Operating Systems

Lecture 5-6
Virtual Memory Management

Department of Computer Science & Technology
Tsinghua University

Review

- Computer Arch/Memory Hierarchy
- Address Space & Address Generation
- Contiguous Memory Allocation
  - Dynamic Allocation of Partitions
- Non-Contiguous Memory Allocation
  - Segmentation
  - Paging
  - Page Table
  - Paged Segmentation Model

OS **Outline**

- ● Principle of Locality & Address Translation
  - Goal
  - Method
  - Characteristics: discontinuous
  - Locality
  - Translation: share , exception
- Virtual Memory
- Mechanisms for Implementing VM
- Local Page Replacement
- Global Page Replacement
- Belady Phenomenon

Memory Management Goals

- Support multiprogramming
  - Provide the abstraction of address space
  - Enforce isolation and protection
  - Enable new programming models like shared memory
- Manage memory resource and use them efficiently
  - Utilize the memory hierarchy
  - Better resource allocation algorithms

Method

- 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

Characteristics of Paging and Segmentation

- Memory references are dynamically translated into physical addresses at run time
  - a process may be swapped in and out of main memory such that it occupies different regions
- A process may be broken up into pieces (pages or segments) that do not need to be located contiguously in main memory
- Hence: all pieces of a process do not need to be loaded in main memory during execution
  - computation may proceed for some time if the next instruction to be fetch (or the next data to be accessed) is in a piece located in main memory

OS

Principle of Locality

- Program and data references within a process tend to cluster
- Only a few pieces of a process will be needed over a short period of time
- Possible to make intelligent guesses about which pieces will be needed in the future
- This suggests that virtual memory may work efficiently

Temporal locality
Spatial locality
Branch locality

OS

Support Needed for Virtual Memory

Hardware

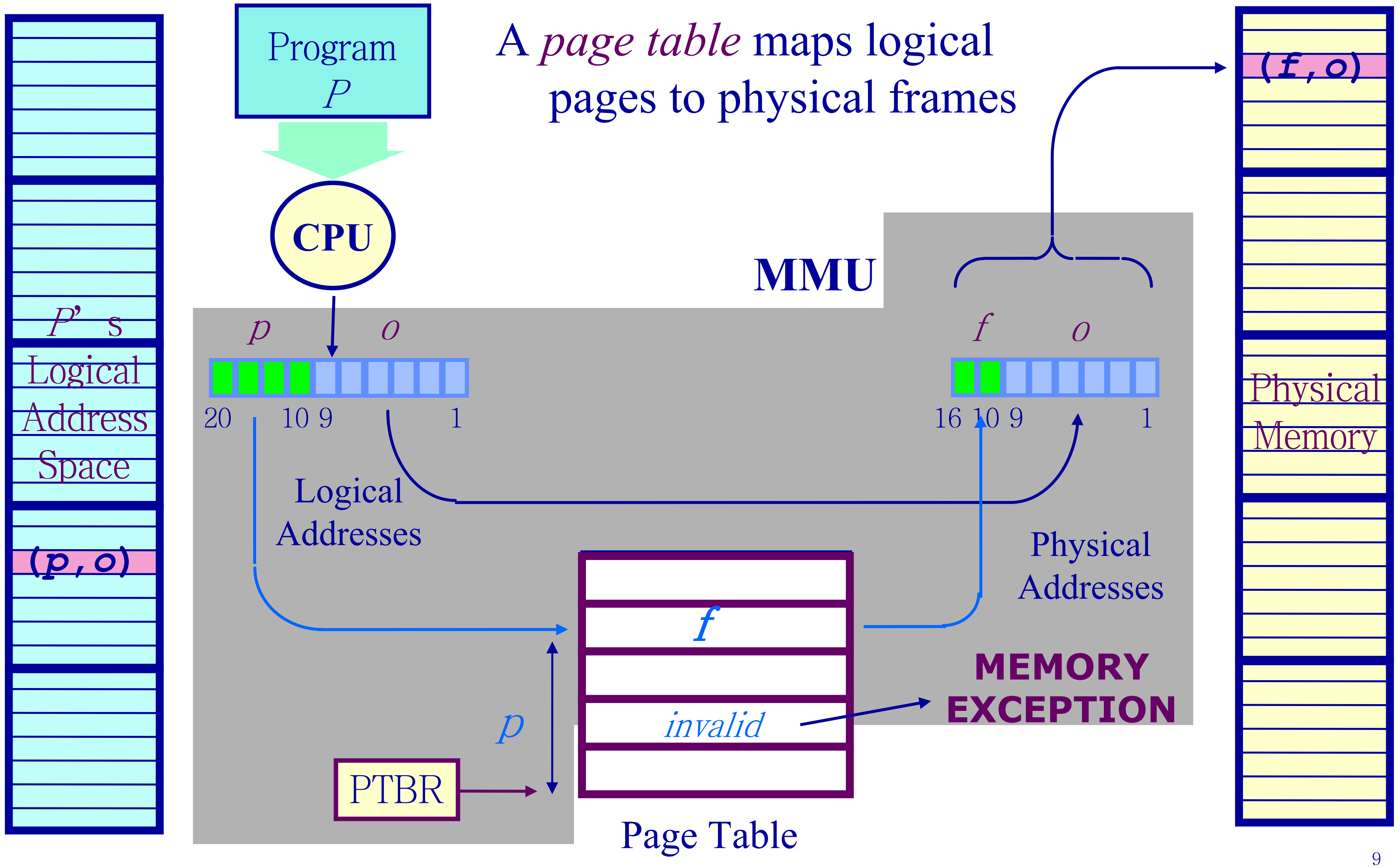
- must support paging and/or segmentation

Operating system

- must be able to management the movement of pages and/or segments between secondary memory and main memory

