SYSC4001 Assignment 3:

Concurrency, Shared Memory, Virtual Memory, Files:
Parts I, II, & III

Group 8
Antoine Hickey,
ID: 101295764
Enzo Chen,
ID: 101306299

Lecture: SYSC4001A
Lab: SYSC4001L2

Part 1 – Scheduler Simulator

https://github.com/hicked/SYSC4001_A3_P1

This report presents an analysis of three process scheduling algorithms: **EP (Priority without preemption)**, **RR (Round Robin)**, and **EP_RR (Priority, Preemption and Round Robin)**. The simulations evaluate their performance under CPU-bound, I/O-bound, and mixed workloads, considering metrics such as throughput, average turnaround time, average wait time, and average response time. These are tracked during execution, and then calculated within the header at the end of each simulation (output in execution.txt). The simulations also showcase edge cases where we can highlight some key findings. It will aim to report insights on both the states of each process, as well the memory usage at each transition between states.

Bonus: Memory Usage Analysis

To be able to better track and analyze the state and memory utilization of the simulation, the memory_analysis.txt was added as an additional form of output for each simulation. These provide insight every tick (ms) into the state and memory utilization of the simulated system. Below is an example of an entry within these output file:

```
--- Memory / Events at Time 33 ---
Transitions this tick:
| PID | Old -> New
|-----|-------------
|  3  | RUNNING -> TERMINATED
|  2  | READY -> RUNNING

Process State:
| NEW | READY | WAITING | RUNNING | TERMINATED |
|-----|-------|---------|---------|------------|
|  6  |   1   |    -    |    2    |     3      |
|  -  |   5   |    -    |    -    |     -      |
|  -  |   4   |    -    |    -    |     -      |

Partition Usage:
| Part | Size | Used | Unused | PID |
|------|------|------|--------|-----|
|  1   |  40  |  5   |   35   |  5  |
|  2   |  25  |  20  |   5    |  4  |
|  3   |  15  |  15  |   0    |  2  |
|  4   |  10  |  10  |   0    |  1  |
|  5   |  8   |  0   |   8    | -1  |
|  6   |  2   |  0   |   2    | -1  |
```

Initial Simulations

Before analyzing the behaviors of the scheduler in a CPU bound, I/O bound and mixed workflows across multiple algorithms, it is important to highlight a few keys behaviors in order to ensure the simulator is working properly. Below are edge cases that were run for each algorithm.

EP

1. BasicPriority Simulation

This is a basic simulation that is used to demonstrate that if two processes arrive at the same time, the one with higher priority (lower number) will be scheduled first. In this case, both PIDs 1 and 10 arrive at the same time, but since PID 1 has higher priority, it runs first:

```
--- Memory / Events at Time 0 ---
Transitions this tick:
| PID | Old -> New
|-----|-------------
| 10  | NOT_ASSIGNED -> NEW
| 10  | NEW -> READY
|  1  | NOT_ASSIGNED -> NEW
|  1  | NEW -> READY
|  1  | READY -> RUNNING

Process State:
| NEW | READY | WAITING | RUNNING | TERMINATED |
|-----|-------|---------|---------|------------|
|  -  |   10  |    -    |    1    |     -      |
```

2. IdenticalPriorities Simulation

This simulation is used to address what happens when two processes of the same priority arrive at the same time. In this case, it will simply resort back to a secondary algorithm, which is just First Come First Served (FCFS). This is why, even though both processes arrive at the same time, in our input file, PID 10 shows up before PID 1, which means that it will be scheduled first.

```
--- Memory / Events at Time 0 ---
Transitions this tick:
| PID | Old -> New
|-----|-------------
| 10  | NOT_ASSIGNED -> NEW
| 10  | NEW -> READY
|  1  | NOT_ASSIGNED -> NEW
|  1  | NEW -> READY
| 10  | READY -> RUNNING

Process State:
| NEW | READY | WAITING | RUNNING | TERMINATED |
|-----|-------|---------|---------|------------|
|  -  |   1   |    -    |    10   |     -      |
```

3. TooMany Simulation (BONUS)

This simulation is used to showcase what will happen if there are more programs queued than partitions (or when there is no partition size available to accommodate the program). In the output below, we can see the NEW state accumulate more programs over time. These programs sit until a partition of adequate size is available. Note this first screenshot shows no partitions available, and the second shows that there is a partition (15Mb) available, but none that will suit the needs of programs 8, 9 or 10, so they just sit there waiting until they are able to enter ready.

```
--- Memory / Events at Time 9 ---
Transitions this tick:
| PID | Old -> New
|-----|------------
| 10  | NOT_ASSIGNED -> NEW

Process State:
| NEW | READY | WAITING | RUNNING | TERMINATED |
|-----|-------|---------|---------|------------|
|  2  |   4   |    -    |    1    |     -      |
|  8  |   3   |    -    |    -    |     -      |
|  9  |   5   |    -    |    -    |     -      |
|  10 |   6   |    -    |    -    |     -      |
|  -  |   7   |    -    |    -    |     -      |

Partition Usage:
| Part | Size | Used | Unused | PID |
|------|------|------|--------|-----|
|  1   |  40  |  40  |   0    |  1  |
|  2   |  25  |  25  |   0    |  3  |
|  3   |  15  |  15  |   0    |  4  |
|  4   |  10  |  10  |   0    |  5  |
|  5   |  8   |  8   |   0    |  6  |
|  6   |  2   |  2   |   0    |  7  |
```

```
--- Memory / Events at Time 22 ---
Transitions this tick:
| PID | Old -> New
|-----|------------
|  4  | RUNNING -> TERMINATED
|  2  | READY -> RUNNING

Process State:
| NEW | READY | WAITING | RUNNING | TERMINATED |
|-----|-------|---------|---------|------------|
|  8  |   3   |    -    |    2    |     1      |
|  9  |   5   |    -    |    -    |     4      |
|  10 |   6   |    -    |    -    |     -      |
|  -  |   7   |    -    |    -    |     -      |

Partition Usage:
| Part | Size | Used | Unused | PID |
|------|------|------|--------|-----|
|  1   |  40  |  30  |   10   |  2  |
|  2   |  25  |  25  |   0    |  3  |
|  3   |  15  |  0   |   15   | -1  |
|  4   |  10  |  10  |   0    |  5  |
|  5   |  8   |  8   |   0    |  6  |
|  6   |  2   |  2   |   0    |  7  |
```

EP_RR

1. BasicPreemption Simulation

This simulation was run to ensure that the simulation would correctly preempt a running process as soon as a higher priority process arrives. In this particular case, PID 1 arrives 3ms into the execution of PID 10, preempting it back to the ready queue, and taking its place as the running process. PID 10 will only continue once PID 1 has finished.

