Lab 2 Report
CSCI 5103
11/12/2017

Jon Huhn(huhnx025)
Paul Rheinberger(rhein055)
Steven Storla(storl060)

1 Introduction

In this project, we implemented a fully functional demand paging virtual memory system using random, first-in-first-out, and round robin page replacement algorithms. The purpose of this experiment was to evaluate the performance of each algorithm with regards to I/O operations across several benchmark programs. The three benchmark programs, scan, sort, and focus, used different memory access patterns to simulate a variety of programs.

When in the project's main directory, the program can be built using the `make` command. The `virtmem` program can then be invoked as follows:

```
./virtmem npages nframes rand|fifo|custom scan|sort|focus,
```

where `npages` is the number of pages and `nframes` is the number of frames. An example of a test invocation with 150 pages, 75 frames, using the fifo replacement algorithm and the focus program is as follows:

```
./virtmem 150 75 fifo focus
```

We wrote a script to run multiple iterations of our implementation with different parameters, including each combination of program and replacement algorithm with 100 virtual pages and between 10 and 100 physical frames at increments of 10 frames. The script outputs CSV data we used to make our charts in the Appendix. Our experiments were run on `csel-vole3d-02` and various machines in Keller 4-250.

2 Custom Page Replacement Algorithm

The custom page replacement algorithm that we implemented had a round robin policy. That is, the next page to be evicted cycles through all of the pages in the system. It is important to note that a page is evicted only if it is currently mapped to a frame. To be more explicit, the first time a page needs to be replaced, the algorithm chooses the first page only if it is mapped to a frame, otherwise it chooses the second page only if it is mapped to a frame, and so on until a valid page is found. The next time a page needs to be replaced, the algorithm picks up where it left off.

The algorithm was implemented using a global variable corresponding to the next page to be tested for valid eviction. When the `next_page_custom()` function is called, the next page global variable is tested to see if it is currently mapped to a frame, if it is, then the function increments the next page global variable and returns the valid page number to be evicted. If the page does not map to a frame, the global next page variable is repeated incremented, modulus the number of pages, until a valid page is found.

3 Analysis

As would be expected, the performance of the three page replacement algorithms depends on the pattern of memory accesses. The random replacement algorithm typically had better results during the Focus and Scan benchmark programs, as can be seen in Figure 1 and Figure 2 in the Appendix. The custom and FIFO replacement algorithms had better performance on the Sort benchmark program when the number of frames was less than 60, as can be seen in Figure 3. When the number of frames was greater than 60, random had better results.

The random replacement algorithm tended to have the same performance across all three benchmark programs. One explanation for this trend is that since the pages to be replaced are chosen at random, the chances that a good page or a bad page, relating to the ideal replacement selection, are about the same no matter what the memory access pattern is. Due to this fact, the performance of the random page replacement algorithms has little variation across the different benchmark test programs.

Now, let us consider the FIFO page replacement algorithm. The biggest downside to FIFO is that the first page in will always be the first out, even if that page will be used in the near future, when others will not be. This is why FIFO performs better during the Sort benchmark program. Data is accessed only when that particular chunk of data is being sorted, since the quicksort algorithm is used. After the chunk of data has been sorted, a different chunk of data is sorted. FIFO performs well at this workload because once the data is used, the program moves on to other data, so it may replace the page storing the previous data without any major performance penalties. However, when the number of frames approaches the number of pages, this advantage becomes less significant because nearly all of the pages can be present in physical memory. There are fewer page faults since most pages map to frames, so there is less of a performance bonus for replacing old pages. This is one reason why the number of page faults for FIFO tends to plateau as he number of frames approaches the number of pages.

The custom round robin page replacement algorithm behaves similarly to FIFO. When pages are accessed in sequential order, the round robin policy behaves the same as first in first out. In these benchmark programs, the data is accessed in a sequential manner inside a for loop. The Scan program randomly selects indices into the data. Since the memory accesses in these benchmark programs is close to sequential, these two replacement algorithms have similar performances, as would be expected.

To take advantage of the fact that these programs exhibit spatial and temporal locality, the least frequently used or least recently used page replacement algorithms could be implemented. Using either of these replacement policies would likely reduce the number of page faults for each of the three benchmark programs. However, these replacement policies are much more complex and harder to implement since they require hardware support.

