Process Scheduling Algorithms

How varying conditions can affect the performance of the scheduling algorithm

Drew Schulte, Jonah Yates, Selorm Sosuh
CS 3800 - Operating Systems
Professor Wilkerson
April 25, 2022

***Abstract* – This work explores the effects of: the process selection function, number of cores, number of processes, and process length on the occurrence of starvation in a given system. The algorithms in question are First-In First-Out (FIFO), Round Robin (RR), Shortest Process Next (SPN), Shortest Remaining Time (SRT), and Highest Response Ratio Next (HRRN). These are examined with respect to starvation.**

I. BACKGROUND

The importance of computers in the 21st century has drastically skyrocketed. Their advantages primarily being the ability to perform many complex functions for the user quickly. However, with developments in computers, comes the user assigning the computer tasks of increasing complexity and quantity, at rates equal to if not greater than computational power. This leads to an important part of computers, the operating system, which is in charge of dictating when each of these tasks, called processes, are given access to the CPU to be computed. The CPU does this using a selection function which is an algorithm that determines what process is allowed access to the CPU next[1].

II. INTRODUCTION

For our project[2] we wanted to examine process scheduling and the impact the selection algorithm has on process throughput, given a number of chosen variables. Throughout this paper, further context and explanations will be provided for those who aren't familiar with the concept of process scheduling.

In almost every computer nowadays, a technique called multiprocessing is used. Multiprocessing is the assignment of processes, which are chunks of a given task being asked of the computer, to a CPU. The CPU, which is the brains of the computer, will then execute said process. Prior to the utilization of multiprocessing, the a single CPU would only be able to work on a single task at a time, running one program at a time. Once given CPU access, the program would run till its completion. This was generally a bad thing since the program would be holding down the CPU, even if for a section of the program CPU access was not needed. Hence, the introduction of multiprocessing as a solution. This allowed for computers to run more than one process at a single time. Allowing for multiple processes to be assigned a CPU to be ran on, improved throughput by ensuring high CPU usage. This was expecially beneficial on multi-core systems where each of the CPUs would be able to select from the processes that need to be ran or on when dealing with tasks containing high I/O demands. Furthermore, we also wanted to study the impact that other changing other variables related to processes had on the overall throughout of completing processes. For that, we also manipulated the number of cores available, the number of processes and the length of the processes. Our dependent variable to measure these changes was starvations. By the end of our programs execution, we had generated 2400 data points when testing the 5 algorithms.

In order to determine the best optimal scheduling algorithm, the number of starvations was used as our metric for comparison. In computer science, starvation occurs when a process fails to complete its execution in an anticipated time. This can be due to a number of factors like number of processes, whether there was high I/O demand of each process or whether each process utilized the CPU or other resources needed for a long period of time. For our project we determined starvation as occurring when the process fails to complete 3 times the length of the process + its ready time. If a processes did have any I/O events it needed to run, its anticipated ending time would be increased by the number of I/O events it has times the average I/O event length. For example let's say the ready time for a process was 10 and its length is 8. The expected end time of that process would be 10 + (3*8) = 34. Since this process has no I/O events, there would be no additional values added to this expected end time. This means that if the process isn't marked as done when the simulation time reaches 34, it is marked as starved and ignored in future scheduling as in terms of our simulation it didn't have time to complete.

Since increasing the number of cores in our simulation, increases the number of processes that can simultaneously be executed, we predict that as the number of cores increases, the number of processes that starve will decrease. Additionally we think that increasing the number of processes and their length will lead to an increase in starvations. However, we are unsure as to how well each scheduling algorithm will perform under different number of CPUs, number of processes, and process lengths. Based on class knowledge, its expected that Shortest-Process-Next (SPN) will fair better when dealing with a few short processes while Round

Robin will have a bad throughput when dealing with many processes.

Finally, understanding the different states a process can be in is also important. We have already discussed at length the starvation state, a point for our purposes where a process can't be brought back from, and briefly touched on the done state that processes enter when they've completed their execution. The next state we want to talk about is the ready state. In our simulation processes will enter the start/ready state when the simulation time passes a processes' randomly generated start time. For example if the start time of a process is 10 then it will be marked as ready once the system hits time = 10. Prior to this the process will exist in the notArrived state. Since a process cannot be ran unless it is ready, there is a possibility that no processes are ready leading to the CPU(s) waiting until a process gets marked as ready.

III. SCHEDULING ALGORITHMS

The first scheduling algorithm that we will be discussing is First-In First-Out (FIFO). This means that whatever process has the earliest start/ready time will get run first. FIFO is also a non-preemptive algorithm meaning previously selected processes will continue to be selected until their completion. For a list of very few processes this algorithm is ok at best because it is not that complicated which means there is less room for error, but there is a much higher risk for starvation when the number of processes increases.

