Operating Systems
CS 415

Lecture 11:

Main Memory

Suyash Gupta

Assistant Professor

Distopia Labs and ONRG

Dept. of Computer Science

(E) suyash@uoregon.edu

(W) gupta-suyash.github.io

UNIVERSITY OF OREGON

Announcements

- **Suyash Gupta**
  - Office Hours: Thursday, 10-11am, Deschutes 334

- **Nihal Balivada (TA)**
  - Office Hours: Wednesday/ Friday, 11-12pm, Deschutes 335

- **Ranjitha Rani (TA)**
  - Office Hours: Monday /Tuesday, 1-2pm, Deschutes 229

Assignment 3 is out!

- **Deadline → June 3, 2026 at 11:59pm PST**
- Please start working and talk to us if you are stuck.

- **Final → June 10, 2026 at 12:30pm PST, STB 145**
  - Closed book, no cheat sheets, no discussions.

Last Class

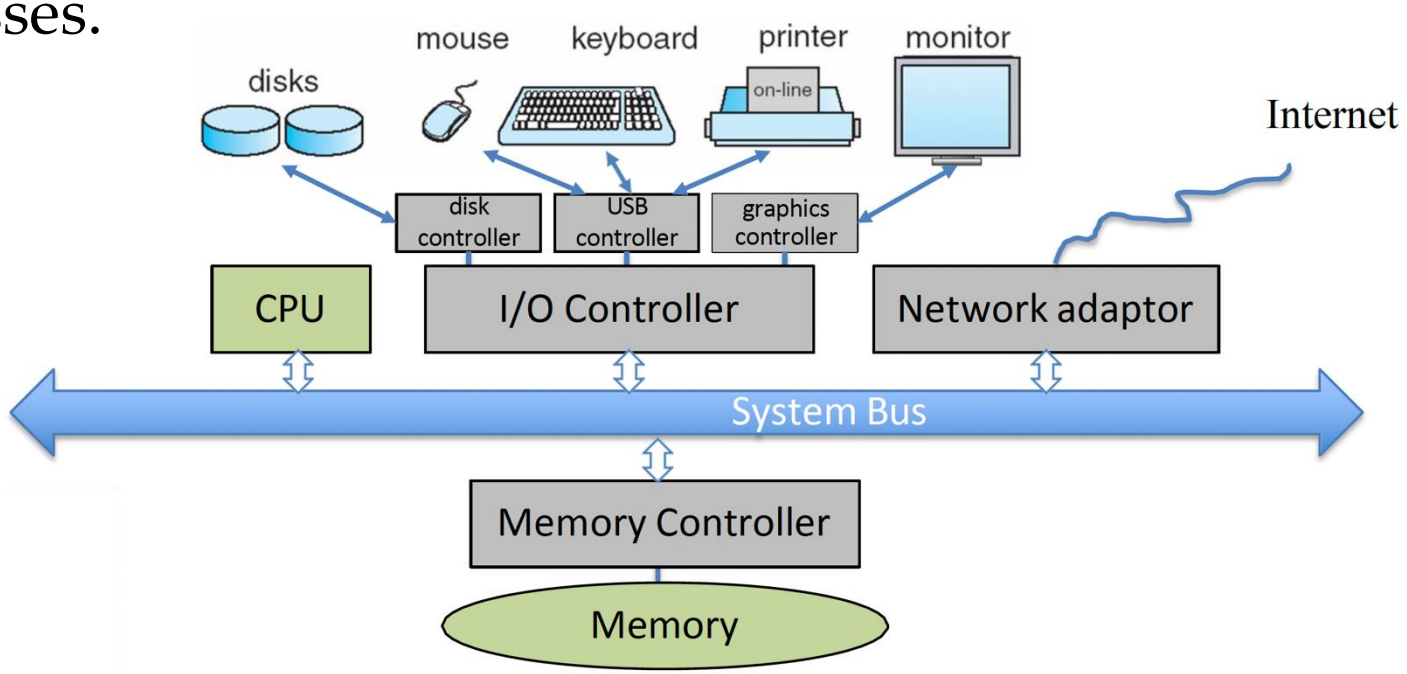
- Deadlocks (Chapter 8)

- Next, we move to Chapter 9!

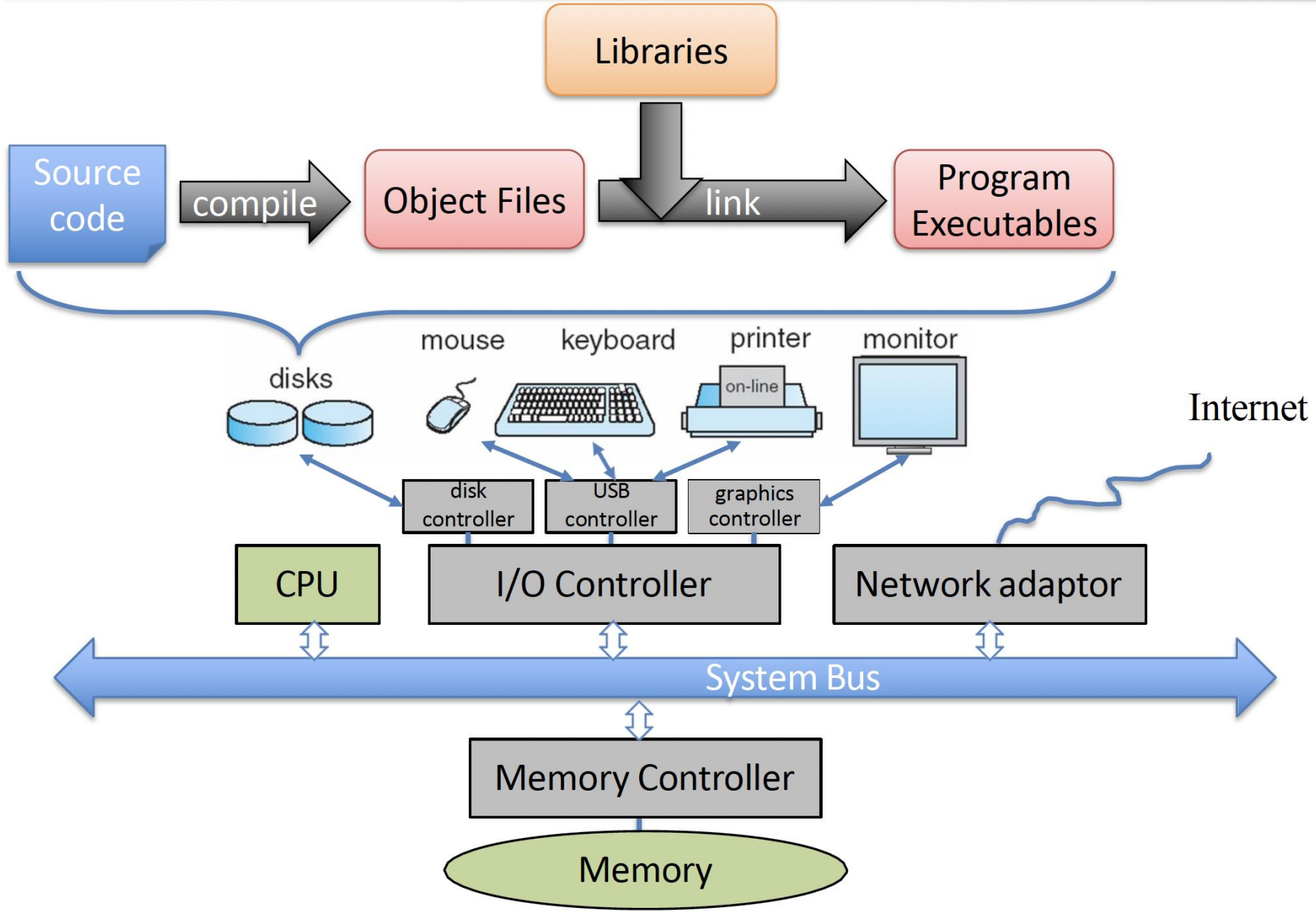
Memory

- Memory is a large array of bytes, each with its own address.

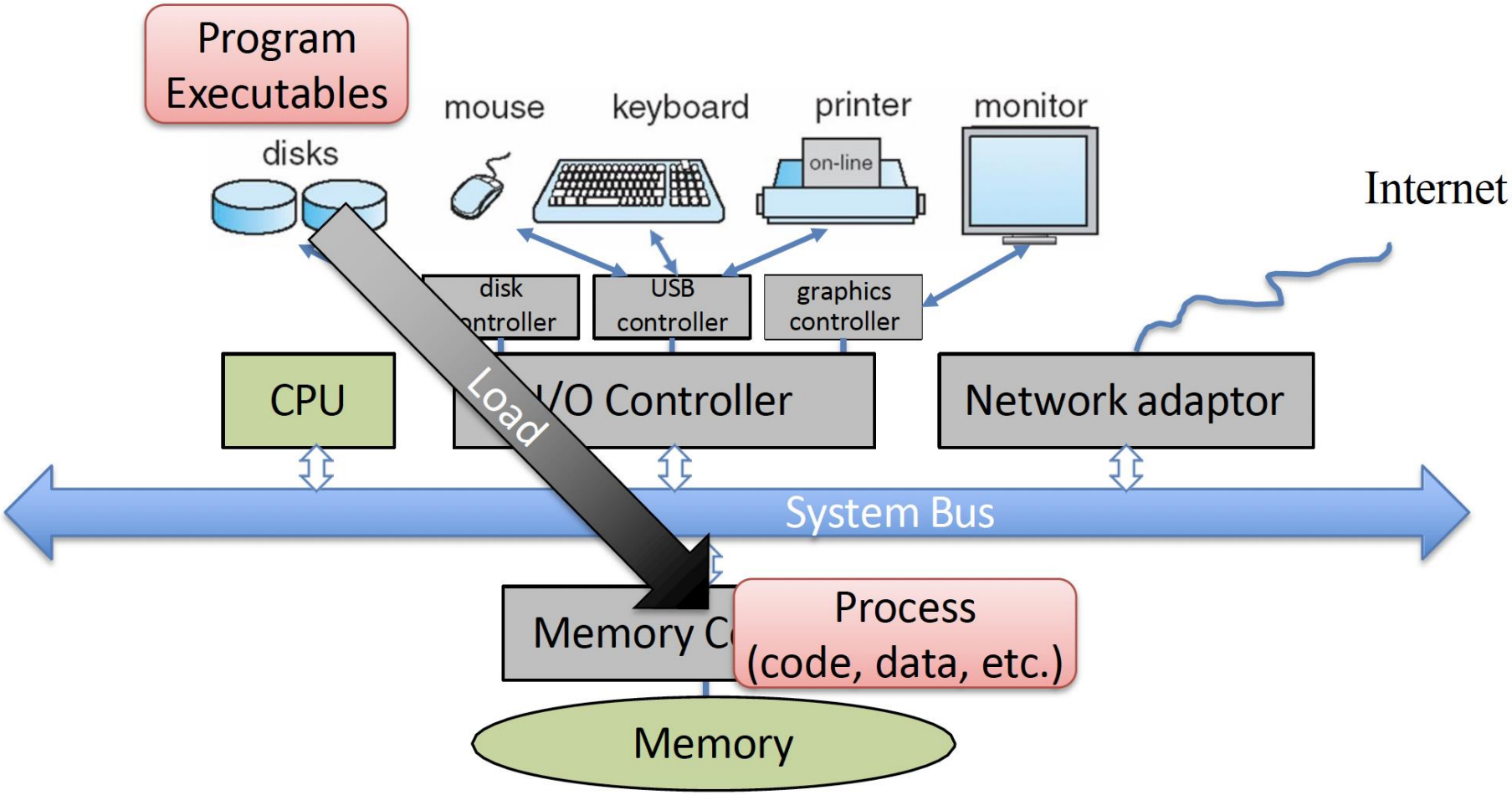
- Recall that the CPU fetches instructions from memory according to the value of the program counter.
  - These instructions may cause additional loading from and storing to specific memory addresses.



