HY-345 Assignment 1

<u>Φροντιστηριο</u>: 7/10
<u>Deadline:</u> 25/10

System Calls:

- A system call (**syscall**) is a request from a process to the kernel, for the execution of a service in the operating system in which the process is executed.
- This service is a process or operation that only the kernel has privilege to execute.This could be a I/O operation or the execution of a new process.

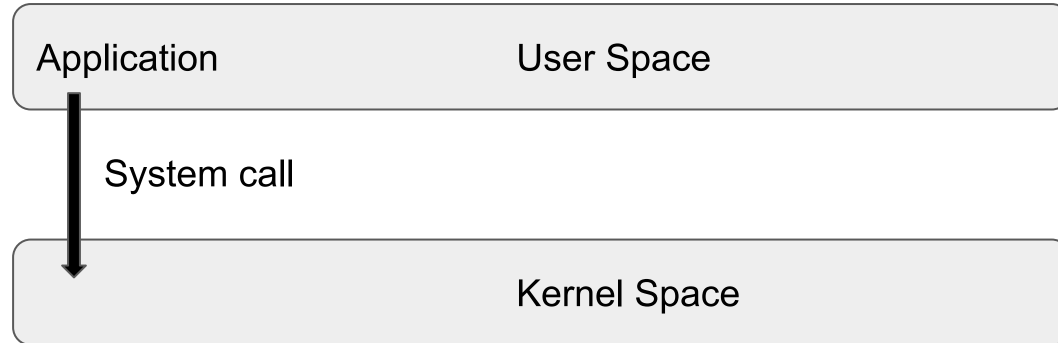
System Calls

Process Control	fork(),exec(),wait(),exit()
File Operations	open(),read(),write(),close()
Directory Management	mkdir(),rmdir(),link(),opendir(),closedir()
Other	chdir(),chmod(),kill(),time()

❖ You can find more information about system calls here: https://technobyte.org/system-calls-in-operating-systems-simple-explanation/

System Calls:

- If a process is running a user program in user mode and needs to execute a system service (e.g. reading a file's contents) it has to execute a **trap** instruction.
- A **trap** instruction is a procedure call that transfers control to the operating system.



System calls: fork()

- fork() creates a new process (**child process**)
  - It creates a process duplicate to the original one (**parent process**), including all file descriptors, registers etc.
  - The child process inherits all variables from the parent at their current state.If in a later step the child process change the value of a variable this change takes place locally (the parent will still have the old value)
- fork is called once but returns twice!
  - After a successful fork, it returns 0 to the child process and an integer representing the child’s process identifier (**PID**) to the parent

➔ Now consider how the fork will be used in your shell. When the shell receives a command it will fork a new process responsible for the command’s execution.

System calls: fork() - PID (Process Identity)

- There are three cases which you need to consider regarding the return value of fork():
  - **pid < 0 :** That means that fork was unsuccessful
  - **pid == 0 :** This is the pid of the child process (child execution space)
  - **pid > 0 :** The pid of the child process passed to the parent (parent execution space)

- You are responsible to handle each of the above cases accordingly.

System calls: fork() example

```
while(1){                  //repeat forever
    command_promt();     //display command prompt on the screen
    read_command(command,parameters);   //read input for terminal

    int pid = fork();    //fork
    if(pid == 0){        //child process,execute command

        execve(command,parameters); //execute the command

    }else if(pid > 0){  //this is the parent process

        waitpid(-1,&status,0);  //wait child process to finish its execution

    }else{

        printf("Fork not executed successfuly!\n")  //handle failed forks

    }
```

- **!You should always wait for a child process to finish smoothly it's execution before continuing with the parent process (use of the waitpid() system call)**

System calls: exec()

- The exec() is family of system calls that is used to execute a command by replacing the current process with the one that the command dictates (loads a new program within the current process).

- File descriptors are preserved across a call to exec .

