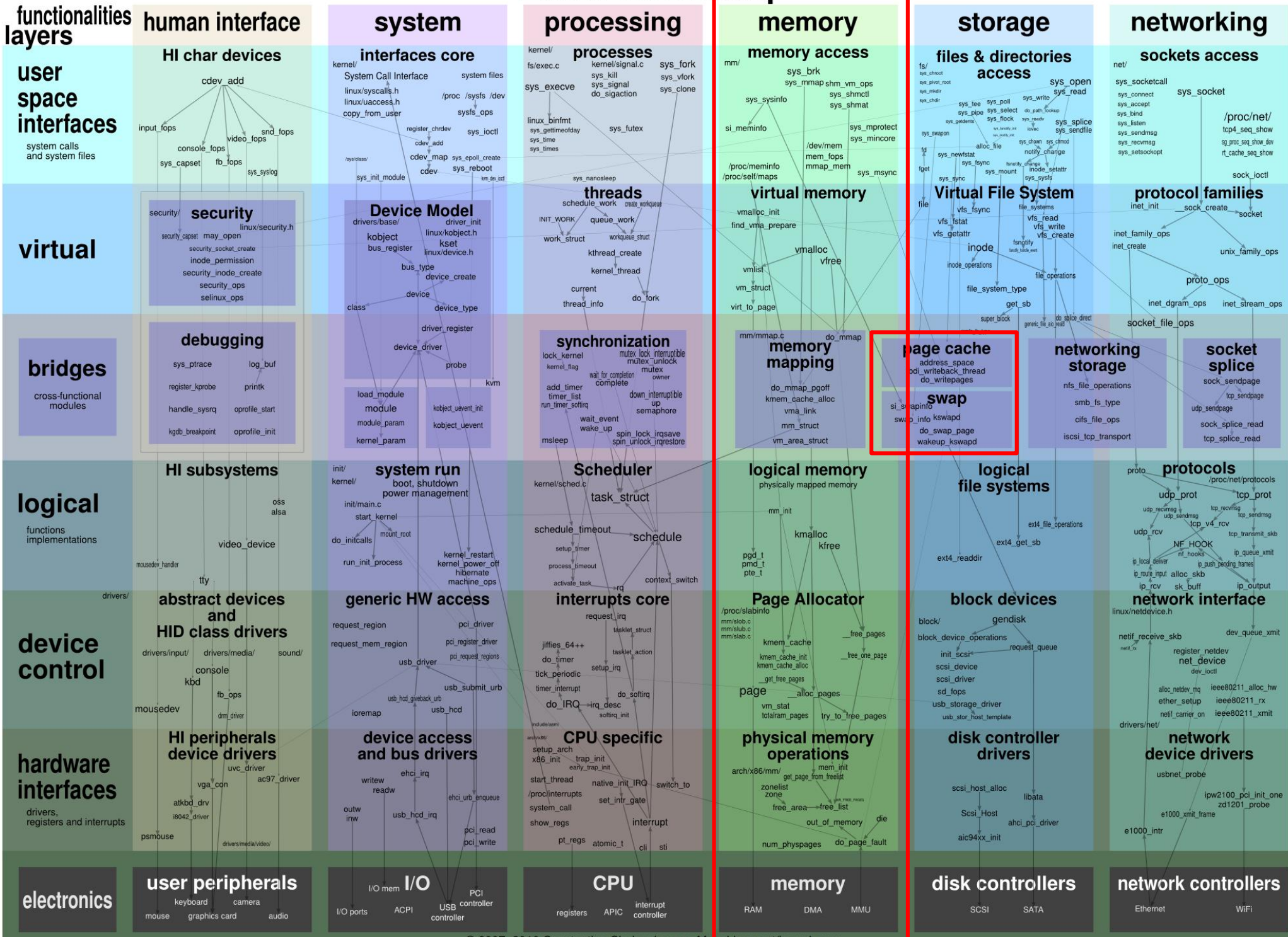
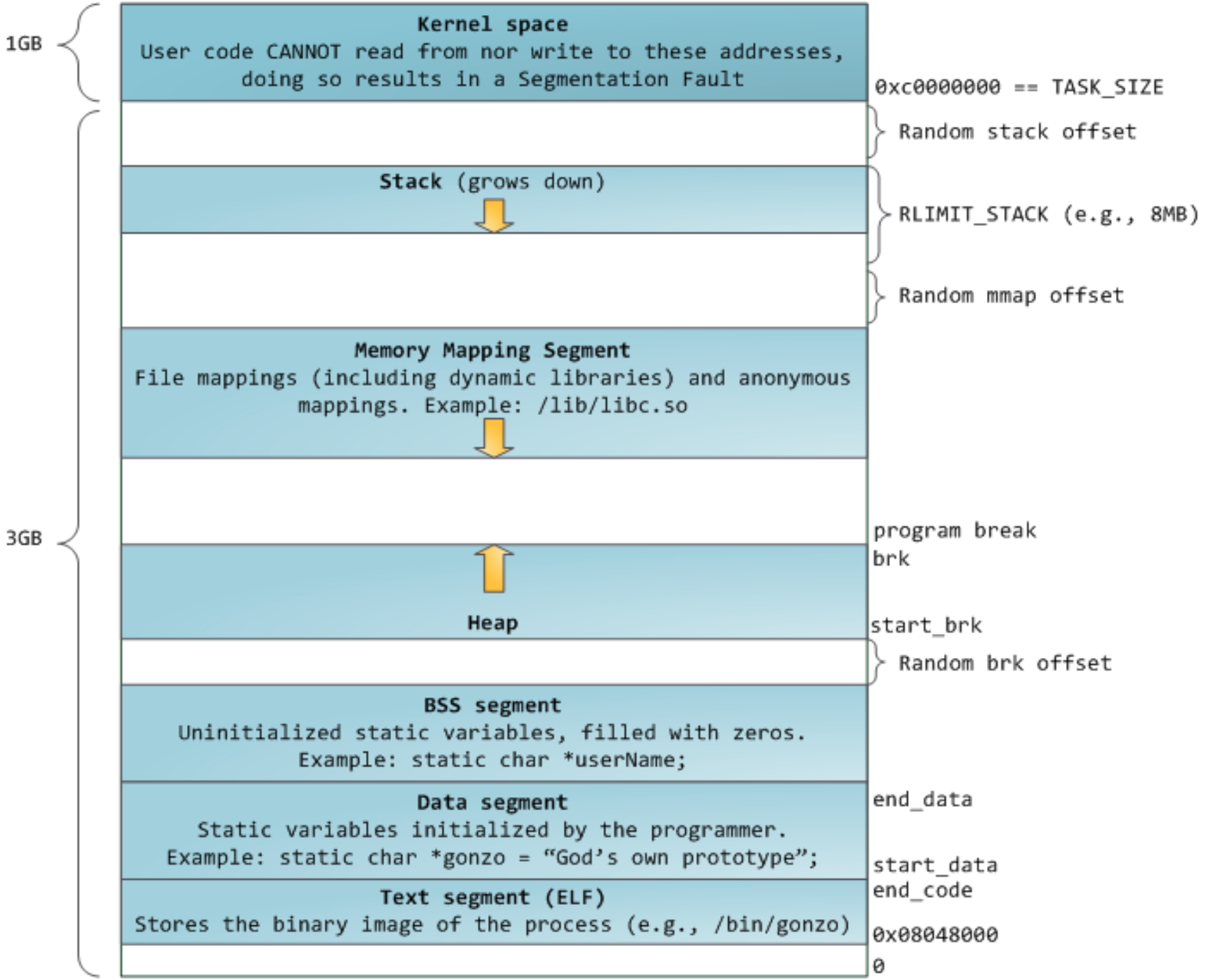
Linux kernel map

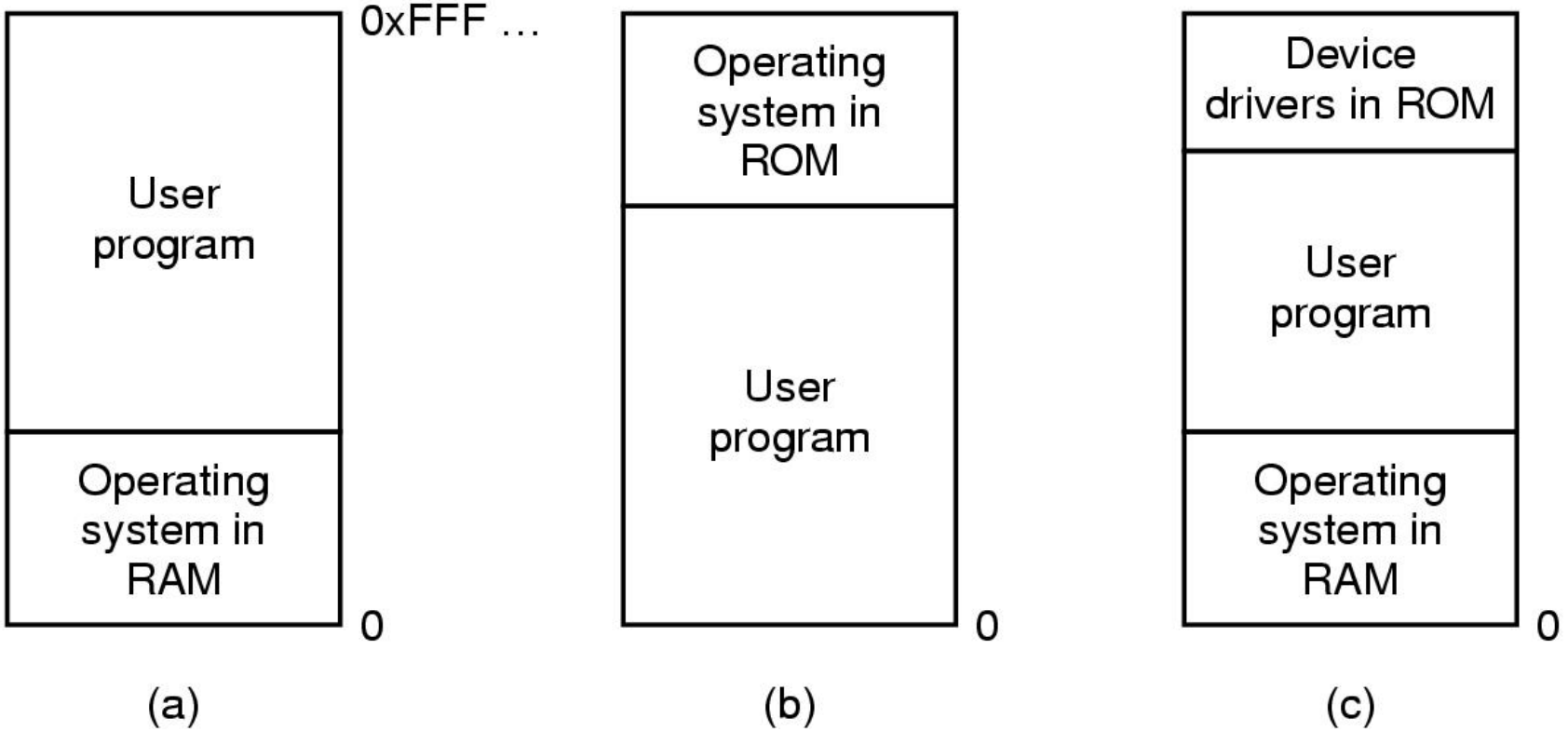


Why is Memory Management Necessary?

- What we want:
  - Large, fast, non volatile, inexpensive memory
- What we have: Memory hierarchy, i.e.
  - Few megabytes: very fast, expensive, volatile cache memory
  - Few gigabytes: fast, volatile main memory
  - Few terabytes: slow, cheap, non-volatile disk storage
- Automate Memory management to handle memory allocation and movement of data across hierarchy


1GB
3GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
end_data
start_data
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
end_code
0x08048000
0

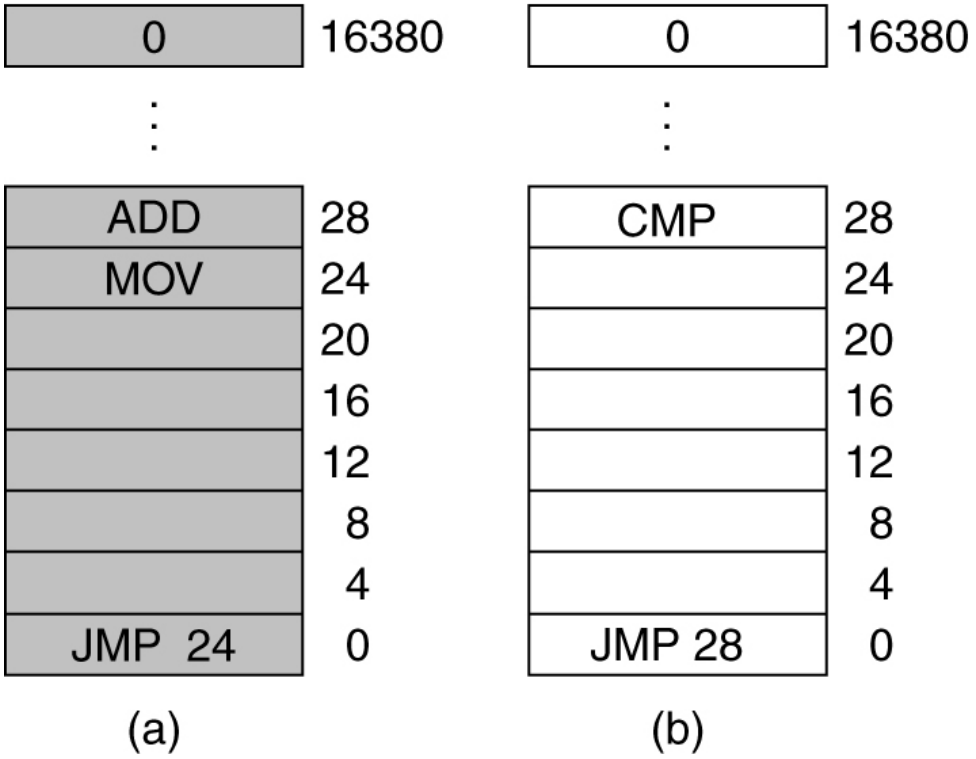
No Memory Abstraction



- Single-process, w/out swapping or paging
  - Early computing (60's – 80's)
  - OS with one user process
  - See e-Cos for example

IBM 360 Had the Right Idea

- Two-process example
- Memory keys mean that the two cannot damage one another
- What about those JMPs?

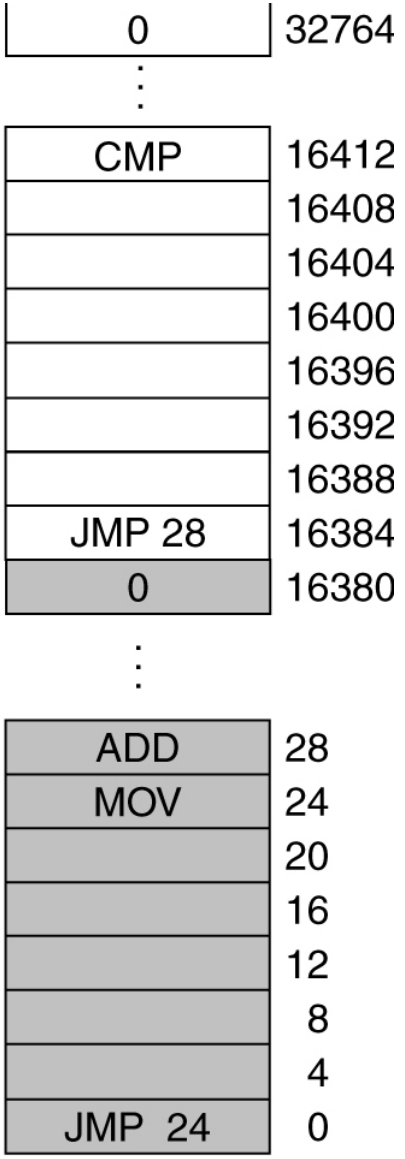


Let's See...

- But, cannot have both reference absolute physical memory
- Use static relocation
  - If program is loaded at 16,380,

    add 16,380 to all constant addresses
  - relocation costs CPU cycles
    - have to know difference between

      JMP 28 and MOV REGISTER1, 28

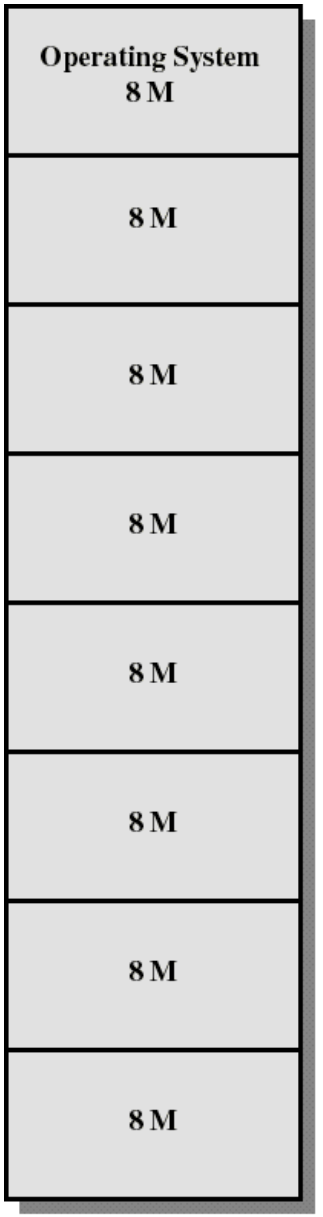


IBM 360 Highlights Some Issues with Main Memory Sharing

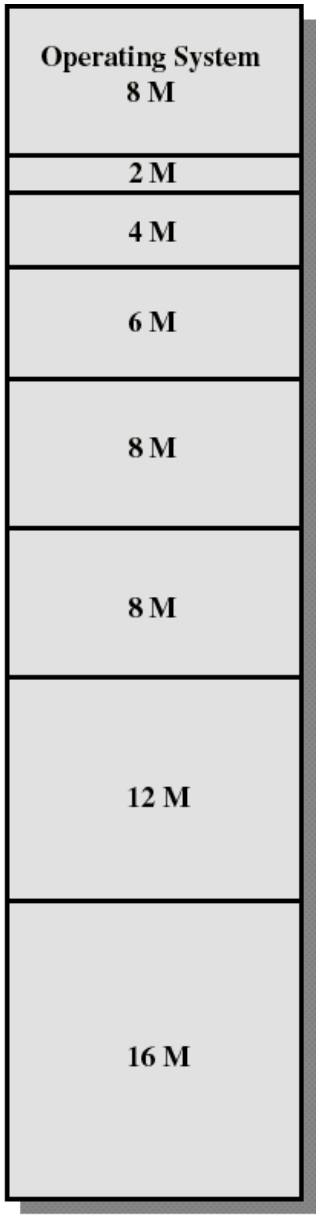
- Coexisting processes:
  - Need transparency
    - No process should know that memory is shared
    - Run regardless of the number/location of processes
  - Need safety
    - Processes should not corrupt each other
  - Need efficiency
    - CPU and memory performance should not be degraded by sharing

One Approach: Fixed Partitioning

- Partition main memory
  - Non-overlapping regions
  - Fixed partitions
    - Equal or unequal sizes
  - Leftover space = internal frag.



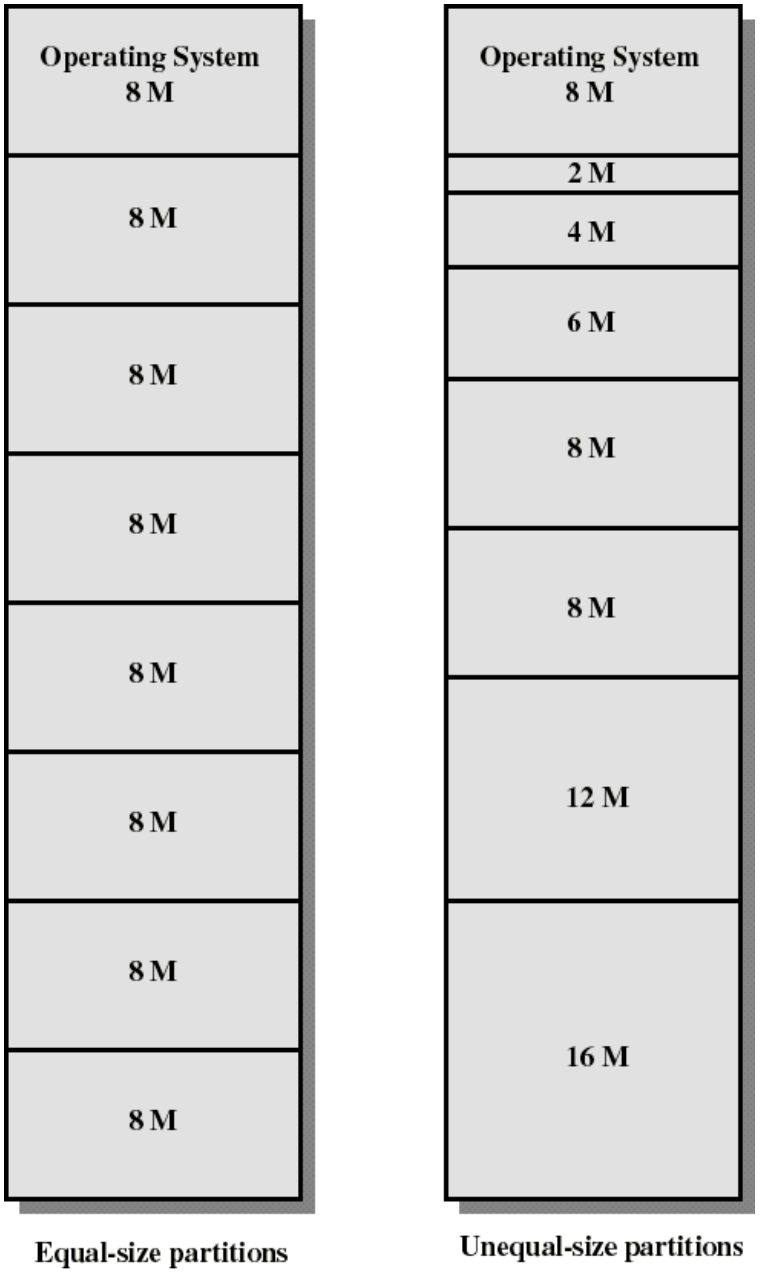
Equal-size partitions



Unequal-size partitions

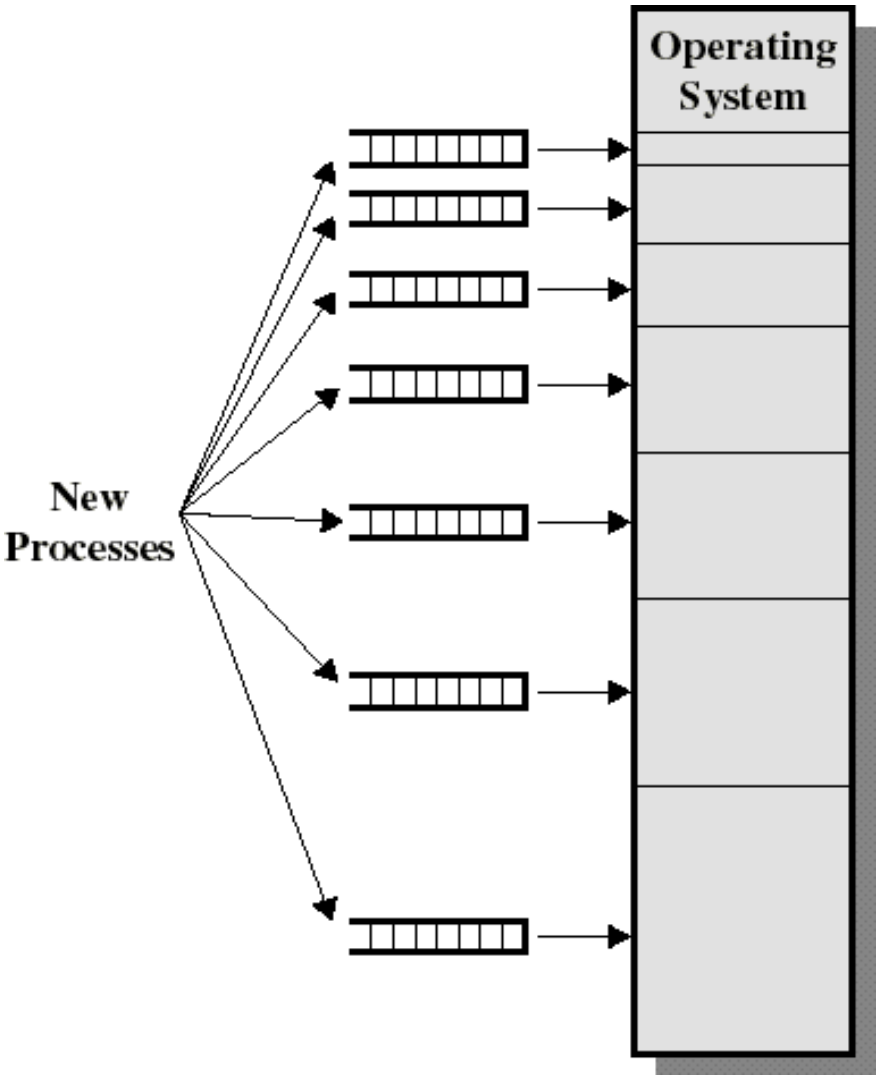
Partition Sizes

- Processes in equal-size partitions can lead to lots of wasted space
  - Internal fragmentation
- Minimize internal frag.
  - Make partitions unequal
  - Assign to smallest partition that fits the process



