# MODULE – III
# Deadlocks & Memory Management

## 4.1 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. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances.
- A process utilize a resource in only the following sequence:
    - **Request**. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
    - **Use**. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
    - **Release**. The process releases the resource.
- **Definition of Deadlock**: 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.
- Here event means resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, files, semaphores, and monitors).

## 4.2 Necessary Conditions for Deadlock (Deadlock Characterization)

All four conditions must hold for a deadlock to occur.

1. **Mutual exclusion.** At least one resource must be held in a non-sharable mode; that is, onlyone 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 {P1, P2, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ……, Pn-1 is waiting for a resource held by Pn and Pn is waiting for a resource held by P0.

## 4.3 Resource-Allocation Graph

- Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation** graph. This 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: 1) Active **processes** {P1, P2,….Pn} represented by circles and 2) **resource types** represented by rectangle with dots representing number of instances.
- The set of edges E is partitioned into **two** different types:
- **1) Allocation (Assignment) edge**, a directed edge from resource type Rj to Process Pi that is Rj→Pi, signifies that an instance of resource type Rj is allocated to Process Pi.
- **2) Request edge**, a directed edge from process Pi to resource type Rj that is Pi→Rj, signifies that process Pi requests an instance of resource type Rj.
- Example: The sets P, R and E:
    - P == {P1, P2, P3}

- R== {R1, R2, R3, R4}
- E == {Pl →Rl, P2 → R3, Rl→P2, R2→P2,R2→ Pl, R3→P3}

- Resource instances:
  - One instance of resource type R1
  - Two instances of resource type R2
  - One instance of resource type R3
  - Three instances of resource type R4.
- Process states:
  - Process P1 is holding an instance of resource type R2 and is waiting foran instance of resource type R1.
  - Process P2 is holding an instance of R1 and an instance of R2 and iswaiting for an instance of R3.
  - Process P3 is holding an instance of R3.

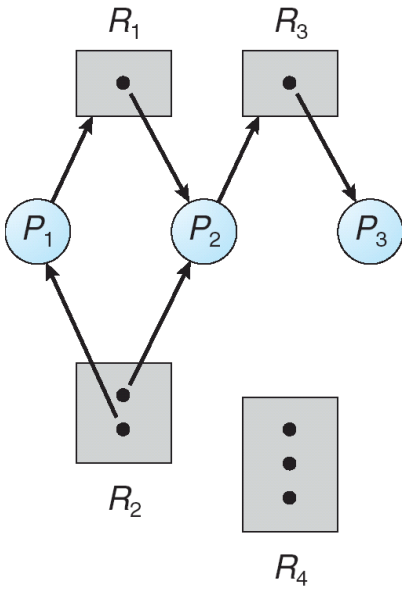



Figure 4.1 Resource-allocation graph.

- Given the definition of a resource-allocation graph, it can be shown that, if the graph contains **no cycles**, then **no process in the system is deadlocked**.
- 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. 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.
- For example: Suppose that in figure 4.1 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 (figure 4.2) At this point, two minimal cycles exist in the system:

  P1 →R 1 → P2 → R3→P3 → R2 → P1

  P2 → R3 →P3 → R2 → P2
- 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.
- Now consider the resource-allocation graph in Figure 4.3. In this example, we also have a cycle:

  P1→R1→P3→R2→P1

- 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.

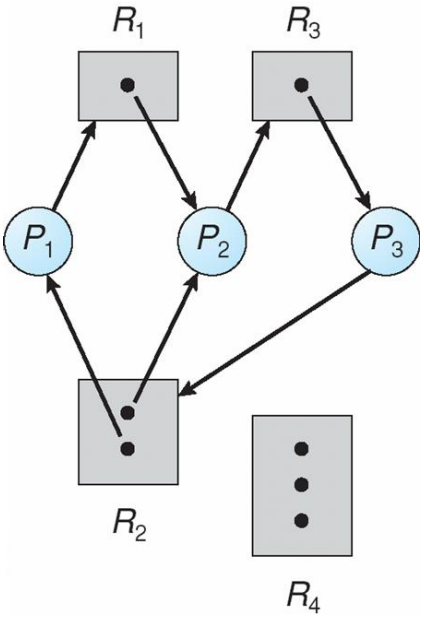



Figure4.2 Resource-allocation graph with a cycle and deadlock.

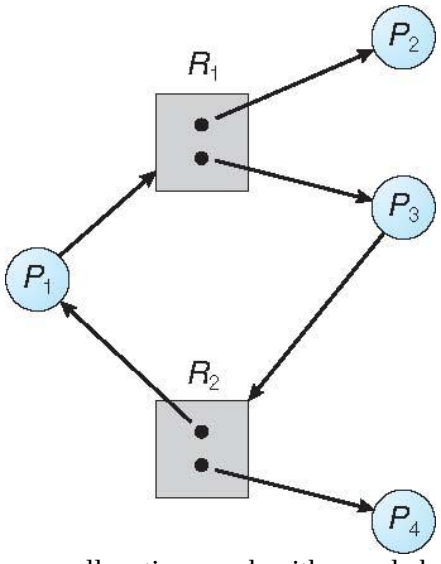



Figure 4.3 Resource-allocation graph with a cycle but no deadlock.

## 4.4 Methods for Handling Deadlocks

We can deal with the deadlock problem in one of **three** ways:

1. We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
2. We can allow the system to enter a deadlocked state, detect it, and recover.
3. We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including UNIX and Windows; it is then up to the application developer to write programs that handle deadlocks.

### 4.4.1 Deadlock Prevention

Deadlock prevention provides a set of methods for ensuring that at least one of the necessary

conditions **cannot hold**.

**i. Mutual Exclusion**

- The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes.
- Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource.
- In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

**ii. Hold and Wait**

- To ensure that the hold-and-wait condition never occurs in the system, we must 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.
- An alternative 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, however, it must release all the resources that it is currently allocated.
- Both these protocols have two main **disadvantages**. **First**, resource utilization may be low, since resources may be allocated but unused for a long period.
- **Second**, starvation is possible.

**iii. No preemption**

- The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated.
- First Method: If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted.
- Second Method: if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we 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.
- This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as printers and tape drives.

**iv. 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, we let R = {R1, R2, ... , Rm} be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function F: R →N, where N is the set of natural numbers.
- A process can initially request any number of instances of a resource type -say, Ri -. After that, the process can request instances of resource type Rjif and only if F(Rj) > F(Ri).

### 4.4.2 Deadlock Avoidance

- 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

#### 4.4.2.1 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. More formally, a system is in a safe state only if there exists a **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.
- A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state.

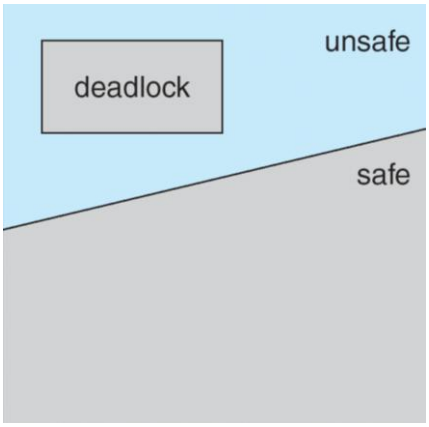



Figure 4.4 Safe, unsafe, and deadlocked state spaces.

- To illustrate, we consider a system with 12 magnetic tape drives and three processes: Po, P1, and P2. Process Po requires ten tape drives, process P1 may need as many as four tape drives, and process P2 may need up to nine tape drives. Suppose that, at time to, process Po is holding five tape drives, process P1 is holding two tape drives, and process P2 is holding two tape drives. (Thus, there are **three free tape drives**.)

| Process | Maximum Need | Allocated | Current Need |
|---|---|---|---|
| P0 | 10 | 5 | 5 |
| P1 | 4 | 2 | 2 |
| P2 | 9 | 2 | 7 |

- At time t0, the system is in a safe state. The sequence <P1, P0, P2> satisfies the safety condition.
- A system can go from a safe state to an unsafe state.
- Suppose that, at time t1, process P2 requests and is allocated one more tape drive. The system is no longer in a safe state.

| Process | Maximum Need | Allocated | Current Need |
|---|---|---|---|
| P0 | 10 | 5 | 5 |
| P1 | 4 | 2 | 2 |
| **P2** | **9** | **3** | **6** |

- Now only 2 tape drives are free.

- At this point, only process P1 can be allocated all its tape drives. When it returns them, the system will have only **four** available tape drives. Since process Po is allocated five tape drives but has a maximum of ten, it may request five more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process P2 may request six additional tape drives and have to wait, resulting in a deadlock. Our mistake was in granting the request from process P2 for one more tape drive.

- Single instance of a resource type
    - Use a resource-allocation graph
- Multiple instances of a resource type
    - Use the banker's algorithm

#### 4.4.2.2 Use a Resource-Allocation Graph

- **Allocation (Assignment) edge**, a directed edge from resource type Rj to Process Pi that is Rj→Pi, signifies that an instance of resource type Rj is allocated to Process Pi.
- **Request edge**, a directed edge from process Pi to resource type Rj that is Pi→Rj, signifies that process Pi requests an instance of resource type Rj.
- We have another kind of edge, **Claim Edge,** A claim edge Pi →Rj indicates that process Pi may request resource Rj at some time in the future. (dotted line)
- When process Pi requests resource R1, the claim edge Pi→Rj is converted to a request edge.
- Similarly, when a resource Rj is released by Pi, the assignment edge Rj→Pi is reconverted to a claim edge Pi →Rj.
- 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. We check for safety by using a cycle-detection algorithm.
- 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.

