

CS 333: Operating Systems Lab

Autumn 2023

Lab3: Shells and Signals

In this lab, you will learn about file descriptors for IO redirection and build a simple interactive shell of your own to execute user commands, much like the bash shell in Linux.

Preamble

In Unix-like operating systems (like Linux), three file descriptors — `stdin`, `stdout`, `stderr` — are available to each process. A file descriptor is a per-process unique identifier (an integer) which refers to a file, directory, sockets, pipes etc., via which IO operations can be performed.

- **Standard Input (`stdin` - File Descriptor 0)** is the default input stream for a process and is typically connected to the terminal's keyboard input.
- **Standard Output (`stdout` - File Descriptor 1)** is the default output stream for a process and is typically connected to the screen/output device.
- **Standard Error (`stderr` - File Descriptor 2)** is the default error stream for a process and is typically connected to the screen/output device.

These three file descriptors provide a standardized way for processes to interact with the users, other programs, and handle error reporting. They are used for input and output redirection, allowing processes to read from and write to different sources, such as files or other programs' output, without needing to modify the program's code.

Programs can (and many programs do) read from `stdin` and write output and errors `stdout` and `stderr`. If these file descriptors are made to point to other end points, e.g., a file or output of another process, then the IO endpoints can be changed (IO redirection).

1. fun with files and file descriptors

Q1a. file IO system calls

This is a warm up exercise to revise the usage of file related systems calls for file IO, and to understand the differences between the file handling C library functions and the IO related system calls.

The source code file **fileio.c** contains implementation of a simple program that reads from a file and writes to a file, character-by-character. The program uses the standard C library file handling operations and file variables.

The exercise is to replace all file handling C library calls—`fopen`, `fclose`, `fread`, `fwrite` with the system calls `open`, `close`, `read`, `write`. Note that each of these calls has its own set of arguments, which in some cases may be different from the C library calls, e.g., with `open` a specification of the file permissions and user group is required.

The system calls version of the the program (cannot use any C library file handling calls) should be named **mycat1.c**

references: man pages for `open`, `close`, `read`, `write`.

auxiliary files: `in.txt`, `out.txt`, `fileio.c`

to submit: `mycat1.c`

Q1b. duplicating file descriptors + reusing file descriptors

Based on the **mycat1.c** solution, write a program **mycat2.c** to read from STDIN (file descriptor 0) and write to STDOUT (file descriptor 1). The program should run as long as input is being provided.

Use CTRL-C to stop execution of the program.

Note: the read calls need a CRLF (Enter key to be pressed) before the read buffer is processed via the system calls. **mycat2** emulates behavior of the **cat** program when used with stdin and stdout for IO.

Extend the implementation of **mycat2.c** to perform file IO on the input file (similar to **mycat1.c**), but with the **dup/dup2** system calls, i.e., input should come from **in.txt** and output should be written to the file **out.txt**.

The read/write loop used in **mycat2.c** should remain the same, i.e., read/write from stdout and stdin. Appropriate usage of the dup system calls should duplicate stdin and stdout for read and write from the file descriptors of the actual files. This version of the program should be named **mycat3.c**

references: man pages for dup/dup2

auxiliary files: in.txt, q1a/mycat1.c

to submit: mycat2.c, mycat3.c

Side note: Once the file descriptors are set up correctly the C library functions for IO — `printf` and `scanf` should work as well, instead of read and write system calls.

Q1c. IO redirection

Now that we have learnt how to use `dup()`/`dup2()` system calls. Write a program that takes a string as an input in the following format - "`command > file.txt`". The task for this question is to execute the `command` and then redirect the output of this command to a file using the `open`, `close` and `dup` system calls only.

Note that the command that is to be executed will by default write to the STDOUT file descriptor.

You are provided with a skeleton program **output_redir.c** which contains a tokenizer and the logic to parse and store the command in a variable named `comm`, and a variable to store the name of the output file called `filename`. Use your knowledge of `execvp()` and `dup()` to store the output of `command` to an output file.