Unequal-Size Partitions

- Depending on common processes:
  - Some queues may be empty
  - Others may be heavily used



Chapter 3 – Memory Management

No Memory Abstraction

Address Spaces

Virtual Memory

Page Replacement Algorithms

Design Issues for Paging Systems

Implementation Issues

Segmentation

What is an Address Space

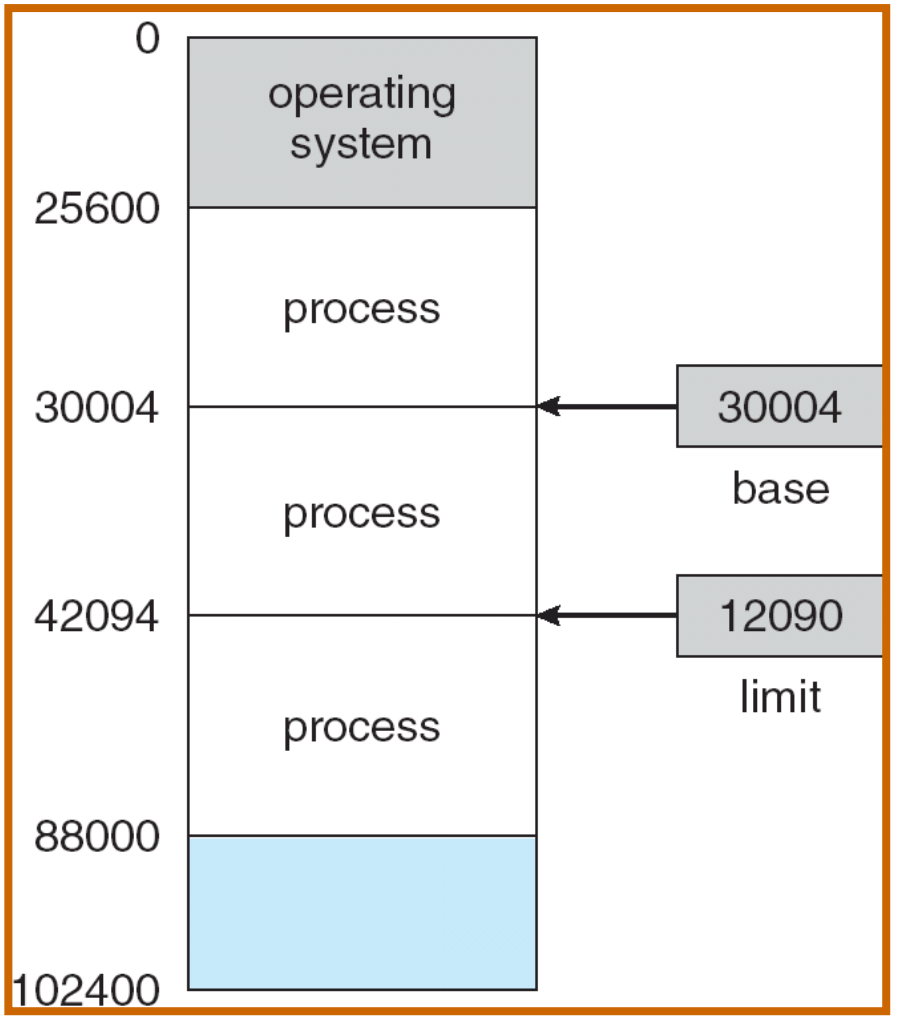
- Set of addresses that a process can use to address memory?
- Each process has OWN address space
  - independent from other processes

Relocation and Protection

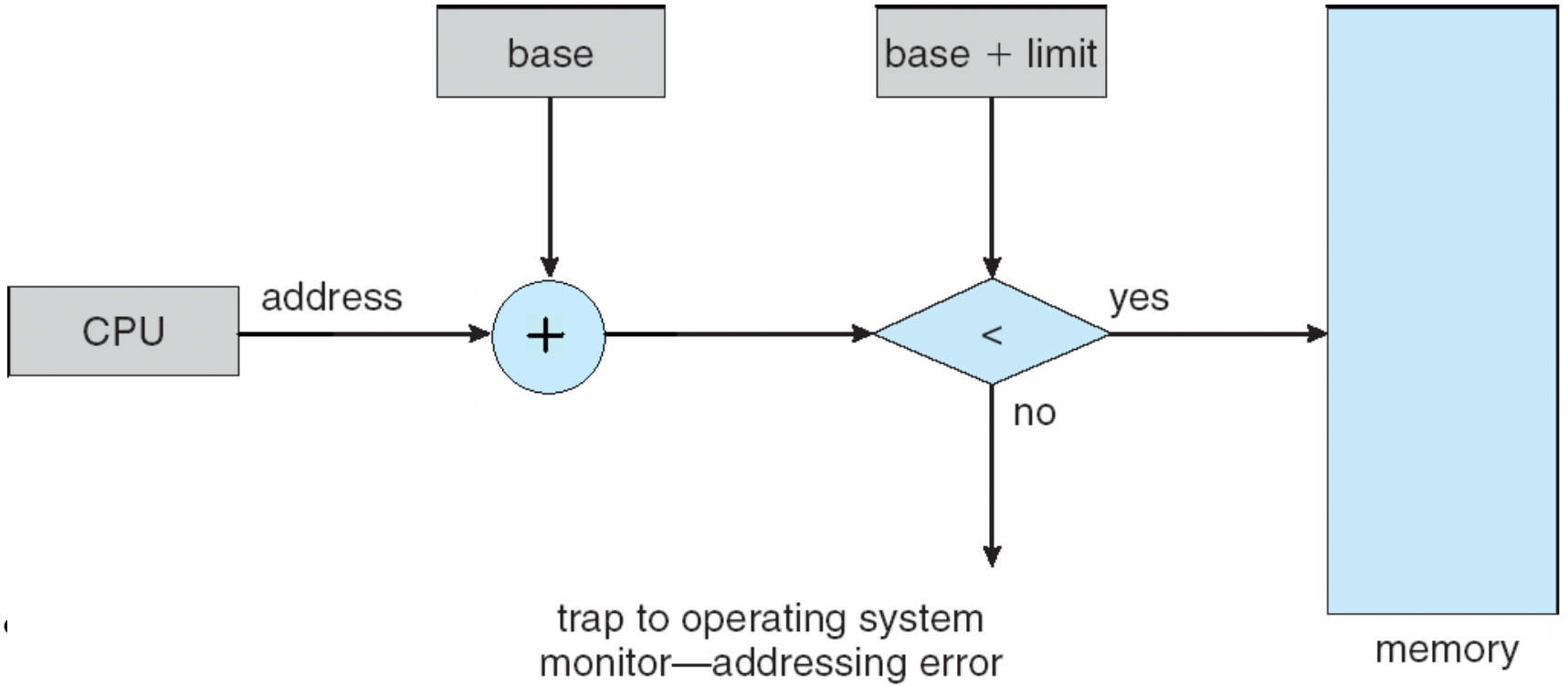
- Don't know where program will be loaded
  - Addresses cannot be absolute
  - Keep program out of others' partitions
- Base and limit values
  - Address locations added to base
  - Address locations bigger than limit are errors
- Can move process between segments

Example: Base and Limit

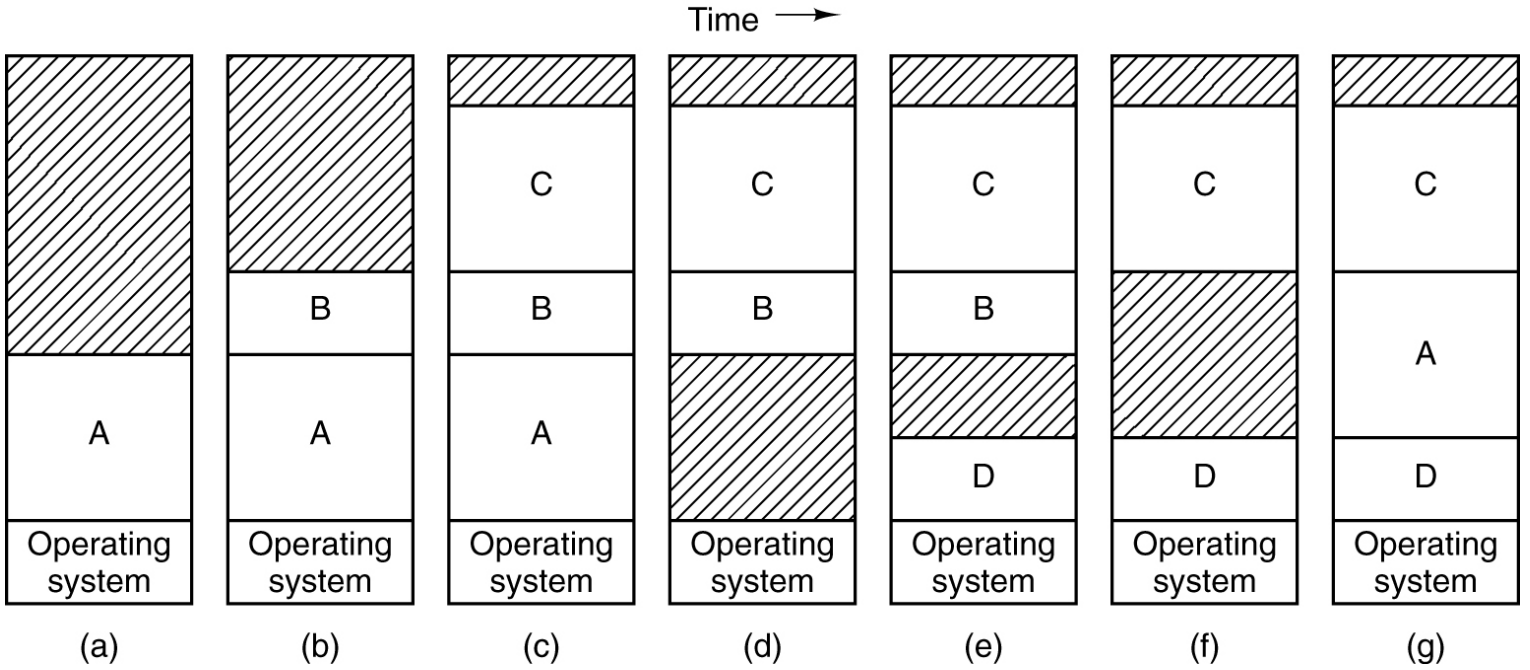
- Base and Limit define the address space



Base and Limit



Contiguous Allocation



- Multiple-partition allocation
  - Hole: block of available memory, somewhere
  - New process placed in large-enough hole
  - OS keeps track of allocated partitions and holes
  - Fragmentation can be managed by compaction

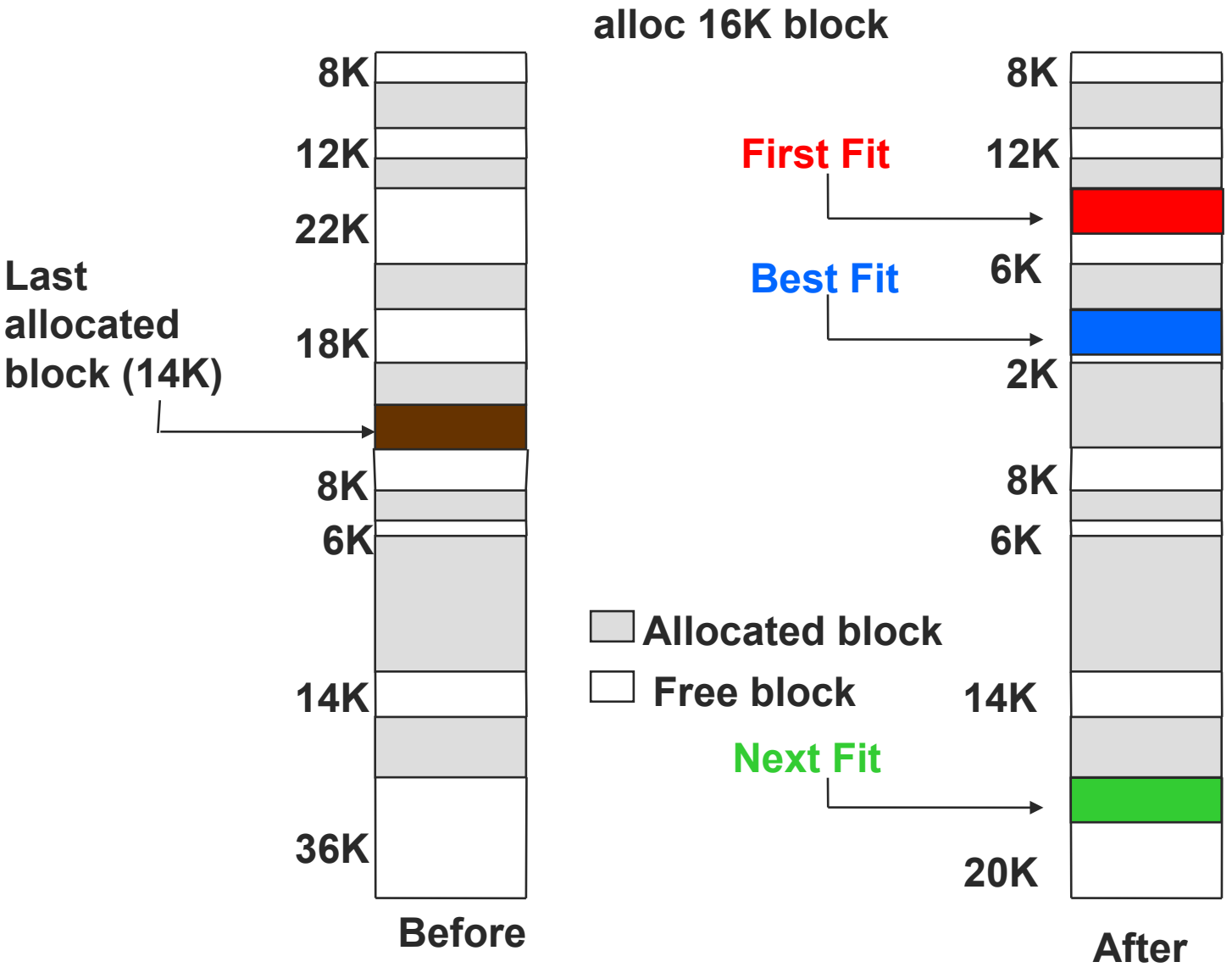
Placement Algorithms

- First-fit
  - Scan list of free memory from beginning
  - Choose first available block that is large enough
  - Fast
- Next-fit
  - Same as first fit, but next scan starts where last one made the placement
  - More often allocates a block of memory from end of memory, where largest free block is
  - Compaction required to obtain large free block at end of memory

Placement Algorithms(2)

- Best-fit
  - Searches entire list, from beginning to end
  - Chooses smallest hole that is adequate
  - Slow
  - Results in most fragmentation, tiny useless holes everywhere
- Worst-fit?
  - Take largest available hole
  - Leaves large new hole, on average

Example


alloc 16K block
8K
12K
22K
18K
8K
6K
14K
36K
Last allocated block (14K)
Before
First Fit
Best Fit
Next Fit
Allocated block
Free block
8K
12K
6K
2K
8K
6K
14K
20K
After

Fragmentation Summary

- External fragmentation
  - Total memory space exists to satisfy an allocation request, but it is not contiguous
- Internal fragmentation
  - Allocated memory is larger than requested memory
  - Size difference is internal to partition
    - This memory is wasted for the life of the process
- Compaction reduces external frag.
  - Move memory contents to create large free blocks
  - Possible only with dynamic relocation, done at execution time

Chapter 3 – Memory Management

No Memory Abstraction

Address Spaces

Virtual Memory

Page Replacement Algorithms

Design Issues for Paging Systems

Implementation Issues

Segmentation

Before Virtual Memory

- Review:
  - Base and limit registers
    - Basic abstraction of address spaces
- What about running programs too large to fit into physical memory?
  - Or multiprogramming where, collectively, the programs don't fit?

Virtual Memory

- Abstraction of physical memory
- Break up each program into pages (fixed size chunks)
- Each page is a contiguous range of memory
- Not all pages have to be present in memory to run the program
  - OS manages pages resident in physical memory
  - Process may be blocked while waiting for nonresident pages to be moved into memory
- As far as program is concerned, it is operating in its own contiguous address space

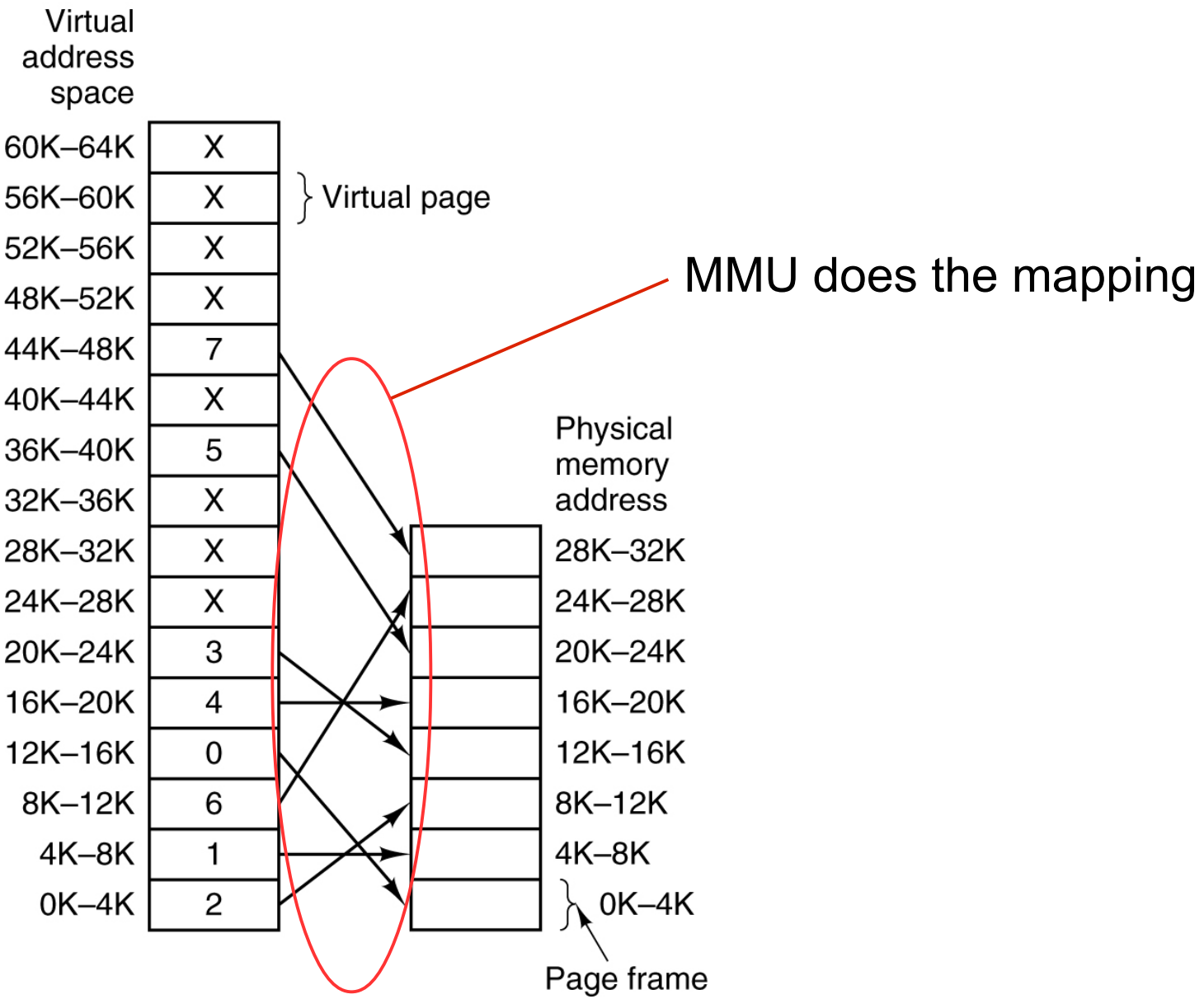
Paging

- Program-generated addresses are virtual addresses
  - Program address space = virtual address space
- Virtual addresses mapped to physical addresses by MMU
  - MMU = memory management unit
- Virtual address space divided into fixed-size units (these are pages)
  - When they are in physical memory, they are page frames

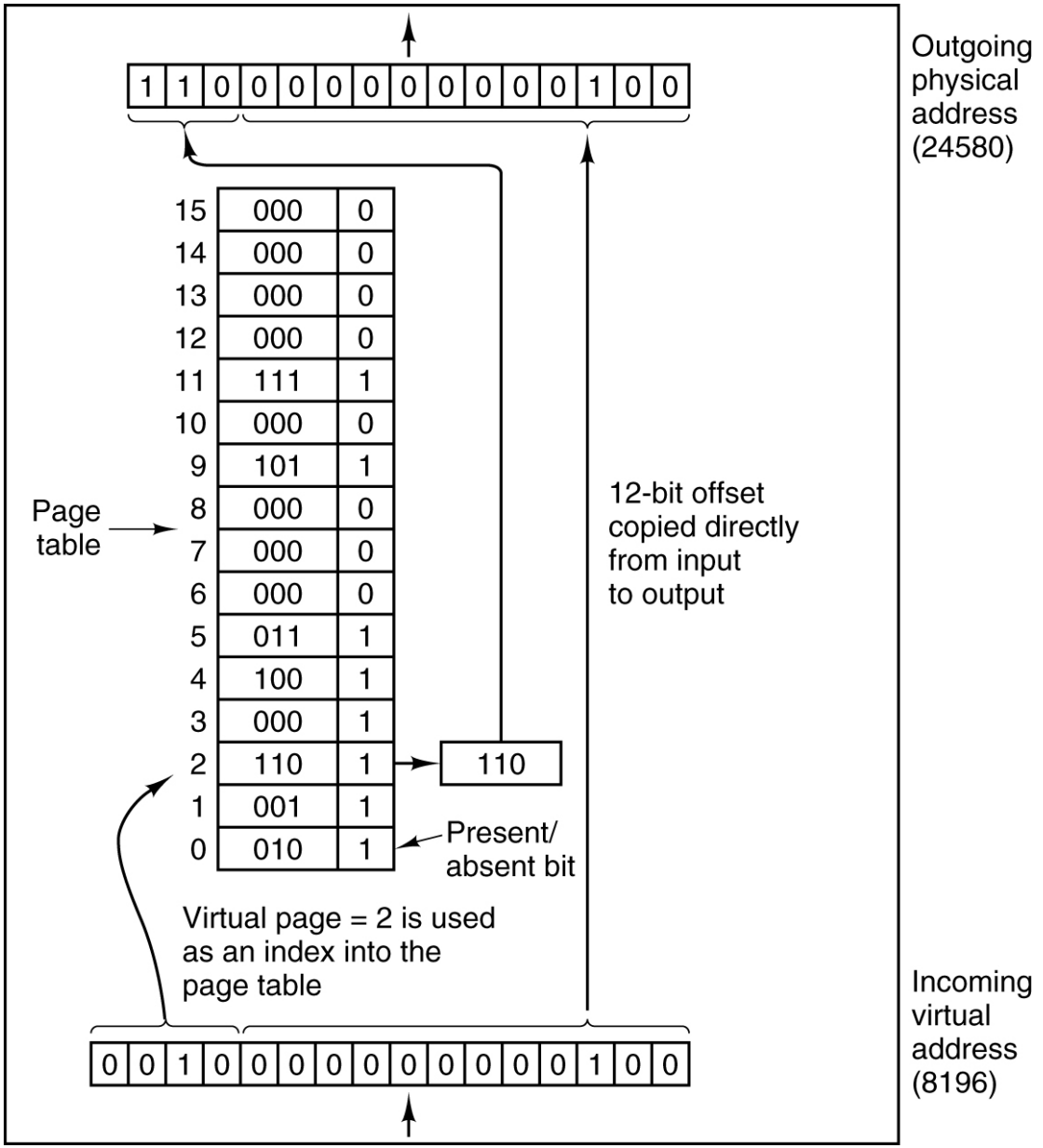
Paging(2)

