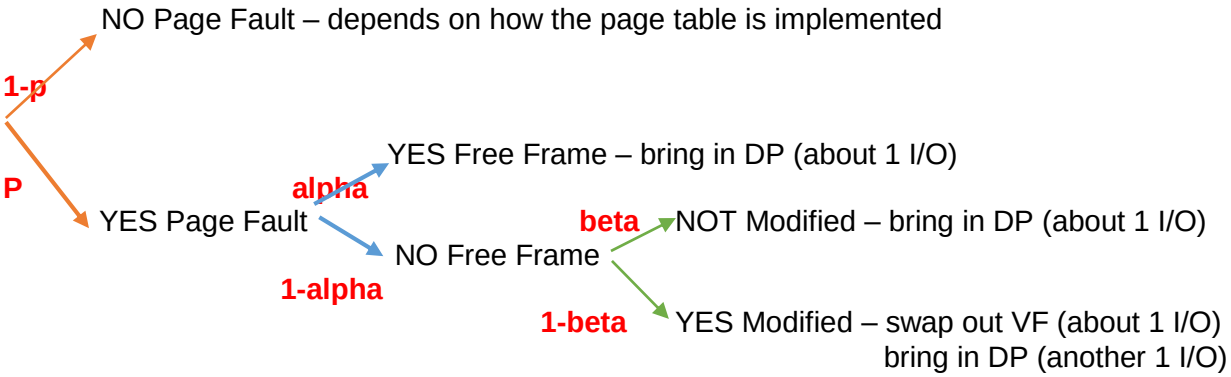
340 Memory 4

Performance of Demand Paging



Performance of Demand Paging

p - the probability of a page fault / page fault rate

effective access time = (1-p) * NO PF time + p x YES PF time

It is important to keep the page fault rate low in a demand-paging system.
An average page-fault service is of order of milliseconds.
A memory access time is of order of nanoseconds.

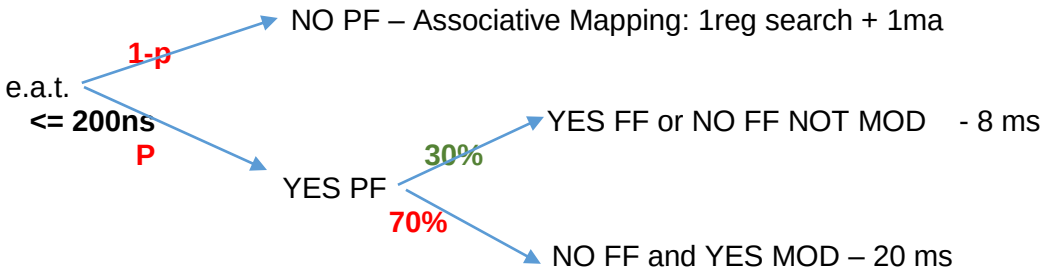
e.a.t = (1-p) * NO PF time + p x YES PF time
YES PF = alpha * YES Free Frame + (1 – alpha)* NO Free Frame
NO Free Frame = beta * NOT Modifier + (1-beta) * YES Modified

10.5 Assume that we have a demand-paged memory. The **page table is held in registers**. It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified, and 20 milliseconds if the replaced page modified. Memory access time is 100 nanoseconds. Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for effective access time of no more than 200 nanoseconds?

e.a.t. = (1-p) * NO PF time + p x YES PF time

p = (e.a.t. – NO PF time) / (YES PF time – NO PF time)

Let's tailor our page fault tree based on what the problem says.



p = (e.a.t. – NO PF time) / (YES PF time – NO PF time)

NO PF = 1 reg search + 1 ma = 0 + 100ns = 100ns

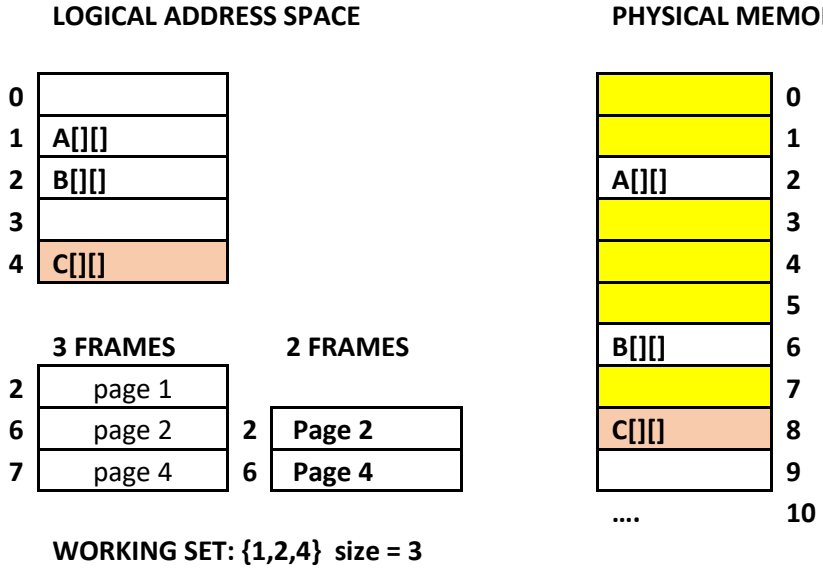
YES PF = .30 * YES FF or NO FF NOT MOD + .70 * NO FF and YES MOD
YES PF = .30 * 8ms + .70 * 20 ms

