RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Operating System**

Subject Code: **IT-501**

Semester: **5th**

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in

Downloaded from www.rgpvnotes.in

RGPVNOTES.IN

Unit 3

A process in operating systems uses different resources and uses resources in following way.

1. Requests a resource
2. Use the resource
3. Releases the resource

Deadlock:

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.

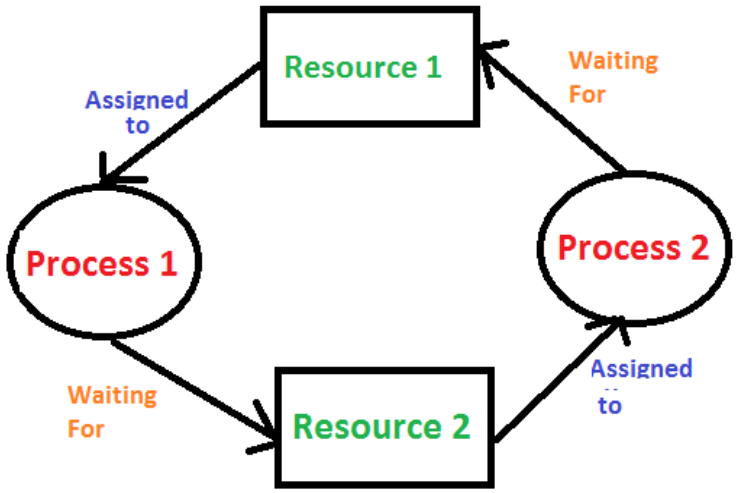


Figure 3.1 Deadlock Example

Deadlock characterization (Necessary Conditions)

Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)

- **Mutual Exclusion:** One or more than one resource is non-sharable (Only one process can use at a time)
- **Hold and Wait:** A process is holding at least one resource and waiting for resources.
- **No Pre-emption:** A resource cannot be taken from a process unless the processes release the resource
- **Circular Wait:** A set of processes are waiting for each other in circular form.

Resource Allocation Graph (RAG).

We know that any graph contains vertices and edges. So, RAG also contains vertices and edges. In RAG vertices are two types –

1. Process vertex – Every process will be represented as a process vertex. Generally, the process will be represented with a circle.

2. Resource vertex – Every resource will be represented as a resource vertex. It is also two types –

- **Single instance type resource –** It represents as a box, inside the box, there will be one dot. So, the number of dots indicates how many instances are present of each resource type.
- **Multi-resource instance type resource –** It also represents as a box, inside the box, there will be many dots present.

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

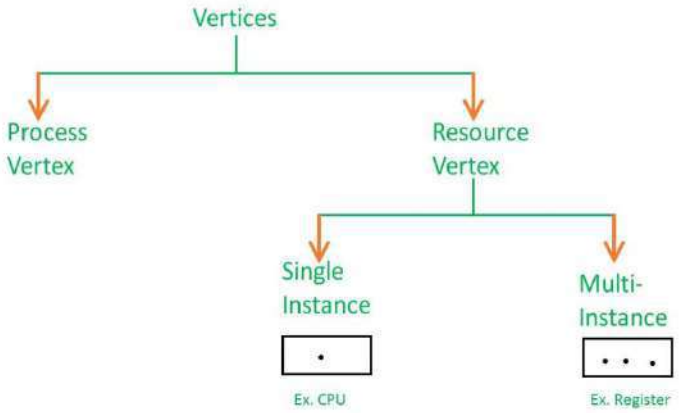


Figure 3.2 Types of Resource

Now coming to the edges of RAG, there are two types of edges in RAG –

1. **Assign Edge –** If you already assign a resource to a process then it is called Assign edge.
2. **Request Edge –** It means in future the process might want some resource to complete the execution that is called request edge.

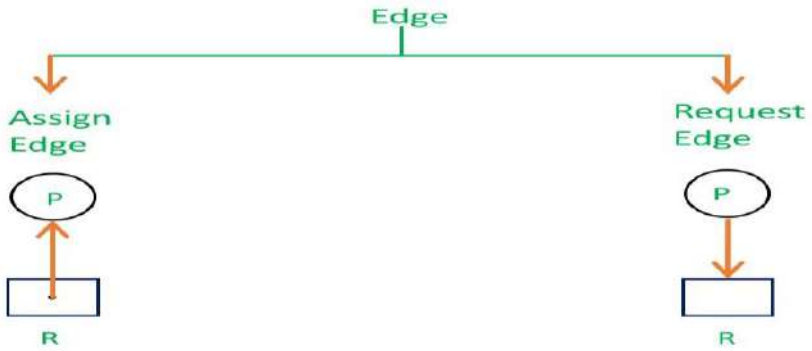
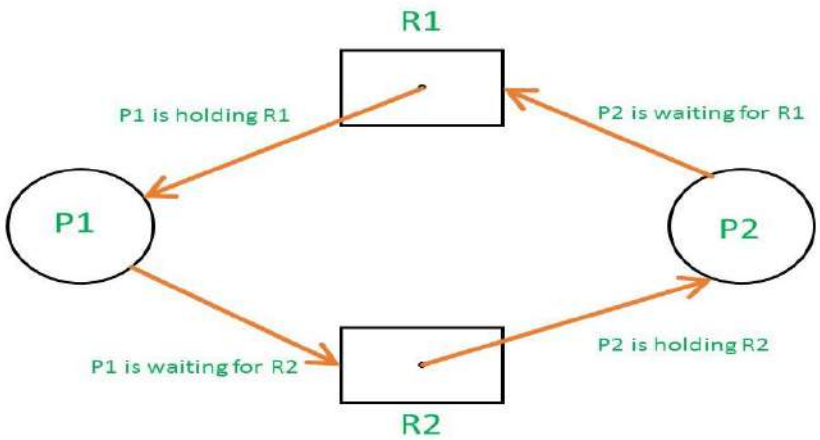


Figure 3.3 Types of Edges in RAG

So, if a process is using a resource, an arrow is drawn from the resource node to the process node. If a process is requesting a resource, an arrow is drawn from the process node to the resource node.

Example 1 (Single instances RAG) –



Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

Figure 3.4 Single instance resource types with deadlock

If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will be in deadlock. For example, if process P1 holds resource R1, process P2 holds resource R2 and process P1 is waiting for R2 and process P2 is waiting for R1, then process P1 and process P2 will be in deadlock.

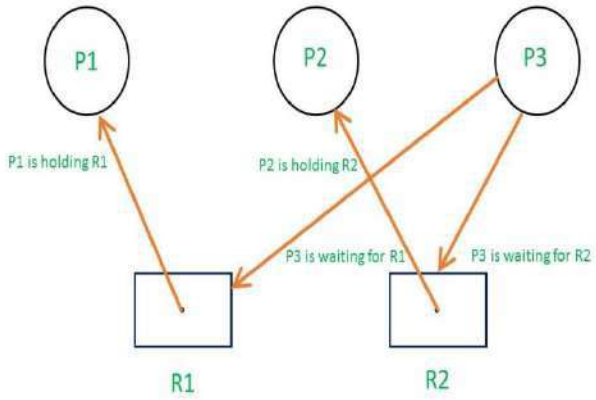


Figure 3.5 Single instance resource types without deadlock

Here's another example that shows Processes P1 and P2 acquiring resources R1 and R2 while process P3 is waiting to acquire both resources. In this example, there is no deadlock because there is no circular dependency.
So cycle in single-instance resource type is the sufficient condition for deadlock.