- Present/absent bit
  - Keeps track of pages that are not currently mapped to physical memory
- Page fault
  - Program calls for address located in a page that is not currently mapped
  - Page must be brought into physical memory

Virtual ^ Memory Example

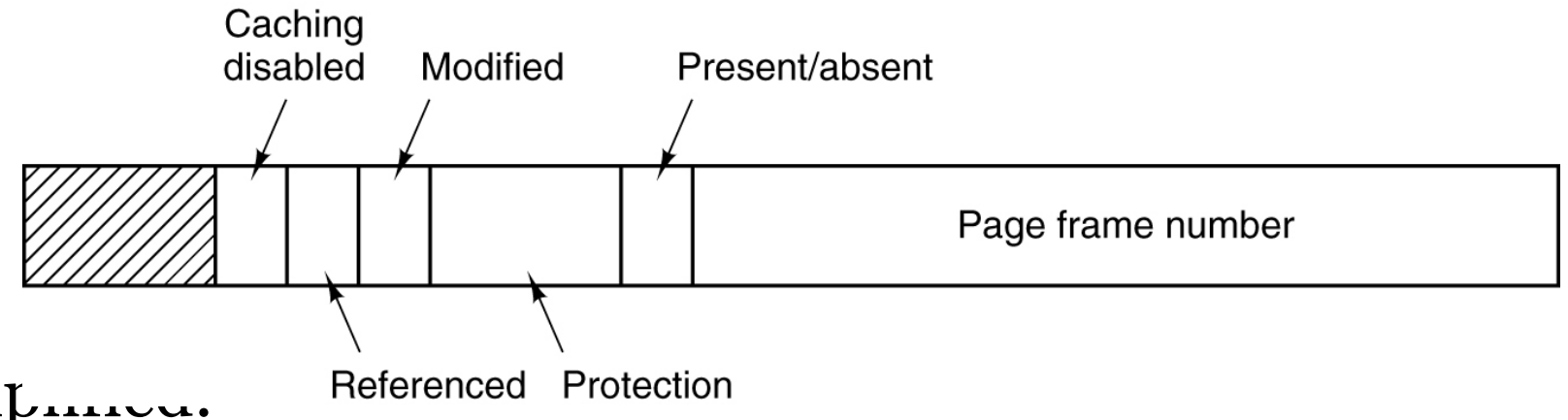


MMU Example

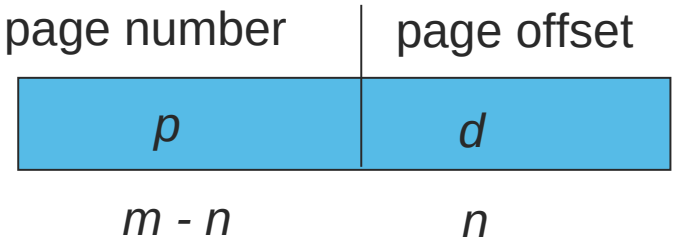

1 1 0 0 0 0 0 0 0 0 0 0 1 0 0
Outgoing physical address (24580)
15 000 0
14 000 0
13 000 0
12 000 0
11 111 1
10 000 0
9 101 1
8 000 0
7 000 0
6 000 0
5 011 1
4 100 1
3 000 1
2 110 1
1 001 1
0 010 1
Page table
110
Present/ absent bit
12-bit offset copied directly from input to output
Virtual page = 2 is used as an index into the page table
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
Incoming virtual address (8196)

Address Translation

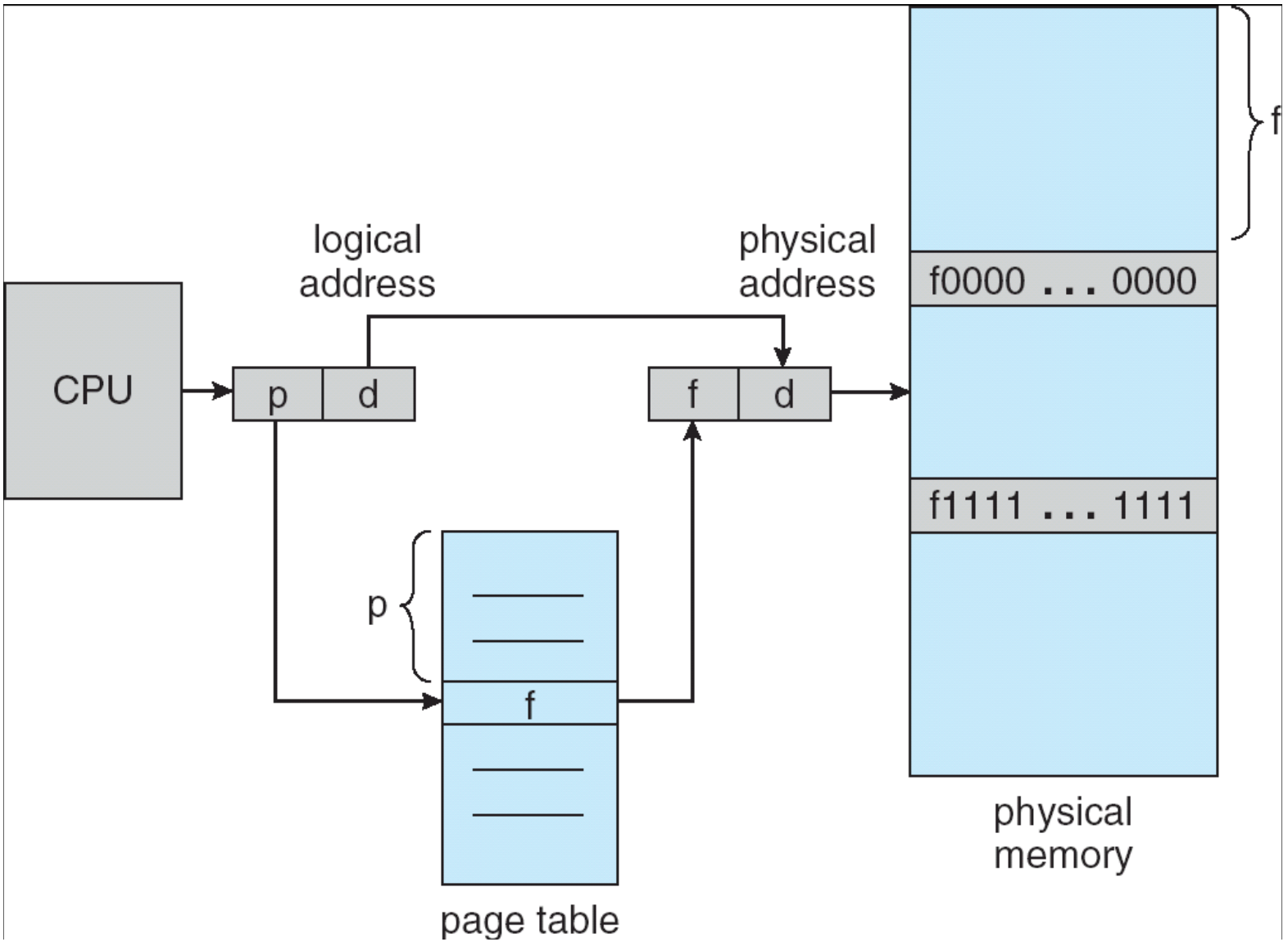
- Virtual address split into:
  - Page number
  - Offset within page
- Typical page table entry structure:



- Simplified:



Address Translation Example

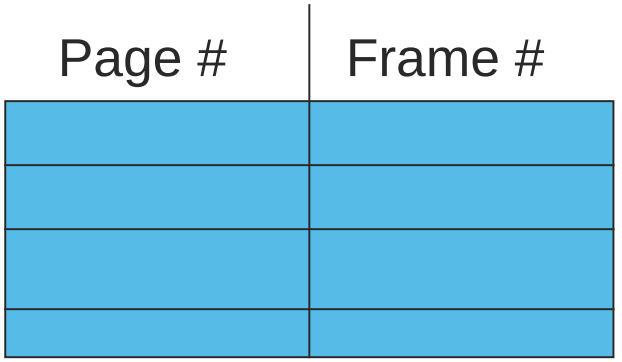


Major Issues

- Virtual ^ physical address mapping must be fast
  - The mapping is done for every memory reference
  - May be one, two, or more times per instruction
- Large virtual address space = large page table

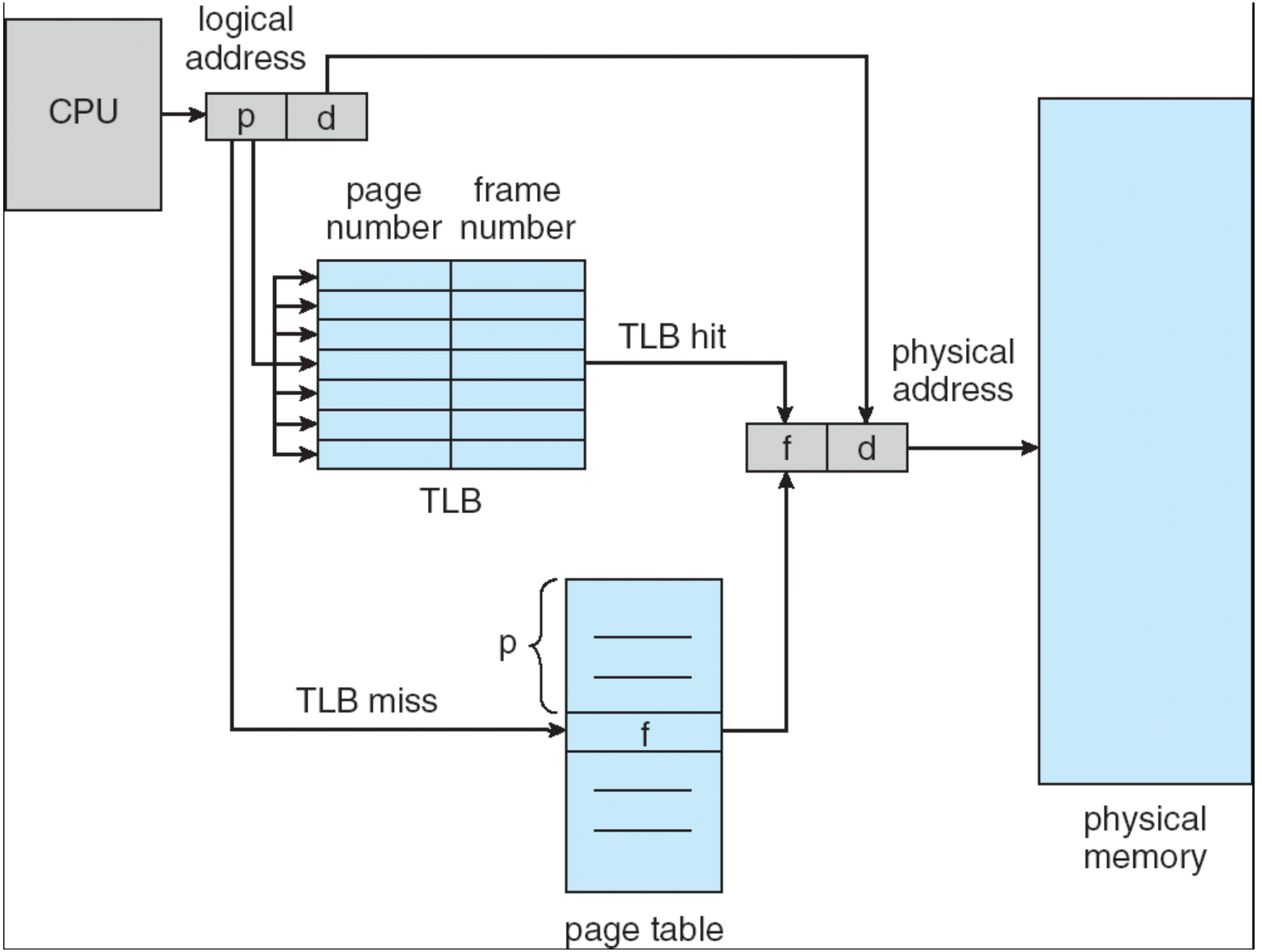
First Issue: Speeding Up Paging

- Fast-lookup associative cache: TLB
  - Translation Lookaside Buffer



  - If page number is in the TLB, get frame number
  - Else, get frame number from main memory
    - Also, update TLB

Paging with TLB example



More Details

- What is in a TLB?

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Hardware TLB Management

- On TLB miss:
  - MMU parses page table, loads PTE into TLB
  - If page invalid (not in memory), generate fault
    - OS performs fault handling
    - Restart instruction that faulted
- On TLB hit:
  - MMU checks valid bit
  - If valid, perform address translation
  - Else, generate fault (see above)

Software TLB Management

- On TLB miss:
  - HW raises exception, trap to OS
  - OS parses page table, load PTE into TLB
    - Replace entry in TLB
    - Restart instruction that faulted
  - If page not valid, OS does page fault handling
  - OS returns from trap handler
- On TLB hit:
  - If valid, perform translation
  - Else, page swapped out to disk, MMU generates page fault, etc… (see above)

Hardware vs Software Management

- Hardware
  - Efficient
  - OS intervenes only when page fault occurs
  - Page structure dictated by hardware
- Software
  - MMU hardware is simple
  - TLB misses are expensive
  - OS designer can choose page structure
    - Important for 2-level page tables, inverted page tables, etc.

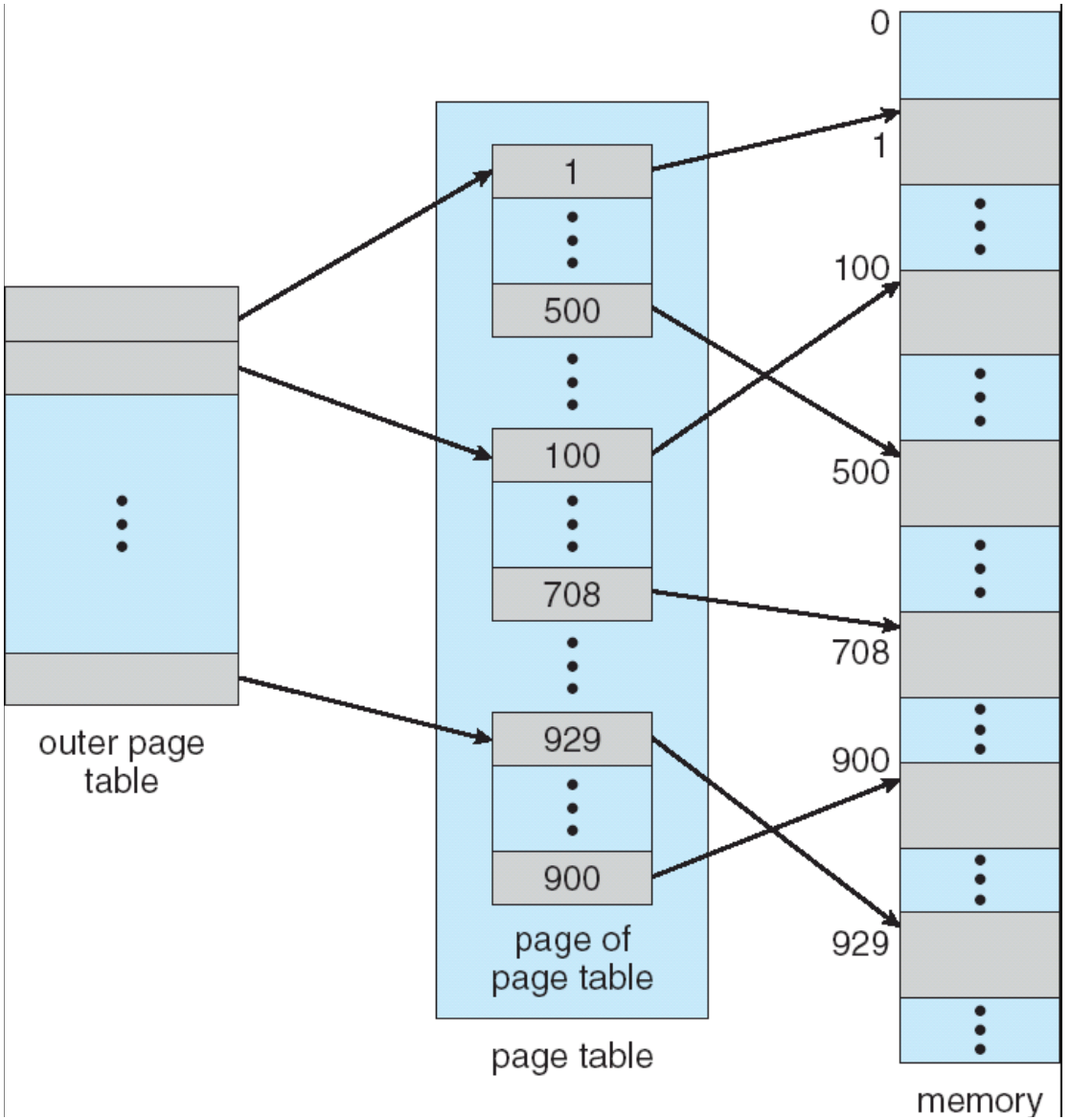
Second Issue: Page Table Storage

- Page tables are created per process
- If we have a 32 bit system… not a big deal
  - 2^32 addressable bytes (4GB)
  - 4KB page=2^12 bits per page, so need 12 bits for page offset
  - Remaining 20 bits used to index into page table
  - So, 4 bytes total per page table entry
  - 2^20 entries in page table per process
  - 4MB page table per process

What About Large Address Spaces?

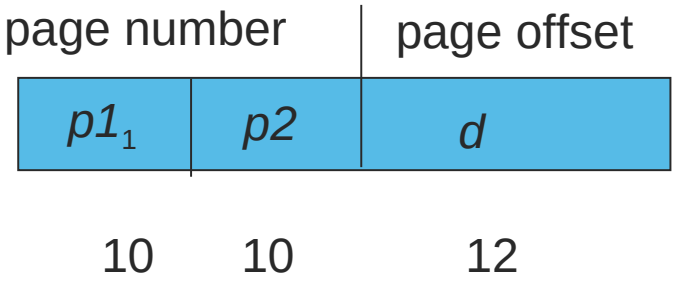
- Why are large address spaces a problem?
  - 64 bit computer = 2^64 addressable bytes
  - Assuming 4KB pages
    - 2^64/2^12 = 2^52 page table entries per process
  - 2^52 * 4 bytes… that's a lot of bytes per process
- Fortunately most entries are unused
- Solutions?
  - Hierarchical paging
  - Hashed page tables
  - Inverted page tables

Hierarchical: Two-Level Page Table

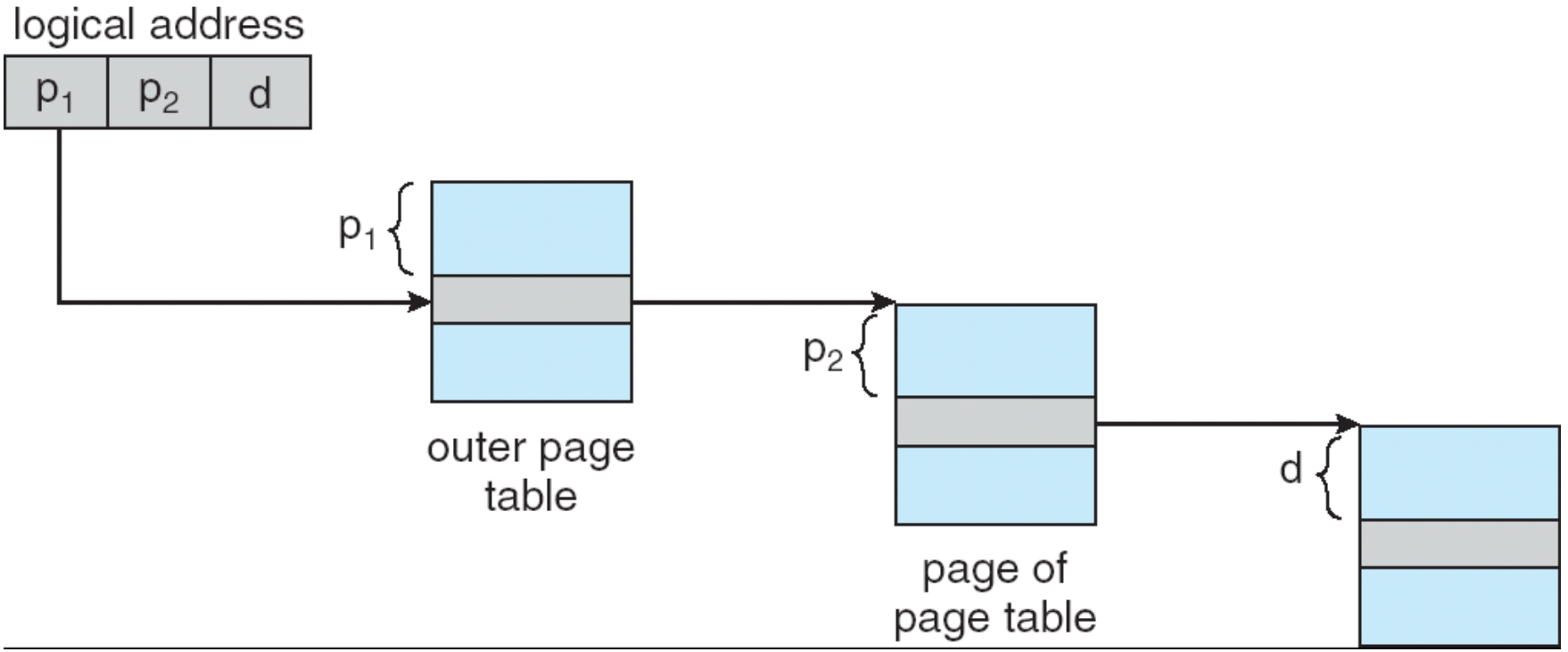


Two-Level Paging Example

- On a 32-bit system, divide logical address into:
  - d: page offset, 12 bits
  - p1: 10 bit offset into outer table
  - p2: 10 bit offset into inner table

