**Project 2**

Abdulrahman Aldhubaiban
Tracy Jackson Jr.

Dr. Iamnitchi
March 4, 2019

## Introduction:

Page replacement is a mechanism to keep track of frequently used memory addresses. The essential problem with page replacement is that you have to keep track of each line of memory address of RAM into the frame. The page replacement algorithm must be used to know which page needs to be replacing in order of a new page to go into the frame. When a string address reads from the file, that means nothing will happen to the frame. The page that is being read in, will overwrites the page being evicted. Also, when a string address is being read that will also cause a page fault. When a page fault occurs, it has to choose a page to remove from memory to make room for the new page that has to be brought into memory. For the page to be removed, it has to be modified while in memory. For that to happen, the memory has to be written into the disk in order for frame to be updated. There are three-page replacement policies algorithms that must be implemented for the simulators to work. The first policy is FIFO, which is First In First Out. This simple policy will keeps track of all pages in the memory in a queue; the oldest page will be placed in front of the queue. When a page needs to be replaced, the page in the front of the queue will be designed to be remove. The second policy is LRU, which is Least Recently Used. This policy will replace the least-frequently used page when an eviction must take place. The LRU algorithm will remove the least recently used page when the frame is full, and a new page is put in from the least recent used page. The VMS policy is a second chance algorithm, but it still operates like FIFO, but it is more in depth when your keeping track of each page. There are two parts that we must track of is the clean bit and dirty bit. The clean bit will just read from the disk, which means that nothing is being modified and it will just be put into the frame. The dirty bit will just write from the disk, which means that the same string address is being modified.

## Methods:

Implemented a data structure of queue to make it more efficient to do all policies using the same data structure. Then this data has arrays of pages to make it as memory and there was struct for pages that contain a dirty bit and a page address which contain a page and memory contain arrays of pages.

### FIFO policy:

First in first out which using a queue is enough to fulfill this task. Because of the way we did our data structure, make this task easy to implement by using queue functions.

### LRU policy:

LRU stands for least recently used. When we have a set of memory addresses in a page and a new reference is made which is not present in the page, the new reference replaces the memory address in the page, which has been unused for the longest time.

### VMS policy:

We have two processes. First, if hexadecimal starts with 3 we perform process A, if hexadecimal doesn’t start with 3 we will process B. process A reference string is found in memory, then it shows that its there. Otherwise if we don’t find the bit in process A, then we check the clean array as the second chance for us to find the bit. When the FIFO is full, we must enqueue it in dirty or enqueue it in clean, so we must dequeue it from the full FIFO. Then we also enqueue and dequeue the string from clean and FIFO.

## Results:

When we reach to 4KB (4096), the values in read, write, hit, and miss stays constant, because we know that all pages and frames are 4KB. It shows the new page is being loaded into the memory disk and the page table will updated.