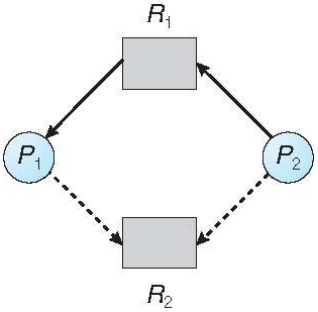



Figure 4.5 An unsafe state in a resource-allocation graph.

### 4.4.2.3 Banker's Algorithm

- The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.
- Following data structures are needed, where n is the number of processes in the system and m is the number of resource types:
    - **Available**. A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type Rj are available.
    - **Max**. An n x m matrix defines the maximum demand of each process. If Max[i] [j] equals 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] equals k, then process Pi is currently allocated k instances of resource type Rj.
- **Need**. An n x m matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process Pi may need k more instances of resource type Rj to complete its task.

Note that **Need[i][j] =Max[i][j]- Allocation [i][j].**

### 1. Safety Algorithm

Used to find whether the system is safe state or not.

1. Let Work and Finish be vectors of length m and n, respectively.

Initialize

Work= Available and finish[i] =false for i= 0, 1, ... ,n - 1.

2. Find an index i such that both
   a. finish[i] = = false
   b. $Need_i$ <= Work

If no such i exists, go to step 4.

3. Work = Work + Allocation
   Finish[i] = true
   Go to step 2.

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

### 2. Resource-Request Algorithm

Used to determine whether the requests can be safely granted.

1. If $Request_i$<= $Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i$<=Available, go to step 3. Otherwise, Pi must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

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

If the resulting resource-allocation state is safe, the transaction is completed, and process P; is allocated its resources. However, if the new state is unsafe, then P; must wait for $Request_i$ ,and the old resource-allocation state is restored.

### An Illustrative Example of Banker's algorithm

To illustrate the use of the banker's algorithm, consider a system with five processes Po through P4 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 T0, the following snapshot of the system has been taken:

| | Allocation | | | Max | | | Available | | | Need (Calculate)<br>$Max_i$ - $Allocation_i$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | 3 | 3 | 2 | 7 | 4 | 3 |
| 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 |

Safety Algorithm,

**Step 1 :** Initialize
Work = (3, 3, 2)
finish[0] = finish[1] = finish[2] = finish[3] = finish[4] = false
**Step 2 : for i=0 Process P0**
finish[0] == false
(7, 4, 3) <≠ (3, 3, 2) i.e $Need_0$ < = Work

**for i=1 Process P1**
finish[1] == false
(1, 2, 2) <= (3, 3, 2) i.e $Need_1$ < = Work
Step 3:
Work = (3, 3, 2) + (2, 0, 0) i.e. Work = Work + $Allocation_1$
Work = (5, 3, 2)
finish[1] = true

**for i=2 Process P2**
finish[2] == false
(6, 0, 0) <≠ (5, 3, 2) i.e $Need_2$ < = Work

**for i=3 Process P3**
finish[3] == false
(0, 1, 1) <= (5, 3, 2) i.e $Need_3$ < = Work
Step 3:
Work = (5, 3, 2) + (2, 1, 1) i.e. Work = Work + $Allocation_3$
Work = (7, 4, 3)
finish[3] = true

**for i=4 Process P4**
finish[4] == false
(4, 3, 1) <= (7, 4, 3) i.e $Need_4$ < = Work
Step 3:
Work = (7, 4, 3) + (0, 0, 2) i.e. Work = Work + $Allocation_4$
Work = (7, 4, 5)
finish[4] = true

repeat the steps for i=0 and i=2
**for i=0 Process P0**
finish[0] == false
(7, 4, 3) <= (7, 4, 5) i.e $Need_0$ < = Work
Step 3:
Work = (7, 4, 5) + (0, 1, 0) i.e. Work = Work + $Allocation_0$
Work = (7, 5, 5)

finish[0] = true

**for i=2 Process P2**

finish[2] == false

(6, 0, 0) <= (7, 5, 5) i.e $Need_2$ < = Work

Step 3:

Work = (7, 5, 5) + (3, 0, 2) i.e. Work = Work + $Allocation_2$

Work = (10, 5, 7)

finish[2] = true

Step 4: finish[i] == true for all i. Hence the system is in safe state.

**Safe Sequence < P1, P3, P4, P0, P2 >**

Suppose now that process P1 requests one additional instance of resource type A and two instances of resource type C, so Request1 = (1,0,2).

Step 1: $Request_1$ <= Need (1, 0, 2) <= (1, 2, 2)

Step 2: $Request_1$ <= Available (1, 0, 2) <= (3, 3, 2)

Step 3: Now,

Available = Available – $Request_1$

Available = (3, 3, 2) – (1, 0, 2) = (2, 3, 0)

$Allocation_1$ = $Allocation_1$ + $Request_1$

$Allocation_1$ = (2, 0, 0) + (1, 0, 2) = (3, 0, 2)

$Need_1$ = $Need_1$ - $Request_1$

Need1 = (1, 2, 2) – (1, 0, 2) = (0, 2, 0)

| | Allocation | | | Max | | | Available | | | Need (Calculate) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 | **2** | **3** | **0** | 7 | 4 | 3 |
| **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 |

Now Apply Safety algorithm as above,

**Step 1 :** Initialize

Work = (2, 3, 0)

finish[0] = finish[1] = finish[2] = finish[3] = finish[4] = false

**Step 2 : for i=0 Process P0**

finish[0] == false

(7, 4, 3) <≠ (2, 3, 0) i.e $Need_0$ < = Work

**for i=1 Process P1**

finish[1] == false

(0, 2, 0) <= (2, 3, 0) i.e $Need_1$ < = Work

Step 3:

Work = (2, 3, 0) + (3, 0, 2) i.e. Work = Work + $Allocation_1$

Work = (5, 3, 2)
finish[1] = true

**for i=2 Process P2**
finish[2] == false
(6, 0, 0) <≠ (5, 3, 2) i.e $Need_2$ < = Work

**for i=3 Process P3**
finish[3] == false
(0, 1, 1) <= (5, 3, 2) i.e $Need_3$ < = Work

Step 3:
Work = (5, 3, 2) + (2, 1, 1) i.e. Work = Work + $Allocation_3$
Work = (7, 4, 3)
finish[3] = true

**for i=4 Process P4**
finish[4] == false
(4, 3, 1) <= (7, 4, 3) i.e $Need_4$ < = Work

Step 3:
Work = (7, 4, 3) + (0, 0, 2) i.e. Work = Work + $Allocation_4$
Work = (7, 4, 5)
finish[4] = true

repeat the steps for i=0 and i=2

**for i=0 Process P0**
finish[0] == false
(7, 4, 3) <= (7, 4, 5) i.e $Need_0$ < = Work

Step 3:
Work = (7, 4, 5) + (0, 1, 0) i.e. Work = Work + $Allocation_0$
Work = (7, 5, 5)
finish[0] = true

**for i=2 Process P2**
finish[2] == false
(6, 0, 0) <= (7, 5, 5) i.e $Need_2$ < = Work

Step 3:
Work = (7, 5, 5) + (3, 0, 2) i.e. Work = Work + $Allocation_2$
Work = (10, 5, 7)
finish[2] = true

Step 4: finish[i] == true for all i. Hence the system is in safe state.
**Safe Sequence < P1, P3, P4, P0, P2 >**

We find that system is in safe state as we have safe state, <P1, P3, P4, Po, P2>Hence, we can **immediately grant** the request of process P1.

A request for (0, 2, 0) by P0.

| | Allocation | | | Max | | | Need | | | Need (Calculate) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| **P0** | **0** | **3** | **0** | 7 | 5 | 3 | **2** | **1** | **0** | **7** | **2** | **3** |
| 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 |

This request cannot be granted, even though the resources are available, since the resulting state is unsafe. (Use safety algorithm to prove unsafe state) Hence, the old resource-allocation state is restored.

## 4.4.3 Deadlock Detection and Recovery

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:

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

### 4.4.3.1 Deadlock Detection

#### Single Instance of Each Resource Type

- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for graph**. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- More precisely, an edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pjto release a resource that Pi needs. An edge Pi →Pj exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi→Rq and Rq→Pj for some resourceRq.
- As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle.
- Example: Figure 4.6.

