OS Challenge - RuffBandits

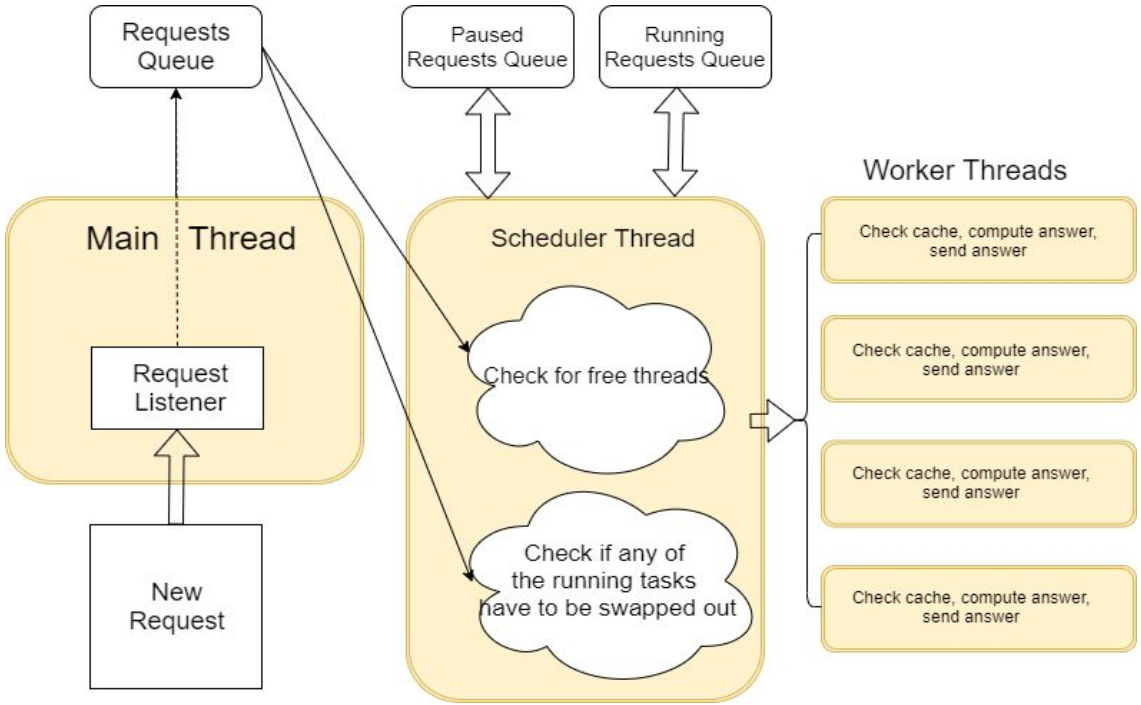
Presentation Content

- Final Configuration
- Processes vs Threads
- Optimal number of threads
- Priority queue for scheduling
- Task switching
- Task switching with aging
- Caching
- Splitting search range

Final Configuration

1. One main thread that accepts requests and adds it to a request queue.
2. Another scheduler thread that is responsible for scheduling tasks from the request queues to the worker threads.
3. **Four** worker threads work on the requests. They are created when there is a new task and destroyed when they are done. Max. limit of 4 maintained through semaphores.
4. Priority queue(written as an array heap) with “priority score” = priority/length. Task switching for high priority tasks.
5. Hash table is used as a cache for requests.

Final Configuration (contd.)



