Swapping Policies Between Main Memory and Secondary Memory

Introduction:

Before 1955 computers were forced to complete programs using punch cards in a linear fashion as computers did not have enough (or any) temporary storage (Random Access Memory (RAM)) to storage programs so they could be executed in a linear or semi random fashion. With the introduction of RAM (temporary volatile storage) in 1948 computers started to become more dynamic as the introduction of RAM allowed for quicker access to complied code in comparison to making disk requests. These disk requests used to be processed by manually altering the configurations of a computer's magnetic storage (disk memory), or by manually placing a pile of punch card into a machine. The first "usable" RAM component was introduced by Professor Fedrick Williams in 1948 and allowed the temporary storage of 1024 bits of information. This development was revolutionary even though it only allowed the storage of 1024 different operations (assuming each operation is one bit) as it minimized the reduction of some disk requests. Since 1948, the storage capacity of RAM has exploded. It is not uncommon to see modern (as of 2021) computers with storage capacity of 16GB and 32GB (17,179,869,184 bits and 34,359,738,368 bits respectively) which would equate to an increase of 16,777,216 and 33,554,432 times respectively[1]. Even with this 16-32 million time increase in storage capacity of data, modern operating systems are still forced to make disk requests. Since CPU requests to disk can be up to 1000 times slower than issuing a request to main memory we would inherently want to optimize speed and performance of our computer by minimizing disk requests[2].

Non complied code is compiled and then is loaded into main memory by the loader (cite this). This loader will and can load any code into main memory. Since the loader can load any code into main memory, we would want to prioritize certain programs over others. For example, if one

computer is acting maliciously (over utilization of resources, or information stealing) we would want to kill this program with a kill request. On the flip side, we would want to remove code that is just sitting in main memory waiting for other code to compile so it can run (an I/O request).

This leads us to the crux of our issue, what is the optimal algorithm that the loader and other computer compartments can utilize to place non complied code into main memory when main memory is full. When main memory is full, complied code can (but doesn't have to) be brought in from secondary memory in a random or last in last out manner. To test this theory, this paper will look at the quantitative (via a controlled experiment) and qualitative implications behind these two swapping policies.

Experiment Setup:

Our simulation will best try to mimic the process Appendix 1 is trying to illustrate. That is swapping complied code in and out of memory based a certain set of criteria:

1. RAM Sized is fixed. This will be mimicked by setting the size of a non-resizable array to size 6. This means only 6 programs (or length 1) can be placed into memory.
2. We will allow some space to be temporary stored in a "buffer" between main and secondary memory to represent buffer storage space. The size of this program will be 2.
3. We will have a finite amount of disk space as well. The size of this space will be 10.
4. Kill requests are allowed. This means when we create a program we have the option to set one of the programs to "true" in our ".add" and ".add_back" methods. This means that this program will be sent to the front of the Main memory queue. If this does happen, a program will be kicked out of main memory if main memory is full.

5. I/O requests are allowed as well. These I/O requests will be added when they are created. When a program is loaded into our fictitious CPU and the program is labelled with a I/O set to true and the program length is greater than one, then when the I/O program is run, this program will wait. When waiting this program is swapped out to simulate what a real program will do.

Based on these set of criteria we can compare our two swapping policies. Our swapping policies will based on runtime speed using the "high_resolution_clock::now();" method. In addition to this, we will also consider qualitative factors that could affect the overall program performance.

In addition to the above, we all want to see the increase in computation time of our complied code when we add functionality to our programs such as kill requests and I/O requests.

Tests:

We have discussed what our tests will be comprise of, now we will discuss what type of tests we will run. We will run four different tests to simulate the worst-case scenario for this program (multiple I/O and Kill Requests) up to and including the best-case scenario (no I/O and no Kill Requests).

For each test (we will perform four tests in total) we will perform six subtests. Each subtest will perform the same operations as all other tests except we will double the iterations amount up to and including 1600 iterations. We will score the aggregate amount of time the code took to compile for each subset and plot it on each chart for both swapping policies for all four tests.

