Chris Cohen
CS354 Spring 2020
Lab 1

3.1

a) In the file **include/process.h**, NPROC is defined as 8 (if not already defined). However, in **include/conf.h**, NPROC is defined as 100. In **include/xinu.h**, conf.h is included first, so the value of **NPROC will be 100**. When it reaches **include/process.h**, it won't be changed to 8 because NPROC is already defined. Therefore, 100 processes can exist simultaneously in XINU.

b) In the file **include/process.h**, NULLPROC is defined as 0, which is the process ID of the null process. In that same file, the process table struct is defined. In that struct, the priority field, **prprio**, has the C type of **pri16**, which is a custom type defined in **include/kernel.h**. From that file, we can see that pri16 is actually a 16 bit int data type. This means a process can have priority values ranging from **-32,768 to 32,767**. Since the null process has a priority value of 0 (found in **system/initialize.c**), any value lower than 0 is where the priority of the null process would fail.

c) << modification to XINU that runs main() from nulluser() directly >>

3.2 → system/mymotd.c and **include/mymotd.h**

3.3

a) **system/rcreate.c** and **include/rcreate.h**

b)

   i) In my test code, the parent and child processes have the same priority level. The convention for this sort of forking program is to have the parent use waitpid() and wait for the child process to finish executing. This prevents the child process from turning into a zombie process, also causing the child process to execute first. Without proper waiting for the child process, either one can finish executing first because they are concurrently scheduled. With my program, the parent process always finishes executing first.

      1) My hypothesis for this situation is that the child process takes a bit longer to execute because of the slight overhead for creating a new process. Since the parent process already exists, that overhead isn't present. That's why, without waitpid(), the parent process executes first. With waitpid(), though, that slight overhead doesn't matter since the parent process will wait as long as it takes for the child process to finish first.

   ii) In the rcreate() version of create() in XINU, which process will run first after rcreate() is called when the process running main() is created in 3.3(a)?

      1) The initial process priority of main() is 20. In main.c, a shell() process is started, which has a priority of 50. Since, in XINU, higher priority numbers mean that the process is more important, the shell() process will run once it is created and resumed using rcreate().

    2) The process that will run very first, though, is main(), because it has to start before shell() can be created with rcreate().

3.4

a) CDECL states that, before pushing the return address onto the stack, push all arguments onto the stack in reverse order. In each _Xint interrupt handler, you can see that ‘esp’ is pushed onto the stack, then the interrupt handler number. trap() takes two arguments, the interrupt number and the saved stack pointer (in that order). Since we must push these arguments in reverse order, ‘esp’ (the pointer to the current top of the stack) is pushed first, then 0, the interrupt number. This is how trap() receives these arguments.
b) When panic() is called, it disables interrupts, prints the panic message passed in, and infinitely loops. This prevents anything else from happening after the error. Interrupts are disabled so that this infinite loop cannot be exited.

3.5

a) Determine the name of the function that XINU sets up as the function that called main()
    i) ctxsw()
b) In the chain of functions being called starting with the virtual function to which main() returns, what is the last function that is called, where is its source code, and what does it do?
    i) The last function that is called is kill() (from userret()), its source code is located in **system/kill.c**. Basically, in **system/create.c**, each process is hardcoded with values in its stack. The process is technically never explicitly called, so it doesn’t have a return address. XINU sets the return address as INITRET, which is a macro for userret(). userret() calls kill(), which kills the process. Therefore, the return address for every process is a function that kills the process.
c) In environments where power consumption is an important issue -- e.g., battery powered mobile devices, or cloud computing servers where overheating must be prevented -- is XINU's approach to "shutting down" after the last process has terminated a good solution? What might be a better solution?
    i) After the last process outside of the null process terminates, the null process goes through an infinite while loop to prevent it from dying.
    ii) NO - not a good solution. The null process is still running (in the form of an infinite while loop) and consuming a bit of the CPU. Potentially, you could suspend the process and wait for an interrupt. Even then, since the null process doesn’t do anything, and there is nothing running on the system (even a shell), there would theoretically be no way for a new process to start, so the null process doesn’t need to be constantly running and wasting power.

BONUS

a) In Linux/UNIX system programming, we say that the fork() is special in that it is the only system call that "returns twice." What does that mean?

    i) fork() 'returns' in both the child and parent process. The return value of fork() in the child process is 0, and the return value of fork() in the parent process is the PID of the child process.

b) Why does this feature of fork() not apply to XINU's create()?
    i) XINU's create() does not start the new process, it creates it in a suspended state. Linux/UNIX's fork() starts the new process. Therefore, there is no new running process to 'return' in for XINU, where Linux/UNIX 'returns' in that new running process.

c) We say that Linux/UNIX's execve() system call is special in that it is the only system call that does not return when it succeeds. What does this mean?
    i) execve() replaces the calling process with the new process being executed. According to the linux manual page, the text, initialized data, uninitialized data (bss), and the stack of the calling process are overwritten according to the contents of the newly loaded program. Therefore, the process that called the new process no longer exists in memory, so it can't return there.

d) Why does this not apply to XINU's create()?
    i) XINU processes all share the same text, data, and bss sections of memory. This means that, when a new process is created, the text, data, and bss are NOT overwritten, like execve() does. Therefore, it returns, since there is still a place to return to (again, unlike execve()).