Virtual Memory and Paging

CPSC 457: Principles of Operating Systems
Winter 2024

Contains slides from Pavol Federl, Mea Wang, Andrew Tanenbaum and Herbert Bos, Silberschatz, Galvin and Gagne

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Friday, 25 November 2024

UNIVERSITY OF CALGARY

Overview

- address binding and spaces – logical vs physical
- memory management unit and virtual memory
- Paging, page faults, and general page tables
- Shared pages/copy on write pages/page table size
- Page table implementations - simple, hierarchical, inverted, hashed inverted
- Page fault handling
- Page replacement algorithms - FIFO, Optimal, LRU, Clock
- Thrashing
- Frame allocation algorithms - equal / proportional / priority
- Swapping

Physical Address Challenges

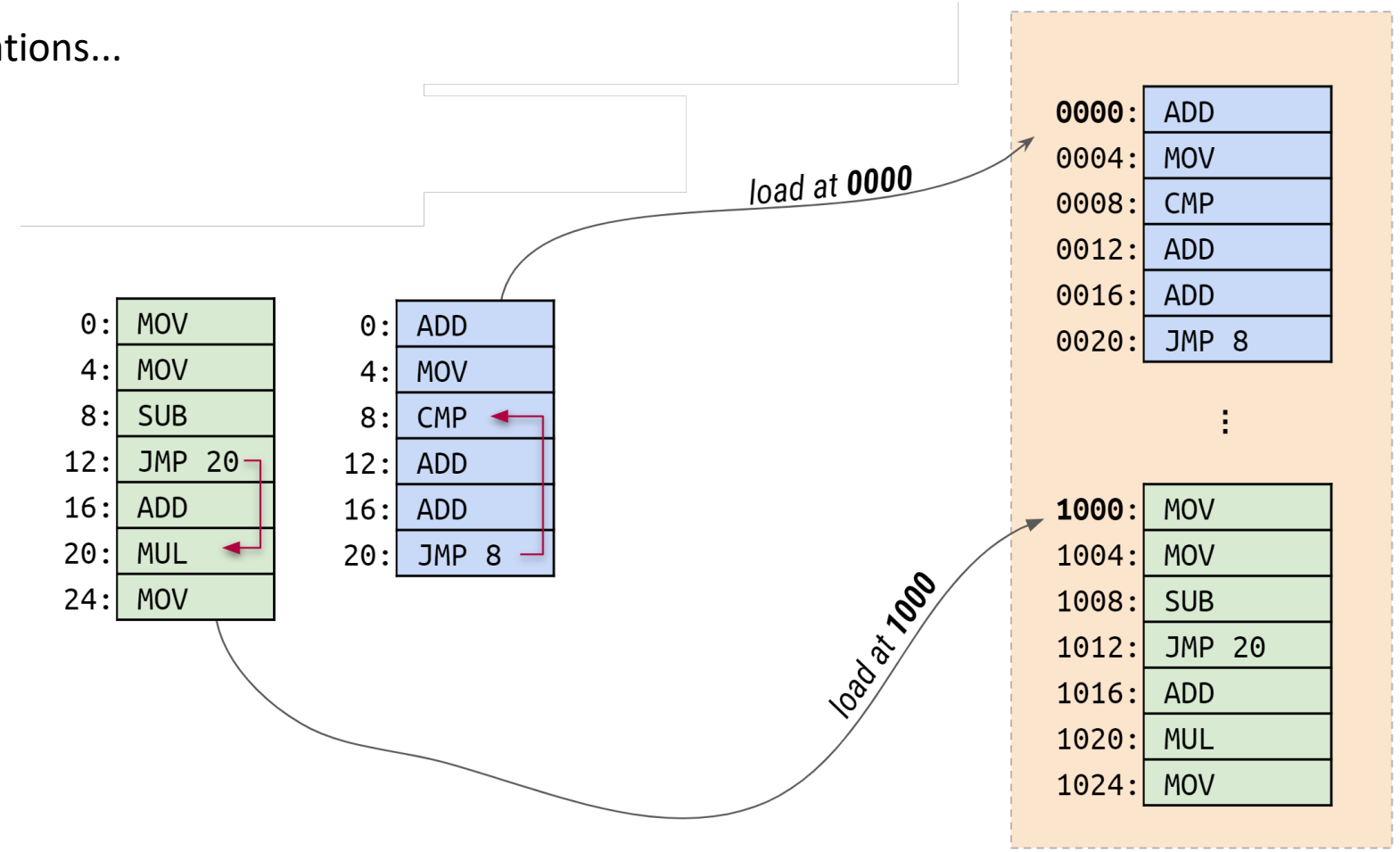
More memory management issues

- OS needs to **protect** memory of a process from other processes
  - how?

- how do we write code if we don't know where the program will be **loaded** in memory?
  - e.g. how do we write `JMP` instruction?

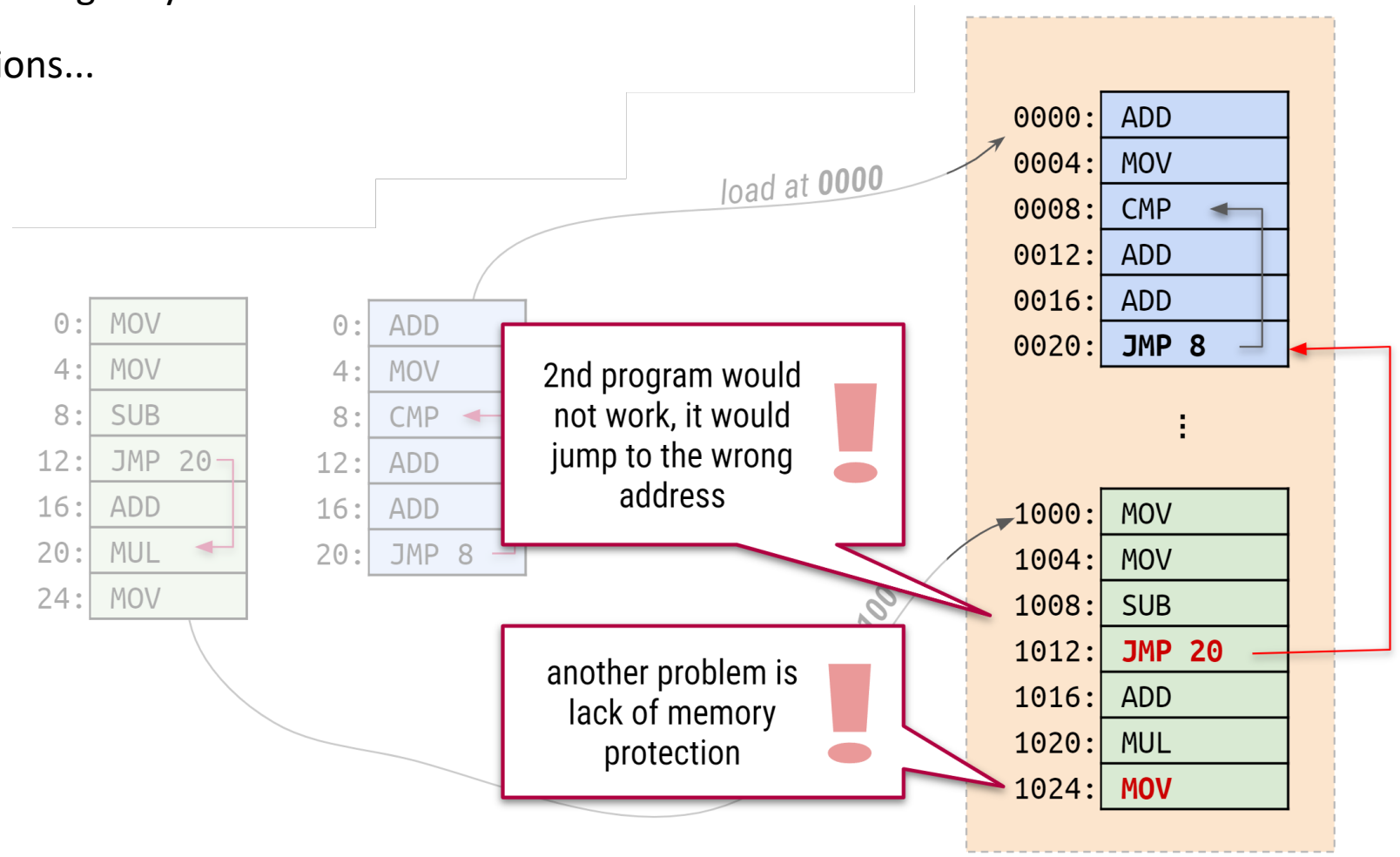
Working with physical addresses

- consider two compiled programs, assuming they will be loaded at address `0000`
- let's place them in two different locations...
- can you spot the problem?



Working with physical addresses

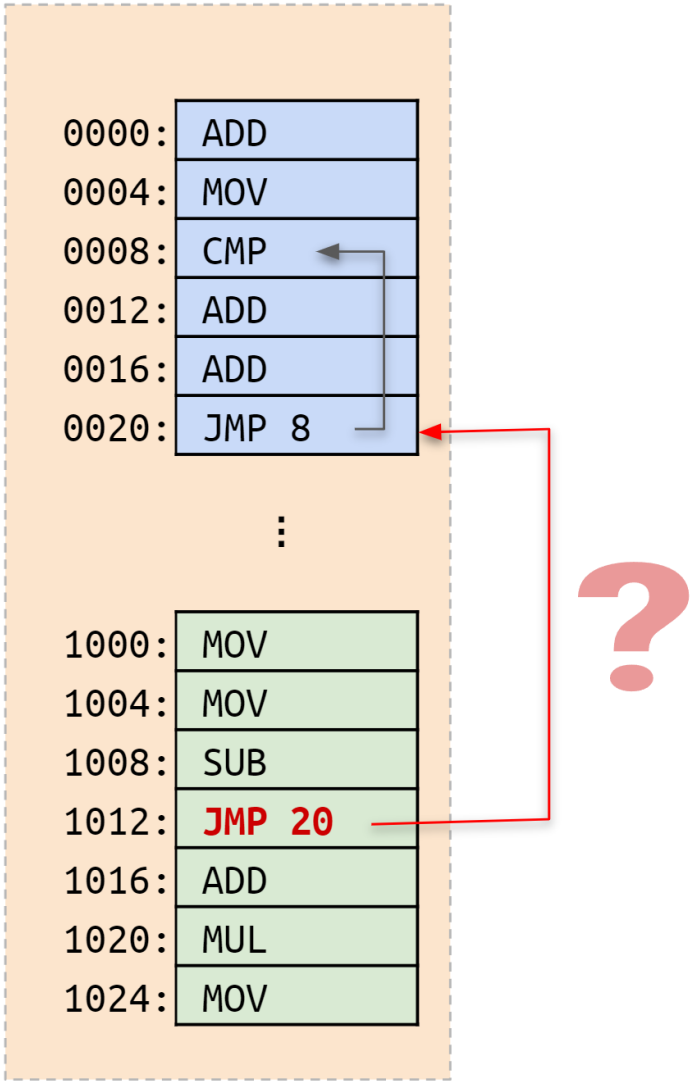
- consider two compiled programs, assuming they will be loaded at address `0000`
- let's place them in two different locations...
- can you spot the problem?



Address Binding

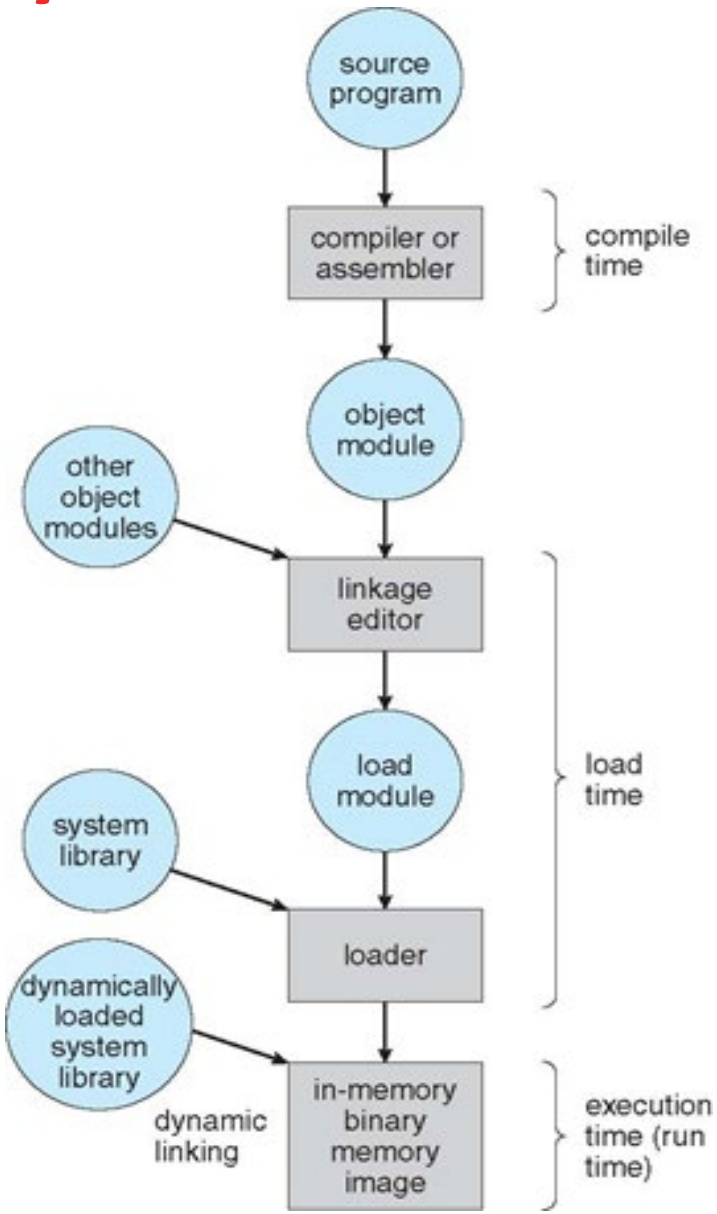
Address binding

- address binding is the process of mapping/converting addresses of a program from one address space to another
- can be used to solve the problem "how to write a program when it's final destination is not known"
- binding can happen at compile time, at load time, or at run time



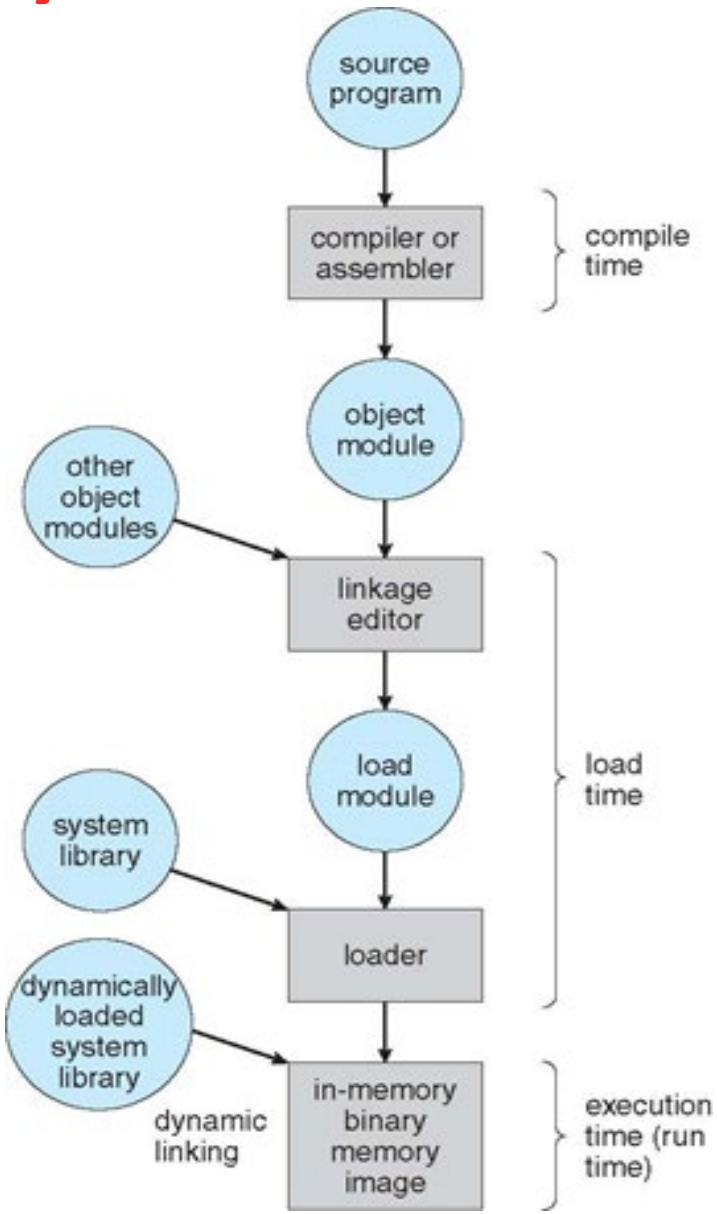
Binding of instructions and data to memory

- at compile time / link time - slowest
  - once the physical location is known, absolute code can be generated and stored by re-compiling the code
  - must recompile every time physical location changes
  - not very useful for multiprocessing systems



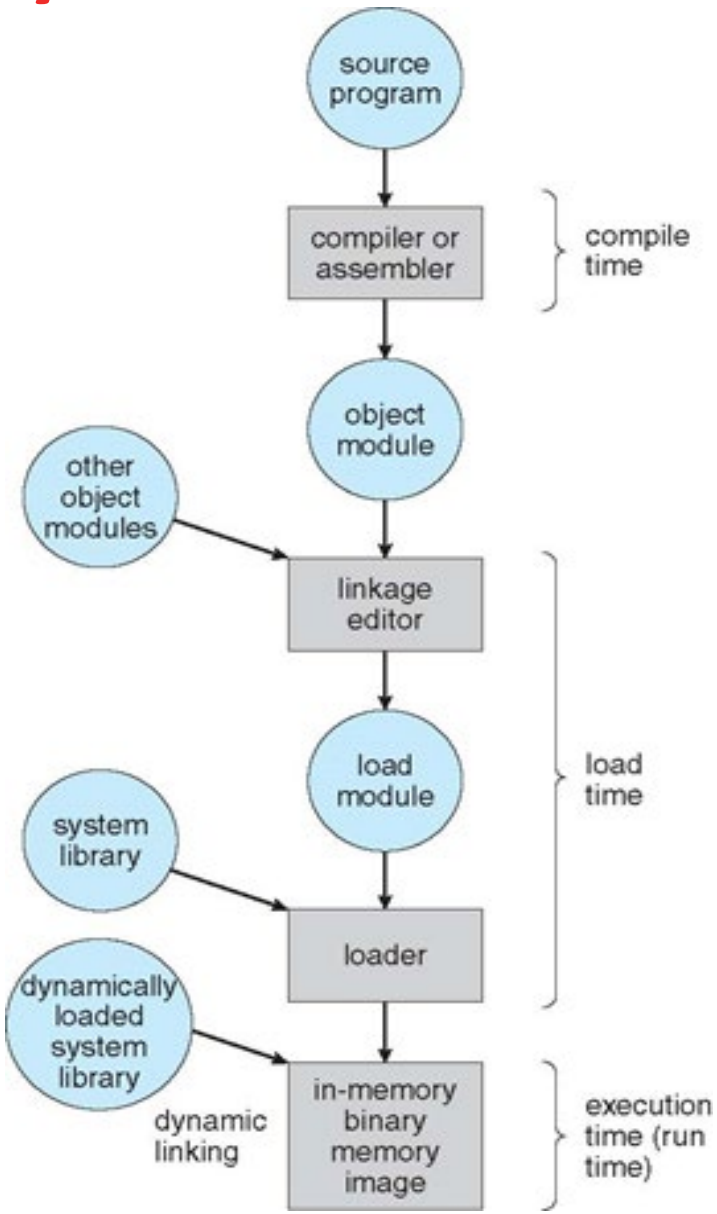
Binding of instructions and data to memory

- at load time - much faster
  - compiler/linker outputs relocatable code
  - binding is done by loader before program starts executing
  - we include position independent code (PIC) in this category, but it could be HW assisted via special register



Binding of instructions and data to memory

- at run time - fastest (with HW support)
  - if process can be moved during its execution, binding is done at run-time, dynamically
  - most flexible, but need hardware support (e.g., memory management unit (MMU))



Address Spaces

Logical addresses

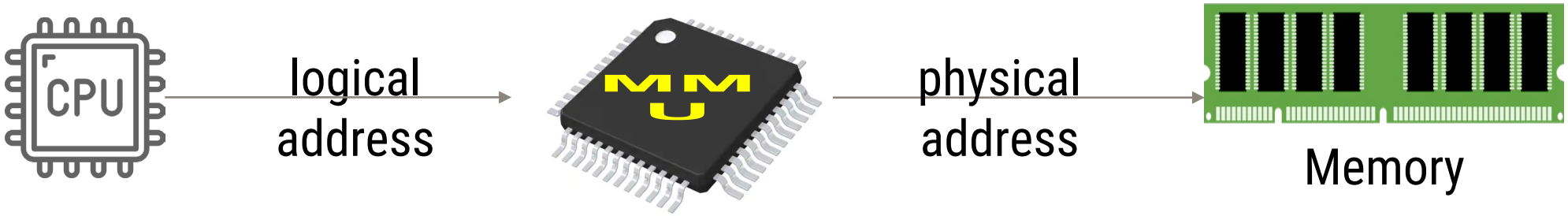
- we can achieve execution-time address-binding and memory-protection by 'virtualizing memory'
- OS gives each process an illusion of logical address space (aka virtual address space)
  - logical address space can be a contiguous space [`0` … `max`]
  - as process executes on CPU, the addresses generated by the CPU are logical addresses
  - if logical address does not fall into the logical address space range → violation (exception)

Physical addresses

- logical addresses are mapped to physical addresses before reaching memory via hardware device called memory management unit (MMU)
- physical address - a real memory address
  - physical address space of a process is the subset of RAM allocated to a process
  - depending on the hardware support, physical address space does not need to be contiguous

MMU

Memory-Management Unit (MMU)



- MMU is a hardware device that maps virtual/logical addresses to physical addresses
- integrated into most/all modern CPUs
- the process running on CPU does not know what the physical addresses are, only OS knows
- execution-time binding occurs automatically whenever memory reference is made
- MMU can also help with memory protection

most of the time, the MMU attaches directly to the CPU address bus and intercepts each CPU read or write cycle. The CPU and MMU combine to form a new functional unit. Several manufacturers have even moved the MMU onto the same silicon as the CPU, in effect declaring that you can't have one without the other.

