

Chapter 10: Virtual Memory

- Background
- Demand Paging
- Process Creation
- Page Replacement
- Allocation of Frames
- Thrashing
- Operating System Examples





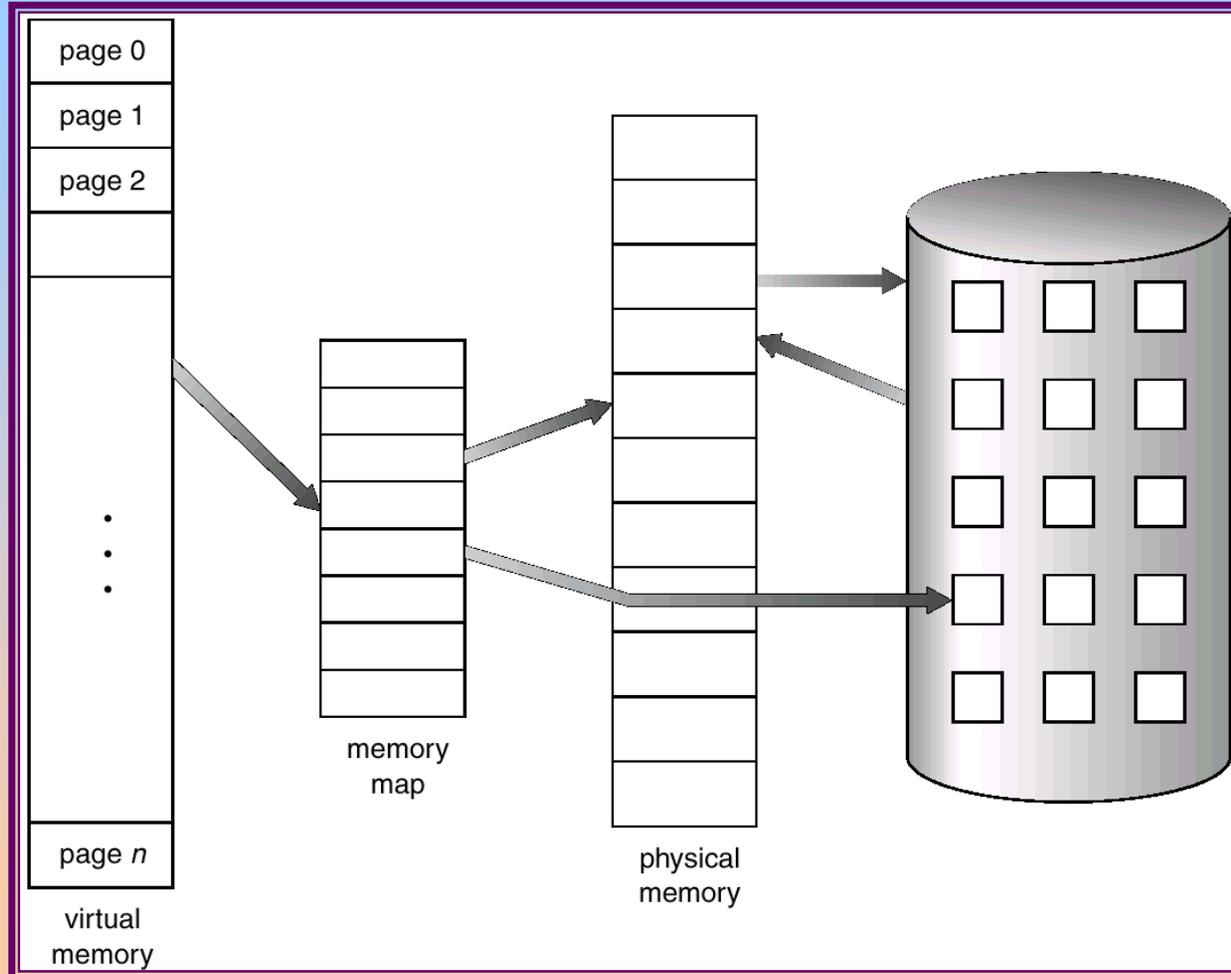
Background

- **Virtual memory** – 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 That is Larger Than Physical Memory





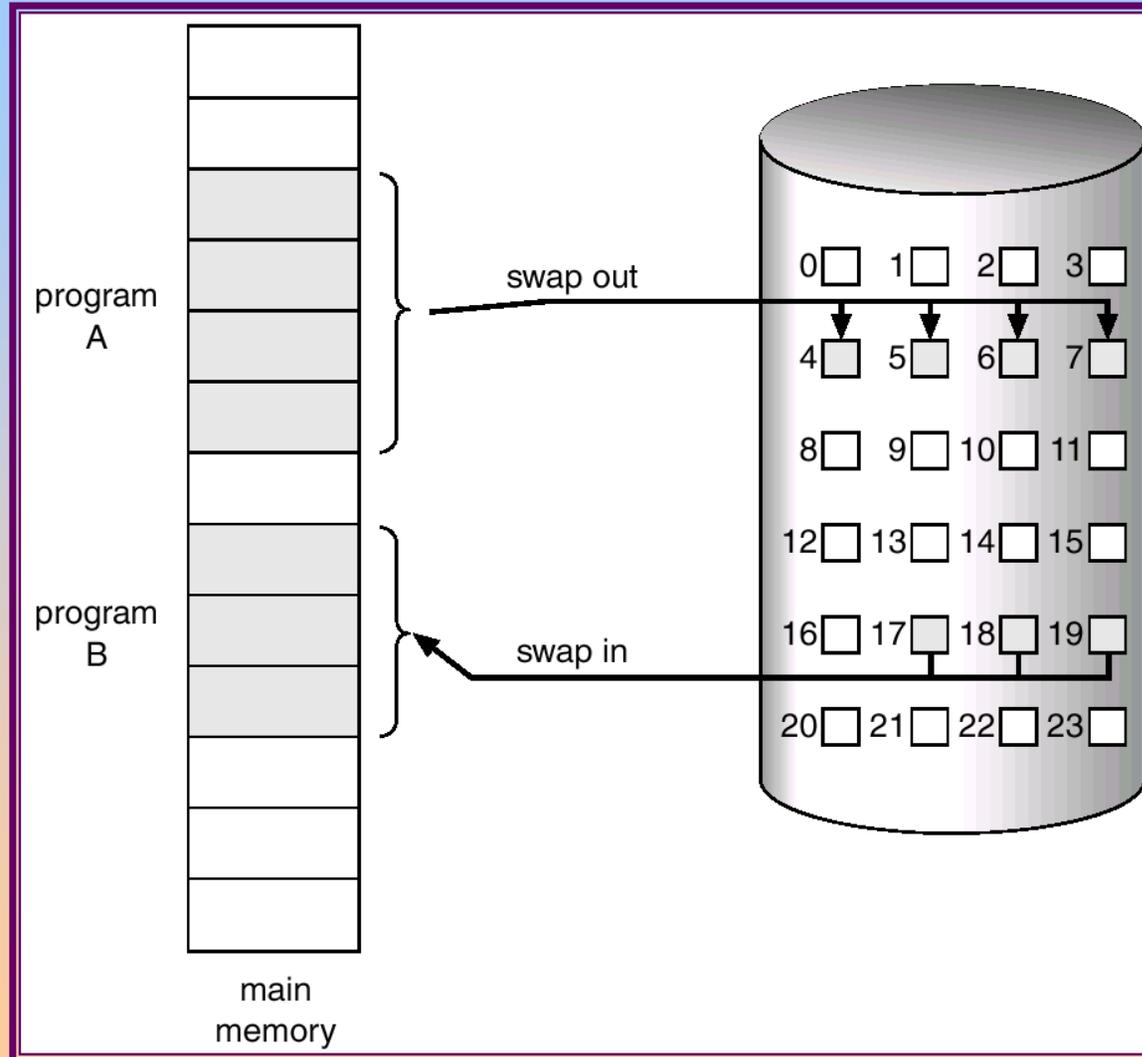
Demand Paging

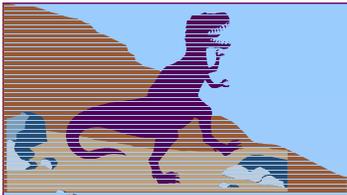
- Bring a page into memory only when it is needed.
 - ◆ Less I/O needed
 - ◆ Less memory needed
 - ◆ Faster response
 - ◆ More users

- Page is needed \Rightarrow reference to it
 - ◆ invalid reference \Rightarrow abort
 - ◆ not-in-memory \Rightarrow bring to memory



Transfer of a Paged Memory to Contiguous Disk Space





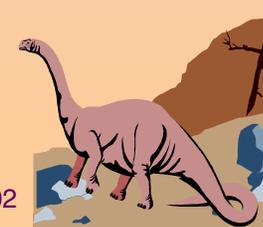
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
(1 \Rightarrow in-memory, 0 \Rightarrow not-in-memory)
- Initially valid–invalid but is set to 0 on all entries.
- Example of a page table snapshot.

