INTRODUCTION TO MEMORY MANAGEMENT

- ✓Program must be brought (from disk) into memory and placed within a process for it to run
- ✓OS loads the program into the memory and then only the CPU can access the instructions and the data from memory while executing the process
- ✓In a uniprogramming system, main memory is divided into two parts: one part for the operating system and one part for the program currently being executed.
- ✓In a multiprogramming system, the "user" part of the memory must be further subdivided to accommodate multiple processes
- ✓The task of subdivision is carried out dynamically by the operating system and it is known as memory management

INTRODUCTION TO MEMORY MANAGEMENT

✓Memory manager in an OS effectively manages the memory : keep track of which parts of memory are currently in use, allocate memory to processes when they need it and deallocate it when they are done

- The part of OS that manages the memory hierarchy is called **memory manager.**

– Keeps track of used/unused part of memory.

– Allocate/de-allocate memory to processes.

– Manages swapping between main memory and disk.

– Protection and sharing of memory

BASIC HARDWARE

- ✓Main memory and registers are only storage CPU can access directly
- ✓Registers that are built into the CPU are generally accessible within one cycle of the CPU clock.
- ✓The same cannot be said of main memory, which is accessed via a transaction on the memory bus.
- ✓Memory access may take many cycles of the CPU clock to complete, in which case the processor normally needs to **stall**, since it does not have the data required to complete the instruction that it is executing.
- ✓The remedy is to add fast memory between the CPU and main memory : cache

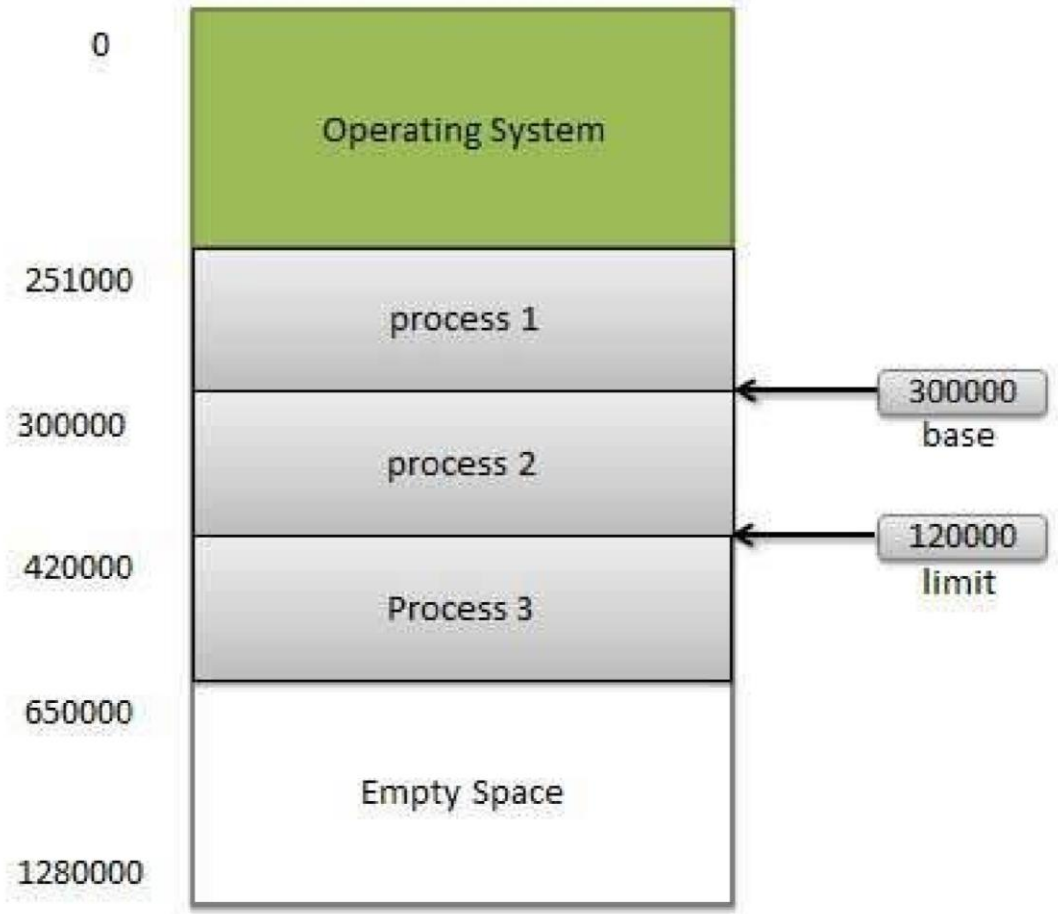
BASIC HARDWARE

- ✓We must not only be concerned with relative speed of accessing the physical memory, but also ensure the protection of one user process from another
- ✓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.
- ✓This protection is ensured by using two registers, a base register and a limit register.
- ✓The base register holds the smallest legal physical memory address ✓ Limit register specifies the size of the range.
- ✓For example, if the base register holds

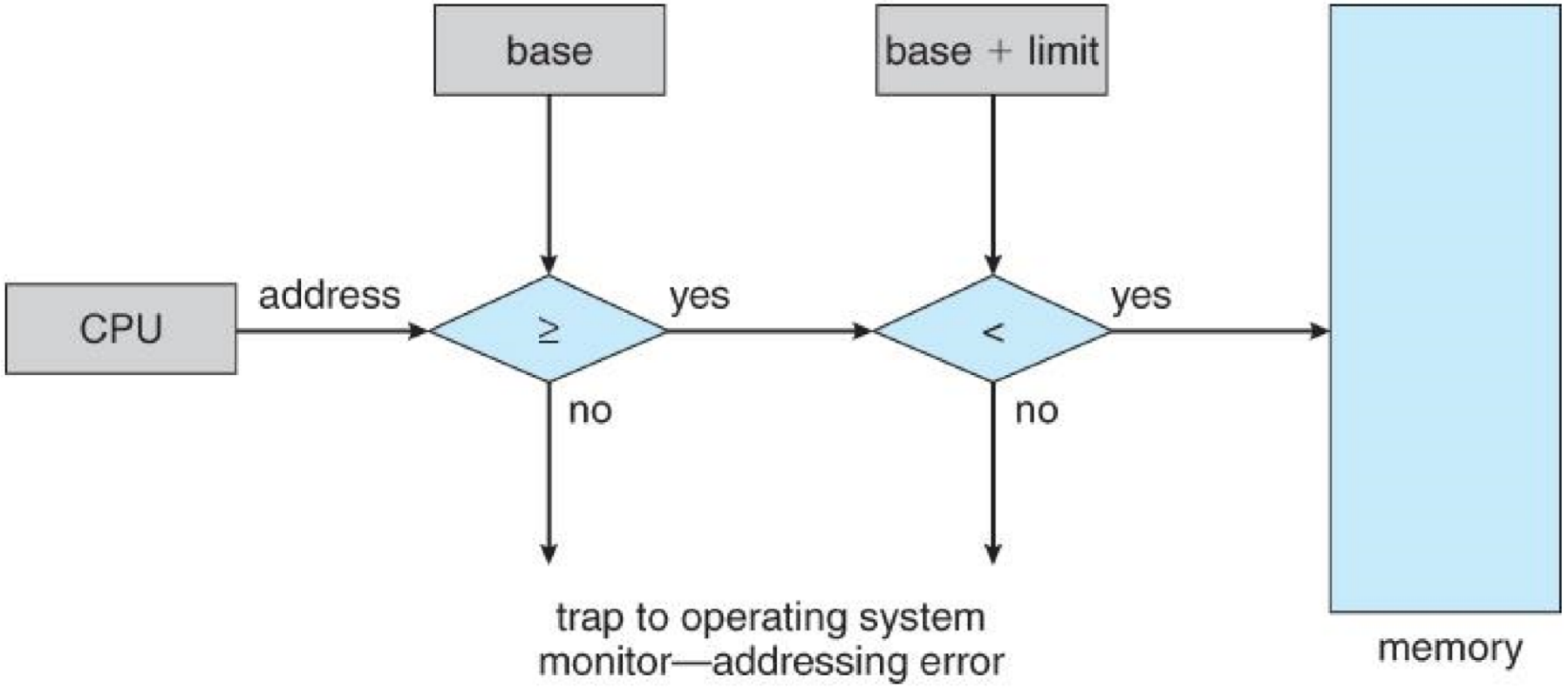
BASE AND LIMIT REGISTERS

300000 and the limit register is 120000, then the program can legally access all addresses from 300000 through 419999.

Q. If base register holds 300040 and Limit register holds 120900 what are the Addresses that the program can legally Access?



PROTECTION : BASE AND LIMIT REGISTER



BASE AND LIMIT REGISTER

- ✓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
- ✓The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction.
- ✓Since privileged instructions can be executed only in kernel mode, and since only the operating system execute in kernel mode, 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 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 etc.

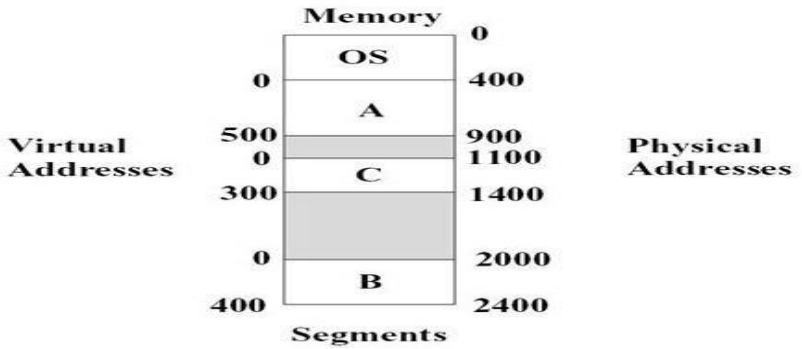
MEMORY MANAGEMENT : TERMINOLOGY

Segment :

- chunk of memory assigned to a process Physical address :

- a real address in memory Virtual address :

- Address relative to start of process's address space



ADDRESS BINDING

✓Address binding refers to mapping an address space into another(e.g. mapping from virtual to physical addresses)

✓How are the instruction and the data addresses generated (i.e. the addresses that the processes is operating on)?

✓Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

1 . Compile time

- The compiler generates the exact physical location in memory starting from some fixed starting position k. The OS does nothing.

-If, at some later time, the starting location changes, then it will be necessary to recompile this code.

ADDRESS BINDING

2.Load time

- If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code.
- At load time, OS decides the starting position
- Once the process loads, it does not move in memory

3.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.
- This creates the virtual/logical addresses unlike load time and compile time where we are actually using the physical memory addresses

LOGICAL VS. PHYSICAL ADDRESS SPACE

- ✓The compile-time and load-time address-binding methods generate identical logical and physical addresses.
- ✓However, the execution-time address binding scheme results in differing logical and physical addresses.
- ✓In this case, we usually refer to the logical address as a **virtual address.**
- ✓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.
- ✓The run-time mapping from virtual to physical addresses is done by hardware device called the **memory-management unit (MMU).**

MEMORY MANAGEMENT REQUIREMENTS

Requirements that the memory management is intended to satisfy

1. Relocation
2. Protection
3. Sharing
4. Logical organization
5. Physical organization

MEMORY MANAGEMENT REQUIREMENTS

Relocation

- ✓Programmer does not know where the program will be placed in memory when it is executed
- ✓While the program is executing, it may be swapped to disk and returned to main memory at a different location (relocated)
- ✓Memory references must be translated in the code to actual physical memory address

Protection

- ✓Processes should not be able to reference memory locations in another process without permission
- ✓Processes also should not be able to corrupt the OS

MEMORY MANAGEMENT REQUIREMENTS

Sharing

- ✓Allow several processes to access the same portion of memory in a controlled way
- ✓Better to allow each process access to the same copy of the program rather than have their own separate copy

Physical Organization

- ✓Memory available for a program plus its data may be insufficient
- ✓*Overlaying* allows various modules to be assigned the same region of memory with a main program responsible for switching the modules in and out as needed
- ✓Programmer does not know how much space will be available

✓The task of moving the information between two levels of memory is the essence of memory management

MEMORY MANAGEMENT REQUIREMENTS

Logical organization

✓Programs are written in modules

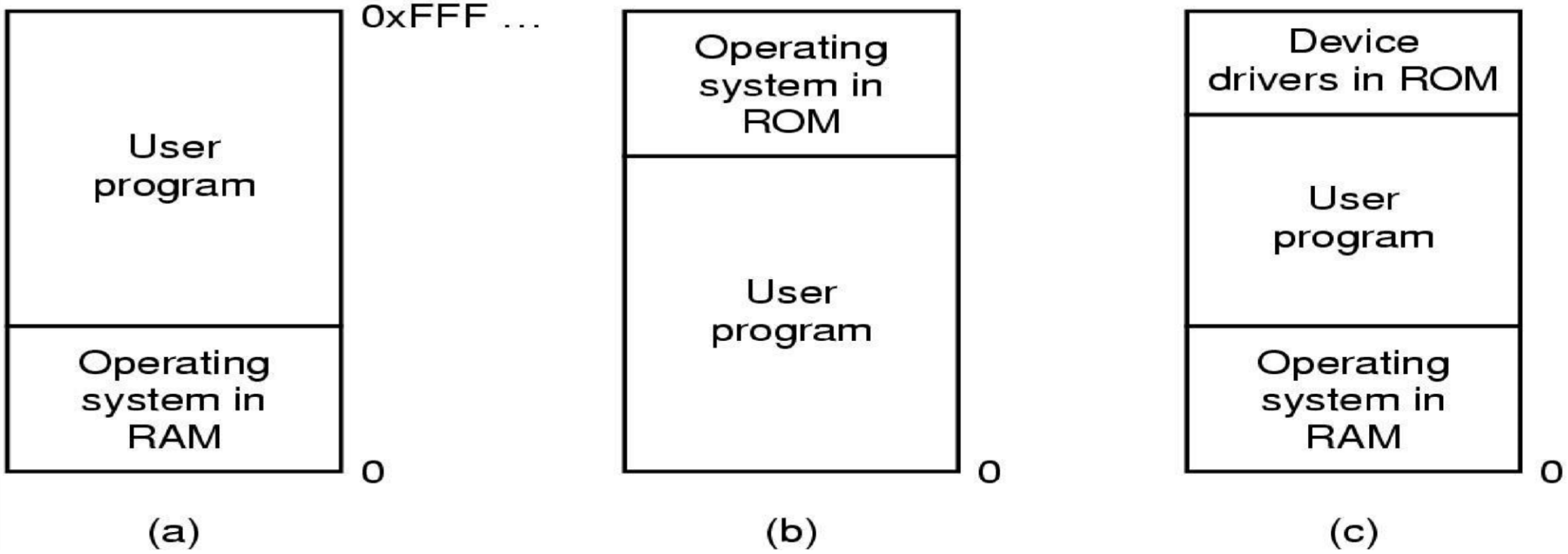
✓Modules can be written and compiled independently

✓Different degrees of protection given to modules (read-only, execute-only) ✓Share modules among processes

UNIPROGRAMMING/MONOPROGRAMMING

- ✓It is the simplest memory management approach in which only one program is running at a time
- ✓OS gets a fixed part if the memory (either highest memory area like in DOS or lowest memory area)
- ✓A single process may use all the remaining memory
- ✓Processes execute in a contiguous section of memory
- ✓Compiler can generate physical addresses which will be used directly (no memory abstraction)
- ✓Maximum address size = memory size – OS size

UNIPROGRAMMING MODEL



HOW FENCE REGISTER PROVIDES PROTECTION?

- ✓OS is protected from processes by using a **Fence register**
- ✓The OS codes usually reside in low memory area
- ✓Fence Register is set to the highest address occupied by the OS code.
- ✓A memory address generated by the user program to access certain memory location is first compared with the fence register's content.
- ✓If the address generated is below the fence, it will be trapped and denied permission. Since modification of the fence register is considered as a privileged operation, therefore, only OS is allowed to make any changes to it.

ISSUES WITH UNIPROGRAMMING

- ✓Since only one program is residing at a time in memory, it may not occupy whole memory, therefore, memory is under-utilized.
- ✓CPU will be sitting idle during the period when a running program requires some I/O. That is, there is no overlapping of I/O and computation

MULTIPROGRAMMING

✓Multiprogramming refers to having multiple processes in memory at once

✓The following considerations must be mode in multiprogramming

1. Transparency

- we want multiple processes to co-exist in memory
- no process should be aware that the memory is shared
- processes should not care what physical portion of memory they are assigned to

2. Safety

- Processes must not be able to corrupt each other
- processes must not be able to corrupt OS

3. Efficiency

-Performance of CPU and memory should not be degraded due to sharing

RELOCATION

- ✓We cannot know ahead of time where a program will be placed, and we must allow that the may be moved about in main memory due to swapping
- ✓In simple terms, relocation is a mechanism to convert logical (or virtual) address into an physical address.
- ✓Two approaches:

1. Static Relocation
2. Dynamic Relocation

STATIC RELOCATION

- ✓At load time, the OS adjusts the addresses in a process to reflect its position in memory
- ✓Once a process is assigned a place in memory, and starts executing it, the OS cannot move it

DYNAMIC RELOCATION

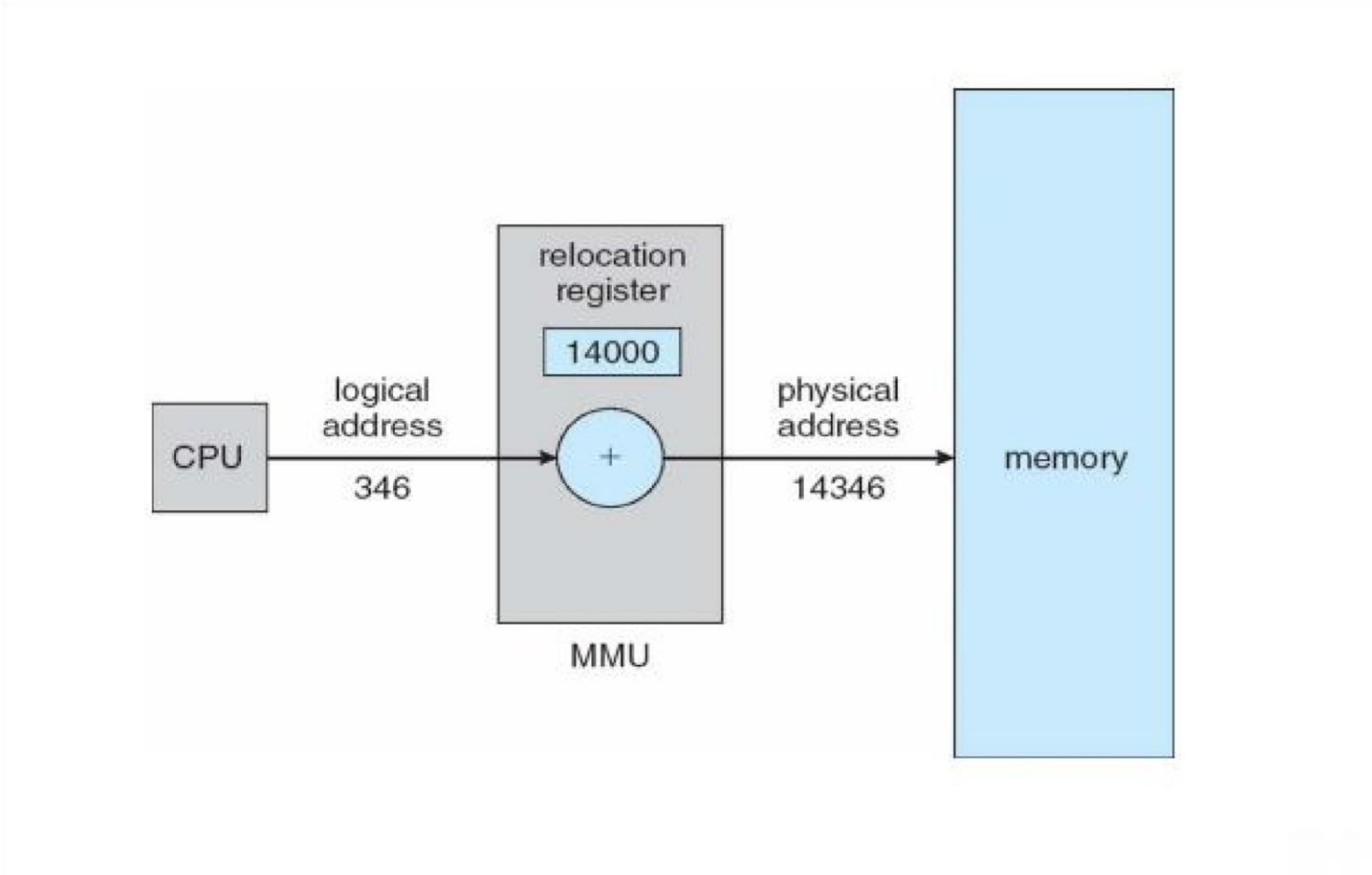
- ✓In dynamic relocation, the final location of a referenced memory address is not determined until the reference is made. (i.e. execution time binding)
- ✓The base register is now called the ***relocation register***

DYNAMIC RELOCATION

- ✓The value in the relocation register is *added* to every address generated by a user process at the time it is sent to memory
- ✓For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.

