Simulation of a simple OS

Unit: Operating Systems

CS17035
Spring 2019

Computer Science Department of
City College, The International faculty of The
University of Sheffield

Abstract

This report serves the purpose of walking someone through the experience gained from implementing part as well as testing a Simple Operating System in Java with a Processing-based GUI.

Table of Contents

1. Introduction

During the semester, we got the chance to familiarize ourselves with many different operating systems' mechanisms. This coursework was on a simple operating system simulator, developed in java. The simulator included a memory manager and a process scheduler developed by our lecturer, namely First Fit and First Come First Served respectively. Our job was to develop another combination of a memory manager and process scheduler, as well as all the other functions that are needed for our algorithms to work and finally conduct experiments, comparing our implementations to the ones that were already included in the simulator code library.

2. Algorithms chosen

My choice of algorithms was the Next Fit algorithm for a memory manager and the Shortest Remaining Time as a process scheduler. I figured that since Next Fit is a variation of First Fit, which was already included in the simulator, I could actually extract some useful information about the effects of their main difference: Where they begin the search for empty space. First Fit is designed so that it begins searching from the beginning of the ram till the end every time [1], while Next Fit continues from the last place it found a position to place a process [2], virtually moving the starting point of the search all around the ram. This could theoretically lead to more suitable free partitions of adequate size being discovered, as the algorithm only searches ahead of where it previously worked, while the previous space is almost for sure unchanged or about to change.

My choice for a process scheduler was more of intuitive than "well-thought". I wanted to observe the effect of picking the smallest amount of work left to be executed first, a well established tactic in network related procedures, in action [3]. Since the implementation of this algorithm is based around this concept, it also meant that it could be fairly easy to implement, as estimating the time a process might need to run, not taking the random factor of process blocking into consideration, is a straightforward method of scheduling processes when we assume that our computer runs one simple instruction per tick at all times. The very way the simulator works would make the process a lot easier for that matter.

3. Implementation technicalities

The data structure chosen for the implementation of all my algorithms was that of an ArrayList. This is the resizable array version of a List data structure. I made this choice because it is an easy built-in structure for managing custom objects (the ones that were implemented in the library) as well as its performance benefits due to the fact it is implemented with arrays [4].

The rest of the procedure up to the point of implementing the algorithms themselves was all about following the interface in implementing the methods. This was a straightforward procedure as it was basic object handling for the most part without a real challenge. That part also helped me revise the theory we have done in class, as how many of the functions work is taken out straight from the theory itself.

When I reached the main algorithm implementation state, I made sure I got done with the process scheduler first, as it was inarguably the least challenging of the two. The assumption I made while implementing that algorithm is that the shortest remaining time

can actually be calculated by subtracting the current counter of a process from its total length. While the libraries provided us with a percentage-completed-related method, my estimations were that this would not be the best way to go about finding the remaining time of the execution of a process, as a percentage does not take into account the amount of instructions left but how many they are in comparison to the size of the process. This could mean that a big process that had progressed further into its instructions could actually take longer time to finish than a smaller process with fewer instructions ahead. Another thing that should also be taken into consideration is that, since we are talking about a percentage, we are also talking about a division inevitably, which is an operation with more overhead than a simple addition (subtraction in this case) for a computer, therefore a subtraction-only algorithm could potentially be a lot faster as the operations increased (meaning the number of processes needed to be checked). My algorithm was basically that of picking the first process and then traversing through all the processes, each time checking if the process at hand has a smaller remaining time than the process in the "minimum" slot. If it indeed did, it would be put in the minimum slot. This leaves us with the process with the minimum remaining time in the minimum slot, which is then passed as the process of choice. As highlighted in comments in my source code, there is room for future work and improvement: we could keep the shortest remaining time processes sorted in a queue, so we don't have the overhead of looking over all the processes again but rather updating the sorting based on what processes actually got executed. This would of course waste a bit of ram to keep the extra queue data structure for our JVM simulator. Since the worst case scenario is the fact of randomness of when the processes wish to block, which is part of their codesize to begin with, the queue-based solution explored here could prove more effective. Tracking how many times the processes want to block is also a possible alternative, but that would generate far higher processing overhead as well as resource hogging, that would be pointless in environments with too few processes. Last important note, the way I handled preemption was an attempt to waste less CPU ticks when only one process is left at the process table. Preemption is enabled when more than one process are in the process table while it's disabled when only one is there.

```
private boolean preemptFlag;


  public SOSProcess selectProcessToRun() {
    if (this.queue.isEmpty()) {
      return null;
    }
    else if (this.queue.size() == 1) {
      this.preemptFlag = false;
      return (SOSProcess) this.queue.get(0);
    }
    else{
    SOSProcess p = (SOSProcess)this.queue.get(0);
    for (int i = 1; i < this.queue.size(); i++) {
      if ((((SOSProcess)this.queue.get(i)).getCodeSize() -
((SOSProcess)this.queue.get(i)).getCounter()) < (p.getCodeSize() -
p.getCounter())) {
      p = (SOSProcess)this.queue.get(i);
      }
    }
    this.preemptFlag = true;
    return p;
    }
```

```
}


public boolean preempt() {
  return preemptFlag;
}
```

Code Snippet 1 : Preemption Handling

As for the memory management, my implementation of Next Fit started by re-implementing First Fit. After that the appropriate modifications to the First Fit algorithm were made, I ended up with the Next Fit algorithm. Essentially, the modifications were 2 major ones: I kept the last position my algorithm was at as an extra memory manager attribute. Then I used this instead of setting the position to zero each time I run the algorithm and also used it throughout all the processes of the algorithm as the main position signifier. Finally, when discovering partitions, if the algorithm reached the end on the ram table, I made sure to reset the last position to 0 so it would continue from all the way back next time. This was only challenging as a strategy because I essentially had to think through two algorithms to come up with the final result. The rest about the memory manager implementation did not depend on heavy assumptions being made. For instance, the only thing that was taken special care of was the casting of datatypes to make sure they would work as I did not have enough time for debugging close to the end of the final week of development, therefore safer actions were preferred.

4. Testing for the truth

The testing that took place to figure out the differences between the algorithms started off as trying to be as diverse in covering them as possible. I get two possible ram size configurations, 512 and 1024 megabytes, and then replicated them across all possible combinations of algorithms. First tests were those of the built-in first come first served process scheduler along the First Fit memory manager, just to get a basic idea of what the defaults actually looked like in both ram sizes. Then I switched the memory manager to Next Fit. At smaller ram sizes, the performance of Next Fit was not the one anticipated, because it turned out to have more fragmentation than First Fit as well as higher times, seemingly making the system perform worse. But as the size of ram increased, First Fit quickly came leveled with Next Fit in terms of process management times and actually surpassed it in terms of fragmentation.

Moving on to my implementation of the shortest remaining time algorithm, in the smaller ram module the First Fit algorithm, although reporting back the same management times with Next Fit, needed to compact once and had higher average fragmentation rates. As we switched to the larger ram module though, First Fit actually beat Next Fit with lower fragmentation rates and the same management times.

While the results that came out where interesting they certainly were perplexing as well. While there were no clear observations about the memory managers, which might be the case because Next Fit is just a modification of First Fit and the processes were not that diverse, there was clear evidence that the process scheduler I developed was a lot faster than the first come first served one in every scenario.

5. Conclusion

To conclude, during this coursework we managed to get a deeper knowledge about algorithms used in operating systems for low level tasks. The implementation of such algorithms gave us an idea of how a slight difference in one's code can make a big difference in the final resulting performance. By going through debugging of our methods, there were certainly a lot of times we made critical decisions about the performance and effectiveness of our code. This, coupled with the theory that we had to study in order to achieve it gave us a comprehensive insight into real world programming decisions that could affect many people. While this was a mere simulation, it was enough to puzzle us and give us food for thought in exploring software development in a more detail-driven approach.

References

[1] GeeksforGeeks. (2019). Program for First Fit algorithm in Memory Management - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/program-first-fit-algorithm-memory-management/ [Accessed 19 May 2019].

[2] GeeksforGeeks. (2019). Operating System | Program for Next Fit algorithm in Memory Management - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/program-next-fit-algorithm-memory-management/ [Accessed 21 May 2019].

[3] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal, 'Size-based scheduling to improve web performance', ACM Transactions on Computer Systems, vol. 21, no. 2, pp. 207–233, May 2003 [Online]. Available at: http://dx.doi.org/10.1145/762483.762486 [Accessed 20 May 2019].

[4] Docs.oracle.com. (2019). ArrayList (Java Platform SE 8). [online] Available at: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html [Accessed 18 May 2019].