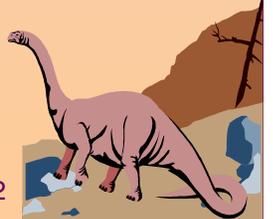
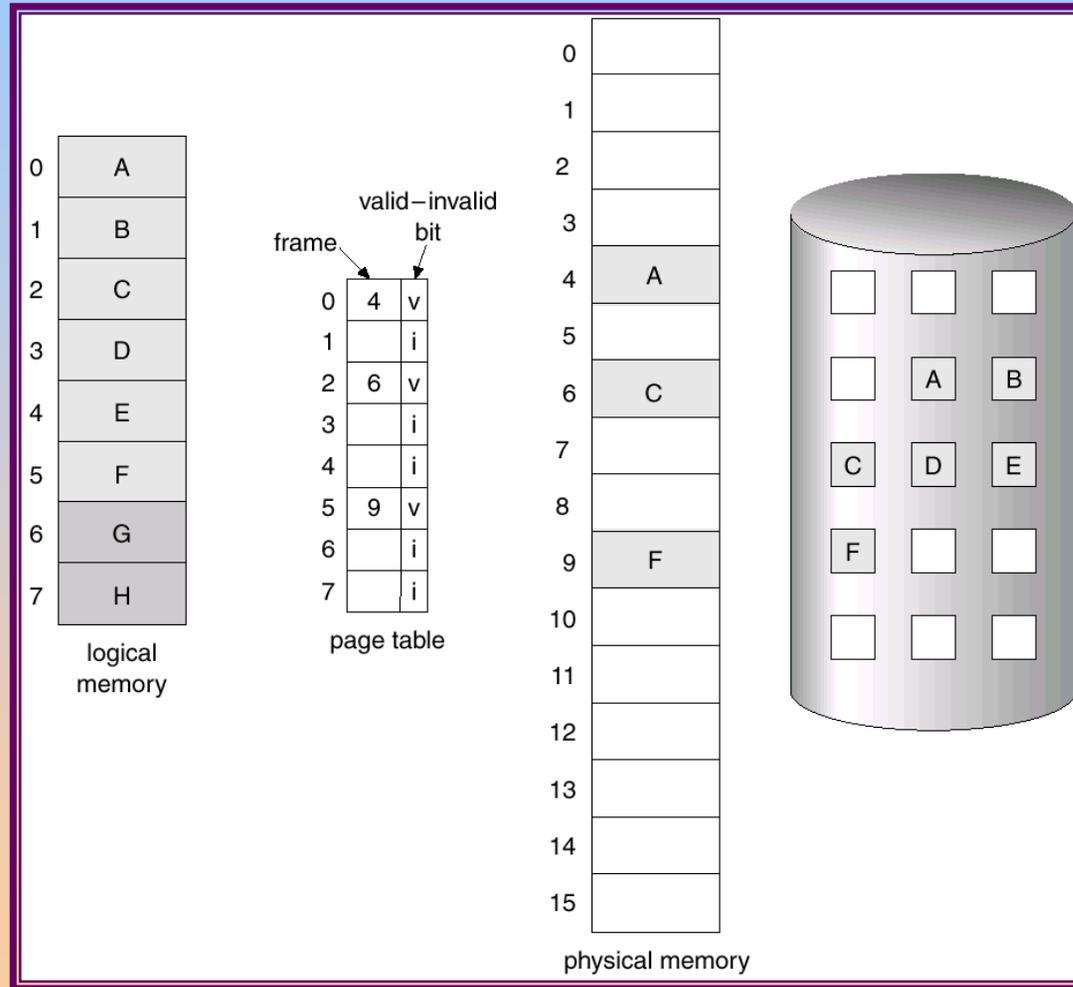
Frame #	valid-invalid bit
	1
	1
	1
	1
	0
⋮	
	0
	0

page table

- During address translation, if valid–invalid bit in page table entry is 0 \Rightarrow page fault.

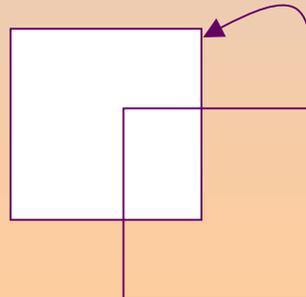


Page Table When Some Pages Are Not in Main Memory



Page Fault

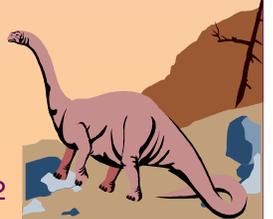
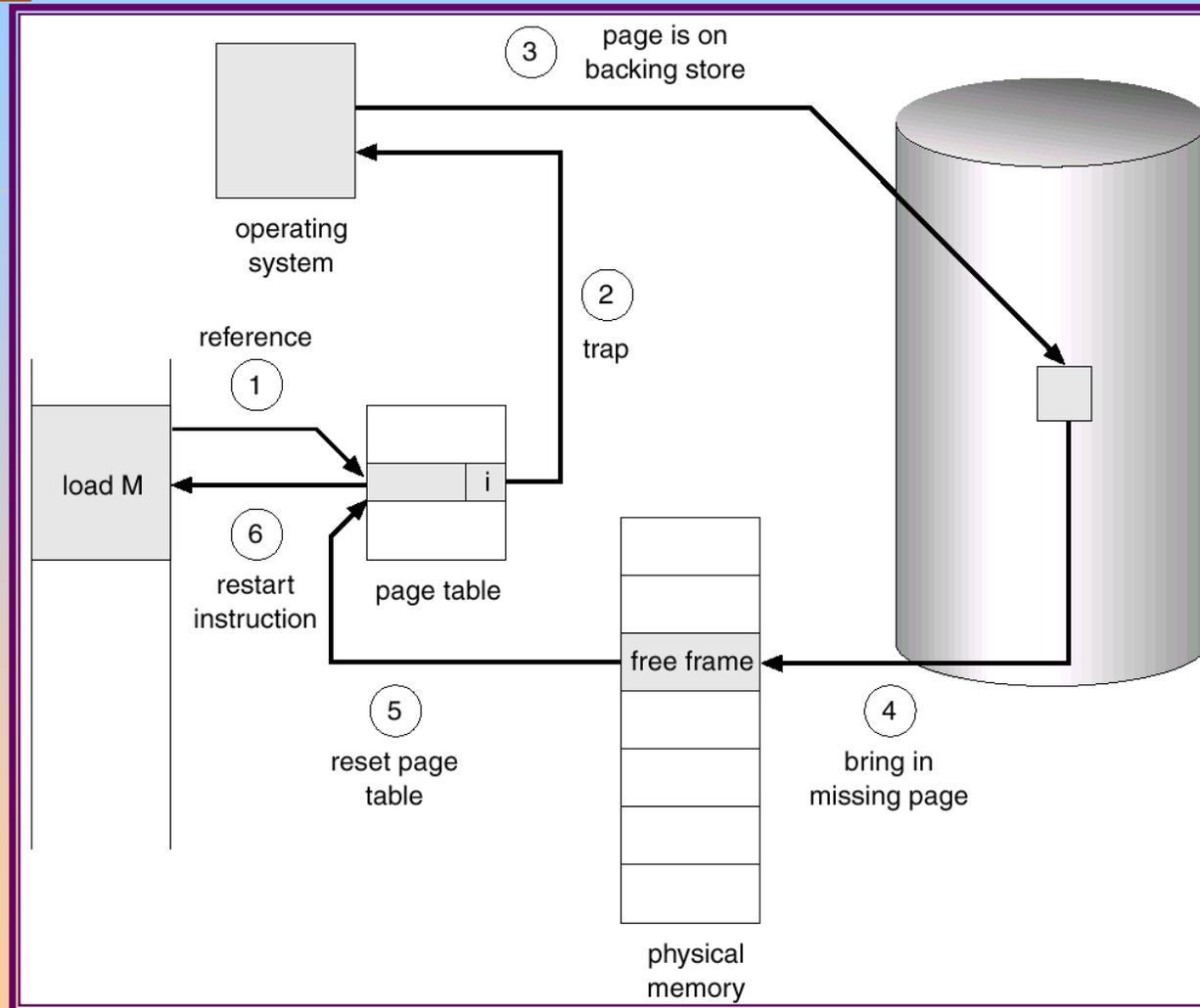
- If there is ever a reference to a page, first reference will trap to OS \Rightarrow page fault
- OS looks at another table to decide:
 - ◆ Invalid reference \Rightarrow abort.
 - ◆ Just not in memory.
- Get empty frame.
- Swap page into frame.
- Reset tables, validation bit = 1.
- Restart instruction: Least Recently Used
 - ◆ block move



- ◆ auto increment/decrement location



Steps in Handling a Page Fault





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.





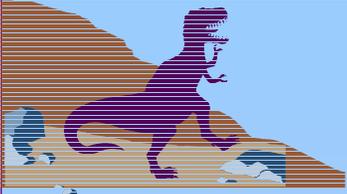
Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1.0$
 - ◆ if $p = 0$ no page faults
 - ◆ if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

$$\begin{aligned} \text{EAT} = & (1 - p) \times \text{memory access} \\ & + p (\text{page fault overhead} \\ & + [\text{swap page out}] \\ & + \text{swap page in} \\ & + \text{restart overhead}) \end{aligned}$$



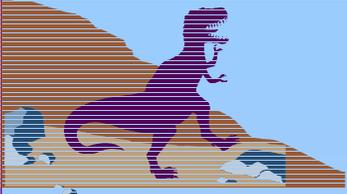


Demand Paging Example

- Memory access time = 1 microsecond
- 50% of the time the page that is being replaced has been modified and therefore needs to be swapped out.
- Swap Page Time = 10 msec = 10,000 msec

$$\text{EAT} = (1 - p) \times 1 + p (15000)$$
$$1 + 15000P \quad (\text{in msec})$$





Process Creation

- Virtual memory allows other benefits during process creation:
 - Copy-on-Write
 - Memory-Mapped Files





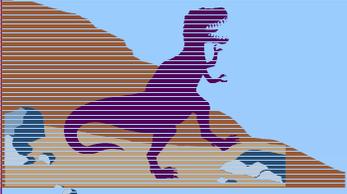
Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory.

