Unit 4: Memory Management

Basics

- Program must be brought (from disk) into memory and placed within a process for it to be run.

- Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional from and storing to specific memory addresses. A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory.

- The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore how a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

Basic Hardware

- Main memory and the registers are the only storage that CPU can access directly.
- Registers are generally accessible within in one cycle of the CPU clock .
- Completing a memory access may take many cycles of the CPU clock. In such cases, the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory. A memory buffer used to accommodate a speed differential, called a cache..
- Protection of memory required to ensure correct operation. This protection must be provided by hardware.

Basic Hardware

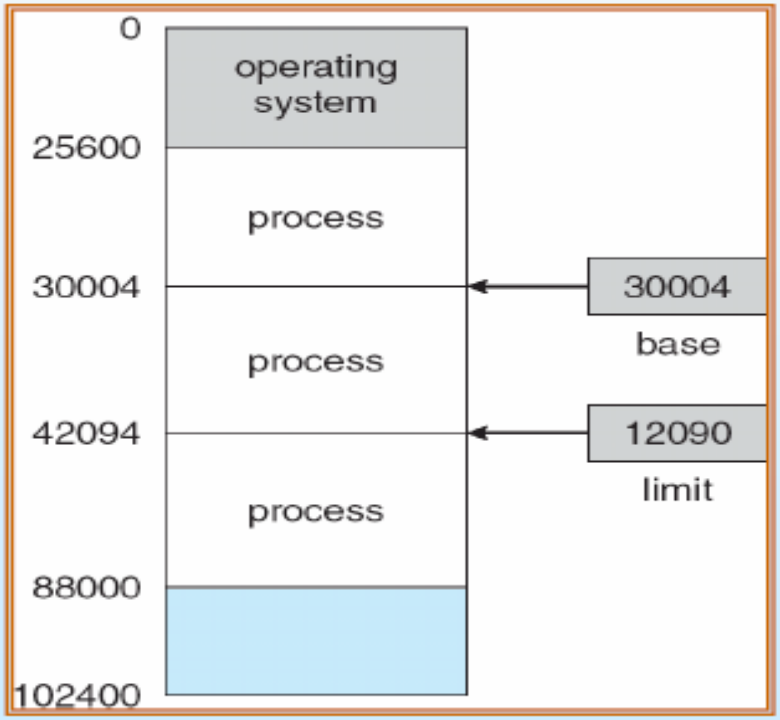
- In a multiprogramming system, in order to share the processor, a number of processes must be kept in memory.

- We need to make sure that each process has a separate memory space. To do this, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit.

- The base register holds the smallest legal physical memory address; the limit register specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

Base and Limit Registers

A pair of **base** and **limit** registers define the logical address space



- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error (Figure below). This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users..
- Only the operating system can load the base and limit registers. This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.
- The operating system, executing in kernel mode, is given unrestricted access to both operating system memory and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, and so on.

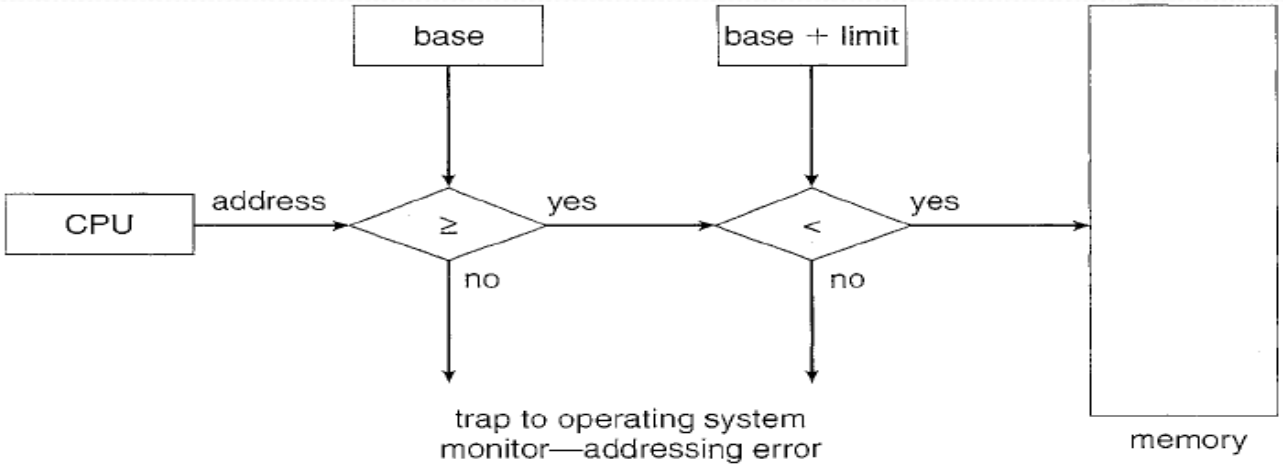
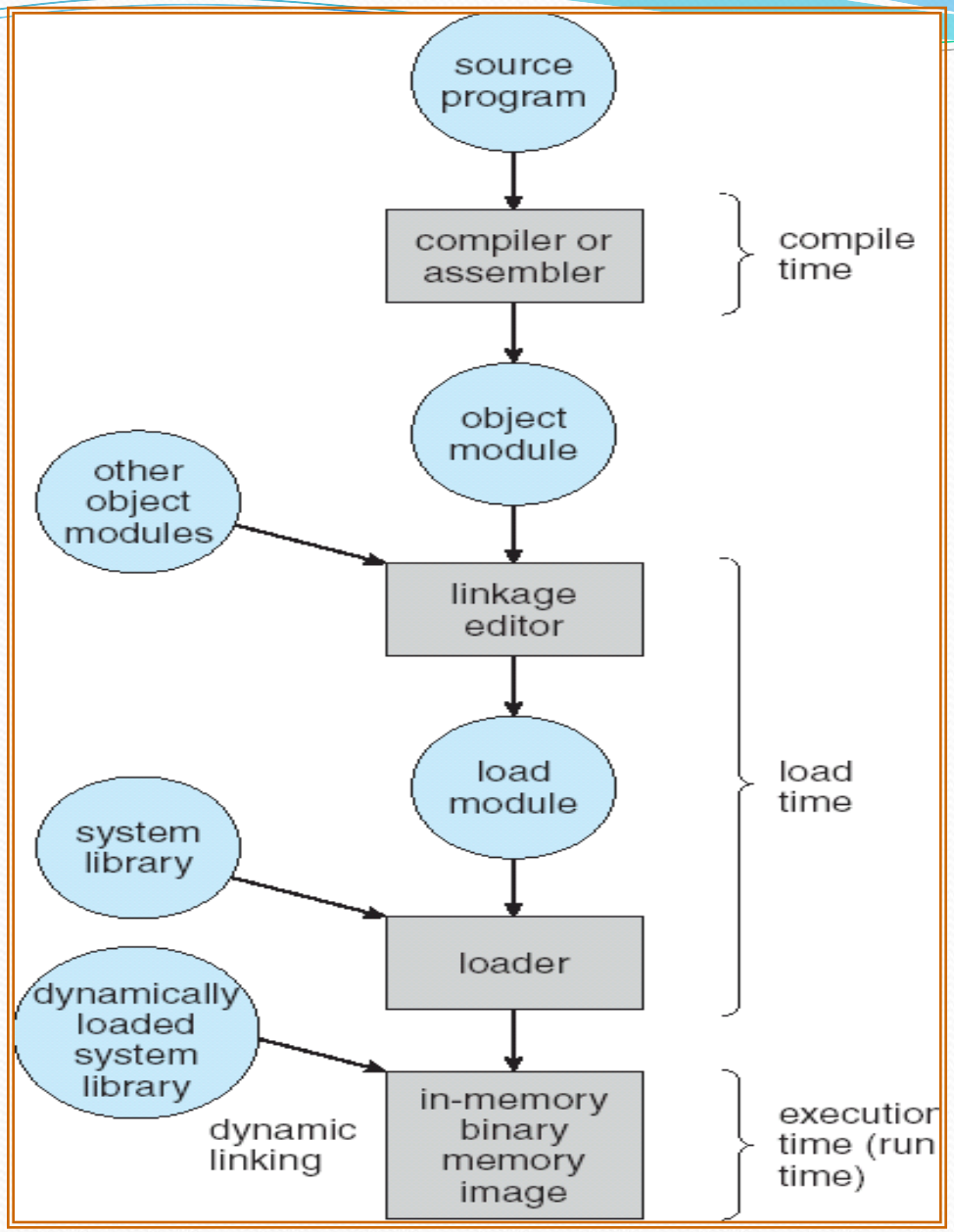


Figure 8.2 Hardware address protection with base and limit registers.

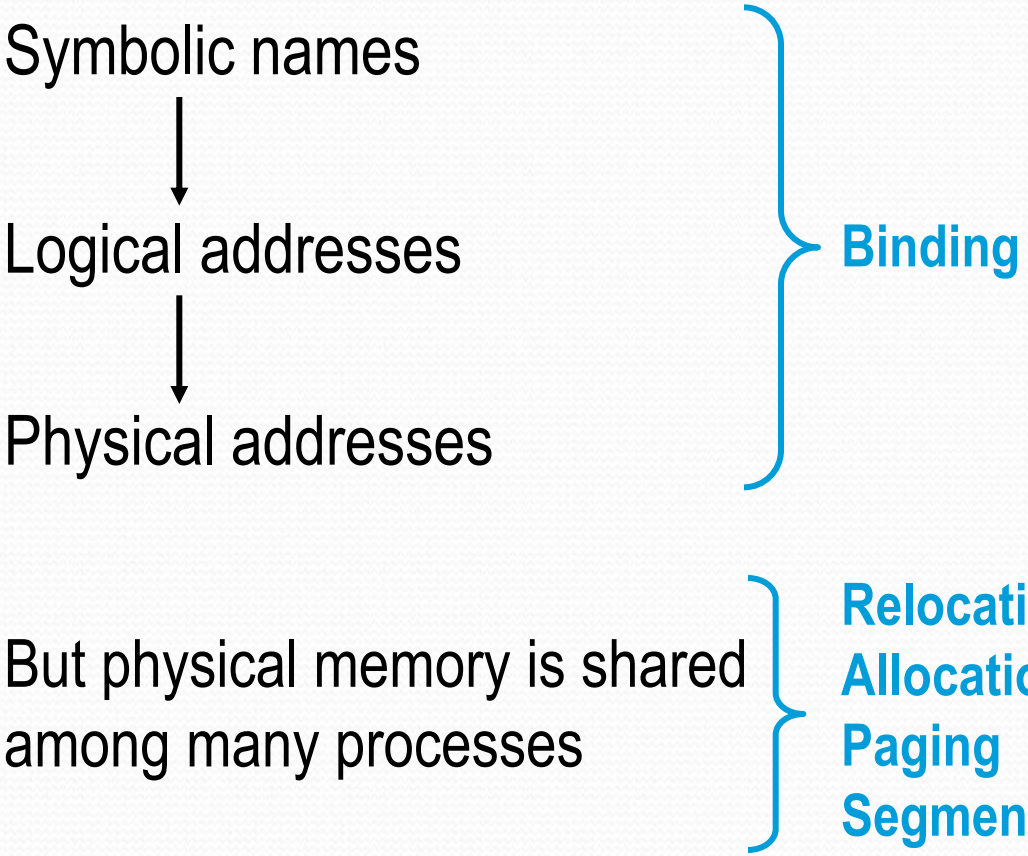
Address Binding

- Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the input queue.
- The normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available.
- Most systems allow a user process to reside in any part of the physical memory. (i.e. if the address space of the computer starts at ooooo, the first address of the user process need not be ooooo). This approach affects the addresses that the user program can use.
- Addresses may be represented in different ways during these steps (as shown in fig- next slide). Addresses in the source program are generally symbolic (such as count). A compiler will typically bind these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.


source program
compiler or assembler
compile time
object module
other object modules
linkage editor
load module
load time
system library
loader
dynamically loaded system library
dynamic linking
in-memory binary memory image
execution time (run time)

Address Binding

The Address Binding refers to the mapping of computer instructions and data to physical memory locations. Both logical and physical addresses are used in computer memory. It assigns a physical memory region to a logical pointer by mapping a physical address to a logical address known as a virtual address. It is also a component of computer memory management that the OS performs on behalf of applications that require memory access.



Binding of Instructions and Data to Memory addresses can be done at any step:

- **Compile time**: If memory location known a priori (compile time), **absolute code** can be generated; must recompile code if starting location changes

- **Load time**: Must generate **relocatable code** if memory location is not known at compile time. In this case, final binding is delayed until load time. If the starting address changes, we need to only reload the user code to incorporate this changed value.

- **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.

