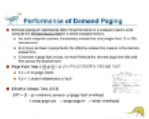
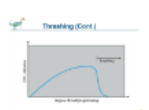
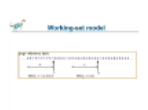
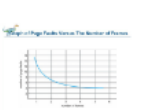
Virtual Memory

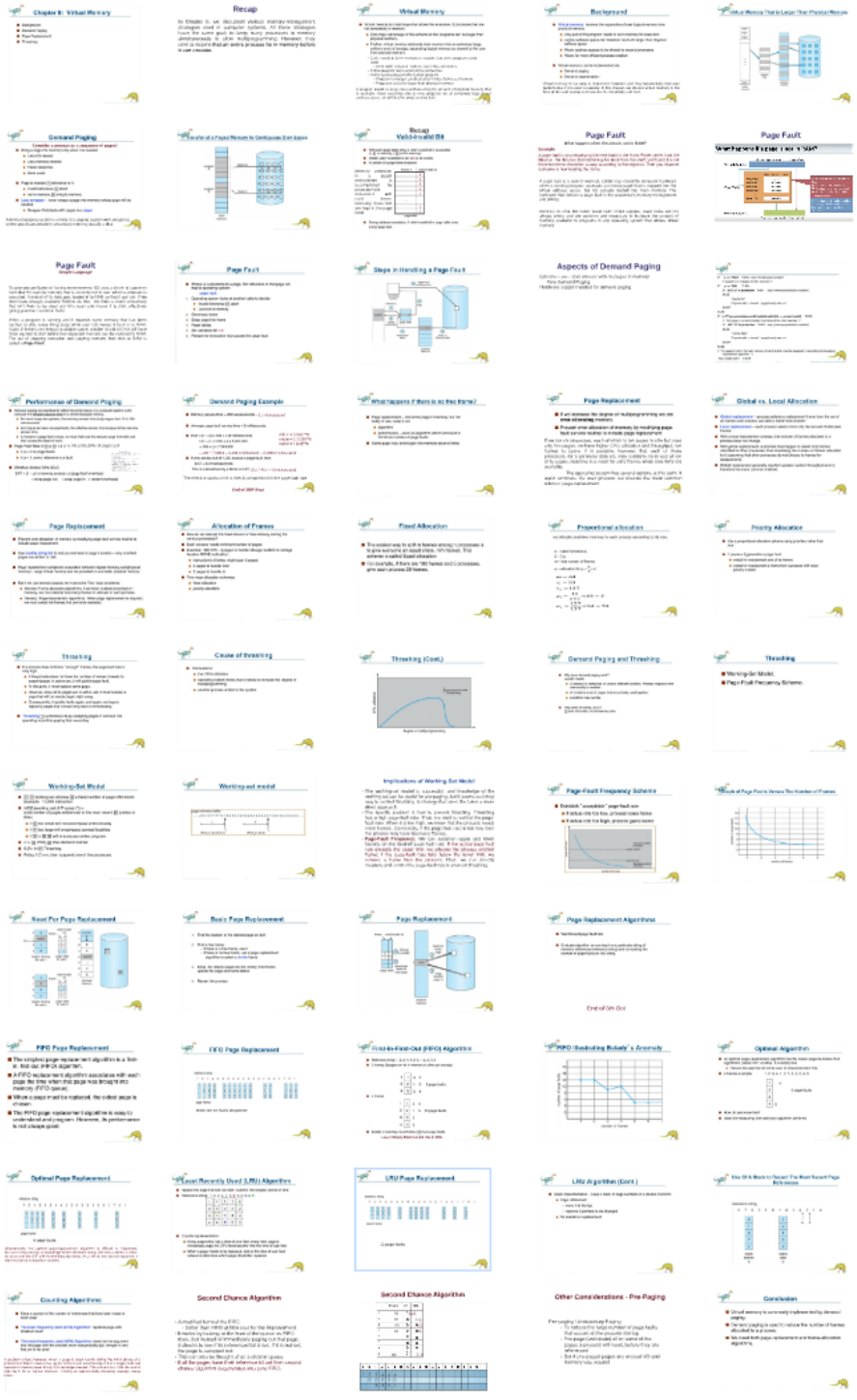








Virtual Memory



Chapter 9: Virtual Memory

- Background
- Demand Paging
- Page Replacement
- Thrashing

Recap

In Chapter 8, we discussed various memory-management strategies used in computer systems. All these strategies have the same goal: to keep many processes in memory simultaneously to allow multiprogramming. However, they tend to require **that an entire process be in memory before it can execute**.

Virtual Memory

- Virtual memory is a technique that allows the execution of processes that are not completely in memory.
  - One major advantage of this scheme is that programs can be larger than physical memory.
  - Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory.
    - Code needs to be in memory to execute, but entire program rarely used
      - Error code, unusual routines, large data structures
    - Entire program not needed at the same time
    - Ability to execute partially-loaded program
      - Program no longer constrained by limits of physical memory
      - Programs could be larger than physical memory

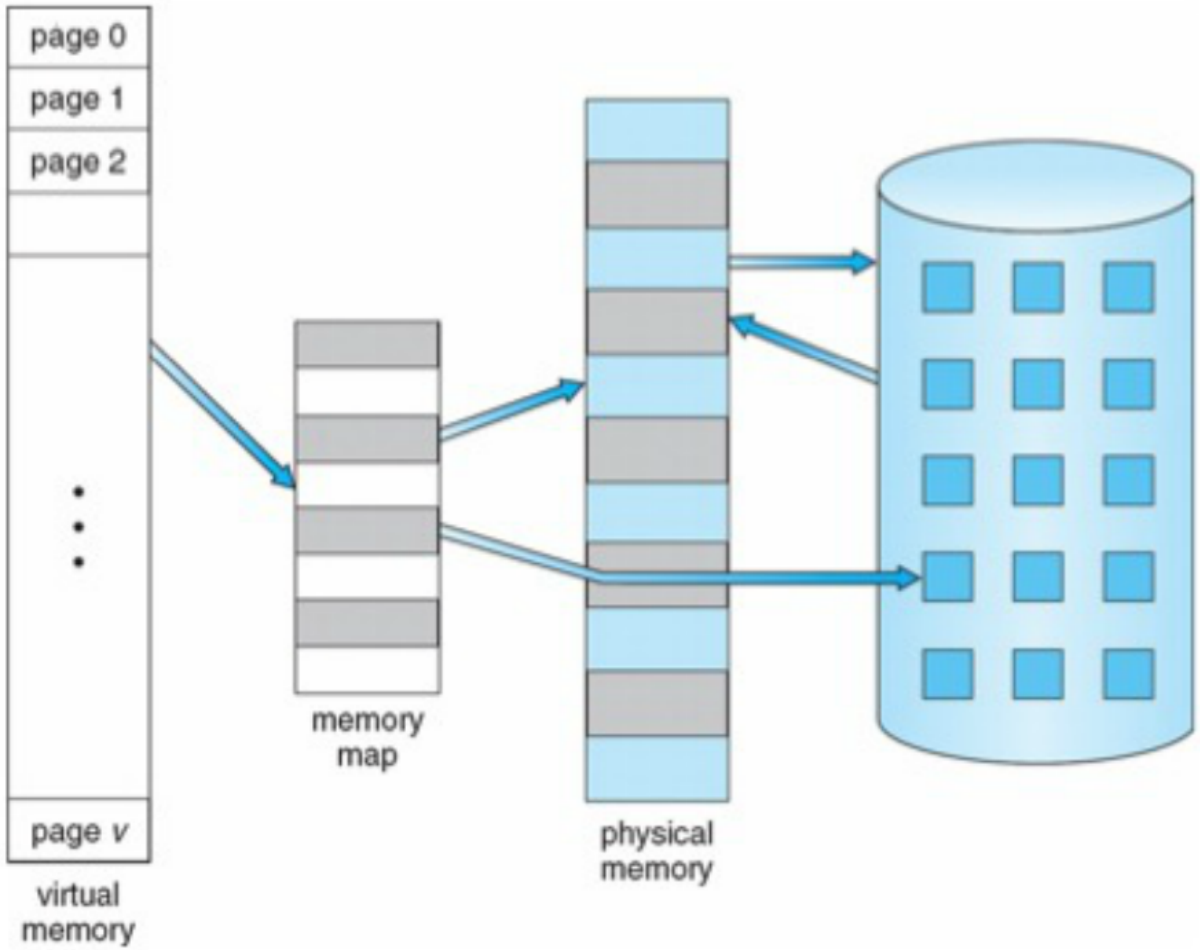
A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large virtual address space, simplifying the programming task.