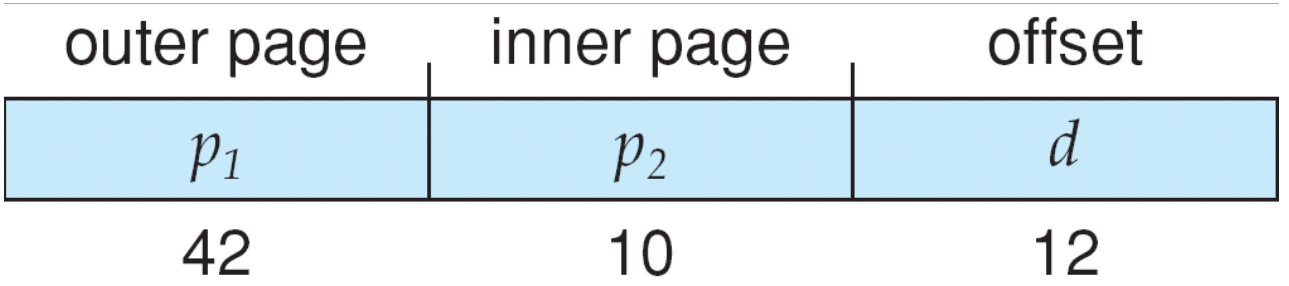


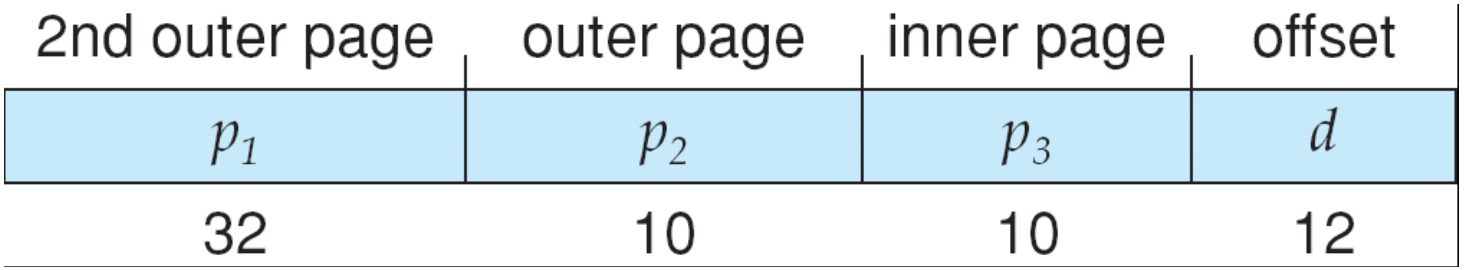
Two-Level Paging Example (2)



- Page table base register points to location of page table in memory
- Outer page table points to inner page table, which points to memory

Three-Level Paging

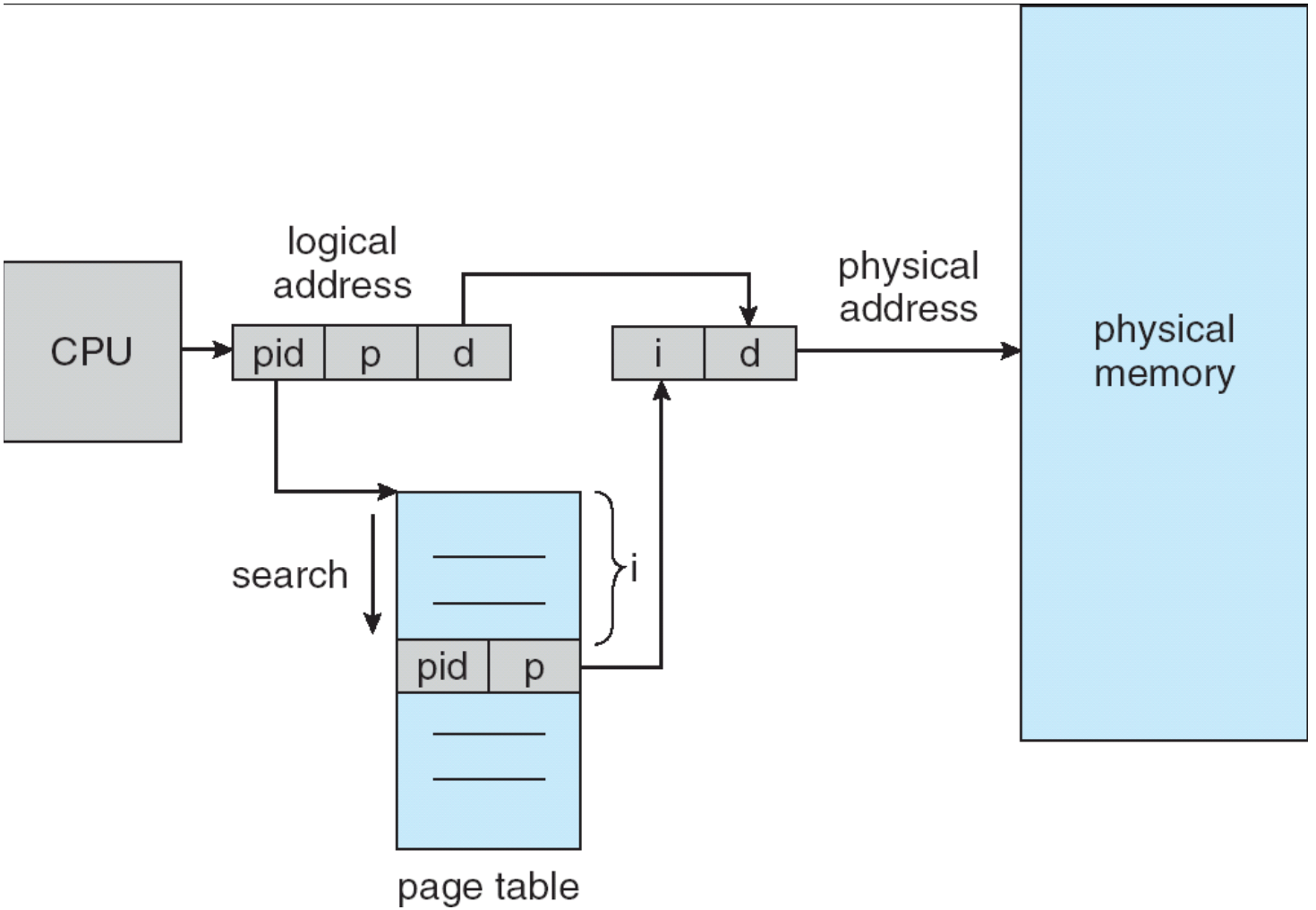




Inverted Page Table

- One entry for each page of physical memory
- Each entry contains information about the virtual address of the page stored in the real memory location, and process that owns the page
- Decreases memory needed for page table
  - But increases time required to search the table
- Use hash table to decrease search time
- (PowerPC, Itanium, UltraSPARC)

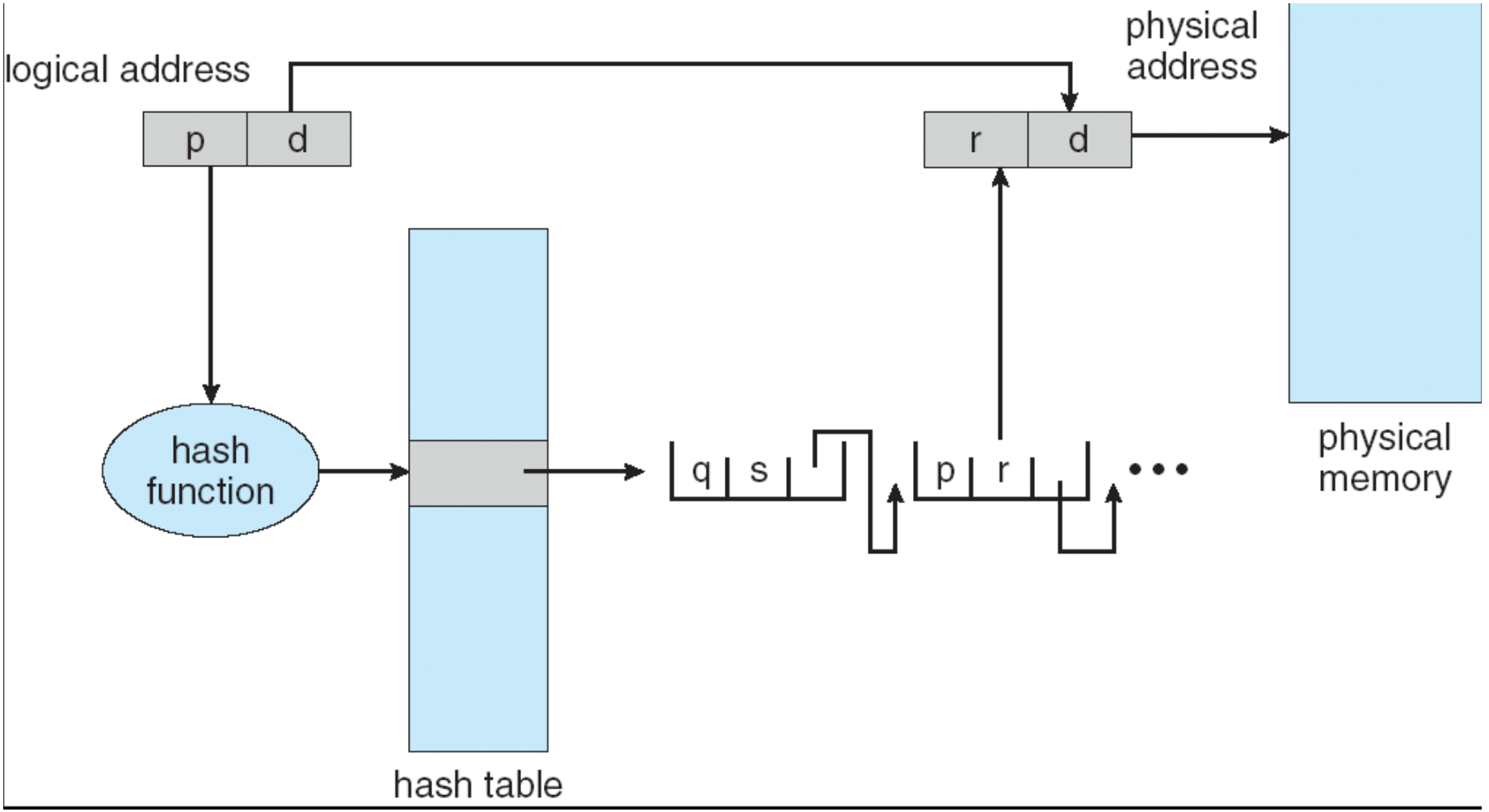
Inverted Page Table



Hashed Page Table

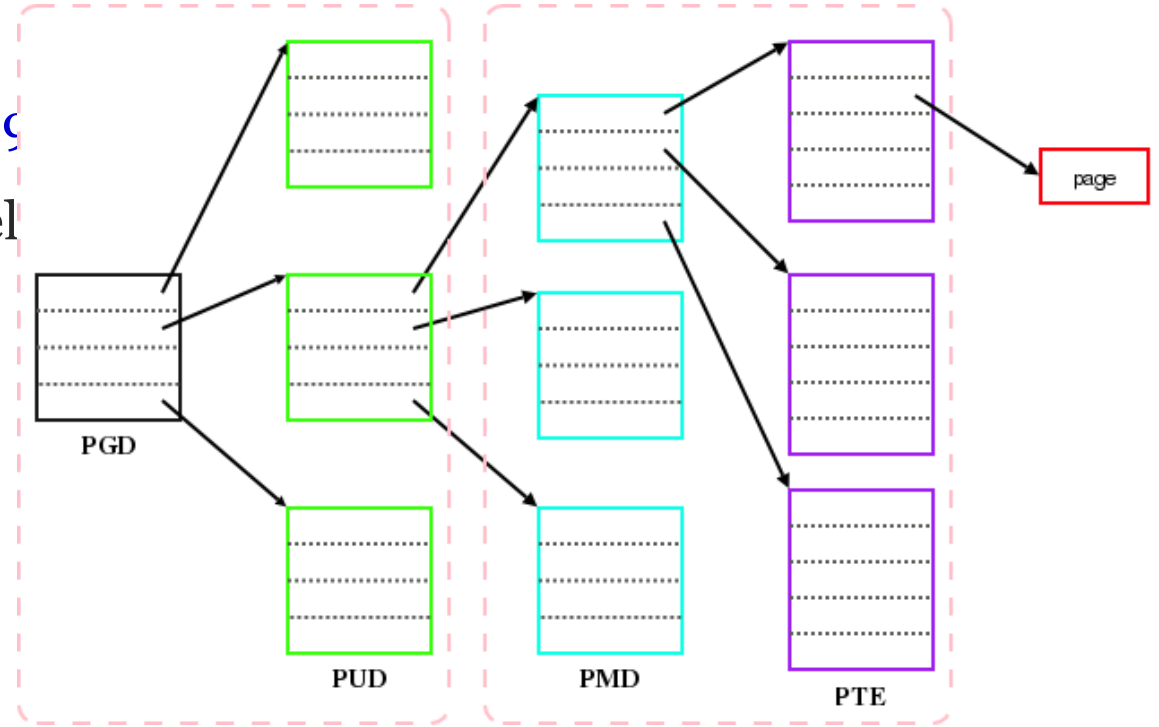
- Virtual page number hashed into a page table
- Page table contains a chain of elements hashing to same location
- Virtual page numbers are compared to contents of the chain, if a match is found, corresponding physical frame is extracted

Hashed Page Table



Real-World Example: x86-64 Linux

- http://lwn.net/Articles/117749
- 47 bit virtual address in kernel
- 4 level tables
- 46 bit physical memory



- PGD – Page Directory
- PUD – Page Upper Directory
- PMD – Page Mid-level Directory
- PTE – Page Table Entries

Architecture	Bits used			
	PGD	PUD	PMD	PTE
i386	22-31			12-21
x86-64	39-46	30-38	21-29	12-20

Chapter 3 – Memory Management

No Memory Abstraction

Address Spaces

Virtual Memory

Page Replacement Algorithms

Design Issues for Paging Systems

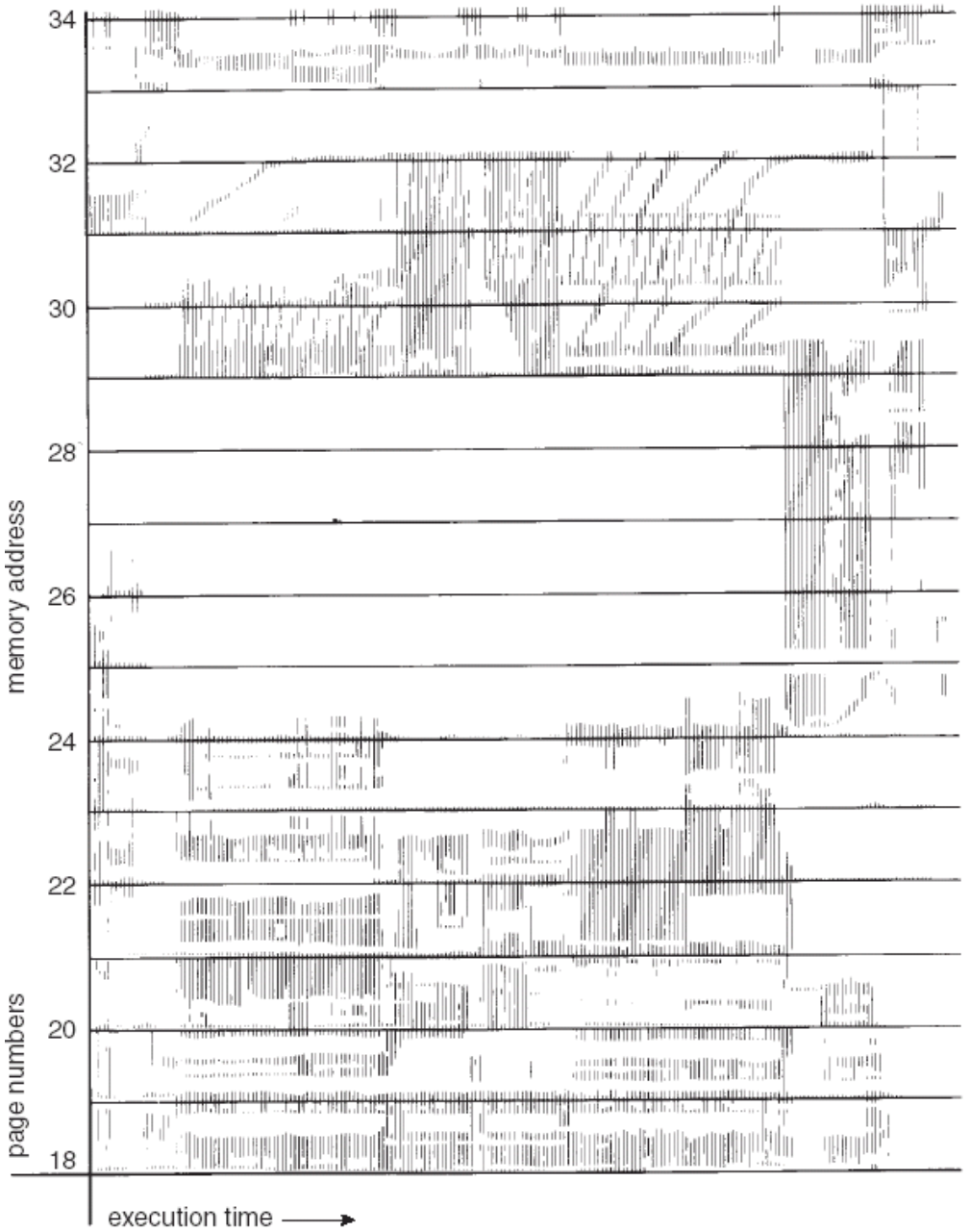
Implementation Issues

Segmentation

Demand Paging

- Bring page into memory only when needed
  - Lazy loading
  - Response to page fault
- What's the alternative?
  - Bring all pages into memory immediately

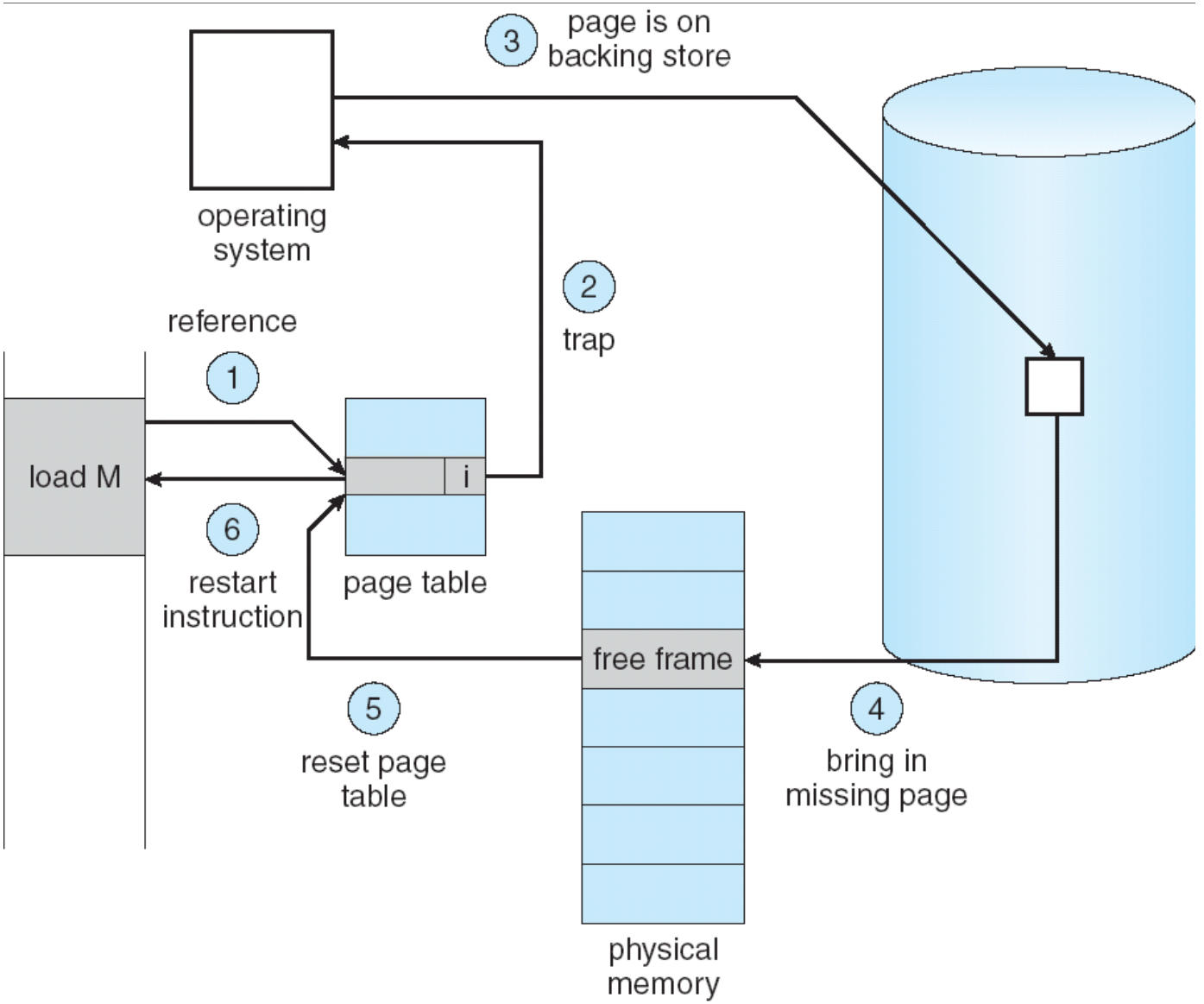
Only part of a program could be resident...



How does a page fault work?

- First reference to a page traps to the OS initiating page fault handling
- OS checks for valid reference
  - Abort if invalid
  - If just not in memory
    - Get empty frame
    - Swap page into frame
    - Set present bit
    - Restart instruction that caused fault

Illustrated Page Fault



Measuring Performance

- Page fault rate 0 < p < 1.0
  - 0 ^ No page faults
  - 1 ^ Every reference is a fault
- Effective Access Time (EAT):

  EAT= (1-p) * (memory access time)
  + p * (page fault overhead)
  - Page fault overhead = page swap in
  + page swap out
  + restart instruction

Example

- Memory access time = 100 nanoseconds
- Page fault overhead = 25 milliseconds
- Page fault rate = 1/1000

EAT=(1-p)*100 + p*25,000,000

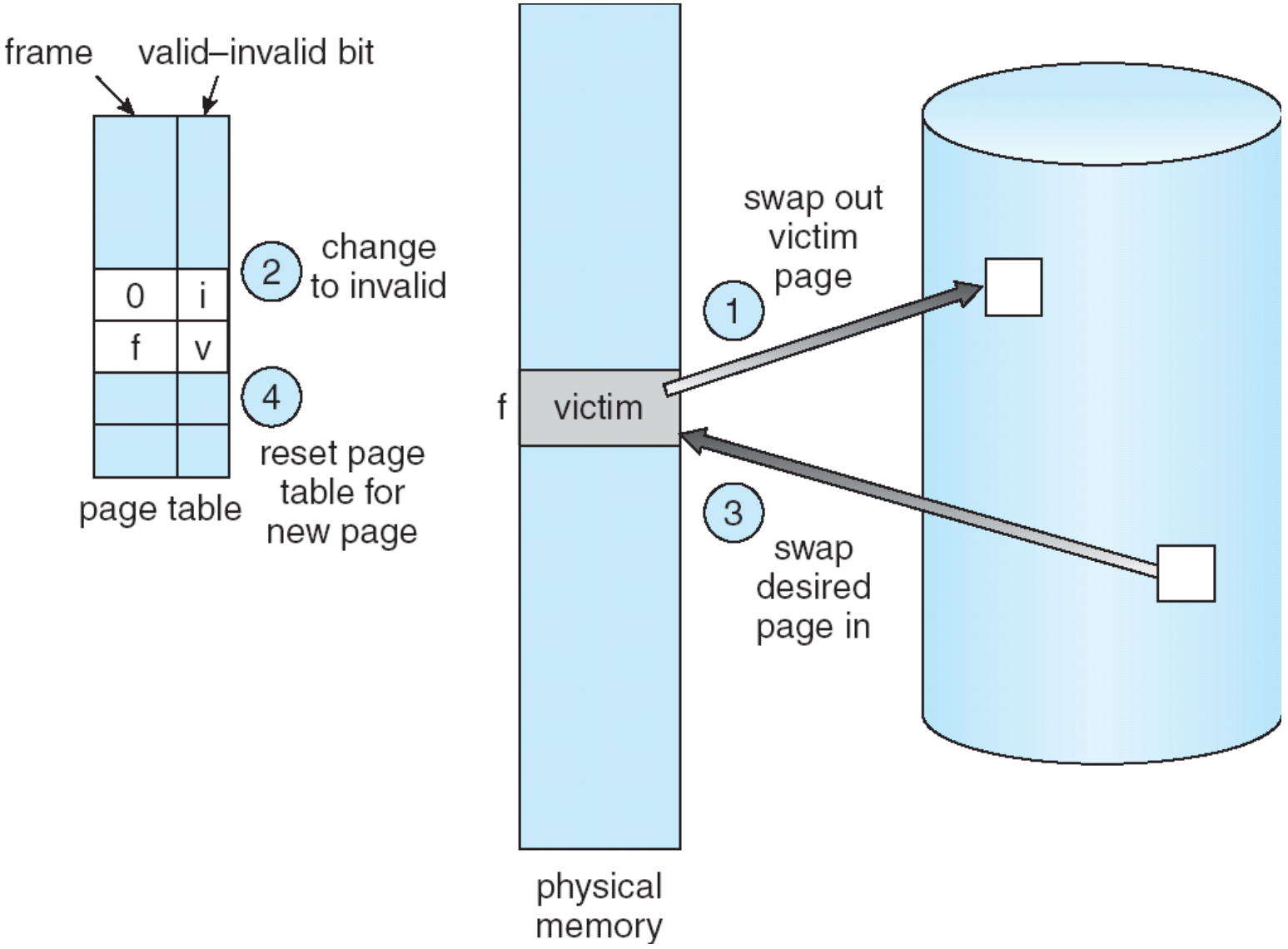
=100 + 24,999,900 * p

=100 + 24,999,900 * 1/1000 = 25 microsec.

