Address Translation

*Throughout the course we will use overheads that were adapted from those distributed from the textbook website. Slides are from the book authors, modified and selected by Jean Mayo, Shuai Wang and C-K Shene.

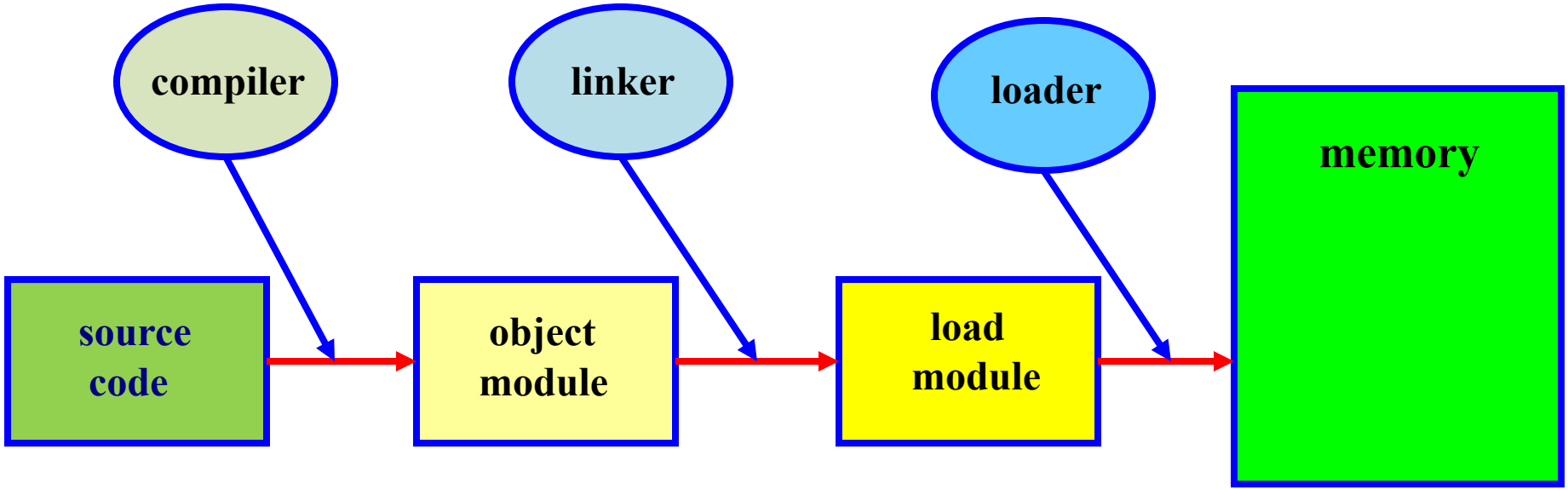
If a machine is expected to be infallible,
It cannot also be intelligent.

Alan Turing

Spring 2019

Address Generation

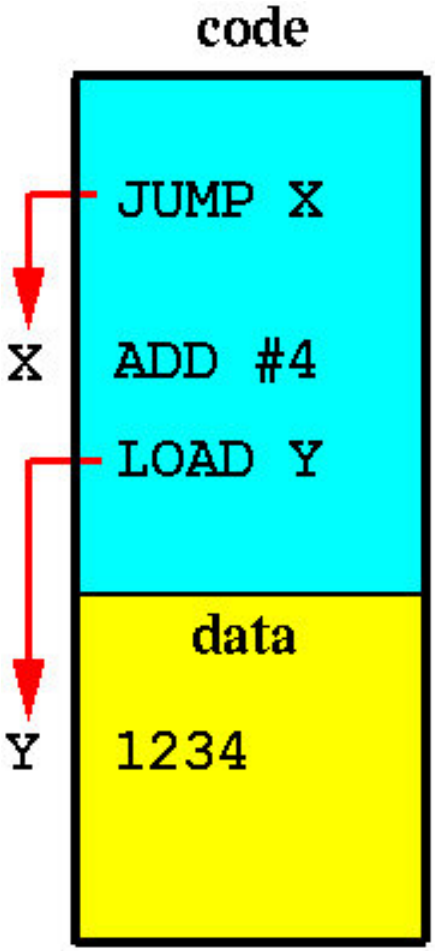
- Address generation has three stages:
  - Compile: compiler
  - Link: linker or linkage editor
  - Load: loader

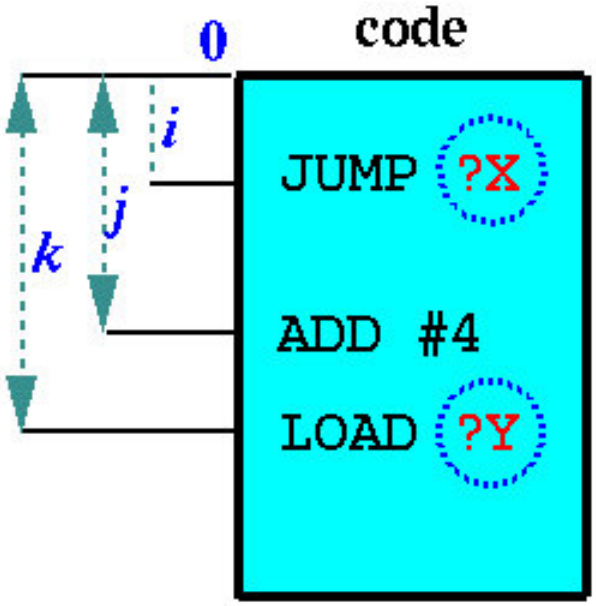


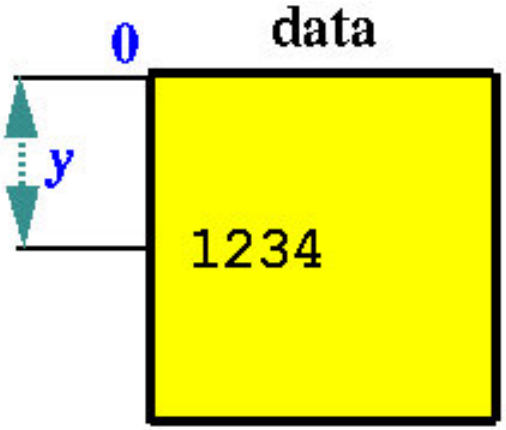
Three Address Binding Schemes

- **Compile Time: If the complier knows the location a program will reside, it can generate absolute code. Example: compile-go systems and MS-DOS .COM-format programs.**
- **Load Time: A compiler may not know the absolute address. So, the compiler generates *relocatable* code. Address binding is delayed until load time.**
- **Execution Time: If the process may be moved in memory during its execution, then address binding must be delayed until run time. This is the commonly used scheme.**

Address Generation: Compile Time



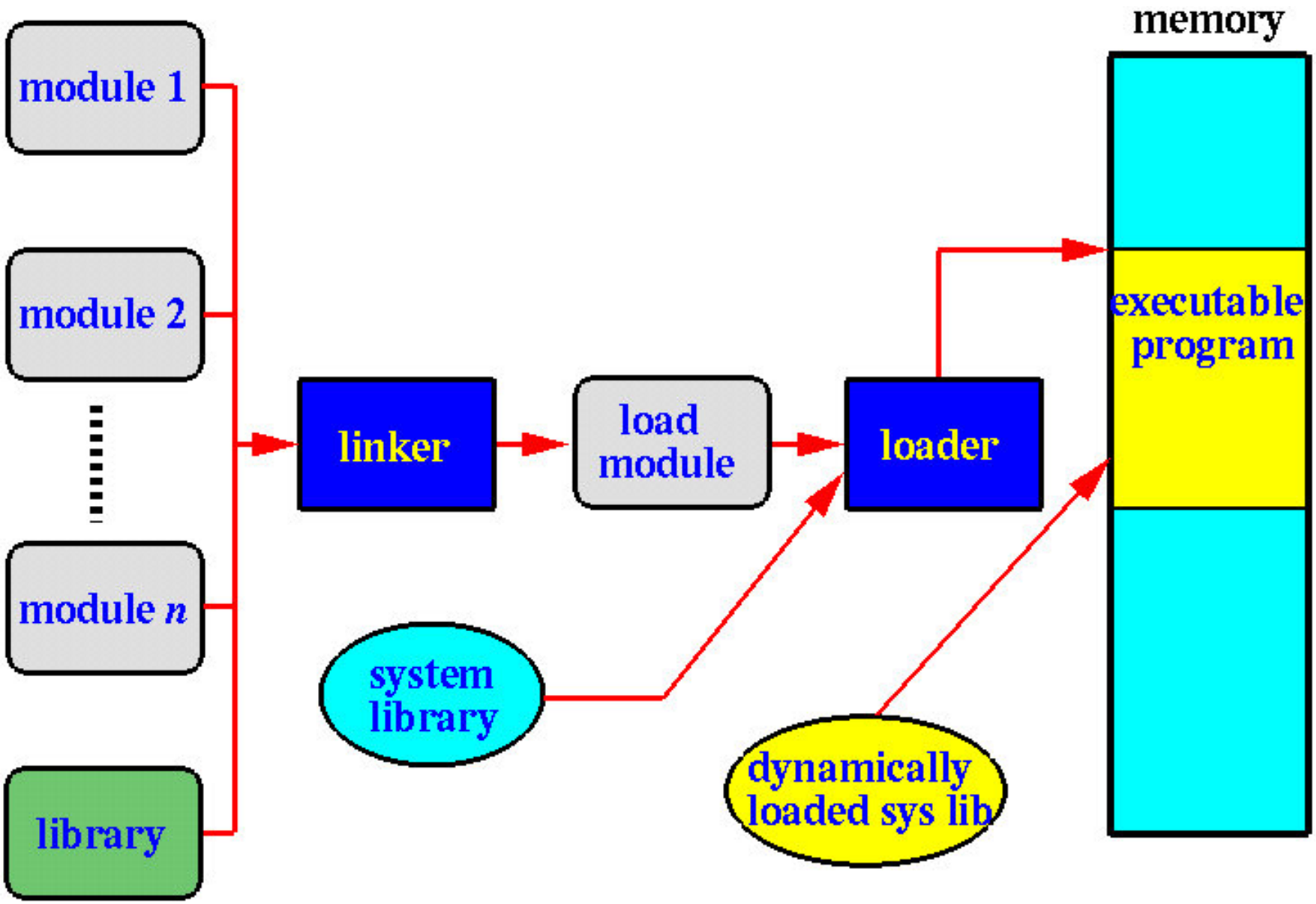




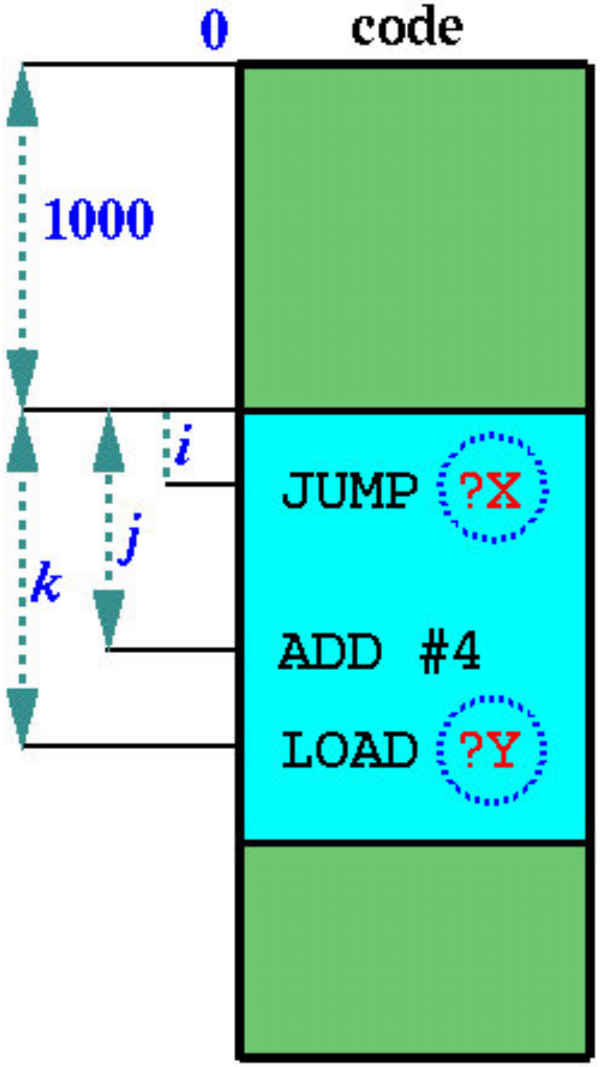
Unresolved Address Table

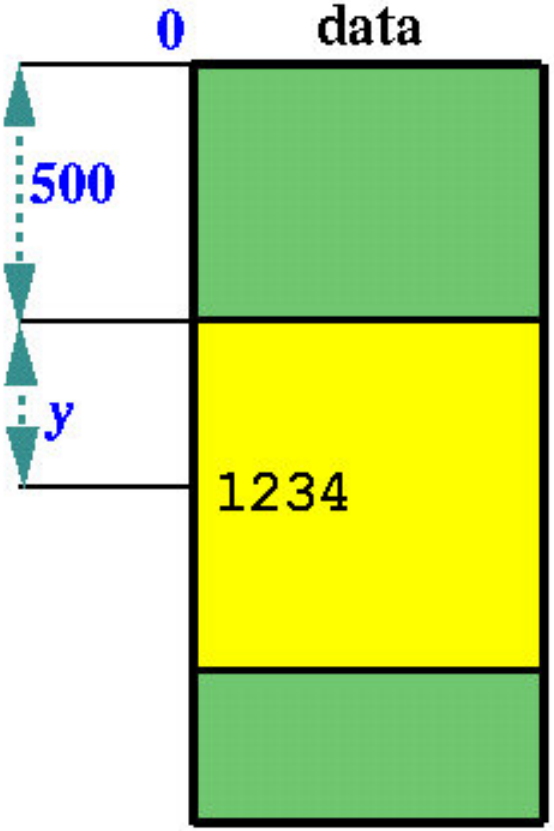
source	target	in which seg?
i	*j*	code
k	*y*	data