The most important function provided by all MMU designs is the ability to relocate a program to another part of memory according to a set of pre-assigned translation rules. This relocation is done in hardware, without requiring any modification to the application software.

Before a system with an MMU runs a program, the operating system configures the MMU so that the program can be moved to and run in an available section of memory. The program then begins execution, unaware of the MMU's actions. For example, if a pro

BY GREGG ZEHR

MEMORY MANAGEMENT UNITS FOR 68000 ARCHITECTURES

Design options that speed up memory management

The Motorola 68000 family of microprocessors has spawned a whole new group of computer systems. The original 68000, with its large, linear addressing range, makes it a natural for single-user, personal graphics workstations such as the Macintosh. And multiuser systems based on the 68020 can offer computing power and speed that rival many minicomputers—often at a fraction of the cost. Not surprisingly, many of the design features for these larger systems have evolved from well-established minicomputer architectures. Memory management units, or MMUs, are one example. The MMU function came about as minicomputer designers began to include special hardware to expand the amount of addressable memory. MMUs have now become a key feature in modern computer architectures. In fact, several MMUs designed specifically for the 68000-family architecture are available (see table 1).

THEORY OF OPERATION

The MMU functions at a very low level in the computer system. Unlike a UART or other peripheral chip that attaches to the system bus and is idle most of the time, the MMU attaches directly to the CPU address bus and intercepts each CPU read or write cycle. The CPU and MMU combine to form a new functional unit. Several manufacturers have even moved the MMU onto the same silicon as the CPU, in effect declaring that you can't have one without the other.

The most important function provided by all MMU designs is the ability to relocate a program to another part of memory according to a set of pre-assigned translation rules. This relocation is done in hardware, without requiring any modification to the application software.

Before a system with an MMU runs a program, the operating system configures the MMU so that the program can be moved to and run in an available section of memory. The program then begins execution, unaware of the MMU's actions. For example, if a program has been compiled and linked with a starting location of 400 but that location is being used for some other purpose, the operating system configures the MMU hardware to convert all the program's memory references to an unused section of memory. Although the MMU is obviously useful in a system that has multiple users running separate programs, it is just as useful in a multitasking single-user system.

In a simple 68000 system that does not have an MMU (figure 1), a typical memory read cycle begins when the CPU asserts an address and address strobe (AS), and the cycle ends when the memory places data on the data bus and activates the data transfer acknowledge (DTACK) line. Assuming that the memory is very fast, the cycle can be completed in eight transitions of the clock, or 500 nanoseconds for an 8-MHz CPU.

In a 68000 system that has an MMU in series with the CPU's address bus (figure 2), for each read cycle the CPU asserts a logical address and logical address strobe (LAS). (The address and address strobe lines are now

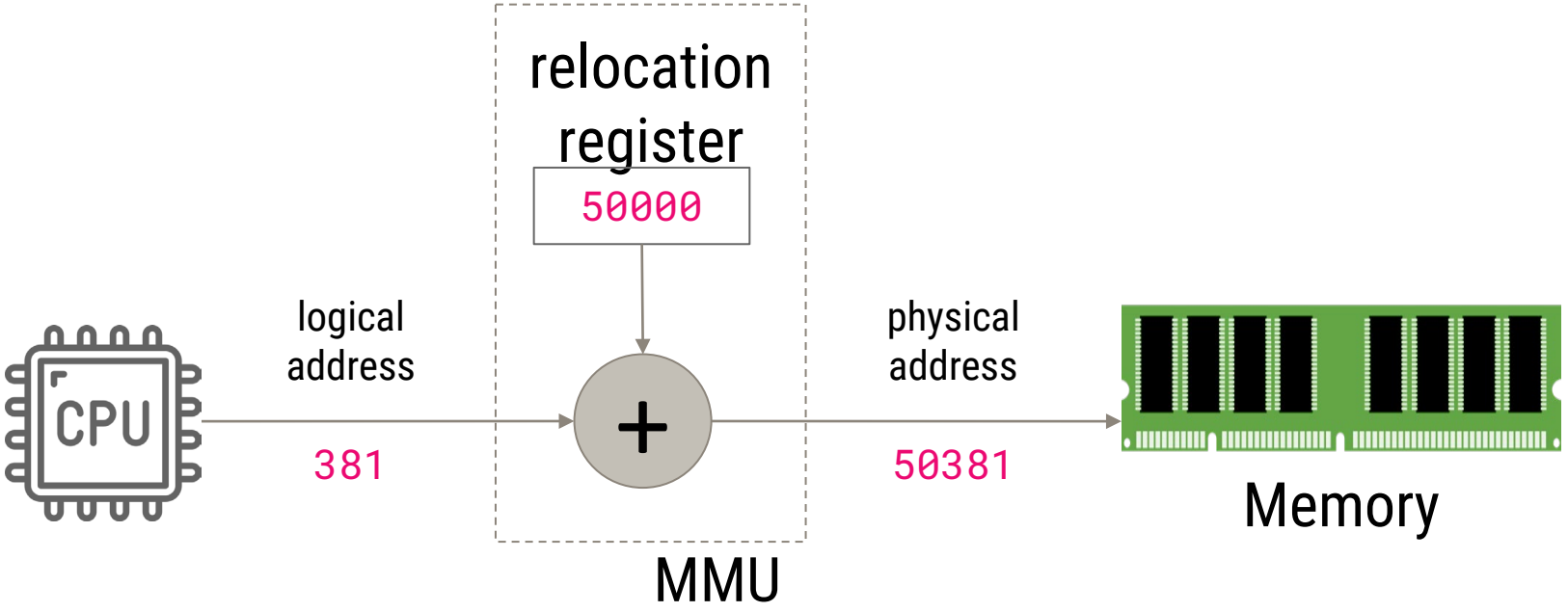
(continued)

Gregg Zehr is a senior design engineer at Altos Computer Systems (2641 Orchard Parkway, San Jose, CA 95121). He received his M.S.E.E. from the University of Illinois and is interested in advanced computer architectures.

http://marc.retronik.fr/motorola/68K/68000/Memory_Management_Units_for_68000_Architectures_[BYTE_1986_9p].pdf

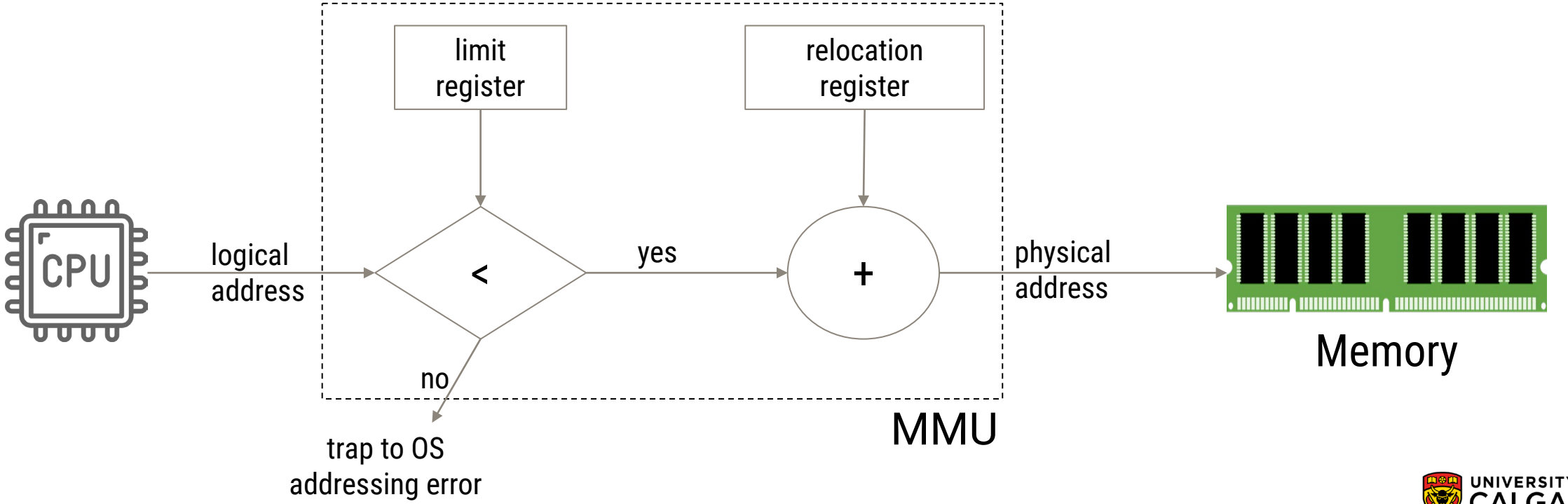
Basic MMU with a Relocation Register

- a simple MMU implementation – using a relocation register
- logical address space starts at 0,
  i.e. programs are written/compiled assuming they start at address 0
- value in the relocation register is added to every address generated by a CPU

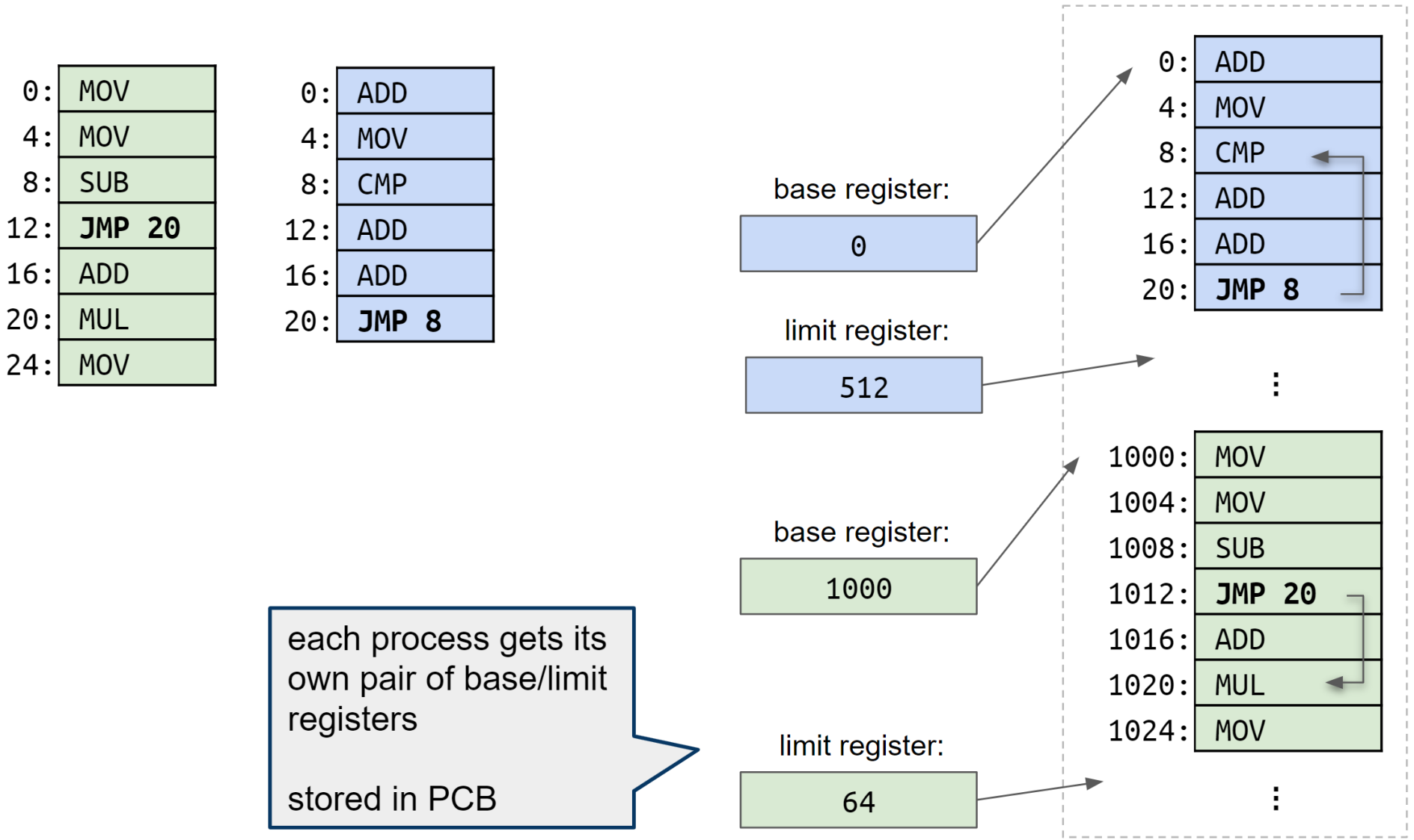


Basic MMU — with Relocation and Limit Registers

- adding limit register to implement address protection
  - relocation (base) register = start of the physical memory address given to process
  - limit register = the size of the chunk of physical memory a process is allowed to use
- achieves execution-time binding as well as memory protection

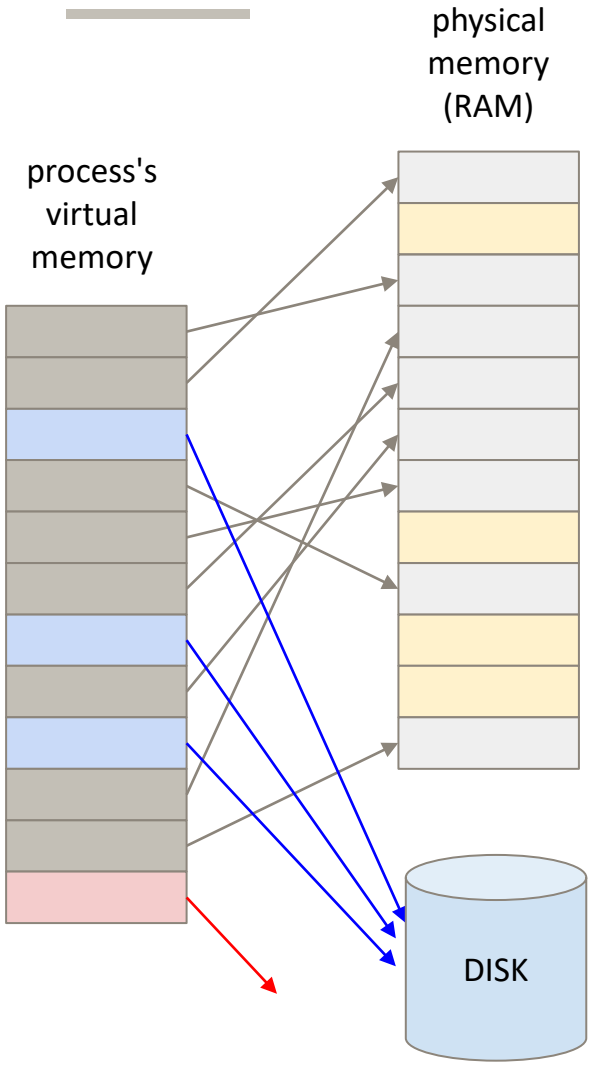


Relocation and Limit Registers Example



Virtual Memory

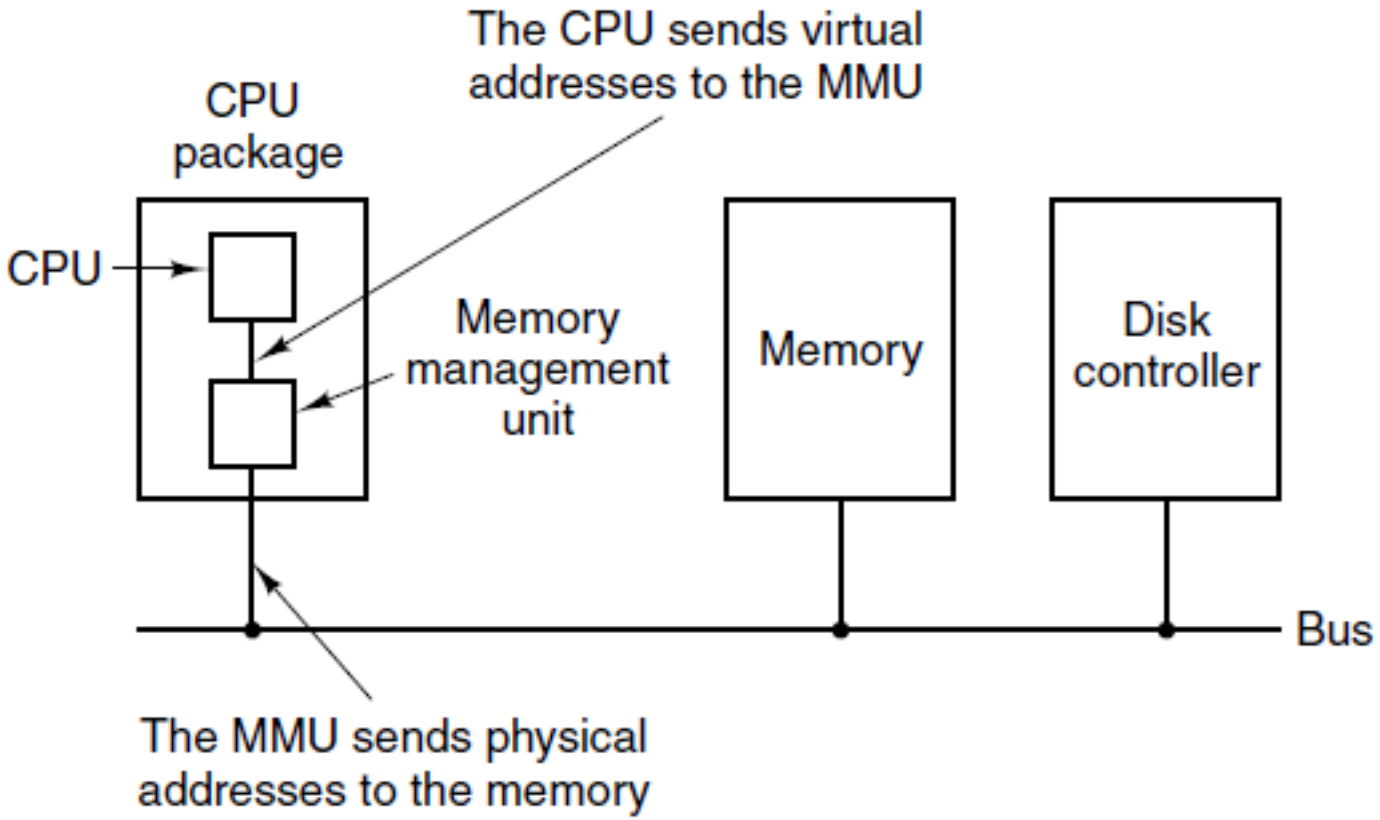
Virtual memory



- virtual memory is a memory management technique that allows the OS to present a process with **contiguous** logical address space, while allowing for **non-contiguous** physical address space
- some parts of logical address space can be even mapped to a backing store, allowing OS to **overallocate** memory
- some logical addresses could even map to **nowhere**
- plus many additional nice features
- implemented as a combination of SW & HW
- present on nearly all modern systems