Background

- **Virtual memory** involves the separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly. In this chapter, we discuss virtual memory in the form of demand paging and examine its complexity and cost.

Virtual Memory That is Larger Than Physical Memory



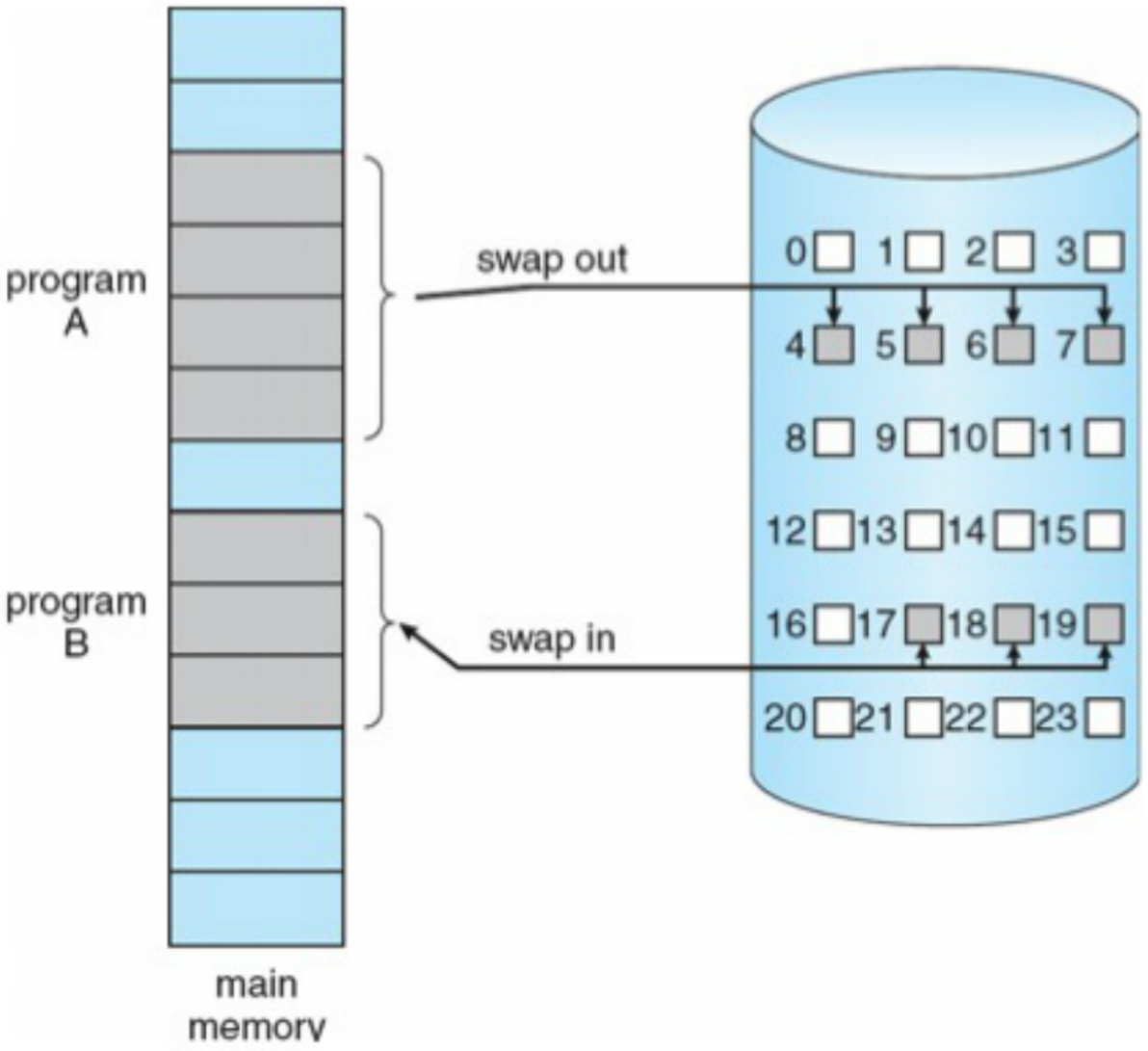
Demand Paging

Consider a process as a sequence of pages!

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory usually a disk.

Transfer of a Paged Memory to Contiguous Disk Space



Recap
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (v ⇨ in-memory, i ⇨ not-in-memory)
- Initially valid–invalid bit is set to i on all entries
- Example of a page table snapshot:

Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table.

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
....	
	i
	i

page table

- During address translation, if valid–invalid bit in page table entry is I ⇨ page fault

Page Fault

What happen when the data is not in RAM?