Linking and Loading



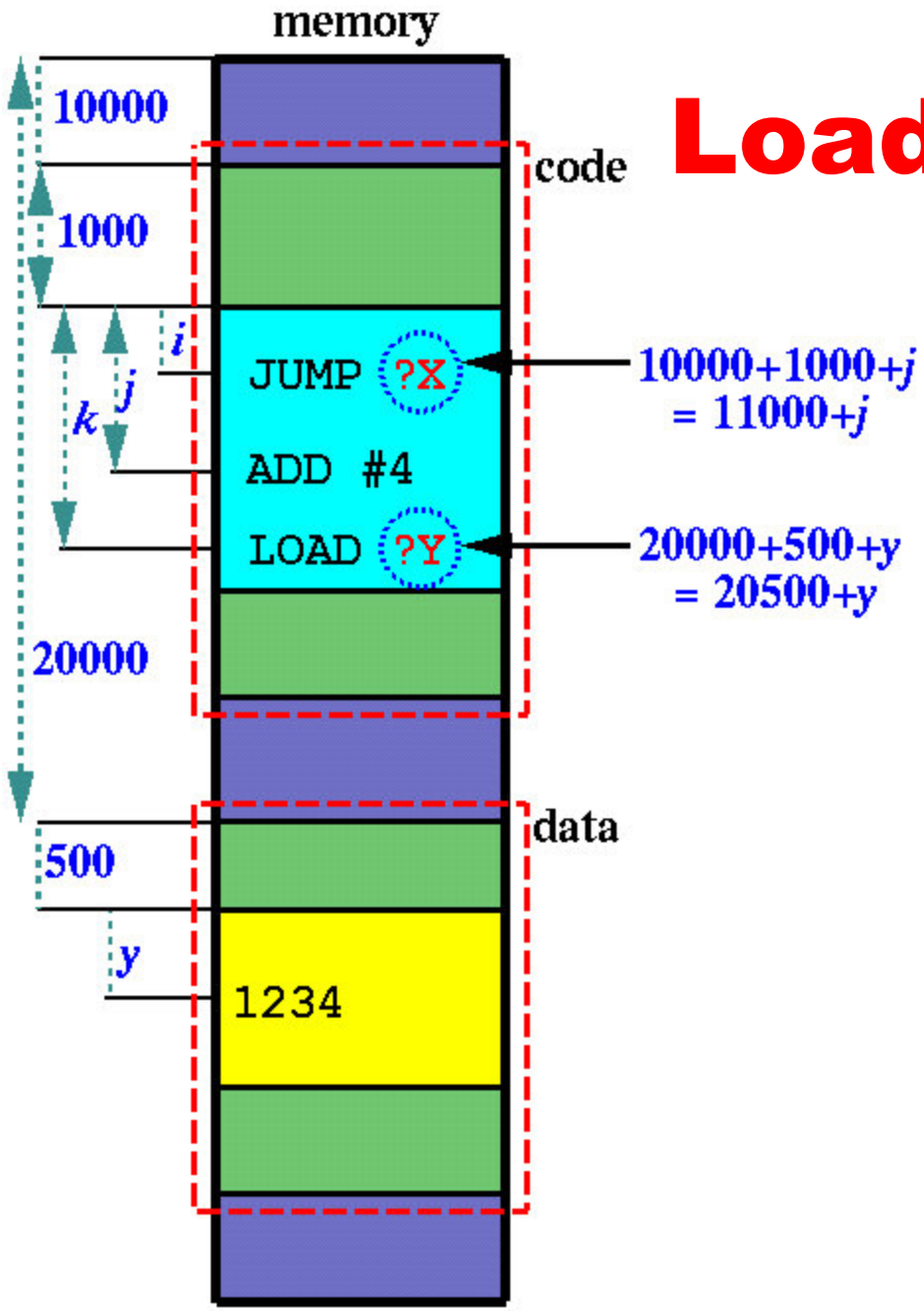
Address Generation: Static Linking





Unresolved Address Table

source	target	which seg?
i+1000	j+1000	code
k+1000	y+500	data



Loaded into Memory

- Code and data are loaded into memory at addresses 10000 and 20000, respectively.
- Every unresolved address must be adjusted.

Main Points

- Address Translation Concept
  - How do we convert a virtual address to a physical address?
- Flexible Address Translation
  - Base and bound
  - Segmentation
  - Paging
  - Multilevel translation
- Efficient Address Translation
  - Translation Lookaside Buffers (TLB)
  - Virtually and physically addressed caches

Address Translation Goals

- Memory protection
- Memory sharing
  - Shared libraries, interprocess communication
- Sparse addresses
  - Multiple regions of dynamic allocation (heaps/stacks)
- Efficiency
  - Memory placement
  - Runtime lookup
  - Compact translation tables
- Portability

Goals: 1/4

- Memory Protection
- Memory Sharing
- Flexible Memory Placement
- Sparse Addresses
- Runtime Lookup Efficiency
- Compact Translation Tables
- Portability

Goals: 2/4

- **Memory Protection**
  - We need the ability to limit the access of a process to certain regions of memory
- **Memory Sharing**
  - We want to allow multiple processes to shared selected regions of memory (e.g., shared memory segments)
- **Flexible Memory Placement**
  - We want to allow the operating system the flexibility to place a process (and each part of a process) anywhere in physical memory.

Goals: 3/4

- **Sparse Addresses**
  - Many programs have multiple dynamic memory regions that can change (e.g., heap, stack, etc.). Modern processors have 64-bit address spaces, but making the address translation more complex.
- **Runtime Lookup Efficiency**
  - Hardware address translation occurs on every instruction fetch and every data load and save. Thus, translation has to be efficient and faster than the instructions.

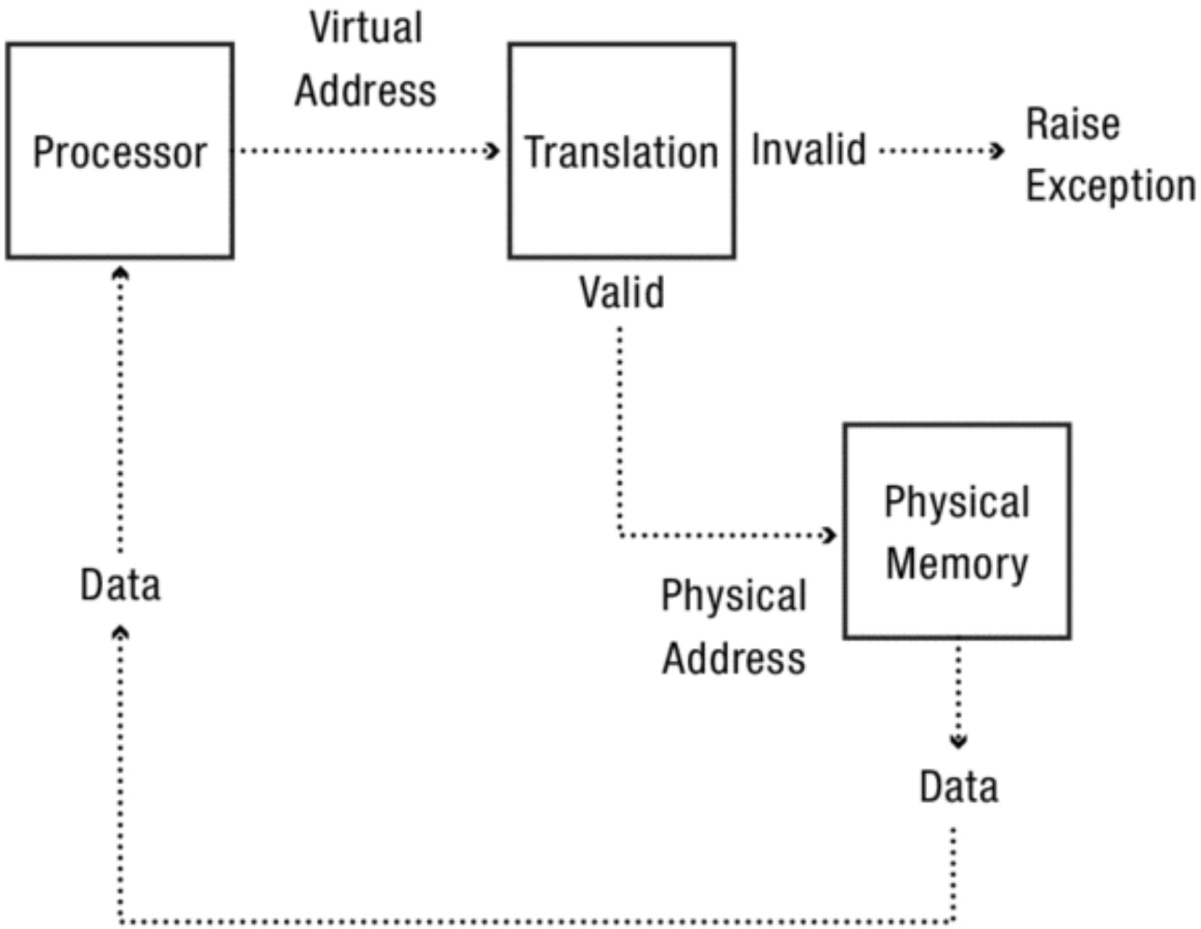
Goals: 4/4

- **Compact Translation Tables**
  - We need some tables to aid address translation. These table data structures have to be compact enough to save memory.
- **Portability**
  - Different hardware implementation use different choices to implement address translation. If an operating system is to be easily portable, it needs to be able hardware independent.

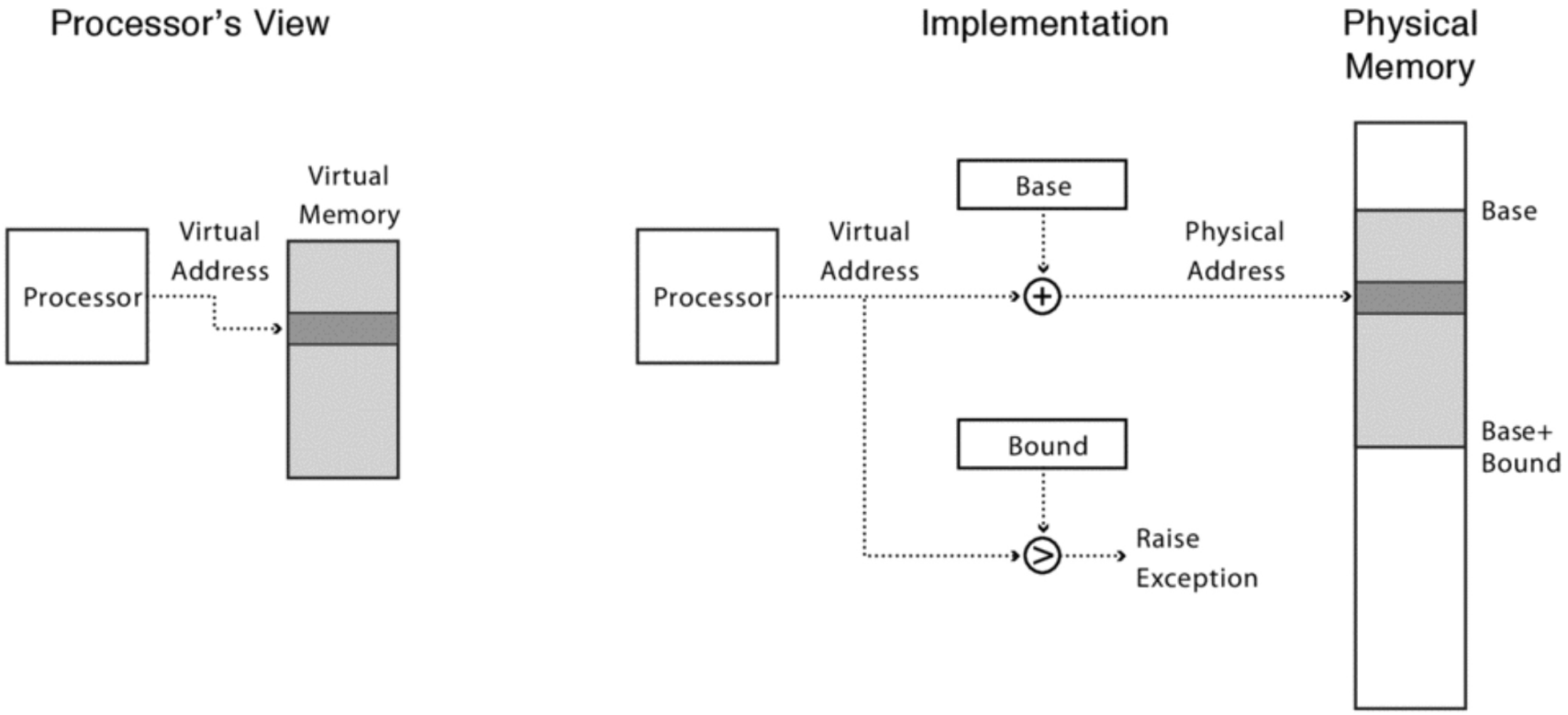
Logical, Virtual, Physical Address

- **Logical Address: the address generated by the CPU.**
- **Physical Address: the address seen and used by the memory unit.**
- **Virtual Address: Run-time binding may generate different logical address and physical address. In this case, logical address is also referred to as virtual address. (Logical = Virtual in this course)**

