Review 03

Yajin Zhou (http://yajin.org)

Zhejiang University

Acknowledgement: Some slides are based on Zhi Wang (FSU)'s slides.

08: Deadlock

The Deadlock Problem

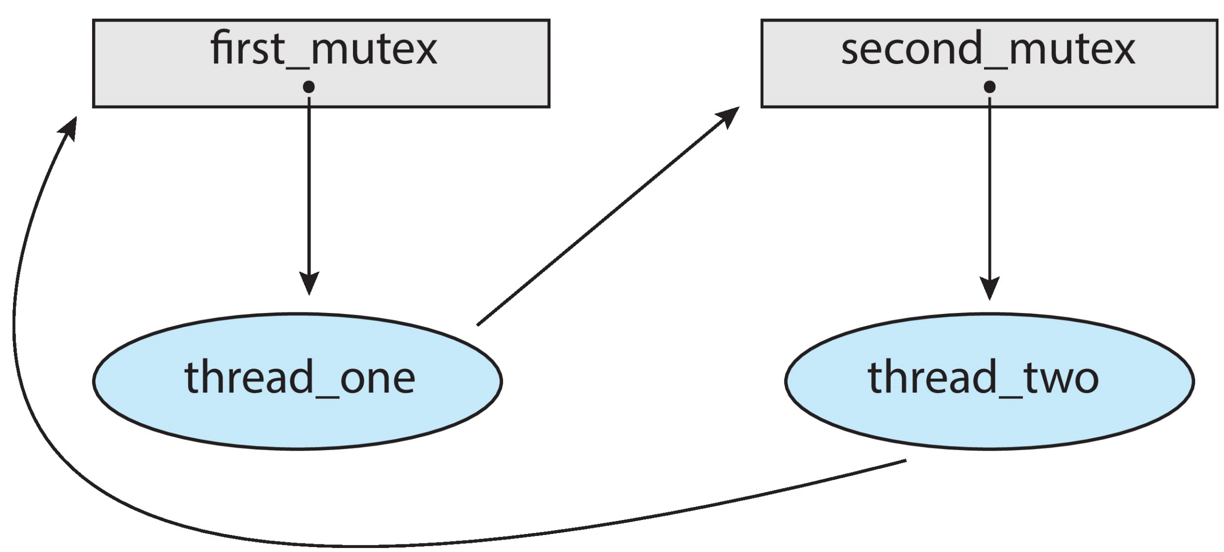
- **Deadlock**: a set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Examples:
  - a system has 2 disk drives, P_1 and P_2 each hold one disk drive and each needs another one
  - semaphores A and B, initialized to 1

P_1	P_2
wait (A);	wait(B)
wait (B);	wait(A)

Deadlock in program

- Deadlock is possible if thread 1 acquires **first_mutex** and thread 2 acquires **second_mutex**. Thread 1 then waits for **second_mutex** and thread 2 waits for **first_mutex**.

- Can be illustrated with a **resource allocation graph**:



Four Conditions of Deadlock

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only **voluntarily** by the process holding it, after it has completed its task
- **Circular wait**: there exists a set of waiting processes $\{P_0, P_1, \ldots, P_n\}$
  - P_0 is waiting for a resource that is held by P_1
  - P_1 is waiting for a resource that is held by P_2 …
  - P_{n-1} is waiting for a resource that is held by P_n
  - P_n is waiting for a resource that is held by P_0

How to Handle Deadlocks

- How?
  - **Prevention: that the possibility of deadlock is excluded!!!**
  - **Avoidance**
  - **Deadlock detection and recovery**
  - **Ignore the problem** and pretend deadlocks never occur in the system

Deadlock Prevention

- How to prevent **mutual exclusion**
  - not required for sharable resources
  - must hold for non-sharable resources
- How to prevent **hold and wait**
  - whenever a process requests a resource, it doesn't hold any other resources
    - require process to request ***all*** its resources before it begins execution
    - allow process to request resources only when the process has none
  - low resource utilization; starvation possible

Deadlock Prevention

- How to handle **no preemption**
  - if a process requests a resource not available
    - release all resources currently being held
    - preempted resources are added to the list of resources it waits for
    - process will be restarted only when it can get all waiting resources
- How to handle **circular wait**
  - **impose a total ordering of all resource types**
  - require that each process requests resources in an increasing order
  - **Many operating systems adopt this strategy for some locks.**

Deadlock Avoidance

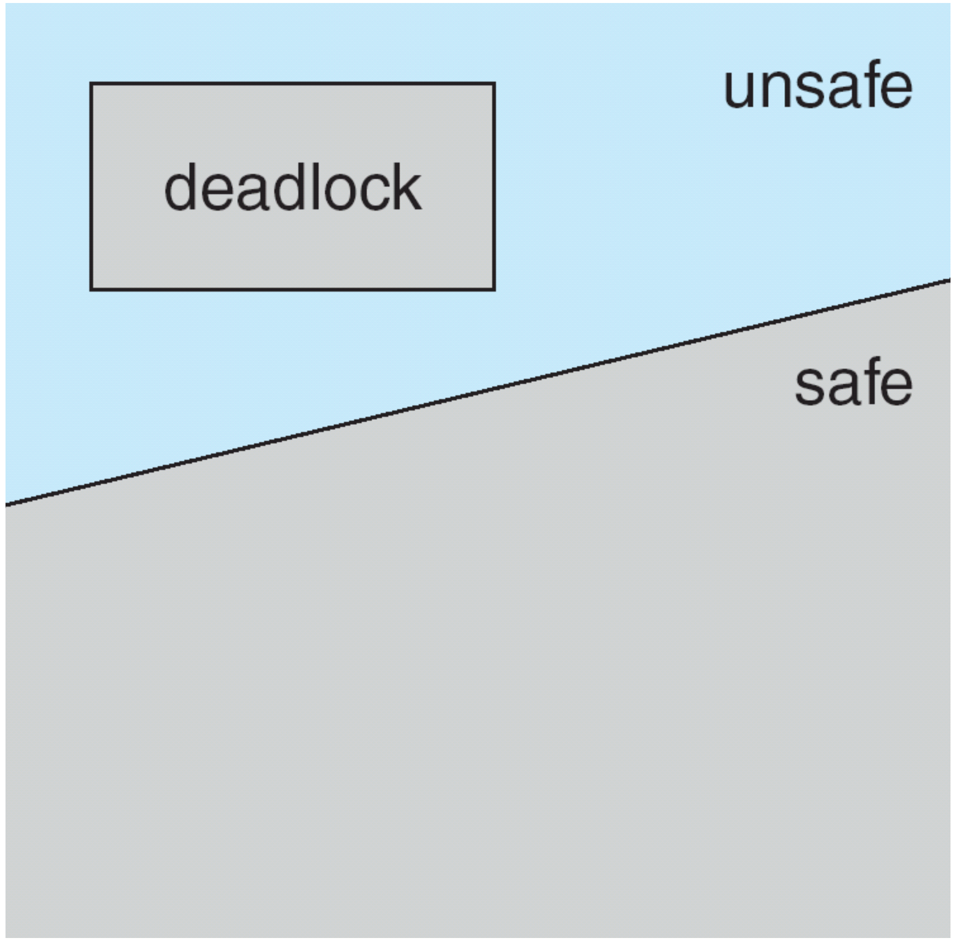
- **Dead avoidance: require extra information about how resources are to be requested**
  - **Is this requirement practical**?
- Each process declares a **max** number of resources it may need
- Deadlock-avoidance algorithm ensure there can **never be a circular-wait condition**
- Resource-allocation state:
  - the number of **available** and **allocated** resources
  - the **maximum demands** of the processes

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a **safe state**:
  - there exists a **sequence <P_1, P_2, …, P_n>** of all processes in the system
  - for each P_i, resources that P_i can still request can be satisfied by currently **available resources** + **resources held by all the P_j, with $j < i$**
- **Safe state can guarantee no deadlock**
  - if P_i's resource needs are not immediately available:
    - wait until all P_j have finished ($j < i$)
    - when P_j ($j < i$) has finished, P_i can obtain needed resources,
  - when P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

- If a system is in **safe state** ➠ **no deadlocks**
- If a system is in **unsafe state** ➠ **possibility of deadlock**
- **Deadlock avoidance** ➠ ensure a system **never enters an unsafe state**



Example

- Resources: 12

	Maximum Needs	Current Needs	Available	Extra need
T_0	10	5	**3**	**5**
T_1	4	2		**2**
T_2	9	2		**7**

- Safe sequences: T1 T0 T2
  - T1 gets and return (5 in total), T0 gets all and returns (10 in total) and then T2
- What if we allocate 1 more for T2?

