# G22.2250-001
# Operating Systems

Lecture 8

Memory Management (cont'd)
Virtual Memory

October 23, 2007

# Outline

- Announcements
  - Lab 3 is past due
- Memory Management
  - Allocation schemes
    - Partitioning
    - Paging
    - Segmentation
  - Examples
- Virtual Memory
  - Introduction
  - Demand paging
  - Page replacement

*[ Silberschatz/Galvin/Gagne: Chapter 8, Sections 9.1 - 9.4 ]*

(Review)

# Memory Mapping Schemes

- Goal: Allocate physical memory to processes
  - translate process logical addresses into physical memory addresses

- Objectives
  - memory protection
    - users from other users, system from users
  - efficient use of memory
  - programmer convenience
    - Relocation, large virtual memory space

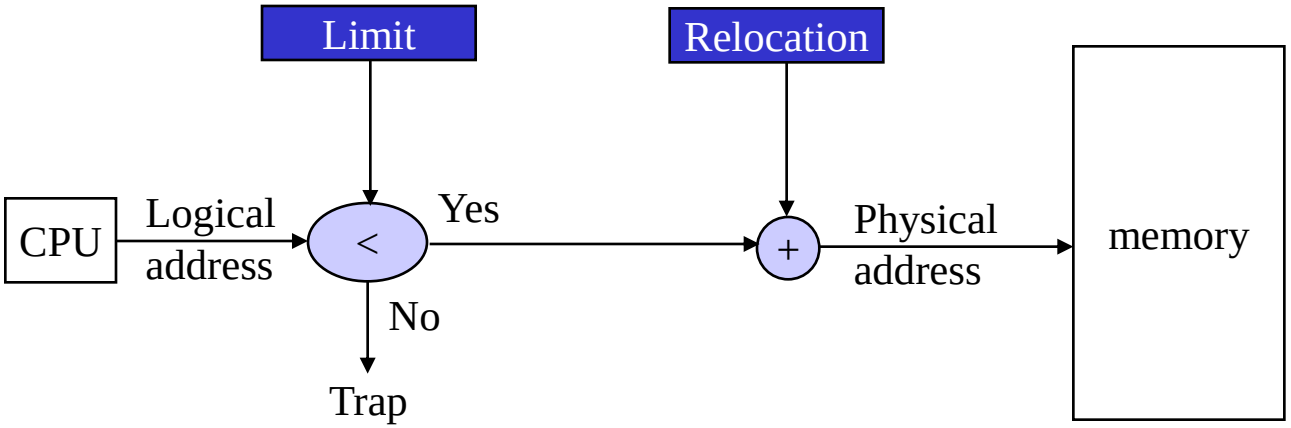
- Three schemes
  - Partitioning
  - Paging
  - Segmentation

(Review)

# Memory Mapping (1): Partitioning

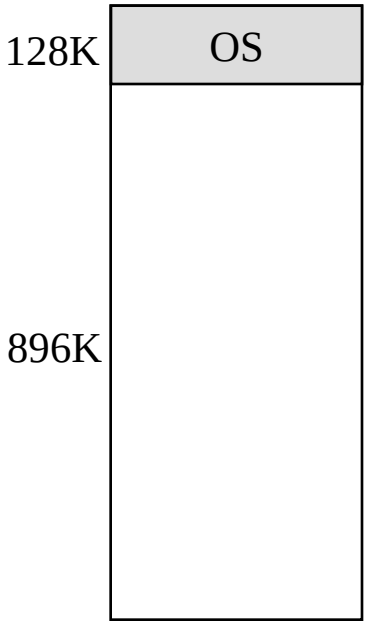
Idea: Divide memory into partitions

- Variable-sized partitions more typical
  - contiguous memory allocated on process loading, released on termination
- Dynamic binding and protection
  - Supported using relocation, limit registers
  - whenever a memory location is accessed:
    - the system computes physical-address = logical-address + relocation register
  - Relocation register can be dynamically reassigned

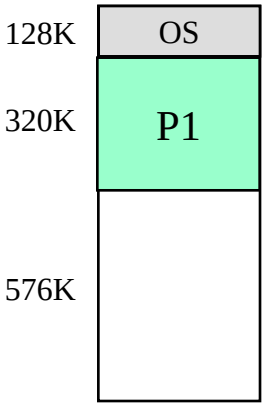


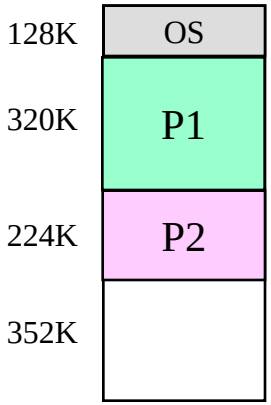
(Review)

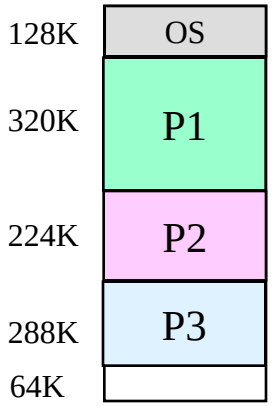
# Memory Allocation and Scheduling

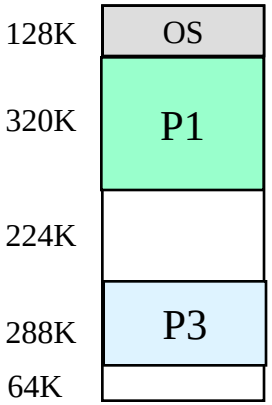



4 Processes: P1 (320K), P2 (224K), P3 (288K), P4 (128K)

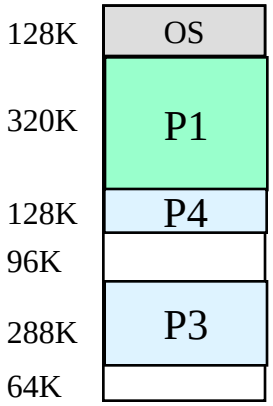


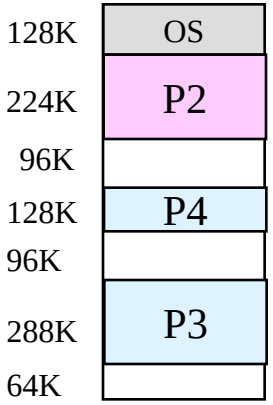




(swap out P2)




P1 ends, swap in P2

# Partitioning Policies

- Memory is viewed as sequence of blocks and voids (holes)
  - blocks are in use
  - voids are available: neighboring voids are coalesced to satisfy request

- Question: Given a request for process memory and list of current voids, how to satisfy the request
  - First fit: allocate space from the first void in the list that is big enough
    - fast and good in terms of storage utilization
  - Best fit: allocate space from a void to leave minimum remaining space
    - very good storage utilization
  - Worst fit: allocate a void such that the remaining space is a maximum
    - requires peculiar memory loads to perform well in terms of storage utilization

