CSCI 5103 Project 3 Writeup

Anlei Chen & Shuo Jiang

1. Setup:

The purpose of the experiments and experimental setup is to gain a better understanding of virtual memory and how demand paging works. Demand paging is much more complicated than just swapping out a page; a lot of thought and consideration must be given to replacement policies in order to minimize page faults and increase system performance. The experimental setup provides a simple environment that allows us to implement various parts of virtual memory, and, more importantly, allows us to test out policies and better understand how they work.

A Makefile is provided with a few helpful targets: `debug`, `run-all`, and `output`. The `debug` target will rebuild the project with debug statements enabled. The `run-all` target will run all combinations of policies and programs with a set page and frame size. The page and frame sizes can be adjusted using `PAGES=N FRAMES=M` or manually adjusted in the Makefile. The `output` target will create an output file (default is `out.txt`) with the experiment results. The experiment essentially runs `run-all` with a set page size and a list of frame sizes.

To clarify, make output will run `./virtmem 100 <frames> <policy> <program>` with every combination of the following frames: 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100; policies: rand, fifo, lifo, rd_only, wr_only, mrw, lrw; and programs: scan, sort, and focus. All testing was done on the CSE Lab Machines, specifically **csel-kh1250-21**. Note: lifo and rd_only are not run with sort, and clock is not run at all, as it has the same behavior as fifo.

2. Policies:

rand:
The random policy selects a page at random from the set of pages that are currently mapped to physical frames.

fifo:
The first-in, first-out policy chooses the chronologically first page that was mapped to a physical frame. This is implemented by maintaining a queue* of actively mapped pages and popping the first element.

lifo:
The last-in, first-out policy chooses the most chronologically recent page that was mapped to a physical frame. This is implemented by maintaining a stack* of actively mapped pages and popping the top element. This means that on every page fault, the same physical frame will always be chosen as the replacement frame.

rd_rand:
The read-random policy follows a similar implementation to the random policy but prioritizes read-only pages over dirty (modified) pages. The rationale behind this policy is that dirty pages could potentially still be in use by the process and require expensive disk write-back operations. By prioritizing the read-only pages, the policy may reduce the number of writes and page faults which would improve performance. This policy is implemented by maintaining a list of dirty pages in addition to the list of mapped pages. If every page is dirty, then a random page will be selected; otherwise, only read-only pages will be selected.

wr_rand:
The write-random policy follows a similar implementation to the read-random policy but prioritizes dirty (modified) pages over read-only pages. The rationale behind this policy is that dirty pages could signify that the page is done being used by the process and can be safely written back to disk. By prioritizing dirty pages, the policy may reduce the number of page faults by freeing up pages that are no longer in use. This policy is implemented by maintaining a list of dirty pages in addition to the list of mapped pages. If there are dirty pages, a random dirty page will be selected; otherwise, a random page will be chosen.

