Roll-In Roll-Out Scheduling Algorithms

Laura M. Strickman

July 29, 2005

1 Abstract

This paper presents an algorithm for fairly and efficiently scheduling processes in a multi-process system. Current scheduling algorithms sacrifice either efficiency or fairness when the process load begins to exceed available memory: either the CPU sits idle while processes bring their pages in and out, or the processes run one at a time until finishing, which allows for full CPU utilization but means that the latter processes must wait a long time before beginning to run, and any idea of fair shares cannot be accommodated. We improve upon both of these algorithms by using RIRO (roll-in, roll-out): a process runs for its turn, then moves its entire working set out of memory while other processes run; then, when its turn arrives again, it moves its working set back in. By using algorithms based on RIRO, we are able to ensure that every process receives its fair share over small windows of time, while maintaining full CPU utilization. We have developed four RIRO-based schedulers, and here present them along with results from their simulations.

2 Background and Motivation

Groups of processes often run simultaneously, both on individual levels (for example, a user running Firefox, Outlook, and Emacs on her laptop) and on group levels (a university network handling simulations from both the physics and chemistry departments). The system must then work out some way of dividing its resources among the processes. Often the system must also take into account processes' shares: values assigned by the user to each process, demanding that Process 1, for example, gets 25% of available CPU time, Process 2 gets 15%, and so on.

Most current solutions to this problem work by running the processes virtually simultaneously, passing out four CPU quanta to Process 1, then three to Process 2, and so on through the processes until cycling back to Process 1. This works well when all processes fit comfortably within available memory: all CPU time is used efficiently, and all processes receive precisely their fair share. The problem arises when the processes begin to grow too large to fit within the system's memory. If it is Process 2's turn to receive CPU quanta, but Process 2 does not have the necessary pages available to it, the system has only two options: either suspend the CPU while Process 2 swaps its pages into memory, or skip over Process 2 and move on to the next process. One choice sacrifices efficiency, the other sacrifices fairness. It is our goal to find a solution which sacrifices neither.

To deal with processes' pages, we use the concept of the working set. As defined by P. J. Denning in "The working set model for program behavior", a process's working set represents some subset of its pages, such that if those pages are available to it in memory, the process can run below the desired threshhold of paging. The working set can thus be used as a kind of footprint for a process.

Problems arise in traditional scheduling algorithms when available memory is too small to contain all of the processes' working sets. Our solution is to Roll In, Roll Out (RIRO): rather than keeping only fragments of every process's working set in memory, and so creating paging, we keep some of the processes' entire working sets in memory, and others' entire working sets out of memory. Then the active processes, those with their working sets in memory, can run efficiently with minimal paging; and when, according to our scheduling algorithms, an active process's turn is up, it rolls out: removes its entire working set from memory. Then a process that had been inactive rolls in: moves its entire working set into memory, in the space that the previous process cleared. This new process can now begin to run. While processes roll in and out, the other active processes continue to run, so the CPU efficiency never drops.

The trick, then, is to schedule the rolling in and out so that every process receives its fair share of CPU quanta. We experimented with several scheduling algorithms to achieve this; they are detailed in the next section. Each has its ups and downs, but the best of them, the Two Heap Scheduler, balances swift runtime with fairness sufficiently that we plan to implement it.

Overview

In Chapter 3, we describe the two kinds of scheduling algorithm and list the four algorithms we have created, including both an overview of each and an explanation of the algorithms themselves.

In Chapter 4, we detail the methods used to test the algorithms' performance. We describe the Java simulator that runs the algorithms, and explain the evolution and uses of the three kinds of plots that are the simulator's eventual output.

In Chapter 5, we show the resulting plots from a representative group of processes, and point out these plots' significant details.

In Chapter 6, we explain the conclusions drawn from the simulators' results, and the probable future of this research.

3 Scheduling Algorithms

Our scheduling algorithms fall into two main categories: *Pre-computed schedulers*, which take a set of processes, calculate the shortest possible fair schedule for those processes, and then run the schedule over and over until the processes have finished their jobs; and *heuristic schedulers*, which take a set of processes and run them on-the-fly, deciding which process to run next in real time. Pre-computed schedulers guarantee complete fairness, which heuristic schedulers cannot do; but pre-computing a schedule for more than three or four processes takes too much time to be practical (we suspect, in fact, that the problem is NP-complete). So our pre-computed schedulers were created not in the hope that they could ever be used on a real system, but only as a way to gauge the performance of our heuristic schedulers.

We created two scheduling algorithms of each type. The Simple Linear scheduler (SL) and the Start-End scheduler (SE) are pre-computed, and the In Order scheduler (IO) and the Two Heap scheduler (TH) are heuristic. Their algorithms follow.

Simple Linear Scheduler

A simple linear scheduler runs each process one by one, rolling out the previous process while the next one runs and rolling in the next process while the previous one runs.

Algorithm

- Make a table of process IDs P, shares S, and working sets WS.
- Make a list of all possible permutations of processes.
- Eliminate all permutations that are the cycled replication of another permutation.
- Determine each permutation's total runtime:
  - Each P_i's individual runtime RT_i must be $\geq WS_{i-1} + WS_{i+1}$ (i.e., each process must run long enough to roll out the previous process and roll in the next)
  - Make a table of each P_i's minimum RT, S, and RT/S
  - Calculate the permutation's share length λ:
    - Find the P_i with the largest RT_i/S_i
    - $\lambda = \lceil RT_i/S_i \rceil$
  - The permutation's total runtime is $\lambda * \sum S_i$
- Choose the permutation with the lowest total runtime.
- Let each P_i run for time $\lambda * S_i$ in the order given by the chosen permutation. Roll out P_{i-1} and roll in P_{i+1} as P_i runs.

