G22.2250-001
Operating Systems

Lecture 7

Process Deadlocks (cont'd)
Memory Management

October 16, 2007

Outline

- Announcements
  - Lab 3 due this Friday (10/19)

- Process Deadlocks
  - (Review) Deadlock prevention
  - Deadlock avoidance
  - Deadlock detection and recovery

- Memory Management
  - logical versus physical address space
  - swapping
  - allocation schemes

[Silberschatz/Galvin/Gagne: Sections 7.3 – 7.8, Chapter 8]

(Review)
Deadlock Prevention

- Main idea: Prevent one of the four necessary conditions
  - mutual exclusion
  - hold-and-wait: ask for all resources at start
  - no preemption
  - circular wait: resource ranking scheme

- Limitations
  - inefficient
    - static allocation of resources reduces concurrency
    - a process may need to be preempted even when there is no deadlock
  - restrictive
    - requires allocation of future resource requirements before it starts executing

- Alternative approaches: Deadlock avoidance, detection and recovery

Deadlock Avoidance

- Deadlock occurs because processes are waiting on each other to release resources

- Main idea of deadlock avoidance:
  - request additional information about how resources are to be requested
  - before allocating request, check if system will enter a deadlock state

    F (resources available, resources allocated, future requests/releases)

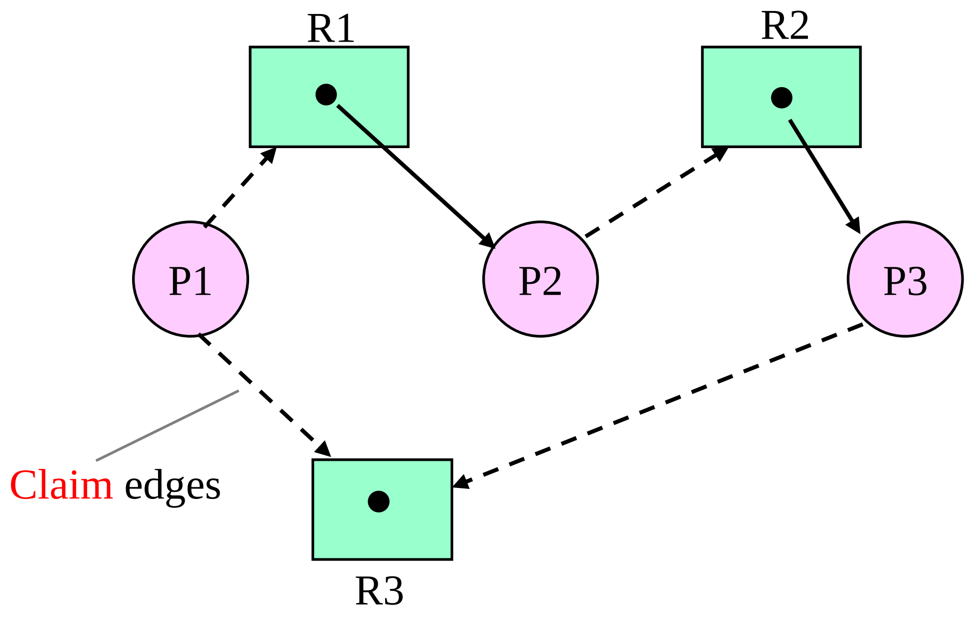
    - if no: grant the request
    - if yes: block the process

- Algorithms differ in amount and type of information
  - simplest (also most useful) model: maximum number of resources
  - other choices
    - sequence of requests and releases
    - alternate request paths

- How can we find out if a system will enter a deadlock state?

Deadlock Avoidance: Notion of a Safe State

- A system is in a safe state iff there exists a safe sequence
- A sequence $<P_1, P_2, \ldots, P_n>$ is a safe sequence for the current allocation if, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus resources held by all the P_j, for $j<i$



<P3, P2, P1> is a safe sequence

P3's future request can be satisfied because R3 is available

P2's future request can be satisfied by P3 yielding R2

P1's future requests can be satisfied by P2 giving up R1 and available R3

Properties of Safe States

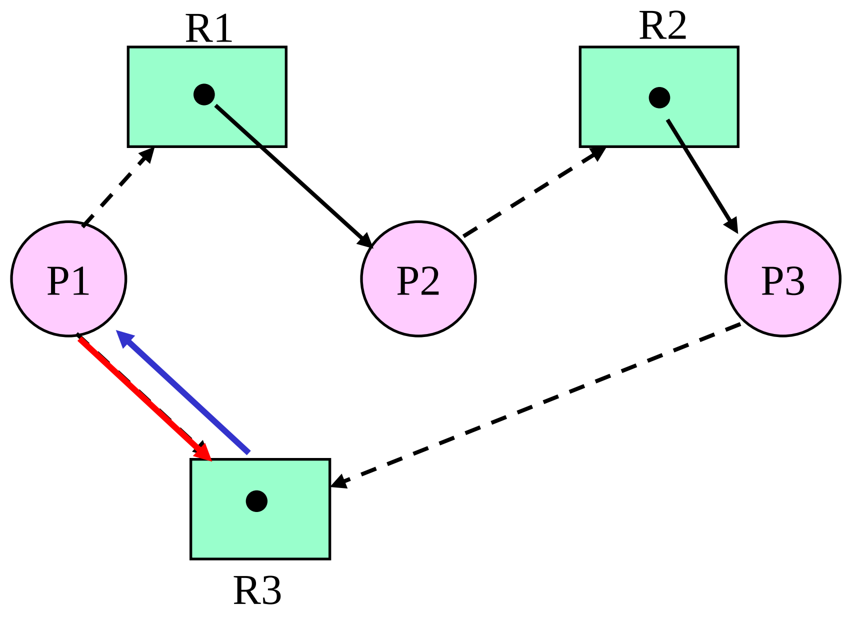
- A safe state is not a deadlock state
- An unsafe state may lead to deadlock
- It is possible to go from a safe state to an unsafe state

- Example: A system with 12 units of a resource
  - Three processes
    - P_1: max need = 10, current need = 5
    - P_2: max need = 4, current need = 2
    - P_3: max need = 9, current need = 2
  - This is a safe state, since a safe sequence $<P_2, P_1, P_3>$ exists
  - P_3 requests an additional unit. Should this request be granted?
  - No, because this would put the system in an unsafe state
    - P_1, P_2, P_3 will then hold 5, 2, and 3 resources (2 units are available)
    - P_2's future needs can be satisfied, but no way to satisfy P_1's and P_3's needs

- Avoidance algorithms prevent the system from entering an unsafe state

Deadlock Avoidance: Single Resource Instances

- Deadlock ≡ Cycle in the resource allocation graph
- A request is granted iff it does not result in a cycle
  - cycle detection: O(V + E) operations



<P3, P2, P1> is a safe sequence

Say P1 requests R3:
should this be granted?

No, because an assignment edge from R3 to P1 would create a cycle in the RAG.
[No safe sequence exists]
Does this always imply a deadlock?

No, because P1 can release R3 before requesting R1

Deadlock Avoidance: Multiple Resource Instances

- Banker's Algorithm
  - upon entering the system, a process declares the maximum number of instances of each resource type that it may need
  - the algorithm decides, for each request, whether granting it would put the system in an unsafe state

resource availability
  Available[1..m]
maximum demand
  Max[1..n, 1..m]
current allocation
  Allocation[1..n, 1..m]
potential need
  Need[1..n, 1..m]

```
1. If $Request_i \leq Need_i$
   goto Step 2, else flag error

2. If $Request_i \leq Available$
   goto Step 3, else wait

3. Allocate the resources
      $Available := Available - Request_i;$
      $Allocation_i := Allocation_i + Request_i;$
      $Need_i := Need_i - Request_i;$
   Check if this is a safe state.
   If not: undo the allocation and wait
```

```
1. $Work := Available;$
   $Finish[i] := false$, for all $i$;
2. Find an $i$ such that
      a. $Finish[i] = false$, and
      b. $Need_i \leq Work$
      if no such $i$, goto Step 4
3. $Work := Work + Allocation_i;$
   $Finish[i] := true;$
   goto Step 2;
4. If $Finish[i] = true$ for all $i$,
   then the system is in a safe state
```

Banker's Algorithm: Example

- Three resource types and
  three processes (P_1, P_2, P_3)

Capacity =	[2, 4, 3]
Max =	[[1, 2, 2], [1, 2, 1], [1, 1, 1]]
Allocation =	[[1, 2, 0], [0, 1, 1], [1, 0, 1]]
Available =	[0, 1, 1]
Need =	[[0, 0, 2], [1, 1, 0], [0, 1, 0]]

- P_1 requests [0, 0, 1]
  Should this be granted?

