

Task: Using Pipes in Linux

Step 1: Using pstree with less

The **ps**tree command is used to display a tree of processes. However, its output is sometimes too long to fit on the screen. To handle this, we can use the **less** command to paginate the output.

```
pstree | less
```

The symbol `|` represents a pipe, which allows the output of one process to be used as input to another. In this case, `ps-tree` and `less` are two processes. The standard output of `ps-tree` becomes the standard input of `less`.

```
systemd--+(ModemManager)---2*[{ModemManager}]
|-NetworkManager---2*[{NetworkManager}]
|-accounts-daemon---2*[{accounts-daemon}]
|-acpid
|-avahi-daemon---avahi-daemon
|-bluetoothd
|-colord---2*[{colord}]
|-cron
|-cups-browsed---2*[{cups-browsed}]
|-cupsd
|-dbus-daemon
|-fwupd---4*[{fwupd}]
|-gdm3+--+gdm-session-wor--+gdm.wayland-ses--+gnome-session-b---2*[{gnome-session-b}]
|                                     |               ^-2*[{gdm-wayland-ses}]
|                                     |               |
|                                     |               +-----2*[{gdm-session-wor}]
|                                     |
|                                     +-----2*[{gdm3}]
|-gnome-keyring-d---3*[{gnome-keyring-d}]
|-irqbalance---(irqbalance)
|-2*[kerneloops]
|-mbim-proxy---(mbim-proxy)
|-networkd-dispat
|-packagekitd---2*[{packagekitd}]
|-polkitd---2*[{polkitd}]
|-power-profiles---2*[{power-profiles}]
|-rsyslogd---3*[{rsyslogd}]
|-rtkit-daemon---2*[{rtkit-daemon}]
|-snapd---10*[{snapd}]
|-switcheroo-cont---2*[{switcheroo-cont}]
-systemd--+(sd-pam)
|
|   |-at-spi2-registr---2*[{at-spi2-registr}]
|   |-dbus-daemon
|   |-dconf-service---2*[{dconf-service}]
|   |-evince---4*[{evince}]
|   |-evinced---2*[{evinced}]
|   |-evolution-adre---5*[{evolution-adre}]
|   |-evolution-calen---8*[{evolution-calen}]
|   |-evolution-sourc---3*[{evolution-sourc}]
```

Figure 1: Output of pstree

Step 2: Searching for a specific process using grep

We can combine `ps` with `grep` to search for specific processes, such as `bash`.

```
pstree | grep bash
```

The **grep** command is used to search for a specific string within input. In this case, it looks for any processes named **bash**. Thus, **ps** and **grep** are two separate processes, but the standard output of **ps** becomes the standard input of **grep**.

```

(base) tazmeen@afroz:~$ pstree | grep bash
|
|--gnome-terminal---+--bash--+grep
(base) tazmeen@afroz:~$

```

Figure 2: Output of `pstree | grep bash`

Step 3: Understanding file descriptors in pipes

When using pipes, two file descriptors are created: one for reading and one for writing. Here's an example of a C program that shows the file descriptors created by the pipe function:

```

1 #include <unistd.h>
2 #include <stdio.h>
3
4 int main() {
5     int pfd[2];
6     pipe(pfd);
7
8     printf("pfd[0] = %d, pfd[1] = %d\n", pfd[0], pfd[1]);
9 }

```

Listing 1: Displaying file descriptors of a pipe

```

(base) tazmeen@afroz:~$ pstree | grep bash
|
|--gnome-terminal---+--bash--+grep
(base) tazmeen@afroz:~$

```

Figure 3: Output of the program showing file descriptors

In this case, `pfd[0]` is the file descriptor for the read end of the pipe, and `pfd[1]` is the file descriptor for the write end. The numbers 3 and 4 are the next available file descriptors after `stdin`, `stdout`, and `stderr` (0, 1, and 2).

Step 4: Using pipes to communicate between parent and child processes

We can use pipes to send data from a child process to its parent process. Here's an example that demonstrates this mechanism:

```

1 #include <unistd.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int main() {
6     int pid;
7     int pfd[2];
8     char aString[20];
9
10    pipe(pfd);
11    pid = fork();
12
13    if (pid == 0) {
14        // Child process writes to the pipe

```

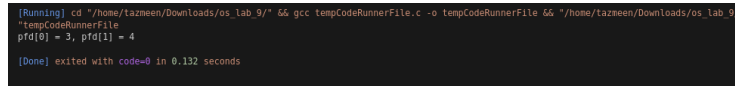
```

15     write(pfd[1], "Hello", 5);
16     printf("Child: '%s'\n", aString);
17 } else {
18     // Parent process reads from the pipe
19     printf("In parent \n");
20     printf("Before read: '%s'\n", aString);
21     read(pfd[0], aString, 5);
22     aString[5] = '\0';
23     printf("After read: '%s'\n", aString);
24 }
25
26 return 0;
27 }

```

Listing 2: Parent and child processes communicating via pipe

The parent process reads the message "Hello" written by the child process via the pipe.



```

[Running] cd "/home/tazmeen/Downloads/os_lab_9/" && gcc TempCodeRunnerFile.c -o TempCodeRunnerFile && "/home/tazmeen/Downloads/os_lab_9/TempCodeRunnerFile"
pfd[0] = 3, pfd[1] = 4
[Done] exited with code=0 in 0.132 seconds

```

Figure 4: Output of parent-child communication through pipe

```

1  #include <unistd.h>
2  #include <stdio.h>
3  #include <string.h>
4
5  int main() {
6      int pid;
7      int pfd[2];
8      char aString[20];
9
10     pipe(pfd);
11     pid = fork();
12
13     if (pid == 0) {
14         // Child process writes to the pipe and closes the write
15         // end
16         write(pfd[1], "Hello", 5);
17         close(pfd[1]); // Close the writing end after use
18         printf("Child: '%s'\n", aString);
19     } else {
20         // Parent process reads from the pipe and closes the read
21         // end
22         printf("In parent \n");
23         printf("Before read: '%s'\n", aString);
24         read(pfd[0], aString, 5);
25         aString[5] = '\0';
26         close(pfd[0]); // Close the reading end after use
27         printf("After read: '%s'\n", aString);
28     }
29
30     return 0;
31 }

```

29 }

Listing 3: Parent and child processes communicating via pipe (with closed input)

The child process writes "Hello" into the pipe and closes its writing end, while the parent process reads from the pipe and closes its reading end after reading.

```
(base) tazmeen@afroz:~/Downloads/os_lab_9$ gcc 5.c -o 5
(base) tazmeen@afroz:~/Downloads/os_lab_9$ ./5
In parent
Before read: ''
After read: 'Hello'
Child: ''
(base) tazmeen@afroz:~/Downloads/os_lab_9$
```

Figure 5: Output of parent-child communication through pipe (with closed input)

output of the last code

```
P1: P1: ^C
(base) tazmeen@afroz:~/task4$
(base) tazmeen@afroz:~/task4$ ./t3
P1: P2 SAYS: (null)
P1: hi
hello
P2 SAYS: hi

P1: P2 SAYS: hello

P1: hello
bye
P2 SAYS: hello
P2 SAYS: bye

P1: P1: bye
wait
P2 SAYS: bye

P1: P2 SAYS: wait
P1:
```

Figure 6: task4()