Memory Management

MAIN MEMORY

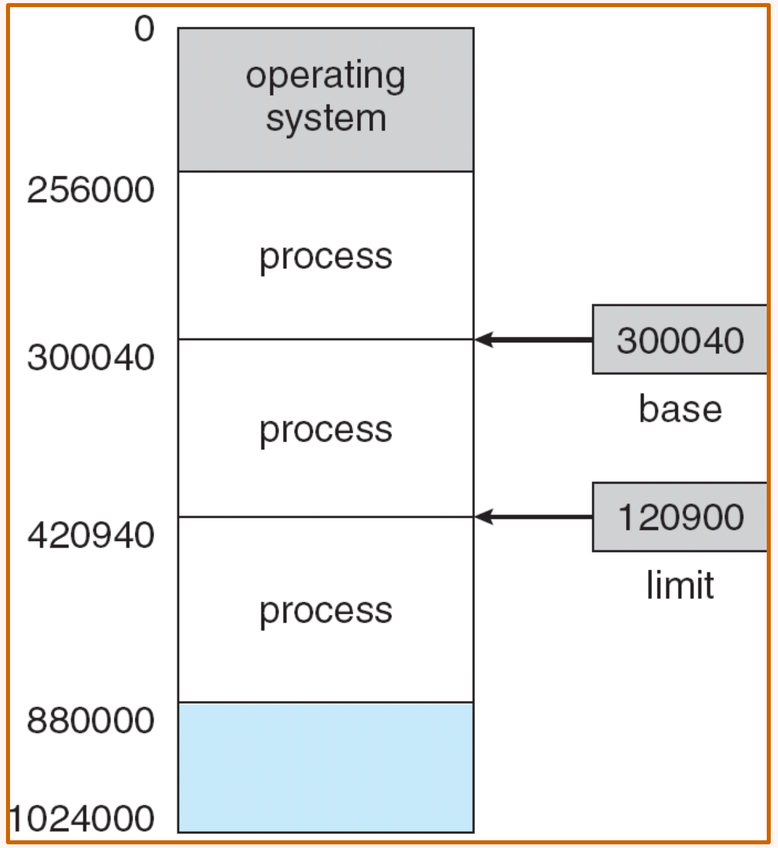
OBJECTIVES

- Background
- Swapping
- Contiguous memory allocation
- Segmentation
- Paging

BACKGROUND

BASIC HARDWARE

- To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
- We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure.
- The base register holds the smallest legal physical memory address; the limit register specifies the size of the range.
- Hece, the program can access all addresses from 300040 through 420939(inclusive).

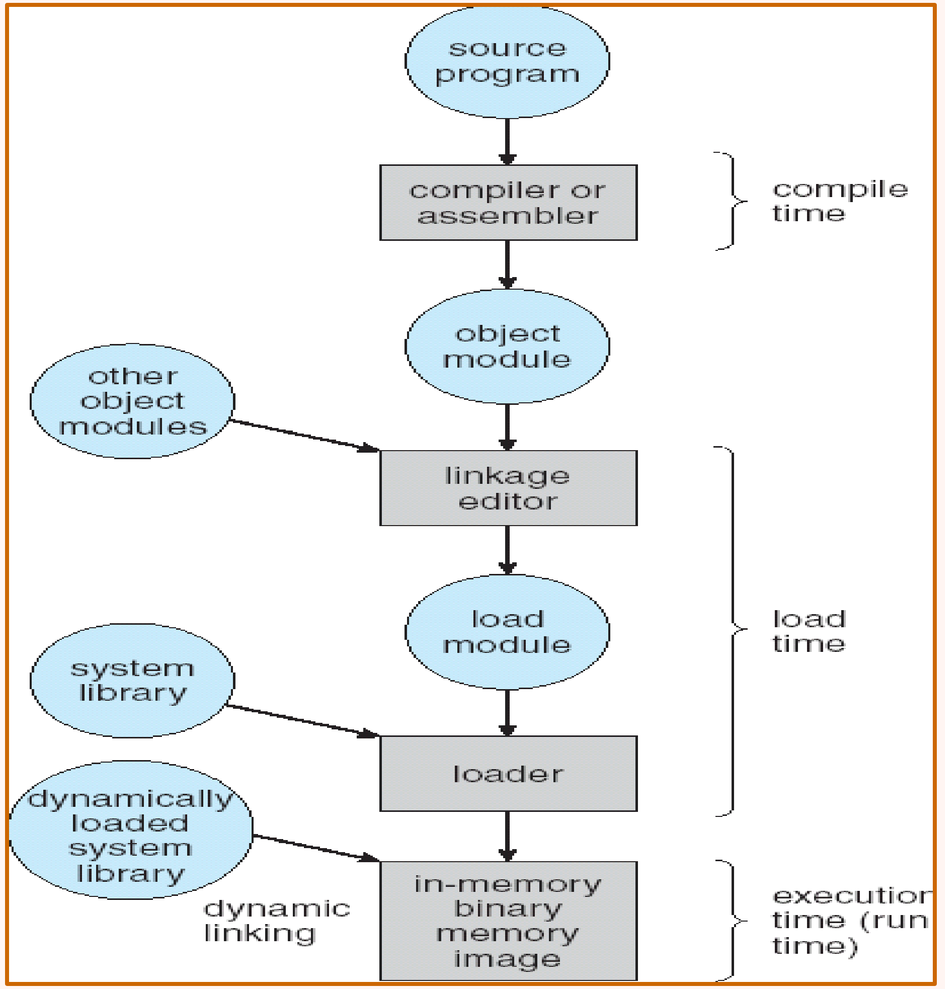


ADDRESS BINDING

- Address binding is the process of mapping from one address space to another address space.

- Logical address is address generated by CPU during execution whereas Physical Address refers to location in memory unit(the one that is loaded into memory).

- Note that user deals with only logical address(Virtual address). The logical address undergoes translation by the MMU or address translation unit in particular. The output of this process is the appropriate physical address or the location of code/data in RAM.



BINDING CAN TAKE PLACE AT ANY OF THE FOLLOWING STEPS

- **Compile Time:** If you know 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.

- **Loadtime:** If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.

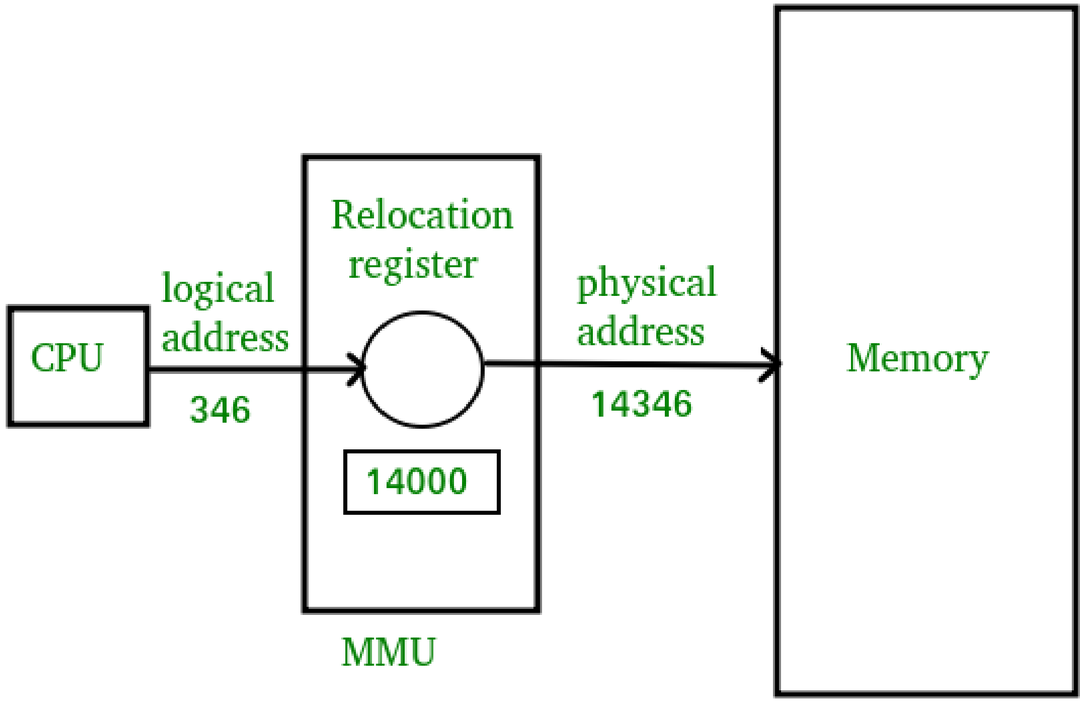
- **Execution time:** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work. Most general-purpose operating systems use this method.

DIFFERENCES BETWEEN LOGICAL AND PHYSICAL

- The basic difference between Logical and physical address is that Logical address is generated by CPU in perspective of a program whereas the physical address is a location that exists in the memory unit.
- Logical Address Space is the set of all logical addresses generated by CPU for a program whereas the set of all physical address mapped to corresponding logical addresses is called Physical Address Space.
- The logical address does not exist physically in the memory whereas physical address is a location in the memory that can be accessed physically.
- Identical logical addresses are generated by Compile-time and Load time address binding methods whereas they differs from each other in run-time address binding method. Please refer this for details.
- The logical address is generated by the CPU while the program is running whereas the physical address is computed by the Memory Management Unit (MMU).

ADDRESS TRANSLATION FROM LOGICAL AND PHYSICAL ADDRESSES IN CONTIGOUS ALLOCATION

- The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU).
- 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 (see Figure).
- For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.
- Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.



DYNAMIC LOADING

- To obtain better memory-space utilisation, 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 relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.
- This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, 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 the users to design their programs to take advantage of such a method. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

STATIC LINKING

- When we click the .exe (executable) file of the program and it starts running, all the necessary contents of the binary file have been loaded into the process's virtual address space. However, most programs also need to run functions from the system libraries, and these library functions also need to be loaded.

- In the simplest case, the necessary library functions are embedded directly in the program's executable binary file. Such a program is statically linked to its libraries, and statically linked executable codes can commence running as soon as they are loaded.

- Disadvantage:
  Every program generated must contain copies of exactly the same common system library functions. In terms of both physical memory and disk-space usage, it is much more efficient to load the system libraries into memory only once. Dynamic linking allows this single loading to happen.

DYNAMIC LINKING

- Linking postponed until execution time
- Small piece of code, stub, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as shared libraries

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 libraries are treated like any other object module and are combined by the loader into the binary program image during compile time.

- Dynamic linking, in contrast, is similar to dynamic loading. Here, though, linking, rather than loading, is postponed until execution time.

- 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. This requirement wastes both disk space and main memory.

- 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 it is not, the program loads the routine into memory.

- 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.

STATIC LOADING AND DYNAMIC LOADING

- Loading the entire program into the main memory before start of the program execution is called as static loading.
- Inefficient utilisation of memory because whether it is required or not required entire program is brought into the main memory.
- Its faster as each program has its routines are already loaded into the memory

- Loading the program into the main memory on demand is called as dynamic loading.
- Efficient utilisation of memory.
- It is slower as it requires to load routines whenever need by any section of the program.

