Operating Systems ICS 431

Weeks 9-10

Main Memory Management

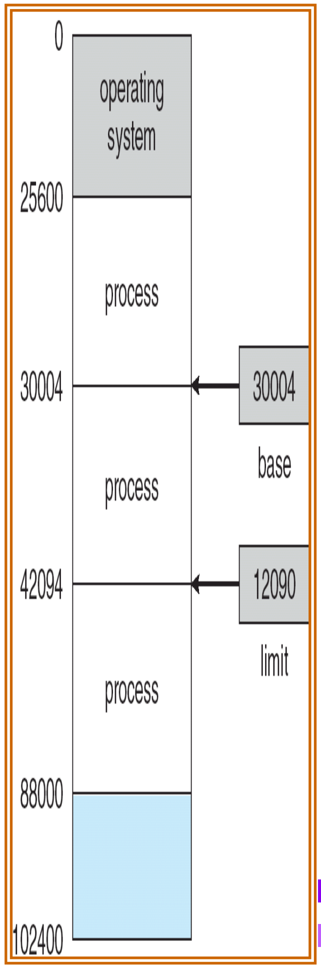
Dr. Tarek Helmy El-Basuny

Main Memory Management

- Main duties of the Memory Management Unit,
- Memory Management Requirements:
  - Relocation, Protection, Sharing, Logical Organization, Physical Organization
- What is Logical address? What is physical address?
  - **Logical/Virtual address:** Generated by the CPU and reflects (A process's view of its own memory)
  - **Physical address:** Address seen by the memory unit (stored in the Memory Address Register)
- Address Binding/Mapping (Logical to Physical Address Translation),
- Processing Steps of a User's Program,
- The dynamic loading & linking and their advantages,
- Swapping, Swapping affect on the context switch, Swapping time and Quantum Time,
- Contiguous and Non-contiguous Allocation with their advantages and disadvantages,
- Memory Partitions Allocation: equal and un-equal size partitioning of the main memory,
- Dynamic Partitioning: partition's size and numbers will be dynamic,
- Fragmentation: Internal and External,
- How to minimize the Fragmentation?
  - **Compaction, Paging, Segmentation, Segmentation with Paging**

Memory Management Unit (MMU)

- We have seen how the processes share the CPU. How?
- Processes also share the physical/main memory. We want to know How?
- As a fact, the OS kernel occupies a part of the main memory. For what?
- The rest of the main memory will be shared by running processes.
- **What happen if only few processes are loaded into the main memory?**
  - The multiprocessing level will be less and resources will not be maximally used.
- Hence, the OS goal is to allocate the main memory efficiently to processes to pack as many processes as possible to increase the concurrency level and to maximize resources utilization.
- It is recommended for the OS to partially (not Fully) load the processes into the main memory to increase the concurrency level.
- Which pages/segments of the process should be loaded?
  - Most frequently used part of the process.



Memory Management Unit (MMU)

- The MMU is responsible for the following activities:
  - Keep track of which parts of memory are currently being used and by which processes.
  - Decide which process to load when memory space becomes available.
    - A policy decision
  - Maintain mappings from physical to virtual memory and vise versa.
    - Through page tables
  - Decide how much memory (pages/segments) to allocate to each process.
    - A policy decision
  - Decide when to remove/swap out a process from the main memory
    - A policy decision.

Memory Management Requirements

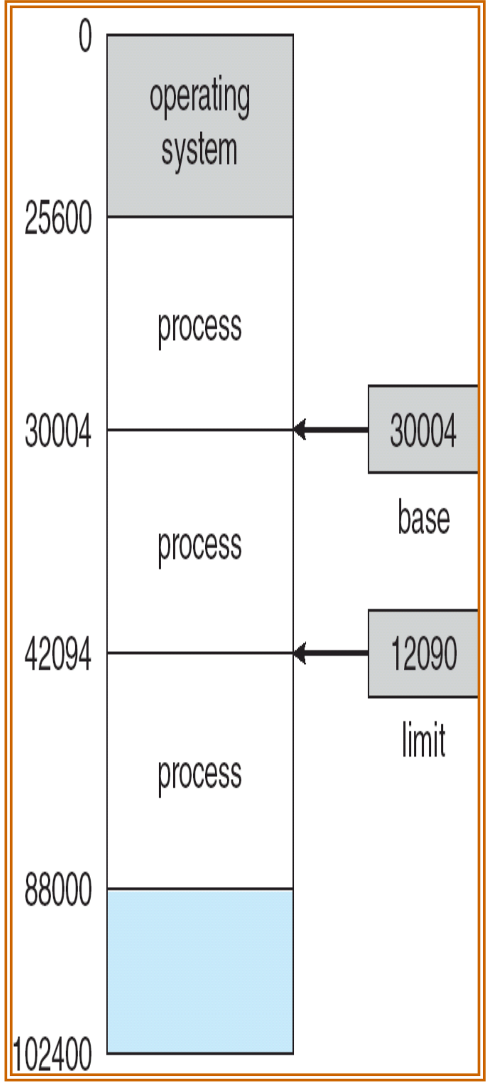
- **Relocation**
  - MMU decides where the process will be placed in memory when it is executed, the programmer does not know about that.
  - A process may be relocated in main memory due to swapping.
    - Swapping enables the OS to have a larger pool of ready-to-execute processes.
  - Memory references in code (for both instructions and data) must be translated to physical memory address.
- **Protection**
  - Processes should not be able to reference memory locations of each other without permission.
  - Address references must be checked/validated at run time by hardware.
    - Impossible to check addresses at compile time in programs since the program could be relocated.
- **Sharing**
  - OS may allow several processes to access a common portion of main memory without compromising the protection.
  - Cooperating processes may need to share access to the same data structure.
    - Better to allow each process to access the same copy of the global data rather than having their own separate copies.

Memory Management Requirements

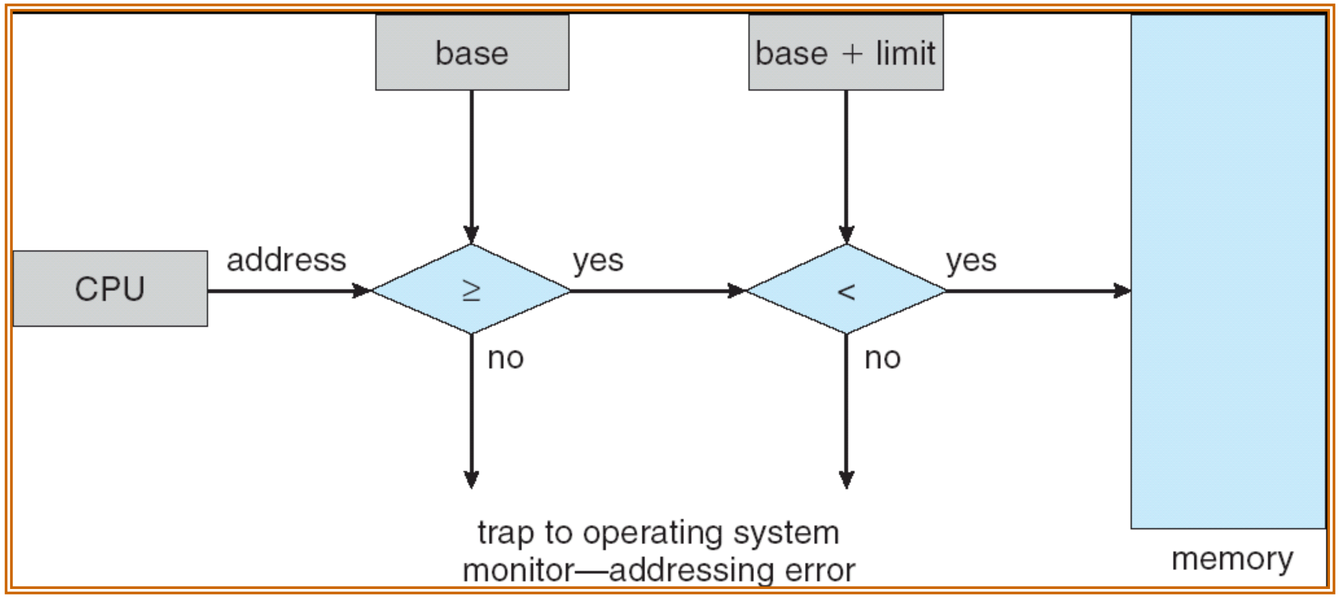
- **Logical Organization**
  - Users write programs in modules with different characteristics
    - Instruction modules are execute-only.
    - Data modules are either read-only or read/write.
    - Some data modules are private and others are public.
  - To effectively deal with user's programs, the OS and HW should support a basic form of modules to provide the required protection and sharing.
- **Physical Organization**
  - Auxiliary memory is the long term store for programs and data .
  - Main memory holds processes and data currently in use.
  - Main Memory may not be available for the whole process plus its data.
    - Swapping allows various modules to be assigned the same region of memory.
  - Moving information between these two levels of memory is a major concern of memory management unit.

Process's Memory Protection

- How does the OS Protect processes from each other?
- Hardware-support scheme
  - Relocation (Base) register contains value of smallest physical address.
  - Limit register contains range of logical addresses – each logical address must be less than the limit register.
- **More advantages of Base and Limit Registers.**
  - The limit register provides an effective way to allow the OS size to change dynamically.
  - As the OS contains some transient code and buffer space for device drivers, if the device driver is not commonly used, **its code can be moved out and give a space for other processes**.
  - The base register allows the process to be relocated by changing the content of the base register.
  - The limit register provides an effective way to allow the process size to change dynamically so that it can grow or shrink.



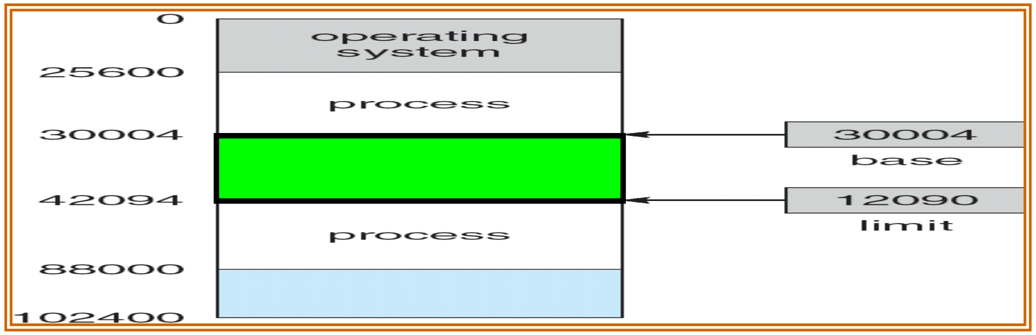
HW address protection with base and limit registers

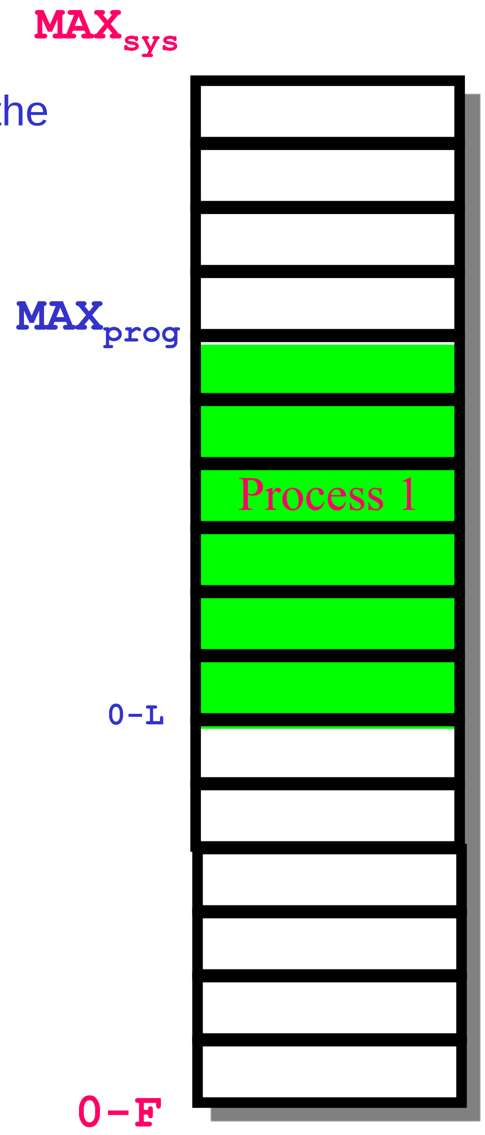


A Base and a limit resister define a logical address space of a process

Physical and Virtual Address Space

- **Physical address space:** The address space supported by the hardware "the address loaded into the memory address register".
- The physical address refers to a specific location in memory, that the hardware can support.
  - Starting at address **0-F**, going to address **MAX$_{sys}$**
- **Logical/Virtual address space:** A process's view of its own memory "generated by the CPU".
- A virtual address refers to a specific location within a process.
  - Address relative to start of process's address space
  - Starting at address **0-L**, going to address **MAX$_{prog}$**

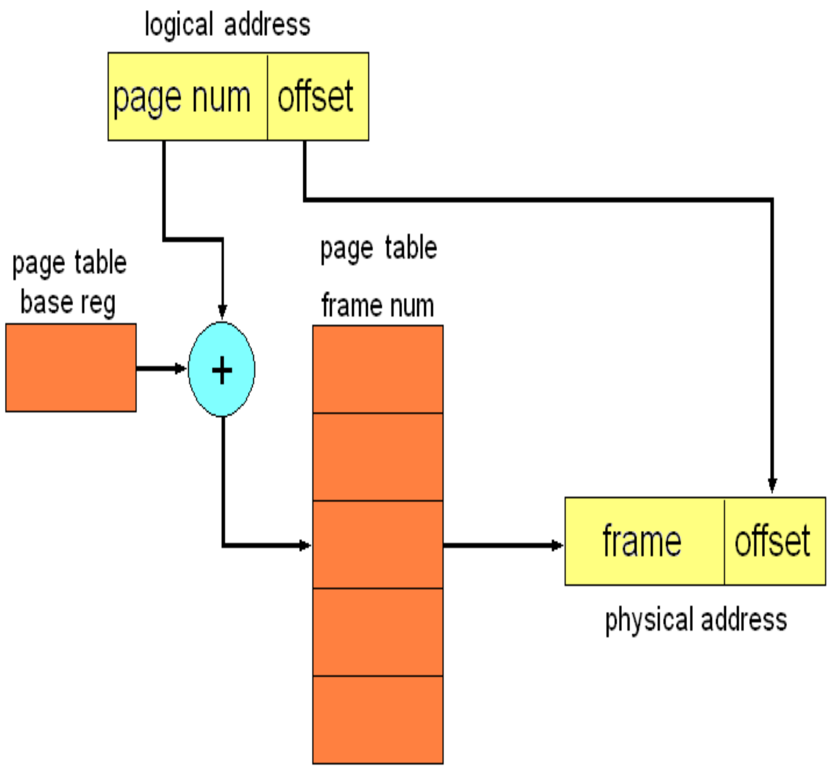


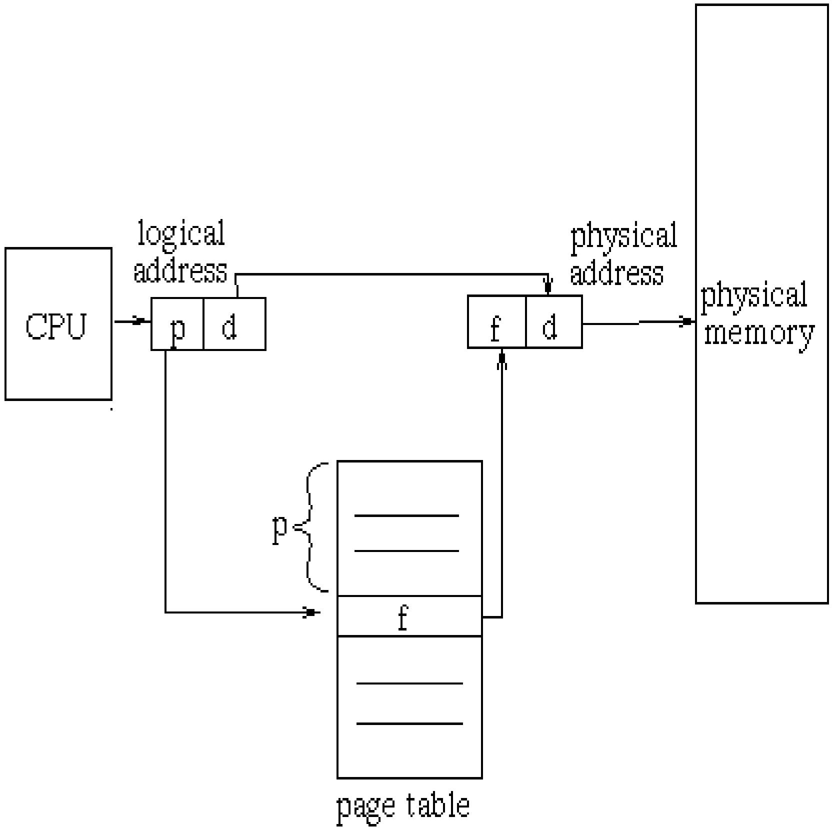


- **A Base and a Limit Registers Define a Logical Address Space**
- CPU dispatcher loads the base and limit registers with the correct values (stored in PCB) as part of the context switch.

Logical to Physical Address Translation

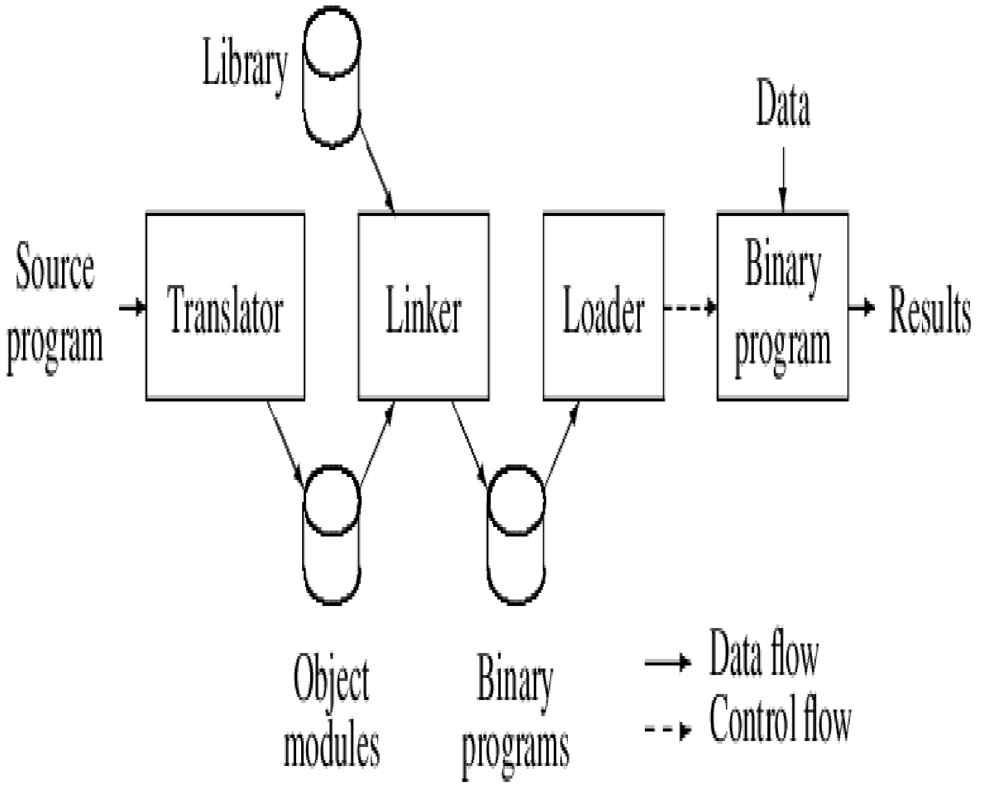
- The run time mapping from Logical to Physical addresses is done by the MMU, many methods to do such mapping, i.e. ”contiguous, paging, segments”

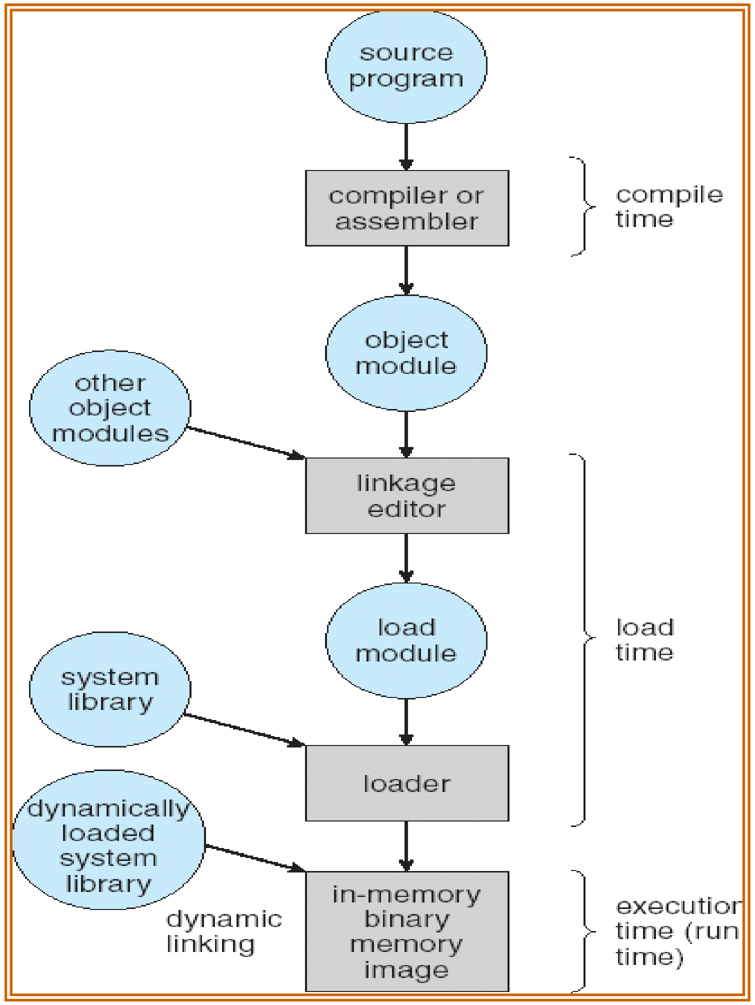




Processing Steps of a User's Program

- User's programs go through many steps (compile, link, load) before running.