The second scheduling algorithm is Round Robin (RR) scheduling. The way that this works is alternating between the processes ready to run and processing each for a specified slice. The first thing that this algorithm will do is go through the list of processes and see which ones are ready. If they are ready then they will be put in the queue. The second step is going through the queue to take out any unnecessary processes. For example we can go ahead and remove done and starved processes because they do not need to be scheduled. After removing said processes, the queue spots get updated and the order gets shifted. A given number of processes are then selected and allowed to run for a time slice. This will cycle through all the processes in the list ensuring each processes gets access to CPU(s). However if at any point a process gets I/O blocked, or uses up all of its time slice, it is moved to the back of the queue.

Another algorithm that we want to implement in our testing is Shortest-Process-Next (SPN). The way that this algorithm works is that it chooses the process that has the least amount of processing time required to run. This allows the shorter processes that may get added later to leapfrog over the longer processes. It is still pretty effective when having a mixture of lengths because a lot of smaller processes run earlier leading to high throughput of short processes. This algorithm is not preemptive meaning once selected, a process will run to its completion. At the end of the day, it still faces some issues and could lead to the starvation of all long processes.

Shortest-Remaining-Time (SRT) is our next algorithm that we used prioritizes processes that have the least amount of time left to run. It is similar to SPN but if a long process has less time left it will select that one over any others.

The last algorithm that was tested is Highest-Response-Ratio-Next (HRRN). This one is more optimal than previous some of the previous algorithms talked about. Before any process gets chosen, the response ratio is calculated for each process. Its equation is[1]:

(Wait Time + Required Time) / Required Time

This leads to short processes being favored at first, however, over time the priority of longer processes will increase as its wait time does. Processes are selected based on their HRRN value, which higher values having higher priority. This allows a longer process who has been waiting for longer a chance to be ran over a shorter process that just became ready.

IV. IMPLEMENTATION

For this project, we wanted to code a simulation of an operating system selecting processes for a number of CPUs to run. This was done in c++, building off the first HW assignment and saving data to .csv files. We then wanted to experiment with using python to read data from our .csv files and create graphical representations via matplotlib of the data. We first needed a data type called a *process* and gave it the necessary attributes for our simulation to schedule them. This is shown below in Figure 1.

```
struct Process {
    Process() : state(notArrived), id(-1), readyTime(-1), doneTime(-1), reqTime(0), processingTime(0), waitTime(0),
                qPos(-1), slice(0), ioBlocked(false), isSelected(false){}

    State state;                // current process state

    int id;                     // unique id (order of process creation) given to a process for identification

    long readyTime;             // when a process should be ready
    long doneTime;              // time when the process enters done state
    long reqTime;               // required amount of process time needed to mark a process as done
    long processingTime;        // time spent running a process so far
    long waitTime;              // time process has spent in ready state

    // RR vars
    int qPos;                   // position in the current queue the process is at
    short slice;                // how far through the time slice the process is
    bool ioBlocked;             // if proc got blocked from an io event this last time unit
    bool isSelected;            // if the proc is selected

    vector<IOEvent> ioEvents;
    IOEvent currentEvent;
};
```

Figure 1. - Process Struct

This struct allowed us to monitor the different states, when it gets marked as ready or done, as well as when using multiple cores show if it has been selected for a core already. This means if we go into a

loop looking for 3 processes for our 3 cores, it will select 3 different processes rather than the same one 3 times.

We also wanted to keep it realistic so we used the so we utilized normal_distributions via #include random to allow for the generation of random values in a normal distribution. This enabled allowed us to specify approximately how many processes would have I/O events as well as the event's length by manipulating the mean and standard deviation of these distributions.

For our main function, we set up for loops so that we could run the program once and get all of our data. Each for loop pertained to a variable we were changing: number of cores, number of processes, and the length of the process as well as which of the 5 algorithms was being ran. For the cores, we ran data on 1, 2, 4, 8, 12, and 16 cores. For the number of processes we started at 4 and increased to 2048, doubling each time whale for process length we started at 4 but increased via doubling up to 512. Inside all of these loops was an output statement to the .csv file ensuring we got 1 datapoint everytime a variable changed.

Once we ran the simulation to get our data, we imported it into python using the pandas library to read_csv() and then stored it in a dataframe. Dataframes, which model SQL databases, then had selection functions ran on them to select a new dataframe containing only the applicable data for a given graph. The matplotlib library was then used to graph a 3D graph of the respective dataframe and save the graph. The functionality we used to do this is located below in Figure 2.

```
for x in range (6):          # values 0-5
    for y in range (5):      # values 0-4

        # getting the data
        data = pd.read_csv(input[x], header = 0)

        numProc = data[data['Algorithm'] == y]
        numProc = numProc['Number of Processes']

        procLen = data[(data['Algorithm'] == y)]
        procLen = procLen['Average Process Length']

        starvations = data[(data['Algorithm'] == y)]
        starvations = starvations['Number of Starvations']

        # creating the graph
        axis = plt.axes(projection = '3d')
        axis.plot_trisurf(numProc, procLen, starvations, cmap=plt.cm.Spectral, linewidth=0.2)

        # graph stylization
        axis.set_xlabel("")
        axis.set_ylabel("")
        axis.set_xticks([])
        axis.set_yticks([])
        axis.set_zlabel("Number of Starvations", fontsize = 12)
        axis.set_zticks([0,512,1024,1536,2048])
        axis.view_init(15, -170)

        # saving the image ouput to the imgs folder
        plt.savefig(output[outputNum])
        axis.set_zlabel("")
        axis.set_zticks([])
        outputNum += 1
```

