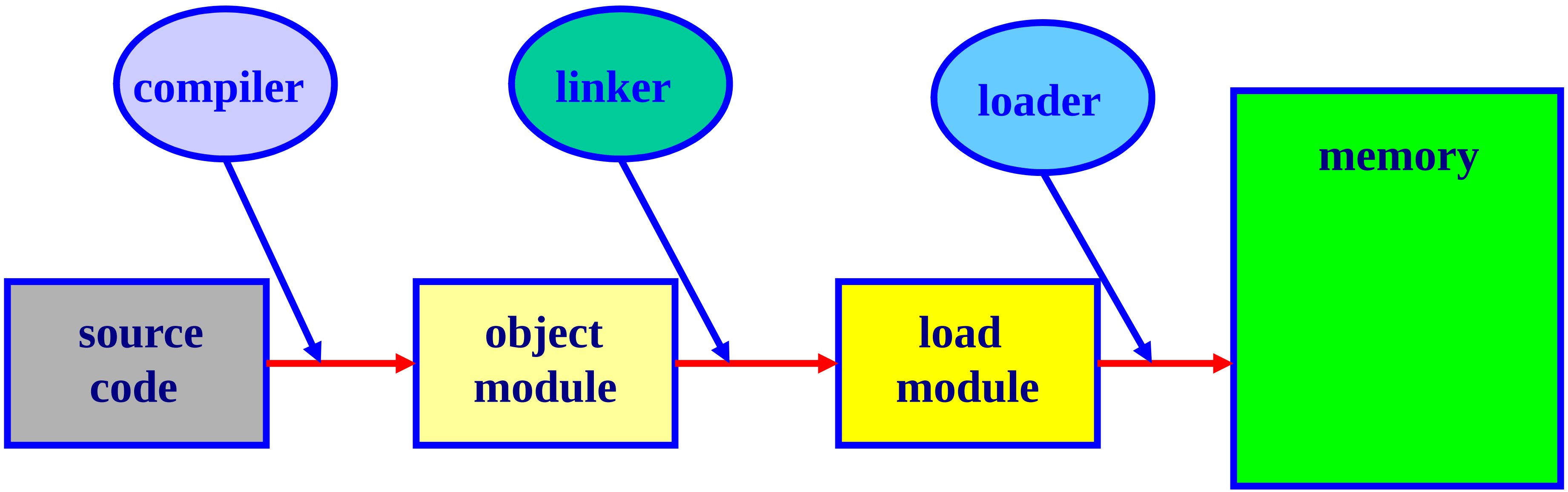
Part III
Storage Management
Chapter 8: Memory Management

Address Generation

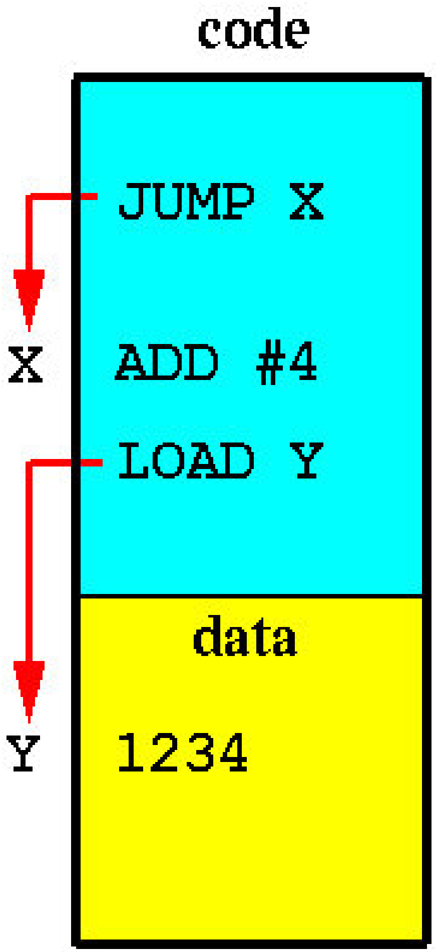
- **Address generation has three stages:**
  - **Compile**: compiler
  - **Link**: linker or linkage editor
  - **Load**: loader

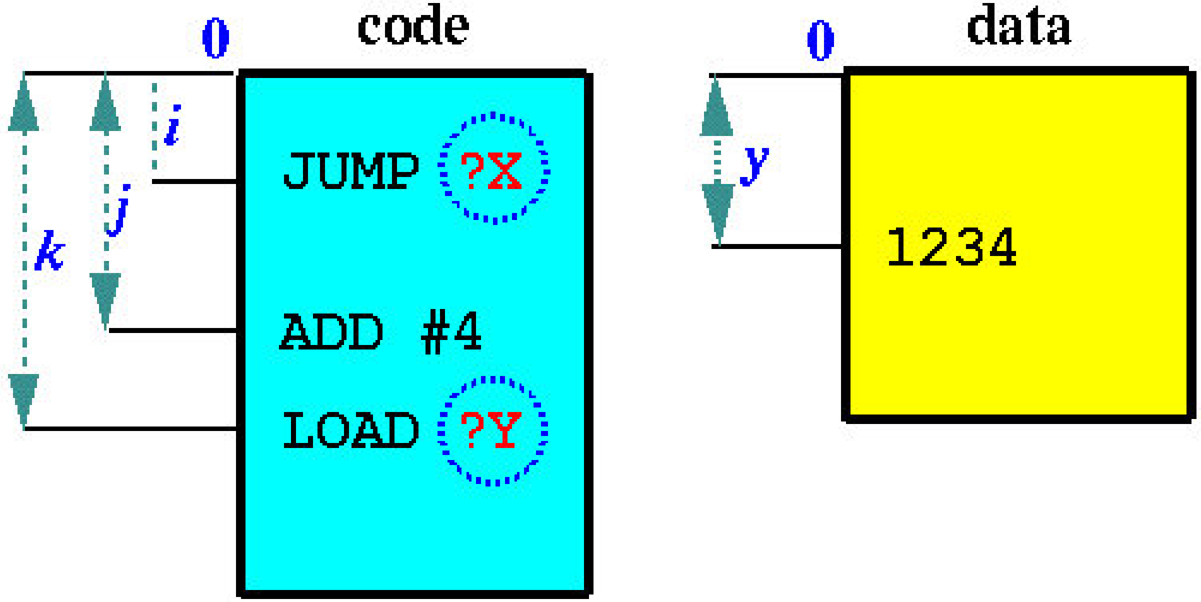


Three Address Binding Schemes

- **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: Since the compiler may not know the absolute address, it 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

