Study Set for Lecture 10: Virtual Memory

Due Nov 2 at 11:59pm **Points** 10 **Questions** 16
Available Oct 27 at 12pm - Dec 8 at 8pm about 1 month **Time Limit** None
Allowed Attempts Unlimited

Instructions

Review lecture notes from **lect10_Virtual_Memory.pdf**.

Then answer the questions from this study set and submit them to gain access to the further part of the course.

Take the Quiz Again

Attempt History

	Attempt	Time	Score
KEPT	Attempt 3	31 minutes	10 out of 10 *
LATEST	Attempt 3	31 minutes	10 out of 10 *
	Attempt 2	34 minutes	10 out of 10
	Attempt 1	less than 1 minute	0 out of 10 *

* Some questions not yet graded

Correct answers are hidden.

Score for this attempt: **10** out of 10 *
Submitted Dec 7 at 8:59pm
This attempt took 31 minutes.

Question 1

Not yet graded / 0 pts

Explain what a virtual memory is including the role of demand paging realized by the lazy swapper in memory virtualization.

Furthermore, describe the benefits of memory virtualization.

Your Answer:

Virtual memory is separation of user logical memory from physical memory where only part of the program needs to be in memory for execution. This is known as the lazy swapper method where a page is swapped into memory only when the page is needed (demand paging). the benefits of memory virtualization are:□Flexible process management: The process doesn't need to wait for an opening in memory so only a portion can go in giving it a better response time, the total logical space for all processes can be larger than the available physical address space giving it more concurrent processes and users, and the address spaces can be shared by several processes such as dynamic libraries.□Less I/O: Fewer pages to swap in and out and improved performance

Advantages :

- More processes may be maintained in the main memory: Because we are going to load only some of the pages of any particular process, there is room for more processes. This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in the ready state at any particular time.
- A process may be larger than all of main memory: One of the most fundamental restrictions in programming is lifted. A process larger than the main memory can be executed because of demand paging. The OS itself loads pages of a process in main memory as required.
- It allows greater multiprogramming levels by using less of the available (primary) memory for each process.

Causes of Thrashing :

1. **High degree of multiprogramming** : If the number of processes keeps on increasing in the memory than number of frames allocated to each process will be decreased. So, less number of frames will be available to each process. Due to this, page fault will occur more frequently and more CPU time will be wasted in just swapping in and out of pages and the utilization will keep on decreasing.

   For example:
   Let free frames = 400

Case 1: Number of process = 100
Then, each process will get 4 frames.

Case 2: Number of process = 400
Each process will get 1 frame.
Case 2 is a condition of thrashing, as the number of processes are increased,frames per process are decreased. Hence CPU time will be consumed in just swapping pages.

2. **Lacks of Frames**:If a process has less number of frames then less pages of that process will be able to reside in memory and hence more frequent swapping in and out will be required. This may lead to thrashing. Hence sufficient amount of frames must be allocated to each process in order to prevent thrashing.

Recovery of Thrashing :

- Do not allow the system to go into thrashing by instructing the long term scheduler not to bring the processes into memory after the threshold.
- If the system is already in thrashing then instruct the mid term schedular to suspend some of the processes so that we can recover the system from thrashing.

