

## CS 236: Operating Systems Lab

### Spring 2025-26

### Lab 2: Keep calm and follow the process!

#### System calls of the day

getpid, getppid, open, close, read, write, fork, exec\*, wait, waitpid, dup/dup2, sleep

#### 1. identity proof?

- a. Write a program **p1a.c** that prints its own process identifier (*pid*) and also the process identifier of its parent (*ppid*) using the `getpid()` and `getppid()` system calls. Verify your output using the `ps/top` commands.

##### Sample output

Shell

```
> ./p1a
Process ID (PID): 77107
Parent Process ID (PPID): 76825
Press Enter to continue...
```

Shell

```
> <ps-command> | grep p1a
arghya      77107  0.0  0.0  2700  1752 pts/0    S+   18:16   0:00 ./p1a
```

- b. Extend p1a.c to **p1b.c** to enter an infinite loop **after** the character input. Check and report the **execution state** of the process before and after the character input.

##### Sample output

Shell

```
> ./p1a
Process ID (PID): 77107
Parent Process ID (PPID): 76825
Press Enter to continue...

Running infinite loop. Press Ctrl+C to exit...
```

## 2. what is your FD?

More about file descriptors: <https://bottomupcs.com/ch01s03.html>

(Downloaded as `fds.html` in `part2` directory)

- a. Write a program **p2a.c** that, in an infinite loop, reads a single character from *STDIN* and writes the same character to *STDOUT* using `read()` and `write()`.  
(**Note:** This is a simple implementation of the `cat` tool available on your machine. First, use the `cat` tool to check for sample usage)

### Sample output

```
Shell
> ./p2a
Hello OS!!
Hello OS!!
I'm great at syscalls...
I'm great at syscalls...
^C
```

- b. Extend `p2a.c` to **p2b.c** to read from a file instead of *STDIN*. The input filename is specified as an input argument of the program. (**Hint:** Use `open()` to open input fd)

### Sample output

```
Shell
> ./p2b input.txt
Konnichiwa!
I'm just an OS sample input file...
```

- c. Extend `p2b.c` to **p2c.c** to write to a file instead of *STDOUT*. The output filename is also specified as the second input argument to the program. (**Hint:** Use `open()` to open output fd)

### Sample output

```
Shell
> ./p2c input.txt output.txt
> ./p2b output.txt
Konnichiwa!
I'm just an OS sample input file...
```

- d. Extend p2a.c to **p2d.c** to get the functionality of p2c.c by using the `dup2()` system call. The loop should read from *STDIN* and write to *STDOUT* just like p2a.c but act like p2c.c

#### Sample output

```
Shell
> ./p2d input.txt output.txt
> ./p2b output.txt
Konnichiwa!
I'm just an OS sample input file...
```

### 3. OS cutlery is incomplete without `fork()`

- a. Analyze the C program **p3a.c** that calls `fork()` three times consecutively. Each resulting process prints its own process ID using `getpid()`. Determine and report the following:
- What is the **number of processes** created?
  - What is the **number of outputs** produced?
  - Explain the **non-deterministic order** of the output.
- (To know more about `fork()`, use: `man fork`)
- b. Write a C program **p3b.c** to demonstrate process creation using the `fork()` system call. Insert the `fork()` call and complete the conditional statements to correctly handle **fork failure**, **child process execution**, and **parent process execution**. Print the *PID* and *PPID* in the child and the *PID* and *child PID* in the parent.

#### Sample output

```
Shell
> ./p3b
Parent: 172473 | Child PID: 172474
Child: 172474 | PPID: 172473
```

- c. Extend p3b.c to **p3c.c** to make the child sleep for 5 secs after printing using `sleep()`, while the parent waits for the child to terminate using `wait()` before printing.

#### Sample output

```
Shell
> ./p3c
Parent: 175137 | Waiting for child to terminate...
```

```
Child: 175138 | PPID: 175137
Child: 175138 | Terminating...
Parent: 175137 | Child terminated!
Parent: 175137 | Child PID: 175138
```

Now further modify the parent process to include a `getchar()` call before invoking `wait()`, so that the parent pauses execution. While the parent is blocked on `getchar()` and the child has exited, observe and report the **process states of both the parent and child** using appropriate tools. Comment on and explain the observed states of the parent and child processes during this period.  
(For usage check: `man sleep/wait`)

- d. Extend p3c.c to **p4d.c** to demonstrate **non-blocking child monitoring** and status collection using `waitpid()`.

Create two child processes of the same parent process, and use sleep with different durations with each child process. One of the child processes should exit normally and the parent process should reap it and print its exit status.

For the second child, make sure the sleep duration is long enough to use its PID and kill the process from another terminal.

Use `kill -9 Child_PID` to **kill** the process, kill -9 is a signal and will be covered in the next lab, you can refer to `man kill` for more information.

The parent process should repeatedly check the child's status using `waitpid()` with the **non-blocking option**, printing an appropriate message (e.g., *"No child to reap"*) while the child processes are still running. Once a child process terminates, the parent should correctly retrieve and process different **exit status**.

