Process States and Signals

M. F. L. Author
A01748161@itesm.mx
ITESM CSF

Abstract— **For this report, several programs were created and tested in order to have a better understanding of processes, signals, and process states.**

Keywords*— *Process, state, signal, Cloud9, C

I. Introduction

Processes, according to the third chapter of *Operating Systems Concepts* (9th ed.), can be simply defined as an instance of a program in execution. They may also be in one of five states, as described next:

- **New:** The process is in the stage of being created.
- **Ready:** The process has all the resources available needed for it to run, but the CPU is not currently working on its instructions.
- **Running:** The CPU is working on the process's instructions.
- **Waiting:** The process cannot run at the moment, as it is waiting for some resource to become available or for some event to occur.
- **Terminated:** The process has completed.

Signals, on the other hand, are events generated by the system in response to a specific condition. Sahitya Maruvada from *100 Days of Linux* explains that this means that when a process receives a signal, it will take action according to the data being passed on to it.

A. Materials

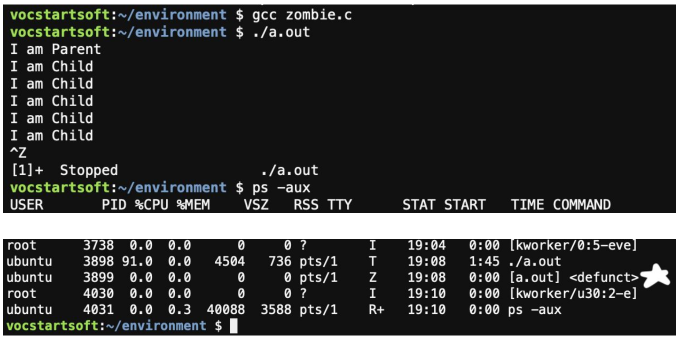
- MacBook Pro, with a 2.2 GHz 6-Core Intel Core i7 processor and macOS Catalina 10.15.6
- *Amazon Web Services (AWS) Educate* membership
  - EC2+Cloud 9 Instance

II. Development

A. Process States

For the first part of this report, it was necessary to create a program(s) where one could observe the change of states in corresponding processes. In order to achieve this, 3 programs were created; each containing a zombie, a suspended, and a terminated process, respectively.

Zombie process: Considered to be a "zombie" as it has already finished executing but still has an entry in the process table (the parent has not finished its child process.

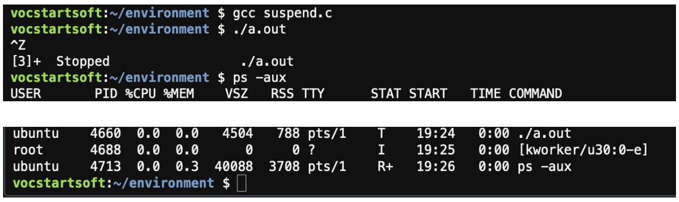


```
vocstartsoft:~/environment $ gcc zombie.c
vocstartsoft:~/environment $ ./a.out
I am Parent
I am Child
I am Child
I am Child
I am Child
I am Child
^Z
[1]+  Stopped                 ./a.out
vocstartsoft:~/environment $ ps -aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
```

```
root      3738  0.0  0.0      0     0 ?        I    19:04   0:00 [kworker/0:5-eve]
ubuntu    3898 91.0  0.0   4504   736 pts/1    T    19:08   1:45 ./a.out
ubuntu    3899  0.0  0.0      0     0 pts/1    Z    19:08   0:00 [a.out] <defunct>
root      4030  0.0  0.0      0     0 ?        I    19:10   0:00 [kworker/u30:2-e]
ubuntu    4031  0.0  0.3  40088  3588 pts/1    R+   19:10   0:00 ps -aux
vocstartsoft:~/environment $ 
```

Figures 1-2. Compiling zombie.c and confirming its defunct/zombie state.

Suspended process: Waiting for an event to occur or for a resource in order to finish.

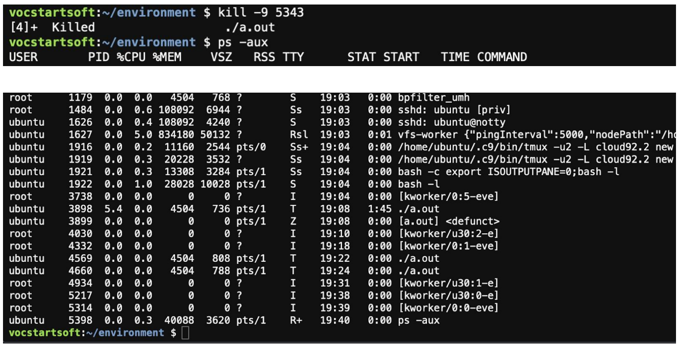


```
vocstartsoft:~/environment $ gcc suspend.c
vocstartsoft:~/environment $ ./a.out
^Z
[3]+  Stopped                 ./a.out
vocstartsoft:~/environment $ ps -aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
```

```
ubuntu    4660  0.0  0.0   4504   788 pts/1    T    19:24   0:00 ./a.out
root      4688  0.0  0.0      0     0 ?        I    19:25   0:00 [kworker/u30:0-e]
ubuntu    4713  0.0  0.3  40088  3708 pts/1    R+   19:26   0:00 ps -aux
vocstartsoft:~/environment $ 
```

Figures 3-4. Compiling and stopping suspend.c; the T indicates that it has been stopped.