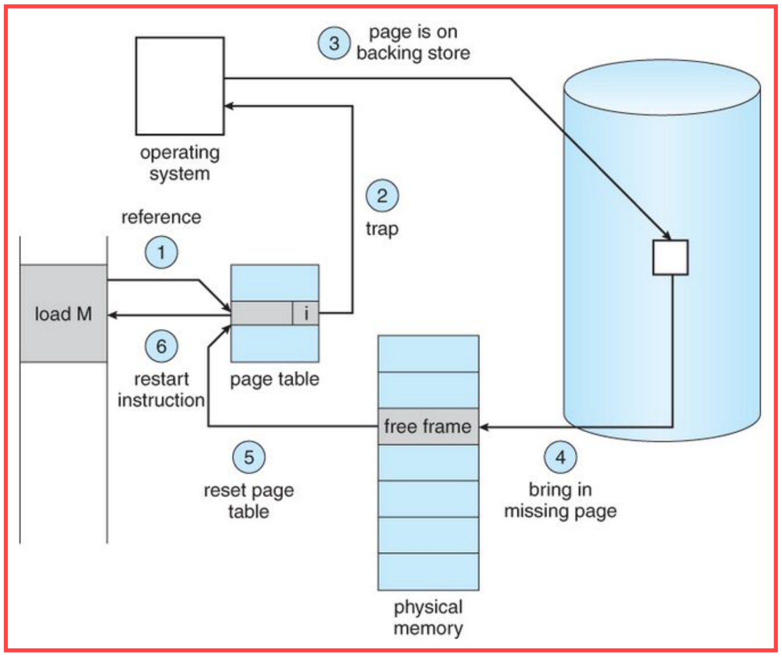
Question 2

Not yet graded / 0 pts

Explain what a page fault is, how it is detected, and describe step-by-step the process of page replacement.

Your Answer:

A page fault occurs when a program attempts to access data or code that is in its address space, but is not currently located in the system RAM. So when page fault occurs then following sequence of events happens :



- The computer hardware traps to the kernel and program counter (PC) is saved on the stack. Current instruction state information is saved in CPU registers.
- An assembly program is started to save the general registers and other volatile information to keep the OS from destroying it.
- Operating system finds that a page fault has occurred and tries to find out which virtual page is needed. Some times hardware register contains this required information. If not, the operating system must retrieve PC, fetch instruction and find out what it was doing when the fault occurred.
- Once virtual address caused page fault is known, system checks to see if address is valid and checks if there is no protection access problem.
- If the virtual address is valid, the system checks to see if a page frame is free. If no frames are free, the page replacement algorithm is run to remove a page.
- If frame selected is dirty, page is scheduled for transfer to disk, context switch takes place, fault process is suspended and another process is made to run until disk transfer is completed.
- As soon as page frame is clean, operating system looks up disk address where needed page is, schedules disk operation to bring it in.
- When disk interrupt indicates page has arrived, page tables are updated to reflect its position, and frame marked as being in normal state.

- Faulting instruction is backed up to state it had when it began and PC is reset. Faulting is scheduled, operating system returns to routine that called it.
- Assembly Routine reloads register and other state information, returns to user space to continue execution.

Question 3

Not yet graded / 0 pts

Discuss local frame allocation dilemma; i.e., options for distributing frames between processes.

Your Answer:

There are a few options for distributing frames between processes:

One is choosing the replacement scopes. Global replacement means the process selects a replacement frame from the set of all frames and one process may be given a frame taken away from another. Another is local replacement where each process selects only from it's own set of allocated frames.

Another option is how each does fixed allocation. Equal allocation means each process is given the same number of frames. On the other hand, proportional allocation allocates according to the size of the process.

Finally, you could do dynamic allocation, also known as priority allocation. Using a proportional allocation scheme using priorities rather than size, if a process generates a page fault, select for replacement a frame from a process with a lower priority.

Question 4

Not yet graded / 0 pts

Compute the memory effective access time in a system with the following characteristics:

- page faults happen once every 2000 memory accesses on average,
- disk access time is 5 ms,
- probability that the dirty bit is set on the vitctim page is 0.1,
- memory access time is 100 nanoseconds,
- page fault overhead is 7 nanoseconds, and
- restart overhead is 3 nanoseconds.

