Operating Systems (COM 306)

Section 1

Professor Daggett

TEAM BATTLEFIELD

Phase 1

Team Members - Alex Klavens, Brinley Bartlett, Hugh Kim, Skylar Levey, Wali Hairan

Introduction

The purpose of this paper is to document the thought process and steps taken to develop the operating system process scheduler. Details include functional and nonfunctional requirements, general and detailed design overviews, goals, guidelines, and restraints. The scope of the documentation is intended to help the team stay focused on the objectives and provide specific guidelines when making change. The intended audience is a software engineer that is interested in the reasoning and rationale of the development of the process scheduler design. The paper is organized by explaining the requirements, design considerations, and architectural strategies, followed by the system architecture description, and the policies and tactics required. It closes with the test plan in order to ensure proper implementation of the process scheduler.

Requirements

This scheduler shall be able to to read in a list of process definitions, update them, and then simulate the completion of tasks, in a GUI. The input file type is a CSV in which each line should be a process that is ordered - process name, priority, cycles to finish, and RAM necessary. The user shall also be able to enter processes dynamically in the GUI.

The file that is being read is a csv file, where each row is a process. On the GUI, the user will input a text file name to be read. That information will then be populated to the scheduler via a priority queue and a fairness queue. The priority queue consists of a heap of queues while the fairness queue is just a normal queue holding executables for a certain (predetermined) amount of time before getting executed. The required amount of RAM will be able to be chosen from a drop down menu based on the acceptable input format.

System Overview

The process scheduler consists if a heap of FCFS queues based on priority, and then another queue that keeps track of the amount of time each process has been waiting to provide an element of fairness. These two data structures are connected to a GUI from which the scheduler is controlled and the user can enter processes.

Design Considerations

Assumptions and Dependencies

It is assumed that a this scheduler will run continuously until it is finished, interrupted, or the user stops it. Secondly we assumed that the computer has a dual core processor and that each core can run one process at a time. We have not incorporated data checking as we assume the user of the processor will not intentionally enter misinformation to break the program. Lastly we assumed that no two processes will require more than 16 gb together.

General Constraints

We limited ourselves to taking in data via csv files.
We limited ourselves to two cpu cores that are able to run programs.

Goals and Guidelines

Our biggest goal was to create a functional process scheduler that has a friendly GUI which allows the user to visualize the process scheduler and interact with it.
We also wanted to prioritize understandability and simplicity over raw performance.

Development Methods

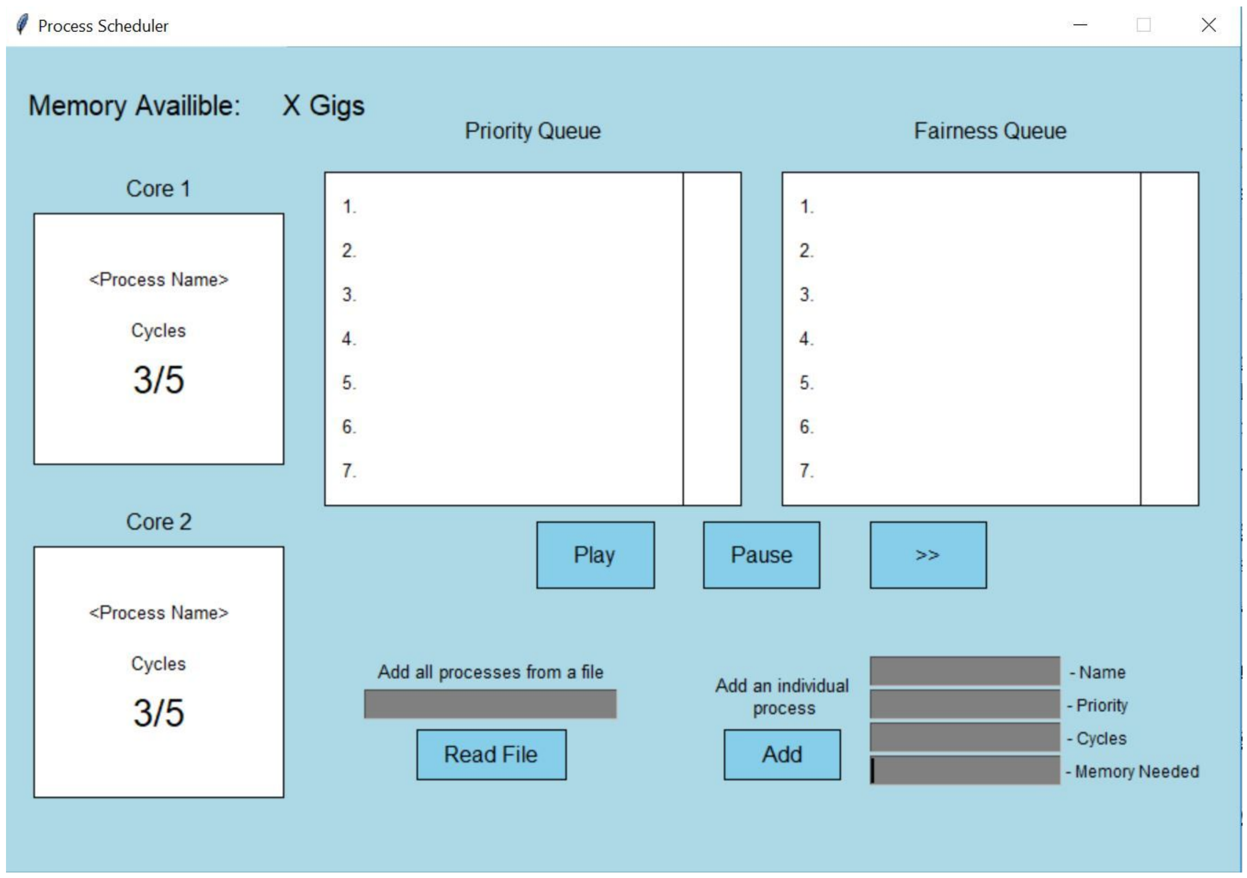
The team created classes for our data structures and then made a GUI in zelle graphics that provided an interface for the user to interact with the data structures

