Synchronization Primitives

Semaphore

Software-based synchronization mechanism/tool (originally proposed by Dijkstra)

It avoids busy waiting i.e. wasting of CPU time

A semaphore S is an integer variable

Two indivisible operations can modify **S**:

P(S) & V(S) or acquire() & **release() or wait() & signal()**

Indivisible testing & modification of value

```
acquire() {
    while (value <= 0)
       ; // no-op
    value--;
}

release() {
    value++;
}
```

© CourseSmart

WAIT (S):
while (S <= 0);
S = S - 1;

SIGNAL (S):
S = S + 1;

FORMAT:

```
wait ( mutex );                    <-- Mutual exclusion: mutex init to 1.
 CRITICAL SECTION
signal( mutex );
REMAINDER SECTION
```

Instead of loop on busy, suspend can be used:

Block on semaphore == False,

Wakeup on signal (semaphore becomes True),

There may be numerous processes waiting for the semaphore, so keep a list of blocked processes,

Wakeup one of the blocked processes upon getting a signal (choice of who depends on strategy).

To PREVENT looping, we redefine the semaphore structure as:

Semaphore *S* – system object

With each semaphore there is an associated waiting queue.

Each entry in the waiting queue has two data items (object properties):

value (of type integer)

pointer to next record in the queue

P(S) and V(S) operations

P(S) : If S ≥ 1 then S :=S-1

else block the process on the semaphore queue;

V(S) : If some processes are blocked on the semaphore S

then unblock a process

else S = S+1;

```
typedef struct {
    int                 value;
    struct process  *list;      /*  linked list of  process id waiting on  S  */
} SEMAPHORE
```

```
SEMAPHORE s;
wait(s) {
    s.value  =  s.value  -  1;
    if (  s.value  <  0 ) {
            add this process to s.L;
            block;
    }
}
```

```
SEMAPHORE s;
signal(s)  {
    s.value  =  s.value  +  1;
    if (  s.value  <=  0 )  {
            remove a process P from s.L;
            wakeup(P);
    }
}
```

Counting semaphore (General semaphore)

- An integer value used for signalling among processes and the integer value can range over an unrestricted domain
- Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment.
- The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process.
- Spin-lock is a general (counting) semaphore using busy waiting instead of blocking
- Blocking and switching between threads and/or processes may be much more time demanding than the time waste caused by short-time busy waiting

Binary Semaphore

- A semaphore that takes on only the values 0 and 1 on which processes can perform two indivisible operations i.e. changing the integer value from 0 to 1 or 1 to 0

- Often known as mutex lock. A key difference between mutex and binary semaphore is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).

Binary Semaphore

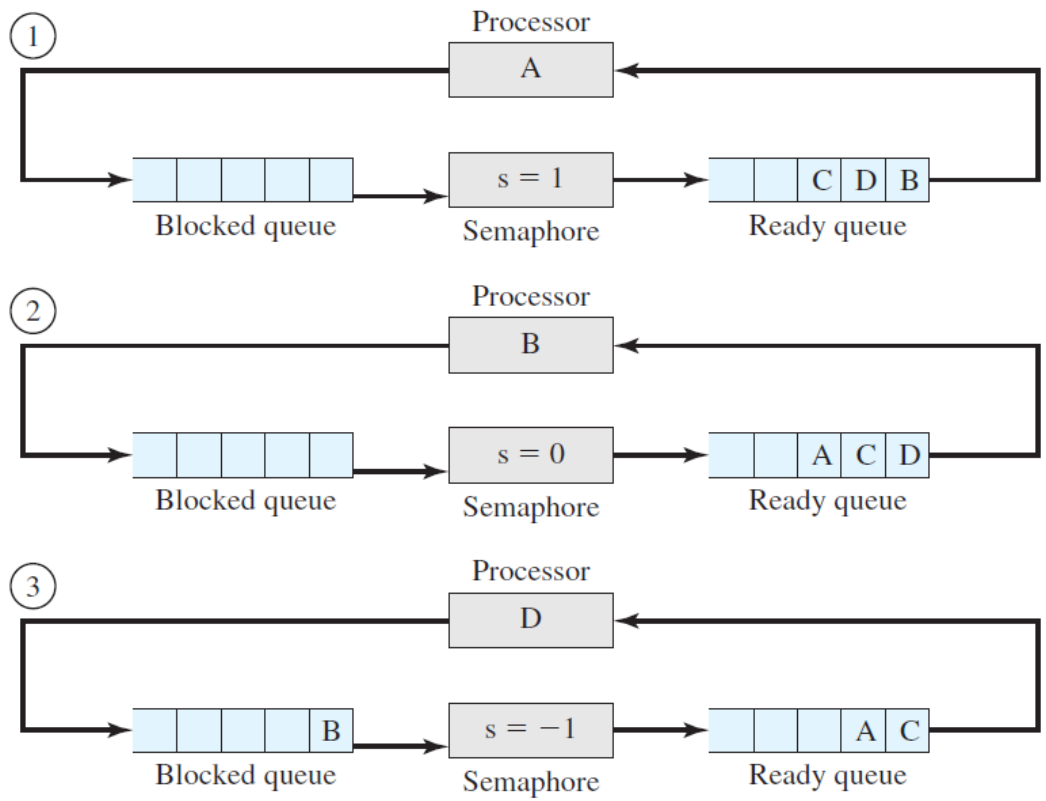
```
struct binary_semaphore {
        enum {zero, one} value;
        queueType queue;
                };
void semWaitB(binary_semaphore s)
                {
                if (s.value == one)
                                s.value = zero;
                else {
                        /* place this process in s.queue */;
                        /* block this process */;
                }
                }
```

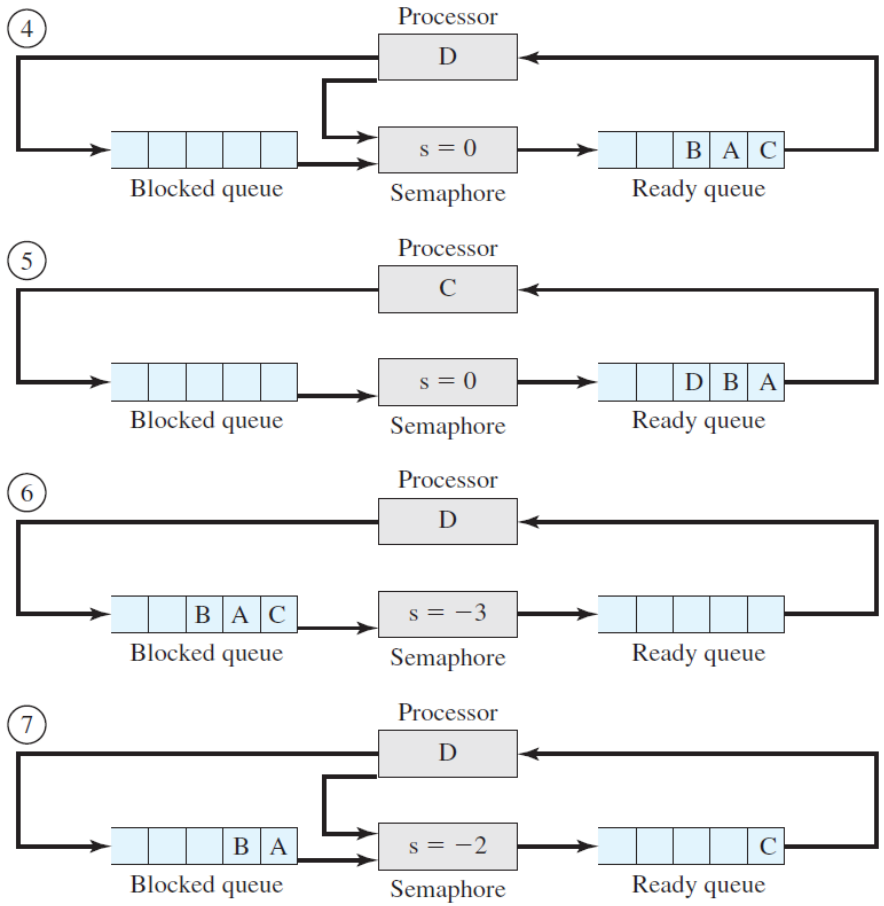
```
void semSignalB(semaphore s)
        {
                if (s.queue is empty())
                        s.value = one;
                else {
                        /* remove a process P from s.queue */;
                        /* place process P on ready list */;
                      }
        }
```

- For both counting semaphores and binary semaphores, a queue is used to hold processes waiting on the semaphore.
- The fairest removal policy is first-in-first-out (FIFO):The process that has been blocked the longest is released from the queue first; a semaphore whose definition includes this policy is called a **strong semaphore**.
- A semaphore that does not specify the order in which processes are removed from the queue is a **weak semaphore**.

Example

- Processes A, B, and C depend on a result from process D.
- Initially (1), A is running; B, C, and D are ready; and the semaphore count is 1, indicating that one of D's results is available.
- When A issues a semWait instruction on semaphore *s*, the semaphore decrements to 0, and A can continue to execute; subsequently it rejoins the ready queue.
- Then B runs (2), eventually issues a semWait instruction, and is blocked, allowing D to run (3).When D completes a new result, it issues a semSignal instruction, which allows B to move to the ready queue (4).
- D rejoins the ready queue and C begins to run (5) but is blocked when it issues a semWait instruction.
- Similarly, A and B run and are blocked on the semaphore, allowing D to resume execution (6).When D has a result, it issues a semSignal, which transfers C to the ready queue.
- Later cycles of D will release A and B from the Blocked state.


1
Processor
A
s = 1
C D B
Blocked queue
Semaphore
Ready queue
2
Processor
B
s = 0
A C D
Blocked queue
Semaphore
Ready queue
3
Processor
D
B
s = −1
A C
Blocked queue
Semaphore
Ready queue


4
Processor
D
s = 0
Blocked queue
Semaphore
B A C
Ready queue
5
Processor
C
s = 0
Blocked queue
Semaphore
D B A
Ready queue
6
Processor
D
B A C
s = −3
Blocked queue
Semaphore
Ready queue
7
Processor
D
B A
s = −2
Blocked queue
Semaphore
C
Ready queue

Semaphores can be used to force synchronization (precedence) if the preceding process does a signal at the end, and the follower does wait at beginning. For example, here we want P1 to execute before P2.

P1:

```
statement 1;
signal ( synch );
```

P2:

```
wait ( synch );
statement 2;
```

DEADLOCKS:

May occur when two or more processes try to get the same multiple resources at the same time.

P1:

```
wait(S);
wait(Q);
.....
signal(S);
signal(Q);
```

P2:

```
wait(Q);
wait(S);
.....
signal(Q);
signal(S);
```

Bounded-Buffer Problem using Semaphores

Three semaphores

- mutex – for mutually exclusive access to the buffer – initialized to 1
- used – counting semaphore indicating item count in buffer – initialized to 0
- free – number of free items – initialized to BUF_SZ

```
void producer() {
   while (1) {  /* Generate new item into nextProduced */
                wait(free);
                wait(mutex);
                buffer[in] =  nextProduced; in = (in + 1) % BUF_SZ;
                signal(mutex);
                signal(used);
   }
}
```

```
void consumer() {
  while (1) { wait(used);
              wait(mutex);
              nextConsumed = buffer[out];  out = (out + 1) % BUF_SZ;
              signal(mutex);
              signal(free);
              /* Process the item from nextConsumed */
  }
}
```

Critical Regions and Monitors

Critical Regions

- High-level synchronization construct
- A shared variable ***v*** of type ***T***, is declared as:

  v: shared T
- Variable ***v*** accessed only inside statement

  region v when B do S

  where ***B*** is a boolean expression.

- While statement ***S*** is being executed, no other process can access variable ***v***.

Critical Regions

- Regions referring to the same shared variable exclude each other in time.

- When a process tries to execute the region statement, the Boolean expression *B* is evaluated. If *B* is true, statement *S* is executed. If it is false, the process is delayed until *B* becomes true and no other process is in the region associated with *v*.

Example – Bounded Buffer Producer Consumer Problem

Shared data:

```
struct buffer {
    int pool[n];
    int count, in, out;
}
```

Bounded Buffer Producer Process

- Producer process inserts **nextp** into the shared buffer

```
region buffer when( count < n) {
pool[in] = nextp;
in:= (in+1) % n;
count++;
}
```

- Consumer process removes an item from the shared buffer and puts it in **nextc**

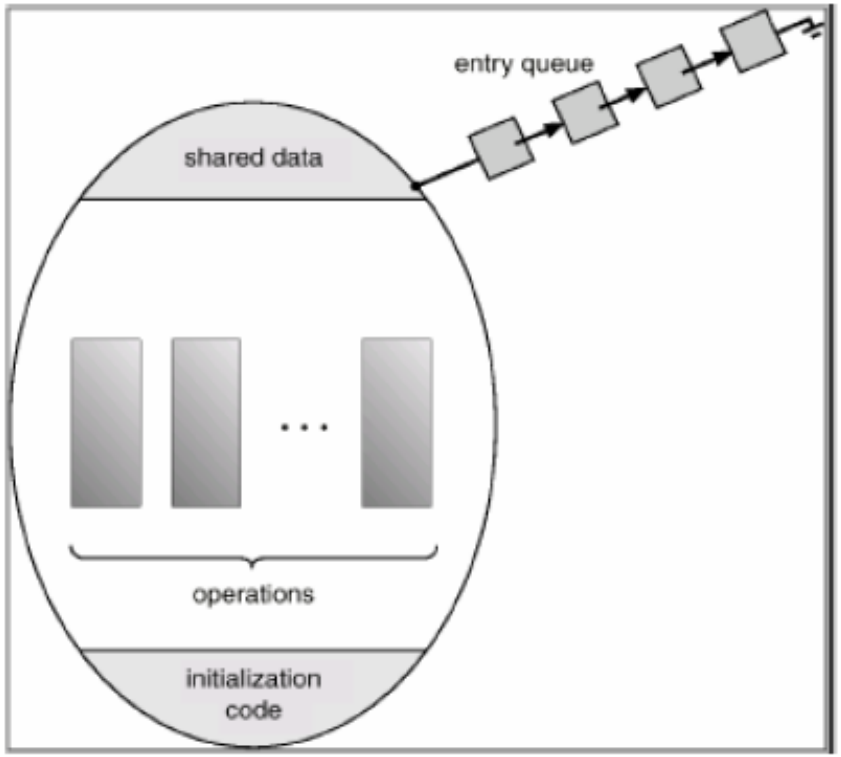
```
region buffer when (count > 0) {
nextc = pool[out];
    out = (out+1) % n;
    count--;
}
```

Monitors

- High-level synchronization construct which allows the safe sharing of an abstract data type among concurrent processes.
- A monitor is a class, in which all data are private, and with the special restriction that only one method within any given monitor object may be active at the same time.
- Monitor methods can only access the shared data within the monitor, they cannot access an outside variable and any data passed to them only as parameters.
- The variables or data local to a monitor cannot be directly accessed from outside the monitor.

```
monitor monitor-name
        {
            shared variable declarations
            procedure body P1 (…)  {
                    . . .
                     }
            procedure body P2 (…) {
                    . . .
                    }
            procedure body Pn (…) {
        . . .
                    }
        {
        initialization code
        }
         }
```

Schematic view of a Monitor



The monitor construct allows only one process at a time to be active within the monitor

To allow a process to wait within the monitor, a condition variable must be declared as

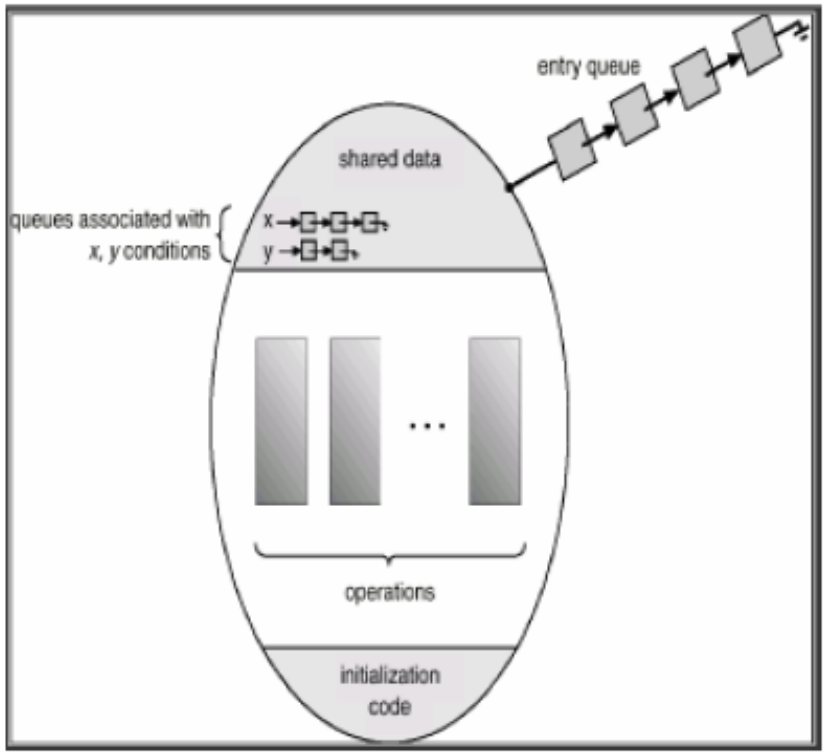
condition x, y;

Condition variable can only be used with the operations wait() and signal().

The operation x.wait() means that the process invoking this operation is suspended until another process invokes x.signal().

The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the *signal* operation has no effect.

Monitor with condition variables



Suppose that when the x.signal() operation is invoked by a process P, there is a suspended process Q associated with condition x.

If the suspended process Q is allowed to resume its execution, the signaling process P must wait.

If not, both P and Q would be active simultaneously within the monitor.

Now two possibilities exist :

i) Signal and wait : P either waits until Q leaves the monitor or waits for another condition

ii) Signal and continue : Q either waits until P leaves the monitor or waits for another condition

The advantage of monitors is the flexibility they allow in scheduling the processes waiting in queues.

Drawbacks:

- The major drawback of monitors is the absence of concurrency if a monitor encapsulates the resource, since only one process can be active within a monitor at a time.
- Another drawback is possibility of deadlocks in the case of nested monitor calls.

Interprocess Communication

Message passing systems

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment where the communicating processes may reside on different computers connected by a network.
- A message passing facility provides at least two operations:

  send (message) and receive (message).

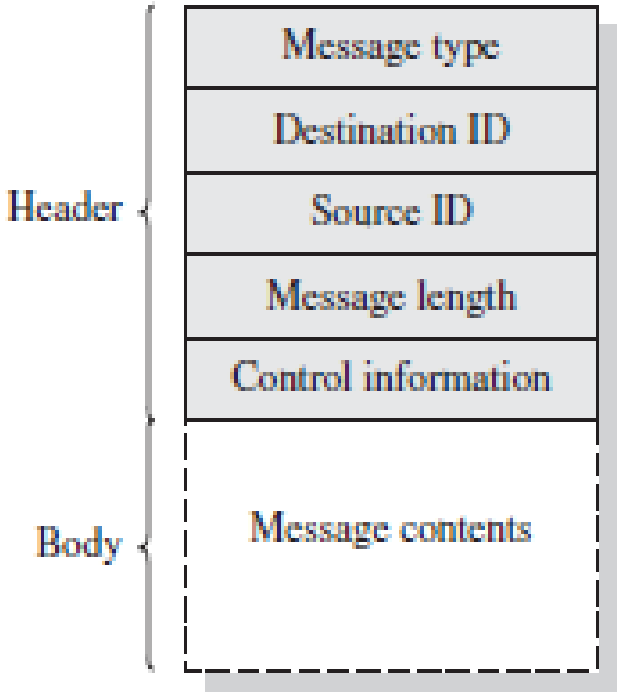
Issues related to message passing systems

- Message size:

Messages sent by a process can be of either fixed or variable size. If only fixed - sized messages can be sent, the system – level implementation is straight forward.

Variable sized messages require a more complex system level implementation.

Message Format



- Communication link:

If two processes want to communicate, they must send messages to and receive messages from each other, a communication link must exist between them.

A communication link has the following properties :

A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other’s identity to communicate.

A link is associated with exactly two processes.

Between each pair of processes, there exists exactly one link.

Direct or indirect communication

Naming: Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. The send () and receive() primitives are defined as:

- Send (P, message) – send a message to process P
- Receive (Q, message) – receive a message from process Q

This scheme exhibits symmetry in addressing; that is both the sender process and receiver process must name the other to communicate.

A variant of this scheme employs asymmetry in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender.

Here, the send () and receive () primitives are defined as:

Send (P, message) – send a message to process P

Receive (id, message) – receive a message from any process, the variable id is set to the name of the process with which communication has taken place.

With indirect communication, the messages are sent to and received from mail boxes or ports.

- A mail box can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- Each mail box has a unique identification.
- A process can communicate with some other process via a number of different mail boxes.

Two processes can communicate only if the processes have a shared mail box.

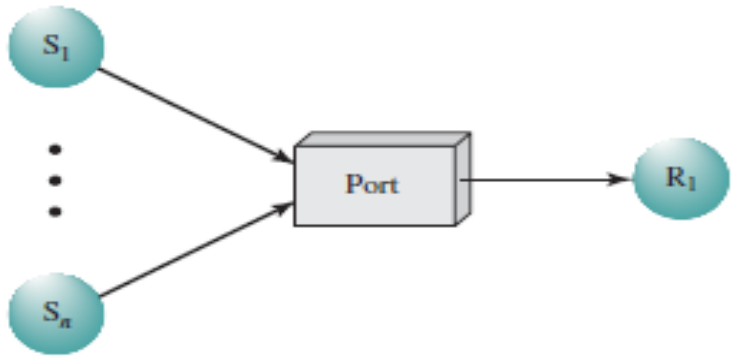
The send () and receive() primitives are defined as:

- Send (A, message) – send a message to mail box A
- Receive (A, message) – receive a message from mail box A.

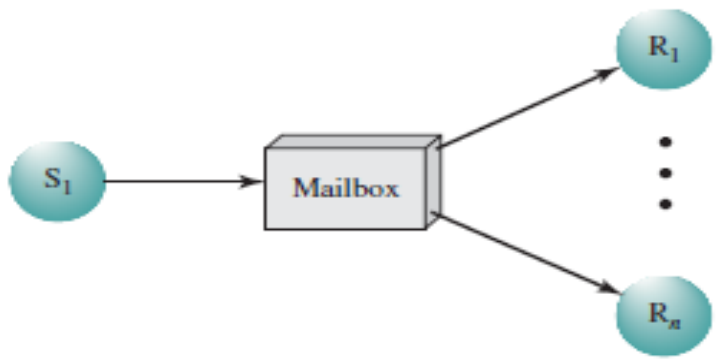
In this scheme, a communication link has the following properties –

- A link is established between a pair of processes only if both members of the pair have a shared mail box.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mail box.

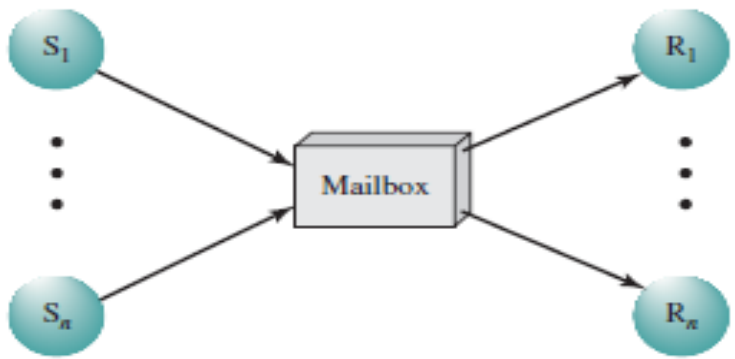
(a) One to one



(b) Many to one



(c) One to many



(d) Many to many

A mail box may be owned either by a process or by the OS.

- If the mail box is owned by a process, then we distinguish between the owner and the user. When a process that owns the mail box terminates, the mail box disappears. Any process that subsequently sends a message to this mail box must be notified that the mail box no longer exists.
- A mail box owned by the OS is independent and is not attached to any particular process. The OS then must provide a mechanism that allows a process to do the following:
  - Create a new mail box
  - Send and receive messages through the mail box
  - Delete a mail box

Synchronous or asynchronous communication

Communication between processes takes place through calls to send () and receive () primitives. Message passing may be either blocking or non blocking – also known as synchronous and asynchronous.

- Blocking send: The sending process is blocked until the message is received by the receiving process or the mail box.
- Non blocking send: The sending process sends the message and resumes operation.
- Blocking receive: The receiver blocks until a message is available.
- Non blocking receive: The receiver retrieves either a valid message or a null.

- When both send () and receive () are blocking, we have a rendezvous between the sender and the receiver.

Automatic or explicit buffering

- Buffering: Whether the communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.
- Such queues can be implemented in three ways –
  - Zero capacity: The queue has the maximum length of zero; thus the link cannot have any messages waiting in it.
  - Bounded capacity: The queue has finite length n; thus, at most n messages can reside in it.
  - Unbounded capacity: The queues length is infinite; thus any number of messages can wait in it. The sender never blocks.
- The zero capacity buffer is sometimes referred to as a message system with no buffering; the other cases are referred to as systems with automatic buffering.

```
const int
    capacity = /* buffering capacity */ ;
    null = /* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
     receive (mayproduce,pmsg);
     pmsg = produce();
     send (mayconsume,pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
     receive (mayconsume,cmsg);
     consume (cmsg);
     send (mayproduce,null);
    }
}
void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1;i<= capacity;i++) send (mayproduce,null);
    parbegin (producer,consumer);
}
```

Barrier Synchronization:

This synchronization mechanism is for group of processes.

Some applications are divided into phases and the set of processes can move on to next phase only when all the processes in that group complete their execution in the current phase.

A process wait on the barrier until all other processes reach the barrier.

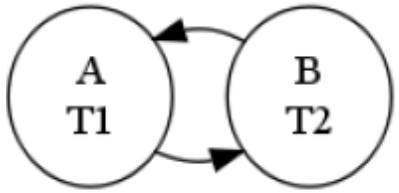
Deadlocks

Process Deadlocks

- *Definition : A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause*
- i.e. A deadlock is a situation where a process or a set of processes is blocked, waiting on an event which may be the release of computing resource or other hardware resources, that can only be caused by a member of the set and that will never occur.
- The formation and existence of deadlocks in a system lowers system efficiency.
- Therefore to avoid performance degradation, a system should be deadlock free or deadlocks should be quickly detected and recovered.

Examples

- System has 2 tape drives. P1 and P2 each hold one tape drive and each needs the other one



- Semaphores A and B each initialized to 1

P0	*P1*
wait(A)	*wait*(*B*)
wait(*B*)	*wait(A)*

Deadlock vs Starvation

- A process is *deadlocked* if it is waiting for an event that will never occur. Typically, more than one process will be involved in a deadlock and the processes are involved in a circular wait.
- Starvation occurs when a process waits for a resource that continually becomes available but is never assigned to that process because of priority or a flaw in the design of a scheduler e.g. the process is ready to proceed but never gets the CPU.

- Two major differences between deadlock and starvation:

  1. In starvation, it is not certain that a process will ever get the requested resource, whereas a deadlocked process is permanently blocked because the required resource never becomes available unless external actions are taken.

  2. In starvation, the resource under contention is in continuous use whereas in deadlock resources are not used as the processes are blocked.

Resources:

Resource is a commodity required by a process to execute.

Under normal operation, a resource allocation proceed like this::

1. Request a resource (suspend until available if necessary).
2. Use the resource.
3. Release the resource.

Resources can be of several types:

- Serially Reusable Resources

  e.g. CPU cycles, memory space, I/O devices, files

A process acquires, uses and then releases the resource.

