Contents

Chapter 9
Virtual-Memory Management

Virtual Memory

- Virtual Memory
  - A technique that allows the execution of a process that may not be completely in memory.
- Motivation:
  - An entire program in execution may not all be needed at the same time!
    - e.g. error handling routines, a large array, certain program features, etc

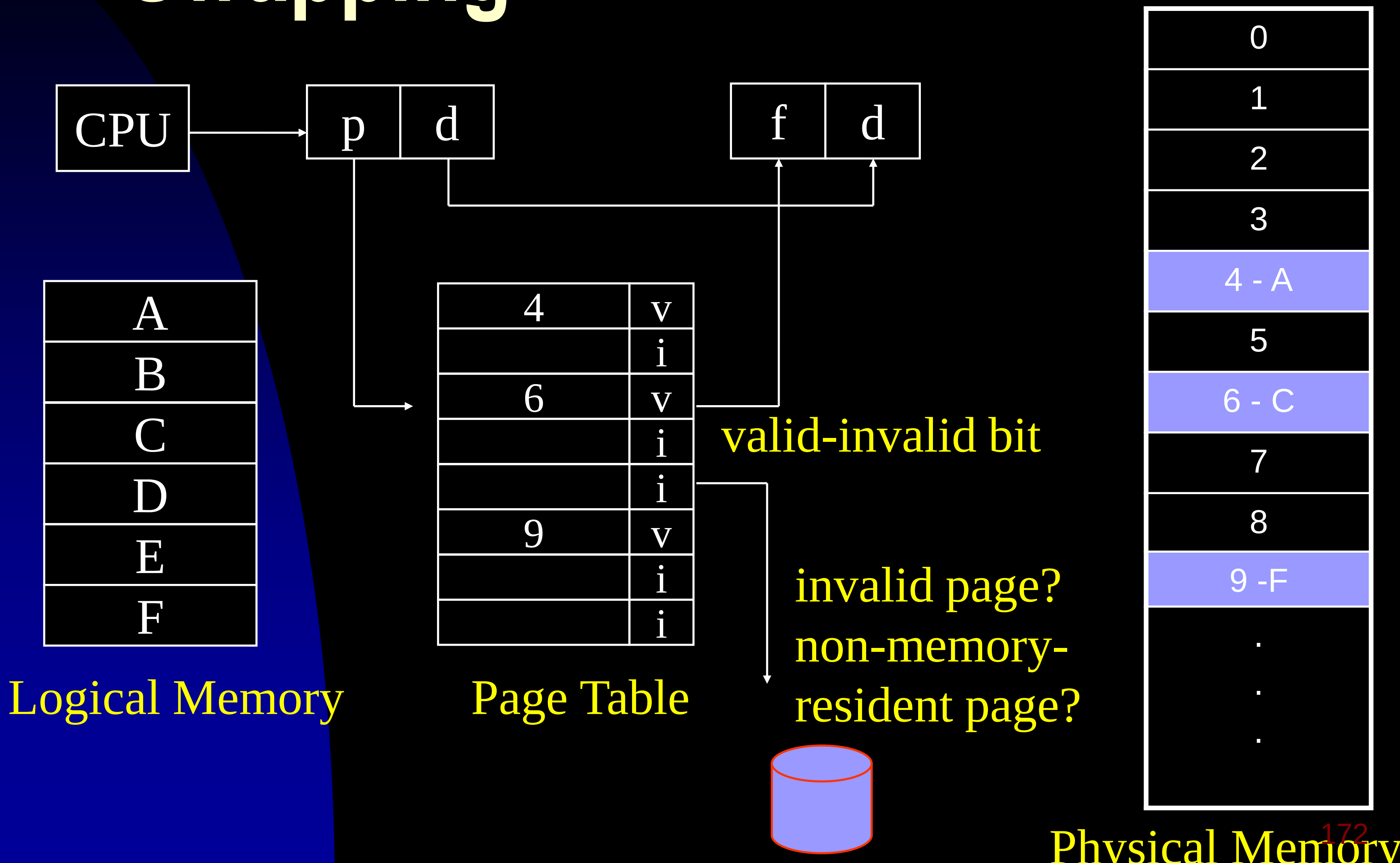
Virtual Memory

- Potential Benefits
  - Programs can be much larger than the amount of physical memory. Users can concentrate on their problem programming.
  - The level of multiprogramming increases because processes occupy less physical memory.
  - Each user program may run faster because less I/O is needed for loading or swapping user programs.
- Implementation: demand paging, demand segmentation (more difficult),etc.

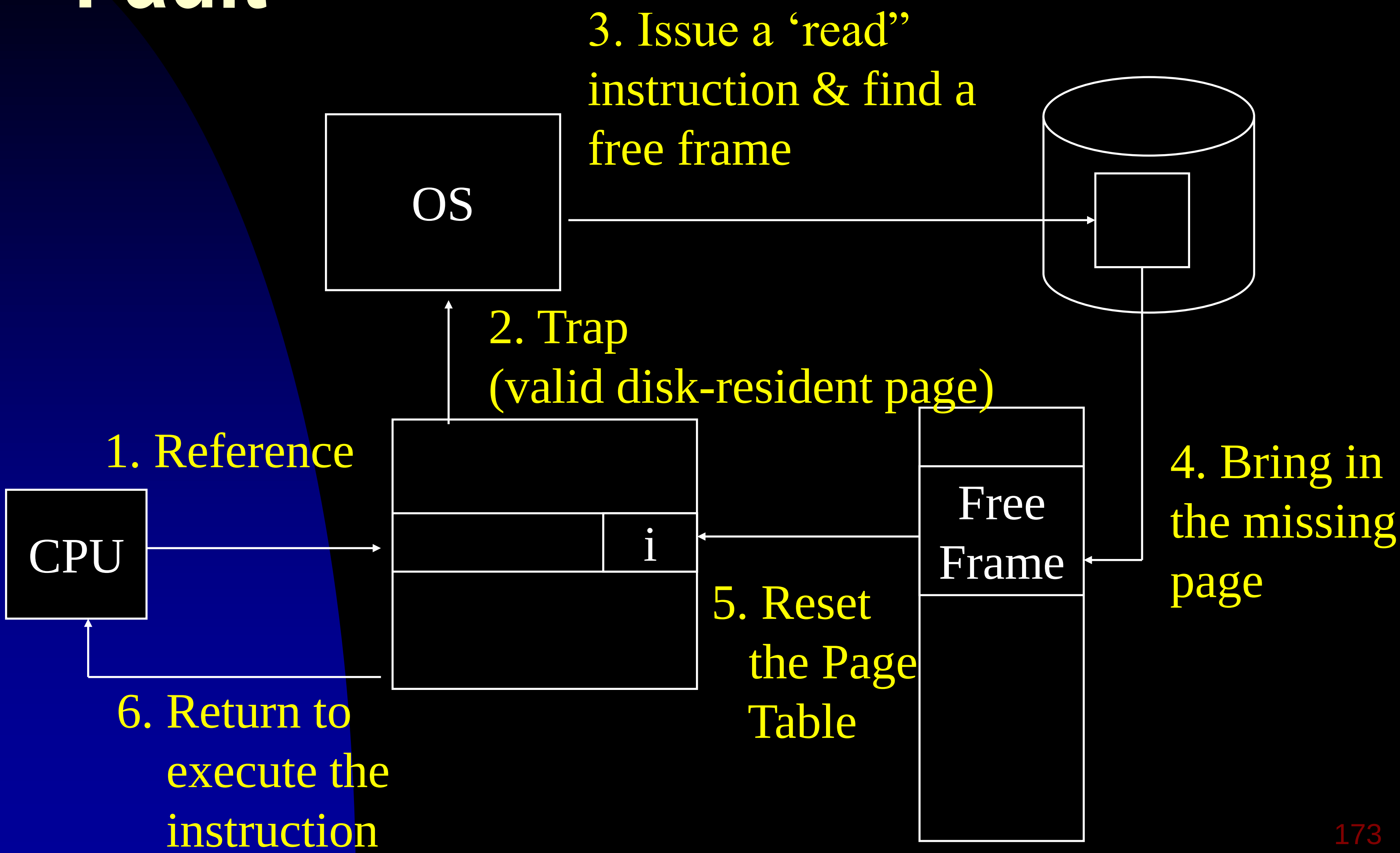
Demand Paging – Lazy Swapping

- Process image may reside on the backing store. Rather than swap in the entire process image into memory, Lazy Swapper only swaps in a page when it is needed!
  - Pure Demand Paging – Pager vs Swapper
  - A Mechanism required to recover from the missing of non-resident referenced pages.
    - A *page fault* occurs when a process references a non-memory-resident page.

Demand Paging – Lazy Swapping



A Procedure to Handle a Page Fault



A Procedure to Handle A Page Fault

- Pure Demand Paging:
  - Never bring in a page into the memory until it is required!
- Pre-Paging
  - Bring into the memory all of the pages that “will” be needed at one time!
  - Locality of reference

Hardware Support for Demand Paging

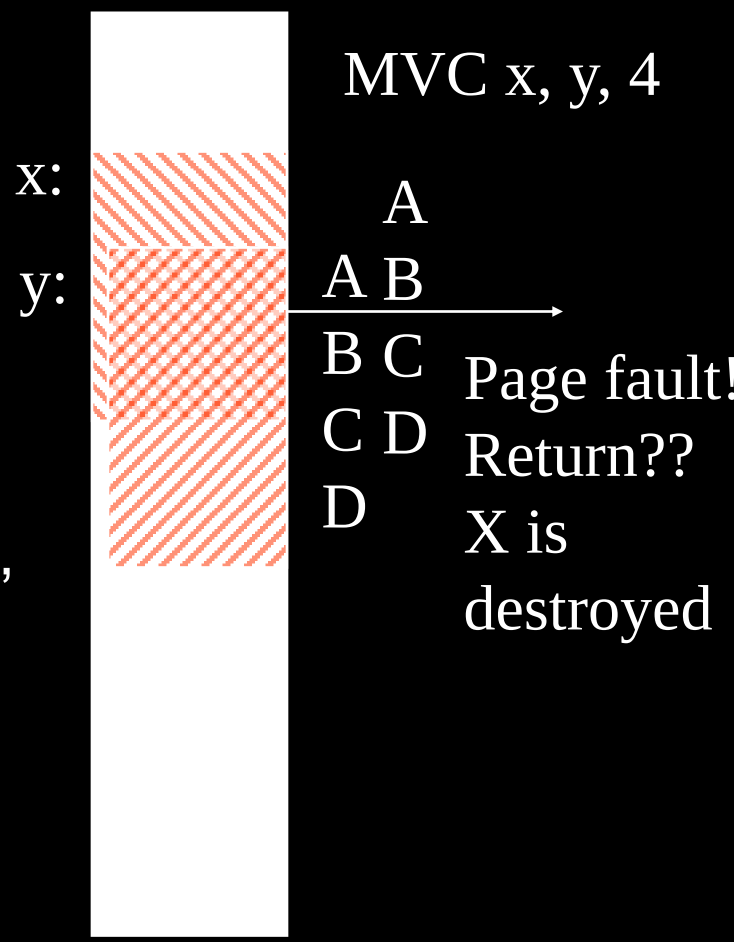
- New Bits in the Page Table
  - To indicate that a page is now in memory or not.
- Secondary Storage
  - Swap space in the backing store
    - A continuous section of space in the secondary storage for better performance.