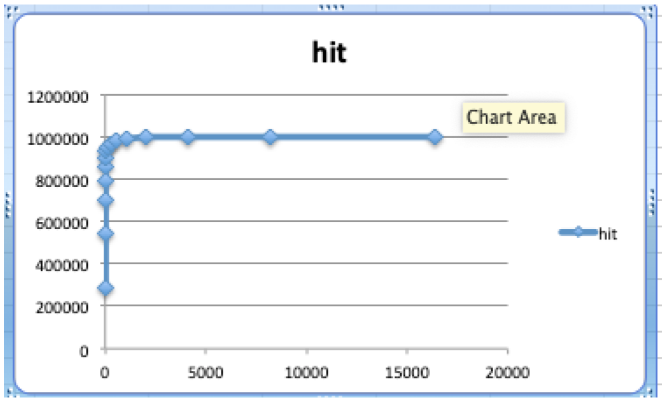
### FIFO policy

**FIFO Table:**

| nframes | Trace | Read | Write | Hit | Miss |
|---|---|---|---|---|---|
| **1** | 1000000 | 716106 | 104942 | 283894 | 716106 |
| **2** | 1000000 | 460912 | 76432 | 539088 | 460912 |
| **4** | 1000000 | 302860 | 55013 | 697140 | 302860 |
| **8** | 1000000 | 205368 | 37524 | 794632 | 205368 |
| **16** | 1000000 | 138539 | 24043 | 861461 | 138539 |
| **32** | 1000000 | 98067 | 16759 | 901933 | 98067 |
| **64** | 1000000 | 70315 | 12053 | 929685 | 70315 |
| **128** | 1000000 | 48526 | 8487 | 951474 | 48526 |
| **256** | 1000000 | 31698 | 5945 | 968302 | 31698 |
| **512** | 1000000 | 15609 | 3425 | 984391 | 15609 |
| **1024** | 1000000 | 6673 | 1747 | 993327 | 6673 |
| **2048** | 1000000 | 3432 | 700 | 996568 | 3432 |
| **4096** | 1000000 | 2852 | 0 | 997148 | 2852 |
| **8192** | 1000000 | 2852 | 0 | 997148 | 2852 |
| **16384** | 1000000 | 2852 | 0 | 997148 | 2852 |

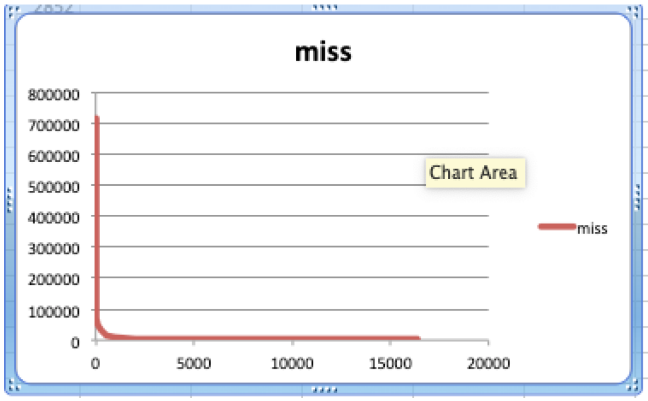
This table has all values for FIFO when used gcc.trace we used power of 2 for nframes

## Hit rate vs cache size:



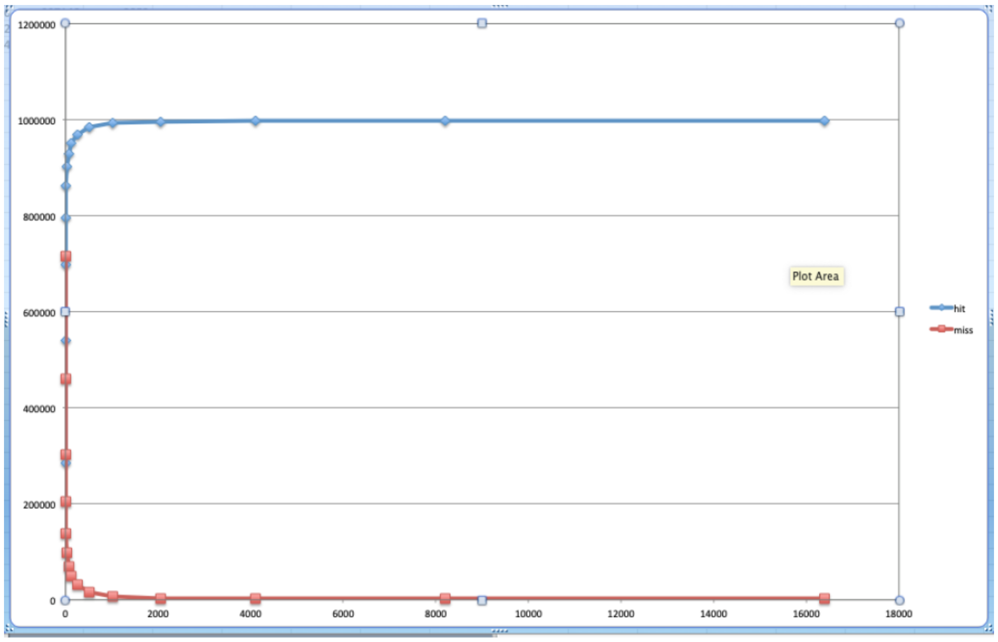
*This graph shows a hit rate for FIFO. It shows how the number of hits increases as the cache size is increased. The hit rate does not change much beyond a certain point. For example from the graph, after a cache size of about 4096 further increasing the cache size does not affect the hit rate.*

