CSE308 Operating Systems

Memory Management

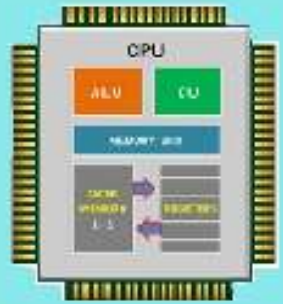
Dr S.Rajarajan

SoC

SASTRA

Types of Memory in a Computer

Registers



Primary

- RAM
  - DRAM
  - SRAM
  - SDRAM
- ROM
  - MROM
  - PROM
  - EPROM
  - EEPROM
  - Flash Memory

Secondary

- HDD
- SDD
- Optical Drives
- Flash Drives
- Memory Cards
- Magnetic Tapes
- NAS

Basic Hardware

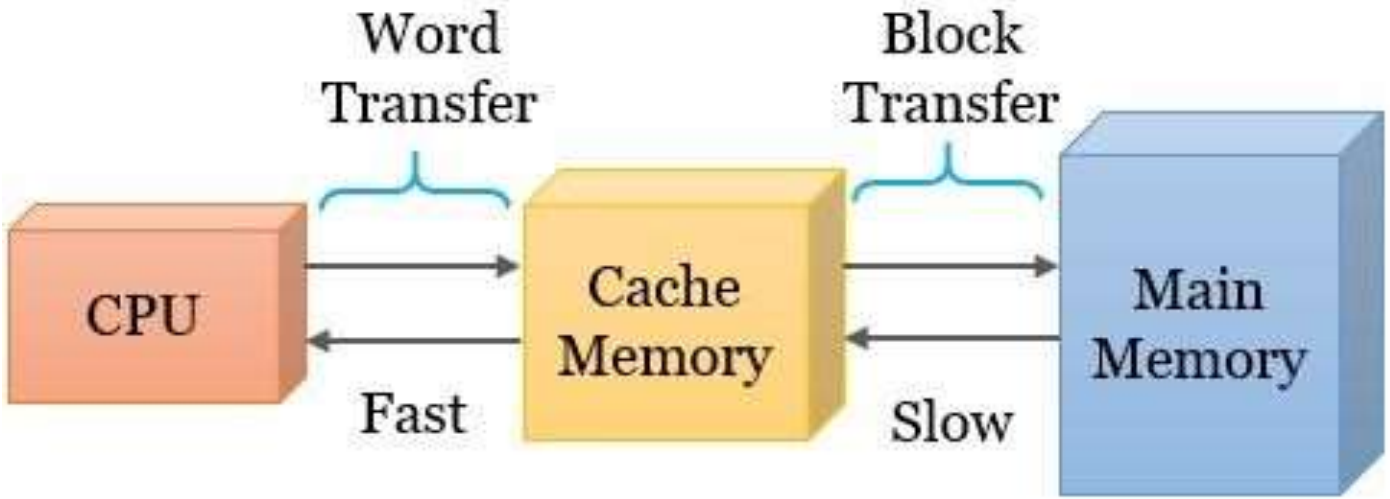
- **Main memory** and the **registers** inside **processor board** are the only storage that the **CPU can access directly**.
- There are **machine instructions** that take **memory addresses** as arguments, but none that take **disk addresses**.
- Therefore,
  - any **instructions** in execution,
  - and any **data being used** by the instructions,
- must be in one of these **direct-access storage devices**.
- If the **data are not in memory**, they must be **moved there** before the CPU can operate on them.

CPU’s Clock Cycle

- The clock speed measures the **number of cycles** your CPU **executes per second**, measured in GHz (gigahertz).
- A “cycle” is the basic unit that measures a CPU’s speed.
- During each cycle, **billions of transistors** within the processor **open and close**
- This is how the **CPU executes the calculations** contained in the instructions it receives.
- A CPU with a clock speed of **3.2 GHz executes 3.2 billion cycles per second**
- Sometimes, multiple instructions are completed in a single clock cycle; in other cases, one instruction might be handled over multiple clock cycles.

Memory Cycle

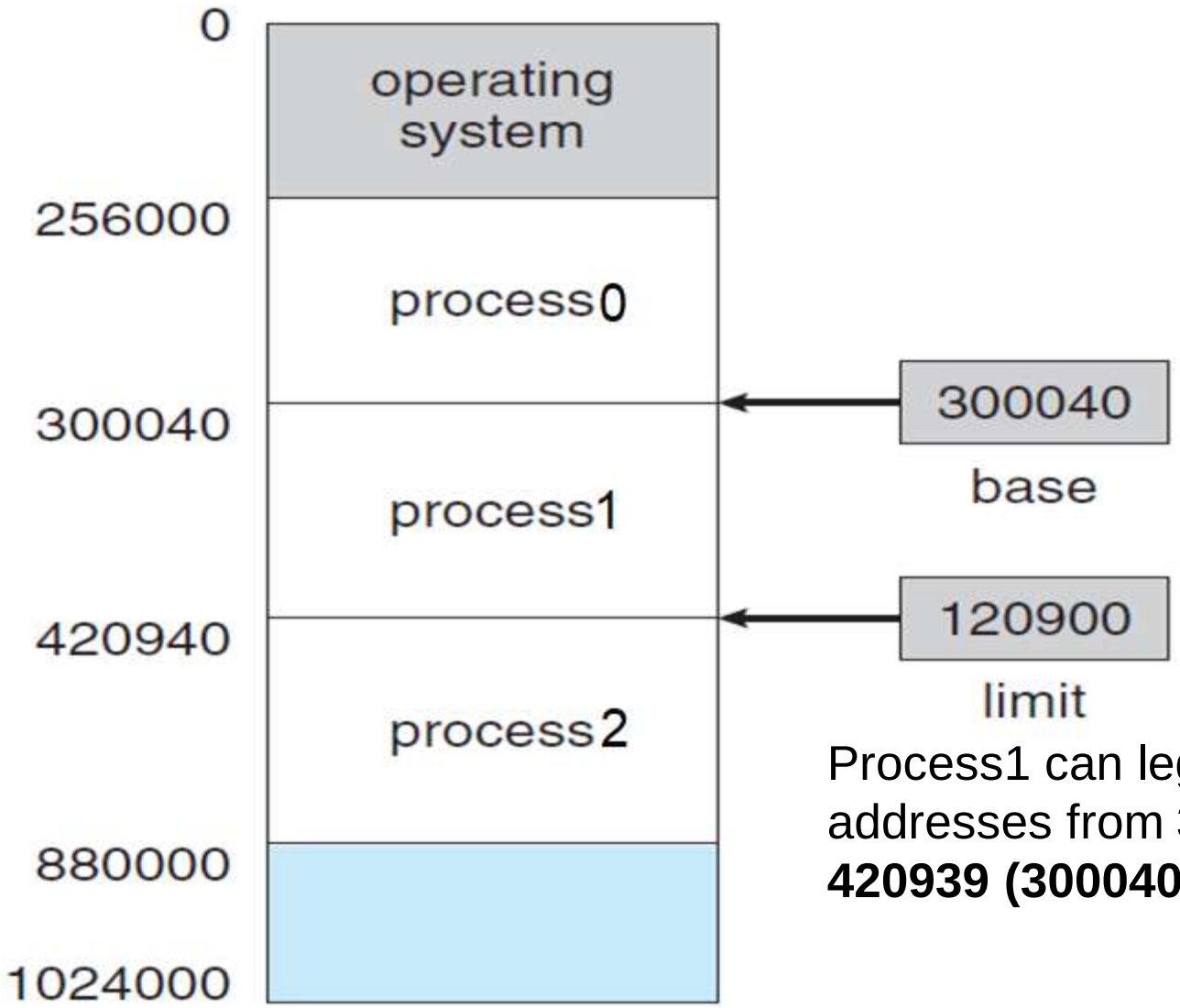
- **Registers** that are built into the CPU are generally accessible **within one cycle** of the CPU clock.
- The same cannot be said of main memory, which is accessed via a transaction on the ***memory bus***.
- Memory access may take **many cycles** of the **CPU clock** to complete, in which case the processor normally needs to **stall.**
- The remedy is to **add fast memory** between the **CPU and main memory**.
- A memory that is used to accommodate a speed differential is called a ***cache***.


Word
Transfer
Block
Transfer
CPU
Cache
Memory
Main
Memory
Fast
Slow

Protection

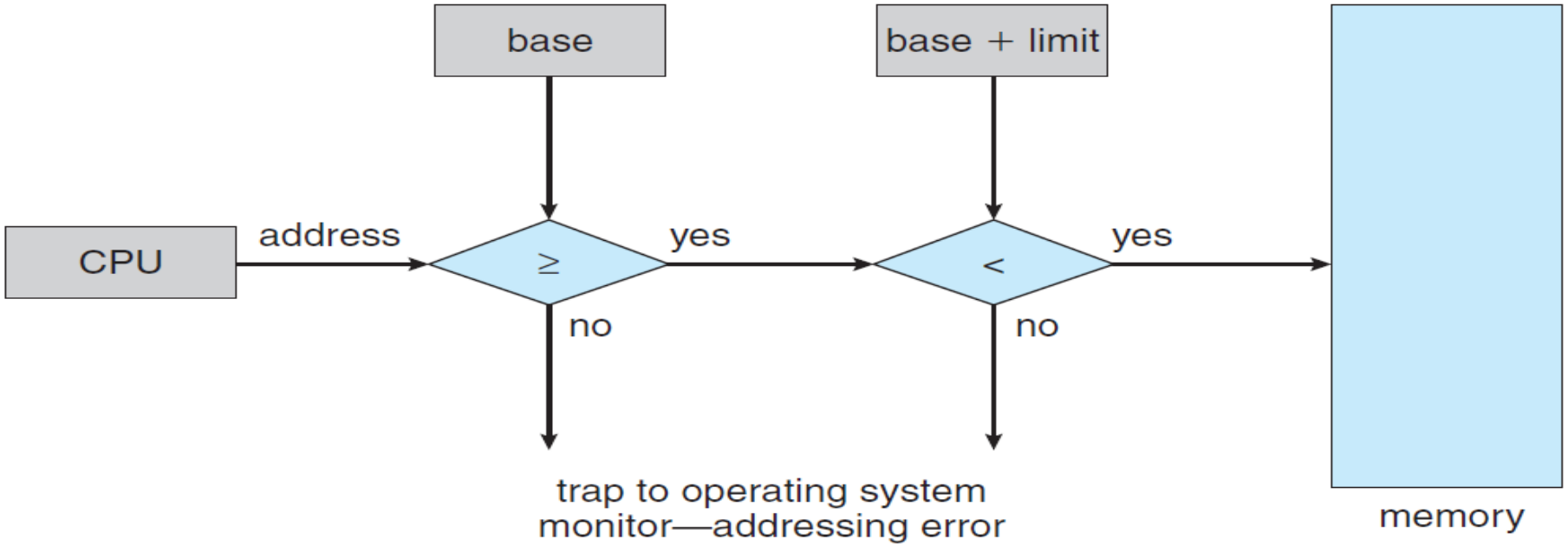
- We also must **ensure correct operation** by
  - **protecting the operating system** from user processes
  - ***protecting*** **user processes** from one another.
- Protection must be provided by the h/w.
- To do this, we need the ability to determine the range of **legal addresses** that the process may access.
- **Base** and **limit** registers could be used for this
- 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,

Base register holds the **smallest legal physical memory** address of a process; the **limit register** specifies the **size of the process**



Process1 can legally access all addresses from **300040** through **420939 (300040 + 120900)**

- Protection of memory space is accomplished by having the CPU hardware **compare every address generated** in user mode with the **registers**



- Any attempt by a user process 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**

Kernel mode and User mode

- This scheme allows the OS to change the value of the registers but prevents user programs from changing the registers' contents.
- The operating system, executing **in kernel mode,** is given **unrestricted access** to both operating-system memory and users' memory.
- This provision allows the operating system
  - to **load users' programs** into users' memory,
  - to **dump out those programs in case of errors**,
  - to access and modify parameters of **system calls**
  - to **perform I/O** to and from user memory, and to provide many other services

Address Binding

- Usually, a **program resides on a disk** as a **binary executable file**.
- To be executed, the program **must be brought into memory** and placed within a region.
- Depending on the memory management in use, the **process may be moved** between disk and memory during its execution (***swapping***).
- The **processes on the disk** that are waiting to be brought into memory for execution form the **job queue/ input queue.**
- The normal procedure is to **select one of the processes in the input queue (FIFO)** and to load that **process into memory**
- As the **process is executed**, it accesses instructions and data from memory
- Eventually, the **process terminates**, and its **memory space** is **released** and **declared as free**

- Most systems allow a user process to **reside in any part of the physical memory.**
- This approach **affects the addresses** that the user program can use
- Address binding means mapping computer instructions and data to locations in RAM
- The address space of a process affects the address binding process

```c
void swap(int* xp, int* yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)

        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = { 5, 1, 4, 2, 8 };
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

```c
void swap(int* xp, int* yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)

        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = { 5, 1, 4, 2, 8 };
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

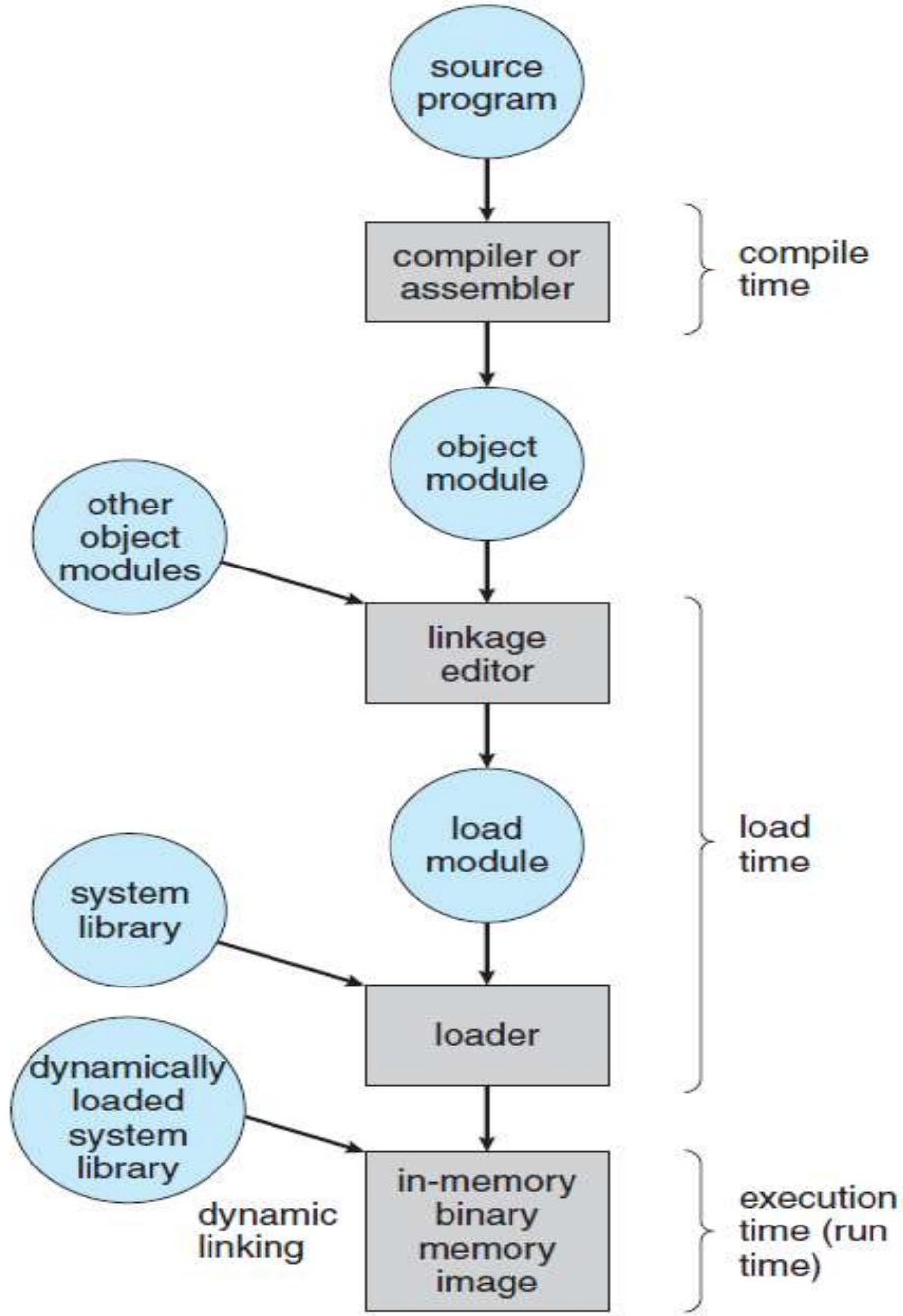
- In most cases, a user **program will go through several steps**— some of which maybe optional-—before being executed.
  - Program development
  - Compilation
  - Execution
- ***Addresses*** may be represented in different ways during these steps.
- Addresses in the source program are generally **symbolic** (such as *count).*
- *A* **compiler** will typically **bind** these symbolic addresses to **re-locatable addresses** (such as "14 bytes from the beginning of this module").
- The ***linkage editor or loader*** will in turn bind the re-locatable addresses to **absolute addresses** (such as 74014).
- Each binding is a **mapping** from one address space to another.

Types of address binding

- Classically, the binding of instructions and data to memory addresses can be done at any step along the way:
- **Compile time.** If compiler **knows at compile time** where the process will reside in memory, then absolute code can be generated.
- For example, if you know that a user process will reside **starting at location *R***, *then the generated* **compiler code will start at that location** and extend up from there.
- If, at some later time, the starting location changes, then it will be necessary to **recompile this code**.

- **Load time.** If it is **not known at compile time** where the process will reside in memory, then the compiler must generate **re-locatable code.**
- In this case ,final binding is **delayed until load time**.
- Compiler generates ***relocatable* code**
- compiler binds names to relative addresses (**offsets from starting address**)
- Loader converts relative addresses to physical addresses
- **No relocation** allowed **during execution**

- **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.**
- CPU generates **relative addresses**
- Relative addresses **bound to physical addresses at runtime** based on location of translated units
- **Special hardware** must be available for this scheme to work


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

Logical Vs Physical Address Space

- An address generated by the CPU is commonly referred to as a **logical address,** whereas an address seen by the memory unit—that is, the one loaded into the **memory-address register (MAR)** is commonly referred to as a **physical address.**
- The **compile-time** and **load-time** address-binding methods **generate identical logical and physical addresses.**
- However, the **execution-time address binding** scheme results in **different logical and physical addresses**.
- In this case, we usually refer to the logical address as a **virtual address.**

Logical addresses

```c
void swap(int* xp, int* yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)

        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = { 5, 1, 4, 2, 8 };
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

- We use ***logical address and virtual address*** interchangeably.
- 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.**
- The **run-time mapping from virtual to physical** addresses is done by a hardware device called the **memory-management unit (MMU).**
- We can choose from **many different methods** to accomplish such mapping

Benefit of maintaining logical and physical address space

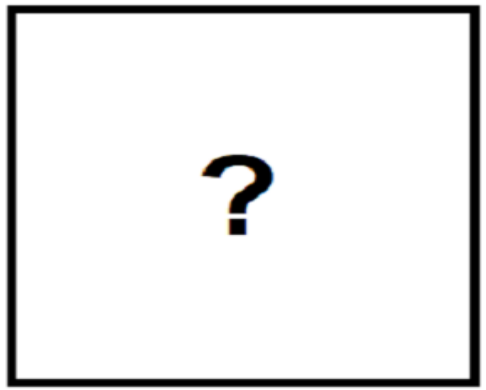
- Since processes are only addressed in terms of logical addresses until they needs to be executed, we can have a **larger logical address space** than the physical address space
- We can **selectively load portions of processes** into main memory instead of loading the entire processes
- That means we can have **more number of processes in the logical address space** than the capacity of the physical memory
- This results in better **memory utilization** & multiprogramming

Assume that the size of operating system is 50 bytes

P1	P2	P3	P4	P5
5 bytes	10 bytes	12 bytes	8 bytes	4 bytes

How much of Main memory is minimum required to load and execute theses processes ?

Main Memory



89 bytes are minimum needed (39 + 50)

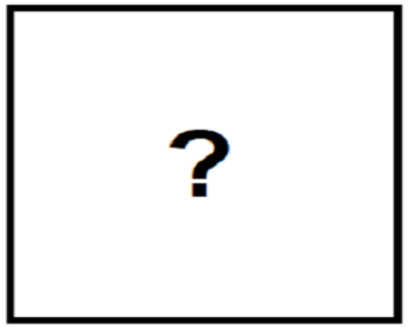
But in a system based dynamic binding and having virtual memory

Assume that the size of operating system is 50 bytes

P1	P2	P3	P4	P5
5 bytes	10 bytes	12 bytes	8 bytes	4 bytes

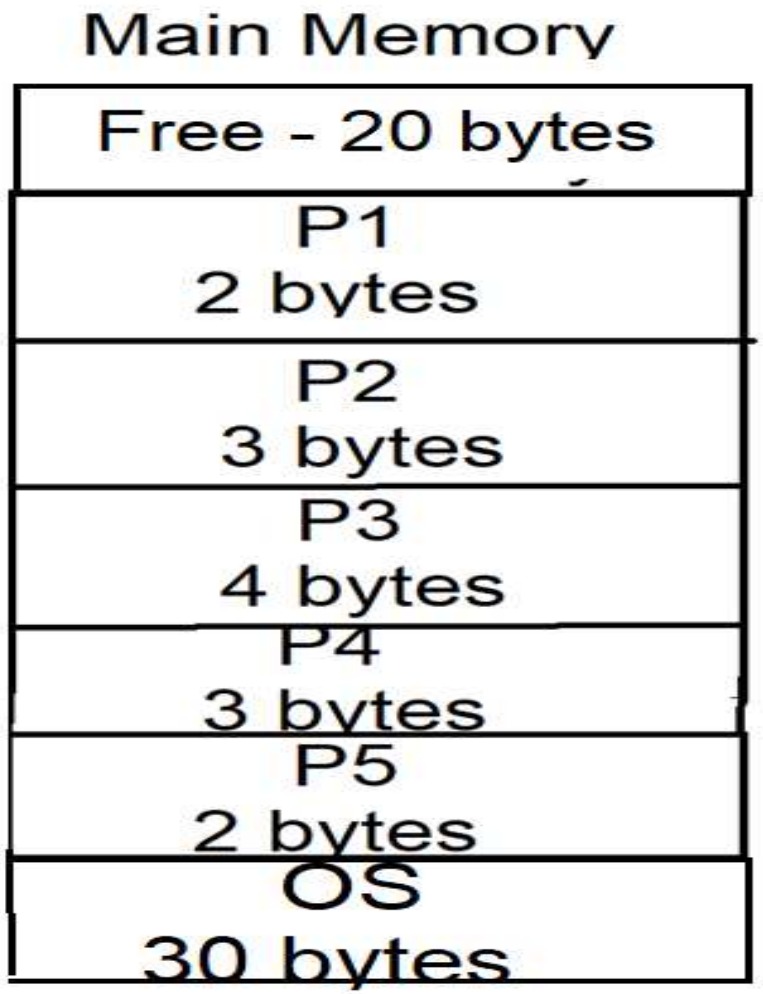
How much of Main memory is minimum required to load and execute theses processes ?

Main Memory



Less than 89 bytes suffiecient

P1	P2	P3	P4	P5
5 bytes	10 bytes	12 bytes	8 bytes	4 bytes



Logical address space is **89 bytes**, physical address space is **64 bytes**

Logical Address

- The **user program** never sees the ***real*** **physical addresses**.
- The program can create **a pointer to location** 346, store it in memory, manipulate it, and compare it with other addresses—all as the number 346.
- Only when it is used as a **memory address** (in an indirect load or store, perhaps) is it **relocated** relative to the base register.
- The **user program** deals with ***logical*** **addresses**.
- The **memory-mapping hardware** converts **logical addresses into physical addresses.**
- The concept of a logical address space that is bound to a separate physical address space is **central to proper memory management.**

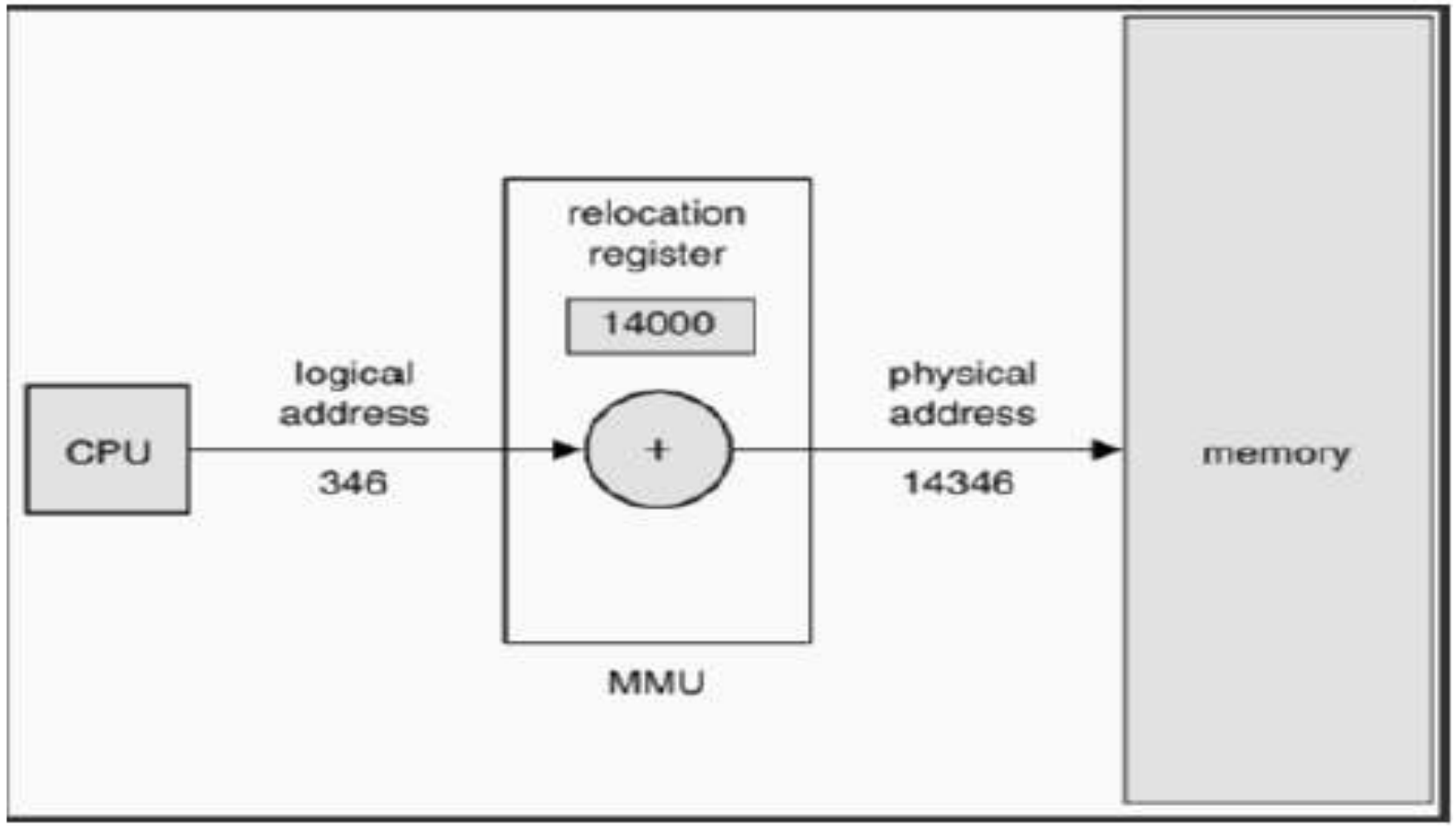
Address Mapping

Logical address to Physical address

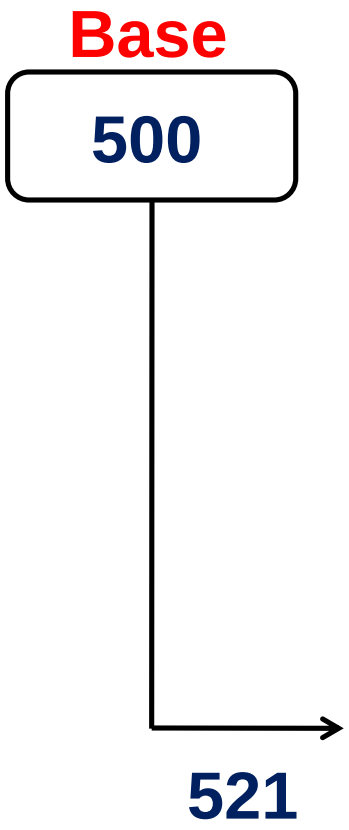
- Logical address space is **larger than** physical address space
- **CPU addresses memory** only in terms of its **logical space** (i.e. It treats all the processes to be completely present in main memory)
- So the **number of logical addresses** or virtual addresses is **more than the physical addresses.**
- For the previous example **logical address** space could be **up to 128 bytes (2^7)**
- But physical address space is only **2^6= 64 bytes**

Address mapping using Base reg

- For the time being, we illustrate this mapping with a **simple MMU scheme** that is a generalization of the base-register scheme.
- The base register is now called a **relocation register.**
- The **value in the relocation register is added** to **every address** generated by a user process at the time the address 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 location14000**; an access to location **346** is mapped to location **14346.**



Kernel loads relocation register when scheduling a process




```c
void swap(int* xp, int* yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)

        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = { 5, 1, 4, 2, 8 };
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

521

Dynamic Loading

- To be executed, **the entire program** and **all data** of a process must be in **physical memory** for the process to execute.
- The size of a process is thus limited to the **size of available physical memory**.
- To obtain better memory-space utilization, we can use **dynamic loading.**
- With dynamic loading, **a routine is not loaded until it is called.**
- All routines are **kept on disk** in a **re-locatable load format**.

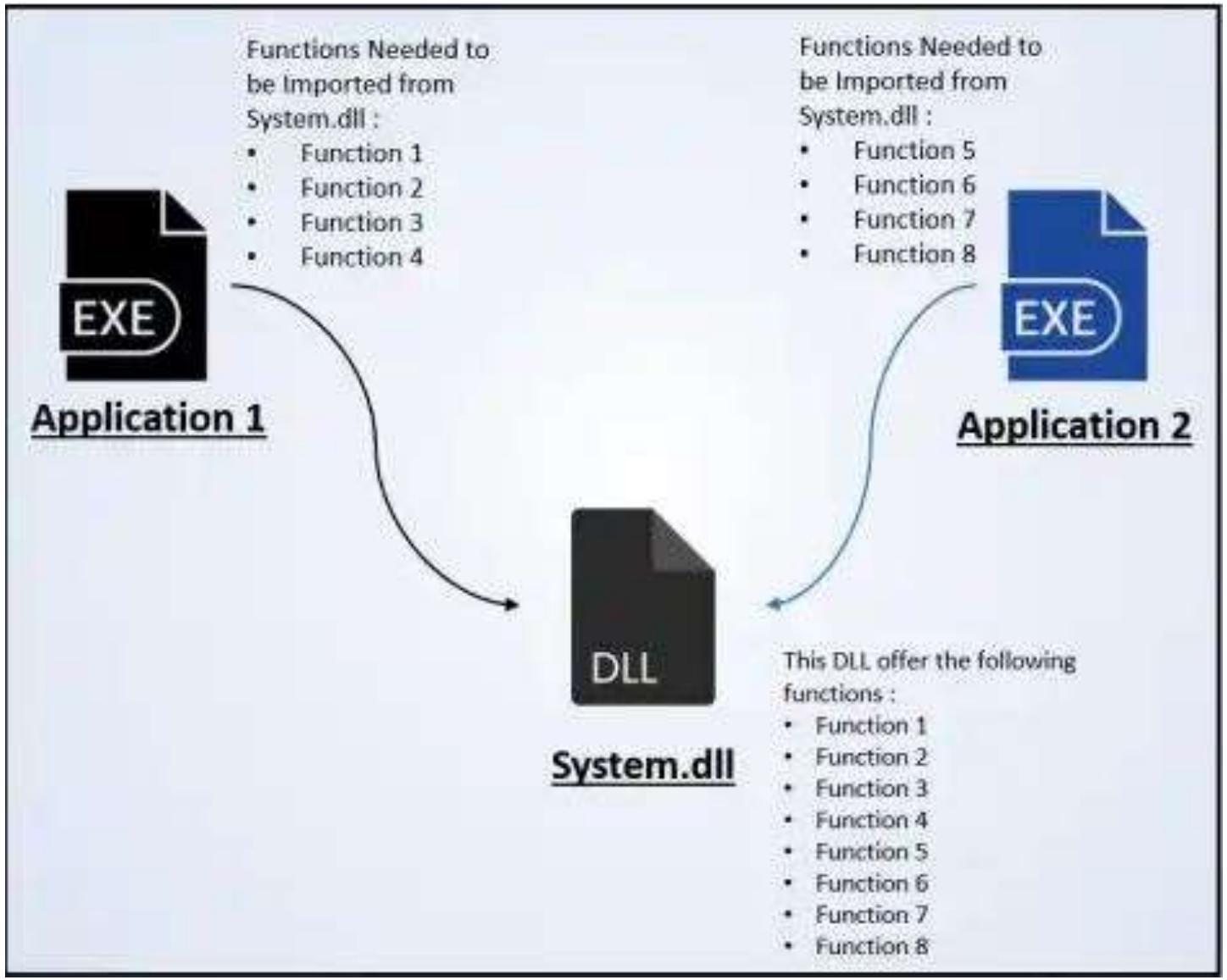
- Advantage of dynamic loading a routine is loaded **only when it is needed**.
- This method is particularly useful when large amounts of code are needed to handle **infrequently occurring cases,** such as **error routines**.
- Although the **total program size** may be large, the portion that is used (and hence loaded) **may be much smaller**.
- Dynamic loading **does not require special support** from the operating system.
- It is the **responsibility of programmers** to design their programs to take advantage of such a method.
- OS may help the programmer by providing **library routines** to implement dynamic loading.

Dynamic Linking and Shared Libraries

- **Dynamically linked libraries** are system libraries that are linked to user programs when the programs are run
- Some operating systems support only **static linking,** in which system language libraries are treated like any other object module and are combined by the loader into the binary program image.
- Under **dynamic linking**, Linking is **postponed until execution** time.
- This feature is usually used with system libraries, such as language subroutine libraries.

- Without this facility, each program on a system must include a **copy of its language library** (or at least the routines referenced by the program) in the executable image e.g ***DLLs.***
- With dynamic linking, a ***stub*** is included in the image **for each library routine reference**
- The **stub is a small piece of code** that indicates
  - **how to locate** the appropriate memory-resident library routine
  - or **how to load** the library if the routine is not already present.
- When the **stub is executed**, it checks to see whether the **needed routine is already in memory**. If not, the program loads the routine into memory
- Either way, the **stub replaces itself** with the **address of the routine** and executes the routine

- Thus, the **next time** that particular code segment is reached, the **library routine is executed directly**, incurring no cost for dynamic linking.
- Under this scheme, all processes that use a language library execute **only one copy of the library code.**
- This feature can be extended to **library updates** (such as bug fixes).
- A library may be replaced by a **new version**, and all programs that reference the library will **automatically use** the new version.
- **Without dynamic linking**, all such programs would need to be relinked to gain access to the new library


Functions Needed to be Imported from System.dll :
• Function 1
• Function 2
• Function 3
• Function 4
Functions Needed to be Imported from System.dll :
• Function 5
• Function 6
• Function 7
• Function 8
EXE
Application 1
EXE
Application 2
DLL
System.dll
This DLL offer the following functions :
• Function 1
• Function 2
• Function 3
• Function 4
• Function 5
• Function 6
• Function 7
• Function 8

- **More than one version** of a library may be loaded into memory, and each program **uses its version information** to decide which copy of the library to use.
- This system is also known as **shared libraries.**
- Unlike dynamic loading, dynamic linking and shared libraries generally require **help from the operating system.**

Swapping

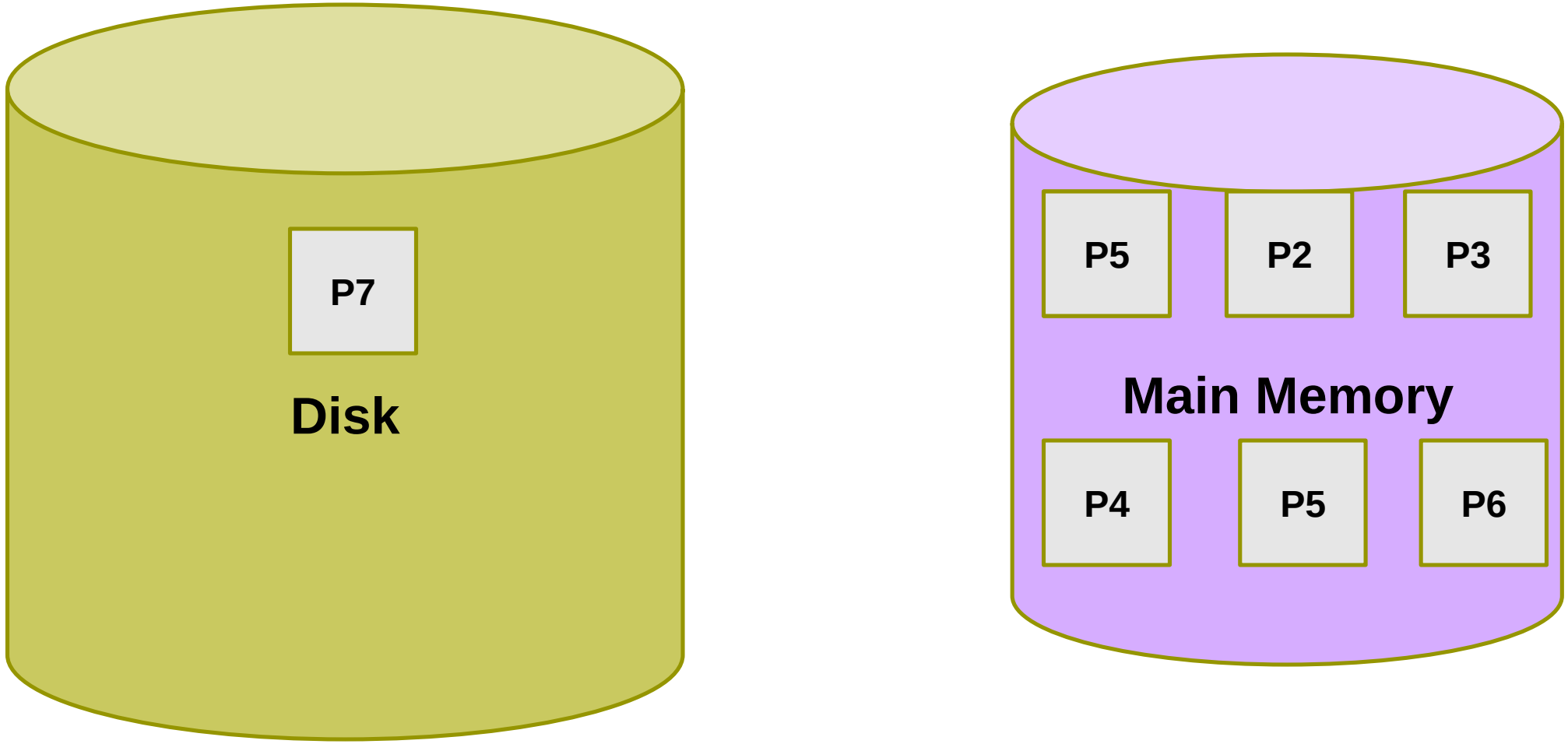
Swapping

- A process **must be in memory** to be **executed**.
- A process, however, can be swapped temporarily out of memory to a **backing store** and then brought back into memory for continued execution.
- Ideally, the memory manager can swap processes fast enough that **some processes will be in memory**, **ready to execute**.
- A variant of this swapping policy is used for **priority-based** scheduling algorithms.
- If a **higher-priority process** arrives and wants service, the **memory manager** can **swap out the lower-priority process** and then load and execute the higher-priority process
- This variant of swapping is sometimes called **roll out, roll in.**

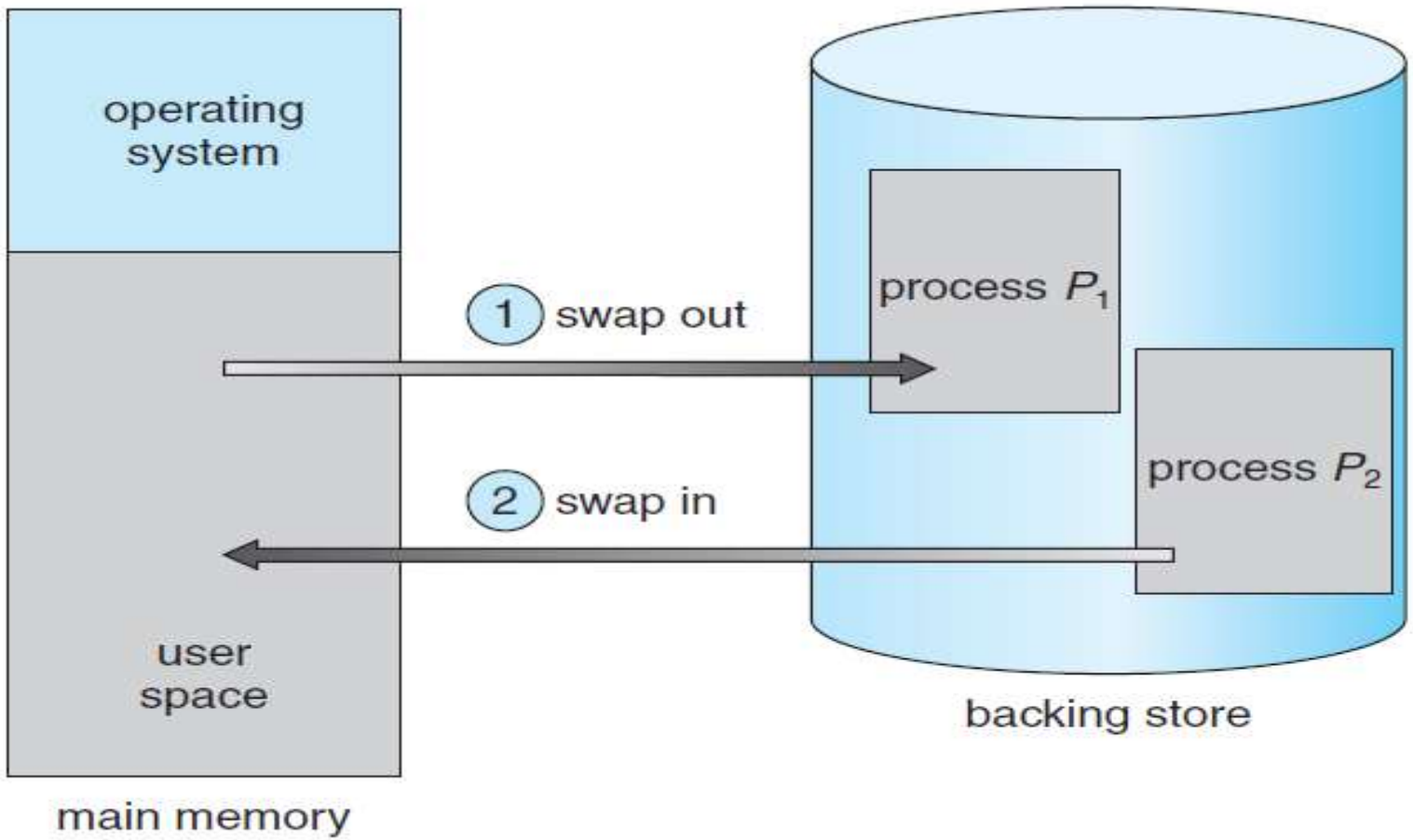
Possible Reasons for Swapping / Suspending a process

Reason	Comment
Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS Reason	OS suspects process of causing a problem.
Interactive User Request	e.g. debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time.
Parent Process Request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.

Swapping



P7 has to be loaded into main memory but it is full


operating system
user space
main memory
1 swap out
2 swap in
process P_1
process P_2
backing store

- Normally, a process that is swapped out will be swapped back into the **same memory space** it occupied previously.
- But if that **space is now occupied** by some other process then the swapped in process has to be loaded into some other address space.
- This option is dictated by the ***method of address binding***.
- If binding is done at **compile/assembly or load time**, then the process cannot be easily moved to a different location.
- If ***execution-time binding*** is being used, however, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.

Cost of context switching

- The context-switch time in such a swapping system is fairly high.
- To get an idea of the context-switch time, consider the **user process is 100 MB** in size and the backing store is a standard disk with a transfer rate of **50 MB per second**.
- The actual transfer of the 100-MB process to or from main memory

$$100\ \text{MB} / 50\ \text{MB per second} = 2\ \text{seconds}$$

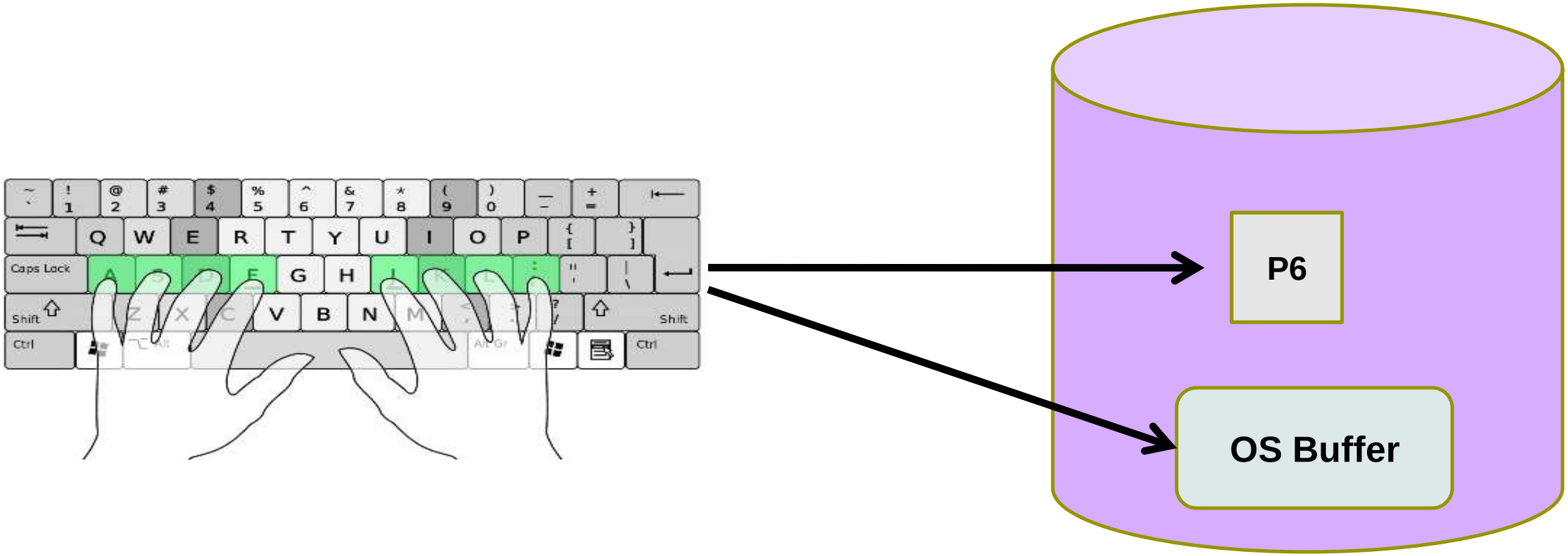
- Since we must swap both out and in, the total swap time is about **4 seconds**.

- Notice that the major part of the swap time is **transfer time**.
- The **total transfer time** is directly **proportional to** the **amount of memory** swapped.
- For efficient CPU utilization, we want the **execution time** for each process to be long relative to the **swap time.**
- Thus, in a **round-robin** CPU-scheduling algorithm, for example, the time quantum should be substantially **larger than 0.516** seconds.

Impact of I/O

- Swapping is constrained by other factors as well. If we want to swap a process, we must be sure that it is ***completely idle***.
- Of particular concern is any ***pending I/O***.
- A process may be waiting for an I/O operation when we want to swap that process to free up memory.
- If the I/O is ***asynchronously*** accessing the user memory for I/O buffers, then the **process cannot be swapped**.
- Assume that the I/O operation is queued because the **device is busy.**
- If we were to swap out process Pi and swap in process Po, the I/O operation might then attempt to use memory that now belongs to process *Pi.*

Impact of I/O in Swapping



- There are ***two main solutions*** to this problem:
  - **Never swap** a process with pending I/O, or
  - Execute I/O operations only into **operating-system buffers.**
  - Transfers between operating-system buffers and process memory then occur only **when the process is swapped in**.
- Note that this **double buffering itself** adds **overhead**.
- We now need to copy the data again, from kernel memory to user memory, before the user process can access it.

Swapping variations

- A modification of swapping is used in many versions of ***UNIX***.
- Swapping is **normally disabled** but will start if many processes are running and are using a ***threshold*** amount of memory.
- Swapping is **again halted** when the load on the system is normal.
- Another variation involves swapping **portions of processes**—rather than entire processes—to decrease swap time

Swapping on Mobile Systems

- Mobile systems typically **do not support swapping** in any form.
- Mobile devices generally use **flash memory** rather than **more spacious hard disks** as their persistent storage.
- The resulting **space constraint** is one reason why mobile operating-system designers avoid swapping
- Other reasons include the **limited number of writes** that flash memory can tolerate before it becomes unreliable (**wear out**).

- Instead of using swapping, when free memory falls below a **certain threshold**, **Apple’s iOS** asks **applications to voluntarily relinquish** allocated memory.
- **Android** does not support swapping and adopts a strategy similar to that used by iOS.
- It may **terminate a process** if insufficient free memory is available.
- However, before terminating a process, Android writes its **application state to** flash memory so that it can be quickly restarted.

Memory Partitioning
&
Memory Allocation

Sharing Memory

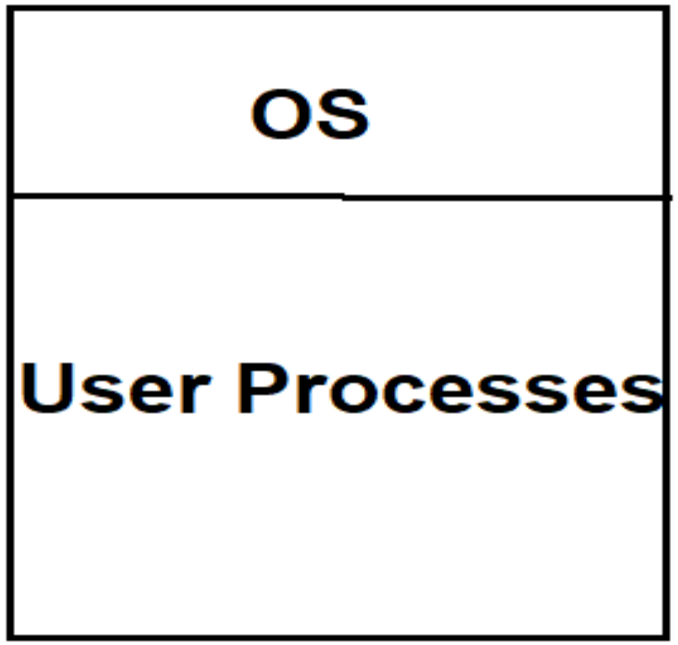
- The available main memory must be **distributed among all the user processes** as well as the **operating system**.
- The allocation must be **done fairly**, **efficiently** and **economically**
- Two things that are needed for sharing memory is
  - Memory partitioning
  - Memory allocation

Memory Partitioning

- The **main memory** must **accommodate** both the **operating system** and the various **user processes**.
- So we need to allocate main memory in the most efficient way possible
- 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**

- The major factor affecting this decision is the location of the ***interrupt vector***.
- Since the interrupt vector is often in ***low memory***, programmers usually place the **operating system in low memory** as well

Two partitioned memory



Contiguous Memory Allocation

- We usually want **several user processes** to reside in memory at the same time.
- We therefore need to consider **how to allocate available memory** to the processes that are in the **input queue(disk)** waiting to be brought into memory.
- In **contiguous memory allocation,** each process is contained in a single partition of memory that is contiguous to the partition containing the next process(**like arrays**)

Memory mapping & Protection

- Before discussing memory allocation further, we must discuss the issue of **memory protection.**
- A process in memory must be prevented from accessing another process's memory and operating system's memory.
- We can **prevent a process** from accessing **memory it does not own** by combining two ideas.
- If we have a system with a **relocation register** together with a **limit register** we accomplish our goal.

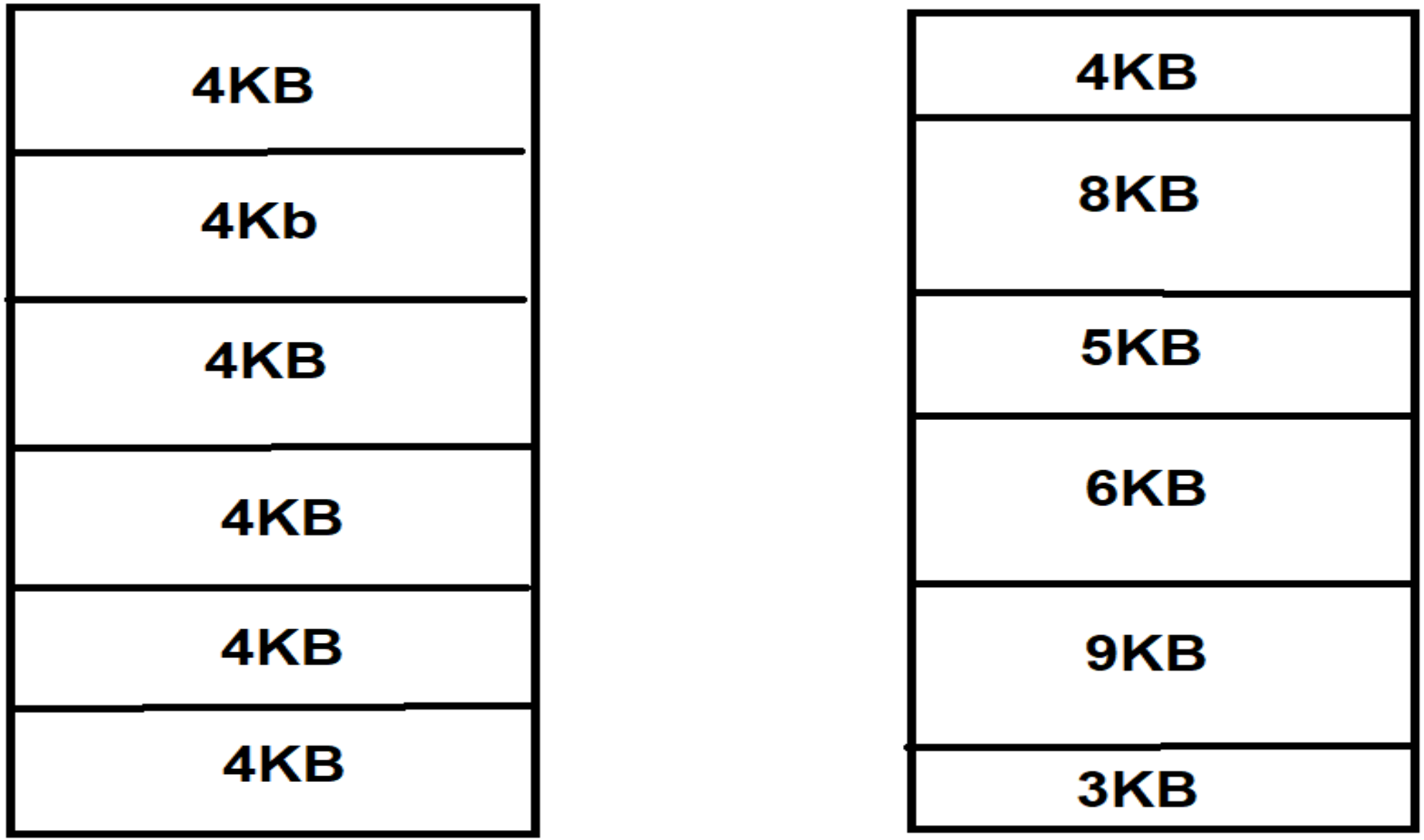
- The **relocation register** contains the value of the smallest physical address(**base address**); the limit register contains the range of logical addresses.
- Each logical address must be less than the **limit register.**
- The **MMU maps** the logical address dynamically by adding the value in the **relocation register**.
- This mapped address is sent to memory.
- When the CPU scheduler selects a process for execution, the ***dispatcher*** loads the ***relocation*** and ***limit*** registers with the correct values as part of the context switch

- The relocation register contains the value of the smallest physical address(base address); the limit register contains the range of logical addresses
- **For example, relocation = 100040 and limit = 74600**
- Logical address 102040 – Valid
- Logical address 130500 – Valid
- Logical address 120040 – Valid
- Logical address 174641 - Invalid & trapped

- Because every address generated by the CPU is checked against these registers, we **can protect both the operating system and the other users'** programs and data from being modified by this running process.
- This scheme provides an effective way to allow the **operating system's size to change dynamically**.
- For example, the operating system contains code and buffer space for **device drivers**.
- If a **device driver** (or other operating-system service) is **not commonly used**, then we need not keep the code and data in memory, as we might be able to use that space for other purposes.
- Such code is sometimes called **transient** operating-system code; it comes and goes as needed.

Memory Allocation – Fixed partitioning

- One of the simplest methods for allocating memory is to divide memory into several ***fixed-sized partitions***.
- Each partition may contain **exactly one process**.
- Thus, the **degree of multiprogramming** is bound by the **number of partitions**.
- In this **multiple partitions** method, when a partition is free, a process is selected from the **input queue** and **is loaded into** the **free partition**.
- When the **process terminates**, the **partition** becomes available for **another process**
- This method was originally used by the IBM OS/360 operating system (called MFT); it is no longer in use.