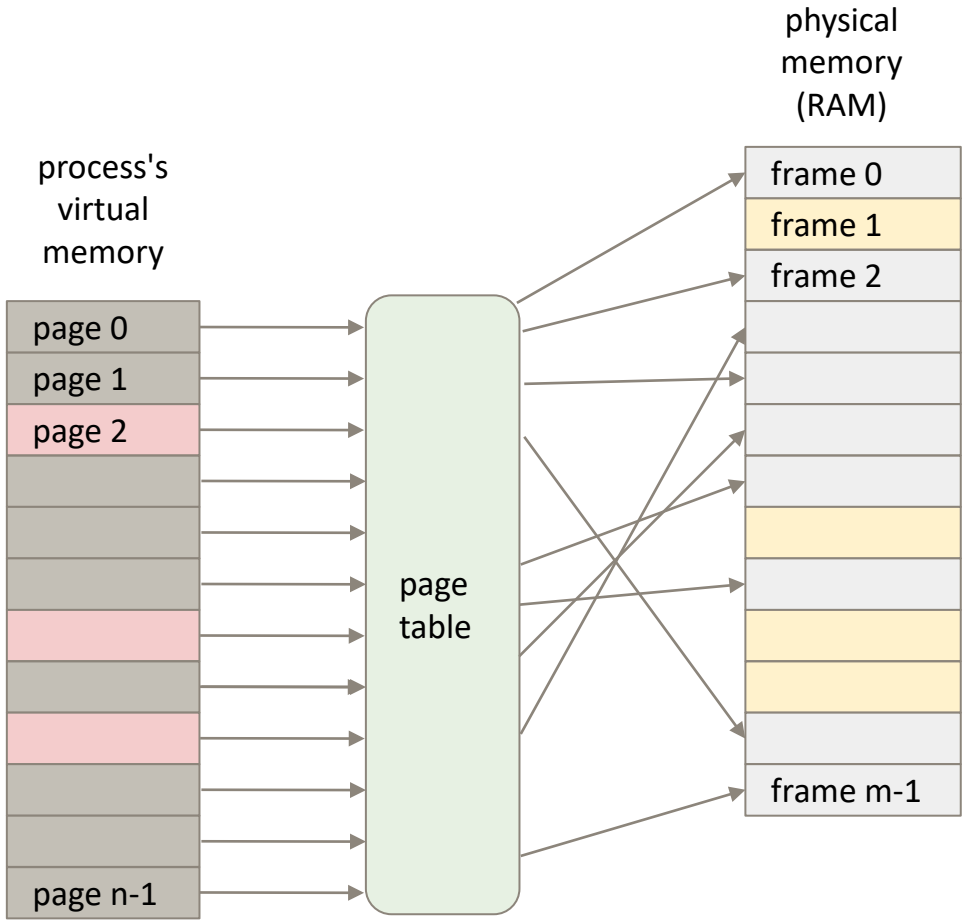
Paging

Paging with MMU



Paged virtual memory

- virtual address space is divided into **pages**:
  - blocks of fixed size, e.g. 4 KiB
  - almost* always power of 2
- physical memory is divided into **frames**:
  - fixed sized blocks, same size as pages
- pages map to frames via a lookup table called **page table** (logical → physical address mapping)
- eliminates external fragmentation
- per-process page table or one system-wide page table



Paging example

- virtual address space = 64KB
- physical address space = 32KB
- page size = 4KB
- calculate:
  - frame size = ?
  - # of pages = ?
  - # of frames = ?

In this course I occasionally use the old notation:

```
1 KB = 1024 B
1 MB = 1024 KB
1 GB = 1024 MB
```

unless explicitly stated otherwise

Paging example

- virtual address space = 64KB
- physical address space = 32KB
- page size = 4KB
- calculate:
  - frame size = **4KB** (same as page size)
  - # of pages = **16** (64KB / 4KB)
  - # of frames = **8** (32KB / 4KB)

Paging example

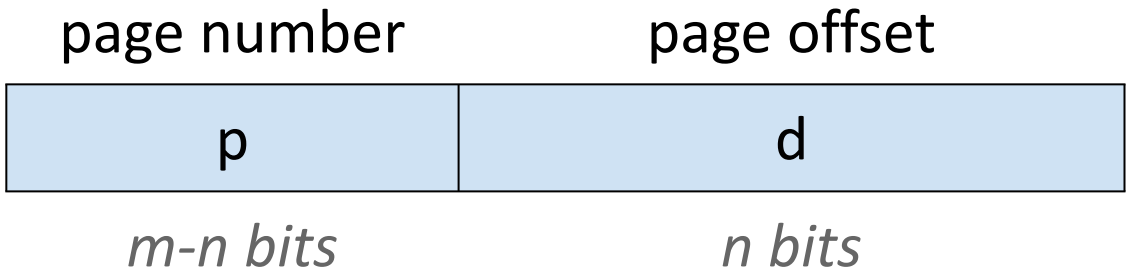
- Assume page size is 2KB, and a process needs 71 KB to load. How many pages do we need?
  - we need 35 + ½ pages
  - OS needs to find 36 free frames

- Do the frames need to be contiguous?
  - No, OS can allocate any 36 frames (discontiguous is fine)
  - OS adjusts the page table to reflect the frame locations
  - logical address space remains contiguous

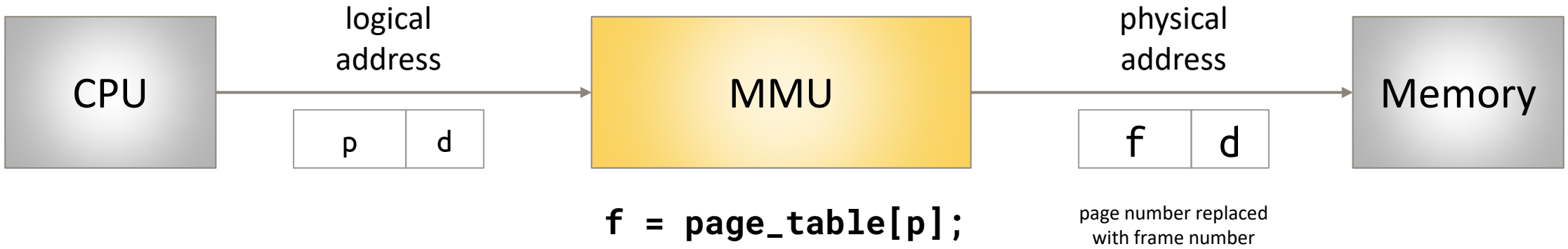
- Notes:
  - one frame will have 1KB of unused space (internal fragmentation)
  - no external fragmentation since all frames are usable
  - but what if there are no free frames? we can map some pages to disk…

Address translation (logical → physical)

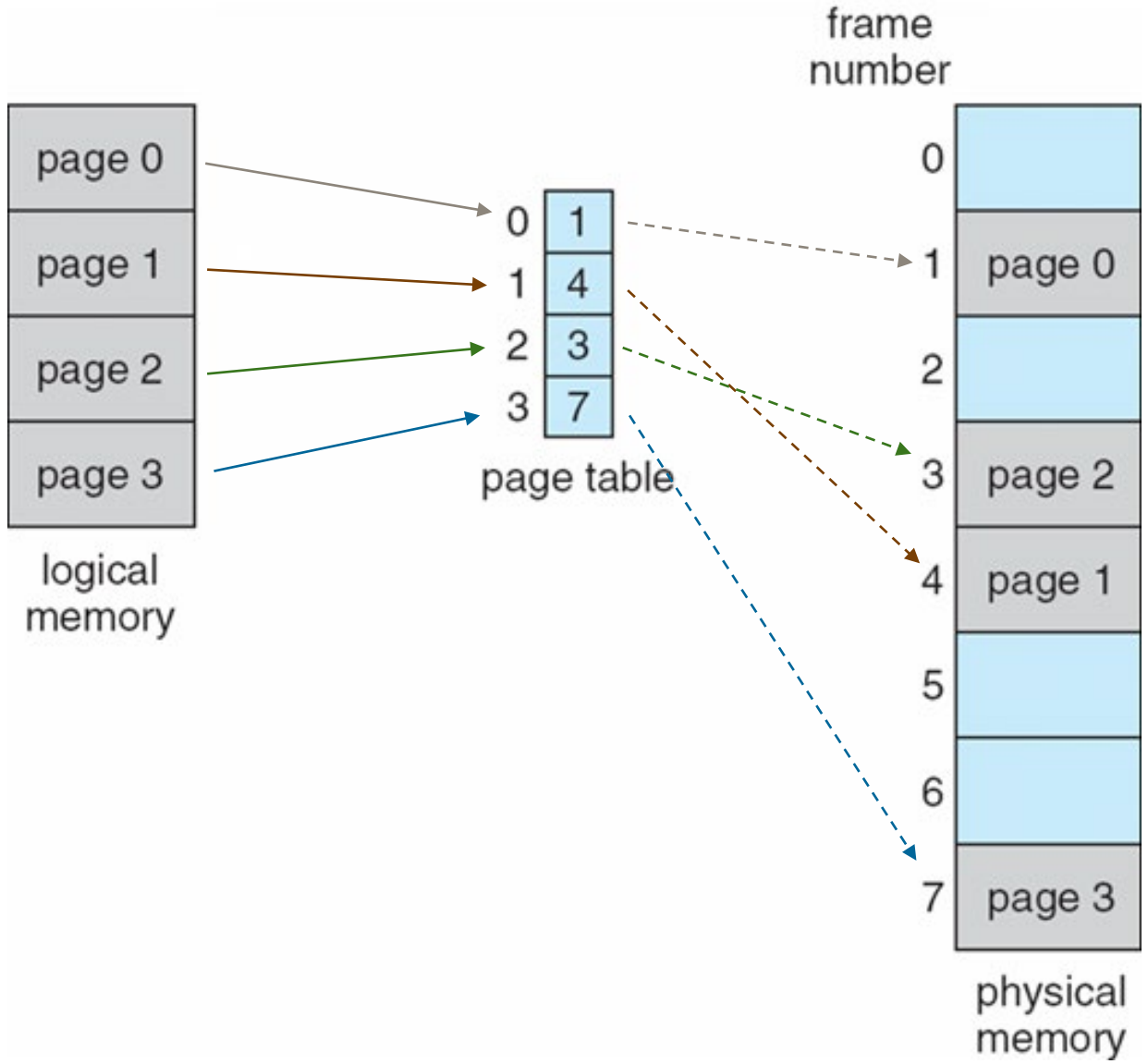
- address generated by CPU is split into:
  - **page number (p)** – used as an index into a page table which contains base address of corresponding frame in physical memory
  - **page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit
- if page sizes are powers of 2, calculating page number and offset is very simple:
  - **m**-bit logical address space (2^m possible addresses)
  - **n**-bit page size (2^n bytes in one page)
  - last **n** bits of the logical address is the offset, the remaining **m-n** bits is the page number



Paged virtual memory - MMU

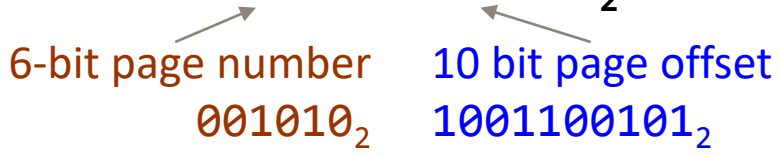


Paging Model of Logical and Physical Memory



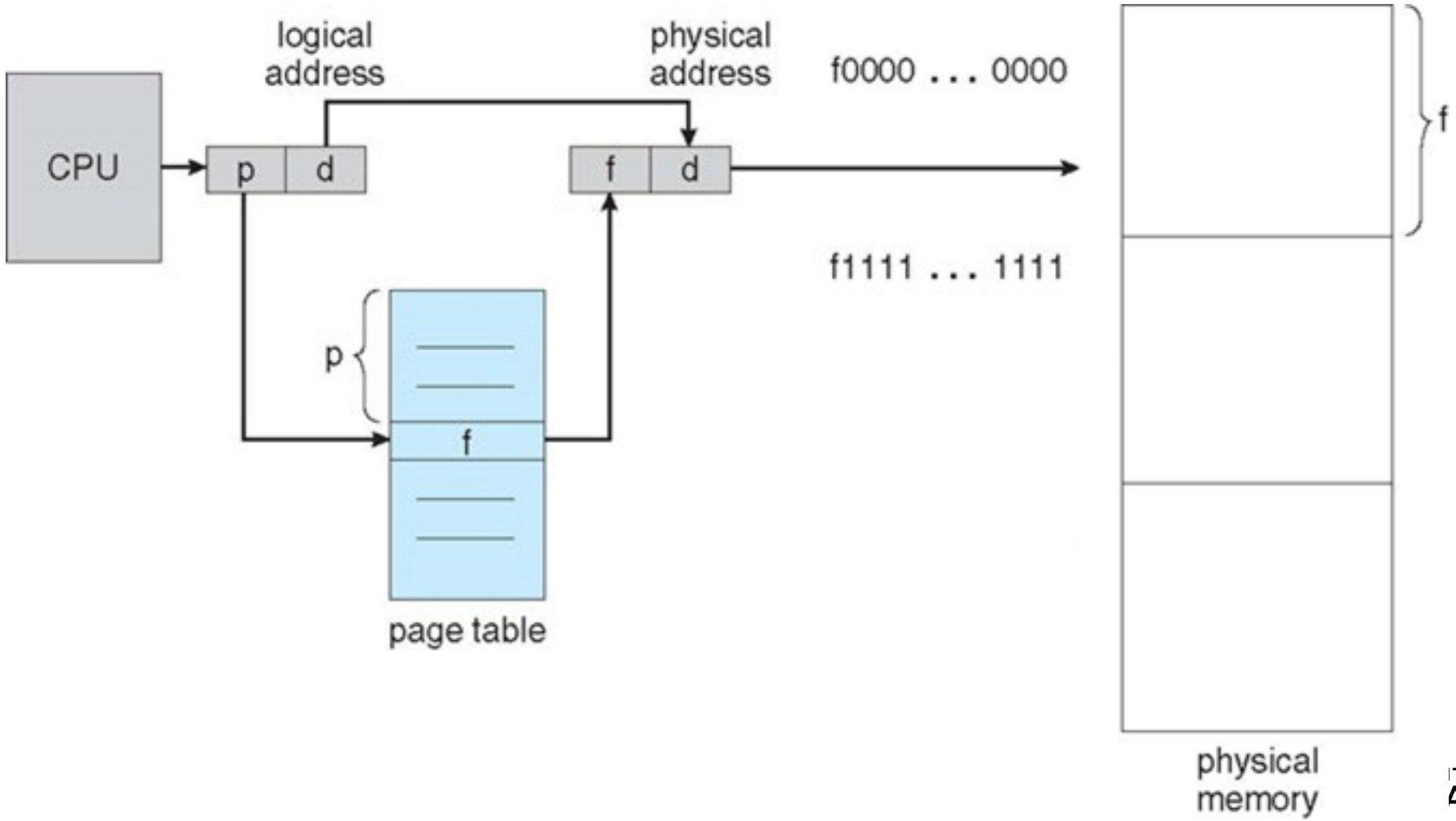
Address translation (logical → physical)

- Question:
  - assume 16-bit logical address space and page size of 1024 bytes
  - what is the page number and offset for a logical address **10853** ?
- Answer:
  - 1024 byte page size means we need 10 bit page offset
  - 10853 written as a binary number is $\mathbf{0010101001100101_2}$



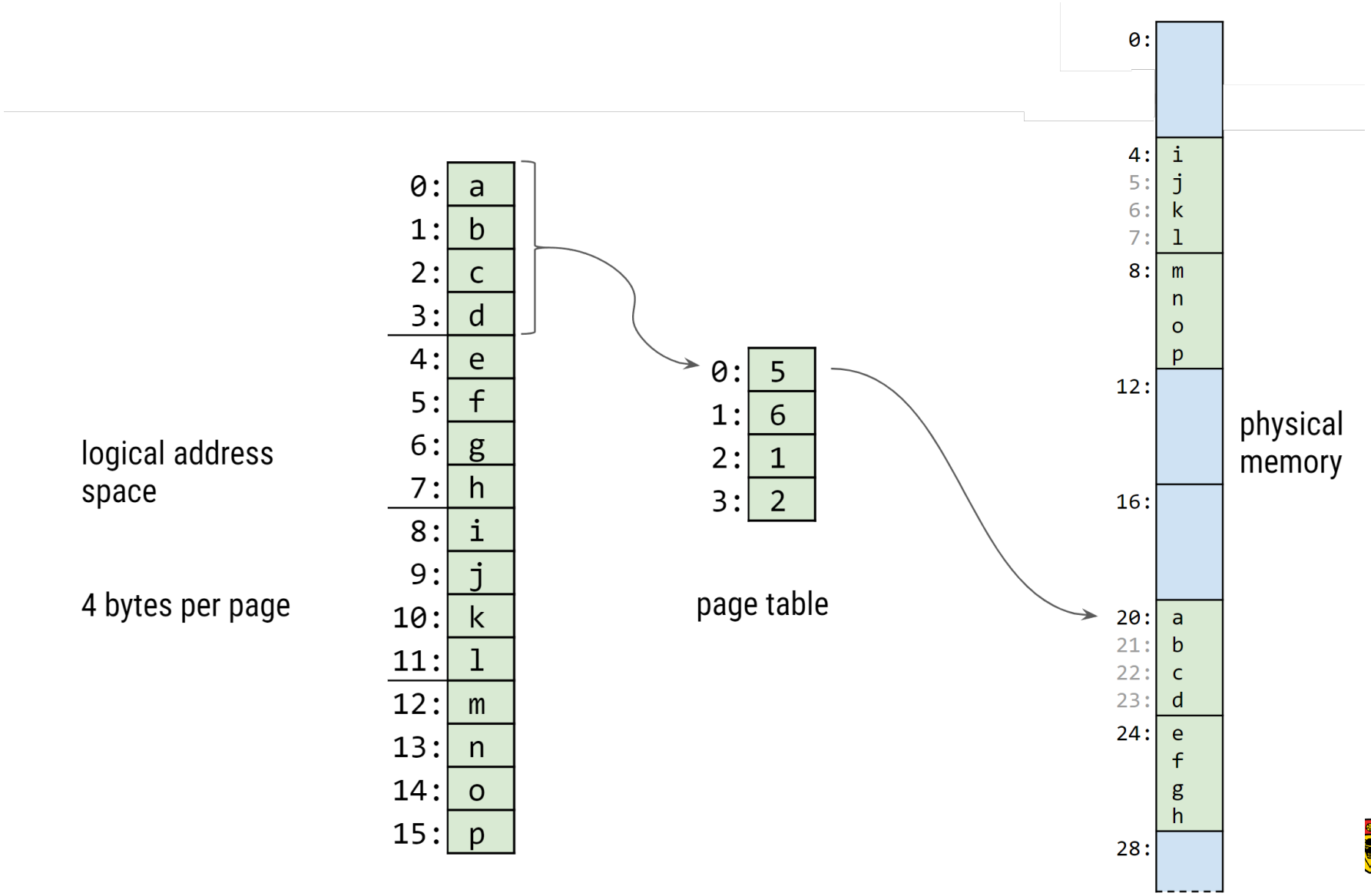
  therefore page number = $\mathbf{001010_2}$ = $\mathbf{10_{10}}$, and page offset = $\mathbf{1001100101_2}$ = $\mathbf{613_{10}}$
  - another way to solve it is using regular division:
    - 10853 / 1024 = 10 remainder 613