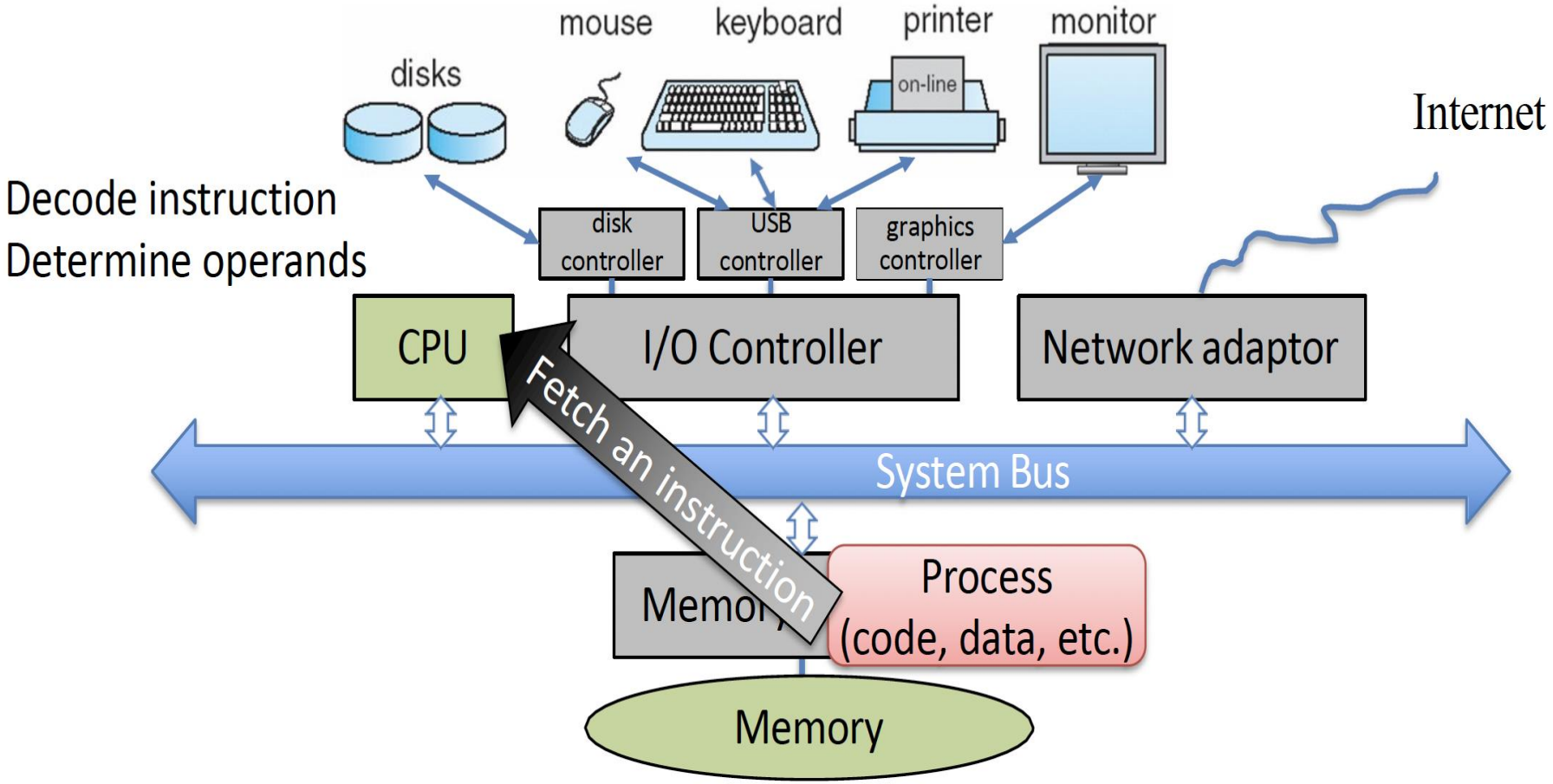
Code to Process Execution



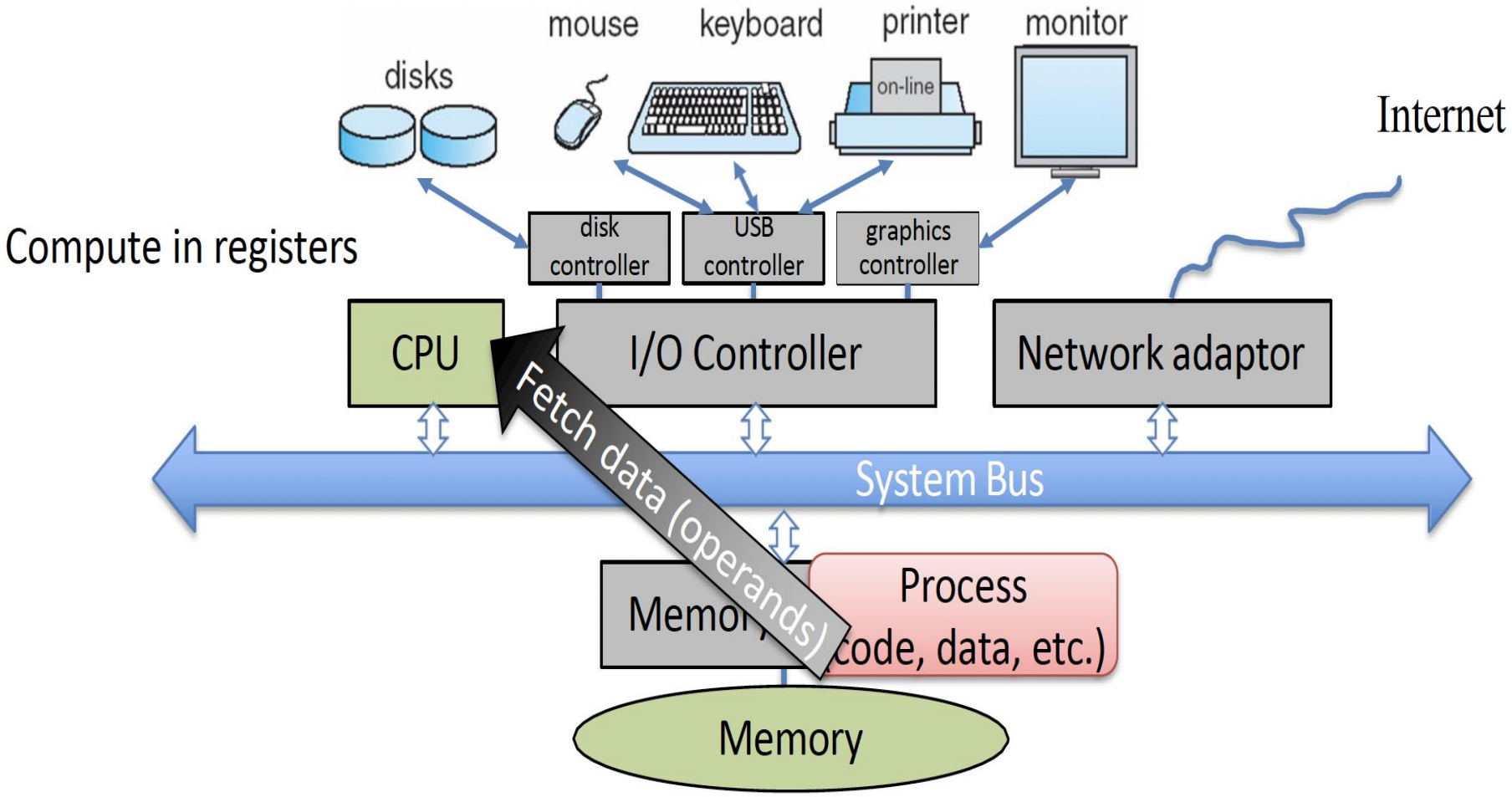
Code to Process Execution



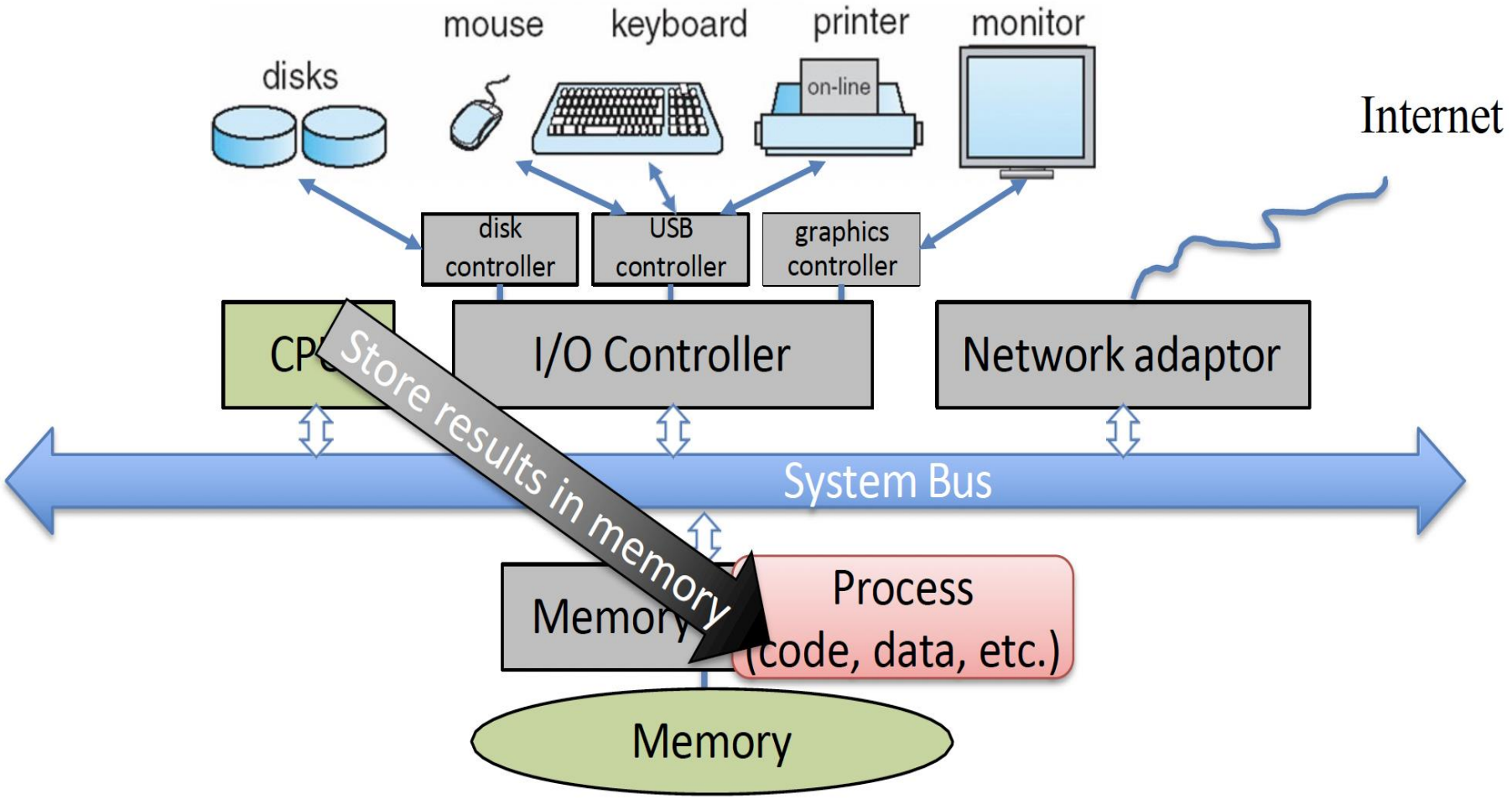
Code to Process Execution



Code to Process Execution



Code to Process Execution



Processes and Memory

- How do multiple processors share the same physical memory?
  - Who has access where?
  - How are these processes protected from each other?

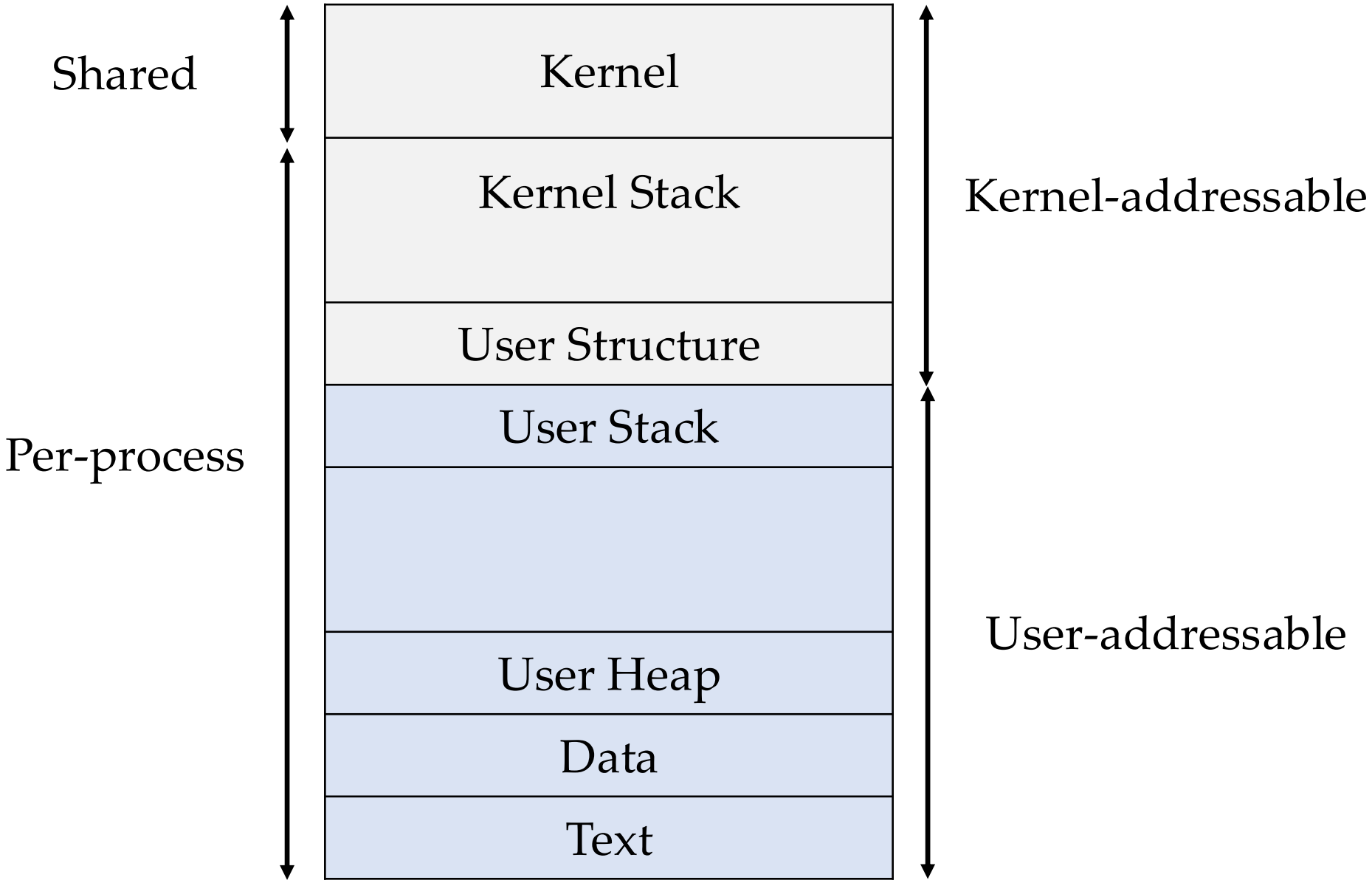
- Why don’t programmers worry about where to put data in memory?
  - We are never worried about where does malloc gets us memory?

- How does each instruction reference memory?
  - Are the addresses baked in each instruction?

Process Address Space

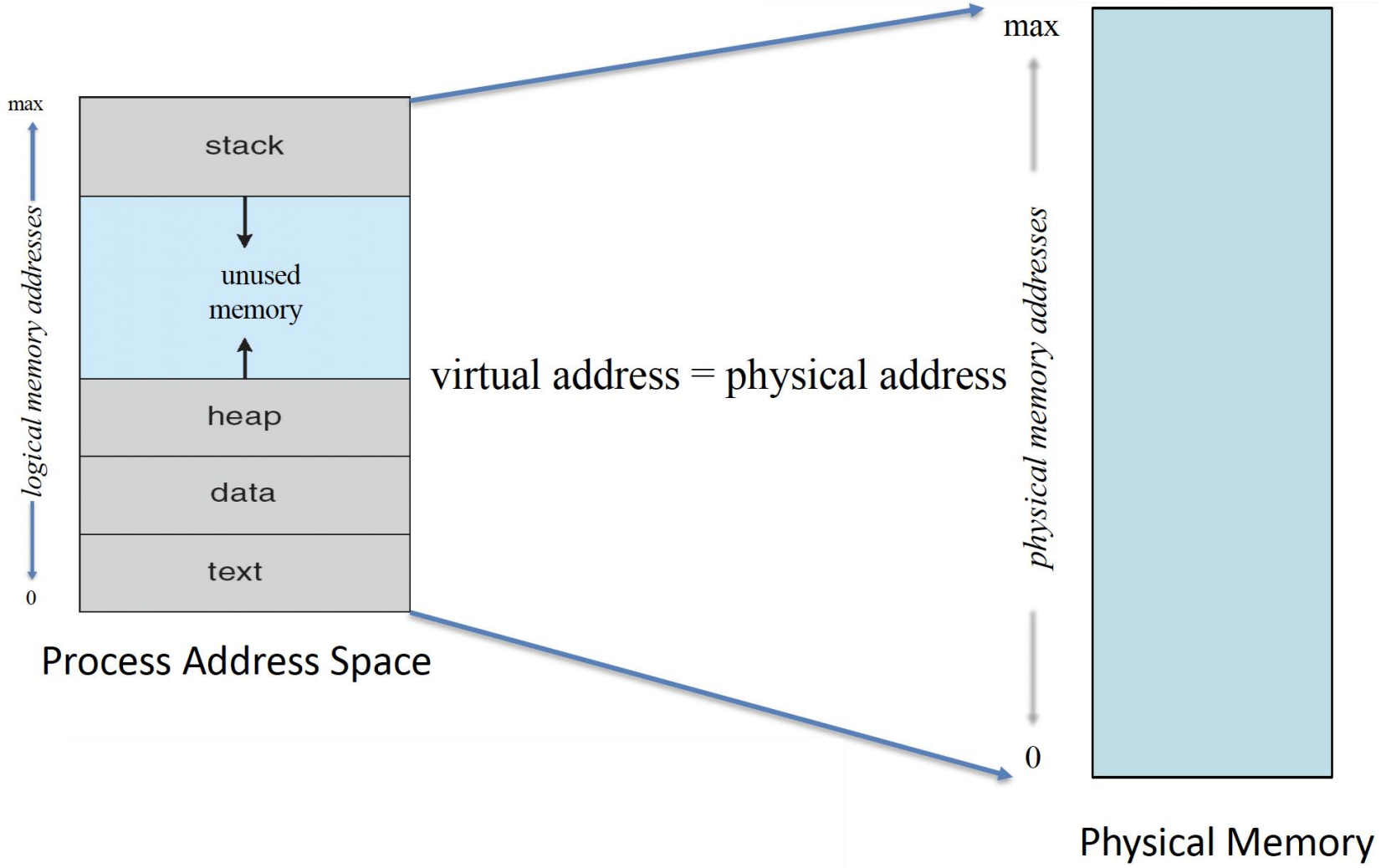
- Process address space is all locations addressable by the process.
- Each process has its own address space.
- Not every location can be accessed in user mode!
- Always logically contiguous (0 to max).
- Each byte is identified by a logical memory address (a.k.a virtual address).

Process Address Space



If only one process?

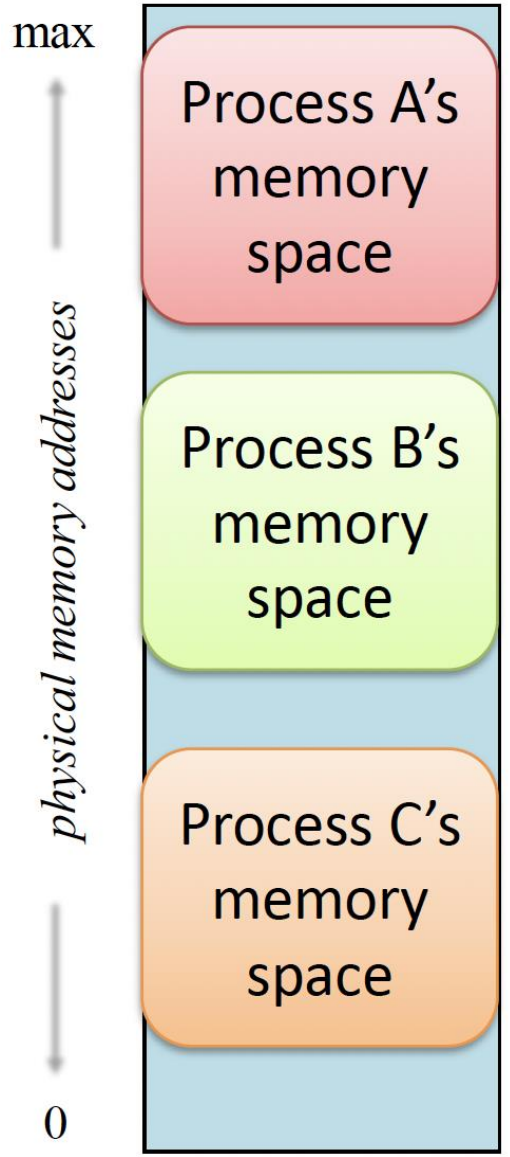
- If only one process is allowed at a time, memory management becomes trivial.
- It can occupy the entire physical memory.
- A virtual address can be 1:1 mapped to a physical memory.



Supporting Multiprogramming

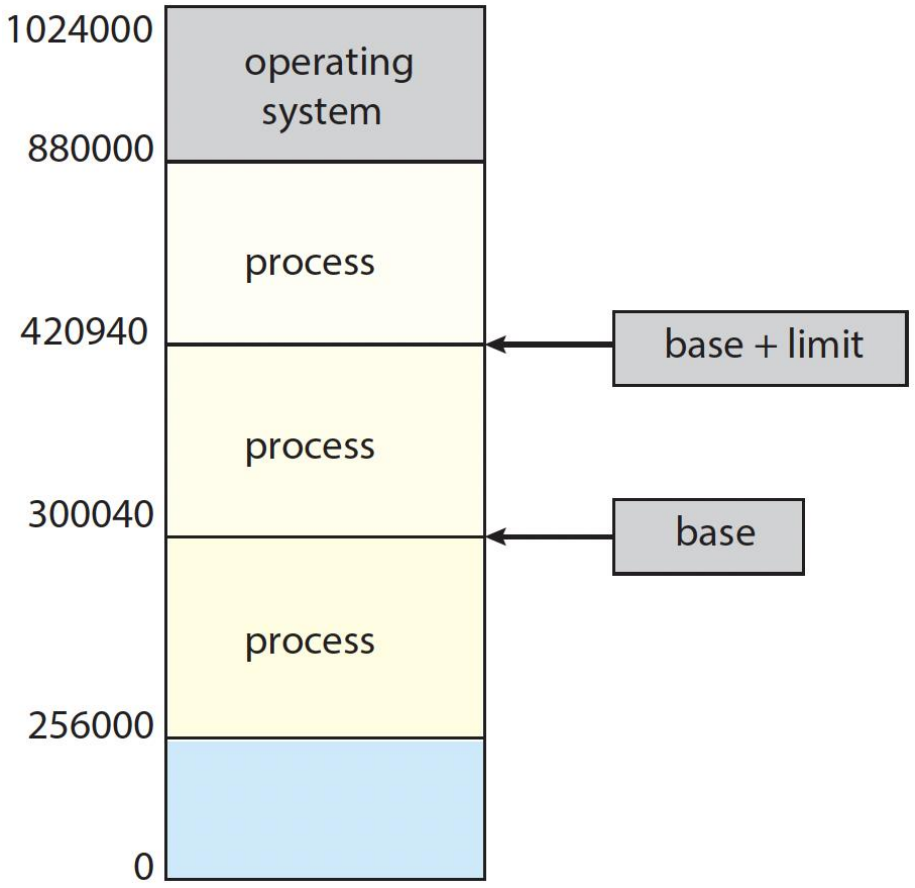
- A user process can reside in any part of the physical memory.
  - The address space of the computer starts at 0, but the first address of a user process need not be 0.

- Mapping user program to physical addresses.
  - A user program goes through several steps before being executed.
  - Addresses may be represented in different ways during these steps.

