Project 1: Virtualization

For this project, I created a program that simulates the way an operating system manages memory and schedules CPU time. The simulator does this by opening a file of "process" data, creating job "objects" using the data, then utilizing the simulated memory and CPU to process the set of jobs to completion. Along the way the program keeps track of statistics such as memory hole percentage and average waiting time for the processes.

**The Process**

In my design for this program, I opted to code in C++ and create two custom classes: a CPUjob class that holds data for each process, and a CPUscheduler class. The CPU scheduler holds all the necessary queues and arrays for implementing the simulation (three vectors for MLFQ levels, a wait queue, a memory map vector) and the numerical data needed to keep track of everything in the process. One of the early decisions I made was to have the queues and vectors contain pointers to CPUjob objects. That way I could maintain one list of the jobs in my driver, and pass references to them through the CPUscheduler rather than trying to pass the objects themselves all around. This made it much easier to have one central data structure containing all the objects for aggregating the data.

As part of the assignment was to implement both next-fit and worst-fit memory mapping algorithms, I decided to separate my memory mapping function from the scheduling function. I split the function into two checkWaitQ functions (one for each algorithm) in which the wait queue is examined to see if there are any ready tasks not inside it. Each algorithm then uses its own method to see if there is space for the first job in the queue in memory before inserting it, or waiting for more space to free up.

After this step, the scheduler proceeds to each priority level queue of the MLFQ system in turn to see which job's turn it is with the CPU at the current clock tick. Each queue level then checks if the process is finished or has used up its time slice. Whichever is the case, the scheduler processes the application appropriately and returns a value so the driver knows what information to print to the screen.

For the priority level queues, I included index numbers to indicate which process in the queue is currently or next to receive CPU attention. That way, if a lower priority queue is interrupted by a new job entering the highest priority, the lower priority queue knows which job to return to later. Also included in each queue is a short term counter that keeps track of how many time units the current job has used up at that level. Once the counter reaches the time quantum size, the process is pushed down a level (unless already at the lowest level), counter resets, and the index moves to the next job in that queue.

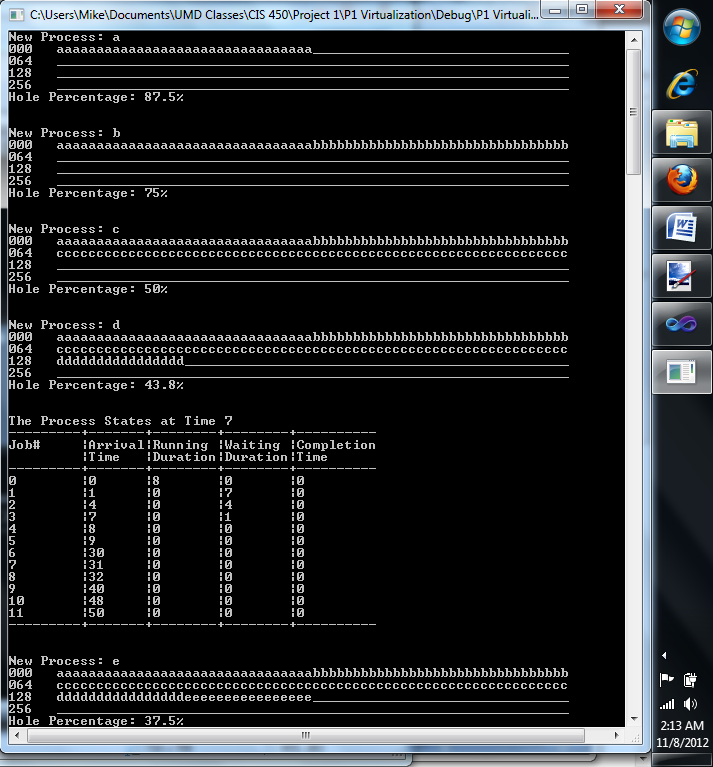
The program driver creates a vector to hold all of the jobs received from the input file. It also contains an array of ordered characters. Each job entering the list receives a character id from this that is later used to identify it in the memory map. Inside the driver begins a loop that cycles once for each simulated clock tick of the CPU. First the loop checks if the next job in the list is ready by comparing it's arrival time to the clock. If it is, the driver calls a CPUscheduler function to push it into the wait queue. It then calls the scheduling function before incrementing the clock and looping back again. Lastly, the driver prints off a final process state table and the necessary data from the simulation.

