CS 533: Advanced Operating Systems Scheduling Simulator

Nandadeep Davuluru, Manpreet Bahl, Thomas Salata,
Saral Bhagat, Mrunal Hirve

June 13, 2018

1   Introduction

Operating systems have been on the rise since the 1970s. Different researchers have spent time analyzing the need for both monolithic and unikernel OSes tailoring them for specific needs. The most important function of a monolithic operating system is the ability to handle processes thrown at it. Specifically, the scheduler is a part of the operating system kernel that handles which process runs at a particular time. There has been a lot of work within this field and therefore, this paper doesn't focus on proposing any new ideas, but the scope is rather restricted to implementing a few popular scheduling algorithms to showcase and analyze their purpose real time. We implemented first in first out, round robin and the Completely Fair Scheduler (running on the Linux kernel) from scratch and visualized the results for different workloads. The rest of the paper is arranged as follows: background of the scheduling algorithms used, the design behind our simulator and processes, the methods used to conduct our simulation, results of our simulation, an analysis of the results, and finally some concluding thoughts.

2   Background

Before we discuss the simulation's design, it is necessary to understand the algorithms behind the different schedulers used within the simulation.

2.1   First In First Out

FIFO, also known as first come first serve (FCFS), is the simplest scheduling algorithm. It simply queues processes in the order that they arrive in the ready queue, commonly used for task queue, and runs them until they are complete [1].

2.2   Round Robin

The scheduler assigns a fixed time slice for the process to execute [1]. If the process completes within the time slice, it gets terminated, otherwise it is rescheduled after a chance is given to all other processes. Although the time slice does not need to be the same as the timer interrupt [1], in our simulation the timer interrupt that invokes the scheduler also acts as the time slice.

2.3   Completely Fair Scheduler

CFS is the default scheduler of the Linux Operating System, starting from Linux 2.6.23 [2]. It aims to maximize overall CPU utilization while also maximizing interactive performance. The data structure used for the scheduling algorithm is a red-black tree [2]. The idea is to try and mimic an ideal processor, such that if there are n processes running on a CPU with 100% power, then each process would get $\frac{100\%}{n}$ power and run in parallel [2]. As this ideal situation is non-existent, the CFS instead focuses on running tasks that have had little chance to run on the CPU and gives them a "fair" amount of CPU time before giving other processes a chance to run [2]. Choosing the amount of time to run is done by setting a bounded target time and then dividing that by the number of runnable processes [3]. CFS chooses the target time by considering how long it has been since the process was last executing, then dividing that time by the number of runnable processes [3]. Both of these methods for calculating how long a process gets to run are implemented in our simulator in CFS 1 and 2 respectively. Note that the CFS algorithm actually implemented in Linux additionally considers process priority and "nice" values for determining how long a process runs [3].

3 Design

The scheduler simulation is developed in Python 3.5 using the object oriented programming technique. In addition, we developed a web application for users to interact with the simulator through Python Flask.

Before we can talk about the schedulers' design, we need to talk about what a process is in our simulation. The Process class contains information about the process which includes:

- Process Identification Number (PID)

- Priority: Low or High (currently unused in our simulation)

- Start Time: The time unit in which the process first got time on CPU

- Burst Time: The total execution time of the process (in time units)

- Counter: The total time the process has been executing (in time units)

- Arrival Time: The time unit at which the process was added to the scheduler

- Completion Time: The time unit at which the process completes execution

- Turnaround Time: The total time it took for the process to complete. This is determined by *Completion Time - Arrival Time*

- Waiting Time: The total time the process was waiting for a chance to run. This is determined by *Turnaround Time - Burst Time*

- Status: Is the process complete or incomplete?

The process "runs" itself by just incrementing the counter. When the process is complete, it calculates completion time, turnaround time, and waiting time. The following snippet of code shows the implementation of process run:

```python
#"Run" the process object by incrementing the
#counter
#Params:
#   sysTime = the current system time
#Return:
#   status = either COMPLETE or INCOMPLETE

def run(self, sysTime):
    if self.counter == 0:
        self.start_time = sysTime

    self.counter += 1
    if self.counter == self.burst_time:
        print("COMPLETED," + str(self.getPid()) +
        ","+ str(sysTime))
        self.status = COMPLETE
        self.completion_time = sysTime
        self.turnaround_time = self.completion_time\
        - self.arrival_time
        self.waiting_time = self.turnaround_time
        - self.burst_time
        return COMPLETE
    else:
        self.status = INCOMPLETE
        return INCOMPLETE
```

Now that we have established what a process is in our simulation, we can talk about the scheduler design. The high level object-oriented design is shown below:
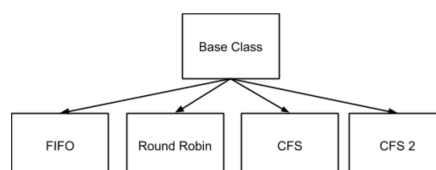
The base class acts as a machine with only a single core, running both user level processes as well as the scheduler. It contains functionalities that are common to all the derived schedulers: process queue, timer interrupt value, and a function that "runs" the scheduler itself. Each of the derived classes (FIFO, Round Robin, CFS 1 & 2) manage their own ready lists (tree in the case of CFS 1), manage how processes are added and removed from the ready lists, and provide the next process to run to the simulator. It's important to note that the base class simulates a machine with only a single core.

The FIFO class maintains a ready list by using a Python deque which is a high-performance list-like container with fast appends and pops on either end. Processes are added to the end of the deque and removed from the beginning utilizing the built-in functions that interact with the deque. The FIFO class determines which process it runs through a function which implements the actual heuristics of the algorithm. First, a check is made on whether the current process running has completed. If it has, then the next process to run is removed from the ready list. If it hasn't, then it continues to run. This can be shown in the code snippet below:

```
def getNext(self, curProc):
    if curProc is not None and curProc.get_status() == INCOMPLETE:
        return curProc
    elif curProc is not None and curProc.get_status() == COMPLETE:
        curProc = None
    return self.removeProcess()
```

The Round Robin class also utilizes a Python deque for its ready list with processes added to the end of the deque and removed from the beginning. As mentioned earlier, the timer interrupt value is also the time slice value for round robin and as such, when a timer interrupt occurs, Round Robin removes the next process to run from the deque. The process that was running gets added back to the ready list if it has not completed. The code snippet below shows the implemented round robin heuristics:

```
def getNext(self, curProc):
    if curProc is not None and curProc.get_status() == INCOMPLETE:
        self.addProcess(curProc)
    elif curProc is not None and curProc.get_status() == COMPLETE:
        curProc = None
    return self.removeProcess()
```

The CFS class uses a red-black tree to manage its ready "list." The red black tree implementation was found online [7] and with the permission of the owner, it was modified to meet our needs. The author's implementation didn't support duplicate keys for the node so we added a deque to each of the nodes to hold processes with the same key. The key represents the process execution time which is the total time that the process has run. With the addition of the deque, we added functions to the tree class itself on adding and removing processes and total process count in the tree. The CFS class also has a static target bound value that's set to the initial timer interrupt value. In CFS, the timer interrupt value is updated by dividing the target bound value by the number of runnable processes. CFS decides which process to run next by picking the process with the lowest execution time by traversing all the way to the left of the tree. The current process that was running is added back into the tree if it has not completed. The following snippet of code shows the implemented heuristic for CFS:

```
def getNext(self, curProc):
    if curProc is not None and curProc.get_status() == INCOMPLETE:
        self.addProcess(curProc)
    elif curProc is not None and curProc.get_status() == COMPLETE:
        curProc = None

    processCount = self.readyTree.getProcessesCount()

    if processCount > 0:
        self.timerInterrupt = math.ceil(self.targetBound / processCount)

    nextProc = self.removeProcess()
    return nextProc
```

The CFS 2 class follows the same idea as CFS 1 but instead of using a red-black tree, it uses a Python List. The reasons for using a list were purely for implementing and testing a slight variation of CFS 1 concurrently with development of CFS 1, while avoiding the issues of integrating and testing the red-black
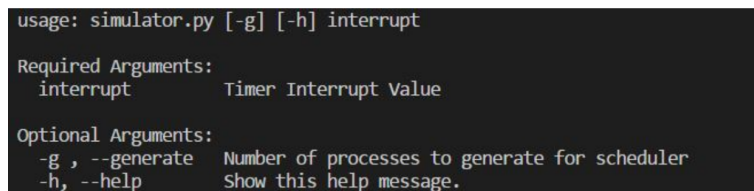
tree from [7]. In addition to maintaining a ready list, it also captures when a process last finished executing, so that it can later determine how long it has been waiting to run. While CFS 1 used a static bounded target, this implementation uses a dynamic bounded target time as described in [3]. It still begins by selecting a process from the ready list with the lowest runtime. Then it determines how much time has passed since it last ran and divides that by how many runnable processes are in the system. This value is then used to update the timer interrupt, so that the process will not execute beyond its "fair" share. This encapsulates the previously mentioned notion of processes sharing CPU power on an ideal processor. The code snippet below shows the implemented heuristics:

```python
def getNext(self, curProc):
    #Later calculation needs to accound for processes not
    #on the readyList (nextProc and possibly curProc)
    extraProc = 1
    if curProc is not None:
        extraProc = 2
    #Get the next process to run
    nextProc = self.removeProcess()
    #If no new process to add, then use current process
    if nextProc is None and curProc is not None:
        nextProc = curProc
        curProc = None
    #Calculate time new proc can run
    if nextProc is not None:
        time = self.systemTime - self.last_time_run[str(nextProc.getPid())]
        #Set interrupt using this time
        self.timerInterrupt = math.ceil(time / (len(self.readyList)+extraProc))
    #Add current process back into readyList
    if curProc is not None and curProc.get_status() != COMPLETE:
        self.readyList.append(curProc)
        self.last_time_run[str(curProc.getPid())] = self.systemTime
    return nextProc
```

The simulator can be run two ways: command line and web application. The user has to specify the timer interrupt value and an optional number of processes to generate (default is 100 processes). The image below shows how to run the simulator through the command line:
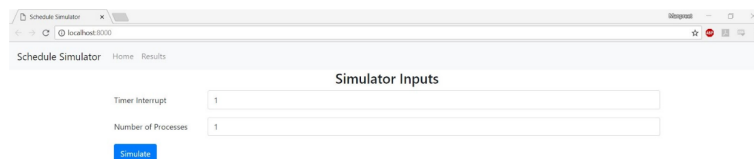

The web application is a basic Python Flask application that provides a frontend for the simulator. The user is able to provide the same parameters as the command line, however, the number of processes to generate is no longer optional. Once the simulation has finished, the user will be directed to a page containing the results of the simulation. The image below shows the home page of the simulator:

4   Methodology

As mentioned in the previous section, the user provides the timer interrupt value and the number of processes to generate for the simulation. The number of processes value is given as input to a random process generator which is used to create the specified number of random processes. Each process is given a random priority, a random start time and burst time between 0 and 200 time units. Once all the processes are generated, they are sorted by starting time to make analysis of the data easier. The code snippet below shows the code for the random process generator:

```
def generateProcess(num):
    #Start the process queue as a list
    processQ = list()
    priorities = ["High", "Low"]

    #Create random processes
    for i in range(1, num + 1):
        processQ.append(process.Process(i, random.choice(priorities),
                                        random.randint(BURST_TIME_START, BURST_TIME_END),
                                        random.randint(START_TIME_START, START_TIME_END)))

    #Sort the process queue by start time in ascending order
    processQ = sorted(processQ, key=lambda process: process.get_startTime())

    #Convert the sorted process list into a process queue before returning it
    return deque(processQ)
```

Once the processes are generated, four scheduler objects are created, one for each of the different schedulers. Each scheduler is given a copy of the same generated process list as input and outputs two CSV (Comma Separated Values) files: one for the overall summary for each of the processes and the other containing the execution logs. The CSV files are then analyzed and graphed using a Python script. There are two types of graphs that are generated. The first type contains the average waiting time, average turnaround time, and average response time and the second type is a timeline summary for each of the processes run by the schedulers.