Start-End Scheduler

A start-end schedule resembles a simple linear scheduler in that it lists all possible schedules, then chooses the one with the lowest runtime; but rather than using permutations of processes to create its schedules, it uses permutations of start events s_i and end events e_i to create its schedules.

Algorithm

Choosing a permutation

- List all permutations of s_i's and e_i's, including one s and one e for each process.
- Eliminate all permutations where:
  - At any time, the sum of WS's of active processes won't fit in memory
  - Any e_i comes before its matching s_i
- Find the share length λ for each remaining permutation
- Choose the permutation with the smallest λ

Finding the share length

- Let $\lambda = \lceil \frac{\sum RI_i + \sum RO_i}{\sum S_i} \rceil$, where RI and RO are the process's roll-in and roll-out times
- Try creating a schedule using this λ
- If the attempt fails, double λ and try again
- If the attempt succeeds, do a binary search between the last two checked λ's to find the smallest successful λ

Creating the schedule

- Find the set of always-active processes. Mark them as always-active, and put them on the active list. Always-active processes do not need to be rolled in and out.
- Devote all resources, both memory and CPU, to reaching the next s or e as quickly as possible.

  To reach s_i: P_i must be completely rolled in

  To reach e_i: P_i must have received $S_i * \lambda$ CPU quanta, and P_i must be completely rolled out

- If at any time the CPU is not busy with the previous step, choose the P_i on the active list with the soonest e_i, and let it run.
- Once a P_i has received $S_i * \lambda$ CPU quanta, take it off the active list.
- If at any point the CPU is left idle and there is nothing on the active list that can run, the schedule has failed for the given λ.

Finding the always-active processes

- All P_i's such that s_i is before all e's and e_i is after all s's are always-active.

In Order Scheduler

An in-order scheduler is a simple, naive algorithm that runs the processes in order of their ID numbers. Each process is run for the length of time given by its share; so if three processes have shares of 2, 8, and 50, they will run for 2, 8, and 50 units of time. If possible, the scheduler rolls out the previous process and rolls in the next one while the current process is running; but if there is not sufficient time to do that, the scheduler simply lets the CPU sit idle while it rolls processes in and out.

The performance of an in-order scheduler, as might be expected, is far from optimal. It was written because we needed a quick, simple, easy-to-understand heuristic to get an early look at how heuristic performance compared to pre-computed performance.

Two Heap Scheduler

A two heap scheduler is a more intelligent heuristic algorithm that keeps two heaps, or priority queues, of processes: an active heap (processes that are rolled in and receiving quanta) and a passive heap (processes that are rolled out). The heaps are ordered by excess or deficit of quanta; the top process of the active heap is the process that has received the most quanta proportional to its desired share, and the top process of the passive heap is the process that has received the fewest. The top processes of the two heaps are continuously swapped as the active processes run.

Algorithm

- Roll in as many processes as will fit within available memory, and add these processes to the active heap. Add all other processes to the passive heap.

- Repeat until all processes have finished running:
  - Take a *new process* from the top of the passive heap.
  - Repeat until the new process will fit within available memory:
    - Take an *old process* from the top of the active heap.
    - If the old process's excess is less than the new process's, give out quanta until the old process's excess is greater.
    - Roll out the old process and move it to the passive heap.
  - Roll in the new process and move it to the active heap.
  - If any other passive processes will fit within available memory, roll them in and move them to the active heap.

Notes

- Throughout the run, the CPU continuously distributes quanta to all active processes.

- Quanta are given out among the active processes proportionally to their goal shares.

- A process P's *excess*

$$= \frac{\text{CPU quanta P has received}}{\text{CPU quanta given to all processes so far}} \Big/ \frac{\text{P's goal share}}{\sum \text{All goal shares}}$$

- Since the active processes' excesses are constantly changing, the active heap must be resorted before a process is taken from it. To cut overhead time, the active heap may only be resorted every k times a process is withdrawn, where k is proportional to the number of processes. This will slightly reduce performance.

4 Methods

A Java simulator, Grapher.java, incorporates the four scheduling algorithms. The simulator accepts a file encoding a set of processes, then runs each of the four algorithms on the processes one by one, creating schedules and simulating the processes' scheduled roll-ins, roll-outs, and runs. The simulator then outputs a data file tracking each process's performance through the run. These data files were used to generate three different kinds of plots:

CPU Plots These simply plot how many CPU quanta each process has received over time. They are useful for determining the order in which the processes run, and getting a general feel for how a scheduler behaves.

Share Plots These plot what share of CPU quanta each process has received over time, with horizontal dotted lines representing what each process's goal share is. They are useful for determining a scheduler's overall fairness, and observing how closely each process conforms to the share the user has given it.

The share plots went through several evolutions during the project. Each process's share over time was first calculated by dividing the quanta that process had received by the total quanta the CPU had given out; this gave an accurate picture of a process's share over the lifetime of the run, but was useless for determining how fair the shares were over small windows of time. The share plots were then updated to calculate shares based on a sliding window: each process's share over time was calculated by dividing the quanta that process had received in the past few cycles over the total quanta given out in those cycles. This gave a more accurate picture of short-term fairness, but the runtime of calculating a new window of time at each step was prohibitive. In their final evolution, the share plots used exponential averaging to calculate each process's share; this method combined a quick runtime and an accurate look at fairness, and is used in the plots below.

Integral Plots These plot the distance between each process's actual share and its goal share over time, and include a label listing the sum area under all integral curves. This value can be used as a measure of a schedule's overall fairness, and is useful for comparing the fairness of several schedules.

5 Results

Many sets of processes were run through the simulator, and their plots examined. Share plots, as we expected, tend to oscillate around each process's goal share; as the process runs, its actual share grows until it exceeds its goal; then its turn finishes, it sits idle, and its actual share drops to below its goal. This oscillation means that no one process is ever too far from its goal share; and having all processes remain as close as possible to their goal, combined with a short runtime, is the mark of a good schedule.