Paging hardware

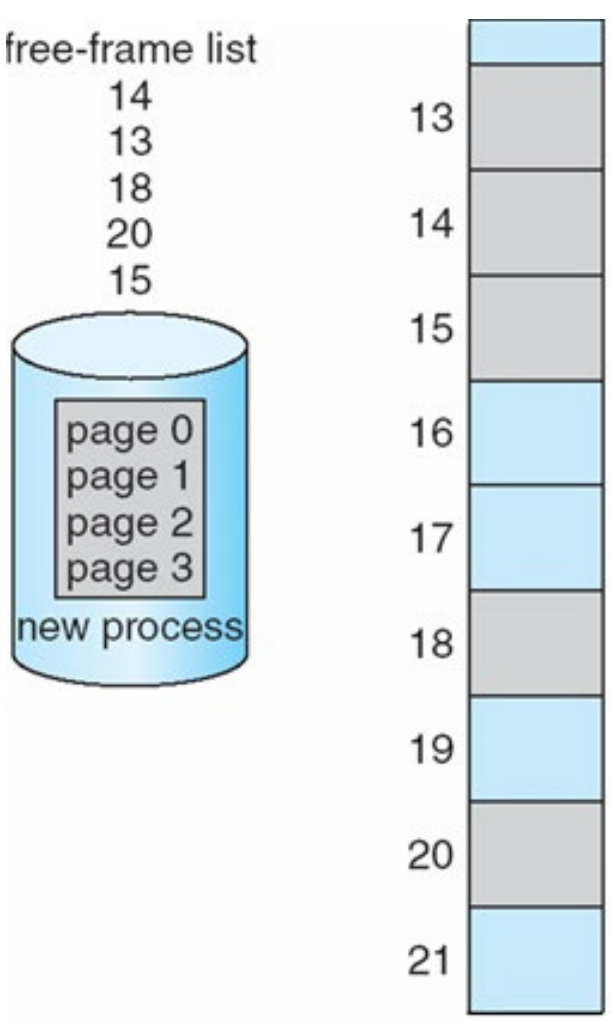


p = page number
d = page offset
f = frame number

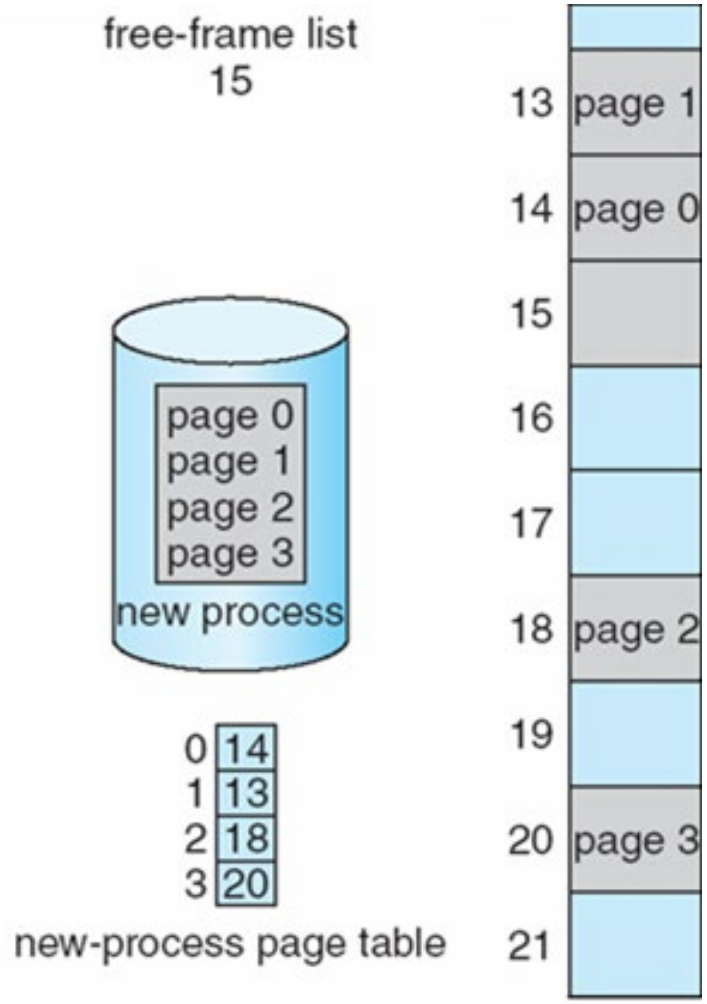
Paging Model of Logical and Physical Memory



Free frames



before allocation



after allocation

Page Faults

Page fault

- What happens when a program tries to access a page that does not map to physical memory?
  - CPU issues a trap — called page fault
  - OS suspends the process
  - OS locates the missing page on disk
    - what happens if not on disk? → invalid page fault
      results in a crash, segmentation fault, core dump ...
  - OS loads the missing page from disk into a free frame
    - if no free frames available – OS will evict one by saving it to backing store
  - □ OS updates the page table
  - OS resumes the process by restarting the offending instruction
- if OS only loads pages as a result of page fault, we call that demand paging

Paging performance

- paging performance is commonly evaluated via **effective access time (EAT)** for memory access
- let **p** = probability of page fault or **page fault rate** (0 ≤ p ≤ 1)

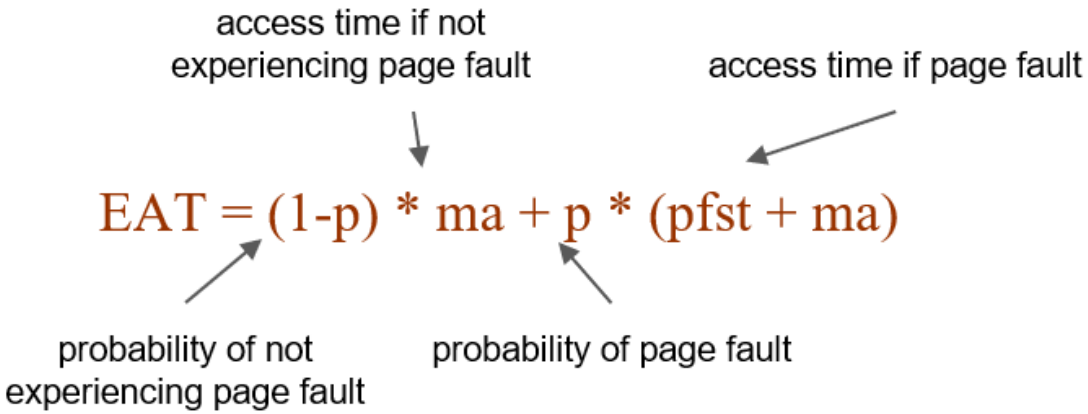
  p = 0 → all pages are in memory, no page fault

  p = 1 → all pages are on disk, all memory accesses are page faults

  ma = memory access time (includes page translation time)

  pfst = page fault service time, ie. how long does it take to service a page fault

- then



Expected value

- let's say we want to find out an average outcome of repeating the same experiment many times
- if the experiment has n possible outcomes: $x_1, x_2, \ldots, x_n$
- and each outcome has the probability of occurring: $p_1, p_2, \ldots, p_n$ respectively, where $p_1 + p_2 + \ldots + p_n = 1$
- we can calculate the expected value of the experiment as a weighted average of the outcomes, using the probabilities as weights:

$$\text{expected value} = \sum_{i=1}^{n} x_i\, p_i = x_1 p_1 + x_2 p_2 + \cdots + x_n p_n$$

Expected value example

- if we rolled a six-sided die "many" times and recorded the outcomes…
- what would be the average of these outcomes?
- six possible outcomes: 1, 2, 3, 4, 5, 6
- probability of each outcome: ⅙

- expected value = $1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} = 3.5.$

Paging performance examples

- EAT = (1-p) * ma + p * (pfst + ma)
- non-realistic example:
  - calculate EAT if page fault probability is 50%, ma = 1ms and pfst = 10ms
  - EAT = (1-0.5) * 1ms + 0.5 * (10ms + 1ms) = 6ms
- more realistic example:
  - calculate EAT if page fault probability is 1/1000, ma = **100ns** and pfst = **10ms**
  - EAT = (1-0.001) * 100ns + 0.001 * 10,000,100ns = 10,100ns = 10.1µs

Paging performance examples

- EAT = (1-p) * ma + p * (pfst + ma)
- non-realistic example:
  - calculate EAT if page fault probability is 50%, ma = 1ms and pfst = 10ms
  - EAT = (1-0.5) * 1ms + 0.5 * (10ms + 1ms) = 6ms
- more realistic example:
  - calculate EAT if page fault probability is 1/1000, ma = **100ns** and pfst = **10ms**
  - EAT = (1-0.001) * 100ns + 0.001 * 10,000,100ns = 10,100ns = 10.1µs

~100 x slower

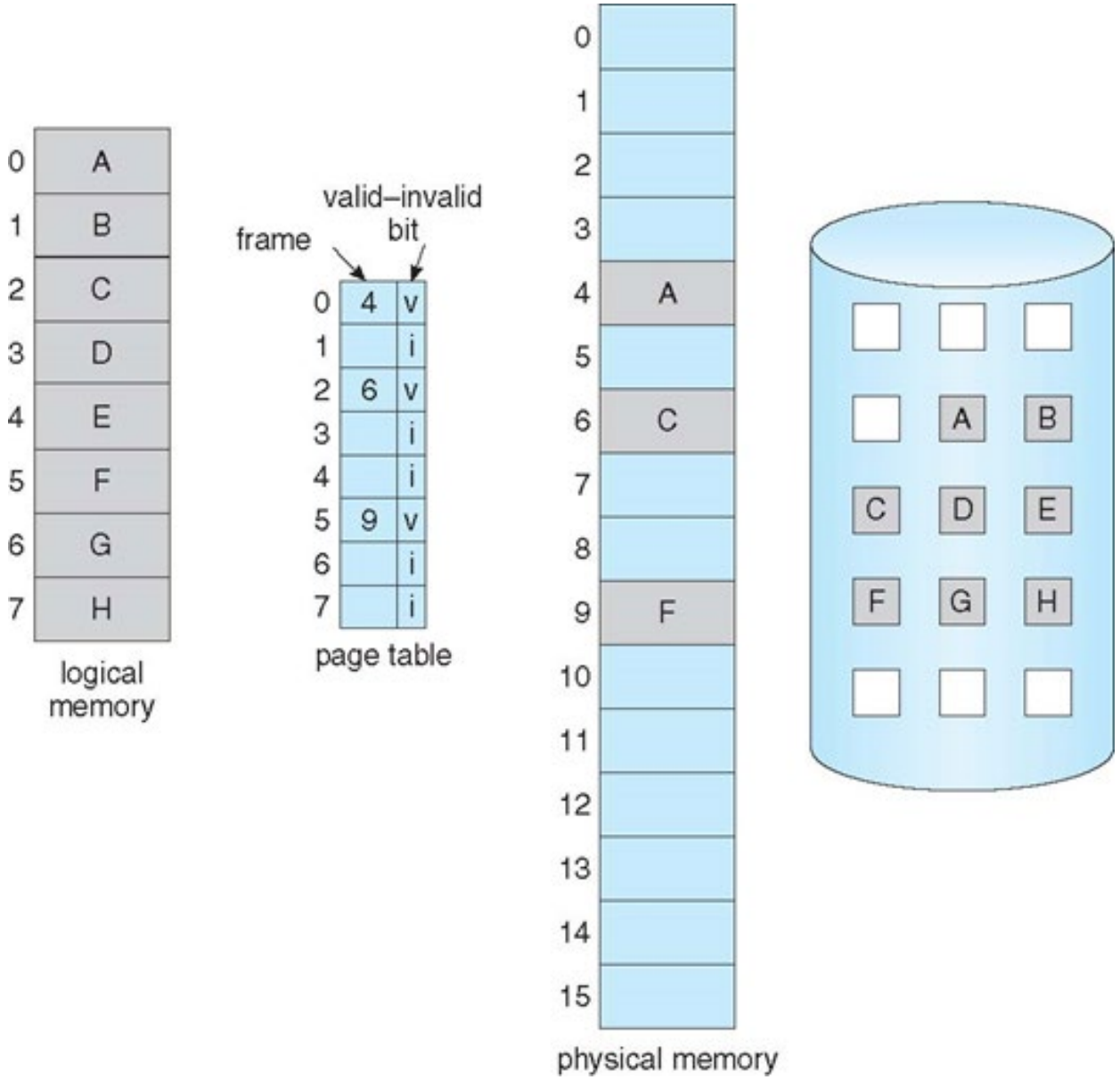
Page Fault Handling

Page fault handling

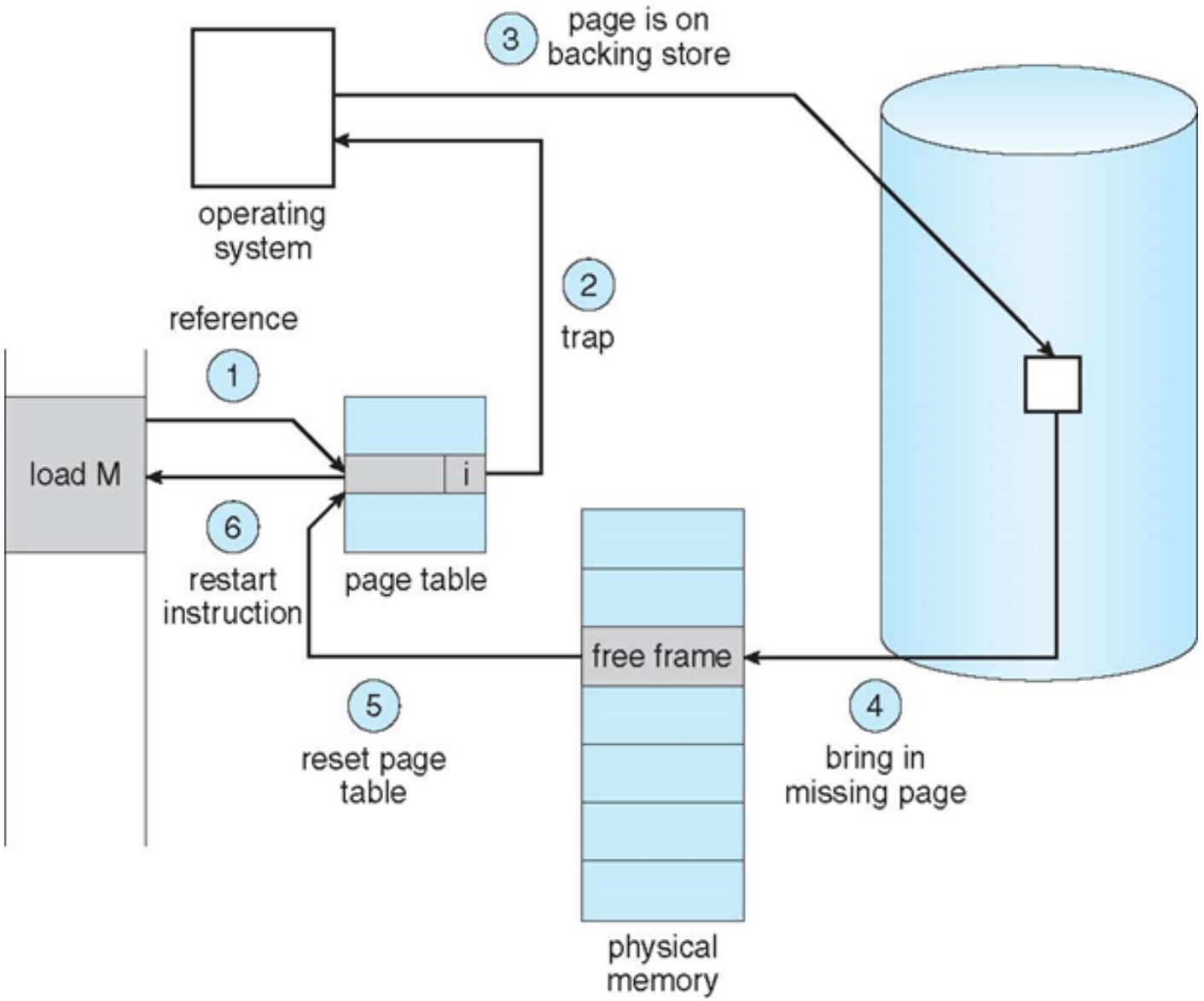
- **page fault** - exception raised when a process accesses a page not currently mapped by MMU
  - e.g. entry in page table marked invalid
  - note: with demand paging, first reference to a page always results in a page fault

- general page fault handling:
  1. operating system looks at *another* table to decide:
     - invalid reference → abort
     - reference valid, but page not in memory, eg. it's in a backing store
  1. find a free frame
  2. load page from backing store into frame by scheduling appropriate disk operation
  3. when done, reset page tables to indicate page now in memory
  4. restart the instruction that caused the page fault

Page table with some pages not in main memory



Page fault handling

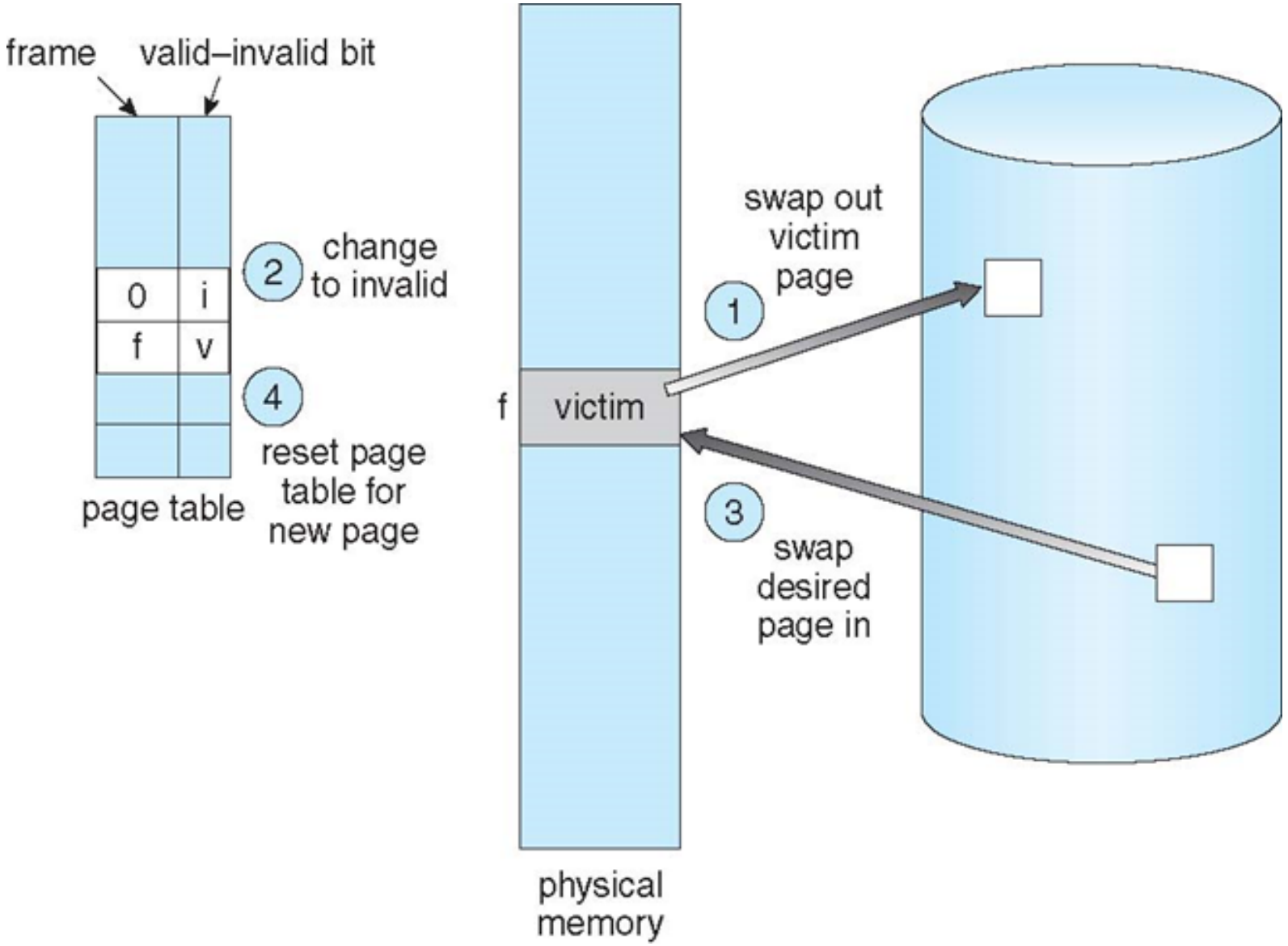


What happens if there are no free frames?