Time of Transition	PID	Old State	New State
0	10	NEW	READY
0	10	READY	RUNNING
3	1	NEW	READY
3	10	RUNNING	READY
3	1	READY	RUNNING
8	1	RUNNING	TERMINATED
8	10	READY	RUNNING
10	10	RUNNING	TERMINATED

2. BasicQuantumReached Simulation

This simulation was run to ensure that the processes that are running correctly get preempted once they have reached the set quantum (in this case, the preprocessor macro was set to 5ms for simplicity: Many of the simulations use this value). Note that the processes were given the same priority. However, even with this fact, we can see that PID 10 still gets preempted at 5ms, instead of running until completion (6ms). PID 1 then also runs for 5ms, before running PID 10 for the remainder of it's duration (1ms).

Time of Transition	PID	Old State	New State
0	10	NEW	READY
0	10	READY	RUNNING
3	1	NEW	READY
5	10	RUNNING	READY
5	1	READY	RUNNING
10	1	RUNNING	READY
10	10	READY	RUNNING
11	10	RUNNING	TERMINATED
11	1	READY	RUNNING
12	1	RUNNING	TERMINATED

3. IdenticalPriorities Simulation

Once again, a simulation was run with two processes of equal priority. However, in this case, it is used to show that there is no preemption if a process arrives while another is running. Basically, EP only uses priorities to simply sort the ready queue before deciding which process goes, whereas EP_RR actually implements preemption. However, in this case where PIDs have the same priority, no preemption happens (at 3ms).

Time of Transition	PID	Old State	New State
0	10	NEW	READY
0	10	READY	RUNNING
3	1	NEW	READY
5	10	RUNNING	TERMINATED
5	1	READY	RUNNING
10	1	RUNNING	TERMINATED

4. SimultaneousRaceConditions Simulation

As described in the README.md, whenever a process is running, checks are done in a specific order, which gives priority to some transition over others:

For Running Processes:

1. Check if the currently running process is finished
2. Check if the currently running process needs to do I/O
3. Check if it has been preempted by another higher priority process
4. Check if quantum has been reached

Therefore, multiple simulations were run in order to prove or show that “finished > IO > preempt > quantum.” Basically, if a process finishes at the same time that it’s scheduled for IO, it will just finish, and if a process is scheduled for IO at the same time that it is preempted, it will do the IO instead of preempting and so on.

FinishAndIO

In this first simulation, we can see that the process is due for IO after 10ms (at the same time as it finishes), so the process just finishes (note there's a quantum reached in the middle):

Time of Transition	PID	Old State	New State
0	1	NEW	READY
0	1	READY	RUNNING
5	1	RUNNING	READY
5	1	READY	RUNNING
10	1	RUNNING	TERMINATED

IOAndPreemp

Similarly, in this simulation, a process is running and needs to do IO 5ms in. However, another process of higher priority arrives at that exact same time. We can see in the simulation output that the process simply goes to IO instead of being preempted (the new process is then run since there is nothing in the ready queue).

```
--- Memory / Events at Time 5 ---
Transitions this tick:
| PID | Old -> New
|-----|------------
|  1  | NOT_ASSIGNED -> NEW
|  1  | NEW -> READY
|  2  | RUNNING -> WAITING
|  1  | READY -> RUNNING

Process State:
| NEW | READY | WAITING | RUNNING | TERMINATED |
|-----|-------|---------|---------|------------|
|  -  |   -   |    2    |    1    |     -      |

Partition Usage:
| Part | Size | Used | Unused | PID |
|------|------|------|--------|-----|
|  1   |  40  |  0   |   40   | -1  |
|  2   |  25  |  0   |   25   | -1  |
|  3   |  15  |  10  |   5    |  1  |
|  4   |  10  |  10  |   0    |  2  |
|  5   |  8   |  0   |   8    | -1  |
|  6   |  2   |  0   |   2    | -1  |
```

PreempAndQuantum

Finally, this is supposed to demonstrate that if a process expires due to quantum and gets preempted any a new arrival at the same time, the preemption of the higher priority process will occur first. However, this is not visible in the outputs since they act the same (new higher priority process runs, meanwhile the old process gets sent to ready).

```
--- Memory / Events at Time 5 ---
Transitions this tick:
| PID | Old -> New
|-----|-------------
|  1  | NOT_ASSIGNED -> NEW
|  1  | NEW -> READY
|  2  | RUNNING -> READY
|  1  | READY -> RUNNING

Process State:
| NEW | READY | WAITING | RUNNING | TERMINATED |
|-----|-------|---------|---------|------------|
|  -  |   2   |    -    |    1    |     -      |

Partition Usage:
| Part | Size | Used | Unused | PID |
|------|------|------|--------|-----|
|  1   |  40  |  0   |   40   | -1  |
|  2   |  25  |  0   |   25   | -1  |
|  3   |  15  |  10  |   5    |  1  |
|  4   |  10  |  10  |   0    |  2  |
|  5   |  8   |  0   |   8    | -1  |
|  6   |  2   |  0   |   2    | -1  |
```

RR

1. BasicQuantumIO Simulation

This simulation was used to showcase that, even after a device has been preempted, it will still do I/O once the frequency has been reached. In this case, PID 1 is scheduled to do I/O every 6ms, however, it gets preempted after 5 (as is the nature of Round Robin and its quantum). But, if we look at the output, we can see that PID 1 will still proceed to do I/O after the next 1ms of execution (when 6ms is reached).

Time of Transition	PID	Old State	New State
0	1	NEW	READY
0	1	READY	RUNNING
5	1	RUNNING	READY
5	1	READY	RUNNING
6	1	RUNNING	WAITING
12	1	WAITING	READY
12	1	READY	RUNNING
17	1	RUNNING	READY
17	1	READY	RUNNING
18	1	RUNNING	WAITING
24	1	WAITING	READY
24	1	READY	RUNNING
29	1	RUNNING	READY
29	1	READY	RUNNING
30	1	RUNNING	WAITING
36	1	WAITING	READY
36	1	READY	RUNNING
38	1	RUNNING	TERMINATED

2. BasicQuantumReached Simulation

Once again, this simulation was run to ensure that the processes that are running correctly get preempted once they have reached the set quantum (in this case, 5ms).