Your Answer:

EAT = (1 – p) * memory access + p * [page fault trap overhead + swap page out + swap page in + restart overhead]

EAT = (1 – (1/2000)) * 100 + (1/2000) * [7 + 5000000 + 5000000 + 3]

EAT = 5099.955 nanoseconds = 5.1e-3 ms

ALSO reference

p = 1/2000 = 0.002

mem access = 100 ns

read/write = 5ms = 5000000 ns

dirty bit = 0.1

page fault/restart overhead = 10 ns

EAT = (1-0.0005) **100 + 0.0005** (10 + 5000000 + 5000000(.1) + 10)

= (.9995)*100 + 0.0005(5500020)

99.95 + 2750.01

2849.96 nanoseconds

Question 5

Not yet graded / 0 pts

Assume that an OS with 3 frames uses OPT replacement policy . For the following reference set

```
{4, 3, 6, 2, 1, 2, 5, 4, 3, 4, 2, 3, 6, 1, 2, 3}
```

1. Show with details all page replacements.
2. State clearly how many page faults have occurred.

Show frame content (which pages are loaded) for each page reference per line. Use "-" for empty frame; otherwise, use the page number of the page loaded into a particular frame. Precede the number corresponding to the page being referenced with "<". If there is a page fault, indicate which page was replaced by preceding the corresponding number with "*".

For example, the following line:

```
- 4 2<
```

indicates that the first frame is free, the second frame holds page number 4, and the page number 2 held in the third frame has just been referenced. The following line:

```
6* 4 2
```

indicates that page 6 was referenced and faulted; it indicates that page replacement occurred in the first frame.

For example, in FIFO, the following string:

```
1 1 5 2 4 2 0 2 5 1 0 5 0 2 3
```

will be processed as follows:

```
 1* -  -
 1< -  -
 1  5* -
 1  5  2*
 4* 5  2
 4  5  2<
```

```
4  0* 2
4  0  2<
4  0  5*
1* 0  5
1  0< 5
1  0  5<
1  0< 5
1  2* 5
1  2  3*
```

Your Answer:

{4, 3, 6, 2, 1, 2, 5, 4, 3, 4, 2, 3, 6, 1, 2, 3}

*4 - -
4 *3 -
4 3 *6
4 3 *2
4 *1 2
4 1 >2
4 *5 2
>4 5 2
4 *3 2
>4 3 2
4 3 >2
4 >3 2
*6 3 2
*1 3 2
1 3 >2

1 >3 2

9 page faults

4: Memory is: 4 * * : Page Fault: (Number of Page Faults: 1)
3: Memory is: 4 3 * : Page Fault: (Number of Page Faults: 2)
6: Memory is: 4 3 6 : Page Fault: (Number of Page Faults: 3)
2: Memory is: 4 3 2 : Page Fault: (Number of Page Faults: 4)
1: Memory is: 4 1 2 : Page Fault: (Number of Page Faults: 5)
2: Memory is: 4 1 2 : Hit: (Number of Page Faults: 5)
5: Memory is: 4 5 2 : Page Fault: (Number of Page Faults: 6)
4: Memory is: 4 5 2 : Hit: (Number of Page Faults: 6)

```
3: Memory is: 4 3 2 : Page Fault: (Number of Page Faults: 7)
4: Memory is: 4 3 2 : Hit: (Number of Page Faults: 7)
2: Memory is: 4 3 2 : Hit: (Number of Page Faults: 7)
3: Memory is: 4 3 2 : Hit: (Number of Page Faults: 7)
6: Memory is: 6 3 2 : Page Fault: (Number of Page Faults: 8)
1: Memory is: 1 3 2 : Page Fault: (Number of Page Faults: 9)
2: Memory is: 1 3 2 : Hit: (Number of Page Faults: 9)
3: Memory is: 1 3 2 : Hit: (Number of Page Faults: 9)
Total Number of Page Faults: 9

Process finished with exit code 0
```

Question 6

Not yet graded / 0 pts

Assume that an OS with 3 frames uses FIFO replacement policy . For the following reference set

```
{4, 3, 6, 2, 1, 2, 5, 4, 3, 4, 2, 3, 6, 1, 2, 3}
```

1. Show with details all page replacements.
2. State clearly how many page faults have occurred.

Show frame content (which pages are loaded) for each page reference per line. Use "-" for empty frame; otherwise, use the page number of the page loaded into a particular frame. Precede the number corresponding to the page being referenced with "<". If there is a page fault, indicate which page was replaced by preceding the corresponding number with "*".