- Allocate and check if system is in a safe state

Allocation =	[[1, 2, 1], [0, 1, 1], [1, 0, 1]]
Available =	[0, 1, 0]
Need =	[[0, 0, 1], [1, 1, 0], [0, 1, 0]]

```
1. Work := Available;
   Finish[i] := false, for all i;
2. Find an i such that
     a. Finish[i] = false, and
     b. $Need_i \leq Work$
     if no such i, goto Step 4
3. $Work := Work + Allocation_i$;
   Finish[i] := true;
   goto Step 2;
4. If Finish[i] = true for all i,
   then the system is in a safe
   state
```

Initially, *Work* = [0, 1, 0]

$Need_3 \leq Work$, so P_3 can finish

Work = [1, 1, 1]

Now, both P1 and P2 can finish

Limitations of Deadlock Avoidance

- Deadlock avoidance vs. deadlock prevention
  - Prevention schemes work with local information
    - What does this process already have, what is it asking
  - Avoidance schemes work with global information
    - Therefore, are less conservative

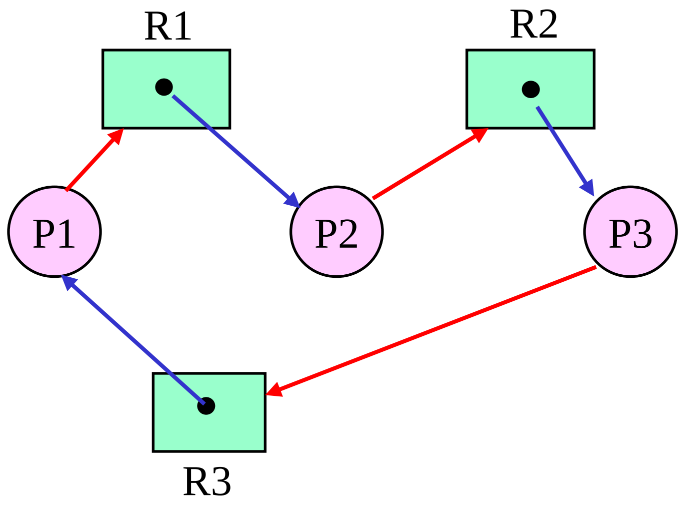
- However, avoidance schemes require specification of future needs
  - not generally known for OS processes
  - more applicable to specialized situations
    - programming language constructs (e.g., transaction-based systems)
    - known OS components (e.g., Unix “exec”)

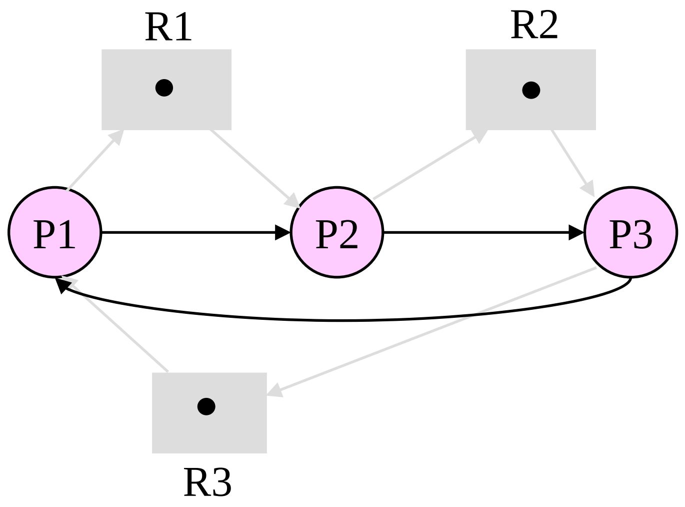
- More general solution: Deadlock detection and recovery

Deadlock Detection: Single Resource Instances

- Go back to using a resource allocation graph in which only
  - request and assignment edges are defined
  - future (potential) requests are not relevant to "is there deadlock now?"

- Deadlock ≡ Cycle in the RAG
  - need only look at the *wait-for graph*
    - obtained by removing resource nodes and collapsing the appropriate edges

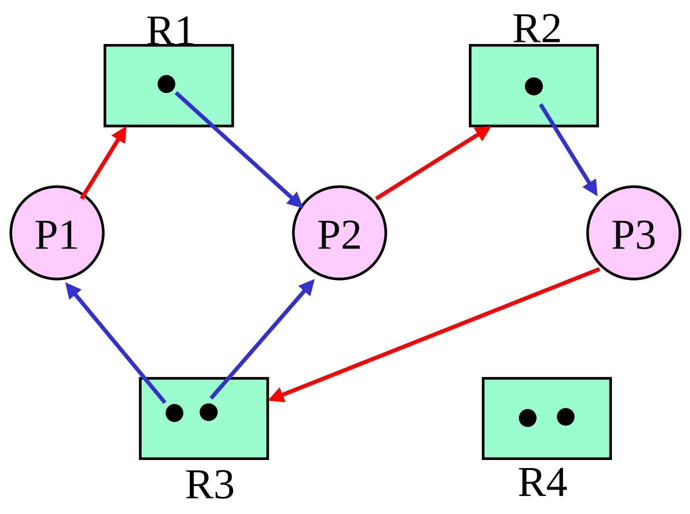


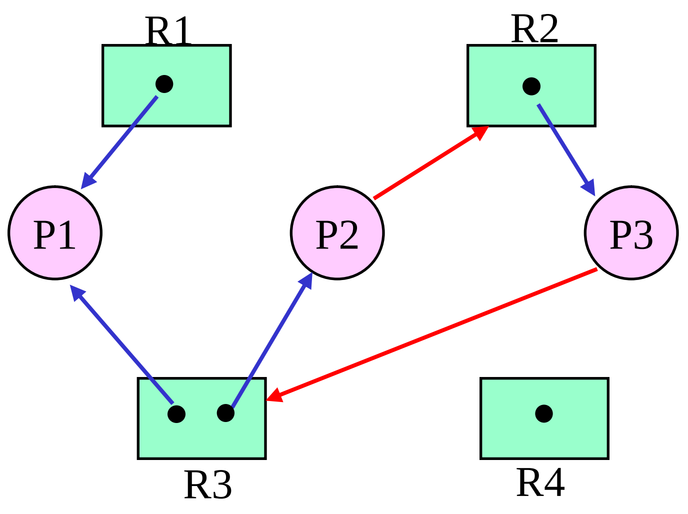


Deadlock Detection: Multiple Resource Instances

- A cycle in the graph is a necessary but not sufficient condition for the existence of a deadlock
  - if a cycle does not exist: no deadlock
  - if a cycle exists: there may or may not be a deadlock

(Examples from earlier in the lecture)





Detection: Multiple Resource Instances (cont'd)

- A new use for the Bankers' algorithm
  - detect if the current set of requests are such that satisfying any of them will put the system in an unsafe state

```
1. Work := Available;
   Finish[i] := false, for all i;
2. Find an i such that
      a. Finish[i] = false, and
      b. $Request_i \le Work$
      if no such i, goto Step 4
3. $Work := Work + Allocation_i$;
   Finish[i] := true;
   goto Step 2;
4. If Finish[i] = false for some i,
   then the system is in a deadlock state
```

Find a process that is currently not deadlocked

Optimistically assume that this process will not need more resources

```
1. Work := Available;
   Finish[i] := false, for all i;
2. Find an i such that
      a. Finish[i] = false, and
      b. $Need_i \le Work$
      if no such i, goto Step 4
3. $Work := Work + Allocation_i$;
   Finish[i] := true;
   goto Step 2;
4. If Finish[i] = true for all i,
   then the system is in a safe state
```

Detection: Multiple Resource Instances (Example)

- System with three resource types and five processes

		Allocation	*Request*	*Available*
✓	~~P0~~	~~[0, 1, 0]~~	~~[0, 0, 0]~~	[3, 1, 3]
✓	P1	[2, 0, 0]	[2, 0, 2]	
✓	~~P2~~	~~[3, 0, 3]~~	~~[0, 0, 0]~~	
✓	P3	[2, 1, 1]	[1, 0, 0]	
✓	P4	[0, 0, 2]	[0, 0, 2]	

No deadlock!