Terminated process: Has completed execution and no longer appears on the process table.



```
vocstartsoft:~/environment $ kill -9 5343
[4]+  Killed                  ./a.out
vocstartsoft:~/environment $ ps -aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
```

```
root      1179  0.0  0.0   4504   768 ?        S    19:03   0:00 bpfilter_umh
root      1484  0.0  0.6 108092  6944 ?        Ss   19:03   0:00 sshd: ubuntu [priv]
ubuntu    1626  0.0  0.4 108092  4240 ?        S    19:03   0:00 sshd: ubuntu@notty
ubuntu    1627  0.0  5.0 834180 50132 ?        Rsl  19:03   0:01 vfs-worker {"pingInterval":5000,"nodePath":"/ho
ubuntu    1916  0.0  0.2  11160  2544 pts/0    Ss+  19:04   0:00 /home/ubuntu/.c9/bin/tmux -u2 -L cloud92.2 new
ubuntu    1919  0.0  0.3  20228  3532 ?        Ss   19:04   0:00 /home/ubuntu/.c9/bin/tmux -u2 -L cloud92.2 new
ubuntu    1921  0.0  0.3  13308  3284 pts/1    Ss   19:04   0:00 bash -c export ISOUTPUTPANE=0;bash -l
ubuntu    1922  0.0  1.0  28028 10028 pts/1    S    19:04   0:00 bash -l
root      3738  0.0  0.0      0     0 ?        I    19:04   0:00 [kworker/0:5-eve]
ubuntu    3898  5.4  0.0   4504   736 pts/1    T    19:08   1:45 ./a.out
ubuntu    3899  0.0  0.0      0     0 pts/1    Z    19:08   0:00 [a.out] <defunct>
root      4030  0.0  0.0      0     0 ?        I    19:10   0:00 [kworker/u30:2-e]
root      4332  0.0  0.0      0     0 ?        I    19:18   0:00 [kworker/0:1-eve]
ubuntu    4569  0.0  0.0   4504   808 pts/1    T    19:22   0:00 ./a.out
ubuntu    4660  0.0  0.0   4504   788 pts/1    T    19:24   0:00 ./a.out
root      4934  0.0  0.0      0     0 ?        I    19:31   0:00 [kworker/u30:1-e]
root      5217  0.0  0.0      0     0 ?        I    19:38   0:00 [kworker/u30:0-e]
root      5314  0.0  0.0      0     0 ?        I    19:39   0:00 [kworker/0:0-eve]
ubuntu    5398  0.0  0.3  40088  3620 pts/1    R+   19:40   0:00 ps -aux
vocstartsoft:~/environment $ 
```

Figure 5-6. Successfully terminating the process, it no longer appears in the process table.

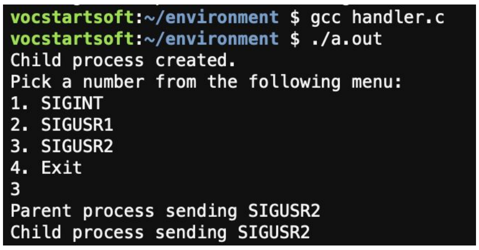
The code corresponding to the programs *(zombie.c, suspend.c, and term.c)* can be found attached to this report's *Canvas* submission.

[1] This report was sent for revision on September 24th, 2020.
The author is with the Instituto de Estudios Superiores Monterrey, Santa Fe.

B. Process Signals

The second part constituting this lab report was to write a program that creates a child process, via the fork function, that handles the signals SIGNINT (the parent should also handle this one), SIGUSR1, and SIGUSR2. Every function should print a message specifying the signal being handled, and the parent process must ask the user which signal to send; finally sending it via the kill function. A sleep function was added after each signal in order to make the output easier to understand.

The code for this program can be found attached to this report's *Canvas* submission *(under the name handler.c).*



```
vocstartsoft:~/environment $ gcc handler.c
vocstartsoft:~/environment $ ./a.out
Child process created.
Pick a number from the following menu:
1. SIGINT
2. SIGUSR1
3. SIGUSR2
4. Exit
3
Parent process sending SIGUSR2
Child process sending SIGUSR2
```

Figure 7. Correct performance from handler.c.

III. Conclusion

Although working with signals can be a confusing process, it is important to understand how these functions are actually being handled. The first part of this report was a great tool for me to apply the theoretical knowledge obtained in class into a working code. The second part was much trickier, but once I understood the parts needed and what was being asked for me to do, it turned out to be much easier than what I originally thought.

For future references, I believe it would be interesting to try to perform additional processes in *handler.c* aside from just sending the signal and printing which signal was received.

References

Ghosh, B. (April, 2020). *How to use signal handlers in C language?* Linuxhint. Recovered on September 24th, 2020 from https://linuxhint.com/signal_handlers_c_programming_language/