If either process modifies a shared page, only then is the page copied.

- COW allows more efficient process creation as only modified pages are copied.
- Free pages are allocated from a *pool* of zeroed-out pages.

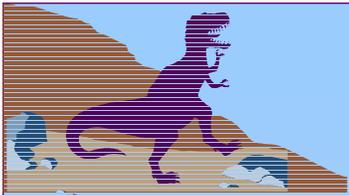




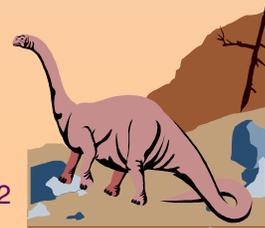
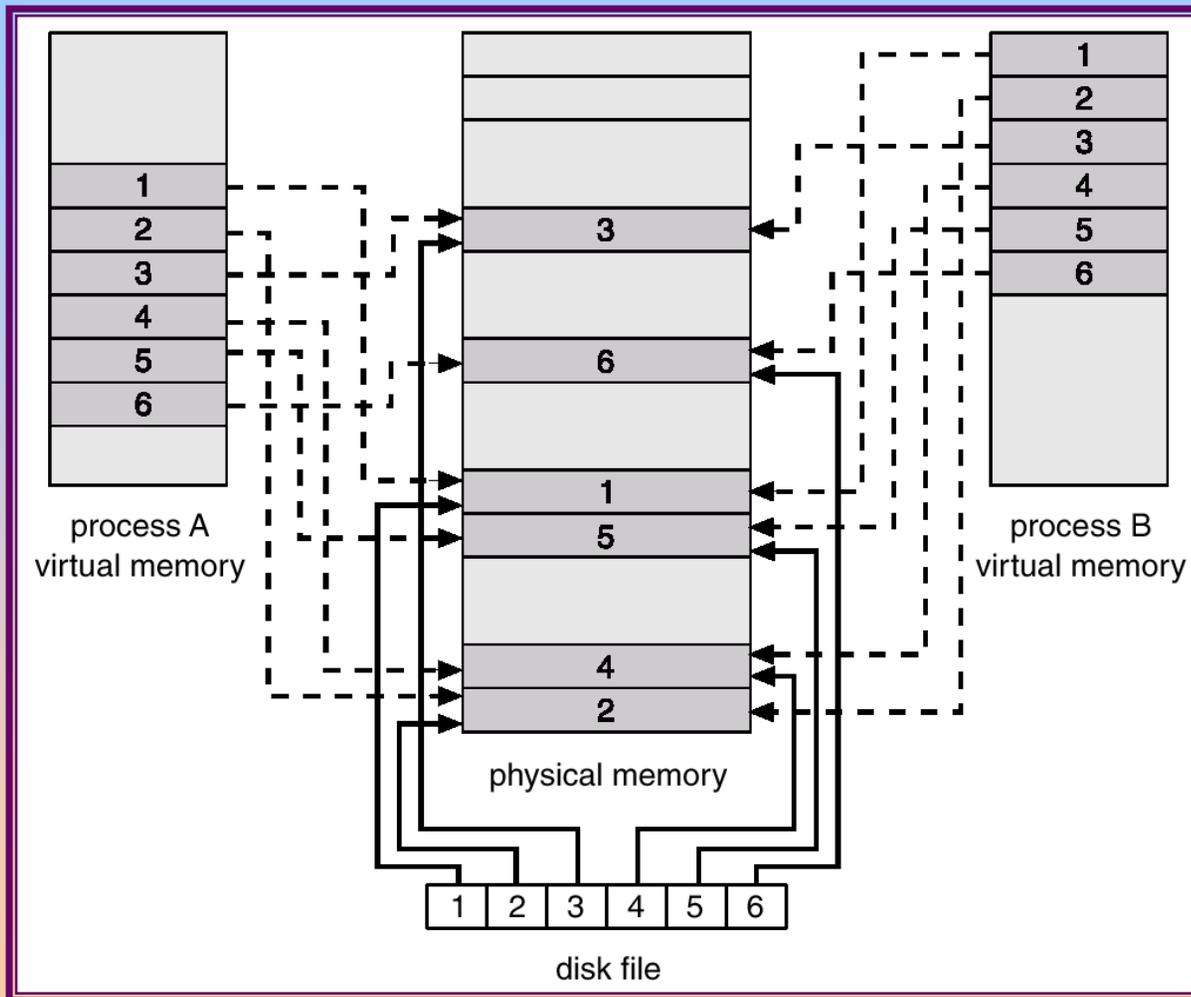
Memory-Mapped Files

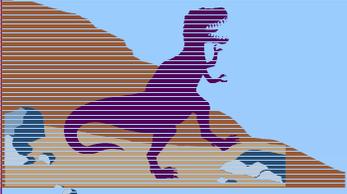
- Memory-mapped file I/O allows file I/O to be treated as routine memory access by *mapping* a disk block to a page in memory.
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **read()** **write()** system calls.
- Also allows several processes to map the same file allowing the pages in memory to be shared.





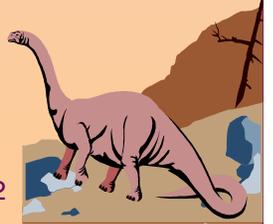
Memory Mapped Files



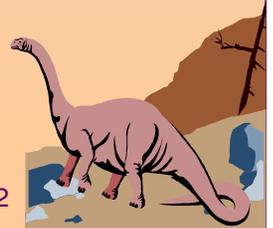
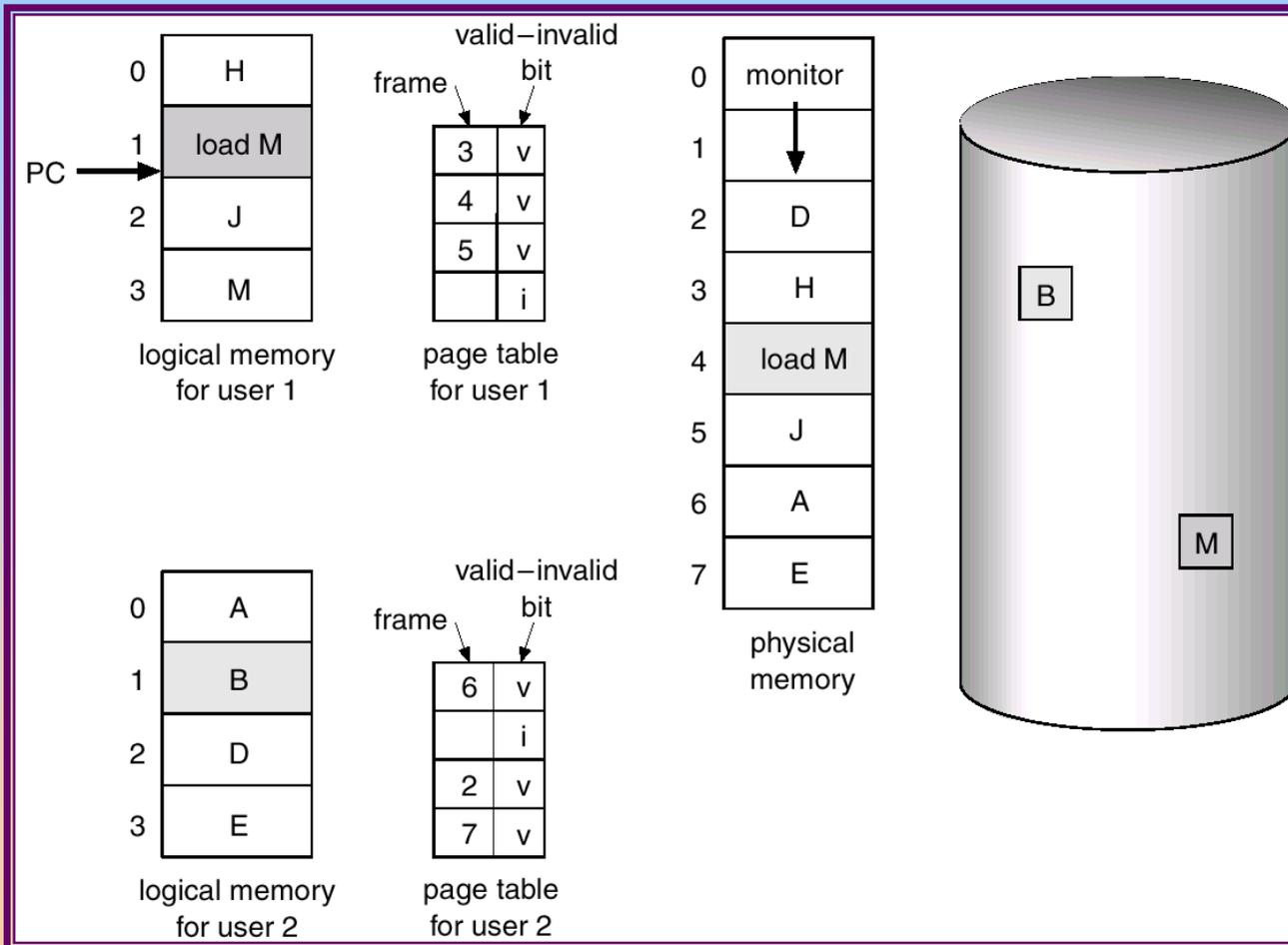


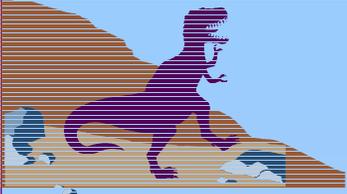
Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement.
- Use *modify (dirty) bit* to reduce overhead of page transfers – only modified pages are written to disk.
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory.