How does OS deal with **over-allocation** of memory?

- OS needs to make room by evicting an existing frame
- OS finds an occupied (victim) frame in memory and pages it out
  - saves it to backing store, and remembers it so that it can find it later
    i.e. update page table & other relevant data structures
  - can use the **modify (dirty) bit** in a page table entry to reduce overhead of page transfers, so that only modified pages are saved to the backing store
  - remember: dirty bit is automatically set by hardware on write access
- but which frame do we evict?
  - we need an algorithm to find a victim page
  - this algorithm must be fast, OS cannot afford to be too fancy
  - also, the algorithm should minimize the overall number of page faults
- we will need a **page replacement** algorithm

Basic page replacement



Global vs. Local Replacement

- when no free frames are available, OS needs to replace a frame...

- **global replacement**
  - OS selects a replacement frame from the set of **all** frames
  - that means one process can steal a frame from another
  - disadvantage: process execution time can vary greatly
  - advantage: greater throughput, so more common
- **local replacement**
  - each process selects only from its own set of allocated frames
  - more consistent per-process performance
  - but can lead to underutilized memory

Page Replacement Algorithms

Page replacement algorithms

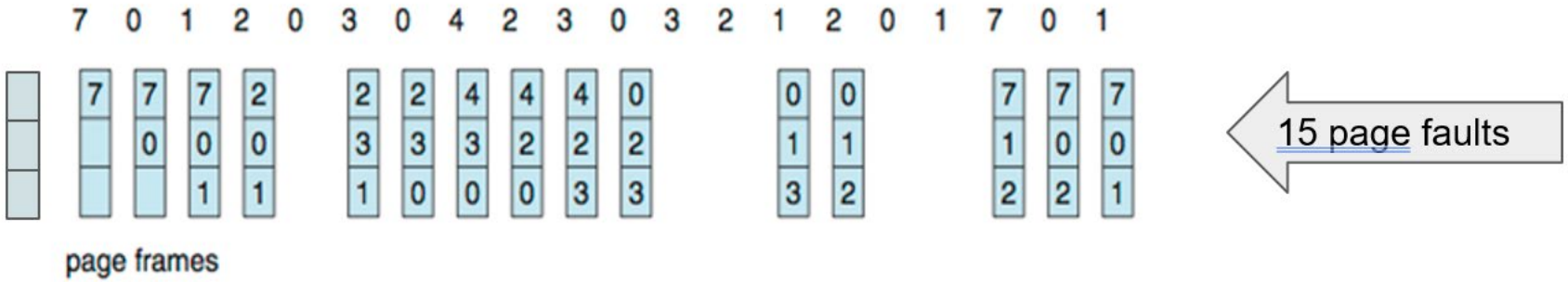
- page replacement algorithm
  - used when OS needs a frame, but all frames are occupied
  - determines the victim frame
  - in a way that minimizes number of page faults
- we will look at different algorithms by running them on a particular string of memory references (reference string) and computing the number of page faults on that string
  - reference string is just a list of page numbers, not full addresses
  - repeated access to the same page does not cause a page fault
  - results will change based on number of frames available
- in all our examples, the reference string will be:

```
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
```

FIFO

First-In-First-Out (FIFO) Algorithm

- FIFO - replaces page that has been in memory for the longest time
- can be implemented using a FIFO queue
- example: 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
  - 3 available frames (3 pages can be in memory at a time per process)



- # page faults varies:
  - by reference string, eg. consider rstring 1,2,3,4,1,2,5,1,2,3,4,5
  - by number of frames available

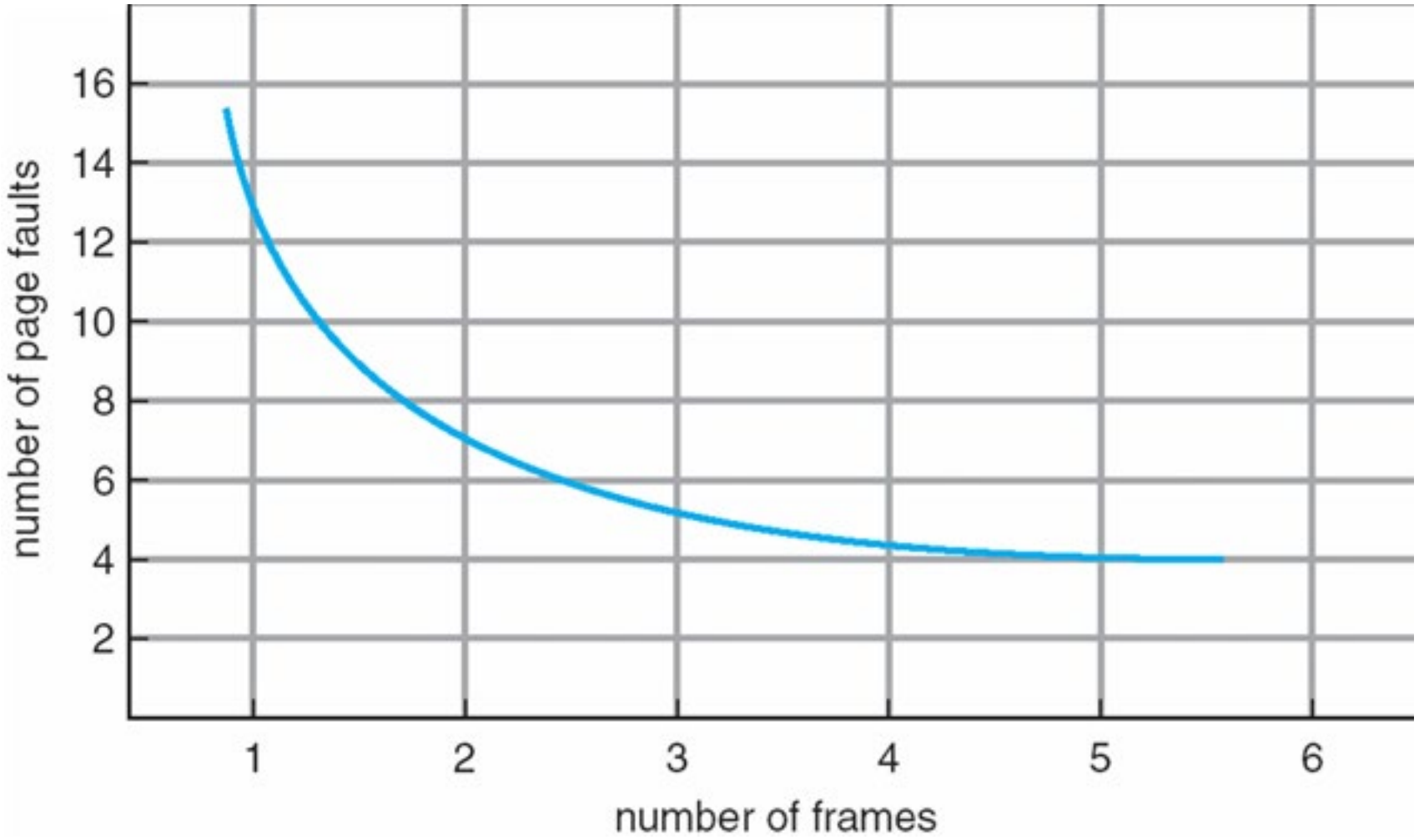
FIFO example 2

- example 2:
  - 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`
  - this time with 4 available frames

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

  - only 10 page faults (compared to 15 faults with 3 available frames)

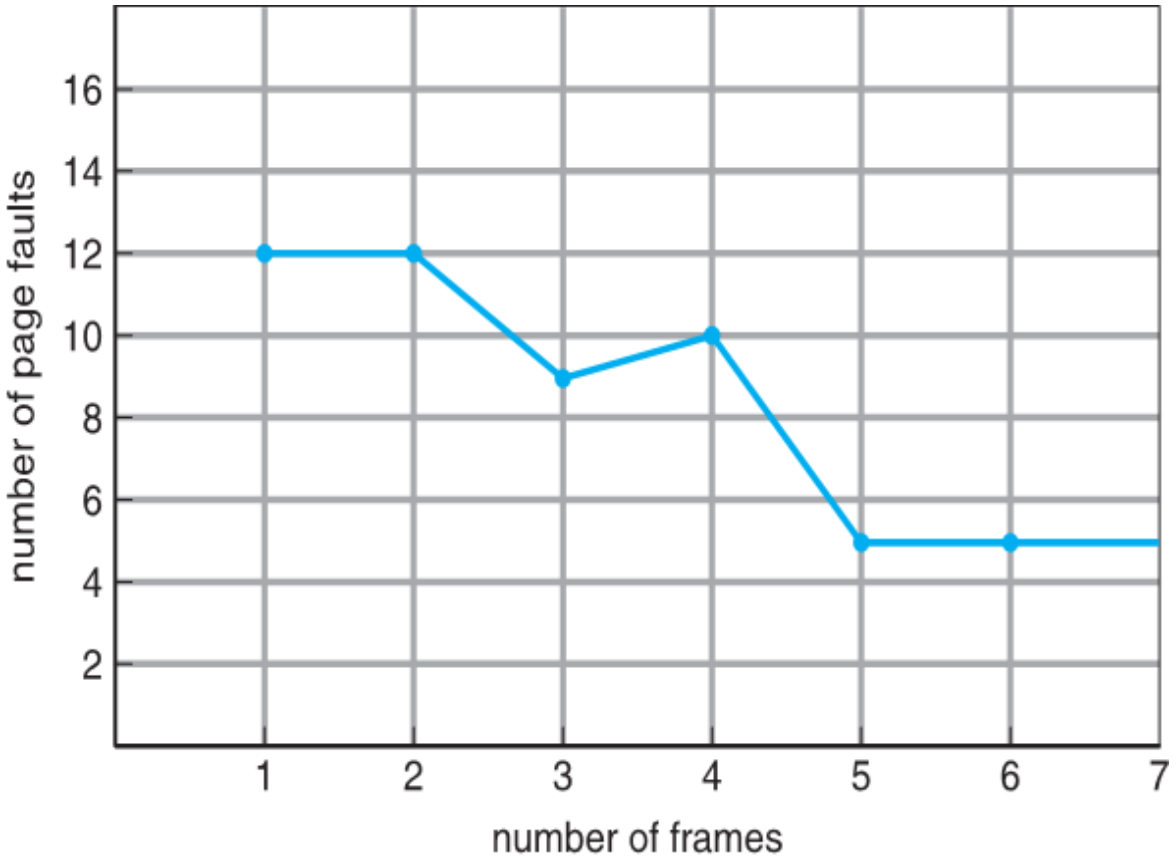
Graph of Page Faults Versus The Number of Frames



Béládys

Bélády's anomaly

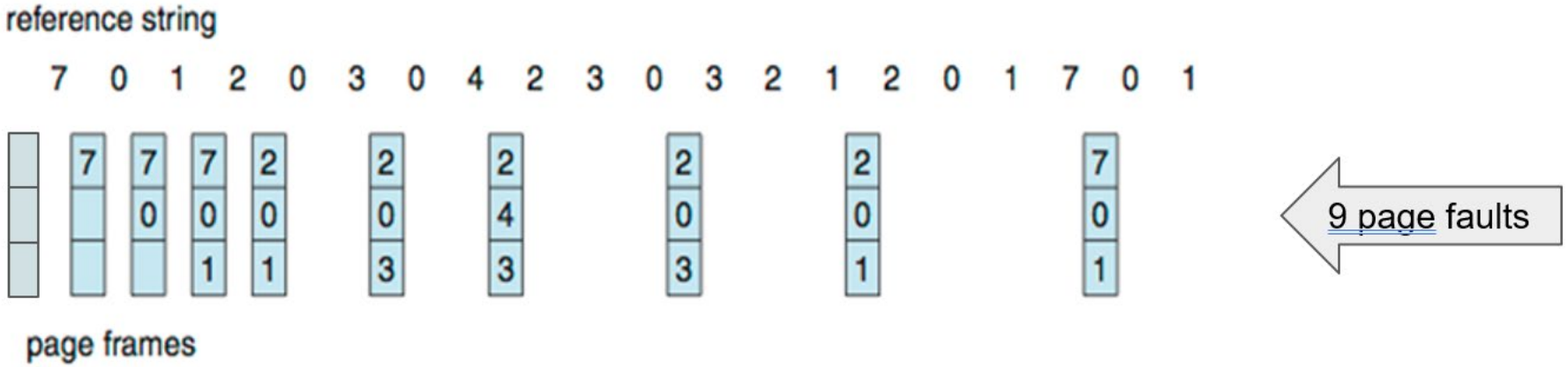
- Bélády's anomaly occurs when increasing the number of page frames results in an increase in the number of page faults for certain memory access patterns
- example:
  - r-string `0 1 2 3 0 1 4 0 1 2 3 4`
  - FIFO with 3 frames → 9 page faults
  - FIFO with 4 frames → 10 page faults



OPT

Optimal algorithm (OPT)

- replaces page that *will* not be used for longest period of time

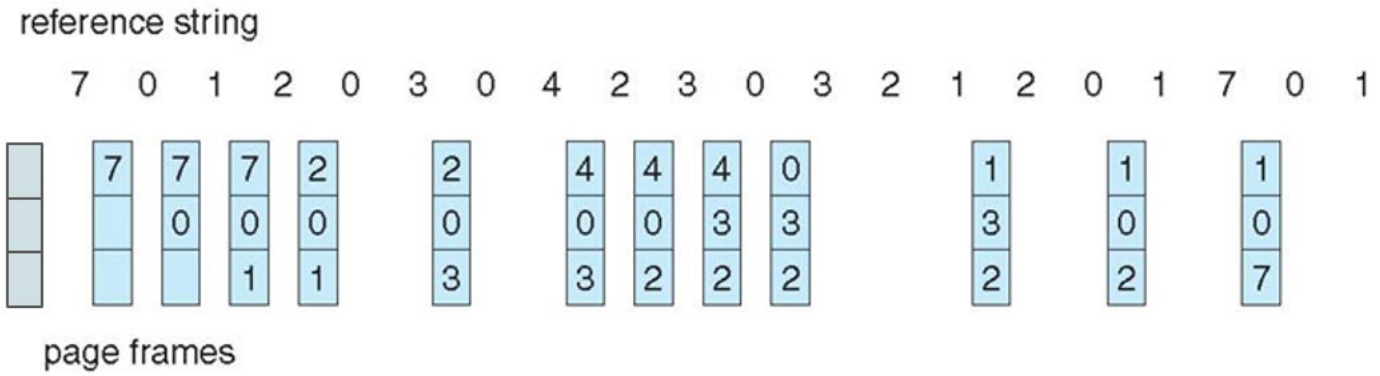


- not practical — requires knowing the future
- but useful for measuring how well other non-optimal algorithms perform
- eg. no algorithm can do better than 9 page faults for the above reference string and 3 available frames

LRU

Least Recently Used Algorithm (LRU)

- uses past knowledge to predict future
- replaces page that has not been used in the most amount of time
- associates time of last use with each page



- 12 faults
- better than FIFO but worse than OPT
- but how can we implement it?

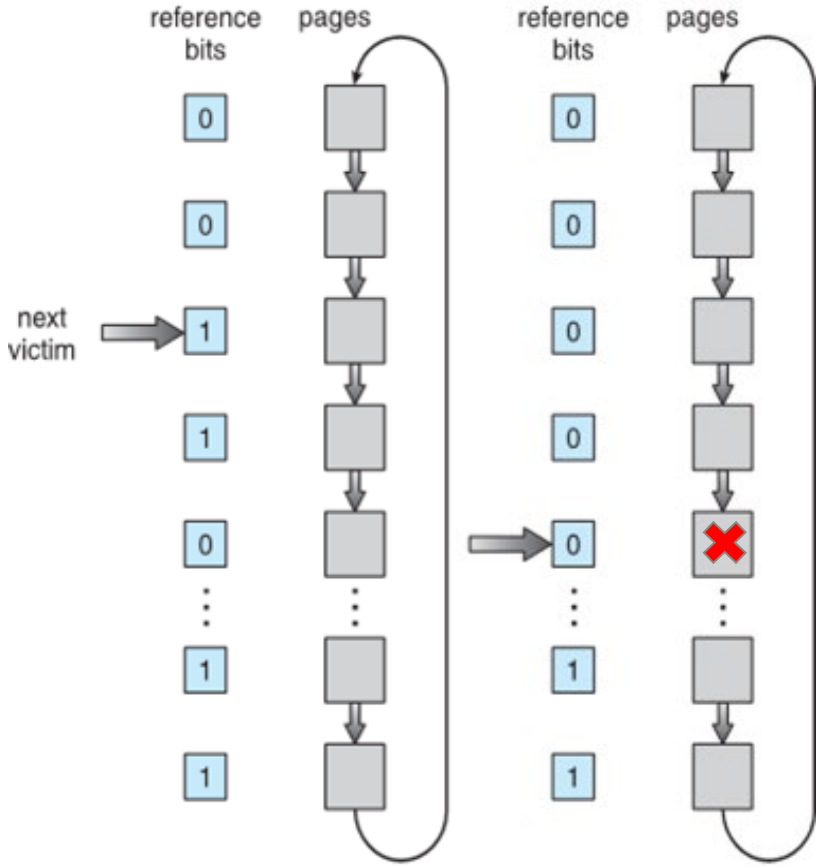
LRU implementation

- counter implementation
  - every page entry has a counter
  - every time page is referenced through this entry, copy current clock into the counter
  - when a page needs to be changed, look at the counters to find smallest value
  - requires search through table
- stack implementation
  - keep a stack of page numbers (eg. doubly linked list)
  - when page referenced - move it to the bottom
  - stack top contains the least recently used page
  - each update is more expensive than counter-based implementation
  - although no search needed for replacement
- LRU and OPT are examples of replacement algorithms that don't exhibit Belady's Anomaly
- pure LRU needs special hardware and is still slow, but there are fast approximations of LRU

CLOCK

CLOCK replacement algorithm

- clock replacement is an approximation of LRU
- uses the reference bit in page table entry, which is **automatically set by hardware** any time page is accessed
- frames are organized as a circular buffer
- maintain one pointer (clock hand) pointing to the page to be inspected next
  - if page under pointer has ref. bit = 0, replace it
  - otherwise set reference bit to 0 and advance pointer to the next page
- this essentially gives a page a 'second chance'
- simple algo. with good performance, can be extended/improved if more bits available
- many more page replacement algorithms, eg. WSClock, Aging LRU



