**Final Project Documentation**

**NOTE: This documentation assumes familiarity with Visual Studio 2019. More info here https://visualstudio.microsoft.com/vs/getting-started/**

**Software/Hardware Requirements**

This project must be compiled run on a Windows 10 PC with QT and Visual Studio 2019 with the QT integration plugin installed. It has always been compiled with Visual Studio 2019 and run through the Visual Studio Debugger.

**Running the Project**

Follow this tutorial for setting up QT in Visual Studio is your environment does not already contain QT: <https://doc.qt.io/qtvstools/qtvstools-getting-started.html>. Once QT is installed on your system and the plugin is enabled in Visual Studio, open the OperatingSystemGUI.vcprojx project file in Visual Studio, compile and run the program in the Visual Studio environment.

**Implemented Solutions**

**Process Implementation and PCB:**

* **Relevant Files:** “Process.h”, “Process.cpp”, “PCB.h”
* **Implementation:** 
  + **Process Creation:**
    - Processes are created through a constructor that takes the string of the name of the program file to pull from. The program file is then read in, and the PCB is populated. (**Process.cpp lines 4 – 17, 59 – 133**).

**Critical Section Within each Process**

* **Relevant Files:** All program files, “Scheduler.h”, “Scheduler.cpp”, “Process.cpp”, “OperatingSystem.cpp”
* **Implementation:**
  + A structure called “ProcessMap” is defined in Process.h (**lines 19 - 31**). The map corresponds to a single instruction, and each process contains a vector of these maps. ProcessMap contains two Booleans, titled “is\_critical” and “end\_critical”. If an instruction is denoted with BEGIN\_CRITICAL in a program file, “is\_critical” is true in the ProcessMap. If an instruction is denoted with END\_CRITICAL, the Boolean end\_critical is true. This is used in the long-term scheduling algorithm to add and remove processes from the Semaphore waiting queue for deadlock avoidance (**OperatingSystem.cpp lines 170 – 175**).

**Critical Section Resolving Scheme and Deadlock Avoidance**

* **Relevant Files:** “Scheduler.h”, “OperatingSystem.cpp”, “Semaphore.h”
* **Implementation:**
  + The project uses a Semaphore implementation with a waiting queue. When a process is in the waiting queue, a Boolean “is\_sleeping” is marked, which tells the long term scheduler to put the process in the “waiting” state. The Semaphore implementation is defined in “**Semaphore.h”**

**Parent-Child Relationships**

* **Relevant Files:** “Process.h”, “Process.cpp”
* **Implementation:**
  + Inter-process communication is handled through the functions “fork” and “abort” defined in Process.h and implemented in Process.cpp (**lines 39 – 57).** Implementation is simple, but powerful, allowing for any number of recursive parent-child relationships.

**Inter-Process Communication**

* **Relevant Files:** “SharedMemory.h”, “SharedMemory.cpp”, “Mailbox.h”
* **Implementation:**
  + Two inter-process communications are employed: Shared Memory, and message-passing through a mailbox. The SharedMemory file contains a vector of all of the processes allowed to access it (usually ones with a parent-child relationship), and a mailbox is initialized with the two processes that can send messages back and forth from it.

**Scheduler**

* **Relevant Files:** “Scheduler.h”, “Scheduler.cpp”, “OperatingSystem.h”, “OperatingSystem.cpp”
* **Implementation:**
  + The project employs both a short term and long term scheduler. The short term scheduler, implemented in “Scheduler.cpp”, gives the choice between both round-robin and shortest-job-first scheduling algorithms. The long-term scheduler deals with I/O interrupts, critical section blocking, and other high-level process scheduling.

**Process Priorities**

* **Relevant Files:** “Process.h”, “Process.cpp”
* **Implementation:**
  + Priorities are randomly assigned to processes upon creation. (**OperatingSystem.cpp lines 274 – 276, lines 96 – 103).** The process class then overwrites the “<” operator to compare priority (**Process.cpp lines 52 – 55),** so that calling std::sort will automatically sort processes by priority. (This is also how shortest-job-first scheduling is implemented, overwriting “<” to compare two processes’ remaining runtimes).

**Process Resources**

* **Relevant Files:** “SharedMemory.h”
* **Implementation**:
  + A struct “SharedResource.h” is defined in SharedMemory.h (**lines 6 – 14)**. The SharedResource is then initialized in SharedMemory, meaning that the resource is shared between any process that has access to the SharedMemory pool that it belongs in.

**Basic Memory**

* **Relevant Files:** “MainMemory.h”, “MainMemory.cpp”, “OperatingSystem.cpp”
* **Implementation:**
  + The MainMemory uses the functions “add\_page”, “delete\_page”, “has\_page”, and “mem\_available” to add processes to the memory queue. It keeps a list of references to processes currently in memory. If there’s no memory available, the long-term scheduler won’t let the process enter the READY state.

**Memory Divided into Hierarchy**

* **Relevant Files:** “Cache.h”, “Cache.cpp”, ExternalStorage.h”, “ExternalStorage.cpp”
* **Implementation:**
  + Memory is rounded out using cache and external storage. ExternalStorage.h will store a copy of a process upon its first initialization, and subsequent initializations will pull from the copy, rather than the program file. Cache limits the amount of processes that can enter the RUNNING state for a given CPU at a time, acting like a CPU cache.

**Virtual Memory and Paging**

* **Relevant Files:** “VirtualMemory.h”, “VirtualMemory.cpp”, “PageTable.h”, “PageTable.cpp”
* **Implementation:**
  + Each process has a VirtualMemory object which contains a vector of integers. The long-term scheduler passes these pages to PageTable.cpp, which uses random chance to say whether or not a page can make it into memory (90% chance for a page hit).

**I/O Interrupts and Handlers**

* **Relevant Files:** “CPU.cpp”
* **Implementation:**
  + By random chance, a running process may be caught in a random IO state (**CPU.cpp lines 22 – 31)**, where its “random\_IO” integer will be set to a random value. The short-term scheduler will then keep the process in the waiting state for the length of the integer value – the time quantum (**Scheduler.cpp lines 30 – 55).**

**Multi-Core and Multithreading**

* **Relevant Files:** “OperatingSystem.cpp”
* **Implementation:**
  + The OperatingSystem class defines two CPUs, CPU0 and CPU1. The dispatcher will schedule up to four processes (for 2 cores/4 threads), and depending on the amount of processes scheduled they either execute on one, two, three, or four threads (**OperatingSystem.cpp lines 54 – 82).** Multithreading is implemented through std::thread in the C++ 11 Standard Library.

**GUI**

* **Relevant Files:** ‘OperatingSystemGUI.cpp”
* **Implementation:**
  + The GUI is implemented using the QT SDK for C++, and its relevant Visual Studio tools and plugins. A timer executes loops of the OperatingSystem program, and three tables display real-time visualizations of process execution.

**Loading External Processes and Generating New Ones on User Request**

* **Relevant Files: “**OperatingSystemGUI.cpp”
* **Implementation:**
  + Embedded in the GUI is a text input object that takes commands to load new processes and cancel existing processes through the commands “new\_process <program file>” and “stop\_processes <process name>” (**OperatingSystemGUI.cpp lines 36 – 70).**