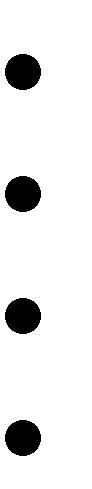
**UNIT -3 CONCURRENCY**

#### Process Synchronization

 To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data

To present both software and hardware solutions of the critical-section problem

To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity Concurrent access to shared data may result in data inconsistency

Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

 Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer

#### Producer

while (true) {

/\* produce an item and put in nextProduced \*/ while (count == BUFFER\_SIZE)

; // do nothing

buffer [in] = nextProduced;

in = (in + 1) % BUFFER\_SIZE;

count++;

}

#### Consumer

while (true) {

while (count == 0)

; // do nothing

nextConsumed = buffer[out];

out = (out + 1) % BUFFER\_SIZE;

count--;

/\* consume the item in nextConsumed

}

#### Race Condition

count++ could be implemented as

register1 = count register1 = register1 + 1 count = register1

count-- could be implemented as

register2 = count register2 = register2 - 1 count = register2

Consider this execution interleaving with “count = 5” initially:

S0: producer execute register1 = count {register1 = 5} S1: producer execute register1 = register1 + 1 {register1 = 6} S2: consumer execute register2 = count {register2 = 5}

S3: consumer execute register2 = register2 - 1 {register2 = 4} S4: producer execute count = register1 {count = 6 }

S5: consumer execute count = register2 {count = 4}

#### Solution to Critical-Section Problem

1. Mutual Exclusion - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted Assume that each process executes at a nonzero speed

No assumption concerning relative speed of the N processes

#### Peterson’s Solution

Two process solution

Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted. The two processes share two variables:

int turn; Boolean flag[2]

The variable turn indicates whose turn it is to enter the critical section.

The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready!

#### Algorithm for Process Pi

do {

flag[i] = TRUE; turn = j;

while (flag[j] && turn == j); critical section

flag[i] = FALSE;

remainder section

} while (TRUE);

#### Synchronization Hardware

Many systems provide hardware support for critical section code Uniprocessors – could disable interrupts

Currently running code would execute without preemption Generally too inefficient on multiprocessor systems

Operating systems using this not broadly scalable

 Modern machines provide special atomic hardware instructions Atomic = non-interruptable

Either test memory word and set value Or swap contents of two memory words

#### Solution to Critical-section Problem Using Locks

do {

acquire lock

critical section release lock

remainder section

} while (TRUE); **TestAndSet Instruction** Definition:

boolean TestAndSet (boolean \*target)

{

boolean rv = \*target;

\*target = TRUE; return rv:

}

#### Solution using TestAndSet

Shared boolean variable lock., initialized to false. Solution:

do {

while ( TestAndSet (&lock ))

; // do nothing

// critical section lock = FALSE;

// remainder section

} while (TRUE);

#### Swap Instruction

Definition:

void Swap (boolean \*a, boolean \*b)

{

boolean temp = \*a;

\*a = \*b;

\*b = temp:

}

#### Solution using Swap

Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key Solution:

do {

key = TRUE;

while ( key == TRUE) Swap (&lock, &key );

// critical section lock = FALSE;

// remainder section

} while (TRUE);

#### Bounded-waiting Mutual Exclusion with TestandSet()

do {

waiting[i] = TRUE; key = TRUE;

while (waiting[i] && key)

key = TestAndSet(&lock); waiting[i] = FALSE;

// critical section j = (i + 1) % n;

while ((j != i) && !waiting[j]) j = (j + 1) % n;

if (j == i)

lock = FALSE;

else

waiting[j] = FALSE;

// remainder section

} while (TRUE);

#### Semaphore

Synchronization tool that does not require busy waiting nSemaphore *S* – integer variable Two standard operations modify S: wait() and signal()

Originally called P() and V() Less complicated

Can only be accessed via two indivisible (atomic) operations wait (S) {

while S <= 0

; // no-op

S--;

}

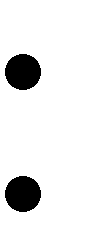
signal (S) {

S++;

}

#### Semaphore as General Synchronization Tool

Counting semaphore – integer value can range over an unrestricted domain Binary semaphore – integer value can range only between 0

and 1; can be simpler to implement

Also known as mutex locksnCan implement a counting semaphore S as a binary semaphore Provides mutual exclusionSemaphore mutex; // initialized to do {

wait (mutex);

// Critical Section signal (mutex);

// remainder section

} while (TRUE);

#### Semaphore Implementation

 Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time

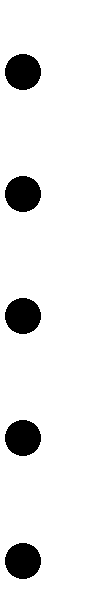
 Thus, implementation becomes the critical section problem where the wait and signal code are placed in the crtical section.

 Could now have busy waiting in critical section implementation But implementation code is short

Little busy waiting if critical section rarely occupied

 Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

#### Semaphore Implementation with no Busy waiting

 With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:

value (of type integer)

pointer to next record in the list Two operations:

block – place the process invoking the operation on the appropriate waiting queue. wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Implementation of wait:

wait(semaphore \*S) {

S->value--;

if (S->value < 0) {

add this process to S->list; block();

}

}

Implementation of signal:

signal(semaphore \*S) {

S->value++;

if (S->value <= 0) {

remove a process P from S->list; wakeup(P);

}

}

#### Deadlock and Starvation

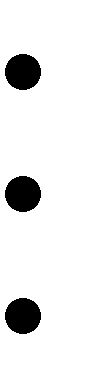
 Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let S and Q be two semaphores initialized to 1

|  |  |  |
| --- | --- | --- |
| *P*0  wait (S);  wait (Q);  . | *P*1  wait (Q);  wait (S); |  |
| .  .  signal (S); | signal (Q); | .  . |
| signal (Q); | signal (S); |  |

 Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended

 Priority Inversion - Scheduling problem when lower-priority process holds a lock needed by higher- priority process

**Classical Problems of Synchronization** Bounded-Buffer Problem Readers and Writers Problem Dining-Philosophers Problem

#### Bounded-Buffer Problem

*N* buffers, each can hold one item Semaphore mutex initialized to the value 1 Semaphore full initialized to the value 0 Semaphore empty initialized to the value N.

The structure of the producer process

do { // produce an item in nextp wait (empty);

wait (mutex);

// add the item to the buffer signal (mutex);

signal (full);

} while (TRUE);

The structure of the consumer process do { wait (full);

wait (mutex);

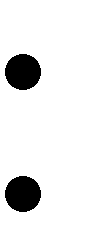
// remove an item from buffer to nextc signal (mutex);

signal (empty);

// consume the item in nextc

} while (TRUE);

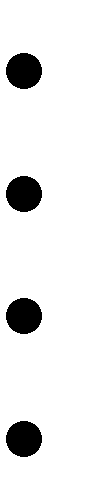
#### Readers-Writers Problem

A data set is shared among a number of concurrent processes

Readers – only read the data set; they do **not** perform any updates

Writers – can both read and writenProblem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time

Shared Data

Data set

Semaphore mutex initialized to 1 Semaphore wrt initialized to 1 Integer readcount initialized to 0

The structure of a writer process

do { wait (wrt) ;

// writing is performed signal (wrt) ;

} while (TRUE);

The structure of a reader process do {

wait (mutex) ; readcount ++ ;

if (readcount == 1)

wait (wrt) ; signal (mutex)

// reading is performed wait (mutex) ;

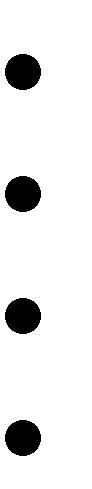
readcount - - ;

if (readcount == 0)

signal (wrt) ; signal (mutex) ;

} while (TRUE);

#### Dining-Philosophers Problem



do {

Shared data

Bowl of rice (data set)

Semaphore chopstick [5] initialized to 1 The structure of Philosopher *i*:

wait ( chopstick[i] );

wait ( chopStick[ (i + 1) % 5] );

// eat

signal ( chopstick[i] );

signal (chopstick[ (i + 1) % 5] );

// think

} while (TRUE);

