This lab will introduce you to the `wait()` system call discussed in our class discussions.

The wait() system call:

It blocks the calling process until one of its child processes exits or a signal is received. wait() takes the address of an integer variable and returns the process ID of the completed process. Some flags that indicate the completion status of the child process are passed back with the integer pointer. One of the main purposes of wait() is to wait for completion of child processes.

The execution of wait() could have two possible situations.

1. If there are at least one child processes running when the call to wait() is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.

2. If there is no child process running when the call to wait() is made, then this wait() has no effect at all. That is, it is as if no wait() is there.

Basically, wait(int *statusp) waits for any child process to complete. When it does, wait() returns the pid of the child process as the return value to wait(), and initializes statusp to contain information on how the child exited. You can use WIF macros in /usr/include/sys/wait.h to examine statusp.

Zombie Process:
When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls wait. The child process entry in the process table is therefore not freed up immediately. Although no longer active, the child process is still in the system because its exit code needs to be stored in case the parent subsequently calls wait. It becomes what is known as defunct, or a zombie process.

Orphan Process:
An orphan process is a process whose parent process has finished or terminated, though itself remains running. A process may also be intentionally orphaned so that it becomes detached from the user's session and left running in the background; usually to allow a long-running job to complete without further user attention, or to start an indefinitely running service. Under LINUX, the latter kinds of processes are typically called daemon processes. The LINUX nohup command is one means to accomplish this.

Daemon Process:
It is a process that runs in the background, rather than under the direct control of a user; they are usually initiated as background processes.

ps command:

The ps command shows the processes we're running; the process another user is running, or all the processes on the system. E.g.

$ ps –ef

By default, the ps program shows only processes that maintain a connection with a terminal, a console, a serial line, or a pseudo terminal. Other processes run without needing to communicate with a user on a terminal. These are typically system processes that LINUX uses to manage shared resources. We can use ps to see all such processes using the -e option and to get "full" information with -f. You could easily see all the processes beginning with the very first process that got created when you started / booted your system.

Also try out the output of the following commands.

$ ps x
$ ps aux

Few programs to review before you go with assignment programs ONE, TWO, and THREE.

Assume there are no errors (e.g. fork doesn't fail), but make no assumptions about how the processes are scheduled. Assume each call to printf flushes its output out just before it returns.

Examine the following program and circle *all* corresponding combinations of output that may be produced when it is run.

```
int main() {
   if (fork() != 0) {
wait(NULL);
printf("0");
   } else if (fork() == 0) {
printf("1");    }
     else { fork();
printf("2");    exit(0);    }
printf("3");    return 0;
}
```

a. 213032 b. 123023 c. 130322
d. 132203 e. 220133 f. 022133

Find out which of the above outputs are possible and which ones not possible. Ary there any possible outputs not shown above? Decide based on your understanding of fork and wait, and verify after running the above program multiple times.

For example, look at the programs fw.c, fwt2.c and fw3.c. They show examples of forking off a child, waiting for it to exit, and then examining statusp to see how it exited. They are all straightforward. In fw3, the child must be killed with a signal, using the command "kill". For example:

```
$ ./fw3 &
[1] 22326
$ Child (22327) doing nothing until you kill it
Kill the child with 'kill -9 22327' or just 'kill 22327'
```

Now, you can kill the child manually with "kill -9 22327", which sends it the "sure kill signal (signal number 9)", or with "kill 22327", which sends it signal 15. Try both:

```
$ ./fw3 &
[1] 22326
$ Child (22327) doing nothing until you kill it
Kill the child with 'kill -9 22327' or just 'kill 22327'
(hit return a few times)
$ kill -9 22327
$ Parent: Child done.
Return value: 22327
Status:         9
WIFSTOPPED:     0
WIFSIGNALED:    1
WIFEXITED:      0
WEXITSTATUS:    0
WTERMSIG:       9
WSTOPSIG:       0
```

Try again:

```
$ ./fw3 &
[1] 22328
$ Child (22329) doing nothing until you kill it
Kill the child with 'kill -9 22329' or just 'kill 22329'
$ kill 22329
$ Parent: Child done.
Return value: 22329
```

```
Status:         15
WIFSTOPPED:     0
WIFSIGNALED:    1
WIFEXITED:      0
WEXITSTATUS:    0
WTERMSIG:       15
WSTOPSIG:       0
```

fw3a.c has the child generate a segmentation violation, and you'll see that the parent can recognize this as the child is terminating with signal 11. We'll go over signals in detail in our upcoming lectures.

