Navigating Linux System Commands

A guide for beginners to the Shell and GNU coreutils

Sayan Ghosh

July 1, 2024

IIT Madras
BS Data Science and Applications

Disclaimer
This document is a companion activity book for the System Commands (BSSE2001) course taught by **Prof. Gandham Phanikumar** at **IIT Madras BS Program.** This book contains resources, references, questions and solutions to some common questions on Linux commands, shell scripting, grep, sed, awk, and other system commands.

This was prepared with the help and guidance of the course instructors:

Santhana Krishnan and **Sushil Pachpinde**

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at:
`https://github.com/sayan01/se2001-book`
(You are welcome to contribute!)

Edition
Compiled on July 1, 2024

UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.

– Dennis Ritchie

Preface

Through this work I have tried to make learning and understanding the basics of Linux fun and easy. I have tried to make the book as practical as possible, with many examples and exercises. The structure of the book follows the structure of the course *BSSE2001 - System Commands*, taught by **Prof. Gandham Phanikumar** at **IIT Madras BS Program.** .

The book takes inspiration from the previous works done for the course,

- Sanjay Kumar's Github Repository
- Cherian George's Github Repository
- Prabuddh Mathur's TA Sessions

as well as external resources like:

- Robert Elder's Blogs and Videos
- Aalto University, Finland's Scientific Computing - Linux Shell Crash Course

The book covers basic commands, their motivation, use cases, and examples. The book also covers some advanced topics like shell scripting, regular expressions, and text processing using `sed` and `awk`.

This is not a substitute for the course, but a companion to it. The book is a work in progress and any contribution is welcome at `https://github.com/sayan01/se2001-book`

Sayan Ghosh

Contents

List of Figures

List of Tables

Process Management 1

1.1 What is sleep?

`sleep` is a command that is used to delay the execution of a process for a specified amount of time. **sleep** itself is a no-op command, * but it takes a variable amount of time to execute, depending on the argument of the command. This is useful when you want to delay the execution of another command or chain of commands by a certain amount of time.

1.1.1 Example

```
$ sleep 5
$ echo "Hello, World!"
Hello, World!
```

1.1.2 Scripting with sleep

If you run the above snippet, you will see that the output is delayed by 5 seconds. Moreover, the prompt itself will not be available for 5 seconds, as the shell is busy with executing the `sleep` command. To run the entire snippet as one process, simply put the two commands on separate lines of a file (say, `hello.sh`), and run the file as a script.

```
$ cat hello.sh
sleep 5
echo "Hello, World!"
$ bash hello.sh
Hello, World!
```

We will be using `sleep` in the examples throughout this chapter to demonstrate process management since it is a simple command that can be used to quickly spawn an idempotent process for any arbitrary amount of time.

1.1.3 Syntax and Synopsis

```
sleep NUMBER[SUFFIX]...
```

Here the `NUMBER` is the amount of time to sleep. The `SUFFIX` can be `s` for seconds, `m` for minutes, `h` for hours, and `d` for days.

* NO-OP stands for No Operation. It is a command that does nothing. More reading on NO-OP can be found here.

1.2 Different ways of running a process

1.2.1 What are processes?

Definition 1.2.1 (Process) A process is an instance of a program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently. Several processes may be associated with the same program; for example, opening up several instances of the same program often means more than one process is being executed. Each process has its own 'process id' or **PID** to uniquely identify it.

Whenever we run an application, or even a command on the linux shell, it spawns a process. Processes are always created by an already existing process [1] This creates a tree-like structure of processes, where each process has a parent process and can have multiple child processes. When the parent of a process dies, the child processes are adopted by the **init** process. **init** is thus the root of the process tree.

1: Other than the very first process, which is always the **init** process. In most distributions, this is done by systemd, which is an init system that does a lot of other things as well. You can learn more about systemd and what all it does here.