5 Results

The simulation was run with 10 processes and a timer interrupt value of 20. We felt that this produced visually coherent graphs, while still capturing the fundamental behavior behind each of the schedulers.
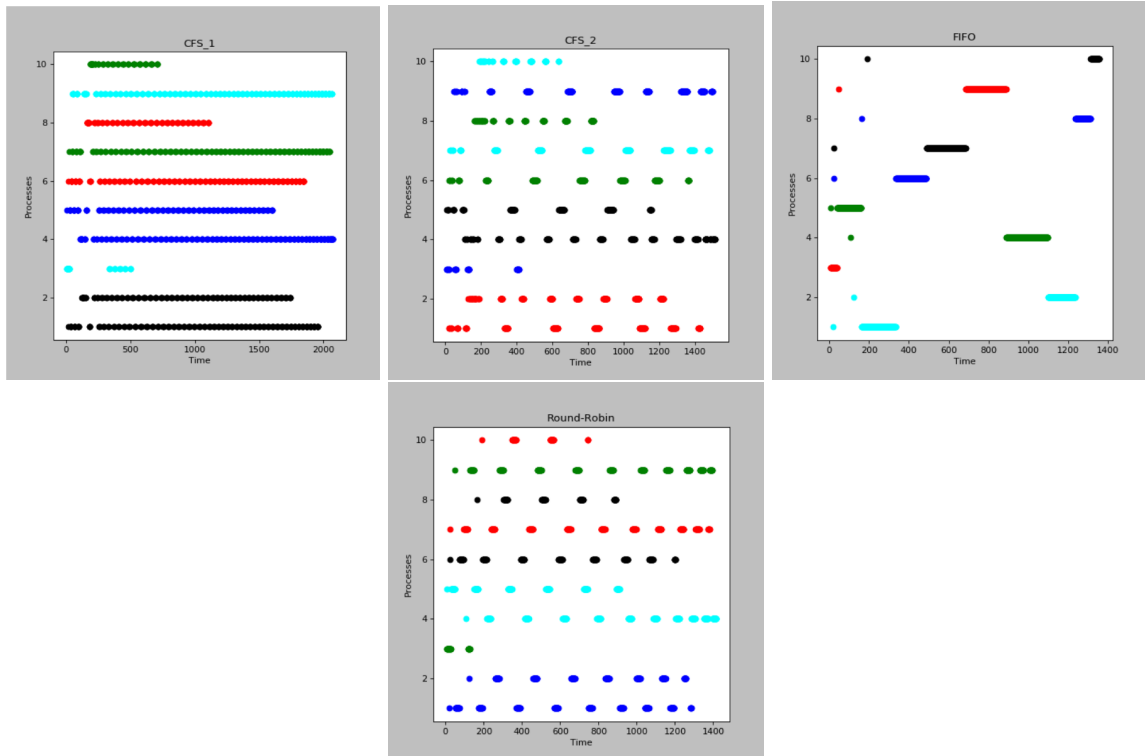

Figure 1: **Clockwise:** Results for CFS 1. Note the high number of context switches; Results for CFS 2. Dynamic target times result in fewer context switches; FIFO results: Although it finishes more quickly, process starvation occurs; Round Robin results: Better response times than FIFO, but much worse than CFS
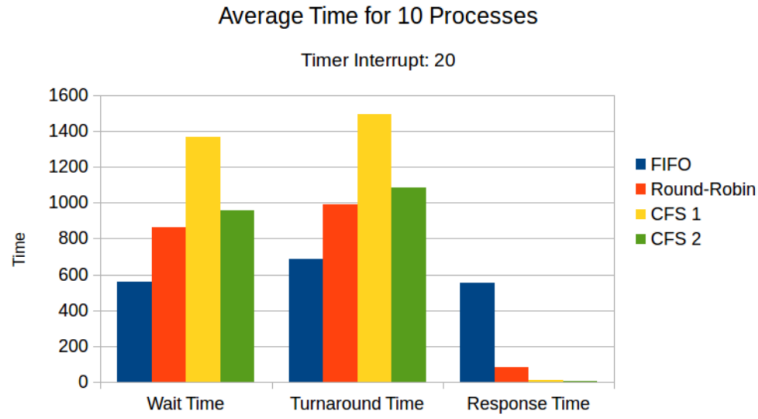
Figure 2: Average wait, turnaround, and response times for each of the four scheduler

Figure 1 in clockwise direction, show the execution timeline of processes for each of the schedulers. Note how any of the schedulers that included additional context switches between processes as a result of time constraints imposed by the scheduler also had their simulation run longer. This was a result of a cost imposed by the simulator on context switches to simulate things like needing to fetch from main memory or repopulate a TLB. Consequently, the more context switches a scheduler imposed, the longer the simulation would run. This in turn affected things like wait times and turnaround times for processes using those schedulers as well. This is particularly evident in both CFS implementations as seen in figure 2.

