hand, **background processes** could be initiated either by users or the system itself. However, users are still be able to manage these processes.

In this lab work, we will initiate some processes and perform simple process management tasks in Linux. We will also develop some simple program using C programming language to demonstrate the use of system calls to manage process.

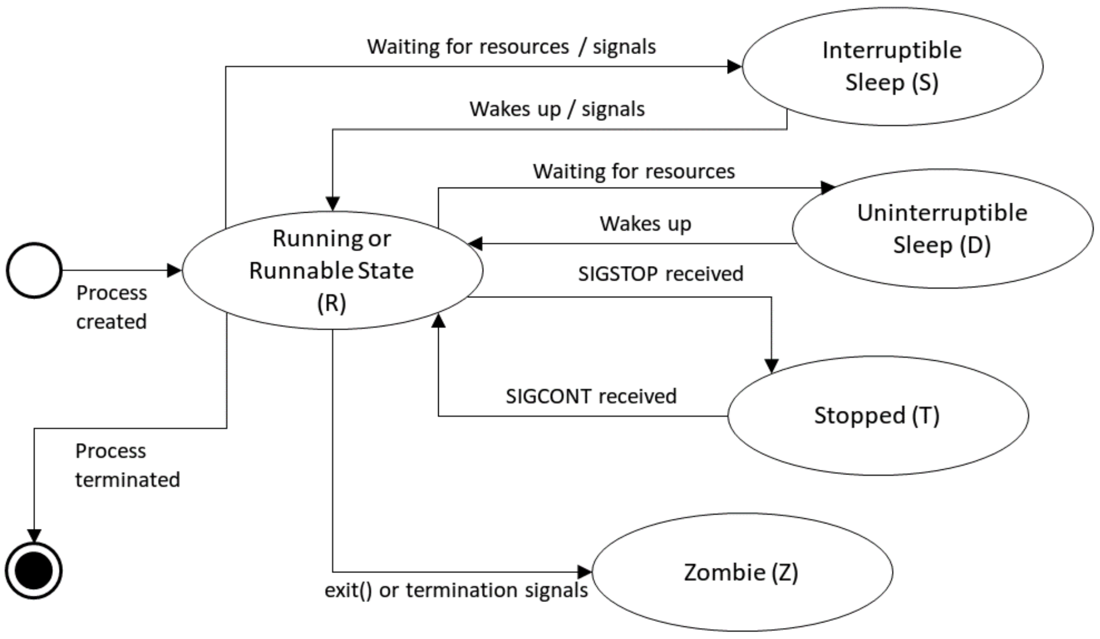

Figure 1 States of a process in Linux (adapted from [2])

4.1 Process Management Commands in Linux

In this activity, first, we will demonstrate running foreground and background processes. Then, we will use some Linux commands to see running processes and their attributes.

4.1.1 Running Background and Foreground Process

To run a process, we just need to type the program name in the terminal. For the purpose of this lab work, we will use the **sleep** command to demonstrate how the foreground and background processes work. The sleep command is a command that provides delay for a set period of time without doing anything.

First, to run a foreground process, let us run the sleep command in terminal:

```
ubuntu@primary:~$ sleep 5
ubuntu@primary:~$
```

If we run the command above, the terminal will wait for five seconds until the sleep process finish. During its execution, we cannot do anything in the terminal. This is another characteristic of a foreground process in Linux terminal.

Next, we will try running a background process. We use the same command to start a background process ended with the ampersand (&) symbol:

```
ubuntu@primary:~$ sleep 5 &
[1] 10462
```

The ampersand sign will make the terminal not wait for the corresponding command to complete, thus the command will be started as a background process.

To see the status of running processes, we can use the **jobs** command:

```
ubuntu@primary:~$ sleep 5 &
[1] 10465
ubuntu@primary:~$ jobs
[1]+  Running                 sleep 5 &
```

The result of the above jobs shows that the "sleep 5 &" process is still running in the background. Using the ampersand sign, the command will run as a background process. However, the background process will end if the terminal is closed.