Names and Binding

- **Symbolic names → Logical names → Physical names**
  - **Symbolic Names:** known in a context or path
    - file names, program names, printer/device names, user names
  - **Logical Names:** used to label a specific entity
    - inodes, job number, major/minor device numbers, process id (pid), uid, gid..
  - **Physical Names:** address of entity
    - inode address on disk or memory
    - entry point or variable address
    - PCB address

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 that is, the one loaded into the memory-address register of the memory
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses are different in execution-time address-binding scheme
- The set of all logical addresses generated by a program is a logical address space; the set of all physical addresses corresponding to these logical addresses is a physical address space. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

Memory-Management Unit (MMU)

- The run time mapping from virtual to physical address is done by a hardware device called the Memory-Management Unit (MMU)
- In MMU scheme, the value in the relocation register (base 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. The memory-mapping hardware converts logical addresses into physical addresses.
- We now have two different types of addresses: logical addresses (in the range o to max) and physical addresses (in the range R+o to R+max for a base value R). The user generates only logical addresses and thinks that the process runs in locations o to max. However, these logical addresses must be mapped to physical addresses before they are used. The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

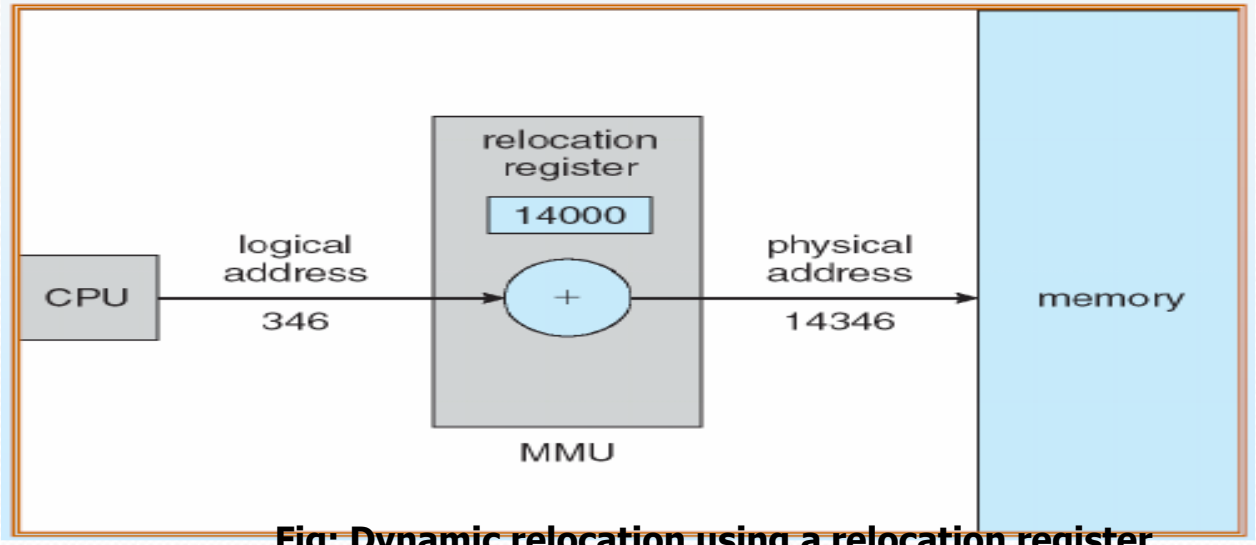
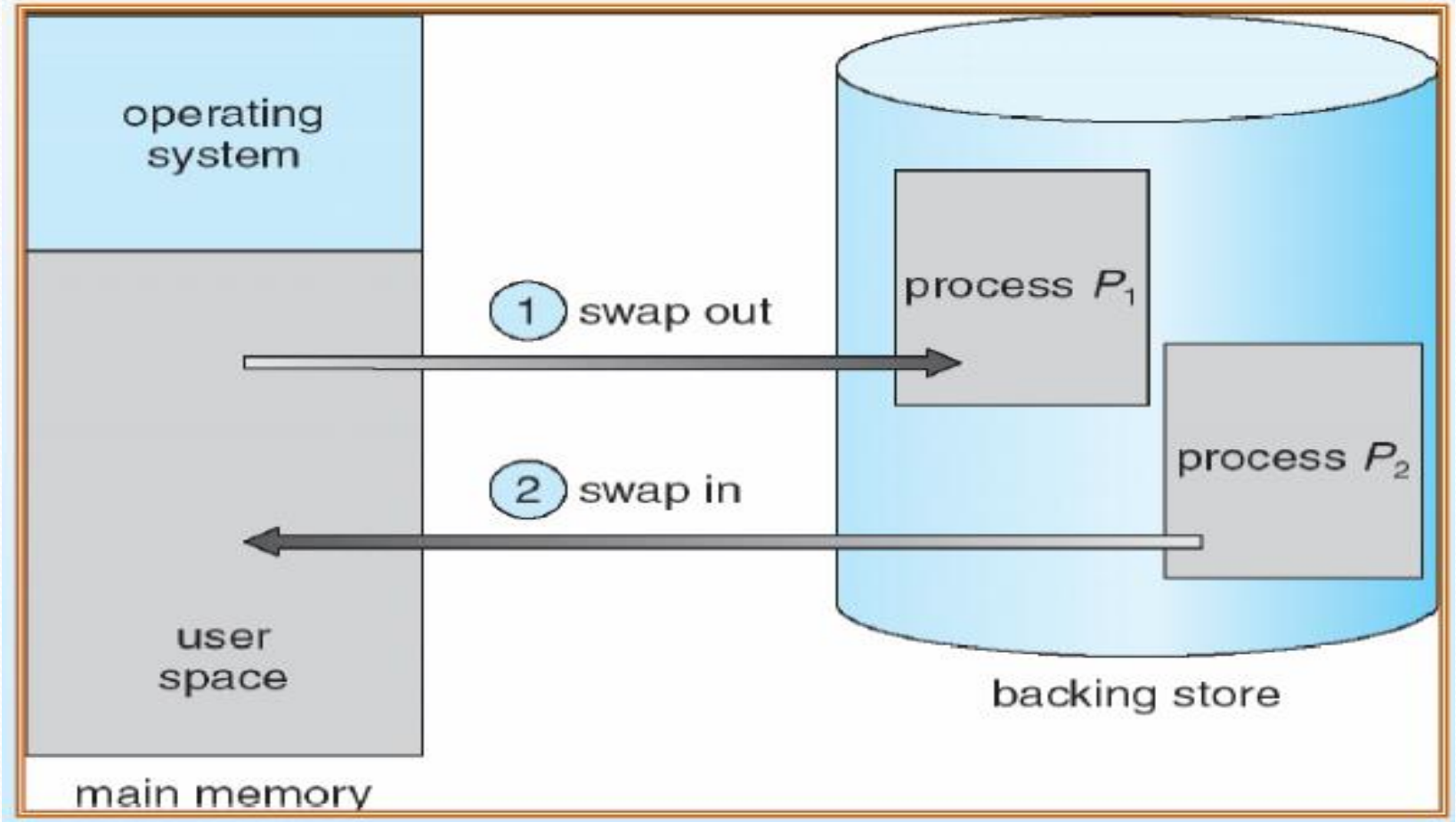


Fig: Dynamic relocation using a relocation register

Swapping

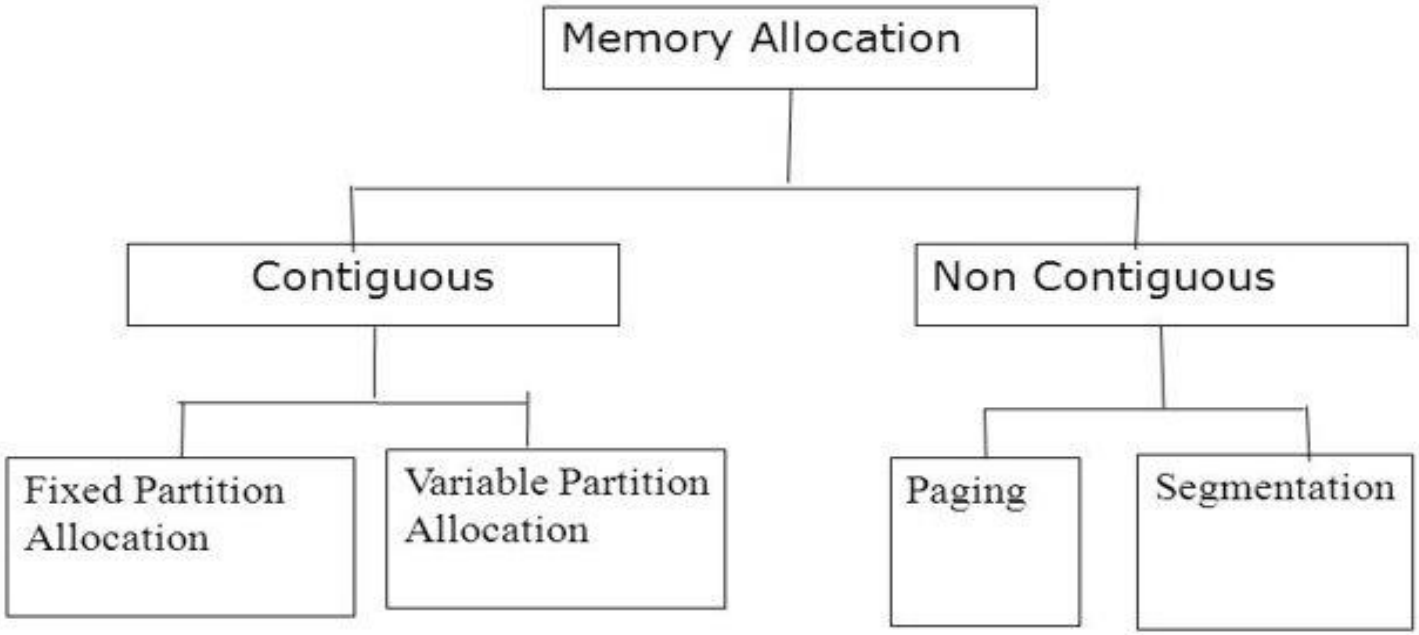
- A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution
- For example, RR CPU-scheduler, Priority scheduler



Swapping

- Swapping requires a backing store. The backing store is commonly a fast disk
- It must be large enough to store all memory images for all users, and it must provide a direct access to these memory images
- The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run
- Whenever the CPU scheduler decides to execute a process, it calls the dispatcher.
- The dispatcher will check whether the next process is in the memory.
- If not, and if there is no free memory region, the dispatcher will swap out a process currently in memory and swaps in the desired process.

Memory Allocation Techniques



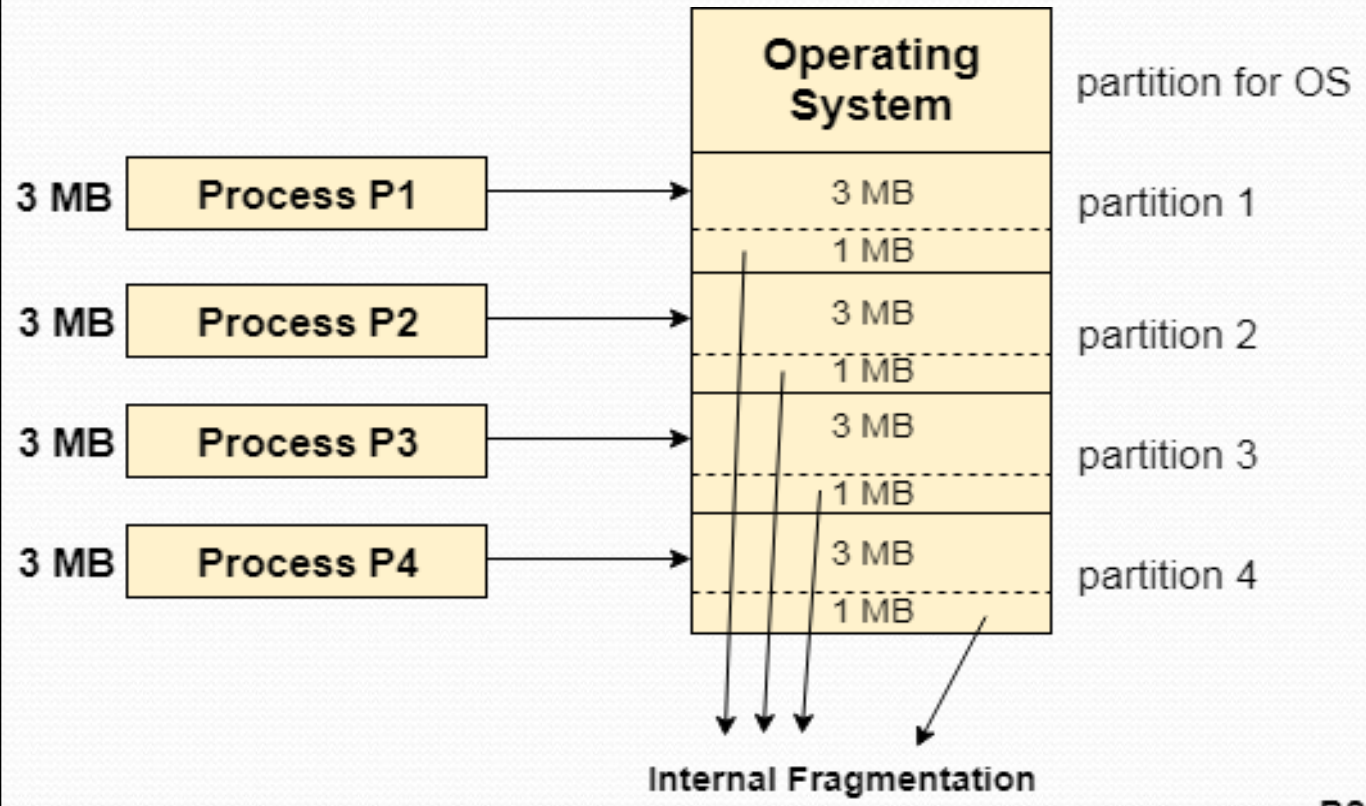
Contiguous Allocation

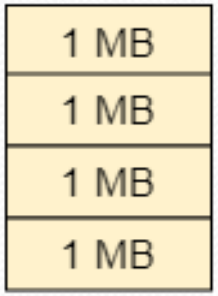
- Main memory usually divided into two partitions: one for the resident operating system and one for the user processes
- Resident operating system, usually held in low memory with interrupt vector.
- User processes then held in high memory.
- In. contiguous memory allocation, each process is contained in a single contiguous section of memory.

