COURSE NAME: OPERATING SYSTEMS

COURSE CODE: 22CEE52

SEMESTER: 5

MODULE: 3 and 4

CONTENTS:

MODULE 3

DEADLOCKS

A process requests resources, if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **Deadlock.**

SYSTEM MODEL

- A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances. Memory space, CPU cycles, files, and I/0 devices are examples of resource types.
- A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires carrying out its designated task. The number of resources requested may not exceed the total number of resources available in the system.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request:** The process requests the resource. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource.
3. **Release:** The process releases the resource.

A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources or logical resources

To illustrate a deadlocked state, consider a system with three CD RW drives.
Suppose each of three processes holds one of these CD RW drives. If each process now requests another drive, the three processes will be in a deadlocked state.
Each is waiting for the event "CD RW is released," which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.

Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process Pi is holding the DVD and process P_j is holding the printer. If P_i requests the printer and P_j requests the DVD drive, a deadlock occurs.

DEADLOCK CHARACTERIZATION

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion:** At least one resource must be held in a non-sharable mode, that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. **Hold and wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. **No preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. **Circular wait:** A set $\{P_0, P_l, \ldots, P_n\}$ of waiting processes must exist such that P_o is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ... , P_{n-1} is waiting for a resource held by P_n and P_n is waiting for a resource held by P_o.

Resource-Allocation Graph

Deadlocks can be described in terms of a directed graph called **System Resource-Allocation Graph**

The graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes:

- $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the active processes in the system.
- $R = \{R_1, R_2, \ldots, R_m\}$ the set consisting of all resource types in the system.

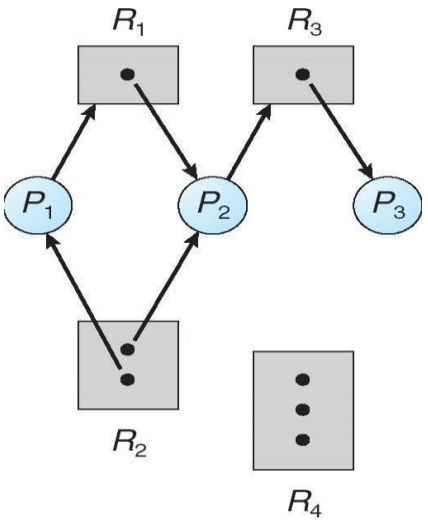
A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$ it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource.
A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$ it signifies that an instance of resource type R_j has been allocated to process P_i.

- A directed edge $P_i \rightarrow R_j$ is called a Request Edge.
- A directed edge $R_j \rightarrow P_i$ is called an Assignment Edge.

Pictorially each process P_i as a circle and each resource type R_j as a rectangle. Since resource type R_j may have more than one instance, each instance is represented as a dot within the rectangle.
A request edge points to only the rectangle R_j, whereas an assignment edge must also designate one of the dots in the rectangle.

When process P_i requests an instance of resource type R_j, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.

The resource-allocation graph shown in Figure depicts the following situation.



The sets P, K and E:

- $P = \{P_1, P_2, P_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

Resource instances:

- One instance of resource type R_1
- Two instances of resource type R_2
- One instance of resource type R_3
- Three instances of resource type R_4

Process states:

- Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
- Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.
- Process P_3 is holding an instance of R_3.

If the graph does contain a cycle, then a deadlock may exist.

- If each resource type has exactly **one instance**, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked.
- If each resource type has **several instances**, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, the resource-allocation graph depicted in below figure:
Suppose that process P3 requests an instance of resource type R2. Since no resource instance is currently available, a request edge P3 → R2 is added to the graph. At this point, two minimal cycles exist in the system:

1. P1 →R1 → P2 → R3 → P3 → R2→P1
2. P2 →R3 → P3 → R2 → P2

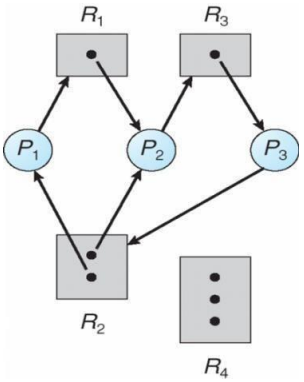


Figure: Resource-allocation graph with a deadlock.

Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

Consider the resource-allocation graph in below Figure. In this example also have a cycle:

P1→R1→P3→R2→P1

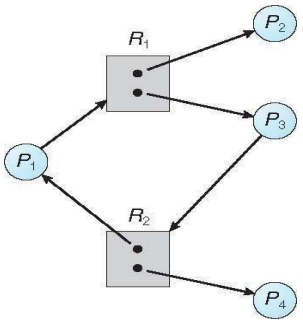


Figure: Resource-allocation graph with a cycle but no deadlock

However, there is no deadlock. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.

METHODS FOR HANDLING DEADLOCKS

The deadlock problem can be handled in one of three ways:

1. Use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
2. Allow the system to enter a deadlocked state, detect it, and recover.
3. Ignore the problem altogether and pretend that deadlocks never occur in the system.

To ensure that deadlocks never occur, the system can use either deadlock prevention or a deadlock-avoidance scheme.

Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

Deadlock-avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an **algorithm** that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock.

In the absence of algorithms to detect and recover from deadlocks, then the system is in a deadlock state yet has no way of recognizing what has happened. In this case, the undetected deadlock will result in deterioration of the system's performance, because resources are being held by processes that cannot run and because more and more processes, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

DEADLOACK PREVENTION

Deadlock can be prevented by ensuring that at least one of the four necessary conditions cannot hold.

Mutual Exclusion

- The mutual-exclusion condition must hold for non-sharable resources. Sharable resources, do not require mutually exclusive access and thus cannot be involved in a deadlock.
- Ex: Read-only files are example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource.
- Deadlocks cannot prevent by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, then guarantee that, whenever a process requests a resource, it does not hold any other resources.

- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- Another protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Ex:

- Consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.
- The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

The two main disadvantages of these protocols:

1. Resource utilization may be low, since resources may be allocated but unused for a long period.
2. Starvation is possible.

No Preemption

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated.

To ensure that this condition does not hold, the following protocols can be used:

- If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources the process is currently holding are preempted.
- The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

If a process requests some resources, first check whether they are available. If they are, allocate them.
If they are not available, check whether they are allocated to some other process that is waiting for additional resources. If so, preempt the desired resources from the waiting process and allocate them to the requesting process.
If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them.
A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

Circular Wait

One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

To illustrate, let R = {R1, R2, ... , Rm} be the set of resource types. Assign a unique integer number to each resource type, which allows to compare two resources and to determine whether one precedes another in ordering. Formally, it defined as a one-to-one function
F: R ->N, where N is the set of natural numbers.

Example: if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

F (tape drive) = 1
F (disk drive) = 5
F (printer) = 12

Now consider the following protocol to prevent deadlocks. Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type -R_i. After that, the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$.

DEADLOCK AVOIDANCE

- To avoid deadlocks an additional information is required about how resources are to be requested. With the knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock
- Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.
- The various algorithms that use this approach differ in the amount and type of information required. The simplest model requires that each process declare the *maximum number* of resources of each type that it may need. Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. Such an algorithm defines the ***deadlock-avoidance approach.***

Safe State

- **Safe state:** A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. A system is in a safe state only if there exists a safe sequence.

- **Safe sequence:** A sequence of processes <P1, P2, ... , Pn> is a safe sequence for the current allocation state if, for each Pi, the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with j <i.

In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+1 can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks as shown in figure. An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe states

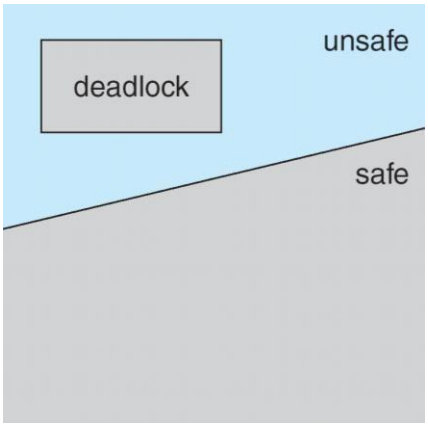


Figure: Safe, unsafe, and deadlocked state spaces.

Resource-Allocation-Graph Algorithm

- If a resource-allocation system has only one instance of each resource type, then a variant of the resource-allocation graph is used for deadlock avoidance.
- In addition to the request and assignment edges, a new type of edge is introduced, called a claim edge.
- A claim edge *Pi ->Rj* indicates that process *Pi* may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a **dashed line**.
- When process *Pi* requests resource *Rj*, the claim edge *Pi ->Rj* is converted to a request edge. When a resource *Rj* is released by *Pi* the assignment edge *Rj->Pi* is reconverted to a claim edge *Pi->Rj*.

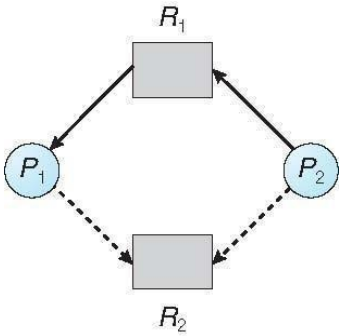


Figure: Resource-allocation graph for deadlock avoidance.

Note that the resources must be claimed a priori in the system. That is, before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph.
We can relax this condition by allowing a claim edge *Pi ->Rj* to be added to the graph only if all the edges associated with process *Pi* are claim edges.

Now suppose that process *Pi* requests resource *Rj*. The request can be granted only if converting the request edge *Pi ->Rj* to an assignment edge *Rj->Pi* does not result in the formation of a cycle in the resource-allocation graph.

There is need to check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of processes in the system.

- If no cycle exists, then the allocation of the resource will leave the system in a safe state.
- If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process Pi will have to wait for its requests to be satisfied.

To illustrate this algorithm, consider the resource-allocation graph as shown above. Suppose that P2 requests R2. Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph.
A cycle, indicates that the system is in an unsafe state. If P1 requests R2, and P2 requests R1, then a deadlock will occur.

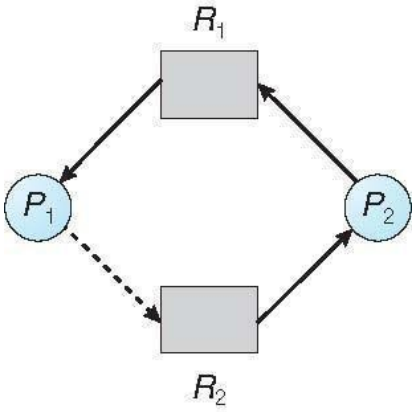


Figure: An unsafe state in a resource-allocation graph

Banker's Algorithm

The Banker's algorithm is applicable to a resource allocation system with multiple instances of each resource type.

- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

To implement the banker's algorithm the following data structures are used.

Let n = number of processes, and m = number of resources types

Available: A vector of length *m* indicates the number of available resources of each type. If available [j] = k, there are k instances of resource type Rj available.

Max: An *n x m* matrix defines the maximum demand of each process. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj

Allocation: An *n x m* matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj

Need: An n x m matrix indicates the remaining resource need of each process. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task.

$$Need [i,j] = Max[i,j] – Allocation [i,j]$$

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize:
   Work = Available
   Finish [i] = false for i = 0, 1,…,n- 1

2. Find an index i such that both:
   (a) Finish[i] = false
   (b) $Need_i \leq$ 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

This algorithm may require an order of m x n^2 operations to determine whether a state is safe.

Resource-Request Algorithm

The algorithm for determining whether requests can be safely granted.
Let $Request_i$ be the request vector for process P_i. If $Request_i[j] == k$, then process P_i wants k instances of resource type R_j. When a request for resources is made by process Pi, the following actions are taken:

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available

3. Have the system pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

If safe ⇒ the resources are allocated to Pi
If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored

Example

Consider a system with five processes P_o through P_4 and three resource types *A*, *B*, and C. Resource type *A* has ten instances, resource type *B* has five instances, and resource type C has seven instances. Suppose that, at time T_0 the following snapshot of the system has been taken:

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of the matrix *Need* is defined to be *Max* - *Allocation*

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

The system is currently in a safe state. Indeed, the sequence <P_1, P_3, P_4, P_2, P_0> satisfies the safety criteria.

Suppose now that process P_1 requests one additional instance of resource type A and two instances of resource type C, so $Request_1$ = (1,0,2). Decide whether this request can be immediately granted.

Check that Request ≤ Available

(1,0,2) ≤ (3,3,2) ⇒ true

Then pretend that this request has been fulfilled, and the following new state is arrived.

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Executing safety algorithm shows that sequence <P_1, P_3, P_4, P_0, P_2> satisfies safety requirement.

DEADLOCK DETECTION

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

Single Instance of Each Resource Type

- If all resources have only a single instance, then define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a ***wait-for*** graph**.**
- This graph is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- An edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_i$ for some resource R_q.