- What about the following?

		Allocation	*Request*	*Available*
✓	~~P0~~	~~[0, 1, 0]~~	~~[0, 0, 0]~~	[0, 1, 0]
	P1	[2, 0, 0]	[2, 0, 2]	
	P2	[3, 0, 3]	[0, 0, 1]	
	P3	[2, 1, 1]	[1, 0, 0]	
	P4	[0, 0, 2]	[0, 0, 2]	

Deadlock!

Deadlock Recovery

- Only general principles known (read Section 7.7 for details)

Two options

- Break the cyclic waiting by terminating some of the processes

  choice 1: abort all deadlocked processes

  choice 2: abort one process at a time till deadlock resolved

- Enable at least one of the processes to make progress (by preempting resources from another)
  - issue 1: how is the victim process selected?
  - issue 2: can the process handle resource preemption?
    - in general, might require rollback and restart
  - issue 3: how does one prevent starvation?
    - bound the number of rollbacks/preemptions for a particular process

Combined Approaches

- Using only a single approach (prevention, avoidance, or detection + recovery) in isolation is not very effective

Combination is superior

- General idea: Classify resources, use different approach for each
- Example: Consider a system with four classes of resources
  - internal resources (e.g., PCBs)
  - main memory
  - job resources (e.g., tape drives, files)
  - swappable space
- A combined deadlock solution

– process control blocks:	use resource ordering (prevention) Why?
– user process memory:	use pre-emption (detection/recovery)
– job resources:	require prior claims (avoidance)
– swappable space:	preallocate; no hold and wait (prevention)

Outline

- Announcements
  - Lab 3 due this Friday (10/19)

- Process Deadlocks
  - (Review) Deadlock prevention
  - Deadlock avoidance
  - Deadlock detection and recovery

- Memory Management
  - logical versus physical address space
  - swapping
  - allocation schemes

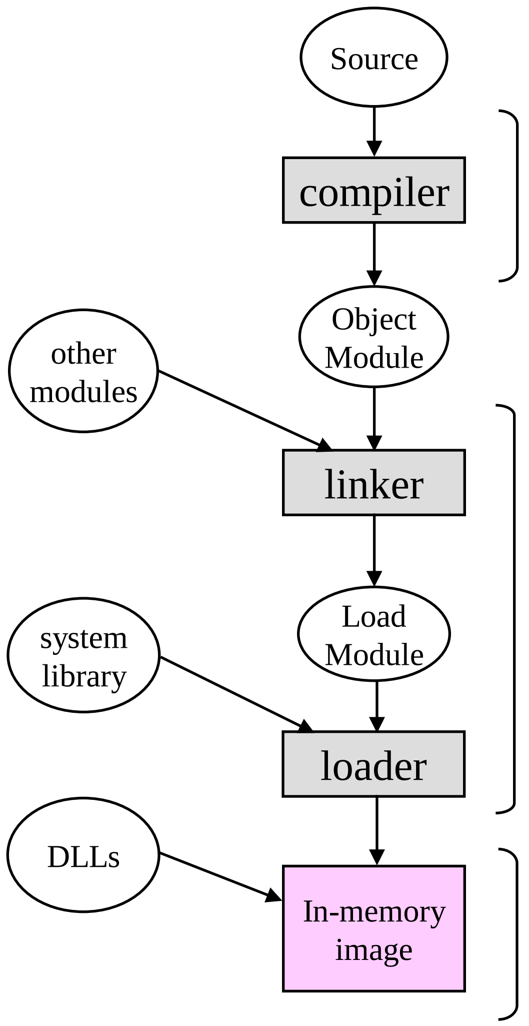
[Silberschatz/Galvin/Gagne: Sections 7.3 – 7.8, Chapter 8]

Memory Management: Background

- Programs operate on data and instructions stored in memory (von Neumann model)
  - memory is shared by multiple processes and is limited in size
  - further, the actual programming prior to compilation uses symbolic representations of these locations which get translated into actual (or physical) memory locations

- Memory management: Providing efficient mechanisms for
  - binding: mapping program names into actual memory locations
  - mapping: utilizing the limited physical memory to bring logical memory objects (belonging to multiple processes) back and forth
    - Lectures 13 and 14: allocation of physical memory to processes
      - assume that the entire process fits in physical memory
    - Lectures 14 and 15: supporting virtual memory in allocated physical memory
      - process data and instructions need not all fit into physical memory

Binding Program Names: Logical to Physical



at compile-time

- mapping of logical-to-physical addresses is done statically
- ⬇ changes in the physical address map require recompilation
- rare for general programs, sometimes for OS components

at load-time

- binding done by the loader when program is brought into memory for execution
- change in the starting address only requires a reload

at run-time

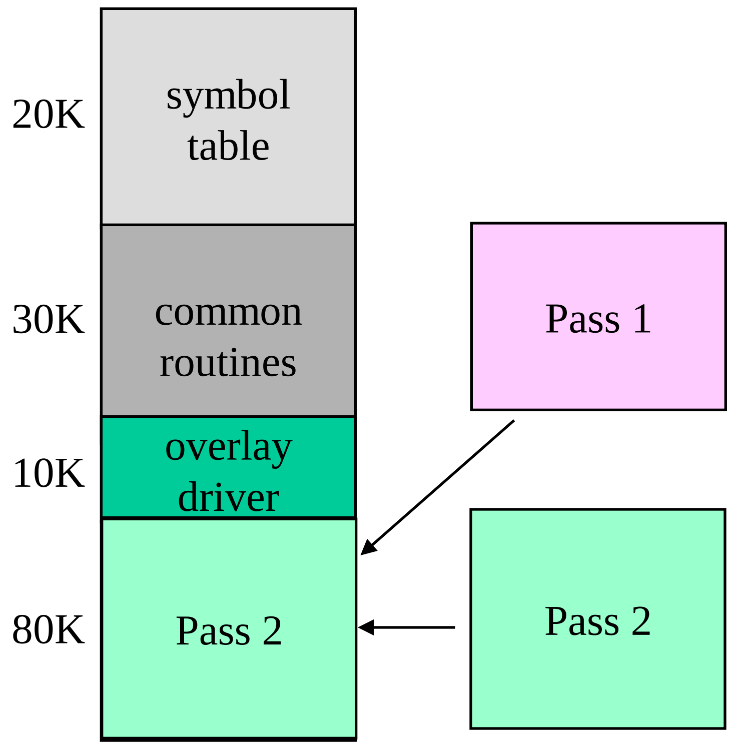
- binding is delayed until the program actually executes
    - special hardware support needed to accomplish this
- more details in the rest of the lecture

Process Memory Requirements

- So far, we have assumed that the entire process and data need to fit into memory for the program to execute
  - Many techniques to reduce amount that needs to fit at any time