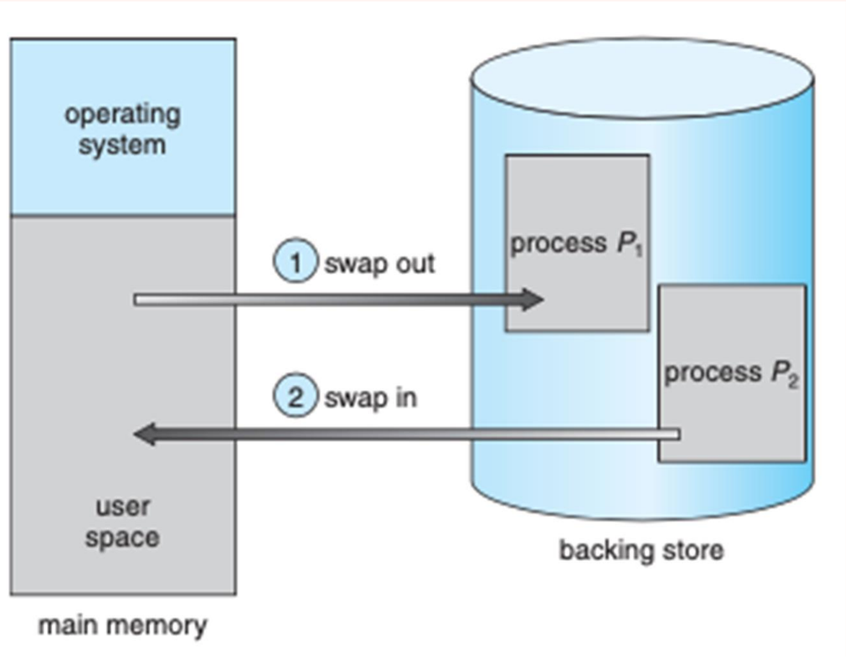
STATIC LINKING AND DYNAMIC LINKING

Static linking is performed by programs called linkers as the last step in compiling a program. Linkers are also called link editors.	In dynamic linking this is not the case and individual shared modules can be updated and recompiled. This is one of the greatest advantages dynamic linking offers.
In static linking if any of the external programs has changed then they have to be recompiled and re-linked again else the changes won't reflect in existing executable file.	In dynamic linking load time might be reduced if the shared library code is already present in memory.

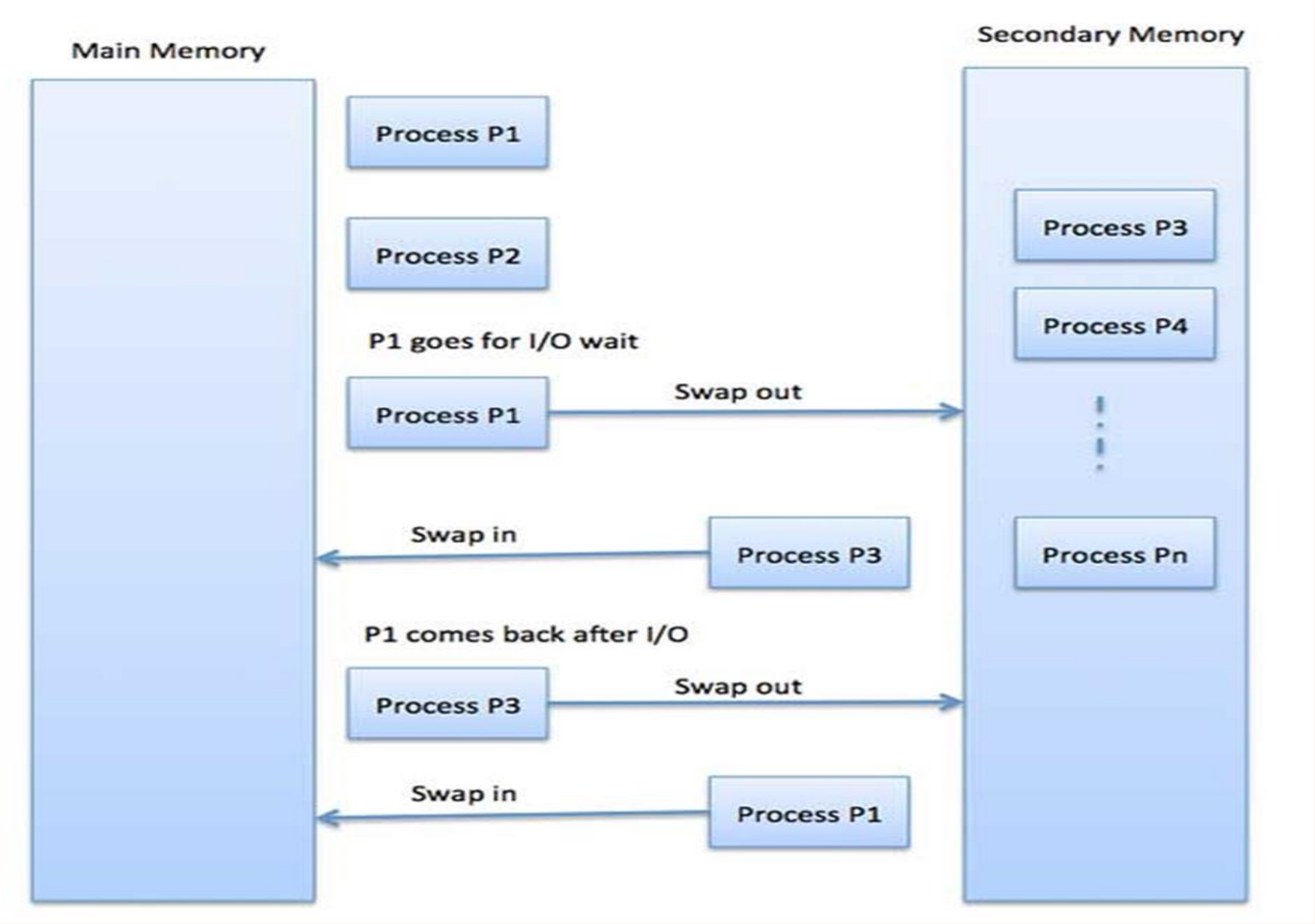
SWAPPING

SWAPPING

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Plus disk latency of 8 ms
  - Swap out time of 2008 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4016ms (> 4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request memory` and `release memory`



SWAPPING


Main Memory
Secondary Memory
Process P1
Process P2
Process P3
Process P4
P1 goes for I/O wait
Process P1
Swap out
Swap in
Process P3
Process Pn
P1 comes back after I/O
Process P3
Swap out
Swap in
Process P1

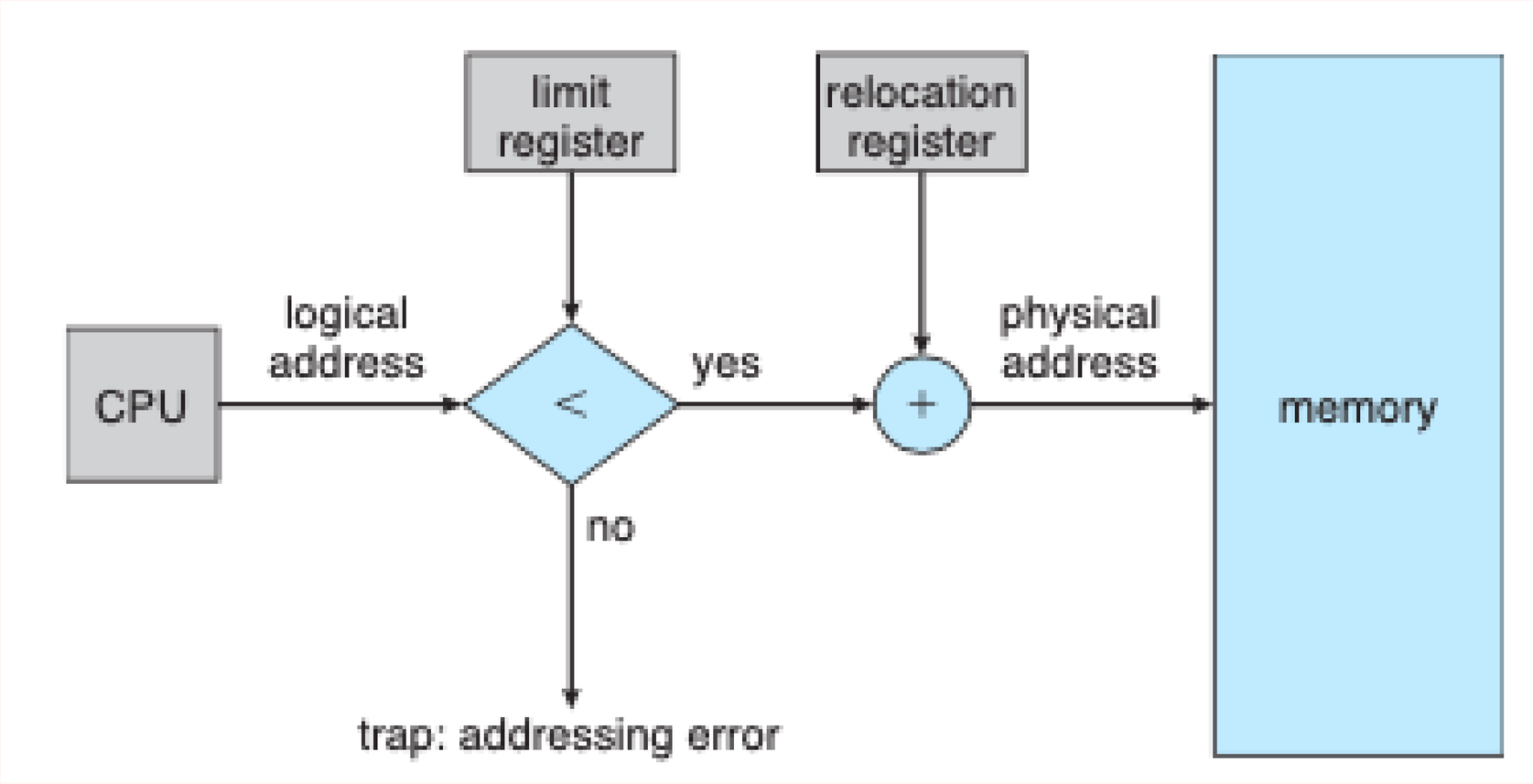
CONTIGUOUS MEMORY ALLOCATION

MEMORY ALLOCATION

Main memory usually has two partitions −
Low Memory − Operating system resides in this memory.
High Memory − User processes are held in high memory.
Operating system uses the following memory allocation mechanism.

S.N.	Memory Allocation & Description
1	**Single-partition allocation** In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register.
2	**Multiple-partition allocation** In this type of allocation, main memory is divided into a number of fixed-sized partitions where each partition should contain only one process. 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.

ADDRESS TRANSLATION FROM LOGICAL TO PHYSICAL ADDRESS IN CONTIGUOUS ALLOCATION



CONTIGUOUS MEMORY ALLOCATION

- The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. The one way is contiguous memory allocation.