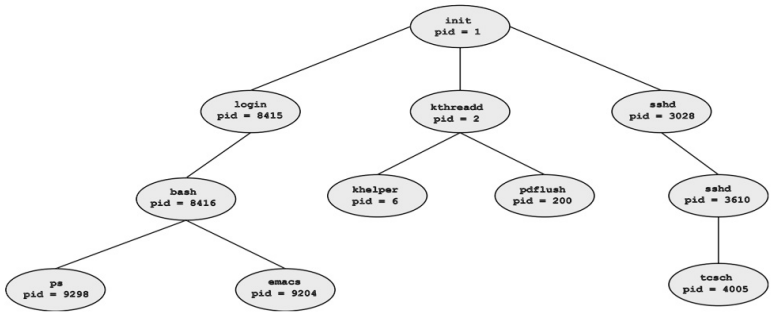

Figure 1.1: Example of a process tree

1.2.2 Process Creation

In linux systems, processes are managed by the kernel. The kernel is responsible for creating, scheduling, and destroying processes. The user can interact with the kernel using system calls to create, manage, and destroy processes. Creating processes is simple, and can be done using the **fork()** system call. This is used when any process wants to create a new process.

To simply create a new process for a command, we can simply type in the command and press enter. This will not only fork a new process from the terminal or terminal emulator as the parent process, but also tie the standard input, standard output, and standard error of the child process to the terminal or terminal emulator. [2]

```
1 $ sleep 5
```

This will create a new process that will sleep for 5 seconds.

Remember that each process has a unique process id (PID). Each process also has a parent process id (PPID), which is the PID of the parent process. If a process is created by the shell, the shell will be the parent process. If

2: Standard Input, Output, and Error are the default streams that are used by the shell to interact with the user. Standard Input is used to take input from the user, Standard Output is used to display output to the user, and Standard Error is used to display errors to the user. We will cover these in details in the next chapter.

the shell's process is killed, the child process will also be killed, as the child process is owned by the shell.

1.2.3 Process Ownership

If you are using a linux operating system with a GUI server (X or Wayland), try the following to understand how process ownership works.

Open two terminals, in the first one, run `echo $$` to see the process ID of that shell. It should print out a random string of digits, that is the PID of the shell. Then run a GUI application, such as **firefox**. [3] This will block your terminal and open a new window of firefox.

3: Make sure you are running something that is not running already.

```
$ echo $$
2277503
$ firefox
```

4: or whatever was your process's name

Now in the other terminal, which is free, run `pgrep firefox` [4] It should print out another random string of digits, it is the PID of firefox.

Now you can use the following command to find the parent process's process ID (PPID) to verify it is the same as the output of `$$` in the first terminal.

```
$ pgrep firefox
2278276
$ ps -efj | awk '$2==2278276;NR==1'
UID          PID    PPID    PGID     SID  C STIME TTY
    TIME CMD
sayan    2278276 2277503 2278276 2277503 12 16:59 pts/5
    00:00:03 /usr/lib/firefox/firefox
```

Here we can see that the PPID of firefox is the PID of the shell.

Note that the second command should put the PID of firefox, which we got from the previous command. This can also be done in a single command which you can directly copy and paste in your terminal.

```
$ ps -efj | awk "\$2==$(pgrep firefox);NR==1"
UID          PID    PPID    PGID     SID  C STIME TTY
    TIME CMD
sayan    2278276 2277503 2278276 2277503  1 16:59 pts/5
    00:00:04 /usr/lib/firefox/firefox
```

Now, what happens if we kill the parent process? To kill a process all we need to use is use the `kill` command with the PID of the process.

The `-9` flag is used to send a **SIGKILL** signal to the process. This signal is used to kill a process immediately. We will cover signals later.

```
$ kill -9 2277503
```

If you have been following along, you will see that both the terminal and firefox dissapear from your screen. You will also notice that if you run the same command to print the PID and PPID of firefox, it does not show anything. This is because the process is killed and the process tree is destroyed, so even firefox, being the child of the shell process, is killed.