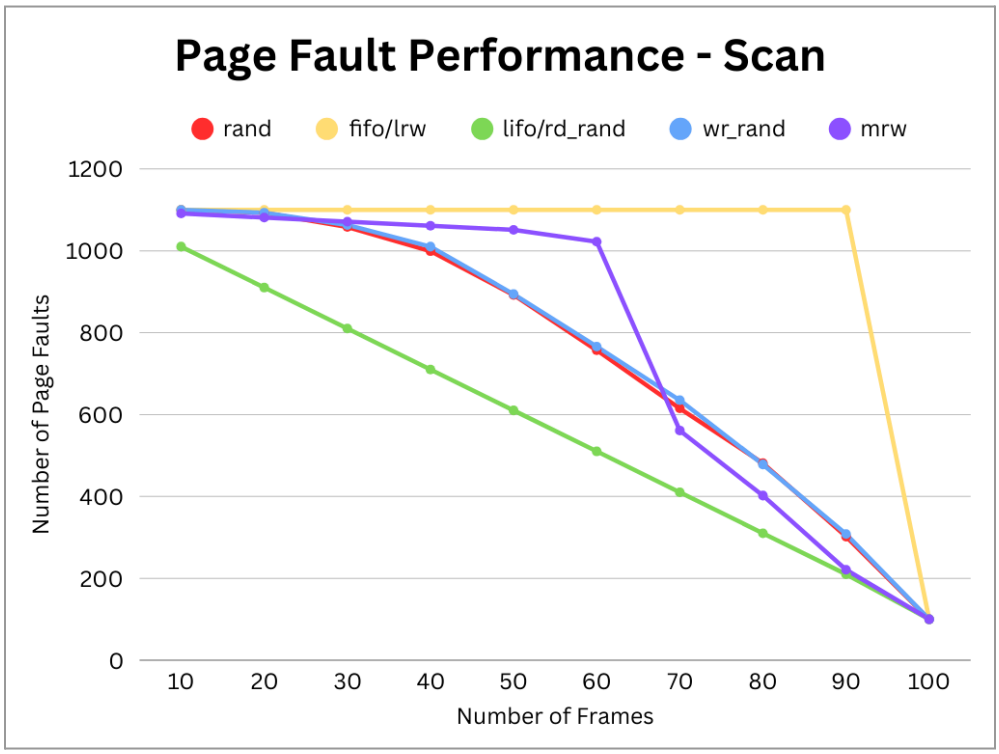
mrw:
The most-recently-written policy chooses the most recent page that was switched from read-only protection to read-write protection. The rationale behind this policy is that many page writes may be one-off events, after which the process no longer uses the page. By prioritizing the most recently written page, the policy replaces pages that are likely to be one-off, allowing pages still in use to remain in the page table. This policy is implemented by maintaining a stack* of dirty pages and popping the top element.

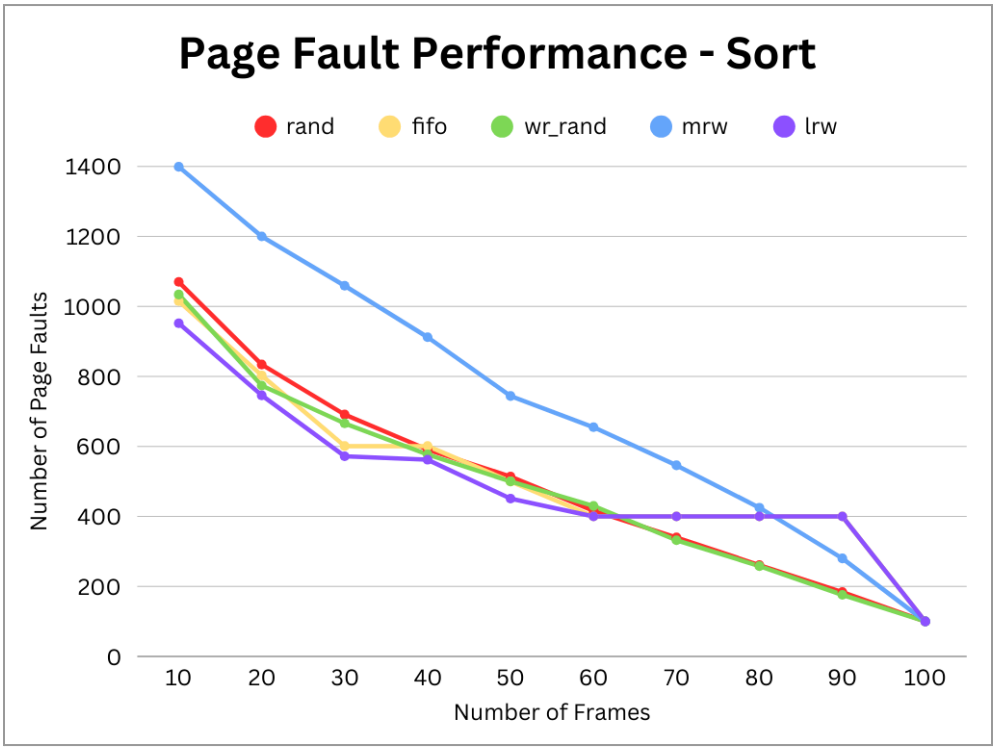
lrw:
The least-recently-written policy chooses the least recent page that was switched from read-only protection to read-write protection. The rationale behind this policy is that older pages that were written to are less likely to be used again. By prioritizing the least recently written page, the process can replace an older dirty page that is likely to be finished being used, reducing page faults. The idea is based on the LRU policy, which replaces the least recently accessed page. Since we don't track usage, the next best option is to track writes. This policy is implemented by maintaining a queue* of dirty pages and popping the first element.