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

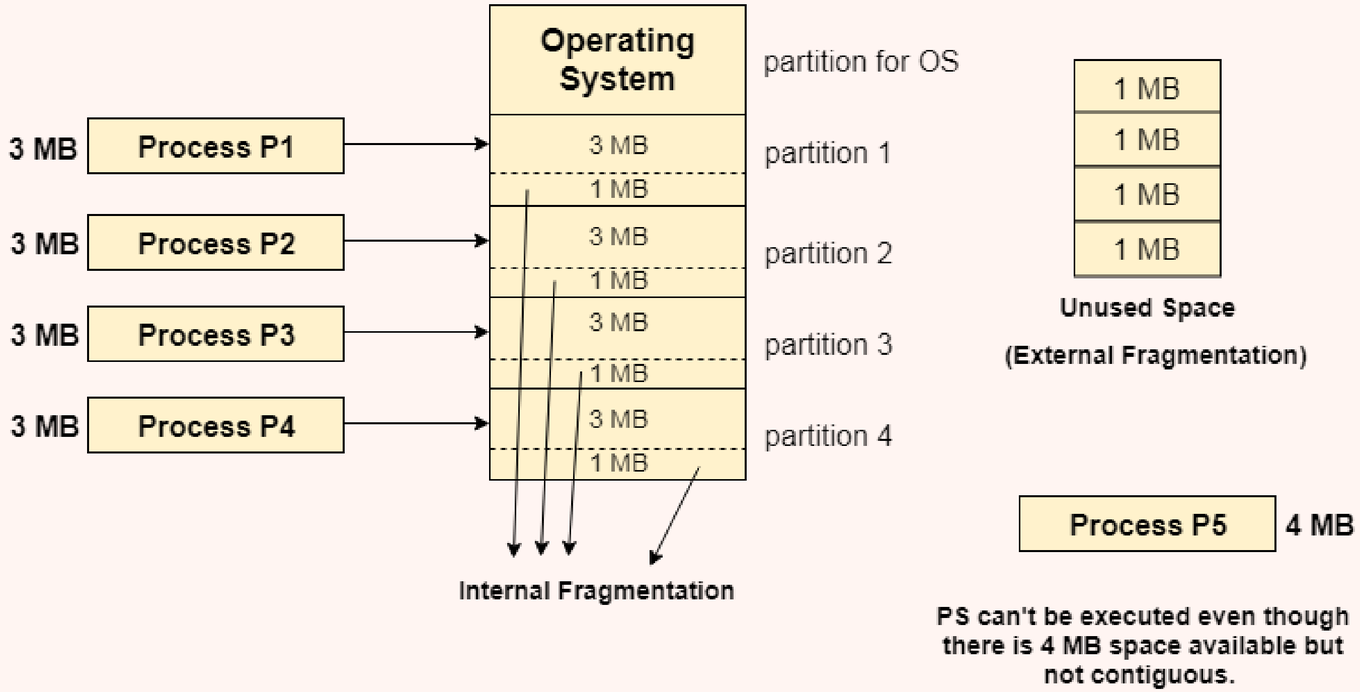
- We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. **In contiguous memory allocation, each process is contained in a single section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

MEMORY ALLOCATION

- It divides the 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- partition 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.

FIXED PARTITIONING



Fixed Partitioning

(Contiguous memory allocation)

PROBLEMS WITH FIXED PARTITIONING SCHEME

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

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

- **Limitation on the size of the process:** If the process size is larger than the size of maximum sized partition than that process cannot be loaded into the memory. Therefore, a limitation can be imposed on the process size that is it cannot be larger than the size of the largest partition.

- **Degree of multiprogramming is less:** By Degree of multi programming, we simply mean the maximum number of processes that can be loaded into the memory at the same time. In fixed partitioning, the degree of multiprogramming is fixed and very less due to the fact that the size of the partition cannot be varied according to the size of processes.

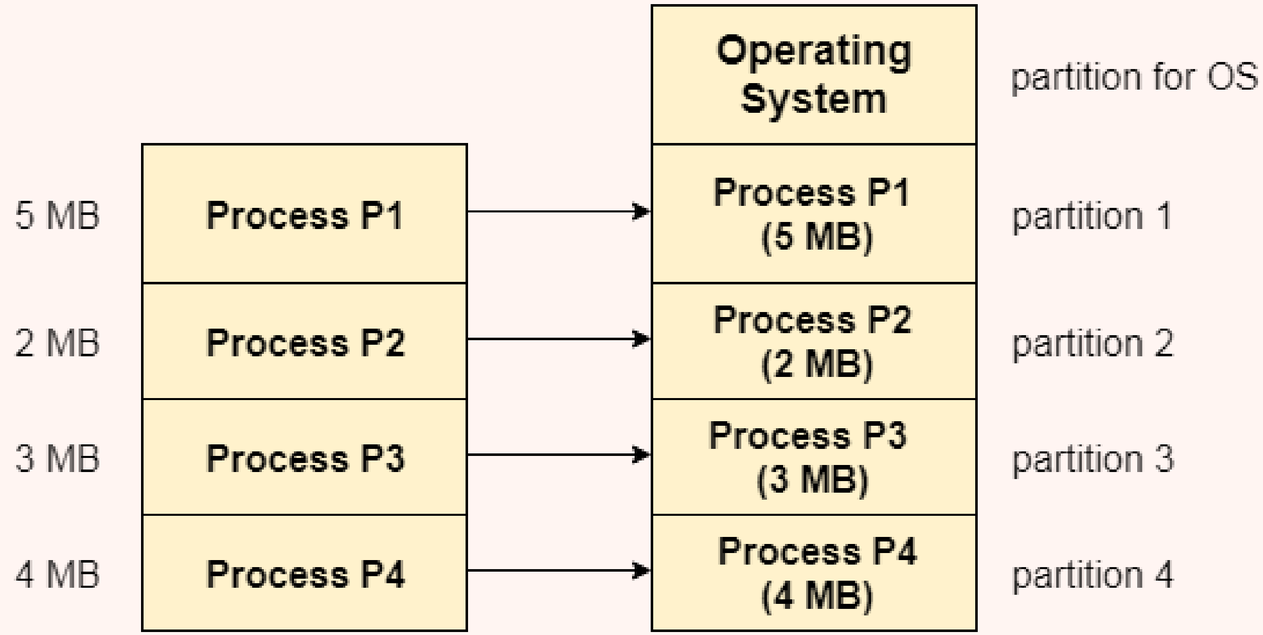
DYNAMIC PARTITIONING SCHEME

- In the variable-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a hole. Eventually, as you will see, memory contains a set of holes of various sizes.

- Dynamic partitioning tries to overcome the problems caused by fixed partitioning. In this technique, the partition size is not declared initially. It is declared at the time of process loading.

- The first partition is reserved for the operating system. The remaining space is divided into parts. The size of each partition will be equal to the size of the process. The partition size varies according to the need of the process so that the internal fragmentation can be avoided

DYNAMIC PARTITIONING

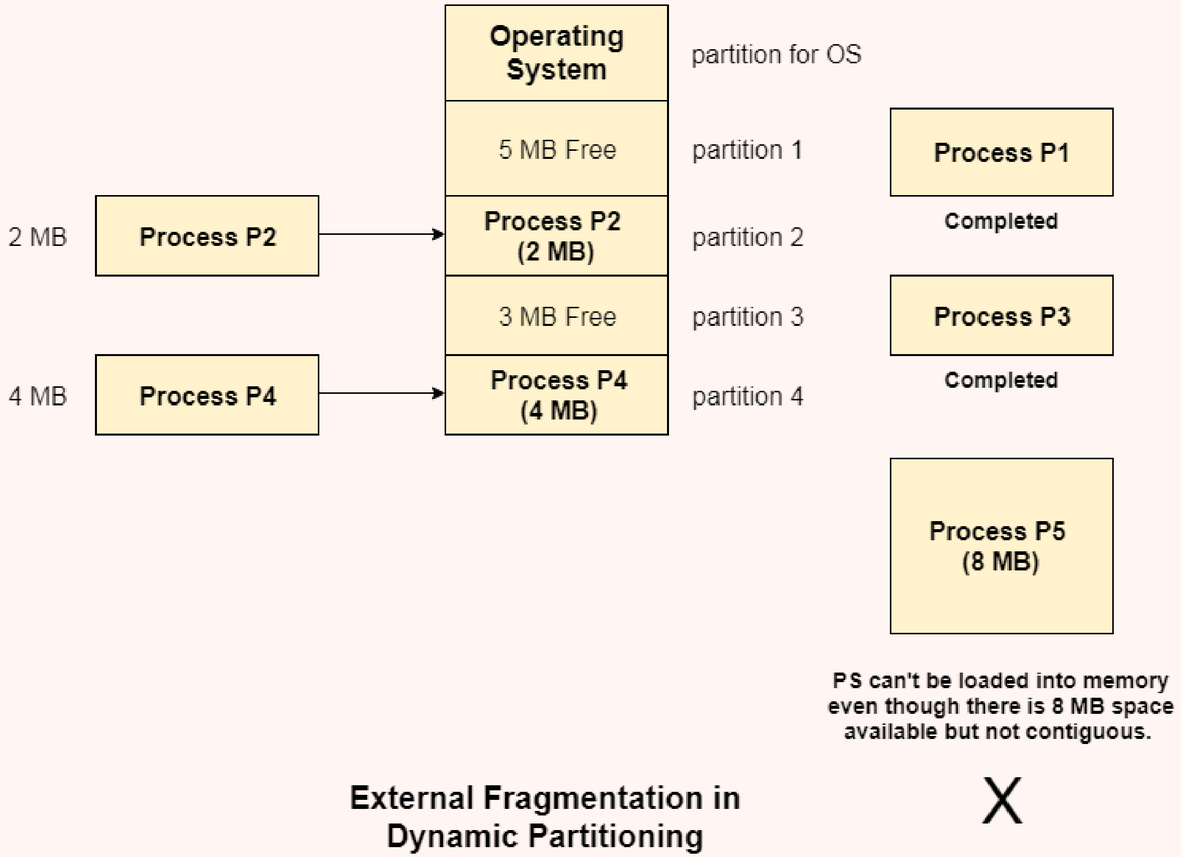


Dynamic Partitioning

(Process Size = Partition Size)

EXTERNAL FRAGMENTATION OCCURS

- **After some time P1 and P3 got completed and their assigned space is freed. Now there are two unused partitions (5 MB and 3 MB) available in the main memory but they cannot be used to load a 8 MB process in the memory since they are not contiguously located.**



External Fragmentation in Dynamic Partitioning

COMPACTION- SOLUTION TO 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.
- Usually the remaining processes are moved to the top and the free spaces are brought together to form a single large memory unit i.e. contiguous memory.

MEMORY ALLOCATION ALGORITHMS

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

- **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.

- **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.

FIRST-FIT

- In this method, first job claims the first available memory with space more than or equal to it’s size. The operating system doesn’t search for appropriate partition but just allocate the job to the nearest memory partition available with sufficient size.

Job Number	Memory Requested
J1	20 K
J2	200 K
J3	500 K
J4	50 K

Memory location	Memory block size	Job number	Job size	Status	Internal fragmentation
10567	200 K	J1	20 K	Busy	180 K
30457	30 K			Free	30
300875	700 K	J2	200 K	Busy	500 K
809567	50 K	J4	50 K	Busy	None
Total available :	980 K	Total used :	270 K		710 K

WORST-FIT

- In this allocation technique the process traverse the whole memory and always search for largest hole/partition, and then the process is placed in that hole/partition.It is a slow process because it has to traverse the entire memory to search largest hole.