Sample usage:

```
./output_redir "cat output_redir.c > out.txt"
```

```
./output_redir "ls > out.txt"
```

Sample output:

```
•→ q1c ls
input_redir input_redir.c in.txt output_redir output_redir.c out.txt
•→ q1c ./output_redir "ls > out.txt"
```

```
•→ q1c ./output_redir "ls > out.txt"
•→ q1c cat out.txt
input_redir
input_redir.c
in.txt
output_redir
output_redir.c
out.txt
```

Similar to **output_redir.c** write a program which takes input in the following format "`command < filename`". However, this time the contents of the files should be input for the command. This program should be named **input_redir.c**

Sample usage:

```
./input_redir " grep monday < in.txt"
```

Sample output:

```
monday
• → q1c ./input_redir " grep monday < in.txt"
monday
monday
○ → q1c
```

```
• → q1c cat in.txt
monday
tuesday
wednesday
thursday
friday
saturday
sunday
monday
○ → q1c
```

references: man pages for dup/dup2, exec

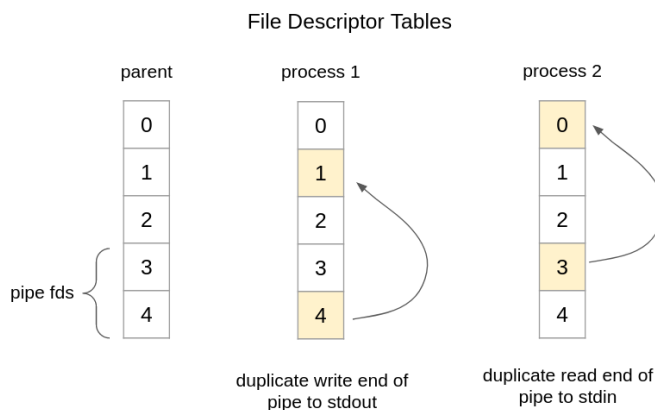
auxiliary files: in.txt, output_redir.c, input_redir.c

to submit: output_redir.c, input_redir.c

2. plumbers of the world unite!

Q 2a. plumbing 101

Write a program that takes a string as input which contains two commands separated by the pipe operator "`|`". Output of the first command should be piped as input to the second command. Both commands by default perform input and output using the stdin and stdout file descriptors.



You are provided with a skeleton program **demo_pipe.c** which contains a tokenizer and the logic to parse and store the command in `comm1` and `comm2` variables.

Hint: A main program forks two processes, in which the two programs specified with `comm1` and `comm2` execute. Before `exec` is used to load the programs, the main process uses the pipe system call for appropriate setting of the inputs and outputs of each process.

Sample usage:

```
./demo_pipe "ls | grep .c"
./demo_pipe "cat file.txt | head -7 "
```

references: man pipe

auxiliary files: demo_pipe.c

to submit: demo_pipe.c

Sample output:

```
• → q2a ls
demo_pipe demo_pipe.c
• → q2a ./demo_pipe "ls | grep .c"
demo_pipe.c
○ → q2a
```

3. simple shell

Functionalities used: fork, chdir, execvp, waitpid, kill, signal, wait

Note: All the subparts require individual C file submissions.

Q3a. basic linux commands

File **shell.c** contains skeleton code for a custom shell implementation, which implements the basic functionality of a shell. Understand the code (tokenizer and prompt input).

Compile and execute the **shell.c** program and **observe** what happens in the following cases

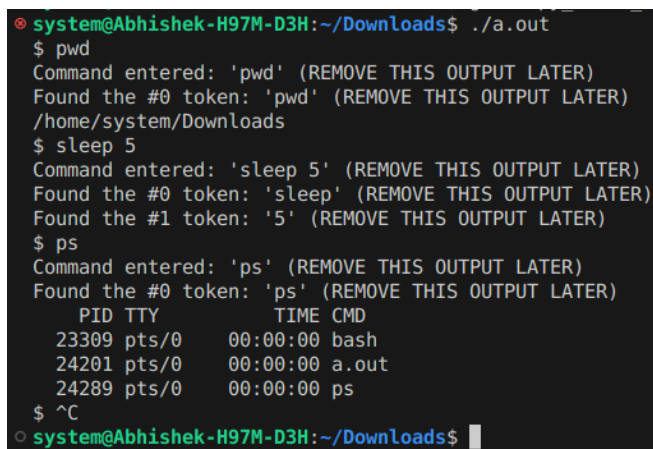
- Entering a zero argument command like - ls, pwd, ps, clear
- Entering a one argument command like - cd, sleep
- Entering a two argument command like - sleep 10 &
- Pressing CTRL+C to exit the shell