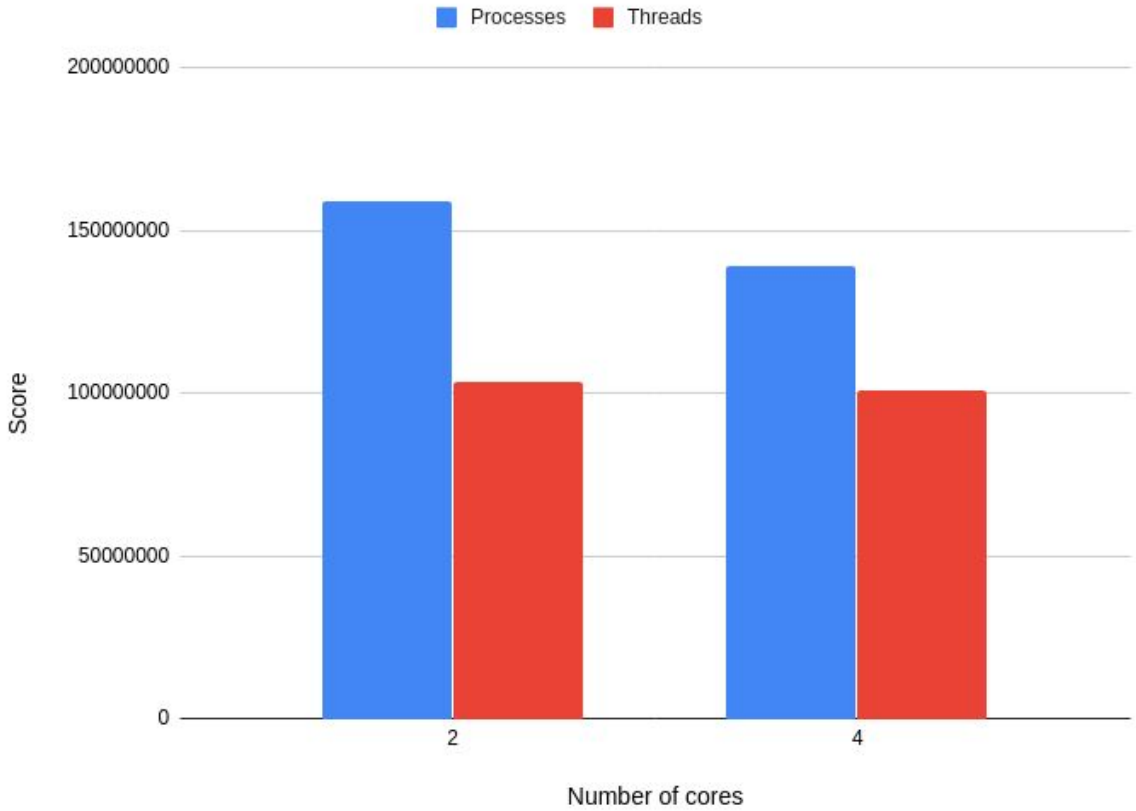
Processes vs Threads

Motivation: Find the fastest way to achieve parallel computing.

Methodology: Comparing the speed of application when using threads versus using processes.

Setup:
Number of requests: 100
Priority and repetition disabled
Difficulty: 30,000,000
Delay: 750000 us

Presented by Marius



Conclusion: Using threads is faster.

Presented by Marius

Determining the optimal number of threads

Motivation: Maybe fixing the total number of worker threads would benefit performance.

Methodology: Test a fixed number of requests using a FIFO scheduler while varying the max. number of threads. The queue is implemented using a linked-list.

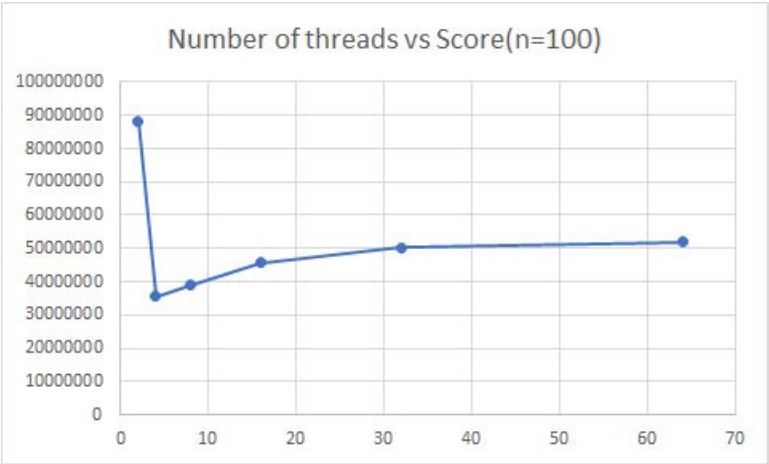
Setup:

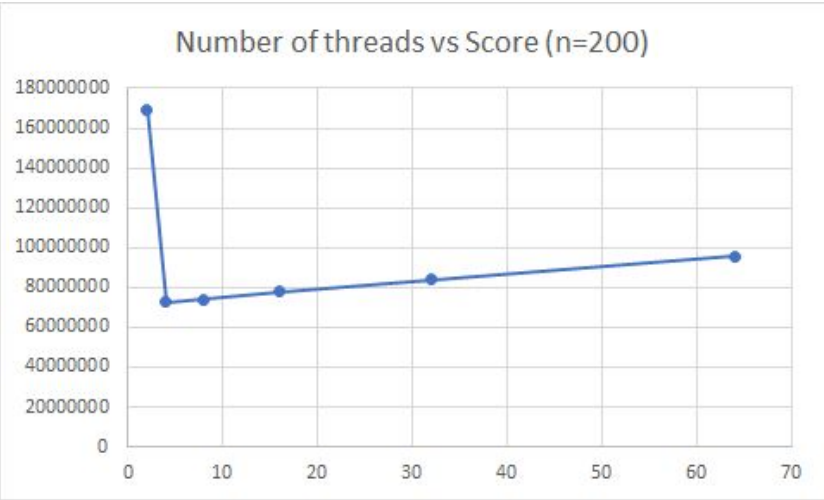
- Number of requests: 100 (and) 200
- Priority and repetition disabled
- Difficulty: 30,000,000
- Delay: 750000 us

Presented by Sumanth

Results

Conclusion: Four threads give the best results!





