Basic Memory Management

CS 256/456

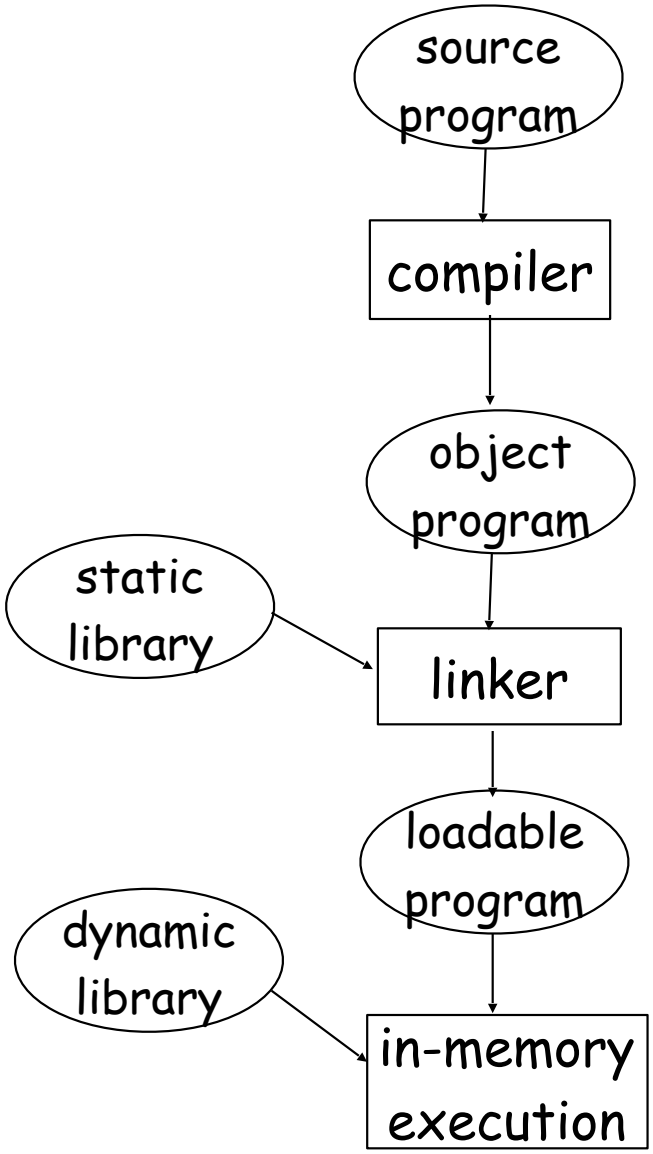
Dept. of Computer Science, University of Rochester

Basic Memory Management

- Program must be brought into memory and placed within a process for it to be run
- Mono-programming
  - running a single user program at a time
- Need for multi-programming
  - utilizing multiple instances of resources (multiple CPUs)
  - overlapping I/O with CPU
- Memory management task #1:
  - Allocate memory space among user programs (keep track of which parts of memory are currently being used and by whom)

Running a user program

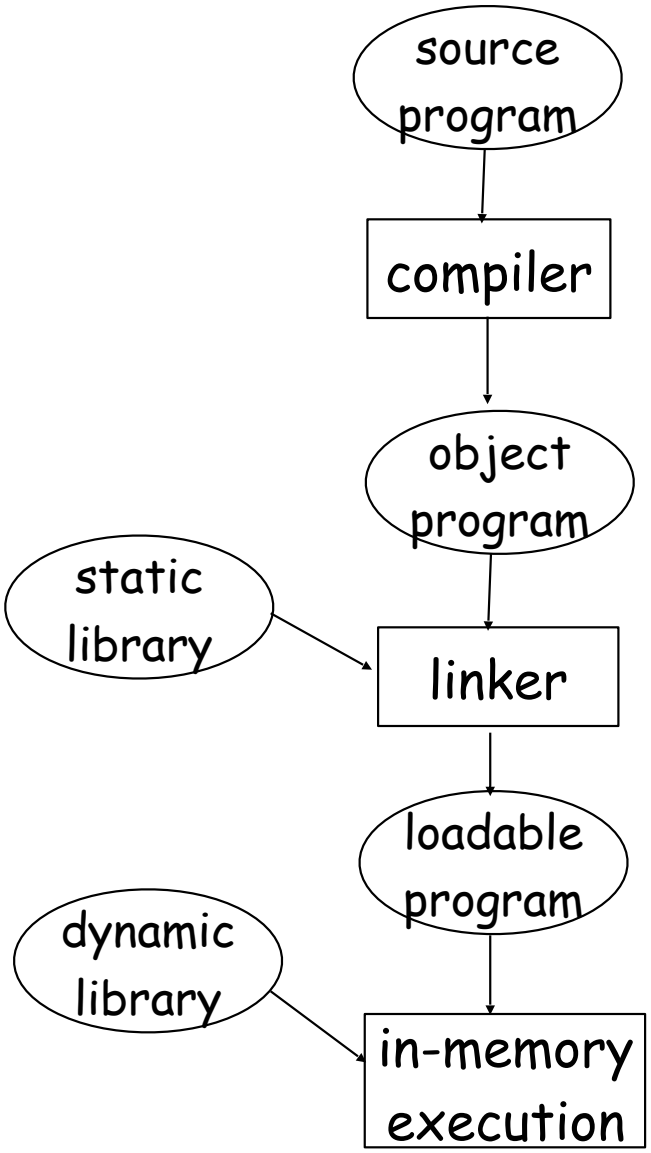
- User programs go through several steps before being run



Address Binding

Binding of instructions and data to physical memory addresses can happen at different stages.

- Compile&link time:
  - If memory location known a priori, absolute code can be generated
  - Must recompile code if starting location changes
- Load time:
  - Must generate relocatable code if memory location is not known at compile time.
- Execution time:
  - Binding delayed until run time
- Compare them on flexibility & protection & overhead



Logical vs. Physical Address Space

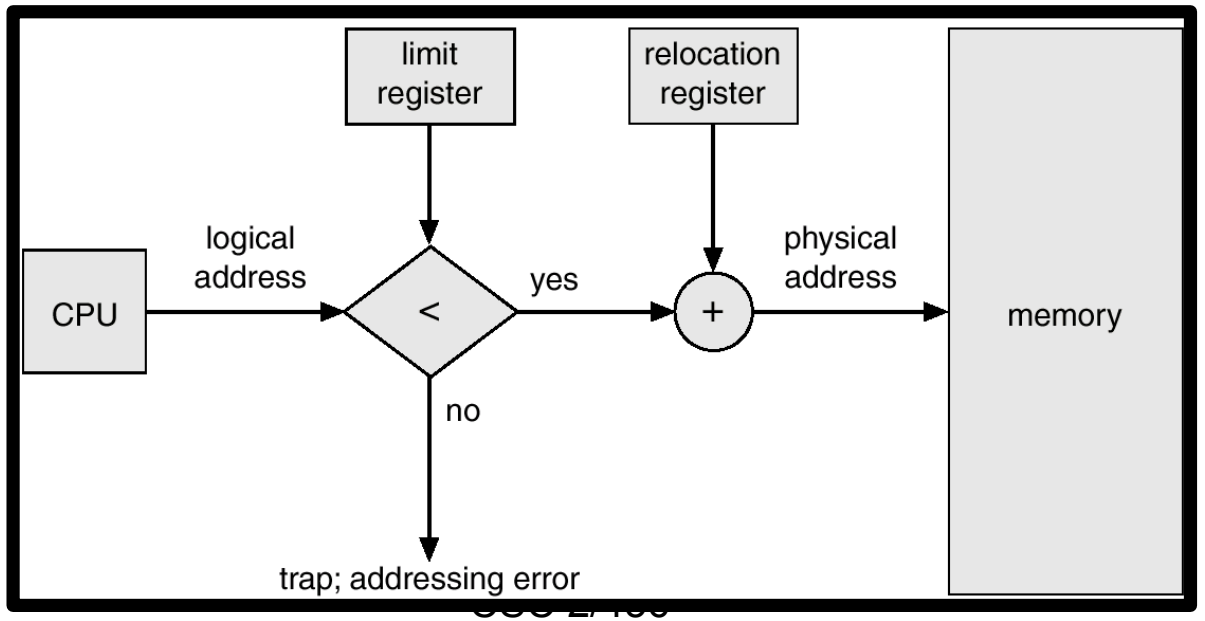
- Two different addresses for execution-time addressing binding:
  - Logical address – those in the loaded user program; often generated at compile time; will be translated at execution time; also referred to as virtual address
  - Physical address – address seen by the physical memory unit

- Memory management task #2:
  - address translation and protection

- Address translation from logical addresses to physical addresses
  - pure software translation is too slow
  - (mostly) done in hardware
    - Memory-mapping unit (MMU): hardware device that maps virtual to physical address; enforces memory protection policies