The task of this exercise is Implement the functionality by which the shell forks a child proces, and **waits** till the child process executes the command entered by the user, which may or may not have arguments. (You do not need to verify if the command is a valid command or not).

System calls of interest: fork, execvp, wait

Verify that the following commands work as expected while making sure the shell does not terminate after the command has executed completely.

Sample output:



```
system@Abhishek-H97M-D3H:~/Downloads$ ./a.out
$ pwd
Command entered: 'pwd' (REMOVE THIS OUTPUT LATER)
Found the #0 token: 'pwd' (REMOVE THIS OUTPUT LATER)
/home/system/Downloads
$ sleep 5
Command entered: 'sleep 5' (REMOVE THIS OUTPUT LATER)
Found the #0 token: 'sleep' (REMOVE THIS OUTPUT LATER)
Found the #1 token: '5' (REMOVE THIS OUTPUT LATER)
$ ps
Command entered: 'ps' (REMOVE THIS OUTPUT LATER)
Found the #0 token: 'ps' (REMOVE THIS OUTPUT LATER)
  PID TTY          TIME CMD
 23309 pts/0    00:00:00 bash
 24201 pts/0    00:00:00 a.out
 24289 pts/0    00:00:00 ps
$ ^C
system@Abhishek-H97M-D3H:~/Downloads$
```

To submit: Build upon the skeleton code provided in the auxiliary file **shell.c** to implement the above mentioned functionality and submit a new file with the solution named **shell_3a.c**

Q 3b. Overriding signals

A process can communicate with another process using signals, which are a type of inter-process communication (IPC) mechanism. Signals allow the transfer of information, which can also be used to control the execution of a process. In our case, we want to learn how to change the behavior of a process on receiving a specific signal.

Some important examples of signals include:

- SIGINT: Issued if the user sends an interrupt signal (**Ctrl + C**)
- SIGKILL: If a process gets this signal it must quit immediately and will not perform any clean-up

operations

- SIGTERM: Software termination signal (allows process to perform clean-up operations)

As we have already tried in Part 3a, issuing CTRL+C in our shell terminates the shell program, but we want to change this using user-level signal handler functions.

- **Signal handler functions** can be defined by programs to override the default functionalities of the system calls, and to give them new, custom functionalities.
- Custom signal handler functions for a signal **must be registered** using the **signal** system call

```
signal(SIGINT, signal_handler_function);  
//SIGINT is the signal to be handled
```

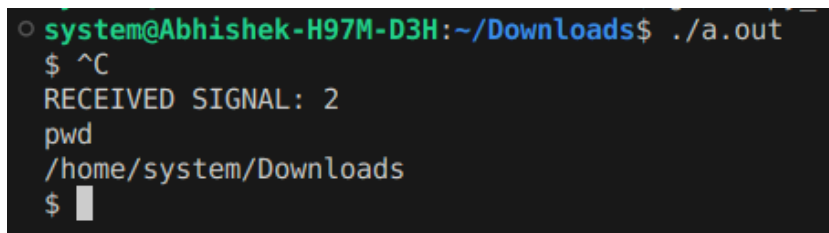
- **man signal** or **kill -l** for a detailed description of signals.

The task of this exercise is to write a signal handler function, which makes our custom shell do nothing on receiving the SIGINT (CTRL+C) signal, and hence the shell does not exit.