Example: In below Figure, a resource-allocation graph and the corresponding wait-for graph is presented.

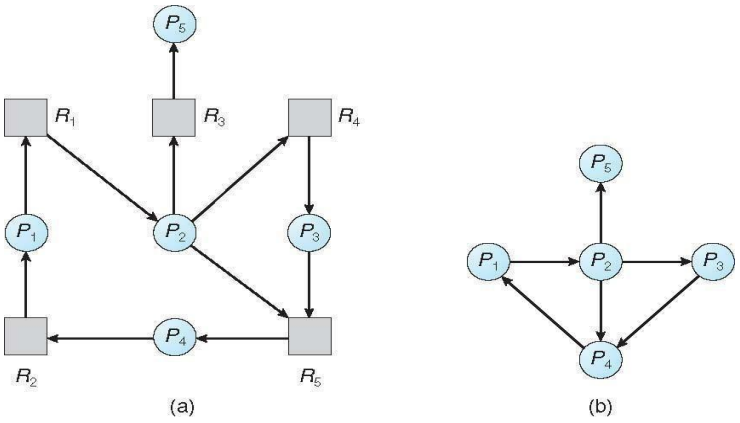


Figure: (a) Resource-allocation graph. (b) Corresponding wait-for graph.

- A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to *maintain* the wait-for graph and periodically *invoke an algorithm* that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Several Instances of a Resource Type

A deadlock detection algorithm that is applicable to several instances of a resource type. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm.

- **Available**: A vector of length *m* indicates the number of available resources of each type.
- **Allocation:** An*n x m* matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An *n* x *m* matrix indicates the current request of each process. If *Request[i][j]* equals *k,* then process *P;* is requesting *k* more instances of resource type *Rj.*

Algorithm:

1. Let *Work* and *Finish* be vectors of length *m* and *n,* respectively Initialize:
   (a) Work = Available
   (b) For i = 1,2, …, *n,* if $Allocation_i \neq 0$, then *Finish*[i] = false;
   otherwise, *Finish*[i] = *true*

2. Find an index *i*such that both:
   (a) Finish[*i*] == *false*
   (b)$Request_i \leq Work$

   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*] == false, for some *i,* $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false,* then P_i is deadlocked

Algorithm requires an order of O(m x $n^{2)}$ operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

Consider a system with five processes *Po* through *P4* and three resource types *A, B,* and C. Resource type *A* has seven instances, resource type *B* has two instances, and resource type C has six instances. Suppose that, at time T_0, the following resource-allocation state:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

After executing the algorithm, Sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in *Finish*[*i*] = true for all *i*

Suppose now that process P2 makes one additional request for an instance of type C. The Request matrix is modified as follows:

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

The system is now deadlocked. Although we can reclaim the resources held by process Po, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P1, P2, P3, and P4.

Detection-Algorithm Usage

The detection algorithm can be invoked on two factors:

1. How *often* is a deadlock likely to occur?
2. How *many* processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken.

If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

RECOVERY FROM DEADLOCK

The system recovers from the deadlock automatically. There are two options for breaking a deadlock one is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

Process Termination

To eliminate deadlocks by aborting a process, use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

1. **Abort all deadlocked processes:** This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
2. **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. Many factors may affect which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what types of resources the process has used.
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated?
6. Whether the process is interactive or batch

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim.** Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution.
2. **Rollback.** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it.

3. **Starvation.** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

MEMORY MANAGEMENT

Main Memory Management Strategies

- Every program to be executed has to be executed must be in memory. The instruction must be fetched from memory before it is executed.
- In multi-tasking OS memory management is complex, because as processes are swapped in and out of the CPU, their code and data must be swapped in and out of memory.

Basic Hardware

- Main memory, cache and CPU registers in the processors are the only storage spaces that CPU can access directly.
- The program and data must be bought into the memory from the disk, for the process to run. Each process has a separate memory space and must access only this range of legal addresses. Protection of memory is required to ensure correct operation. This prevention is provided by hardware implementation.
- Two registers are used - a base register and a limit register. The base register holds the smallest legal physical memory address; the limit register specifies the size of the range.
- For example, The base register holds the smallest legal physical memory address; the limit register specifies the size of the range. For example, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 (inclusive).

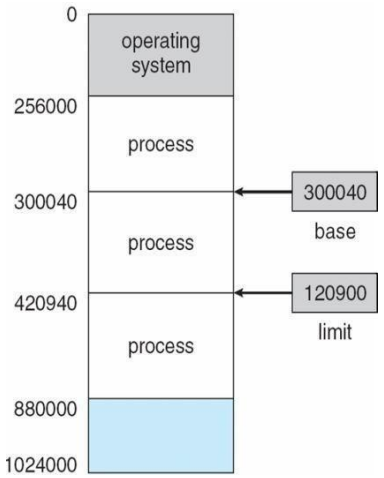


Figure: A base and a limit-register define a logical-address space

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

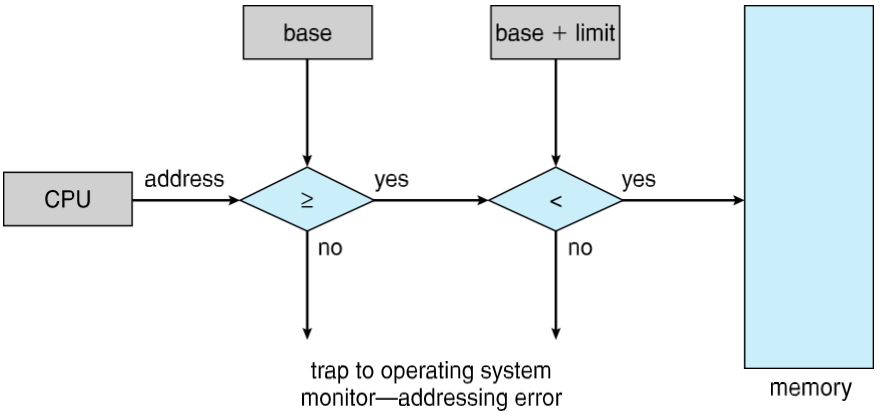


Figure: Hardware address protection with base and limit-registers

Address Binding

- User programs typically refer to memory addresses with symbolic names. These symbolic names must be mapped or bound to physical memory addresses.
- Address binding of instructions to memory-addresses can happen at 3 different stages.

1. **Compile Time** - If it is known at compile time where a program will reside in physical memory, then absolute code can be generated by the compiler, containing actual physical addresses. However, if the load address changes at some later time, then the program will have to be recompiled.
2. **Load Time** - If the location at which a program will be loaded is not known at compile time, then the compiler must generate relocatable code, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.
3. **Execution Time** - If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time.

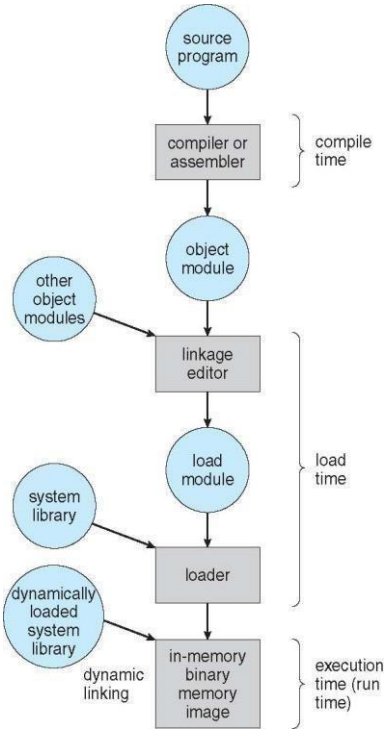


Figure: Multistep processing of a user program

Logical Versus Physical Address Space

- The address generated by the CPU is a logical address, whereas the memory address where programs are actually stored is a physical address.
- The set of all logical addresses used by a program composes the logical address space, and the set of all corresponding physical addresses composes the physical address space.
- The run time mapping of logical to physical addresses is handled by the memory-management unit (MMU).
    - One of the simplest is a modification of the base-register scheme.
    - The base register is termed a relocation register
    - The value in the relocation-register is added to every address generated by a user-process at the time it is sent to memory.
    - The user-program deals with logical-addresses; it never sees the real physical-addresses.

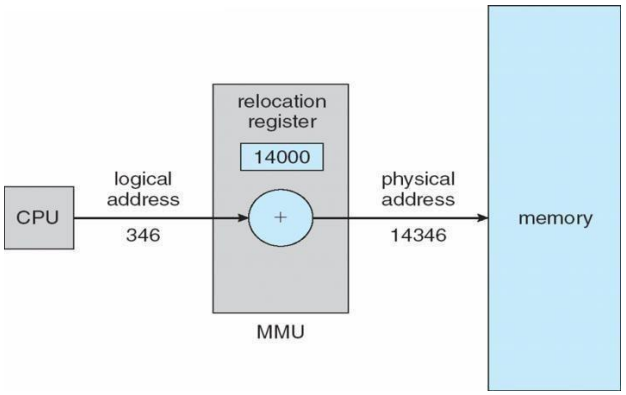


Figure: Dynamic relocation using a relocation-register

Dynamic Loading

- This can be used to obtain better memory-space utilization.
- A routine is not loaded until it is called.

This works as follows:

1. Initially, all routines are kept on disk in a relocatable-load format.
2. Firstly, the main-program is loaded into memory and is executed.
3. When a main-program calls the routine, the main-program first checks to see whether the routine has been loaded.
4. If routine has been not yet loaded, the loader is called to load desired routine into memory.
5. Finally, control is passed to the newly loaded-routine.

Advantages:

1. An unused routine is never loaded.
2. Useful when large amounts of code are needed to handle infrequently occurring cases.
3. Although the total program-size may be large, the portion that is used (and hence loaded) may be much smaller.
4. Does not require special support from the OS.

Dynamic Linking and Shared Libraries

- With **static linking** library modules get fully included in executable modules, wasting both disk space and main memory usage, because every program that included a certain routine from the library would have to have their own copy of that routine linked into their executable code.
- With **dynamic linking**, however, only a stub is linked into the executable module, containing references to the actual library module linked in at run time.
    - The stub is a small piece of code used to locate the appropriate memory-resident library-routine.
    - This method saves disk space, because the library routines do not need to be fully included in the executable modules, only the stubs.
    - An added benefit of dynamically linked libraries (DLLs, also known as shared libraries or shared objects on UNIX systems) involves easy upgrades and updates.

Shared libraries

- A library may be replaced by a new version, and all programs that reference the library will automatically use the new one.
- Version info. is included in both program & library so that programs won't accidentally execute incompatible versions.

Swapping

- A process must be loaded into memory in order to execute.
- If there is not enough memory available to keep all running processes in memory at the same time, then some processes that are not currently using the CPU may have their memory swapped out to a fast local disk called the ***backing store***.
- *Swapping is the process of moving a process from memory to backing store and moving another process from backing store to memory*. Swapping is a very slow process compared to other operations.
- A variant of swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is ***called roll out, roll in.***

Swapping depends upon address-binding:

- If binding is done at load-time, then process cannot be easily moved to a different location.
- If binding is done at execution-time, then a process can be swapped into a different memory-space, because the physical-addresses are computed during execution-time.

Major part of swap-time is transfer-time; i.e. total transfer-time is directly proportional to the amount of memory swapped.

Disadvantages:

1. Context-switch time is fairly high.
2. If we want to swap a process, we must be sure that it is completely idle.
   Two solutions:
   i) Never swap a process with pending I/O.
   ii) Execute I/O operations only into OS buffers.

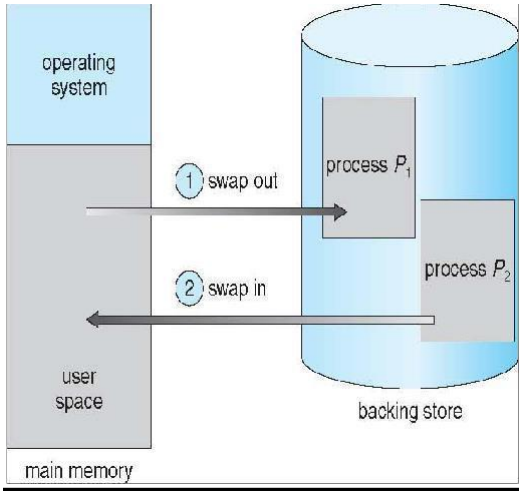


Figure: Swapping of two processes using a disk as a backing store

Example:

Assume that the user process is 10 MB in size and the backing store is a standard hard disk with a transfer rate of 40 MB per second.

The actual transfer of the 10-MB process to or from main memory takes

10000 KB/40000 KB per second = 1/4 second

= 250 milliseconds.