Fixed partitioning

- The simplest method for memory allocation 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
- Initially, all memory is available for user processes and is considered one large block of available memory, a **hole.**
- In this 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

Fixed partitioning





Unused Space

(External Fragmentation)

Process P5 **4 MB**

PS can't be executed even though there is 4 MB space available but not contiguous.

Fixed Partitioning

(Contiguous memory allocation)

Limitations of Fixed partitioning

- **1. Internal Fragmentation**
- If the size of the process is lesser then the total size of the partition then some size of the partition get wasted and remain unused. This is wastage of the memory and called internal fragmentation.
- As shown in the image below, the 4 MB partition is used to load only 3 MB process and the remaining 1 MB got wasted.

- **2. External Fragmentation**
- The total unused space of various partitions cannot be used to load the processes even though there is space available but not in the contiguous form.
- As shown in the image below, the remaining 1 MB space of each partition cannot be used as a unit to store a 4 MB process. Despite of the fact that the sufficient space is available to load the process, process will not be loaded.

Dynamic partitioning

In the variable-partition 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

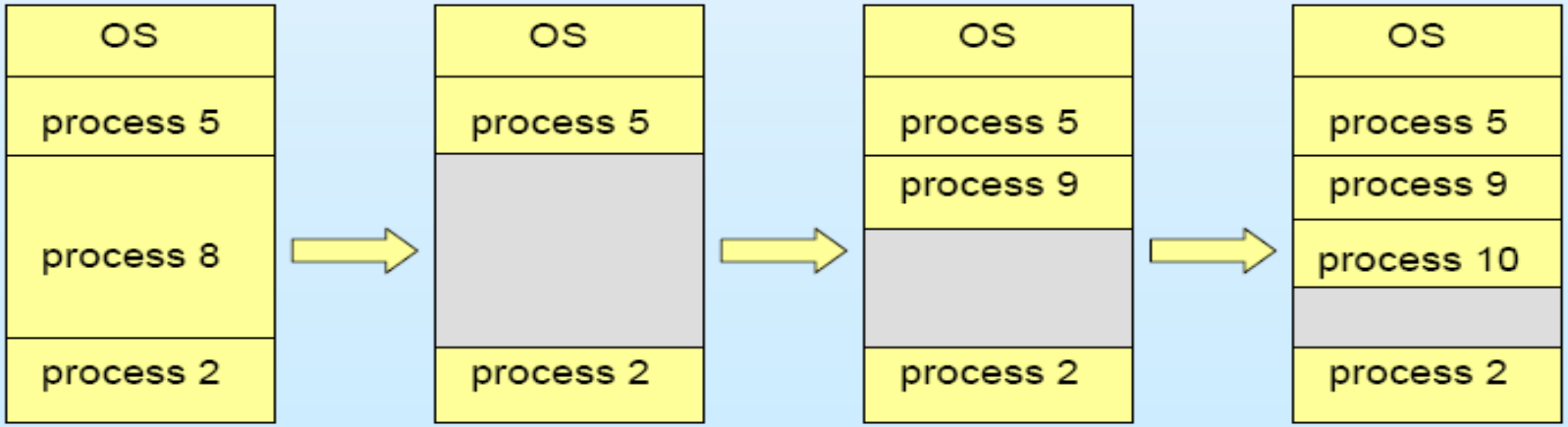
When a process arrives and needs memory, the system searches the set for a hole that is large enough for it.

If it is too large, the space divided into two parts. One part is allocated to the process and another part is freed to the set of holes.

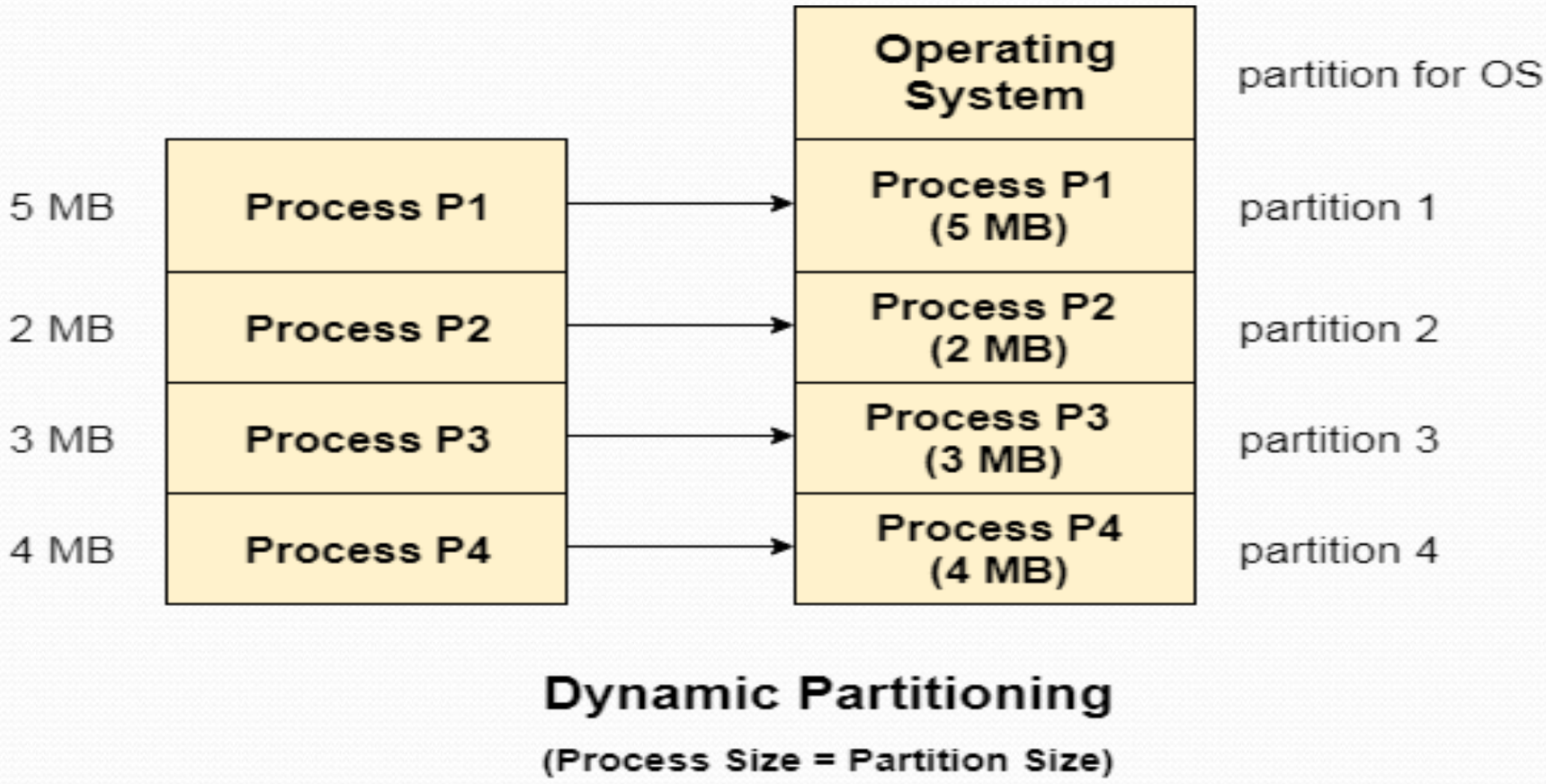
When the process terminate, the space is placed back in the set of holes.

If the space is not big enough, the process wait or next available process comes in.

- Operating system maintains information about:
  a) allocated partitions b) free partitions (hole)



Dynamic Partitioning



Dynamic Partitioning

(Process Size = Partition Size)

Dynamic Storage- Allocation Problem

- 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.

- **First-fit**: Allocate the *first* hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough

- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless the list is ordered by size (Produces the smallest leftover hole).

- **Worst-fit**: Allocate the *largest* hole; must also search entire list, unless sorted by size (Produces the largest leftover hole).

- Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

Advantages of Dynamic Partitioning

1. No internal fragmentation

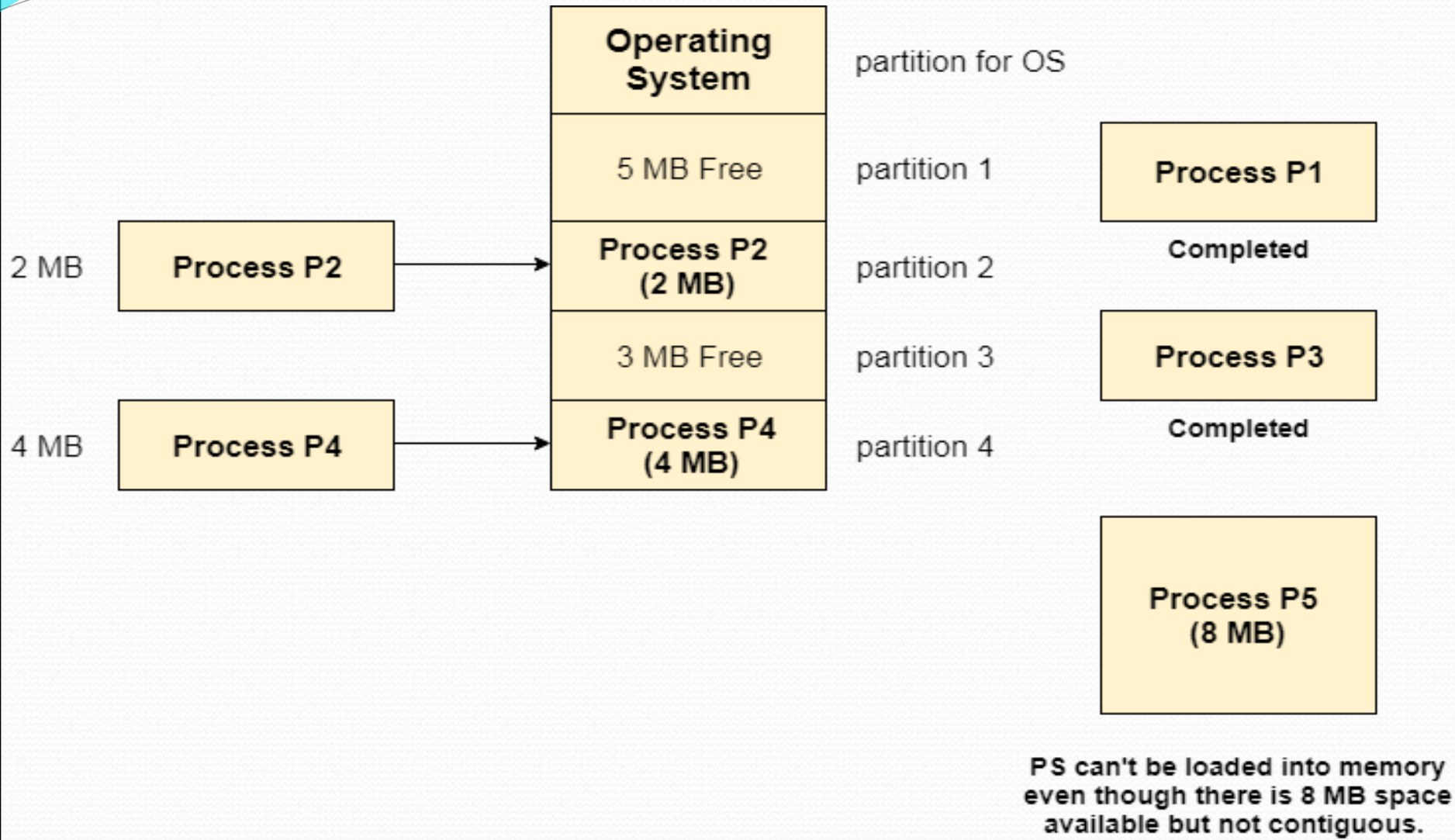
- **Disadvantages :**

1. **External Fragmentation :**

- Let's consider three processes P1 (1 MB) and P2 (3 MB) and P3 (1 MB) are being loaded in the respective partitions of the main memory.

- After some time P1 and P3 got completed and their assigned space is freed. Now there are two unused partitions (1 MB and 1 MB) available in the main memory but they cannot be used to load a 2 MB process in the memory since they are not contiguously located.

External Fragmentation in Dynamic partitioning



External Fragmentation in Dynamic Partitioning

Fragmentation

- All strategies for memory allocation suffer from **external fragmentation.**

- **external fragmentation:** as process are loaded and removed from memory, the free memory space is broken into little pieces.

- External fragmentation exists when there is enough total memory space to satisfy the request, but available spaces are not contiguous, storage is fragmented into a large number of small holes

Solution to external fragmentation problem

- One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done; compaction is possible *only* if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost.
- The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.
- Another possible solution to external-fragmentation problem is to permit the logical address space of the process to be **noncontiguous**.
- Thus, allowing a process to be allocated physical memory wherever the space is available.
- Two complementary techniques achieves this solution: paging and segmentation.