Now, look at fw4.c. What it does is have the child exit immediately, and have the parent wait 4 seconds, print out the output of the "ps aux" command, and then have it call wait(). It should be clear that by the time the parent calls system("ps aux"), the child has exited. Thus, we might expect there to be no listing in the "ps aux" command for the child, and possibly that the wait() might wait forever, since the child is completed. However, this is not the case.

When a child exits, its process becomes a "zombie" until its parent process either dies or calls wait() for it. By a "zombie", we mean that it takes up no resources, and doesn't run, but it is just being maintained by the OS so that when the parent calls wait(), it will get the proper information. Check and look at the output of fw4 on your machine.

```
$ ./fw4
```

What happens if the parent exits without calling wait()? Then the child zombie process should transfer parentage to /sbin/init. Instead, the child simply goes away. Check by reviewing the output of the fw4 program. Find out what do we mean by capital letters S, S+, R etc. in the output of ps x command. Does it indicate any information about zombie processes?

wait() returns whenever a child exits. If a process has more than one child, then you can't force wait() to wait for a specific child. You simply get whichever child exits first. For example, see multiple-child-proc.c. This program forks off 4 child processes and then calls wait() four times. The child processes sleep for a random period of time, and then they exit. As you see, the first wait() call returns the first child to return:

```
$ ./multiple-child-proc
Fork 0 returned 14160
Fork 1 returned 14161
Fork 2 returned 14162
```

```
Fork 3 returned 14163
Child 1 (14161) exiting
Wait returned 14161
Child 3 (14163) exiting
Wait returned 14163
Child 0 (14160) exiting
Wait returned 14160
Child 2 (14162) exiting
Wait returned 14162
```

Now, you can use waitpid() to wait for a specific process, and you can even have it return if the specified process has not exited.

Program ONE

Enter and compile the following program into an executable called proc-zombi.

```
/* proc-zombi.c */

#include <sys/types.h>
#include <unistd.h>
 int main() {
    if (fork() > 0) {    /* Line A */
        while (1);
}         return 0; }
```

In your shell, execute proc-zombi in the background (append your command with &):

```
./proc-zombi &
```

Now run ps to examine your processes:

```
ps  -f
```

You should see a process labeled as <defunct>. What does this mean? Explain how the program above would lead to a defunct process (also known as zombie process - recall our class discussion)

To see why such a process is called zombie process, issue the command kill to terminate the process, by passing in the process ID of the <defunct> process. For example, if the <defunct> process has process ID 1234, issue command: `kill 1234`

Run `ps -f` again to see if the defunct process is still there. To return to normal, kill the parent of the defunct process.

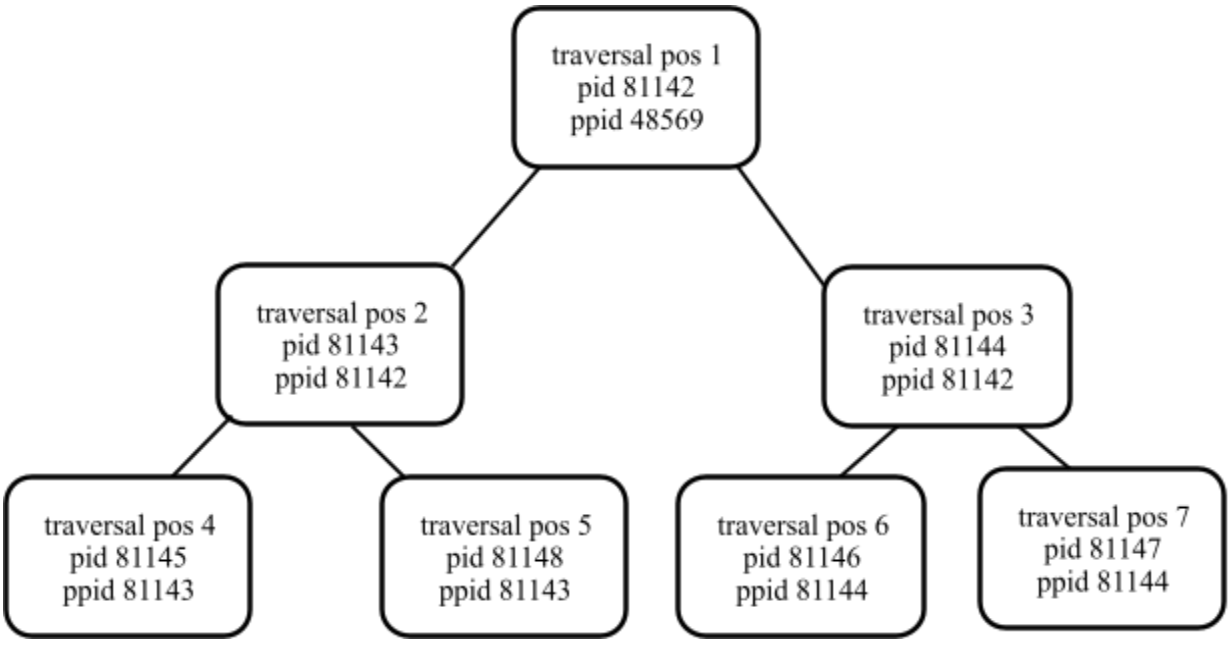
Make a copy of proc-zombi.c and name it proc-orphan.c. In proc-orphan.c, change the line labeled Line A to:

```
if (fork() == 0)  {
```