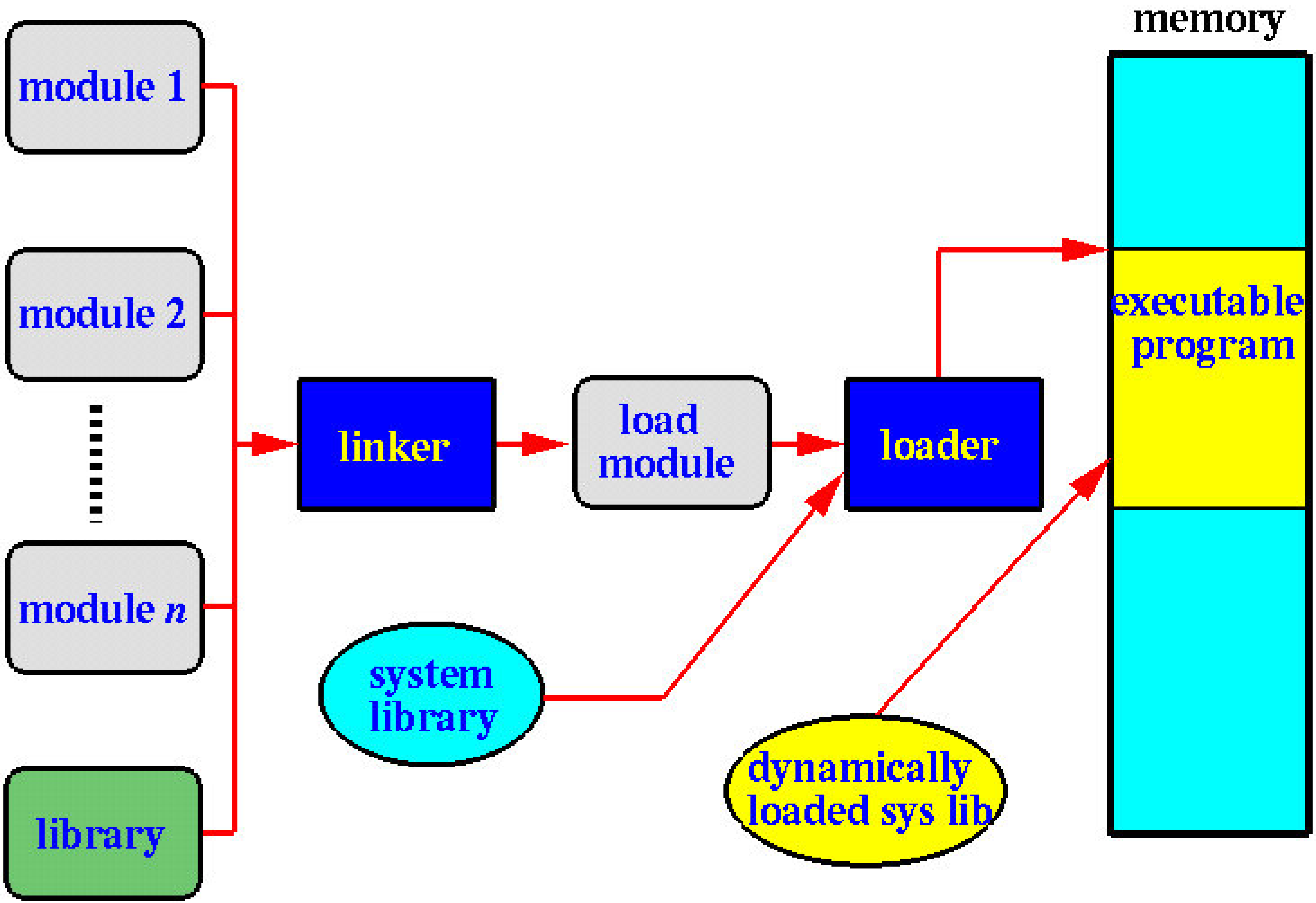




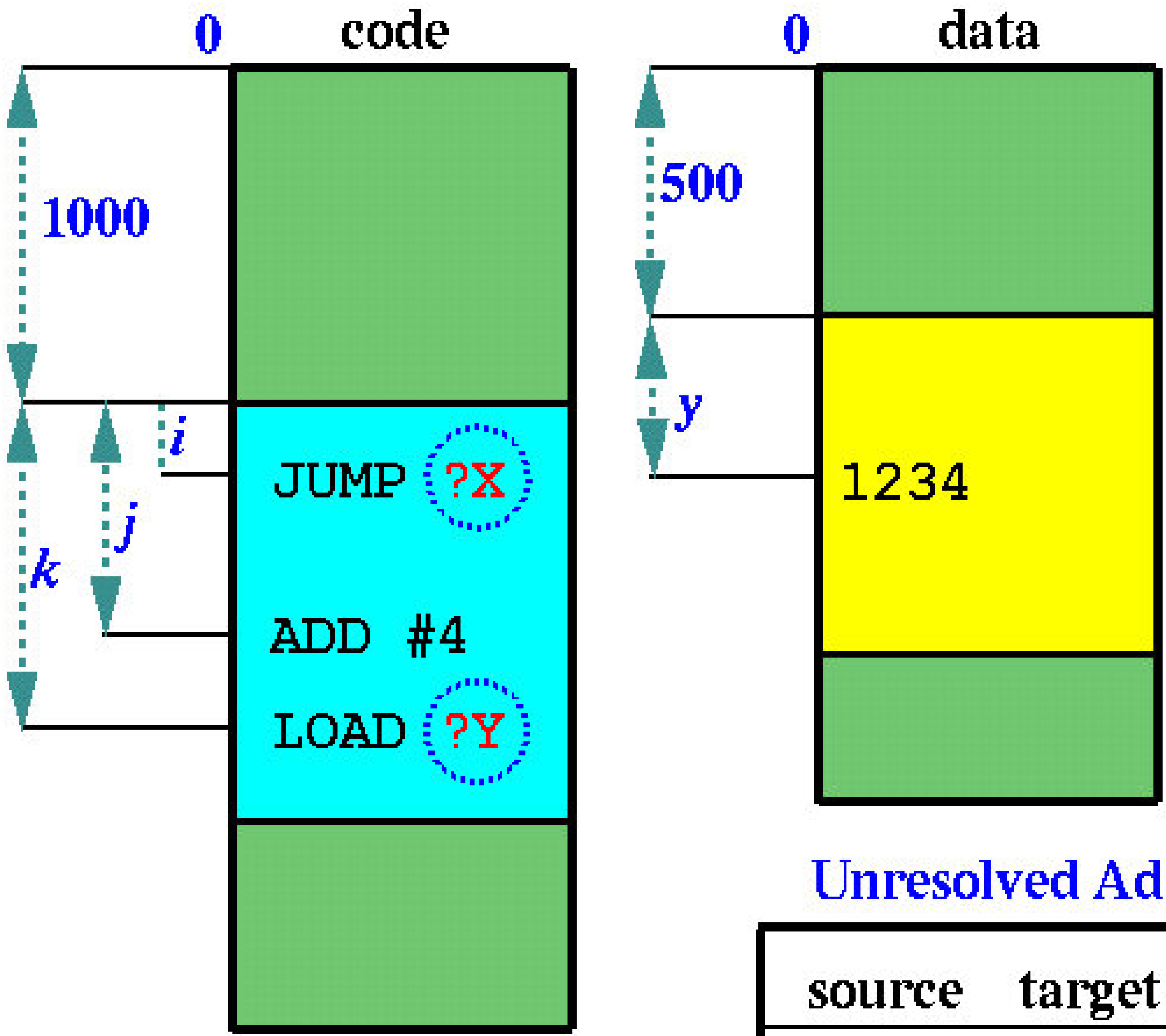
Unresolved Address Table

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

Linking and Loading

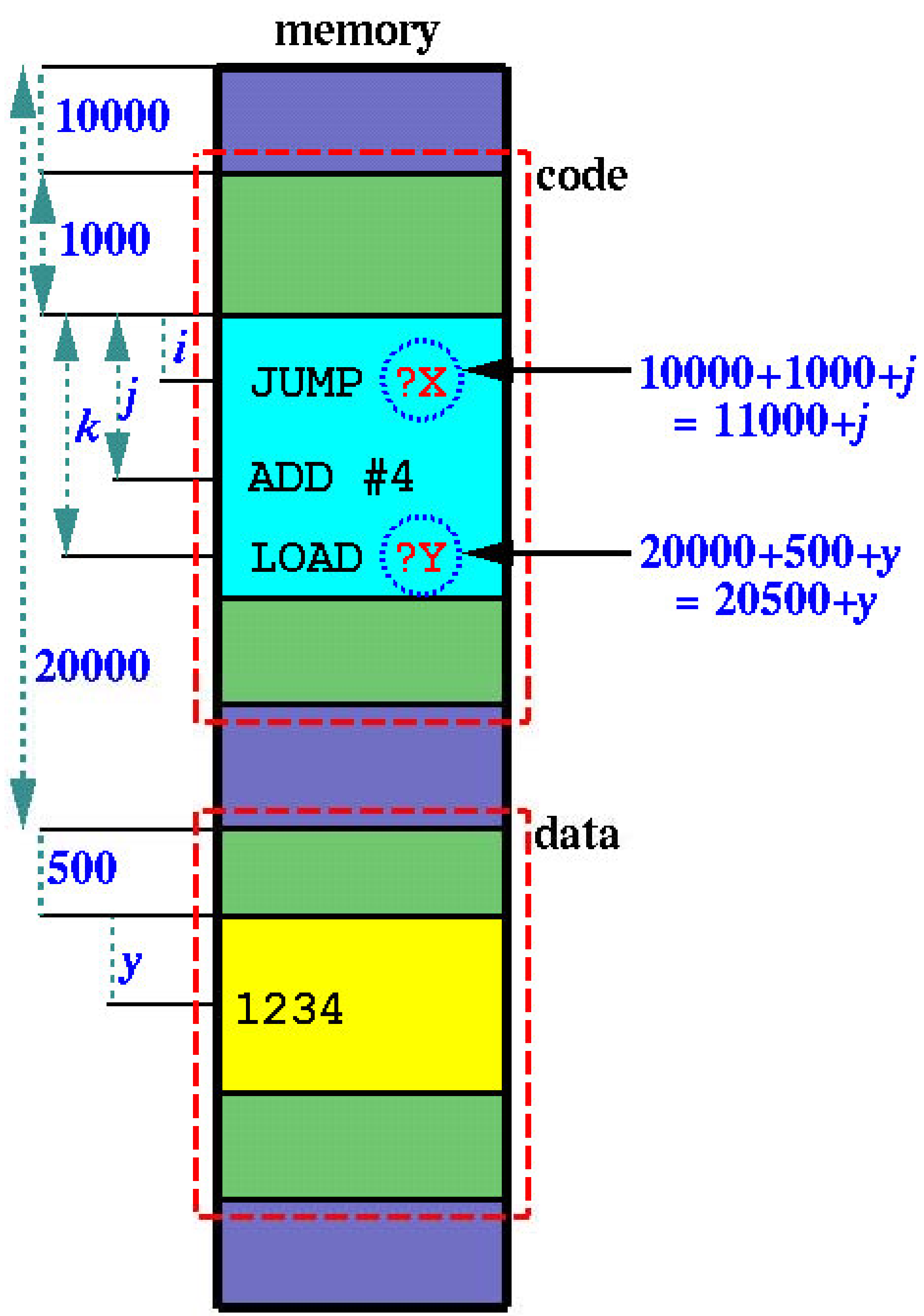


Address Generation: Static Linking



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.