Paging: Mechanisms

A *page table* maps logical pages to physical frames

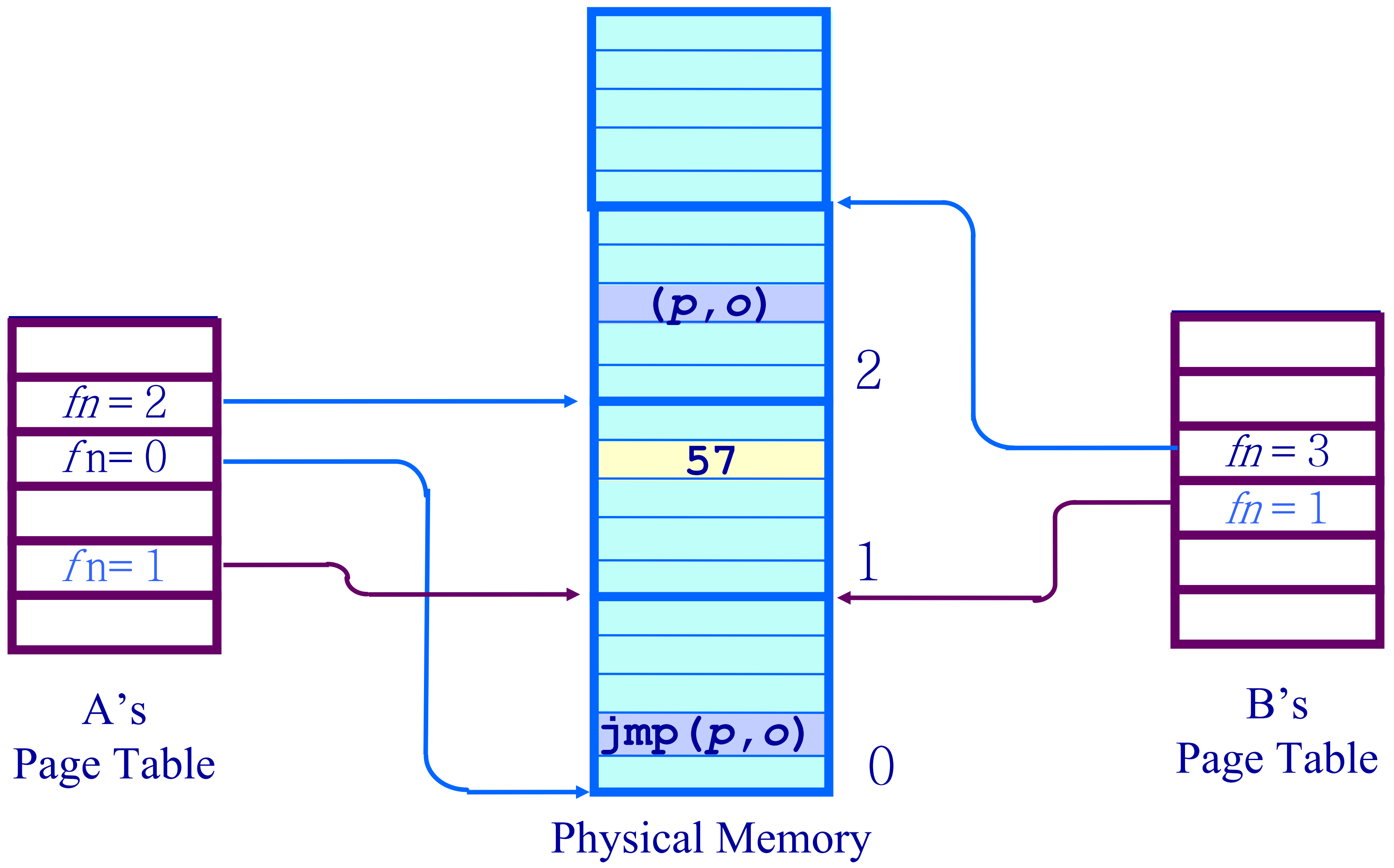


Address Translation

- Mapping from logical address space to physical memory space
  - MM: L->P
  - Each process has its own mapping
- How memory management achieves isolation?
  - Each concurrent process is mapped to disjointed physical space
- How to support sharing (e.g., shared libraries)?
  - Shared segment (or page) of two or more processes is mapped to the same physical address
- If translation fails: memory exception

OS

Shared Page



Memory Exceptions

Must be dealt with in all memory models

- Memory access issues in MMU

When do memory exceptions happen?

- Contiguous Allocation: address out-of-bound (LIMIT)
- Segmentation: address out-of-bound
- Segmentation: segmentation number doesn’t exist
- Paging: page not mapped to a frame

What happens when there is memory exception?

- MMU will raise the exception line in CPU
- CPU will jump to the corresponding exception handler (an kernel subroutine pre-registered to this exception type)
- Now up to the handler to do what is necessary (like kill the process, or do something else)

OS Outline

- Principle of Locality & Address Translation
- ● Virtual Memory
    - Demand Paging
    - Page Fault Handling
- Mechanisms for Implementing VM
- Local Page Replacement
- Global Page Replacement
- Belady Phenomenon

OS

Virtual Memory

Problem: how can one support running programs that requires more memory than the computer's physical main memory?

The concept of virtual memory

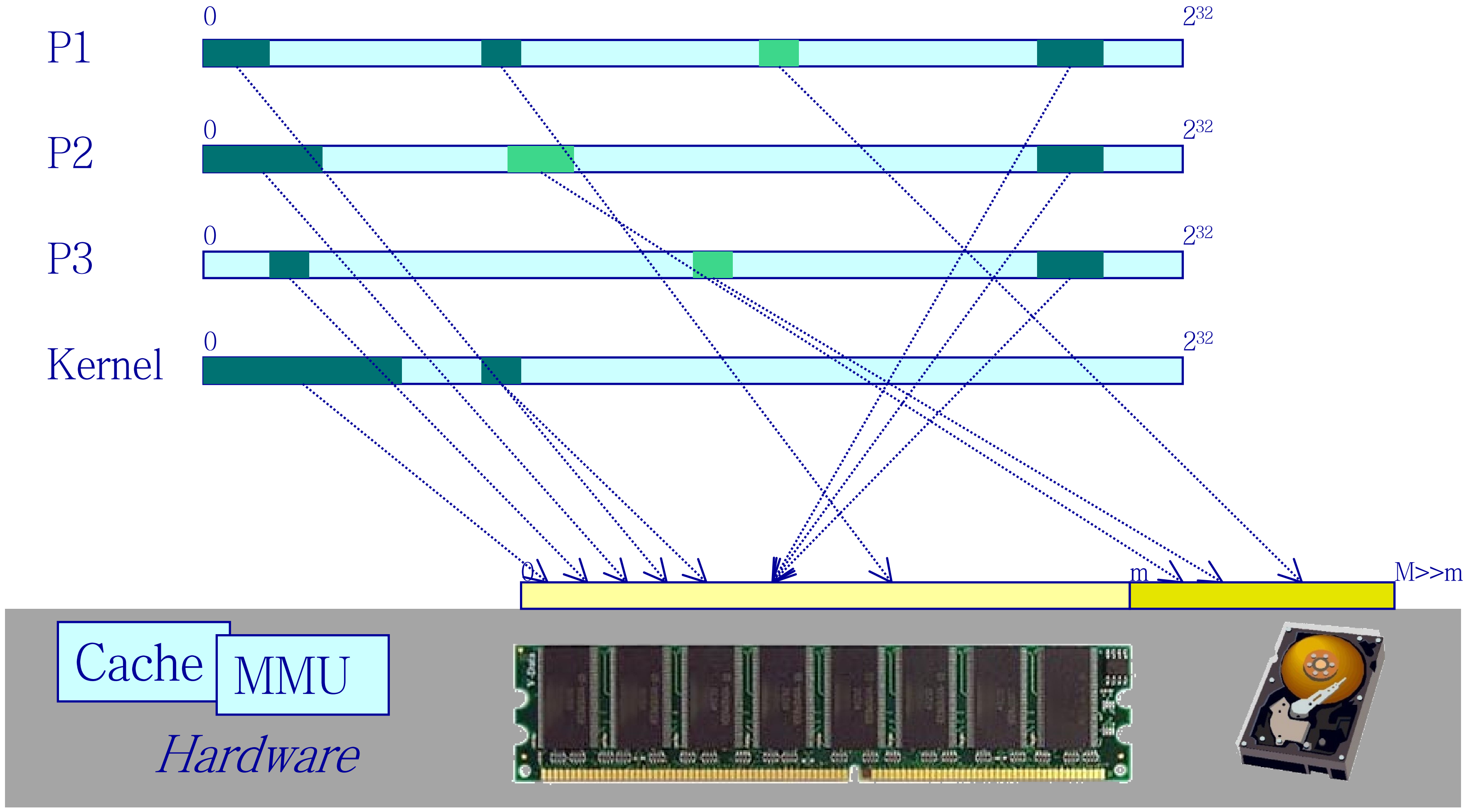
- Process views memory by logical (virtual) address space
- Only part of the logical address space needs to be in main memory at a given time
- Other parts may be in secondary storage (e.g., disk)
- The resident place may change dynamically (on-demand)
- Secondary storage can be viewed as an "extension" of physical memory