Crucial issues

- Example 1 – Cost in restarting an instruction
  - Assembly Instruction: Add a, b, c
  - Only a short job!
    - Re-fetch the instruction, decode, fetch operands, execute, save, etc
  - Strategy:
    - Get all pages and restart the instruction from the beginning!

Crucial Issues

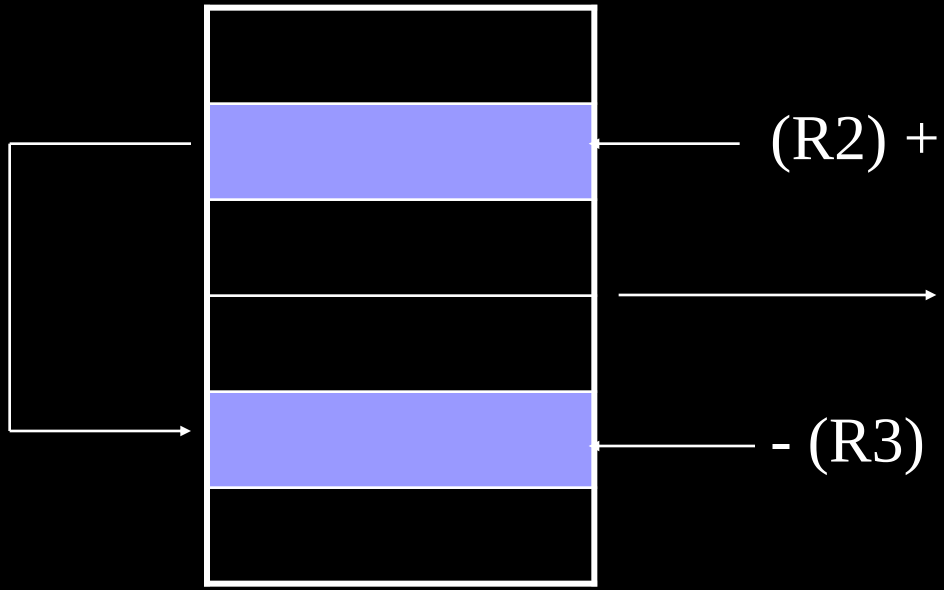
- Example 2 – Block-Moving Assembly Instruction
  - MVC x, y, 256
    - IBM System 360/ 370
  - Characteristics
    - More expensive
    - “self-modifying” “operands”
  - Solutions:
    - Pre-load pages
    - Pre-save & recover before page-fault services



Crucial Issues

- Example 3 – Addressing Mode

MOV (R2)+, -(R3)



When the page fault is serviced,
R2, R3 are modified!
- Undo Effects!

Performance of Demand Paging

- Effective Access Time:
  - ma: memory access time for paging
  - p: probability of a page fault
  - pft: page fault time

$$(1 - p) * ma + p * pft$$

Performance of Demand Paging

- Page fault time - major components
  - Components 1&3 (about 10^3 ns ~ 10^5 ns)
    - Service the page-fault interrupt
    - Restart the process
  - Component 2 (about 8ms)
    - Read in the page (multiprogramming! However, let's get the taste!)
    - pft ≈ 8ms = 8,000,000 ns
- Effect Access Time (when ma = 100ns)
  - (1-p) * 100ns + p * 8,000,000 ns
  - 100ns + 7,999,900ns * p

Performance of Demand Paging

- Example (when ma = 100ns)
  - p = 1/1000
  - Effect Access Time ≈ 8,000 ns
    $\rightarrow$ Slowed down by 80 times
  - How to only 10% slow-down?

    $110 > 100 * (1-p) + 8{,}000{,}000 * p$

    $p < 0.00000125$

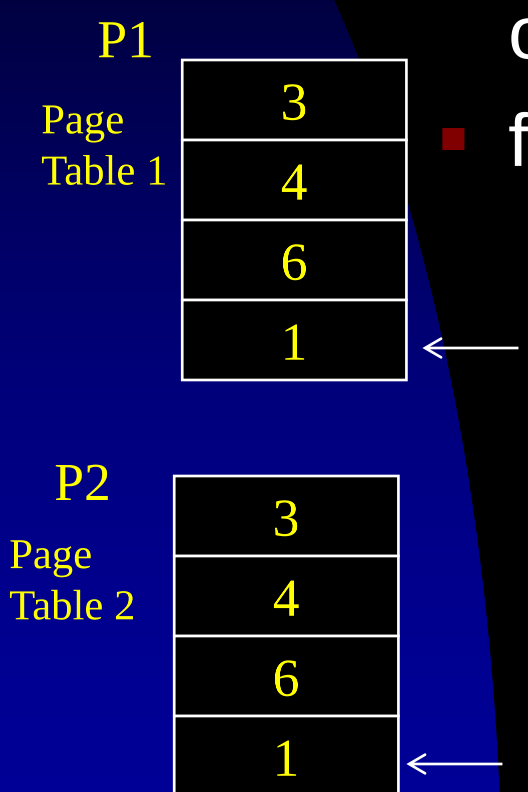
    $p < 1 / 799{,}990$

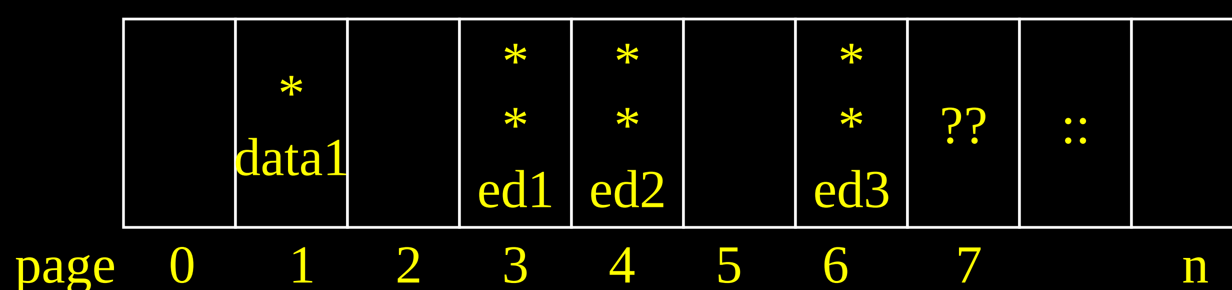
Performance of Demand Paging

- How to keep the page fault rate low?
  - Effective Access Time ≈ 100ns + 7,999,900ns * p
- Handling of Swap Space – A Way to Reduce Page Fault Time (pft)
  - Disk I/O to swap space is generally faster than that to the file system.
    - Preload processes into the swap space before they start up.
    - Demand paging from file system but do page replacement to the swap space. (BSD UNIX)

Copy-on-Write

- Rapid Process Creation and Reducing of New Pages for the New Process
- fork(); execve()
  - Shared pages → copy-on-write pages
    - Only the pages that are modified are copied!





* Windows 2000, Linux, Solaris 2 support this feature!

Copy-on-Write

- zero-fill-on-demand
  - Zero-filled pages, e.g., those for the stack or bss.
- vfork() vs fork() with copy-on-write
  - vfork() lets the sharing of the page table and pages between the parent and child processes.
  - Where to keep the needs of copy-on-write information for pages?

Page Replacement

- Demand paging increases the multiprogramming level of a system by "potentially" over-allocating memory.
  - Total physical memory = 40 frames
  - Run six processes of size equal to 10 frames but with only five frames. => 10 spare frames
- Most of the time, the average memory usage is close to the physical memory size if we increase a system's multiprogramming level!