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)**

Dynamic Loading

- **Some routines in a program (*e.g.*, error handling) may not be used frequently.**
- **With *dynamic loading*, a routine is not loaded until it is called.**
- **To use dynamic loading, all routines must be in a relocatable format.**
- **The main program is loaded and executes.**
- **When a routine *A* calls *B*, *A* checks to see if *B* is loaded. If *B* is not loaded, the relocatable linking loader is called to load *B* and updates the address table. Then, control is passed to *B*.**

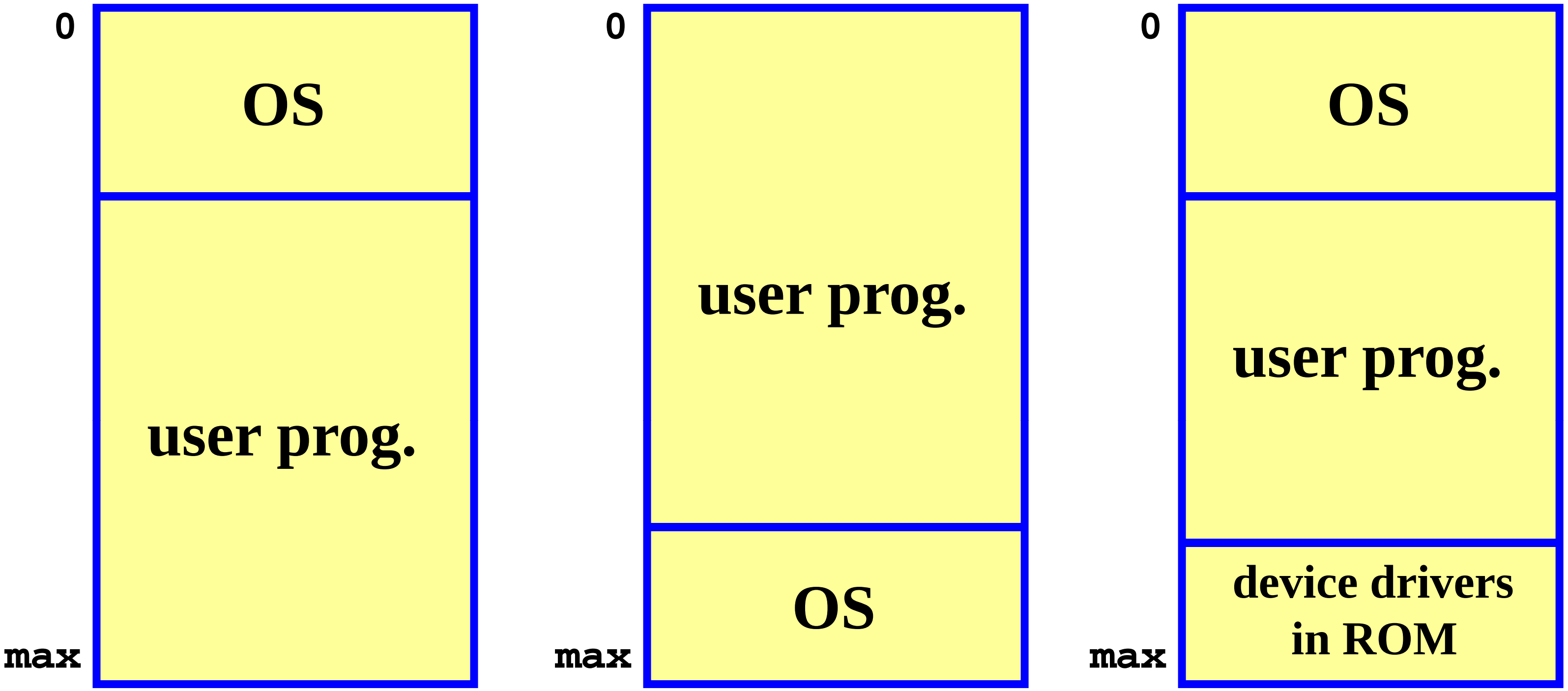
Dynamic Linking

- **Dynamic loading postpones the loading of routines until run-time. *Dynamic linking* postpones both linking and loading until run-time.**
- **A *stub* is added to each reference of library routine. A stub is a small piece of code that indicates how to locate and load the routine if it is not loaded.**
- **When a routine is called, its stub is executed. The called routine is loaded, its address replaces the stub, and executes.**
- **Dynamic linking usually applies to language and system libraries. A Windows `DLL` is a dynamic linking library.**

Memory Management Schemes

- **Monoprogramming Systems: MS-DOS**
- **Multiprogramming Systems:**
  - **Fixed Partitions**
  - **Variable Partitions**
  - **Paging**

Monoprogramming Systems

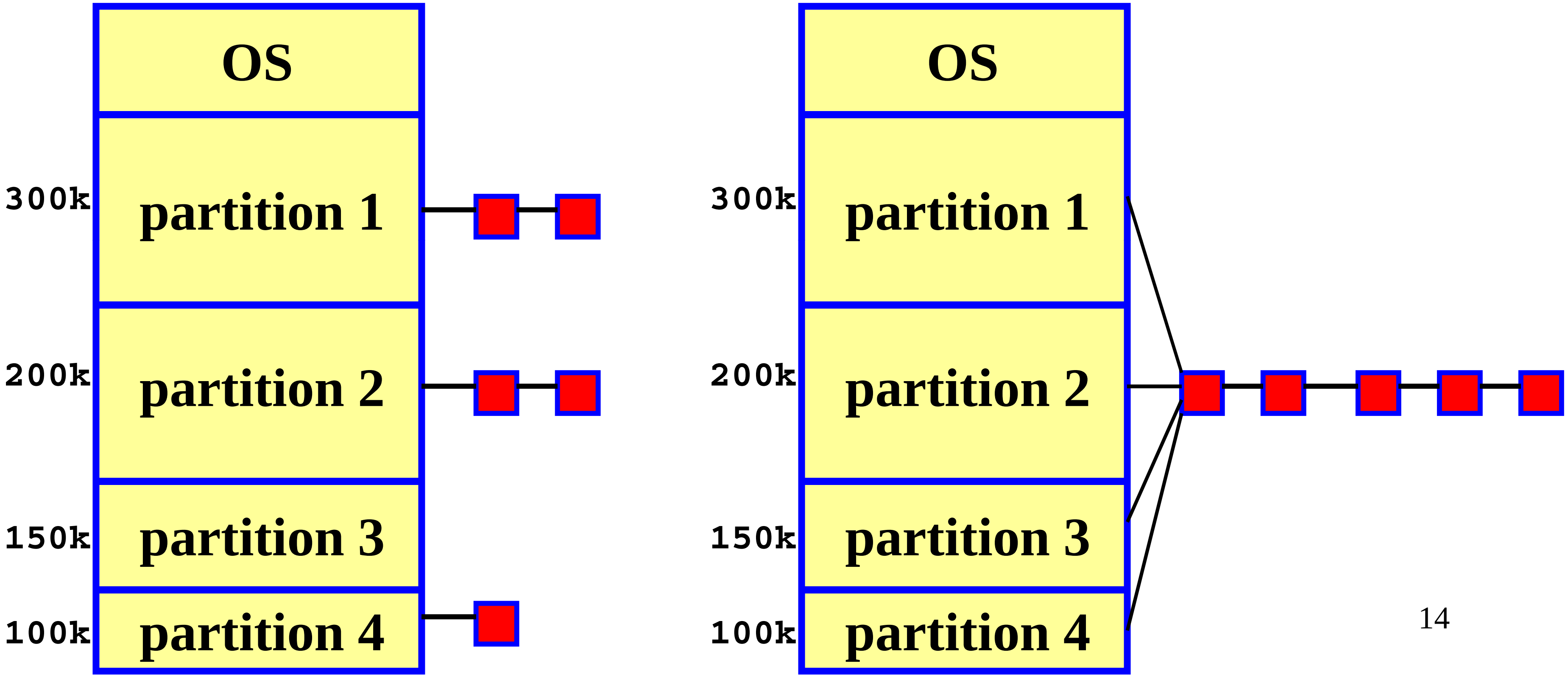


Why Multiprogramming?

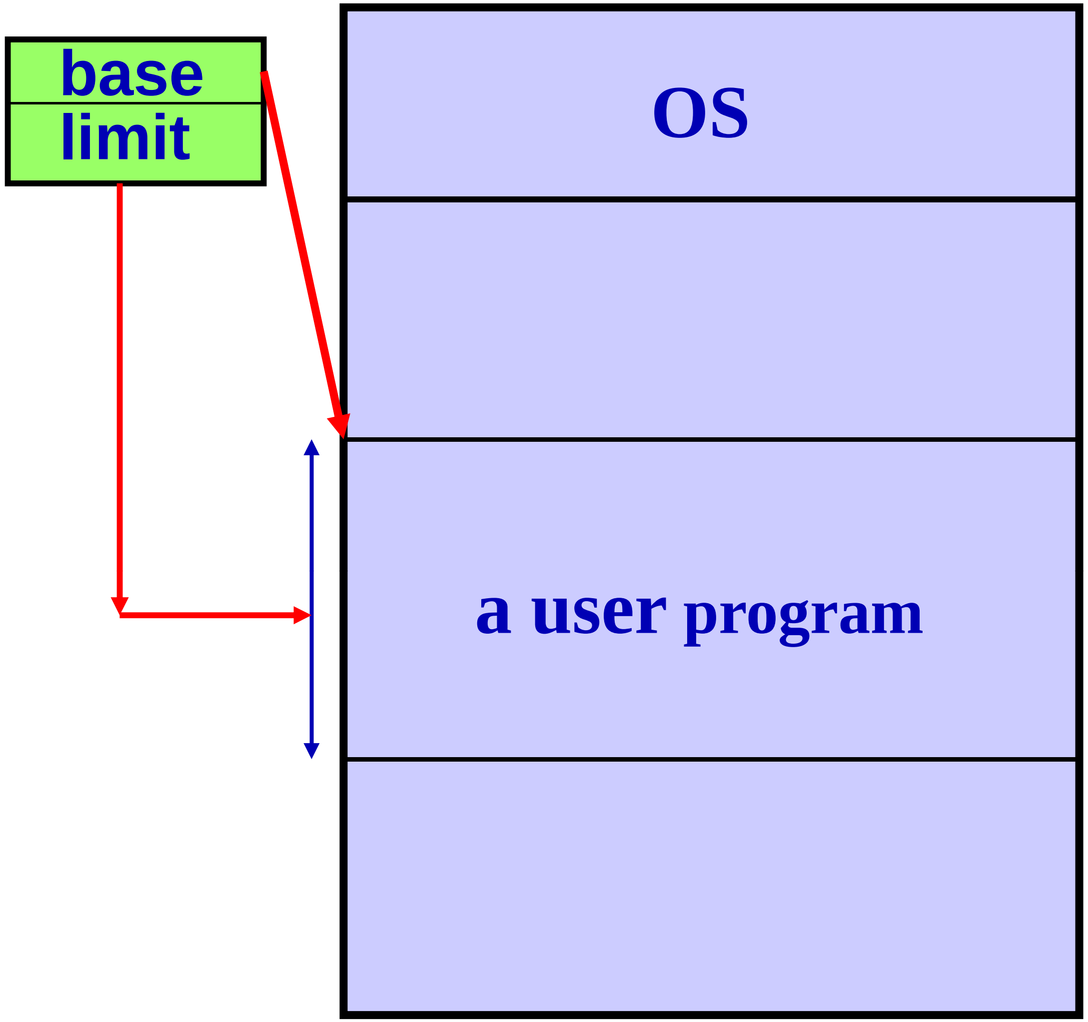
- **Suppose a process spends a fraction of p of its time in I/O wait state.**
- **Then, the probability of n processes being *all* in wait state at the same time is p^n.**
- **The CPU utilization is $1 - p^n$.**
- **Thus, the more processes in the system, the higher the CPU utilization.**
- **Well, since CPU power is limited, throughput decreases when n is sufficiently large.**

Multiprogramming with Fixed Partitions

- Memory is divided into *n* (possibly unequal) partitions.
- Partitioning may be done at the startup time and altered later.
- Each partition may have a job queue. Or, all partitions share the same job queue.