Abstraction: <u>"infinite"</u> amount of main memory!

Virtual Memory Concept

Operating System

OS abstraction: **Address Space**



OS

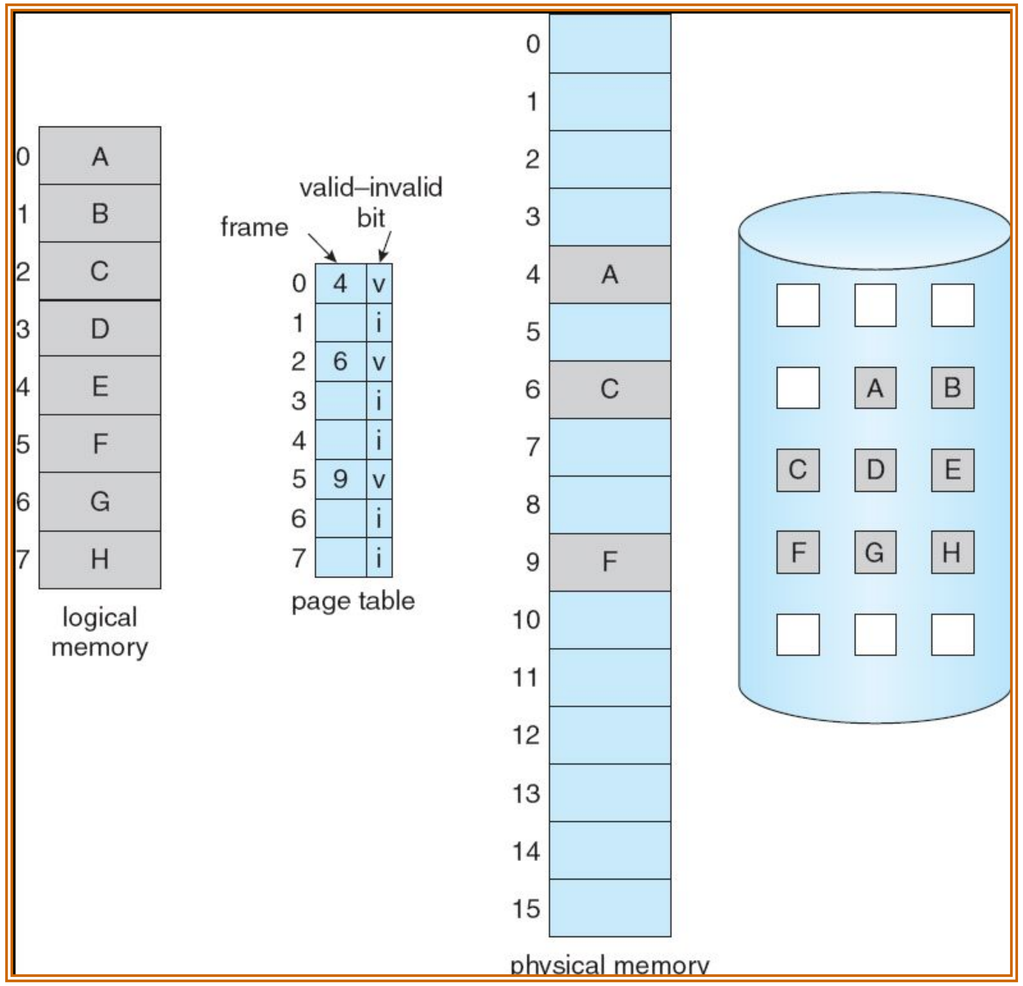
Demand Paging: a Mechanism for VM

- Based on the Paging model
  - Some pages are mapped to frames in main memory
  - Some pages are not (but in secondary storage)
  - Page table entry has a flag (resident bit) to denote which case
  - If CPU needs to access an address in a page that is not in main memory, the whole page should be loaded in memory first
- Demand paging memory management
  - OS should maintain the mapping and know where each page is stored in secondary storage