Example

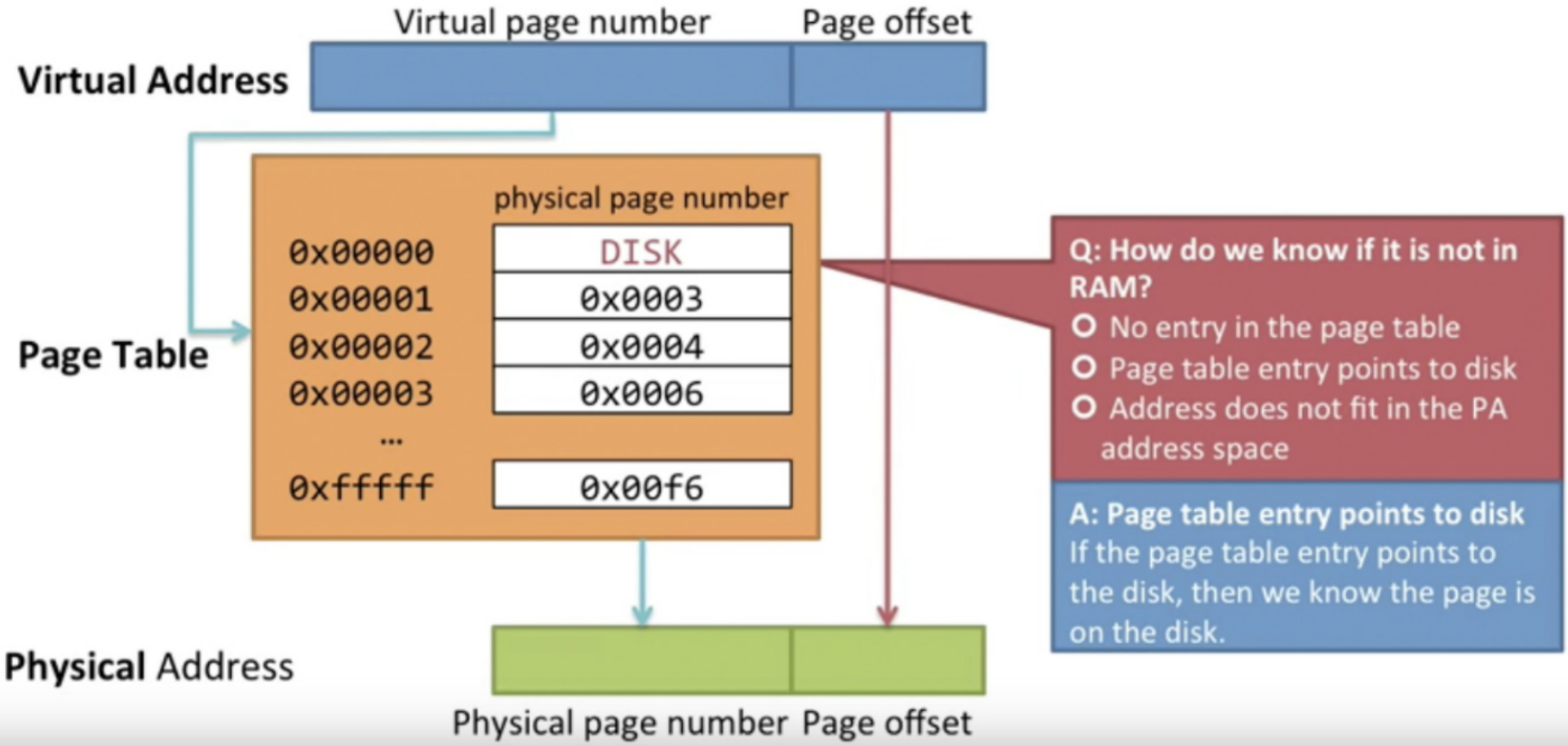
A page fault is you requesting the next book in the Harry Potter series from the librarian, the librarian tried retrieving the book from the shelf, and found it is not there but there should be a copy according to the registrar. Then you discover someone is now reading the same.

A page fault is a type of interrupt, called trap, raised by computer hardware when a running program accesses a memory page that is mapped into the virtual address space, but not actually loaded into main memory. The hardware that detects a page fault is the processor's memory management unit (MMU)

Contrary to what the name "page fault" might suggest, page faults are not always errors and are common and necessary to increase the amount of memory available to programs in any operating system that utilizes virtual memory

Page Fault

What happens if a page is not in RAM?



Page Fault

Simple Language

To give you an illusion of having more memory, OS uses a chunk of space on hard disk for caching memory that is currently not in use. When a program is executed, it and all of its data gets loaded in to RAM so that it can run. If we don't have enough available RAM to do this, OS finds a chunk of memory that isn't likely to be used any time soon and moves it to disk, effectively giving you more available RAM.

When a program is running and it requests some memory that has been cached to disk, every thing stops while your OS moves it back in to RAM. Again, if there is not enough available space, another chunk of RAM will have to be cached to disk before the requested memory can be restored to RAM. The act of stopping execution and copying memory from disk to RAM is called a ***Page Fault***.

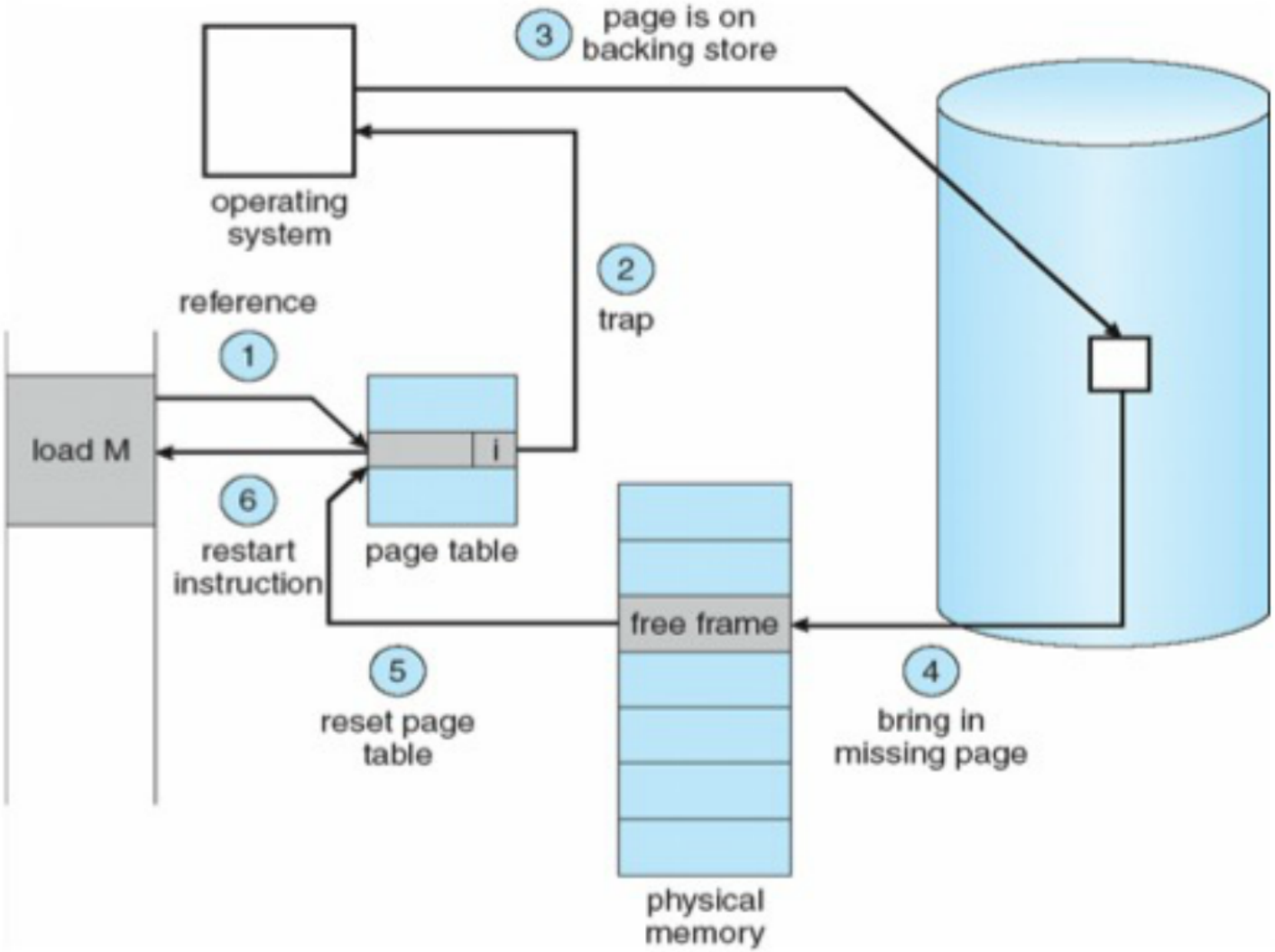
Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Aspects of Demand Paging

