Project Goals

The goal of this project is to implement two scheduling policies, the shortest job first (SJF) and shortest remaining time first (SRTF), and evaluate their performance in terms of average job completion time ("average makespan"), average maximum waiting time, and average waiting time with the goal of preventing long-running jobs from causing small jobs to starve. Moreover, we implement optimizations to improve the policies' performance with respect to these metrics. We chose this topic because scheduling policies are a critical part of operating systems, as they facilitate wise utilization of CPU resources, increasing throughput (that is, processes executed to the number of jobs completed in a unit of time), waiting time, fairness, adaptability to varying workloads, and responsiveness.

The shortest job first (SJF) scheduling algorithm selects the processes with the smallest estimated burst times (time to finish the job) to execute next. It is non-preemptive, which means that once a job is taken out of the queue to run, the system must wait for that job to finish before adding a new process to execute. Advantages of this algorithm include: 1) reduced average waiting time, 2) better performance for batch-type processing where runtimes are known in advance, and 3) optimal average turnaround time (time to complete process). Some disadvantages of SJF are: 1) it is necessary to know the burst time for each job beforehand, as it is hard to predict, and 2) this policy can cause extremely long turnaround times or starvation[1]. For instance, if there is a steady supply of short processes, a long process may never get the chance to be executed by the processor, leading to a longer waiting and turnaround time for this long process.

The shortest remaining time first (SRTF) scheduling policy algorithm is the preemptive version of the shortest job first algorithm. As the name indicates, it selects the process with the smallest amount of time remaining to complete the process. Preemptive means that a process will continue running until its completion unless a new process is added to the scheduler that has a smaller burst time/remaining time. In such a scenario, the newly added process would replace the currently running process, the latter being placed into the queue to wait until its remaining time is smaller than all other jobs in order to obtain the processor again. Benefits of the SRTF include: 1) processes with shorter burst times are executed quickly; 2) minimal system overhead, as the system only needs to decide what process to execute when either a process completes execution or a new process is added to the scheduler. Some disadvantages of this scheduling algorithm are: 1) increased handling time, as context switch is performed frequently, which consumes CPU time; 2) like SJF, it has the potential for process starvation since it selects the shortest jobs first. If a shorter process is continuously added, then the longer processes may be held off indefinitely, increasing these processes' waiting and turnaround times[2].

Since both SJF and SRTF have the potential for process starvation, we implement an aging mechanism, which increases the priority of processes that wait in the queue for a long time (we define a threshold) to access the processor and execute to completion. We discuss this aging mechanism in detail in the next section.

[1] https://data-flair.training/blogs/shortest-job-first-sjf-scheduling-in-operating-system/
[2] https://www.educative.io/answers/introduction-to-shortest-remaining-time-first-srtf-algorithm

Design

The SJF and SRTF scheduling policies follow a similar design in terms of struct types and corresponding methods. This was intentional, as it improves portability between the two policies and allows for easier cross comparison during performance evaluation.

The schedulers rely on the use of two main data structures: a map where the keys are process ids and the values are corresponding pointers to a process struct storing various information on that process, and a queue which maintains priority ordering of processes to be executed. These data structures are stored in the fields of a policy struct, representing the scheduler itself. The structs also contain fields such as ‘processId’ which indicates the process that the scheduler (which could also be thought of as the CPU) is running right now and ‘remainingTime’ which indicates the time remaining in the process that is currently running.

Each version of the scheduler has its own run() function which takes in a process at some time t, simulating the arrival of a new task at the given time t. If the process passed into this function is empty, then it means that no new jobs are being added to the scheduler at this time. However, an empty process does not necessarily mean the CPU is idle (as there could be running tasks from previous calls of run() that are waiting to be executed). Due to this, run() calls also simulates the task which the CPU carries out at a given time (i.e what the CPU “runs” at time t). For the purpose of testing, the function operates on the assumption that the time parameter passed into the run function is unique (i.e. there cannot be multiple calls of run that contain the same time parameter).

