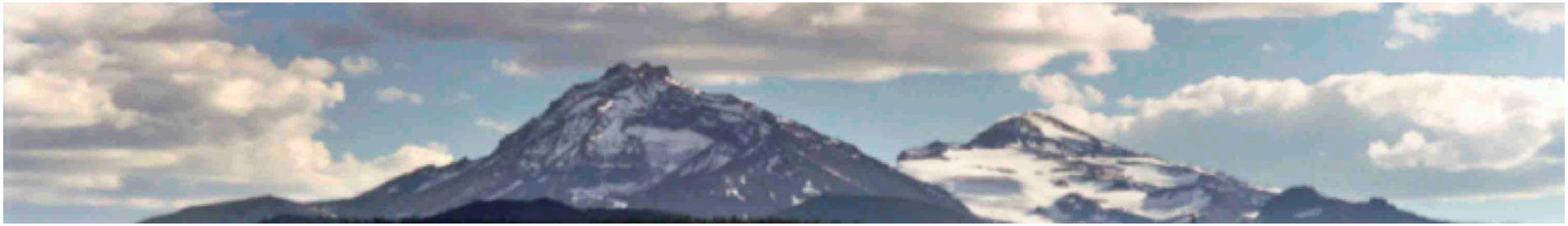


CS 415
Operating Systems
Memory Management

Prof. Jieyang Chen
Department of Computer and Information Science
Fall 2025

UNIVERSITY OF OREGON

Logistics

- Project 3
- Read chapter 9
- Student Experience Survey 11/26

Outline

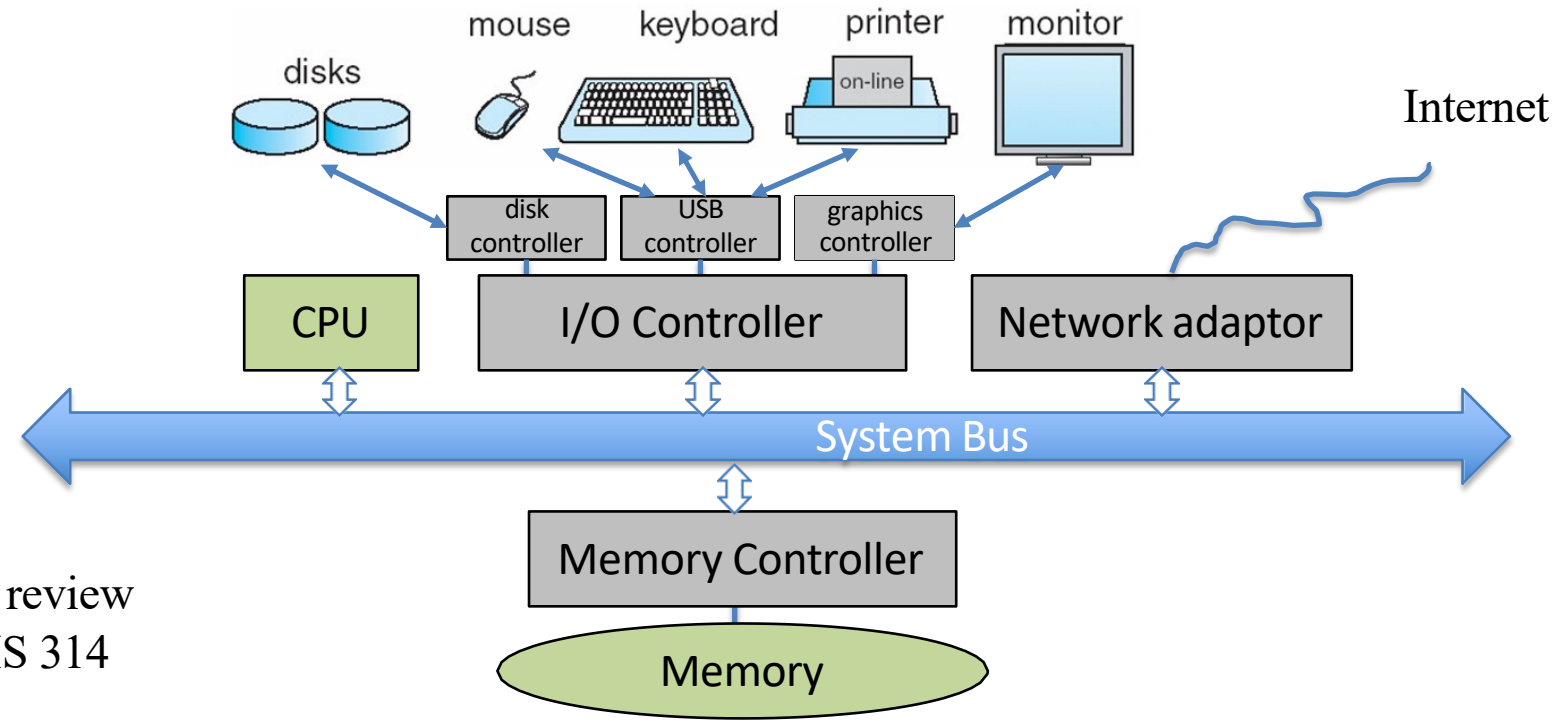
- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To give a detailed example pure segmentation and segmentation with paging

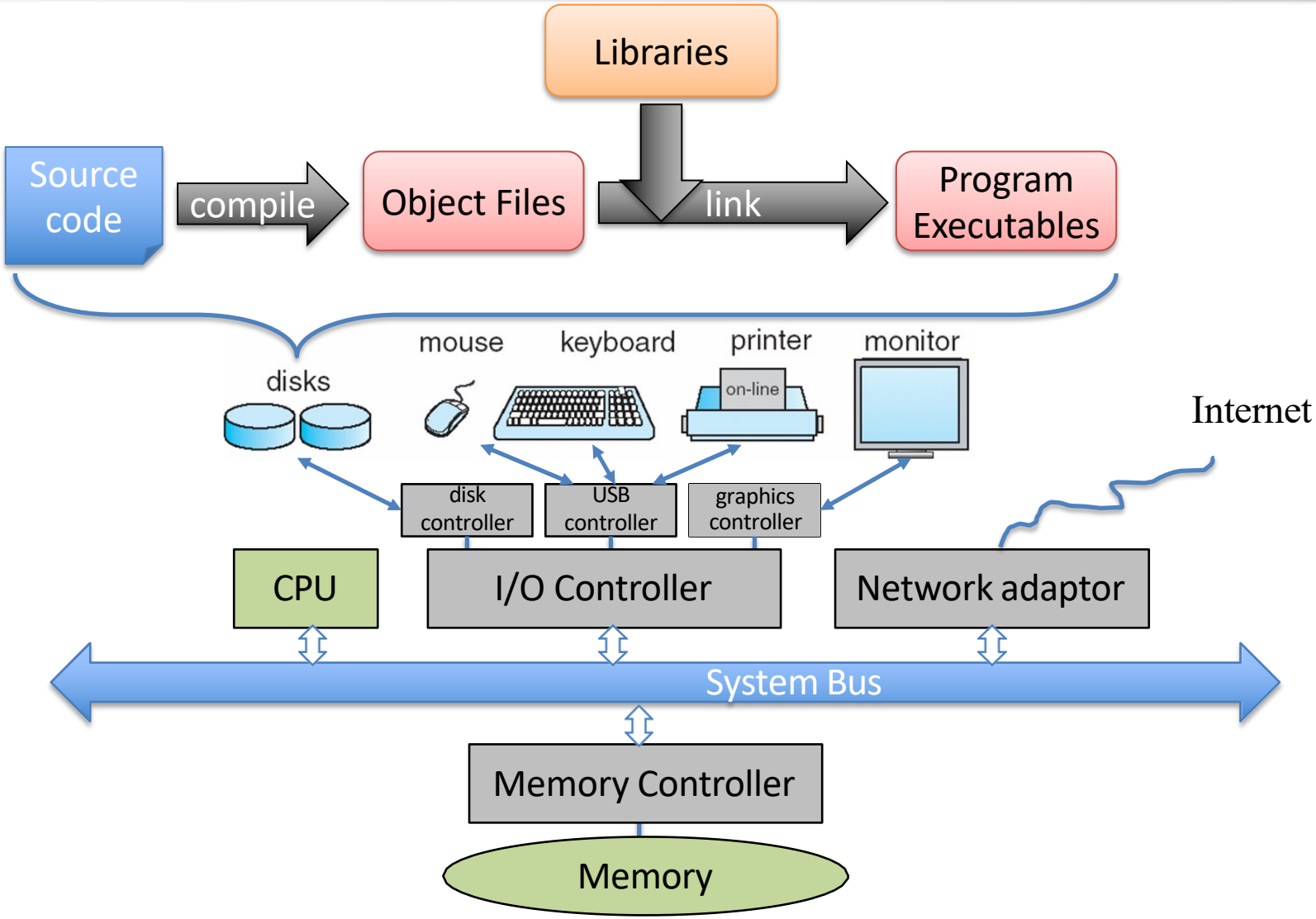
Review: Canonical System Hardware

- *CPU*: processor to perform computations
- *Memory*: hold instructions and data
- *I/O devices*: disk, monitor, network, printer, …
- *Bus*: systems interconnection for communication

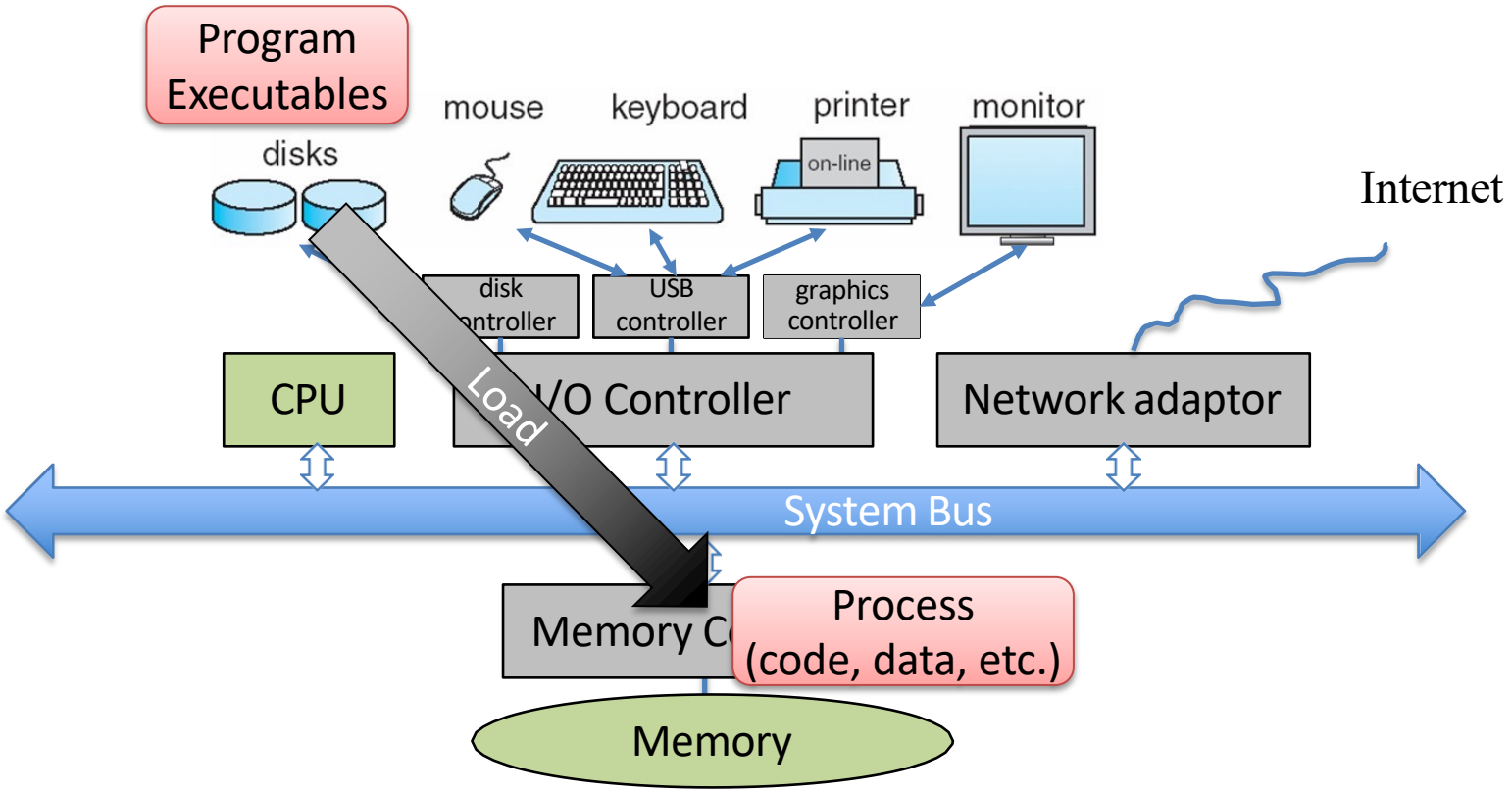


Quick review of CIS 314

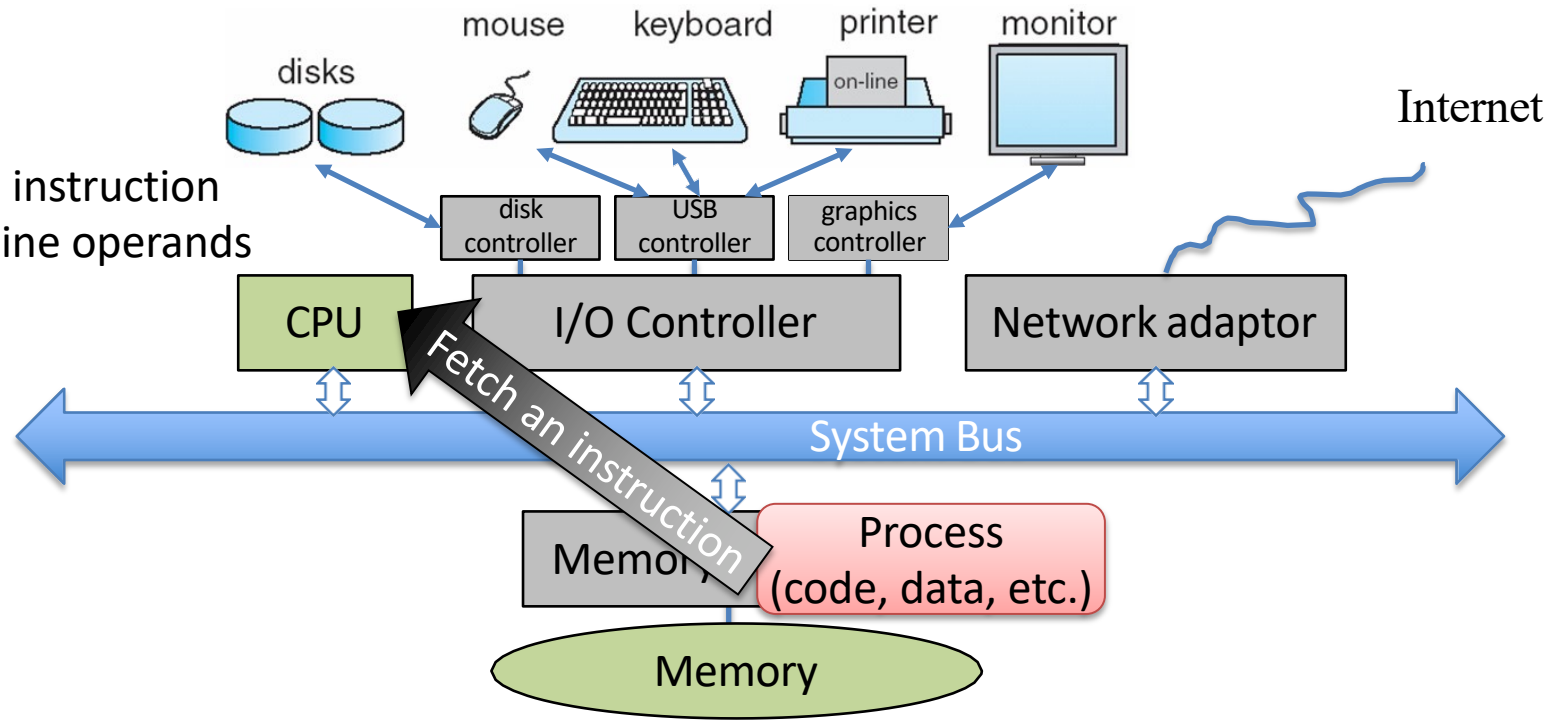
How does code become processes?



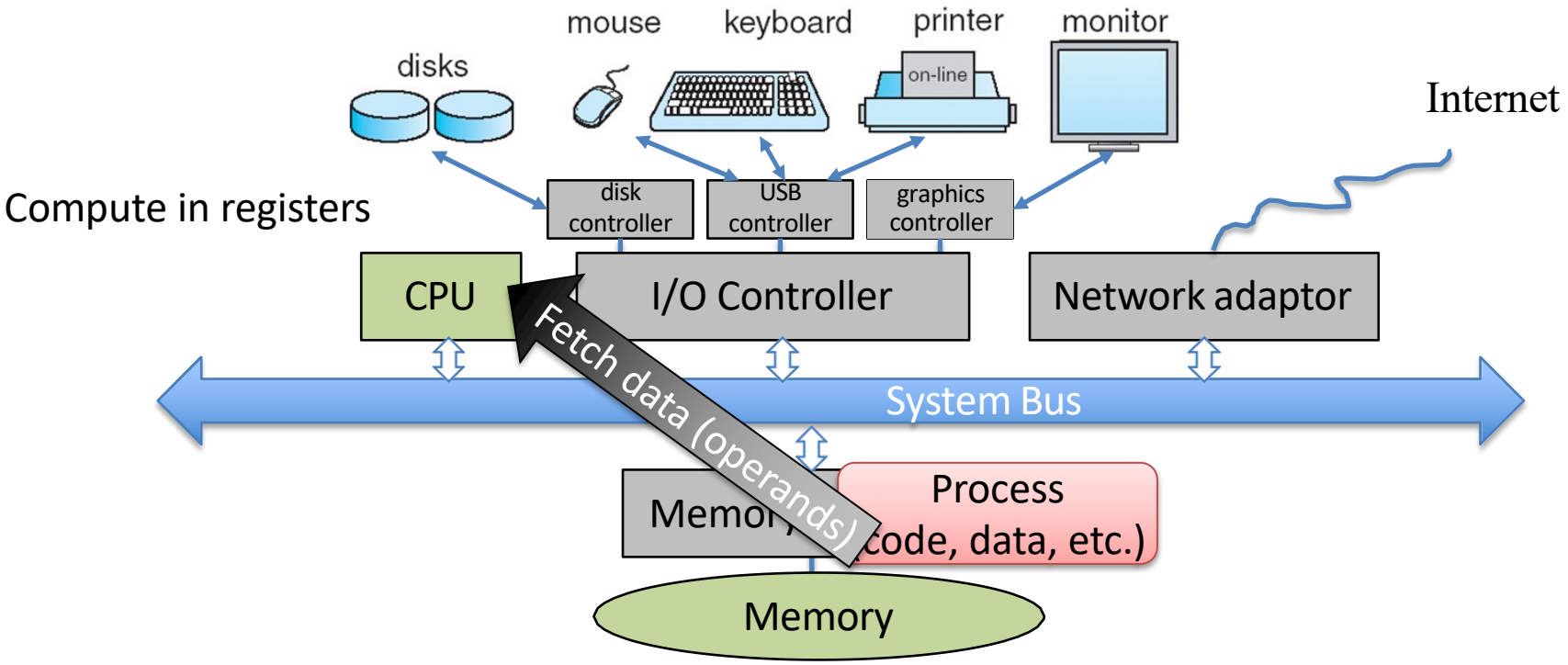
How does code become processes?



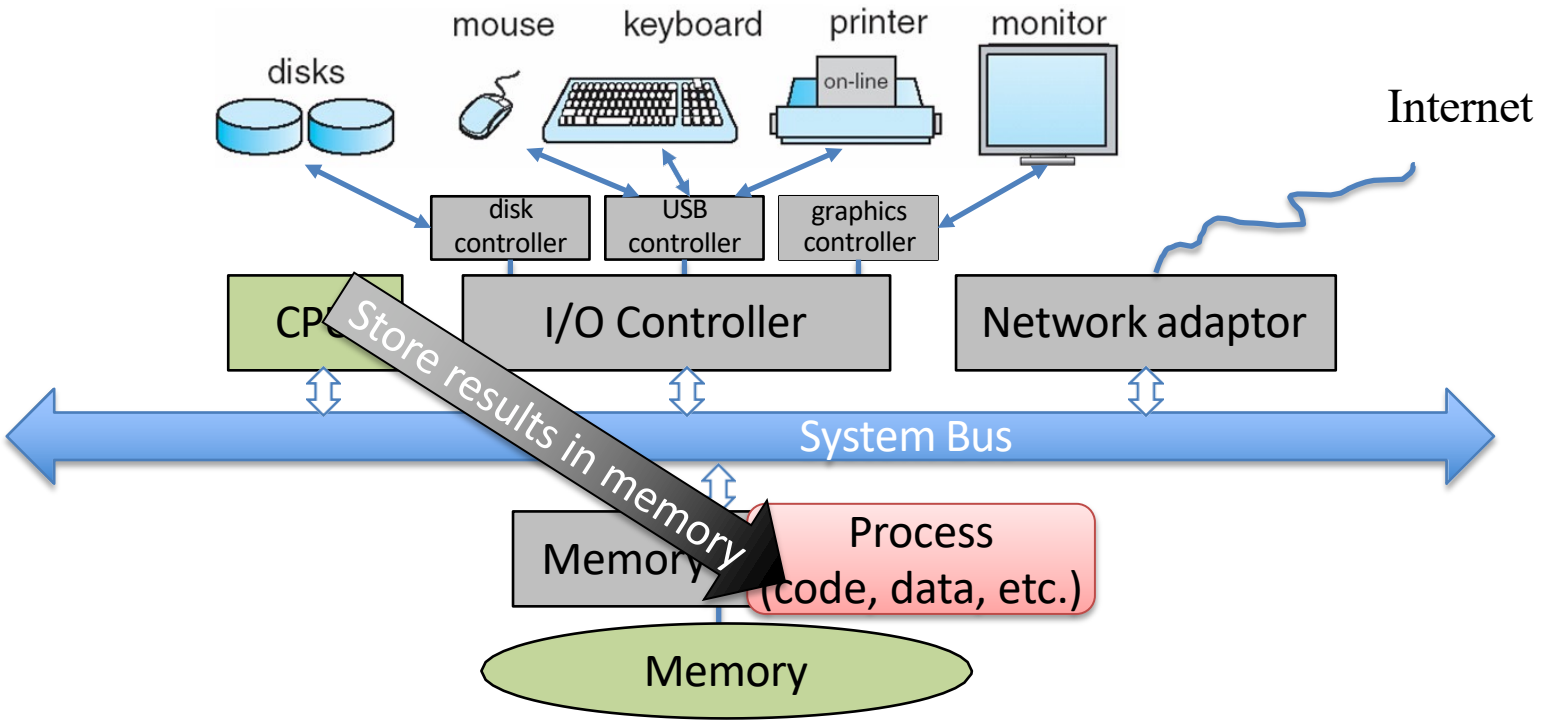
How does code become processes?



How does code become processes?

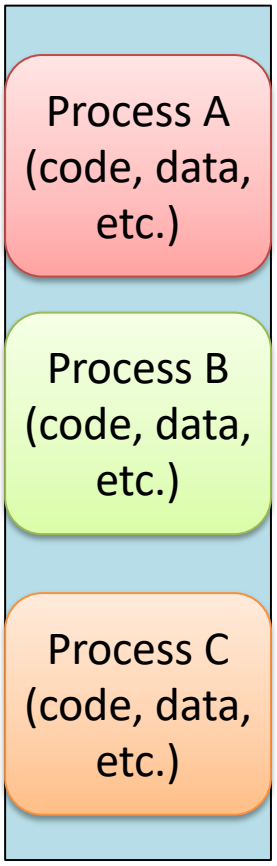


How does code become processes?



Let's think about….