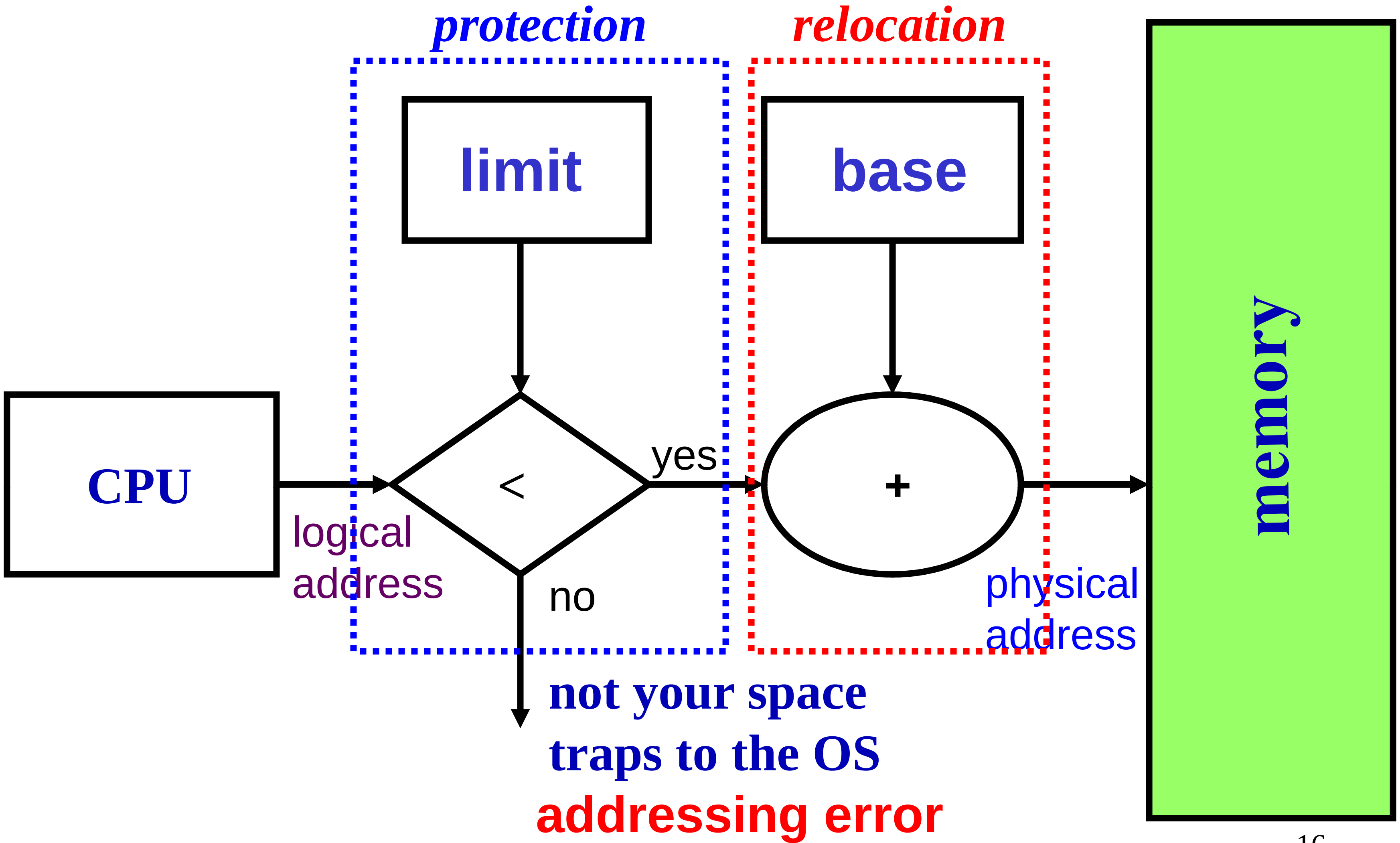


Relocation and Protection: 1/2

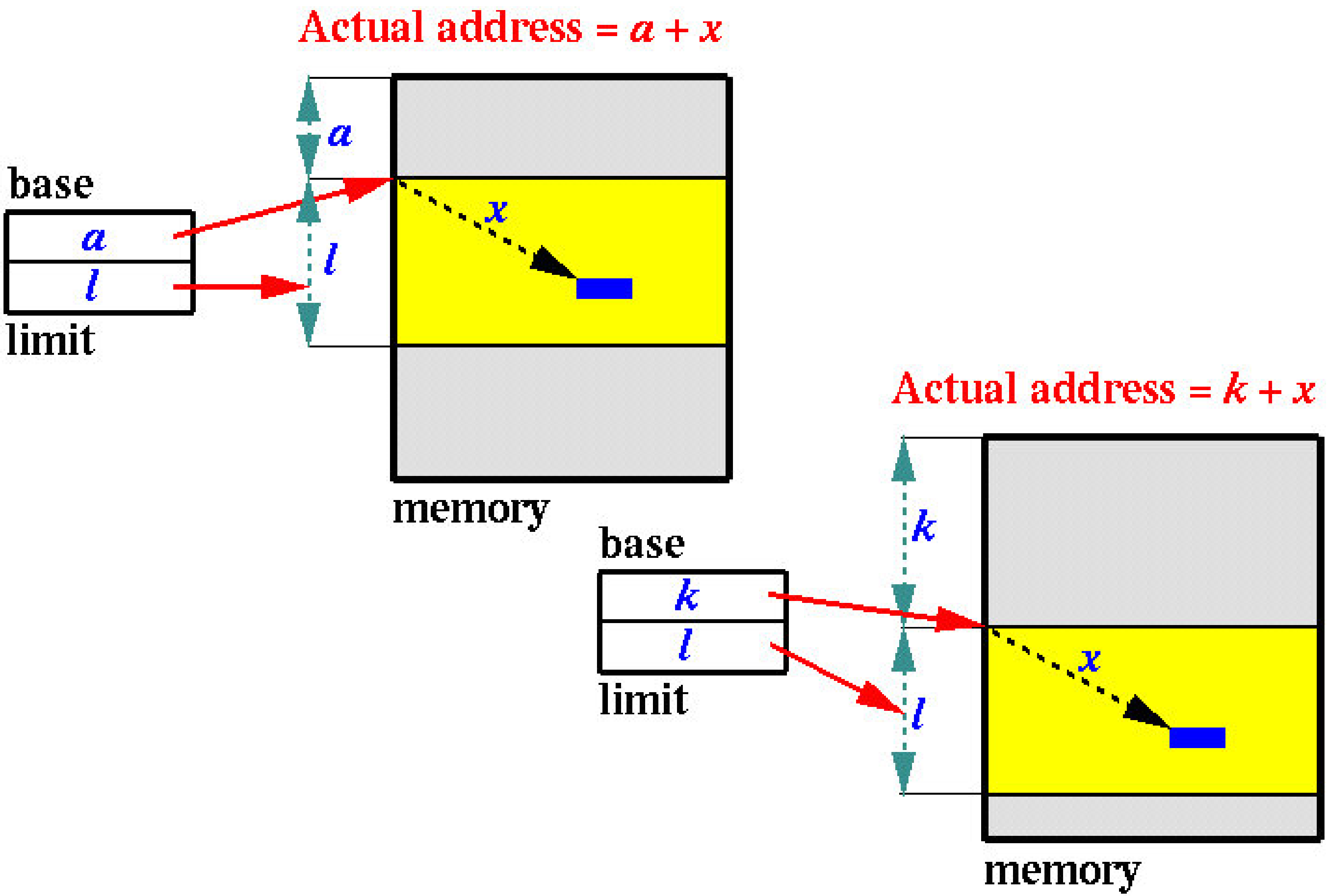


- **Because executables may run in any partition, relocation and protection are needed.**
- **Recall the base/limit register pair for memory protection.**
- **It could also be used for relocation.**
- **Linker generates *relocatable* code starting with 0. The base register contains the starting address.**