Here is one representative set of processes and the plots it generated. The set's notable features are that it contains four processes and that two of its processes' working sets are much larger than the other two; so it is referred to as 4bigWS.

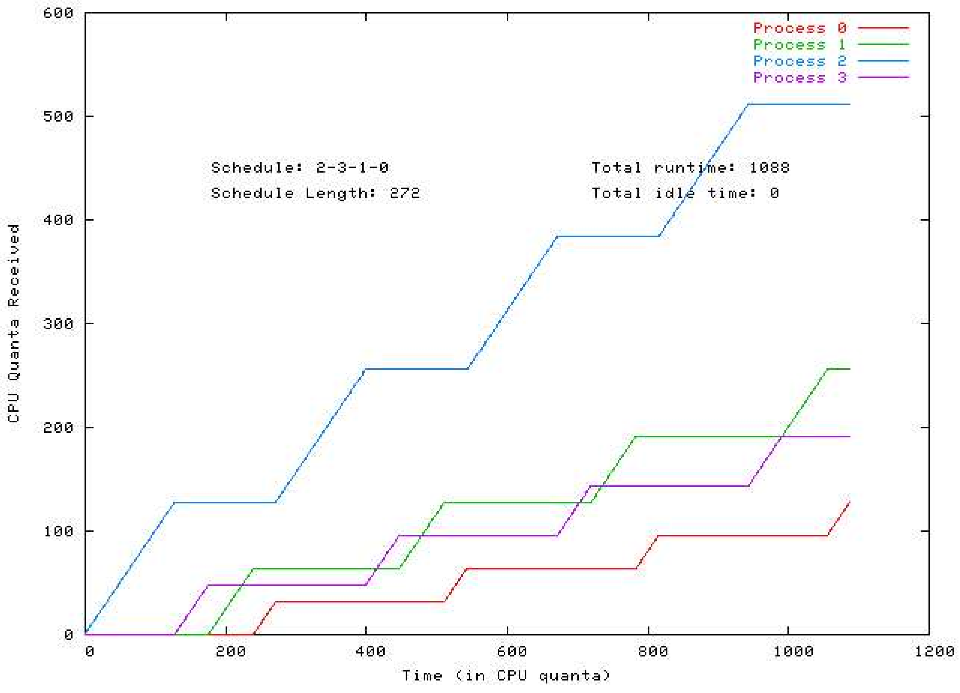
Process Set for 4bigWS

Process ID	Share	Working Set	Runtime
0	2	60	100
1	4	30	100
2	8	2	100
3	3	1	100

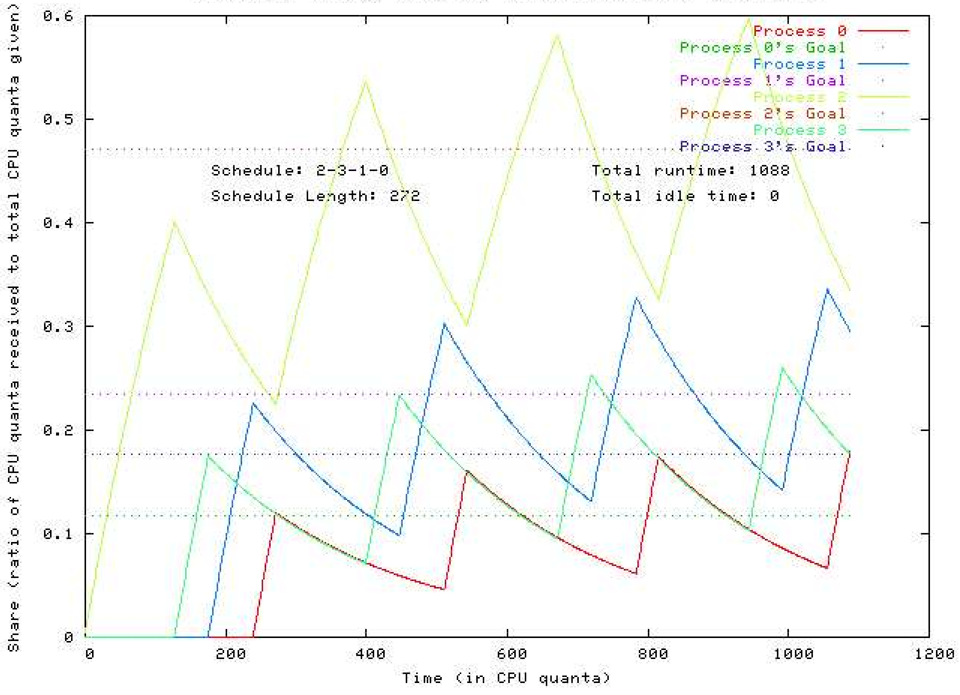
Simple Linear plots

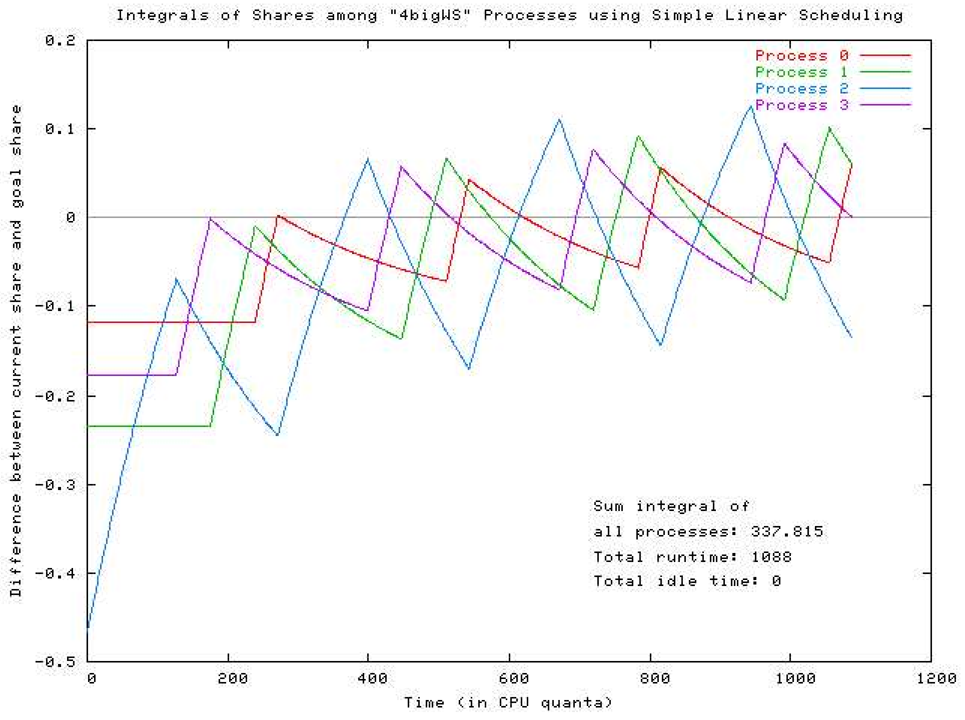