Non Contiguous---Paging

- **Paging** is a memory-management scheme that permits the physical address space of a process to be non-contiguous. This mechanism allows OS to retrieve processes from the secondary storage into the main memory in the form of pages

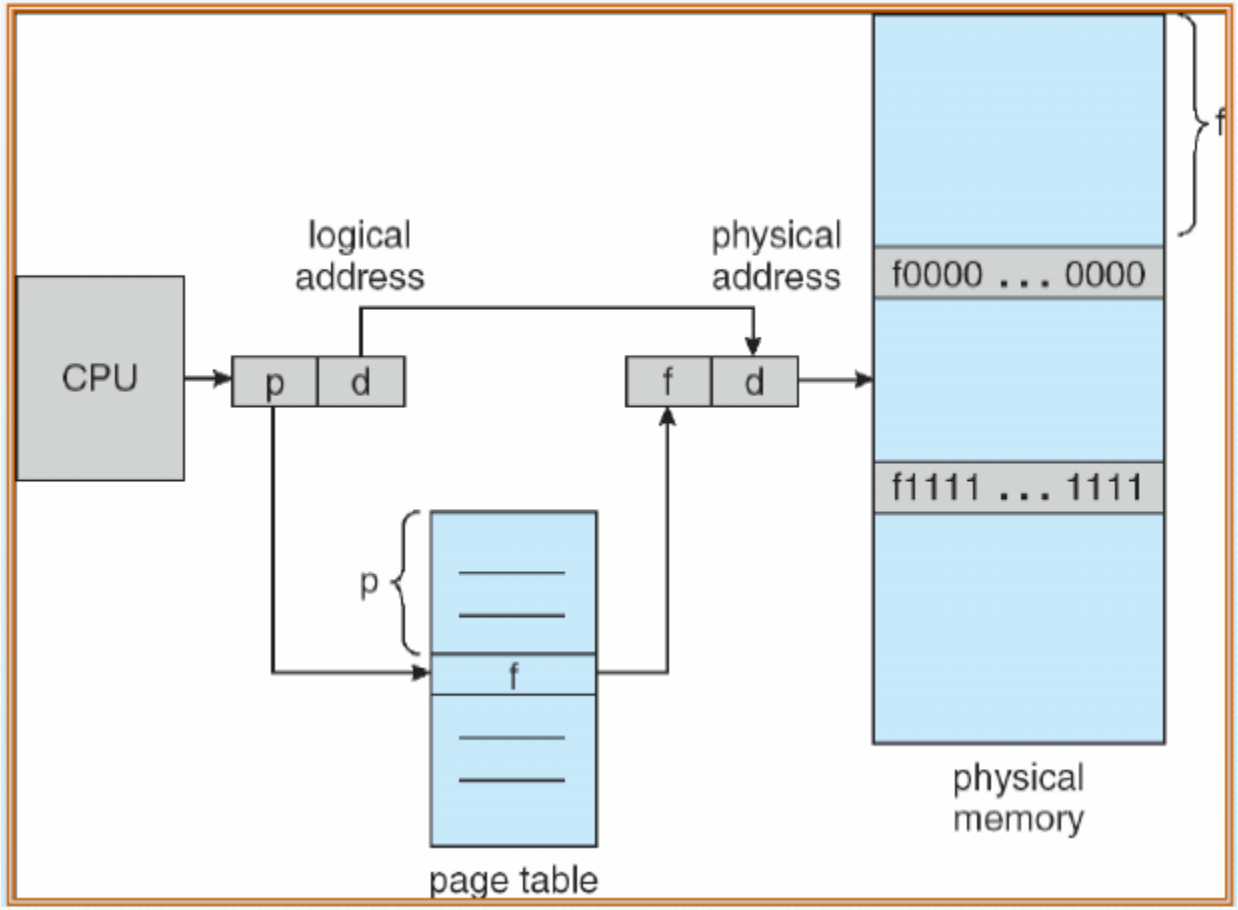
- The basic method for implementation involves breaking physical memory into fixed-sized blocks called **FRAMES** and break logical memory into blocks of the same size called **PAGES**

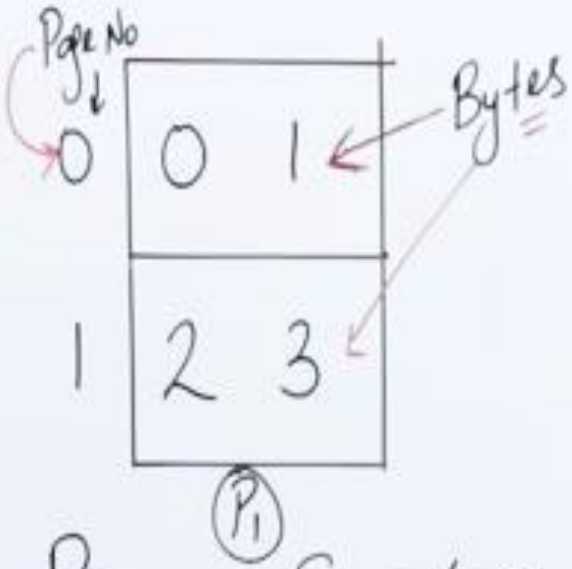
- Frames = physical blocks
- Pages = logical blocks
- Size of frames/pages is defined by hardware (power of 2 to ease calculations

Paging

- Every address generated by the CPU is divided into two parts: **Page number (p)** and **Page offset (d)**
- The page number is used as an index into a Page Table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.
- A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses

Paging

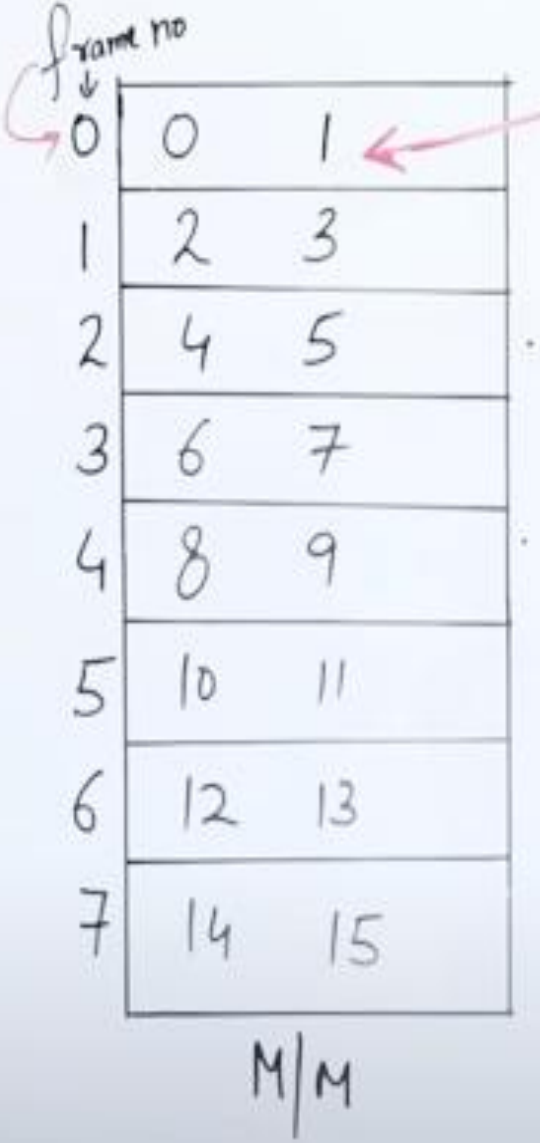

logical
address
physical
address
CPU
p
d
f
d
f0000 . . . 0000
f1111 . . . 1111
f
p
f
physical
memory
page table



Process Size = 4B

Page Size = 2B

No. of Pages / Process = $\frac{4B}{2B}$ = (2)