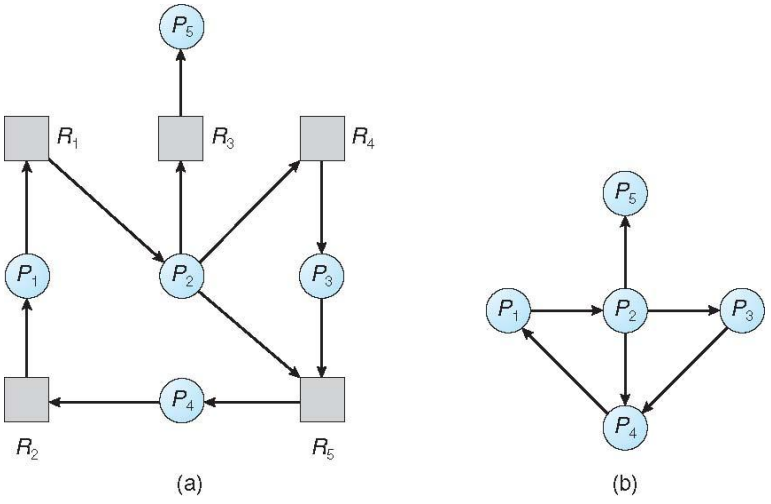



Figure 4.6 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

### Several Instances of a Resource Type

- The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock detection algorithm that is applicable to such a system.
- **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 Pi is requesting k more instances of resource type Rj.

**Deadlock Detection Algorithm:**

1. Let Work and Finish be vectors of length m and n, respectively. Initialize
   Work= Available. Fori= 0, 1, ... , n-1, if Allocation != 0, then Finish[i] =false; otherwise, Finish[i] = true.
2. Find an index isuch that both,
   Finish[i] ==false
   $Request_i$<=Work
3. If no such iexists, go to step 4.Work= Work+ Allocation;
   Finish[i] = true
   Go to step 2.
4. If Finish[i] ==false for some i, 0 < = i<n, then the system is in a deadlockedstate. Moreover, if Finish[i] ==false, then process P; is deadlocked.

**Illustrative example:**

| | Allocation | | | Need | | | Request | | |
|---|---|---|---|---|---|---|---|---|---|
| | 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 | 2 | | | | 0 | 0 | 0 |
| P3 | 2 | 1 | 1 | | | | 1 | 0 | 0 |
| P4 | 0 | 0 | 2 | | | | 0 | 0 | 2 |

Deadlock detection algorithm,

| Process | P0 | P2 | P3 | P1 | P4 |
|---|---|---|---|---|---|
| Available | 0, 0, 0 | 0, 1, 0 | 3, 1, 2 | 5, 2, 3 | 7, 2, 3 |
| Allocation | 0, 1, 0 | 3, 0, 2 | 2, 1, 1 | 2, 0, 0 | 0, 0, 2 |
| New Available | 0, 1, 0 | 3, 1, 2 | 5, 2, 3 | 7, 2, 3 | 7, 2, 5 |
| Finish[i] | True | True | True | True | True |

Hence no deadlock.

Suppose now that process P2 makes one additional request for an instance of type C.

| | Allocation | | | Need | | | Request | | |
|---|---|---|---|---|---|---|---|---|---|
| | 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 | 2 | | | | 0 | 0 | 1 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| P3 | 2 | 1 | 1 | | | | 1 | 0 | 0 |
| P4 | 0 | 0 | 2 | | | | 0 | 0 | 2 |

We claim that the system is now deadlocked. As Finish[i] != true for all i

## 4.5 Recovery from Deadlock

There are **two** options for breaking a deadlock.

### 4.5.1 Process Termination

To eliminate deadlocks by aborting a process, we 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.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

Many factors may affect which process is chosen for termination, 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? (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete?
5. How many processes will need to be terminated?

### 4.5.2 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.
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

## 5.1 Background :

### 5.1.1 Basic Hardware:

- Main memory and the registers built into the processor itself are the only storage that the CPU can access directly.
- There are machine instructions that take memory addresses as arguments, but none that take disk addresses.
- Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices.
- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock.
- Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.

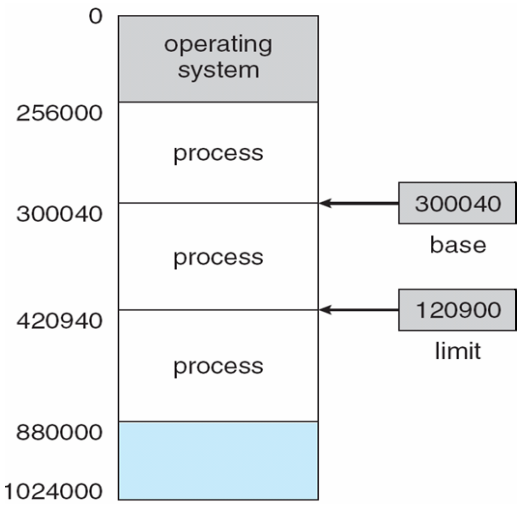



Fig. 8.1 A base and limit register define a logical address space

- We can provide the protection by using two registers, usually a base and a limit, as illustrated in Figure 8.1.
- 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).
- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
- Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error (Figure 8.2). This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

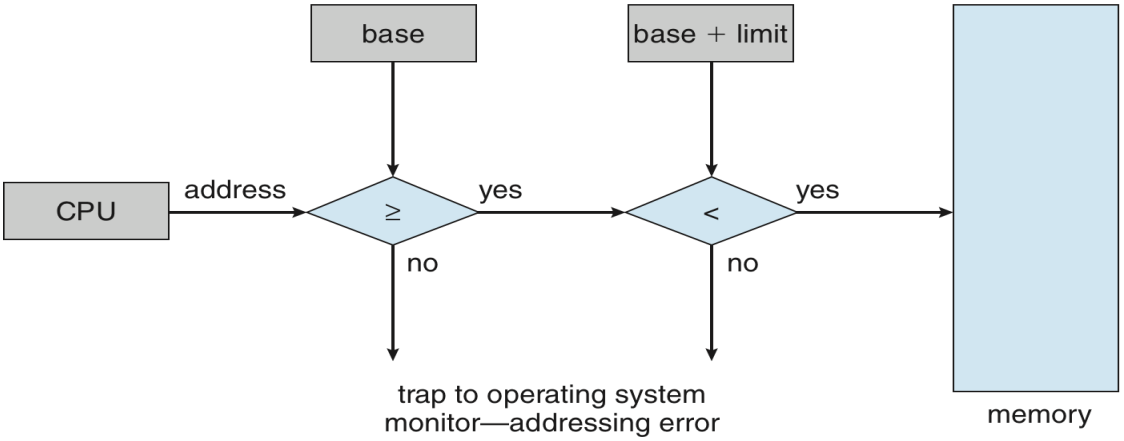



Fig. 8.2 Hardware address protection with base and limit registers

### 5.1.2 Address Binding

- Programs are stored on the secondary storage disks as binary executable file.
- To be executed, the program must be brought into memory and placed within a process.
- Depending on the memory management in use, the process may be moved between disk and memory during its execution.
- The normal procedure is to select one of the processes in the input queue and to load that process into memory.
- As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available.
- In most cases, a user program will go through several steps—some of which maybe optional—before being executed (Figure 8.3).
- Classically, the binding of instructions and data to memory addresses can be done at any step along the way:
    1. **Compile time.** If you know at compile time where the process will reside in memory, then absolute code can be generated. For example, if you know that a user process will reside starting at location *R*, then the generated compiler code will start at that location and extend up from there.
    2. **Load time.** If it is not known at compile time where the process will reside in memory, then the compiler must generate reloadable code. In this case, final binding is delayed until load time.
    3. **Execution time.** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Most general-purpose operating systems use this method.

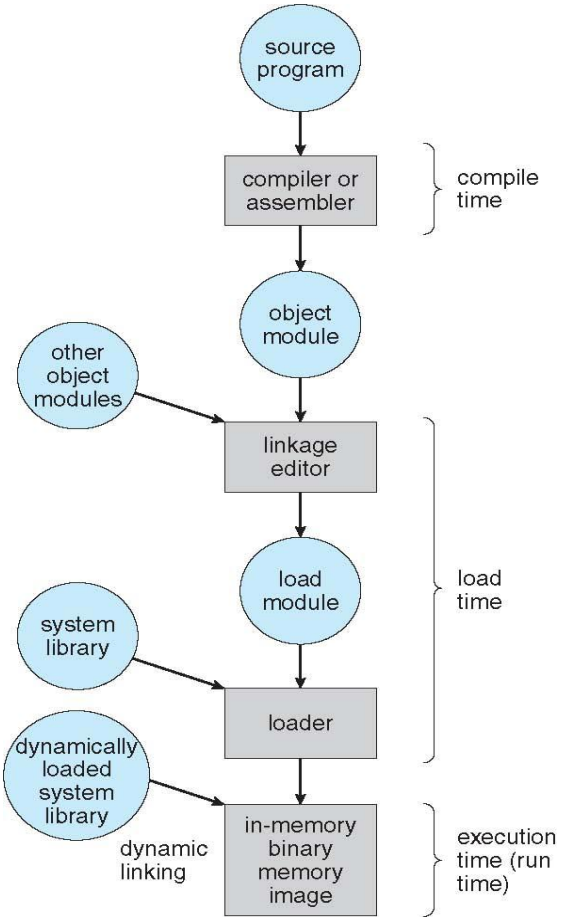



Fig. 8.3 Multistep Processing of user Program

### 5.1.3 Logical versus Physical Address Space

- An address generated by the CPU is commonly referred to as a **logical address** whereas an address seen by the memory unit-that is, the one loaded into the **memory address register** of the memory-is commonly referred to as a **physical address**.
- The compile-time address-binding methods generate identical logical and physical addresses.
- However, the execution-time and load-time address binding schemes results in differing logical and physical addresses.
- The set of all logical addresses generated by a program is a logical address space; the set of all physical addresses corresponding to these logical addresses is a physical address space.

