**Readme and Summary: Scroll down for report/data.**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* HEADER

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Authors: Cory Wixom

Unix Login: masc1454

Class: CS 570 Summer 2014

Assignment 2

Overall, after numerous tests (you’ll see them in the pages.txt file I have

in my a2 folder) I prefer the clock algorithm. I have not seen any cases

where it was "bad" unlike LRU which I found one string where no matter

how many frames you add it will not decrease the number of page faults.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* ASSIGNMENT REQUIREMENTS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

CS570 Summer 2014 Assignment 2

The program shall perform the following:

Upon startup, prompt the user for the number of frames in main memory

Read the sequence of pages accessed from the file "pages.txt"

Run each of the algorithms (OPT, LRU, and Clock) on the same input string from the step above

Printout the number of faults each algorithm generates to the terminal.

Perform an analysis of the page replacement algorithms:

Use the simulation and page replacement algorithms developed in steps 1 and 2 above.

Develop at least two different input data sets for analyzing the algorithms.

The sequences of pages in these two sets should be designed such that they demonstrate

the differences in the LRU vs Clock algorithms.

Run the input sets through the simulation, collect the data.

You should use at least two different memory systems (different numbers of frames) on each data set.

Perform an analysis of the data (include controls, variables, outcomes, and input characterizations).

Document your experiment and findings in your README file. -- Note, your README file shall contain the usual requirements for a README file (see

READMEformat.pdf file posted on Blackboard) and it shall also include a summary of your findings from your analysis.

The assignment is due 1800 (6:00 PM) on Thursday, 5 June 2014

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* COMPILE/EXECUTE INSTRUCTIONS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

make, or to manually g++ Page.cpp DataConverter.cpp FileHelper.cpp PageTable.cpp Program.cpp PageReplacementAlgorithm.cpp Fifo.cpp LRU.cpp Clock.cpp ClockNode.cpp Optimal.cpp SecondChanceFifo.cpp

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS DIAGRAMS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*\*

\* The main program. This will excute the

\* entire Assignment 2 application.

\*/

Program.cpp

/\*\*

\* A Page represents a single page in the page table.

\*/

class Page

Page (int id);

int GetID();

void SetID(int id);

std::string ToString();

/\*\*

\* A page table represents a list of pages currently

\* loaded into memory. It uses a dictionary lookup

\* based on a integer pageid.

\*/

class PageTable

PageTable();

Page Get(int pageID);

void Add(Page page);

void Remove(int pageID);

bool Exists(int pageID);

int Size();

std::vector<int> GetPages();

/\*\*

\* A PageReplacementAlgorithm is the base class

\* for each algorithm.

\*/

class PageReplacementAlgorithm

/\*\*

\* The constructor for an PageReplacementAlgorithm;

\*/

PageReplacementAlgorithm (std::vector<int> pages, int numberOfFrames);

/\*\*

\* Performs a memory get on the page table.

\*/

virtual void DoIt(int pageID);

/\*\*

\* The name of the algorithm for printing.

\*/

virtual std::string Name();

/\*\*

\* The current number of page faults.

\*/

int NumberOfFaults();

/\*\*

\* The current number of frames used.

\*/

int NumberOfFrames();

/\*\*

\* The optimal algorithm is a PageReplacementAlgorithm

\* that is theoretical and cannot be created in

\* a real environment.

\*/

class Optimal : public PageReplacementAlgorithm

/\*\*

\* The constructor for an Optimal;

\*/

Optimal (std::vector<int> pages, int numberOfFrames);

/\*\*

\* Performs a memory get on the page table.

\*/

virtual void DoIt(int pageID);

/\*\*

\* The name of the algorithm for printing.

\*/

virtual std::string Name();

/\*\*

\* The LRU PageReplacementAlgorithm handles

\* the logic of running a LRU page replacement

\* on the pages passed in.

\*/

class LRU : public PageReplacementAlgorithm

/\*\*

\* The constructor for an LRU;

\*/

LRU (std::vector<int> pages, int numberOfFrames);

/\*\*

\* Performs a memory get on the page table.

\*/

virtual void DoIt(int pageID);

/\*\*

\* The name of the algorithm for printing.