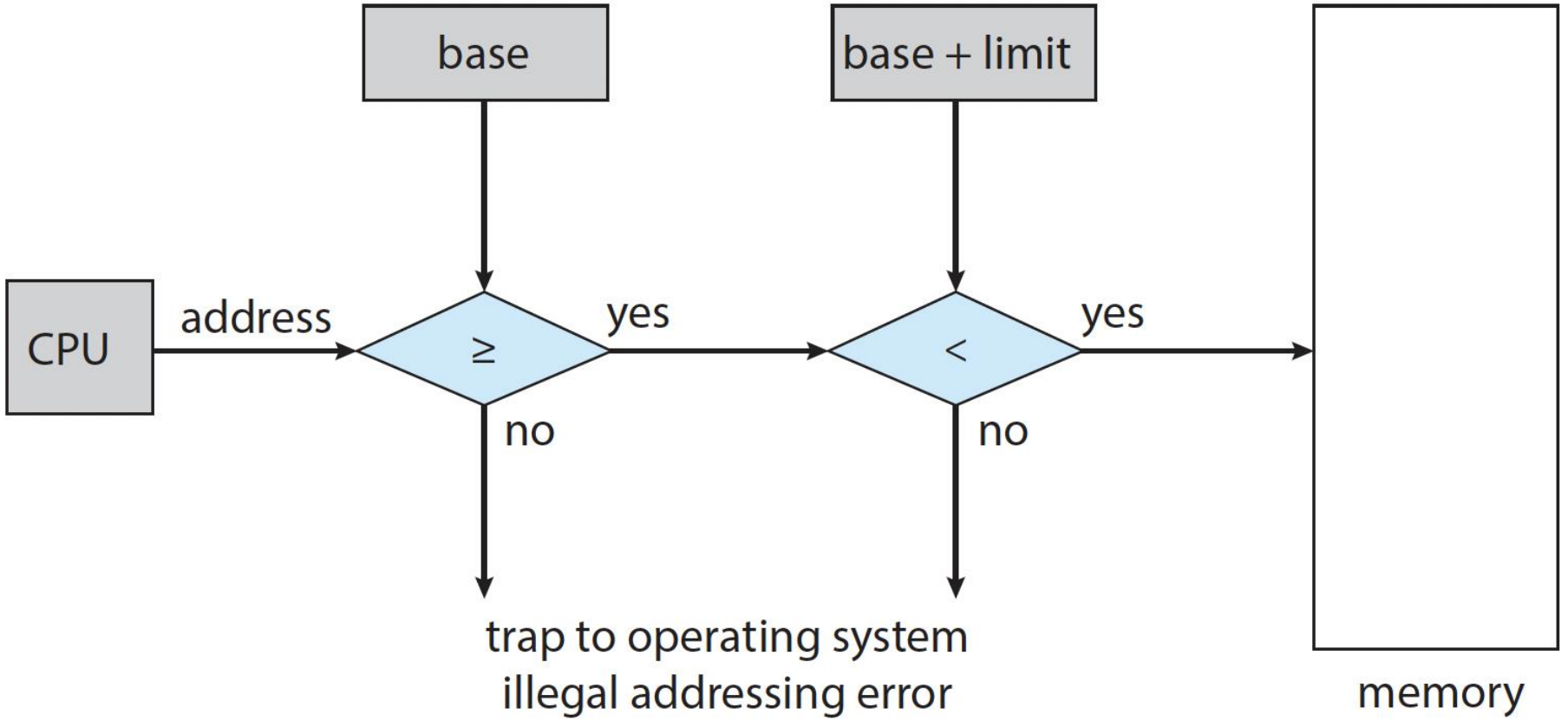


Supporting Multiprogramming

- Ensure that each process has a separate memory space → Protects the processes from each other.
- Need a mechanism to determine the range of legal addresses that a process may access.
- Two registers: **base** and **limit**.
  - Base register holds the smallest legal physical memory address.
  - Limit register specifies the size of the range.
  - can be loaded only by the OS using a special privileged instruction in kernel mode!



Supporting Multiprogramming

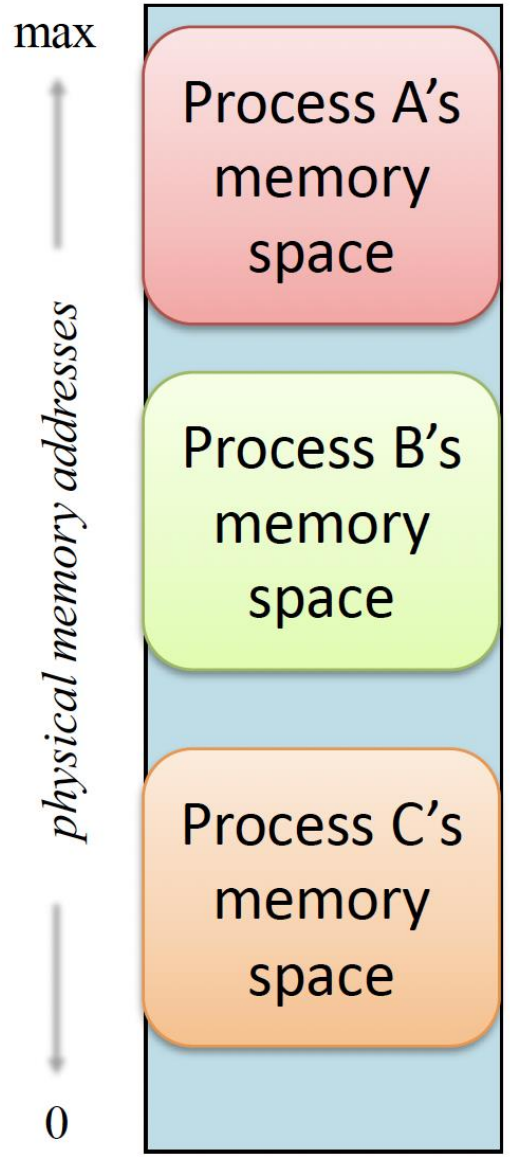


Address Binding

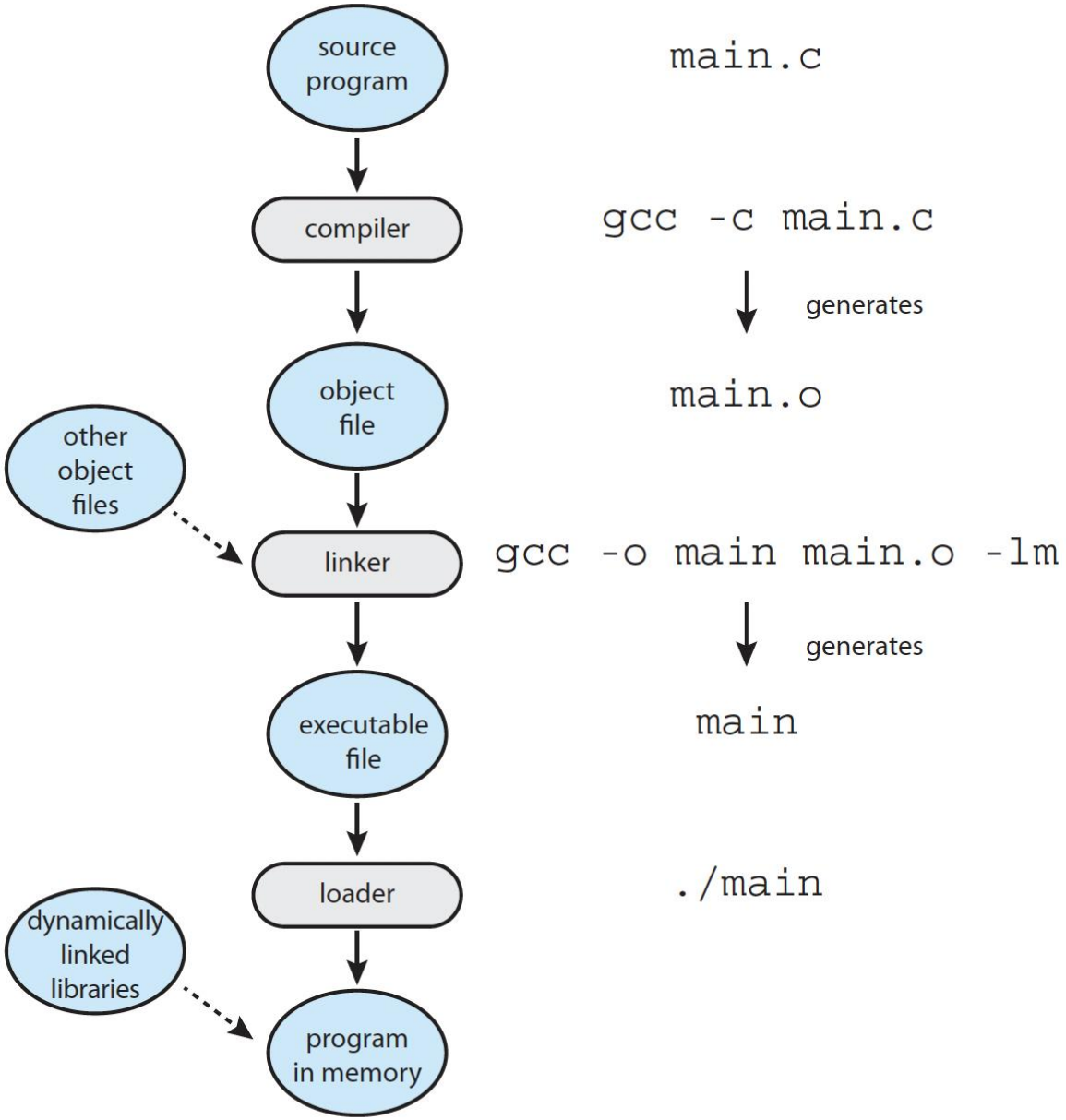
Allowing multiple processes to be resident in memory!

Two ways:

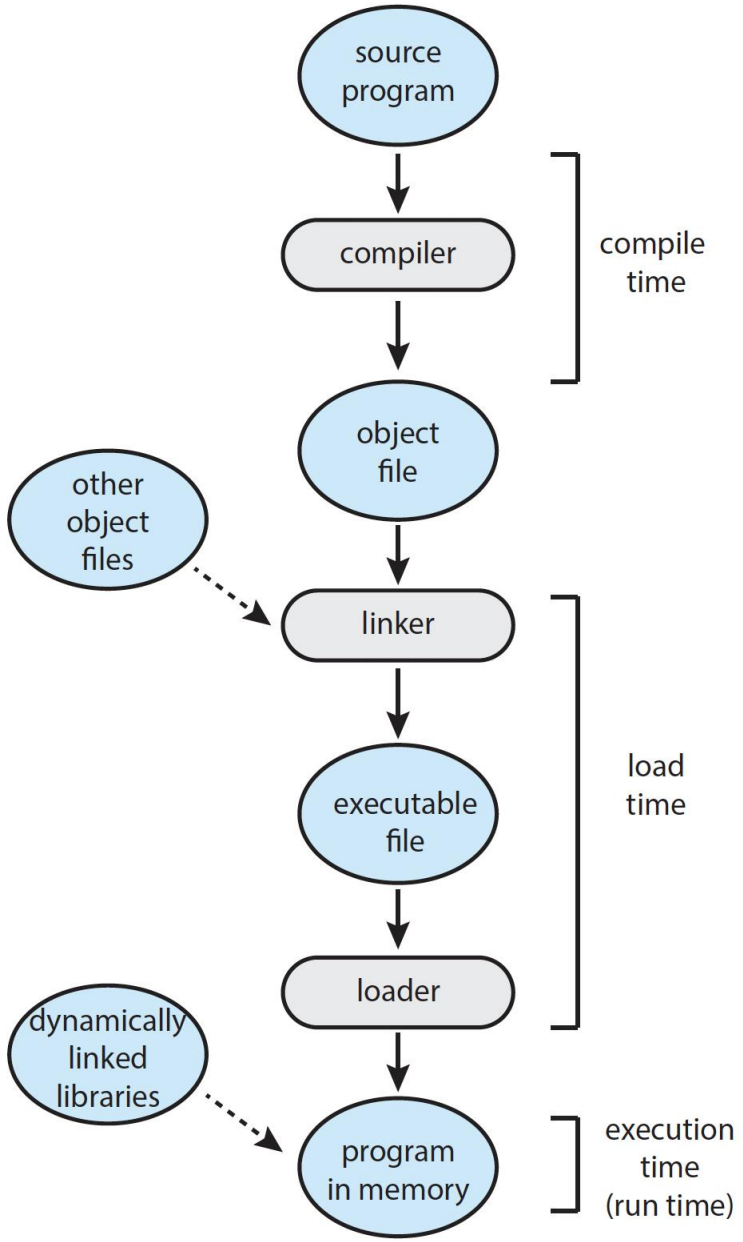
- Contiguous
- Non-Contiguous



Address Binding



Address Binding



Compile-Time Address Binding

- The programmer or system designer has to pre-decide and hardcode a starting address.

- Thus, compiler needs to know where in memory the program will sit at compile time.
  - The compiler can embed the real physical addresses directly into the binary.

- These are called **absolute addresses**.

- If that memory location is unavailable, the program can't run and must be recompiled.
  - Example: MS-DOS.

Load-Time Address Binding

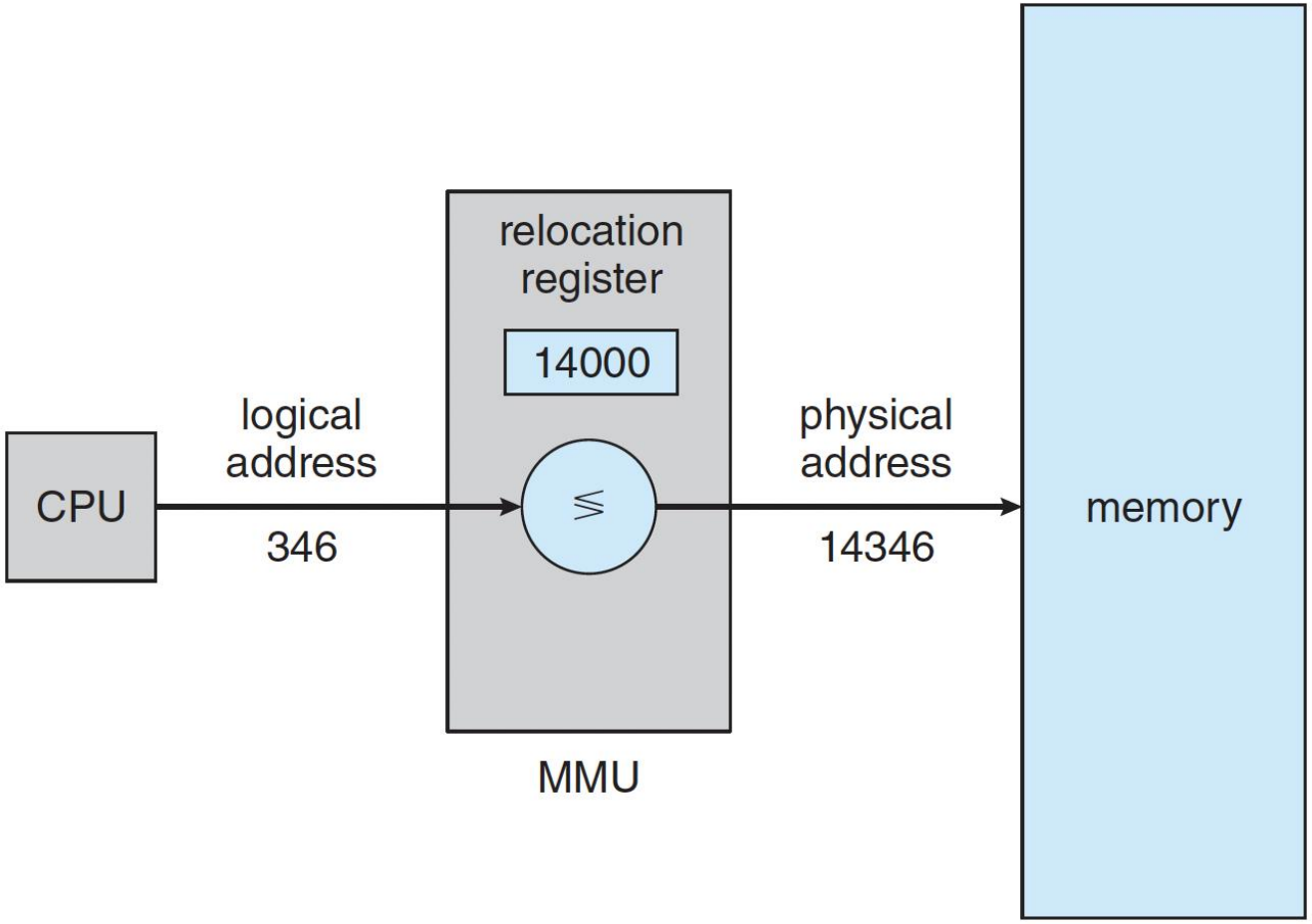
- Also known as relocatable addresses.
- The compiler generates all addresses as offsets from a base (e.g., “base + 4”, “base + 20") without committing to any specific physical location.
- When the OS loads the program into RAM, the loader adds the actual base address to all these offsets to produce real physical addresses.

Execution-Time Address Binding

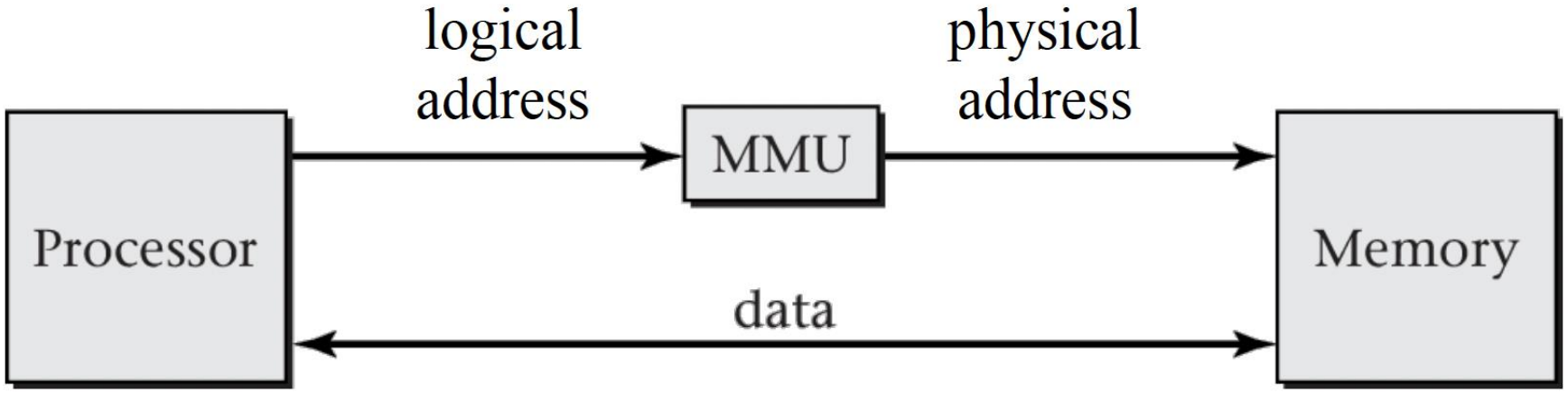
- Relocatable binding limitation:
  - Once the loader picks a base address and fixes all the offsets, the process cannot be moved in memory while it's running .
  - If the OS needs to swap the process out and reload it at a different address, all the fixed physical addresses become wrong.
- Binding must be deferred until runtime.
- The CPU no longer work with physical addresses → uses **logical (virtual) addresses**.
- The hardware chip → Memory Management Unit (MMU) translates them to physical addresses on every single memory access.
- This is how modern OSes like Windows and Linux work.

