Lab 1

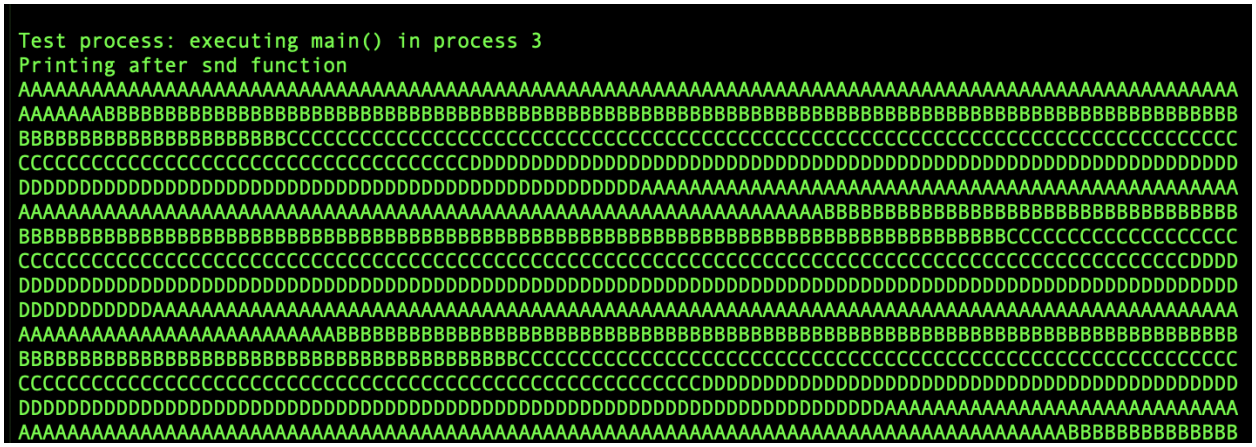
3.1(d):

In *process.h*, *INITRET* is defined as an address to which the process returns to. It does so using a function, *userret()*, which is called when a process returns from the top-level function. *userret()* terminates processes by calling *kill()* on it, which forces it to exit. So, *INITRET* does not return to any address, but instead terminates the process.

System calls in Linux that do not return even if they succeed are *exit()* and *exec()*. This is because *exit()* terminates the execution of the process that invoked it and destroys its memory address space, so there is no address for it to return to. Similarly, *exec()* and those similar to it do not return because it replaces the current process image with a new process image.

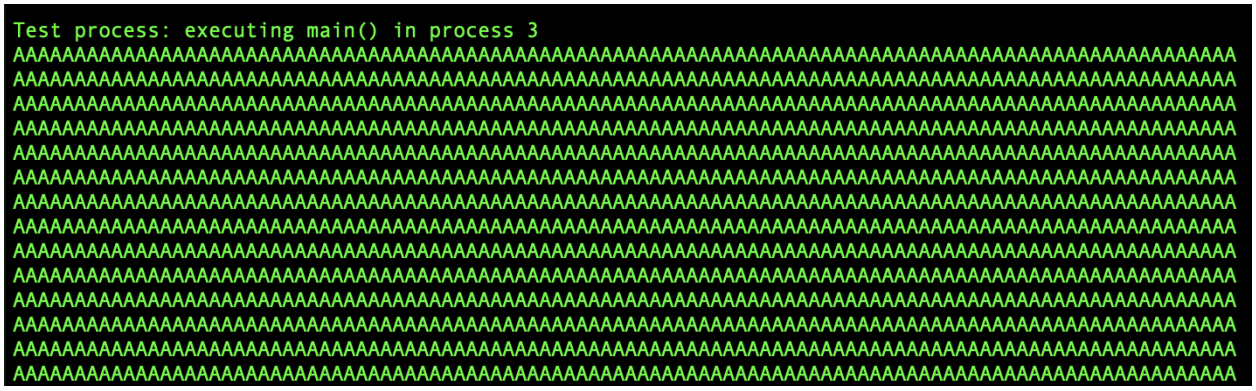
3.2:

main()'s priority = 30, children priority = 20:



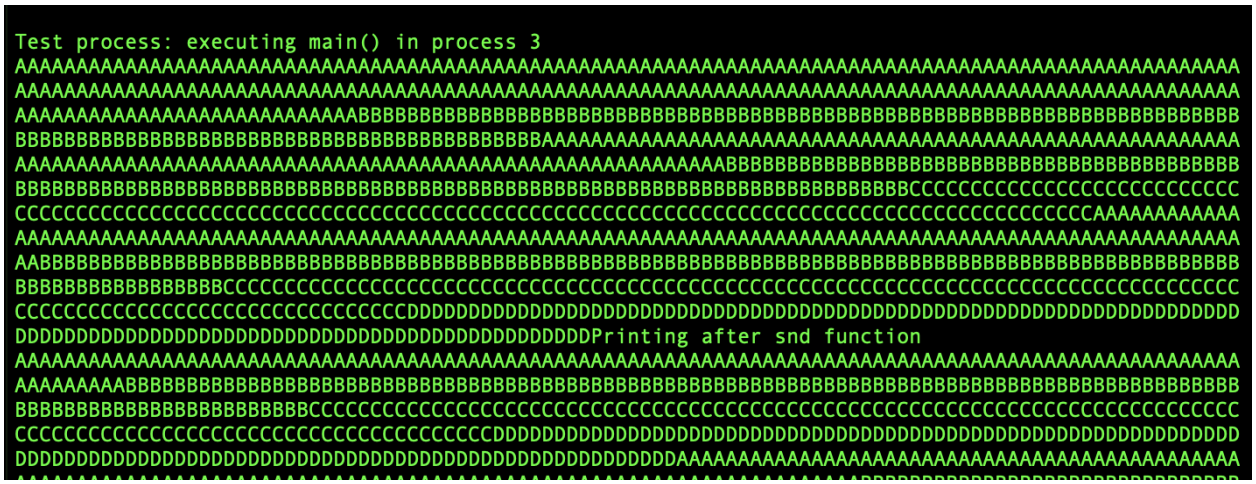
The main process printed "Test process: executing main() in process 3" and then it prints a string of As, followed by Bs, then Cs, and Ds. Then, the ABCD sequence keeps looping over and over again. This is because *main()* has the highest priority, so the scheduler will run that first and then terminate it once it's done. After it's terminated, the scheduler will run *sndA()*, *sndB()*, *sndC()*, and *sndD()* in a round-robin style since all of them have equal priority. Each child process is temporarily suspended once its time slice reaches 0 and the next child process gets to run.

main()'s priority = 10, children priority = 20:



The main process printed “Test process: executing main() in process 3” and then it only prints a string of As. This is because *main()*’s priority is lower than all child processes. So, after *sndA*() has been created, the scheduler will keep running *sndA()* since it has a higher priority than *main()*. As *main()* never gets to run again, the other three processes will not be created and, thus, won’t be printed.

main()’s priority = 20, children priority = 20:



The main process printed “Test process: executing main() in process 3” and then a string of As, a string of ABs, a string of ABCs, and then only prints the ABCD sequence over and over again. This is because as the child processes are being created, the scheduler gives control to those child processes since it has equal priority to *main()*. The scheduler then keeps looping through the child processes and *main()* in a round-robin style because all of them have equal priority.

4.1:

To test the implementation of *clkcountermsec*, a while loop is used to continuously print the values of *clktime* and *clkcountermsec* for 5 seconds/5000 milliseconds. An if condition is used to check that both clktime and *clkcountermsec* variables are updated before printing it out. It is found that both *clktime* and *clkcountermsec* are equal, but *clktime* is represented in seconds and *clkcountermsec* is represented in milliseconds.

4.3:

When QUANTUM is modified from 2 to 10, calling main() resulted in a longer string of ‘A’s, ‘B’s, ‘C’s, and ‘D’s compared to 3.2. This is because each child process has a bigger time slice before XINU’s scheduler is called to schedule another child process. Hence, each child process can print more characters before the scheduler executes the next child process in round-robin style.

Bonus problem:

Based on my findings, *clone()* is the most similar to XINU’s *create()*. Both functions create a separate process, its first argument is a pointer to the function to execute, has its own private stack, takes in optional arguments, and returns the process ID upon success. The slight difference between *clone()*and *create()* is that the former does not take in a priority parameter. Additionally, even though both functions use their own private stack, it is the parent process’ responsibility to set up the stack for the child process in *clone()*, but it is *create()*’s responsibility to allocate space for the child process’ stack.

fork() is not as similar to *create()* because it does not take any arguments to specify what code the child process should execute, and instead duplicates its parent process. *execve()* does not create a process, but only causes the program that is currently being run by the calling process to be replaced with a new program. While *CreateProcess()* is pretty similar to *create()* in terms of the arguments it takes in, it uses command lines as optional arguments, whereas *create()* has a parameter to specify the number of optional arguments. It also returns a nonzero value if the function succeeds instead of its PID