Keep in mind that: 1ms = 10^{-3} seconds; 1ns = 10^{-9} seconds

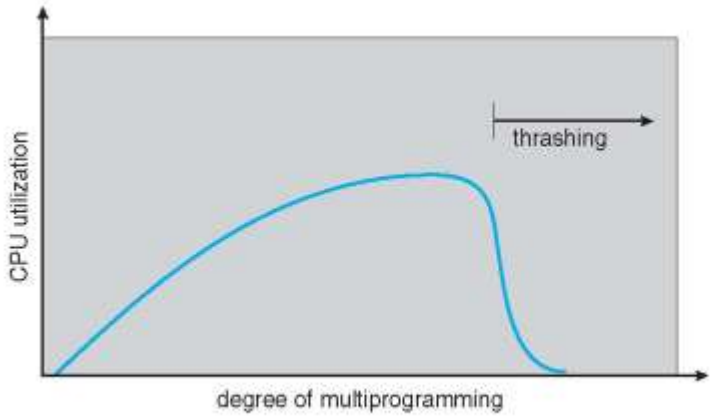
You will end up with a very small page fault rate but its value makes sense based on the problem's data and it is doable as well.

Principle of Locality: memory references then to cluster.
Spatial Locality
Temporal Locality

As mentioned in previous lecture:
Consider A[100] [100] and B [100] [100] both with 10 000 elements.
And the computer will execute C = A + B, C will also have 10 000 elements.



Thrashing If a process spends more time paging than executing, then that process is thrashing. As the degree of multiprogramming increases, CPU utilization increases until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in and CPU utilization drops sharply.



Thrashing results in severe performance problems. Consider the following scenario, which is based on the actual behavior of early paging systems.
The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.
The CPU scheduler sees the decreasing CPU utilization and ***increases*** the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer
queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred, and system throughput plunges. The page fault rate increases tremendously. As a result, the effective memory-access time increases. No work is getting done, because the processes are spending all their time paging.

Trying to limit trashing:
- use a **local replacement algorithm**. At least the process that will start thrashing will not steal frames from other processes. Anyway, because any process will have to wait longer on the paging queue, the effective access time will increase even for a process that is not thrashing.

Other Considerations

Pre-paging In general, at the beginning of execution, all the frames allocated to a process are empty. In the process of bringing the initial locality into memory, with each frame being filled up we will have a page fault. When a swapped-out process is restarted, all its pages on the disk must be brought in by its own page fault. By prepaging, a set of pages is brought in advance into main memory, before the process starts its execution.

10.10 Consider the two-dimensional array A:

```
int A [] [] = new int [100] [100];
```

where A [0] [0] is stored at location 200, in a paged memory system with pages of size 200. A small process resides in page 0 (location 0 to 199) for manipulating the A matrix; thus, every instruction fetch will be from page 0.

For three page frames, how many page faults are generated by the following array-initialization loops, using LRU replacement, and assuming page frame 1 has the process in it, and the two are initially empty:

a.
```
for (int j = 0; j < 100; j++)
    for ( int i = 0; i < 100; i++)
        A [i] [j] = 0;
```

b.
```
for (int i = 0; i < 100; i++ )
    for (int j = 0; j < 100; j++)
        A [i] [j] = 0;
```

First make sure that you understand what the problem gives you as data and what you need to find out.

Known: array of 100x100 elements = 10,000 elements (words)

Page size = 200 → this means that A will take 50 pages.

A[0][0] start at location 200; this means at the very beginning of page 1.

Page 0 covers location 0 to 199, and contains the instructions

Page 1 covers location 200 to 399, and so on.

All the above information allows you to do is to draw the logical address space and write down the information that it contains.

Based on the fact that the **OS always allocates space for arrays and vectors by row,** since the page size is 200 and the array contains 100 elements /row, you will have 2 rows fitting in each page.

