SYSC 4001 Assignment 3

Student 1: Braedy Gold-Conlin 101305254

Student 2: Maxim Creanga 101298069

Part 1

https://github.com/Volley24/Assignment_3

Introduction

In this assignment's coding section, we were tasked with implementing three schedulers, which would grab processes (which are defined by a PID, memory size, arrival time, total cpu time, i/o freq and i/o durr) and schedule them, based on an algorithm. The three schedulers implemented are External Priorities without preemption (EP), Round-Robin (w/ R = 5ms for better comparisons with other smaller files!!!), and External Priorities with Round-Robin and R = 5ms. Note that the code on github does use R = 100ms, as the instructions specified. In order to write a report about the data found, four sets of inputs were created: CPU-bound, I/O bound, Mixed, and Memory Bound (where memory partition constraints are the key factor here). Each category has five examples (found in the github), and there are four graphs targeting the four key metrics tracked in the simulator: Average Wait Time, Throughput, Turnaround Time, and Response Time.

Assumptions

For EP_RR, an assumption was made: since PID's must be unique, EP_RR doesn't use an RR that only kicks out processes of the same priority (since, this would effectively be round robin with one element for each process)

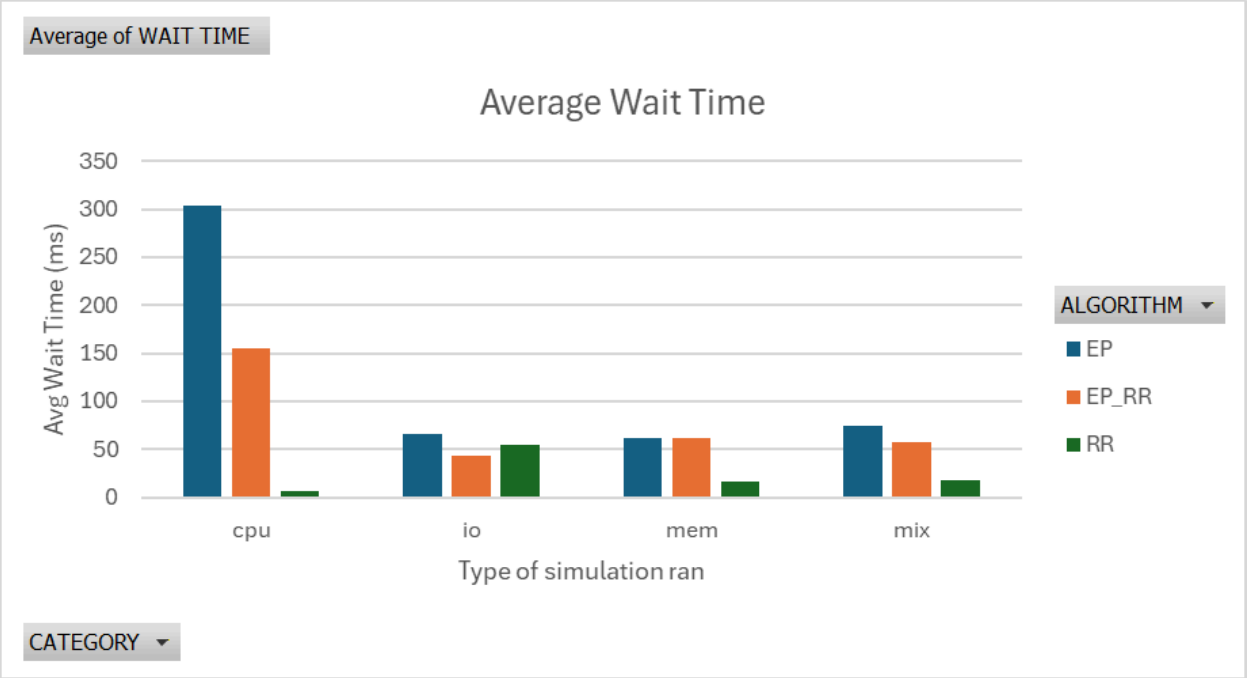


Figure 1: Average Wait Time

By analyzing the chart above, we can see a very obvious trend: Round-Robin has a MUCH lower average wait time then the External Priority algorithm. This is very clear from RR's algorithm, which likes to fairly cycle around running processes, to ensure that every process gets a fair chance to run, without getting blocked by a massive process. Actually, by observing the CPU bound processes, we can see that with EP, we see a MASSIVE average waiting time (300ms), while RR has an average time of around 7ms! This is due to the convoy effect, where larger processes take up all of the CPU time, while smaller processes which may have lower prorities have to wait an insane amount of time to wait for the bigger processes to finish. Round-Robin ignores the PID's priority system, allowing each process to get a fair time quantum (R = 5ms) to execute, eliminating the convoy effect and starvation of smaller processes. Even by looking at I/O, Mem and Mix, RR still always outperformed EP. EP_RR provides a sort of middle ground, allowing for prioritzation of processes, which is useful if you want at least SOME prioritization (instead of round robin having no priorities), while also keeping the average wait time mostly below EP.