Need For Page Replacement



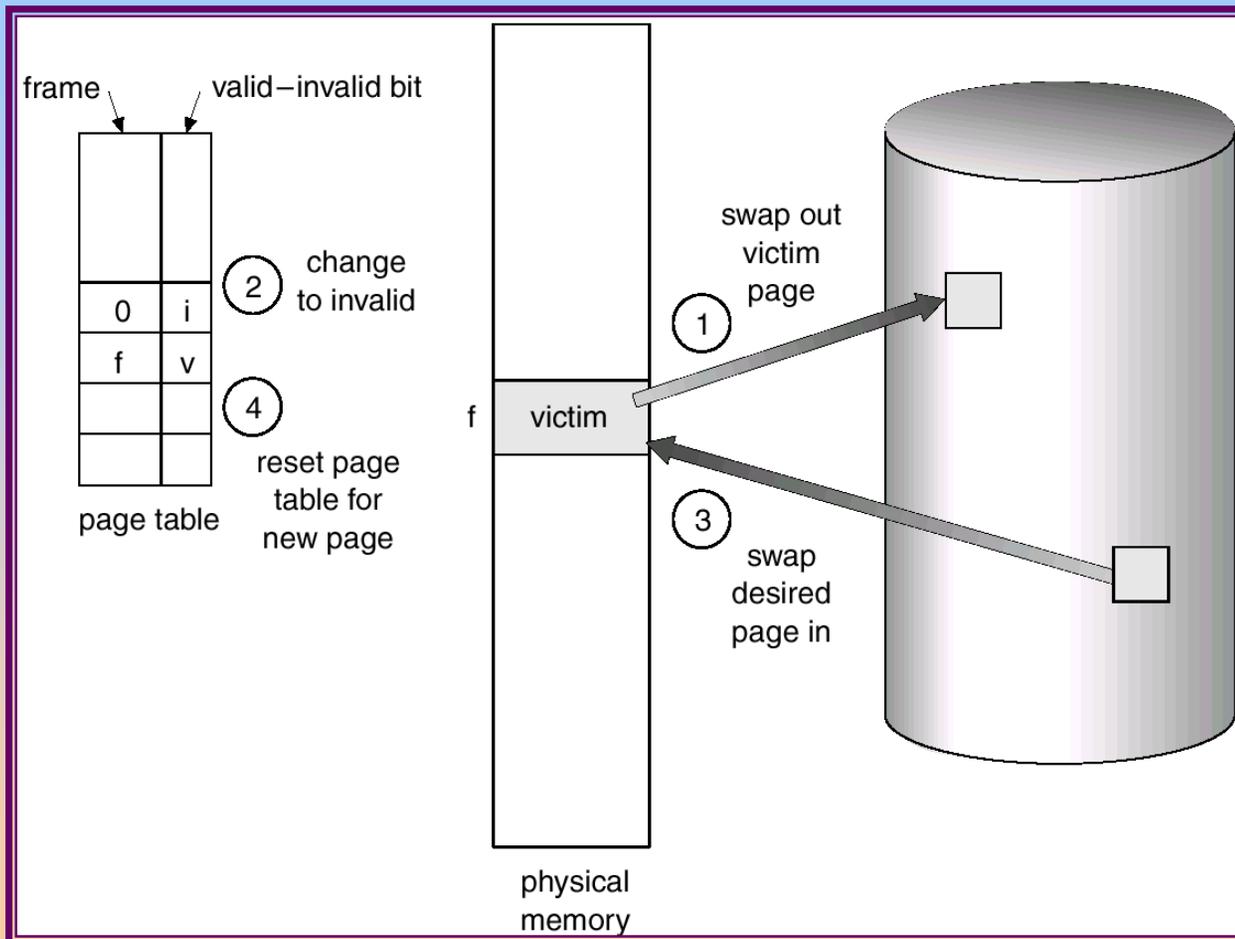


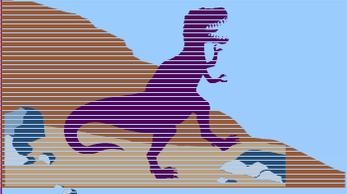
Basic Page Replacement

- Find the location of the desired page on disk.
- 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.
- Read the desired page into the (newly) free frame. Update the page and frame tables.
- Restart the process.



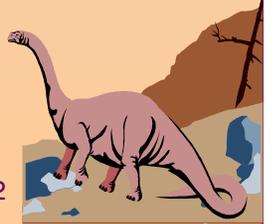
Page Replacement





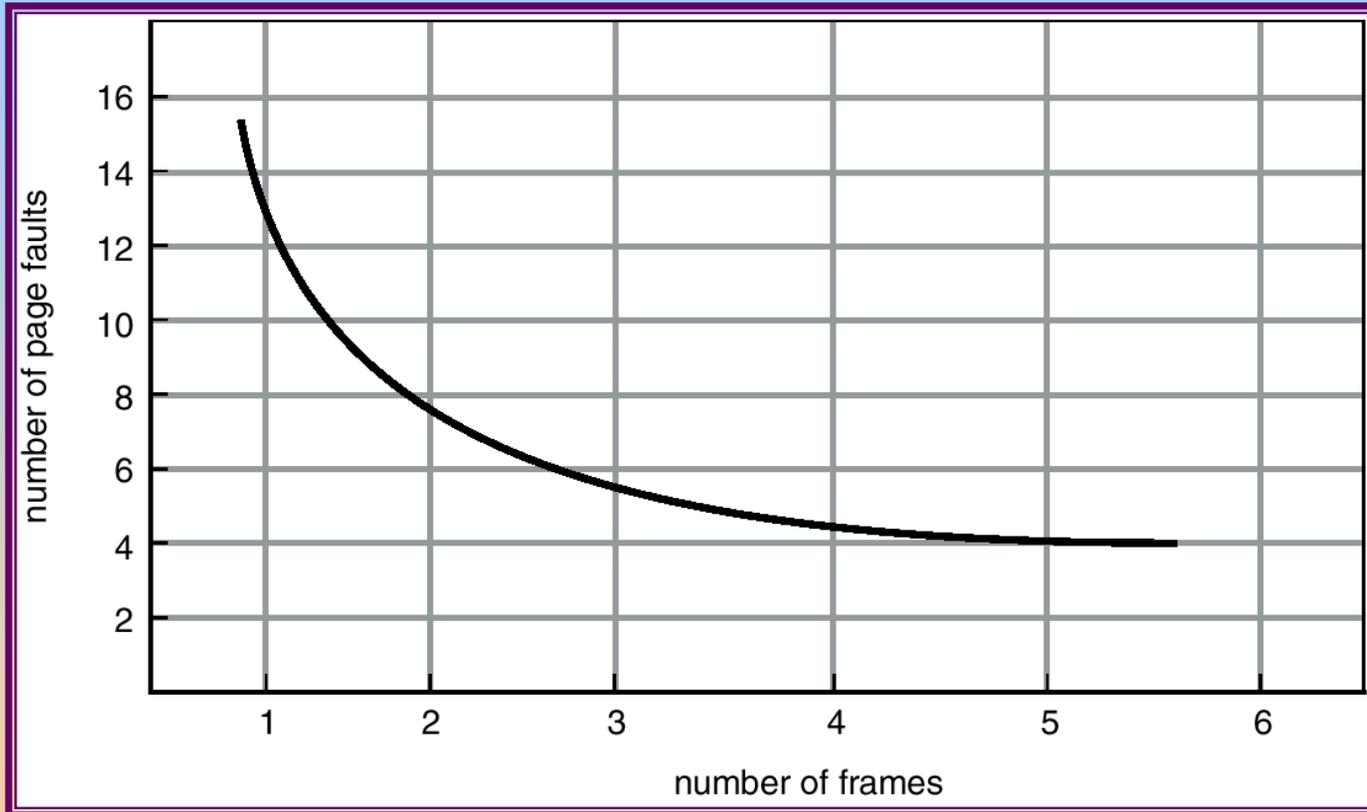
Page Replacement Algorithms

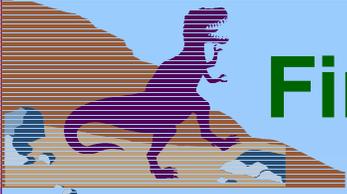
- Want lowest page-fault rate.
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.
- In all our examples, the reference string is
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.





