Lab 3 and 4 - Multi Processing - fork(), waitpid(), execvp(), and system()

fork()

- A system call that creates a new child process
- The "parent" is the process that creates the other "child" process
- From then on, both processes are running the code after the fork
- The child process is identical to the parent, except:
  - it has a new Process ID (PID)
  - for the parent, fork() returns the PID of the child; for the child, fork() returns 0
  - fork() is called once, but returns twice

```
pid_t pidOrZero = fork();
// both parent and child run code here onwards
printf("This is printed by two processes.\n");
```

waitpid()

- A function that a parent can call to wait for its child to exit:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- pid: the PID of the child to wait on (we'll see other options later)
- status: where to put info about the child's termination (or NULL)
- options: optional flags to customize behavior (always 0 for now)
- the function returns when the specified child process exits
- the return value is the PID of the child that exited, or -1 on error (e.g. no child to wait on)
- If the child process has already exited, this returns immediately - otherwise, it blocks

execvp()

- **execvp()** is a function that lets us run another program in the current process.

```
int execvp(const char *path, char *argv[]);
```

- It runs the executable at the specified path, completely cannibalizing the current process.
  - If successful, execvp never returns in the calling process
  - If unsuccessful, execvp returns -1
- To run another executable, we must specify the (NULL-terminated) arguments to be passed into its main function, via the argv parameter.
- For our programs, path and argv[0] will be the same
- execvp has many variants (execle, execlp, and so forth. Type man execvp for more).

Programming Exercises

1. fork.cpp : This program illustrates the basics of fork. It has the clear flaw that the parent can finish before its child, and the child process isn't reaped by the parent.
2. fork_file_sharing.cpp : This program creates a child process where both the parent and child seemingly access or modify the same string at the same address. But the string is different for both the parent and child.
3. fork_random.cpp : An example that shows how a process and its child copy a random variable.
4. fork_output.cpp : Small example illustrating fork and the potential for many possible outcomes that result from the concurrency and the various ways all of the children, grandchildren, etc. can be scheduled.
5. waitpid.cpp : In this program a parent waits for its child to terminate.
6. waitpid_status.cpp : Here's is a program that's written in a style more conistent with how fork is normally used.
7. fork_exit_order.cpp : This example spawns off 8 children, each of which returns a different exit status. The main program waits until all child processes exit.
8. execvp.cpp : Demonstrates the build-in execvp() function.
9. simple-shell.cpp : A program containing an implementation of a working shell that can execute entered text commands. It relies on our own implementation of system(), called mysystem(), that creates a process to execute a given shell command.
10. system.cpp : Demonstrates the build-in system() function.