Mem is an interesting discussion point, as EP_RR and EP are identical in that case. This is because in MEM, the bottleneck is actually the memory parition allocations, where each process make take up a lot of memory. In that case, the short term scheduler has little to no impact on the wait time, as it's the ready queue that is the bottle neck (RAM).

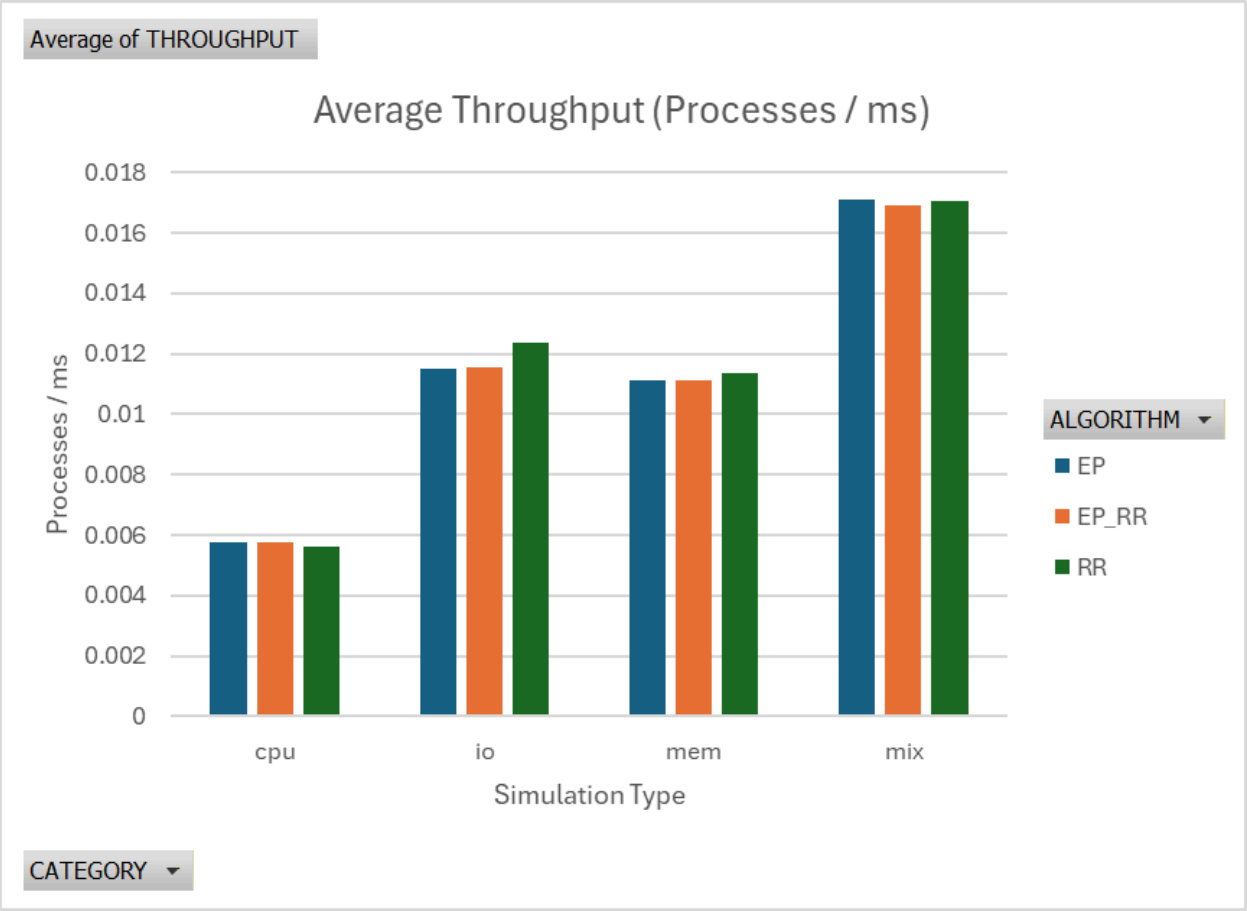


Figure 2: Average Throughput

By analyzing the average throughput in all simulation types, we can see that for CPU bound processes and mixed processes, the throughput is around the same. Since there is zero overhead in context-switches between processes, algorithms like round robin who switch often aren't penalized for this, and as such, the number of jobs done in a unit time is the same, as overall the same number of jobs will be ran globally. Since the CPU is always busy during this time, the small difference observed is noise. However, we can observe that for I/O bound processes, round-robin is better, is due to concurrency. Since RR switches around processes quicker, this allows for better utilization of one process doing CPU while another does I/O. This eliminates problems where a process runs for a while at the start, and there isn't any I/O to do. This gives us the clear winner at the RR, which is able to complete slightly more jobs this way.

Mem is still tied due to the bottleneck being the wait queue once more, not the short term scheduler.