- Upon success the exec() **never** returns a value:
  - If it returns something then the execution of the command failed

System calls: exec()

- The exec() family consists of the following system calls:
  - int execl(char *path, char *arg, ...);
  - int execlp(const char *file, const char *arg, ...);
  - int execle(const char *path, const char *arg, ..., char * const envp[]);
  - int execv(const char *path, char *const argv[]);
  - int execvp(const char *file, char *const argv[]);
  - int execvpe(const char *file, char *const argv[], char *const envp[]);

- You can visit the man page of exec(3) for more information about those system calls : https://man7.org/linux/man-pages/man3/exec.3.html

System calls: wait()

- The wait() syscall forces the parent to suspend its execution and wait for the children process/es to finish its execution (or to be terminated e.g. by a signal)

- When a child process terminates, it returns an exit status to the OS, which is returned to the parent process waiting. Upon this value is received by the parent, it continues the execution

- If a child process terminates but the parent never waited, then this child process becomes a **zombie** process and continues to exist in the process table entry.This might lead to unwanted behavior in our system (e.g. all PIDs are reserved by zombie processes and thus we can’t launch any new processes)

System calls: exit()

- This call gracefully terminates the process of execution, meaning that it cleans up and releases resources taken by this process.

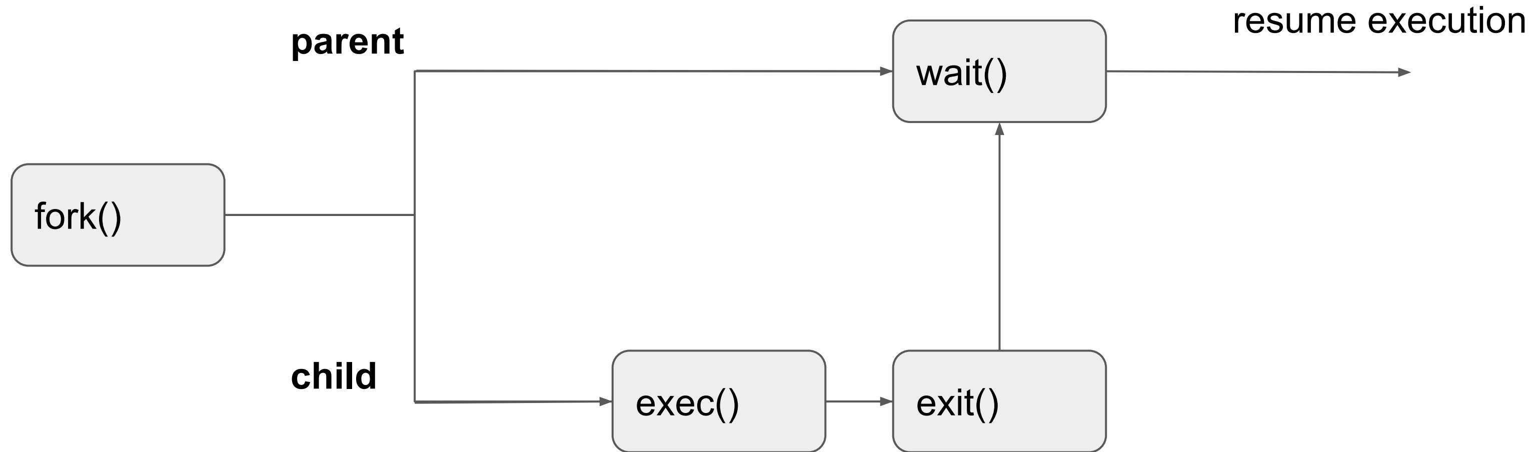
- When a child process is terminated an exit() status is returned to the OS and a signal is being send to the parent.