**Dynamic re-location using a re-location registers**

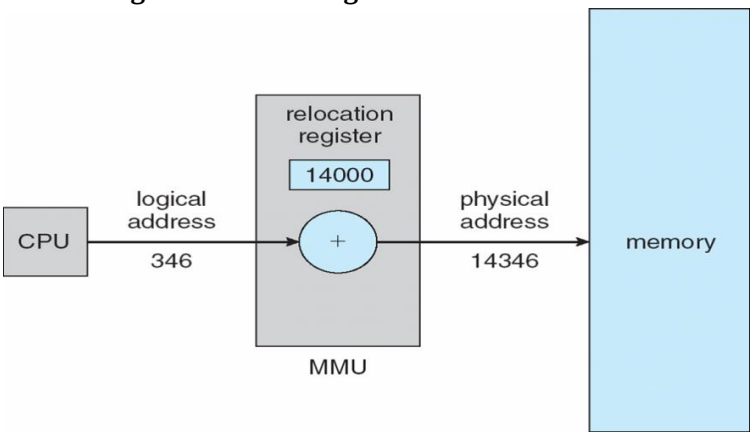



Figure 5.1 Dynamic relocation using a relocation register.

- The above figure shows that dynamic re-location which implies mapping from virtual (logical) addresses space to physical address space and is performed by the hardware at run time.
- The value in the relocation register is *added* to every address generated by a user process at the time the address is sent to memory
- Re-location is performed by the hardware and is invisible to the user.
- Dynamic relocation makes it possible to move a partially executed process from one area of memory to another without affecting.

### 5.1.4 Dynamic Loading

- For a process to be executed it should be loaded in to the physical memory.
- The size of the process is limited to the size of the physical memory.
- Dynamic loading is used to obtain better memory utilization.
- In dynamic loading the routine or procedure will not be loaded until it is called.
- Whenever a routine is called, the calling routine first checks whether the called routine is already loaded or not. If it is not loaded it cause the loader to load the desired program in to the memory and updates the programs address table to indicate the change and control is passed to newly called routine.
- Advantage:-
    - Gives better memory utilization.
    - Unused routine is never loaded. Do not need special operating system support.
    - This method is useful when large amount of codes are needed to handle in frequently occurring cases.

### 5.1.5 Dynamic Linking and Shared Libraries

- Static linking – system libraries and program code combined by the loader into the binary program image.
- Dynamic linking –linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine.
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
    - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as shared libraries
- Consider applicability to patching system libraries
    - Versioning may be needed

## 5.2 Swapping

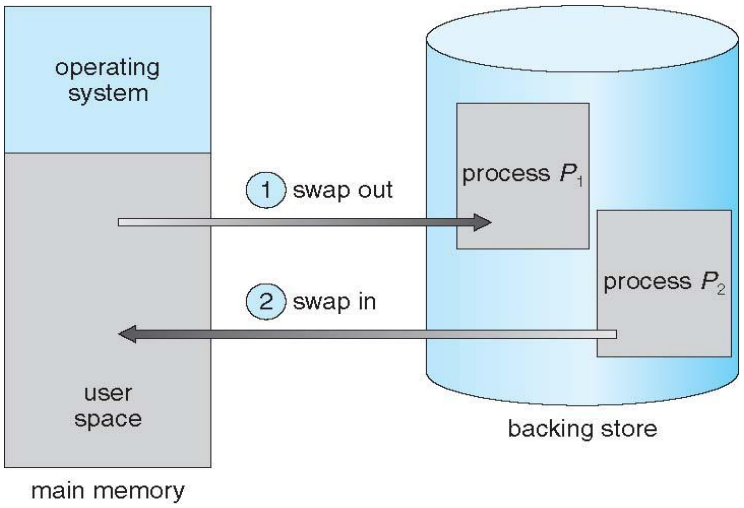



Figure 5.2 Swapping of two processes using a disk as a backing store.

- A process must be in memory to be executed.
- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
    - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk
- Does the swapped out process need to swap back in to same physical addresses?
    - Depends on address binding method
    - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
    - Swapping normally disabled
    - Started if more than threshold amount of memory allocated
    - Disabled again once memory demand reduced below threshold
- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high

## 5.3 Contiguous Memory Allocation

### 5.3.1. Memory Mapping and Protection

- Main memory is usually divided into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
- **Base register** contains value of smallest physical address
- **Limit register** contains range of logical addresses – each logical address must be less than the limit register
- MMU maps logical address *dynamically* by adding the value in the relocation register.
- This mapped address is sent *to* memory (as shown in figure 5.3)

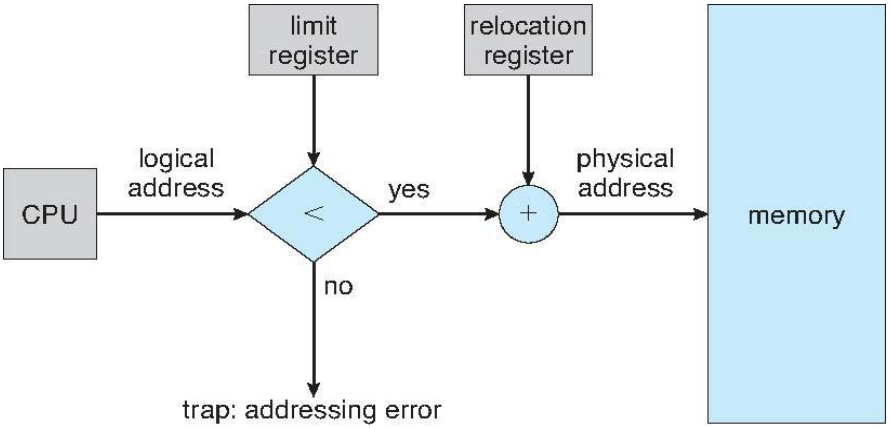
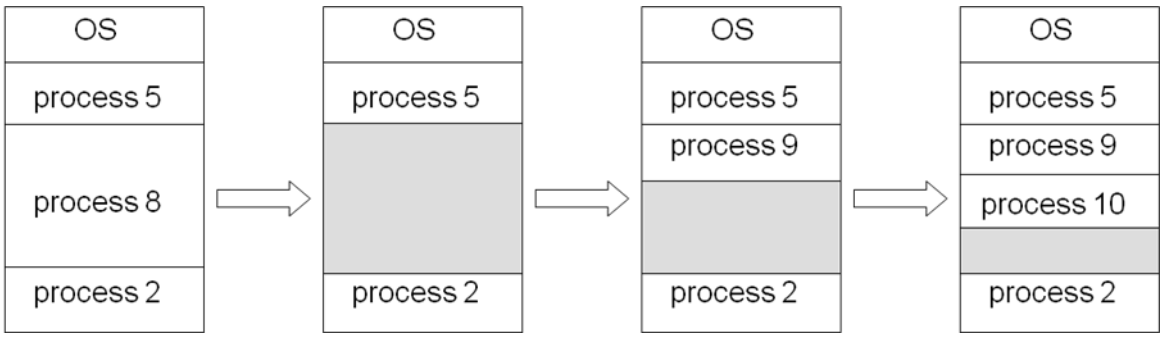



Figure 5.3 Hardware support for relocation and limit registers

### 5.3.2. Memory Allocation

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions **(MFT-Multiprogramming with fixed number of partitions).**
- Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions.
- In this multiple partition method when a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.
- Generalization of the fixed-partition scheme called **MVT (Multiprogramming with variable number of partitions)**.
- In the variable partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user processes and is considered one large block of available memory a hole.
- Eventually memory contains a set of holes of various sizes.




- As processes enter the system, they are put into an input queue.
- The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.
- When a process is allocated space, it is loaded into memory, and it can then compete for CPU time.
- When a process terminates, it releases its memory which the operating system may then fill with another process from the input queue.
- When the memory has a set of holes the first fit, best fit and worst fit strategies are used to select the free hole for the new process.
    - **First fit.** Allocate the *first* hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
    - **Best fit.** Allocate the *smallest* hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
    - **Worst fit.** Allocate the *largest* hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Eg: Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?

First-fit:
212K is put in 500K partition
417K is put in 600K partition
112K is put in 288K partition (new partition 288K = 500K - 212K)
426K must wait

Best-fit:
212K is put in 300K partition
417K is put in 500K partition
112K is put in 200K partition
426K is put in 600K partition

Worst-fit:
212K is put in 600K partition
417K is put in 500K partition
112K is put in 388K partition

426K must wait

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

### 5.3.3. Fragmentation

- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
- **External fragmentation** exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous; storage is fragmented into a large number of small holes.
- This fragmentation problem can be severe.
- Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem.
- Statistical analysis of first fit, for instance, reveals that, even with some optimization, given *N* allocated blocks, another 0.5 *N* blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the 50-percent rule.
- Memory fragmentation can be **internal** as well as **external**.
- Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes.
- The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size.
- With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation** that is unused memory that is internal to a partition.
- One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible.
- Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available.