Process Number	Process Size
P1	30K
P2	100K
P3	45K

MEMORY LOCATION	MEMORY BLOCK SIZE	PROCESS NUMBER	PROCESS SIZE	STATUS	INTERNAL FRAGMENTATION
12345	50K	P3	45K	Busy	5K
45871	100K	P2	100K	Busy	None
1245	400K	P1	30K	Busy	370K
TOTAL AVAILABLE:	550K	TOTAL USED:	175K		375K

BEST-FIT

- This method keeps the free/busy list in order by size – smallest to largest. In this method, the operating system first searches the whole of the memory according to the size of the given job and allocates it to the closest-fitting free partition in the memory, making it able to use memory efficiently. Here the jobs are in the order from smalest job to largest job.

Job Number	Memory Requested
J1	20 K
J2	200 K
J3	500 K
J4	50 K

Memory location	Memory block size	Job number	Job size	Status	Internal fragmentation
10567	30 K	J1	20 K	Busy	10 K
30457	50 K	J4	50 K	Busy	None
300875	200 K	J2	200 K	Busy	None
809567	700 K	J3	500 K	Busy	200 K
Total available :	980 K	Total used :	770 K		210 K

PROBLEMS

Que. 1. Consider a variable partition memory management scheme. Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)?
Which algorithm makes the most efficient use of memory?

Let p1, p2, p3 & p4 are the names of the processes

a. First-fit:

P1>>> 100, 500, 200, 300, 600
P2>>> 100, 288, 200, 300, 600
P3>>> 100, 288, 200, 300, 183
100, 116, 200, 300, 183 <<<<< final set of hole
P4 (426K) must wait

b. Best-fit:

P1>>> 100, 500, 200, 300, 600
P2>>> 100, 500, 200, 88, 600
P3>>> 100, 83, 200, 88, 600
P4>>> 100, 83, 88, 88, 600
100, 83, 88, 88, 174 <<<<< final set of hole

c. Worst-fit:

P1>>> 100, 500, 200, 300, 600
P2>>> 100, 500, 200, 300, 388
P3>>> 100, 83, 200, 300, 388
100, 83, 200, 300, 276 <<<<< final set of hole
P4 (426K) must wait

In this example, Best-fit turns out to be the best because there is no wait processes

SOLUTION

a. First-fit:

b. 212K is put in 500K partition

c. 417K is put in 600K partition

d. 112K is put in 288K partition (new partition 288K = 500K - 212K)

e. 426K must wait

f. Best-fit:

g. 212K is put in 300K partition

h. 417K is put in 500K partition

i. 112K is put in 200K partition

j. 426K is put in 600K partition

k. Worst-fit:

l. 212K is put in 600K partition

m. 417K is put in 500K partition

n. 112K is put in 388K partition

o. 426K must wait

In this example, Best-fit turns out to be the best.

SEGMENTATION

SEGMENTATION

- In Operating Systems, Segmentation is a memory management technique in which, the memory is divided into the variable size parts. Each part is known as segment which can be allocated to a process.

- The details about each segment are stored in a table called as segment table. Segment table is stored in one (or many) of the segments.

- Segment table contains mainly two information about segment:

  - Base: It is the base address of the segment.

  - Limit: It is the length of the segment.

- For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a *two tuple:*

  <segment-number, offset>

- We must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses. This mapping is effected by a segment table. Each entry in the segment table has a segment base and a segment limit. The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.

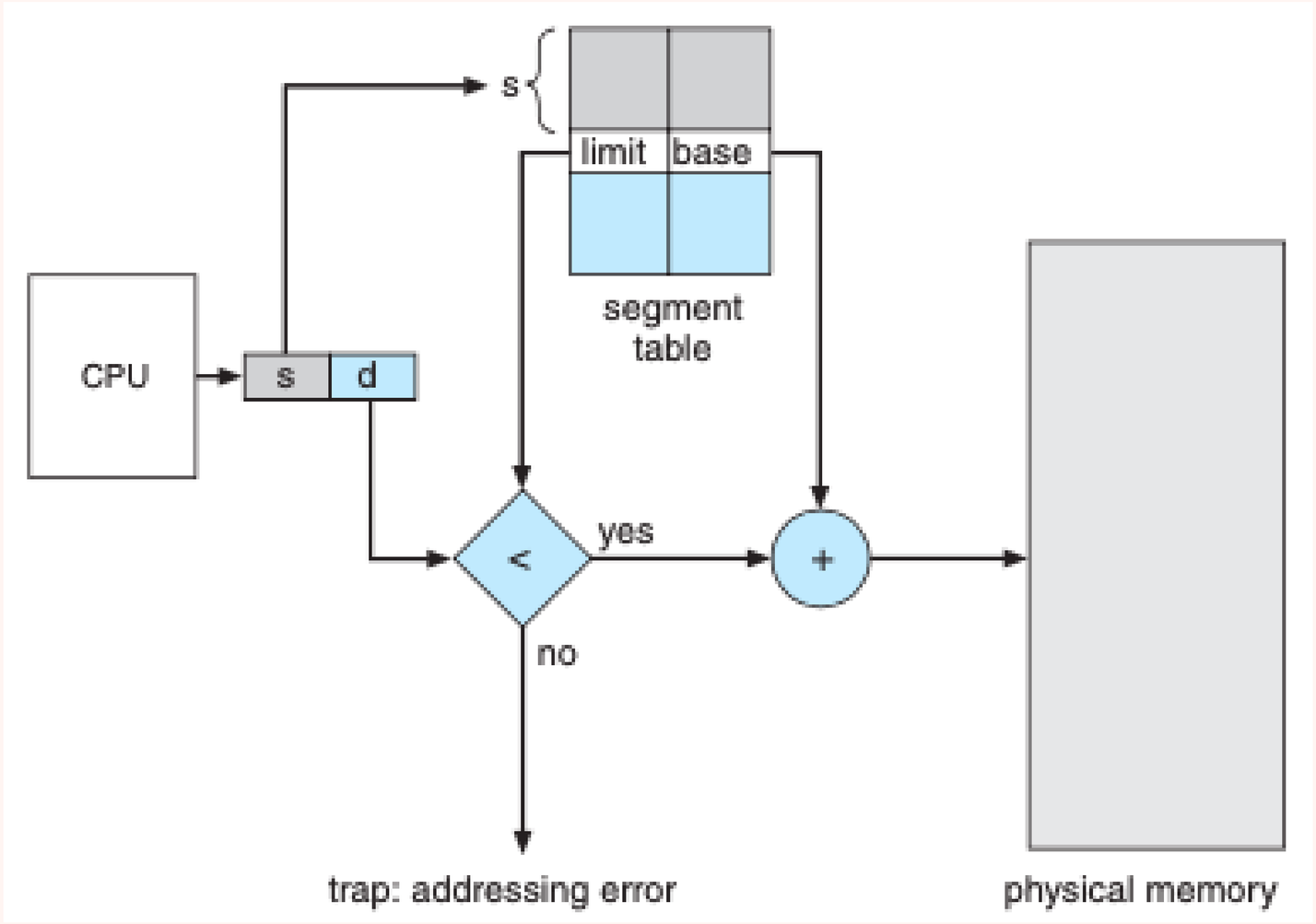
The use of a segment table is illustrated in Figure.

A logical address consists of two parts: a segment number, *s,* and an offset into that segment, *d.*

The segment number is used as an index to the segment table.

The offset *d* of the logical address must be between 0 and the segment limit.

If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

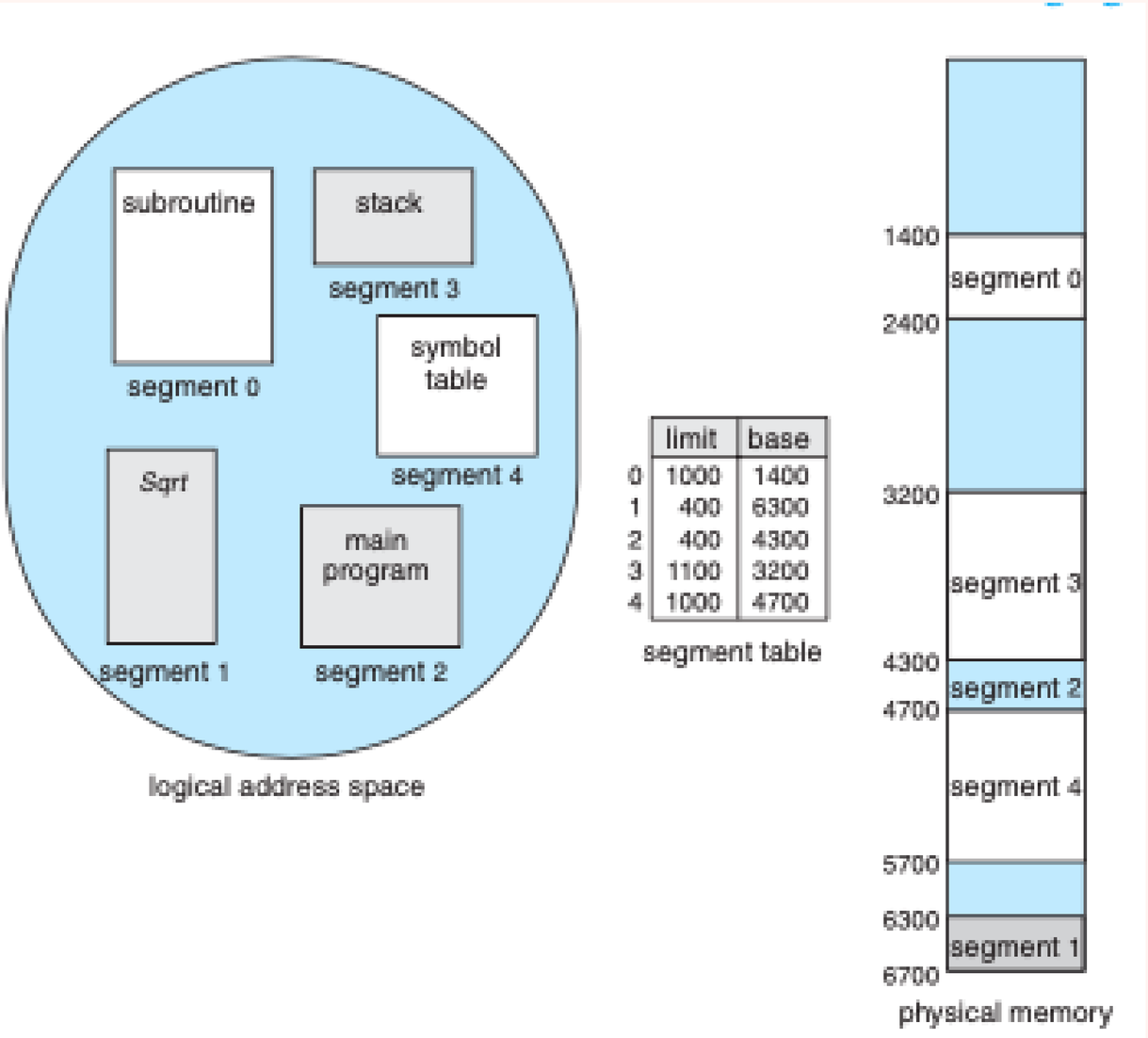


As an example, consider the situation shown in Figure.