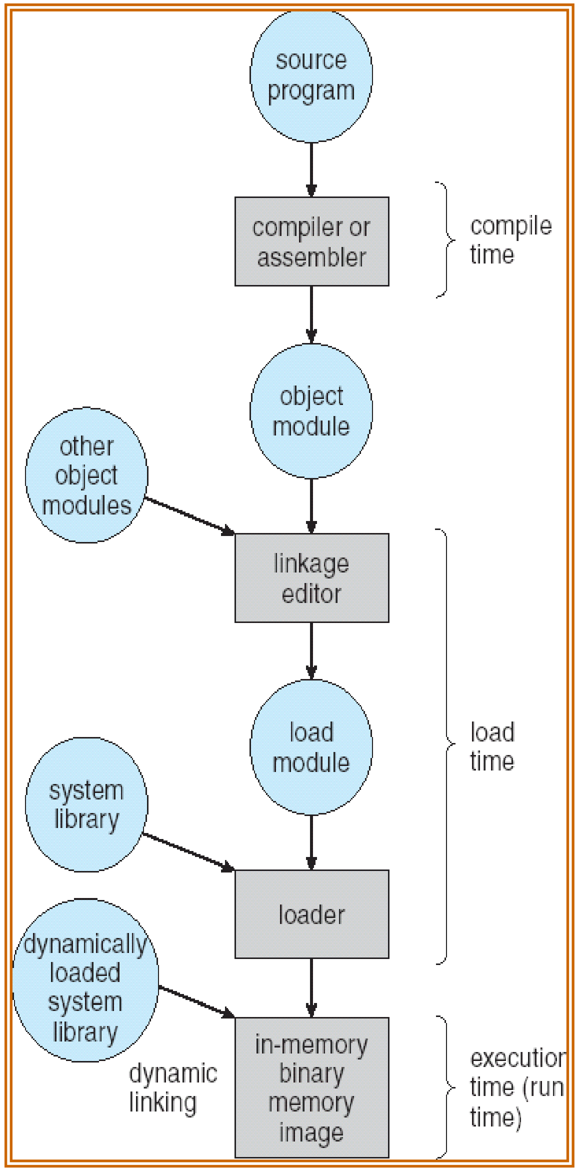




Linking and Loading Steps

- Linking: Different parts of a process are linked together in order to make them a runnable entity. This can be done:
  - Before execution (Static linking)
  - On demand during execution (Dynamic)

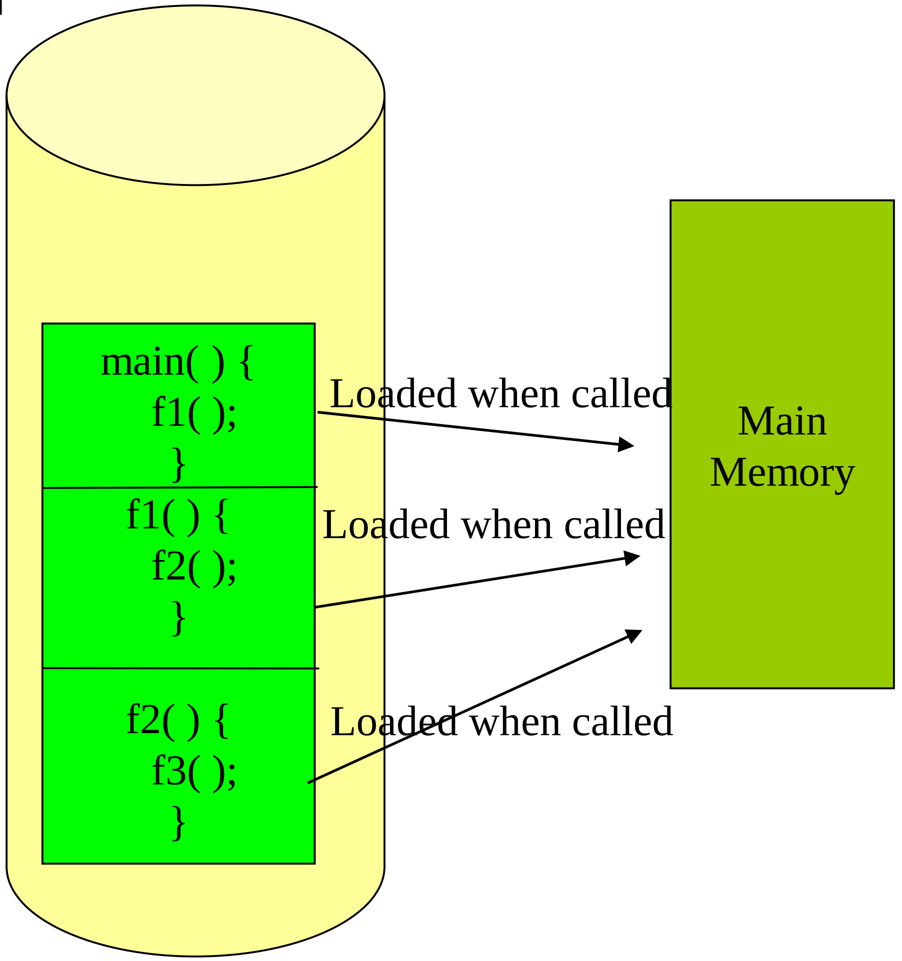
- Loading: Load the linked parts into the main memory (ready to execute). May involve address translation.
  - Absolute/static, or re-locatable/dynamic
- Aspects of Loading
  - Finding free memory for loading a process.
  - Could be contiguous or Non-contiguous/scattered.
  - Adjust addresses in the process (if required) once it is known where the process will be loaded.



Dynamic Linking

- Dynamic linking is the process of postponing the linking of a certain library routine till the execution time.
- This feature is used with programming language’s and system’s library.
- Without this feature, all processes on a system need to have a copy of their programming language’s libraries included in the executable image.
  - This wastes both the disk and main memory spaces.
- With dynamic linking, stub is included in the image for each library routine.
  - The Stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present.
  - Stub replaces itself with the address of the routine, and executes the routine so that the next time the code segment is reached, the library routing executed directly, incurring the cost of dynamic linking.
- Dynamic linking is particularly useful for libraries.

Dynamic Loading



- With dynamic loading, a routine is not loaded until it is called.
- Dynamic loading allows better memory-space utilization.

Advantages of Dynamic Loading:

- Un-used routines never loaded into the memory.
- Useful when large amounts of code are needed to handle infrequently occurring cases, such as error routine.

Advantages of Dynamic Linking and Loading

- **Advantages of Dynamic Linking**
  - Supports portability: if the external module is an OS/language utility. Executable files can use another version of the external module of another OS/language without the need of being migrated.
  - Code sharing: The same external module needs to be loaded in main memory only once. Each process is linked to the same external module.
  - Saves memory and disk space.

Advantages of Dynamic Loading:

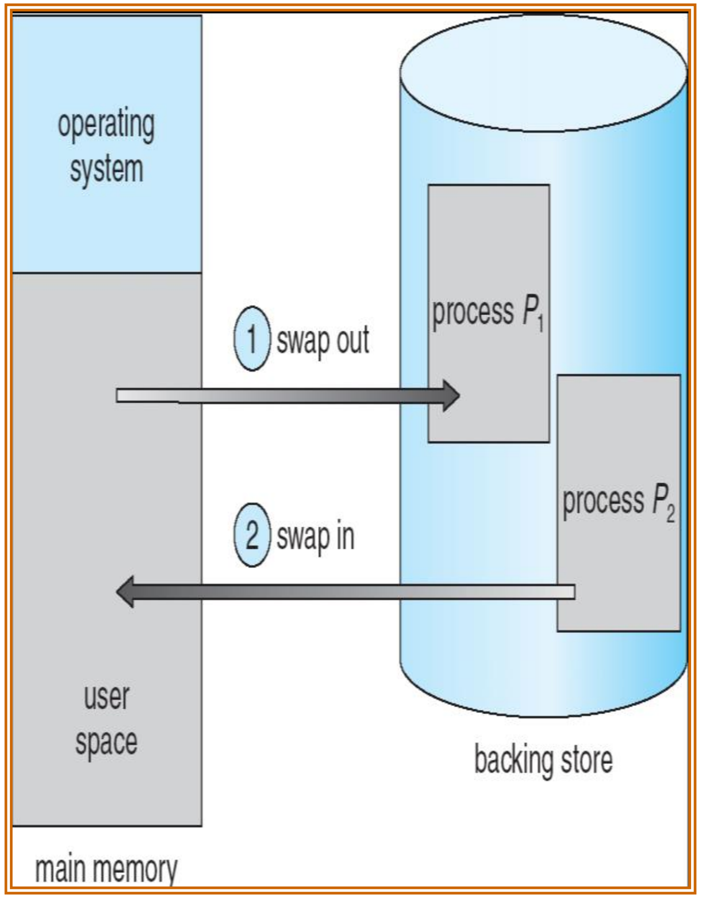
- Un-used routines never loaded into the memory.
- Useful when large amounts of code are needed to handle infrequently occurring cases, such as error routine.

Time of Address Binding/Mapping

- Address binding of processes and data (Logical address to Physical address mapping) can happen at three different stages:
  - **Compile time:** the process will be fixed to an absolute address. Recompilation is necessary if the starting location changes.
  - **Link and Load time:** Codes can be linked then loaded to any portion of memory. (Re-locatable code)
  - **Run time:** Code can be moved to any portion of memory during its execution.
  - **But it is recommended to be done during the run time to support the relocation of processes in the main memory.**

Swapping

- Swapping allows processes to be moved from the main memory to auxiliary memory and then back to the main memory again for execution.
- For instances, in RR scheduling policy, when the time quantum of a process expires, the memory manger starts to swap-out/roll-out the current process and swaps-in/roll-in the next process in turn.
- In priority-based scheduling algorithms; lower-priority process is swapped-out so higher-priority process can be swapped-in/rolled-in and executed.



Swapping and Context Switch

- With swapping, context switches will be more expensive because auxiliary memory (generally HD) is much slower than main memory.
- Swapping needs a backing store (HD) that is: Fast, large enough to accommodate copies of all memory images for all users, and must provide direct access to these memory images.

- **The effect of address binding on swapping**
  - If address binding has been done at linking or loading time then the **process cannot be moved** to different locations.
  - If address binding has been done at running-time: **It's possible to swap a process into a different memory space.**

Swapping & Execution/Quantum Time

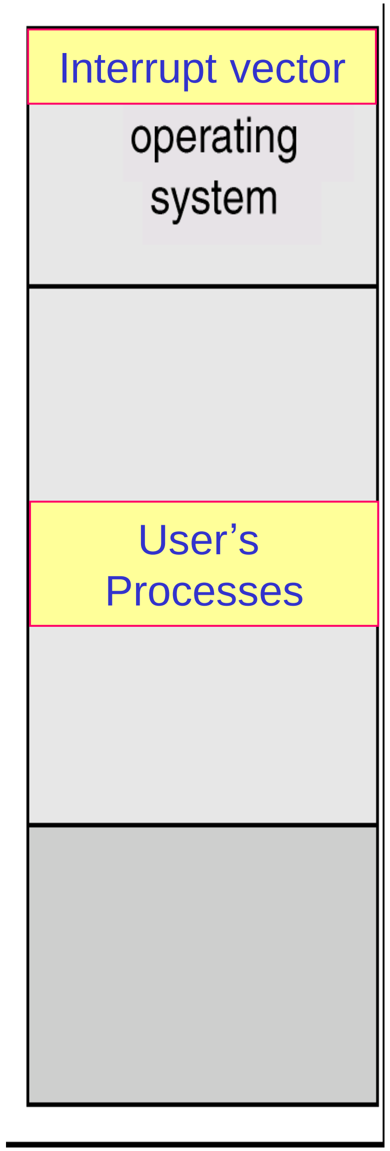
- Swapping is affected by transferring rate of the HD:
- The dispatcher checks if the scheduled process's pages are in memory or not.
- If not and there is no free memory space, then the dispatcher swaps out a process currently in and swaps in the desired process.
- The Context-switch time is fairly high, i.e. assume: a 1 MB user process,
- HD (backing store) transfer rate = 5 MB/Sec, avg. Latency= 8 ms
- Latency: The time it takes to position the proper sector under the R/W head.
  - Transferring 1MB process to/from memory takes 1MB/5MB per second + 8 millisecond = 208 ms
  - Swap in and out when context switch = 208 * 2= 416 ms
- For efficient CPU utilization, we want the process execution time to be long relative to the swap time. In RR algorithm we need time quantum be >416 ms
  - Major part of any swap time is the transfer time.
    - Total transfer time is directly proportional to the amount of memory swapped. Swapping 100 KB is faster than 1 MB.
    - Better to know exactly how much memory a process is using

Frequent Swapping causes Thrashing

- Swapping frequency should be minimized because:
  - It requires too much swapping time and provides too little execution for the processes, this is called thrashing.
- Thrashing: means the CPU time is used in swapping processes in or out without executing the processes themselves.
- **Thrashing** is a condition in which excessive swapping operations are taking place.
- Swapping is affected by other factors:
- If we want to swap out a process, it must be idle not on turn to run:
  - It may be waiting for I/O operation, etc.
- Modified versions of swapping are found on many OSs, i.e., UNIX & Windows.
  - UNIX: Start swapping if many processes were running and were using a threshold amount of memory.

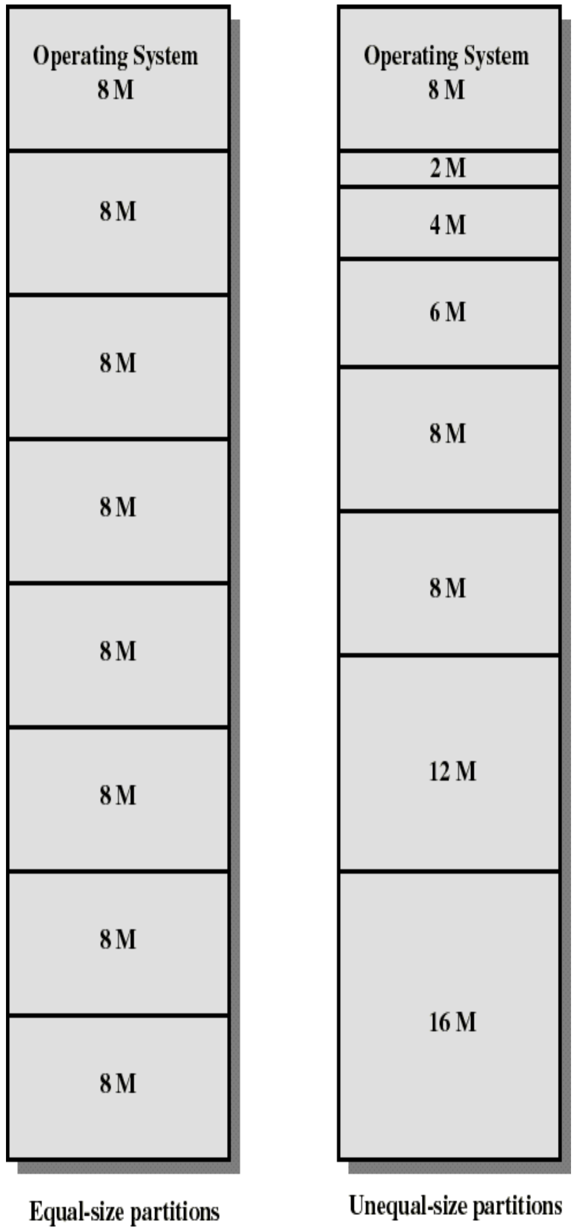
Contiguous and non-Contiguous Allocation of a Process

- The main memory must accommodate both OS and various user processes. We need to allocate it in the most efficient way.
- Main memory usually divided into two partitions:
  - Resident OS, may held in either low/high memory.
  - The major factor affecting this is the location of interrupt vector.
  - Since the interrupt vector is often in lower part, **the OS is also in the lower part.**
  - User processes then held in high memory.
- **Contiguous allocation:** all pages/segments of a process (address space) are allocated together in one chunk.
- **Non-Contiguous allocation**: pages/segments of the process (address space) can be scattered everywhere in the memory.
- What are the advantages and disadvantages of both approaches?

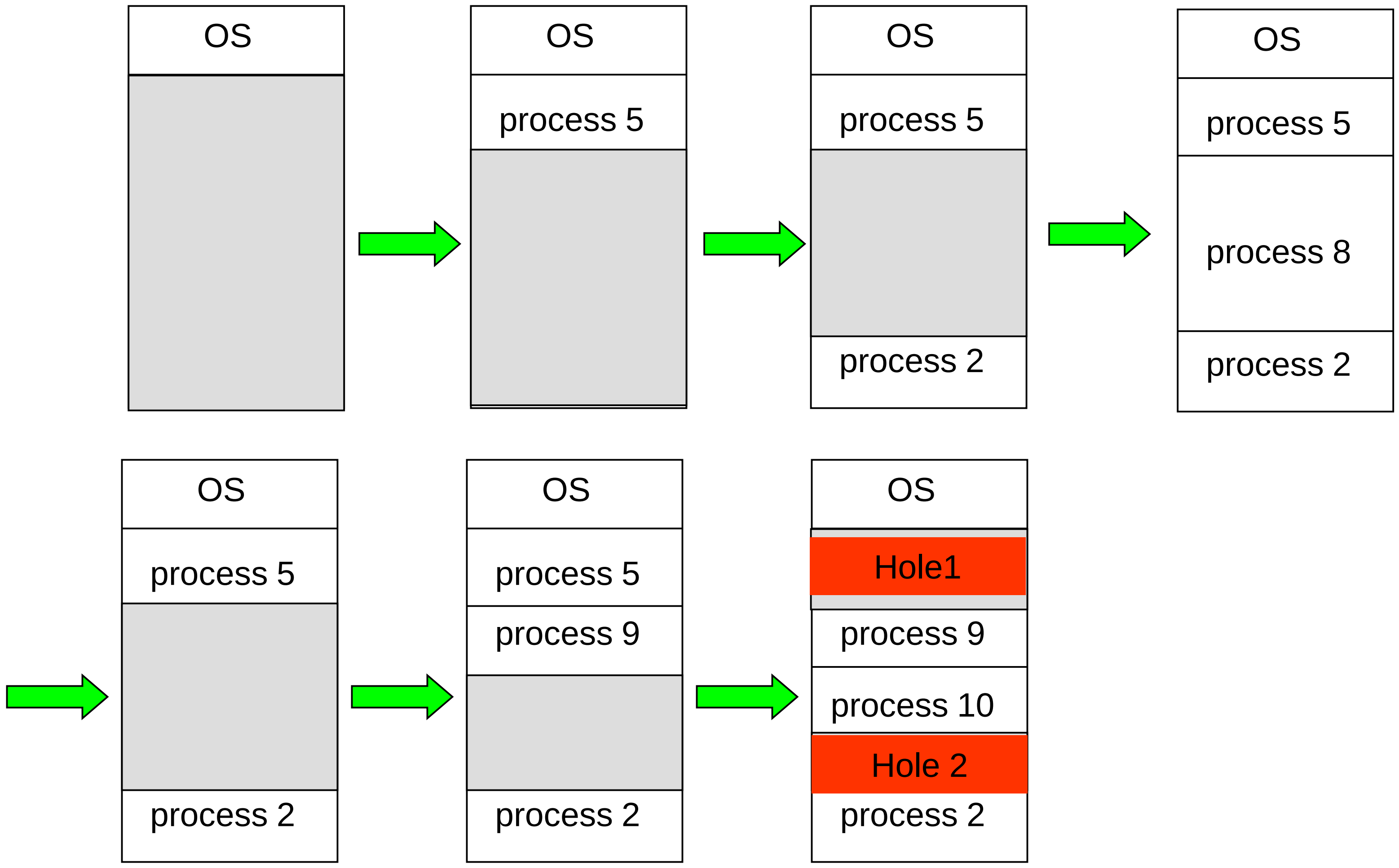


Memory Partitioning

- Partition the main memory into a set of non overlapping regions called partitions or frames.
- Partitions/Frames can be of equal or unequal sizes.
- The OS uses a table to know about which parts/frames of memory are available and which are occupied.
- A hole means un-used Partition/Frames of the main memory
  - Holes of various size may be scattered throughout memory.
- When a process arrives, it is allocated a free hole that is large enough to accommodate it.
  - The hole is split into two parts:
    - One for the arriving process,
    - One returned to the set of holes.
- When a process terminates, its allocated memory is returned to the set of holes.
  - Maybe merged with adjacent holes.