Explicit management by the programmer

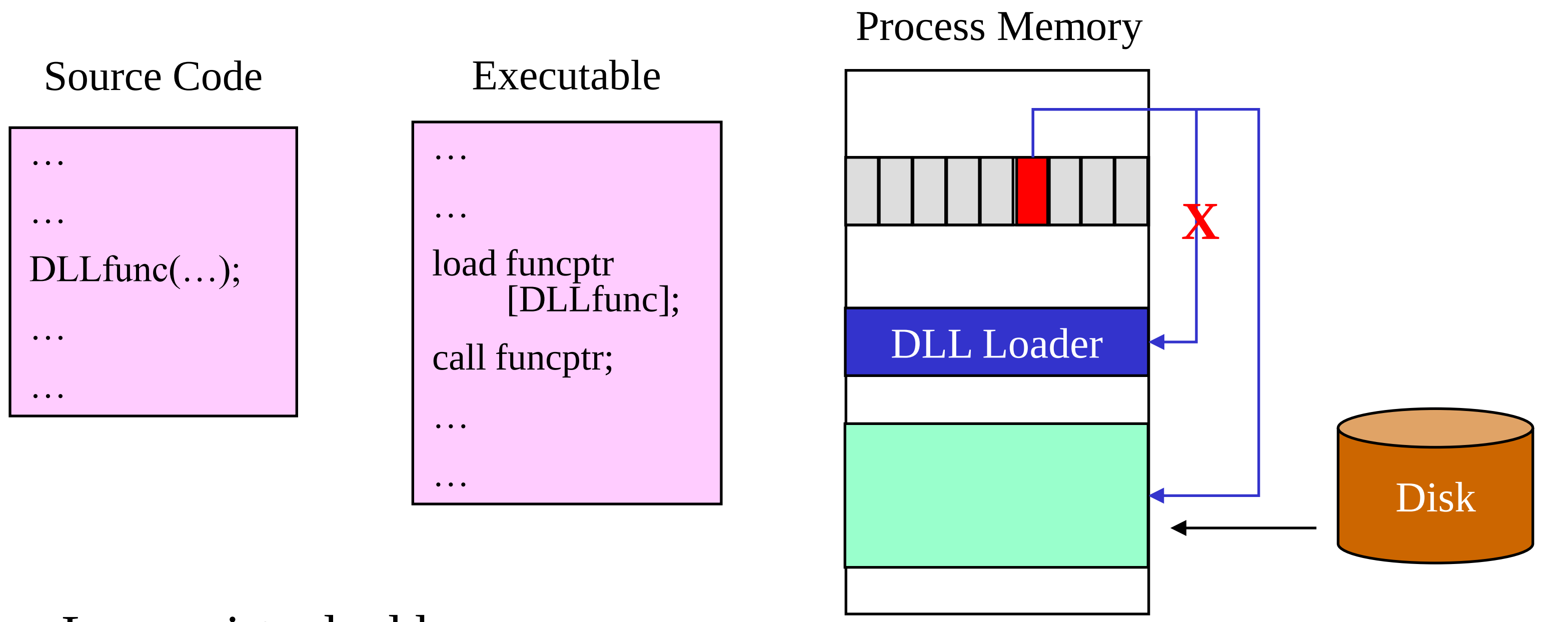
- dynamic loading
  - load procedures "on demand"
- overlays
  - keep in memory only those instructions/data that are needed at any given time
  - rewrite portions of the address space with new instructions/data as required



Process Memory Requirements (cont'd)

Implicit management by the OS

- Dynamic linking
  - typically used with shared system libraries that are loaded on demand
    - calls resolved using an "import table": initially point to the loading stub



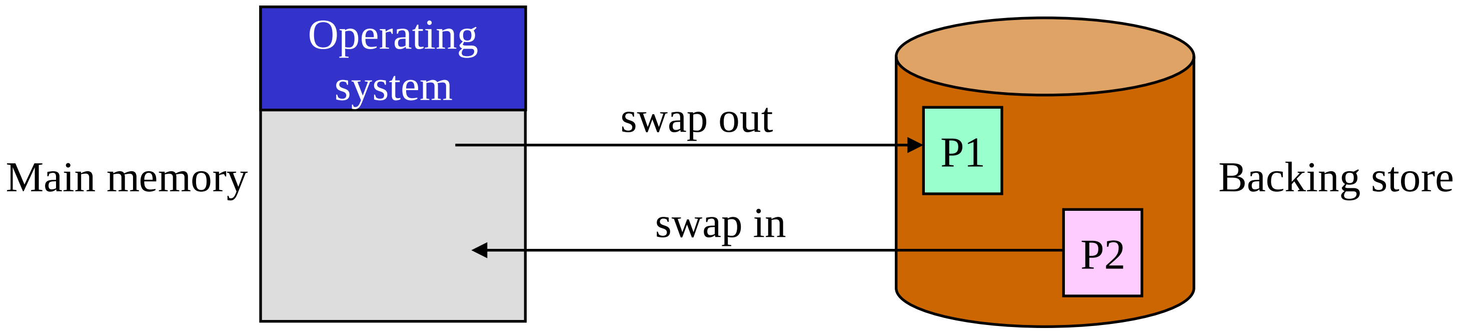
- Large virtual address spaces
  - more about this in Lecture 8

Multiprogramming and Swapping

- Problem: Memory requirements of all the processes cannot be simultaneously met

Solution: Swapping

- "Dynamically" move a process out of memory into a backing store (and back in) as dictated by the medium-term scheduler
  - backing store is typically a fast disk
  - choice of which processes to swap out/in
    - can be influenced by short-term scheduling policy (e.g., priority-driven)
    - knowledge of process' actual memory requirements
      - requires the process to reserve, commit, decommit, and release memory



Swapping: Issues

- High context-switch times
  - assume a user process of size 100 KB
  - backing store is a standard hard disk with transfer rate of 5 MB/s
  - actual transfer of 100 KB from and to memory takes

    $2 \times (100\text{ KB} / 5000\text{ KB/s}) = 2 \times (1/50\text{ second})$
    $= 2 \times (20\text{ ms}) = 40\text{ ms} + \text{disk time}$

  - helps to know exactly how much memory is being used
  - also, determines frequency

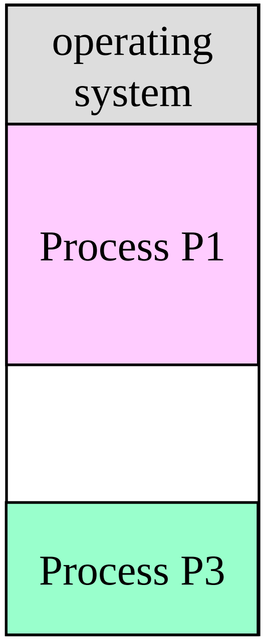
- Swapping out a process that is currently in the middle of I/O
  - I/O completion might store values in memory, now occupied by a new process
  - common solutions
    - never swap out a process while in a wait state induced by I/O requests
    - all I/O interactions are via a special set of buffers that are controlled by the OS and are part of its space; not swapped out

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

Memory Mapping (1): Partitioning

- Idea: Divide memory into partitions
- Protection
  - each partition protected with a “key”
  - at run time, process key (stored in a register) matched with partition key
    - on mismatch, generates a trap
- Allocation
  - fixed partitions
    - memory is divided into a number of fixed size partitions
    - each partition is allotted to a single process
    - used in the early IBM 360 models
    - no longer in use
  - variable partitions
    - contiguous memory is allocated on loading
    - released on termination
    - this is what you will use in Nachos Lab 4



Memory Mapping: Partitioning (cont'd)

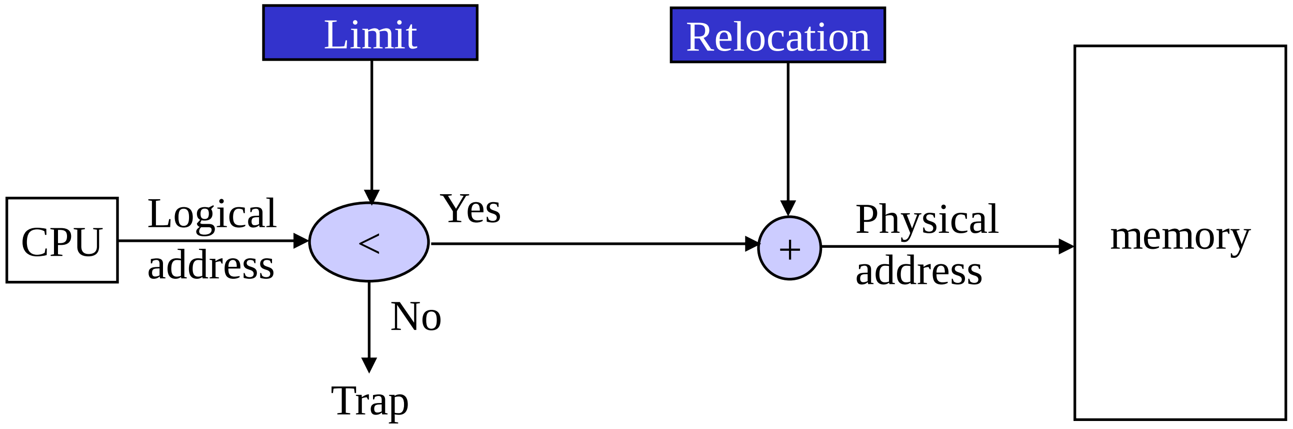
- Partitioning for statically-bound programs
  - programs must execute in the same place
  - allocation is inefficient, and swapping is very constrained
  - no provision for changing memory requirements

- Partitioning for dynamically-bound programs
  - relocation registers
    - a CPU register keeps track of the starting address where the process is loaded
    - whenever a memory location is accessed:
      - the system computes physical-address = logical-address + relocation register
      - fetches the value from the resulting memory location
    - the stream of physical addresses are seen only by the MMU

  - how to prevent a process from accessing addresses outside its partition?