## Miss rate vs cache size:



*This graph shows a miss rate for FIFO. It shows how the number of misses decreases as the cache size is increased. The miss rate does not change much beyond a certain point. For example from the graph, after a cache size of about 4096, further increasing the cache size does not affect the miss rate.*

## Hit and miss rate vs cache size:



*This graph shows a miss and hit rate for FIFO*

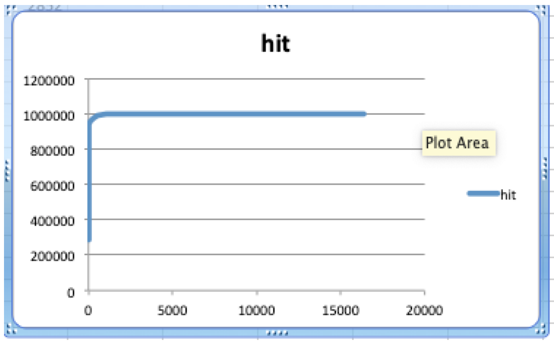
## <u>LRU policy</u>

**LRU Table:**

| nframes | Trace | Read | Write | Hit | Miss |
|---|---|---|---|---|---|
| **1** | 1000000 | 716106 | 104458 | 283894 | 716106 |
| **2** | 1000000 | 403955 | 66326 | 596045 | 403955 |
| **4** | 1000000 | 243809 | 33664 | 756191 | 243809 |
| **8** | 1000000 | 171186 | 19030 | 828814 | 171186 |
| **16** | 1000000 | 116604 | 11119 | 883396 | 116604 |
| **32** | 1000000 | 84401 | 8647 | 915599 | 84401 |
| **64** | 1000000 | 59089 | 6082 | 940911 | 59089 |
| **128** | 1000000 | 40821 | 4131 | 959179 | 40821 |
| **256** | 1000000 | 25308 | 2776 | 974692 | 25308 |
| **512** | 1000000 | 10425 | 1389 | 989575 | 10425 |
| **1024** | 1000000 | 4391 | 693 | 995609 | 4391 |
| **2048** | 1000000 | 2904 | 375 | 997096 | 2904 |
| **4096** | 1000000 | 2852 | 0 | 997148 | 2852 |
| **8192** | 1000000 | 2852 | 0 | 997148 | 2852 |
| **16384** | 1000000 | 2852 | 0 | 997148 | 2852 |

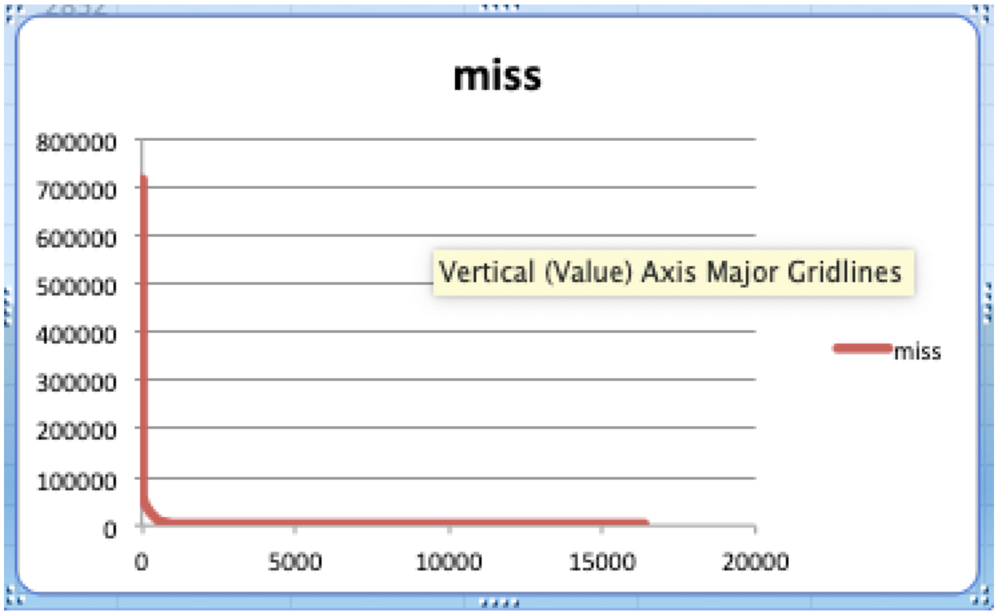
This table has all values for LRU when used gcc.trace we used power of 2 for nframes