- The exit status is captured by the parent via the wait() system call.If the parent waited and receives such signal then the child terminates (dies).If the parent wasn't waiting then the child process enters a **zombie state.**

Process state

- There are various states that a process can be found (running,sleeping,waiting etc).For our child process we need to manage the end of its execution so it does not enter one of the following states:
  - **Orphan** is a process whose parent has finished or terminated even though it remains running itself
  - **Daemon** is a process that runs in the background and is not being controlled over the user
  - **Zombie** is a process that has completed its execution but still has an entry at the process table.

Working together: fork(),wait(),exec()



<u>Assignment 1:</u> A C shell implementation

- You are asked to implement a shell that can reads and executes commands provided by the user

- Your shell should be named **cs345sh**

- The command prompt should be **<user>@cs345sh/<dir>**, where:
  - **<user>** is the currently logged in user name (use the getlogin() function)
  - **<dir>** is the current working directory path (use the getcwd() function)

Assignment 1: Simple commands

- Some examples of simple commands are:
  - ls
  - exit
  - cd
- Some more complicated examples of commands that should be supported are:
  - ls -l
  - cat file.txt

Assignment 1: Signal Handling

- Signals are **standardized** messages sent to a running process to trigger a specific behavior (e.g. termination of execution)
- Signals can originate from a user or another process.
- Each signal has a **signal number** and a **signal handler**
  - **signal number:** A symbolic name (number) that characterizes a signal
  - **signal handler:** A function that is being called when a signal occurs and handles it properly
- You have to register the signal handler using the signal() function

Assignment 1: Signal Handling

- An example below depicts a simple signal handler for the CTRL C signal.

```c
1   #include <unistd.h>
2   #include <signal.h>
3   #include <stdio.h>
4   #include <stdlib.h>
5
6   // Define the signal handler for the ctrl-c (SIGINT) signal
7   void signal_callback_handler(int signum) {
8       printf("Caught signal %d\n",signum);// Print the signal number that was caught
9       exit(signum);// Terminate program
10  }
11
12  int main(){
13      // Register signal and signal handler
14      signal(SIGINT, signal_callback_handler);
15
16      while(1){ // infinate loop
17          printf("Program processing...\n");
18          sleep(1);
19      }
20
21      return 0;
22  }
```

<u>Assignment 1:</u> Signal Handling

- You are asked to implement the functionality that handles the signals occurred by the following actions:
  - CTRL - S
  - CTRL - Q
  - CTRL - Z
  - fg
- In order to test that the control flow commands you have to disable the default handling of them (within your shell) by using the command
  - stty -ixon -ixoff

<u>Assignment 1:</u> Global Variables

- A shell variable is a variable that is available only to the current shell

- You can declare variables to a shell by simply using the equal (‘=’) operator.
  - The left side of the command is the name of the variable
    - The name can contain only alphanumeric characters (a-zA-z)and the underscore character (‘_’)
  - The right side of the command is the value of the variable
    - The value can be anything (string,integer,float etc)

- An example of declaring such variable is
  - **my_shell$ my_var=”Hello World!”**
  - **Where**
    - **variable name: my_var**
    - **variable value: “Hello World”**

- You can access the **value** of a variable by using the **"$"** character.
    - e.g. to print a variable from the bash prompt :
        - **my_shell$ echo $my_var**
        - **Output: Hello World**
- You are asked to implement the functionality of defining global variables accessible to all the shells and subshells
- Declaring shell variables to a global scope is being done through the **export** command.

Assignment 1: Global Variables

- An example of creating a global variable **my_var**

```
pbekos@pbekos:~/Desktop/345$ my_var="Hello World"
pbekos@pbekos:~/Desktop/345$ echo $my_var
Hello World
pbekos@pbekos:~/Desktop/345$ bash
pbekos@pbekos:~/Desktop/345$ echo $my_var

pbekos@pbekos:~/Desktop/345$ exit
exit
pbekos@pbekos:~/Desktop/345$ export my_var
pbekos@pbekos:~/Desktop/345$ bash
pbekos@pbekos:~/Desktop/345$ echo $my_var
Hello World
pbekos@pbekos:~/Desktop/345$
```

