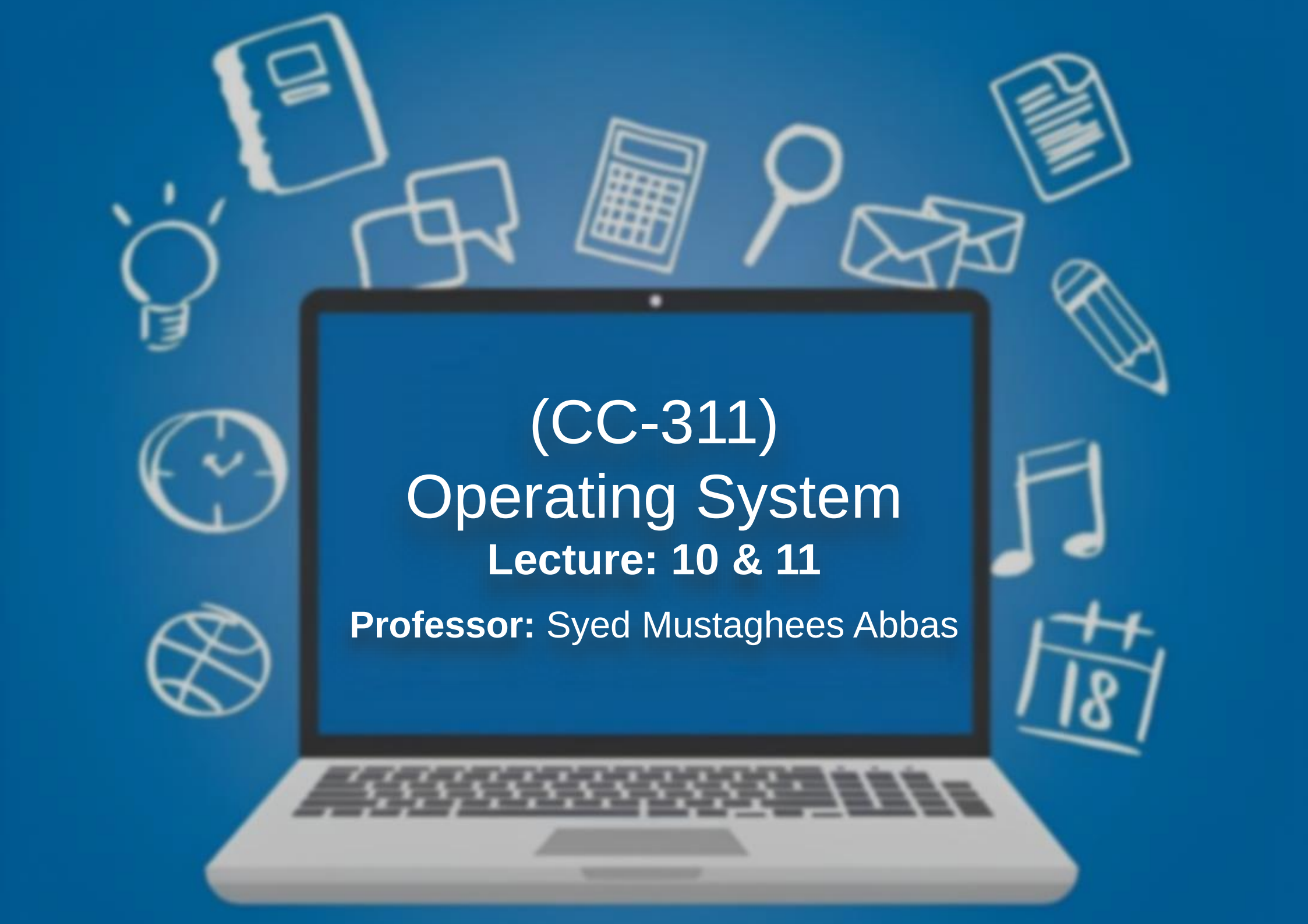

(CC-311)
Operating System
Lecture: 10 & 11
Professor: Syed Mustaghees Abbas

Memory Management

Background

- The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be in main memory (at least partially) during execution.

- Main memory and the registers built into the processor itself are the only storage that the CPU can access directly.

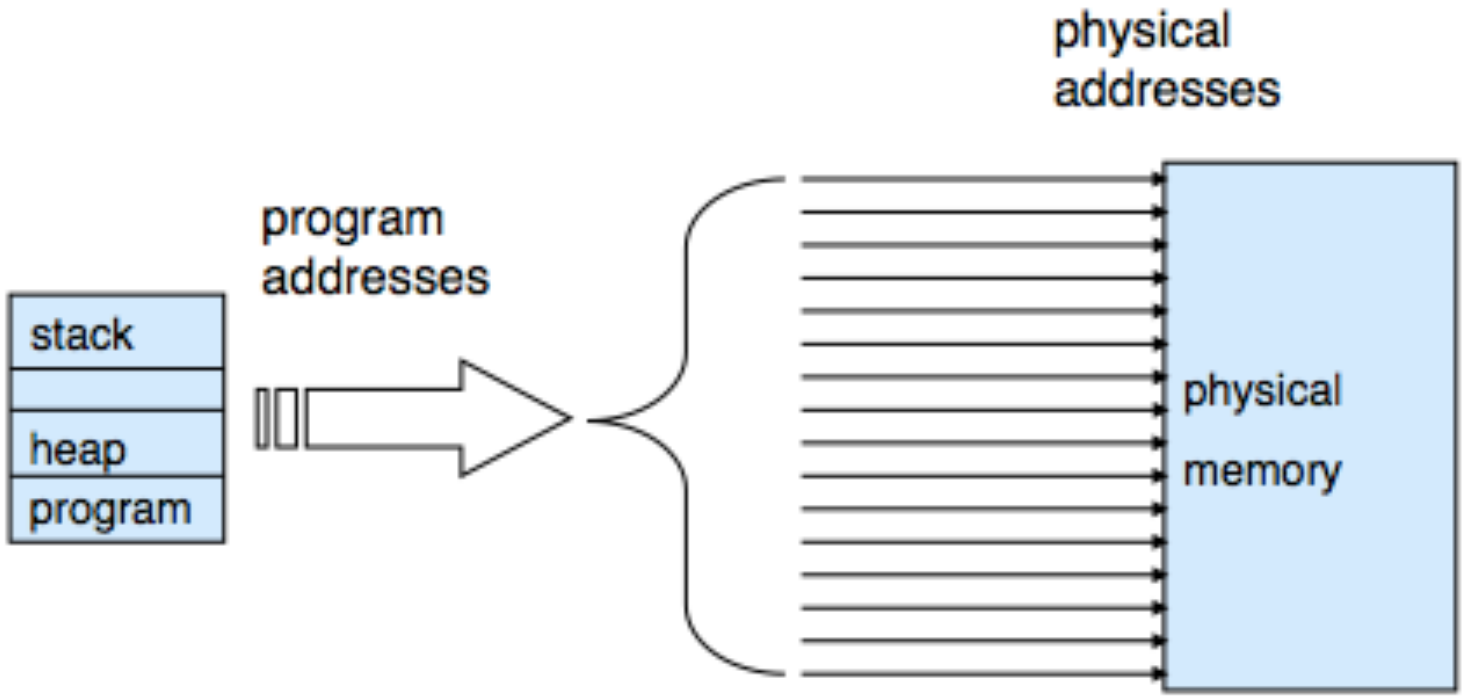
- Registers are generally accessible within one cycle of the CPU clock. The same cannot be said of main memory, which is accessed via a transaction on the memory bus.

- Memory access may take many clock cycles to complete, in which case the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing.