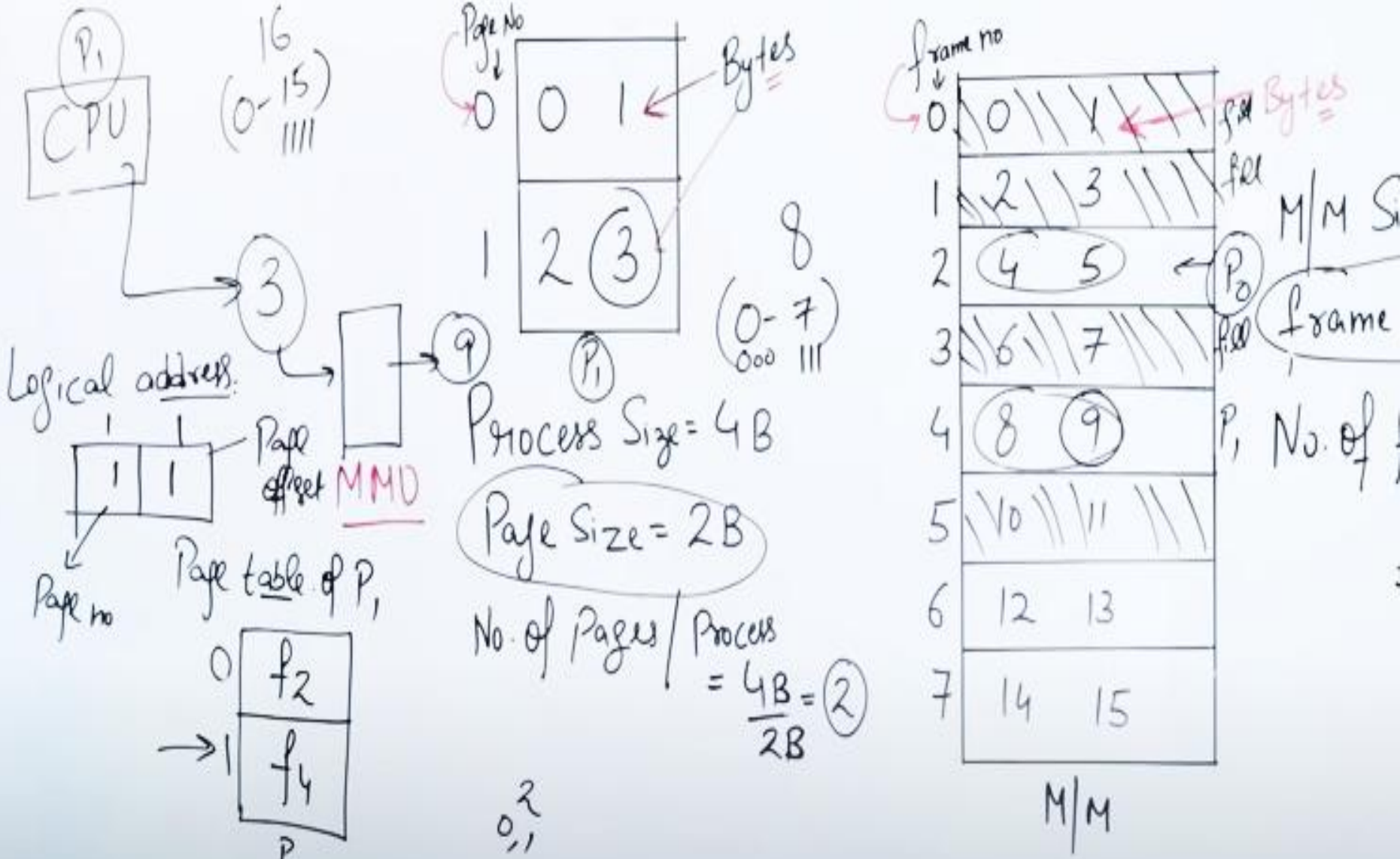


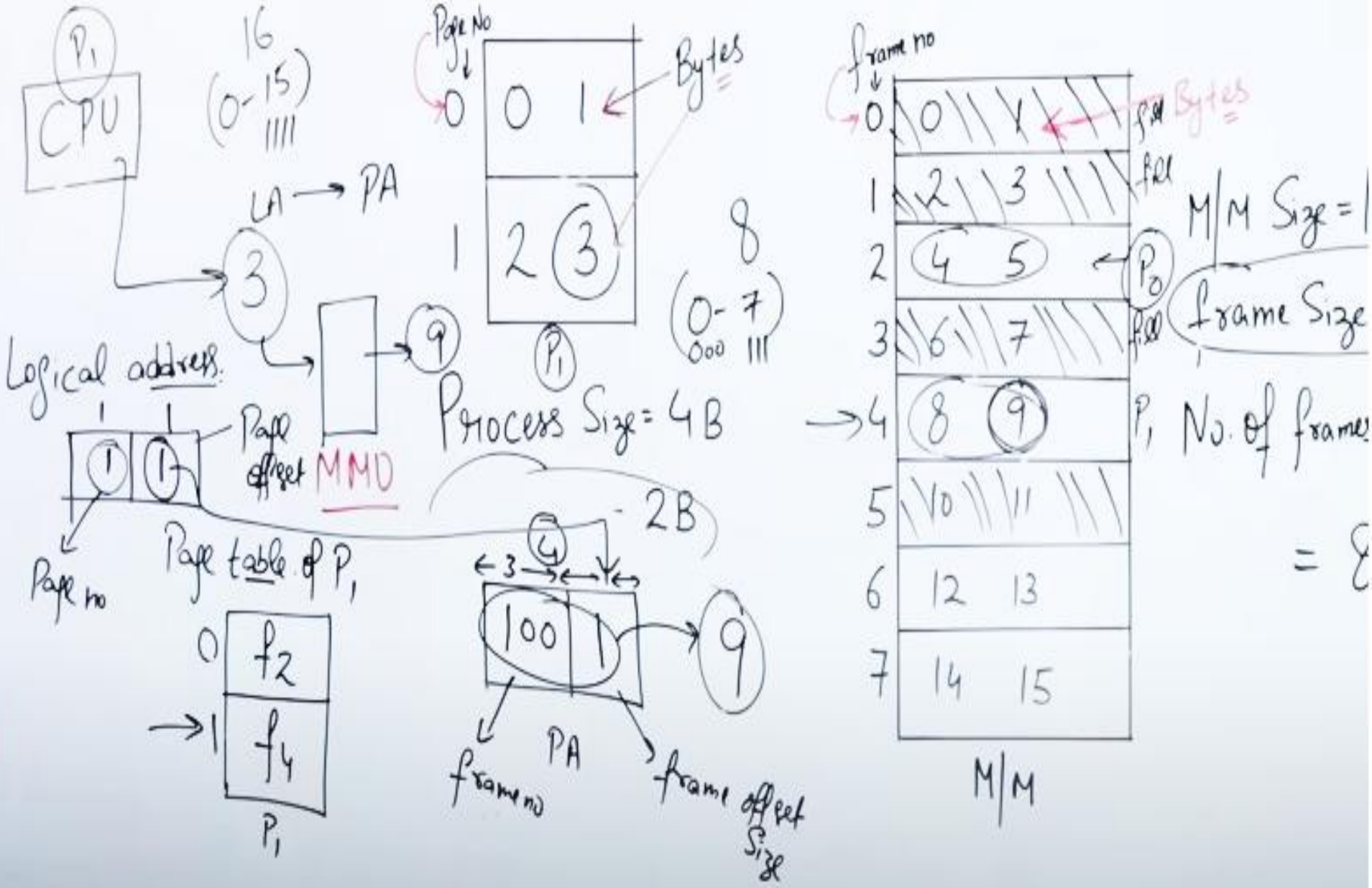
M/M Size = 16B

frame Size = 2B

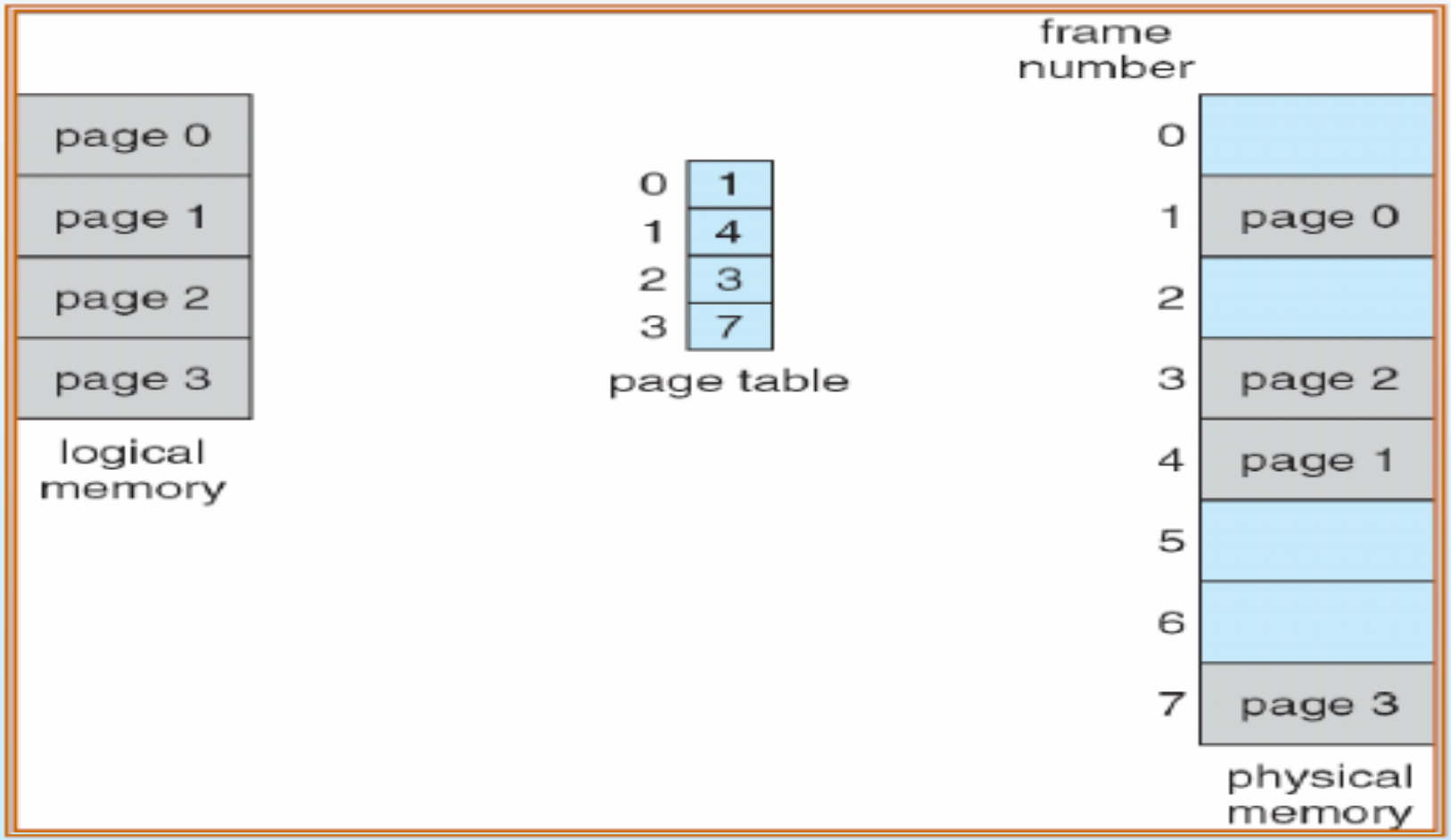
No. of frames = $\frac{16B}{2B}$

= 8 frames


P1
CPU
16
(0-15)
1111
3
Logical address
1
1
Page no
Page offset
MMU
9
Page table of P1
0
f2
1
f4
P1
Page No
0
0
1
Bytes
1
2
3
P1
8
(0-7)
000
111
Process Size = 4B
Page Size = 2B
No. of Pages / Process
= 4B / 2B = 2
frame no
0
0
1
1
2
3
2
4
5
P0
3
6
7
4
8
9
P1
5
10
11
6
12
13
7
14
15
M/M
Bytes
M/M Si
frame
P1 No. of

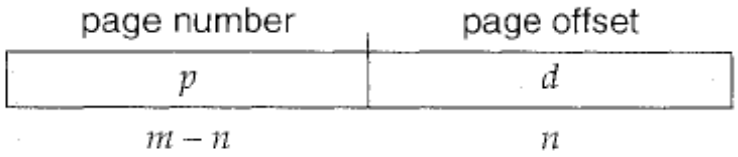

P1
CPU
16
(0-15)
1111
LA → PA
3
9
Logical address
1
1
Page offset
MMD
Page no
Page table of P1
0
f2
1
f4
P1
Page No
0
0
1
Bytes
1
2
3
P1
8
(0-7)
000
111
Process Size = 4B
2B
4
3
100
1
9
PA
frame no
frame offset
Size
frame no
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
P0
P1
M/M
M/M Size =
frame Size
P1 No. of frames
=

Paging


frame number
page 0
page 1
page 2
page 3
logical memory
0 1
1 4
2 3
3 7
page table
0
1 page 0
2
3 page 2
4 page 1
5
6
7 page 3
physical memory

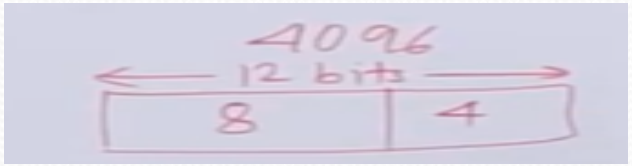
Paging

The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of the logical address space is 2^m, and a page size is 2^n addressing units (bytes or words then the high-order (m- n) bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:



where p is an index into the page table and d is the displacement within the page.

Let we have a particular space of 4096 bytes, to address this space, 12 bits are required (because 2^{12}=4096). Divide this into 8 and 4 as:



So it means that we will have 2^8 = 256 pages in page table and each page will have 2^4=16 locations. Go to the pth location and get the frame no then frame no and displacement will be clubbed to give the physical memory address.

How to select page size?

Advantages of big pages:

- Page table size is smaller
- Frame allocation accounting is simpler

Advantages of small pages:

- Internal fragmentation is reduced

Paging

- When we use a paging scheme, we have no external fragmentation: ANY free frame can be allocated to a process that needs it.

Paging

- However, we may have internal fragmentation.

- Suppose we have a memory system that uses paging with fixed-size pages of 4 KB each. When a program requests memory allocation, it may not always request memory in multiples of 4 KB.
- Let's say a program requests memory for three data blocks:

  Block A: Requires 6 KB of memory.

  Block B: Requires 5 KB of memory.

  Block C: Requires 2 KB of memory.

  To accommodate these blocks, the system assigns them to pages:

  Block A: Assigned to two pages (4 KB + 4 KB), resulting in 2 KB of internal fragmentation within the second page.

  Block B: Assigned to two pages (4 KB + 4 KB), resulting in 3 KB of internal fragmentation within the second page.

  Block C: Assigned to one page (2 KB), fully utilizing the allocated page.

  In this example, even though the allocated memory satisfies the program's requirements, internal fragmentation occurs within the allocated pages due to the mismatch between the block sizes and the page size. As a result, some space within the pages remains unused, leading to inefficiency in memory utilization.

Paging

- If the process requires n pages, at least n frames are required

- The first page of the process is loaded into the first frame listed on free-frame list, and the frame number is put into page table

Hardware Support on Paging

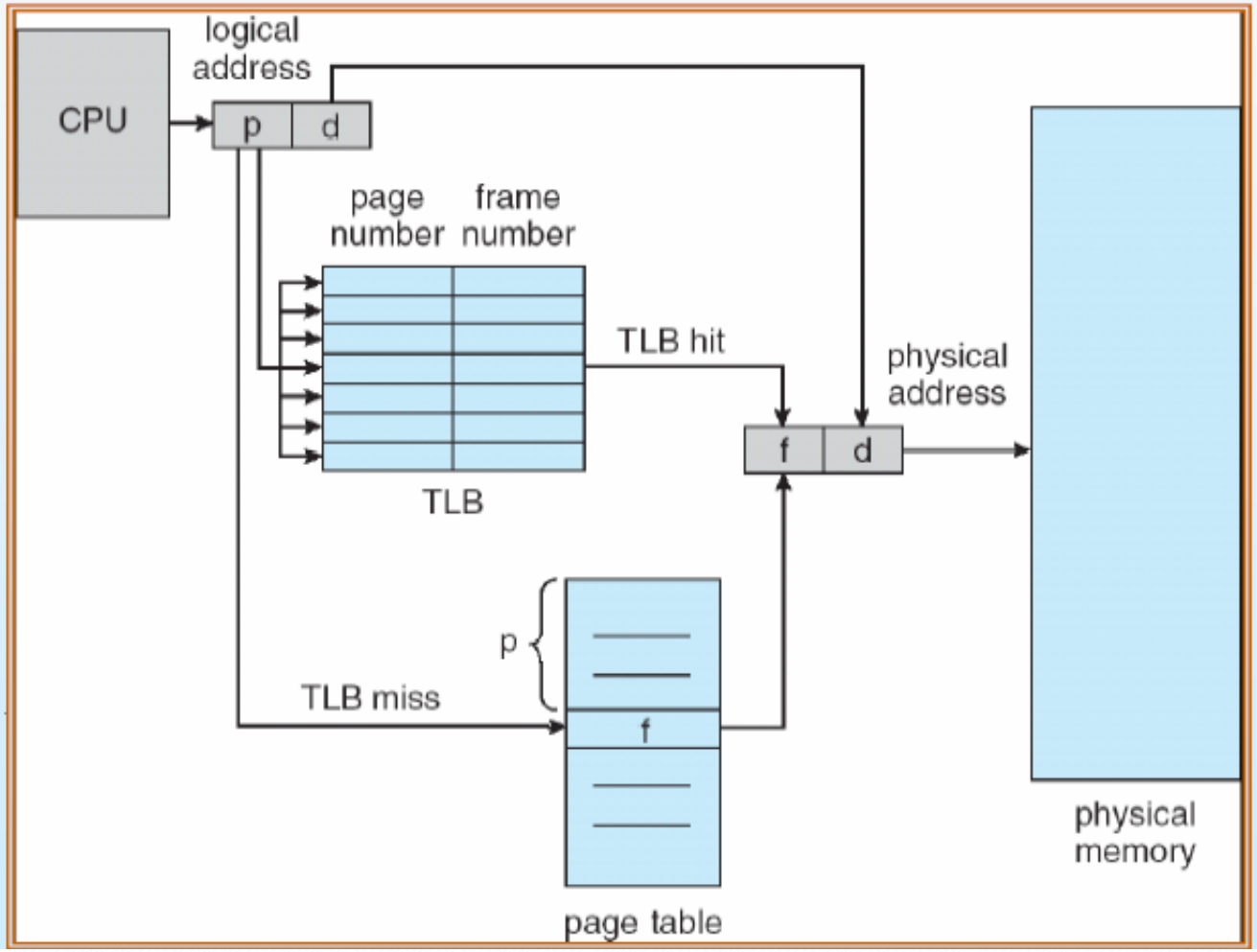
- To implement paging, the simplest method is to implement the page table as a set of registers
- However, the size of register is limited and the size of page table is usually large
- Therefore, the page table is kept in main memory and a page table base register(PTBR) is used to point the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time