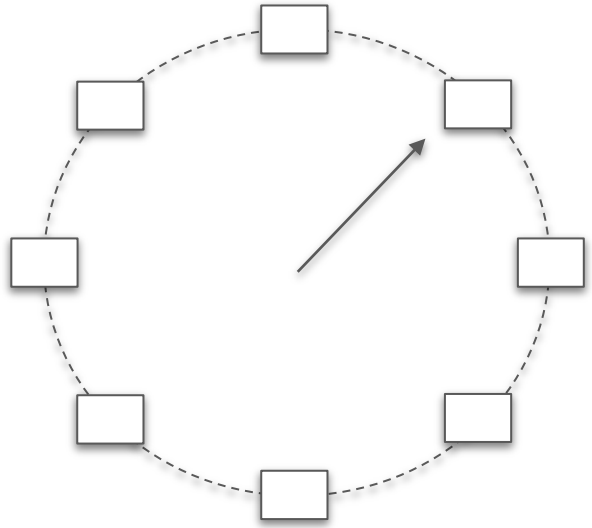
UNIVERSITY OF CALGARY

CLOCK replacement simulation

```
# inputs
n_frames = number of available frames
ref_string = array representing array string

# initialization:
pages = array of n_frames * integers, initialized to -1
refs = array of n_frames * booleans, initialized to false
hand = 0

# process each page in reference string
for p in ref_string:
    find 'ind' such that pages[ind] == p
    if ind was found: # no page fault
        refs[ind] = true # simulating H/W
        continue
    else # page fault
        while refs[hand] == true:  # give page second chance
            refs[hand] = false
            hand = (hand + 1) % n_frames
        pages[hand] = p # evict and replace
        refs[hand] = true # do we need this?!?
        hand = (hand + 1) % n_frames
```



Comparative Example

Optimal with 4 frames											
1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1			1				4	
	2	2	2			2				2	
		3	3			3				3	
			4			5				5	

6 page faults

LRU with 4 frames											
1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1			1			1	1	5
	2	2	2			2			2	2	2
		3	3			5			5	4	4
			4			4			3	3	3

8 page faults

Clock with 4 frames												
	1	2	3	4	1	2	5	1	2	3	4	5
*	1:1	1:1	1:1	*1:1			5:1	5:1	5:1	*5:1	4:1	4:1
	*	2:1	2:1	2:1			*2:0	1:1	1:1	1:1	*1:0	5:1
		*	3:1	3:1			3:0	*3:0	2:1	2:1	2:0	*2:0
			*	4:1			4:0	4:0	*4:0	3:1	3:0	3:0

10 page faults

the star (*) represents the hand position
the number after colon (:) represents the value of the referenced bit

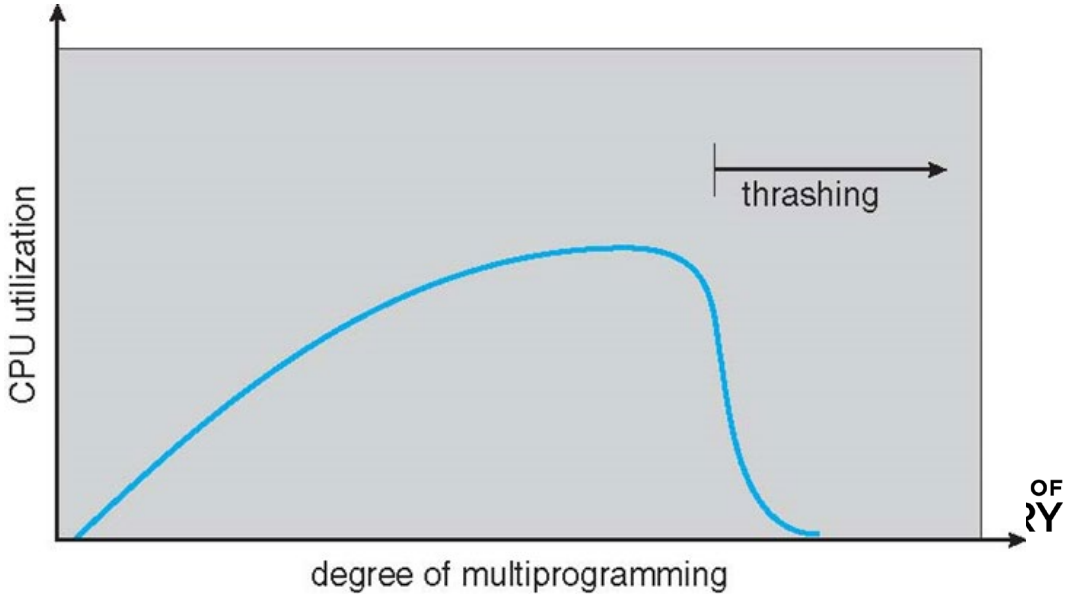
Clock with 3 frames												
	1	2	3	4	1	2	5	1	2	3	4	5
*	1:1	1:1	*1:1	4:1	4:1	*4:1	5:1	5:1	5:1	5:0	*5:0	*5:1
	*	2:1	2:1	*2:0	1:1	1:1	*1:0	*1:1	*1:1	3:1	3:1	3:1
		*	3:1	3:0	*3:0	2:1	2:0	2:0	2:1	*2:0	4:1	4:1

9 page faults

Trashing

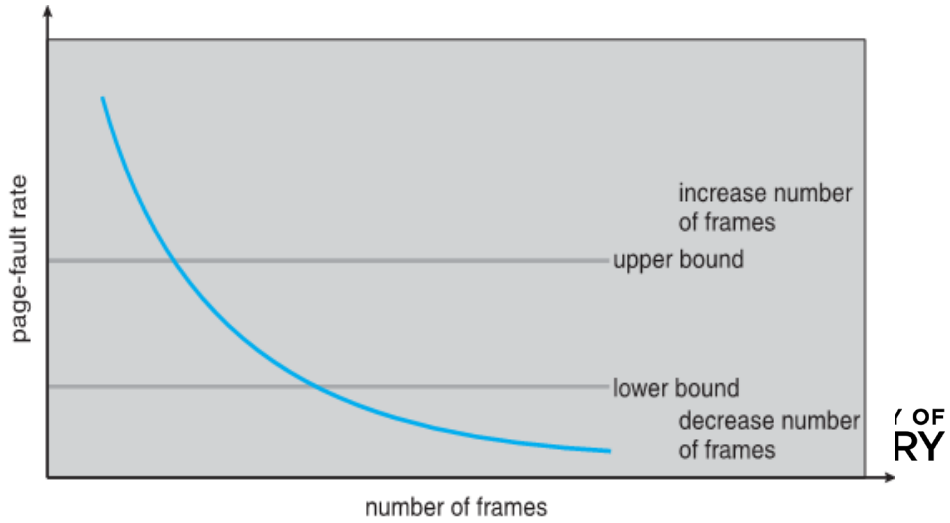
Thrashing

- if a process does not have “enough” pages, the page-fault rate is very high
  - page fault to get page
  - replace existing frame
  - but quickly need replaced frame back
- **thrashing process** = process is progressing slowly due to frequent page swaps
  - a process spends more time waiting for page faults than it spends executing
- this can lead to an entire **system thrashing**:
  - many processes thrashing → low CPU utilization
  - OS thinks that it needs to increase the degree of multiprogramming
  - OS adds another process to the system making things even worse



Dealing with thrashing

- local page replacement
  - when a process is thrashing, OS prevents it from stealing frames from other processes
  - at least the thrashing process cannot cause the entire system to thrash
- working set model
  - OS keeps track of pages that are actively used by a process (working set)
  - working set of processes changes over time
  - OS periodically updates the working set for each process, using a moving time window
  - before resuming a process, OS loads the entire working set of the process
- page fault frequency
  - establish acceptable bounds on page fault rate
  - if actual page fault rate of a process too high
    → process gains a frame
  - if actual page fault rate of a process too low
    → process loses a frame



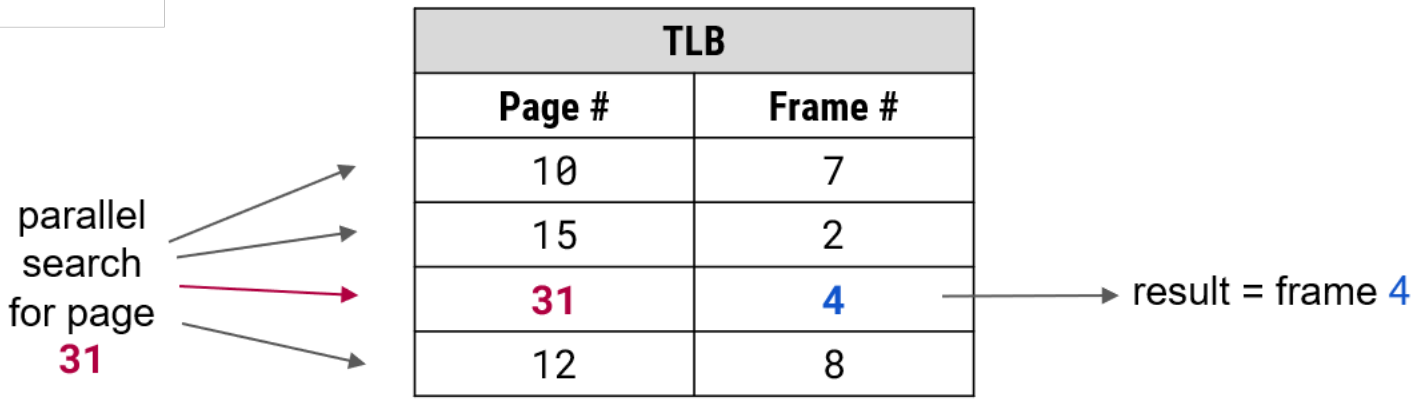
General Page Tables

Page Table Implementation

- page table is kept in main memory
  - page-table base register (PTBR) points to the page table
  - page-table length register (PTLR) indicates size of the page table
- every instruction access requires **at least** two memory accesses
  - one for page table lookup + one more for instruction fetch
- this can be reduced by using a translation lookaside buffer (TLB)
  - TLB is a special hardware cache
  - TLBs are extremely fast, but have very small capacity
  - TLBs can remember a small part of the page table, ~64 to ~1K entries
  - on TLB miss, value is saved in TLB for faster access next time

TLB as associative memory

- TLB is often implemented as associative memory - hardware capable of fast parallel search based on content



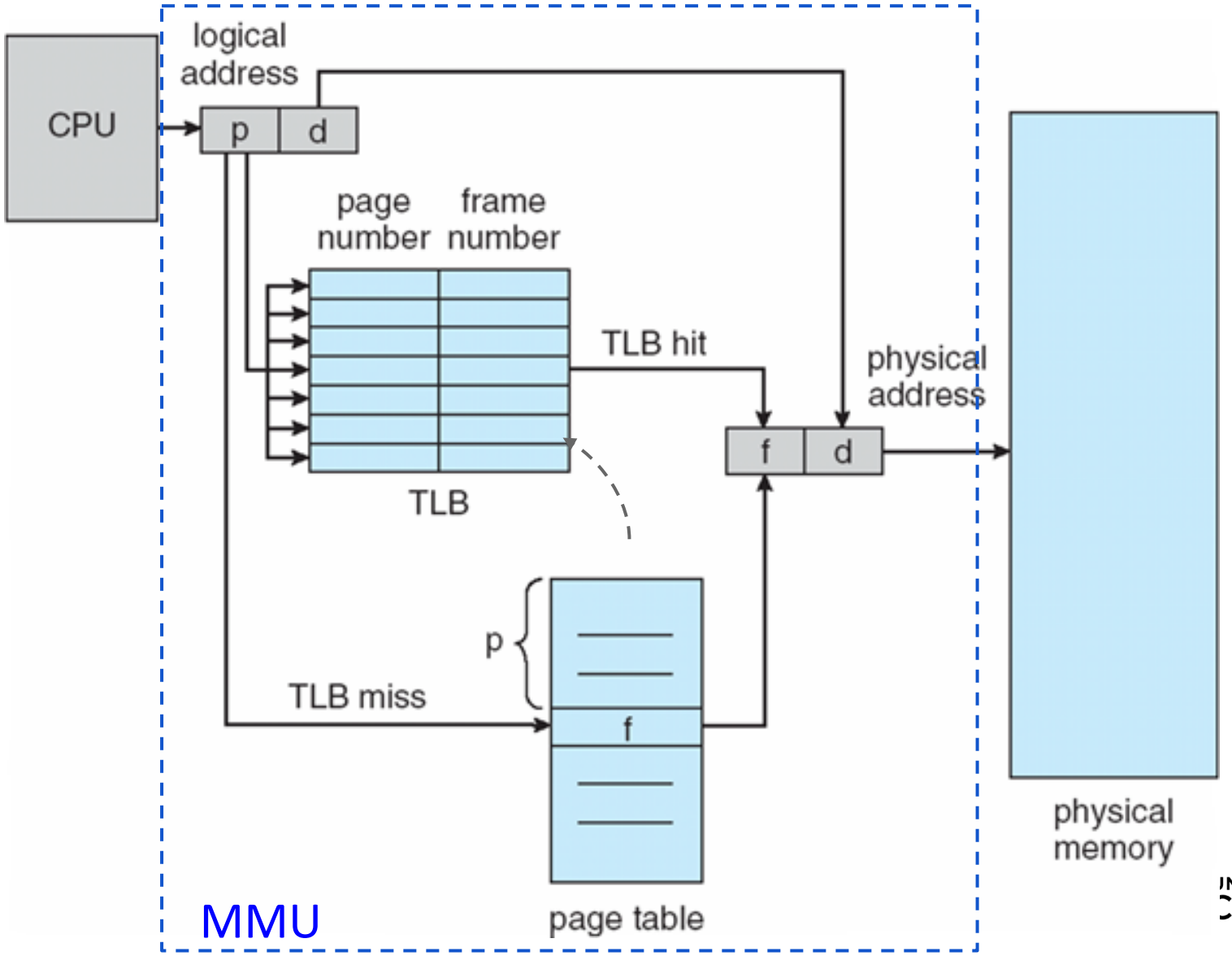
- given a page#, TLB will return corresponding frame# in constant amount of time (TLB-hit)
- if TLB does not contain entry for page#, the search continues in page table in memory (TLB-miss)
- effective memory-access time = $(1 - p) * (tlbs + 2 * ma) + p * (tlbs + ma)$

p = probability of TLB-hit (TLB-hit ratio)

tlbs = TLB search time

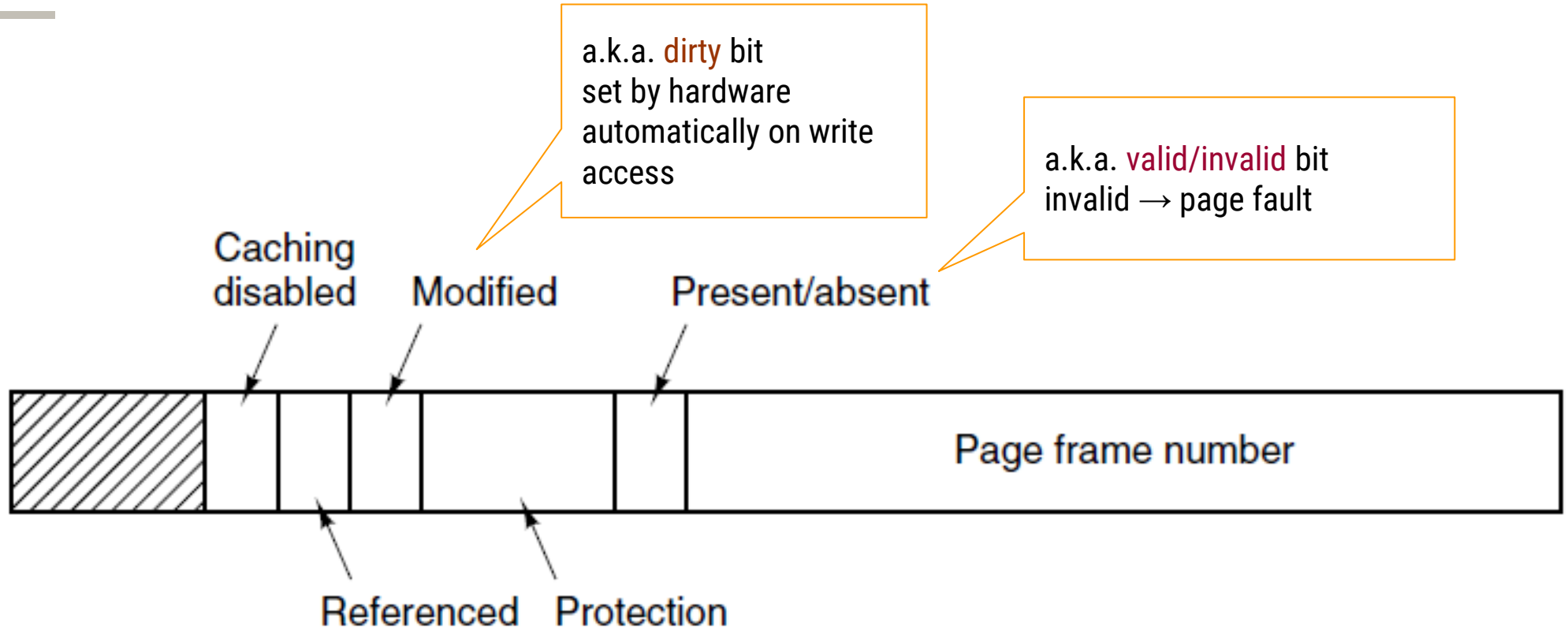
ma = memory access time

Paging hardware with TLB



$$EAT = ma + tlbs + (1 - p) * ma$$

Typical structure of page table entry



Memory protection

- memory protection is usually implemented by associating a **protection bit** with each frame
  - the bit indicates if read-only or read-write access is allowed
  - note: we can also add other protection bits, such as execute-only bit
- **valid bit** — another bit in each page table entry:
  - valid=1 indicates that the corresponding frame is in physical memory
  - valid=0 (“invalid”) the corresponding frame is not in physical memory
- violations result in trap to the kernel, e.g.:
  - accessing page with invalid bit set → page fault
  - accessing page past the PTLR → page fault
  - trying to write to a page with read-only bit set → general protection fault