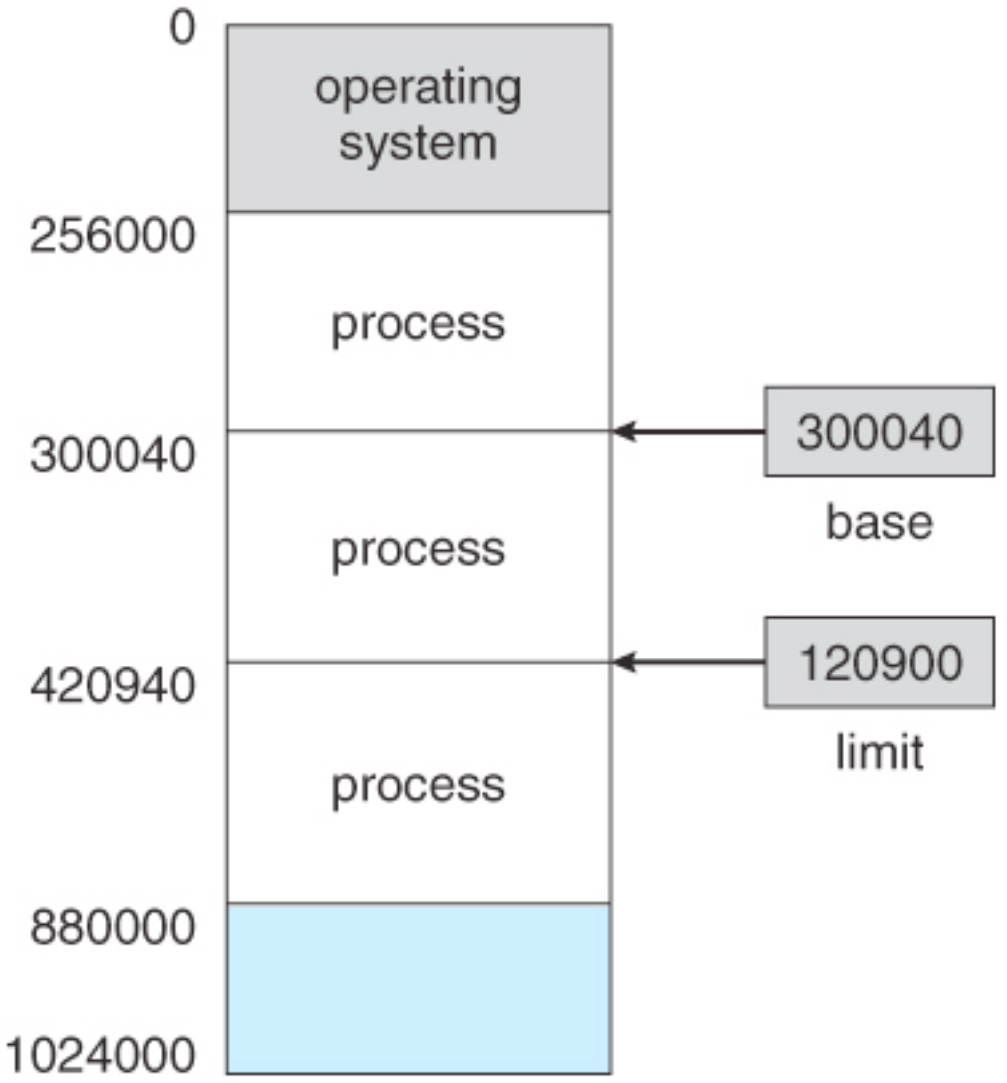
Address Binding

- Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory.
- **The process of associating program instructions and data to physical memory addresses is called address binding, or relocation**.
- Addresses in the source program are generally relocatable addresses generated by the compiler.
- Loader will in turn bind the relocatable addresses to absolute addresses.
- Each binding is a mapping from one address space to another.

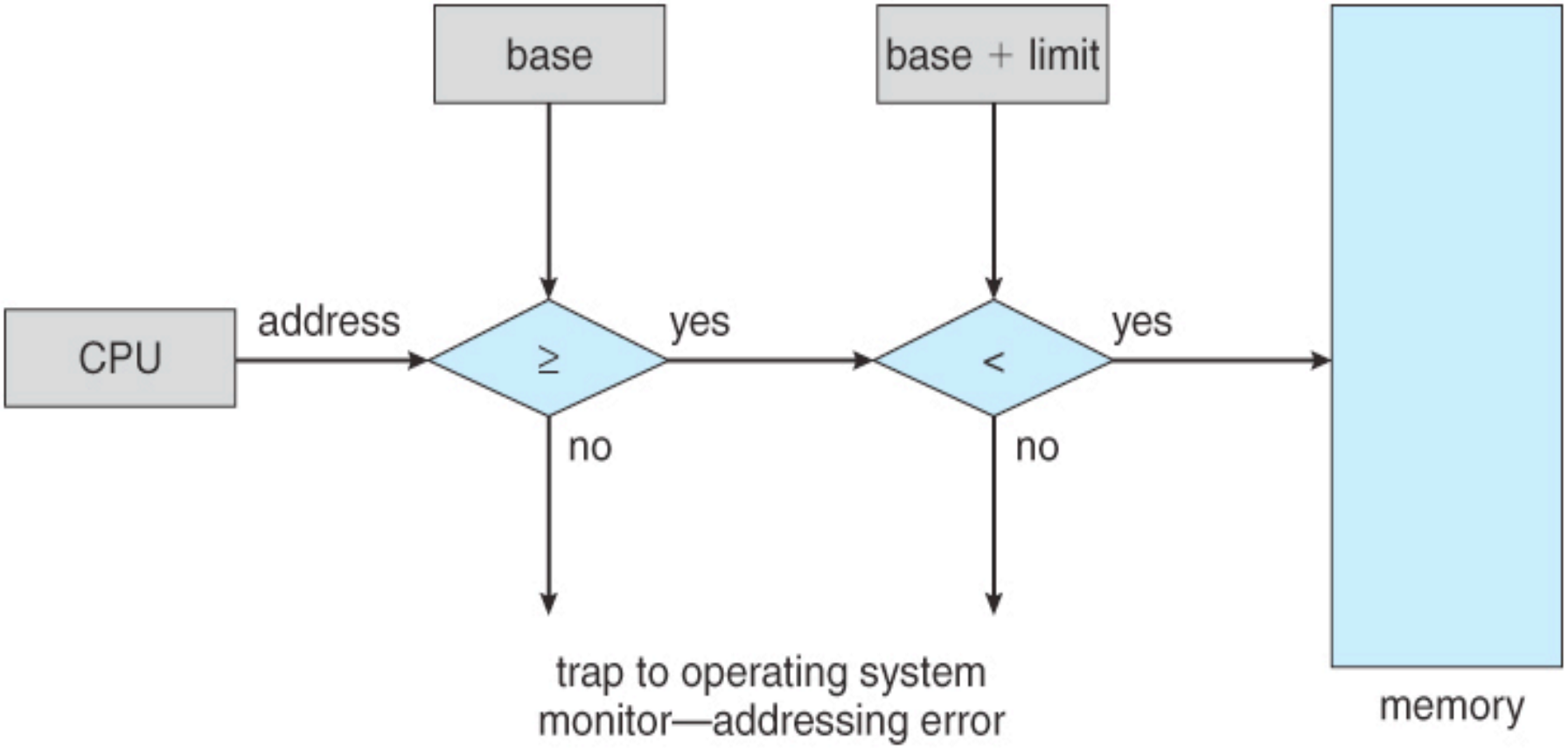
Address Binding (cntd)



- User processes must be restricted so that they only access memory locations that "belong" to that particular process.
- This is usually implemented using a **base register** and a **limit register** for each process.
- Every memory access made by a user process is checked against these two registers, and if a memory access is attempted outside the valid range, then a fatal error is generated.



Changing the contents of the base and limit registers is a privileged activity, allowed only to the OS kernel.