Assuming that no head seeks are necessary, and assuming an average latency of 8 milliseconds, the swap time is 258 milliseconds. Since we must both swap out and swap in, the total swap time is about 516 milliseconds.

Contiguous Memory Allocation

- The main memory must accommodate both the operating system and the various user processes. Therefore we need to allocate the parts of the main memory in the most efficient way possible.
- Memory is usually divided into 2 partitions: One for the resident OS. One for the user processes.
- Each process is contained in a single contiguous section of memory.

1. Memory Mapping and Protection

- Memory-protection means protecting OS from user-process and protecting user-processes from one another.
- Memory-protection is done using
  - Relocation-register: contains the value of the smallest physical-address.
  - Limit-register: contains the range of logical-addresses.
- Each logical-address must be less than the limit-register.
- The MMU maps the logical-address dynamically by adding the value in the relocation-register. This mapped-address is sent to memory
- When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit-registers with the correct values.
- Because every address generated by the CPU is checked against these registers, we can protect the OS from the running-process.
- The relocation-register scheme provides an effective way to allow the OS size to change dynamically.
- Transient OS code: Code that comes & goes as needed to save memory-space and overhead for unnecessary swapping.

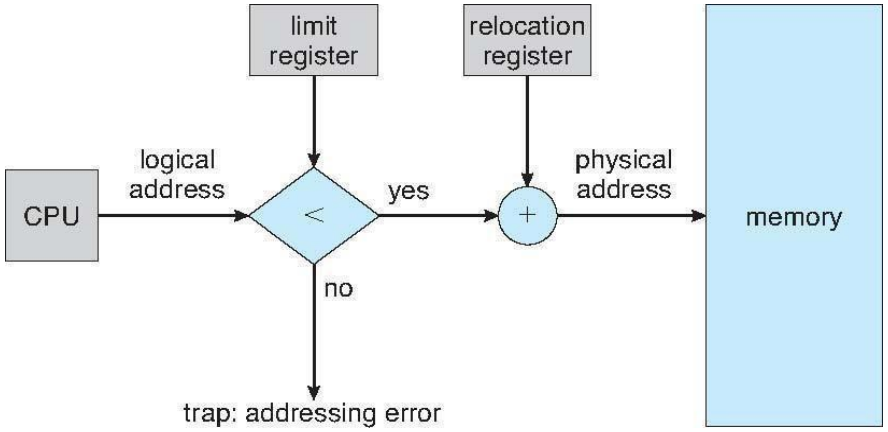


Figure: Hardware support for relocation and limit-registers

2. Memory Allocation

Two types of memory partitioning are:

1. Fixed-sized partitioning
2. Variable-sized partitioning

1. Fixed-sized Partitioning

- The memory is divided into fixed-sized partitions.
- Each partition may contain exactly one process.
- The degree of multiprogramming is bound by the number of partitions.
- When a partition is free, a process is selected from the input queue and loaded into the free partition.
- When the process terminates, the partition becomes available for another process.

2. Variable-sized Partitioning

- The OS keeps a table indicating which parts of memory are available and which parts are occupied.
- A hole is a block of available memory. Normally, memory contains a set of holes of various sizes.
- Initially, all memory is available for user-processes and considered one large hole.
- When a process arrives, the process is allocated memory from a large hole.
- If we find the hole, we allocate only as much memory as is needed and keep the remaining memory available to satisfy future requests.

Three strategies used to select a free hole from the set of available holes:

1. 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.

2. 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.

3. 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.

First-fit and best fit are better than worst fit in terms of decreasing time and storage utilization.

3. Fragmentation

Two types of memory fragmentation:
1. Internal fragmentation
2. External fragmentation

1. Internal Fragmentation

- The general approach is to break the physical-memory into fixed-sized blocks and allocate memory in units based on block size.
- The allocated-memory to a process may be slightly larger than the requested-memory.
- The difference between requested-memory and allocated-memory is called internal fragmentation i.e. Unused memory that is internal to a partition.

2. External Fragmentation

- External fragmentation occurs when there is enough total memory-space to satisfy a request but the available-spaces are not contiguous. (i.e. storage is fragmented into a large number of small holes).
- Both the first-fit and best-fit strategies for memory-allocation suffer from external fragmentation.
- Statistical analysis of first-fit reveals that given N allocated blocks, another 0.5 N blocks will be lost to fragmentation. This property is known as the 50-percent rule.

Two solutions to external fragmentation:

- Compaction: The goal is to shuffle the memory-contents to place all free memory together in one large hole. Compaction is possible only if relocation is dynamic and done at execution-time
- Permit the logical-address space of the processes to be non-contiguous. This allows a process to be allocated physical-memory wherever such memory is available. Two techniques achieve this solution: 1) Paging and 2) Segmentation.

Paging

- Paging is a memory-management scheme.
- This permits the physical-address space of a process to be non-contiguous.
- This also solves the considerable problem of fitting memory-chunks of varying sizes onto the backing-store.
- Traditionally: Support for paging has been handled by hardware.
- Recent designs: The hardware & OS are closely integrated.

Basic Method of Paging

- The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages.
- When a process is to be executed, its pages are loaded into any available memory frames from the backing store.
- The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

The hardware support for paging is illustrated in Figure 1.

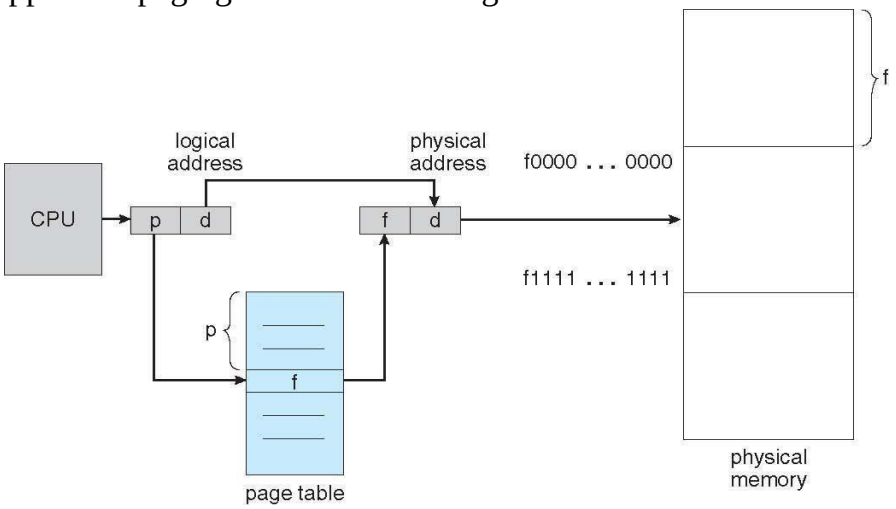


Figure 1: Paging hardware

- Address generated by CPU is divided into 2 parts (Figure 2):
  1. Page-number (p) is used as an index to the page-table. The page-table contains the base-address of each page in physical-memory.
  2. Offset (d) is combined with the base-address to define the physical-address. This physical-address is sent to the memory-unit.
- The page table maps the page number to a frame number, to yield a physical address
- The page table maps the page number to a frame number, to yield a physical address which also has two parts: The frame number and the offset within that frame.
- The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame.

The paging model of memory is shown in Figure 2.

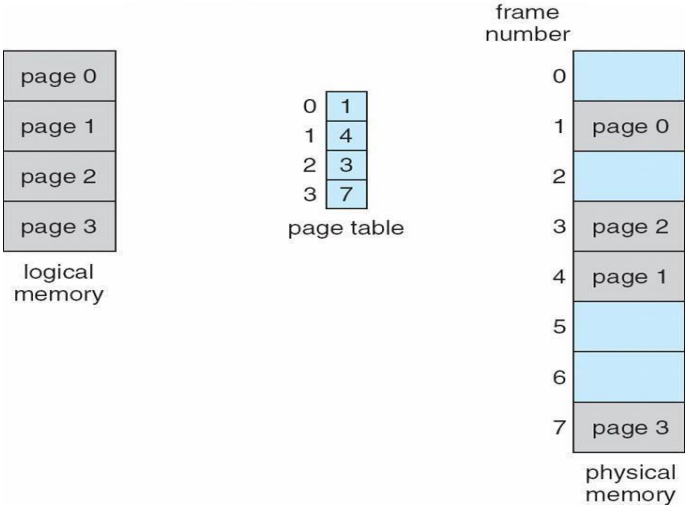


Figure 2: Paging model of logical and physical memory.

- The page size (like the frame size) is defined by the hardware.
- The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture.
- The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset.
- If the size of logical address space is 2^m and a page size is 2^n addressing units (bytes or words), then the high-order m – n bits of a logical address designate the page number, and the n low-order bits designate the page offset.

Thus, the logical address is as follows:

page number	page offset
p	d
m -n	n

- When a process requests memory (e.g. when its code is loaded in from disk), free frames are allocated from a free-frame list, and inserted into that process's page table.
- Processes are blocked from accessing anyone else's memory because all of their memory requests are mapped through their page table. There is no way for them to generate an address that maps into any other process's memory space.
- The operating system must keep track of each individual process's page table, updating it whenever the process's pages get moved in and out of memory, and applying the correct page table when processing system calls for a particular process. This all increases the overhead involved when swapping processes in and out of the CPU.

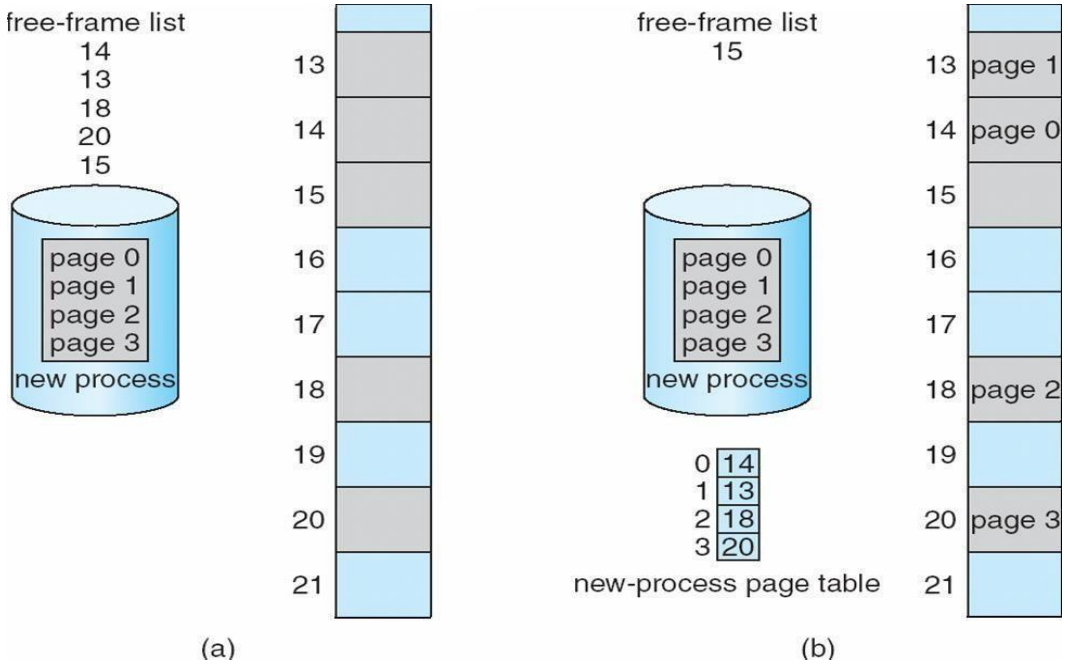


Figure: Free frames (a) before allocation and (b) after allocation.

Hardware Support

Translation Look aside Buffer

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

Working:

- When a logical-address is generated by the CPU, its page-number is presented to the TLB.
- If the page-number is found (TLB hit), its frame-number is immediately available and used to access memory
- If page-number is not in TLB (TLB miss), a memory-reference to page table must be made. The obtained frame-number can be used to access memory (Figure 1)

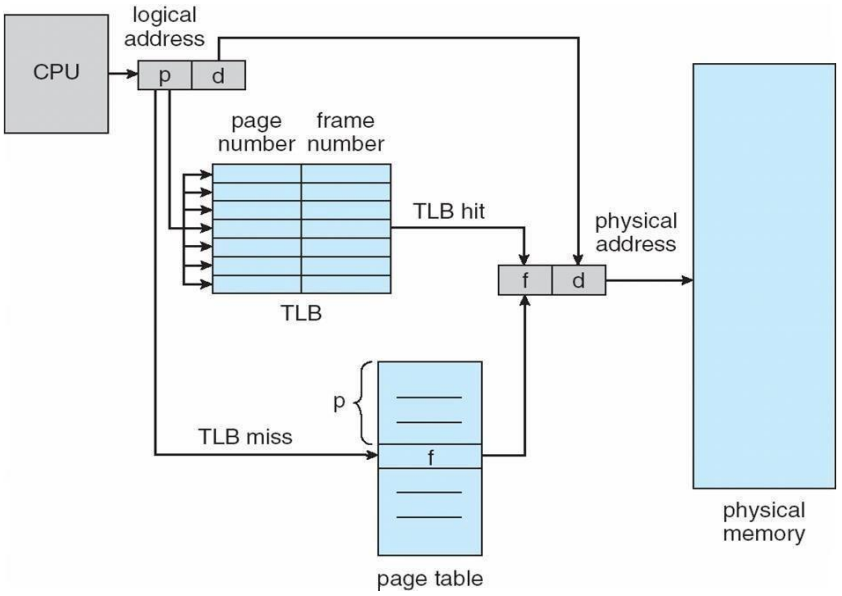


Figure 1: Paging hardware with TLB

- In addition, we add the page-number and frame-number to the TLB, so that they will be found quickly on the next reference.
- If the TLB is already full of entries, the OS must select one for replacement.
- Percentage of times that a particular page-number is found in the TLB is called hit ratio.

