Department of Computer Science and Engineering
Indian Institute of Technology Bombay

# CS 333: Operating Systems Lab
# Autumn 2023
# Lab 2: A matter of processes

The scope of this lab is to understand system calls that relate to the process abstraction and the functionality provided by the operating system to create and manage new processes, execute programs, monitor execution state of a process, and to allow processes to communicate with each other.

Required reading/reference: https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf

## Q1: identify yourself

Write a program `p1.c` that forks a child process and prints the process identifiers of itself and its parent or the child process.

**Sample output :**

```
Parent : My process ID is:12345
Parent : The child process ID is:15644
Child : My process ID is:15644
Child : The parent process ID is:12345
```

Note: Order of the print statement outputs is not deterministic and may show up in a different order.
*Figuring out why this is the case is a homework question.*

System calls of interest: `fork, getpid, getppid`

## Q2: the waiting game

### Q2a: (auxiliary file : `p2a.c`)

You are provided a program `p2a.c` that reads an integer **n** as input from the terminal/console. Add code that forks a child process that prints if its a parent or child followed by its **PID** and numbers from $1\ to\ n$ and the parent prints its **PID** and numbers from $(n\ +\ 1)\ to\ 2n$.

**Input : 3**

**Sample output :**

```
C 3451 1
C 3451 2
P 3448 4
C 3451 3
P 3448 5
P 3448 6
```

Note: In the sample output, numbers 1, 2, and 3 are printed by the child process with pid 3451. The ordering of the sequence from $1\ to\ 2n$ does not matter. Also, note that the value of variable **n** is required and used by both the parent and child processes.

## Q2b:

Write a program **p2b.c** that reads an integer **n** as input from from the terminal/console and forks a child process that prints its **PID** and numbers from $1\ to\ n$ and the parent prints its **PID** and numbers from $(n\ +\ 1)\ to\ 2n$,
with an additional requirement that the numbers $1\ to\ 2n$ should be printed in an increasing order.

```
Input : 3
Desired Sample output :
C 3451 1
C 3451 2
C 3451 3
P 3448 4
P 3448 5
P 3448 6
```

Note: In the sample output, the numbers 1, 2, and 3 are printed by the child process with pid 3451.

System calls of interest: `fork, getpid, wait, waitpid`

## Q3: no longer paper weights

(auxiliary files : **p3a.c, helloworld.c** and **byeworld.c**)

## Q3a:

Edit the program given in `p3a.c` that takes as input the name of an executable program (You can create an executable from helloworld.c and byteworld.c) located in the same folder. Next, `p3a.c` should call a variant of exec with the specified program to execute the new program.
If everything goes well, any statement after the exec call in your program p3a.c should not execute.

## Q3b:

Write a program `p3b.c` that prints a prompt (">>> ") and takes as input the name of an executable program located in the same folder. Next, `p3b.c` should first call fork and from within the child process call a variant of exec with the specified program to execute the new program.
The parent process should fallback to the prompt and wait for input specifying the name of another executable to execute (and repeat).

For example,
Consider two executable programs `helloworld` and `byeworld` located in the same folder.

Prompt ">>> " should be printed before the user input is read
and then the name of the executable provided should be executed.

Sample output:

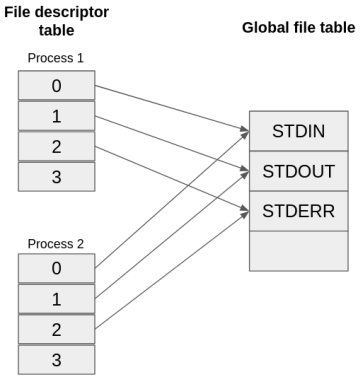
```
>>> helloworld
this is hello world program
>>> byeworld
this is bye world program
>>> ^C
```

You should be able to close this program using Ctrl + C.
Note: The name of the program at input prompt should not exceed 50 characters.

Hint: Use variations of the `exec` system call to replace the current process with a new/different process. (`man 3 exec`)

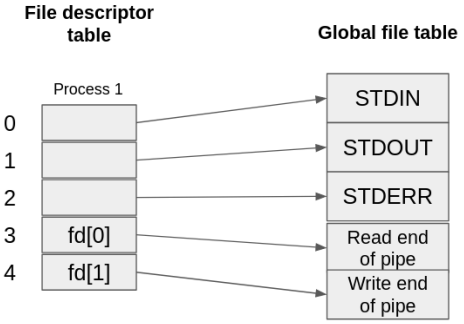
## **Q4:** just pipe it!

(auxiliary file : **`p4a.c`**)




Each process in linux is assigned a **per-process** file descriptor table which keeps track of all the open files in that process. Further, each entry in the file descriptor table points to a **global file table** as shown in the figure below which contains actual metadata about the file (an example would be where the file is actually located on disk). As shown in the figure, entries at index 0, 1 and 2 in each process's file descriptor table point to the STDIN (input from your terminal/console), STDOUT (output to your terminal/console) and STDERR (output to your terminal/console) files respectively. File descriptor entry at 0 is read only, and entries at index 1 and 2 are write only in nature.

A pipe is a mechanism for inter-process communication using file descriptors. As shown in the pseudo code below a program initializes a pipe by passing an array of 2 uninitialized file descriptors to the **`pipe`** system call. The pipe system call initializes fd[0] to point to the read end of the pipe, and fd[1] to point to the write end of the pipe as shown in the figure below. After the **`pipe`** system call, anything written to fd[1] can be read using fd[0] file descriptor.