#### Problems with Semaphores

Incorrect use of semaphore operations:

l signal (mutex)

….

wait (mutex)

wait (mutex) … wait (mutex)

Omitting of wait (mutex) or signal (mutex) (or both)

#### Monitors

A high-level abstraction that provides a convenient and effective mechanism for process synchronization Only one process may be active within the monitor at a time

monitor monitor-name

{

// shared variable declarations procedure P1 (…) { …. }

…

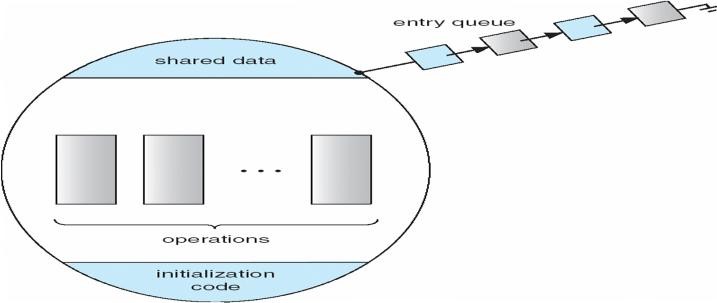
procedure Pn (…) {……} Initialization code ( ….) { … }

…

}

}

#### Schematic view of a Monitor



**Condition Variables**

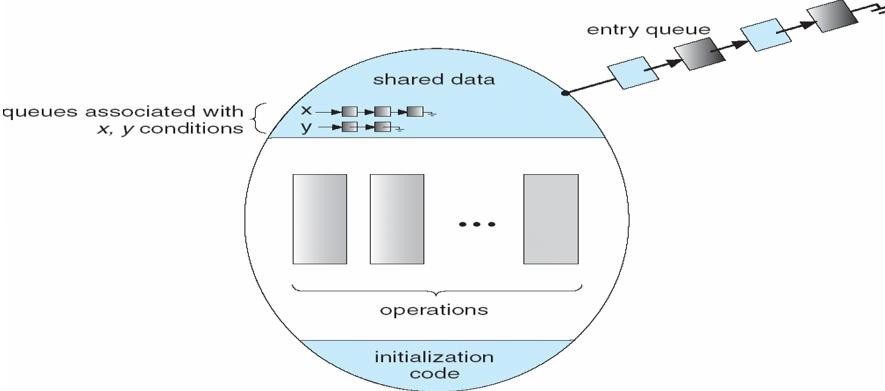
condition x, y;

Two operations on a condition variable:

x.wait () – a process that invokes the operation is suspended.

x.signal () – resumes one of processes (if any) that invoked x.wait ()

#### Monitor with Condition Variables



**Solution to Dining Philosophers**

monitor DP

{

enum { THINKING; HUNGRY, EATING) state [5] ;

condition self [5]; void pickup (int i) {

state[i] = HUNGRY; test(i);

if (state[i] != EATING) self [i].wait;

}

void putdown (int i) {

state[i] = THINKING;

// test left and right neighbors test((i + 4) % 5);

test((i + 1) % 5);

}

void test (int i) {

if ( (state[(i + 4) % 5] != EATING) && (state[i] == HUNGRY) &&

(state[(i + 1) % 5] != EATING) ) {

state[i] = EATING ; self[i].signal () ;

}

}

initialization\_code() {

for (int i = 0; i < 5; i++) state[i] = THINKING;

}

}

Each philosopher *I* invokes the operations pickup() and putdown() in the following sequence:

DiningPhilosophters.pickup (i); EAT

DiningPhilosophers.putdown (i);

#### Monitor Implementation Using Semaphores

**Variables**

semaphore mutex; // (initially = 1) semaphore next; // (initially = 0)

int next-count = 0;nEach procedure ***F*** will be replaced by wait(mutex);

… body of *F*;

…

if (next\_count > 0) signal(next)

else

signal(mutex);nMutual exclusion within a monitor is ensured.

#### Monitor Implementation

For each condition variable ***x***, we have:

semaphore x\_sem; // (initially = 0)

int x-count = 0;nThe operation x.wait can be implemented as:

x-count++;

if (next\_count > 0) signal(next);

else signal(mutex); wait(x\_sem);

x-count--;

The operation x.signal can be implemented as:

if (x-count > 0) {

next\_count++; signal(x\_sem); wait(next); next\_count--;

}

#### A Monitor to Allocate Single Resource

monitor ResourceAllocator

{

boolean busy; condition x;

void acquire(int time) {

if (busy)

x.wait(time);

busy = TRUE;

}

void release() {

busy = FALSE; x.signal();

initialization code() {

}

}

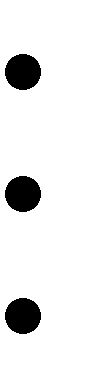
busy = FALSE;

}

#### Synchronization Examples

Solaris Windows XP Linux Pthreads

#### Solaris Synchronization

 **Implements a variety of locks to support** multitasking, multithreading (including real-time threads), and multiprocessing

Uses adaptive mutexes for efficiency when protecting data from short code segments

Uses condition variables and readers-writers locks when longer sections of code need access to data Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

#### Windows XP Synchronization

Uses interrupt masks to protect access to global resources on uniprocessor systems Uses spinlocks on multiprocessor systems

Also provides dispatcher objects which may act as either mutexes and semaphores Dispatcher objects may also provide events

An event acts much like a condition variable

#### Linux Synchronization

Linux:lPrior to kernel Version 2.6, disables interrupts to implement short critical sections Version 2.6 and later, fully preemptive

Linux provides:

semaphores spin locks

#### Pthreads Synchronization

Pthreads API is OS-independent It provides:

mutex locks

condition variablesnNon-portable extensions include: read-write locks

spin locks

#### Atomic Transactions

System Model

Log-based Recovery Checkpoints

Concurrent Atomic Transactions

#### System Model

Assures that operations happen as a single logical unit of work, in its entirety, or not at all Related to field of database systems

Challenge is assuring atomicity despite computer system failures

Transaction - collection of instructions or operations that performs single logical function Here we are concerned with changes to stable storage – disk

Transaction is series of read and write operations

Terminated by commit (transaction successful) or abort (transaction failed) operation Aborted transaction must be rolled back to undo any changes it performed

#### Types of Storage Media

Volatile storage – information stored here does not survive system crashes Example: main memory, cache

Nonvolatile storage – Information usually survives crashes Example: disk and tape

Stable storage – Information never lost

Not actually possible, so approximated via replication or RAID to devices with independent failure modes

 Goal is to assure transaction atomicity where failures cause loss of information on volatile storage

#### Log-Based Recovery

Record to stable storage information about all modifications by a transaction Most common is write-ahead logging

Log on stable storage, each log record describes single transaction write operation, including Transaction name

Data item name Old value

New value



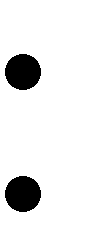
<Ti starts> written to log when transaction Ti starts

<Ti commits> written when Ti commits

Log entry must reach stable storage before operation on data occurs

#### Log-Based Recovery Algorithm

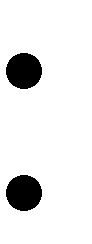
**Using the log, system can handle any volatile memory** errors Undo(Ti) restores value of all data updated by Ti



Redo(Ti) sets values of all data in transaction Ti to new values Undo(Ti) and redo(Ti) must be idempotent

Multiple executions must have the same result as one execution

If system fails, restore state of all updated data via log



If log contains <Ti starts> without <Ti commits>, undo(Ti) If log contains <Ti starts> and <Ti commits>, redo(Ti)

#### Checkpoints

Log could become long, and recovery could take long Checkpoints shorten log and recovery time.