1.2.4 Don't kill my children

However, there are also ways to create a new process in the background. The easiest way to do this is to append an ampersand (&) to the end of the command. This is a shell syntax that tells the shell to fork the command as a child process and run it in the background. What this means is the shell will not wait for the command to finish, and will return the prompt to the user immediately. However, the standard output and standard error may still be tied to the terminal or terminal emulator. So if the process writes something to the standard output or standard error, it will be displayed on the terminal. Furthermore, the process is still owned by the shell, and if the shell is killed, the process's parent will be changed to the **init** process.

Lets try the same exercise as earlier, but now with the & at the end.

Open two terminals, and in the first one, execute the following command.

```
$ echo $$
2400520
$ firefox &
[1] 2401297
$ echo "hello"
hello
$
ATTENTION: default value of option mesa_glthread overridden by
    environment.
$
```

You can observe that the firefox window opens up similar to last time, but now the prompt returns immediately. You can also see that the output of the `echo` command is displayed on the terminal.

If you try to perform some operations in the browser, it may also print some messages to the terminal screen, even though it is not waiting for the command to finish. The "ATTENTION" message is an example of this.

Also observe that as soon as we launched **firefox**, it printed out two numbers, [1] and 2401297. The number in the square brackets is the job id of the process, and the number after that is the PID of the process. So now we dont even need to use `pgrep` to find the PID of the process.

Now in the other terminal, run the following command.

```
$ ps -efj | awk "\$2==$(pgrep firefox);NR==1"
UID          PID    PPID    PGID     SID  C STIME TTY
    TIME CMD
sayan    2401297 2400520 2401297 2400520  3 17:13 pts/5
    00:00:08 /usr/lib/firefox/firefox
```

Still we can see that the PPID of firefox is the PID of the shell.

Now, if we kill the parent process, the child process will be adopted by the init process, and will continue to run.

```
$ kill -9 2400520
```

If you re-run the command to print the PID and PPID of firefox, you will see that the PPID of firefox is now set to 1, which is the PID of the **init** command.

```
$ ps -efj | awk "\$2==$(pgrep firefox);NR==1"
UID          PID    PPID    PGID     SID  C STIME TTY
    TIME CMD
sayan    2401297       1 2401297 2400520  3 17:13 ?
    00:00:09 /usr/lib/firefox/firefox
```

You can also see that the TTY column is now set to `?`, which means that the process is no longer tied to the terminal.

However, if instead of killing the parent process using the **SIGKILL** signal, if you sent the **SIGHUP** signal to the parent, the child process will still be terminated, as it will propagate the hangup signal to the child process.

1.2.5 Setsid

So how do we start a process directly in a way that it is not tied to the terminal? Many times we would require to start a process in the background to run asynchronously, but not always do we want to see the output of the process in the terminal from where we launched it. We may also want the process to be owned by the **init** process from the get go.

To do this, we can use the `setsid` command. This command is used to run a command in a new session. This will create a new process group and set the PPID of the process to the **init** process. The TTY will also be set to `?`.

Lets try the same exercise with the `setsid` command. Open two terminals, in one of them, run the following command.

```
$ echo $$
2453741
$ setsid -f firefox
$
```

Observe that firefox will open up, but the prompt will return immediately.

In another terminal, run the following command.

```
$ ps -efj | awk "\$2==$(pgrep firefox);NR==1"
UID          PID    PPID    PGID     SID  C STIME TTY
    TIME CMD
sayan    2454452       1 2454452 2454452  2 17:19 ?
    00:00:07 /usr/lib/firefox/firefox
```

Observe that even without killing the parent process, the PPID of firefox is already set to 1, which is the PID of the **init** process. So the process will not be killed if the shell is killed.

This is called a hang-up signal. We can still artificially send the **SIGHUP** signal, which tells firefox that its parent has stopped by using the `kill -1` command.

```
$ kill -1 2454452
```

This will still close firefox, even though the parent process (**init**) didn't actually get killed.