Shared Pages

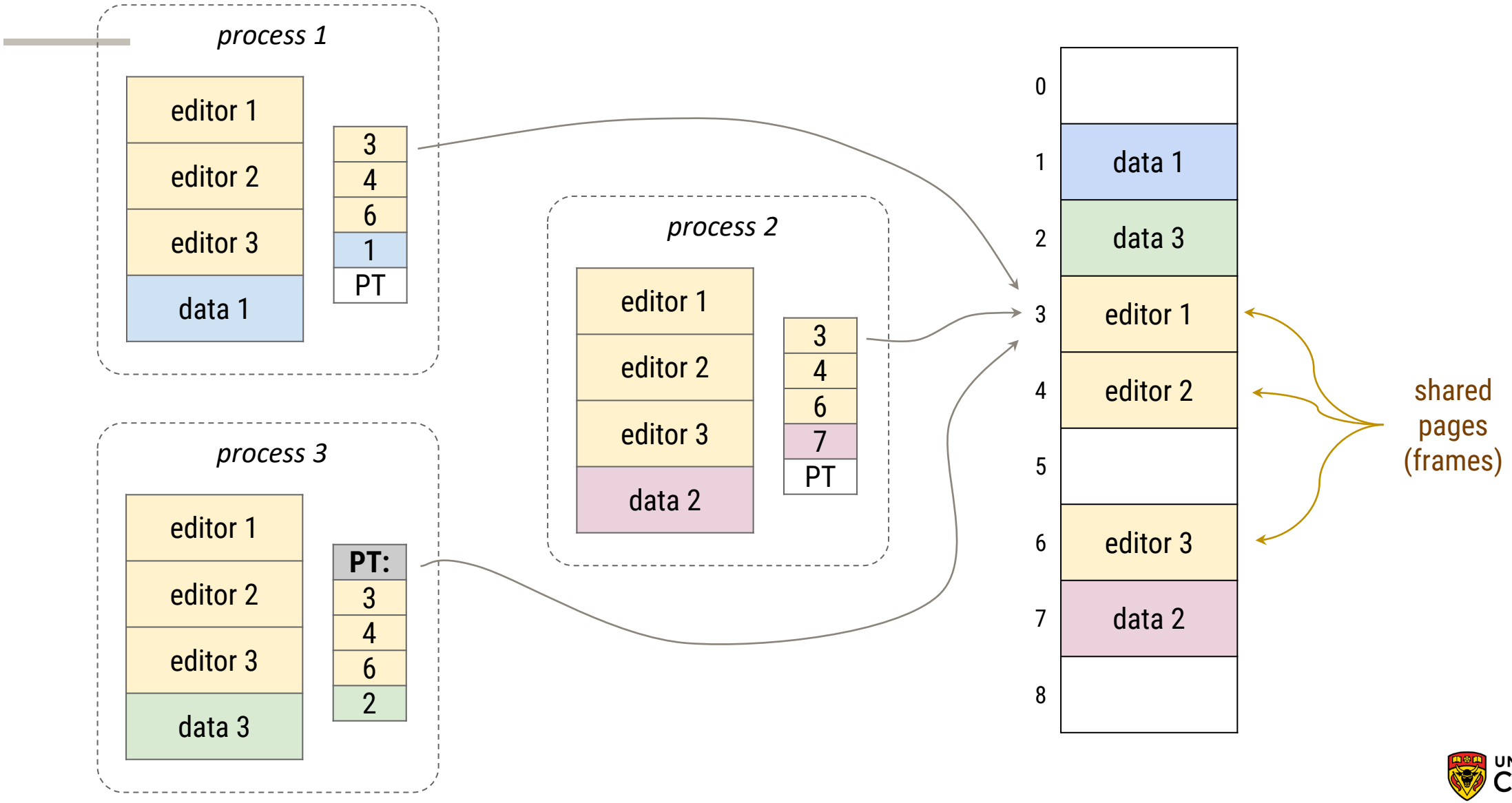
Shared Pages

- sometimes it can be useful for processes to share memory with other processes
- this can be implemented using shared pages

- example 1:
  - running multiple instances of the same program,
    or different programs using the same shared library
  - only one copy of the executable code needs to be in physical memory
  - implemented using shared read-only pages, with read-only bit set in page table entry

- example 2:
  - shared memory for interprocess communication
  - implemented using shared read-write pages

Shared Pages Example

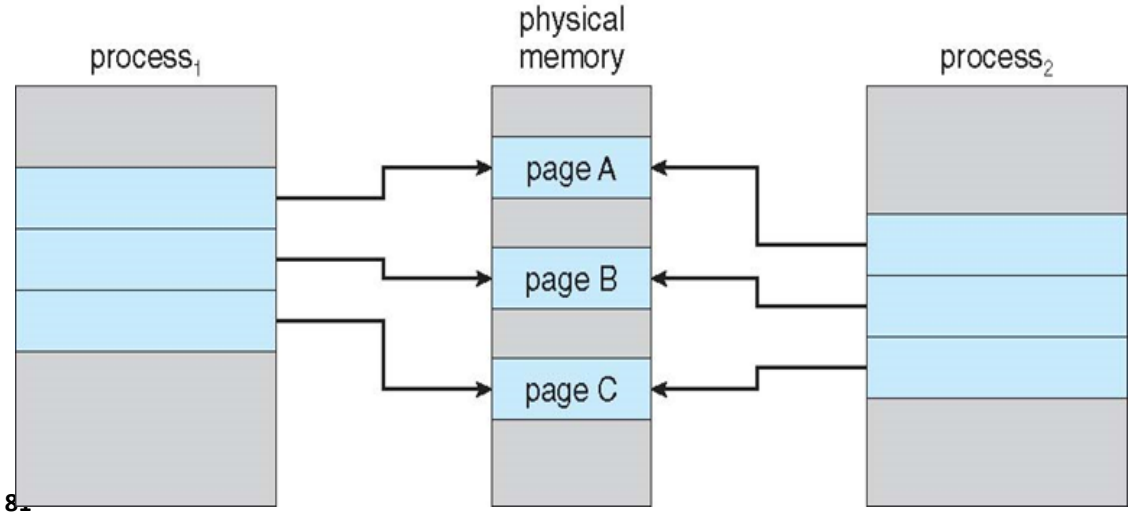


Copy On Write

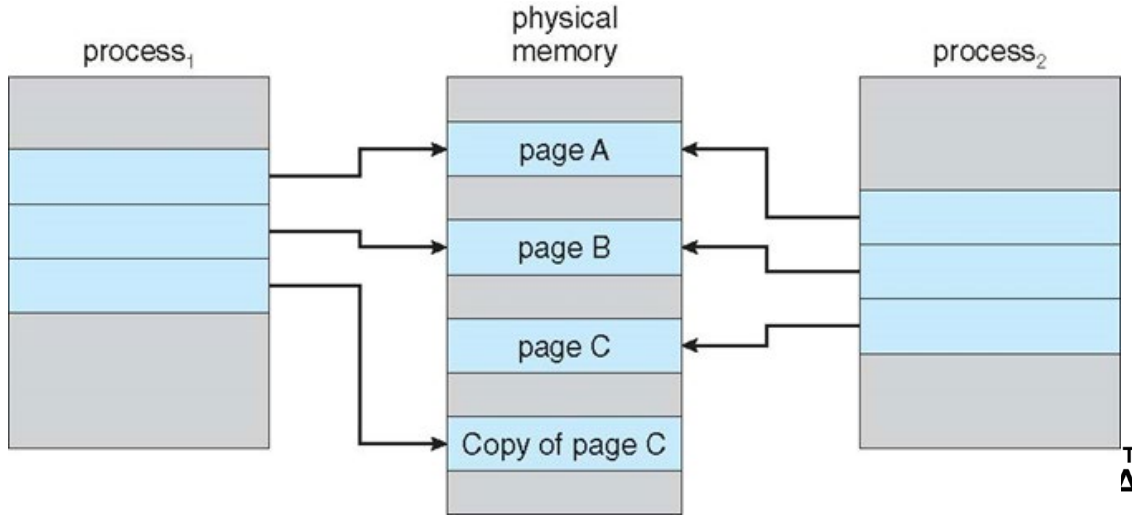
Copy-on-Write

- copy-on-write (COW) allows parent and child processes to initially share some pages
- only if either process tries to modify a shared page, the page is copied and then modified
- implemented using copy-on-write bit in page table entries
- COW allows very efficient process implementation of `fork()`, since only modified pages are copied (on demand)

Before process 1 tries to modify page C



After process 1 tries to modify page C



Page Table Size

Page table size

- simple page tables can become very large

- consider a 32-bit logical address space, with page size of 4 KB (2^{12} bytes)
  - page table would have to contain ~1 million entries (2^{32} / 2^{12} = 2^{20})
  - if each entry is 4 bytes → page table would use up 4MB of memory (homework: verify)
  - for 64-bit systems, page table can get impractically big (homework: do the math)

Page Table Size

- consider a 64-bit logical address space (most common)
  - page size of 4 KB (2^{12})
  - page table would have ($2^{64} / 2^{12} = 2^{52}$) entries
  - 52 bits to address it → each entry would need to be at least $\lceil 52/8 \rceil = 7$ bytes long
  - page table would need at minimum 2^{52} entries * 7 bytes/entry → petabyte range!!!
- some solutions:
  - hierarchical paging
  - inverted page tables
  - hashed inverted page tables

in 2021 petabyte is still a problem

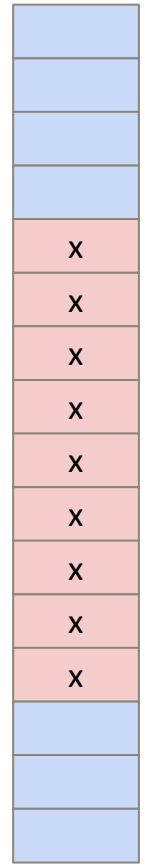
Page Table Implementations

Hierarchical Page Tables

Hierarchical Page Tables

- observation: most programs do not use all address space at the same time
  - instead, only some entries in the page table are used at any given time (temporal locality of reference)
  - also, the used entries tend to be clusters / groups of consecutive pages (spatial locality of reference)

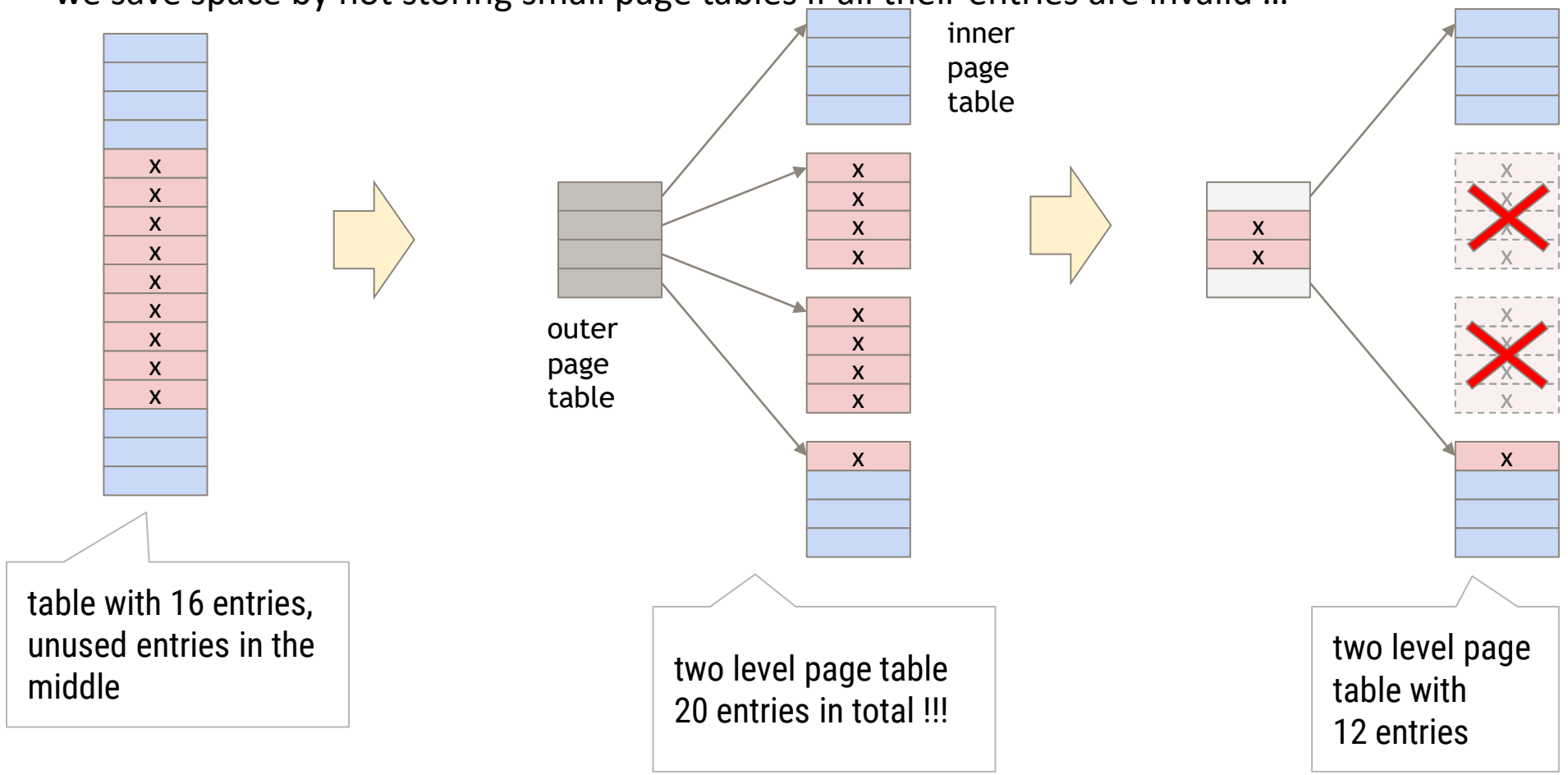
- so let's break up the page table into multiple smaller page tables
  - with the hope that not all of the smaller page tables will be used
  - the ones that are not used, don't have to be in memory



page table with unused entries in the middle

Two-Level Page-Table Example

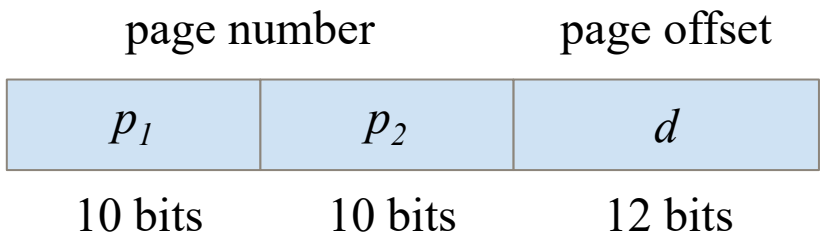
- a simple technique is a two-level page table — think of it as paging the page table
- we save space by not storing small page tables if all their entries are invalid …



Two-Level Page-Table Example

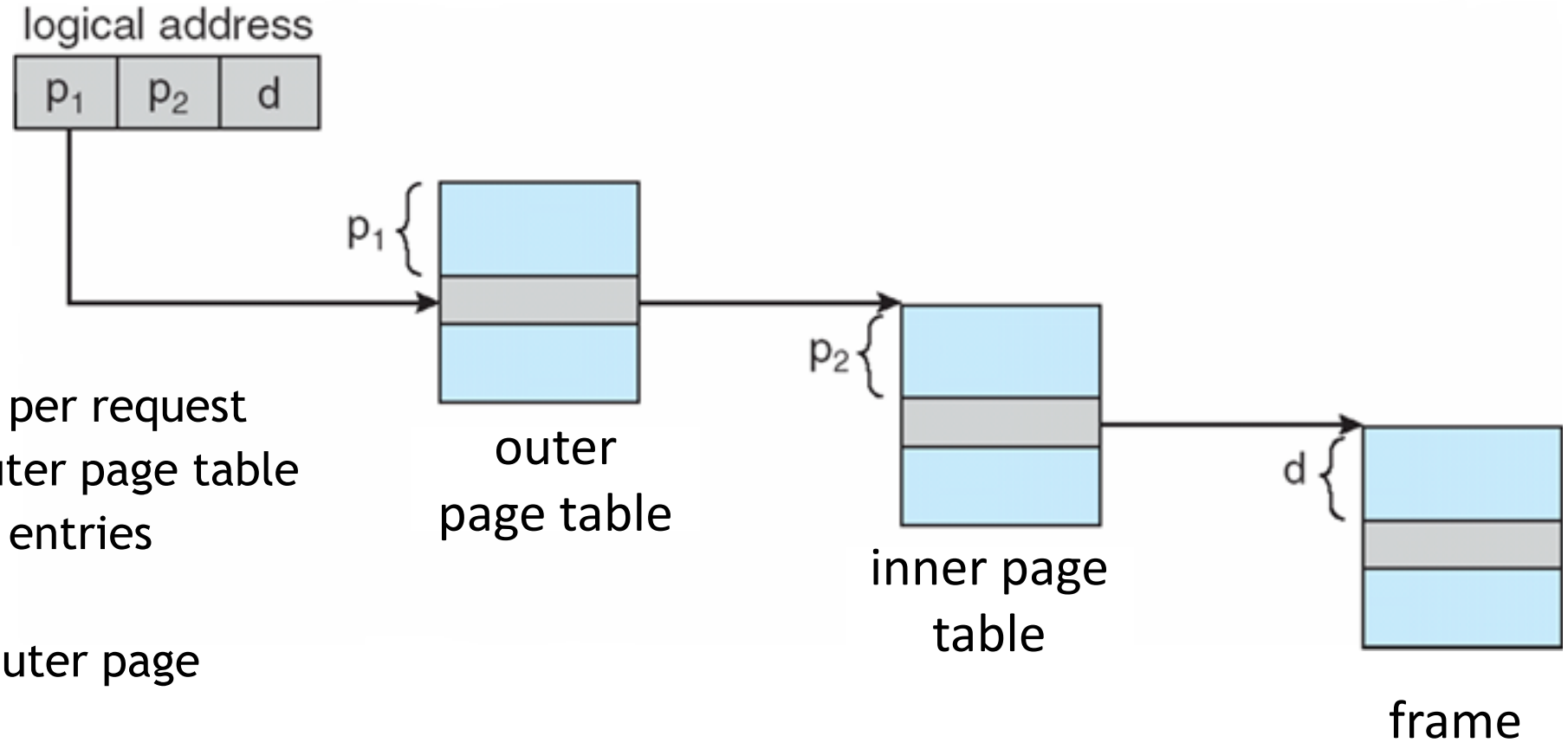
- a 32-bit logical address with 4K page size is divided into:

  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- we can add another level of indirection – and divide the page number further:
  - a 10-bit outer page number *p1*
  - a 10-bit inner page number *p2*



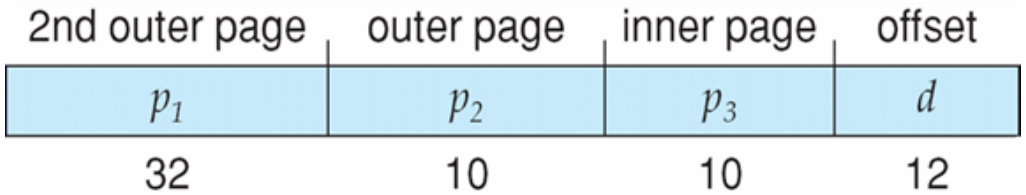
  - p_1 is an index into the **outer page table**
  - p_2 is an index into an **inner page table**
  - also known as forward-mapped page table

Address-Translation Scheme



- 3 memory accesses per request
- on 64-bit system outer page table would still have 2^{42} entries

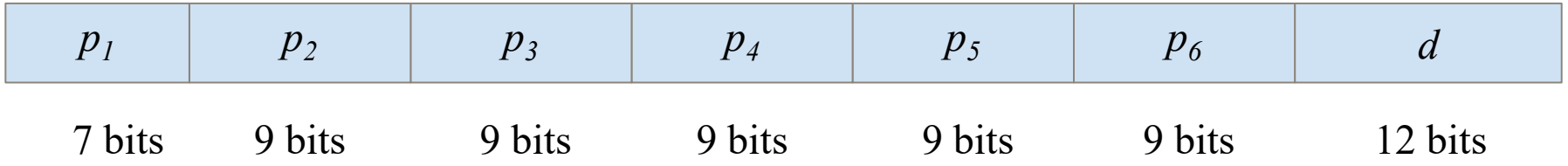
- we could add 2nd outer page table
  - 2^{32} entries for outer PT
  - 4 memory accesses per req.



Multi-Level Page Table

Multi-level page table example

- consider a 64-bit system, with 4KiB page size (2^{12} bits), and 8 bytes per entry
- single page table → 2^52 entries x 8B/entry → 2^55 bytes → ~36 petabytes
- how many page table levels do we need if we want each page table fit inside a frame?
  - a frame can fit 4KiB/8B = 512 = 2^9 entries
  - with 12-bit offsets, that means we need [52bits/9bits-per-level] = 6 levels!!!