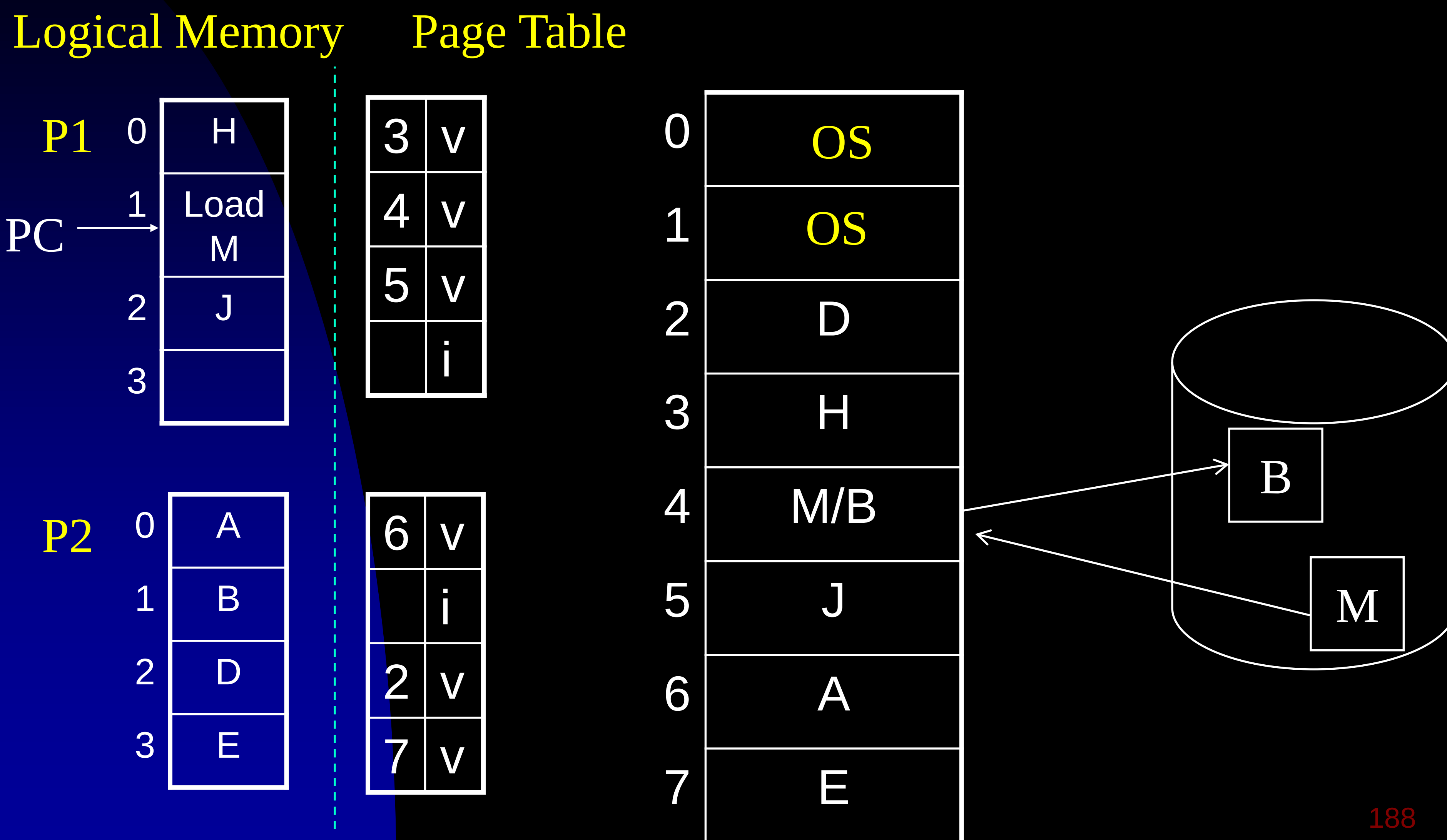
Page Replacement

- Q: Should we run the 7th processes?
  - How if the six processes start to ask their shares?
- What to do if all memory is in use, and more memory is needed?
- Answers
  - Kill a user process!
    - But, paging should be transparent to users?
  - Swap out a process!
  - Do page replacement!

Page Replacement

- A Page-Fault Service
  - Find the desired page on the disk!
  - Find a free frame
    - Select a victim and write the victim page out when there is no free frame!
  - Read the desired page into the selected frame.
  - Update the page and frame tables, and restart the user process.

Page Replacement



Page Replacement

- Two page transfers per page fault if no frame is available!

Page Table

6	V	N
4	V	N
3	V	Y
7	V	Y

Valid-Invalid Bit

Modify Bit is set by the hardware automatically!

Modify (/Dirty) Bit! To "eliminate" 'swap out" => Reduce I/O time by one-half

Page Replacement

- Two Major Pieces for Demand Paging
  - Frame Allocation Algorithms
    - How many frames are allocated to a process?
  - Page Replacement Algorithms
    - When page replacement is required, select the frame that is to be replaced!
  - Goal: A low page fault rate!
- Note that a bad replacement choice does not cause any incorrect execution!

Page Replacement Algorithms

- Evaluation of Algorithms
  - Calculate the number of page faults on strings of memory references, called reference strings, for a set of algorithms
- Sources of Reference Strings
  - Reference strings are generated artificially.
  - Reference strings are recorded as system traces:
    - How to reduce the number of data?

Page Replacement Algorithms

- Two Observations to Reduce the Number of Data:
  - Consider only the page numbers if the page size is fixed.
    - Reduce memory references into page references
  - If a page *p* is referenced, any immediately following references to page *p* will never cause a page fault.
    - Reduce consecutive page references of page *p* into one page reference.

Page Replacement Algorithms

- Example

$$\underbrace{\boxed{\text{XX}}}_{\text{page\#}}\underbrace{\boxed{\text{XX}}}_{\text{offset}}$$

0100, 0432, 0101, 0612, 0103, 0104, 0101, 0611

⇩

01, 04, 01, 06, <u>01, 01, 01</u>, 06

⇩

01, 04, 01, 06, 01, 06

Does the number of page faults decrease when the number of page frames available increases?

FIFO Algorithm

- A FIFO Implementation
  1. Each page is given a time stamp when it is brought into memory.
  2. Select the oldest page for 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	7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
		0	0	0		3	3	3	2	2	2			1	1			1	0	0
			1	1		1	0	0	0	3	3			3	2			2	2	1
FIFO queue	7	7	7	0		1	2	3	0	4	2			3	0			1	2	7
		0	0	1		2	3	0	4	2	3			0	1			2	7	0
			1	2		3	0	4	2	3	0			1	2			7	0	1

FIFO Algorithm

- The Idea behind FIFO
  - The oldest page is unlikely to be used again.
  - ??Should we save the page which will be used in the near future??
- Belady’s anomaly
  - For some page-replacement algorithms, the page fault rate may increase as the number of allocated frames increases.

FIFO Algorithm

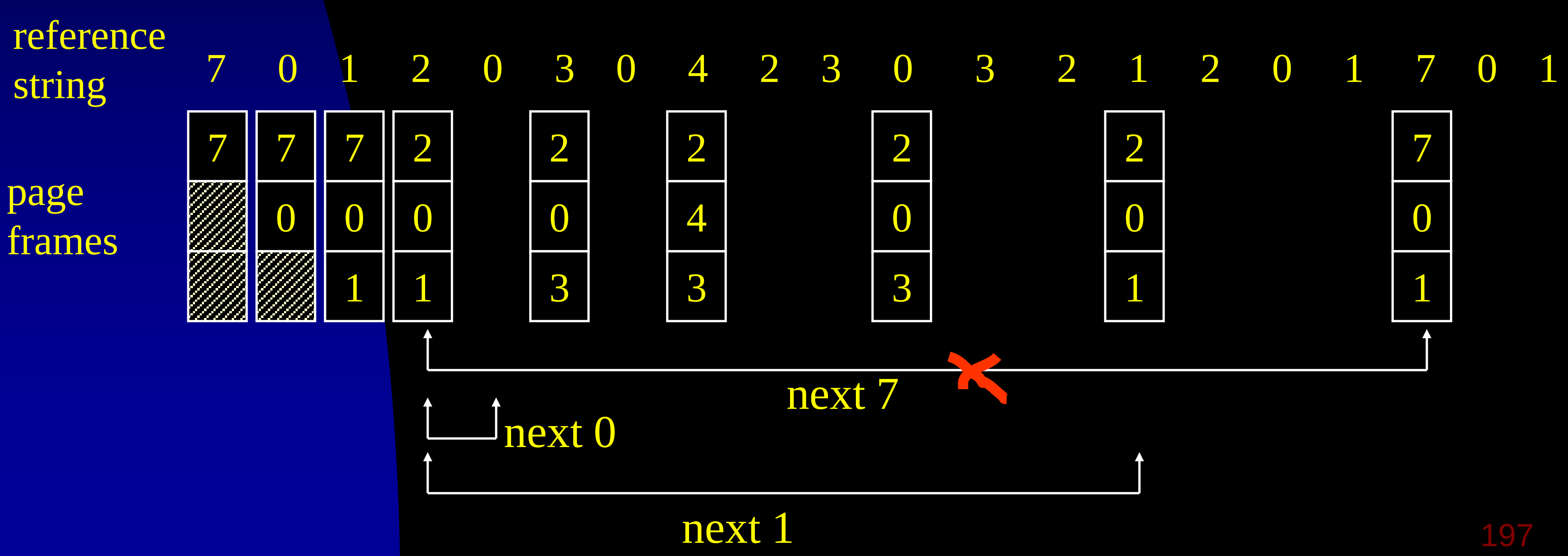
Run the FIFO algorithm on the following reference:

string:	1	2	3	4	1	2	5	1	2	3	4	5
	●	●	●	●	●	●	●			●	●	
3 frames	1	1	1	2	3	4	1	1	1	2	5	5
		2	2	3	4	1	2	2	2	5	3	3
			3	4	1	2	5	5	5	3	4	4
	●	●	●	●			●	●	●	●	●	●
4 frames	1	1	1	1	1	1	2	3	4	5	1	2
		2	2	2	2	2	3	4	5	1	2	3
			3	3	3	3	4	5	1	2	3	4
				4	4	4	5	1	2	3	4	5

→ Push out pages that will be used later!

Optimal Algorithm (OPT)

- Optimality
  - One with the lowest page fault rate.
- Replace the page that will not be used for the longest period of time. ←→ Future Prediction



Least-Recently-Used Algorithm (LRU)

- The Idea:
  - OPT concerns when a page is to be used!
  - “Don’t have knowledge about the future”?!

$$\Downarrow$$

- Use the history of page referencing in the past to predict the future!

$S \stackrel{?}{=} S^R$ (S^R is the reverse of S !)