Extreme case - start process with *no* pages in memory

- Pure Demand Paging

Hardware support needed for demand paging

Performance of Demand Paging

- Demand paging can significantly affect the performance of a computer system. Let's compute the effective access time for a demand-paged memory.
  - For most computer systems, the memory-access time (ma) ranges from 10 to 200 nanoseconds.
  - As long as we have no page faults, the effective access time is equal to the memory access time.
  - If, however a page fault occurs, we must first read the relevant page from disk and then access the desired word.
- Page Fault Rate 0 ≤ p ≤ 1.0 p is the probability of page fault
  - if p = 0 no page faults
  - if p = 1, every reference is a fault
- Effective Access Time (EAT)

  EAT = (1 – p) x memory access+ p (page fault overhead

  + swap page out + swap page in + restart overhead)

Page Fault Time

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

- Trap to the operating system.
- Save the user process state.
- Determine that the interrupt was a page fault.
- Check that the page reference was legal and determine the location of the page on the disk.
- Issue a read from the disk to a free frame:
  - a. Wait in a queue for this device until the read request is serviced.
  - b. Wait for the device seek and/ or latency time.
  - c. Begin the transfer of the page to a free frame.
- While waiting, allocate the CPU to some other user (CPU scheduling, optional).
- Receive an interrupt from the disk I/O subsystem (I/O completed).
- Save the process state for the other user.
- Determine that the interrupt was from the disk
- Correct the page table and other tables to show that the desired page is now in memory.
- Wait for the CPU to be allocated to this process again.
- Restore the user process state, and new page table, and then resume the interrupted instruction.

Page Fault Time

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

- Trap to the operating system.
- Save the user process state.
- Determine that the interrupt was a page fault.
- Check that the page reference was legal and determine the location of the page on the disk.
- Issue a read from the disk to a free frame:
  - a. Wait in a queue for this device until the read request is serviced.
  - b. Wait for the device seek and/ or latency time.
  - c. Begin the transfer of the page to a free frame.
- While waiting, allocate the CPU to some other user (CPU scheduling, optional).
- Receive an interrupt from the disk I/0 subsystem (I/0 completed).
- Save the process state for the other user.
- Determine that the interrupt was from the disk
- Correct the page table and other tables to show that the desired page is now in memory.
- Wait for the CPU to be allocated to this process again.
- Restore the user process state, and new page table, and then resume the interrupted instruction.

Performance of Demand Paging

- Demand paging can significantly affect the performance of a computer system. Let's compute the effective access time for a demand-paged memory.
  - For most computer systems, the memory-access time (ma) ranges from 10 to 200 nanoseconds.
  - As long as we have no page faults, the effective access time is equal to the memory access time.
  - If, however a page fault occurs, we must first read the relevant page from disk and then access the desired word.
- Page Fault Rate 0 ≤ p ≤ 1.0 p is the probability of page fault
  - if p = 0 no page faults
  - if p = 1, every reference is a fault
- Effective Access Time (EAT)

  EAT = (1 – p) x memory access+ p (page fault overhead

  + swap page out + swap page in + restart overhead)

Page Fault Time

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

- Trap to the operating system.
- Save the user process state.
- Determine that the interrupt was a page fault.
- Check that the page reference was legal and determine the location of the page on the disk.
- Issue a read from the disk to a free frame:
  - a. Wait in a queue for this device until the read request is serviced.
  - b. Wait for the device seek and/ or latency time.
  - c. Begin the transfer of the page to a free frame.
- While waiting, allocate the CPU to some other user (CPU scheduling, optional).
- Receive an interrupt from the disk I/O subsystem (I/O completed).
- Save the process state for the other user.
- Determine that the interrupt was from the disk
- Correct the page table and other tables to show that the desired page is now in memory.
- Wait for the CPU to be allocated to this process again.
- Restore the user process state, and new page table, and then resume the interrupted instruction.

Demand Paging Example

- Memory access time = 200 nanoseconds = 0.2 microsecond
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)
  = (1 – p) x 200 + p x 8,000,000
  = 200 + p x 7,999,800

mili = 1 / (10) **3
micro = 1 / (10)**6
nano=1 / (10)**9

= 200 + 7,999.8 = 8,199.8 milisecond = 8.1998 microsecond

- If one access out of 1,000 causes a page fault, then
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!! (0.2 * 40) == 8 microsecond

The effective access time is directly proportional to the page-fault rate.

End of 30th Sept

What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Page Replacement

- If we increase the degree of multiprogramming we are **over-allocating** memory.
- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

If we run six processes, each of which is ten pages in size but uses only five pages, we have higher CPU utilization and throughput, ten frames to spare. It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all ten of its pages, resulting in a need for sixty frames when only forty are available.

The operating system has several options at this point. It could terminate the user process. we discuss the most common solution: page-replacement

Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
- **Local replacement** – each process selects from only its own set of allocated frames
- With a local replacement strategy, the number of frames allocated to a process does not change.
- With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it (assuming that other processes do not choose its frames for replacement).
- Global replacement generally results in greater system throughput and is therefore the more common method.

Allocation of Frames

- How do we allocate the fixed amount of free memory among the various processes?
- Each process needs *minimum* number of pages
- Example: IBM 370 – 6 pages to handle Storage location to storage location MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- The easiest way to split m frames among n processes is to give everyone an equal share, m/n frames. This scheme is called Equal allocation
- For example, if there are 100 frames and 5 processes, give each process 20 frames.

Proportional allocation

we allocate available memory to each process according to its size.