Test 1:

The first test we will run is the best-case scenario. The best-case scenario will occur when have no I/O and no Kill requests. This is the best-case scenario since not issuing an I/O or kill request will not result in a disk call or a main memory transfer to buffer storage. This test is used as our base test since this test will mirror a first in first out process since no programs need to be swapped. The results of this test can see in appendix 2.

In appendix 2, the random swap algorithm preformed slightly worse than the last in last out algorithm on all test cases expect for the first subtest. This is line with what we are expecting since both algorithms should perform in a similar fashion since we do not need to swap any program out of main memory. The runtime difference between these two algorithms is small and can most likely be contributed to how my computer's operating software is running this test.

Test 2:

For test 2, we preformed two kill requests and no I/O requests. Here the random swapping policy preformed worse on all six tests when compared to the last in last out swapping policy. This is in line with our expectations since the random swapping algorithm will be forced to generate a random number each time, we make a swap which adds more lines of complied code than our non-random swapping algorithm[3]. Over 16,000 iterations this extra complied code can add up.

Test 3:

The third test we ran was comparing our two swapping policies when we issue two I/O requests. As we can see from them result of Appendix 4, both algorithmic swapping policies preformed similar in execution speed. There was a slight time advantage towards using the last in last out policy (since this minimize time revival of code for 16,000 test). These results were in line with test 1 expect for subtest 5 where swapping policy performed worse than the random swapping policy.

The last in last out policy (swap algorithm) performed better than the random policy algorithm because the random policy as stated before needs to make an additional CPU request to determine which code(s) line should be evicted from main memory into secondary memory.

Test 4:

For test 4, we preformed two Kill requests and two I/O requests. Here the random swapping policy preformed worse on all six subtests when compared to the last in last out swapping policy. This is in line with our expectations since the random swapping algorithm will be forced to generate a random number each time, we make a swap as stated above.

As expected, the run time of the complied programs increased from our best-case scenario to our worst-case scenario for both swapping policies. By adding I/O requests and kill requests we are forcing our CPU to preform more operations to complete our programs.

Hypothetically, we could issue a bunch of I/O and kill requests to the point where our OS will not be able to handle the amount of program requests. Since this is not optimal, we can limit the amount of kill and I/O requests a user can make during a certain period. This limitation also has

it downside since we could prevent a user from making a kill or I/O request when it is legitimately needed.

Summary of Tests:

Quantitative Analysis:

In general, for our four tests (with our six subsets for each test), the random swapping algorithm preformed slightly worse than our last in last out algorithm. As stated before, this in line with our expectations as the random swapping algorithm will require more operations since we need to generate a random number.

Qualitative Analysis:

Even though there are slight differences in terms of computation time between these two swapping algorithms there are also qualitative factors we should consider. Firstly, we should think about what happens when the random swapping algorithm removes a program needed to execute another program. This could occur (albeit unlikely to occur) since the random algorithm does not discriminate which algorithm is removed since the removal process is random. This removal process could result in unforeseen negative circumstances like a deadlock or the shutdown of an important program that the user is trying to run[4].

While the random swapping algorithm could cause unforeseen issues due to its randomness, the last in last out algorithm can also cause unforeseen circumstances. Since we know which program will be evicted, we could prevent certain programs from running. We can potentially counteract this sub optimal performance but limiting the amount of I/O and Kill requests we can make. But again, this checking policy will also slow down our computer performance since the CPU will needs to perform these checks[4].

Future work:

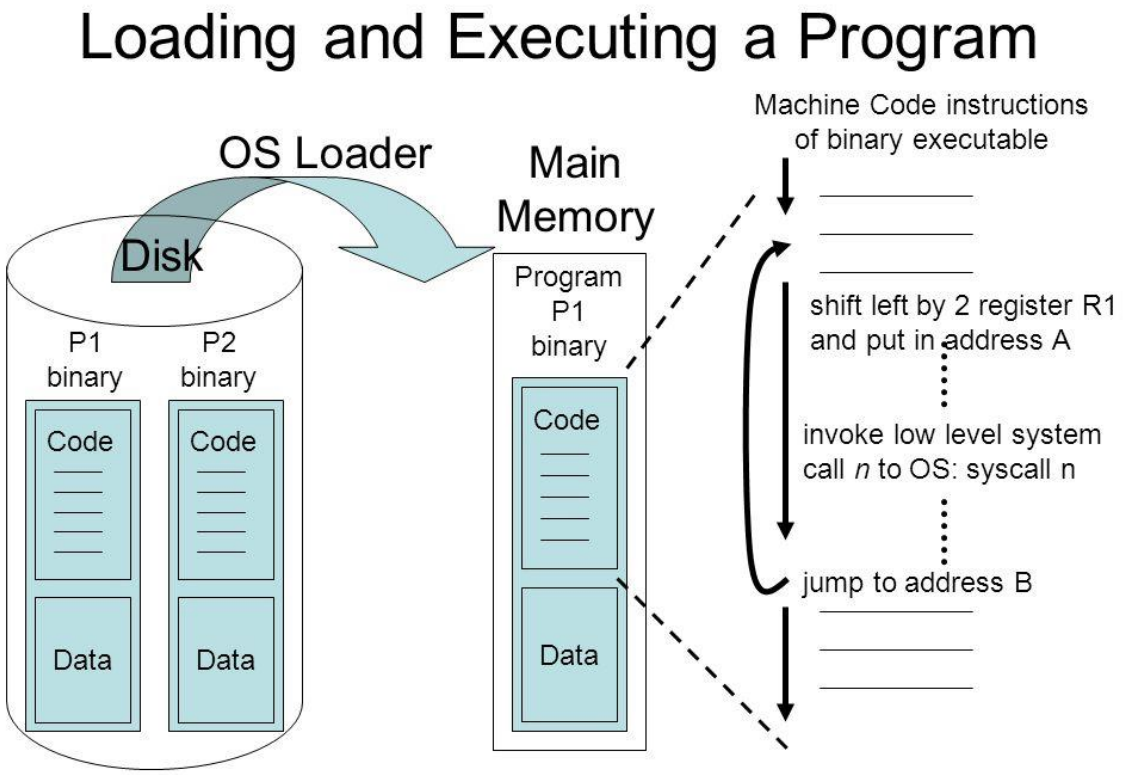
The code was written in a format that would best simulate how an operating system would perform different swapping policies when we have I/O and kill requests. The results are somewhat in line with what we expected, but the tests were limited. For example, when we make an I/O request we are not actually making a real I/O request we are just swapping out our lines of code in main memory. Since this process is happening in main memory, we are not perfectly simulating a real I/O request which would significantly slow down the performance of our simulations.

Another issue we are facing with these simulations is the windows 10 operating system dictates when the code is being complied.

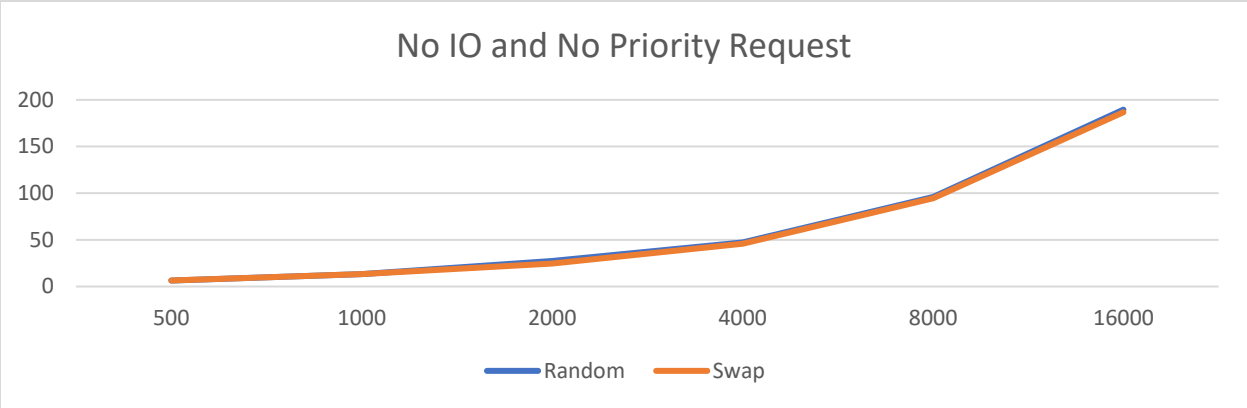
Lastly, there was variability in the results of our tests when we changed the position of the I/O and kill requests. For example, the runtime speed would slow down when we shifted the two I/O and two kill requests to be further apart from each other. In addition, when we placed these requests closer to the middle of the queue, the execution time also increased.