Time of Transition	PID	Old State	New State
0	10	NEW	READY
0	10	READY	RUNNING
3	1	NEW	READY
5	10	RUNNING	READY
5	1	READY	RUNNING
10	1	RUNNING	READY
10	10	READY	RUNNING
11	10	RUNNING	TERMINATED
11	1	READY	RUNNING
12	1	RUNNING	TERMINATED

3. LargeQuantum Simulation

This simulation serves the purpose of demonstrating that if a large enough quantum is used (where it is larger than either the IO frequency or CPU time – essentially just bigger than the CPU burst of each process), then it will simply act as a FCFS scheduling algorithm.

Time of Transition	PID	Old State	New State
0	10	NEW	READY
0	10	READY	RUNNING
2	1	NEW	READY
3	2	NEW	READY
6	10	RUNNING	TERMINATED
6	1	READY	RUNNING
12	1	RUNNING	TERMINATED
12	2	READY	RUNNING
18	2	RUNNING	TERMINATED

```
========== Scheduling Metrics ==========
Throughput:              4.500000 ms/process
Average Turnaround Time: 11.500000 ms
Average Wait Time:       5.500000 ms
Average Response Time:   5.500000 ms
========================================
```

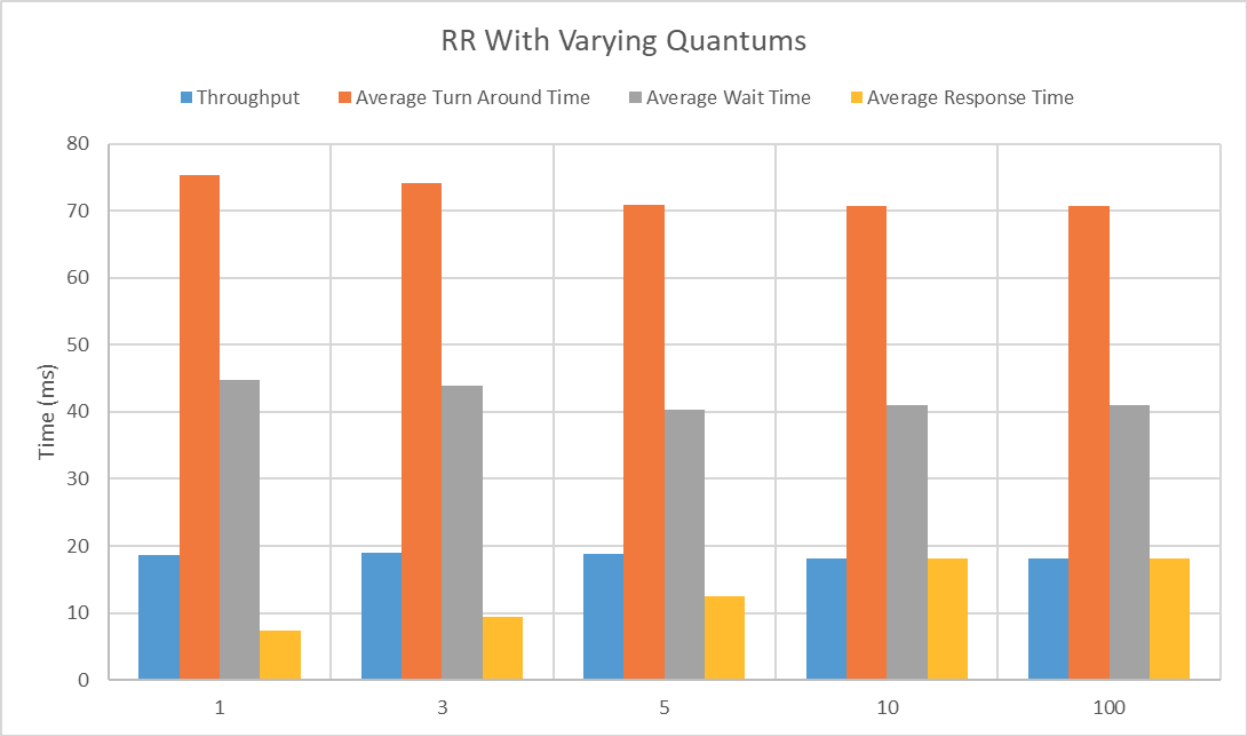
4. SmallQuantum Simulation

This simulation shows how having a very small quantum will lower your response times, but at the expense of longer wait times and turnaround times due to the number of transitions. Note this simulation does not take into account the amount of time it takes to do all these context switches.

```
|                  7 |  2 |     READY |   RUNNING |
|                  8 |  2 |   RUNNING |     READY |
|                  8 |  1 |     READY |   RUNNING |
|                  9 |  1 |   RUNNING |     READY |
|                  9 | 10 |     READY |   RUNNING |
|                 10 | 10 |   RUNNING |     READY |
|                 10 |  2 |     READY |   RUNNING |
|                 11 |  2 |   RUNNING |     READY |
|                 11 |  1 |     READY |   RUNNING |
|                 12 |  1 |   RUNNING |     READY |
|                 12 | 10 |     READY |   RUNNING |
|                 13 | 10 |   RUNNING |TERMINATED |
|                 13 |  2 |     READY |   RUNNING |
|                 14 |  2 |   RUNNING |     READY |
|                 14 |  1 |     READY |   RUNNING |
|                 15 |  1 |   RUNNING |     READY |
|                 15 |  2 |     READY |   RUNNING |
|                 16 |  2 |   RUNNING |     READY |
|                 16 |  1 |     READY |   RUNNING |
|                 17 |  1 |   RUNNING |TERMINATED |
|                 17 |  2 |     READY |   RUNNING |
|                 18 |  2 |   RUNNING |TERMINATED |
+-------------------------------------------------+


========== Scheduling Metrics ==========
Throughput:              4.500000 ms/process
Average Turnaround Time: 14.500000 ms
Average Wait Time:       8.500000 ms
Average Response Time:   0.500000 ms
========================================
```

5. VaryingQuantum Simulations



After running the two previous simulations, we decided it may be a good idea to analyze the quantum further, to see what will happen as it increases. We determined that throughput was essentially the same (although in theory it would get worse due to more frequent context switches). Additionally, we found that the average wait time decreased, along with the average turnaround time, but at the cost of the average response times increasing. Note that it would do so until it hits a specific point, at which point it is simply FCFS and remains the same (10 and 100 were identical, since all process CPU bursts were at or under 10).