Advantage: Search operation is fast.
Disadvantage: Hardware is expensive.

- Some TLBs have wired down entries that can't be removed.
- Some TLBs store ASID (address-space identifier) in each entry of the TLB that uniquely identify each process and provide address space protection for that process.

Protection

- Memory-protection is achieved by protection-bits for each frame.
- The protection-bits are kept in the page-table.
- One protection-bit can define a page to be read-write or read-only.
- Every reference to memory goes through the page-table to find the correct frame-number.
- Firstly, the physical-address is computed. At the same time, the protection-bit is checked to verify that no writes are being made to a read-only page.
- An attempt to write to a read-only page causes a hardware-trap to the OS (or memory protection violation).

Valid Invalid Bit

- This bit is attached to each entry in the page-table.
- Valid bit: "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
- Invalid bit: "invalid" indicates that the page is not in the process' logical address space

Illegal addresses are trapped by use of valid-invalid bit.
The OS sets this bit for each page to allow or disallow access to the page.

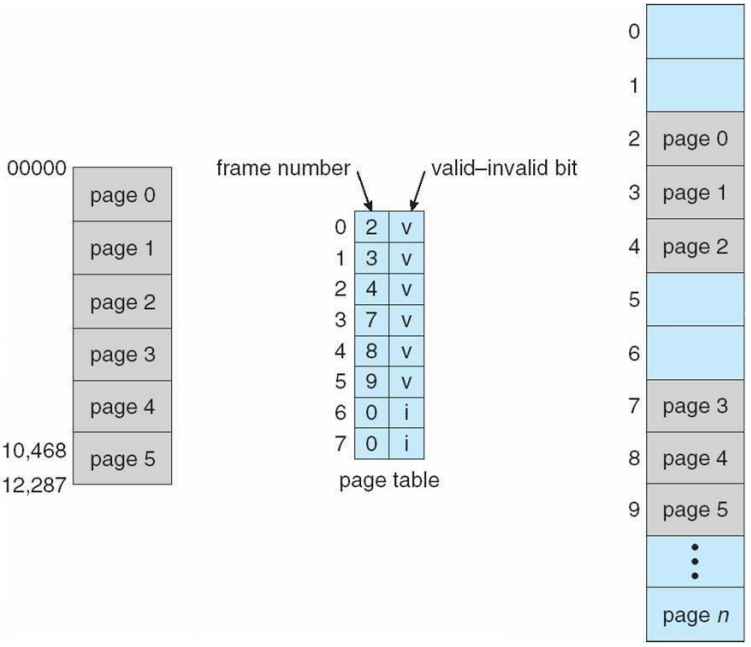


Figure: Valid (v) or invalid (i) bit in a page-table

Shared Pages

- An advantage of paging is the possibility of sharing common code.
- Re-entrant code (Pure Code) is non-self-modifying code, it never changes during execution.
- Two or more processes can execute the same code at the same time.
- Each process has its own copy of registers and data-storage to hold the data for the process's execution.
- The data for 2 different processes will be different.
- Only one copy of the editor need be kept in physical-memory (Figure 5.12).
- Each user's page-table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.

Disadvantage:
Systems that use inverted page-tables have difficulty implementing shared-memory.

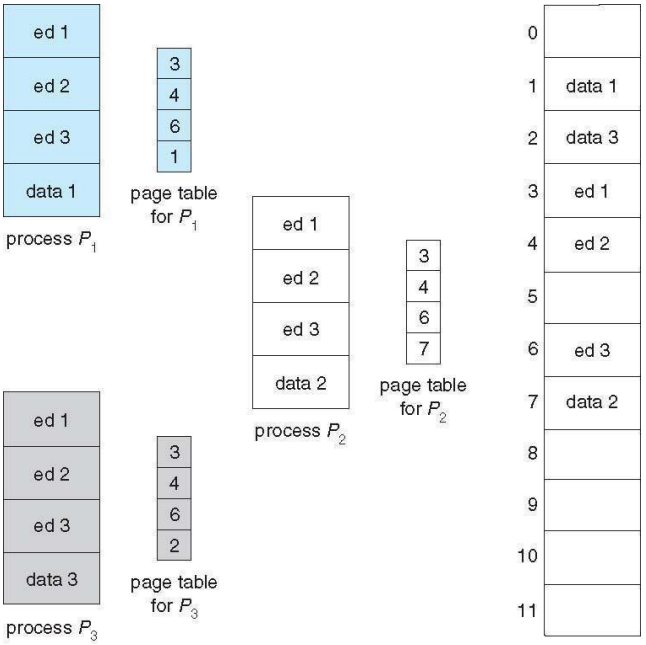


Figure: Sharing of code in a paging environment

Structure of the Page Table

The most common techniques for structuring the page table:

1. Hierarchical Paging
2. Hashed Page-tables
3. Inverted Page-tables

1. Hierarchical Paging

- Problem: Most computers support a large logical-address space (232 to 264). In these systems, the page-table itself becomes excessively large.
- Solution: Divide the page-table into smaller pieces.

Two Level Paging Algorithm:

- The page-table itself is also paged.
- This is also known as a forward-mapped page-table because address translation works from the outer page-table inwards.

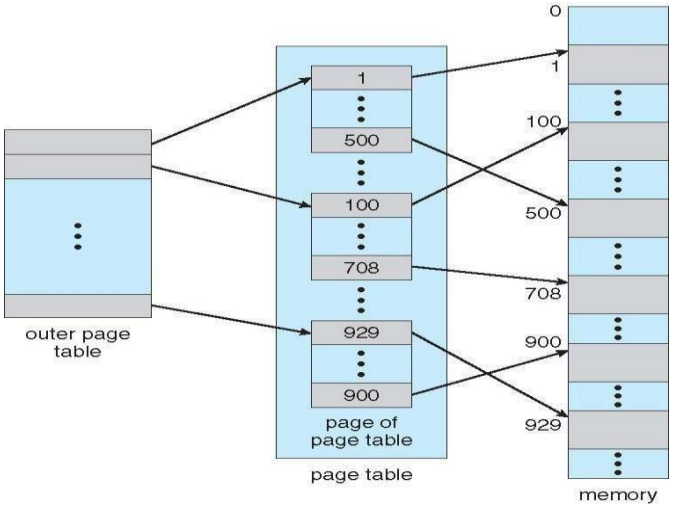


Figure: A two-level page-table scheme

For example:

Consider the system with a 32-bit logical-address space and a page-size of 4 KB.

A logical-address is divided into

→ 20-bit page-number and

→ 12-bit page-offset.

Since the page-table is paged, the page-number is further divided into

→ 10-bit page-number and

→ 10-bit page-offset.

Thus, a logical-address is as follows:

page number		page offset
p_1	p_2	d
12	10	10

- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table

The address-translation method for this architecture is shown in below figure. Because address translation works from the outer page table inward, this scheme is also known as a forward-mapped page table.

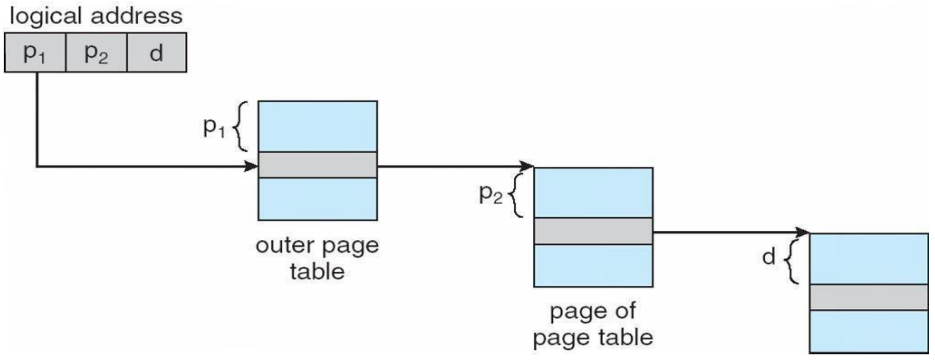


Figure: Address translation for a two-level 32-bit paging architecture

2. Hashed Page Tables

- This approach is used for handling address spaces larger than 32 bits.
- The hash-value is the virtual page-number.
- Each entry in the hash-table contains a linked-list of elements that hash to the same location (to handle collisions).
- Each element consists of 3 fields:
    1. Virtual page-number
    2. Value of the mapped page-frame and
    3. Pointer to the next element in the linked-list.

The algorithm works as follows:

- The virtual page-number is hashed into the hash-table.
- The virtual page-number is compared with the first element in the linked-list.
- If there is a match, the corresponding page-frame (field 2) is used to form the desired physical-address.
- If there is no match, subsequent entries in the linked-list are searched for a matching virtual page-number.

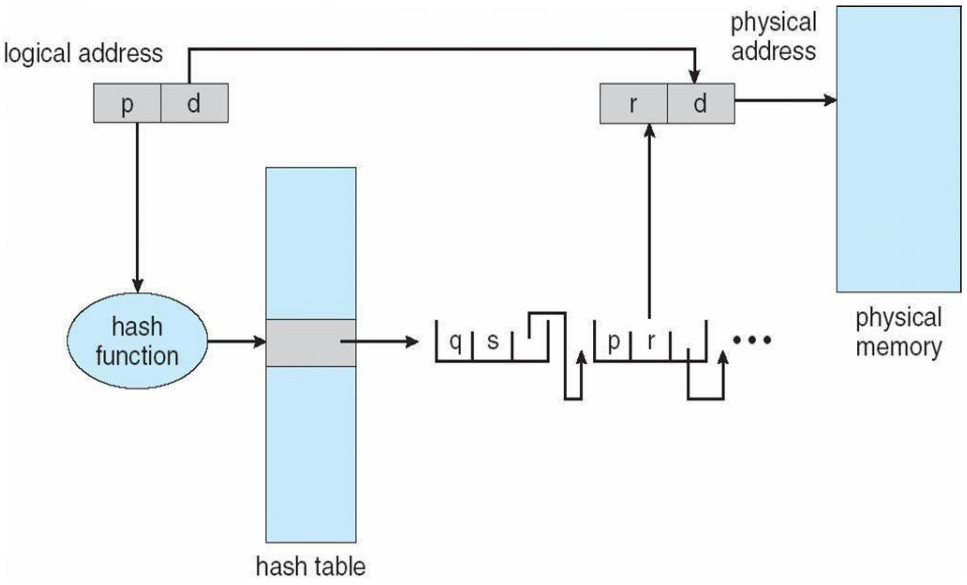


Figure: Hashed page-table

3. Inverted Page Tables

- Has one entry for each real page of memory.
- Each entry consists of virtual-address of the page stored in that real memory-location and information about the process that owns the page.
- Each virtual-address consists of a triplet <process-id, page-number, offset>.
- Each inverted page-table entry is a pair <process-id, page-number>

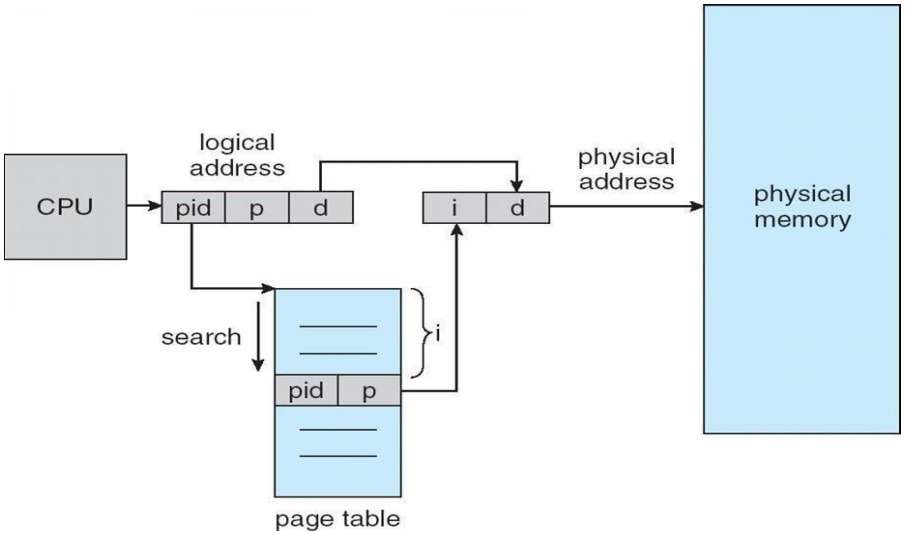


Figure: Inverted page-table

The algorithm works as follows:

1. When a memory-reference occurs, part of the virtual-address, consisting of <process-id, page-number>, is presented to the memory subsystem.
2. The inverted page-table is then searched for a match.
3. If a match is found, at entry i-then the physical-address <i, offset> is generated.
4. If no match is found, then an illegal address access has been attempted.