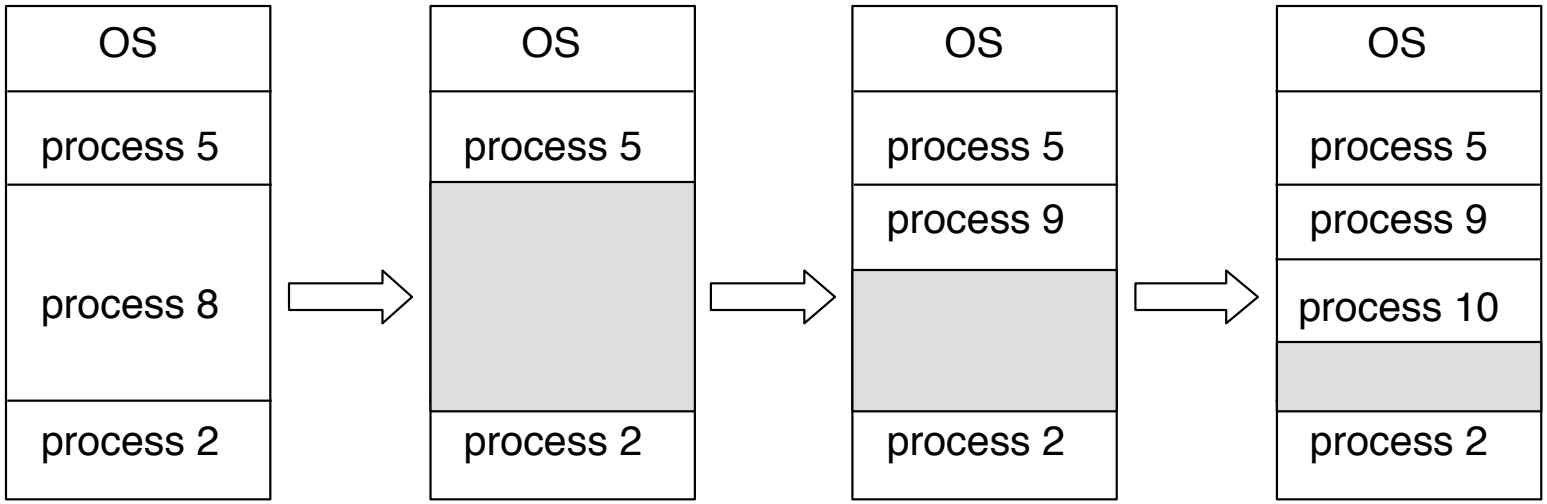
Contiguous Allocation

- Contiguous allocation
  - allocate contiguous memory space for each user program
- MMU: address translation and protection
  - Assume that logical address always starts from 0;
  - Relocation register contains starting physical address;
  - Limit register contains range of logical addresses – each logical address must be less than the limit register.



Contiguous Allocation (Cont.)

- Memory space allocation
  - Available memory blocks of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a free block large enough to accommodate it
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



Space Allocation Strategies

How to satisfy a request of size n from a list of free memory blocks (holes)

- **First-fit**: Allocate the first hole that is big enough.
- **Best-fit**: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole
- **Worst-fit**: Allocate the largest hole; max-heap (the data structure) can help here

Speed & space utilization?

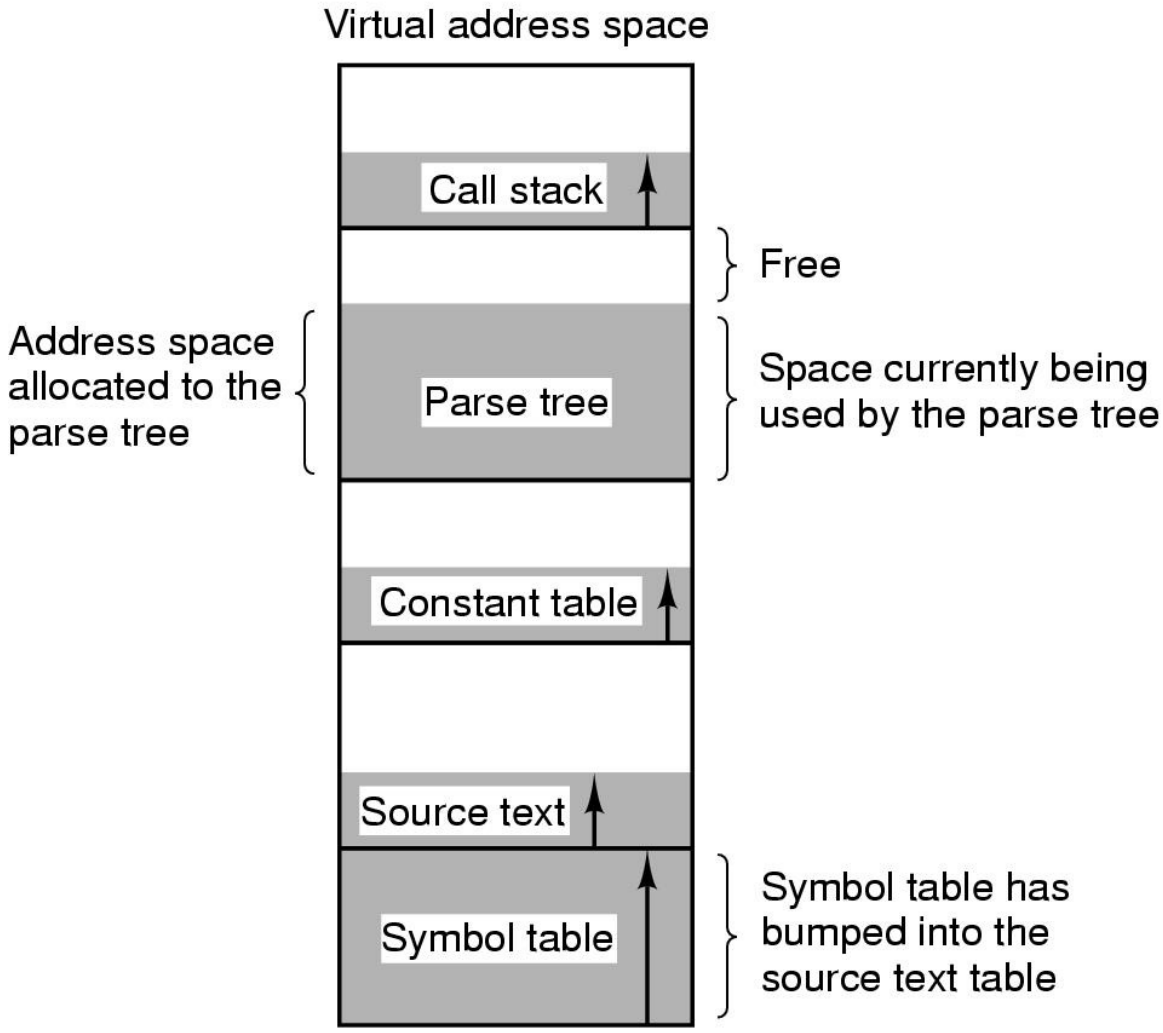
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a minimal allocation unit, but not being used
- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block
  - Issues:
    - overhead
    - problems with programs currently doing I/O