Relocation and Protection: 2/2

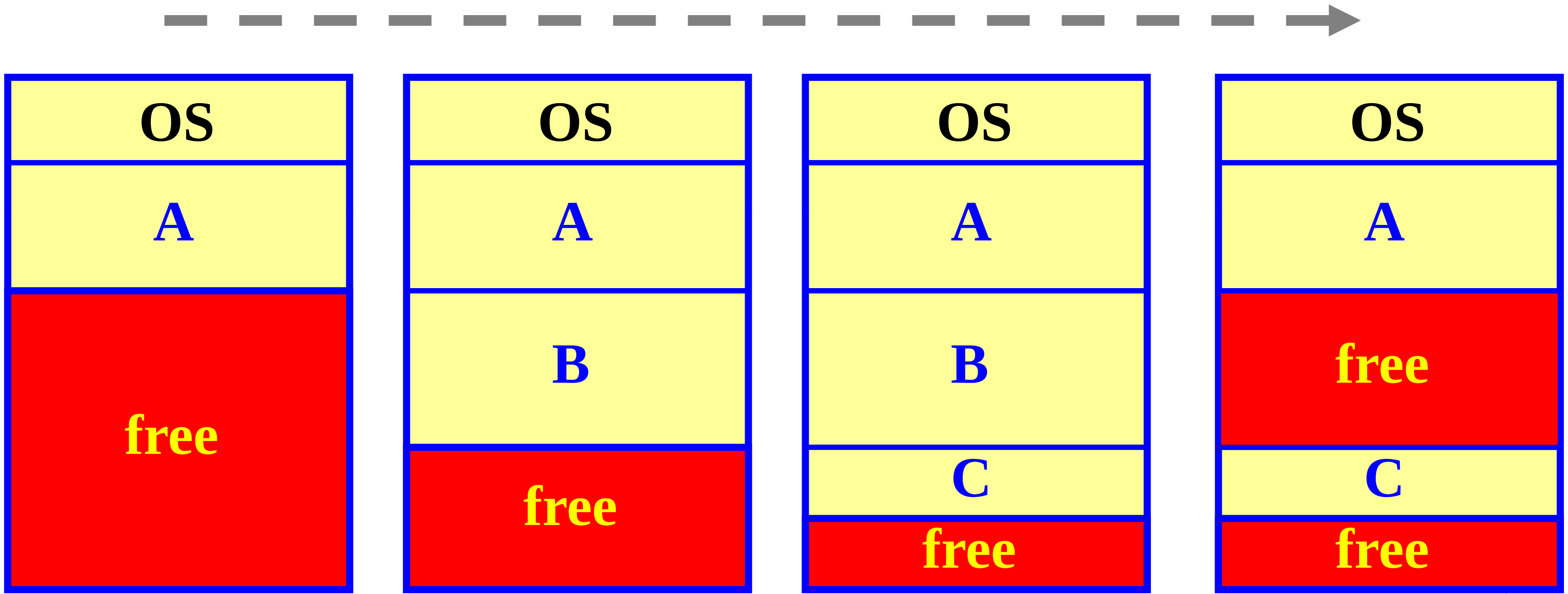


Relocation: How does it work?



Multiprogramming with Variable Partitions

- The OS maintains a memory pool, and allocates whatever a job needs.
- Thus, partition sizes are not fixed, The number of partitions also varies.

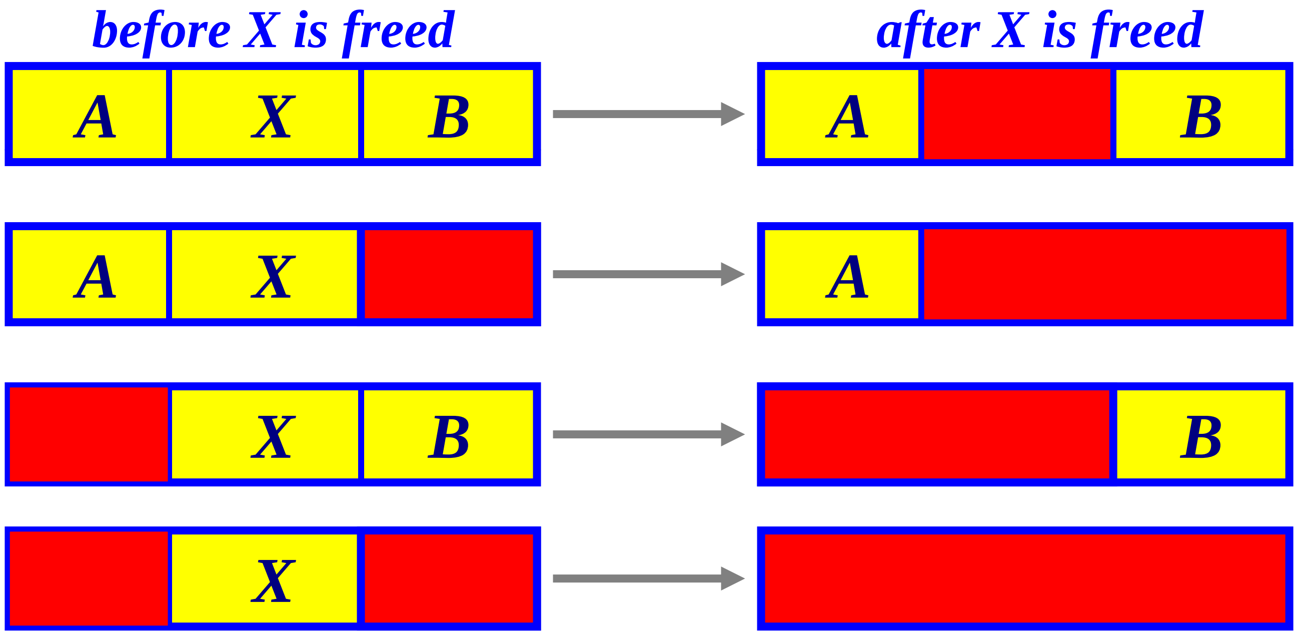


Memory Allocation: 1/2

- **When a memory request is made, the OS searches all free blocks (*i.e.*, holes) to find a *suitable* one.**
- **There are some commonly seen methods:**
  - **First Fit: Search starts at the *beginning* of the set of holes and allocate the first large enough hole.**
  - **Next Fit: Search starts from *where the previous first-fit search ended*.**
  - **Best-Fit: Allocate the *smallest* hole that is larger than the request one.**
  - **Worst-Fit: Allocate the *largest* hole that is larger than the request one.**

Memory Allocation: 2/2

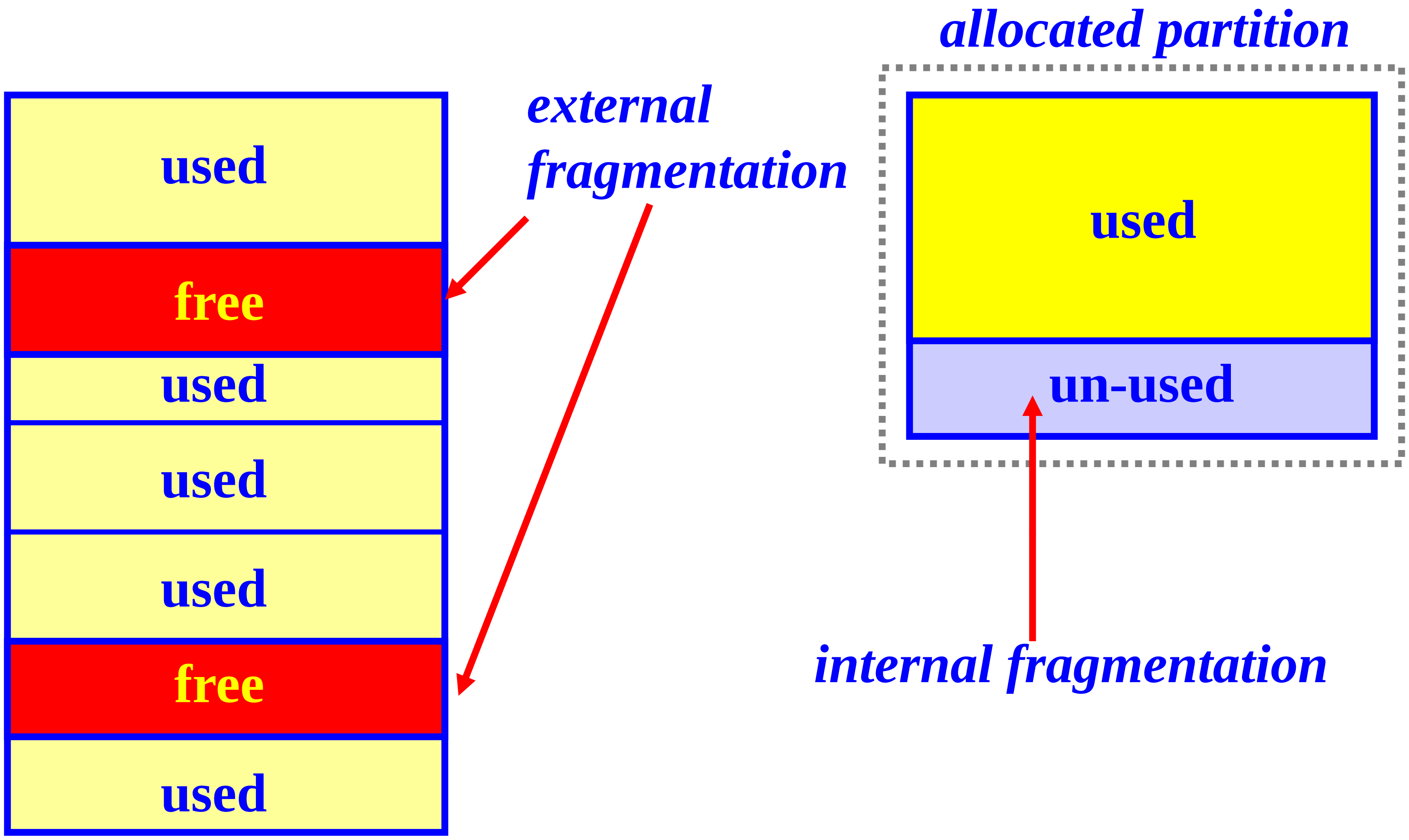
- **If the hole is larger than the requested size, it is cut into two. The one of the requested size is given to the process, the remaining one becomes a *new* hole.**
- **When a process returns a memory block, it becomes a hole and must be combined with its neighbors.**