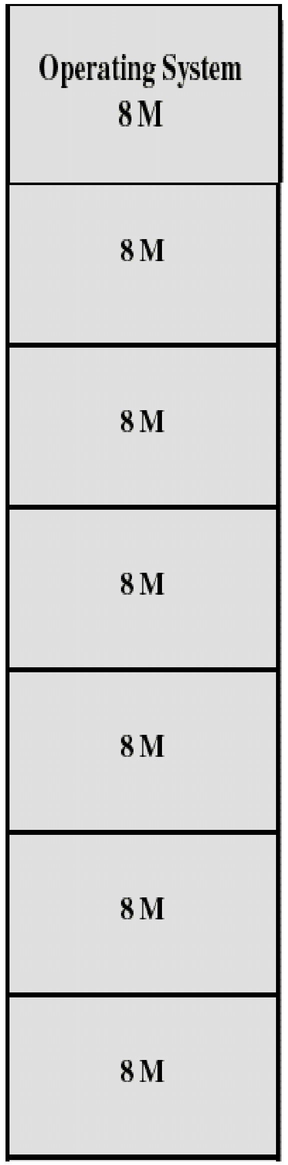


Memory Partitioning & Contiguous Allocation (Example)



Placement with Equal Size Partitions

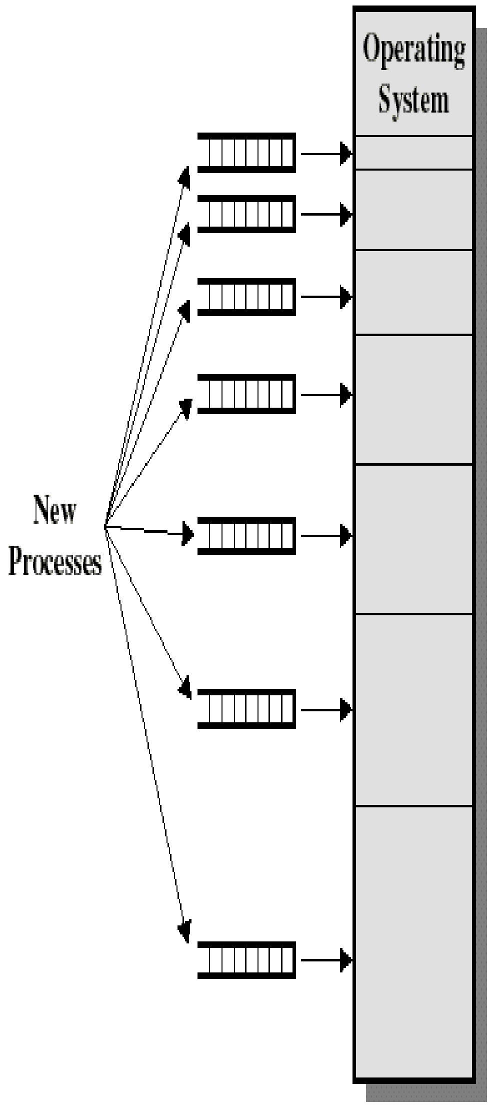
- With equal-size partitions
  - If there is an available partition, a process can be loaded into that partition.
    - Because all partitions are of equal size, it does not matter which partition is used.
  - If the process is too large to fit in a partition, then OS must swap out another process or support partial allocation.
    - When the module needed is not present, the OS must load that module into the process's address space.
    - If all partitions are occupied by blocked processes, OS chooses one process to swap out to make room for the new process.
    - When swapping out a process, its state changes to a Blocked/Suspend state, and gets replaced by a new process or a process from the Ready/Suspend queue.
- Inefficient memory utilization: Any process, no matter how small, occupies an entire partition. **The remaining hole is called internal fragmentation.**



Equal-size partitions

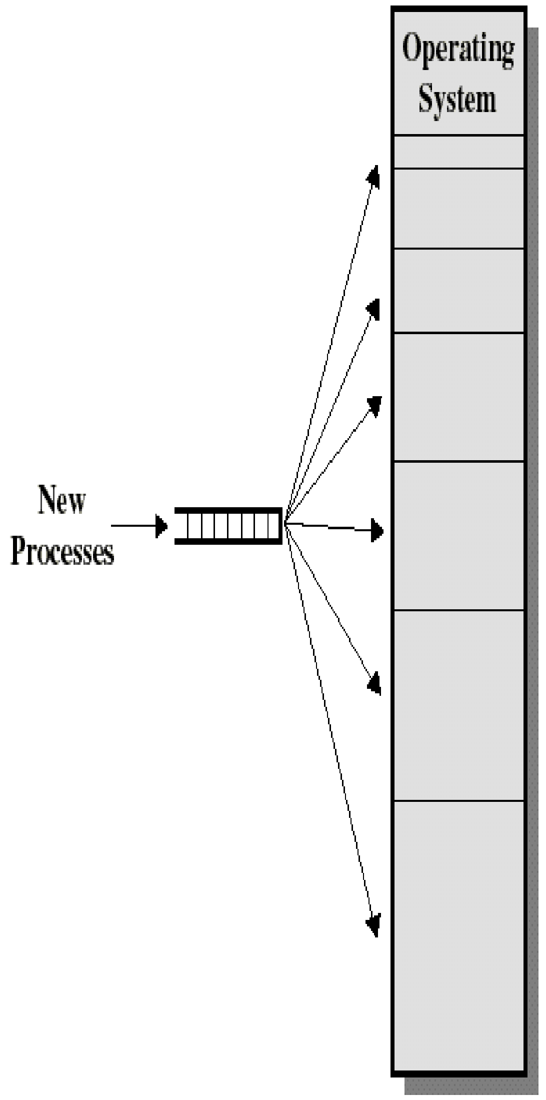
Placement with non-equal Size Partitions

1. Non-equal-size Partitions/Frames with multiple queues
   - Assign each process to the smallest partition within which it will fit.
   - A queue for each partition size.
   - Tries to minimize the problem of internal fragmentation.
   - Efficient memory utilization.
   - Problem: **Some queues will be empty if no processes within a size range is present**. That means their partitions will not be used!



Placement with non-equal Size Partitions

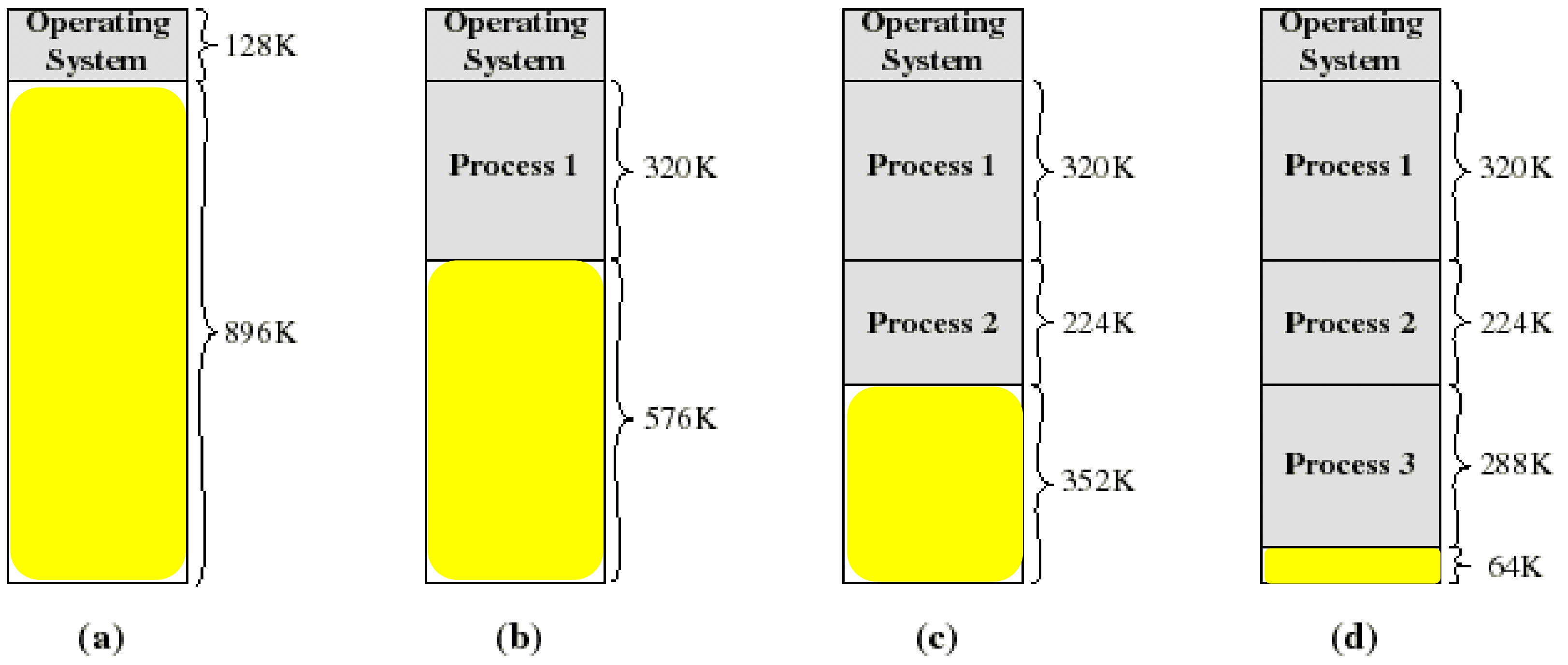
- Non-equal-size Partitions/Frames with a single queue
  - To load a process into the MM, the smallest available partition that fits the process is selected.
  - The OS can skip down the queue to see whether smaller memory requirements of some other processes can be met.
  - Increases the level of multiprocessing at the expense of queue processing and.
  - **It may lead into internal fragmentation but will be less than in equal size partitioning.**



Dynamic Partitioning

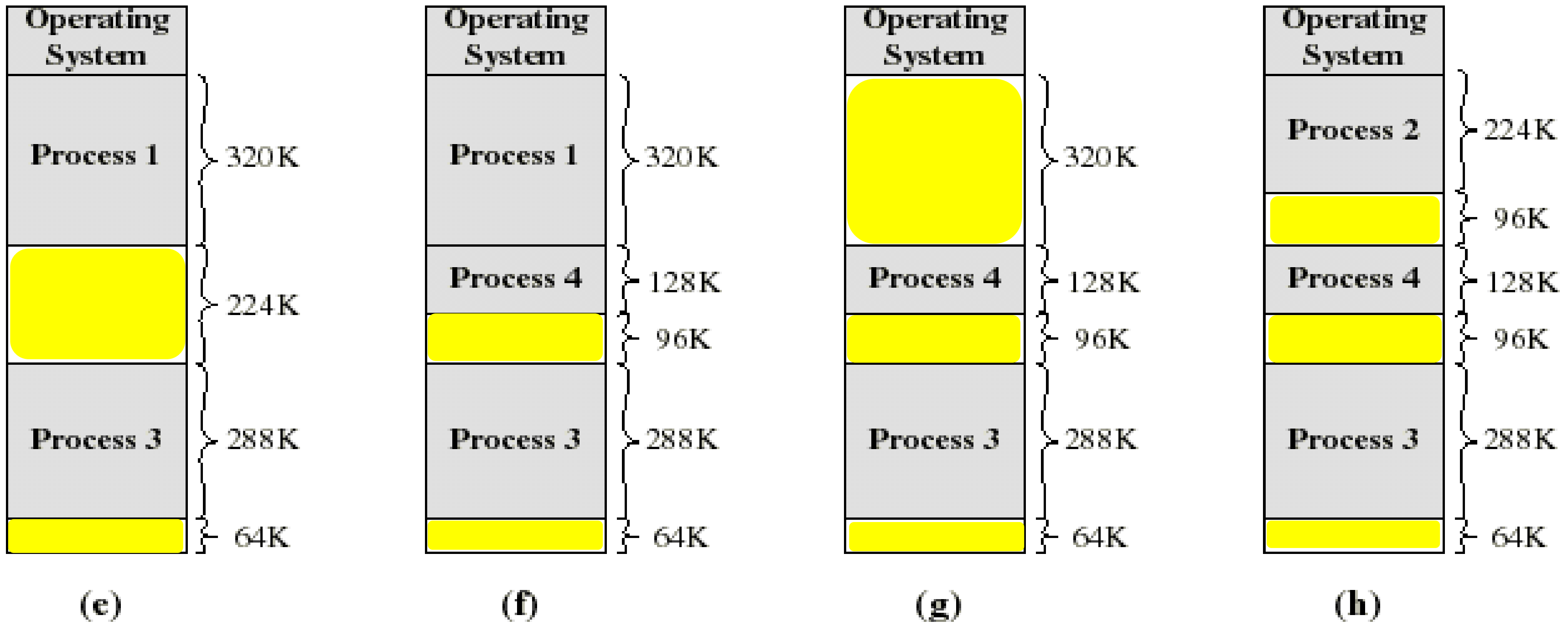
- Modern OSs use dynamic partitioning,
- Partitions are of variable length and number (called Frames).
- Each process is allocated as many frames as possible according to its working set.
- Eventually holes in between the processes (called external fragmentation) or holes inside the last allocated frame(called internal fragmentation) are formed in main memory.
- Must use compaction to shift processes so they are contiguous and all free memory is in one block.

Dynamic Partitioning: An Example



- A hole of 64K is left after loading 3 processes: not enough room for another process.
- The OS selects P2 and swaps it out to bring in P4.

Dynamic Partitioning: An Example



P2 swapped out **P4 swapped in** **P1 swapped out** **P2 swapped in**

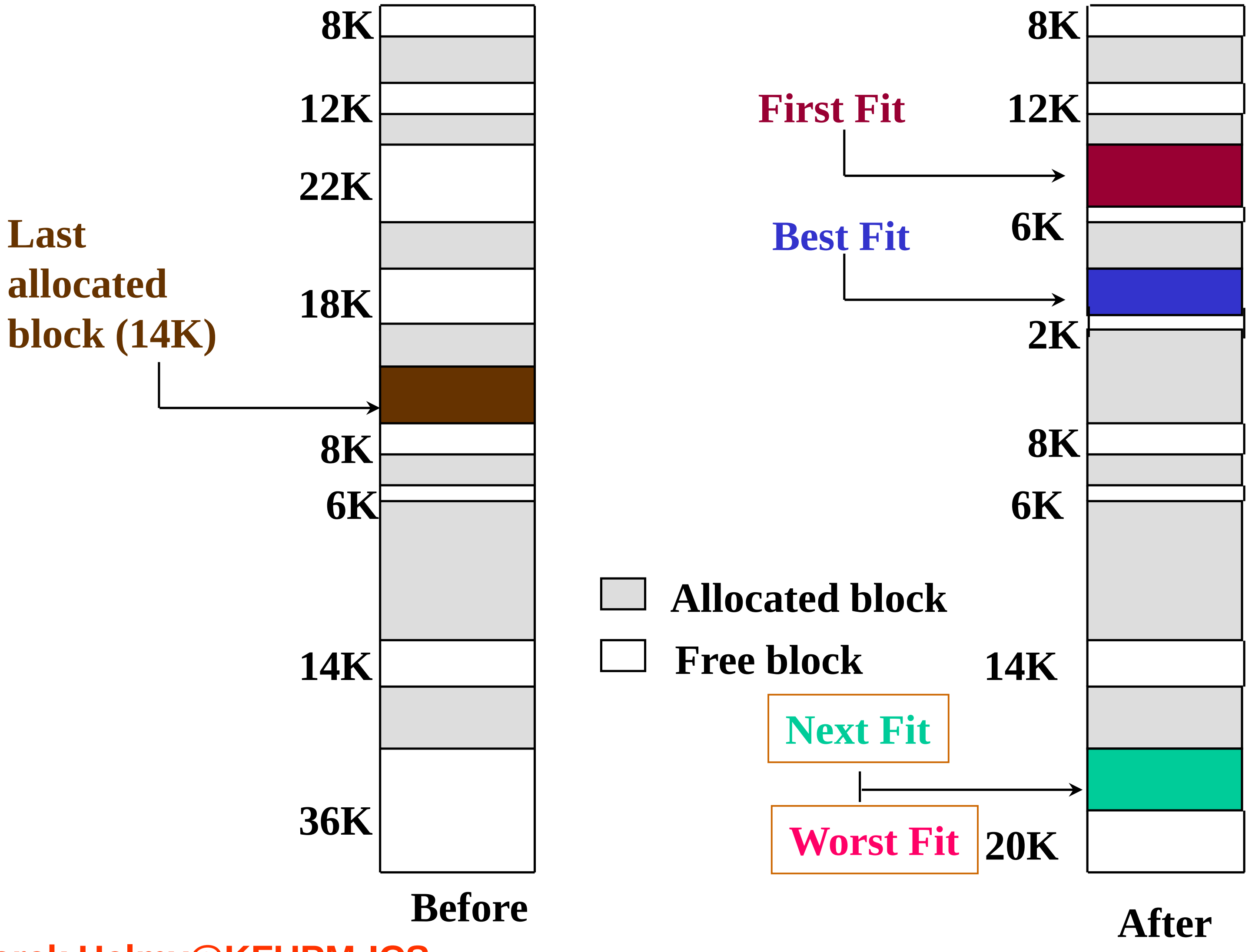
- Another hole of 96K is created
- Eventually each process is blocked. The OS swaps out P1 to bring in again P2 and another hole of 96K is created...
- **Compaction would produce a single hole of** (96+96+64)= 256K

Placement Algorithms

- OS must decide which free block/partition to allocate to a process in order to reduce the compaction time. The following algorithms are used:
- Best-fit algorithm
  - Chooses a block/partition in the entire memory that is close enough in size to the loaded process.
  - Results in minimally sized fragments that require compaction.
  - We must search the entire list unless they have been sorted by size.
- First-fit algorithm
  - Starts scanning the MM from the beginning and chooses the first available block/partition that is large enough to fit the process.
  - May have many process loaded in the front end of memory that must be scanned.
- Next-fit
  - Scans the MM from the location of the last allocation and chooses the next available block/partition that is large enough to fit the process.
  - More often allocate a block of memory at the end of memory where the largest block is found.
  - Compaction is required to obtain a large block at the end of memory.
- Worst-fit
  - Chooses the biggest block/partition first. Makes bigger partitions more useful.
- Quick-fit
  - Chooses a block/partition from a common-size partition list.

Example: Placement Algorithm

alloc 16K block



▨ Allocated block

□ Free block

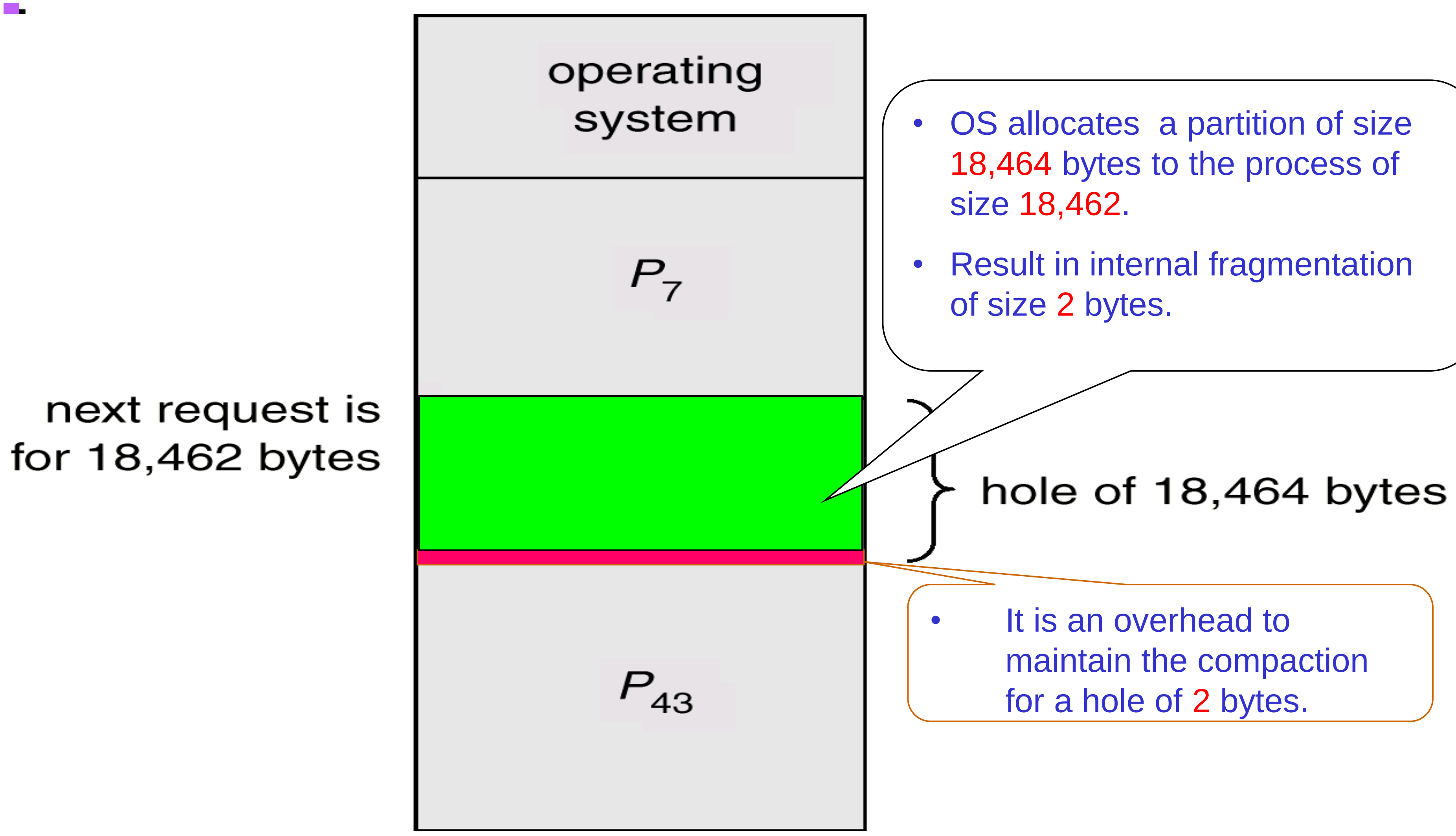
Placement Algorithm: Comments

- Best-fit searches for the smallest partition: the fragment left behind is as small as possible.
  - Main memory quickly forms holes too small to hold any process: **compaction needs to be done more often.**
- First-fit favors allocation near the beginning: tends to create less fragmentation than Next-fit.
- Next-fit often leads to allocation of the largest blocks at the end of memory.
- Worst-fit and quick-fit have still fragmentation (useless holes) problems.