### Hit rate vs cache size:



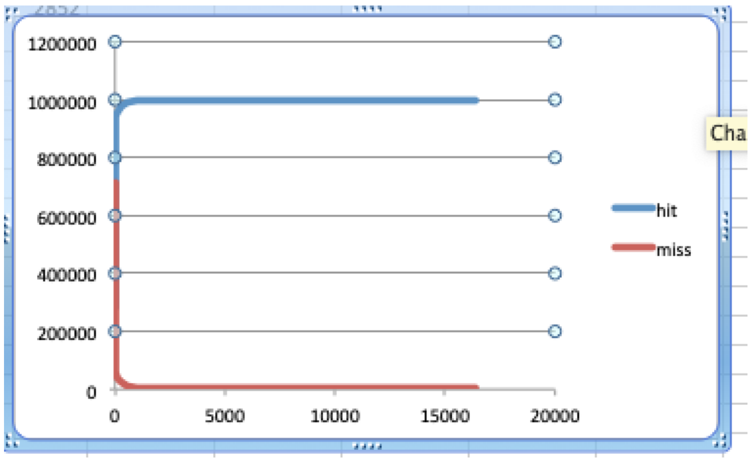
*This graph shows a hit rate for LRU. It shows how the number of hits increases as the cache size is increased. The hit rate does not change much beyond a certain point because of physical memory. For example from the graph, after a cache size of about 4096 further increasing the cache size does not affect the hit rate.*

## Miss rate vs cache size:



*This graph shows a miss rate for LRU. It shows how the number of misses decreases as the cache size is increased. The miss rate does not change much beyond a certain point. For example from the graph, after a cache size of about 4096, further increasing the cache size does not affect the miss rate.*

## Hit and miss rate vs cache size:



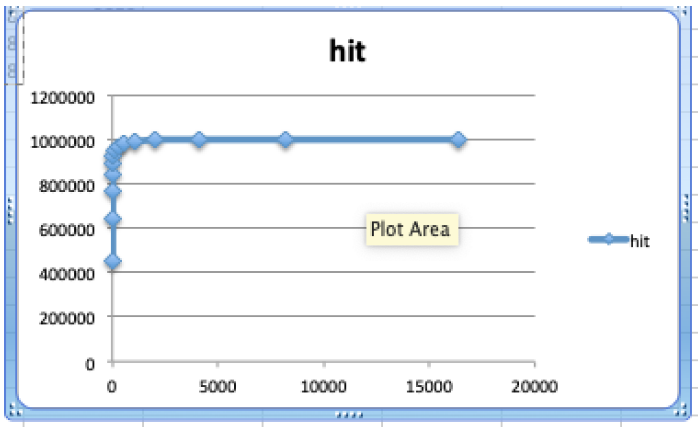
## VMS policy

## VMS Table:

| nframes | Trace | Read | Write | Hit | Miss |
|---|---|---|---|---|---|
| **1** | 1000000 | NA | NA | NA | NA |
| **2** | 1000000 | 545640 | 77389 | 454360 | 545640 |
| **4** | 1000000 | 356224 | 55899 | 643776 | 356224 |
| **8** | 1000000 | 233401 | 38636 | 766599 | 233401 |
| **16** | 1000000 | 155145 | 26324 | 844855 | 155145 |
| **32** | 1000000 | 106063 | 17956 | 893937 | 106063 |
| **64** | 1000000 | 75015 | 12709 | 924985 | 75015 |
| **128** | 1000000 | 50915 | 8837 | 949085 | 50915 |
| **256** | 1000000 | 31392 | 5622 | 968608 | 31392 |
| **512** | 1000000 | 16109 | 3438 | 983891 | 16109 |
| **1024** | 1000000 | 7446 | 1757 | 992554 | 7446 |
| **2048** | 1000000 | 3318 | 33 | 996682 | 3318 |
| **4096** | 1000000 | 2852 | 0 | 997148 | 2852 |
| **8192** | 1000000 | 2852 | 0 | 997148 | 2852 |
| **16384** | 1000000 | 2852 | 0 | 997148 | 2852 |