Example

- Resources: 12

	Max need	Current have	available	Extra need
P0	10	5	2	5
P1	4	2		2
P2	9	3		6

- p1 gets and return (**4 in total**), P0 P2 have to wait …

Deadlock Detection

- Allow system to enter deadlock state, but detect and recover from it
- Detection algorithm and recovery scheme

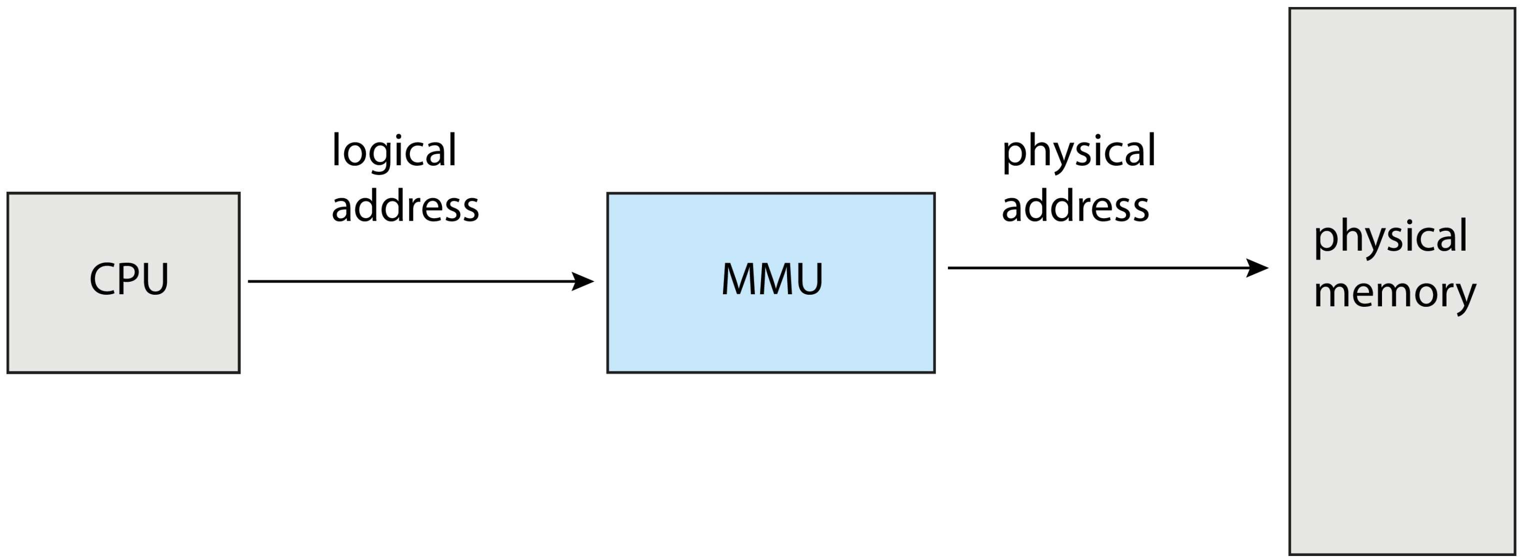
09: Main Memory

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a **separate physical address** space is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as virtual address
  - **Physical address** – address seen by the memory unit
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address



- Many methods possible, covered in the rest of this chapter

Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

Memory Allocation

- How to satisfy a request of size *n* from a list of free memory blocks?
  - **first-fit**: allocate from the first block that is big enough
  - **best-fit**: allocate from the smallest block that is big enough
    - must search entire list, unless ordered by size
    - produces the smallest leftover hole
  - **worst-fit**: allocate from the largest hole
    - must also search entire list
    - produces the largest leftover hole
- **Fragmentation** is big problem for all three methods
  - first-fit and best-fit usually perform better than worst-fit

Fragmentation

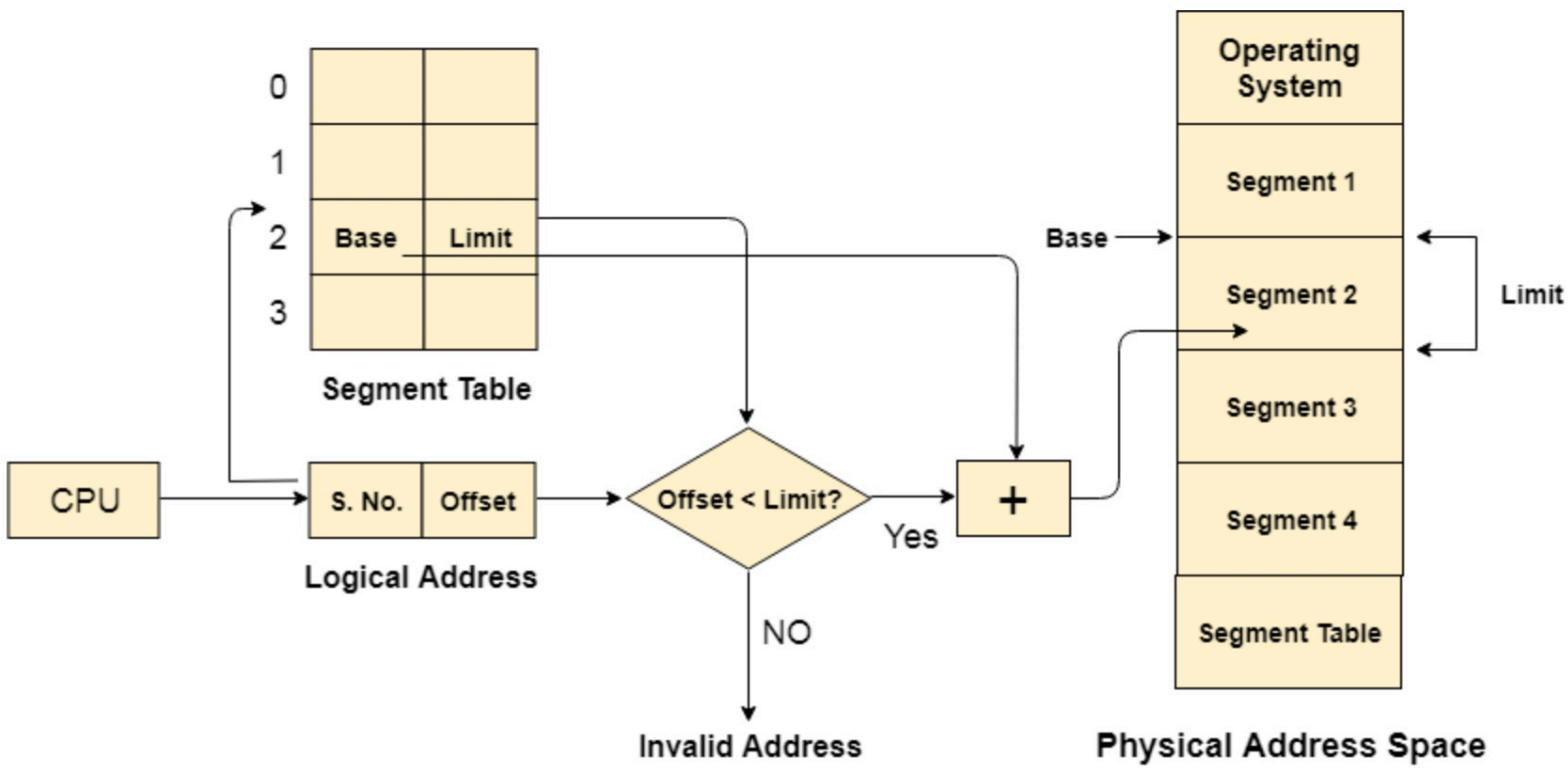
- **External fragmentation**
  - unusable memory between allocated memory blocks
    - **total amount** of free memory space **is larger than a request**
    - the request cannot be fulfilled because the free memory is not **contiguous**
  - external fragmentation can be reduced by **compaction**
    - shuffle memory contents to place all free memory in one large block
    - program needs to be **relocatable** at runtime
    - Performance overhead, timing to do this operation
  - Another solution: **paging**
  - 50-percent rule: N allocated blocks, 0.5N will be lost due to fragmentation. 1/3 is unusable!

