Computer Engineering Department – ITU
CE301L: Operating Systems Lab

Course Instructor: Dr. Rehan Ahmed	**Dated:**
Lab Engineer: Muhammad Umair Shoaib	**Semester: Fall 2023**
Batch: BSCE21	

Lab 5 – OS Shell Utility: Basic Structure Implementation with Built-in Commands

Name	**Roll number**	**Total (out of 35)**

Checked on: _____

Signature: _____

5 OS Shell Utility: Basic Structure Implementation with Built-in Commands

Introduction

In this lab, we will build a simple Unix shell. The shell is the heart of the command-line interface, and thus is central to the Unix/C programming environment. Mastering the use of the shell is necessary to become proficient in the Unix programming world; knowing how the shell itself is built is the focus of this lab.

Objectives

The following are the key objectives of this lab:

- To further familiarize yourself with the Linux programming environment
- To learn how processes are created, destroyed, and managed
- To gain exposure to the necessary functionality in shells
- Implement built-in commands and parallel commands in shell

Theory and background

5.1 Unix system calls

To perform the lab tasks, you will need to be familiar with a few Unix *system calls* required for *process creation* and *execution* of commands; these are: `fork()`, `wait()` and `exec()`. These three functions are discussed here briefly.

5.1.1 The fork() system call

The `fork()` system call is used to create a new *process*. Let's understand `fork` by looking at the following example:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
printf("hello world (pid:%d)\n", (int) getpid());
int rc = fork();
if (rc < 0) {
// fork failed
fprintf(stderr, "fork failed\n");
exit(1);
} else if (rc == 0) {
// child (new process)
printf("hello, I am child (pid:%d)\n", (int) getpid());
} else {
// parent goes down this path (main)
printf("hello, I am parent of %d (pid:%d)\n",
rc, (int) getpid());
}
return 0;
}
```

When you run this program (called `p1.c`), you'll see the following:

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

Let us understand what happened in more detail in `p1.c`. When it first started running, the process prints out a hello world message; included in that message is its *process identifier*, also known as a *PID*. The process has a PID of 29146; in UNIX systems, the PID is used to name the process if one wants to do something with the process, such as (for example) stop it from running.

Then, the process calls the `fork()` system call, which the OS provides as a way to create a new process. The process that is created is an (almost) exact copy of the calling process. That means that to the OS, it now looks like there are two copies of the program p1 running, and both are about to return from the `fork()` system call. The newly-created process (called the *child*, in contrast to the creating *parent*) doesn't start running at `main()`, like you might expect (note, the "hello, world" message only got printed out once); rather, it just comes into life as if it had called `fork()` itself.

You might have noticed: the child isn't an exact copy. Specifically, although it now has its own copy of the address space (i.e., its own private memory), its own registers, its own PC, and so forth, the value it returns to the caller of `fork()` is different. Specifically, while the parent receives the PID of the newly-created child, the child receives a return code of *zero*. This differentiation is useful, because it is simple then to write the code that handles the two different cases (as above).

You might also have noticed: the output (of `p1.c`) is not *deterministic*. When the child process is created, there are now two active processes in the system that we care about: the parent and the child. Assuming we are running on a system with a single CPU (for simplicity), then either the child or the parent might run at that point. In our example (above), the parent did and thus printed out its message first. In other cases, the opposite might happen, as we show in this output trace:

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

The CPU scheduler determines which process runs at a given moment in time; because the scheduler is complex, we cannot usually make strong assumptions about what it will choose to do, and hence which process will run first.

5.1.2 The wait() system call

In the example code of `fork()`, we just created a child that prints out a message and exits. Sometimes, as it turns out, it is quite useful for a parent to *wait* for a child process to finish what it has been doing. This task is accomplished with the `wait()` system call (or its more complete sibling `waitpid()`); have a look at the following code (`p2.c`):

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(int argc, char *argv[]) {
printf("hello world (pid:%d)\n", (int) getpid());
int rc = fork();
if (rc < 0) { // fork failed; exit
fprintf(stderr, "fork failed\n");
exit(1);
} else if (rc == 0) { // child (new process)
printf("hello, I am child (pid:%d)\n", (int) getpid());
} else { // parent goes down this path (main)
int rc_wait = wait(NULL);
printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
rc, rc_wait, (int) getpid());
}
return 0;
}
```

In this code, the parent process calls `wait()` to delay its execution until the child finishes executing. When the child is done, `wait()` returns to the parent. Adding a `wait()` call to the code above makes the output deterministic. Here is the output:

```
prompt> ./p2
```

```
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (rc_wait:29267) (pid:29266)
prompt>
```

With this code, we now know that the child will always print first. Why do we know that? Well, it might simply run first, as before, and thus print before the parent. However, if the parent does happen to run first, it will immediately call `wait()`; this system call won't return until the child has run and exited. Thus, even when the parent runs first, it politely waits for the child to finish running, then `wait()` returns, and then the parent prints its message.