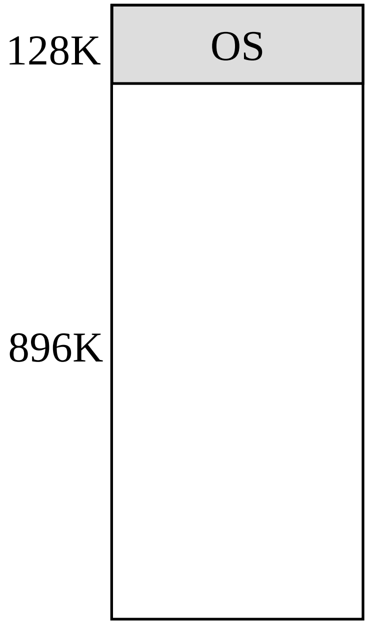
Memory Mapping: Partitioning (contd.)

- Protection and relocation for dynamically-bound programs
  - Two registers keep info for each partition: limit, relocation

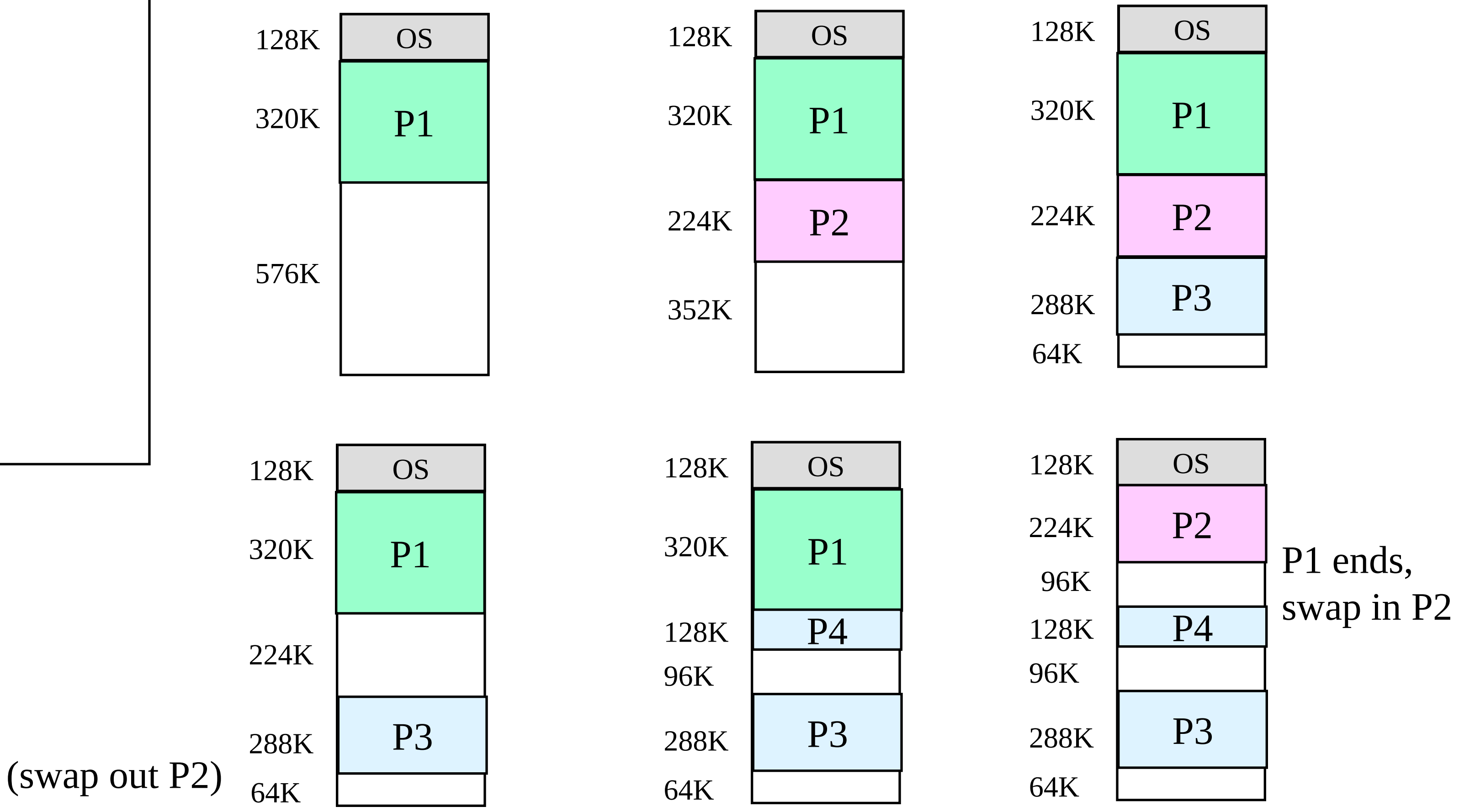


- Other advantages
  - relocation register can be changed on the fly
  - why is this useful?

Memory Allocation and Scheduling



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



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

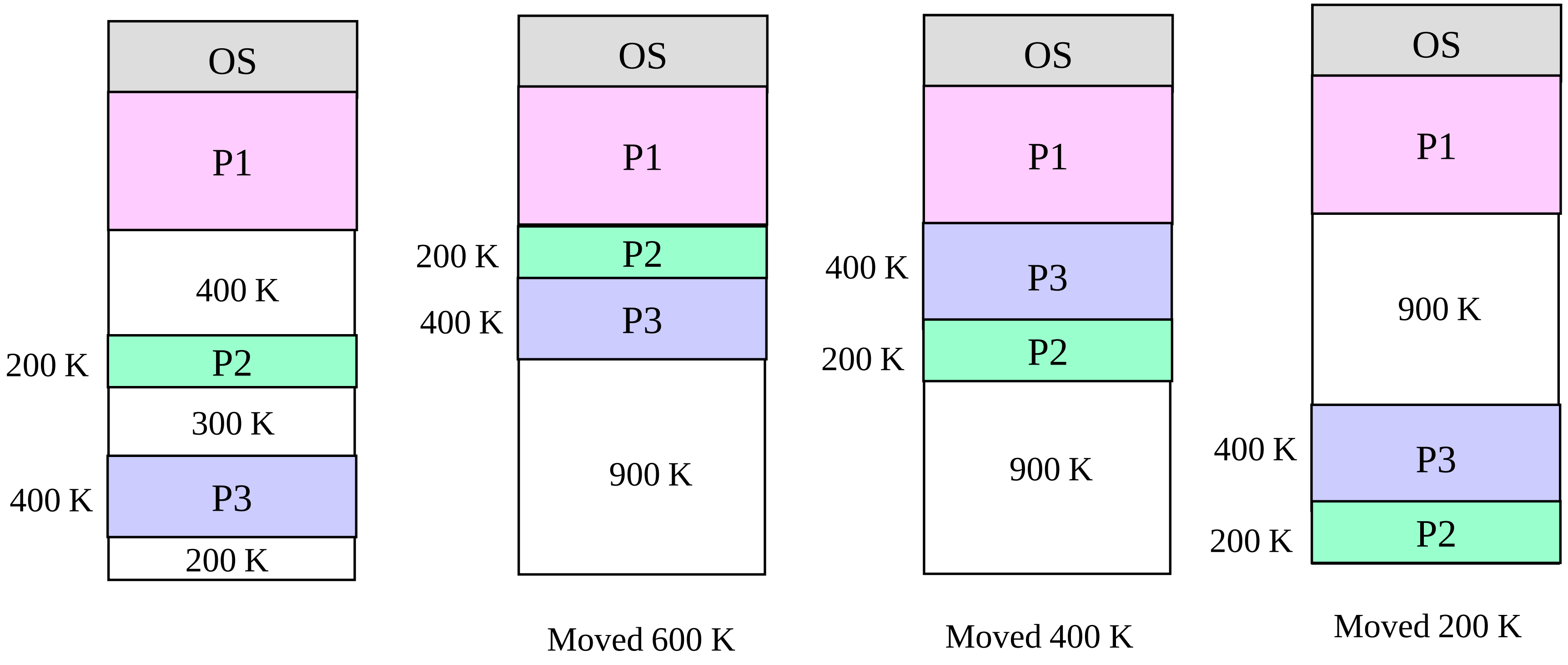
Partitioning Policies (contd.)