For example, the following line:

```
- 4 2<
```

indicates that the first frame is free, the second frame holds page number 4, and the page number 2 held in the third frame has just been referenced. The following line:

```
6* 4 2
```

indicates that page 6 was referenced and faulted; it indicates that page replacement occurred in the first frame.

For example, in FIFO, the following string:

```
1 1 5 2 4 2 0 2 5 1 0 5 0 2 3
```

will be processed as follows:

```
1* -  -
1< -  -
1  5* -
1  5  2*
4* 5  2
4  5  2<
4  0* 2
4  0  2<
4  0  5*
1* 0  5
1  0< 5
1  0  5<
1  0< 5
1  2* 5
1  2  3*
```

Your Answer:

{4, 3, 6, 2, 1, 2, 5, 4, 3, 4, 2, 3, 6, 1, 2, 3}

*4 - -
4 *3 -
4 3 *6
*2 3 6
2 *1 6
>2 1 6
2 1 *5
*4 1 5
4 *3 5
>4 3 5
4 3 *2
4 >3 2
*6 3 2
6 *1 2
6 1 >2
6 1 *3

12 page faults

```
4 -1 -1
4 3 -1
4 3 6
2 3 6
2 1 6
Hit & column > 0 :2 1 6
2 1 5
4 1 5
4 3 5
Hit & column > 0 :4 3 5
4 3 2
Hit & column > 1 :4 3 2
6 3 2
6 1 2
Hit & column > 2 :6 1 2
6 1 3
No. of page hit : 4

12 page faults

Process finished with exit code 0
```

Question 7

Not yet graded / 0 pts

Assume that an OS with 3 frames uses LRU replacement policy . For the following reference set

```
{4, 3, 6, 2, 1, 2, 5, 4, 3, 4, 2, 3, 6, 1, 2, 3}
```

1. Show with details all page replacements.
2. State clearly how many page faults have occurred.

Show frame content (which pages are loaded) for each page reference per line. Use "-" for empty frame; otherwise, use the page number of the page loaded into a particular frame. Precede the number corresponding to the page being referenced with "<". If there is a page fault, indicate which page was replaced by preceding the corresponding number with "*".

For example, the following line:

```
- 4 2<
```

indicates that the first frame is free, the second frame holds page number 4, and the page number 2 held in the third frame has just been referenced. The following line:

```
6* 4 2
```

indicates that page 6 was referenced and faulted; it indicates that page replacement occurred in the first frame.

For example, in FIFO, the following string:

```
1 1 5 2 4 2 0 2 5 1 0 5 0 2 3
```

will be processed as follows:

```
 1* -  -
 1< -  -
 1  5* -
 1  5  2*
 4* 5  2
 4  5  2<
 4  0* 2
 4  0  2<
 4  0  5*
 1* 0  5
 1  0< 5
 1  0  5<
 1  0< 5
 1  2* 5
 1  2  3*
```

Your Answer:

{4, 3, 6, 2, 1, 2, 5, 4, 3, 4, 2, 3, 6, 1, 2, 3}

*4 - -
4 *3 -
4 3 *6
*2 3 6
2 *1 6
>2 1 6
2 1 *5

2 *4 5
*3 4 5
3 >4 5
3 4 *2
>3 4 2
3 *6 2
3 6 *1
*2 6 1

2 *3 1

13 page faults

Question 8

Not yet graded / 0 pts

Explain what Bellady's Anomaly is.

Illustrate the problem by computing page faults in an OS that uses memory with three frames and with four frames.

Use the following reference set:

```
{1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5}
```

Your Answer:

Belady's Anomaly is where more frames can cause more page faults where it should cause less page faults.