Advantage:

1. Decreases memory needed to store each page-table

Disadvantages:

1. Increases amount of time needed to search table when a page reference occurs.
2. Difficulty implementing shared-memory

Segmentation

Basic Method of Segmentation

- This is a memory-management scheme that supports user-view of memory (Figure 1).
- A logical-address space is a collection of segments.
- Each segment has a name and a length.
- The addresses specify both segment-name and offset within the segment.
- Normally, the user-program is compiled, and the compiler automatically constructs segments reflecting the input program.
- For ex: The code, Global variables, The heap, from which memory is allocated, The stacks used by each thread, The standard C library

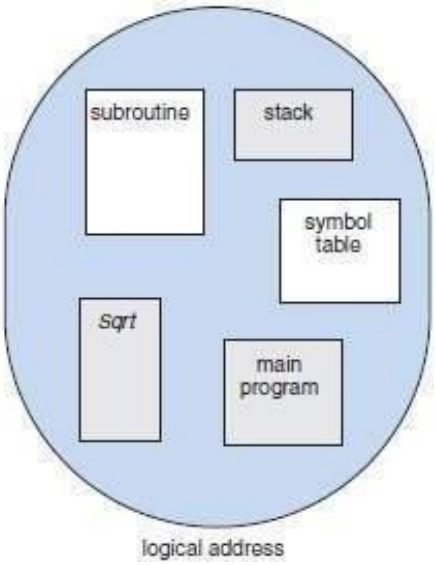


Figure: Programmer's view of a program

Hardware support for Segmentation

- Segment-table maps 2 dimensional user-defined addresses into one-dimensional physical addresses.
- In the segment-table, each entry has following 2 fields:
  1. Segment-base contains starting physical-address where the segment resides in memory.
  2. Segment-limit specifies the length of the segment (Figure 2).
- A logical-address consists of 2 parts:
  1. Segment-number(s) is used as an index to the segment-table
  2. Offset(d) must be between 0 and the segment-limit.
- If offset is not between 0 & segment-limit, then we trap to the OS(logical-addressing attempt beyond end of segment).
- If offset is legal, then it is added to the segment-base to produce the physical-memory address.

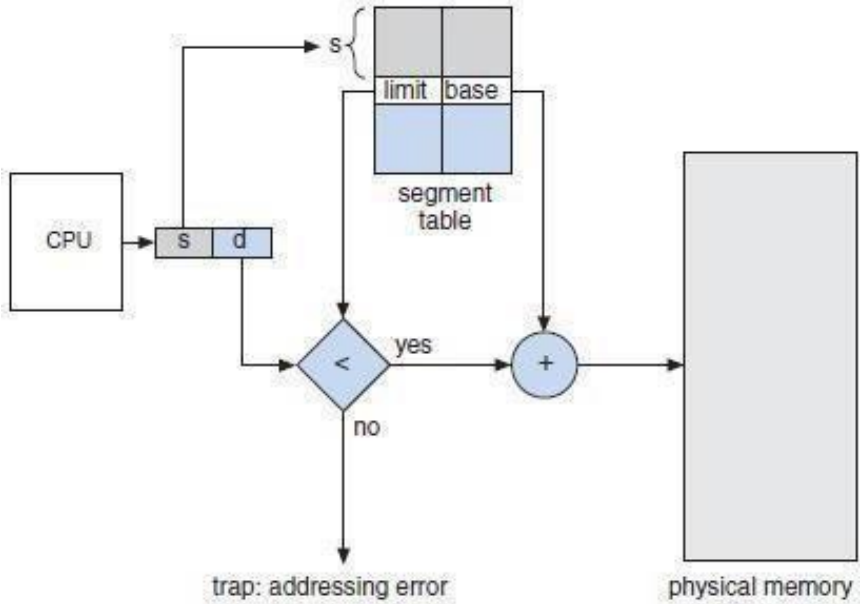


Figure: Segmentation hardware

VIRTUAL MEMORY MANAGEMENT

Background

- Virtual memory is a technique that allows for the execution of partially loaded process.
- Advantages:
  - A program will not be limited by the amount of physical memory that is available user can able to write into large virtual space.
  - Since each program takes less amount of physical memory, more than one program could be run at the same time which can increase the throughput and CPU utilization.
  - Less i/o operation is needed to swap or load user program in to memory. So Each user program could run faster.

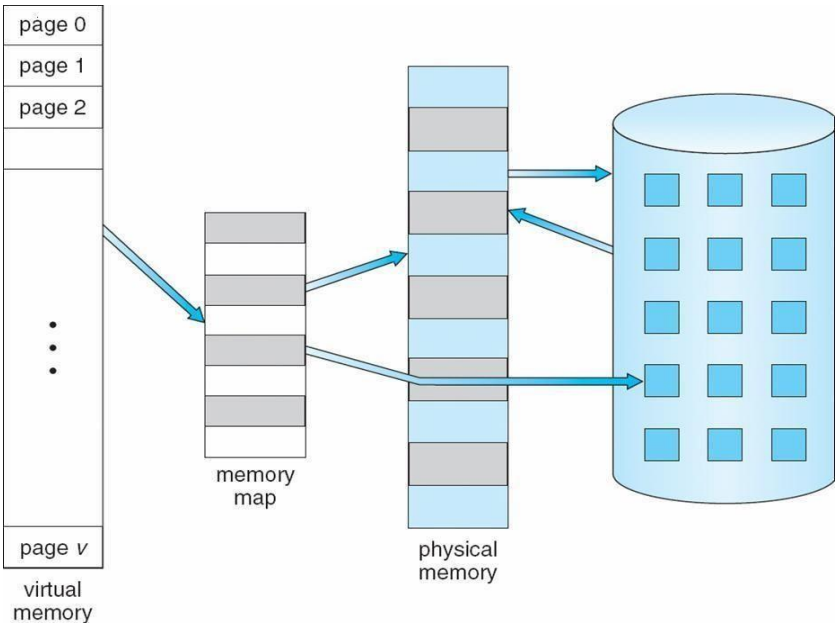


Fig:Virtualmemorythatislargerthanphysicalmemory.

- Virtual memory is the separation of users logical memory from physical memory. This separation allows an extremely large virtual memory to be provided when these is less physical memory.
- Separating logical memory from physical memory also allows files and memory to Be shared by several different processes through pagesharing.

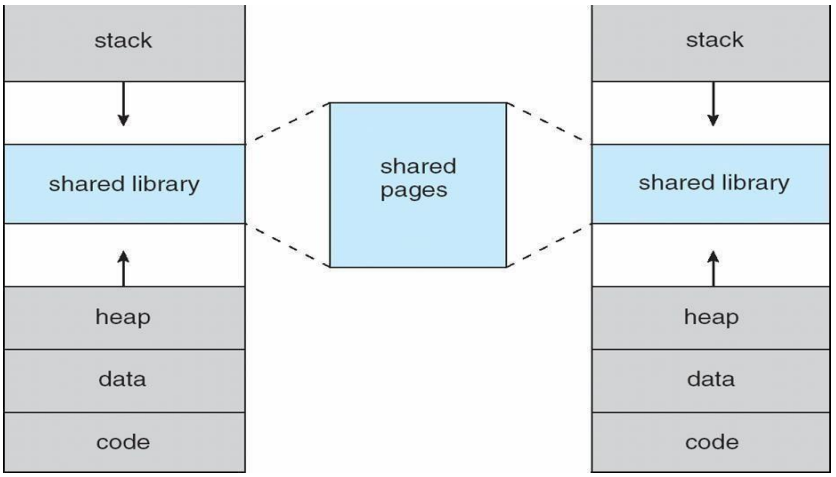


Fig:SharedLibraryusingVirtual Memory

- Virtualmemoryis implementedusingDemandPaging.
- Virtual address space: Every process has a virtual address space i.e used as the stack orheapgrowsin size.

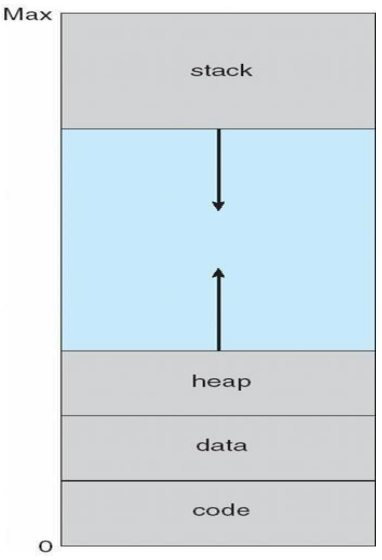


Fig:Virtualaddressspace

DEMANDPAGING

- A demand paging is similar to paging system with swapping when we want to execute aprocess we swap the process the in to memory otherwise it will not be loaded in tomemory.
- A swapper manipulates the entire processes, where as a pager manipulates individualpagesofthe process.
  - Bringapageintomemoryonlywhenit isneeded
  - LessI/Oneeded
  - Lessmemoryneeded
  - Fasterresponse

- Moreusers
- Pageisneeded⇒referenceto it
- invalidreference⇒abort
- not-in-memory⇒bringtomemory
- ***Lazyswapper***–neverswapsapageintomemoryunlesspagewillbeneeded
- Swapperthatdealswithpagesisa**pager.**

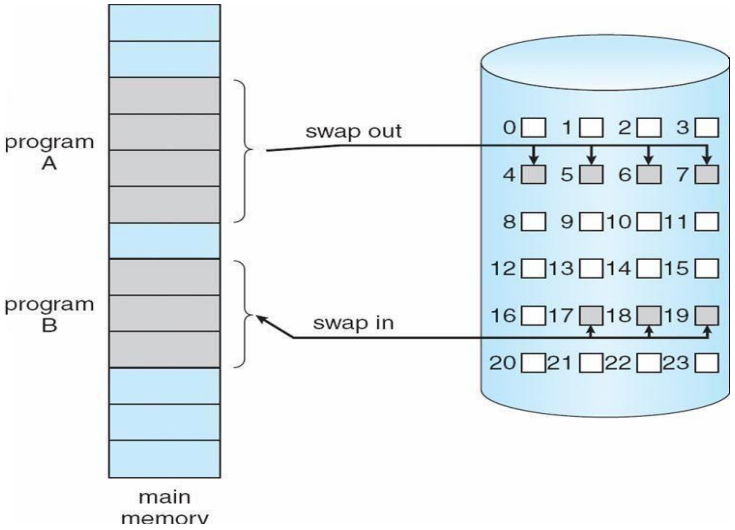


Fig:Transferofapaged memoryintocontinuousdiskspace

- **Basic concept:** Instead of swapping the whole process the pager swaps only the necessarypages in to memory. Thus it avoids reading unused pages and decreases the swap time andamountofphysicalmemoryneeded.
- The valid-invalid bit scheme can be used to distinguish between the pages that are on the diskandthatareinmemory.
  - Witheachpagetableentryavalid–invalidbitisassociated
  - (**v**⇒in-memory,**i**⇒not-in-memory)

  - Initiallyvalid–invalidbitissettoionallentries
  - Exampleof apagetablesnapshot:

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
....	
	i
	i

page table

- Duringaddresstranslation,ifvalid–invalidbitinpagetableentryis**I**⇒pagefault.

- Ifthebitisvalidthenthepageisbothlegalandisinmemory.
  - Ifthebitisinvalidtheneitherpageisnotvalid orisvalidbutiscurrentlyonthedisk. Marking

a page as invalid will have no effect if the processes never access to that page. Suppose if itaccess the page which is marked invalid, causes a page fault trap. This may result in failure ofOStobringthedesiredpageintomemory.