Fragmentation Manipulation

- External fragmentation
  - Unused memory partitions (un-allocated partitions to processes).
- Internal fragmentation
  - Unused memory locations within a partition because the allocated process may be slightly smaller than the allocated partition. For example, consider a hole of 18,464 bytes to allocate to a process with 18,462 bytes.
  - Also happen when physical memory is broken into fixed-sized large blocks, and memory is allocated to processes in unit of block sizes.
- Fragmentation Manipulation:
  - Compaction/defragmentation: re-locate memory contents to place all free memory locations together in one large partition that can host a process.
  - It is possible *only* if relocation is supported, and the address mapping is done at execution time.
    - After compaction, the contents of both Limit and Base registers must be modified.
  - Paging: allow the logical address spaces of the process to be non-contiguous, thus a process can be allocated a memory partition wherever it is available.
  - Segmentation: divide the process logical address space into variable-sized segments, with semantic base, i.e. (Data segment, Code segment, Tables segment, Arrays segment, Stacks segment, etc.)

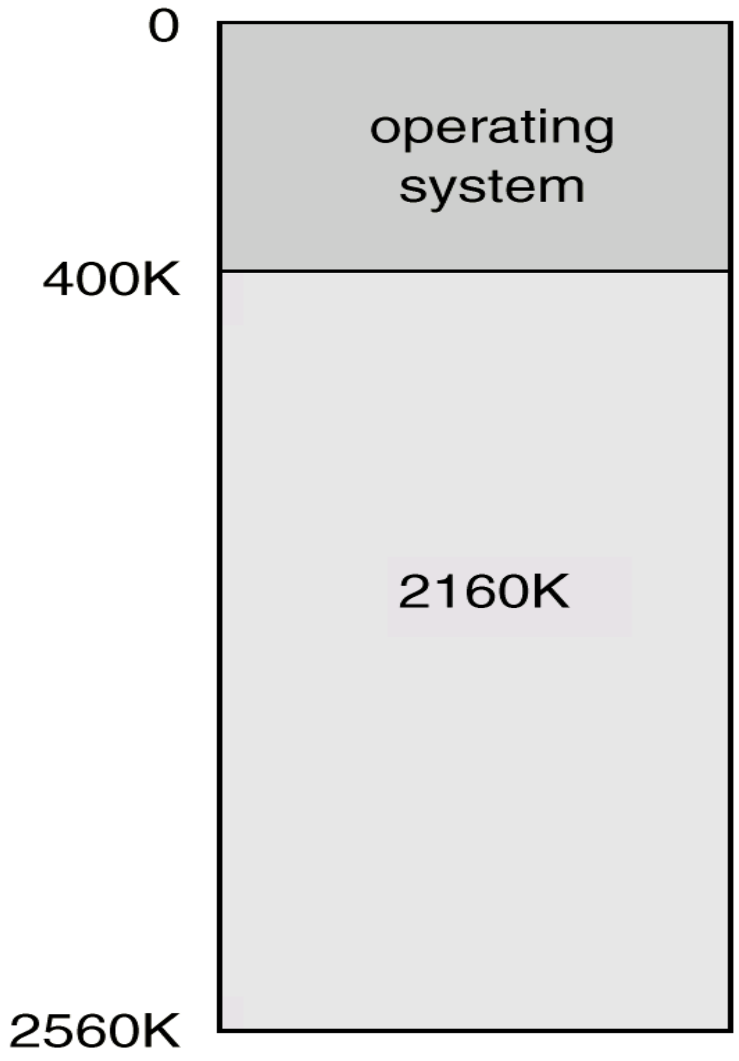
Internal Fragmentation



- OS allocates a partition of size 18,464 bytes to the process of size 18,462.
- Result in internal fragmentation of size 2 bytes.

- It is an overhead to maintain the compaction for a hole of 2 bytes.

External Fragmentation: Example

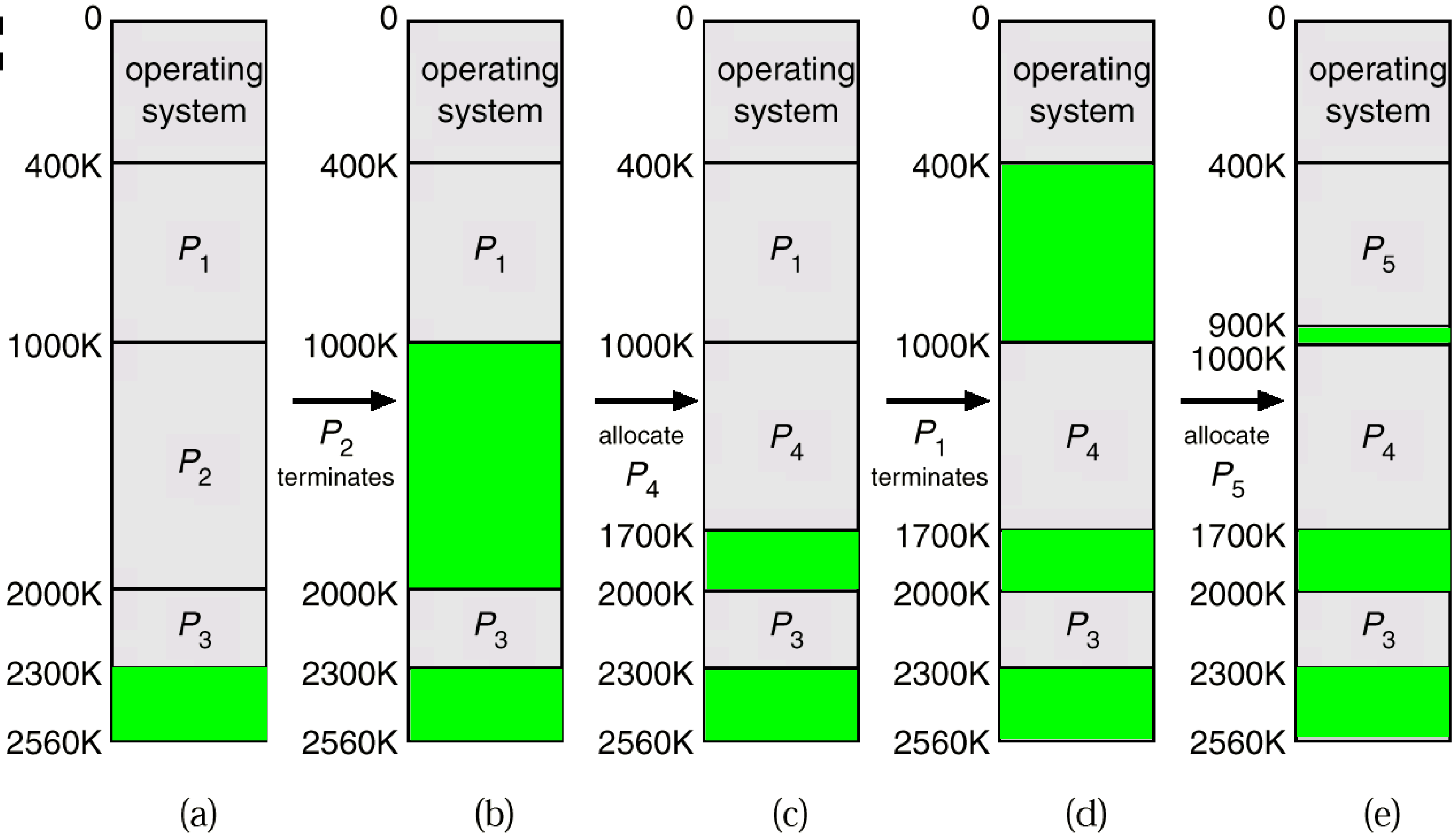


job queue		
process	memory	time
P_1	600K	10
P_2	1000K	5
P_3	300K	20
P_4	700K	8
P_5	500K	15

External Fragmentation: Example

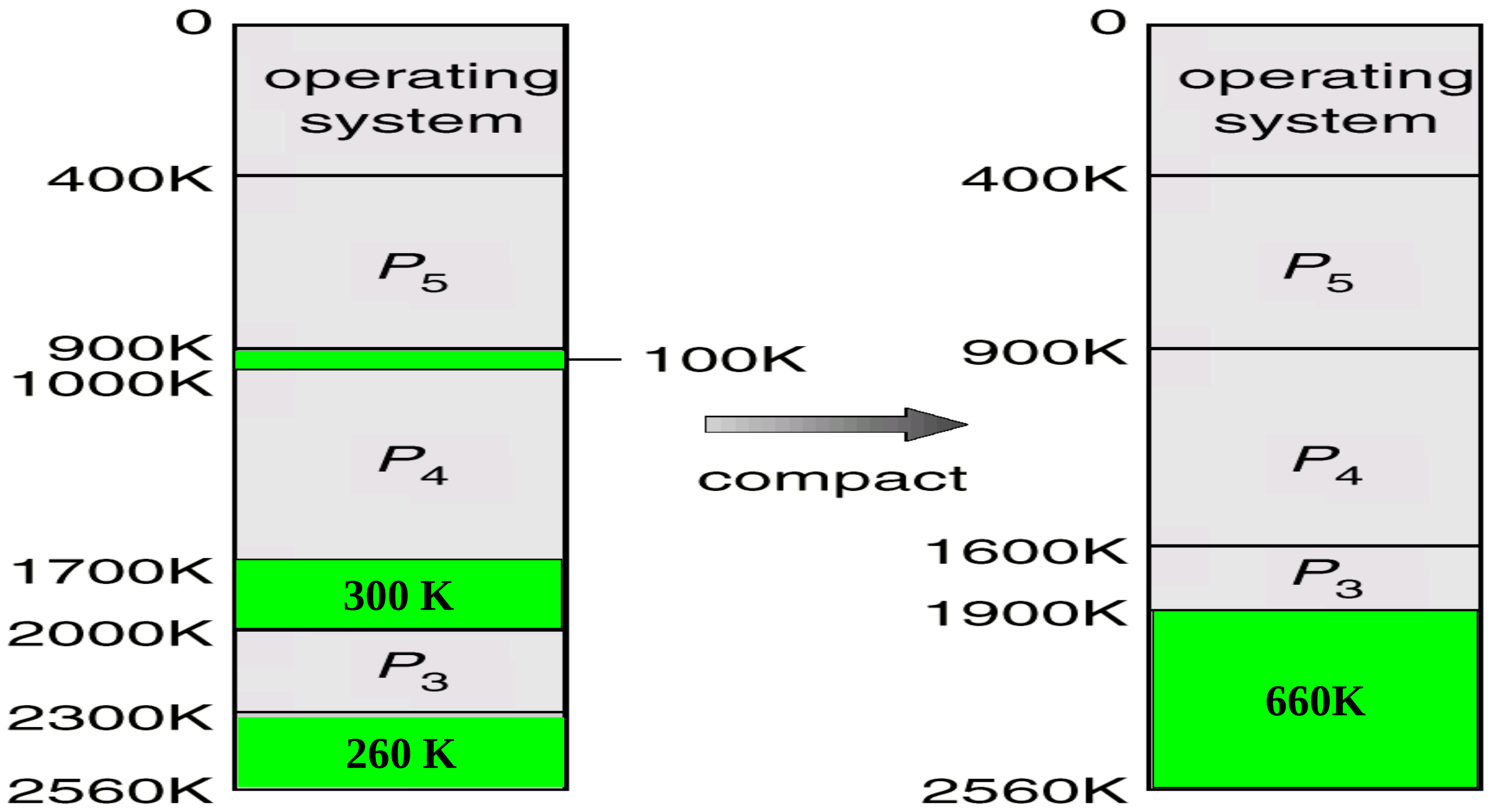
⬜ Allocated block

🟩 Free block



Compacting External Fragments

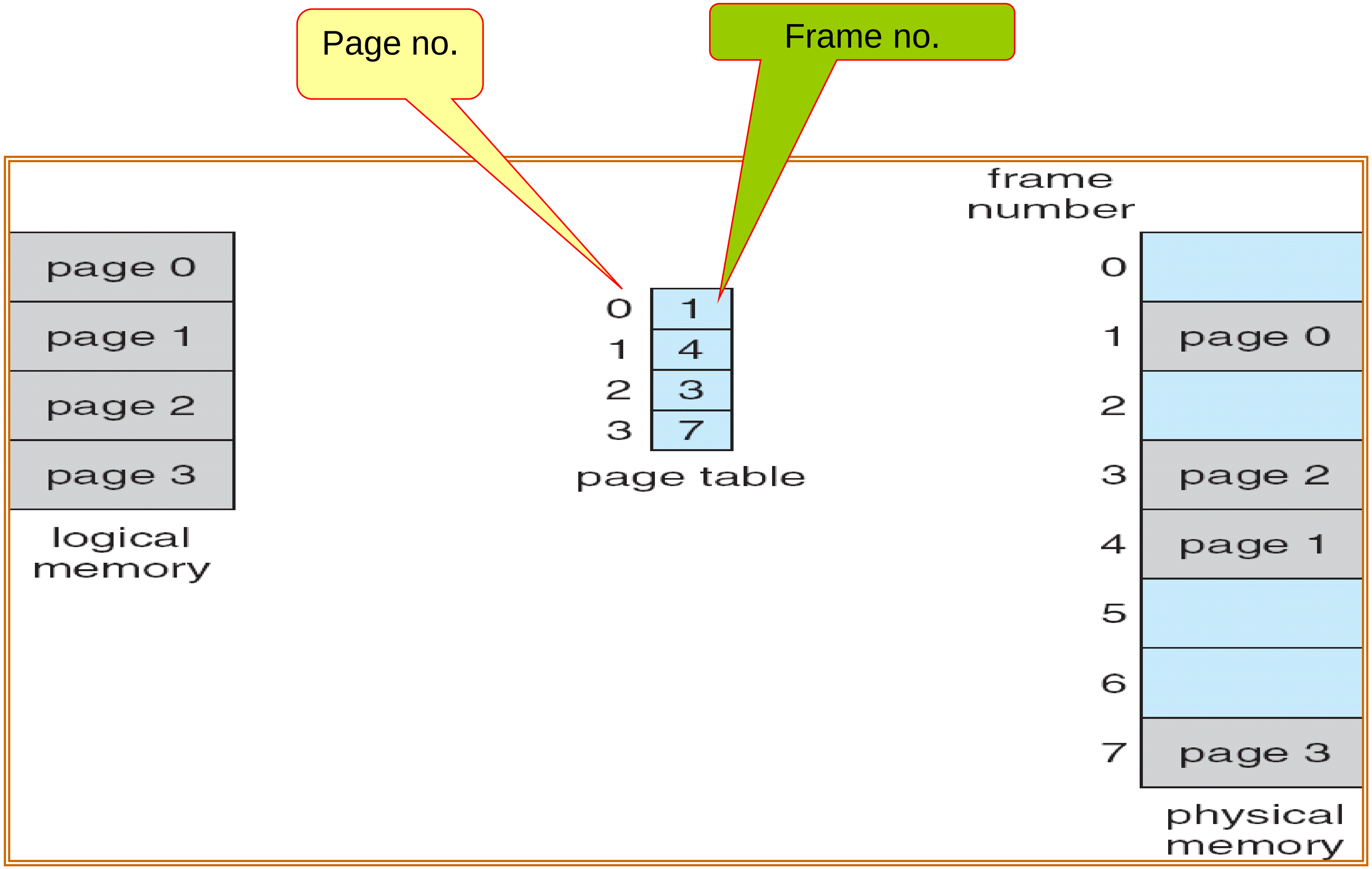
Compaction: re-locate memory contents to place all free memory together in one large block.



Paging to minimize Fragmentations

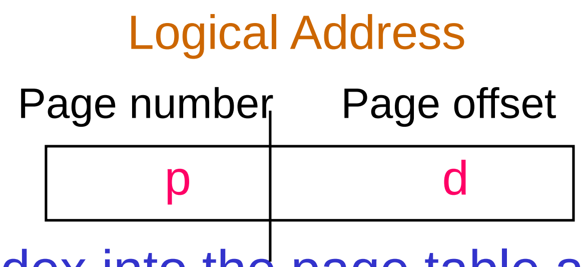
- Paging permits the logical address space of a process to be non-contiguous.
- Divide physical memory into blocks called frames.
  - The size of the frame is a power of 2, between 2^9=512 KB and 2^{24}=16 MB
- Divide logical memory (process space) into blocks of same size called pages.
- The size of a page is typically a power of 2, from 512 KB to 16 MB. **The page's size will be the same size as of the frame.**
- OS has to keep track of all free frames,
- To run a process of size n pages, the OS needs to find n free frames and loads the process's n pages into n free frames. Otherwise partial loading will be done.
- The OS uses a page table to translate the logical address to physical address.
  - Each process has a page table
    - A pointer to the page table is stored with the other registers in PCB.
    - CPU dispatcher loads the page table into the system-wide hardware page table (like PC and registers) as part of the context switch.
- With paging, internal fragmentation may be there (a process may use some bytes from the last page), while external fragmentation will not be there (any free frame wherever it is can be allocated to a process).
- Page and frame sizes depends on the HW architecture of the processor.

Paging Example



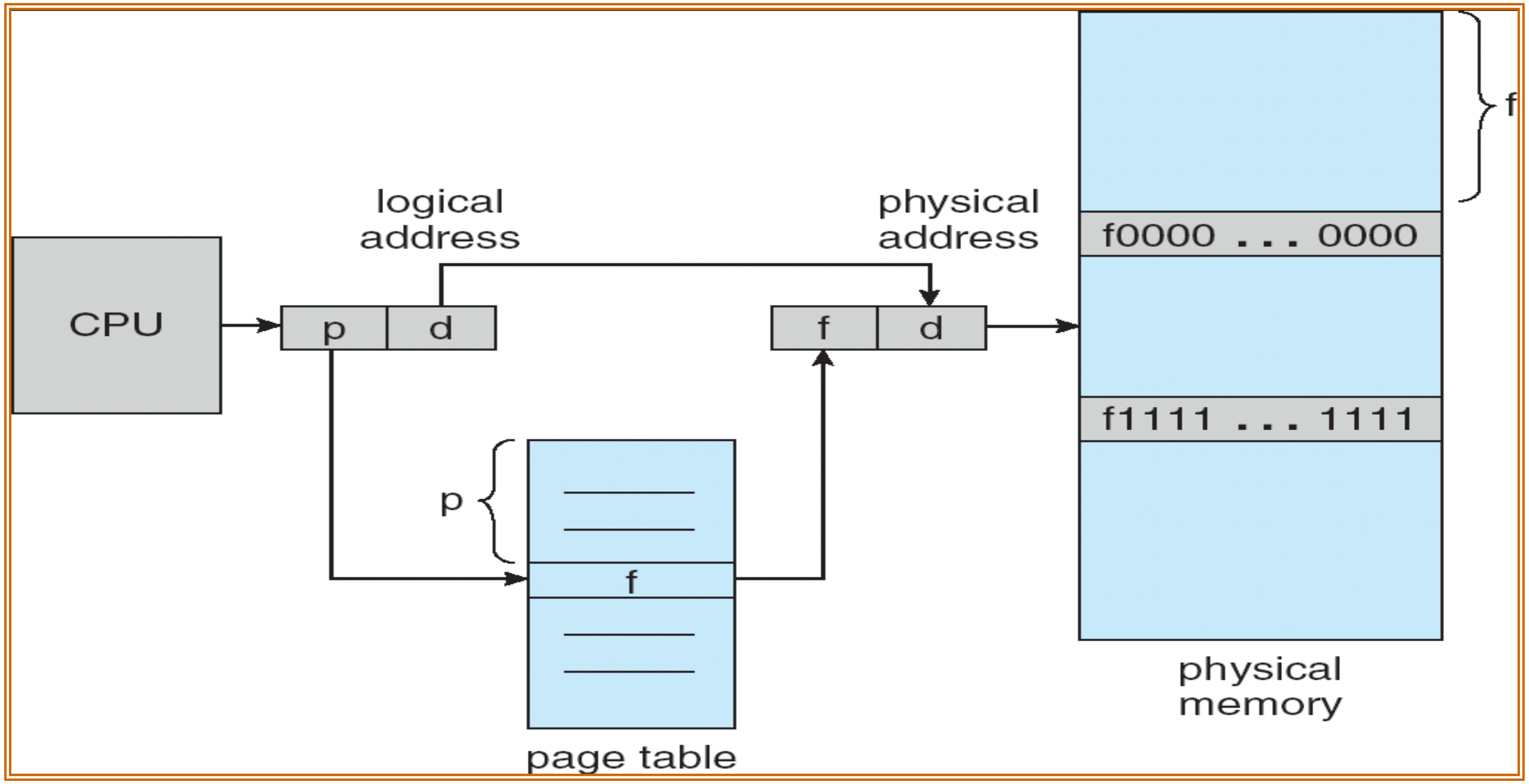
Address Translation/Mapping Scheme

- Address generated by CPU (logical address) is divided into:
  - Page number (p): Used as an index into a page table to identify the **frame number** (base address) of each page in physical memory.
  - Page offset (d): Combined with base address (frame number) to define the physical memory address that is sent to the memory address register.
- If the size of **logical address space** is 2^m, and a page size is $2^{n.}$
- The page table will have $(2^m/2^n)$ m-n entries.