Fragmentation

- **Internal fragmentation**
  - memory allocated may be larger than the requested size
  - this size difference is memory *internal to a partition*, but not being used
  - Example: free space 18464 bytes, request 18462 bytes
- Sophisticated algorithms are designed to avoid fragmentation
  - none of the first-/best-/worst-fit can be considered sophisticated

Segment

- base address + offset



Paging

- Physical address space of a process can be **noncontiguous**; process is allocated physical memory whenever the latter is available
    - Avoids **external fragmentation** **-> avoid for compacting**
    - Avoids problem of varying sized memory chunks
- Basic methods
    - Divide physical memory into fixed-sized blocks called **frames**
        - Size is power of 2, between 512 bytes and 16 Mbytes
    - Divide logical memory into blocks of same size called **pages**
    - Keep track of all free frames
    - To run a program of size **N** pages, need to find **N** free frames and load program
    - Set up a **page table** to translate logical to physical addresses
    - Backing store likewise split into pages
    - **Still have Internal fragmentation**

Paging: Address Translation

- A logical address is divided into:
  - **page number** (p)
    - used as an index into a page table
    - **page table entry contains the corresponding physical frame number + plus v bit, permissions**
  - **page offset** (d)
    - offset within the page/frame
    - combined with frame number to get the physical address

page number	page offset
p	d
m - n *bits*	n *bits*

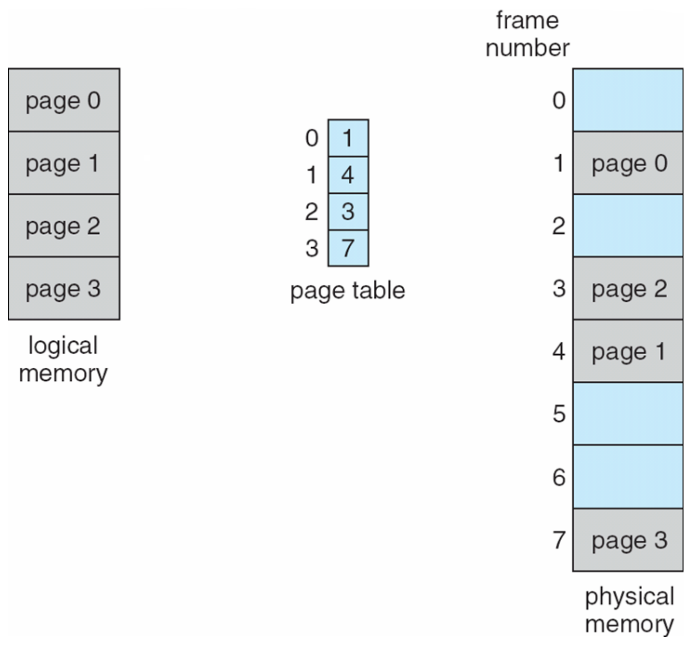
m bit logical address space, n bit page size

Hardware Support: Simplest Case

- Page table is in a set of dedicated registers
  - Advantages: very efficient - access to register is fast
  - Disadvantages: the table size is very small, and the context switch need to save and restore these registers

Hardware Support: Alternative Way

- One big page table maps logical address to physical address
  - the page table should be **kept in main memory**
  - **page-table base register** (**PTBR**) points to the page table
    - **does PTBR contain physical or logical address?**
  - **page-table length register** (**PTLR**) indicates the size of the page table
- Every data/instruction access requires **two memory accesses**
  - one for the page table and one for the data / instruction
  - CPU can cache the translation to avoid one memory access (**TLB**)



TLB

- TLB (**translation look-aside buffer**) caches **the address translation for the current process (if without ASID)**
  - if page number is in the TLB, no need to access the page table
  - if page number is not in the TLB, need to replace one TLB entry
  - TLB usually use a fast-lookup hardware cache called **associative memory**
  - TLB is usually small, 64 to 1024 entries
- Use with page table
  - TLB contains a few page table entries
  - Check whether page number is in TLB
    - If -> frame number is available and used
    - If not -> **TLB miss**. access page table and then fetch into TLB
      - TLB flush: TLB entries are full
      - TLB wire down: TLB entries should not be flushed

Structure of Page Table

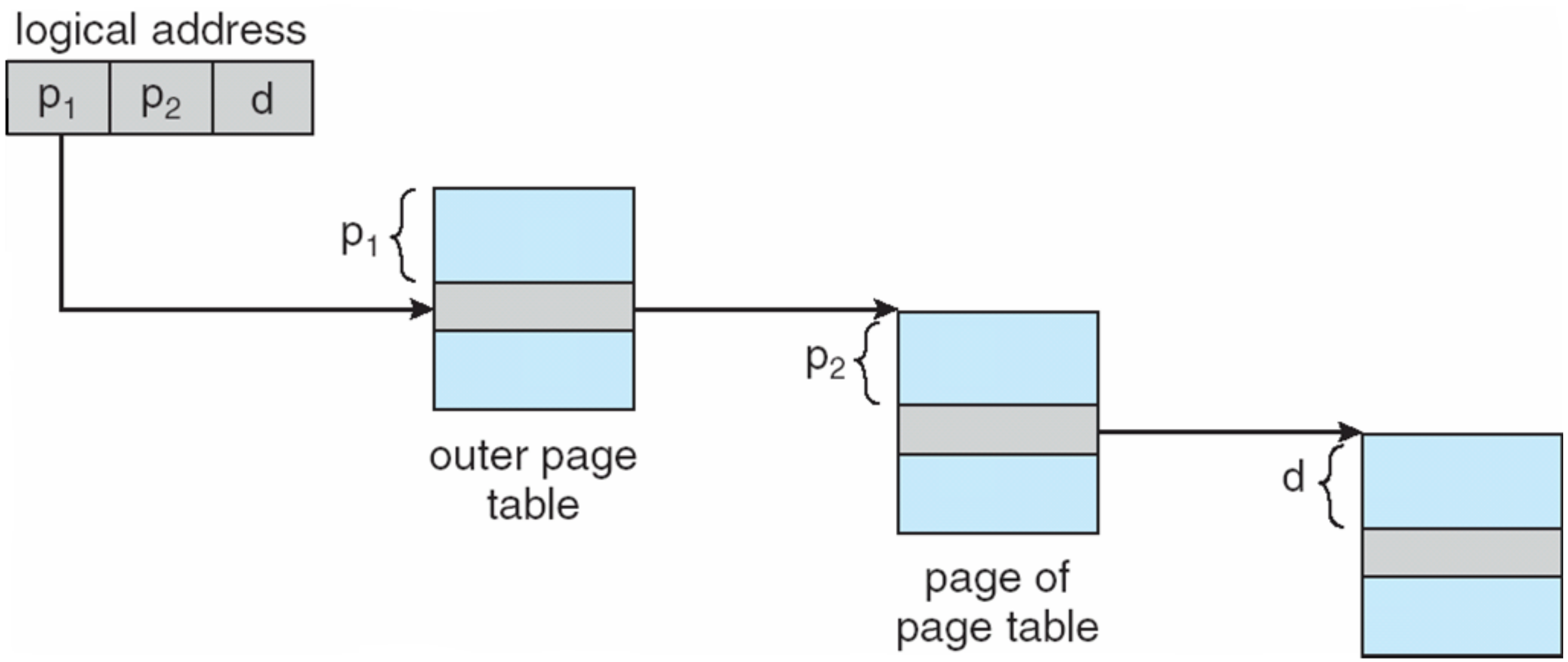
- One-level page table can consume lots of memory for page table
  - e.g., 32-bit logical address space and 4KB page size
    - page table would have 1 million entries (2^{32} / 2^{12})
    - if each entry is 4 bytes ➔ 4 MB of memory for page table alone
  - each process requires its own page table
  - page table must be **physically contiguous**
- To reduce memory consumption of page tables:
  - **hierarchical page table**
  - **hashed page table**
  - **inverted page table**

Two-Level Paging

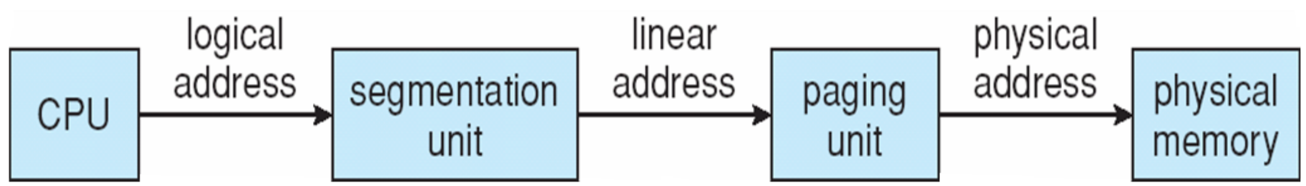
- A logical address is divided into:
  - a **page directory number** (first level page table)
  - a **page table number** (2nd level page table)
  - a **page offset**
- Example: 2-level paging in 32-bit Intel CPUs
  - 32-bit address space, 4KB page size
  - 10-bit page directory number, 10-bit page table number
  - each page table entry is 4 bytes, one frame contains 1024 entries (2^{10})

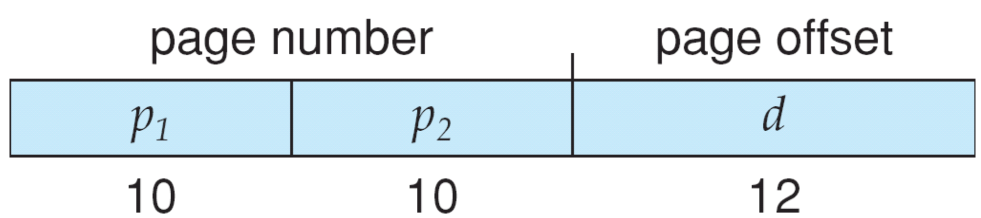
p_1	p_2	d
10	10	12

Address-Translation Scheme

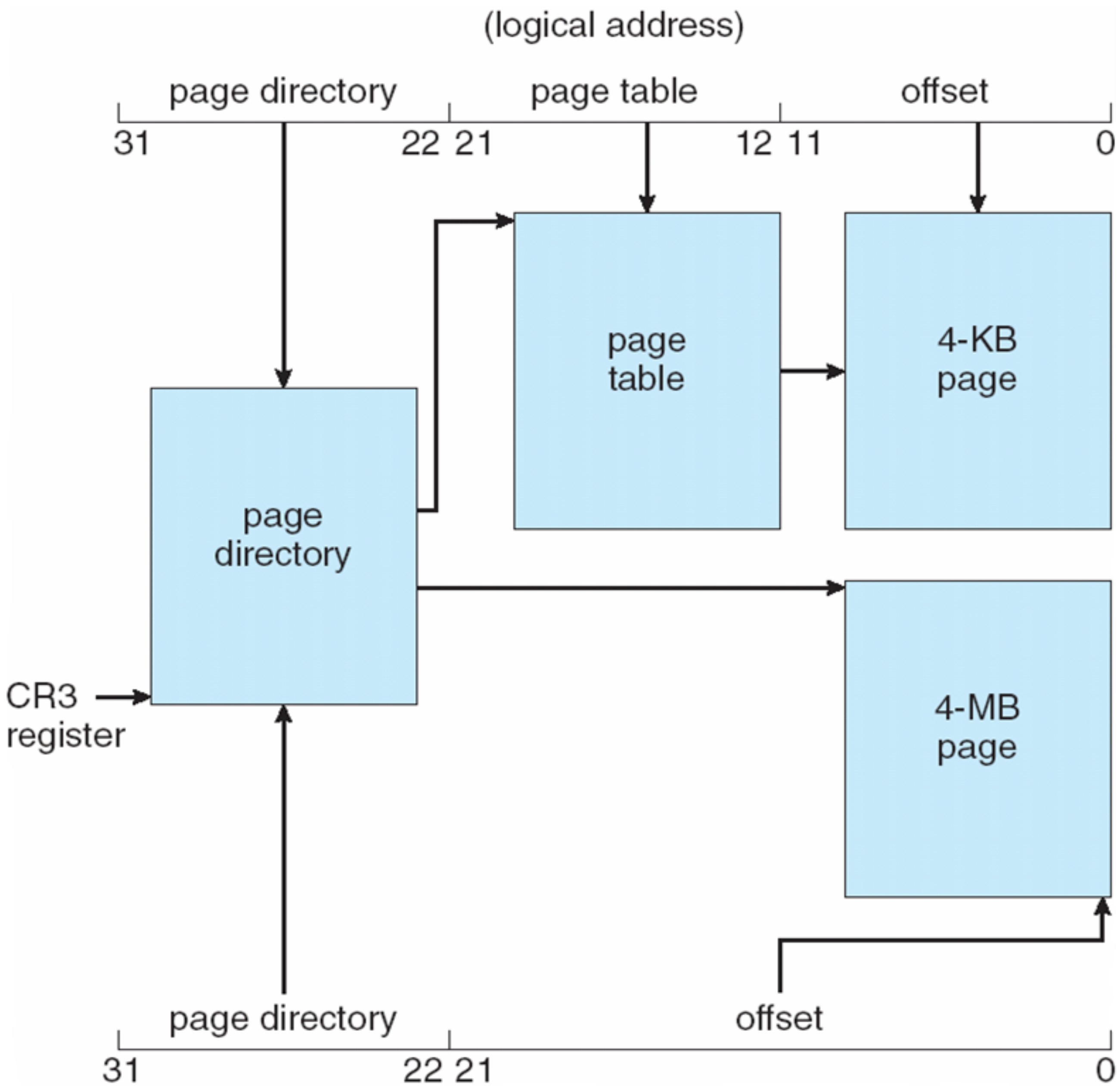


Logical to Physical Address Translation in IA-32





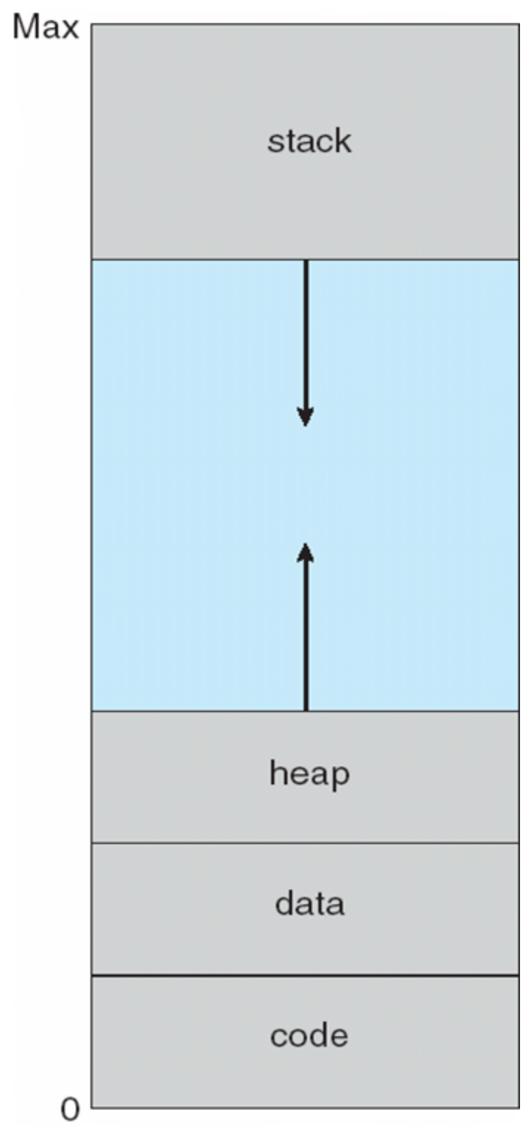
Intel IA-32 Paging Architecture



10: Virtual Memory

Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- **Shared memory by mapping pages read-write into virtual address space**
- Pages can be shared during fork(), speeding process creation: COW

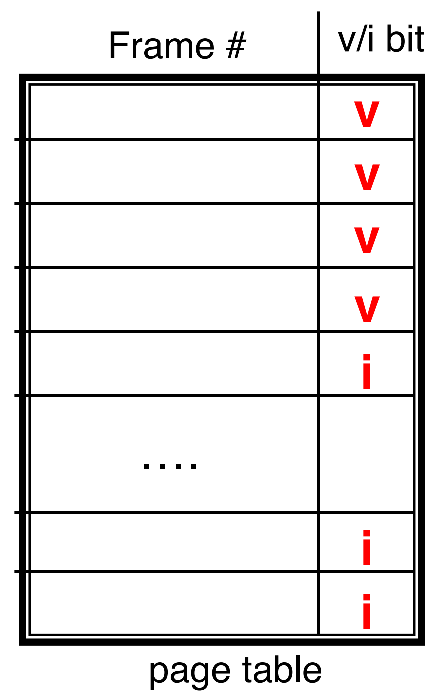


Demand Paging

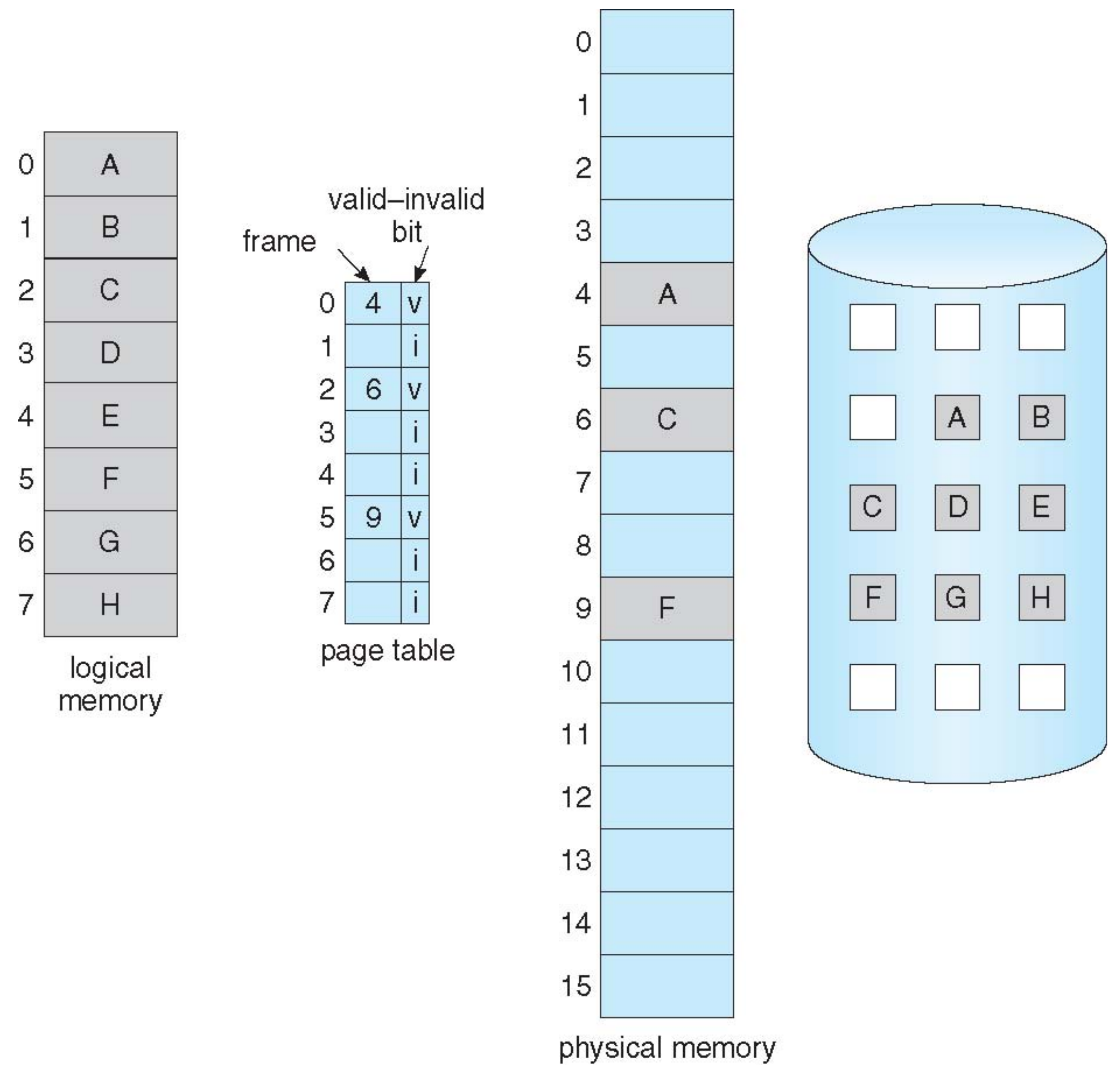
- **Demand paging** brings a page into memory **only when it is accessed**
  - if page is invalid ➠ abort the operation
  - if page is valid but not in memory ➠ bring it to memory via swapping
  - no unnecessary I/O, less memory needed, faster response, more apps
- **Lazy swapper**: never **swaps a page in** memory unless it will be needed
  - the swapper that deals with pages is also caller a pager
- **Pre-Paging**: pre-page all or some of pages a process will need, before they are referenced
  - it can reduce the number of page faults during execution
  - if pre-paged pages **are unused**, I/O and memory was wasted
    - although it reduces page faults, total I/O# likely is higher

Valid-Invalid Bit

- Each page table entry has a valid–invalid (present) bit
  - *V* ➠ in memory (memory is resident), *I* ➠ not-in-memory
  - initially, valid–invalid bit is set to *i* on all entries
  - during address translation, if the entry is invalid, it will trigger a **page fault**
- Example of a page table snapshot:



Page Table (Some Pages Are Not in Memory)



Page Fault

- First reference to a non-present page will trap to kernel: **page fault**
- **Page fault:**
  - **Page is not in memory**
  - **Illegal memos address in the memory space**
- Operating system looks at memory mapping to decide:
  - **invalid reference** ➟ deliver an exception to the process
  - **valid but not in memory** ➟ swap in
- get an empty physical frame
- swap page into frame via disk operation
- set page table entry to indicate the page is now in memory
- restart the instruction that caused the page fault

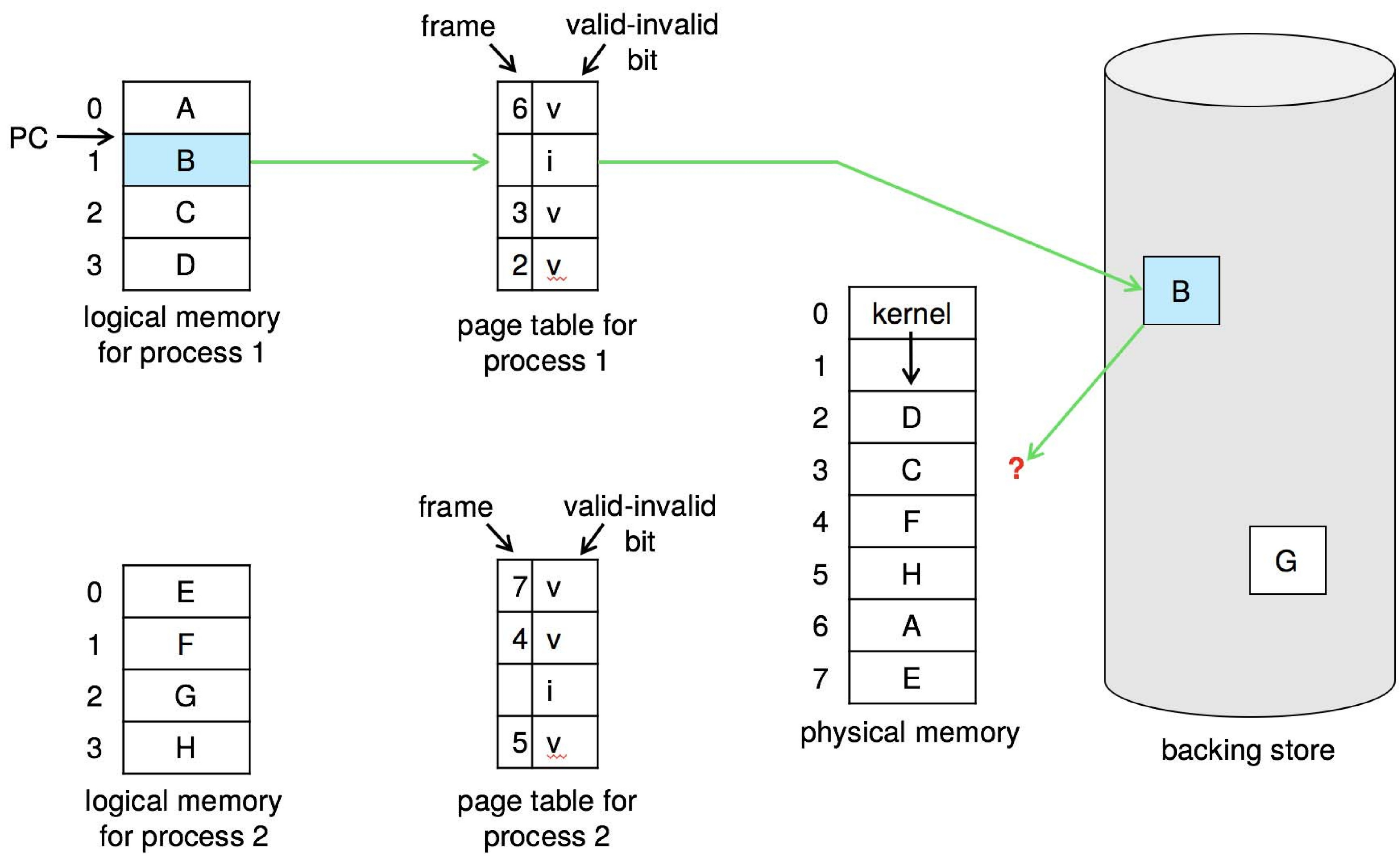
What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- **Page replacement** – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

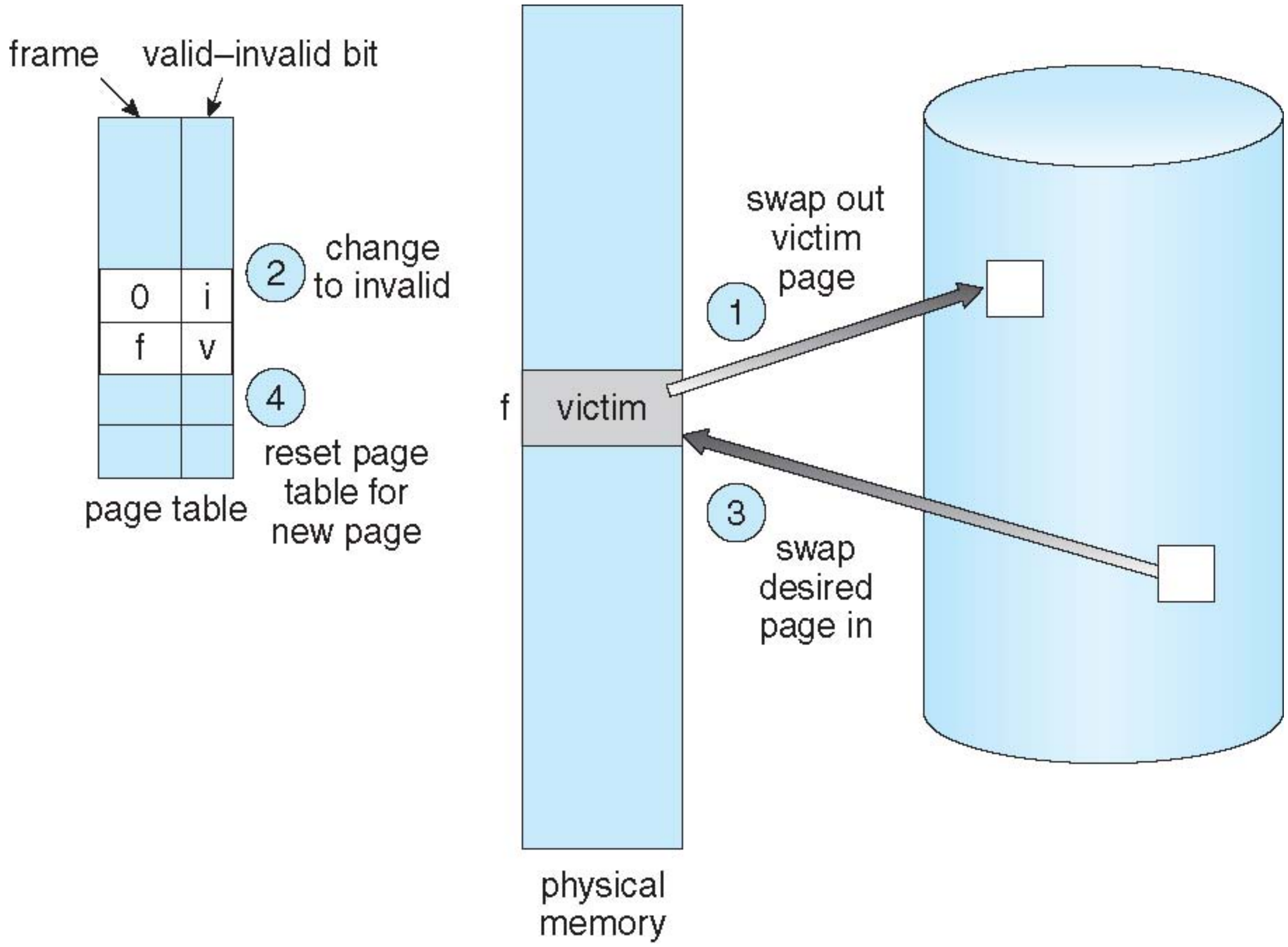
Page Replacement

- Memory is an important resource, system may run out of memory
- To prevent out-of-memory, swap out some pages
  - page replacement usually is a part of the page fault handler
  - policies to select victim page require careful design
    - need to reduce overhead and avoid **thrashing**
  - use modified (dirty) bit to reduce number of pages to swap out
    - only modified pages are written to disk
  - select some processes to kill (last resort)
- 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



Page Replacement

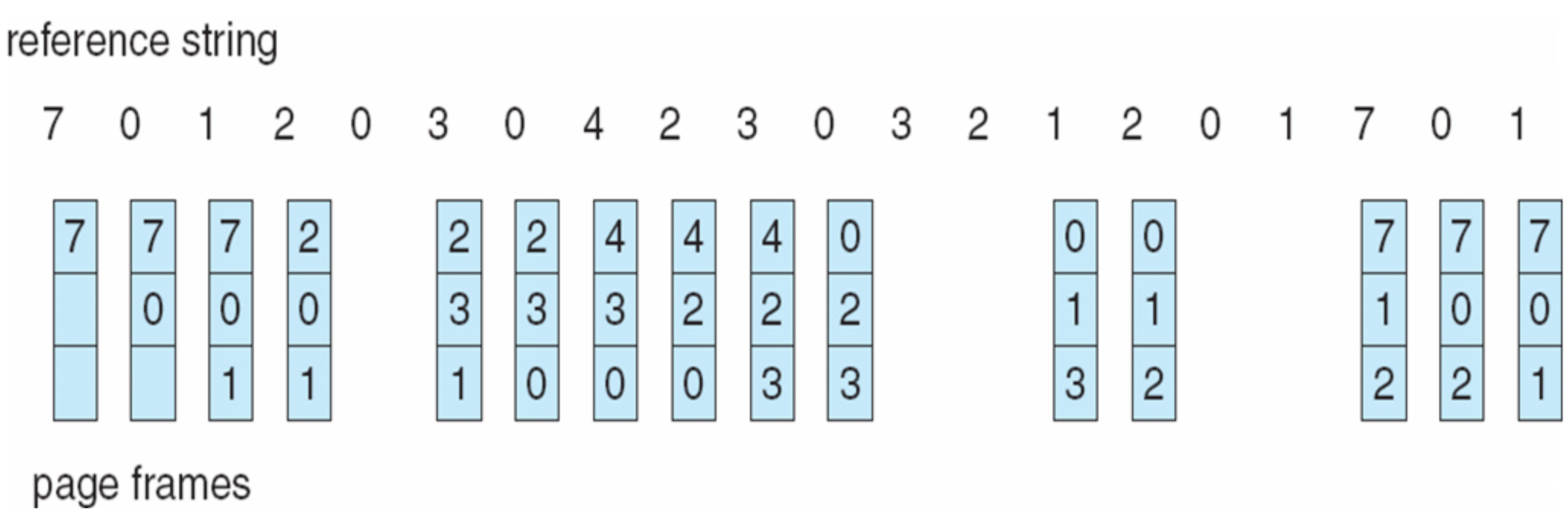


Page Replacement Algorithms

- Page-replacement algorithm should have lowest page-fault rate on both first access and re-access
  - **FIFO**, **optimal**, **LRU**, **LFU**, **MFU**…
- To evaluate a page replacement algorithm:
  - run it on a particular string of memory references (reference string)
    - string is just page numbers, not full addresses
  - compute the number of page faults on that string
    - repeated access to the same page does not cause a page fault
  - in all our examples, the reference string is 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

First-In-First-Out (FIFO)

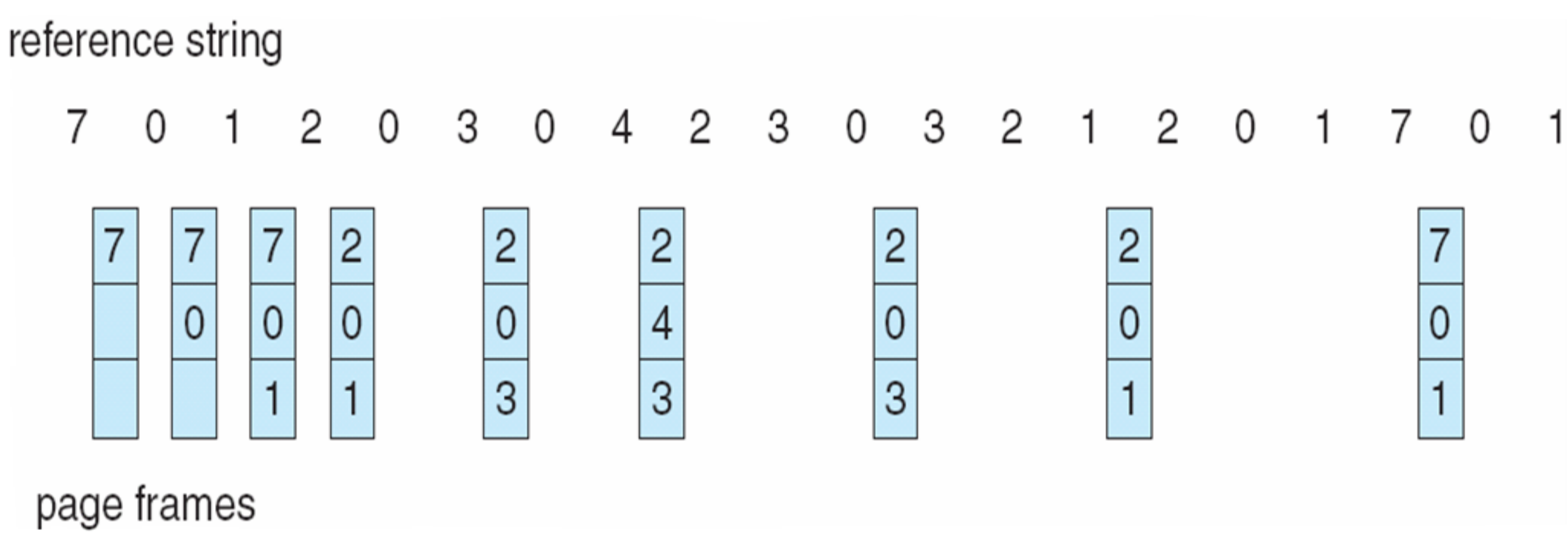
- **FIFO**: replace the first page loaded
  - similar to sliding a window of n in the reference string
  - our reference string will cause 15 page faults with 3 frames
  - how about reference string of 1,2,3,4,1,2,5,1,2,3,4,5 /w 3 or 4 frames?
- For FIFO, adding **more frames** can cause **more page faults**!
  - **Belady's Anomaly**