Architectural Strategies

- The language we chose was Python 3.6.6. The group was most familiar with this language.
- We chose a heap for our priority optimization because we felt it was the best data structure to organize processes via priority. Our heap consists of different Queues that each hold processes of a certain priority, so for all processes with priority 1 it is a first come first serve, but all priority 1 processes will be served before any of the priority 2 processes.
- We chose a regular Queue for our fairness data structure. It works in parallel with the priority heap and is simply a first in first out Queue, the catch is that it is just a holder for the processes and all it does is hold them until they have been waiting longer than a set value of cycles. Once this occurs they are removed from both data structures and then added back into the Priority heap with a priority of -1 so it is the next process served.
- Given that we chose to restrain ourselves to two cores the memory was not as big of a restraint as most combinations of two processes will not require more than 16 GB of RAM.

System Architecture

The below image is the GUI as it loads up. The user is then able to either enter processes directly or have a CSV file read into the system. With both the processes are then added to both the priority heap and the fairness queue. The user can then run cycles of the processor in which the cores are populated with processes at the top of the priority heap. These processes are run until they are finished at which point they leave the cores and the next set of instructions are loaded in. While this is occuring the visual representations of the two data structures are being updated to always reflect the actual state of the data structures themselves.



Subsystem Architecture

At this point in the project, no other subcomponents were involved in the file reading process and inputting that data into the two different data structures.

Policies and Tactics

The user interface will display how many cores the operating system has, the priority queue, and fairness queue. The intractability is demonstrated in the Read File and text box input where the user can enter in the file name that has contains the processes. The user can also add an individual process by entering their own Name, Priority, Cycles, and Memory Needed through input boxes. The user also has the ability click Play, Pause, or >> which shows the movement of the processes in the different queues.

A test file will be read and then the information in that file will be populated in the two data structures. By clicking play the user will be able to see the order in which the processes are run. This ensures that test data works and follows the correct path. An error will display on the user side if there is an issue, although this features has not been implemented in the code yet.

<u>**Detailed System Design**</u>

Classification:

- Heap of Queues- this data structure is in charge of taking the processes and making sure that it will run in the order of their priority.

Definition:

- The purpose of this data structure ensures that processes will run in order.

Responsibilities:

- This priority queue will allow for the proper order of processes run. If a user were to input a new process that should be run immediately, the priority queue will take ensure that it will be next in the queue.

Constraints:

- We reduced the amount of constraints by adding a fairness queue that will allow a process to be run despite its priority based on the number of cycles.

Composition:

- The high priority will be the process to run next.

User/Interactions:

- Once the file is read, it interacts with the two data structures used to correctly populate and run the processes.

Resources:

- This queue was created by the team without downloading a previous existing queue data structure.

Processing:

- The information read from the file will be inputted into the heap of queues. The process with the highest priority will run first.

Interface/Exports:

- The data structure is called as soon as the file is read. It will be shown in the GUI under the priority queue box. The ability to export the data being shown to the user has not been implemented yet.

Classification:

- Queue-this data structure holds the order of priorities that haven't been run yet.

Definition:

- The purpose of this data structure ensures that a process will never not run.

Responsibilities:

- The data structure ensures that once a certain time is reached, the process that is still waiting in the queue, will eventually run.

Constraints:

- The constraints to this data structure is assigning the correct amount of time that goes by before the process waiting should be put over to the priority queue (heap of queues).

Composition:

- The priority of the event will be ordered from high to low. The high priority will be the process to run next.

User/Interactions:

- Once the file is read, it interacts with the two data structures used to correctly populate and run the processes.

Resources:

- This queue was created by the team without downloading a previous existing queue data structure.

Processing:

- The information read from the file will be inputted into the queue.

Interface/Exports:

- The data structure is called as soon as the file is read. It will be shown in the GUI under the fairness queue box. The ability to export the data being shown to the user has not been implemented yet.

Detailed Subsystem Design

The heap of queues will take in the processes and run them based on their priority value. The higher the priority, the closer they are to having the process run. While populating in the heap of queues, the processes are also being populated in the fairness queue. In order to prevent a process from never running, the fairness queue makes it so that following a specific amount of time, the process in the queue will then be pushed higher in the priority queue and then will be run next.

Test Plan

A test file will be read and then the information in that file will be populated in the two data structures. By clicking play the user will be able to see the order in which the processes are run. This ensures that test data works and follows the correct path. An error will display on the user side if there is an issue.

Glossary

n/a

Bibliography

n/a