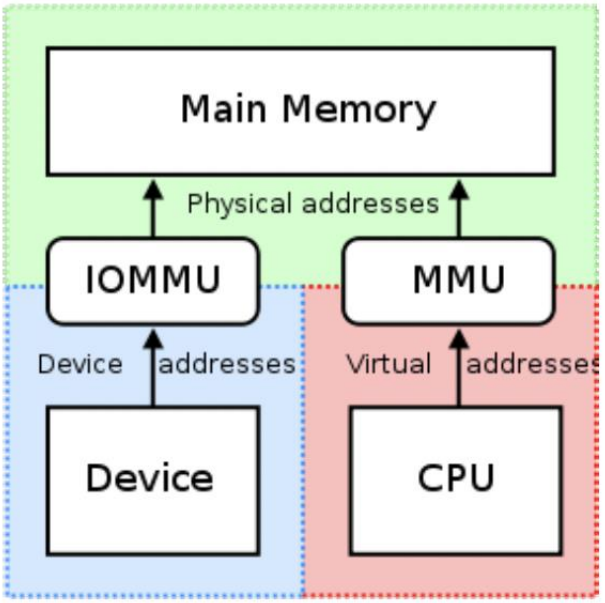
Dynamic Relocation via Relocation Register

- Logical (virtual) address is converted to a physical address by adding the value in the relocation register.
- Each process has its own relocation register.



Memory Management Unit



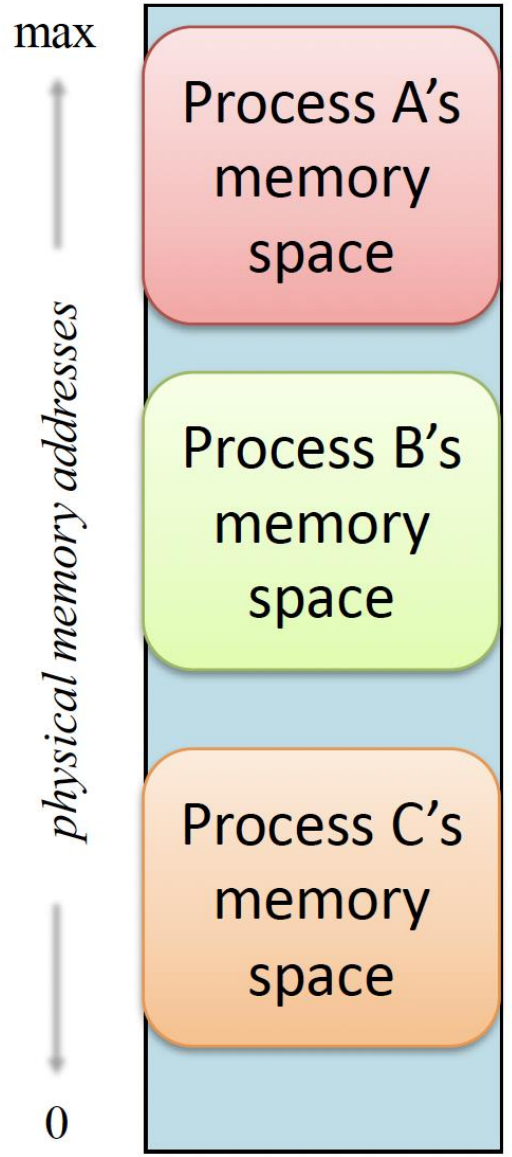


Supporting Multiprogramming

Allowing multiple processes to be resident in memory!

Two ways:

- Contiguous
- Non-Contiguous



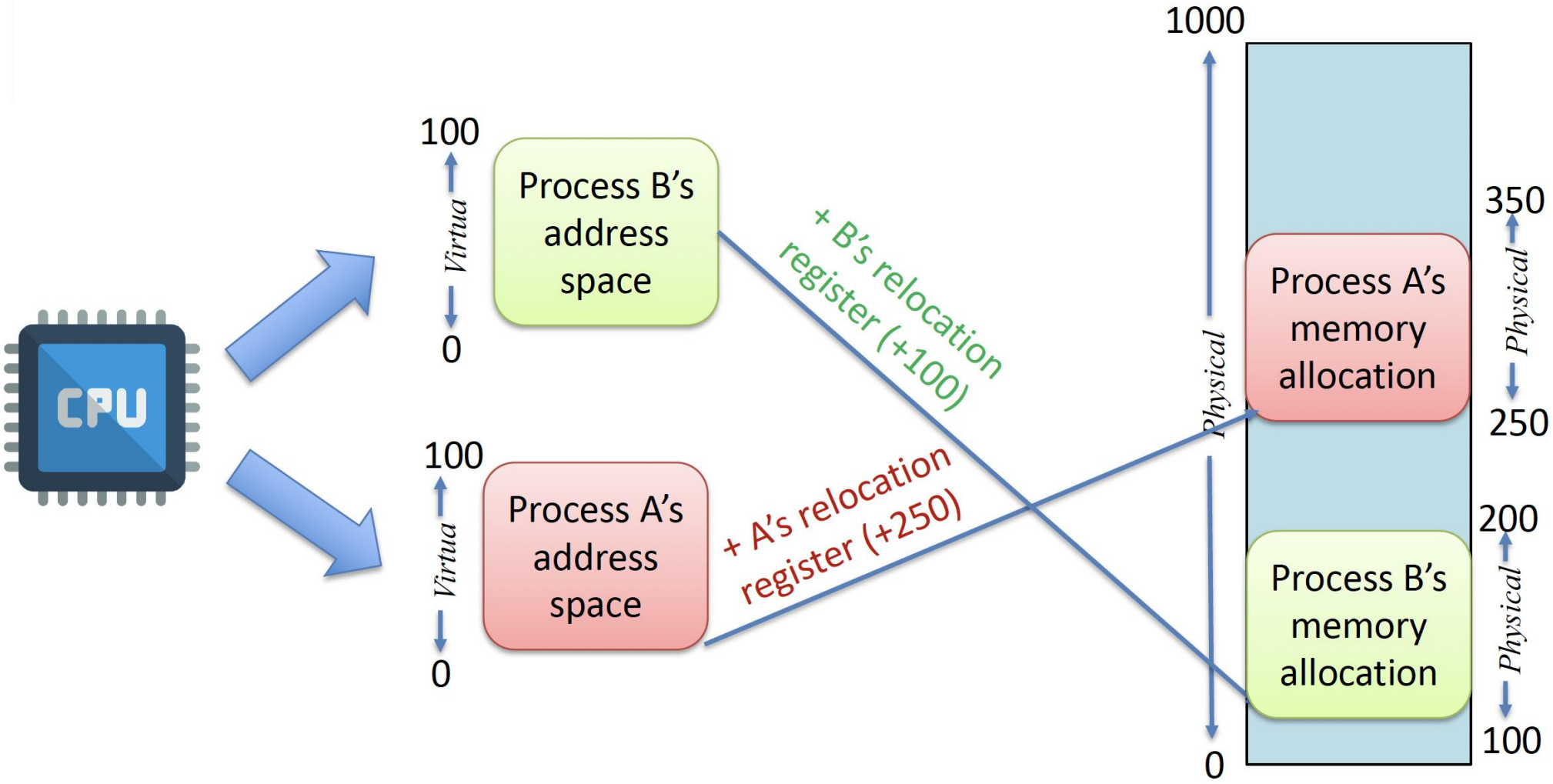
Contiguous Memory Management

- All logical memory used by a process is mapped to a single physical memory region with contiguous addresses.

- How? → Find a physical region for each process
  - Process address space MUST be mapped to a contiguous physical memory allocation.

- Benefits:
  - Less complex for OS to manage
  - Easily implemented in hardware

Contiguous Memory Management



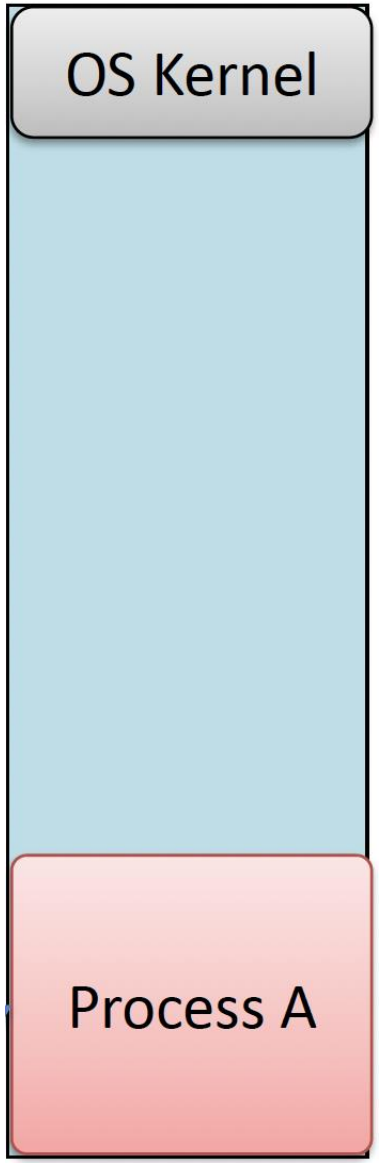
Relocation Register = Base Register

Contiguous Memory Management Limitations?

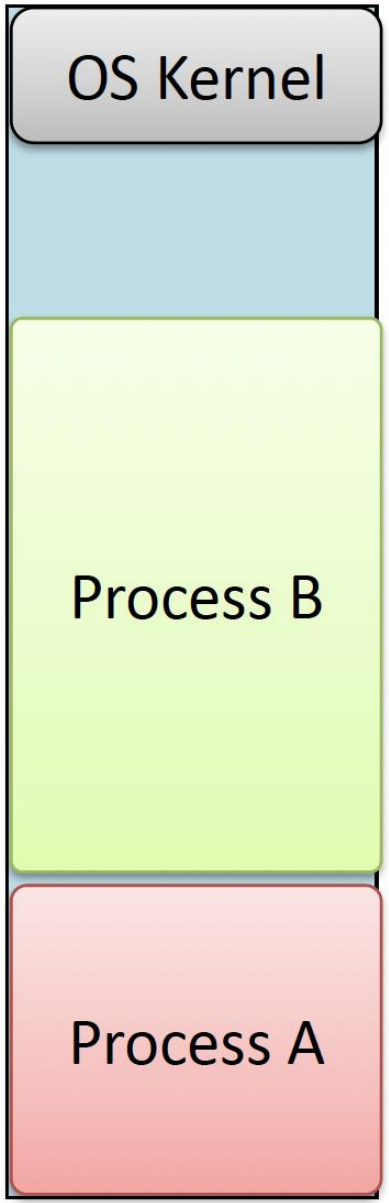
Contiguous Memory Management Limitations?

- One process per partition.
  - Degree of multiprogramming limited by number of partitions.
- Variable-partition sizes for efficiency.
  - Sized to a given process' needs.
- Free partitions of 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 are combined.
- OS maintains information about allocated/free partitions.

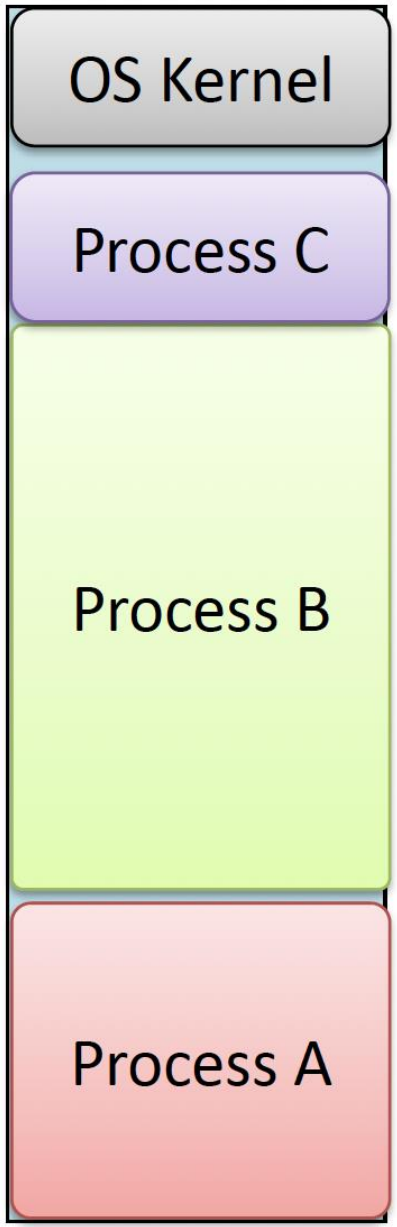
Contiguous Memory Management



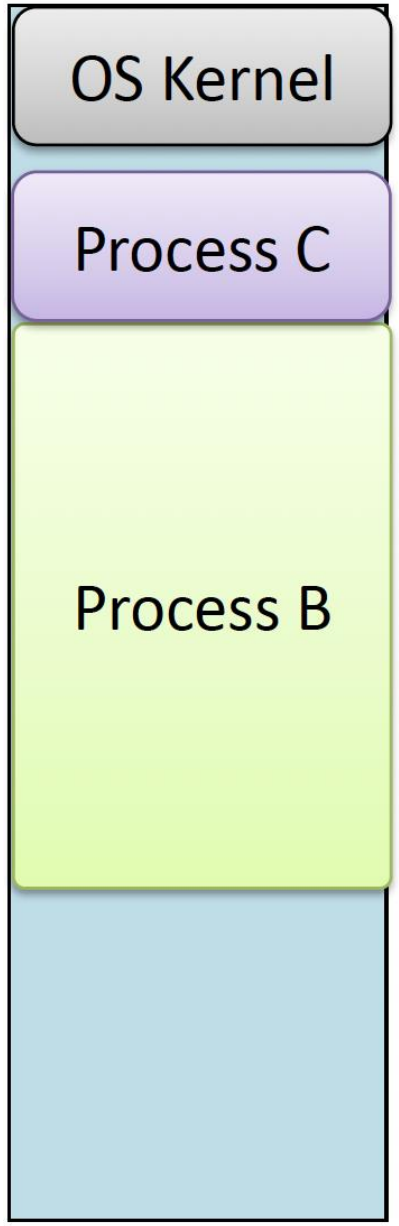
Contiguous Memory Management



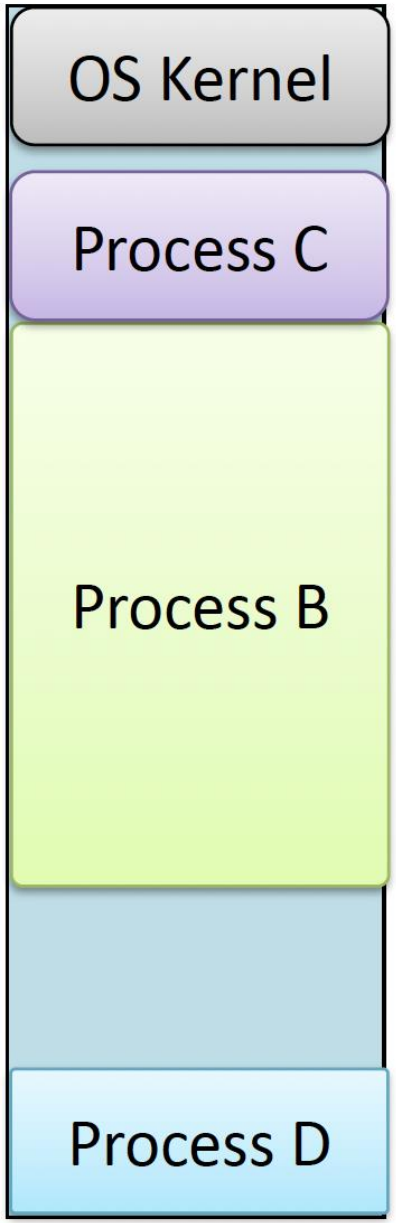
Contiguous Memory Management



Contiguous Memory Management

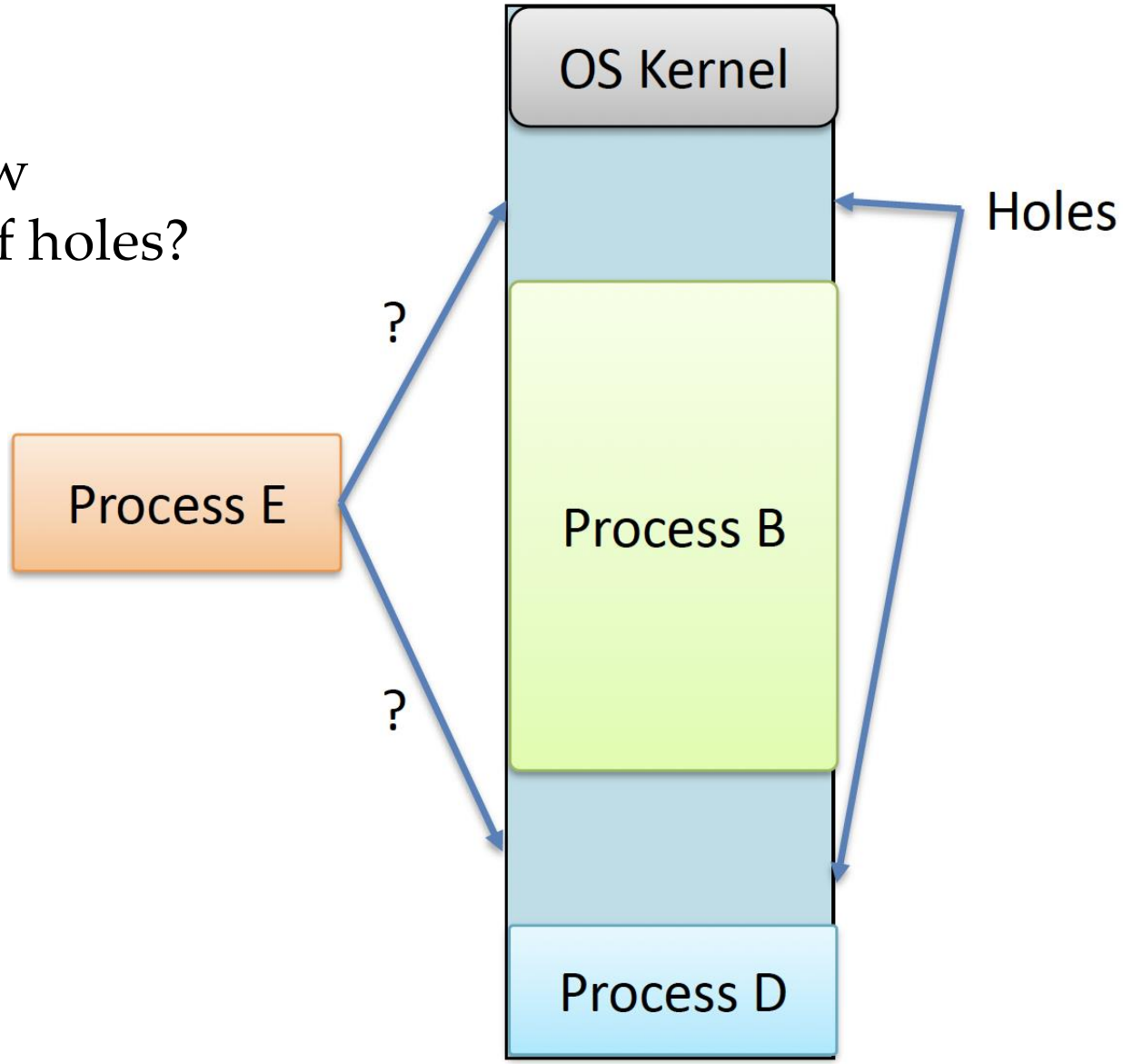


Contiguous Memory Management



Contiguous Memory Management

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



Dynamic Storage Allocation Problem

- **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.