15 page faults

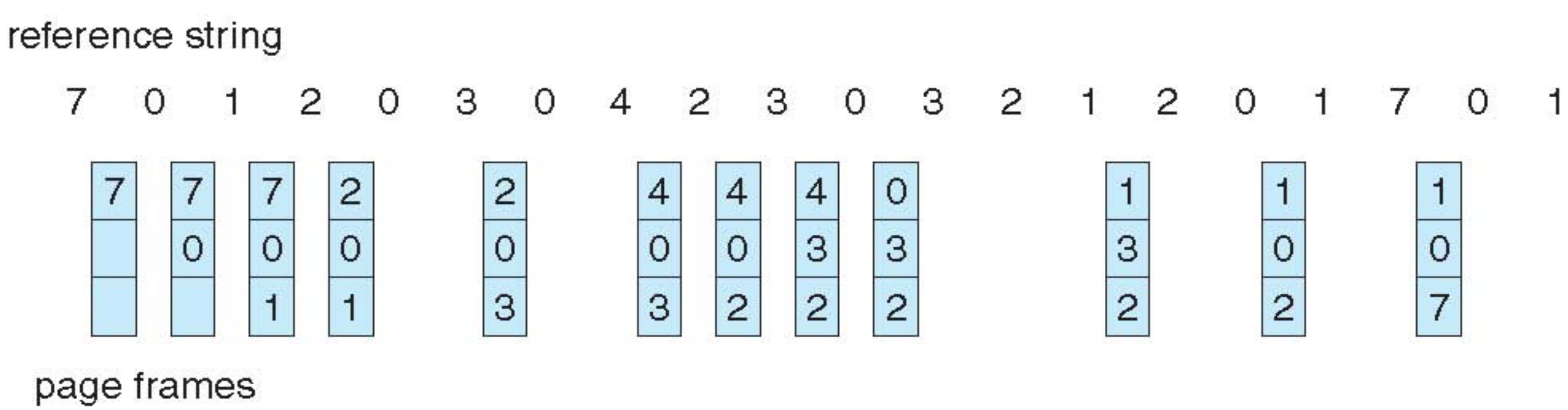
Optimal Algorithm

- **Optimal** : replace page that will not be used for the longest time
  - 9 page fault is optimal for the example on the next slide
- **How do you know which page will not be used for the longest time?**
  - **can’t read the future**
  - used for measuring how well your algorithm performs



Least Recently Used (LRU)

- **LRU** replaces pages that **have not been used for the longest time**
  - associate time of last use with each page, select pages w/ oldest timestamp
  - generally good algorithm and frequently used
  - 12 faults for our example, better than FIFO but worse than OPT
- LRU and OPT do **NOT** have Belady's Anomaly



Thrashing

- If a process doesn't have "enough" pages, page-fault rate may be high
  - page fault to get page, replace some existing frame
  - but quickly need replaced frame back
  - this leads to:

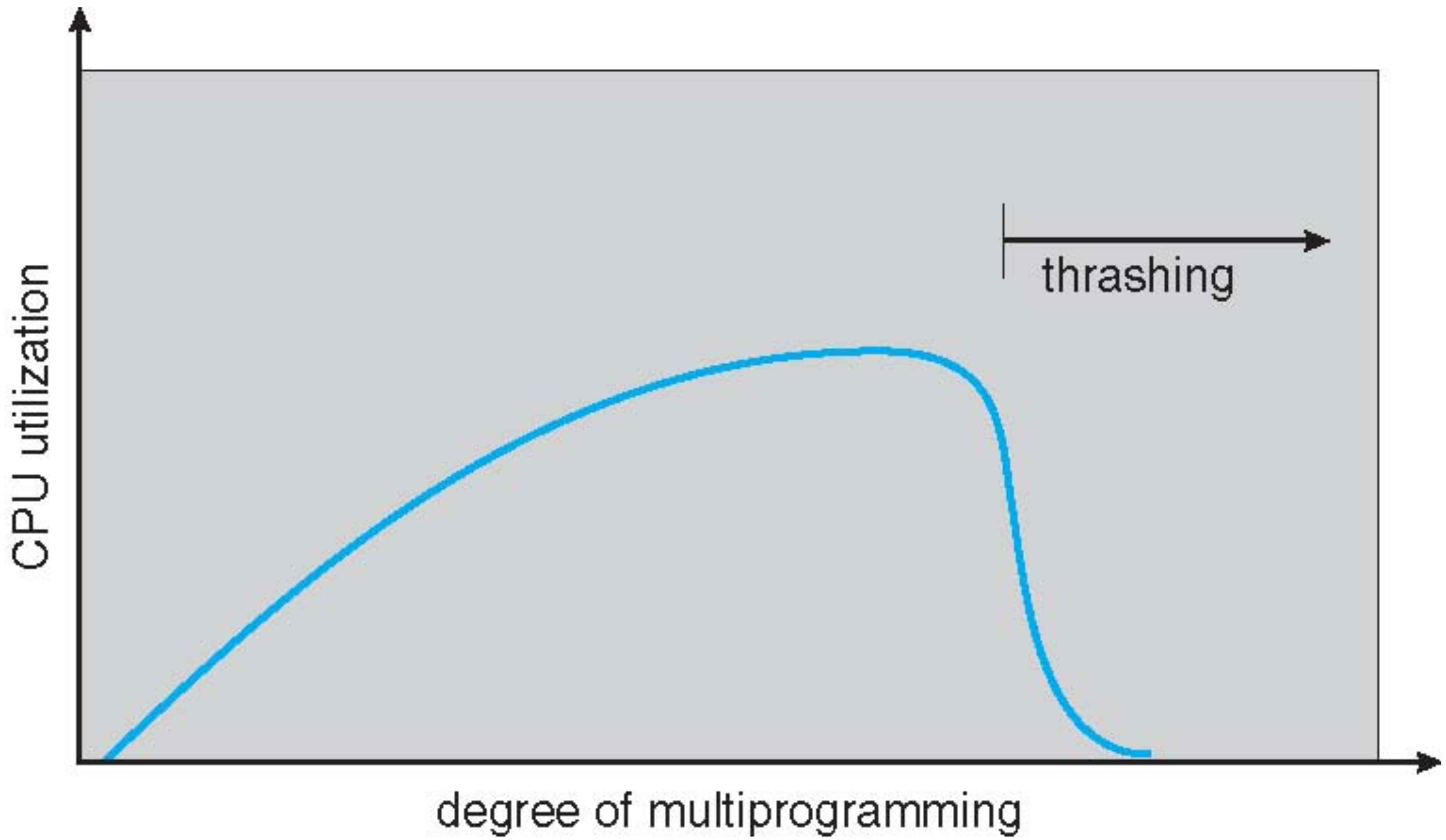
    low CPU utilization ➠

    kernel thinks it needs to increase the degree of

    multiprogramming to maximize CPU utilization ➠

    another process added to the system
- **Thrashing**: a process is busy swapping pages in and out

Thrashing



Demand Paging and Thrashing

- Why does demand paging work?
  - process memory access has **high locality**
    - **What's temporal locality**
  - process migrates from one locality to another, localities may overlap
- Why does thrashing occur?
  - total size of locality > total memory size

Array access is very fast!!