- 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

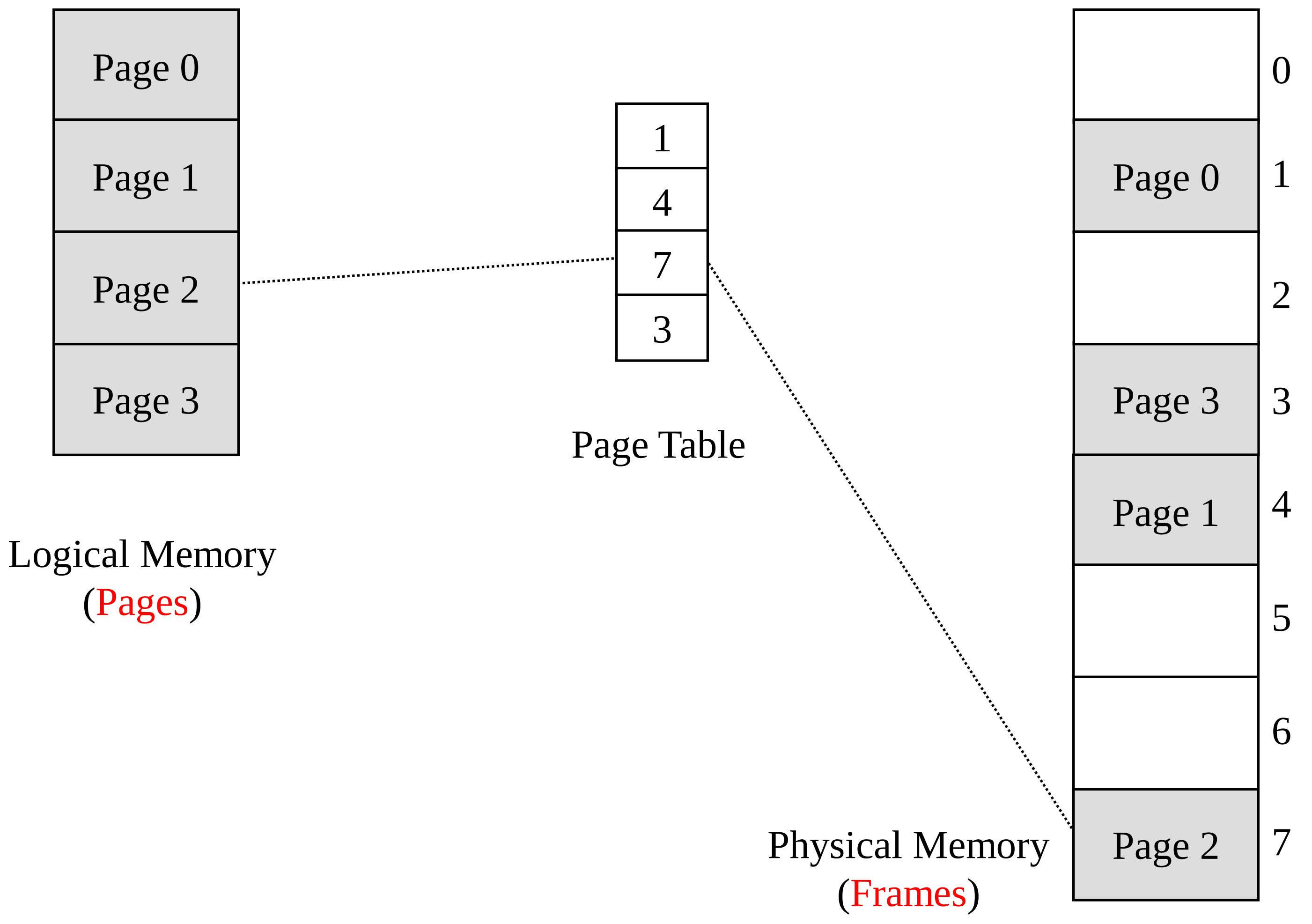
- Motivation: Partitioning suffers from large external fragmentation

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)

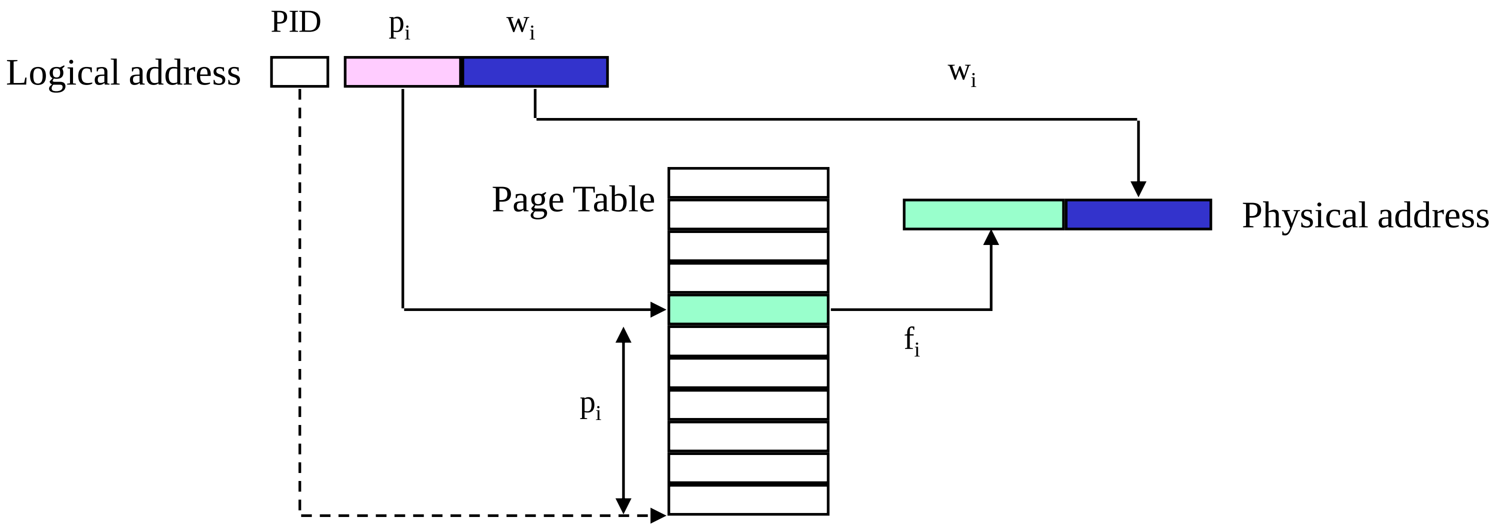


Paging (cont'd)

- 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_ith 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