Example 2 (Multi-instances RAG) –

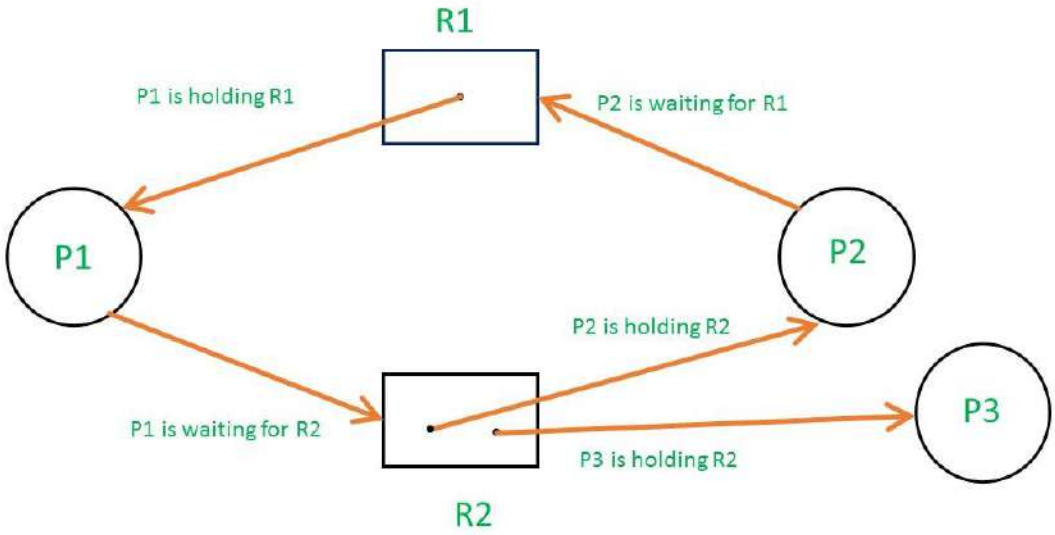


Figure 3.6 Multi instance without deadlock

Methods for handling deadlock

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

There are three ways to handle deadlock
1) Deadlock prevention or avoidance: The idea is to not let the system into deadlock state.
2) Deadlock detection and recovery: Let deadlock occur, then do pre-emption to handle it once occurred.
3) Ignore the problem all together: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

Deadlock Prevention

Deadlock prevention algorithms ensure that at least one of the necessary conditions (Mutual exclusion, hold and wait, no pre-emption and circular wait) does not hold true. However, most prevention algorithms have poor resource utilization, and hence result in reduced throughputs.

- **Mutual Exclusion**

Not always possible to prevent deadlock by preventing mutual exclusion (making all resources shareable) as certain resources are cannot be shared safely.

- **Hold and Wait:** We will see two approaches, but both have their disadvantages -

A resource can get all required resources before it starts execution. This will avoid deadlock but will result in reduced throughputs as resources are held by processes even when they are not needed. They could have been used by other processes during this time.
Second approach is to request for a resource only when it is not holing any other resource. This may result in a starvation as all required resources might not be available freely always.

- **No pre-emption**

We will see two approaches here. If a process request for a resource which is held by another waiting resource, then the resource may be pre-empted from the other waiting resource. In the second approach, if a process request for a resource which are not readily available, all other resources that it holds are pre-empted.
The challenge here is that the resources can be pre-empted only if we can save the current state can be saved and processes could be restarted later from the saved state.

- **Circular wait**

To avoid circular wait, resources may be ordered, and we can ensure that each process can request resources only in an increasing order of these numbers. The algorithm may itself increase complexity and may also lead to poor resource utilization.

Deadlock Avoidance

This requires that the system has some information available up front. Each process declares the maximum number of resources of each type which it may need. Dynamically examine the resource allocation state to ensure that there can never be a circular-wait condition.
The system's resource-allocation state is defined by the number of available and allocated resources, and the maximum possible demands of the processes. When a process requests an available resource, the system must decide if immediate allocation leaves the system in a *safe state*.
The system is in a safe state if there exists a safe sequence of all processes:
Sequence < P1, P2, Pn> is safe for the current allocation state if, for each P*i*, the resources which P*i* can still request can be satisfied by

- The currently available resources plus
- The resources held by all of the P*j*'s, where *j*<*i*.

If the system is in a safe state, there can be no deadlock. If the system is in an unsafe state, there is the *possibility* of deadlock.

Deadlock Avoidance Algorithms

Two deadlock avoidance algorithms:

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

- Resource-allocation graph algorithm
- Banker's algorithm

Resource-allocation graph algorithm

- Only applicable when we only have 1 instance of each resource type
- Claim edge (dotted edge), like a *future* request edge
- When a process requests a resource, the claim edge is converted to a request edge
- When a process releases a resource, the assignment edge is converted to a claim edge
- Cycle detection: $O(n^2)$

Banker's Algorithm

- A classic deadlock avoidance algorithm
- More general than resource-allocation graph algorithm (handles multiple instances of each resource type), but
- Is less efficient

Resource-allocations graphs for deadlock avoidance

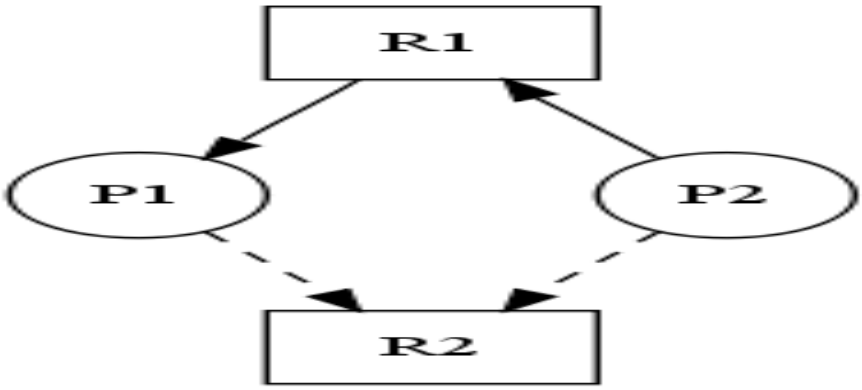


Figure 3.7 Resource-allocations graphs for deadlock avoidance

Banker's Algorithm in Detail

We call Banker's algorithm when a request for R is made. Let n be the number of processes in the system, and m be the number of resource types.

Define:

- Available[m]: the number of units of R currently unallocated (e.g., available [3] = 2)
- Max[n][m]: describes the maximum demands of each process (e.g., max [3][1] = 2)
- Allocation[n][m]: describes the current allocation status (e.g., allocation [5][1] = 3)
- Need[n][m]: describes the *remaining* possible need (i.e., need[i][j] = max[i][j] - allocation[i][j])

Resource-request algorithm:

Define:

- Request[n][m]: describes the current outstanding requests of all processes (e.g., request [2][1] = 3)

1. If request[i][j] <= need[i][j], go to step 2; otherwise, raise an error condition.
2. If request[i][j] > available[j], then the process must wait.
3. Otherwise, *pretend* to allocate the requested resources to p*i* :
   Available[j] = available[j] - request[i][j]
   Allocation[i][j] = allocation[i][j] + request[i][j]
   Need[i][j] = need[i][j] - request[i][j]
   Once the resources are allocated, check to see if the system state is safe. If unsafe, the process must

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

wait, and the old resource-allocated state is restored.

Safety algorithm (to check for a safe state):

1. Let work be an integer array of length m, initialized to available. Let finish be a Boolean array of length n, initialized to false.
2. Find an i such that both:
   - finish[i] == false
   - need[i] <= work

   If no such i exists, go to step 4
3. work = work + allocation[i]; finish [i] = true; Go to step 2

4. If finish[i] == true for all i, then the system is in a safe state, otherwise unsafe.

Run-time complexity: $O(m \times n^2)$.