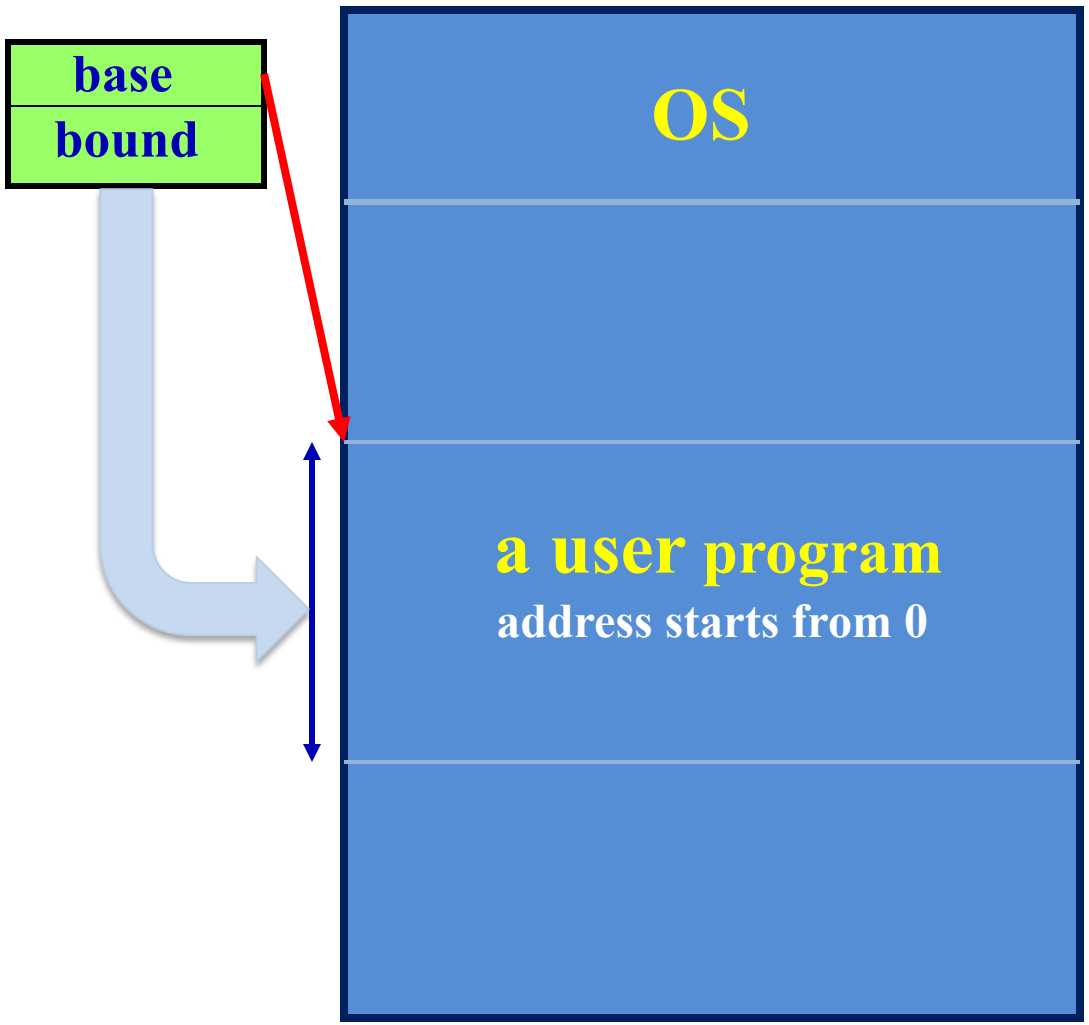
Address Translation Concept



Virtually Address Base and Bounds

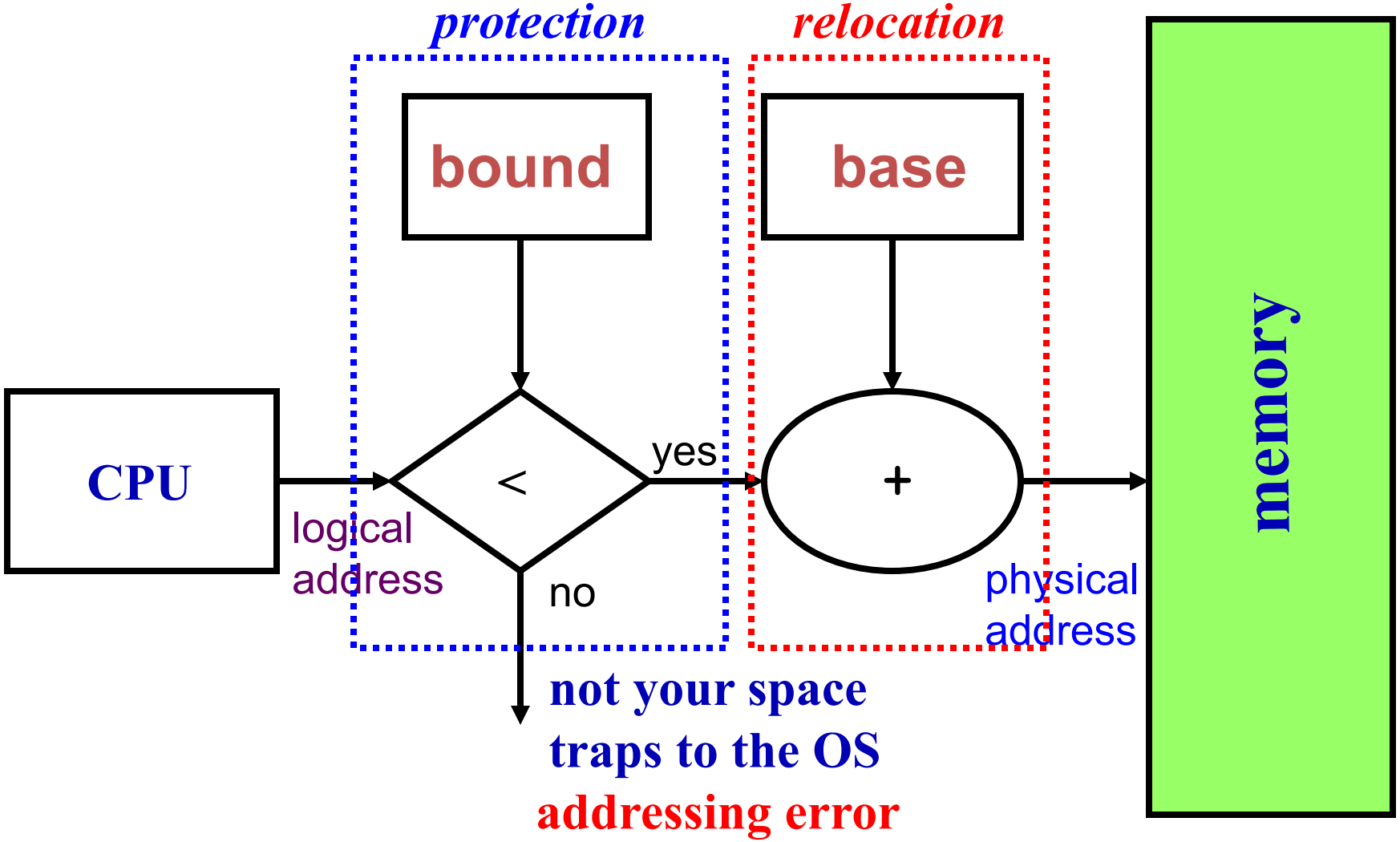


Relocation and Protection: 1/2

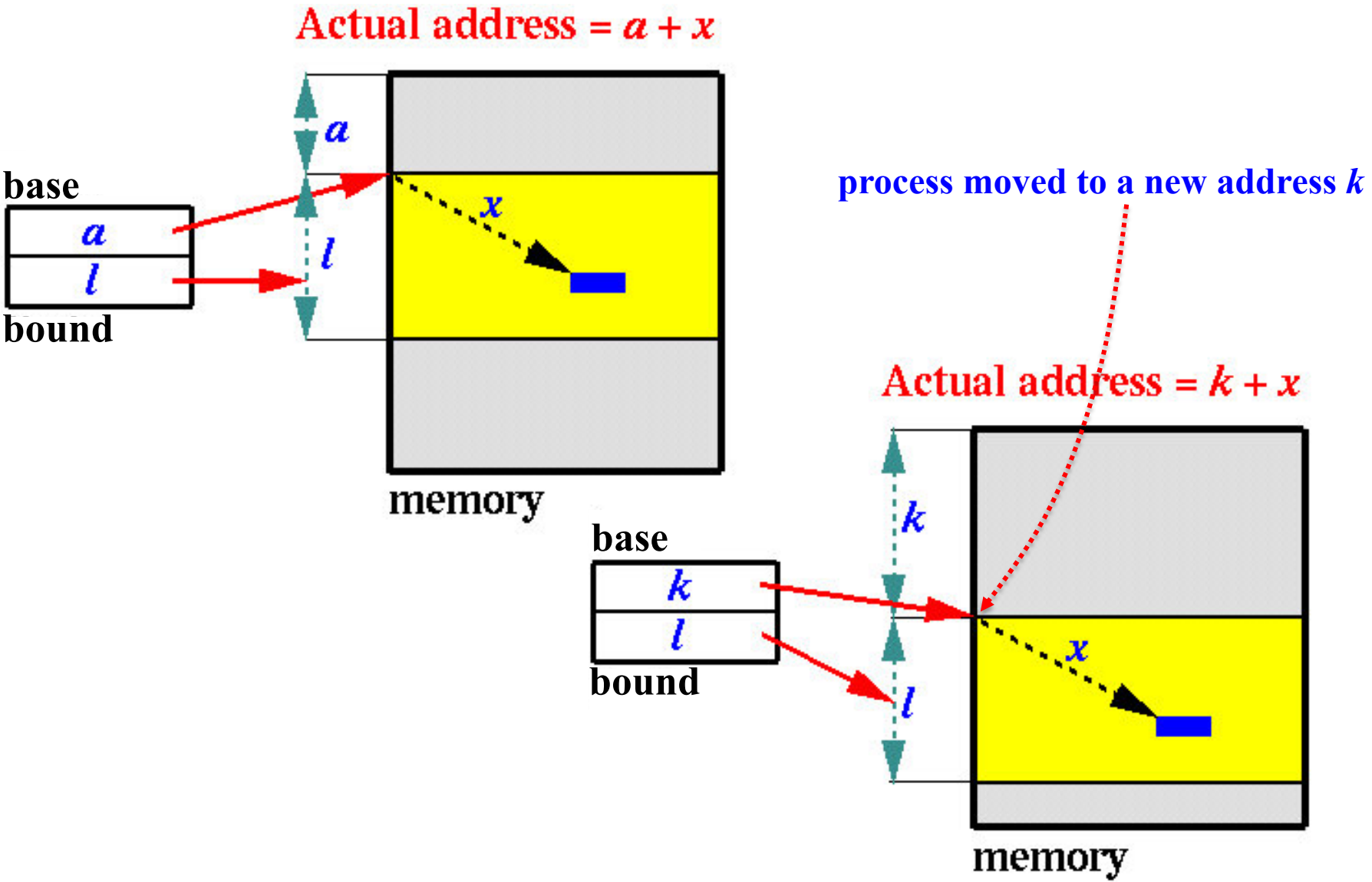


- Because executables may run in any area, relocation and protection are needed.
- Recall the base/limit register pair for memory protection.
- It could also be used for relocation if the linker generates executables starting from 0.
- Linker generates *relocatable* code starting with 0. The base register contains the starting address.

Relocation and Protection: 2/2



Relocation: How does it work?



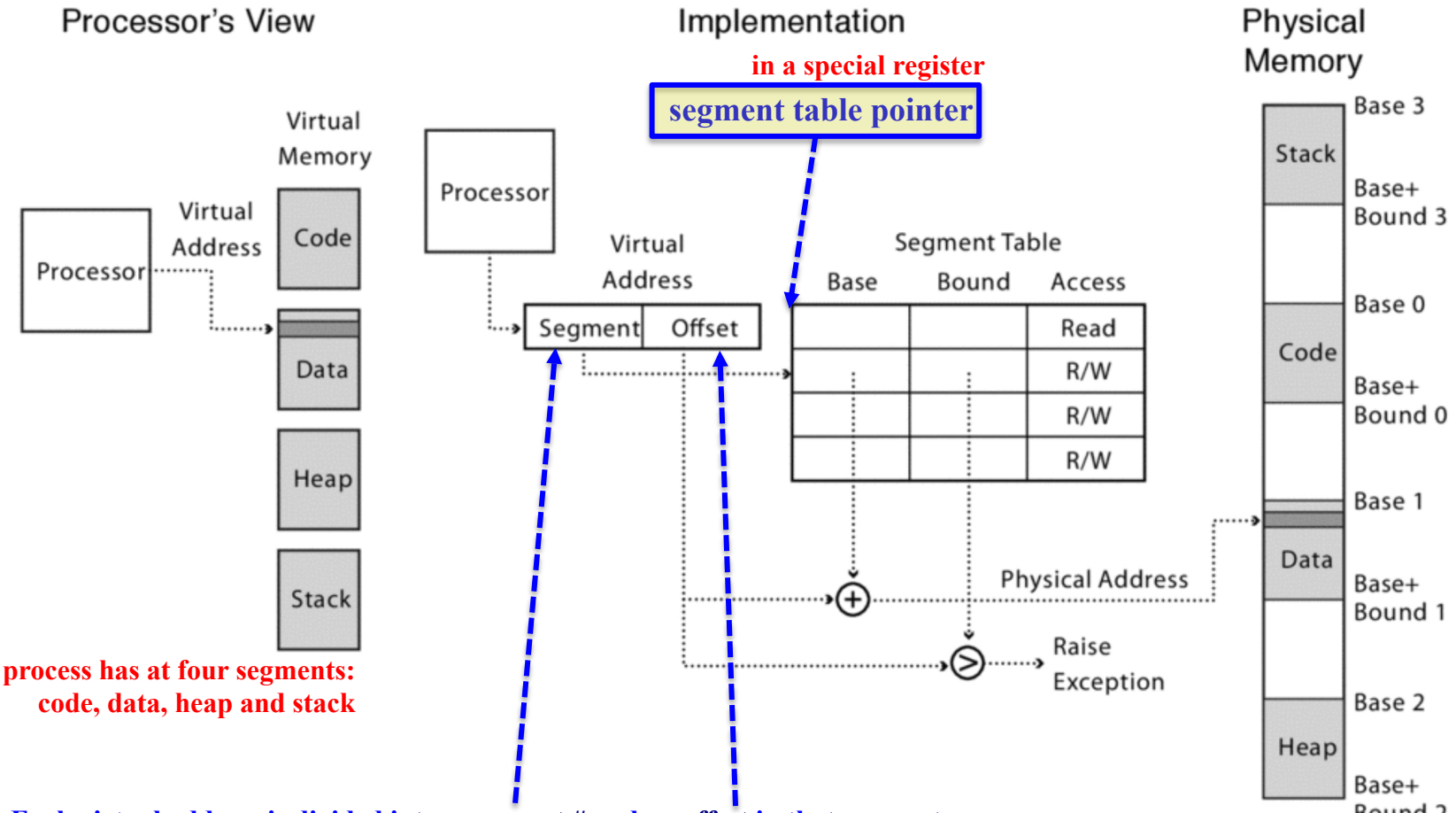
Virtually Addressed Base and Bound

- **Pros?**
  - Simple
  - Fast (2 registers, adder, comparator)
  - Safe
  - Can relocate in physical memory without changing process
- **Cons?**
  - Can't keep program from accidentally overwriting its own code
  - Can't share code/data with other processes
  - Can't grow stack/heap as needed

Segmentation: 1/7

- Segment is a contiguous region of *virtual* address space
- Each process has a segment table, and each entry in the table has a pointer to a segment
- Segment can be located anywhere in physical memory, and each segment has: start (i.e., base), length (i.e., bound), access permission, etc.
- Therefore, segmentation can be considered as having multiple base/bound registers stored in a table.
- Processes can share segments.

Segmentation: 2/7



Each process has at four segments: code, data, heap and stack

Each virtual address is divided into a segment # and an offset in that segment. Suppose we have 31-bit address, and this address is divided into 16-bit for offset and 15-bit for segment number. In this way, 16-bit offset means segment max. size is 2^{16} = 64K bytes and 2^{15} segments.