Various Use Case Outputs

EP

1. CPU Bound
Throughput: 18.00 ms/process
Avg Turnaround Time: 67.17 ms
Avg Wait Time: 43.17 ms
Avg Response Time: 49.17 ms
2. IO Bound
Throughput: 22.50 ms/process
Avg Turnaround Time: 79.17 ms
Avg Wait Time: 28.33 ms
Avg Response Time: 29.17 ms
3. Mixed
Throughput: 22.50 ms/process
Avg Turnaround Time: 72.00 ms
Avg Wait Time: 26.17 ms
Avg Response Time: 25.17 ms

EP is effective in scenarios where completion of high-priority tasks is important, like in the case of a hard real-time system (where deadlines must be met, and missing said deadline is considered a system failure with major consequences). However, its non-preemptive nature makes it less suitable for such cases as high priority processes will still have to wait, sometimes, for very long, unimportant processes. As such, the process could be stuck hogging the CPU for an extended period of time.

RR

1. CPU Bound

Throughput: 18.00 ms/process
Avg Turnaround Time: 71.67 ms
Avg Wait Time: 46.33 ms
Avg Response Time: 16.33 ms

2. IO Bound

Throughput: 22.50 ms/process
Avg Turnaround Time: 76.17 ms
Avg Wait Time: 29.83 ms
Avg Response Time: 12.50 ms

3. Mixed

Throughput: 22.67 ms/process
Avg Turnaround Time: 74.50 ms
Avg Wait Time: 30.00 ms
Avg Response Time: 14.50 ms

RR is ideal for interactive systems where fairness and low response times are important. It handles both CPU-bound and I/O-bound processes well due to its time-slicing or preemption every quantum, ensuring that every process has a fair change.

EP_RR

1. CPU Bound

Throughput: 18.00 ms/process
Avg Turnaround Time: 71.67 ms
Avg Wait Time: 50.67 ms
Avg Response Time: 43.17 ms

2. IO Bound

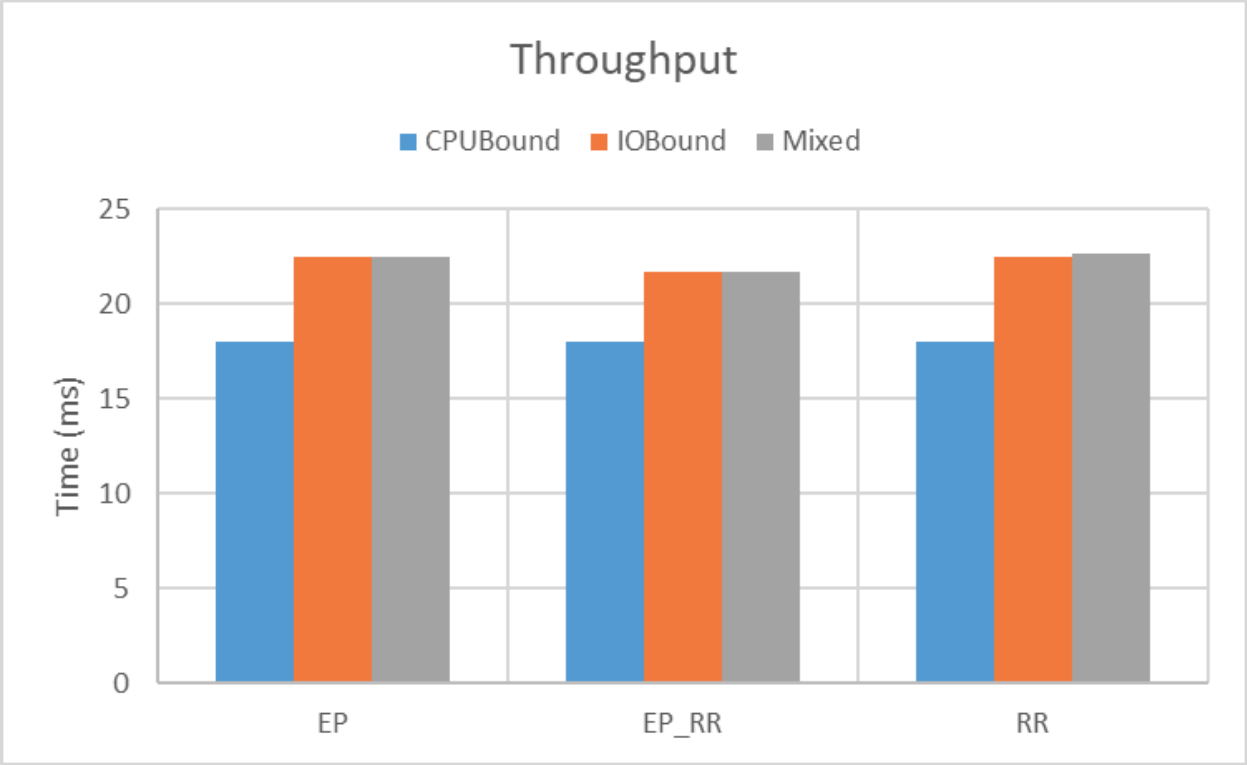
Throughput: 21.67 ms/process
Avg Turnaround Time: 62.33 ms
Avg Wait Time: 33.00 ms
Avg Response Time: 27.33 ms

3. Mixed

Throughput: 21.67 ms/process
Avg Turnaround Time: 80.50 ms
Avg Wait Time: 34.17
Avg Response Time: 30.50 ms