Example: consider a system with 5 processes (P0 ... P4) and 3 resources types (A (10) B (5) C (7))

Resource-allocation state at time t0:

Process	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2			
P2	3	0	2	9	0	2	6	0	0			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

Table 3.1

Above given table is in safe state and sequence for execution is:

< P1, P3, P4, P2, P0 >

Now suppose, P1 requests an additional instance of A and 2 more instances of type C

Request [1] = (1, 0, 2)

1. Check if request [1] <= need[i] (yes)
2. Check if request [1] <= available[i] (yes)
3. Do pretend updates to the state

Process	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	3	0	2	3	2	2	0	2	0			
P2	3	0	2	9	0	2	6	0	0			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

Table 3.2

Above given table is in safe state and sequence for execution is:

<P1, P3, P4, P0, P2>

Hence, we immediately grant the request.

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

Deadlock Detection

- Requires an algorithm which examines the state of the system to determine whether a deadlock has occurred
- Requires overhead
  - Run-time cost of maintaining necessary information and executing the detection algorithm
  - Potential losses inherent in recovering from deadlock

Single instance of each resource type

- Wait-graph
- P*i* → p*j* = p*i*>r*q* and r*q* → p*j*
- Detect cycle: ***o*(*n*²)**
- Overhead: maintain the graph + invoke algorithm

Resource-allocations graphs for deadlock detection

Resource-allocation graph:

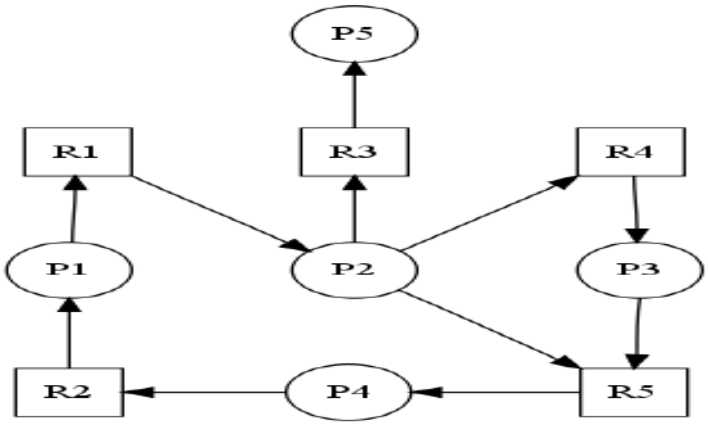


Figure 3.8 Resource-allocations graphs for deadlock detection

Corresponding wait-for graph:

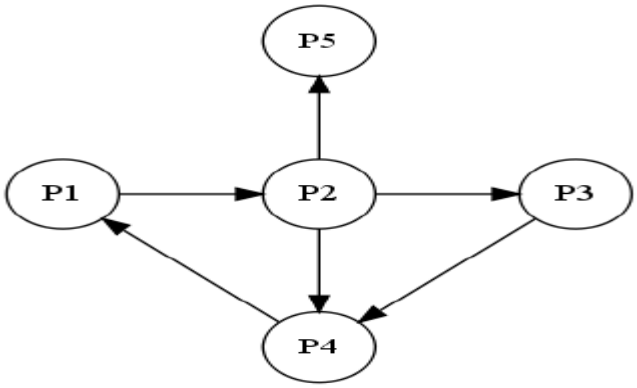


Figure 3.9 wait-for graph for deadlock detection

Multiple instances of a resource type: use an algorithm similar to Banker's, which simply investigates *every* possible allocation sequence for the processes which remain to be completed.

Define:

- Available[m]
- Allocation[n][m]
- Request[n][m]

with their usual semantics.

Algorithm:

1. Let work be an integer array of length m, initialized to available. Let finish be a Boolean array of length n. For all i,

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

If allocation[i]! = 0, then finish[i] = false;
Otherwise finish[i] = true.

2. Find an i such that both
   - Finish[i] == false // P*i* is currently *not* involved in a deadlock
   - Request[i] <= work

   If no such i exists, go to step 4
3. Reclaim the resources of process P*i*work = work + allocation[i]; finish[i] = true; Go to step 2
4. If finish [i] == false for some i, Then the system is in a deadlocked state. Moreover, if finish[i] == false, then process P*i* is deadlocked.

Run-time complexity: $O(m \times n^2)$.

Example: consider a system with 5 processes (P0 ... P4) and 3 resources types (A (7) B (2) C (6))

Resource-allocation state at time t0:

Process	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

Table 3.3

Above given table is in safe state and sequence for execution is:
< P0, P2, P3, P1, P4 >

Now suppose, P2 requests an additional instance of C:

Process	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	1			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

Table 3.4

We can reclaim the resources held by P0, the number of available resources is insufficient to fulfil the requests of the other processes.

Thus, a deadlock exists, consisting of processes P1, P2, P3, and P4.

When should we invoke the detection algorithm? Depends on:

- How *often* is a deadlock likely to occur
- How *many* processes will be affected by deadlock when it happens

If deadlocks occur frequently, then the algorithm should be invoked frequently.

Deadlocks only occur when some process makes a request which cannot be granted (if this request is completes a chain of waiting processes).

- **Extreme:** invoke the algorithm every time a request is denied

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

- **Alternative**: invoke the algorithm at less frequent time intervals:
    - Once per hour
    - Whenever CPU utilization < 40%
    - Disadvantage: cannot determine exactly which process 'caused' the deadlock

Deadlock Recovery

How to deal with deadlock:

- Inform operator and let them decide how to deal with it manually
- Let the system recover from the deadlock automatically:
    - Abort or more of the deadlocked processes to break the circular wait
    - Pre-empt some resources from one or more of the processes to break the circular wait

1. Process termination

Aborting a process is not easy; involves clean-up (e.g., file, printer).

- Abort all deadlocked processes (disadvantage: wasteful)
- Abort one process at a time until the circular wait is eliminated
    - Disadvantage: lot of overhead; must re-run algorithm after each kill
    - How to determine which process to terminate? Minimize cost
        - Priority of the process
        - How long has it executed? How much more time does it need?
        - How many and what type of resources has the process used?
        - How many more resources will the process need to complete?
        - How many processes will need to be terminated?
        - Is the process interactive or batch?

2. Resource pre-emption

Incrementally pre-empt and re-allocate resources until the circular wait is broken.

- Selecting a victim (see above)
- Rollback: what should be done with process which lost the resource? Clearly it cannot continue; must rollback to a safe state (???) => total rollback
- Starvation: pick victim only small (finite) number of times; use number of rollbacks in decision

Method for handling deadlock:

- Deadlock prevention
- Deadlock avoidance
- Deadlock detection

Introduction to Memory management:

Memory management is the functionality of an operating system, which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each memory location, regardless of it allocated to some process or it is free. It checks how much memory is to allocate to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

Process Address Space

The process address space is the set of logical addresses that a process references in its code. For example, when 32-bit addressing is in use, addresses can range from 0 to 0x7fffffff; that is, 2^31 possible numbers, for a total theoretical size of 2 gigabytes.

The operating system takes care of mapping the logical addresses to physical addresses at the time of memory

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

allocation to the program. There are three types of addresses used in a program before and after memory allocated –

S.N.	Memory Addresses & Description
1	Symbolic addresses The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.
2	Relative addresses At the time of compilation, a compiler converts symbolic addresses into relative addresses.
3	Physical addresses The loader generates these addresses at the time when a program is loaded into main memory.

Address binding

Memory consists of large array of words or arrays, each of which has address associated with it. Now the work of CPU is to fetch instructions from the memory-based program counter. Now further these instructions may cause loading or storing to specific memory address.