Segmentation: 3/7

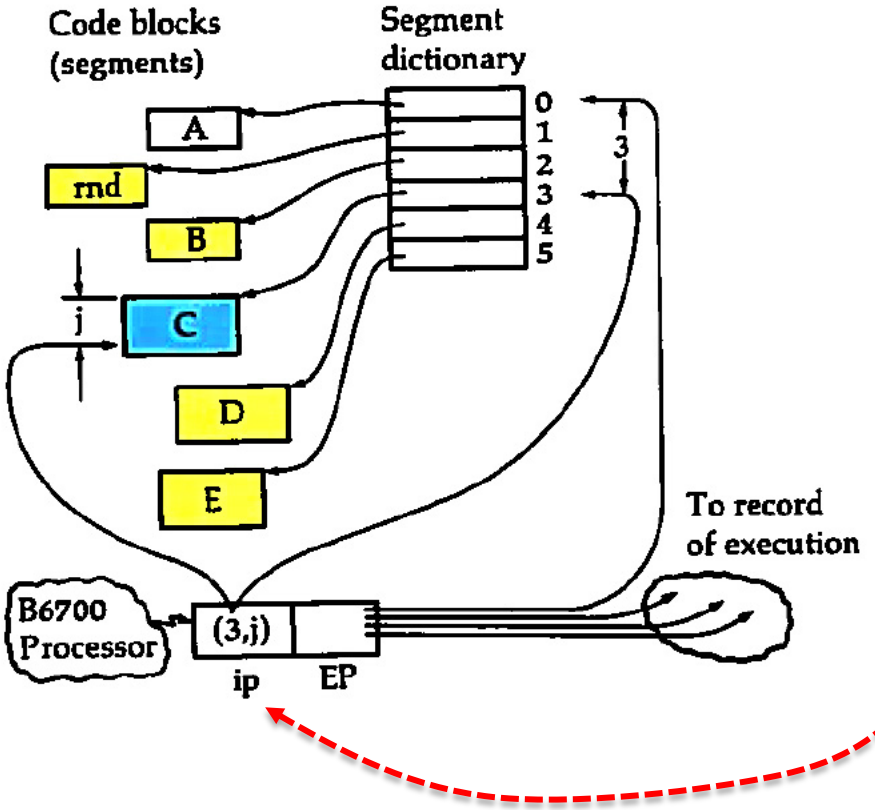
- A process is divided into segments. The chunks that a program is divided into which are not necessarily of the same length.
- Early systems (MULTICS and Burroughs B5700/B6700) used segmentation memory management.
- Burroughs Corporation was founded in 1886, in 1986 merged with Sperry UNIVAC and renamed Unisys. In the 1970'sm Burroughs developed some large systems based on the block-based (i.e., ALGOL) languages.

Segmentation: 4/7

- Burroughs B5700/B6700 are interesting as their processors are designed around the language blocked-structured (e.g., ALGOL, PL/I, etc.).
- Procedures can be declared as local procedures, which are called by the containing procedure.
- Thus, procedures do have a tree structure.
- Each procedure is in its own segment.
- A hardware pointer `ip` is the program counter.
- Another pointer `EP` points to the activation record on a stack.

Segmentation: 5/7

- There are two sets of pointers, one pointing to the executing code and the order to the its corresponding execution environment.



Each "procedure" is in its own segment, and the segment directory table has pointers to each segment.

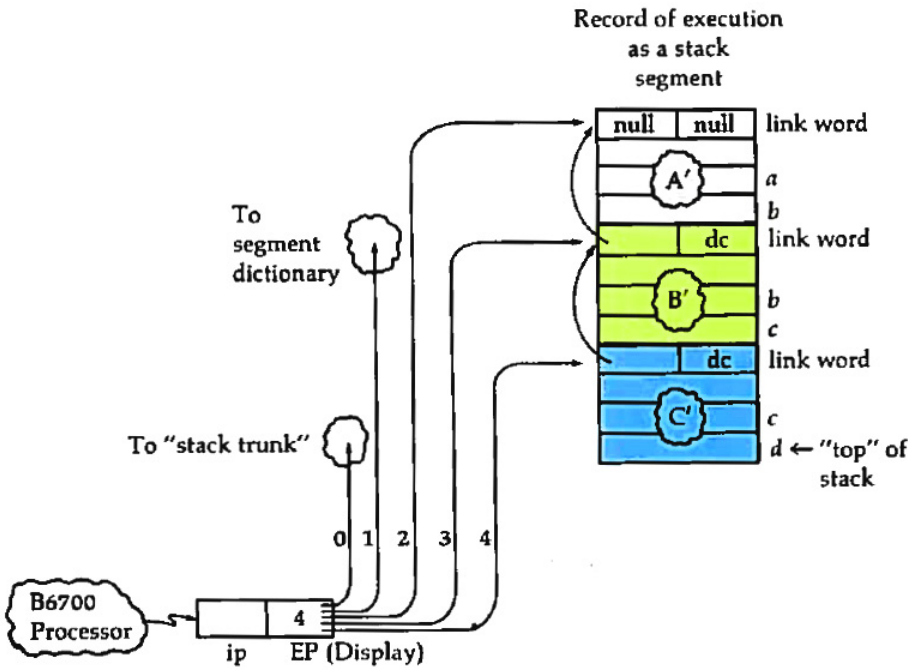
The left diagram shows 6 procedures and the `ip` pointer indicates that the processor is executing procedure `C`!

The `ip` (3,j) indicates the next instruction is in segment `3` and offset `j`.

Diagram is taken from Elliott I. Organick, *Computer System Organization*: *The B5700/B6700*, ACM Press, 1973.

Segmentation: 6/7

- There are two sets of pointers, one pointing to the executing code and the order to the its corresponding execution environment.



Each "procedure" has its own "environment" (i.e., segment), and the segment Display points to the Location of its activation record.

This diagram shows the corresponding activation record on the stack.

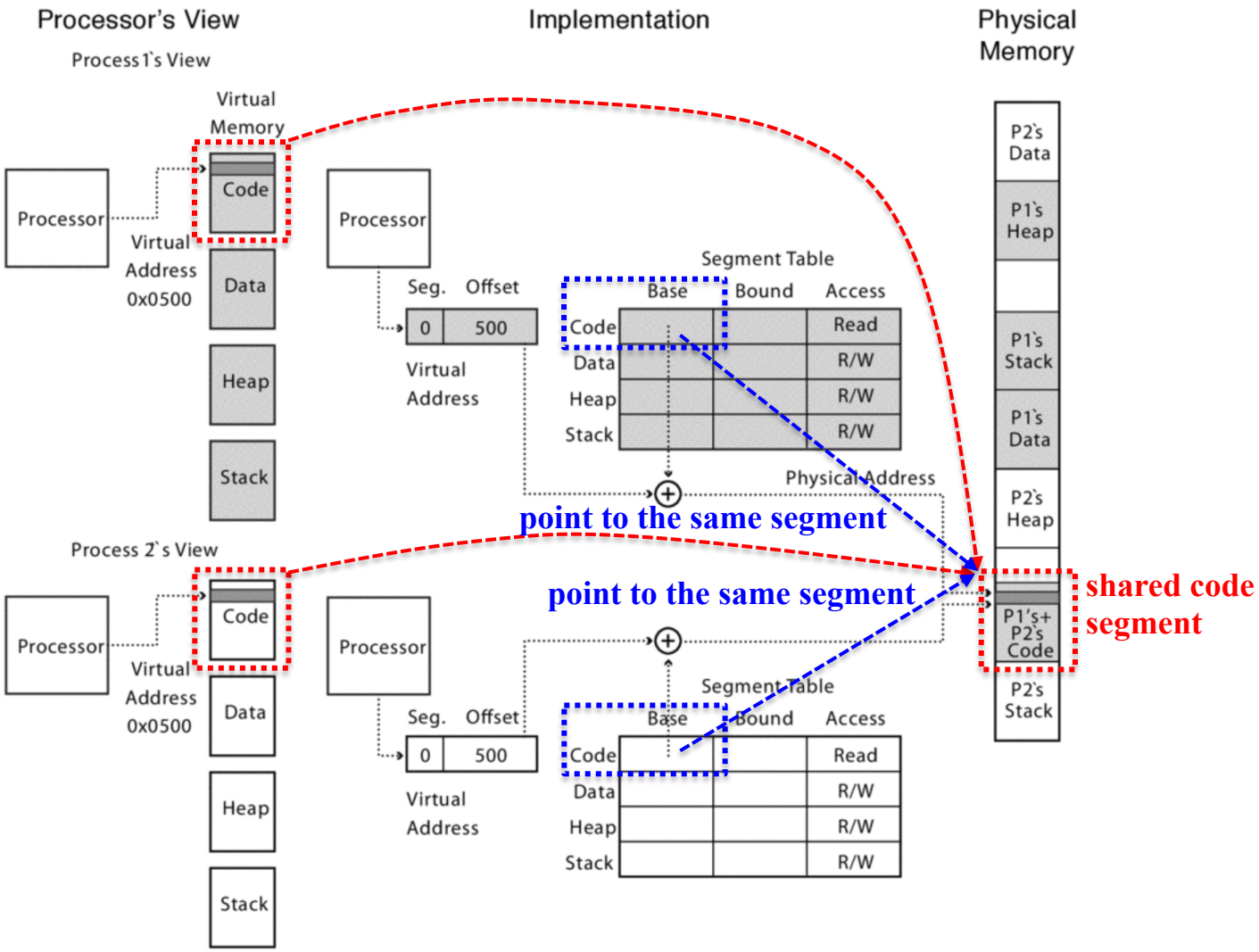
The left diagram shows 5 procedures and the `EP` pointers indicate the processor is executing procedure `C`!

Diagram is taken from Elliott I. Organick, *Computer System Organization: The B5700/B6700*, ACM Press, 1973.

Segmentation: 7/7

- **Pros?**
  - Can share code/data segments between processes
  - Can protect code segment from being overwritten
  - Can transparently grow stack/heap as needed
  - Can detect if need to copy-on-write
- **Cons?**
  - Complex memory management
    - ✓Need to find chunk of a particular size
  - May need to rearrange memory from time to time to make room for new segment or growing segment
    - ✓External fragmentation: wasted space between chunks

Segmentation Sharing



Segmentation Fault

- Each entry in the segment table controls a portion of the virtual address space.
- When a request for creating a new segment comes but the system does not have enough space, a **segment fault** is generated. Note that the system may still have enough memory in total; but, each piece of free space is smaller than the requested one (e.g., **fragmented**).
- Some systems (e.g., Unix) may also generate a segment fault if a process accesses an address not in any existing segment.

UNIX `fork` and Copy on Write

- UNIX `fork`
  - Makes a complete copy of a process
- Segments allow a more efficient implementation
  - Copy segment table into child
  - Mark parent and child segments read-only
  - Start child process; return to parent
  - If child or parent writes to a segment (ex: stack, heap)
    - ✓ trap into kernel
    - ✓ make a copy of the segment and resume

Zero-on-Reference

- How much physical memory is needed for the stack or heap?
  - **ANS**: only what is currently in use
- When program uses memory beyond end of stack
  - Segmentation fault into OS kernel
  - Kernel allocates some memory
    - ✓How much?
  - Zeros the memory
    - ✓avoid accidentally leaking information!
  - Modify segment table
  - Resume process

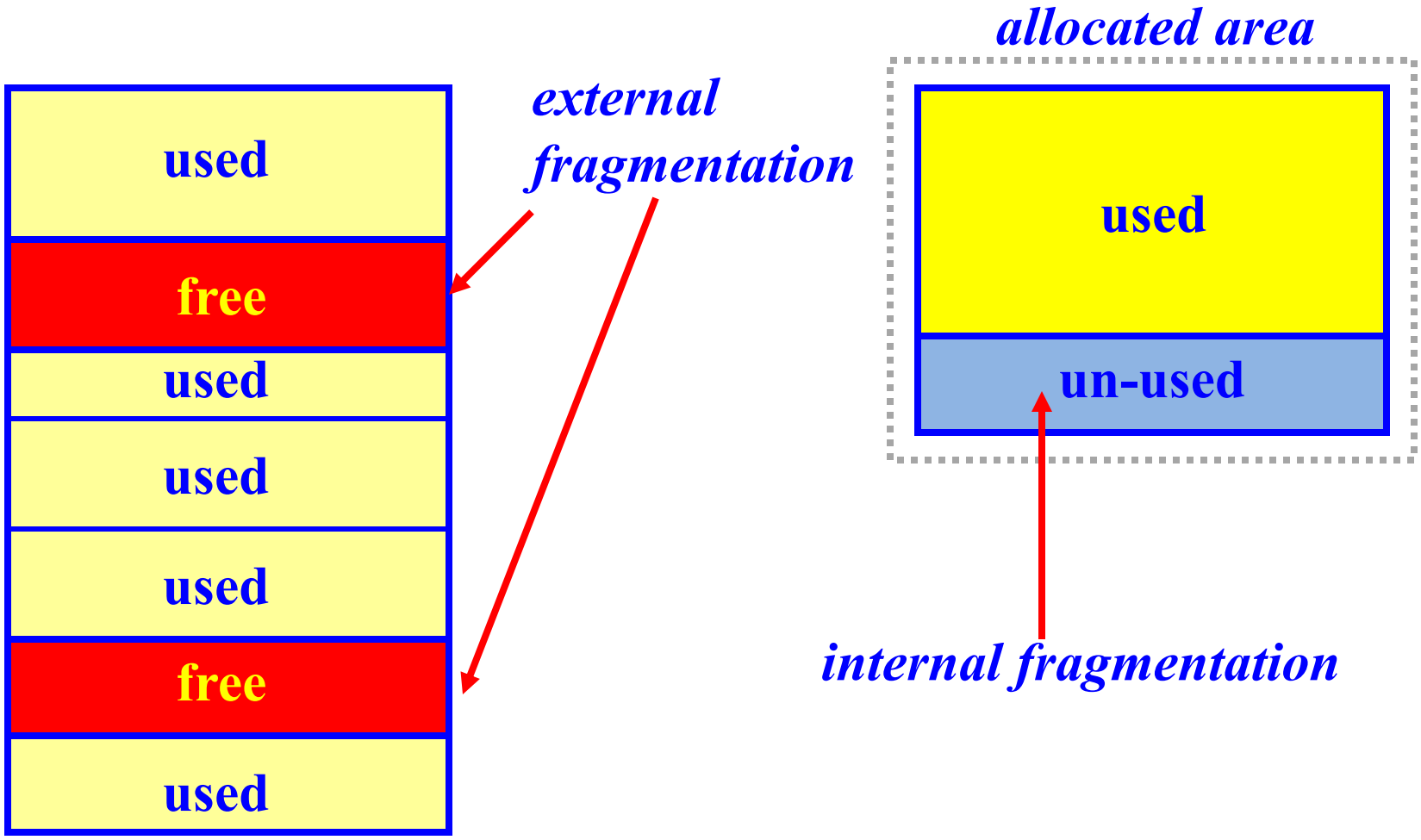
Segmentation Downside

- The number of segments can be large, and their size vary significantly.
- What if a request for a 14K segment arrives?
- This is impossible even though the system does have 37k = 12+10+8+7.
- The free space is not contiguous even though the total is good enough for this allocation..
- These free slots not in any segment are **external fragmentation**.
- We could move the allocated space around to get a large enough space for the request provided that each allocated space is **relocatable** (e.g., segment).