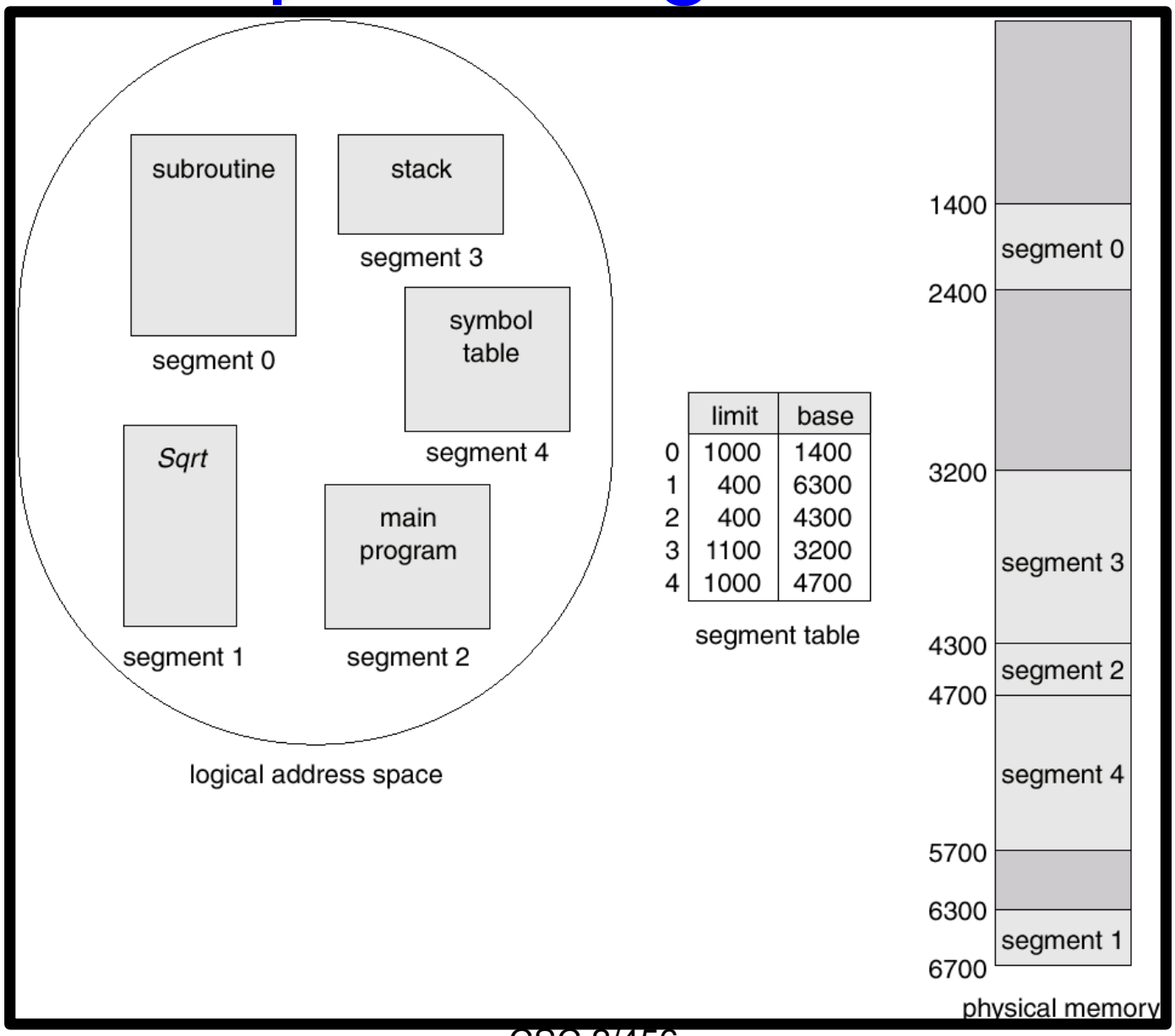
Pure Segmentation

- One-dimensional address space with growing pieces
- At compile time, one table may bump into another

- Segmentation:
  - generate segmented logical address at compile time
  - segmented logical address is translated into physical address at execution time

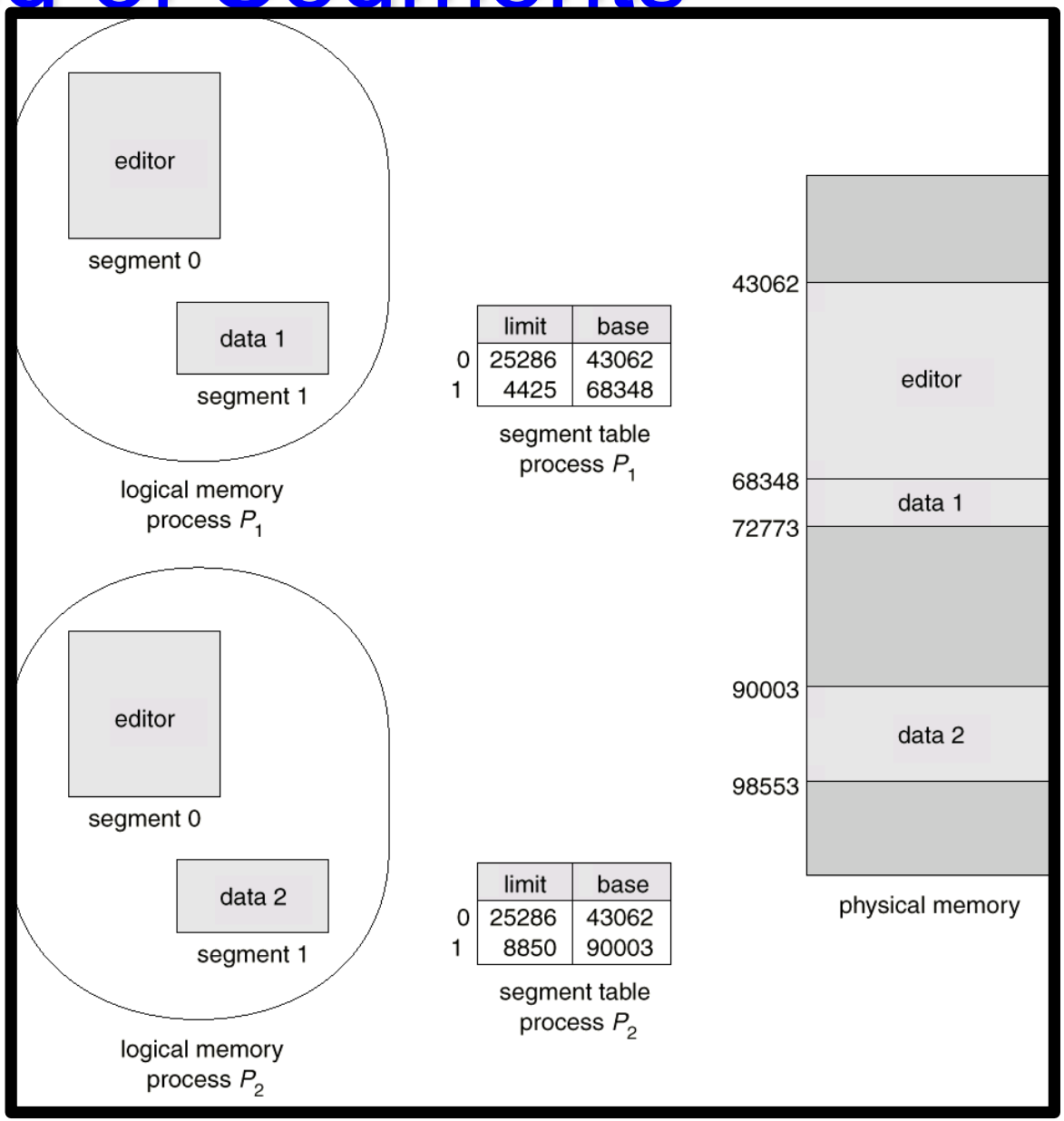


Example of Segmentation



Sharing of Segments

- Convenient sharing of libraries



Segmentation

- Two-dimensional (logical) view of memory
  - Segment (independent address space) + offset
  - Variable length
- Facilitates sharing (e.g., shared libraries)
- Suffers from the external fragmentation problem
- Solution: segmentation with paging
  - E.g., Intel x86
    - Contains 6 segment registers

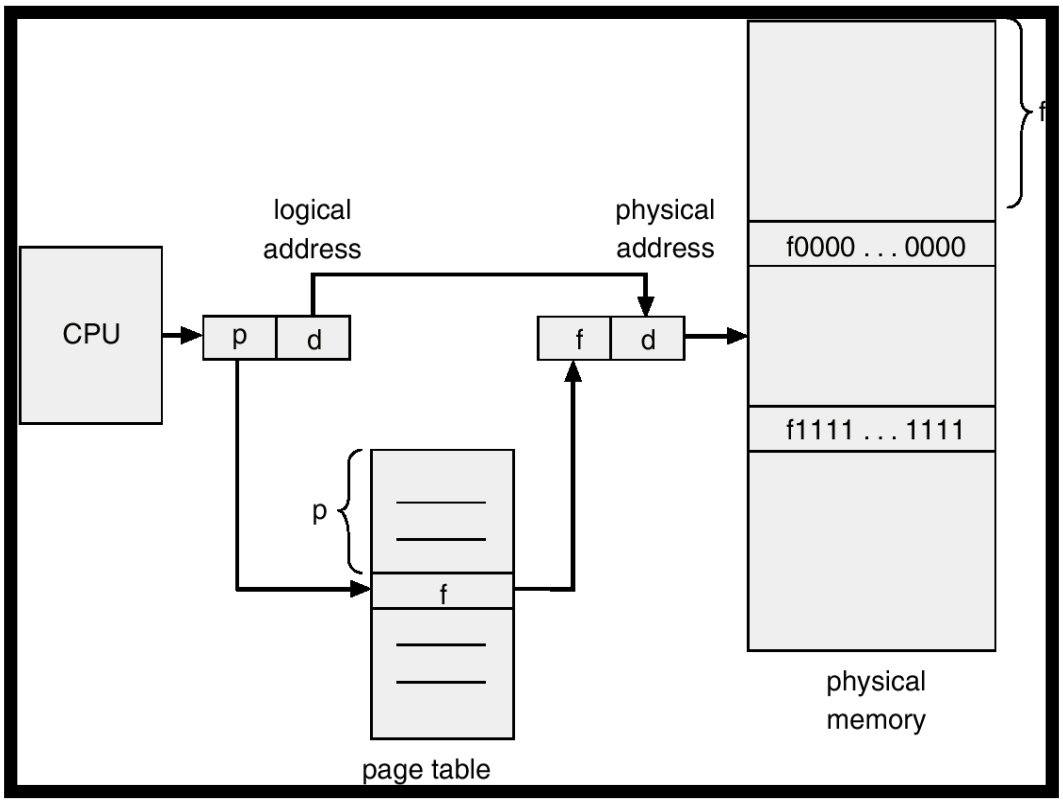
Paging (non-contiguous allocation)