Drawback - Internal Fragmentation

Variable Partitioning Scheme

- This is a generalization of the fixed-partition scheme (called **MVT**).
- The operating system **keeps a table** indicating which parts of memory are **available** and which are **occupied**
- Initially, all the memory is available for user processes and is considered one large block of available memory, a **hole.**
- Eventually memory will contain a **set of holes** of various sizes

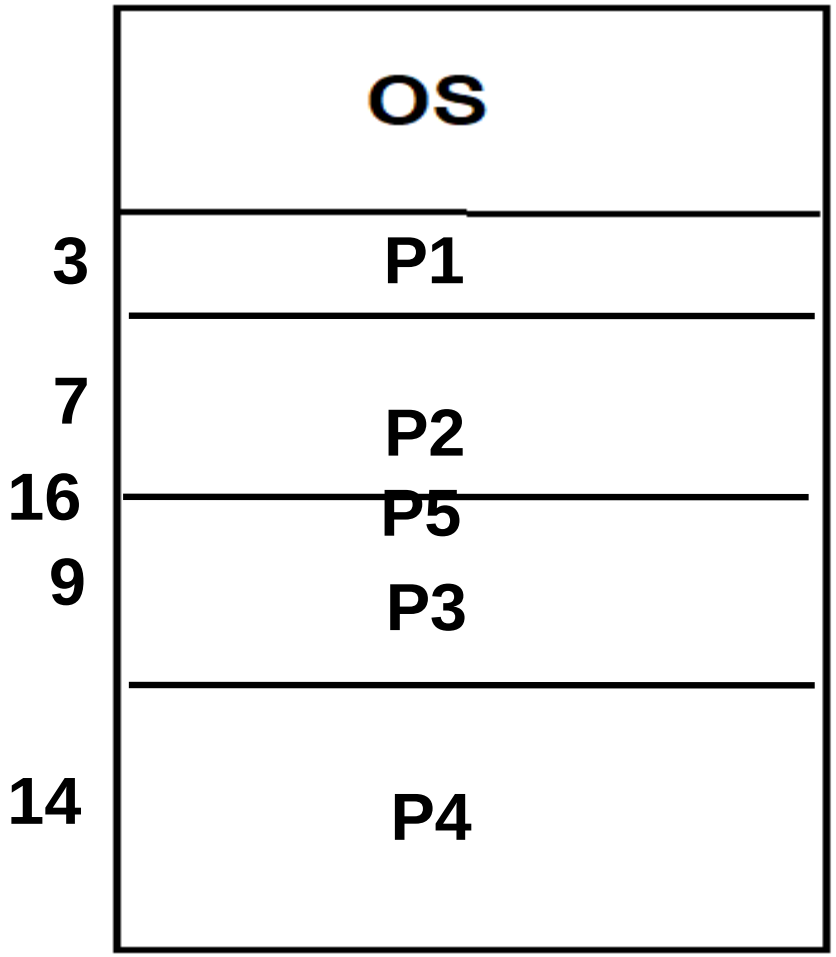
- As processes enter the system, they are put into an **input queue**
- Then OS **search for a hole large enough** for this process and **allocate it memory as required** by the process by partitioning the hole into two.
- If the hole is too large, it is split into two parts.
- When a process is allocated space, it is loaded into memory, and it can then compete for CPU time.
- When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

- At any given time, then, we have a **list of available block** sizes and an input queue.
- The operating system can order the input queue according to a scheduling algorithm.
- Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied—that is, no available block of memory (or hole) is large enough to hold that process.
- In general, as mentioned, the memory blocks available comprise a ***set of*** holes of various sizes scattered throughout memory.

- When a process terminates, it **releases its block of memory**, which is then placed back in the set of holes.
- If the new hole is adjacent to other holes, these **adjacent holes are merged** to form one larger hole

Input Queue (disk)

P1	P2	P3	P4	P5	P6	P7
3	7	9	14	16	21	25



Partition selection algorithms

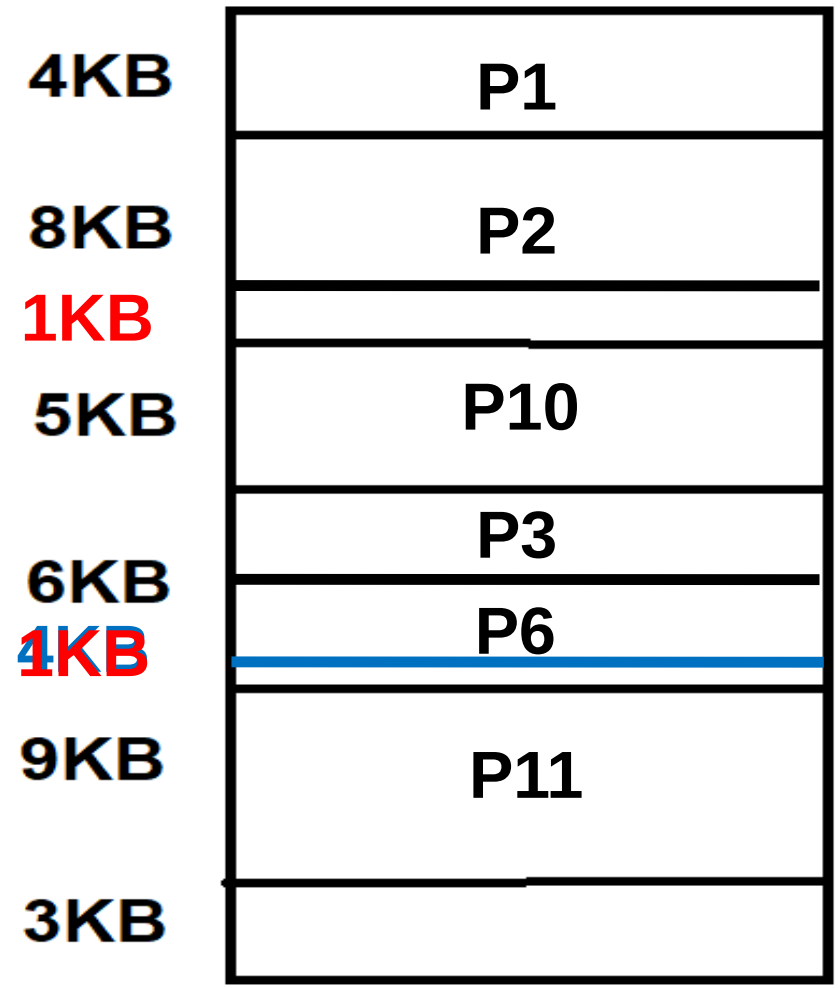
- This procedure is a particular instance of the general **dynamic storage allocation problem,** which concerns how to satisfy a request of size *n* from a list of free holes.
- There are many solutions to this problem. The **first-fit, best-fit,** and **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

- **First fit.** Allocate the ***first*** **hole** that is **big enough**. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best fit.** Allocate the *smallest* hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit.** Allocate the *largest* hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

- **Question:** Given the list of available partitions and processes to be allocated, identify the allocation scheme that generates more external fragments (assume 1 KB as a fragment)

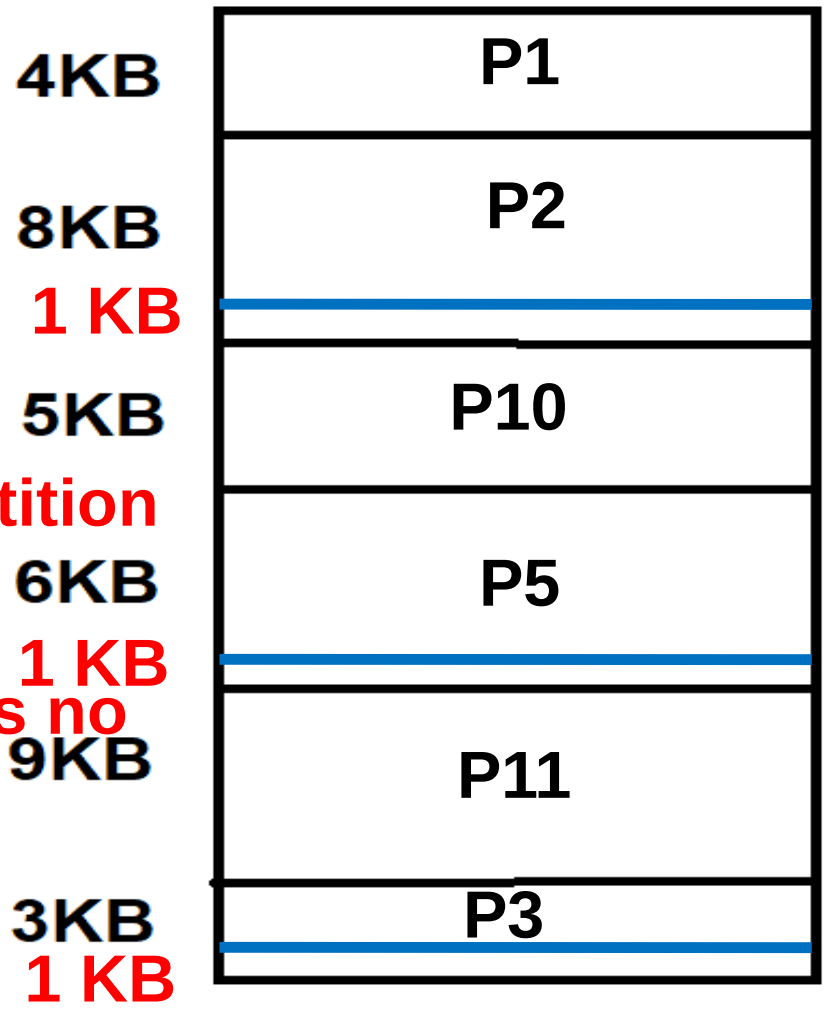
First Fit

P1	P2	P3	P4	P5	P6	P7	P8
4	7	2	8	5	3	7	6



Best Fit

P1	P2	P3	P4	P5	P6	P7	P8
4	7	2	8	5	3	7	6

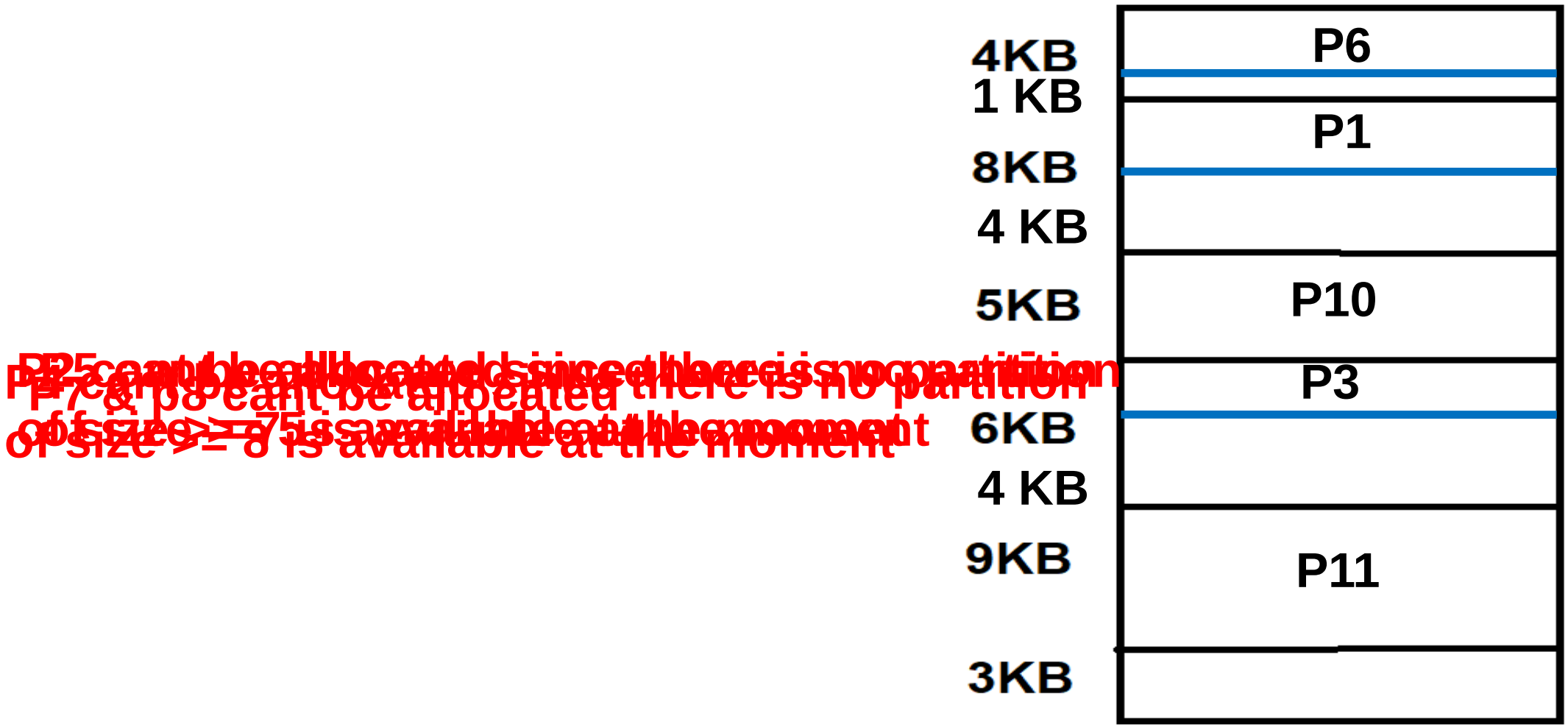


P4 cant be allocated since there is no partition of size >= 8 is available at the moment

P6, P7, P8 cant be allocated since there is no Partition of required size available

Worst Fit

P1	P2	P3	P4	P5	P6	P7	P8
4	7	2	8	5	3	7	6



- Simulations have shown that both **first fit** and **best fit** are **better** than **worst fit** in terms of **decreasing time** and **storage utilization**
- First fit is generally faster

Fragmentation – Internal & External

- Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**
- As processes are loaded and removed from memory, the free memory space is broken into **little pieces.**
- External fragmentation exists when **there is enough total memory space** to satisfy a request, but the available spaces are **not contiguous**; storage is **fragmented into** a large number of **small holes**.
- In the worst case, we could have a block of free (or wasted) memory between every two processes.
- If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

- Depending on the total amount of memory storage and the average process size, **external fragmentation** may be a minor or a major problem.
- Statistical analysis of **first fit** reveals that, even with some optimization, **one-third of memory** may be unusable!
- This property is known as the ***50-percent rule***.
- Memory fragmentation can be **internal as well as external**
- Under ***fixed partitioning,*** the memory allocated to a process may be slightly larger than the requested memory – **Internal fragmentation.**

Solution for external fragmentation

- One solution to the problem of external fragmentation is **compaction.**
- The goal is to **shuffle the memory contents** so as to place **all free memory together** in one large block.
- Compaction is not possible **If relocation is static** and is done **at compile or load time**.
- It is possible **only if relocation is dynamic** and is **done at execution time**.
- If addresses are relocated dynamically, relocation requires only moving the program and data and then **changing the base register** to reflect the new base address

- When compaction is possible, we must **determine its cost.**
- The simplest compaction algorithm is to **move all processes toward one end of memory**; all holes move in the other direction, producing one **large hole** of available memory.
- This scheme can be **expensive**.
- Another possible solution to the external-fragmentation problem is to permit the **logical address space** of the processes to be **noncontiguous,** thus allowing a process to be allocated physical memory wherever such memory is available e.g **Paging**

Sr. No.	Key	Internal Fragmentation	External Fragmentation
1	Definition	When there is a difference between required memory space vs allotted memory space, problem is termed as Internal Fragmentation.	When there are small and non-contiguous memory blocks which cannot be assigned to any process, the problem is termed as External Fragmentation.
2	Memory Block Size	Internal Fragmentation occurs when allotted memory blocks are of fixed size.	External Fragmentation occurs when allotted memory blocks are of varying size.
3	Occurrence	Internal Fragmentation occurs when a process needs more space than the size of allotted memory block or use less space.	External Fragmentation occurs when a process is removed from the main memory.
4	Solution	Best Fit Block Search is the solution for internal fragmentation.	Compaction is the solution for external fragmentation.
5	Process	Internal Fragmentation occurs when Paging is employed.	External Fragmentation occurs when Segmentation is employed.

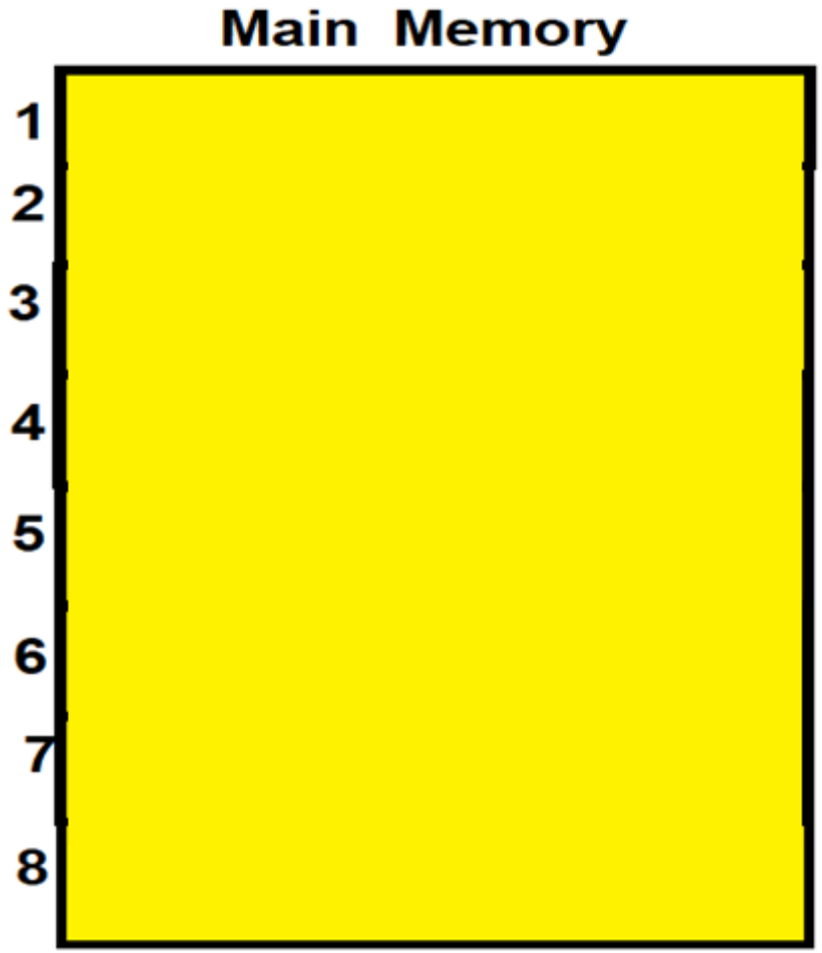
Paging

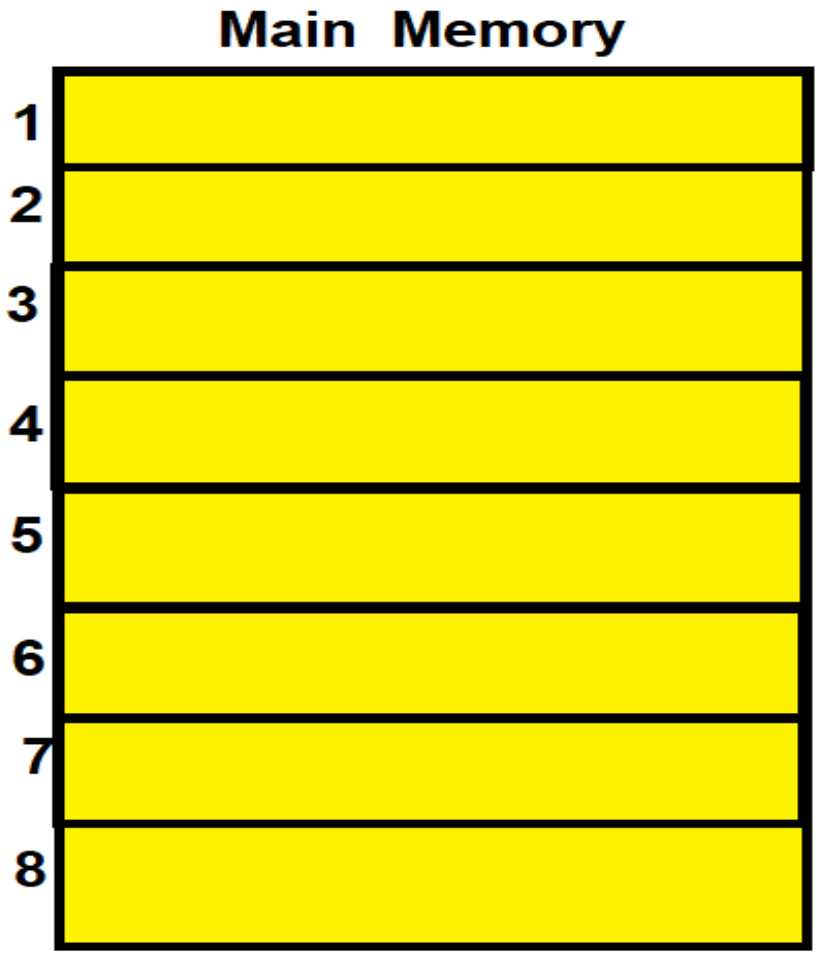
Paging

- Paging eliminates the need for processes to be handled as a whole while loading into memory.
- It allows **processes** to be partitioned into units called **pages.**
- The **main memory** is also partitioned into units called **frames**
- When a process needs to be loaded into memory sufficient number of frames need to be identified to load the pages of the process.
- The frames of a process **need not be contiguous**.
- Pages and frames are mapped with the help of **page tables**

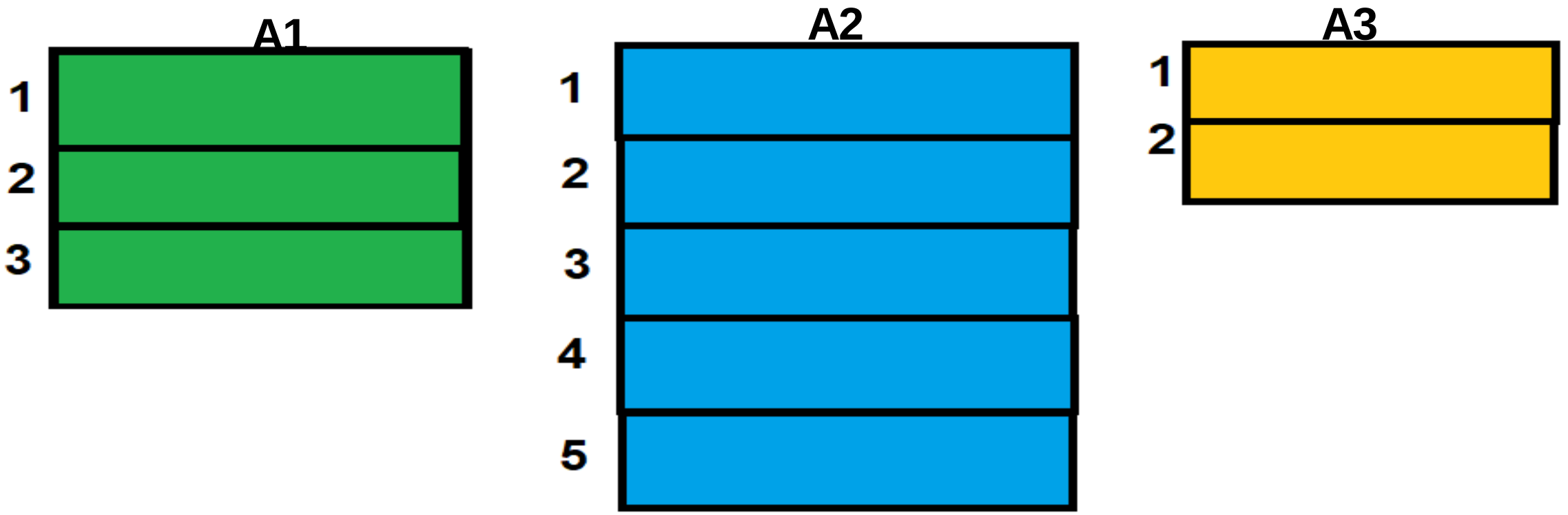
Example

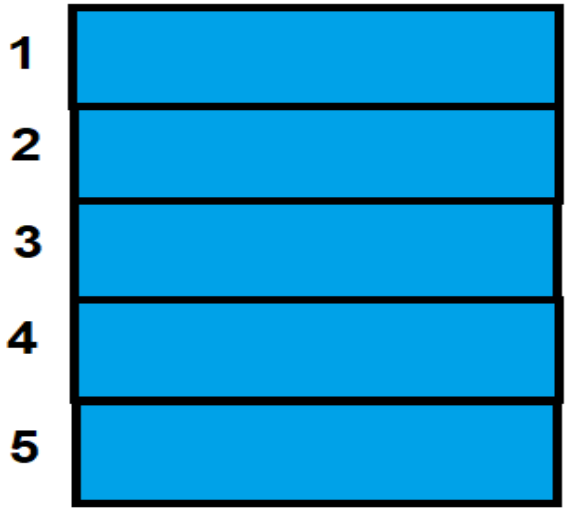
- For example, if the **main memory size is 8 KB** and **Frame size is 1 KB**. Hence, the main memory will be divided into the collection of **8 frames** of 1 KB each.





- There are 4 processes in the system **A1 – 3 KB, A2 – 5 KB, A3 – 2 KB each**. All the processes are divided into pages of **1 KB** each so that operating system can store one page in one frame.





Main Memory

1	Page 1
2	Page 2
3	Page 1
4	Page 3
5	Page 1
6	Page 2
7	Page 2
8	Page 3

A2's Page Table

1	Frame 1
2	Frame 7
3	Frame 4
4	NULL
5	NULL

Page Fault

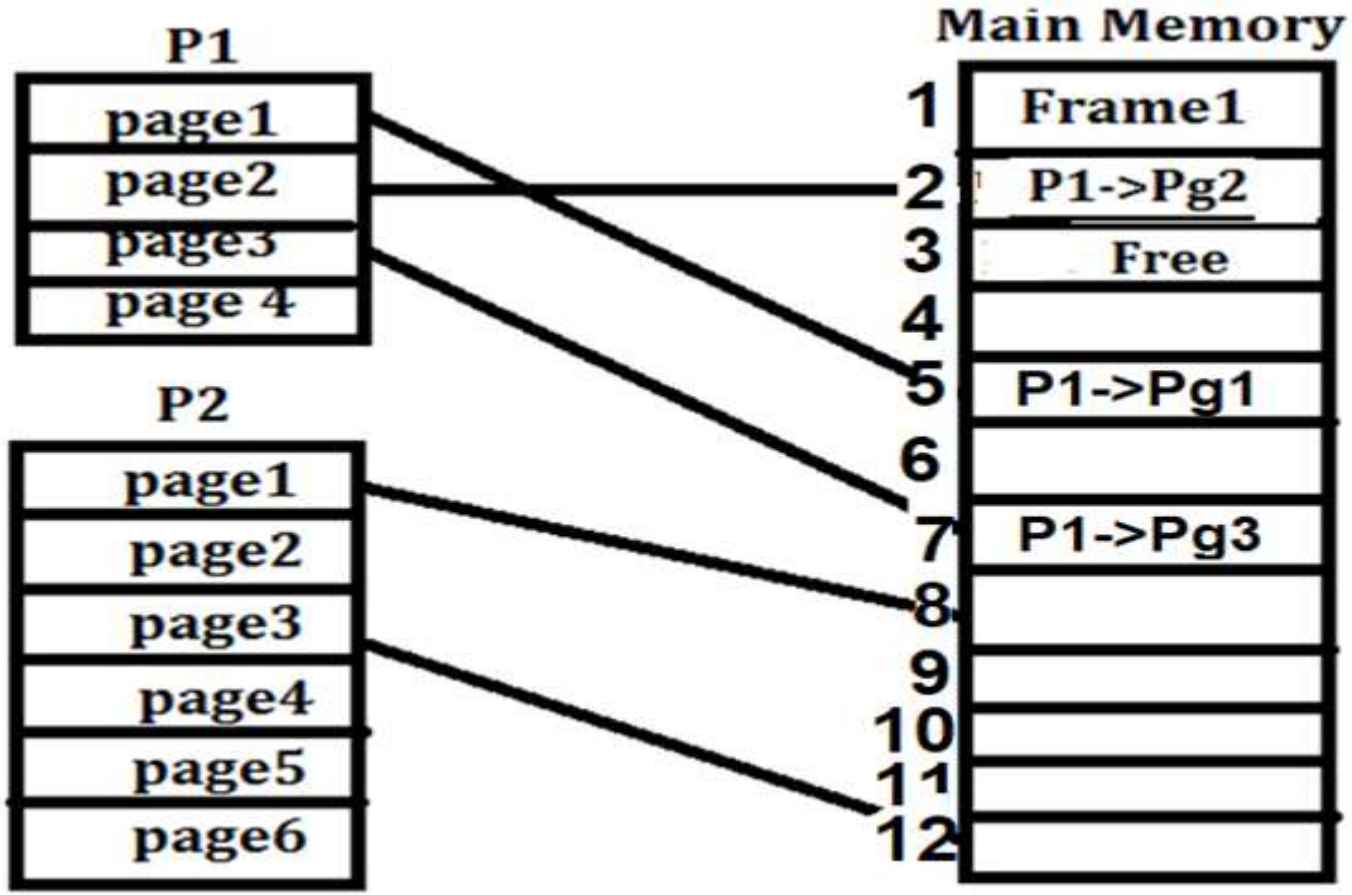
Benefits of paging

- To run a process, **only a portion of it** (**few pages**) need to be loaded.
- Remaining pages can be kept in the **virtual memory**.
- This results in
  - More number of processes can be loaded than the actual capacity of memory.
  - Pages that are rarely executed can be avoided from loading.
  - Pages that are already executed can be removed from memory.
  - External fragmentation is eliminated.

Total size of main memory is 600 MB & size of page/frame is 50 MB

Consider that size of Proces P1 is 200 MB and size of Process P2 is 300 MB

600/50 = 12 frames P1: 200/50 = 4 pages P2: 300/50 = 6 pages



Page table of Process1

Process1

1	5
2	2
3	7
4	X

Addresses in Paging

- Every address generated by the CPU is divided into two parts:
  - **page number (p)**
  - **page offset (d).**
- The page number is used as an index into a **page table.**
- The page table contains the **base address** (**frame no**) of each page in physical memory.
- **Offset** points to **one of the bytes** in **a page** that need to be accessed
- This base address is combined with the page offset to define the physical memory address that is sent to the memory unit **(base_address offset)**

Logical Memory size is 64 bytes (2^6)

&

Page size is 16 bytes (2^4)

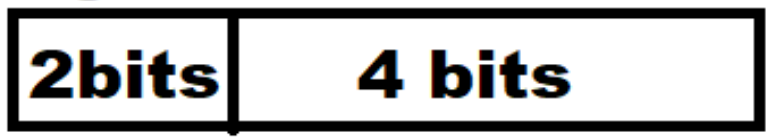
64/16 = 4 pages (2^2)

No. of address bits 6

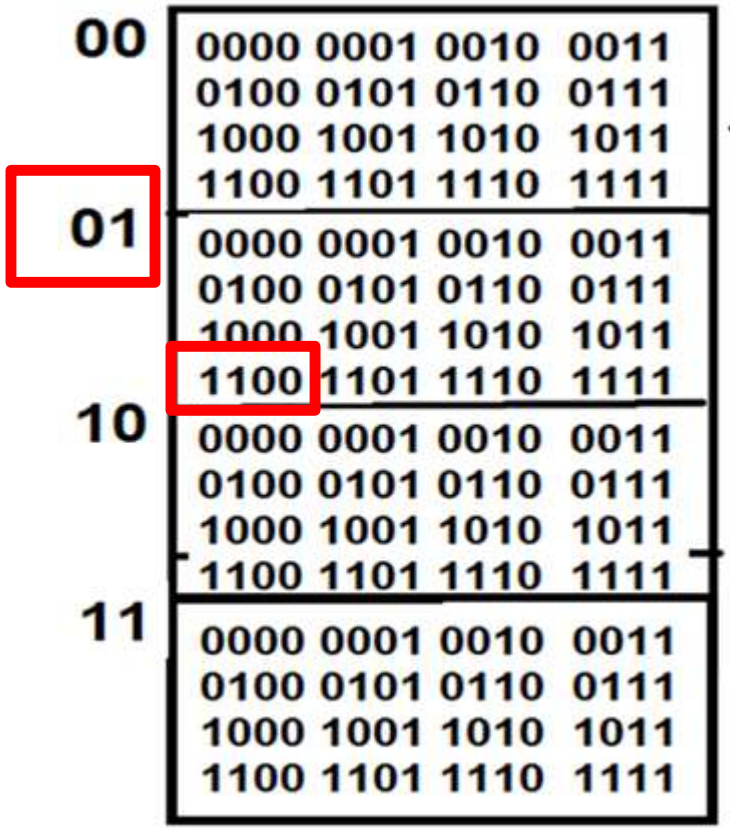
No of bits for page no 2

No of bits for offset 4

Logical address



Logical Memory size is 64 bytes (2^6)
&
Page size is 16 bytes (2^4)
64/16 = 4 pages (2^2)



Logical Address

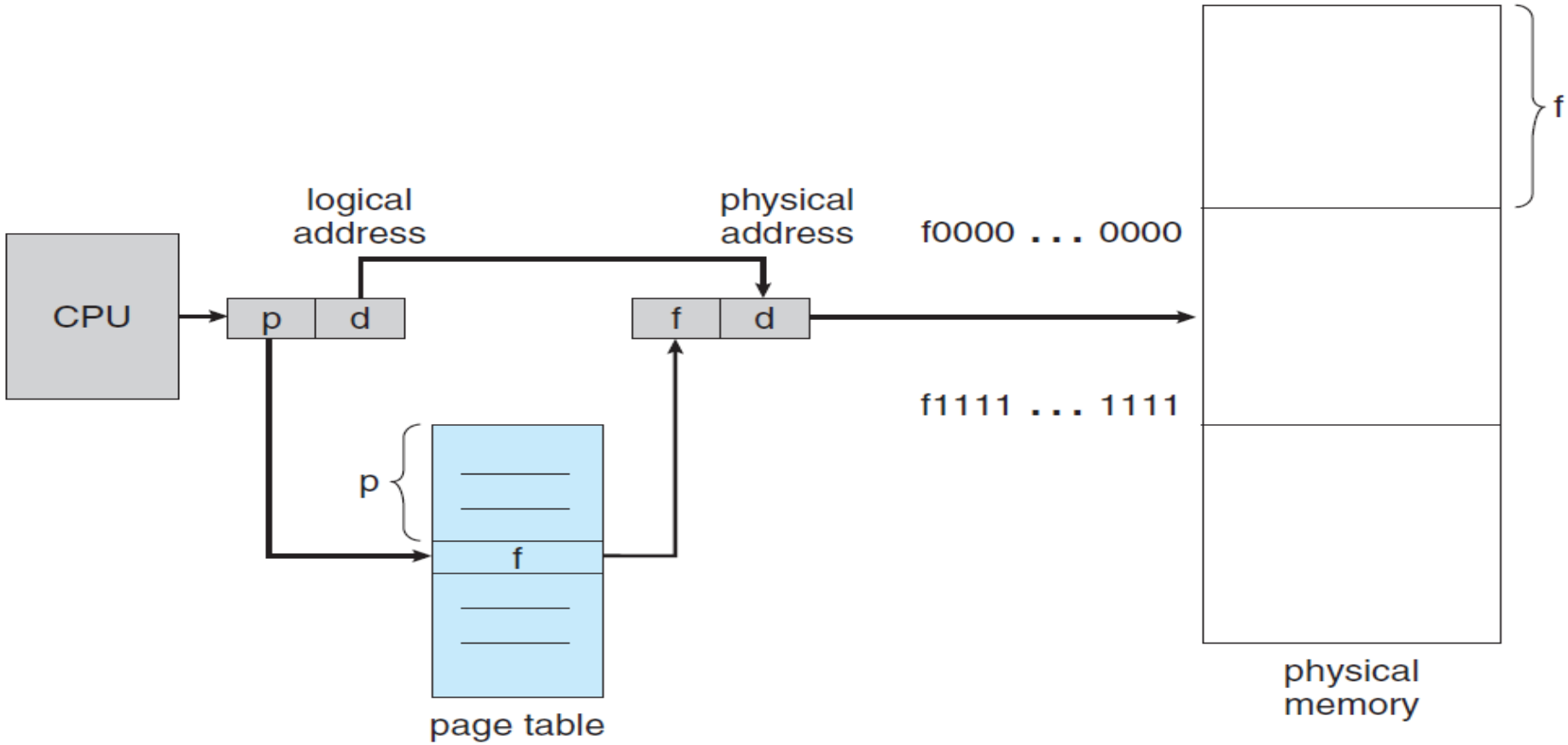
page no	offset
01	1100

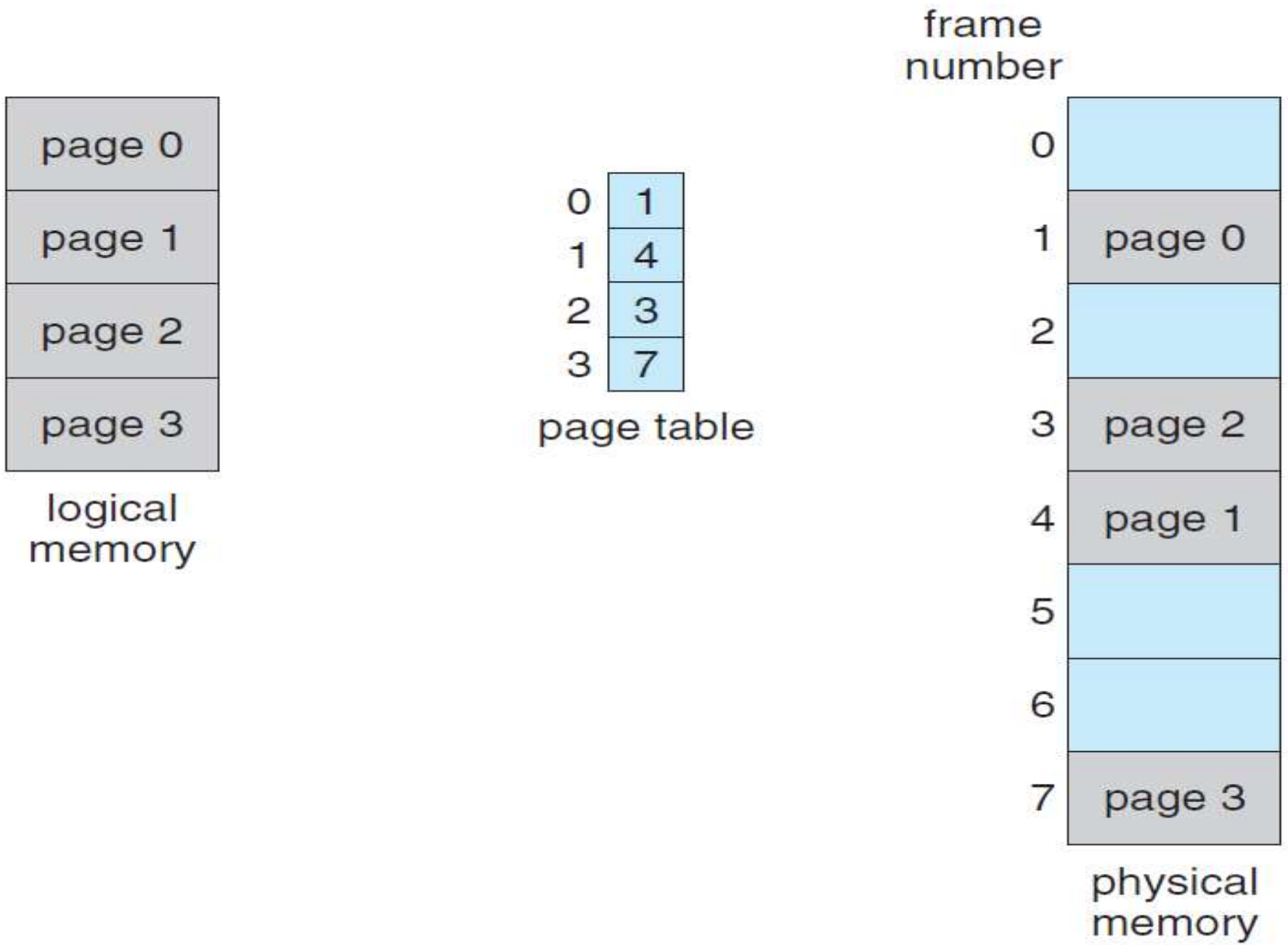
Physical Address

frame no	offset
10	1100

Page Table

0(00)	11
1(01)	10
2(10)	01
3(11)	XX

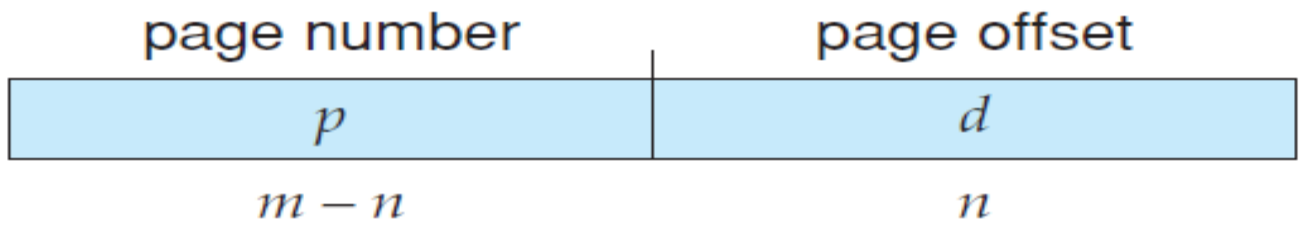

CPU
logical address
p
d
physical address
f
d
f0000 . . . 0000
f1111 . . . 1111
f
p
f
page table
physical memory


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

Page Size

- The **page size** (and the **frame size**) is defined by the hardware.
- The size of a page is typically a **power of 2**, varying between **512 bytes and 16 MB** per page, depending on the computer architecture.
- The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.

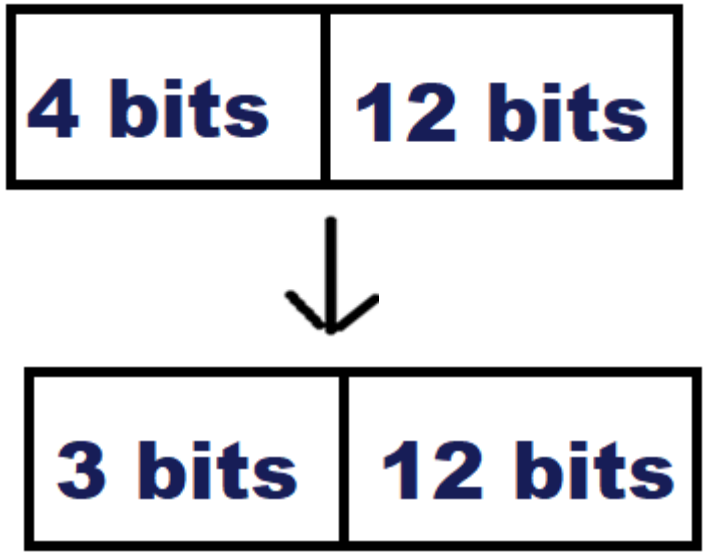
- If the size of **logical address space is 2^m** and a **page size is 2^n** addressing units (bytes or words), then the high-order $m - n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset.

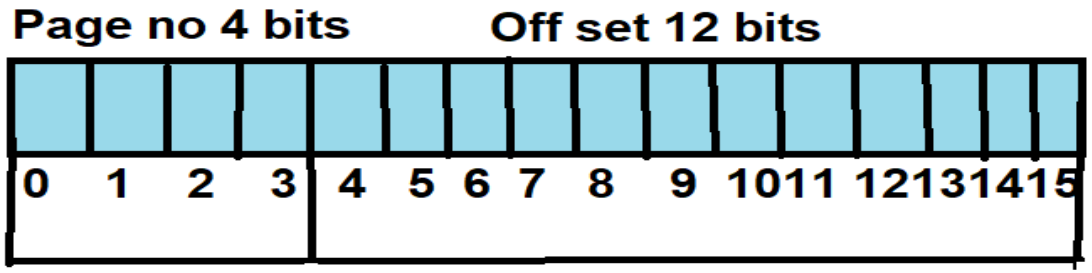


where *p* is an index into the page table and *d* is the displacement within the page.

- We have a computer that generates **16-bit addresses (64 KB)**, from **0 to 65535 (2^{16}).** These are **virtual/logical addresses**.
- So the **logical address space size** is 64KB
- Assume the **page size** is **4 KB**
- We could have **16 pages**
- **No of bits for representing page no. 4 (2^4) (64 / 4).**
- **No of bits** for representing **offset is 12 (4 KB=4096 bytes = 2^{12}).**
- This computer has only **32 KB** of physical memory.
- Then we have **8 frames**
- **No of bits** for representing **frame no is 3 (2^3)(32/4)**
- **No of bits** for representing **offset is 12 (4 KB)**

- Then no of address bits in **physical address** is **15 (32 KB = 32768 = 2^{15})**
- That means we could load **maximum 8** pages into memory at a time.
- Remaining pages will have to kept in the **virtual memory**



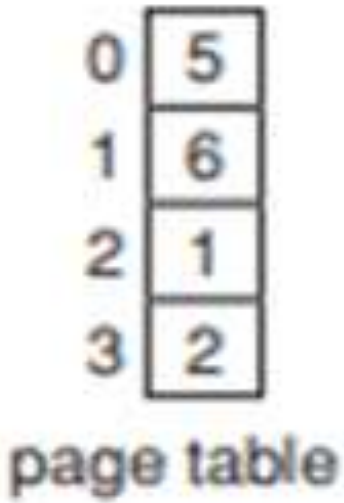


2^4 = 16 pages 2^{12} = 4 KB or 4096 bytes

Frame no 3 bits Offset 12 bits

2^3 = 8 frames 2^{12} = 4 KB

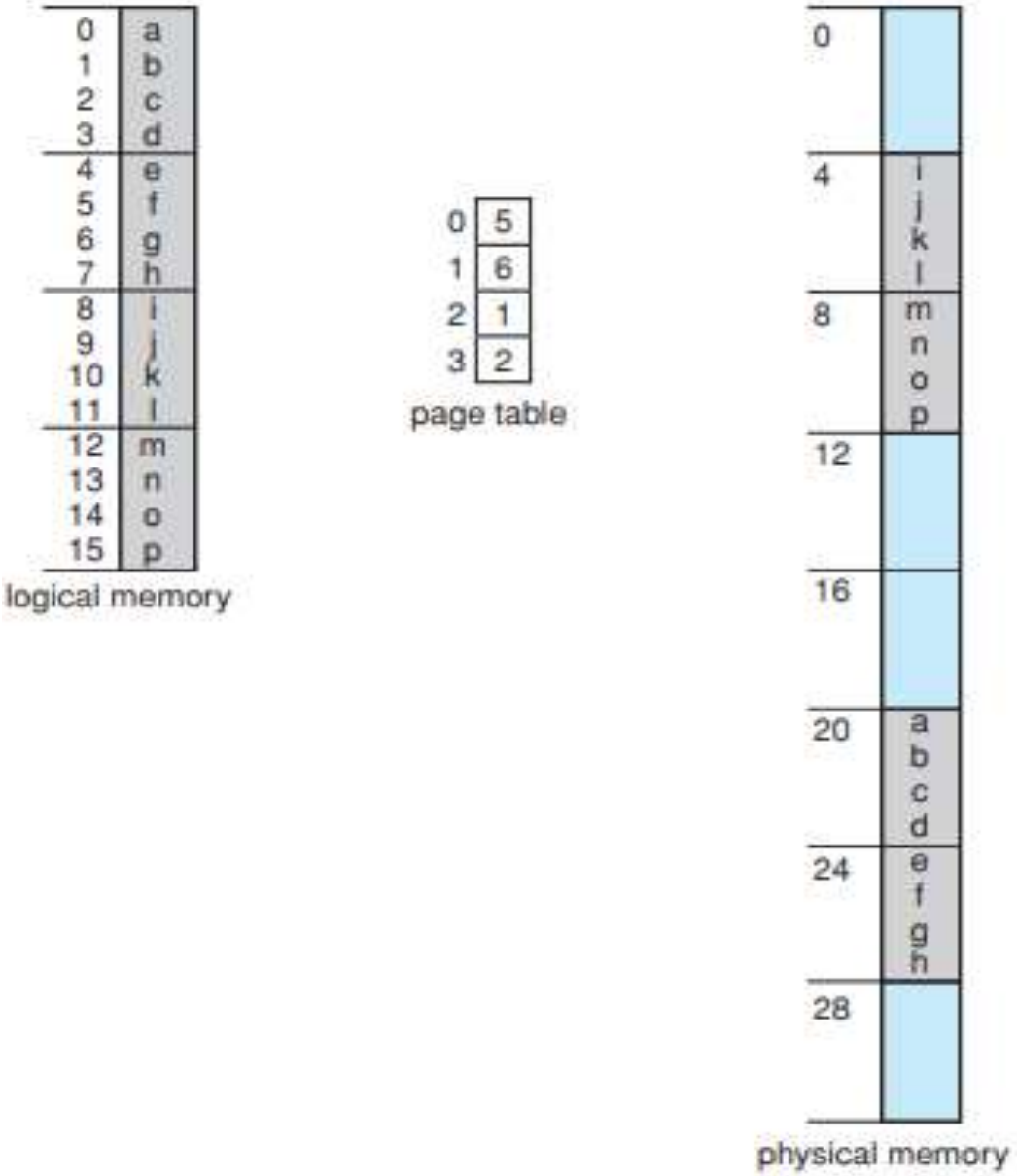
- Consider that a process consist of 4 pages of each 4 bytes. Main memory consist of 8 frames. Given that the logical address is 13, what would be the physical address as per the page table given below:



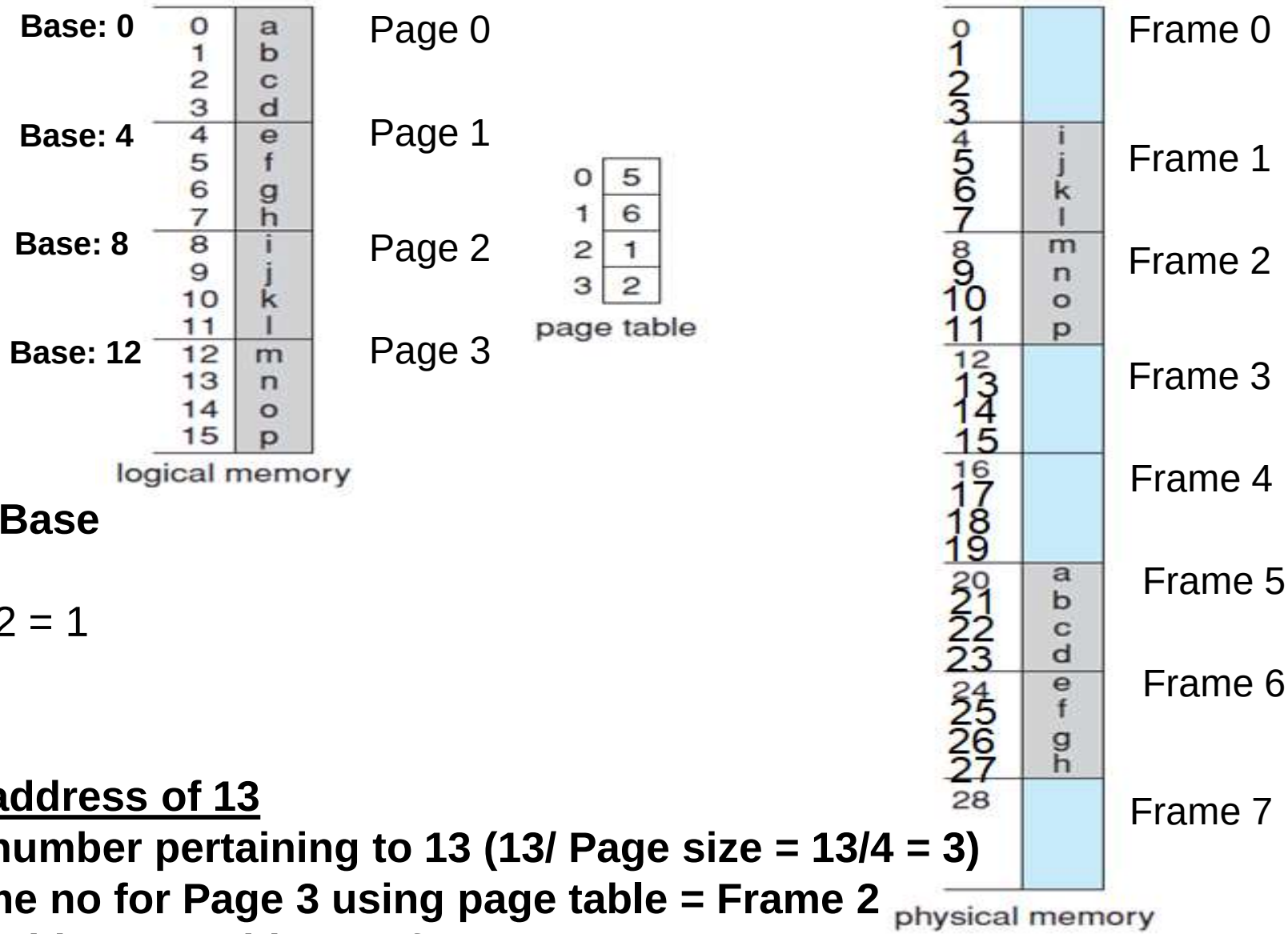
- **To know the physical address of 13**
- **Step 1. Find the page number pertaining to 13 (13/ Page size = 13/4 = 3)**
- **Step 2. Locate the frame no for Page 3 using page table = 2nd frame**
- **Step 3. Add the offset with Base address of Frame 2**
  - **Base address of Frame 2 = 2 X 4 = 8**
  - **Offset = Logical address – Base address of Page 3**
  - **Base address of Page 3 = 3 X 4= 12**
  - **Offset = 13 – 12 = 1**
- **Physical address = 8 +1 = 9**

Find the physical address for logical address 13

Process 1



Find the physical address for logical address 13



Offset = Address – Base

Examples

Address 13 = 13 – 12 = 1

<u>To know the physical address of 13</u>

Step 1. Find the page number pertaining to 13 (13/ Page size = 13/4 = 3)

Step 2. Locate the frame no for Page 3 using page table = Frame 2

Step 3. Add the offset with Base address of Frame 2 = 8+ 1 = 9

Example

- Using a page/frame size of 4 bytes and a physical memory of 32 bytes (8 frames).
- **Logical address 0** is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5.
- Thus, logical address 0 maps to physical address 20 (= (5 x 4) + 0).
- **Logical address 3** (page 0, offset 3) maps to physical address 23 *(*= (5x4) + 3).
- **Logical address 4** is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6x4) + 0).

Example

- Find the page no and offset of logical address 1502 when the address consist of 16 bits and page size is 1 KB.

- In this example, 16-bit addresses are used, and the page size is 1K = 1024 bytes.
- No of bits for Page no, Offset ? No of pages a process can have ?
- With a page size of 1K, an **offset field of 10 bits** is needed (2^{10} = 1024), leaving **6 bits for the page number**.
- Thus a program can consist of a maximum of 2^6 = 64 pages of 1K bytes each.

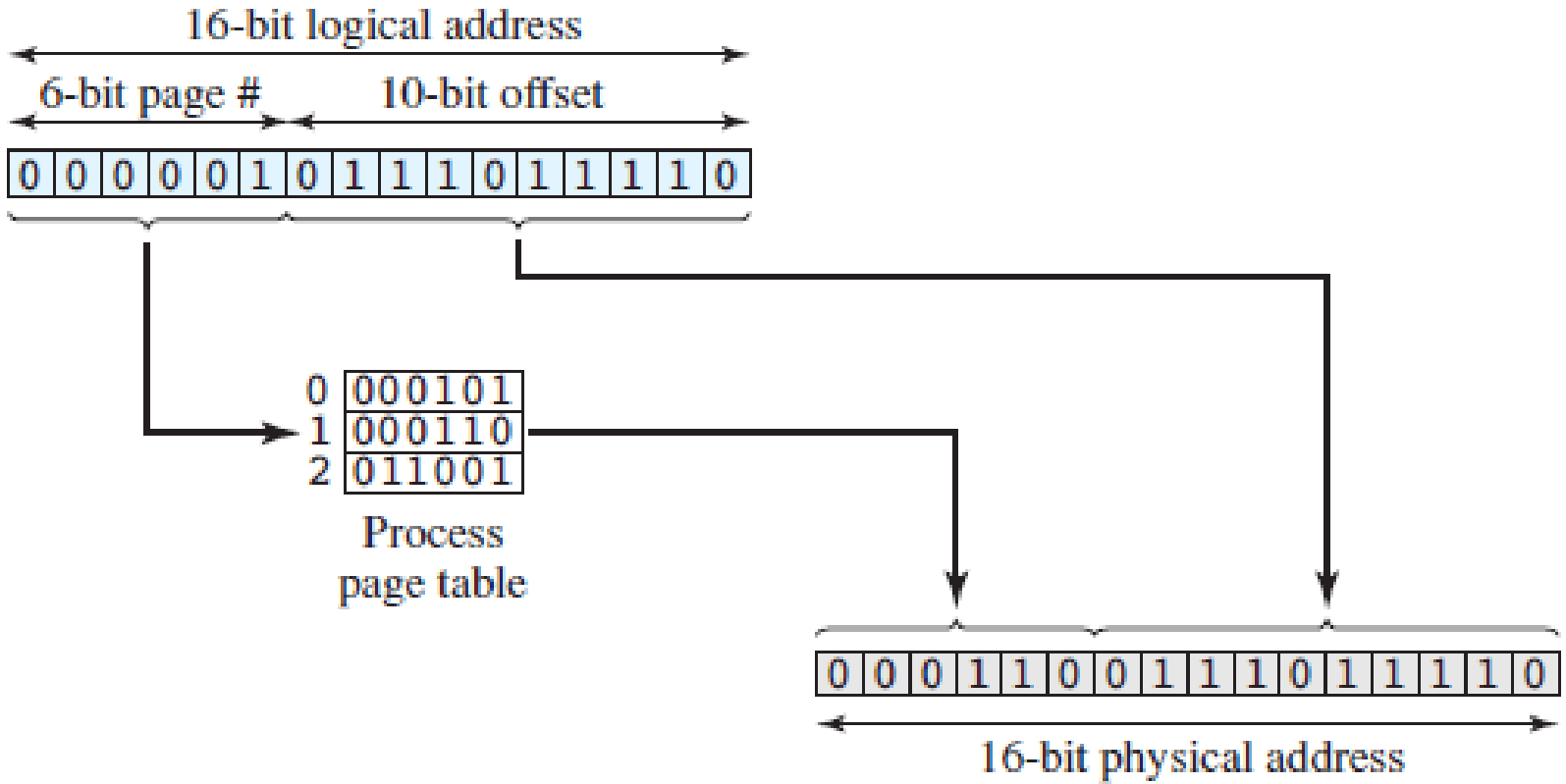
Example

- 1502, in binary form, is 0000010111011110

Page# = 1, Offset = 478

000001|0111011110

- 1502 corresponds to an offset of 478 (0111011110) on page 1 (000001),


16-bit logical address
6-bit page #
10-bit offset
0 0 0 0 0 1 0 1 1 1 0 1 1 1 1 0
0 000101
1 000110
2 011001
Process
page table
0 0 0 1 1 0 0 1 1 1 0 1 1 1 1 0
16-bit physical address

- The operating system maintains a **copy of the page table for each process**
- Paging itself is a form of **dynamic relocation-**Every **logical address** is bound by the paging hardware to some **physical address**.
- Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.
- When we use a paging scheme, we have **no external fragmentation**: *An* 1/ free frame can be allocated to a process that needs it.
- However, we may have some **internal fragmentation**.

- Given the physical memory of **60 MB** partitioned into frames of **4 MB (15 frames)**, compute the internal fragmentation if any(in KB) if the following processes are assigned frames:
- **Page table**
  - **P1: 50 MB / 4 MB = 13 frames**
  - Lat frames occupies **only 2 MB** so remaining **2 MB** becomes Internal fragmentation
  - **P2: 47 MB / 4 MB = 12 frames**
  - Lat frames occupies **only 3 MB** so remaining **1 MB** becomes Internal fragmentation

P1: 50 MB
1
5
2
7
10
9
X
X
X
X
X
X
X

p2: 47 MB
3
12
X
4
X
15
X
X
X
X
11
8

Choosing page size

- If the memory requirements of a process do not happen to coincide with page boundaries, the **last frame** allocated may not be completely full.
- For example, if **page size is 2,048 bytes**, a process of **72,766 bytes** would need **35 pages** and **1,086 bytes (72766-71680)**.
- But It would be allocated **36 frames**, resulting in an internal fragmentation of **2,048 — 1,086 = 962 bytes**.

- In the worst case, a process would need ***n* pages plus 1 byte**. It would be allocated, ***n* + 1 frames**, resulting in an internal fragmentation of almost an entire frame.
- If process size is independent of page size, we expect internal fragmentation to average one-half page per process.
- This consideration suggests that **small page sizes are desirable.**
- However, **overhead is involved** in each **page-table entry**, and this overhead is **reduced** as the **size of the pages increases**

- Also, **disk I/O** is more efficient when the number of data being **transferred is larger**.
- Generally, page sizes have **grown over time** as processes, data sets, and main memory have become larger.
- Today, pages typically are between **4 KB and 8 KB in size**, and some systems support even larger page sizes
- Some CPUs and kernels even support **multiple page sizes.**
- For instance, **Solaris** uses page **sizes of 8 KB and 4 MB**, depending on the data stored by the pages

Frame Table

- Since the OS is managing physical memory, it must be aware of the allocation details of physical memory— **which frames are allocated**, which frames are available, how many total frames there are, and so on.
- This information is generally kept in a **data structure called a frame table**.
- The frame table has **one entry for each physical page frame**, indicating whether the latter is **free or allocated** and, if it is allocated, to which page of which process or processes.
- Page table is for each process, but Frame table is for entire main memory

Hardware Support –Where to store page tables ?

- Each operating system has its **own methods** for **storing page tables**.
- Most allocate a **page table for each process**.
- A **pointer to the page table** is stored with the other register values (like the instruction counter) in the **process control block**.
- The hardware implementation of the page table can be done in several ways.
- **Option 1:** In the simplest case, the page table is implemented as a set of ***dedicated registers***.
- These registers should be built with very **high-speed logic** to make the **paging-address translation efficient**

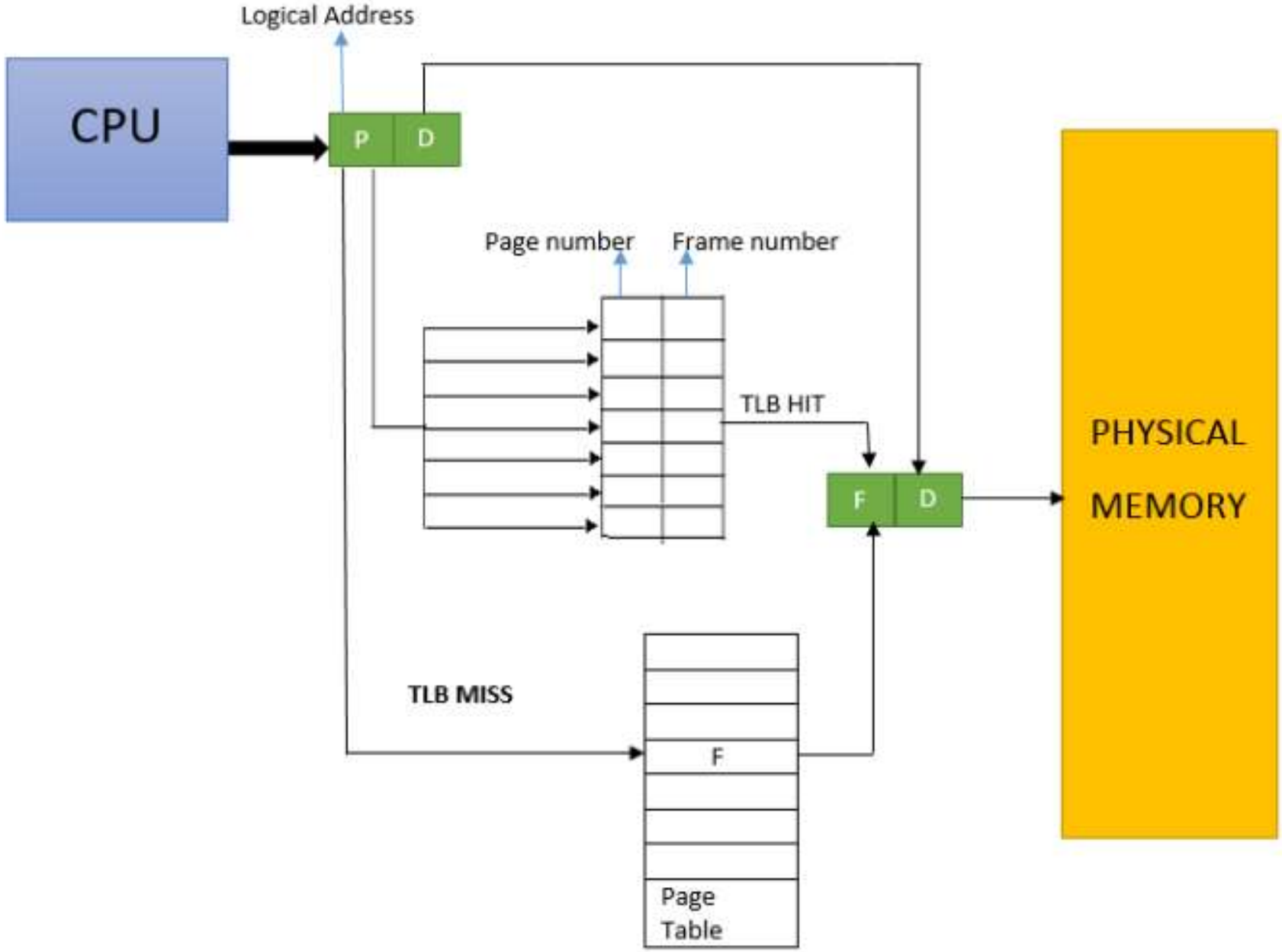
- The use of registers for the page table is satisfactory if the **page table is reasonably small**.
- **Option 2:** Rather, the page table is kept in ***main memory***, and a **page-table base register (PTBR**) points to the page table.
- Changing page tables requires changing only this one register, substantially reducing context-switch time.
- The problem with this approach is the **time required to access** a user memory location- **two memory accesses – one for page table, another for accessing frame**

Option 3: Translation Look-aside Buffer (TLB)

- The standard solution to this problem is to use a special, small, fast lookup hardware cache, called a ***translation look-aside buffer (TLB).***
- The TLB is **associative, high-speed memory**. Each entry in the TLB consists of two parts: **a key (or tag) and a value.**
- When the **associative memory** is presented with an item, the item is compared with ***all keys simultaneously***.
- If the item is found, the corresponding value field is returned.
- The **search is fast**; the hardware, however, is **expensive.**
- Typically, the number of entries in a **TLB is small**, often numbering between **64 and 1,024**.

- The TLB is used with page tables in the following way.
- The TLB contains only a **few of the page-table** entries.
- When a **logical address** is generated by the CPU, its **page number is presented to the TLB**. If the page number is found, its frame number is immediately available and is used to access memory (***TLB Hit***)
- If the page number is not in the TLB (known as a ***TLB miss***), a memory reference to the page table must be made.
- When the frame number is obtained, we can use it to access memory.
- In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.

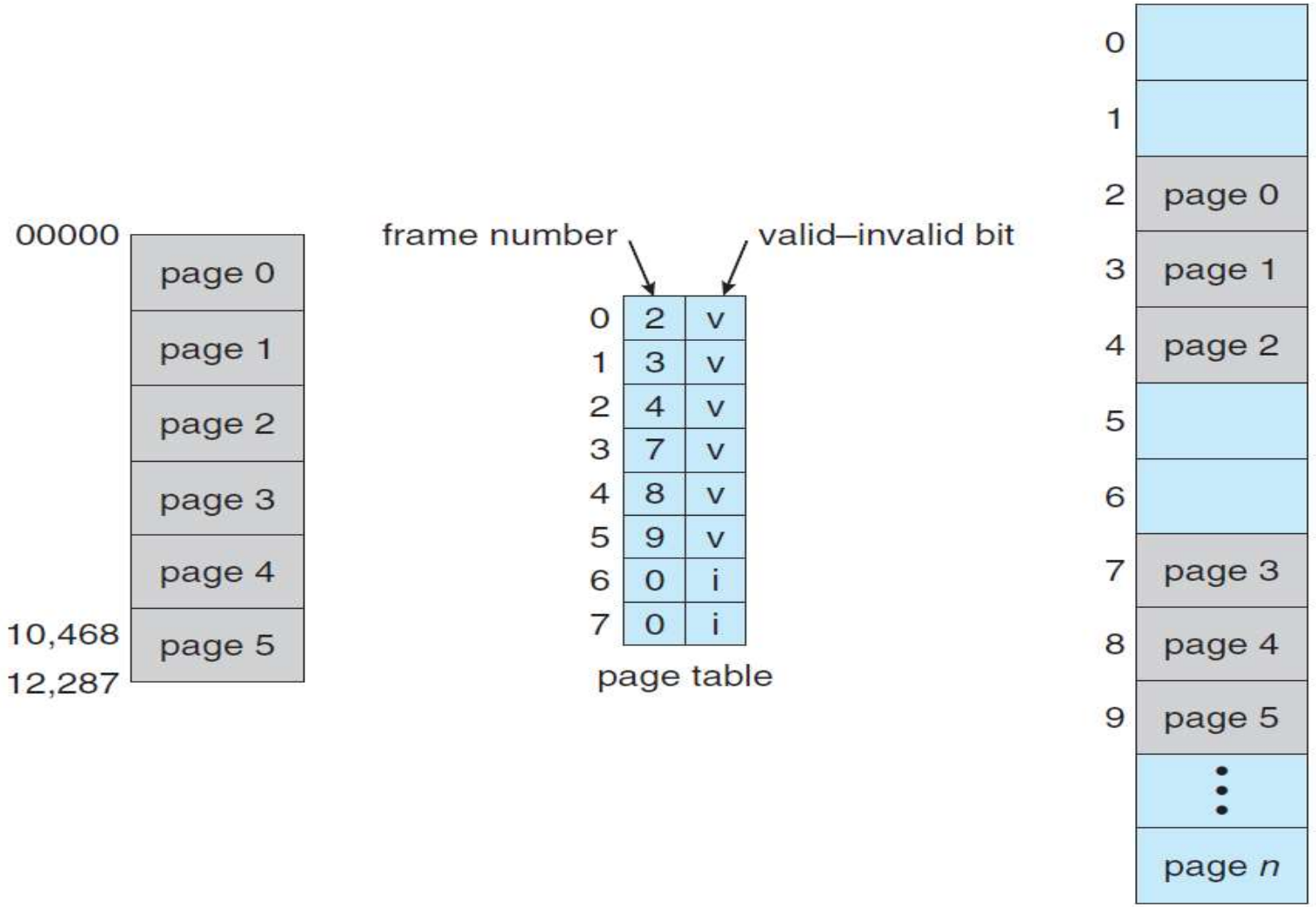
- If the TLB is already full of entries, the operating system must select one for **replacement**.
- Replacement policies range from least recently used (**LRU**) to **random**. Furthermore, some TLBs allow entries to be **wired down**, meaning that they **cannot be removed** from the TLB.
- Typically, TLB entries **for kernel code** are wired down.


Logical Address
CPU
P
D
Page number
Frame number
TLB HIT
F
D
PHYSICAL
MEMORY
TLB MISS
F
Page
Table

Protection

- Memory protection in a paged environment is accomplished by **protection bits** associated with each frame.
- Normally, these bits are kept in the page table.
- **One bit** can define a page to be **read-only**, **read-write**, or **execute-only** protection;
- One additional bit is generally attached to each entry in the page table: a **valid-invalid** bit.
- When this bit is set to **"valid,"** the byte is in the process's logical address space and is thus a **legal (or valid) page**.
- When the bit is set to "**invalid**,'" the bit is not in the process's logical address space.
- Illegal addresses are **trapped**.

- Suppose, for example, that in a system with a **14-bit address** space (**0 to 16383**), we have a program that should use only addresses **0 to 10468.**
- Given a page size of **2 KB**, we have the situation shown in Figure.
- No of frames required = 10468/1024 = 10.22 KB= 10.22 / 2= 5.11. So **6 frames** required.
- Addresses in pages **0, 1, 2, 3, 4, and 5** are mapped normally through the page table.
- Any attempt to generate an address in page **6 beyond the address range of the process** will find that the **valid–invalid bit** is set to **invalid**, and the computer will trap to the operating system (invalid page reference).


00000
page 0
page 1
page 2
page 3
page 4
10,468
page 5
12,287
frame number
valid–invalid bit
0 2 v
1 3 v
2 4 v
3 7 v
4 8 v
5 9 v
6 0 i
7 0 i
page table
0
1
2 page 0
3 page 1
4 page 2
5
6
7 page 3
8 page 4
9 page 5
page n

Impact of internal fragmentation

- Because the program extends only to address **10468**, any reference beyond that address is illegal.
- However, references to page 5 are classified as valid, so accesses to addresses up to **12287 (6 * 2048)** are valid.
- Addresses from **12288 to 16383** are invalid

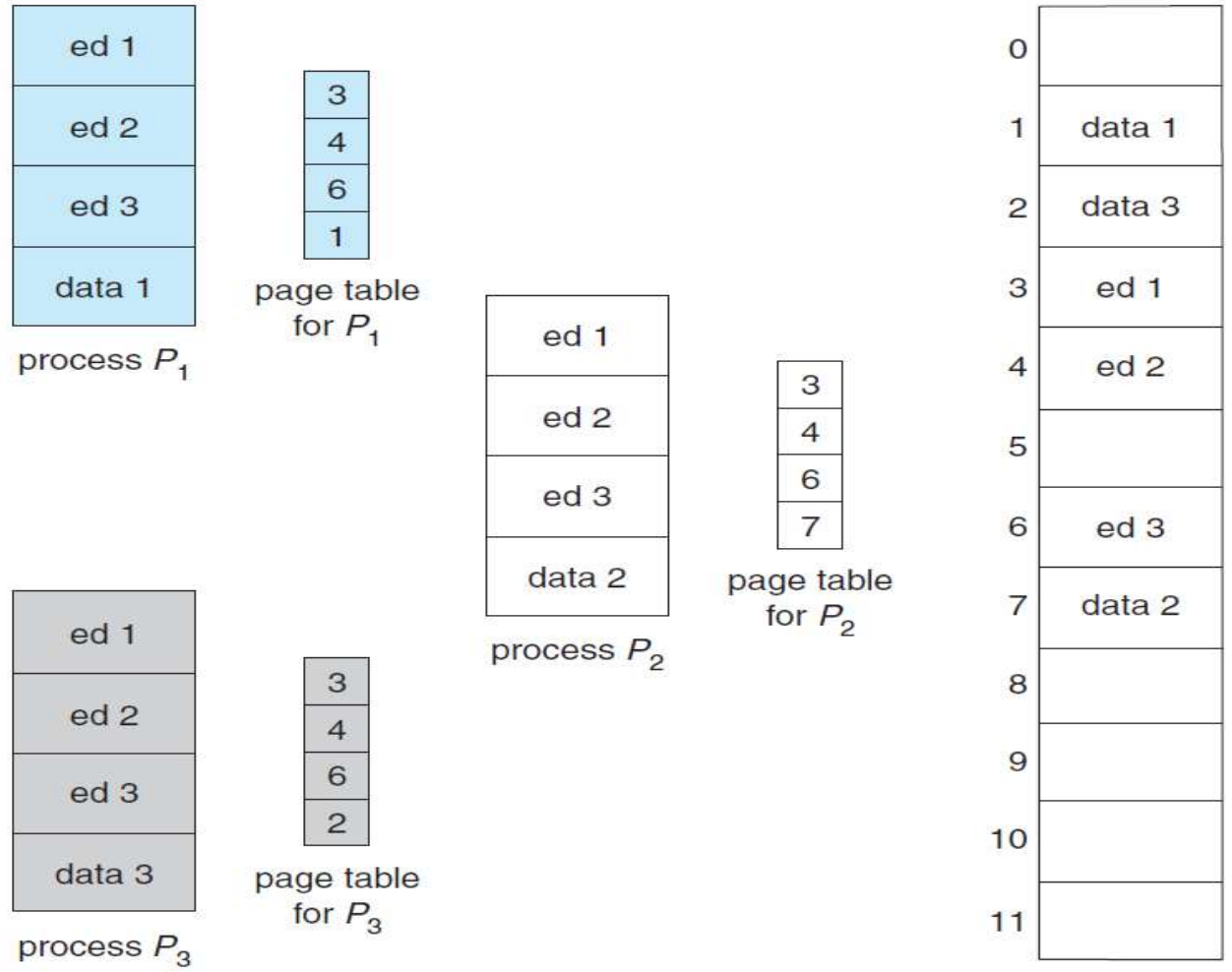
Page-table length register (PTLR)

- Rarely does a process use all its address range.
- In fact, many processes use only a **small fraction** of the address space available to them.
- It would be **wasteful** in these cases to **create a page table with entries for every page** in the address range.
- Most of this table would be **unused** but would take up valuable memory space.
- Some systems provide hardware, in the form of a **page-table length register** (PTLR), to indicate the **size of the page table**.
- This value is **checked against every logical address** to verify that the **address is in the valid range** for the process
- Failure of this test causes an **error trap** to the operating system.

Shared Pages

- An advantage of paging is the possibility of *sharing* common code.
- Consider a system that supports **40 users**, each of whom executes a **text editor**.
- If the **text editor** consists of **150 KB of code** and **50 KB of data** space, we need **8,000 KB (200 KB X 40)** to support the **40 users.**
- If the code is **reentrant code** (or **pure code),** however, it can be shared
- Reentrant code is **non-self-modifying code**; it never changes during execution.
- Thus, two or more processes can **execute the same code at the same time.**

- Each process has its **own copy of registers and data** storage to hold the data for the process's execution.
- The **data** for two different processes will be different.



- Only **one copy of the editor** need be kept in physical memory.
- Each user's **page table maps onto the same physical copy (frames)** of the editor, but data pages are mapped onto different frames.
- Thus, **to support 40 users**, we need **only one copy of the editor** (**150 KB**), plus **40 copies of the 50 KB of data** space per user.
- The total space required is **now 2,150 KB** instead of **8,000 KB**—a significant savings.

- Other **heavily used programs** can also be **shared**—
  - **compilers, window systems, run-time libraries, database systems**, and so on.
- To be sharable, the code must be **reentrant**.

Copy-on-Write

- Recall that the **fork()** system call creates a **child process** that is a **duplicate of its parent**.
- Traditionally, fork() worked by **creating a copy of the parent's address space** for the child, duplicating the pages belonging to the parent.
- However, considering that many child processes **invoke the exec() system call** immediately after creation, the copying of the parent's address space may be unnecessary.
- Instead, we can use a technique known as **copy-on-write,** which works by allowing the parent and child processes **initially to share the same pages.**

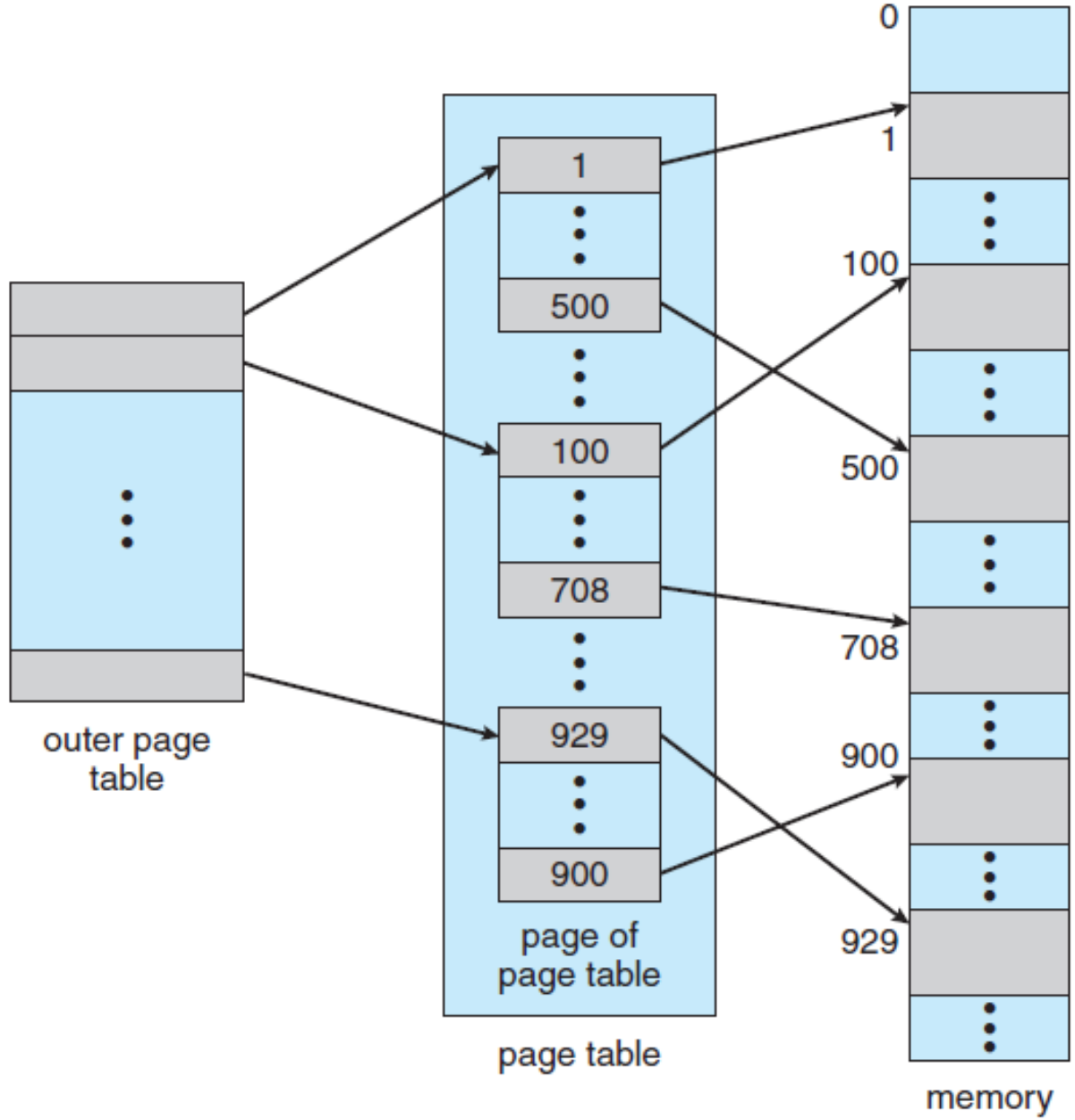
- These shared pages are **marked as copy-on-write pages**, meaning that if either process **writes to a shared page**, a **copy of the shared page is created.**
- For example, assume that the **child process attempts to modify** a page containing portions of the stack, with the pages set to be copy-on-write.
- The operating system will **create a copy of this page**, mapping it to the address space of the child process.
- Copy-on-write is a **common technique** used by several operating systems, including **Windows XP, Linux, and Solaris**

- When it is determined that **a page is going to be duplicated using copy on-write**, it is important to note the location from which the **free page** will be allocated.
- Many operating systems provide a **pool of free pages** for such requests.
- These **free pages** are typically allocated when the **stack or heap** for a process must **expand** or when there are **copy-on-write pages** to be managed.

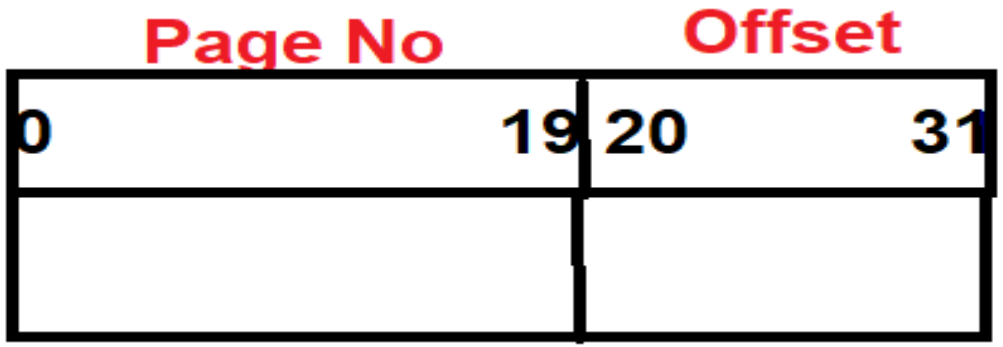
Structure of the Page Table

Hierarchical page table

- Most modern computer systems support a large logical address space (**2^{32} to 2^{64}).**
- In such an environment, the **page table** itself becomes **excessively large**.
- Clearly, we would not want to allocate the page table contiguously in main memory
- One simple solution to this problem is to **divide the page table into smaller pieces**.
- We can accomplish this division in several ways.
- One way is to use a **two-level paging algorithm**, in which the page table itself is also paged

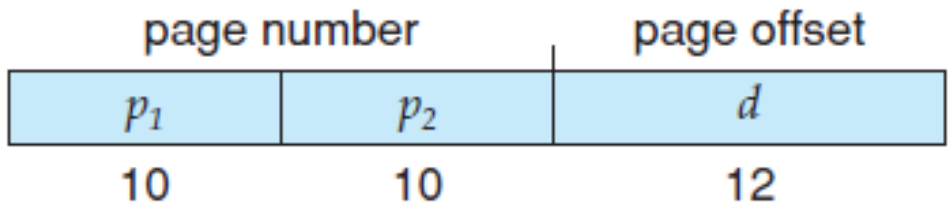

0
1
100
500
708
900
929
1
500
100
708
929
900
outer page table
page of page table
page table
memory

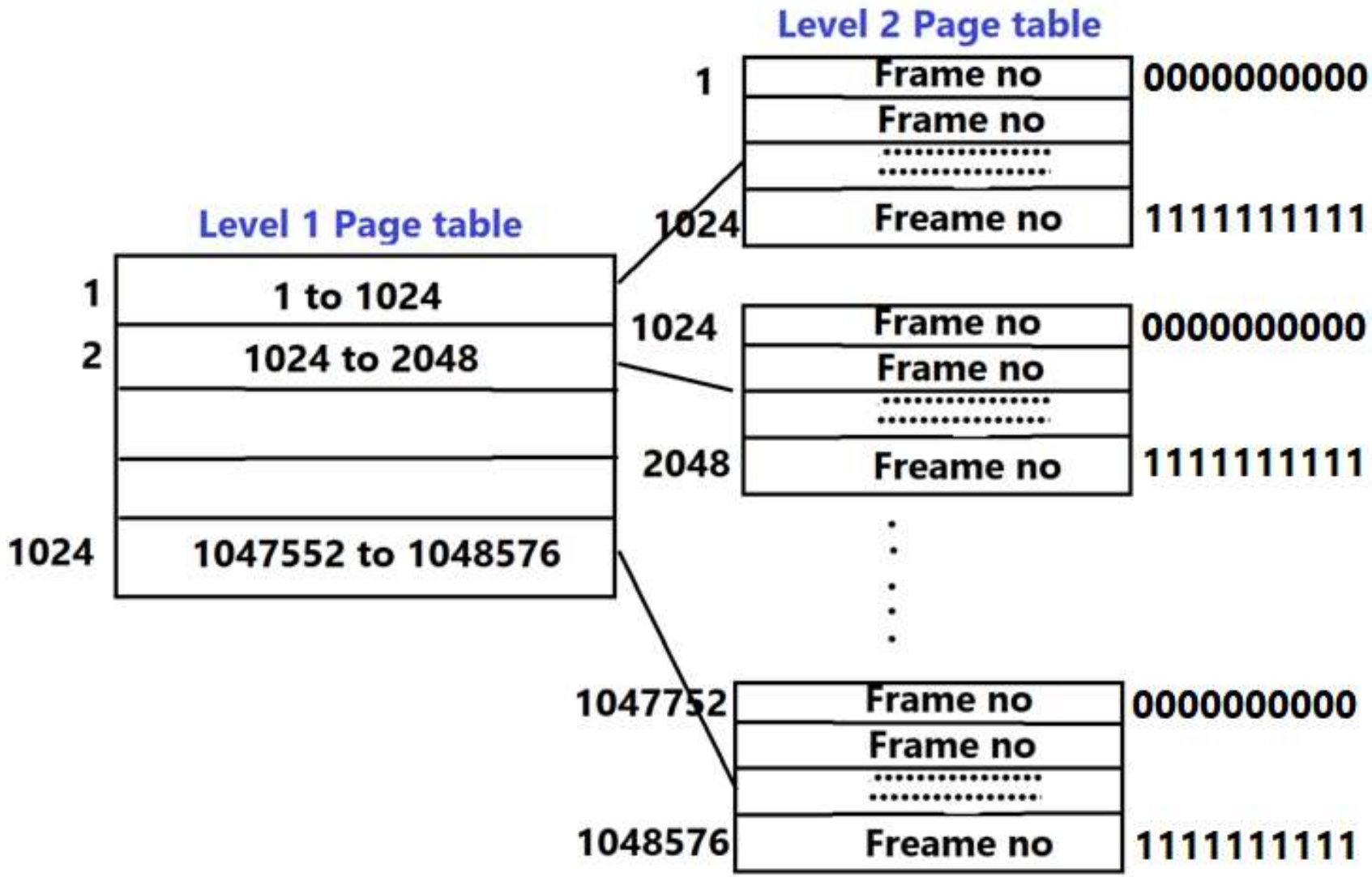
- Consider an example of a **32-bit** machine with a **page size of 4 KB**.
- That means there could be totally **10,48,576** pages ((2^32 / 1024) / 4) per process
- A logical address is divided into a **page number** consisting of **20 bits (2^{20} = 10,48,576)**and a **page offset** consisting of **12 bits (2^{12} = 4096 = 4 KB)**



- We **cant swap the page table** into disk since it is essential for address translation

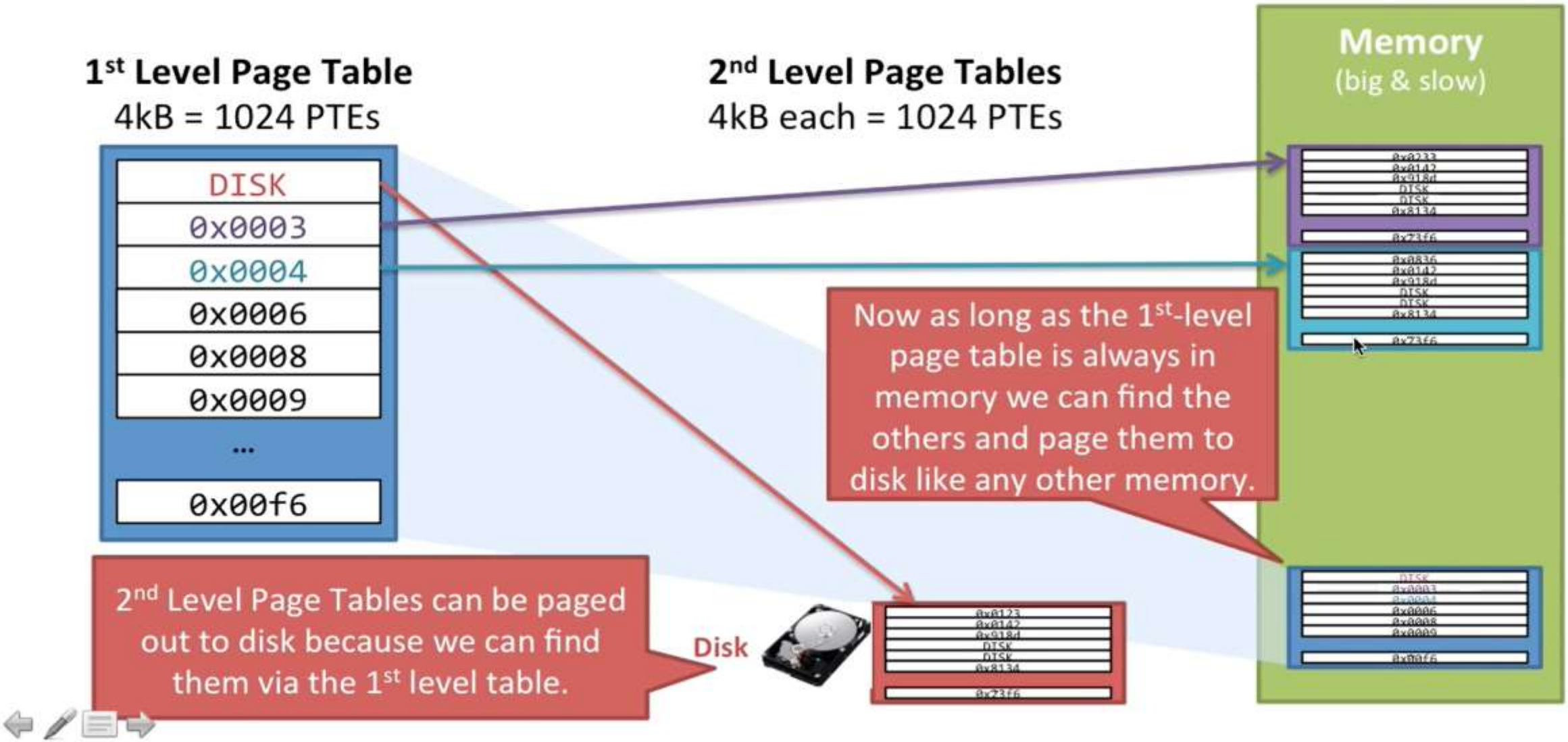
- Because we **page the page table** under **Hierarchical Paging**, the page number is further divided into a **10-bit page** number and a **10-bit page offset**.
- So the outer table will only have **1024 entries**




Level 2 Page table
1
Frame no
0000000000
Frame no
1024
Freame no
1111111111
Level 1 Page table
1
1 to 1024
2
1024 to 2048
1024
1047552 to 1048576
1024
Frame no
0000000000
Frame no
2048
Freame no
1111111111
1047752
Frame no
0000000000
Frame no
1048576
Freame no
1111111111

- Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped page table.**

Multi-level page tables



Multi-level page table translation

Virtual Address

0x00402204

Physical Address

Multi-level page table translation

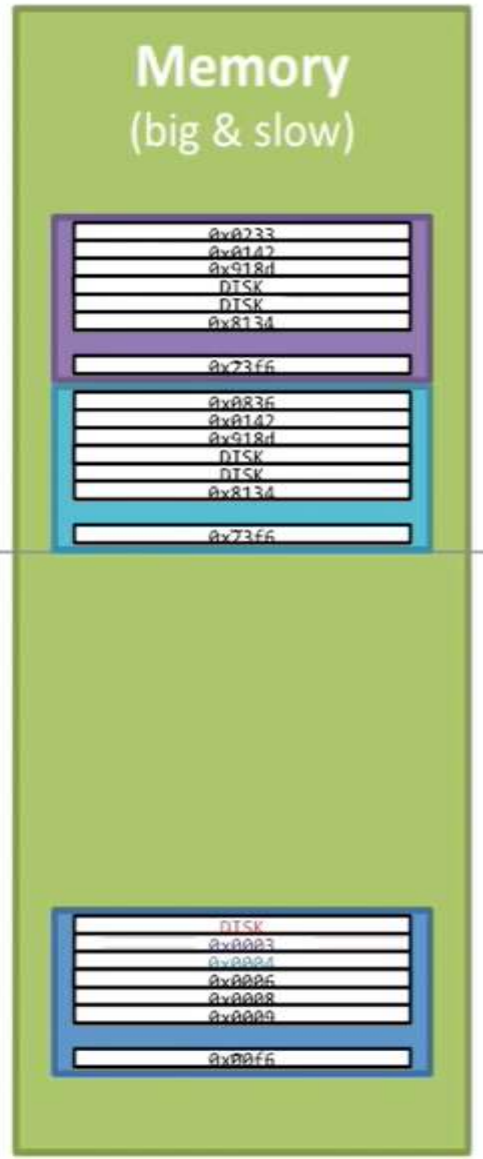
Virtual Address

0x00402204

Virtual page number

Page offset

11 0x204 0

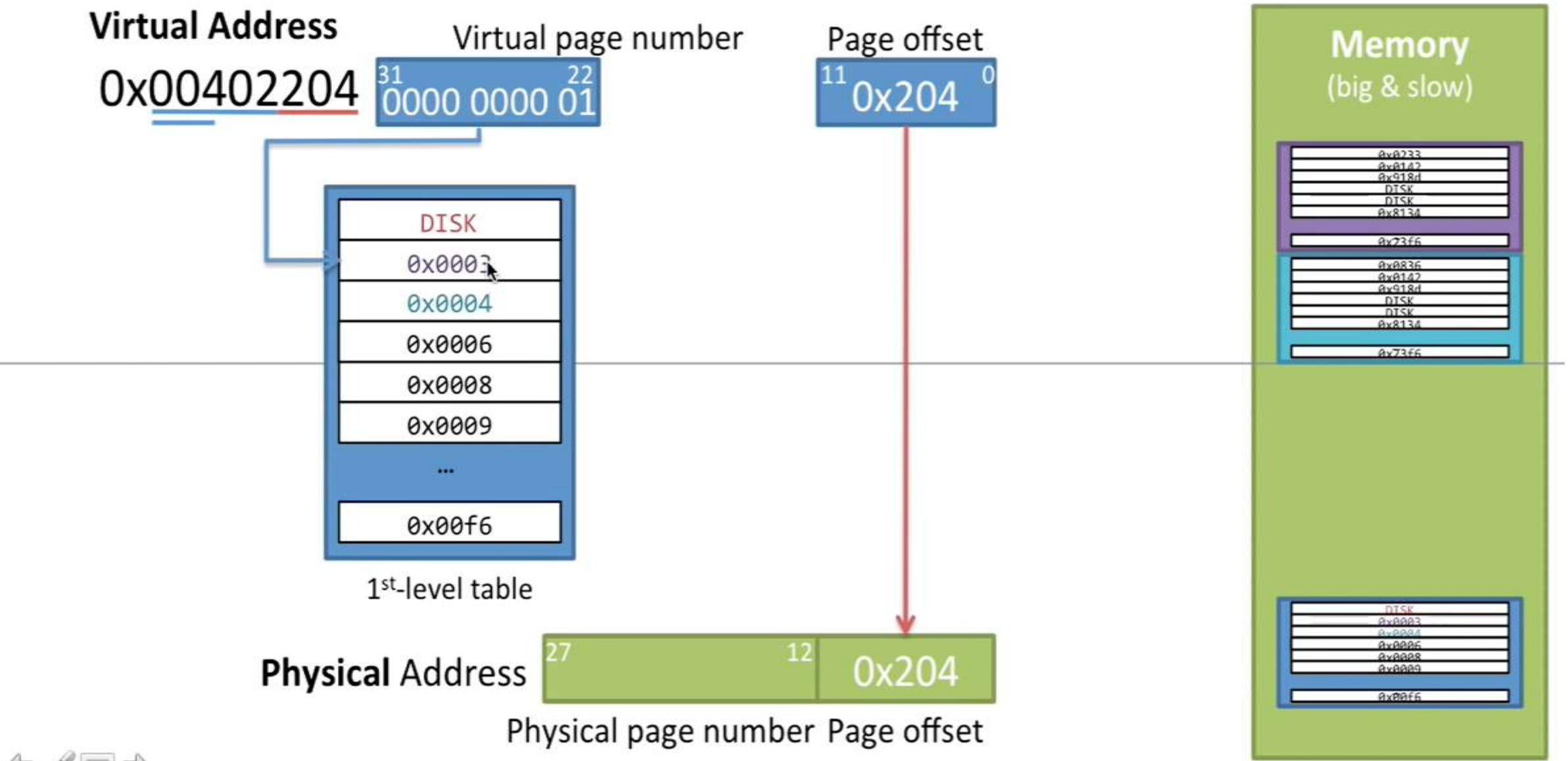


Physical Address

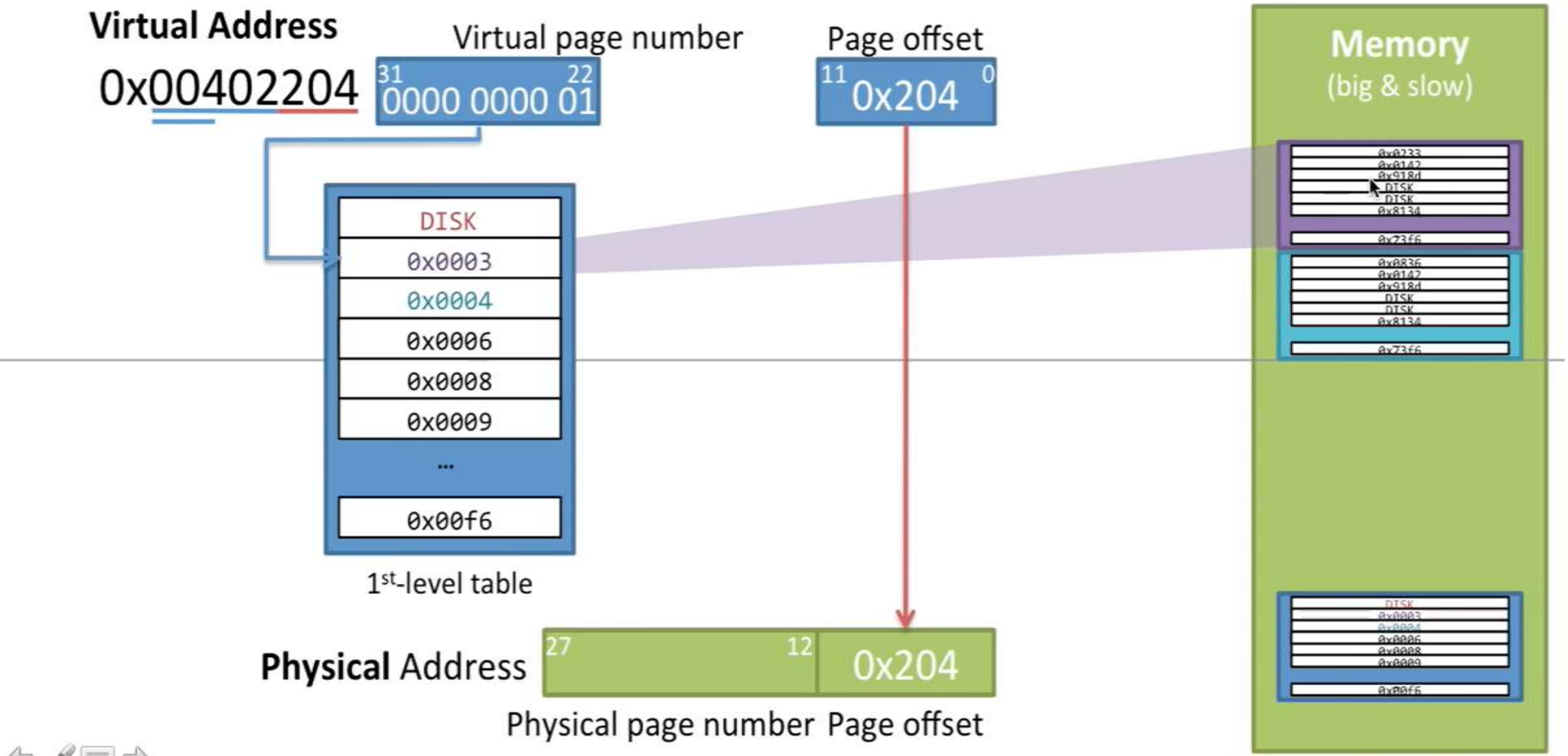
27 12 0x204

Physical page number Page offset

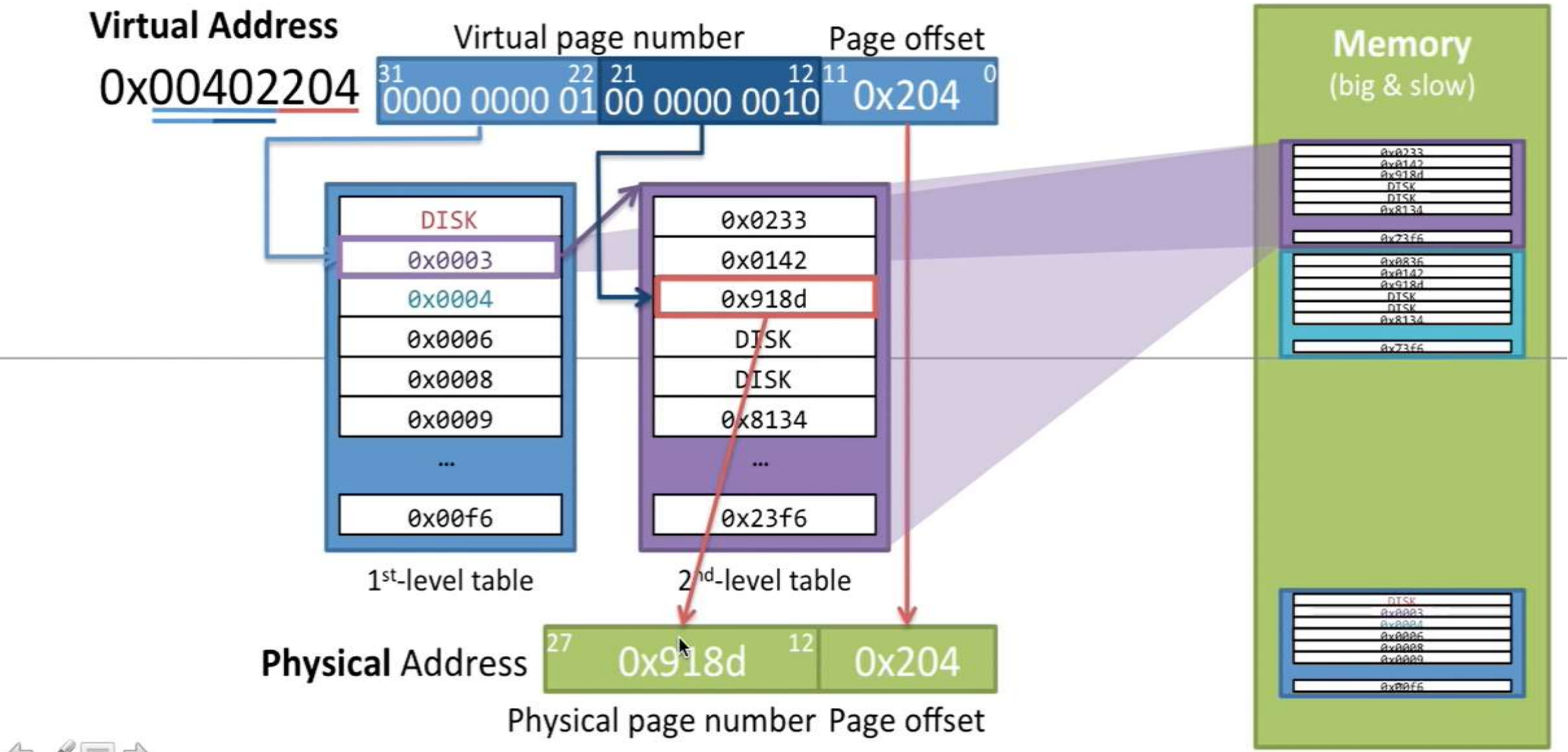
Multi-level page table translation



Multi-level page table translation

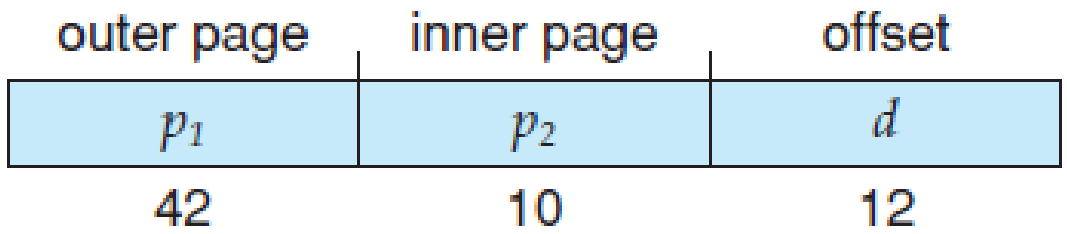


Multi-level page table translation

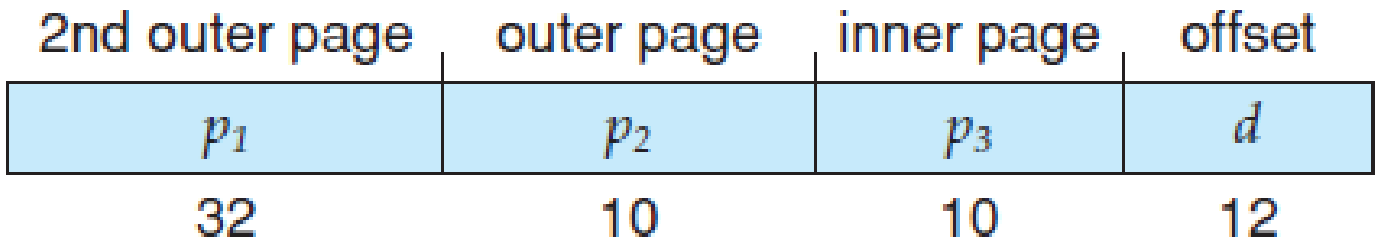


Multi-level Page tables

- For a system with a **64-bit** logical address space, **a two-level** paging scheme is **no longer appropriate**
- suppose that the page size in such a system is **4 KB (2^{12}).**
- In this case, the page table consists of up to **2^{52} entries**.
- If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain **2^{10}** 4-byte entries.
- The outer page table consists of **2^{42} entries**



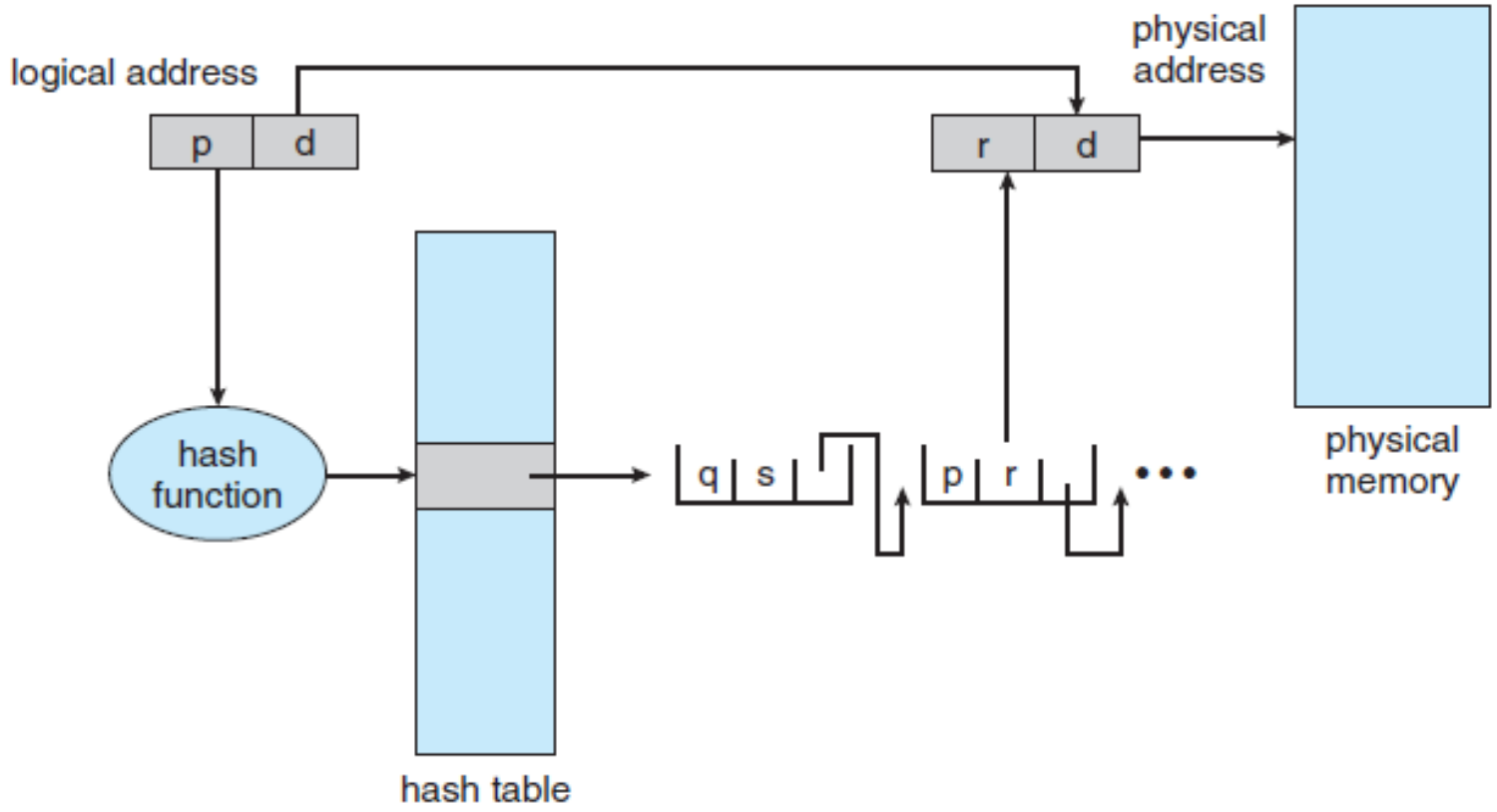
- To avoid such a large table, **divide the outer page table** into smaller pieces.
- We can divide the outer page table in various ways.
- For example, we can page the outer page table, giving us a three-level paging scheme



Hashed Page Tables

- A common approach for handling address spaces **larger than 32 bits** is to use a **hashed** page table, with the **hash value** being the **virtual page number.**
- Each entry in the **hash table** contains a **linked list of elements** that hash to the same location (to **handle collisions**).
- Each element consists of three fields: (1) the **virtual page number**, (2) the value of the mapped **page frame**, and (3) a **pointer to the next element** in the linked list.
- The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table.
- The virtual page number is compared with field 1 in the first element in the linked list.

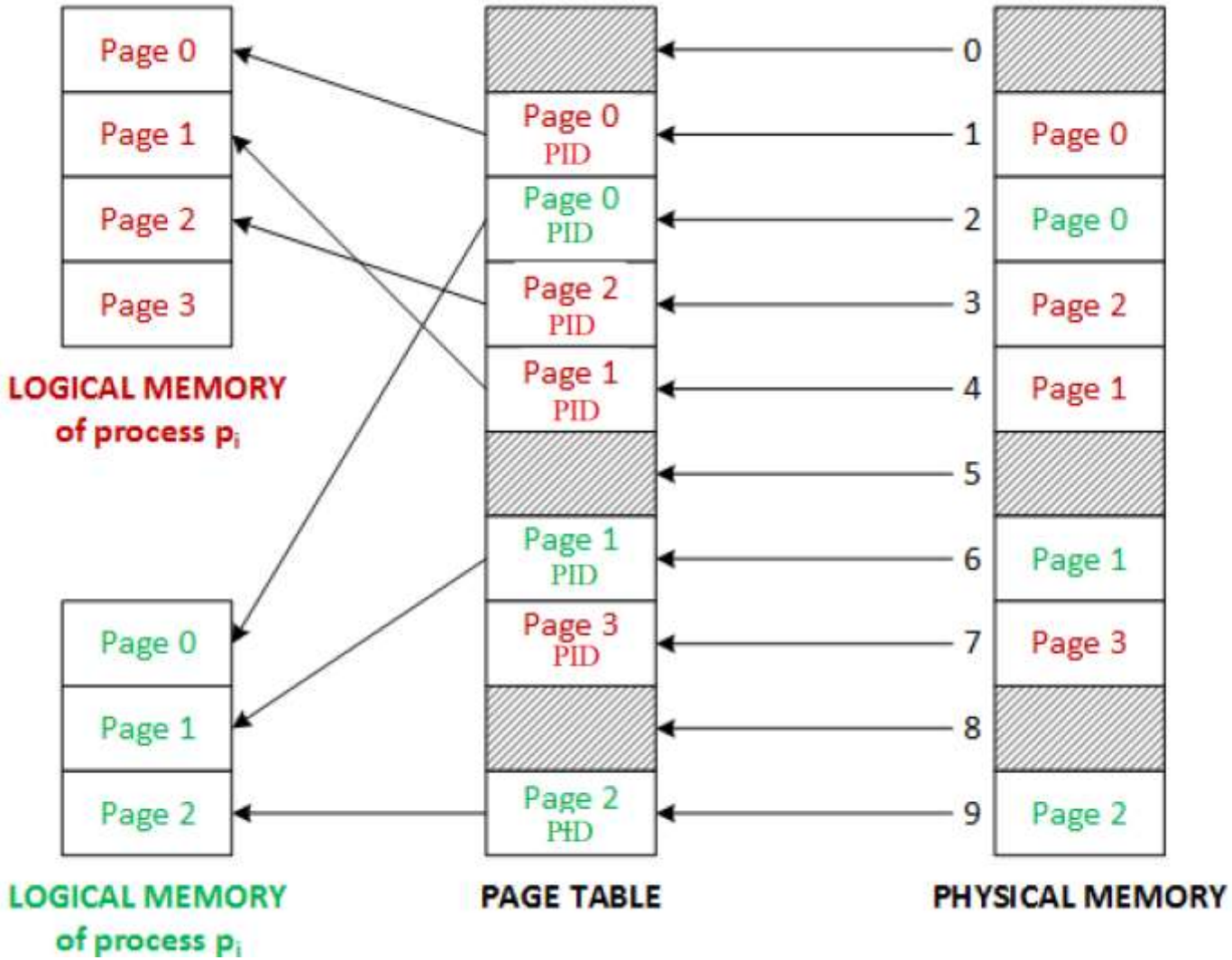
- If there is a match, the corresponding page frame (field 2) is used to form the desired physical address.
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.



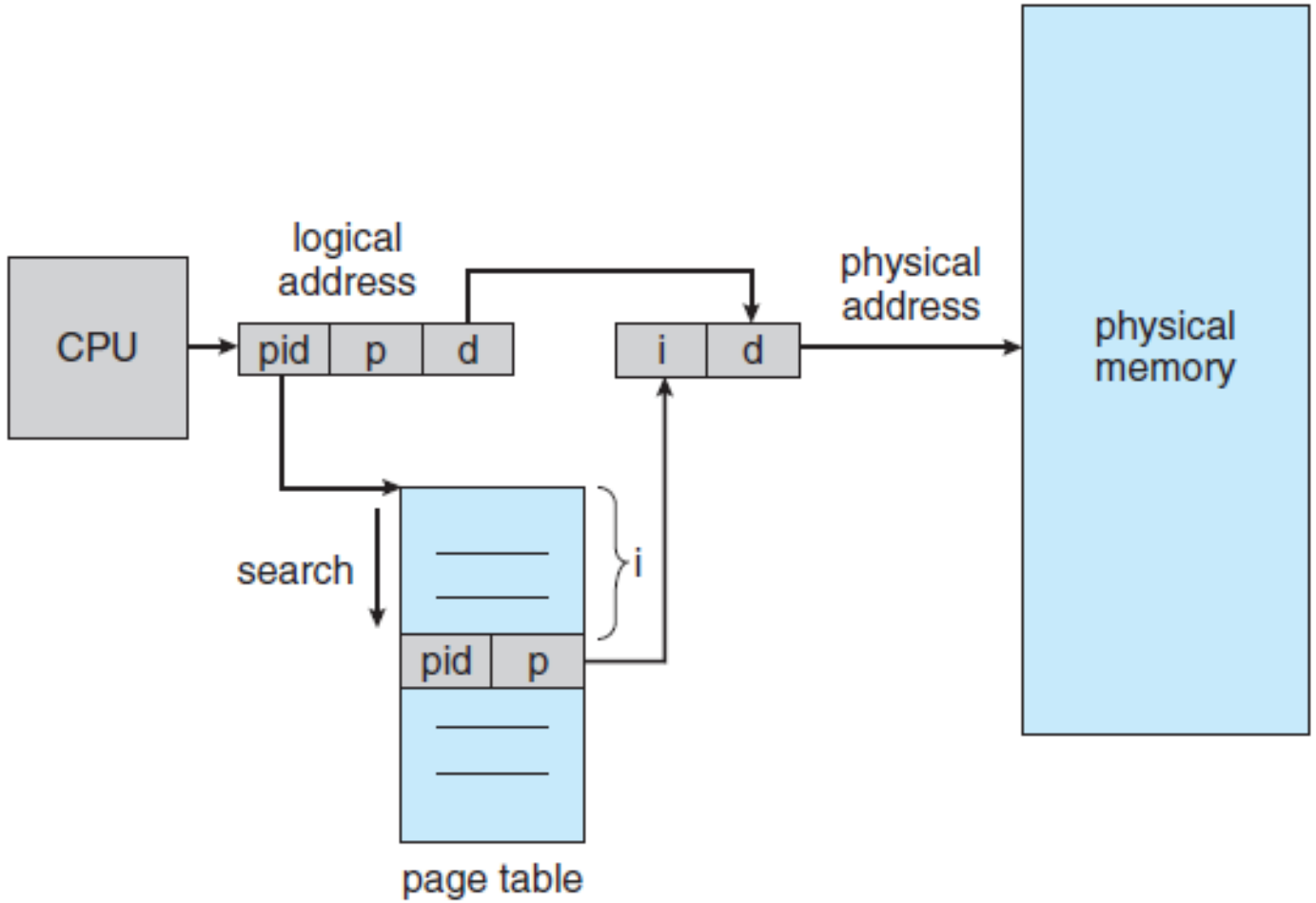
Inverted Page Tables

- Page tables may **consume large amounts of physical memory** just to keep track of how other physical memory is being used.
- To solve this problem, we can use an **inverted page table.**
- An inverted page table has **one entry for each frame** of memory.
- Each **entry consists of the page no** of the page stored in that real memory location, with information about the **process that owns** that page.
- Thus, **only one page table** is in the system, and it has only **one entry for each page** of physical memory.
- An inverted page table *always* fits in DRAM because it is proportional in size to the DRAM.

- Inverted page tables often require that an **address-space identifier (ASID)** be stored in each entry of the page table, since the table usually contains several different address spaces mapping physical memory.
  - <**process-id, page-number, offset**>.
- Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame.


Page 0
Page 1
Page 2
Page 3
LOGICAL MEMORY
of process p_i
Page 0
Page 1
Page 2
LOGICAL MEMORY
of process p_j
Page 0 PID
Page 0 PID
Page 2 PID
Page 1 PID
Page 1 PID
Page 3 PID
Page 2 PID
PAGE TABLE
0
1
2
3
4
5
6
7
8
9
Page 0
Page 0
Page 2
Page 1
Page 1
Page 3
Page 2
PHYSICAL MEMORY

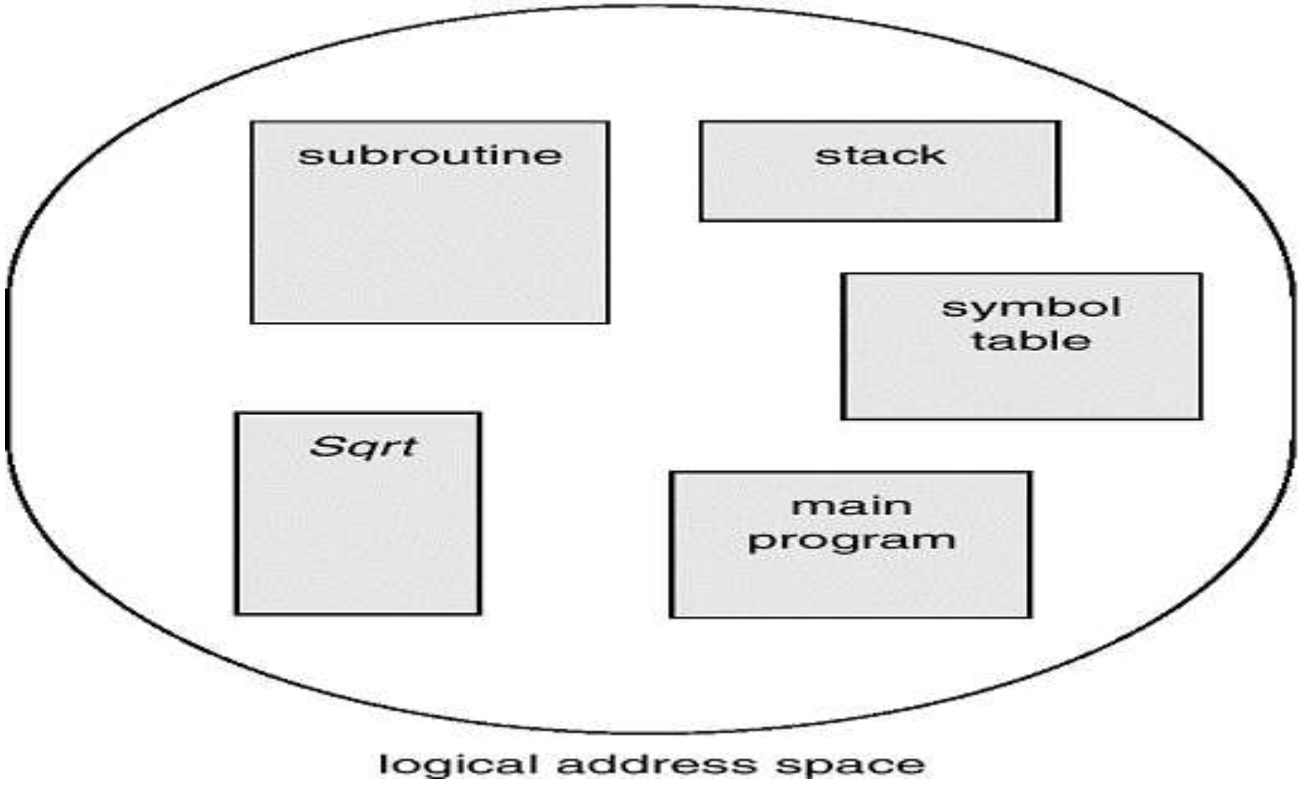
- Although this scheme decreases the amount of memory needed to store each page table, it **increases the amount of time needed to search** the table when **a page reference** occurs
- To alleviate this problem, we use a **hash table to limit** the search to one—or at most a few—page-table entries.
- Of course, each access to the hash table **adds a memory reference** to the procedure, so one virtual memory reference requires **at least two real memory reads**—one for the hash-table entry and one for the page table.
- To improve performance, recall that the **TLB is searched first**, before the hash table is consulted.


CPU
logical address
pid
p
d
i
d
physical address
physical memory
search
i
pid
p
page table

Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments.
- A segment is a logical unit such as:
  - **procedure**
  - **function**
  - **method**
  - **object**
  - **local variables**
  - **global variables**
  - **common block**
  - **stack**
  - **symbol table**
  - **arrays**

User's View of a Program

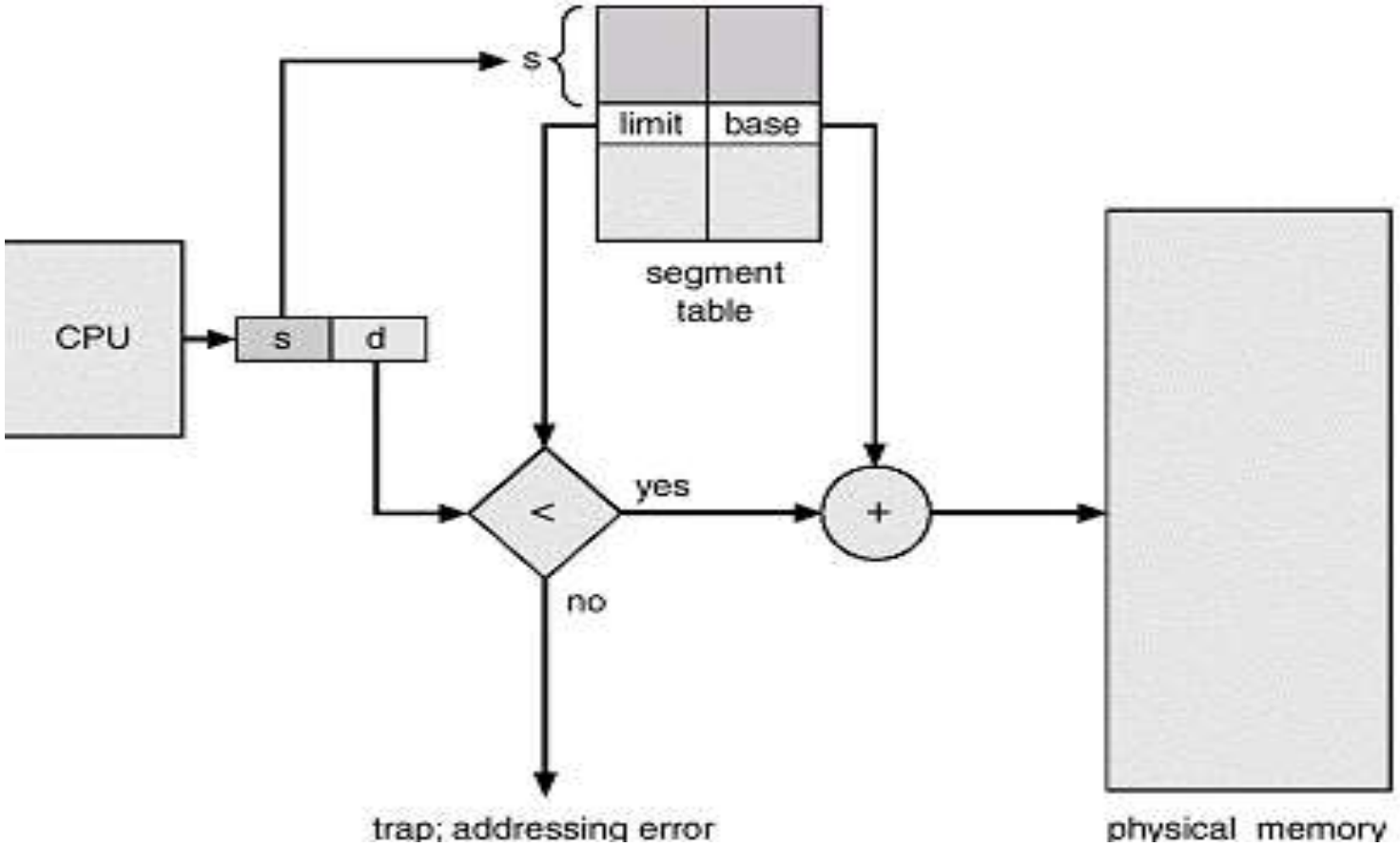


- Whereas **paging is invisible to the programmer**, **segmentation is usually visible** and is provided as a convenience of organizing program and data
- Typically the **programmer or compiler** will assign **programs and data to different segments**
- Segmentation **overcomes external fragmentation**

Segmentation Architecture

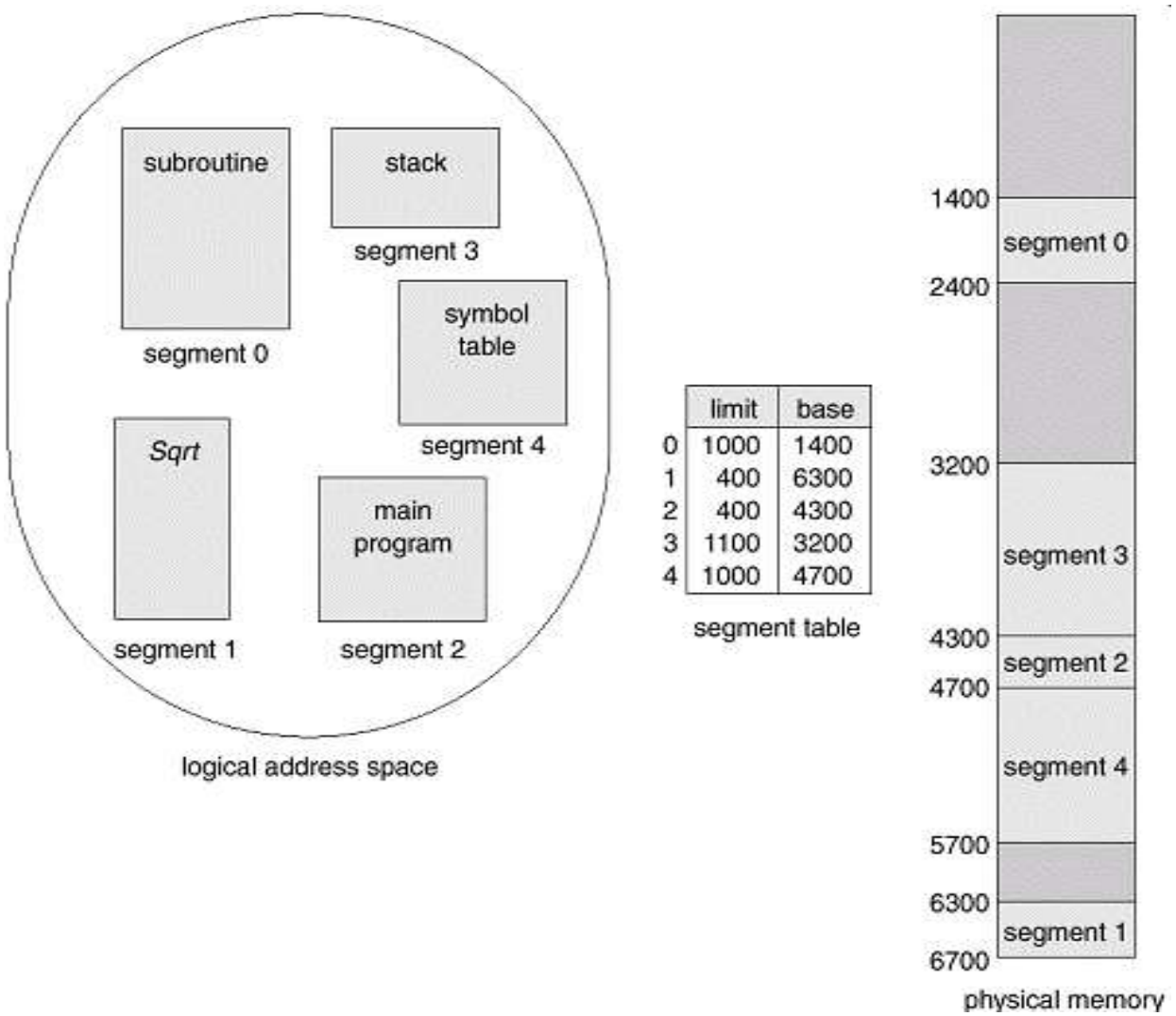
- Logical address consists of a two tuple: **<segment-number, offset>**
- *Segment table* – maps two-dimensional physical addresses; each table entry has:
  - 1. **base** – contains the starting physical address where the segments reside in memory.
  - 2. ***limit*** – specifies the length of the segment.
- ***Segment-table base register (STBR)*** points to the segment table's location in memory.
- ***Segment-table length register (STLR)*** indicates number of segments used by a program;
- segment number *s* is legal if *s* < STLR.

Segmentation Hardware

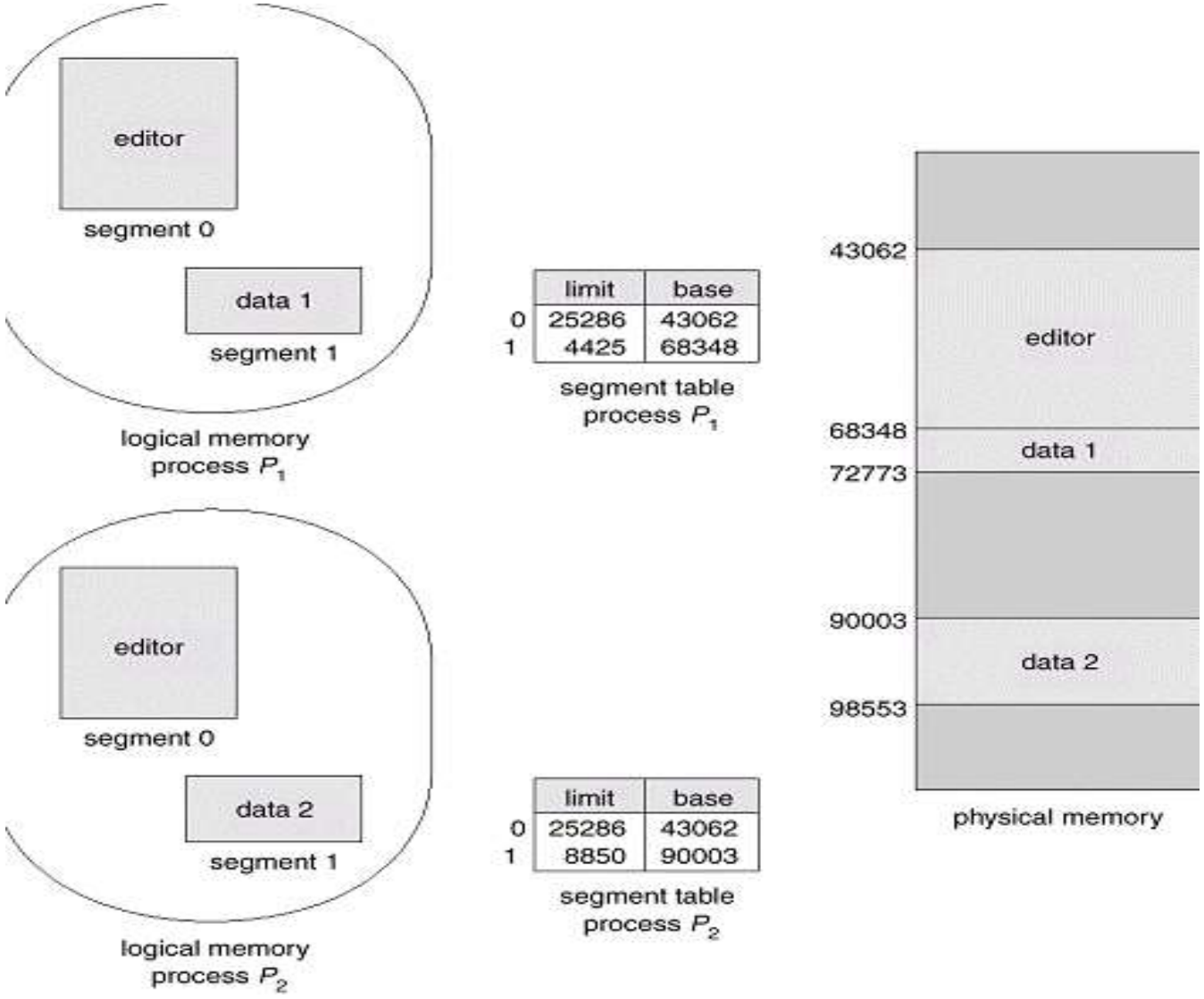


- Relocation is dynamic
- Shared segments with same segment number
- Allocation first fit/best fit
- External fragmentation
- Protection. With each entry in segment table associate:
  - 1. validation bit = 0 _ illegal segment
  - 2. read/write/execute privileges

Example of Segmentation

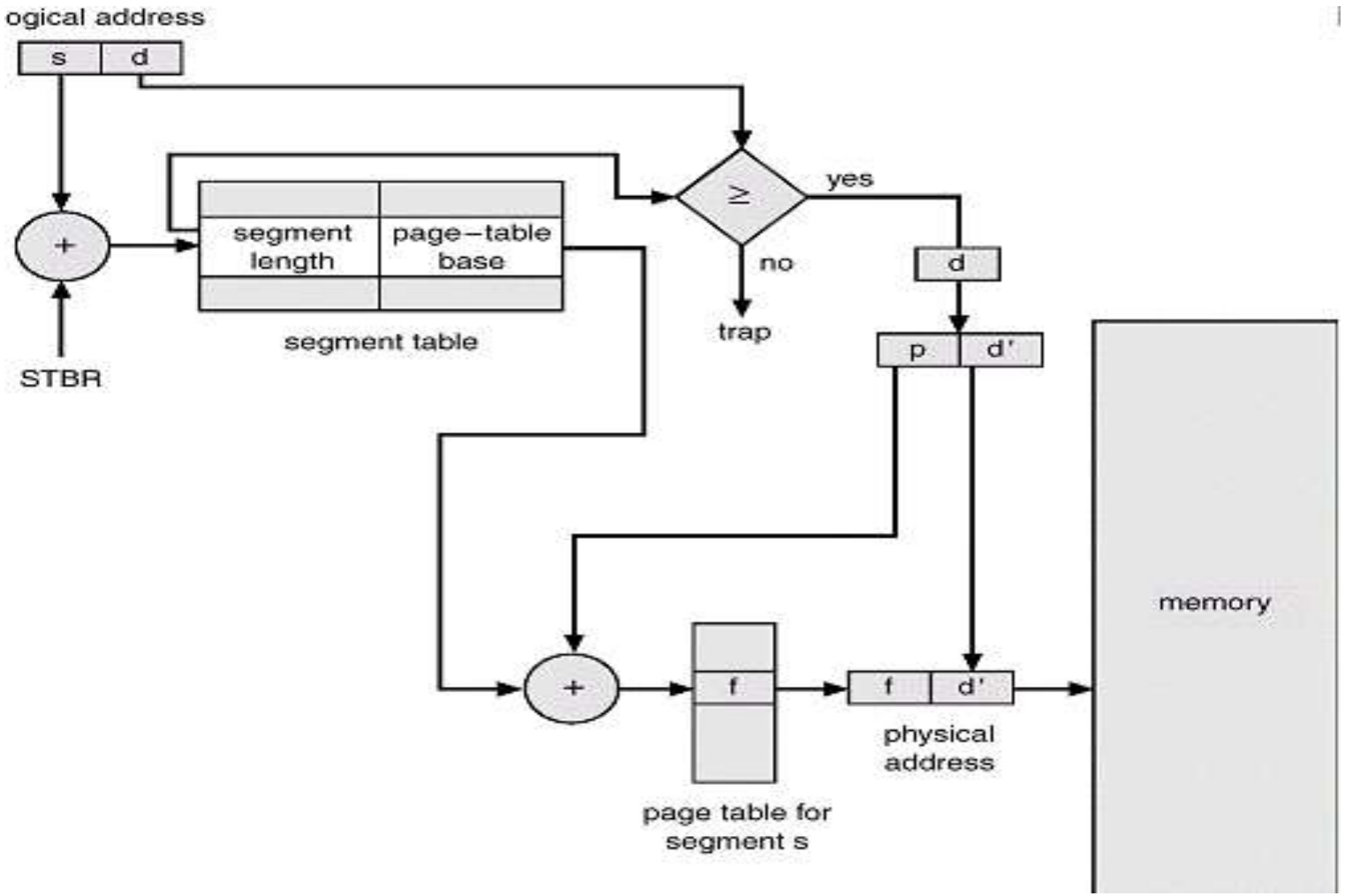


Sharing of Segments



Segmentation with Paging – MULTICS

- The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.
- Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment.


ogical address
s
d
+
STBR
segment
length
page–table
base
segment table
≥
yes
no
trap
d
p
d'
+
f
f
d'
physical
address
page table for
segment s
memory

Segmentation Vs Paging

Page Replacement

- We assumed that **each page faults at most once**, when it is first referenced.
- This representation is **not strictly accurate**.
- But If a process of ***ten pages*** actually uses ***only half*** of them, then **demand paging** saves the I/O necessary **to load the five pages** that are never used.
- We could also **increase our degree of multi programming** by running twice as many processes.
- Thus, if we had **forty frames**, we could run **eight processes**, rather than the **four**

- We have higher CPU utilization and ***throughput.***
- What if **all the processes suddenly try to use all of its pages**?.
- **System memory** is not used only for holding **program pages.**
- **Buffers for I/O** also consume a significant amount of memory.
- Deciding how much memory to allocate to I/O and how much to program pages is a significant challenge.
- Some systems allocate a **fixed percentage of memory for I/O buffers**, whereas others allow both user processes and the I/O subsystem **to compete** for all system memory.

- When a **page fault occurs** and the OS determines where the desired page is residing on the **disk** but then finds that there are ***no* free frames**.
- OS has several options at this point.
- It could **terminate the user process**.
- OS could instead **swap out a process**, freeing all its frames and reducing the level of multiprogramming.
- Second strategy requires **page replacement**

Basic Page Replacement

- If **no frame is free**, we find one that is **not currently being used** and free it.
- We can free a frame by **writing its contents** to swap space and changing the page table.
- We can **now use the freed frame** to hold the page for which the process faulted.
- 1. Find the location of the desired page on the disk.
- 2. Find a free frame:
  - a. If there is a free frame, use it.

- b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
- c. Write the victim frame to the disk; change the page and frame tables accordingly.

- Notice that, if no frames are free, ***two*** **page transfers** (one out and one in) are required.
- This situation effectively **doubles the page-fault service time** and increases the effective access time accordingly
- We can reduce this overhead by using a **modify bit** (or **dirty bit).**

- When this scheme is used, each page or frame has a **modify bit** associated with it in the hardware.
- The modify bit for a page is **set by the hardware** whenever **any word or byte** in the page is **written into**, indicating that the page has been **modified**.
- When we select a page for replacement, we **examine its modify bit.**
- If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write that page to the disk.
- If the **modify bit is not set**, however, the page has *not* been modified since it was read into memory.
- Therefore, if the copy of the page on the **disk has not been overwritten** (by some other page, for example), then we need not write the memory page to the disk**: It is already there**

- This technique also applies to ***read-only pages***. Such pages cannot be modified; thus, they may be discarded when desired.
- We must solve two major problems to implement demand paging:
  - We must develop a **frame-allocation algorithm** and a **page-replacement 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.
- Disk I/O is so expensive

How do we select a particular replacement algorithm?

- In general, we want the one with the **lowest page-fault rate**.
- 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.**

7, 0,1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,1, 2, 0, 1, 7, 0,1 with 3 frames

- **FIFO**
  - Replace the page that is the oldest one.
  - Number of page faults: ?
- **Optimal Page Replacement Algorithm**
  - Replace the page that will not be used for the longest period of time.
- **LRU**
  - Replace page that was least recently used or the one that was not reference in the recent past.

Virtual Memory

- Virtual memory is a technique that allows the execution of **processes that are not completely in memory.**
- Programs can be **larger than physical memory**.
- Further, virtual memory **abstracts main memory into an extremely large**, uniform array of storage, separating logical memory as viewed by the user from physical memory.
- This technique **frees programmers** from the concerns of **memory-storage limitations**
- Virtual memory also **allows processes to share files** easily by implementing **shared pages**
- Virtual memory is not easy to implement, however, and may **substantially decrease performance** if it is used carelessly.

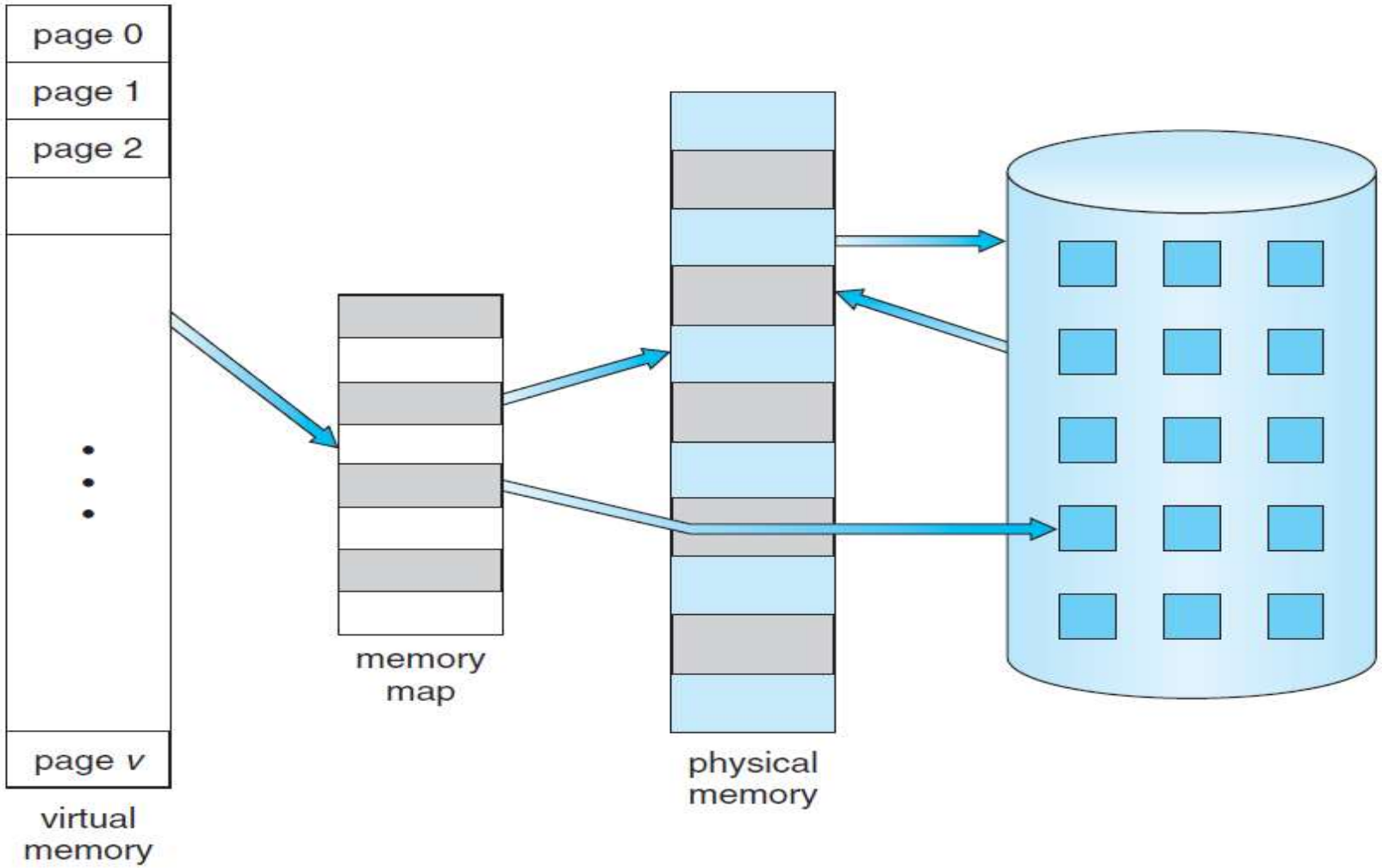
Background

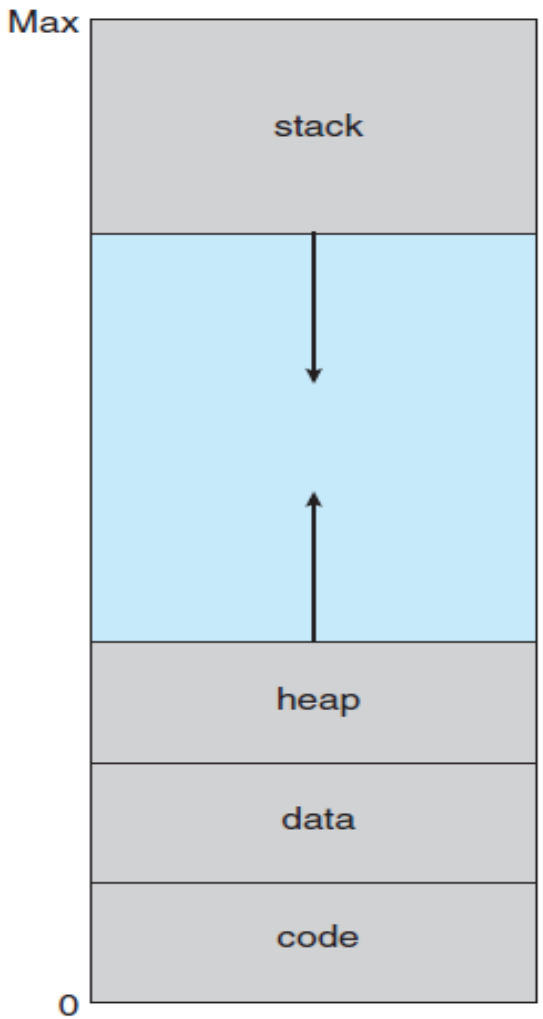
- The memory-management algorithms are necessary because of one basic requirement:-
  - The **instructions being executed must be in physical memory**.
- It **limits the size of a program to the size of physical memory.**
- The first approach to meeting this requirement is to place the **entire logical address space in physical memory**.
- ***Dynamic loading*** can help to ease this restriction, but it generally requires special precautions and extra work by the programmer.

- An examination of real programs shows us that, in many cases, the **entire program is not needed**. Consider the following:
  - Programs often have code to handle unusual ***error conditions***. These errors occur seldom.
  - **Arrays, lists, and tables** are often allocated more memory than they actually need.
  - Certain **options and features** of a program may be used **rarely**
- Even in those cases where the entire program is needed, it may **not all be needed at the same time**.

- The ability to execute a program that is only **partially in memory** would confer many benefits:
  - A **program would no longer be constrained by the amount of physical memory** that is available. Users would be able to write programs for an extremely large *virtual* address space.
  - Because each user program could take less physical memory, **more programs could be run** at the same time, with a corresponding increase in **CPU utilization** and throughput.
  - **Less I/O would be needed to load or swap each user program into memory**, so each user program would run faster.

- Virtual memory involves the **separation of logical memory from physical memory.**
- The **virtual address space** of a process refers to the **logical (or virtual) view of how a process is stored** in memory.
- Physical memory may be organized in **page frames** and that the physical page frames assigned to a process **may not be contiguous**.
- Virtual address spaces that include holes are known as ***sparse address spaces***.
- It is up to the memory management unit (**MMU**) to **map logical pages to physical page frames** in memory.


page 0
page 1
page 2
page v
virtual memory
memory map
physical memory


Max
stack
heap
data
code
0

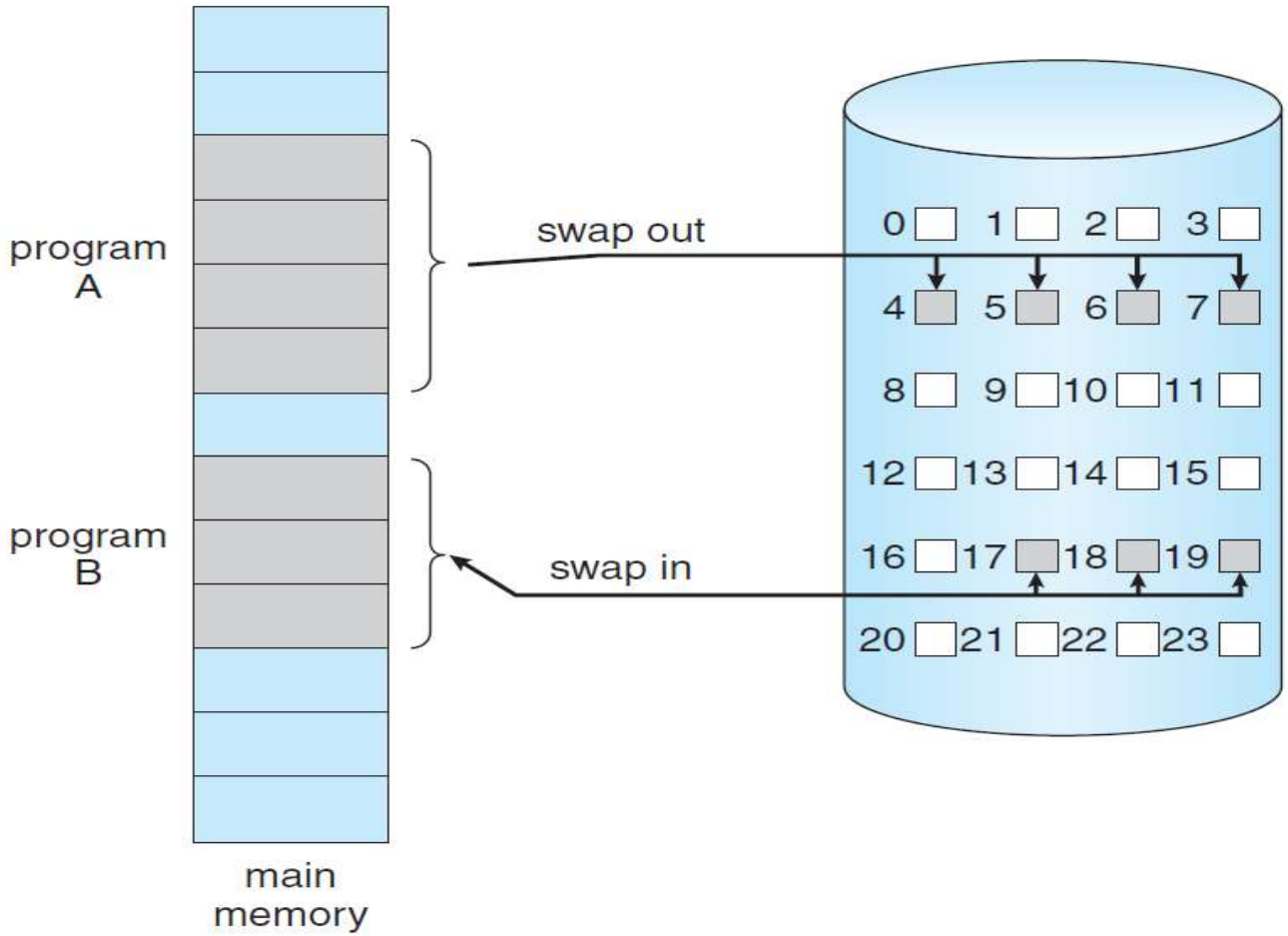
- We allow the **heap to grow upward** in memory as it is used for **dynamic memory allocation.**
- Similarly, we allow for the **stack to grow downward in memory** through **successive function calls.**
- The large **blank space** (**or hole**) between the heap and the stack is **part of the virtual address space** but will **require actual physical pages** only if the heap or stack grows.
- Virtual address spaces that include holes are known as **sparse address spaces**
- Using a sparse address space is beneficial because the **holes can be filled as the stack or heap segments grow** or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

- In addition to separating logical memory from physical memory, virtual memory also **allows files and memory to be shared by two or more processes** through **page sharing**.
- This leads to the following benefits:
  - **System libraries can be shared** by several processes through mapping of the shared object into a virtual address space.
  - Virtual memory **enables processes to share memory** for shared memory based **communication.**
  - Virtual memory can allow **pages to be shared** during process creation with the **fork() system call**, thus speeding up process creation.

Demand Paging

- Consider how an executable program might be loaded from disk into memory.
- One option is **to load the entire program in physical memory** at program execution time.
- A problem with this approach, is that **we may not initially *need* the entire program** in memory.
- An alternative strategy is to **initially load pages only as they are needed**.
- This technique is known as demand paging and is commonly used in virtual memory systems.
- 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

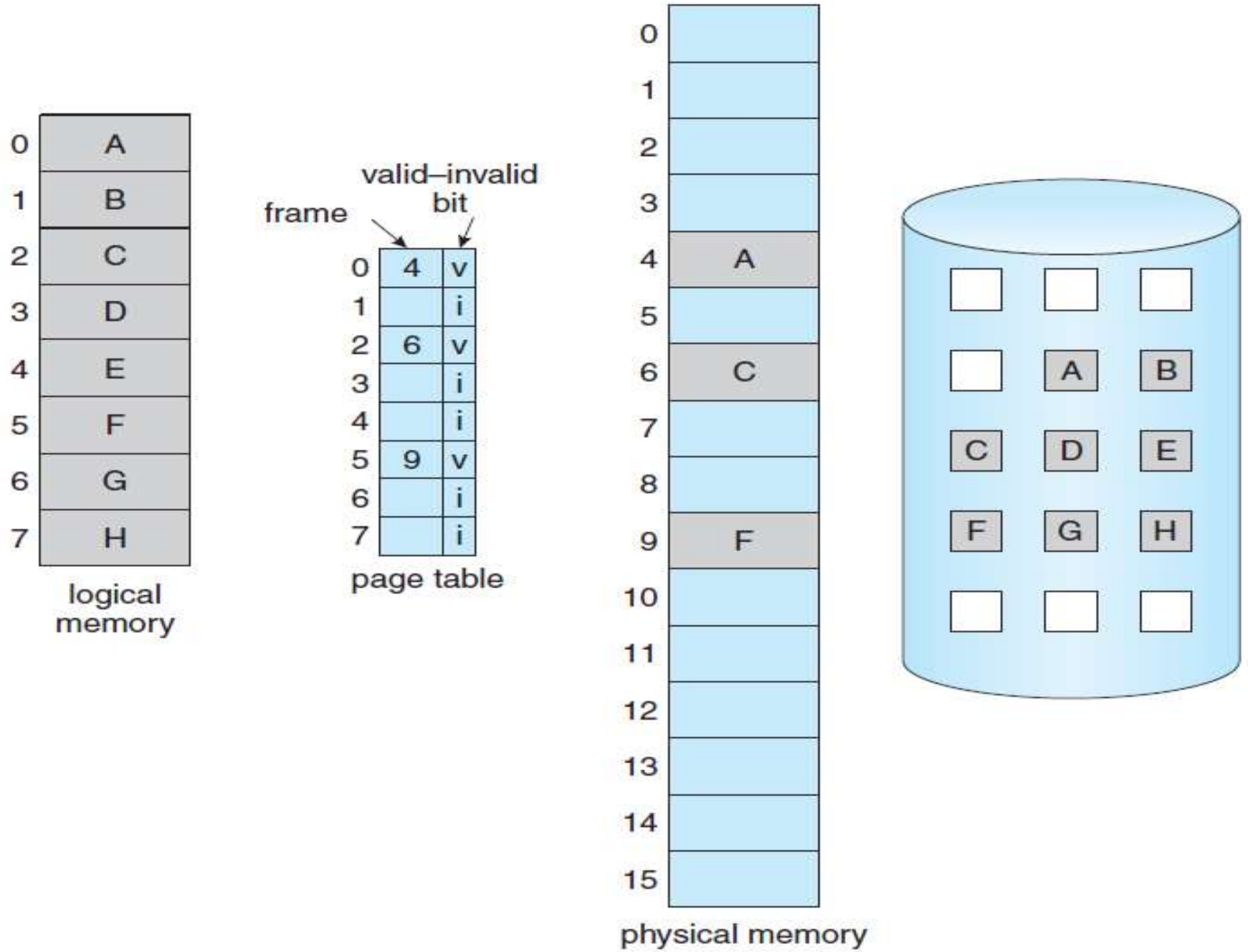
- A demand-paging system is similar to a paging system with swapping where **processes reside in secondary memory** (usually a disk).
- When we want to execute a process, we **swap it into memory.**
- Rather than swapping the entire process into memory, however, we use a **lazy swapper.**
- A lazy swapper never swaps a page into memory unless that page will be needed.
- A **swapper** manipulates entire processes, whereas a **pager** is concerned with the individual pages of a process


program
A
program
B
swap out
swap in
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
16 17 18 19
20 21 22 23
main
memory

Basic Concepts

- When a **process is to be swapped in(loaded)**, the **pager guesses which pages will be used** during execution
- Instead of swapping in a whole process, the **pager brings only those necessary pages** into memory.
- With this scheme, we need some form of hardware support to distinguish between the **pages that are in memory and the pages that are on the disk**.
- The ***valid-invalid*** bit scheme can be used for this purpose.
- This time, however, when this bit is set to "valid," the associated page is **both legal and in memory.**
- If the bit is **set to "invalid,"** the page either is **not valid** (that is, **not in the logical address space** of the process) or is valid but is **currently on the disk.**

- The page-table entry for a page that is brought into memory is **set as usual**, but the page-table entry for a page that is not currently in memory is either **simply marked invalid** or contains the **address of the page on disk.**
- Notice that marking a page invalid will have no effect **if the process never attempts** to access that page.
- Hence, **if we guess right** and page in **all pages that are actually needed** and only those pages, the **process will run** exactly as though we had brought in all pages.
- While the process executes and accesses pages that are **memory resident,** execution proceeds normally.


0
1
2
3
4
5
6
7
A
B
C
D
E
F
G
H
logical memory
frame
valid–invalid bit
0 4 v
1 i
2 6 v
3 i
4 i
5 9 v
6 i
7 i
page table
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
A
C
F
physical memory
A
B
C
D
E
F
G
H

Handling Page Fault

- what happens if the process tries to access a page that was not brought into memory?
- Access to a page marked invalid causes a **page fault.**
- The paging hardware, in translating the address through the page table, will notice that the **invalid bit is set**, causing **a trap** to the **operating system**.
- This trap is the result of the operating system's **failure to bring the desired page** into memory

- The procedure for handling this page fault is straightforward:
- 1. We check an **internal table** (usually kept with the **process control block**) for this process to determine whether the **reference was a valid or an invalid memory access**.
- 2. If the reference was **invalid**, we **terminate the process**. If it was valid but we have not yet brought in that page, we now have to **page it in**.
- 3. We **find a free frame** (by taking one from the free-frame list, for example).
- 4. We schedule a **disk operation** to read the **desired page** into the newly **allocated frame**

- 5. When the **disk read is complete**, we **modify the internal table** kept with the process and the page table to indicate that the page is now in memory.
- 6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

Pure demand paging

- In the extreme case, we can start executing a process with ***no* pages in memory**.
- When the operating system sets the **instruction pointer** to the **first instruction of the process**, which is on a non-memory-resident page, the process **immediately faults** for the page.
- After this page is **brought into memory**, the process continues to execute, faulting as necessary until every page that it needs is in memory.
- At that point, it can execute with no more faults - ***pure demand paging***: Never bring a page into memory until it is required.

Locality of reference

- Theoretically some programs may access several new pages of memory causing multiple page faults
- Fortunately analysis of running processes shows programs tend to have ***locality of reference***

Restarting Execution

- A crucial requirement for demand paging is the ability **to restart any instruction** after a page fault.
- Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be **able to restart the process** in ***exactly the same place and*** state, except that the desired page is now in memory and is accessible

- In most cases, this requirement is easy to meet.
- A page fault may occur **at any memory reference.**
- If the page fault occurs on the **instruction fetch,** we can **restart by fetching the instruction again**.
- If a page fault occurs while we **are fetching an operand,** we must **fetch and decode the instruction again** and then **fetch the operand**

- As a worst-case example, consider a **three-address instruction** such as **ADD the content of A to B**, placing the **result in C**.
- These are the steps to execute this instruction:
- 1. Fetch and decode the instruction (ADD).
- 2. Fetch A.
- 3. Fetch B.
- 4. Add A and B.
- 5. Store the sum in C.
- If we **fault when we try to store in C** (because C is in a page not currently in memory), we will have to get the desired page, bring it in, correct the page table, and restart the instruction.

- The **restart will require fetching the instruction again**, **decoding it again**, **fetching the two operands again**, and then adding again.
- The major difficulty arises **when one instruction may modify several different locations**.
- For example, consider the **IBM System 360/370 MVC (move character) instruction**, which can move **up to 256 bytes** from one location to another (possibly overlapping) location.
- If either block (source or destination) **straddles a page boundary**, a **page fault might occur** after the move is partially done

- In addition, **if the source and destination blocks overlap**, the source block may have been modified, in which case we cannot simply restart the instruction.

Solutions

- This problem can be solved in **two different ways.**
- **Solution 1**
- In one solution, the **microcode computes** and **attempts to access both ends of both blocks (source & destination)**.
- If a page fault is going to occur, it will happen at this step, before anything is modified.
- The move can then take place; **we know that no page fault can occur,** since all the relevant pages are in memory.

- **Solution 2**
- The other solution **uses temporary registers** to hold the **values of overwritten locations**.
- If there is a page fault, all the **old values are written back into memory** before the trap occurs.
- This action **restores memory to its state** before the instruction was started, so that the instruction can be repeated.

zero-fill-on-demand

- Operating systems typically allocate these pages using a technique known as **zero-fill-on-demand.**
- Zero-fill-on-demand pages have been **zeroed-out** before being allocated, thus **erasing the previous contents.**
- Several versions of UNIX (including Solaris and Linux) provide a variation of the fork() system **call—vfork() (**for **virtual memory fork**)—that operates differently from fork() with **copy-on-write.**

- This technique is used in the VAX/VMS system along with a FIFO replacement algorithm.
- When the FIFO replacement algorithm mistakenly replaces a page that is still in active use, that page is quickly retrieved from the free-frame pool, and no I/O is necessary.

Allocation of Frames

- How do we allocate the **fixed amount of free memory** among the various processes?.
- If we have **93 free frames** and **two processes**, **how many frames** does each process get?
- The simplest case is the single-user system.
- Consider a single-user system with **128 KB** of memory composed of **pages 1 KB** in size.
- This system has **128 frames**. The **operating system** may take **35 KB**, leaving **93 frames** for the **user process.**
- Under **pure demand paging**, all 93 frames would initially be put on the **free-frame list**.
- When a user process started execution, it would generate a **sequence of page faults**.

- The **first 93 page faults** would all get free frames from the free-frame list.
- When the free-frame list was exhausted, a ***page-replacement algorithm*** would be used to select one of the 93 in-memory pages to be replaced with the 94^{th} page, and so on
- When the **process terminated**, the **93 frames** would once again be **placed on the free-frame list**.

- There are many **variations** on this simple strategy.
- **1).**We can require that the operating system allocate all its **buffer and table space** from the free-frame list.
- When this space is **not in use by the OS**, it can be used to support **user paging**.
- **2).** We can try to **keep three free frames reserved** on the free-frame list at all times.
- Thus, when a **page fault** occurs, **there is a free frame** available to page into.

Minimum Number of Frames

- Our **strategies for the allocation** of frames are **constrained** in various ways.
- We ***cannot allocate more than the total*** number of **available frames.**
- We must ***also allocate at least a minimum number*** **of frames**.
- One reason for allocating at least a minimum number of frames involves ***performance***.
- Obviously, as the **number of frames allocated** to each process **decreases**, the ***page-fault rate*** **increases**, slowing process execution.
- In addition, remember that, when a page fault occurs before an executing instruction is complete, the ***instruction must be restarted***.

- Consequently, we **must have enough frames** to hold all the different pages that ***any single instruction*** can reference.
- For example, consider a machine in which **all memory-reference instructions** have ***only one memory address***.
- In this case, we **need at least one frame for the instruction** and one **frame for the memory reference**.
- In addition, ***one-level indirect addressing*** may be allowed (for example, a load instruction on **page 16** can refer to an address on **page 0**, which contains an indirect reference to **page 23**).
- Then paging requires **at least three frames** per process.

- **The minimum number of frames** is defined by the **computer architecture**.
- For example, the ***move*** instruction for the **PDP-11** includes **more than one word** for some addressing modes, and thus the **instruction itself may straddle two pages.**
- In addition, **each of its two operands** may be **indirect references**, for a **total of six frames**.
- Another example is the **IBM 370 MVC** instruction.
- Since the instruction is from storage location to storage location, it **takes 6 bytes** and can **straddle two pages**

- The **block of characters to move** and the **area to which it is to be moved** can each also straddle two pages.
- The worst case occurs when the ***MVC*** instruction is the operand of an ***EXECUTE*** instruction that straddles a page boundary; in this case, we need ***eight frames***.

- The ***worst-case scenario occurs*** in computer architectures that allow multiple levels of ***indirection***.
- Theoretically, a **simple load instruction could reference an indirect address** that could reference **an indirect address** (on another page) that could also **reference an indirect address** (on yet another page), and so on until **every page in virtual memory had been touched.**
- To overcome this difficulty, we must place a ***limit on the levels of indirection***

- When the **first indirection occurs**, a ***counter*** is set to 16; the counter is then decremented for each successive indirection for this instruction. Tf the counter is decremented to 0, a trap occurs (excessive indirection).
- Whereas the minimum number of frames per process is defined by the architecture, the ***maximum number*** is defined by the **amount of available physical memory**.
- In between, we are still left with significant choice in frame allocation.

Allocation Algorithms

- The easiest way to split *in* frames among *n* processes is to give everyone an - equal share, *m/n* frames- ***Equal allocation***
- An alternative is to recognize that various processes will need differing amounts of memory.
- we can use **proportional** allocation, in which we allocate available memory to each process **according to its size**.
- With proportional allocation, we would split **62 frames between two processes**, one of **10 pages** and one of **127 pages**, by allocating **4 frames** and **57 frames**, respectively.

- In both equal and proportional allocation, of course, the allocation may vary according to the ***multiprogramming level***
- with either equal or proportional allocation, a ***high-priority*** process is treated the same as a low-priority process.
- One solution is to **use a proportional allocation** scheme wherein **the ratio of frames depends** not on the relative sizes of processes but rather on the **priorities of processes** or on a **combination of size and priority.**

Global versus Local Allocation

- Another important factor in the way frames are allocated to the various processes is **page replacement.**
- With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: **global replacement** and **local replacement.**
- **Global replacement** allows a process to **select a replacement frame** from the **set of all frames.**
- **Local replacement** requires that each process **select from only its own set** of **allocated frames**

- With a **local replacement strategy**, the **number of frames** allocated **to a process does not change**.
- With **global replacement**, a process may **happen to select only frames allocated to other processes**, thus **increasing the number of frames** allocated to it.
- One problem with a **global replacement** algorithm is that a **process cannot control its own page-fault rate**.
- **Global replacement** generally **results in greater system throughput** and is therefore the more common method.

Thrashing

- If the **number of frames** allocated to a **low-priority process** falls below the **minimum number required** by the **computer architecture**, we **must suspend**, that process's execution.
- We should then **page out** its remaining pages, **freeing all** its allocated frames.
- This provision **introduces a swap-in, swap-out** level of intermediate CPU scheduling.

- If the **process does not have the number of frames** it needs to support pages in active use, it will **quickly page-fault**.
- At this point, it must **replace some page**.
- However, since all its pages are in active use, it **must replace a page** that will be **needed again right away.**
- Consequently, it **quickly faults again**, **and again, and again**, replacing pages that it must bring back in immediately.
- This **high paging activity** is called **thrashing.**
- A process is thrashing if it is **spending more time paging than executing**.

Cause of Thrashing

- Thrashing results in **severe performance problems**.
- The operating system **monitors CPU utilization**.
- **If CPU utilization is too low**, it increases the **degree of multiprogramming** by introducing a **new process to the system.**
- A **global page-replacement algorithm** is used; it replaces pages without regard to the process to which they belong.
- Now suppose that a **process enters a new phase** in its execution and **needs more frames**.

- It **starts faulting** and **taking frames away** from other processes.
- These processes need those pages, however, and **so they also fault**, taking frames from other processes.
- These faulting processes must use the paging device to swap pages in and out.
- As they queue up for the paging device, the **ready queue empties.**
- As processes wait for the paging device, **CPU utilization decreases.**

- **Thrashing has occurred**, and system **throughput plunges.**
- The **page fault rate increases** tremendously
- As a result, the effective **memory-access time increases.**
- No work is getting done, because the processes are spending all their time paging.

Solutions

- We can limit the effects of thrashing by using a **local replacement algorithm** (or **priority replacement algorithm).**
- With local replacement, if one process starts thrashing, it **cannot steal frames from another process** and cause the latter to thrash as well.
- However, the problem is **not entirely solved**.
- If processes are thrashing, they **will be in the queue for the paging device** most of the time.
- The **average service time** for a **page fault will increase** because of the longer average queue for the paging device.
- Thus, the **effective access time will increase** even for a **process that is not thrashing**.

- To prevent thrashing, we must **provide a process with as many frames as it needs**.
- But how do we know **how many frames it "needs**'?
- There are **several techniques**.
- The **working-set strategy** starts by looking at how many frames a process is actually using.
- This approach defines the **locality model** of process execution.
- The locality model states that, as a process executes, it moves from locality to locality.
- A **locality is a set of pages** that are **actively used together**

- A **program is generally composed** of several different **localities,** which **may overlap**.
- For example, when a **function is called**, it defines a **new locality.**
- In this locality, memory references are made to the **instructions of the function** call, its **local variables**, and a **subset of the global variables**.
- When we **exit the function,** the process **leaves this locality,** since the local variables and instructions of the function are no longer in active use.
- Thus, we see that **localities are defined by the program structure** and **its data structures**.
- The locality model states that all programs will exhibit this basic memory reference structure.