4 Appendix

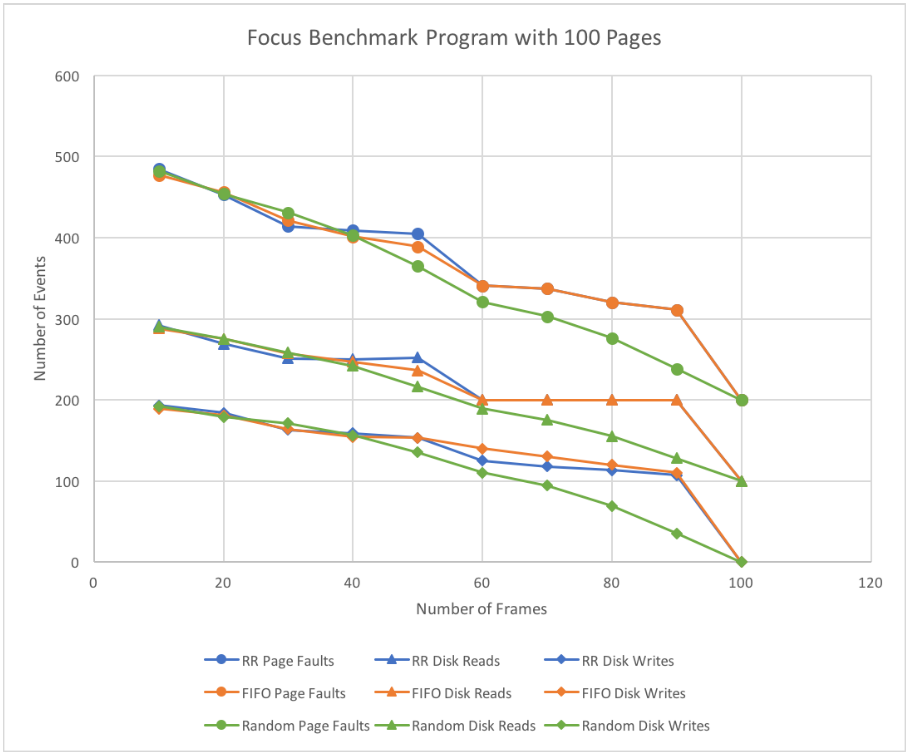

Figure 1. Focus Benchmark

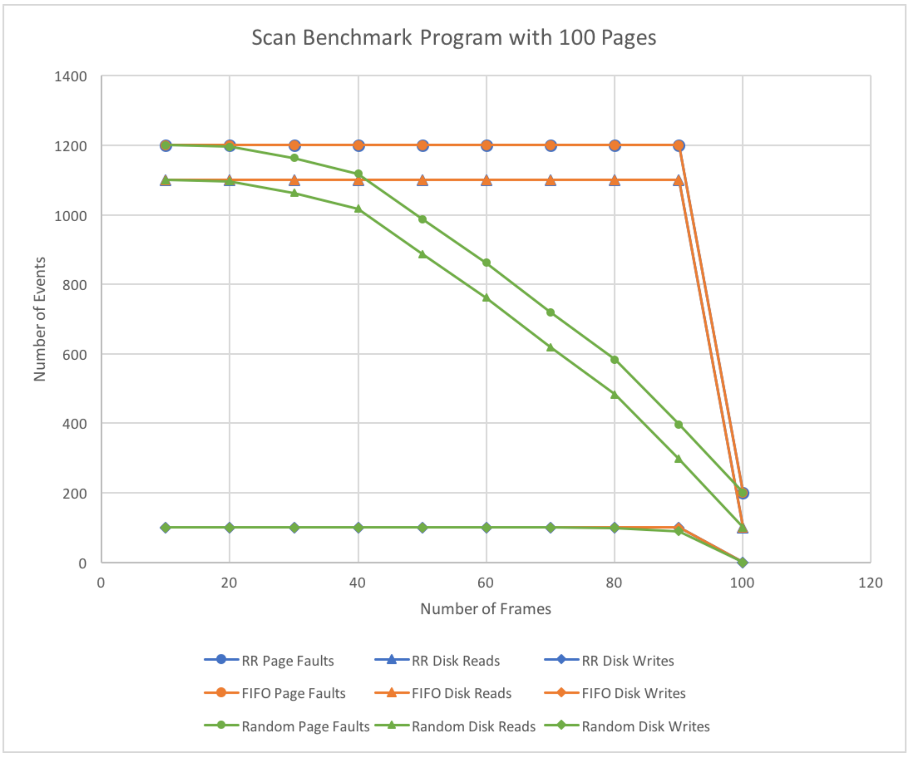

Figure 2. Scan Benchmark

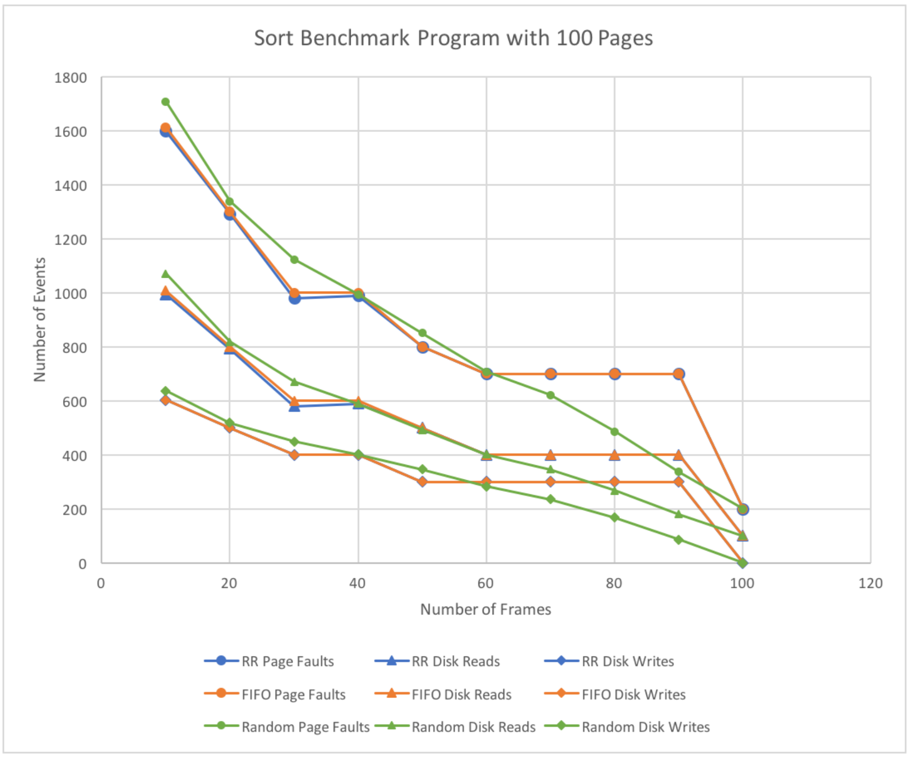

Figure 3. Sort Benchmark