Graph of Page Faults Versus The Number of Frames





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)

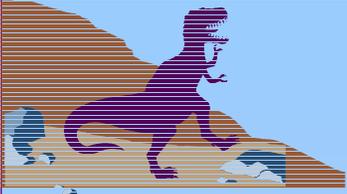
1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- 4 frames

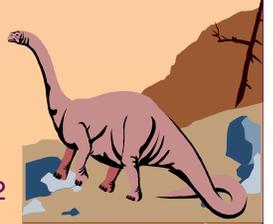
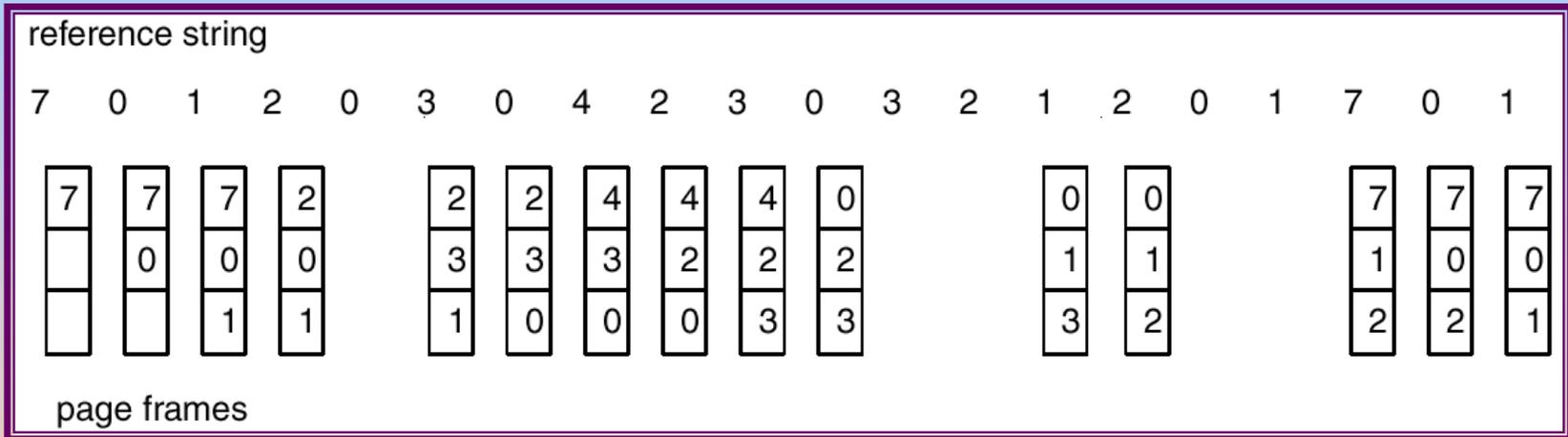
1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

- FIFO Replacement – Belady’s Anomaly
 - ◆ more frames \Rightarrow less page faults



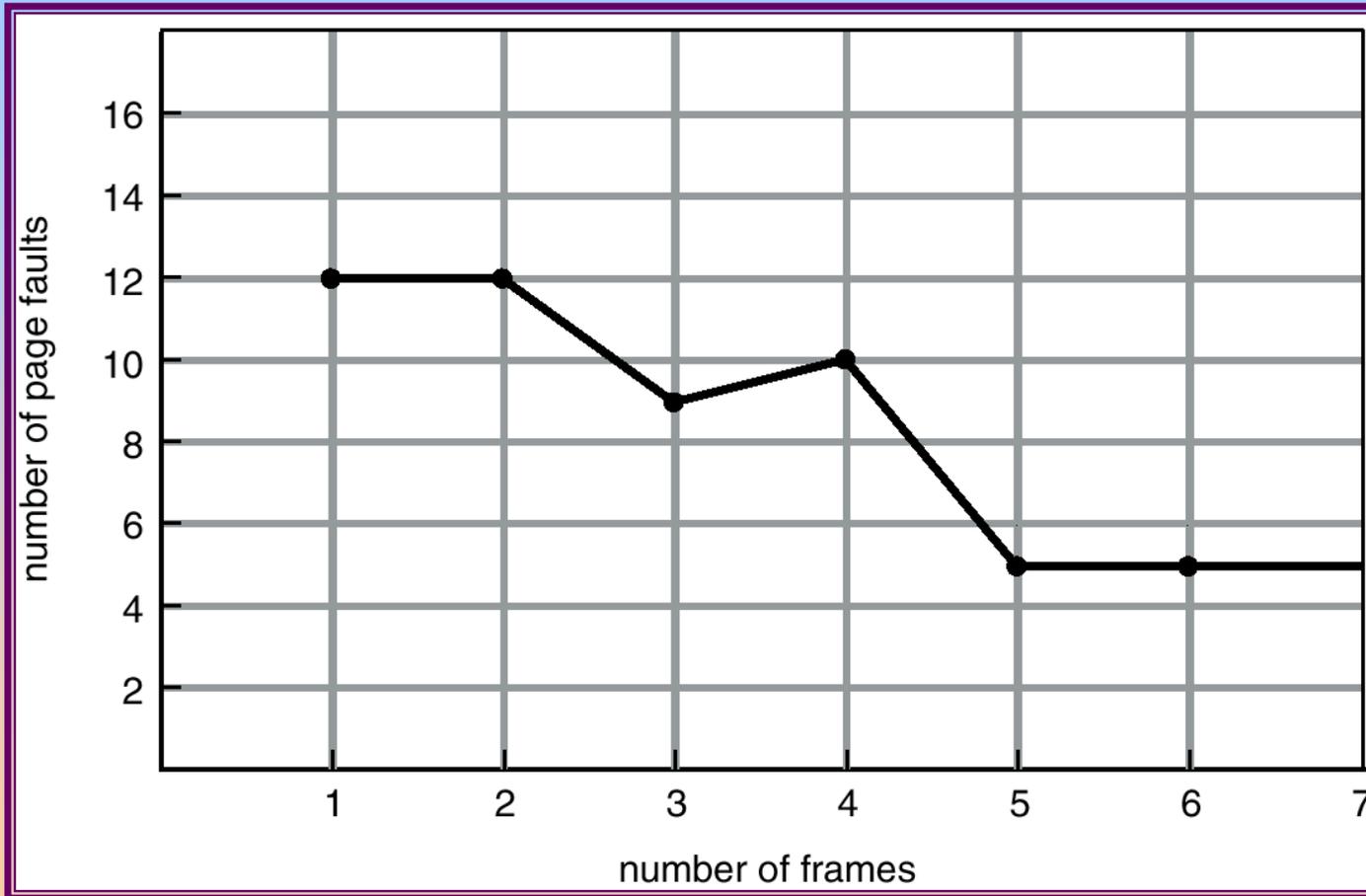


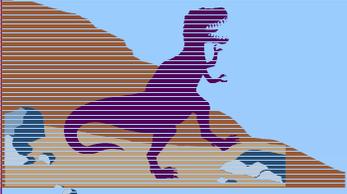
FIFO Page Replacement





FIFO Illustrating Belady's Anamoly

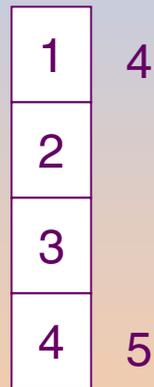




Optimal Algorithm

- Replace page that will not be used for longest period of time.
- 4 frames example

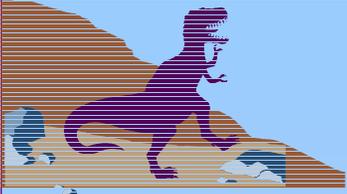
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



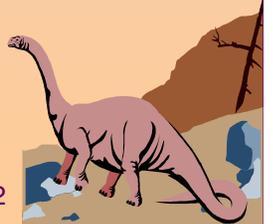
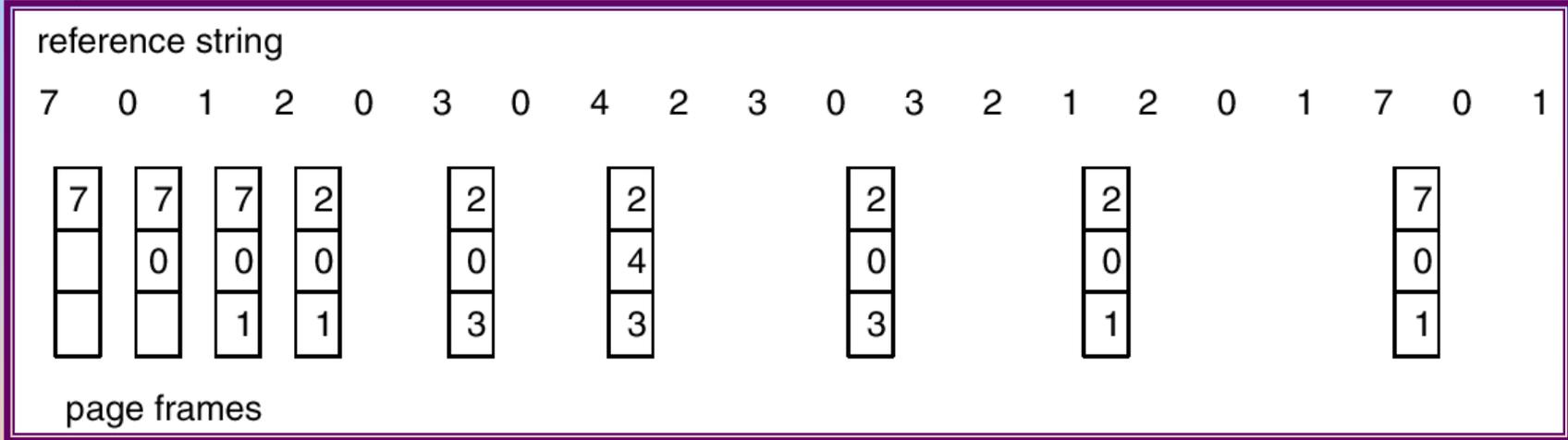
6 page faults

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





Optimal Page Replacement





Least Recently Used (LRU) Algorithm

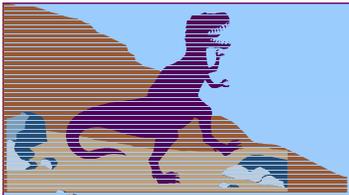
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	5
2	
3	5 4
4	3

- Counter implementation

- ◆ Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
- ◆ When a page needs to be changed, look at the counters to determine which are to change.