Fragmentation

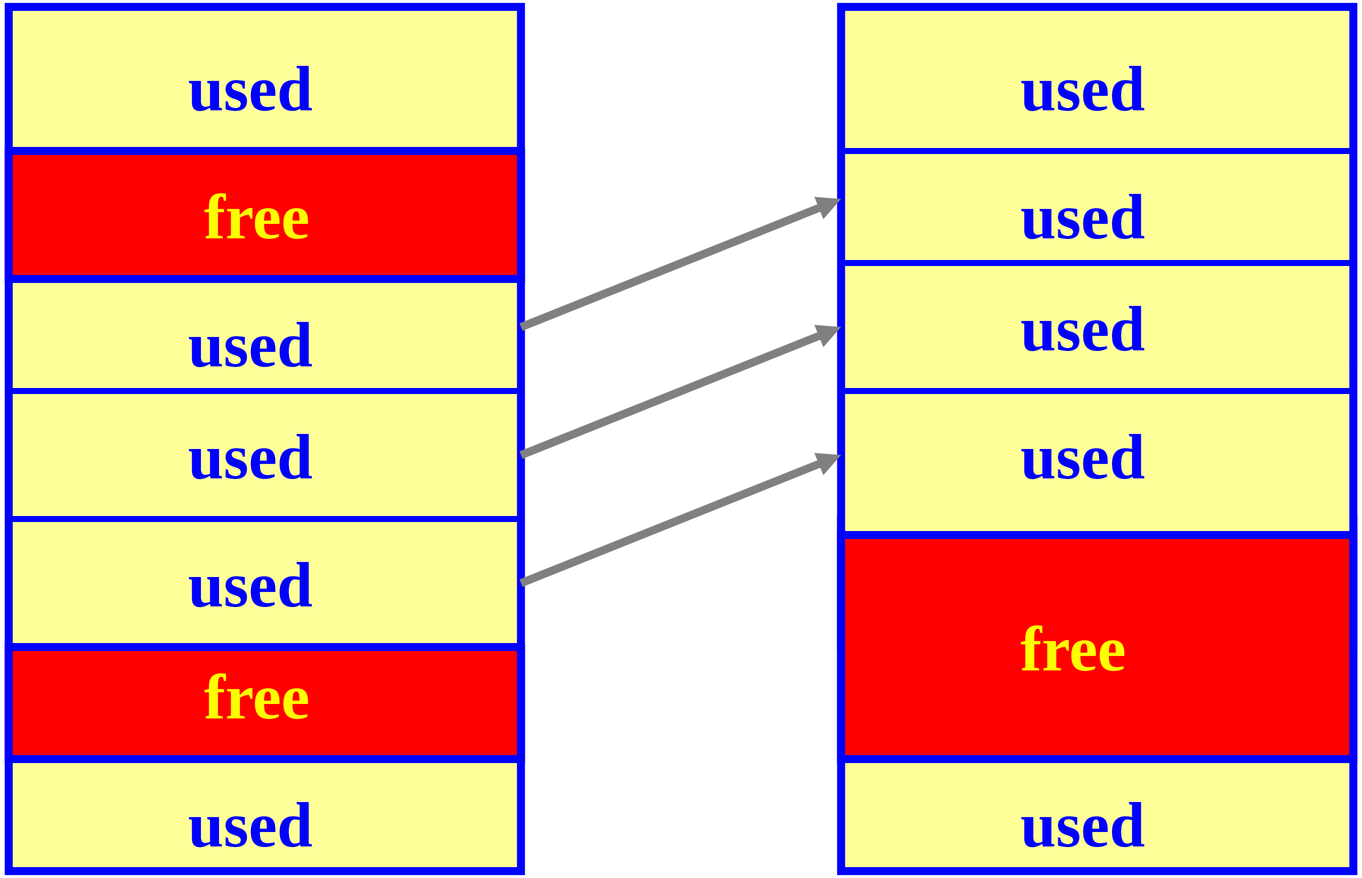
- **Processes are loaded and removed from memory, eventually memory is cut into small holes that are not large enough to run any incoming process.**
- **Free memory holes between allocated ones are called *external fragmentation*.**
- **It is unwise to allocate exactly the requested amount of memory 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

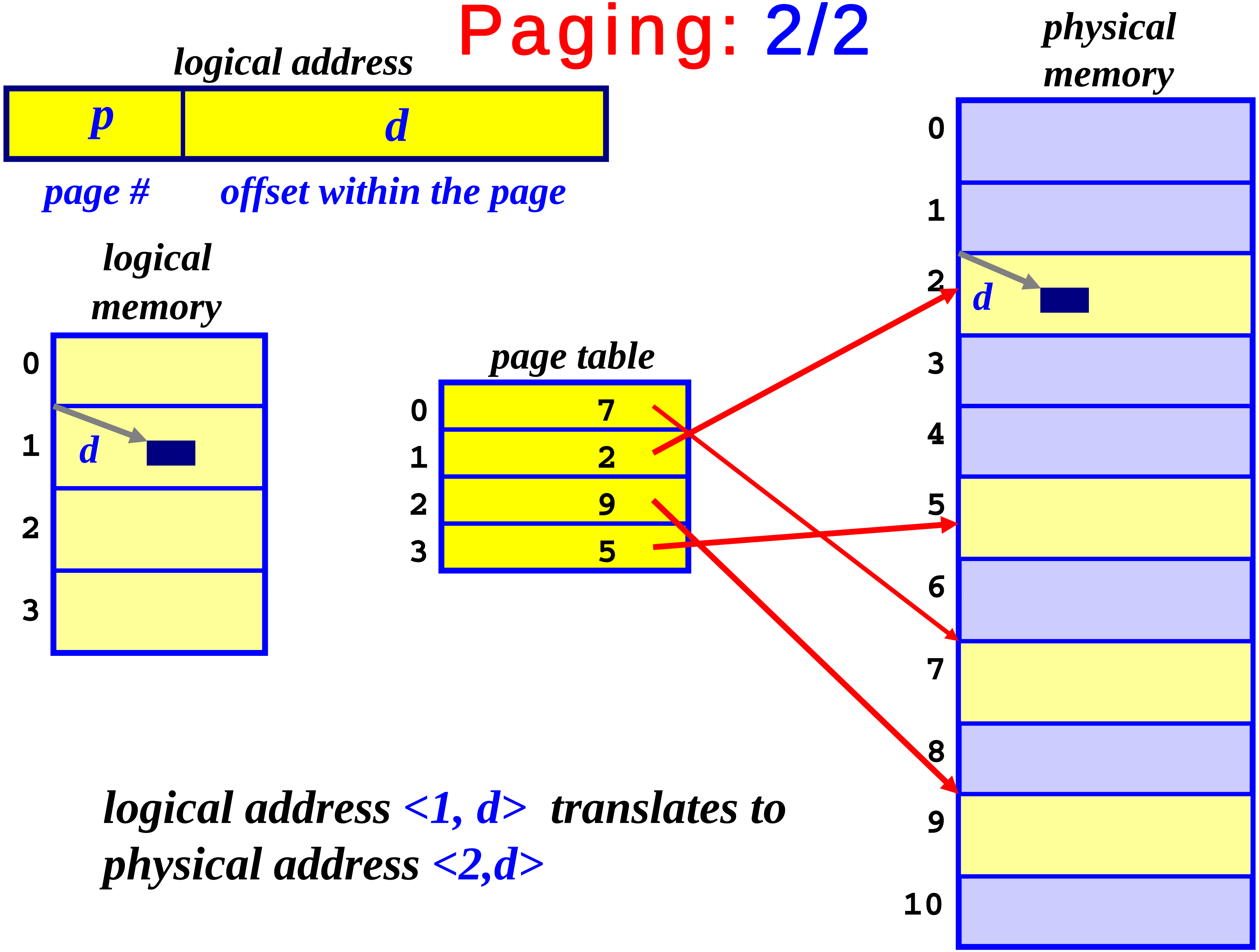
- **If processes are relocatable, we may move used memory blocks together to make a larger free memory block.**