The Simple Linear plots proceed in a straightforward way: each process receives the amount of CPU quanta determined by the schedule, over and over, until all processes have finished running. This produces a short total runtime and a window of fairness equal to the schedule length: within every 272 CPU quanta, each process is guaranteed to receive its fair share.

Divison of CPU Quanta among "4bigWS" Processes using Simple Linear Scheduling



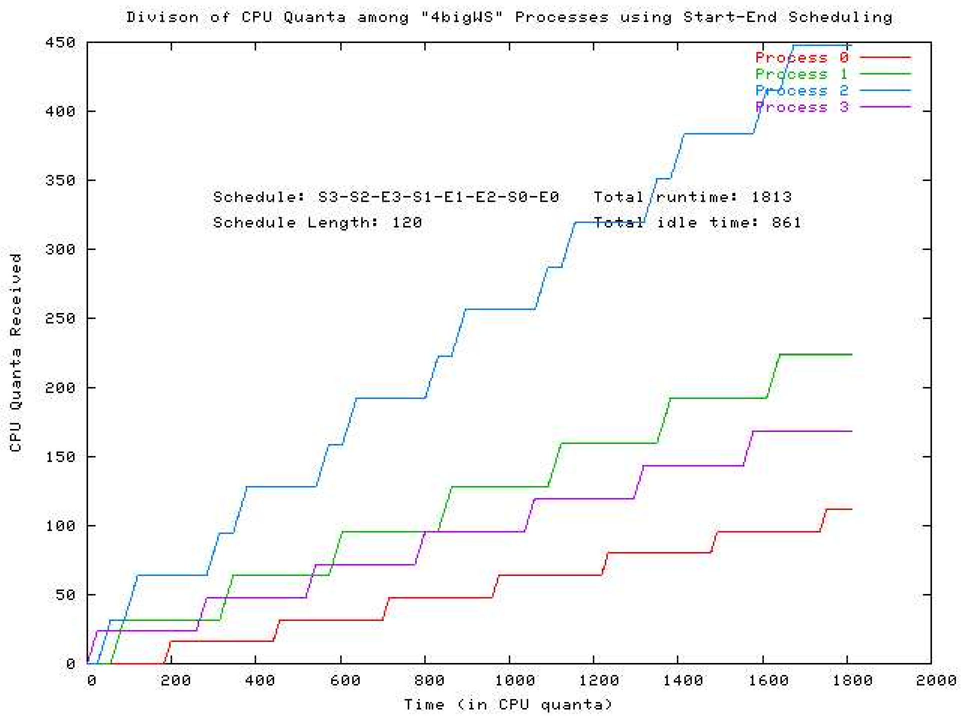
Shares of "4bigWS" Processes using Simple Linear Scheduling