1.2.6 Nohup

If you do not want to give up the ownership of a child process, and also don't really need to get the prompt back, but you do not want to see the output of the command in your terminal. You can use the `nohup` command followed by the command you want to run. It will still be tied to the terminal, and you can use `Ctrl+C` to stop it, `Ctrl+Z` to pause it, etc. The prompt will also be blocked till the process runs. However, the input given to the terminal will not be sent to the process, and the output of the process will not be shown on the terminal. Instead, the output will be saved in a file named `nohup.out` in the current directory.

However, this is different from simply running the command with a redirection operator (`>`) at the end, [5] because the `nohup` command also makes the process immune to the hang-up signal.

5: We will cover redirection operators in the next chapter.

> **Exercise 1.2.1** Try the same exercise as before, but this time use the `nohup` to run firefox, then in another terminal, find the PID and PPID of firefox. Then try to kill the parent process and see if firefox dies or not.

1.2.7 coproc

The `coproc` command is used to run a command in the background and tie the standard input and standard output of the command to a file descriptor. This is useful when you want to run a command in the background, but still want to interact with it using the shell. This creates a two way pipe between the shell and the command.

Syntax

```
$ coproc [NAME] command [redirections]
```

This creates a coprocess named `NAME` and runs the command in the background. If the `NAME` is not provided, the default name is `COPROC`.

However, the recommended way to use `coproc` is to use it in a subshell, so that the file descriptors are automatically closed when the subshell exits.

```
$ coproc [NAME] { command; }
```

coproc can execute simple commands or compound commands. For simple commands, name is not possible to be specified. Compound commands like loops or conditionals can be executed using coproc in a subshell.

The name that is set becomes a array variable in the shell, and can be used to access the file descriptors for stdin, stdout, and stderr.

For example, to provide input to the command, you can use `echo` and redirection operators to write to the file descriptor.

Similarly you can use the `read` command to read from the file descriptor.

```
$ coproc BC { bc -l; }
$ jobs
[1]+  Running                 coproc BC { bc -l; } &
$ echo 22/7 >&"${BC[1]}"
$ read output <&"${BC[0]}"
$ echo $output
3.14285714285714285714
```

This uses concepts from redirection and shell variables, which we will cover in later weeks.

1.3 Process Management

1.3.1 Disown

Disown is a shell builtin command that is used to remove a job from the shell's job table. This is useful when you have started a process in the background and you want to remove it from the shell's job table, so that it is not killed when the shell is killed. What it means is that if the parent process recieves a hang-up signal, it will not propagate it to the child job if it is removed from the job table. This is applicable only for processes started from a shell.

Open two terminals, in one, open firefox in background using the &

```
$ firefox &
$
```

and then in the other terminal, run the following command.

```
$ ps -efj | awk "\$2==$(pgrep firefox);NR==1"
UID          PID    PPID    PGID     SID  C STIME TTY
    TIME CMD
sayan    3216429 3215856 3216429 3215856 69 18:45 pts/5
    00:00:02 /usr/lib/firefox/firefox
$ kill -1 3215856
```

Observe that firefox will close, even though it was running in the background. This is because the shell will propagate the hang-up signal to the child process. If the parent shell was forcefully killed using the **SIGKILL** signal, then it wont have the opportunity to propagate the hang-up signal to the child process. This is a separate process than the natural killing of firefox running in foreground even when shell is killed with **SIGKILL** signal.

Now, to fix this, we can simply run the `disown` command in the terminal where we started the firefox process.

Again open a terminal emulator and run the following command.

```
$ firefox &
$ disown
$
```

Now, in the other terminal, run the following command.

```
$ ps -efj | awk "\$2==$(pgrep firefox);NR==1"
UID          PID    PPID    PGID     SID  C STIME TTY
    TIME CMD
sayan    3216429 3215856 3216429 3215856 69 18:45 pts/5
    00:00:02 /usr/lib/firefox/firefox
$ kill -1 3215856
$ ps -efj | awk "\$2==$(pgrep firefox);NR==1"
UID          PID    PPID    PGID     SID  C STIME TTY
    TIME CMD
sayan    3216429       1 3216429 3215856 10 18:45 ?
    00:00:03 /usr/lib/firefox/firefox
```

Firefox does not close anymore, even when the parent process is hanged up.