(The parent process **should not block**, use `man waitpid` to check how you can use different options with `waitpid`, also check for **wait-related status macro** to handle different exit status)

### Sample output

```
Shell
> ./p3d
Parent PID: 146555
Child PID: 146556
Child PID: 146557

Parent process waiting for children to exit ...
```

```

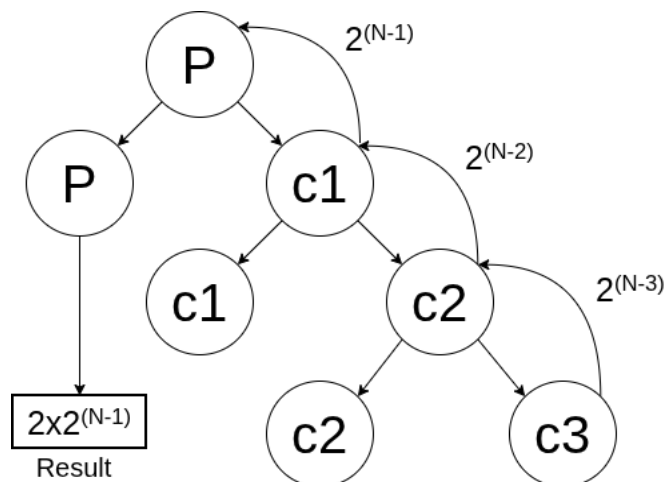
Parent: 146555 | No child to reap, parent continues...
Parent: 146555 | No child to reap, parent continues...
Parent: 146555 | Reaped child (PID): 146557
Child killed by signal 9
Parent: 146555 | No child to reap, parent continues...
Parent: 146555 | No child to reap, parent continues...
Child exited
Parent: 146555 | Reaped child (PID): 146556
Child exited normally
Exit code = 0
Parent: 146555 | No more children to reap!
Parent exited

```

- e. Write a C program **p3e.c** that uses **process creation and exit status propagation** to compute powers of two. The program should accept a single command-line argument **N**, representing the depth of process creation. Using a loop and the **fork()** system call, create a **linear chain of child processes**, where each child continues execution to the next level.

The **deepest child process** should terminate with an exit status of 1. Each parent process must wait for its child using **wait()**, retrieve the child's exit status, multiply it by 2, and then exit with the updated value. The original (root) process should print the final computed result, which corresponds to  **$2^N$** .

(In the beginning stick with  $N \leq 8$ , then after completing this part check what happens if you keep  $N > 8$ .)



### Sample output

```
Shell
> ./p3e
Usage: ./p3e N

> ./p3e 5
Root Parent: 228591
Level 1 | Child PID: 228592
Level 2 | Child PID: 228593
Level 3 | Child PID: 228594
Level 4 | Child PID: 228595
Level 5 | Child PID: 228596
Level 5 | Child PID: 228596 | Exit Status: 1
Level 4 | Child PID: 228595 | Exit Status: 2
Level 3 | Child PID: 228594 | Exit Status: 4
Level 2 | Child PID: 228593 | Exit Status: 8
Level 1 | Child PID: 228592 | Exit Status: 16
Parent(228591) returned Power: 32
```

## 4. to exec is to exist!

- Compile both **power2.c** and **power3.c**, and run **./power2** to observe what it does. Analyze the C program **power2.c** and justify this behaviour, e.g., how many exit messages do you see and why?
- Write a program **p4b.c** that takes single word commands as input, forks into a child and runs the command with **exec**. (**Note:** Use the correct version of exec, so that your shell supports simple linux commands like **ls/ps/top** etc.)

### Sample output

```
Shell
> ./p4b
$ ls
Makefile  power2  power2.c  power3  power3.c  p4b  p4b.c
$ ps
  PID TTY          TIME CMD
 76825 pts/0      00:00:00 fish
128475 pts/0      00:00:00 shell
128511 pts/0      00:00:00 ps
```

```
$ ./power3
Enter a positive integer: 5
3^5 = 243
Exiting from power3.c
$
Exiting shell...
```

## Submission Guidelines

All submissions via Moodle.

- Name your submissions as: <rollno>\_lab2.tar.gz  
For example, if your roll number is **25b0371**, then the submission will be **2bb0371\_lab2.tar.gz**
- The tar should contain the following files in the following directory structure:  
**<rollno>\_lab2/**

```
|__ report.pdf
|__ part1/
|    |__ p1a.c
|    |__ p1b.c
|__ part2/
|    |__ p2a.c
|    |__ p2b.c
|    |__ p2c.c
|    |__ p2d.c
|__ part3
|    |__ p3b.c
|    |__ p3c.c
|    |__ p3d.c
|    |__ p3e.c
|__ part4
|    |__ p4b.c
```

- Deadline: **January 15, 2026, 5:00 p.m.**

## 5. more the merrier v0.1 (optional)

- a. Write a C program **p5a.c** that accepts a **filename** and a **byte offset** as command-line arguments. The program should open the specified file, create a child process using `fork()`, and perform file read operations in both the parent and child processes.

After process creation, the **parent process** should reposition the file offset using `lseek()` based on the given offset and then read and print a fixed number of bytes from the file. The **child process** should perform its own read operation **after** the parent completes its read. (Refer to [man lseek, read](#) for usage)

Observe and analyze the output produced by both processes and **comment on the behavior of the file reads**, especially with respect to the data read by the parent and the child.

### Sample output

```
Shell
> ./p5a
Usage: ./p5a <filename> <offset>

> ./p5a ./dummy_file.txt 1000
Parent read: ///Some Text///
Child read: ///Some Text///
```

- b. Write a C program **p5b.c** to achieve the same functionality of p2d.c, but instead of running the read and write loop, `exec` the `cat` program with no arguments to read and write from files.

(Hint1: Use `fork`, `exec`, `open`, `close`, `dup2`)

(Hint2: Without `dup2`)

### Sample output

```
Shell
> ./p5b input.txt output.txt
> ./p5b output.txt
Konnichiwa!
I'm just an OS sample input file...
```