CS 333: Operating Systems Lab
Autumn 2022
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.

Problem 1: identify yourself.

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

Problem 2: children don't wait.

Problem 2a:

Write a program `p2a.c` that reads an integer **n** as input from from the terminal/console and forks a child process that prints if its 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 2*n .

```
Input : 3

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

Note: in this sample output the numbers 1, 2, and 3 are printed by the child process with pid 3451. The ordering of the sequence from 1 to 2*n does not matter.

Problem 2b:

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 2*n,
with an additional requirement that the numbers 1 to 2*n 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 this sample output that the numbers 1, 2, and 3 are printed by the child process with pid 3451.

Problem 3: no longer paper weights. (to be done
(auxiliary files : **helloworld.c** and **byeworld.c**)

Write a program `p3.c` that prints a prompt (">>> ") and takes as input the name of an executable
program located in the same folder. Next, `p3.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`)

Problem 4: system calls only.

Write a program `p4.c` that takes as input two filenames as input from the console/terminal, and copies the content of the first filename to the second file name.
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, …).

Hints:
Need to use the system call interface.
Lookup and read the man pages of `read, write ,open` and `close` system calls.
Also read about STDIN_FILENO for input without `scanf/fscanf`.
(auxiliary file : `file1`)

Sample output: `file1` is the input file and `file2` does not exist.

```
λ › cat file1
what you seek is seeking you
blaha blha balha
λ › ./a.out
file1
file2
λ › cat file2
what you seek is seeking you
blaha blha balha
λ › 
```

Problem 5: just pipe it!

A pipe is a mechanism for inter-process communication using file descriptors. A typical example is to **connect** two processes by connecting the standard output from one process to the standard input of the other process. The programmer needs to explicitly set up this connection using the `pipe` system call.

C library interface for pipes —
int pipe(int fds[2]);
Parameters: fd[0] fd for the read end of the pipe
                fd[1] fd for the write end of the pipe
Returns: 0 on Success
          -1 on error

Write a program `p5.c` which creates three processes—A, B and C. Process A should take user input (a number) which should be passed to process B. Process B should take another user input and add its value to the value passed by Process A. This modified value should be passed to Process C which displays the number received.
Processes A, B and C should communicate with each other using the pipe system call only.

Print the pipe file descriptors with something like this in the different processes for verification.
```
printf("Read File Descriptor Value: %d\n", pipefds[0]);
printf("Write File Descriptor Value: %d\n", pipefds[1]);
```

Sample Output:

```
→  task5 ./a.out
Process A : Input value of x : 4
Process B : Input value of y : 5
Process C : Result after addition : 9
```

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.

Problem 6: 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 will be adopted by a special process, the *init* process (first user-level process). 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 executes the wait system call or when the parent process exits.

Problem 6a:

Write a program `p6a.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:

```
→  task6 ./a.out                        →  task6 ps 2179
Parent : My process ID is: 11313          PID TTY       STAT  TIME COMMAND
Parent : The child process ID is: 11314   2179 ?        Ss    0:00 /lib/systemd/systemd --user
Child : My process ID is: 11314         →  task6
Child : The parent process ID is: 11313
→  task6
Child : After sleeping for 5 seconds
Child : My process ID is: 11314
Child : The parent process ID is: 2179
→  task6 ▯
```

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

Problem 6b:

Write a program `p6b.c` to demonstrate the presence of zombie processes.
The program forks a process, and the processes print PID information similar to 6a.
Subsequently, the parent process sleeps for 1 minute and then waits for the child process to exit.

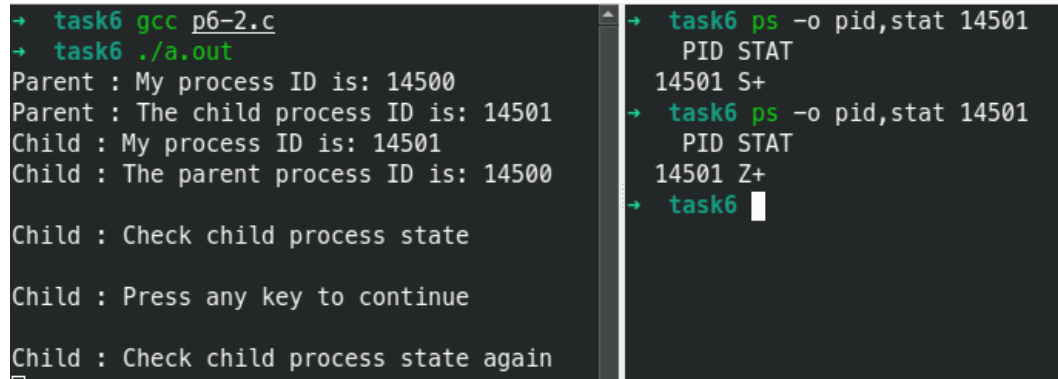The child process waits for keyboard input from the user after displaying the messages and then exits.

Display the process state of the **child process** while it was waiting for input and after the input using the ps command.
```
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:


```
→  task6 gcc p6-2.c
→  task6 ./a.out
Parent : My process ID is: 14500
Parent : The child process ID is: 14501
Child : My process ID is: 14501
Child : The parent process ID is: 14500

Child : Check child process state

Child : Press any key to continue

Child : Check child process state again
```

```
→  task6 ps -o pid,stat 14501
    PID STAT
  14501 S+
→  task6 ps -o pid,stat 14501
    PID STAT
  14501 Z+
→  task6 
```

Problem 7: more fun with fork

Problem 7a:
Write a program `p7a.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 : p7a-output.txt

Problem 7b:
Write a program `p7b.c` that takes a number **n** as command line argument and creates **n** child processes sequentially, i.e. The parent process (`p7b`) 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 : p7b-output.txt

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

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
          |____p3.c
          |____p4.c
          |____p5.c
          |____p6a.c
          |____p6b.c
          |____p7a.c
          |____p7b.c
  ```

- **Due date:** 8th August 5:00 PM