- Where p is an index into the page table and d is the displacement within the page.
- **If the size of the physical memory** is 2^k
- The width (# of bits) of each page table entry= $(2^k/2^n)$ k-n bits

Paging Hardware: Address Translation



If the size of Logical address space is 2^m bytes, the size of the MM is 2^k, page size is 2^n then entries of the PT are equal to the number of pages $(2^m/2^n)$ (m-n) and the width of each entry equals k-n bits.

Physical Address

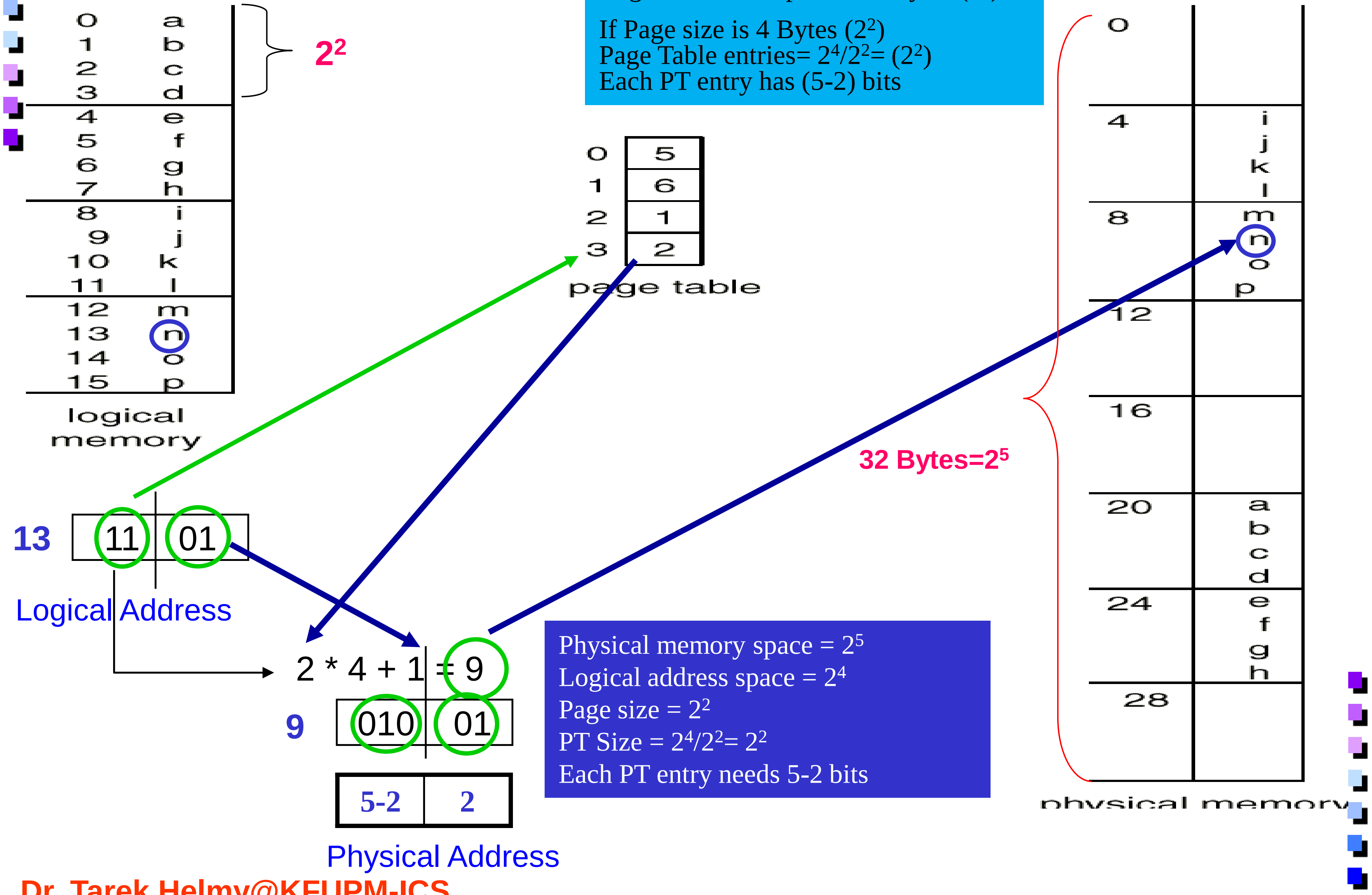
frame number	frame offset
f (k-n)	d (n)

Paging Example

Physical memory space =32 Bytes, (2^5)

Logical address space =16 Bytes (2^4)

If Page size is 4 Bytes (2^2)
Page Table entries= $2^4/2^2$= (2^2)
Each PT entry has (5-2) bits



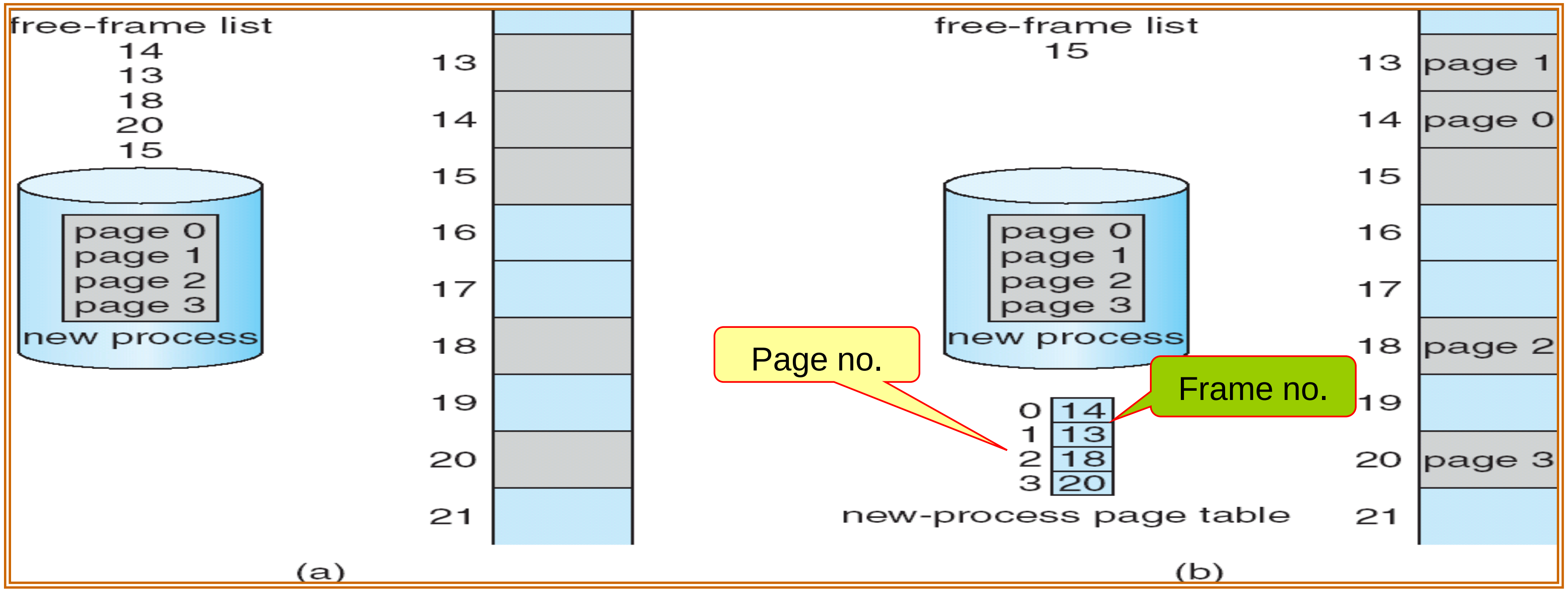
Physical memory space = 2^5
Logical address space = 2^4
Page size = 2^2
PT Size = $2^4/2^2$= 2^2
Each PT entry needs 5-2 bits

Exercise

- Consider a logical address space of 128 pages of 2048 bytes each, mapped onto a physical memory of 64 frames.
  - How many bits are there in the logical address?
  - How many bits are there in the physical address?

OS has to keep a list of Free-Frames

1. At the time of executing a process, its size expresses in pages will be determined. Each page will be mapped to one frame.
2. If the process requires urgently n pages, at least n frames must be available in memory. Otherwise partial loading will be supported.
3. If n frames are available, they are allocated to this process. The first page of the process is loaded into one of the allocated frames and the frame number is put in the page table for this process and so on.



Before allocation

After allocation

Page Table Size & Page Size

- With each process having its own page table, and with each page table consuming considerable amount of memory.
- A lot of memory will be used to keep track of the main memory. i.e.
- Consider a process with 32-bit logical address space (4GB), if the page size is 4 KB (2^{12}) then a page table may consists of up to 1 million entries, $2^{32}/2^{12}$=1MB. Assuming that each entry needs 4 bytes (**to represent the # of frames**) then each process may need up to 4 MB of the MM for its page table only.
  - Smaller page size leads to more pages, and more pages lead to larger page table's size.
  - Want to set page size to reduce internal fragmentation

FYI: Optimal Page Size

- What is the optimal page size (p)?
  - The OS researchers came out with an equation to determine **the optimal page size** as following:

$$p = \sqrt{2se}$$

    - s = average process size,
    - e = size of each page table entry
  - Overhead = (s/p)*e + p/2
    - s/p = average number of pages per process
    - (s/p)*e = space taken up by average process in page table.
    - p/2 = average wasted memory in the last page of process due to internal fragmentation.

Implementation of Page Table

- Most OSs allocate a page table for each process.
- A pointer to the process's page table is stored with other register values in the PCB.
- When the dispatcher is asked to start a process, it reloads that pointer and defines the correct page table value.
- **Most OSs allow page table to be very large and so**, page table is kept in main memory (OS memory) and,
  - Page-Table Base Register (PTBR) points to the page table.
  - Changing the page table requires only to change PTBR.

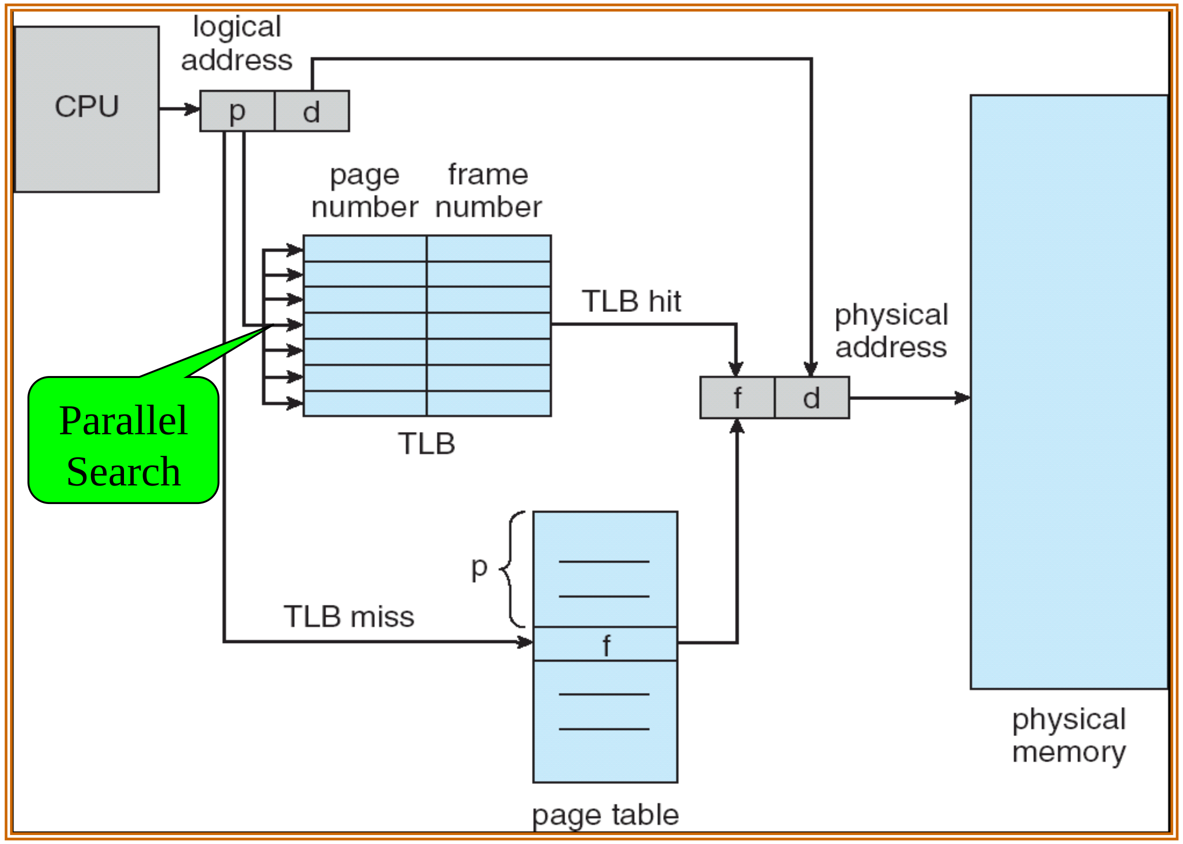
  The problem is:
- Every data/instruction access requires **two memory accesses**
  - One for the **address mapping** and one for **the data/instruction**.
- The simplest way to implement the page table is to use fast dedicated registers if the size of the page table is small.

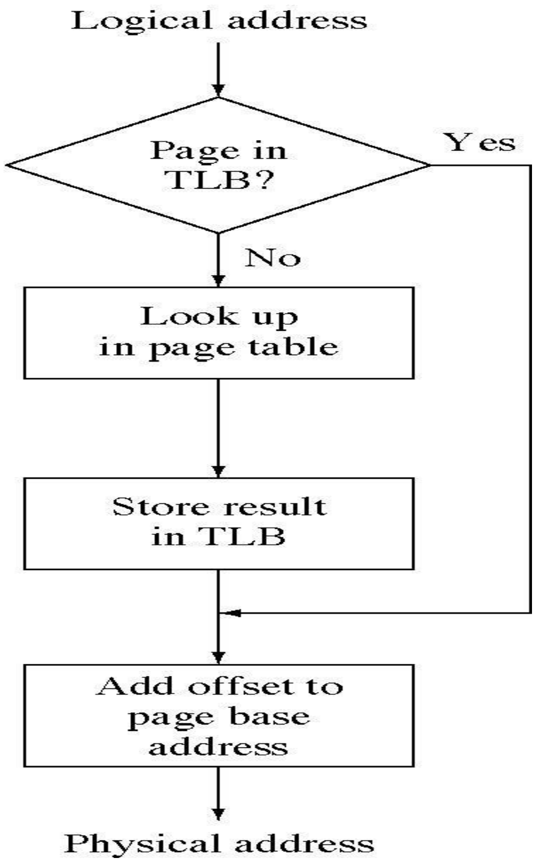
Associative Register (Hardware)

- Two access problem of the MM can be solved by using of a special fast-lookup hardware cache called associative registers or Translation Look-aside Buffers (TLBs).
- Each entry of the TLB consists of a key and a value
  - The TLB contains only (**the working set**) a few of the page table entries.
  - When the logical address is generated by the CPU, its page number is presented to the TLB in parallel search scheme,
  - If the page number is found, its frame number is used to access the MM.
  - If the page number is not in the TLB, a memory reference to the page table must be done.
  - When the frame number is obtained, use it to access MM and add the page and frame number to the TBL for the next reference.
- **Every time a new page table is selected, the TLB entries must be erased to ensure that the next executing process does not use the wrong translation information.**
- Some TLB stores Address-Space Identification (ASID) in each TLB,
- ASID uniquely identifies each process and is used for protection.

Translation Look-aside Buffers (TLBs)

- Two accesses of the MM can be solved by the use of a special fast-lookup hardware cache called associative or Translation Look-aside Buffers (TLBs).
- Each entry of the TLB consists of a key (page number) and a value (frame number)
- The TLB contains only a few of the page table's entries (**working set**).





Effective Access Time (EAT)

- Associative (TLB) Lookup time = β time units
- Assume memory access time is x time units
- Hit ratio (α): Percentage that a page number is found in the TLB.
- Hit ratio (α): should be increased by increasing the number of entries in the TLB, however it is costly as the associative memory is too expensive.

- Effective memory-access time (EAT)

$$EAT = (x + \beta)\,\alpha + (2x + \beta)(1 - \alpha)$$

Effective Access Time (EAT)

$$EAT = (x + \beta)\,\alpha + (2x + \beta)(1 - \alpha)$$

- Example 1
  - Associate lookup $\beta = 20$
  - Memory access $x = 100$
  - Hit ratio (α) = 0.8
  - EAT = (100 + 20) * 0.8 + (200 + 20) * 0.2 = 1.2 * 100 + 20 = 140

40% slow in memory access time

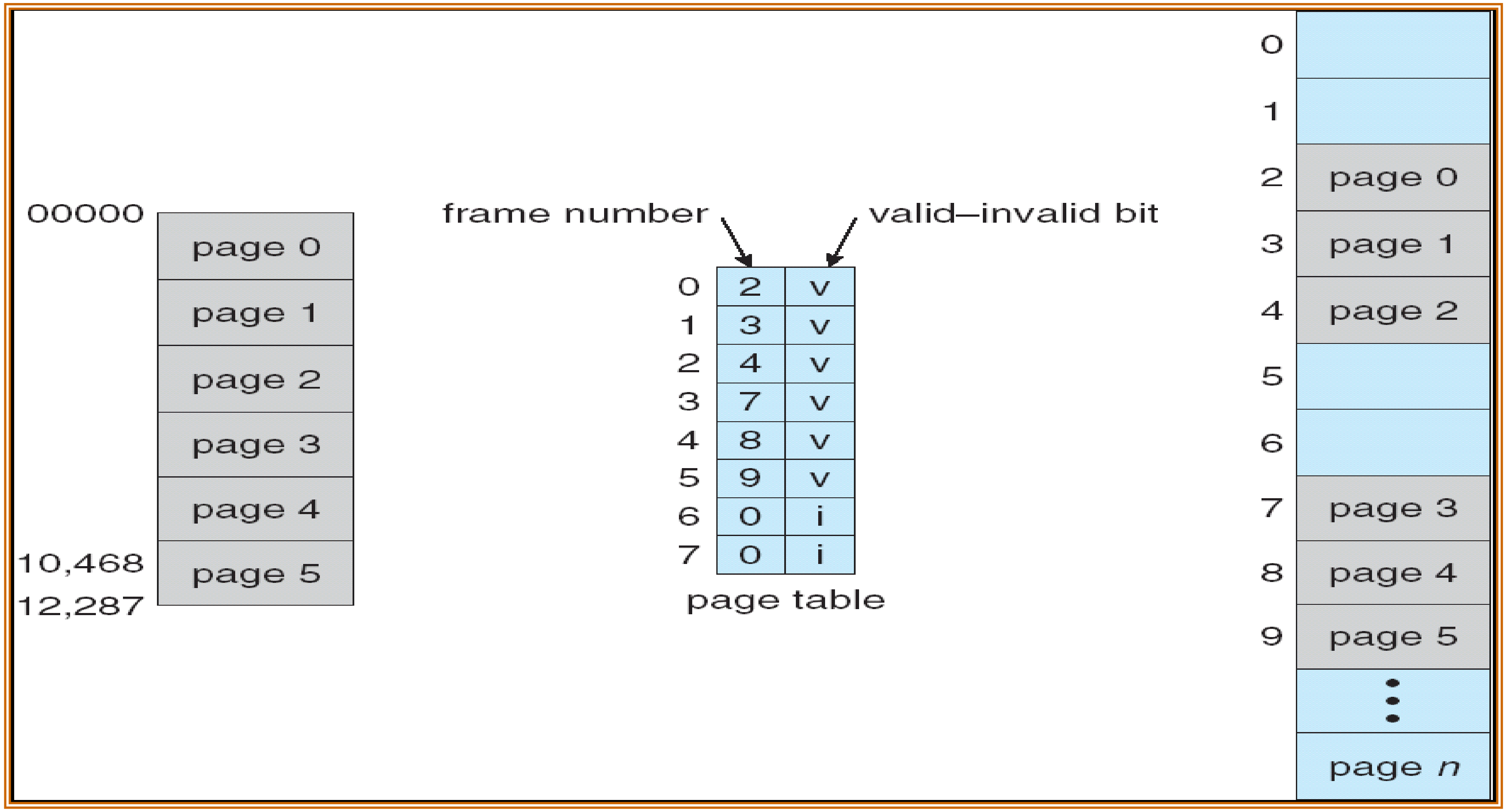
- Example 2
  - Associate lookup $\beta = 20$
  - Memory access $x = 100$
  - Hit ratio (α)= 0.98
  - EAT = (100 + 20) * 0.98 + (200 + 20) * 0.02 = 1.02 * 100 + 20 = 122

22% slow in memory access time

Validity of Pages and EAT

- Additional bits can be added to the page table to identify:
  - Validity of pages, access type (read-only or read-write)
- Such kind of bits can enhance the protection and also minimize the EAT by avoiding the second time access to memory if the page is not there in the MM.
- Valid-invalid bit attached to each entry in the page table:
  - "Valid" indicates that the associated page is in MM.
  - "Invalid" indicates that the page is not in MM and thus the second acess to the MM should be avoided.
  - The OS uses this bit to allow or disallow access to that page.