Checkpoint scheme:

1.Output all log records currently in volatile storage to stable storage 2.Output all modified data from volatile to stable storage

1. Output a log record <checkpoint> to the log on stable storage

Now recovery only includes Ti, such that Ti started executing before the most recent checkpoint, and all transactions after Ti All other transactions already on stable storage

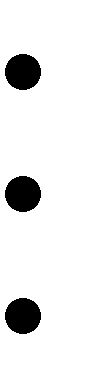
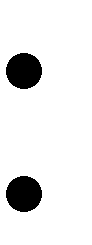
#### Concurrent Transactions

Must be equivalent to serial execution – serializability Could perform all transactions in critical section Inefficient, too restrictive

Concurrency-control algorithms provide serializability

#### Serializability

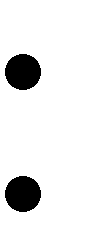
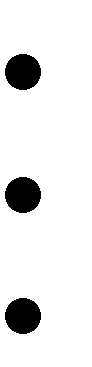
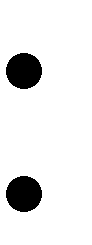
Consider two data items A and B Consider Transactions T0 and T1 Execute T0, T1 atomically Execution sequence called schedule



Atomically executed transaction order called serial schedule For N transactions, there are N! valid serial schedules

#### Schedule 1: T0 then T1

**Nonserial Schedule**



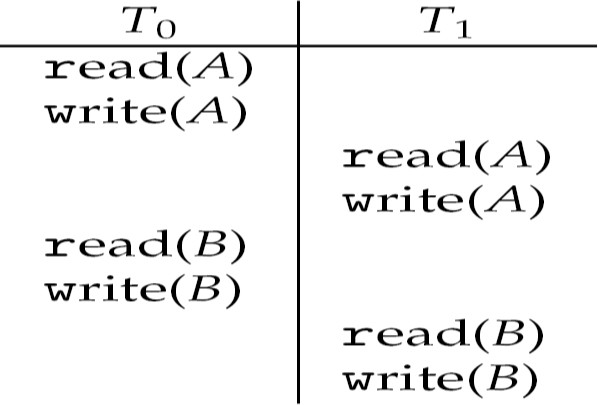
Nonserial schedule allows overlapped execute Resulting execution not necessarily incorrect Consider schedule S, operations Oi, Oj

Conflict if access same data item, with at least one write

If Oi, Oj consecutive and operations of different transactions & Oi and Oj don’t conflict Then S’ with swapped order Oj Oi equivalent to S

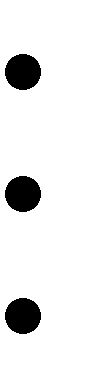
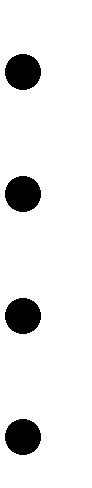
If S can become S’ via swapping nonconflicting operations S is conflict serializable

#### Schedule 2: Concurrent Serializable Schedule



**Locking Protocol**

Ensure serializability by associating lock with each data item Follow locking protocol for access control



Locks

Shared – Ti has shared-mode lock (S) on item Q, Ti can read Q but not write Q Exclusive – Ti has exclusive-mode lock (X) on Q, Ti can read and write Q Require every transaction on item Q acquire appropriate lock

If lock already held, new request may have to wait Similar to readers-writers algorithm

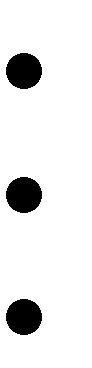
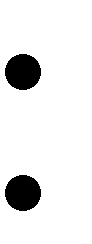
#### Two-phase Locking Protocol

Generally ensures conflict serializability

Each transaction issues lock and unlock requests in two phases Growing – obtaining locks

Shrinking – releasing locks Does not prevent deadlock

#### Timestamp-based Protocols

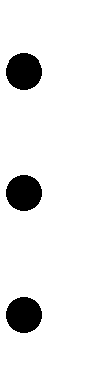


Select order among transactions in advance – timestamp-ordering Transaction Ti associated with timestamp TS(Ti) before Ti starts TS(Ti) < TS(Tj) if Ti entered system before Tj

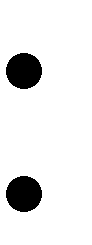
TS can be generated from system clock or as logical counter incremented at each entry of transaction Timestamps determine serializability order

If TS(Ti) < TS(Tj), system must ensure produced schedule equivalent to serial schedule where Ti appears before Tj

#### Timestamp-based Protocol Implementation

Data item Q gets two timestamps

W-timestamp(Q) – largest timestamp of any transaction that executed write(Q) successfully R-timestamp(Q) – largest timestamp of successful read(Q)

Updated whenever read(Q) or write(Q) executed

Timestamp-ordering protocol assures any conflicting read and write executed in timestamp order Suppose Ti executes read(Q)

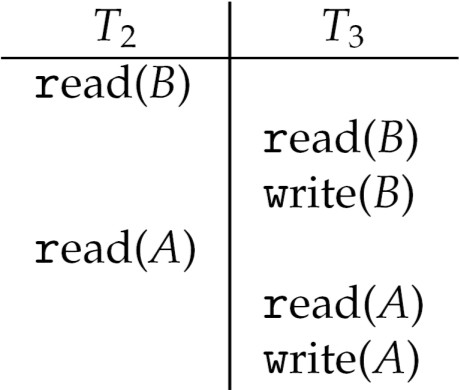
If TS(Ti) < W-timestamp(Q), Ti needs to read value of Q that was already overwritten read operation rejected and Ti rolled back

If TS(Ti) ≥ W-timestamp(Q) read executed, R-timestamp(Q) set to max(R-timestamp(Q), TS(Ti))

#### Timestamp-ordering Protocol

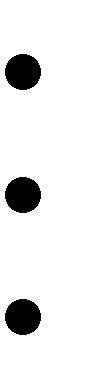
Supose Ti executes write (Q)

If TS(Ti) < R-timestamp(Q), value Q produced by Ti was needed previously and Ti assumed it would never be produced Write operation rejected, Ti rolled back If TS(Ti) < W-tiimestamp(Q), Ti attempting to write obsolete value of Q Write operation rejected and Ti rolled back Otherwise, write executed Any rolled back transaction Ti is assigned new timestamp and restarted Algorithm ensures conflict serializability and freedom from deadlock **Schedule Possible Under Timestamp Protocol**

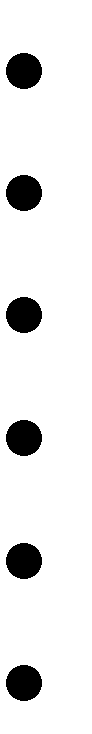


**UNIT IV**

**Memory Management**

To provide a detailed description of various ways of organizing memory hardware

To discuss various memory-management techniques, including paging and segmentation

To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

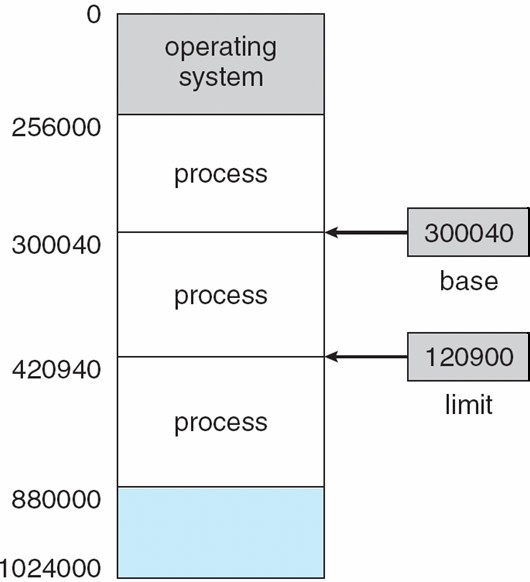
Program must be brought (from disk) into memory and placed within a process for it to be run Main memory and registers are only storage CPU can access directly

Register access in one CPU clock (or less) Main memory can take many cycles

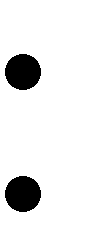
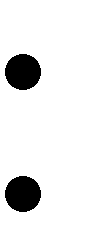
**Cache** sits between main memory and CPU registers Protection of memory required to ensure correct operation