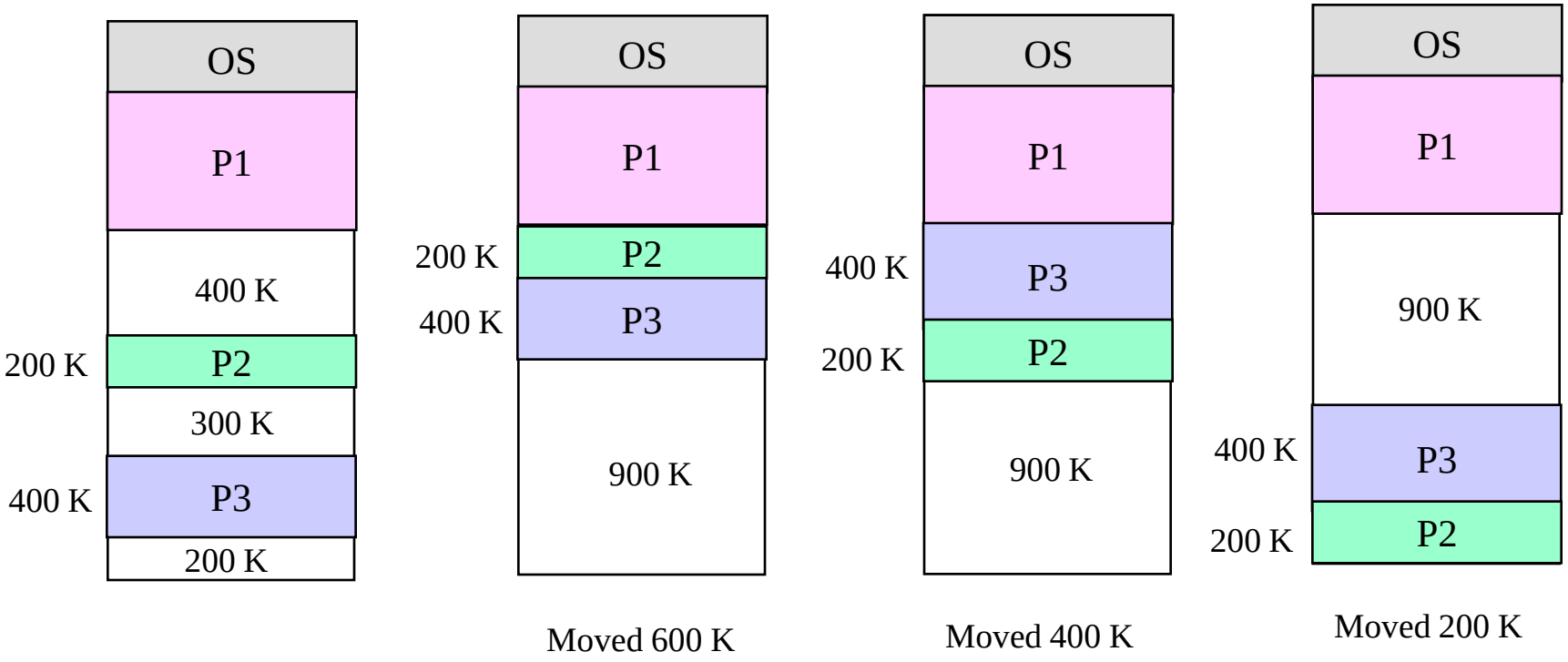
# Partitioning Policies (contd.)

- Criterion for evaluating a policy: Fragmentation
- External fragmentation
  - void space between blocks that does not serve any useful purpose
  - statistical analysis of first-fit:
    - (Knuth) “Fifty-percent rule” Free-list size tends toward ~0.5N blocks
    - (Denning) Fragmentation ends up increasing memory usage by 0.5
    - (Shore) On realistic workloads, free-list and best-fit both perform much better
  - can be avoided by compaction
    - Swap out a partition
    - Swap it back into another part of memory: requires relocation
- Internal fragmentation
  - it is not worth maintaining memory that leaves very small voids (e.g., a few bytes) between used regions
    - occurs more obviously when unit of allocation is large (e.g. disks)
  - Happens when memory request is smaller than the smallest partition size

# Memory Compaction: Reducing Fragmentation

- Moving partitions around can group the voids together
  - increase likelihood of their being used to satisfy a future request

- Many ways of doing this:



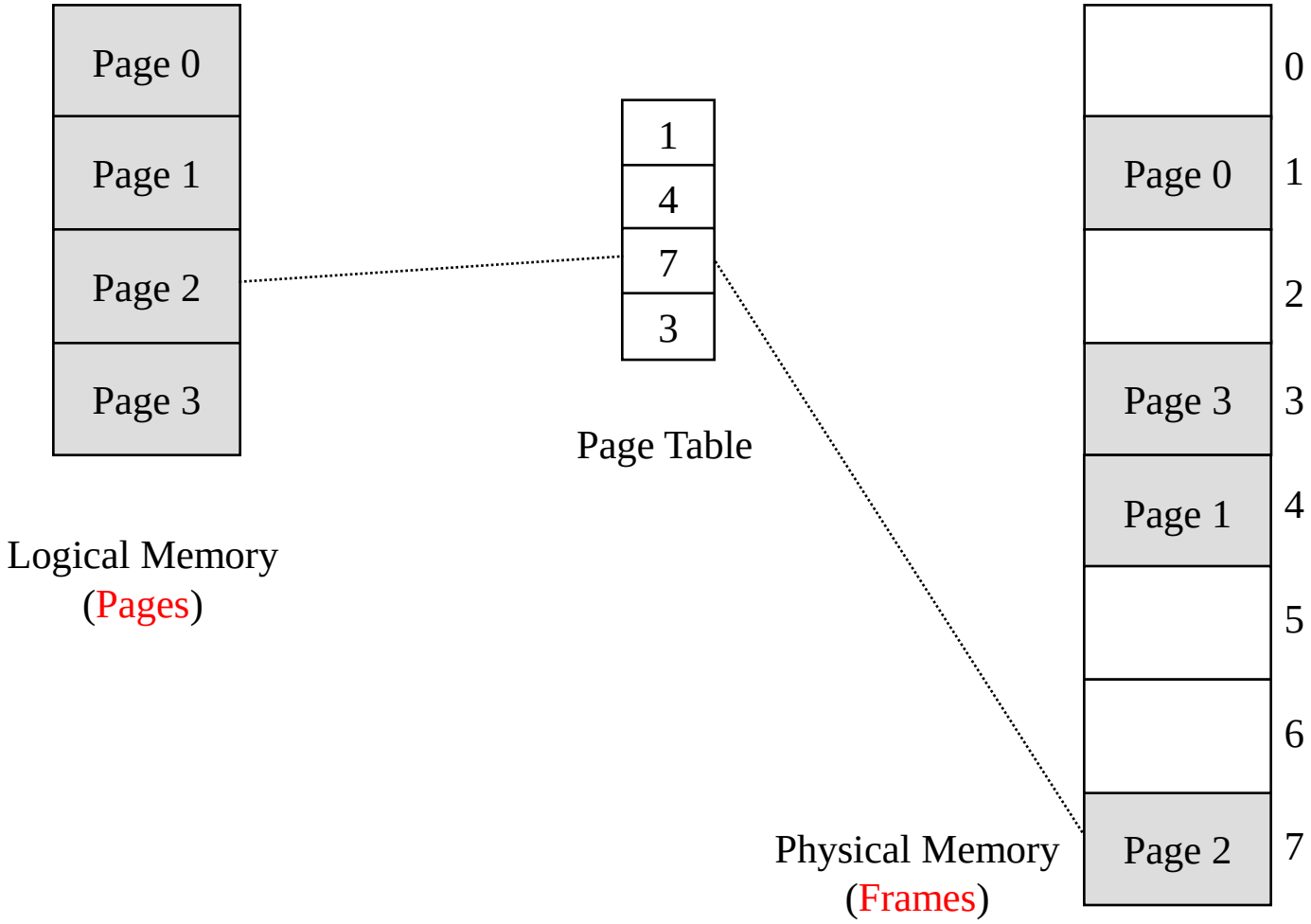
# Memory Mapping (2): Paging

- Motivation: Partitioning suffers from large external fragmentation

Paging

- view physical memory as composed of several fixed-size frames
  - a "frame" is a physical memory allocation unit
- view logical memory as consisting of blocks of the same size: pages
- allocation problem
  - put "pages" into "frames"
    - a page table maintains the mapping
  - allocation need not preserve the contiguity of logical memory
    - e.g., pages 1, 2, 3, 4 can be allocated to frames 3, 7, 9, 14
  - how does this avoid external fragmentation?
- paging played a major role in virtual memory design
  - separation between the meaning of a location in the user's virtual space and its actual physical storage

# Paging (example)

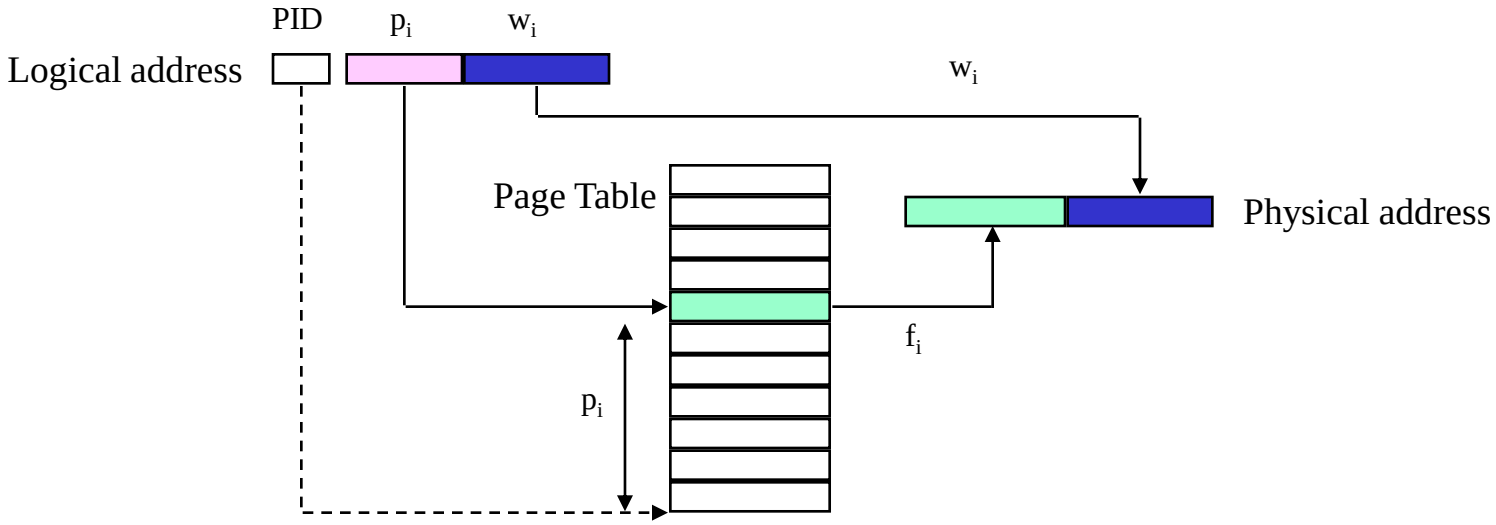


# Paging (cont'd)

- Mapping of pages to frames
  - the mapping is hidden from the user and is controlled via the OS
- Allocation of frames to processes (Nachos Lab 4)
  - the OS maintains a map of the available and allotted frames via a structure called a frame table
    - whether a frame is allocated or not
    - if allocated, to which page of which process
- Address translation
  - performed on every memory access
  - must be performed extremely efficiently so as to not degrade performance
  - typical scheme
    - frames (and pages) are of size $2^k$
    - for each logical address of $a = m + n$ bits
      - the higher order $m$ bits indicate the page number $p_i$ and
      - the remaining $n$ bits indicate the offset $w_i$ into the page

# Page Table Lookup

- Mapping between pages and frames is maintained by a page table
  - the page number $p_i$ is used to index into the $p_i^{th}$ entry of the (process') page table where the corresponding frame number $f_i$ is stored




- All of this requires hardware support
  - since performed on every memory access

# Page Table Structure

- Page table typically stored in memory
  - a single page table base register that
    - points to the beginning of the page table
    - $p_i$ is now the offset into this table
  - problem
    - requires two accesses to memory for each value
    - even with caches, can become very slow

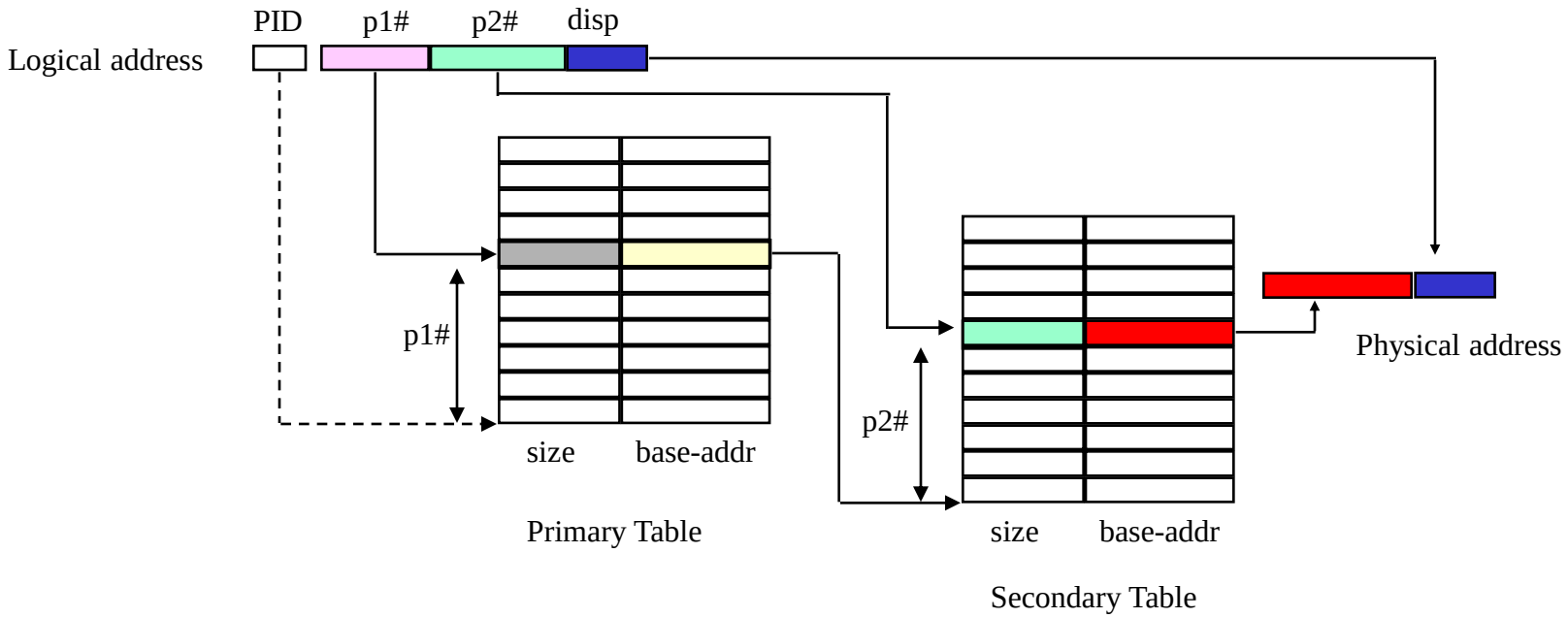
- Solution: Translation Lookaside Buffer (TLB)
  - a portion of the page table is cached in the TLB
    - little performance degradation if a value is a *hit* in the TLB
    - if not: a memory access is needed to load the value into the TLB
      - an existing value must be *flushed* if the TLB is full
  - E.g.: Average memory access time for a system with 90% hit rate in TLB