Memory Protection with Valid-Invalid bit



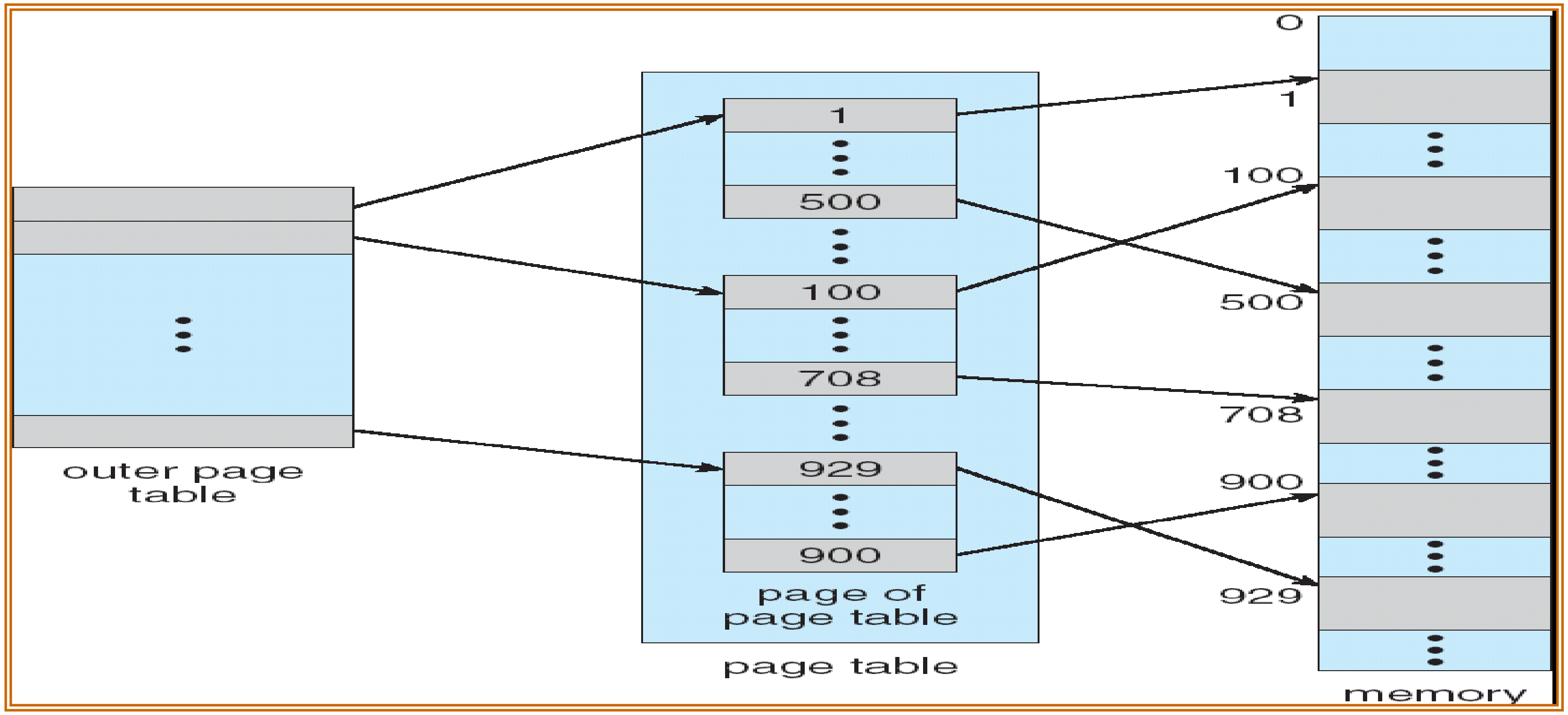
1. **Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through page table.**
2. **Second access to generate an address for pages 6 or 7 should be avoided.**

Page Table Structure

- With each process having its own page table stored in memory and used to map logical address into physical one.
- A huge a mount of memory will be used to map logical addresses into physical addresses.
- It also needs more free and contiguous memory space to be stored in.
- How to solve this problem?
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables

Hierarchical Page Tables

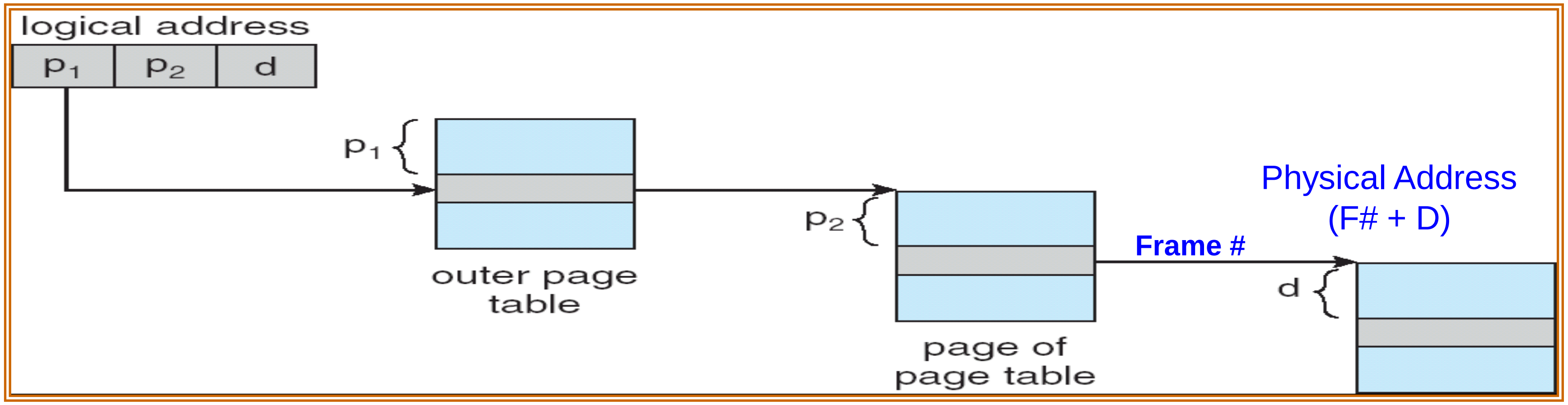
- To resolve the large size page tables problem, we want to allocate the page table into noncontiguous blocks.
- And this means, **break up the page table address space into multiple page tables**.
- A simple technique is to use two-level paging algorithm, in which the page table itself is also paged.



Two-Level Page-Table Scheme

Multilevel Page Tables

- Since a page table will generally require several pages to be stored. One solution is to organize page tables into a multilevel hierarchy.
  - When 2 levels are used, the page number is split into two numbers p1 and p2
  - P1 indexes the outer paged table (directory) in main memory who's entries point to a page containing page table entries which is itself indexed by P2.
  - Page tables, other than the directory, are swapped in and out as needed.



Two-Level Paging Example

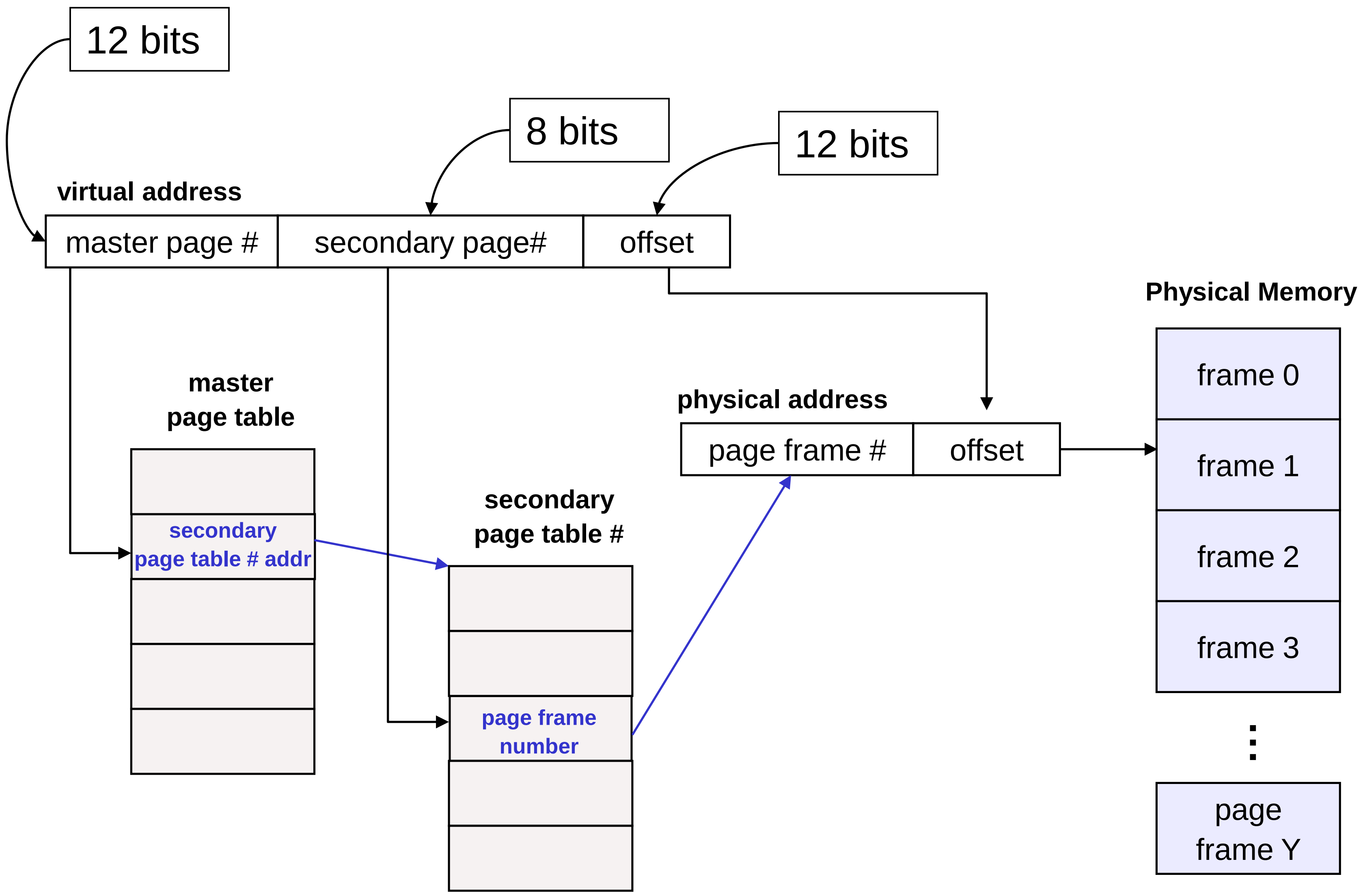
- A logical address (on 32-bit machine with 4K page size) is divided into:
  - A page number consisting of 20 bits (32 - 12 = 20)
    - If each entry needs 4 bytes➔2^{20} * 4 bytes = 4 MB
  - A page offset consisting of 12 bits

page number	page offset
20	12

- Since the page table is paged, the page number is further divided into:
  - A 10-bit page number.
  - A 10-bit page offset.
- Thus, a logical address is as follows:
  - P_1 is an index into the outer page table, and P_2 is the displacement within the page of the outer page table.
  - Because address translation works from the outer page table inwards, it is also known as a forward-mapped page table and is used in Pentium 2.

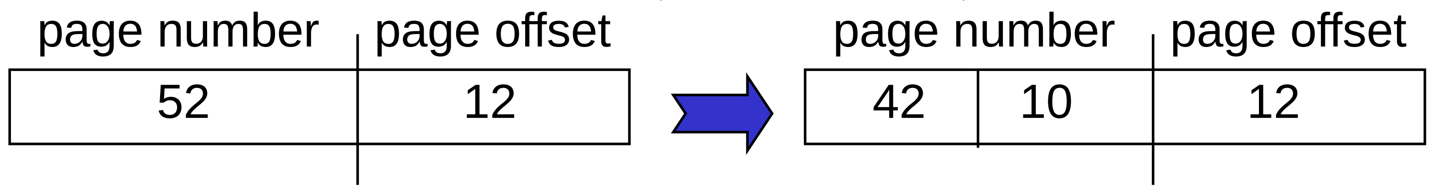
Page number		Page offset
p_1	p_2	d
10	10	12

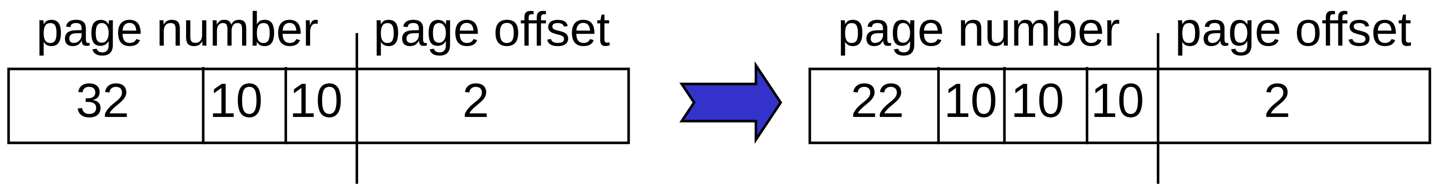
Two-Level Page Tables



Multilevel Paging and Performance

- A 64-bit logical address space with 4K page size:
  - # of PT entries = 2^{52} (64 -12 = 52)





- Since each level is stored as a separate table in memory, **converting a logical address to a physical one may take many memory accesses**.

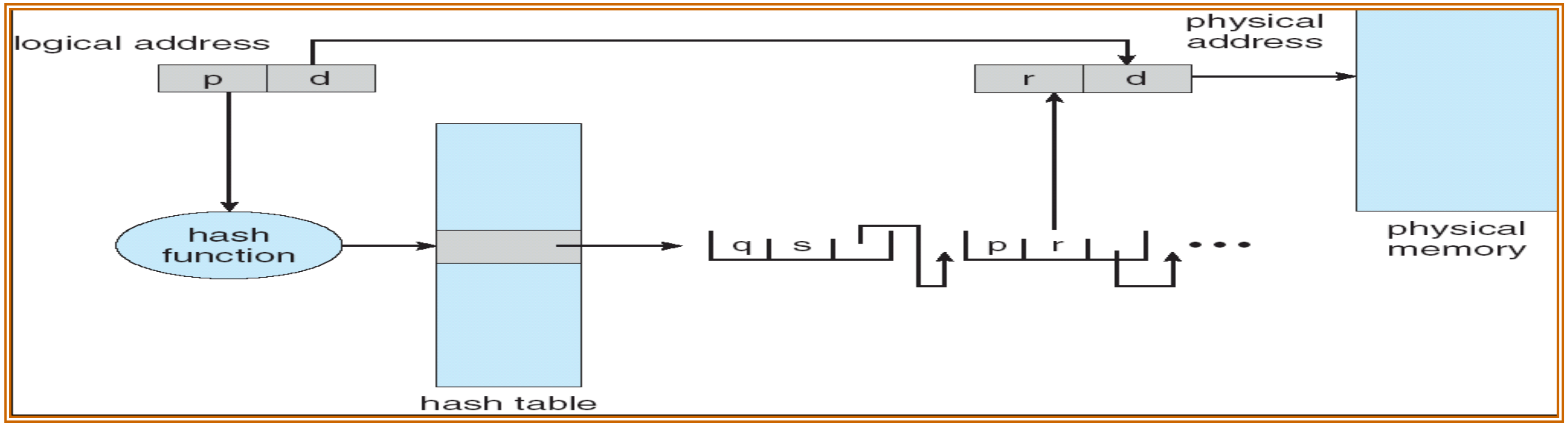
- Even though time needed for one effective memory access is increased, **caching permits performance to remain reasonable**.

Hashed Page Tables

- If we have a collection of **n** pages whose keys [base addresses] are unique integers then we can store them in a hash table.
- Lookup time is minimized.
- **Common in address spaces > 32 bits,**
- The virtual page number **is hashed into a page table** that contains a chain of elements hashing to the same location.
- Each element consists of three fields, virtual page number (Key), value of the mapped page frame, a pointer to the next element in the list.

The algorithm works as follows:

- The page index [Key] is compared with the virtual page number in linked list for a match.
- If a match is found, the corresponding value of the mapped page's frame is extracted to form the desired physical address.
- If there is no match, subsequent entries in the list are searched for matching.



Inverted Page Table

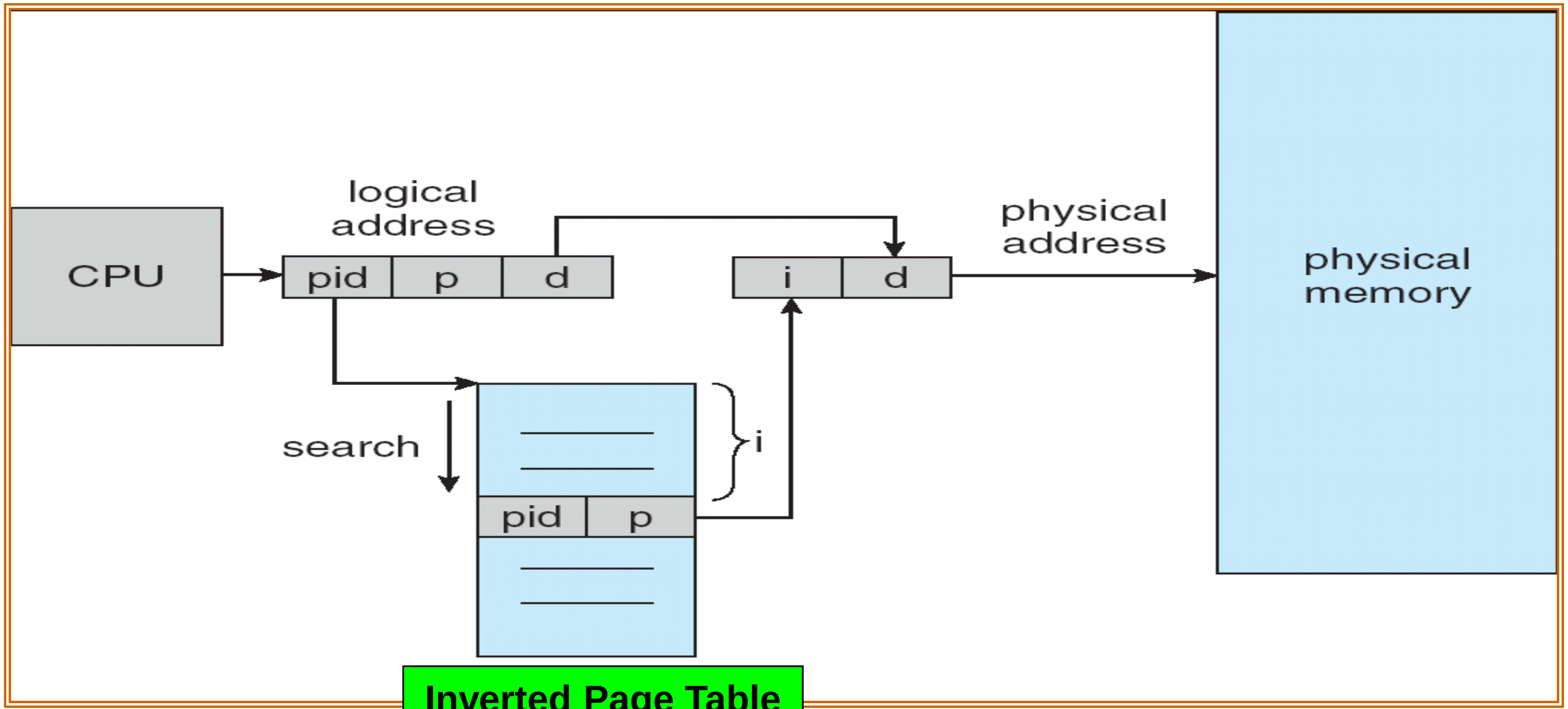
- Another solution to the problem of maintaining large page tables is to use an Inverted Page Table (IPT).
- We generally have only one IPT for the whole system.
- There is only one entry per physical frame in the IPT (rather than one per virtual page).
  - This reduces a lot the amount of memory needed for page tables.
- The 1st entry of the IPT is for frame #1. The nth entry of the IPT is for frame #n and each of these entries contains the virtual page number.
- The process ID with the virtual page number could be used to search the IPT to obtain the frame #.
  - Maintain inverted page table in associative memory hardware TLB **whose entries are searched in parallel**.
  - Use hash table to hash the virtual page address.
- A page fault occurs if no match is found.

Inverted Page Table

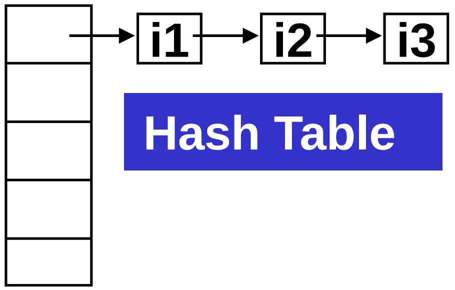
The algorithm works as follows:

- Each inverted page table entry is a pair of <process-id, page-number>
- When a memory reference occurs, the <process-id, page-number> is compared with the contents of the inverted page table.
- If a match is found, say at ith frame then the physical address <i, offset> is generated.
- It decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.