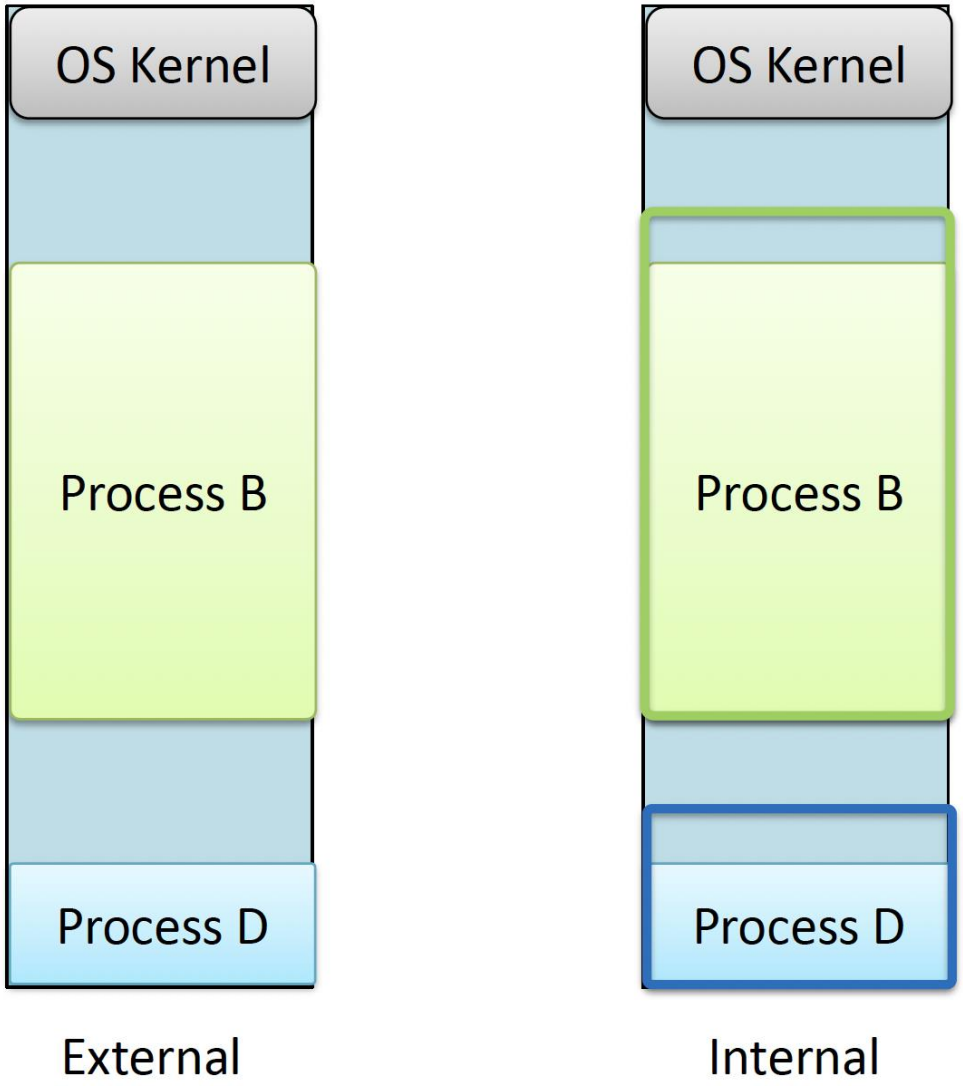
Fragmentation

Fragmentation

- There is enough total memory space to satisfy a request but the available spaces are not contiguous.
  - Wastes memory space.
  - Decrease degree of multiprogramming.

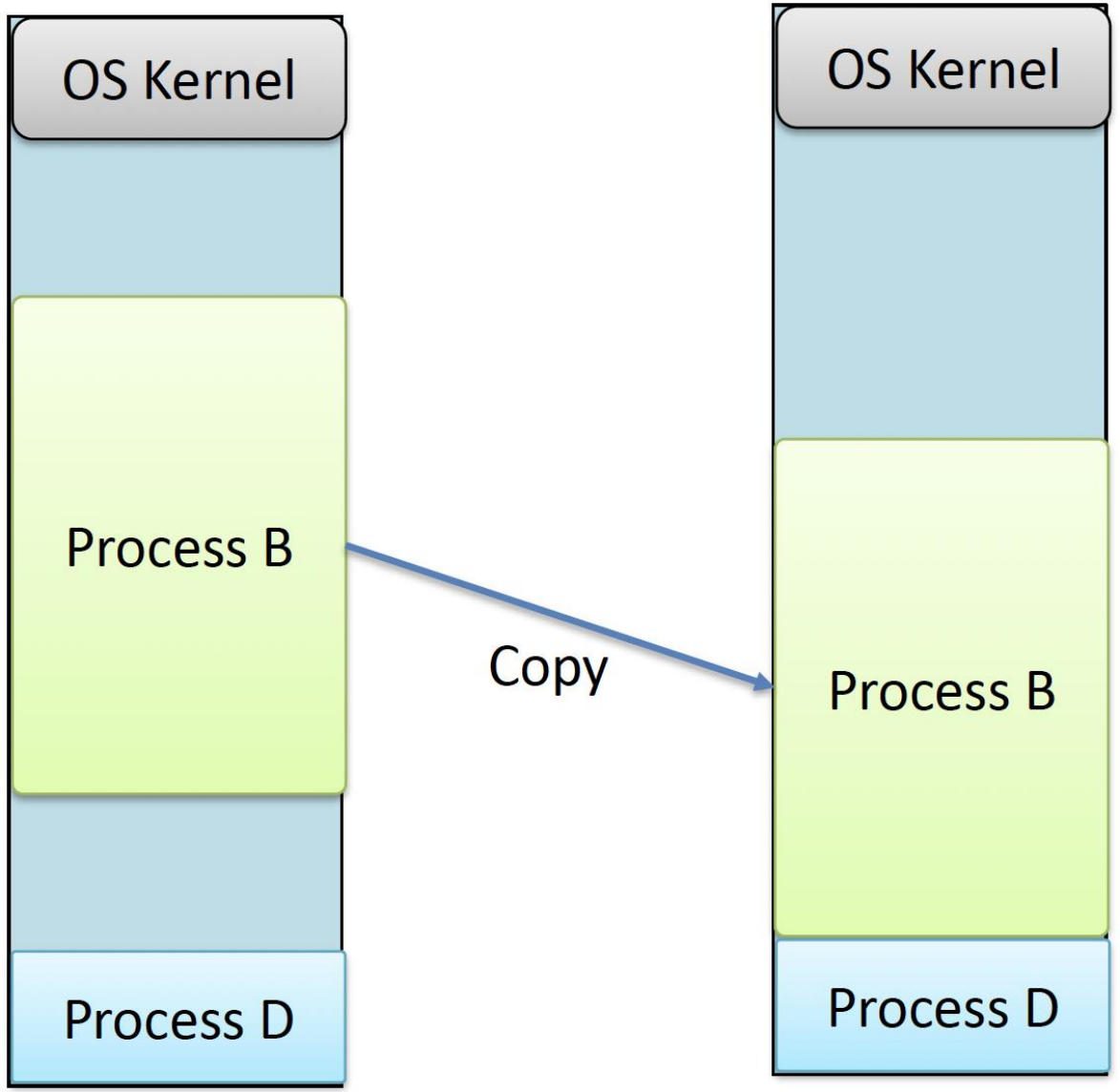
Fragmentation

- **External fragmentation**
  - Total memory space exists to satisfy a request, but it is not contiguous.
- **Internal fragmentation**
  - Allocated memory may be slightly larger than requested memory.



Compaction

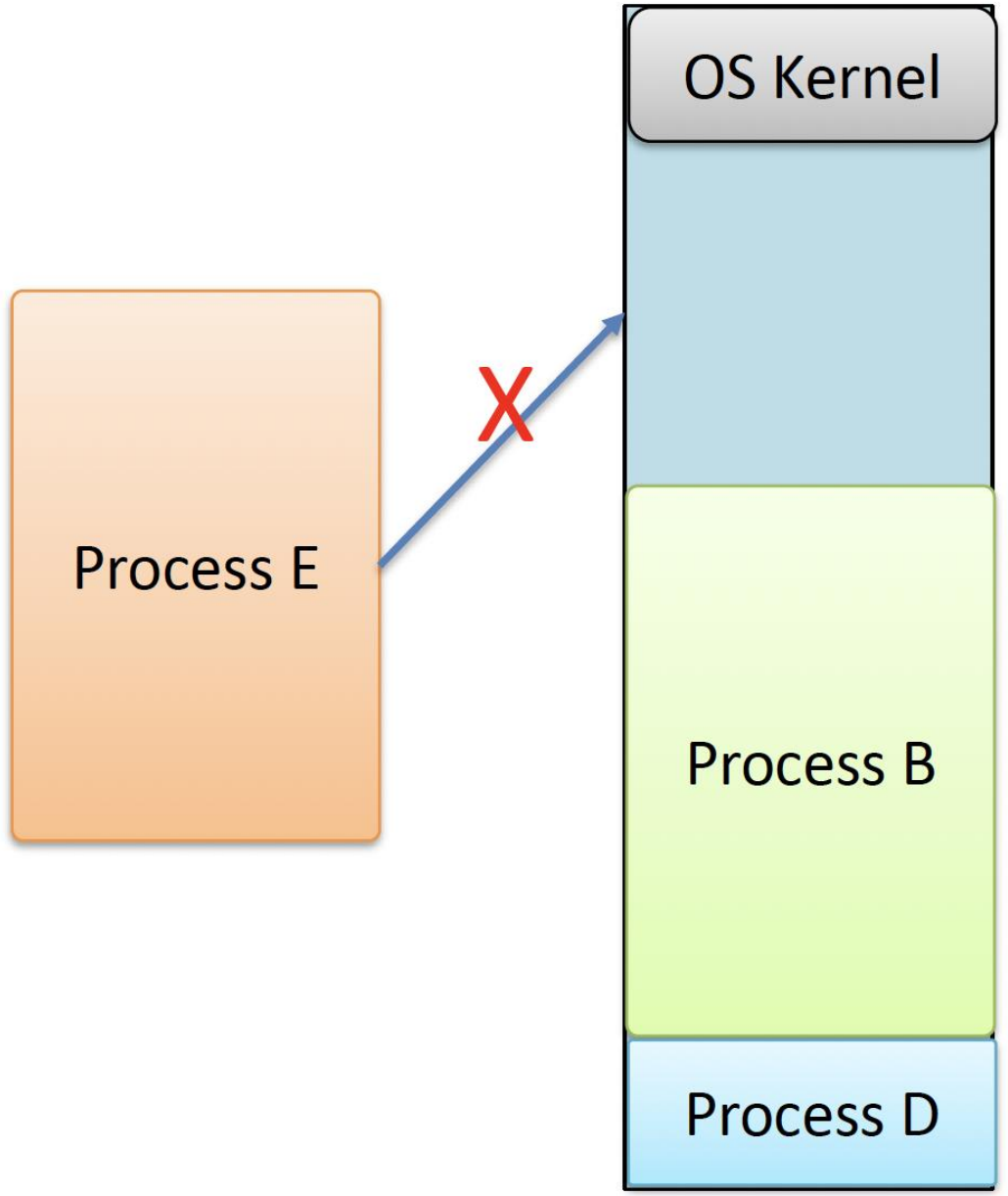
- Reduce external fragmentation by compaction.
- Shuffle memory contents to place all free memory together in one large block.
- Compaction is possible only if relocation is dynamic, and is done at execution time.
- Cost → memory copy.



What happens when the total physical memory space requested by processes exceeds physical memory?

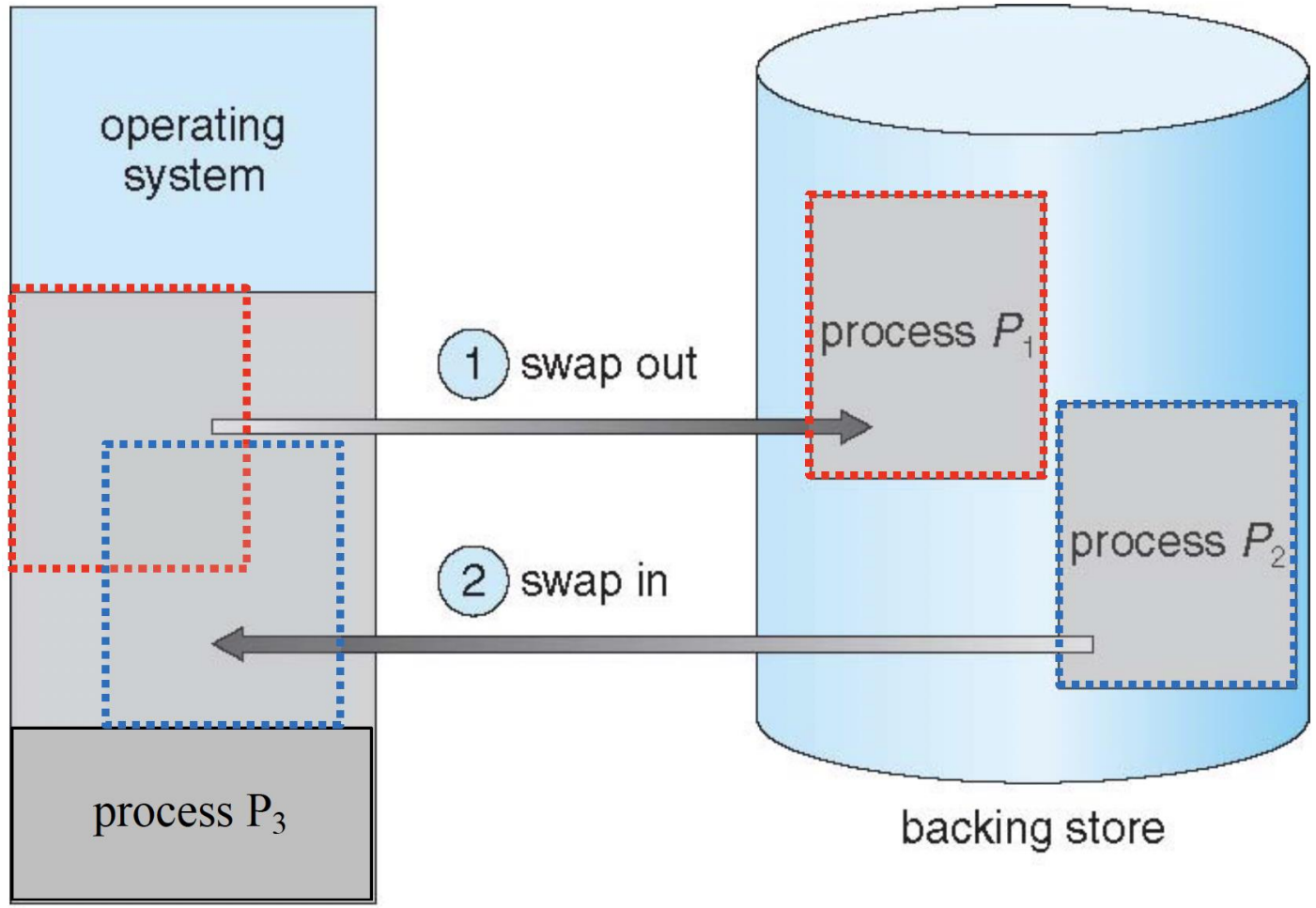
Swapping

Swapping allows the total physical address space of all processes to exceed the real physical memory of the system.



Swapping

- The backing store is commonly fast secondary storage.
- Must be large enough to accommodate whatever parts of processes need to be stored and retrieved.
- When a process is swapped to the backing store, the data structures associated with the process must be written to the backing store.