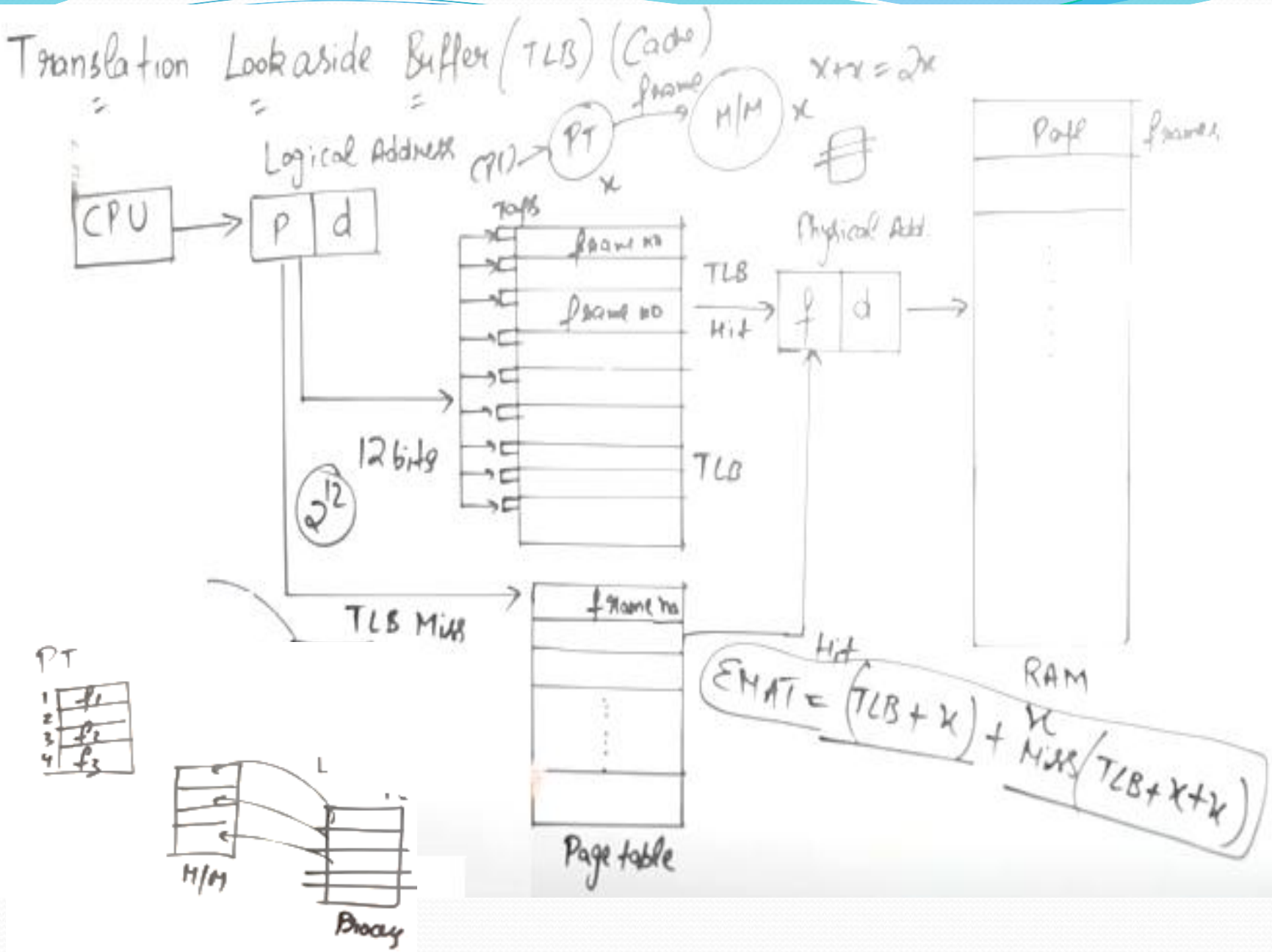
Hardware Support on Paging

- The problem with this approach is the time required to access a user memory location. If we want to access location I, we must first index into page table. It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory
- With this scheme, **TWO** memory access are needed to access a byte (one for the page-table entry, one for the byte).
- The standard solution is to use a special, small, fast lookup hardware cache, called **Translation look-aside buffer (TLB)** or **associative memory.**

- The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. 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.
- The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found (TLB hit), its frame number is immediately available and is used to access memory.

Paging hardware with TLB




Translation Lookaside Buffer (TLB) (Cache)
x+x = 2x
Logical Address
CPU
p
d
PT
Frame
M/M
x
x
Page
Frame
Tags
Frame no
Frame no
TLB
Hit
Physical Add.
f
d
12 bits
2^{12}
TLB
TLB Miss
Frame no
Page table
RAM
Hit
EMAT = (TLB + x) + Miss (TLB + x + x)
PT
1
2
3
4
L
M/M
Binary

TLB

- If the page number is not in the TLB (TLB miss) a memory reference to the page table must be made.
- In addition, we add the page number and frame number into TLB
- If the TLB already full, the OS must select one for replacement
- Some TLBs allow entries to be **wire down**, meaning that they cannot be removed from the TLB, for example kernel codes

TLB

- The percentage of times that a particular page number is found in the TLB is called **hit ratio**
- If it takes 20 nanosecond to search the TLB and 100 nanosecond to access memory.
- If we fail to find the page number in the TLB, 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).
- If our hit ratio is 80%, the effective memory access time equal:

  0.8*(100+20) + 0.2 *(100+100+20)=140
- If our hit ratio is 98%, the effective memory access time equal:

  0.98*(100+20) + 0.02 *(100+100+20)=122

Memory Protection

- Memory protection is implemented by associating protection bit with each frame

- **Valid-invalid** bit attached to each entry in the page table:
- "valid" indicates that the associated page is in the process's logical address space, and is thus a legal page.

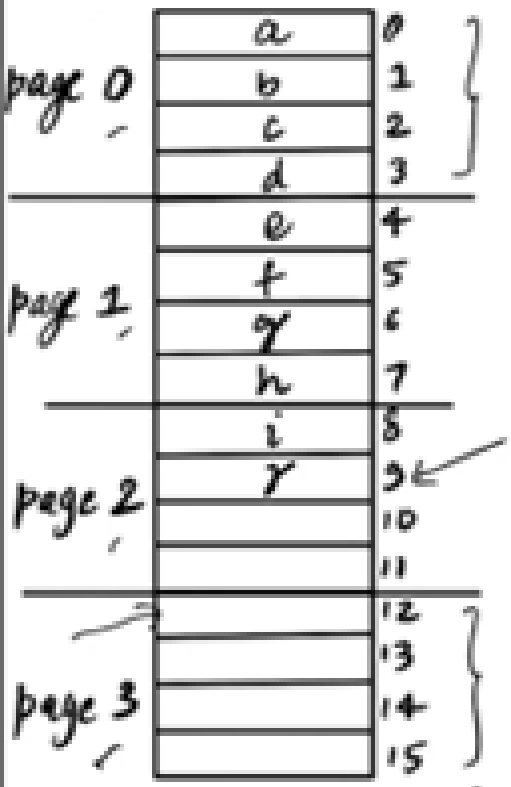
- "invalid" indicates that the page is not in the process logical address space.

Memory Protection

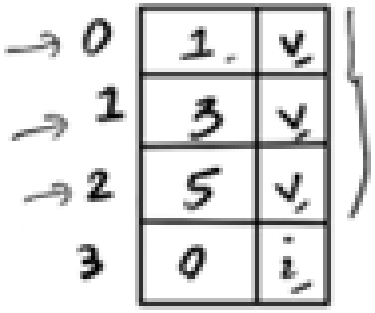
- Suppose a system with a 4bit address space (0 to 16), we have a program that should use only address 0 to 10. Given a page size of 4Bits, we may have the following figure:

Assume that the System has 4-bit logical address space.

page size = 4 bit



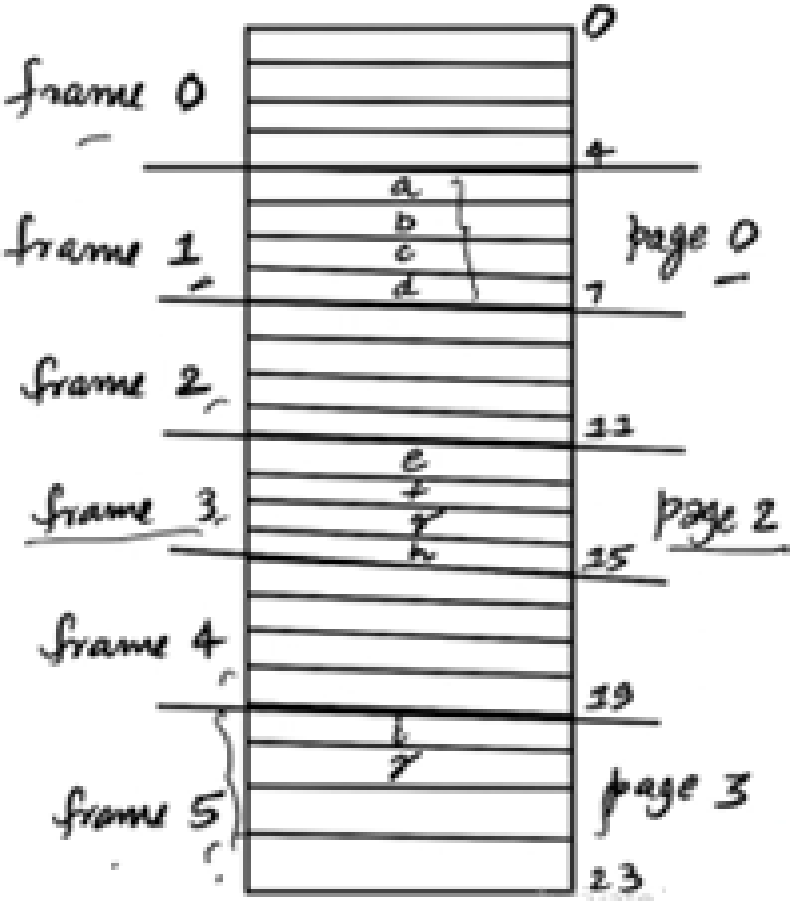
Logical address



Page Table

10 bytes

16 bytes

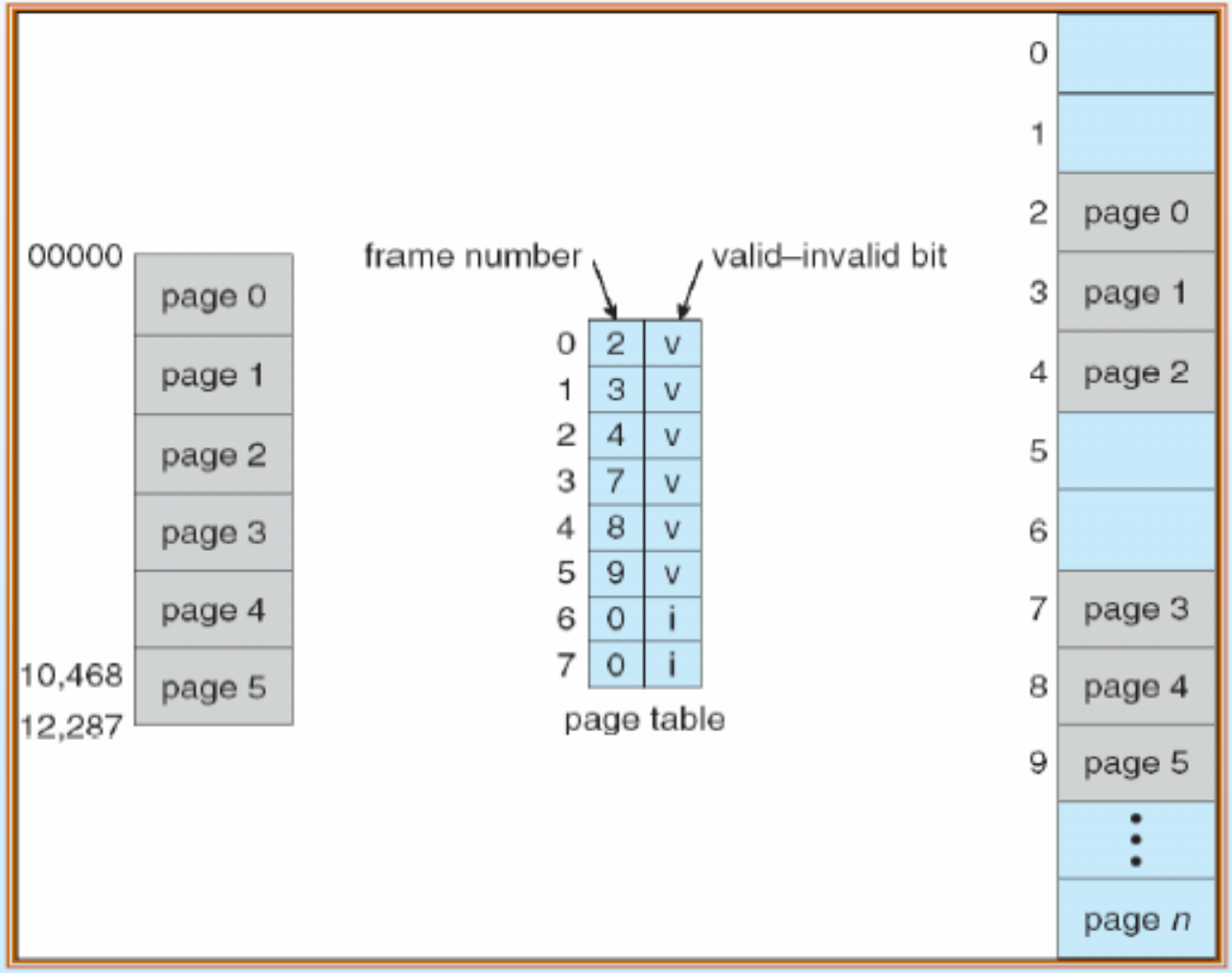


Physical address

Memory Protection

- Suppose a system with a 14bit address space (0 to 16383), we have a program that should use only address 0 to 10468. Given a page size of 2KB, we may have the following figure:

Memory Protection



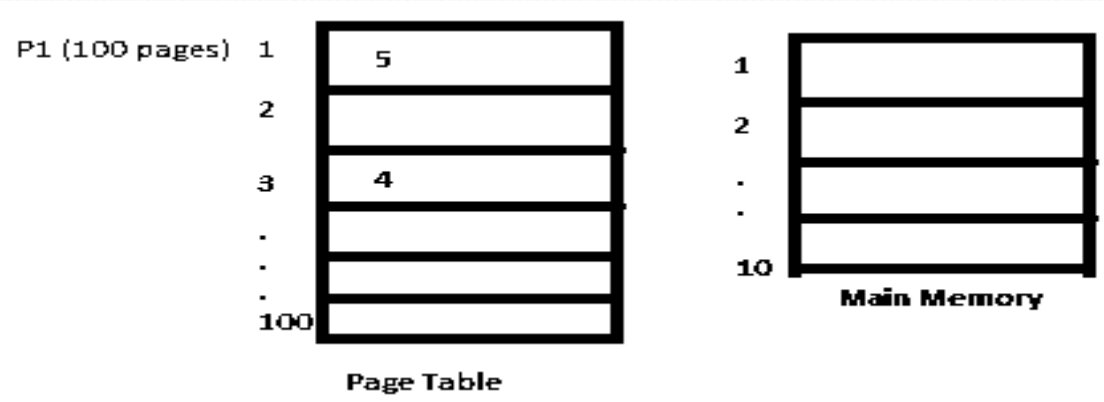
Memory Protection

- Any attempt to generate an address in page 6 or 7 will be invalid

- This scheme has created a problem. Because the program extends only to address 10468, any reference beyond that address is illegal. Howeve1~ references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid. This problem is a result of the 2-KB page size and reflects the internal fragmentation of paging.

