CS3210 Lab 3: Custom Scheduler Implementation

Josh Korol and Gal Ovadia

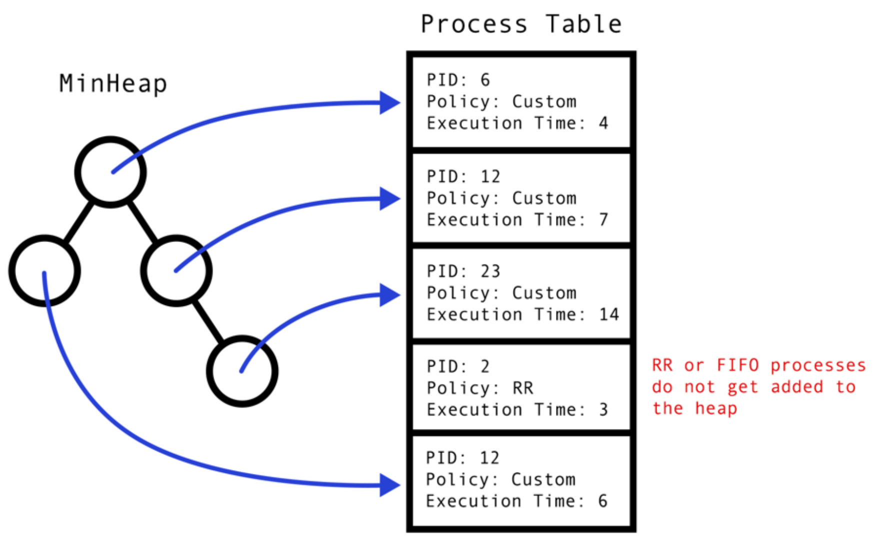
Our overarching idea was to prioritize processes that had received less time on the CPU, much like the "Deep Learning" workflow discussed in lecture. We were heavily inspired by CFS in designing our scheduling algorithm. Initial approaches we explored included a decaying priority system (i.e., start at 1000, and exponentially decay), and a multi-queue system which allowed processes a varying number of time slices depending on the queue (i.e., 8-timeslice, 4-timeslice, and 1-timeslice queue with sequential queue demotion).

We ultimately decided to pursue an implementation like CFS that used the `execution_time` statistic, to select the runnable process with the lowest execution time at each time slice to provide opportunities for fairness.

We decided against using an AVL or a Red-Black tree (like in CFS) in our implementation because we found such data structures too cumbersome to manage safely. Instead, we decided to make use of a MinHeap (see `heap.c`), as it would still provide efficient access to the minimum element (which is the primary usage of the tree structure in CFS). The MinHeap holds pointers to processes in the `ptable`, and entries in the heap are ordered based on the `execution_time` of the process to which the pointer points. Upon entering the scheduler, the process with the lowest execution time (at the root of the heap) is selected to run.

The new custom scheduler interacts with xv6 at 3 distinct locations:
- In `pinit()`we initialize our MinHeap (which sets `heap_initialized = 1` to ensure consistency amongst multiple cores)
- In `yield()` we add processes with the custom scheduling policy using `heap_push()`
- In `scheduler()`, if there are any processes on the heap, we pop the process and select it for execution.

Nits:
- The custom policy takes precedence over FIFO and RR.
- Whenever we use the heap, we make sure to acquire `ptable.lock` to avoid data races.