    $= 0.9*(Access_{TLB} + Access_{mem}) + 0.1*(Access_{mem} + Access_{mem})$

    $=\sim 1.1*(Access_{mem})$

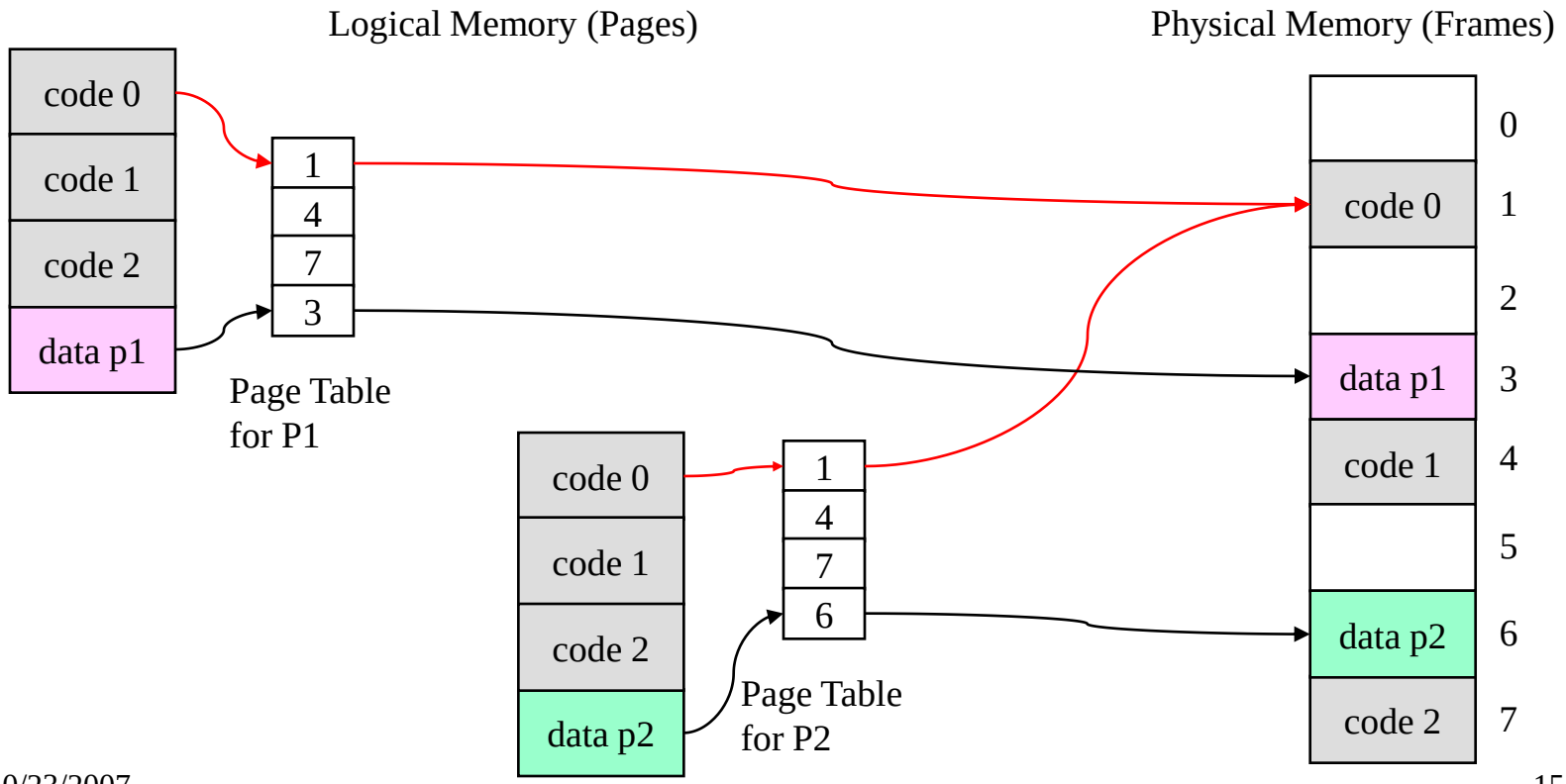
# Multi-level Page Tables

- Rationale: Modern systems support a very large logical address space
  - page tables themselves become very large
    - e.g., for a system with 32-bit logical addresses and 4K pages
      - we need $2^{20}$ page table entries (4 bytes per PTE implies 4 MB of space)
- Solution: page the page table itself
  - cost: additional memory accesses (but caching helps)