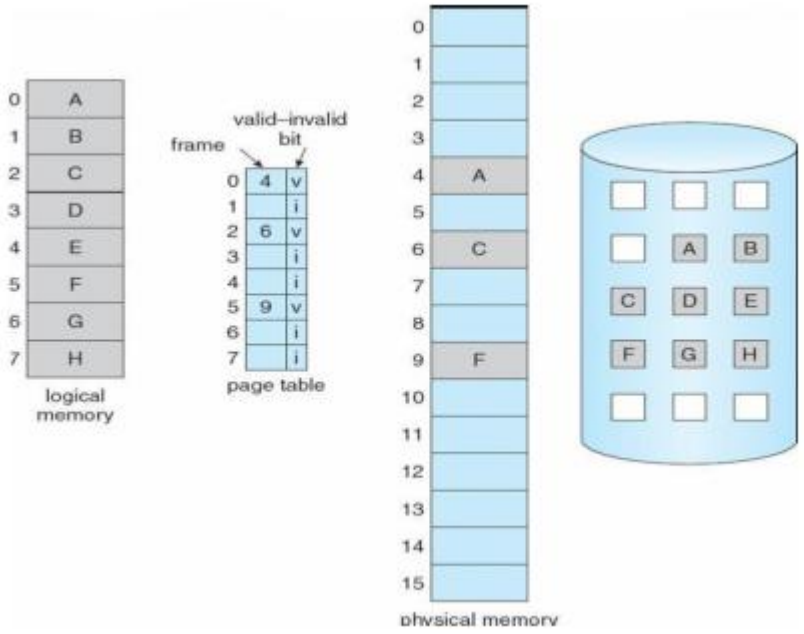


Fig:PageTable whensome pages arenot in mainmemory

PageFault

Ifapageisneededthatwas notoriginallyloaded up,then a***pagefaulttrap***isgenerated.

Stepsin HandlingaPageFault

1. Thememory addressrequestedis first checked,tomakesureitwasavalidmemoryrequest.
2. Ifthereferenceistoaninvalidpage,theprocessisterminated.Otherwise,ifthepageisnotpresentin memory,itmustbepagedin.
3. Afreeframeislocated, possiblyfromafree-framelist.
4. Adiskoperationisscheduledtobringinthenecessarypagefromdisk.
5. After the page is loaded tomemory, the process's page table is updated with thenewframe number, and the invalid bit is changed to indicate that this is now a valid pagereference.
6. Theinstructionthatcausedthepagefaultmustnowberestartedfromthebeginning

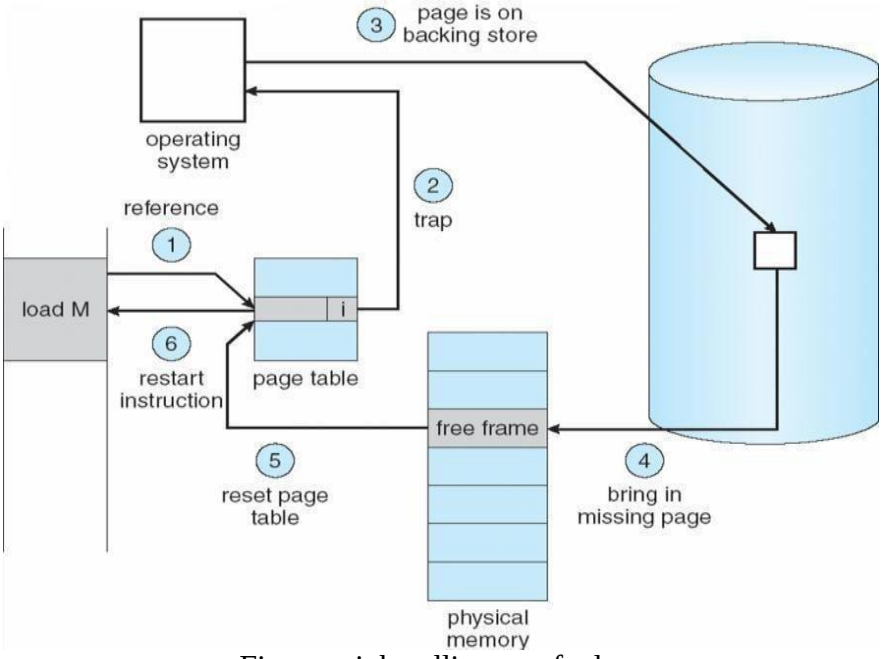


Fig:stepsinhandlingpagefault

PureDemandPaging: Neverbringapageintomainmemoryuntilit isrequired.

- We canstartexecuting aprocesswithoutloadinganyofitspagesinto mainmemory.
- Pagefaultoccursforthenonmemoryresidentpages.
- Afterthepageisbroughtintomemory,processcontinues toexecute.
- Againpagefaultoccursforthenextpage.

Hardwaresupport: Fordemandpagingthesamehardwareisrequiredaspagingandswapping.

1. Pagetable:-Has theabilitytomarkanentryinvalidthroughvalid-invalid bit.
2. Secondarymemory:-Thisholdsthepagesthat arenotpresent in main memory.

PerformanceofDemandPaging: Demandpagingcanhavesignificanteffectontheperformanceofthecomputer system.

- LetPbetheprobabilityofthepagefault(0<=P<=1)
- **Effectiveaccesstime=(1-P)*ma+P *pagefault.**
  - WhereP=pagefaultand ma=memoryaccesstime.
- Effectiveaccesstimeisdirectlyproportionaltopagefaultrate.Itisimportanttokeeppagefaultratelowindemandpaging.

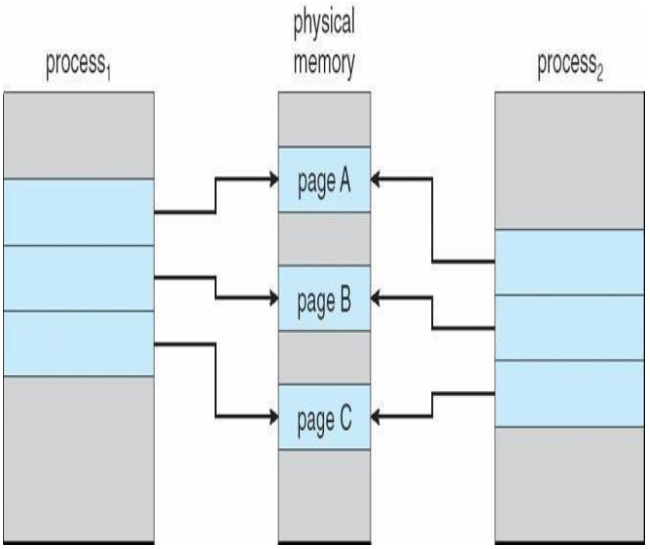
DemandPagingExample

- Memoryaccesstime=200 nanoseconds
- Averagepage-faultservicetime=8milliseconds
- EAT=(1–p)x200+p(8milliseconds)
  =(1 –px200+px8,000,000
  =200+ px 7,999,800
- Ifoneaccessoutof1,000causesapagefault,thenEAT=8.2microseconds.Thisisaslowdownbyafactorof40.

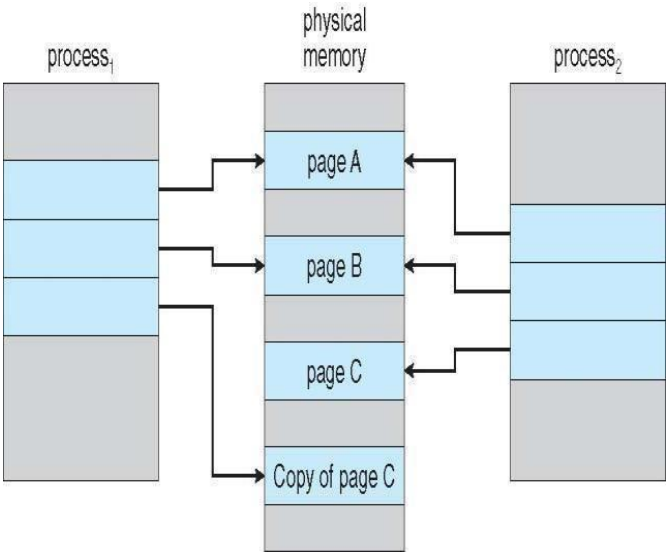
COPY-ON-WRITE

- Technique initially allows the parent and the child to share the samepages.Thesepagesaremarkedascopyon-writepagesi.e.,ifeitherprocesswritestoasharedpage,acopyofsharedpageiscreated.
- *Eg*:-If a child process try to modify a page containing portions of the stack; the OSrecognizes them as a copy-on-write page and create a copy of this page and maps it onto the address space of the child process. So the child process will modify its copiedpageandnotthepagebelongingtoparent.Thenewpagesareobtainedfromthepooloffree pages.
- The previous contents of pages are erased before getting them into main memory. Thisiscalled**Zero –onfill demand.**

a) BeforeProcess1modifiespageC



b) Afterprocess1modifiespageC



PAGEREPLACEMENT

- Page replacement policy deals with thesolution ofpages inmemory to be replacedby anewpagethat must bebroughtin.Whenauserprocessisexecutingapagefaultoccurs.
- Thehardwaretrapstotheoperatingsystem,whichchecksttheinternaltabletoseethatthisisapage faultandnotanillegalmemoryaccess.
- The operating system determines where the derived page is residing on the disk, and thisfindsthattherearenofreeframesonthelistoffreeframes.
- When all the frames are in main memory, it is necessary to bring a new page to satisfy thepage fault, replacement policy is concerned with selecting a page currently in memory to bereplaced.
- Thepagei,etoberemovedshouldbethe pagei,eleastlikelytobe referencedinfuture.

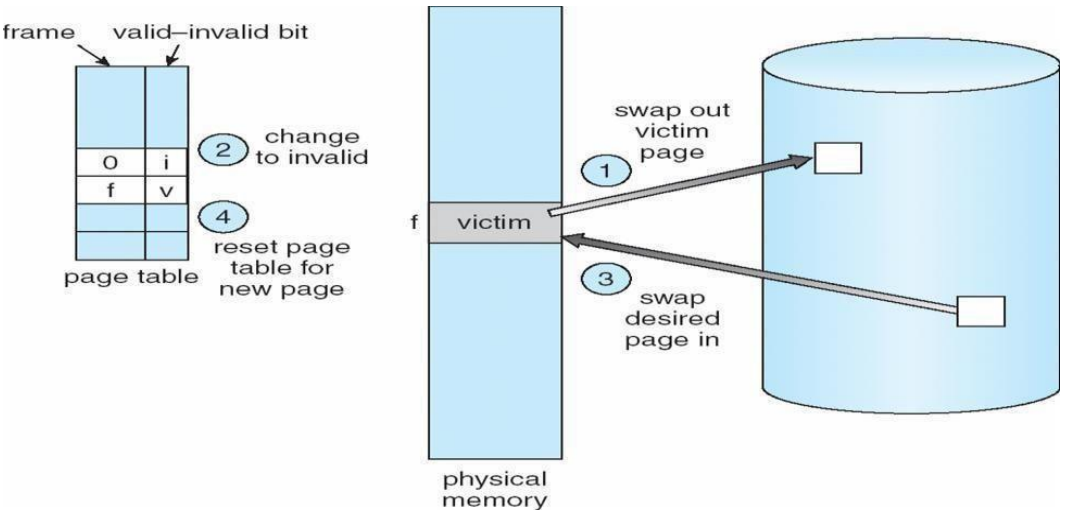


Fig:PageReplacement

WorkingofPage ReplacementAlgorithm

1. Findthelocationof derivedpageonthe disk.
2. FindafreeframexIfthereisafreeframe,useit.xOtherwise,useareplacementalgorithmto selectavictim.
   - Writethevictimpagetothedisk.
   - Changethepageandframetablesaccordingly.
3. Readthedesiredpageintothefreeframe;changethepageandframetables.
4. Restarttheuserprocess.

VictimPage

- Thepagethatissupportedoutof physicalmemoryiscalledvictimpage.
- Ifno framesarefree,thetwopagetransformscome(outandonein)areread.Thiswillseetheeffectiveaccesstime.
- Eachpageorframemayhaveadirty(modify)bitassociatedwiththehardware.Themodifybitforapageissetbythehardwarewheneveranywordorbyteinthepageis

writteninto,indicating thatthepagehasbeenmodified.

- When we select the page for replacement, we check its modify bit. If the bit is set, then thepageismodifiedsinceitwasreadfromthedisk.
- If the bit was not set, the page has not been modified since it was read into memory.Therefore, if the copy of the page has not been modified we can avoid writing the memorypagetothedisk,if itisalreadythere.Sumpagescannotbemodified.