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

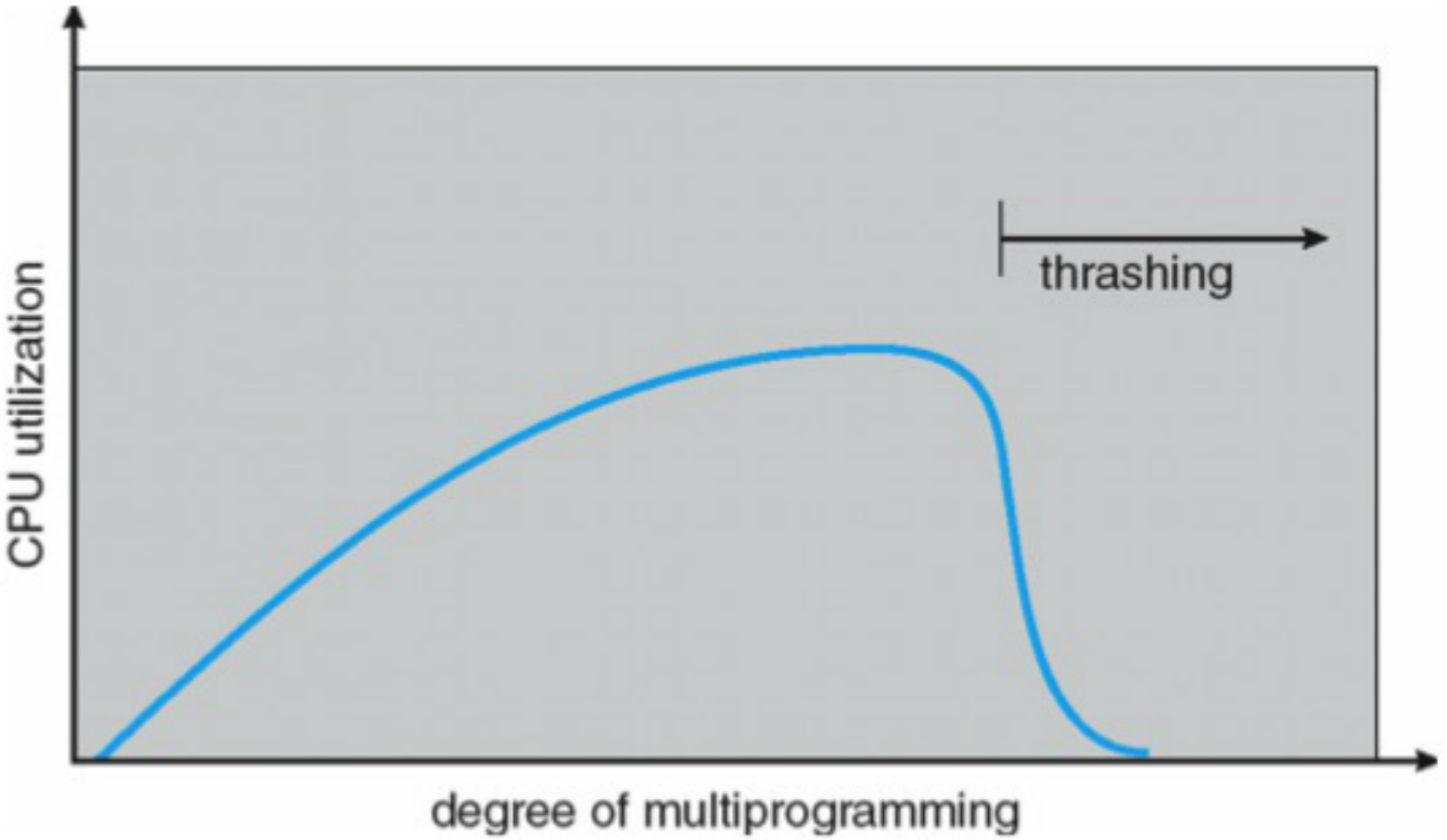
Thrashing

- If a process does not have "enough" frames, the page-fault rate is very high.
  - If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault.
  - At this point, it must replace some page.
  - However, since all its pages are in active use, it must replace a page that will be needed again right away.
  - Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.
- **Thrashing** ⁘ a process is busy swapping pages in and out, it is spending more time paging than executing.

Cause of thrashing

- This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system

Thrashing (Cont.)



Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - A process is composed of several different localities. Process migrates from one locality to another.
  - A locality is a set of pages that are actively used together.
  - Localities may overlap

- Why does thrashing occur?
  ☒ size of locality > total memory size

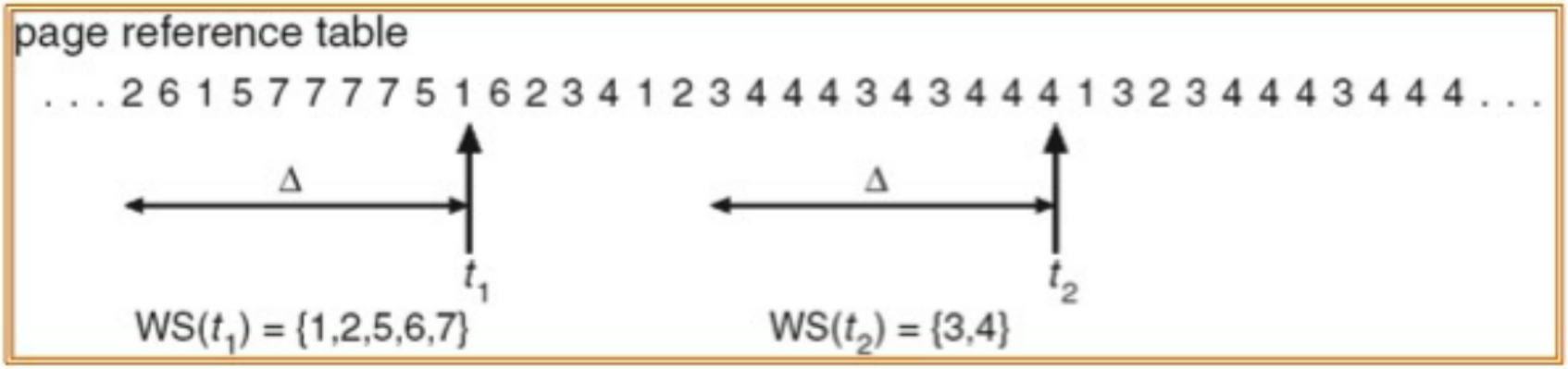
Thrashing

- Working-Set Model.
- Page-Fault Frequency Scheme.

Working-Set Model

- [illegible] working-set window [illegible] a fixed number of page references
  Example: 10,000 instruction
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent [illegible] (varies in time)
  - if [illegible] too small will not encompass entire locality
  - if [illegible] too large will encompass several localities
  - if [illegible] = [illegible] [illegible] will encompass entire program
- D = [illegible] WSS_i [illegible] total demand frames
- if $D > m$ [illegible] Thrashing
- Policy if $D >$ m, then suspend one of the processes

