Lab 3 Design Document

Group 2
Aaron Meurer
Oran Wallace
Sheng Lundquist

Introduction

Our group has decided to implement a Round-Robin scheduling approach for Fair-Share Group algorithm and a Priority based scheduling algorithm for the interactive needs of a user. Both will be explained in this document.

The process scheduler will handle creating unique PIDs with a global counter. If it overflows beyond $2^{31} - 1$ (or $2^{63} - 1$) we will just consider it to be an unrecoverable error for now (it won't happen very often). We will use the New queue as a holding place for processes before we do initialization tasks on them. For now, this just means that we will put new processes in the new queue first, then create unique PIDs for them. In the future this will also probably involve other tasks, such as memory allocation.

There are a few changes that will be made to the original process manager. One of them is a new command to clear out the terminated queue, so that we can continually create new processes without resetting the process manager. This will decrement the process counter to allow for a maximum of 20 processes in the process manager at a time.

Secondly, the Go and Eoquantum command will be merged. This is done to help optimize the runtime of the interactive scheduler. This will be explained in further detail later on.

Finally, the test implementer will have the option to run the program through commands in stdin, as well as the ability to read from a file.

Process Control Block

There will be two new variables implemented in the process control block to implement priority: A priority variable and a quantum_count. The usage of these two variables will be explained in the description of the interactive scheduler.

Fair-share Scheduler

For our fair-share scheduling algorithm our group decided to implement a Round Robin approach. The time quantum for the algorithm will be emulated through the eoquantum function call that comes from the hardware. The method for the Round-Robin Scheduler is simple. The scheduler will choose the process at the front of the Ready Queue to be the one to use the CPU. Three things can happen at this point: (1) The process finishes executing while using the CPU, to which the scheduler will simply dequeue from the ready queue to have the CPU. (2) The process is waited for whatever reason, and is moved into the waiting queue. It can move back to the bottom of the ready queue with an unwait. (3) The time quantum is reached, an eoquantum is issued, and the process that is using the CPU will be enqueued into the ready queue. This process is repeated until there are no more processes in the Ready Queue. (Note that our current implementation of the process manager already essentially does round-robin scheduling.)

Interactive Scheduler

Our interactive scheduler will be a priority based scheduling algorithm. The priority of processes will always be between 1 and 20, inclusive (20 being the highest priority, 1 being the lowest priority). As processes are created, the default priority will be 10 (the fair-share scheduler will set this variable to 0). The priority will change based on how the process returns to the ready queue. If a running process “eoquantums” when a Go command is issued, its priority will decrease by one. This will emulate a long computational process, which will have a low priority relative to other processes (to aviod starvation). If a process waits, its priority will increase by one. This will emulate a process that does lots of waits, which will have a high priority relative to other processes, because it will tend to free the CPU. A set_priority function will also be implemented for the user/OS (preferably only OS) to set an exact priority for certain processes. This could also be useful for testing.

The starvation problem exists in the above solution for long computational processes. To alleviate this problem, an aging solution will be implemented. The following sequence of commands will be implemented every time the Go function is called:

1. Iterate through each process in the ready queue, increasing their corresponding quantum_count by one.
2. If the process's quantum_count is greater or equal than it's own priority, it's quantum_count will reset back to 0, and this process's priority will be incremented.

This implementation will increase the priority of a process faster if the process's priority is lower. As an example, if a process of priority 1 (lowest priority) is in the ready queue and one eoquantum is called (note that quantum_count will not be increased to the current running process that eoquantum is called upon), that process's priority will be increased to 2. The same process's priority will increase again after two eoquantums are called, and so on. This will create a dynamic time to increase priority based on what priority the process has.

The go command will dequeue the current process in the running queue (as if eoquantum has been called), pick the process from the ready queue with the highest priority (it will break ties by picking the one closest to the head of the queue), and enqueue it to the running queue.

Design Choices

We choose to implement the interactive priority scheduler as a single queue. This will make the go command as linear time based on the number of processes that exists in the ready queue. This was done to alleviate the starvation problem when scheduling processes and to provide an interactive scheduling algorithm. To avoid costly multiple iterations through the ready queue, the Go and Eoquantum commands were merged. This allows for one iteration through the ready queue to find the process with the highest priority and to increment the quantum_count of all processes in the queue.

The alternative to this design was to create a multi-level feedback queue. This would have been a complex implementation containing multiple queues each with their or priority level. Although this implementation would have decreased the running time of the Go command to constant time, it would have been difficult to avoid starvation of lower priority processes. Our current implementation of aging processes would still have been linear time, which would have negated the advantage of having constant time Go command. Therefore, we decided that this implementation will emulate a long-term scheduler rather than a short-term scheduler. Runtime speed was sacrificed to achieve a solution to the starvation problem.

Eoquantum was the command we chose to implement the aging process to simulate a time lapse for aging. Since a timer was not available for the process manager, we decided to use the time quantum as our unit of time to age. Since the quantum is simulated through the hardware calling Go, this would alleviate the usage of a timer in the operating system's scheduler.

Round-Robin Scheduler (Flow Chart)

This flow chart begins with a process being enqueued to the New Queue. Next the process is dequeued from the New Queue, assigned a unique PID, and enqueued to the Ready Queue. Following this the process is dequeued from whatever queue it is in (most likely the Ready queue). A process waiting in the Ready Queue will then have a Go command issued, this will automatically dequeue a process in the Running queue (if there is one) and enqueue the process at the head of the Ready queue to the Running queue. From here two commands can be issued for a process in the Running queue: (1) A Eolife command will be performed and the current Running process will be dequeued form the Running queue and enqueued to the Terminated queue. (2) A Wait command can be issued, and the process will be dequeued from the Running queue and enqueue to the Waiting queue. From the Waiting queue a Unwait command must be performed to dequeued the process from the Waiting queue back into the Ready queue.

Priority Scheduler (Flow Chart)

This flow chart is very similar the the Round-Robin one except that priorities are increased and decreased depending on the command issued. If a process is in the Ready Queue and a Go command is issued, the same method is used as in the other chart except the aging process will be implemented, and the priority for the current running process will be decreased by one. If a process is in the Running Queue and a wait command is issued then the process remains the same as before except that the priority for that process is increased by one.