To fix the above problems, in the future we can run these programs on multiple different devices and operating systems (Linux, Unix, Windows) to minimize the effect underlying operating system[5]. We can also program more accurate I/O requests. An example of this would be opening a file then immediately closing it. Lastly, we can randomize the location of when we make our I/O and kill requests to make our tests more robust.

Appendix 1:


Loading and Executing a Program
OS Loader
Disk
P1
binary
P2
binary
Code
Data
Code
Data
Main
Memory
Program
P1
binary
Code
Data
Machine Code instructions
of binary executable
shift left by 2 register R1
and put in address A
invoke low level system
call n to OS: syscall n
jump to address B

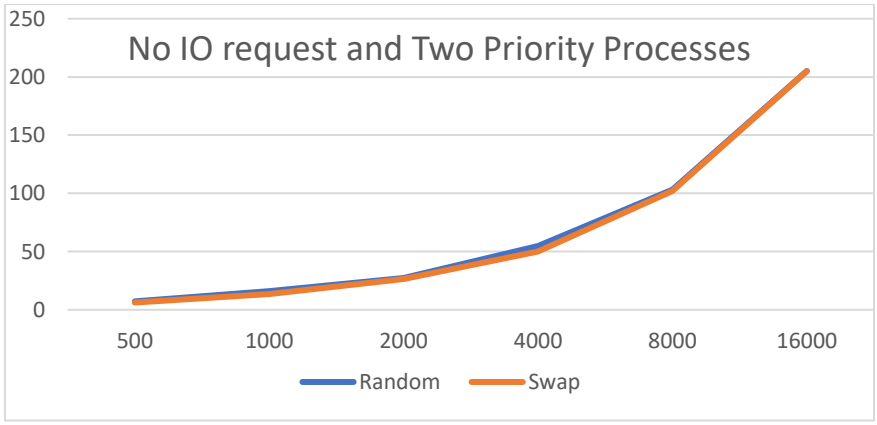
Appendix 2:



No IO request and No Priority

(Milliseconds)	500	1000	2000	4000	8000	16000
Random	6.3798	13.324	27.335	47.3367	95.9348	189.547
Swap	6.4466	13.24	24.71	45.981	94.5492	186.765

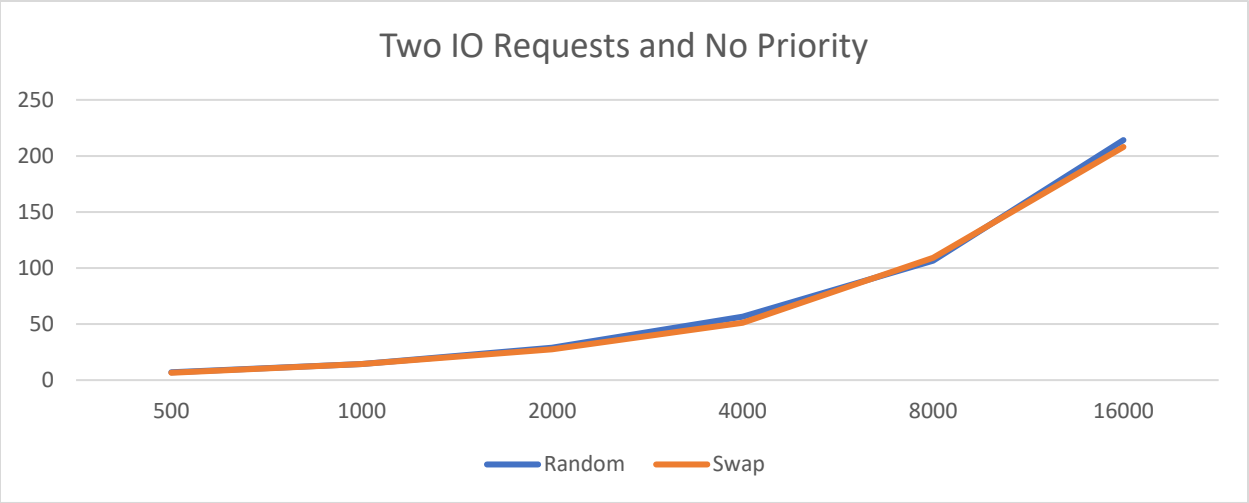
Appendix 3:



No IO request and Two Priority Processes

(Milliseconds)	500	1000	2000	4000	8000	16000
Random	7.23	16.066	27.249	54.853	103.08	205.078
Swap	6.225	13.541	26.57	49.9443	101.96	204.947

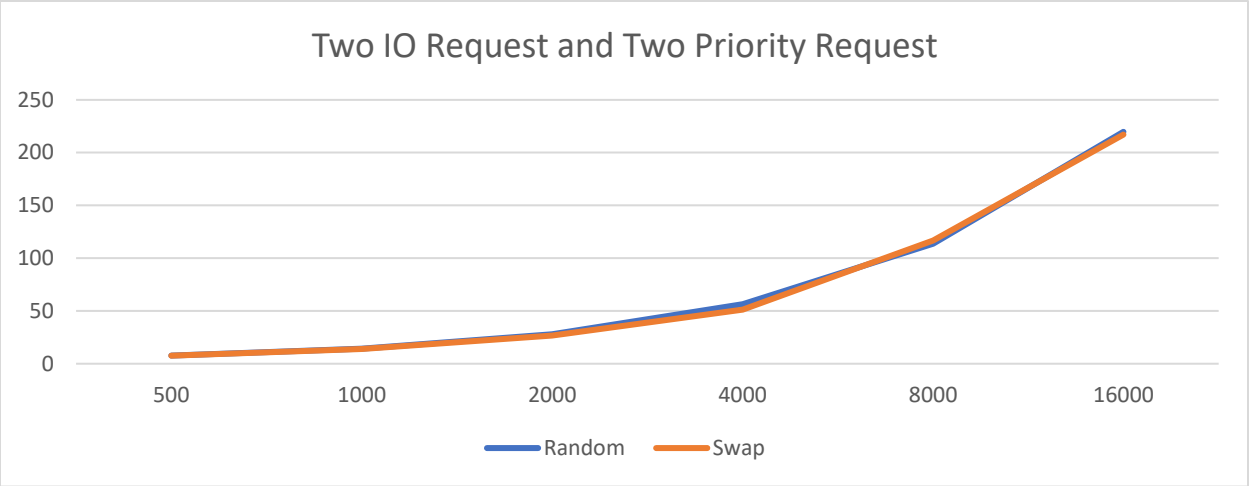
Appendix 4:



Two IO Requests and No Priority

(Milliseconds)	500	1000	2000	4000	8000	16000
Random	6.98	14.199	28.919	56.667	106.543	214.123
Swap	6.557	14.389	27.342	51.29	109.213	207.977

Appendix 5:



Two IO Request and Two Priority requests

(Milliseconds)	500	1000	2000	4000	8000	16000
Random	7.707	14.325	27.922	56.5	113.78	219.718
Swap	7.627	13.897	26.7301	51.255	116.787	217.119

Bibliography:

1. K. Hausknecht, D. Foit and J. Burić, "RAM data significance in digital forensics," 2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015, pp. 1372-1375, doi: 10.1109/MIPRO.2015.7160488.
2. F. Alted, "Why Modern CPUs Are Starving and What Can Be Done about It," in Computing in Science & Engineering, vol. 12, no. 2, pp. 68-71, March-April 2010, doi: 10.1109/MCSE.2010.51.
3. Z. Gutterman, B. Pinkas and T. Reinman, "Analysis of the Linux random number generator," 2006 IEEE Symposium on Security and Privacy (S&P'06), 2006, pp. 15 pp.-385, doi: 10.1109/SP.2006.5.
4. Arpaci-Dusseau, Remzi H., and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2018.
5. S. Yoo, G. Nicolescu, L. Gauthier and A. A. Jerraya, "Automatic generation of fast timed simulation models for operating systems in SoC design," Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition, 2002, pp. 620-627, doi: 10.1109/DATE.2002.998365.