CSCB09 Week 6 Notes

1. **Forks:**
    - The fork system call creates a child process and a duplicate of the currently running program. Both processes run concurrently and independently. The child gets a new PID (Process ID) and PPID (Parent Process ID).
    - The fork function creates a new process by duplicating the calling process. The new process, called the child, is an exact duplicate of the calling process, referred to as parent, except for the following :
        1. The child has its own unique process ID, and this PID does not match the ID of any existing process group.
        2. The child's parent process ID is the same as the parent's process ID.
    - The return value from the fork call is different:
        1. On success:
            - fork() returns 0 to the child.
            - fork() returns a positive value, which is the child's PID, to the parent.
        2. On failure:
            - No child is created, fork returns a negative value, usually -1, to the parent, and errno is set appropriately.
    - The fork function takes no arguments.
      pid_t fork(void); is the syntax for the fork function.
    - You need to use **#include <unistd.h>**.
    - A child process terminates when the program's main function returns or the program calls exit.
    - In shell, you can use $? to get the exit status of the most recent command. The exit status is set by the exit function or main's return.
2. **Wait:**
    - Wait suspends execution of the calling process until one of its children terminates.
    - Syntax: **pid_t wait(int *status);**
    - After calling wait() a process will:
        - Block the calling process if all of its children are still running.
        - Return immediately with the PID of a terminated child, if there is a terminated child.
        - Return immediately with an error, -1, if it doesn't have any child processes.
    - Wait returns the pid of the terminated child or -1 on error.
    - **status** encodes the exit status of the child and how a child exited (normally or killed by signal).
    - There are macros to process exit status:
        1. **WIFEXITED** tells you if child terminated normally
        2. **WEXITSTATUS** gives you the exit status
    - A child becomes a zombie when it terminates, but its parent process is not waiting for it. The child's exit code is kept around as a zombie until parent

collects its exit code through wait or until parent terminates. Shows up as Z in ps.
- A child becomes an orphan if the parent process terminates before the child.
- Orphans get adopted by the init process.
- init is the first process started during booting. It's the root of the process hierarchy. init has a PID of 1 so the PPID of orphans is 1.
- Waitpid is used if a process wants to wait for a particular child rather than any child or if a process does not want to block when no child has terminated.
- Syntax: **pid_t waitpid(pid_t pid, int *status, int options);**
- First parameter specifies PID of child to wait for.
- If pid is -1 then it means any arbitrarily child. Here waitpid() work same as wait() work.
- If options is 0, waitpid blocks, just like wait.
- If options is WNOHANG, it immediately returns 0 instead of blocking when no terminated child.

3. **Exec:**
   - The exec family of functions replaces the current process with a new process. The new program starts executing from the beginning.
   - On success, exec never returns, on failure, exec returns -1.
   - The new process inherits from calling process:
     - PID and PPID, UID, GID
     - Controlling terminal
     - CWD, resource limits
     - Pending signals
   - Exec is not one specific function, but a family of functions.
     1. **execvp**:
        Syntax: int execvp (const char *file, char *const argv[]);
        file: Points to the file name associated with the file being executed.
        argv: Is a null terminated array of character pointers**.**

     2. **execlp**:
        Syntax: **int execlp(const char *file, const char *arg,.../* (char *) NULL */);**
        file: The file name associated with the file being executed.
        const char *arg and ellipses: Describes a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program.

     3. **execv**:
        Syntax: int execv(const char *path, char *const argv[]);
        path: A pointer that points to the path of the file being executed.
        argv[]: is a null terminated array of character pointers.

4. **execl**:
   Syntax: int execl(const char *path, const char *arg,.../* (char *) NULL */);
   file: The file name associated with the file being executed.
   const char *arg and ellipses: Describes a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program.

- First parameter: name of executable; then commandline parameters for executable; these are passed as argv[0], argv[1], …, to the main program of the executable.
- execl and execv differ from each other only in how the arguments for the new program are passed.
- execlp and execvp differ from execl and execv only in that you don't have to specify full path to new program.

4. **Difference Between Fork and Exec:**
   - Fork starts a new process which is a copy of the one that calls it while exec replaces the current process with a different one.
     I.e. Fork creates a duplicate of the current process while exec replaces the current process with a different one.
5. **How a shell runs commands:**
   - When a command is typed, shell forks and then execs the typed command.
6. **Processes and File Descriptors:**
   - File descriptors are handles to open files.
   - They belong to processes not programs.
   - They are a process's link to the outside world.
7. **Initializing Unix:**
   - Use the top or ps –aux command to see what's running.
   - The only way to create a new process is to duplicate an existing process. Therefore the ancestor of all processes is init with pid = 1.
   - The only way to run a program is with exec.