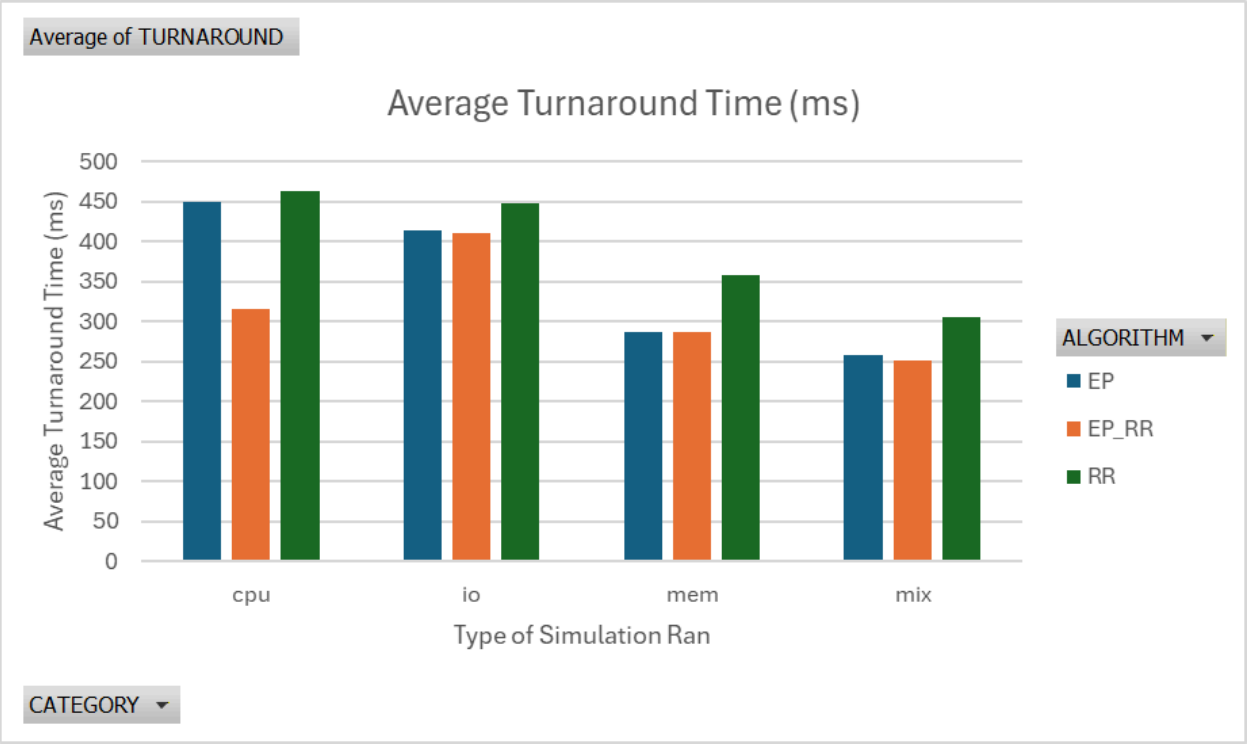


Figure 3: Average Turnaround Time

Unlike in other situations where RR excels other metrics, in this case, EP and EP_RR are the clear winners. While RR is known for giving each process a slice of time to execute fairly, this completely destroys its turnaround metric. Turnaround time is the time between the start and end of a process, so as such, if it is delayed, the time will suffer. In round robin, while each process may get to start fast, they will often all end at the same time, showcasing the tradeoff you get for fairness vs completion speed. As such, since most of the processes finish their execution towards the end of the simulation, the turnaround time will be significantly large.

EP_RR actually performed better than EP in this test, which was more evident in CPU-bound processes. While EP still suffered from the convoy effect, where a larger process got in the way of a smaller process, EP with RR allowed shorter jobs to take priority, bypassing this trap.