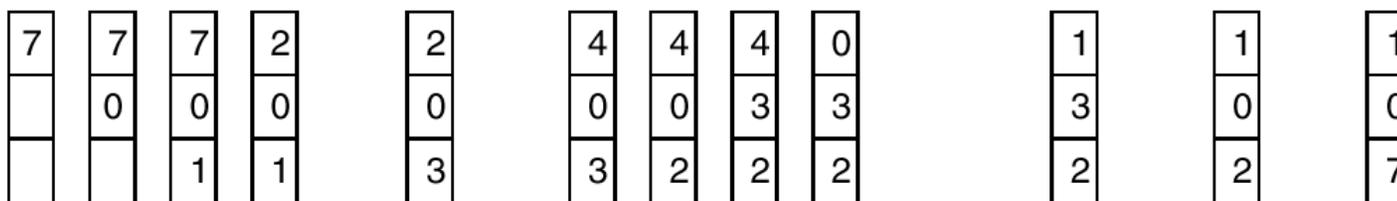




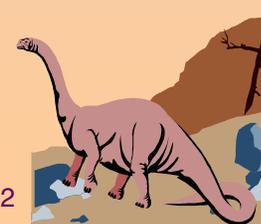
LRU Page Replacement

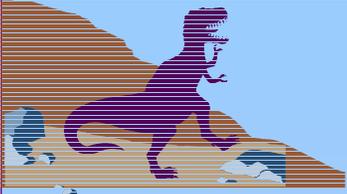
reference string

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



page frames





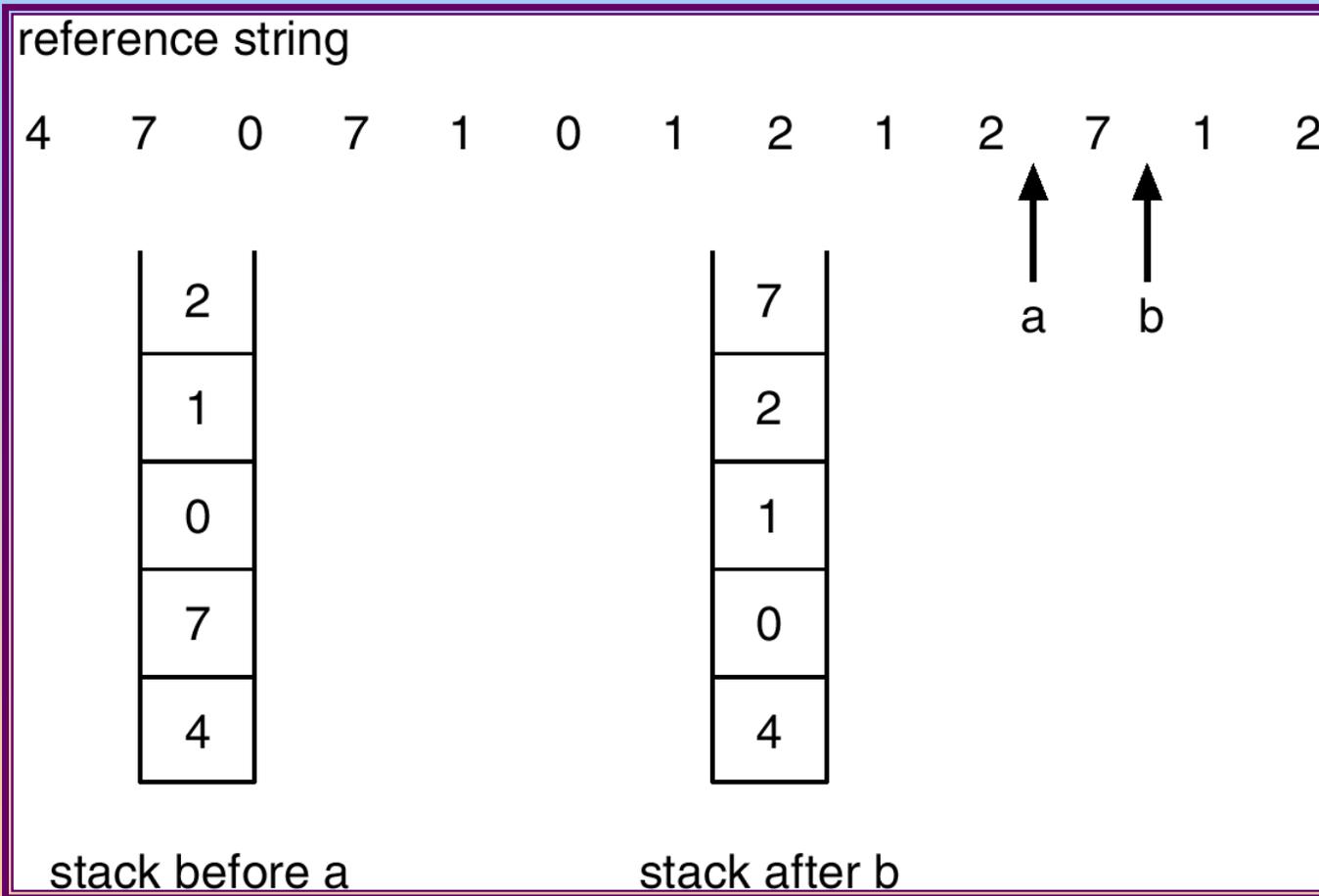
LRU Algorithm (Cont.)

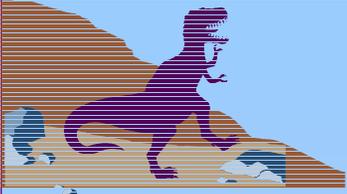
- Stack implementation – keep a stack of page numbers in a double link form:
 - ◆ Page referenced:
 - ✓ move it to the top
 - ✓ requires 6 pointers to be changed
 - ◆ No search for replacement





Use Of A Stack To Record The Most Recent Page References





LRU Approximation Algorithms

■ Reference bit

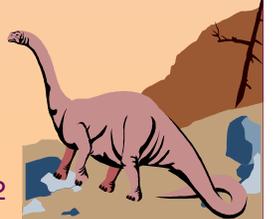
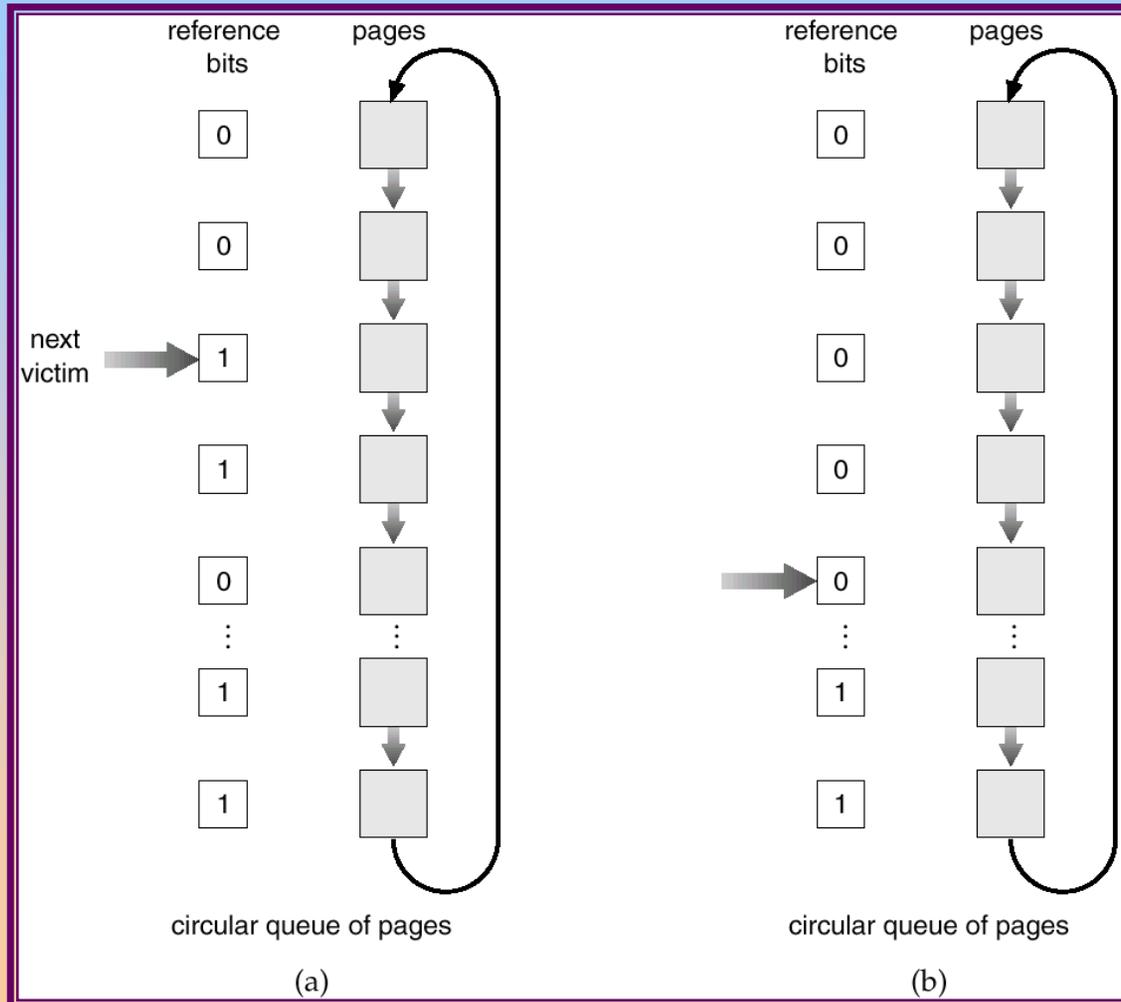
- ◆ With each page associate a bit, initially = 0
- ◆ When page is referenced bit set to 1.
- ◆ Replace the one which is 0 (if one exists). We do not know the order, however.