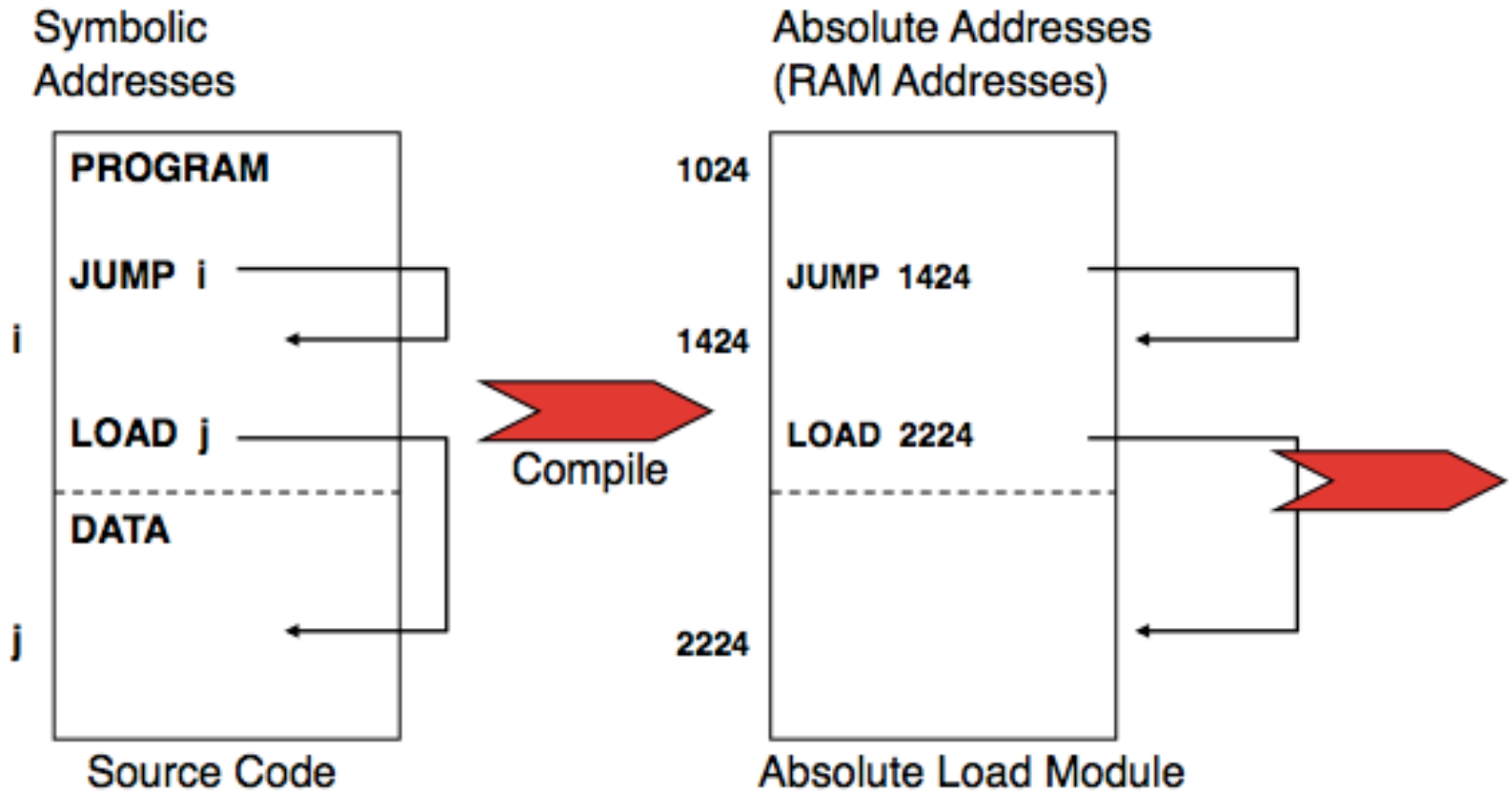
Address Binding Techniques

- Classically, the binding of program addresses can be done at any step along the way:

- **Compile time**: The compiler translates symbolic addresses to absolute addresses. If you know at compile time where the process will reside in memory, then absolute addresses can be generated (Static).

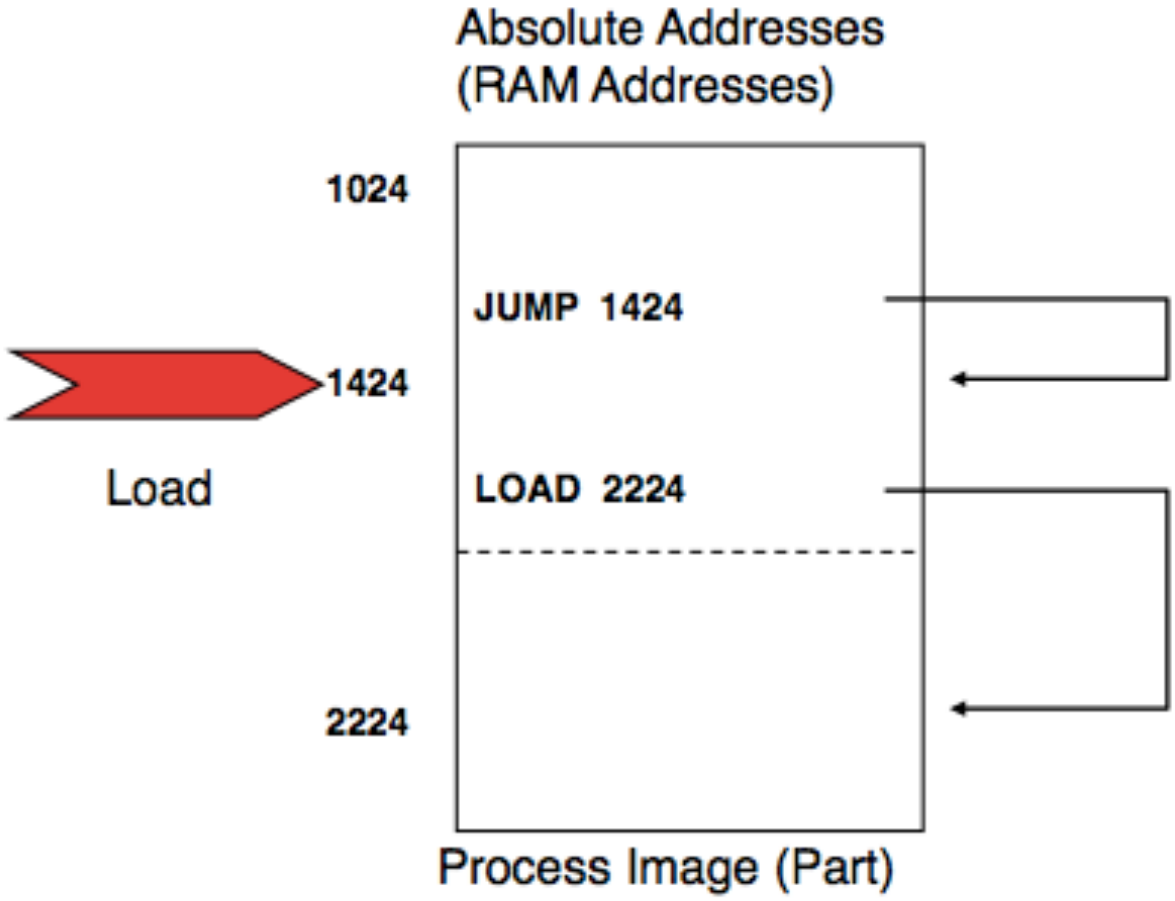
- For example, if we know that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.

Binding at Compile Time



The CPU generates the absolute addresses

Binding at Compile Time (Cont.)



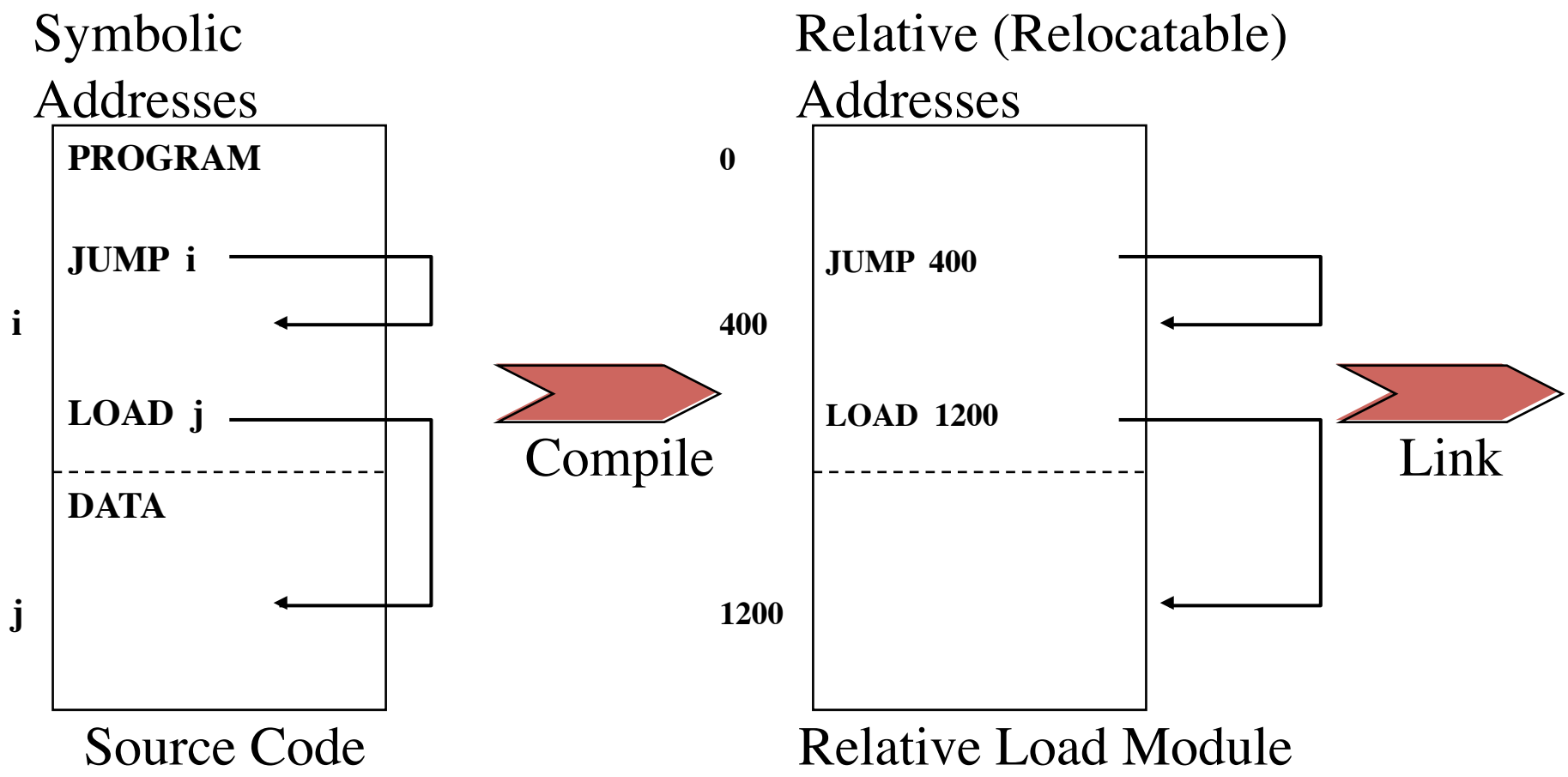
Address Binding (cntd)

- **Load time**: If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code**. In this case, final binding is delayed until load time.