Inverted Page Table Hardware



Inverted Page Table

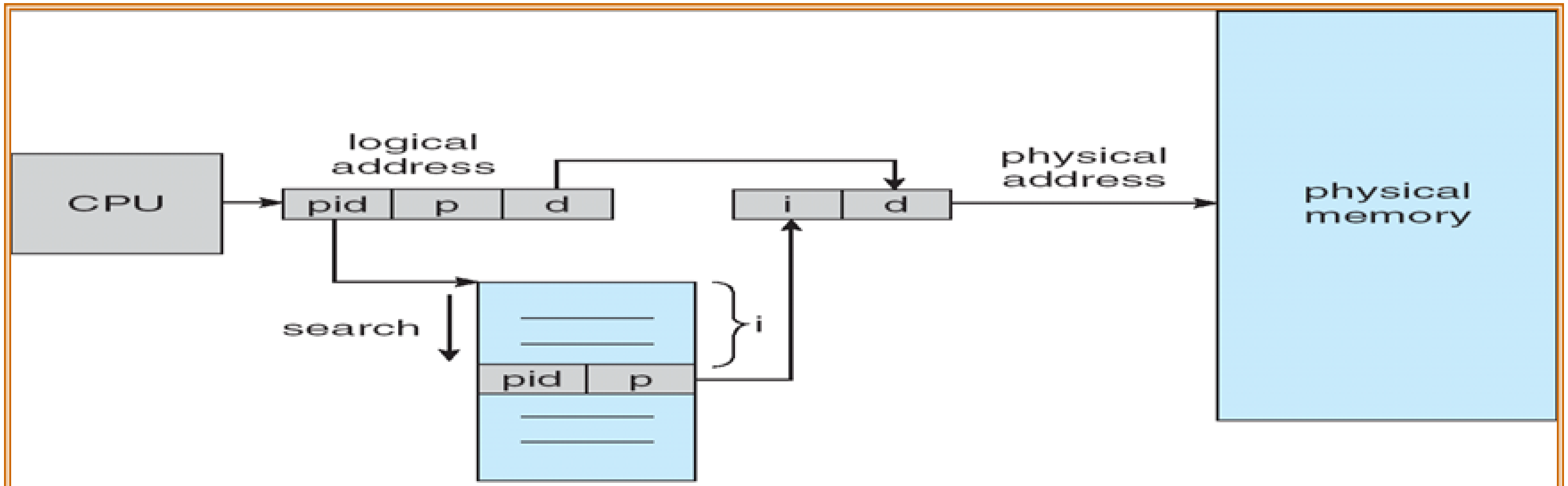


Hash Table

- Hash table with <pid, p> as hash key

Inverted Page Table

- The IPT does not contains complete information about the logical address space of a process and this information is necessary in case of a page fault.
- For this information to be available an external page table (one per a process) must be kept in memory.
- Referring to page tables may negate the benefit of the IPT. However,
  - These referenced page tables will occur only when a page fault occurs.
  - On demand paging, we need only the page table of the active process to be kept in memory by swapping it in.

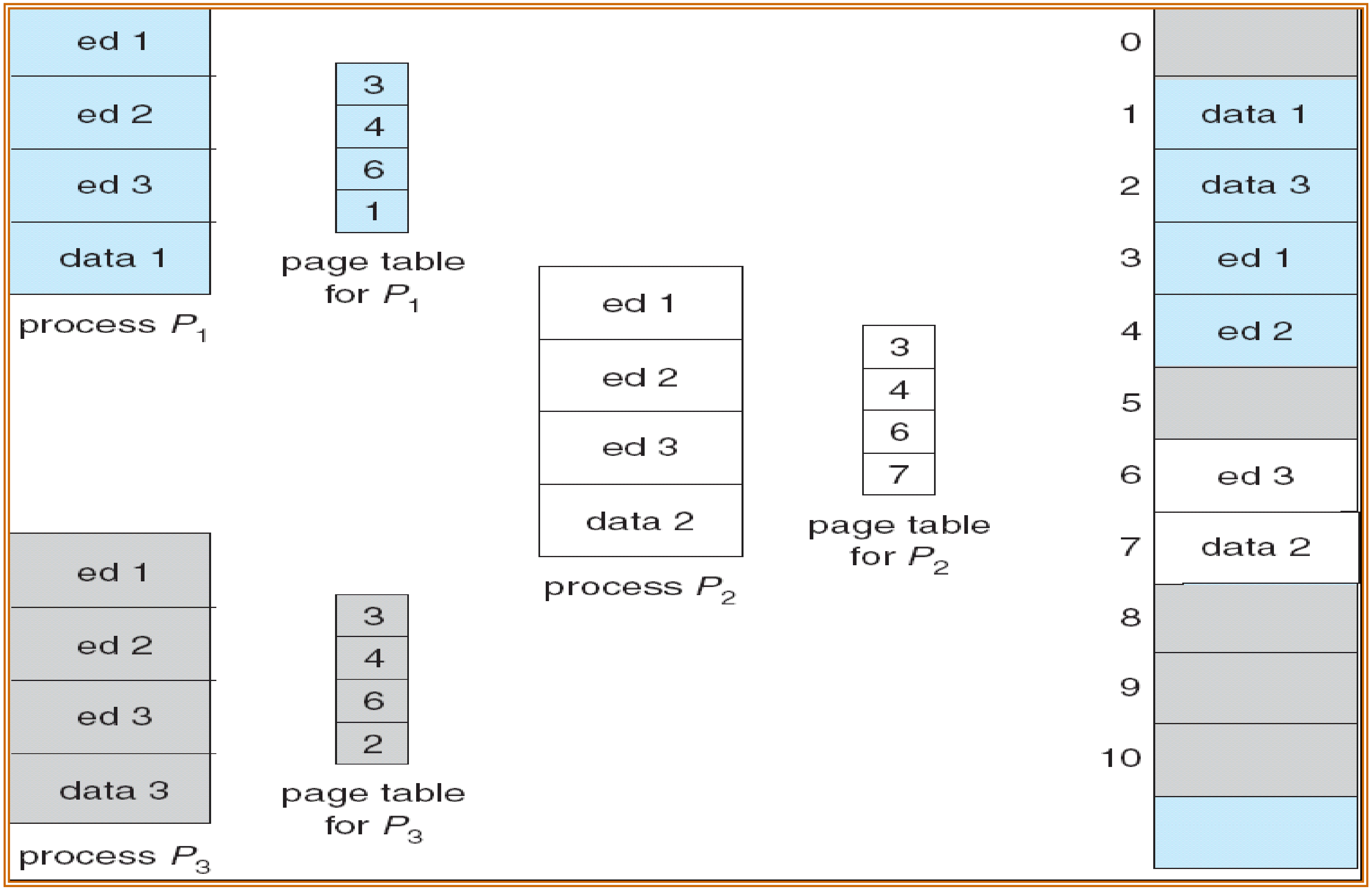


Inverted Page Table

Sharing of Pages

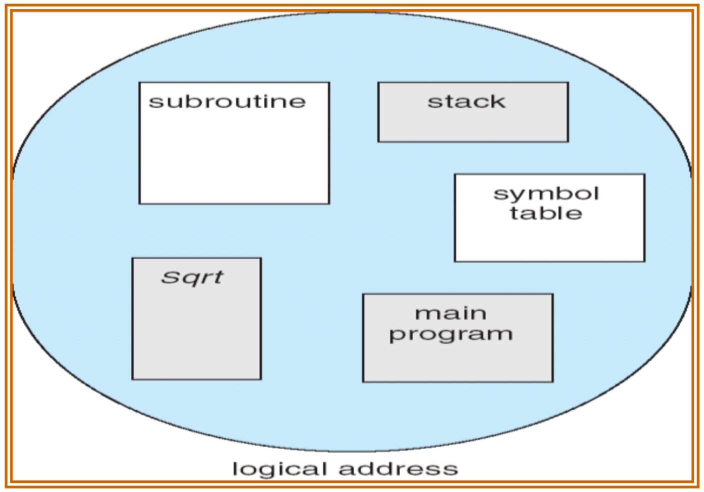
- As we have seen, paging allows the process to be non-contagiously loaded which reduces the fragmentations and improves the MM utilizations.
- One more advantage of paging is the possibility of sharing common pages, if we have 40 processes share a text editor of 150 KB, we can share one copy of the text editor rather than consuming 40*150 KB if we can not share it.
- Heavily used programs like Compilers, OSs, Editors, etc. can be shared.
- Shared Code
  - One copy of read-only (reentrant code, non-self-modifying) code can be shared among processes (i.e., text editors, compilers, OS).
  - The shared pages should be protected by OS (page protection).
  - Shared code must appear in the same location in the logical address space of all processes.
  - **Inverted page tables have difficulties implementing shared pages. why?**
- 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



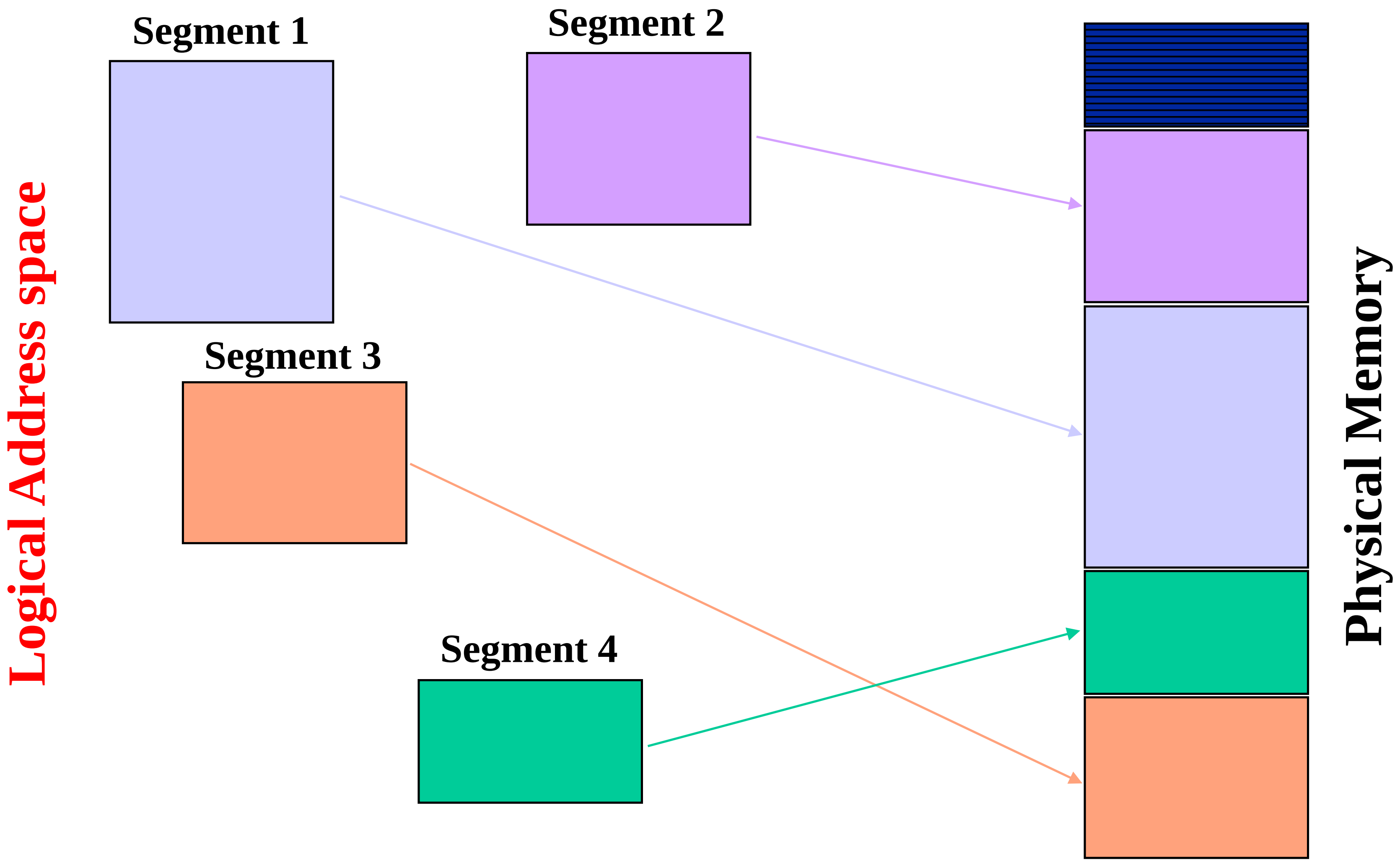
Segmentation

- **Split the Process address space into dynamic size segments.**
- Each segment is an independent, and separately-addressable unit.
- Every segment is assigned (**by software**) a base address, which is the starting address in the memory space.
- A program is a collection of segments. A segment is a logical unit with name and a length such as:
  - Main program (code section),
  - Subroutines,
  - Functions,
  - Local variables,
  - Global variables,
  - Stack,
  - Symbol table,
  - Arrays



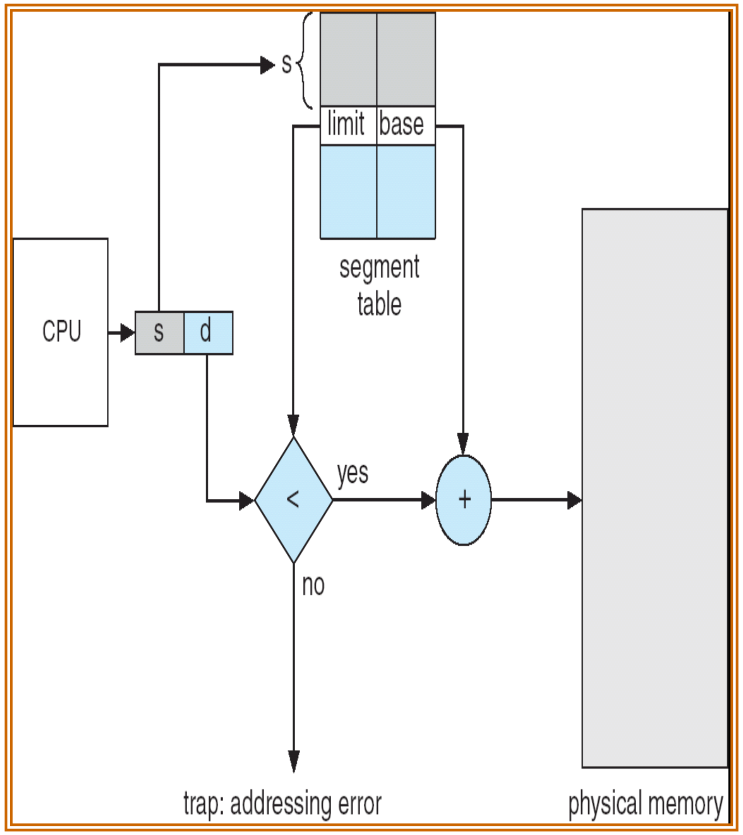
- The compiler automatically constructs segments reflecting the input program.
- Different compilers may create separate segments for global variables, stack, code portion of functions, etc…
- Segments are numbered and referred by number.

Memory Segmentation



Segmentation

1. If the segment is in main memory, the entry contains the starting address and the length of that segment.
2. Logical to physical address translation is similar to paging except that the offset is added to the starting address (instead of being concatenated).
3. Similarly to paging, each segment table entry contains a present bit and a modified bit.
4. Other control bits may be present if protection and sharing is managed at the segment level.



Virtual Address

Segment Number	Offset

P= present bit

M = Modified bit

Segment Table Entry

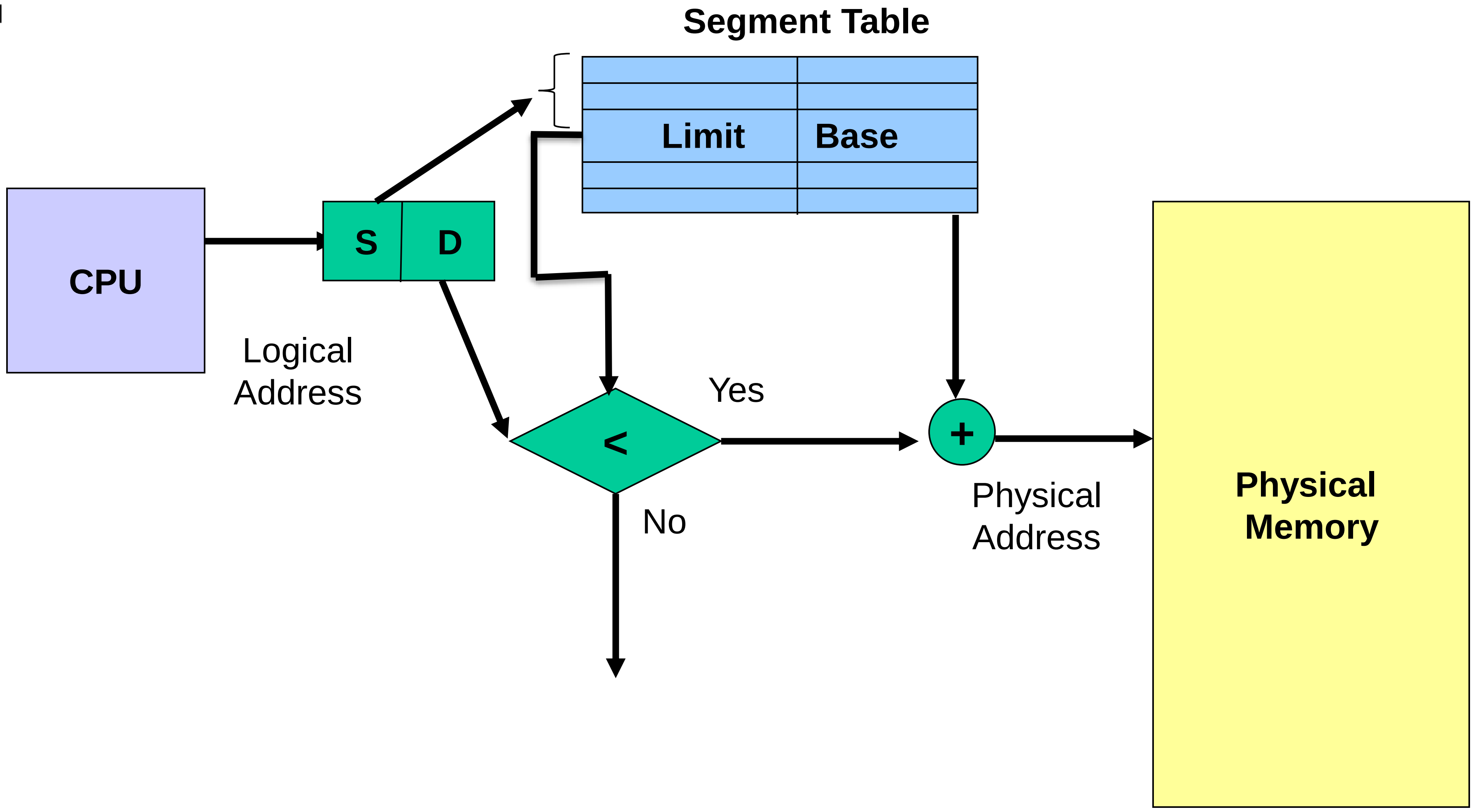
P	M	Other Control Bits	Length	Segment Base

Segmentation Architecture

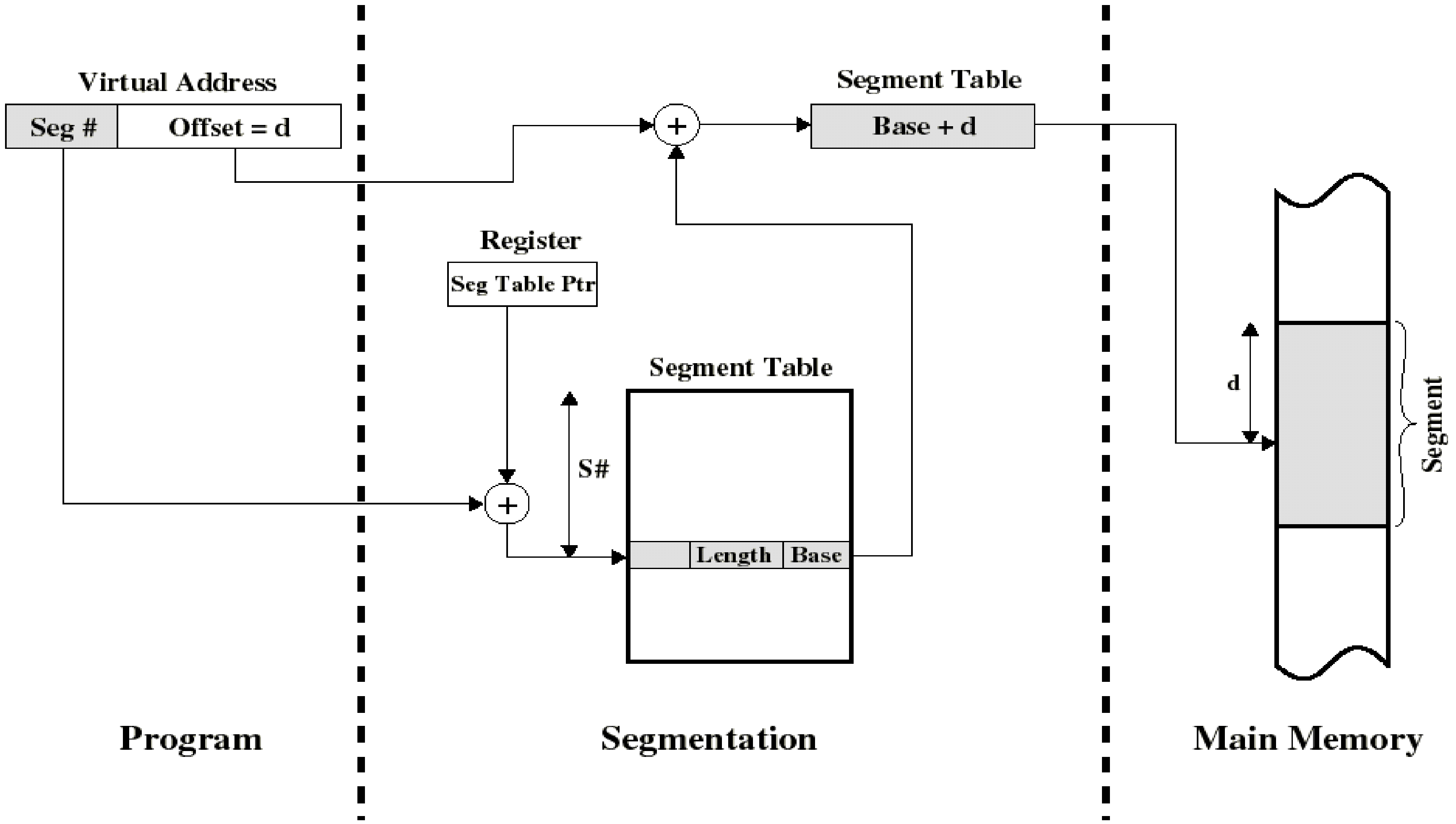
- Logical address consists of a two parts
  - <segment-number, offset>
- Segment table has:
  - Base: contains the starting physical address where the segment resides in memory.
  - Limit: specifies the length of the segment.
- Segment-table base register (STBR) points to the segment table's location in memory.
- Segment-table length register (STLR) indicates number of segments used by a program.
- Logical address consists of S [seg. Number] and D [offset into seg.]
  - Segment number S is legal if S < STLR
- S is used as an index into the segment table.
- D must be between 0 and the limit. If not, trap an error to the OS.
- If yes, **it is added to the segment base to produce the physical address**.