\*/

virtual std::string Name();

/\*\*

\* The Fifo PageReplacementAlgorithm handles

\* the logic of running a Fifo page replacement

\* on the pages passed in.

\*/

class Fifo : public PageReplacementAlgorithm

/\*\*

\* The constructor for an Fifo;

\*/

Fifo (std::vector<int> pages, int numberOfFrames);

/\*\*

\* Performs a memory get on the page table.

\*/

virtual void DoIt(int pageID);

/\*\*

\* The name of the algorithm for printing.

\*/

virtual std::string Name();

/\*\*

\* The SecondChanceFifo PageReplacementAlgorithm handles

\* the logic of running a Fifo page replacement

\* on the pages passed in, but it also gives

\* each page a second chance before being

\* evicted.

\*/

class SecondChanceFifo : public PageReplacementAlgorithm

/\*\*

\* The constructor for an SecondChanceFifo;

\*/

SecondChanceFifo (std::vector<int> pages, int numberOfFrames);

/\*\*

\* Performs a memory get on the page table.

\*/

virtual void DoIt(int pageID);

/\*\*

\* The name of the algorithm for printing.

\*/

virtual std::string Name();

/\*\*

\* A clock node is a node in a clock

\* paging algorithm.

\*/

class ClockNode

/\*\*

\* The constructor for an ClockNode;

\*/

ClockNode (int pageID);

int PageID();

void SetPageID( int pageID);

bool Flag();

void SetFlag(bool flag);

/\*\*

\* The Clock PageReplacementAlgorithm handles

\* the logic of running a clock page replacement

\* on the pages passed in.

\*/

class Clock : public PageReplacementAlgorithm

/\*\*

\* The constructor for an Clock;

\*/

Clock (std::vector<int> pages, int numberOfFrames);

/\*\*

\* Performs a memory get on the page table.

\*/

virtual void DoIt(int pageID);

/\*\*

\* The name of the algorithm for printing.

\*/

virtual std::string Name();

/\*\*

\* A static class that helps with file manipulations.

\*/

class FileHelper

/\*\*

\* Writes a single line in a file.

\*/

static bool WriteLine(std::string fileName, std::string text);

/\*\*

\* Checks to see if a file exists.

\*/

static bool Exists(std::string fileName);

/\*\*

\* Deletes a file from disk.

\*/

static void Delete(std::string fileName);

/\*\*

\* Reads the first line of the file.

\*/

static std::string PeakLine(std::string fileName);

/\*\*

\* The DataConverter is a helper utility

\* that will convert between different data types

\* and parse data into different formats.

\*/

/\* static \*/ class DataConverter

static bool ConvertToIntFromHexString(int& t,const std::string& s,std::ios\_base& (\*f)(std::ios\_base&));

static std::vector<int> SplitStringToInts(std::string str, char delimeter);

static std::string GetBinaryStringFromHexString (std::string hexString);

static std::string Rtrim(std::string s);

static std::string GetBinaryStringFromHexChar (char hexChar);

static int GetIntFromBinaryString (std::string binaryString);

static std::string GetNibble(int index, std::string instruction);

static std::string GetNibbles(int startIndex, int endIndex, std::string instruction);

static std::string GetIntToString(int number);

static int GetStringToInt(std::string str);

static std::string GetStringFromBool(bool flag);

static bool GetBoolFromChar(char flag);

static std::string ConvertIntToHex(int i);

**Report:**

Page replacement algorithms are used in order to minimize the number of page faults that occur when replacing a “lesser used” page with a page that is required currently. The different algorithms were developed with different strengths and considerations in mind.