Fragmentation

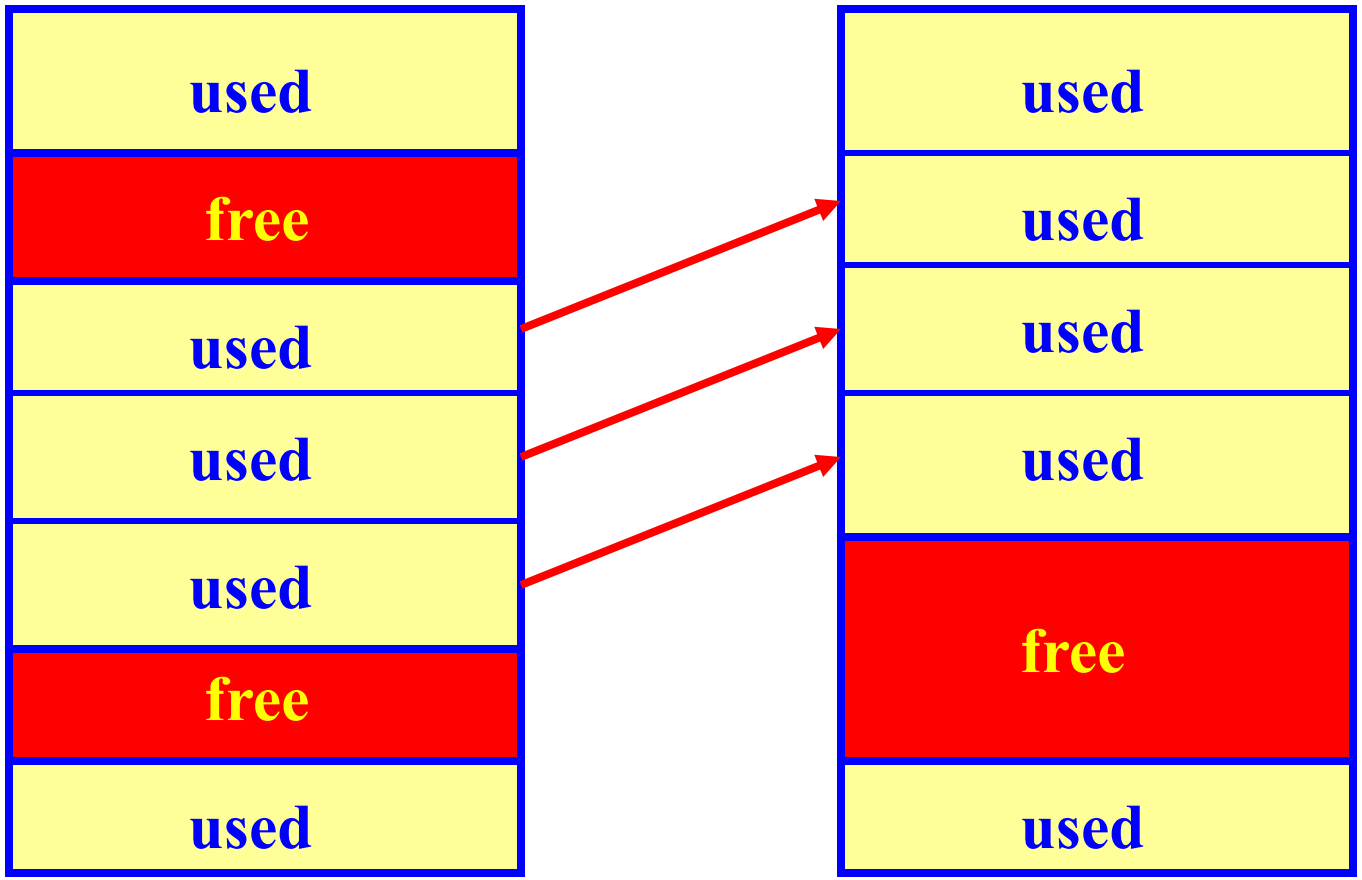
- Processes are loaded and removed from memory, eventually memory is cut into small holes that are not large enough to run any incoming process.
- Free memory holes between allocated ones are called **external fragmentation**.
- It is unwise to allocate exactly the requested amount of memory to a process, because of address boundary alignment requirements or the minimum requirement for memory management.
- Thus, memory that is allocated to a partition, but is not used, is an **internal fragmentation**.

External/Internal Fragmentation



Compaction for External Fragmentation

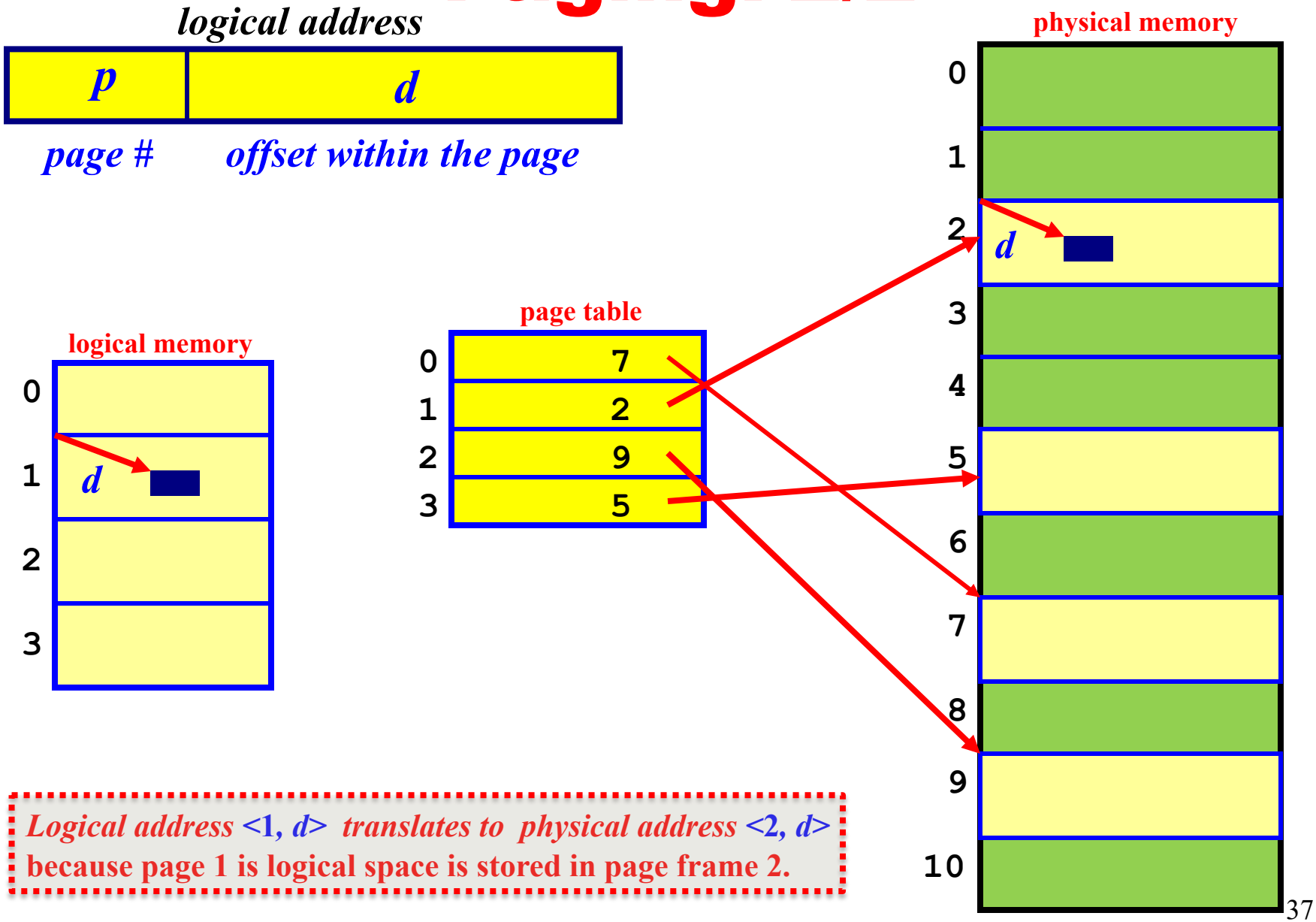
- If processes are relocatable, we may move used memory blocks together to make a larger free memory block.



Paging: 1/2

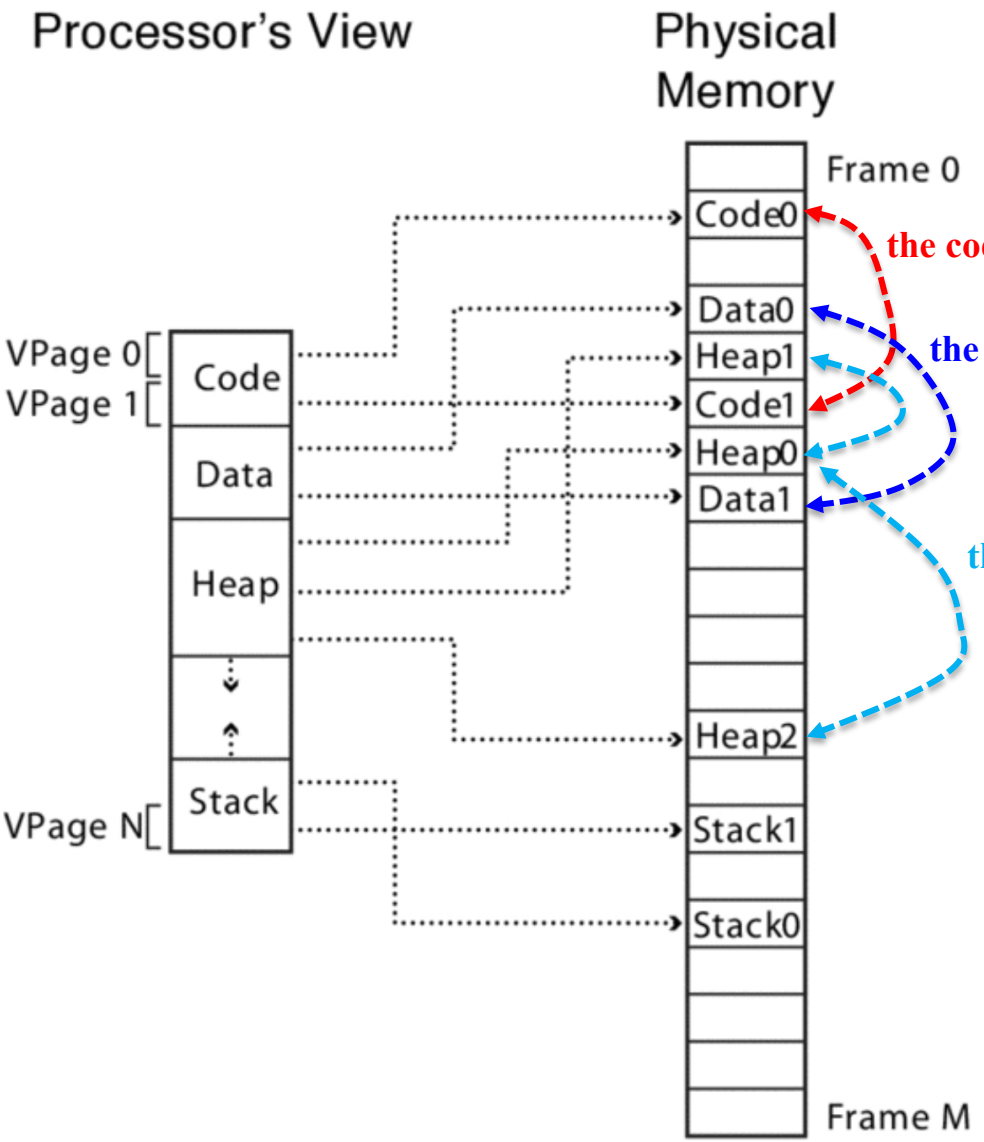
- The physical memory is divided into fixed-sized *page frames*, or *frames*.
- The virtual address space is also divided into blocks of the same size, called *pages*.
- When a process runs, its pages are loaded into page frames.
- A page table stores the *page numbers* and their corresponding *page frame numbers*, etc.
- The virtual address is divided into two fields: *page number* and *offset* (in that page).

Paging: 2/2



Logical address <1, *d*> *translates to physical address* <2, *d*>
because page 1 is logical space is stored in page frame 2.