Working-set model

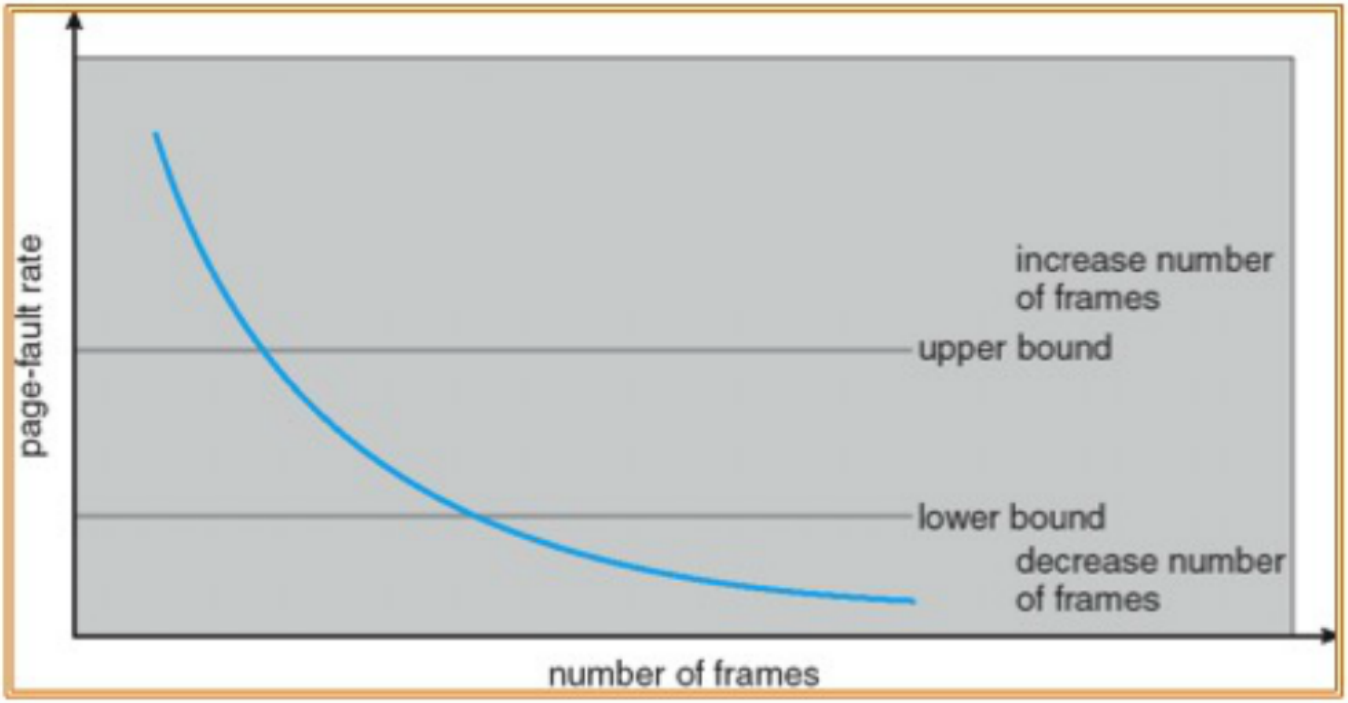


Implications of Working-Set Model

- The working-set model is successful, and knowledge of the working set can be useful for pre-paging, but it seems a clumsy way to control thrashing. A strategy that uses the takes a more direct approach.
- The specific problem is how to prevent thrashing. Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames.
- **Page-Fault Frequency:** We can establish upper and lower bounds on the desired page-fault rate. If the actual page-fault rate exceeds the upper limit, we allocate the process another frame; if the page-fault rate falls below the lower limit, we remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.

Page-Fault Frequency Scheme

- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Basic Page Replacement

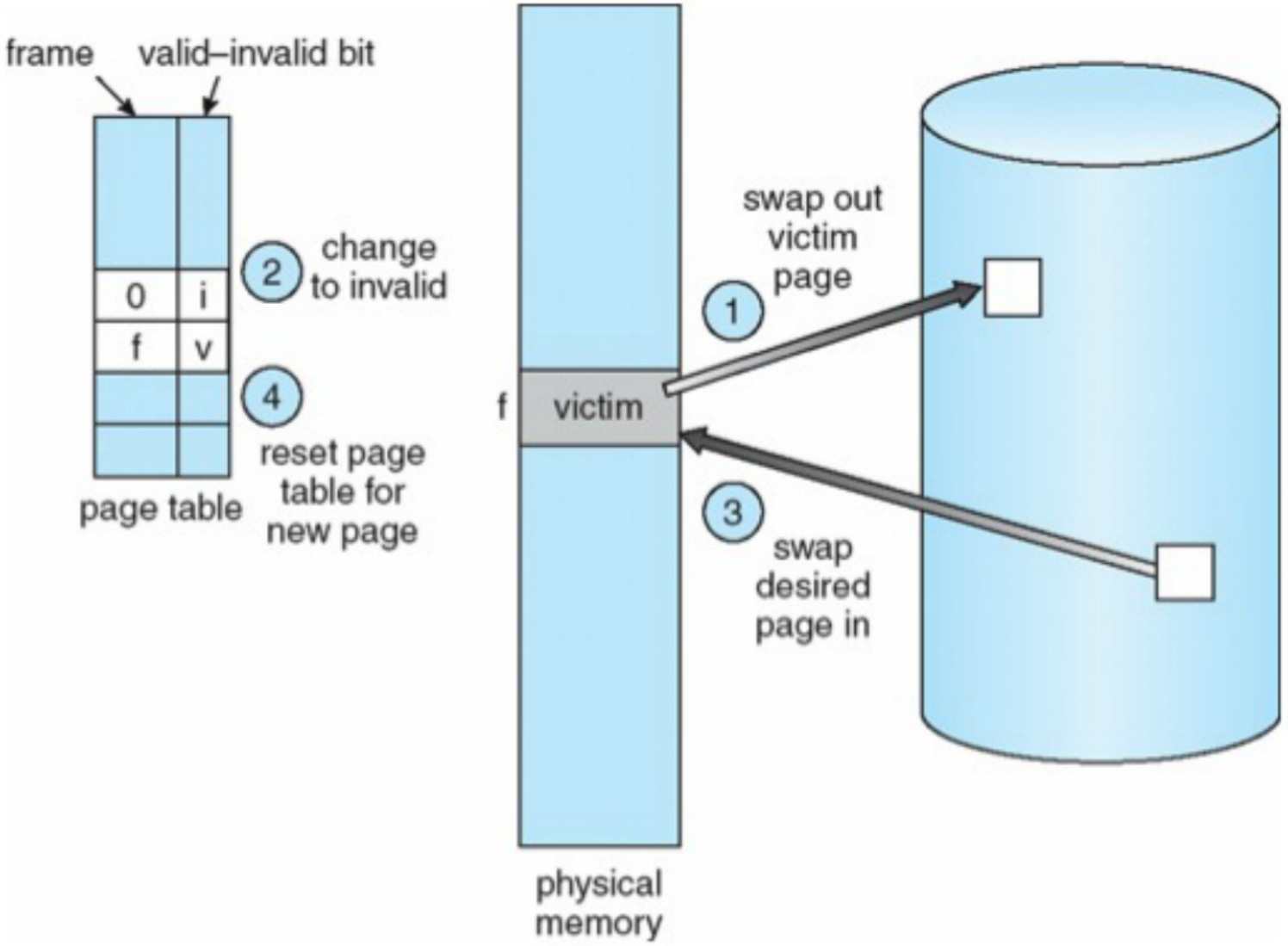
1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Restart the process

Page Replacement



Page Replacement Algorithms

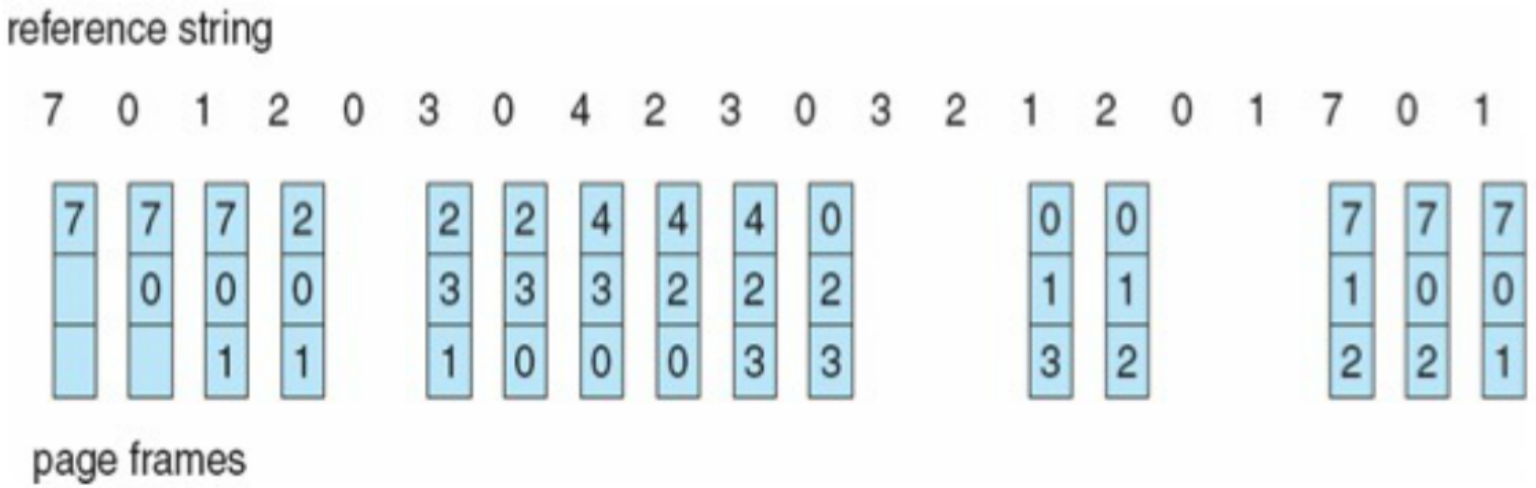
- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

End of 5th Oct

FIFO Page Replacement

- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm.
- A FIFO replacement algorithm associates with each page the time when that page was brought into memory (FIFO queue).
- When a page must be replaced, the oldest page is chosen.
- The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good.