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
- Divide physical memory into fixed-sized blocks called **frames** (typically 4KB)
- Divide logical memory into blocks of same size called **pages**
- To run a program of size n pages, need to find n free frames and load program
- Internal fragmentation

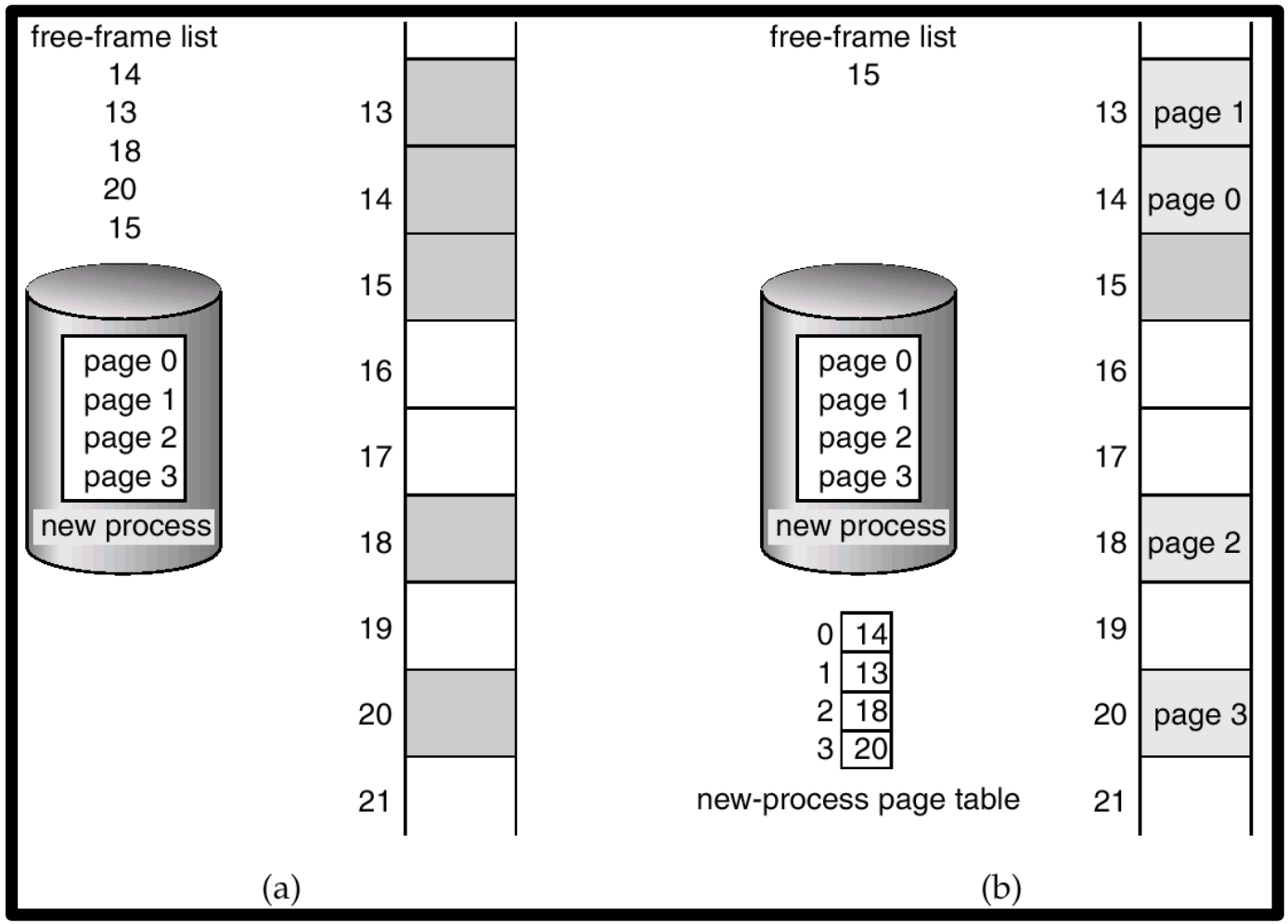
Paging: Address Translation Scheme

A logical address is divided into:

- Page number (p) – used as an index into a page table which contains base address of each page in physical memory
- Page offset (d) – the offset address within each page/ frame. The same for both logical address and physical address



Load A User Program: An Example



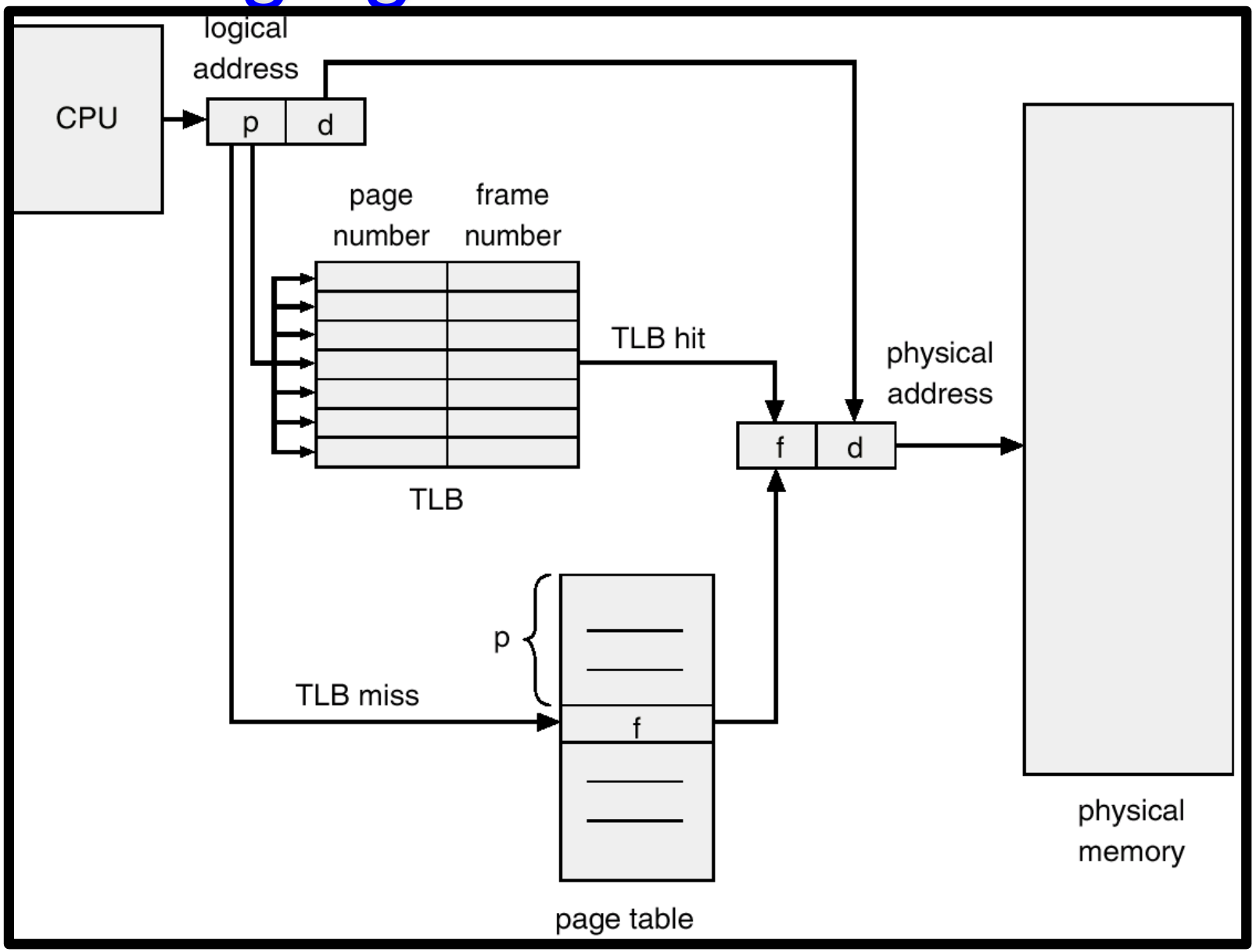
Before loading

After loading

Implementation of Page Table

- Page table is (usually) kept in main memory
  - why not in registers?
  - kernel or user space?
- Hardware MMU:
  - Page-table base register points to the page table
  - Page-table length register indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses: One for the page table and one for the data/instruction
- Solution:
  - A special fast-lookup hardware cache called translation look-aside buffers (TLBs)

Paging MMU With TLB



Page Table Formats

Effective Access Time

- Assume
  - TLB Lookup = 1 ns
  - Memory cycle time is 100 ns
- Hit ratio (α)– percentage of times that a page number is found in the TLB
- Effective memory Access Time (EAT)

$$EAT = 101 \times \alpha + 201 \times (1 - \alpha)$$

Layout of A Page Table Entry

- Physical page frame address
- No logical page number
- Other bits for various page properties

Page Table Structure

- Problem with a flat linear page table
  - assume a page table entry is 4 bytes; page size is 4KB; the 32-bit address space is 4GB large
  - how big is the flat linear page table?

