Topics to be covered: Memory Management

- Basic Memory Management: Definition
- Logical and Physical address map
- Memory allocation
- Paging
- Virtual Memory: Basics of Virtual Memory
- Hardware and control structures – Locality of reference
- Page fault
- Working Set
- Dirty page/Dirty bit – Demand paging (Concepts only)
- Page Replacement Algorithms

What is Memory?

- Computer memory is any **physical device capable of storing information** temporarily or permanently.

- Types of memory

  1. **Random Access Memory (RAM)**, is a **volatile memory** that loses its contents when the computer or hardware device loses power.

  2. **Read Only Memory (ROM)**, is a **non-volatile memory**, sometimes abbreviated as NVRAM, is a memory that keeps its contents even if the power is lost.

     - Computer uses special ROM called **BIOS (Basic Input Output System)** which permanently stores the software needed to access computer hardware such as hard disk and then load an operating system into RAM and start to execute it.

What is Memory? (cont…)

3. **Programmable Read-Only Memory (PROM)**, is a memory chip on which you can store a program. But once the PROM has been used, you **cannot wipe it clean** and use it to store something else. Like ROMs, PROMs are non-volatile. E.g CD-R

4. **Erasable Programmable Read-Only Memory (EPROM)**, is a special type of PROM that **can be erased** by **exposing it to ultraviolet light**. E.g CD-RW

5. **Electrically Erasable Programmable Read-Only Memory (EEPROM)**, is a special type of PROM that **can be erased** by **exposing it to an electrical charge**. E.g Pendrive

What is Memory Hierarchy?

- The hierarchical arrangement of storage in current computer architectures is called the memory hierarchy.
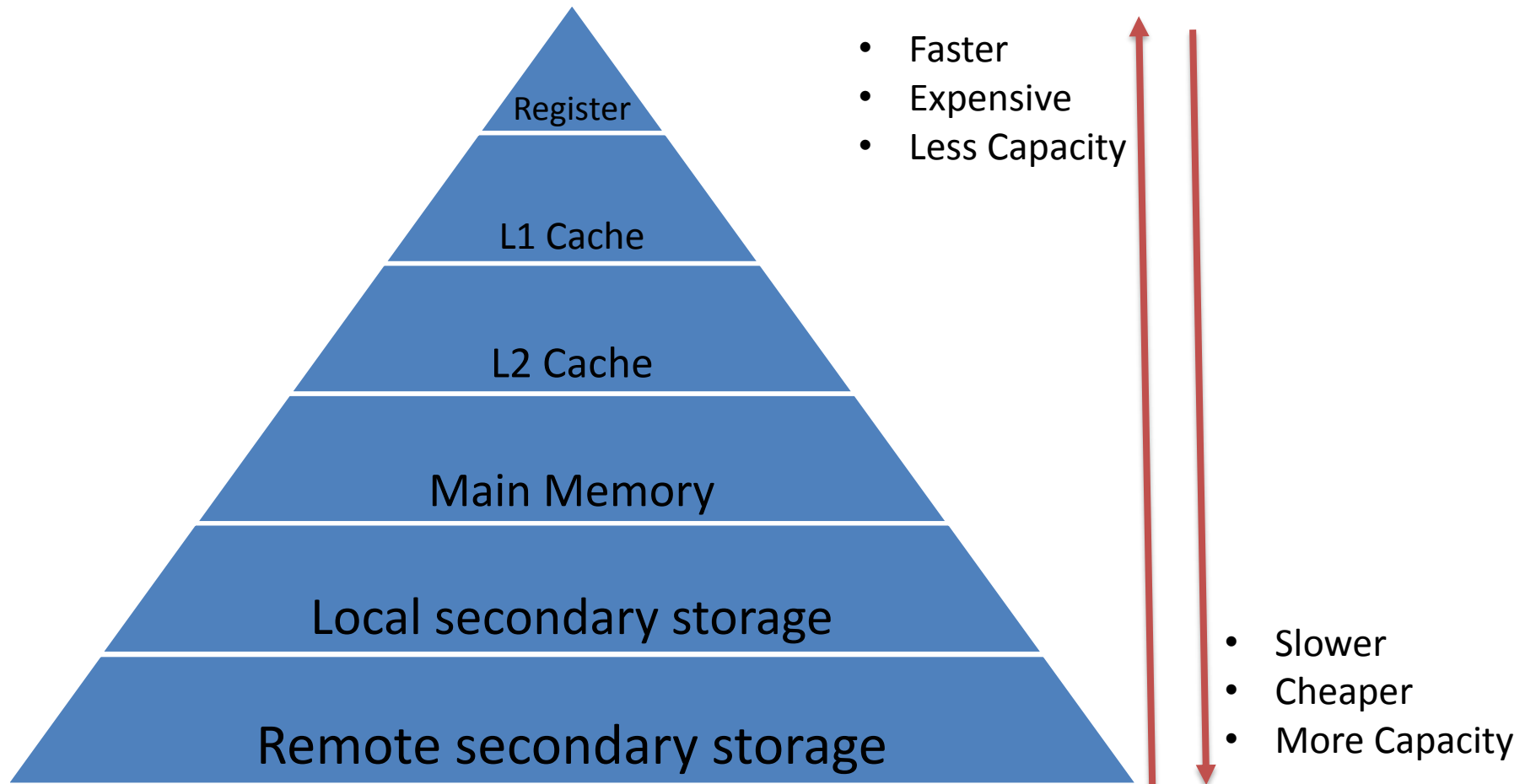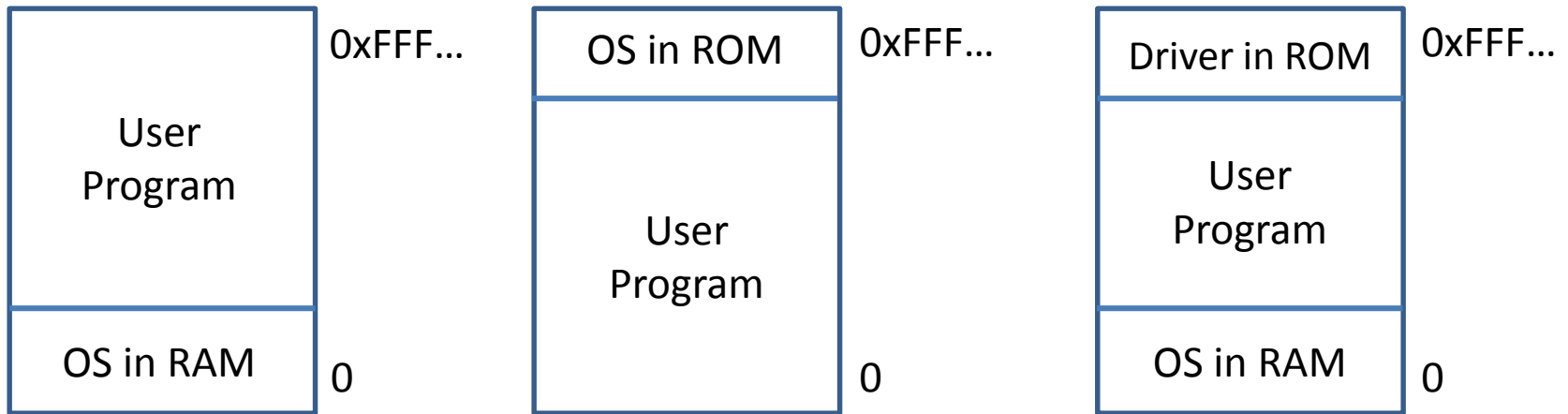
Memory abstraction

- The hardware and OS memory manager makes you see the memory as a single contiguous entity.

- How do they do that?
  - Abstraction

- Is abstraction necessary?

No memory abstraction

- In this model the **memory presented** to the programmer was simply **a single block** of physical memory
  - having a set of addresses from 0 to some maximum
  - with each address corresponding to a cell containing some number of bits, commonly eight.

User Program	0xFFF...
OS in RAM	0

OS in ROM	0xFFF...
User Program	0

Driver in ROM	0xFFF...
User Program	
OS in RAM	0

Even with no abstraction, we can have several setups!

No memory abstraction

- When program execute instruction like
  - MOV *REGISTER1, 1000*
- If at the **same time another program execute same instruction** then value of **first program will be overwrite**.
- So **only one process at a time can be running**.

User Program	0xFFF...
OS in RAM	0

OS in ROM	0xFFF...
User Program	0

Driver in ROM	0xFFF...
User Program	
OS in RAM	0
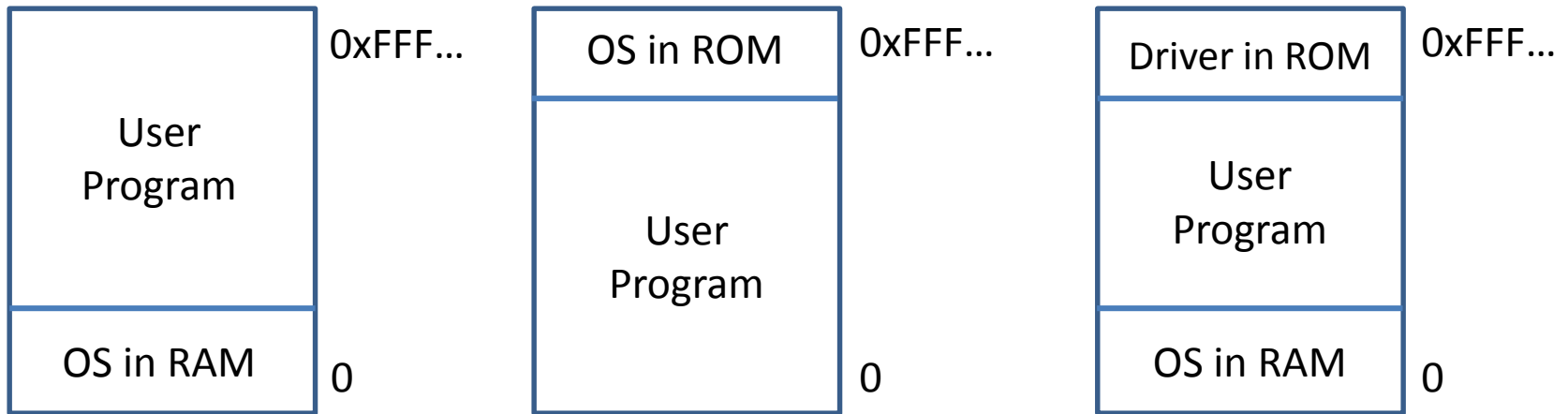
Even with no abstraction, we can have several setups!

No memory abstraction

- What if we want to run multiple programs?
  - OS saves entire memory on disk
  - OS brings next program
  - OS runs next program
- We can **use swapping** to **run multiple programs concurrently**.
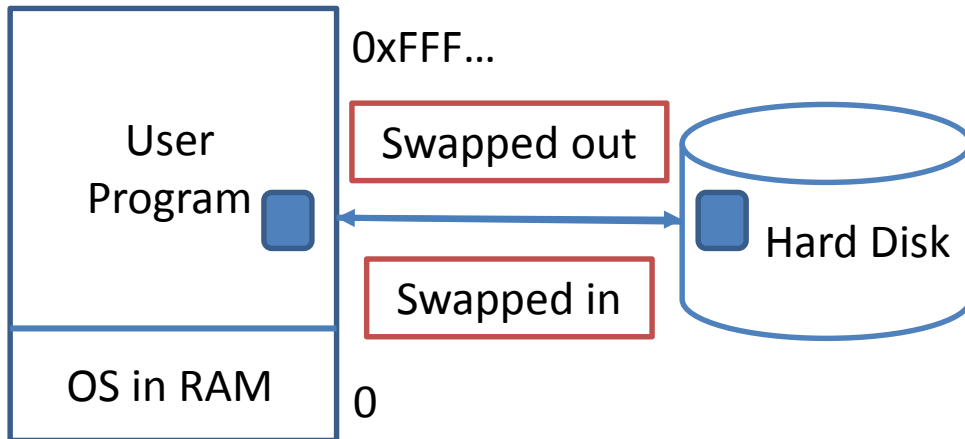

- The process of **bringing in each process** in its entirely **in to memory**, **running** it for a while and then **putting it back** on the **disk** is called swapping.

Ways to implement swapping system

- Two different ways to implement swapping system
    1. Multiprogramming with **fixed partitions**
    2. Multiprogramming with **dynamic partitions**

Multiprogramming with Fixed partitions

- Here memory is **divided into fixed sized partitions**.

- Size can be **equal** or **unequal** for different partitions.

- Generally **unequal partitions** are **used** for **better utilizations**.

- Each partition can accommodate exactly one process, means only single process can be placed in one partition.

- The **partition boundaries** are **not movable**.

Partition 4
Partition 3
Partition 2
Partition 1
OS
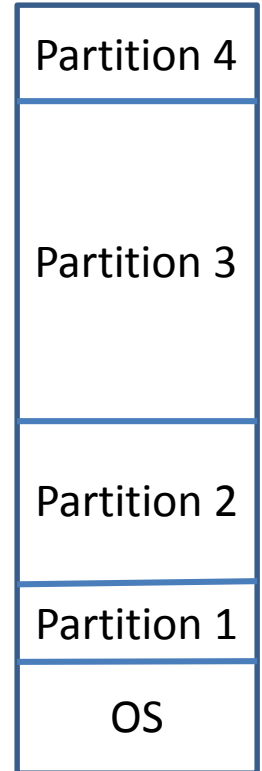
Multiprogramming with Fixed partitions

- Whenever any program needs to be loaded in memory, **a free partition big enough to hold the program is found**. This partition will be allocated to that program or process.

- If there is **no free partition available of required size**, then the process **needs to wait**. Such process will be put in a queue.

Partition 4
Partition 3
Partition 2
Partition 1
OS
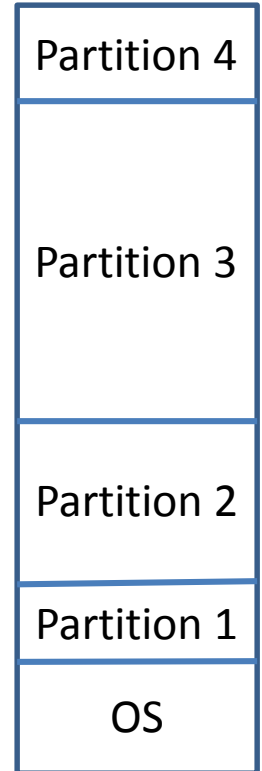
Multiprogramming with Fixed partitions

- There are two ways to maintain queue
  1. Using **Multiple Input Queues**.
  2. Using **Single Input Queue**.