Page Replacement

- What if we have a page fault but no free frames
  - Terminate the user process (not desirable)
  - Swap out process (reduces degree of multiprogramming, so likewise not desirable)
  - Replace some other page with the needed one
- Page replacement
  - If there exists a free frame, use it
  - Otherwise:
    - Find victim frame
    - Write page to disk, update tables, read in new page
    - Restart process

Page Replacement



Page Replacement Algorithms

- Want lowest page fault rate
  - Number of page faults over time
- Evaluate algorithm by running it with a particular string of references and computing the number of page faults
  - e.g.: 1,2,3,4,1,2,5,1,2,3,4,5
- Useful bits: referenced bit and modified bit

Optimal Replacement

- Not realizable
- Page with highest number of instructions until next reference is the next victim
  - No way of knowing this

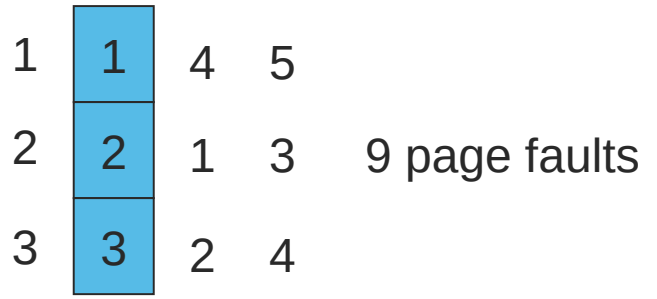
Not Recently Used (NRU)

- Use R and M bits (Referenced and Modified)
- At start of process execution
  - Set R and M bits to 0
- Periodically (at clock interrupt)
  - Set R bit to 0
- At page fault divide pages by category
  - Class 0: R=0, M=0 Class 2: R=1, M=0
  - Class 1: R=0, M=1 Class 3: R=1, M=1
- Remove random page from lowest numbered non-empty class

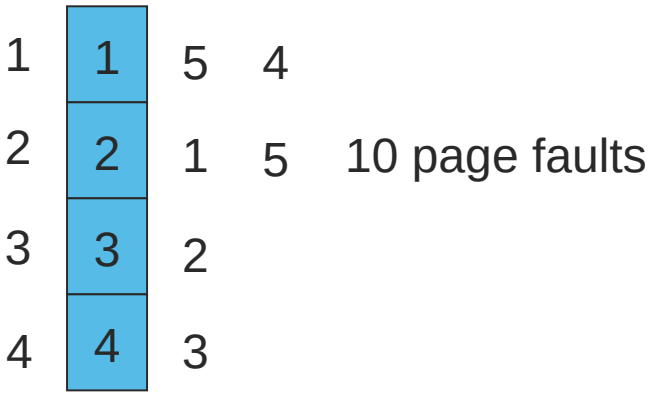
FIFO: First-In-First-Out

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

  3 frames per process

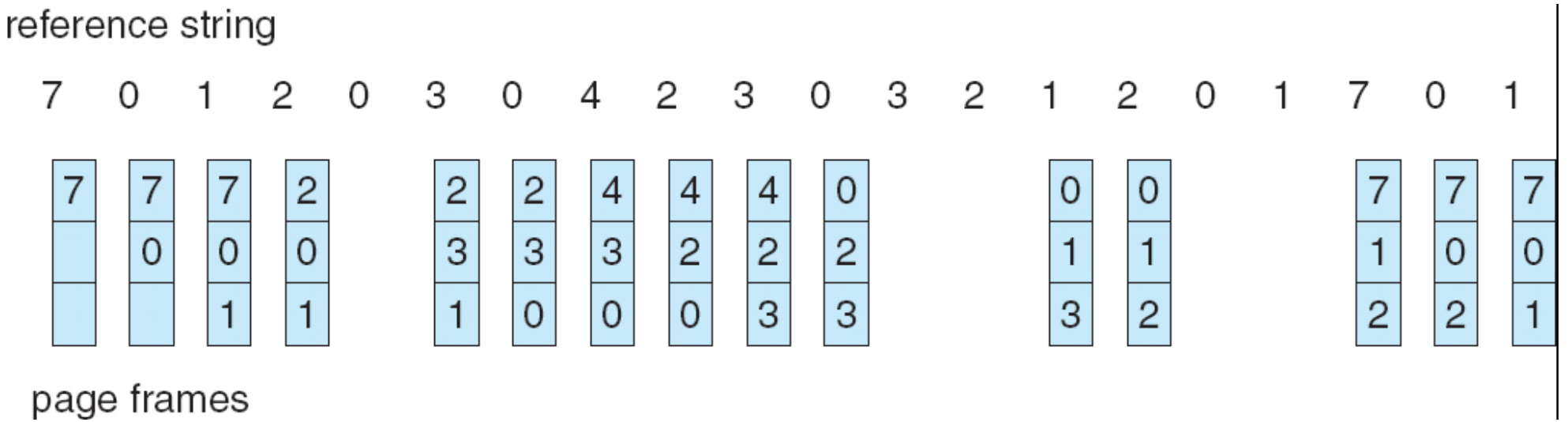


  4 frames per process



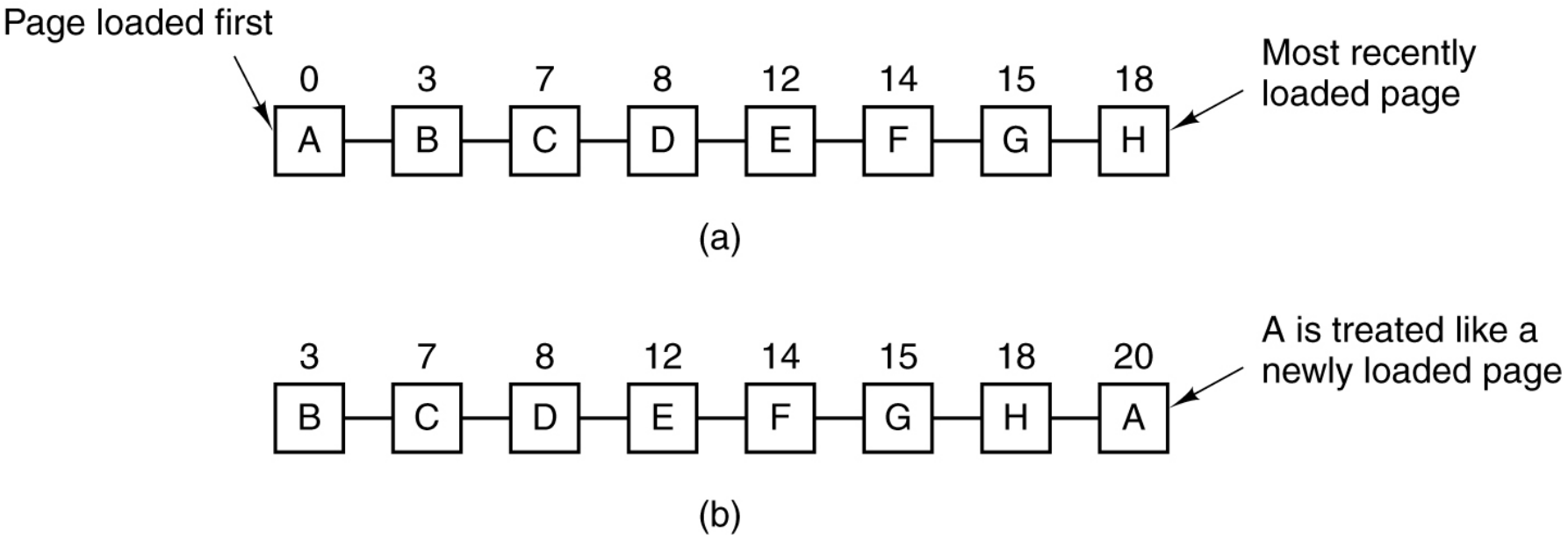
- This is known as Belady's Anomaly

FIFO: First-In-First-Out



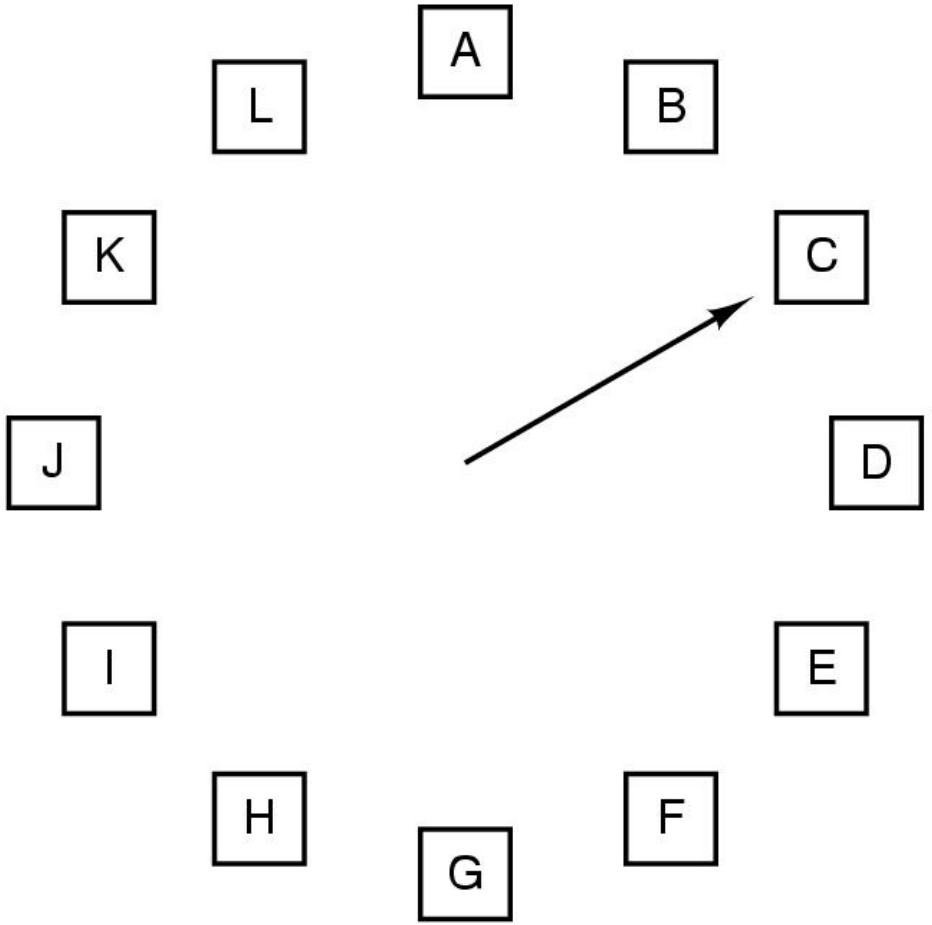
- Problem? Page in memory the longest may be frequently used
- Solution? Use a referenced bit.
  - If a page is marked as having been referenced, unmark and place at end of FIFO list.
  - This is known as Second Chance

Second Chance



- So, if a page is referenced enough, it is never replaced
- Can degenerate to FIFO
- Inefficient due to moving pages around in the list (solved by next algorithm)

Clock Replacement Algorithm

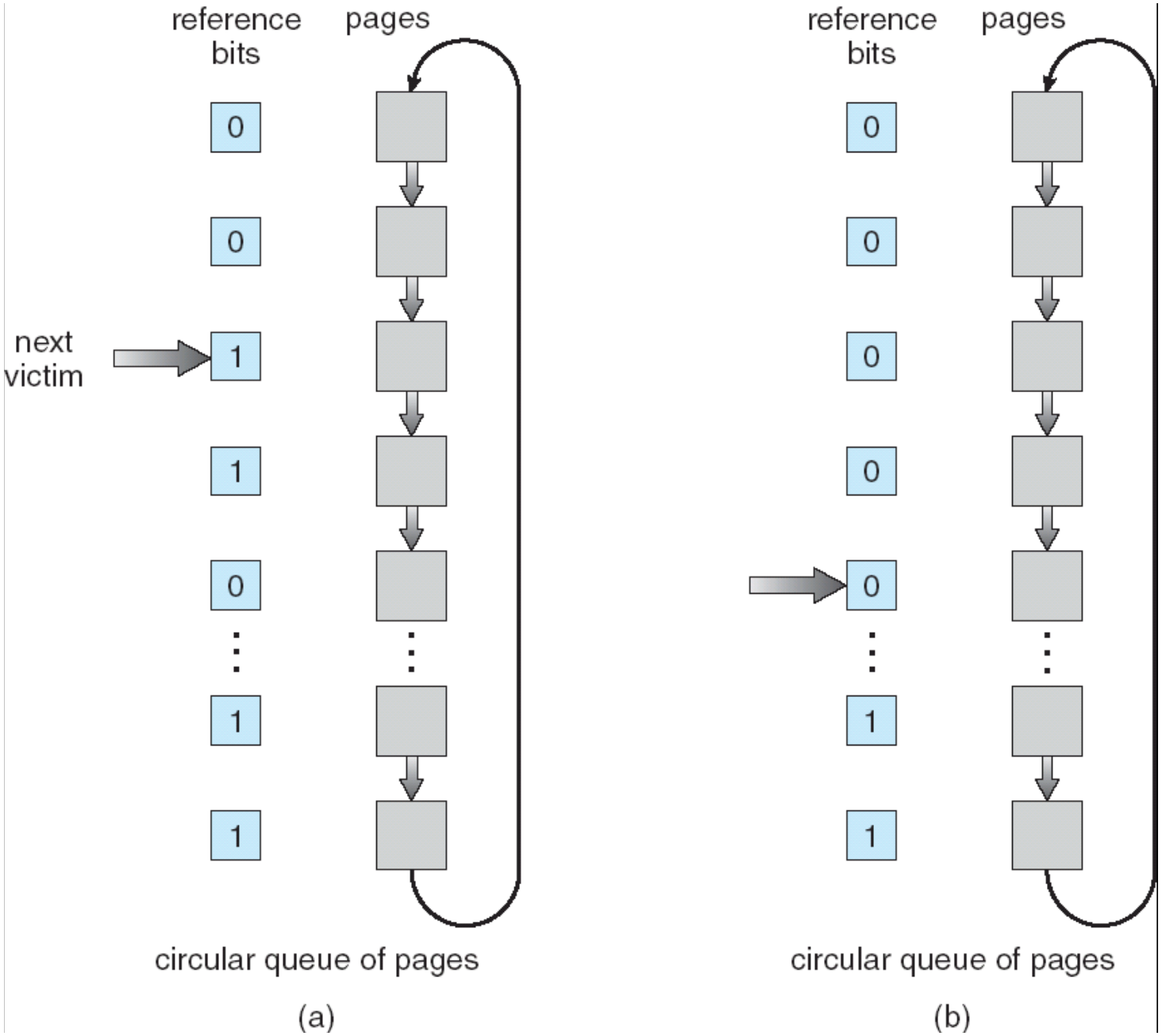


When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

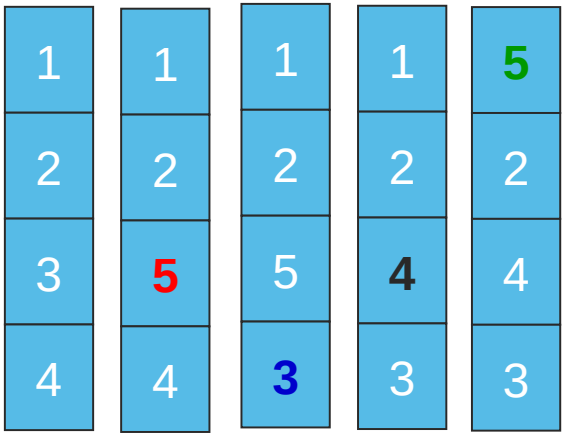
R = 1: Clear R and advance hand

Clock Replacement Example



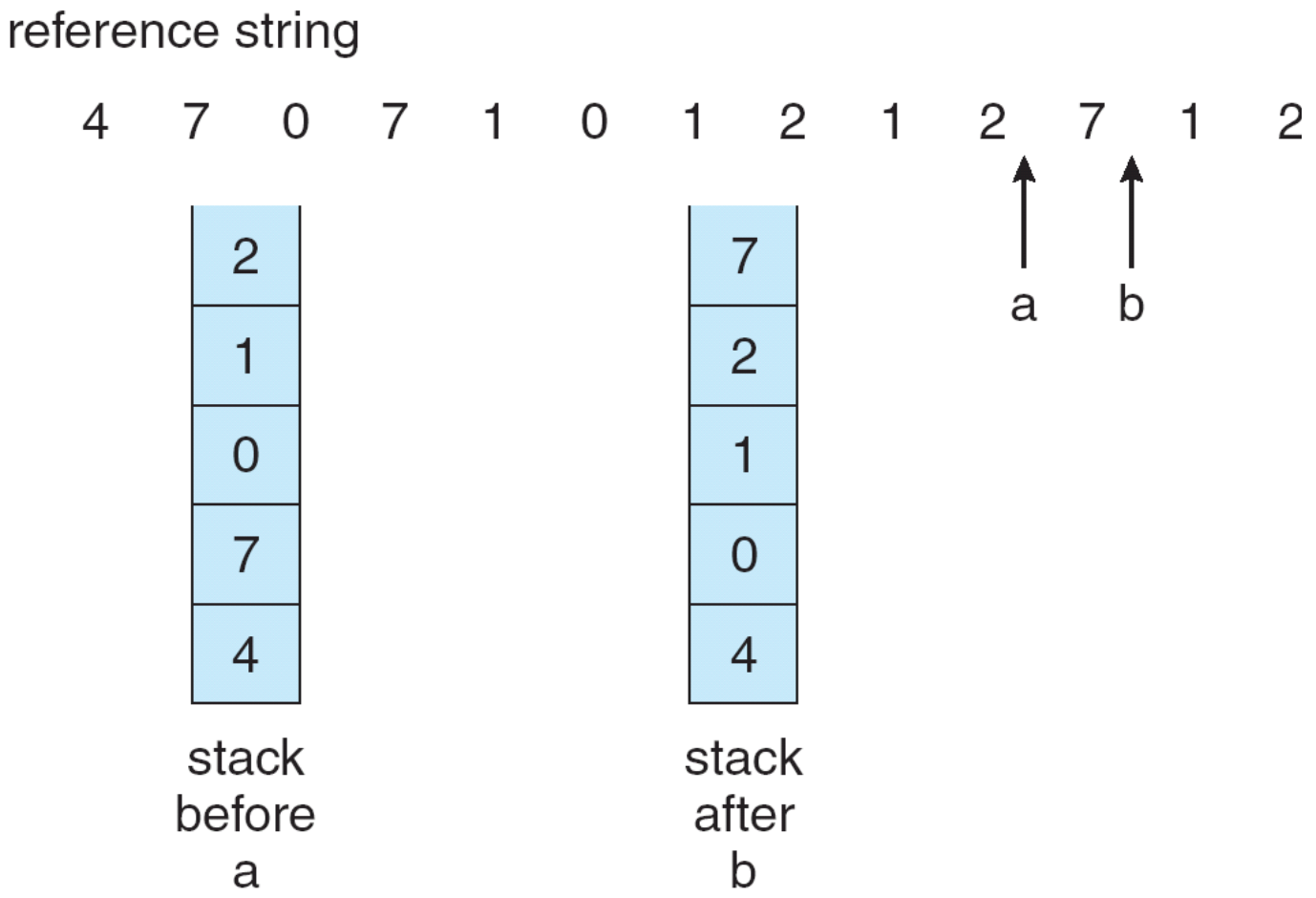
Least Recently Used (LRU)

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



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

Use Of A Stack to Record The Most Recent Page References



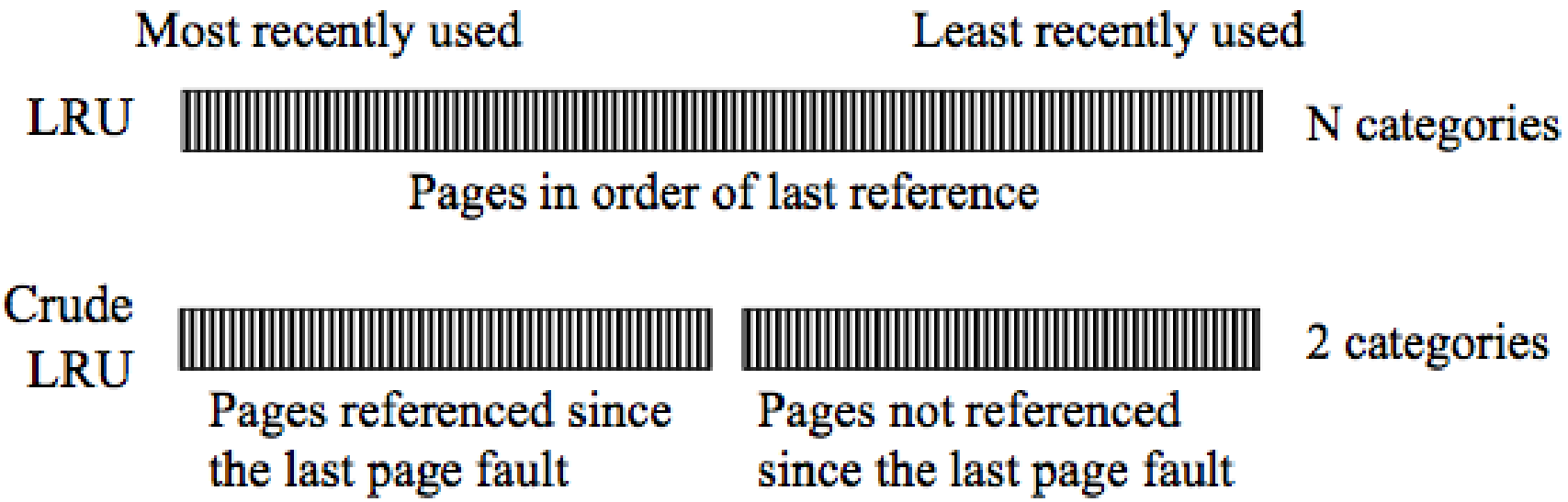
LRU Stack Algorithm

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement
- Do we need other searches?
  - Update this list every memory reference !!
  - Hash table to speedup searches
- Overhead
  - Needs stack to keep entries
  - Needs additional hash table