6   Analysis

Figures 1a shows the CFS implementation that relies on a static bounded target time to determine how long processes get to run. Once the scheduler is loaded with all of processes for the simulation, each process runs for a very small amount of time before the scheduler switches to a different process. In effect, this acts like a round robin algorithm with a small time slice, and consequently has very high overhead resulting from these switches. The second CFS implementation, seen in figure 1b, fairs slightly better in terms of reducing the number of context switches. Note that the second CFS implementation also gives processes that haven't had a chance to run for some time longer periods in which to run. Both implementations, however, have extremely fast response times, indicating that this algorithm would be particularly useful for interactive process loads. Both CFS implementations also suffered from higher wait and turnaround times resulting from the higher number of context switches they imposed.

In figure 1c, it is evident that the FIFO algorithm imposes very little overhead as context switches to new processes are only needed upon completion. This results in the simulation completing more quickly than with any of the other schedulers, thus the time a process spends on the system is reduced. This effect can be seen in the wait and turnaround times as they are relatively low compared to other schedulers which suffer from higher context switching overhead. However, FIFO becomes disadvantageous when it comes to allowing other processes a chance to run, reflected in the high response time.

Round robin, seen in figure 1d, fairs somewhere in between FIFO and CFS. It generally gives processes a longer amount of time to run per time slice compared to either CFS implementation. This led to processes completing slightly faster than CFS 2, but the added context switches caused it to still run slower overall than FIFO. As round robin utilizes a queue for a ready list, processes generally have a slower response time than CFS provides, but much faster than FIFO.

7   Conclusions

The purpose of this paper was to compare different task scheduling algorithms based on identified qualitative parameters. The First In First Out (FIFO) algorithm is recommended for minimizing average CPU utilization. The Round Robin scheduling algorithm is generally fair to every process, but imposes extra overhead and presents the difficulty of choosing an optimal time slice value. The Completely Fair scheduler (CFS), which is the default Linux scheduler, aims to maximize overall CPU utilization while also maximizing interactive performance. In conclusion, there is no universal "best" scheduling algorithm, and many operating systems use extended or combinations of scheduling algorithms. It all depends on the user needs and

objectives.

References

[1] Walpole, J. (2017). CS 510 TOP: Operating Systems Foundations, Scheduling [PowerPoint slides]. Retrieved May 15, 2018 from http://web.cecs.pdx.edu/ walpole/class/cs510/fall2017/slides/10.pdf

[2] CFS Scheduler. (n.d.). Retrieved May 15, 2018, from https://www.kernel.org /doc/Documentation/scheduler/sched-design-CFS.txt

[3] Ishkov, N. (2015). A complete guide to Linux scheduling (Unpublished master's thesis). University of Tampere. Retrieved May 15, 2018, from https://tampub.uta.fi/bitstream/handle/10024/96864/GRADU-1428493916.pdf

[4] Wikipedia contributors. (2018, May 31). Completely Fair Scheduler. In Wikipedia, The Free Encyclopedia. Retrieved June 10, 2018, from https://en.wikipedia.org/w/index.php?title=Completely_Fair _Scheduler&oldid=843812906

[5] Jones 15, M. T. (2009, December 15). Inside the Linux 2.6 Completely Fair Scheduler. Retrieved May 15, 2018, from https://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/

[6] Operating System — Process Management — CPU Scheduling. (2017, October 24). Retrieved May 15, 2018, from https://www.geeksforgeeks.org/gate-notes-operating-system-process-scheduling/

[7] Kozlovski, S. (2017, June 05). Enether/Red-Black-Tree. Retrieved May 23, 2018, from https://github.com/Enether/Red-Black-Tree