**Introduction**

In our project we evaluated how real applications respond to 3 different page replacement algorithms fifo (first in first out), lru (least recently used), vms (segmented fifo). Fifo operates by removing the element that was loaded first. When adding entries to fifo they are pushed to the back. To illustrate an example, consider a line of students waiting to cross a stage for graduation. LRU operates by updating an element to the front if a matching address is found. If LRU is full, it removes the least recently used element which would be located in the back. Segmented fifo (vms) is implemented by using a FIFO, primary buffer and a LRU, secondary buffer. When the primary buffer becomes full it removes the first element and inserts it into the secondary buffer. If both buffers are full then the oldest element in the secondary is removed, again the first element in the primary is inserted into the secondary, finally the new element is pushed to the back of the primary.

From these algorithms we wrote a memory simulator, in our desired language, and evaluated memory performance using two trace files. These trace files contained multiple entries, each of which were a 32 bit hex address and a read or write character.

The memory is simulated when the operating system implements a page replacement policy. This policy will decide when to evict a page and how a page is inserted into the cache/caches. A variety of algorithms are used so that different performances can be measured. We were able to compare the three algorithms above for two trace files which were tested using a number of cases.

**Methods**

We were able to compare the algorithms using trace files given to us named bzip.trace and sixpack.trace. The algorithms FIFO, LRU, VMS were implemented testing all cases with different frame numbers between the two trace files. While testing we compared results to the samples that the professor and TA’s provided. Each algorithm was given the same tests between the files to compare the results. The results of our tests are shown below using charts. These charts compared the number of frames and how this affected the amount of reads and writes that would have happened in memory.

**Results**

| **bzip.trace** | **Reads** | **1000000 Events** |  |  |
| --- | --- | --- | --- | --- |
| **Number of Frames** | **FIFO** | **LRU** | **VMS (30)** | **VMS (60)** |
| 4 | 128601 | 92770 | 96509 | 92954 |
| 8 | 47828 | 30691 | 44006 | 33997 |
| 16 | 3820 | 3344 | 3445 | 3394 |
| 32 | 2497 | 2133 | 2322 | 2210 |
| 64 | 1467 | 1264 | 1356 | 1274 |
| 128 | 891 | 771 | 785 | 776 |
| 256 | 511 | 397 | 426 | 402 |
| 512 | 317 | 317 | 317 | 317 |

| **bzip.trace** | **Writes** | **1000000 Events** |  |  |
| --- | --- | --- | --- | --- |
| **Number of Frames** | **FIFO** | **LRU** | **VMS (30)** | **VMS (60)** |
| 4 | 38644 | 35650 | 35951 | 35791 |
| 8 | 18797 | 11092 | 15391 | 14288 |
| 16 | 1335 | 1069 | 1124 | 1093 |
| 32 | 851 | 702 | 768 | 730 |
| 64 | 514 | 420 | 460 | 420 |
| 128 | 305 | 224 | 227 | 224 |
| 256 | 125 | 48 | 61 | 51 |
| 512 | 0 | 0 | 0 | 0 |

| **sixpack.trace** | **Reads** | **1000000 Events** |  |  |
| --- | --- | --- | --- | --- |
| **Number of Frames** | **FIFO** | **LRU** | **VMS (30)** | **VMS (60)** |
| 4 | 351810 | 282320 | 306866 | 288955 |
| 8 | 230168 | 176496 | 189222 | 178933 |
| 16 | 140083 | 108682 | 114938 | 109479 |
| 32 | 85283 | 67747 | 70663 | 68288 |
| 64 | 48301 | 41186 | 42042 | 41635 |
| 128 | 27778 | 21090 | 23163 | 21685 |
| 256 | 15440 | 11240 | 11798 | 11371 |
| 512 | 8089 | 5823 | 6095 | 5855 |
| 1024 | 5492 | 4484 | 4540 | 4472 |
| 2048 | 4314 | 3951 | 3981 | 3959 |

| **sixpack.trace** | **Writes** | **1000000 Events** |  |  |
| --- | --- | --- | --- | --- |
| **Number of Frames** | **FIFO** | **LRU** | **VMS (30)** | **VMS (60)** |
| 4 | 94710 | 71269 | 79965 | 72077 |
| 8 | 57121 | 32717 | 39274 | 34057 |
| 16 | 31314 | 19342 | 21120 | 19464 |
| 32 | 18805 | 13730 | 14481 | 13951 |
| 64 | 11936 | 9672 | 10242 | 9759 |
| 128 | 8346 | 6526 | 7157 | 6761 |
| 256 | 5426 | 4092 | 4317 | 4145 |
| 512 | 3353 | 2444 | 2564 | 2456 |
| 1024 | 2368 | 1846 | 1893 | 1861 |
| 2048 | 1581 | 1356 | 1381 | 1378 |

The data provided above are a handful of test cases that we were able to test the algorithms given the trace files. We decided to use multiple powers of 2 as it provides a large range of overview for the outcome. Though we excluded 1 and 2 this was done on purpose, we affirmed that attempting to traverse through the algorithms with such a low number of frame numbers would not be ideal. The outcome would be valid, but would consist of little variation between the algorithms. As compared between the two trace files we see that the bzip file loads a smaller number of different pages than sixpack does. This shows that the memory requirements for bzip are around 2MB and that for sixpack can increase from 16MB. We also can see that LRU performs the best out of the others in almost all cases, FIFO performs the worst, and VMS (segmented fifo) is the middle case. LRU performs best as there are a number of pages that are repeatedly used during events. By keeping the most recently used pages to the front and deleting the least used allows us to decrease the number of misses. Whereas FIFO would have an increased number of misses due to accessing pages earlier than other operations. Finally VMS ( segmented fifo) operates closely to LRU the closer we get to 100% and vice versa for fifo as we approach 0%. This is also seen within our data when comparing 30% and 60% for VMS, as the 60% VMS outperforms the other. Overall we find segmented fifo being the best intermediate choice as it reduces the number of faults.

**Conclusion**

The results show that using the segmented fifo with percentages between thirty and forty provide a performance that is similar to lru. Given a higher percentage the performance becomes closer to lru and a lower percent it becomes closer to fifo.

Total time working on Project: 20+ hours

Eric Kemmer: 50%

Randall Hunt: 50%