1.3.2 Jobs

To list the jobs that are running in a shell, you can use the `jobs` command.

```
$ firefox &
$ sleep 50 &
$ jobs
[1]-  Running                 firefox &
[2]+  Running                 sleep 50 &
```

Here + denotes the current job, and - denotes the previous job. The first column is the job number, it can also be used to refer to the job inside that same shell. The process ID of a process can be used to refer to the process from anywhere, but the job ID is only valid in the shell where it is created.

The process id can be listed using the `jobs -l` command.

```
$ jobs -l
[1]- 3303198 Running                 firefox &
[2]+ 3304382 Running                 sleep 50 &
```

Using disown removes the job from this table. We can selectively remove only some jobs from the table as well.

```
$ jobs
[1]-  Running                 firefox &
[2]+  Running                 sleep 50 &
disown %1
$ jobs
[2]+  Running                 sleep 50 &
```

Whereas using `disown -a` will remove all jobs from the table. `disown -r` will remove only running jobs from the table.

If you dont really want to lose the job from the table, but you want to prevent it from being killed when the shell is killed, you can use `disown -h` to mark the jobs to be ignored by the hang-up signal. It will have same effect as last exercise, but it will still be present in the output of the `jobs` command.

1.3.3 Suspending and Resuming Jobs

Sometimes you may want to pause a job and resume it later. This is supported directly by the linux kernel. To pause any process you can send it the **SIGSTOP** or **SIGTSTP** signal. [6] This can be done using the same **kill** command. The signal number for **SIGSTOP** is 19, and for **SIGTSTP** is 20.

To resume the process, you can send it the **SIGCONT** signal. The signal number for **SIGCONT** is 18.

Exercise 1.3.1 Try to pause a job using the **SIGSTOP** signal, then resume it using the **SIGCONT** signal. Open firefox from a terminal using `firefox &` and note the PID, then pause it using `kill -19 <PID>`, try to click on the firefox window, and see if it responds. Then resume it using `kill -18 <PID>`. Does the firefox window respond now?

If you start a command from the shell without using the `&` operator, you can pause the command using `Ctrl+Z` and resume it using the `fg` command. This sends the same signals as above, and uses the shell's job table to keep track of the jobs.

Remark 1.3.1 Just like disown, the `fg` command can also take the job number as an argument to bring that job to the foreground. The default job is the current job. (Marked with a `+` in the `jobs` command))

You can also use the `bg` command to resume a job, but in the background. This has same effect as using the `&` operator at the end of the command.

Remark 1.3.2 Since the disown, fg, and bg commands work on the shell's job table, they are shell builtins, and not a executable binary. You can verify this using the `type` command.

You cannot perform job control on a process that is not started from the shell, or if you have disowned the process.

6: The difference between **SIGSTOP** and **SIGTSTP** is that **SIGSTOP** is a signal that cannot be caught or ignored by the process, so the process will be paused immediately. **SIGTSTP** is a signal that can be caught or ignored by the process, so the process can do some cleanup before pausing. The default action of **SIGTSTP** is to pause the process.

Question 1.3.1 How to get a snapshot of all the processes running? What are the commonly used flags used with it?

Answer 1.3.1 The **ps** command is used to get a snapshot of all the processes running.

- **ps** will get a snapshot of all the processes running.
- **ps -e** will show all the processes.
- **ps -f** will show full format listing.
- **ps -l** will show long format listing.
- **ps -u** will show user-oriented format listing.
- **ps -x** will show processes without controlling terminals.
- **ps -a** will show all processes with a terminal.
- **ps -A** will show all processes
- **ps aux** is a common command to see all processes
- **ps –forest** will show the processes in a tree form

Streams, Redirections, Piping | 2

Package Management 3