Lab 1 CS354 Answers Ley Yen Choo 0028729283

Problem 3.1

a. The source code of nulluser() is located in the file initialize.c.

b. The ancestor of all processes is located in the file named “process.h” and its name is NULLPROC and its value is 0.

c. The ancestor process will keep the CPU occupied when there are no processes that are ready to be run.

d. The nulluser() function will never return anything after being called by the code in start.S because there is a infinite while loop at the end of the function that does nothing.

e. The source code of halt() is located in the file named “intr.s” and it basically does nothing forever since it jumps to itself and return or loads IP with the value pointed to by SP.

f. When I removed nulluser() from start.S, XINU does not run as before, it was stuck at the loading XINU part.

g. When the while-loop at the end of nulluser() is replaced with a call to halt(), it acts the same as before since both of them does nothing forever.

Problem 3.2

a. When fork() is used, a new process (child process) is created, unlike create(), fork() does not take argument and it will return a process ID. After a child process is created, both parent and child processes will execute the next instruction following the fork() system call. When fork() is executed successfully, Linux will create two identical copies of address spaces, one is for the parent and the other one is for the child. Both of the processes will start their execution at the next statement after the fork() call. Since both processes have identical but separate address spaces, those variables initialized before the fork() system call will have the same values in both address spaces. Also, since every process has its own address space, any changes will be independent of the others. If the value of the variable in parent changes, the changes will only take effect in the parent process’s address space. Other address spaces created by fork() system calls will not be affected even though they have identical variable names.

b. From the result of the test code (test.c and test2.c) in system/, the process that always runs first is the parent process.

c. As an app programmer, I do not have a preference as to which process should run next in Linux since they still return value at the end, they can exit in either order and context switching will make the processes to be executed in random.

e. First of all, one of the fundamental differences between newProcess() and XINU's create() is newProcess() uses fork() and execve() while create() does not. In create(), it creates a new process that will begin its execution at location *void *funcaddr*, with a stack size of *uint32 ssize* bytes, process priority *pril6 priority*, and an identifying name *char* name*. If the creation is successful, the process id of the new process is returned to the caller. The process that is newly created is left in the suspended state (state of limbo) and it will not begin its execution until it is started by a resume command. In newProcess(), we only pass one argument into the parameter, which is the full pathname of an executable binary. When the function newProcess() is called, a system call to fork() is made, fork() will make two identical copies of address spaces, one for the parent and one for the child. Both parent and child will then execute the next statement following the fork() call. If the return value is negative then the creation of a child process was unsuccessful. If the return value is 0, then creation is successful and execve() is called. execve() executes the program pointed to by the *filename*, which is a binary executable. If the return value is positive then the parent will wait for the child process to end or stop until it continues doing its stuffs with *waitpid()*. Also, in create(), instead of copying the address space of the parent and create an identical copy address space for child, the newly created processes are stored in a process table. Other than that, newProcess() will execute immediately when it is called but create() will have to wait until the state of the process to be ready by calling resume() before it can be executed.

f. Clone() in Linux is similar to fork() but unlike fork(), it allows the child process to share parts of its execution context with the parent process, such as virtual address space, the table of file descriptors and the table of signal handlers. Also, clone() is used to implement threads. When the child process is created, it starts its execution by calling the function pointed to by its first argument unlike create() where its execution starts only after resume is called to change the suspended state of the child process to ready state.

g. Linux's posix_spawn() takes in 6 arguments while newProcess() only takes in 1 argument. posix_spawn() function creates a new process (child process) from a specified process image. The new process image shall be constructed from a regular executable file called the new process image file. Upon successful completion, posix_spawn() will return process ID of the child process to the parent process, in the variable pointed to by a non-NULL pid argument, it shall return zero as the function return value. Otherwise, no child process will be created. Quite similar to newProcess() which takes in the argument that specifies the full pathname of an executable binary , one of the posix_spawn()'s argument (*path*) is the pathname that identifies the new process image file to execute.

h. In my opinion, there is no best way to create process. It depends on the app or the app developer on how to create a process. Perhaps some app developers prefer to use fork(), some might prefer posix_spawn(). It also depends on the Operating system that

the developer is using, different Operating system has a different way of creating processes. Hence, there is no real best way to create a process.

Bonus problem
A new command: “lab1cmd” is added.