Assignment 2

Part I Questions – This part must be done on your own. Each question is worth 20 points.

1. Monolithic kernel architectures tend to perform better than microkernel architectures. The main reason is because monolithic kernel loads all OS modules into one kernel process with a single address space. As a result, OS modules can directly call each other to communicate. While with the microkernel OS architecture, OS modules are separated into user mode processes and a kernel mode process which provides a minimal set of essential kernel functions to the OS user mode processes, such as inter-process communication, process scheduling, etc.
   a. With the microkernel setup, explain why frequent communication amongst the OS user mode processes might have a noticeable negative impact on the overall OS performance.
   b. What are some of the drawbacks of a monolithic kernel in terms of its reliability and security?

2. Processes P2 and P14 are executing. The system has two types of pending I/O requests. P31 (first in queue) and P55 are waiting on secondary storage (hard drive) reads. P17 is waiting for a network write to complete. P25 (first in queue) and P19 are awaiting to be scheduled for execution on the CPU. Draw a process queueing diagram for these processes. Draw an arrow showing where P17 will go once it completes. Write the state that each process is in next to the process bubble.

3. Threads vs. Processes
   a. Explain why communication between threads (in a same process) can be more efficient than the communication between processes.
   b. Explain why context switch between threads is less costly than context switch between processes.

4. Suppose you are implementing a musical composition collaboration application. It provides video conferencing and audio recording capabilities to connect composers to work on a same piece of music, it automatically processes the composed musical data in the background, it frequently backups musical assets (ones not being processed) to a cloud storage, and it allows the collaborating composers to search for existing musical pieces from an online music marketplace. Composers would usually run the application on a **multiple-core CPU**. At run-time, the application process creates several threads, with each thread performing one of the tasks mentioned above.

Which threading model (**kernel-level or user-level**) you would use for the above-mentioned threads? Justify your answer.

Part II Programming – Poor person's Progress Bar (100 points)

This is a small programming assignment designed to give you experience in using POSIX threads. This programming assignment can be done with pair programming; you may form a group of 2 to work on this programming assignment.

Be sure to read the material on POSIX threads which is accessible through Canvas as well as information in the FAQ. **Do not implement a critical section** for this assignment. We have not yet covered critical sections and this assignment is designed such that they are not needed. You will have an opportunity to work with critical sections in a subsequent assignment.

It is frequently the case with GUI programs that a progress bar displays the progress that a task has made towards the completion of a task. You will write a subroutine for a "poor person's" progress bar which displays how much progress has been made towards a specific goal. The poor person's progress bar is textual in nature and alleviates you of the need to learn X Windows programming.

You are to write a function called **progress_monitor** that monitors the progress of a task that executes in a **separate thread**. Function **progress_monitor** must have the signature that all POSIX threads have:

```
void * progress_monitor(void *)
```

and should expect the void * argument to be a pointer to the following structure (you will need to typecast the argument in the function to be able to access member fields of this structure):

```
typedef struct {
   long *CurrentStatus;
   long  InitialValue;
   long  TerminationValue;
} PROGRESS_STATUS;
```

- CurrentStatus is a pointer to a long which represents the current status of the computation being tracked. *We will refer to the long integer to which this variable dereferences as the progress indicator.* The parent execution thread will be modifying the value to which this points as a task is completed (see below).

- InitialValue is the starting value for the computation.
- TerminationValue is the value at which the computation is complete.

You may assume that TerminationValue >= Progress Indicator (*CurrentStatus) >= InitialValue.

The function **progress_monitor** is only invoked once by the thread. It loops until the progress bar is complete and exits. Each time the thread executing the function progress_monitor is allocated the CPU time; it should compute the percentage of the task that has been completed and add to **a progress bar of 50 characters** representing the amount of progress that has been made. **Most characters are hyphens (-), but every 10th progress mark should be displayed as a plus (+)**.

Example:

Suppose that InitialValue = 0, TerminationValue = 50, and the progress indicator (*CurrentStatus) = 40. If the progress_monitor thread is scheduled under these conditions, 80% (40/50ths) of the task has been completed. Consequently, .80 * 50 = 40 progress markers should be displayed.

```
--------+---------+---------+---------+
```

When new marker characters need to be added, print them **without** a line feed character, so that the user will see a smooth progression of progress bar on their terminal. Note that putchar()/printf()/cout() typically buffer their output and only make a system call for output once the buffer is full. Request that they be printed immediately by using fflush(stdout) for C or cout.flush() for C++.