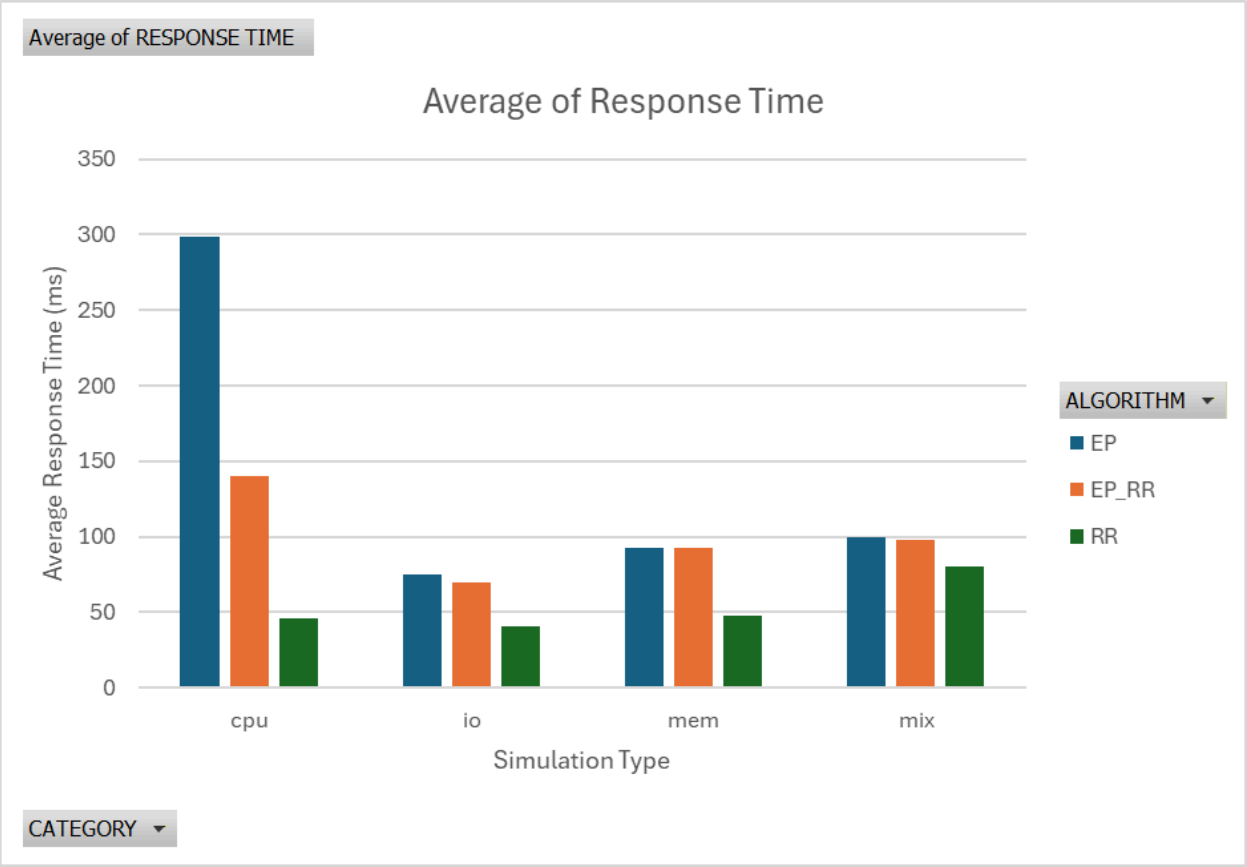


Figure 4: Average Response Time

Similarly to the average waiting time, the average response time for RR compared to EP is much lower. The average response time is the average time between I/O calls, which can be thought of as lag. If a process tries to run a very big CPU process, the EP algorithm would force that entire job to finish at once. As such, this drastically increases the time between I/O events, as smaller processes haven't had time to run, and thus no time to allow for the triggering of I/O events (since in our simulator, processes must do some CPU work before doing i/o work). However, in RR, each process gets a slice of time to run, which will allow for a MUCH better distribution of I/O handling, and as such, reduce lag in the entire system (response time.)

In conclusion, this simulation proves that there is no objective best algorithm for CPU short-term scheduling. It all depends on what you are looking to optimize, as not all variables (throughput, waiting time, response time and turnaround time) can be improved at once. Round-Robin excels for interactive systems, as it provides immediate responses and lowest wait times due to its fairness system. EP proved efficient in throughput, minimizing turnaround time, at the cost of the convoy effect. Finally, EP_RR proved to be the most balanced algorithm, as it mixes the fairness of RR with some

priority scheduling and preemption, allowing for balanced designs that aren't "all-or-nothing" on no priorities like RR.

Part 2

Github Link: https://github.com/BraedyCoding/Assignment3-Part2
pdf found in repo

SYSC 4001 Part 3

Part 3 – Concepts [1.5 marks]

Answer the following questions. **Justify your answers. Show all your work.**

1. **[0.6 marks]** Consider the following page reference string in an Operating System with a Demand Paging memory management strategy:

415, 305, 502, 417, 305, 415, 502, 518, 417, 305, 415, 502, 520, 518, 417, 305, 502, 415, 520, 518

(i) **[0.3 marks]** Assume we have 3 Frames Allocated. How many page faults will occur with 3 frames allocated to the program using the following page replacement algorithms?

(a) FIFO (First-In-First-Out)
(b) LRU (Least Recently Used) [Student 1: explain the LRU algorithm]
(c) Optimal

Show your work for each of the algorithms. Calculate the hit ratio for each of the algorithms.

(ii) **[0.1 marks]** Assume that the case above is repeated with 4 Frames Allocated. Repeat all three algorithms from Part (i) with 4 frames allocated to the program. Show your work for each of the algorithms. Calculate the hit ratio for each of the algorithms.

(iii) **[0.2 marks]** Based on your results, answer the following questions: Which algorithm performs best with 3 frames and why? Which algorithm performs best with 4 frames and why? How do the results change when more frames are allocated? What is the relationship? Why is the Optimal algorithm impractical in real-world operating systems? Compare the performance of FIFO and LRU. When might FIFO be better or worse than LRU?