## Base and Limit Registers

A pair of **base** and **limit** registers define the logical address space

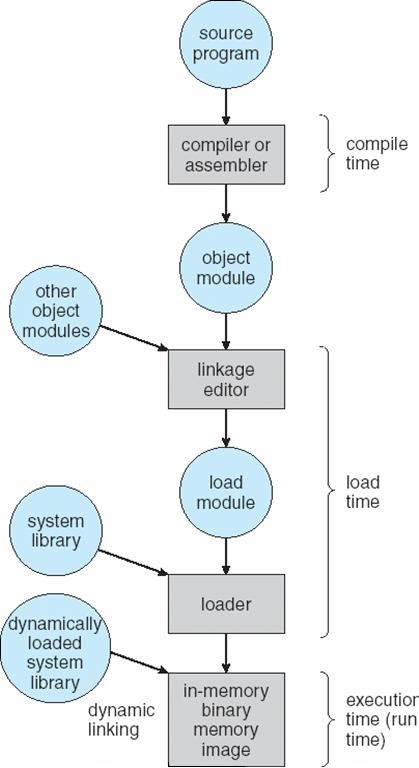


## Binding of Instructions and Data to Memory

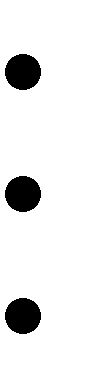
Address binding of instructions and data to memory addresses can happen at three different stages **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes

**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)

## Multistep Processing of a User Program



**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

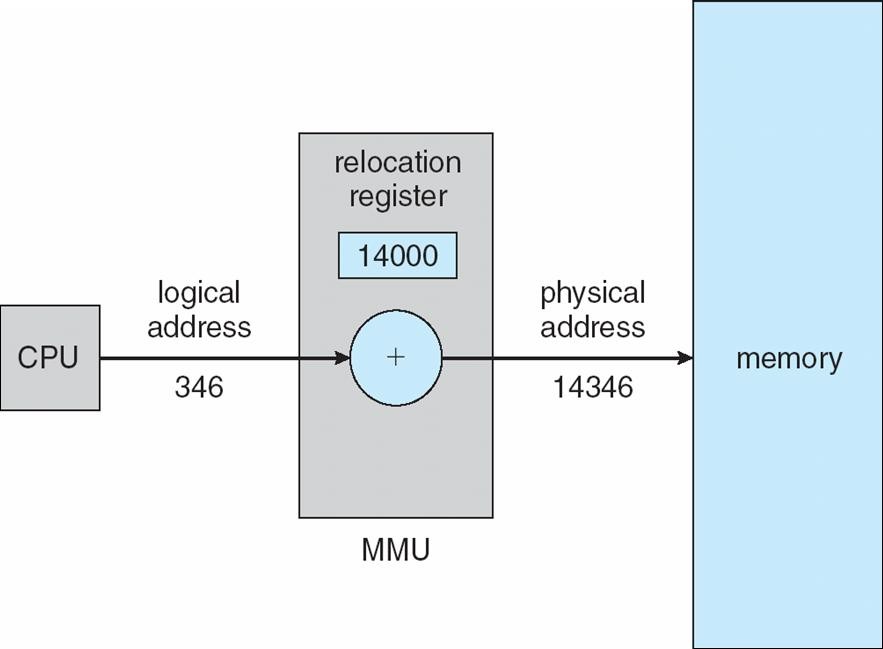
## Memory-Management Unit (MMU)

Hardware device that maps virtual to physical address

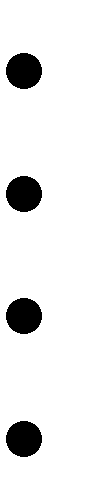
In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory

 The user program deals with *logical* addresses; it never sees the *real* physical addresses

## Dynamic relocation using a relocation register



**Dynamic Loading**

Routine is not loaded until it is called

Better memory-space utilization; unused routine is never loaded

Useful when large amounts of code are needed to handle infrequently occurring cases

No special support from the operating system is required implemented through program design

## Dynamic Linking

Linking postponed until execution time

Small piece of code, *stub*, used to locate the appropriate memory-resident library routine Stub replaces itself with the address of the routine, and executes the routine

Operating system needed to check if routine is in processes’ memory address Dynamic linking is particularly useful for libraries

System also known as **shared libraries**

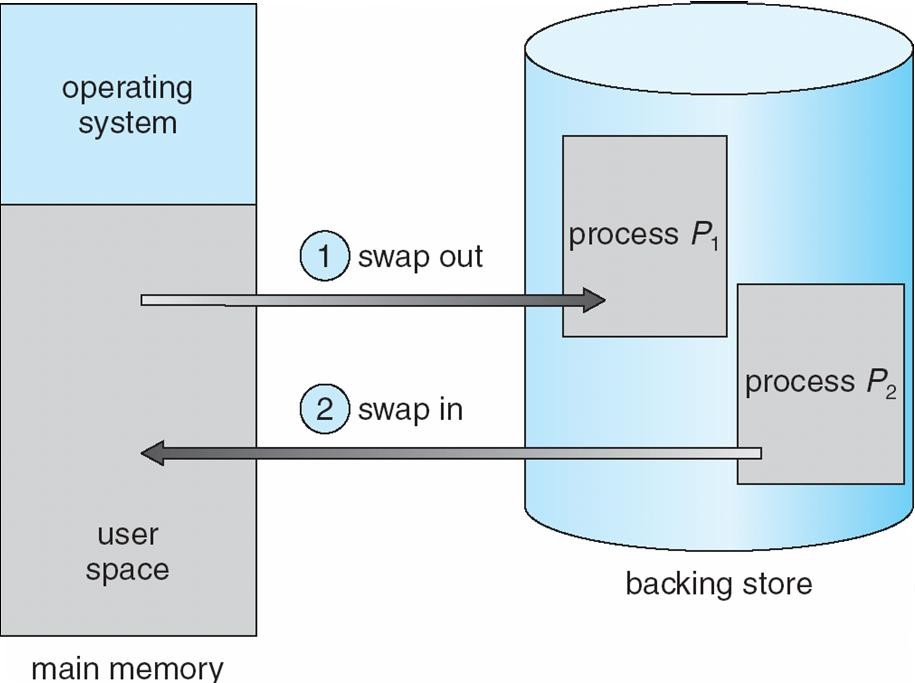
## Swapping

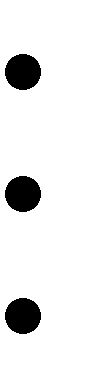
A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued executionn**Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory imagesn**Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executednMajor part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swappednModified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

System maintains a **ready queue** of ready-to-run processes which have memory images on disk

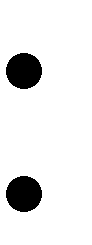
## Schematic View of Swapping

**Contiguous Allocation**



Main memory usually into two partitions:

Resident operating system, usually held in low memory with interrupt vector

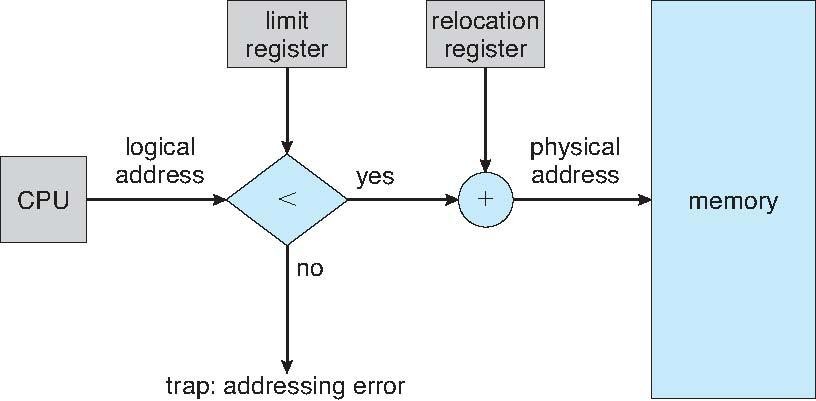
User processes then held in high memorynRelocation registers used to protect user processes from each other, and from changing operating-system code and data

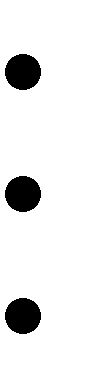
Base register contains value of smallest physical address

Limit register contains range of logical addresses – each logical address must be less than the limit register

 MMU maps logical address *dynamically*

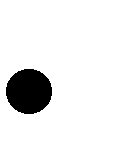
## Hardware Support for Relocation and Limit Registers