* **Optimal**
  + This is a theoretical algorithm, and cannot be implemented on a real machine unless it can look into the future.
  + The Algorithm:
    - The page that is evicted from the page table that will not be used for the longest period of time. That means the way to know what page to evict is to know what future memory locations the program will access.
  + Pluses/Minuses:
    - This is the goal of what all the algorithms try to achieve.
    - The minus to it is it is impossible to achieve. There are certain things that can be done with the programs to “look ahead” and try to anticipate what memory locations the programs will access next but that is outside the scope of this analysis.
* **FIFO**
  + Fifo stands for first in/first out. The purpose of the algorithm assumes that a memory location that a program accesses have an even distribution, and all memory locations have an equal shot at being accessed again. Many types of algorithms in computer science, and in fact in life, follow a FIFO format. But, programs tend to cluster around a small number of memory locations, so what tends to happen with FIFO is it will evict a heavily used memory location for a lesser used because it assumes equality of all memory locations.
  + The Algorithm:
    - FIFO is one of the easier to implement algorithms.
    - Consider the data string “1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9” with 3 frames.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Pages | 1 | 2 | 1 | 3 | 1 | 4 | 1 | 5 | 1 | 6 | 1 | 7 | 1 | 8 | 1 | 9 |  |
|  | 1 | 2 | 2 | 3 | 3 | 4 | 1 | 5 | 5 | 6 | 6 | 7 | 1 | 8 | 8 | 9 |  |
|  |  | 1 | 1 | 2 | 2 | 3 | 4 | 1 | 1 | 5 | 5 | 6 | 7 | 1 | 1 | 8 |  |
|  |  |  |  | 1 | 1 | 2 | 3 | 4 | 4 | 1 | 1 | 5 | 6 | 7 | 7 | 1 |  |
| Page Fault? | 1 | 1 |  | 1 |  | 1 | 1 | 1 |  | 1 |  | 1 | 1 | 1 |  | 1 | 11 |

* + - For each page in the string, as the page is accessed if there is room in the page table, it will add it to the table. Conceptually, it would be stored as a queue and each page will be added to the queue from the back.
    - As more pages are added, eventually it will fill up the table. The algorithm specifies that the page that gets evicted is the one that arrived in the queue first. So, a simple dequeue from the top for the one to remove, and the new one is queued in from the back.
  + Pluses/Minuses:
    - Pluses
      * If the data is more evenly distributed, then FIFO can be a good approximation of the optimal, and it’s simple to implement.
      * Example: 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
      * With 3 frames that produces 72 page faults, whereas the optimal was only 56.
    - Minuses:
      * The above data stream provided 11 page faults with 3 frames, whereas the optimal is 9. So, with this simple test it does not appear to be that greatly different.
      * But, if the data string was expanded to 192 pages (just copy the string 12 times see “Large Bad String” below) that will expand the number of page. You start to see a big difference in the number of page faults. FIFO would provide 128 page faults and optimal would have 84. That is an increase of ~50% more than the optimal algorithm.
      * What is even more interesting is if you increase the number of frames to 8. FIFO will produce 108 page faults, whereas the optimal algorithm would only produce 21. Therein lies the failure of FIFO, that it will evict page id 1 from the page table even though it is called every other time.
      * Large Bad String:
        + 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9
* **Second Change Fifo**
  + Fifo stands for first in/first out. The purpose of the algorithm is to try to improve upon the regular Fifo algorithm by giving each page a “second chance”. Meaning if a page is accessed, it will set a flag on the page which will allow it not to be evicted until it has exhausted all other options.
  + The Algorithm:
    - Much like FIFO, this will use a queue and add pages to the queue as they come in. The only difference is once a page is accessed, it will mark the R bit on the page as true. This means that when it’s time to find a page to evict it can only evict pages with an R bit of 0. As it traverses the queue it sets the R bit to 0 so the next time around it will find something to evict.
  + Pluses/Minuses:
    - Pluses
      * This algorithm is designed to improve upon FIFO by keeping more heavily used pages in memory.
      * Comparing using some of the same strings it can improve upon the performance of FIFO in some situations.
      * Example: 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
        + Just like FIFO it produces 72 page faults, whereas the optimal was only 56.
      * Example: 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9
        + This one does improve over regular FIFO with 97 page faults instead of the 128 that FIFO does.
    - Minuses:
      * Even though this does improve upon FIFO. It does not prove to be better than Clock or LRU, therefore it is not a better algorithm.
      * Large Random String:
        + 1 4 3 6 2 1 7 8 8 2 1 9 7 1 3 1 4 4 5 6 7 8 9 2 1 5 2 3 3 33 3 3 2 5 5 5 5 7 2 2 2 6 7 8 9 4 3 2 3 4 7 7 7 7 7 7 7 4 99 89 01 34 2 5 4 3 6 453 6 3 4 6 7 3 32 23 4 6 67 8 8 23 34 345 4 4 4 43 6 6 7 3 7 32 346 345 1 4 3 6 2 1 7 8 8 2 1 9 7 1 3 1 4 4 5 6 7 8 9 2 1 5 2 3 3 33 3 3 2 5 5 5 5 7 2 2 2 6 7 8 9 4 3 2 3 4 7 7 7 7 7 7 7 4 99 89 01 34 2 5 4 3 6 453 6 3 4 6 7 3 32 23 4 6 67 8 8 23 34 345 4 4 4 43 6 6 7 3 7 32 346 345
* **LRU**
  + Because of some of the downfalls of FIFO, a variation on it was created that will help to eliminate its shortfalls. LRU stands for “least recently used”. That means that instead of just keeping into account FIFO, we also need to account for when a memory location was last accessed.
  + The Algorithm:
    - I’m implementing this algorithm in the same way that was discussed in class. I saw other ways to implement it, but this appeared to be the simplest. Much like FIFO the LRU algorithm we went over in class will add pages to a queue, and pop them from the front/back as the pages are accessed. The change is that when a page is accessed that already exists in the page table; it will be moved to the back of the queue so it will be evicted last.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Pages | 1 | 2 | 1 | 3 | 1 | 4 | 1 | 5 | 1 | 6 | 1 | 7 | 1 | 8 | 1 | 9 |  |
|  | 1 | 2 | 1 | 3 | 1 | 4 | 1 | 5 | 1 | 6 | 1 | 7 | 1 | 8 | 1 | 9 |  |
|  |  | 1 | 2 | 1 | 3 | 1 | 4 | 1 | 5 | 1 | 6 | 1 | 7 | 1 | 8 | 1 |  |
|  |  |  |  | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 7 | 7 | 8 |  |
| Page Fault? | 1 | 1 |  | 1 | 1 | 1 |  | 1 |  | 1 |  | 1 |  | 1 |  |  | 9 |