LRU Algorithm

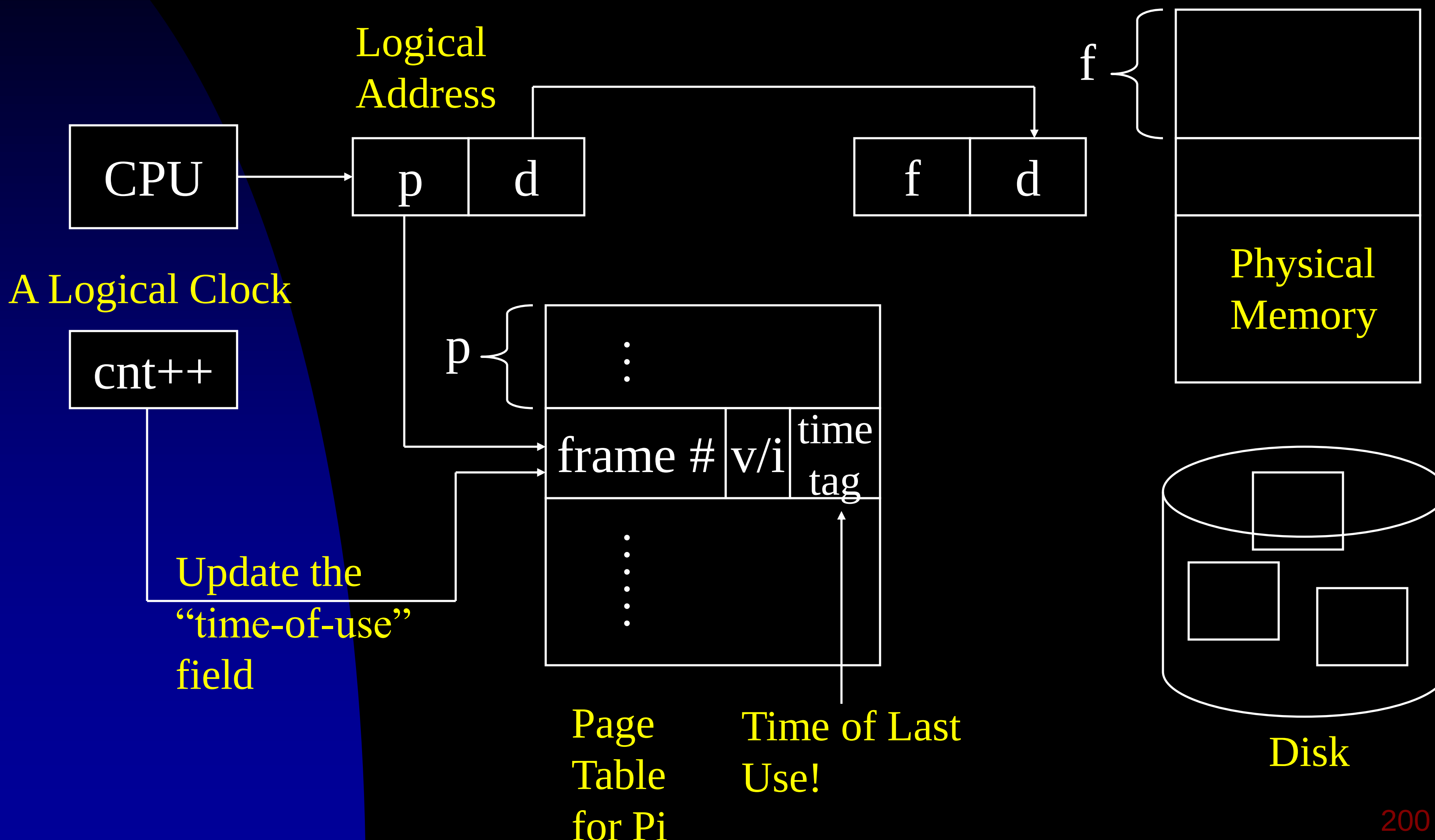
- Example

reference string	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
page frames	7	7	7	2		2		4	4	4	0			1		1		7		
		0	0	0		0		0	0	3	3			3		0		0		
			1	1		3		3	2	2	2			2		2		7		
LRU queue	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
		7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0
			7	0	1	2	2	3	0	4	2	2	0	3	3	1	2	0	1	7

a wrong prediction!

Remark: LRU is like OPT which "looks backward" in time.

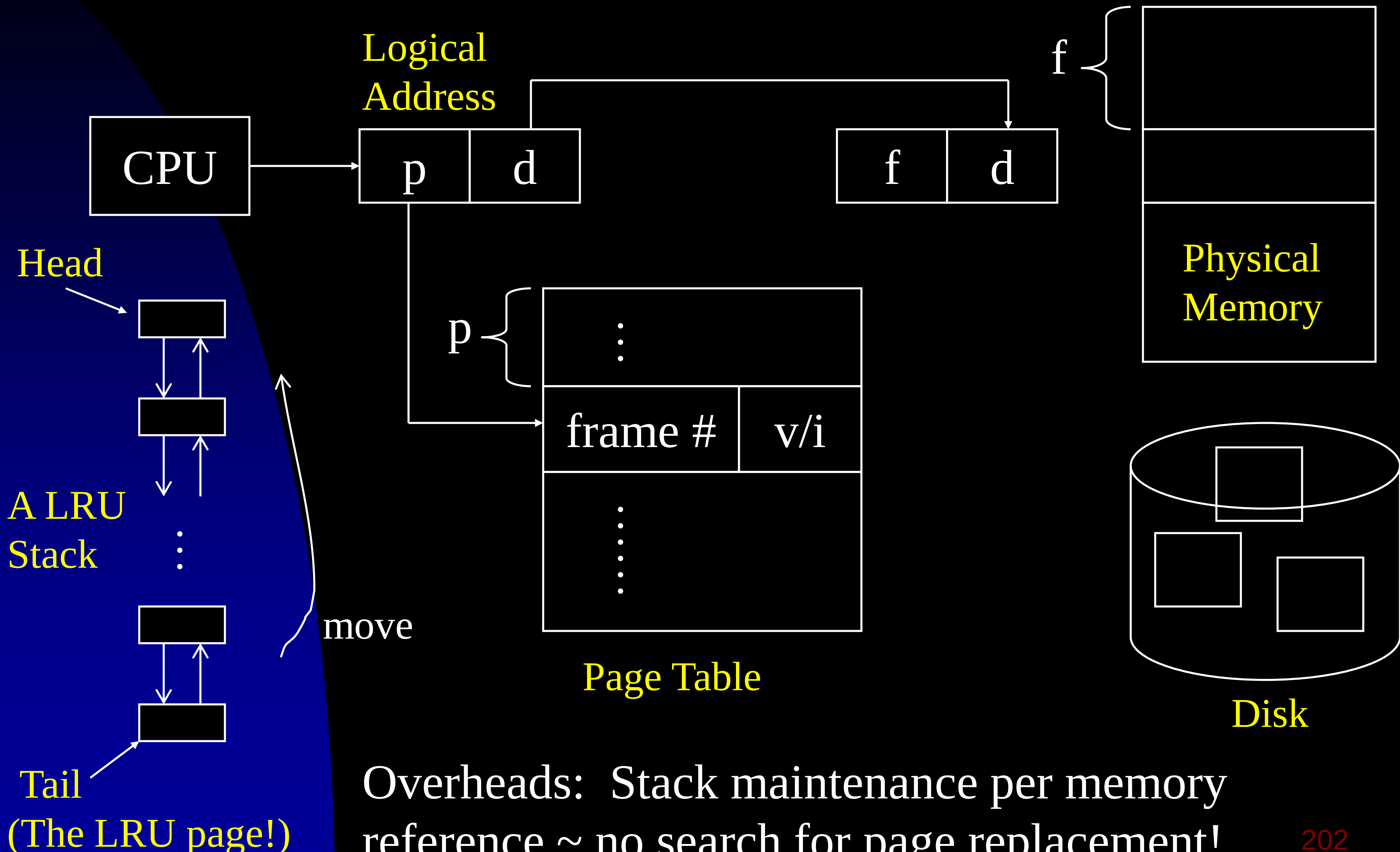
LRU Implementation – Counters



LRU Implementation – Counters

- Overheads
  - The logical clock is incremented for every memory reference.
  - Update the “time-of-use” field for each page reference.
  - Search the LRU page for replacement.
  - Overflow prevention of the clock & the maintenance of the “time-of-use” field of each page table.

LRU Implementation – Stack



Overheads: Stack maintenance per memory reference ~ no search for page replacement!

A Stack Algorithm

$$\left\{ \begin{array}{l} \text{memory-} \\ \text{resident} \\ \text{pages} \end{array} \right\}_{\begin{array}{l}\text{n frames} \\ \text{available}\end{array}} \subseteq \left\{ \begin{array}{l} \text{memory-} \\ \text{resident} \\ \text{pages} \end{array} \right\}_{\begin{array}{l}\text{(n +1) frames} \\ \text{available}\end{array}}$$

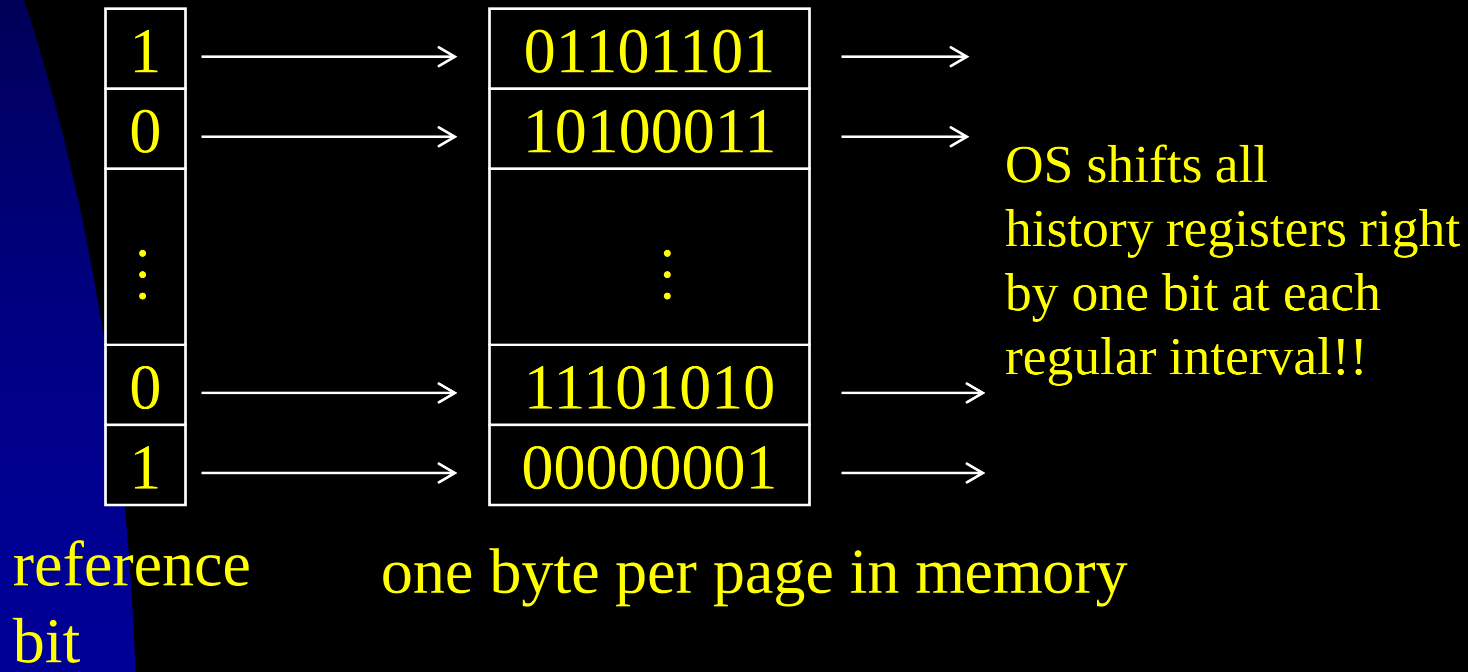
- Need hardware support for efficient implementations.
- Note that LRU maintenance needs to be done for every memory reference.