Paging: 1/2

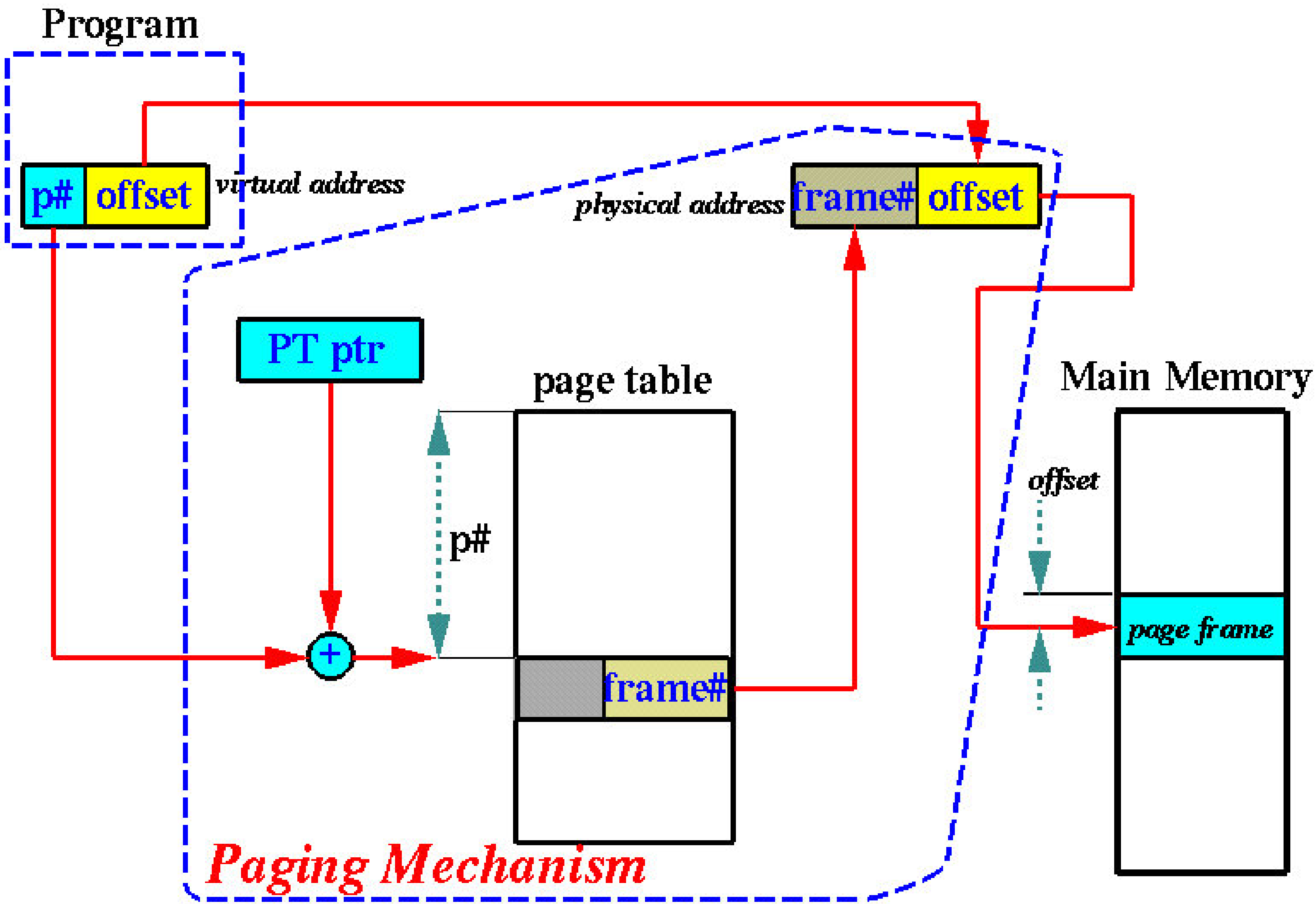
- **The physical memory is divided into fixed-sized *page frames*, or *frames*.**
- **The virtual address space is also divided into blocks of the same size, called *pages*.**
- **When a process runs, its pages are loaded into page frames.**
- **A page table stores the *page numbers* and their corresponding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

Paging: 2/2

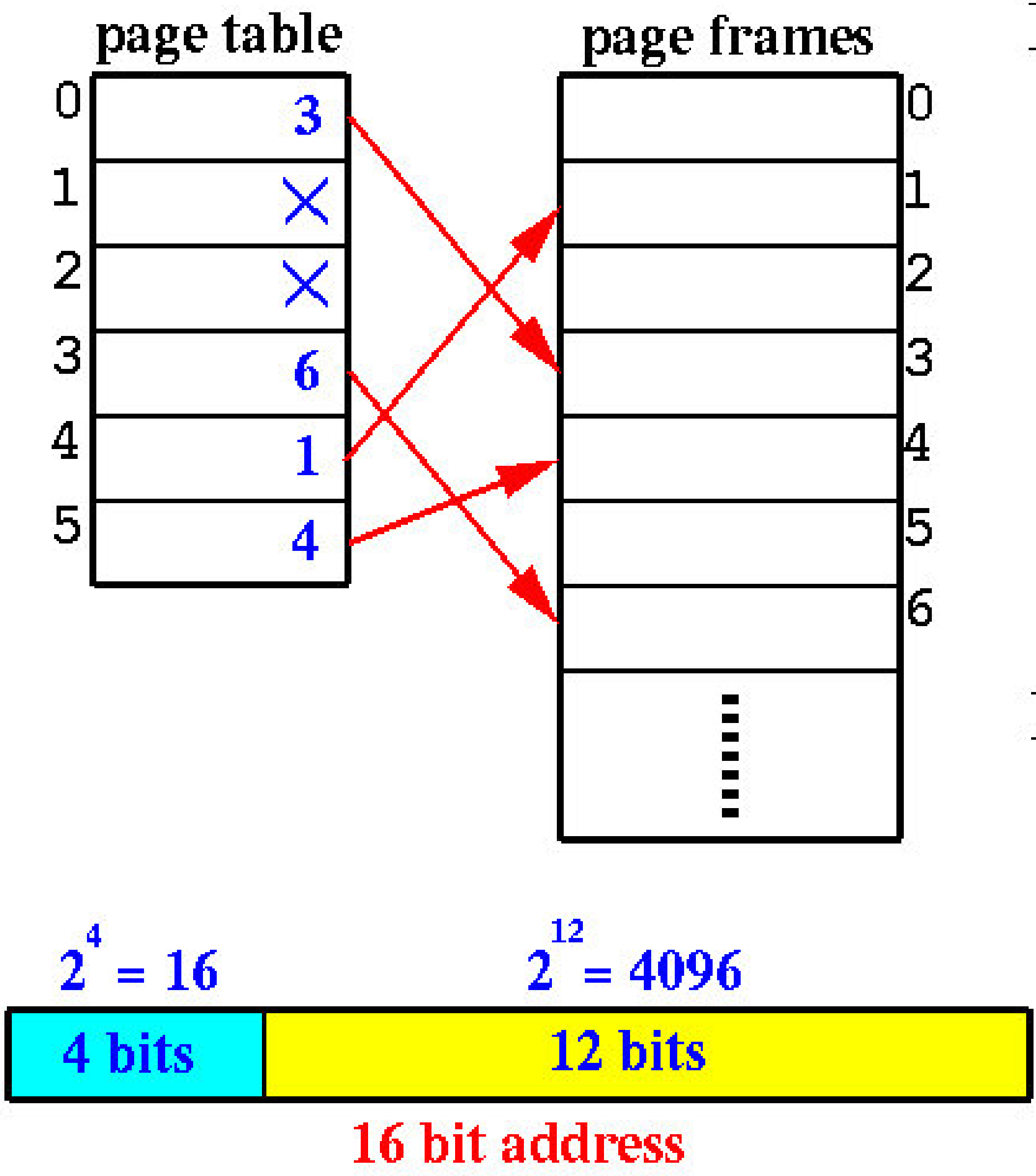


logical address <1, d> translates to physical address <2,d>

Address Translation



Address Translation: Example



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.**

Translation Look-Aside Buffer

valid	*page #*	*frame #*	
Y	123	79	
Y	374	199	
N	906	3	
Y	767	100	
N	222	999	
Y	23	946	

p (page #)

if page # = 767,
Output frame # = 100

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

Fragmentation in a Paging System

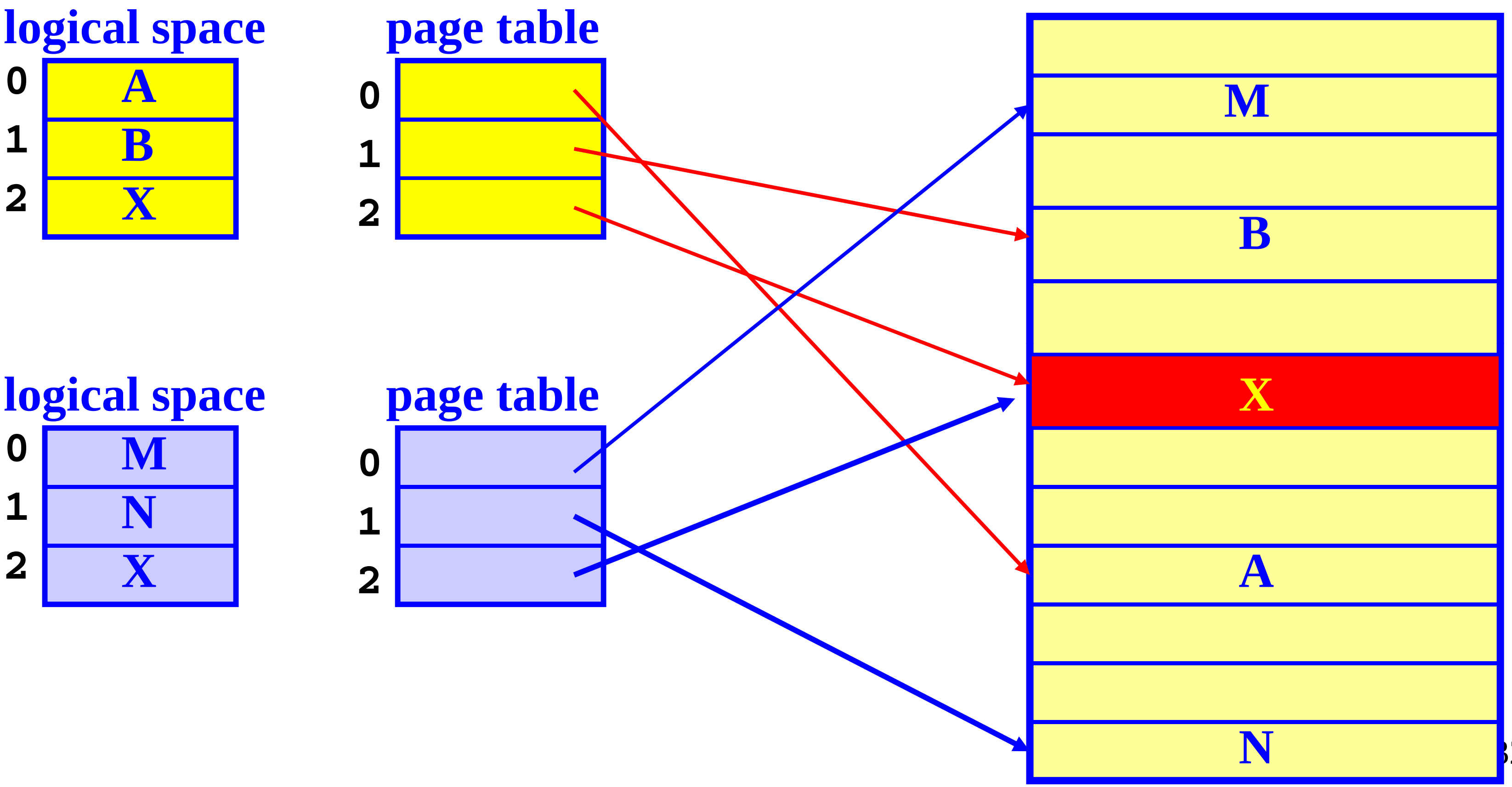
- **Does a paging system have fragmentation?**
- **Paging systems do not have external fragmentation, because un-used page frames can be used by other process.**
- **Paging systems do have internal fragmentation.**
- **Because the address space is divided into equal size pages, all but the last one will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