(practice exercise for the final: repeat with LFU) [Student 2: explain the LFU algorithm]

Assumption: We are assuming that each page reference is a singular page index in the page table, rather than being some formatted page number, offset from the MMU.

a) FIFO

1. 415 → page fault → frames: [415] → faults = 1
2. 305 → page fault → frames: [415, 305] → faults = 2
3. 502 → page fault → frames: [415, 305, 502] → faults = 3
4. 417 → page fault, replace 415 (FIFO oldest) → frames: [305, 502, 417] → faults = 4
5. 305 → already in frames → no fault
6. 415 → page fault, replace 305 (FIFO oldest) → frames: [502, 417, 415] → faults = 5
7. 502 → already in frames → no fault
8. 518 → page fault, replace 502 (FIFO oldest) → frames: [417, 415, 518] → faults = 6

9. 417 → already in frames → no fault
10. 305 → page fault, replace 417 (FIFO oldest) → frames: [415, 518, 305] → faults = 7
11. 415 → already in frames → no fault
12. 502 → page fault, replace 415 (FIFO oldest) → frames: [518, 305, 502] → faults = 8
13. 520 → page fault, replace 518 (FIFO oldest) → frames: [305, 502, 520] → faults = 9
14. 518 → page fault, replace 305 (FIFO oldest) → frames: [502, 520, 518] → faults = 10
15. 417 → page fault, replace 502 (FIFO oldest) → frames: [520, 518, 417] → faults = 11
16. 305 → page fault, replace 520 (FIFO oldest) → frames: [518, 417, 305] → faults = 12
17. 502 → page fault, replace 518 (FIFO oldest) → frames: [417, 305, 502] → faults = 13
18. 415 → page fault, replace 417 (FIFO oldest) → frames: [305, 502, 415] → faults = 14
19. 520 → page fault, replace 305 (FIFO oldest) → frames: [502, 415, 520] → faults = 15
20. 518 → page fault, replace 502 (FIFO oldest) → frames: [415, 520, 518] → faults = 16

Total page faults= 16
Hit ratio = 4/20

b) Using LRU: Note I use a queue for showing my work where the if a page was accessed, then they go at the front of the queue, and the least recently used is at the back of the queue.

415 → page fault, frames: [415] → faults = 1

305 → page fault, frames: [415, 305] → faults = 2

502 → page fault, frames: [415, 305, 502] → faults = 3

417 → page fault, replace 415 → frames: [417, 305, 502] → faults = 4

305 → **hit**, frames: [417, 305, 502] → faults = 4

415 → page fault, replace 502 → frames: [417, 305, 415] → faults = 5

502 → page fault, replace 305 → frames: [417, 502, 415] → faults = 6

518 → page fault, replace 417 → frames: [518, 502, 415] → faults = 7

417 → page fault, replace 415 → frames: [518, 502, 417] → faults = 8

305 → page fault, replace 502 → frames: [518, 305, 417] → faults = 9

415 → page fault, replace 518 → frames: [415, 305, 417] → faults = 10

502 → page fault, replace 417 → frames: [415, 305, 502] → faults = 11

520 → page fault, replace 305 → frames: [415, 520, 502] → faults = 12

518 → page fault, replace 415 → frames: [518, 520, 502] → faults = 13

417 → page fault, replace 502 → frames: [518, 520, 417] → faults = 14

305 → page fault, replace 520 → frames: [518, 305, 417] → faults = 15

502 → page fault, replace 518 → frames: [502, 305, 417] → faults = 16

415 → page fault, replace 305 → frames: [502, 415, 417] → faults = 17

520 → page fault, replace 417 → frames: [502, 415, 520] → faults = 18

518 → page fault, replace 502 → frames: [518, 415, 520] → faults = 19

Hit ratio: 1/20

c) Optimal

Note: This is essentially where we look into the future and check which frames are not going to be used for the longest time, and we replace those.

Rule: Replace the page that will not be used for the longest time in the future.

15 → page fault → frames: [415] → faults = 1

305 → page fault → frames: [415, 305] → faults = 2

502 → page fault → frames: [415, 305, 502] → faults = 3

417 → page fault, replace 502 → frames: [415, 305, 417] → faults = 4

305 → hit → frames: [415, 305, 417] → faults = 4

415 → hit → frames: [415, 305, 417] → faults = 4

502 → page fault, replace 415 → frames: [502, 305, 417] → faults = 5

518 → page fault, replace 502 → frames: [518, 305, 417] → faults = 6

417 → hit → frames: [518, 305, 417] → faults = 6

305 → hit → frames: [518, 305, 417] → faults = 6

415 → page fault, replace 518 → frames: [415, 305, 417] → faults = 7

502 → page fault, replace 415 → frames: [502, 305, 417] → faults = 8

520 → page fault, replace 502 → frames: [520, 305, 417] → faults = 9

518 → page fault, replace 520 → frames: [518, 305, 417] → faults = 10

417 → hit → frames: [518, 305, 417] → faults = 10

305 → hit → frames: [518, 305, 417] → faults = 10

502 → page fault, replace 305 → frames: [518, 502, 417] → faults = 11

415 → page fault, replace 502 → frames: [518, 415, 417] → faults = 12

520 → page fault, replace 415 → frames: [518, 520, 417] → faults = 13

518 → hit → frames: [518, 520, 417] → faults = 13

Hit ratio = 7/ 20

(ii) **[0.1 marks]** Assume that the case above is repeated with 4 Frames Allocated. Repeat all three algorithms from Part (i) with 4 frames allocated to the program. Show your work for each of the algorithms. Calculate the hit ratio for each of the algorithms.