4.1.2 Displaying Running Processes

To display a list of all running process, we can use the **ps** command. This command reports the current running process.

```
ubuntu@primary:~$ ps
    PID TTY          TIME CMD
  10806 pts/0    00:00:00 bash
  11072 pts/0    00:00:00 ps
```

Here, PID represents the identifier of a process, TTY represents the terminal, TIME represents total time the process has been running, and CMD is the process name. We can also print more comprehensive information using the flag **-u**.

```
ubuntu@primary:~$ ps -u
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
ubuntu     10806  0.0  0.5  10164  5160 pts/0    Ss   13:04   0:00 -bash
ubuntu     11074  0.0  0.3  10612  3276 pts/0    R+   14:27   0:00 ps -u
```

In the above result,%CPU shows the percentage of CPU used by the process, %MEM represents the percentage of memory used by the process, and STAT represents the process state. To filter the result of the ps command, we can append the **ps** command with the **grep** command using the pipe symbol.

```
ubuntu@primary:~$ sleep 100 &
[1] 11146
ubuntu@primary:~$ ps -u | grep sleep
ubuntu     11146  0.0  0.0   7228   516 pts/0    S    14:48   0:00 sleep 100
ubuntu     11148  0.0  0.0   8160   656 pts/0    S+   14:48   0:00 grep --
color=auto sleep
```

The result above shows that the sleep command is listed and in the interruptible sleep state (S). This state means the sleep command is now idle and waiting until it obtain CPU resource or receive a signal. Because CPU is a resource that being used by many processes, we may see a process is in the interruptible sleep state (S) when it is waiting for CPU resource, unless we caught it when it obtain the CPU resource (i.e., in the running state (R)).

4.1.3 Stopping or Terminating a Process

In Linux, we can stop or terminate a running process. If we stop a running process, it will switch to the stopped state (T). This state means that the stopped process can still continue to run. On the other hand, if we terminate a running process, it will be terminated and cannot be continued.

To stop a running process, we can use the shortcut CTRL+Z.

```
ubuntu@primary:~$ sleep 100
^Z
[1]+  Stopped                 sleep 100
ubuntu@primary:~$ jobs
[1]+  Stopped                 sleep 100
ubuntu@primary:~$ ps -u | grep sleep
ubuntu     11336  0.0  0.0   7228   580 pts/0    T    15:54   0:00 sleep 100
ubuntu     11338  0.0  0.0   8160   656 pts/0    S+   15:55   0:00 grep --
color=auto sleep
```

The above result shows that the sleep command is now stopped. If we check using the ps command, we will see that the sleep process is now in the T or stopped state.

To continue the sleep process or change its state into running state, we can use a command that will bring a process as a foreground or background process, namely **fg** or **bg**. If we use the **fg** command, the stopped process will be brought as a foreground process and continue its execution. Similarly, the **bg** command also continues the stopped process, but as a background process.

```
ubuntu@primary:~$ fg
sleep 100
^Z
[1]+  Stopped                 sleep 100
ubuntu@primary:~$ jobs
[1]+  Stopped                 sleep 100
ubuntu@primary:~$ bg
[1]+ sleep 100 &
ubuntu@primary:~$ jobs
[1]+  Running                 sleep 100 &
```

To terminate a process, we can use the CTRL+C shortcut. Using this command, the process will be terminated and no longer be listed in the process table.

```
ubuntu@primary:~$ sleep 100 &
[1] 11478
ubuntu@primary:~$ ps -u | grep sleep
ubuntu     11478  0.0  0.0   7228   584 pts/0    S    16:37   0:00 sleep 100
ubuntu     11482  0.0  0.0   8160   656 pts/0    S+   16:37   0:00 grep --
color=auto sleep
ubuntu@primary:~$ fg
sleep 100
^C
ubuntu@primary:~$ jobs
ubuntu@primary:~$ ps -u | grep sleep
ubuntu     11484  0.0  0.0   8160   656 pts/0    S+   16:38   0:00 grep --
color=auto sleep
```

The above result shows the execution of the sleep command as a background process such that we can see it shown in the process list. Now, we enter the fg command to move the sleep process to the foreground so that we can terminate it using CTRL+C. Afterwards, we cannot see the sleep process anymore because it has been terminated.