- Consumable Resources

  Produced by a process, needed by a process

  e.g. Messages, buffers of information, interrupts

A process creates, acquires and then use the resource. Resource ceases to exist after it has been used.

Another classification is

- Preemptable - the resource can be taken away from its current owner (and given back later). An example is memory.
- Non-preemptable- the resource cannot be taken away. An example is a printer.

Fundamental causes of deadlocks

The following four conditions are *necessary* for a deadlock to occur.

- Mutual exclusion: A resource can be assigned to at most one process at a time (no sharing).
- Hold and wait: A processing holding a resource is permitted to request another. i.e. a process holding at least one resource is waiting to acquire additional resources held by other processes
- No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task i.e. A process must release its resources; they cannot be taken away.
- Circular wait: There must be a chain of processes such that each member of the chain is waiting for a resource held by the next member of the chain.

  There exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

System model

System can be modelled using Resource allocation graph. Resource types are denoted by, R_1, R_2, ..., R_n , and each resource R has W_i instances.

Resource Allocation Graph (RAG)

A set of vertices V and a set of edges E

V is partitioned into 2 types

P = {P1, P2,…,Pn} - the set of processes in the system

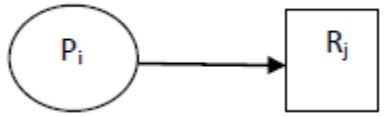
R = {R1, R2,…,Rn} - the set of resource types in the system

Two kinds of edges

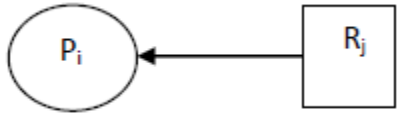
Request edge - Directed edge Pi ---> Rj

Assignment edge - Directed edge Rj ----> Pi

P_i requests an instance of R_j

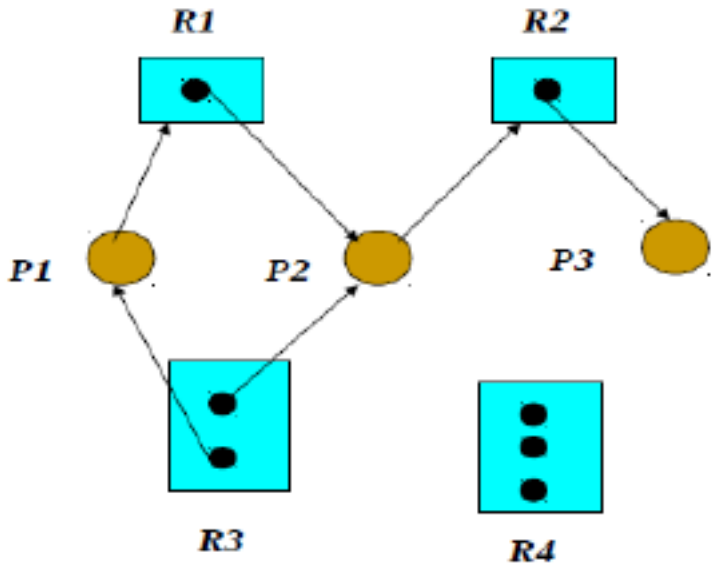


P_i is holding an instance of R_j

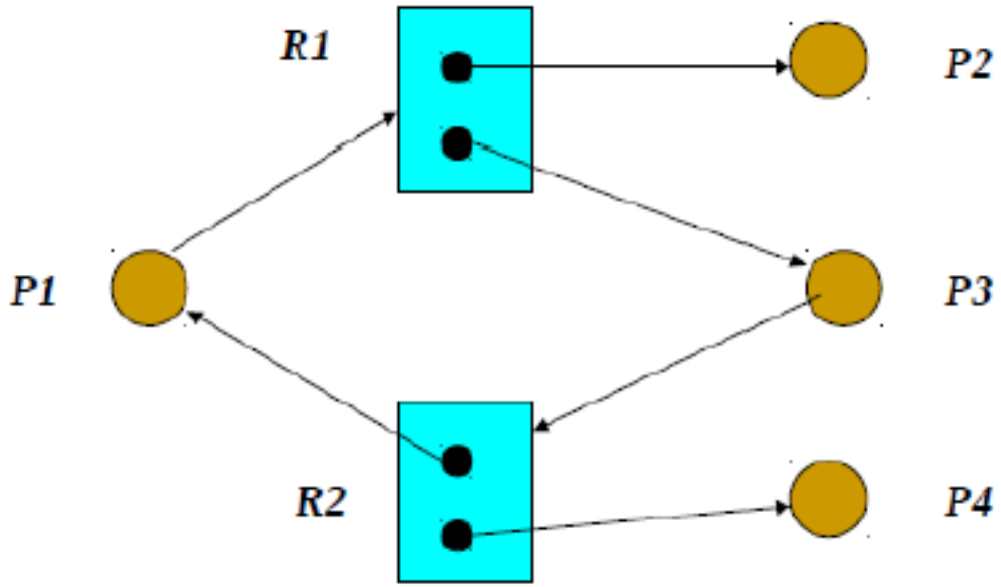


- The request for resources by processes can be modelled using RAG.
- If a resource allocation graph contains no cycles, then no process is deadlocked.
- If a RAG contains a cycle, then a deadlock may exist. Therefore, a cycle means deadlock is *possible*, but not necessarily *present*.
- A cycle is not sufficient proof of the presence of deadlock. A cycle is a *necessary* condition for deadlock, but not a *sufficient* condition for deadlock.

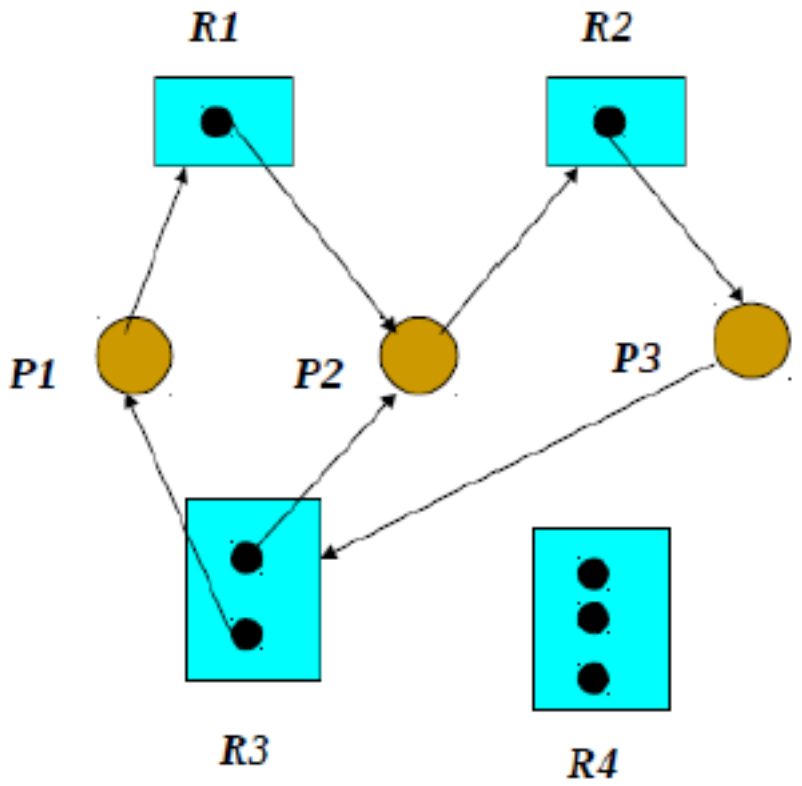
RAG with no cycles



RAG with cycles



RAG with cycles and deadlocks



If RAG contains no cycles then NO DEADLOCK.

If RAG contains a cycle and

if only one instance per resource type, then deadlock

if several instances per resource type, possibility of deadlock.

Four *strategies* for handling with deadlocks

- Ignore the problem - Ostrich Algorithm Popular in Distributed Systems.
  - Reasonable if
    - deadlocks occur very rarely
    - cost of prevention is high
  - UNIX and Windows takes this approach
- Deadlock Prevention - Prevent deadlocks by violating one of the 4 necessary conditions.
- Deadlock Avoidance - Avoid deadlocks by carefully deciding when to allocate resources.
- Deadlock Detection and recovery - Detect deadlocks and recover from them

Deadlock Prevention

Restrain the ways request can be made

- Prevent Mutual Exclusion
  - Not required for sharable resources (e.g., read-only files);
  - Must hold for non-sharable resources
  - Prevention not possible, since some devices are intrinsically non-sharable

- Prevent Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

1. Pre-allocate
   - do not pick up one chopstick if you cannot pick up the other
   - for a process that copies data from DVD drive to a file on disk and then prints it from there:
     - request DVD drive
     - request disk file
     - request printer

2. A process can request resources only when it has none
   - request DVD drive and disk file
   - release DVD drive and disk file
   - request disk file and printer (no guarantee data will still be there)
   - release disk file and printer
- Disadvantages: inefficient, possibility of starvation.

- Prevent No Preemption
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
  - some resources cannot be feasibly preempted (e.g., printers, tape drives)

- Prevent Circular Wait
  - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

- Requires that the system has some additional ***a priori*** information available
- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

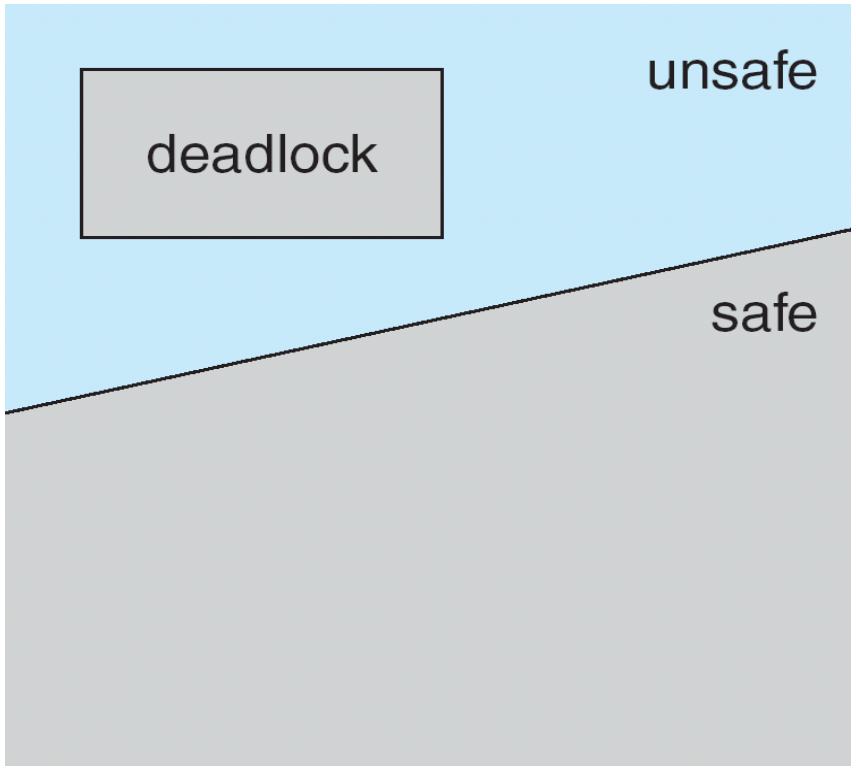
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State



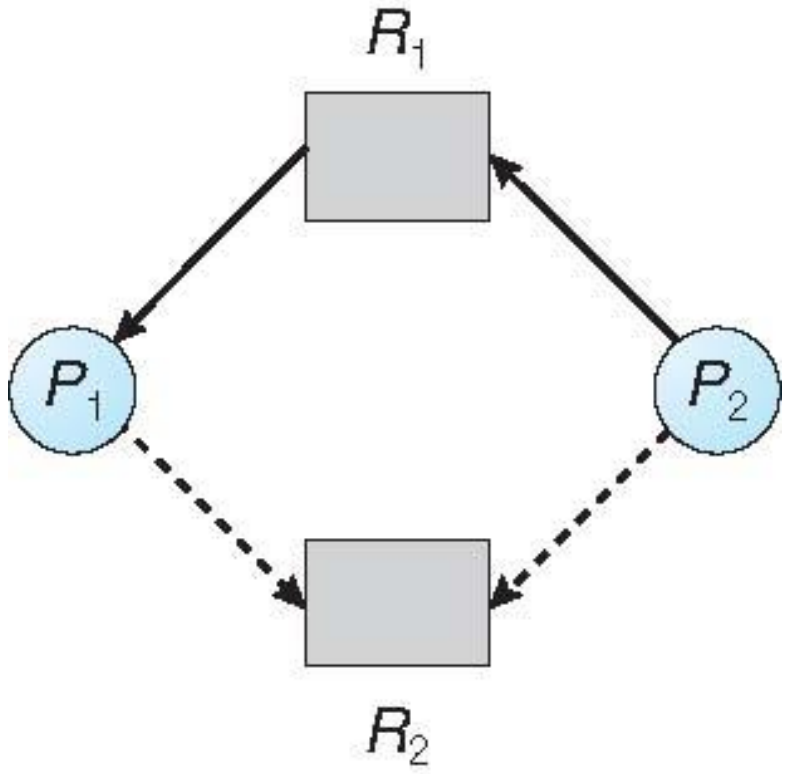
Avoidance Algorithms

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

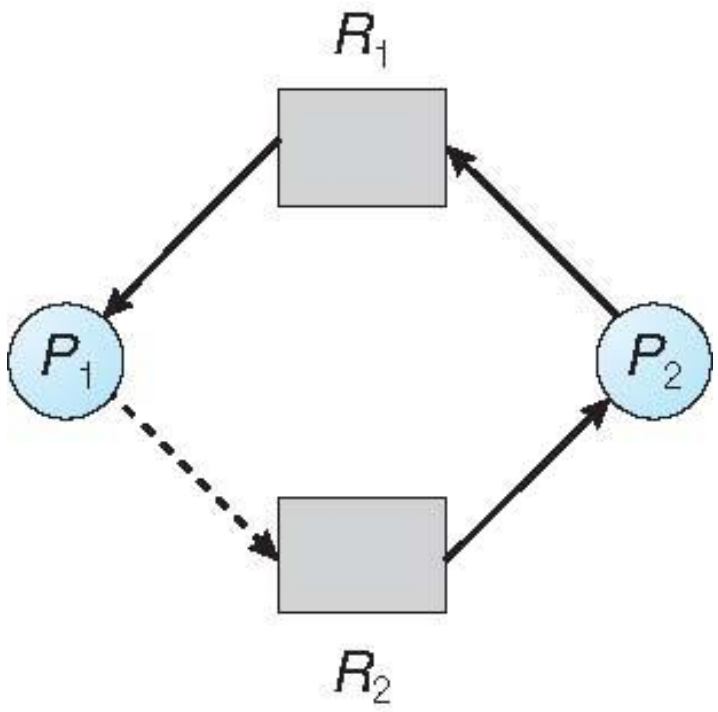
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker’s Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

- Let n = number of processes, and m = number of resources types.
- **Available***:* Vector of length m. If available [j] = k, there are k instances of resource type R_j available
- **Max***: n x m* matrix. If *Max* [i,j] = k, then process P_i may request at most k instances of resource type R_j
- **Allocation***:* n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
- **Need***:* n x m matrix. If *Need*[i,j] = k, then P_i may need k more instances of R_j to complete its task

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

Safety Algorithm

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 *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

Resource-Request Algorithm for Process P_i

Request*$_i$** = request vector for process ***P*$_i$**. If ***Request*$_i$ [*j*] = *k then process ***P*$_i$** wants ***k*** instances of resource type ***R*$_j$**

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. Pretend to allocate requested resources to ***P*$_i$** by modifying the state as follows:

Available = Available – Request*$_i$*;

***Allocation*$_i$ = *Allocation*$_i$ + *Request*$_i$;**

***Need*$_i$ = *Need*$_i$ – *Request*$_i$;**

- If safe $\Rightarrow$ the resources are allocated to ***P*$_i$**
- If unsafe $\Rightarrow$ ***P*$_i$** must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P_0 through P_4;
  3 resource types:
  A (10 instances), *B* (5 instances), and *C* (7 instances)
- Snapshot at time T_0:

	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	

Example

- 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 in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request $\leq$ Available (that is, (1,0,2) $\leq$ (3,3,2) $\Rightarrow$ true

	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 < $\boldsymbol{P_1}$, $\boldsymbol{P_3}$, $\boldsymbol{P_4}$, $\boldsymbol{P_0}$, $\boldsymbol{P_2}$> satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

- It requires an algorithm which examines the state of the system to determine whether a deadlock has occurred
- It has overhead of run-time cost for maintaining necessary information and executing the detection algorithm and potential losses inherent in recovering from deadlock.

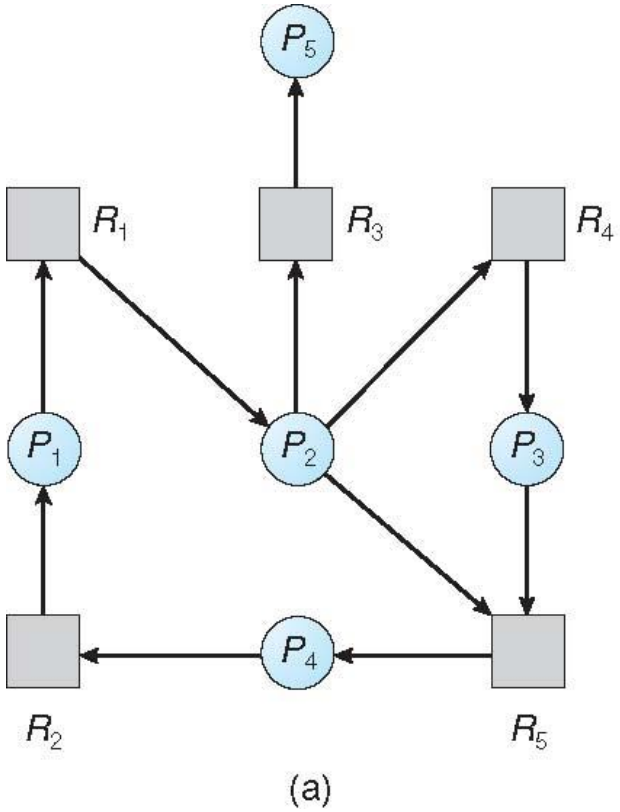
Single Instance of Each Resource Type

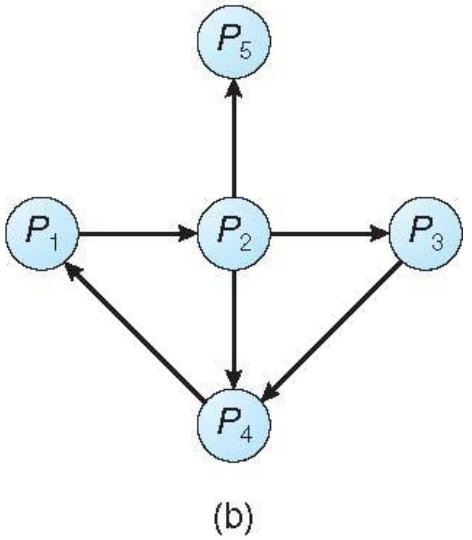
- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j

- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

- 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

Resource-Allocation Graph and Wait-for Graph





Resource-Allocation Graph

Corresponding wait-for graph

Several Instances of a Resource Type

- **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*] = *k***, then process P_i is requesting ***k*** more instances of resource type R_j.

Detection 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 $\boldsymbol{Allocation_i \neq 0}$, then
   Finish*[i] = *false; otherwise, ***Finish*[i] = *true***

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

   If no such ***i*** exists, go to step 4

Detection Algorithm

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

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)
- Snapshot at time T_0:

	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	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

Example

- P_2 requests an additional instance of type ***C***

	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

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle

- 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: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – criteria for selection to be made - minimize cost
- Rollback – return to some safe state, restart process for that state - some process may lose resources and cannot continue.
- Starvation – same process may always be picked as victim - include number of rollback in cost factor

- Not one solution is best for deadlock handling.
- Better solution can be provided by combining the three approaches namely prevention, avoidance and recovery, allowing the use of optimal approach for each class of resources in the system.
- Partition resources into hierarchically ordered classes and use most appropriate technique for handling deadlocks within each class.

Memory Management

Main Memory

- Main memory is the most critical resource as the speed of the programs depend on memory.
- Consists of large array of bytes or words each having their own address
- Limited size
- Stores programs and data required by CPU and I/O devices.

- Program must be brought into memory and placed within a process for it to be run.
- *Input queue* – collection of processes on the disk that are waiting to be brought into memory to run the program.
- User programs go through several steps before being run.

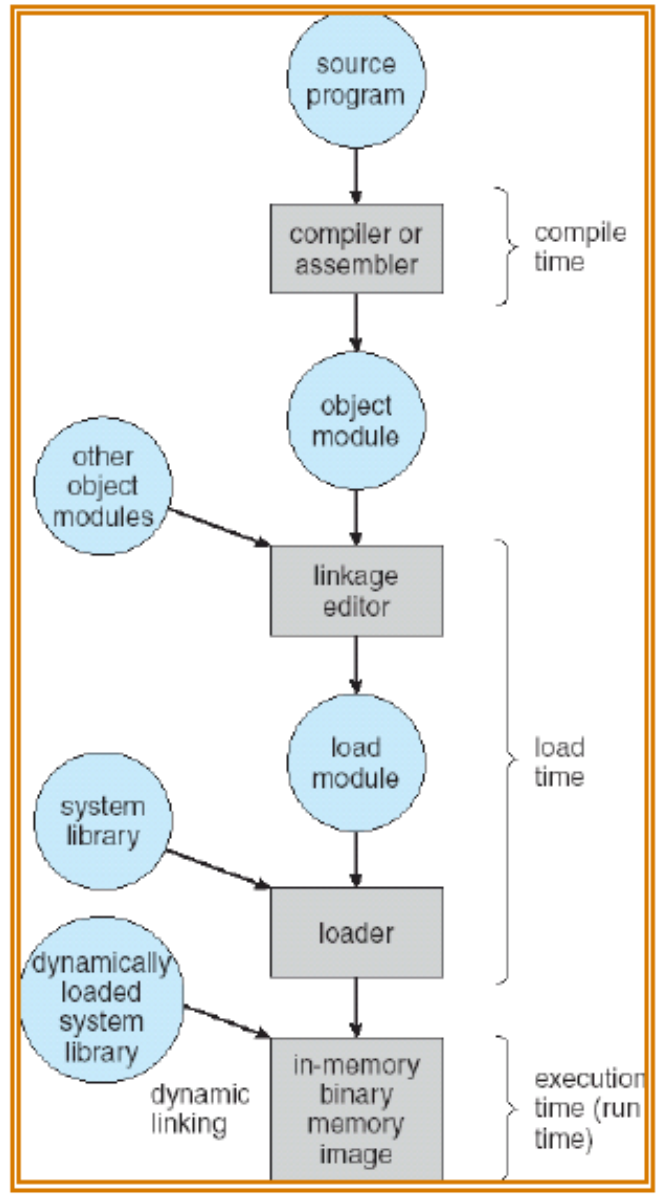
Memory Management

- Ideally programmers want memory that is
  - large
  - fast
  - non volatile
- Memory hierarchy
  - small amount of fast, expensive memory – cache
  - some medium-speed, medium price main memory
  - gigabytes of slow, cheap disk storage
- Memory manager handles the memory hierarchy

Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages.

- **Compile time**: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
- **Load time**: Must generate *relocatable* code if memory location is not known at compile time.
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., *base* and *limit registers*).


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

Logical vs. Physical Address Space

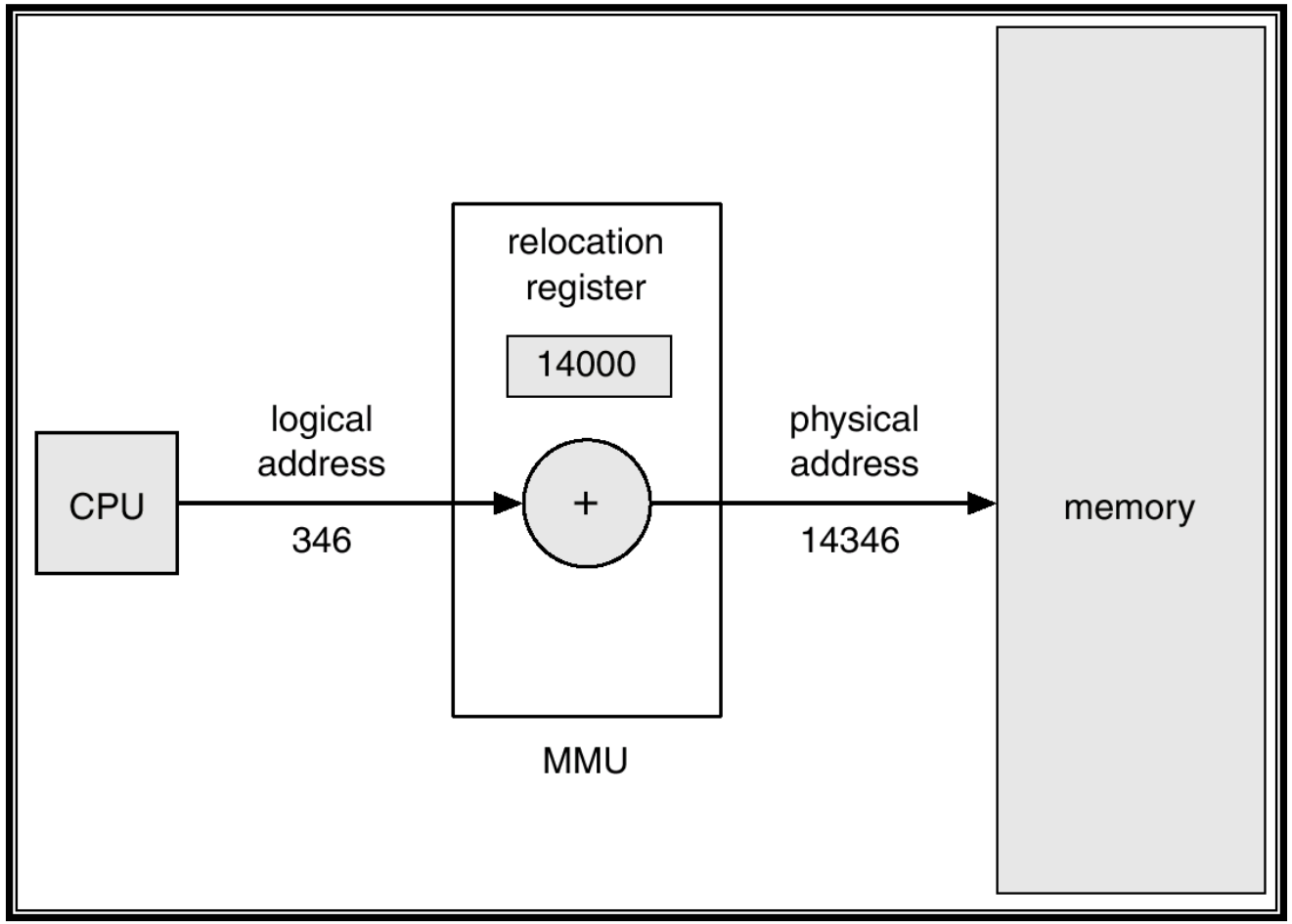
- The concept of a logical *address space* that is bound to a separate *physical address space* is central to proper memory management.
  - *Logical address* – generated by the CPU; also referred to as *virtual address*.
  - *Physical address* – address seen by the memory unit.

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address.
- In MMU scheme, 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.