Inverted Page Table

- Most of the Operating Systems implement a separate page table for each process, i.e. for 'n' number of processes running on a Multiprocessing/ Timesharing operating system, there are 'n' number of page tables stored in the memory. Sometimes when a process is very large in size and it occupies virtual memory then with the size of the process, it's page table size also increases substantially.
- As per the diagram below, lets assume that 1st page is available in frame 5 , page 2 is not contained in main memory, page 3 is available in frame 4 and so on. So the problem here is that process has 100 pages and main memory has only 10 frames and the page table contains the information of all the 100 pages irrespective of the fact whether that page resides in main memory or not. If the page resides in main memory, page table provides the corresponding frame no and if it doesnt reside in main memory then it doesn't provide any information (blank block). It means that space of page table is not being utilized in efficient manner as there is no need to store information inside the page table regarding the pages that are not residing in main memory.
- A considerable part of memory is occupied by page tables only. The amount of memory occupied by the page tables can turn out to be a huge overhead and is always unacceptable as main memory is always a scarce resource
- To solve this problem, we can use an inverted page table.



Inverted Page Table

- An inverted page table has one entry for each real page (or frame) of main memory So the number of page table entries in the Inverted Page Table reduces to the number of frames in physical memory and a single page table is used to represent the paging information of all the processes.

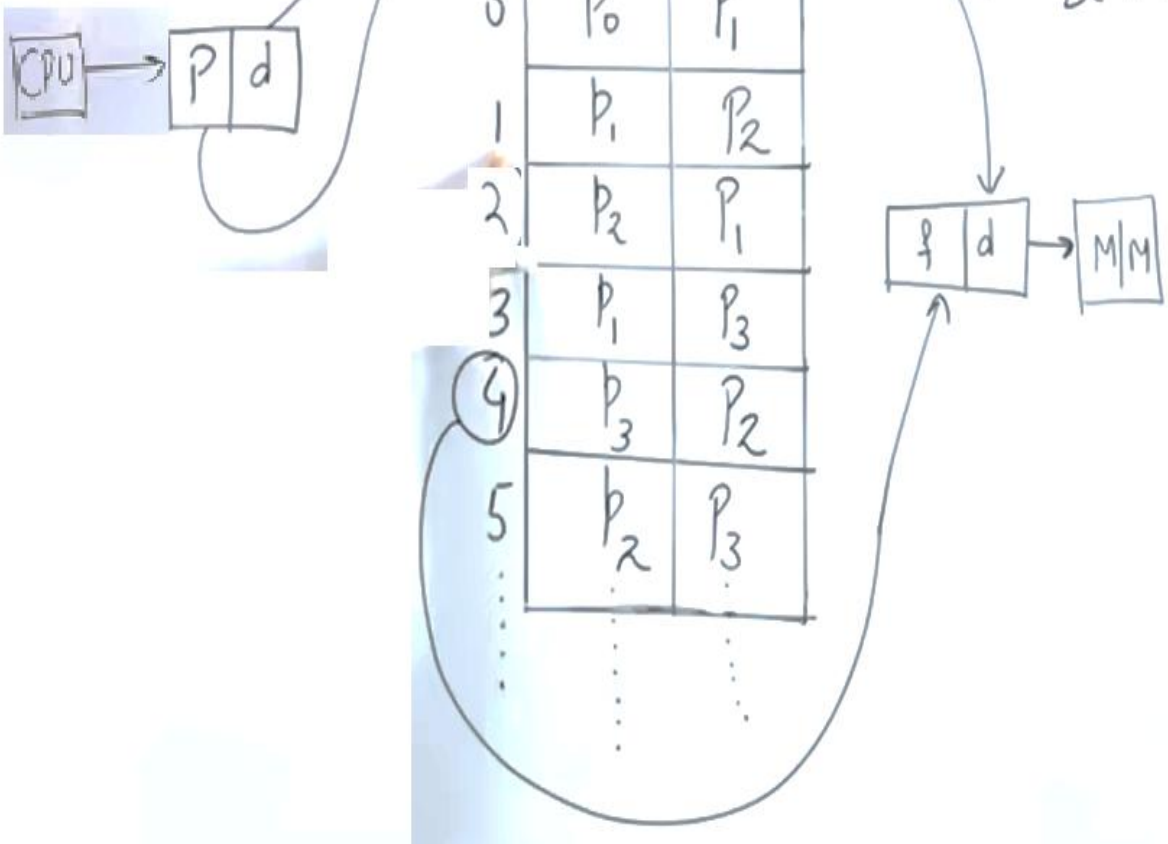
- Through the inverted page table, the overhead of storing an individual page table for every process gets eliminated and only a fixed portion of memory is required to store the paging information of all the processes together. This technique is called as inverted paging as the indexing is done with respect to the frame number instead of the logical page number.

- Each virtual address in the system consists of a triple: <process-id, page-number, offset>.

- Process id – An inverted page table contains the address space information of all the processes in execution. Since two different processes can have similar set of virtual addresses, it becomes necessary in Inverted Page Table to store a process Id of each process to identify it's address space uniquely. This is done by using the combination of PId and Page Number. So this Process Id acts as an address space identifier and ensures that a virtual page for a particular process is mapped correctly to the corresponding physical frame.

Inverted Paging

Global PT



1) Each process has its own PT

2) PT will be in M/M

0	f_0
1	X
2	f_2
3	X

Page table of P_1

0	X
1	f_3
2	f_5
3	X

Page table of P_3

0	X
1	f_1
2	X
3	f_4

Page table of P_2

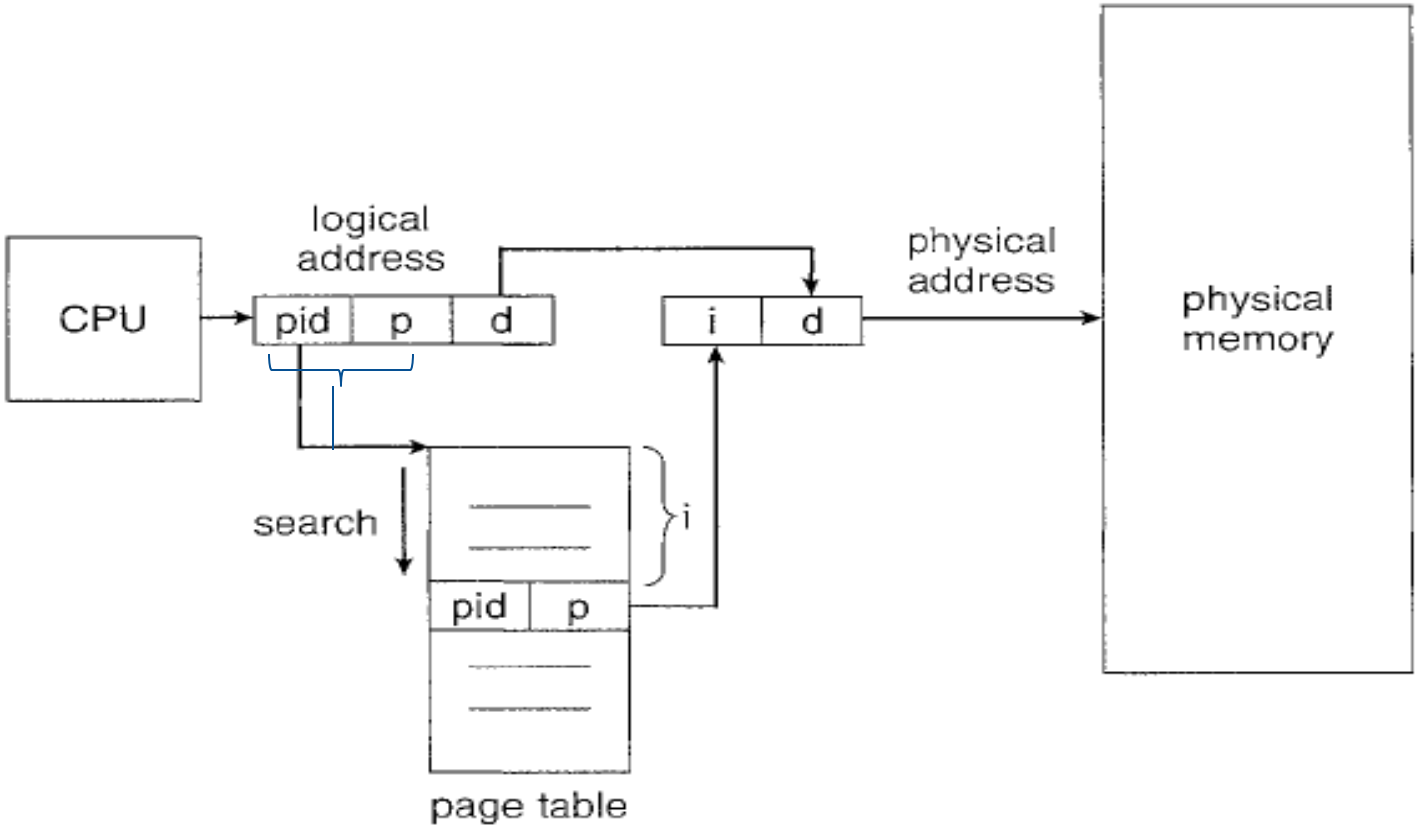


Figure 8.17 Inverted page table.

Number of Entries in Inverted page table = Number of frames in Physical address Space(PAS)

Segmentation

- Paging is more close to the Operating system rather than the User. It divides all the processes into the form of pages regardless of the fact that a process can have some relative parts of functions which need to be loaded in the same page.
- Operating system doesn't care about the User's view of the process. It may divide the same function into different pages and those pages may or may not be loaded at the same time into the memory. It decreases the efficiency of the system.
- It is better to have segmentation which supports user's view of memory and divides the process into the segments. Each segment contains the same type of functions such as the main function can be included in one segment and the library functions can be included in the other segment.

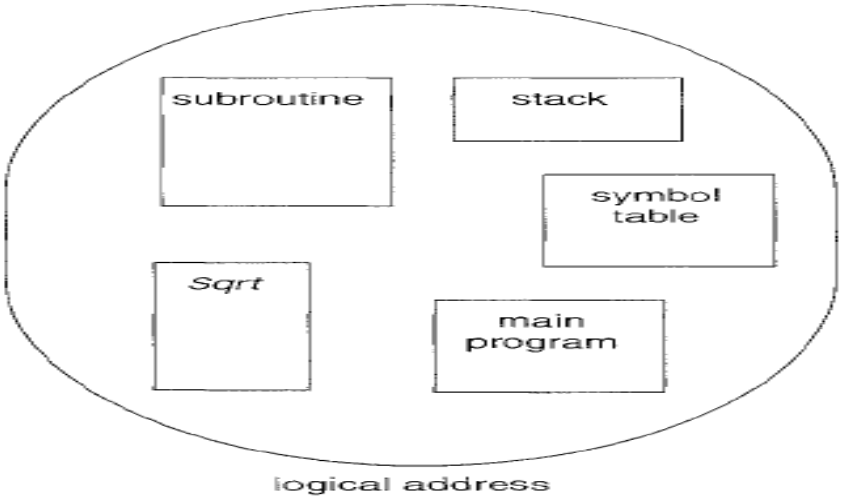


Figure 8.18 User's view of a program.