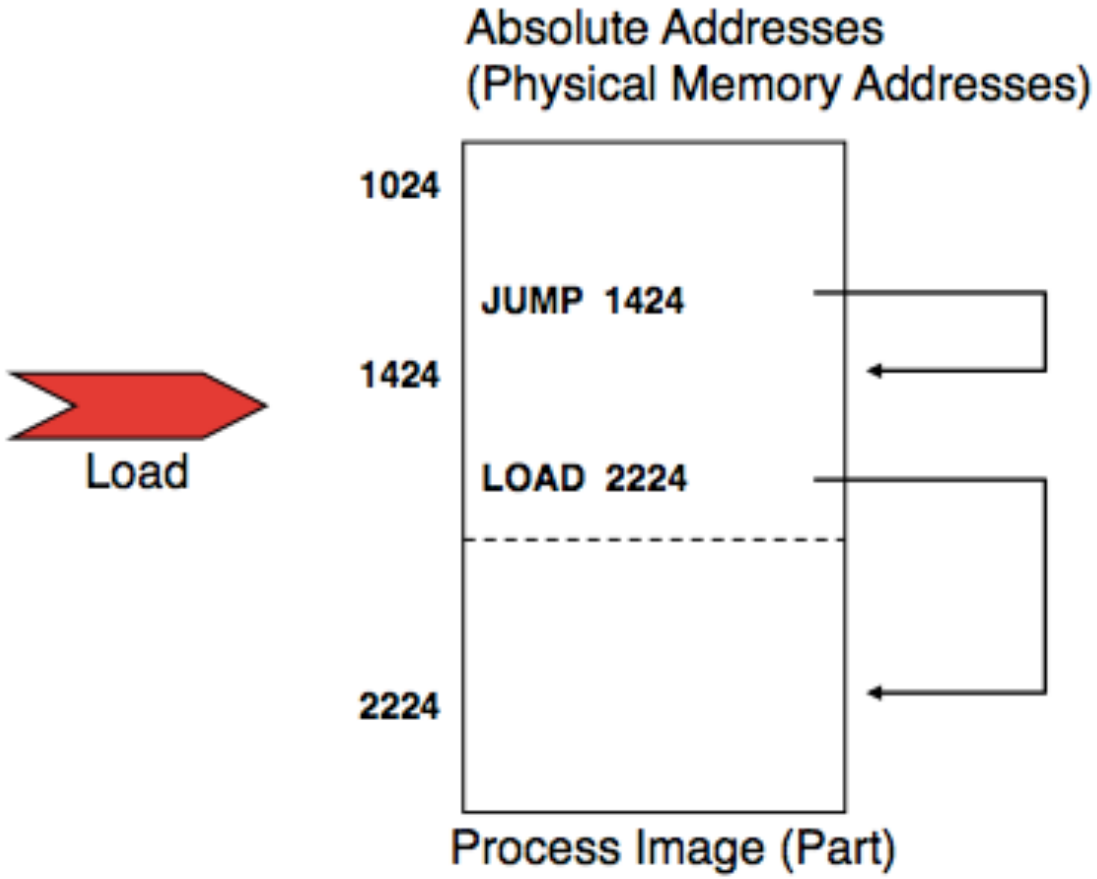
- It references addresses relative to the start of the program.

- If the starting address changes in RAM, then the program must be reloaded but not recompiled.

Binding at Load Time



Binding at Load Time (Cont.)



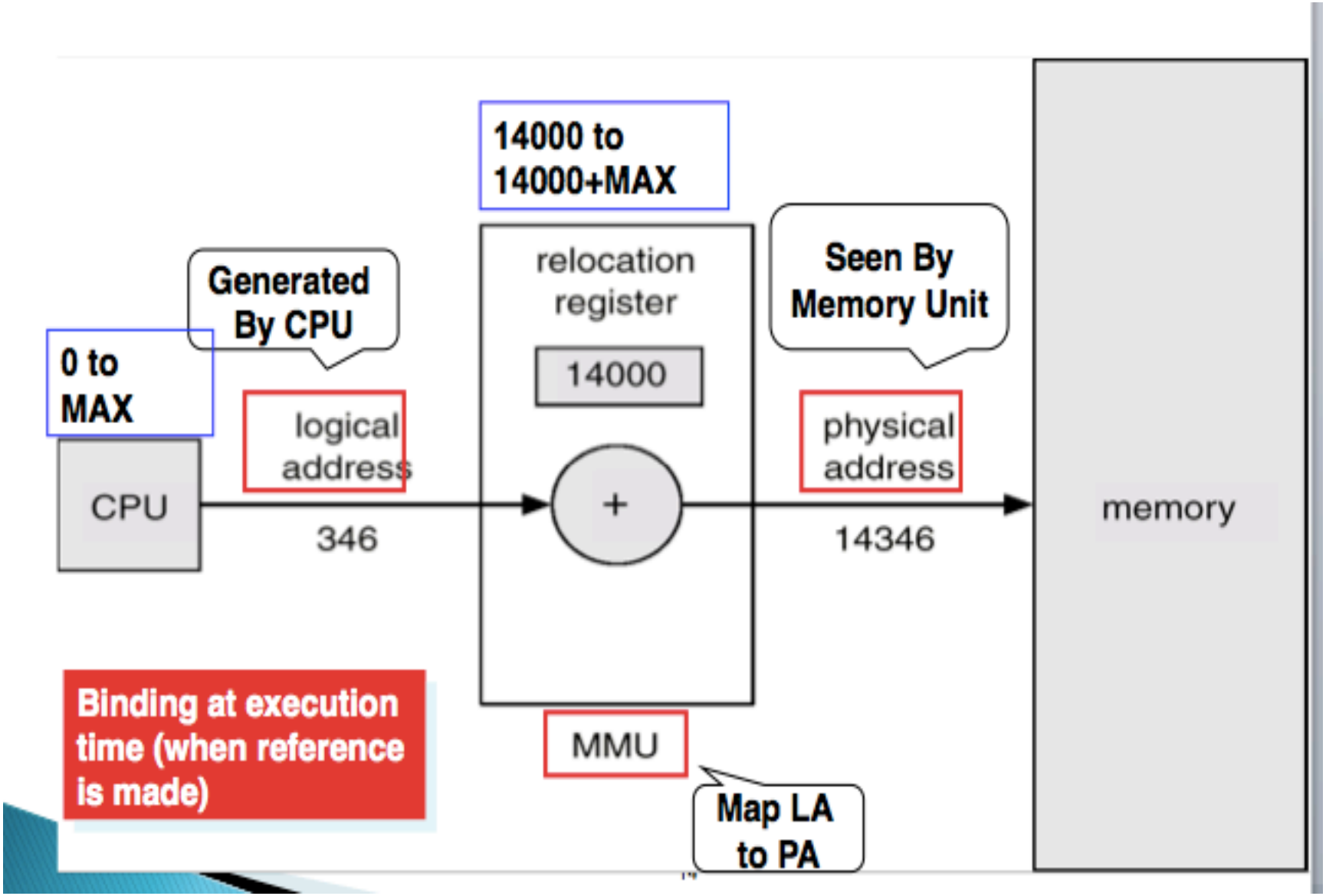
The CPU generates the absolute addresses

Address Binding (cntd)

- **Execution time**: If the process can be moved **during its execution** from one memory segment to another, then binding must be delayed until run time.

- Special hardware must be available for this scheme to work. Most general-purpose operating systems use this method.