In computer storage (https://en.wikipedia.org/wiki/Computer_storage) , **Bélády's anomaly** is the phenomenon in which increasing the number of page frames results in an increase in the number of page faults (https://en.wikipedia.org/wiki/Page_fault) for certain memory access patterns. This phenomenon is commonly experienced when using the first-in first-out (**FIFO (https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics))**) **page replacement algorithm (https://en.wikipedia.org/wiki/Page_replacement_algorithm)** . In FIFO, the page fault may or may not increase as the page frames increase, but in Optimal and stack-based algorithms like LRU, as the page frames increase the page fault decreases. **László Bélády (https://en.wikipedia.org/wiki/L%C3%A1szl%C3%B3_B%C3%A9l%C3%A1dy)** demonstrated this in 1969.[1] (https://en.wikipedia.org/wiki/B%C3%A9l%C3%A1dy%27s_anomaly#cite_note-1)

FIFO

***1 - -**
1 *2 -
1 2 *3
***4 2 3**
4 *1 3
4 1 *2
***5 1 2**
5 >1 2
5 1 >2
5 *3 2
5 3 *4
>5 3 4

9 page faults

***1 - - -**
1 *2 - -
1 2 *3 -
1 2 3 *4
>1 2 3 4
1 >2 3 4
***5 2 3 4**
5 *1 3 4

5 1 *2 4

5 1 2 *3

***4 1 2 3**

4 *5 2 3

10 page faults

Question 9

Not yet graded / 0 pts

Assume the following page reference sequence:

```
{0, 3, 4, 3, 3, 0, 6, 5, 5, 6, 0, 2, 1, 2, 4, 3, 3, 4, 2, 1, 1, 1,
2, 3, 4, 5, 6, 4, 5, 6, 5, 5, 6, 6, 1}
```

Show all successive working sets using a window Δ = 4. Use 4 reference bits that are set from the left for the pages that are referenced in the current window and then shifted right on every timeout. The "timeout" is simulated by counting page references; in this task assume a timeout every two page references.

Write C code that declares appropriate data structures and implements the algorithm using bit shifting to maintain the references bits. You do not need to use packing in this solution (i.e., you are allowed to utilize just 4 bits in a byte to hold the reference information.

Your Answer:

Step 1: Slide the window

Step 2: Do the right shift

Step 3: Look at everything within the window and mark the reference

Every 2 references is when the reference table gets updated

First working set is the first 2 in the window

Second working set takes into account all 4 in the window

When the window gets shifted, all 4 are taken into account

If the number of page references isn't a multiple of the window, then just use however many end up in the window

ALSO NOT SURE

-1 -1 -1

3 -1 -1

3 4 -1

3 4 -1

3 4 -1

3 4 -1

3 4 6

3 5 6

3 5 6

3 5 6

3 5 6

2 5 6

2 1 6

2 1 6

2 1 4

2 1 4

2 1 4

2 1 4

2 1 4

2 1 4

2 1 4

2 1 4

2 1 4

2 1 4

2 1 4

2 5 4

6 5 4

6 5 4

6 5 4

6 5 4

6 5 4

6 5 4

6 5 4

6 5 4

6 5 4

Total Page Faults = 12

Question 10

Not yet graded / 0 pts

Can a quasi-stack the top of which always keeps the last referenced page number and which provides access to every element (hence "quasi" in contrast to a true stack that can only push and pop at the top) be a foundation for implementing the LRU page replacement algorithm? How can you find the least recently used element?

Your Answer:

In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

In **L**east **R**ecently **U**sed (LRU) algorithm is a Greedy algorithm where the page to be replaced is least recently used. The idea is based on locality of reference, the least recently used page is not likely

```
Let capacity be the number of pages that
memory can hold.  Let set be the current
set of pages in memory.

ALGORITHM

1- Start traversing the pages.
 i) If set holds less pages than capacity.
   a) Insert page into the set one by one until
      the size  of set reaches capacity or all
      page requests are processed.
   b) Simultaneously maintain the recent occurred
      index of each page in a map called indexes.
   c) Increment page fault
 ii) Else
   If current page is present in set, do nothing.
   Else
     a) Find the page in the set that was least
     recently used. We find it using index array.
     We basically need to replace the page with
     minimum index.
     b) Replace the found page with current page.
     c) Increment page faults.
     d) Update index of current page.

2. Return page faults.
```

Question 11

Not yet graded / 0 pts

Consider a double-linked list-based implementation of the LRU quasi-stack that was discussed in the lecture.