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

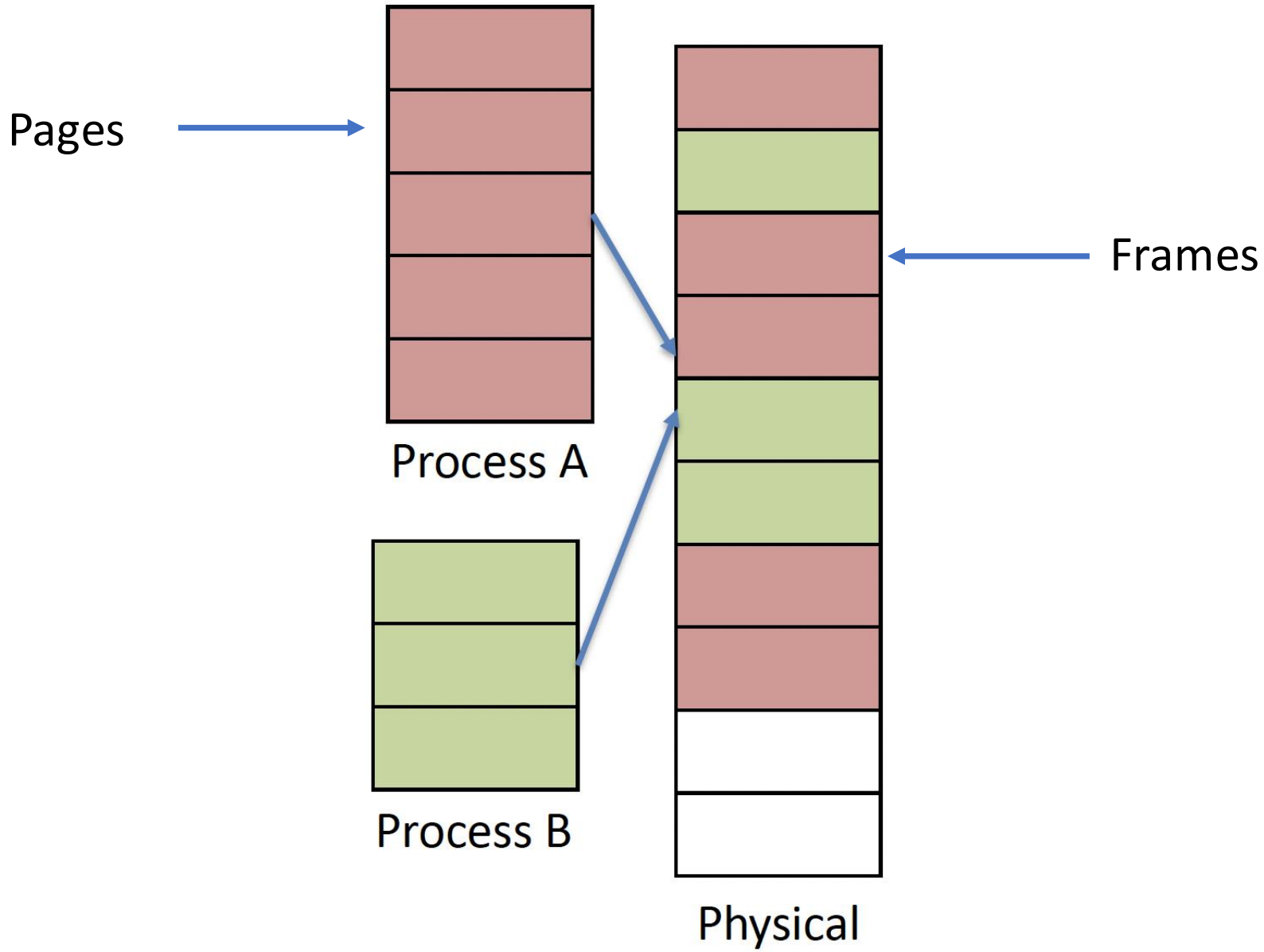
Non-Contiguous Memory Management

Logical memory used by a process can be mapped to multiple physical memory regions.

Page-based Memory Management

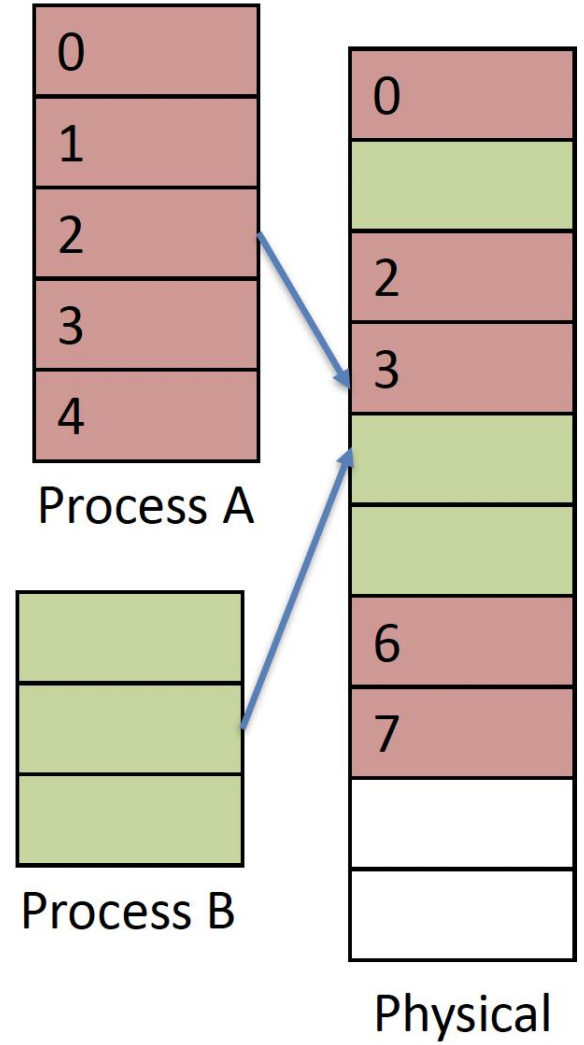
- Divide physical memory into fixed-sized blocks → **Frames**.
  - Size is power of 2 → usually 512 bytes to 4K bytes.
- Divide logical memory into fixed-sized blocks → **Pages**.
  - Same size as frames.
- Each page is allocated to some frame.
  - OS tracks frame-to-page assignment and all free frames.
  - Frames assigned do not have to be contiguous.
- To run a program of size **N** pages:
  - Find **N** free frames and assign them to **N** pages.
  - Load program.

Page-based Memory Management



Page-based Memory Management

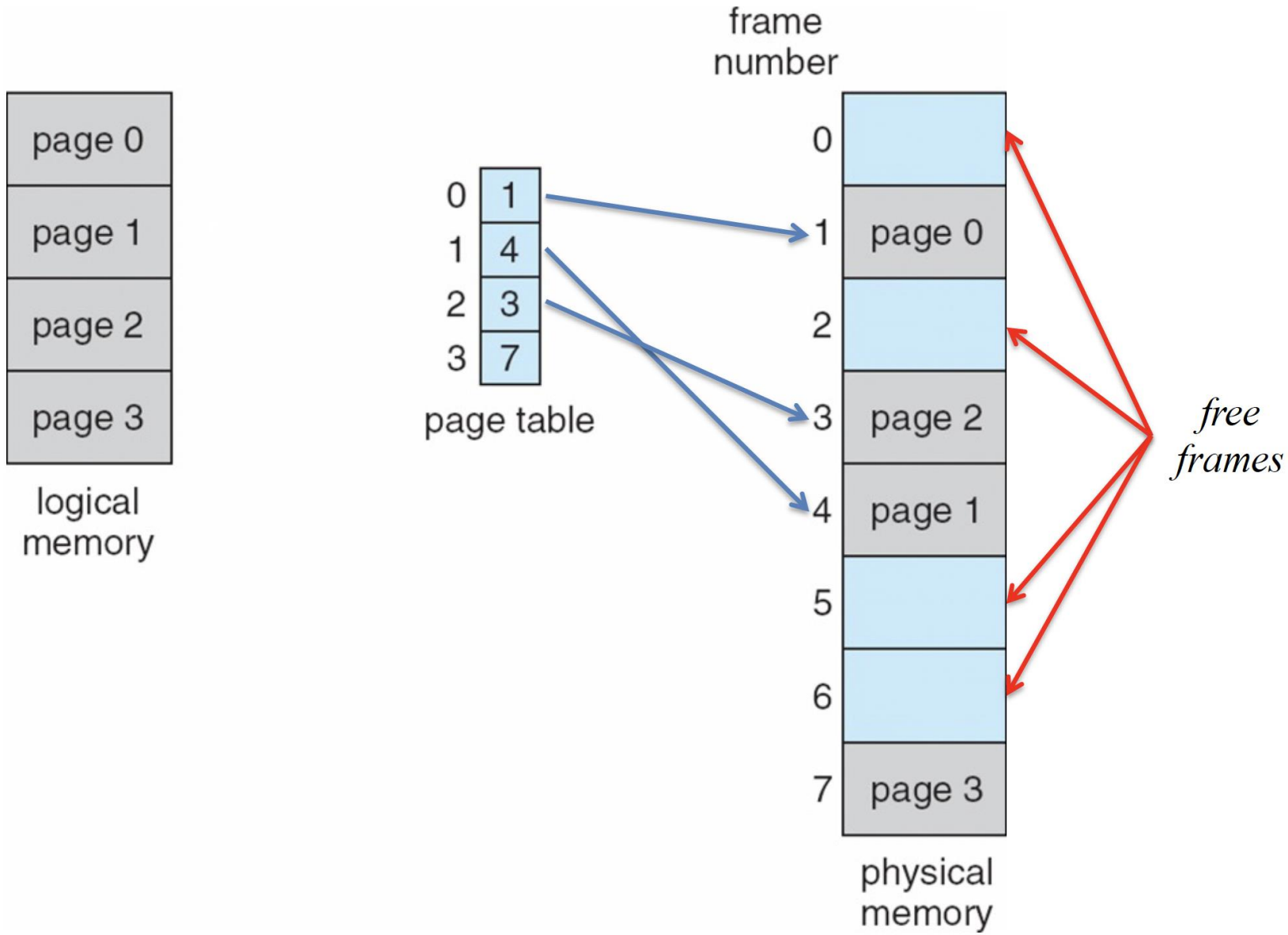
- **Page Table**:
  - To keep track of frame-to-page allocation.
  - Maps logical pages to physical frames.
  - One Page table per process.



Page table for process A:

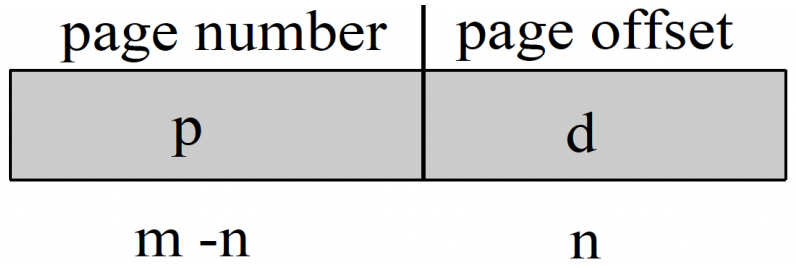
Page id	Frame id
0	0
1	2
2	3
3	6
4	7

Page-based Memory Management



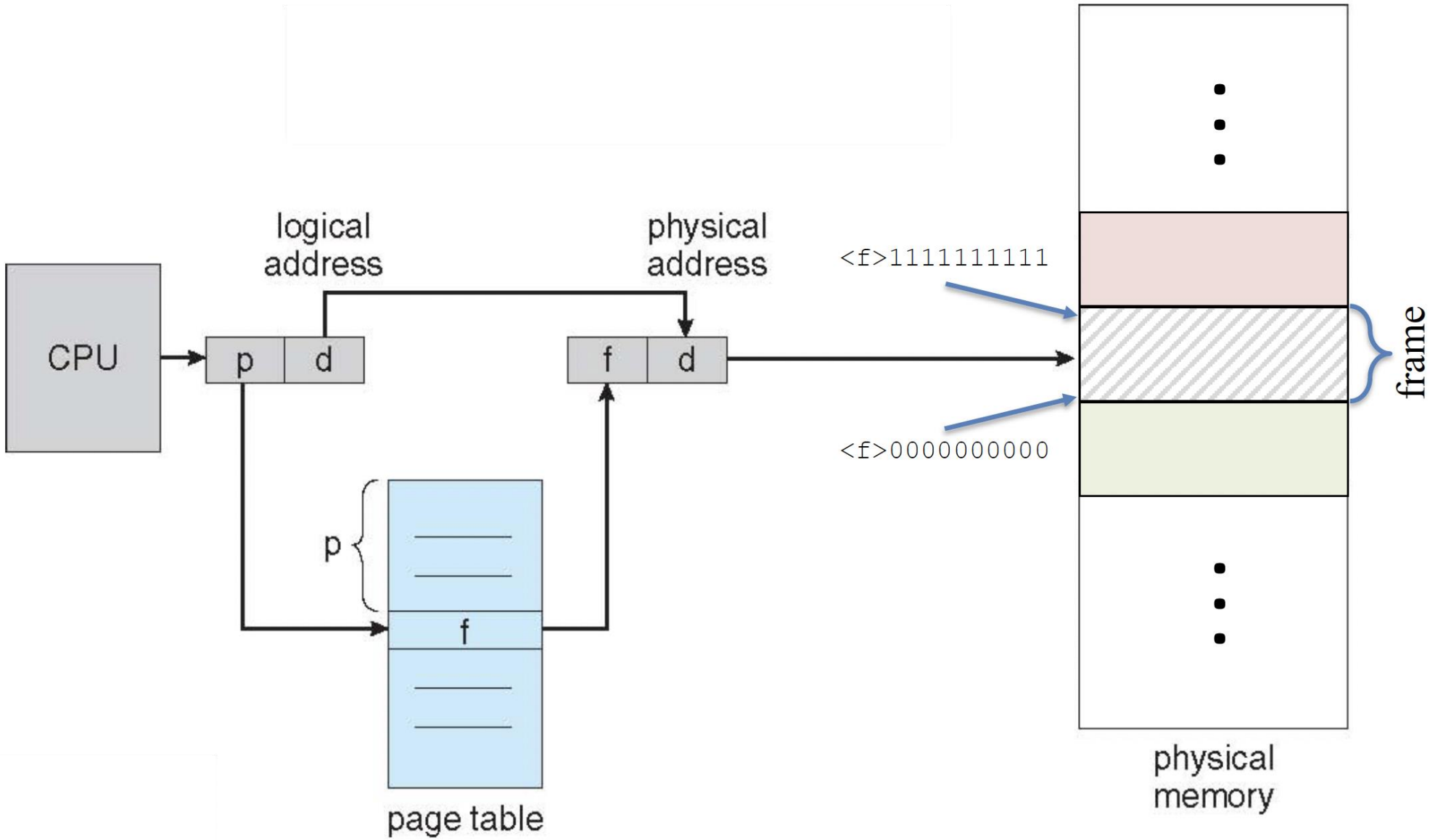
Page Address Translation Scheme

- Every virtual address is divided into two parts:
  - **Page number (p)** – used as an index into a page table which contains base address of each page in physical memory.
    - MMU looks up this number in the page table and finds the corresponding frame number.
  - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit.
    - Tells the MMU where within that page the data lives, essentially, how many bytes into the page you need to go.

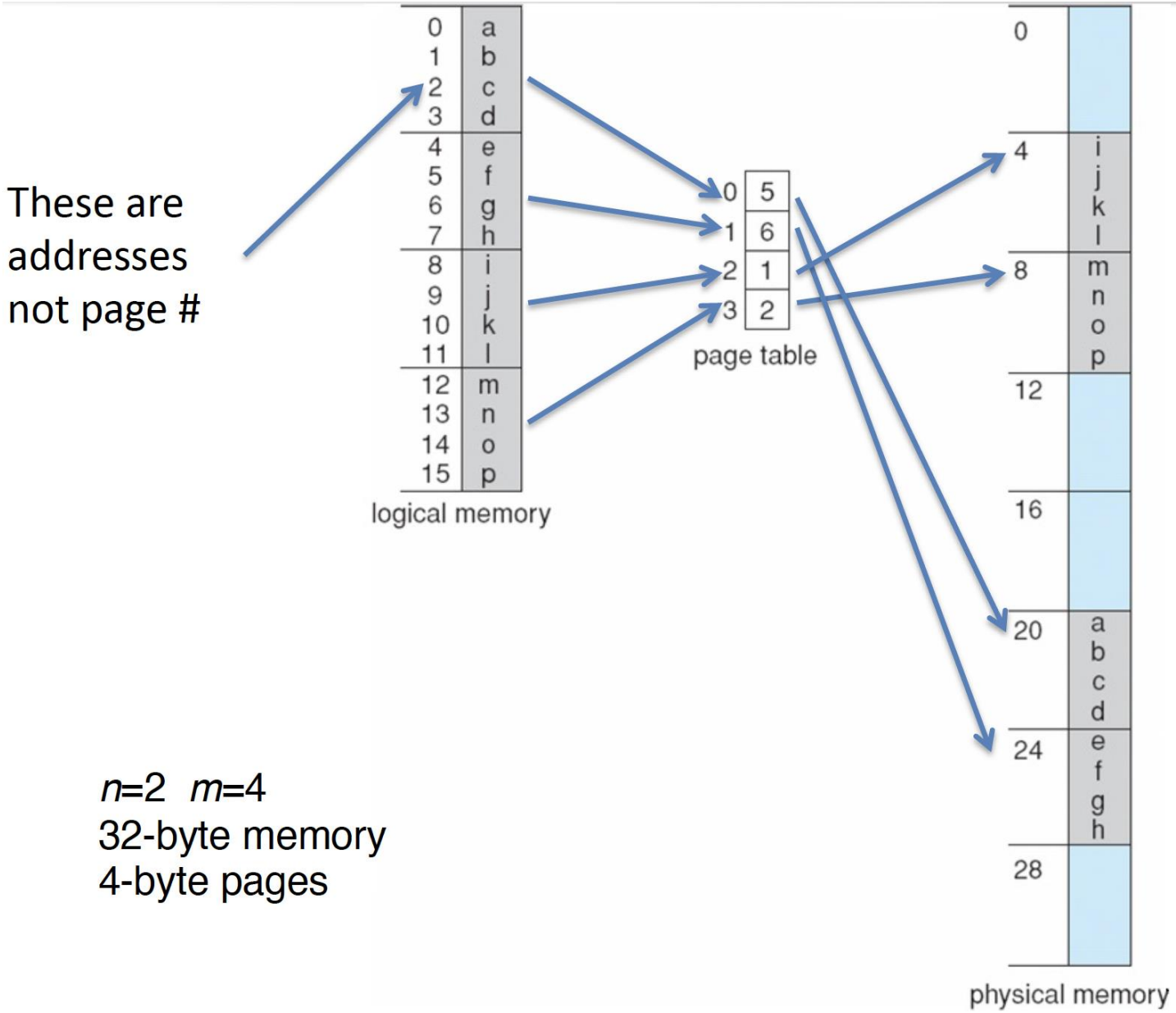


- For given logical address space 2^m and page size 2^n.