## 5.4 Non-Contiguous Memory Allocation

- Two techniques of non-contiguous memory allocation are paging and segmentation.

### 5.4.1. Paging

#### 5.4.1.1 Basic Method

- Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous.
- Paging avoids external fragmentation and the need for compaction.
- It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store.
- 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 their source. The hardware support for paging is illustrated in Figure 5.4

- Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d).** The page number is used as an index into a **page table.**
- The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

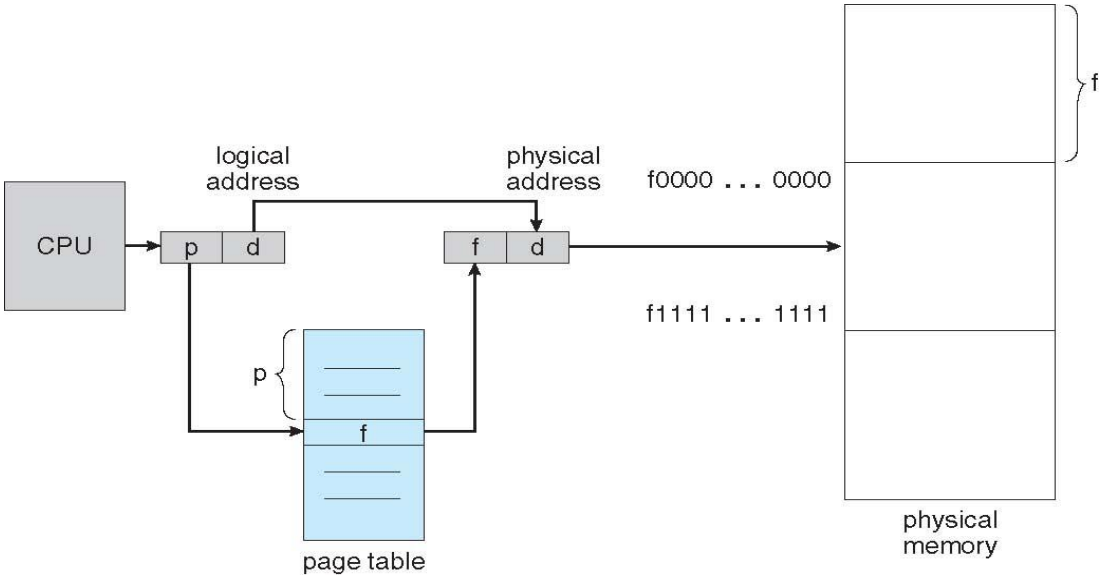
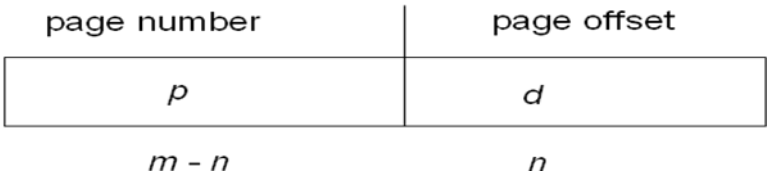



Figure 5.4 Paging Hardware

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




**Example:**

- Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages).
- Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 (= (5 x 4) + 0).
- Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5x4) + 3).
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6.
- Logical address 4 maps to physical address 24 (= (6x4) + 0).
- Logical address 13 maps to physical address 9.

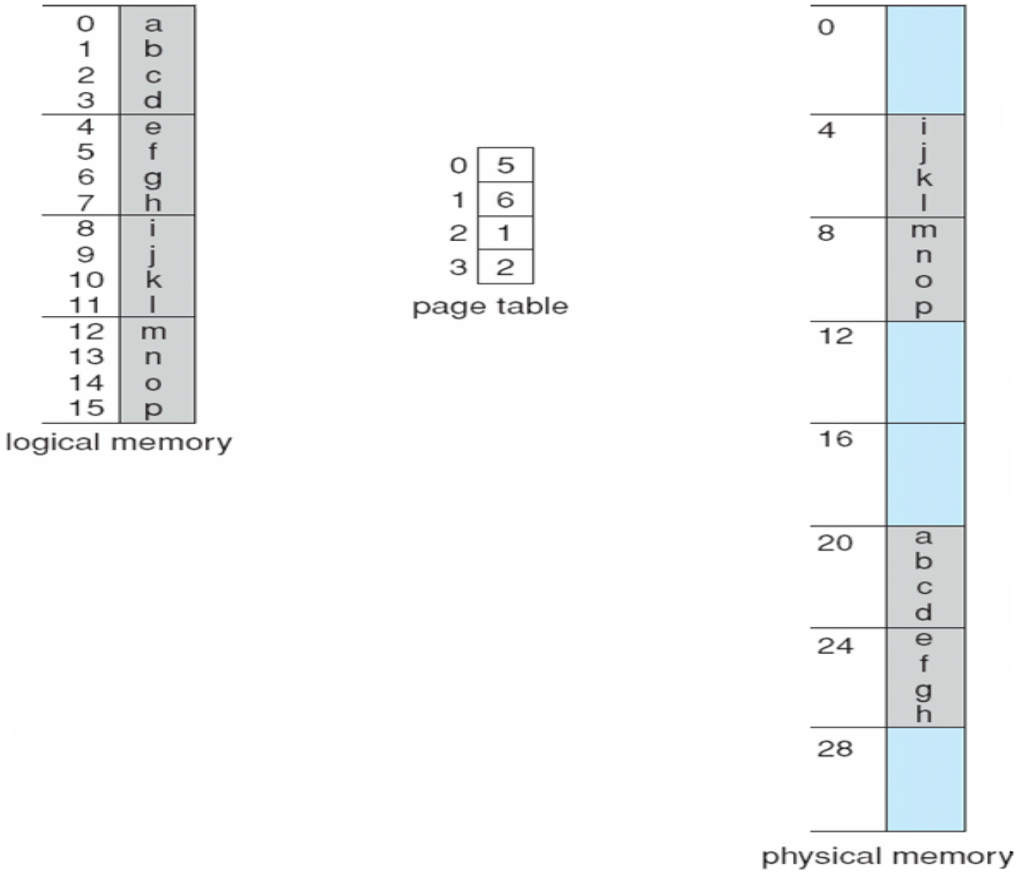



Figure 5.5 Paging Example

**5.4.1.2 Translation Look aside Buffer (TLB)**

- Each operating system has its own methods for storing page tables. Most allocate a page table for each process.
- A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block.
- The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary computers, however, allow the page table to be very large (for example, 1 million entries).
- For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in **main memory**.
- The problem with this approach is the time required to access a user memory location. If we want to access location i, we must first index into the page table.
- This task requires a memory access. It provides us with the frame number, which is combined with the page offset to produce the actual physical address.
- With this scheme, ***two* memory accesses** are needed to access a byte (one for the page-table entry, one for the byte).
- The standard solution to this problem is to use a special, small, fast lookup hardware cache, called **a translation look-aside buffer (TLB)**.
- The TLB is used with page tables in the following way.
- The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, page number is first searched in TLB.
- If the page number is found, its frame number is immediately available and is used to access memory.

- The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used.
- If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made.

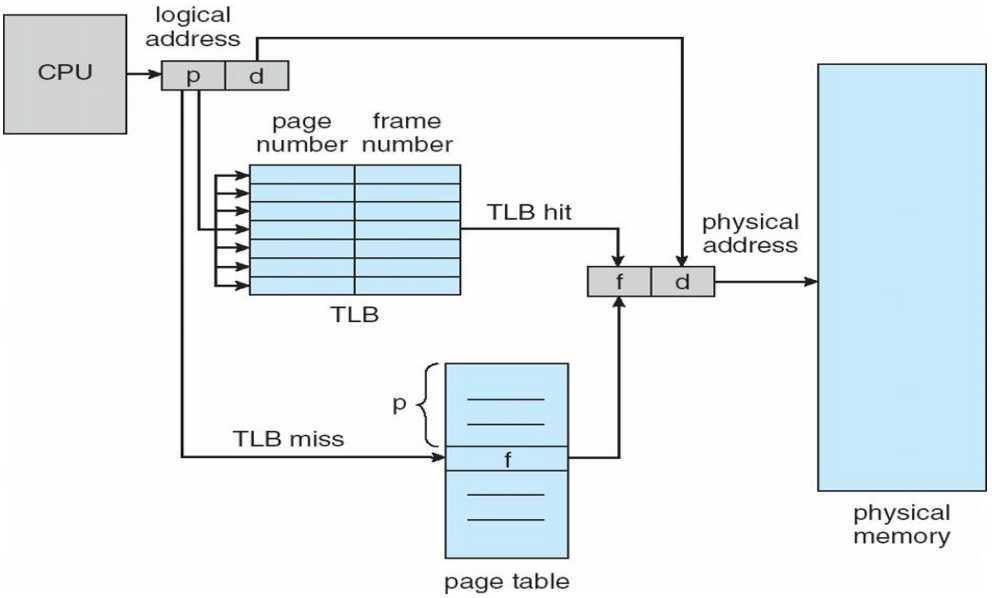



Figure 5.6 Paging hardware with TLB.