* + - So, comparing it to FIFO, we achieve a page fault count of 9 vs 11. 9 is also the optimal number of faults for this configuration.
    - As more pages are added and the table gets filled up, the pages that are accessed most recently are upgraded back to the bottom of the queue so they will not leave as early.
  + Pluses/Minuses:
    - Pluses:
      * LRU takes a similar approach to FIFO, but it adds one thing that helps to eliminate its downfall. Since it takes into account the last time a memory location was accessed the issue of something getting evicted when it should not is better. Note how in the diagram above, the 1 never gets evicted, but it did for FIFO.
      * LRU is considered an “excellent” algorithm by the book (page 215). Below is a random string that I created for testing. LRU takes 56 page faults, vs FIFOs 69 and the optimal 36.
      * Large Random String:
        + 1 4 3 6 2 1 7 8 8 2 1 9 7 1 3 1 4 4 5 6 7 8 9 2 1 5 2 3 3 33 3 3 2 5 5 5 5 7 2 2 2 6 7 8 9 4 3 2 3 4 7 7 7 7 7 7 7 4 99 89 01 34 2 5 4 3 6 453 6 3 4 6 7 3 32 23 4 6 67 8 8 23 34 345 4 4 4 43 6 6 7 3 7 32 346 345 1 4 3 6 2 1 7 8 8 2 1 9 7 1 3 1 4 4 5 6 7 8 9 2 1 5 2 3 3 33 3 3 2 5 5 5 5 7 2 2 2 6 7 8 9 4 3 2 3 4 7 7 7 7 7 7 7 4 99 89 01 34 2 5 4 3 6 453 6 3 4 6 7 3 32 23 4 6 67 8 8 23 34 345 4 4 4 43 6 6 7 3 7 32 346 345
    - Minuses
      * The book mentions that LRU is considered an excellent algorithm, but the downfall of it is that it requires a completely extra copy of the page table in memory in order to implement. This obviously adds quite an additional overhead.
      * Also, for the same large string as described in the FIFO section, LRU only incurs 97 page faults, vs the 84 of optimal.
      * What is interesting is if we increase the frame size to 8. We end up with the same, 97 page faults. The reason for this is that our string will alternate between 1 and some other memory location. So, for each hit of 1, nothing needs to be evicted. But, once the page table fills up, every other hit, something needs to be evicted, and it will be evicted in the FIFO order that they came in. Since, the string follows a consistent order coming it, the algorithm follows a consistent order coming out.
      * Even though LRU is considered a good algorithm, it is interesting that this string will always be 97 page faults for > 1 and < 9 frames. Even though this is because of the data I provided I’m considering it a negative to the algorithm. Just like FIFO, if there is a repeating alternating page ids, it can be poorly affected and increasing the number of frames does not help.
      * 97 faults string:
        + 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9
* **Clock**
  + The clock algorithm is considered an approximation to the LRU. Data is stored in a circle, and to find the page to evict you traverse the circle and find pages that have not been accessed recently.
  + The Algorithm:
    - Describing the clock algorithm is going to be difficult over text, but essentially what happens is you have a circular array, with a pointer to the current value. As data is accessed that exists, the pointer is incremented to the next “time” on the clock, and a bit is set indicating that the page was accessed. To determine the page to evict is to continue to increment the pointer (circle back to 0 if it reaches the end) until you find a page that has not been accessed recently. As it’s traversing the clock, the accessed bit is set to false, indicating it’s a candidate to be evicted the next time around.
    - In our standard data string, clock produces 10 page faults vs the optimal which is 10.
    - DataString:
      * 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9
  + Pluses/Minuses:
    - Pluses:
      * The clock algorithm is considered a very good approximation by the book.
      * If we look at our standard list of data strings:
      * FIFO large string:
        + With 3 frames, clock produces 115 page faults. Which is not so hot considering the LRU algorithm gets 97.
        + But if we increase to 8 frames, it drops down to 30 page faults; whereas optimal has 21 and LRU has 97 (remember this is the weird data set that LRU does the funky 97 with).
        + 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9
    - Minuses
      * It’s no question that Clock out performs FIFO. But how does it really hold up against LRU?
      * LRU random string:
        + Using the LRU string devised above for LRU we find that LRU does perform better.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Number of page faults | | | |
| Number of Frames | FIFO | Clock | LRU | Optimal |
| 3 | 134 | 135 | 128 | 101 |
| 5 | 116 | 112 | 112 | 71 |
| 8 | 78 | 80 | 78 | 43 |
| 10 | 69 | 55 | 56 | 36 |
| 15 | 47 | 44 | 39 | 27 |
| 20 | 42 | 23 | 29 | 22 |