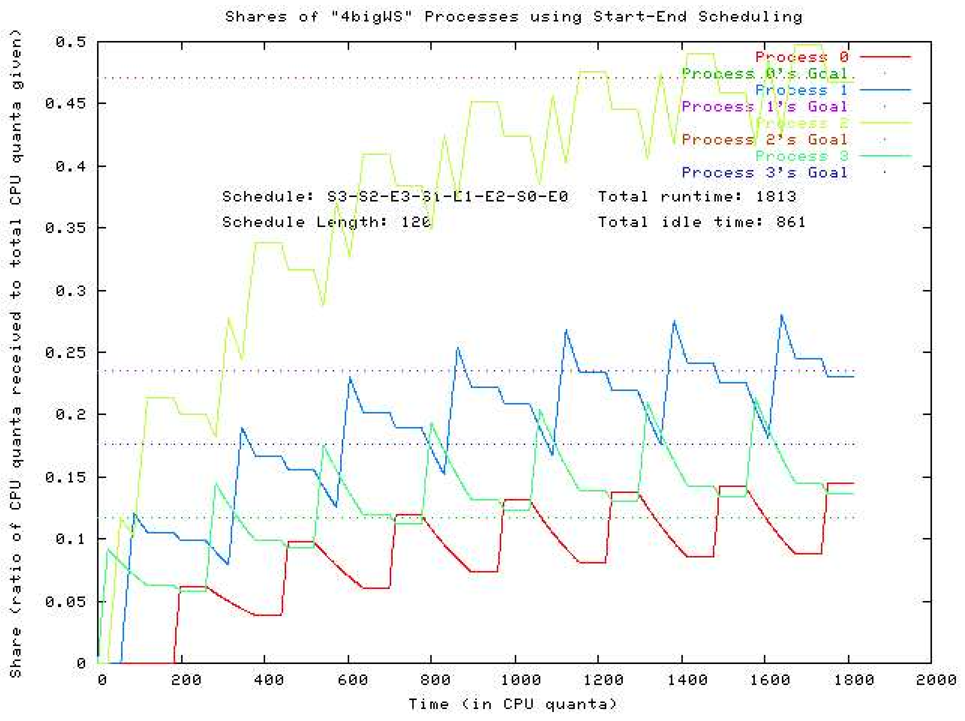


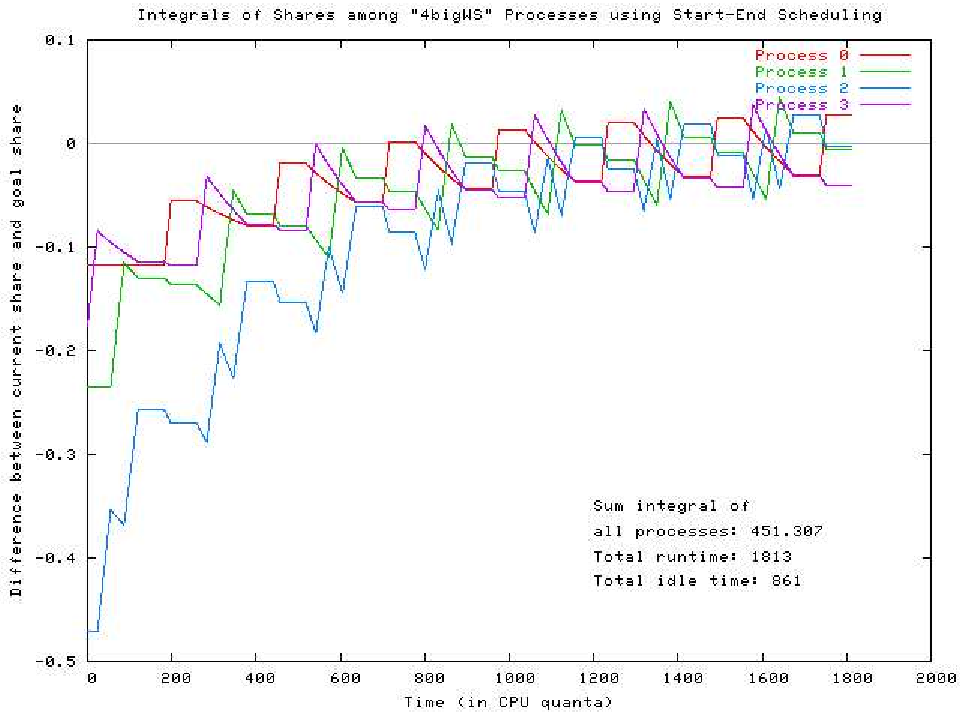


Start-End plots

The Start-End plot in this case has a much longer runtime than the Simple Linear plot, which at first glance makes little sense: if the Start-End algorithm can do everything the Simple Linear algorithm can to create the shortest possible schedule, with the addition of allowing processes' runtimes to overlap, why would the Start-End schedule take longer to run? The answer can be found in the high idle time. We have thus far found no way to make the Start-End algorithm strictly cyclic; it requires, at the beginning and end of each cycle, extra time to roll processes in and out. For some groups of processes, this time is insignificant, and the Start-End algorithm's shorter schedule lengths prevail; but in this case the idle time far outweighs the shorter schedule length. The Start-End algorithm does, however, have a smaller window of fairness than the Simple Linear algorithm.

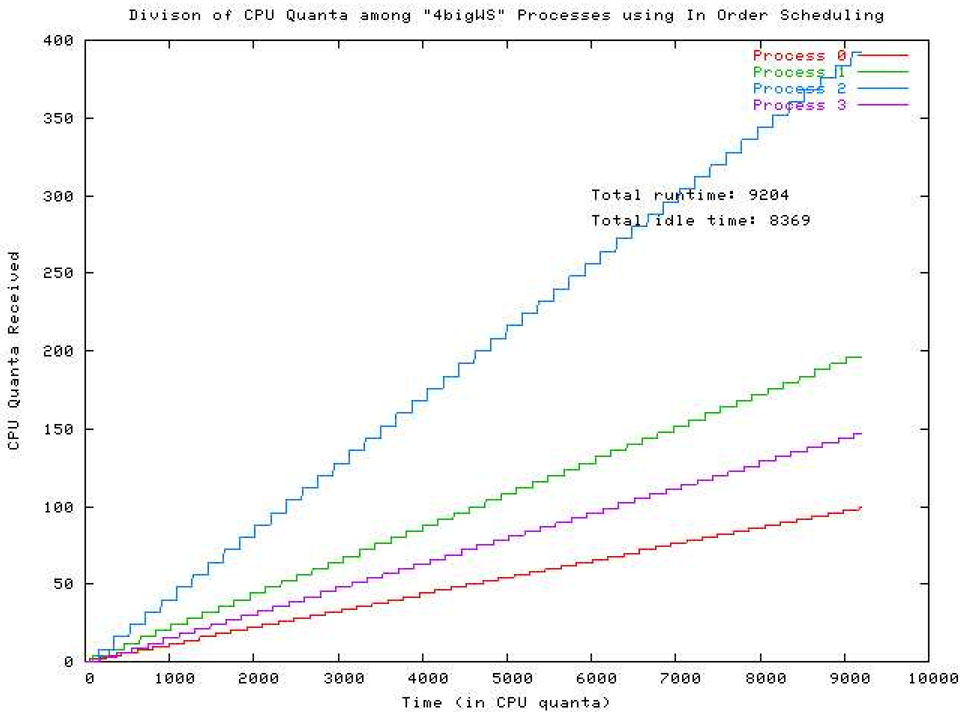

Divison of CPU Quanta among "4bigWS" Processes using Start-End Scheduling
Process 0
Process 1
Process 2
Process 3
Schedule: S3-S2-E3-S1-E1-E2-S0-E0
Total runtime: 1813
Schedule Length: 120
Total idle time: 861
CPU Quanta Received
Time (in CPU quanta)
0
50
100
150
200
250
300
350
400
450
0
200
400
600
800
1000
1200
1400
1600
1800
2000