- The percentage of times that a particular page number is found in the TLB is called the **hit ratio.**
- An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time.
- If it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory, then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB.
- If we fail to find the page number in the TLB (20 nanoseconds), 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 220 nanoseconds.
- To find the **effective memory-access time,** we weight each case by its probability:

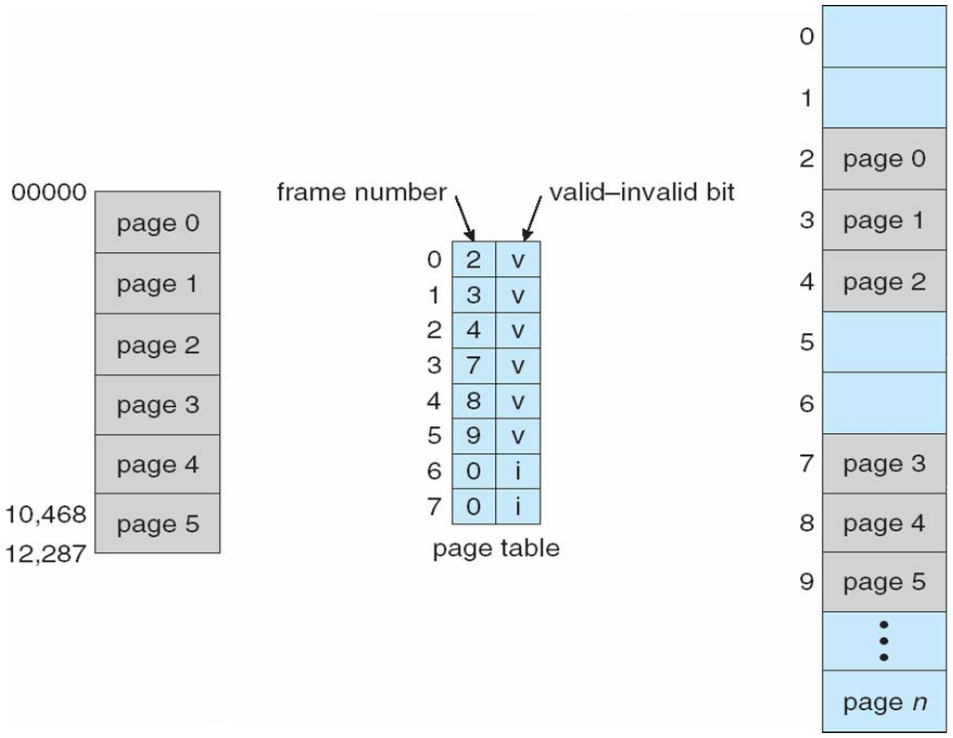
  Effective access time = 0.80 * 120 + 0.20 * 220 = 140 nanoseconds.
- In this example, we suffer a 40-percent slowdown in memory-access time (from 100 to 140 nanoseconds).
- For a 98-percent hit ratio,

  we have effective access time = 0.98 * 120 + 0.02 * 220 = 122 nanoseconds.

**8.4.1.3 Protection**

- Memory protection in a paged environment is accomplished by protection bits associated with each frame.
- Normally, these bits are kept in the page table.
- One bit can define a page to be read-write or read-only.
- Every reference to memory goes through the page table to find the correct frame number.
- At the same time that the physical address is being computed, the protection bits can be 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 operating system.
- One additional bit is generally attached to each entry in the page table: a **valid-invalid** bit.

- When this bit is set to "valid," the associated page is in the process's logical address space and is thus a legal (or valid) page.
- When the bit is set to "invalid,"' the page is not in the process's logical address space.




**5.4.1.4 Shared Pages**

- One of the advantages of paging is the possibility of *sharing* common code.
- This consideration is particularly important in a time-sharing environment.
- Example: Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users.
- If the code is **reentrant code** (or **pure code),** however, it can be shared, as shown in below Figure. Here we see a three-page editor—each page 50 KB in size being shared among three processes. Each process has its own data page.
- Reentrant code is non-self-modifying code; it never changes during execution. Thus, two or more processes can execute the same code at the same time.
- Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different.
- Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB - a significant savings.

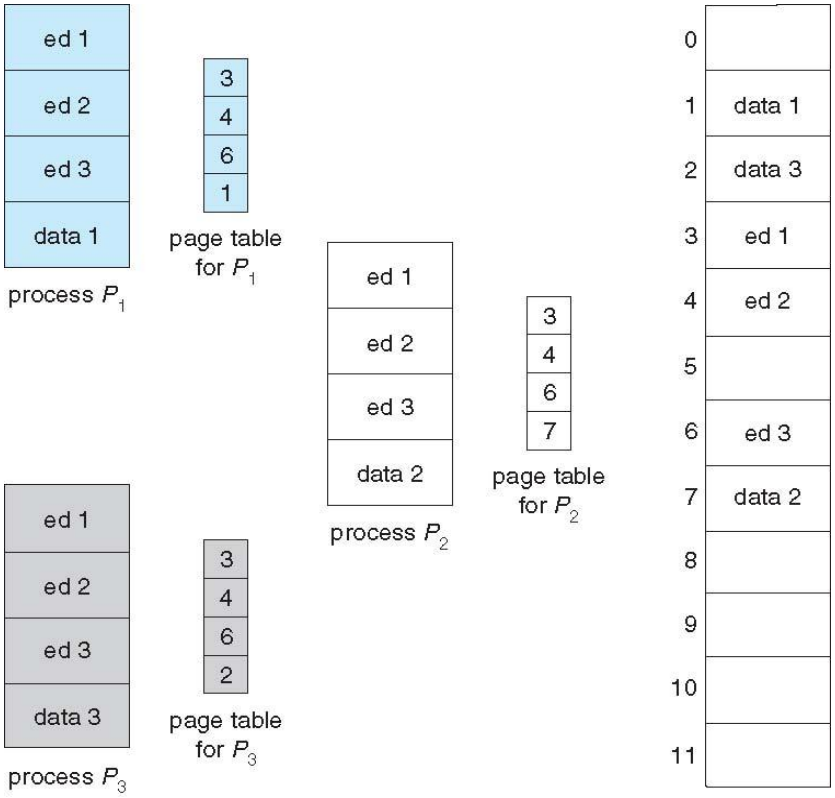



Figure 5.6 Sharing of Code (Share Pages)

### 5.4.2. Segmentation

- Consider how you think of a program when you are writing it. You think of it as a main program with a set of methods, procedures, or functions.
- It may also include various data structures: objects, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. You talk about "the stack," "the math library," "the main program," without caring what addresses in memory these elements occupy.

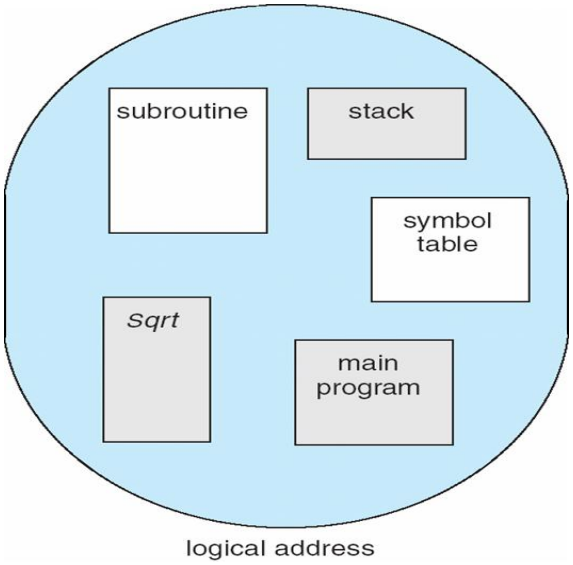



Figure 5.7 User view of Program

- Segmentation is a memory-management scheme that supports this user view of memory.
- A logical address space is a collection of segments.
- Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment.

- The user therefore specifies each address by two quantities: a segment name and an offset.
- (Contrast this scheme with the paging scheme, in which the user specifies only a single address, which is partitioned by the hardware into a page number and an offset, all invisible to the programmer.)
- For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name.
- Thus, a logical address consists of a ***two tuple:*** **< segment-number, offset >.**
- Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes.
- This is done with the help of segment table. Each entry in the segment table has a ***segment base*** **and a** ***segment limit.***
- The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.
- The use of a segment table is illustrated in Figure 5.8. A logical address consists of two parts: a segment number, *s,* and an offset into that segment, *d.*
- The segment number is used as an index to the segment table.
- The offset *d* of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system. When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.
- As an example, consider the situation shown in Figure 5.9.