We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown.

The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit).

For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.



Q1.

Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

a. 0,430

b. 1,10

c. 2,500

d. 3,400

e. 4,112

Answer:

a. 219 + 430 = 649

b. 2300 + 10 = 2310

c. illegal reference, trap to operating system

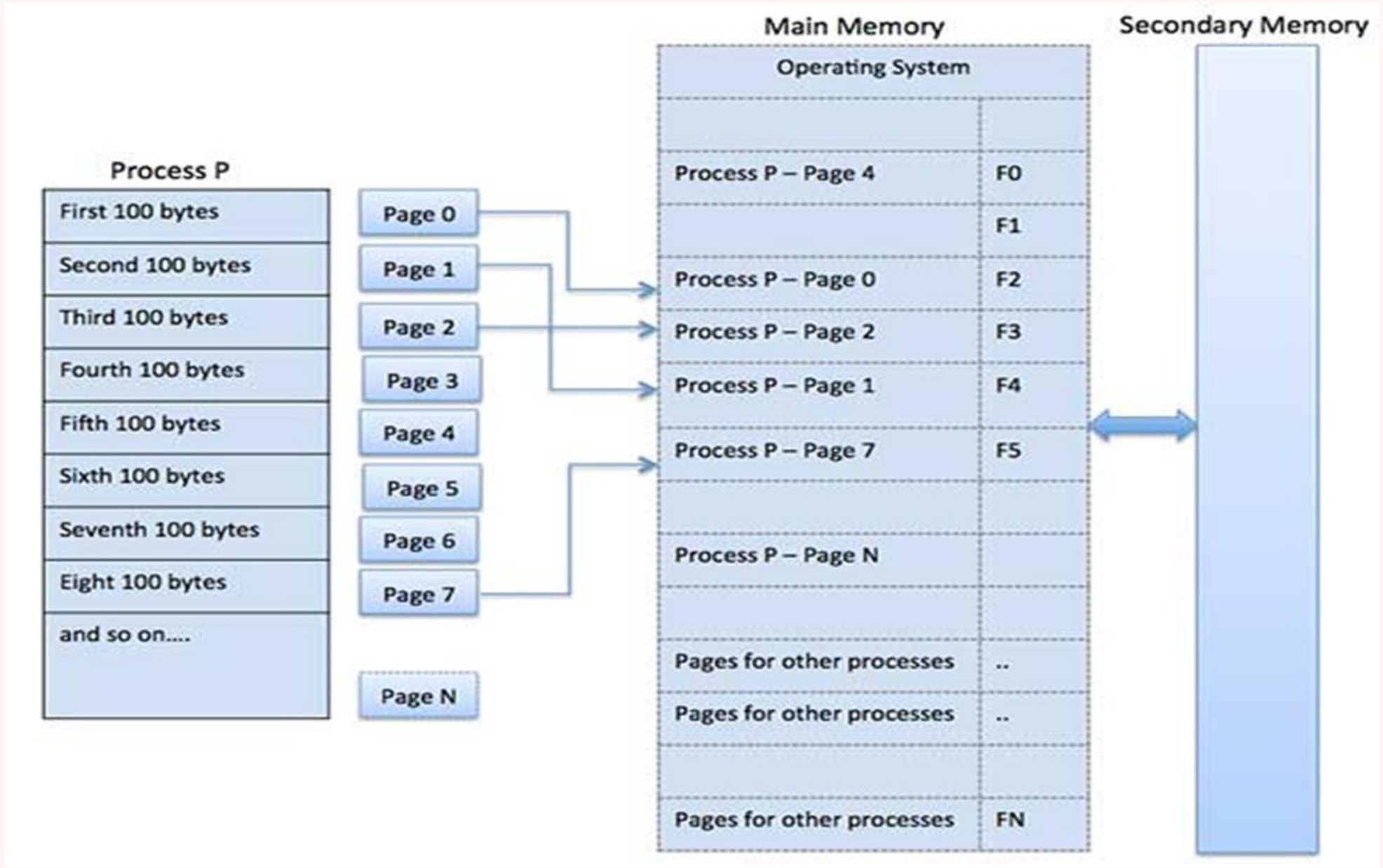
d. 1327 + 400 = 1727

e. illegal reference, trap to operating system

PAGING

PAGING

- Paging technique plays an important role in implementing virtual memory.
- Paging is a memory management technique in which process address space is broken into blocks of the same size called pages (size is power of 2). The size of the process is measured in the number of pages.
- Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called frames and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.


Main Memory
Secondary Memory
Process P
First 100 bytes
Second 100 bytes
Third 100 bytes
Fourth 100 bytes
Fifth 100 bytes
Sixth 100 bytes
Seventh 100 bytes
Eight 100 bytes
and so on....
Page 0
Page 1
Page 2
Page 3
Page 4
Page 5
Page 6
Page 7
Page N
Operating System
Process P – Page 4
F0
F1
Process P – Page 0
F2
Process P – Page 2
F3
Process P – Page 1
F4
Process P – Page 7
F5
Process P – Page N
Pages for other processes
..
Pages for other processes
..
Pages for other processes
FN

PAGING

- Whenever a process arrives in the system for execution, its size is expressed in pages.
- Each page requires one frame in main memory.
- The corresponding information of each page coming into main memory is updated into Page Table.
- There are chances of internal fragmentation. (How ?)
- Page Table is Indexed with the page number.
- Page Table stores the base address or Index number of corresponding frames.

PAGING

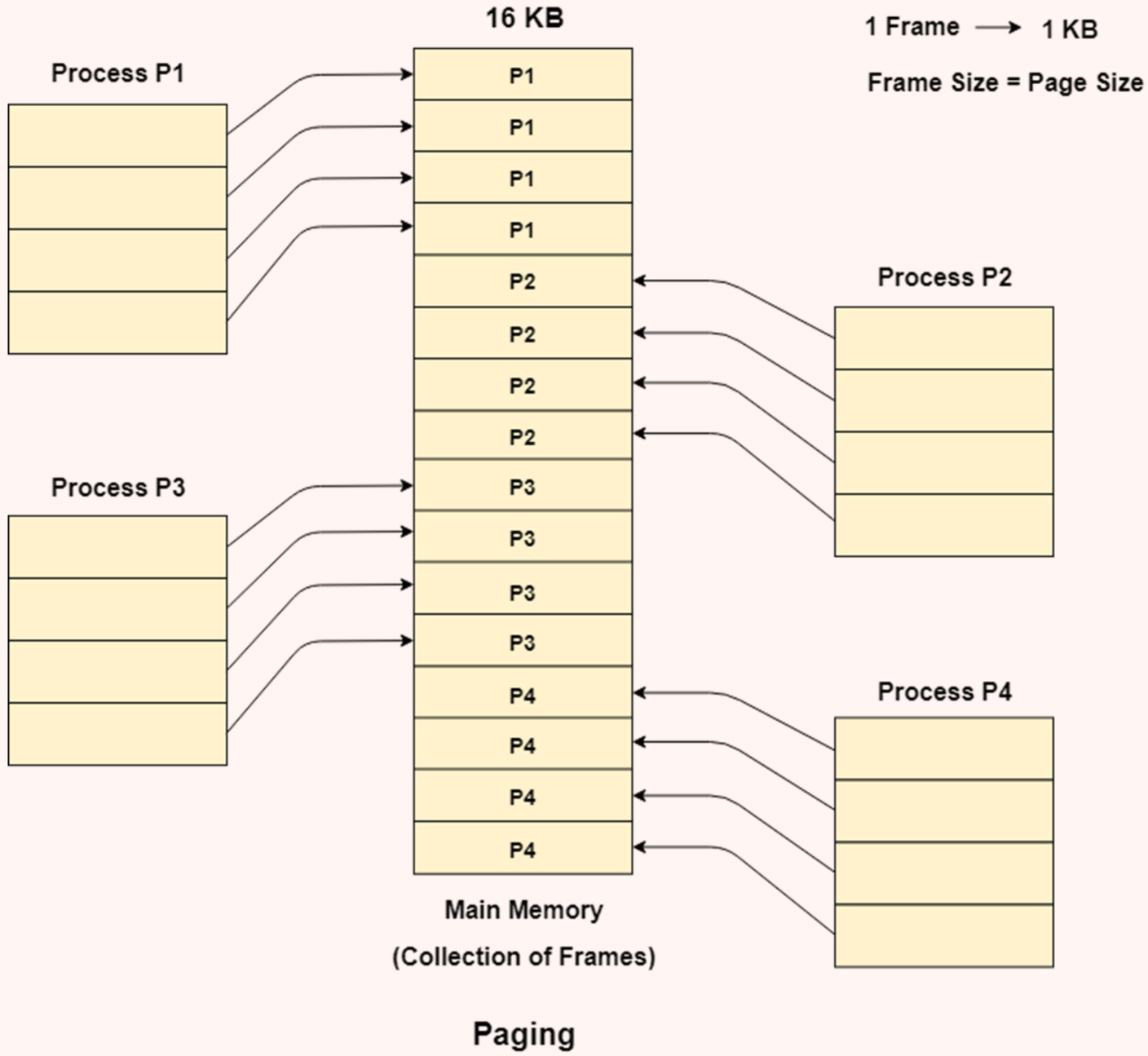
- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size called pages
- Keep track of all free frames
- To run a program of size n pages, need to find n free frames and load program
- Set up a page table to translate logical to physical addresses
- Internal fragmentation

Let us consider the main memory size 16 Kb and Frame size is 1 KB therefore the main memory will be divided into the collection of 16 frames of 1 KB each.

There are 4 processes in the system that is P1, P2, P3 and P4 of 4 KB each. Each process is divided into pages of 1 KB each so that one page can be stored in one frame.

Initially, all the frames are empty therefore pages of the processes will get stored in the contiguous way.

Frames, pages and the mapping between the two is shown in the image below.