- Solutions:
  - Hierarchical Page Tables
    - break the logical page number into multiple levels

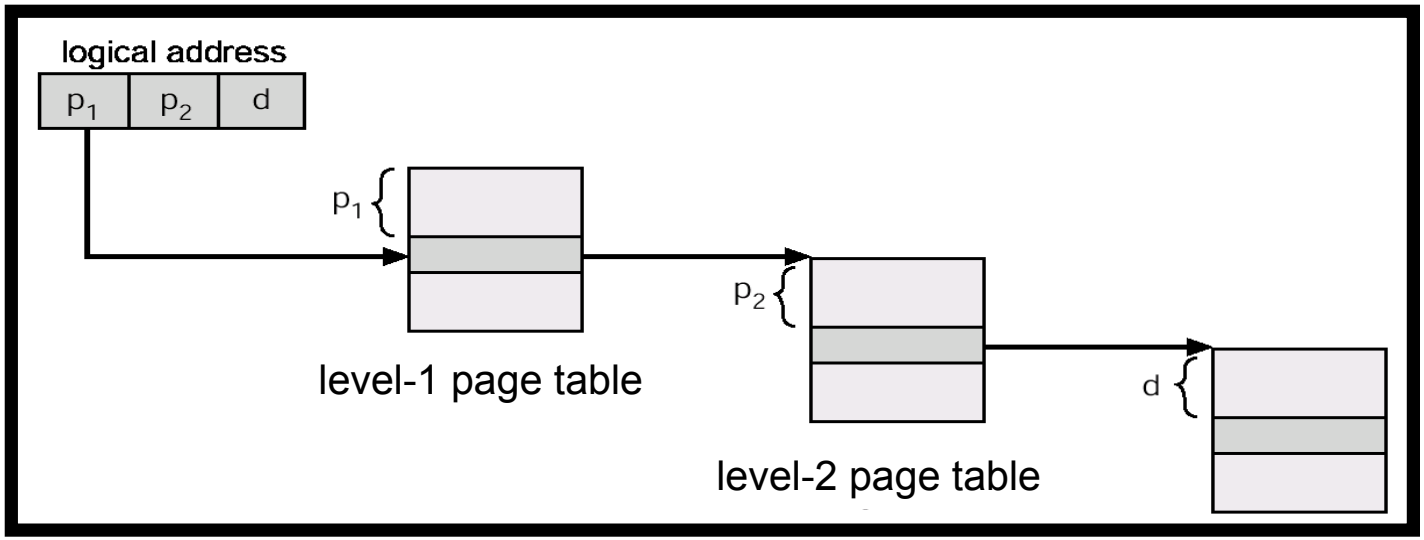
- Metrics:
  - Space consumption and lookup speed

Two-Level Page Table

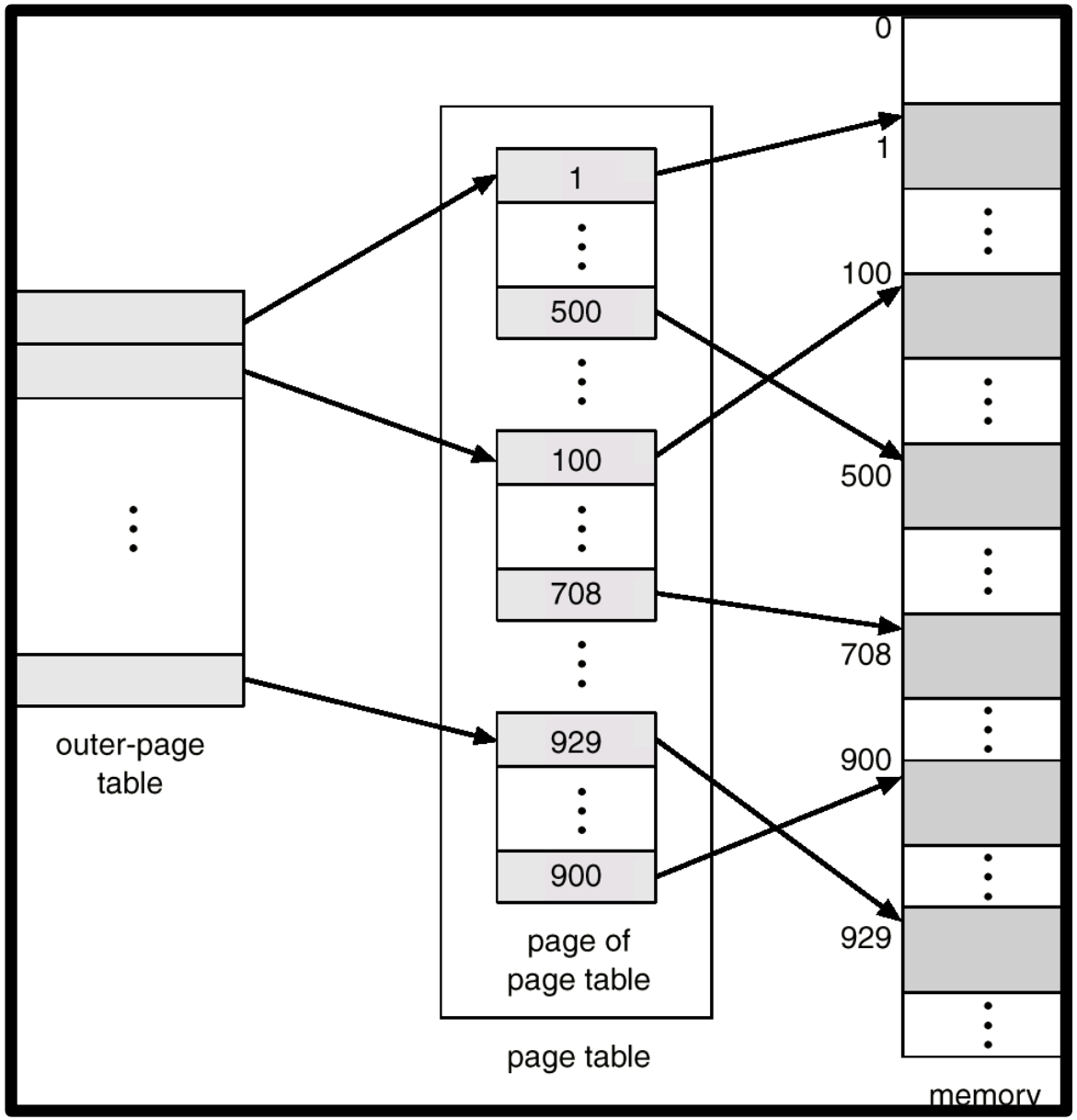
- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page offset consisting of 12 bits.
  - a page number consisting of 20 bits; further divided into:
    - a 10-bit level-2 page number.
    - a 10-bit level-1 page number.
- Thus, a logical address looks like:

page number		page offset
p_i	p_2	d
10	10	12

- Address translation scheme:



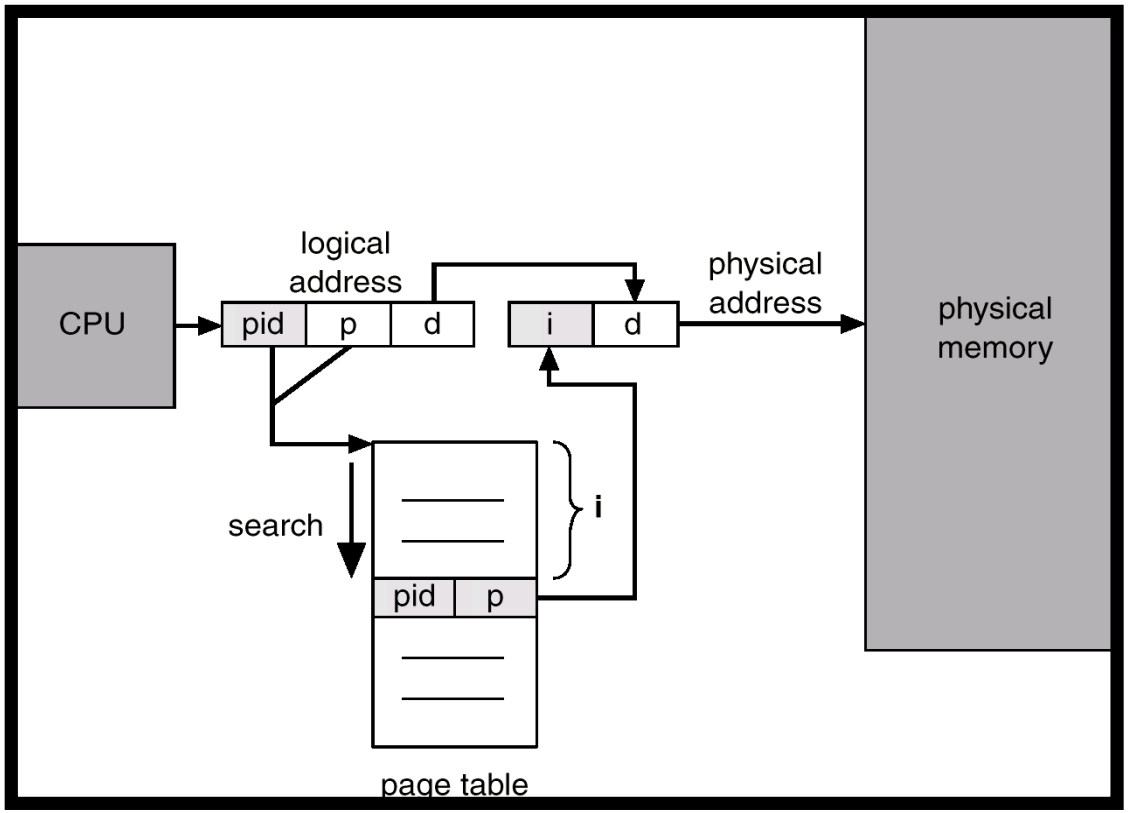
Two-Level Page Table: Example



- Space consumption
- Lookup speed
  - What happens to EAT?

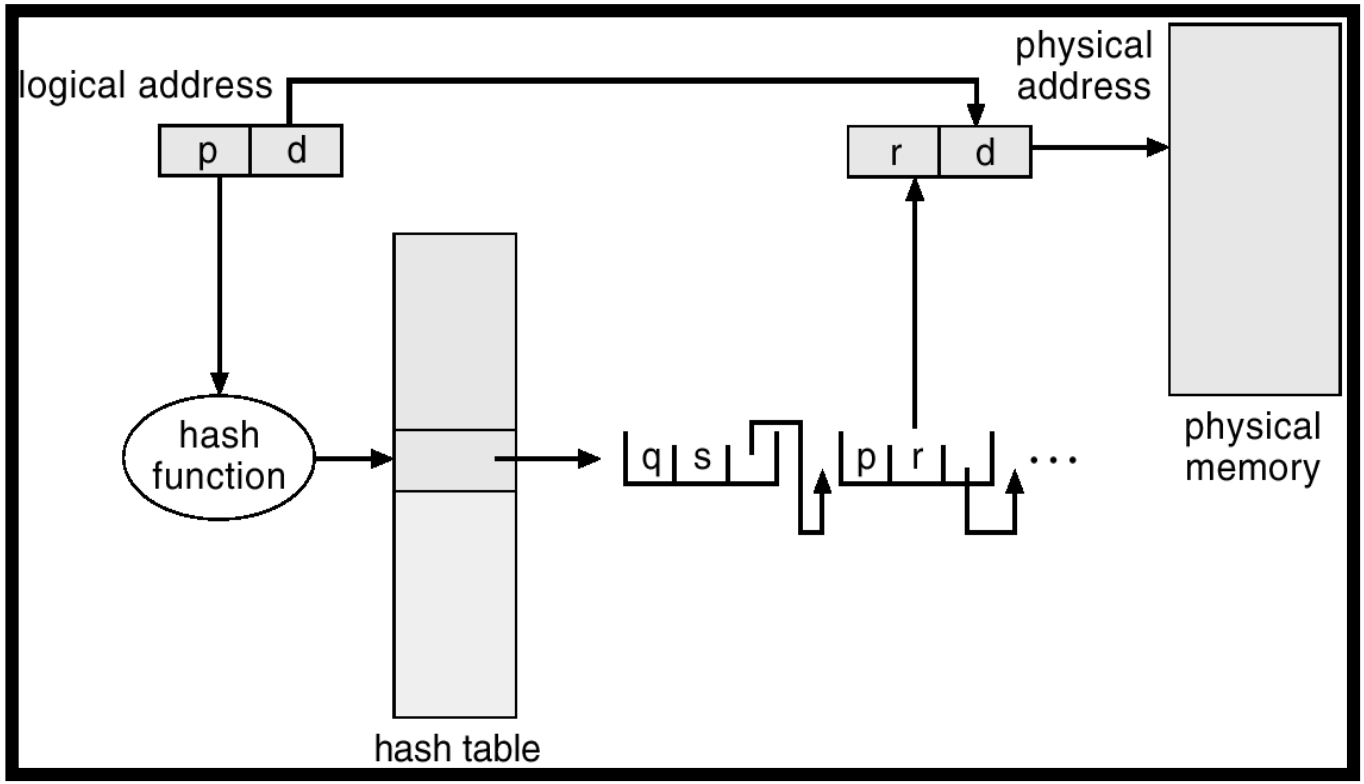
Deal With 64-bit Address Space

- Two-level page tables for 64-bit address space
  - more levels are needed
- Inverted page tables
  - One entry for each real page of memory
  - Entry consists of the process id and virtual address of the page stored in that real memory location
- Problems:
  - search takes too long
  - difficult to share memory



Hashed Page Tables

- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.



Memory Access Setting in Page Table

- Parts of the logical address space may not be mapped
  - Valid-invalid bit attached to each entry in the page table
  - indicating whether the associated page is in the process' logical address space, and is thus a legal page
- Some pages are read-only, or can't contain executable code
  - access bits in page table to reflect these
- Software exception if attempting to access an invalid page, or to perform disallowed actions

Address Space Identifiers (ASID)

- Associate a process ID (or other identifier) with a translation
- Called Address Space Identifier
- Register holds current ASID
- Translation only used if its ASID matches current ASID
- Why add this feature?

Global Bit

- Keeps translation in TLB even on TLB flush
- Special TLB flush instruction/operand flushes global pages
- Why do we have a global bit?

Other Bits

- Present bit
  - Denotes whether a translation is valid
  - Page fault exception when invalid page is accessed
- Read bit
  - Set by hardware when a page is read
- Modified (aka Dirty) bit
  - Set by hardware when a page is written

Page Size Selection

- Issues concerning page size
  - fragmentation
  - page table size
  - TLB reach

- TLB Reach - the amount of memory accessible from the TLB.
  - TLB Reach = (TLB Size) X (Page Size)
- Large TLB reach means fewer TLB misses

- Multiple page sizes:
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Paging Hardware Uses

Demand Paging

- In the old days, swap a process from disk to memory as a whole unit
- Demand paging brings in code/data as needed
- Here's how it works:
  - First access to a page generates a fault
  - Page fault handler finds data that needs to be paged in
  - Data is loaded into page
  - Memory access resumes
- Pre-paging brings in data before it is needed

Process Creation: Copy-on-Write

- Basic idea:
  - fork() semantics says the child process has duplicate copy of the parent's address space
  - child process often calls exec() right after fork()
  - Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory

- Implementation:
  - shared pages are marked readonly after fork()
  - if either process modifies a shared page, a page fault occurs and then the page is copied
  - the other process (who later faults on write) discovers it is the only owner; so it doesn't copy again

Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory

- At page fault:
  - A certain portion of the file is read from the file system into physical memory
  - Subsequent reads/writes to/from the file are like ordinary memory accesses
  - Modified pages are written back to disk

- Simplifies file access by treating file I/O through memory rather than **read()/write()** system calls