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. The output of this process is the appropriate physical address or the location of code/data in RAM.

An address binding can be done in three different ways:

Compile Time – It work is to generate logical address (also known as virtual address). If you know that during compile time where process will reside in memory, then absolute address is generated.
Load time – It will generate physical address. If at the compile time it is not known where process will reside then relocatable address will be generated. In this if address changes then we need to reload the user code.
Execution time- It helps in differing between physical and logical address. This is used if process can be moved from one memory to another during execution (dynamic linking-Linking that is done during load or run time).

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.
The set of all logical addresses generated by a program referred to as a logical address space the set of all-physical addresses corresponding to these logical addresses referred to as physical address space

- The value in the base register added to every address generated by a user process, which treated as offset at the time it sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will dynamically reallocated to location 10100.
- The user program deals with virtual addresses; it never sees the real physical addresses.

Logical and Physical Address space: Logical Address is address generated by CPU while a program is running. Logical address is also called virtual address as it does not exist physically. The set of all logical address space is called logical address space. Logical address is mapped to corresponding physical address using MMU (Memory Management Unit).
Compile time and load time address binding produces same logical and physical address, while run time address binding produces different.

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

Physical Address is the actual address location in memory unit. This is computed by MMU. User can access physical address in the memory unit using the corresponding logical address.

Static vs. Dynamic Loading

The choice between Static or Dynamic Loading is to make at the time of computer program developed. If you have to load your program statically, then at the time of compilation, the complete programs compiled, linked without leaving any external program or module dependency the linker combines the object program with other necessary object modules into an absolute program, which also includes logical addresses.

If you are writing a dynamically loaded program, then your compiler will compile the program and for all the modules, which you want to include dynamically, only references provided and rest of the work done at the time of execution

At the time of loading, with static loading, the absolute program (and data) is loaded into memory in order for execution to start.

If you are using dynamic loading, dynamic routines of the library are stored on a disk in re-locatable form and are loaded into memory only when they are needed by the program

Static vs. Dynamic Linking

As explained above, when static linking is to use, the linker combines all other modules needed by a program into a single executable program to avoid any runtime dependency.

When dynamic linking used, it is not required to link the actual module or library with the program, rather a reference to the dynamic module provided at the time of compilation and linking. Dynamic Link Libraries (DLL) in Windows and Shared Objects in UNIX are good examples of dynamic libraries.

MMU

before discussing memory allocation further, we must discuss the issue of memory mapping and protection. 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. 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.

The relocation-register scheme provides an effective way to allow the operating-system size to change dynamically. This flexibility is desirable in many situations. 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, we do not want to 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. Thus, using this code changes the size of the operating system during program execution.

To protect the operating system code and data by the user processes as well as protect user processes from one another using relocation register and limit register.

This is depicted in the figure below:

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

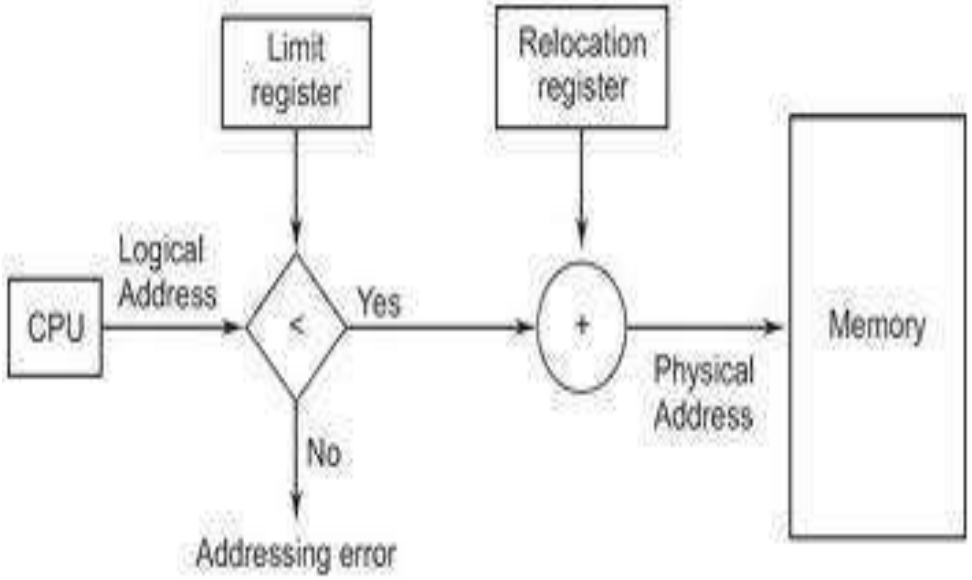


Figure 3.10 MMU

Contiguous Memory Allocation

In Contiguous Memory allocation strategies, the Operating System allocates memory to a process that is always in a sequence for faster retrieval and less overhead, but this strategy supports static memory allocation only. This strategy is inconvenient in the sense that there are more instances of internal memory fragmentation than compared to Contiguous Memory Allocation strategies. The Operating System can also allocate memory dynamically to a process if the memory is not in sequence; i.e. they are placed in non–contiguous memory segments. Memory is allotted to a process as it is required. When a process no longer needs to be in memory, it is released from the memory to produce a free region of memory or a memory hole. These memory holes and the allocated memory to the other processes remain scattered in memory. The Operating System can compact this memory at a later point in time to ensure that the allocated memory is in a sequence and the memory holes are not scattered. This strategy has support for dynamic memory allocation and facilitates the usage of Virtual Memory. In dynamic memory allocation there are no instances of internal fragmentation.

Memory management Techniques

In operating system, following are four common memory management techniques.

Single contiguous allocation: Simplest allocation method used by MS-DOS. All memory (except some reserved for OS) is available to a process.

Partitioned allocation: Memory is divided in different blocks

Paged memory management: Memory is divided in fixed sized units called page frames, used in virtual memory environment.

Segmented memory management: Memory is divided in different segments (a segment is logical grouping of process' data or code) In this management, allocated memory doesn't have to be contiguous.

Memory Allocation:

Single Partition Allocation

In this scheme Operating system is residing in low memory and user processes are executing in higher memory.

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

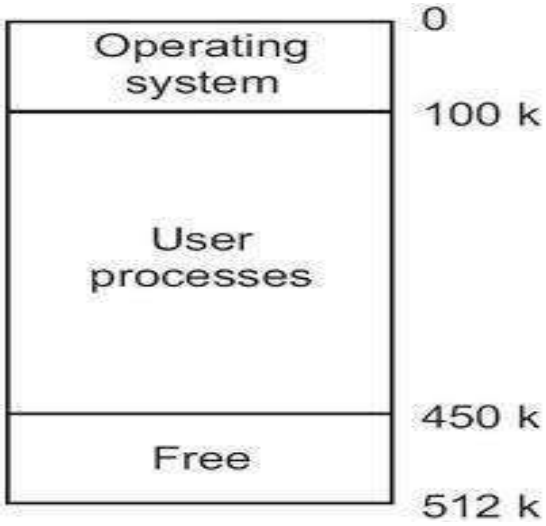


Figure 3.11 Single Partition Allocations

Advantages

- It is simple.
- It is easy to understand and use.

Disadvantages

- It leads to poor utilization of processor and memory.
- User's job is limited to the size of available memory.

Multi-partition Allocation (fixed sized partitions)

One of the simplest methods for allocating memory is to divide memory into several fixed sized partitions. There are two variations of this.

- **Fixed Equal-size Partitions**
  It divides the main memory into equal number of fixed sized partitions, operating system occupies some fixed portion and remaining portion of main memory is available for user processes.
  Advantages
  - Any process whose size is less than or equal to the partition size can be loaded into any available partition.
  - It supports multiprogramming.