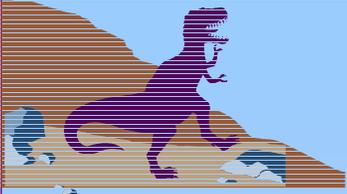
■ Second chance

- ◆ Need reference bit.
- ◆ Clock replacement.
- ◆ If page to be replaced (in clock order) has reference bit = 1. then:
 - ✓ set reference bit 0.
 - ✓ leave page in memory.
 - ✓ replace next page (in clock order), subject to same rules.



Second-Chance (clock) Page-Replacement Algorithm

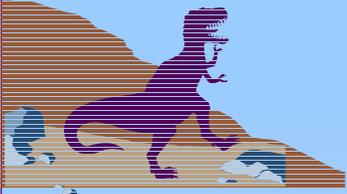




Counting Algorithms

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

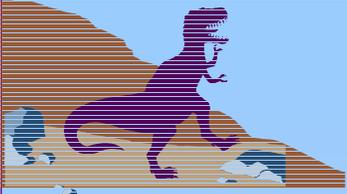




Allocation of Frames

- Each process needs **minimum** number of pages.
- Example: IBM 370 – 6 pages to handle SS 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

- Equal allocation – e.g., if 100 frames and 5 processes, give each 20 pages.
- Proportional allocation – Allocate according to the size of process.

– s_i = size of process p_i

– $S = \sum s_i$

– m = total number of frames

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

$$m = 64$$

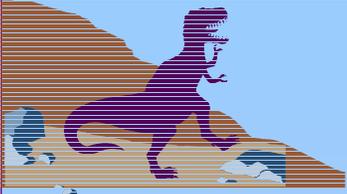
$$s_1 = 10$$

$$s_2 = 127$$

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

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



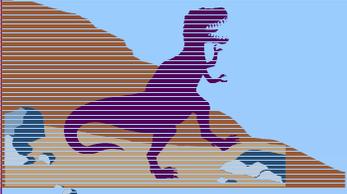


Priority Allocation

- 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.

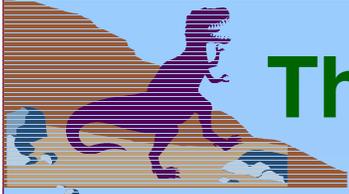




Global vs. Local Allocation

- **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.





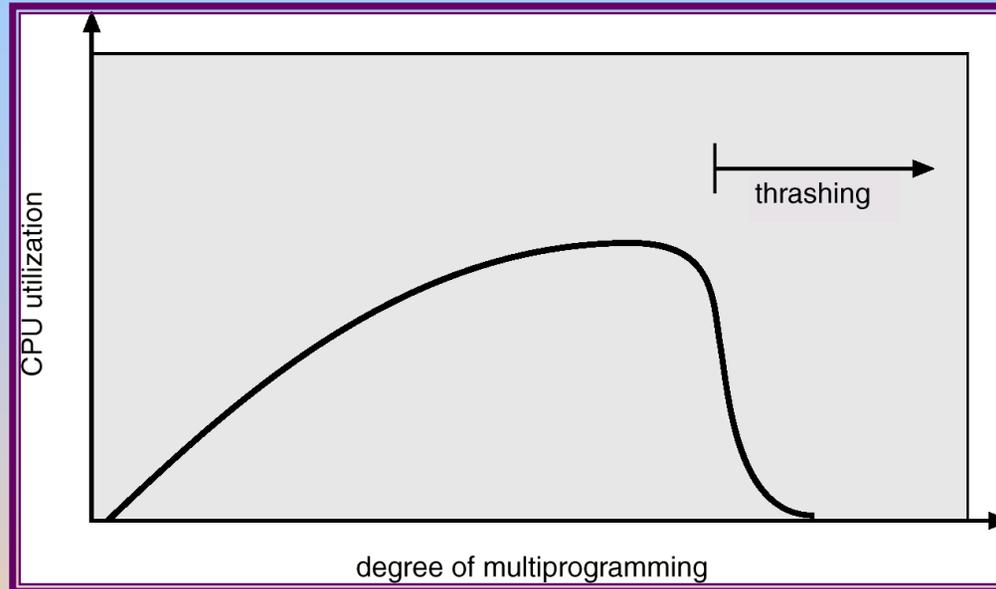
Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. 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** \equiv a process is busy swapping pages in and out.

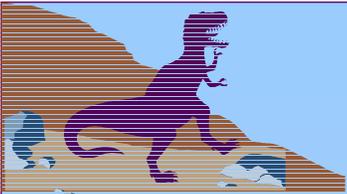


Thrashing

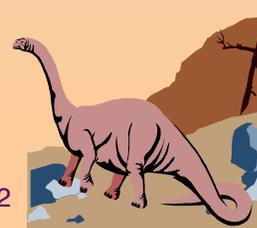
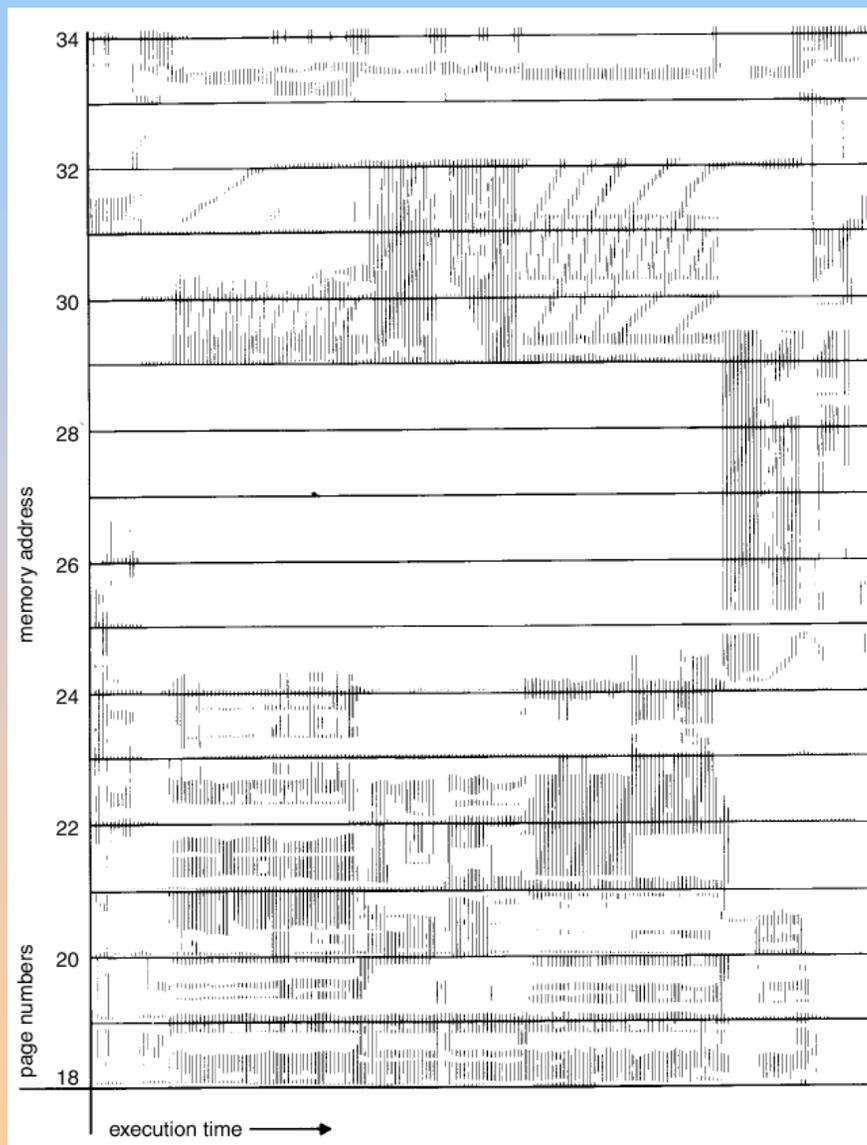


- Why does paging work?
Locality model
 - ◆ Process migrates from one locality to another.
 - ◆ Localities may overlap.
- Why does thrashing occur?
 Σ size of locality > total memory size





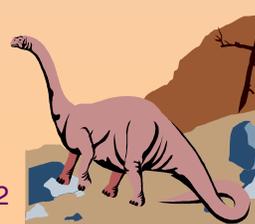
Locality In A Memory-Reference Pattern

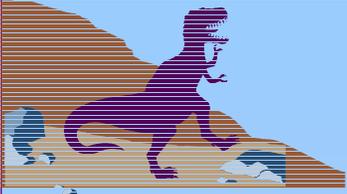




Working-Set Model

- $\Delta \equiv$ working-set window \equiv 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 Δ
(varies in time)
 - ◆ if Δ too small will not encompass entire locality.
 - ◆ if Δ too large will encompass several localities.
 - ◆ if $\Delta = \infty \Rightarrow$ will encompass entire program.
- $D = \sum WSS_i \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend one of the processes.

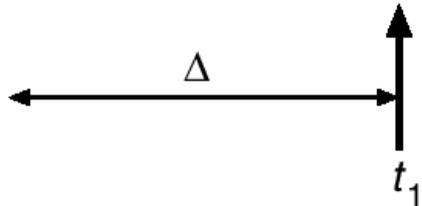




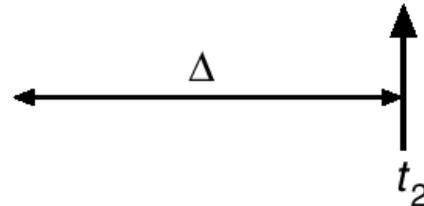
Working-set model

page reference table

... 2 6 1 5 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...