Resident Bit in Page Table

A valid/invalid bit in the page table entry

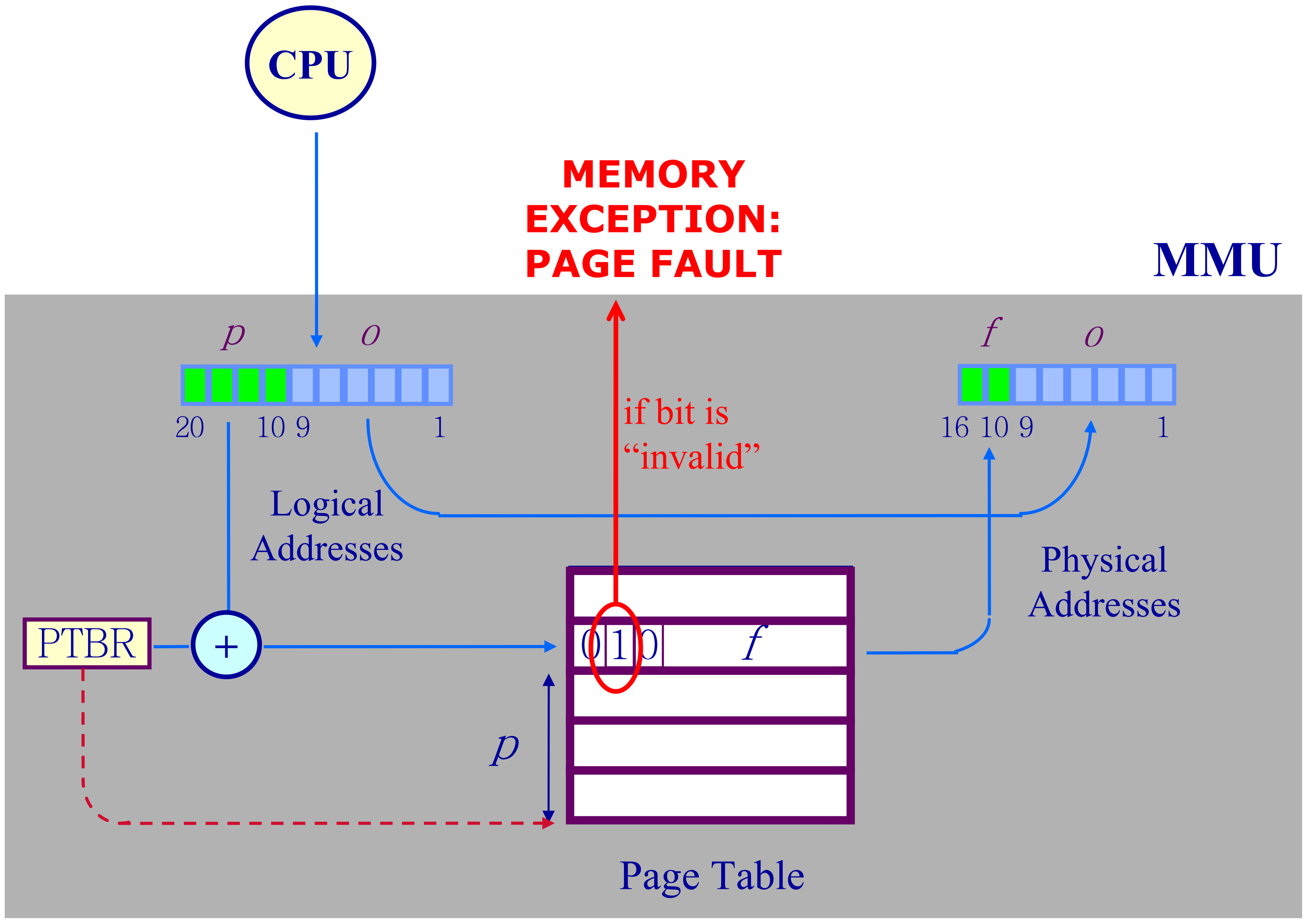
- If page is mapped to a frame in main memory, the page is resident (or the entry is "valid")
- MMU translates as usual
- Otherwise: the entry is invalid.



What if a Page is not in Main Memory

- Demand paging
  - If CPU access an address of a page that is not in memory
  - OS must load the page from secondary storage into a frame in main memory (before CPU can access the page)
- Step 1: find a frame for this page
  - Most likely there is not free frame
  - Find a frame in use and replace the content
  - Involve replacement policy (which page to replace)
  - May involve writing content to secondary storage
- Step 2: load the content of the page
  - Update the page table with new mapping (Page->Frame)
  - CPU can now access the page
- Q: How does OS know?