Explain step-by-step the algorithm to keep the last referenced page at the top of the stack. Note that the page being referenced may, but does not need to, be already on the stack. Your algorithm must handle both cases.

What is the complexity of updating the stack? That is, how many operations are required to add a new page number at the top, or to move an already existing page number from some location in the stack to the top?

Your Answer:

It would keep the top pointing to the correct victim page because it would run through the array to figure out which number would be used closest, so in that cause it will be able to figure out what to pop off the stack.

Question 12

Not yet graded / 0 pts

Consider a fixed-size non-circular array-based implementation of the LRU quasi-stack that was discussed in the lecture.

Explain step-by-step the algorithm to keep the last referenced page at the top of the stack. Note that the page being referenced may, but does not need to, be already on the stack. Your algorithm must handle both cases.

Is it an efficient approach to implementing the LRU policy? What is the complexity of updating the stack? That is, how many operations are required to add a new page number at the top, or to move an already existing page number from some location in the stack to the top?

Your Answer:

When an LRU cache is full, we discard the *Least Recently* ***Used*** item.

If we're discarding items from the front of the queue, then, we have to make sure the item at the front is the one that hasn't been used for the longest time.

We ensure this by making sure that an item goes to the back of the queue whenever it *is* used. The item at the front is then the one that hasn't been moved to the back for the longest time.

To do this, we need to maintain the queue on every `put` OR `get` operation:

- When we `put` a new item in the cache, it becomes the *most* recently used item, so we put it at the back of the queue.
- When we `get` an item that is already in the cache, it becomes the *most* recently used item, so we *move* it from its current position to the back of the queue.

Moving items from the middle to the end is *not* a deque operation and is not supported by the `ArrayDeque` interface. It's also not supported efficiently by the underlying data structure that `ArrayDeque` uses. Doubly-linked lists are used because they *do* support this operation efficiently.

Question 13

Not yet graded / 0 pts

Consider a fixed-size circular array-based implementation of the LRU quasi-stack that was discussed in the lecture.

Explain step-by-step the algorithm to keep the last referenced page at the top of the stack. Note that the page being referenced may, but does not need to, be already in the stack. Your algorithm must handle both cases.

What is the complexity of updating the stack on each new page reference? That is, how many operations are required to add a new page number at the top, or to move an already existing page number from some location in the stack to the top?

HINT: Note that when the stack implemented as a circular array is full, the top is "circulated" by incrementing (modulo the size of the array) the index of the top and the index of the bottom, and then overwriting what used to be the stack bottom (which holds the number of currently least recently used page number) with the number of the newly referenced page.

Your Answer:

When an LRU cache is full, we discard the *Least Recently **Used*** item.

If we're discarding items from the front of the queue, then, we have to make sure the item at the front is the one that hasn't been used for the longest time.

We ensure this by making sure that an item goes to the back of the queue whenever it *is* used. The item at the front is then the one that hasn't been moved to the back for the longest time.

To do this, we need to maintain the queue on every `put` OR `get` operation:

- When we `put` a new item in the cache, it becomes the *most* recently used item, so we put it at the back of the queue.
- When we `get` an item that is already in the cache, it becomes the *most* recently used item, so we *move* it from its current position to the back of the queue.

Moving items from the middle to the end is *not* a deque operation and is not supported by the `ArrayDeque` interface. It's also not supported efficiently by the underlying data structure that `ArrayDeque` uses. Doubly-linked lists are used because they *do* support this operation efficiently.

Question 14

Not yet graded / 0 pts

Explain how to implement efficient page replacement algorithms that approximate LRU. What is a "second chance" approach?