  Disadvantages
  - If a program is too big to fit into a partition use overlay technique.
  - Memory use is inefficient, i.e., block of data loaded into memory may be smaller than the partition. It is known as internal fragmentation.
- **Fixed Variable Size Partitions**
  By using fixed variable size partitions, we can overcome the disadvantages present in fixed equal size partitioning. This is shown in the figure below:

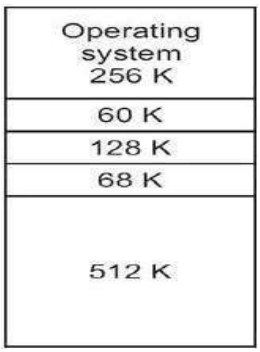


Figure 3.12 Fixed Variable Size Partitions

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

With unequal-size partitions, there are two ways to assign processes to partitions.

Use multiple queues: - For each and every process one queue is present, as shown in the figure below. Assign each process to the smallest partition within which it will fit, by using the scheduling queues, i.e., when a new process is to arrive it will put in the queue it is able to fit without wasting the memory space, irrespective of other blocks queues.

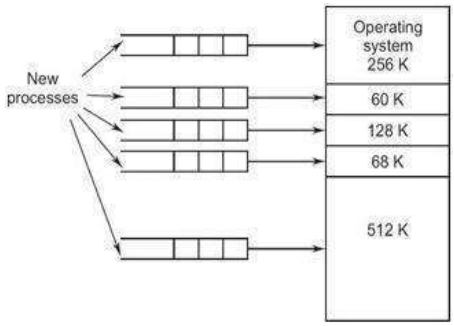


Figure 3.13 Fixed Variable Size Partitions

Advantages

- Minimize wastage of memory.

Disadvantages

- This scheme is optimum from the system point of view. Because larger partitions remains unused.

Use single queue: - In this method only one ready queue is present for scheduling the jobs for all the blocks irrespective of size. If any block is free even though it is larger than the process, it will simply join instead of waiting for the suitable block size. It is depicted in the figure below:

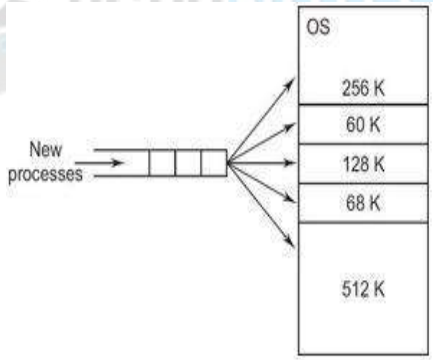


Figure 3.14 Use single queues

Advantages

- It is simple and minimum processing overhead.

Disadvantages

- The number of partitions specified at the time of system generation limits the number of active processes.
- Small jobs do not use partition space efficiently.

Dynamic Partitioning

Even though when we overcome some of the difficulties in variable sized fixed partitioning, dynamic partitioning requires more sophisticated memory management techniques. The partitions used are of variable length. That is when a process is brought into main memory, it allocates exactly as much memory as it requires. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this 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 used by IBM's mainframe operating system, OS/MVT (Multiprogramming with variable number of tasks) and it is no longer in use now.

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

Let us consider the following scenario:

Code:

Process	Size (in kB)	Arrival time (in milli sec)	Service time (in milli sec)
P1	350	0	40
P2	400	10	45
P3	300	30	35
P4	200	35	25

Figure below is showing the allocation of blocks in different stages by using dynamic partitioning method. That is the available main memory size is 1 MB. Initially the main memory is empty except the operating system shown in Figure a. Then process 1 is loaded as shown in Figure b, then process 2 is loaded as shown in Figure c without the wastage of space and the remaining space in main memory is 146K it is free. Process 1 is swaps out shown in Figure d for allocating the other higher priority process. After allocating process 3, 50K whole is created it is called internal fragmentation, shown in Figure e. Now process 2 swaps out shown in Figure f. Process 1 swaps in, into this block. But process 1 size is only 350K, this leads to create a whole of 50K shown in Figure g.

Like this, it creates a lot of small holes in memory. Ultimately memory becomes more and more fragmented and it leads to decline memory usage. This is called 'external fragmentation'. To overcome external fragmentation by using a technique called "compaction". As the part of the compaction process, from time to time, operating system shifts the processes so that they are contiguous and this free memory is together creates a block. In Figure h compaction results in a block of free memory of length 246K.

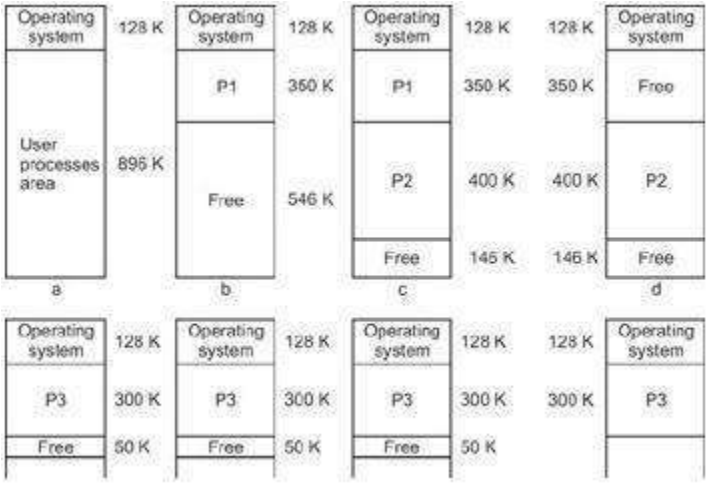


Figure 3.15 Dynamic Partitioning

Advantages

- Partitions are changed dynamically.
- It does not suffer from internal fragmentation.

Disadvantages

- It is a time-consuming process (i.e., compaction).
- Wasteful of processor time, because from time to time to move a program from one region to another in main memory without invalidating the memory references.

Placement Algorithm

If the free memory is present within a partition, then it is called "internal fragmentation". Similarly, if the free blocks are present outside the partition, then it is called "external fragmentation". Solution to the "external fragmentation" is compaction.

Solution to the "internal fragmentation" is the "placement" algorithm only.

Because memory compaction is time-consuming, when it is time to load or swap a process into main memory

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate by using three different placement algorithms.
This is shown in the figure below:

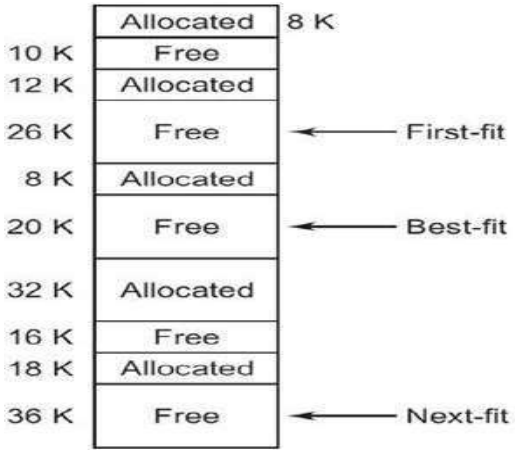


Figure 3.16 Placement Algorithm

- Best-fit: - It chooses the block that is closest in size to the given request from the beginning to the ending free blocks. We must search the entire list, unless it is ordered by size. This strategy produces the smallest leftover hole.
- Worst-fit: - It allocates the largest block. We must search the entire 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.
- First-fit: - It begins to scan memory from the beginning and chooses the first available block which is large enough. Searching can start either at the beginning of the set of blocks or where the previous first-fit search ended. We can stop searching as soon as we find a free block that is large enough.
- Last-fit: - It begins to scan memory from the location of the last placement and chooses the next available block. In the figure below the last allocated block is 18k, thus it starts from this position and the next block itself can accommodate this 20K block in place of 36K free block. It leads to the wastage of 16KB space.