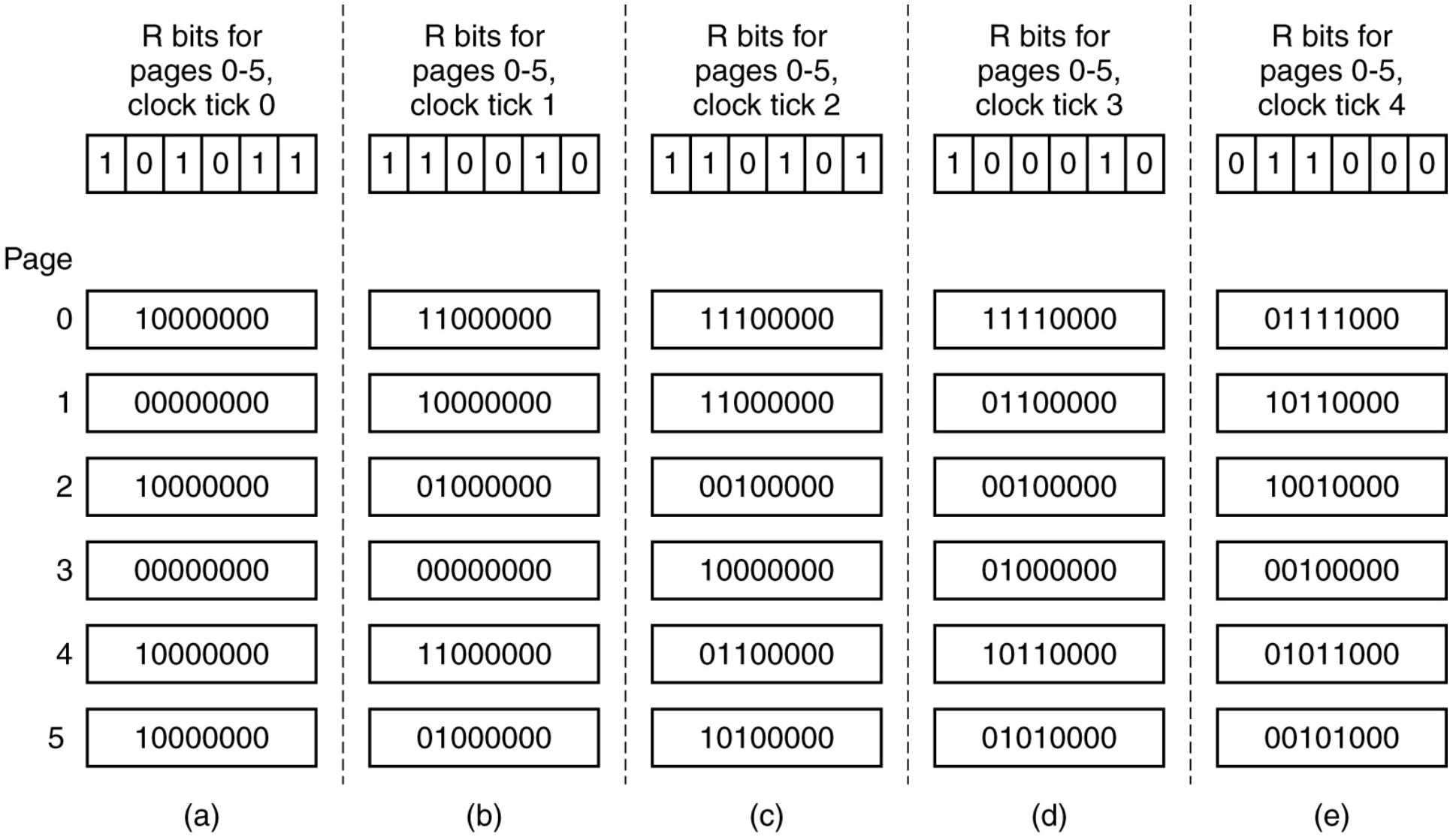
LRU Approximation

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists)
- Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules

LRU Approximation

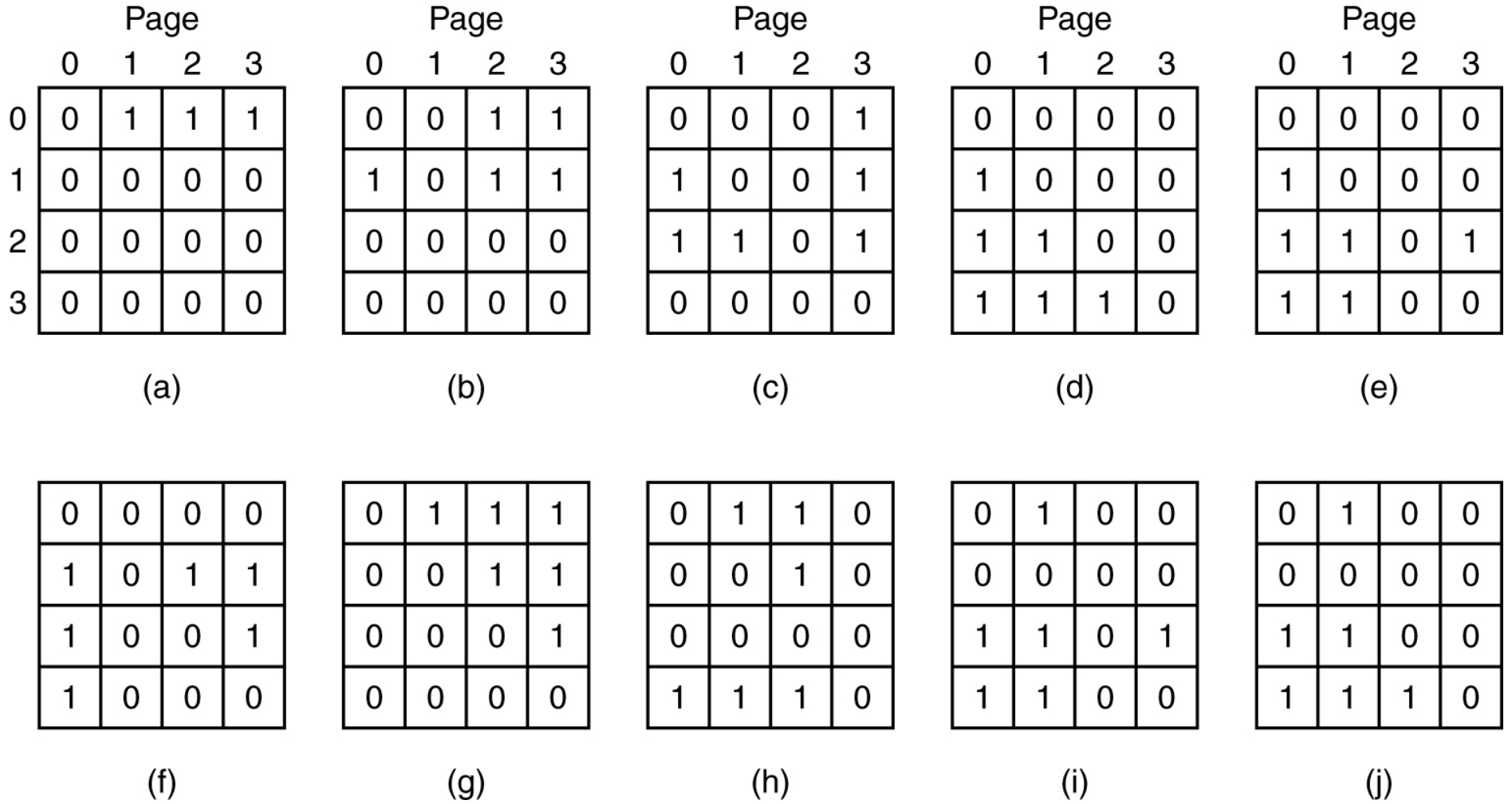


Aging Algorithm for LRU approximation



LRU Hardware

0, 1, 2, 3, 2, 1, 0, ?, ?, ?



Counting Algorithms

- Keep count of references made to each page
- LFU: Least Frequently Used
  - Replace page with smallest count
  - May leave pages that are initially hot but never used again
- MFU: Most Frequently Used (bad idea)
  - Replace page with highest count (pages with smallest count may have just been brought in)
  - Replaces popular pages
- Counting algorithms perform poorly in general

Review: Effective Access Time

- Page fault rate 0 < p < 1.0
  - 0 ^ No page faults
  - 1 ^ Every reference is a fault
- Effective Access Time (EAT):

  EAT= (1-p) * (memory access time)
  + p * (page fault overhead)
  - Page fault overhead = page swap in
  + page swap out
  + restart instruction

Thrashing

Thrashing ⇨ process is busy swapping pages in and out

- Suppose there are many users, processes are making frequent references to 50 pages, memory has 49
- Each time one page is brought in, another page, whose contents will soon be referenced, is thrown out
- What is the average memory access time?
- The system is spending most of its time paging!
- The progress of programs makes it look like memory access is as slow as disk, rather than disk being as fast as memory

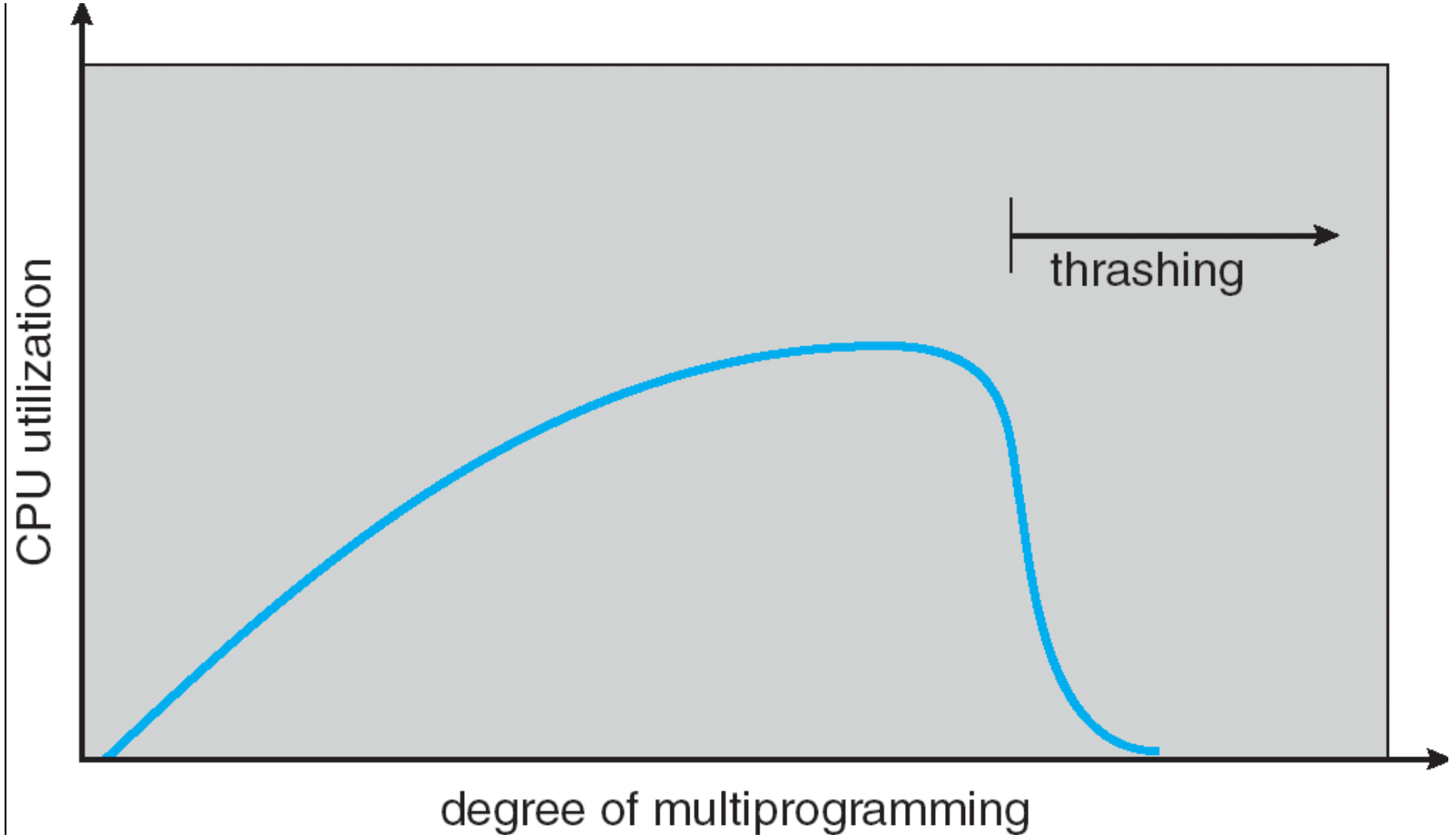
Example: Looping Reference

- Application repeatedly scanning 5 pages
  - 4 Frames, 5 pages
  - Reference pattern: 1, 2, 3, 4, 5

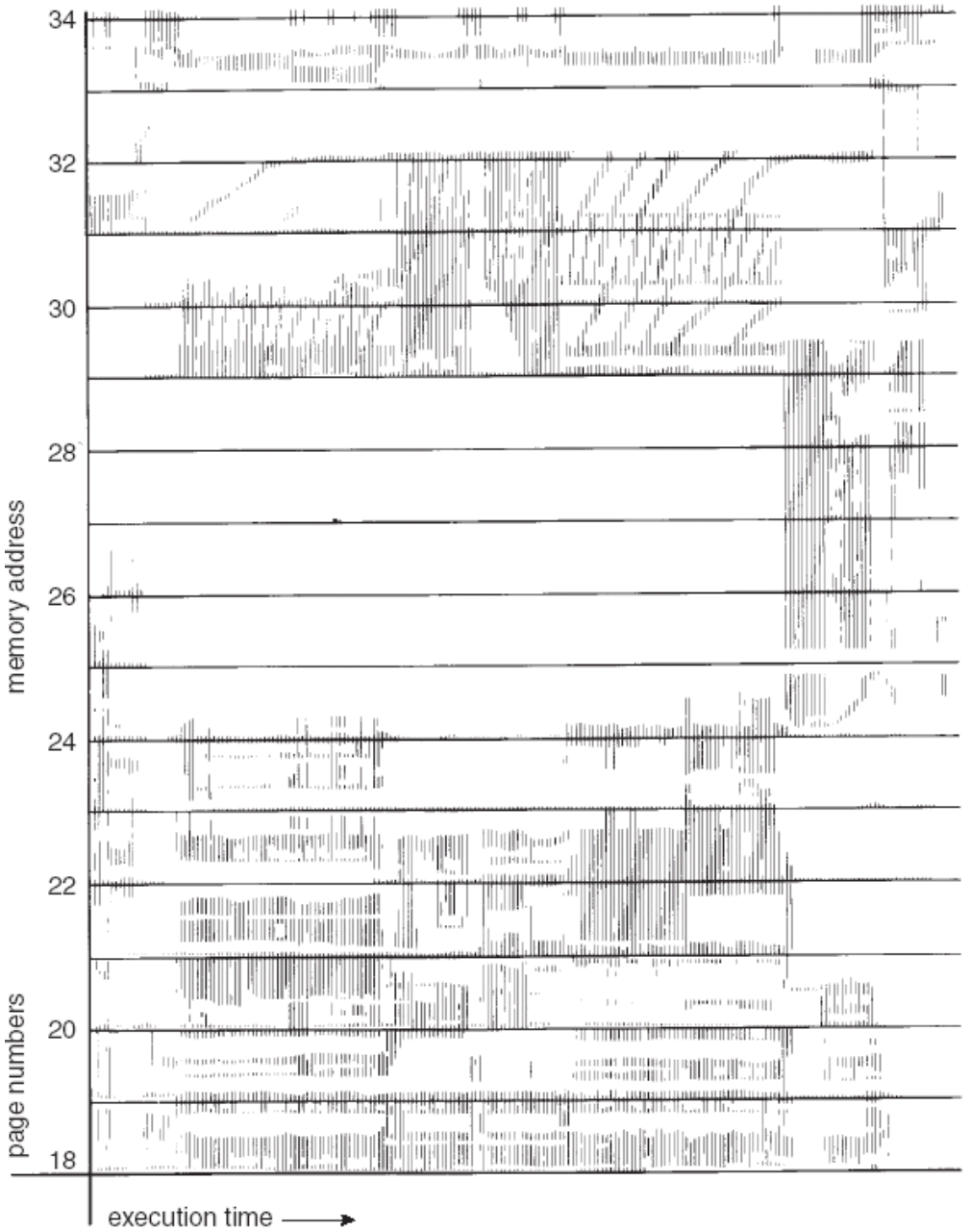
- What happens if LRU is used
  - Thrashing

- Btw, what is the optimal strategy here?
  - MRU ⇨replace the most recent one

Impact of Thrashing



Review: Locality of Memory References



Reference Locality

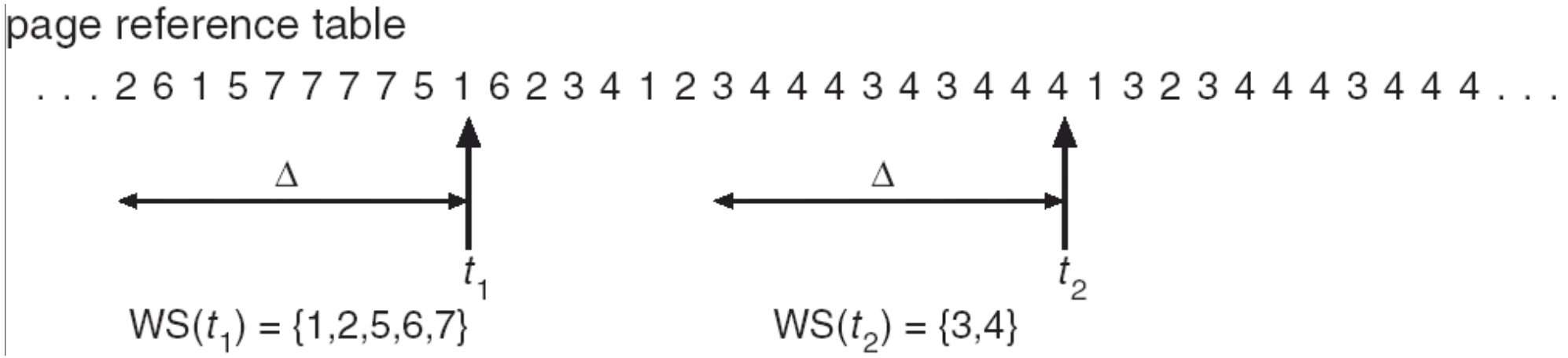
- 80/20 rule
  - > 80% memory references are made by < 20% of code

- Locality in memory references
  - Spatial – adjacent pages are references
  - Temporal – repeated references to a page

Working Sets

- An informal definition:
  - The collection of pages that a process is working with, and which must thus be resident if the process is to avoid thrashing
- The idea is to use recent needs of a process to predict its future needs
  - Choose delta, the WS parameter
  - At any given time, all pages referenced by a process in its last delta seconds comprise its working set
  - Don’t execute a process unless its working set is resident in memory

Working-set model



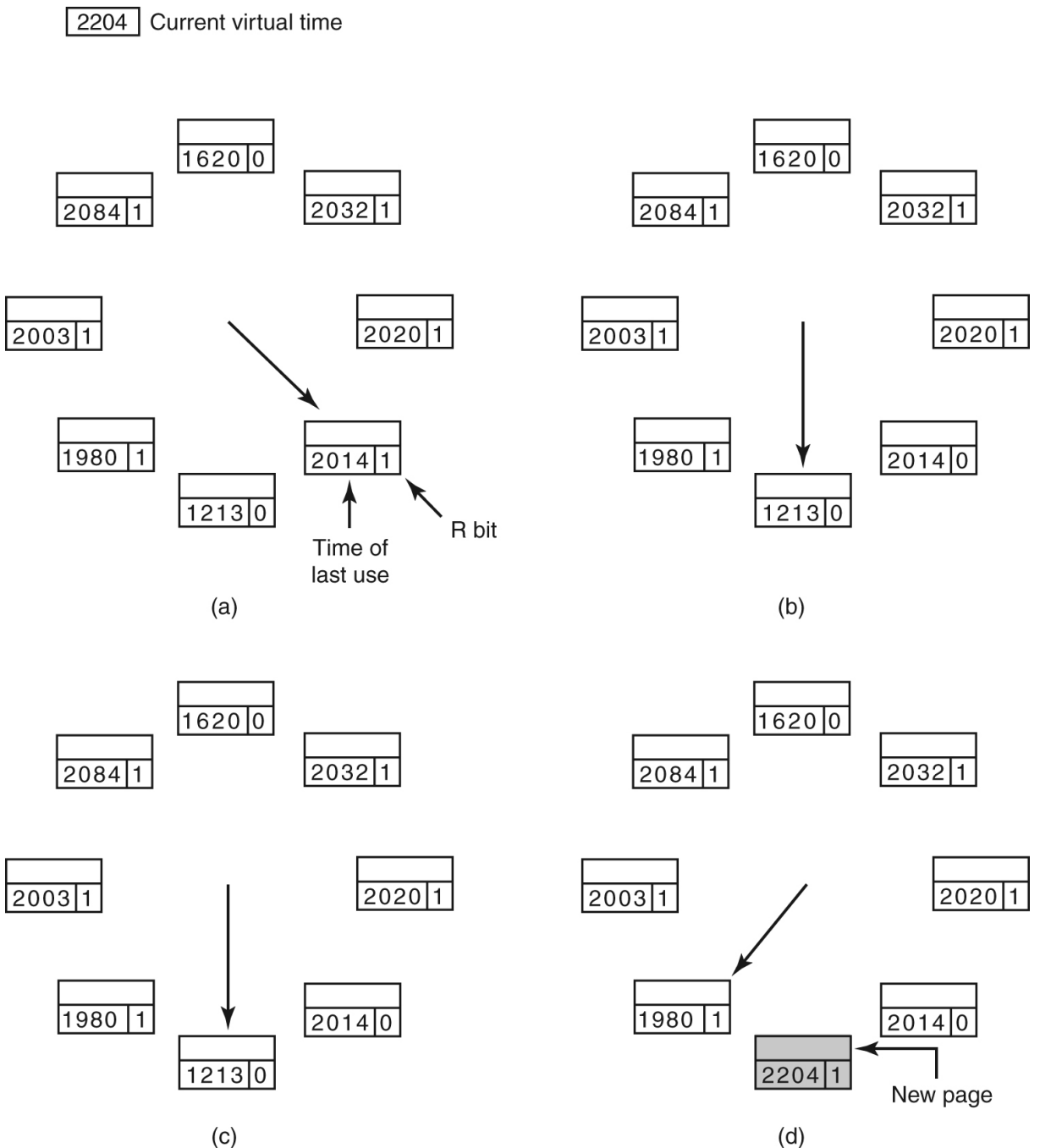
Working Set Implementation

- Take advantage of reference bits
- On a page fault, scan through all pages of the process
- If the reference bit is 1, clear the bit, record the current time for the page
- If the reference bit is 0, check the “last use time”
  - If the page has not been used within Δ, replace the page
  - Otherwise, go to the next
- Add the faulting page to the working set

WSClock Paging Algorithm

- Follow the clock hand
- If the reference bit is 1, set reference bit to 0, set the current time for the page and go to the next
- If the reference bit is 0, check “last use time”
  - If page has been used within δελτα, go to the next
  - If page hasn’t been used within δελτα and modify bit is 1
    - Schedule the page for page out and go to the next
  - If page hasn’t been used within δελτα and modified bit is 0
    - Replace this page

The WSClock Page Replacement Algorithm



Chapter 3 – Memory Management

No Memory Abstraction

Address Spaces

Virtual Memory

Page Replacement Algorithms

Design Issues for Paging Systems

Implementation Issues

Segmentation

Frame Allocation

- Equal allocation
  - If there are 100 frames and 5 processes, each proces gets 20 frames
- Proportional allocation
  - Allocate frames proportionally to size of process