Dynamic relocation using a relocation register



- The OS itself permanently occupies a portion of the memory for its programs and their static data.
- The remaining portion of the memory stores application programs and their static data and the dynamic data of processes and of OS.
- Memory space is recycled to store applications programs, process data and dynamic kernel programs and data. The subsystem that manages the allocation and de-allocation of memory space is called the *memory manager*.

Memory Manager

- It is an OS component concerned with the system's memory organization scheme and memory management strategies.
- It determines how available memory space is allocated to processes and how to respond to changes in a process's memory usage.
- It also interacts with special purpose memory management hardware (if any is available) to improve performance.
- An ideal memory manager should thus minimize wasted memory and have minimal time complexity and memory access overhead, while providing good protection and flexible sharing.

Allocation of memory can be classified as contiguous allocation and noncontiguous allocation.

- Contiguous allocation scheme allocates a single block of memory for the requesting process
- Noncontiguous scheme allocates chunks or pieces of memory to a process.

Contiguous Memory Management

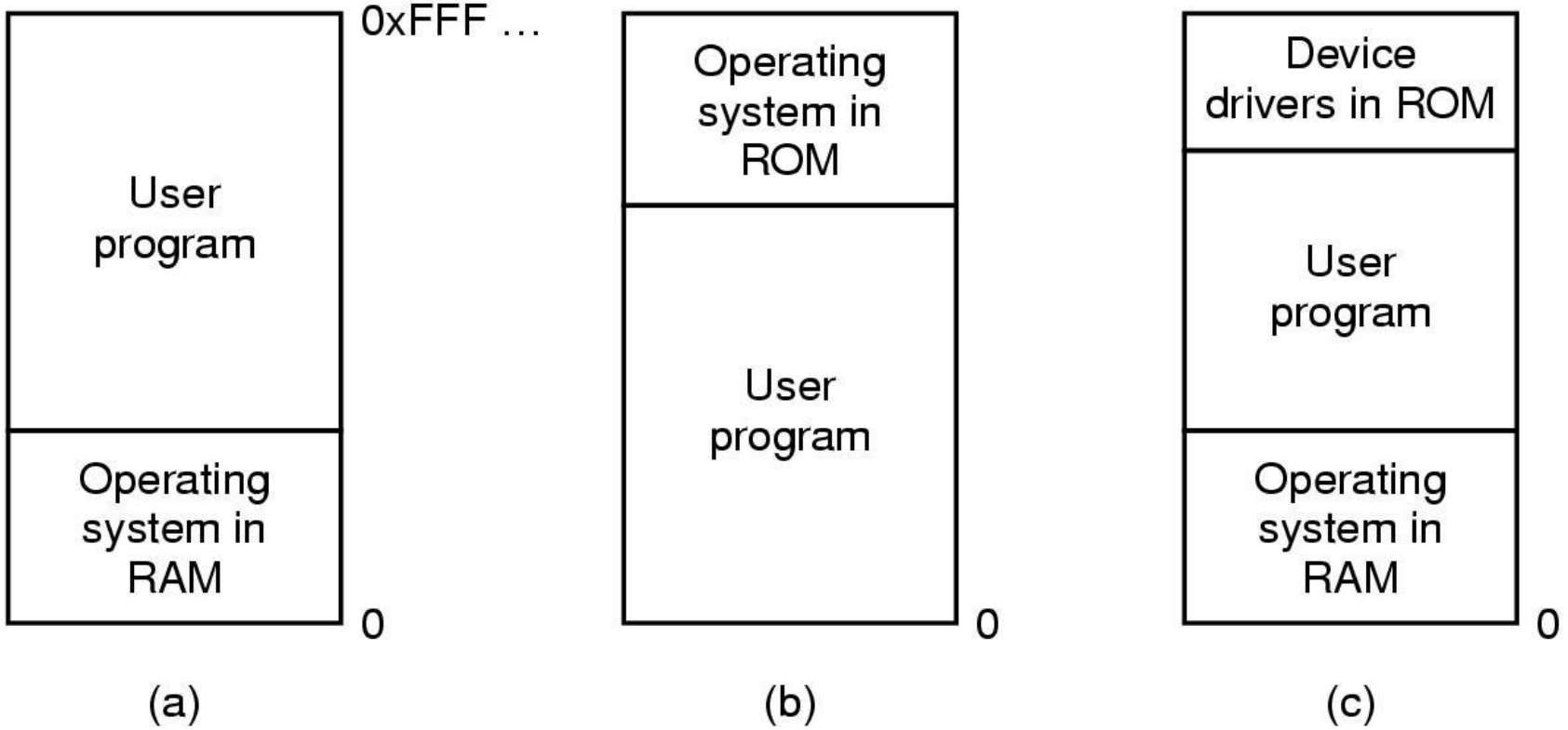
Single Process Monitor/ Single Partition Monoprogramming :

- This approach is one of the simplest ways of managing memory used in single process microcomputer.
- Memory is divided into two contiguous areas and one portion is permanently allocated to the resident portion of OS (monitor) and the remaining portion is allocated to the transient processes which are loaded and executed one at a time in response to user commands.

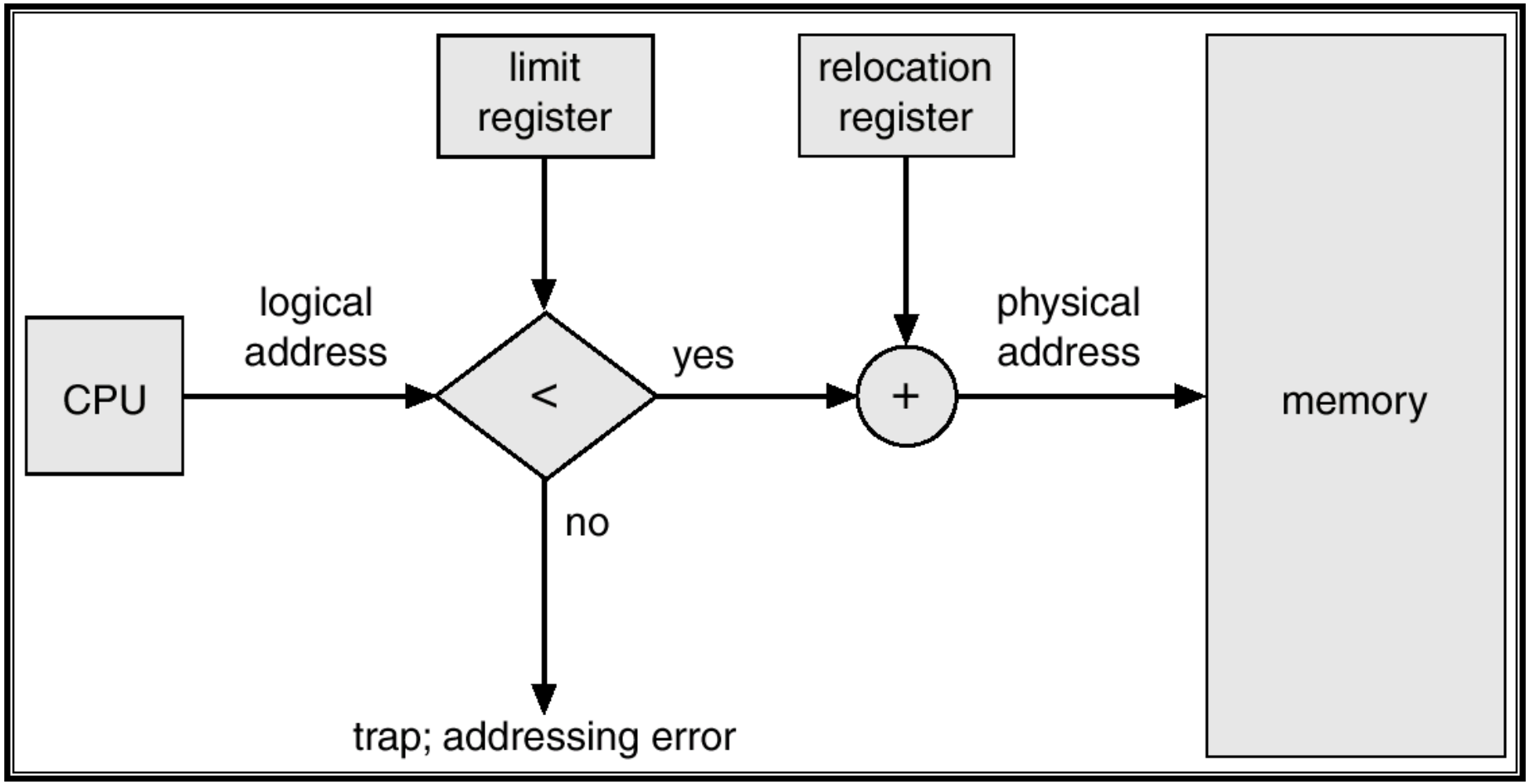
Basic Memory Management
Monoprogramming without Swapping or Paging

Three simple ways of organizing memory

- an operating system with one user process



Hardware Support for Relocation and Limit Registers



Advantages:

- Straightforward memory allocation; Absence of multiprogramming complexities; Relatively simple to design and to comprehend; Used in systems with little hardware support for more advance forms of memory management.

Disadvantages:

- Lack of support for multiprogramming reduces utilization of both processor and memory.

Partitioned memory allocation

- One way to support multiprogramming is to divide the available physical memory into several partitions each of which may be allocated to a different process.
- Depending on when and how partitions are created and modified, memory partitioning may be static or dynamic.

Static Partitioned memory allocation

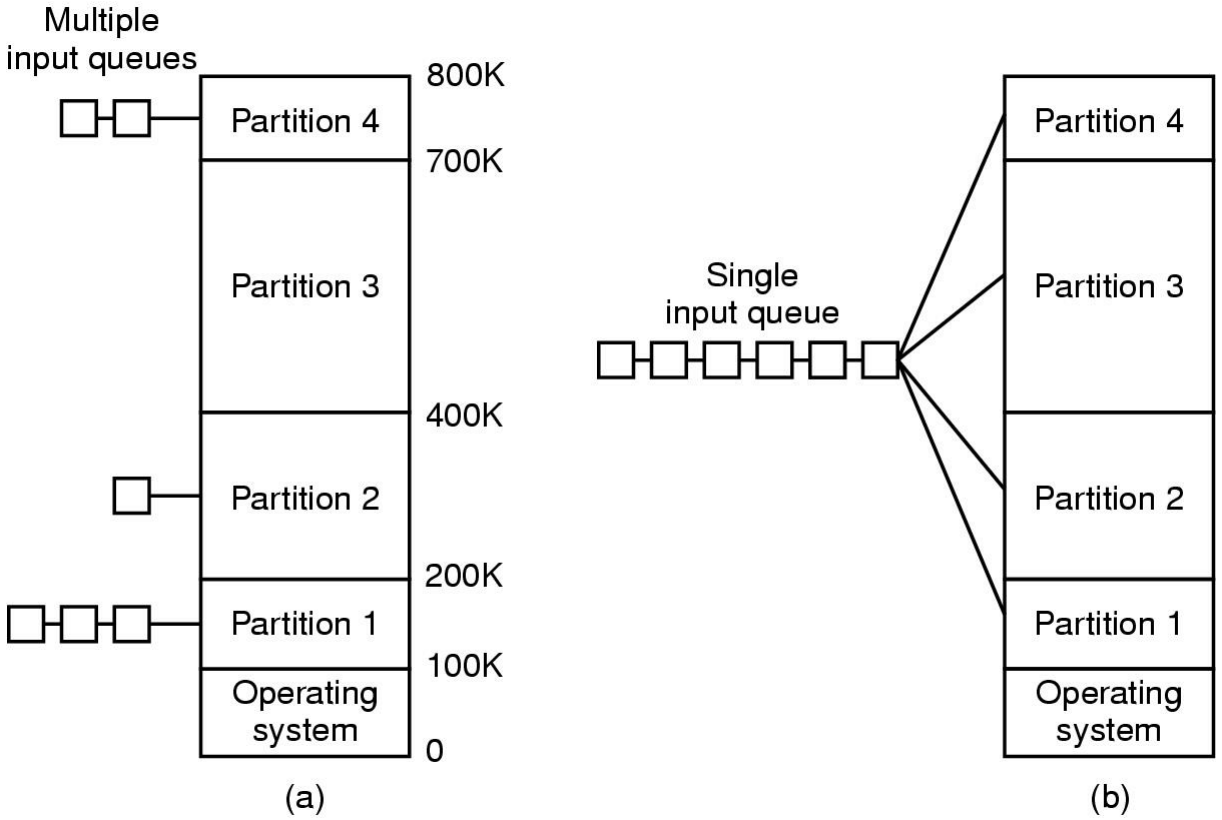
- Static partitioning implies that the division of memory is made at some time prior to the execution of user programs and that partitions remain fixed thereafter.

Allocation of partition:

- The OS must first allocate a memory region large enough to hold the process image which is a file that contains a program in executable form and the related data and may also contain process attributes such as priority and memory requirements.

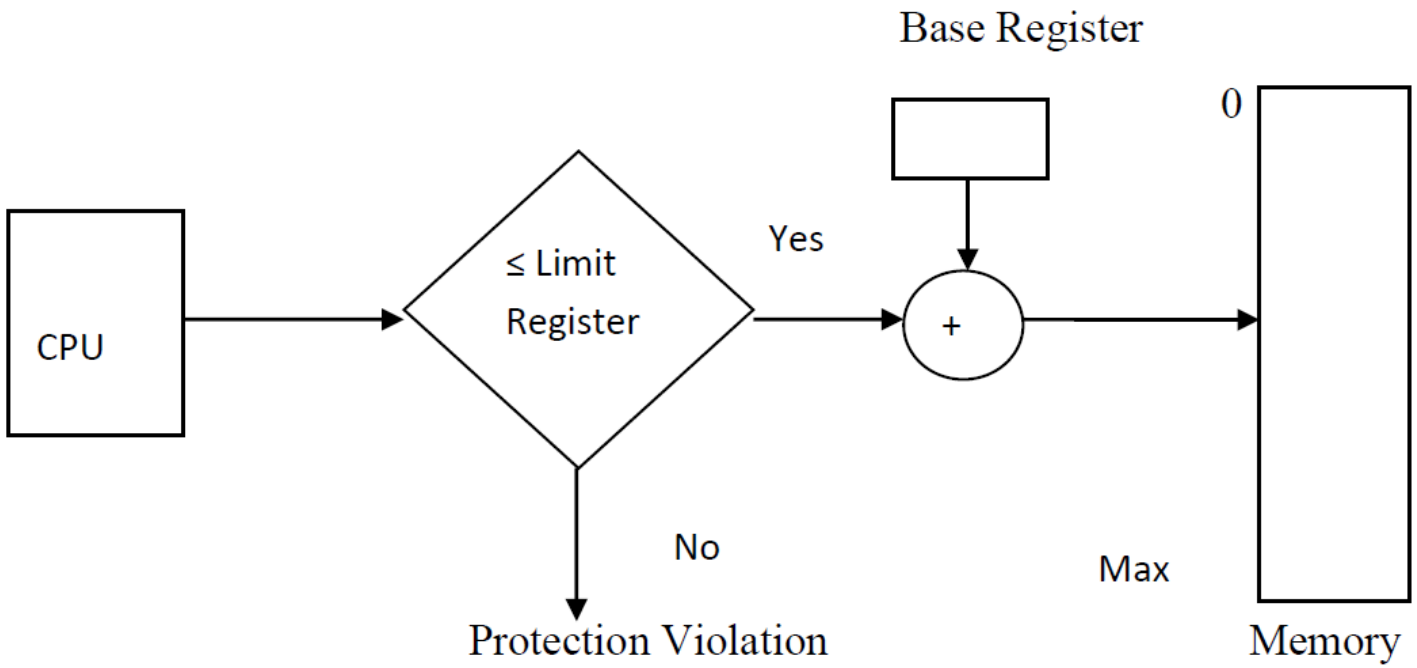
Multiprogramming with Fixed Partitions

- Fixed memory partitions
  - separate input queues for each partition
  - single input queue



- Partition Descriptor Table (PDT) is a data structure used by OS to collect current Partition Status and attributes such as Partition Number, Partition Base and Partition Size.
- In static partitioning, partition number, base and size are fixed, only the partition status varies depending on whether the partition is allocated or not.
- When a non-resident process is to be created or activated, OS attempts to allocate a free memory partition of sufficient size by searching the PDT.

Partition Number	Partition Base	Partition Size	Partition Status
0	0K	100K	Allocated
1	100K	200K	Free
2	300K	100K	Allocated
3	400K	200K	Allocated
4	600K	100K	Free
5	700K	150K	Allocated
6	850K	150K	Free


Base Register
0
CPU
≤ Limit
Register
Yes
+
No
Max
Protection Violation
Memory

The partition allocation strategy can be

- First Fit: Allocating the first free partition large enough to accommodate the process being created.
- Best Fit: OS allocates the smallest free partition that meets the requirements of the process under consideration.

Problems associated with allocation

1. No partition is large enough to accommodate the incoming process.

   Solution is to redefine the partitions accordingly or reduce the memory requirements by recoding or by using overlays.

2. All partitions are allocated.

   Solution is by deferring the loading of the incoming process until a suitable partition can be allocated to it or forcing a memory resident process to vacate a sufficiently large partition.

   Additional overhead of selecting a suitable victim and rolling it out to disk will be incurred. This operation is called swapping.

3. Some partitions are free, but none of them is large enough to accommodate the incoming process.

   Solution to this problem is by both deferring and swapping.

Swapping

Removing suspended or pre-empted processes from memory and their subsequent bringing back is called swapping.

The major responsibilities of swapper are

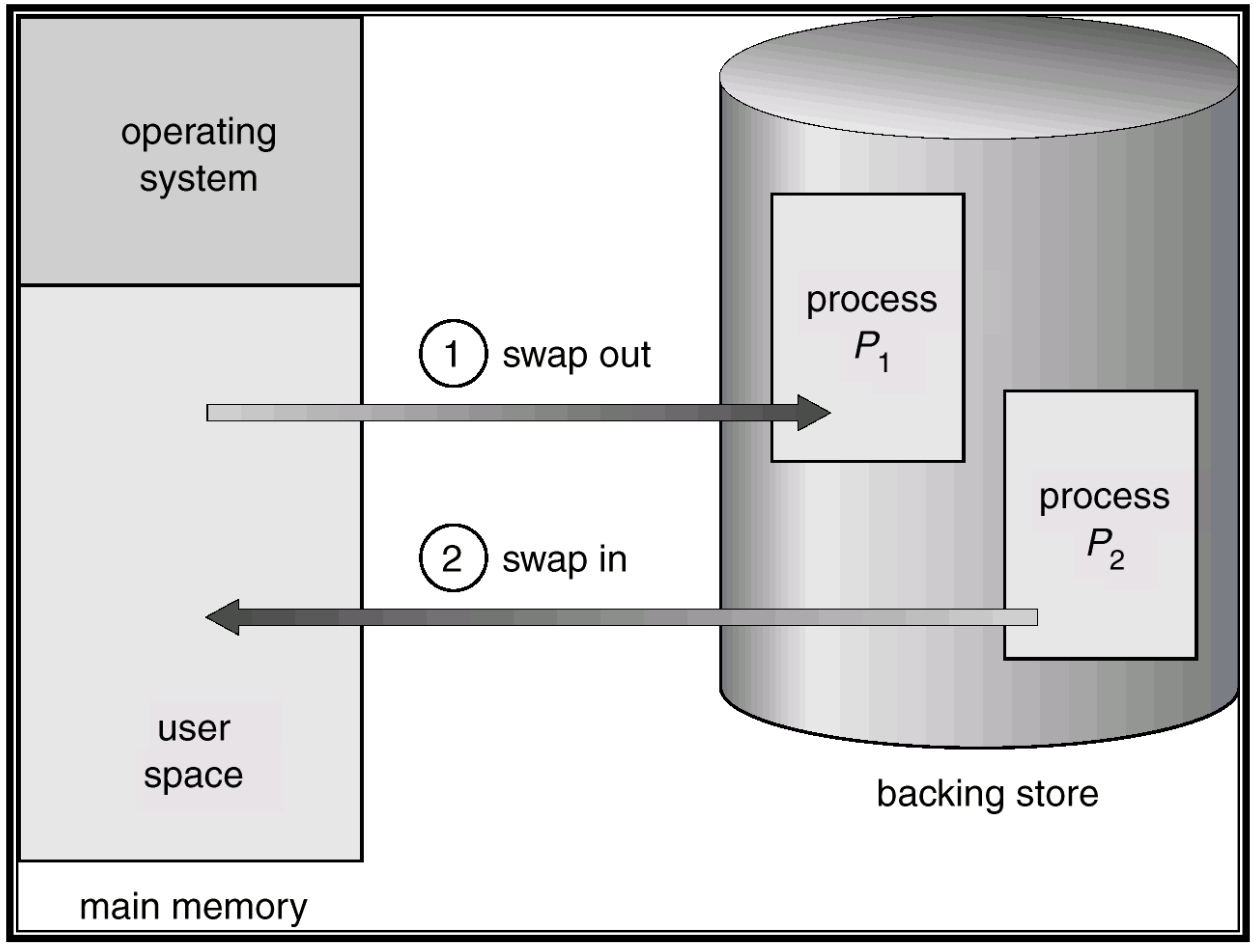
- Selection of processes to swap out,
- Selection of processes to swap in and
- Allocation and management of swap space.

- **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

Schematic View of Swapping

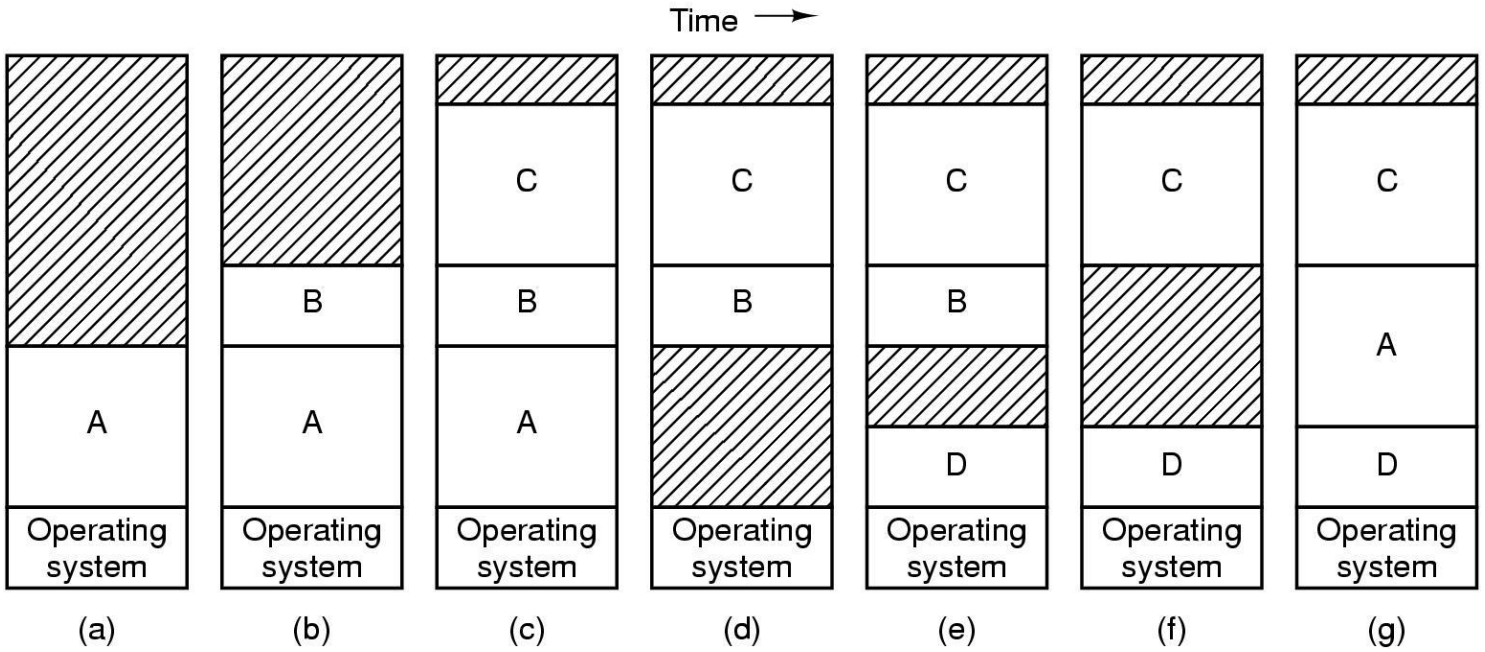


Swapping

Memory allocation changes as

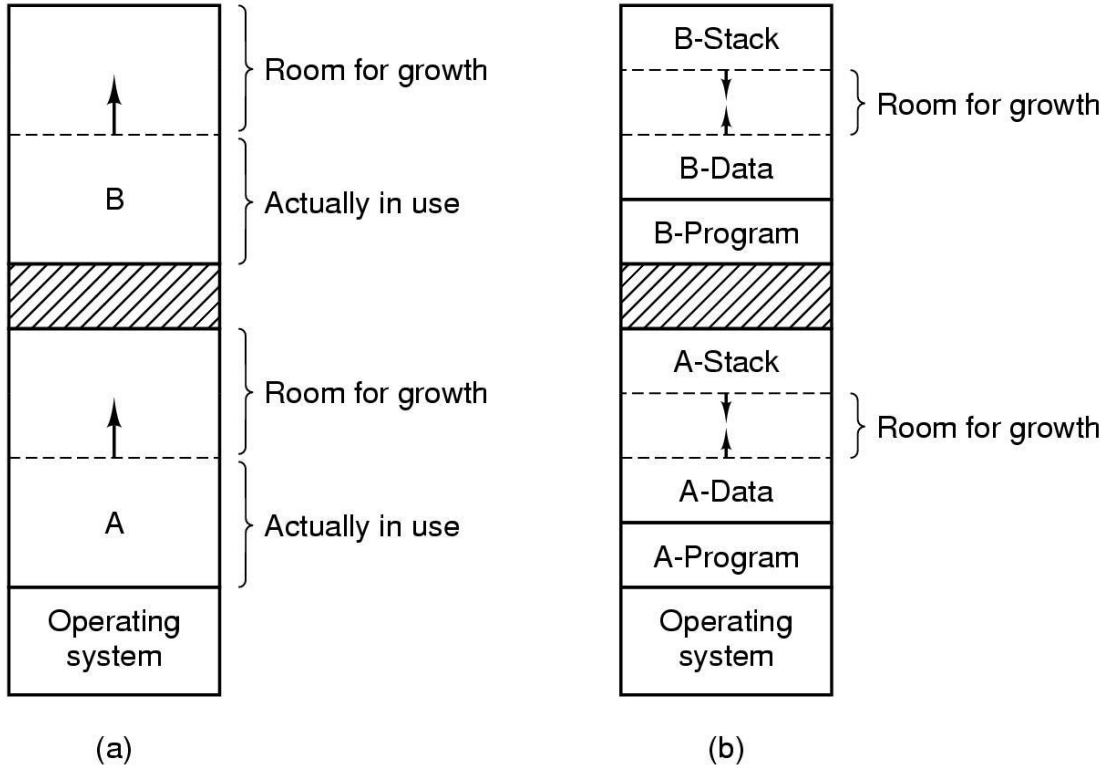
- processes come into memory
- leave memory

Shaded regions are unused memory



Swapping

- Allocating space for growing data segment
- Allocating space for growing stack & data segment



A separate swap file must be available for storing the dynamic image of a rolled out process.

Two options for placing a swap file

i) System wide swap file: Stored in a fast secondary storage device so as to reduce the latency of swapping. The disadvantage is the size of the file. If small it may lead to run time errors and system stoppage due to inability to swap a designated process.

ii) Dedicated per process swap file. File overflow errors are avoided but it consumes more disk space.