When the progress indicator has reached the termination value, the thread will print a linefeed and exit the thread. Remember that it is possible that you may need to print more than one hyphen at a time if more than an additional **1/50th** of the task has completed since the last time that the thread executing the function progress_monitor was scheduled.

The task whose progress you will measure is simple. You will need to write a program which **given a file name determines the number of words in the file**. Your Makefile should compile the program to an executable file named **wordcount**.

A sample session should appear as follows; you start the execution of your **`wordcount`** program from the console prompt:

```
edoras> wordcount big.txt
```

The output format of your program should be exactly the same as below (input in italics):

```
edoras> ./wordcount ~bshen/cs570/a2/big.txt
---------+---------+---------+---------+
There are 1095695 words in ~bshen/cs570/a2/big.txt.
```

You will be required to write at least three functions although you are certainly encouraged to use more if you see your program becoming unwieldy:

- **main** - Takes a command line argument of the filename to be counted. Main calls **wordcount** function (see below) with either the filename or the file descriptor of the file. Appropriate error handling should be present. It must print to stdout the following error messages when appropriate:
    - "no file specified"
    - "could not open file"

    After **wordcount** returns, **it prints the number of words** (**Use the EXACT format** shown above).

- **progress_monitor** - As described above. Note that the signature for thread entry point functions is special void * fn(void *) and you will need to typecast your arguments, see the FAQ for details.

- **wordcount** - Returns a long integer with the number of words and takes a file descriptor or filename as input (your choice). If you choose to pass in a file descriptor and you decide to use high-level input (e.g. the file descriptor returned from fopen as opposed to open) you may add an additional argument indicating the size of the file as that is easier to obtain from the file name using **stat or lstat** (see below). If you select to pass in a filename, you will need to open the file and provide error handling if needed. **wordcount will spawn a progress_monitor thread with a populated PROGRESS_STATUS structure as the argument.** PROGRESS_STATUS should contain:
    - *CurrentStatus - A pointer to a long used by wordcount to store the number of bytes processed so far.
    - InitialValue = 0
    - TerminationValue = Number of bytes in file. (See man **stat, lstat, or fstat** for how to obtain this. fstat requires a file descriptor from a low-level I/O call: e.g. open, whereas stat or lstat uses a filename. If you are using high-

level I/O, either use stat (or lstat) or open the file first with the low-level I/O, then call fstat, then close it.)

It will then read one character a time, updating the number of bytes processed and counting the number of words in the file. We will define a word as a non-zero length sequence of non whitespace characters (whitespace characters are tab, space, linefeed, newline, etc.). You may find it useful to use the library routine **isspace or iswspace** (see man page). You can check your results by using the UNIX command **wc** (see man page) which provides information about the number of lines, words, and bytes.

Important: Once it is done counting, **it needs to wait** for the progress_monitor thread to exit and returns the number of words counted.

In addition to the functions described above, you will need to provide a **Makefile** (see the class FAQ for a tutorial) which can be used with make to compile your program.

The directory ~bshen/cs570/a2 contains a number of classic texts along with a conglomeration of many books in the file big.txt. Use these to test your program along with any other files you wish. For the smaller works of Austen and Poe, you are unlikely to see your progress bar increment as the data file is small enough to count it too quickly.

What to Turn In

For each pair programming group, submit the following program artifacts by **ONLY ONE** group member in your group through the **Gradescope**, make sure you use the Group Submission feature in Gradescope submission to include your partner there.

Make sure that all files mentioned below (Source code files, Makefile, Affidavit) contain each team member's **name and RedID**!

- Canvas: Individual responses to part I must be submitted to Canvas.
- **Gradescope**:
  - Program Artifacts:
    i. **Source code files** (.h, .hpp, .cpp, .C, or .cc files, etc.), **Makefile**.
    ii. **A sample output (in a text file) from a test of your program.**
  - **Academic Honesty Affidavit** (no digital signature is required, type all student names and their REDIDs as signature):
    i. Pair programming Equitable Participation & Honesty Affidavit with all members' names listed on it. Use the pair-programmer affidavit template.

    ii. If you worked alone on the assignment, use the single-programmer affidavit template.
- Pair programmers: Turn in only one copy of your code. Gradescope has a button at the top right to add the name of your pair programming partner, please use it.

Plagiarism Check: Remember that all work must be your own. If plagiarism is found in either part of your assignment, you will be given a zero for the entire assignment and reported to SDSU Student Rights and Responsibilities. This is true for every assignment in this class. If you have any questions, about plagiarism, please take the library tutorial discussed in the syllabus and come to your professor if you still have questions. Automated tools are used in this course to help detect plagiarism.