Dynamic Relocation Using a Relocation Register



Logical vs. Physical Address Space

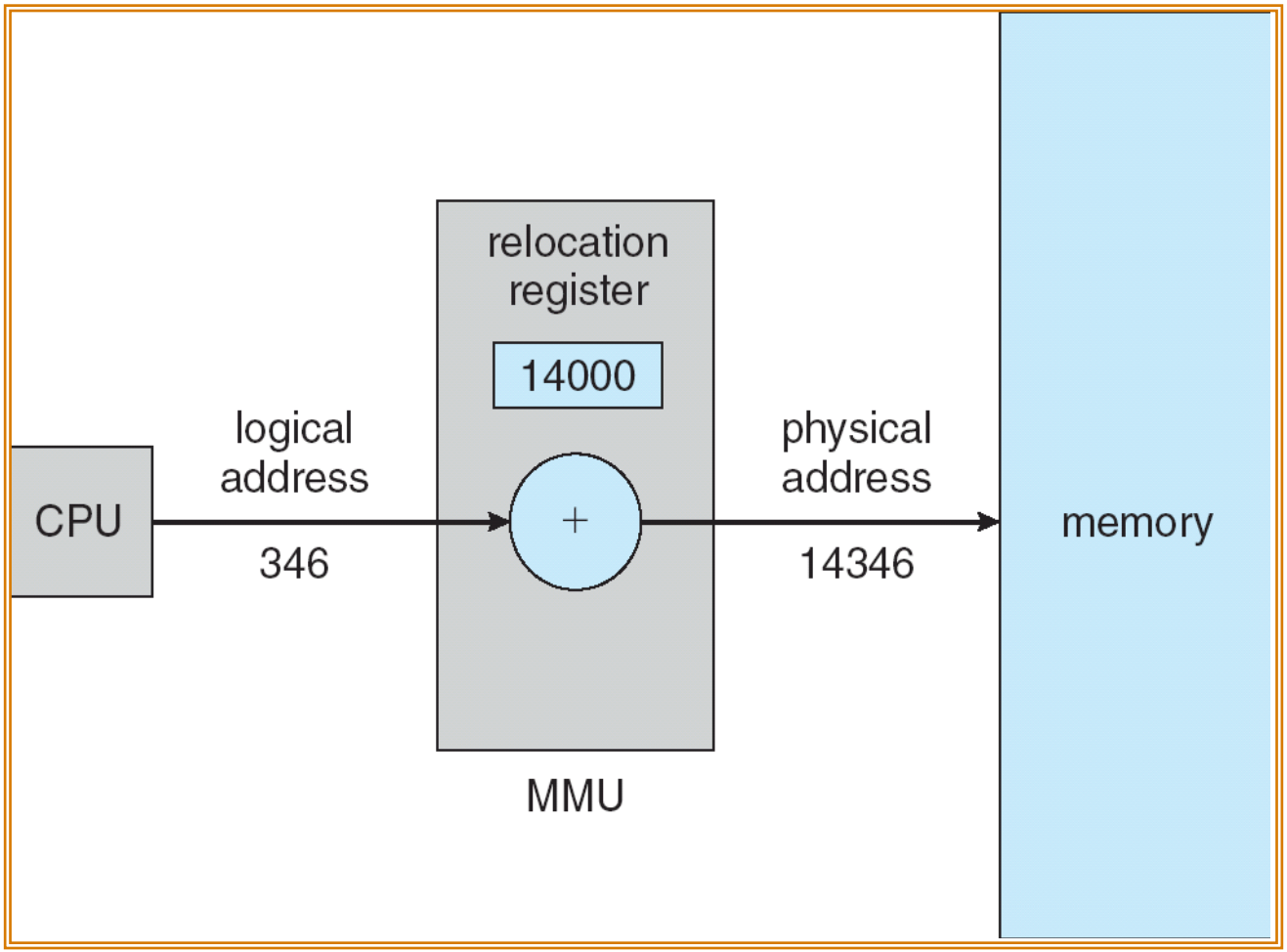
- **Logical address** – generated by the CPU; also referred to as *virtual address*
- **Physical address** – address seen by the memory unit. Actual addresses of RAM

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

Memory-Management Unit (MMU)

- Hardware device that maps virtual/logical to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

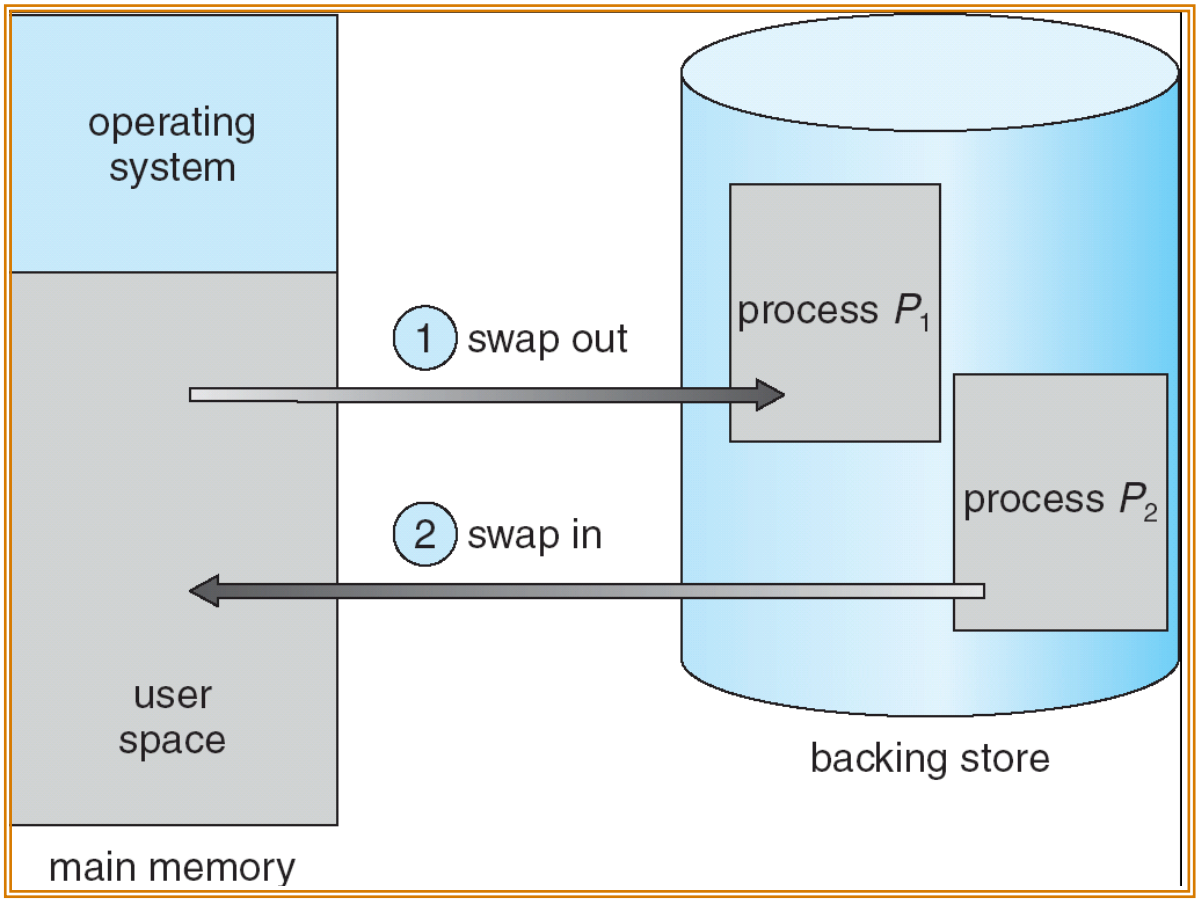
Dynamic relocation using a relocation register



Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed



Swapping (cntd)

Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.

- A modification of swapping is used in many versions of UNIX. Swapping is normally disabled but will start if many processes are running and are using a threshold amount of memory.

- Swapping is again halted when the load on the system is reduced.

Contiguous Memory Allocation

- Main memory must accommodate both the operating system and the various user processes.
- Memory is usually divided into **two partitions**: one for the resident operating system and one for the user processes.
- Usually the operating system is in low memory.
- We will now discuss how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.
- In this contiguous memory allocation, each process is contained in a single contiguous section of memory.

Memory Allocation

- One of the simplest methods for allocating memory is to divide memory into several ***fixed-sized partitions***.

- Each partition may contain exactly one process.

- Thus, the degree of multiprogramming is bound by the number of partitions.

- In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition.

- When the process terminates, the partition becomes available for another process. This method is no longer in use.

Multiprogramming With Variable Tasks (MVT)

- In the this *scheme*, the operating system keeps a table indicating which parts of memory are available and which are occupied.

- Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**.

- When a process arrives and needs memory, we search for a hole large enough for this process. If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.

- At any given time, we have a list of available block sizes and the input queue.

Multiprogramming With Variable Tasks (MVT)

- Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied—that is, no available block of memory (or hole) is large enough to hold that process.

- The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

- At any given time we have a set of holes of various sizes scattered throughout memory.

- When a process terminates, it releases its block of memory, which is then placed back in the set of holes.

Multiprogramming With Variable Tasks (MVT)

- If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

- The system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

- This procedure is a particular instance of the general **dynamic storage allocation problem**, which concerns how to satisfy a request of size n from a list of free holes.