Shares of "4bigWS" Processes using Start-End Scheduling
Process 0
Process 0's Goal
Process 1
Process 1's Goal
Process 2
Process 2's Goal
Process 3
Process 3's Goal
Schedule: S3-S2-E3-S1-E1-E2-S0-E0
Total runtime: 1813
Schedule Length: 120
Total idle time: 861
Share (ratio of CPU quanta received to total CPU quanta given)
Time (in CPU quanta)
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
0
200
400
600
800
1000
1200
1400
1600
1800
2000

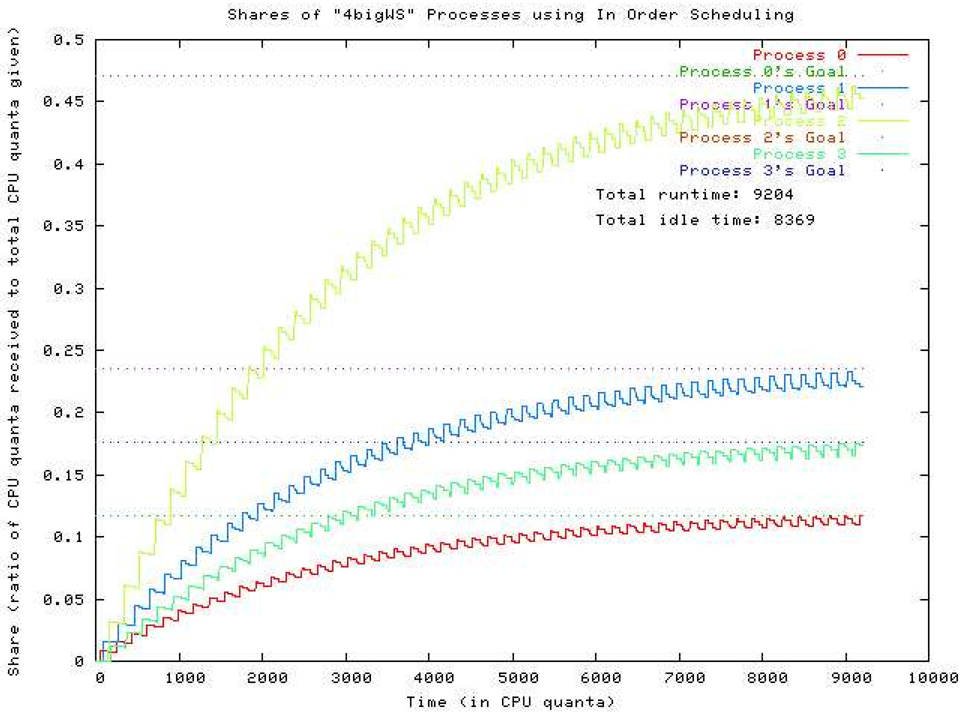


In Order plots

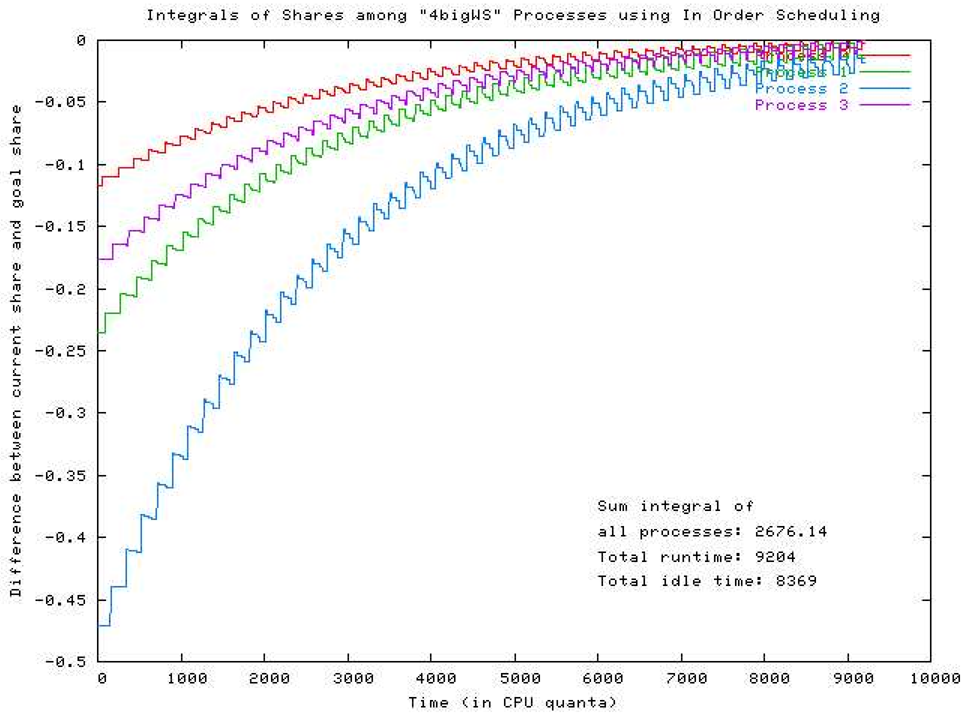
These plots demonstrate just how ludicrous the In Order algorithm's behavior can be. Because it allows the CPU to sit idle whenever it needs extra time to roll a process in or out, it racks up vast amounts of extra time, nearly stretching its runtime by a factor of ten. It does, however, have the smallest possible window of fairness, since each process runs only for the length of its share; in fact, if the working set sizes are very small compared to the shares, or if there is enough room in memory for almost all of the working sets, the In Order algorithm's behavior can be the best of all the four algorithms. Its obvious shortcomings, though, mean that it would be a poor choice for actual implementation.