4.1.4 Sending Signal to Process

We can send a signal to an existing process to change its state from one to another. We can send this signal using the command **kill**. To list all the signals that can be sent to a process, we can list them using the parameter -L.

```
ubuntu@primary:~$ kill -L
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37)
SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42)
SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47)
SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-
12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-
7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-
2
63) SIGRTMAX-1  64) SIGRTMAX
```

If we do not specify the signal, the default SIGTERM will be used. This means that the default action of the kill command is to terminate a process. To send a signal to a specific process, we can specify the process id as a parameter for the kill command.

```
ubuntu@primary:~$ sleep 100 &
[1] 12082
ubuntu@primary:~$ ps -u | grep sleep
ubuntu     12082  0.0  0.0   7228   580 pts/0    S    17:16   0:00 sleep 100
ubuntu     12084  0.0  0.0   8160   656 pts/0    S+   17:16   0:00 grep --
color=auto sleep
ubuntu@primary:~$ kill 12082
ubuntu@primary:~$ ps -u | grep sleep
ubuntu     12086  0.0  0.0   8160   656 pts/0    S+   17:16   0:00 grep --
color=auto sleep
[1]+  Terminated              sleep 100
```

In the above list of commands, we first create a sleep process as a background process. Then, we check the process list to obtain the pid of the sleep process that we just created (e.g., 12082). The process id resulted may be different from time to time. Then we execute the kill command to this process such that the sleep process is terminated.

4.1.5 Showing Processes in Real Time

To show all processes in real time, we can use the command **top**. This command will show all processes in real time, including how many processes are running and sleeping, the percentage of CPU and memory currently used, and so on.

```
top - 17:52:56 up  9:19,  2 users,  load average: 0.00, 0.02, 0.03
Tasks: 103 total,   1 running, 102 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :    976.9 total,     88.1 free,    502.1 used,    386.7 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.    320.2 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    772 mysql     20   0 1336300 356944   3844 S   1.7  35.7   7:13.30 mysqld
  12385 ubuntu    20   0   10972   3900   3264 R   0.3   0.4   0:00.03 top
      1 root      20   0  170584  12152   6404 S   0.0   1.2   0:08.65 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.01 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-
kblockd
      9 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
     10 root      20   0       0      0      0 S   0.0   0.0   0:01.55 ksoftirqd/0
     11 root      20   0       0      0      0 I   0.0   0.0   0:01.83 rcu_sched
     12 root      rt   0       0      0      0 S   0.0   0.0   0:00.39 migration/0
     13 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 idle_inject/0
     14 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/0
     15 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kdevtmpfs
     16 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
     17 root      20   0       0      0      0 S   0.0   0.0   0:00.00
rcu_tasks_kthre
```

Activity 4.1

1. Create a sleep process and send a SIGSTOP signal to put this process into stopped state. Take a screenshot that shows that this process is now in the stopped state (T). *Hint: Use --help to see how to use the kill command.*
2. Send a SIGCONT signal to the stopped sleep process. Take a screenshot that shows that this process is no longer in the stopped state.

4.2 System Calls related to Process

In this part of lab work, we are going to create a zombie child process and how to address it using system calls.

4.2.1 Creating a Zombie Process

Child process is a process created by other process. This typical objective of creating a child process is to help the parent process finish its task. The child process obtain its program from the parent program as specified. Thus, the child process is somehow a duplicate of its parent process. To create a child process, we are going to use the fork system call.

```
pid_t fork(void);
```

The fork system call does not need any parameters. It returns -1 when it fails. Otherwise, it returns 0 in the child process and the process id of the child process in the parent process.

Let us start by typing the following code using "**nano fork.c**".

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

int main()
{
   pid_t p;
   printf("Starting the fork\n");
   p = fork();

   // block code for the child process
   if(p == 0) {
      printf("I am a child process: %d\n", getpid());
      printf("My parent id is %d\n", getppid());
   }
   // block of code for the parent process
   else {
      printf("I am parent process: %d\n", getpid());
      printf("My child id is %d\n", p);
      sleep(15);
   }

}
```

Then, let us compile this code and run the executable in the background such that we can see the processes in the process list.