Modifybit/Dirty bit:

- Eachpage/framehas a modifybitassociated withit.
- Ifthepageisnotmodified(read-only)thenonecandiscardsuchpagewithoutwritingitontothedisk**.** Modifybitofsuchpageissetto0.
- Modifybitissetto1,ifthepagehasbeenmodified.Suchpagesmustbewrittentothedisk.
- Modifybitisusedtoreduceoverheadofpagetransfers–onlymodifiedpagesarewrittentodisk

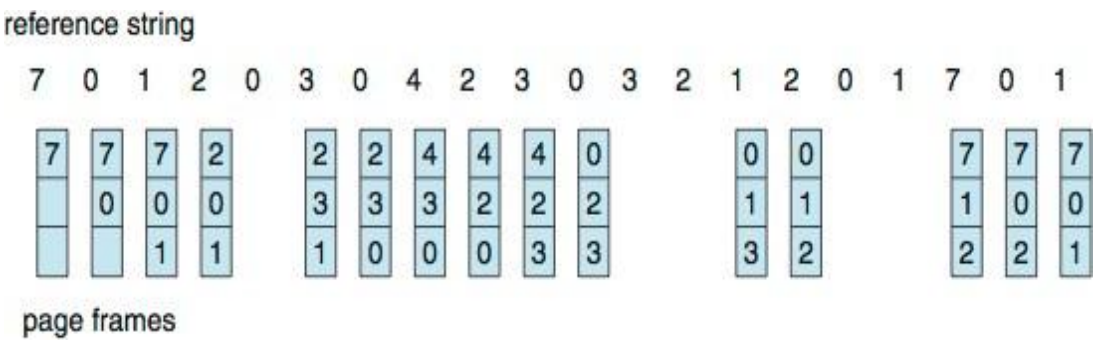
PAGEREPLACEMENTALGORITHMS

- Wantlowestpage-faultrate
- Evaluatealgorithmbyrunningitonaparticularstringofmemoryreferences(referencestring)andcomputingthenumberofpagefaultsonthatstring
- Inallourexamples,thereferencestringis

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

FIFOAlgorithm:

- Thisisthesimplestpagereplacementalgorithm.AFIFOreplacementalgorithmassociateseachpage thetimewhen thatpagewasbrought intomemory.
- WhenaPageisto bereplacedtheoldestoneis selected.
- Wereplacethequeueattheheadofthequeue.Whenapageisbroughtintomemory,weinsertitattheta ilofthequeue.
- Inthefollowingexample,areferencestringisgivenandthereare3freeframes.Thereare20pagerequests,whichresultsin15page faults



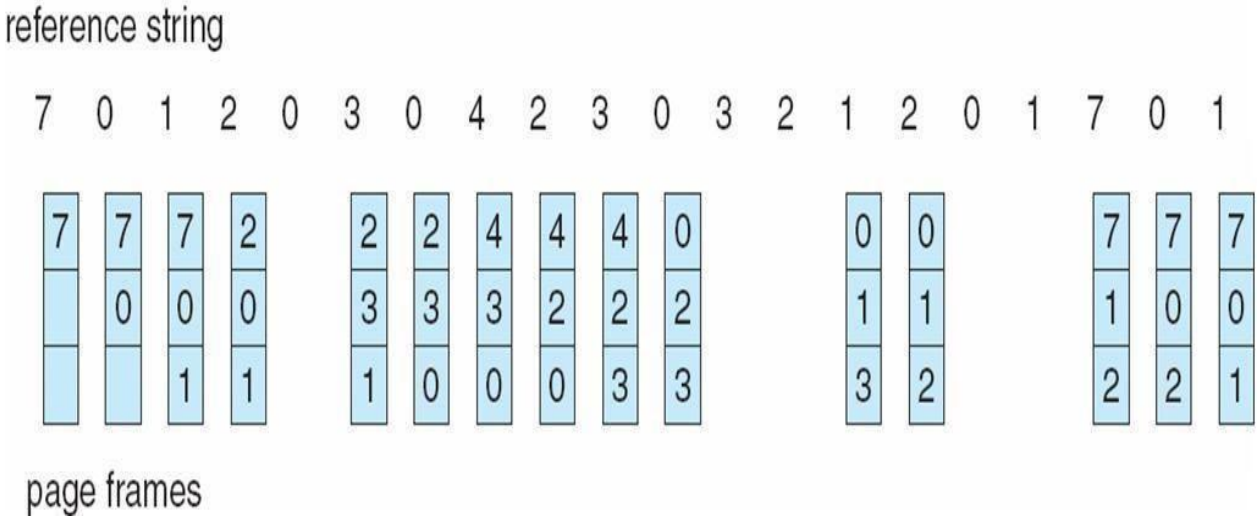
Belady'sAnomaly

- Forsomepagereplacementalgorithm,thepagefaultmayincreaseasthenumberofallocatedframes increases.FIFOreplacement algorithmmayfacethisproblem.

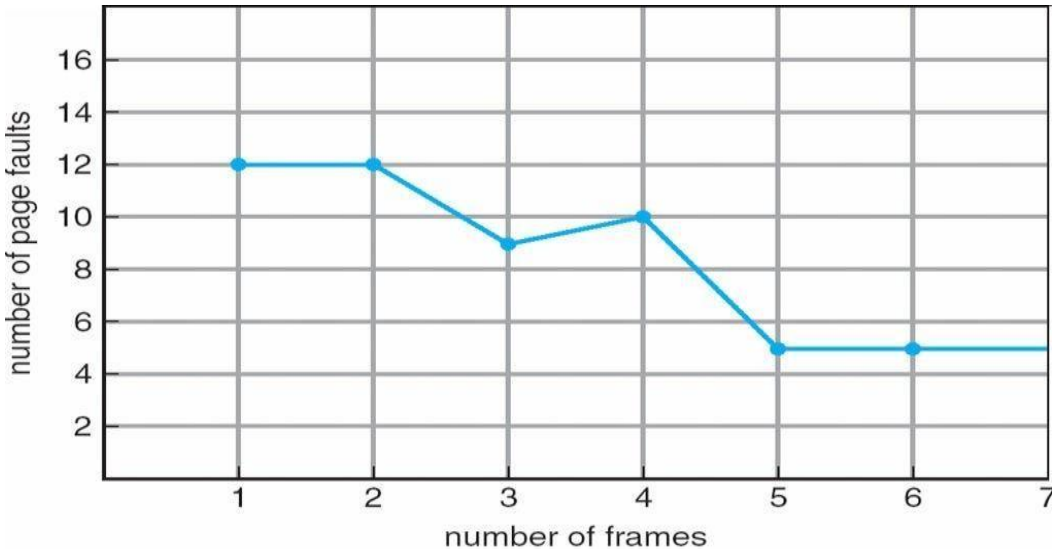
moreframes⇒ morepagefaults

Example: Consider thefollowingreferencesstringwithframesinitiallyempty.

- Thefirstthreereferences(7,0,1)casespagefaultsandarebroughtintotheemptyframes.
- Thenextreferences2replacespage7becausethepage7wasbroughtinfirst.xSince0isthenext referencesand0isalreadyinmemoryehasnopagefaults.
- Thenextreferences3resultsinpage0beingreplacedsothatthenextreferencesto0causerpagefault. This willcontinuetilltheendof string.Thereare 15faultsalltogether.



FIFOIllustratingBelady'sAnomaly



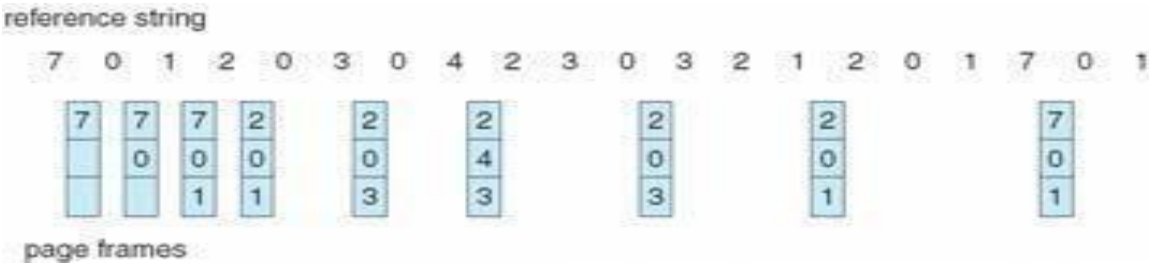
OptimalAlgorithm

- Optimalpagereplacementalgorithmis mainlytosolvetheproblemofBelady'sAnomaly.
- Optimalpagereplacementalgorithmhasthelowestpagefaultrateofallalgorithms.
- Anoptimalpage replacementalgorithmexists andhasbeencalledOPT.

The working is simple "Replace the page that will not be used for the longest period of time"Example:considerthefollowingreferencestring

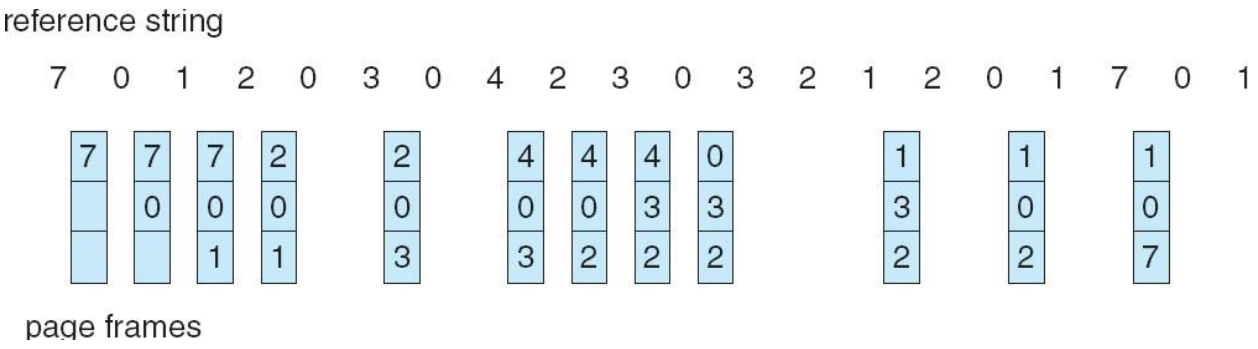
- Thefirstthreereferencescausefaultsthatfillthethreeemptyframes.
- The references to page 2 replaces page 7, because 7 will not be used until reference 18. xThepage0willbeusedat5andpage1at 14.
- With only 9 page faults, optimal replacement is much better than a FIFO, which had 15faults. This algorithm is difficult t implement because it requires future knowledge ofreferencestrings.
- Replacepagethatwill notbeusedforlongest periodoftime

OptimalPageReplacement



LeastRecently Used(LRU)Algorithm

- The***LRU(LeastRecentlyUsed)***algorithm,predictsthatthepagethathasnotbeenusedinthelongesttimeistheonethatwillnot beusedagaininthenearfuture.
- SomeviewLRUasanalogoustoOPT,butherewelookbackwardsintimeinsteadofforwards.



ThemainproblemtohowtoimplementLRU istheLRUrequiresadditionalh/wassistance.

Twoimplementationarepossible:

1. **Counters:** In this we associate each page table entry a time -of -use field, and add tothecpualogicalclockorcounter.Theclockisincrementedforeachmemoryreference. When a reference to a page is made, the contents of the clock register arecopiedtothetime-of-usefieldinthepagetableentryforthatpage.Inthiswaywehave the time of last reference to each page we replace the page with smallest timevalue.Thetimemustalsobemaintainedwhenpagetablesarechanged.
2. **Stack:** Another approach to implement LRU replacement is to keep a stack of pagenumbers when a page is referenced it is removed from the stack and put on to the topof stack. In this way the top of stack is always the most recently used page and thebottom in least recently used page. Since the entries are removed from the stack it isbestimplementbyadoublylinkedlist.Withaheadandtailpointer.

Note: Neither optimal replacement nor LRU replacement suffers from Belady's Anamoly.Thesearecalledstackalgorithms.

LRU-ApproximationPageReplacement

- Manysystemsoffersomedegreeofhardwaresupport,enoughto approximateLRU.
- Inparticular,manysystemsprovidea***referencebit***foreveryentryinapagetable,whichis set anytime that page is accessed. Initially all bits are set to zero, and they can also all becleared at any time. One bit distinguishes pages that have been accessed since the last clearfromthosethathavenotbeenaccessed.