(1) Initializing local variable my_var

(2) Printing my_var

(3) Initializing a sub-shell

(4) Trying to print a local variable of another shell (not accessible thus empty)

(5) Exiting sub-shell

(6) Exporting my_var to global scope

(7) Initializing a sub-shell

(8) Print the global variable my_var from the sub-shell

Assignment 1: Pipelines

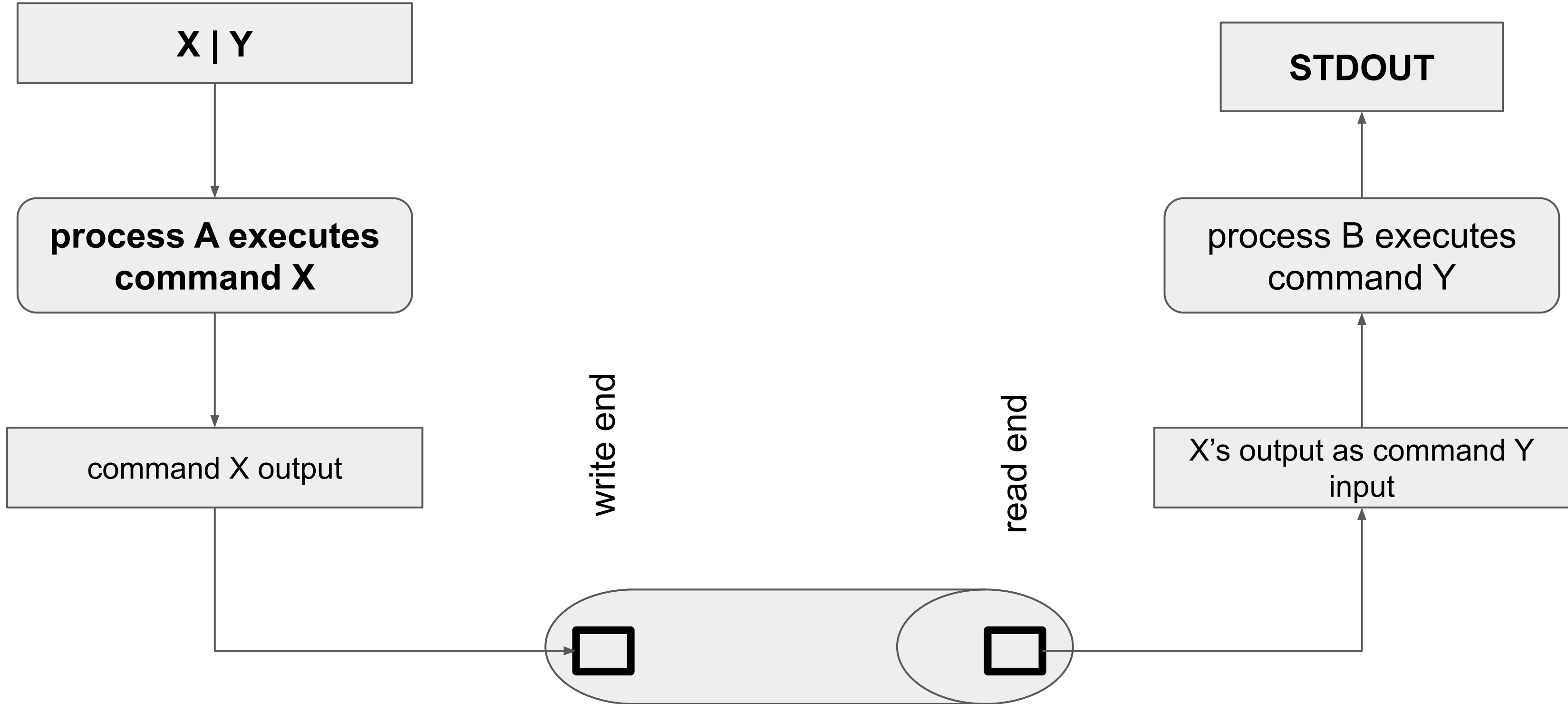
- Pipelines (pipes) provide a unidirectional interprocess communication channel.
- The operator **“|”** between two commands, directs the standard output of the left to the standard input of the right.
  - In a simpler manner, whatever is the output of the left command is used as the input argument of the right command.
  - A pipe is a “channel” which has 2 ends:
    - one for writing data on this channel
    - one for reading data from this channel
- You are asked to implement **simple** pipes : **command 1 | command 2**
  - **example : echo “Hello world” | wc -w**