Paging Hardware (Address Translation)

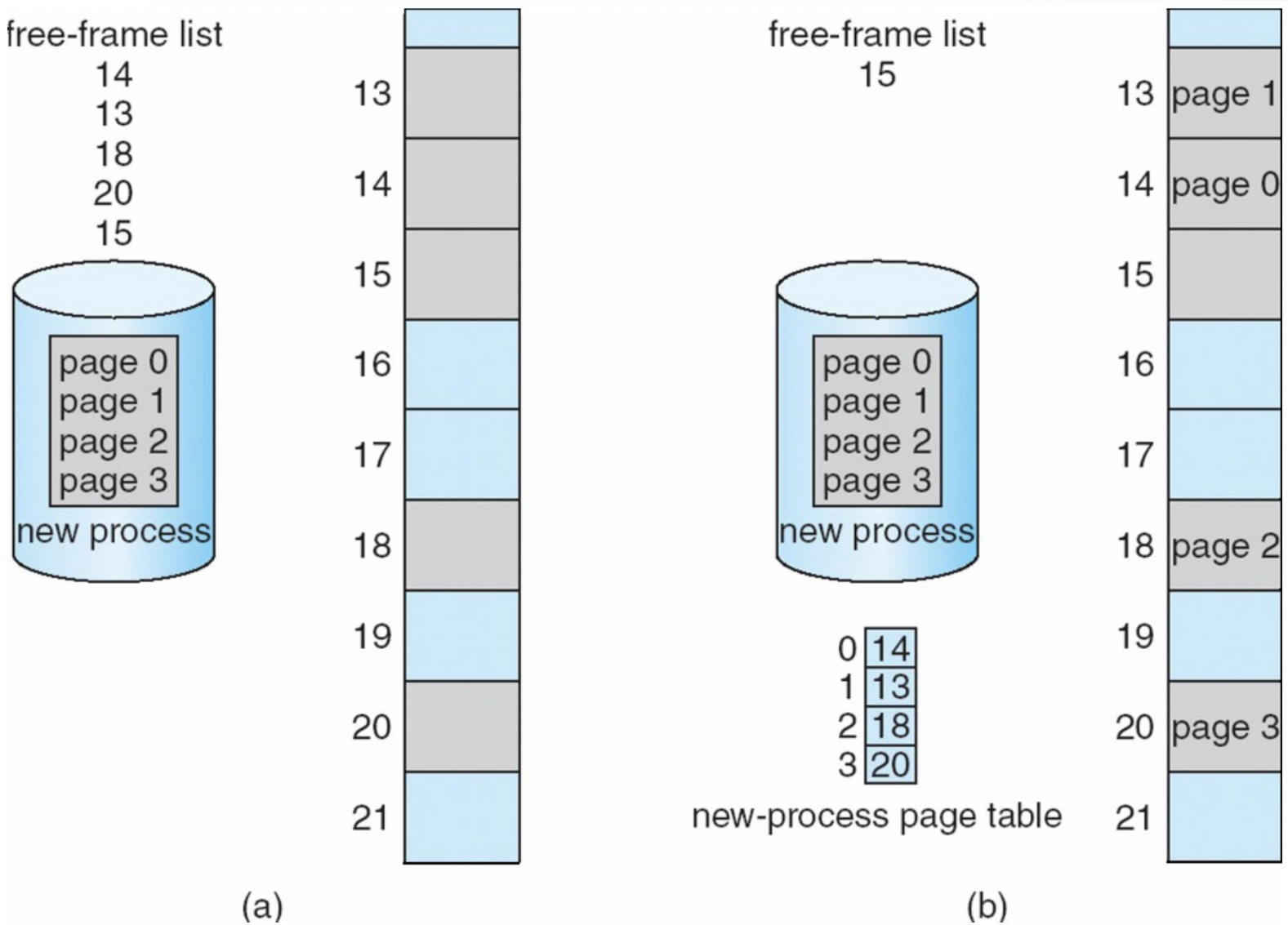


Paging Example



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

Free Frames

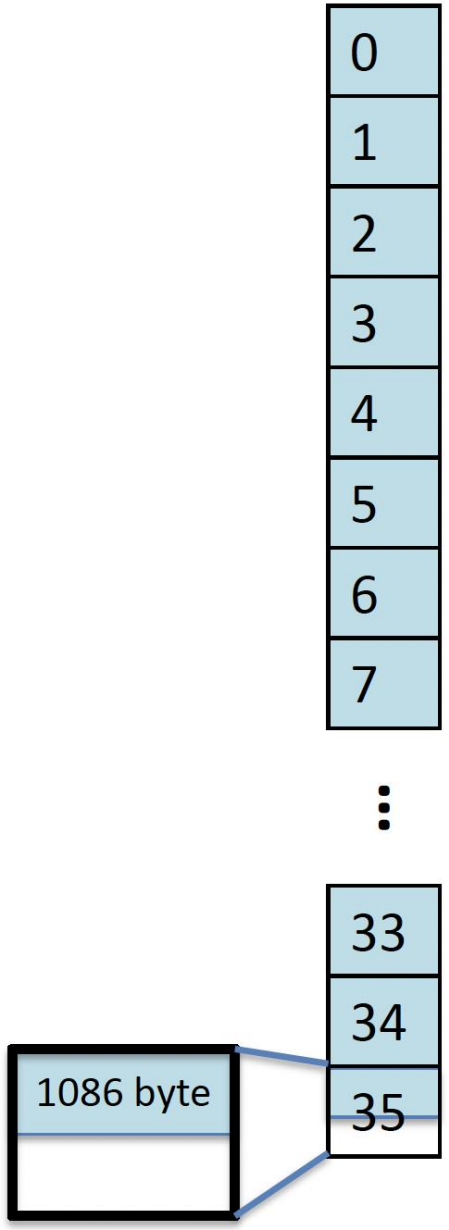


Before allocation

After allocation

Internal fragmentation

- Calculating internal fragmentation
  - Page size = 2048 bytes
  - Process size = 72,766 bytes (different bytes addressed)
  - 35 pages (all full) + 1086 bytes (in last page) (best case)
  - Internal fragmentation of 2048 - 1086 = 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.



Page Table Implementation

Page Table Implementation

- Page table is kept in main memory.

- To locate a page table:
  - **Page-table base register (PTBR)**
    - Points to the page table.

  - **Page-table length register (PTLR)**
    - Indicates size of the page table.

- Stored in PCB and needed during context switch.

Page Table Implementation

- Page tables can quickly become extremely large.

- Consider a 32-bit logical address space
  - Page size of 4 KB (212)
  - Page table would have 1 million entries (232 / 212)
  - If each entry is 4 bytes -> 4 MB for page table alone!!!

- Need approaches to address large page table problem.

Page Table Implementation

- Page tables can quickly become extremely large.

- Consider a 32-bit logical address space
  - Page size of 4 KB (212)
  - Page table would have 1 million entries (232 / 212)
  - If each entry is 4 bytes -> 4 MB for page table alone!!!

- Need approaches to address large page table problem.
  - Hierarchical paging
  - Hashed page tables
  - Inverted page tables

Hierarchical Page Tables

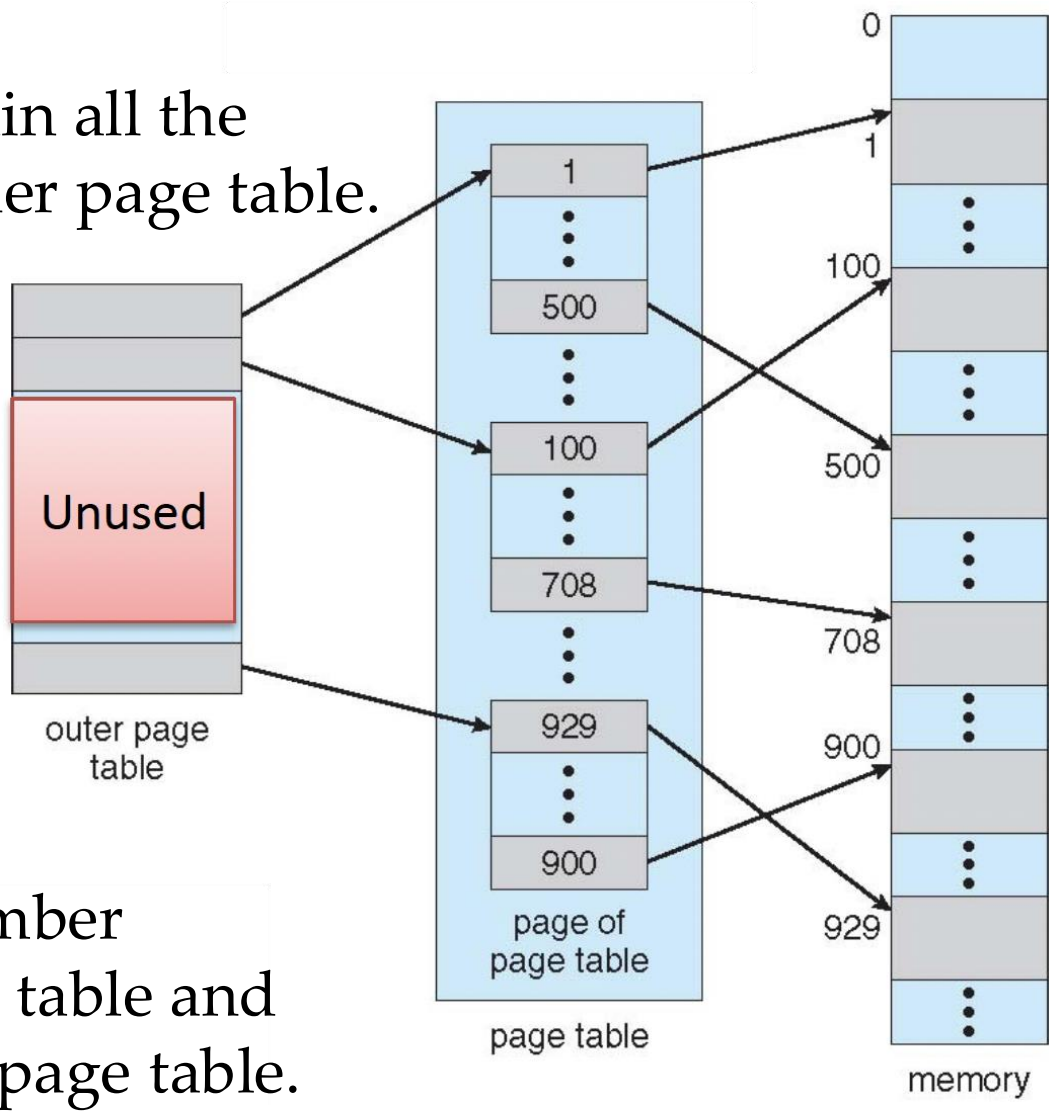
- Break up the logical address space into multiple page tables.

- Use multiple levels to keep track of them.
  - Hierarchy of page tables.
  - Page number is used index across the hierarchy.

- Example: a two-level page table.
  - Page table itself is allocated in pages.

Two-Level Page-Table Scheme

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



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

Two-Level Page-Table Scheme

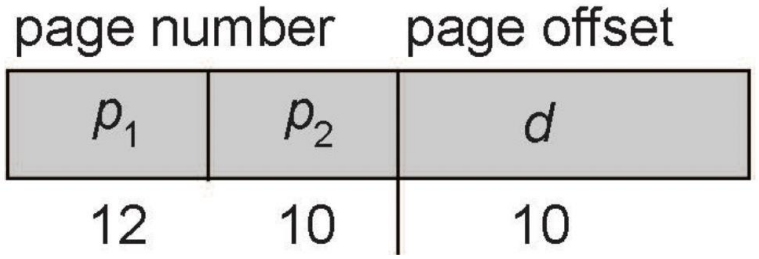
- The PCB stores only the physical address of the outer (top-level) page table.

- The page table structure itself can also be paged.
  - The OS treats inner page tables as regular memory pages; they can be swapped out to disk if memory is tight.

- When a process is swapped back in :
  - The outer page table is brought into RAM first (its address is in the PCB).
  - Next, the inner page table is brought in on demand → only when the process actually tries to access a page that requires it.
  - If the inner page table itself is not in RAM, a **page fault** occurs, the OS fetches it from disk, and then resumes the address translation

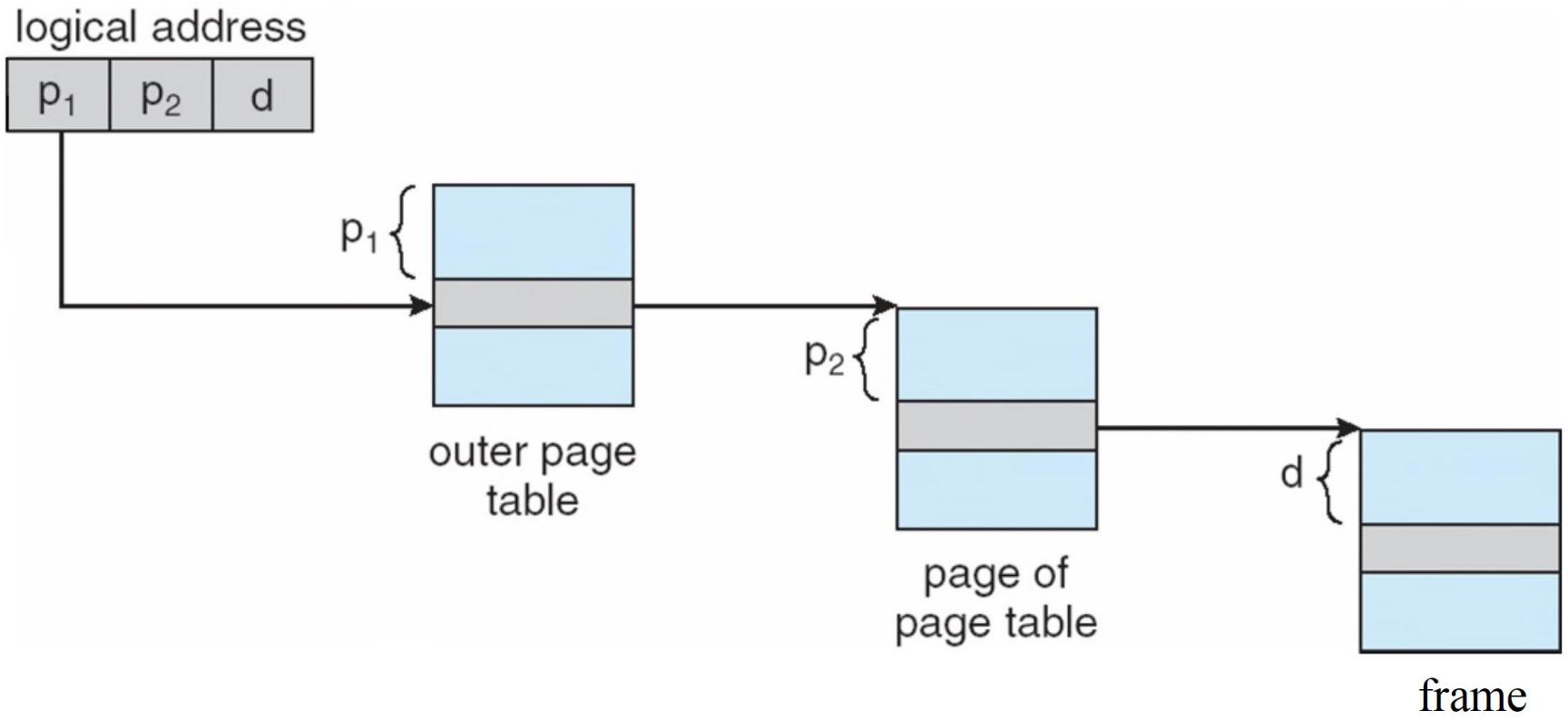
Two-Level Page-Table Scheme

- 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 p1 is an index into the outer page table, and p2 is the displacement within the page of the inner page table.

Two-Level Page-Table Scheme



Hashed Page Tables

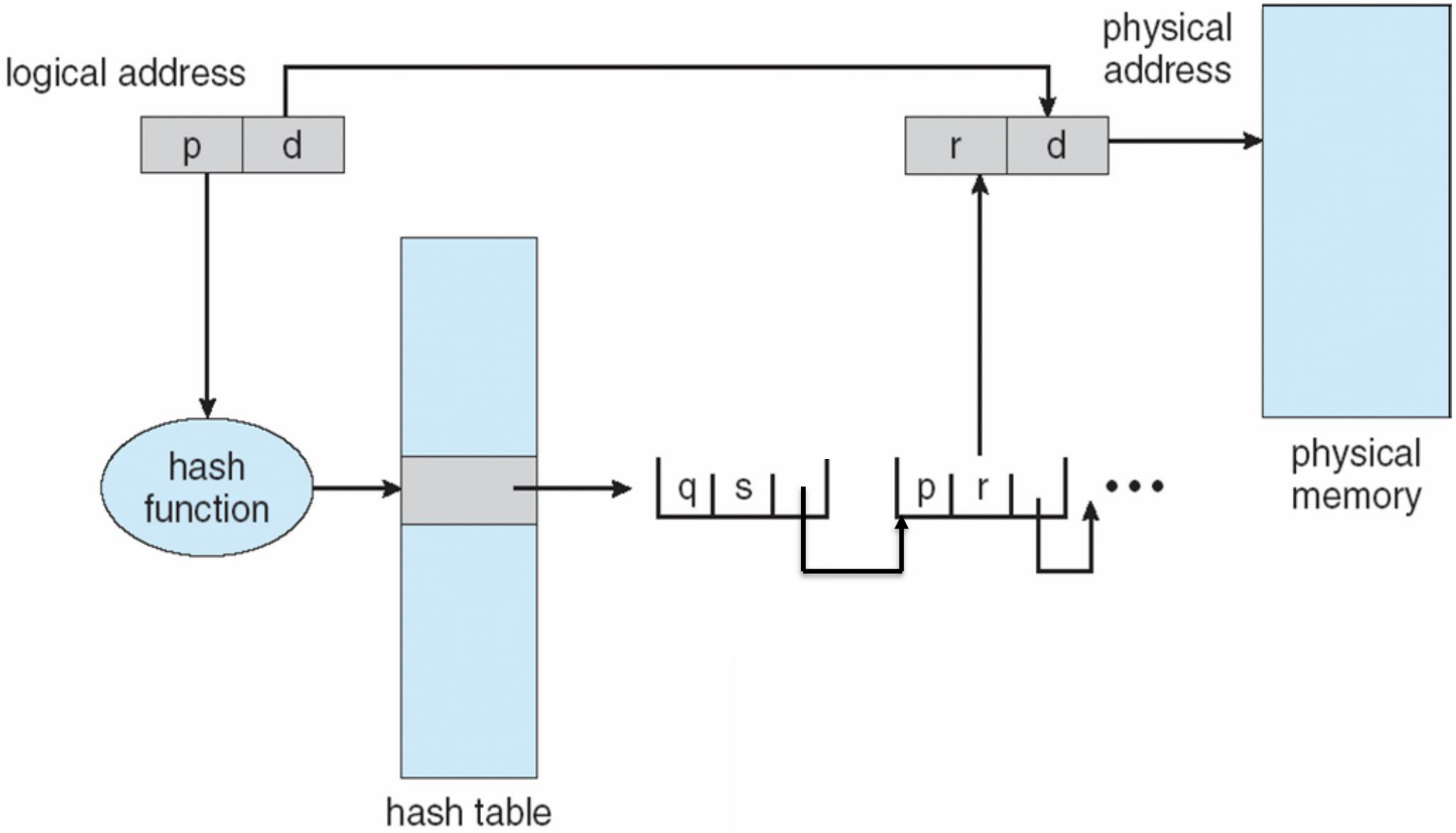
- Common in address spaces > 32 bits.