Task: Advantage and disadvantage of dynamic relocation

PROTECTION IN DYNAMIC RELOCATION



MEMORY ALLOCATION TECHNIQUES

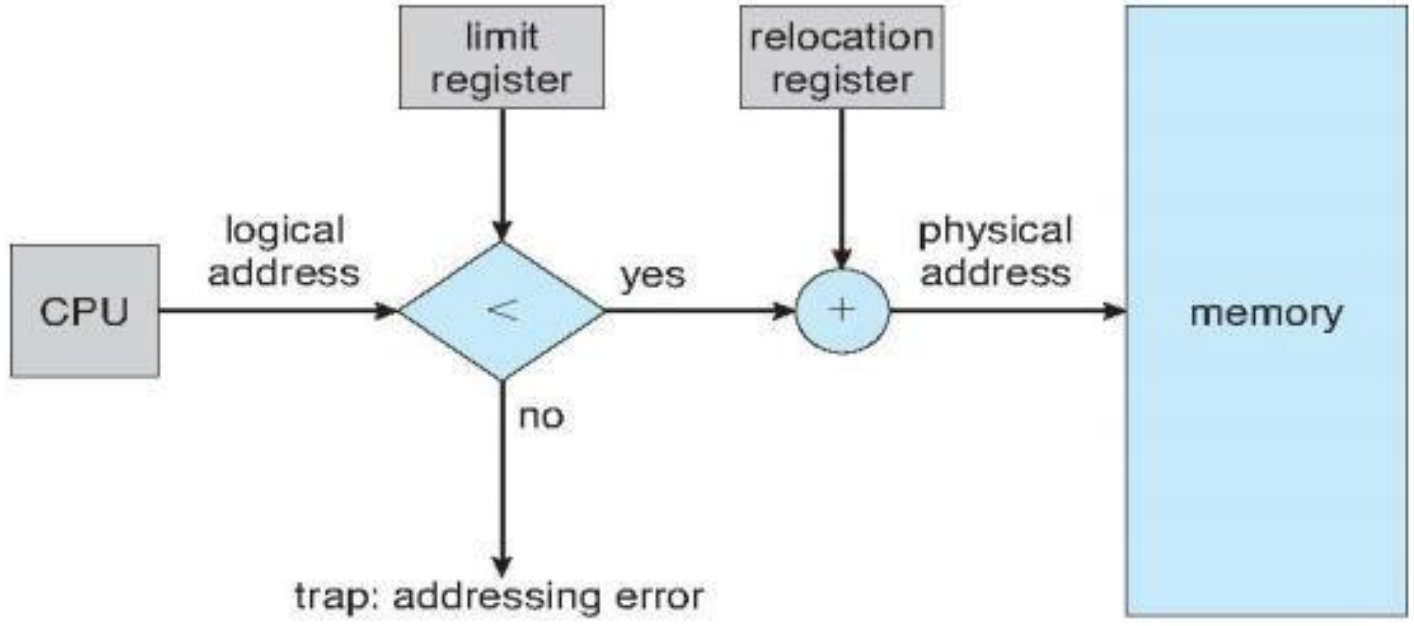
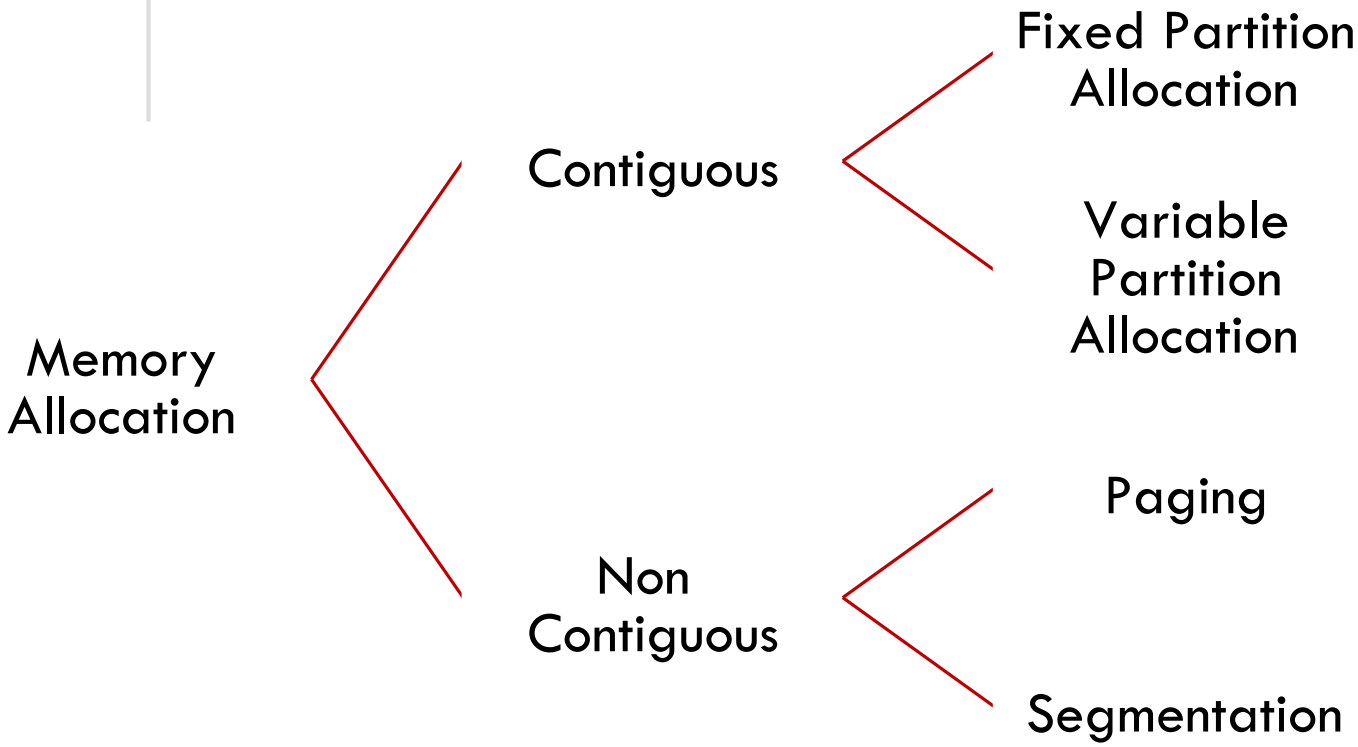


Fig : Hardware protection mechanism in dynamic relocation:


Memory Allocation
Contiguous
Fixed Partition Allocation
Variable Partition Allocation
Non Contiguous
Paging
Segmentation

CONTIGUOUS MEMORY ALLOCATION

✓The memory is usually divided into two partitions: one for the resident operating system and one for the user processes.

✓We can place the operating system in either low memory or high memory (usually in low memory)

✓In this contiguous memory allocation, each process is contained in a single contiguous section of memory (i.e. memory resident program occupies a single contiguous block of physical memory.) ✓The memory is partitioned into blocks of different sizes to accommodate the programs.

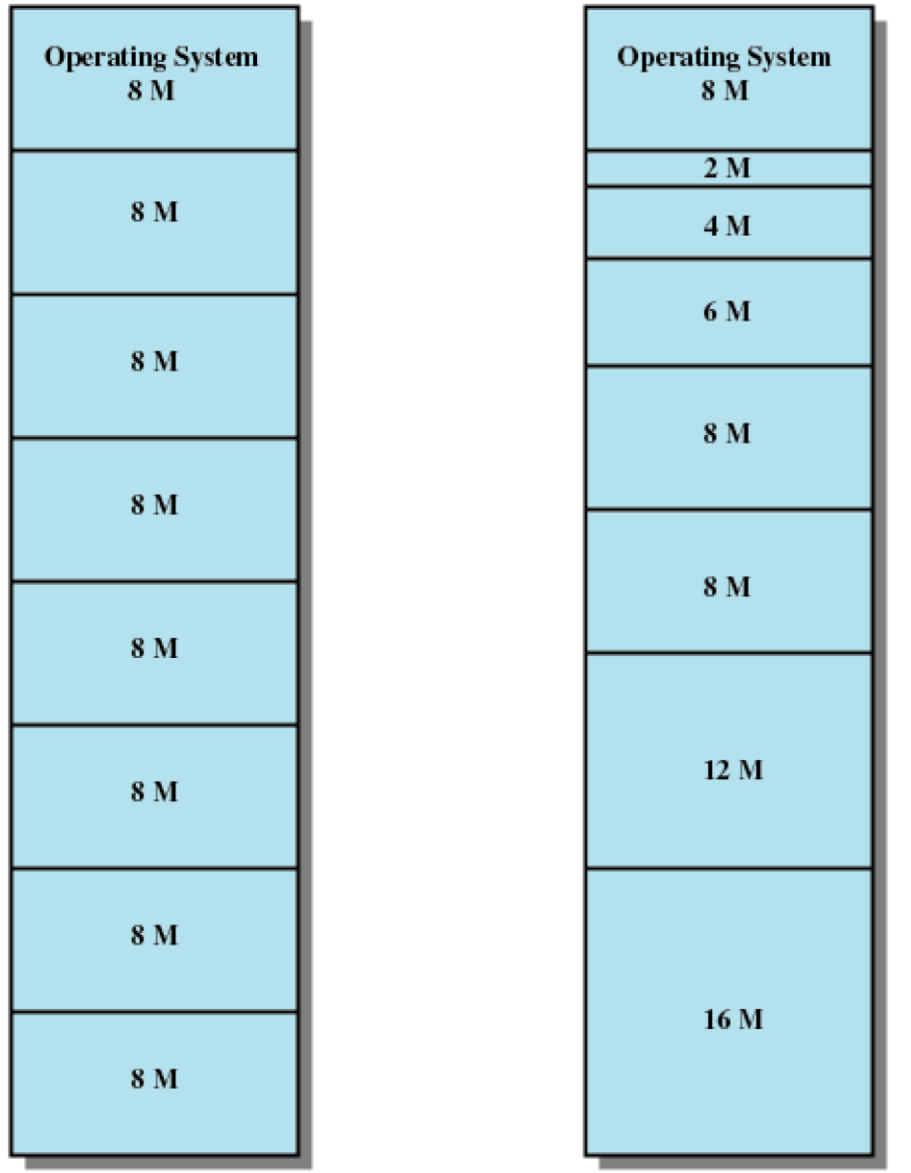
✓Difficult to grow or shrink process ✓This partitioning is of two type:

1. Fixed/Static Partition Allocation
2. Variable/Dynamic Partition Allocation

FIXED PARTITION ALLOCATION

- ✓The simplest way to manage the memory available for the processes is to partition it into regions with fixed boundaries.
- ✓The partition may be equal sized or unequal sized
- ✓A process whose size is less than or equal to the partition size can be loaded into that partition
  - ✓Difficulties:

- A program may be too big to fit into a partition
- Main memory utilization is extremely inefficient(*Internal Fragmentation*)



(a) Equal-size partitions

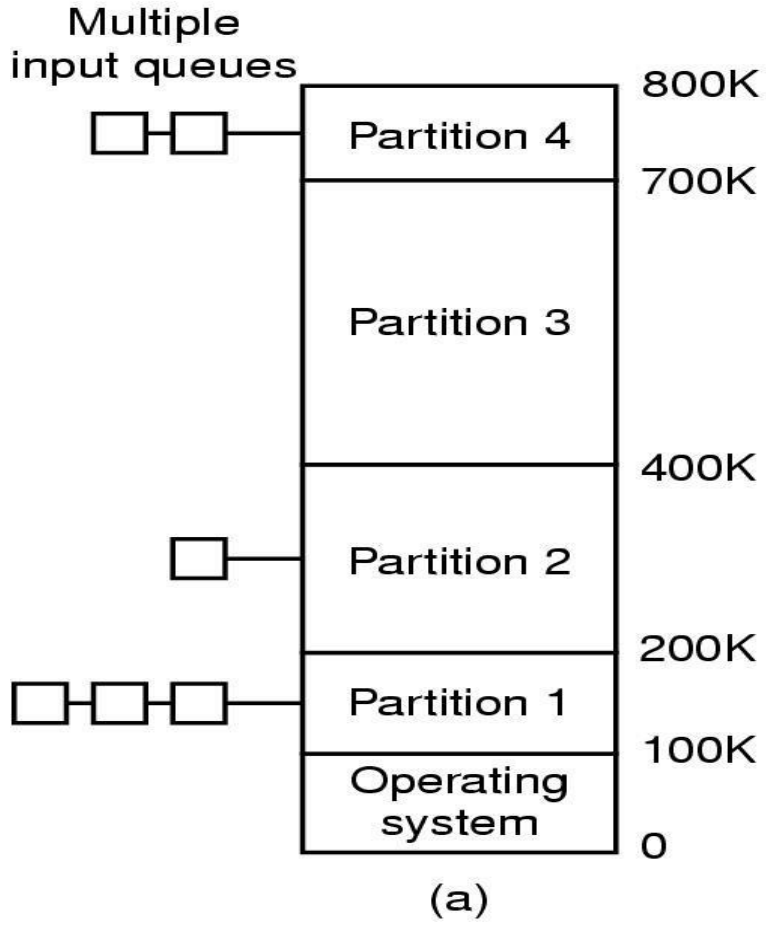
(b) Unequal-size partitions

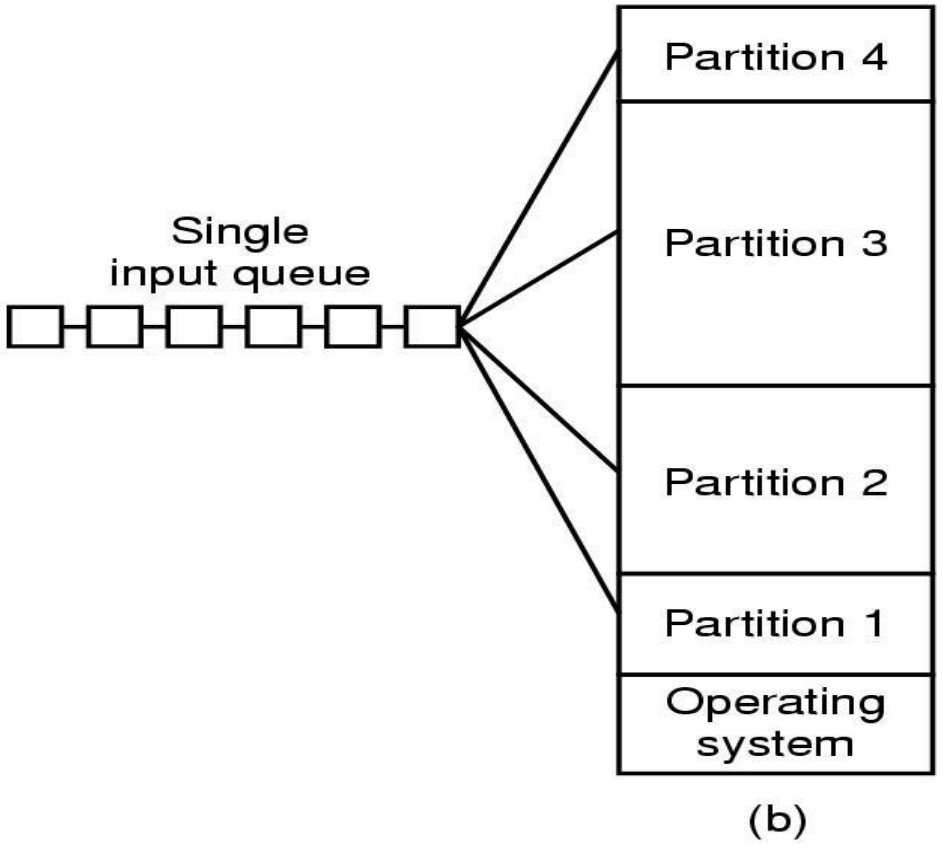
Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

FIXED PARTITIONING

- ✓ We divide the memory into several fixed size partitions where each partition will accommodate only one program for execution.
- ✓ The number of programs (i.e. degree of multiprogramming) residing in the memory will be bound by the number of partitions.
- ✓ When a program terminates, that partition is freed for another program waiting in a queue.

FIXED PARTITION ALLOCATION PLACEMENT ALGORITHM





FIXED PARTITION ALLOCATION PLACEMENT ALGORITHM

Equal-size partitions

- Because all partitions are of equal size, it does not matter which partition is used

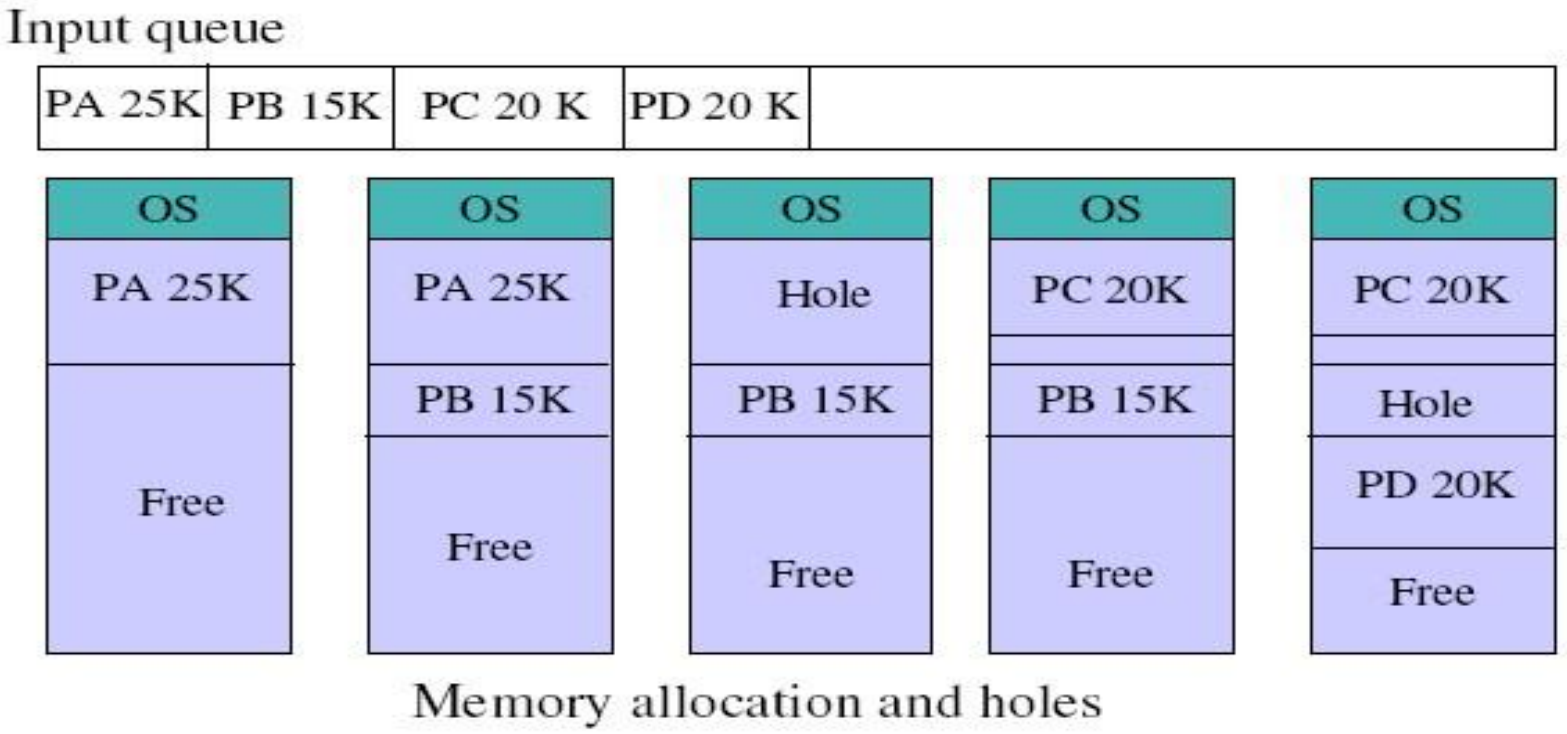
Unequal-size partitions

- Can assign each process to the smallest partition within which it will fit
- Queue for each partition
- Processes are assigned in such a way as to minimize wasted memory within a partition

DYNAMIC PARTITIONING

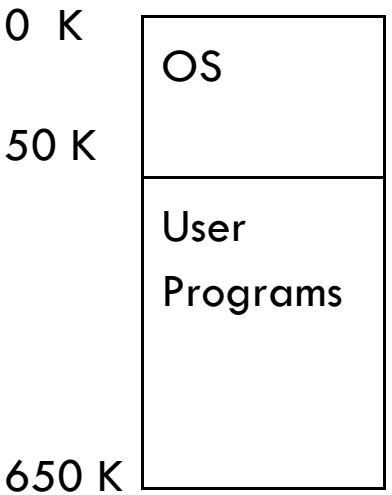
- ✓In this method, partitions are created dynamically to meet the requirements of each requesting process
- ✓When a process terminates or is swapped-out, the memory manager can return the vacated space to the pool of free memory area and can be provided to another process later
- ✓Holes (pieces of free memory) might be created during dynamic partitioning.
- ✓Given a new process, the OS must decide which hole to use for new process

DYNAMIC PARTITIONING

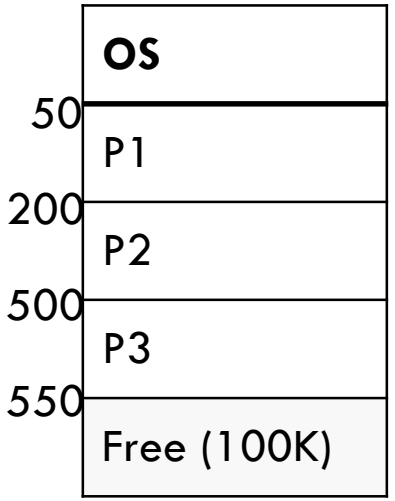


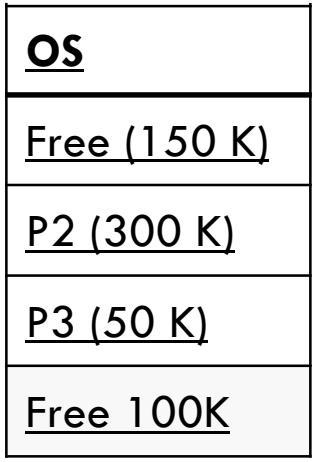
Memory allocation and holes

EXTERNAL FRAGMENTATION

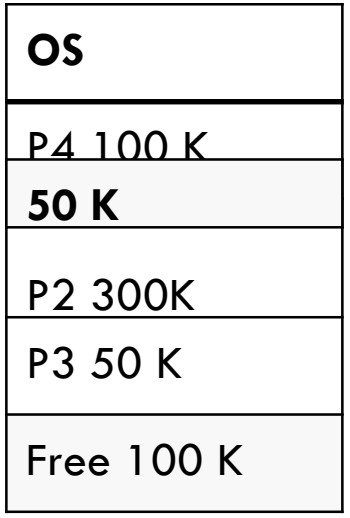


P5	P4	P3	P2	P1
150	100	50	300	150





P1 terminates



EXTERNAL FRAGMENTATION

- ✓Frequent loading and unloading of programs causes free space to be broken into little pieces
- ✓External fragmentation exists when there is enough memory to fit a process in memory, but the space is not contiguous

Task: Fixed Vs. Variable partitioning

MEMORY ALLOCATION POLICIES

First-Fit

✓The memory manager allocates the first hole that is big enough. It stops the searching as soon as it finds a free hole that is large enough.