**Testing**

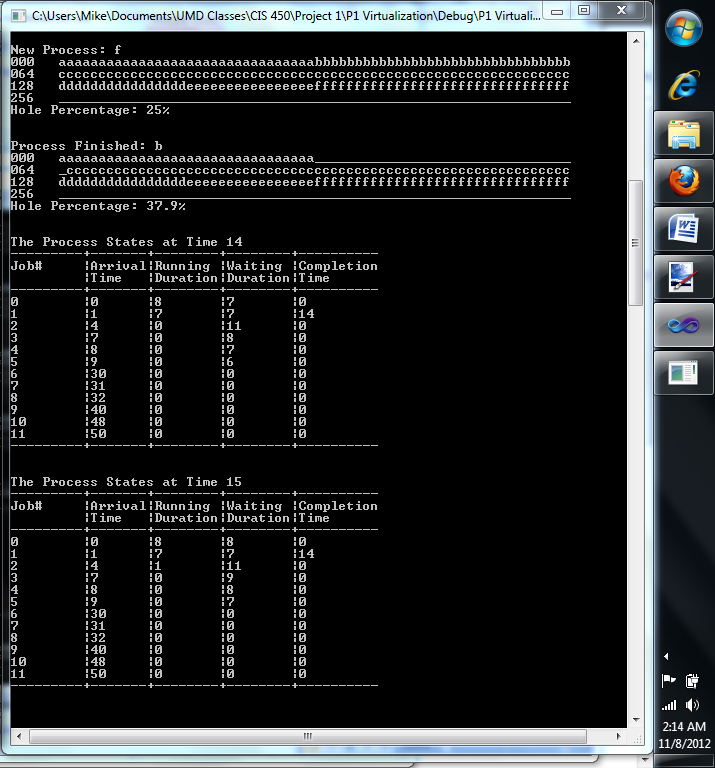
Next-Fit

To test the basic functionality of the program and the next-fit algorithm, I used the following data.

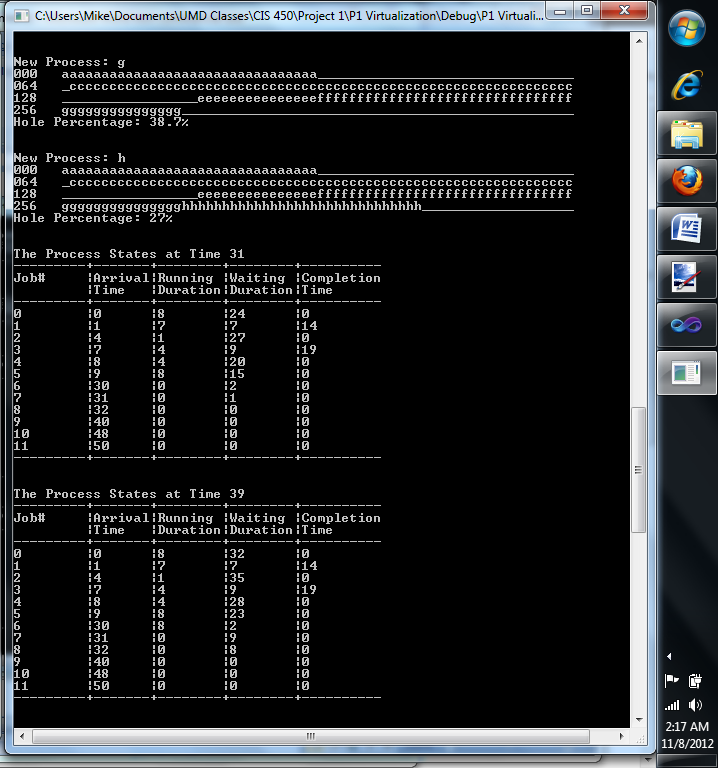
|  |  |  |  |
| --- | --- | --- | --- |
| **Job Number/Char ID** | **Arrival Time** | **Service Time** | **Address Space Size** |
| 0 / a | 0 | 30 | 32 |
| 1 / b | 1 | 7 | 32 |
| 2 / c | 4 | 25 | 64 |
| 3 / d | 7 | 4 | 16 |
| 4 / e | 8 | 20 | 16 |
| 5 / f | 9 | 25 | 32 |
| 6 / g | 30 | 20 | 15 |
| 7 / h | 31 | 30 | 30 |
| 8 / i | 32 | 11 | 62 |
| 9 / j | 40 | 8 | 12 |
| 10 / k | 48 | 6 | 9 |
| 11 / l | 50 | 22 | 28 |