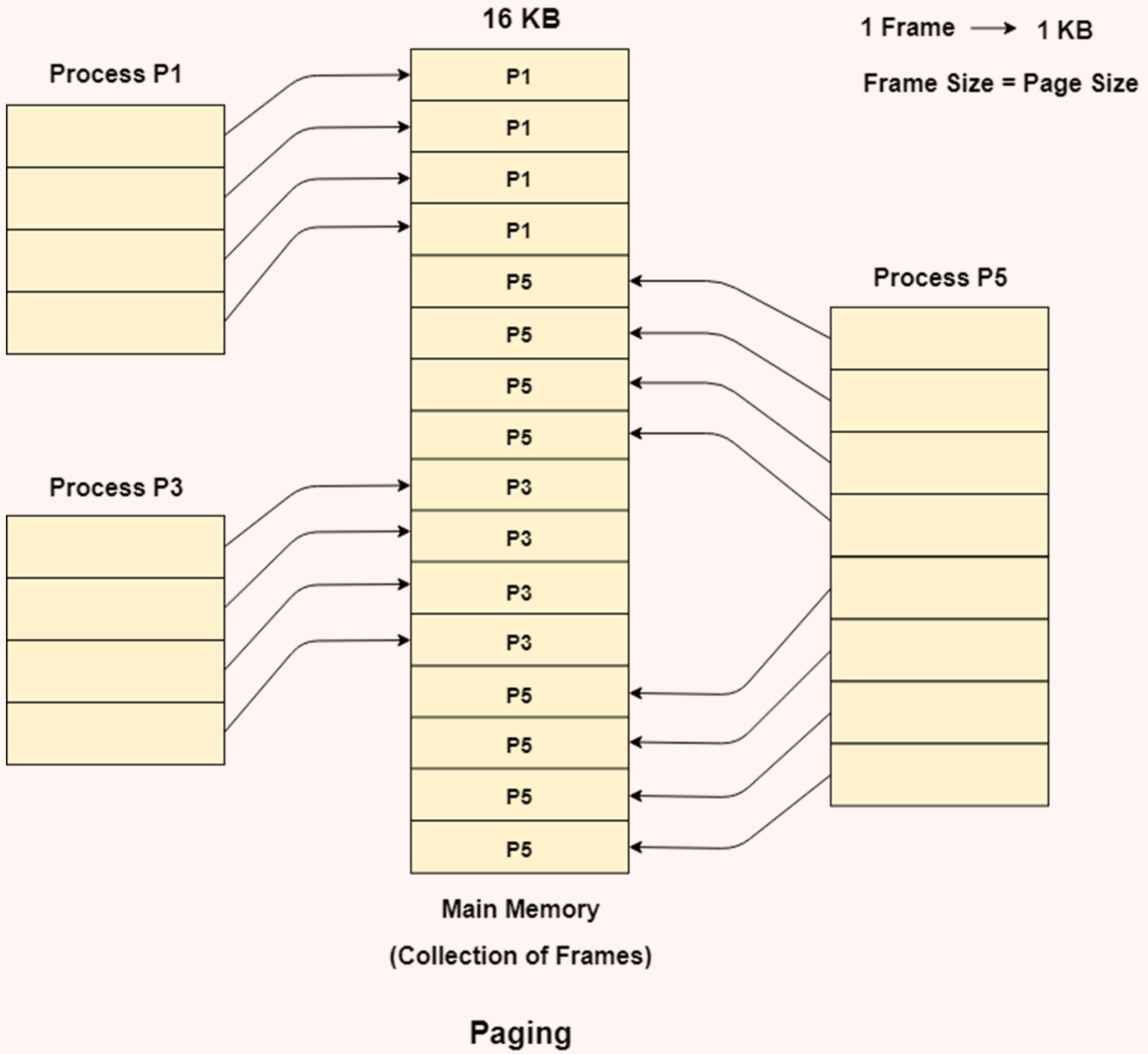
Paging

Let us consider that, P2 and P4 are moved to waiting state after some time. Now, 8 frames become empty and therefore other pages can be loaded in that empty place.

The process P5 of size 8 KB (8 pages) is waiting inside the ready queue.

Given the fact that, we have 8 non contiguous frames available in the memory and paging provides the flexibility of storing the process at the different places.

Therefore, we can load the pages of process P5 in the place of P2 and P4.



Paging

MMU

The purpose of Memory Management Unit (MMU) is to convert the logical address into the physical address. The logical address is the address generated by the CPU for every page while the physical address is the actual address of the frame where each page will be stored.

When a page is to be accessed by the CPU by using the logical address, the operating system needs to obtain the physical address to access that page physically.

The logical address has two parts.
1.Page Number
2.Offset
Memory management unit of OS needs to convert the page number to the frame number.

Basic of Binary Addresses

Computer system assigns the binary addresses to the memory locations. However, The system uses amount of bits to address a memory location.

Using 1 bit, we can address two memory locations. Using 2 bits we can address 4 and using 3 bits we can address 8 memory locations.

A pattern can be identified in the mapping between the number of bits in the address and the range of the memory locations.

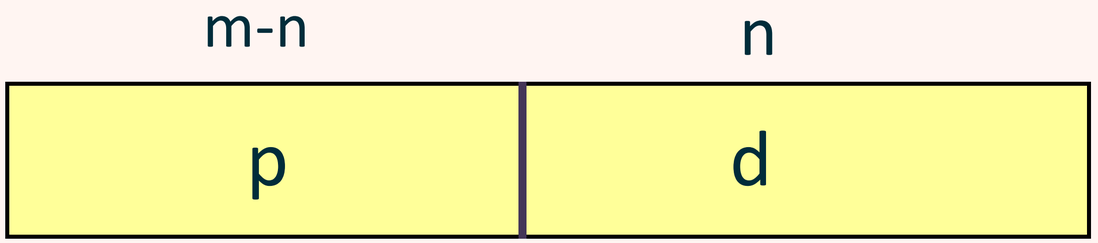
Using 1 Bit we can represent 2^1 i.e 2 memory locations.
Using 2 bits, we can represent 2^2 i.e. 4 memory locations.
Using 3 bits, we can represent 2^3 i.e. 8 memory locations.
Therefore, if we generalize this,
Using n bits, we can assign 2^n memory locations.

n bits of address → 2 ^ n memory locations

these n bits can be divided into two parts, that are, K bits and (n-k) bits.

ADDRESS TRANSLATION SCHEME

- Address generated by CPU is divided into:
  - **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory
  - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit



  - For given logical address space 2^m *and page size* 2^n

PAGE SIZE

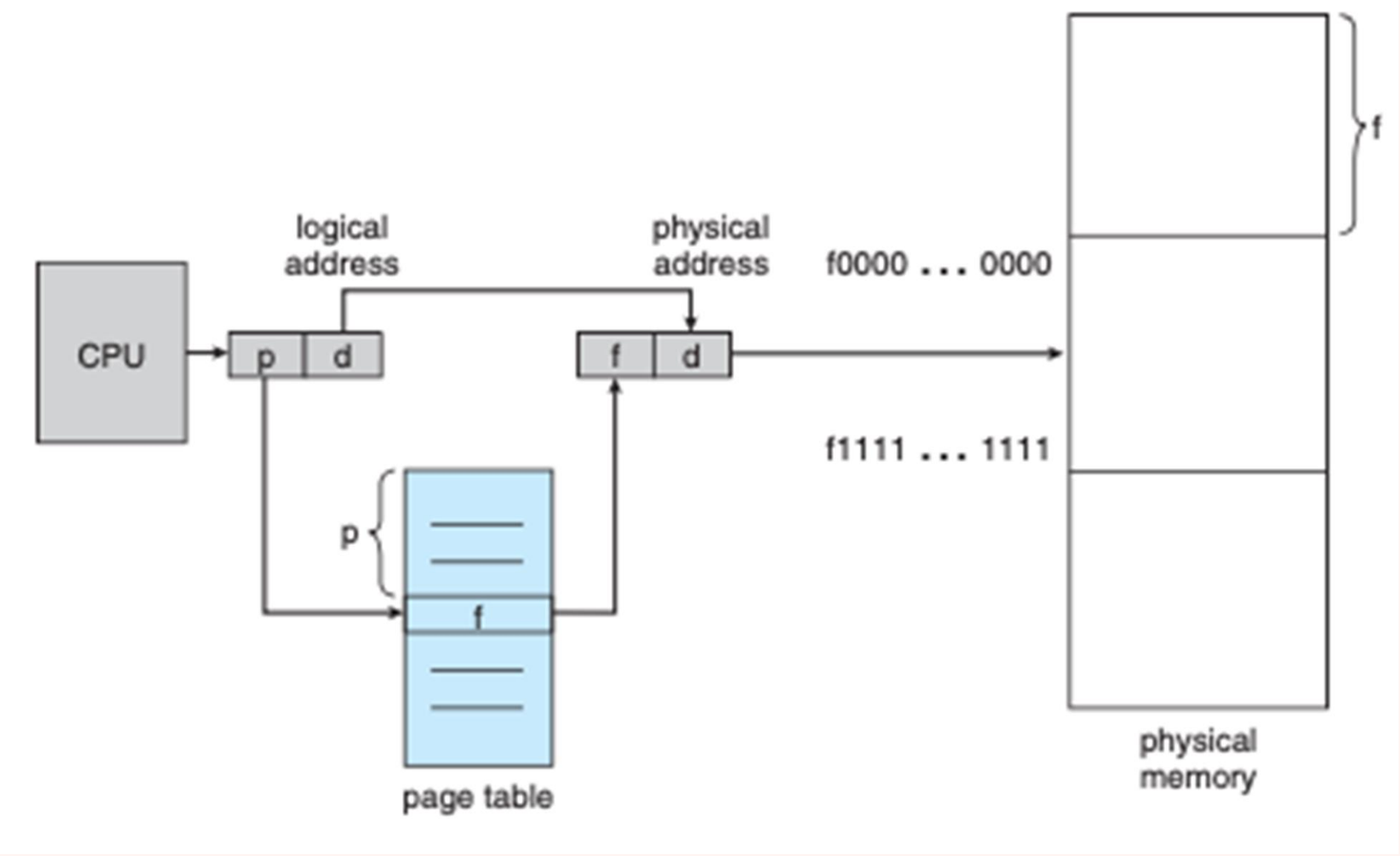
The page size (like the frame size) is defined by the hardware.

The size of a page is a power of 2, varying between 512 bytes and 1GB per page, depending on the computer architecture.

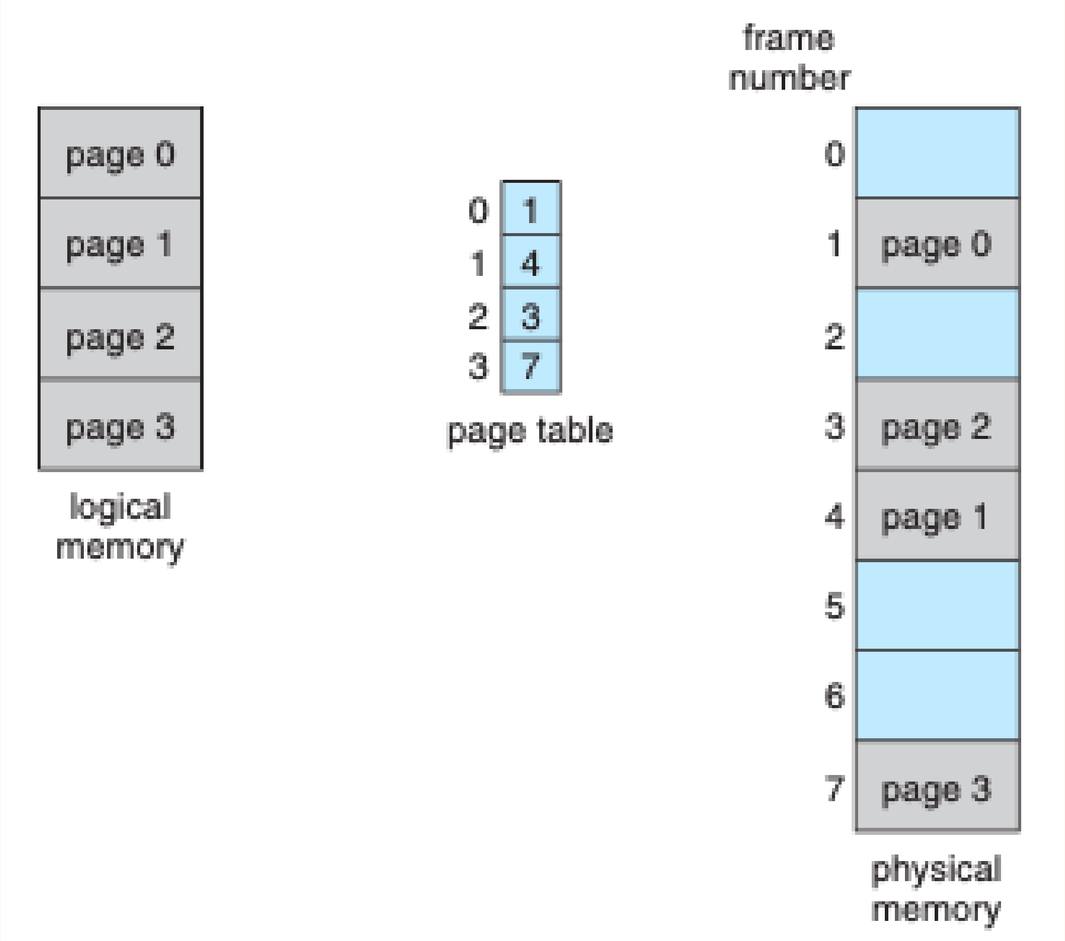
The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.