Advantages: It is a fast algorithm because it searches as little as possible.

Disadvantages: Not good in terms of storage utilization

Next-Fit

✓Variation of first fit

✓It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole.

MEMORY ALLOCATION POLICIES

Best-Fit

✓Allocate the smallest hole that is big enough for a process

✓The OS must search the entire list of holes or store the list sorted by size of the hole ✓Best fit try to find a hole that is close to the actual size needed.

Advantages: more storage utilization than first fit.

Disadvantages: slower than first fit because it requires searching whole list at time.

MEMORY ALLOCATION POLICIES

Worst-Fit

✓Allocate the largest hole.

✓ It search the entire list, and takes the largest hole, rather than creating a tiny hole

✓ it produces the largest leftover hole, which may be more useful.

Advantages: some time it has more storage utilization than first fit and best fit.

Disadvantages: not good for both performance and utilization.

Note: Simulations show that first-fit (including next-fit) and best-fit usually yield better storage utilization than worst-fit. First-fit is generally faster than best-fit

MEMORY ALLOCATION POLICIES : PROBLEM

Q: Given the memory partitions of 100K, 500K, 200K, 300K and 600K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K,112K, and 426K (in order)?

Which algorithm makes the most efficient use of memory?

212
417
112
426

processes

100
500
200
300
600

SOLUTION : FIRST FIT

- First-fit: search the list of available memory and allocate the first block that is big enough

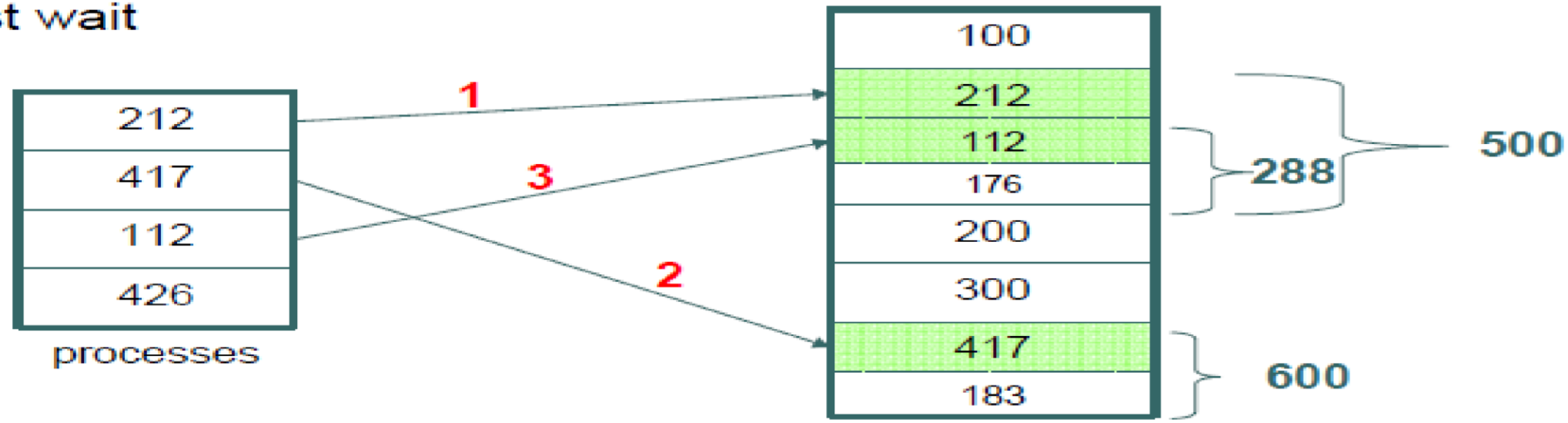
Processes placement:

212 K → 500 K partition

417 K → 600 K partition

112 K → 288 K partition (New partition 288 K = 500 K-212 K)

426 K must wait



SOLUTION : BEST FIT

- Best-fit: search the entire list of available memory and allocate the smallest block that is big enough

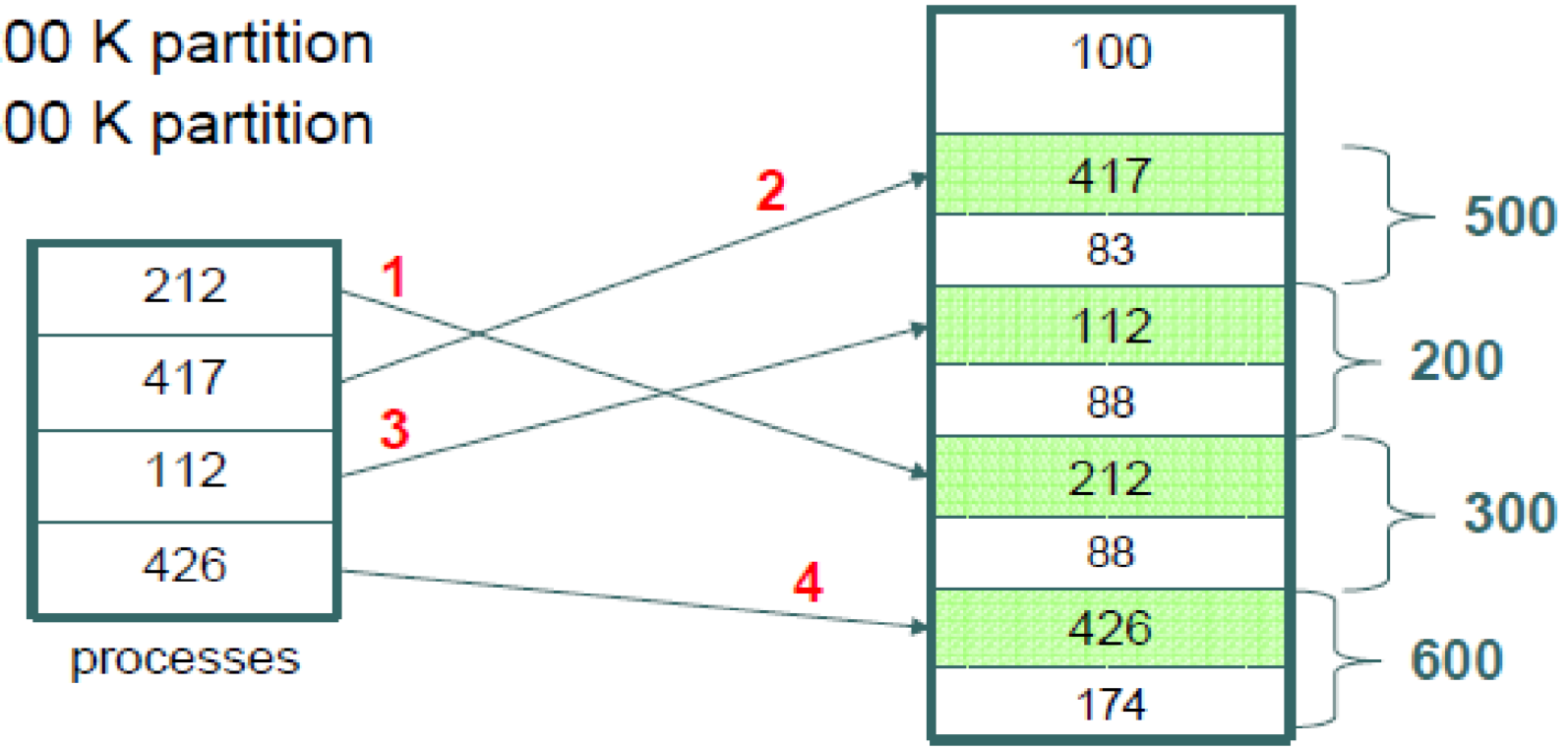
Processes placement:

212 K → 300 K partition

417 K → 500 K partition

112 K → 200 K partition

426 K → 600 K partition

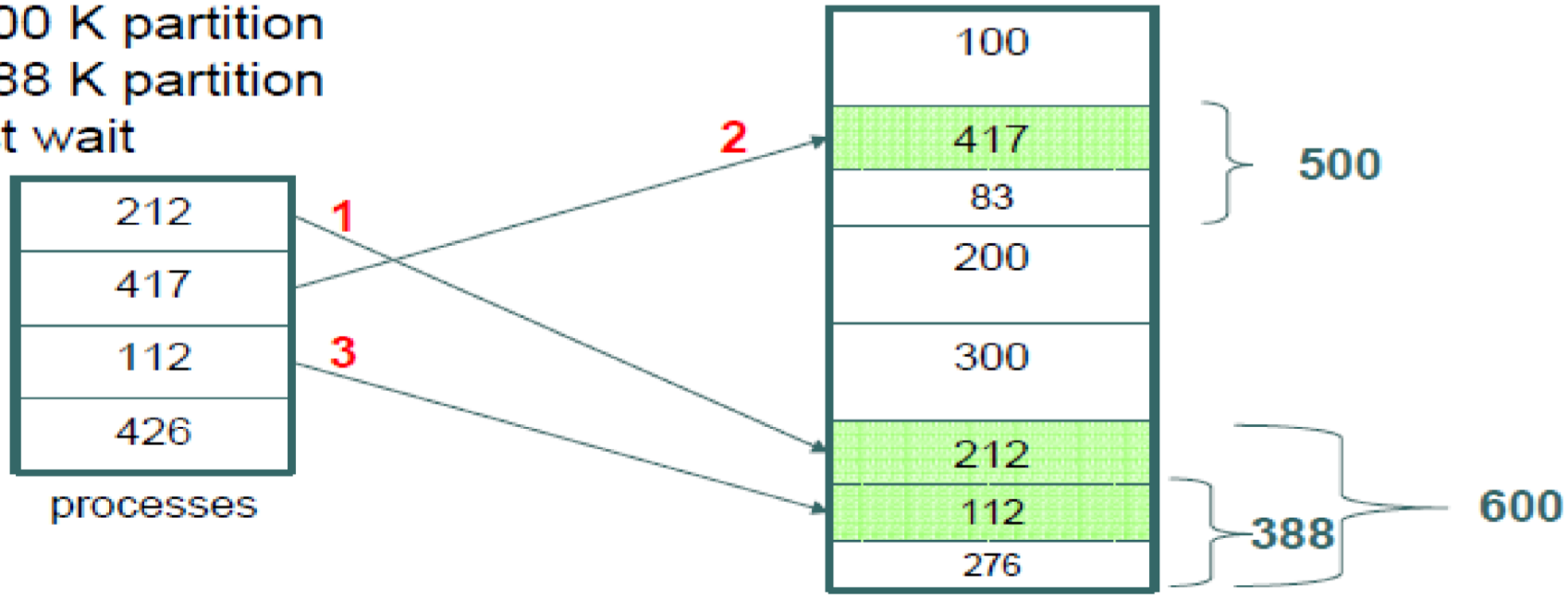


SOLUTION : WORST FIT

- Worst-fit: search the entire list of available memory and allocate the largest block. The justification for this scheme is that the leftover block produced would be larger and potentially more useful than that produced by the best-fit approach

Processes placement:

212 K → 600 K partition
417 K → 500 K partition
112 K → 388 K partition
426 K must wait



MEMORY ALLOCATION POLICIES : PROBLEM

Q: Consider a swapping system in which memory consists of the following hole sizes in memory order: 10K, 4K, 20K, 18K, 7K, 9K, 12K, and 15K. Which hole is taken for successive segment requests of:

I) 12K

II) 10K

III) 9K

for

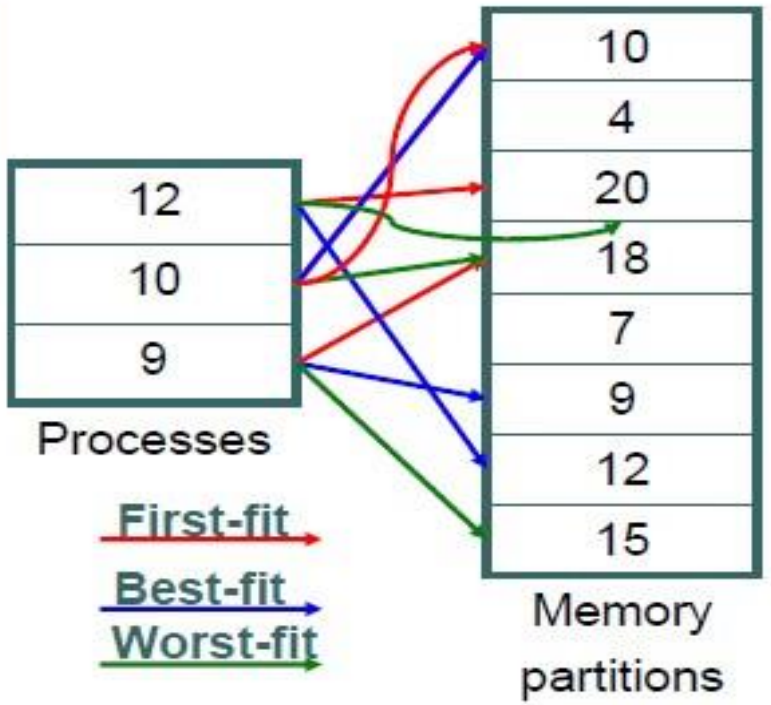
(a) First-fit?

(b) Best-fit?

(c) Worst-fit?

SOLUTION

	First-Fit	Best-Fit	Worst-Fit
12	20	12	20
10	10	10	18
9	18	9	15



INTERNAL VS. EXTERNAL FRAGMENTATION

- ✓External fragmentation exists when there is enough memory to fit a process in memory but the space is not contiguous. Internal fragmentation exists when memory internal to a partition is wasted
- ✓External fragmentation exists between the blocks of allocated memory and internal fragmentation is within the allocated block
- ✓Internal Fragmentation occurs when a fixed size memory allocation technique is used. External fragmentation occurs when a dynamic memory allocation technique is used.

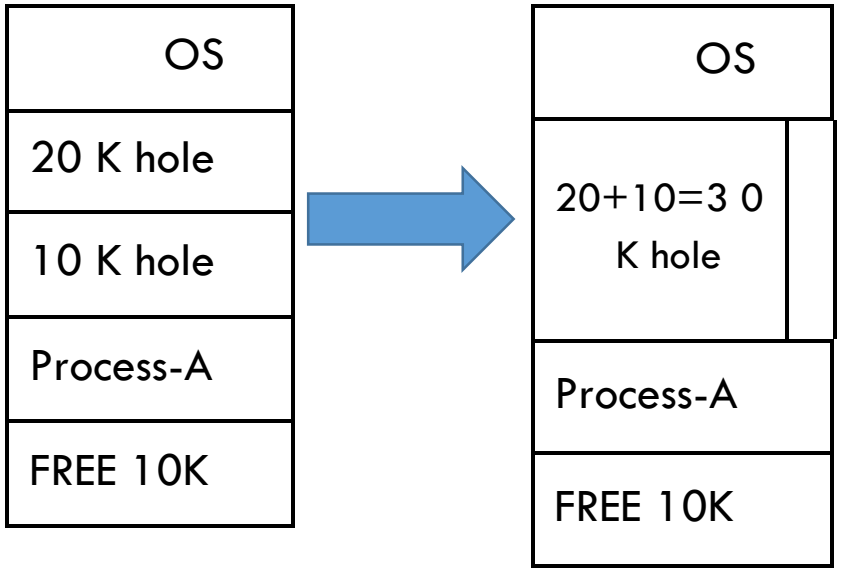
DEALING WITH FRAGMENTATION

- ✓External fragmentation can be mined by compaction
- ✓External fragmentation can be prevented by mechanisms such as segmentation and paging.
- ✓Internal fragmentation can be maimed by having partitions of several sizes and assigning a program based on the best fit. However, still internal fragmentation is not fully eliminated

COALESCING

Coalescing

✓The process of merging two adjacent holes to form a single larger hole is called coalescing.



COMPACTION

Compaction

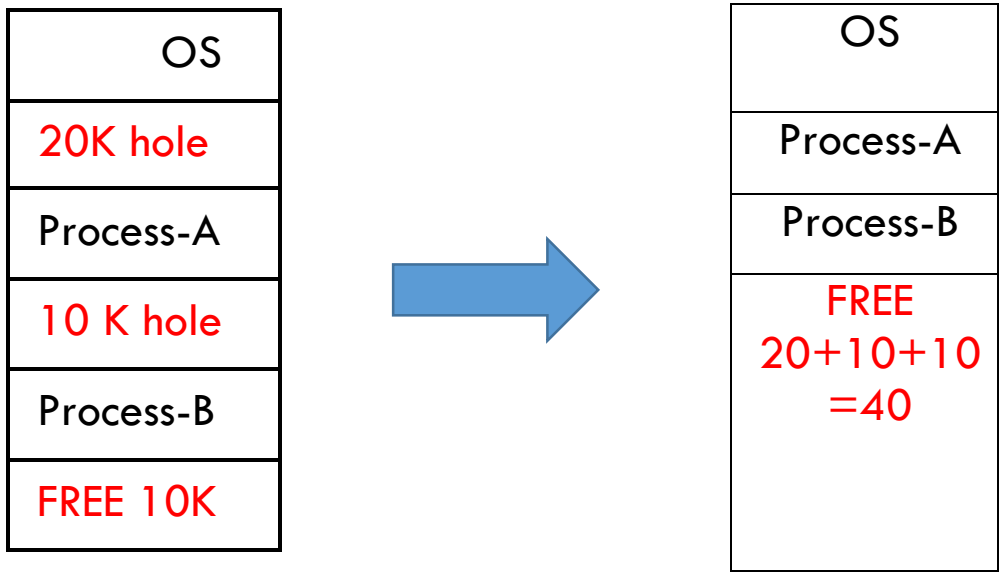
✓Even when holes are coalesced, no individual hole may be large enough to hold the job, although the sum of holes is larger than the storage required for a process.

✓It is possible to combine all the holes into one big one by moving all the processes downward as far as possible; this technique is called memory compaction. ✓Compaction maybe

-Full (Expensive)

-Partial (Cheaper)

COMPACTION


OS
20K hole
Process-A
10 K hole
Process-B
FREE 10K
OS
Process-A
Process-B
FREE
20+10+10
=40

DRAWBACK OF COMPACTION

- ✓Reallocation info must be maintained.
- ✓System must stop everything during compaction.
- ✓Memory compaction requires lots of CPU time.

