Assignment 5

Using what you have learned in the class lecture and your textbook reading about kernel scheduling, implement the single function in schedule.cpp (the scheduler() function).

This function is engaged during each system call as you can see below in the syscallHandler() function in syscalls.cpp.

```
1054    else if ((unsigned int)syscallNumber == SYS_BEEP)                { sysBeep(); }
1055    else if ((unsigned int)syscallNumber == SYS_CREATE)              { sysCreate((struct fileParameter *)arg1, cu
1056    else if ((unsigned int)syscallNumber == SYS_DELETE)              { sysDelete((struct fileParameter *)arg1, cu
1057    else if ((unsigned int)syscallNumber == SYS_OPEN_EMPTY)          { sysOpenEmpty((struct fileParameter *)arg1,
1058
1059    scheduler(currentPid);
1060
1061    returnedPid = readValueFromMemLoc(RUNNING_PID_LOC);
1062    newSysHandlertaskStructLocation = PROCESS_TABLE_LOC + (TASK_STRUCT_SIZE * (returnedPid - 1));
1063    newSysHandlerTask = (struct task*)newSysHandlertaskStructLocation;
1064
1065    asm volatile ("popa\n\t");
1066    asm volatile ("leave\n\t");
```

The scheduler() function in schedule.cpp will need to perform the following actions:

1. Check the KERNEL_CONFIGURATION to know if the scheduler should be active. If not, return without doing anything. If runScheduler is active, do the remaining steps here.
2. Devise and implement an algorithm that will give a process a fair proportion of CPU time based on its priority. For example, a process with a priority of 80 should get four times more runtime than a process with a priority of 20. **(This runtime should accumulate naturally over subsequent syscalls and settle on the ratio mentioned above. You should not lock up all processes for a long time while you give one process a large amount of upfront runtime. See my demo video for proper behavior.)**
3. Your algorithm must also eliminate the possibility of starvation for any sleeping process.

Also, you should not be calling sysWait() or a similar function in your scheduler to give the appearance of runtime (as this is really only burning CPU time in the kernel and not actually giving runtime to the user space binary). Additionally, you should not be enabling or disabling interrupts in your function.

You are free to use other Task struct elements as you see fit to implement your algorithm. These include Task->recentRuntime, Task->nice, etc. **You cannot modify Task->runtime or Task->priority, however.**

Additionally, be sure to use the constants (from constants.h) whenever possible. You should not be using a hard-coded value if there is a corresponding constant in constants.h, (but do not modify, add to, or alter constants.h in any way). You cannot create new helper functions and you should not create any new global variables. **Be sure to follow a professional coding standard, including the naming of your variables, structures, etc. You should use descriptive names that are appropriate. Nonsensical or whimsical names will lose points. You must also use the custom types where I have supplied them (e.g., uint8_t, uint16_t, uint32_t) unless you**

need a signed typed or one I haven't supplied. You will lose points if you don't follow this convention.

When you are happy with everything and have tested the functionality in the shell, upload the SUBMIT-ME.zip file to Canvas.

Additionally, you must attest and add the following honor statement to your Canvas submission (in the Canvas submission comments):

I, <state your name>, attest that this submission is my own work. I have not worked with anyone else on my implementation and have not used AI or other unpermitted sources to help me with my solution, in accordance with our definition of class cheating shown in Lecture 1.

Any submission that does not include this attestation will not receive any credit.