First-fit algorithm is the simplest, best and fastest algorithm. Next-fit produce slightly worse results than the first-fit and compaction may be required more frequently with next-fit algorithm. Best-fit is the worst performer, even though it is to minimize the wastage space. Because it consumes the lot of processor time for searching the block which is close to its size.

Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be an allocated to memory blocks considering their small size and memory blocks remains unused this problem is known as Fragmentation.

Fragmentation is of two types –

S.N.	Fragmentation & Description
1	**External fragmentation** Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used
2	**Internal fragmentation** Memory block assigned to process is bigger. Some portion of memory left unused, as it cannot be a used by another process.

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory −

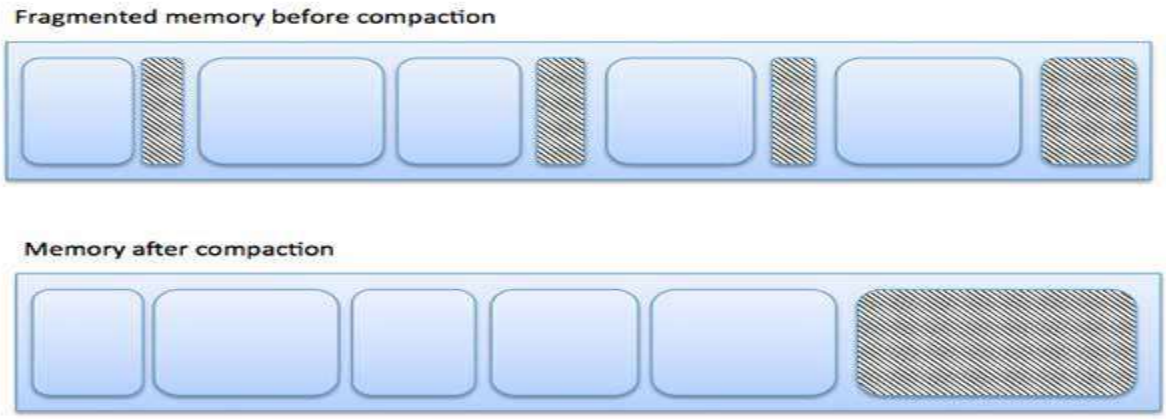


Figure 3.17 Fragmentation

External fragmentation can be a reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic
The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.

Paging

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non – contiguous.

- Logical Address or Virtual Address (represented in bits): An address generated by the CPU
- Logical Address Space or Virtual Address Space (represented in words or bytes): The set of all logical addresses generated by a program
- Physical Address (represented in bits): An address available on memory unit
- Physical Address Space (represented in words or bytes): The set of all physical addresses corresponding to the logical addresses

Example:

- If Logical Address = 31 bit, then Logical Address Space = 2^{31} words = 2 G words (1 G = 2^{30})
- If Logical Address Space = 128 M words = $2^7 * 2^{20}$ words, then Logical Address = log2 2^{27} = 27 bits
- If Physical Address = 22 bit, then Physical Address Space = 2^{22} words = 4 M words (1 M = 2^{20})
- If Physical Address Space = 16 M words = $2^4 * 2^{20}$ words, then Physical Address = log2 2^{24} = 24 bits

The mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device and this mapping is known as paging technique.

- The Physical Address Space is conceptually divided into a number of fixed-size blocks, called **frames**.
- The Logical address Space is also splitted into fixed-size blocks, called **pages**.
- Page Size = Frame Size

Let us consider an example:

- Physical Address = 12 bits, then Physical Address Space = 4 K words
- Logical Address = 13 bits, then Logical Address Space = 8 K words
- Page size = frame size = 1 K words (assumption)

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

RGPVNOTES.IN

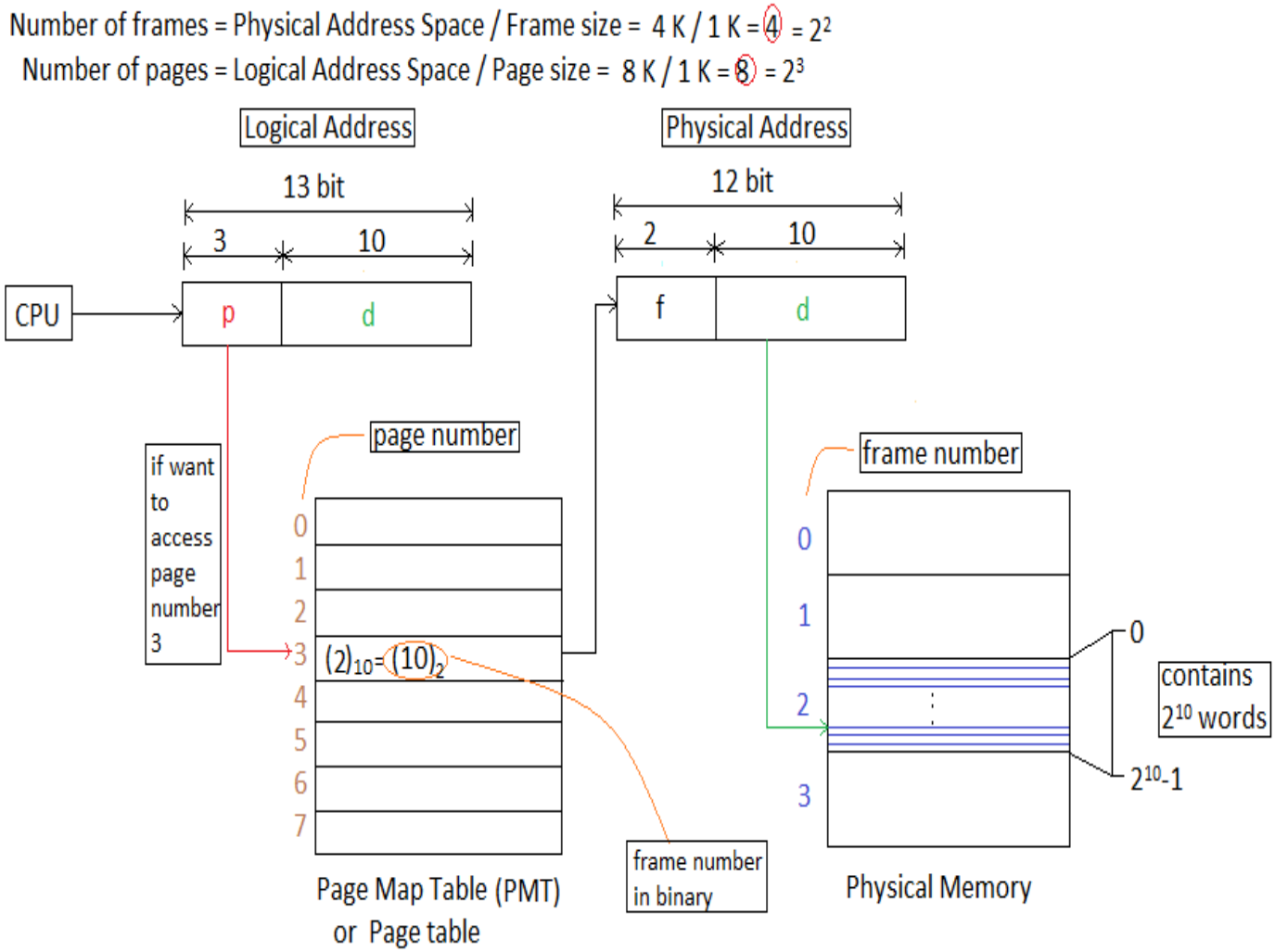


Figure 3.18 Paging

Address generated by CPU is divided into

- **Page number(p):** Number of bits required to represent the pages in Logical Address Space or Page number
- **Page offset(d):** Number of bits required to represent particular word in a page or page size of Logical Address Space or word number of a page or page offset.