5.1.3 The exec() system call

A final and important piece of the process creation API is the `exec()` system call. This system call is useful when you want to run a program that is different from the calling program. For example, calling `fork()` in `p2.c` is only useful if you want to keep running copies of the same program. However, often you want to run a different program; `exec()` does just that. See the following code (let's call it `p3.c`)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
int main(int argc, char *argv[]) {
printf("hello world (pid:%d)\n", (int) getpid());
int rc = fork();
if (rc < 0) { // fork failed; exit
fprintf(stderr, "fork failed\n");
exit(1);
} else if (rc == 0) { // child (new process)
printf("hello, I am child (pid:%d)\n", (int) getpid());
char *myargs[3];
myargs[0] = strdup("wc"); // program: "wc" (word count)
myargs[1] = strdup("p3.c"); // argument: file to count
myargs[2] = NULL; // marks end of array
execvp(myargs[0], myargs); // runs word count
printf("this shouldn't print out");
} else { // parent goes down this path (main)
int rc_wait = wait(NULL);
printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
rc, rc_wait, (int) getpid());
}
return 0;
}
```

In this example, the child process calls `execvp()` in order to run the program `wc`, which is the *word counting program*. In fact, it runs `wc` on the source file `p3.c`, thus telling us how many lines, words, and bytes are found in the file:

```
 prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
29 107 1030 p3.c
hello, I am parent of 29384 (rc_wait:29384) (pid:29383)
prompt>
```

Given the name of an executable (e.g., `wc`), and some arguments (e.g., `p3.c`), `exec` system call loads code (and *static data*) from that executable and overwrites its current code segment (and current static data) with it; the heap and stack and other parts of the memory space of the program are re-initialized. Then the OS simply runs that program, passing in any arguments as the `argv` of that process. Thus, it does not create a new process; rather, it transforms the currently running program (formerly `p3`) into a different running program (`wc`). After the `exec()` in the child, it is almost as if `p3.c` never ran; a successful call to `exec()` never returns.

5.2 Shell specifications for this lab

In the lab tasks, you will implement a *command line interpreter (CLI)* or, as it is more commonly known, a *shell*. The shell should operate in this basic way: when you type in a *command* (in response to its *prompt*), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished. The shell you implement will be similar to, but simpler than, the one you have been using in Linux in the previous labs; the *bash shell*.

5.2.1 Basic shell: itush

Your basic shell, called *itush* (short for *ITU Shell*), is basically an *interactive* loop: it repeatedly prints a prompt `itush>` (note that there is a space after the greater-than sign), parses the input, executes the command specified on that line of input, and waits for the command to finish. This is repeated until the user types exit. The name of your final executable should be `itush`.

The shell can be invoked with no argument; anything else is an error. Here is how it should appear when the executable is called:

```
prompt> ./itush
itush>
```

At this point, `itush` is running, and ready to accept commands. This mode is called *interactive mode*, and allows the user to type commands directly. In next lab, we will add to our shell the functionality to accept *batch files*.

The shell is very simple (conceptually): it runs in a while loop, repeatedly asking for input to tell it what command to execute. It then executes that command. The loop continues indefinitely, until the user types the built-in command `exit`, at which point it exits.

For reading lines of input, you should use `getline()`. This allows you to obtain arbitrarily long input lines with ease. Generally, the shell will be run in interactive mode, where the user types a command (one at a time) and the shell acts on it.

To *parse* the input line into constituent pieces, you might want to use `strsep()`. Read the `man` page (carefully) for more details.

To execute commands, you will need to use `fork()`, `exec()`, and `wait()`/`waitpid()`. See the `man` pages for these functions, and also read the brief overview given above. You will note that there are a variety of commands in the `exec` family; for this project, you must use `execv`. Remember that if `execv()` is successful, it will not return; if it does return, there was an error (e.g., the command does not exist). The most challenging part will be getting the arguments correctly specified.

5.2.2 Built-in commands

In this lab, you should implement `exit`, `cd`, and `path` as built-in commands.

- `exit`: When the user types `exit`, your shell should simply call the `exit` system call with `0` as a parameter. It is an error to pass any arguments to `exit`.
- `cd`: `cd` always takes one argument (0 or >1 arguments should be signaled as an error). To change directories, use the `chdir()` system call with the argument supplied by the user; if `chdir` fails, that is also an error.
- `path`: The `path` command takes `0` or more arguments, with each argument separated by *whitespace* from the others. A typical usage would be like this: `itush> path /bin /usr/bin`, which would add `/bin` and `/usr/bin` to the search path of the shell. If the user sets path to be empty, then the shell should not be able to run any programs (except *built-in commands*). The `path` command always overwrites the old path with the newly specified path. In the next lab we will use this command to specify the path of Linux commands.

Whenever your shell accepts a command, it should check whether the command is a built-in command or not. If it is not, it should give an error, saying “command not found”. Linux commands will be implemented in the next lab. If it is a built-in command, your shell will invoke your implementation of the built-in command. For example, to implement the `exit` built-in command, you simply call `exit(0)`; in your `itush` source code, which then will exit the shell.

Lab tasks

5.3 Task 1: Working of the shell [Marks: 8]

Write your `itush.c` program such that it is able to prompt the user to give command, and it can print the command obtained from the user on the terminal. The shell should operate according to the specifications given in section 5.2.1.

Show working terminal output to the lab engineer and paste code and screenshot of terminal window in your report. [8]

5.4 Task 2: Built-in commands [Marks: 12]

Modify your program to implement the built-in commands of `exit`, `cd` and `path` as described in section 5.2.2.

Show working to the lab engineer and paste code and screenshots of output in all use cases of use of the built-in commands in your report. [12]

5.5 Analysis [Marks: 5]

1. What does the `fork` command return? [2]

2. What is the difference between the `exec` variants, `execl` and `execv`? [1]

3. What does the C function `chdir` do? [2]

The following table will be filled by the lab engineer during grading:

Task	Max marks	Obtained marks	Remarks if any
1	8		
2	12		
Analysis	5		

Assessment rubric for Lab 5

Performance	**Lab tasks**	**Exceeds expectation (4 – 5)**	**Meets expectation (2 – 3)**	**Does not meet expectation (0 – 1)**	**Marks**
1. Realization of experiment	1, 2	Conceptually understands the topic under study and develops the experimental setup accordingly	Needs guidance to understand the purpose of the experiment and to develop the required setup	Incapable of understanding the purpose of the experiment and consequently fails to develop the required setup	
2. Conducting experiment	1, 2	Sets up the required software, writes and executes bash/C/C++ programs according to the requirement of task and examines the program output carefully	Needs assistance in setting up the software, makes minor errors in writing codes according to task requirements	Unable to set up the software and to write and execute program according to task requirements	
3. Data collection	1, 2	Interprets program output and completes data collection as required in the lab task, ensures that the data is entered in the lab report according to the specified instructions	Completes data collection with minor errors and enters data in lab report with slight deviations from provided guidelines	Fails at observing output states of experimental setup and collecting data, unable to fill the lab report properly	
4. Data analysis	-	Analyzes the data obtained from experiment thoroughly and accurately verifies it with theoretical understanding, accounts for any discrepancy in data from theory with sound explanation, where asked	Analyzes data with minor error and correlates it with theoretical values reasonably. Attempts to account for any discrepancy in data from theory	Unable to establish a relationship between practical and theoretical results and lacks in-depth understanding to justify the results or to explain any discrepancy in data	
5. Computer use	1, 2	Possesses sufficient hands-on ability to work with Linux OS, GNU toolchain and other relevant software	Is able to work on Linux OS with GNU toolchain or other relevant software with some assistance	Cannot operate the Linux OS and other software without significant assistance	
6. Teamwork	-	Actively engages and cooperates with other group members in an effective manner	Cooperates with other group members in a reasonable manner	Distracts or discourages other group members from conducting the experiments	
7. Lab safety and disciplinary roles	-	Observes lab safety rules; and adheres to the lab disciplinary guidelines aptly	Observes safety rules and disciplinary guidelines with minor deviations	Disregards lab safety and disciplinary rules	
				Total (out of 35)	