- There are many solutions to this problem. The **first-fit, best-fit** and **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

Best Fit Strategy

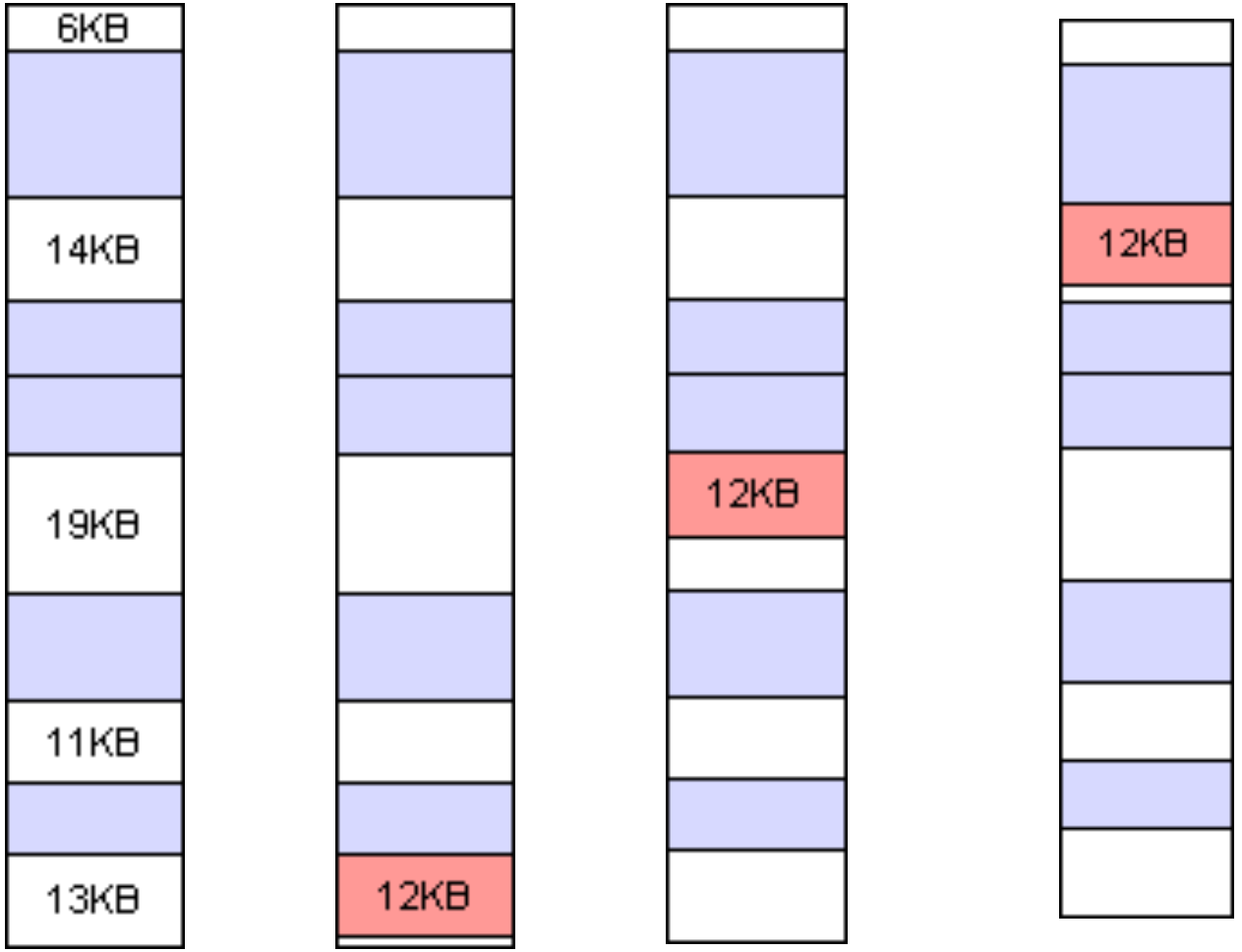
- Best fit: The allocator places a process in the smallest block of unallocated memory in which it will fit. For example, suppose a process requests 12KB of memory and the memory manager currently has a list of unallocated blocks of 6KB, 14KB, 19KB, 11KB, and 13KB blocks. The best-fit strategy will allocate 12KB of the 13KB block to the process.

Worst Fit Strategy

- **Worst fit**: The memory manager places a process in the largest block of unallocated memory available. The idea is that this placement will create the largest hold after the allocations, thus increasing the possibility that, compared to best fit, another process can use the remaining space. Using the same example as above, worst fit will allocate 12KB of the 19KB block to the process, leaving a 7KB block for future use.

First Fit Strategy

- **First fit**: There may be many holes in the memory, so the operating system, *to reduce the amount of time* it spends analyzing the available spaces, begins at the start of primary memory and allocates memory from the first hole it encounters large enough to satisfy the request. Using the same example as above, first fit will allocate 12KB of the 14KB block to the process.


6KB
14KB
19KB
11KB
13KB
12KB
12KB
12KB
Primary Memory
Best Fit
Worst Fit
First Fit

Fragmentation

- In the previous diagram above that the Best fit and First fit strategies both leave a tiny segment of memory unallocated just beyond the new process.

- Since the amount of memory is small, it is not likely that any new processes can be loaded here.

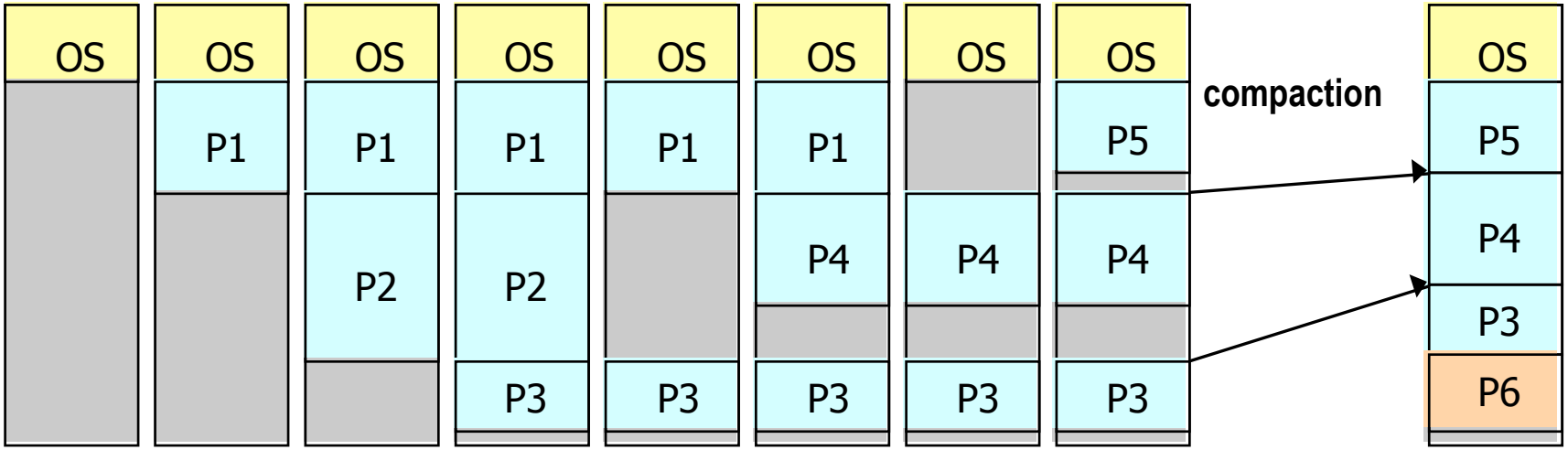
- This condition of splitting primary memory into segments as the memory is allocated and deallocated is known as **fragmentation**.

- The Worst fit strategy attempts to reduce the problem of fragmentation by allocating the largest fragments to new processes.

External Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous

- 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

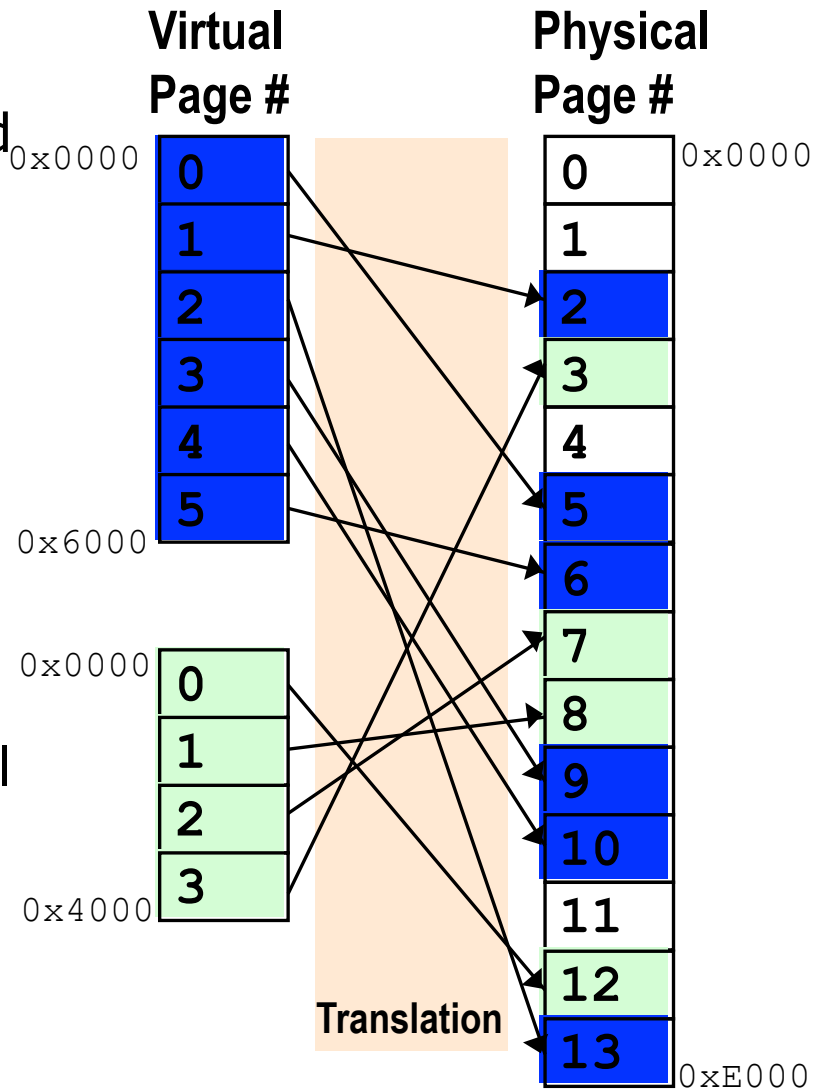


Internal Fragmentation

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Paging

- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 4096 bytes and 8192 bytes)
- Divide logical memory into blocks of same size called **pages**.
- **Any virtual page can be located at any physical page**
- Translation box converts from virtual pages to physical pages



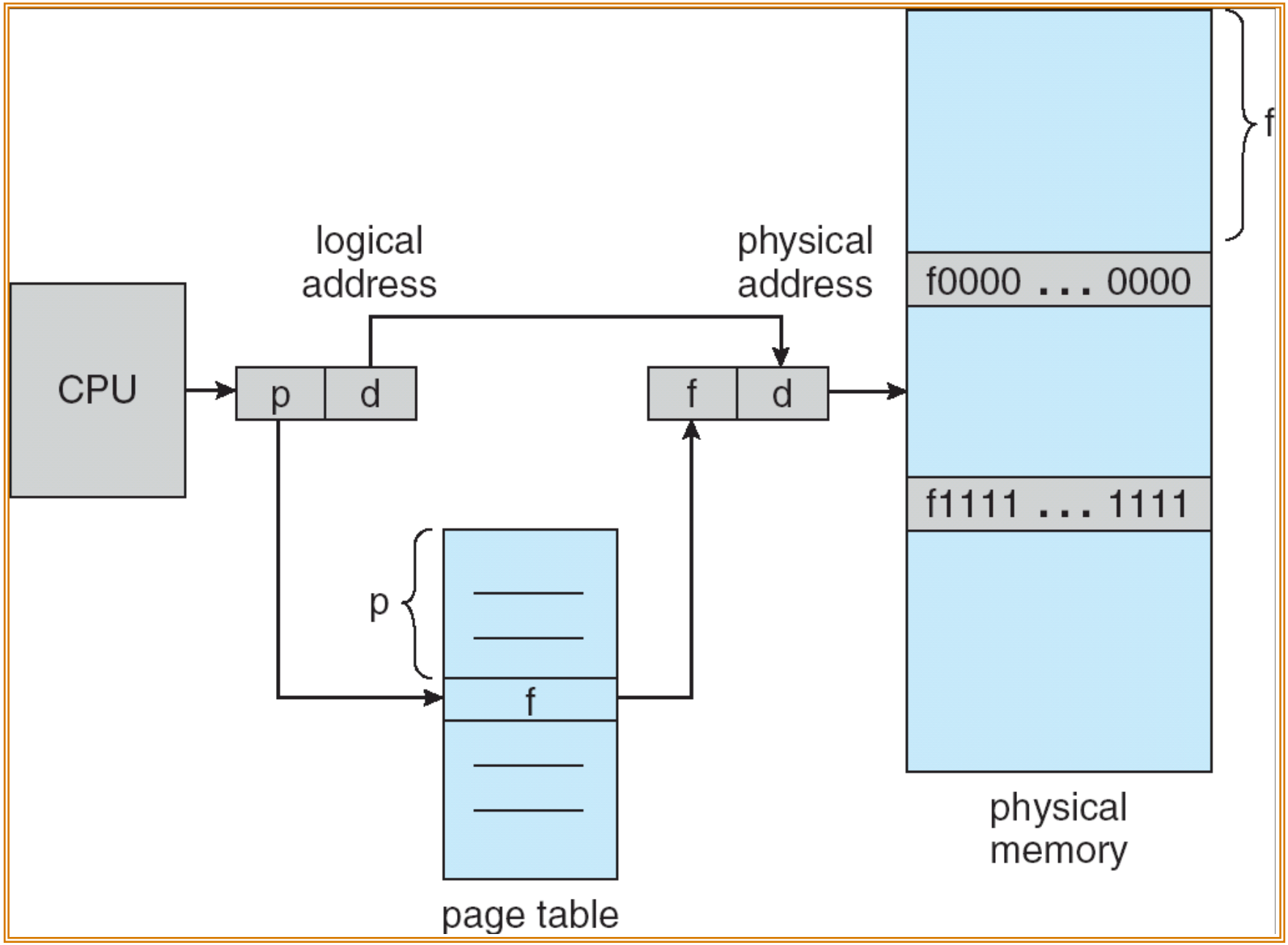
Address Translation Scheme

- Address generated by CPU is divided into:

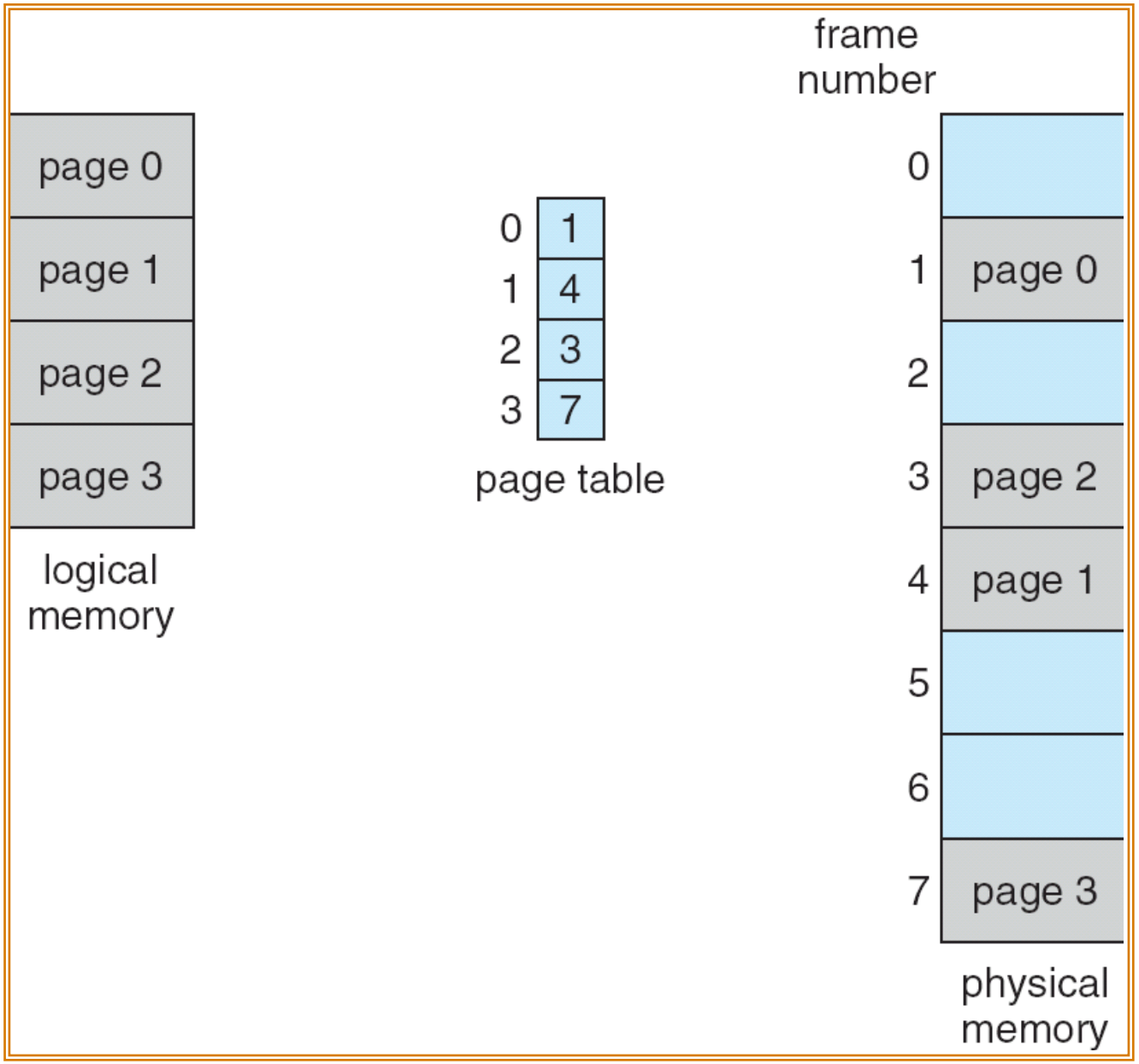
  - ***Page number (p)*** – used as an index into a *page table* which contains base address of each page in physical memory