Instead, the signal number of the signal is to be printed by the shell:

```
"\nRECEIVED SIGNAL: <signal_num>\n".
```

The expected output after CTRL+C is pressed is as follows (you can continue writing other commands):



```
system@Abhishek-H97M-D3H:~/Downloads$ ./a.out  
$ ^C  
RECEIVED SIGNAL: 2  
pwd  
/home/system/Downloads  
$
```

- **How will the shell quit?**

To submit: Build upon the code you have written in the Part 3a to implement the above mentioned functionality and submit the new file as **shell_3b.c**

Q3c. exiting a shell

Now that we've made our shell powerful enough to overcome CTRL+C, we have to find a way to terminate it before it becomes too powerful.

The task of this exercise is to add functionality so that when the entered command matches the string **exit**, the shell program **should quit**. The termination of the shell program should be achieved using the **kill** system call. The kill system call is to be used for **self termination** of the shell process. The syntax of the **kill** system call is as follows:

- `int kill(pid_t pid, int signum);` // int can also be used on the place of pid_t
- pid is the process ID of the process to whom the signal is to be sent, and signum is the number/name of the signal to be sent. Some of these names are given in Q3b above.

Additionally, register a signal handler for the signal being sent for termination and print a message "Exiting via SIGTERM" before exiting the shell program.

Sample output

```
system@Abhishek-H97M-D3H:~/Downloads$ ./a.out
$ pwd
/home/system/Downloads
$ exit
Exiting via SIGTERM
system@Abhishek-H97M-D3H:~/Downloads$
```

To submit: Build upon the code you have written in the Part 3b to implement the above mentioned functionality and submit the new file as **shell_3c.c**

Submission Instructions

- All submissions have to be done via moodle. Name your submission as <rollnumber_lab3>.tar.gz (e.g 190050096_lab3.tar.gz)
- The tar should contain the following files in the following directory structure:
<rollnumber_lab3>/
 - q1a
 - __mycat1.c
 - q1b
 - __mycat2.c
 - __mycat3.c
 - q1c
 - __output_redir.c
 - __input_redir.c
 - q2a
 - __demo_pipe.c
 - q2b
 - __multipipe.c [optional]
 - q3
 - __shell_3a.c
 - __shell_3b.c
 - __shell_3c.c
 - __shell_3d.c [optional]
 - __shell_3e.c [optional]
 - __shell_3f.c [optional]
- Your code should be well commented and readable.
- `tar -czvf <rollnumber_lab3>.tar.gz <rollnumber_lab3>`

Deadline: Thursday 17th August 2023 5:00 PM via moodle.

Optional Exercises

Q2b. (optional) multi-pipe network

Write a program that takes a string as input argument where multiple commands are separated by "|". The first command outputs to the second, and second to third and so on. The output of the last command should be displayed to the console. You are provided with a skeleton program **multipipe.c** which contains a tokenizer.

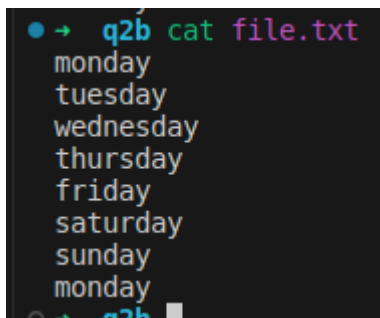
Syntax:

```
./multipipe "command_1 | command_2 | command_3 | .... | command_N"
```

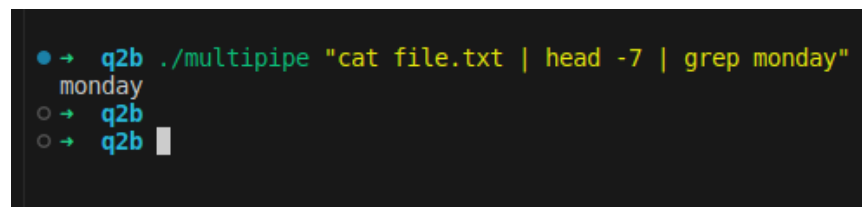
Sample Input:

```
./multipipe "cat file.txt | head -7 | grep monday"
./multipipe "cat file.txt | head -7 | tail -2 | grep i "
```

