HY-345 Assignment 1

Φροντιστηριο: 22/10
Deadline: 05/11

Assignment 1: A C shell implementation

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

- 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 -l
  - exit
  - cd
  - cat
  - mkdir
- Moreover you have to implement the execution of sequences of commands like:
  - ls ; pwd ; whoami

Assignment 1: Redirections

- Your shell should support redirection.Those operators change the way from where a command reads its input to where the command writes its output
- Some examples:
  - **cat < data.txt :** cat will read its input from the file data.txt
  - **ls > data.txt :** the output of the ls command will be written in the files.txt file replacing previous contents
  - **ls >> data.txt :** the output of the ls command will be appended at the end of the files.txt file.

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.

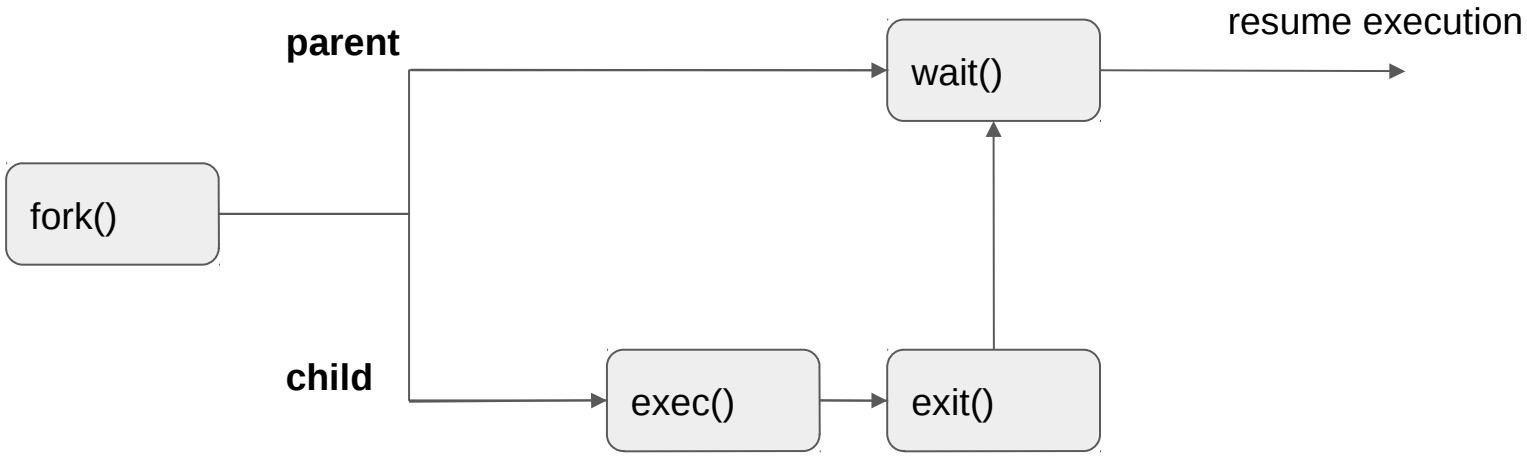
- **Simple** pipes : **command 1 | command 2**
  - **example : echo “Hello world” | wc -w**

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

Assignment 1: Pipelines

```
int pipefd[2];
pid_t cpid;
char buf;
if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
}
cpid = fork();
if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
}
if (cpid == 0) {                        /* Child reads from pipe        */
        close(pipefd[1]);               /* Close unused write end       */
        while (read(pipefd[0], &buf, 1) > 0)
        write(STDOUT_FILENO, &buf, 1);
        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        exit(EXIT_SUCCESS);
} else {                                /* Parent writes argv[1] to pipe */
        close(pipefd[0]);               /* Close unused read end        */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]);               /* Reader will see EOF          */
        wait(NULL);                     /* Wait for child               */
        exit(EXIT_SUCCESS);
}
```

Shell execution overview



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: 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)
- 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)

System calls: fork() example

```
// Forking a child
pid_t pid = fork();

if (pid == -1) {
    printf("\nFailed forking child..");
    return;
} else if (pid == 0) {
    if (execvp(parsed[0], parsed) < 0) {
        printf("\nCould not execute command..");
    }
    exit(0);
} else {
    // waiting for child to terminate
    wait(NULL);
    return;
}
```

- **!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, to continue its execution.

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 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: Setting up QEMU

- Qemu is installed to the department’s machines as a virtual disk image

- You have to copy this image to your working directory using the following command:
  - **cp ~hy345/qemu-linux/hy345-linux.img ~/<your_directory>**

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-linux.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 : **user**
  - password : **csd-hy345**
- To login as root the credentials are:
  - user_name : **root**
  - password : **hy345**

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 :
  - Hit **ALT + 2**
  - and then type : **“quit”**

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/pipe.2.html
  - https://man7.org/linux/man-pages/man2/dup.2.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?