Physical Address is divided into

- **Frame number (f):** Number of bits required to represent the frame of Physical Address Space or Frame number.
- **Frame offset (d):** Number of bits required to represent particular word in a frame or frame size of Physical Address Space or word number of a frame or frame offset.

Paging Issues:

- Page Table Structure. Where to store page tables?
- Issues in page table design: In a real machine, page tables stored in physical memory. Several issues arise like
  - How much memory does the page table take up?
  - How to manage the page table memory. Contiguous allocation? Blocked allocation?
  - What about paging the page table?
- On TLB misses, OS must access page tables. Issue: how the page table design affects the TLB miss penalty.

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

- Real operating systems provide the abstraction of sparse address spaces.
- Issue: how well does a particular page table design support sparse address space.

Implementation Issues of Paging

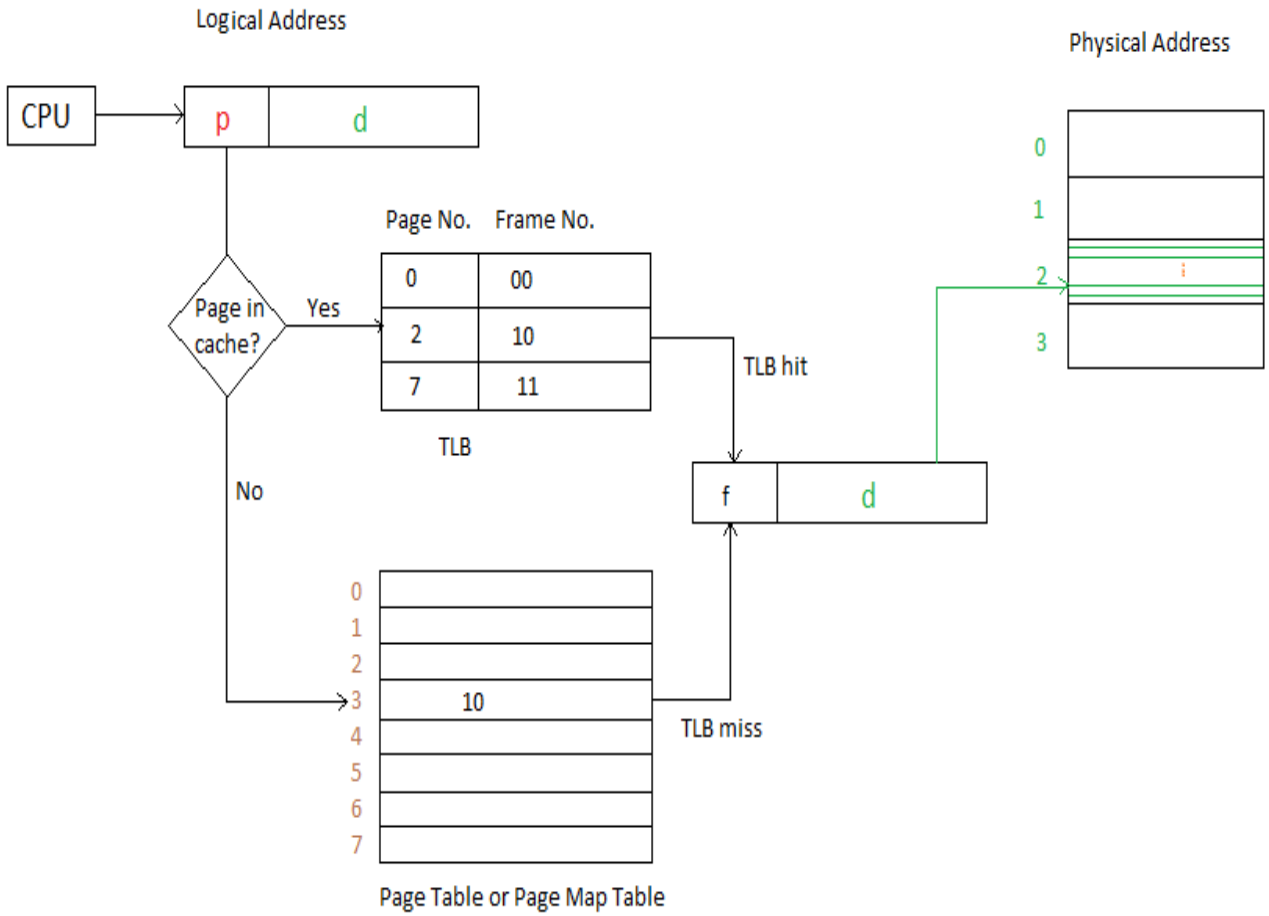
Page Size

- Basic page size is determined by hardware.
- An OS can allocate pages in (small) groups, simulating larger pages for some purposes.
- Most common size is 4K; systems have used 512 bytes to 64K.
- Advantages of larger pages.
  - Smaller page table, and fewer TLB entries needed.
  - For page I/O, less time per byte moved. (I/O operations have a fixed time plus a variable time.)
- Advantages of smaller pages.
  - Reduced internal fragmentation in the last page.
  - For page I/O, less time per page.
- When a reference brings a page into memory, less chance some of its contents is unrelated and not actually needed.

TLB (translation Look - aside buffer)

The hardware implementation of page table can be done by using dedicated registers. But the usage of register for the page table is satisfactory only if page table is small. If page table contain large number of entries then we can use TLB (translation Look - aside buffer), a special, small, fast look up hardware cache.

- The TLB is associative, high speed memory.
- Each entry in TLB consists of two parts: a tag and a value.
- When this memory is used, then an item is compared with all tags simultaneously. If the item is found, then corresponding value is returned.



Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

Figure 3.19 TLB

Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging –

- Paging reduces external fragmentation, but still suffers from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.
- Page table requires extra memory space, so may not be good for a system having small RAM.

Page Fault

A page fault occurs when a program attempts to access a block of memory that is not stored in the physical memory, or RAM. The fault notifies the operating system that it must locate the data in virtual memory, then transfer it from the storage device, such as an HDD or SSD, to the system RAM. Though the term "page fault" sounds like an error, page faults are common and are part of the normal way computers handle virtual memory. In programming terms, a page fault generates an exception, which notifies the operating system that it must retrieve the memory blocks or "pages" from virtual memory in order for the program to continue. Once the data is moved into physical memory, the program continues as normal. This process takes place in the background and usually goes unnoticed by the user.