Sample output:



```
q2b cat file.txt
monday
tuesday
wednesday
thursday
friday
saturday
sunday
monday
```



```
q2b ./multipipe "cat file.txt | head -7 | grep monday"
monday
```

auxiliary files: file.txt, multipipe.c

To submit: multipipe.c

Q3d. Implementing the “cd” command

In Linux, the cd command stands for *change directory* and is used to navigate between directories or folders within the file system. With the cd command, you can move into different directories and change your current working directory. The syntax of cd command is - *cd [directory_path]*

Observe that it seems like entering the “cd” command (say, cd ..) does not perform anything right now, as cd changes the working directory of the child process but as soon as the cd command executes and the parent (shell here) runs again, its directory still remains the same, as **the child can not change the working directory of the parent.**

The task of this exercise is that we want to add the functionality of the “cd” command to our shell. When a name (in case of a sub-directory) or the absolute path (in all cases) is entered after cd, the shell should change the current working directory of the shell to the new directory. (Note: Verify the correctness using “ls”, you can also use “pwd” to get the current working directory.)

- Hint:** you can use the “chdir” function to achieve this functionality, [by calling chdir from the](#)

parent itself

- **Hint:** you can use `strcmp` to compare the entered token with the desired string

Also, add checks to print the following error messages in the respective cases.

- Print “ERROR: NO DIRECTORY SPECIFIED\n” when 0 directory names are passed as arguments
- Print “ERROR: TOO MANY DIRECTORIES\n” when more than 1 directory names are passed as arguments
- Print “ERROR: INVALID DIRECTORY\n” when the directory entered by the user does not exist (Hint: see [man chdir](#) for return type)

Expected output:

```
system@Abhishek-H97M-D3H:~/Documents/lab3_shell$ ./a.out
$ cd
ERROR: NO DIRECTORY SPECIFIED
$ cd dir1 dir2
ERROR: TOO MANY DIRECTORIES
$ cd non_existent_dir
ERROR: INVALID DIRECTORY
$ ls
a.out demo_folder shell.c shell_sol.c
$ pwd
/home/system/Documents/lab3_shell
$ cd demo_folder
$ ls
file_in_demo_folder.txt
$ pwd
/home/system/Documents/lab3_shell/demo_folder
$
```

To submit (optional): Build upon the code you have written in the Question 3 (shell) to implement the above mentioned functionality and submit the new file as **shell_3d.c**

Q3e. Implement background process execution

Background processes/commands are those which keep executing in the background while the user can keep interacting with the shell to execute further commands. At a time, multiple background processes spawned by a shell can run concurrently.

To create a background process in linux, a succeeding ampersand(&) is added after the command name. For e.g. in the default linux shell,

- “sleep 10” runs as a foreground process and will execute the sleep command in foreground, rendering the shell unable to take any input for the next 10 seconds.
- “Sleep 10 &” runs as a background process and will execute the sleep command in background while keeping the shell active, **so that the user can execute other commands in the shell in the meantime.**

Note: To monitor the currently running and zombie processes, use the command “watch -n 1 ps -au” in a new terminal tab, which will update the output from “ps -au” every 1 second.

The task of this exercise is to implement the following:

- Add a functionality in the provided shell code to **check if the last character** entered is **ampersand(&)**. If yes, the process should execute in the **background**, i.e., the shell must not block while waiting for this process to end.
- The shell **MUST** block and hence wait for the foreground process (commands without an ampersand).
- Make sure to remove/replace with NULL the ampersand(&) sign from the tokens array in case of background commands, before passing it to “execvp”.
- Verify your code by executing “sleep 10” and “sleep 10 &”.