- And **multiple** pipes: **command 1 | command 2 | … | command n**
  - **example : cat test.txt | sort | head -2**

Assignment 1: Pipelines

- Let’s describe the workflow of a simple pipe command:
  - **echo “Hello world” | wc -w**
- The first command is supposed to print **Hello world** to the prompt
- The second statement is supposed to count the amount of words of the left command
- So in the end this will print **2**
- **How this works:**
  - One process (commonly the child) handles the execution of the first command
  - Instead of writing the result on the STDOUT, it writes it to the **write** end of a pipe
  - Then a second process, receives as **input** the data that were written in the pipe through the **read** end.
  - Having the aforementioned data as input, then computes the second command on those and prints the outcome to the STDOUT.

Assignment 1: Pipelines



Working with QEMU: Setting up the environment

- Qemu is already installed in the department’s computers.

- You have to copy the disk image from the course’s directory to one of your own local directories.

- The disk image can be found here : **~hy345/qemu-linux/hy345-development.img**
  - To copy the image execute:
    - **cp ~hy345/qemu-linux/hy345-development.img <your_dir_path>**

- **<your_dir_path>** is the path to the directory you want to place the disk image

Working with QEMU: Booting up QEMU

- Now that you have the disk image locally you can boot it up:
  - To launch qemu run:
    - **qemu-system-i386 -hda hy345-development.img -curses**
  - The -curses parameter dictates that the Virtual machine will run without graphics
  - If you are working remotely using this argument in the boot up is recommended (if not necessary).
- To login as a user the credentials are:
  - user_name : **hy345**
  - password : **hy345**

Working with QEMU:

- An emulator mimics the properties of a system to run in another platform efficiently
  - It might bring some additional overhead but:
    - It is inexpensive
    - Easy to access
    - Helps us run programs that might be obsolete to the available system
- You are asked to use the QEMU emulator in order to utilize the department’s computers safely amongst all students (e.g. prevent crashes)
- You should always **compile and run** your shell in the virtual environment of **qemu**.You can implement the code in the machine you prefer but the testing of your shell should not take place in that machine.

Working with QEMU:

- To transfer a file from your local directory to a directory in qemu:
  - **From within qemu** run : **scp username@10.0.2.2:<path>/test1.c <qemu-dir>**
  - **Where:**
    - **username** is your username (e.g. csd1234)
    - **<path>** is the path to your file, in your local machine
    - **<qemu_dir>** is the directory you wish to copy the file inside of qemu
- To transfer a file within qemu to your local machine:
  - **From within qemu** run : **scp test1.c username@10.0.2.2:~/<path>** (parameters same as the above in meaning).
- To exit qemu simply use the command **turnoff**

Useful sources:

- You can visit the documentation pages of the system calls.Some useful links are provided below:
  - https://man7.org/linux/man-pages/man2/fork.2.html
  - https://man7.org/linux/man-pages/man3/exec.3.html
  - https://man7.org/linux/man-pages/man2/wait.2.html
  - https://man7.org/linux/man-pages/man2/chdir.2.html
  - https://www.tutorialspoint.com/c_standard_library/c_function_fopen.htm
  - https://www.geeksforgeeks.org/fork-system-call/
  - https://www.geeksforgeeks.org/zombie-processes-prevention/
  - https://winscp.net/download/WinSCP-5.19.3-Setup.exe

Questions?