So when page fault occurs then following sequence of events happens:

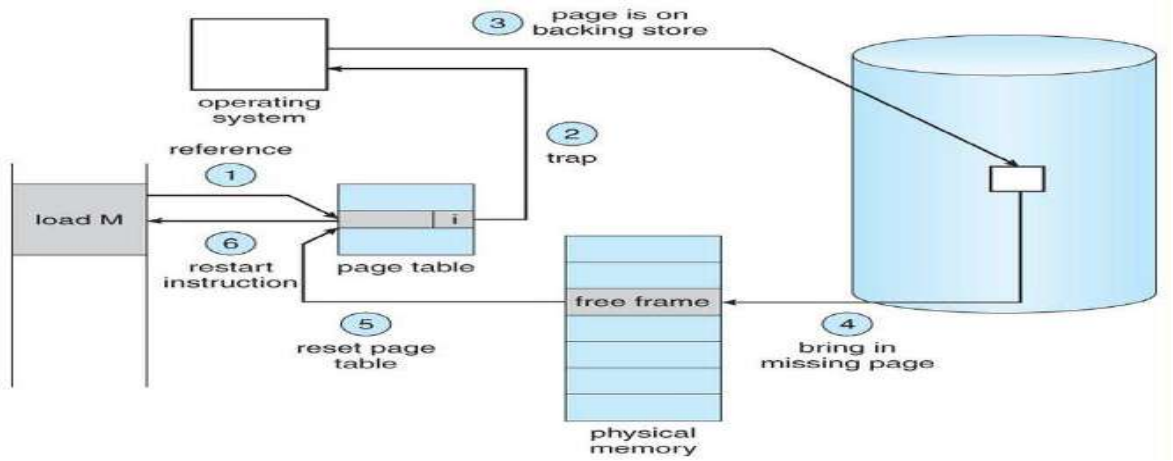


Figure 3.20 Page fault

Segmentation

- Segmentation is a memory management technique in which each job is a divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is a different logical address space of the program.
- When a process is to be an executed, its corresponding segmentation is loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.
- Segmentation memory management works very similar to paging but here segments are of variable-length whereas in paging pages are of fixed size.
- A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a segment map table for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.

Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

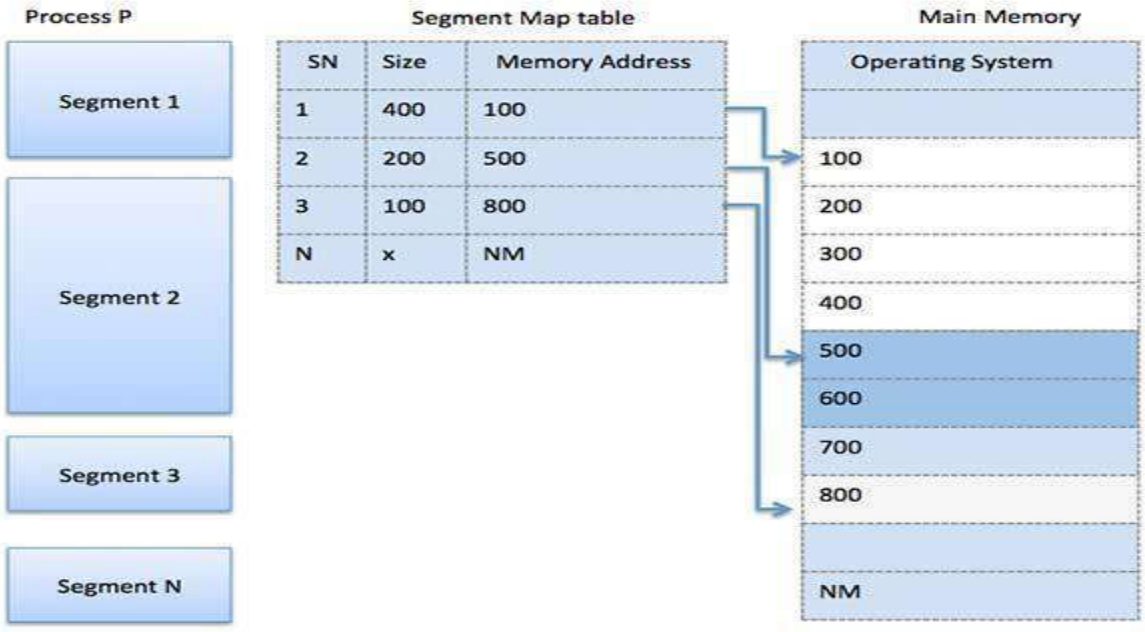


Figure 3.21 Segmentation

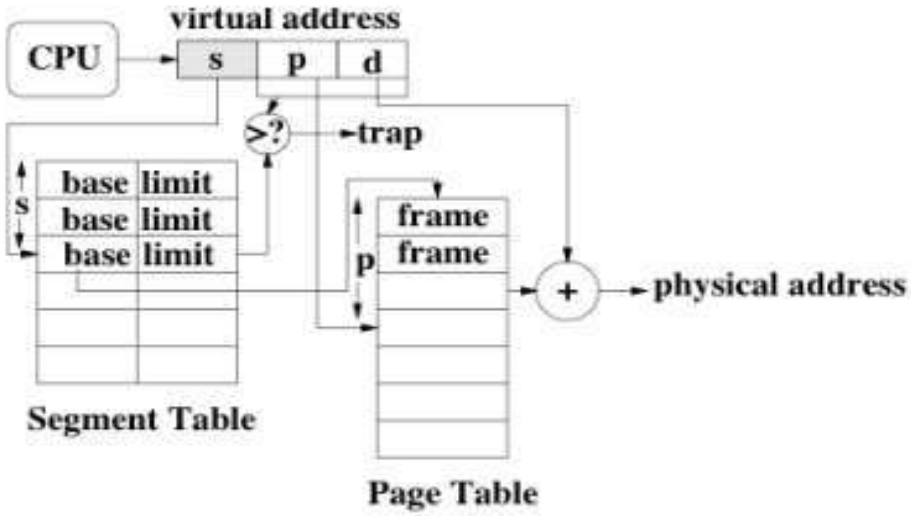
Segmentation with paging

Treat virtual address space as a collection of segments (logical units) of arbitrary sizes.

- Treat physical memory as a sequence of fixed size page frames.
- Segments are typically larger than page frames,
  - Map a logical segment onto multiple page frames by paging the segments

Addresses in a Segmented Paging System

- A virtual address becomes a segment number, a page within that segment, and an offset within the page.
- The segment number indexes into the segment table which yields the base address of the page table for that segment.
- Check the remainder of the address (page number and offset) against the limit of the segment.
- Use the page number to index the page table. The entry is the frame. (The rest of this is just like paging.)
- Add the frame and the offset to get the physical address.



Get real-time updates from RGPV

Downloaded from www.rgpvnotes.in

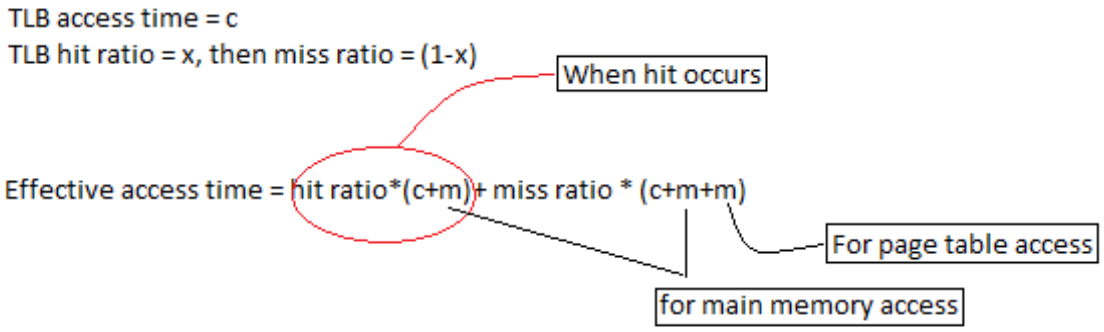
Figure 3.22 Segmented Paging

Effective access time

Main memory access time = m

If page table are kept in main memory,

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



The percentage of times that the page number of interest is found in the TLB is called the hit ratio. An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 200 nanoseconds. (We are assuming that a page-table lookup takes only one memory access, but it can take more, as we shall see.)

To find the effective memory-access time, we weight the case by its probability: effective access time = 0.80 × 100 + 0.20 × 200 = 120 nanoseconds

Get real-time updates from RGPV

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in