SWAPPING

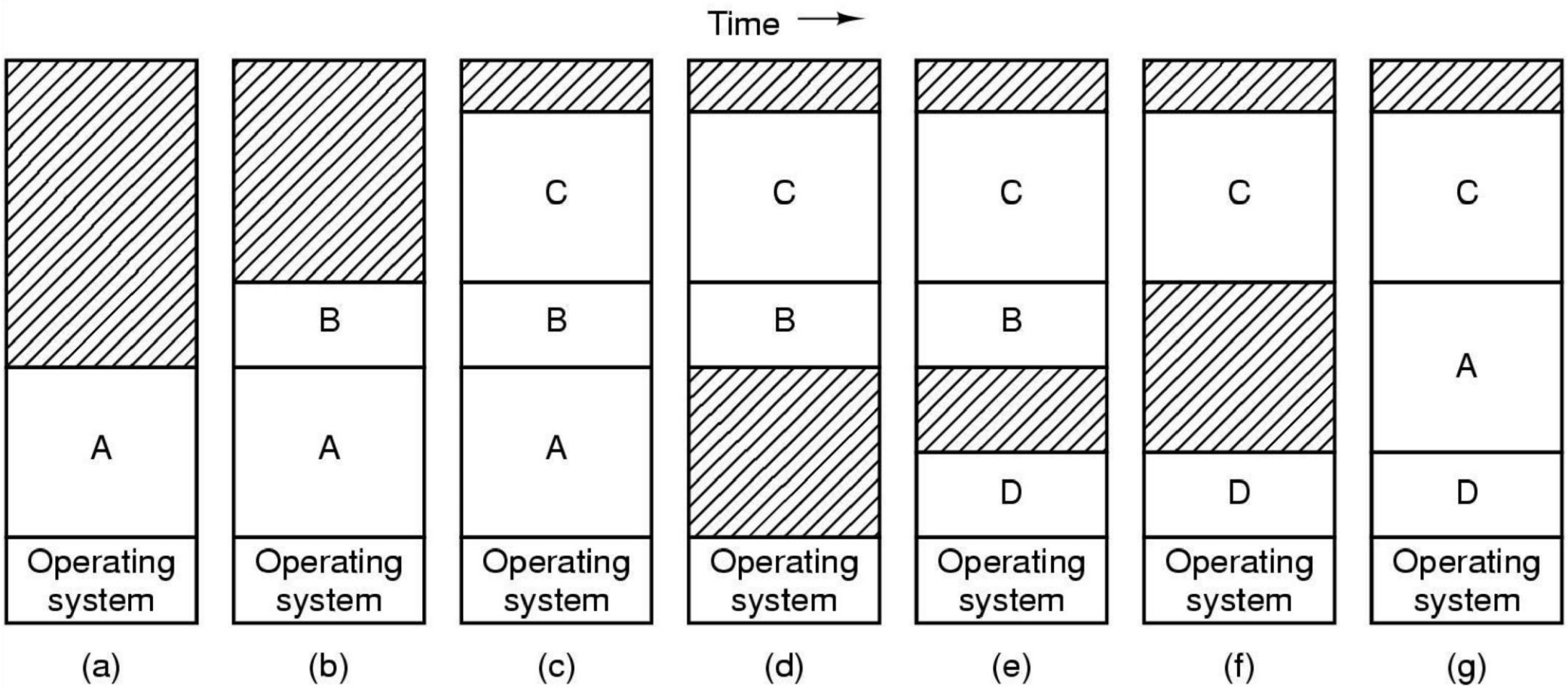
✓Swapping refers to rolling out a process to disk, releasing all the memory it holds ✓When process becomes active again, the OS must reload it in memory

- With *static relocation*, the process must be put in the same position
- With *dynamic relocation*, the OS finds a new position in memory for the process and updates the relocation and limit registers

✓Swapping is one of the methods to deal with the memory overload (memory compaction, swapping and overlays are three methods to deal with insufficient memory)

✓If swapping is a part of the system, compaction is really easy to add ✓What does swapping have to do with CPU scheduling?

SWAPPING



MANAGING FREE MEMORY

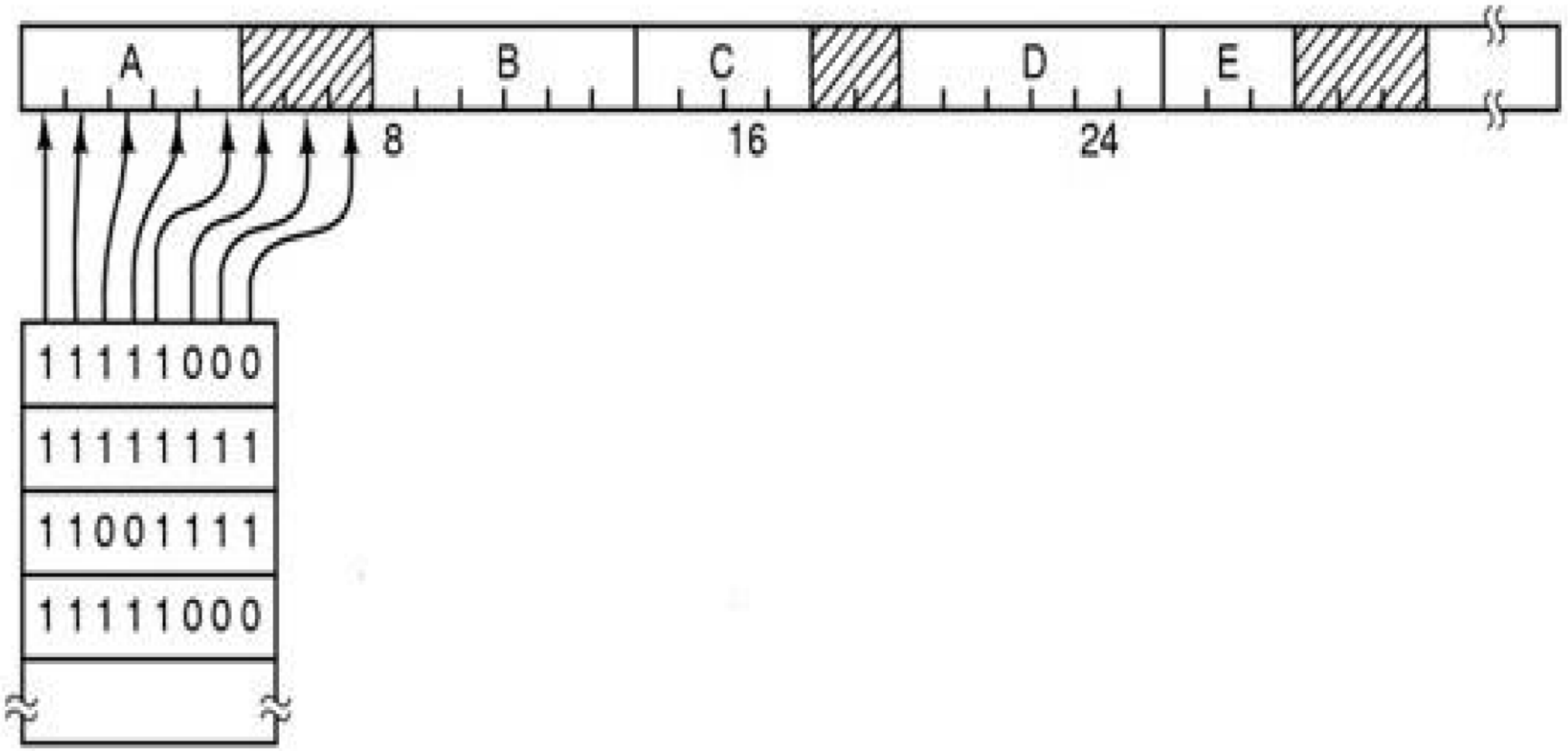
We can keep track of memory by the following two ways:

1. Memory management with Bitmaps
2. Memory management with Linked list

MEMORY MANAGEMENT WITH BITMAP

- ✓With bitmaps, memory is divided into *allocation units* which may be a few words long or several kilobytes
- ✓Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if the unit is occupied
- ✓Size of allocation unit is important design issue. The smaller the allocation unit, larger is the bitmap and vice versa.
- ✓Size of allocation unit may as well affect the memory utilization
- ✓When a *k* unit process is decided to bring into memory, the memory manager must search for *k* consecutive 0s in the bitmap
- ✓Searching bitmap for a run of given length may be very slow

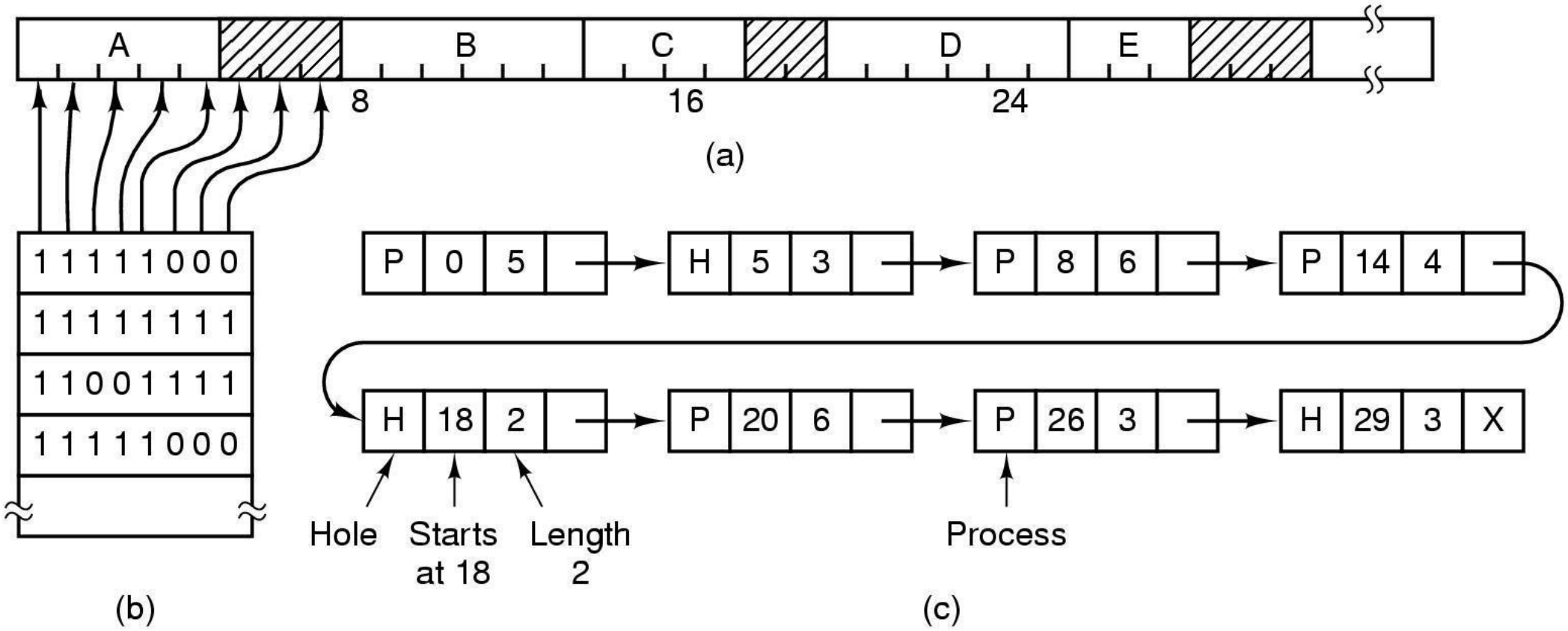
MEMORY MANAGEMENT WITH BITMAP



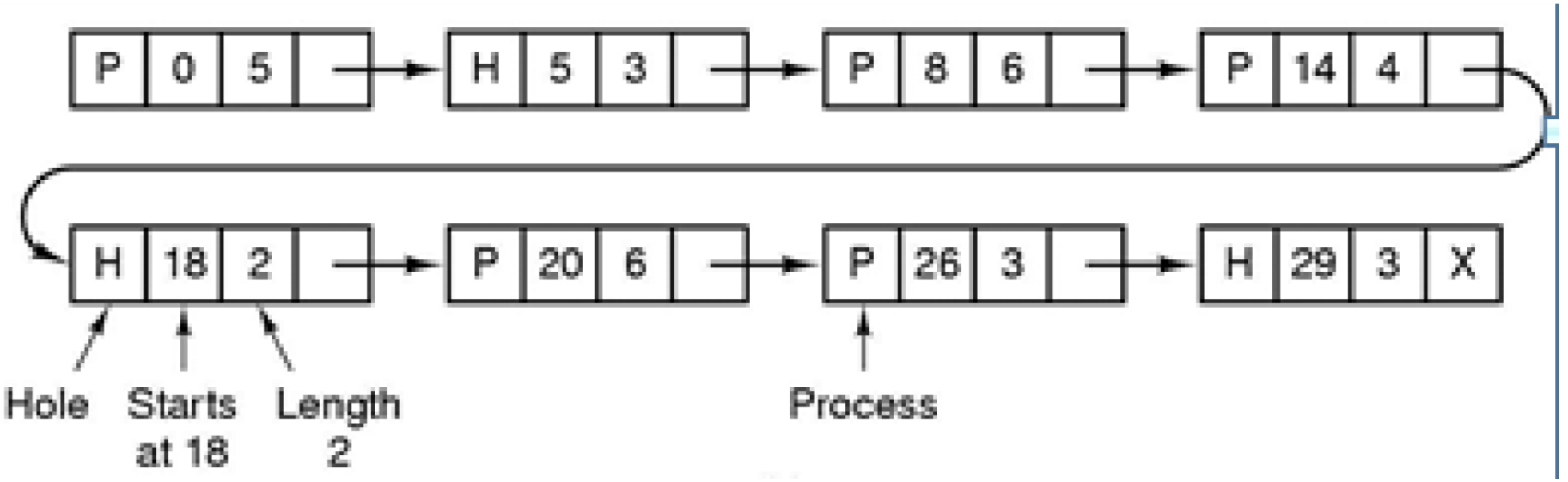
MEMORY MANAGEMENT WITH LINKED LIST

- ✓Another way to keep track of the memory is to maintain a linked list of allocated and free memory segments, where a segment either contains a process or is an empty hole between two processes
- ✓Each entry in the list specifies a process (P) or a hole (H) , the address at which it starts, the length and a pointer to the next entry
- ✓In the above slide, the segment list is kept *sorted by address*. Sorting that way has the advantage that when a process terminates or is swapped out, updating the list is easier.
- ✓A terminating process has two neighbors (except when it is at top or bottom of memory). These may be either processes or holes leading to four combinations shown in next slide

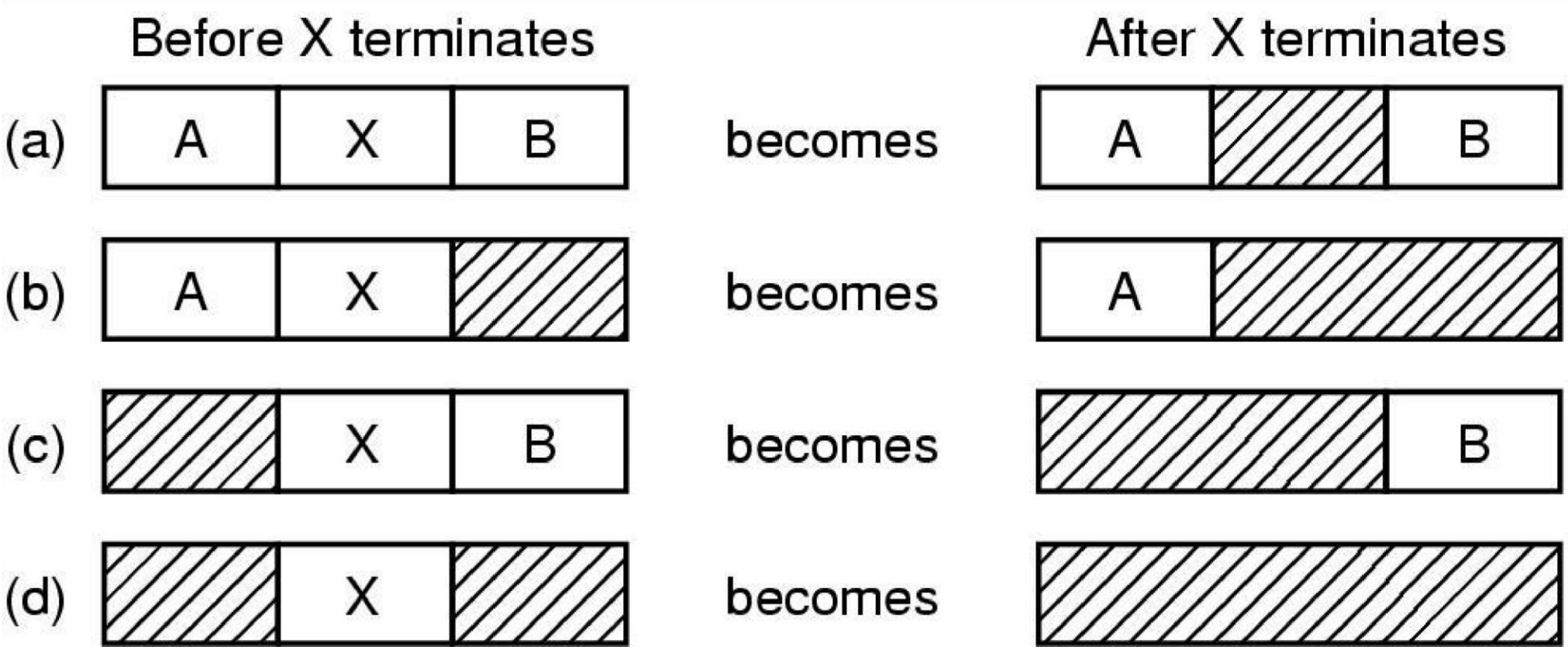
MEMORY MANAGEMENT WITH LINKED LIST



MEMORY MANAGEMENT WITH LINKED LIST



MEMORY MANAGEMENT WITH LINKED LIST



MEMORY MANAGEMENT WITH LINKED LIST

In the above diagram:

(a) Replace process X by hole H.

(b) And (c) Two entries are coalesced.

(d) Three entries are merged.

HOW TO DEAL WITH INSUFFICIENT MEMORY?

Swapping

-temporarily move processes to disk

-require dynamic relocation Memory Compaction

- movie blocks around to create larger holes

Virtual Memory

-allow programs larger than physical memory

- portions of programs loaded as needed

Also see : *overlays*

VIRTUAL MEMORY

- ✓Virtual memory is a memory management technique that allows programs larger than physical memory to be executed
- ✓Virtual Memory is a space where large programs can store themselves in form of pages while their execution and only the required pages or portions of processes are loaded into the main memory
- ✓Virtual memory takes the advantage of the fact that processes typically do not use all the allocated memory. In other words, most processes never need all their pages at once.
- ✓Logical address space can therefore be much larger than physical address space

PAGING

✓The basic idea behind paging is to only keep those part of the processes in memory that are actually being used

✓Paging is non-contiguous memory allocation technique ✓Paging

- divides processes to a fixed sized ***pages***

-then selectively allocates pages to ***frames*** in memory, and

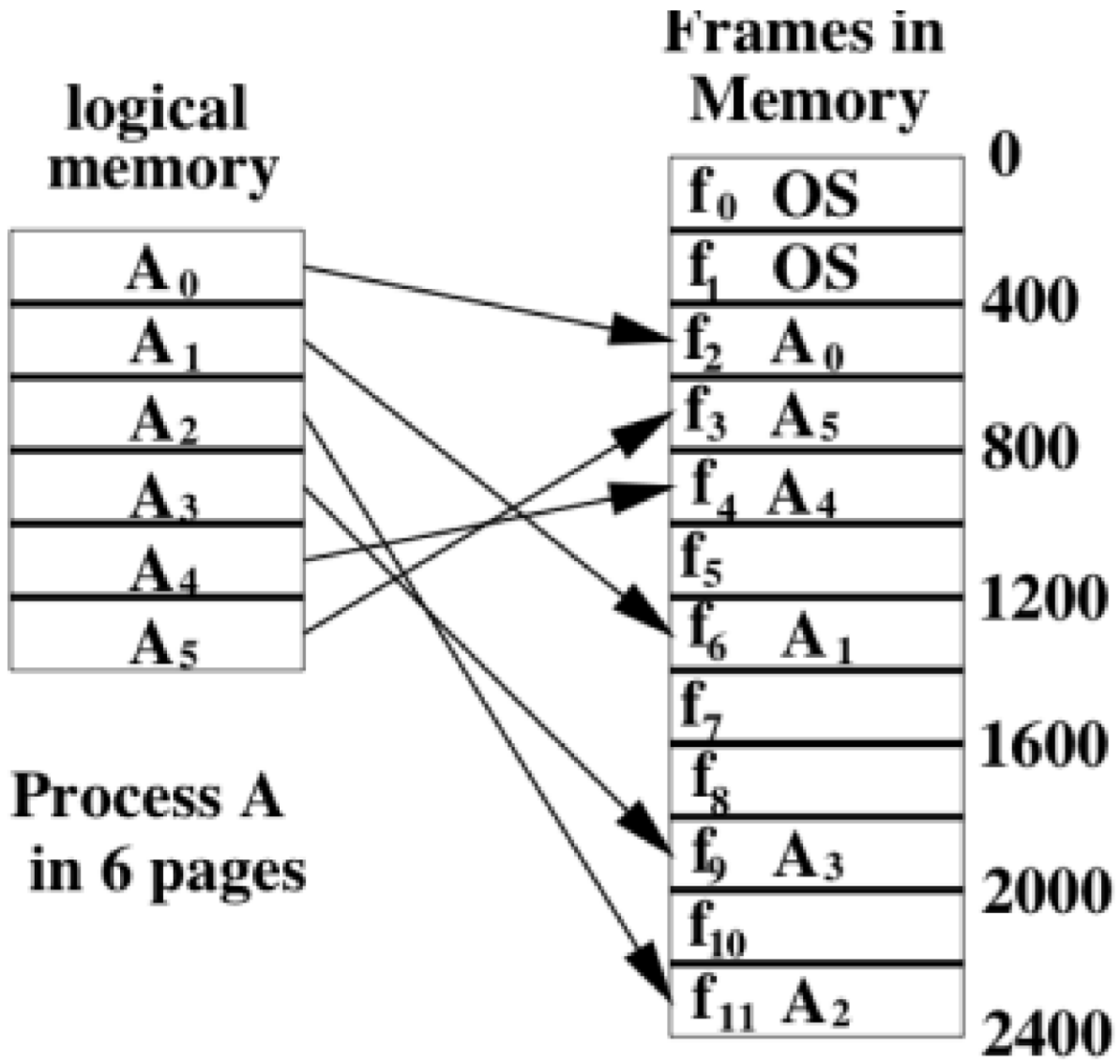
-manages (moves, removes, reallocates) pages in memory

✓*Pages* and *frames* are blocks of the same size

PAGING

- ✓The logical memory of the process is still contiguous but the pages need not be allocated contiguously in memory
- ✓By dividing the memory into fixed sized pages, we can eliminate external fragmentation
- ✓However, paging dose not eliminate internal fragmentation (e.g 4KB frame size and 15 KB of process)

- ✓Summary: breaking process memory into fixed sized pages and map them to a physical frame of memory



Mapping pages from logical memory to frames in physical memory

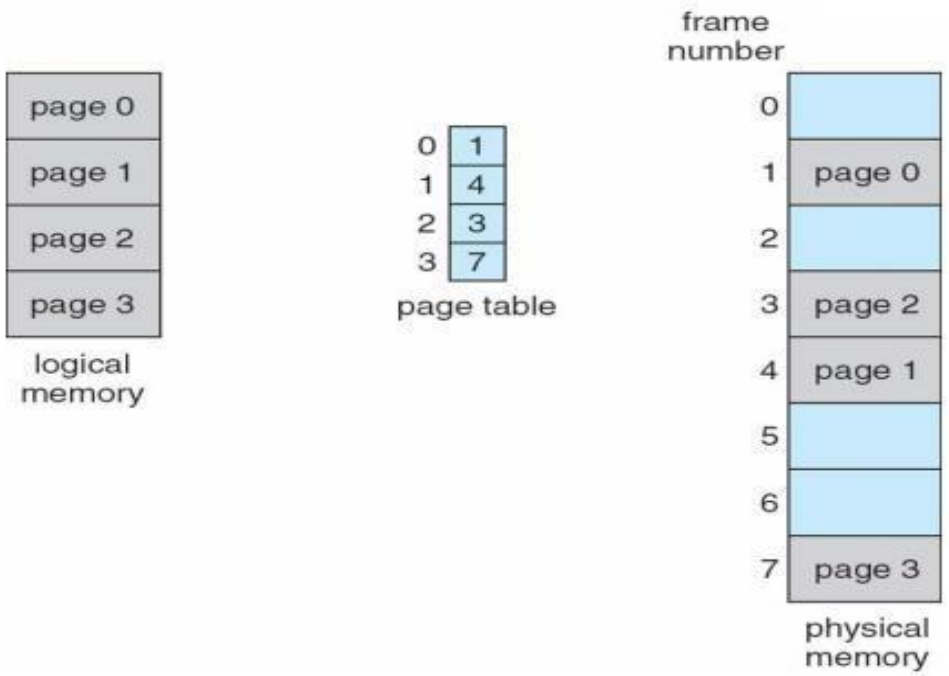
PAGING

OS needs to do two things

1. Mapping the pages to frame, maintain a ***page table***
2. Translate the virtual address (contiguous) into physical address (not so easy as the physical address for a process is scattered all over the memory)

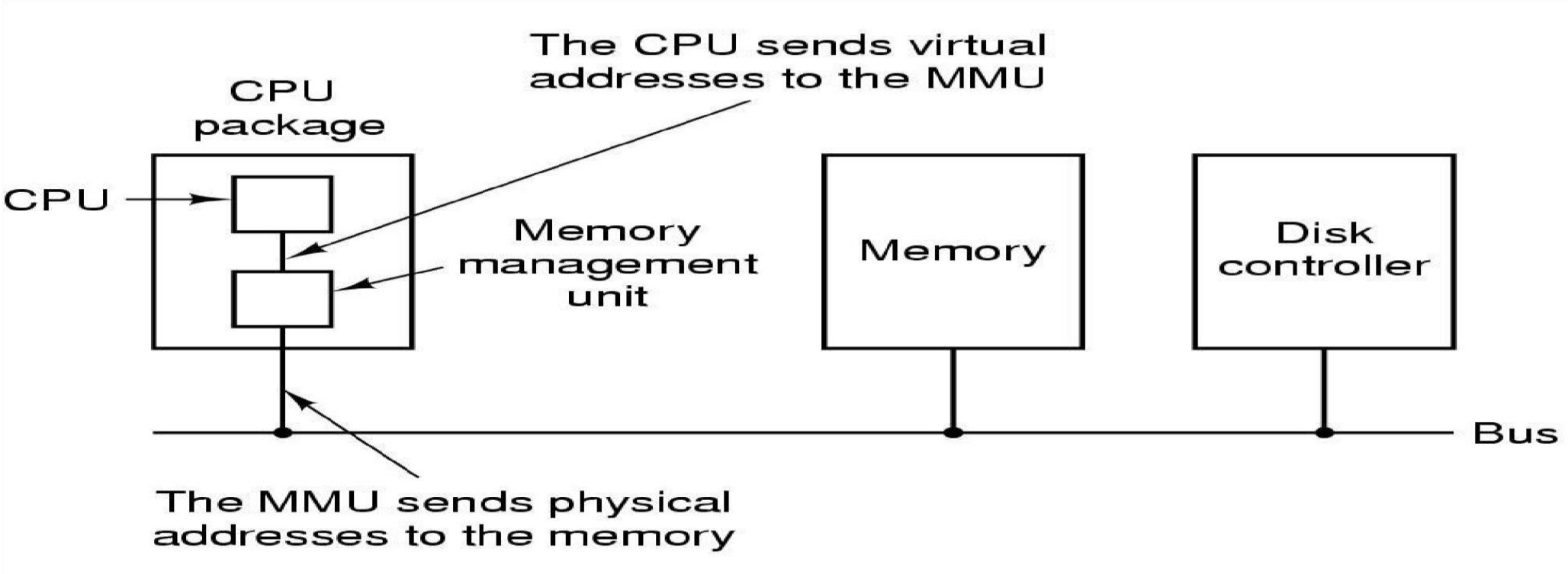
PAGE TABLE

Page table is a data structure that contains frame location for each page in a process

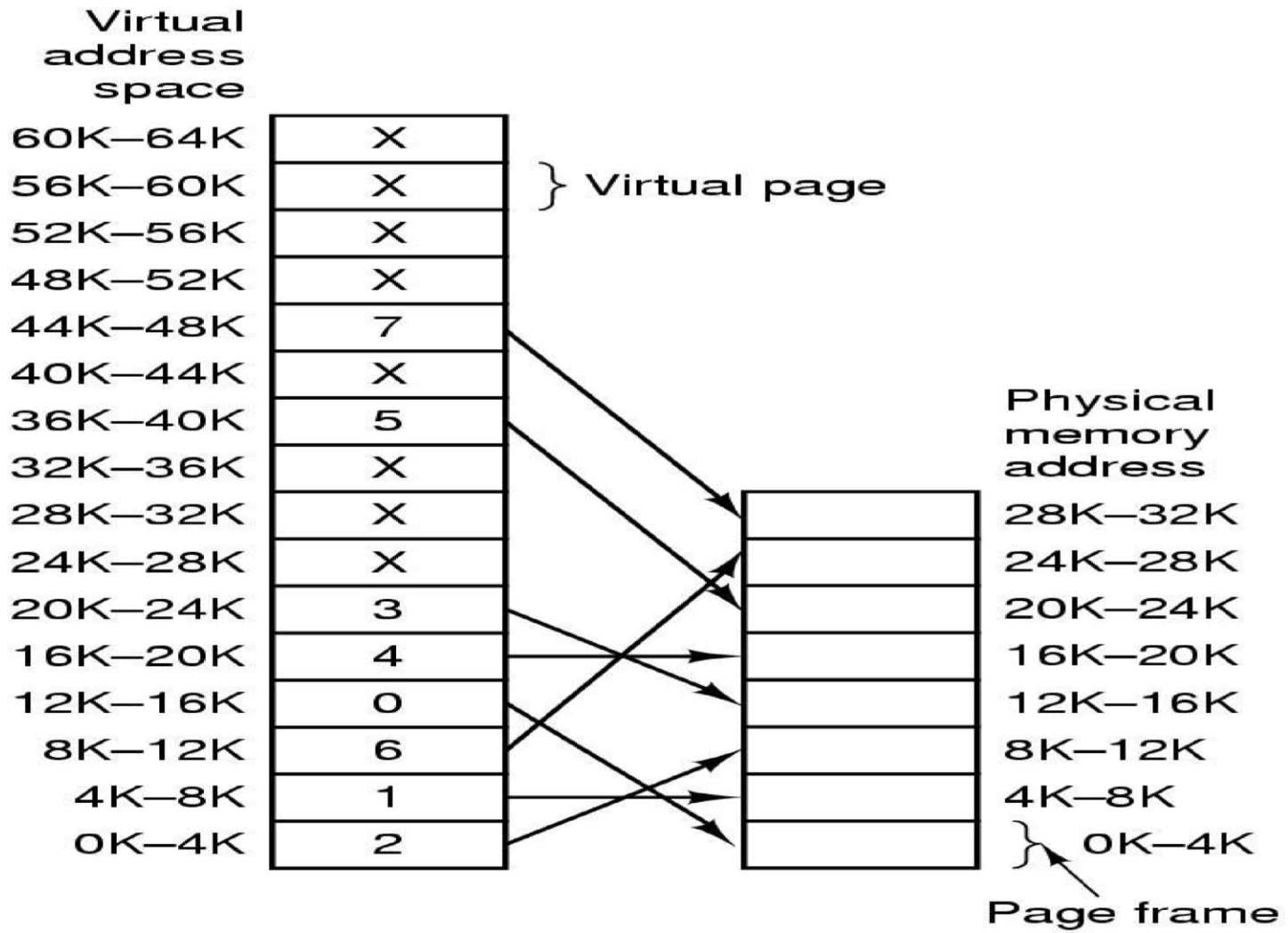


ADDRESS TRANSLATION

The conversion from virtual address to the physical address is done by MMU



An example: 16 bit addresses, 4 KB pages32 KB physical memory, 16 virtual pages and 8 page frames



ADDRESS TRANSLATION

Consider an instruction **MOV REG,0**

- ✓Virtual address 0 is sent to MMU. The MMU sees that this virtual address falls in page 0(0 to 4095), which is mapped to page frame 2(8192 to 12287).
- ✓Thus it transforms the address to 8192 & outputs 8192 onto the bus.
- ✓Similarly **MOV REG 8192** (virtual page 2) is effectively transformed into **MOV REG, 24576** (physical page frame 6)
- ✓Also, a virtual address 20500 is 20 bytes from the start of the virtual page 5 (virtual addresses 20480 to 24575) and maps onto physical address

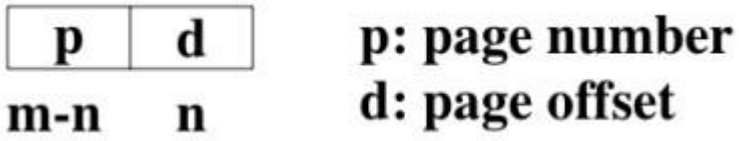
12288+20=12308

ADDRESS TRANSLATION

✓Every address generated by the CPU is divided into two parts:

1. **page number (p)**
2. **page offset (d)**

✓ virtual address space of size 2^m bytes and a page of size 2^n, then the high order m-n bits of a virtual address select the page, and the low order n bits select the offset in the page

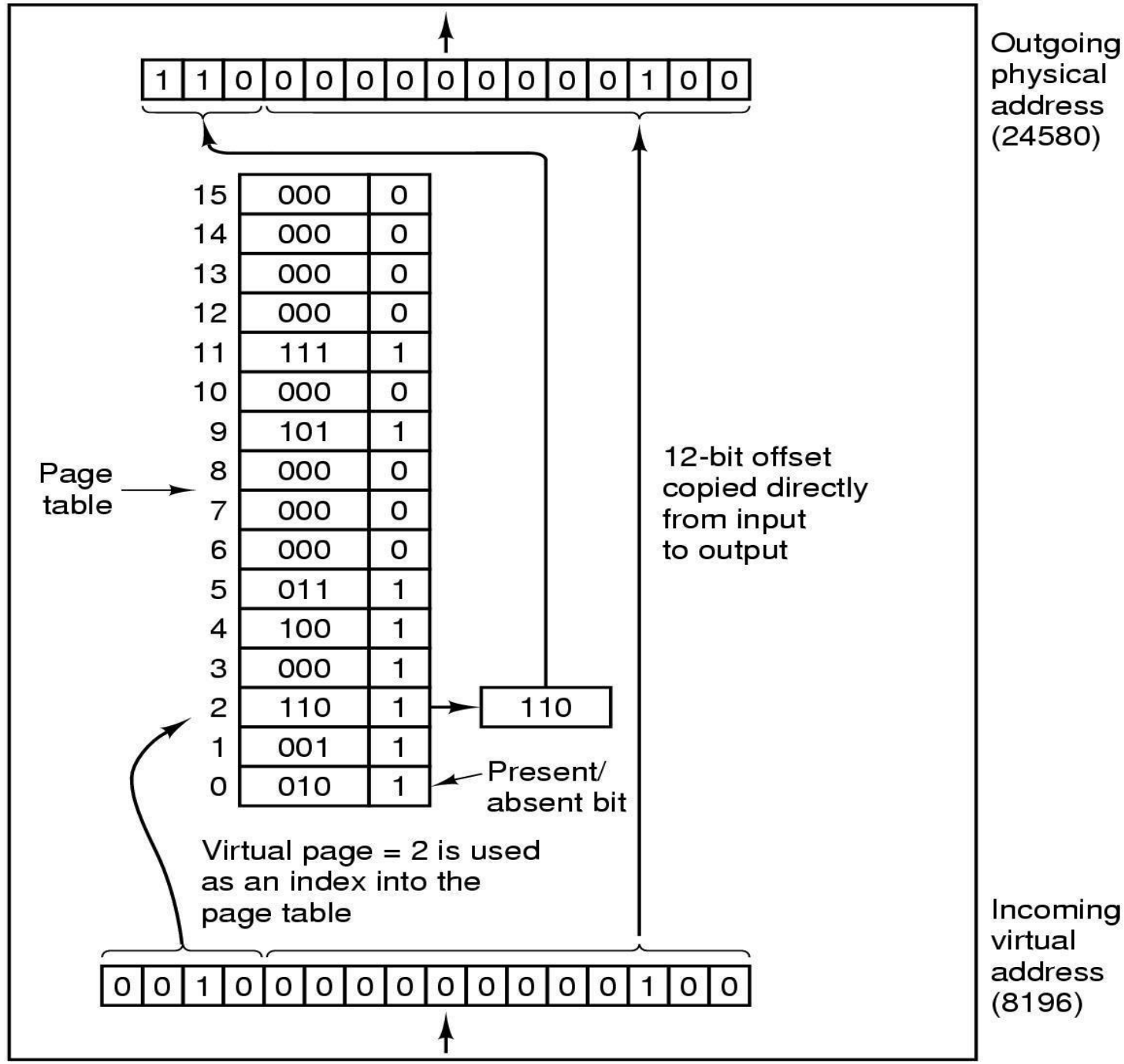


✓The page number is used as an index into a **page table.**

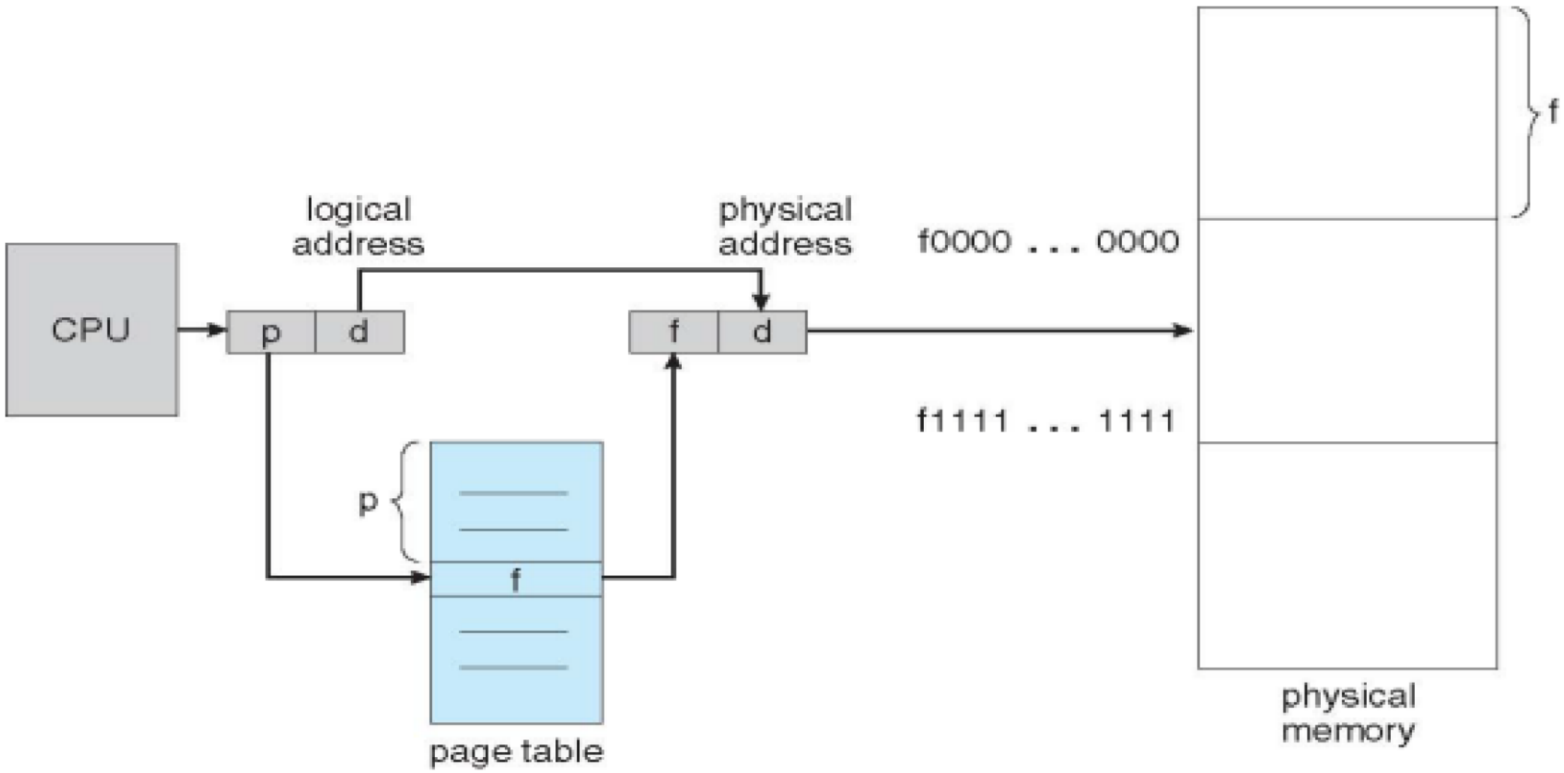
✓ The page table contains the base address of each page in physical memory.
✓This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

MAPPING/PAGING MECHANISM

- ✓Let us see how the incoming virtual address 8196(0010000000000100 in binary) is mapped to physical address 24580.
- ✓Incoming 16-bit virtual address is split into 4-bit page number and 12-bit offset.
- ✓With 4-bits, we can have 16 pages and with 12-bits for the offset, we can address all 4096 bytes within a page.
- ✓The page number is used as an index into the page table, gives number of virtual pages.
- ✓Page table contains Present/Absent bit also to keep track of whether a particular page number is present in memory or not


Outgoing physical address (24580)
1 1 0 0 0 0 0 0 0 0 0 0 1 0 0
15 000 0
14 000 0
13 000 0
12 000 0
11 111 1
10 000 0
9 101 1
8 000 0
7 000 0
6 000 0
5 011 1
4 100 1
3 000 1
2 110 1
1 001 1
0 010 1
Page table
110
Present/ absent bit
12-bit offset copied directly from input to output
Virtual page = 2 is used as an index into the page table
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
Incoming virtual address (8196)

ADDRESS TRANSLATION



PAGE FAULT

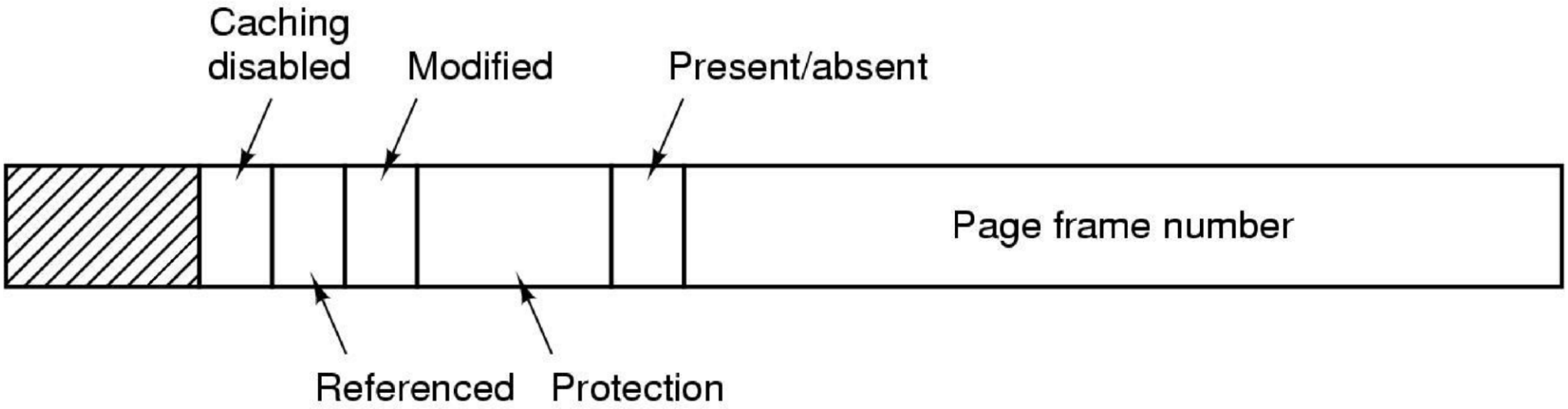
- ✓When the page (data) requested by a program is not available in the memory, it is called as a page fault.
- ✓In previous example, what happens if a program references an unmapped address? For example the instruction

MOV REG, 32780

which is 12 byte within virtual page 8 (starting at 32768).

- ✓The MMU notices that the page is unmapped and causes the CPU to trap the OS
- ✓This trap is called page fault
- ✓We know that there is page fault by consulting a page table. If there is no entry in the page table or the entry points to DISK, we know that there is page fault

STRUCTURE OF PAGE TABLE ENTRY



STRUCTURE OF PAGE TABLE ENTRY (PTE)

- **Frame number:** The actual page frame number.
- **Present (1) / Absent (0) bit:** Defines whether the virtual page is currently mapped or not.
- **Protection bit:** Kinds of access permission; read/write/execution.
- **Modified bit/dirty bit:** Defies the changed status of the page since last access.
- **Referenced bit:** is set when a page is referenced. Used for replacement strategy.
- **Caching disabled:** allows caching to be disabled for the page

The size of PTE varies from computer to computer, but 32 bits (4 bytes) is a common size

ISSUES WITH PAGING

✓In any paging system, two major issues must be faced:

1. Many memory access (slower process)
2. Large page tables

SPEEDING UP PAGING

-For each memory access with virtual memory, we have to :

-Access the page table in RAM

-Translate the address

-Access the data in RAM

- This is a lot of work for every memory access

-The bottom-line is that paging can make computer slower

Q:How can we make up page-table lookup faster??

A:We will need a cache for page table (hardware based) and that is **Translation Lookaside Buffer (TLB)**

TRANSLATION LOOKASIDE BUFFER (TLB)

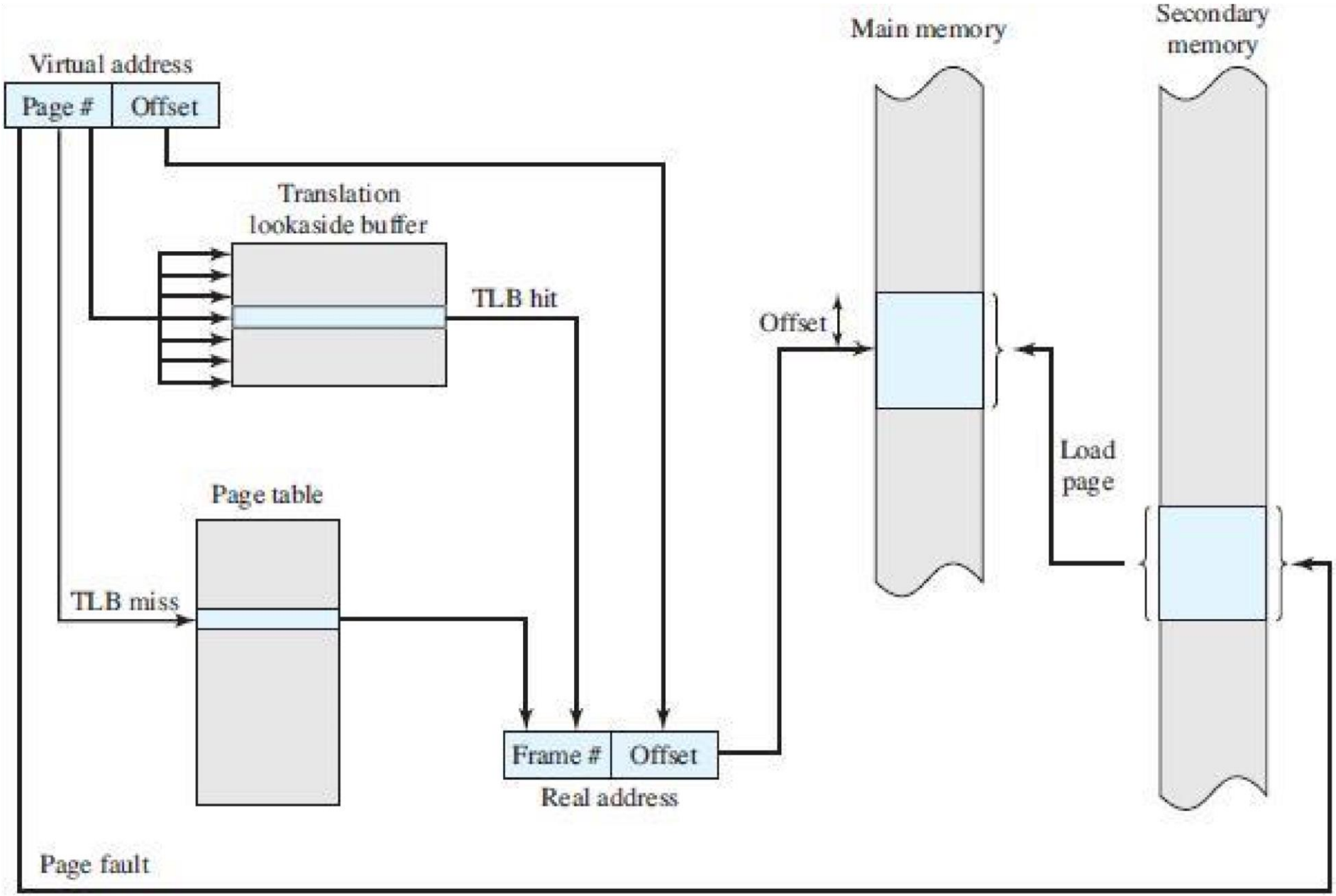
- ✓Every virtual memory reference can cause two physical memory access: one to fetch the appropriate page table entry and one to fetch the desired data
- ✓Most program tend to make a large number of references to a smaller number of pages, and not the other way around
- ✓That is, only a small fraction of the page table entries are heavily read; the rest are barely used
- ✓Using this observation, a solution has been devised to equip computer with a small hardware device for mapping virtual addresses to physical addresses without going through page table
- ✓The device is called **Translation Lookaside Buffer (TLB)** or sometimes called an *associative memory*

TRANSLATION LOOKASIDE BUFFER (TLB)

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

TRANSLATION LOOKASIDE BUFFER (TLB)

- ✓TLB is usually inside the MMU and consists of small number of entries
- ✓Each entry contains information about one page, including a virtual page number, a bit that is set when the page is modified, the protection code, and physical frame in which the page is located
- ✓When a virtual address is provided to the MMU for translation, the hardware first checks to see if its virtual page number is present in the TLB
- ✓If a valid match is found, and the access does not violate the protection bits, the page frame is taken directly form the TLB
- ✓If virtual page frame is present in table but instruction is trying to write on a read only page, it generates a protection fault
- ✓If virtual page number is not found in the TLB, MMU detects the miss and does an ordinary page table lookup


Main memory
Secondary memory
Virtual address
Page #
Offset
Translation lookaside buffer
TLB hit
Offset
Load page
Page table
TLB miss
Frame #
Offset
Real address
Page fault

DIRECT VS. ASSOCIATIVE MAPPING

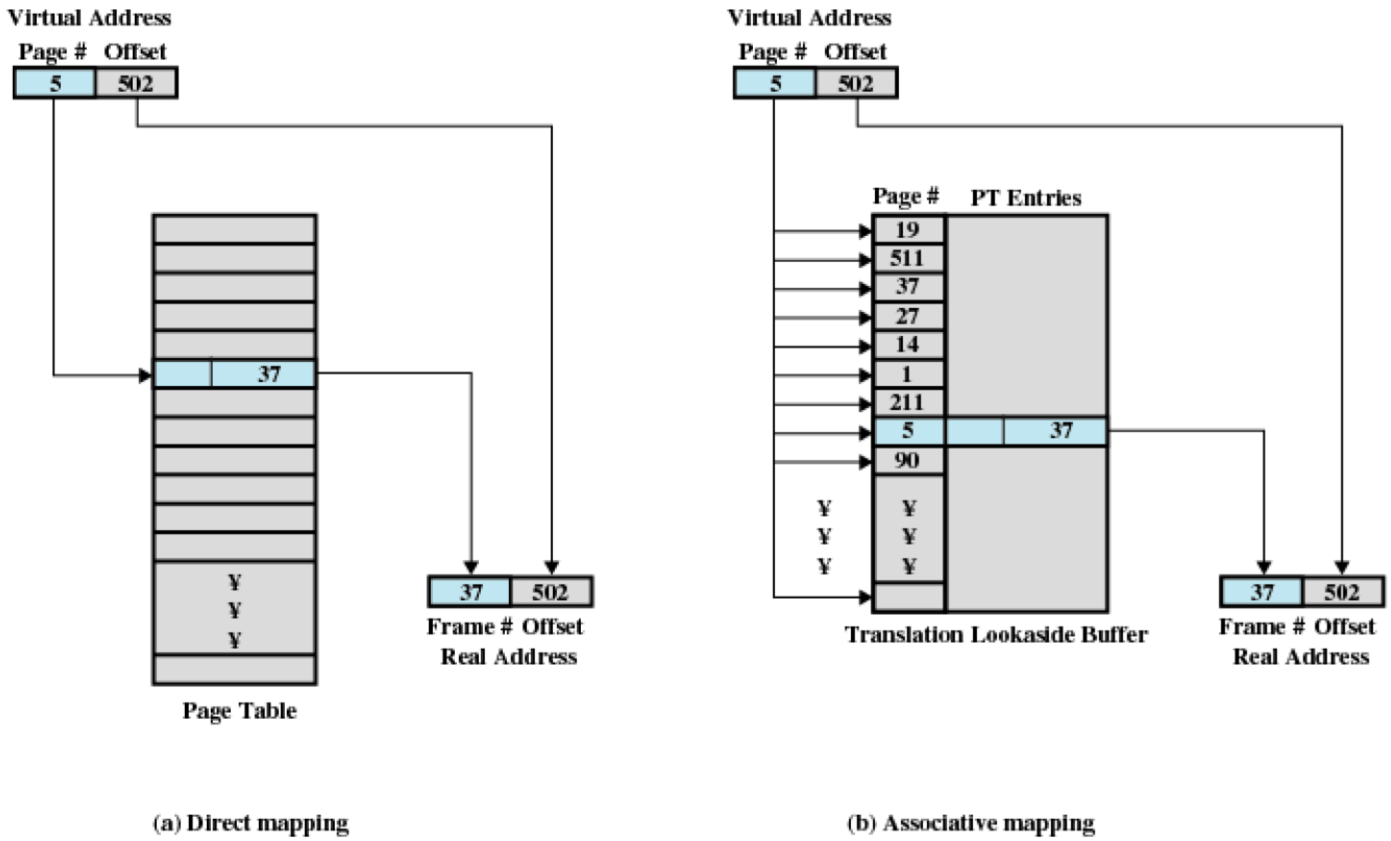
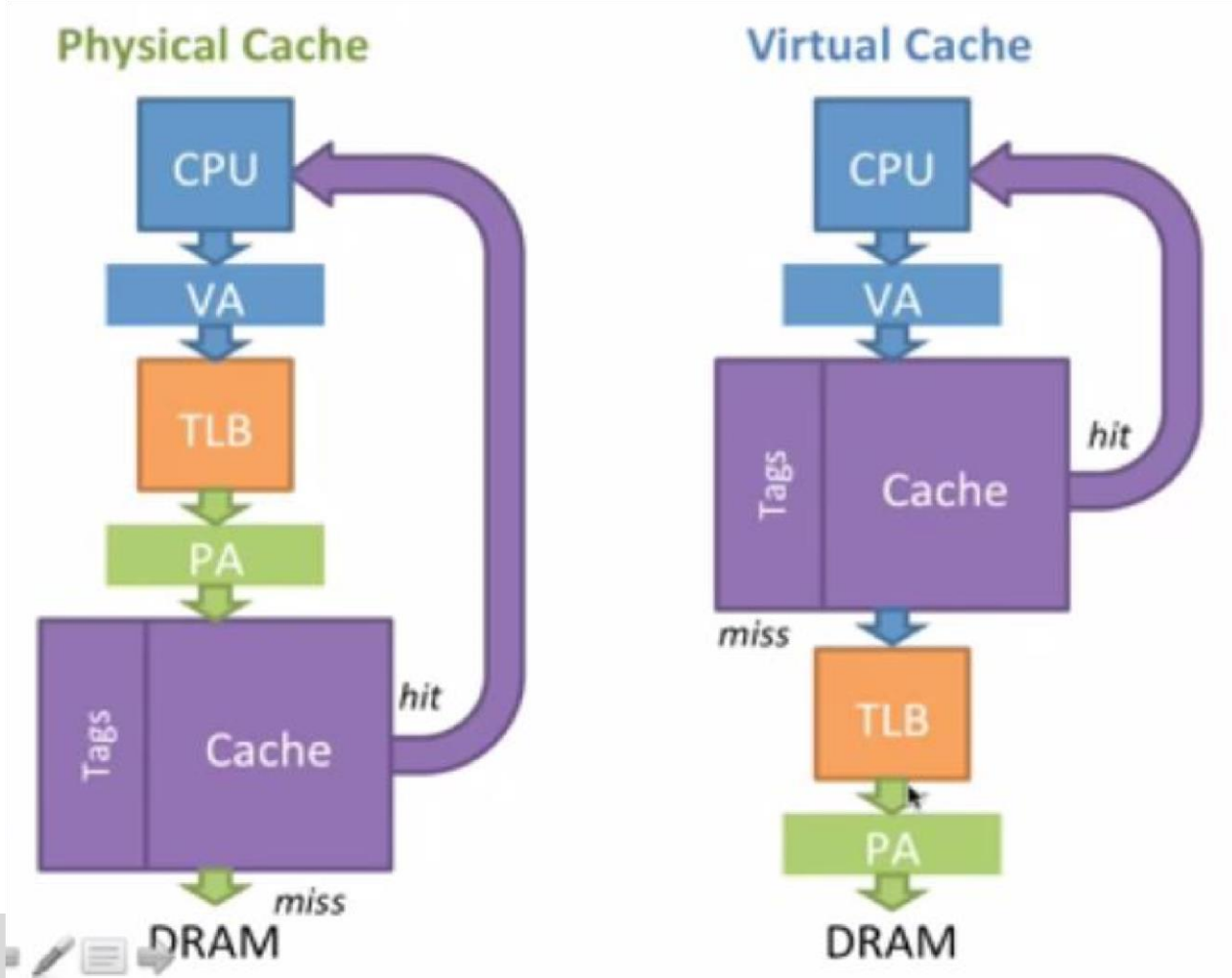


Figure 8.9 Direct Versus Associative Lookup for Page Table Entries

TLB AND



Physical Cache

Slow: Must do a TLB lookup *before* accessing the cache

Virtual Cache

Fast: TLB lookups *only* when we miss in the cache

Q What is the problem with virtual cache?
A : Two programs cannot share a virtual cache

Solution: We can have a separate bit in the VA for process ID or we can flush the cache when we switch process

Best of both worlds: VIPT cache (Virtually indexed physically tagged)

DEALING WITH LARGER PAGE TABLES

-If the virtual address space is large, the page table will be large

-For example, if we use 32 bit virtual addresses and 4KB page size, we will have around 1 million page table entries

-Each PTE is about 4 bytes, making a page table of size 4MB

-4MB is not bad, but each program need its own page table (100 programs=400MB) We can deal with such situations using:

1. Multilevel Page Tables
2. Inverted Page Tables (self study)

MULTILEVEL PAGE TABLES

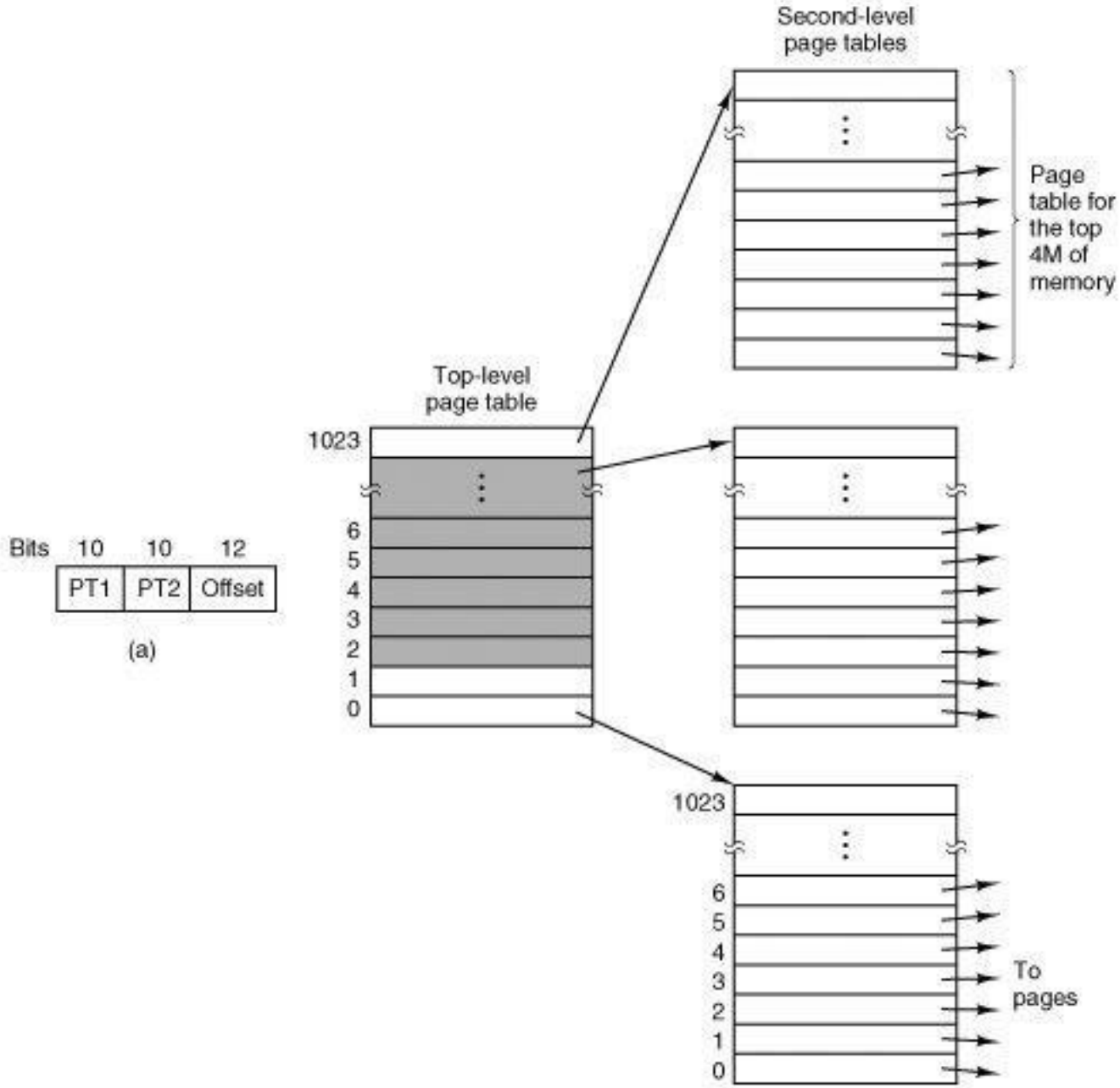
- The idea is to avoid keeping all the page tables in memory all the time
- In this method, we divide the virtual address page section into two or more sections as:
- assuming 32 bit virtual address, with 4Kb pages, we have 20 bit for page

-we divide the 20 bit page section into 2 separate 10 bits

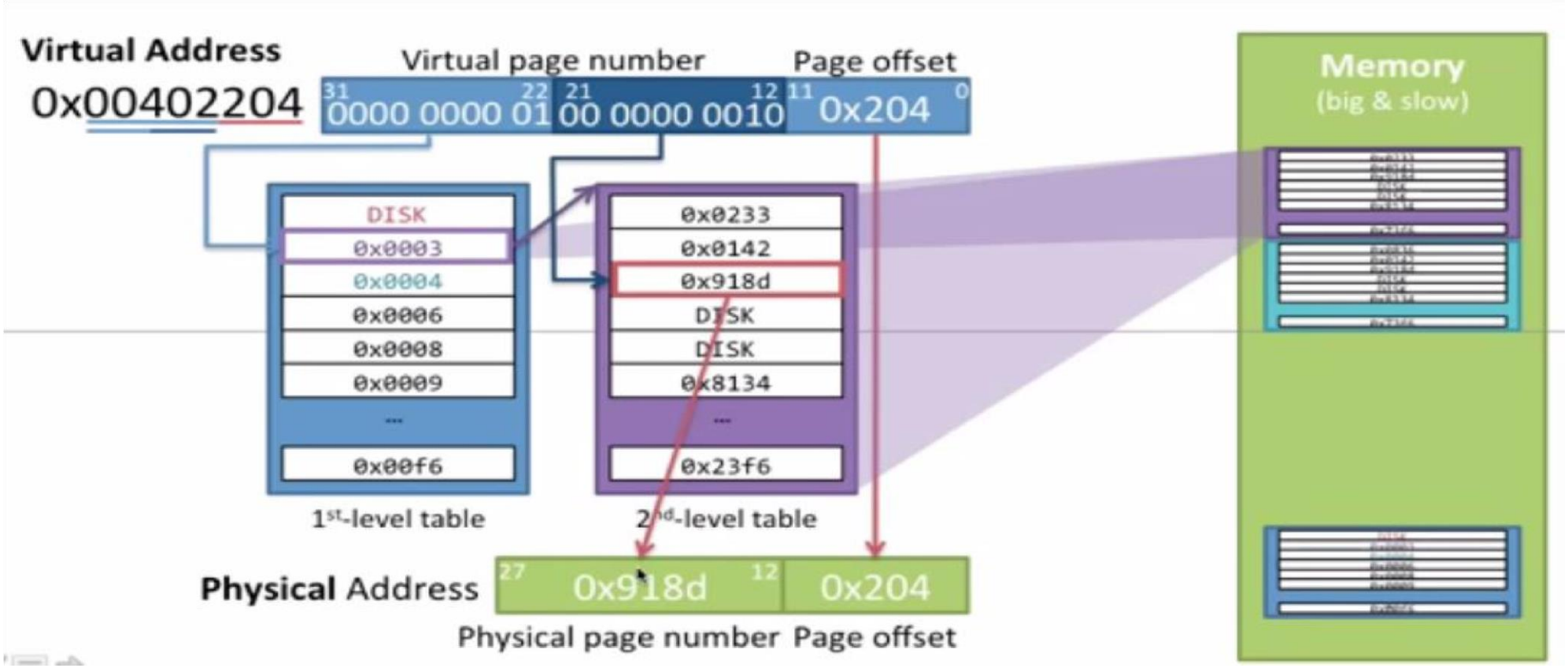
- the first 10 bit points towards the first level page table and second 10 bits point towards the second level page tables

-We would always need the first level page table in memory with at least one second level page table

Q. With multilevel page tables, what is the smallest amount of page table data we need to keep in memory for each 32 bit application ?


Second-level
page tables
Page
table for
the top
4M of
memory
Top-level
page table
1023
6
5
4
3
2
1
0
Bits 10 10 12
PT1
PT2
Offset
(a)
1023
6
5
4
3
2
1
0
To
pages
(b)

ADDRESS TRANSLATION EXAMPLE



DEMAND PAGING

- ✓Load pages to memory only when they are needed
- ✓With demand-paged virtual memory, pages are only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory.
- ✓A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory
- ✓When we want to execute a process, we swap it into memory.
- ✓Rather than swapping the entire process into memory, however, we use a **lazy swapper.**
- ✓A lazy swapper never swaps a page into memory unless that page will be needed.
- ✓Its better to use the term ***pager*** than ***swapper*** in demand pages since we are viewing a process as a sequence of pages, rather than as one large contiguous address space

DEMAND PAGING

- ✓We must solve two major problems to implement demand paging:
- ✓We must develop a **frame-allocation algorithm** (slide no 120) and a **pagereplacement algorithm.**
- ✓If we have multiple processes in memory, we must decide how many frames to allocate to each process.
- ✓Further, when page replacement is required, we must select the frames that are to be replaced.

PAGE REPLACEMENT

- ✓When a page fault occurs, the OS has to choose a page to remove from memory to make room for incoming page
- ✓If the page to be removed has been modified while in memory (check the modified bit), it must be rewritten to the disk to make the copy of the page in the disk up to date
- ✓If the page has not been changed, the disk copy is up to date and no rewrite is needed, the page to be brought in simply replaces the old page in memory
- ✓Choosing a page randomly to be removed might not be effective. We will discuss the some algorithms in doing so.

BASIC PAGE REPLACEMENT

1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
     - Write victim frame to disk if dirty

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

PAGE REPLACEMENT ALGORITHMS

1. The optimal page replacement algorithm
2. The least recently used (LRU) page replacement algorithm
3. First-In First-Out page (FIFO) replacement algorithm
4. Counting based Page replacement
5. Second chance page replacement algorithm
6. The clock page replacement algorithm

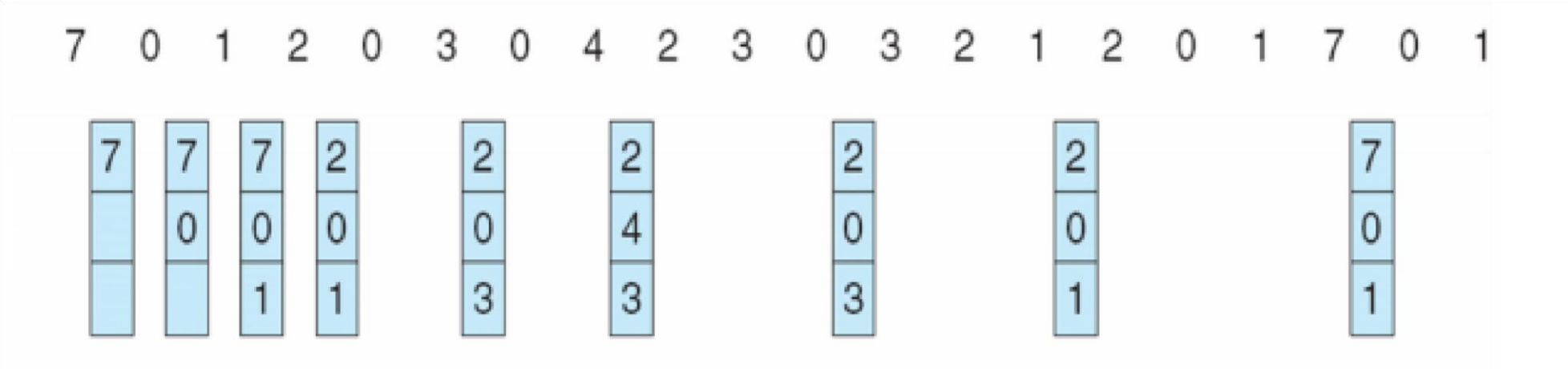
Note: We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string.**

OPTIMAL PAGE REPLACEMENT ALGORITHM

- ✓It is the best possible page replacement algorithm, which is easy to describe but impossible to implement
- ✓Replace page that will not be used for longest period of time
- ✓At the moment that page fault occurs, some set of pages are in the memory that will be referenced some time in the future (i.e. after some number of instruction)
- ✓This algorithm assumes that each page can be labelled with the number of instructions that will be executed before that page is referenced
- ✓The algorithm then says to remove the page with the highest label
- ✓For example, if a page will not be referenced for 8 million instruction and another for 6 millions, removing the former pushes the page fault as far into the future as possible

OPTIMAL PAGE REPLACEMENT ALGORITHM

Example: page frame size is 3 and the *reference string* is **7 ,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0, 1**



There will be 9 page faults using this algorithm

Task : page frame 3 and reference string : 2 3 5 4 2 5 7 3 8 7

LEAST RECENTLY USED (LRU) ALGORITHM

✓Use past knowledge rather than future

✓Replace page that has not been used in the most amount of time

1.Counter implementation

-Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter

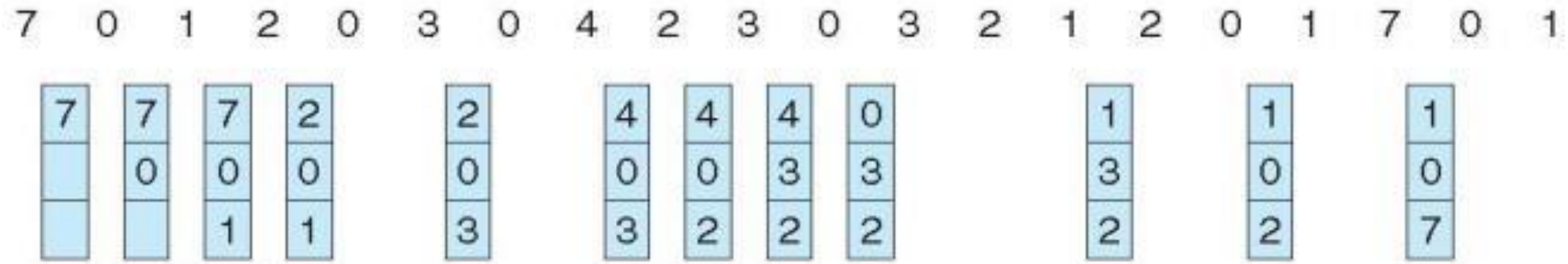
-When a page needs to be changed, look at the counters to find smallest value

2.Stack Implementation

- maintaining a stack of page references

LEAST RECENTLY USED(LRU) ALGORITHM

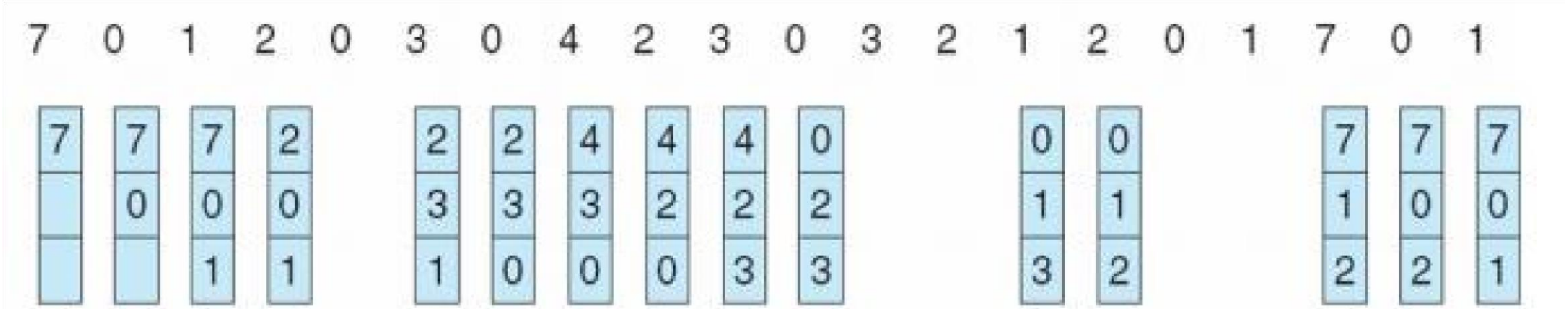
Example: page frame size is 3 and the *reference string* is **7 ,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0, 1**



There are 12 page faults

Task: 2 3 2 1 5 2 4 5 3 2 5 2 (3 page faults)

FIRST IN FIRST OUT (FIFO) ALGORITHM



There are 15 page faults in this case

Task: 2 3 2 1 5 2 4 5 3 2 5 2

COUNTING BASED PAGE REPLACEMENT

✓we can keep a counter of the number of references that have been made to each page and develop the following two schemes.

1. Least Frequently Used (LFU) page replacement Algorithm

- requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.

2. Most frequently Used (MFU) page replacement Algorithm

-based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

QUESTIONS!!

✓Consider the following page reference string: 1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2 3 6. How many page faults would occur for the FIFO, Optimal, LRU and LFU replacement algorithms having frames. Remember all frames are initially empty, so your first unique page will cost one fault each.

✓Calculate hits and faults using various page replacement algorithm policies (FIFO, LRU, Optimal) for the following page sequence. (The frame size is 3)

2 3 5 4 2 5 7 3 8 7

SECOND CHANCE PAGE REPLACEMENT ALGORITHM

- ✓ uses reference bit (R)
- ✓ R is associated with each page and initially 0
- ✓ When page is referenced, bit set to 1
- ✓ If R is 0, the page is both unused and old, so it is replaced immediately.
- ✓ If R is 1, the bit is cleared and the page is put onto the end of the list of pages and its load time is updated as though it had just arrived in memory. Then the search continues
- ✓ The goal is to find an old page that has not been referenced for a long time

SECOND CHANCE PAGE REPLACEMENT ALGORITHM

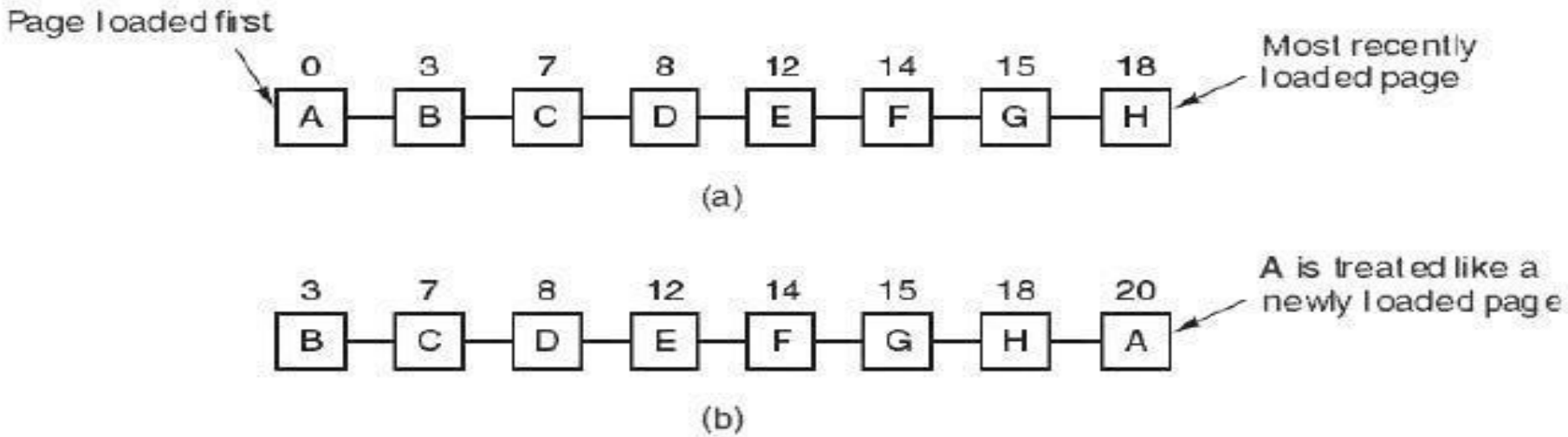


Figure 1. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and A has its R bit set. The numbers above the pages are their load times.

CLOCK PAGE REPLACEMENT ALGORITHM

- second chance algorithm is inefficient as it constantly moves pages around on its list
- A better approach is to keep all the pages on a circular list in the form of clock
- The hand points to the oldest page

-When new page is inserted, R is set to 1

- When a page fault occurs, the page being pointed at is inspected

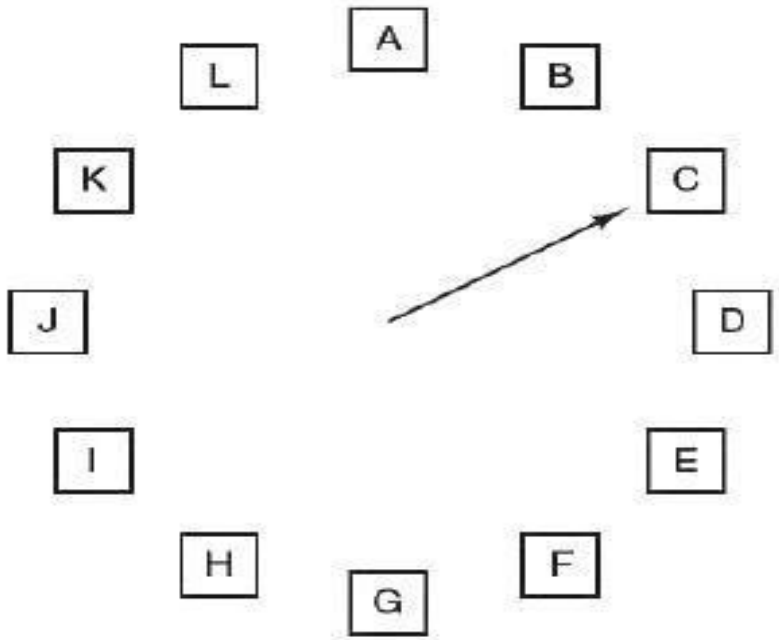
-If its R bit is 0, the page is evicted, the new page is inserted into the clock in its place R is set to 1and the hand is advanced one position

-If R bit is 1, it is set to 0 and the hand is advanced to the next page.

- The process is repeated until a page is found with R=0 - Exercise: 3 3 5 4 7 1 5 5 1 4 3 7 6 3 4 1

: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

CLOCK PAGE REPLACEMENT ALGORITHM



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

Figure 2. The clock page replacement algorithm.

Behavior of Page Replacement Algorithms

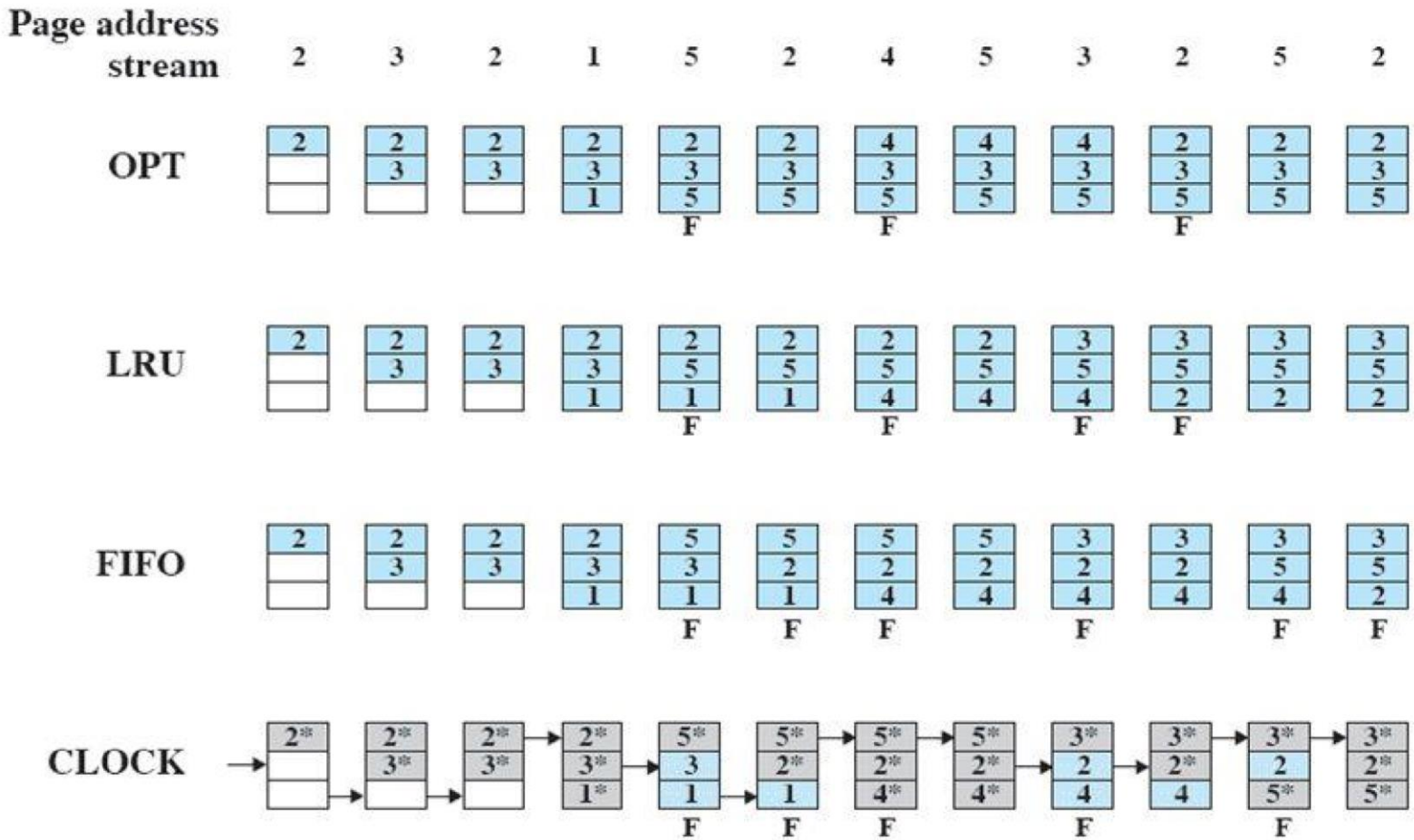


Figure 8.15 Behavior of Four Page-Replacement Algorithms

SEGMENTATION

- ✓Program is divided into parts known as segments
- ✓These segments may vary in size. But there exist a maximum limit to the size of segment
- ✓In other words, segmentation is a technique to break memory into logical pieces where each piece (segment) represents a group of related information.
- ✓Like paging, a logical address using segmentation consists of two parts:

1. Segment number
2. Offset

SEGMENTATION

- ✓Because of the use of unequal size segments, segmentation is similar to dynamic partitioning
- ✓The difference is that the segments of a same program need not be contiguous
- ✓Segmentation eliminates internal fragmentation but like dynamic partitioning, it suffers from external fragmentation
- ✓Each segment consists of a linear sequence of addresses, from 0 to some maximum
- ✓Segment length may change during execution. For example, the length of stack segment may be increased whenever something is pushed into the stack

ADVANTAGE OF SEGMENTATION

1. It simplifies the handling of growing data structure (OS can grow or shrink the segments as needed)
2. It allows programs to be altered and recompiled independently, without requiring the entire set of programs to be relinked and reloaded
3. It lends itself to sharing among a process. A programmer can place a utility program or a useful table of data in a segment that can be referenced by other processes
4. It lends itself to protection. Because a segment can be constructed to contain a well defined set of programs or data, the programmer/system administrator can assign access privileges in a convenient fashion.

SEGMENTATION

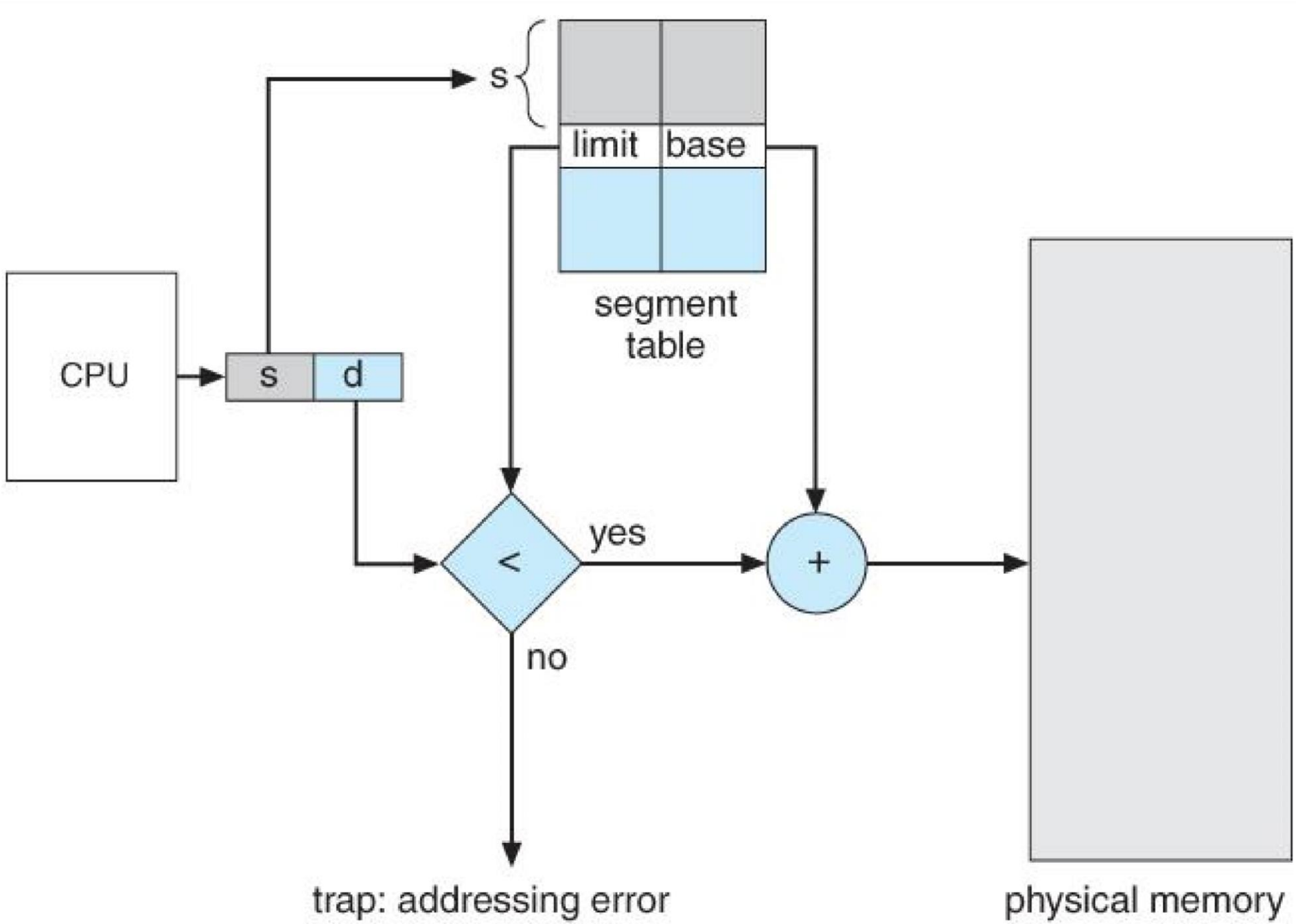
- ✓Due to unequal size segments, there is no simple relation between logical and physical address (which was pretty clear in paging)
- ✓Analogous to paging, we could create a segment table for each process
- ✓Each segment table entry would contain starting address in the main memory of the corresponding segment
- ✓Besides that, an entry would also provide the length of the segment so that invalid addresses are not used

ADDRESS TRANSLATION

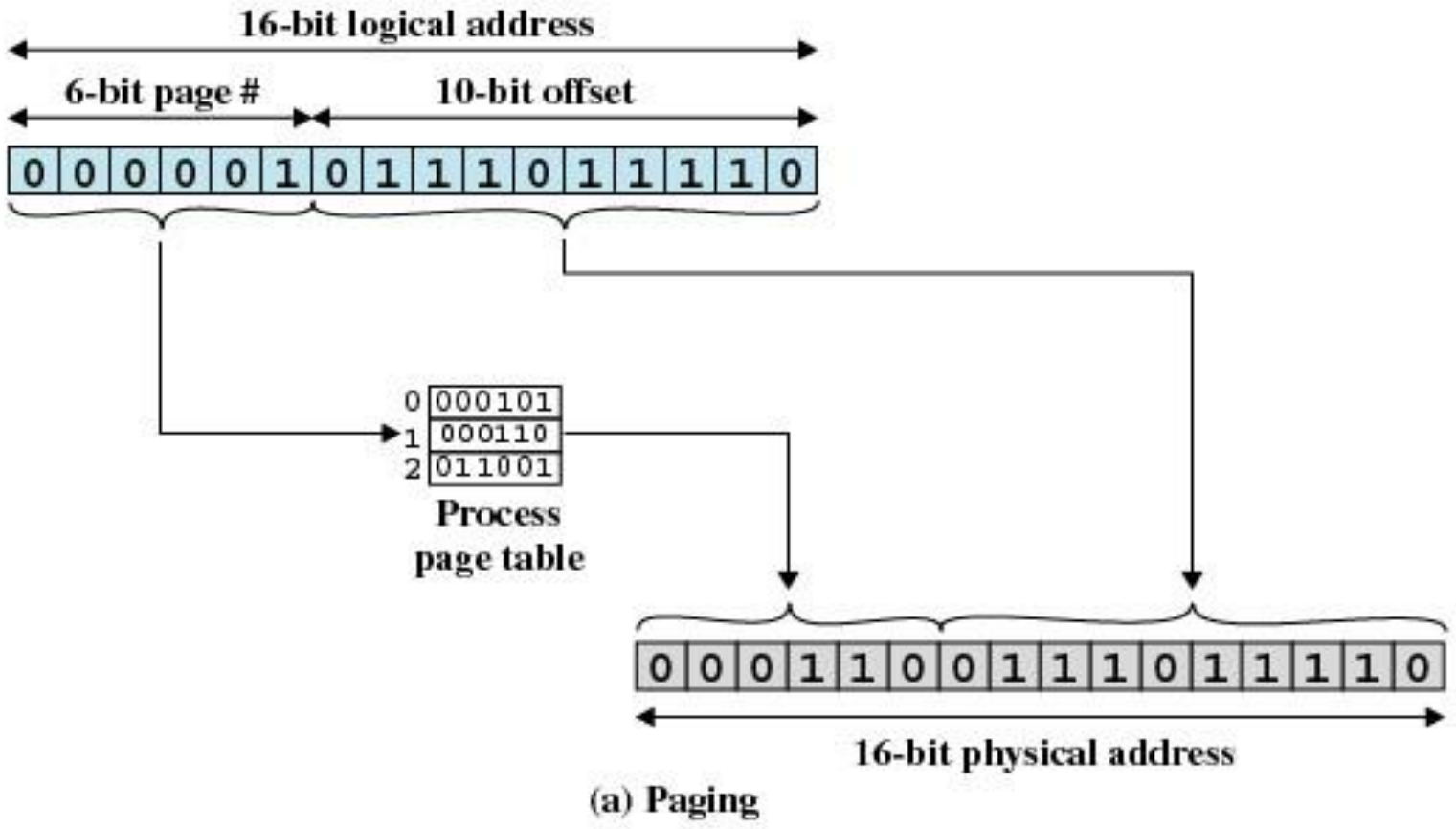
- ✓The logical address is n+m bits where the leftmost n bits are the segment number and rightmost m bits are the offset.

Translation:

- ✓Extract segment number
- ✓Use the segment number as index into the process segment table to find the starting physical address of the segment
- ✓Compare the offset expressed in the rightmost m bits to the length of segment. If the offset is greater than or equal to the length, the offset is invalid
- ✓The desired physical address is the sum of starting physical address of the segment plus the offset


s
limit
base
segment table
CPU
s
d
<
yes
+
no
trap: addressing error
physical memory

WHAT HAPPENED IN PAGING?



(a) Paging

WHAT HAPPENS IN SEGMENTATION?

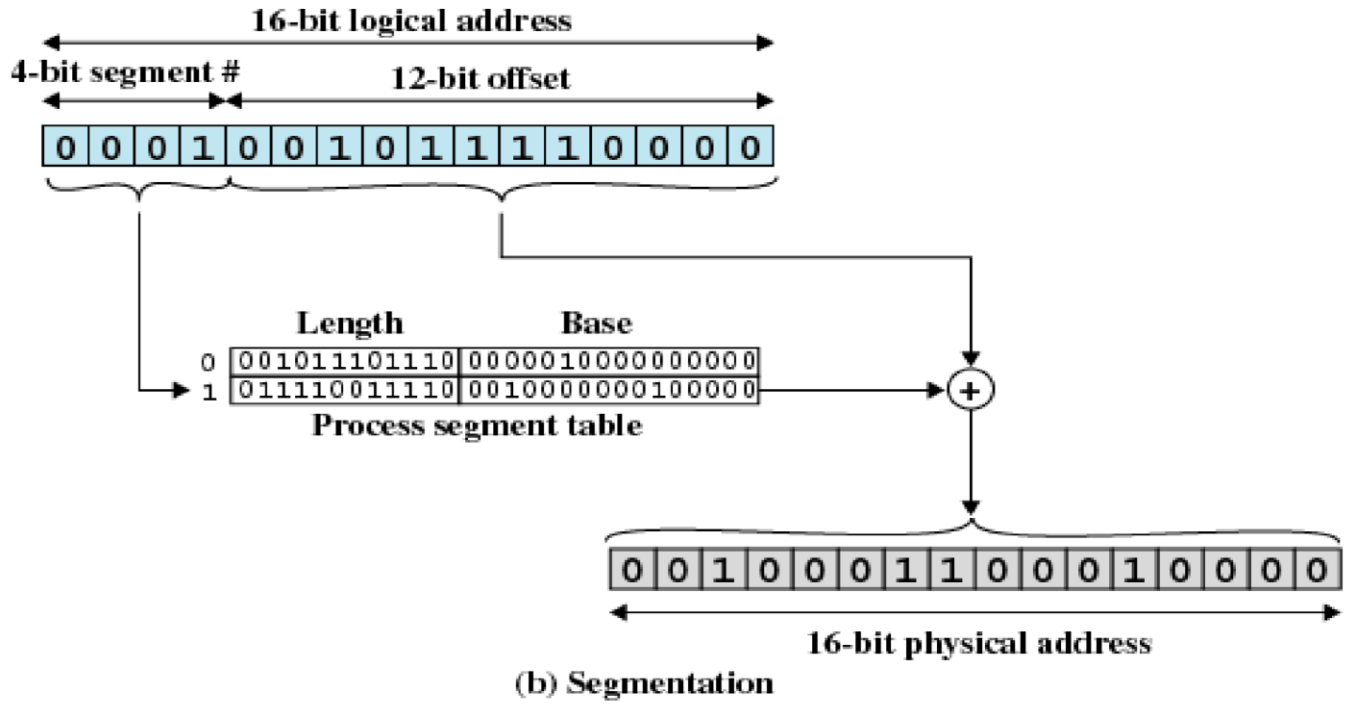


Figure 7.12 Examples of Logical-to-Physical Address Translation

COMBINING PAGING AND SEGMENTATION (PAGED SEGMENTATION)

- ✓In a combined paging and segmentation system, a user's address space is broken down into a number of segments
- ✓Each segment is then divided into fixed sized pages which are equal in length to a main memory frame
- ✓Associated with each process is a segment table and a number of page tables, one per process segment
- ✓*When a process is running, a registers holds the starting address of the segment table for that process*
- ✓The virtual address is broken down into segment number, page number and offset.
- ✓Using the segment number, and the segment table, we find the page table for that segment
- ✓Then the page number portion of the virtual address is used to look up the corresponding frame number in the page table
- ✓The frame number combined with the offset forms the physical address

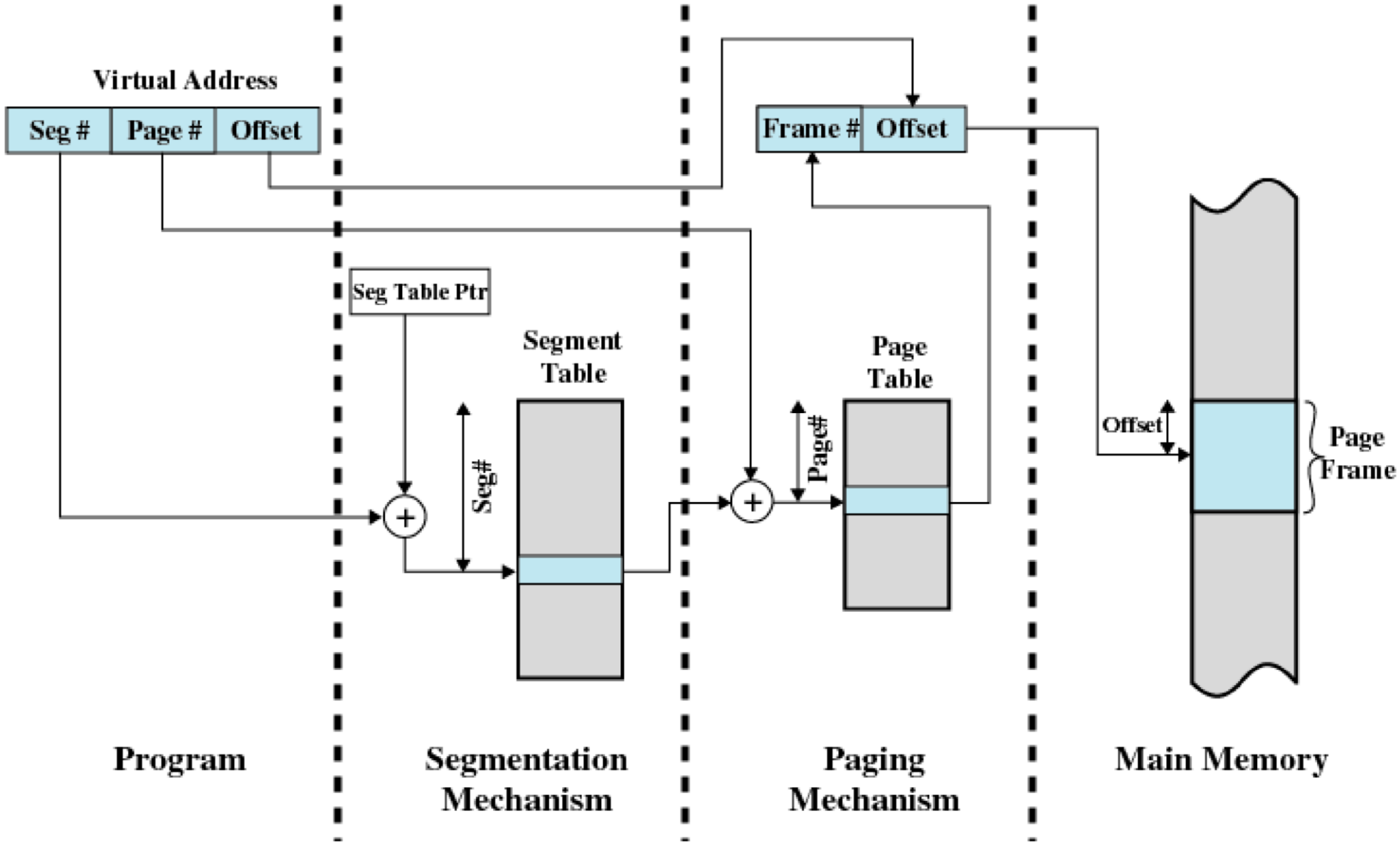


Figure 8.13 Address Translation in a Segmentation/Paging System

FRAME ALLOCATION

✓Frame allocation determines how many frames to give each process ✓Each process needs some minimum number of frames to operate correctly ✓Two major allocation schemes:

1. Fixed Allocation
2. Priority Allocation

FIXED ALLOCATION

Equal Allocation : For examples, if there are 100 frames (after allocating frames for OS), and 5 processes, give each process 20 frames

Proportional Allocation : Allocate according to size of process

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

a_i = allocation for $p_i = \frac{s_i}{S} \times m$

m= 64 frames

s1 = 10 (size of P1)

s2 = 127 (size of P2)

a1 = 10/137 * 64 = 5 frames for p1

a2 = 127/137 * 64 = 59 frames for p2

PRIORITY ALLOCATION

- ✓Uses proportional allocation scheme using priorities rather than size
- ✓Higher priority process gets more frames than lower priority process

THRASING

✓If a process does not have “enough” pages, the page fault rate is very high

- Page fault to get page

- Replace existing frame

-But quickly need replaced frame back - This leads to :

-low CPU utilization

-OS thnks that it needs to increase the increase the degree of multi programming and adds another process to the system Thrasing is a condition where a process is busy swapping pages in and out Task: How can you deal with this??