LRU Approximation Algorithms

- Motivation
  - No sufficient hardware support
  - Most systems provide only “reference bit” which only indicates whether a page is used or not, instead of their order.
- Additional-Reference-Bit Algorithm
- Second-Chance Algorithm
- Enhanced Second Chance Algorithm
- Counting-Based Page Replacement

Additional-Reference-Bits Algorithm

- Motivation
  - Keep a history of reference bits



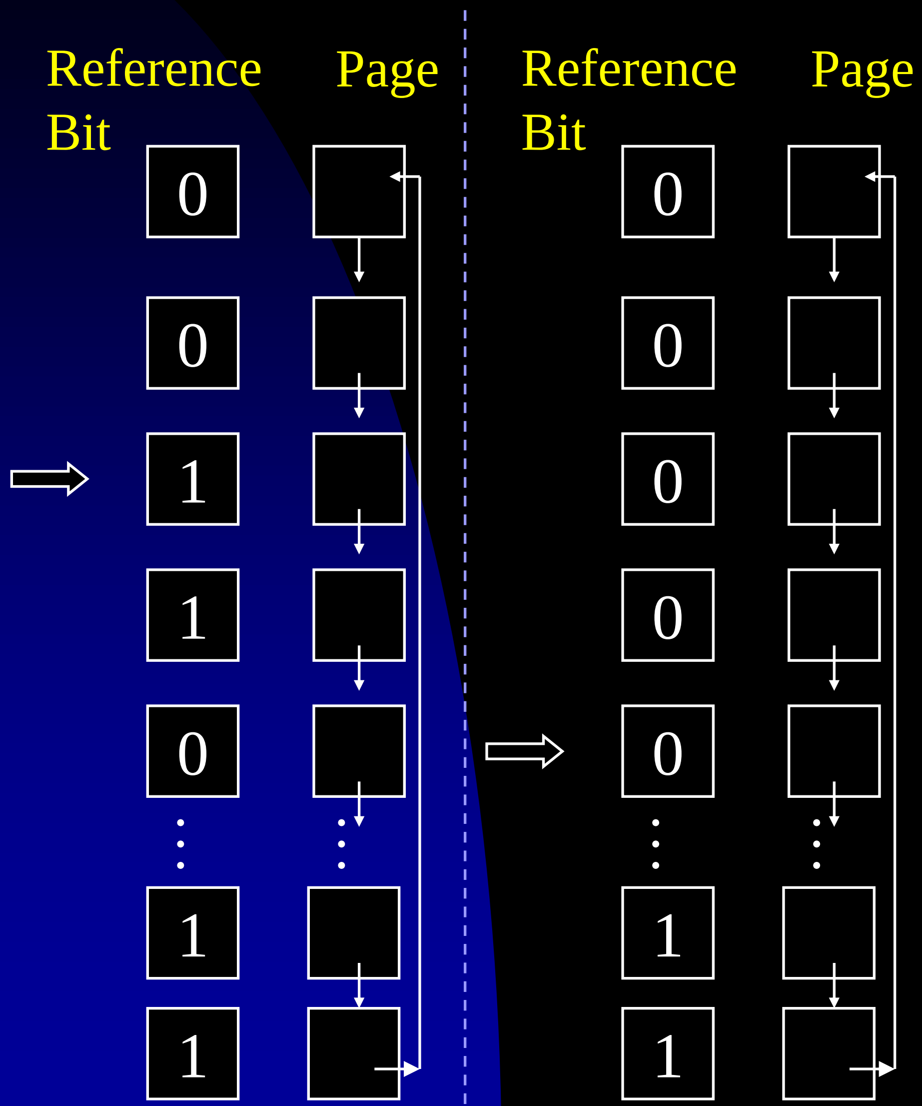
Additional-Reference-Bits Algorithm

- History Registers

LRU (smaller value!)	0 0 0 0 0 0 0 0	← Not used for 8 times
	0 0 0 0 0 0 0 1	
	1 1 1 1 1 1 1 0	
MRU	1 1 1 1 1 1 1 1	← Used at least once every time

- But, how many bits per history register should be used?
  - Fast and cost-effective!
  - The more bits, the better the approximation is.

Second-Chance (Clock) Algorithm



- Motivation
  - Use the reference bit only
- Basic Data Structure:
  - Circular FIFO Queue
- Basic Mechanism
  - When a page is selected
    - Take it as a victim if its reference bit = 0
    - Otherwise, clear the bit and advance to the next page

Enhanced Second-Chance Algorithm

- Motivation:
  - Consider the cost in swapping out pages.
- 4 Classes (reference bit, modify bit)
  - (0,0) – not recently used and not “dirty”
  - (0,1) – not recently used but “dirty”
  - (1,0) – recently used but not “dirty”
  - (1,1) – recently used and “dirty”

low priority

high priority

Enhanced Second-Chance Algorithm

- Use the second-chance algorithm to replace the first page encountered in the lowest nonempty class.

  => May have to scan the circular queue several times before find the right page.
- Macintosh Virtual Memory Management

Counting-Based Algorithms

- Motivation:
  - Count the # of references made to each page, instead of their referencing times.
- Least Frequently Used Algorithm (LFU)
  - LFU pages are less actively used pages!
  - Potential Hazard: Some heavily used pages may no longer be used !
  - A Solution – Aging
    - Shift counters right by one bit at each regular interval.

Counting-Based Algorithms

- Most Frequently Used Algorithm (MFU)
  - Pages with the smallest number of references are probably just brought in and has yet to be used!
- LFU & MFU replacement schemes can be fairly expensive!
- They do not approximate OPT very well!

Page Buffering

- Basic Idea
  a. Systems keep a pool of free frames
  b. Desired pages are first “swapped in” some frames in the pool.
  c. When the selected page (victim) is later written out, its frame is returned to the pool.
- Variation 1
  a. Maintain a list of modified pages.
  b. Whenever the paging device is idle, a modified page is written out and reset its “modify bit”.

Page Buffering

- Variation 2
  a. Remember which page was in each frame of the pool.
  b. When a page fault occurs, first check up whether the desired page is there already.
     - Pages which were in frames of the pool must be “clean”.
     - “Swapping-in” time is saved!
- VAX/VMS with the FIFO replacement algorithm adopt it to improve the performance of the FIFO algorithm.

Frame Allocation – Single User

- Basic Strategy:
  - User process is allocated any free frame.
  - User process requests free frames from the free-frame list.
  - When the free-frame list is exhausted, page replacement takes place.
  - All allocated frames are released by the ending process.
- Variations
  - O.S. can share with users some free frames for special purposes.
  - Page Buffering - Frames to save “swapping” time

Frame Allocation – Multiple Users

- Fixed Allocation

  a. Equal Allocation

     m frames, n processes → m/n frames per process

  b. Proportional Allocation

     1. Ratios of Frames $\propto$ Size

        $S = \Sigma S_i$, $A_i \propto (S_i / S) \times m$, where (sum <= m) & ($A_i$ >= minimum # of frames required)

     2. Ratios of Frames $\propto$ Priority

        S_i : relative importance

     3. Combinations, or others.

Frame Allocation – Multiple Users

- Dynamic Allocation
  a. Allocated frames $\propto$ the multiprogramming level
  b. Allocated frames $\propto$ Others
- The minimum number of frames required for a process is determined by the instruction-set architecture.
  - ADD A,B,C → 4 frames needed
  - ADD (A), (B), (C) → 1+2+2+2 = 7 frames, where (A) is an indirect addressing.

Frame Allocation – Multiple Users

- Minimum Number of Frames (Continued)
  - How many levels of indirect addressing should be supported?
    - It may touch every page in the logical address space of a process

      => Virtual memory is collapsing!
  - A long instruction may cross a page boundary.

    MVC X, Y, 256 → 2 + 2 + 2 = 6 frames
    - The spanning of the instruction and the operands.

16 bits

	address

0 **direct**
1 **indirect**

Frame Allocation – Multiple Users

- Global Allocation
  - Processes can take frames from others. For example, high-priority processes can increase its frame allocation at the expense of the low-priority processes!
- Local Allocation
  - Processes can only select frames from their own allocated frames → Fixed Allocation
  - The set of pages in memory for a process is affected by the paging behavior of only that process.

Frame Allocation – Multiple Users

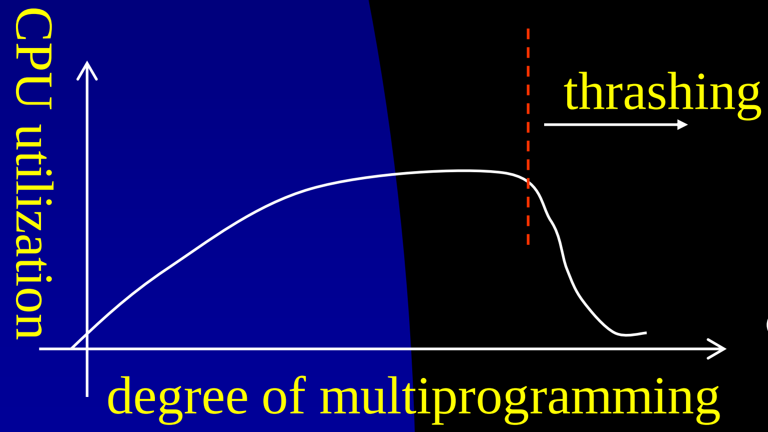
- Remarks
  a. Global replacement generally results in a better system throughput
  b. Processes might not control their own page fault rates such that a process can affect each another easily under global replacement.

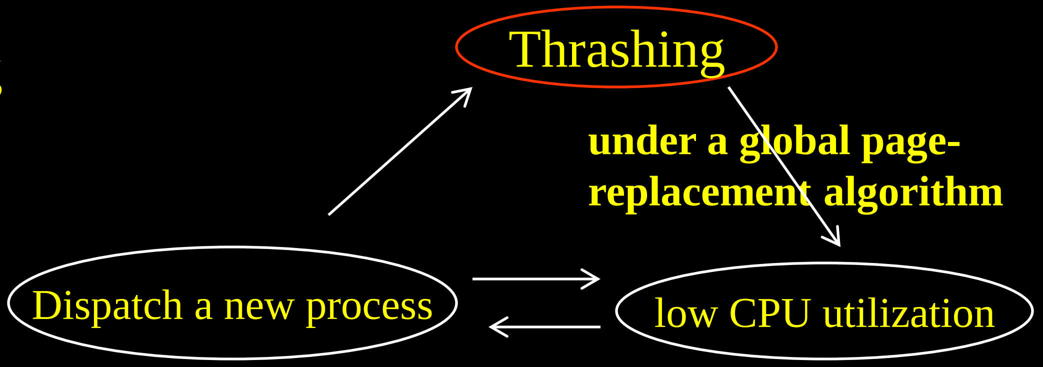
Non-Uniform Memory Access

- Non-Uniform Memory Access (NUMA) Systems:
  - The access of some memory section might be faster than that of another.
    - It is due to how CPUs and memory are interconnected.
  - Potential Issues
    - Memory frame allocation versus CPU scheduling
      - Threading Considerations over Cores

Thrashing

- Thrashing – A High Paging Activity:
  - A process is thrashing if it is spending more time paging than executing.
- Why thrashing?
  - Too few frames allocated to a process!