If the size of the logical address space is 2^m, and a page size is 2^n bytes, 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.

ADDRESS TRANSLATION IN PAGING



PAGE TABLE


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

PROBLEMS

Q1. Assuming a 1-KB page size, what are the page numbers and offsets for the following address reference (provided as decimal numbers):
2378
19360
34560

2. Consider a logical address space of 64 pages with 2-KB frame size mapped onto a physical memory of 128 KB.
How many bits are there in the logical and physical addresses?
How what is the breakup of offset and page number in the logical address?

3. Consider a virtual address space of eight pages with 1024 bytes each, mapped onto a physical memory of 32 frames. How many bits are used in the virtual address ? How many bits are used in the physical address ?

Q1.

Page size=1KB=1024 Bytes

(i) 2378/1024=2.3, therefore address 2378 will be in page no. 3.
And offset will be 330 (1024*2+330=2378).

(ii) 19360/1024=9.14, therefore address 9360 will be in page no. 10
and offset will be 144 (1024*9+144=9360).

2. Solution:

There are 13 bits in the virtual address.

There are 15 bits in the physical address.

FREE FRAMES

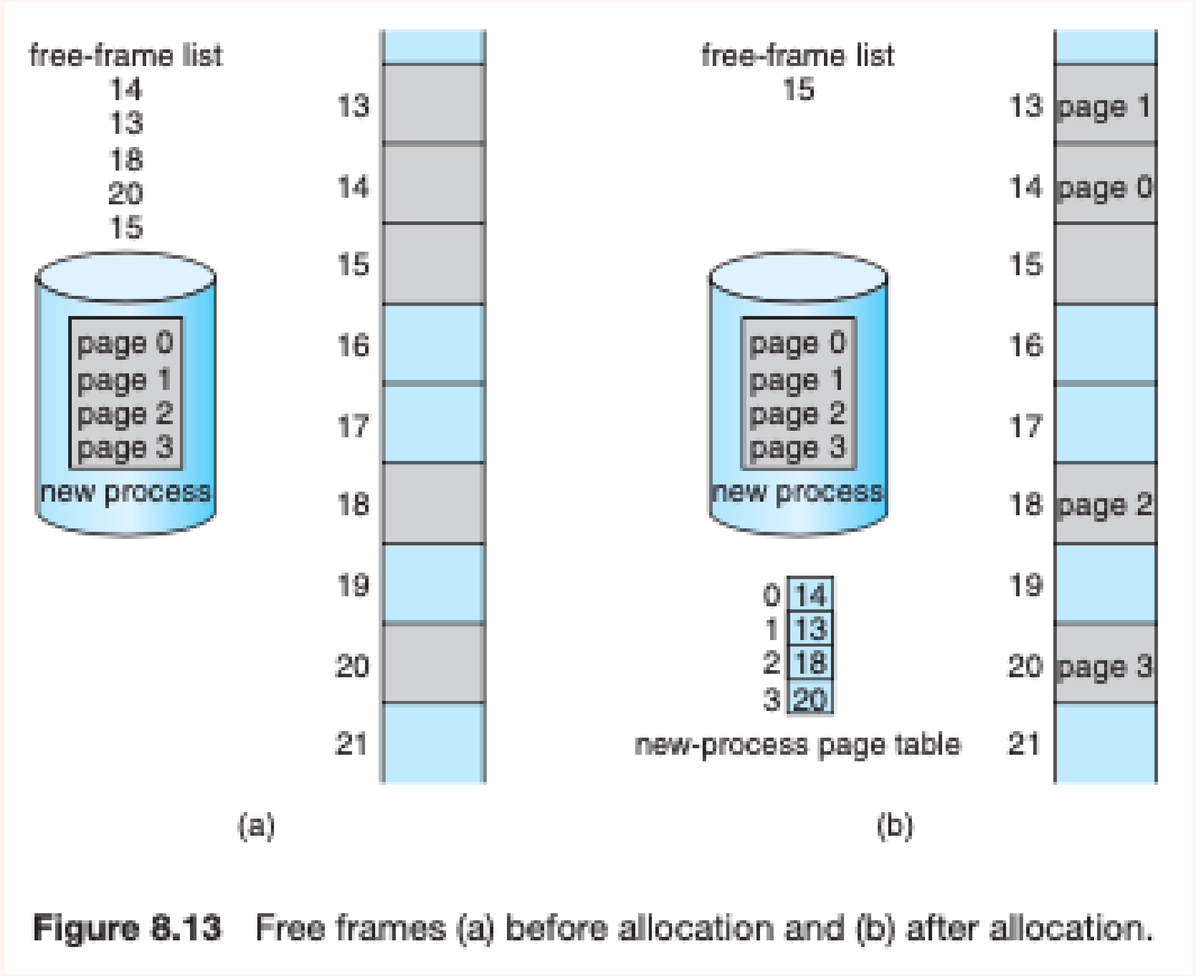
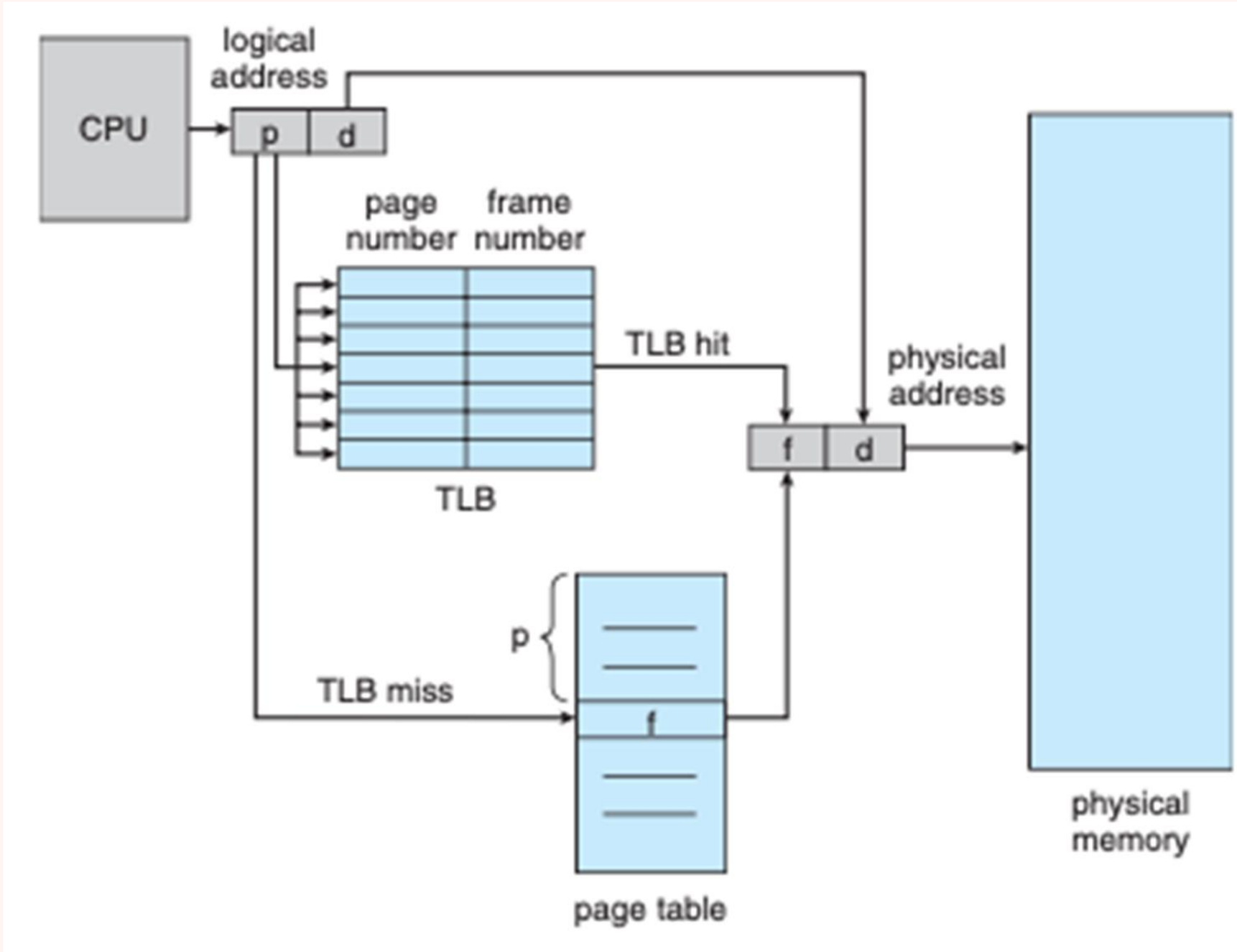


Figure 8.13 Free frames (a) before allocation and (b) after allocation.

IMPLEMENTING PAGE TABLE

- 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 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.
- *Two* memory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2.
- 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.


CPU
logical address
p
d
page number
frame number
TLB hit
physical address
f
d
TLB
p
TLB miss
f
page table
physical memory

THERE ARE TWO WAYS TO IMPLEMENT THE PAGE TABLE.

Hardware Page Table

I. It is implemented as a set of dedicated register, Which need to load and store in same way as other registers like program counter during context switching.
II. Therefore it increases the context switch time.
III. But access is so fast.
IV. It requires one memory access and one hardware access to fetch instruct..
V. It is reasonable for small Page Table.

Using Main Memory

I. For Large Page Table, it is kept in main memory.
II. Instead of copying the whole page table while context switch we use one register called PTBR(Page Table Base Register), store in PCB of process.
III. Thus it decreases the context switch time.
IV. But with this scheme two memory access are required to fetch the particular instruction.
V. Thus it provides slow access.

PAGING-TLB

WHAT REALLY HAPPENS……?

- Every address generated by the CPU is divided into two parts: Page number(P) and a Page offset(d).
- The page number is used as an Indexed into a Page Table.
- The Page Table contains the base address of each page in physical memory.
- This base address is combined with Page offset(d) to fetch the particular instruction and is given to CPU.