Implementation

The general procedure of the run function is as follows. The function first checks if the process is empty. If it is not, we want to insert the process into the scheduler. To keep track of all processes, we add each new process into our dictionary of processes. Then, we need to decide where to insert the process in our existing process queue, and the criteria for selecting this position is dependent on the algorithm we are running. In the case that the queue is empty (i.e., this is our first process), we simply append the process to the queue. Then, regardless of whether there is a new process being added, the run function checks if there is a current process running and if it needs to select a new process to run (if more exist). This is done by checking if the current remaining time is equal to zero (which would mean that no process is currently running). If it is zero, we do a second check to see if there are more processes left in the queue to run. If there is, then we pop the first job (at index 0) from the queue (which is sorted from highest to lowest priority) and update the current process Id and remainingTime to simulate the running of this new process. At the same time, we also need to update the waiting times, and the calculation for this differs based on the algorithm we are trying to implement. Lastly, we need to decrement the time remaining in the process that is currently running, if one exists.

For SJF, we want to insert new processes at the first instance in the queue where the process’s burst time is greater than the new process’ burst time. Since SJF does not switch out processes after it decides to run it, we just need to set the waiting time of the process when it is run. Its waiting time would simply be the difference between its arrival time (stored in its field) and the current time (provided as a parameter of the run function).

The optimized version of SJF differs in that we need to increment the priority of every single process in the queue at each time step. Then, in deciding the next job to execute, we want to first identify the process with the maximum priority that is greater than the threshold. If one exists, then it means that the process has waited for too long, and we want to run that

immediately (as opposed to running the first job in the queue which is sorted by burst-time in ascending order; that is, the first job at index 0 contains the job with the smallest burst time).

For SRTF, we want to check if the new process' burst time is less than the time remaining in the process that is currently running. If it is, we want to run the new process (which is simulated by updating the process Id of the scheduler and setting the remaining time of the scheduler to be the burst time of the new process) and put the process running now back into the queue with the updated burst time. This check is sufficient to ensure that we are always running a process with the minimal remaining/burst time (since the old process would already have been one that had the least burst time). In terms of waiting time, since processes can be switched out after running for a bit, we only want to update a process' waiting time after it has been completely run (i.e. when its remaining time becomes 0), in which case its waiting time would simply be its current time subtracted by the sum of its arrival time and burst time.

The optimized version of SRTF differs in that instead of simply comparing new process' burst time with current process remaining time, it will also require that the current process' priority is less than the set threshold. If the current process' priority is greater than the threshold, then it means that the current process currently has a really high waiting time, which could be the result of it being switched out very often or that it's simply been waiting for too long; in either case, we would not want to switch out the current process for the new one.

Performance Evaluation Method

Within the implementation, there were a number of considerations to ensure a proper policy. Ideally, there should exist fairness, that long-running jobs shouldn't cause small jobs to starve (or disproportionately wait a long period of time). We evaluate this using the average waiting time and maximum waiting time. All the jobs should not have to wait a long time to be executed. Further, the GPU should not be idling (in other words that if a job can be run, it will be). Clearly, if the makespan, the total time to run all the jobs in the workload trace, is low, this is an indicator that the GPU is idling as little as possible. These metrics, in the real world, would ensure that jobs are completed efficiently.

We decided to compare each policy (SJF and SRTF) against its optimization, which aimed to solve starvation via the aging mechanism (SJF_OPT and SRTF_OPT). There are a few variables in regard to testing. One of them was deciding the optimal threshold, which is the minimum priority a job reaches before it should be executed before jobs with shorter burst times. Second, we also needed to make assumptions about the range of values of burst times and waiting times, as well as the number of test cases to average over. Finally, we used the metrics of makespan, average waiting time, and maximum waiting time to evaluate the performance.

We hope to analyze and improve schedulers in close to real world scenarios, which often need to determine the schedule of a massive set of processes. Considering this scale, it is essential to account for cases where starvation becomes a prominent issue– for example, when outliers of jobs waiting reach near-infinite time units. To address the issue of scale, we decided to automate the creation of test cases, rather than via hard-coding to allow for large test cases. We found it to be most efficient to code using randomized integers and apply the ranges there.

In particular, we have chosen that burst times range [1,101] and waiting times [0,10], and 100 jobs per test case, for a total of10 test cases. We then take the average waiting time, average max waiting time, and average makespan from the 10 test cases and compare against the performance of the optimization to evaluate how successful the optimization is.