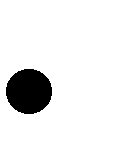




Multiple-partition allocation

Hole – block of available memory; holes of various size are scattered throughout memory When a process arrives, it is allocated memory from a hole large enough to accommodate it

 Operating system maintains information about:

 a) allocated partitions b) free partitions (hole)

|  |
| --- |
| OS |
| process 5 |
| process 8 |
| process 2 |

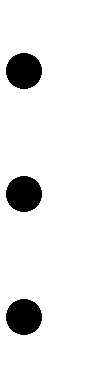
|  |
| --- |
| OS |
| process 5 |
|  |
| process 2 |

|  |
| --- |
| OS |
| process 5 |
| process 9 |
|  |
| process 2 |

|  |
| --- |
| OS |
| process 5 |
| process 9 |
| process 10 |
|  |
| process 2 |

## Dynamic Storage-Allocation Problem

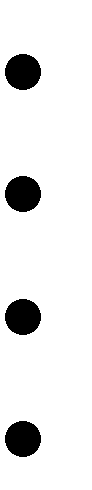
**First-fit**: Allocate the *first* hole that is big enough

**Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size Produces the smallest leftover hole

**Worst-fit**: Allocate the *largest* hole; must also search entire list Produces the largest leftover hole

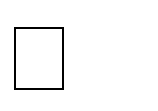
First-fit and best-fit better than worst-fit in terms of speed and storage utilization

## Fragmentation

**External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Reduce external fragmentation by **compaction**

Shuffle memory contents to place all free memory together in one large block Compaction is possible *only* if relocation is dynamic, and is done at execution time. I/O problem



Latch job in memory while it is involved in I/O

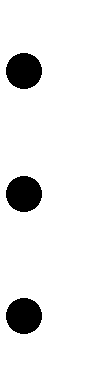
Do I/O only into OS buffers

## Paging

 Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available

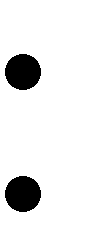
 Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)

Divide logical memory into blocks of same size called **pages**nKeep track of all free frames

To run a program of size ***n*** pages, need to find *n* free frames and load program Set up a page table to translate logical to physical addresses

Internal fragmentation

## Address Translation Scheme

Address generated by CPU is divided into

**Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory

 **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit

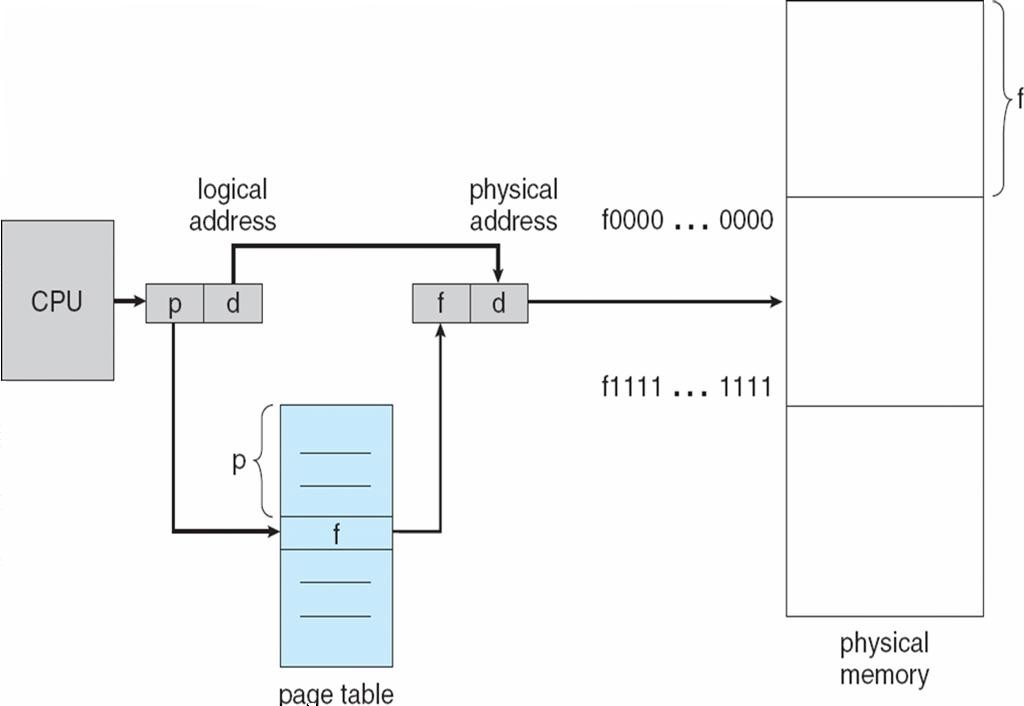
 For given logical address space 2*m and page size 2*n