  - ***Page offset (d)*** – combined with base address to define the physical memory address that is sent to the memory unit

Address Translation Scheme

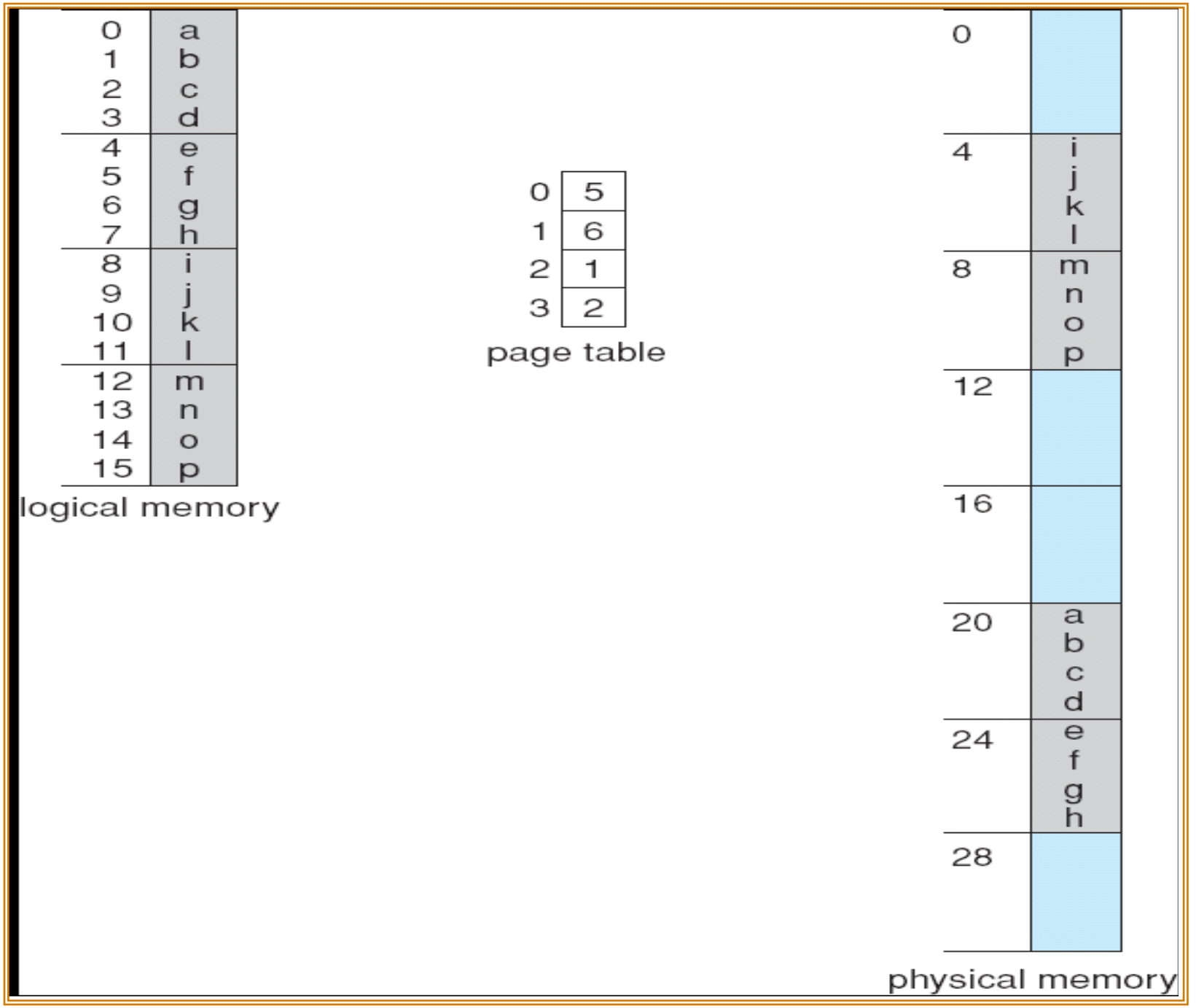


Paging Example



Page Size= 4bytes

Physical Memory= 32bytes



Paging

- In paging we have **no external fragmentation**. Any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation?

- Today, pages typically are between 4 KB and 8 KB in size, and some systems support even larger page sizes.

- Important aspect of paging is the clear separation between the user's view of memory and the actual physical memory.

- The user program views memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs.

Paging Example

- Given the following logical address find the corresponding physical address.

- The memory management scheme is paging with 16-bit addressing and frame size of 1024.

- Logical address: 0000101011110000

Page table

Page number	Frame number
0	101011
1	111100
2	110011
3	011010
4	010101

Paging Example(cntd)

- Because the frame size is 1024, which is 2^{10}, then the 10 rightmost bits of the logical address are used to store the offset within the page.

- Logical address is 16 bits long and therefore the remaining leftmost 6 bits are used to indicate the page number.

- So the logical address is divided as follows:

Page number	Offset within page
000010	1011110000

- That is, the binary page number is 000010 which is 2 in decimal.

Paging Example(cntd)

- Get the information on page 2 from the page table:

Page number	Frame number
2	110011

- So the physical address is the combination of the frame number 110011 and the offset 1011110000.That is

Frame number	offset
110011	1011110000
i.e. 1100111011110000	

Hardware Support

- Most operating systems allocate a page table for each process.
- A pointer to the page table is stored with the other register values in the process control block.
- When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table.
- The hardware implementation of the page table can be done in several ways.
- In the simplest case, the page table is implemented as a **set of dedicated registers**.
- These registers should be built with very high-speed logic to make the paging-address translation efficient.

Hardware Support (cntd)

- The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries).

- Most OS's, however, allow the page table to be very large (for example, 1 million entries).

- For such machines, the use of fast registers to implement the page table is *not feasible*. Rather, the page table is kept in main memory, and a **page-table base register (PTBR)** points to the page table.

- Changing page tables requires changing only this one register, substantially reducing context-switch time.

- The problem with this approach is the time required to access a user memory location. If we want to access location, we must first index into the page table, using the value in the PTBR offset by the page number.

Hardware Support (cntd)

- With this scheme, two memory accesses are needed to access a byte (one for the page-table entry, one for the actual data).

- Thus, memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances.

- The standard solution to this problem is to use a special, small, fastlookup hardware cache, called a **translation look-aside buffer (TLB)**.

Translation Look-Aside Buffer (TLB).

- The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value.

- TLB— act as cache for the page table

- Whenever a program performs a memory reference, the virtual address sent to the TLB to determine if it contains a translation for the address

- Yes— a **TLB Hit** occurs and the TLB returns the physical address of the data, and the memory reference continues

- No—, a **TLB miss** occurs, and the system searches the page table for the translation

- The search is fast; the hardware, however, is expensive. Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.

Translation Look-Aside Buffer (TLB).

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process.

- When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page.

- If the ASIDs do not match, the attempt is treated as a TLB miss.

- In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously.

Translation Look-Aside Buffer (TLB).

- If the TLB does not support separate ASIDs, then every time a new page table is selected (for instance, with each context switch), the TLB must be **flushed** (or erased) to ensure that the next executing process does not use the wrong translation information.

- Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

- The percentage of times that a particular page number is found in the TLB is called the **hit ratio**.

- An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time.

Effective Access Time

- If it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory, then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB.

- If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds.

- To find the effective memory-access time, we weight each case by its probability:

effective access time = 0.80 x 120 + 0.20 x 220

= 140 nanoseconds.

- In this example, we suffer a 40-percent slowdown in memory-access time (from 100 to 140 nanoseconds).

Paging hardware with TLB