- How do multiple processors share the same physical memory?
  - Who has access where?
  - Protection?
- Why do programmers not need to worry exactly where to put data in memory?
  - We always ask for a contiguous allocation (malloc). We never worried about where they are in memory.
- How does each instruction reference memory?
  - Are the addresses baked in each instruction?



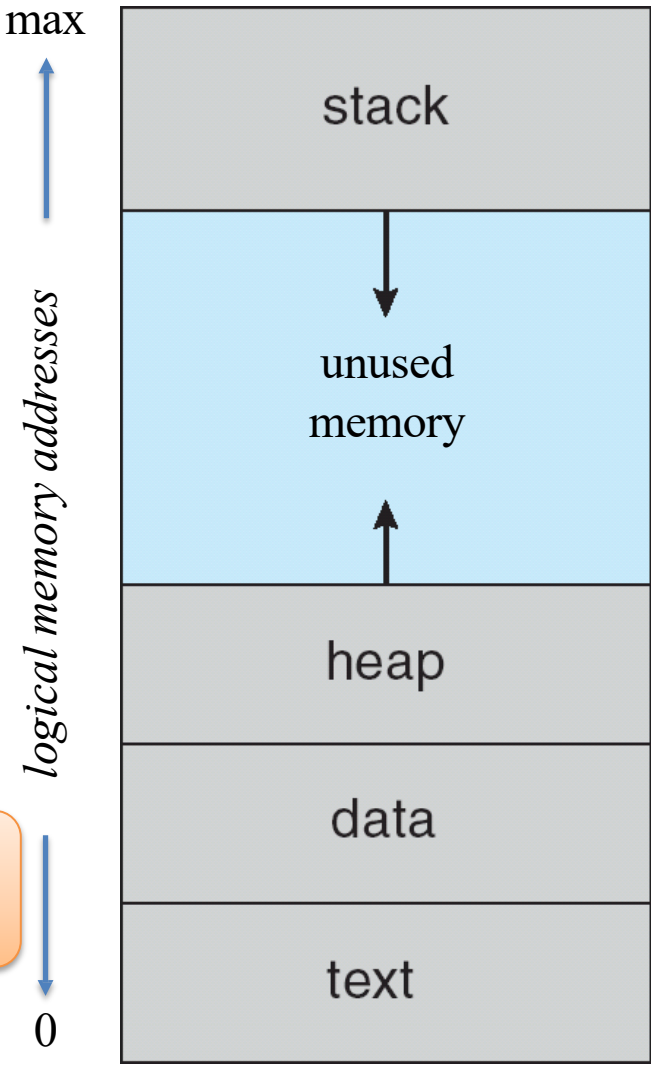
You should be able to answer those questions once we learn memory management.

Process Address Space

- *Process address space* is all locations addressable by the process
  - Each process has its own address space
- Code (*Text*)
- Global Data (*Data*)
- Dynamic Data (*Heap*)
  - Grows up
- Local Data (*Stack*)
  - Grows down

This is the memory space that is presented by OS to each process

Why call this "presented"? Does it mean it is not real?

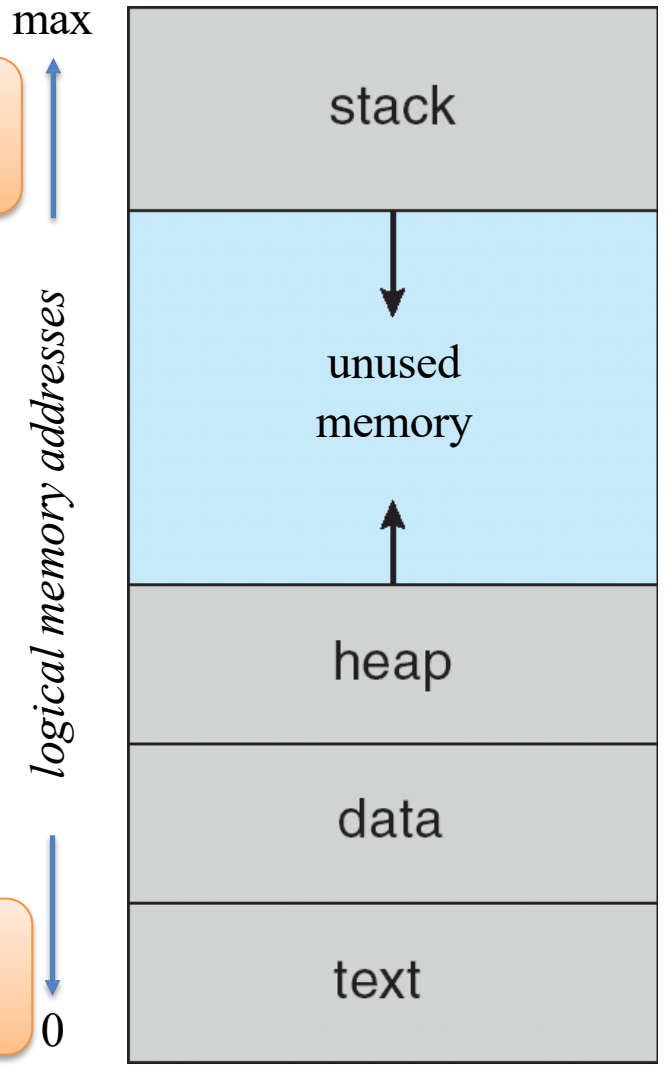


Process Address Space

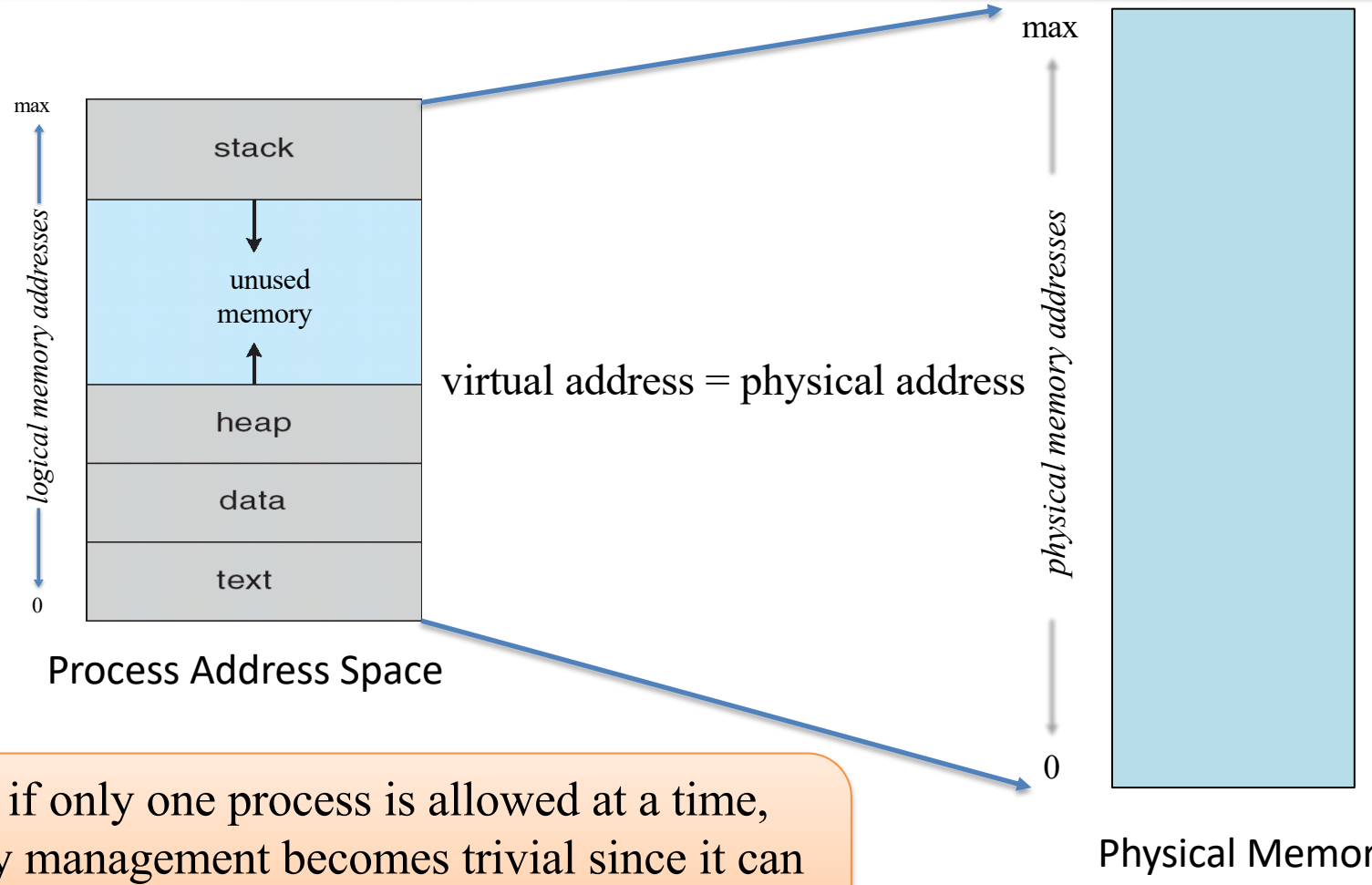
This is the memory space that is presented by OS to each process

- Programmers do not want to manage memory sharing
- Process address space
  - The only space programmers and processes can see
  - Always logically contiguous (0 to max)
  - Each byte is identified by a logical memory address (a.k.a virtual address)

Note: logical memory address = virtual address



What if we only have one process?

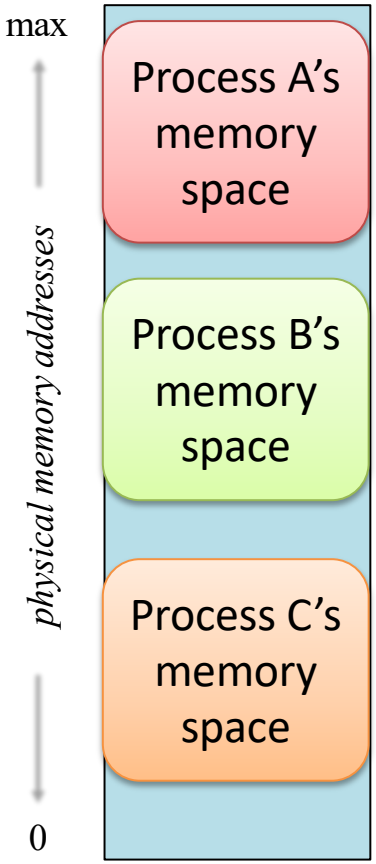


Note: if only one process is allowed at a time, memory management becomes trivial since it can occupy the entire physical memory. A virtual address can be 1:1 mapped to a physical memory.

If this is the case, we lost multiprogramming…. ☹

We need to support multiprogramming

Illustration of contiguous memory management:

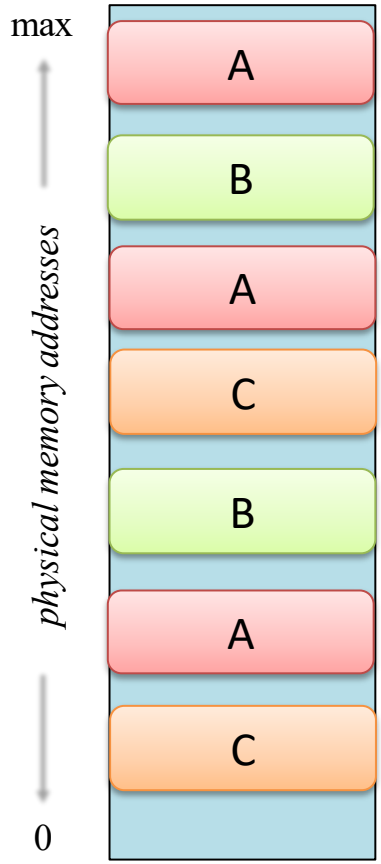


Two general types of memory management schemes

- Contiguous
  - All logical memory used by a process is mapped to a single physical memory region with contiguous addresses
- Non-contiguous
  - Portions of the logical memory **used** by a process can be mapped to multiple physical memory regions
- Allocation and memory management go hand in hand
  - with the hardware support

We need to support multiprogramming

Illustration of non-contiguous memory management:

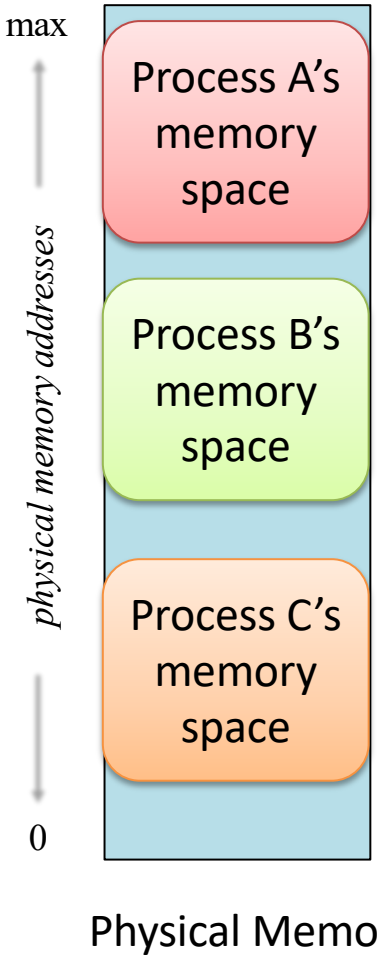


Two general types of memory management schemes

- Contiguous
  - All logical memory used by a process is mapped to a single physical memory region with contiguous addresses
- Non-contiguous
  - Portions of the logical memory **used** by a process can be mapped to multiple physical memory regions
- Allocation and memory management go hand in hand
  - with the hardware support

Deep Dive: Contiguous Memory Management

Illustration of contiguous memory management:



Two general types of memory management schemes

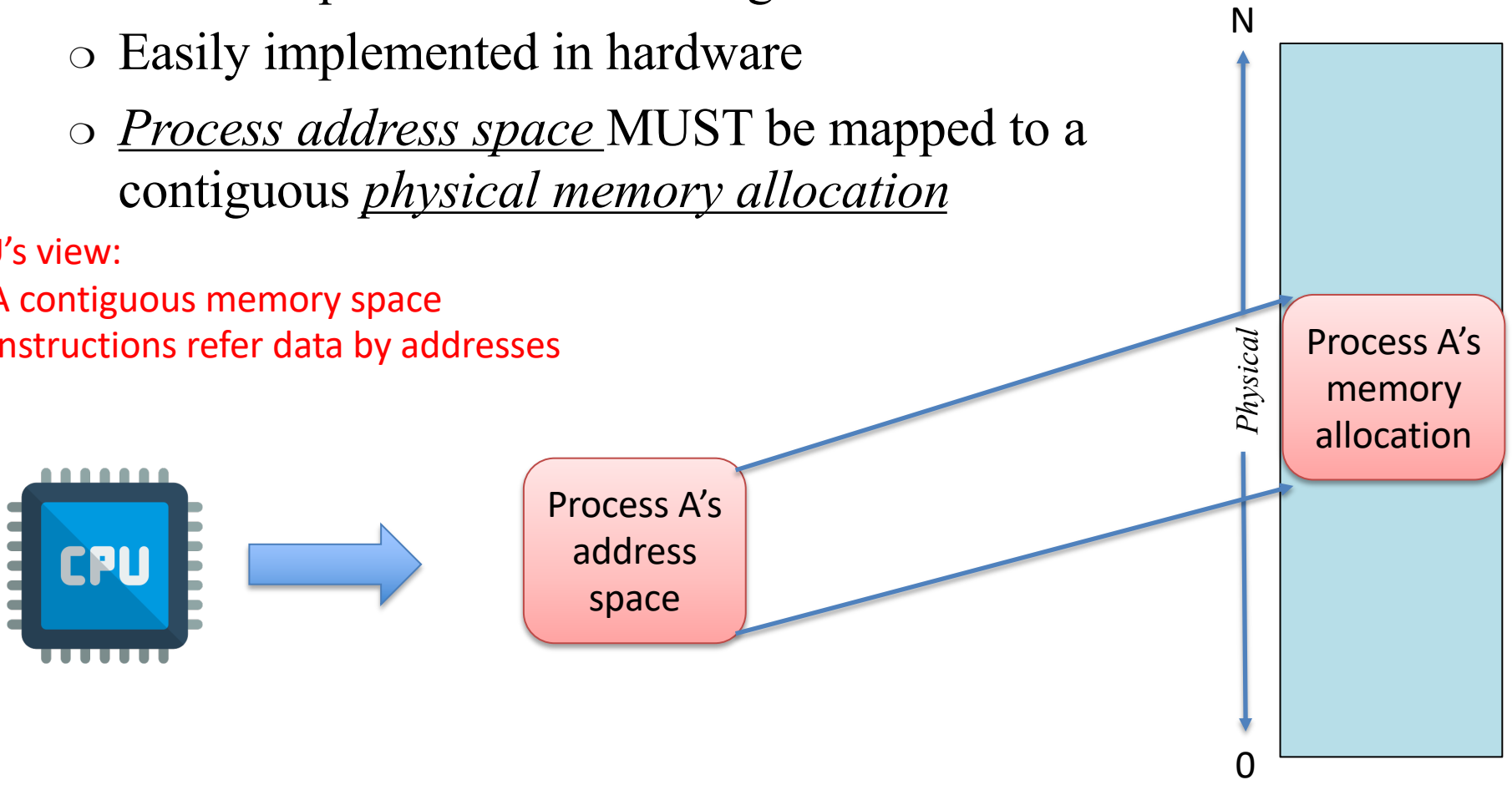
- Contiguous
  - All logical memory used by a process is mapped to a single physical memory region with contiguous addresses
- Non-contiguous
  - Portions of the logical memory **used** by a process can be mapped to multiple physical memory regions
- Allocation and memory management go hand in hand
  - with the hardware support

Contiguous Allocation and Memory Management

- Find a physical region for each process
  - Less complex for OS to manage
  - Easily implemented in hardware
  - *Process address space* MUST be mapped to a contiguous *physical memory allocation*

CPU's view:

- A contiguous memory space
- Instructions refer data by addresses

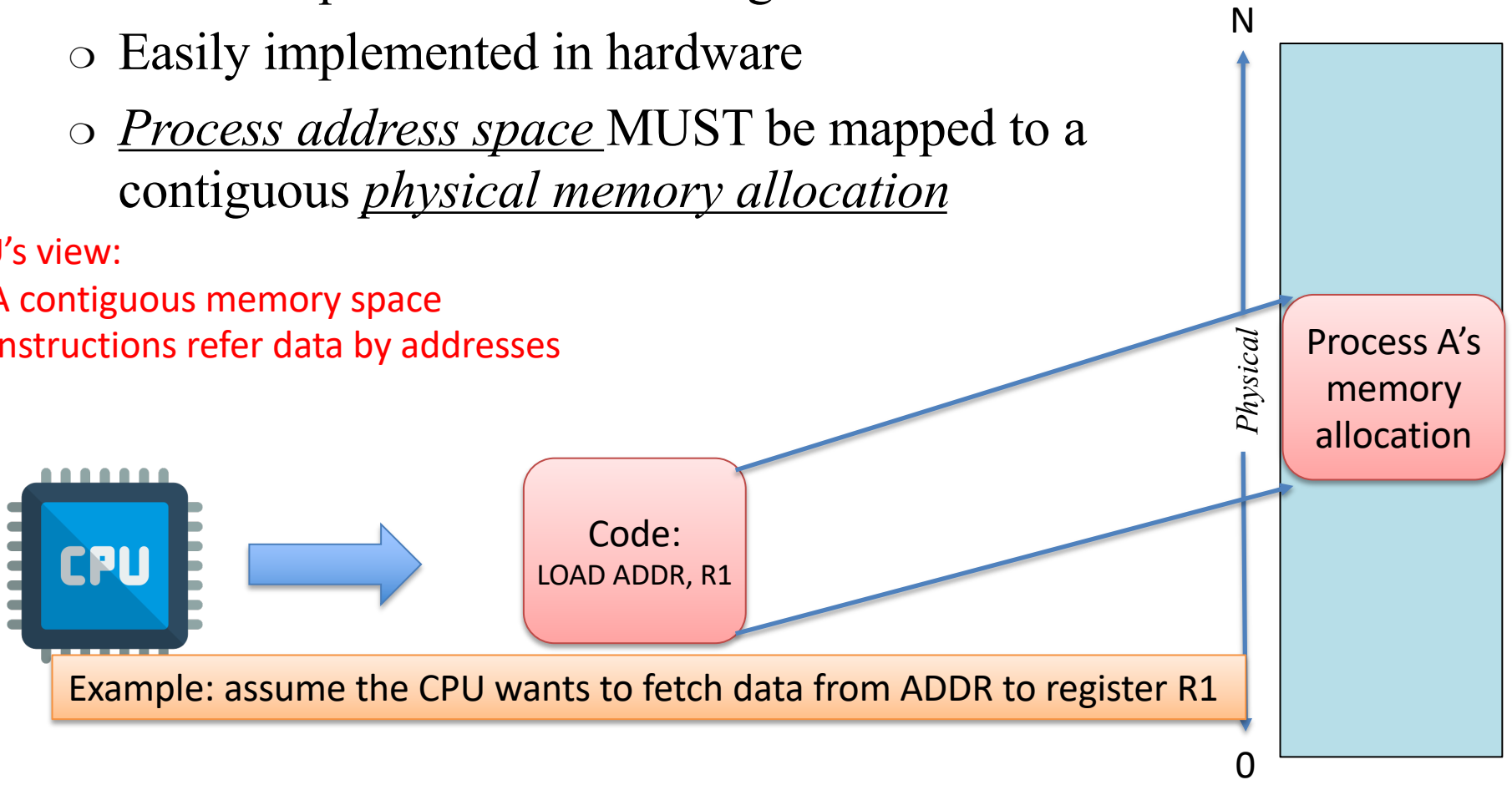


Contiguous Allocation and Memory Management

- Find a physical region for each process
  - Less complex for OS to manage
  - Easily implemented in hardware
  - *Process address space* MUST be mapped to a contiguous *physical memory allocation*

CPU's view:

- A contiguous memory space
- Instructions refer data by addresses

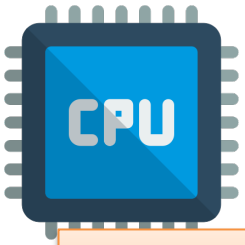


Contiguous Allocation and Memory Management

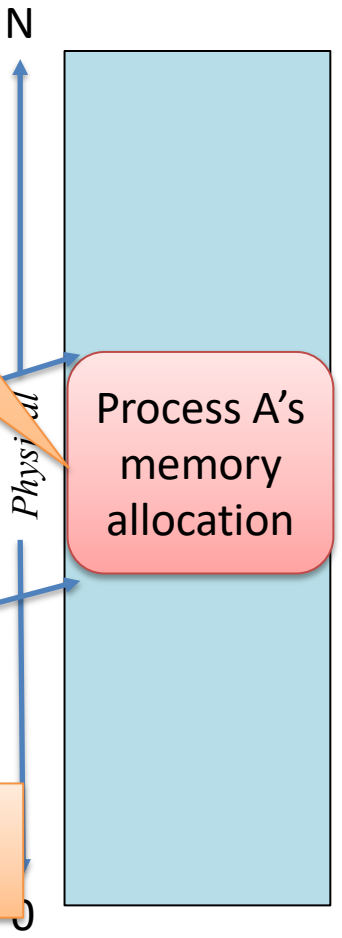
How to satisfy both physical memory and programmers' need?

Physical memory: *I only know about physical memory addresses, so your load request must use physical addresses*

Process address space: *I was created by loading from an executable, which was compiled from source code. The programmer does not know the physical address. So, I can only put virtual addresses in the instructions.*



Code:
LOAD ADDR, R1



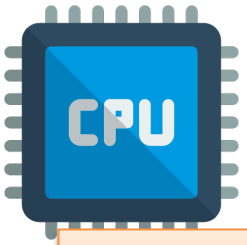
Example: assume the CPU wants to fetch data from ADDR to register R1
What should the ADDR refer to? Virtual or physical address space?

Contiguous Allocation and Memory Management

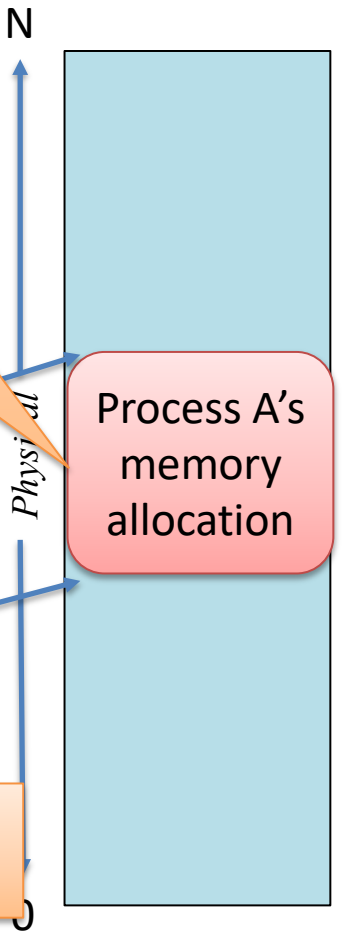
How to satisfy both physical memory and programmers' need? --> Address binding (see next few slides)

Physical memory: *I only know about physical memory addresses, so your load request must use physical addresses*

Process address space: *I was created by loading from an executable, which was compiled from source code. The programmer does not know the physical address. So, I can only put virtual addresses in the instructions.*

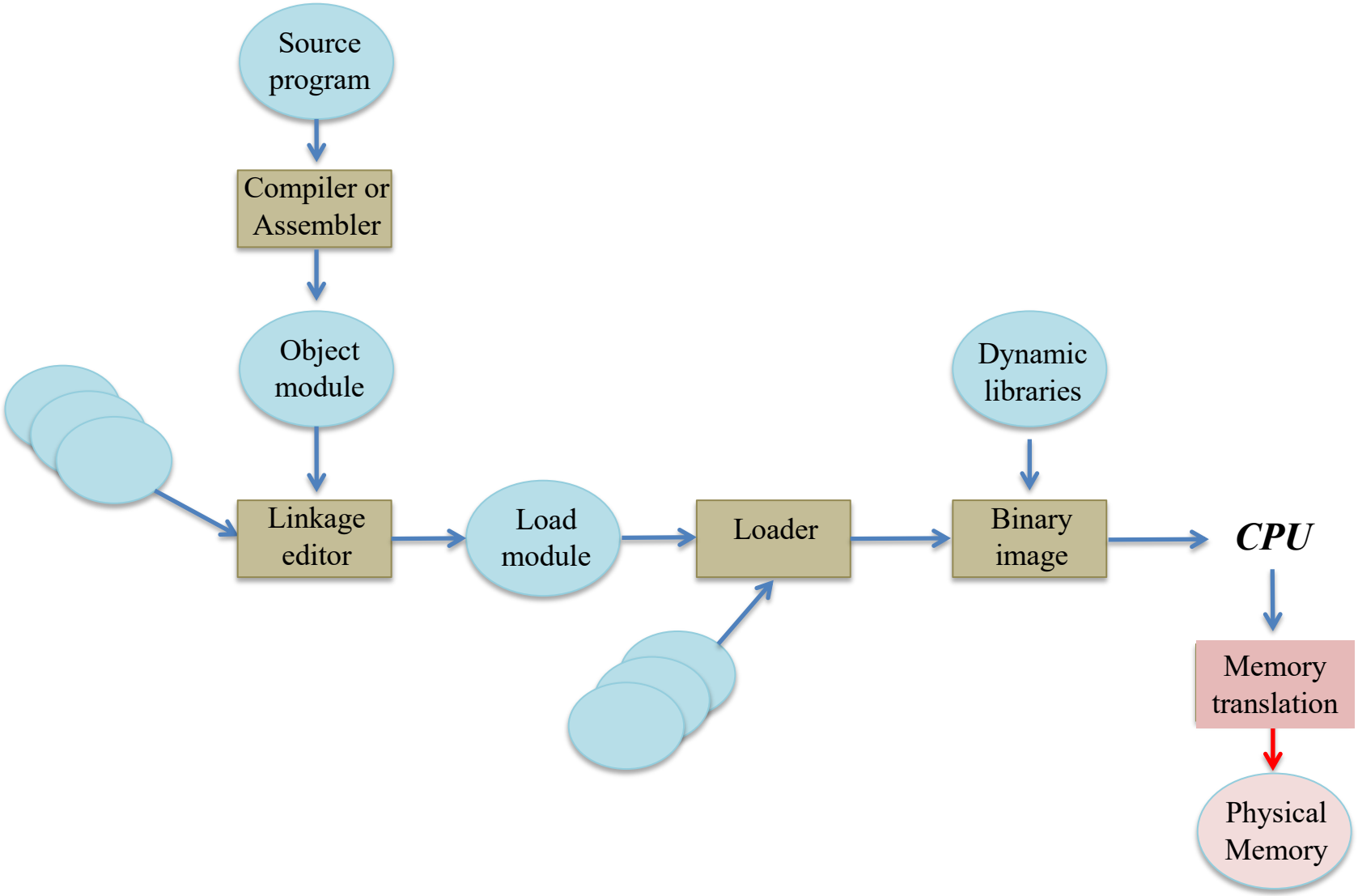


Code:
LOAD ADDR, R1

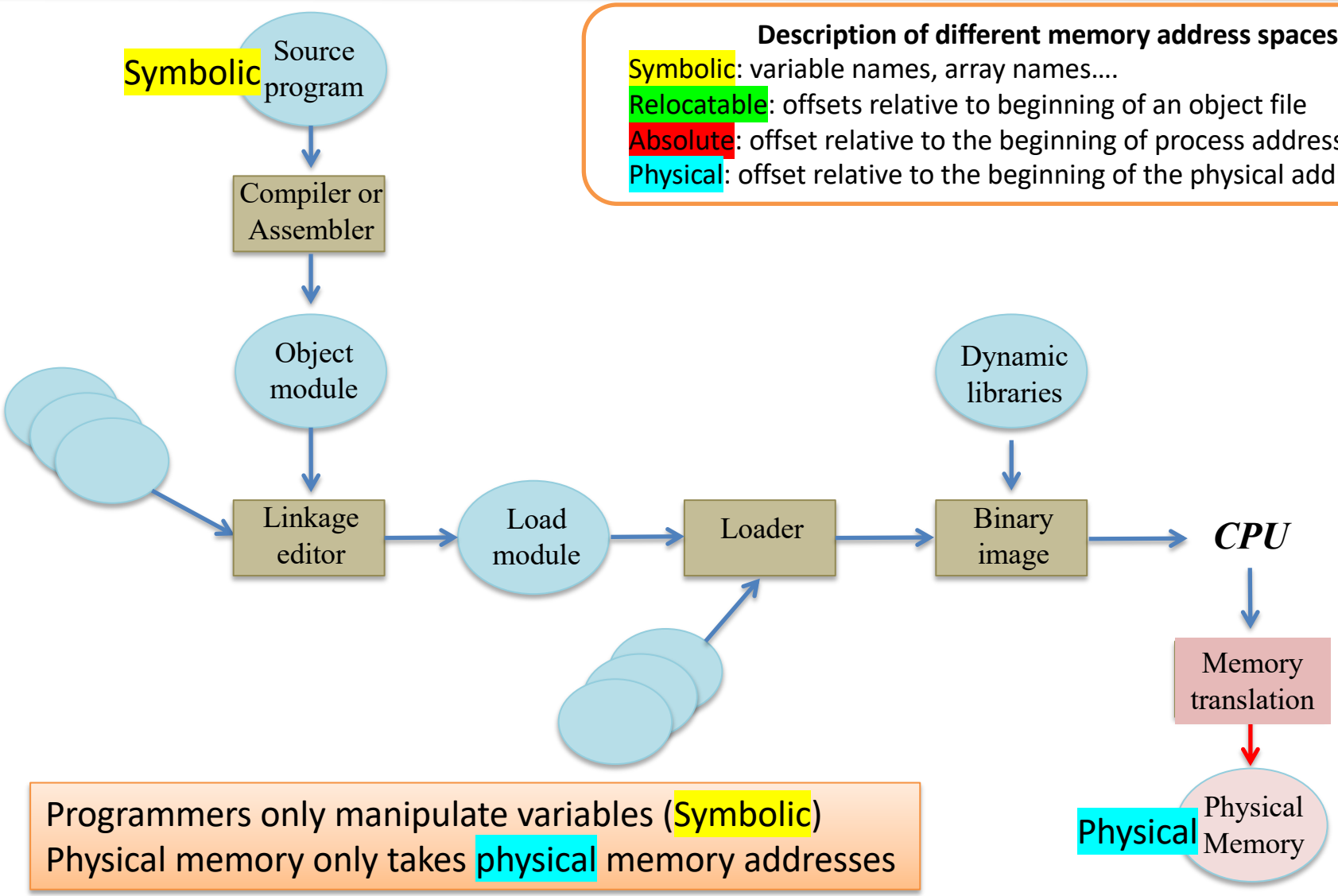


Example: assume the CPU wants to fetch data from ADDR to register R1
What should the ADDR refer to? Virtual or physical address space?

Multistep Processing of a User Program



When does address binding happen?



Description of different memory address spaces

Symbolic: variable names, array names….

Relocatable: offsets relative to beginning of an object file

Absolute: offset relative to the beginning of process address space

Physical: offset relative to the beginning of the physical address space

Programmers only manipulate variables (Symbolic)

Physical memory only takes physical memory addresses

Address Binding

- A program goes through several "translation" steps before producing an executable code image
  - Program executable "loaded" into memory
- A program starts with "logical" addresses that are in the "logical" address space its process
  - Instructions produce addresses that reference the logical process address space (i.e., logical addresses)
    - sometime called "virtual" addresses (not related to virtual memory)
  - Need to *bind* (i.e., map between) "logical" addresses and "physical" addresses in memory
- All this depends on both software and hardware
  - Software includes compilers, loaders, and OS
  - Hardware include memory system, address translation, …

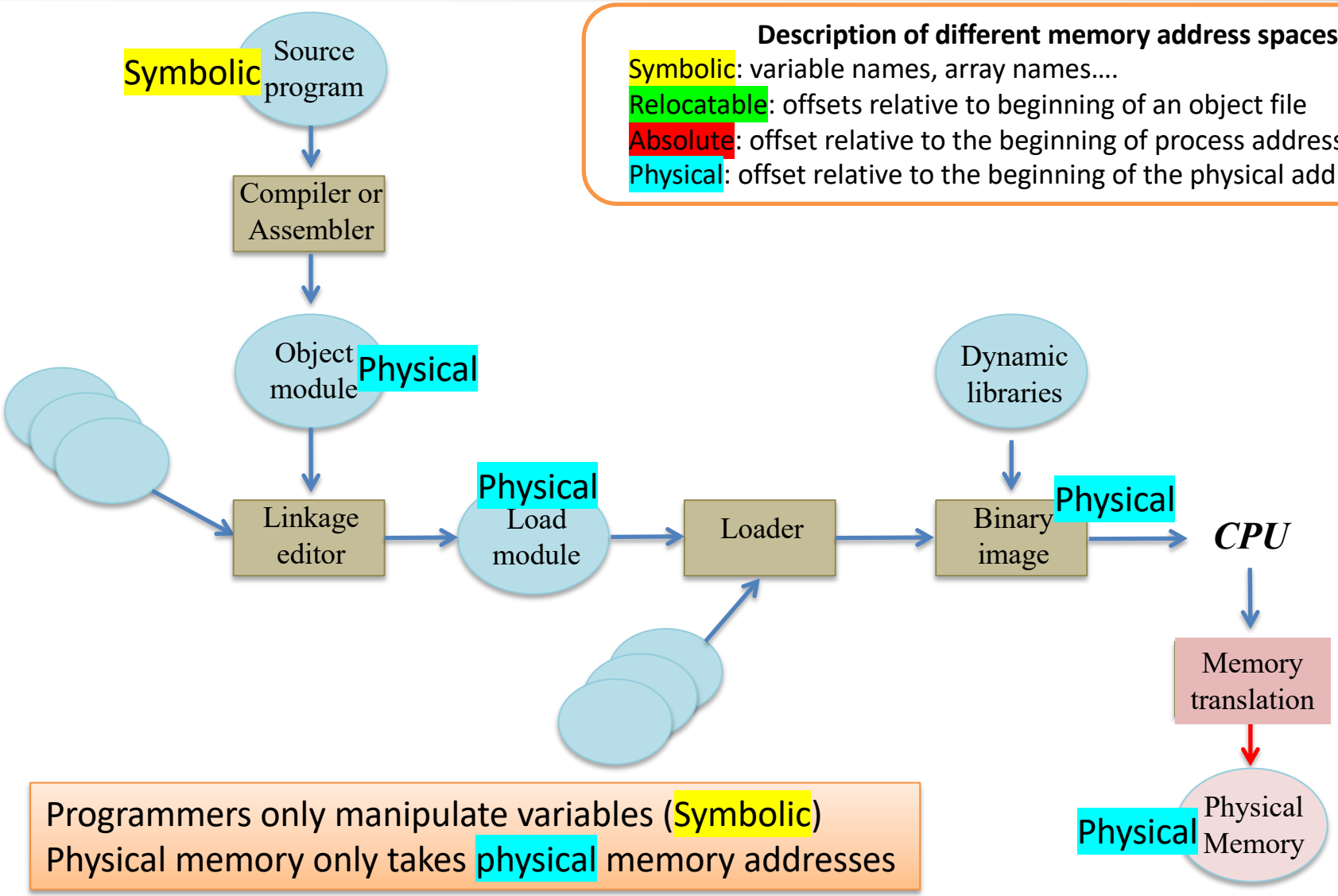
Memory and Address Binding

Description of different memory address spaces

Symbolic: variable names, array names….
Relocatable: offsets relative to beginning of an object file
Absolute: offset relative to the beginning of process address space
Physical: offset relative to the beginning of the physical address space

- When does address binding (address translation) happen?
  - *Compile time*: If we know where the process will be placed in physical memory before compilation. Compiler can directly bind to a physical address.
  - *Link/Load time*: If we know where the process will be placed before we linking/loading. Compiler can first generate relocatable code, and linker/loader can bind relocatable to physical.
  - *Execution time*: Address binding happens for each memory request. Absoluate address is used in a loaded process, and translated to physical online by hardware

Compile time binding



Description of different memory address spaces

Symbolic: variable names, array names….

Relocatable: offsets relative to beginning of an object file

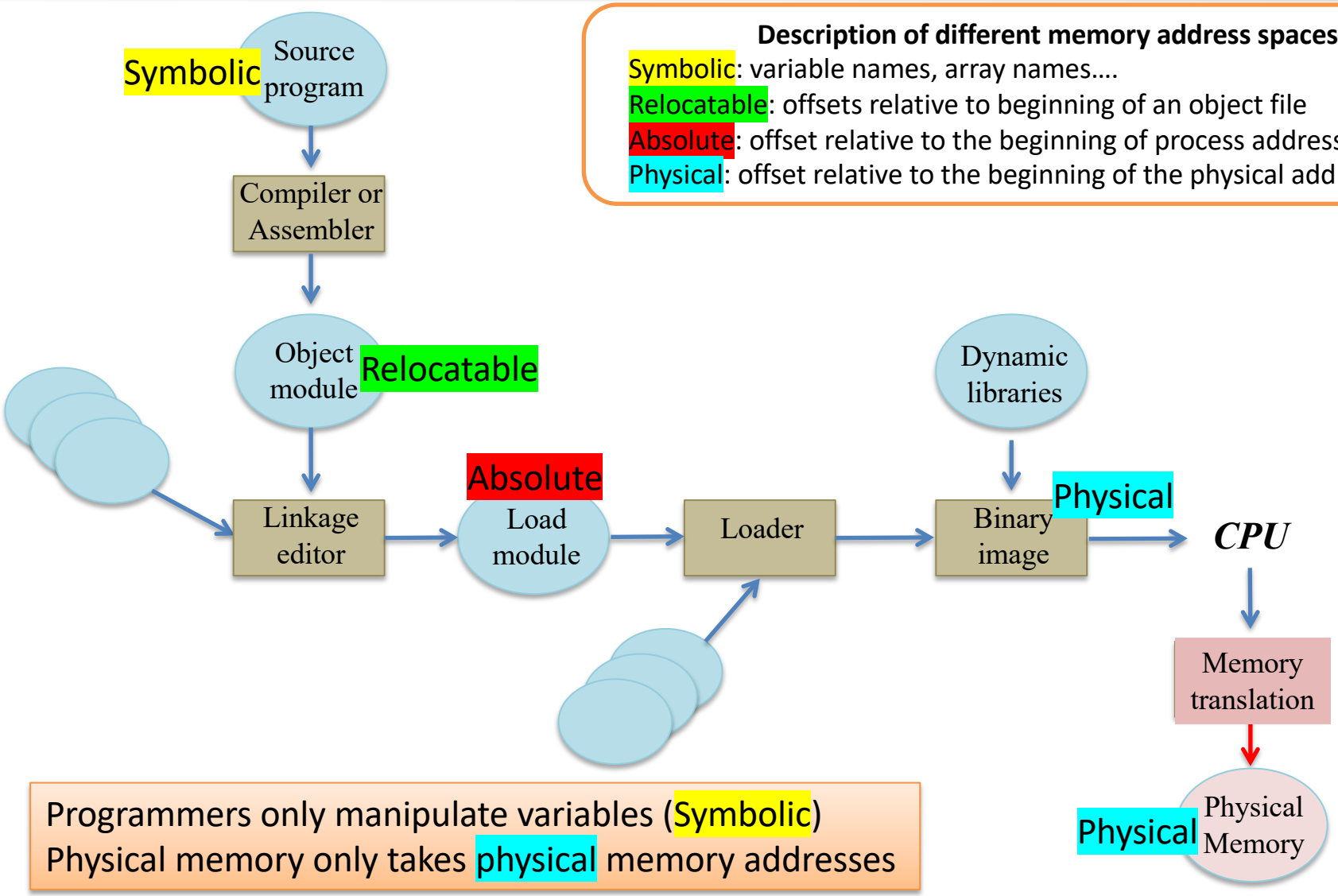
Absolute: offset relative to the beginning of process address space

Physical: offset relative to the beginning of the physical address space

Programmers only manipulate variables (Symbolic)

Physical memory only takes physical memory addresses

Load time binding



Description of different memory address spaces

Symbolic: variable names, array names….

Relocatable: offsets relative to beginning of an object file

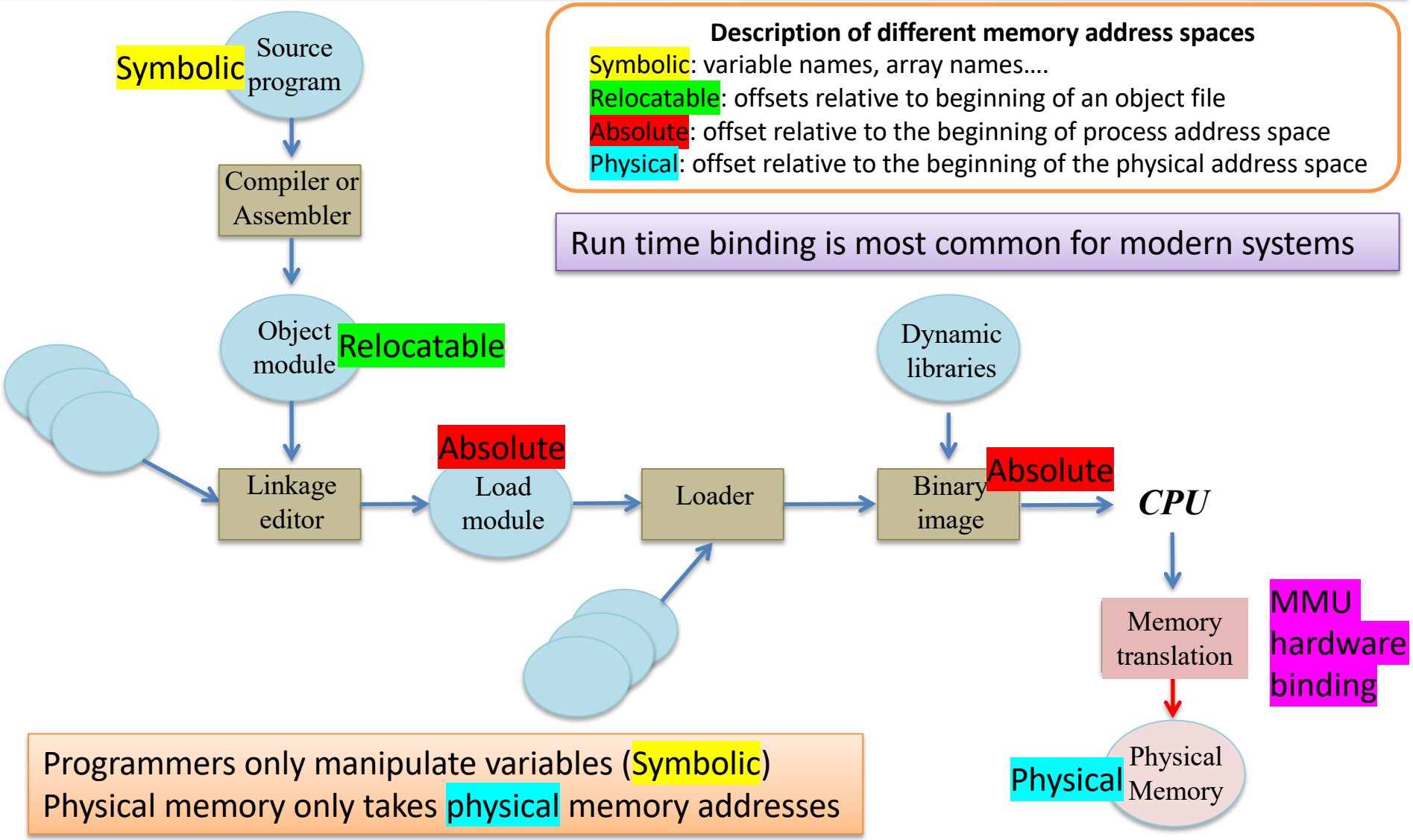
Absolute: offset relative to the beginning of process address space

Physical: offset relative to the beginning of the physical address space

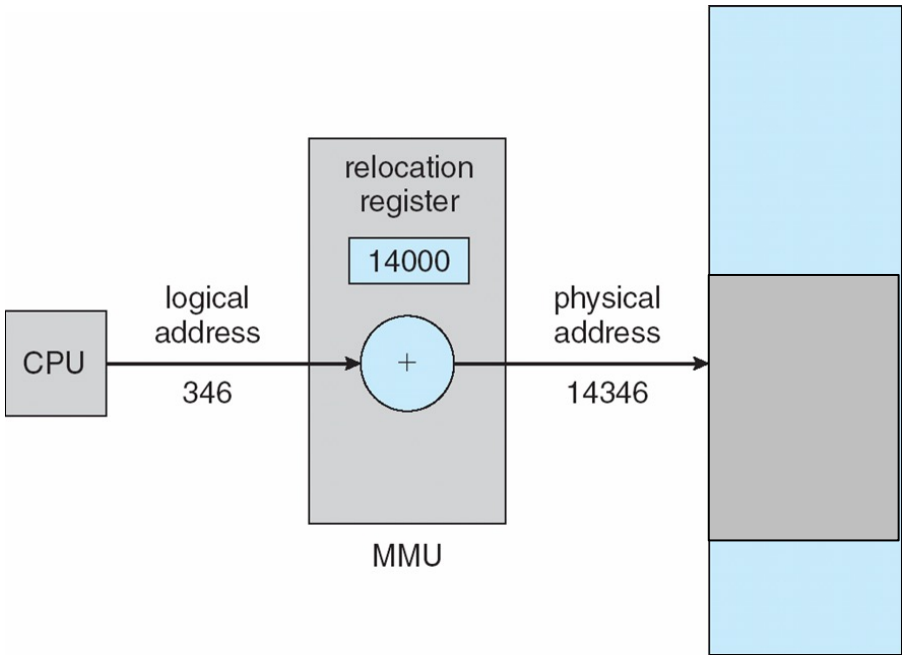
Programmers only manipulate variables (Symbolic)

Physical memory only takes physical memory addresses

Run time binding



Dynamic Relocation via Relocation Register

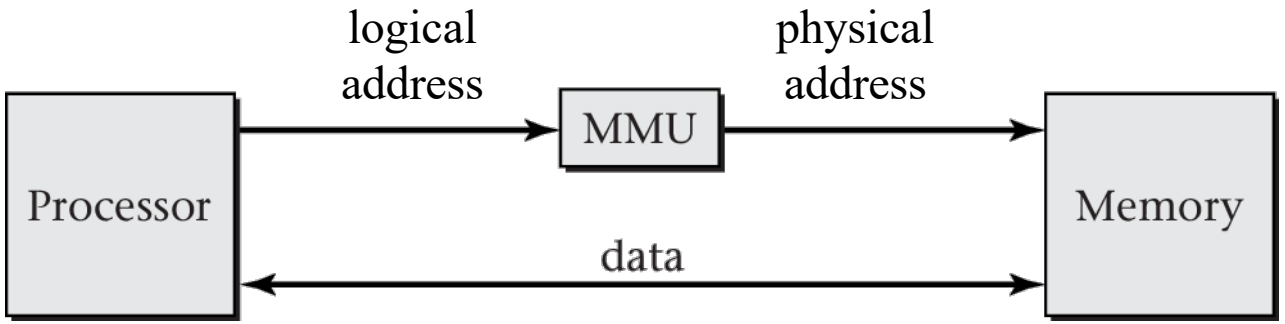


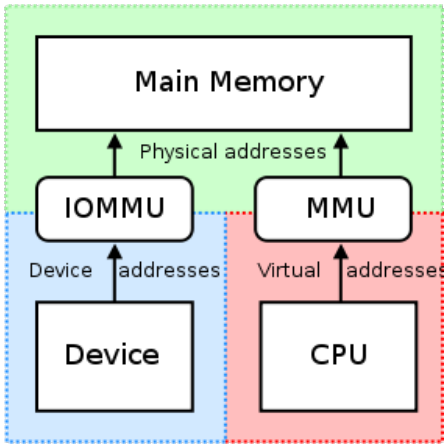
- Logical (virtual) address is converted to a physical address by adding the value in the relocation register.
- Each process has its only relocation register
- Memory management unit (MMU) hardware supported
  - Little support from the OS is required
  - Used in coordination with base/limit registers (will introduce this)

What should context switch do about relocation register?

Memory Management Unit

- Hardware device that maps a logical (aka, virtual for reasons discussed later) address to a physical address
- How it does this is at the heart of all memory management schemes
- Memory management unit (*MMU*)
  changes the logical address into a physical address

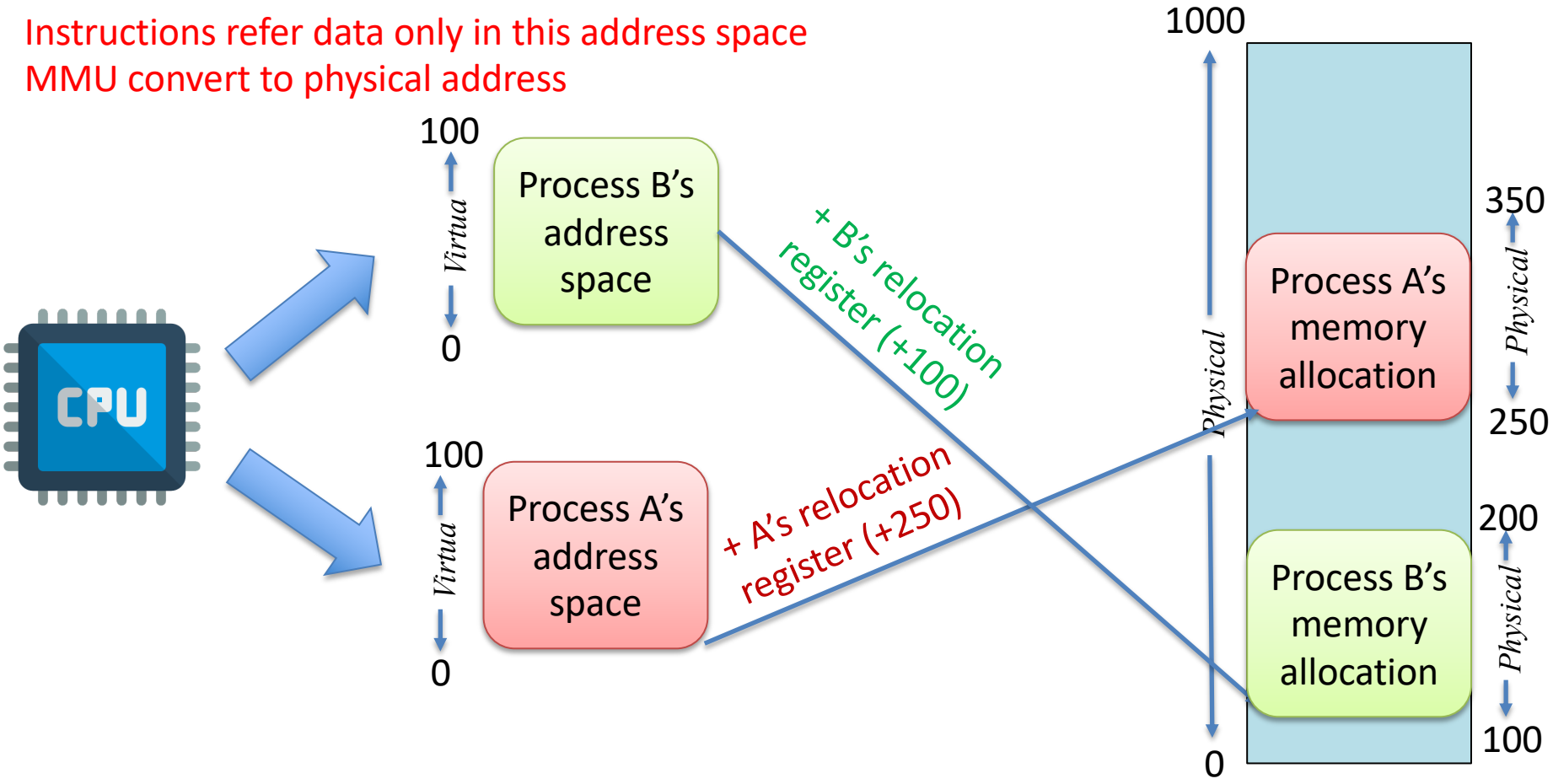




Contiguous Allocation and Memory Management

CPU/programmer has a consistent memory view of all processes:

- A contiguous memory space
- Address: 0 – M
- Instructions refer data only in this address space
- MMU convert to physical address



Logical vs. Physical Address Space

Takeaway:

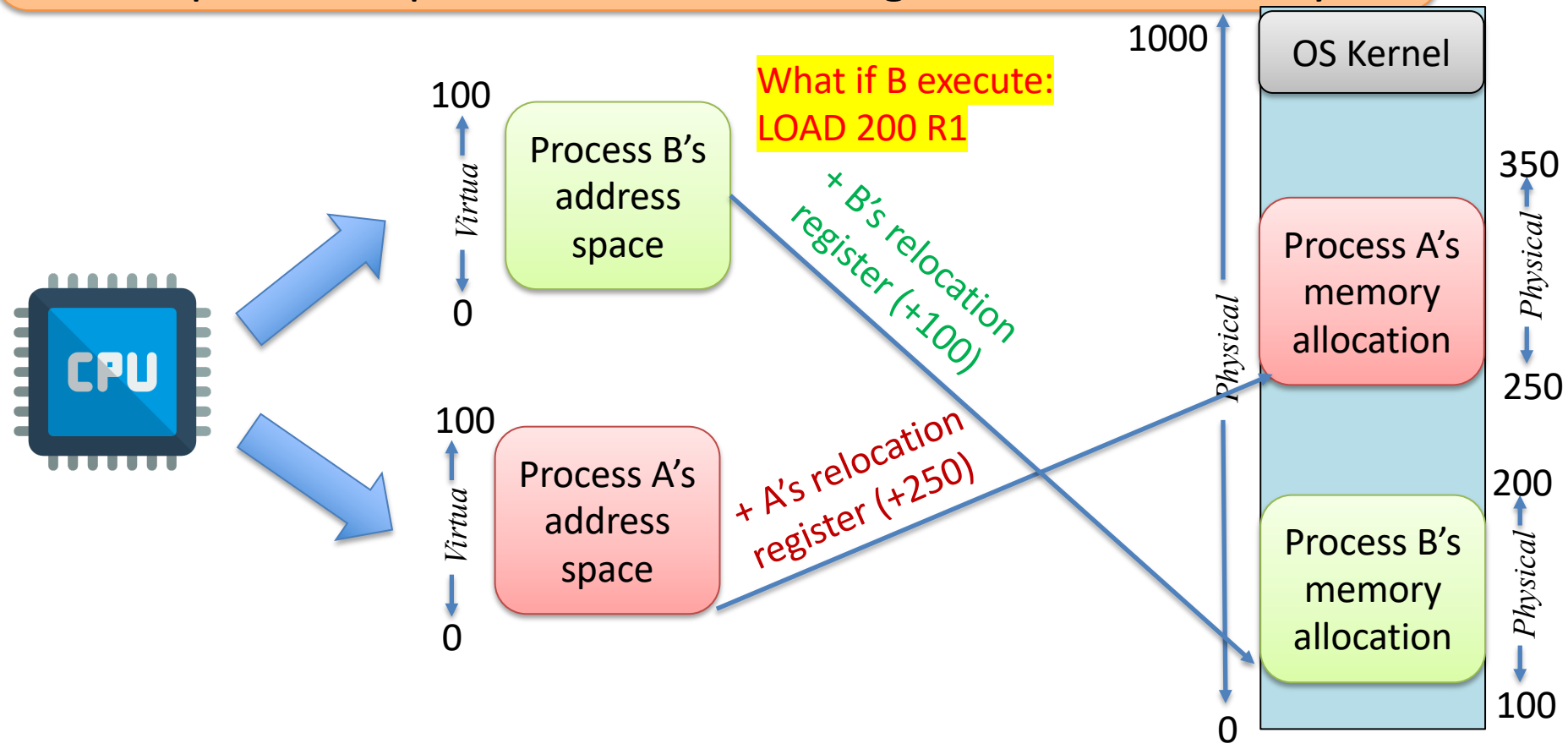
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes
- Logical and physical addresses MUST be different at execution time … Why?
  - Hardware-based address binding schemes here must do this
  - They must be fast!

Dynamic Linking

- *Static linking* – system libraries and program code combined by the loader into the binary program image
- *Dynamic linking* –linking postponed until execution time
- OS checks if routine is in the processes memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries (Why?)
  - System libraries loaded when the program starts are also known as *shared libraries*
- Consider dynamic linking use to patching system libraries
  - Versioning may be needed

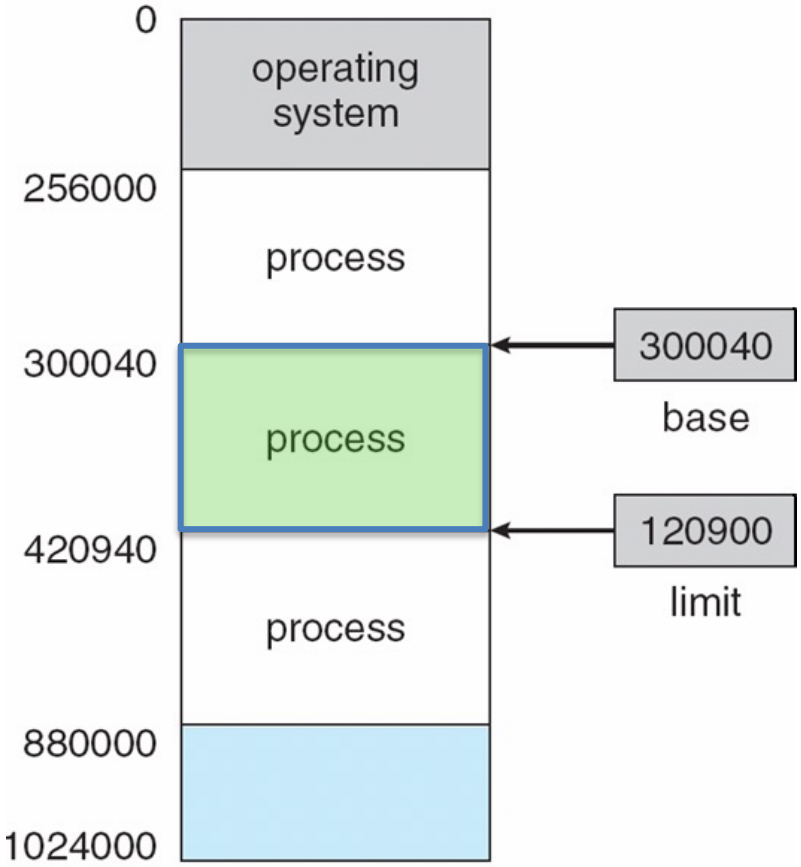
Contiguous Allocation: Memory Protection

How to prevent one process from accessing another process memory allocation?
How to prevent a process from accessing the kernel memory?



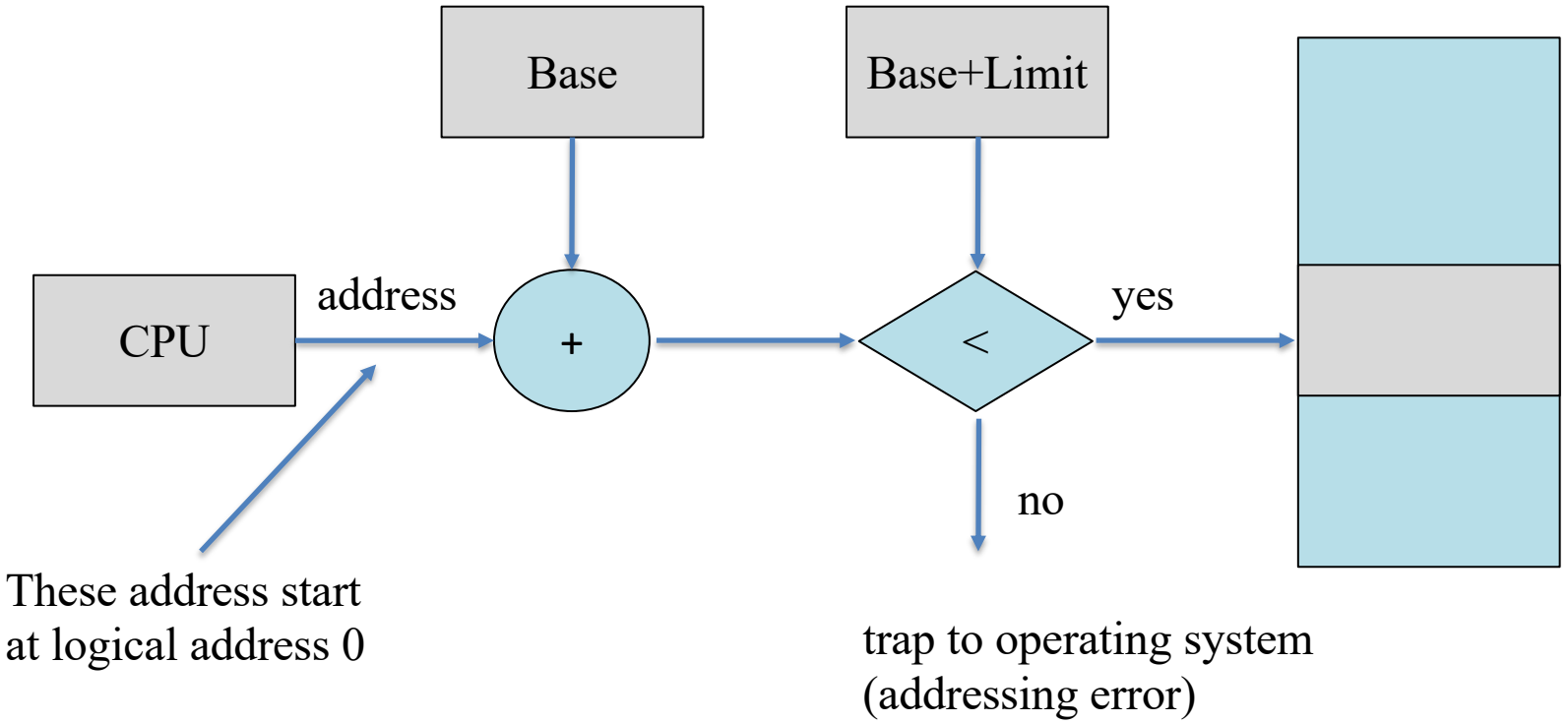
Base and Limit Registers with Partitioning

- How to translate the *logical address space*
  - A process is given a pair of *base* and *limit registers*
  - *Base register = relocation register*
- CPU must check every memory access generated by the process to be sure it is between base and limit (i.e., in the allocated partition)
  - If not, an exception occurs
- Need hardware support for doing this … Why?
- Must map entire process address space into a fixed address region … Why?
- OS sets up base/limit registers when process is created



Hardware Address Translation

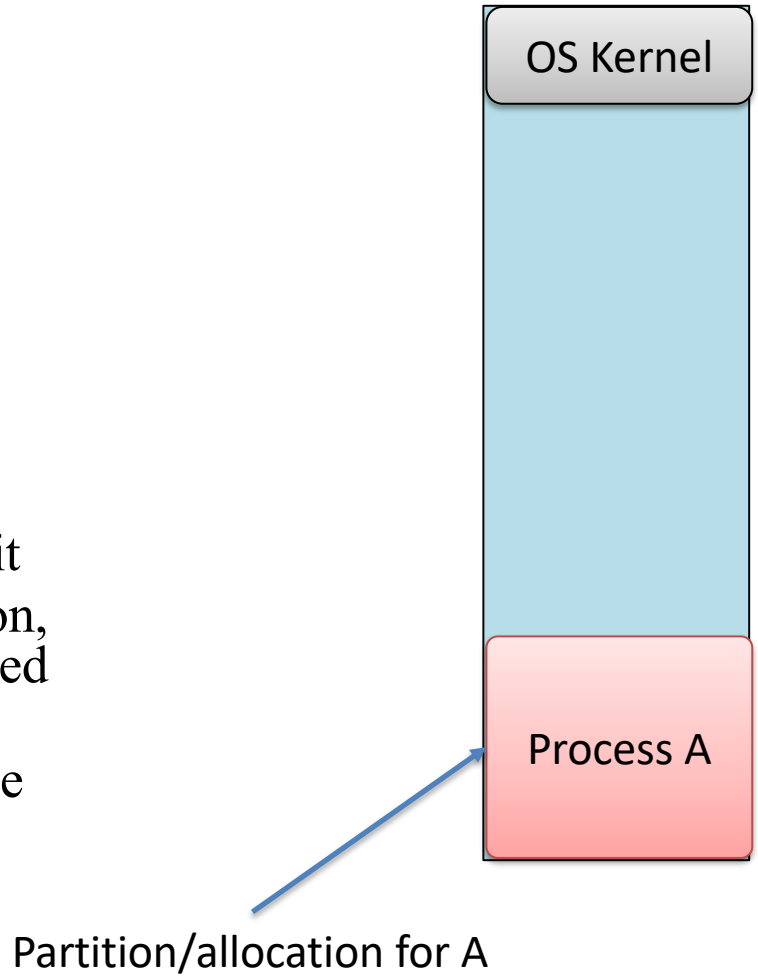
- If the OS did the check, it would be very slow
- Assume contiguous logical addresses start at 0



(Note: Figure 9.2 in book is confusing.)

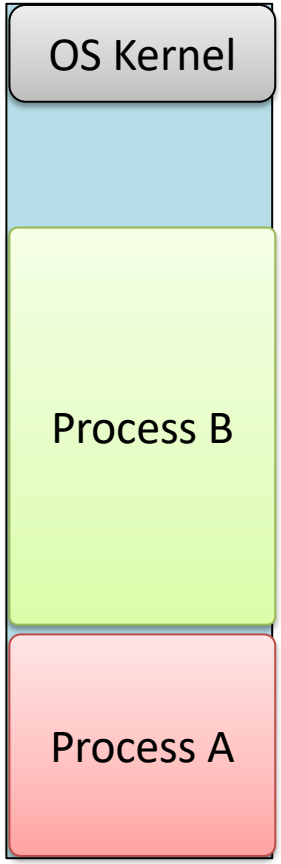
Contiguous Allocation: Multi-process Partition

- One process per partition
  - Degree of multiprogramming limited by #partitions
  - Variable-partition sizes for efficiency
    - sized to a given process' needs
  - Free partitions various size are scattered throughout memory (*holes*)
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about allocated/free partitions



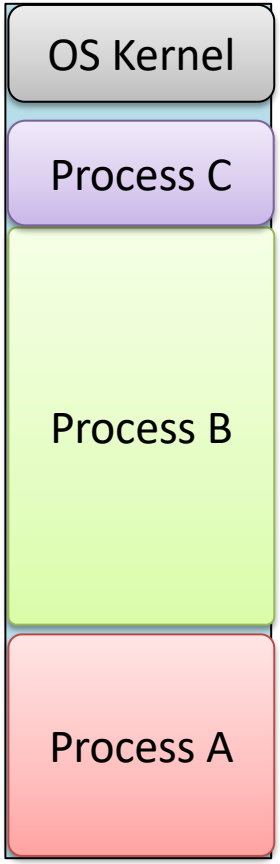
Contiguous Allocation: Multi-process Partition

- One process per partition
  - Degree of multiprogramming limited by #partitions
  - Variable-partition sizes for efficiency
    - sized to a given process' needs
  - Free partitions various size are scattered throughout memory (*holes*)
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about allocated/free partitions



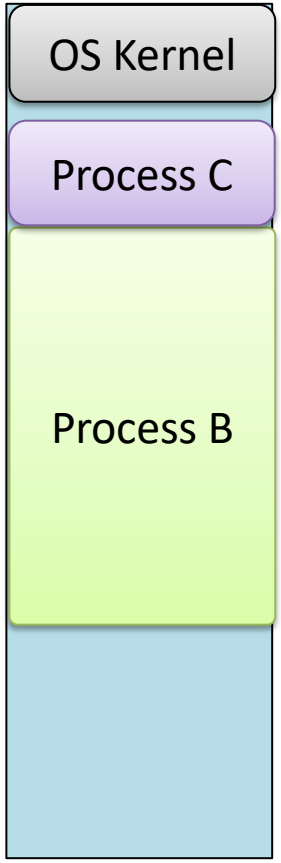
Contiguous Allocation: Multi-process Partition

- One process per partition
  - Degree of multiprogramming limited by #partitions
  - Variable-partition sizes for efficiency
    - sized to a given process' needs
  - Free partitions various size are scattered throughout memory (*holes*)
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about allocated/free partitions



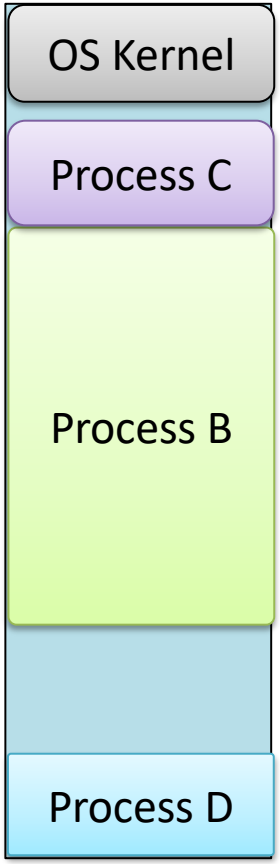
Contiguous Allocation: Multi-process Partition

- One process per partition
  - Degree of multiprogramming limited by #partitions
  - Variable-partition sizes for efficiency
    - sized to a given process' needs
  - Free partitions various size are scattered throughout memory (*holes*)
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about allocated/free partitions



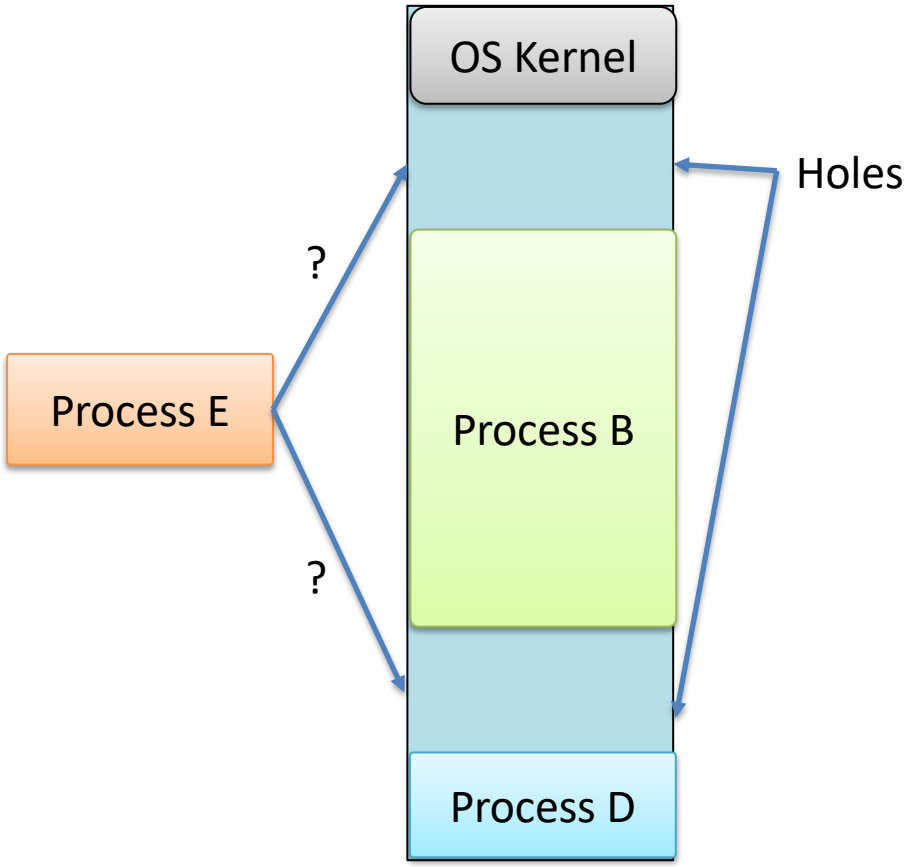
Contiguous Allocation: Multi-process Partition

- One process per partition
  - Degree of multiprogramming limited by #partitions
  - Variable-partition sizes for efficiency
    - sized to a given process' needs
  - Free partitions various size are scattered throughout memory (*holes*)
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about allocated/free partitions



Contiguous Allocation: Multi-process Partition

- Where to fit a new process to a list of holes?



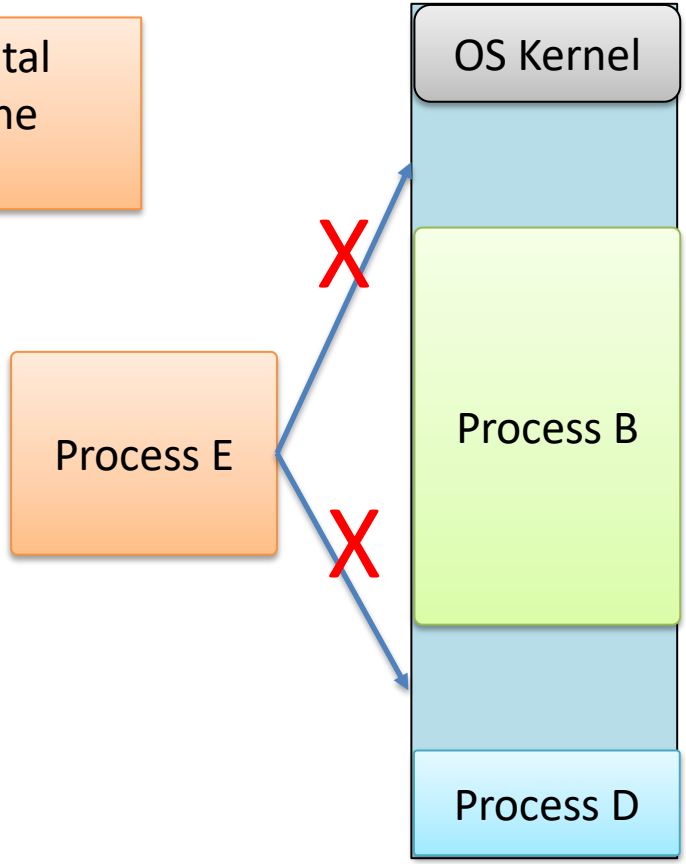
Dynamic Storage Allocation Problem

- How to satisfy a request of size *n?*
  - Assume there is a list of free 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; must also search entire list
  - Produces the largest leftover hole
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Contiguous Allocation: Fragmentation Issue

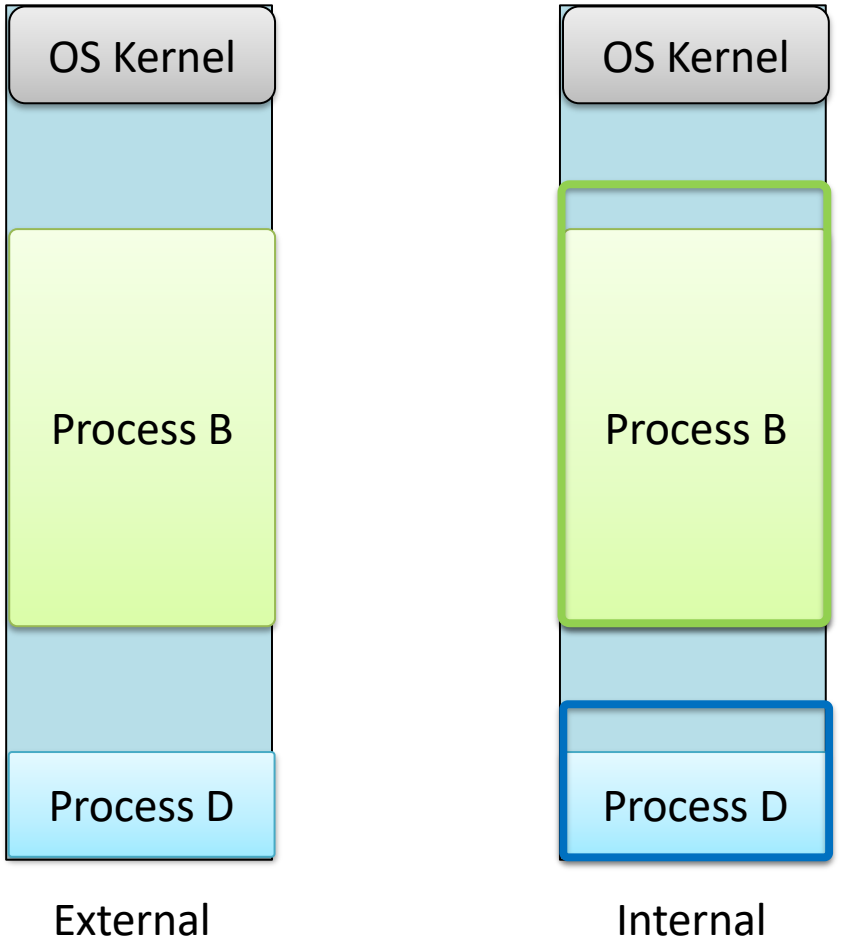
Fragmentation Issue: there is enough total memory space to satisfy a request but the available spaces are not contiguous.

- Waste memory space
- Decrease degree of multiprogramming
- First/best fit is better for some systems
- External fragmentation will be a problem (regardless of which algorithm is used)



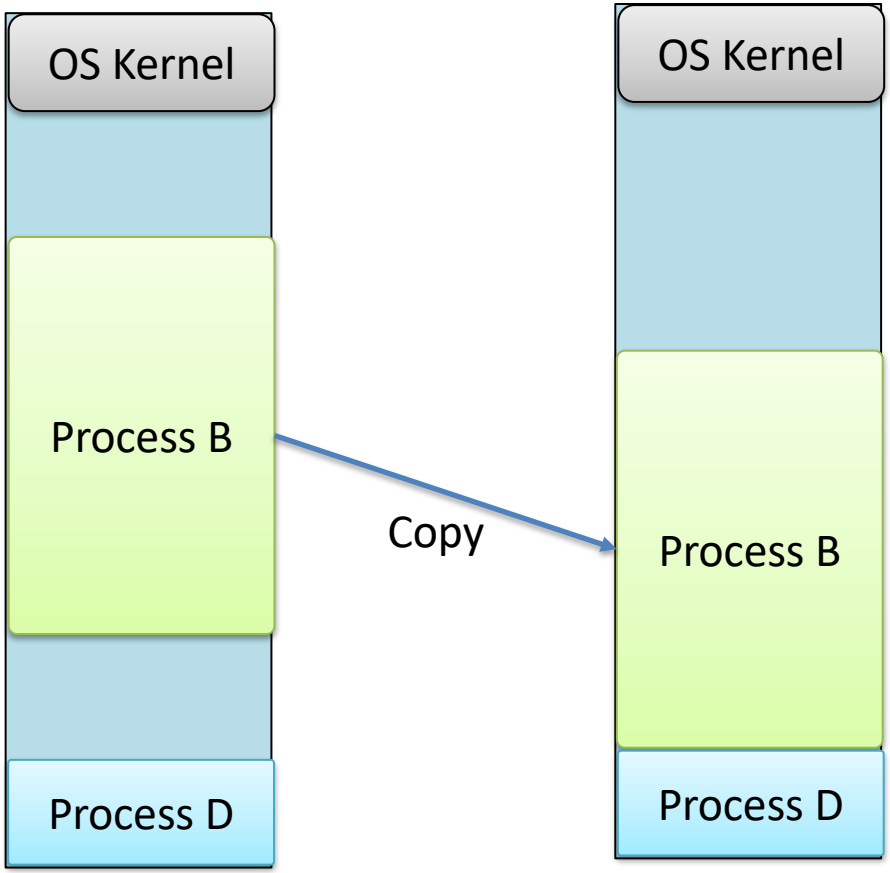
Fragmentation

- *External fragmentation*
  - *T*otal memory space exists to satisfy a request, but it is not contiguous
- *Internal fragmentation*
  - *A*llocated memory may be slightly larger than requested memory
  - This size difference is memory internal to a partition, but not being used



External is often a more serious problem

Fragmentation Improvement: Compaction



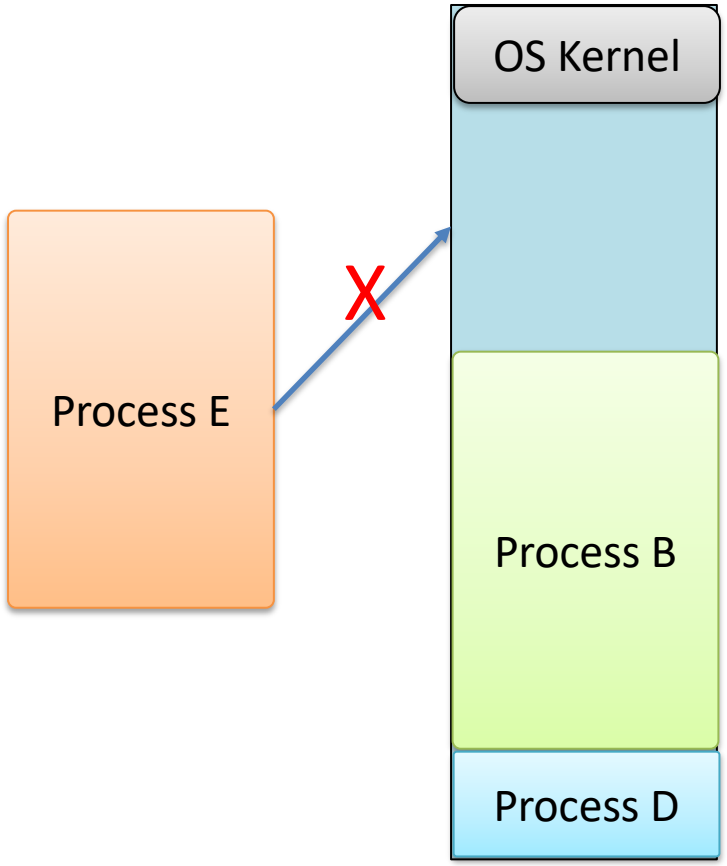
- Reduce external fragmentation by *compaction*
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time
  - Cost is issue (memory copy)

Summary of Contiguous Memory Allocation

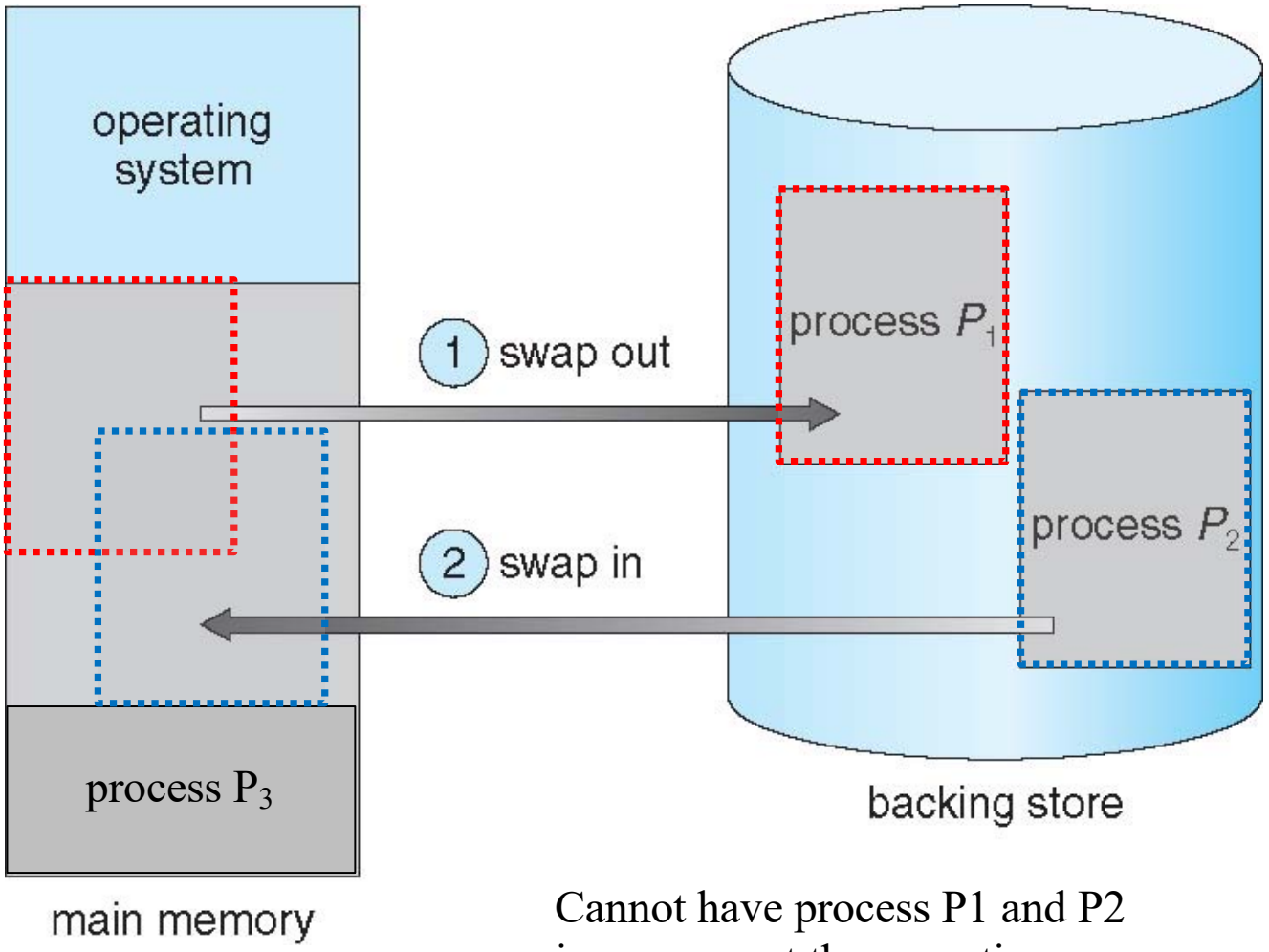
- Easy to implement
- Used in old systems
- External fragmentation issue cannot be easily handled

Swapping (now for something interesting)

- Physical memory space is finite
- Suppose we allocate physical memory to meet the needs of a process as discussed
  - What happens when the total physical memory space requested by processes exceeds physical memory?
- *Swapping* is the idea of sharing physical memory space between 2 or more processes such that only 1 process uses the space at any time
- Still think about this with respect to contiguous memory allocation for a process



Schematic View of Swapping



Cannot have process P1 and P2 in memory at the same time

Swapping Support

- A process can be *swapped* temporarily out of memory
  - Moved to a *backing store (e.g.,* disk)
  - Later brought back into memory for continued execution
- Can be used with scheduling
  - *Roll out, roll in* – swapping based on priority
  - Remember medium-term scheduler?
- Major cost
  - Total transfer time is directly proportional to the amount of memory swapped
- System maintains a ready queue of ready-to-run processes which have memory images on disk

Note: Swap is usually disable by modern OS. Can be enabled with configuration

Can swapping used with short-term scheduler?

Does the swapped out process need to swap back in to same physical addresses?

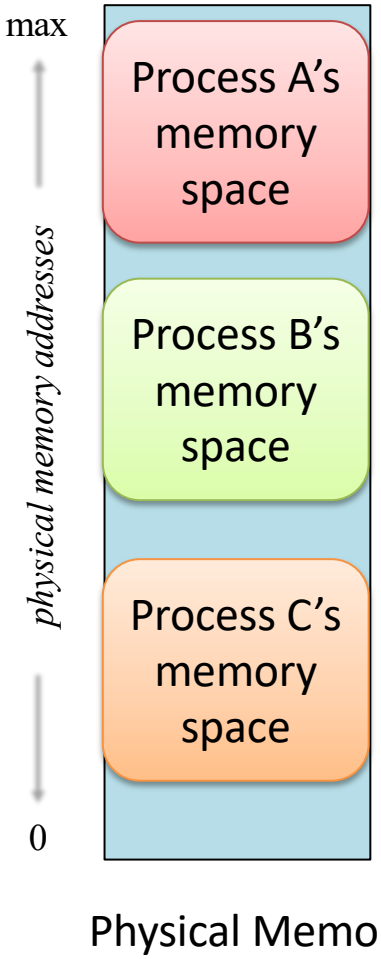
Context Switch Time

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Swapping time is included in context switch time
- Context switch time can then be very high
  - 100MB process swapping to hard disk
  - Transfer rate of 50MB/sec
    - swap out time of 2000 ms
    - plus swap in of same sized process
    - total context switch swapping component time of 4000ms

Cost of context switch for medium term scheduler

Deep Dive: Contiguous Memory Management

Illustration of contiguous memory management:

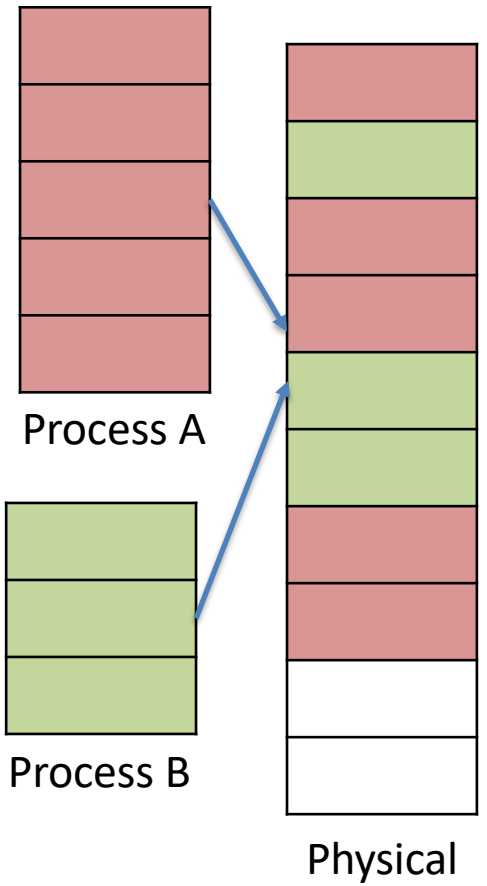


Two general types of memory management schemes

- Contiguous
  - All logical memory used by a process is mapped to a single physical memory region with contiguous addresses
- Non-contiguous
  - Portions of the logical memory **used** by a process can be mapped to multiple physical memory regions
- Allocation and memory management go hand in hand
  - with the hardware support

Page-based Memory Management (1)

- Divide physical memory into fixed-sized blocks: called *frames*
  - Size is power of 2 (usually 512 bytes to 4K bytes)
- Divide logical memory into fixed-sized blocks: called *pages*
  - Same size as frames
- Frame are allocated to pages
  - OS tracks frame-to-page assignment and all free frames
- To run a program of size *N* pages
  - Find *N* free frames and assign them to *N* pages
  - Load program



Frames assigned do not have to be contiguous
Avoids external fragmentation!

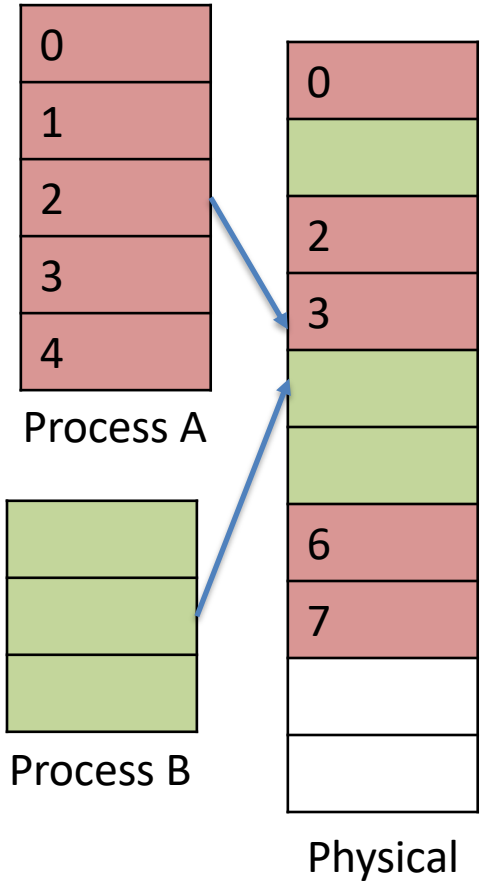
Page-based Memory Management (2)

- All pages in logical memory need to have frames
- Need to keep track of frame-to-page allocation
- Set up a *page table*
  - Used to map logical pages to physical frames

Page table for process A:

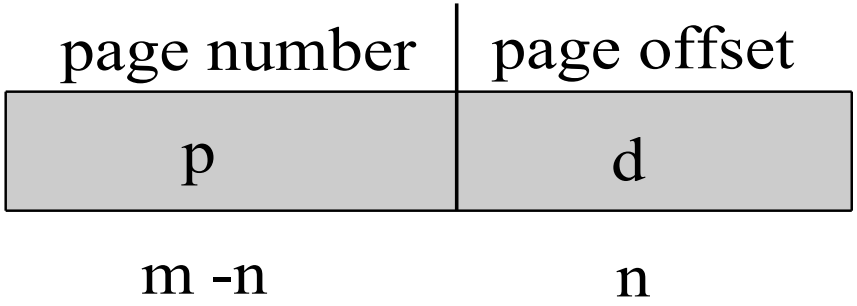
Page id	Frame id
0	0
1	2
2	3
3	6
4	7

Can you figure out what does page table for process B look like?



Page Address Translation Scheme

- Address generated by CPU 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)* – combined with base address to define the physical memory address that is sent to the memory unit

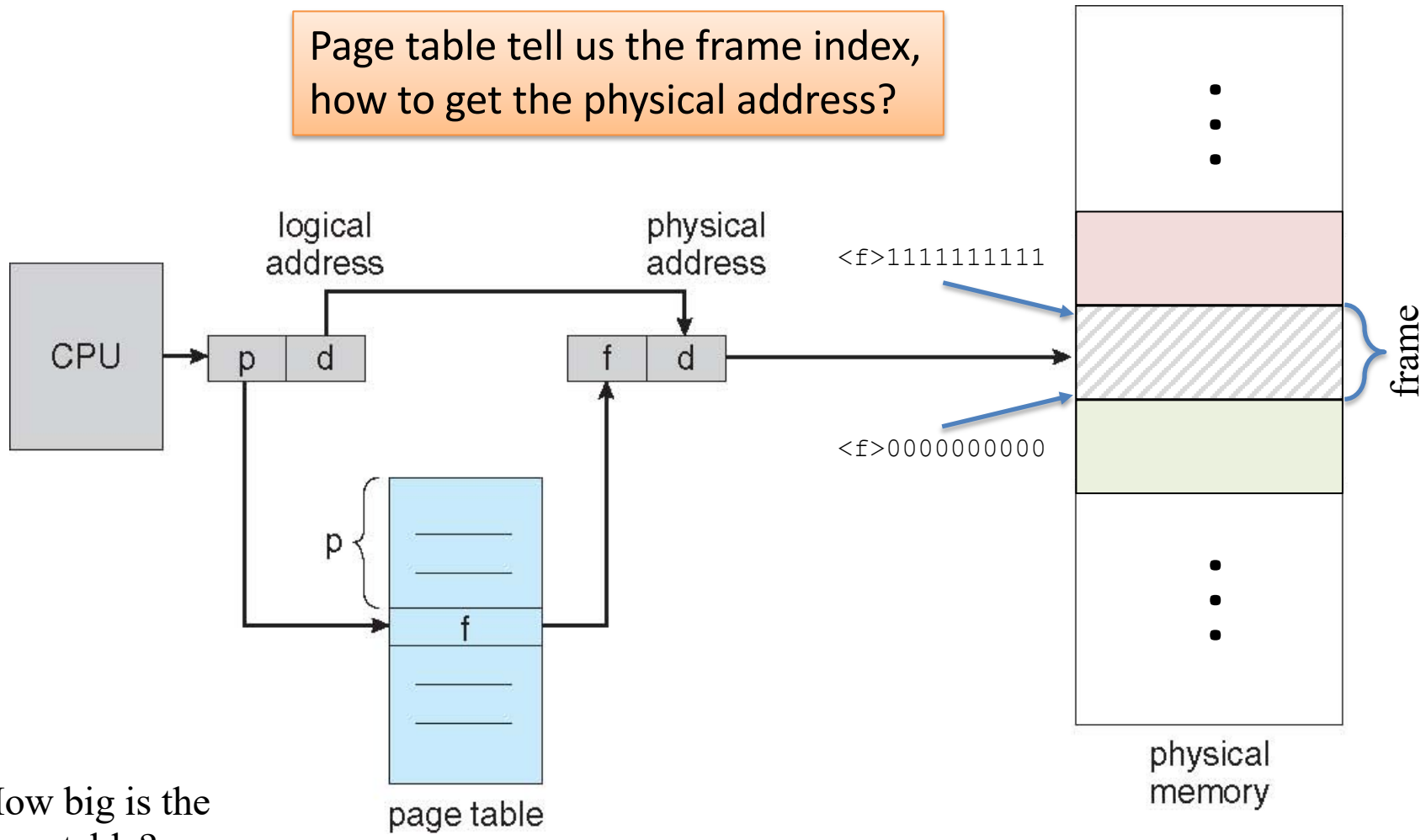


  - For given logical address space 2^m and page size 2^n

What if the page/frame size is not a power of 2?

Paging Hardware (Address Translation)

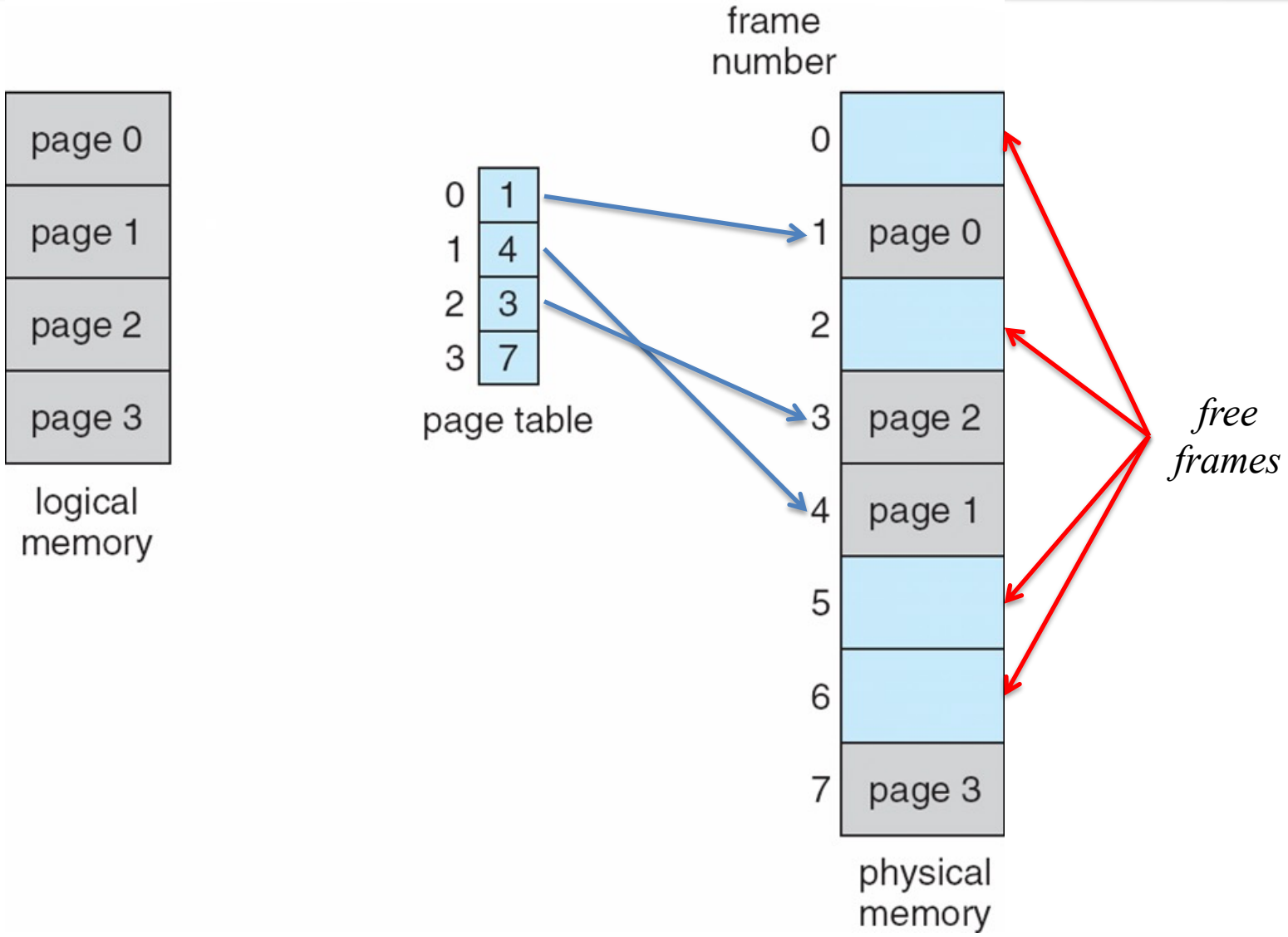
Page table tell us the frame index, how to get the physical address?



How big is the page table?

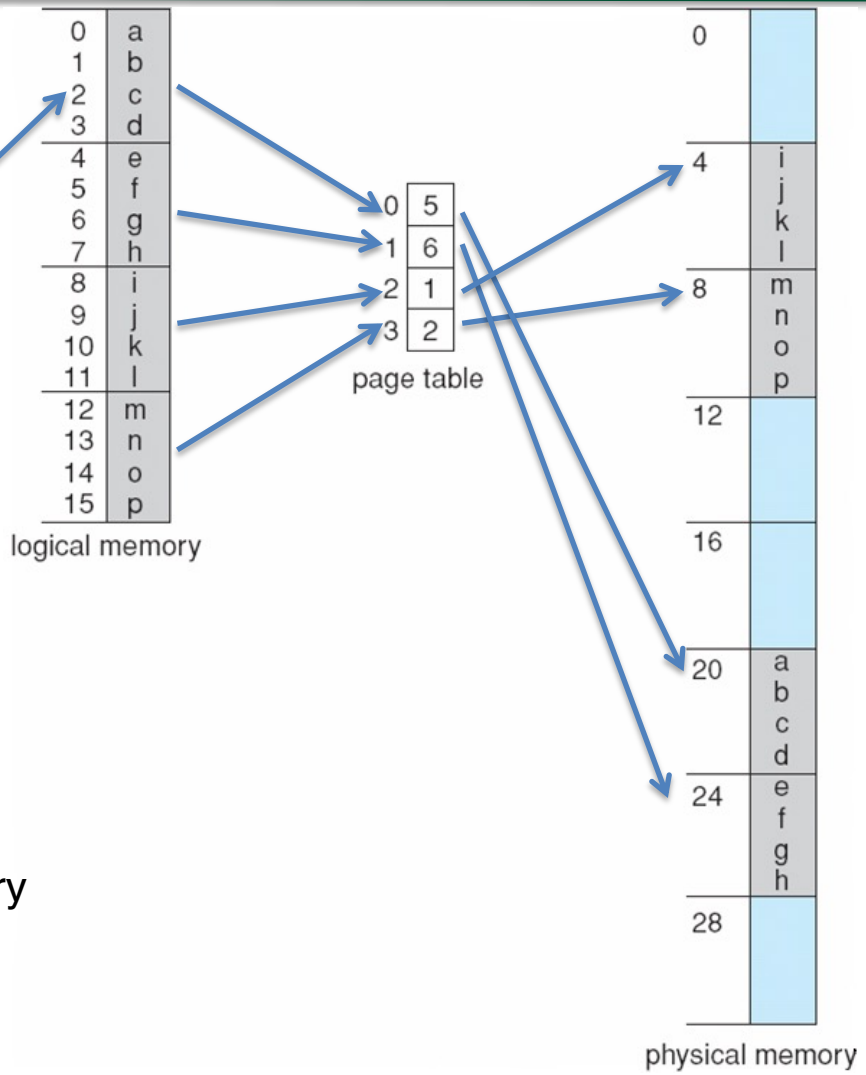
1K byte frame size

Paging Model of Logical / Physical Memory



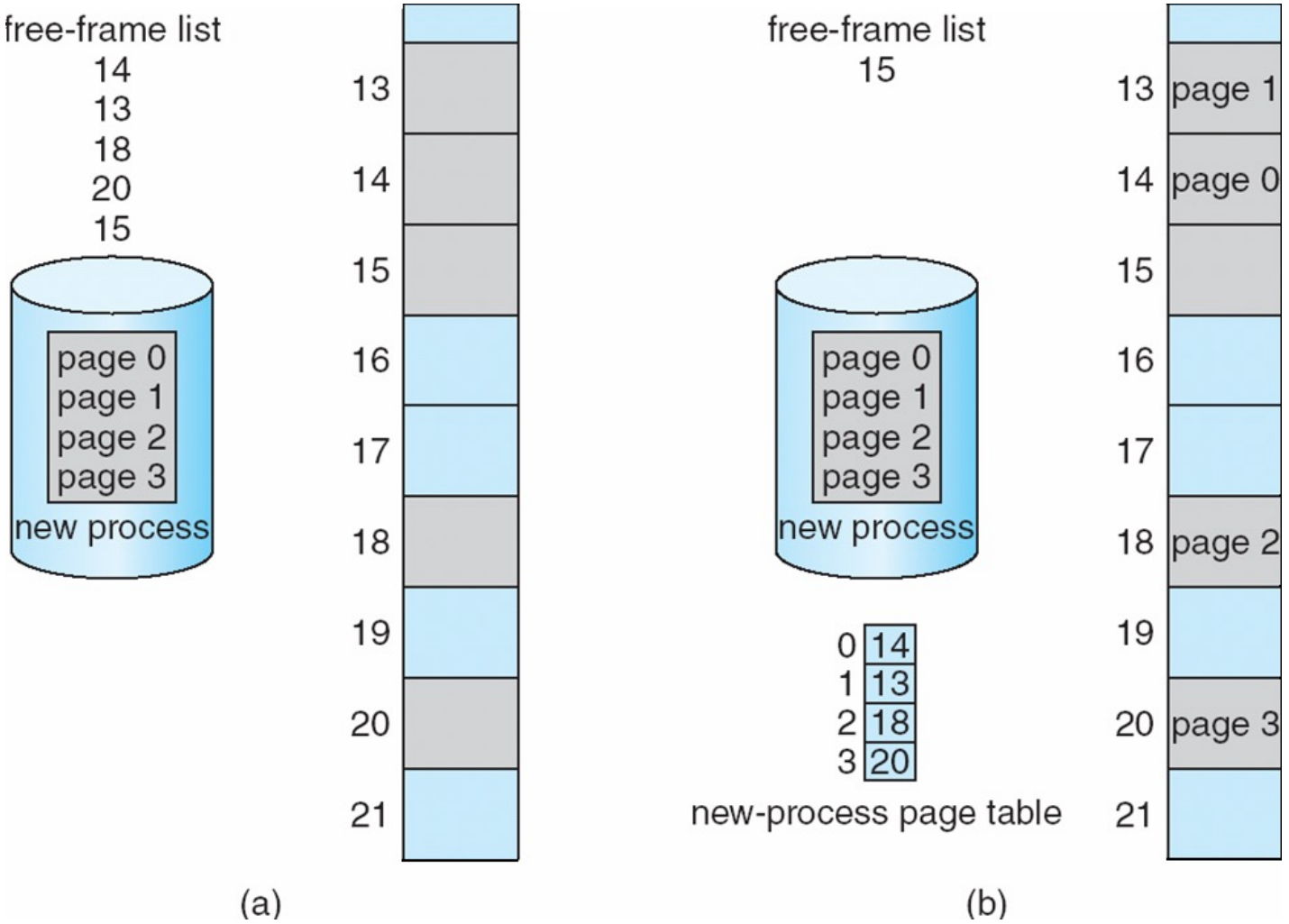
Paging Example

These are addresses not page #



n=2 *m*=4
32-byte memory
4-byte pages

Free Frames



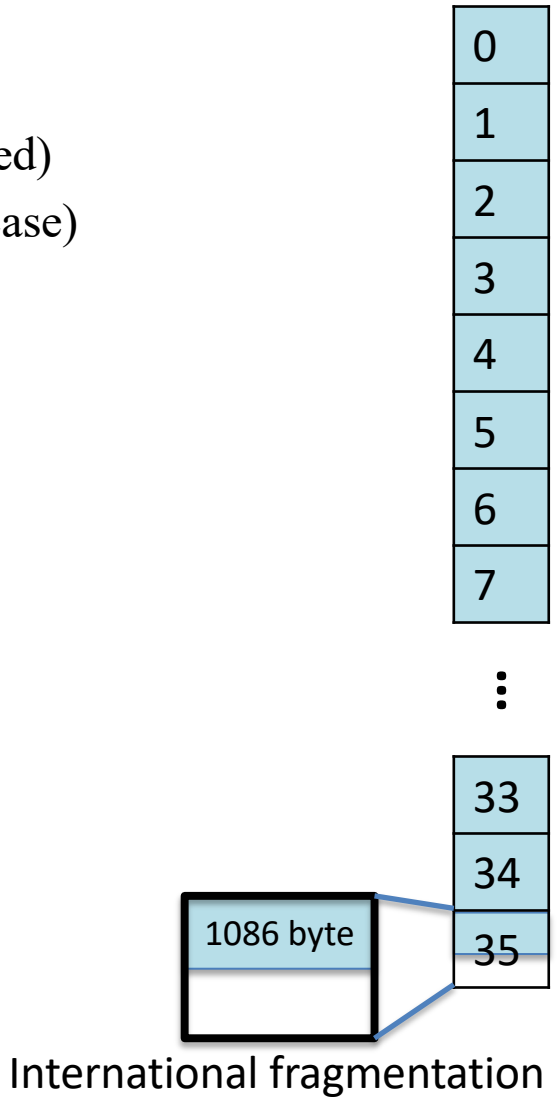
Before allocation

After allocation

Paging and Internal Fragmentation

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes (different bytes addressed)
  - 35 pages (all full) + 1,086 bytes (in last page) (best case)
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
- Worst case fragmentation
  - A page has only 1 addressed byte
  - So small frame sizes are more desirable?
- Page table is stored in memory
  - One entry per page used by a process
  - Smaller frame sizes result in larger page tables

Note: typical page/frame size of Linux systems: 4K or 8K



International fragmentation

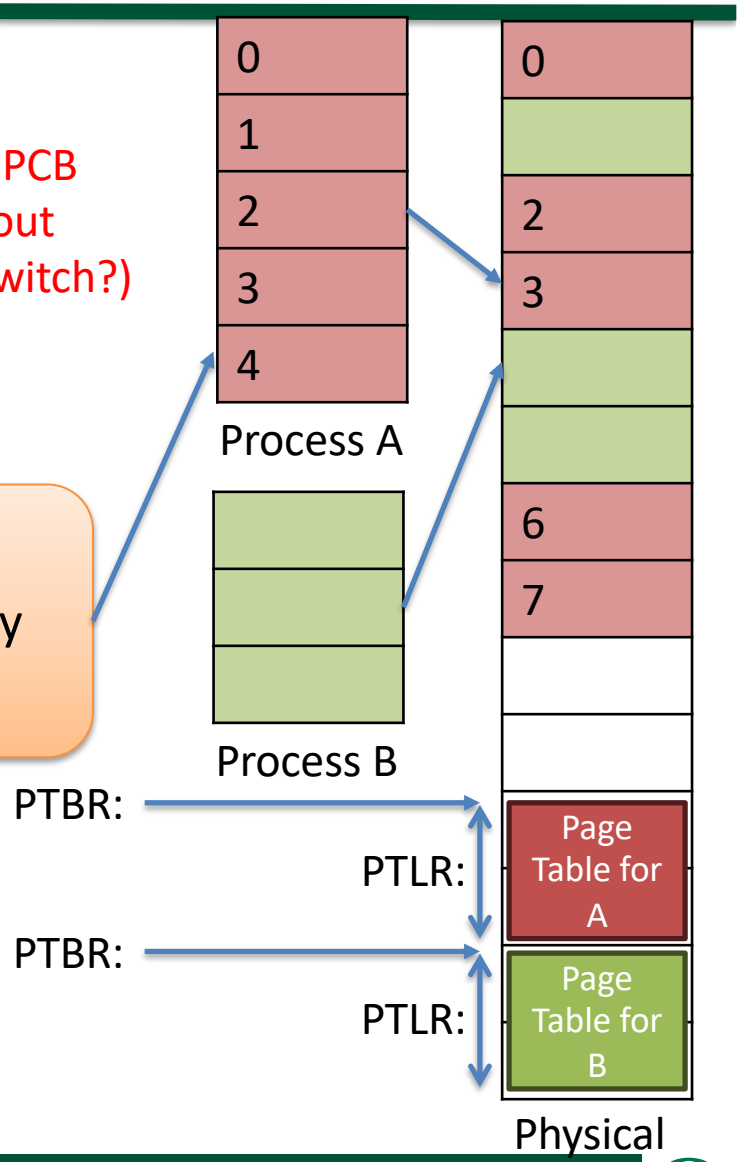
Implementation of Page Table

- Page table is kept in main memory
- *Page-table base register (PTBR)*
  - Points to the page table
- *Page-table length register (PTLR)*
  - Indicates size of the page table
- Each process has its own page table

Stored in PCB (think about context switch?)

Note: Process always sees a single contiguous space regardless of the choice of physical allocation strategy used (contagious or paging)

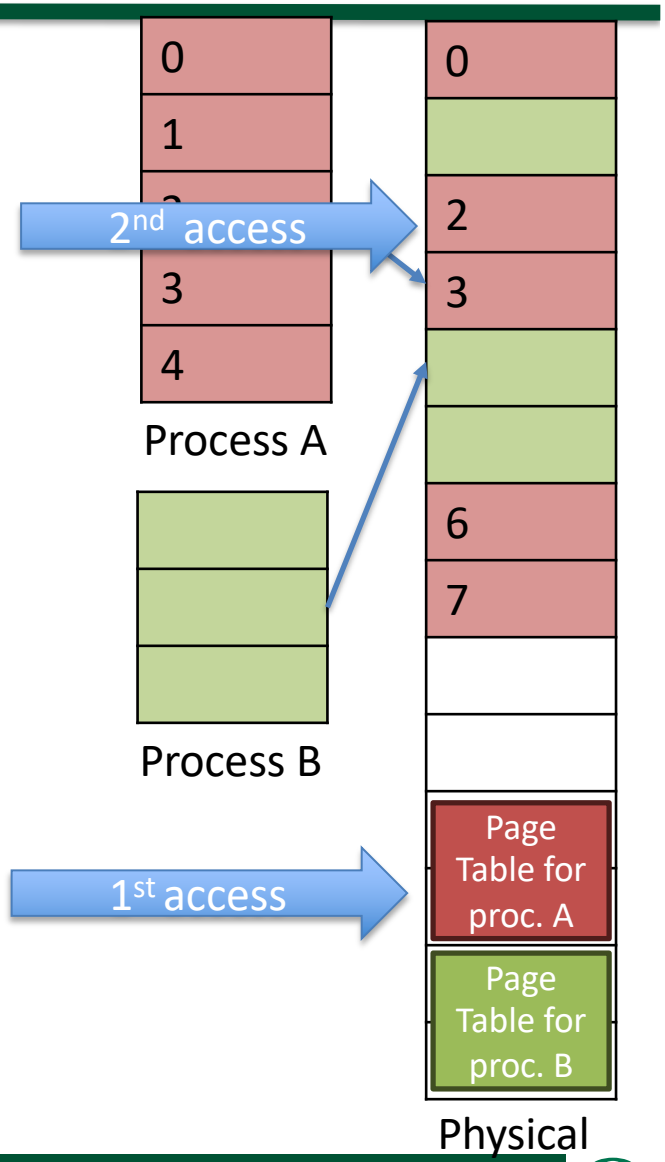
Note: OS kernel also maintains a frame table: tracking which frame (physical memory) is used or free



Implementation of Page Table

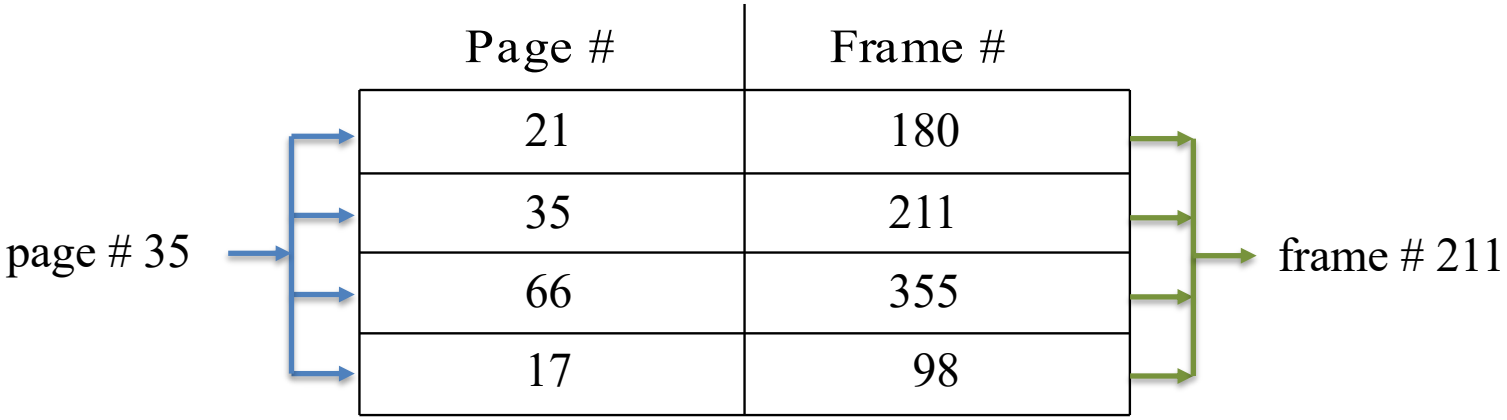
- Every data/instruction reference requires two memory accesses
  - One for the page table and one for the data / instruction
- Address translation must be fast
  - Wasting a memory reference to access the page table
  - 50% overhead

The 50% overhead is too much. How to reduce it? Can hardware help in any way?



Translation Look-Aside Buffer (TLB) (1)

- *Translation look-aside buffer (TLB)*
  - *Hardware cache for page tables*
  - Implemented with high-speed associative memory
  - TLBs are small (64 to 1,024 entries, perhaps larger)
  - *Holds a small amount of page-to-frame mapping info*
- A TLB for a page table holds <*page, frame*> pairs



- Implemented with *associative memory*
  - Enables access by "value" instead of by "address"
  - expensive (monetary) but cheap (time)

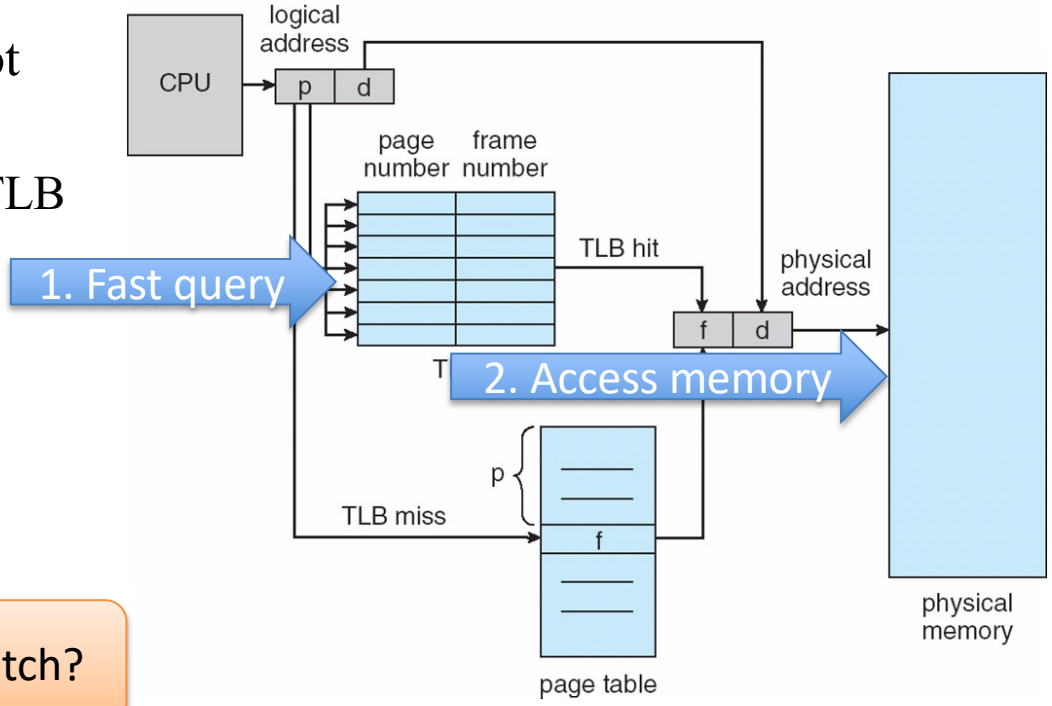
Translation Look-Aside Buffer (TLB) (2)

Without TLB: access page table (memory) → access frame (memory)

With TLB: query TLB (fast memory)
- If hit: access frame (memory)
- If miss: access page table (memory) → access frame(memory)

How to maximize the chances?

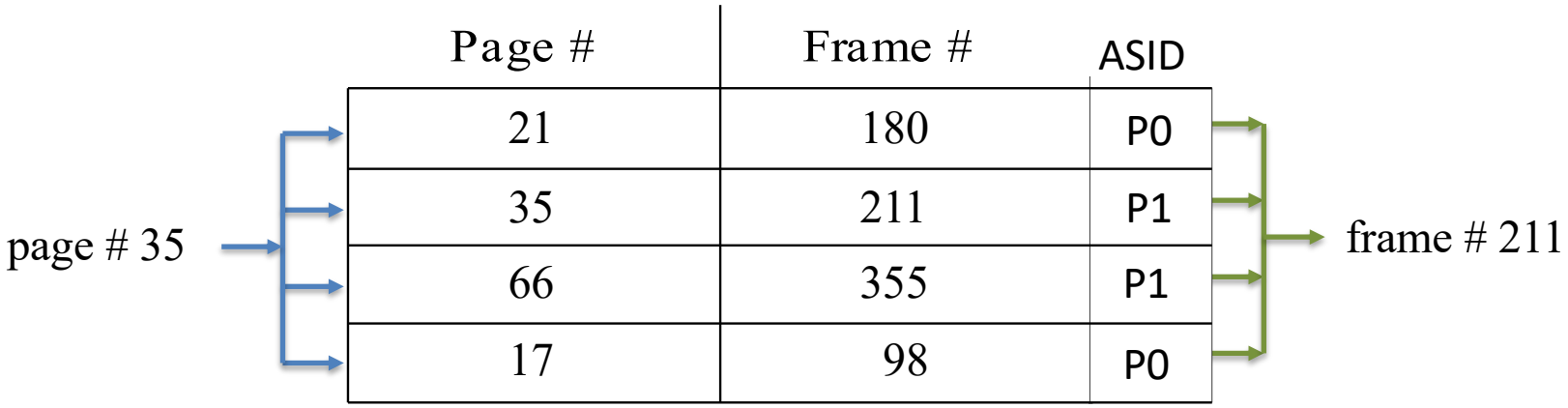
- TLB miss occurs when page # is not present
  - <page #, frame#> loaded into the TLB
  - Faster access next time
  - Replacement policies must be considered
  - Some entries can be permanent for guaranteed access



What happens at a context switch?

Translation Look-Aside Buffer (TLB) (3)

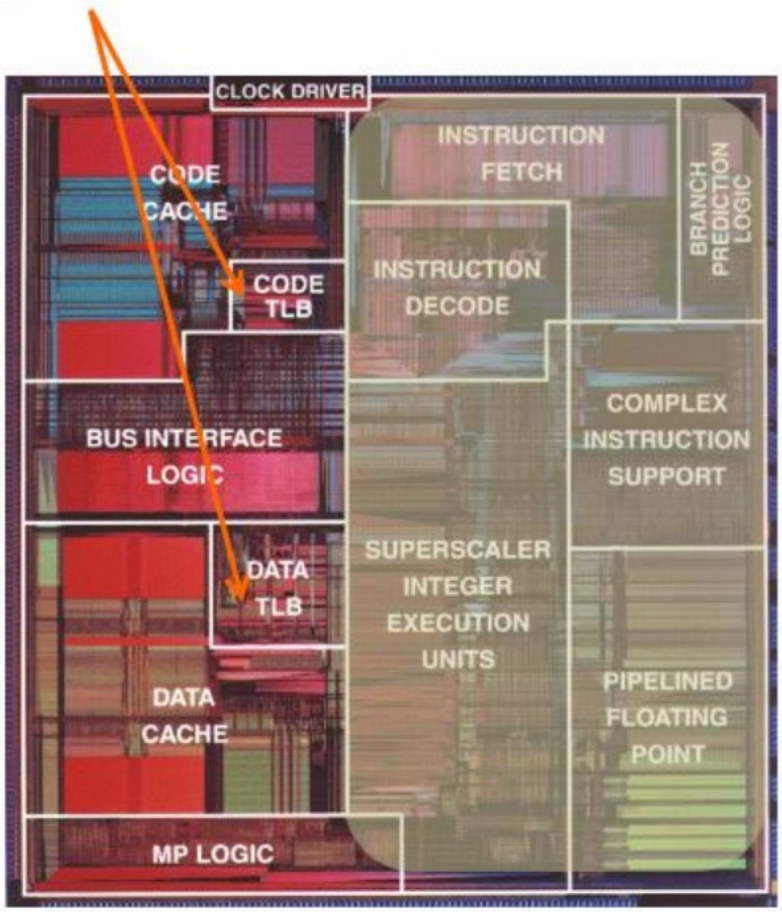
- Some TLBs store *address-space identifiers (ASIDs)*
  - Uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch



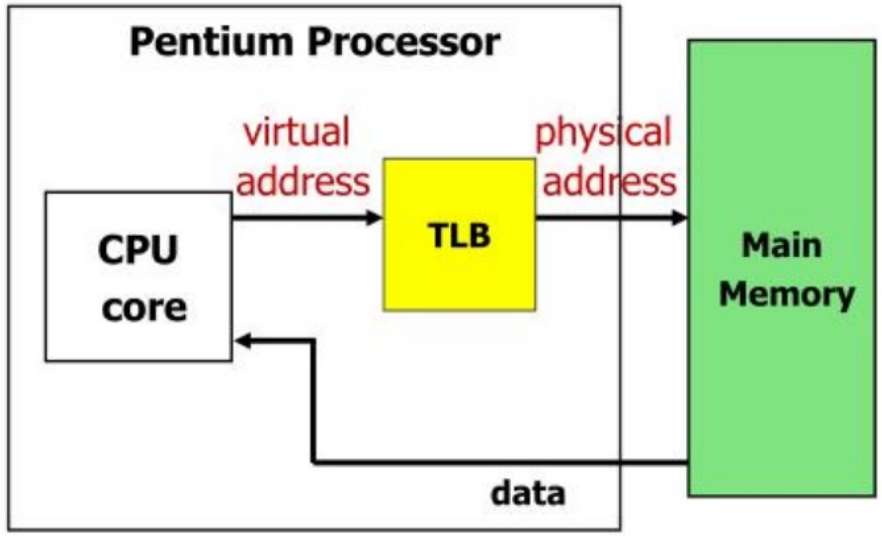
What is the benefit of having ASID?

TLB on Intel Pentium Processor

TLB for instructions and data



Intel Pentium Processor (1993)



Pentium (1993)
60MHz
3.1M Transistors
(800nm process)

Effective Access Time

- TLB lookup time (fraction of memory time) = ε
  - If M is the time to access memory, ε * M is the TLB lookup time
- Hit ratio = α
  - How often a page number is found in the TLB

Effective Access Time (EAT) (to access memory)

EAT= (time if hit in TLB) + (time is miss in TLB)

$= \alpha(\varepsilon M + M) + (1 - \alpha)(\varepsilon M + M + M)$

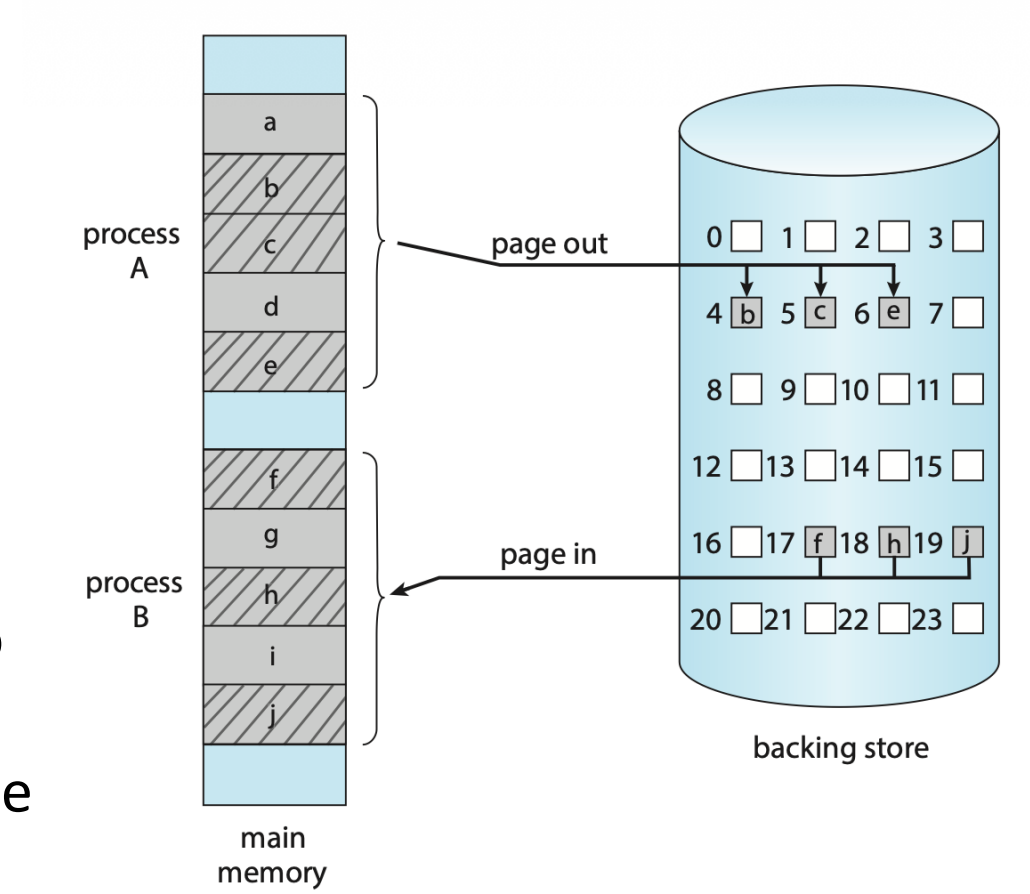
$= M * (\alpha(\varepsilon+1)+ (1 - \alpha)(\varepsilon + 2))$

$= M * (\varepsilon - \alpha+2)$

- Examples:
  - Consider α = 80%, ε = 20% , 100ns for memory access
  - EAT = 100 * (0.2 - 0.8 + 2) = 100 * 1.4 = 140ns
  - Consider α = 99%, ε = 20%, 100ns for memory access
  - EAT = 100 * (0.2 – 0.99 + 2) = 100 * 1.21 = 121ns

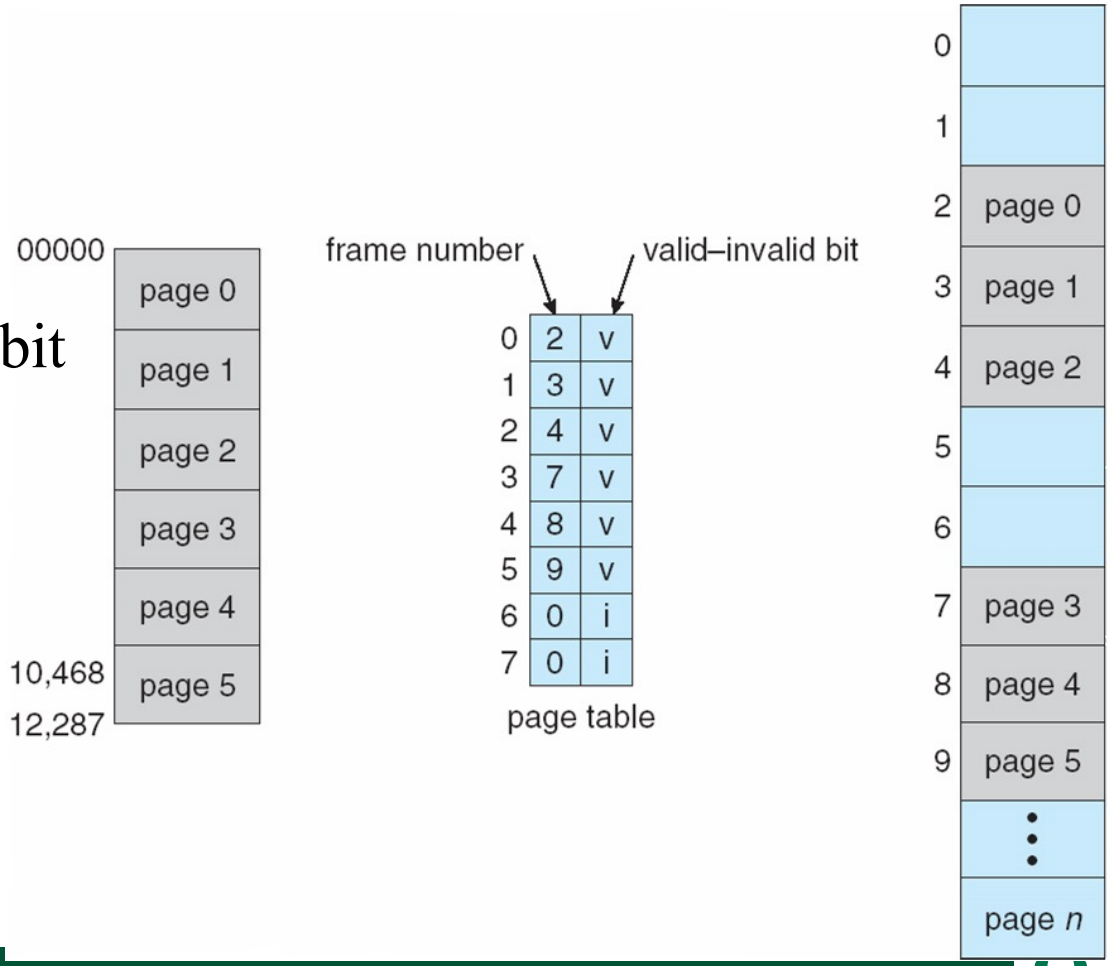
Swapping Pages

- Every process thinks it has a big, virtual address space.
  - $2^{32} – 2^{64}$ bytes
- Physical memory is tiny compared to how much memory programs pretend they have
- Swapping
  - Allow less-used frame to be moved to storage to temporarily free up space



Valid (v) or Invalid (i) Bit In A Page Table

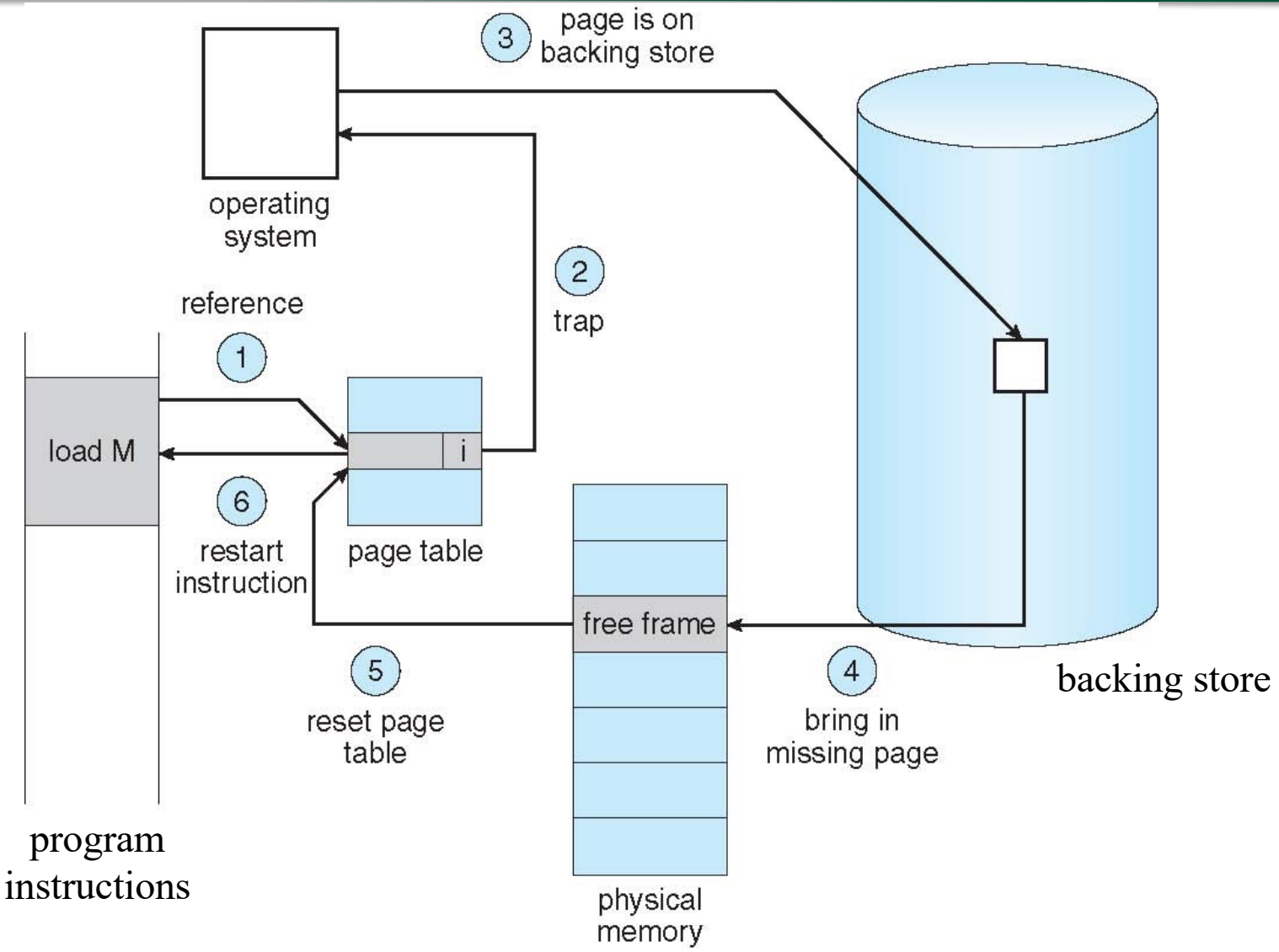
- Each page table entry has a *valid–inv*alid bit
  - *v* ⇒ in-memory (memory resident)
  - *i* ⇒ not-in-memory
- Initially valid–invalid bit is set to *i* on all entries
- During MMU address translation, if valid–invalid bit in page table entry is *i*, a page fault occurs



Page Fault Processing

- If there is a reference to a page, first reference to that page will trap to operating system … why?
- Operating system looks at the page table to decide:
  - Invalid reference ⇒ abort
  - Just not in memory (*page fault*) (first reference is)
- Find free frame
- Swap page into frame via scheduled disk operation
- Reset tables to indicate page now in memory
  - Set validation bit = *v*
- Restart the instruction that caused the page fault

Steps in Handling a Page Fault

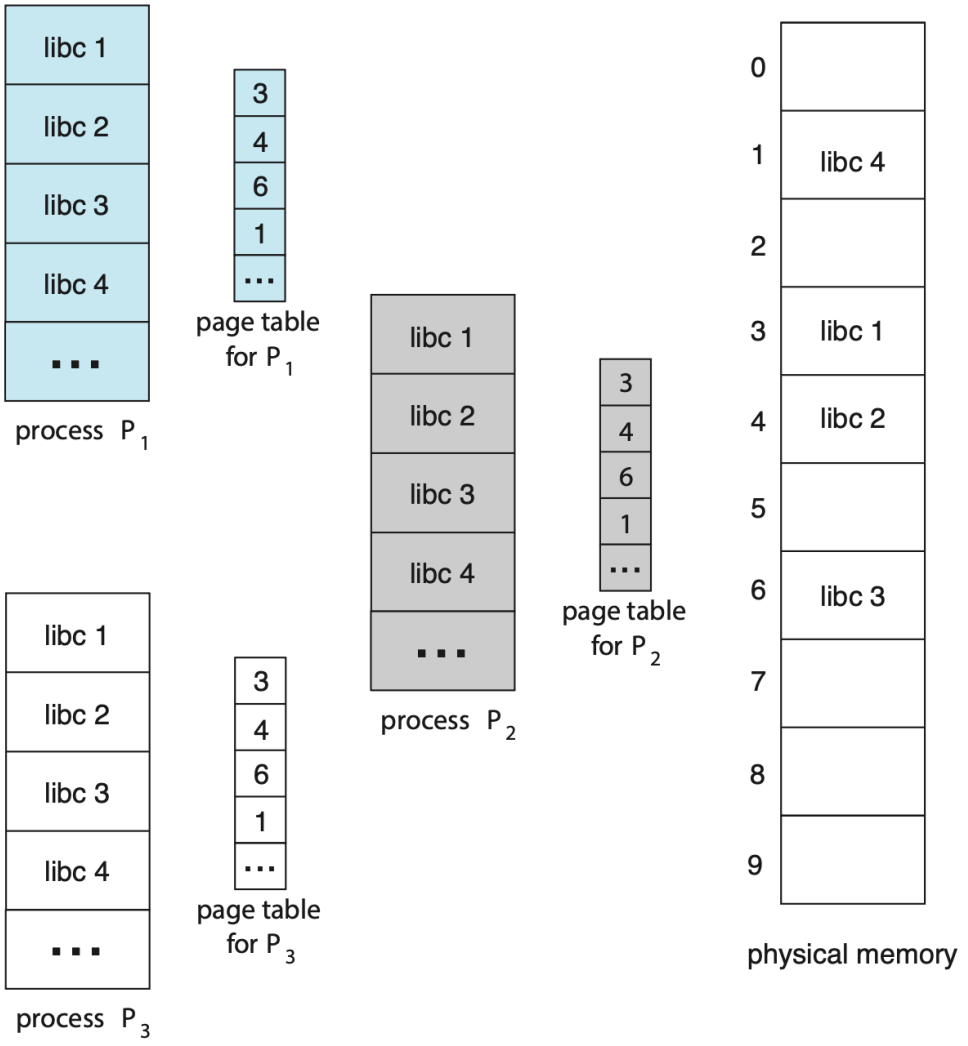


Shared Pages

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- Private code and data
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example

Note: All processes have a reference to libc, not a copy



Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes -> 4 MB for page table alone!!!
- Approaches to address large page table problem
  - Hierarchical paging
  - Hashed page tables
  - Inverted page tables

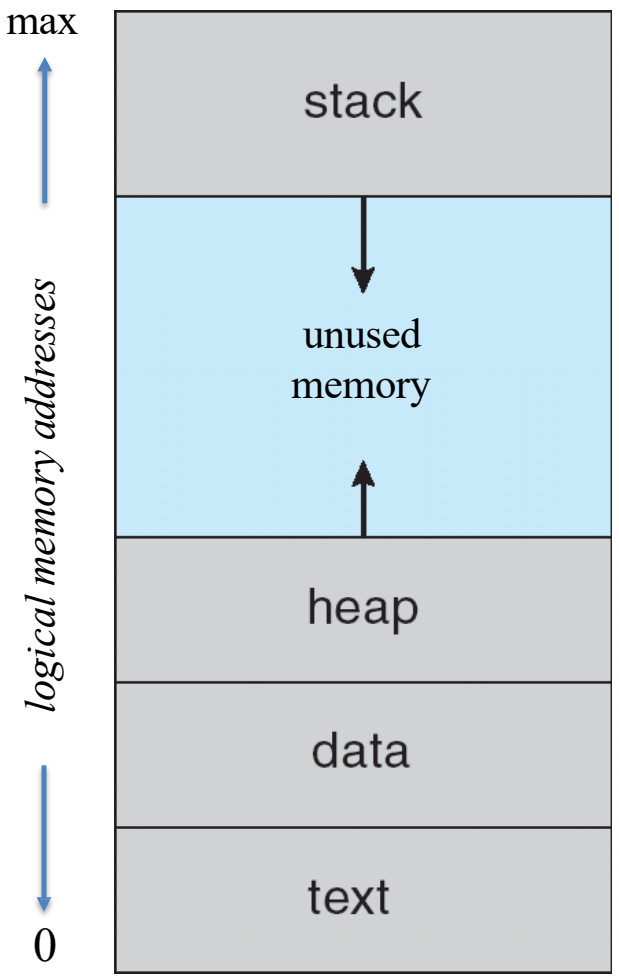
Question: can you estimate the page table length for 64-bit logical address space with page size 16 KB?

Why does the page table have to track all addresses?

If a process only uses 1000 addresses, can we shorten the page table?

Issues

- Page table is accessed with index (like array[idx])
- Stack and heap always grow from both ends
- Often, there is a big hole in the middle
- A lot of the page tables are unused, but we have to have entries for those spaces.



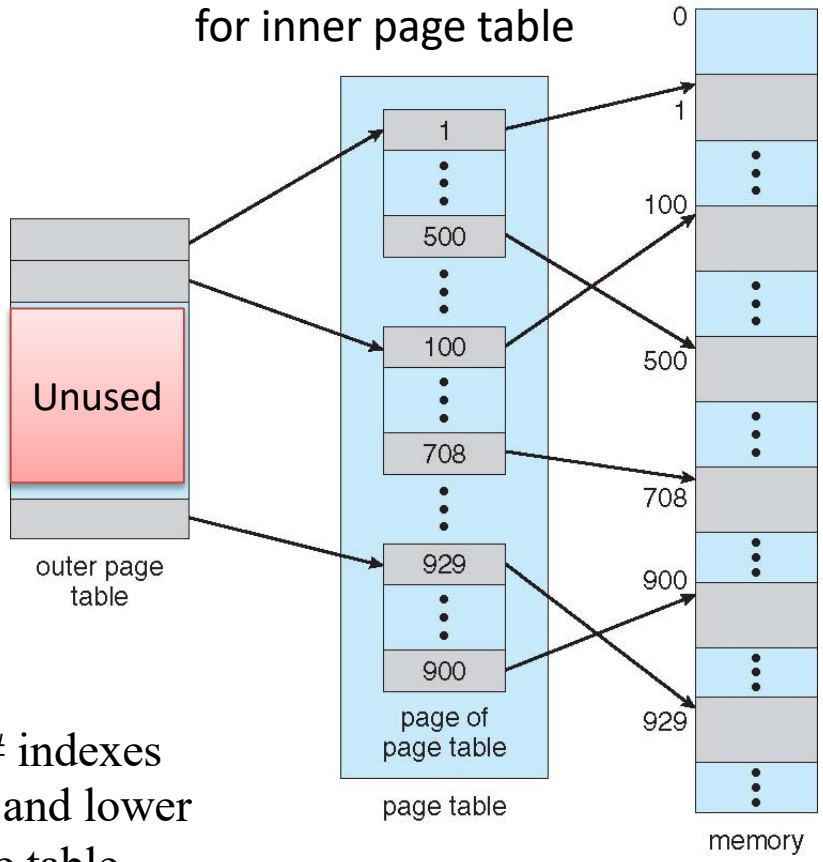
Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- Use multiple levels to keep track of them
  - Hierarchy of page tables
  - Page # is used index across the hierarchy
- A simple technique is a two-level page table
- Page table itself is allocated in pages

Hierarchical page tables = Page the page table

Two-Level Page-Table Scheme

Outer page table contain all the possible entries for inner page table

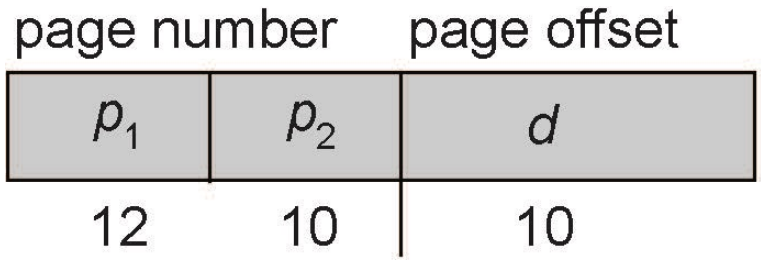


Only need to save the inner page table for the used part of memory space

Upper bits of page # indexes the outer page table and lower bits index inner page table

Two-Level Paging Example

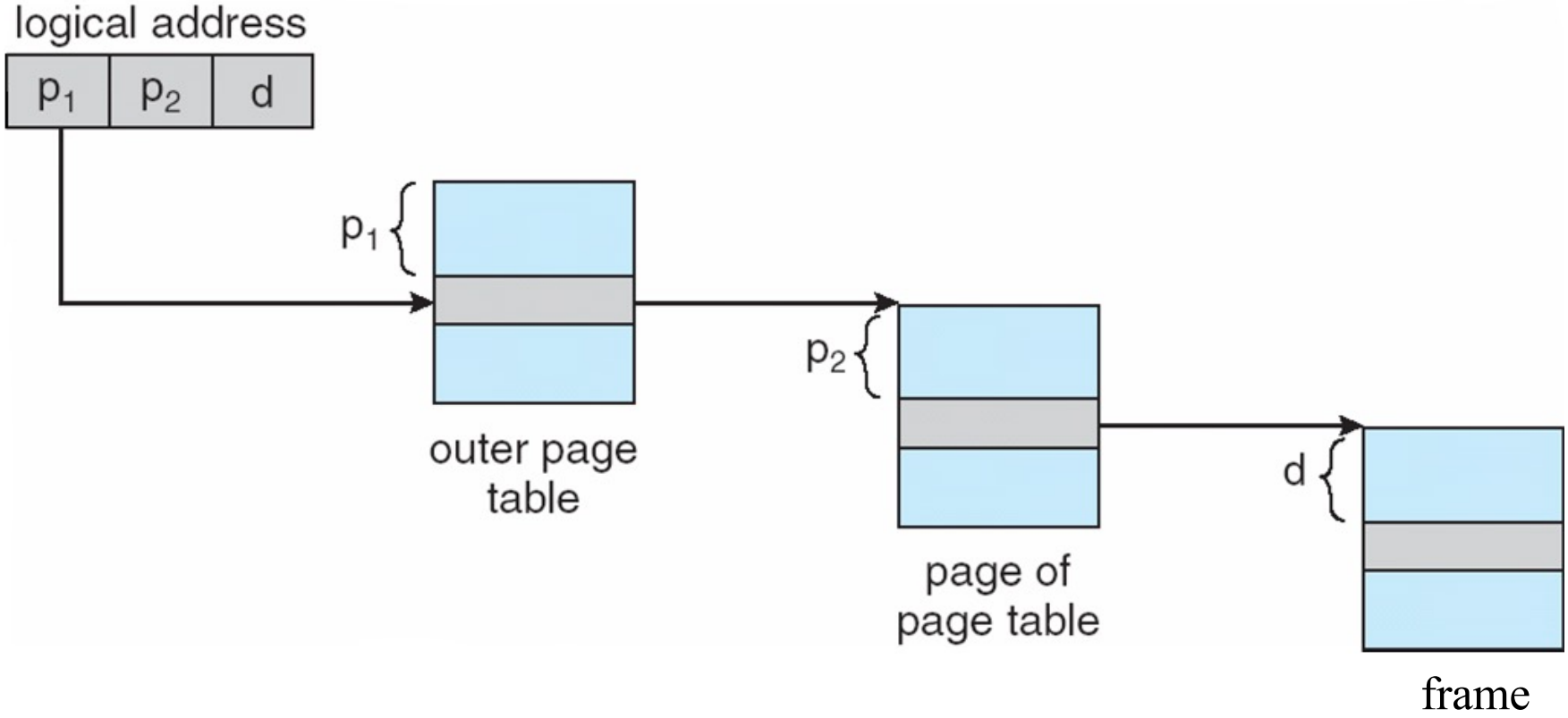
- A logical address (on 32-bit machine with 1K page size) is divided into:
  - A page number consisting of 22 bits
  - A page offset consisting of 10 bits
- Since the page table is paged, the page number is divided into:
  - 12-bit page number
  - 10-bit page offset
- Thus, a logical address is as follows:



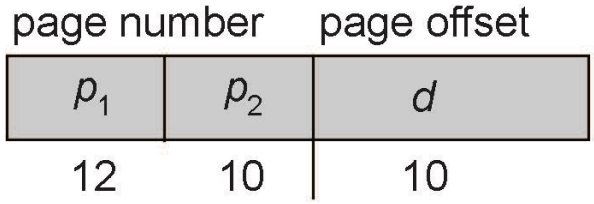
where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table

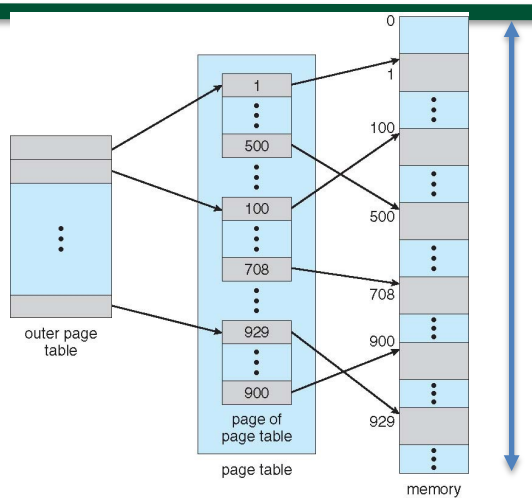
- Known as *forward-mapped page table*

Address-Translation Scheme (2-level)



Two-Level Paging Example



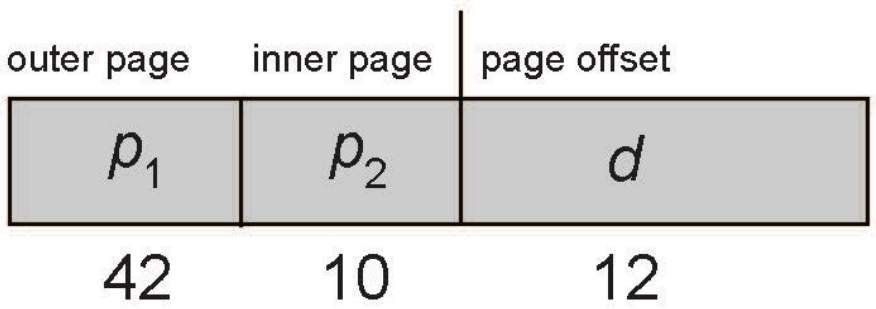


How much space are we saving if we only need 10 inner page tables? (Assuming 4 bytes per entry)

- One level page table
  - $2^{22} \times 4$ bytes $= 16$ MB
- Two level page table
  - $2^{12} \times 4$ bytes $+ 10 \times 2^{10} \times 4$ bytes $= 57$KB

64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like

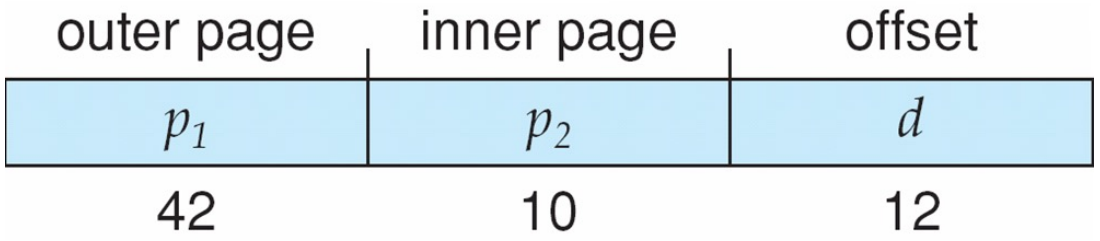


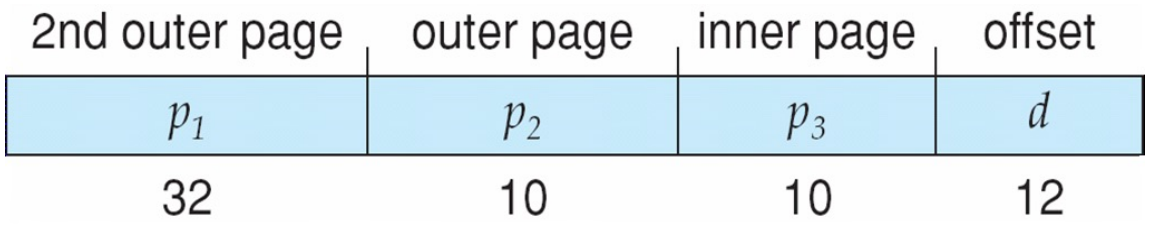
  - Outer page table has 2^{42} entries or 2^{44} bytes

2nd Outer Page Table

- One solution is to add a 2nd outer page table
- In this example, the 2nd outer page table contains 2^{32} entries (2^{34} bytes in size)
  - Possibly 4 memory access to get to one physical memory location

What is the problem with the growing number of levels?

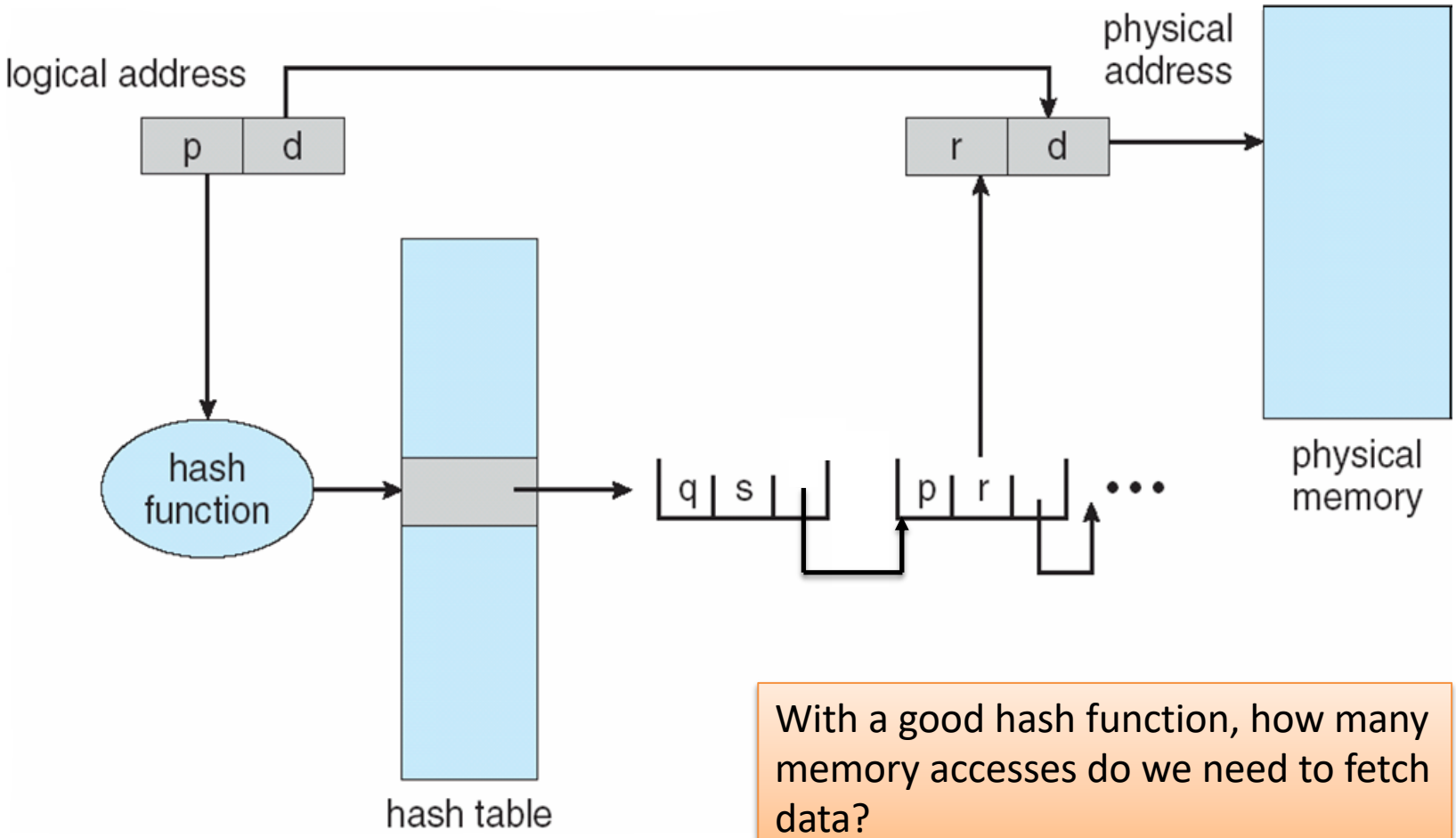




Hashed Page Tables

- Common in address spaces > 32 bits
- The page number is hashed into a page table
  - Multiple page numbers could map to the same page table entry
  - Store list of page numbers in this case
- Each entry contains
  - (1) the page number
  - (2) the value of the mapped page frame
  - (3) a pointer to the next element
- Page numbers are compared in this chain for a match
  - If a match is found, corresponding frame is extracted
- Hashing has to be done by the MMU … Why?

Hashed Page Table

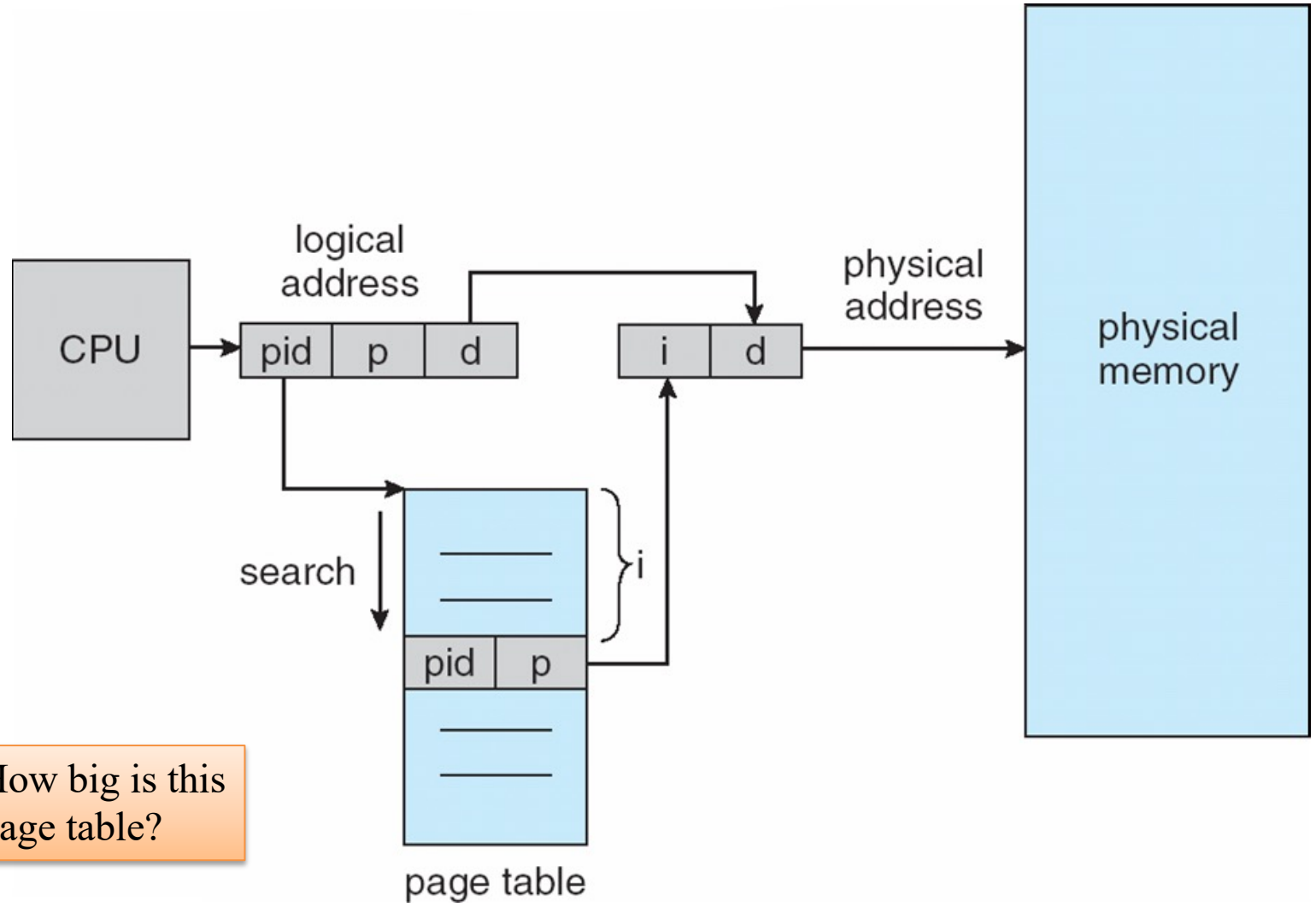


With a good hash function, how many memory accesses do we need to fetch data?

Inverted Page Table

- Converting: page-to-frame mapping to frame-to-page mapping
- One entry for each real frame of memory
- Entry = virtual address + owner proc. Id.
  - Decreases memory needed to store each page table
  - Increases time needed to search the table when a page reference occurs
- How to accelerate search?
  - Hash table (how is this different from the previous hash table?)
  - TLB
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Inverted Page Table Architecture



How big is this page table?

Next Class

- Virtual memory
  - No lecture on this due to Thanksgiving holiday
  - Slides will be provided
  - Read chapter 10