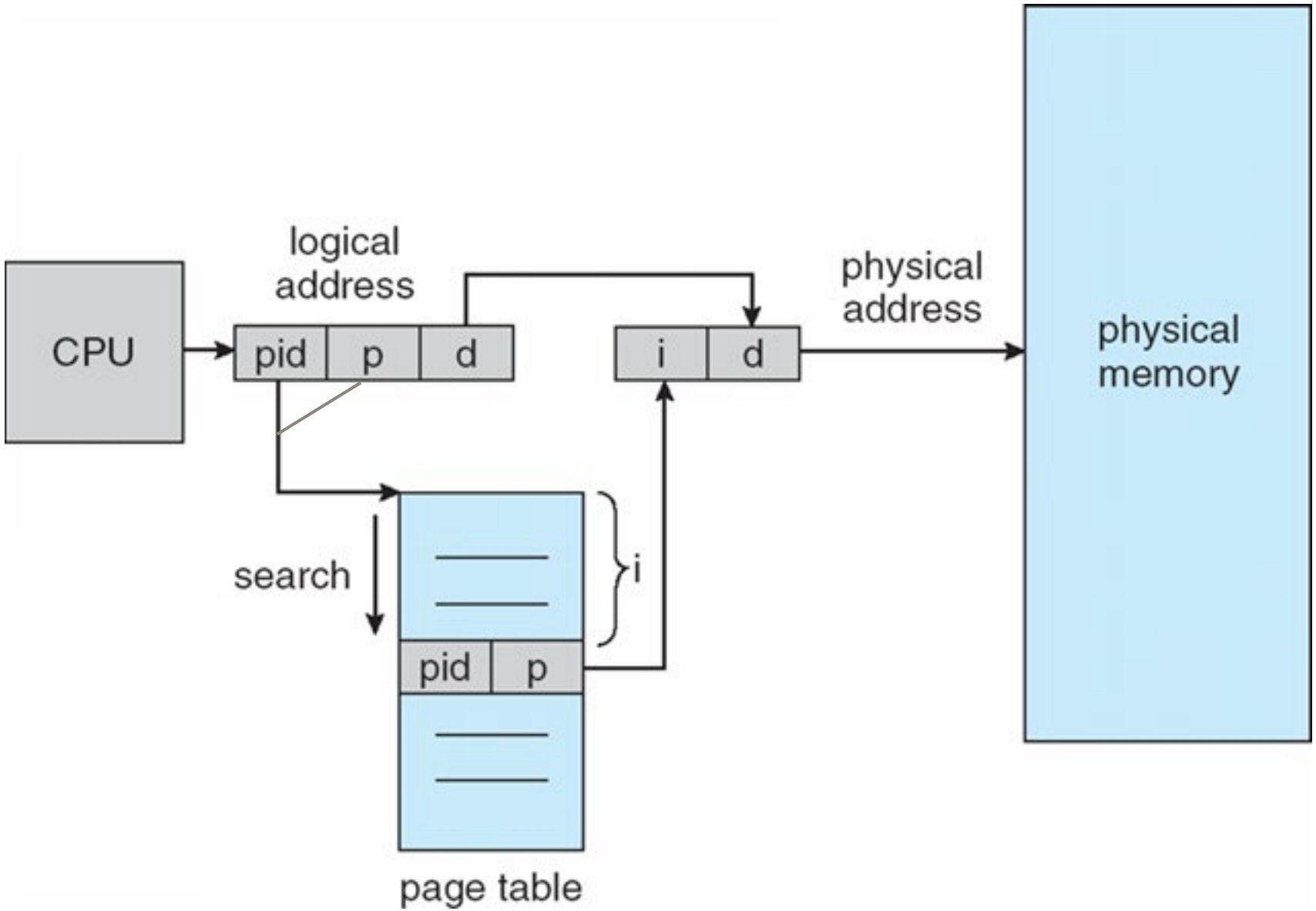
  - with 6-level hierarchical PT each memory request would require **7** memory accesses
    - 6 accesses for translating logical address → physical address
    - 1 access for the actual memory location
  - newest Intel processors support 5-level page tables, AMD supports 4-level PTs

Inverted Page Table

Inverted Page Table

- rather than each process having its own page table,
  let's track all physical pages in one global **inverted page table** (IPT)
- this global IPT has one entry for each real page of memory, containing:
  - virtual address, and
  - owning process ID
- IPT decreases memory needed to store a page table:
  - IPT size is proportional to the amount of physical memory available
  - eg. 16GB memory with 4KB page size and 8B/entry → only 32MB page table
- but IPT increases time needed to search the table when a page reference occurs
  - with above example, page table has ~4 million entries, on average a translation would require ~2 million memory accesses !!!
  - TLB could help accelerate the lookup somewhat, but TLB is very tiny ...
- another (smaller) problem: shared pages are problematic with IPT

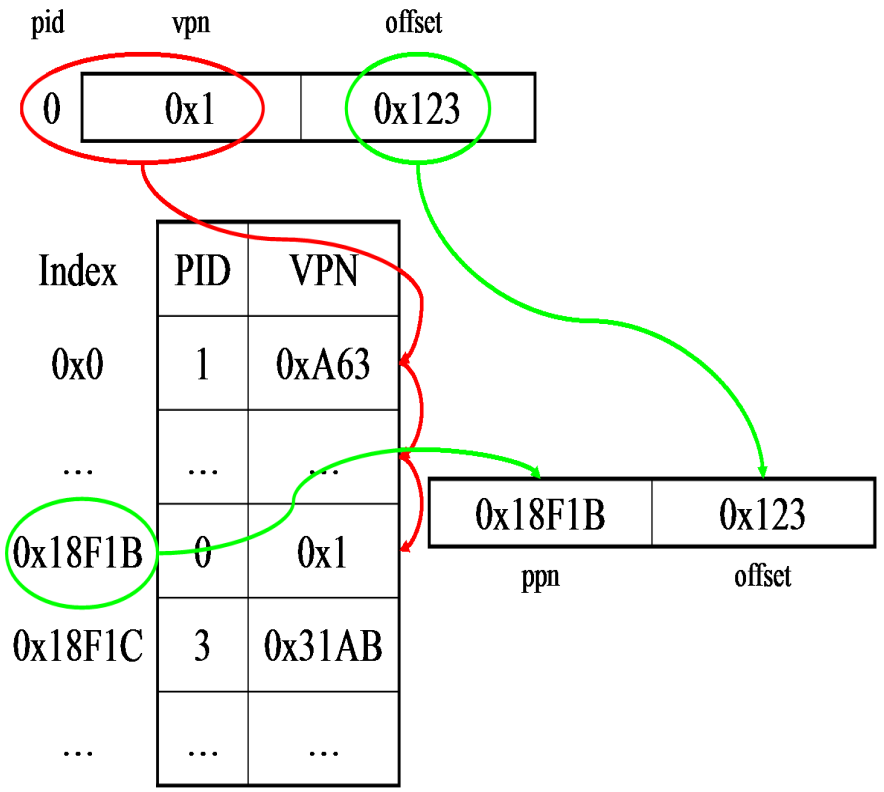
Inverted Page Table Architecture

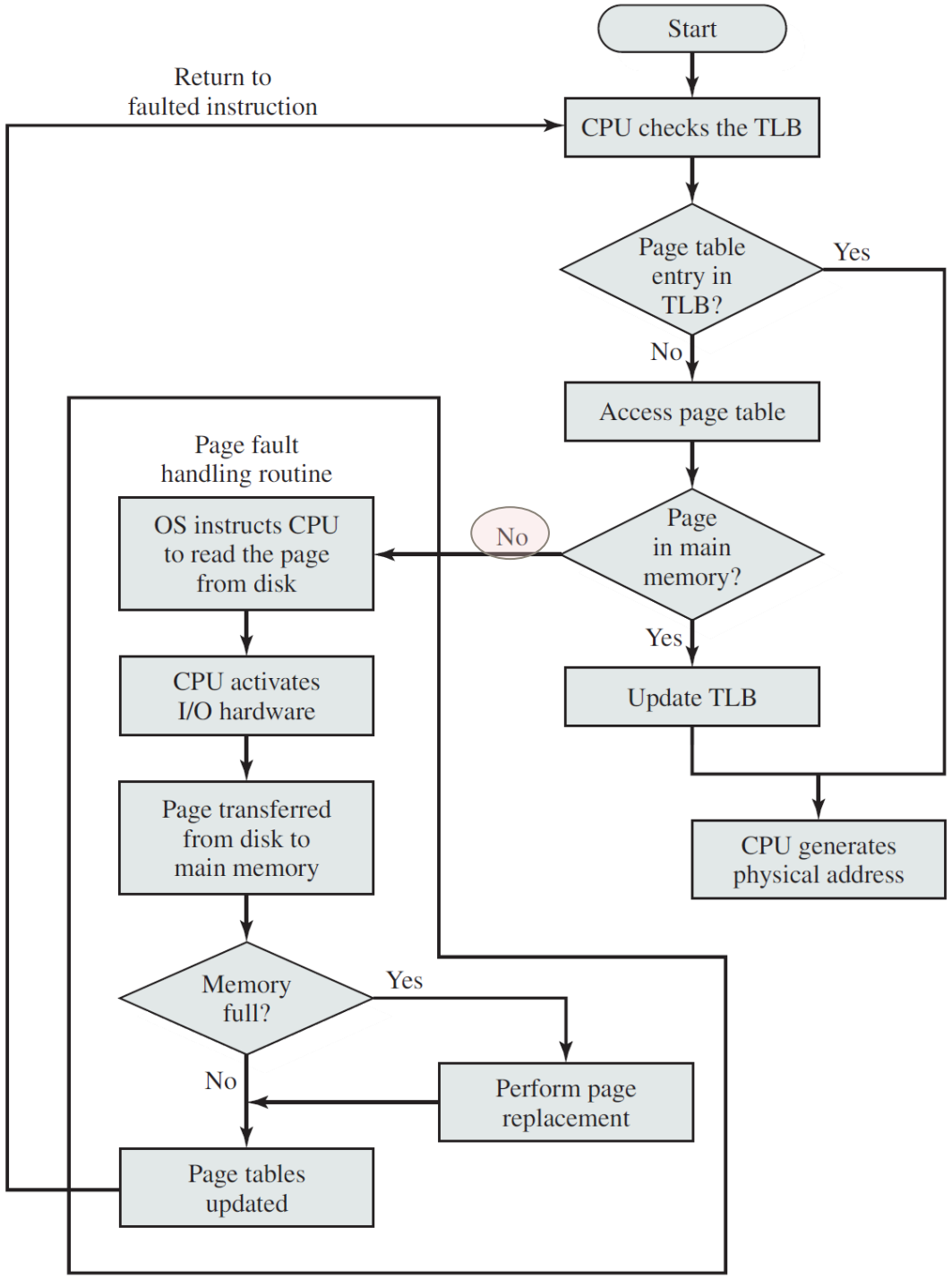


Hash Inverted Page Table

Hashed Inverted Page Table

- used in some 64-bit systems
- the virtual page number and process ID is hashed into a page table
- this page table contains chains of elements which hash to the same location
- each element contains: PID, virtual page number and a pointer to the next element
- virtual page numbers are compared in the chain until a match is found
- if a match is found, the corresponding physical frame is extracted, otherwise page fault
- with a good hash function → average access time is O(1)





Frame Allocation

Frame allocation

- **frame allocation algorithm** determines how many frames to give to each process
- for a single-process system the OS claims some frames and leaves the rest to the running process

- for a multiprogramming system, each process needs a **minimum** number of frames (OS/architecture dependent)
  - but what about maximum?

- examples of allocation schemes:
  - fixed allocation (equal and proportional)
  - priority allocation

Fixed allocation

- **equal allocation** – for example, if there are 100 frames (after allocating frames for the OS) and 5 processes, OS gives each process 20 frames
- **proportional allocation** – allocate according to the size of process
  - dynamic — as degree of multiprogramming and process sizes change

$$s_i = \text{size of process } p_i$$

$$S = \sum s_i$$

$$m = \text{total number of frames}$$

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

example with 2 processes:

$$m = 62$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 62 \simeq 4$$

$$a_2 = \frac{127}{137} \times 62 \simeq 57$$

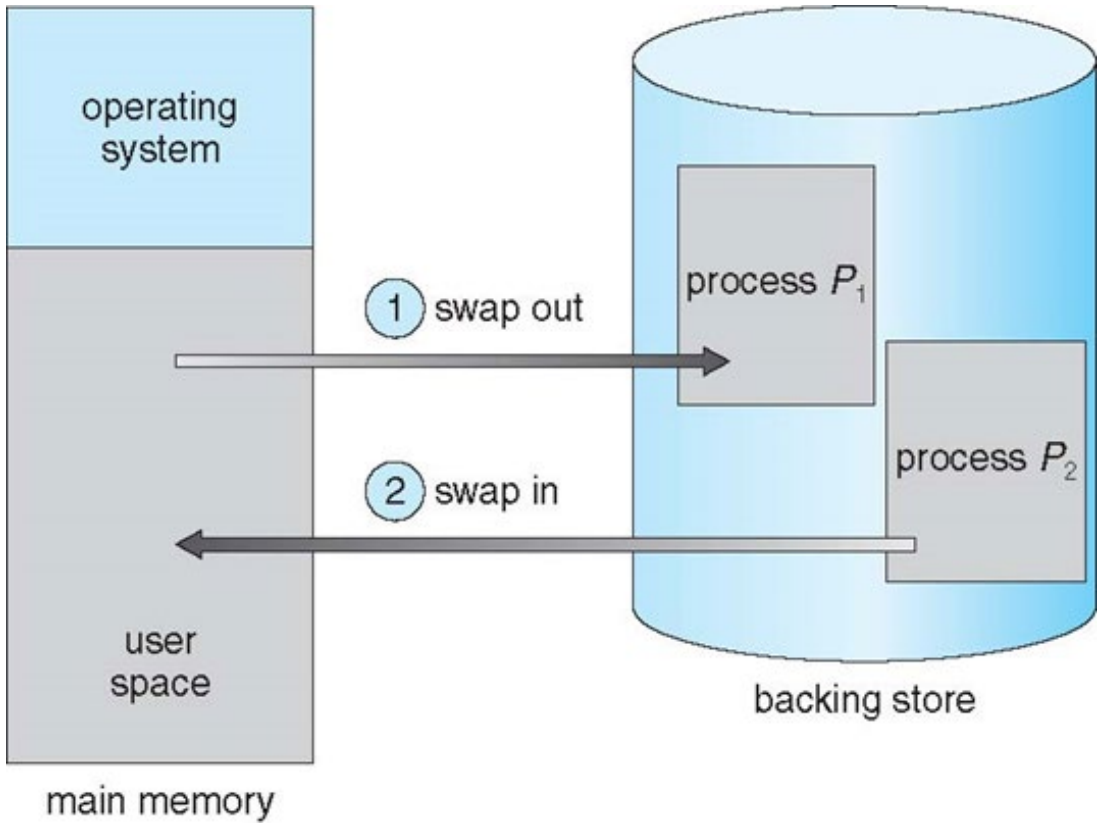
Priority allocation

- similar to proportional allocation scheme
  - but using priorities rather than size
  - the higher the priority of a process, the more frames it gets
  - the lower the priority of a process, the less frames it gets

Swapping

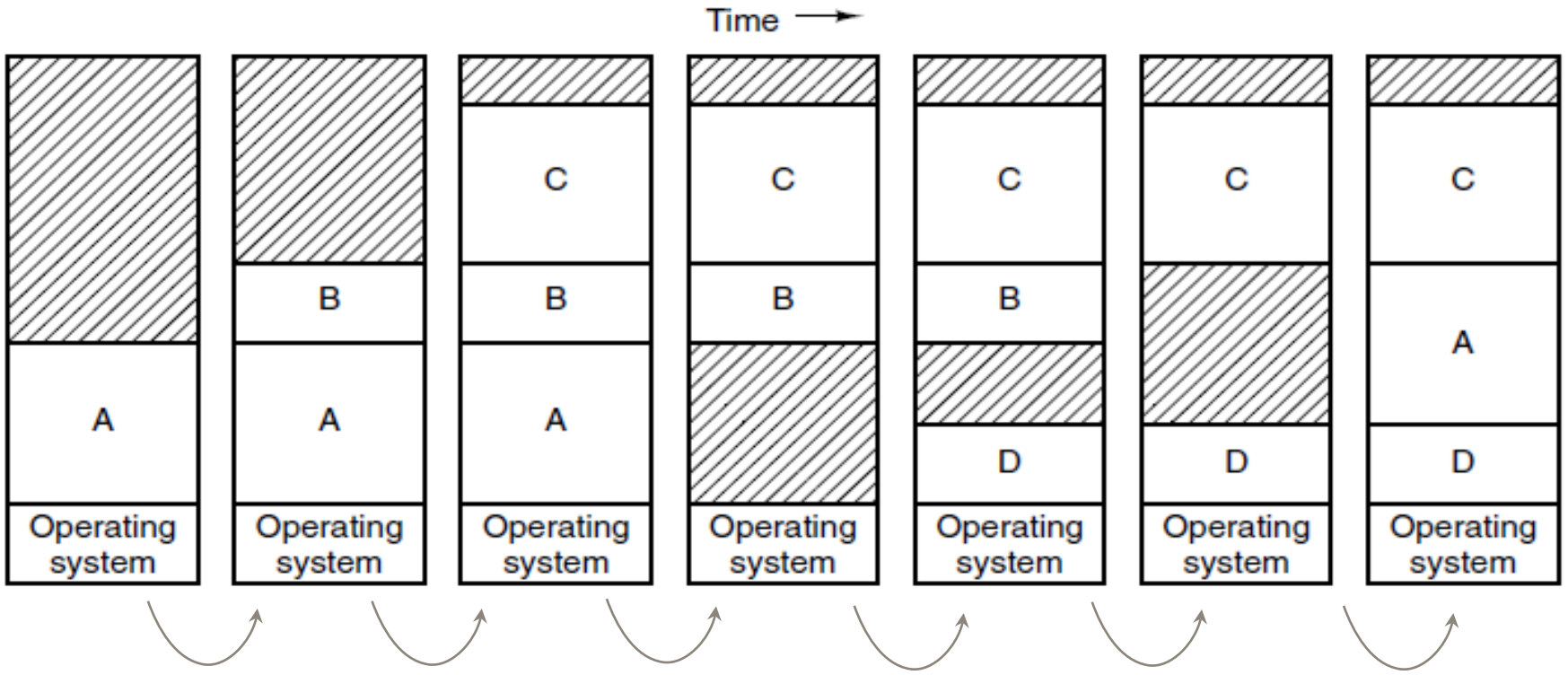
Swapping

- older form of paging (entire process vs parts)
- swapping allows the OS to load more processes than the available physical memory
- a process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **backing store** – fast storage large enough to accommodate copies of all memory images for all processes
- why would we want that?!?
  – programs grow over time
  – program don't always use all allocated memory at the same time



Swapping and memory

- memory allocation changes as processes are swapped out and swapped in
- the shaded regions are unused memory



Swapping

- does the swapped out process need to swap back into the same physical addresses?
  - depends on address binding method
  - much easier if MMU is used
  - must be careful with pending I/O, especially when using memory-mapped device registers
  - I/O results could be sent to kernel, then to the process (double-buffering)

- context switch time can be extremely high

- standard swapping not used in modern operating systems
- note: Linux uses the term 'swapping' to mean paging
  - paging is similar to swapping, but paging can swap out parts of a process

Review

Review

- Name two registers used in a simple MMU implementation.
- Explain logical address / physical address.
- What is the purpose of an MMU?
- Best fit memory allocation is faster than first fit. True or False
- Virtual address space is the same as physical address space. True or False
- Page size is the same as frame size. True or False
- Define: page, frame, demand paging, page table, page fault

Review

- Which one of the following page replacement algorithms requires future knowledge about memory referencing?
  A. FIFO
  B. Optimal
  C. LRU
- What is Belady's Anomaly?
- Which one of the above page replacement algorithms suffers from Belady's Anomaly?

- What is thrashing?

- Describe copy-on-write.

Review - Basic page replacement

1. find the location of the desired page on disk
2. find a free frame:
    - if there is a free frame, go to step 3
    - if there is no free frame, use a **page replacement** algorithm to select a **victim frame**
    - if victim frame is dirty, write it to backing store
    - set the invalid bit in page table corresponding to victim frame
1. load desired page into the free frame and update the page and frame tables
    - frame table is a (simple) data structure that keeps track of free frames
1. restart the instruction that caused the trap

Note: now potentially 2 page transfers for page fault, further increasing EAT

Onward to…
Disk Scheduling and RAID

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY