Operating Systems – Exercise 3

C/Unix Lab

General Guidelines:

- Submission deadline is **Wednesday, April 29th, 23:55**
- No late submission will be accepted!
- You should work on your exercise by yourself. Misconducts will be punished harshly.
- Submit your code as C/H files and a makefile, and the theoretical answers as a PDF
- Pack your files in a ZIP file named Ex3-StudentID.zip, for example: Ex3-012345678.pdf
- Submit the ZIP file through the submission page in course website
- Document your code
- Place your ID at the top of every source file
- **Note:** This exercise contains 2 parts (programming and theoretical)
- **Note:** This exercise includes a bonus question worth 15 points, however you **cannot** get over 100 points in this exercise!

NOTE: This exercise will be mainly graded automatically. Your code MUST work in order to get a high grade. If you know your code does not work, add a file named notes.txt to your submitted ZIP and write exactly what does not work. We will consider this and try to grade the non-working parts manually. If you do not do that and the code does not work you will not get any points.

Part 1 – UNIX Process Programming (60 points + 15 points bonus)

In this part you will implement a UNIX command interpreter (shell). A shell is the program that shows a command line and prompts for user's command. Once the user enters a command, the shell executes it.

A UNIX shell supports two execution modes: **foreground** and **background**. When running a command in foreground mode, the shell executes the command and waits for it to finish before letting the user enter a new command. In background mode, the shell immediately lets the user execute a new command, while the old command is still running in background.

To run a command in background mode on a UNIX shell, add an ampersand (`&`) symbol at the end of the line, for example: `shell> some_command &`
(You can try this in the standard UNIX shell to understand the required behavior)

Another useful feature of a UNIX shell is the ability to run **several consecutive commands**, one after each other, in a single line. This is done by placing the `&&` operator between two commands. For example:

```
shell> some_command1 arg1 arg2 && some_command2 arg3 arg4
```

In this example, `some_command2` will be executed after `some_command1` is done.
Your shell should also support this feature.

A. Write a basic UNIX shell – myshell.c

Your shell should start by showing some prompt as you would like (see also section B), and read commands from user. User commands may (and may not) have an ampersand character at the end of the line (to run processes in background mode).
You should interpret the input and execute the requested command, with the requested arguments, in the requested mode.

If user command execution has failed (because it was not found or any other reason) you should display an informative message.
Your shell should exit upon receiving an "`exit`" command.

Guidelines

- Use the `fork()` function to split your process into two. Then, use its return value to separate your code between the 'parent' and the 'child' processes. To use `fork()` you should include `unistd.h` and `sys/types.h`.
- `fork()` return value is of type `pid_t` and actually contains a numeric value. After invoking `fork()` there are two running processes with the same (duplicated) memory and they start to run from the next command after `fork()`. The only thing that is different is the return value of `fork()` - for one process (the 'child') it's a zero and for the other process (the 'parent') it's the ID of the 'child' process.
- Remember that in UNIX there is a clear separation between a process and an application. Your new process should explicitly execute the application you want to run.
  Use `execvp(char* path, char* argv[])` to execute an application. `path` is a full path to the application and `argv` is a NULL terminated array of arguments. This command never returns <u>unless it fails</u>. It replaces current process code and data with the requested program and executes it.
  Note that if `execvp()` fails, the process continues to execute the next commands after it.
- Remember that in UNIX, the arguments array also contains the command itself as the first element of it.
- If you would like a 'parent' process to wait for a 'child' process, use the `waitpid(pid_t pid, int* status, int options)` function. It blocks until the process whose ID is `pid` finishes and then you can retrieve its exit code using: `WEXITSTATUS(status)`.
- Remember that the `&` symbol used for background execution, and the `&&` operator, are NOT arguments to the program. You should NOT pass them as arguments to the program!
- You may use `strtok()` to split user input and other C string functions if you need to use them. You may go over the material from the "Programming Workshop" course or other courses you have done in C/UNIX.
- You may assume that a full command line does not exceed 256 characters (including the line break and null terminator).
- You may assume that at most 16 commands are executed using a single command line (when using the `&&` operator)
- You may assume that command line arguments (and program names/paths) do not contain spaces. That is, you may always assume that a space is a valid delimiter.
- You may assume that when using the `&&` operator, only the last command may have to be ran in background (meaning, the `&` symbol may only appear at the end of the command line).

Recommended Steps:

1. Write a function that reads user input and parses it to an array of arrays of tokens (strings). Array of tokens corresponds to a separate command to be executed.
2. Write a function that given an array of tokens (which contains a command and its argument), and a flag that indicates if a command should be ran in background, executes the given command in the desired mode.
3. Write a main function that calls the above functions over and over, until an "`exit`" command is given.

B. Support directory change

Your current shell only works from the directory where it was executed. It does not support the '`cd`' command and therefore user cannot really traverse system directory tree.

In this extension you will add the ability to do that:

- If the user command is '`cd`' (with some argument), you should call the `chdir()` function with the argument given by user as its argument.
- You should change your prompt to show the current working directory. You can show it however you would like but the full path must be displayed. To get the full path, use the `getcwd(0,0)` function call. Example prompt:
  `/home/user>`

C. Test your shell

Compile the two test programs: `test` and `sleeper`, using the attached makefile.
You can extend the makefile as explained in Appendix 1 so it will also be able to compile your shell. Compile your shell and run it to test it.

You are provided with two simple test programs. Use them to test your shell:

1. `test.c` – a program that prints how many command line arguments it received and displays these arguments.
2. `sleeper.c` – a program that prints the numbers from 0 to 9 with a delay of a second between them. Use this program to test the background execution capabilities of your shell.

Run these programs from your shell's command line, optionally with some command line arguments for `test`. You may also run them in a regular UNIX shell to compare behaviors.
You may also use internal UNIX commands to test your shell, such as `ls` (directory listing) and `netstat` (network status).
Important: note that like in bash, to run programs in current directory you should use the '`./`' prefix. For example, if you want to run the program `test` from the current directory you should type:
`/home> ./test` (your prompt may look different than it looks here)
You are advised to deeply test your shell behavior and get into end cases. Your shell is expected to work just as a regular UNIX shell. You may compare the behavior of your shell to the behavior of the regular UNIX shell in your system.
See an example run of the shell in the next page.

D. Bonus – Add the *pipe* feature to your shell (Up to 15 points)
Note: this bonus will only receive points if it works properly.

If you decide to implement the bonus, make your shell <u>display a clear message</u> stating that it was implemented and should be graded.

Modern UNIX shells also support the *pipe* feature when running more than one commands in a single command line. Piping two commands makes the shell send the standard output of the first command as the standard input of the second command (and the output of the second command can also be piped to a third one, and so on…).

<u>The bonus task:</u> make your shell support the pipe feature using the `|` operator.

<u>For example:</u>

```
shell> ./test 1 2 3 | head -n 3
```

Will output to the screen:

```
Found 4 arguments
Arguments:
1
```

And when using more than one pipe:
`shell> ./test 1 2 3 | head -n 3 | tail -n 2`
Will output to the screen:
```
Arguments:
1
```

Guidelines for Part D:

- Use the `pipe()` function to create a pipe. A pipe is a unidirectional channel for data transfer between processes. It is based on a file that is being shared among processes. Calling pipe creates such a file and provides you with two file descriptors: one for reading from the pipe, and another for writing into the pipe. Note that these are file descriptors and not file pointers (integers, not `FILE*`).
- When calling `fork()`, the pipe remains opened by both processes. You should use this fact to make them communicate correctly. Remember to close the unused endpoints of the pipe in each process (both pipe endpoints should be closed in both processes!).
- To replace a process's `stdin` file, close it by calling `close(0)`, and then call the `dup(...)` function with the appropriate (read) endpoint of the pipe. In this case, the read endpoint should not be closed in this process.
- To replace a process's `stdout` file, close it by calling `close(1)`, and then call the `dup(...)` function with the appropriate (write) endpoint of the pipe. In this case, the write endpoint should not be closed in this process.
- You can assume that if a line includes a pipe, nothing will be executed in background mode.
- Use the following guide to further learn about pipes: http://beej.us/guide/bgipc/output/html/multipage/index.html