## Paging Hardware

*n*

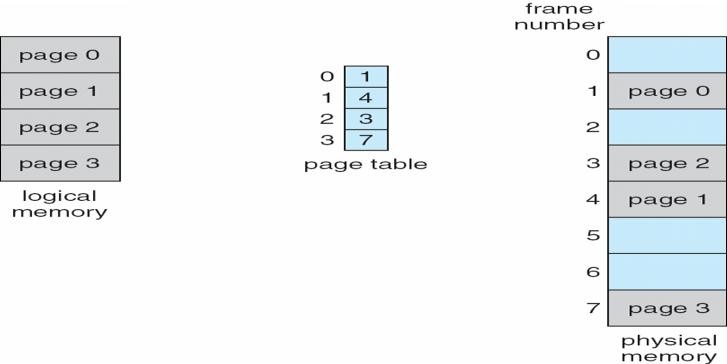
*m - n*

p

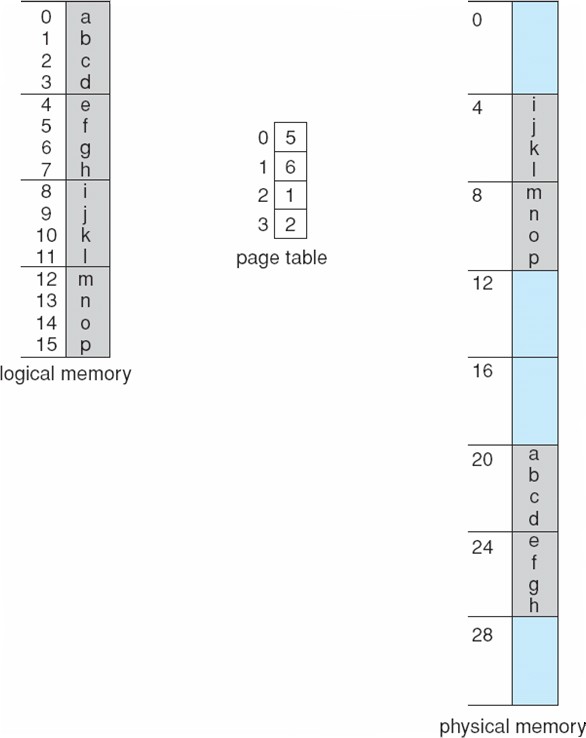


|  |  |
| --- | --- |
| age number | page offset |
| *p* | *d* |

**Paging Model of Logical and Physical Memory**

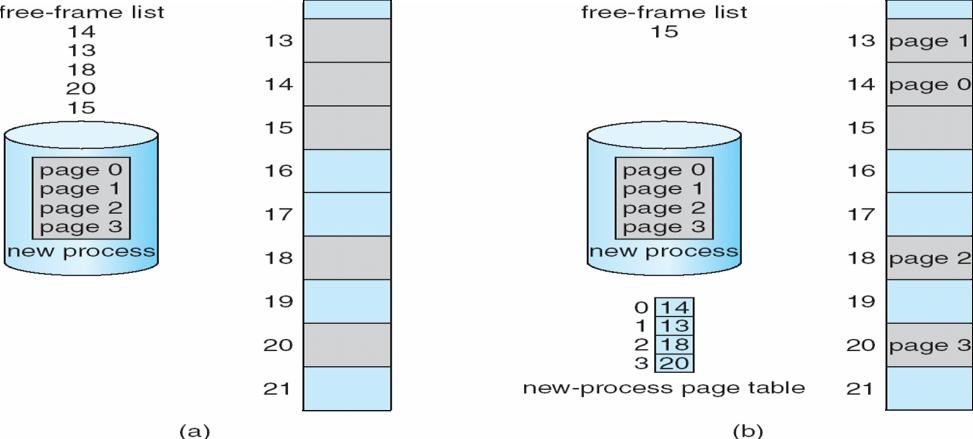


**Paging Example**

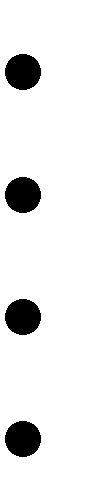


**32-byte memory and 4-byte pages**

## Free Frames



**Implementation of Page Table**

Page table is kept in main memory

**Page-table base register (PTBR)** points to the page table

**Page-table length register (PRLR)** indicates size of the page table

In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

 The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

 Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

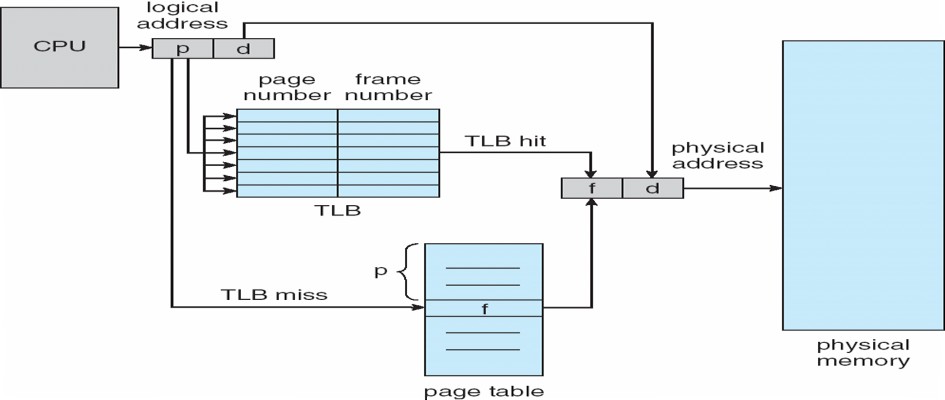
## Associative Memory

Associative memory – parallel search Address translation (p, d)

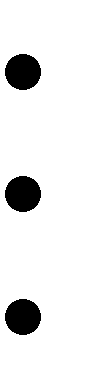
If p is in associative register, get frame # out Otherwise get frame # from page table in memory

|  |  |
| --- | --- |
| Page # | Frame # |
|  |  |
|  |  |
|  |  |
|  |  |

## Paging Hardware With TLB



**Effective Access Time**

Associative Lookup = e 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 = an **Effective Access Time** (EAT)

EAT = (1 + e) a + (2 + e)(1 – a)

= 2 + e – a

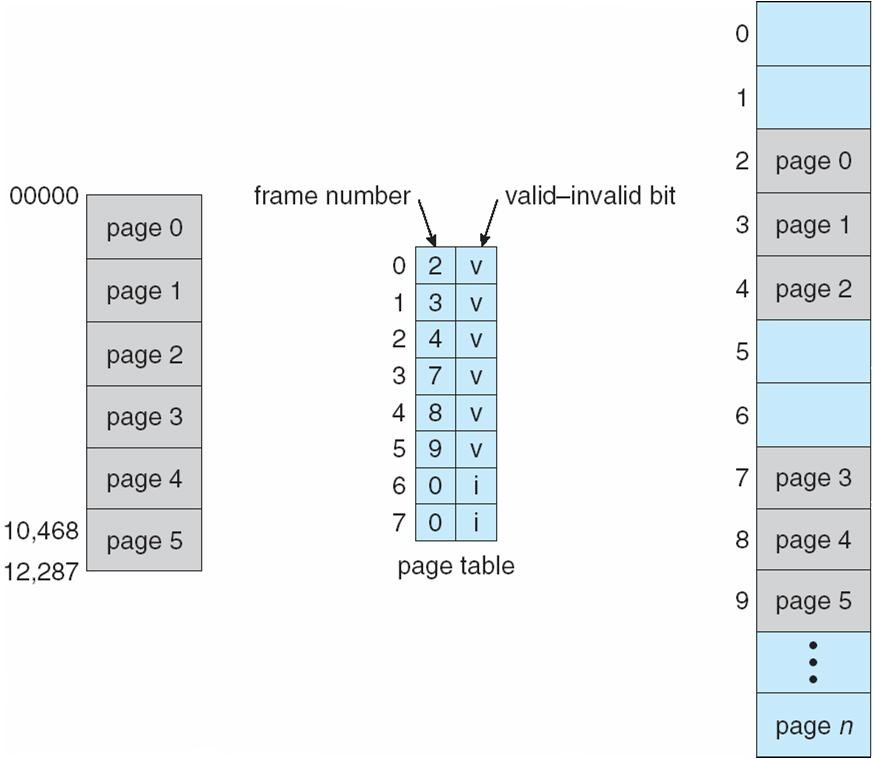
## Memory Protection

Memory protection implemented by associating protection bit with each frame

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

“valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page “invalid” indicates that the page is not in the process’ logical address space

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



**Shared Pages Shared code**

 One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).

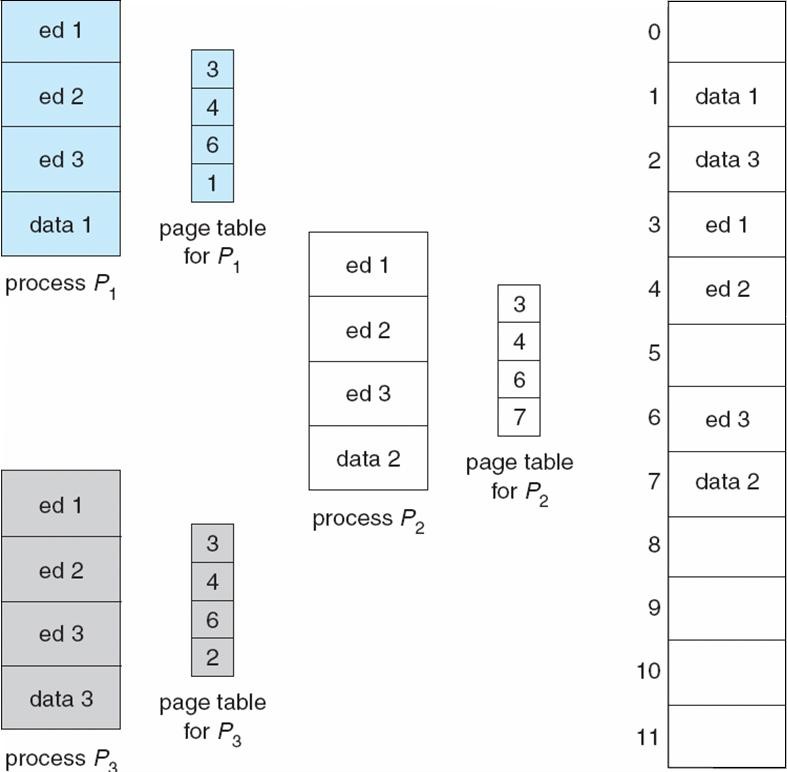
 Shared code must appear in same location in the logical address space of all processes

## Private code and data

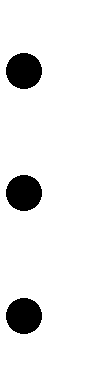
Each process keeps a separate copy of the code and data

The pages for the private code and data can appear anywhere in the logical address space

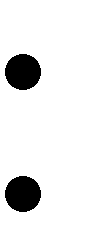
## Shared Pages Example



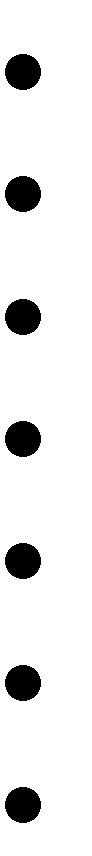
**Structure of the Page Table**

Hierarchical Paging Hashed Page Tables Inverted Page Tables

## Hierarchical Page Tables

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

## Two-Level Page-Table Scheme

**Two-Level Paging Example**

A logical address (on 32-bit machine with 1K page size) is divided into: a page number consisting of 22 bits

a page offset consisting of 10 bits

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

a 12-bit page number a 10-bit page offset

Thus, a logical address is as follows:

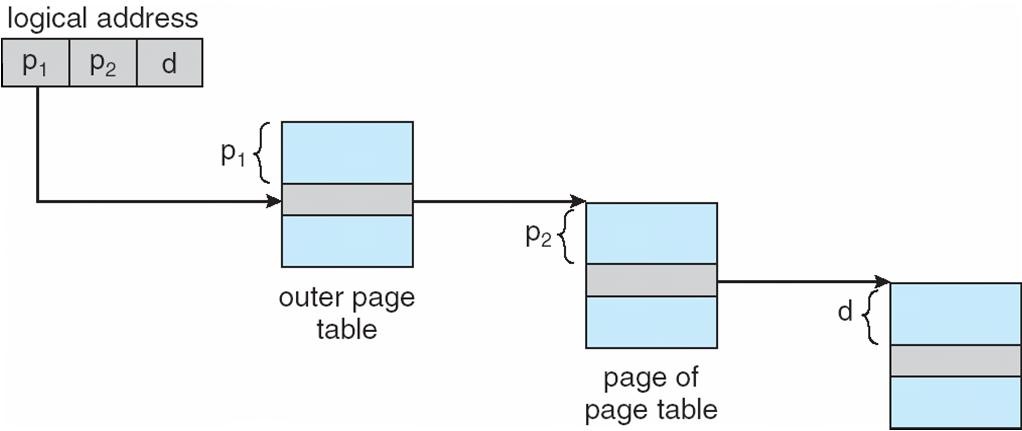
where *pi* is an index into the outer page table, and *p2* is the displacement within the page of the outer page table

# p

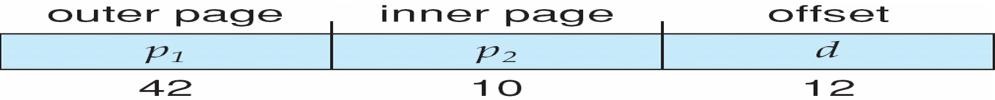
|  |  |  |
| --- | --- | --- |
| age number | | page offset |
| *p*i | *p*2 | *d* |

12 10 10

## Address-Translation Scheme

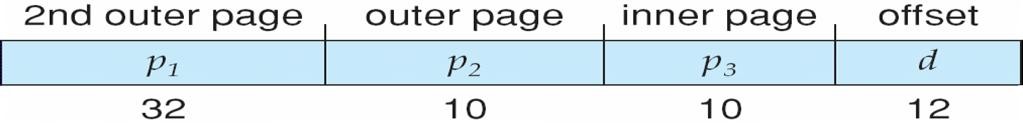


**Three-level Paging Scheme**

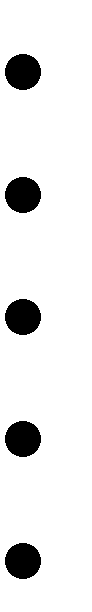


Page 78

Dept. of Computer Science and Engineering



**Hashed Page Tables**

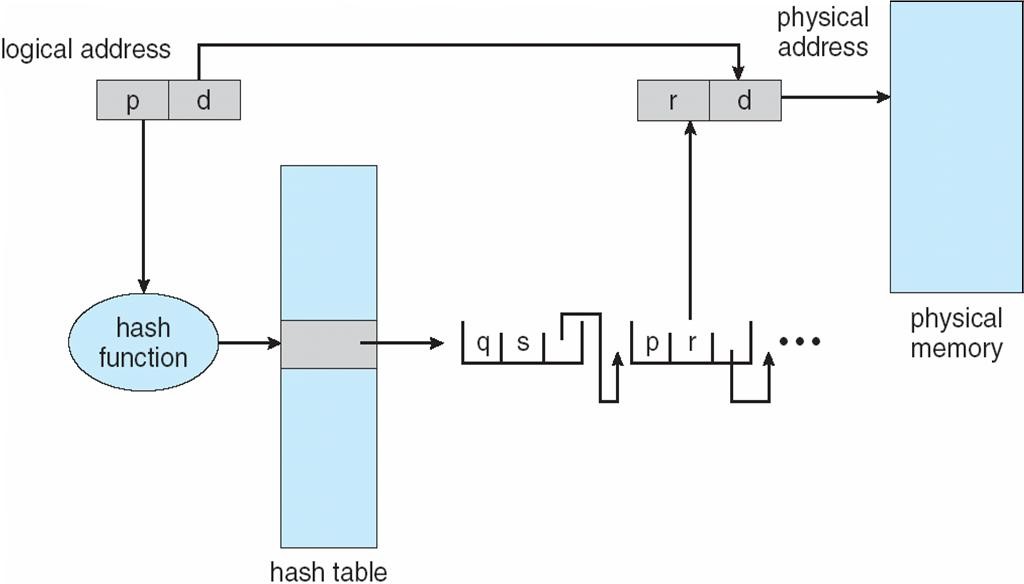
Common in address spaces > 32 bits

The virtual page number is hashed into a page table

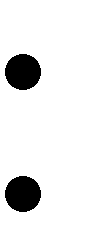
This page table contains a chain of elements hashing to the same location Virtual page numbers are compared in this chain searching for a match

If a match is found, the corresponding physical frame is extracted

**Hashed Page Table**



## Inverted Page Table

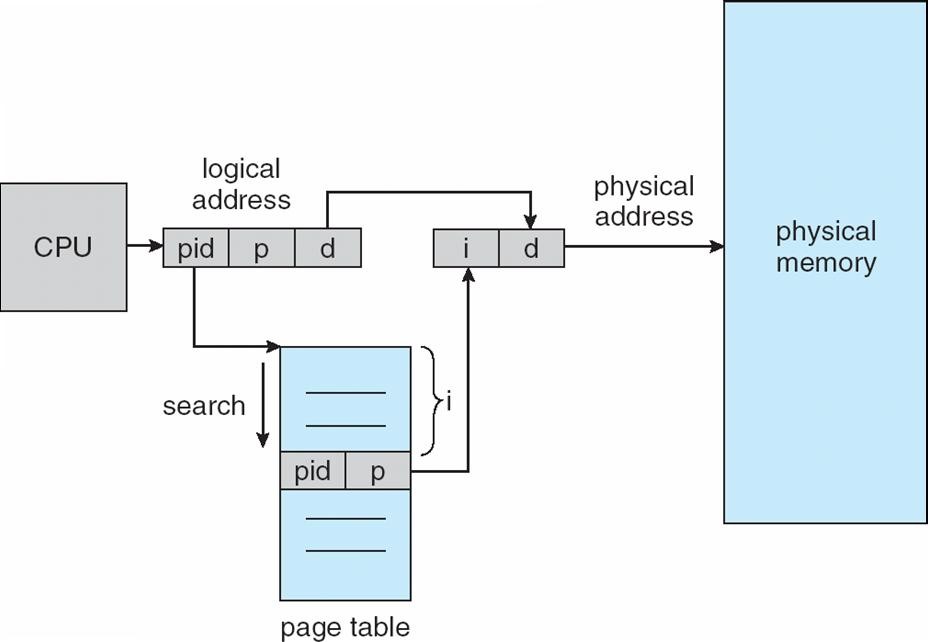
One entry for each real page of memory

Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page

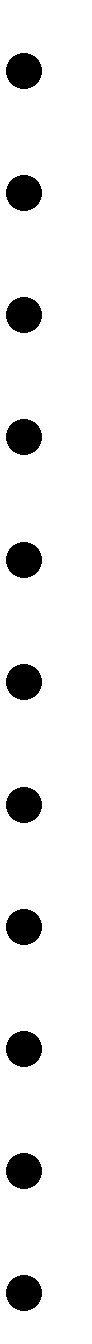
 Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs

Use hash table to limit the search to one — or at most a few — page-table entries