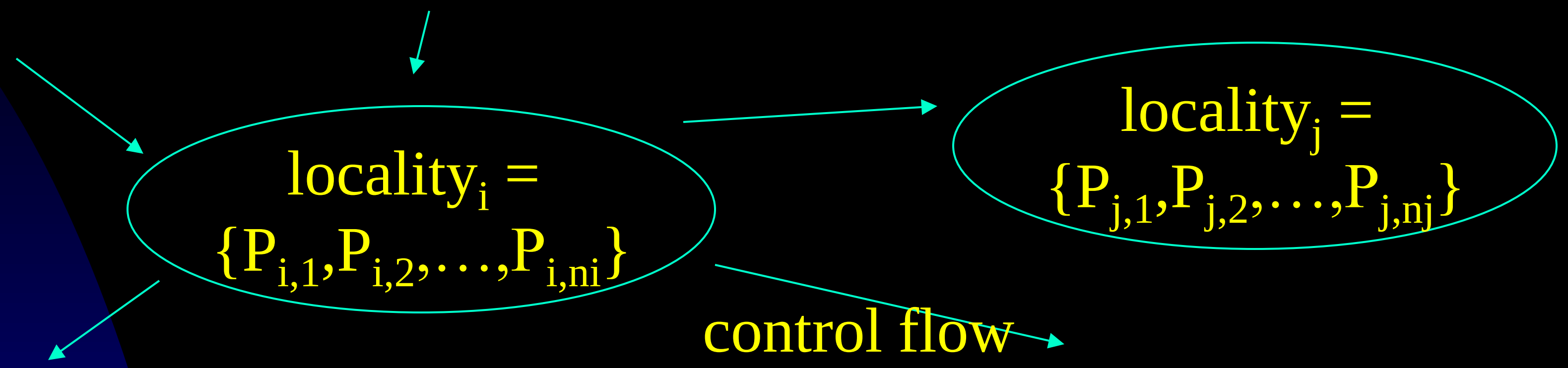




Thrashing

- Solutions:
  - Decrease the multiprogramming level → Swap out processes!
  - Use local page-replacement algorithms
    - Only limit thrashing effects “locally”
    - Page fault services of other processes also slow down.
  - Give processes as many frames as they need!
    - But, how do you know the right number of frames for a process?

Locality Model



- A program is composed of several different (overlapped) localities.
  - Localities are defined by the program structures and data structures (e.g., an array, hash tables)
- How do we know that we allocate enough frames to a process to accommodate its current locality?

Working-Set Model

Page references

…2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4

$\longleftarrow \Delta \longrightarrow$ working-set window t1

$\longleftarrow \Delta \longrightarrow$ working-set window t2

working-set(t1) = {1,2,5,6,7}

working-set(t2) = {3,4}

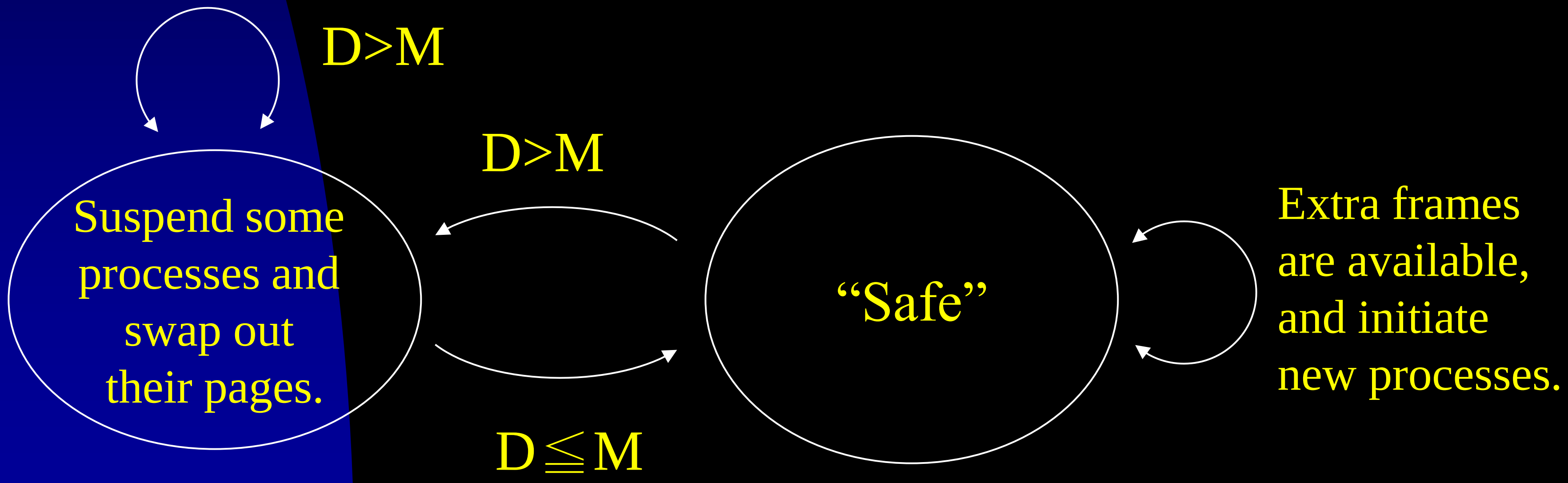
- The working set is an approximation of a program's locality.

The minimum allocation $\longleftarrow \Delta \quad \infty \longrightarrow$ All touched pages may cover several localities.

Working-Set Model

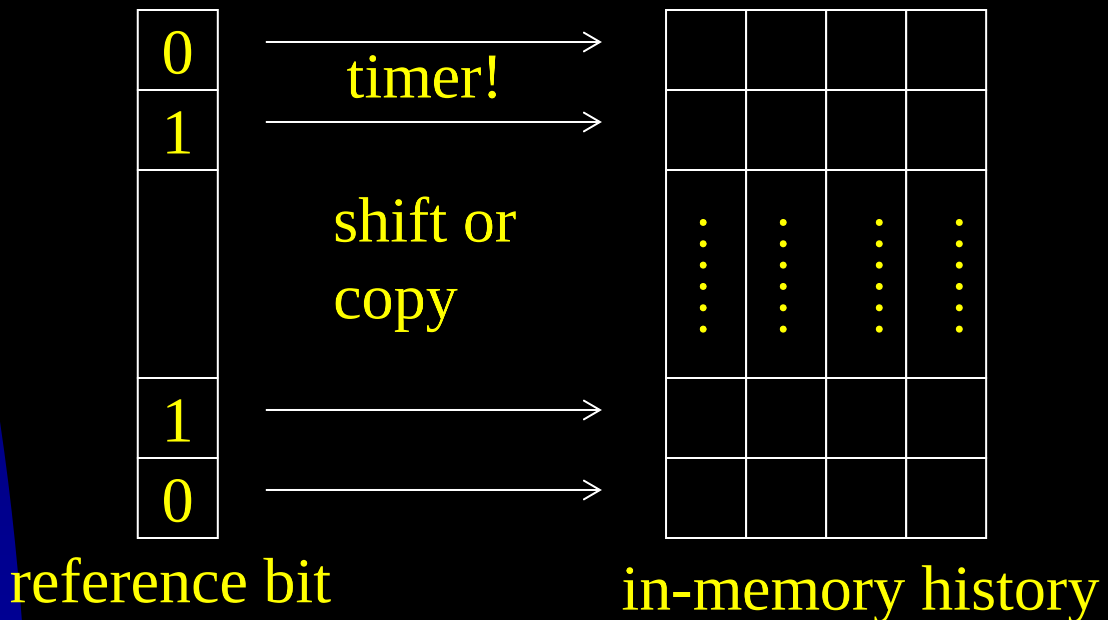
$$D = \sum working - set - size_i \le M$$

where M is the total number of available frames.



Working-Set Model

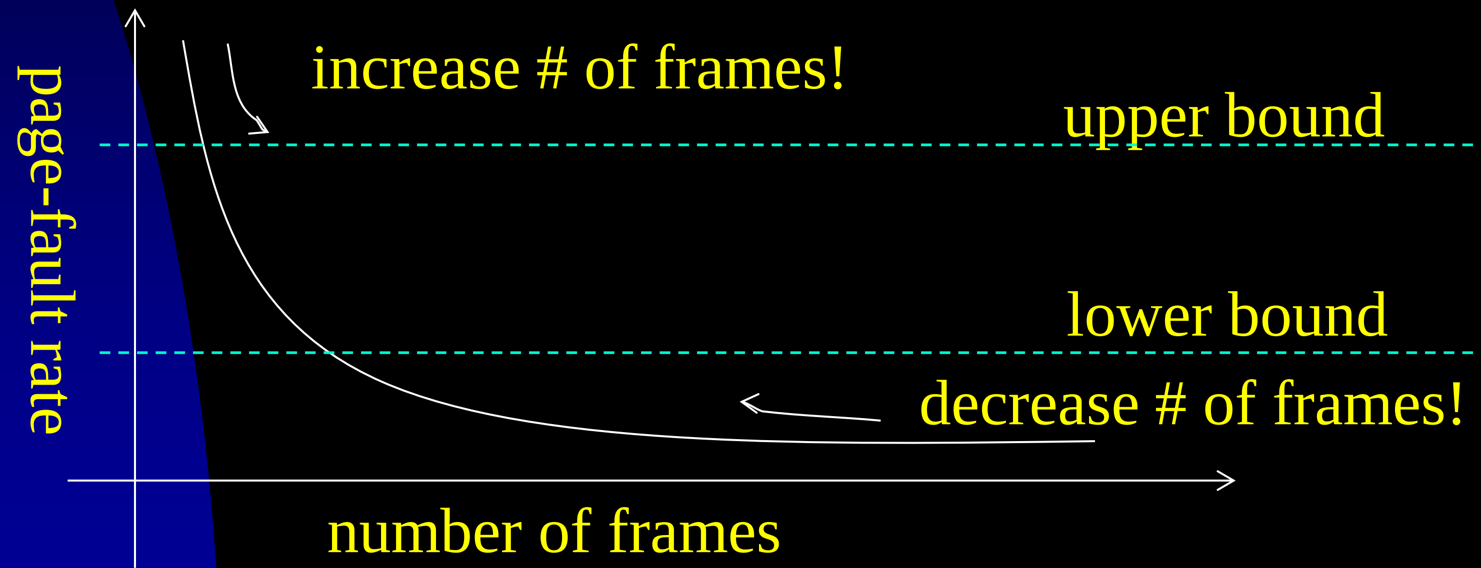
- The maintenance of working sets is expensive!
  - Approximation by a timer and the reference bit



  - Accuracy v.s. Timeout Interval!