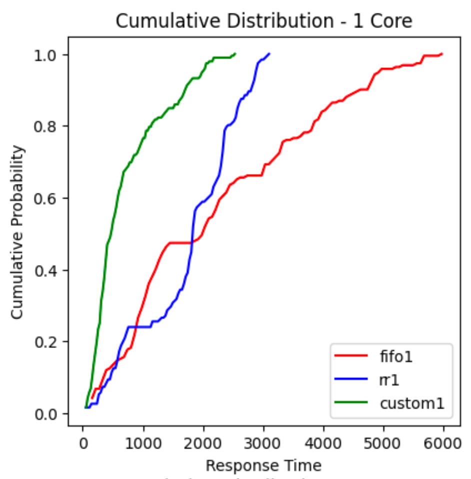


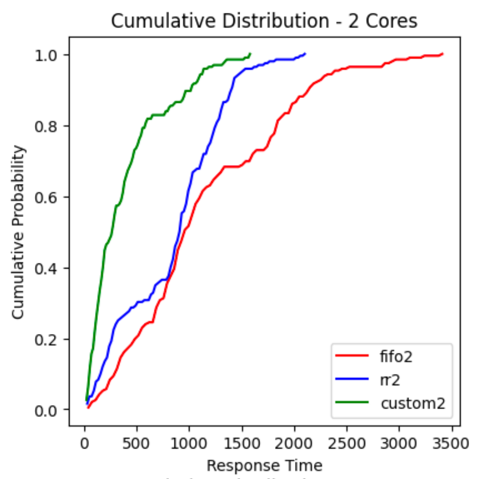
Our custom scheduler consistently outperforms FIFO and RR for the provided workload. This is thanks to efficient process selection via the MinHeap and the fairness element introduced.

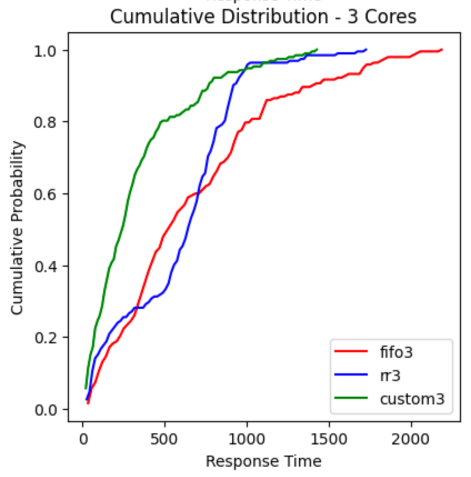
	p50 Scores			
	1 core	2 cores	3 cores	4 cores
FIFO	3037	1709	1101	967
RR	1561.5	1055.5	868.5	840
Custom	1282.5	794	717.5	666.5

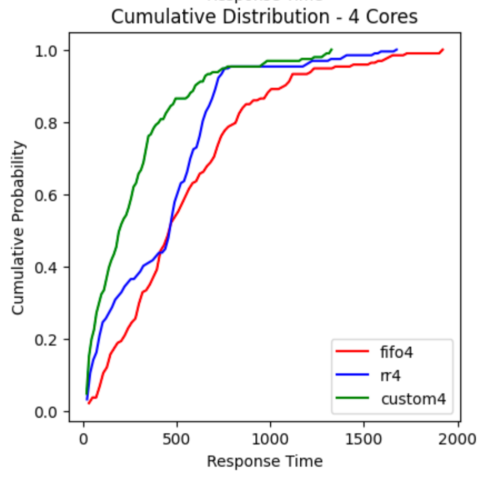
	p95 scores			
	1 core	2 cores	3 cores	4 cores
FIFO	5681.2	3239.9	2080.2	1825.6
RR	2947.05	1996.45	1642.05	1591.5
Custom	2407.05	1501.4	1356.95	1260.95

	p99 Scores			
	1 core	2 cores	3 cores	4 cores
FIFO	5916.24	3375.98	2167.24	1901.92
RR	3070.21	2080.09	1710.81	1658.3
Custom	2507.01	1564.28	1413.79	1313.79


Cumulative Distribution - 1 Core
Cumulative Probability
Response Time
fifo1
rr1
custom1


Cumulative Distribution - 2 Cores
Cumulative Probability
Response Time
fifo2
rr2
custom2


Cumulative Distribution - 3 Cores
Cumulative Probability
Response Time
fifo3
rr3
custom3


Cumulative Distribution - 4 Cores
Cumulative Probability
Response Time
fifo4
rr4
custom4