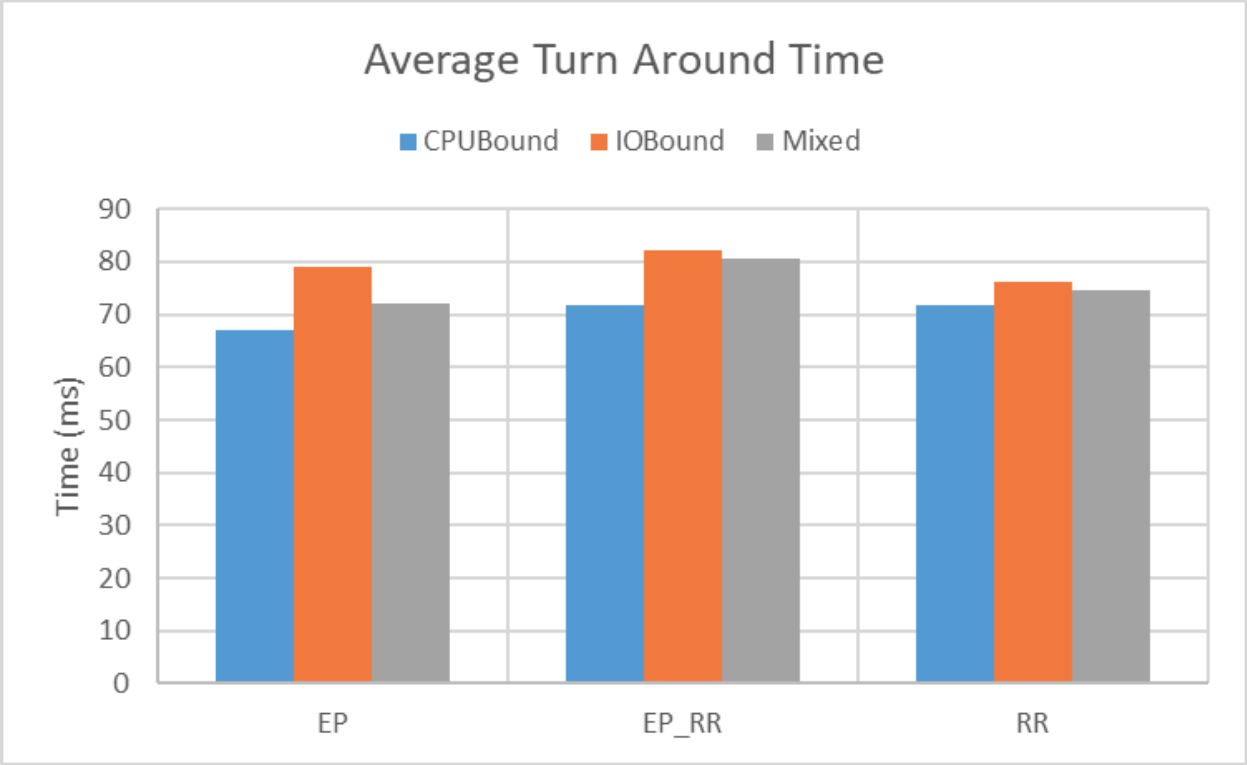
The EP_RR scheduler tries to combine priority scheduling with Round Robin, but this can sometimes not only make things more complicated but often results in more context switches. As a result, it may have higher average wait and turnaround times than a Round Robin scheduler. However, EP_PR has two advantages over RR: The first is the fact that there is preemption, meaning that if you have a higher priority task that needs to be executed immediately, it can. Additionally, it ensures that long processes don't hog the CPU (as is possibly the case in EP depending on the process length). If you want low response times and fairness for all tasks, Round Robin often does better. EP_RR can be useful if some tasks need to go first, but it needs careful setup and may add extra overhead.

Average Throughput



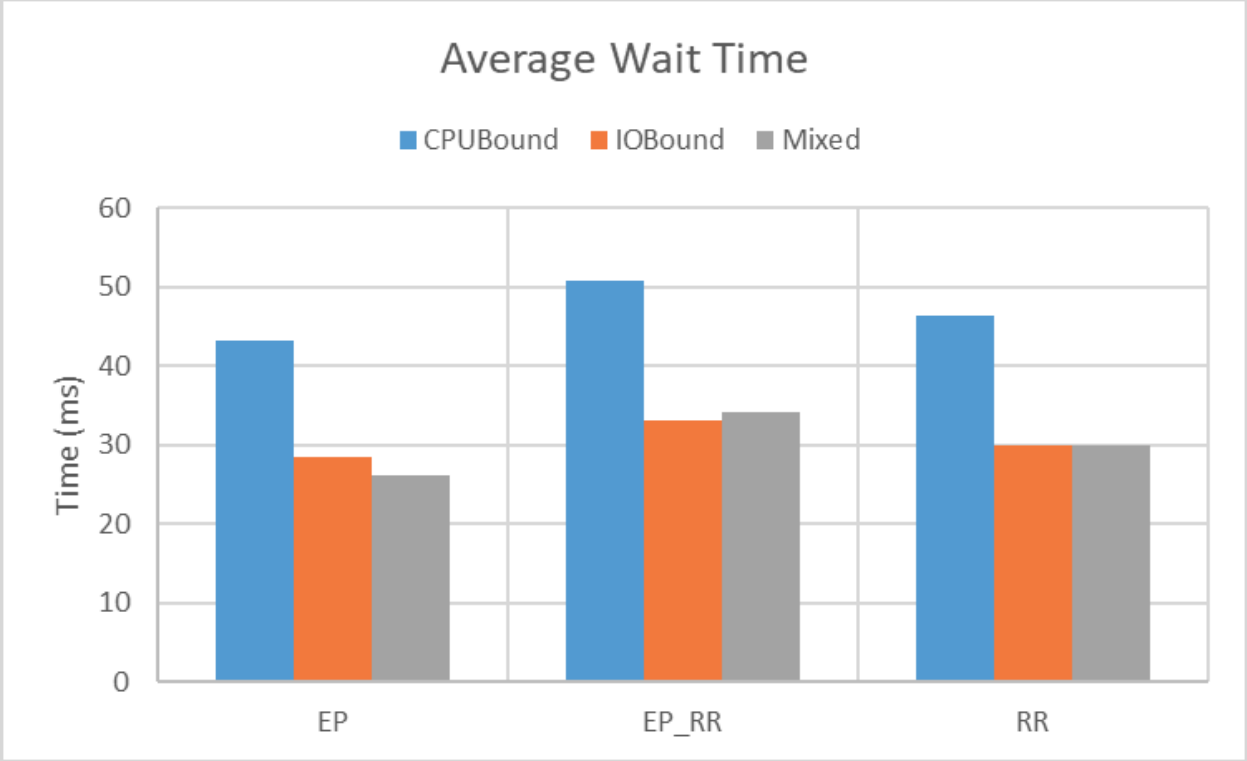
All algorithms show similar throughput across all tests; however, I/O bound and mixed use cases seem to be slightly larger. This makes sense since the processes must wait in the WAITING state, when they could be in the READY queue. Additionally, we see little to no difference between the mixed and I/O bound use cases. This is because the CPU is very efficient in always staying busy even when I/O is running in the background: Both of the throughputs are mostly dependent on the total CPU time, since the I/O operations are able to be handled in the background, not hindering or bottlenecking the CPU.