Advantage:

- Suitable for static environments where the workload is predictable and its characteristics are known.

Disadvantages:

- Inflexible and inability to adapt to changing system needs.
- Internal fragmentation of memory resulting from the difference between the size of a partition and the actual requirements of the process that resides in it.
- Fixed size of the partitions does not support dynamically growing data structures such as heaps or stacks.

Dynamic/ Variable Partitioned memory allocation

- Partition size may vary dynamically to overcome the problem of determining the number and sizes of partitions and to minimize internal fragmentation.

Allocation of Partition:

- In this method, in addition to PDT, a free list of partitions is maintained. Initially, the whole memory is free and it is considered as one large block.
- When a new process arrives, the OS searches for a block of free memory large enough for that process. The rest of the available free space are kept for the future usage.

- If a block/partition becomes free, then the OS tries to merge it with its neighbours if they are also free otherwise append it to the free list. This appending may lead to a free list of holes of very small size.
- External fragmentation is possible i.e. a suitable single partition meeting the process requirement may not be available, but if the sizes of the holes are added, it may be greater than or equal to the requesting process's size.

The procedure for creating a partition of size P for a requesting process T is as follows:

1. Search for a partition of size $F \geq P$. If no match is found then error.

2. Calculate $D = F - P$. If $D \leq c$ where c is a small constant value, allocate the entire free area by setting P = F and base address of P as F's base address and adjust the links in free list.

If $D > c$, then allocate space by setting base address of P as F's base address. Modify F's base address as P's base address + P's size and F's size as F's original size - P's size.

3. Find an unused entry in the PDT and record the base address, size of the partition and set the status as allocated.

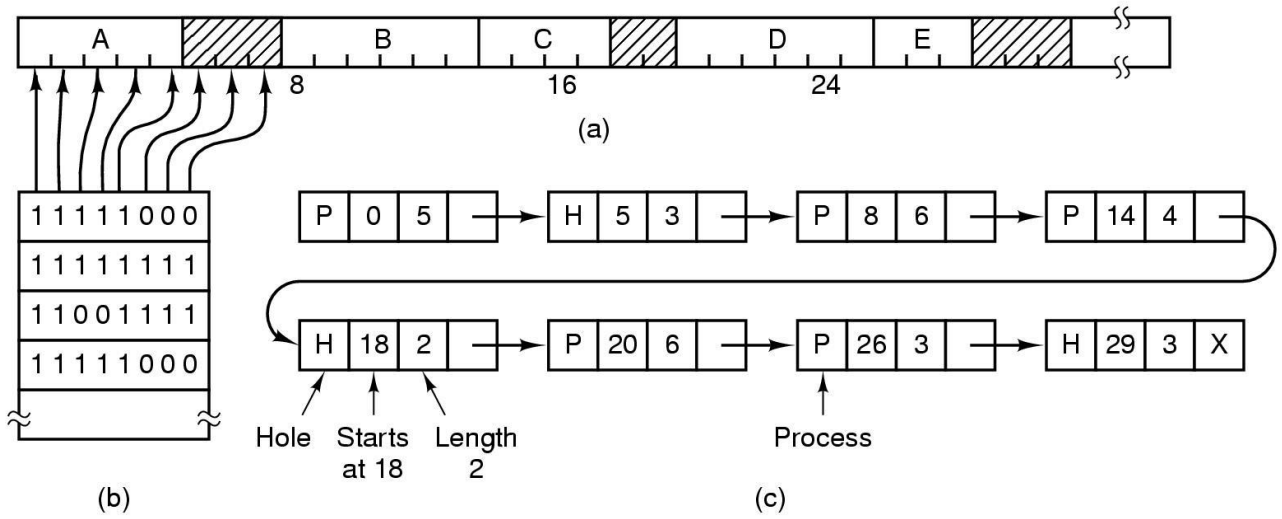
4. Record the PDT's entry number in the process control block of the process T.

Memory Allocation – Mechanism

- MM system maintains data about free and allocated memory alternatives
  - Bit maps – 1 bit per “allocation unit”
  - Linked Lists – free list updated and coalesced when not allocated to a process
- At swap-in or process create
  - Find free memory that is large enough to hold the process
  - Allocate part (or all) of memory to process and mark remainder as free
- Compaction
  - Moving things around so that holes can be consolidated
  - Expensive in OS time

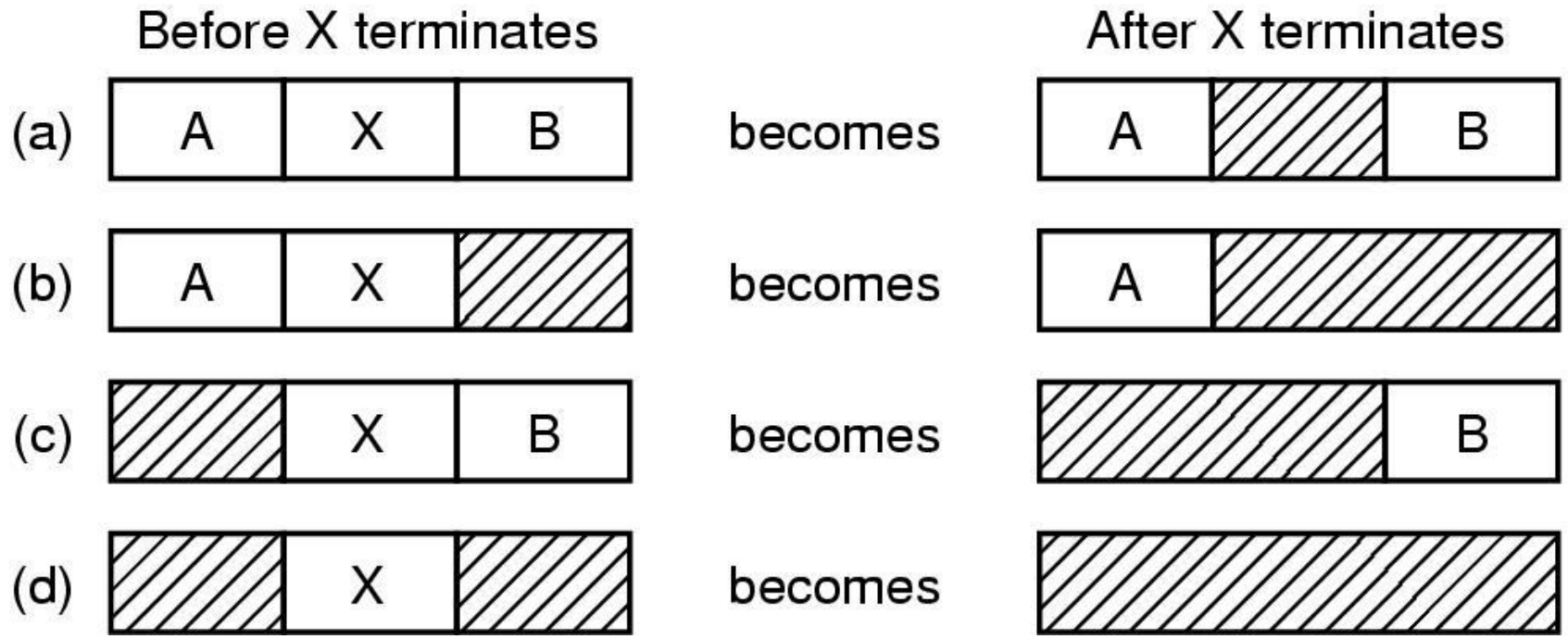
Memory Management with Bit Maps

- Part of memory with 5 processes, 3 holes
  - tick marks show allocation units
  - shaded regions are free
- Corresponding bit map
- Same information as a list



Memory Management with Linked Lists

Four neighbor combinations for the terminating process X



Selection of partitions

- First Fit : Allocate the first free block that is large enough for the new process. This is a fast algorithm.
- Next Fit : A variant of first fit in which the pointer to the free list is saved following an allocation and used to begin the search for the subsequent allocation.
- Best Fit : Allocate the smallest block among those that are large enough for the new process.
- Worst Fit : Allocate the largest block among those that are large enough for the new process. Again a search of the entire list or sorting is needed.

Buddy System

- An allocation-deallocation strategy, called Buddy system, facilitates merging of free space by allocating free areas with an affinity to recombine.
- The size of the partitions are powers of base 2 and the requests for free areas are rounded to the next power of base 2.
- If there is no partition of size of requesting process then a partition of next larger size is split into two buddies to satisfy the request.
- When the block is freed then the block may be recombined with its buddy if it is free.

Compaction

- It is a method to overcome the external fragmentation problem.
- All small free blocks are brought together as one large block of free space. Compaction requires dynamic relocation.
- Selection of an optimal compaction strategy is difficult.
- One method for compaction is swapping out those processes that are to be moved within the memory, and swapping them into different memory locations.

Protection and Sharing

- The mechanisms used are same as that of static partitioning except that dynamic partitioning allows adjacent partitions in physical memory to overlap. Sharing of code must be reentrant or executed in a mutually exclusive fashion with no preemptions.

Advantages:

- All available memory except for the resident portion of the OS may be allocated to a single program.
- It can accommodate processes whose memory requirements are increasing during execution.

Disadvantages:

- It needs complex bookkeeping and memory management algorithms.
- External fragmentation may lead to time penalty for compaction.

Paging Memory management

Noncontiguous Allocation

This means that memory is allocated in such a way that parts of a single logical object may be placed in noncontiguous areas of physical memory.

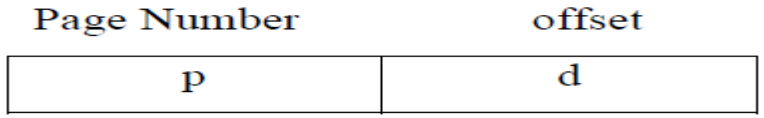
Address translation performed during the execution of instructions establishes the necessary correspondence between a contiguous virtual address space and the noncontiguous physical addresses of locations where object items reside in physical memory at run time.

Paging

- It is a memory management scheme that removes the requirement of contiguous allocation of physical memory.
- The physical memory is conceptually divided into a number of fixed size slots called page *frames*.
- The virtual address space of a process is also split into fixed size blocks of the same size called *pages*.
- The page sizes are usually an integer power of base 2. Allocation of memory consists of finding a sufficient number of unused page frames for loading the requesting process's page.
- An address translation mechanism is used to map virtual pages to their counterparts. Since each page is mapped separately, different page frames allocated to a single process need not occupy contiguous areas of physical memory.
- Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory.

Allocation

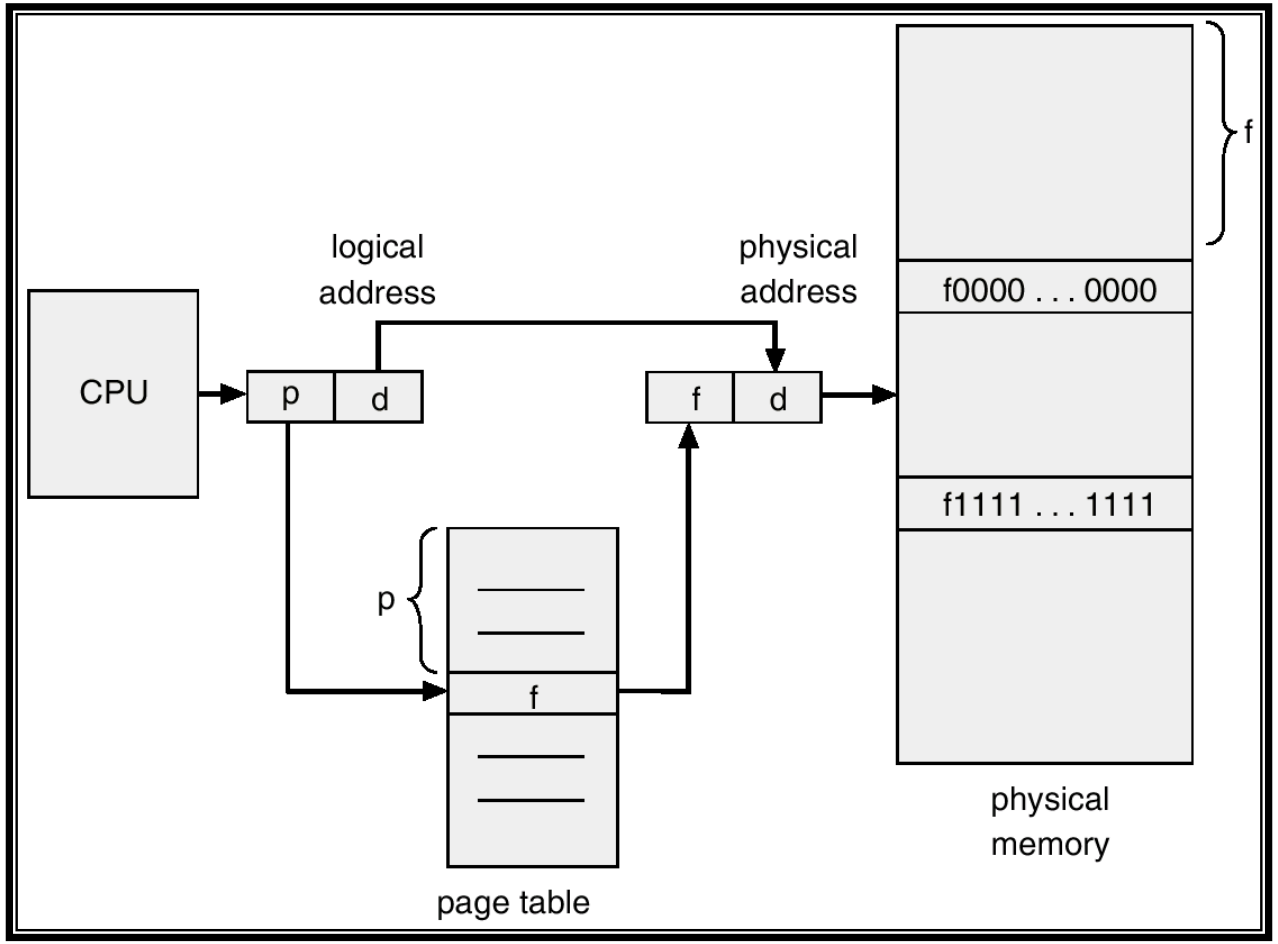
- When a process is to be executed, its corresponding pages are loaded into any available memory frames.
- The mapping of virtual address to physical addresses in paging systems is performed at the page level.
- Each virtual address is divided into two parts: Page Number (Virtual) and the offset within that page.



where p is the index of PT and d is the displacement within the page.

- In paging systems, address translation is performed with the aid of a mapping table called the Page Table (PT).
- There is one PT entry for each virtual page of a process and the higher order bits of the physical base address (Page frame number) is stored in a PT entry.
- The logic of the address translation process in paged system is as follows:
  - The virtual address is split by hardware into the page number and the offset within that page.
  - The page number is used to index the PT and to obtain the corresponding physical frame number.
  - This value is then concatenated with the offset to produce the physical address, which is used to refer the target item in memory.

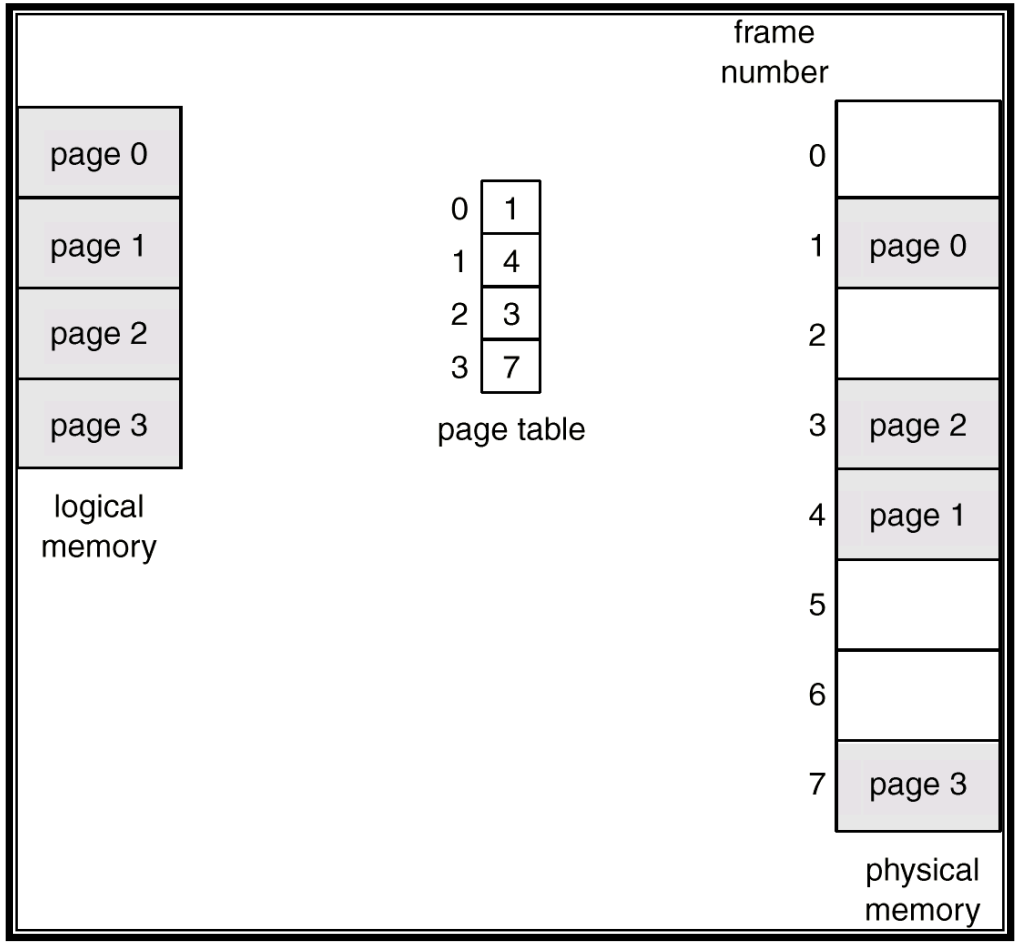
- OS keeps track of the status of each page frame by means of a physical memory map that may be structured as a static table referred as Memory Map Table (MMT). MMT has a fixed number of entries that is identical to the number of page frames in a given system.
- f = m/p where f is the number of page frames, m is the capacity of the installed physical memory and p is the page size.
- When a process of size s requests the OS for allocation, OS must allocate n free page frames so that n = ⌈(s/p)⌉ where p is the page size.
- If the size of a given process is not a multiple of the page size, then the last page frame may be partly unused which is called page fragmentation or page breakage.
- All frames fit all pages and any fit is as good as any other.


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

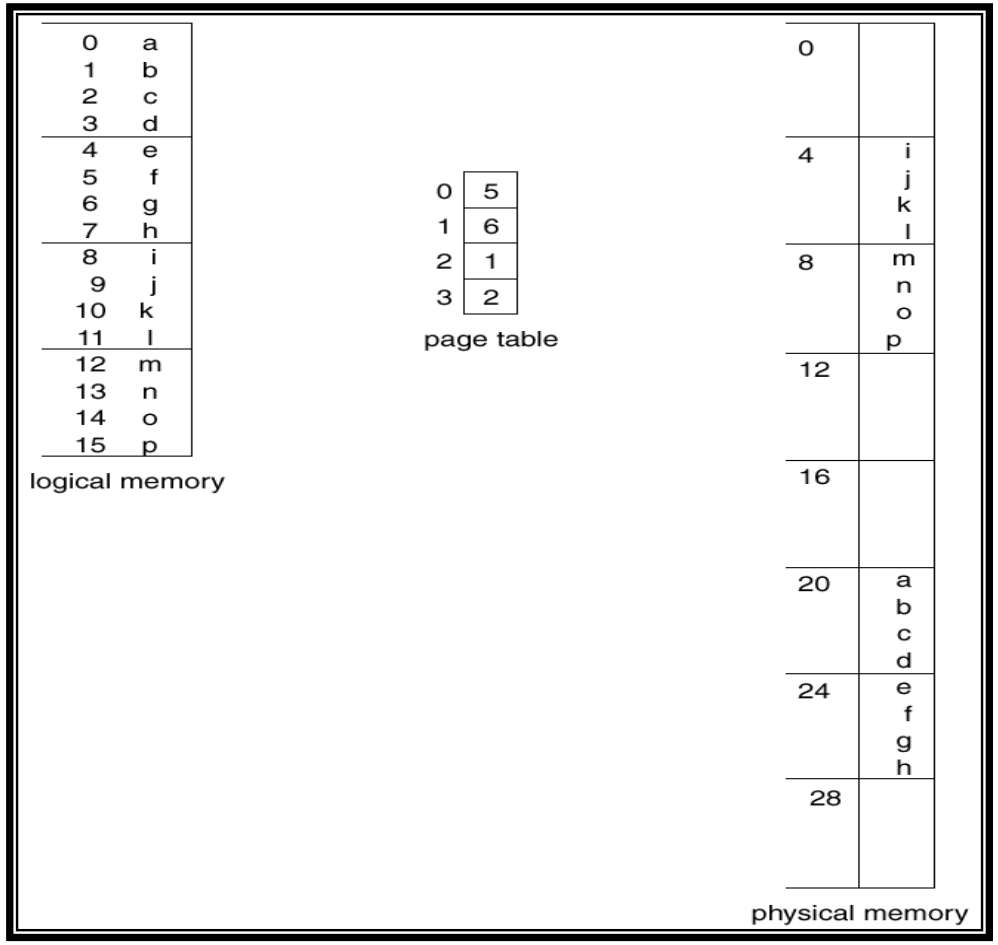
- The efficiency of the memory allocation algorithm depends on the speed with which it can locate free page frames.
- Assuming free frames are randomly distributed in memory, average number of MMT entries x, that needs to be examined in order to find n free frames may be expressed as

  $x = n/q$ where q is the probability that a given frame is available for allocation.
- The number of MMT entries searched x is directly proportional to kn, where $k = 1/q$ and $k \geq 1$.
- If static table form is used then inverse proportion of x to q implies that the number of MMT entries that must be examined in order to locate x frames, which increases with the amount of memory in use.
- An alternative solution is to link the page frames into a free list.

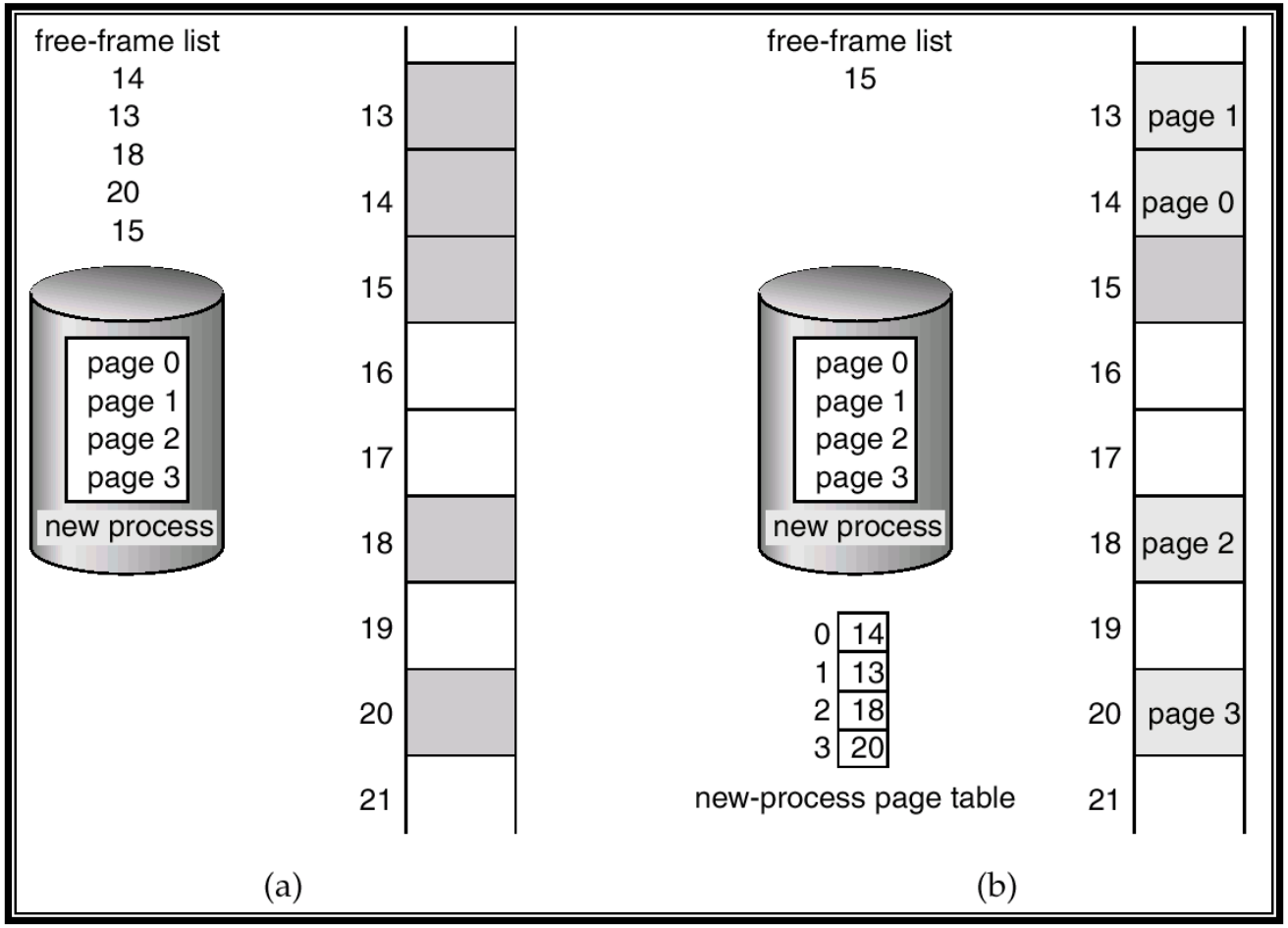
Example


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

Example


0 a
1 b
2 c
3 d
4 e
5 f
6 g
7 h
8 i
9 j
10 k
11 l
12 m
13 n
14 o
15 p
logical memory
0 5
1 6
2 1
3 2
page table
0
4 i j k l
8 m n o p
12
16
20 a b c d
24 e f g h
28
physical memory

Free Frames


free-frame list
14
13
18
20
15
page 0
page 1
page 2
page 3
new process
13
14
15
16
17
18
19
20
21
(a)
free-frame list
15
page 0
page 1
page 2
page 3
new process
0 14
1 13
2 18
3 20
new-process page table
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
(b)

Hardware support for paging

- It concentrates on conserving the memory necessary for storing of the mapping tables and on speeding up the mapping of virtual to physical address.
- Each PT is constructed with only as many entries as its related process has pages.
- Page Table Limit Register (PTLR) is set to the highest virtual page number defined in the PT of the running process.
- Accessing of the PT of the running process may be facilitated by means of the PT base register (PTBR) which points to the base address of the PT of the running process.