$s_i =$ size of process p_i

$S = \sum s_i$

$m =$ total number of frames

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

$m = 64$

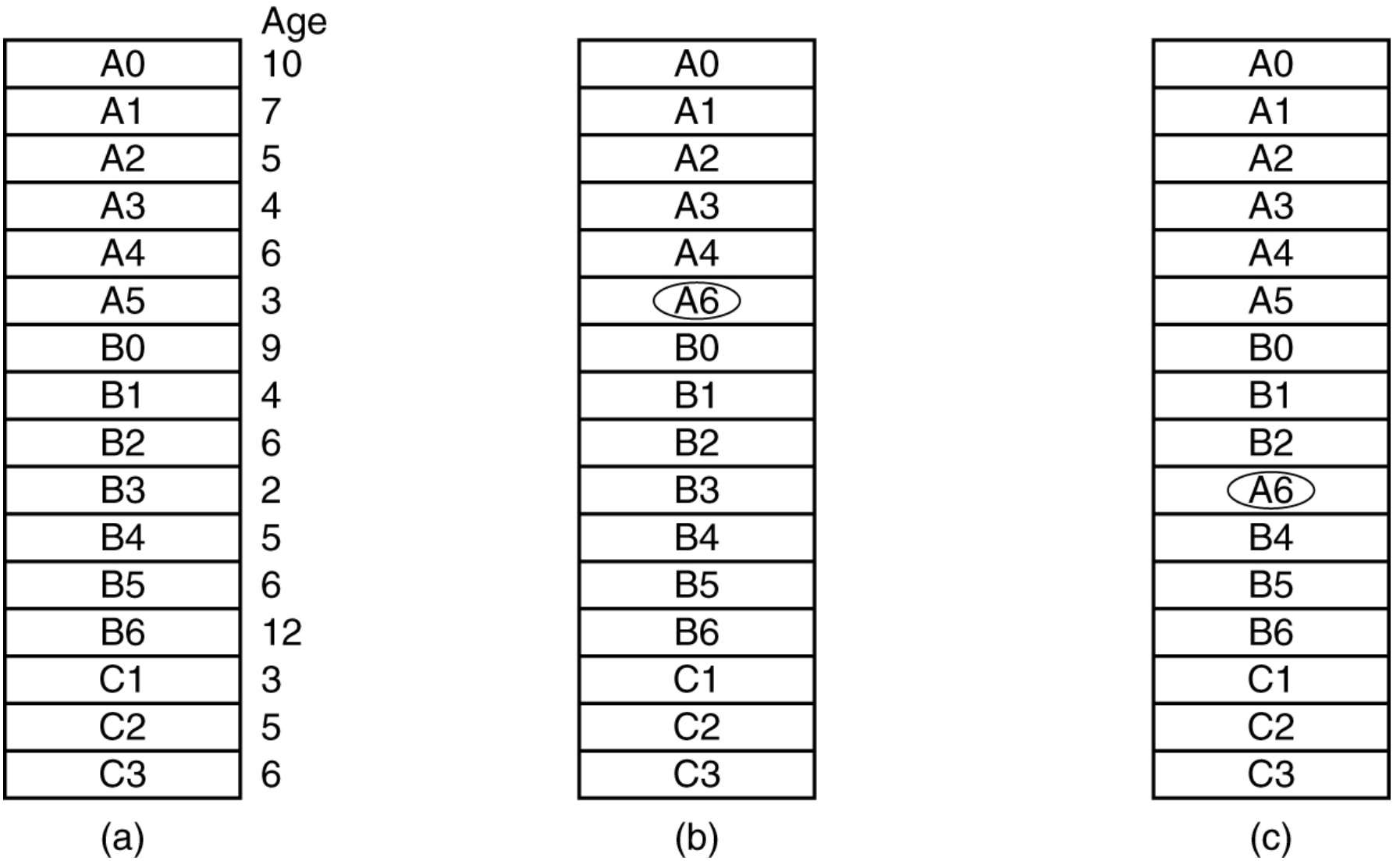
$s_i = 10$

$s_2 = 127$

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

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

Global and Local Allocation



Global vs Local Allocation

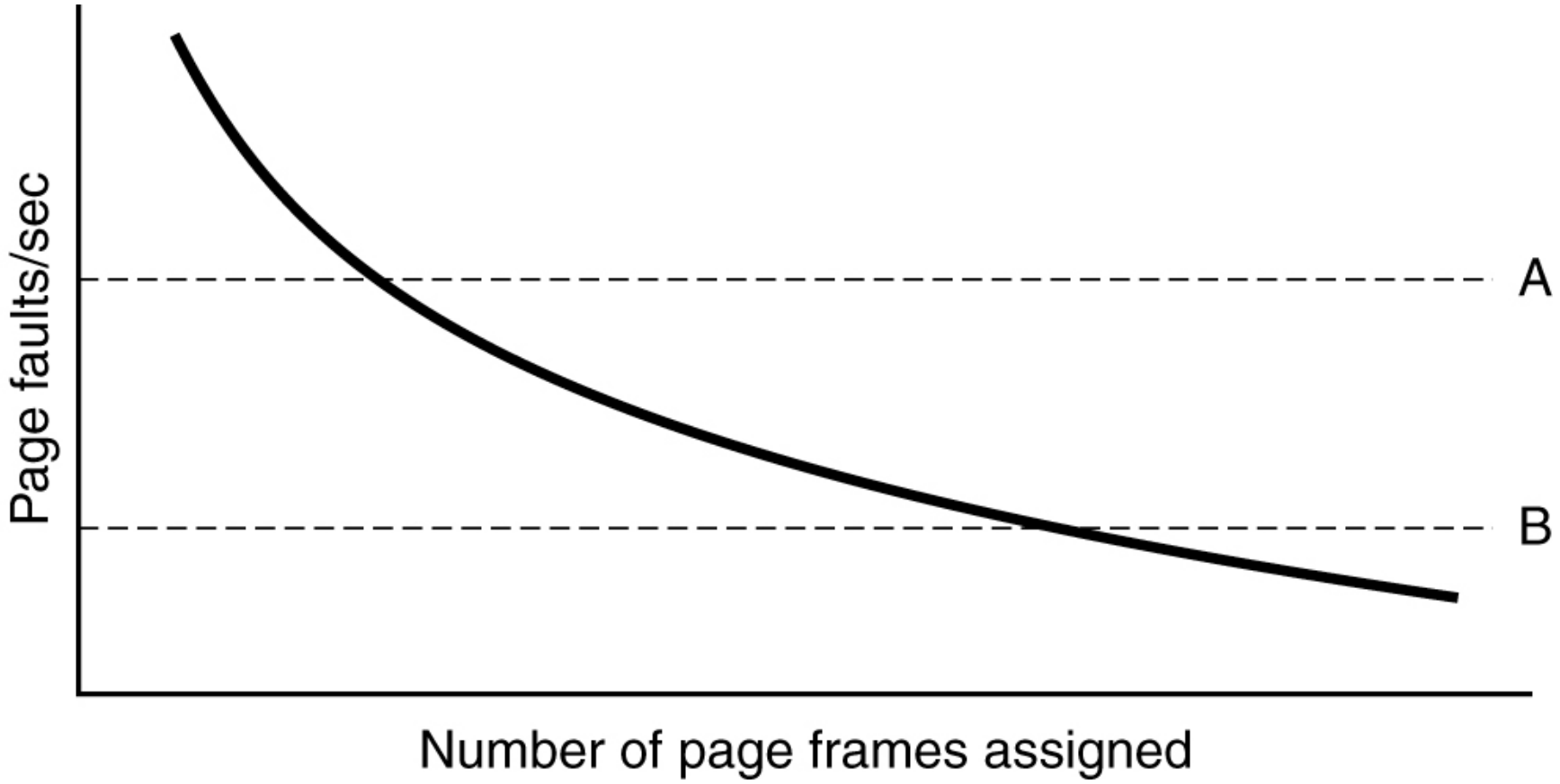
- Global: Process selects a replacement frame from the set of all frames (one process can take a frame from another)
  - Replaces least useful pages in the system
  - May penalize slow processes
  - May benefit processes that quickly scan pages
- Local: Each process selects from its own set of allocated frames
  - May not be fair (processes grow and shrink)
  - May replace frames not optimal for multiprogramming

PFF for Global Replacement

Page Fault Frequency algorithm

- Used to dynamically update the memory allocation as a process runs
- Increase or decrease the size of the allocation set
- Count number of faults per second
- Use running average
  - Add PFF count from previous time period to current mean and divide by two
- If count/average is too high, add frames
- If count/average is too low, remove frames

PFF



- PFF > A : Increase number of frames
- PFF < B: Decrease number of frames

Load Control

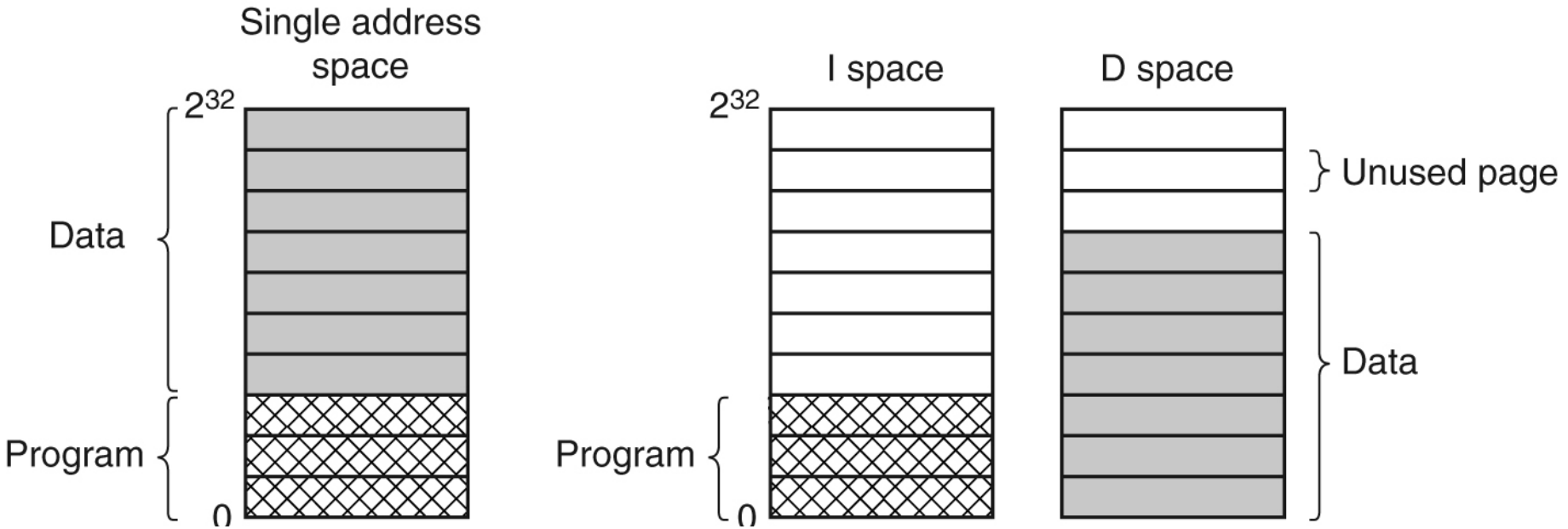
- Despite good design, system can still thrash
- e.g.: PFF indicates
  - Some processes need more memory
  - No processes need less
- Solution?
  - Reduce number of processes competing for mem
    - Swap one or more to disk, divide their frames amongst others
    - Reconsider degree of multiprogramming

Page Size

- Small page size
  - Less internal fragmentation
  - Better fit for various data structures,code sections
  - Less unused program in memory
  - Programs need many pages ^ large page tables
- Large page size
  - More internal fragmentation
  - More unused program in memory
  - Small page table
  - Optimizes disk access(same to fetch 4kb or 16kb)

Instruction and Data Spaces

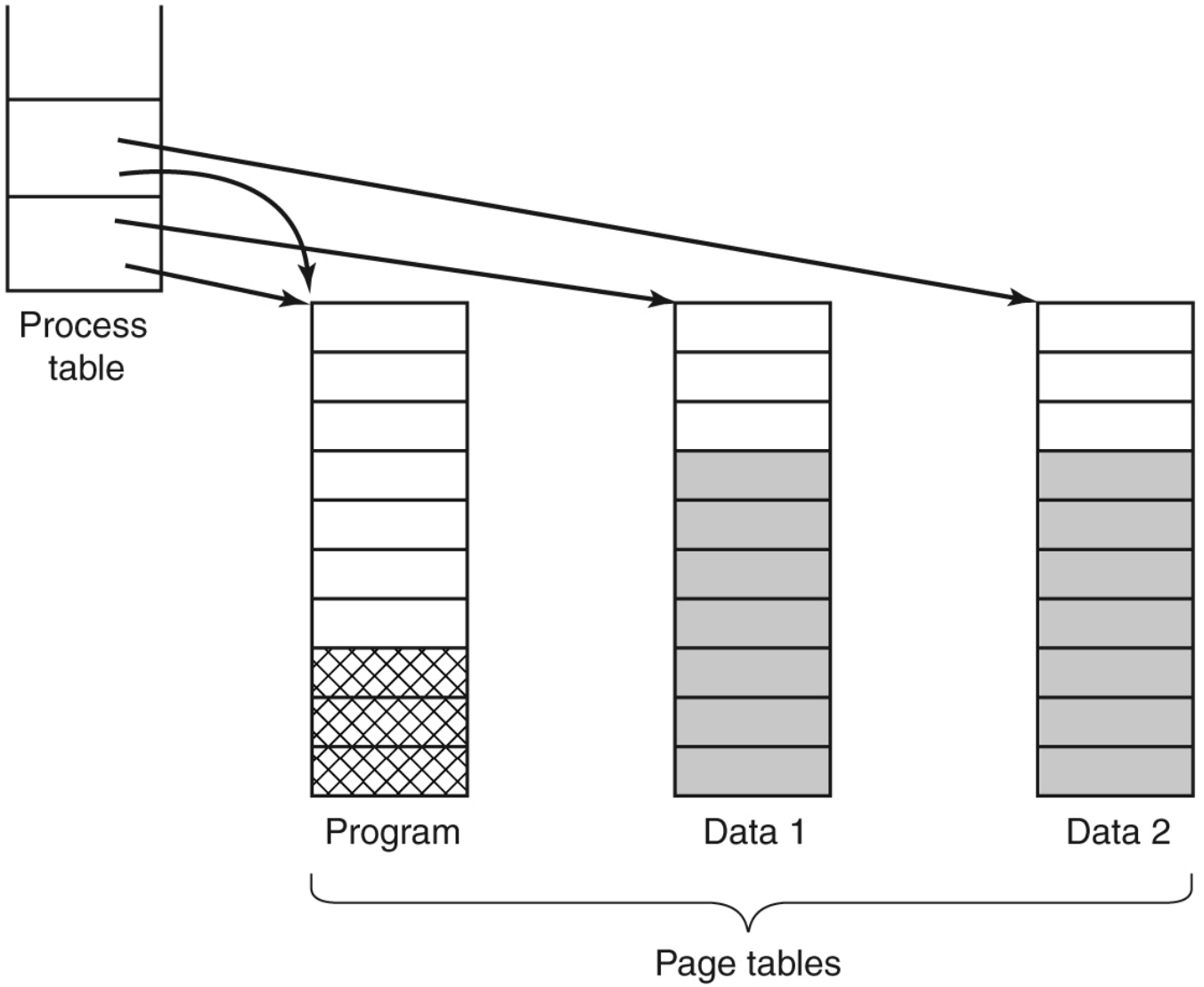
- Separating instructions and data into separate spaces may be beneficial for small address spaces
  - Data and instructions paged separately



Shared Pages

- Shared code
  - One copy of read-only code shared among processes
  - Text editors, compilers, window systems
- Private code and data
  - Each process keeps a separate copy of code and data

Sharing Example



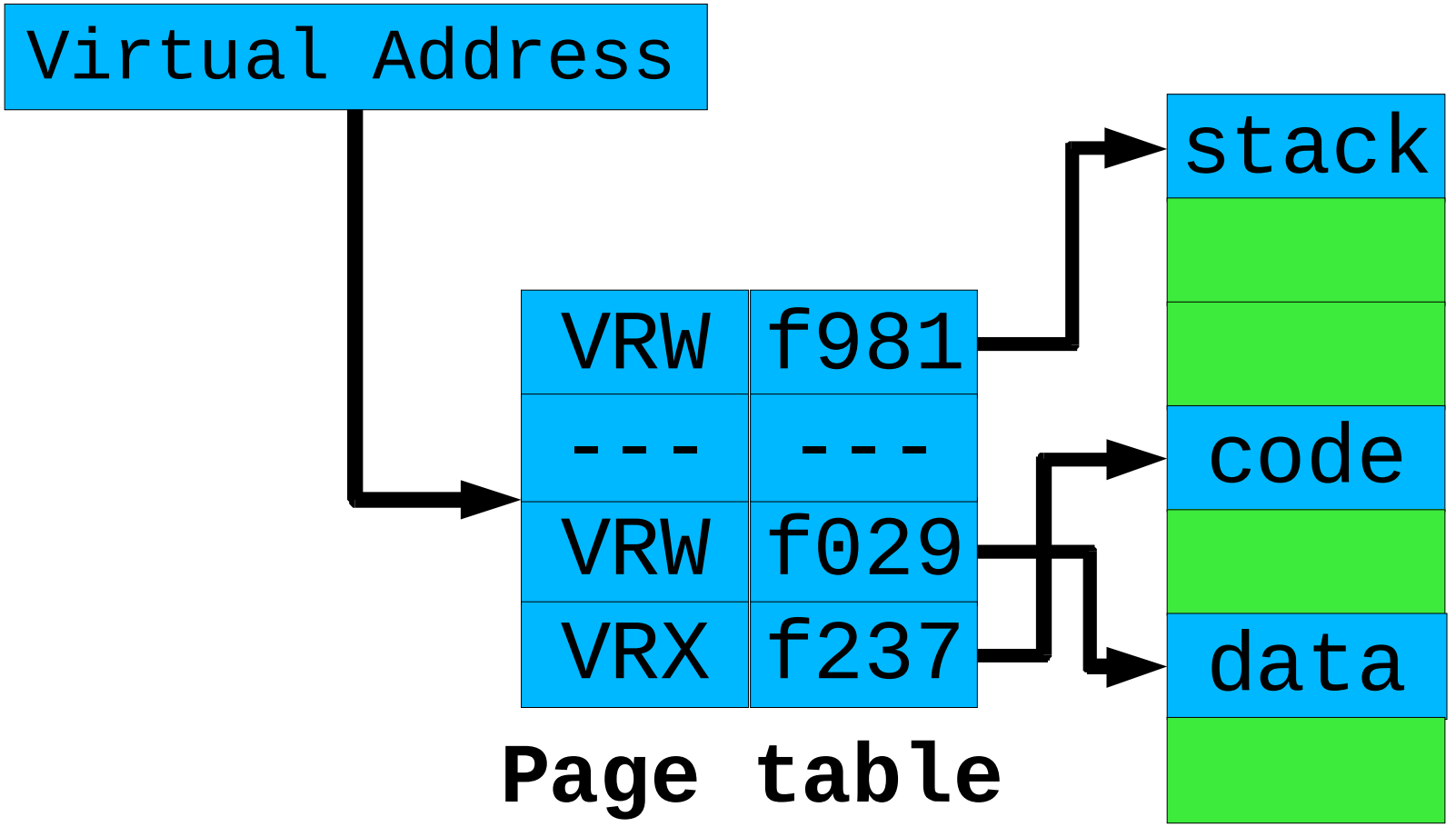
Sharing Issues

- A and B share instruction pages (both are instances of the emacs, for example)
  - A: removed from memory^ B: many page faults
  - A: terminates ^ must retain all pages in use
- A forks to create B
  - Same code, data, stack
  - Copying pages is expensive
    - Many are never modified by new process
      - Think about fork(), exec()
  - Share physical frames?
    - Ok for code pages (read-only), stack pages?

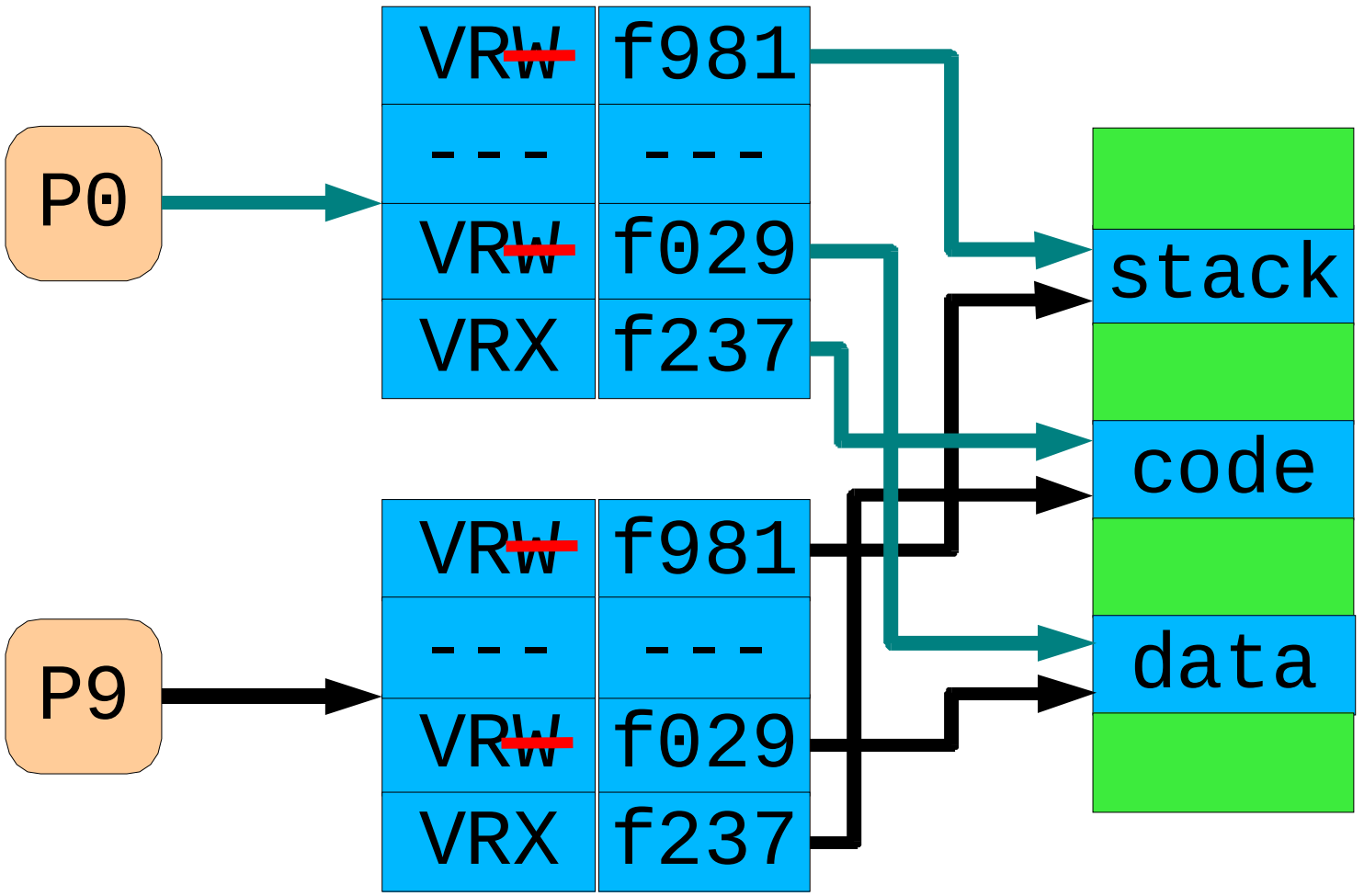
Copy-On-Write

- COW allows both parent and child to initially share same pages in memory
  - Copy page table entries to new process
  - Mark PTEs read-only in both old and new process
  - On process write to a page:
    - Call page fault handler
    - Copy written page into empty frame
    - Mark pages RW in both PTEs
- COW is efficient
  - Only modified pages are copied