Paged Translation (Abstract)



the code section has 2 pages (& page frames)

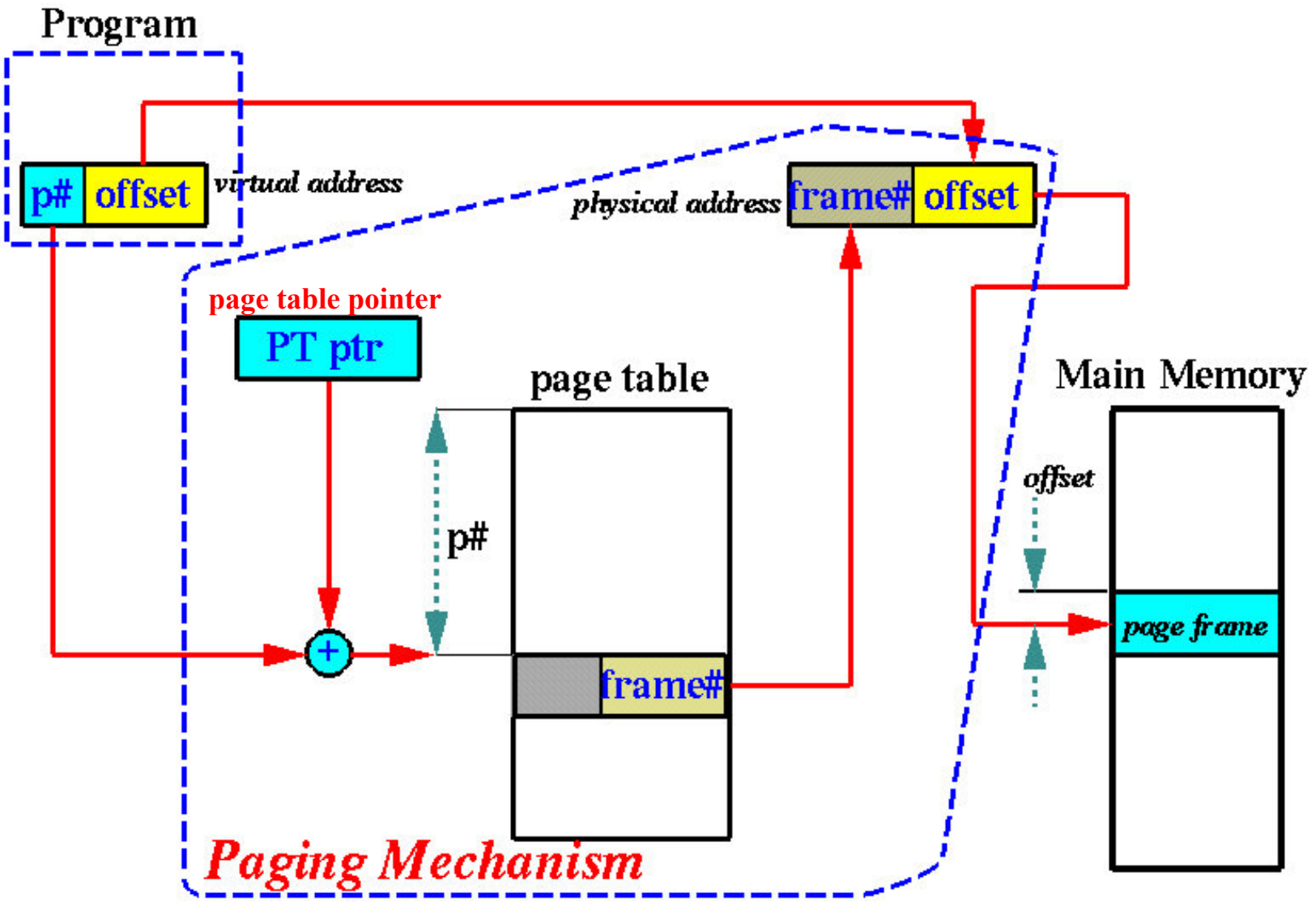
the data section has 2 pages (& page frames)

the heap section has 3 pages (& page frames)

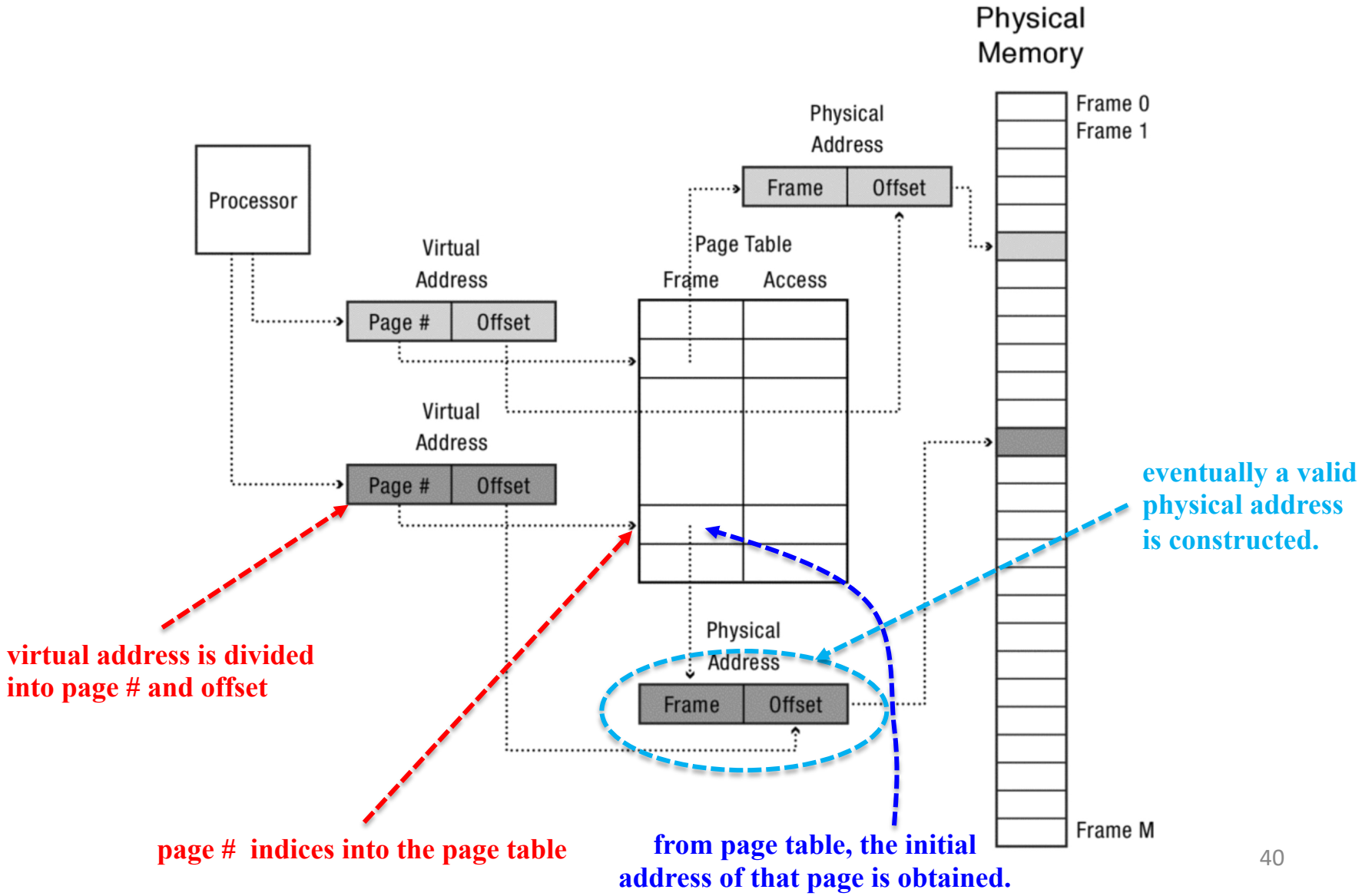
Physical space is divided into page frames of equal size usually 2k, 4k, 8k.

Virtual space is also divided into [ages of the same size so that each page can be fit into a page frame.

Address Translation



Paged Translation (Implement)



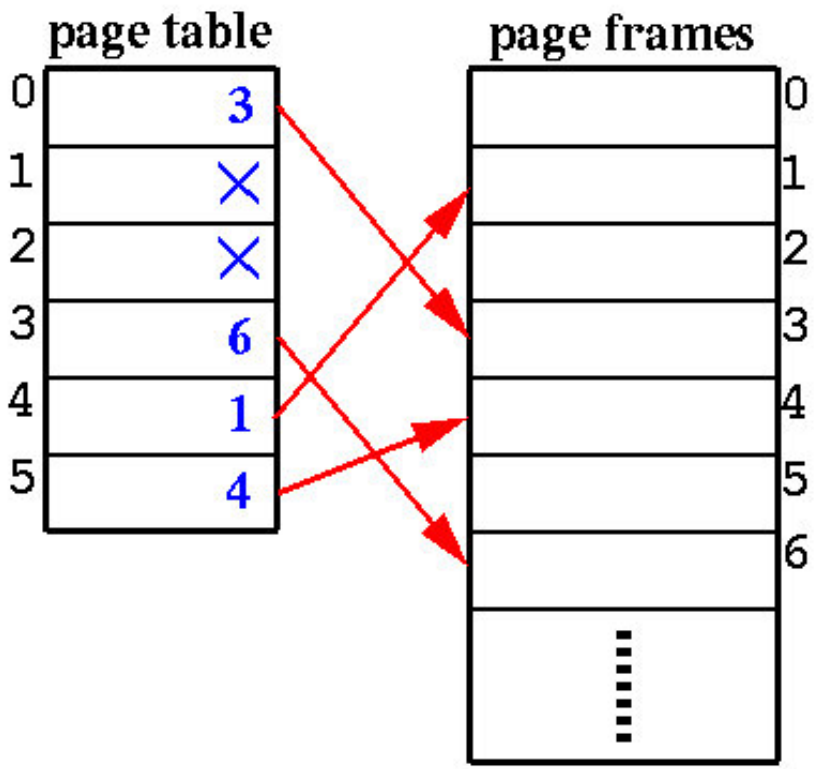
Translation Example: 1/3

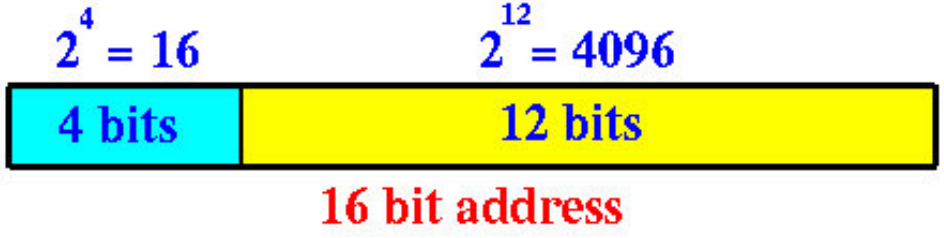
- Suppose we have a logical/virtual address *x*, and suppose the page size is *y*.
- Then, *x*/*y* is the group (of size *y*) *x* is in, and *x*%*y* is the offset within that (*x*/*y*) group.
- **Example**: Suppose *y* = 16 and *x* = 76. Then, *x*/*y* = 76/16 = 4. Therefore, if 76 elements are grouped into 16 each, then 76 is in the 4th group. Because 76%16 = 12, 76 is the 12th element in the 4th group.

Translation Example: 2/3

- Suppose page size uses k bits. The address of a page is from 0 to 2^k-1. If page size is 4K = 2^{12}, then the address in a page is from 0 to $2^{12}-1=4095$.
- Dividing by 2 can be achieved by shifting to the right. For example, for $13_{10} = 1101_2$, dividing 13 by 2 is obtained by shifting 1101_2 to the right 1 bit yielding $110_2 = 6_{10}$. Thus, dividing a number x by 2^k can be obtained by shifting x to the right k bits.
- **Example**: Suppose an address $110110\mathbf{01101110}_2$ is divided by 2^8. The result is 110110_2, which is the page number. The remainder $\mathbf{01101110}_2$ is the offset in this page.
- **Conclusion**: Given a logical/virtual address of n bits and page size 2^k, the page number of this address is the first $n-k$ bits and the offset in this page is the last k bits.

Translation Example: 3/3





15000 (virtual address):

15000/4096:
quotient = 3 (page #)
remainder = 2712 (offset)

From page table,
page #3 is in frame #6

Real address
= (frame#)*4096+offset
= 6*4096 + 2712 = 27288

10000 (virtual address):

10000/4096:
quotient = 2 (page #)
remainder = 1808 (offset)

From page table:
page 2 not in memory
a *page fault* occurs

Hardware Support

- Page table may be stored in special registers if the number of pages is small.
- Page table may also be stored in physical memory, and a special register, page-table base register, points to the page table.
- Use translation look-aside buffer (TLB). TLB stores recently used pairs (page #, frame #). It compares the input page # against the stored ones. If a match is found, the corresponding frame # is the output. Thus, no page table access is required.
- This comparison is done in *parallel* and is *fast*.
- TLB normally has 64 to 1,024 entries.

Paging Questions

- With paging, what is saved/restored on a process context switch?
  - pointer to page table, size of page table
  - page table itself is in main memory
- What if page size is very small?
- What if page size is very large?
  - **Internal fragmentation**: if we don't need all of the space inside a fixed size chunk

Paging and Copy on Write

- Can we share memory between processes?
  - Set entries in both page tables to point to same page frames
  - Need *core map* of page frames to track which processes are pointing to which page frames (e.g., reference count)
- UNIX `fork` with copy on write
  - Copy page table of parent into child process
  - Mark all pages (in new and old page tables) as read-only
  - Trap into kernel on write (in child or parent)
  - Copy page
  - Mark both as writeable
  - Resume execution

Fill On Demand

- Can I start running a program before its code is in physical memory?
  - Set all page table entries to invalid
  - When a page is referenced for first time, kernel trap
  - Kernel brings page in from disk
  - Resume execution
  - Remaining pages can be transferred in the background while program is running

Sparse Address Spaces

- Might want many separate dynamic segments
  - Per-processor heaps
  - Per-thread stacks
  - Memory-mapped files
  - Dynamically linked libraries
- What if virtual address space is large?
  - 32-bits, 4KB pages => 500K page table entries
  - 64-bits => 4 quadrillion page table entries