Segmentation Hardware

- The hardware must map a two dimensional (segment # and offset) into one-dimensional address.

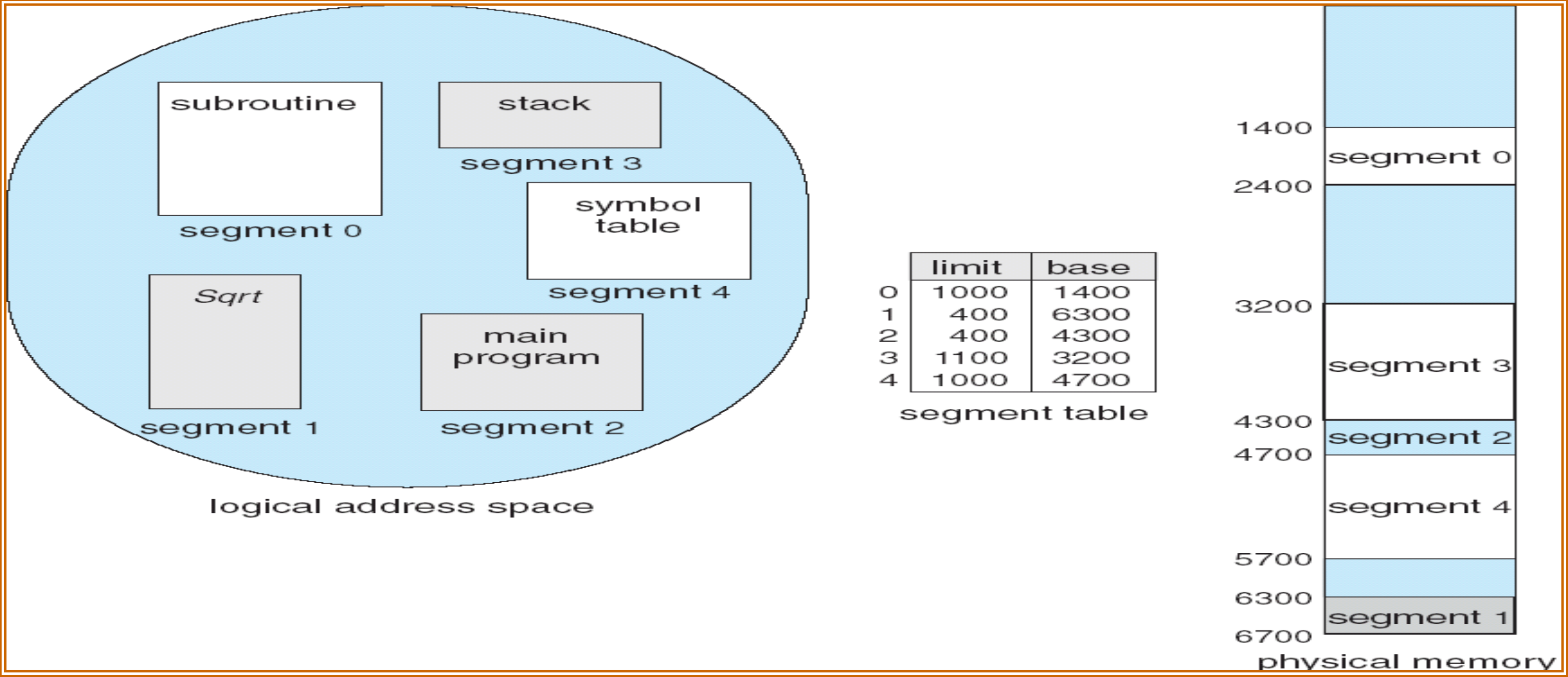


Address Translation in a Segmentation System



Example of Segmentation

1. We have 5 segments numbered 0 to 4. They stored in the physical memory .
2. The segment table has a separate entry for each segment, the base address and the limit.
3. Example, segment 2 is 400 bytes long and started at 4300.
4. A reference to byte 53 of segment 2 is mapped to location 4300 +53=4353.



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

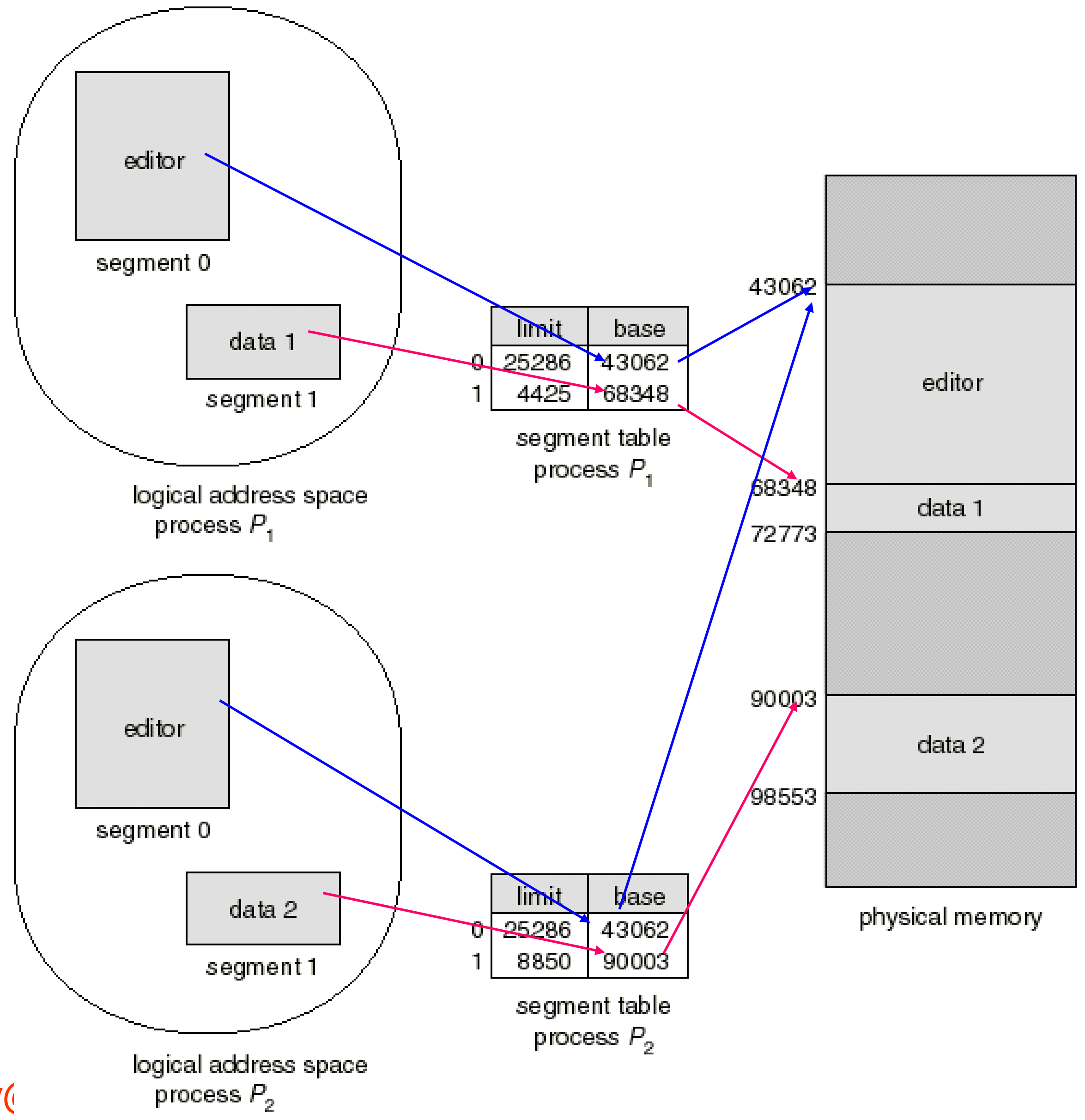
Protection and Sharing of Segments

- A particular advantage of segmentation is the association of protection with the segments, because the segments represent a semantically defined portion of the program, instructions, data.
- For example, the **instruction segment** can be defined as read only.
  - Because the instructions are non-self modifying,
- Useful protection bits in segment table entry:
  - Read-only/read-write bit
  - Supervisor/User bit
- The memory mapping HW will check the protection bits associated with each segment table entry to prevent illegal memory access.
- Another advantages of segmentation involves **the sharing of code or data**. Each process has a segment table associated with it.
- Note, local data segments can not be shared.

Sharing in Segmentation Systems

- Segments are shared when entries in the segment tables of 2 different processes point to the same physical locations.
- Ex: the same code of a text editor can be shared by many users
  - Only one copy is kept in main memory
- **Shared segments should not be modified**
  - So that several processes can share them
- So each user would still need to have its own private data segment
- More logical than sharing pages

Sharing of Segments: Text Editor Example

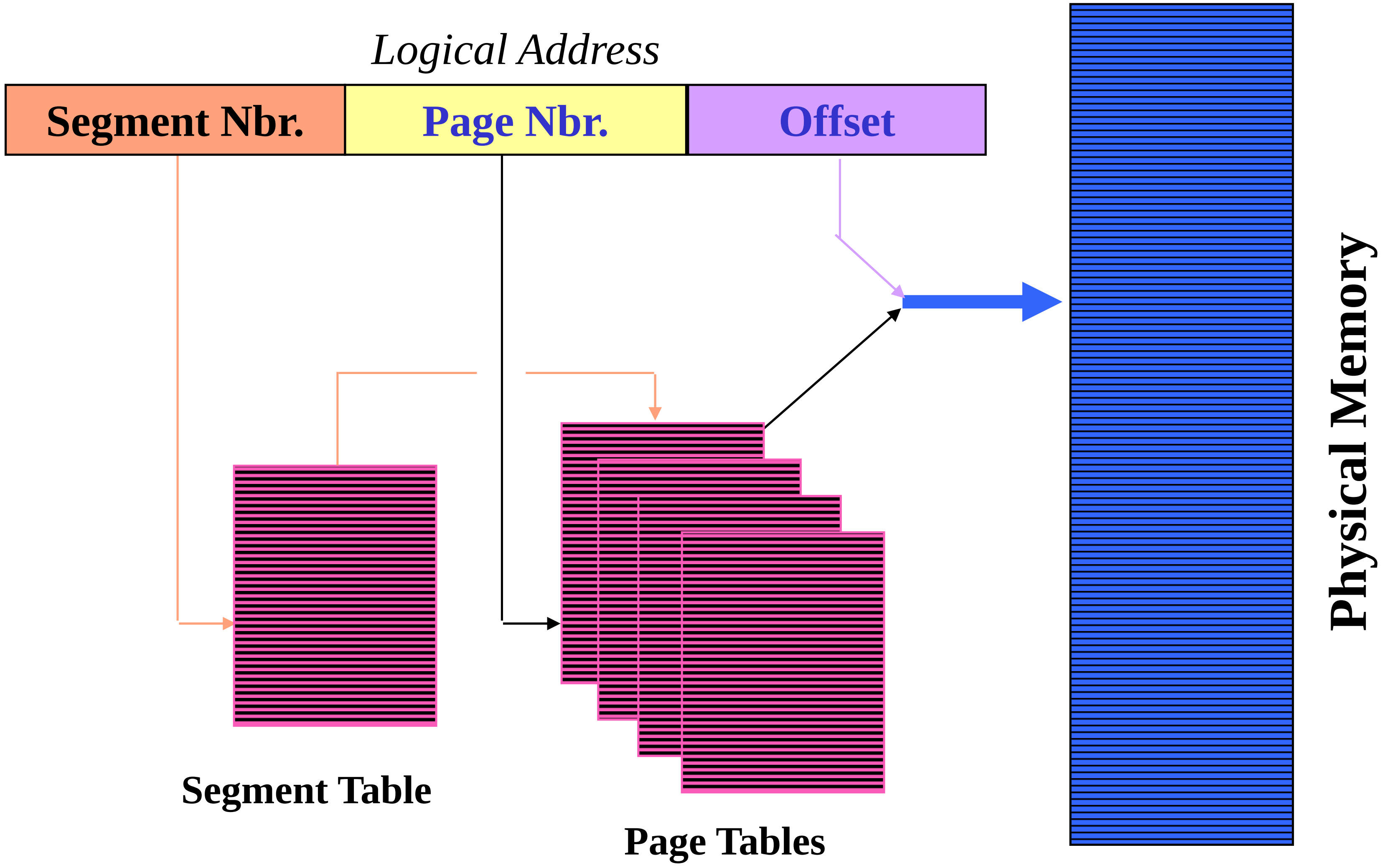


Combined Segmentation and Paging

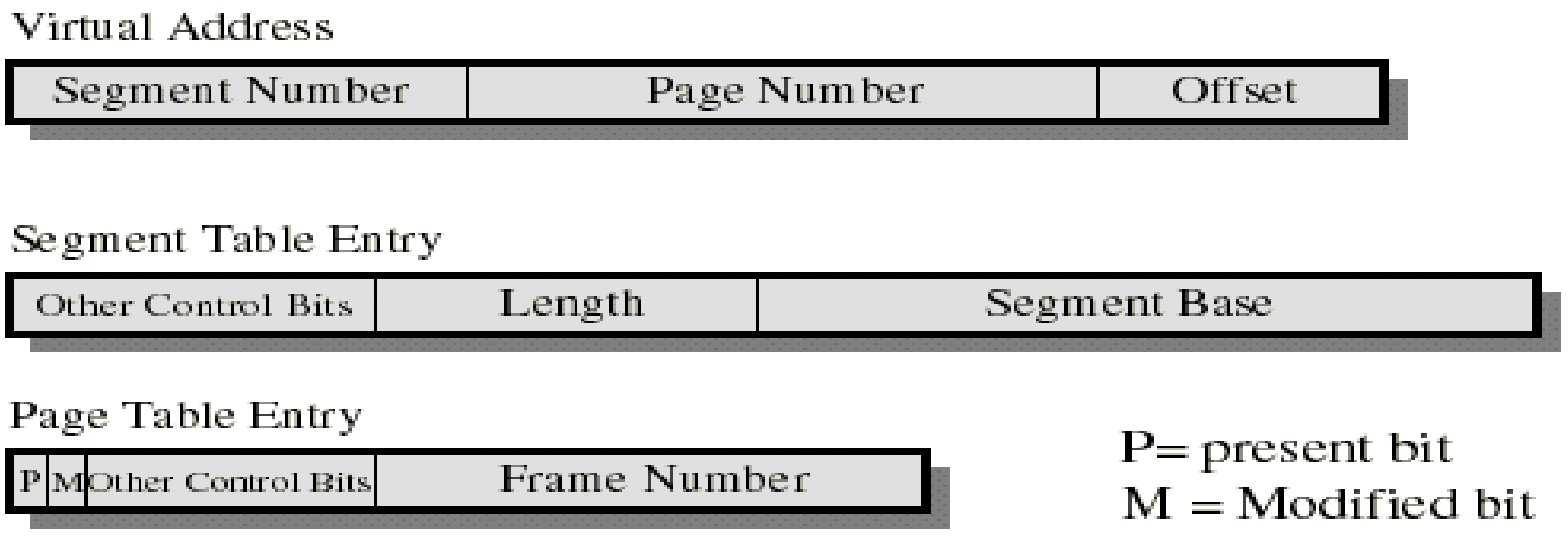
- To combine their advantages, some OSs page the segments into pages and use a page table per each segment. Why?
- Each process has:
  - One segment table
  - One page table per segment

- The virtual address consist of:
  - A segment number: used to index the segment table whose entry gives the starting address of **the page table for that segment**.
  - A page number: used to index that page table to obtain the corresponding **frame number.**
  - An offset: used to locate the word within the frame
- **Segment and page tables can themselves be paged**!

Segmentation & Pagination

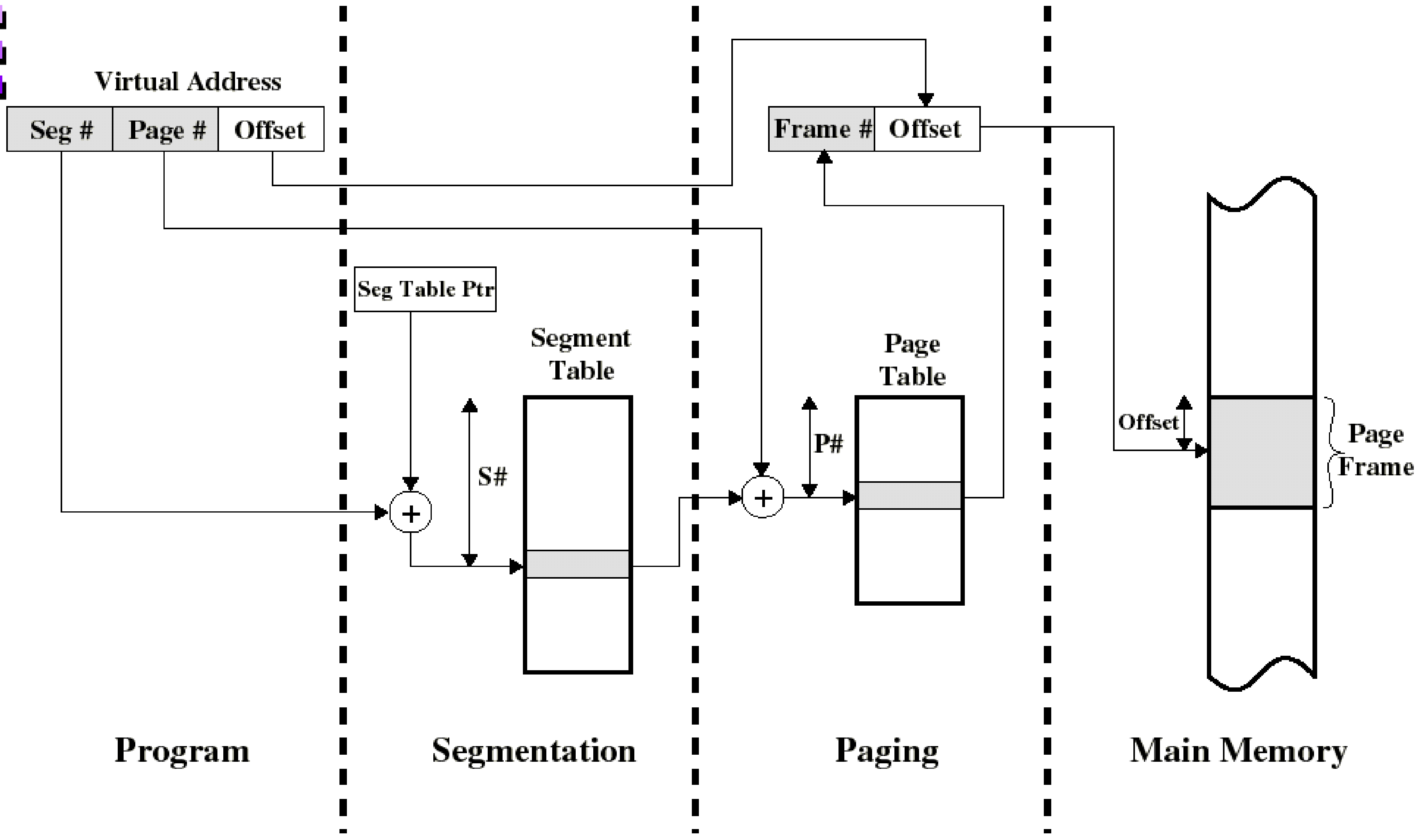


Simple Combined Segmentation and Paging



- The Segment Base is the physical starting address of that segment
- If the page and segment tables are paged, in the virtual address, the segment and page numbers are divided into two parts.
- Present and modified bits are present only in page table entry
- Protection and sharing info most naturally resides in segment table entry
  - Ex: a read-only/read-write bit, a kernel/user bit...

Address Translation in combined Segmentation/Paging System



Advantages of Segmentation + Paging

- Supports both dynamic loading and linking:
  - Linking a new segments needs to add a new entry to a segment table.
- Segments can grow without having to be moved in physical memory.
  - They just need more pages in physical memory.
- Protection and sharing can be done at the ‘logical’ segment level.
  - Pages inherit protection and sharing attributes of the segments to which they belong.

Memory Management Review

- Problem: to run large processes (larger than the available physical memory) and to increase the number of processes simultaneously loaded into the memory.
- Solution: Support non-contiguous allocation with partial loading, **Swapping out non needed processes**
  - Swapping problem:
    - Swapping time and quantum time
- Problem: The internal and external fragmentation.
  - Solution: Compaction or dividing the process memory space into smaller pieces (Paging, Segmentation or Paging of Segments).
- Problem: Managing the allocation and accessing of page or segment tables
  - Solutions: Using, caching, hierarchical page tables, hash tables, IPT.
- **We should think of a way to extend the MM by using a part of the auxiliary memory which will be called Virtual Memory.**

The End!!

Thank you

Any Questions?