Virtual Memory and the Backing Store

Backing Store

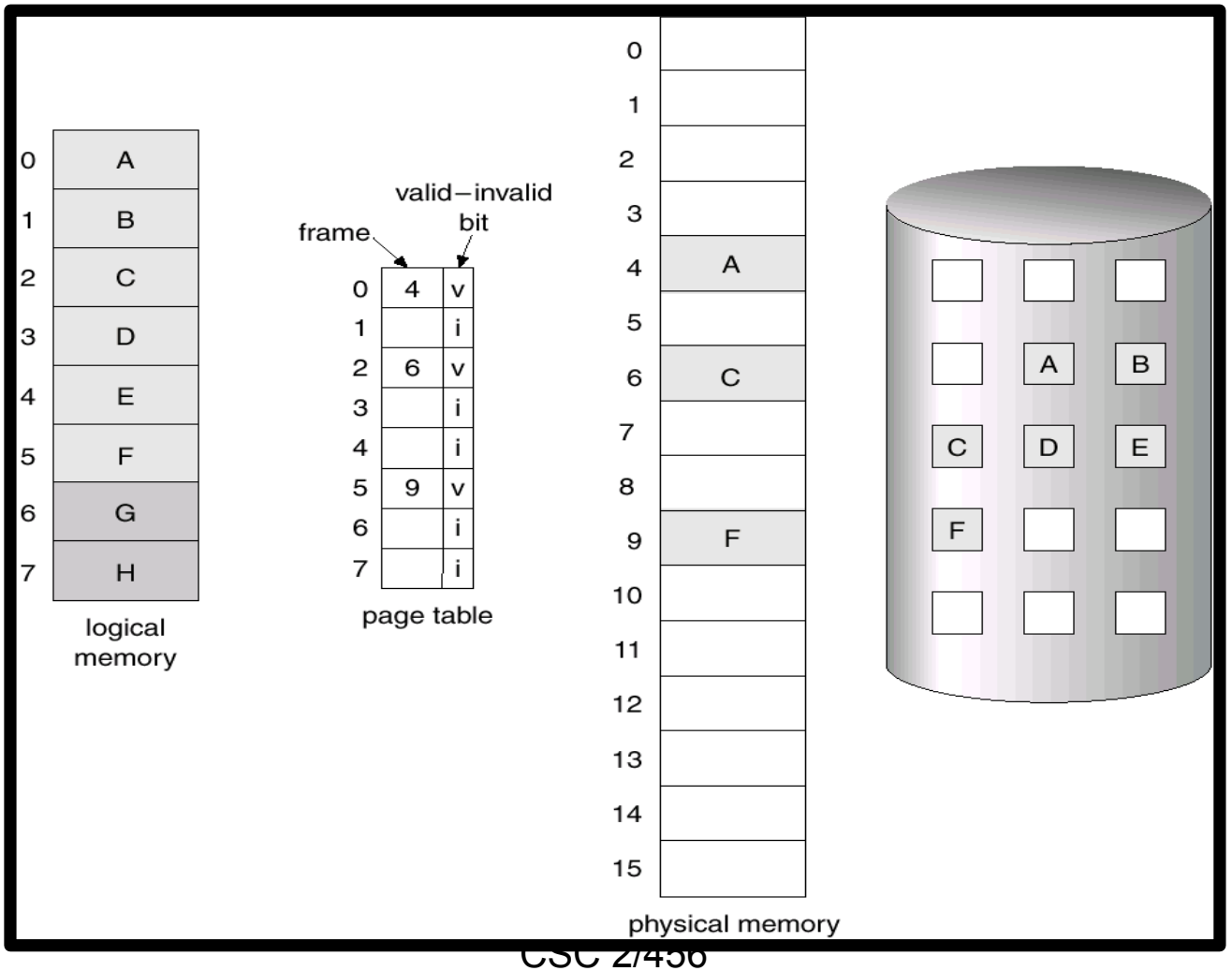
- With virtual memory, the whole address space of each process has a copy in the backing store (i.e., disk)
  - program code, data/stack

- Consider the whole program actually resides on the backing store, only part of it is cached in memory

- With each page table entry, a valid–invalid bit is associated (1 ⇒ in-memory, 0 ⇒ not-in-memory or invalid logical page)

Page Table with Virtual Memory

- With each page table entry a valid-invalid bit is associated (1 ⇒ in-memory, 0 ⇒ not-in-memory or invalid logical page)

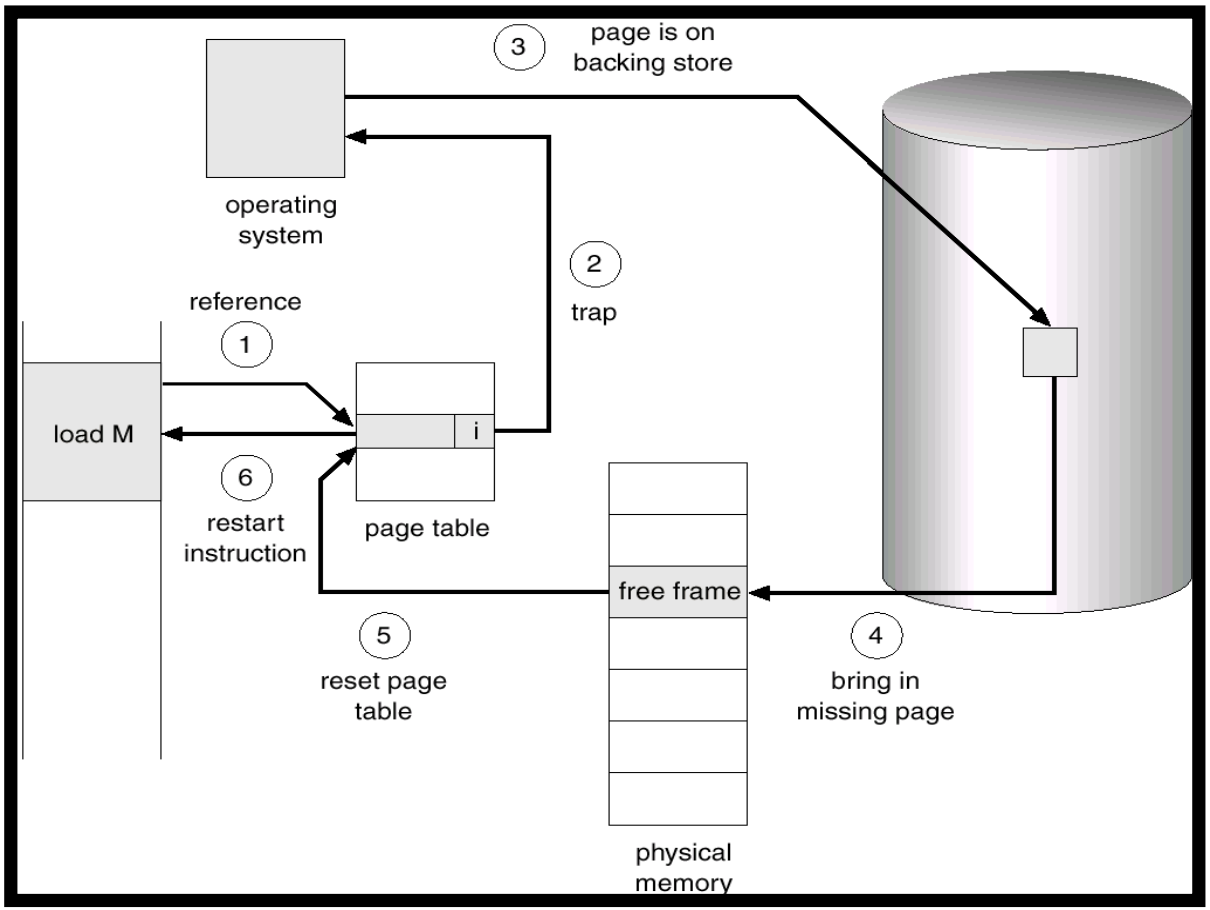


Page Fault

- A reference to a page with the valid bit set to 0 will trap to OS ⇒ page fault

- Invalid logical page:
  - ⇒ abort

- Just not in memory:
  - Get a free frame
  - Swap page into the free frame
  - Reset the page table entry, valid bit = 1
  - Restart the program from the fault instruction.



- What if there is no free frame?

Page Fault Overhead

- Page fault exception handling
- [swap page out]
- swap page in
- restart user program
- memory access

Page Replacement

- Page replacement is necessary when no physical frames are available for demand paging
  - a victim page would be selected and replaced

- A dirty bit for each page
  - indicating if a page has been changed since last time loaded from the backing store
  - indicating whether swap-out is necessary for the victim page
  - How is it maintained? Does it need to be in the page table entry?

Page Replacement Algorithms

- Page replacement algorithm: the algorithm that picks the victim page
- Metrics:
  - low page-fault rate
  - implementation cost/feasibility
- For the page-fault rate:
  - Evaluate an algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time)

1	4	5	
2	1	3	9 page faults
3	2	4	

- 4 frames

1	5	4	
2	1	5	10 page faults
3	2		
4	3		

Wait a minute!
Adding more frames created *more* page faults!