Multiple Input Queues

| Partition - 4 |
| Partition - 3 |
| Partition - 2 |
| Partition - 1 |
| Operating System |

Single Input Queue

| Partition - 4 |
| Partition - 3 |
| Partition - 2 |
| Partition - 1 |
| Operating System |
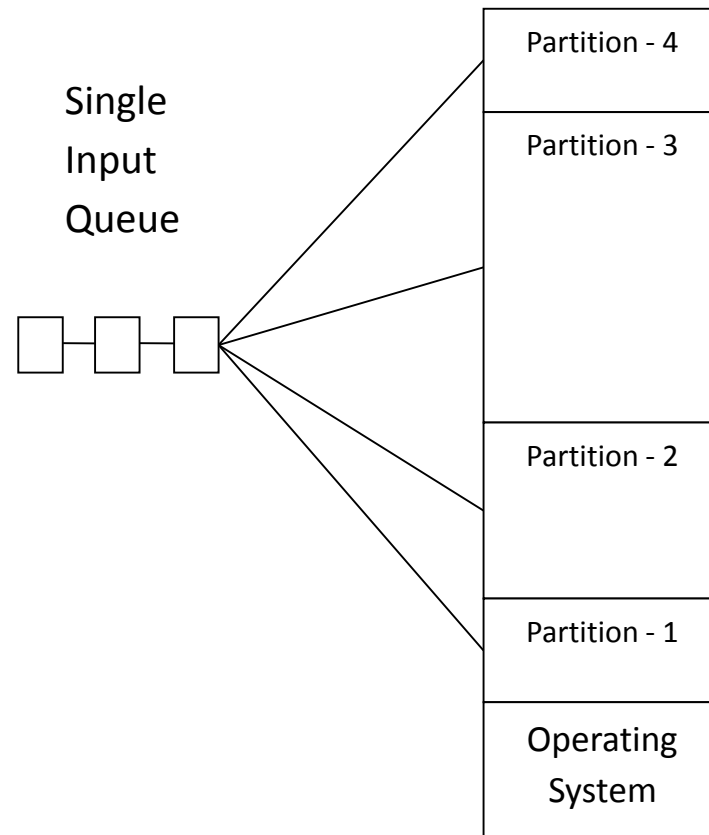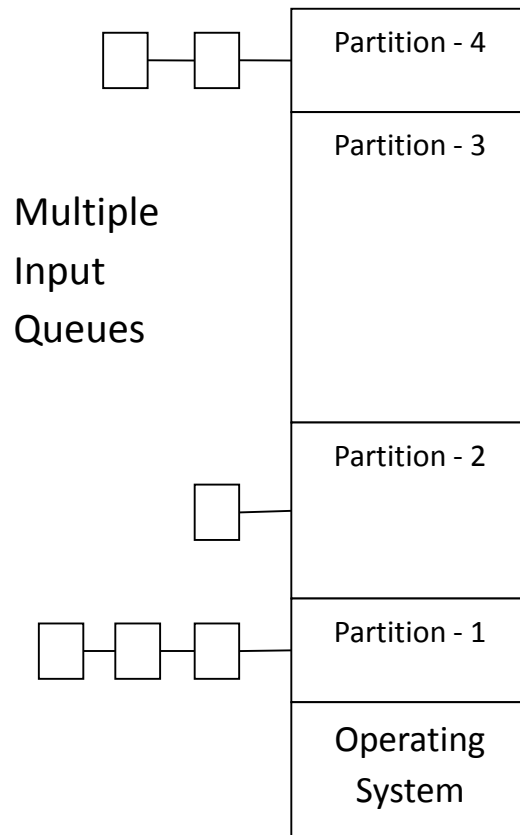
Multiprogramming with Dynamic partitions

- Here, memory is **shared among operating system** and various **simultaneously running processes**.

- Initially, the **entire available memory is treated as a single free partition**.

- Process is **allocated exactly as much memory as it requires**.

- Whenever any **process enters** in a system, a **chunk of free memory big enough to fit the process is found and allocated**. The **remaining unoccupied space is treated as another free partition**.

User Program

OS

Multiprogramming with Dynamic partitions

- If **enough free memory is not available** to fit the process, **process needs to wait** until required memory becomes available.

- Whenever any **process gets terminate**, it **releases the space occupied**. If the released free space is contiguous to another free partition, both the free partitions are merged together in to single free partition.

- **Better utilization** of memory than fixed sized size partition.

User
Program

OS

Multiprogramming with Dynamic partitions

Time →

(a)   (b)   (c)   (d)   (e)   (f)   (g)
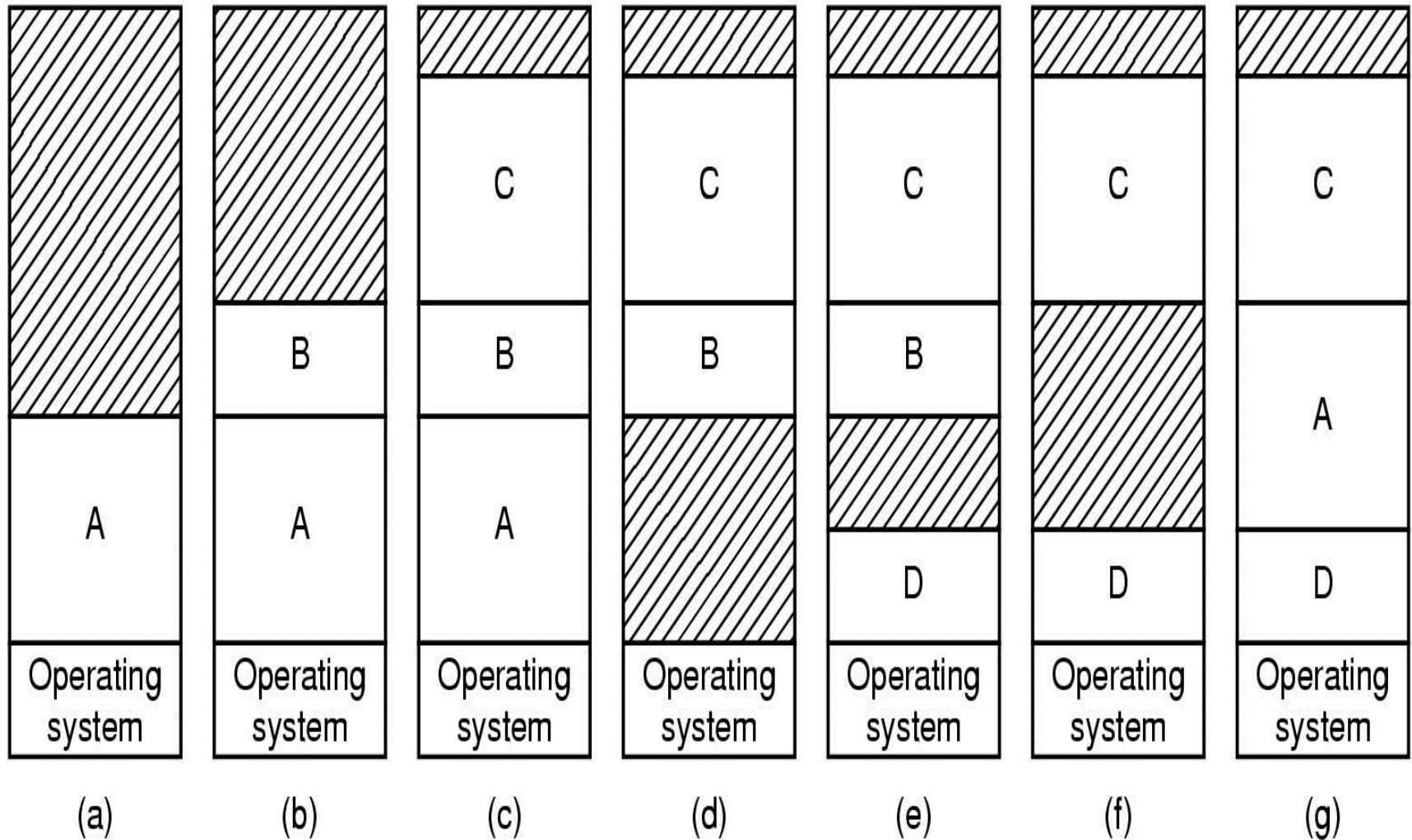
Memory compaction

- **When swapping creates multiple holes** in memory, it is possible to **combine them** all **in one big hole** by **moving all the processes downward** as far as possible. This techniques is known as **memory compaction**.

- It **requires lot of CPU time**.
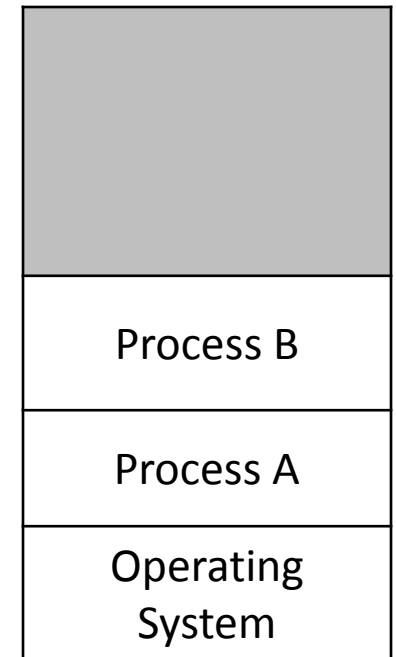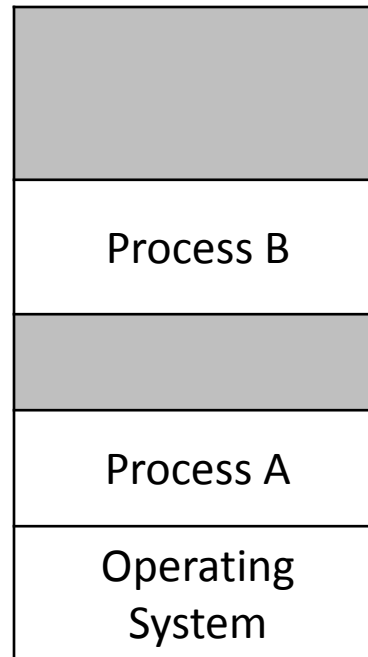
Multiprogramming without memory abstraction

Static relocation

When program was loaded at address 16384, the constant 16384 was added to every program address during the load process.

- Slow
- Required extra information from program

Program 1		Program 2			
				0	32764
				.	
				.	
				CMP	16412
					16408
					16404
					16400
					16392
					16388
				JMP 16412	16384
0	16380	0	16380	0	16380
.		.		.	
.		.		.	
ADD	28	CMP	28	ADD	28
MOV	24		24	MOV	24
	20		20		20
	16		16		16
	8		8		8
	4		4		4
JMP 24	0	JMP 28	0	JMP 24	0

Base and Limit register

- **An address space** is **set of addresses that a process can use** to address memory.
- **An address space** is a **range of valid addresses in memory that are available** for a program or process.
- Two registers: Base and Limit
  1. **Base register**: **Start address** of a program in physical memory.
  2. **Limit register**: **Length** of the program.
- For every memory access
  - **Base is added** to the address
  - **Result compared** to Limit
- Only OS can modify Base and Limit register.

0	32764

CMP	16412
	16408
	16404
	16400
	16392
	16388
JMP 16412	16384
0	16380

ADD	28
MOV	24
	20
	16
	8
	4
JMP 24	0

Dynamic relocation

- Steps in dynamic relocation
  1. Hardware **adds relocation register (base) to virtual address to get a physical address**
  2. Hardware **compares address with limit register**; address **must be less than or equal limit**
  3. If test fails, the processor takes an address trap and ignores the physical address.

```
        ┌──────────┐                    ┌──────────┐
        │   Base   │                    │  Limit   │
        │ Register │                    │ Register │
        └────┬─────┘                    └────┬─────┘
             │                               │
             ▼                               ▼
┌─────┐   ┌──────────┐   ┌─┐   ┌──────────┐   ┌────┐   Yes   ┌────────┐
│ CPU │──▶│ Logical  │──▶│+│──▶│ Physical │──▶│ <= │────────▶│ MEMORY │
│     │   │ Address  │   └─┘   │ Address  │   └─┬──┘         └────────┘
└─────┘   └──────────┘        └──────────┘     │
                                               │ No
                                               ▼
                                    Trap: Address error
```
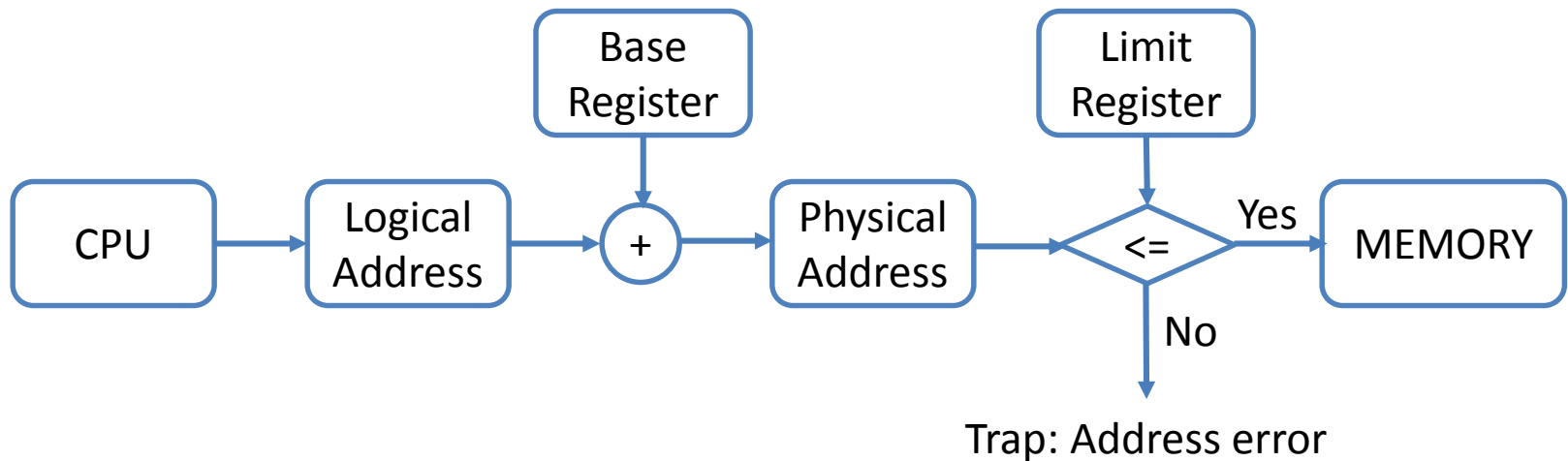
Managing free memory

- Two ways to keep track of memory usage (free memory)
    1. Memory management with **Bitmaps**
    2. Memory management with **Linked List**

Memory management with Bitmaps
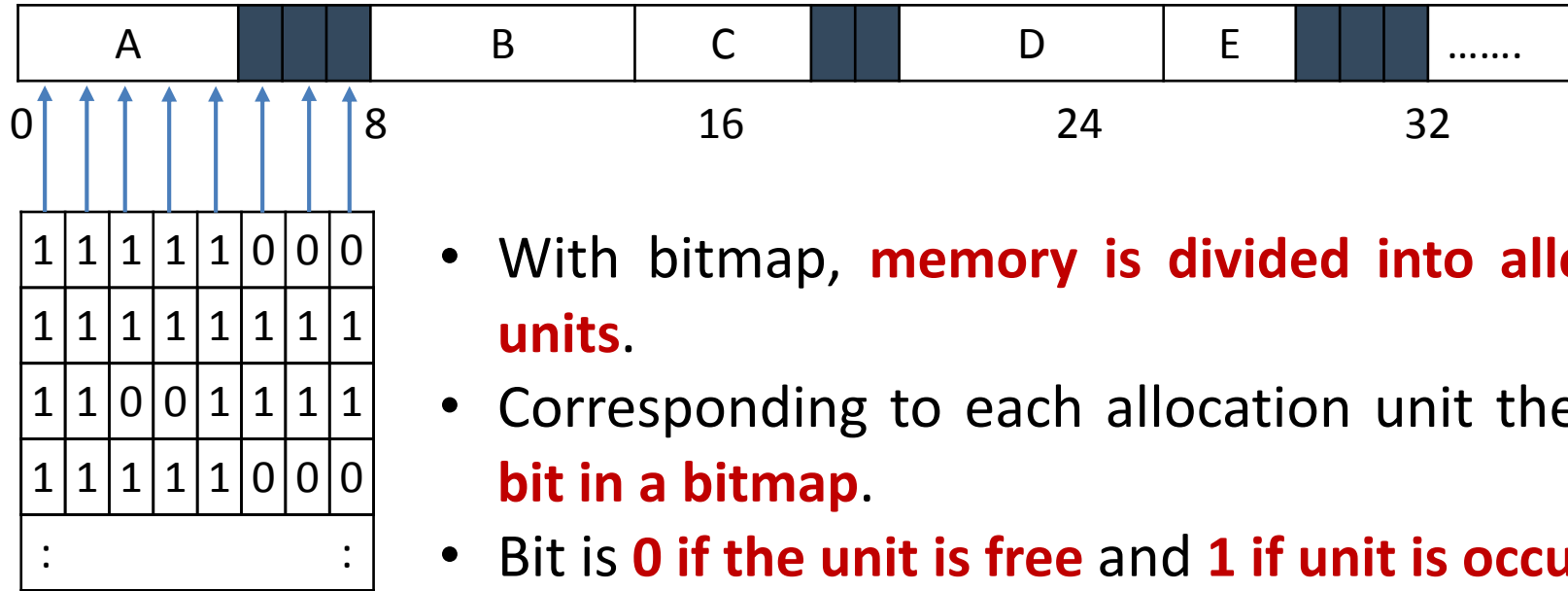
- With bitmap, **memory is divided into allocation units**.
- Corresponding to each allocation unit there is a **bit in a bitmap**.
- Bit is **0 if the unit is free** and **1 if unit is occupied**.
- The **size of allocation unit** is an important design issue, the **smaller the size, the larger the bitmap**.

Memory management with Bitmaps

A							

B	C	D	E	....

0   8   16   24   32

1	1	1	1	1	0	0	0
1	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	0	0	0
:							:
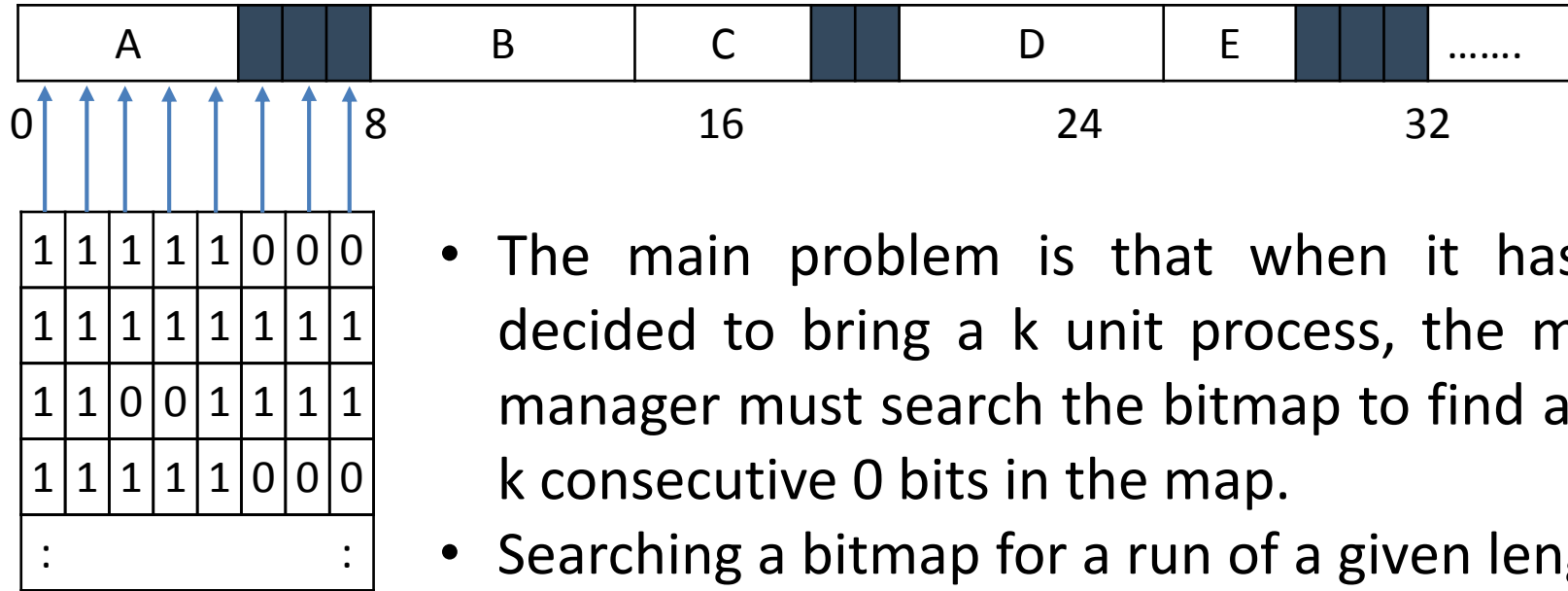
- The main problem is that when it has been decided to bring a k unit process, the memory manager must search the bitmap to find a run of k consecutive 0 bits in the map.
- Searching a bitmap for a run of a given length is a **slow operation**.

Memory management with Linked List

| P | 0 | 5 | → | H | 5 | 3 | → | P | 8 | 6 | → | P | 14 | 4 |

| H | 18 | 2 | → | P | 20 | 6 | → | P | 26 | 3 | → | H | 29 | 3 | X |

Hole   Starts   Length   Process
       at 18   2
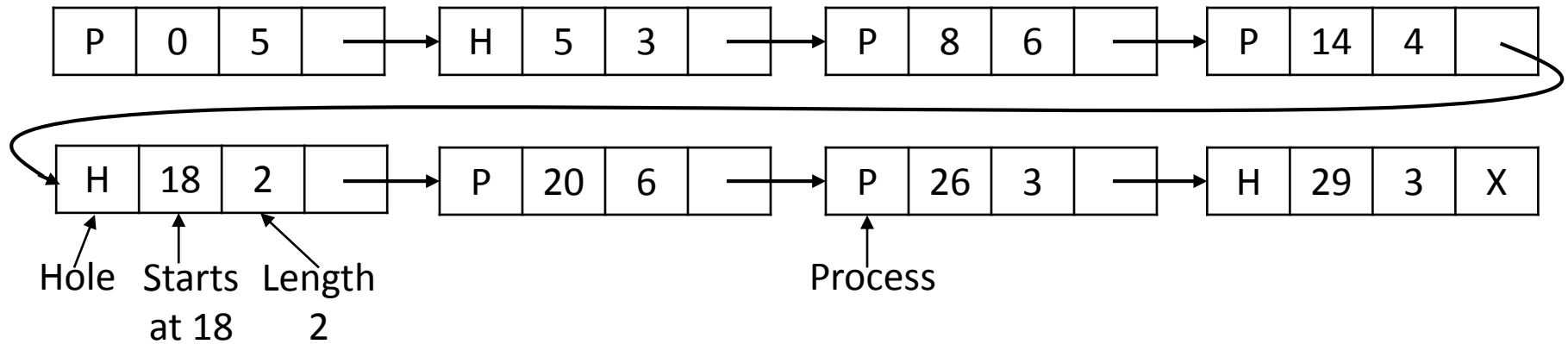
- Another way to keep track of memory is to **maintain a linked list of allocated and free memory segments**, where segment either contains a process or is an empty hole between two processes.
- Each entry in the list specifies a **hole (H)** or **process (P)**, the **address at which it starts** the **length** and a **pointer** to the next entry.
- The segment list is kept **sorted by address**.

Memory management with Linked List

| P | 0 | 5 | → | H | 5 | 3 | → | P | 8 | 6 | → | P | 14 | 4 |

| H | 18 | 2 | → | P | 20 | 6 | → | P | 26 | 3 | → | H | 29 | 3 | X |

Hole   Starts   Length
       at 18   2
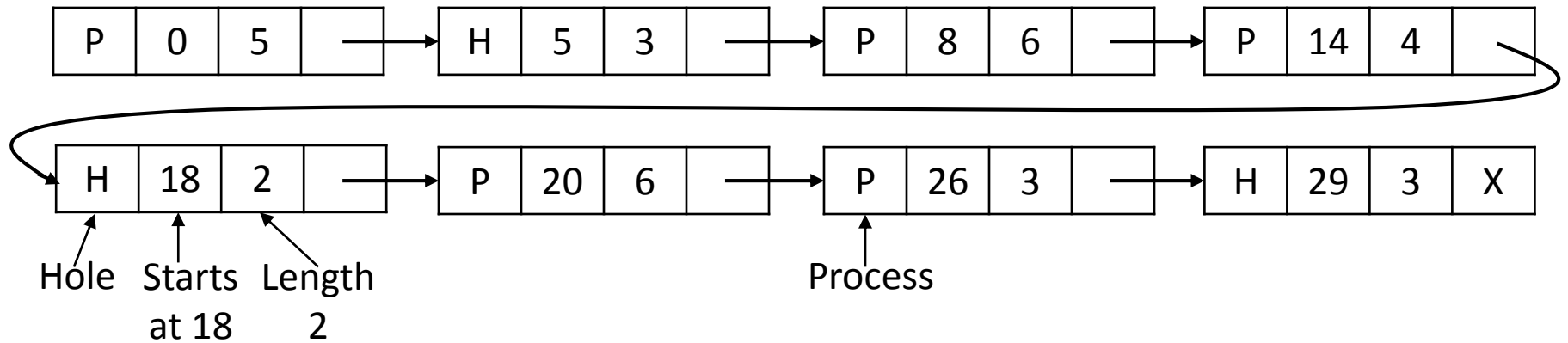
Process

- **Sorting this way has the advantage** that when a process terminates or is swapped out, **updating the list is straightforward**.
- **A terminating process normally has two neighbors** (except when it is at the very top or bottom of memory).

Memory management with Linked List

| P | 0 | 5 | → | H | 5 | 3 | → | P | 8 | 6 | → | P | 14 | 4 | |

| H | 18 | 2 | → | P | 20 | 6 | → | P | 26 | 3 | → | H | 29 | 3 | X |
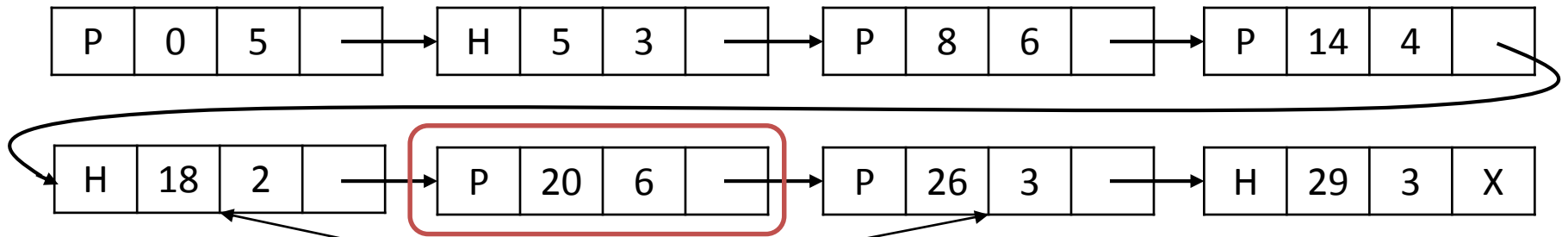
Neighbors

Before P terminate

| A | P | B |
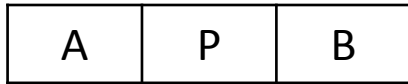
P is replaced by H

After P terminate
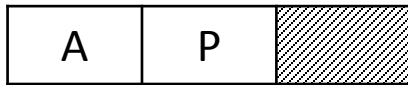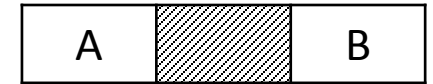
| A | ░░░ | B |

| A | P | ░░░ |

P is replaced by H
and two H are merged

| A | ░░░░ |

| ░░░ | P | B |

P is replaced by H
and two H are merged

| ░░░░ | B |

| ░░░ | P | ░░░ |
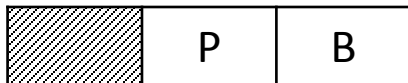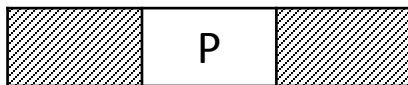
P is replaced by H
and three H are merged

| ░░░░░░ |

Memory allocation algorithms

- Four memory allocation algorithms are as follow
  1. First fit
  2. Next fit
  3. Best fit
  4. Worst fit

First fit

- Search **starts from the starting location** of the memory.

- **First available hole** that is large enough to hold the process is selected for allocation.

- The hole is then **broken up into two pieces**, **one for process** and **another for unused memory**.

- Example: Processes of *212K, 417K, 112K and 426K* arrives in order.

100k	500k	200k	300k	600k

100k	500k 288k 176k	200k	300k	600k 183k
	212k 112k			417k

- Here process of size **426k will not get any partition** for allocation.

First fit

- **Fastest algorithm** because it **searches as little as possible**.
- **Memory loss is higher**, as very large hole may be selected for small process.
- Here process of size 426k will not get any partition for allocation.

100k	500k	200k	300k	600k

100k		176k	200k	300k		183k
	212k	112k			417k	

Next fit

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

- The **next time when it is called to find a hole, it starts searching the list from the place where it left off last time**.

- Processes of *212K, 417K, 112K and 426K* arrives in order.

100k	500k	200k	300k	600k

100k	288k	200k	300k		71k
	212k			417k	112k
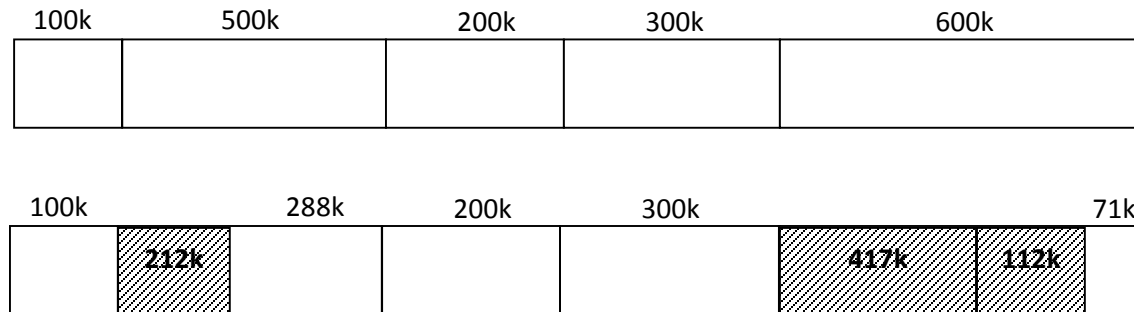
- Here process of size **426k will not get any partition** for allocation.

Next fit

- **Search time is smaller**.
- Memory manager must have to **keep track of last allotted hole** to process.
- It gives **slightly worse performance** than first fit.
- Here process of size 426k will not get any partition for allocation.

100k	500k	200k	300k	600k

100k	212k	288k	200k	300k	417k	112k	71k
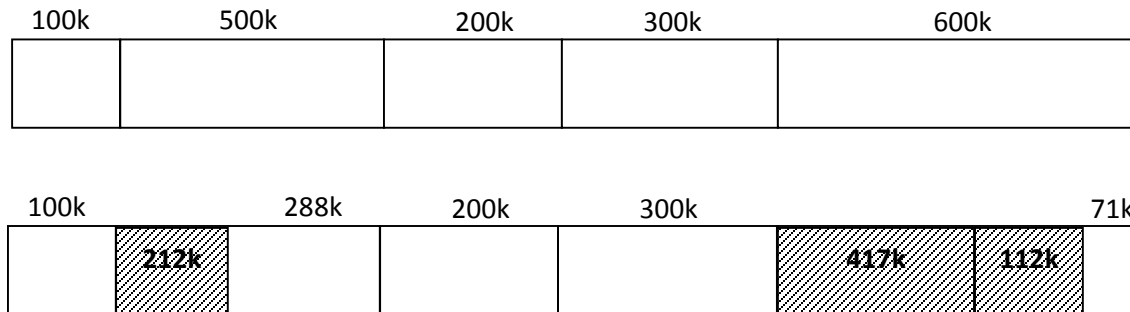
Best fit

- **Entire memory is searched** here.
- The **smallest hole**, which is **large enough to hold the process**, is **selected for allocation**.
- Processes of *212K, 417K, 112K and 426K* arrives in order.

100k	500k	200k	300k	600k

| 100k | 417k | 83k | 112k | 88k | 212k | 88k | 426k | 174k |
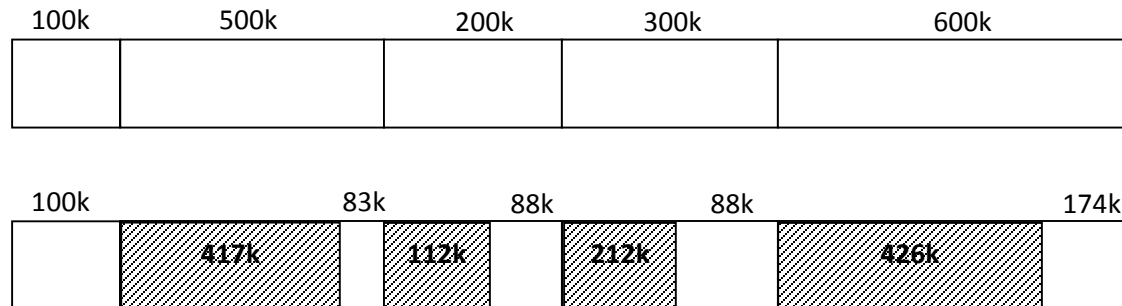
Best fit

- **Search time is high**, as it searches entire memory every time.
- **Memory loss is less**.

100k	500k	200k	300k	600k

100k		83k		88k		88k		174k
	417k		112k		212k		426k	
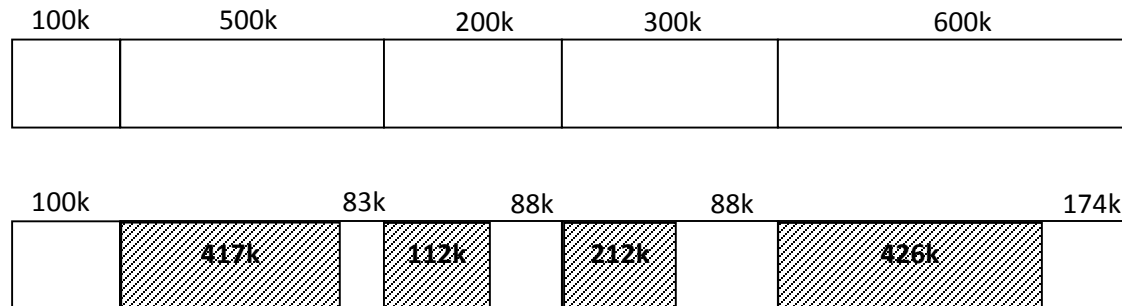
Worst fit

- **Entire memory is searched** here also. The **largest hole**, which is **largest enough to hold the process**, is **selected for allocation**.

- Processes of *212K, 417K, 112K and 426K* arrives in order.

100k	500k	200k	300k	600k

100k	83k	200k	300k	276k
	417k			212k 112k
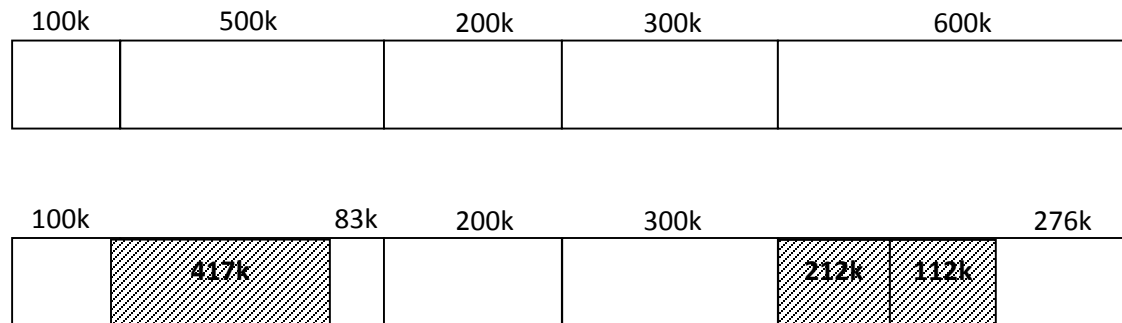
- Here process of size **426k will not get any partition** for allocation.

Worst fit

- **Search time is high**, as it searches entire memory every time.
- This algorithm can be **used only with dynamic partitioning**.
- Here process of size **426k will not get any partition** for allocation.

100k	500k	200k	300k	600k

100k	417k	83k	200k	300k	212k	112k	276k
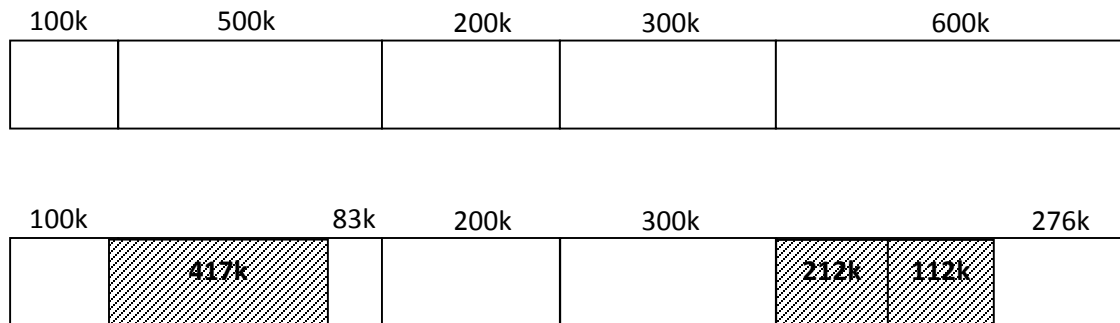
Virtual Memory

- Memory is hardware that your computer uses to **load the operating system and run programs**.

- Computer consists of **one or more RAM chips** that each have several memory modules.

- The **amount of real memory in a computer is limited** to the amount of RAM installed. Common memory sizes are 1GB, 2GB, and 4GB.

Virtual Memory

- Because your **computer has a finite amount of RAM**, it is possible to **run out of memory when too many programs are running** at one time.

- This is **where virtual memory comes in**.

- Virtual memory **increases the available memory of your computer by enlarging the "address space," or places in memory where data can be stored**.

- It does this **by using hard disk space** for additional memory allocation.

- However, since the **hard drive is much slower than the RAM**, data stored in **virtual memory must be mapped back to real memory** in order to be used.

Virtual Memory

- Each program has its own address space, which is broken up into pages.

- Each **page is a contiguous range of addresses**.

- These **pages are mapped onto the physical memory** but, to **run the program**, **all pages are not required to be present** in the physical memory.

- The operating system **keeps those parts of the program currently in use in main memory**, and the **rest on the disk**.

RAM

RAM

HDD

RAM

HDD

Virtual
Address space

Another
Process's
Memory

RAM

HDD
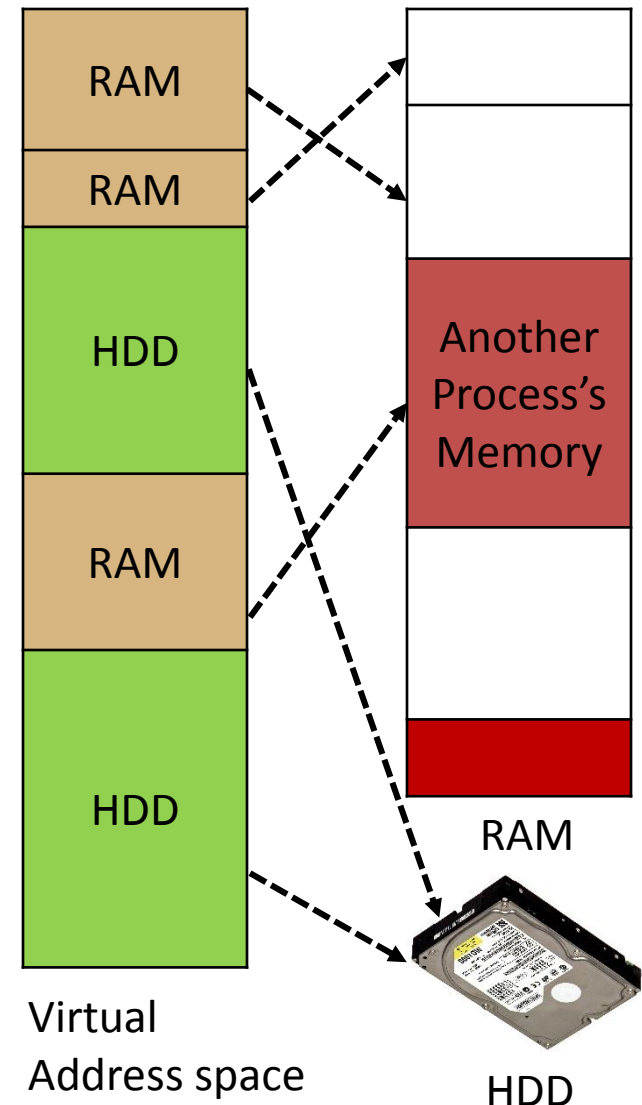
Virtual Memory

- In a system using virtual memory, the **physical memory is divided into page frames** and the **virtual address space is divided in into equally-sized partitions called pages**.

- Virtual memory works fine in a multiprogramming system, with bits and pieces of many programs in memory at once.

RAM

RAM

HDD

RAM

HDD

Virtual
Address space

Another
Process's
Memory

RAM

HDD
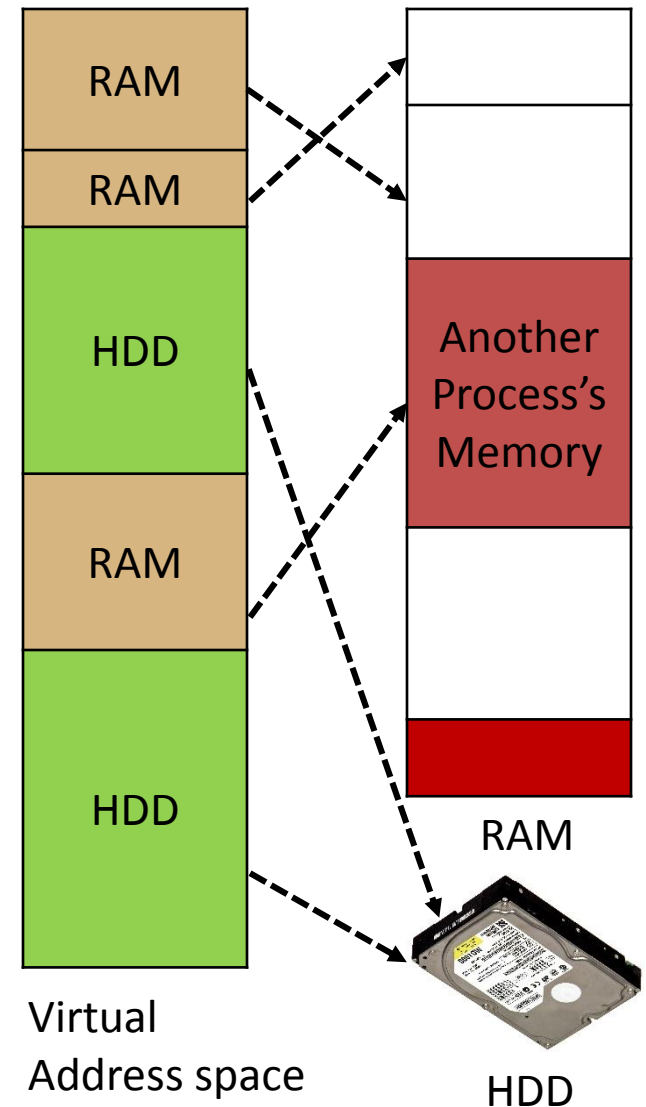
Paging

- Paging is a storage **mechanism used to retrieve processes from the secondary storage (Hard disk) into the main memory (RAM)** in the form of pages.

- The main idea behind the paging is to **divide each process in the form of pages**. The main memory will also be divided in the form of frames.

- One page of the process is to be stored in one of the frames of the memory.

- The pages can be stored at the different locations of the memory but the priority is always to find the contiguous frames or holes.

Paging

- **Pages of the process are brought into the main memory only when they are required** otherwise they reside in the secondary storage.

- The **sizes of each frame must be equal**. Considering the fact that the pages are mapped to the frames in Paging, **page size needs to be as same as frame size**.

- Different operating system defines different frame sizes.

Paging

40K – 44K	Page 11
36K – 40K	Page 10
32K – 36K	Page 9
28K – 32K	Page 8
24K – 28K	Page 7
20K – 24K	Page 6
16K – 20K	Page 5
12K – 16K	Page 4
8K – 12K	Page 3
4K – 8K	Page 2
0K – 4K	Page 1

Virtual Address Space

Virtual page

Page frame

Frame 6	20K – 24K
Frame 5	16K – 20K
Frame 4	12K – 16K
Frame 3	8K – 12K
Frame 2	4K – 8K
Frame 1	0K – 4K

Physical Memory Address

Paging

Process 4
Page 3
Page 2
Page 1

Process 1 and process 4 are moved out from memory and process 5 enters into memory.

Process 1
Page 3
Page 2
Page 1

Frame 12
Frame 11
Frame 10
Frame 9
Frame 8
Frame 7
Frame 6
Frame 5
Frame 4
Frame 3
Frame 2
Frame 1

Process 3
Page 3
Page 2
Page 1

Process 2
Page 3
Page 2
Page 1

Paging

Process 5
Page 6
Page 5
Page 4
Page 3
Page 2
Page 1

Memory
Frame 12
Frame 11
Frame 10
Frame 9
Frame 8
Frame 7
Frame 6
Frame 5
Frame 4
Frame 3
Frame 2
Frame 1

Process 3
Page 3
Page 2
Page 1

Process 2
Page 3
Page 2
Page 1

We have 6 non contiguous frames available in the memory and paging provides the flexibility of storing the process at the different places.

Paging

Virtual Address Space				Physical Memory Address	
40K – 44K					
36K – 40K		Virtual page			
32K – 36K	5				
28K – 32K					
24K – 28K					
20K – 24K	3	Page frame	5	20K – 24K	
16K – 20K	4		4	16K – 20K	
12K – 16K	0		3	12K – 16K	
8K – 12K			2	8K – 12K	
4K – 8K	1		1	4K – 8K	
0K – 4K	2		0	0K – 4K	

Paging

- Size of **Virtual Address Space is greater than that of main memory**, so **instead of loading entire address space** in to memory to run the process, **MMU copies only required pages** into main memory.

- In order **to keep the track of pages and page frames**, OS maintains a **data structure called page table**.

Logical Address vs Physical Address

- Logical Address is **generated by CPU while a program is running**. The logical address is virtual address as it **does not exist physically** therefore it is also known as Virtual Address.

- Physical Address **identifies a physical location of required data in a memory**. The **user never directly deals** with the physical address but can access by its corresponding logical address.

- The hardware device called **Memory-Management Unit is used for mapping (converting) logical address to its corresponding physical address**.

Conversion of virtual address to physical address

- When virtual memory is used, the **virtual address is presented to an MMU** (Memory Management Unit) that **maps the virtual addresses onto the physical memory addresses**.

40K – 44K	X
36K – 40K	X
32K – 36K	5
28K – 32K	X
24K – 28K	X
20K – 24K	3
16K – 20K	4
12K – 16K	0
8K – 12K	X
4K – 8K	1
0K – 4K	2

Virtual Address Space

Virtual page

Page frame

5	20K – 24K
4	16K – 20K
3	12K – 16K
2	8K – 12K
1	4K – 8K
0	0K – 4K

Physical Memory Address
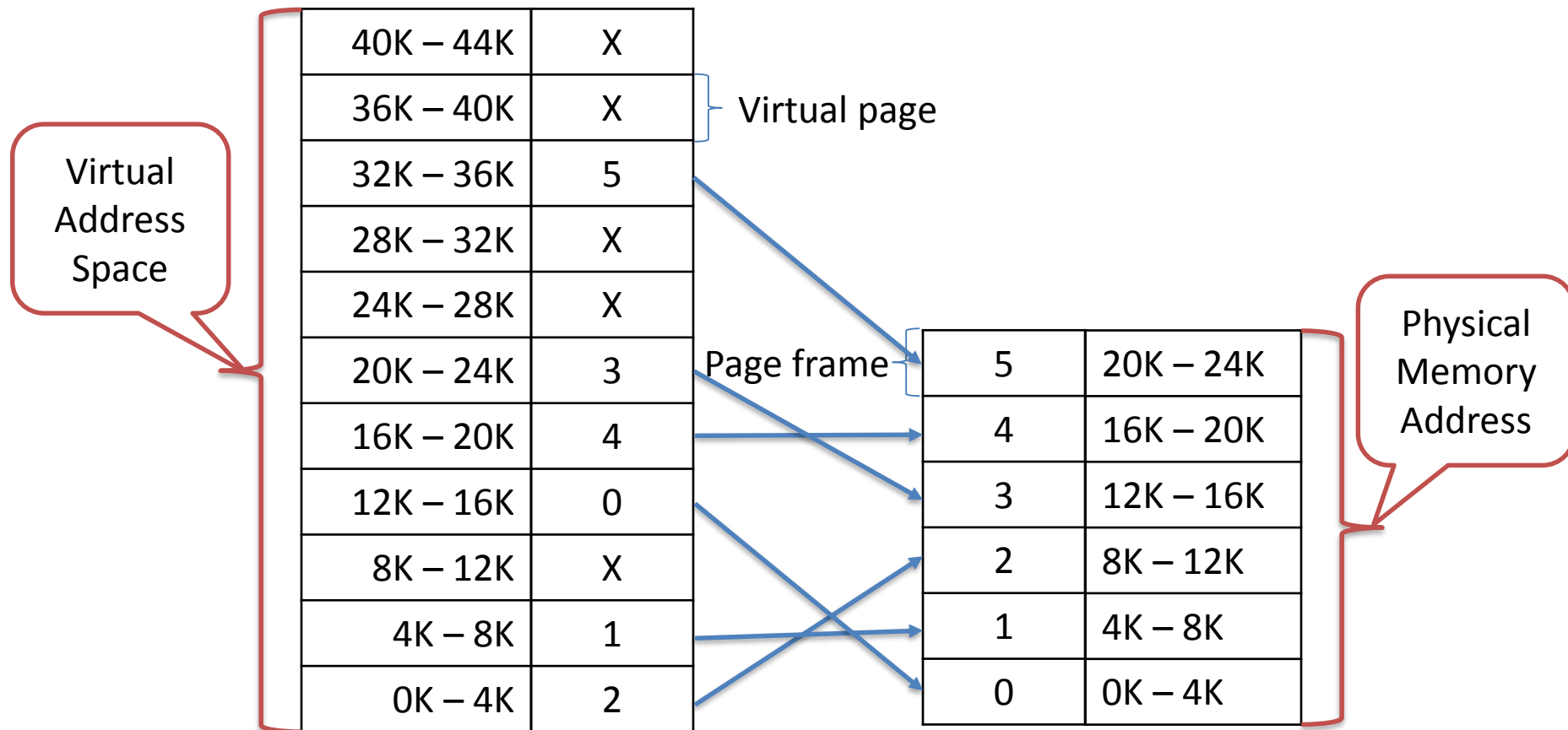
Conversion of virtual address to physical address

- We have a computer generated 16-bit addresses, from 0 up to 44K. These are the virtual addresses.

With 44 KB of virtual address space , we get 11 virtual pages and 24 KB of physical memory, we get 6 page frames.

Virtual Address Space

Virtual page

Virtual	Page frame
40K – 44K	X
36K – 40K	X
32K – 36K	5
28K – 32K	X
24K – 28K	X
20K – 24K	3
16K – 20K	4
12K – 16K	0
8K – 12K	X
4K – 8K	1
0K – 4K	2

Page frame

Frame	Physical
5	20K – 24K
4	16K – 20K
3	12K – 16K
2	8K – 12K
1	4K – 8K
0	0K – 4K

Physical Memory Address
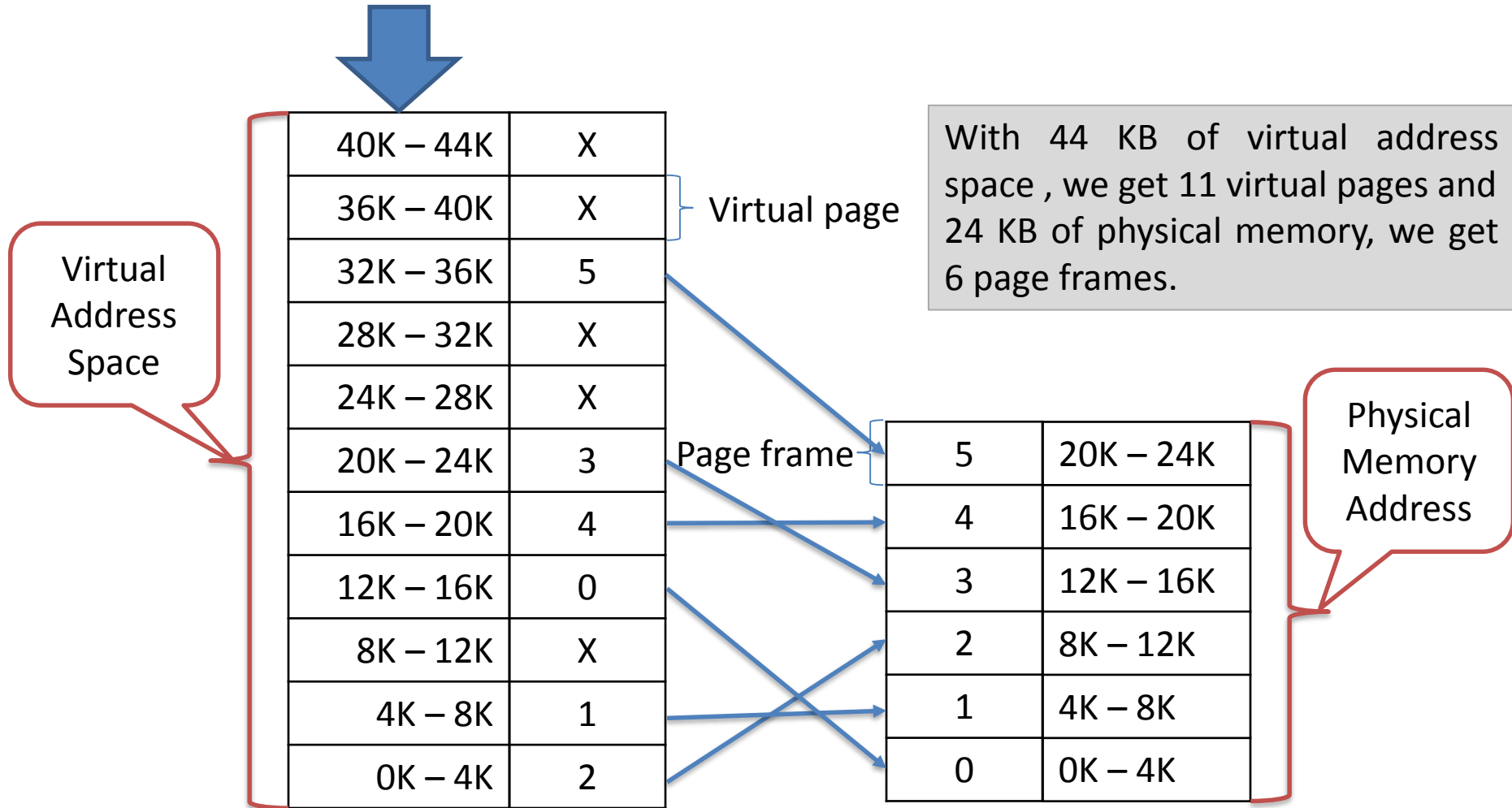
Conversion of virtual address to physical address

- However, only 24KB of physical memory is available, so although 44KB programs can be written, they cannot be loaded in to memory in their entirety and run.
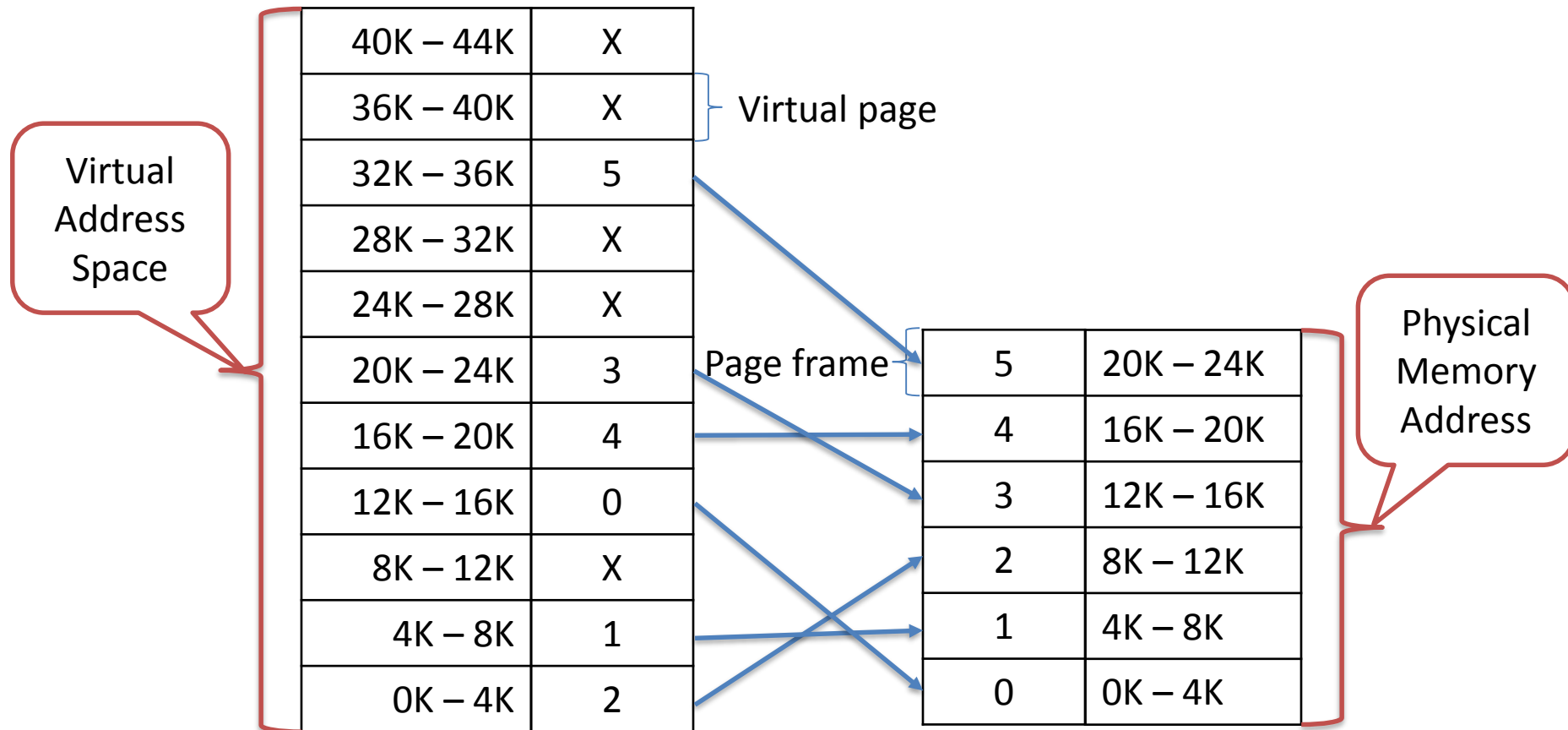
Virtual Address Space

40K – 44K	X
36K – 40K	X
32K – 36K	5
28K – 32K	X
24K – 28K	X
20K – 24K	3
16K – 20K	4
12K – 16K	0
8K – 12K	X
4K – 8K	1
0K – 4K	2

Virtual page

Page frame

Physical Memory Address

5	20K – 24K
4	16K – 20K
3	12K – 16K
2	8K – 12K
1	4K – 8K
0	0K – 4K

Conversion of virtual address to physical address

- A complete copy of a program's core image, up to 44 KB, must be present on the disk.

- Only required pages are loaded in the physical memory.

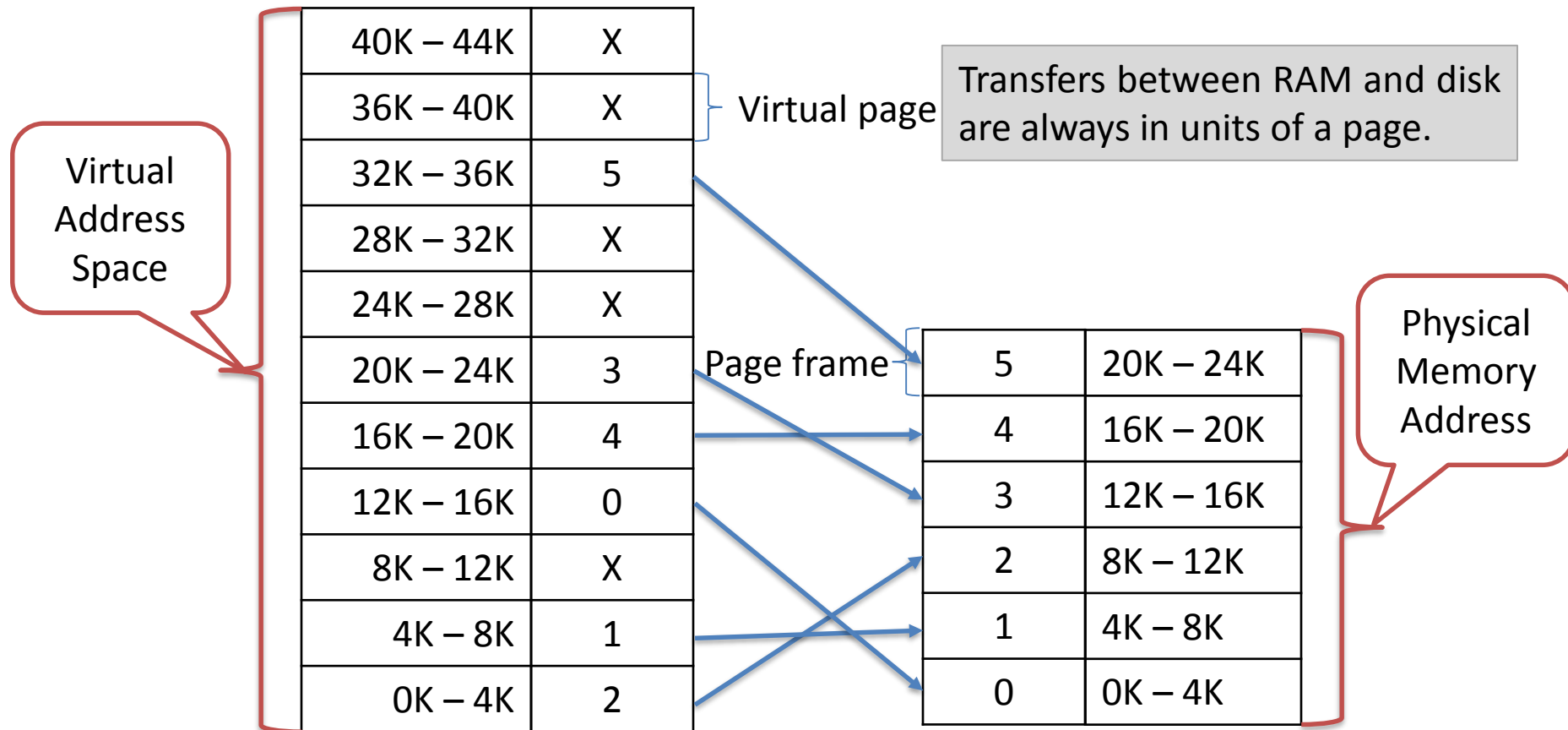| Virtual Address Space				Physical Memory Address
40K – 44K	X			
36K – 40K	X	Virtual page		
32K – 36K	5			
28K – 32K	X			
24K – 28K	X			
20K – 24K	3	Page frame	5	20K – 24K
16K – 20K	4		4	16K – 20K
12K – 16K	0		3	12K – 16K
8K – 12K	X		2	8K – 12K
4K – 8K	1		1	4K – 8K
0K – 4K	2		0	0K – 4K

Conversion of virtual address to physical address

- A complete copy of a program's core image, up to 44 KB, must be present on the disk.

- Only required pages are loaded in the physical memory.

Virtual Address Space		
40K − 44K	X
36K − 40K	X
32K − 36K	5
28K − 32K	X
24K − 28K	X
20K − 24K	3
16K − 20K	4
12K − 16K	0
8K − 12K	X
4K − 8K	1
0K − 4K	2

Virtual page

Transfers between RAM and disk are always in units of a page.

Page frame
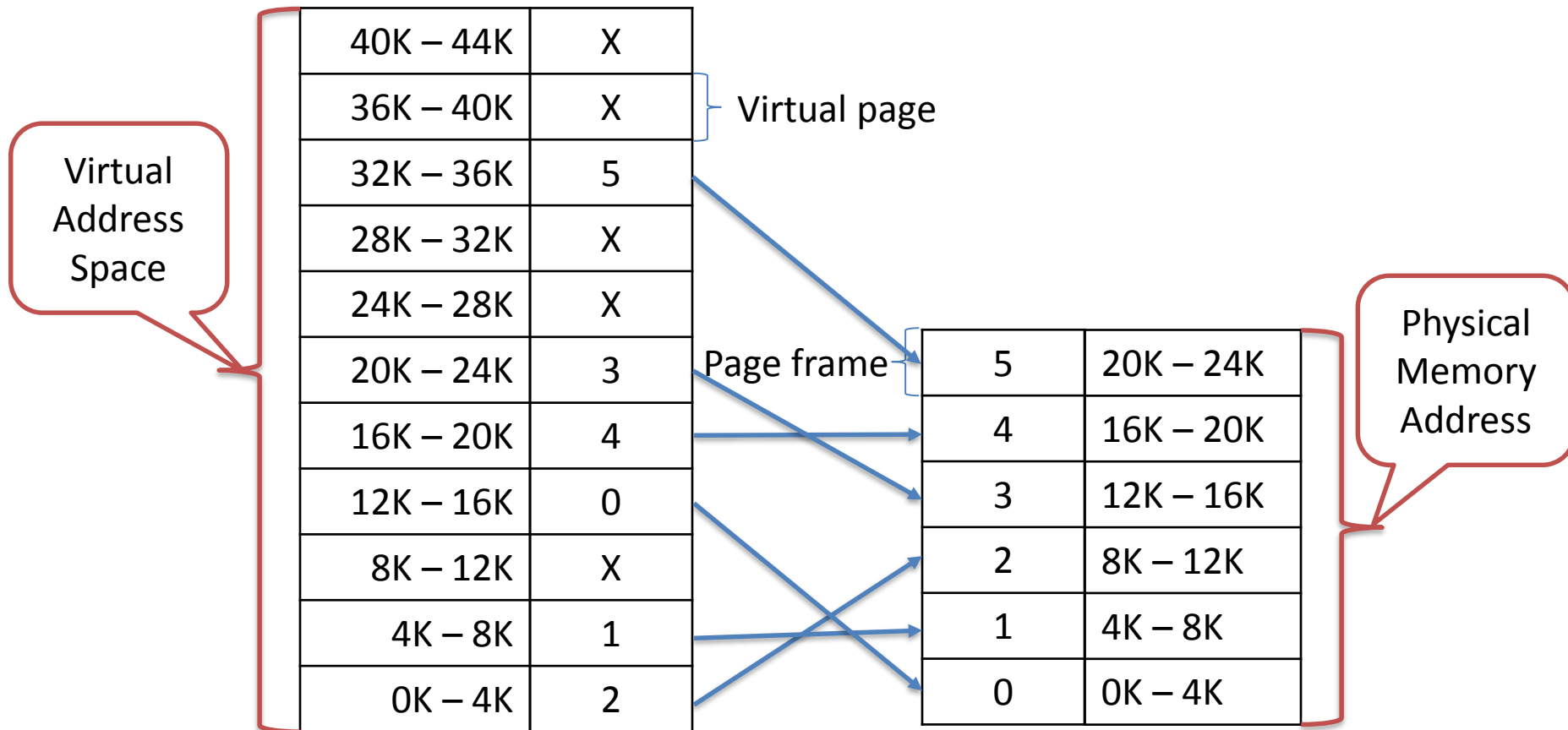
Page frame	Physical Memory Address
5	20K − 24K
4	16K − 20K
3	12K − 16K
2	8K − 12K
1	4K − 8K
0	0K − 4K

Internal operation of the MMU

Outgoing physical address (24580)

| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Page Table

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

12 bit offset copied directly from input to output

Virtual page=2 is used as an **index** into the page table

110

Present/ Absent bit

| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Incoming logical address (8196)
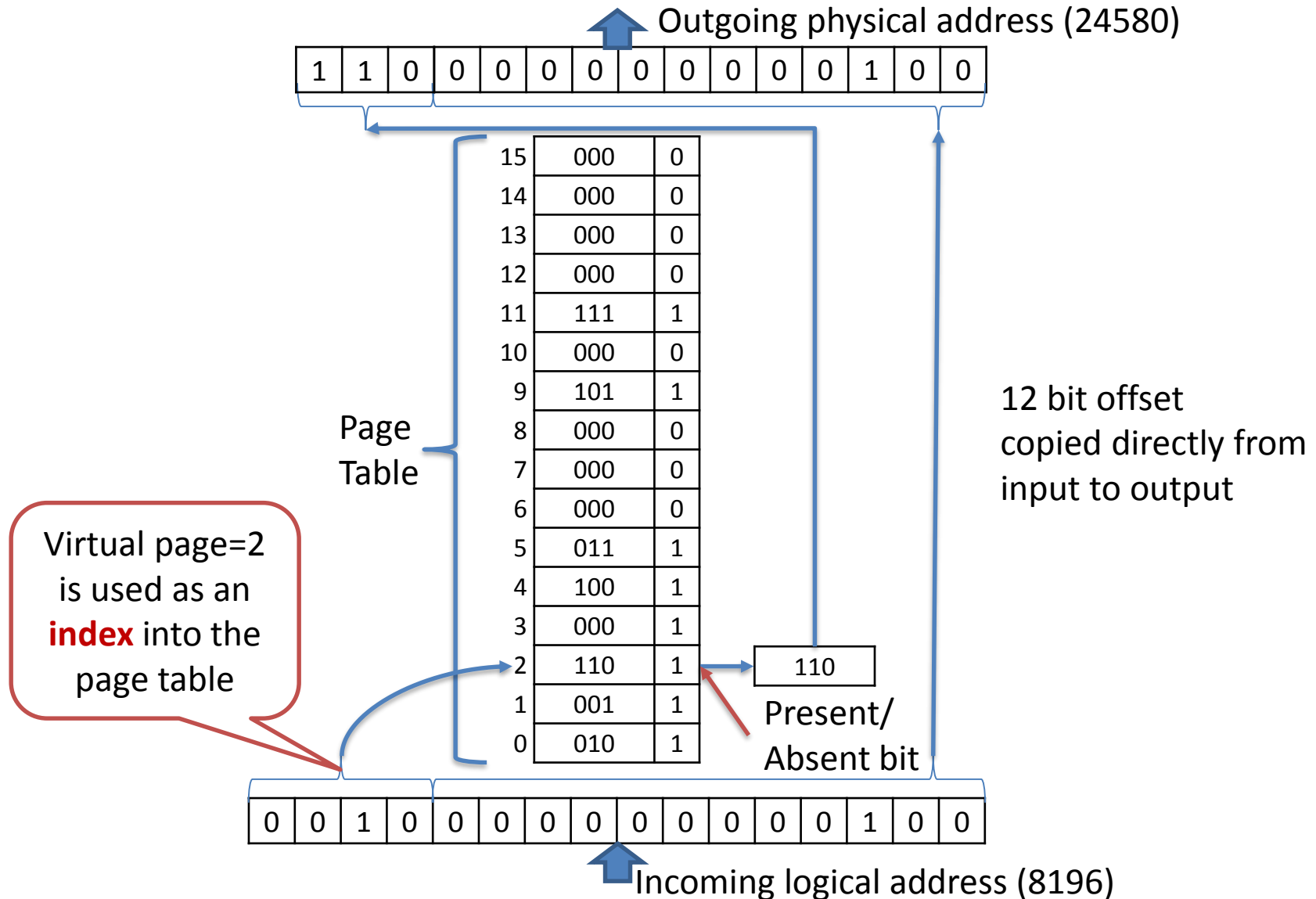
Conversion of virtual address to physical address

- The **virtual address is split into a virtual page number** (high order bits) and **an offset** (low-order bits).

- With a 16-bit address and a 4KB page size, the **upper 4 bits could specify one of the 11 virtual pages** and the **lower 12 bits would then specify the byte offset** (0 to 4095) within the selected page.

- The **virtual page number is used as an index** into the Page table.

Conversion of virtual address to physical address

- If the **present/absent bit is 0, it is page-fault**; a **trap to the operating system is caused to bring required page** into main memory.

- If the **present/absent bit is 1**, **required page is there with main memory** and page frame number found in the page table is copied to the higher order bit of the output register along with the offset.

- Together Page frame number and offset creates physical address.

- **Physical Address = Page frame Number + offset of virtual address**.

Page table

- Page table is a **data structure which translates virtual address into equivalent physical address**.

- The virtual page number is used as an index into the Page table to find the entry for that virtual page and from the Page table physical page frame number is found.

- Thus the **purpose of page table is to map virtual pages onto page frames**.

Page table structure

Page frame number

Protection   Present/ absent
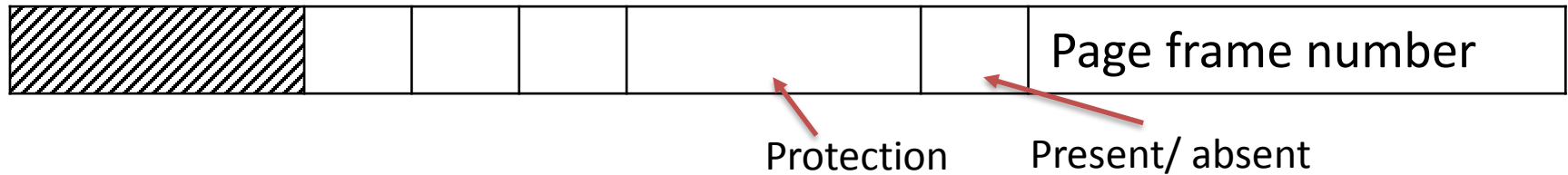
- Page frame Number: It gives the **frame number in which the current page you are looking for is present**.

- Present/Absent bit: Present or absent bit **says whether a particular page you are looking for is present or absent**. If it is not present, that is called **Page Fault**. It is **set to 0 if the corresponding page is not in memory**. Sometimes this bit is also known as **valid/invalid bits**.

- The Protection bits: Protection bit says that **what kind of protection you want on that page**. In the simplest form, **0 for read/write** and **1 for read only**.

Page table structure

						Page frame number

Caching Disabled   Referenced   Modified   Protection   Present/ absent
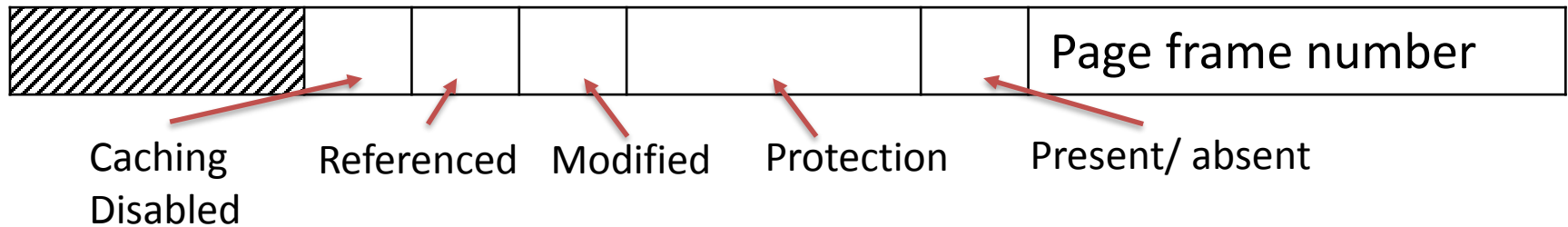
- Modified bit: Modified bit says **whether the page has been modified or not**. If the **page in memory has been modified**, it **must be written back to disk**. This bit is also called as **dirty bit** as it reflects the page's state.

- Referenced bit: A references bit is **set whenever a page is referenced, either for reading or writing**. Its **value helps operating system in page replacement algorithm**.

- Cashing Disabled bit: This feature is important for **pages that maps onto device registers rather than memory**. With this bit cashing can be turned off.

Demand Paging

- In paging system, processes are started up with none of their pages in memory.

- When CPU tries to fetch the first instruction, it gets page fault, other page faults for global variables and stack usually follow quickly.

- After a while, the process has most of the pages it needs in main memory and it has few page faults.

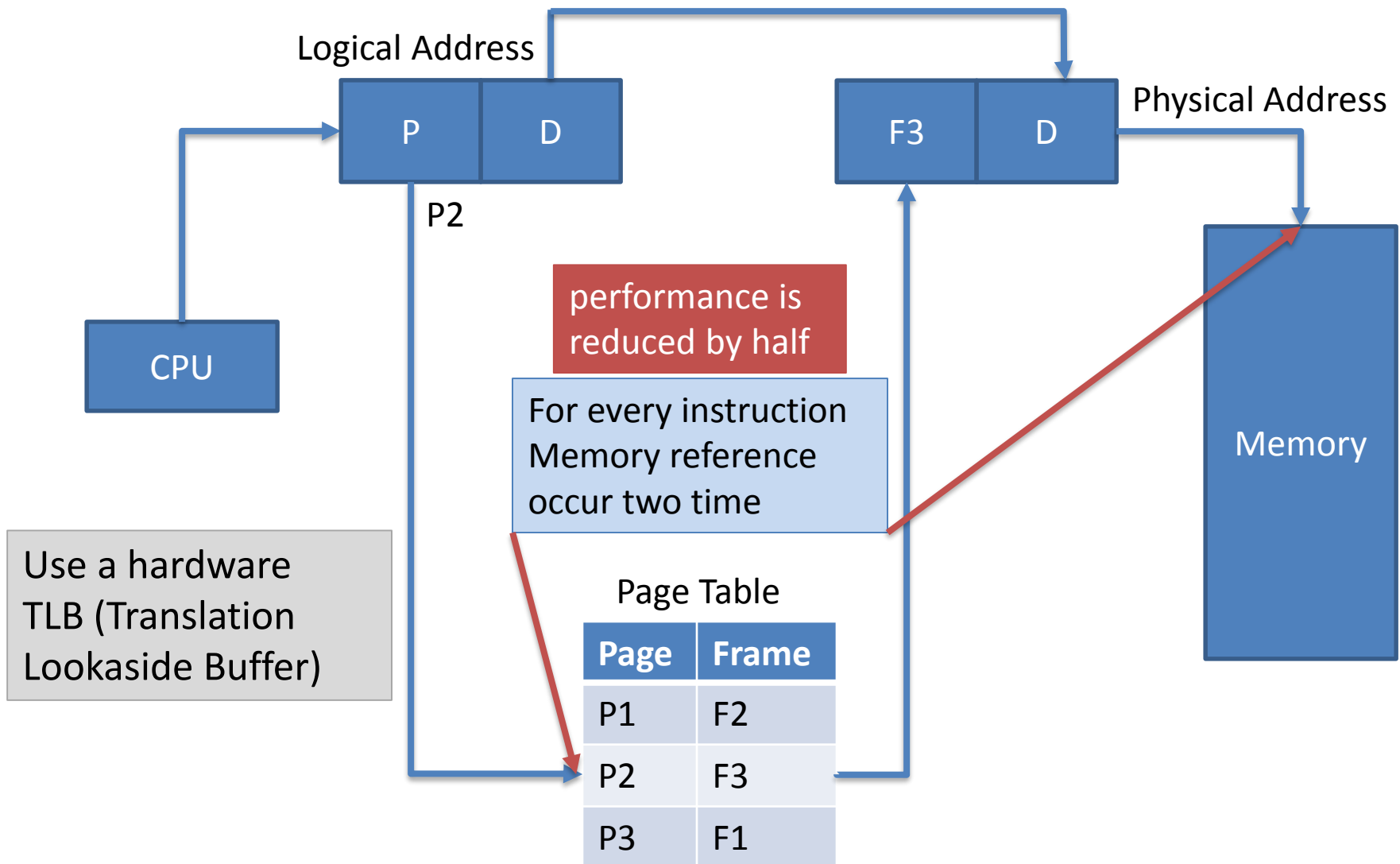- This strategy is called demand paging because **pages are loaded only on demand, not in advance**.

Definitions

- **Working Set**: The **set of pages that a process is currently using** is known as working set.

- **Thrashing**: A **program causing page faults every few instructions** is said to be thrashing.

- **Pre-paging**: Many paging systems **try to keep track of each process's working set** and **make sure that it is in memory before letting the process run**.

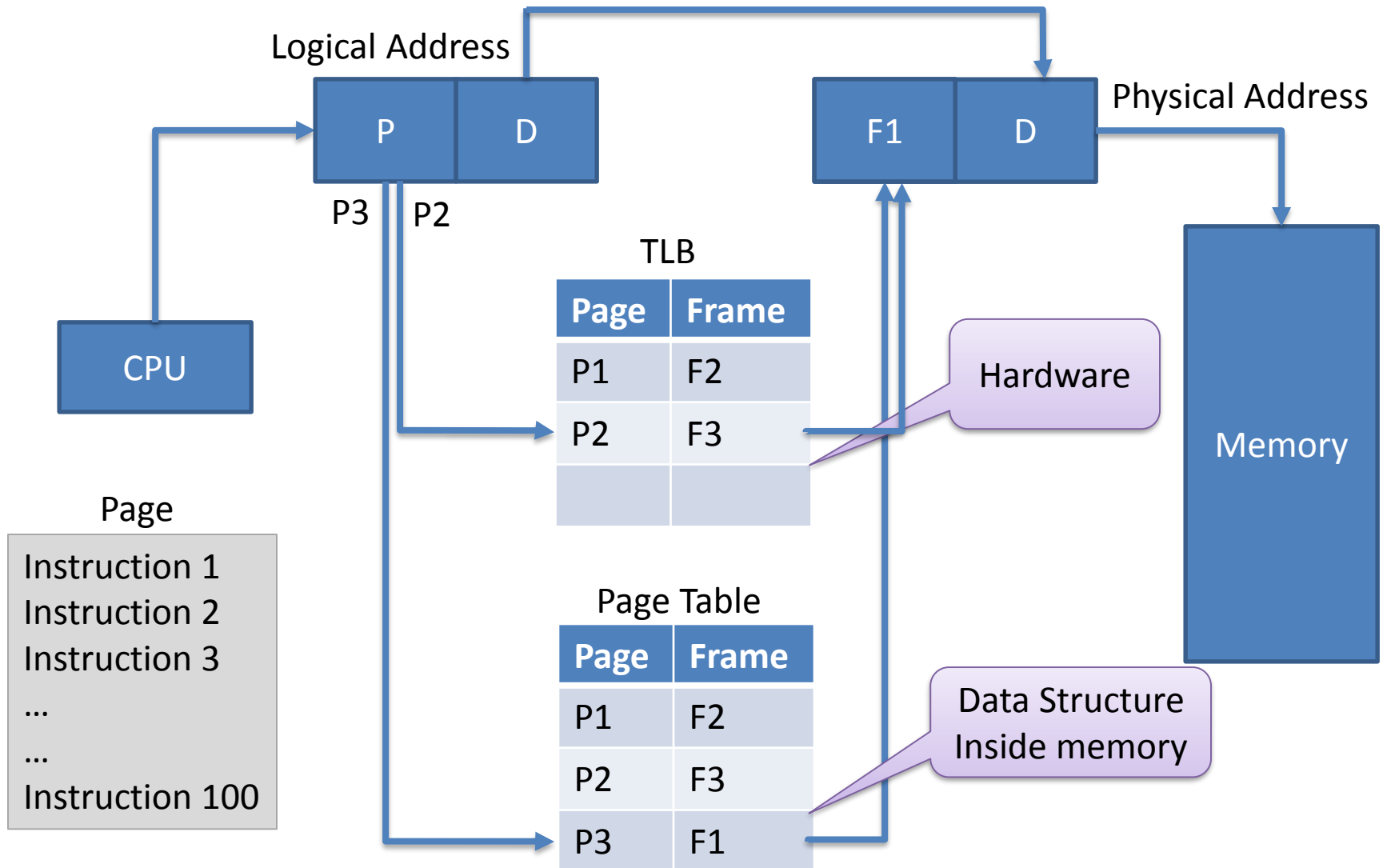- **Loading pages before allowing processes to run** is called pre-paging.

Issues in Paging

- In any paging system, two major issues must be faced:
    1. The **mapping from virtual address to physical address must be fast**.
    2. If the **virtual address space is large**, the **page table will be large**.

Mapping from virtual address to physical address must be fast

Logical Address

P	D

P2

F3	D

Physical Address

CPU

performance is reduced by half

For every instruction Memory reference occur two time

Memory

Use a hardware TLB (Translation Lookaside Buffer)

Page Table

Page	Frame
P1	F2
P2	F3
P3	F1

Mapping from virtual address to physical address must be fast

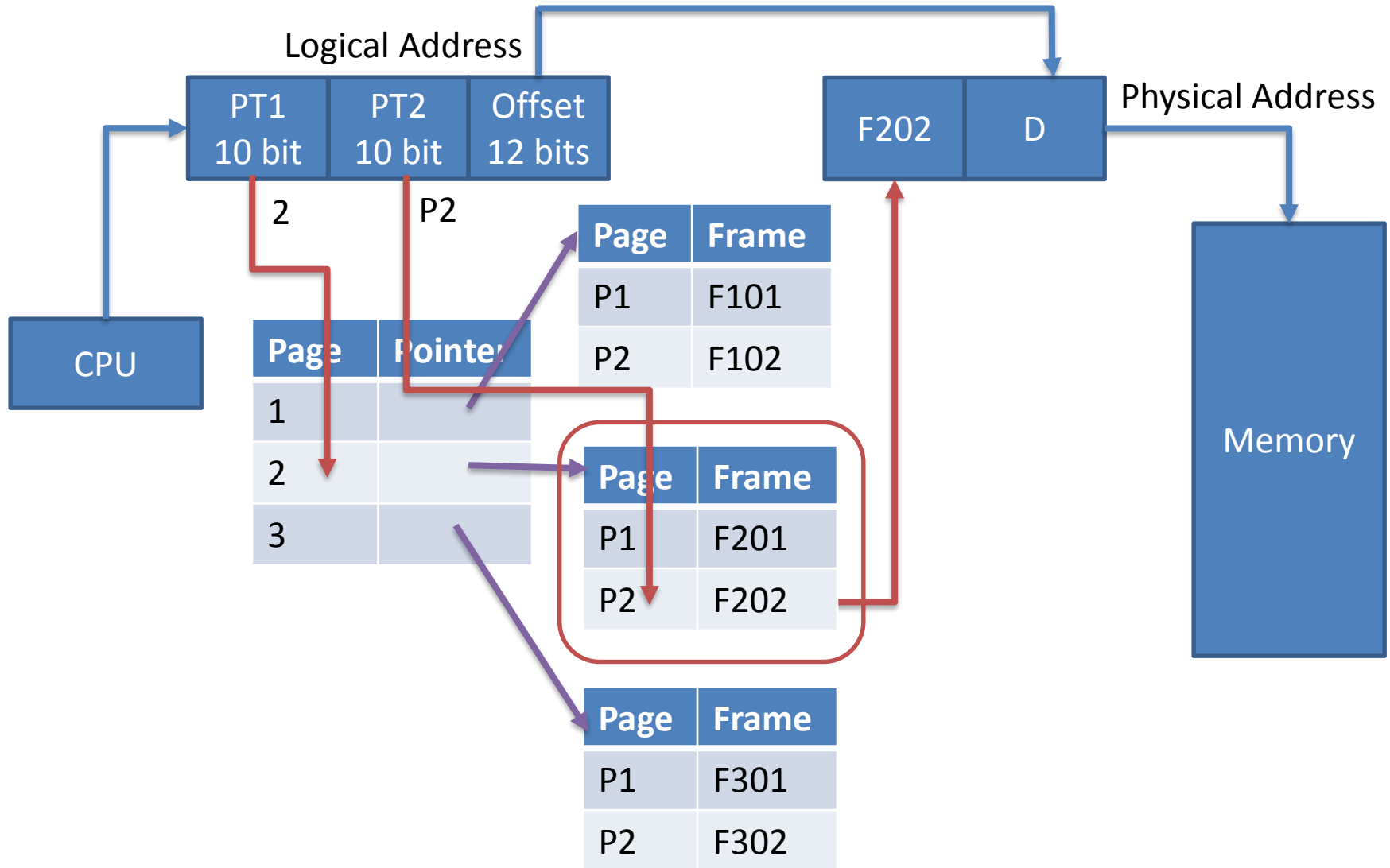# Mapping from virtual address to physical address using TLB

- Steps in TLB hit:
  1. CPU generates virtual address.
  2. It is checked in TLB (present).
  3. Corresponding frame number is retrieved, which now tells where in the main memory page lies.

- Steps in Page miss:
  1. CPU generates virtual address.
  2. It is checked in TLB (not present).
  3. Now the page number is matched to page table residing in main memory.
  4. Corresponding frame number is retrieved, which now tells where in the main memory page lies.
  5. The TLB is updated with new Page Table Entry (if space is not there, one of the replacement technique comes into picture i.e either FIFO, LRU or MFU etc).

Virtual address space is large, the page table will be large

- Two different ways to deal with large page table problems:
    1. Multilevel Page Table
    2. Inverted Page Table

Multilevel Page Table

Logical Address

PT1 10 bit	PT2 10 bit	Offset 12 bits

2   P2

F202	D

Physical Address

CPU

Page	Frame
P1	F101
P2	F102

Page	Pointer
1	
2	
3	

Page	Frame
P1	F201
P2	F202

Memory

Page	Frame
P1	F301
P2	F302

Inverted Page Table

Current process ID - 245

Page No 2	Offset

Process IDs do not match.
So follow chaining pointer

Frame No	Page No	Process ID	Pointer
1	2	211	
2			
3			
4	2	245	
5			
6			

Frame No 4	Offset

Physical Address

Process IDs match.
So frame no added to
physical address.

Inverted Page Table

- Each entry in the page table contains the following fields.

  1. Page number – It specifies the page number range of the logical address.

  2. 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 processID 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 Page Table

- Each entry in the page table contains the following fields.

   3. Control bits – These bits are used to store extra paging-related information. These include the valid bit, dirty bit, reference bits, protection and locking information bits.

   4. Chained pointer – It may be possible sometime that two or more processes share a part of main memory.

      In this case, two or more logical pages map to same Page Table Entry then a chaining pointer is used to map the details of these logical pages to the root page table.

Page replacement algorithms

- Following are different types of page replacement algorithms
  1. Optimal Page Replacement Algorithm
  2. FIFO Page Replacement Algorithm
  3. The Second Chance Page Replacement Algorithm
  4. The Clock Page Replacement Algorithm
  5. LRU (Least Recently Used) Page Replacement Algorithm
  6. NRU (Not Recently Used)

Optimal Page Replacement Algorithm

- The moment a page fault occurs, some set of pages will be in the memory.

- One of these pages will be referenced on the very next instruction.

- Other pages may not be referenced until 10, 100, or perhaps 1000 instructions later.

- Each **page can be labeled with the number of instructions that will be executed before that page is first referenced**.

- The optimal page algorithm simply says that the **page with the highest label should be removed**.

- The only problem with this algorithm is that it is **unrealizable**.

- At the time of the page fault, the operating system has no way of knowing when each of the pages will be referenced next.

Optimal Page Replacement Algorithm

- Page Reference String:
  - 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 0, 2, 0, 1, 7, 0, 1
  - Three frames

Page Requests	7	0	1	2	0	3	0	4	2	3	0	3	2	0	2	0	1	7	0	1
Frame 1	7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
Frame 2		0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
Frame 3			1	1	1	3	3	3	3	3	3	3	3	3	3	3	3	7	7	7
Page Faults (9)	F	F	F	F		F		F			F						F	F		

FIFO Page Replacement Algorithm

- The first in first out page replacement algorithm is the simplest page replacement algorithm.

- The operating system **maintains a list of all pages currently in memory**, with the **most recently arrived page at the tail and least recent at the head**.

- On a page fault, the **page at head is removed** and the **new page is added to the tail**.

- When a page replacement is required the **oldest page in memory needs to be replaced**.

- The performance of the FIFO algorithm is not always good because **it may happen that the page which is the oldest is frequently referred by OS**.

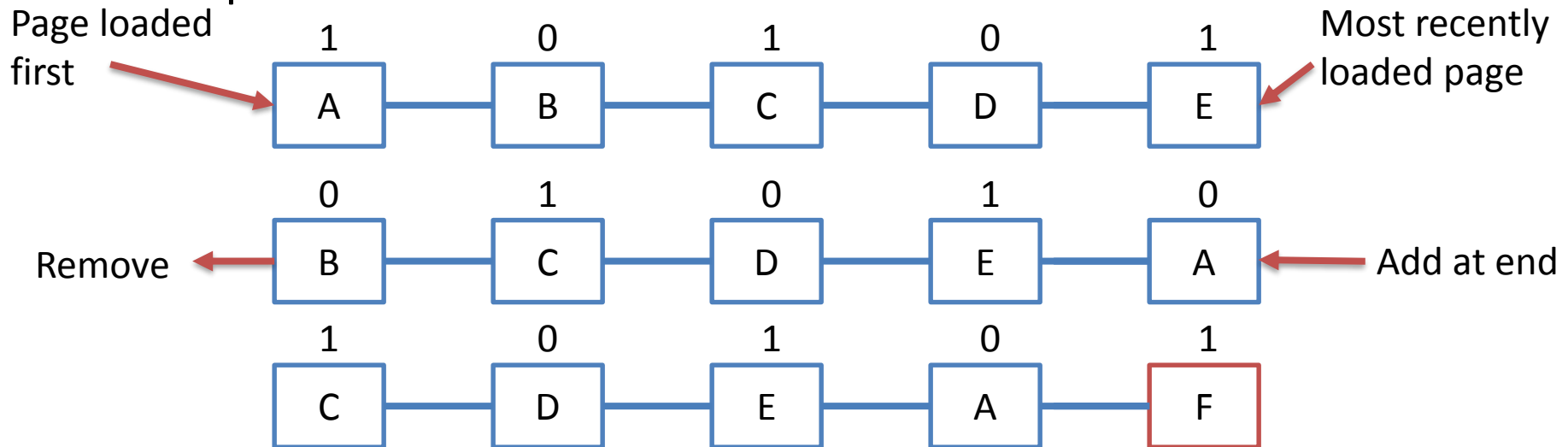- Hence removing the oldest page may create page fault again.

FIFO Page Replacement Algorithm

- Page Reference String:
  - 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
  - Three frames

| Page Requests	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Frame 1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
Frame 2		0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
Frame 3			1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1
Page Faults (15)	F	F	F	F		F	F	F	F	F	F			F	F			F	F	F

Second Chance Page Replacement Algorithm

- It is **modified form of the FIFO** page replacement algorithm.

- It looks at the front of the queue as FIFO does, but instead of immediately paging out that page, it **checks to see if its referenced bit is set**.

  - If it is **not set (zero)**, the **page is swapped out**.

  - Otherwise, the **referenced bit is cleared**, the **page is inserted at the back of the queue** (as if it were a new page) and this process is repeated.

Page loaded first

1	0	1	0	1
A	B	C	D	E

Most recently loaded page

0	1	0	1	0
B	C	D	E	A

Remove   Add at end

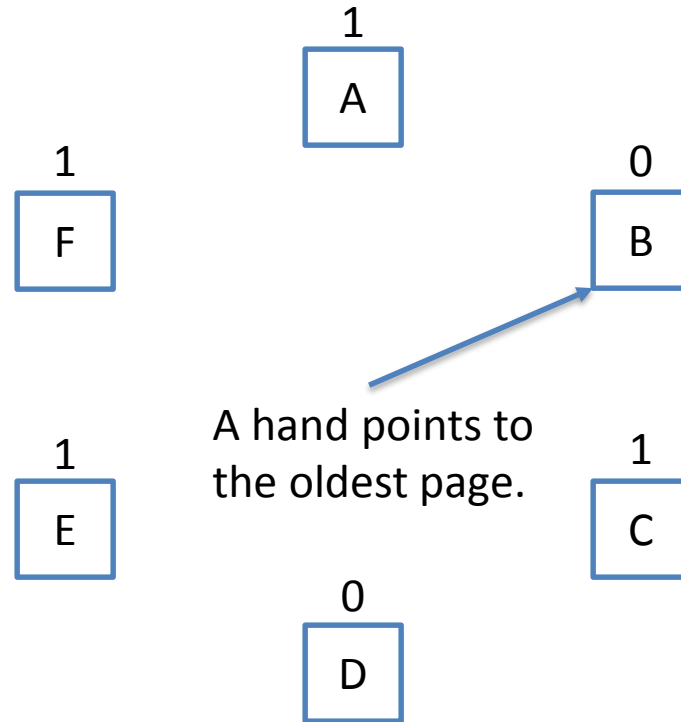| 1	0	1	0	1
C	D	E	A	F

Second Chance Page Replacement Algorithm

- If all the pages have their referenced bit set, on the second encounter of the first page in the list, that page will be swapped out, as it now has its referenced bit cleared.