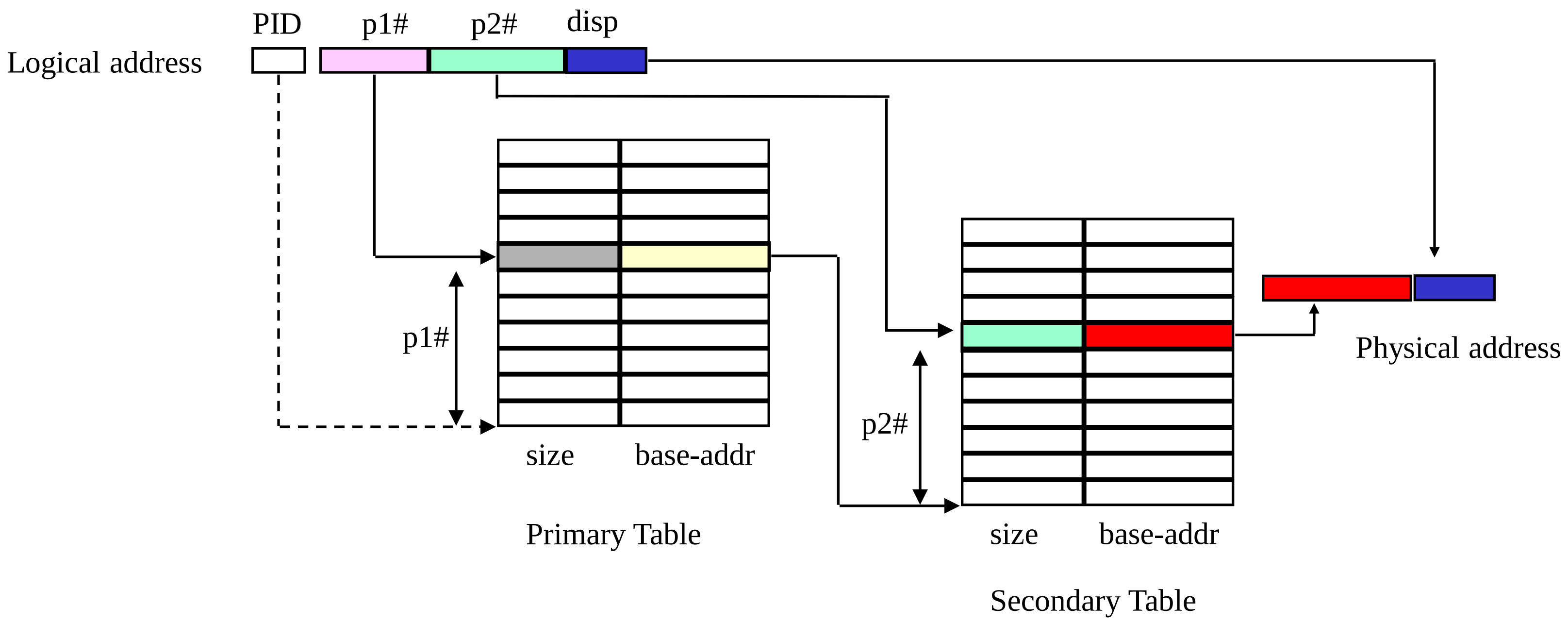
- Solution: Translation Lookaside Buffer (TLB)
  - a portion of the page table is cached in the TLB
    - little performance degradation if a value is a *hit* in the TLB
    - if not: a memory access is needed to load the value into the TLB
      - an existing value must be *flushed* if the TLB is full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

    $= 0.9*(\text{Access}_{\text{TLB}} + \text{Access}_{\text{mem}}) + 0.1*(\text{Access}_{\text{mem}} + \text{Access}_{\text{mem}})$

    $=\sim 1.1*(\text{Access}_{\text{mem}})$

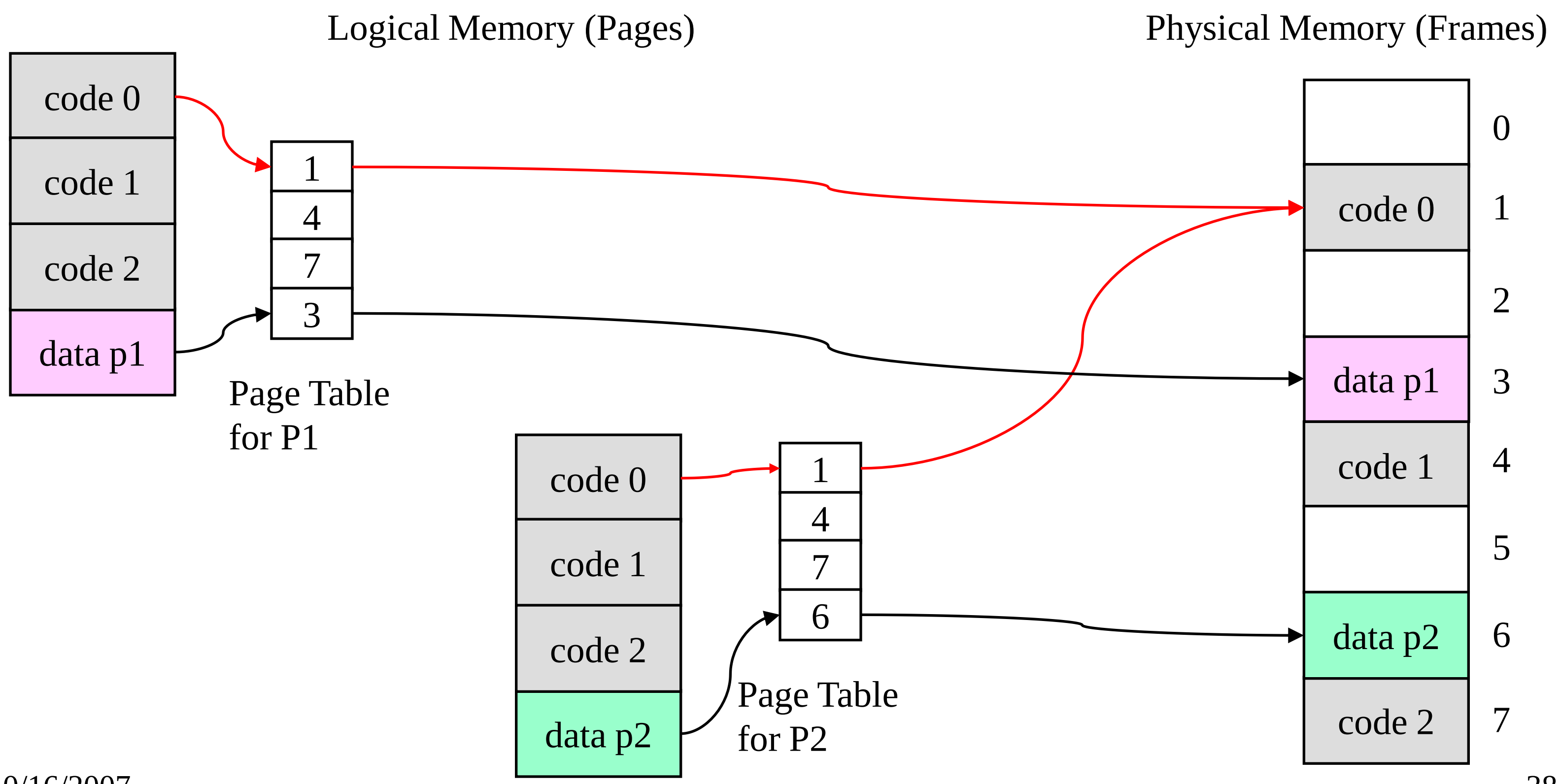
Multi-level Page Tables

- 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

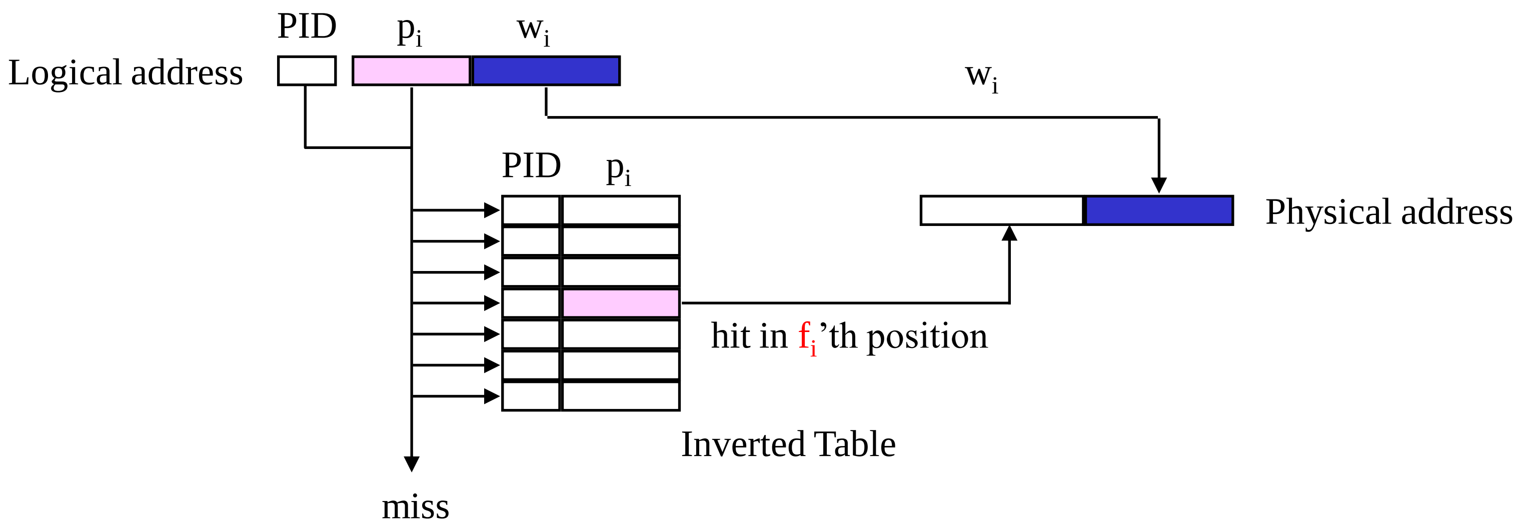
- Page tables permit different virtual addresses (frames of different processes) to map to the same physical address
  - convenient sharing of common code (dynamically-linked system libraries)
  - shared data segments for IPC



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?