Segmentation

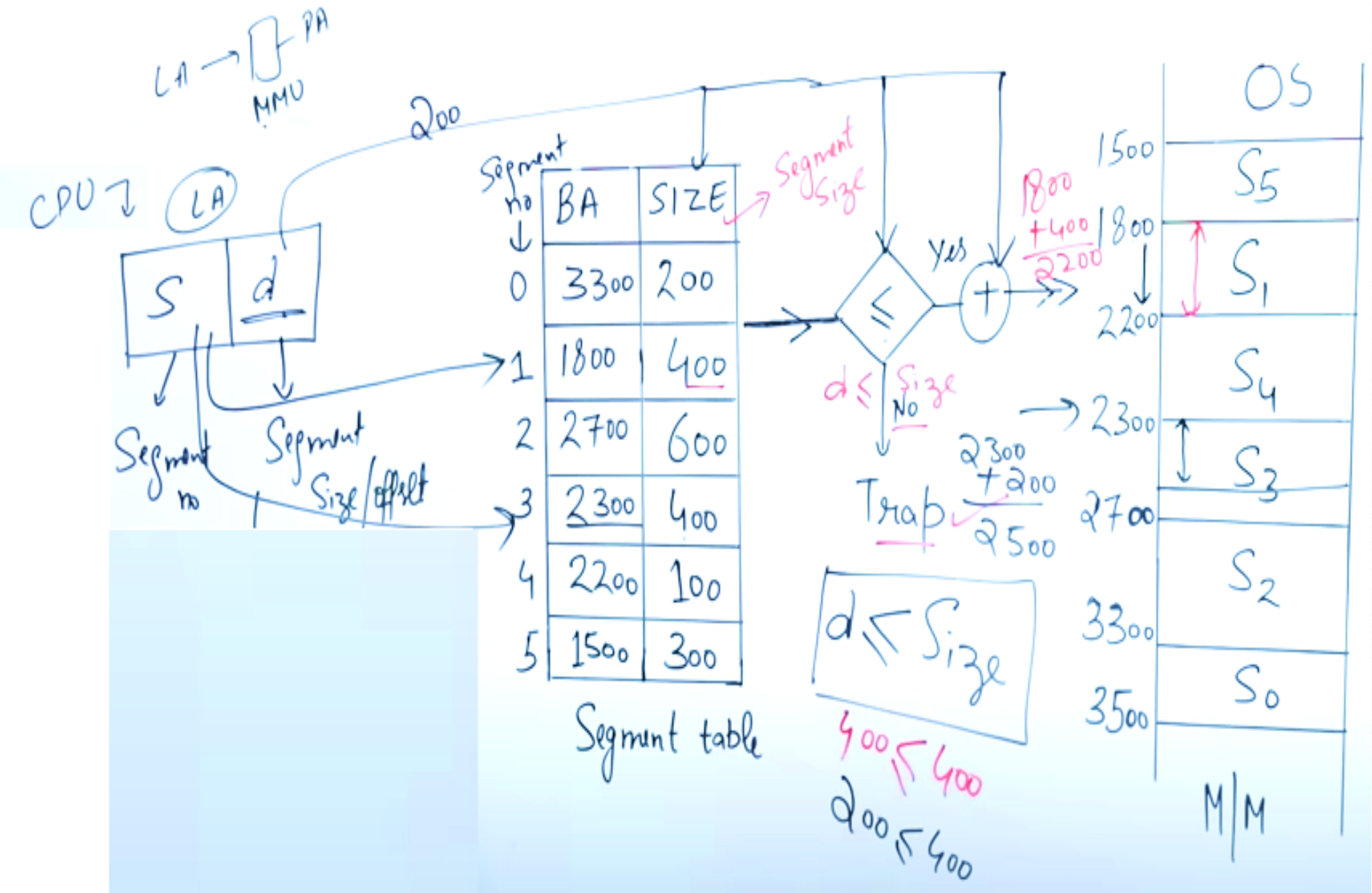
- A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset

- For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a two tuple: <segment-number, offset>. The segment number is used as an index to the segment table. The offset d of the logical address must be between o and the segment limit.

- When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

- Segment Table – It maps two-dimensional Logical address into one-dimensional Physical address. It's each table entry has:

  - Base Address: It contains the starting physical address where the segments reside in memory.
  - Limit: It specifies the length of the segment.


LA
PA
MMU
200
CPU
LA
S
d
Segment no
Segment Size/offset
Segment no
BA
SIZE
Segment Size
0 3300 200
1 1800 400
2 2700 600
3 2300 400
4 2200 100
5 1500 300
Segment table
yes
d ≤ Size
No
Trap
1800 +400 2200
2300 +200 2500
d ≤ Size
400 ≤ 400
200 ≤ 400
OS
1500
S5
1800
S1
2200
S4
2300
S3
2700
S2
3300
S0
3500
M/M

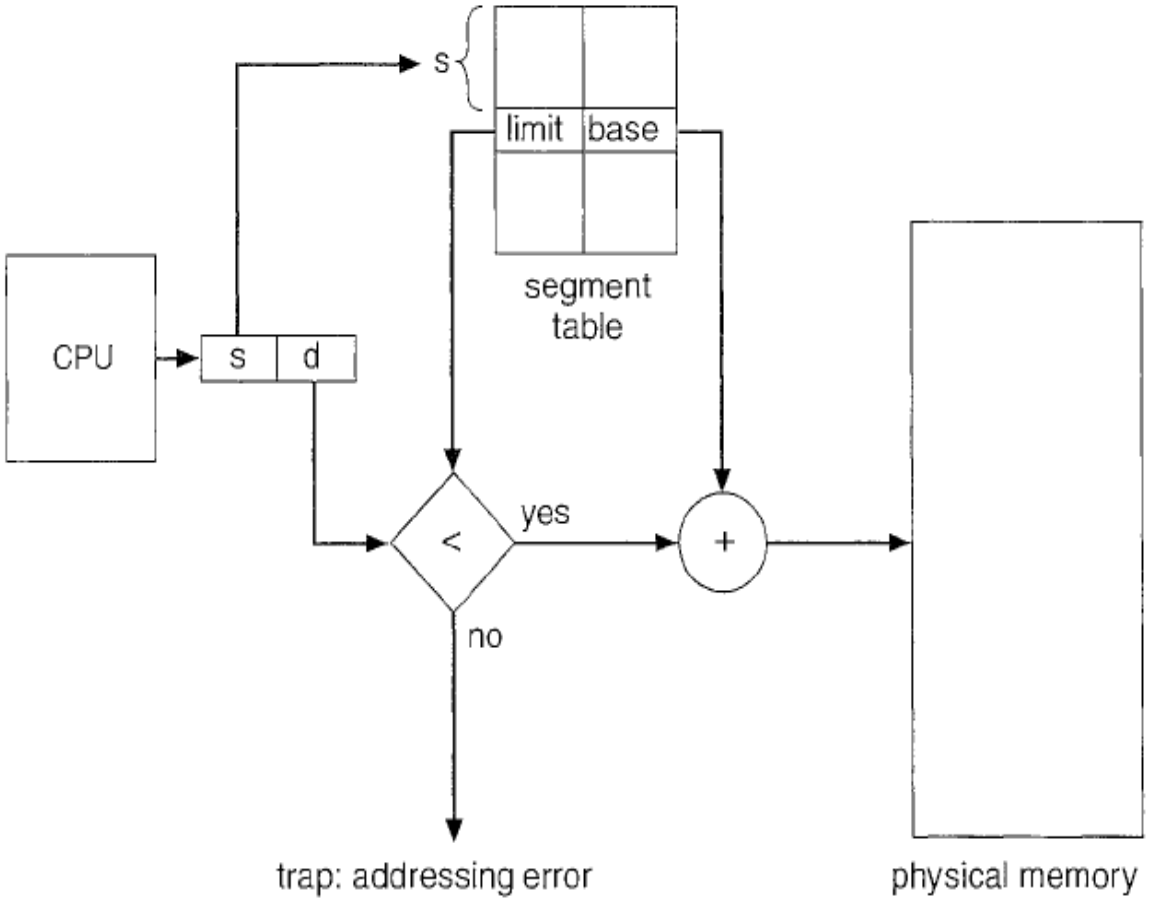


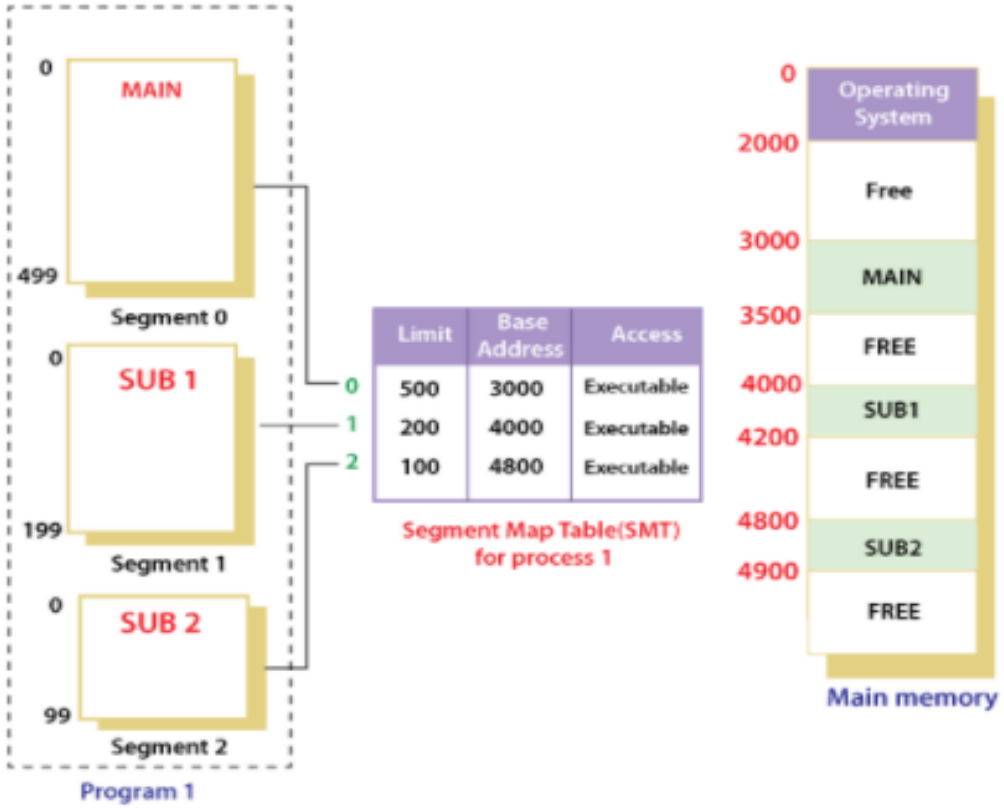
Figure 8.19 Segmentation hardware.

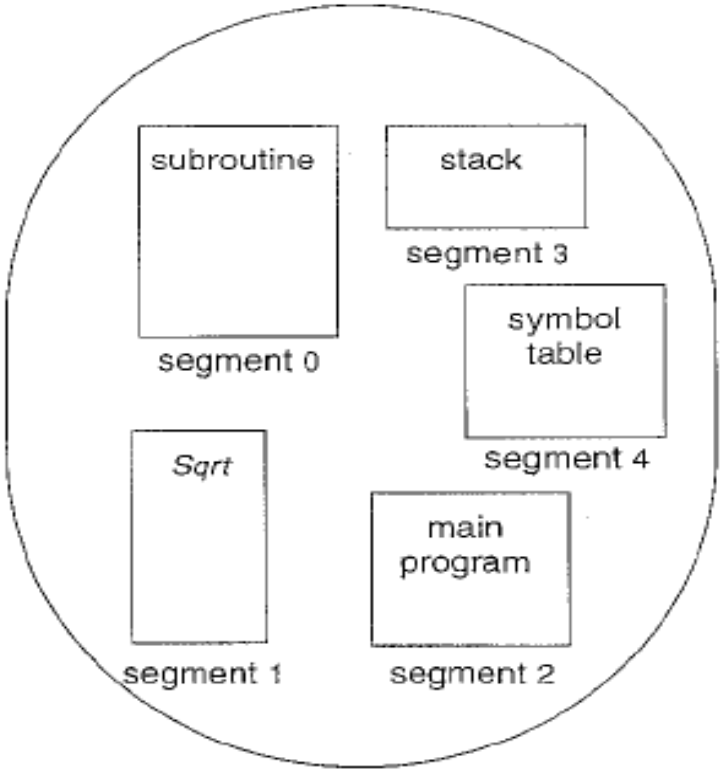
For Example:

Suppose a 16 bit address is used with 4 bits for the segment number and 12 bits for the segment offset so the maximum segment size is 4096 and the maximum number of segments that can be refereed is 16.

When a program is loaded into memory, the segmentation system tries to locate space that is large enough to hold the first segment of the process, space information is obtained from the free list maintained by memory manager. Then it tries to locate space for other segments. Once adequate space is located for all the segments, it loads them into their respective areas.

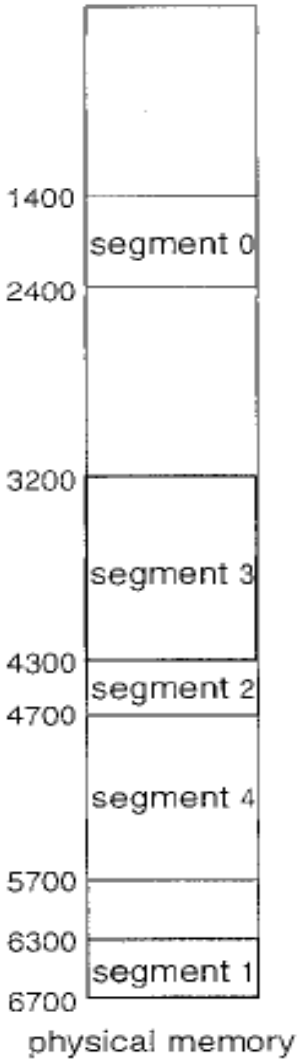
The operating system also generates a segment map table for each program.





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

segment table



Thank You!!!