You can observe the working of foreground and background processes respectively as:

```
system@Abhishek-H97M-D3H:~/Documents/Lab3_shell$ ./a.out
$ sleep 10
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
system	2346	0.0	0.0	162384	4856	tty2	Ssl+	Jul24	0:00	/usr/libexec/gdm-wayland
system	2349	0.0	0.0	223032	7024	tty2	SL+	Jul24	0:00	/usr/libexec/gnome-shell
system	488055	0.0	0.0	11268	5452	pts/0	Ss	15:35	0:00	bash
system	488989	0.0	0.0	11152	4172	pts/0	S+	15:44	0:00	man signal
system	488998	0.0	0.0	8752	2724	pts/0	S+	15:44	0:00	pager
system	490507	0.0	0.0	8748	5604	pts/1	Ss	16:22	0:00	/usr/bin/bash --init
system	490529	0.1	0.0	6556	3352	pts/1	S+	16:22	0:05	watch -n 1 ps -au
system	490908	0.0	0.0	8748	5652	pts/5	Ss+	16:23	0:00	/usr/bin/bash --init
system	494855	0.0	0.0	8748	5632	pts/6	Ss	16:33	0:00	/usr/bin/bash --init
system	517624	0.0	0.0	8748	5632	pts/7	Ss+	17:30	0:00	/usr/bin/bash --init
system	518913	0.0	0.0	8748	5632	pts/8	Ss	17:32	0:00	/usr/bin/bash --init
system	519903	0.1	0.0	6552	3308	pts/8	S+	17:33	0:00	watch -n 1 ps -au
system	528163	0.0	0.0	2776	952	pts/6	S+	17:44	0:00	./a.out
system	528312	0.0	0.0	5724	1016	pts/6	S+	17:44	0:00	sleep 10
system	528313	0.0	0.0	6552	1168	pts/8	S+	17:44	0:00	watch -n 1 ps -au
system	528314	0.0	0.0	2892	968	pts/8	S+	17:44	0:00	sh -c ps -au
system	528315	0.0	0.0	9996	3284	pts/8	R+	17:44	0:00	ps -au

```
system@Abhishek-H97M-D3H:~/Documents/Lab3_shell$ ./a.out
$ sleep 10
$ sleep 15 &
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
system	2346	0.0	0.0	162384	4856	tty2	Ssl+	Jul24	0:00	/usr/libexec/gdm-wayland
system	2349	0.0	0.0	223032	7024	tty2	SL+	Jul24	0:00	/usr/libexec/gnome-shell
system	488055	0.0	0.0	11268	5452	pts/0	Ss	15:35	0:00	bash
system	488989	0.0	0.0	11152	4172	pts/0	S+	15:44	0:00	man signal
system	488998	0.0	0.0	8752	2724	pts/0	S+	15:44	0:00	pager
system	490507	0.0	0.0	8748	5604	pts/1	Ss	16:22	0:00	/usr/bin/bash --init
system	490529	0.1	0.0	6556	3352	pts/1	S+	16:22	0:05	watch -n 1 ps -au
system	490908	0.0	0.0	8748	5652	pts/5	Ss+	16:23	0:00	/usr/bin/bash --init
system	494855	0.0	0.0	8748	5632	pts/6	Ss	16:33	0:00	/usr/bin/bash --init
system	517624	0.0	0.0	8748	5632	pts/7	Ss+	17:30	0:00	/usr/bin/bash --init
system	518913	0.0	0.0	8748	5632	pts/8	Ss	17:32	0:00	/usr/bin/bash --init
system	519903	0.1	0.0	6552	3308	pts/8	S+	17:33	0:00	watch -n 1 ps -au
system	528163	0.0	0.0	2776	952	pts/6	S+	17:44	0:00	./a.out
system	528558	0.0	0.0	5724	1016	pts/6	S+	17:44	0:00	sleep 15
system	528612	0.0	0.0	6552	1168	pts/8	S+	17:44	0:00	watch -n 1 ps -au
system	528613	0.0	0.0	2892	968	pts/8	S+	17:44	0:00	sh -c ps -au
system	528614	0.0	0.0	9996	3308	pts/8	R+	17:44	0:00	ps -au

Optional optional:

Reap the background process (zombies) when going back to the main shell loop to wait for a new command at the prompt.

Hint: waitpid, WNOHANG

To submit (optional): Build upon the code you have written in the Question 3 (shell) to implement the above mentioned functionality and submit the new file as **shell_3e.c**

Q3f Generating signals

As now you're familiar with suppressing signals, let's learn to generate signals. The “kill” function in C can be used to send signals to other processes or to itself, its syntax is provided below:

- `int kill(pid_t pid, int sig);` // int can also be used on the place of pid_t
- Here, pid is the Process ID of the process to whom the signal is sent, and sig is the

number/name of the signal to be sent. Some of these names are given in Q3 above.