415 → page fault, frames: [415]
305 → page fault, frames: [415, 305]
502 → page fault, frames: [415, 305, 502]
417 → page fault, frames: [415, 305, 502, 417]
305 → already in frames → no fault
415 →already in frames → no fault

502 → already in frames → no fault
518 → page fault, FIFO replaces 415, frames: [518, 305, 502, 417]
417 → already in frames → no fault
305 →already in frames → no fault
415 → page fault, FIFO replaces 305, frames: [518, 415, 502, 417]
502 →already in frames → no fault
520 → page fault, FIFO replaces 502, frames: [518, 415, 520, 417]
518 → already in frames → no fault

417 → already in frames → no fault

305 → page fault, FIFO replaces 417, frames: [518, 415, 520, 305]
502 → page fault, FIFO replaces 518, frames: [502, 415, 520, 305]
415 → no page fault

520 → no page fault

518 → page fault , FIFO replaces 415: [502,518,520,305]

Total page faults: 10
Hit ratio: 50%

Doing LRU:

415 → page fault → frames: [415] → faults = 1
305 → page fault → frames: [415, 305] → faults = 2
502 → page fault → frames: [415, 305, 502] → faults = 3
417 → page fault → frames: [415, 305, 502, 417] → faults = 4
305 → hit → frames: [415, 305, 502, 417] → faults = 4
415 → hit → frames: [415, 305, 502, 417] → faults = 4
502 → hit → frames: [415, 305, 502, 417] → faults = 4
518 → page fault, replace least recently used 417 → frames: [415, 305, 502, 518] → faults = 5
417 → page fault, replace least recently used 305 → frames: [415, 417, 502, 518] → faults = 6
305 → page fault, replace least recently used 502 → frames: [415, 417, 305, 518] → faults = 7
415 → hit → frames: [415, 417, 305, 518] → faults = 7
502 → page fault, replace least recently used 518 → frames: [415, 417, 305, 502] → faults = 8
520 → page fault, replace least recently used 417 → frames: [415, 520, 305, 502] → faults = 9
518 → page fault, replace least recently used 305 → frames: [415, 520, 518, 502] → faults = 10
417 → page fault, replace least recently used 502 → frames: [415, 520, 518, 417] → faults = 11
305 → page fault, replace least recently used 415 → frames: [305, 520, 518, 417] → faults = 12
502 → page fault, replace least recently used 520 → frames: [305, 502, 518, 417] → faults = 13
415 → page fault, replace least recently used 518 → frames: [305, 502, 415, 417] → faults = 14
520 → page fault, replace least recently used 417 → frames: [305, 502, 415, 520] → faults = 15
518 → page fault, replace least recently used 305 → frames: [518, 502, 415, 520] → faults = 16

Page faults: 16 total

Hit ratio 4/20

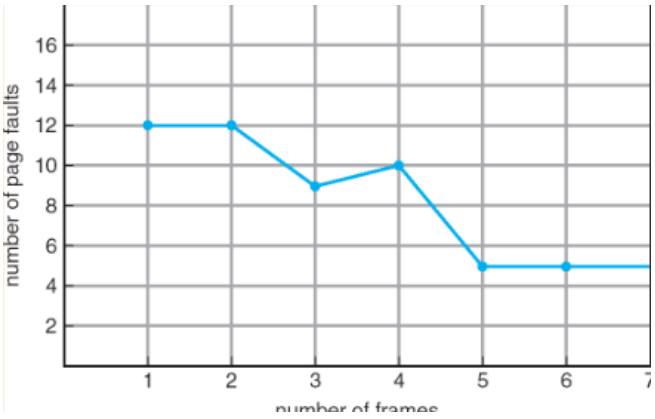
Optimal:
415 → page fault → frames: [415] → faults = 1
305 → page fault → frames: [415, 305] → faults = 2
502 → page fault → frames: [415, 305, 502] → faults = 3
417 → page fault → frames: [415, 305, 502, 417] → faults = 4
305 → already in frames → no fault
415 → already in frames → no fault

502 → already in frames → no fault
518 → page fault, replace 502 (next use: ref 12) → frames: [415, 305, 417, 518] → faults = 5
417 → already in frames → no fault
305 → already in frames → no fault
415 → already in frames → no fault
502 → page fault, replace 415 (next use: ref 17) → frames: [305, 417, 518, 502] → faults = 6
520 → page fault, replace 502 (next use: ref 17) → frames: [305, 417, 518, 520] → faults = 7
518 → already in frames → no fault
417 → already in frames → no fault
305 → already in frames → no fault
502 → page fault, replace 305 (next use: never) → frames: [417, 518, 520, 502] → faults = 8
415 → page fault, replace 417 (next use: never) → frames: [518, 520, 502, 415] → faults = 9
520 → already in frames → no fault
518 → already in frames → no fault

Total faults = 9
Hit ration = 11/20

(iii) **[0.2 marks]** Based on your results, answer the following questions: Which algorithm performs best with 3 frames and why? Which algorithm performs best with 4 frames and why? How do the results change when more frames are allocated? What is the relationship? Why is the Optimal algorithm impractical in real-world operating systems? Compare the performance of FIFO and LRU. When might FIFO be better or worse than LRU?