FIFO Page Replacement

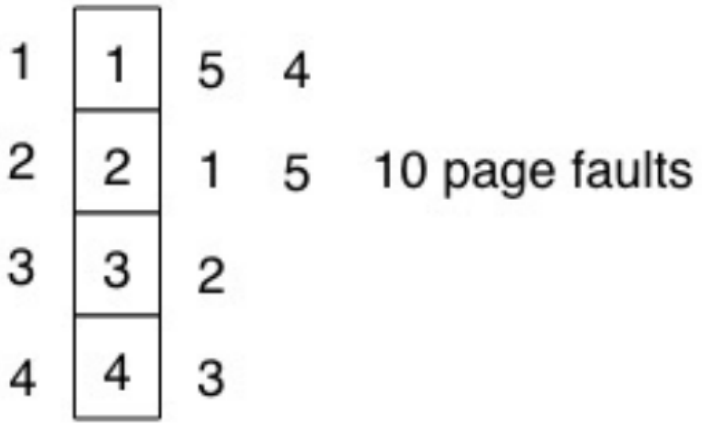


there are 15 faults altogether

First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

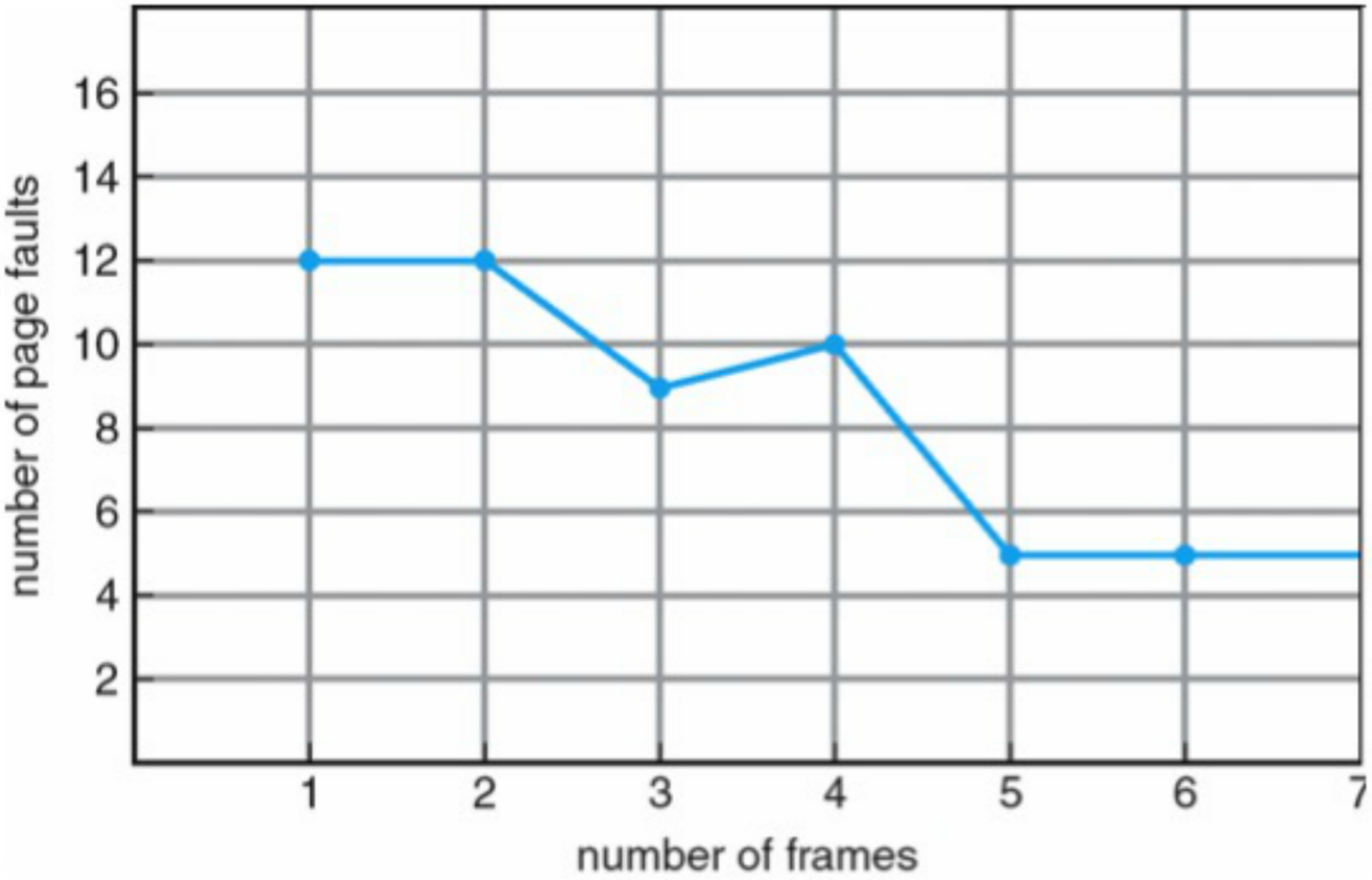
- 4 frames



- Belady's Anomaly: more frames ⇒ more page faults

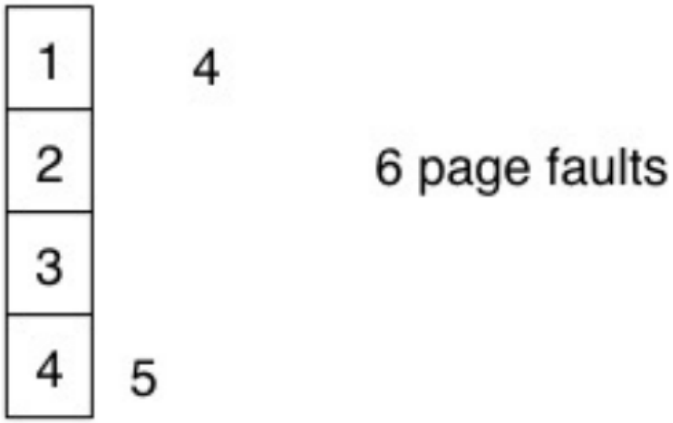
László Bélády demonstrated this in 1969.