Belady's Anomaly

- more frames do not necessarily lead to less page faults

1	4	5	
2	1	3	9 page faults
3	2	4	

1	5	4	
2	1	5	10 page faults
3	2		
4	3		

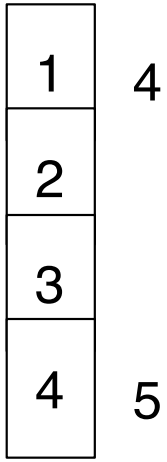
Stack Algorithm

- Stack algorithm: One for which it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n+1 frames
- Does not suffer from Belady's Anomaly

Optimal Algorithm

- Optimal (called OPT or MIN) algorithm:
  - Replace page that will not be used for longest period of time
- 4 frames example

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



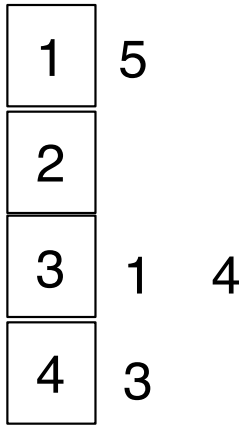
4

5

6 page faults

Least Recently Used (LRU) Algorithm

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



- Not always better than FIFO, but more frames always lead to less or equal page faults
  - imagine a virtual stack (infinite size) of pages
  - each page is moved to the top after being accessed
  - this virtual stack is independent of the number of frames
  - page fault number when there are N frames:
    - the number of accesses that do not hit the top N pages in the virtual stack

Implementations

- FIFO implementation

- Time-of-use LRU implementation:
  - Every page entry has a time-of-use field; every time page is referenced through this entry, copy the clock into the field
  - When a page needs to be changed, look at the time-of-use fields to determine which are to change

- Stack LRU implementation – keep a stack of page numbers in a double link form:
  - Page referenced: move it to the top
  - Always replace at the bottom of the stack

Feasibility of the Implementations

- FIFO implementation
- LRU implementations:
  - Time-of-use implementation
  - Stack implementation

- What needs to be done at each memory reference?
- What needs to be done at page loading or page replacement?

LRU Approximation Algorithms

- LRU approximation with a little help from the hardware.

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced, the bit is set to 1 by the hardware
  - Replace a page whose reference bit is 0 (if one exists). We do not know the order, however
- Second chance
  - Combining the reference bit with FIFO replacement
  - If page to be replaced (in FIFO order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in FIFO order), subject to same rules
  - Also called CLOCK algorithm

LRU Approximation Algorithms

- Enhancing the reference bit algorithm:
  - it would be nice if there is more information about the reference history than a single bit.
  - with some more help from software, e.g., a memory reference counter (in page table entry and TLB)

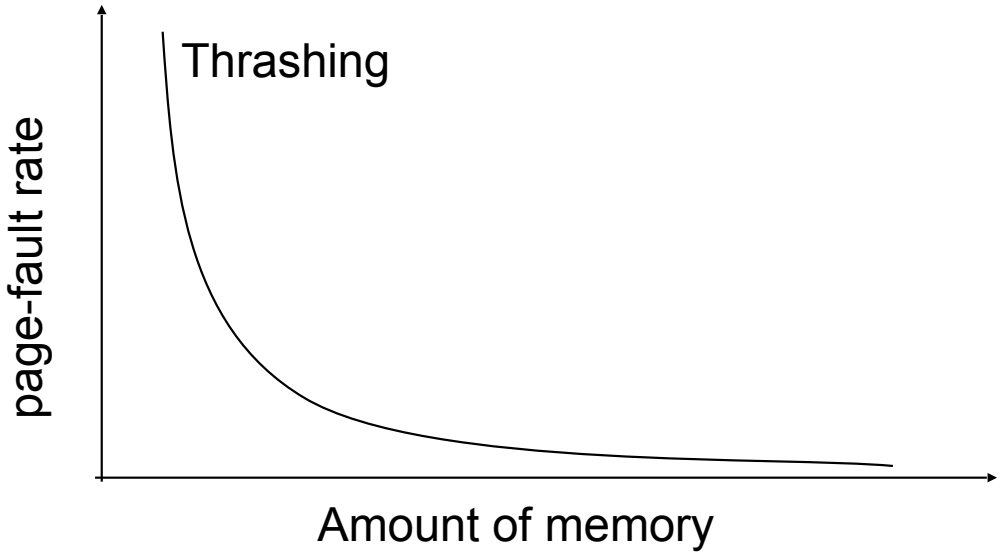
- Maintain more reference bits in software:
  - at every N-th clock interrupt, the OS moves each hardware page reference bit (in page table entry and TLB) into a multi-bit page reference history word (in software-maintained memory).

Counting-based Page Replacement

- Least frequently used page-replacement algorithm
  - the page with smallest access count (within a period of time) is replaced
- Implementation difficulties
  - Requires per-reference count increment

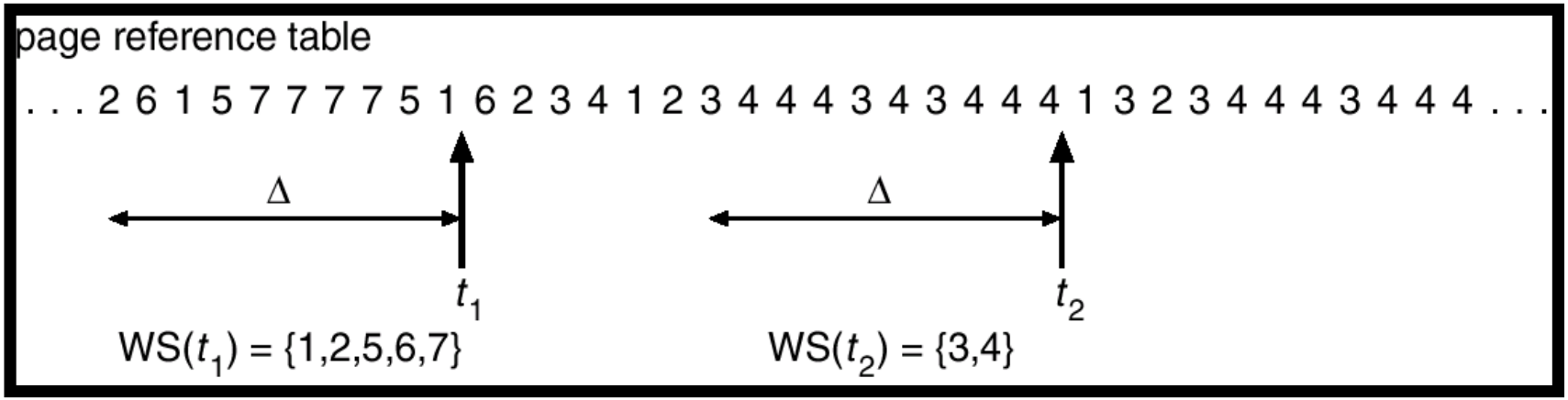
How much memory does a process need?

- Our discussion so far is "Given the amount of memory, what order should we evict pages?"
- Now we look at "How much memory does a process need?"
- If a process does not have "enough" pages, the page-fault rate is very high
  - **Thrashing** ≡ a process is mostly busy with swapping pages



Working-Set Model

- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (working-set window)



- data access locality:
  - working set does not change or changes very slowly over time.
  - so enough memory for the working set should be good.

- How to choose Δ?

Working-Set-Based Memory Allocation

- Two components

- How much memory does a process need?
  - try to allocate enough frames for each process's working set.
  - if $\Sigma WSS_i > m$, then suspend one of the processes.
  - How to determine the working set size over a recent period Δ?

- Given the amount of memory, what order should we evict pages?
  - LRU and augment (WSClock)

Working Set Algorithm

- Define working set based on process's virtual time tau
- For each page, track
  - Time of last use
  - Referenced bit
  - Modified bit
- On page fault:
  - Update time of last use for referenced pages while looking for a page to evict
  - Evict unreferenced pages that are older than tau (prefer Referenced over Modified)
  - Clear Referenced bits

Working Set Clock (WSClock) Algorithm

- Put frames into a circular list
- Cycle through ring of frames on page fault
  - Clean frames older than tau are used for replacement
  - Dirty frames older than tau are scheduled for write-back; marked clean when write-back finishes
- If we get back around:
  - Write-back scheduled: keep cycling until a clean page is found
  - No write-back scheduled: pick a clean page and use it

Pitfall of Working-Set-Based Memory Allocation

- Pitfall:
  - The working set size is not a good indicator of how much memory a process "actually" needs.

- Example:
  - Consider a process that accesses a large amount of data over time but rarely reuses any of them (e.g., sequential scan).
  - It would exhibit a large working set but different memory sizes would not significantly affect its page fault rate.

Other Memory Management Issues

- When to swap out pages?

- Prepaging
  - swap in pages that are expected to be accessed in the future

Kernel Memory Allocation

Kernel Memory Allocation

- Distinguishing features
  - Sometimes require physically contiguous region
  - Usually request memory for data structures of varying size
  - Data structures typically reused
    - task_struct
    - cred_t
  - Need to allocate specific pages for I/O

Kernel Memory Page Allocation

- Buddy system
  - Power-of-2 allocator (Linux kernel originally used this)
  - Advantage: coalescing
  - Drawback: fragmentation
- Some kernel functions use it as a “regular” memory allocator

Kernel Memory Heap Allocator

- Slab allocator
  - Creates a “pool” of objects of the same size
  - Improves spatial locality for cache
  - Improves reuse of memory
- General allocator (kmalloc/kfree)
  - Allocates “one-off” types of memory
  - Backed by slab allocator (one pool for power-of-two size)

Disclaimer

- Parts of the lecture slides contain original work of Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Andrew S. Tanenbaum, and Gary Nutt. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).