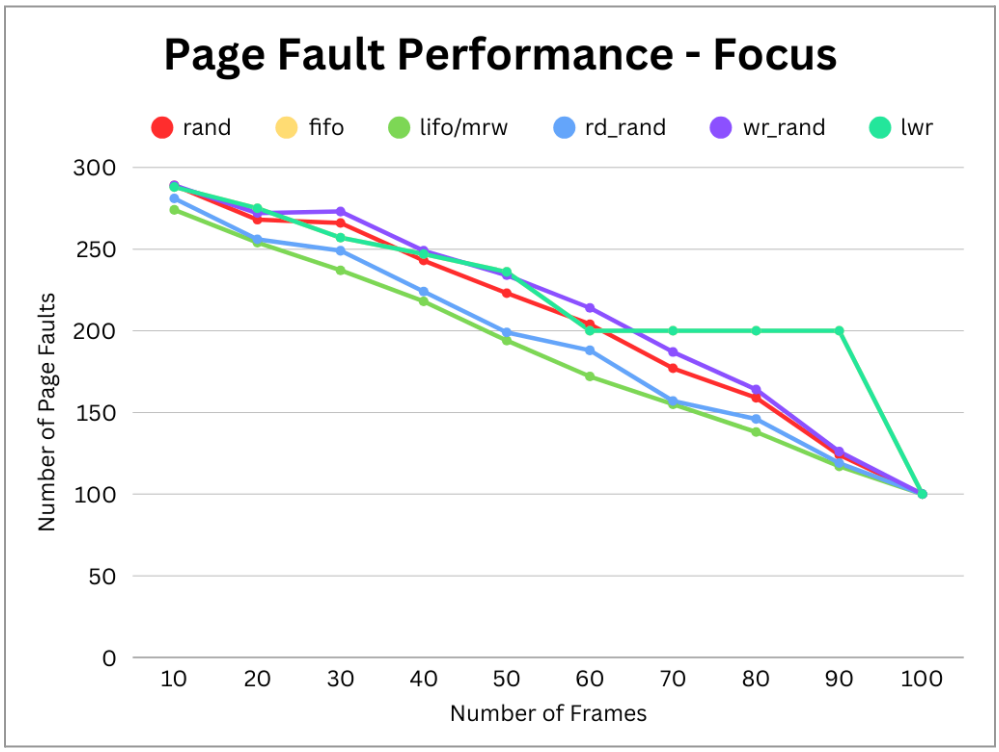
clock:
The clock policy maintains a circular list of mapped pages and a clock hand (starting at index 0) that points to the current page under consideration. When a page fault occurs, the policy checks the page pointed to by the clock hand. If the use bit of the corresponding frame is unset, the frame is replaced with the new page. If the use bit is set, the policy clears the bit and advances the clock hand by one. This process repeats until a frame with an unset use bit is found. However, due to the limitations of the current page setup, page accesses are not tracked, so use bits can only be updated during page faults or the switch from read-only to read-write. As a result, the behavior and performance of this policy are the same as fifo.