- 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 such a case.

- Each entry contains:
  - the page number.
  - the value of the mapped page frame.
  - 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.

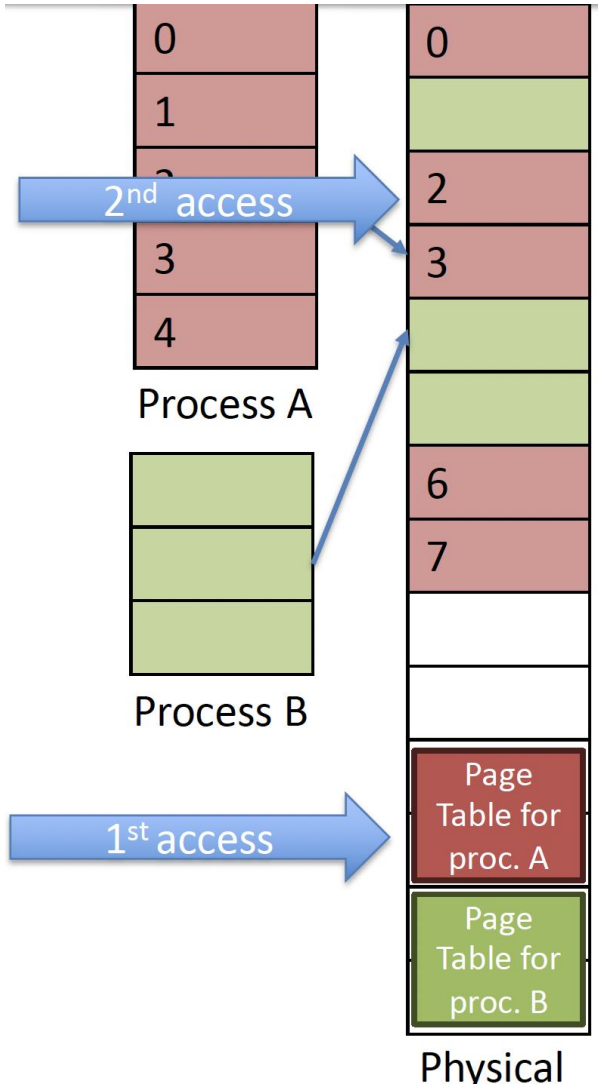
Hashed Page Tables



How many memory accesses for each data/instruction?

How many memory accesses for each data/instruction?

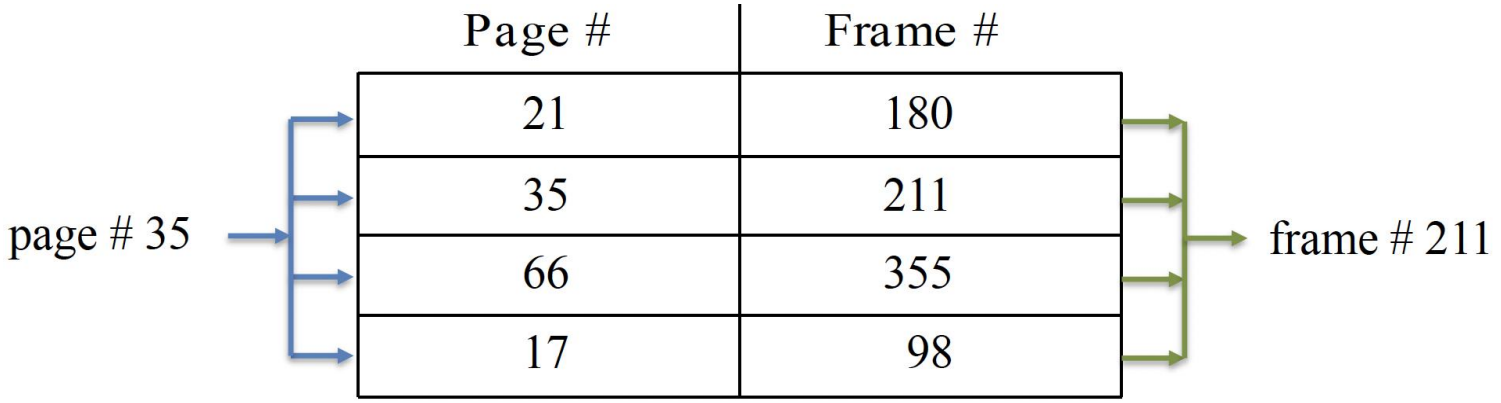
- Every data/instruction reference requires **two** memory accesses.
  - One for the page table and one for the data / instruction.
- Address translation must be fast!
  - Wasting a memory reference to access the page table.
  - 50% overhead.
  - **Solution?**



Translation Look-Aside Buffer (TLB)

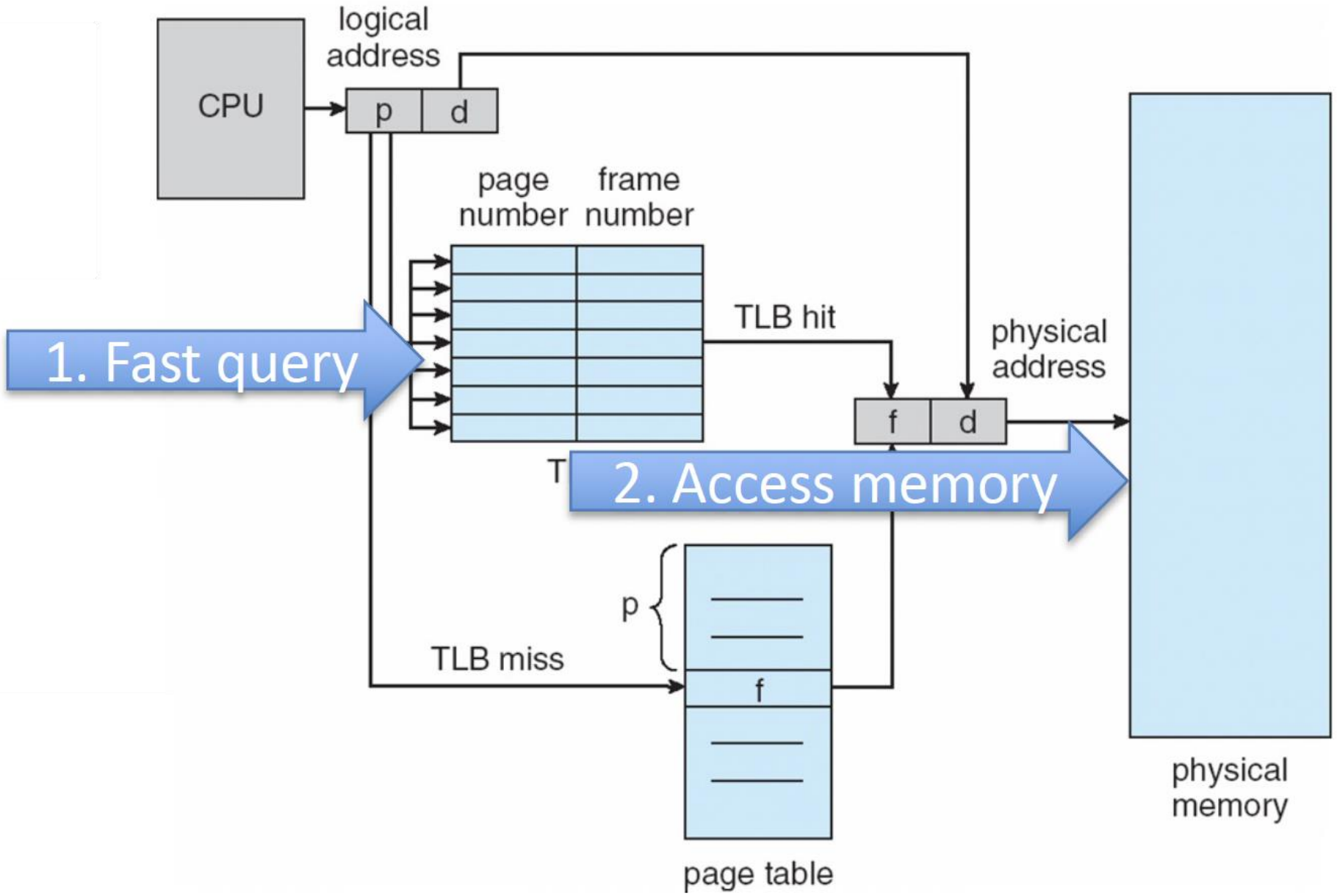
- Translation look-aside buffer (TLB)
  - Hardware cache for page tables.
  - Implemented with high-speed associative memory.
  - TLBs are generally small (64 to 1,024 entries, possibly larger).
  - Holds a small amount of page-to-frame mapping info.

- A TLB for a page table holds <page, frame> pairs.



- Implemented with associative memory.
  - Enables access by "value" instead of by "address" .

Translation Look-Aside Buffer (TLB)



Translation Look-Aside Buffer (TLB)

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

- **With TLB:** query TLB (cache)
  - If hit → access frame (memory)
  - If miss → access page table (memory) → access frame(memory)
  - Some OS may perform both routes in parallel!

- TLB miss occurs when page number is not present.
  - Next load the necessary <page #, frame#> into the TLB
  - Faster access next time.
  - May need to replace an existing entry from TLB.

TLB with multiple processes?

- Only one TLB but multiple concurrent processes!

- Each process has its own page table.
  - After a context switch the TLB's cached translations belong to the old process.

- Naive solution → flush (wipe) the entire TLB on every context switch.

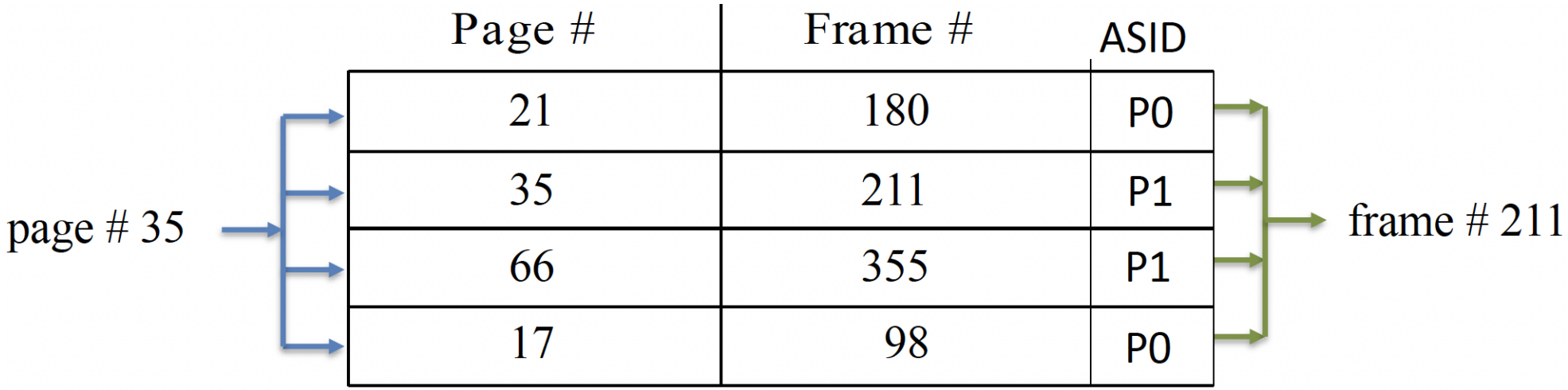
- Use an **Address Space Identifier (ASID)**.
  - A small tag (typically 8–16 bits) stored alongside each TLB entry.
  - It uniquely identifies which process that translation belongs to.

Address Space Identifier

- When the MMU looks up a virtual address in the TLB, it checks two things:
  - The virtual page number matches.
  - The ASID matches the currently running process's ASID.

- If the ASID doesn't match, it's treated as a TLB miss, even if the virtual address happens to be the same number.

- No need of flushing TLB.



TLB with multiple processes?

- Only one TLB but multiple concurrent processes!

Effective Access Time (EAT) with TLB

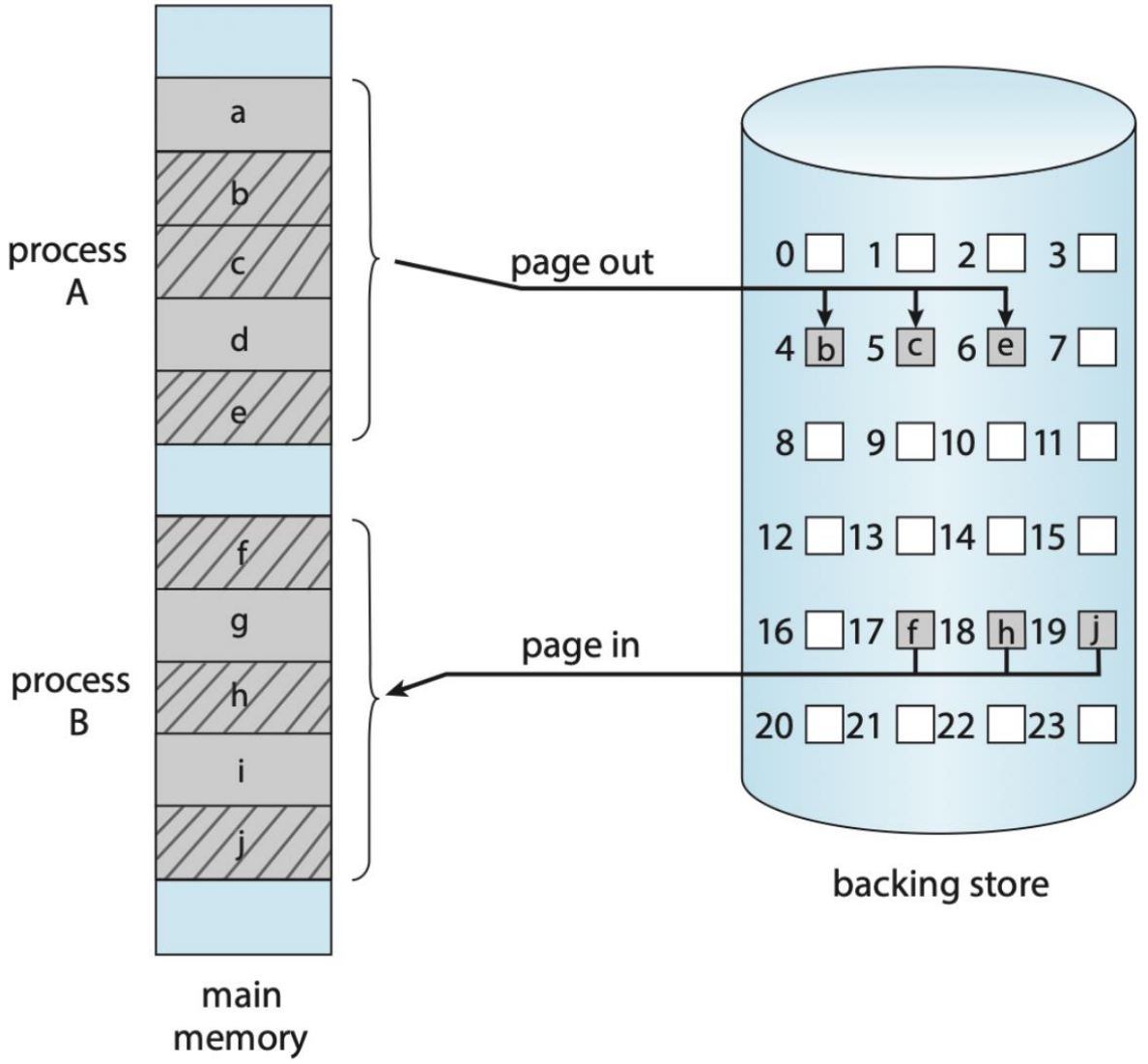
- TLB lookup time (fraction of memory time) = **e**
  - If M is the time to access memory → **e * M** is the TLB lookup time
- Hit ratio of TLB = **a** (How often a page number is found in the TLB).

- **EAT** (to access memory) = (time if hit in TLB) + (time is miss in TLB)
  = a(eM + M) + (1 – a)(eM + M + M)
  = M * (a(e+1)+ (1 – a)(e + 2))
  = **M * (e – a+2)**

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

Swapping under Paging

- 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 assume they have.
- **Swapping:**
  - Allows less-used frame to be moved to storage to temporarily free up space



How to determine if the page table includes a correct page-to-frame mapping?

Valid/Invalid Bit

- Each page table entry has a valid–invalid bit.
  - **v** → in-memory (memory resident)
  - **i** → not-in-memory.
- Initially valid-invalid bit is set to **i** on all entries.
- MMU observes for a page valid-invalid bit set to **i.**
  - Triggers page fault.
  - OS brings the page back from disk.
  - Flips bit back to **v**.