Presented by Sumanth

Using a priority queue for scheduling

Motivation: Higher priority tasks should get a higher priority in execution

Methodology: Use a priority queue implemented as a heap using arrays. Run two separate experiments - one with “priority_score=priority” and another with “priority_score=priority/length”.

Setup: Same as before, except lambda = 1.5

Presented by Sumanth

Results

Conclusion: priority/length is the best criteria!

Number of requests	Run 1	Run 2	Run 3	Median
50	15082754	17251782	16149072	16,149,072
100	30077713	31962473	31579166	31,579,166
200	76743828	86365248	85838146	85,838,146

FIFO Scheduler

Number of requests	Run 1	Run 2	Run 3	Median
50	13631516	14278223	14029912	14,029,912
100	28805148	29753985	33077179	28,805,148
200	62889042	61192206	60826242	61,192,206

Scoring based on priority

Number of requests	Run 1	Run 2	Run 3	Median
50	12514493	14686980	13488864	13,488,864
100	27468806	28737112	29557531	28,737,112
200	54449134	54446125	62710380	54,449,134

Scoring based on priority/length

Presented by Sumanth

Task Switching

Motivation While a low priority task is running, a high priority task may be added into the queue. The task might have to wait for a long time before it gets to run.

Configuration: The following changes are made:

- Two additional priority queues are maintained, one for paused requests(max-heap) and one for running requests(min-heap).
- Every thousand iterations, the running thread updates its `priority_score` in the running_requests queue, based on it's progress.
- If a more important request is unprocessed, it switches out a low priority task from the `running_requests` queue for the more important task. Meanwhile, the thread running the low priority task is suspended (waiting on a condition), and the paused task is added to the `paused_requests` queue.
- The paused tasks are then resumed when they are the highest priority tasks left.

Presented by Sumanth

Results

Score for Priority Scheduler without task switching:

Number of requests	Run 1	Run 2	Run 3	Median
50	12514493	14686980	13488864	13,488,864
100	27468806	28737112	29557531	28,737,112
200	54449134	54446125	62710380	54,449,134
400	120468449	122498267	123357713	122,498,267
600	217826075	231594647	227784506	227,784,506

Score for Priority Scheduler with task switching:

Number of requests	Run 1	Run 2	Run 3	Median
50	13361824	14680197	14633426	14,633,426
100	28469379	31878181	37527414	31,878,181
200	66440345	69651739	74331241	69,651,739
400	144018834	163847950	158393054	158,393,054
600	321528820	353091996	175797660	321,528,820

Conclusion: Overhead is probably a lot. Paused tasks are delayed quite a bit.

Presented by Sumanth

Task Switching with “Aging”

Motivation: Paused tasks incur a huge penalty for being in the queue for too long. Solution? Some form of “aging”.

Configuration: Same as previous experiment, except when a task is paused, it’s priority is increased by a factor of 1000, essentially meaning that it would be the next task picked up.

Setup: Same as before

Results

Score for Priority Scheduler without task switching:

Number of requests	Run 1	Run 2	Run 3	Median
50	12514493	14686980	13488864	13,488,864
100	27468806	28737112	29557531	28,737,112
200	54449134	54446125	62710380	54,449,134
400	120468449	122498267	123357713	122,498,267
600	217826075	231594647	227784506	227,784,506

Score for Priority Scheduler with task switching:

Number of requests	Run 1	Run 2	Run 3	Median
50	13361824	14680197	14633426	14,633,426
100	28469379	31878181	37527414	31,878,181
200	66440345	69651739	74331241	69,651,739
400	144018834	163847950	158393054	158,393,054
600	321528820	353091996	175797660	321,528,820

Score with task switching and aging:

Number of requests	Run 1	Run 2	Run 3	Median
50	14079701	14853068	14660201	14,660,201
100	26643355	31606668	37686529	31,606,668
200	62658209	74516205	72941512	72,941,512
400	160758916	151362085	152202125	152,202,125
600	190983652	199134136	181146696	190,983,652

Conclusion: We have better performance than even the regular priority scheduler when we have a large number of requests.