DRAW THE LOGICAL ADDRESS SPACE (VERY IMPORTANT)

instructions	
A[0][0]…A[0][99] A[1][0]…A[1][99]	0 (locations 0 to 199) 1 (locations 200 to 399)
row 2, row 3	2
row 4 row 5	3
	4
	5

.
.
.

Basically now, if a specific element is mentioned, for example A[37][52] you should be able to figure out in which page it is in. Placing the array correctly into the logical address space is the most important part of the problem.

Demand paging is used. The process gets 3 frames and the instructions (page 0) is already pre paged in one of the frames.

page 0	Frame 1
	Frame 2
	Frame 3

The problem asks for the number of page faults for two different initializations.
a. initializes the array by column
b. initializes the array by row

The outcome is the same no matter how you initialize the array (by column or by row) but the performance is different.

a.
```
for (int j = 0; j < 100; j++)
    for ( int i = 0; i < 100; i++)
        A [i] [j] = 0;
```

We need to number the page faults based on the reference string:
For each initialization, first the instruction must be fetched.

Fetch instruction	reference to page 0	no page fault, already prepaged
Initialize A[0][0]	reference to page 1	**page fault**

page 0
page 1

Fetch instruction	reference to page 0	no page fault, already prepaged
Initialize A[1][0]	reference to page 1	no page fault
Fetch instruction	reference to page 0	no page fault, already prepaged
Initialize A[2][0]	reference to page 2	**page fault**

page 0
page 1
page 2

Fetch instruction	reference to page 0	no page fault,	already prepaged
Initialize A[3][0]	reference to page 2	no page fault	
Fetch instruction	reference to page 0	no page fault,	already prepaged
Initialize A[4][0]	reference to page 3	**page fault**	

(In order to save time) You can write your reference string as:

0	**1**	0	1	0	**2**	0	2
instr	A[0][0]	Instr	A[1][0]	Instr	A[2][0]	instr	A[3][0]
no pf	yes pf	no pf	no pf	no pf	yes pf	no pf	no pf

0	**3**	……..
Instr	A[4][0]	
no pf	yes pf	

All frames are taken; we need to use replacement using Least Recently Used.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. We can think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward.

Between pages 0,1,2 the page that was the least recently used will be replaced. In this case it will be page 1.

page 0
page 3
page 2

Fetch instruction	reference to page 0	no page fault, already prepaged
Initialize A[5] [0]	reference to page 3	no page fault
Fetch instruction	reference to page 0	no page fault, already prepaged
Initialize A[6][0]	reference to page 4	page fault; replacement is used and page 2 will be replaced.

page 0
page 3
page 4

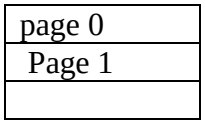
And so on. I hope you see that there is a trend.

There is a page fault for every other element being initialized. In total 10,000 words/2 words per page fault = 5000 page faults;
Or you can think of it as initializing the first column will take 50 page faults. We have 100 columns which means 5000 page faults.

b. for (int i = 0; i < 100; i++)
for (int j = 0; j < 100; j++)
A [i] [j] = 0;

This is initialization by row. Nothing changes in here when it comes to the contents of the logical address space. What changes is the reference string and the number of page faults.

Fetch instruction	reference to page 0	no page fault,	already prepaged
Initialize A[0][0]	reference to page 1	**page fault**	



Fetch instruction	reference to page 0	no page fault,	already prepaged
Initialize A[0][1]	reference to page 1	no page fault	
Fetch instruction	reference to page 0	no page fault,	already prepaged
Initialize A[0][2]	reference to page 1	no page fault	

Initialize A[0][99]	reference to page 1	no page fault	
Fetch instruction	reference to page 0	no page fault,	already prepaged

---

Initialize A[1][99]	reference to page 1	no page fault	
Fetch instruction	reference to page 0	no page fault,	already prepaged
Initialize A[2][0]	reference to page 2	page fault	

Page 2 is brought in main memory; and ……so on.
We can see that we can initialize 200 elements, 2 rows of the array, with only one page fault. Since we have 100 rows, in this case the total number of page faults will be 50.

For this array and problem, the initialization by row is much more efficient than the one by column. That is one of the reasons for which you are taught to initialize and traverse arrays by rows rather than by columns.

In the exam you will have a similar problem; It will be a smaller array, maybe it starts from the beginning of a page or maybe somewhere in the middle, or end. Again it is important to draw the array correctly in the logical address space. The array will be small enough such that you will be able to see it all. It's also useful to also mark the locations that each page covers.