# Page Tables and Sharing

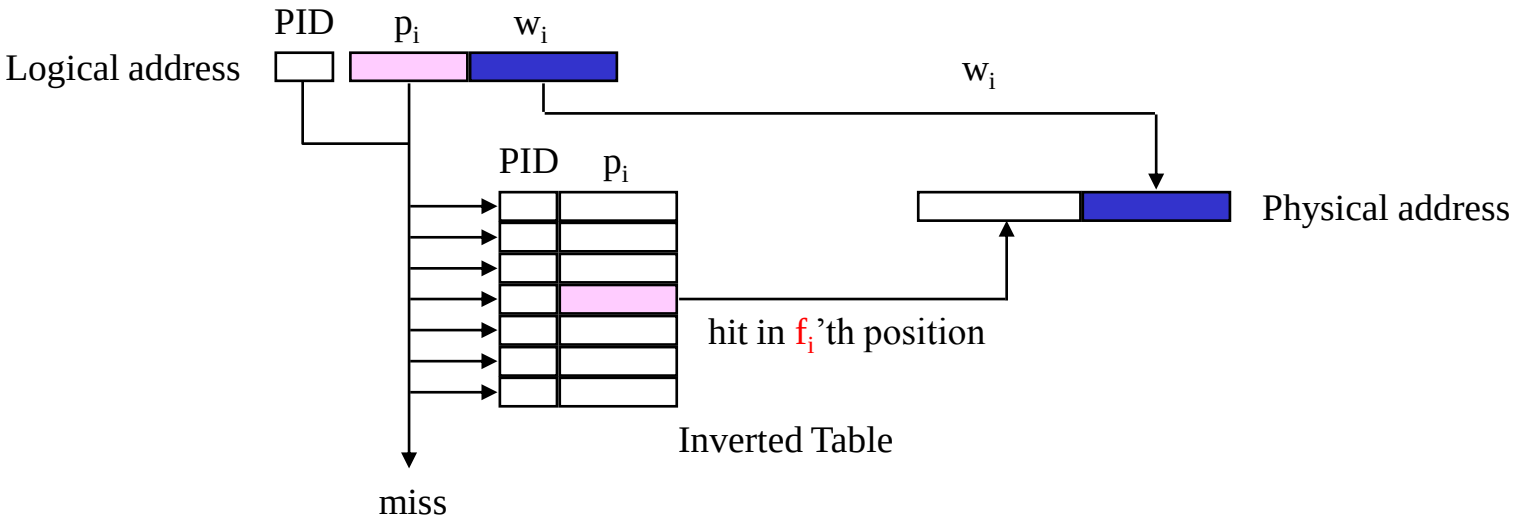
- Page tables permit different virtual addresses (frames of different processes) to map to the same physical address
  - convenient sharing of common code (dynamically-linked system libraries)
  - shared data segments for IPC



# Inverted Page Tables

- Observation
  - usually, only a portion of all the pages from the system's memory can be stored in the physical memory
  - so while the required page table for all of logical memory might be massive, only a small subset of it contains useful mappings

- We can take advantage of this fact in both TLB and page table design



# Inverted Page Tables (cont'd)

- Efficiency considerations
  - the inverted page table is organized based on physical addresses via frame numbers
    - searching for the frame number can be very slow
  - use a hash table based on
    - the PID and logical page number as keys
  - recently located entries of the inverted page table can be stored in a TLB-like structure based on associative registers

- Main disadvantage of inverted page tables: sharing
  - each process that shares an object will have its own (disjoint) space where the shared object is mapped
  - not possible to maintain with standard inverted page tables
    - since space for only one <PID, page number> tuple

# Protection Issues with Paging

- Partition protection scheme
  - Check that address lies between base and base+limit
  - Cannot be used on page-based systems: WHY?

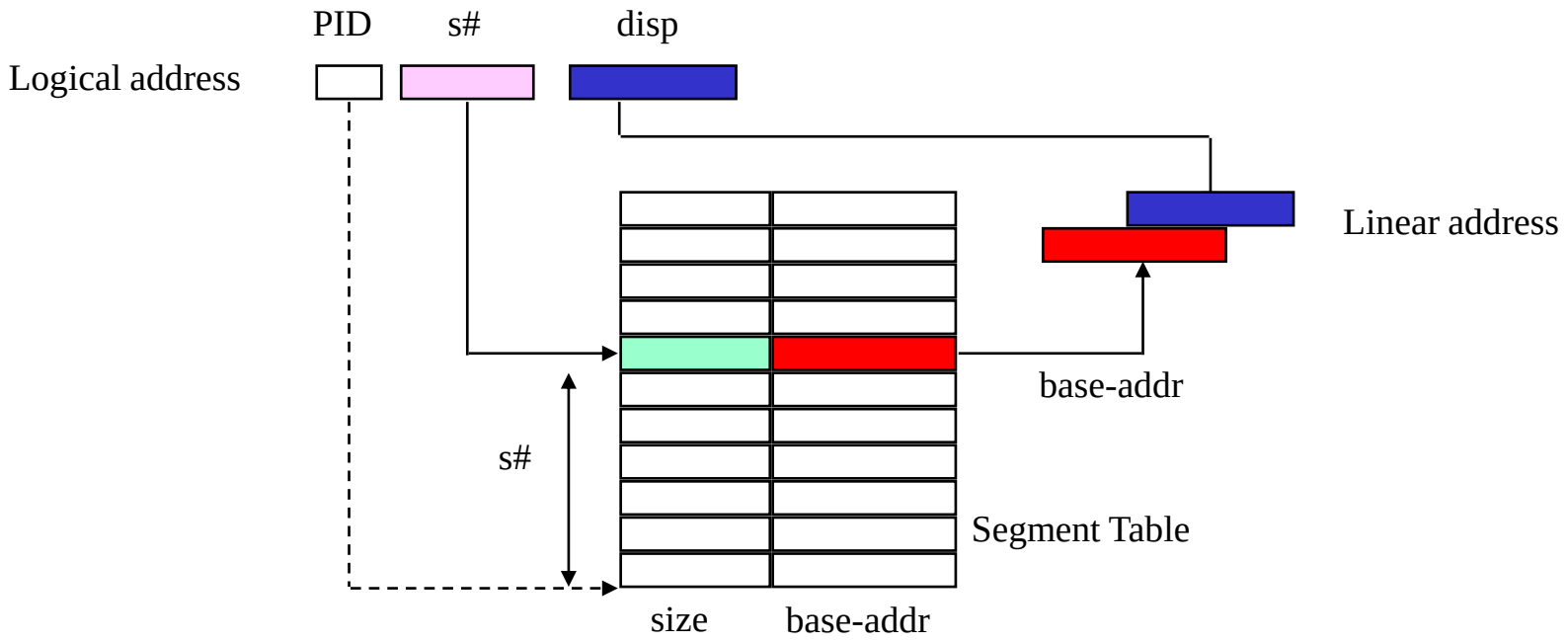
- Physical memory can only be accessed through page table mappings
  - all addresses are interpreted by the MMU
  - OS intervention required to manipulate page tables and TLBs
- Special bits in the page table entry enforce per-frame protection
  - an accessibility bit
    - whether a page is invalid, readable, writable, executable
  - a valid/invalid bit to indicate whether a page is in the user's (logical) space
- Sometimes, the hardware may support a page-table length register
  - specifies size of the process page table
    - trailing invalid pages can be eliminated
    - useful when processes are using a small fraction of available address space

# Memory Mapping (3): Segmentation

- A segment is a logical piece of the program
  - e.g., the code for the program functions, its data structures, symbol tables
- Segmentation views logical memory as broken into such segments
  - segments are of variable size (unlike pages)
- Accessing a segment
  - the logical address is regarded as two-dimensional
    - a segment pointer to an entry in the segment table
    - a displacement into the segment itself
- Allocating a segment
  - a segment is a partition with a single base-limit pair
    - the limit attribute stores the segment length
      - prevents programs from accessing locations outside the segment space
  - differs from partitioning in that there can be multiple segments/process