So the issue with FIFO is that they can experience something called belady's anomaly. This is caused by the initial loading of into the empty frame. However, this didn't happen at all in our case as increasing the number of frames actually reduced the number of page faults. But



With LRU, it just looks at the past. Increasing the number of frames always decreases the number of page faults.

Optimal on the other hand looks into the future and sees which pages won't be used for a while, so it knows which ones to remove. This is not practical whatsoever, as no current technology can time travel.

2. **[0.3 marks]** Consider a system with memory mapping done on a page basis. Assume that the necessary page table is always in main memory. A single main memory access takes 120 nanoseconds (ns).

(a) **[0.1 marks]** How long does a paged memory reference take in this system without a TLB? Explain your answer.

(b) **[0.1 marks]** If we add a Translation Lookaside Buffer (TLB) that imposes an overhead of 20 ns on a hit or a miss. If we assume a TLB hit ratio of 95%, what is the effective memory access time (EMAT)? Explain your answer.

(c) **[0.1 marks]** Why does adding an extra layer, the TLB, generally improve performance? Are there situations where the performance may be worse with a TLB than without one? Explain all cases.

a) It would take 240ns because it has to access the page table (120ns) to get the frame number, then add the offset with that, and then access the frame +offset (another 120ns).

b) The TLB is located in the processor, so it doesn't need to go into RAM (which takes a while), accessing the buffer takes 20ns.
Best case: 20ns + 120ns = 140
Worse case: 20ns + 120ns + 120ns = 260ns
EMAT = (0.05)(260) + (0.95)(140) = 146ns

c) If the range of logical addresses for a process is super large, and page sizes are small then the TLB will not have much reach. Because, we essentially have a lot of pages, and if the TLB is not that large, then there will be a higher miss rate (worse).
→ in general, to maximize the usage out of the TLB, we want to give it a certain reach.
Reach of TLB = Number of pages it can hold x page size

So, if the page size is super small, you want to increase the number of pages it can hold. If any of these are off what so ever, it can greatly affect your hit/miss ratio.

3. **[0.3 marks]** Consider a system with a paged logical address space composed of 128 pages of 4 Kbytes each, mapped into a 512 Kbytes physical memory space. Answer the following questions and justify your answers.

(a) [0.1 marks] What is the format and size (in bits) of the processor's logical address?

(b) [0.1 marks] What is the required length (number of entries) and width (size of each entry in bits, disregarding control bits) of the page table?

(c) [0.1 marks] What is the effect on the page table width if now the physical memory space is reduced by half (from 512 Kbytes to 256 Kbytes)? Assume that the number of page entries and page size remain the same.

[Student 1: explain what the physical memory space is]
[Student 2: explain what the logical memory space is]

a) We know that the MMU splits the logical address into a page number + offset. If each page is 4kB that means that the offsets must be 4kB which is 12 bits. So we know the offset portion alone is 12 bits.

   As for the page number, there are 128 pages (page entries in the page table), for each processed page table, so 128 is 7bits.

   Thus, the total number of bits is 19.

b) Well there would be 128 entries in the page table, then there would need to be 7 bits to choose which frame number we want. So the page table is used to give the MMU a frame number for a processed page number. The MMU then (assuming each frame is the same size), does physical address = (page number x frame size) + offset.
   So it is important to note that the page table holds frame numbers, so if you have 128 page numbers, that would mean there are 128 page entries (frame numbers) and the number of bits for 128 is 7, so there would be 7 bits for each entry in the page table.

c) If somehow, the page size and most importantly the number of page entries stay the same, the memory is cut in half, this means some of the page entries are not being used (which is fine, it is common for the page table to have more page entries than frames in RAM). The number of frames is being cut in half, from 128 to 64. This means that if there are 64 frames the same size as before, there would be **6 bits** in each page table entry rather than 7 bits because 6 bits is the lowest number to represent 64 possible frames that the pages could map to.

4. **[0.1 marks]** Explain, in detail, the sequence of operations and file system data structure accesses that occur when a process executes the **lseek(fd, offset, SEEK_END)** system call. Consider a system using a hierarchical directory structure and assume the file described by the file descriptor (fd) is not currently open by any other process.

The lseek system call in Linux/Unix is used to change the current file position (offset) of an open file. Essentially, it moves the “cursor” in the file so that the next read or write happens at a different location.

What happens is the fd is a unique integer used to index the per-process file descriptor table (you can think of it almost like a pointer in a way). Since this is the only process that has the file open, the reference count in the system-wide open file table entry will be 1.

When the system call is invoked, it raises a software interrupt, switching the CPU into kernel mode. The system call handler examines a specific register that encodes the system call number. This number is used as an index into the system call table, which is an array of function pointers, and the handler then branches to the appropriate function in this case, the kernel’s implementation of lseek(...).