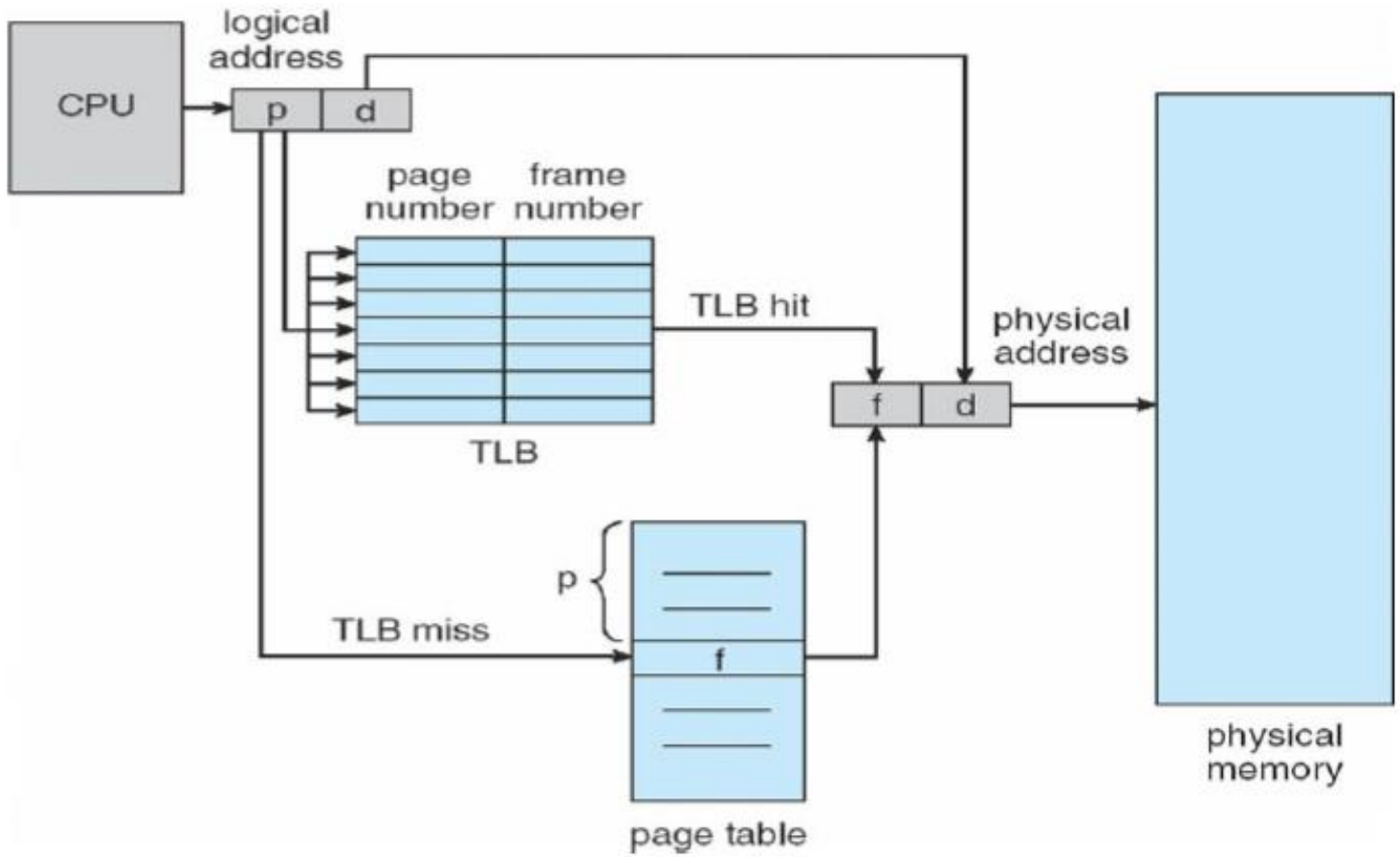
- The values loaded in the register are defined at process loading time and stored in the related PCBs.
- Address translation in paging systems also requires two memory references: One to access the PT for mapping and the other to refer the target item in physical memory.

- One popular approach is to use a high speed associative memory for storing a subset of often used PT entries called as Translation Lookaside Buffer (TLB).
- Address translation begins by presenting the page number portion of the virtual address to the TLB.
- If it is present then the page frame is combined with offset to produce the physical address. The target entry is searched in PT if it is not in TLB.

Associative Memory

Page #	Frame #

- Associative memory – parallel search
- Address translation (A´, A´´)
  - If A´ is in associative register, get frame # out.
  - Otherwise get frame # from page table in memory


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

Effective Access Time

- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers.
- Hit ratio = α
- Effective Access Time (EAT)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$

$$= 2 + \varepsilon - \alpha$$

Structure of PTs

Hierarchical Paging:

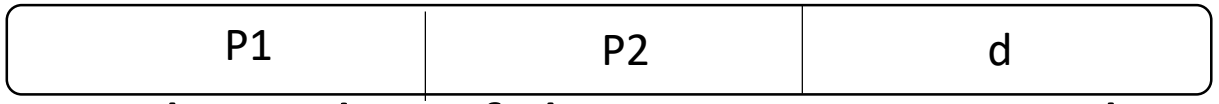
For systems with large logical/virtual address space, the size of PT needed will be in large size.

A PT of large size can be divided to smaller pieces to avoid allocating a large contiguous space.

This can be accomplished in several ways.

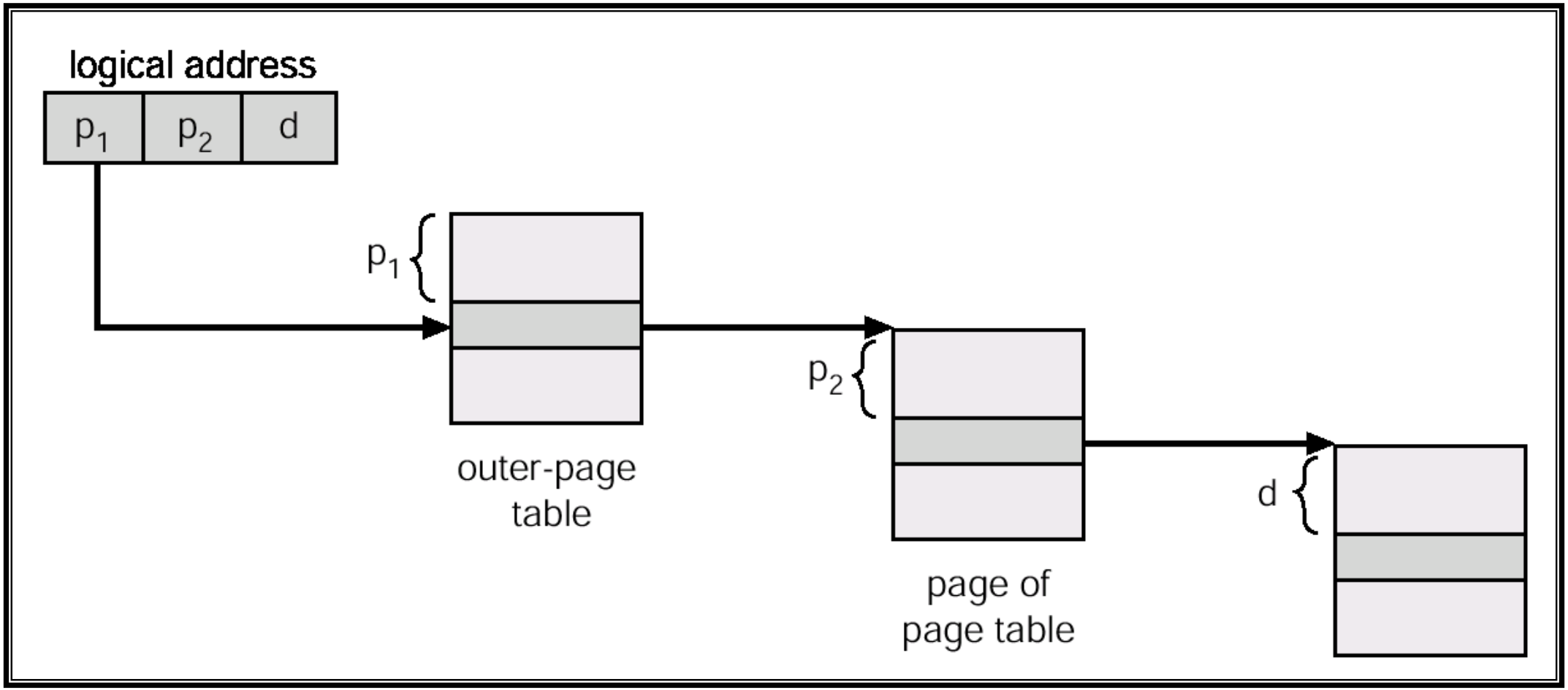
Two level paging

PT itself is treated as pages. The page number component of virtual address is split into two portions one is the index of the outer table and second is the offset for PT.

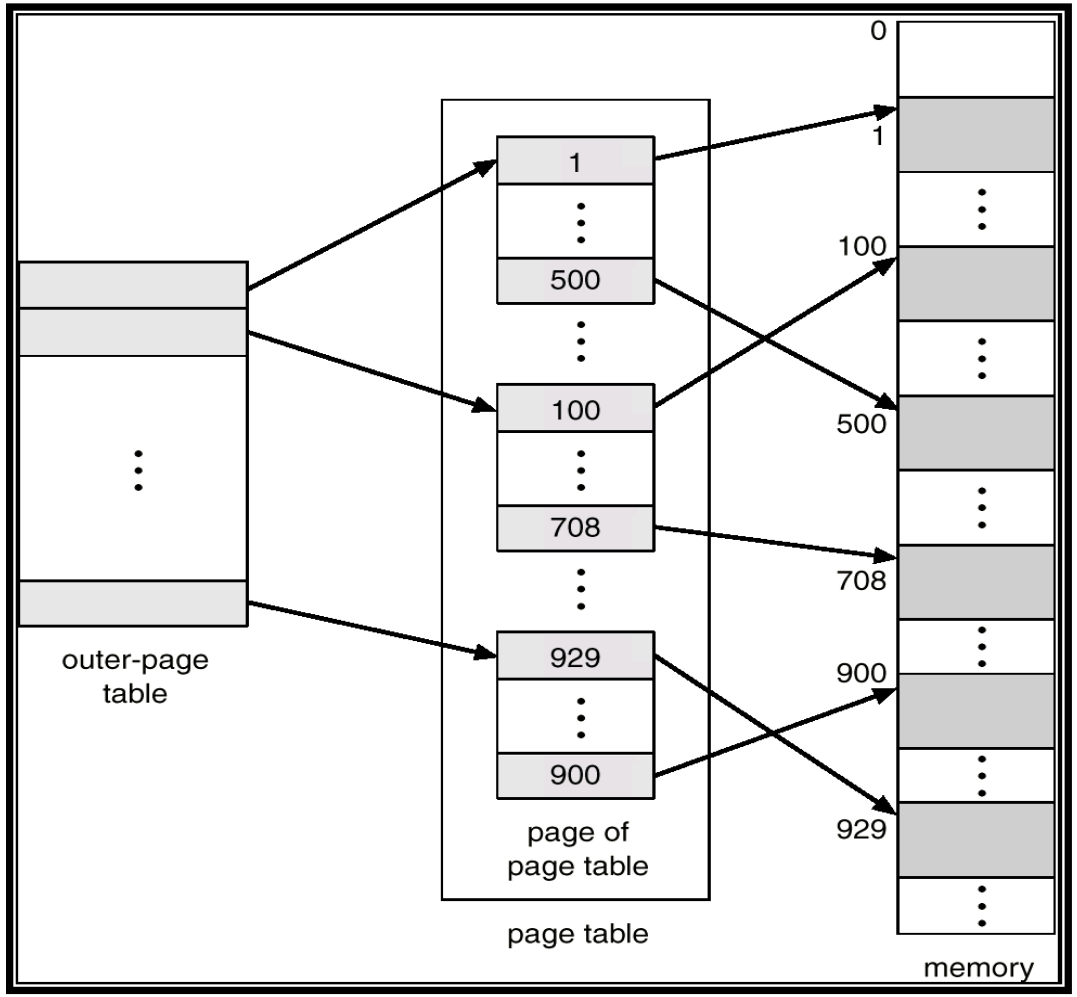


where P1 is the index of the outer PT, P2 is the displacement within the page of the outer page table and d is the displacement within the appropriate page.

Address-Translation Scheme



Two-Level Page-Table Scheme



Variation of Two level Paging:

- The virtual address space of a process is divided into equal sections. The virtual address consists of section number, index to the PT and offset.

For systems with very large virtual address space three level or four level paging can be used.

Hashed Page Tables

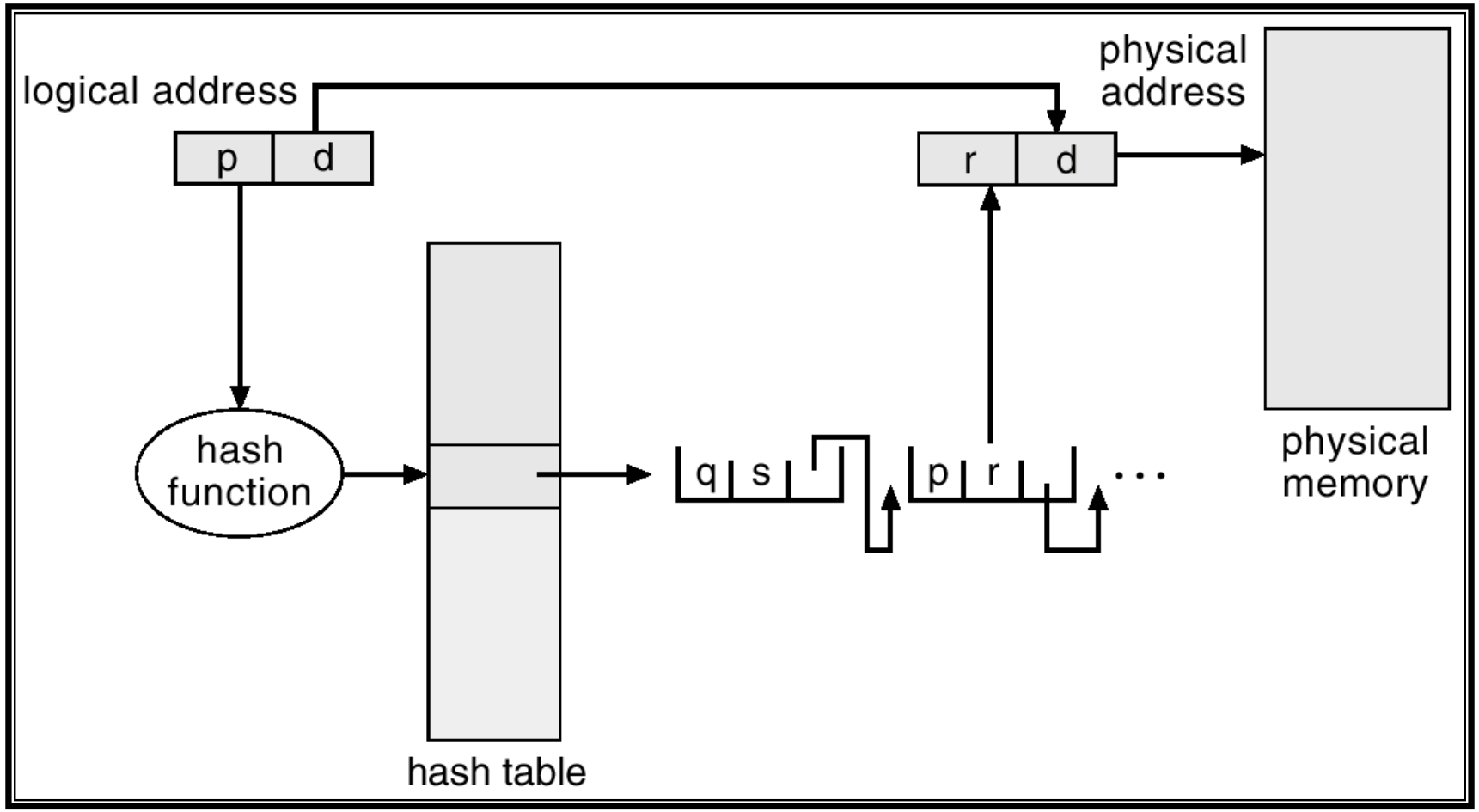
For handling address space larger than 32 bits, a hash page map table is used where the hash value is the virtual page number.

Each entry in the hash table contains a linked list of elements that hash to the same locations.

Each element consists of three fields:

- Virtual Page number
- Value of the mapped page frame
- A pointer to the next element in the linked list.

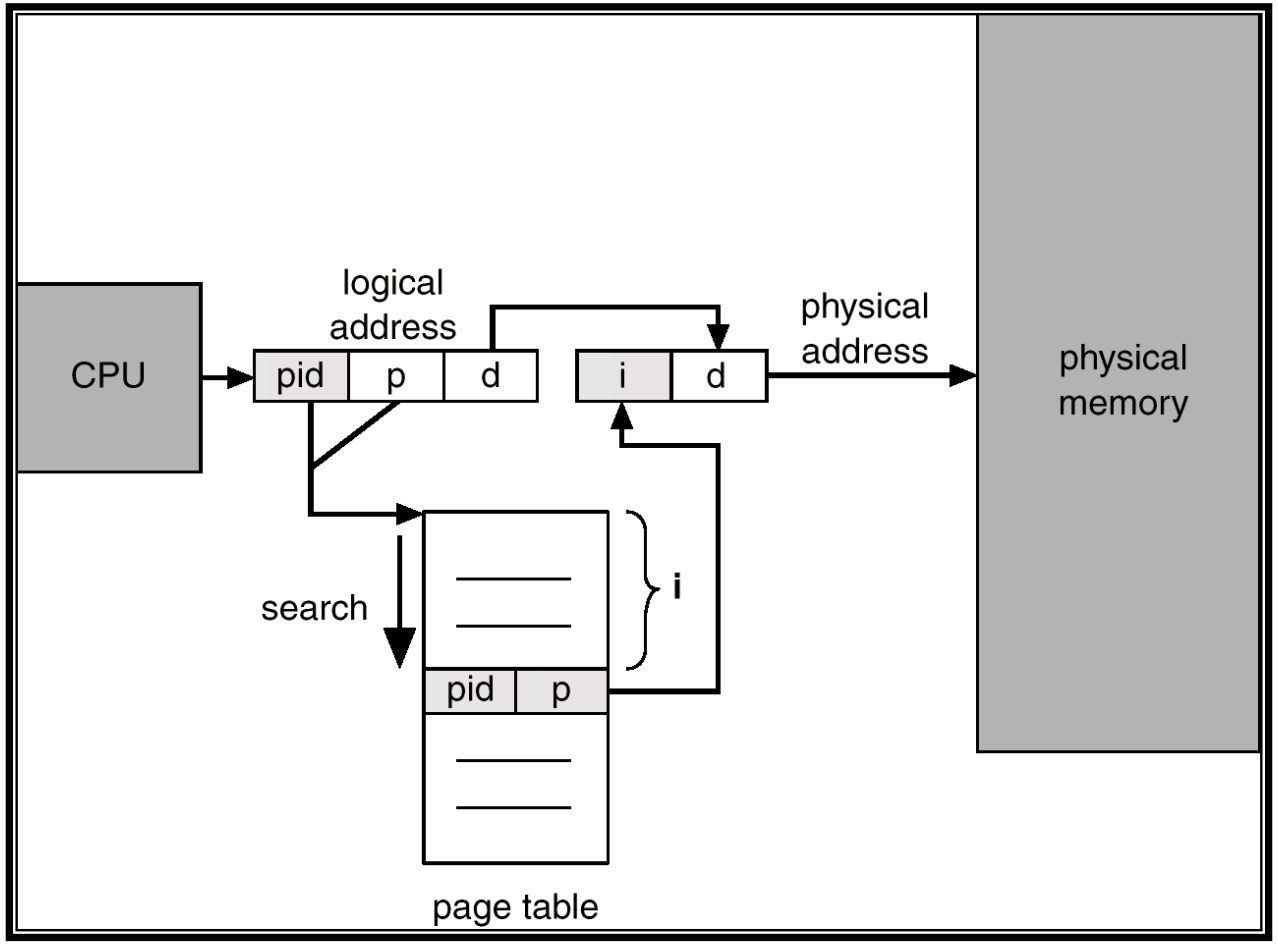
Hashed Page Table



Inverted Page table

- An inverted PT has one entry for each page frame of memory.
- Each entry consists of the virtual address of the page stored in that physical location with information about the process that owns the page.
- Virtual address consists of process id, page number and offset.

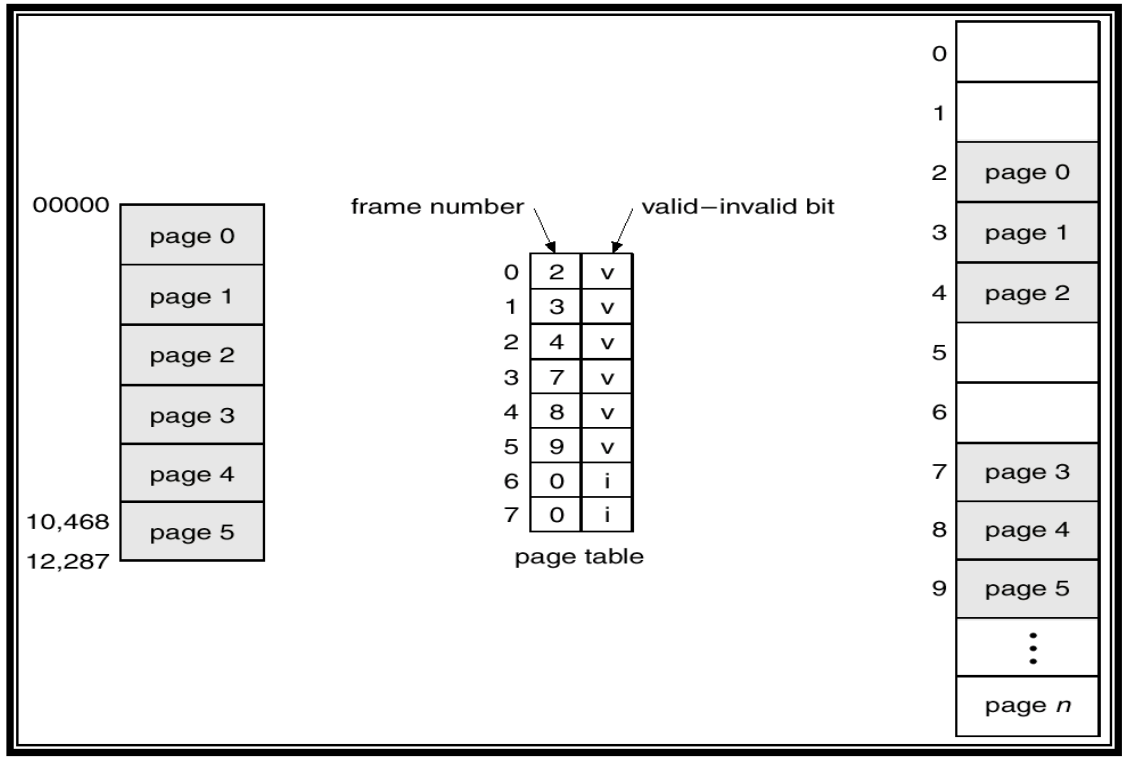
Inverted Page Table Architecture



Protection

- PTLR is used to detect and to abort attempts to access any memory beyond the legal boundaries of a process.
- Memory protection implemented by associating protection bit with each frame. Access bits can be added to PT entries and they can be made transparent to programmers.
- *Valid-invalid* bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page.
  - “invalid” indicates that the page is not in the process’ logical address space.

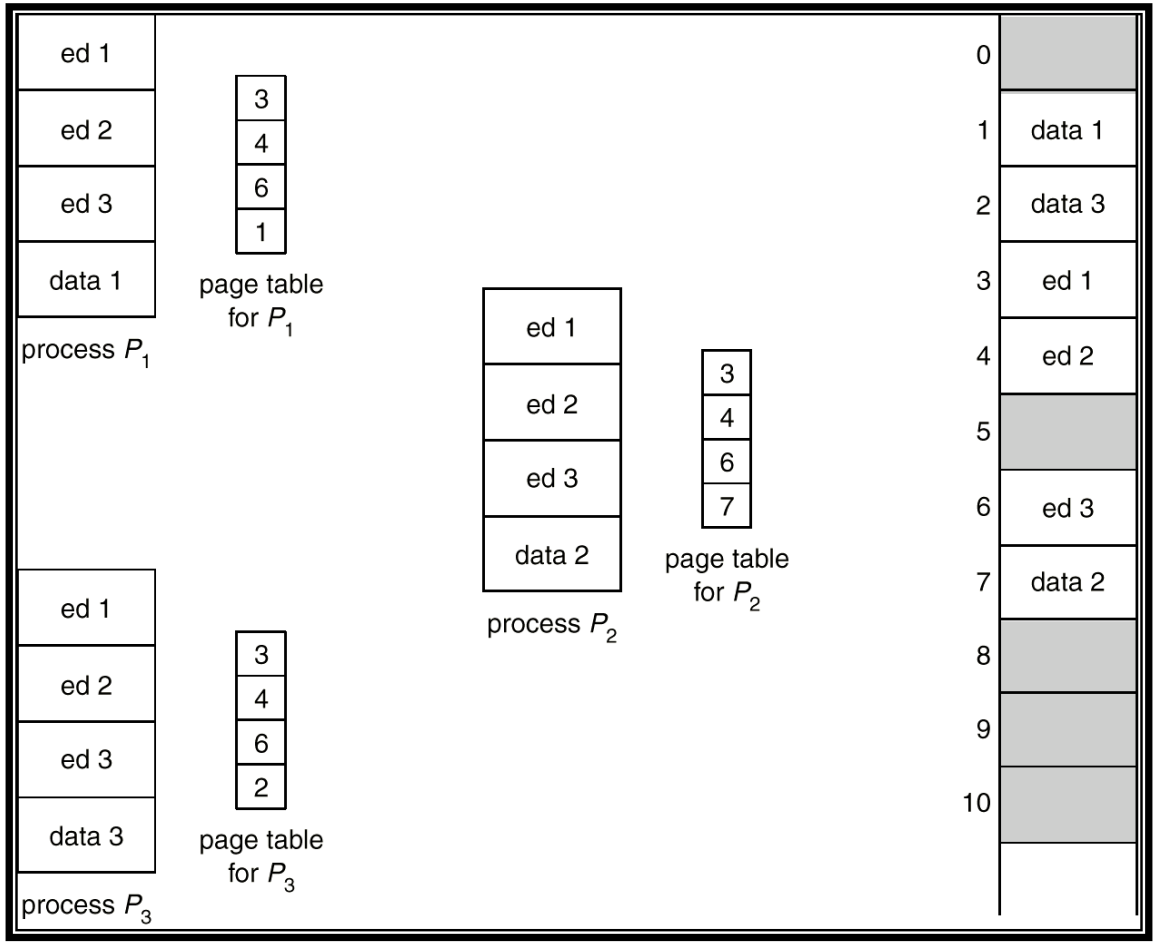
Valid (v) or Invalid (i) Bit In A Page Table



Sharing

- Sharing of pages is straightforward and a single physical copy of a shared page can be easily mapped into as many distinct address spaces as desired.
- Different processes may have different access rights to the shared page.