Protection in a Paging System

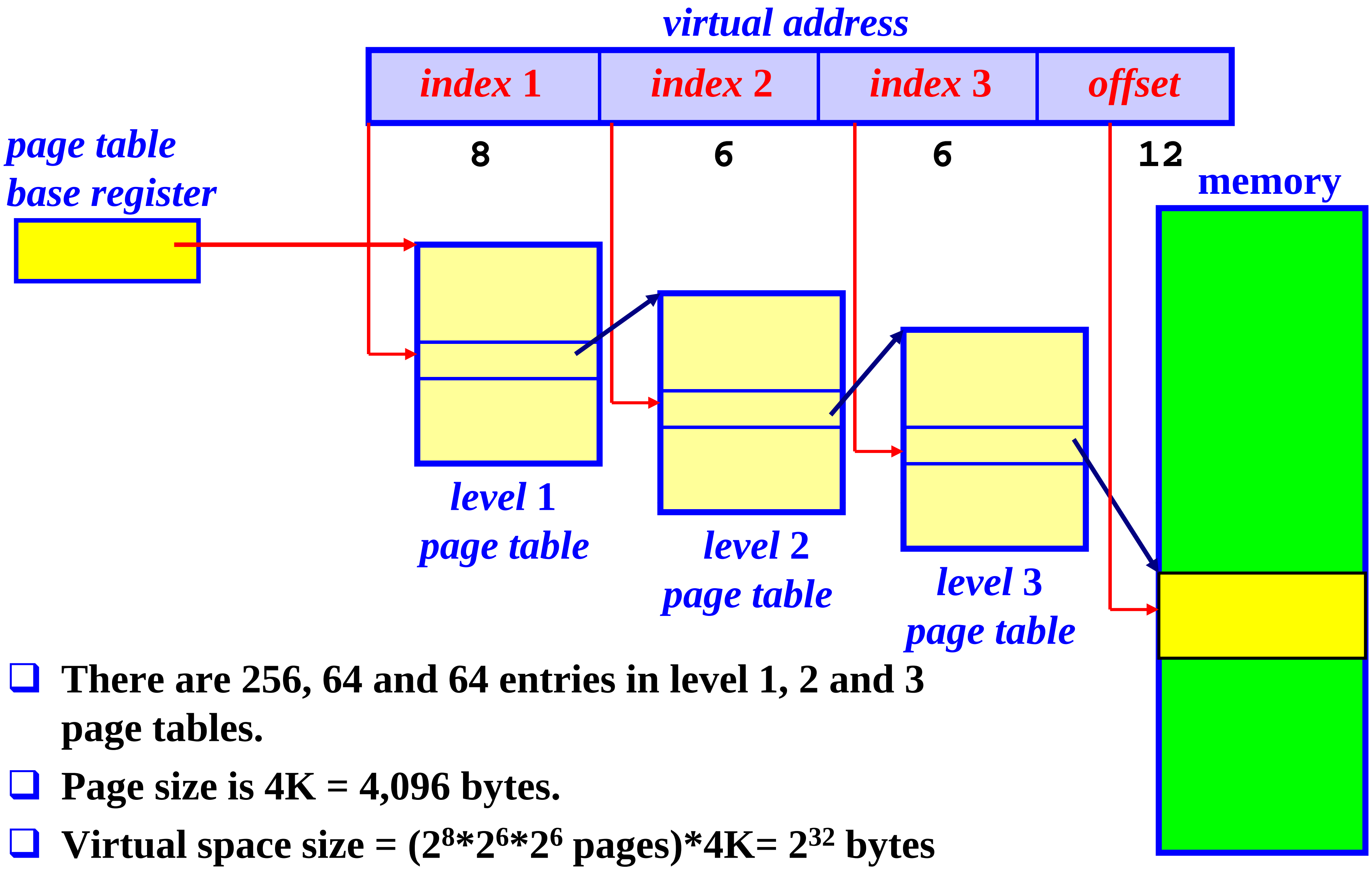
- **Is it required to protect among users in a paging system? No, because different processes use different page tables.**
- **However, we may use a page table length register that stores the length of a process's page table. In this way, a process cannot access the memory beyond its region. Compare 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!



Multi-Level Page Table



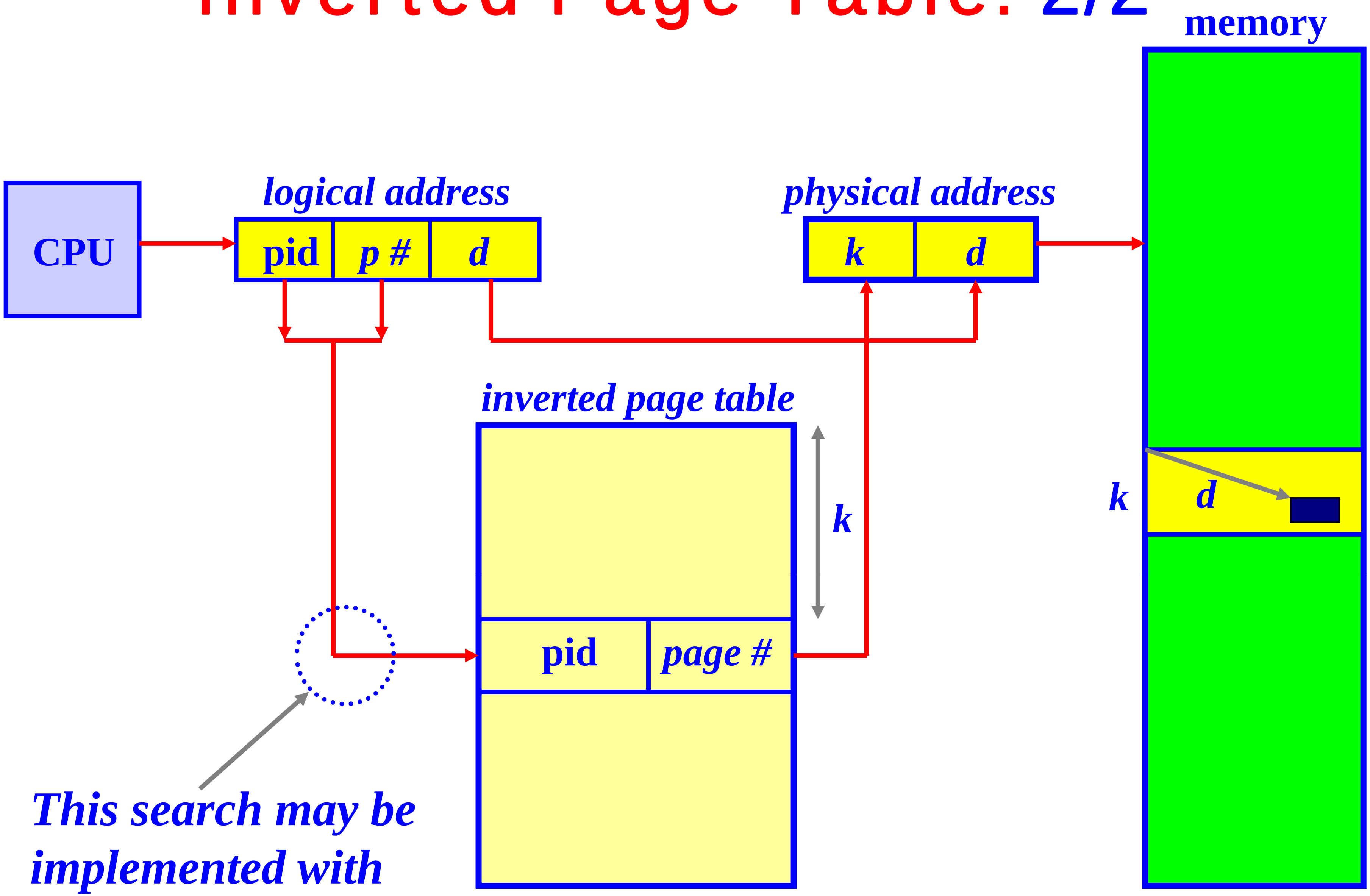
- There are 256, 64 and 64 entries in level 1, 2 and 3 page tables.
- Page size is 4K = 4,096 bytes.
- Virtual space size = ($2^8 * 2^6 * 2^6$ pages)*4K= 2^{32} bytes

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



This search may be implemented with hashing

The End