* + - * + Judging by the data provided above, it’s hard to tell. Clock beats LRU sometimes, and LRU beats Clock other times.
        + 1 4 3 6 2 1 7 8 8 2 1 9 7 1 3 1 4 4 5 6 7 8 9 2 1 5 2 3 3 33 3 3 2 5 5 5 5 7 2 2 2 6 7 8 9 4 3 2 3 4 7 7 7 7 7 7 7 4 99 89 01 34 2 5 4 3 6 453 6 3 4 6 7 3 32 23 4 6 67 8 8 23 34 345 4 4 4 43 6 6 7 3 7 32 346 345 1 4 3 6 2 1 7 8 8 2 1 9 7 1 3 1 4 4 5 6 7 8 9 2 1 5 2 3 3 33 3 3 2 5 5 5 5 7 2 2 2 6 7 8 9 4 3 2 3 4 7 7 7 7 7 7 7 4 99 89 01 34 2 5 4 3 6 453 6 3 4 6 7 3 32 23 4 6 67 8 8 23 34 345 4 4 4 43 6 6 7 3 7 32 346 345
      * The thing that I disliked most about clock is it is more complicated. It’s harder to explain, and harder to demonstrate in this paper.

Overall, after numerous tests (you’ll see them in the pages.txt file I have in my a2 folder) I prefer the clock algorithm. I have not seen any cases where it was “bad” unlike LRU which I found one string where no matter how many frames you add it will not decrease the number of page faults.