Compile and name your executable proc-orphan. In your shell, execute orphan in the background, and examine your processes just like before as in the case of the proc-zombi example. What is the name of the parent process of the process proc-orphan? Explain why the parent process is different from the parent process of the non-defunct process zombie in comparison to the proc-zombi example.

Program TWO

Write a program that creates a "binary tree" of processes. Call your program proctree.c. Your program should take a single command-line parameter which specifies the number of levels in the binary process tree. Each process should be assigned a number corresponding to its position in a level-order traversal of the tree. The tree can be thought of as this binary tree, where the parent-child links are not explicitly stored by your program but are part of the process hierarchy. If the depth is 3 levels, the tree should look something like this:



You won't draw the above shown visual / graphical representation, but your program's processes should print out the information about the tree, as follows:

```
 $ ./proctree 3
[1] pid 81142, ppid 48569
[1] pid 81142 created left child with pid 81143
[1]pid 81142 created right child with pid 81144
[2]pid 81143, ppid 81142
[3]pid 81144, ppid 81142
[2]pid 81143 created left child with pid 81145
[3]pid 81144 created left child with pid 81146
[3]pid 81144 created right child with pid 81147
[4]pid 81145, ppid 81143
```

```
[2]pid 81143 created right child with pid 81148
[6] pid 81146, ppid 81144
[7] pid 81147, ppid 81144
[5]pid 81148, ppid 81143
[3]right child 81147 of 81144 exited with status 7
[3]left child 81146 of 81144 exited with status 6
[2] right child 81148 of 81143 exited with status 5
[2] left child 81145 of 81143 exited with status 4
[1] right child 81144 of 81142 exited with status 3
[1] left child 81143 of 81142 exited with status 2
```

Note that each line of output is indented according to the depth of the node in the process tree and begins by printing the traversal position of the process that prints it. Your program's processes should produce output in the following situations:

- When each process is created, it should print its traversal position in [ ] brackets, its PID (process ID, obtained using `getpid())` and PPID (parent process ID, obtained using `getppid()).`
- After a process spawns a child process, it should print its own (not the new child's) traversal position, its own PID, and the PID of the newly-spawned child along with an indication of whether this child forms its "left" or "right" subtree.
- When a child exits (using `exit()`), it should provide its traversal position as its exit status. This value should be obtained by the parent when it calls `waitpid()` and printed along with the parent's traversal position, whether the terminated child is a left or right subtree, the parent's PID, the terminated child's PID and the exit status, which should be the child's traversal position.

This program, as you are developing it, has a good chance of becoming an infinite forking program, that is a program that keeps spawning new processes which can render a Linux system nearly useless (and will require a reboot). To reduce the chances that this happens, you should check the return value of your fork() calls and stop if it returns -1, which indicates that you were unable to spawn a process. You should also limit your trees to small heights (starting with 2 onwards) when debugging. Feel free to try larger tree sizes once you're confident that your program is working to see how large a tree you can get before you run out of processes. The number of maximum processes each one of you may be getting on running such a program may be different, as you may be having different resources (particularly the primary memory) on your computer systems.

Hint: Use recursion

Program THREE

Implement the C program in which the main program accepts the integers to be sorted. Main program uses the **`fork()`** system call to create a new process called a child process. Parent process sorts the integers using any sorting algorithm and waits for the child process using the `wait()` system call to sort the integers using any other sorting algorithm. Also demonstrate zombie and orphan states. **Also write an algorithm using suitable pseudo code for this problem statement mentioned above.**