Shared Pages Example



Advantages:

- Managed entirely by OS and is transparent to programmers
- No compaction is needed thereby it eliminates external fragmentation
- Allocation and deallocation is simple and incurs less overhead
- Memory utilization is very high and is optimal

Disadvantages:

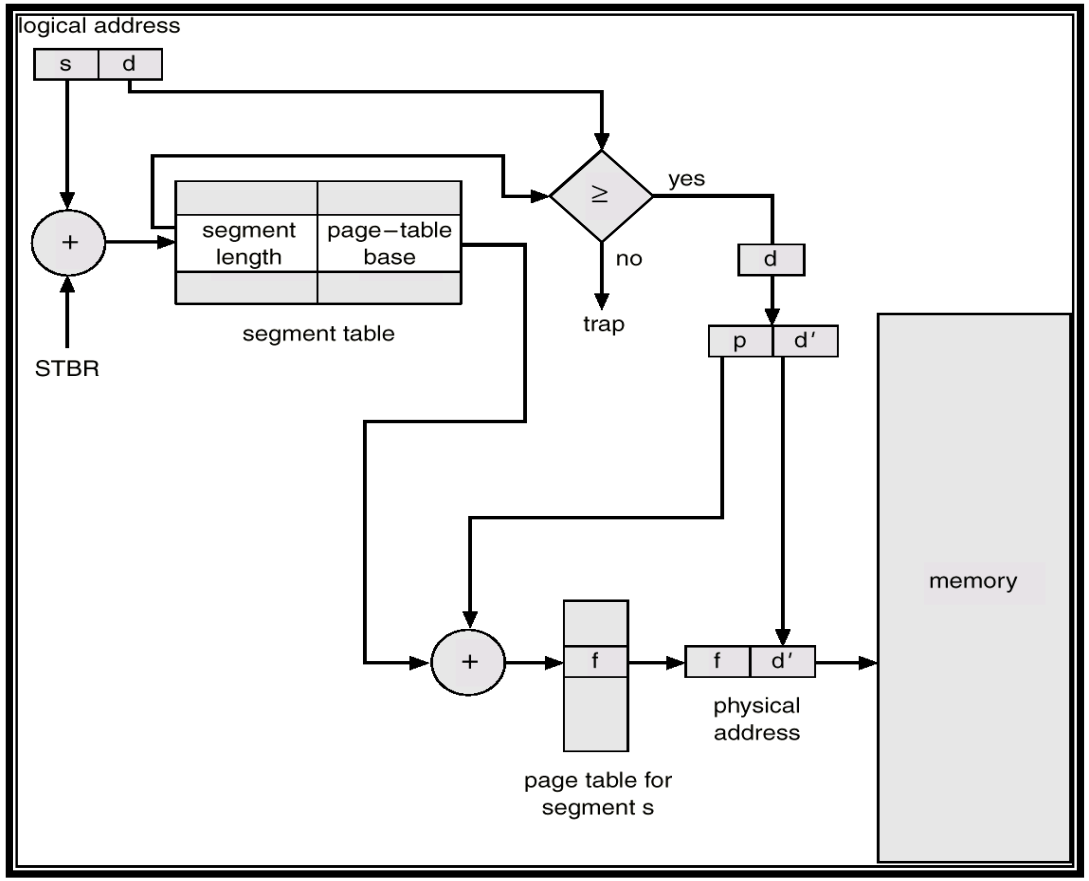
- Storage overhead
- PT per process and MMT lead to page fragmentation
- Table fragmentation is quite large when page size is small
- TLB is expensive and reduces effective memory bandwidth
- Sharing of pages is restrictive

Segmentation with Paging – MULTICS

- The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.

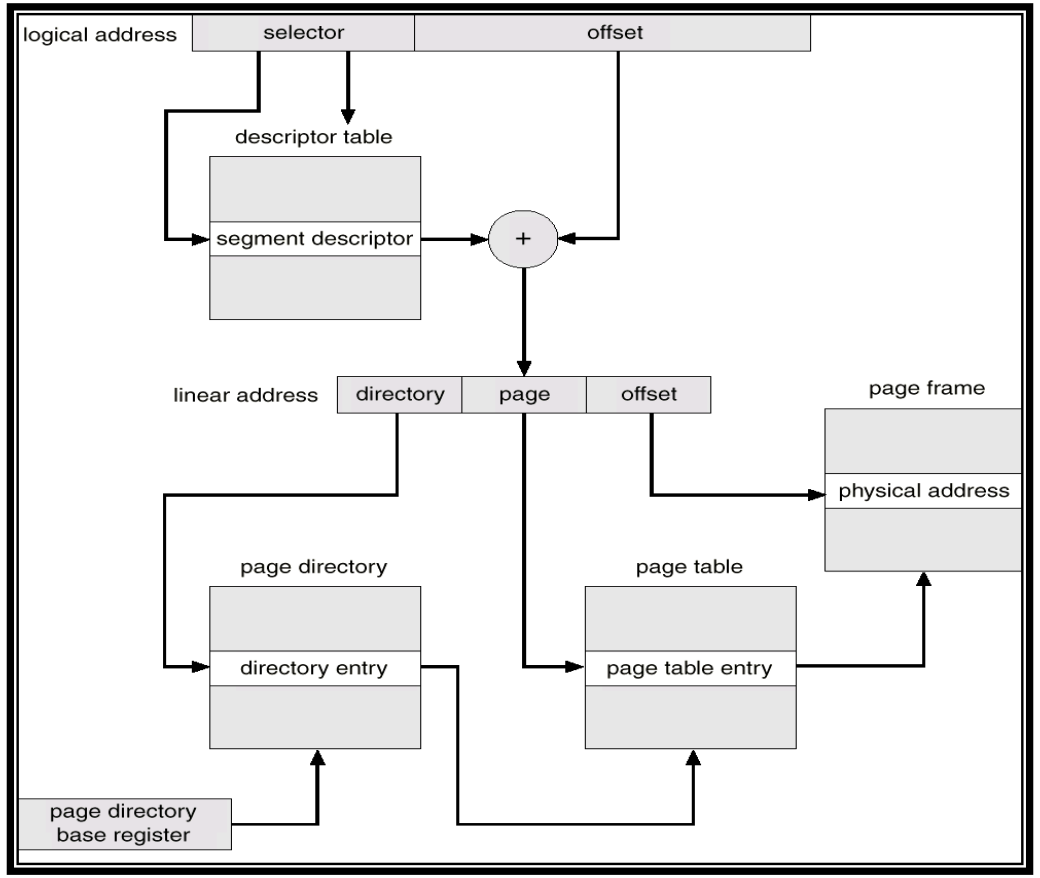
- Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment.

MULTICS Address Translation Scheme



Segmentation with Paging – Intel 386 Address Translation

segmentation with paging for memory management with a two-level paging scheme.

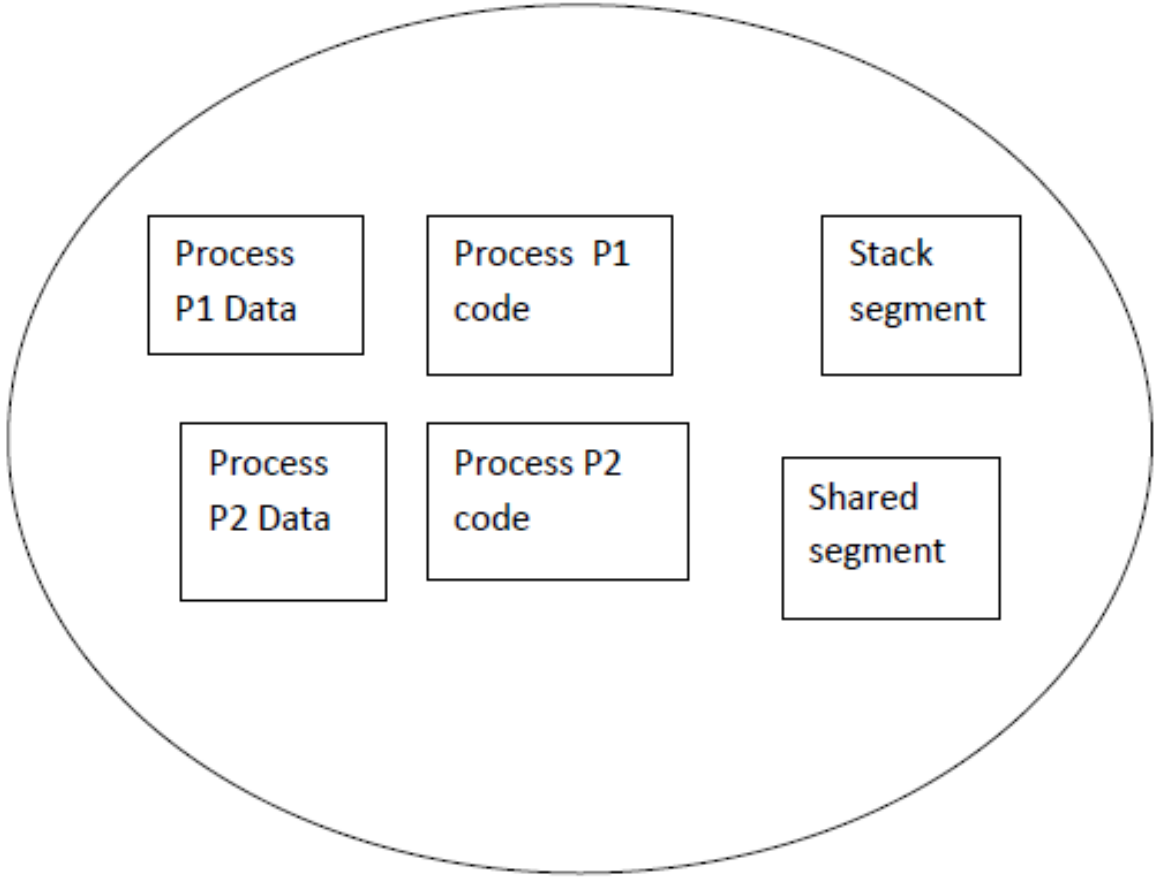


Segmentation Memory Management

Why Segmentation?

- A program can be visualized as a collection of methods, procedures or functions.
- It may also include various data structures such as objects, arrays, stacks, variables etc.
- Users view the program as a collection of modules and data.
- The extent of external and internal fragmentation and their negative impact on wasted memory should be reduced in systems where the average size of a request for allocation memory to segments is small.
- Segmentation is a way to reduce the average size of a request for memory by dividing the process's address space into blocks that may be placed into noncontiguous areas of memory.

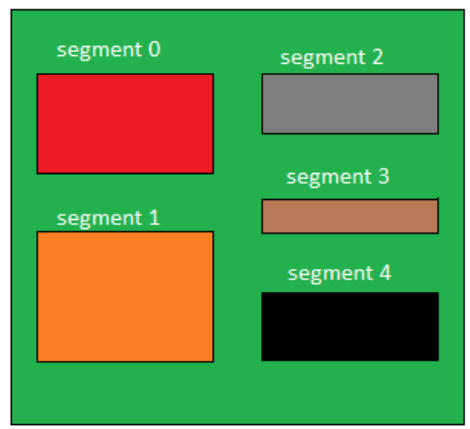
User’s View of a Program



- It is a technique to break memory into logical pieces where each piece represents a group of related information.
- These related information are called segments and are formed at program translation.
- The segments may be data segments, code segments, shared segments and stack segments.
- These segments may be placed in separate noncontiguous areas of physical memory, but the items belonging to a single segment must be placed in contiguous areas of physical memory.
- Thus segmentation possesses some properties of both contiguous (individual segments) and noncontiguous (address space of a process) schemes for memory management.

- These segments are of varying size and thus eliminates internal fragmentation.
- The elements within a segment are identified by their offset from the beginning of the segment. External fragmentation still exists but to lesser extent.

Logical View of Segmentation

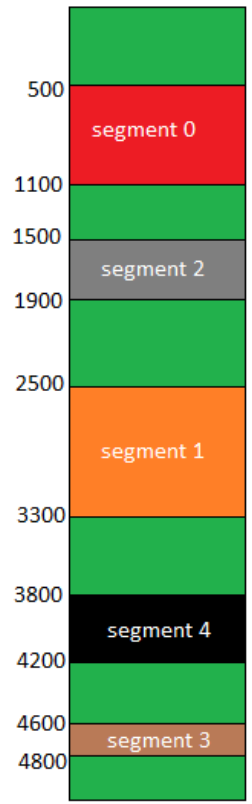


Logical Address Space

Segment Number

	base address	Limit
0	500	600
1	2500	800
2	1500	400
3	4600	200
4	3800	400

Segment Table



Physical Address Space

- Relocation.
  - dynamic
  - by segment table
- Allocation.
  - first fit/best fit
  - external fragmentation

- For relocation purposes, each segment is compiled to begin at its own virtual address 0.
- An individual item within a segment is then identifiable by its offset relative to the beginning of the enclosing segment.
- The unique designation of an item in a segmented address space requires the specification of both its segment and the relative offset.
- Address in segmented systems have two components:

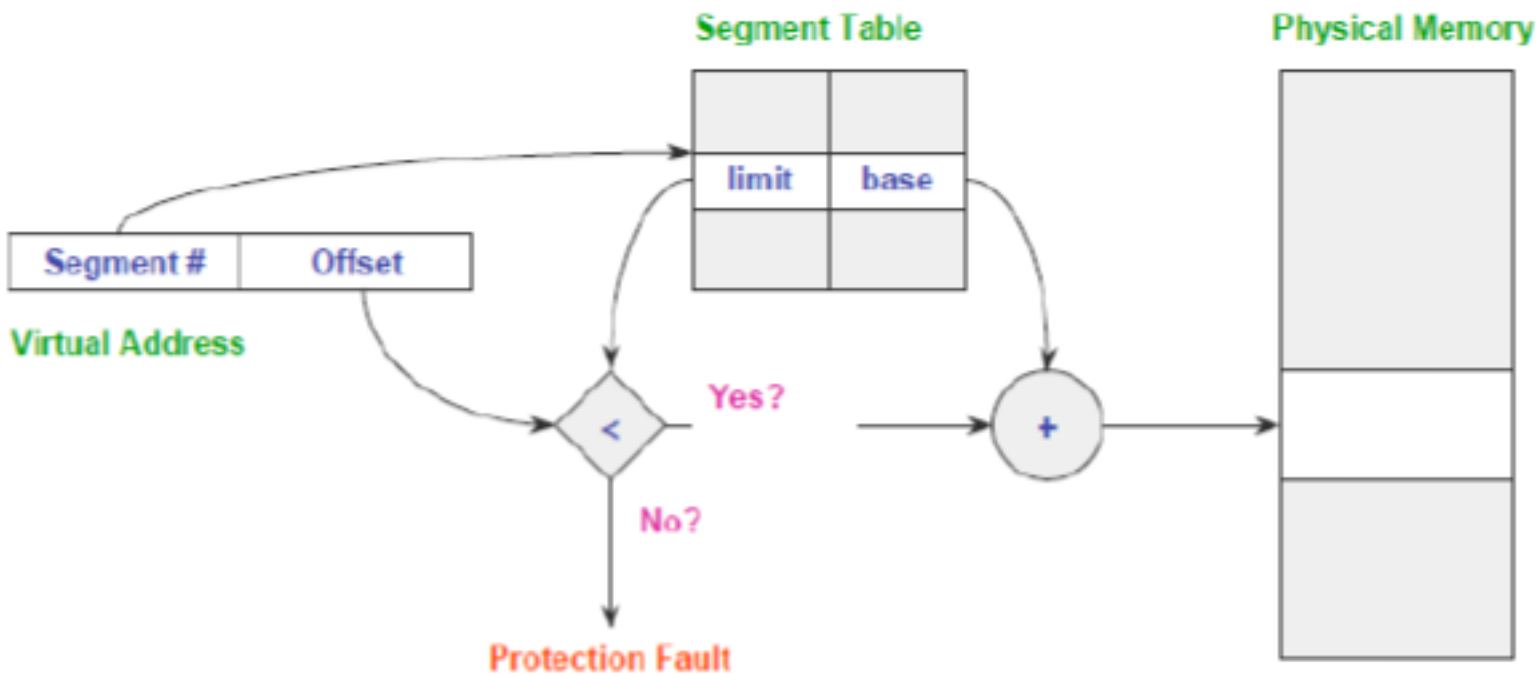
  Virtual Address : Segment name (number) and Offset within the segment.

  Segment number - used to index the SDT and to obtain the physical base address.

  Offset- used to produce physical address by adding with the base value

- When a segmented process is to be loaded onto the memory, OS attempts to allocate memory for all the segments of the process.
- It may create separate partitions to suit the needs of each segment and allocation of partition to segments is similar to dynamic partitioning.
- In segmented systems, an address translation scheme is used to convert a two dimensional virtual segment address into its single dimensional physical equivalent.
- The base (obtained during partition creation) and size (specified in the load module) of loaded segment are recorded as a tuple called the segment descriptor.

- All segment descriptors of a given process are collected in a table called the Segment Descriptor Table (SDT).
- The size of a SDT is related to the size of the virtual address space of a process.
- SDT is treated as a special type of segment. Two registers are used to access the SDT.
  - Segment Descriptor table base register (SDTBR) - points to the base of the running process's SDT
  - Segment Descriptor table limit register (SDTLR)- provided to mark the end of the SDT pointed to by the SDTBR.


Segment Table
Physical Memory
limit
base
Segment #
Offset
Virtual Address
<
Yes?
+
No?
Protection Fault

- Segmentation is multiple base limit version of dynamically partition memory. The price paid for segmenting the address space of a process is the overhead of storing and accessing SDTs.
- Mapping each virtual address requires two physical memory references for a single virtual (Program) reference.
  - memory reference to access the segment description in the SDT.
  - memory reference to access the target item in physical memory.

Segment Descriptor Caching:

- Hardware accelerators are used to speed the translation. Most frequently used segment descriptors are stored in registers.

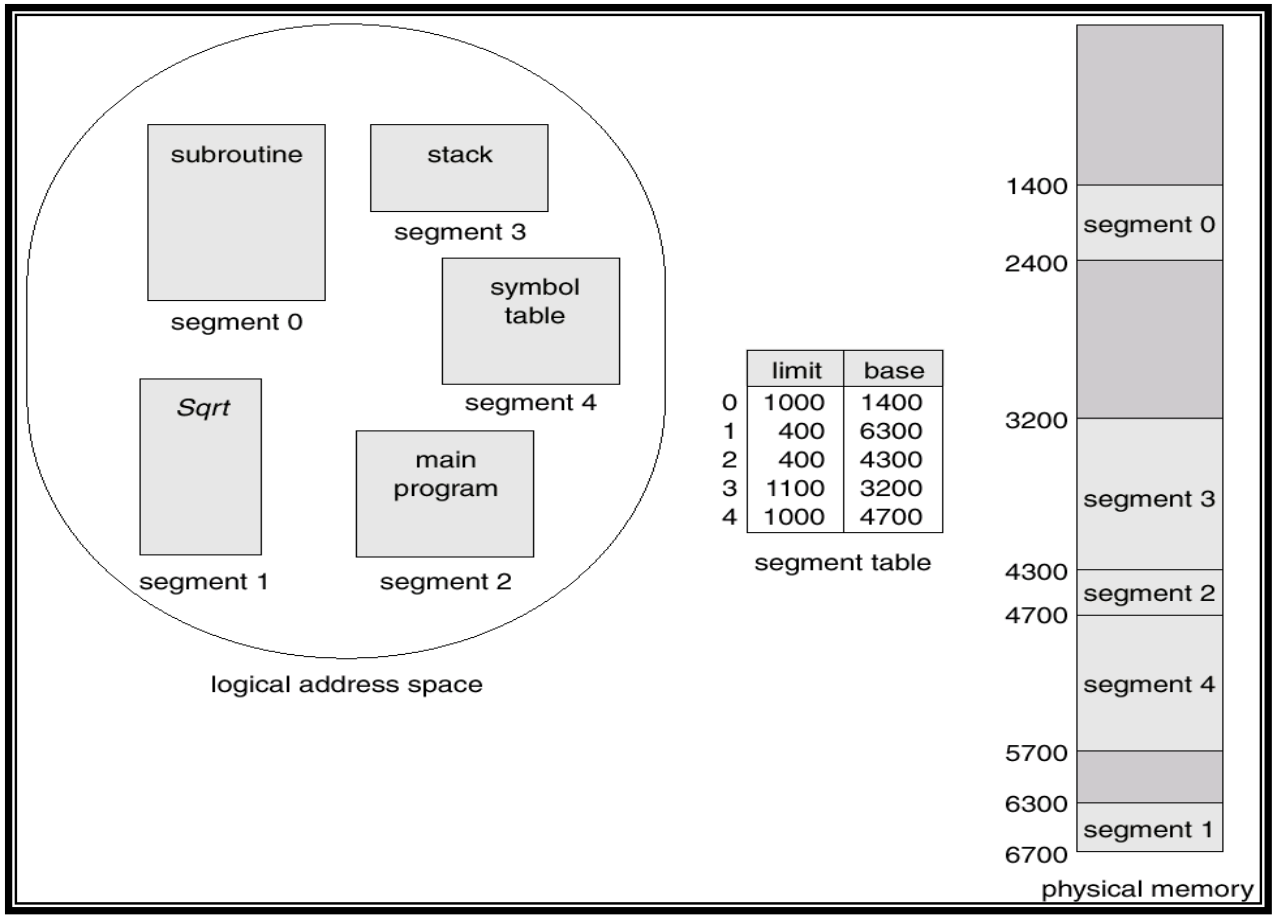
- Protection

The base limit form of protection is used. To provide protection within the address space of a single process, access rights such as read only, write only, execute only can be used depending on the information stored in the segments.

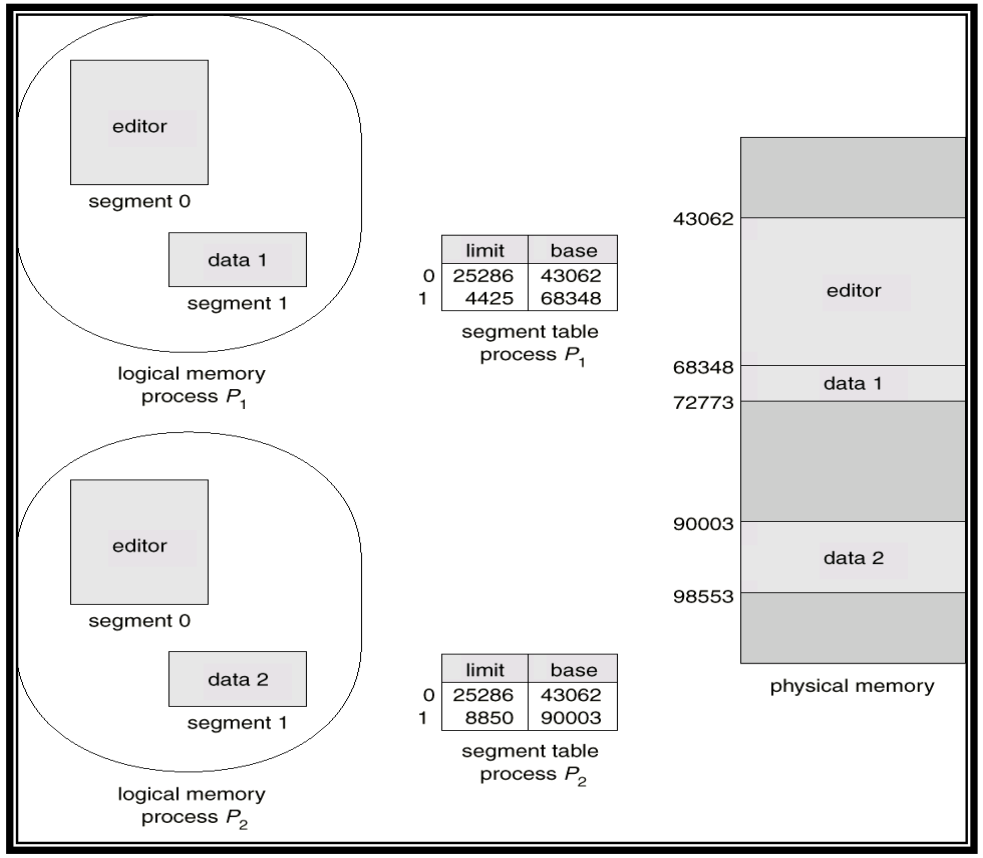
- Sharing

A shared segment may be mapped via the appropriate SDTs to the virtual address spaces of all processes that are authorized to reference it.

Example of Segmentation



Sharing of Segments



- Advantages:
  - Elimination of internal fragmentation
  - Support for dynamic growth of segments
  - Protection and sharing of segments
  - Modular program development
  - Dynamic linking and loading
- Disadvantages:
  - More complex strategies for compaction need to be employed.
  - The two step translation of virtual addresses to physical addresses has to be supported by dedicate hardware to avoid a drastic reduction in the effective memory bandwidth.
  - No single segment may be larger than the available physical memory.
  - Large data structures are split into several segments which results in run time overhead.

Virtual Memory

- It is a memory management scheme where only a portion of the virtual address space of a resident process may actually be loaded into physical memory.
- The sum of the virtual address spaces of active processes in a virtual memory system can exceed the capacity of the available physical memory provided that the physical memory is large enough to hold a minimum amount of the address space of each active process.

- Virtual memory can be accomplished by maintaining an image of the entire virtual address space of a process on secondary storage and by bringing its sections into main memory when needed.
- The choice of which sections to bring in, when to bring in and where to place them is made by OS. The allocation of physical memory is on demand basis.

- Demand paging
  - Do not require all pages of a process in memory
  - Bring in pages as required
- Page fault
  - Required page is not in memory
  - Operating System must swap in required page
  - May need to swap out a page to make space
  - Select page to throw out based on recent history

- We do not need all of a process in memory for it to run
- We can swap in pages as required
- So - we can now run processes that are bigger than total memory available!
- Main memory is called real memory
- User/programmer sees much bigger memory - virtual memory

Illusion of larger memory has many advantages.

1. Programmers are practically relieved of the burden of trying to fit a program into limited memory.

2. Some program may run without reprogramming or recompilation on systems with significantly different capacities of installed memory.

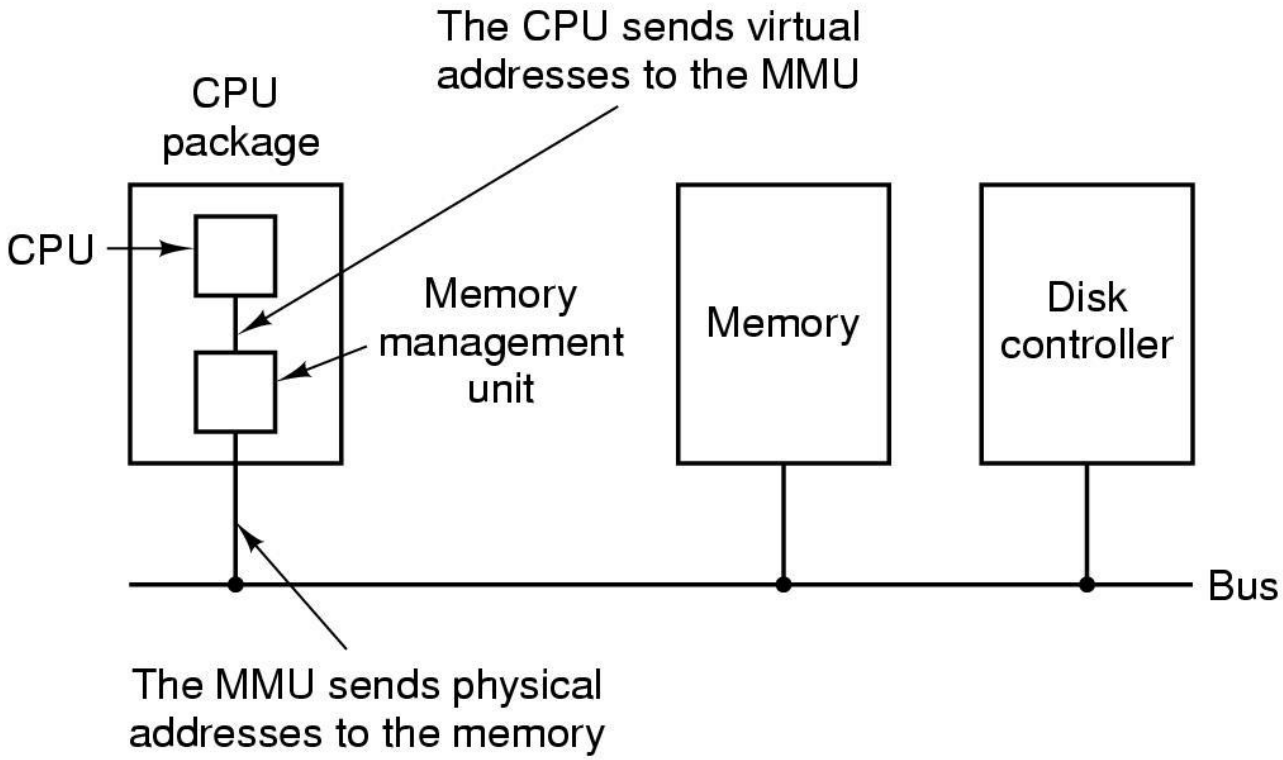
3. A process may be loaded into a space of arbitrary size. This may be used to reduce external fragmentation without the need to change the scheduled order of process execution.