# Memory Mapping: Segment Table Lookup

- Mapping logical addresses to physical addresses
  - the mapping is maintained by the segment table
  - the segment number s# is used to index into the (process') segment table where the corresponding segment size and base address are stored



# Memory Mapping: Segmentation Hardware

- Segment registers
  - some designs (e.g. Intel x86) provide registers to identify segments
    - loading a segment register loads a (hidden) segment specification register from the segment table
    - construction of the logical address is done explicitly
- TLBs
  - some designs, such as the MIPS 2000, only provide a TLB
    - the OS is responsible for loading this, and doing appropriate translation
- Traditional approach: Store the segment table in memory
  - segment table base register (STBR), segment table length register (STLR)
    - saved and restored on each context switch
  - translation of address (s,d)
    - check that s is valid: s < STLR
    - Look up base address, limit: segment table entry at address (STBR + s)
    - check that offset d is valid: d < length
    - compute physical address

# Segmentation: Pros and Cons

- Pros
  - protection in terms of ensuring that illegal address accesses are avoided, comes for free
    - the segment length check plays an important role here
  - sharing segments across programs is straightforward by loading identical segment table base register values
    - Caveat: How do instructions refer to addresses within segments?
      - Relative addressing works well with sharing
      - Absolute addressing does not: requires same segment number

- Cons
  - external fragmentation is potentially a big problem
  - contrast this with paging where only internal fragmentation is possible

# Memory Mapping: Segmentation and Paging

- Overlay a segmentation scheme on a paging environment
  - several examples
    - originally proposed for GE 645 / Multics
    - Intel x86 uses segment registers to generate 32-bit logical addresses, which are translated to physical addresses by an optional multi-level paging scheme
  - alleviates the problem of external fragmentation

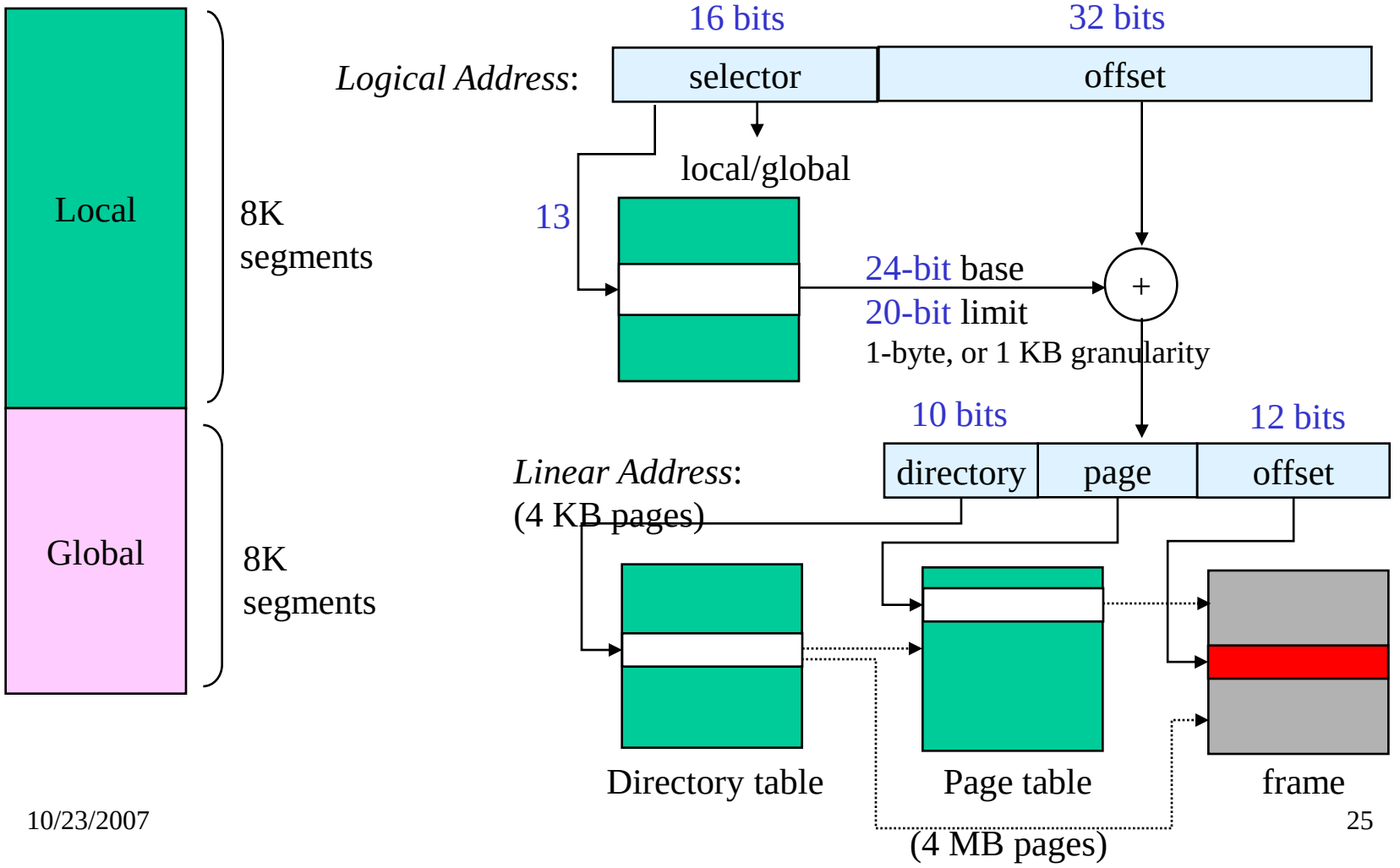
# Memory Mapping: Examples

Multics (c. 1965)

- 34-bit logical address
  - 18-bit segment number, 16-bit offset
  - [8-bit major segment,10-bit minor segment], [6-bit page, 10-bit offset]
  - Both the segment table and segment itself are paged!
- Segmentation structure
  - Segment table is paged
  - major segment number indexes page table for segment table
  - minor segment number is offset within the page of the segment table
    - this gives the page table of the desired segment and the segment length
- Paging structure
  - one-level page table, 1KB pages
- TLB
  - 16 entries; key = 24-bit (seg# & page#); value = frame#

# Memory Mapping: Examples (cont'd)

- Pentium segmentation and paging



# Pentium Memory Mapping (cont'd)

- Very flexible addressing scheme
  - pure paging
    - All segment registers set up with the same selector
      - Descriptor for this selector has base = 0, limit = MAXVAL
    - Offset becomes the address
  - pure segmentation
    - How can this be done?
  - options in between

- See Section 8.7.3 for a description of how Linux uses the Pentium memory addressing support

# Memory Mapping: Summary

- Partitioning: Process is allocated a single contiguous region of memory
  - Translation and protection using size, limit registers
  - Suffers from external fragmentation

- Paging: Process pages are mapped into memory frames
  - Translation using per-process page table (TLBs cache translations)
    - Sharing possible by having multiple pages point to same frame
  - Protection because page-table mappings controlled by OS, extra bits ensure page being accessed in a valid fashion (e.g., read-only)
  - Internal fragmentation possible, but no external fragmentation