***Pipes are used to perform Inter Process Communication (IPC), and that is what you will do now.***

```
int fd[2];
char buffer[11];
pipe(fd);
write(fd[1], "hello pipe!", 11);
read(fd[0], buffer, 11); // buffer will now contain "hello pipe!"
```

Write a program `p4.c` which creates a process A. Process A should take user input (a number) and then use the `fork()` system call to spawn process B. Process B should take another user input and add its value to the value previously taken in Process A. This sum must be printed in process A (the parent process).

**Processes A and B should communicate with each other using the pipe system call only.**

**Sample Output:**

```
→  lab2 ./a.out
Process A : Input value of x : 4
Process B : Input value of y : 6
Process A : Result after addition : 10
```

Note:
If a process tries to read before something is written to the pipe, the process is suspended until something is written and is available in the pipe.

## **Q5:** zombie and done!

A running process becomes an orphan when its parent has finished the execution or terminated. In a Unix-like operating system, any orphaned process is adopted by a special process (one of the first user-level processes).
A process that has completed its execution or terminated but still has some state (pid, memory allocation, stack, etc.) in the memory and has not been **cleaned-up** (***reaped***) is called a ***zombie*** process. A zombie process is reaped when the parent process makes a wait system call or when the parent process exits.

## Q5a:

Write a program `p5a.c` to demonstrate the state of process as an orphan.
The program should fork a child process, the parent and child processes print their PIDs and the child/parent PIDs. The child process sleeps for a few seconds (5 seconds) and re-prints information about its parent PID. By the time the child process wakes up from sleep, the parent process should have exited and the child process would have a different parent process.

**Sample Output:**

```
❯ ./a.out
Parent : My process ID is: 660712
Parent : The child process ID is: 660713
Child : My process ID is: 660713
Child : The parent process ID is: 660712

~/Downloads ····· 04:16:02 PM
❯
Child : After sleeping for 5 seconds
Child : My process ID is: 660713
Child : The parent process ID is: 1588
```

```
❯ ps 1588
    PID TTY      STAT   TIME COMMAND
   1588 ?        Ss     0:13 /lib/systemd/systemd --user

~ ·····
❯
```

**Hint:**
Use the C-library function `sleep()` for the sleep functionality. (`man 3 sleep`)

### Q5b:

Write a program `p5b.c` to demonstrate the presence of zombie processes.
The program forks a process, and the processes print PID information similar to Q5a.
Subsequently, the parent process sleeps for 1 minute and then waits for the child process to exit.
The child process waits for keyboard input from the user after displaying the messages and then exits.

Display the process state of the **child process** while it was waiting for input and after the input using the ps command, in a separate terminal window.

```
ps -o pid,stat --pid <child's PID>
```

Refer to `man ps` for the details of different process states. Reason about the output.

**Note:** The state check of the child process before and after the user input should happen before the parent process wakes up from sleep.

**Sample Output:**

```
❯ ./a.out
Parent : My process ID is: 660947
Parent : The child process ID is: 660948
Child : My process ID is: 660948
Child : The parent process ID is: 660947

Child : Check child process state

Child : Press any key to continue
a
Child : Check child process state again
Parent reaping child
```

```
❯ ps -o pid,stat 660948
    PID STAT
660948 S+

~ ·····
❯ ps -o pid,stat 660948
    PID STAT
660948 Z+

~ ·····
❯ ps -o pid,stat 660948
    PID STAT
```

## Q6 and Q7 need your attention, but do not require a submission.

## Q6: more fun with fork

### Q6a:

Write a program `p6a.c` that takes a number **n** as a command line argument and creates **n** child processes recursively, i.e., parent process creates the first child, the first child creates the second child, and so on. The child processes should exit in the reverse order of the creation, i.e., The innermost child exits first, then second innermost, and so on. Each process prints a short message (along with its PID) in the order of creation and subsequently, in the order of exit also it prints the parent id while exiting as shown in the sample. Refer to sample output files to understand the desired output format.
Sample output file : **`p6a-output.txt`**

## Q6b:

Write a program `p6b.c` that takes a number **n** as command line argument and creates **n** child processes sequentially, i.e. The parent process (`p6b`) creates all child processes in a loop without any delays. Let each child process sleep for a small duration of time (say 1 sec) and then exit.
**The parent process should exit only after all the child processes have exited.**
Each process prints a short message (along with its PID) . Refer to sample output files to understand the desired output format. Sample output file : **`p6b-output.txt`**

**Hint:** Check the return behavior of the `wait` function call.

## Q7: system calls only

We said this lab was all about processes but in the last problem you'll write a program which uses system calls dealing with files. Write a program `p7.c` that takes three filenames and an integer offset as argument, and copies the content of the first filename to the third file name, and then additionally copies the content of the second file name to the third file name starting at the mentioned offset.
The catch is that you should not use the file and IO related functions provided via `stdio.h` (e.g., scanf, printf, fopen, fread, fwrite, fprintf, fscanf, …).
op
**Hints:**
Lookup and read the man pages of `read`, `write`, `lseek`, `open` and `close` system calls. Also have a look at the `atoi` C library function.

**Sample output:** `file1` and `file2` are the input file and `file3` does not exist.

```
→  lab2 cat file1
I am Batman
→  lab2 cat file2
Superman
→  lab2 ./a.out file1 file2 file3 5
→  lab2 cat file3
I am Superman
→  lab2
```

## Submission instructions:

- All submissions via moodle. Name your submissions as: `<rollno_lab2>.tar.gz`
- Auxiliary files are available in the `auxiliaryfiles` folder.
- The tar should contain the following files in the specified directory structure:

```
<roll_number_lab2>/
        |____p1.c
        |____p2a.c
        |____p2b.c
        |____p3a.c
        |____p3b.c
        |____p4.c
        |____p5a.c
        |____p5b.c
        |____p6a.c
        |____p6b.c
        |____p7.c
```

- **Due date:** 11th August 11:59 PM