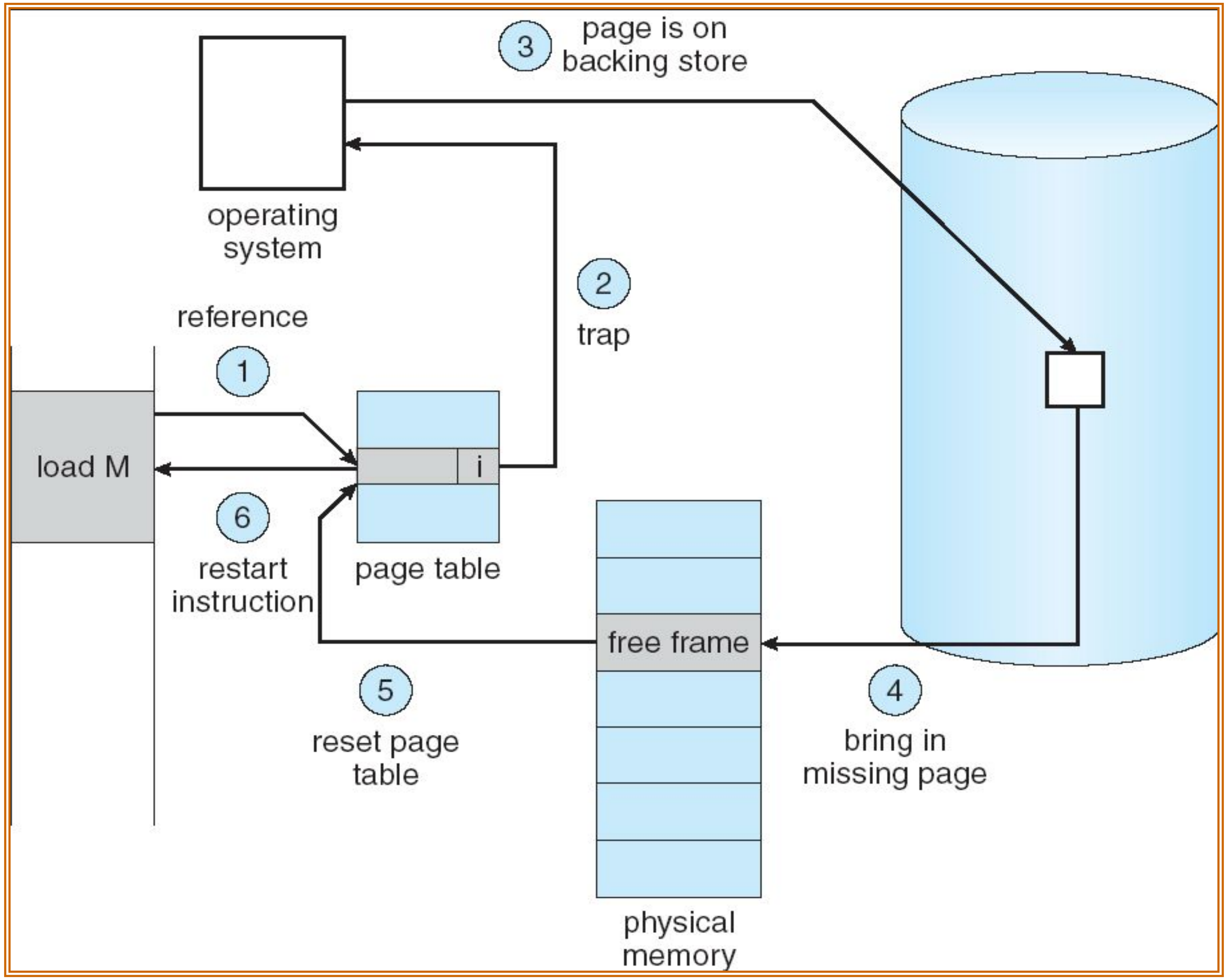
Paging Hardware Checking Resident Bit



Page Fault Handling

- CPU jumps to the exception handler (an OS kernel subroutine pre-registered to page fault exception)
  - Check if it is really a valid/legal location in logical address space
    - If not, send memory fault signal or abort process
  - Pick a page/frame to swap out (may involve write I/O)
  - Request a read I/O for the missing page (secondary storage)
  - Block the process and put in waiting state (why?)
    - Call scheduler (to schedule other processes)
- In interrupt handler (upon above I/O finishes)
  - Maps the missing page into memory (i.e., update the page table)
  - Resume the faulting process (put to ready state)

Page Fault Handling



Outline

- Principle of Locality & Address Translation
- Virtual Memory
- ● Mechanisms for Implementing VM
  - Dirty Bit
  - Backing Store
  - Virtual Memory Performance
- Local Page Replacement
- Global Page Replacement
- Belady Phenomenon

OS

Mechanisms for Implementing VM

- Demand paging
  - Based on paging
  - Bring a page into memory only when it is needed
  - Page fault: mechanism to implement demand paging
- Other mechanisms
  - Demand segmentation
  - Swapping (of the whole process)
- Replacement policy
  - Selecting which page (or segment, or process) to be replaced

Dirty Bit

- Another flag in page table entry
  - Whether the page has had write access since it is mapped to the main memory
  - If yes, the page is called a “dirty” page
- A dirty page must be written to secondary storage when it is picked for replacement
  - May slow down the access to a missing page
- A pager program may run in the background and periodically “clean” the dirty pages in memory
  - According to some strategy

Backing Store

Where to keep the unmapped pages?

- Must be easy to identify the pages in secondary storage
- Swap space (partition or file): specially formatted for storing the unmapped pages

The concept of backing store

- A page (in virtual address space) can be mapped to a location in a file (in secondary storage)
- Code segment: mapped to the executable binary file
- Dynamically loaded shared library segment: mapped to the dynamically loaded library file
- Other segment: may be implicitly mapped to swap file

Virtual Memory Performance

To understand the overhead of paging, compute the effective memory access time (EAT)

- EAT = memory access time * probability of a page hit +
  page fault service time * probability of page fault

- Example:
- Memory access time: 10 ns
- Disk access time: 5 ms
- Let p = the probability of a page fault
- Let q = the probability of a dirty page
- EAT = 10(1−p) + 5,000,000p(1+q) ?

OS

Recap of Virtual Memory Management

Key concept: Demand paging

- Load pages into memory only when a page fault occurs

Issues:

- Placement strategies
  - Place pages anywhere − no placement policy required
- Replacement strategies
  - What to do when there exist more jobs than can fit in memory
- Load control strategies
  - Determining how many jobs can be in memory at one time
  - Long-term scheduling

System Design Exercise

- Many computer architecture maintain 4 bits per TLB entries: *resident, used, dirty, read-only*
  - Will raise exception if write access to read-only page
- Suggest how you can do that in OS

Outline

- Principle of Locality & Address Translation
- Virtual Memory
- Mechanisms for Implementing VM
- ● Local Page Replacement
  - Optimal Page Replacement
  - FIFO
  - Least Recently Used (LRU)
  - Clock algorithm (Second Chance Algorithm)
  - Enhanced Clock algorithm
- Global Page Replacement
- Belady Phenomenon

Page Replacement: Concept

- Typically memory needs for concurrent processes total greater than physical memory
- With demand paging, physical memory fills quickly
- When a process faults & memory is full, some page must be swapped out
  - Handling a page fault now requires 2 disk accesses not 1!
  - Though writes are more efficient than reads (why?)
- Which page should be replaced?
  - Local replacement －Replace a page of the faulting process
  - Global replacement－Possibly replace the page of another process

OS

Evaluation methodology

Record a trace of the pages accessed by a process

- Example: (Virtual) address trace (Page Num, Offset)...
  - (3,0), (1,9), (4,1), (2,1), (5,3), (2,0), (1,9), (2,4), (3,1), (4,8)
- generates page trace
  - 3, 1, 4, 2, 5, 2, 1, 2, 3, 4 (represented as c, a, d, b, e, b, a, b, c, d)

Simulate the behavior of a page replacement and record the number of page faults generated

- fewer faults, better performance

Optimal Page Replacement (Clairvoyant)

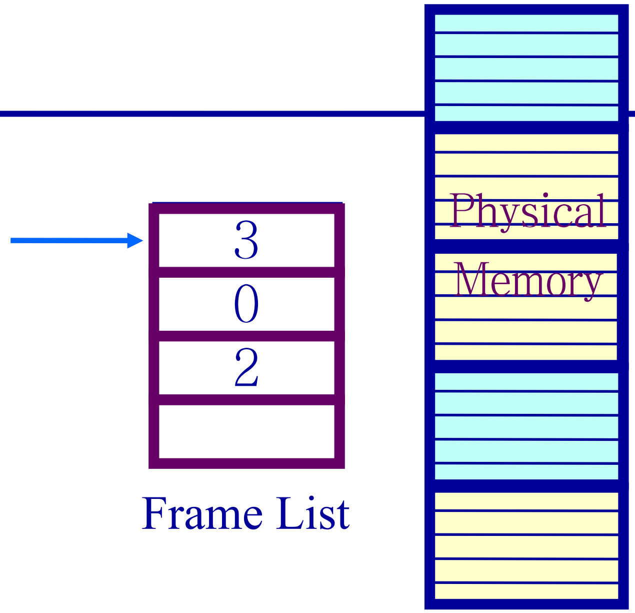
Replace the page that won't be needed for the longest time in the future

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	
	2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	
	3	*d*	*d*	*d*	*d*	*d*	→ *e*	*e*	*e*	*e*	*e*	
Faults							●					
Time page needed next							$a = 7$ $b = 6$ $c = 9$ $d = 10$					

FIFO

Simple to implement

- A single pointer suffices



Performance with 4 page frames:

- Assuming initial a->b->c->d order

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	→*e*	*e*	*e*	*e*	*e*	→*d*
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	→*a*	*a*	*a*	*a*
	2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	→*b*	*b*	*b*
	3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	→*c*	*c*
Faults							●		●	●	●	●