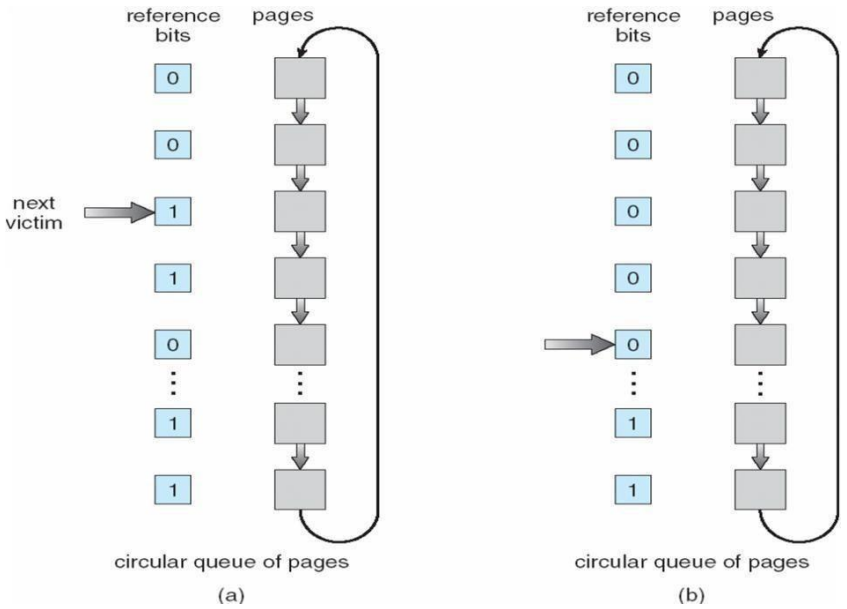
Additional-Reference-BitsAlgorithm

- An8-bitbyte (referencebit)isstoredforeachpageinatableinmemory.
- At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to theoperating system. The operating system shifts the reference bit for each page into thehigh-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-orderbit.
- These8-bitshiftregisterscontainthehistoryof pageuseforthelasteighttimeperiods.
- If the shift register contains 00000000, then the page has not been used for eight timeperiods.
- A page with a history register value of 11000100 has been used more recently than onewithavalueof01110111.

Second-chance(clock)pagereplacement algorithm

- The***secondchancealgorithm***isaFIFOreplacementalgorithm,exceptthereferencebitisusedto givepagesasecondchanceatstayinginthepagetable.
- Whenapagemustbereplaced,thepagetableisscannedinaFIFO(circularqueue)manner.
- Ifapageisfoundwithitsreferencebitas'0',thenthatpageisselectedasthenextvictim.

- If the reference bitvalueis‘1’, then the page is given a second chance and its reference bitvalueiscleared (assigned as‘0’).
- Thus, a page that is given a second chance will not be replaced until all other pages havebeen replaced (or given second chances). In addition, if a page is used often, then it setsitsreferencebitagain.
- Thisalgorithmisalsoknownasthe***clock***algorithm.



EnhancedSecond-ChanceAlgorithm

- The**enhancedsecond chancealgorithm**looksatthereferencebitandthe modifybit(dirtybit)asanorderedpage,andclassifiespagesintooneoffourclasses:
    1. (0,0)-Neitherrecentlyusednormodified.
    2. (0,1)-Notrecentlyused,butmodified.
    3. (1,0)-Recentlyused,butclean.
    4. (1,1)-Recentlyusedandmodified.
- This algorithm searches the page table in a circular fashion, looking for the first page it canfind in the lowest numbered category. i.e. it first makes a pass looking for a (0, 0), and thenifitcan'tfindone, itmakes anotherpasslookingfor a(0,1),etc.
- The main difference between this algorithm and the previous one is the preference forreplacingcleanpagesifpossible.

CountBasedPageReplacement

Thereismanyotheralgorithmsthatcanbeusedforpagereplacement,wecankeepacounterofthenumberofreferencesthathasmadetoapage.

a) **LFU**(leastfrequentlyused):

This causes the page with the smallest count to be replaced. The reason for this selectionisthatactivelyusedpageshouldhavealargereferencecount.

This algorithm suffers from the situation in which a page is used heavily during theinitial phase of a process but never used again. Since it was used heavily, it has a largecountandremains inmemoryeventhoughit is no longerneeded.

b)**MFU**Algorithm:

based on the argument that the page with the smallest count was probably just brought inandhasyetto be used

ALLOCATIONOFFRAMES

- The absolute minimum number of frames that a process must be allocated is dependentonsystem architecture.
- Themaximumnumberisdefinedbytheamountof availablephysicalmemory.

AllocationAlgorithms

After loading of OS, there are two ways in which the allocation of frames can be done totheprocesses.

1. **Equal Allocation**- If there are m frames available and n processes to share them, eachprocessgetsm/nframes,andtheleftover'sarekeptinafree-framebufferpool.
2. **Proportional Allocation** - Allocate the frames proportionally depending on the sizeof the process. If the size of process i is S_i, and S is the sum of size of all processes inthe system, then the allocation for process P_i is a_i= m * S_i/ S. where m is the freeframesavailableinthesystem.

- Consider a system with a 1KB frame size. If a small student process of 10 KB and aninteractive database of 127 KB are the only two processes running in a system with 62freeframes.
- withproportionalallocation,wewouldsplit62framesbetweentwoprocesses,asfollows
  m=62,S=(10+127)=137
  Allocationforprocess1=62X10/137~4Allocationforprocess2=62X127/137 ~57
  Thusallocates4framesand57framestostudentprocessanddatabaserespectively.
- Variationsonproportionalallocationcouldconsiderpriorityofprocessratherthanjusttheirsize.

GlobalversusLocalAllocation

- Pagereplacementcanoccurbothatlocalorgloballevel.
- With local replacement, the number of pages allocated to a process is fixed, and pagereplacement occursonlyamongstthepagesallocatedtothisprocess.
- With global replacement, any page may be a potential victim, whetherit currentlybelongstotheprocessseekingafreeframeornot.
- Local page replacement allows processes tobetter control their own page fault rates,and leads to more consistent performance of a given process over different system loadlevels.

- Global page replacement is over all more efficient, and is the more commonly usedapproach.

Non-UniformMemoryAccess (New)

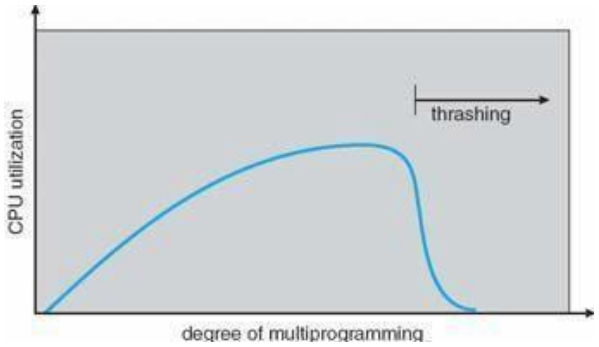
- Usuallythe timerequiredtoaccessall memoryinasystemisequivalent.
- This may not be the case in multiple-processor systems, especially where each CPU isphysically located on a separate circuit board which also holds some portion of theoverallsystemmemory.
- In such systems, CPU s can access memory that is physically located on the same boardmuchfasterthanthememoryontheotherboards.
- Thebasicsolutionis akintoprocessoraffinity -Atthesametimethatwetry toschedule processes on the same CPU to minimize cache misses, we also try to allocatememoryforthoseprocessesonthesameboards,tominimizeaccesstimes.

THRASHING

- If the number of frames allocated to a low-priority process falls below the minimumnumberrequiredbythe computerarchitecturethenwesuspendtheprocessexecution.
- Aprocessisthrashingifitisspending moretimeinpagingthanexecuting.
- If the processes do not have enough number of frames, it will quickly page fault.During this it must replace some page that is not currently in use. Consequently itquicklyfaultsagainandagain.
- The process continuesto fault, replacing pages for whichit thenfaults and bringsback. This high paging activity is called thrashing. The phenomenon of excessivelymovingpagesbackandforthb/w memoryandsecondaryhasbeencalled***thrashing***.

CauseofThrashing

- Thrashingresultsinsevereperformanceproblem.
- Theoperatingsystemmonitorsthecpuutilizationislow.Weincreasethedegreeofmultiprogramming byintroducingnewprocesstothesystem.
- Aglobalpagereplacementalgorithmreplacespageswithnoregardstotheprocesstowhichthey belong.



Thefigureshowsthethrashing

- Asthedegreeofmultiprogrammingincreases,moreslowlyuntilamaximumis reached.Ifthedegreeofmultiprogrammingisincreasedfurtherthrashingsetsinandthecpuutilization dropssharply.
- Atthispoint,toincreasesCPUutilizationandstopthrashing,wemustincreasedegreeofmultiprogramming.
- wecanlimittheeffectofthrashingbyusingalocalreplacementalgorithm.Topreventthrashing ,wemustprovideaprocessasmanyframesasitneeds.

LocalityofReference:

- Astheprocessexecutesit movesfromlocalitytolocality.
- Alocalityisasetofpagesthatareactivelyused.
- Aprogrammayconsistof severaldifferentlocalities,whichmayoverlap.
- Localityiscausedbyloopsincodethatfindtoreferencearraysandotherdatastructuresbyindices.
- Theorderedlistofpagenumberaccessedbyaprogramiscalledreferencestring.
- Localityis oftwotypes:
  1.spatiallocality 2.temporallocality

Workingsetmodel

- Working set model algorithm uses the current memory requirements to determine thenumberofpageframestoallocatetotheprocess,aninformaldefinitionis“thecollection of pages that a process is working with and which must be resident if theprocess to avoid thrashing”. The idea is to use the recent needs of a process to predictitsfuture reader.
- The working set is an approximation of programs locality. Ex: given a sequence ofmemory reference, if the working set window size to memory references, then workingsetattimet1 is{1,2,5,6,7}andatt2ischangedto{3,4}
- At any given time, all pages referenced by a process in its last 4 seconds of executionareconsidered to compromise itsworking set.
- Aprocess willneverexecuteuntilitsworking setisresidentin main memory.
- Pagesoutsidetheworkingsetcanbediscardedatanymovement.
- Workingsetsarenotenoughandwemustalsointroducebalanceset.
  - If the sum of the working sets of all the run able process is greater than the sizeofmemorytherefusesomeprocessforawhile.
  - Divide the run able process into two groups, active and inactive. The collectionof active set is called the balance set. When a process is made active its workingsetis loaded.
  - Some algorithm must be provided for moving process into and out of the balanceset. As a working set is changed, corresponding change is made to the balanceset.
  - Working set presents thrashing by keeping the degree of multi programming ashigh as possible. Thusif optimizes the CPUutilization. Themain disadvantageofthisiskeepingtrackoftheworkingset.

Page-FaultFrequency

- When page- fault rate is too high, the process needs more frames and when it is too low,the process may havetoomanyframes.
- The upper and lower bounds can be established on the page-faultrate.If the actual page-fault rate exceeds the upper limit, allocate the process another frame or suspend the process.
- If the page-fault rate falls below the lower limit, remove a frame from the process.Thus,we can directly measure and control the page- fault rate to prevent thrashing.

QUESTION BANK

DEADLOCKS

1. What are deadlocks? What are its characteristics? Explain the necessary conditions for its occurrence.
2. Explain the process of recovery from deadlock.
3. Describe RAG:
   i) With deadlock
   ii) With a cycle but no deadlock
4. What is Resource Allocation Graph (RAG)? Explain how RAG is very useful in describing deadly embrace (dead lock) by considering your own example.
5. With the help of a system model, explain a deadlock and explain the necessary conditions that must hold simultaneously in a system for a deadlock to occur.
6. Explain how deadlock can be prevented by considering four necessary conditions cannot hold.
7. Using Banker's algorithm determines whether the system is in a safe state.
8. How is a system recovered from deadlock? Explain the different methods used to recover from deadlock.
9. Explain deadlock detection with algorithm and example
10. Define the terms: safe state and safe sequence. Give an algorithm to find whether or not a system is in a safe state.

MEMORY MANAGEMENT

1. Explain the multistep processing of a user program with a neat block diagram.
2. Distinguish between internal and external fragmentation.
3. Explain segmentation with an example.
4. Explain with a diagram, how TLB is used to solve the problem of simple paging scheme.
5. With a supporting paging hardware, explain in detail concept of paging with an example for a 32-byte memory with 4-type pages with a process being 16-bytes. How many bits are reserved for page number and page offset in the logical address. Suppose the logical address is 5, calculate the corresponding physical address, after populating memory and page table.
6. What are the draw backs of contiguous memory allocation?
7. Consider a paging system with the page table stored in memory.
   i. if a memory reference takes 200 nano seconds, how long does a paged memory reference take?
   ii. if we add associative register and 75 percentage of all page table references are found in the associative registers, what is the effective memory access time? (Assume that finding a page table entry in the associative memory/registers takes zero time, if the entry is found).
8. Distinguish between:
   i. Logical address space and physical address space.
   ii. Internal fragmentation and external fragmentation.
   iii. Paging and segmentation.
9. Explain with the help of supporting hardware diagram how the TLB improves the performance of a demand paging system.
10. Explain the concept of forward mapped page table.
11. What is fragmentation? Explain two types of memory fragmentation
12. What is swapping? Explain in detail.
13. What do you mean by address binding? Explain with the necessary steps, the binding of instructions and data to memory addresses.