Average Turn Around Time



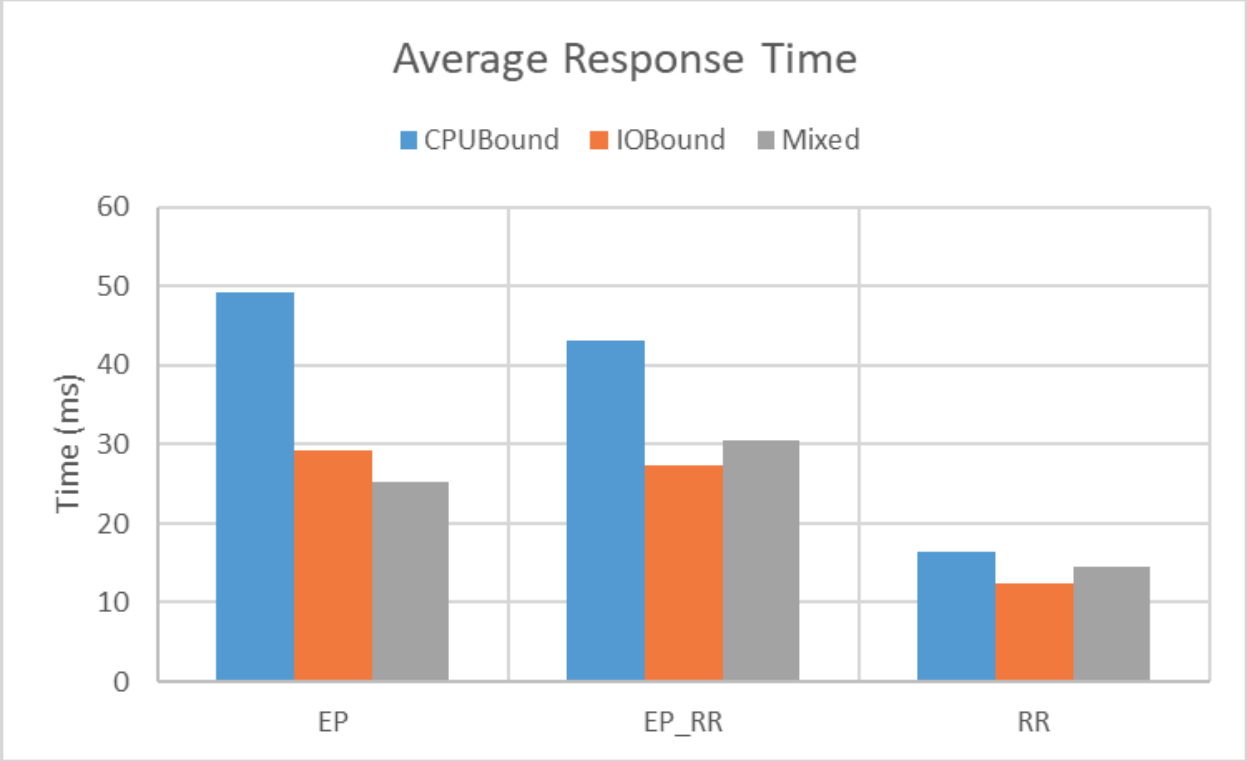
This diagram shows that I/O bound and mixed workloads have higher average turnaround times than CPU bound ones for all schedulers. EP_RR has the highest turnaround times overall. This happens because processes doing more I/O spend extra time waiting for I/O to finish, which increases their total turnaround time. EP_RR's additional preemption can add even more waiting, while RR and EP are a bit more efficient for these workloads. We can see that EP performs the best in most situations since it does not have any preemption, closely followed by Round Robin, which usually did slightly worse due to the preemption, but had the most consistent results across all use cases.

Average Wait Time



CPU-bound processes have the highest average wait times in all schedulers, while I/O-bound and mixed workloads wait less. This is because CPU-bound tasks spend more time in the READY queue, waiting for their turn, effectively doing “useless” work. EP_RR shows the highest wait times overall, especially for CPU-bound cases, likely due to its preemption and quantum. This is followed by Round Robin, once again, probably due to its quantum and therefore forcing processes back to the ready state. I/O-bound and mixed processes wait less since they spend more time doing I/O instead of waiting in the READY queue.

Average Response Time



RR gives the lowest response times for all workloads, meaning processes start running quickly after arriving. EP and EP_RR have much higher response times, especially CPU-bound tasks. In the case of EP, this is because if the ready process is of low priority, it may have to wait a long time before running. EP_RR also performs worse for the same reason. However, it is slightly better due to the quantum that is implemented. I/O-bound and mixed workloads respond faster since they spend less time waiting to be scheduled (in ready). Overall, RR is best for quick responses, while EP and EP_RR may slow down some tasks, especially those that don't have high priority.

Part 2 – Concurrent Processes in Unix

https://github.com/e95400411-cmd/SYSC4001_A3P2.git

Part 3 – Concepts

Student 1: Antoine Hickey

Student 2: Enzo Chen

1. Demand paging memory management

i, ii. Page fault algorithms

Page reference string	415	305	502	417	305	415	502	518	417	305	415	502	520	518	417	305	502	415	520	518
Requests:	20																			
Algorithm																				
FIFO																				
	415		417		417		417		305		305		305		518		518		518	
3 Frames	305		305		415		415		415		502		502		502		417		417	
	502		502		502		518		518		518		520		520		520		305	
Page Faults	3		1		1		1		1		1		1		1		1		1	
	502		502		502		518		Total:	12										
	417		415		415		415		Hit ratio:	40%										
	305		305		520		520													
Page Faults	1		1		1		1													
	415		518		518		518		518		502		502		Total:	10				
4 Frames	305		305		415		415		415		415		518		Hit ratio:	50%				
	502		502		502		520		520		520		520							
	417		417		417		417		305		305		305							
Page Faults	4		1		1		1		1		1		1							
LRU																				
	415		417		417		502		502		502		305		305		305		520	
3 Frames	305		305		305		305		518		518		518		415		415		415	
	502		502		415		415		415		417		417		417		502		502	
Page Faults	3		1		1		1		1		1		1		1		1		1	
	520		520		305		305		305		520		520		Total:	12				
	518		518		518		502		502		502		518		Hit ratio:	40%				
	502		417		417		417		415		415		415							
Page Faults	1		1		1		1		1		1		1							
	415		415		415		305		305		305		305		518		518		518	
4 Frames	305		305		417		417		417		417		520		520		520		520	
	502		502		502		502		415		415		415		415		417		417	
	417		518		518		518		518		502		502		502		502		305	
Page Faults	4		1		1		1		1		1		1		1		1		1	
	518		415		415		415		Total:	13										
	502		502		502		502		Hit ratio:	35%										
	417		417		520		520													
	305		305		305		518													
Page Faults	1		1		1		1													
Optimal																				
	415		415		502		518		518		518		518		518		518		518	
3 Frames	305		305		305		305		415		502		520		520		520		520	
	502		417		417		417		417		417		417		305		502		415	
Page Faults	3		1		1		1		1		1		1		1		1		1	
	Total:	12																		
	Hit ratio:	40%																		
	415		415		502		520		520		520		Total:	9						
4 Frames	305		305		305		305		502		502		Hit ratio:	55%						
	502		518		518		518		518		518									
	417		417		417		417		417		415									
Page Faults	4		1		1		1		1		1									