Page-Fault Frequency

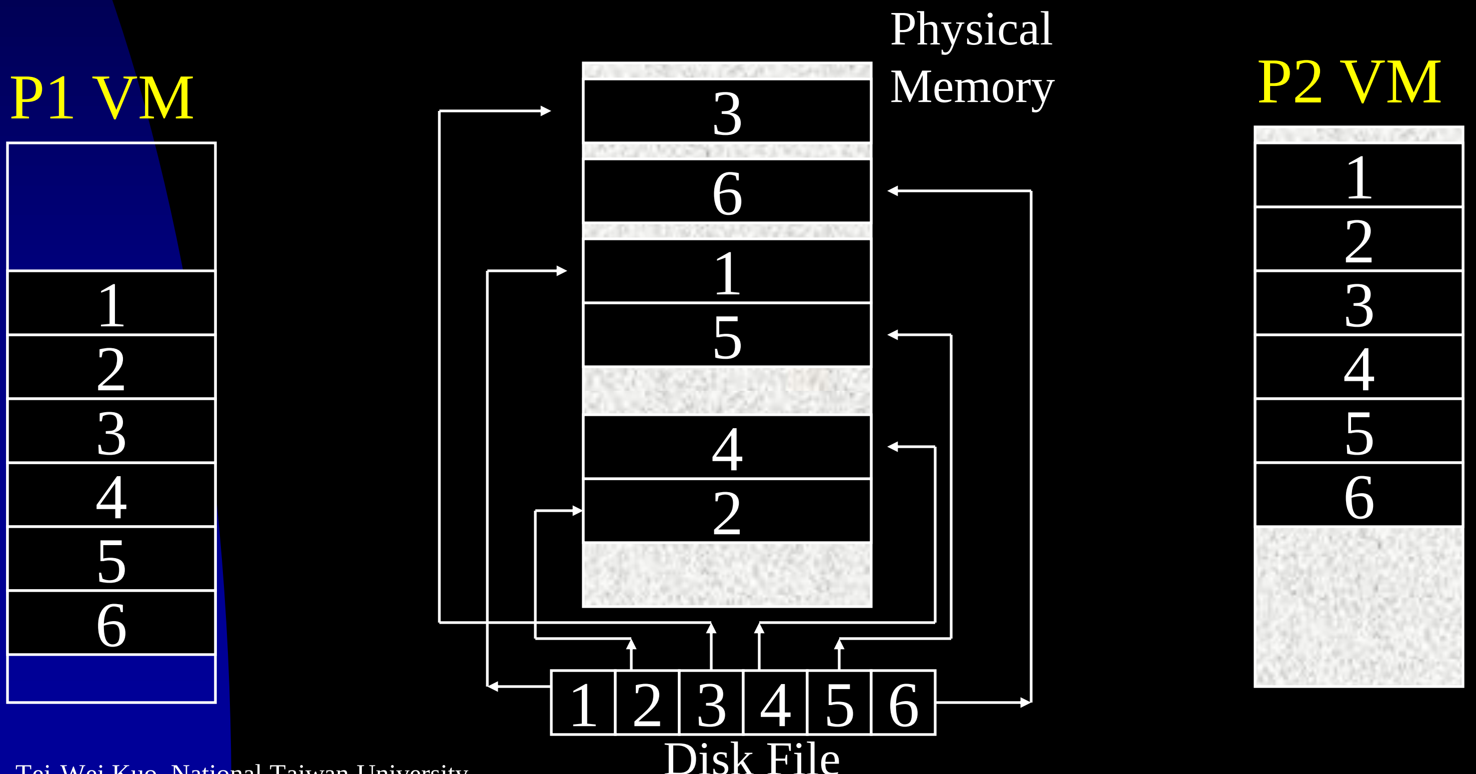
- Motivation
  - Control thrashing directly through the observation on the page-fault rate!



*Processes are suspended and swapped out if the number of available frames is reduced to that under the minimum needs.

Memory-Mapped Files

- File writes might not cause any disk write!
- Solaris 2 uses memory-mapped files for open(), read(), write(), etc.



Shared Memory – Windows API

- Producer

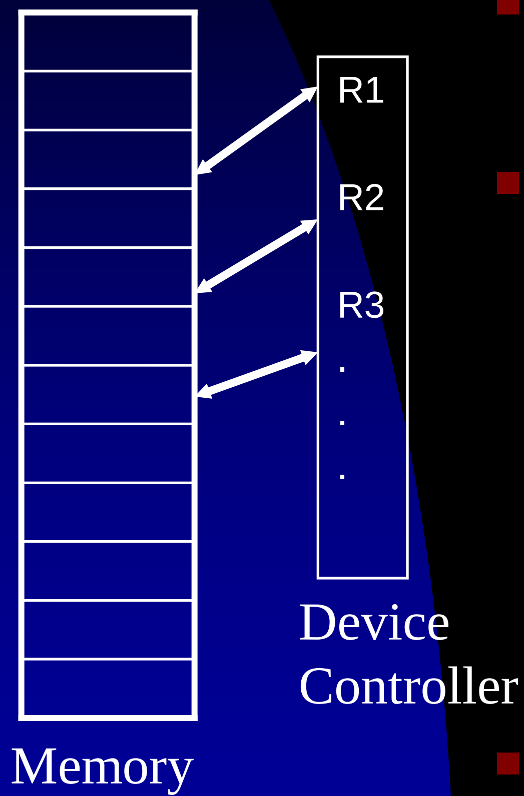
1. hfile=CreateFile("temp,txt", …);
2. hmapfile=CreateFileMapping(hfile, …, TEXT("Shared Object"));
3. lpmapaddr=MapViewOfFile(hmapfile, …);
4. sprintf(lpmapaddr,"for consumer");
5. UnmapViewOfFile(lpmapaddr);
6. CloseHandle(hfile);
7. CloseHandle(hmapfile);

- Consumer

1. hmapfile=OpenFileMapping(…, TEXT("Shared Object"));
2. lpMapAddress=MapViewOfFile(hmapfile, …);
3. printf("Read msg %s", lpMapAddress);
4. UnmapViewOfFile(lpmapaddr);
5. CloseHandle(hmapfile);

* Named shared-memory objects

Memory-Mapped I/O



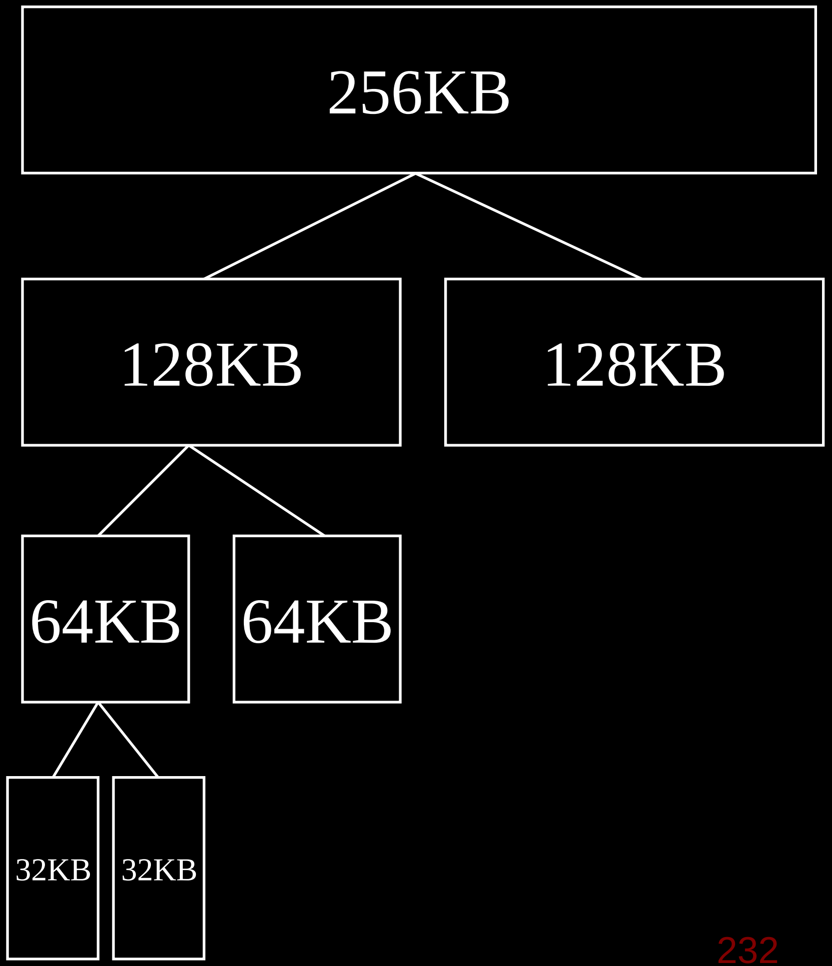
- Processor can have direct access!
- Intermediate storage for data in the registers of device controllers
- Memory-Mapped I/O (PC & Mac)
  (1) Frequently used devices
  (2) Devices must be fast, such as video controller, or special I/O instructions are used to move data between memory & device controller registers
- Programmed I/O – polling
  - or interrupt-driven handling

Kernel Memory Allocation

- Separation from user-mode memory allocation
  - The kernel might request memory of various sizes, that are often less than a page in size.
  - Certain hardware devices interact directly with physical memory, and the accesses memory must be in physically contiguous pages!

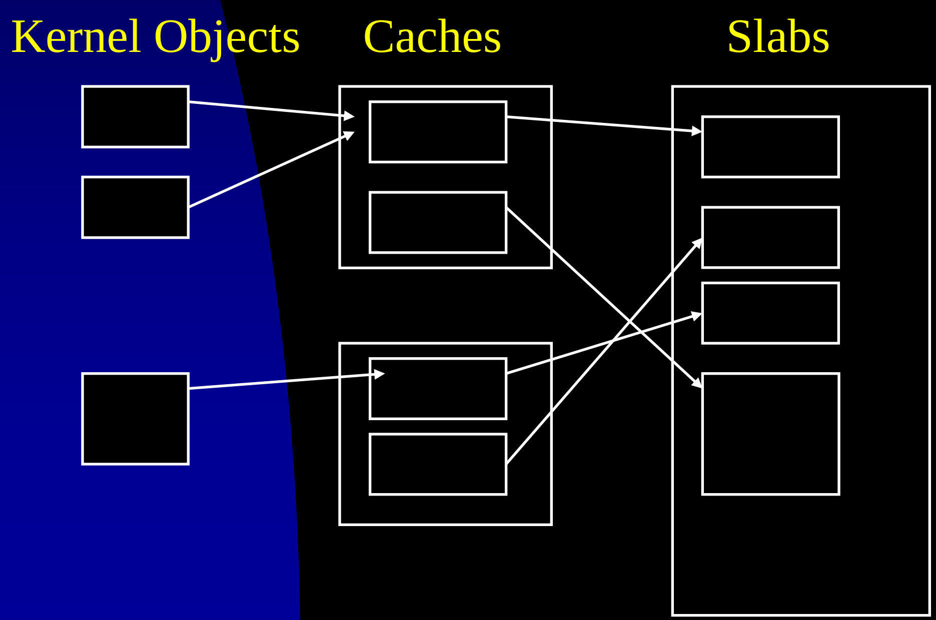
Kernel Memory Allocation

- The Buddy System
  - A fixed-size segment of physically contiguous pages
  - A power-of-2 allocator
  - Advantage: quick coalescing algorithms
  - Disadvantage: internal fragmentation