Least Recently Used (LRU) Page Replacement

Replace the page that hasn’t been referenced for the longest time

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	$\rightarrow e$	*e*	*e*	*e*	*e*	$\rightarrow d$
	3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	$\rightarrow c$	*c*
Faults							●				●	●
Time page last used							$a = 2$ $b = 4$ $c = 1$ $d = 3$				$a = 7$ $b = 8$ $e = 5$ $d = 3$	$a = 7$ $b = 8$ $e = 5$ $c = 9$

Implementing LRU with Stack

Maintain a “stack” of recently used pages

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	$\rightarrow$ ***e***	*e*	*e*	*e*	*e*	$\rightarrow$ ***d***
	3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	$\rightarrow$ ***c***	*c*
Faults							●				●	●

LRU page stack	*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
		c	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*
			c	*a*	*d*	*d*	*e*	*e*	*a*	*b*
				c	*a*	*a*	*d*	*d*	*e*	*a*
Page to replace					*c*				*d*	*e*

Implementing LRU with Aging Register

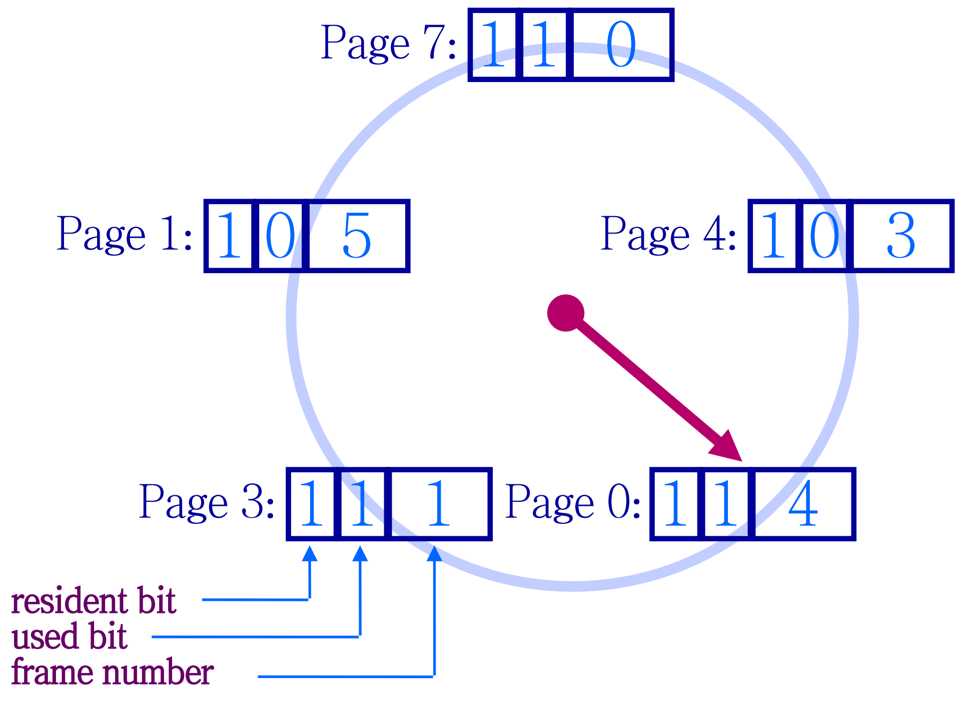
- Maintain an n-bit aging register $R = R_{n-1}R_{n-2}\cdots R_0$ for each page frame
  - On a page reference, set R_{n-1} to 1
  - Every T units of time, shift the aging vector right by one bit
  - Why not use a monotonically increasing reference count?

- Key idea:
  - Aging vector can be interpreted as a positive binary number
  - Value of R decreases periodically unless the page is referenced

- Page replacement algorithm:
  - On a page fault, replace the page with the smallest value of R

Approximate LRU: The *Clock* algorithm

- Maintain a circular list of pages resident in memory
  - Use a *clock* (or *used/referenced*) bit to track how often a page is accessed
  - The bit is set (to 1) whenever a page is referenced
- Clock hand sweeps over pages looking for one with *used* bit = 0
  - Replace pages that haven't been referenced for one complete revolution of the clock



```
func Clock_Replacement
begin
while (victim page not found) do
  if(used bit for current page = 0) then
        replace current page (& set used bit to 1)
  else
        reset used bit (to 0)
  end if
     advance clock pointer
end whileend Clock_Replacement
```

Clock Page Replacement

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	***e***	*e*	*e*	*e*	*e*	***d***
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	***a***	*a*	*a*	*a*
	3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	***c***	*c*
Faults							●		●		●	●

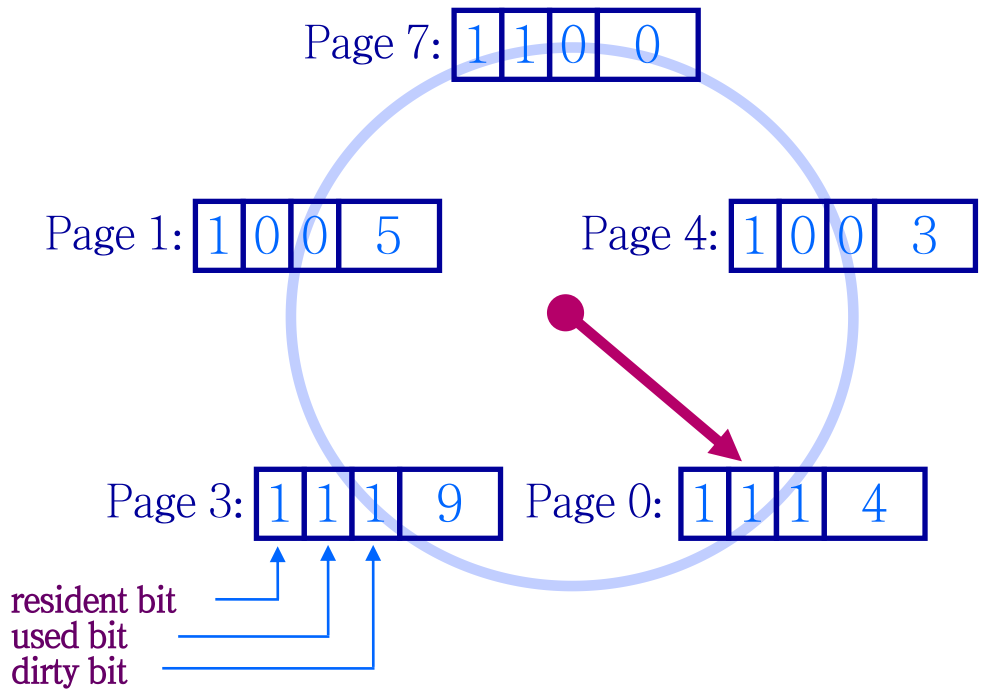
Page table entries for resident pages:

1	*a*	1	*e*	1	*e*	1	*e*	1	*e*	1	*e*	1	*d*
1	*b*	0	*b*	1	*b*	0	*b*	1	*b*	1	*b*	0	*b*
1	*c*	0	*c*	0	*c*	1	*a*	1	*a*	1	*a*	0	*a*
1	*d*	0	*d*	0	*d*	0	*d*	0	*d*	1	*c*	0	*c*

OS

Enhanced Clock algorithm

- There is a significant cost to replacing "dirty" pages
- Modify the Clock algorithm to allow dirty pages to always survive one sweep of the clock hand
  - Use both the *dirty bit* and the *used bit* to drive replacement



Enhanced Clock algorithm

Before clock sweep: *used*	Before clock sweep: *dirty*		After clock sweep: *used*	After clock sweep: *dirty*
0	0	→	*replace page*	
0	1	→	0	0
1	0	→	0	0
1	1	→	0	1

Enhanced Clock algorithm

Time			0	1	2	3	4	5	6	7	8	9	10
Requests				*c*	a^W	*d*	b^W	*e*	*b*	a^W	*b*	*c*	*d*
Page Frames	0		*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
	1		*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	***d***
	2		*c*	*c*	*c*	*c*	*c*	***e***	*e*	*e*	*e*	*e*	*e*
	3		*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	***c***	*c*
Faults								●				●	●

Page table entries for resident pages:

10	*a*	11	*a*	00	*a**	00	*a*	11	*a*	11	*a*	00	*a**
10	*b*	11	*b*	00	*b**	10	*b*	10	*b*	10	*b*	10	*d*
10	*c*	10	*c*	10	*e*	10	*e*	10	*e*	10	*e*	00	*e*
10	*d*	10	*d*	00	*d*	00	*d*	00	*d*	10	*c*	00	*c*

OS Outline

- Principle of Locality & Address Translation
- Virtual Memory
- Mechanisms for Implementing VM
- Local Page Replacement
- ● Global Page Replacement
  - Working Set Page Replacement
  - Page-Fault-Frequency Page Replacement
  - Load Control
- Belady Phenomenon

OS

The Problem With Local Page Replacement

FIFO page replacement: Assuming initial a–>b–>c order

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
Requests		*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*

Page Frames														
	0	*a*	*a*	*a*	*a*	***d***	*d*	*d*	***c***	*c*	*c*	***b***	*b*	*b*
	1	*b*	*b*	*b*	*b*	*b*	***a***	*a*	*a*	***d***	*d*	*d*	***c***	*c*
	2	*c*	*c*	*c*	*c*	*c*	*c*	***b***	*b*	*b*	***a***	*a*	*a*	***d***
Faults						唷	唷	唷	唷	唷	唷	唷	唷	唷

Page Frames														
	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*
	3	–				***d***	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*
Faults						唷								

OS

Introducing Global Page Replacement

- Local page replacement
  - LRU — Ages pages based on when they were last used
  - FIFO — Ages pages based on when they’re brought into memory
- Towards global page replacement ... with variable number of page frames allocated to processes
  - The principle of locality argues that a fixed number of frames should work well (over short intervals).
  - Programs need different amounts of memory at different times.
  - allow a process’s memory allocation to grow (and shrink) over time.
  - determine what this number of frames is (what we’ll later call the “working set”).

Optimal Replacement with Variable Frames

Replace a page that is not referenced in the *next* τ accesses.

Example: $\tau = 4$

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory	Page *a*	🙂 $t = 0$	🚫								😊	🚫
	Page *b*					😊	🚫					
	Page *c*		😊	🙂	🙂	🙂	🙂	🙂	🙂	🚫		
	Page *d*	🙂 $t = -1$	🙂	🙂	🙂	🚫						😊
	Page *e*							😊	🙂	🙂	🚫	
Faults			🛑			🛑		🛑			🛑	🛑

OS

The Working Set Model

- Assume recently referenced pages are likely to be referenced again soon…
- ... and only keep those pages recently referenced in memory (called **the working set**)
    - Thus pages may be removed even when no page fault occurs
    - The number of frames allocated to a process will vary over time
- A process is allowed to execute only if its working set fits into memory
    - The working set model performs implicit load control

Working Set Page Replacement

Keep track of the last τ references

- The pages referenced during the last τ memory accesses are the working set, τ is called the *window size.* Example: τ = 4 references:

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory: Page *a*	☺ $t = 0$	☺	☺	☺	⊘					😊	☺
Page *b*					😊	☺	☺	☺	⊘		
Page *c*		😊	☺	☺	☺	☺	☺	☺	☺	☺	☺
Page *d*	☺ $t = -1$	☺	☺	☺	☺	☺	☺	⊘			😊
Page *e*	☺ $t = -2$	☺	⊘				😊	☺	☺	☺	☺
Faults		⬣			⬣		⬣			⬣	⬣

Page-Fault-Frequency Page Replacement

An alternate working set computation

Explicitly attempt to minimize page faults

- When page fault frequency is high — increase working set
- When page fault frequency is low — decrease working set

<u>Algorithm</u>:

```
Keep track of the rate at which faults occur
    When a fault occurs, compute the time since the last page fault
        Record the time, $t_{last}$, of the last page fault
    If the time between page faults is “large” then reduce the working set
        If $t_{current} - t_{last} > g$ , then remove from memory all pages not referenced in
        $[t_{last},\ t_{current}]$
    If the time between page faults is “small” then increase working set
        If $t_{current} - t_{last} \le g$ , then add faulting page to the working set
```

Page-Fault-Frequency Page Replacement

Example: window size = 2

- If $t_{current} - t_{last} > 2$, remove pages not referenced in $[t_{last}, t_{current}]$ from the working set
- If $t_{current} - t_{last} \le 2$, just add faulting page to the working set

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory: Page *a*	☺	☺	☺	☺	⊘					😊	☺
Page *b*					😊	☺	☺	☺	☺	⊘	
Page *c*		😊	☺	☺	☺	☺	☺	☺	☺	☺	☺
Page *d*	☺	☺	☺	☺	☺	☺	☺	☺	☺	⊘	😊
Page *e*	☺	☺	☺	☺	⊘		😊	☺	☺	☺	☺
Faults		●			●		●			●	●
$t_{cur} - t_{last}$		1			3		2			3	1

