Jordan Carr

Jcc117

CS1550

Project 3 Write Up

**NRU Refresh Options (See below for data and graphs)**

Gathered from the data presented in the graphs, the refresh option seems to depend on the number of frames. The relationship seems to be quadratic given the spread of the data points. My implementation has a counter based system where everything will be refreshed after a certain number of instructions are processed by the algorithm. The refresh options indicate how many instructions to wait until it is time to refresh. The graphs show the correlation between the number of page faults and the number of instructions to wait until refreshing. This implementation was used in order to rule out preemption, time between other processes running, and other factors which CPU time could randomly affect how the page selection process could happen on a single process. Given this based upon the results of running the tests on the gcc.trace file, the approximate estimates for each number of frames is given below to minimize the number of page faults:

8: 20

16: 60 to 80

32: 170-250

64: 520-740

These ranges also seem to hold for swim.trace.

To get consist tests for both files the following refresh options were used:

8: 20

16: 70

32: 200

64:520

For most of these tests the minimum number of page faults was never hit but it was fairly consistent across the data. However, in some cases as the number of page faults went down the number of writes went up. The costs of writing to disk seemed to be offset by increasing the number of page faults slightly. Therefore, the above refresh options were used for each test.

**Analysis of the Given Algorithms(See below for data)**

For obvious reasons the Optimal algorithm will only be used as a baseline for the minimum number of page faults achievable for page selection. It is noteworthy to mention that this does not necessarily correlate with the minimum number of writes to disk since this algorithm does not take this into account.

Asymptotically, the fastest algorithm to implement is the Random algorithm. With an O(1) runtime, it far out achieves that of Clock and NRU. However, the number of page faults and writes to disk can vary greatly with each run through and is on average much higher than that of the other two. Although it is easier to implement and faster overall, its unpredictable nature and overall inefficiency makes it not suitable to run.

NRU is able to handle writes to disk and page faults much better. However, it is the slowest of the algorithms, running on 3 O(n) loops to find a suitable page to replace. However, the likely hood of reaching the third loop is unlikely and will usually be found in the first or second loop. This algorithm can compete with Clock given the right program and refresh option, but optimal refresh option is not likely to hit the minimal page fault to write ratio for most processes. It will usually end up having more page faults and costing more time.

Clock in some cases is able to cut down on page faults more than NRU. However, it does not take into account whether a page is dirty or not. It’s writes to disk, given the trace files, seem to be higher than that of NRU. Overall, its runtime seems to be better than that of NRU, only utilizing one loop that will iterate at most 1 full time. The costs of writing to disk seem to be outweighed by this, however.

Overall, the best algorithm to implement out of this set is Clock. With a good refresh option NRU can minimize page faults and writes to disk as much as possible and can on average have a runtime that is minimized as much as possible. On the other hand, clock is more generalized and more adaptable to larger frame sizes and differing programs. With the wrong refresh options NRU can have many more page faults and writes to disk, which is likely to happen for a generalized algorithm with varying page sizes and page accesses.

\*The following charts are based off of results from gcc.trace

\*\*Tests done with the following refresh options: 8- 20; 16-70; 32- 200; 64-520

\*Tests done average over 3 run throughs