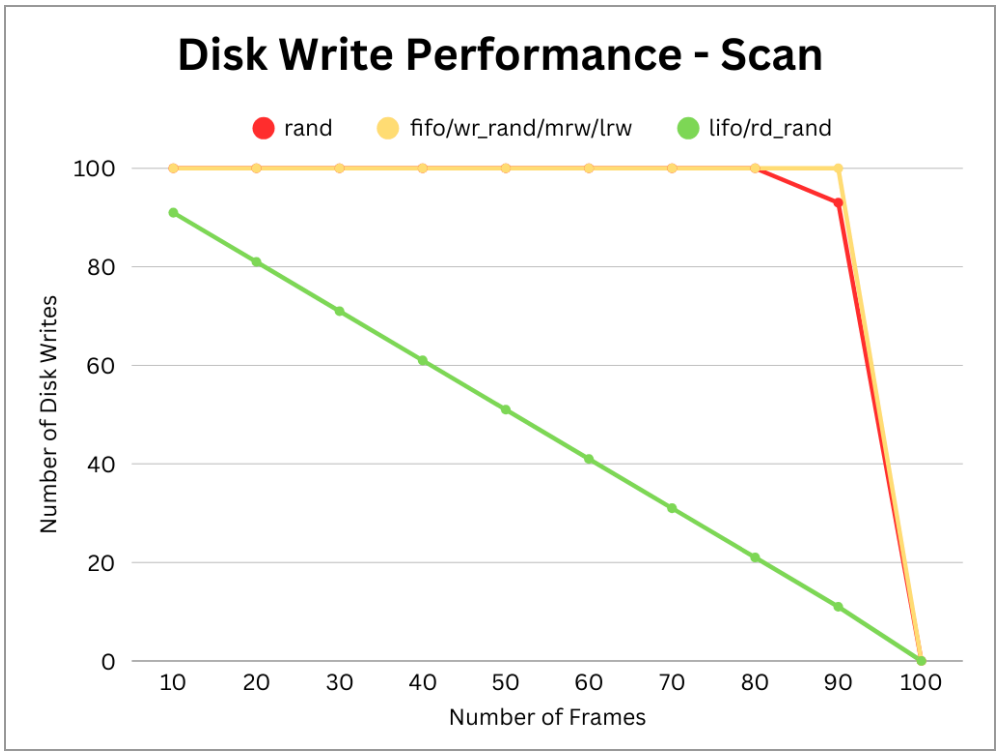
*A double-ended queue was used to implement both the stack and queue to simplify code.

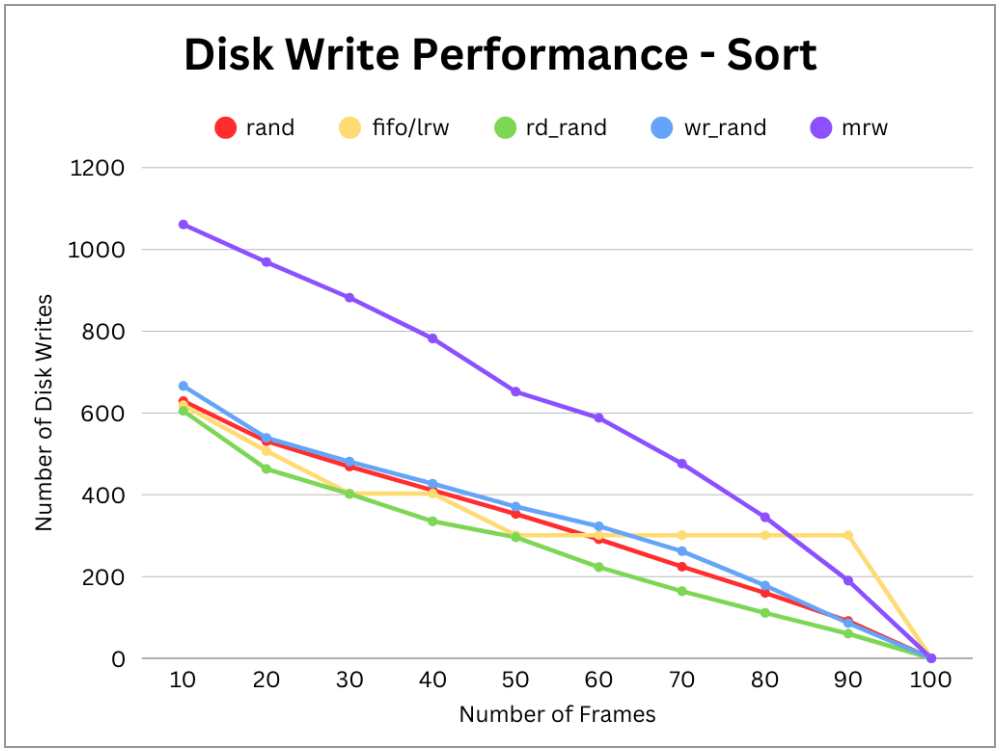
3. Graphs:

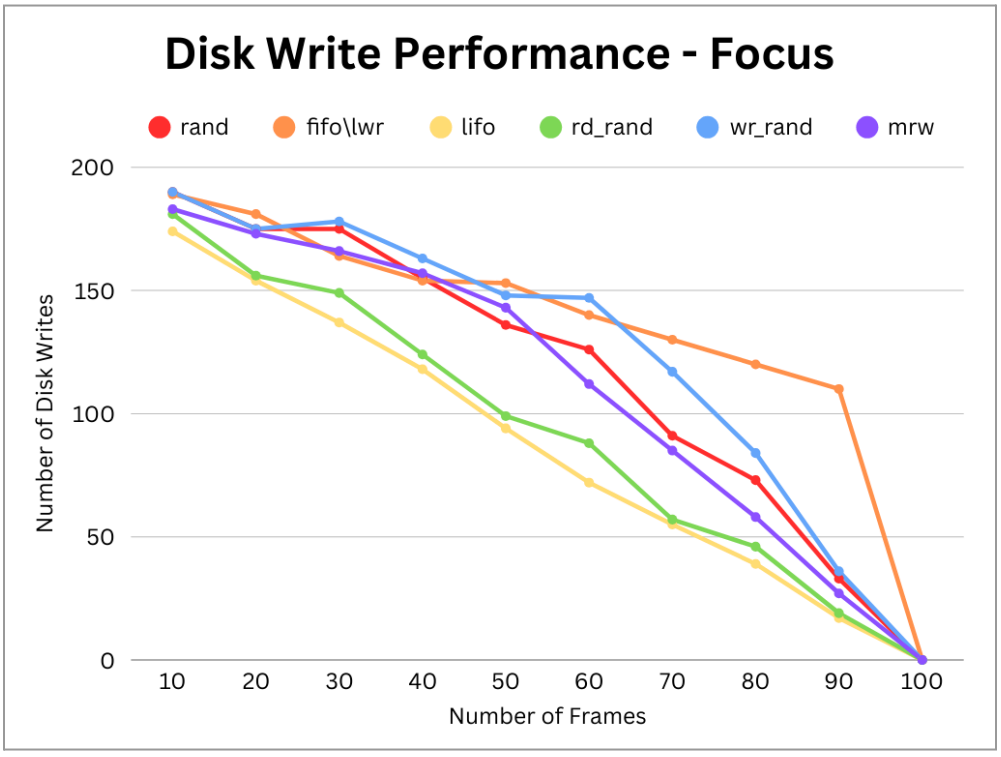

Page Fault Performance - Scan
rand
fifo/lrw
lifo/rd_rand
wr_rand
mrw
Number of Page Faults
0
200
400
600
800
1000
1200
10
20
30
40
50
60
70
80
90
100
Number of Frames


Page Fault Performance - Sort
rand
fifo
wr_rand
mrw
lrw
Number of Page Faults
0
200
400
600
800
1000
1200
1400
10
20
30
40
50
60
70
80
90
100
Number of Frames