Load Control: Fundamental tradeoff

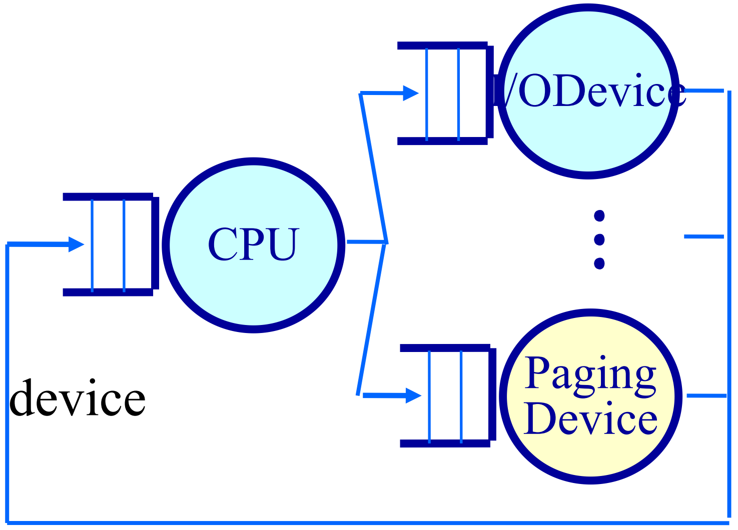
High multiprogramming level

$$MPL_{max} = \frac{\textit{number of page frames}}{\textit{minimum number of frames required for a process to execute}}$$

- Low paging overhead
  - MPL_{min} = 1 process
- Issues
  - What criterion should be used to determine when to increase or decrease the *MPL*?
  - Which task should be swapped out if the *MPL* must be reduced?

OS

Load Control: How *not* to do it



Base load control on CPU utilization?

Assume memory is nearly full

A chain of page faults occur

- A queue of processes forms at the paging device
- CPU utilization falls
- Operating system increases MPL
- New processes fault, taking memory away from existing processes
- CPU utilization goes to 0, the OS increases the MPL further...

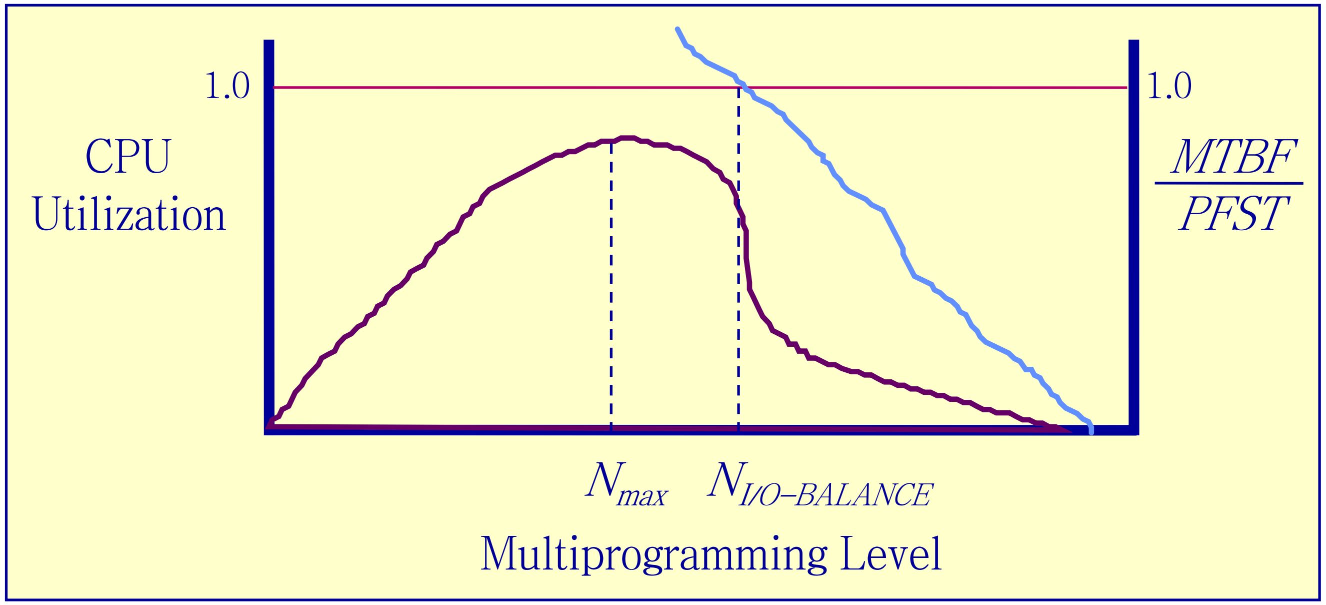
System is *thrashing* — spending all of its time paging

Load Control: Thrashing

Thrashing can be ameliorated by *local* page replacement

Better criteria for load control: Adjust MPL so that:

- *mean time between page faults (MTBF) = page fault service time (PFST)*
- H WS_i *= size of memory*



Outline

- Principle of Locality & Address Translation
- Virtual Memory
- Mechanisms for Implementing VM
- Local Page Replacement
- Global Page Replacement
- ● Belady Phenomenon

OS

Belady Phenomenon

FIFO Page Replacement

Access Sequence : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Frame Size: 3 Page Fault: 9

FIFO	1	2	3	4	1	2	5	1	2	3	4	5
Tail	1	2	3	4	1	2	5	5	5	3	4	4
		1	2	3	4	1	2	2	2	5	3	3
Head			1	2	3	4	1	1	1	2	5	5
PF	X	X	X	X	X	X	X			X	X	

Belady Phenomenon

FIFO Page Replacement

Access Sequence : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Frame Size: 4 Page Fault: 10

FIFO	1	2	3	4	1	2	5	1	2	3	4	5
Tail	1	2	3	4	4	4	5	1	2	3	4	5
		1	2	3	3	3	4	5	1	2	3	4
			1	2	2	2	3	4	5	1	2	3
Head				1	1	1	2	3	4	5	1	2
PF	X	X	X	X			X	X	X	X	X	X

OS

Belady Phenomenon

LRU Page Replacement

Frame Size: 3 Page Fault: 10

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	2	3	4	1	2	5	1	2	3
	2	2	3	4	1	2	5	1	2	3	4
		3	4	1	2	5	1	2	3	4	5
x	**x**	**x**	**x**	**x**	**x**	**x**	**v**	**v**	**x**	**x**	**x**

Frame Size: 4 Page Fault: 8

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	2	3	4	4	4	5	1	2
	2	2	2	3	4	1	2	5	1	2	3
		3	3	4	1	2	5	1	2	3	4
			4	1	2	5	1	2	3	4	5
x	**x**	**x**	**x**	**v**	**v**	**x**	**v**	**v**	**x**	**x**	**x**

How about Clock /Second Chance Page Replacement ?
Why LRU Page Replacement has no Belady Phenomenon?