The task is to implement a functionality where, when we give the **user-defined command** “end <pid>”, where <pid> is the process ID of one of the background children process, that background process is terminated.

- Hint: To test this functionality, execute a long running background command like “sleep 60 &”. Open a new terminal tab and use “watch -n 1 ps -au” to monitor and get the PID of that process. Finally, pass that PID as an argument to the “end” command to your shell and see the status of that background process.
- Hint: Use `atoi(tokens[1])` to convert the PID passed as an argument from char array to int
- Assume that only 1 argument with the correct PID is provided

The expected outputs before and after executing the “end” command are as follows:

```
system@Abhishek-H97M-D3H:~/Documents/lab3_shell$ ./a.out
$ sleep 60 &
$
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
system	2346	0.0	0.0	162384	4856	tty2	Ssl+	Jul24	0:00	/usr/libexec/gdm-wayl
system	2349	0.0	0.0	223032	7020	tty2	Sl+	Jul24	0:00	/usr/libexec/gnome-se
system	551122	0.0	0.0	8748	5636	pts/0	Ss	18:36	0:00	/usr/bin/bash --init-
system	556696	0.0	0.0	11268	5432	pts/4	Ss	18:48	0:00	bash
system	556840	0.0	0.0	10768	3796	pts/4	S+	18:48	0:00	man 2 kill
system	556848	0.0	0.0	8752	2736	pts/4	S+	18:48	0:00	pager
system	561237	0.0	0.0	8748	5588	pts/6	Ss	18:58	0:00	/usr/bin/bash --init-
system	561359	0.1	0.0	6552	3296	pts/6	S+	18:58	0:00	watch -n 1 ps -au
system	563622	0.0	0.0	2776	952	pts/0	S+	19:01	0:00	./a.out
system	564091	0.0	0.0	5724	1016	pts/0	S+	19:01	0:00	sleep 60
system	564166	0.0	0.0	6552	1160	pts/6	S+	19:01	0:00	watch -n 1 ps -au
system	564167	0.0	0.0	2892	968	pts/6	S+	19:01	0:00	sh -c ps -au
system	564168	0.0	0.0	9996	3260	pts/6	R+	19:01	0:00	ps -au

```
system@Abhishek-H97M-D3H:~/Documents/lab3_shell$ ./a.out
$ sleep 60 &
$ end 564091
$
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
system	2346	0.0	0.0	162384	4856	tty2	Ssl+	Jul24	0:00	/usr/libexec/gdm-wayl
system	2349	0.0	0.0	223032	7020	tty2	Sl+	Jul24	0:00	/usr/libexec/gnome-se
system	551122	0.0	0.0	8748	5636	pts/0	Ss	18:36	0:00	/usr/bin/bash --init-
system	556696	0.0	0.0	11268	5432	pts/4	Ss	18:48	0:00	bash
system	556840	0.0	0.0	10768	3796	pts/4	S+	18:48	0:00	man 2 kill
system	556848	0.0	0.0	8752	2736	pts/4	S+	18:48	0:00	pager
system	561237	0.0	0.0	8748	5588	pts/6	Ss	18:58	0:00	/usr/bin/bash --init-
system	561359	0.1	0.0	6552	3296	pts/6	S+	18:58	0:00	watch -n 1 ps -au
system	563622	0.0	0.0	2776	952	pts/0	S+	19:01	0:00	./a.out
system	564091	0.0	0.0	0	0	pts/0	Z+	19:01	0:00	[sleep] <defunct>
system	564375	0.0	0.0	6552	1160	pts/6	S+	19:02	0:00	watch -n 1 ps -au
system	564376	0.0	0.0	2892	972	pts/6	S+	19:02	0:00	sh -c ps -au
system	564377	0.0	0.0	9996	3296	pts/6	R+	19:02	0:00	ps -au

To submit (optional): Build upon the code you have written in the Question 3 (shell) to implement the above mentioned functionality and submit the new file as **shell_3f.c**