Project Objective

The best way for you to understand the concepts of an Operating System is to build an operating system and then to experiment with it to see how the OS manages resources and processes. In this project, you are asked to build a simulation of an operating system. The main focus of the project will be on building a correct architecture that simulates a real operating system. You will be graded on both a correct implementation and a correct architecture.

Milestone 1

In this milestone, you are asked to implement a basic interpreter. You have a text file that represents a program. When you read that text file and start executing it, it becomes a process. You are provided with 3 program files each representing a program. You are asked to create an interpreter that reads the txt files and executes their code. You are also asked to implement mutexes that ensure mutual exclusion over the critical resources. And Finally, you are also asked to implement a scheduler that schedules the processes that we have in our system.

System Calls

System calls are the process's way of requesting a service from the OS. In order for a process to be able to use any of the available hardware, it makes a request, system call, to the operating system.

Types of system calls required:

1. Read the data of any file from the disk.
2. Write text output to a file in the disk.
3. Print data on the screen.
4. Take text input from the user.
5. Reading data from memory.
6. Writing data to memory.

The memory for now refers to the variables you will initialise and assign a value to. This is achieved through the assign instruction discussed in the Program Syntax section. These variables are unique and owned by each program/process and thus are not shared. The variable names are given by the assign instruction at runtime and are not only a and b, thus your memory needs to store data with the process that owns it and the name given to it.

Programs

We have 3 main Programs:

Program 1: Given 2 numbers, the program prints the numbers between the 2 given numbers on the screen.
Program 2: Given a filename and data, the program writes the data to the file. Assume that the file doesn't exist and should always be created.

Program 3: Given a filename, the program prints the contents of the file on the screen.

Program Syntax

For the programs, the following syntax is used:

- print: to print the output on the screen. Example: print x
- assign: to initialize a new variable and assign a value to it. Example: assign x y, where x is the variable and y is the value assigned. The value could be an integer number, or a string. If y is input, it first prints to the screen "Please enter a value", then the value is taken as an input from the user.
- writeFile: to write data to a file. Example: writeFile x y, where x is the filename and y is the data.
- readFile: to read data from a file. Example: readFile x, where x is the filename
- printFromTo: to print all numbers between 2 numbers. Example: printFromTo x y, where x is the first number, and y is the second number.
- semWait: to acquire a resource. Example: semWait x, where x is the resource name. For more details, refer to section Mutual Exclusion
- semSignal: to release a resource. Example: semSignal x, where x is the resource name. For more details, refer to section Mutual Exclusion

Mutual Exclusion

A mutex is a directive provided by the OS used to control access to a shared resource between processes in a concurrent system such as a multi-programming operating system by using two atomic operations, semwait and semsignal. Mutexes are used to ensure mutual exclusion over the critical section.

You are required to implement 3 mutexes, one for each resource we have:

1. Accessing a file, to read or to write.
2. Taking user input
3. Outputting on the screen.

Whenever a mutex is used, either a semWait or semSignal instruction is followed by the name of the resource, userInput, userOutput or file. For an illustration, to print on the screen: -

1. semWait userOutput: any process calls it whenever it wants to print something on the screen to acquire the key of the resource.
2. semSignal userInput: any process calls it whenever it finishes printing to release the key of the resource.

Note: ONLY ONE process is allowed to use the resource at a time. If a process requests the use of a resource while it is being used by another process, it should be blocked and added to the blocked queue of this resource and the general blocked queue.

Scheduler

A scheduler is responsible for scheduling between the processes in the Ready Queue. It ensures that all processes get a chance to execute. A scheduling Algorithm is an algorithm that chooses the process that gets to execute. As mentioned in the lecture, there are many different scheduling algorithms to schedule processes. In this project, you are required to implement the Round Robin algorithm. Round robin is a scheduling algorithm where each process is assigned a fixed time slice. For this project, each process executes 2 instructions in its time slice.

Processes arrive in this order: Process 1 arrives at time 0, Process 2 arrives at time 1, and Process 3 arrives at time 4.

Queues

- Ready Queue: For the processes currently waiting to be chosen to execute on the processor
- Blocked Queue: For the processes currently waiting for resources to be available

Output

For this Milestone, your Simulated OS should be able to read the provided programs and execute them. You should make sure to have the following outputs read to show for the evaluation:

- Queues should be printed after every scheduling event, i.e. when a process is chosen, blocked, or finished.
- Which process is currently executing.
- The instruction that is currently executing
- Time slice is subject to change, i.e. you might be asked to change it to x instructions per time slice.
- Order in which the processes are scheduled are subject to change.
- The timings in which processes arrive are subject to change. Please make sure that the output is readable, and presentable.

Milestone 2

In this milestone, you are required to extend your milestone 1 code by implementing a memory.

Memory Management

In milestone 1, your system stored all the created variables in a single data structure and there was no protection offered for each process's data. In this milestone, you are expected to augment your memory system by making the OS manage it and assign a space for each process. The memory is of a fixed size. It is made up of 40 memory words. The memory is large enough to hold the un-parsed lines of code, variables and PCB for any of the processes. The memory is divided into memory words, each word can store 1 variable and its corresponding data. For simplicity, feel free to specify a naming convention for the variable names associated with the lines of code and elements of the PCB. Processes should not access any data outside their allocated memory block. A process should only be created at its arrival time. A process is considered created when it's program file is read into lines and it gets assigned a part of the memory for instructions, variables and its PCB. Assume that each process needs enough space for 3 variables.

Feel free to separate the lines of code, variables and PCB within the memory if needed as long as they fall within the same data structure meant to represent the memory.

Your code needs to be able to handle the case where the memory is not large enough to run all the processes. When a new process is created, the system checks if there is enough space, if not the system will unload one of the existing processes and store its data on the disk. Feel free to assume the format for the memory when stored on the disk. However, while unloaded, the process remains in the scheduler, thus when it is time for the unloaded process to run again, the process memory is swapped back into the memory from the disk. As a result, you need to program a way to read and write an existing processes memory from the disk as well manage the memory swapping process and protect each processes memory.

Process Control Block

A process control block is a data structure used by computer operating systems to store all the information about a process. In order to schedule your processes, you will need to keep a PCB for every process. The PCB should contain the following information:

1. Process ID (The process is given an ID when is being created)
2. Process State
3. Program Counter
4. Memory Boundaries

Output

For this Milestone, your Simulated OS should be able to manage the memory based on the arrival times and execution times of the program files in milestone 1. You should make sure to have the following outputs ready to show for the evaluation:

- The memory shown every clock cycle in a human readable format.
- The ID of any process whenever it is swapped in or out of disk.

- The format of the memory stored on Disk.