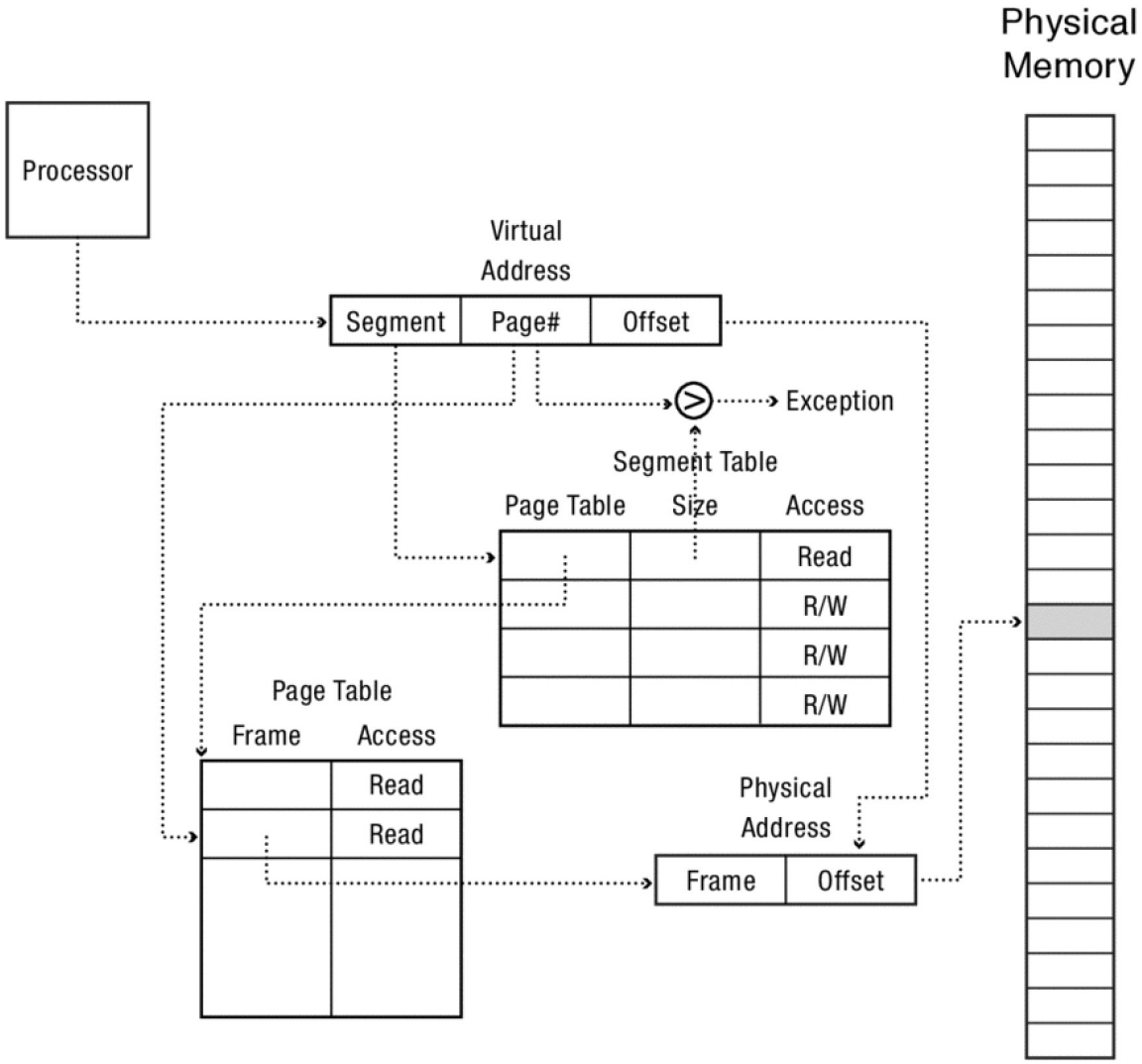
Multi-level Translation

- Tree of translation tables
  - Paged segmentation
  - Multi-level page tables
  - Multi-level paged segmentation
- Fixed-size page as lowest level unit of allocation
  - Efficient memory allocation (compared to segments)
  - Efficient for sparse addresses (compared to paging)
  - Efficient disk transfers (fixed size units)
  - Easier to build translation lookaside buffers
  - Efficient reverse lookup (from physical -> virtual)
  - Variable granularity for protection/sharing

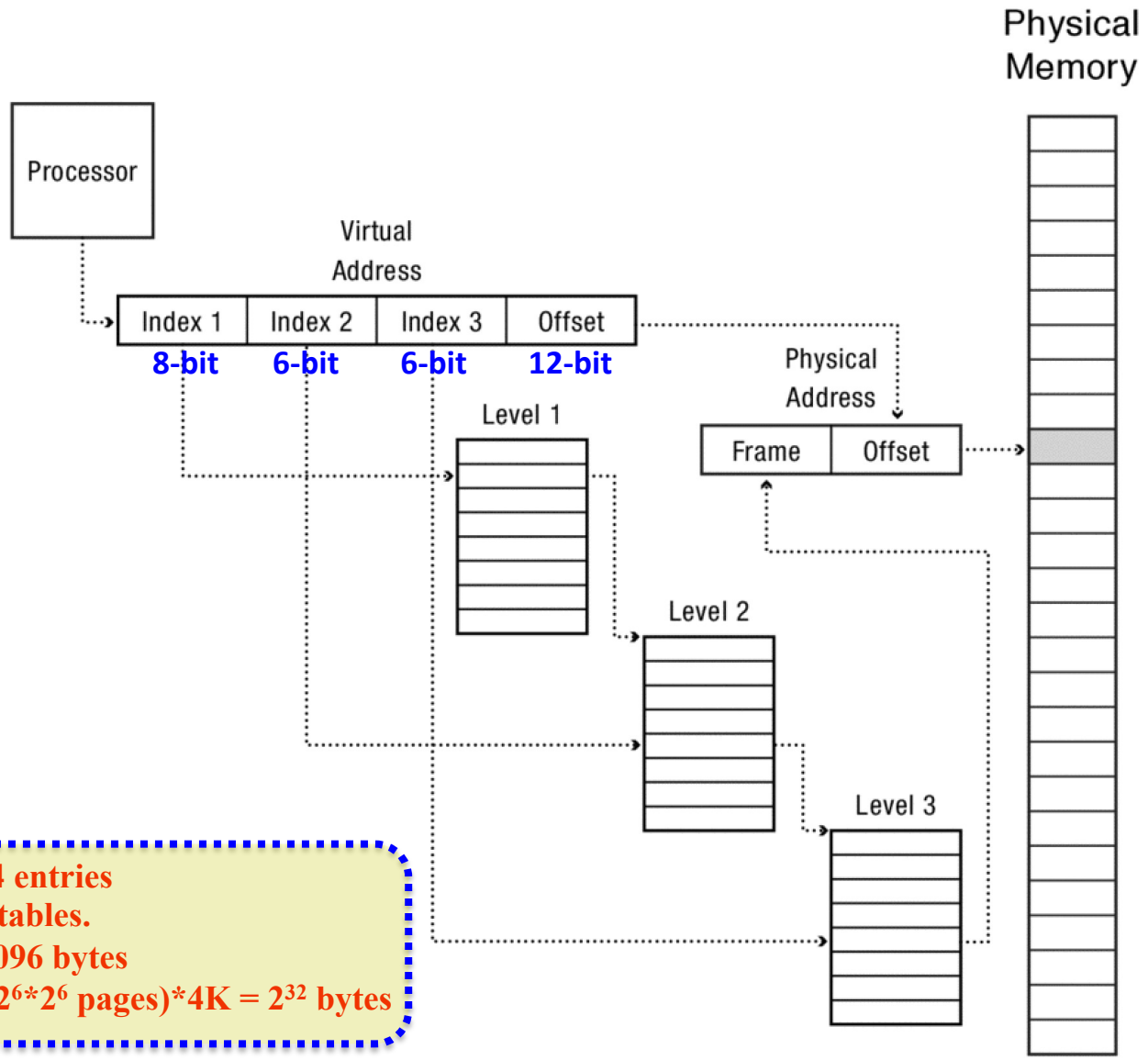
Paged Segmentation

- Process memory is segmented
- Segment table entry:
  - Pointer to page table
  - Page table length (# of pages in segment)
  - Access permissions
- Page table entry:
  - Page frame
  - Access permissions
- Share/protection at either page or segment-level

Paged Segmentation (Implementation)



Multilevel Paging



- There are 256, 64 and 64 entries in level 1, 2, and 3 page tables.
- Page size is 4k = 2^{12} = 4,096 bytes
- Virtual space size = (2^8*2^6*2^6 pages)*4K = 2^{32} bytes

x86 Multilevel Paged Segmentation

- **Global Descriptor Table (segment table)**
  - Pointer to page table for each segment
  - Segment length
  - Segment access permissions
  - Context switch: change global descriptor table register (GDTR, pointer to global descriptor table)
- **Multilevel page table**
  - 4KB pages; each level of page table fits in one page
  - 32-bit: two level page table (per segment)
  - 64-bit: four level page table (per segment)
  - Omit sub-tree if no valid addresses

Multilevel Translation

- **Pros:**
  - Allocate/fill only page table entries that are in use
  - Simple memory allocation
  - Share at segment or page level
- **Cons:**
  - Space overhead: one pointer per virtual page
  - Two (or more) lookups per memory reference

Portability

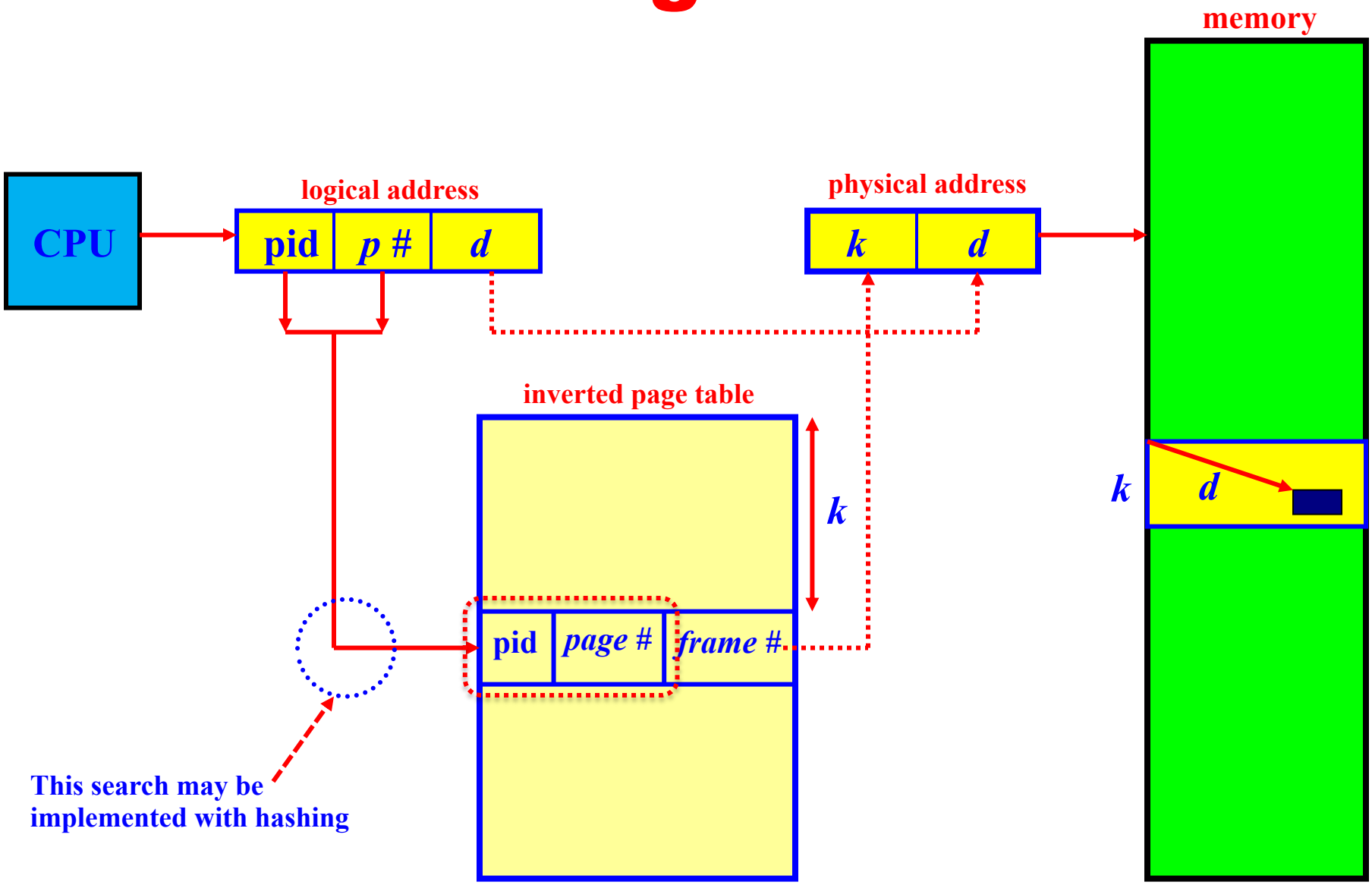
- Many operating systems keep their own memory translation data structures
  - List of memory objects (segments)
  - Virtual page -> physical page frame
  - Physical page frame -> set of virtual pages
- One approach: Inverted page table
  - Hash from virtual page -> physical page
  - Space proportional to # of physical pages

Inverted Page Table: 1/2

- In a paging system, each process has its own page table, which usually has many entries.
- To save space, we may build a page table which has one entry for each page frame. Thus, the size of this **inverted page table** is equal to the number of page frames. Why is this saving memory?
- Each entry in an inverted page table has two items:
  - Process ID: the owner of this frame
  - Page Number: the page number in this frame
- Each virtual address has three sections:

  <process-id, page #, offset>

Inverted Page Table: 2/2



Fragmentation in a Paging System

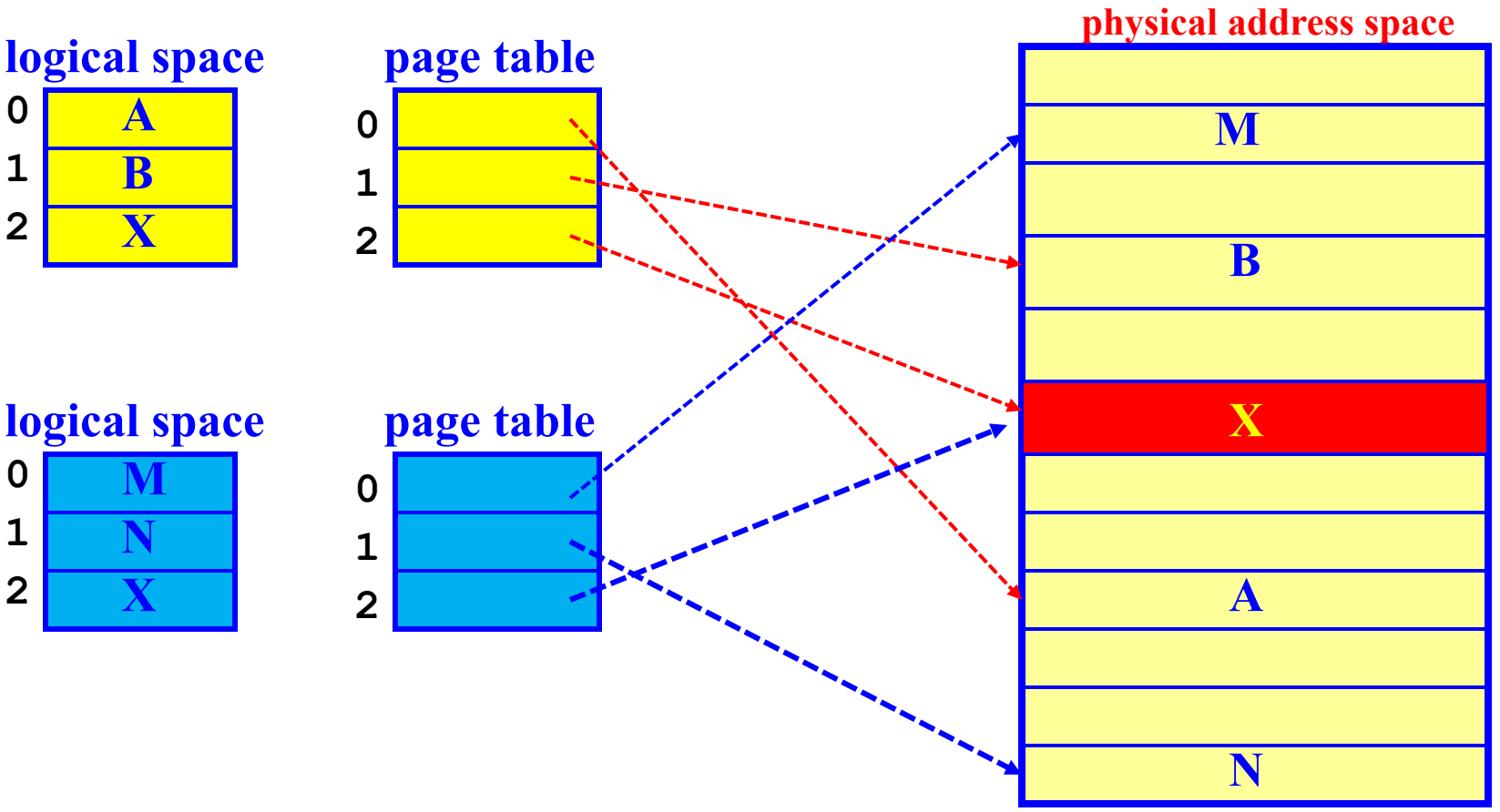
- Does a paging system have fragmentation?
- Paging systems do not have external fragmentation, because un-used page frames can be used by other process.
- Paging systems do have internal fragmentation.
- Because the address space is divided into equal size pages, all but the last one will be filled completely. Thus, the last page may have internal fragmentation and may be 50% full.