Divison of CPU Quanta among "4bigWS" Processes using In Order Scheduling
CPU Quanta Received
Time (in CPU quanta)
Process 0
Process 1
Process 2
Process 3
Total runtime: 9204
Total idle time: 8369



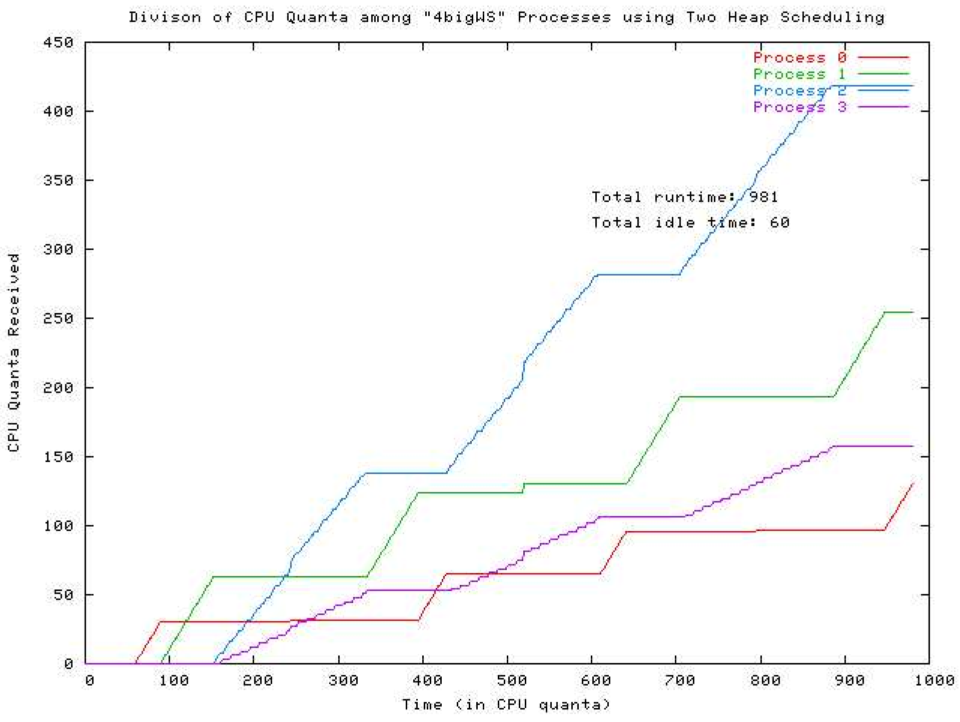
Shares of "4bigWS" Processes using In Order Scheduling
Share (ratio of CPU quanta received to total CPU quanta given)
Time (in CPU quanta)
Process 0
Process 0's Goal
Process 1
Process 1's Goal
Process 2
Process 2's Goal
Process 3
Process 3's Goal
Total runtime: 9204
Total idle time: 8369