Page Fault Performance - Focus
rand
fifo
lifo/mrw
rd_rand
wr_rand
lwr
300
250
200
150
100
50
0
Number of Page Faults
10
20
30
40
50
60
70
80
90
100
Number of Frames


Disk Write Performance - Scan
rand
fifo/wr_rand/mrw/lrw
lifo/rd_rand
100
80
60
40
20
0
Number of Disk Writes
10
20
30
40
50
60
70
80
90
100
Number of Frames


Disk Write Performance - Sort
rand
fifo/lrw
rd_rand
wr_rand
mrw
1200
1000
800
600
400
200
0
Number of Disk Writes
10
20
30
40
50
60
70
80
90
100
Number of Frames


Disk Write Performance - Focus
rand
fifo\lwr
lifo
rd_rand
wr_rand
mrw
200
150
100
50
0
Number of Disk Writes
10
20
30
40
50
60
70
80
90
100
Number of Frames

4. Analysis:

scan:

The scan program loops through all elements of an array in sequential order. Taking a closer look, we see that it makes one complete pass to write elements to the array, followed by another pass to read the elements. This type of linear access strongly benefits both the lifo and the rd_rand policies, because once a page is no longer needed, it will not be accessed again until the next cycle which occurs much later in time.

Looking more closely at the behavior of the rd_rand policy, it randomly selects dirty pages for replacement until the write loop finishes. Once the read loop begins, one random dirty page is replaced. Then, every subsequent page fault will replace a read-only page. This allows many write-only frames to remain mapped, which helps improve performance. Similarly, the lifo policy replaces the most recent page with the new page. It continues replacing the most recent dirty page until the read loop begins, at which point the read-only pages are always replaced, leaving the remaining dirty pages mapped.

Policies such as fifo and lrw will suffer significantly due to the sequential flow of the program. Both policies replace the oldest accessed page, and since there are more virtual pages than physical pages, page faults will always occur. The other policies strike a balance between the two cases described above, leading to mediocre performance.

sort:

The sort program takes an array and performs an in-place sorting algorithm on it. Since the exact implementation of qsort() is unknown, it is difficult to fully explain the behavior of the replacement policies. However, we know that the nature of in-place sorting means there are strong spatial and temporal relationships between the elements.

Most replacement policies have similar performance, with LRU showing slightly better performance for smaller physical memory sizes, but plateauing as the physical memory size increases. On the other hand, the mrw policy is much worse compared to the other algorithms, suggesting that there is some form of temporal locality related to writes.

Due to the structure of the sort program, the rd_rand policy takes an incredibly long amount of time, preventing us from reasonably benchmarking the program. There are many orders of magnitude more page faults compared to the other policies (can be verified using `run-all`). This suggests that, in addition to temporal write locality, there is also significant read locality. Furthermore, the lifo policy cannot be completed. We believe this is due to the last frame in the page table being repeatedly replaced by the policy before it can perform any write.

focus:
The focus program repeatedly writes to a small, random subset of memory, simulating a temporal locality pattern. The rd_rand policy prioritizes evicting read-only pages, making it well-suited for tasks like focus. It keeps all the dirty pages in the physical memory, as they tend to be reused frequently, while removing the less frequently used read-only pages.

The mrw and lifo policies have the fewest page faults among all the policies. Both policies reduce the number of page faults by retaining previously written pages in memory for reuse and replacing only the most recently written page, which is less likely to be used in the near future. However, these policies perform slightly worse in terms of disk writes, as each replacement of the most recently written page triggers a write-back to disk.

The wr_rand policy evicts dirty pages, which does not work well with the access pattern of the focus program. The policy randomly replaces a dirty page when the page table is full, leading to write-intensive pages being evicted. Similar to mrw, each eviction of a dirty page triggers a write-back to disk.

In theory, the clock algorithm should perform well in the focus program because the program issues random writes to a small window of memory, resulting in frequent re-access of the same page. Each re-access resets the use bit to 1, allowing the clock algorithm to skip over the active pages. However, due to limitations in the environment, we cannot actually test the policy.