FIFO Illustrating Belady's Anomaly



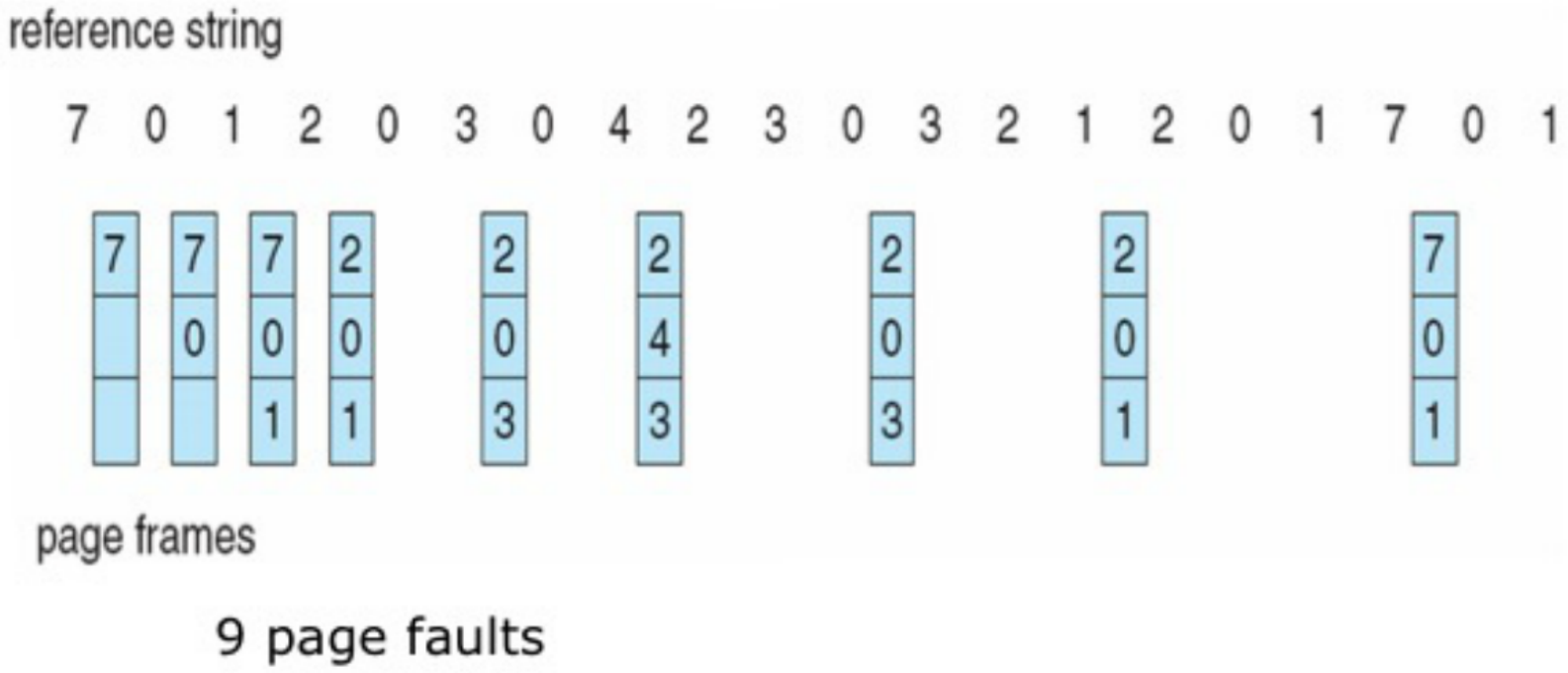
Optimal Algorithm

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms (called OPT or MIN). It is simply this:
  - Replace the page that will not be used for longest period of time
- 4 frames example : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- How do you know this?
- Used for measuring how well your algorithm performs

Optimal Page Replacement



9 page faults

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. (We encountered a similar situation with the SJF CPU-scheduling algorithm). As a result, the optimal algorithm is used mainly for comparison studies.

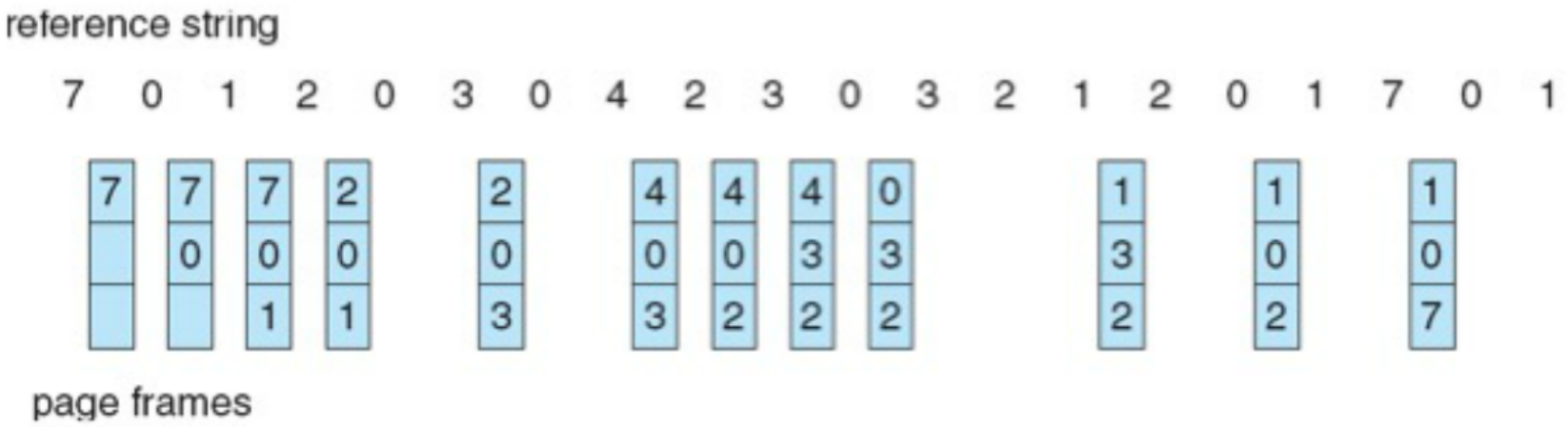
Least Recently Used (LRU) Algorithm

- replace the page that has not been used for the longest period of time
- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**

1	1	1	1	**5**
2	2	2	2	2
3	**5**	5	**4**	4
4	4	**3**	3	3

- Counter implementation
  - Every page entry has a time-of-use field; every time page is referenced, copy the CPU clock/counter into the time-of-use field
  - When a page needs to be replaced, look at the time-of-use field values to determine which page should be replaced

LRU Page Replacement



12 page faults

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **The least frequently used (LFU) Algorithm**: replaces page with smallest count
- **The most frequently used (MFU) Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

Second Chance Algorithm

- A modified form of the FIFO
  - better than FIFO at little cost for the improvement.
- It works by looking at the front of the queue as FIFO does, but instead of immediately paging out that page, it checks to see if its referenced bit is set. If it is not set, the page is swapped out.
- This can also be thought of as a circular queue.
- If all the pages have their reference bit set then second chance algorithm degenerates into pure FIFO.

Second Chance Algorithm

	Frame	ref	valid
0	01	0 1/0	1
1	10	0	1/0
2	00	0	0/1
3	10	0	0/1
4		0	0
5		0	0
6		0	0
7	00	0	1/ 0

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2														
	0	0	0	0	0														
		1	1	1	3														

Other Considerations - Pre-Paging

Pre-paging / Anticipatory Paging

- To reduce the large number of page faults that occurs at the process startup
- Pre-page (anticipate) all or some of the pages a process will need, before they are referenced
- But if pre-paged pages are unused I/O and memory was wasted

Conclusion

- Virtual memory is commonly implemented by demand paging.
- Demand paging is used to reduce the number of frames allocated to a process.
- We need both page-replacement and frame-allocation algorithms.

Assignments

Generate random input, test (*implement*) on
- FIFO
- LRU

Compare with Optimal Page Replacement and *implement* Second Chance Algorithm