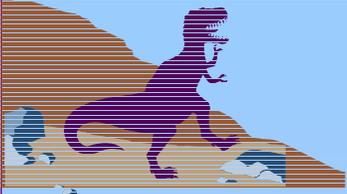


$$WS(t_1) = \{1, 2, 5, 6, 7\}$$



$$WS(t_2) = \{3, 4\}$$



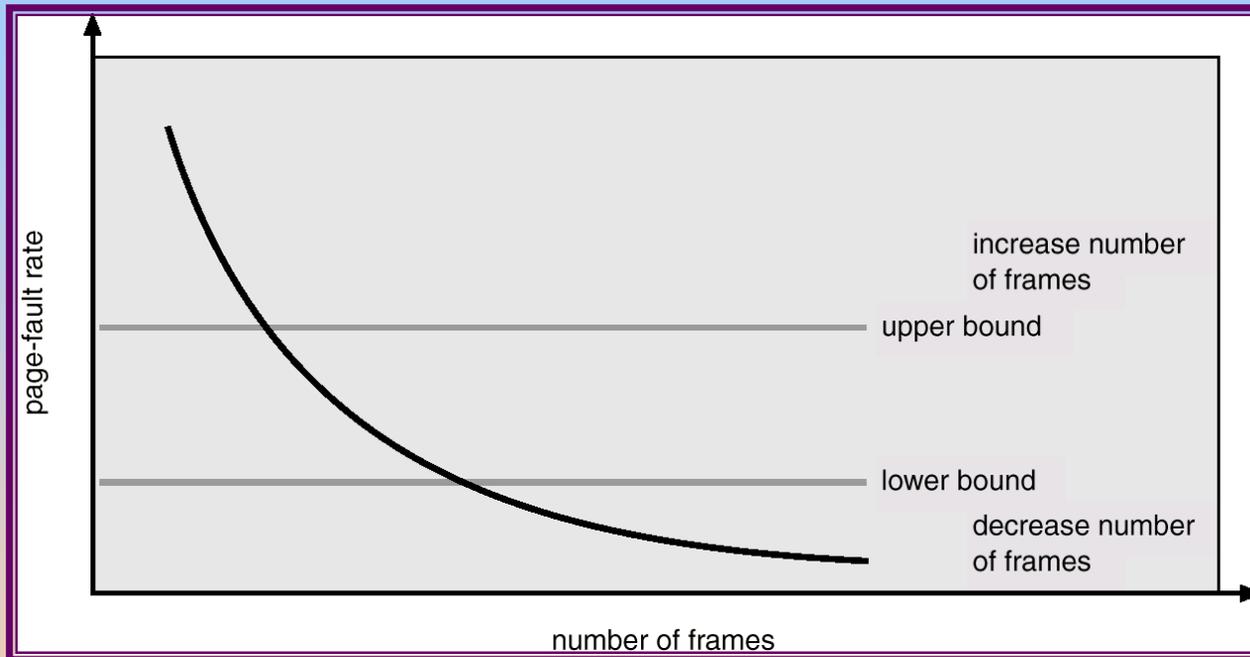


Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
 - ◆ Timer interrupts after every 5000 time units.
 - ◆ Keep in memory 2 bits for each page.
 - ◆ Whenever a timer interrupts copy and sets the values of all reference bits to 0.
 - ◆ If one of the bits in memory = 1 \Rightarrow page in working set.
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units.

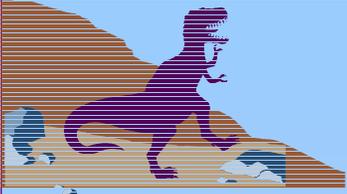


Page-Fault Frequency Scheme



- Establish “acceptable” page-fault rate.
 - ◆ If actual rate too low, process loses frame.
 - ◆ If actual rate too high, process gains frame.





Other Considerations

- Prepaging
- Page size selection
 - ◆ fragmentation
 - ◆ table size
 - ◆ I/O overhead
 - ◆ locality





Other Considerations (Cont.)

- **TLB Reach** - The amount of memory accessible from the TLB.
- $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
- Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of page faults.





Increasing the Size of the TLB

- **Increase the Page Size.** This may lead to an increase in fragmentation as not all applications require a large page size.
- **Provide Multiple Page Sizes.** This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.





Other Considerations (Cont.)

■ Program structure

◆ `int A[][] = new int[1024][1024];`

◆ Each row is stored in one page

◆ Program 1

```
for (j = 0; j < A.length; j++)  
  for (i = 0; i < A.length; i++)  
    A[i,j] = 0;
```

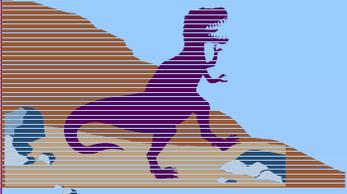
1024 x 1024 page faults

◆ Program 2

```
for (i = 0; i < A.length; i++)  
  for (j = 0; j < A.length; j++)  
    A[i,j] = 0;
```

1024 page faults



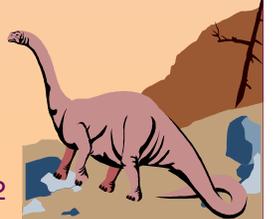
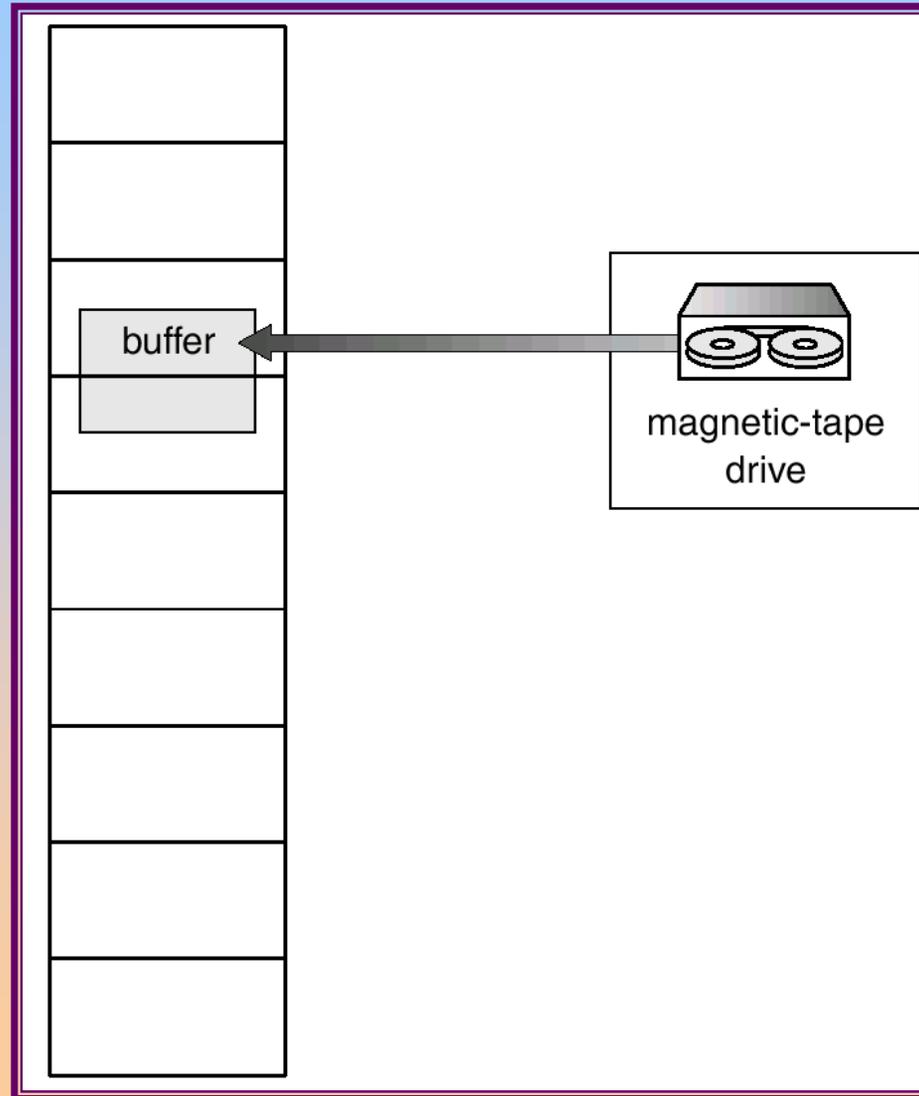


Other Considerations (Cont.)

- **I/O Interlock** – Pages must sometimes be locked into memory.
- Consider I/O. Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.



Reason Why Frames Used For I/O Must Be In Memory





Operating System Examples

- Windows NT

- Solaris 2





Windows NT

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page.
- Processes are assigned **working set minimum** and **working set maximum**.
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory.
- A process may be assigned as many pages up to its working set maximum.
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory.
- Working set trimming removes pages from processes that have pages in excess of their working set minimum.

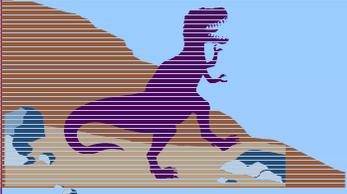




Solaris 2

- Maintains a list of free pages to assign faulting processes.
- **Lotsfree** – threshold parameter to begin paging.
- Paging is performed by *pageout* process.
- Pageout scans pages using modified clock algorithm.
- **Scanrate** is the rate at which pages are scanned. This ranged from **slowscan** to **fastscan**.
- Pageout is called more frequently depending upon the amount of free memory available.





Solar Page Scanner