TRANSLATION LOOK-ASIDE BUFFER(TLB)

- It is the solution to the problem of slow access when you kept Page Table in memory.
- It is a special, small and first hardware cache.
- It stores some entries of Page Table in Key Value Pair .
- Now whenever CPU generates the logical address, it first search and match with TLB entries if it is found(TLB hit) it immediately returns the corresponding frame thus reducing one memory access to access Page Table in main memory.

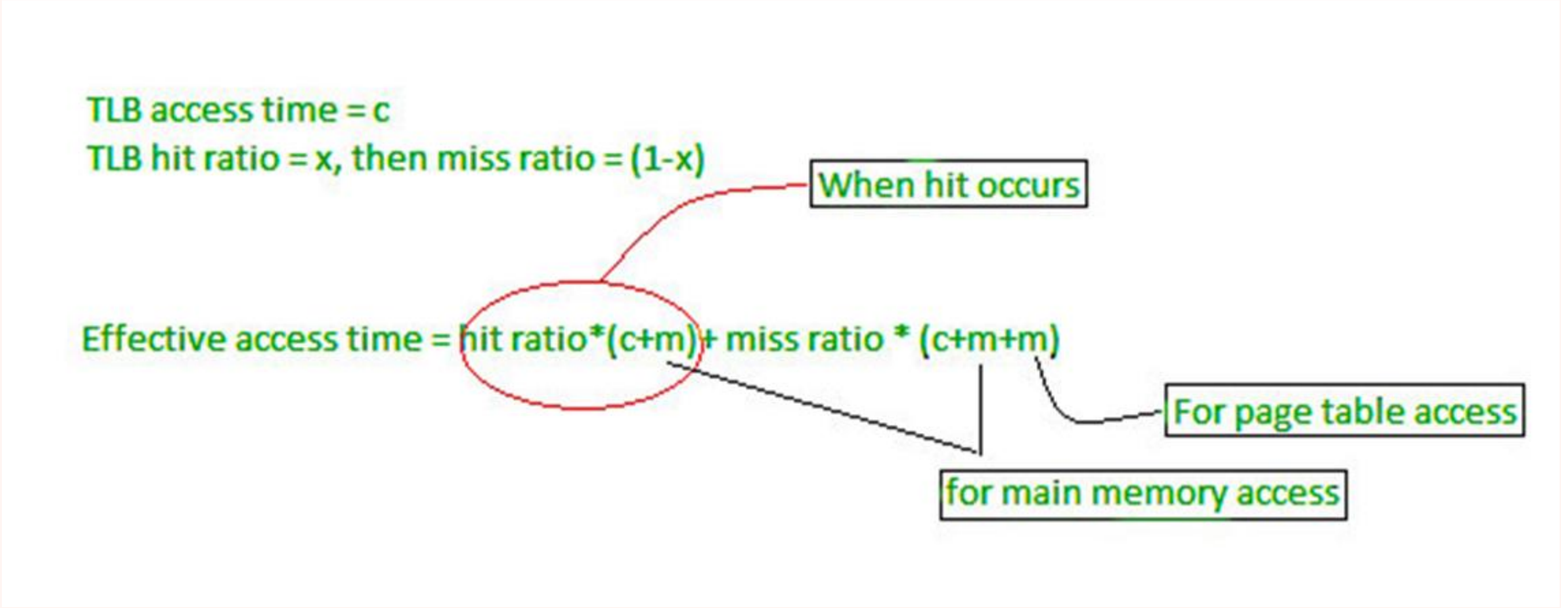
PAGING-TLB

- And if no entry matches (TLB miss), then memory access is made to look up Page Table and take corresponding entry into TLB and also use it to access frame in main memory.
- When there is context switch, TLB must be cleared or flushed before taking the next process, so that this process can use this TLB for its use. Is it good..?
- So we have to wired-down or reserved some TLB entries for OS to run interrupts that prevents unnecessary flushing of TLB.

EFFECTIVE MEMORY ACCESS TIME

- The percentage of times that the page number of interest is found in the TLB is called the hit ratio.
- If page table are kept in main memory,

Effective access time = m(for page table) + m(for particular page in page table)



Q1. Consider a paging system with the page table stored in memory.

a. **If a memory reference takes 200 nanoseconds, how long does a paged memory reference take?**
b. **b. If we add associative registers, and 75 percent of all page-table references are found in the associative registers, what is the effective memory reference time? (Assume that finding a page-table entry in the associative registers takes zero time, if the entry is there.)**

Q2. Consider a paging system with the page table stored in memory:

a. **If a paged memory reference takes 220 nanoseconds, how long does a memory reference take?**
b. **b. If we add associative registers, and we have an effective access time of 150 nanoseconds, what is the hit-ratio? Assume access to the associative registers takes 5 nanoseconds.**

SOLUTION

a. 400 nanoseconds; 200 nanoseconds to access the page table and 200 nanoseconds to access the word in memory.

b. Effective access time = 0.75 × (200 nanoseconds) + 0.25 × (400 nanoseconds) = 250 nanoseconds

Ans2.

a. 110ns

b. 150 = hit * (110+5) + (1-hit) * (110+5+110)

hit = .68, or 68%

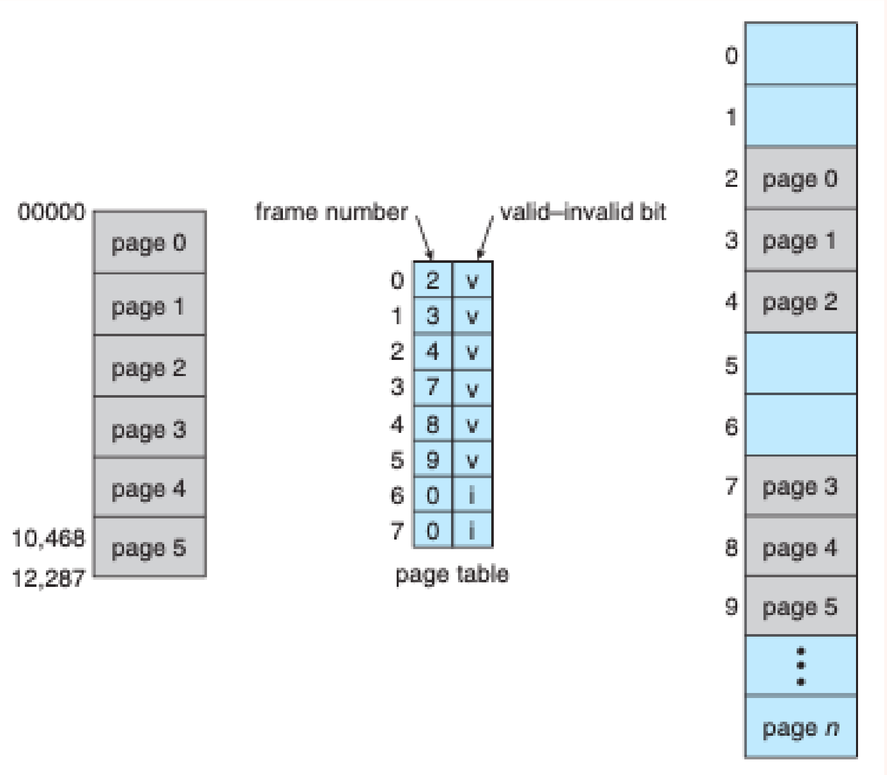
PROTECTION

- **Memory protection implemented by associating protection bit with each frame.**

- **Valid-invalid bit attached to each entry in the page table:**

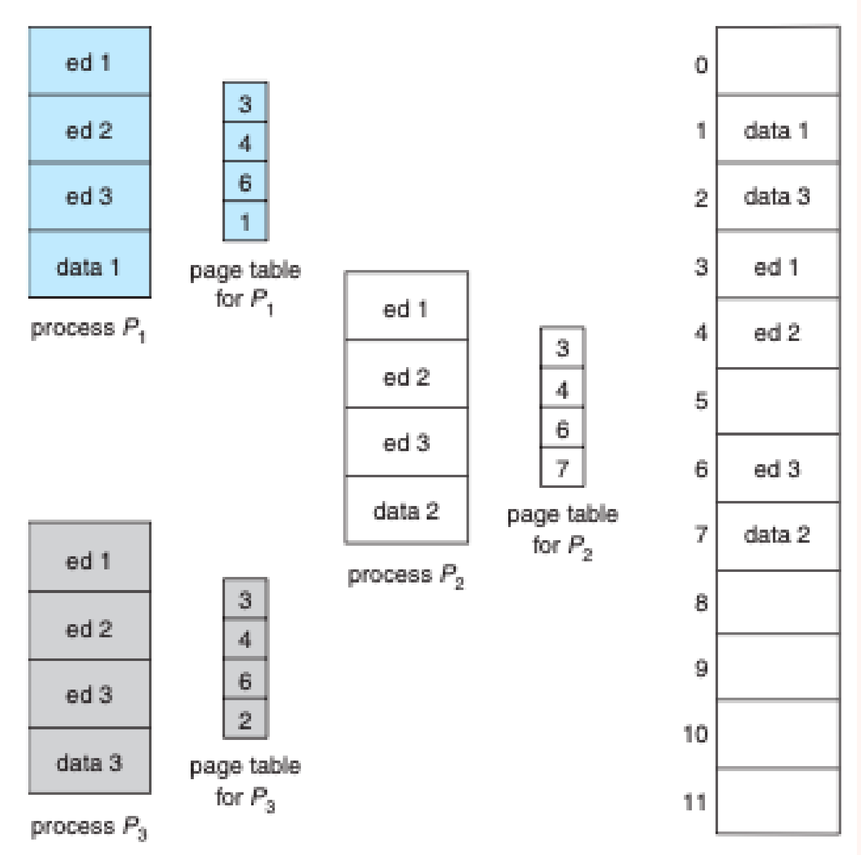
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page

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



SHARED PAGES

- An advantage of paging is the possibility of sharing common code.
- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes.
- Re-entrant 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.
- Private code and data
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space



1. Which address binding scheme generates different logical and physical addresses?

2. Name the memory management technique that supports the programmer's view of memory.

3. Assuming a 1-KB page size, what are the page numbers and offsets for the following address reference (provided as decimal numbers):
   8370
   4700
4. What do you understand by Swapping? List any two reasons why swapping is not supported on Mobile systems.
   Swapping is a MM scheme in which any process can be temporarily swapped from main memory to secondary so that main memory can be made available for other processes.
   Because mobile devices use flash memory with limited capacity and no support of secondary memory.
5. Given memory partitions of 100KB, 500KB, 200KB, 300KB and 600KB (in order), how would each of the best-fit and worst-fit algorithms place processes of 212KB, 417KB, 120KB and 426KB(in order)? Which algorithm makes the most efficient use of memory?
6. Differentiate between external and internal fragmentation by taking suitable examples.
7. Consider a logical address space of eight pages of 1024 words each, mapped onto a physical memory of 64 frames.
   How many bits are there in the logical address?
   How many bits are there in the physical address?
8. f page size=2048 bytes and process size=80,766 bytes, then find the number of pages needed for a process to be allocated using paging memory management technique.
9. Consider a paging system with the page table stored in memory. If a memory reference takes 150 ns, TLB is added and 70% of all page table reference are found in the TLBs, what is the effective memory time?(Assume that finding the page-table entry in the TLBs takes 10 ns, if the entry is there).
10. What is stub in dynamic linking.