Student 1: Explain the LRU Algorithm

The LRU or Least Recently Used algorithm is page replacement algorithm or technique that involves removing the page that has not been used/accessed for the longest period. In other words, LRU assumes that the pages that are older, and haven't been used in a long time, will not be used in the future. It could accomplish this by using a timestamp to know when the last use occurred. LRU is therefore similar to FIFO, since they both remove the older pages. However, where they differ is the fact that for FIFO, a page that is used frequently can still be swapped out since although it is used recently, it did not enter (from external storage) recently. LRU updates the timestamp every time a page is accessed, protecting frequently used pages from replacement regardless of when they first arrived in memory. Note however, that timestamps may require far more overhead since they keep track of the count (time) both in memory and in the TLB. For that reason, it is sometimes better to simply use an "old/new" bit, where the programs that haven't been used are set to 0 (new) or otherwise 1 (old). Then, after some given time (usually under a millisecond), the timer ISR would reset all of them to 0, in order to give the older pages a better chance of staying (since the older pages will be swapped out more). This decreases the overhead needed immensely.

iii. Explanations

With both 3 and 4 frames, the Optimal page replacement algorithm yields the best performance because it always selects the page that will not be used for the longest time in the future. However, Optimal is not possible in real-world operating systems because it requires perfect knowledge of future memory accesses.

One note is that as the number of frames increases, page faults generally decrease (inverse relationship). This is because more frames allow more pages to be stored in memory simultaneously, reducing the need for replacements. In theory, with infinite memory (pages), there would be no page faults at all.

This being said, between FIFO and LRU, LRU typically performs better because it uses the assumption that recently used pages are more likely to be accessed again soon. FIFO, which swaps out the oldest page regardless of usage, may perform worse in these cases where pages are reused often. However, FIFO can outperform LRU in patterns where older pages are genuinely less likely to be reused, such as in cyclic workloads (as was seen in pt3 q1. However, in general we should expect LRU to perform better.

Student 2: Explain the LFU Algorithm

LFU, or Least Frequently Used, is a page replacement counting algorithm, which replaces the page in memory that has been used the least. As this algorithm is a counting algorithm, it associates a counter to each page, updating it every time the page has been accessed. When the CPU needs a page, it checks the page table to determine if the page is stored in memory. If the page is not in memory, a page fault occurs, and the control unit part of the CPU fetches the page from the disk. If there is no free frame available in memory, using the LFU algorithm, the CPU selects the page in memory with the smallest count to become the victim, replacing it with the fetched page from the disk. The problem with LFU is that it tends to replace pages that have recently been loaded into memory, since they have been accessed only a few times. This causes some older and potentially no longer needed pages to be left in memory, as their counters are high enough that they will never be replaced. A solution for this problem is to use a timer ISR and to reset the counts after a certain amount of time. This can also shrink the number of bits used for the counters, as resetting the counters frequently means each counter is less likely to reach its maximum value, as there is less time in between resets for parges to be accessed. As with other counting algorithms, LFU is expensive to implement as it requires a lot of space both in memory and on the TLB to store a counter for each page.

2. Memory mapping on page basis

a.

Without a TLB, a paged memory reference would take two main memory accesses: one to check the page table and one to access the page. Since one main memory access takes $m = 120ns$, the total time required to do a paged memory reference is:

$$2 \cdot m = 2 \cdot 120 = 240ns$$

b.

If we add a TLB with a time of $\epsilon = 20ns$, and assuming a hit ratio $\alpha = 95\%$, the Effective Memory Access Time (EAT) is calculated as follows:

$$\begin{aligned} & EAT \\ & = (m + \epsilon)\alpha + (2m + \epsilon)(1 - \alpha) \\ & = (120 + 20)0.95 + (2(120) + 20)0.05 \\ & = 146ns \end{aligned}$$

To access a page that is found in the TLB, it takes $m + \epsilon$ time, since it must access the TLB once to find the page, then it must use main memory once to access the page. To access a page that is not found in the TLB, it takes $2m + \epsilon$ time, since it must check the TLB to see if the page is there, then it must use the page table in memory to find the page, and then it must access the page in memory. Then, a weighted average of these two scenarios is taken, with the hit ratio corresponding to the percentage of times the page is found in the TLB.

c.

Adding the TLB generally improves performance because accessing the TLB is much faster for the CPU to access than main memory. If the CPU can find a page's entry in the TLB, it saves one memory access, as it no longer needs to check the page table in main memory, and therefore only needs to use main memory once to access the page. This can save almost half the time in accessing a page.

As the TLB is a small piece of cache connected directly to the CPU, it is expensive and therefore is too small to store the entire page table. If the page's entry, is not in the TLB, the CPU must still use main memory twice, to check the page table, and then to access the page. In this scenario, the TLB actually slows the process of accessing a page, as it introduces the overhead of the CPU checking it, without the benefit of finding the page's entry and saving a memory access.

In reality, both scenarios happen, and the TLB hit ratio dictates the relative frequency of them. If the hit ratio is too low, meaning the CPU rarely finds the required page entry in the TLB, the Effective Memory Access Time can be worse with the TLB than without one. In this case, the CPU needs to check the TLB, then access memory twice, which is slower than accessing memory twice. This, however, is rare, as the hit ratio tends to be high enough that enough memory accesses are saved to improve the Effective Memory Access Time compared to without the TLB. Therefore, the TLB will improve the Effective Memory Access Time, unless the TLB hit ratio is extremely low.

3.

a.