Inside the lseek function:

1. The fd parameter indexes the per-process file descriptor table, giving access to the entry for this file.
2. That per-process table entry contains a pointer to the system-wide open file table entry (the open file description), which stores the current file offset and a pointer to the inode.
3. The inode is consulted to get the file size, because SEEK_END requires moving relative to the end of the file.
4. The current file offset in the system-wide table entry is updated and it is literally just

   New offset = current_offset + offset_from_lseek

5. No disk blocks are accessed at this stage; the inode provides the necessary metadata.

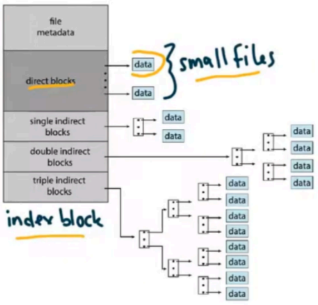
Finally, the new offset is returned to the process, and the process can continue to read() or write() at this position.

Super summed up: so all it really did was system call→use fd to get per process file table (for the process) → use system wide file pointer from per process file table to access system wide file table entry, increment the offset in the system wide file table for that file (entry) using the one passed into lseek.

5. File System Organization

a) **[0.1 marks]** (from Silberschatz) Consider a file system that uses inodes to represent files. Disk blocks are 8Kb in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?

b) **[0.1 marks]** Explain what you can do in case (a) if you need to store a file that is larger than the maximum size computed. Give an example showing how you can define a larger file, and what the size of that file would be.



I will be referring to this diagram a good amount:

a) The maximum file size is calculated by adding the space accessible through direct and indirect blocks. Each disk block is 8 KB, and a pointer to a block takes 4 bytes. The 12 direct blocks alone can store 12 * 8 KB = 96 KB. The single indirect block points to a block full of pointers. Since each block is 8 KB and each pointer is 4 bytes, there are 8192 / 4 = 2048 pointers, each pointing to a data block, giving 2048 * 8 KB = 16 MB. The double indirect block points to 2048 single indirect blocks, so it can address 2048 * 2048 * 8 KB = 32 GB. The triple indirect block points to 2048 double indirect blocks, giving 2048 * 2048 * 2048 * 8 KB = 64 TB. Adding them together, the maximum file size is dominated by the triple indirect block and is roughly 64 TB.

b) If a file needs to be larger than the maximum size allowed by the inode structure, one solution is to split the file across multiple inodes. Each inode would store a portion of the file up to its maximum size, and you could link the inodes together in a higher-level structure, such as a file table or list of inodes, to treat them as a single logical file. For example, if each inode can store up to roughly 64 TB, two inodes could be combined to

represent a file of 128 TB, three inodes for 192 TB, and so on. This way, there is no hard limit on the logical size of the file, as long as additional inodes are available.

student 2 individual assn (maxim creanga 101298069)
as copied from individual

1) Explain the LFU (Least Frequently Used) algorithm

The LFU algorithm, or Least Frequently Used, is an algorithm used by an operating system, in the context of page-swapping. Firstly, we are assuming a system with logical memory, physical memory, and a hard disk holding virtual memory. Secondly, we are assuming data is packaged in “pages”, which are chunks of data. The logical memory may be larger than the physical memory, and as such, we do not want to store all the pages from every program on the physical memory. Rather, with the use of algorithms, we aim to have the most “relevant” pages be in the physical memory, by using algorithms and approximations, to try and reduce the number of page-swaps needed, known as page faults, as they result in slower memory access. These algorithms are executed when a page fault occurs and have a goal of trying to minimize future page faults, in the way they select the victim page. The LFU algorithm is a type of page-swapping algorithm that is counting based, meaning that each page has a counter associated with it, and will be incremented when the page is accessed by a program. A simple version of this has a counter that is big, like 12 bits, as the counter will not be reset. As such, these values can reach very large numbers and won’t get reset. A more advanced approach uses the Timer ISR to reset the count every timer interval t, (something relatively large, so that frequency is still measured), which would reset this counter. This allows us to use less bits for the counter, which will save a lot of space. The meaning of what the counter value represents is the following: it works on the philosophy that pages that are accessed more often will probably need to be accessed again shortly, so therefore, pages with a lower count will be targeted as the victim page and swapped out for our target page.

2) Explain what the logical memory space is.

In an operating system, the logical memory space is the memory as seen from the program’s perspective. This memory is sequential, and always starts at 0, and ends at the program size, having a physical limit (as a maximum) that the OS imposes on the logical memory space. This memory view is seen as an abstraction, as it is not physical memory which holds all the information, such as variables, data, and stack/heap. As such, whenever the program attempts to access a variable at a certain location in the logical memory space, a special controller known as the MMU will decode that address, ensure that it is between 0 and the program’s size, and raise an exception, which is also known as a “trap”, if the values do not fall between that range. Then, the MMU will then add the logical memory address with the offset of the current program in physical memory (base register value), assuming no paging is used in the physical memory. If paging is used, more complicated setups exist, where there is a page table that will determine where in physical memory our page is located, assuming it is within bounds again. It is important to note that the (sum of all) logical memory may exceed the physical memory size, as there are techniques involving page swapping which tries to hold more relevant pages (“relevant” = some page switching algorithm, there are many.), while swapping out victim pages in hard disk, known as virtual memory. Finally, the whole point of the abstraction is to allow freedom to implement the actual physical memory as needed, without the actual logical memory accessing being drastically changed, and can range from simple as sequential allocation to as complex as pre-paging with page swapping algorithms using virtual memory