Presented by Sumanth

Experiment: “Caching”

Motivation: In the final client configuration hashes are repeated with a relatively high frequency. By storing the results of successful reverse hashing in a “cache” for later retrieval, perhaps time can be saved.

Methodology: Compare the control (No cache) versus various implementations of the “cache” (Linear search and a hash table).

Linear Search: Every time a reverse hash is computed, store a key-value pair in the next open cache slot. Key = the hash. Value = reverse hash. Upon a new request, do a linear search through the cache keys to check for matches.

Hash Table: Every time a reverse hash is computed, store a key-value pair in the cache. The cache index is determined by summing the first few bytes of the corresponding hash. Collisions are handled by simply replacing the old entry. Upon a new request, calculate the index the same as before and check the cache. If empty, reverse hash as normal. If not empty, verify that the key matches the received hash.

Results

Testing was done with the configuration to the right. The idea was to make the tests run faster to have more data points but also emphasize the effects of caching by keeping the reverse hashing difficult.

- TOTAL=500
- START=1
- DIFFICULTY=10000000
- REP_PROB_PERCENT=20
- DELAY_US=100000
- PRIO_LAMBDA=0

Trial	No Cache / Baseline	Arraylist with Linear Search	Hashtable
1	7m32.484s	7m16.300s	5m48.219s
2	7m34.122s	7m11.293s	5m35.874s
3	7m32.372s	7m17.450s	5m52.562s
4	7m31.428s	7m15.359s	5m33.792s
5	7m33.465s	7m19.773s	5m35.413s

Conclusion:

-Running the program without a cache performed performed the worst.

-Using linear search was also inefficient, as lots of time spent searching the cache and matching hashes.

-The hash table functions the fastest. By storing successful reverse hash computations we save a lot of time and a hash table allows for fast lookup. This was integrated into the final solution.

Presented by Weston

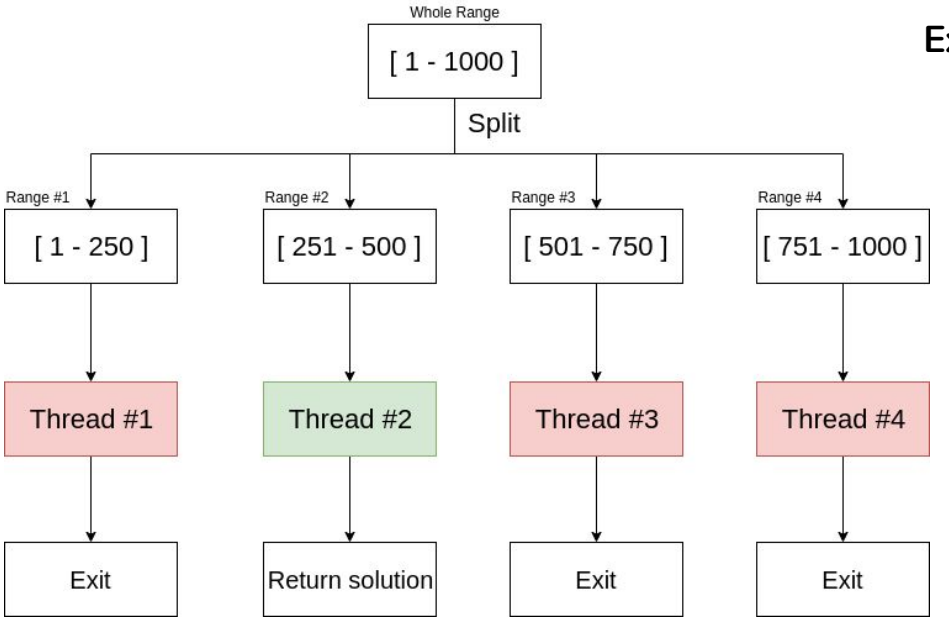
Experiment: Splitting the search range into K smaller ranges and assign them to different threads

Motivation
Searching smaller ranges in parallel should achieve higher performance.

Setup:
Number of requests: 100
Priority and repetition disabled
Difficulty: 30,000,000
Delay: 750000 us

Presented by Alexandros

Experiment: Splitting the search range into K smaller ranges and assign them to different threads



Example K=4

Presented by Alexandros

Experiment: Splitting the search range into K smaller ranges and assign them to different threads

Results and conclusion:

K / # of Threads	Score with no split	Score with K splits
2	88261857	67845146
4	35527196	56558719

Slightly better performance for **K=2**. No improvements for **K=4.**

Presented by Alexandros

QUESTIONS?