- If all the pages have their reference bit cleared, then second chance algorithm degenerates into pure FIFO.
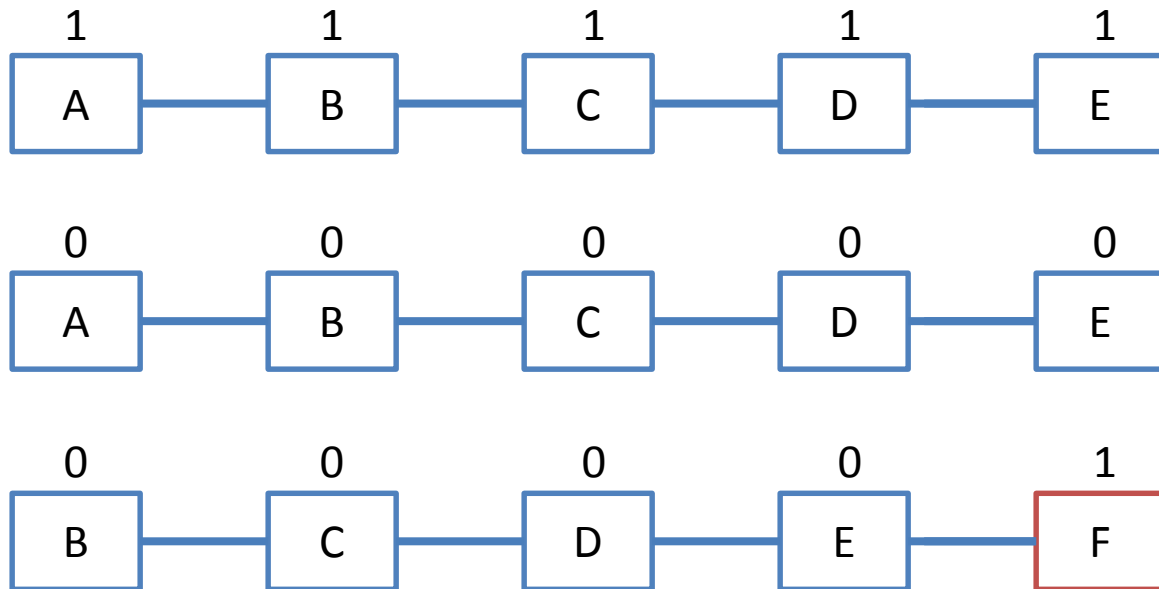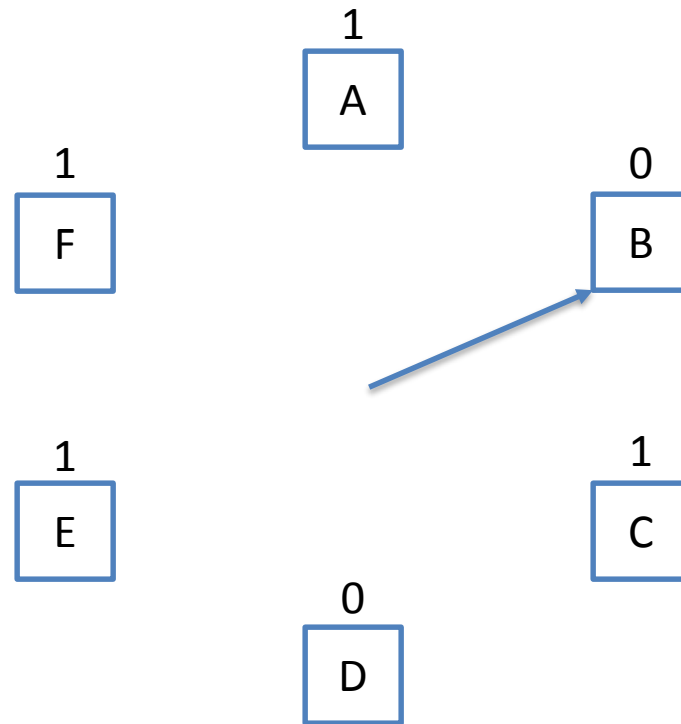
Clock Page Replacement Algorithm

- In second chance algorithm **pages are constantly moving around on its list**. So it is not working efficiently.

- A better approach is to **keep all the page frames on a circular list** in the form of a clock.

```
              1
             [A]

   1                    0
  [F]                  [B]
                      ↗
                A hand points to
                the oldest page.
   1                    1
  [E]                  [C]
              0
             [D]
```

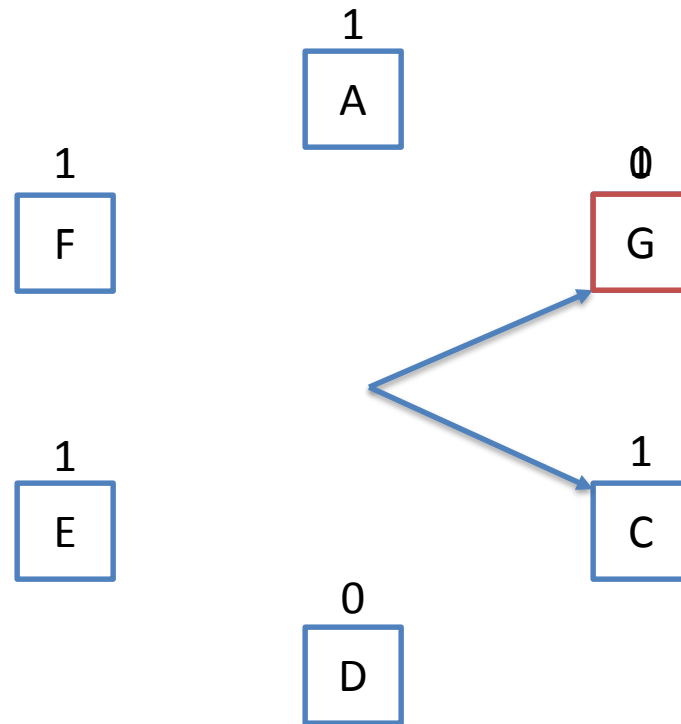A hand points to the oldest page.

Clock Page Replacement Algorithm

- When a page fault occurs, the **page being pointed to by the hand is inspected**.

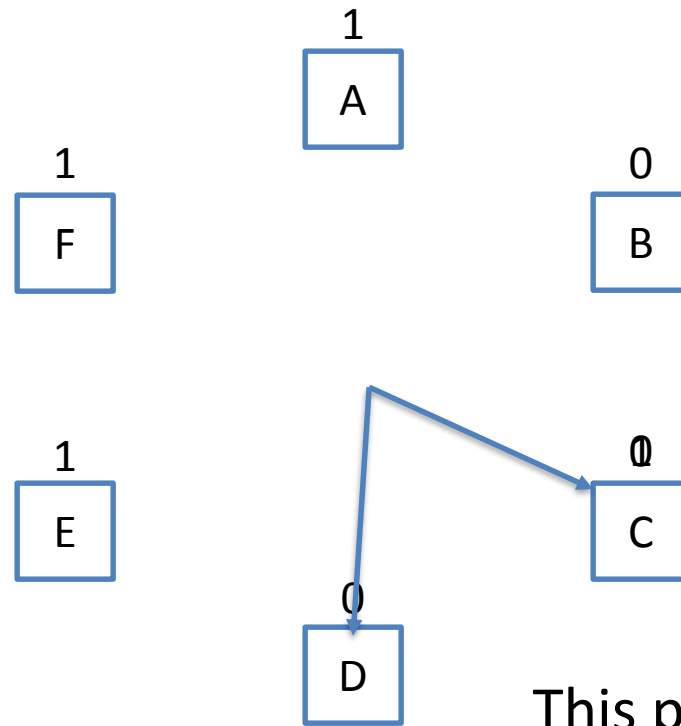# Clock Page Replacement Algorithm

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

Clock Page Replacement Algorithm

- If **R is 1**, it is **cleared** and the **hand is advanced to the next page**.

1

A

1

F

0

B

1

E

0

C

0

D

This process is repeated until a page is found with R = 0.

LRU (Least Recently Used) Page Replacement Algorithm

- A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in last few instructions will probably be heavily used again in next few instructions.

- When page fault occurs, **throw out the page that has been used for the longest time**. This strategy is called LRU (Least Recently Used) paging.

- To fully implement LRU, it is necessary to **maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear**.

- The list must be updated on every memory reference.

- Finding a page in the list, deleting it, and then moving it to the front is a **very time consuming operations**.

LRU (Least Recently Used) Page Replacement Algorithm

- Page Reference String:
  - 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
  - Three frames

Page Requests	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Frame 1	7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
Frame 2		0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
Frame 3			1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7
Page Faults (12)	F	F	F	F		F		F	F	F	F			F		F		F		

NRU (Not Recently Used) Page Replacement Algorithm

- NRU makes approximation to replace the page based on **R (referenced) and M (modified) bits**.

- When a **process is started up**, **both page bits for all pages are set to 0** by operating system.

- Periodically, the **R bit is cleared**, to distinguish pages that have not been referenced recently from those that have been.

- When page fault occurs, the operating system inspects all the pages and divide them into 4 categories based on current values of their R and M bits

  1. Case 0 : not referenced, not modified
  2. Case 1 : not referenced, modified
  3. Case 2 : referenced, not modified
  4. Case 3 : referenced, modified

- The NRU (Not Recently Used) algorithm removes a page at random from the lowest numbered nonempty class.

NRU (Not Recently Used) Page Replacement Algorithm

- For example if,
    1. Page-0 is of class-2 (referenced, not modified)
    2. Page-1 is of class-1 (not referenced, modified)
    3. Page-2 is of class-0 (not referenced, not modified)
    4. Page-3 is of class-3 (referenced, modified)
- So lowest class page-2 needs to be replaced by NRU.

Belady's Anomaly

- Page Reference String:
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Page Faults of 3 Frame > Page Faults of 4 Frame

Belady's Anomaly

Three Frames

Page Requests	1	2	3	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	4	4	4	5	5	5	5	5	5
Frame 2		2	2	2	1	1	1	1	1	3	3	3
Frame 3			3	3	3	2	2	2	2	2	4	4
Page Faults **(9)**	F	F	F	F	F	F	F			F	F	

Four Frames

Page Requests	1	2	3	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	1	1	1	5	5	5	5	4	4
Frame 2		2	2	2	2	2	2	1	1	1	1	5
Frame 3			3	3	3	3	3	3	2	2	2	2
Frame 4				4	4	4	4	4	4	3	3	3
Page Faults **(10)**	F	F	F	F			F	F	F	F	F	F

Belady's Anomaly

- Belady's anomaly is the **phenomenon in which increasing the number of page frames results in an increase in the number of page faults for certain memory access patterns**.

- This **phenomenon is commonly experienced** when using the **first-in first-out (FIFO)** page replacement algorithm.

- In **FIFO, the page fault may or may not increase as the page frames increase**, but **in Optimal and stack-based algorithms like LRU, as the page frames increase the page fault decreases**.

Sum

- A computer has four page frames. The time of loading, time of last access and the R and M bit for each page given below. Which page FIFO, LRU and NRU will replace.

Page	Loaded	Last Ref.	R	M
0	126	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

- **FIFO**:- Page which is arrived first needs to be removed first, so **page-3** needs to replace.

Sum

- A computer has four page frames. The time of loading, time of last access and the R and M bit for each page given below. Which page FIFO, LRU and NRU will replace.

Page	Loaded	Last Ref.	R	M
0	126	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

- **LRU**:- When page fault occurs, throw out the page that has been used for the longest time.
- Page **page-1** is not used for the long time from all four, so LRU suggest replacing **page-1**.

Sum

- A computer has four page frames. The time of loading, time of last access and the R and M bit for each page given below. Which page FIFO, LRU and NRU will replace.
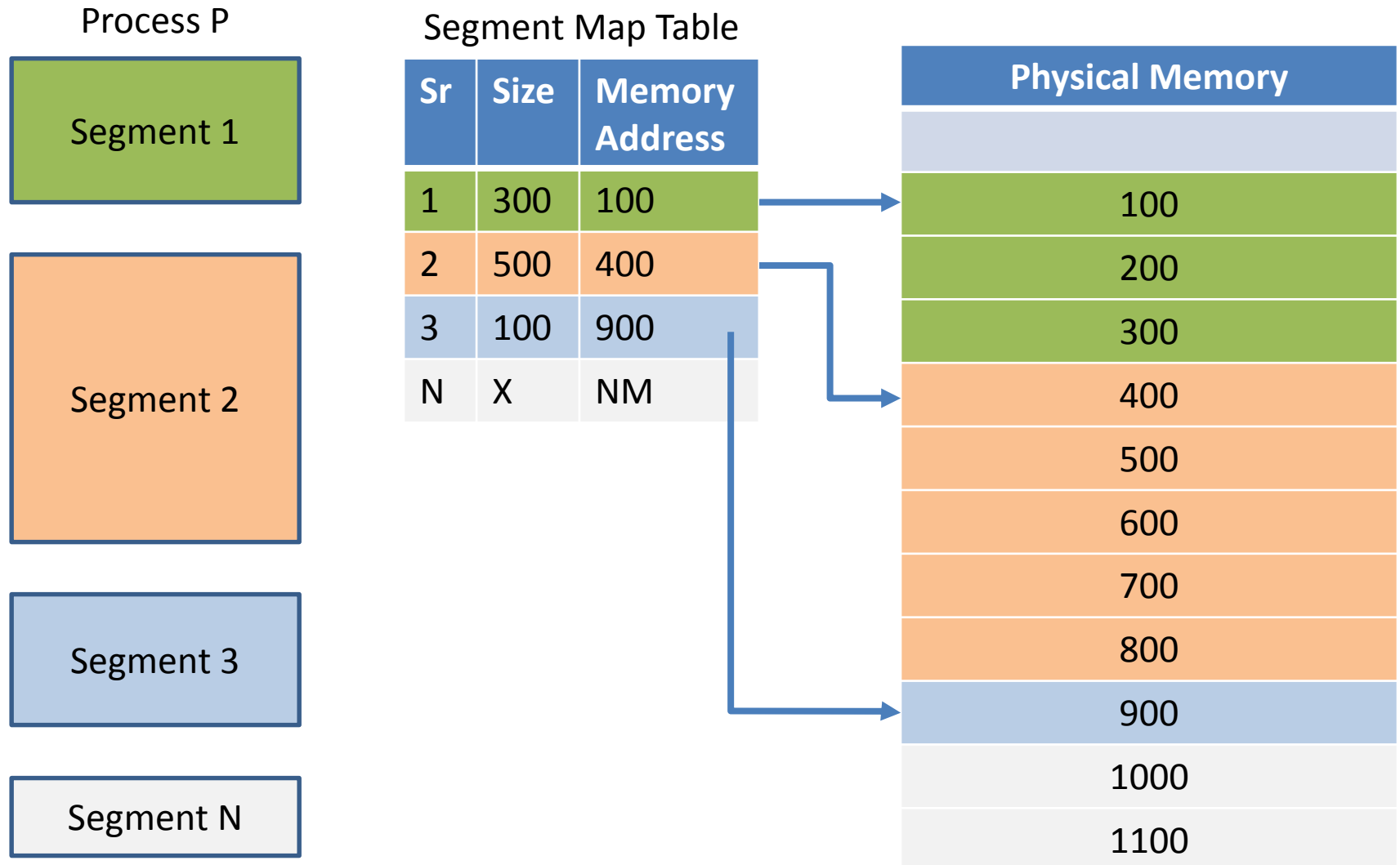
Page	Loaded	Last Ref.	R	M
0	126	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

- **NRU**:-
  - Page-0 is of class-2 (referenced, not modified)
  - Page-1 is of class-1 (not referenced, modified)
  - Page-2 is of class-0 (not referenced, not modified)
  - Page-3 is of class-3 (referenced, modified)
  - So lowest class **page-2** needs to be replaced by NRU

Segmentation

- Segmentation is a **memory management technique** in which each **job is divided into several segments of different sizes**, one for each module that contains pieces that perform related functions.

- Each segment is actually a **different logical address space** of the program.

- When a process is to be executed, its **corresponding segmentation are loaded into non-contiguous memory** though every segment is loaded into a contiguous block of available memory.

- Segmentation memory management works very similar to paging but here **segments are of variable-length where as in paging pages are of fixed size**.

Segmentation

Process P

Segment 1

Segment 2

Segment 3

Segment N

Segment Map Table

Sr	Size	Memory Address
1	300	100
2	500	400
3	100	900
N	X	NM

Physical Memory
100
200
300
400
500
600
700
800
900
1000
1100

Segmentation

- A program segment contains the program's main function, utility functions, data structures, and so on.

- The operating system **maintains** a **segment map table** for every process.

- Segment map table contains **list of free memory blocks along with segment numbers**, their **size** and **corresponding memory locations in main memory**.

- For each segment, the **table stores the starting address** of the segment and the **length of the segment**.

- A reference to a memory location includes a value that identifies a segment and an offset.

Paging VS Segmentation

Paging	Segmentation