Example of a shell session:
(Highlighted text is the output of the background execution of sleeper. Red text is standard error output)

- Try to avoid copy-paste the commands, both word and acrobat change the '.' and '/' signs and more.
  You should type everything

```
/> ls
bin    dev   initrd.img      lost+found  opt   sbin   sys  var      boot   etc
initrd.img.old
media   proc  selinux      tmp  vmlinuz cdrom  home  lib  mnt       root  srv
usr  vmlinuz.old
/> cd home/omer
/home/omer> ls
Desktop    Downloads            Music     Public      sleeper.c  test    Videos
Documents  examples.desktop  Pictures  sleeper Templates  test.c  workspace
/home/omer> ./test a b
Found 3 arguments
Arguments:
./test
a
b
/home/omer> ./sleeper
1
2
3
4
5
6
7
8
9
10
/home/omer> ./foo
Error: cannot start program ./foo
```

```
/home/omer> ./test hello
Found 2 arguments
Arguments:
./test
hello
/home/omer> ./sleeper &
1
2
/home/omer> 3
./test foo
4
5
6
7
8
Found 2 arguments
Arguments:
./test
9
foo
/home/omer>
10
./test x y
Found 3 arguments
Arguments:
./test
x
y
/home/omer> cd Documents
/home/omer/Documents> cd ..
/home/omer> cd Desktop
/home/omer/Desktop> cd ..
/home/omer> ls
Desktop    Downloads            Music     Public      sleeper.c  test    Videos
Documents  examples.desktop  Pictures  sleeper Templates  test.c  workspace
/home/omer> ./sleeper && ./test foo
1
2
3
4
5
6
7
8
9
10
Found 2 arguments
Arguments:
./test
foo
/home/omer> ./test 1 | cat
Found 2 arguments
Arguments:
./test
1
/home/omer> ./test 1 | tail -n 2 | head -n 1
./test
/home/omer> exit
Thank you for using my shell!
```

Many other outputs are possible here - text might have messed up much more…

Bonus

Bonus

Part 2 – Theoretical (40 points)

A. (15 points)

The `swap(int *a, int *b)` operation performs a swapping of the values in two memory addresses `a` and `b`, in a single atomic operation. We are going to use the method `swap()` in the code below to try and solve the mutual exclusion problem for any number of threads:

```
// Shared variable:
int lock = 0;

// Local variables:
int value;
int id; // initialized as the number of current thread

// Entry code:
do {
    value = id;
    swap(&lock, &value);
} while (value != 0 && value != id);

// Critical Section

// Exit code:
lock = 0;
```

Prove (formally) or disprove (by a counterexample) each of the following statements:

i. The algorithm satisfies mutual exclusion

ii. The algorithm satisfies deadlock freedom

iii. The algorithm satisfies starvation freedom

B. (9 points)

For each of the following statements, choose whether it is true or false and explain:

A. An algorithm that satisfies deadlock freedom will satisfy starvation freedom as well
True / False

B. Running a single threaded program on a single core will always yield better performance compared to running a multi-threaded program on a single core (with the same logic)
True / False

C. If a thread is running at its critical section of the code (meaning that it has passed the entry code of the mutual exclusion algorithm/utility), it cannot be context switched (that is, it will stay in 'Running' state until it finishes the critical section)
True / False

C. (16 points)

We would like to schedule the run of five processes on a single system as follows:

1. P1 starts running
2. When P1 finishes running, P2 starts running
3. When P2 finishes running, P3 and P4 start running (simultaneously)
4. When P3 finishes running, P5 starts running (P4 may still be running at this time)
5. When both P4 and P5 finish, P1 starts running again (from step 1 above)

Using only semaphores as synchronization tools, build the synchronization system for the five processes:

- List the semaphores you are using and their initial values. You may either use binary or counting semaphores, just state the type of each semaphore.
- Write the pseudo code for each process.
- You can use as much semaphores as you would like, but your code should be efficient.

Example:
I use one binary semaphore named S1, initial value = 1
Process 1:

```
while (true) {
    down(S1)
    // run process 1
    up(S1)
}
```

(Of course, you have to specify the pseudo code for each process of the five and use more than one semaphore)