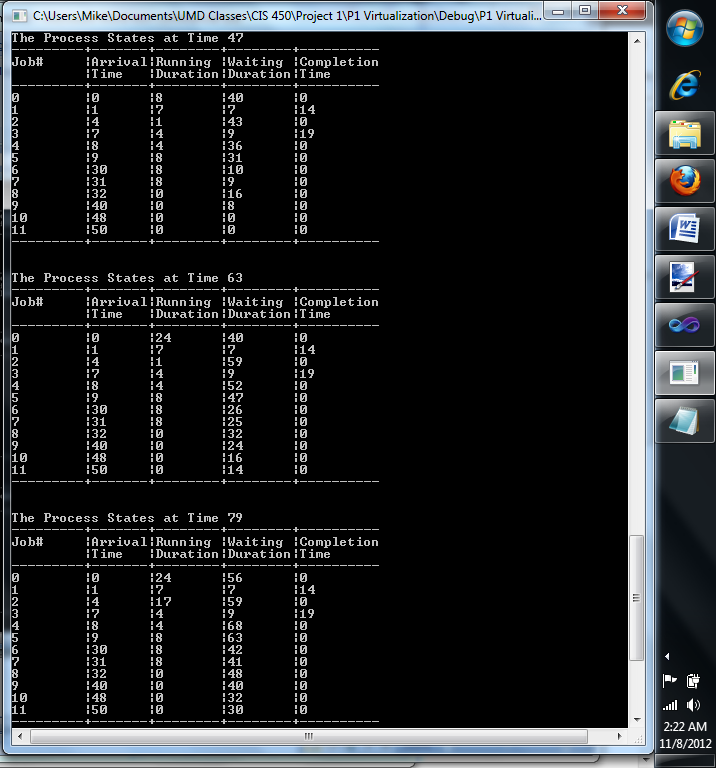
As seen above, the first several jobs are inserted one after the other. Little searching of memory is taking place, since the memory pointer remains at the last process inserted. As soon as it starts moving forward, it comes across the large empty block leading to the end of the memory space. No holes are forming yet and only job a has received CPU time up until clock tick 7. Jobs b, c, and d however, are in memory and are accruing wait time. You can also see that the Hole Percentage is slowly decreasing as more processes make their way into memory.



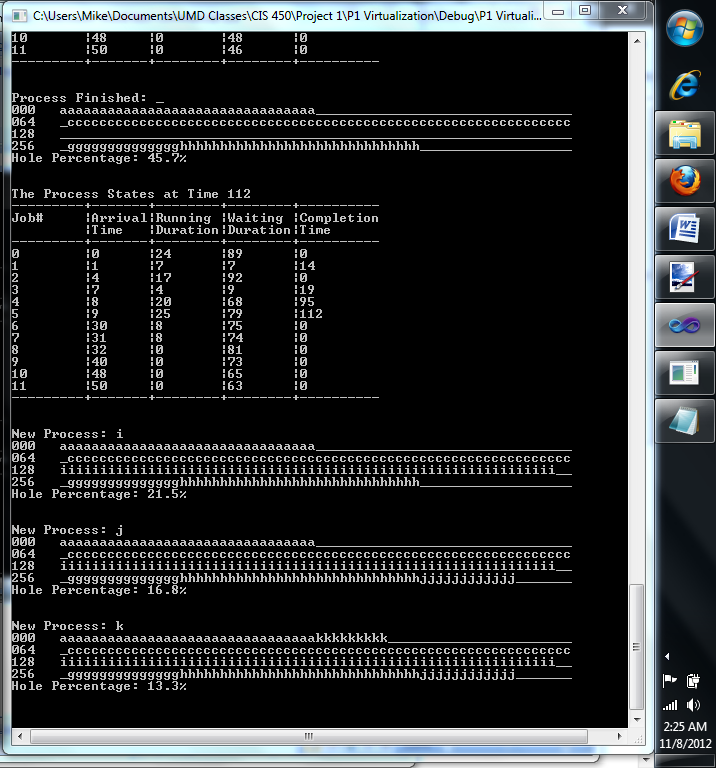
Here, you can see that job 1/a has used up its level 1 time slice and job 2/b takes over CPU time. Job b runs to completion in 7 clock ticks and is removed from memory. It's completion time is recorded at 14. With job 1/a now at priority level 2, job c begins receiving CPU time.