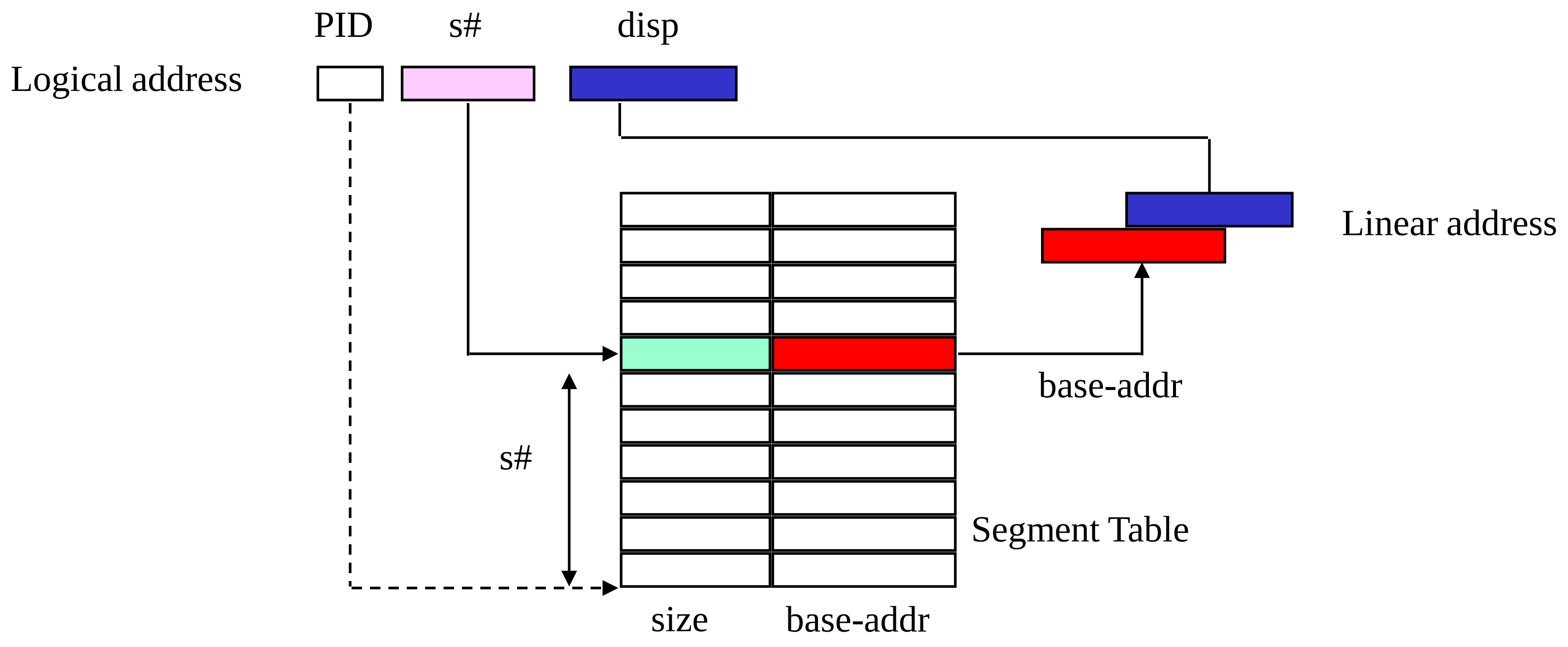
- Physical memory can only be accessed through page table mappings
  - all addresses are interpreted by the MMU
  - OS intervention required to manipulate page tables and TLBs
- Special bits in the page table entry enforce per-frame protection
  - an accessibility bit
    - whether a page is invalid, 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

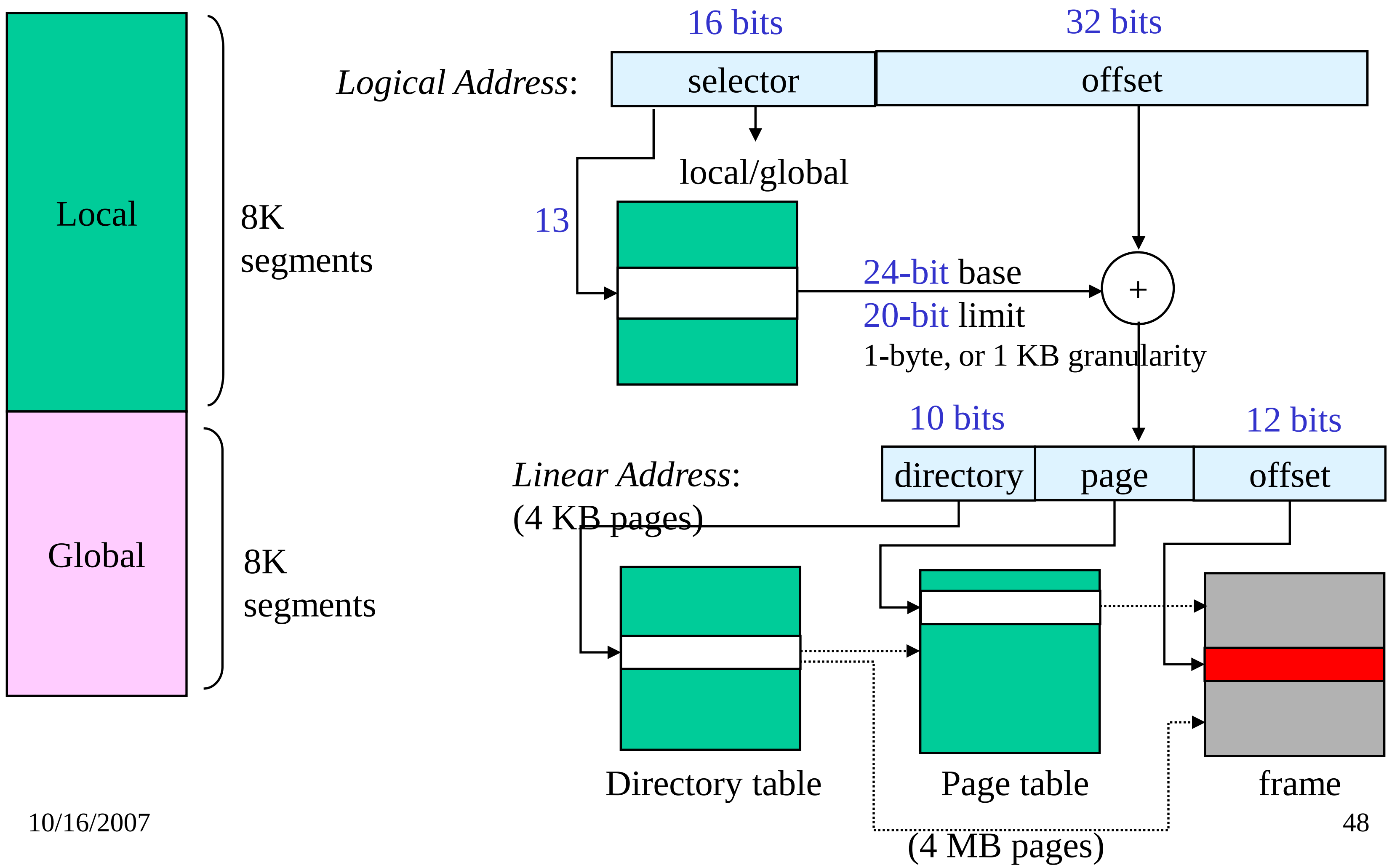
Memory Mapping: Examples

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