```
ubuntu@primary:~/Codes$ ./fork.out &
[1] 13352
ubuntu@primary:~/Codes$ Starting the fork
I am parent process: 13352
My child id is 13353
I am a child process: 13353
My parent id is 13352

ubuntu@primary:~/Codes$ ps -u
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
ubuntu     13325  0.0  0.5  10164  5328 pts/0    Ss   20:01   0:00 -bash
ubuntu     13352  0.0  0.0   2488   576 pts/0    S    20:05   0:00 ./fork.out
ubuntu     13353  0.0  0.0      0     0 pts/0    Z    20:05   0:00 [fork.out]
<defunct>
ubuntu     13354  0.0  0.3  10612  3280 pts/0    R+   20:05   0:00 ps -u
```

The above result shows the child process with an id of 13353 now turns into a zombie state (Z). This happens because the parent process takes longer than the child process. After some time, the parent process is finished and it will clear the zombie process from the process table.

```
ubuntu@primary:~/Codes$ ps -u
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
ubuntu     13325  0.0  0.5  10164  5328 pts/0    Ss   20:01   0:00 -bash
ubuntu     13358  0.0  0.3  10612  3264 pts/0    R+   20:06   0:00 ps -u
[1]+  Done                    ./fork.out
```

4.2.2 Addressing Zombie Process

To avoid a zombie process, the parent process must read the status of the child process using the wait system call. This system call will make the parent process wait until it receives the finish status of the child process.

```
pid_t wait(int *wstatus);
```

This wait system call receives one parameter wstatus that represents pointer to the wait status of the finished child process. If we do not care about the status of the child process, as long as it finishes, then we can specify this parameter with NULL. To try this system call, let us create a copy of **fork.c** into **wait.c**.

```
ubuntu@primary:~/Codes$ cp fork.c wait.c
ubuntu@primary:~/Codes$ nano wait.c
```

Then, let us modify the content as follows.

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

int main()
{
   pid_t p;

   printf("Starting the fork\n");
   p = fork();

   // block code for the child process
   if(p == 0) {
      printf("I am a child process: %d\n", getpid());
      printf("My parent id is %d\n", getppid());
   }
   // block of code for the parent process
   else {
      printf("I am parent process: %d\n", getpid());
      printf("My child id is %d\n", p);
      wait(NULL);
      sleep(15);
   }
}
```

Let us run this on the background again and see the process table.

```
ubuntu@primary:~/Codes$ ./wait.out &
[1] 13494
ubuntu@primary:~/Codes$ Starting the fork
I am parent process: 13494
My child id is 13495
I am a child process: 13495
My parent id is 13494

ubuntu@primary:~/Codes$ ps -u | grep wait
ubuntu     13494  0.0  0.0   2488   512 pts/0    S    20:22   0:00 ./wait.out
ubuntu     13497  0.0  0.0   8160   724 pts/0    S+   20:23   0:00 grep --
color=auto wait
ubuntu@primary:~/Codes$ ps -u | grep wait
ubuntu     13509  0.0  0.0   8160   720 pts/0    S+   20:23   0:00 grep --
color=auto wait
[1]+  Done                    ./wait.out
```

This time, the child process is cleared by the parent process from the process list.

Activity 4.2

1. Modify the wait.c by putting the wait() function after the sleep() function. Compile and run the result in the background. Take some screenshots.
2. Take a screenshot of the process table. What happens with the child process?

References

[1] A. Silberschatz, P. B. Galvin and G. Gagne, Operating Systems Concepts, Tenth Edition, John Wiley & Sons, 2018.

[2] C. M. Jung, "Baeldung Linux," 28 10 2021. [Online]. Available: https://www.baeldung.com/linux/process-states. [Accessed 17 9 2022].

[3] Manav014, "Geeks for Geeks," 17 5 2020. [Online]. Available: https://www.geeksforgeeks.org/process-management-in-linux/. [Accessed 17 9 2022].