Your Answer:

LRU cannot be implemented without special hardware. Aging is a way to approximate LRU which is based on LRU and the workign set. We can approximate the age/usage. With each page associate a reference bit. Initially set to 0; Change to 1 upon reference.
Search: if 1, then change to 0 and continue if 0, replace the page and change to 1.
Second change approach needs a second reference bit to simulate a clock by shifting the first reference bit to the second one.

Question 15

Not yet graded / 0 pts

Let’s consider the following array:

```
char data[4][4];
```

on a trivial machine with the memory consisting of a single 4-byte frame.

1) How many pages does `data` encompass?

Consider two alternate ways to traverse `data`:

```
// A
for (i = 0; i < 4; i++)
   for (j = 0; j < 4; j++)
      data[i][j] = 0;
```

```
// B
for (j = 0; j < 4; j++)
   for (i = 0; i < 4; i++)
      data[i][j] = 0;
```

2) Assuming that the row major array order is used on this machine (i.e., arrays are stored by rows — like in C), calculate how many page faults will occur executing A and how many will occur executing B.

Your Answer:

ssuming the generality that the character size is of 1 Byte, the given character array data[4][4] contains 16 elements each of size 1 Byte.

So the total space required to store the array = 4*4 Bytes = 16 Bytes.

Also, given that the frame size is of 4 Byte which implies the page size is also of 4 Byte according to standard memory management technique.

1) Number of pages that data encompasses = 16 Bytes/ 4 Bytes = 4 pages.

2) Given the array is stored in row major order. And given only one frame exists in the memory of the trivial machine given.

So, at a time it implies the memory can store four elements of the data array in a frame which is loaded with a page.

Execution of A is similar to row-wise access of the array because the variable i in data[i][j] is fixed as it is outer loop intially and then the inner loop variable j is varying from 0 to 3. So the data is accesses in a fashion data[0][0], data[0][1], data[0][2], data[0][3], data[1][0],...... so on. It is similar to the way the elements get stored in a page.

There will be four pages in the Virtual address space as shown below

Page 1----> data[0][0], data[0][1], data[0][2], data[0][3]

Page 2----> data[1][0], data[1][1], data[1][2], data[1][3]

Page 3----> data[2][0], data[2][1], data[2][2], data[2][3]

Page 4----> data[3][0], data[3][1], data[3][2], data[3][3]

So, while executing the A code, assuming initally there is no page in the only single frame present in the physical address space, the first page fault occurs while accessing data[0][0]. So, as the first page(Page 1) is loaded into the frame, accessing data[0][1], data[0][2], data[0][3] there won't be any page faults because the Page 1 itself contains all these three elements. Again while accesing data[1][0] there will be page fault and similar case for the rest of the elements in the page as told previously. And for every first element of the row, there will be a page fault.

*************So, totally while executing code A, there will be 4 page faults.

Execution of B is similar to column-wise access of the array because the variable j in data[i][j] is fixed as it is outer loop intially and then the inner loop variable i is varying from 0 to 3. So the data is accesses in a fashion data[0][0], data[1][0], data[2][0], data[3][0], data[0][1], data[1][1]...... so on. It is similar to the way the elements get stored in a page.

So, while executing the B code, assuming initally there is no page in the only single frame present in the physical address space, the first page fault occurs while accessing data[0][0]. So, as the Page 1 is loaded into the frame, accessing data[1][0] next to data[0][0] as per the code will again cause a page fault since the Page 1 which is already in the frame do not contain data[1][0] which is accessed next to data[0][0]. So, Page 2 is loaded on to the frame replacing Page 1. Similar is the case with data[2][0] and data[3][0]. After that data[0][1] is accessed when Page 4 is present in the frame. So, again page fault occurs in this case. Similarly, as told previously again the page faults occurs for each and every access in this case.

*************So, totally while executing code B, there will be 16 page faults.

Question 16

10 / 10 pts

I have submitted answers to all questions in this study set.

[✓ True]

False

Quiz Score: **10** out of 10