Results

SJF vs. SJF_OPT

Threshold	Average Waiting Time		Average Max Waiting Time		Average Makespan	
	SJF	SJF_OPT	SJF	SJF_OPT	SJF	SJF_OPT
100	1365.08	2133.86	4587.20	4307.00	4920.80	4920.80
1000	1397.94	1535.62	4625.80	4328.90	4947.80	4947.80
4000	1448.89	1449.77	4670.40	4456.30	5044.50	5044.50
4500	1427.43	1427.68	4677.80	4559.70	5019.00	5019.00

As seen above, we have found that for SJF, a non-preemptive scheduling policy, the optimization significantly improves maximum waiting time regardless of threshold. Additionally, the makespans are identical across the two versions, indicating that both equally use the GPU in terms of efficiency, or reduce idling.

In our test scenario, as in a real scenario, there are points in time where some jobs come along which have been waiting for significant amounts of time as a result of their high burst times; they are regularly obstructed from the beginning of the queue by newer jobs which have lower burst times. These jobs' waiting times increase the maximum waiting time. Most importantly, these jobs may never run in a real life situation, when there is no "end" to job arrivals. With the aging mechanism optimization, the outlier values – these high waiting times – are reduced in size (as jobs are forcibly pulled to the front), thus the maximum waiting time decreases. We see this happening here for SJF.

There is, however, a nuance in that there is an interesting balance to be positioned for the other shorter jobs which have to wait for this long job to finish running. While the longer job's waiting time is shortened, a number of jobs have their waiting times increased that would not have occurred without this optimization. Therefore, the optimal threshold is not simply an arbitrarily high (or low) value, rather a value that must be curated and estimated. There doesn't exist a universal optimal threshold, and thus we ran multiple tests to find the threshold which does decrease average waiting time most significantly, which allowed us to situate on the value of 4500, as seen above, which yields nearly the same average waiting time as the non-optimized version but with a lower average max waiting time.

Thus, the aging mechanism serves to improve the max waiting time, while maintaining roughly the same average waiting time and makespan. This is clearly an optimization on top of the original.

We now turn to our preemptive scheduling policy, SRTF.

SRTF vs. SRTF_OPT

Threshold	Average Waiting Time		Average Max Waiting Time		Average Makespan	
	SRTF	SRTF_OPT	SRTF	SRTF_OPT	SRTF	SRTF_OPT

100	1497.86	2290.32	4887.90	4616.30	5206.40	5206.40
1000	1450.45	1592.99	4748.80	4460.50	5081.50	5081.50
4000	1444.58	1445.61	4727.00	4473.10	5064.80	5064.80
4500	1479.41	1479.54	4773.80	4648.80	5102.70	5102.70

On the other hand, similarly to SJF, we see that the optimized SRTF is truly an optimization: the max waiting time is reduced while keeping the average waiting time and makespan the same. In other words, GPU utilization and fairness are still kept at high performance. We do see, however, that while the 4500 threshold may yield closer average waiting times, the improvement in max waiting time is reduced and SRTF may be better served with a threshold like 4000, where the difference in average waiting time is still not significant, but which has more improvement in the average maximum waiting time. We speculate that this is because a preemptive scheduling policy has more flexibility by permitting the interruption of jobs, but the aging was implemented such that the forcibly executed high-priority job was non-preemptive (else it would occur that the job would just be executed every other time unit), which has disrupted the preemptive nature of the policy, and has perhaps resulted in a reversal of an optimization.

Next Steps

There are other methods, besides aging mechanisms dealing with placing a priority on jobs, that could potentially work more effectively in reducing starvation for long processes in both the shortest job first and shortest remaining time first scheduling algorithms, which we would have liked to include given more time. One example is the application of machine learning to learn patterns from historical data about burst times. For instance, linear regression could be used to identify relevant features, such as CPU utilization, memory requirements and necessary operations to complete jobs within a certain domain/field, to more accurately estimate the burst time of such jobs[3].

[3] https://medium.com/@dixitvibhu20/enhancing-shortest-job-first-sjf-with-machine-learning-bridging-the-gap-between-theory-and-ee204a85ae1a

Links

Github Repo:
https://github.com/donnawang47/COS316-Final-Project

Video Demo Link:
https://drive.google.com/file/d/1Fmm95Sa13L-SFLLvr-_4kbBs2V6GY7i3/view?usp=drive_link