The format of the processor's logical address is $\log_2 128 = 7$ bits, followed by $\log_2(4 \cdot 2^{10}) = 12$ bits. As it takes 7 bits to select the page, followed by 12 bits to select the word (assumed to be 1 byte), the processor's total logical address length is 19 bits.

b.

The page table length is the number of pages within the logical address space, which is 128. The page table's width, disregarding control bits, is calculated by determining the number of bits required to store the total amount of frames. Since there are $\frac{512\ KiB\ physical\ memory}{4\ KiB\ page\ size} = 128$ frames, the page table width is $\log_2 128 = 7$ bits.

c.

Now, if the physical memory is halved, $\frac{256\ KiB\ physical\ memory}{4\ KiB\ page\ size} = 64$ frames, the page table width is $\log_2 64 = 6$ bits. Therefore, the page table width is 1 bit smaller when the physical memory is halved.

Student 1: Explain what is the physical memory space

Physical memory refers to the actual hardware memory (i.e. RAM) available in a computer system. They are typically measured in bytes in sequential memory locations, each with a unique physical address. The operating system manages physical memory, allocating some of it to itself (monitor), and other sections to processes that are loaded from hard drive. Note that the programs themselves will not get the physical addresses, but logical memory: Physical and virtual memory are the same to the process (later explained by student 2).

Student 2: Explain what is the logical memory space

The logical memory space is the memory space as seen by the programmer and programs. It is a large, contiguous array, abstracted from physical hardware details. Each program has its own logical memory pace, starting from address 0, which is then mapped onto the physical hardware by the MMU. As logical memory is abstracted from the hardware, it can be larger than physical memory and therefore can be spread across both the memory and the disk.

4.

When the lseek system call is run, it creates a software interrupt, or trap, which stops the execution of the current process and context switches the CPU to kernel mode. The CPU then uses the provided interrupt vector to access the vector table, finding the start of the corresponding ISR. It then runs the ISR, which services the lseek call. The CPU uses the file descriptor (fd) index to accesses the process's file descriptor table in the process's PCB. From there, it accesses the file record's system-wide open-file table to get the file control block (inode) pointer. It then accesses the inode to get the size of the file. From there, the CPU calculates the new logical address of the file pointer as 0 (start of file) + file size (from inode) + offset (parameter). It then updates the current read/write pointer in the file record in the system-wide open-file table. After, it returns the result of the calculation as an off_t struct. Finally, another context switch occurs, switching back to user mode and transferring control back to other processes. Using lseek, we can move the pointer past the end of the file. In combination with write, we can use this to create sparsely allocated files, with empty space (holes) between the previous end of the file and the new location of the file pointer.

5.

a.

Assume the file offset word length does not limit the file size.

Let block size = $8\ Kb$ = 1 KB

Direct:

$$12 \cdot 1\ KB\ =\ 12\ KB$$

Single:

As single indirect points to one block which contains pointers to other blocks, there is 1 KB of space for pointers.

$$\frac{10^3\ B\ for\ pointers}{4\ B\ per\ block\ pointer}$$
$$= 2.5 \cdot 10^2\ block\ pointers$$
$$= 2.5 \cdot 10^2 \cdot 10^3\ bytes\ per\ block$$
$$= 250\ KB$$

Double:

Double indirect has the same space for pointers as single indirect does for storing data, 250 KB.

$$\frac{250 \cdot 10^3\ B\ for\ pointers}{4\ B\ per\ block\ pointer}$$
$$= 6.25 \cdot 10^4\ block\ pointers$$
$$= 6.25 \cdot 10^4 \cdot 10^3\ bytes\ per\ block$$
$$= 62.5\ MB$$

Triple:

Triple indirect has the same space for pointers as double indirect does for storing data, 62.5 MB.

$$\frac{62.5 \cdot 10^6\ B\ for\ pointers}{4\ B\ per\ block\ pointer}$$
$$= 1.5625 \cdot 10^7\ block\ pointers$$
$$= 1.5625 \cdot 10^7 \cdot 10^3\ bytes\ per\ block$$
$$= 15.625\ GB$$

Therefore, the total maximum file size is $12\ KB + 250\ KB + 62.5 \cdot 10^3\ KB + 15.625 \cdot 10^6\ KB = 15\ 687\ 762\ KB$, or approximately 15.69 GB.

b.

If you need to store a file larger than the maximum possible size as calculated earlier, you would need to copy any existing files off the drive and reformat the drive to increase the block size. This would greatly increase the maximum possible file size, since the blocks would not only get larger, but the blocks containing the pointers would also get larger, increasing the number of blocks which can be allocated. For example, for a block size of 2 KB instead of 1 KB:

Direct:

$$12 \cdot 2\ KB\ =\ 24\ KB$$

Single:

As single indirect points to one block which contains pointers to other blocks, there is 1 KB of space for pointers.

$$\frac{2 \cdot 10^3\ B\ for\ pointers}{4\ B\ per\ block\ pointer}$$
$$= 5 \cdot 10^2\ block\ pointers$$
$$= 5 \cdot 10^2 \cdot (2 \cdot 10^3)\ bytes\ per\ block$$
$$= 1\ MB$$

Double:

Double indirect has the same space for pointers as single indirect does for storing data, 250 KB.

$$\frac{10^6\ B\ for\ pointers}{4\ B\ per\ block\ pointer}$$
$$= 2.5 \cdot 10^5\ block\ pointers$$
$$= 2.5 \cdot 10^5 \cdot (2 \cdot 10^3)\ bytes\ per\ block$$
$$= 500\ MB$$

Triple:

Triple indirect has the same space for pointers as double indirect does for storing data, 62.5 MB.

$$\frac{500 \cdot 10^6\ B\ for\ pointers}{4\ B\ per\ block\ pointer}$$
$$= 1.25 \cdot 10^8\ block\ pointers$$
$$= 1.25 \cdot 10^8 \cdot (2 \cdot 10^3)\ bytes\ per\ block$$
$$= 250\ GB$$

Therefore, the total maximum file size is now $24\ KB + 10^3\ KB + 500 \cdot 10^3\ KB + 250 \cdot 10^6\ KB = 250\ 501\ 024\ KB$, or approximately 250.5 GB. By doubling the block size, the total maximum file size has increased by a factor of 16. This is because the triple indirect block can hold twice as many double indirect pointers, which each can hold twice as many single indirect pointers, which each can hold twice as many direct pointers, which each can hold twice as much data. This makes for an approximately $2^4 = 16$ times increase in file size, since the double, single, and direct pointers do not have a large impact on the maximum file size.