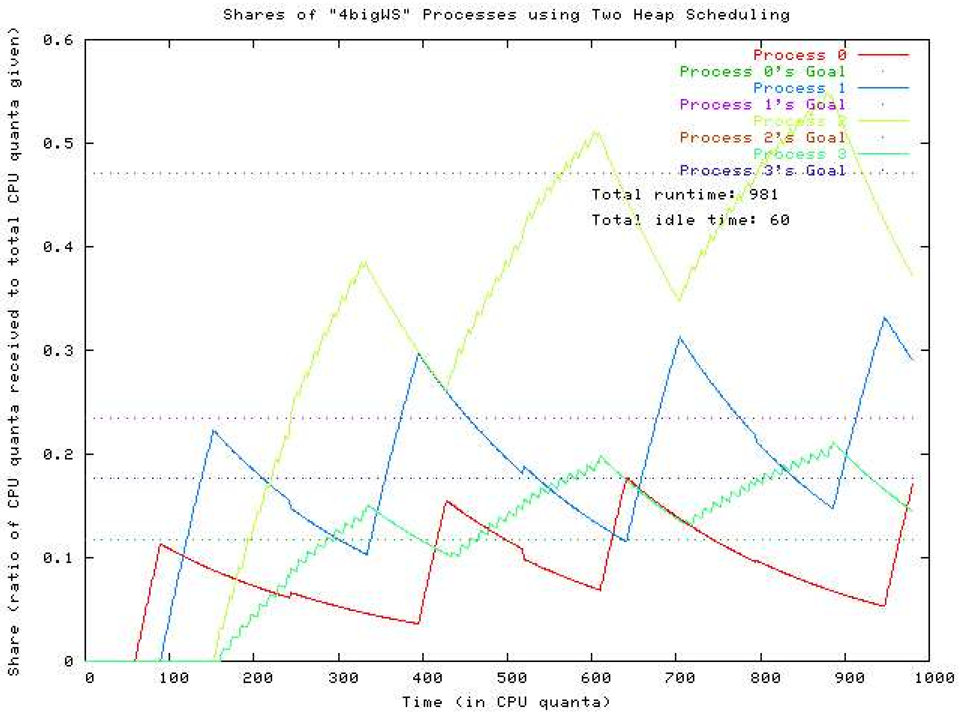


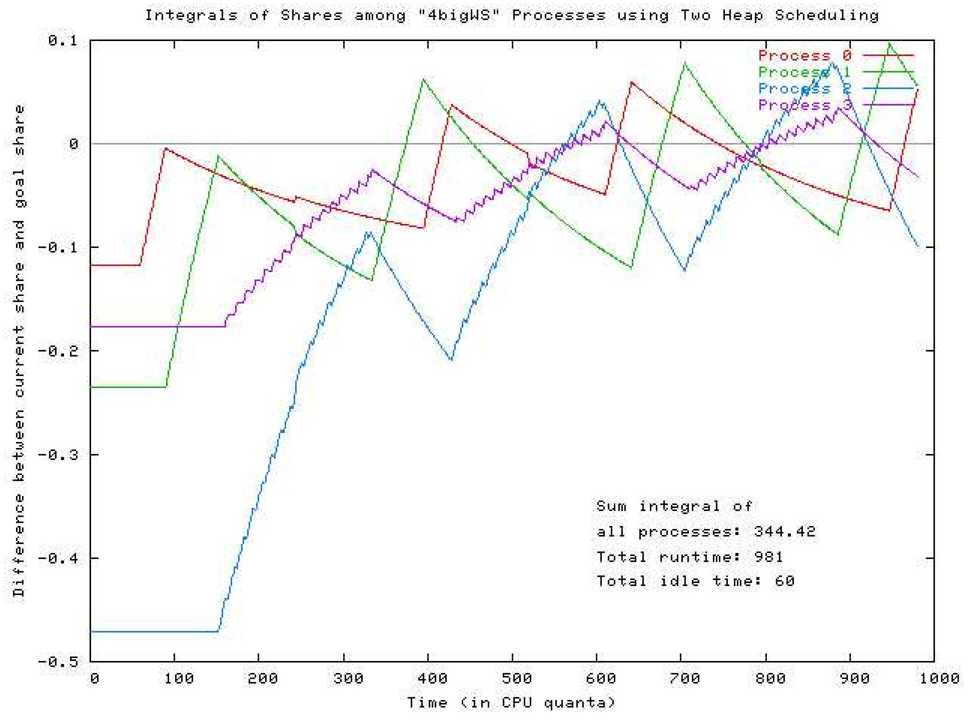
Two Heap plots

The Two Heap algorithm produces the only truly non-schedule-based plots: depending on the processes' attributes, they may run in any order at any time. This on-the-fly computation means that the Two Heap algorithm would be more practical for actual implementation than either of the two pre-computed algorithms; and since its behavior approaches the level of the Simple Linear algorithm for fairness (its total integral is only slightly higher) and efficiency (its idle time is not excessive), we believe it is a good choice to use for scheduling processes on a real system.

In the share plot, notice the jagged curves of processes 2 and 3, compared to the smooth lines of 0 and 1. These are a result of the difference in working set size among the processes. 0 and 1 have large working sets, and tend to sit in either the active or passive heap for long stretches; 2 and 3, with their tiny working sets, flit rapidly from heap to heap, picking up a few quanta here, a few there. This freedom to change behavior based on working-set size is unique to the Two Heap scheduler.


Divison of CPU Quanta among "4bigWS" Processes using Two Heap Scheduling
CPU Quanta Received
Time (in CPU quanta)
Process 0
Process 1
Process 2
Process 3
Total runtime: 981
Total idle time: 60
0
50
100
150
200
250
300
350
400
450
0
100
200
300
400
500
600
700
800
900
1000


Shares of "4bigWS" Processes using Two Heap Scheduling
Share (ratio of CPU quanta received to total CPU quanta given)
Time (in CPU quanta)
Process 0
Process 0's Goal
Process 1
Process 1's Goal
Process 2
Process 2's Goal
Process 3
Process 3's Goal
Total runtime: 981
Total idle time: 60
0
0.1
0.2
0.3
0.4
0.5
0.6
0
100
200
300
400
500
600
700
800
900
1000



6 Conclusions

As seen in the results, each of the four algorithms has its strengths and weaknesses. The simple linear scheduler produces the best plots of the four, combining a low total integral with a short runtime, but if it were implemented in an actual system, the cost of calculating a new schedule every time a process is added or changed would completely outweigh its benefits. The start-end scheduler suffers from the same problem, and thus would be useless for a real system; but if a solution could be found to the problem of making it truly cyclic, it would be the best of the pre-computed schedulers, and useful for judging the performance of the others.

The heuristic schedulers have no lengthy schedule-calculation time; their calculation takes place in small sections throughout the run, and so they could be used on a system without slowing it too grievously. The in order scheduler, however, would be a very poor choice; its immense idle time means that while it is very fair, its efficiency is almost as bad as using no scheduler at all; in some cases, it may even be worse. The two heap scheduler alone combines good fairness, short runtimes, and small calculation overhead. If a system running multiple processes suffers from memory shortage causing excessive paging, using the two heap scheduler to manage its processes would result in higher CPU utilization while maintaining the balance of shares desired by the user.

We next plan to create an actual implementation of the two heap scheduler and test it on real processes. Some aspects of a real system (changing working set sizes, for one) will need to be accounted for in the move from simulator to implementation; but the two heap algorithm, not being schedule-based, has the advantage of allowing processes to join in, drop out, or change midway through the run without suffering much loss of performance, and so we believe it will be up to the challenge. The algorithm also thrives under high-stress conditions: when a system has an excess of processes for its available memory, the two heap scheduler performs better because it always has processes of the right size available to swap from heap to heap.

7 Acknowledgements

This research was conducted at the University of Massachusetts at Amherst, and supported by the Distributed Mentor Project. The author thanks her mentor, Professor Eliot Moss, and her other co-researchers, Professor Scott Kaplan, Professor Emery Berger, and Xin Zheng, for their boundless support, interest, and expertise. She would also like to thank Natalie Podrazik and Maria Kazandjieva, her fellow UMass DMP mentees, and wish them luck with the remainders of their projects.

References

[1] P.J. Denning, "The Working Set Model for Program Behavior", CACM 19(5) pp.285-294 (1976).

[2] Richard William Carr, "Virtual Memory Management" (1981).