Kernel Memory Allocation

- Slab Allocation
  - Slab: one or more physically contiguous pages
  - Cache: one or more slabs



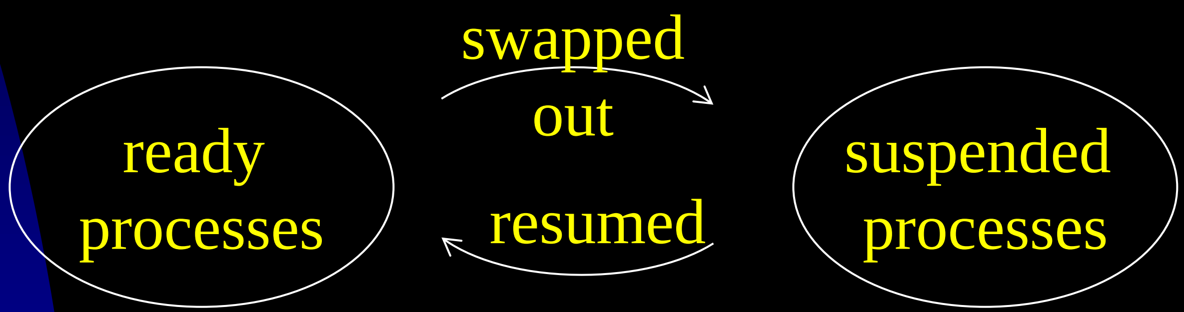
- Slab States
  - Full
  - Empty
  - Partial

Kernel Memory Allocation

- Slab Allocator
  - Look for a free object in a partial slab.
  - Otherwise, allocate a new slab and assign it to a cache.
- Benefits
  - No space wasted in fragmentation.
  - Memory requests are satisfied quickly.
- Implementations
  - Solaris 2.4 kernel, Linux version 2.2+

Other Considerations

- Pre-Paging
  - Bring into memory at one time all the pages that will be needed!



Do pre-paging if the working set is known!

- Issue

Pre-Paging Cost ⟷ Cost of Page Fault Services

Not every page in the working set will be used!

Other Considerations

- Page Size

Better Resolution for Locality & Internal Fragmentation

Page Size

small | p | d | large

512B(2^9)~16,384B(2^{12})

Smaller Page Table Size & Better I/O Efficiency

- Trends - Large Page Size

∵ The CPU speed and the memory capacity grow much faster than the disk speed!

Other Considerations

- TLB Reach
  - TLB-Entry-Number * Page-Size
- Wish
  - The working set is stored in the TLB!
  - Solutions
    - Increase the page size
    - Have multiple page sizes – UltraSparc II (8KB - 4MB) + Solaris 2 (8KB or 4MB)

Other Considerations

- Inverted Page Table
  - The objective is to reduce the amount of physical memory for page tables, but they are needed when a page fault occurs!
  - More page faults for page tables will occur!!!

Other Considerations

- Program Structure
  - Motivation – Improve the system performance by an awareness of the underlying demand paging.

```
var A: array [1..128,1..128] of integer;
    for j:=1 to 128
        for i:=1 to 128
            A(i,j):=0
```

128 words

A(1,1)	A(2,1)	……	A(128,1)
A(1,2)	A(2,2)		A(128,2)
.	.		.
.	.		.
A(1,128)	A(2,128)		A(128,128)

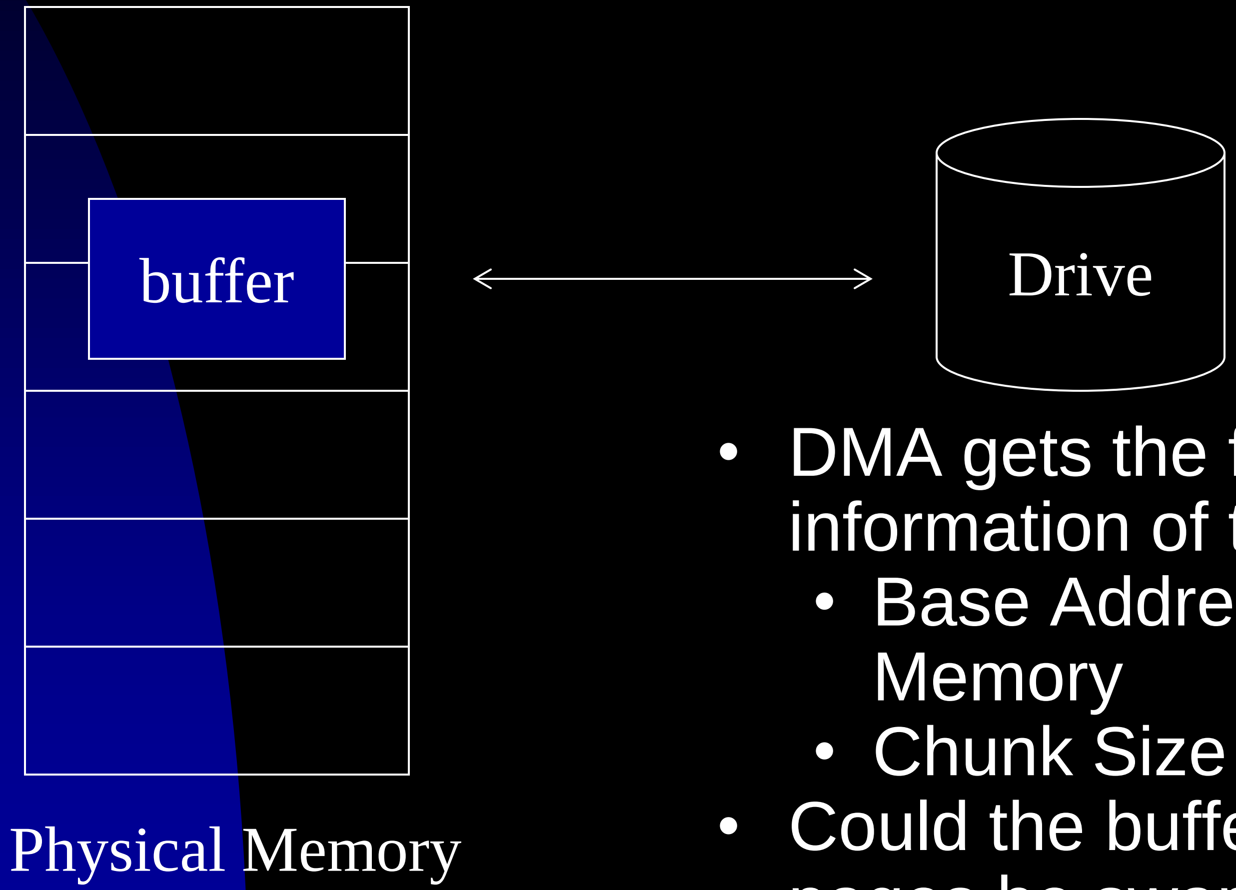
128 pages

→ 128x128 page faults if the process has less than 128 frames!!

Other Considerations

- Program Structures:
  - Data Structures
    - Locality: stack, hash table, etc.
    - Search speed, # of memory references, # of pages touched, etc.
  - Programming Language
    - Lisp, PASCAL, etc.
  - Compiler & Loader
    - Separate code and data
    - Pack inter-related routines into the same page
    - Routine placement (across page boundary?)

I/O Interlock



- DMA gets the following information of the buffer:
  - Base Address in Memory
  - Chunk Size
- Could the buffer-residing pages be swapped out?

I/O Interlock

- Solutions
  - I/O Device ←→ System Memory ←→ User Memory
    - Extra Data Copying!!
  - Lock pages into memory
    - The lock bit of a page-faulting page is set until the faulting process is dispatched!
    - Lock bits might never be turned off!
    - Multi-user systems usually take locks as “hints” only!

Real-Time Processing

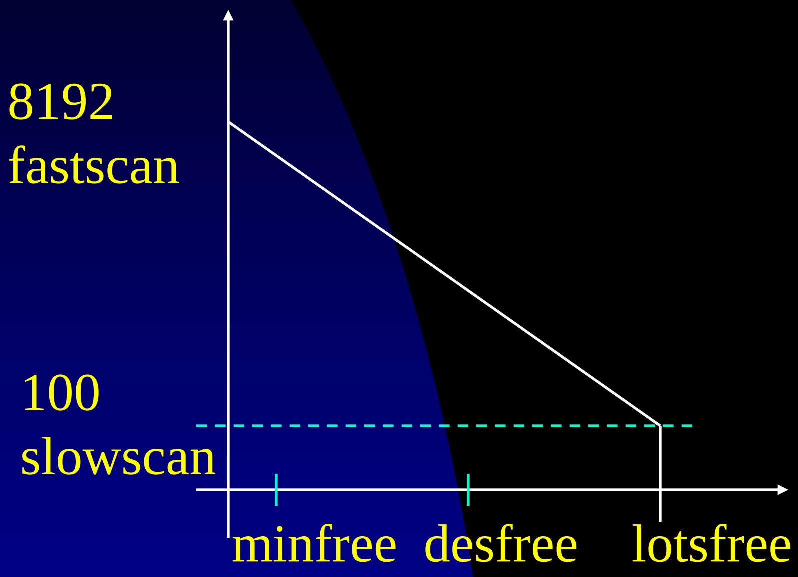
Predictable Behavior ⟷ Virtual memory introduces unexpected, long-term delays in the execution of a program.

- Solution:
  - Go beyond locking hints → Allow privileged users to require pages being locked into memory!

OS Examples – Windows

- Virtual Memory – Demand Paging with Clustering
  - Clustering brings in more pages surrounding the faulting page!
- Working Set
  - A Min and Max bounds for a process
    - Local page replacement when the max number of frames are allocated.
  - Automatic working-set trimming reduces allocated frames of a process to its min when the system threshold on the available frames is reached.

OS Examples – Solaris



- Process *pageout* first clears the reference bit of all pages to 0 and then later returns all pages with the reference bit = 0 to the system *(handspread).*
  - 4HZ → 100HZ when *desfree* is reached!
    - Swapping starts when *desfree* fails for 30s.
  - *pageout* runs for every request to a new page when *minfree* is reached.

Demand Segmentation

- Motivation
  - Segmentation captures better the logical structure of a process!
  - Demand paging needs a significant amount of hardware!
- Mechanism
  - Like demand paging!
  - However, compaction may be needed!
    - Considerable overheads!