4. OS can speed up the execution of important programs by allocating more memory.

5. Degree of multiprogramming can be increased by reducing the real memory for the resident processes.

The maximum program execution speed of a virtual memory system can be equal but never exceed the execution speed of the same program with virtual memory turned off.

Virtual Memory



The position and function of the MMU

Virtual memory can be implemented as an execution of paged or segmented memory management or as a combination of both.

Accordingly address translation is performed by means of PT, SDT or both.

Process of Address Mapping:

Assume Virtual memory is implemented using paging memory management.

Let Virtual address space be V = | 0,1, ...V-1| and Physical address space be M = |0,1,.. M-1|

At any time the mapping hardware must realize the function f : V-> M such that

f(x) = r if item x is in physical memory at location r else missing item exception if item x is not in physical memory.

- The detection of missing item is done by checking the presence indicator bit of each entry of PTs. Before loading the process OS clears all presence bits in the related PT.
- When pages are moved to main memory on demand, corresponding presence bits are set. When a page is evicted from main memory its presence bit is reset.
- The address translation hardware checks the presence bit during the mapping of each memory reference. If the bit is set, the mapping is completed as in paging scheme.

- If the bit is not set then the hardware generates a missing item exception which is referred as page fault in paging systems.
- When a running process experiences a page fault then the process is suspended until the missing page is brought to the main memory.
- Since the disk access time is longer than memory access time, another process is scheduled by OS.
- The disk address of the faulted page is provided by the File Map table (FMT). FMT contains secondary storage addresses of all pages.
- One FMT is maintained for each active process and has the number of entries identical to PT of the process. OS uses the virtual page number to index the FMT and to obtain the related disk address.

- Virtual memory needs more hardware provisions compared to paging and segmentation.
- One instruction may require several pages.
- For example, a block move of data. In some cases page fault may occur during part way through an operation and may have to undo what was done.
- Example: an instruction crosses a page boundary.

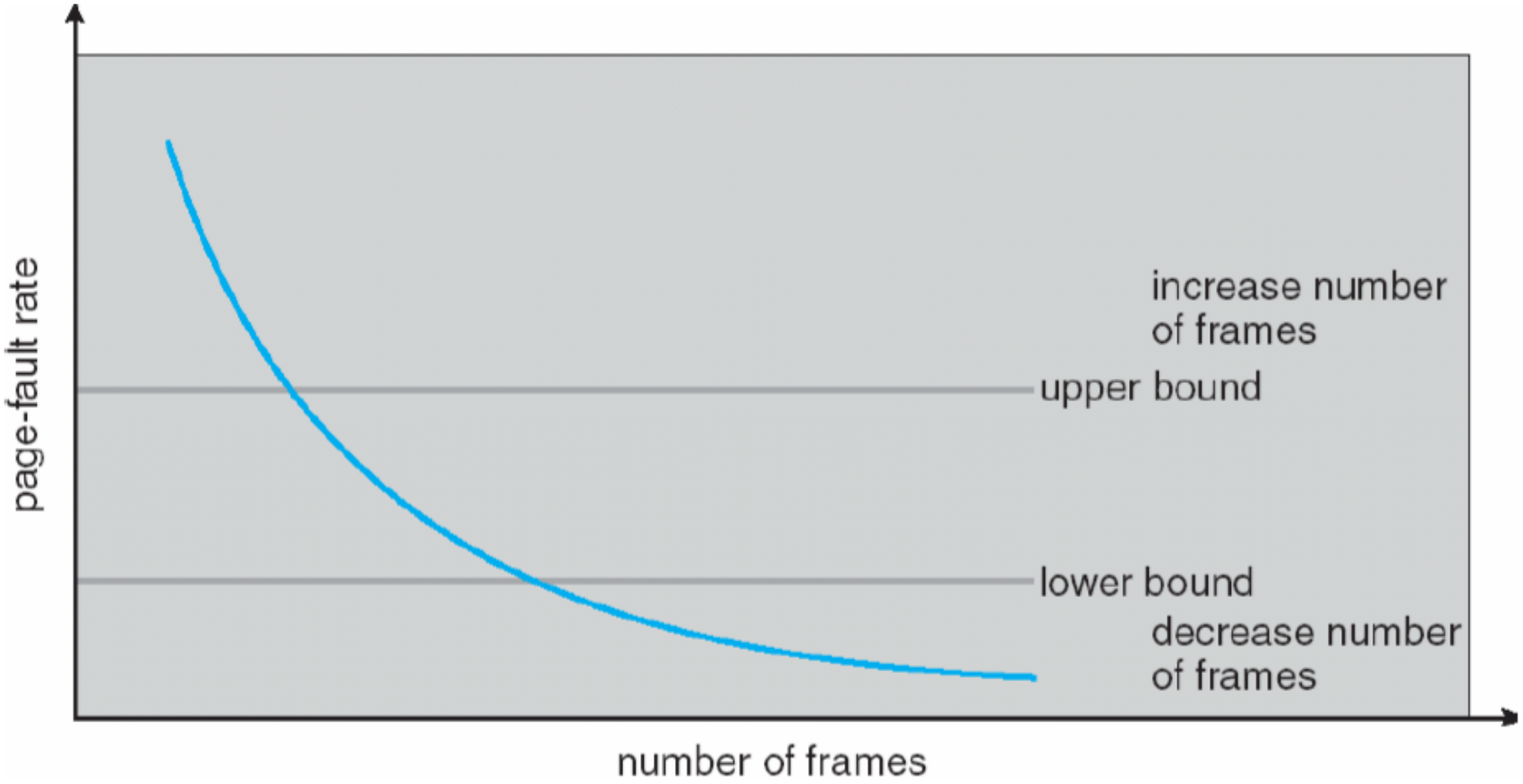
Management of Virtual Memory

- The allocation of subset of page frames to the virtual address space of a process requires the incorporation of certain policies into the virtual memory manager.
- Allocation policy: How much real memory to allocate to each process?
- Fetch Policy: What are the items to be brought and when to bring them?
- Placement policy: Where to place the incoming item?
- Replacement Policy: If there is no free memory, which item is to be evicted so that the new item can be stored in the real memory.

Allocation Policy

- OS should determine how many page frames should be allocated to a process.
- If more page frames are allocated then the advantages are reduced page fault frequency and improved turnaround time.
- If too few pages are allocated to a process, then page fault frequency and turnaround times may deteriorate to unacceptable levels and also in systems supporting restarting of faulted instructions page fault frequency may increase.
- The allocation module may fix two thresholds lower and upper for each process. The actual page fault rate experienced by a running process may be measured and recorded in PCB.
- When a process exceeds the upper threshold then more page frames will be allocated. If the process page fault rate reaches the lower threshold then allocation of new page frames may be stopped

Page fault rate vs allocation of page frames



- The fetch policy is mostly on demand and *Demand paging* is the most common virtual memory management system.
- It is based on the locality model of program execution. As a program executes, it moves from locality to locality.
- *Locality* is defined as a set of pages actively used together. A program is composed of several different localities which may overlap.
- For example, a small procedure when called, defines a new locality. A while-do loop when being executed defines a new locality.

- The Placement policy follows the underlying memory management scheme i.e. paging or segmentation.

Replacement Policy

- A page replacement algorithm *determines how the victim page (the page to be replaced) is selected when a page fault occurs*. find some page in memory, but not really in use, swap it out.
- There is a possibility that same page may be brought into memory several times. The aim is *to minimize the page fault rate*.

Steps in handling page faults including page replacement are:

1. Check the PT and FMT of the process, to determine whether the reference is a valid or an invalid memory access.

2. If the reference is invalid then the process is terminated; if it is valid and page have not yet brought in, find the location of the desired page on the disk.

3. Find a free frame. If there is a free frame use it. Otherwise, use a page-replacement algorithm to select a victim frame. Write the victim page to the disk; change the PT and MMT accordingly.

4. Schedule a disk operation to read the desired page into the newly allocated frame.

5. When the disk read is complete, modify the FMT, PT and MMT to indicate that the page is now in memory.

6 Restart the user process.

Page Replacement Algorithms

- Page fault forces choice
  - which page must be removed
  - make room for incoming page
- Modified page must first be saved
  - unmodified just overwritten
- Better not to choose an often used page
  - will probably need to be brought back in soon

- The efficiency of a page replacement algorithm is evaluated by running it on a particular string of memory references and computing the number of page faults.
- Reference strings are either generated randomly, or by tracing the paging behaviour of a system and recording the page number for each logical memory reference.
- The performance of a page replacement algorithm is evaluated by running it on a particular string of memory references and computing the number of page faults.

First-In-First-Out (FIFO)

- A FIFO algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. It is easy to implement.
- A FIFO queue/ list is used to hold all pages in memory. The page at the head of the queue is replaced. When a page is brought into memory, it is inserted at the tail of the queue.
- FIFO algorithm is easy to understand and to program.
- Drawback is Belady's Anomaly - When the number of page frames allotted increases the page fault also increases for some cases.

Example: FIFO

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream: A B C A B D A D B C B

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

- FIFO: 7 faults.
- When referencing D, replacing A is bad choice, since need A again right away

Modeling Page Replacement Algorithms Belady's Anomaly

All pages frames initially empty

	0	1	2	3	0	1	4	0	1	2	3	4
Youngest page	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
Oldest page			0	1	2	3	0	0	0	1	4	4
	P	P	P	P	P	P	P			P	P	

9 Page faults

(a)

	0	1	2	3	0	1	4	0	1	2	3	4
Youngest page	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
Oldest page			0	1	1	1	2	3	4	0	1	2
				0	0	0	1	2	3	4	0	1
	P	P	P	P			P	P	P	P	P	P

10 Page faults

(b)

- FIFO with 3 page frames
- FIFO with 4 page frames
- *P*'s show which page references show page faults

Least Recently Used (LRU)

- Page which has not been used for the longest time in main memory is the one which will be selected for replacement. Easy to implement, keep a list, replace pages by looking back into time.
- The OS using this method, has to associate with each page, the time it was last used which means some extra storage.
- In the simplest way, the OS sets the reference bit of a page to "1" when it is referenced. This bit will not give the order of use but it will simply tell whether the corresponding frame is referenced recently or not. The OS resets all reference bits periodically.

LRU Algorithm Implementations

Counter implementation:

- A time-of-use field is associated with each page-table entry, and a logical clock or counter is added to the CPU.
- The clock is incremented for every memory reference. Whenever a reference to a page is made, the content of the clock register is copied to the time-of-use field in the page table for that page.
- The page with the smallest time value is replaced. It requires a search of the page table to find LRU page and a write to memory for each memory access.

Stack implementation:

- A stack is used to store page numbers. Whenever a page is referenced, it is removed from the stack and put on the top.
- Top of the stack is always the mostly recently used page and the bottom is the LRU page.
- It is implemented by a double linked list with a head and tail pointer because entries must be removed from the middle of the stack. The tail pointer points to the bottom of the stack (which is LRU page).

Consider the page reference string A B C B B A D A B F B. If LRU page replacement is followed and the number of page frames allocated is 3, find the total number of page faults.

Total number of page faults = 5

A	B	C	B	B	A	D	A	B	F	B
A	A	A	A	A	A	A	A	A	A	A
	B	B	B	B	B	B	B	B	B	B
		C	C	C	C	D	D	D	F	F
*	*	*				*			*	

A	B	C	B	B	A	D	A	B	F	B
A	B	C	B	B	A	D	A	B	F	B
	A	B	C	C	B	A	D	A	B	F
		A	A	A	C	B	B	D	A	A

Optimal Page Replacement Algorithm

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. It has been called OPT or MIN.
- Replace the page that will not be used for the longest period of time. (Optimal but unrealizable)
- In this algorithm, the victim is the page which will not be used for the longest period. Estimate by logging page - use on previous runs of process although this is impractical
- It will never suffer from Belady’s anomaly.
- OPT is not possible to be implemented in practice because it requires future knowledge. However, it is used for performance comparison.

Page Buffering algorithm

- The replaced pages will first go to one of two page lists in main memory, depending whether it has been modified or not.
- The advantage of this scheme is that if in a short time, the page is referenced again, it may be obtained with little overhead.
- When the length of the lists surpasses a specific limit, some of the pages that went into the lists earlier will be removed and written back to secondary memory. Thus the two lists actually act as a cache of pages.
- Another advantage is the modified pages may be written out in cluster rather than one at a time, which significantly reduces the number of I/O operations and therefore the amount of time for disk access pool.

Counting Algorithms

It keeps a counter of the number of references that have been made to each page. Two schemes use this concept.

1. Least Frequently Used (LFU) Algorithm:

    Page with the smallest count is the one which will be selected for replacement.

This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again

2. Most Frequently Used (MFU) Algorithm:

    It is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Second-Chance Algorithm/ Clock Algorithm

- The clock/ second chance policy is basically a variant of the FIFO policy, except that it also considers to some extent the last accessed times of pages.
- When a page has been selected, its reference bit is inspected. If the value is 0 replace the page. If the value is 1, then the page is given a second chance and move on to select the next FIFO page.

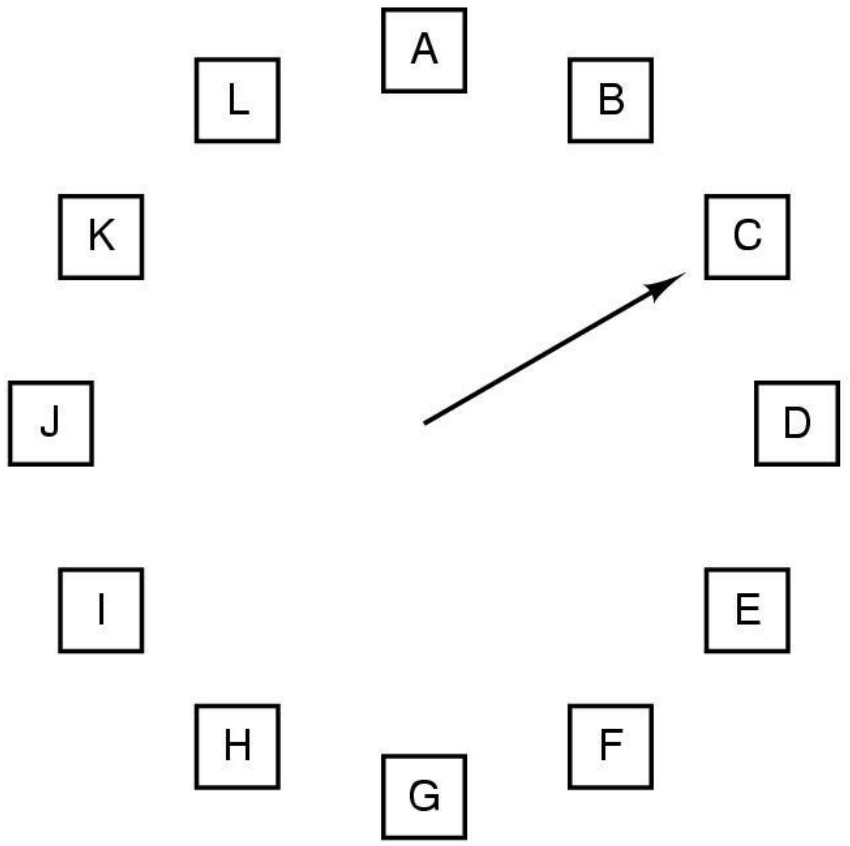
When a page gets a second chance:

1. Its reference bit is cleared.

2. Its arrival time is reset to the current time.

3. It will not be replaced until all other pages are replaced or given second chance.

The Clock Page Replacement Algorithm



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

Enhanced Second-Chance Algorithm (Not Recently Used)

Each page has Reference bit, Modified bit : bits are set when page is referenced, modified. It uses the reference bit and the modify bit as an ordered pair.

With 2-bits, four classes are possible:

1. (0,0)– neither recently used nor modified (best page to replace).
2. (0,1)– not recently used but modified the page will need to be written out before replacement.
3. (1,0)– recently used but clean (probably will be used again).
4. (1,1)– recently used and modified probably will be used again and write out will be needed before replacing it.

The steps of the algorithm are as follows:

1. Scan the frame buffer and select the first frame with (0, 0) if any.

2. If step 1 fails, scan again, look for a frame with (0,1) and select it for replacement if such a frame exists. During the scan, the reference bits of 1 are set to 0.

3. If step 2 fails, all the reference bits in the buffer are 0. Repeat step 1 and if necessary step 2. This time, a frame will be definitely chosen to be replaced.

The recently accessed pages are given higher priority than those that have been modified, based on the consideration that though the latter need to be written back to secondary memory first before replacement, they still involve less overhead than probable reloading a recently accessed but replaced page.

Frame Allocation

In order to decide on the page replacement scheme of a particular reference string, the number of page frames available should be known.

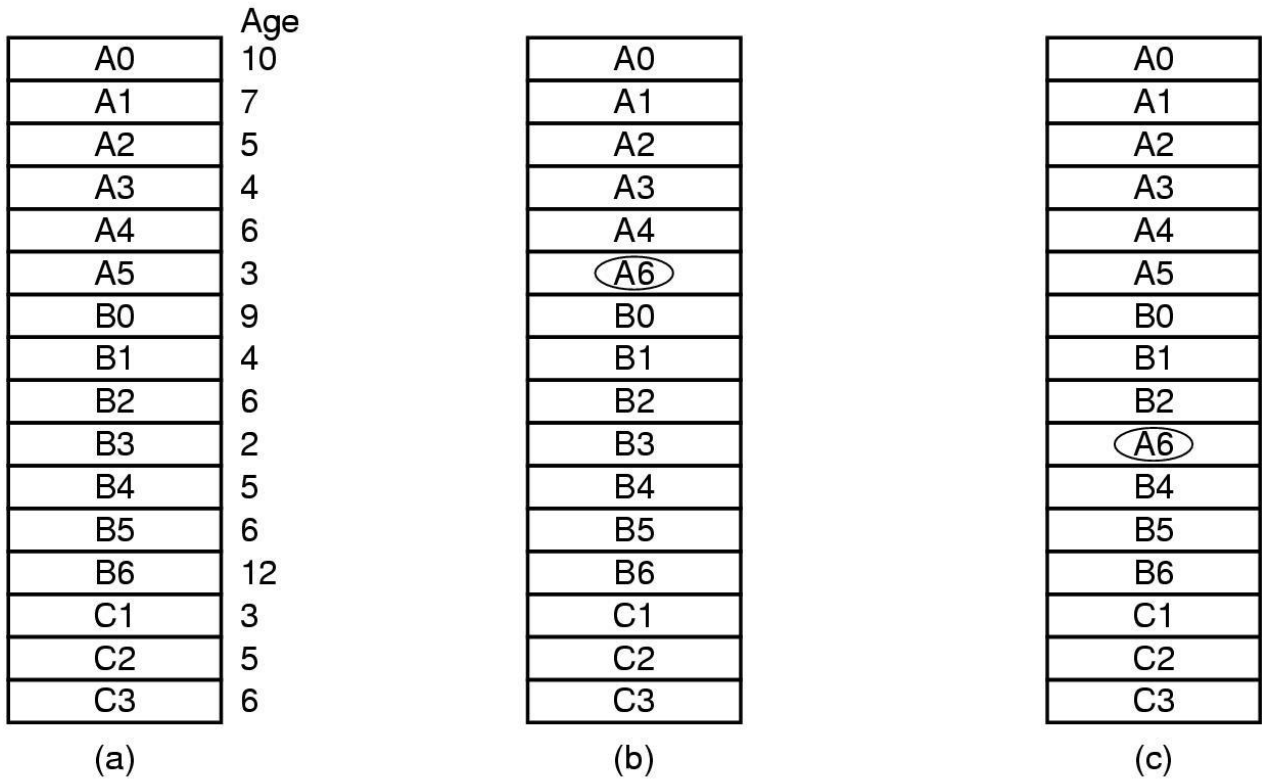
In page replacement, some frame allocation policies may be followed.

- Global Replacement: A process can replace any page in the memory.
- Local Replacement: Each process can replace only from its own reserved set of allocated page frames.

In case of local replacement, the operating system should determine how many frames should the OS allocate to each process.

The number of frames for each process may be adjusted by using two ways: Equal ; Proportional

Design Issues for Paging Systems
Local versus Global Allocation Policies (1)



- Original configuration
- Local page replacement
- Global page replacement

Thrashing

- Too many processes in too little memory
- Operating System spends all its time swapping
- Little or no real work is done
- Disk light is on all the time

- Solutions
  - Good page replacement algorithms
  - Reduce number of processes running
  - Fit more memory

Working Set Model

To prevent thrashing, a process must be provided with as many frames as it needs. For this, a model called *the working set model* is developed which depends on the locality model of program execution.

The pages used by a process within a window of time are called its working set. A parameter, Δ, called the working set window size is used. The set of pages in the last Δ page references is the working set of a process.

Choice of Δ is crucial. If Δ is too small, it will not cover the entire working set. If it is too large, several localities of a process may overlap.

The working set principle states that

1. A program should be run if and only if its working set is in memory.

2. A page may not be removed from memory if it is the member of the working set of a running process.

Protection and Sharing

- In virtual systems protection and sharing retains the characteristics of the underlying memory management systems such as paging or segmentation. However, the frequent moving of items between main memory and secondary memory may complicate the management of tables.
- If a portion of a shared object is selected for replacement then the concerned tables should be updated accordingly. All copies of the mapping information must be kept in synchrony and updated to reflect the changes of residence of shared objects.

Advantages:

User's point of view:

Illusion of large address space almost eliminates the considerations imposed by limited physical memory.

Automatic management of memory makes the program run faster.

OS's point of view:

Ability to vary the amount of physical memory in use by any program.

CPU utilization is increased and wastage of memory is reduced.

•

Disadvantages:

Complex hardware and software needed to support virtual memory.

Both time and space complexities are more when compared to other memory management schemes.

Higher table fragmentation.

Possibility of thrashing leads to complex page replacement algorithms.

Average Turnaround time is increased due to missing item exceptions.

File System

- Users make use of computers to create, store, retrieve and manipulate information.
- The file abstraction provides a uniform logical view of physical contents from a wide variety of storage devices that have different characteristics.
- A file in a computer system has a name for its identification, space to store data, a location for convenient access, access restrictions and other attributes and support mechanisms.

- OS implements a software layer on top of the I/O subsystem (device drivers) for users to access data with ease from storage devices. The software layer is called the file management system or file system.
- The file management system contains files, directories and control information (metadata) and supports convenient and secured access on files and directories.

The functions of file system are

- Organization of files
- Execution of file operations
- Synchronization of file operations
- Protection of file contents
- Management of space in the file system.

File concepts

File

- A file is a named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks which normally represents programs and data.
- In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by the files creator and user.

Naming

- A symbolic file name given to a file, for the convenience of users, is a string of characters. When a file is named, it is independent of the process, the user and the system it created. It is the only information kept in human readable form.

File structure

- A file has a certain defined structure, which depends on its type of information stored in the file, like source program, object programs, executable programs, numeric data, text, payroll records, graphic images etc.
  - A text file is a sequence of characters organized into lines.
  - A source file is a sequence of procedures and functions.
  - An object file is a sequence of bytes organized into blocks that are understandable by the machine.
  - When operating system defines different file structures, it also contains the code to support these file structure.

Attributes define the characteristics of a file varying with respect to OS and useful for protection, security and usage monitoring.

The following are the file attributes:

- *Name*: The symbolic file name is the only information kept in human readable form
- *Type*: This information is needed for those systems that support different types.
- *Location*: It is a pointer to a device used to the location of the file on that device.
- *Size*: The current size of the file (in bytes, words or blocks) and possibly the maximum allowed size are included in this attribute.
- *Protection*: Access control information controls who can do reading, writing, executing and so on.
- *Time*, *Date* and *User Identification*: This information may be kept for creation, last modification and last use.

Logical file structure

There are three possible forms of atomic units in a file.

- *Bytes*: File is a sequence of bytes called as flat file. (No internal structure)
- *Fixed length record*: Many OS require files to be divided into fixed length records. Each record is a collection of information about one thing. Easy to deal with but do not reflect the realities of data.
- *Variable length records*: These are used to meet the various workload lengths but the problem is unless the location of the file is known it is hard to perform search operation. To overcome this problem keyed file is used. Each record has a specified field which is the key field which is used for finding the location of the file.

File Meta data

- A file contains information, but the file system also keeps information about the file,
- i.e. meta data (information about information) such as Name, Type, Size and Owner of the file; Group(s) of users that have special access to the file; Access rights; Last Read time; Last Written time; The time of the file was created; which disk the file is stored on and where the file is stored on disk.

File Types

File type refers to the ability of the operating system to distinguish different types of file such as text files source files and binary files etc. The file system supports various classes of file types. A file may or may not have certain internal structure depending on its type.

Many operating systems support many types of files. Operating system like MS-DOS and UNIX have the following types of files −

Ordinary files

- These are the files that contain user information.
- These may have text, databases or executable program.
- The user can apply various operations on such files like add, modify, delete or even remove the entire file.

- Directory files
  - These files contain list of file names and other information related to these files.

Special files

- These files are also known as device files.
- These files represent physical device like disks, terminals, printers, networks, tape drive etc.
- These files are of two types −
  - **Character special files** − data is handled character by character as in case of terminals or printers.
  - **Block special files** − data is handled in blocks as in the case of disks and tapes.