- Segmentation: Process is allocated multiple regions, one per segment
  - Translation and protection using size, limit registers
  - Sharing enabled by associating segment descriptors with same information
  - Suffers from external fragmentation, but this has smaller impact

# Outline

- Announcements
  - Lab 3 is past due
- Memory Management
  - Allocation schemes
    - Partitioning
    - Paging
    - Segmentation
  - Examples
- Virtual Memory
  - Introduction
  - Demand paging
  - Page replacement

*[ Silberschatz/Galvin/Gagne: Chapter 8, Sections 9.1 - 9.4 ]*

# Virtual Memory

- Key ideas
  - Separation of logical and physical address spaces
  - Automatic memory mapping mechanisms which support
    - A large logical address space (bigger than physical memory)
    - On-demand movement of program components between the disk and memory (performed transparently by the OS using hardware support)
    - Demand paging + page replacement + frame allocation
- Potential advantages
  - The programmer
    - Is not constrained by limitations of actual physical memory
    - Gets a clean abstraction of storage without having to worry about cumbersome attributes of the execution environment
      - Overlays, dynamic loading, disk transfers, etc.
  - The system
    - Benefits from a higher degree of multiprogramming
      - And hence utilization, throughput, …

# VM Support (1): Demand Paging

- Key mechanism for supporting virtual memory
  - Paging-based, but similar scheme can also be developed for segments
- The idea
  - Allocate (physical) frames only for the (logical) pages being used
  - Some parts of the storage reside in memory and the rest on disk
    - For now, ignore how we choose which pages reside where
- Strategy
  - Allocate frames to pages only when accessed
    - A lazy approach to page allocation
  - Deallocate frames when not used
- Implementation (must be completely transparent to the program)
  - Identifying an absent page
  - Invoking an OS action upon accesses to such pages
    - To bring in the page

# Demand Paging: Identifying Absent Pages

- Goal: Determine when a page is not present in physical memory
- Extend the interpretation of valid/invalid bits in a page-table entry
  - *valid*: the page being accessed is in the logical address space and is present in a (physical) frame
  - *invalid*: the page being accessed is either not in the logical address space or is currently not in active (physical) memory
    - An additional check (of the protection bits) is required to resolve these choices
- The (hardware) memory mapping mechanism
  1. Detects accesses to pages marked invalid
     - Runs on each memory access: instruction fetch, loads, stores
  2. Causes a trap to the OS: a page fault
     - As part of the trap processing, the OS loads the accessed page

# What Happens on a Page Fault?

On a page fault, the OS

1. Determines if the address is legal
   - Details are maintained in the PCB regarding address ranges
2. If illegal, "informs" the program (in Unix: a "signal")
3. Otherwise, allocates a frame
   - May involve "stealing" a frame from another page
4. Reads the requested page into the frame
   - Involves a disk operation
   - CPU can be context-switched to another process
5. Updates the page table
   - Frame information
6. Resumes the process
   - Re-executes the instruction causing the trap

# Interrupting and Restarting

- Must make sure that it is possible to redo the side-effects of an instruction
  - Requires hardware support for precise exceptions
  - Note that page faults are only detected during instruction execution
    - An instruction can cause multiple page faults
- Some subtleties
  - Some architectures support primitive "block copying" instructions
    - Consider what happens if there is a page fault during the copy
    - Need to handle the situation where source and destination blocks overlap
  - What does it mean for the instruction to restart?
- See text book for other pathological cases that must be handled

# Uses of Demand Paging

- Process creation
  - Load executable from disk on demand
  - UNIX fork semantics: child process gets a copy of parent address space
    - fork often followed by exec: explicit copying is wasteful
    - Demand-paging + page-protection bits enable copy-on-write
      - Child gets copy of parent's page table, with every page tagged read-only
      - When a write is attempted to this page, trap to the OS
        - » Allocate frame to hold (child's copy of) the page, copy contents, permit write
- Process execution
  - Frames occupied by unused data structures will eventually be reclaimed
    - Available for use by this and other processes
  - memcpy optimization: uses copy-on-write technique above
- Efficient I/O (Memory-mapped I/O)
  - Map files to virtual memory
  - Disk operations only initiated for accessed portions of the file

# Cost of Demand Paging

- The cost of accessing memory
  - effective access time = *(1 – p).ma + p.pa*
  - where
    - *ma* is the memory access time when there is no page fault
    - *pf* is the page fault time
    - *p* is the probability of a page fault occurring
  - typical values
    - *p* is usually estimated empirically (and grossly) for the system
    - *ma* is 5-6 orders of magnitude smaller than *pf* (order of tens of milliseconds)

- disk access time
  +
- trapping the OS and saving user state
- checking legality of page reference
- context switch
- when disk read is complete, interrupt existing user and save state
- updating page table
- restarting interrupted user process

# Controlling Demand Paging Costs

Three degrees of freedom

- Program structure
  - Selection of data structures and programming structures

```
var A: array [1..128] of array [1..128] of integer;
```

```
for j := 1 to 128                    for k := 1 to 128
  for k := 1 to 128                    for j := 1 to 128
    A[k][j] := 0;                        A[k][j] := 0;
```

- Page replacement
  - Given an allocation of frames to a process, how are these frames managed?
  - Algorithm must ensure that pages likely to be accessed are in memory
- Frame allocation
  - More frames allocated to a process → fewer page faults
  - How should the OS allocate frames to processes?

# VM Support (2): Page Replacement

- In a fully-loaded system, all frames would be in use

- In general, page allocation involves
  - Selecting a page to "evict"
  - Writing it to disk (if it was modified)
  - Reading the new page from disk

- Objectives of page replacement/eviction policy
  - Remove a page with the least overall impact on system performance
    - (from the process' perspective)
      Minimize number of page faults
    - (from the system's perspective)
      Minimize disk activity

# Page Replacement Algorithms: Components

- Reference strings: the sequence of page numbers being accessed
  - Example
    - A logical address sequence 0400, 0612, 0235, 0811, …
    - Will yield the reference string 4, 6, 2, 8, … (for 100-byte pages)
- Hardware support
  - Extra bits associated with the frames to store information about page use
    - Different from the bits stored in each page table entry
  - Commonly available: a page-referenced bit and a page-modified bit
  - Restriction: Must incur very low overhead to maintain
    - Potentially updated on every memory access
- Algorithms
  - FIFO algorithms
  - OPT (Clairvoyant) scheme
  - LRU algorithms and approximations

# Page Replacement: FIFO

- Evict the page that was brought in the earliest

- Pro: Simple to implement
  - OS can maintain a FIFO queue and evict the one at the beginning
- Con: Assumes that a page brought in a long time ago has low utility
  - Obviously not true in general (e.g., much-used library routines)

- How does FIFO perform?
  - Consider reference string (length 12)

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

(with 3 frames) ↑ ↑ ↑ ↑1 ↑2 ↑3 ↑4 ↑1 ↑2 (9)