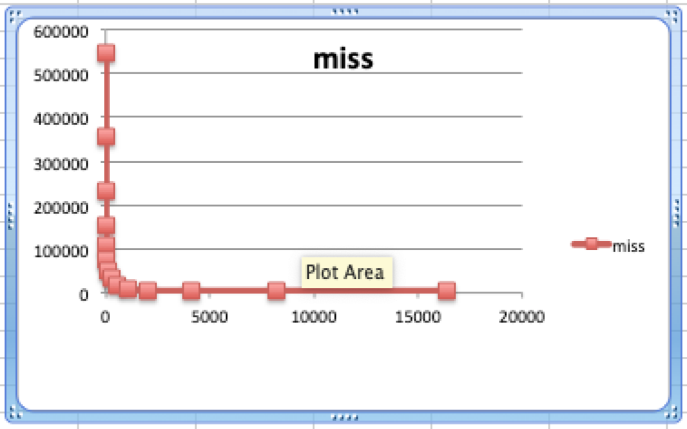
This table has all values for VMS when used gcc.trace we used power of 2 for nframes

## Hit rate vs cache size:



*This graph shows a hit rate for VMS. It shows how the number of hits increases as the cache size is increased. The hit rate does not change much beyond a certain point because of physical memory. For example from the graph, after a cache size of about 4096 further increasing the cache size does not affect the hit rate*

## Miss rate vs cache size:



*This graph shows a miss rate for VMS. It shows how the number of misses decreases as the cache size is increased. The miss rate does not change much beyond a certain point. For example from the graph, after a cache size of about 4096, further increasing the cache size does not affect the miss rate*

## Conclusion:

In conclusion from the results that we are getting, the number of frames increases shows we’re getting less disk reads and disk writes. For the FIFO and LRU table, when we read from the disk that will also count as a miss because different pages are being put into the frame. So that means from the results table above the total number of reads will equal to the same number of misses from the table. When we look at the hit rate from FIFO and LRU, when the number of frames increases, the number of hits increase as well because the same page is getting placed into the frame, which will be cause for no modifications. That means more pages with the same page are being incremented from the frame. The only difference in table between FIFO and LRU is the disk write, because the FIFO algorithm instantly swaps the page at the front of the queue which will be eliminate at the front of the queue. That means more pages has be modified at the front of the queue. LRU algorithm is slightly different than the FIFO algorithm because it tracks the oldest page from the queue. Once it is found, the new page will swap out the old page. When we look at the disk write between FIFO and LRU, that means less pages are being modified for LRU than FIFO. As for the VMS algorithm, also known as the second-chance page replacement algorithm, is relatively better than LRU and FIFO at little cost of improvement. So instead of immediately paging out a page, it gives it a second chance to checks to see if its referenced bit is clean. If it is not clean, the page is swapped out. Otherwise, the page is inserted at the back of the queue as if it’s being put in as a new page and this process is repeated. So that’s means there will be less disk reads and disk write than FIFO and LRU. So overall from the VMS table, it is more improved than FIFO and LRU. At the Hit and Miss vs Cache size graph for FIFO, it shows that when the cache size gets larger, the hit rate increases and the miss rate decreases. But once it reaches up to 32 or more frames, the hit rate stays slightly constant and the miss rate continues to reduce over a certain increase amount of cache size. At the Hit and Miss vs Cache size graph for LRU, it displays that when the cache size gets larger, the hit rate increases a little more than FIFO and the miss rate decreases a little less than FIFO. But once it reaches up to 32 or more frames, the hit rate stays slightly constant and the miss rate continues to reduce over a certain increase amount of cache size. At the Hit and Miss vs Cache size graph for VMS, it shows that at 2 frames, the hit rate starts doubling the amount and the miss rate starts dropping more than FIFO and LRU when the cache size gets larger. But once it reaches up to 64 or more frames, the hit rate stays slightly constant and the miss rate continues to reduce over a certain increase amount of cache size.