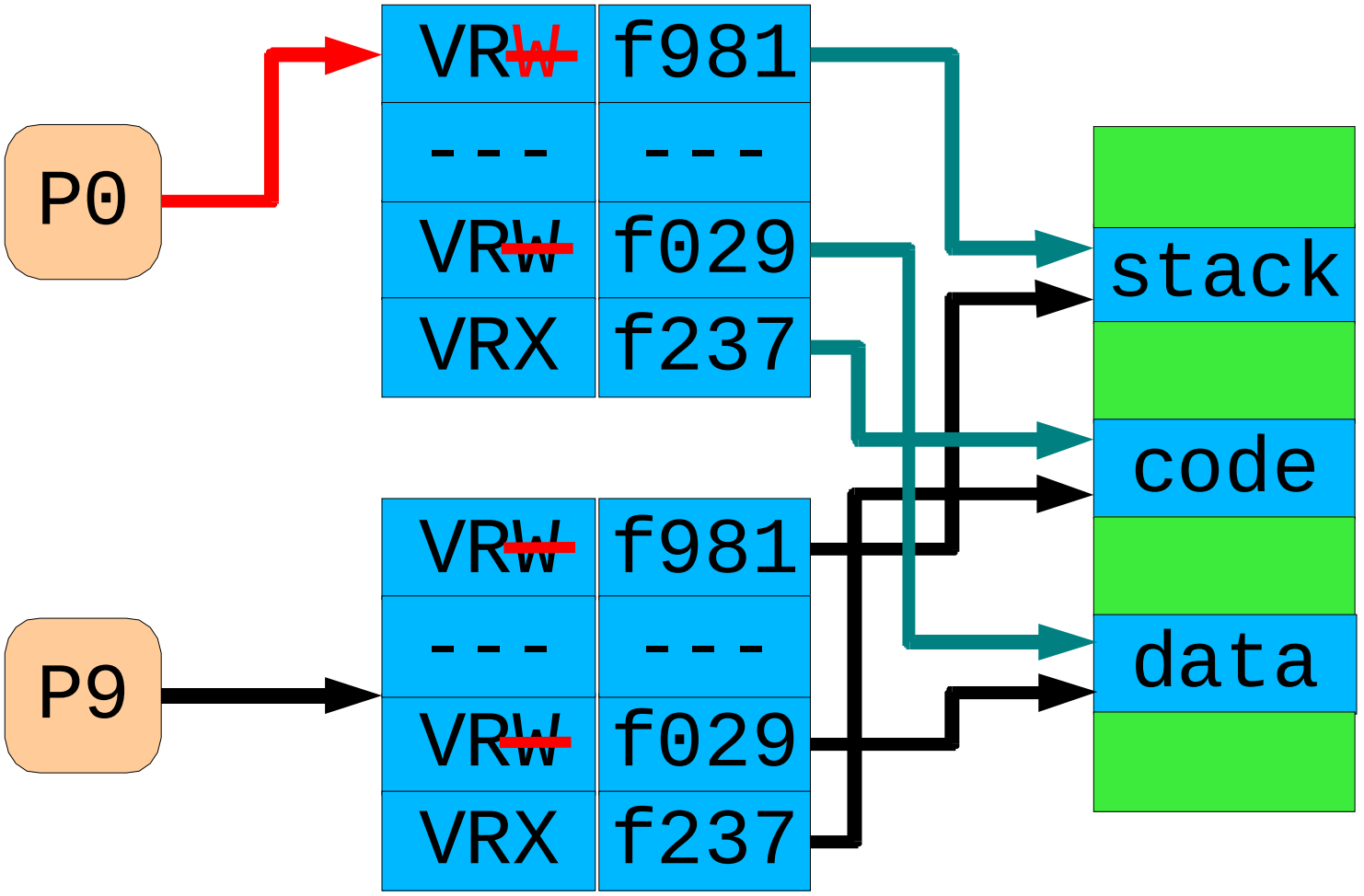
Example Page Table



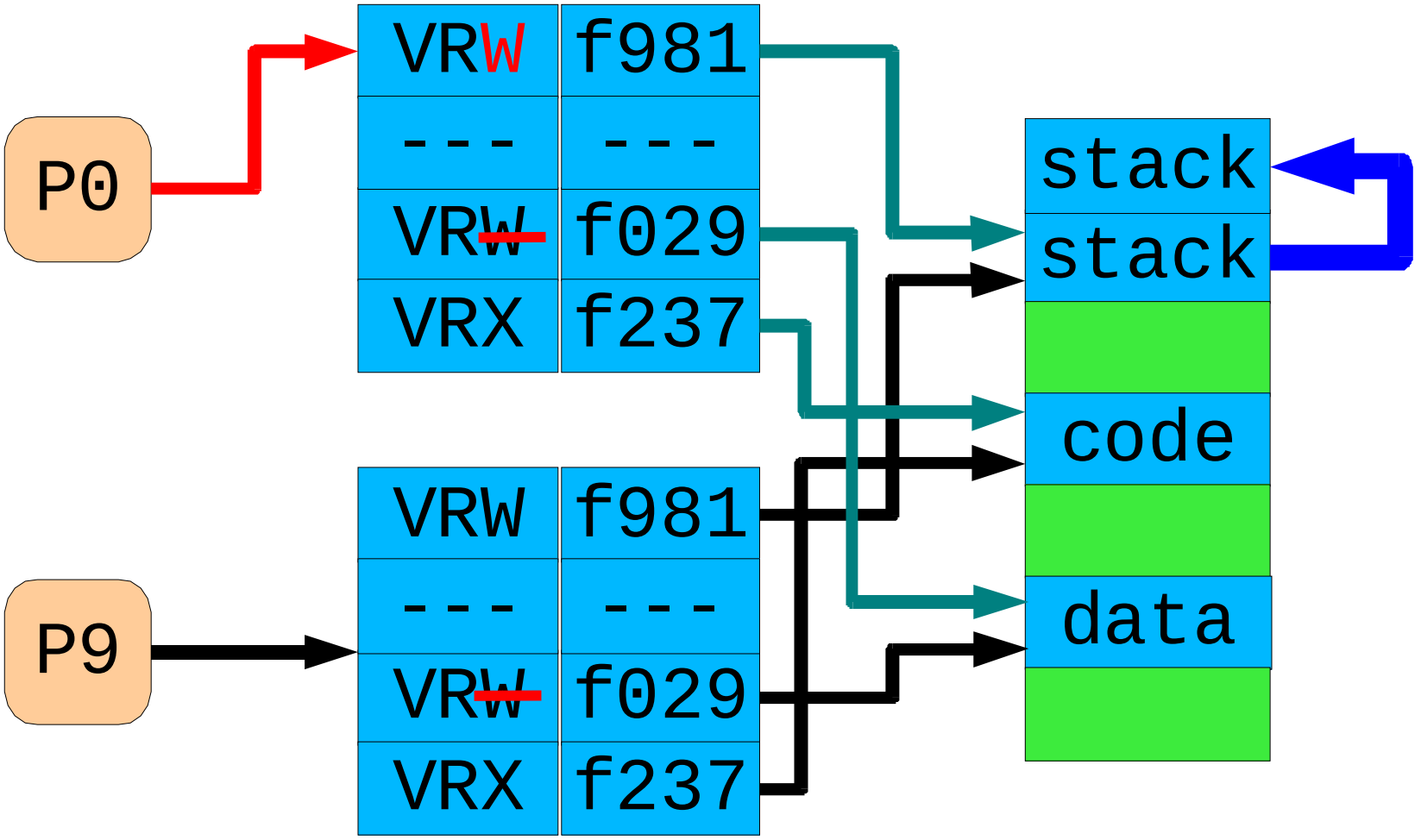
Copy-on-Write of Address Space



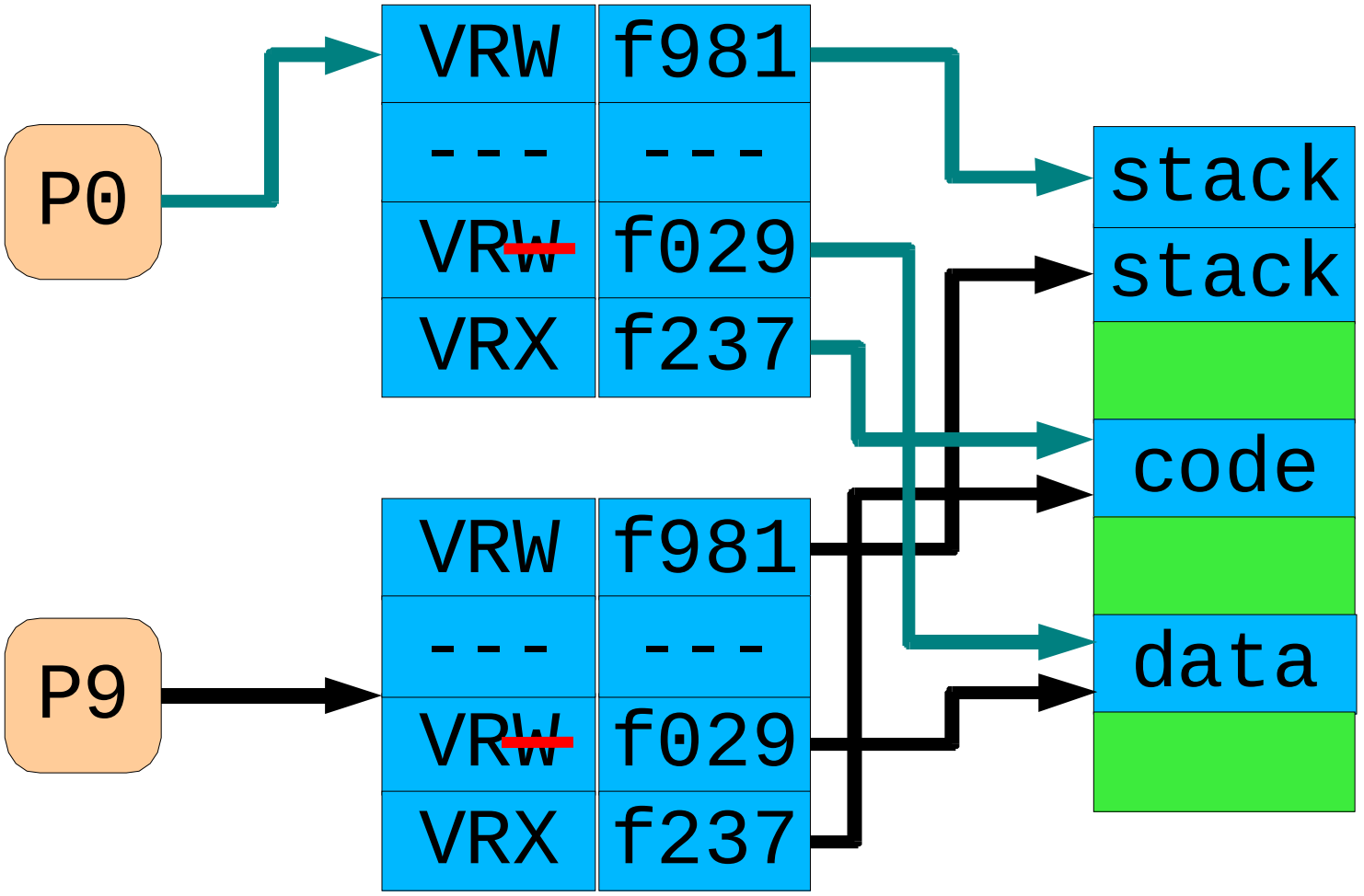
Memory Write ⇨ Permission Fault



Copy Into Blank Frame



Adjust PTE frame pointer, access



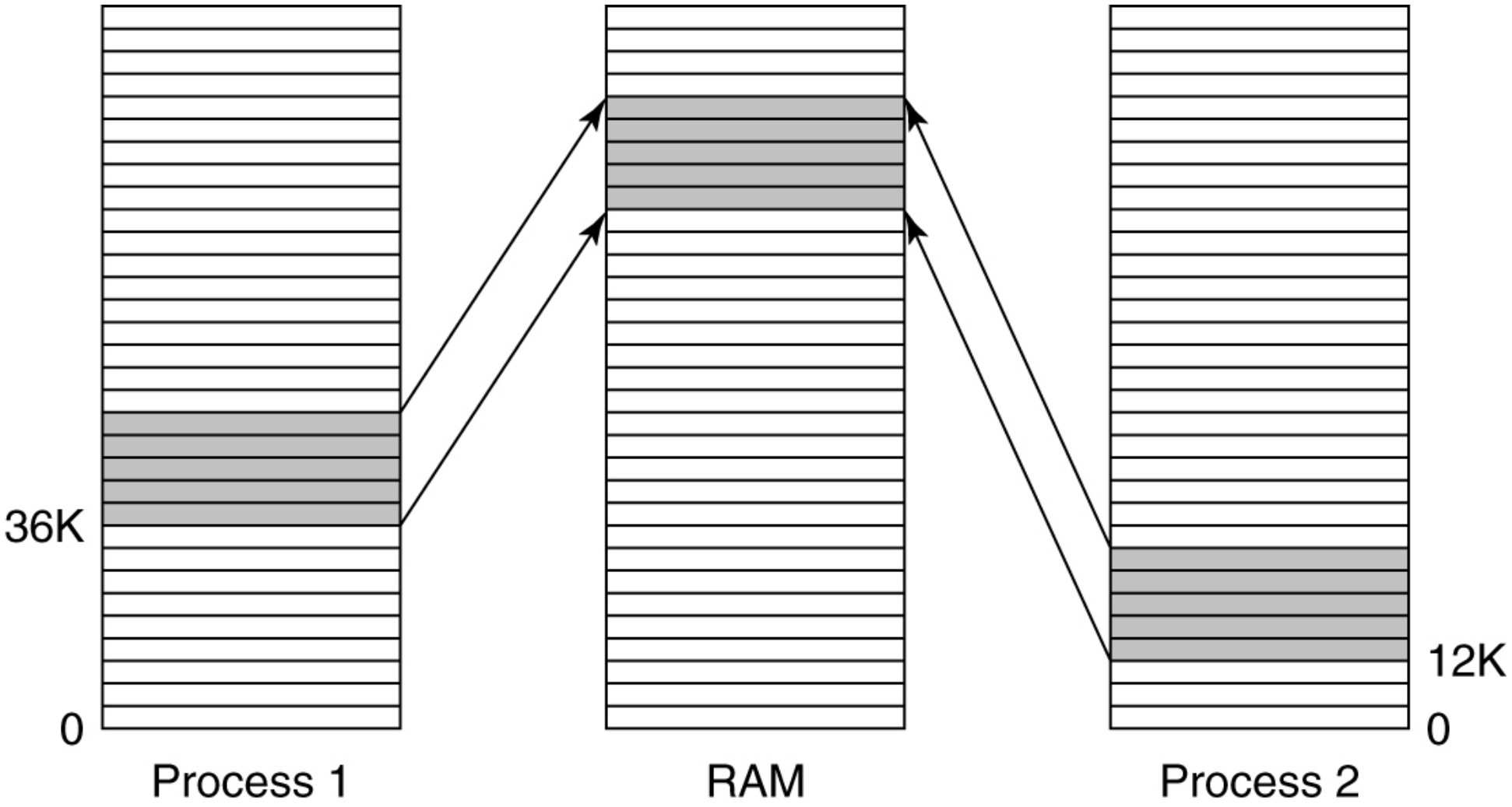
Shared Libraries

- Called DLLs in Windows
- Traditional linking

  ld *.o -lc -lm

  - Functions called in object files that aren't defined
    - Undefined external
    - Look for in libraries
    - Undefined externals can call other undefined externals
  - Called functions from libraries included in binary
- Shared libraries solve problem of wasting space

Shared Libraries



Memory-Mapped Files

- Process can map a file into its virtual space via system call
- Other processes can also map the same file – data sharing
- Writes to mapped file are visible to all processes mapping the file
- Used by process loader to bring in executables and loadable modules into memory

Cleaning Policy

- Need for a background process, paging daemon
  - Periodically inspects state of memory
  - Want to avoid last-minute saving of dirty pages
- When too few frames are free
  - Selects pages to evict using a replacement algorithm
  - Write dirty pages but keep data around
- If hit in the clean list reallocate, replace otherwise

- Can use same circular list (clock)
  - Additional hand in the clock to clean pages

Chapter 3 – Memory Management

No Memory Abstraction

Address Spaces

Virtual Memory

Page Replacement Algorithms

Design Issues for Paging Systems

Implementation Issues

Segmentation

OS Requirements for Paging

- Process creation
  - Determine how large the process is initially
  - Allocate page table and initialize
  - Allocate disk swap area
- Process execution
  - Reset MMU for new process
  - Flush TLB
  - Make new process' page table current
-

OS Requirements for Paging (2)

- Page fault
  - Read hardware registers to find faulting address
  - Compute needed page, find it on disk
  - Find available page frame
    - Replace old one if necessary
  - Read needed page into page frame
  - Back up program counter to rerun faulting inst.
- Process exit
  - Release page table, pages, and disk space
  - Do not release shared pages unless last process

Page Fault

- 1 Hardware traps to kernel
  - Save PC on the stack
- 2 Assembly routine saves registers
  - Calls OS
  - Remember interrupt handler in DLXOS and call to OS's dointerrupt(...)
- 3 OS discovers page fault and finds virtual page
  - Either from hw register or instruction
- 4 Check address is valid and protection consistent
  - Otherwise, signal process or kill
  - If valid, find free frame or replacement victim
-

Page Fault

- 5 If selected victim is dirty, must transfer to disk
  - Dirty page marked busy
  - Context switch suspends faulting process
  - Another runs until disk transfer complete
- 6 Have clean frame
  - OS looks up disk address, schedules read
  - While page is loaded, other processes may run
- 7 Disk interrupt indicates page read complete
  - Page tables updated, frame marked normal

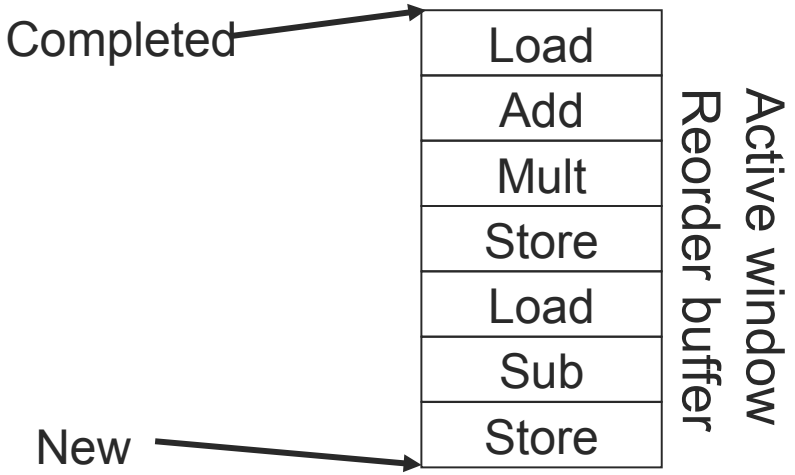
Page Fault

8 Faulting instruction backed up

- 9 Faulting process scheduled, OS returns to routine that called it
- 10 Registers reloaded, return to user space to continue operation as if nothing happened

Instruction Backup

- An instruction causing a page fault has to be restarted
- Not a problem in current implementations
- Instructions are not committed before finished
- Hardware stores temporary results
  - Once everything is resolved (address, alu, operands) the permanent register file is updated with the state



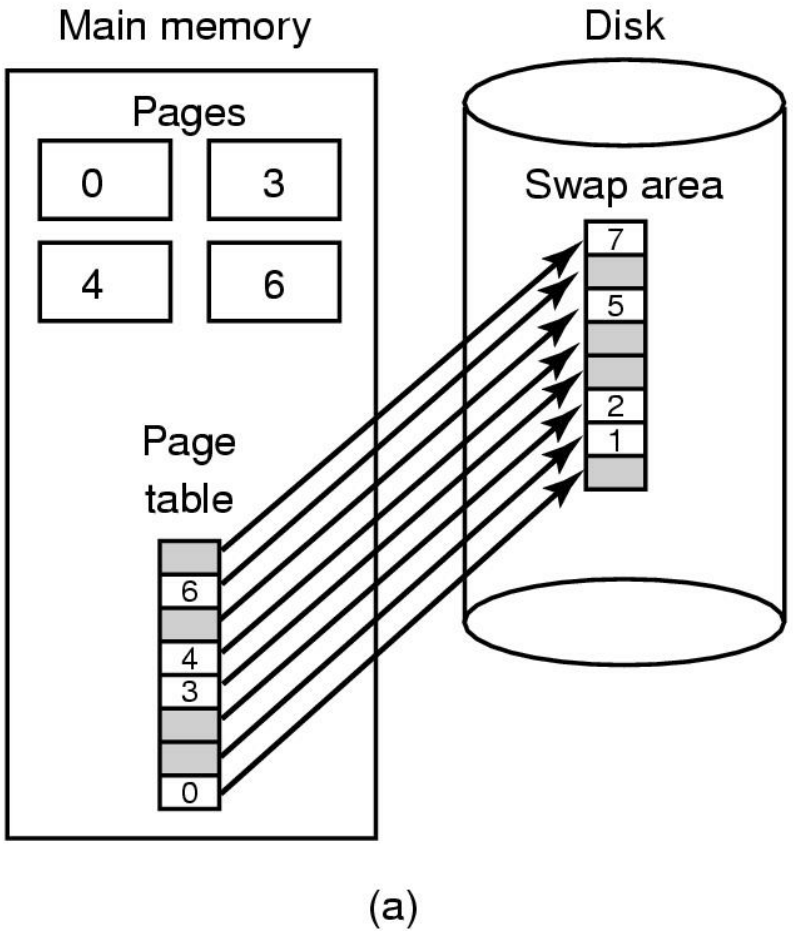
Locking Pages in Memory

- Virtual memory and I/O occasionally interact

- Proc issues call for read from device into buffer
  - while waiting for I/O, another process starts up
  - has a page fault
  - buffer for the first proc may be chosen to be paged out

- Need to specify some pages locked
  - Exempted from being target pages

Paging to Static Swap Area

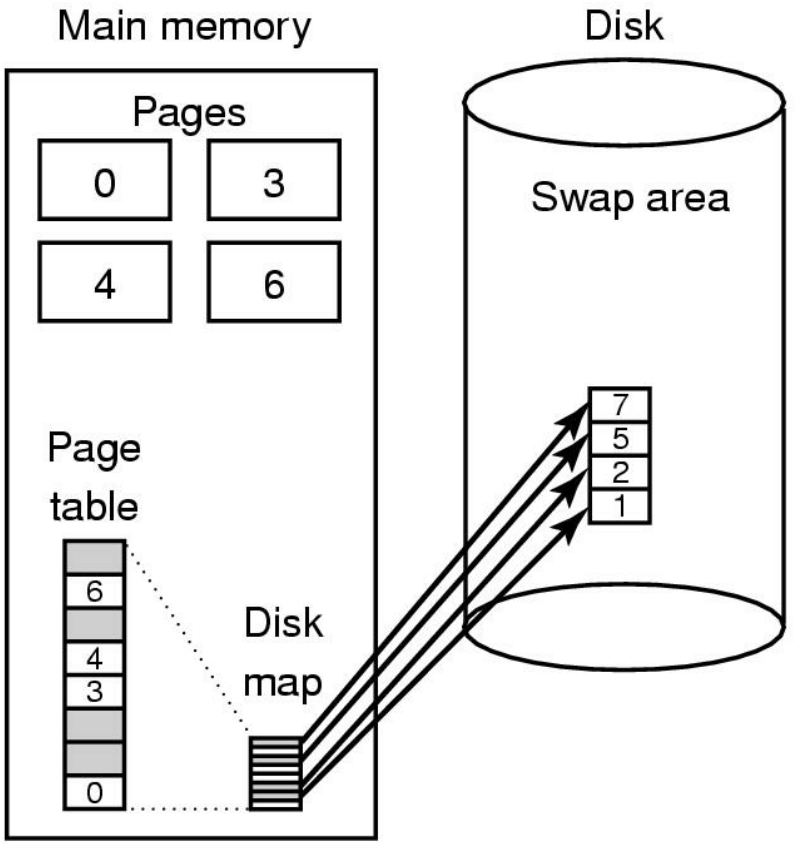


(a)

- Swap area for entire process allocated
  - Overhead: up to 4GB for 32 bit and more for 64
- Easy mapping
  - One-to-one for each page table entry
  - Have to know only where proc's paging area begins
- Shadow pages on disk must be carefully updated

Paging to Dynamic Swap Area

- No fixed disk address
- Empty pages in swap chosen on the fly
- No shadow pages on disk
- Need a disk map



(b)

Swap Area

- Fixed-size
  - Create a disk partition
- Dynamic
  - Use regular preallocated files
  - Can use one or more
  - Can use executable files for swapping
    - i.e. don't keep a separate copy of program text, reread executable if needed
    - Applicable to shared libraries

Chapter 3 – Memory Management

No Memory Abstraction

Address Spaces

Virtual Memory

Page Replacement Algorithms

Design Issues for Paging Systems

Implementation Issues

Segmentation

Segmentation

- Memory-management scheme that supports user view of memory/program, i.e. a collection of segments.
- A program being compiled is a collection of segments:
  - Source text
  - Symbol table: names/ attributes of variables
  - Table of constants
  - Parse tree: syntactic analysis of the program
  - Call stack