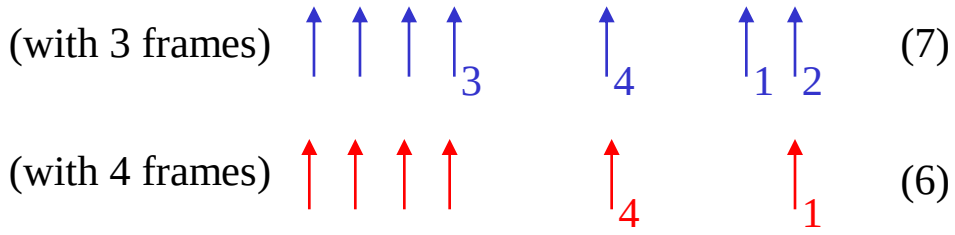
(with 4 frames) ↑ ↑ ↑ ↑ ↑1 ↑2 ↑3 ↑4 ↑5 ↑1 (10)

***Belady's anomaly***
Algorithms that don't exhibit this behavior are known as stack algorithms

# Page Replacement: What is the Best Algorithm?

- For read-only pages (discounting clean-page preference issues), it can be proven that the optimal algorithm (OPT) is
  - Replace the page whose next use is the farthest

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

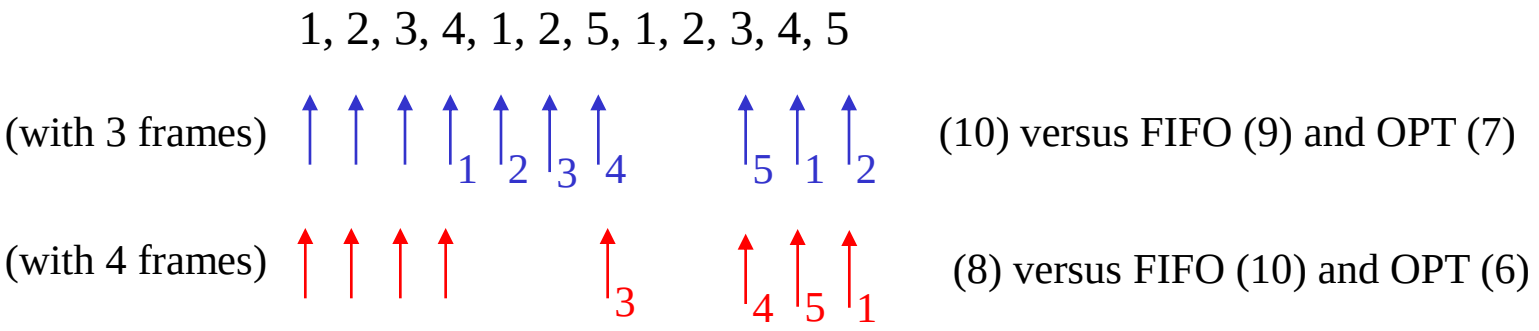



- Optimality stems from the fact that
  - The page replaced will cause a page fault far away
  - Any other page will cause a fault at least as quickly

- How do you prove that OPT does not suffer from Belady's anomaly?

# Page Replacement: LRU

- Problem with OPT: Clairvoyance is generally not possible
  - But sometimes possible to analyze deterministic algorithms
  - In any case, a good baseline to compare other policies against
- LRU (least recently used) is a good approximation of OPT
  - Assumes that recent past behavior is indicative of near future behavior
    - A phenomenon called locality which is exploited repeatedly in virtual memory
- Main idea: Evict the page that has not been used for the longest time

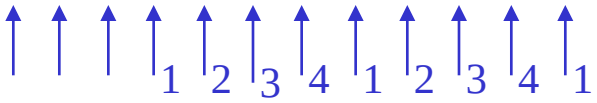


# Page Replacement: LRU (cont'd)

- LRU works reasonably well in simulations
  - "real" program traces exhibit locality
  - but, some pathological access patterns

1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, …

(with 3 frames)




- Main problem with LRU: How does one maintain an active "history" of page usage?
  - Counters
  - Stack

# Page Replacement: Implementing LRU

- Counters
  - Attach to each frame, a counter that serves as a logical clock
    - Updated by the hardware on every reference
  - Page replacement: choose page in frame with smallest counter value
    - Counter is reset when a new page is loaded
  - Problems: Elaborate hardware, Search time
  - Largely of theoretical value

- Stack
  - Maintain a stack of page numbers
    - On each access, hardware moves the page# to the top of the stack
  - Page replacement: the LRU page is at the bottom of the stack
  - Typical implementation: microcoded doubly linked list
    - Used by one of the earlier CDC machines
  - Still too high a hardware cost

# Page Replacement: LRU Approximations

- Page reference bit
  - Stored with the frame containing the page
  - Bit is set whenever the page is accessed
  - Periodically, the OS (or hardware) resets all reference bits
  - Page replacement: Choose an unreferenced page

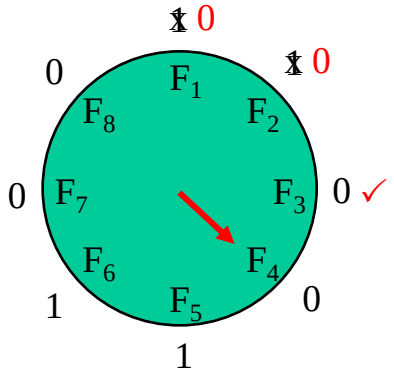
- Additional reference bits
  - For each page *p*, OS maintains an n-bit last-reference-time *lrt[p]*
  - Periodically, OS (or hardware)
    - Shifts right *lrt[p]*, adds current reference bit as MSB, and resets reference bit
  - Note that the additional bits can be maintained in software

  - Page selected is the one with the lowest *lrt*

    *lrt[p1]* = 11000100 has been used more recently than *lrt[p2]* = 01110111

# Page Replacement: LRU Approximations (cont'd)

- Second-chance Algorithm (also known as Clock)
  - Only uses single-bit page reference information
  - Maintains a list of frames as a circular list
  - Maintains a pointer into the list
  - Replacement: search for a page with reference bit zero
    - If there is a page with reference bit 1
      - Set the bit to 0, and continue searching
    - Each page gets a second chance before being evicted




- Enhanced second-chance algorithm
  - Make decision using two bits: page reference and page modify
    - (0, 0): neither recently used nor modified: *best candidate*
    - (0, 1): not recently used but modified
    - (1, 0): recently used, but not modified
    - (1, 1): recently used and modified: *worst candidate*
  - Used in the Macintosh

# Page Replacement: Performance Enhancements

- Maintain a pool of free frames
  - Buffered (delayed) writes
    - Frame allocation precedes deallocation
    - Allocate immediately from pool, replace later
  - Rapid frame and page reclaim
    - Keep track of which page was in which frame
    - Reclaim pages from free pool if referenced before re-use
      - Can be used as an enhancement to FIFO schemes
- Background updates of writes to secondary store
  - Whenever the disk update mechanism is free
    - Write out a page whose modified bit is set and then reset the bit
- Delayed write (copy-on-write)
  - Create a *lazy* copy (on the first write): defer allocation
    - Used to optimize Unix fork, memcpy