Figure 2. - Python Graphing Code

The above code shows what eventually allowed for 3-dimensional graphs to be created. Previous attempts were not as successful, as seen in Figure 3.

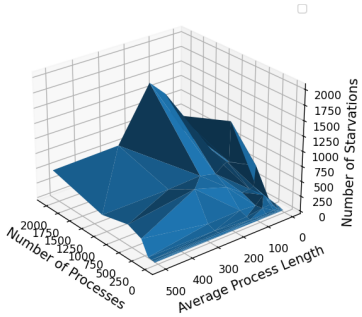

Figure 3. - Previous attempt at graphing in 3 dimensions using matplotlib

V. RESULTS & ANALYSIS

Before talking about the actual results of this project it should be good to go over some of the basic assumptions/predictions that happened before coding. When adding multiple cores the number of starvations should theoretically be lower since more and more processes can be run. Under that same logic it would be safe to assume that decreasing the number of processes and/or the length of the processes should also decrease the number of starvations. The higher the number of starvations means that less processes get run and the sorting algorithm has a lower efficiency.

For the most part it seems as if everything went according to plan. Figures 4 and 5 show the graphs from our results from the same algorithm but with different cores. The first graph shows it with 1 core and the second shows it with 16 cores.

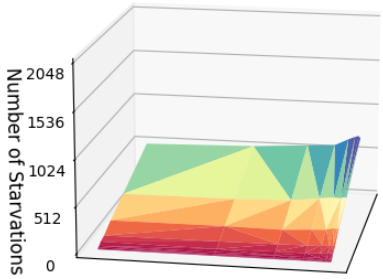

Figure 4.

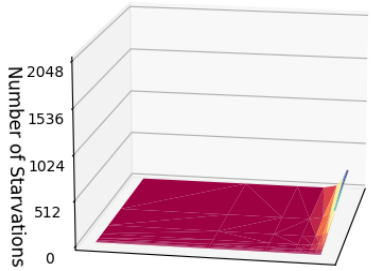

Figure 5.

As seen in the above figures, our assumptions were accurate. 1 cores vs 16 cores made a significant difference in the amount of starvations. Towards the right side of the graph we can see that the number of starvations has increased for both and that is mostly due to there being more processes that ran. Going deeper into the graph will show us the increase in length of the processes which is why it also increases in that direction. In our presentation we will go into greater detail of how each algorithm worked.

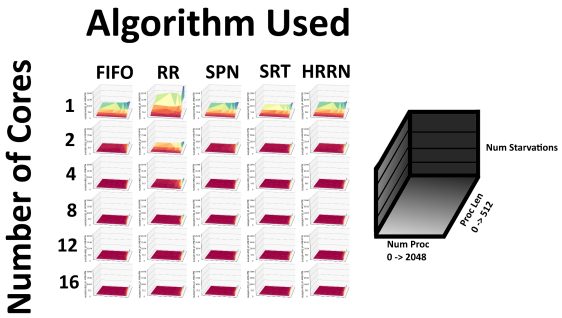

Figure 6.

Figure 6 shows all of the graphs and how the number of cores and the algorithm used impacted starvations. One thing that sticks out is that regardless of the algorithm, number of processes and process length, the number of starvations was generally high for 1 core. This is likely due to the fact that the 1 core was a bottleneck for all of the algorithms and prevented them from truely using the advantages they offer. Another important thing to look at is the performance of round robin. From the graphs, we can see that when round robin fails, it fails hard. When one process starves, its generally followed by a sequence of more starvations as all of the processes are around the same length and the time slice simply isn't large enough for them to all complete each iteration.

.

VI. TAKEAWAYS

Overall, everything in this project went as expected as far as results were concerned. There were a few small hiccups regarding learning how to create graphs in python and manipulating python dataframes. We also faces issues at first regarding implementing RR. In particular, creating and updating a queue that persisted past the scheduling algorithm's lifetime. At the end of the day this project was really exciting because we got to learn in depth the impacts various factors have on quickly executing processes.

VII. REFERENCES

[1] Class Notes - Processes
https://umsystem.instructure.com/courses/43634/pages/lecture-slides

[2] Github Repo - Project Code
https://git-classes.mst.edu/jynqz/operatingsystemsfinalproject