Alexandra Hurst & Kalyani Khutale
CISC-361
2018-05-24
Final Project

Final Project Report

Compile instructions:

```
$ make
```

Run instructions:

```
$ ./SchedulingSimulator <input_file.txt>
```

Clean instructions:

```
$ make clean
```

Note: all commands should be run from the directory containing the project source code.

Explanation

The program opens the input file and then reads it line by line, parsing each line's event and adding it to an event queue, which is ordered by time and type (internal or external). After an event is parsed, the queue is processed up through the time of the parsed event, executing all intermediate events and then the parsed event in sequence. The types of event are: system configuration (not a true event but necessary for setup), job arrival, device request, device release, state display, and quantum end. All but quantum end are external.

When a configuration directive is read at the beginning of the file, it initializes the system state with the given maximum memory, maximum devices, and quantum length at the specified time.

When a job arrival directive is read, a job and its corresponding event are created with the given job number, memory requirements, maximum device requirements, running time, and priority. The event is then scheduled on the event queue to be run at the specified time. When the event is processed, one of three things may happen. If the job's resource requirements exceed the maximum system resources, it will be rejected completely. If the job instead requires more memory than is currently available but not more than the system maximum, it will be placed into the corresponding-priority hold queue to wait for memory to be freed. Finally, if the job's memory requirements are less than or equal to the available system memory, the job will be placed on the ready queue. When the queues are updated following event processing, the job may be moved onto the CPU if there is no job currently occupying it.

When a device request directive is read, a device request event is created with the id of the requesting job and the number of devices requested. It is scheduled on the event queue to be run at the specified time. When the event is processed, it will first check that the requesting job is on the CPU and error out if not. If the job is on the CPU, it will be marked with the number of devices it is requesting and its quantum will be ended. Then, during the queue update, the banker's algorithm will be run to determine whether the request can be granted immediately. If so, the requested devices are allocated and the job is scheduled back on the ready queue. If not, the devices are not allocated and the job is instead scheduled on the device wait queue.

When a device release directive is read, a device release event is created with the id of the releasing job and the number of devices to be released. It is scheduled on the event queue to be run at the specified time. When the event is processed, it will first check that the releasing job is on the CPU and error out if not. If the job is on the CPU, its devices will be released and its quantum will be ended. Then, during the queue update, the job will be scheduled back on the ready queue, and each job in the device wait

queue will be checked in first-in-first-out order using the banker's algorithm to determine if its request can be granted and it can be moved to the ready queue or not.

When a state display directive is read, a display event is created and scheduled on the event queue to be run at the specified time. When the event is processed, the system's state is printed out to the console in a textual format, and a json representation of the system state is written to a file on the disk. Both representations contain data about the current state of each job in the system (e.g. in the ready queue, waiting for devices, completed at a certain time, etc.). If it is not explicitly specified in the input file, a final state display event will be generated to be processed at time 9999, so as to show the final state of the system.

Quantum end events are not read directly from the input file, but are instead implicitly generated each time a new job is placed on the CPU. The generated quantum end event will be scheduled at the current time of the system plus the length of one quantum, thus marking the end of the just-scheduled job's time on the CPU if it is not interrupted first by another event. The quantum end event does nothing when processed, instead serving only to trigger the queue update function so that the job can be moved off the CPU and another job can be moved onto the CPU from the ready queue.

The queue update function handles the majority of the complex scheduling logic, including moving jobs on and off the CPU and between queues.

First, it moves the currently running job off the CPU if necessary (i.e. if the current quantum has ended, either naturally or by a device request/release). If the job has completed it's runtime, it is moved to the complete queue. If the job still has runtime remaining, the function checks whether the job has an outstanding device request. If so, the banker's algorithm is run to determine whether the request can be serviced immediately (job is moved to ready queue) or must wait (job is moved to wait queue). If the job has no outstanding device request, however, it is simply placed back into the ready queue.

Next, the queue update function moves jobs between the various queues, if necessary. It first checks each job in the wait queue using the banker's algorithm to see if it can be moved to the ready queue or not. Jobs in the wait queue are checked in first-in-first-out order. Then, it checks hold queues 1 and 2, in that order, to see if any jobs therein now fit into memory and thus can be moved to the ready queue. Jobs in hold queue 1 are checked in order of shortest to longest runtime, while Jobs in hold queue 2 are checked in first-in-first-out order.

Finally, if a job was moved off the CPU earlier in the function, the queue update function now moves the next job (if any) from the ready queue onto the CPU and returns.