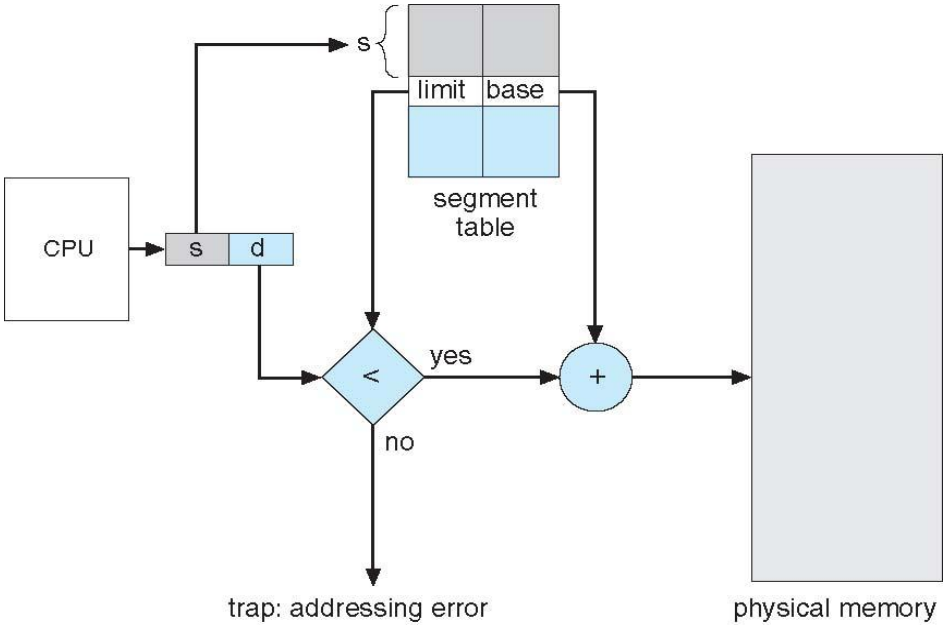



Figure 5.8 Segmentation Hardware

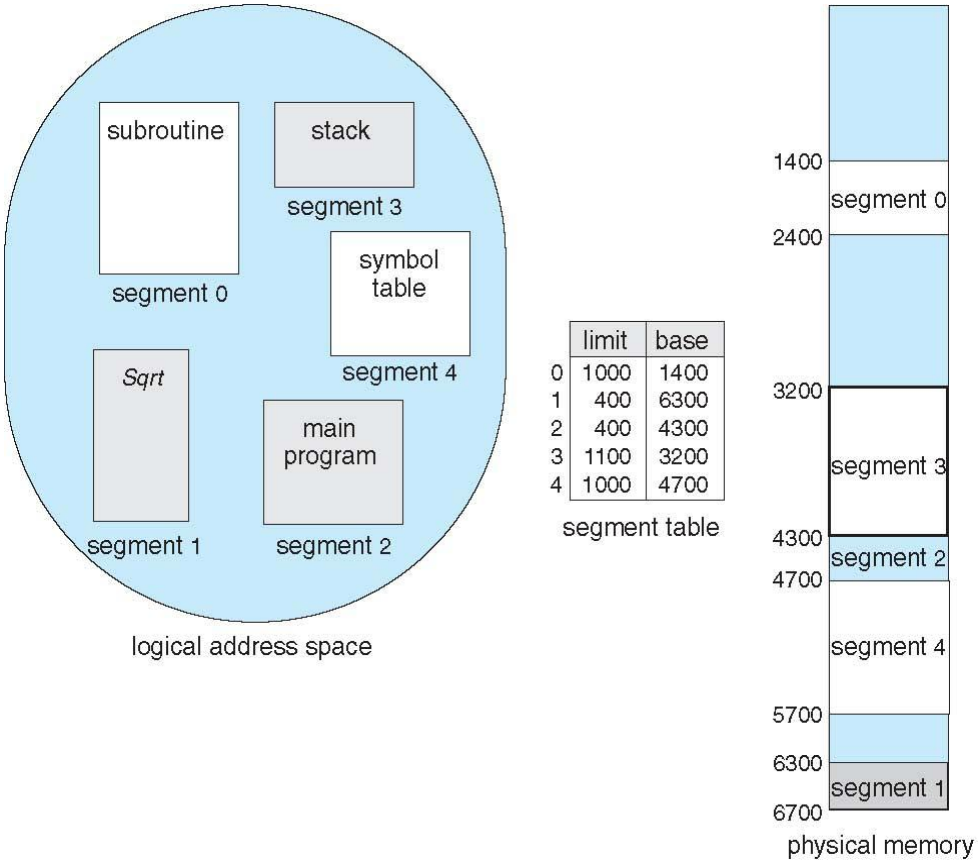



Figure 5.9 Example of segmentation

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

## 5.5 Structure of Page Table

- Memory structures for paging can get huge using straight-forward methods
    - Consider a 32-bit logical address space as on modern computers
    - Page size of 4 KB ($2^{12}$)
    - Page table would have 1 million entries ($2^{32}$ / $2^{12)}$
    - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
        - That amount of memory used to cost a lot
        - Don't want to allocate that contiguously in main memory
- Three methods of representing Page Table
    1. Hierarchical Paging
    2. Hashed Page Tables
    3. Inverted Page Tables

### 5.5.1 Hierarchical Paging

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table

- We then page the page table

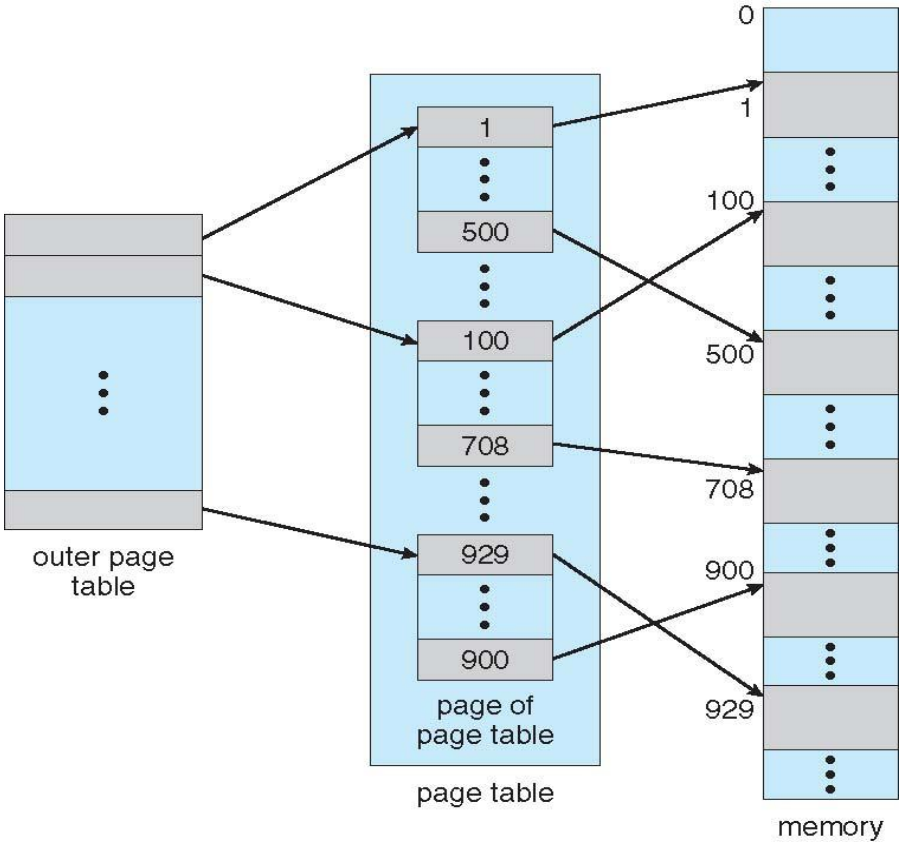
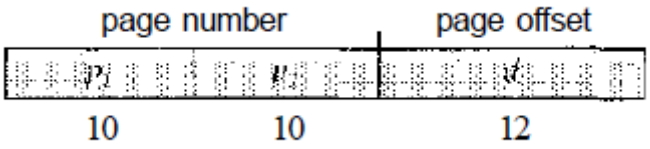



Fig. 8.14 A two-level page table scheme

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:




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

- Known as **forward-mapped page table**

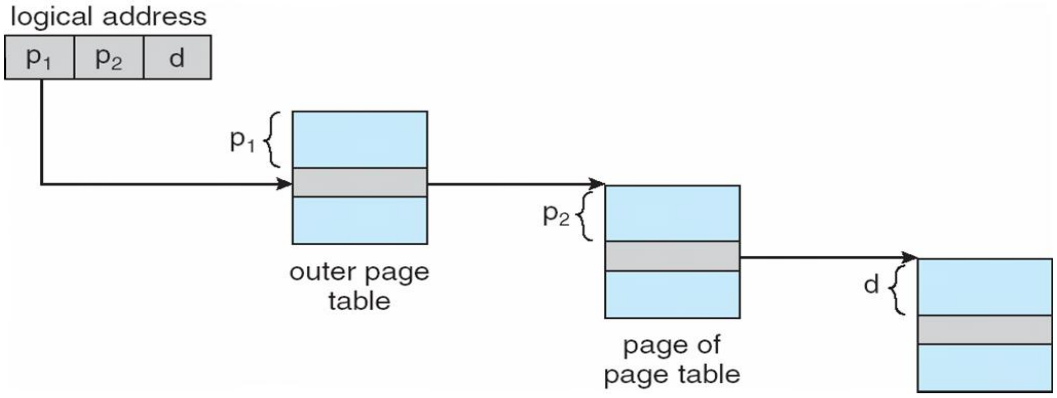
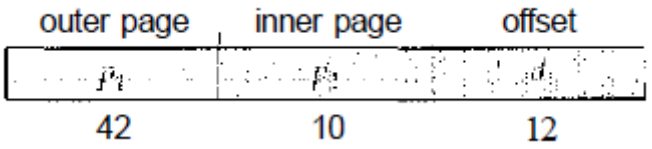



