# Project Topic One

## A Simulated Operating System

CSC3002

January 20, 2020

# Contents

# Chapter 1

# Operating System

You may feel curious about why your personal computers can store files, run games, and play videos. Actually, all these tasks are managed and scheduled by your operating system.

The system itself is a special software. Different from normal programs, an operating system operates as a layer between the hardware and the other applications. It helps user-level programs to get the required hardware resources and interact with the "real world". This is done by providing some pre-programmed system calls. Usually, some system-level libraries like libc will provide wrappers around these system calls by creating several low-level functions and then other programs are built on these libraries. For example (on Linux), when printing words, you may use the `std::cout` class. Actually, the `std::cout` will finally call the `write` function in libc (at `unistd.h`) which will invoke the `SYS_write` system call and let the OS kernel to handle the writing and flush the data to some specified addresses so that the hardware is able to display your "hello, world" (the procedure may be much more complicated, but it just goes like this).

There are many different operating systems. Some of the famous ones are Windows, MacOS, Linux and FreeBSD. There are mainly two categories of system kernels, the monolithic (or macro) kernel and the micro kernel. The previous one encapsulate all system services into one single program for performance consideration while the later one split the kernel into several micro programs for scalability and stability. There are pros and cons in both two designs.

Though the implementation of OS varies a lot among different distributions, we can find most of them contain the following components:

1. Memory Allocation: Allocate and release memory for program execution and protect the memory area from being read or modified by the programs without permission.

2. Task Scheduling: Schedule programs and balance the work load on CPU cores.

3. File System: Store and open the files. Arrange the data wisely for speed and space-saving. Self-heal when an error happens.

4. Exception System: Stop the execution when errors occur. Try to fix the problems and possibly reset the whole system if a fatal error (i.e. a triple fault) happens.

# Chapter 2

# Project Goals

As the startup of an operating system may involve a lot of operations on hardware and lines of assembly codes will be required, it might become too complicated for a term project and improper for the goal of practicing C/C++ language skills. Therefore, we only expect that you can implement a software that simulates an operating system. Basically, you can choose some of the system components to write and integrate them together as a program that can indicate what you have achieved.

For example, you program provides a simulated GUI environment with limited resources. We can spawn sub-processes within your program. However, when we open too many sub-processes, the memory will run out and the program will yield an error. You can pre-define your own routines for us to test the features of your simulated operating system and provide a user manual for it.

If you decide to write a simulated kernel using low-level system APIs, your work will definitely be more complicated and appreciated, and we will take it into consideration when grading your project. However, you can also write a fake GUI/TUI system as a program that can illustrate what is happening in a real OS in a really creative and interactive way. Remember, the design, the amount of effective code, and the language skills are the most important criteria in grading. There is a demo system running in the browser.

You need to decide the following things in your proposal:

1. What modules would you like to choose?
2. How will you display your final outcome?
3. Time table and work distribution in your group.

You are expected to hand in the following things when the project is finished:

1. Source code (archived).
2. A manual on how to prepare the environment and compile your source code (please make sure it is reproducible).

3. A report of your implementation with a manual of how to test the features.

# Appendix A

# References and Tips

**These references are listed here for those who are really interested in the modules of real operating systems. It is not required to go through the following materials but they can be useful if you write some advanced features.**

## A.1 OS

The Design and Implementation of the FreeBSD Operating System (link) is a good choice if you want to know what should be cared about in the practical implementation of a monolithic kernel.

Minix is another tiny but elegant operating system. It has a micro kernel with a feature to heal on itself when running into an error. Here is a book ( Operating Systems Design and Implementation, 3rd Edition) written for it. The book aslo talks about how to implement an operating system.

Musl is yet another implementation of libc. It may have worse performance compared to Glibc, but the code is much more readable as Glibc uses quite a lot "dark magics" (like cache awareness, SIMD and assembly code to interfere the calculation procedures) in its implementation (therefore, it is realy fast).

If you want to know more about Linux's system calls, manpages can be your good helper.

## A.2 Memory Allocation

There is a fantastic post on the memory model of a real operating system. This model can be applied in almost all popular operating systems. However, if you are implementing a simulated environment on an existed system, you can write a memory allocator instead, which is a wrapper of raw system calls.

In the real world, the reason why we are using allocators is that allocating directly with system calls is slow and unsafe, and the operation is kind of complicated. There are many implementations of allocator, such as the ptmalloc in

Glibc, the tcmalloc from Google, the jemalloc from Facebook and the mimalloc from Microsoft.

The source and paper of mimalloc is a good point to start with if you want to write an allocator on your own.

## A.3 Exceptions

You can read this post if you want to know about the exception system of CPUs, but it may be very hard to simulate it in the same way.

If you do want to simulate the exception system, you will find the exception library in STL is quite handy.

## A.4 Filesystem

If you are using C++17 or higher, you can the filesystem library in STL to iteract with the real filesystem in your operating system.

On Linux, you can also use mmap/munmap to help you map a file to memory and then run operations on it. (Please check the manpages for details if you are interested.)

## A.5 Thread Scheduling

If you are using C++11 or higher, you can the thread library in STL to help you with thread management. However, you must be aware that thread scheduling is quite hard. The atomic library is also ready to help you with atomic operations.

You can also write the mutex or other tools on your own and show your work in your report. Here are some tips on what else you can write in this module (no required, just some suggestions):

1. Atomic Operations using C++'s inline assembly grammar (knowledge on assembly is required)
2. Mutex using system calls (like `SYS_futex` on Linux).
3. Read-Write Lock based on Mutex.

## A.6 Language and VCS

The CPP Reference is a good place for you to learn more about the C++ language.

The abseil-cpp is a wonderful library to use as an extension to STL. It provides many handy functions and data structures to make C++ developers, lives become easier.

As the term project is a large project. It is important to learn about VCS. This tutorial can help you get familiar with Git.