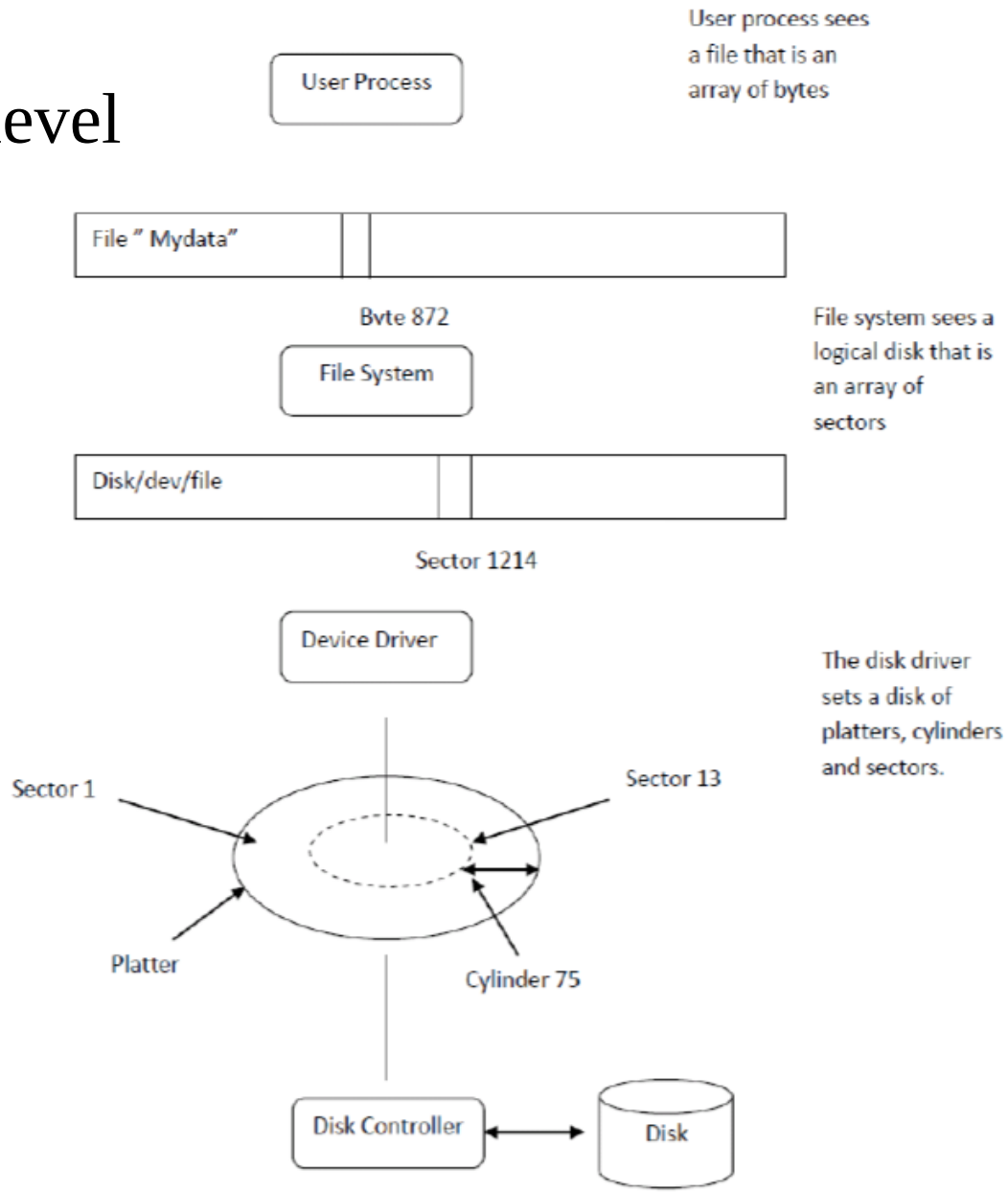
File System

The file system consists of two distinct parts:

i) a collection of files, each storing related data and

ii) a directory structure which organizes and provide information about all the files in the system.

view of a file at each level



File operations

Create:

- Essential if a system is to add files. Need not be a separate system call (can be merged with open).
- Requires the name of the file/directory being created, the name of the directory and some file attributes. Two steps are necessary to create a file:
- space in the file system must be found for the file.
- an entry for the new file must be made in the directory i.e. the directory entry records the name of the file and the location in the file system.

Delete:

- Essential if a system is to delete files.
- Requires the file/directory name to be deleted. To delete a file, the directory is searched for the named file and having found the associated directory entry, all file spaces are released and the directory entry is erased.

Open

- Not essential. An optimization in which the translation from file name to disk locations is perform only once per file rather than once per access.
- Open operation requires a filename to be opened. File system checks for permission to access the file and if the user has the permission, it creates a file descriptor that will be used by the application for future reference.

Close:

- Not essential. Free resources.
- The close operation requires a file descriptor that was obtained in the open operation. It releases the file descriptor and other related resources allocated for the open file session.

Read

- Essential. Must specify filename, file location, number of bytes, and a buffer into which the data is to be placed. Several of these parameters can be set by other system calls and in many OS's they are.
- A system call is issued that specifies the name of the file and where in the memory the file should be put for reading.
- A read operation takes as parameters file descriptor, a positive integer number and a buffer address.
- The operation copies into the buffer those many number of consecutive bytes starting at the current file pointer position from the file. It repositions the file pointer past the last byte it read.

Write

- Essential if updates are to be supported.
- A system call is made specifying both the name of the file and the information to be written to the file.
- A write operation takes as parameters a file descriptor, a byte string and the size of the string. The byte string is written in the file identified by the file descriptor.
- It overwrites the file content starting at the current file pointer position within the file. It allocates more space to the file when it needs to write past the current last byte in the file. It repositions the file pointer after the last byte written.

Truncate

- To erase the contents of the file but keep attributes same, truncating can be used i.e. attributes of the file remain same but the length of the file is reset to zero.
- A truncate operation takes as its parameters a file descriptor and a positive integer number. It reduces the size of the corresponding file to the specified number. If needed, it frees up space from the file.

Memory map

- Memory mapping a file creates a region in the process address space, and each byte in the region corresponds to a byte in the file.
- Conventional memory read and write operations on the mapped sections by applications are treated by the system as file read and write operations respectively.
- When the mapped file is closed, all the modified data are written back to the file and the file is unmapped from the process address space.

File Pointer

- On systems that do not include a file offset as part of the read and write system calls, the system must track the last read / write location as current file position pointer.
- This pointer is unique to each process operating on the file, and therefore must be kept separate from the disk file attributes.

Reposition

- adjusts a file pointer to a new offset. The operation takes a file descriptor and an offset as parameters and sets the associated file pointer to the offset. The directory is searched for the appropriate entry and current file position is set to a given value.

Seek

- Not essential (could be in read/write). Specify the offset of the next (read or write) access to this file. Repositioning within a file does not needed to involve any actual I/O. This file operation is also known as file seek.

Directory

Directories

Files in an OS are generally named using a hierarchical naming system based on directories. Almost all OS use a hierarchical naming system. In such a system, a path name consists of component names separated with a separator character.

- A directory is a name space and the associated name maps each name into either a file or another directory.
- Directories contain bookkeeping information about files.
- Directories are usually implemented as files and in OS point of view there is no distinction between files and directories.

Directory operations

- Create: Produces an ``empty'' directory. Normally the directory created actually contains . and .., so is not really empty
- Delete: Requires the directory to be empty (i.e., to just contain . and ..). Commands are normally written that will first empty the directory (except for . and ..) and then delete it. These commands make use of file and directory delete system calls.
- Opendir: Same as for files (creates a ``handle'')
- Closedir: Same as for files
- Readdir: In the old days (of unix) one could read directories as files so there was no special readdir (or opendir/closedir). It was believed that the uniform treatment would make programming (or at least system understanding) easier as there was less to learn.
- Rename: As with files
- Link: Add a second name for a file;
- Unlink: Remove a directory entry. But if there are many links and just one is unlinked, the file remains.

- The structure of the directories and the relationship among them are the main areas where file systems tend to differ, and it is also the area that has the most significant effect on the user interface provided by the file system.
- The most common directory structures used by multi-user systems are:
  - **Single-level directory**
  - **Two-level directory**
  - **Tree-structured directory**
  - **Acyclic directory**

Single-Level Directory

- In a single-level directory system, all the files are placed in one directory. This is very common on single-user OS's.
- Limitations
  - Unique name is a problem

    More number of files

    More than one user

Two-Level Directory

- In the two-level directory system, the system maintains a master block that has one entry for each user. This master block contains the addresses of the directory of the users.

Limitations

- This structure effectively isolates one user from another. This is an advantage when the users are completely independent, but a disadvantage when the users want to cooperate on some task and access files of other users.

Tree-Structured Directory

- In the tree-structured directory, the directory themselves are files. This leads to the possibility of having sub-directories that can contain files and sub-subdirectories.

Limitations

- An interesting policy decision in a tree-structured directory structure is how to handle the deletion of a directory.
- If a directory is empty, its entry in its containing directory can simply be deleted.
- However, suppose the directory to be deleted id not empty, but contains several files, or possibly sub-directories.
- Some systems will not delete a directory unless it is empty.

Acyclic-Graph Directory

- The acyclic directory is an extension of the tree-structured directory structure. In the tree-structured directory, files and directories starting from some fixed directory are owned by one particular user.
- In the acyclic structure, this prohibition is taken out and thus a directory or file under directory can be owned by several users.

Path Name

- An absolute path name is a name that gives the path from the root directory to the file that is named.
- When hierarchical file system is used, the absolute path may be long if many levels of subdirectories are present.
- Hierarchical naming system use compound names with several parts. Each part is looked up in a flat name space usually called a directory. If this look up leads to another directory, then the next part of the compound name is looked up in that directory.
- To reduce the length of the path name, working/current directory concept is used. This allows the use of relative path names.
- A relative path name is a path name that starts at the working directory rather than the root directory.

Aliases

- Hierarchical directory systems provide a means for classifying and grouping files.
- All the files in a single directory tend to be related in some way, but sometimes a file is related to several different groups and it is an inconvenient restriction to have no place it in just one group.
- This problem can be handled with the concept of a file alias. It is a file name that refers to a file that also has another name. There may be two or more absolute path names.

File Access Mechanisms

File access mechanism refers to the manner in which the records of a file may be accessed. There are several ways to access files −

Sequential access

Direct/Random access

Indexed sequential access

Sequential access

A sequential access is that in which the records are accessed in some sequence, i.e., the information in the file is processed in order, one record after the other. This access method is the most primitive one. Example: Compilers usually access files in this fashion.

Direct/Random access

Random access file organization provides, accessing the records directly.

Each record has its own address on the file with by the help of which it can be directly accessed for reading or writing.

The records need not be in any sequence within the file and they need not be in adjacent locations on the storage medium.

Indexed sequential access

This mechanism is built up on base of sequential access.

An index is created for each file which contains pointers to various blocks.

Index is searched sequentially and its pointer is used to access the file directly.

File System Implementation

File System Implementation

- The file system is implemented using files, directories and the files that are currently opened.
- Each one will be implemented as a data structure and a set of procedures to manipulate the data structures

The open file table contains a structure for each open file. An open file structure contains the following information

- Current file position
- Other status information about the open file (eg. Whether the file is locked or not)
- Pointer to the file descriptor that is open

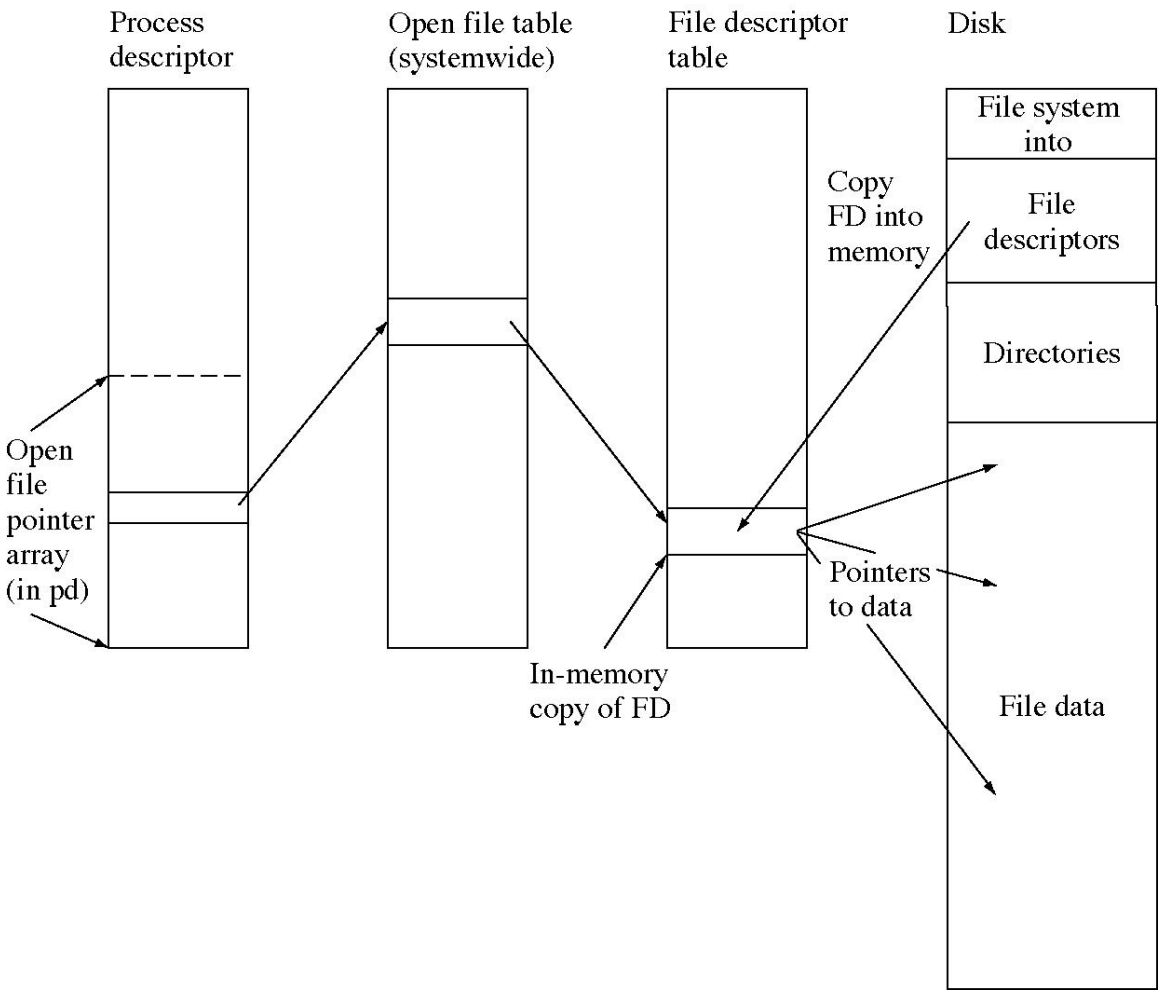
Most open files are connected to files. Files exist on disk and consist of two parts

- File descriptor : contains all the meta data about the file
- File data: Kept in disk blocks

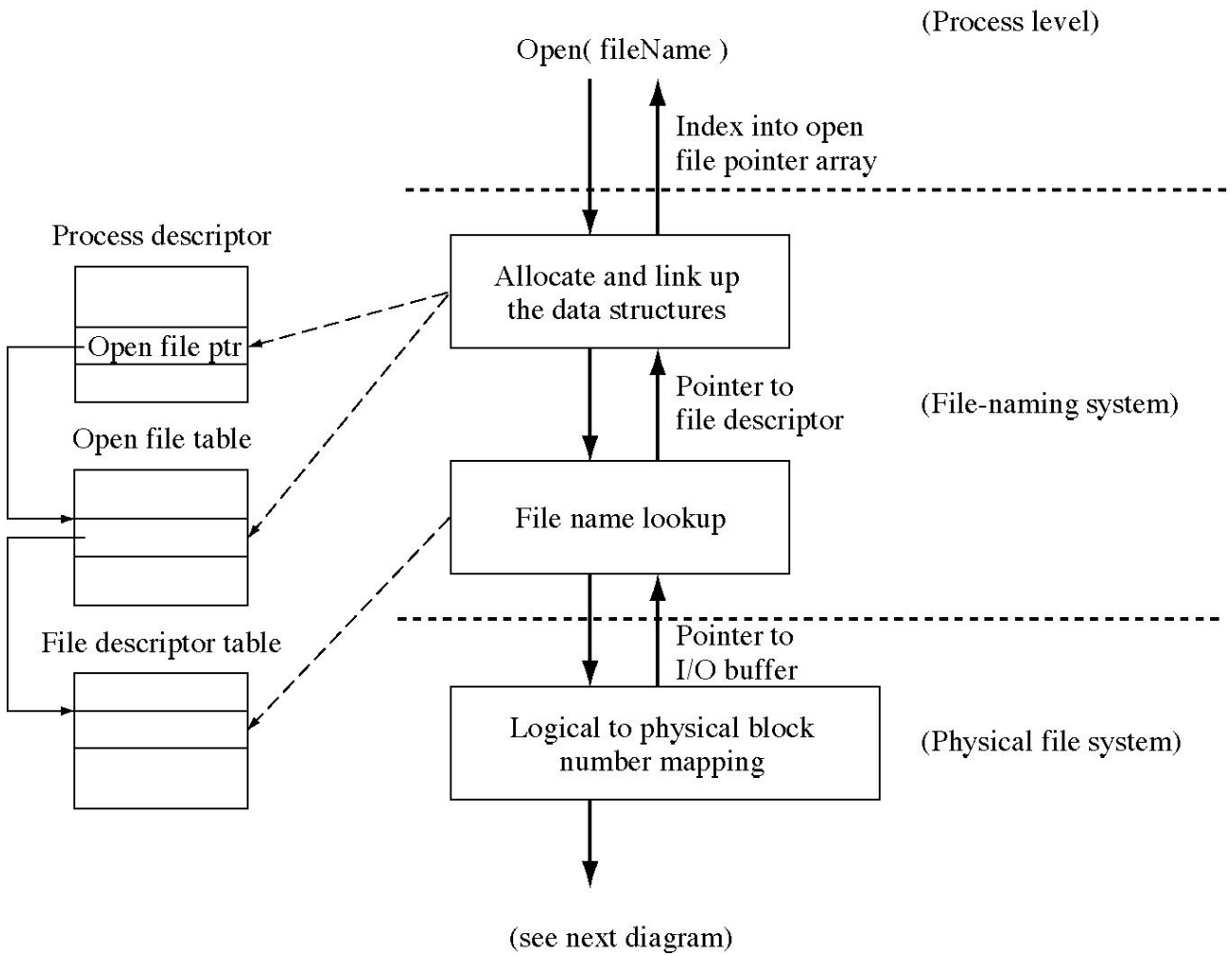
File Descriptor data structure contains the following information

- Owner of the file
- File protection information
- Time of creation, last modification and last use
- Other file meta data
- Location of the file data

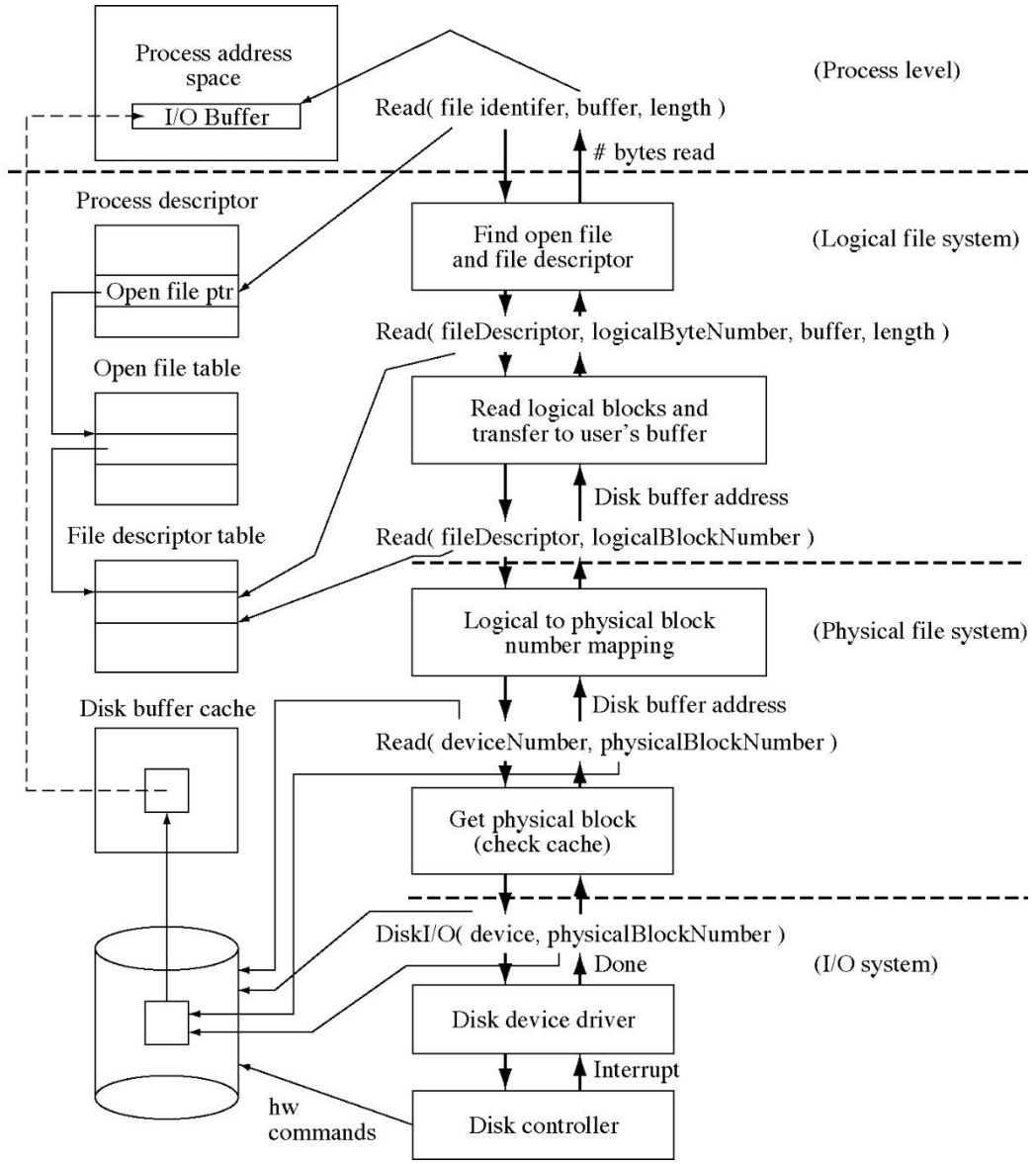
File system data structures



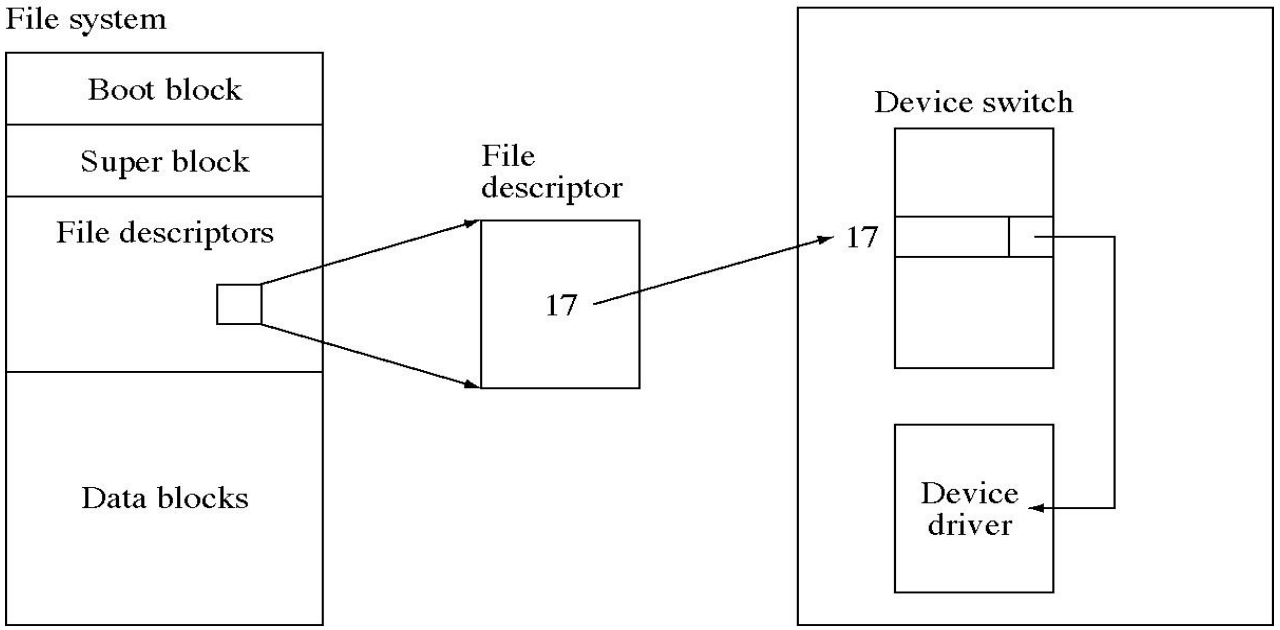
Flow of control for an open



Flow of control for a read



Connecting files and devices



Directory Implementation

A directory is a table that maps component names to file descriptors. The steps involved in a path lookup are

1) Let FD be the root if the path starts with '/' or FD be current directory and start at the beginning of the path name.

2) If the path name is at the end then return FD.

3) Isolate the next component in the path name and let it be C. Move past the component in the path name.

4) if FD is not a directory, then return an error.

5) Search through the directory FD for the component name C. This involves a loop that reads the next name/fd pair and compares the name with C. Loop until a match is found or the end of the directory is reached.

6) If no match is found, then return an error.

7) If a match was found, then its associated file descriptor becomes the new FD. Go back to step2.

Free Space Management

Space Allocation

Files are allocated disk spaces by operating system. Operating systems deploy following three main ways to allocate disk space to files.

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

Contiguous Allocation

- Each file occupies a contiguous address space on disk.
- Assigned disk address is in linear order.
- Easy to implement.
- The difficulty with contiguous allocation is finding space for a new file. If the file to be created is n blocks long, then the OS must search for n free contiguous blocks.
- First-fit, best-fit, and worst-fit strategies are the most common strategies used to select a free hole from the set of available holes.
- External fragmentation is a major issue with this type of allocation technique.
- Another problem with contiguous allocation is determining how much disk space is needed for a file.

Linked Allocation

- Each file carries a list of links to disk blocks.
- Directory contains link / pointer to first block of a file.
- This pointer is initialized to NIL (the end-of-list pointer value) to signify an empty file.
- A write to a file removes the first free block and writes to that block. This new block is then linked to the end of the file.
- To read a file, the pointers are just followed from block to block.
- No external fragmentation
- Effectively used in sequential access file.
- Inefficient in case of direct access file.

Indexed Allocation

- Provides solutions to problems of contiguous and linked allocation.
- A index block is created having all pointers to files.
- Each file has its own index block which stores the addresses of disk space occupied by the file which is an array of disk sector of addresses. The i^{th} entry in the index block points to the i^{th} sector of the file.
- Directory contains the addresses of index blocks of files.
- To read the i^{th} sector of the file, the pointer in the i^{th} index block entry is read to find the desired sector.
- Indexed allocation supports direct access, without suffering from external fragmentation. Any free block anywhere on the disk may satisfy a request for more space.

Free Space List

- Since there is only a limited amount of disk space, it is necessary to reuse the space from deleted files for new files.
- To keep track of free disk space, the system maintains a free-space list.
- The free-space list records all disk blocks that are free (i.e., are not allocated to some file).
- To create a file, the free-space list has to be searched for the required amount of space, and allocate that space to a new file. This space is then removed from the free-space list.
- When a file is deleted, its disk space is added to the free-space list.

Free Space List

Bit-Vector

- Frequently, the free-space list is implemented as a bit map or bit vector. Each block is represented by a 1 bit. If the block is free, the bit is 0; if the block is allocated, the bit is 1.
- For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free, and the rest of the blocks are allocated. The free-space bit map would be: 11000011000000111001111110001111…
- The main advantage of this approach is that it is relatively simple and efficient to find ‘n’ consecutive free blocks on the disk.
- Unfortunately, bit vectors are inefficient unless the entire vector is kept in memory for most accesses.
- Keeping it main memory is possible for smaller disks such as on microcomputers, but not for larger ones.

Linked List

- In this approach all the free disk blocks are linked together, keeping a pointer to the first free block. This block contains a pointer to the next free disk block, and so on.
- In the previous example, a pointer could be kept to block 2, as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on.
- This scheme is not efficient; to traverse the list, each block must be read, which requires substantial I/O time.

Grouping

- A modification of the free-list approach is to store the addresses of ‘n’ free blocks in the first free block.
- The first n-1 of these is actually free. The last one is the disk address of another block containing addresses of another n free blocks.
- The importance of this implementation is that addresses of a large number of free blocks can be found quickly.

Counting

- Another approach is to take advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when contiguous allocation is used.
- Thus, rather than keeping a list of free disk addresses, the address of the first free block is kept and the number ’n’ of free contiguous blocks that follow the first block.
- Each entry in the free-space list then consists of a disk address and a count.