Fig. Address Translation for two-level 32bit paging architecture

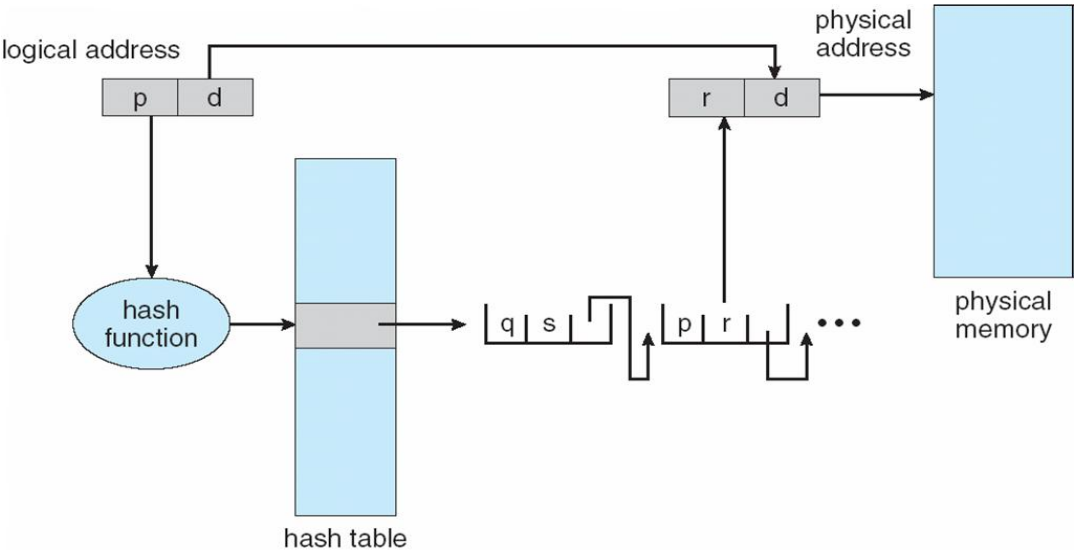
- Even two-level paging scheme not sufficient
- If page size is 4 KB ($2^{12}$)
    - Then page table has $2^{52}$ entries
    - If two level scheme, inner page tables could be $2^{10}$ 4-byte entries
    - Address would look like




    - Outer page table has $2^{42}$ entries or $2^{44}$ bytes
    - One solution is to add a $2^{nd}$ outer page table
    - But in the following example the $2^{nd}$ outer page table is still $2^{34}$ bytes in size
        - And possibly 4 memory access to get to one physical memory location

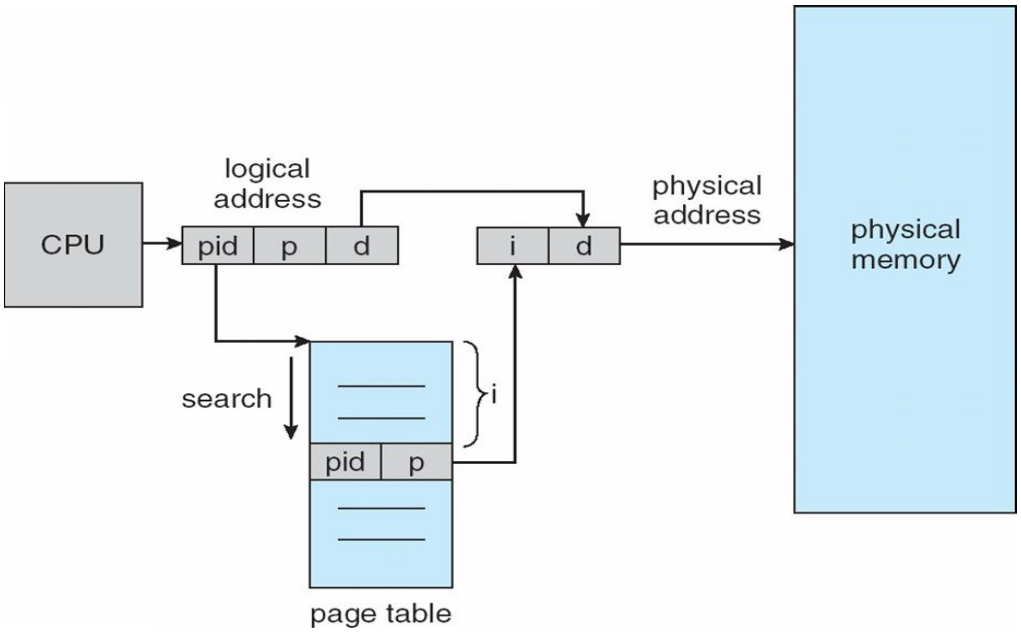
**5.5.2 Hashed Page Table**

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
    - This page table contains a chain of elements hashing to the same location
    - Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- The algorithm works as follows:
    - The virtual page number in the virtual address is hashed into the hash table.
    - The virtual page number is compared with field 1 in the first element in the linked list.
    - If there is a match, the corresponding page frame (field 2) is used to form the desired physical address.
    - If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.
    - This scheme is shown in Figure 8.16.




**5.5.3 Inverted Page Table**

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
    - TLB can accelerate access
- But how to implement shared memory?
    - One mapping of a virtual address to the shared physical address



# Questions from recent VTU papers on Deadlocks

1. What is a deadlock? Explain the necessary conditions for its occurrence. (June11)
2. Explain how Resource-Allocation graphs are used to describe deadlocks. (June10)
3. Write a note on deadlock prevention.
4. What are the methods available for handling deadlocks? Explain Banker's algorithm. (June10 / Dec 13)
5. Deadlock occurs if cycle exists. Justify your answers. (Dec 13)
6. For the following information find the safe sequence using bakers algorithm, the number of resources for R1, R2, R3 are 7, 7, 10 respectively. (Dec 13)

| Process | Allocated resources | | | Maximum Requirements | | |
|---|---|---|---|---|---|---|
| | R1 | R2 | R3 | R1 | R2 | R3 |
| P1 | 2 | 2 | 3 | 3 | 6 | 8 |
| P2 | 2 | 0 | 3 | 4 | 3 | 3 |
| P3 | 1 | 2 | 4 | 3 | 3 | 4 |

7. "A safe state is not a deadlock state but a deadlock state is an unsafe state". Explain. (June10)
8. List any four examples of deadlock that are related to computer systems. (Dec 11)
9. Discuss the various approaches used for deadlock recovery. (June09)
10. Consider the following snapshot of a system. (June 11)

| Process | Allocation | | | | Max Needs | | | | Available | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | A | B | C | D | A | B | C | D |
| P0 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 2 | 1 | 5 | 2 | 0 |
| P1 | 1 | 0 | 0 | 0 | 1 | 7 | 5 | 0 | | | | |
| P2 | 1 | 3 | 5 | 4 | 2 | 3 | 5 | 6 | | | | |
| P3 | 0 | 6 | 3 | 2 | 0 | 6 | 5 | 2 | | | | |
| P4 | 0 | 0 | 1 | 4 | 0 | 6 | 5 | 6 | | | | |

Answer the following question using Banker's algorithms:

a) What is the concept of matrix need?
b) Is the system in a safe state? If yes, what is the safe sequence they have followed?

11. Consider the following snapshot of the system, (Dec 12)

| Process | Allocated resources | | | Maximum Requirements | | | Total Resources | | |
|---|---|---|---|---|---|---|---|---|---|
| | R1 | R2 | R3 | R1 | R2 | R3 | R1 | R2 | R3 |
| P1 | 2 | 2 | 3 | 3 | 6 | 8 | 7 | 7 | 10 |
| P2 | 2 | 0 | 3 | 4 | 3 | 3 | | | |
| P3 | 1 | 2 | 4 | 3 | 3 | 4 | | | |

a) What is content of need matrix?
b) Is the system in safe state?
c) If the following requests are made, can they be granted /satisfied immediately in the current state, P1 request (1, 1, 0), P3 request for (0, 1, 0) immediately.

12. Give 3 processes A, B, C, 3 resources X, Y, Z and the following events,

i) A requests X, ii) A requests Y, iii) B requests Y, iv) B request Z, v) C requests Z, vi) C requests X and vii) C requests Y.

Assume the resource is always given to requesting process if available. Draw the resource allocation graph (RAG) for the sequence (2, 6, 3, 5, 1, 4 and 7). Also mention is deadlock occurs. If so, how to recover from the deadlock. (Dec 12)

## Important Questions from previous VTU Papers on Memory Management

1. Discuss the issues that are pertinent to the various techniques for managing memory.
2. Write a note on dynamic Loading and Linking.
3. What is swapping? Does this increase OS overhead? Justify your answer.
4. Write a note on Contiguous memory allocation.
5. What is internal & external fragmentation?
6. Bring out differences between internal & external fragmentation? How are they overcome?
7. Explain the buddy-system, used for managing free memory assigned to kernel process.
8. What is paging? Explain basic hardware of paging.
9. Why TLB is important. In simple paging what information is stored in TLB.
10. Write a note on shared pages.
11. Describe the Segmentation technique.