Protection in a Paging System

- Is it required to protect among users in a paging system? No, because different processes use different page tables.
- However, we may use a page table length register that stores the length of a process's page table. In this way, a process cannot access the memory beyond its region. Compare this with the base/limit register pair.
- We may add read-only, read-write, or execute bits in page table to enforce r-w-e permission.
- We may also add a valid/invalid bit to each page entry to indicate if a page is in memory.

Shared Pages

- Pages may be shared by multiple processes.
- If the code is a *re-entrant* (or *pure*) one, a program does not modify itself, routines can also be shared!

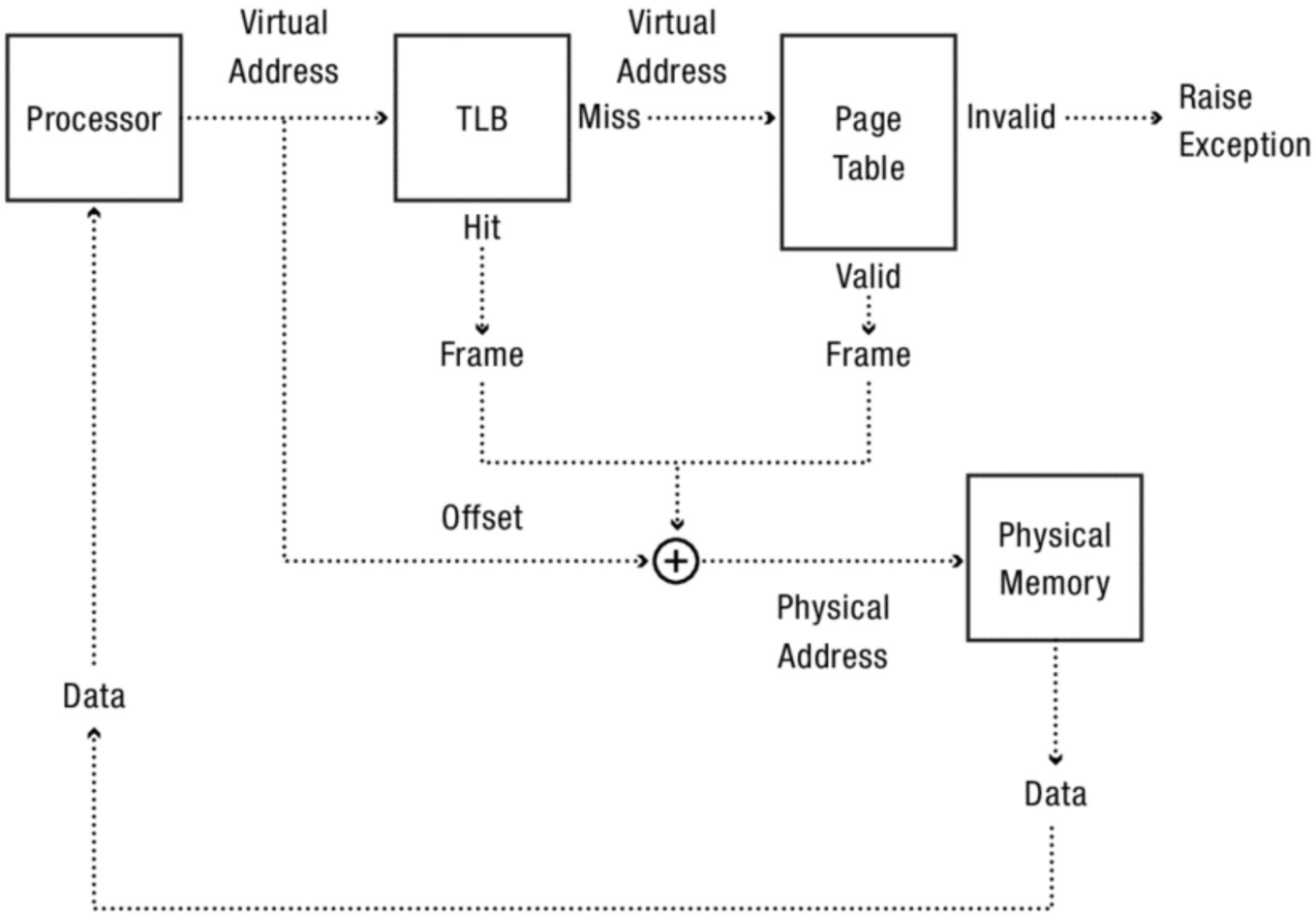


Efficient Address Translation

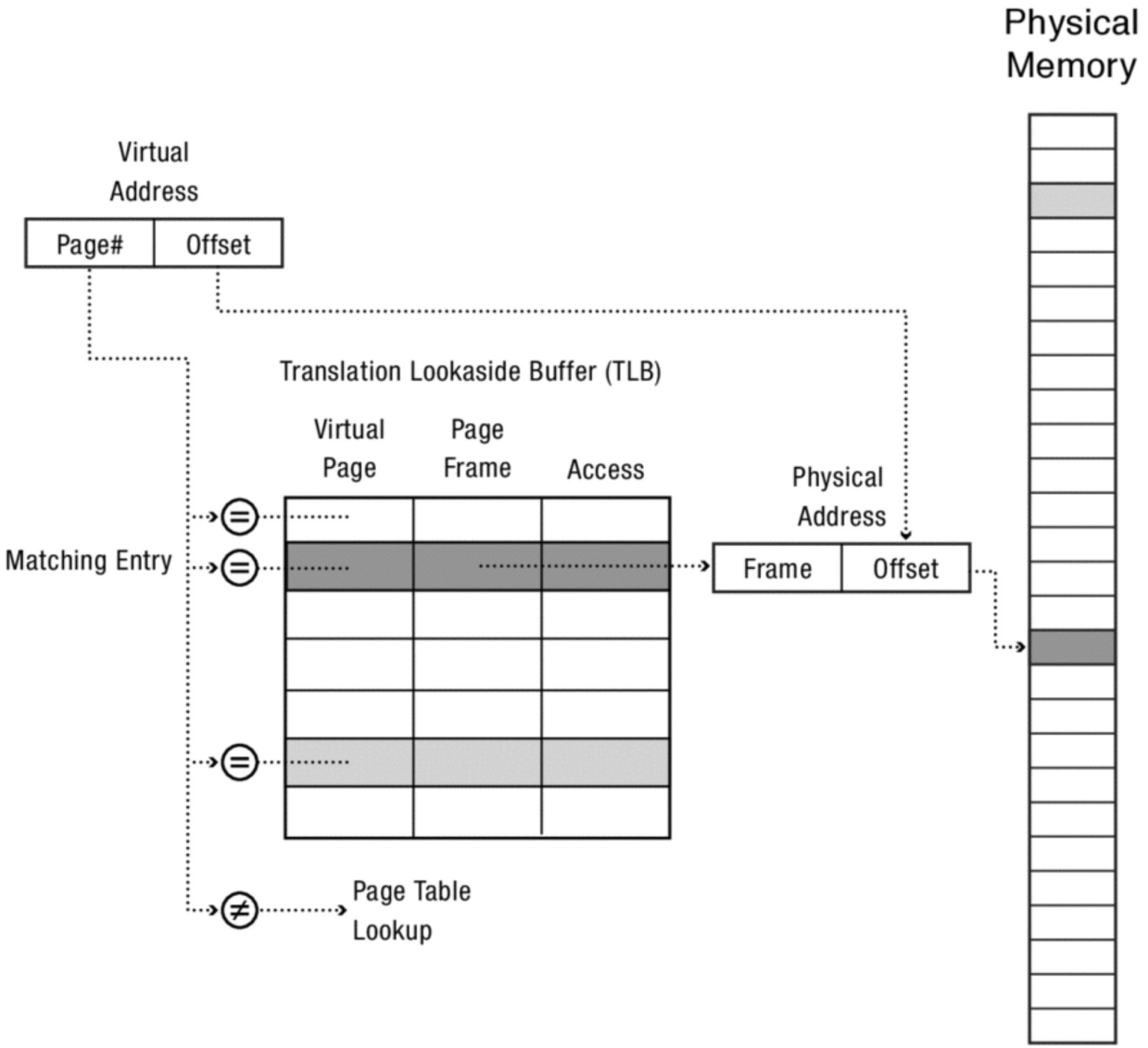
- Translation lookaside buffer (TLB)
  - Cache of recent virtual page -> physical page translations
  - If cache hit, use translation
  - If cache miss, walk multi-level page table
- Cost of translation =
  - Cost of TLB lookup +

    Prob(TLB miss) * cost of page table lookup

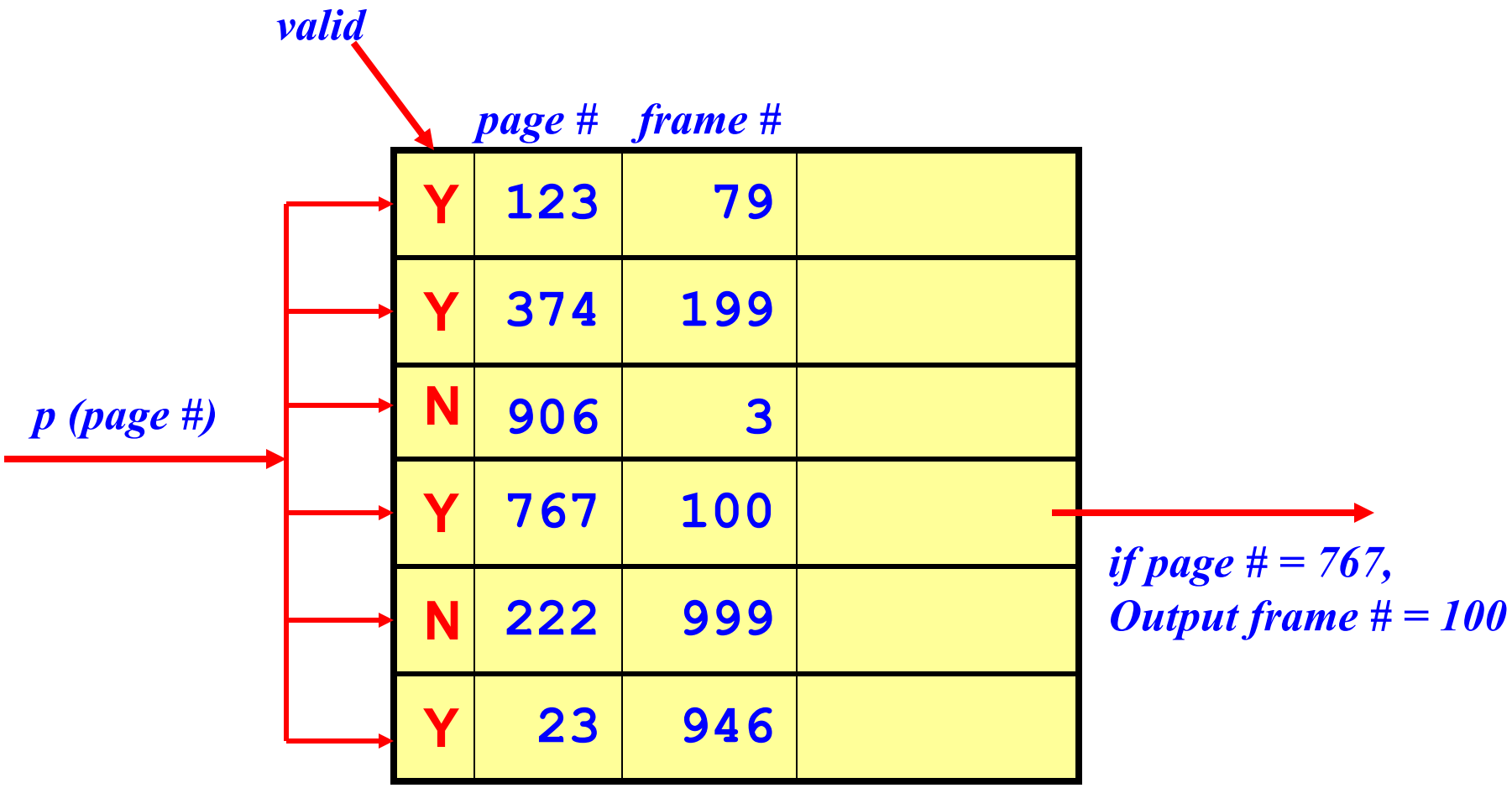
TLB and Page Table Translation



TLB Lookup


Physical Memory
Virtual Address
Page#
Offset
Translation Lookaside Buffer (TLB)
Virtual Page
Page Frame
Access
Matching Entry
Physical Address
Frame
Offset
Page Table Lookup

Translation Look-Aside Buffer



If the TLB reports no hit, then we go for a page table look up!