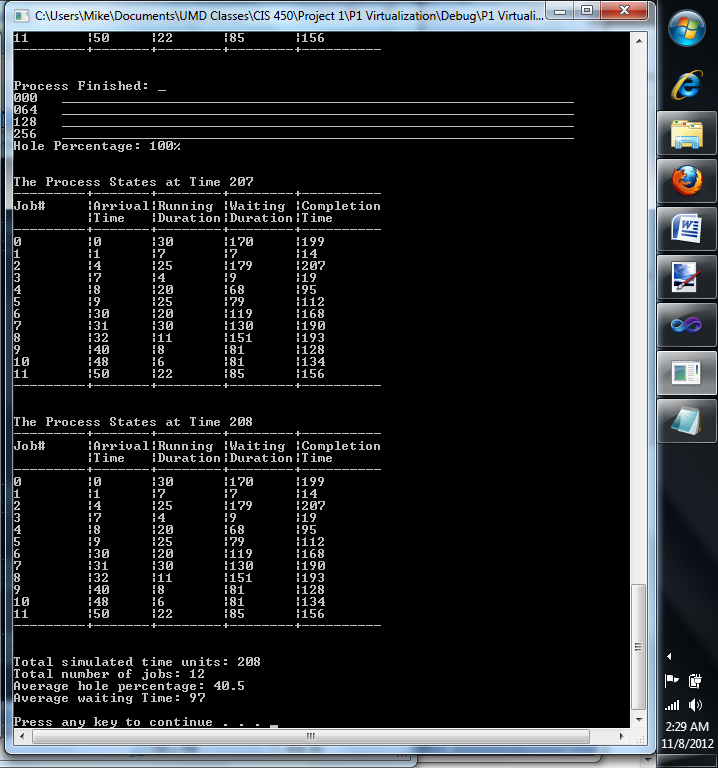
You can see here that the simulator has reached time 39, however job 8/i is not in memory. Because this job has a rather large address space size (62 units) it cannot fit into memory at the moment and sits in the wait queue.



By now jobs j and k have become ready, and while there are large enough blocks of space to accommodate them, job still sits at the top of the wait queue, blocking them. We can also see that 16 clock ticks are taking place between each context switch. This shows that all jobs are at priority level 2 or lower.



With jobs d, e, and f all completed, they are removed from memory, leaving enough space for i, j and k to come in, all within the same clock tick. Job k is the first to not find space moving towards the end of the memory space, and it ends up cycling back to just after job a's address space.

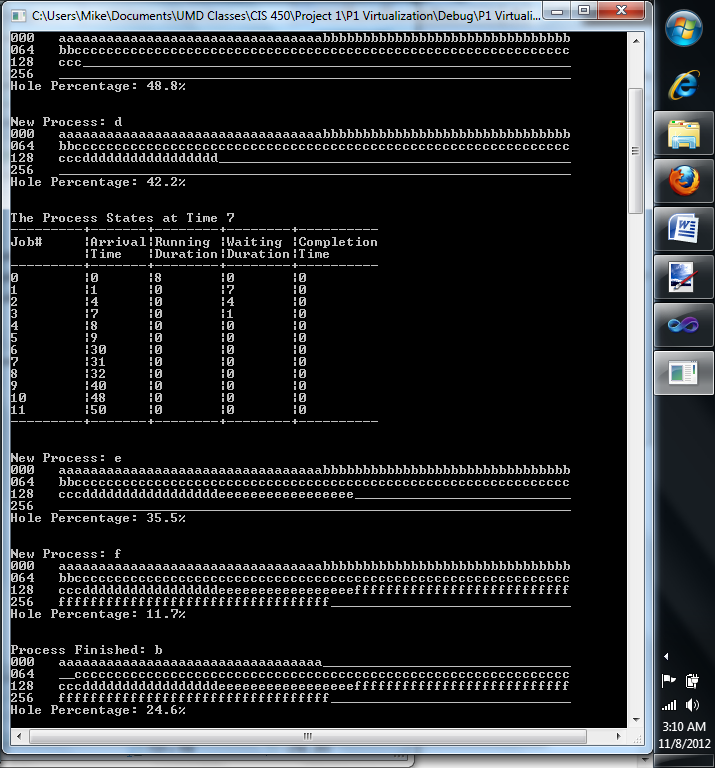


Finally, c is the last job to finish at time 207. Unfortunately, the way I built the system, there is still one more clock tick before the scheduler recognizes that all jobs are finished. A final list of data for all jobs is outputted, along with averages for hole percentage and waiting time.

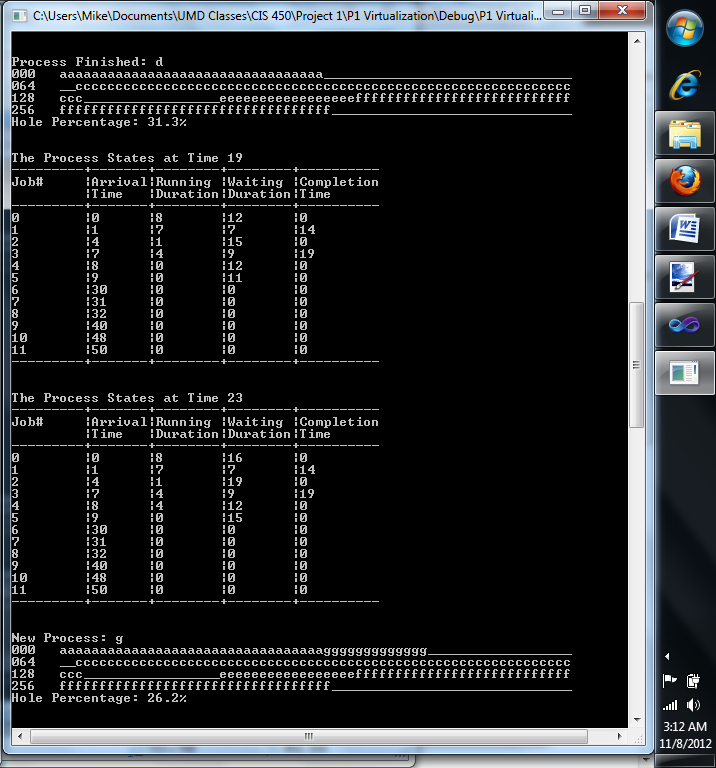
Worst-Fit

To demonstrate the worst fit algorithm, I used close to the same set of data. However I tweaked the address space sizes for jobs 5/f and 6/e in order to show how processes are placed differently into memory than in Next-Fit.

|  |  |  |  |
| --- | --- | --- | --- |
| **Job Number/Char ID** | **Arrival Time** | **Service Time** | **Address Space Size** |
| 0 / a | 0 | 30 | 32 |
| 1 / b | 1 | 7 | 32 |
| 2 / c | 4 | 25 | 64 |
| 3 / d | 7 | 4 | 16 |
| 4 / e | 8 | 20 | 16 |
| 5 / f | 9 | 25 | **60** |
| 6 / g | 30 | 20 | **12** |
| 7 / h | 31 | 30 | 30 |
| 8 / i | 32 | 11 | 62 |
| 9 / j | 40 | 8 | 12 |
| 10 / k | 48 | 6 | 9 |
| 11 / l | 50 | 22 | 28 |



The Worst-Fit algorithm begins the same as before. The only difference is it holds no pointer to the last placed process. Instead it searches through the list start to finish and keeps track of the largest open block of memory. This is then compared to the size of the next address space to see if it will fit.



Here we can see how the Worst-Fit algorithm differs. Job g becomes ready and can fit into any of the three holes currently open. Under Next-Fit, it would have been placed just after job f as that is the first block the pointer would have arrived at. Instead, the empty space after job a is determined to be the largest available, so g is placed there.

**Conclusion**

What this project may have lacked in difficult coding techniques, it certainly made up for in scale. There are a lot of systems working in place just for this simple simulation. In trying to code it using proper object-oriented practices, I ran into several problems debugging it. I definitely gained an appreciation for how complicated and yet efficient of a system it is that governs processes in memory and CPU time.