## Inverted Page Table Architecture



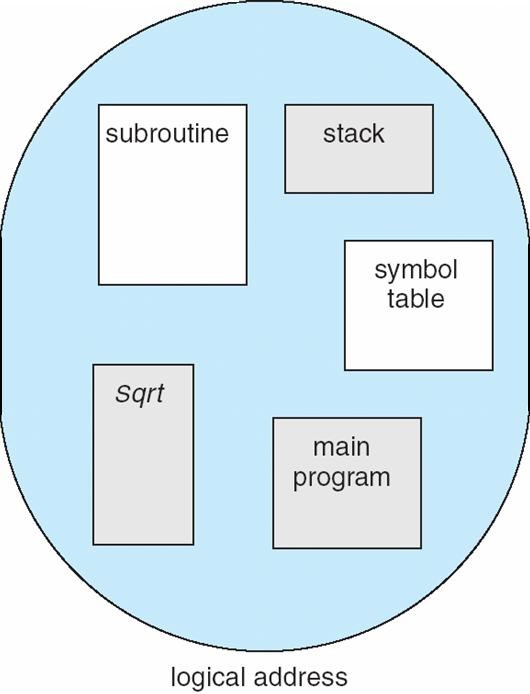
**Segmentation**

Memory-management scheme that supports user view of memory A program is a collection of segments

A segment is a logical unit such as: main program

procedure function method

object

local variables, global variables common block

stack symbol table arrays

## User’s View of a Program

**Logical View of Segmentation**

|  |
| --- |
| 1 |
| 4 |
|  |
|  |
| 2 |
| 3 |
|  |

4

3

2

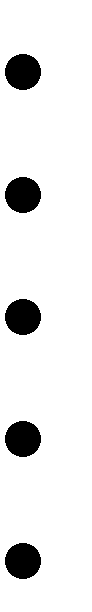
1

user space

**Segmentation Architecture**

 Logical address consists of a two tuple:

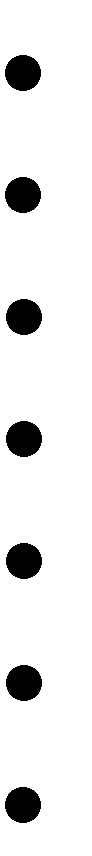
o <segment-number, offset>,

**Segment table** – maps two-dimensional physical adpdrhesysess;iecaachl tambleeemntroy rhyas:space

**base** – contains the starting physical address where the segments reside in memory

**limit** – specifies the length of the segment

**Segment-table base register (STBR)** points to the segment table’s location in memory

**Segment-table length register (STLR)** indicates number of segments used by a program; segment number ***s*** is legal if ***s*** < **STLR**

Protection

With each entry in segment table associate:

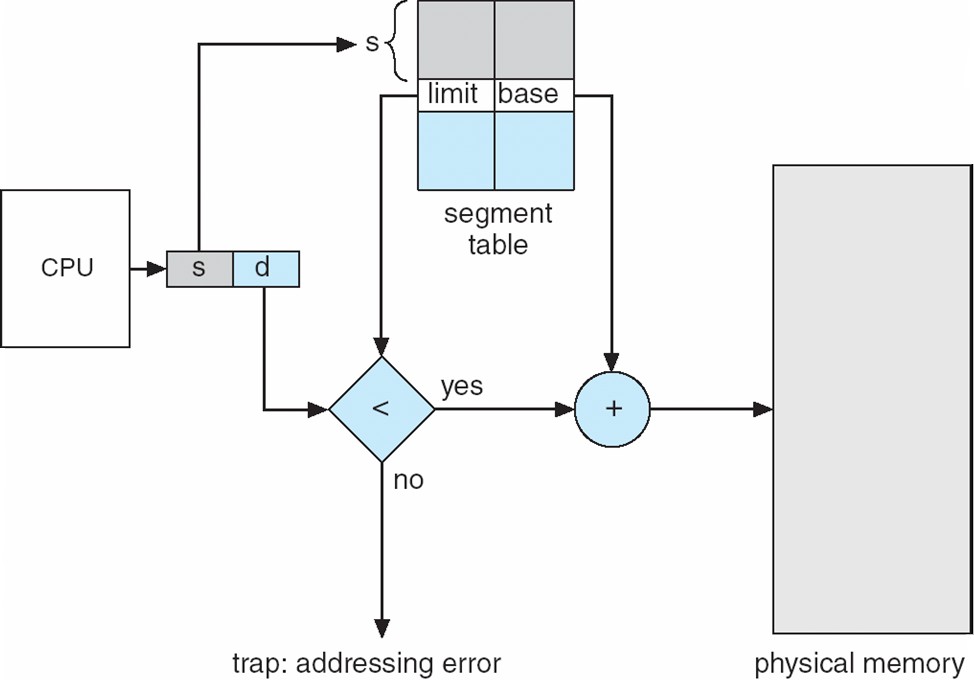
validation bit = 0 Þ illegal segment read/write/execute privileges



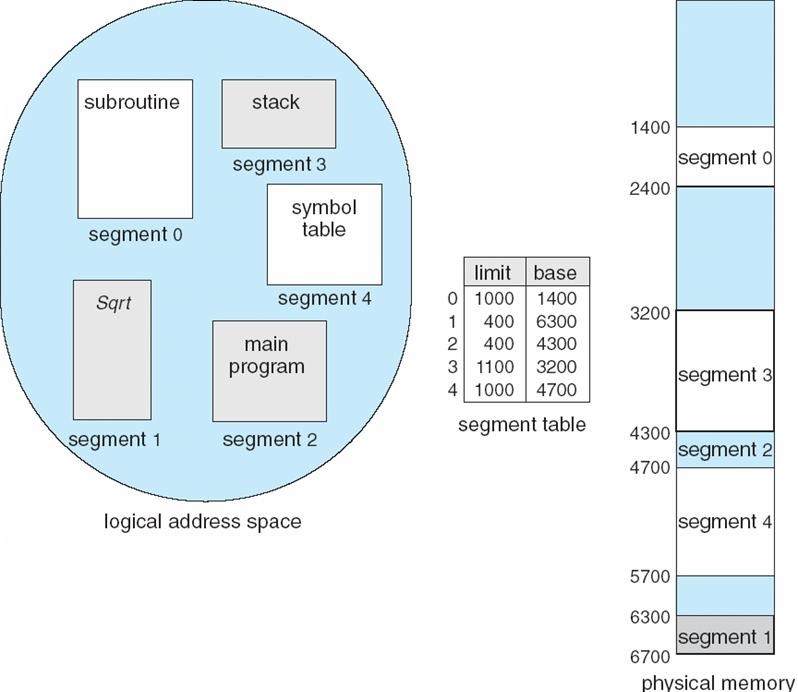
Protection bits associated with segments; code sharing occurs at segment level

Since segments vary in length, memory allocation is a dynamic storage-allocation problem A segmentation example is shown in the following diagram

## Segmentation Hardware



**Example of Segmentation**



## Example: The Intel Pentium

Supports both segmentation and segmentation with paging CPU generates logical address

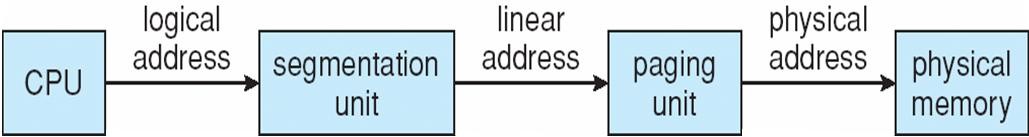
Given to segmentation unit

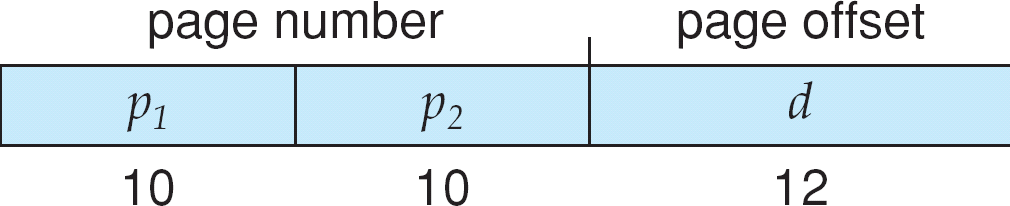
Which produces linear addresses  Linear address given to paging unit



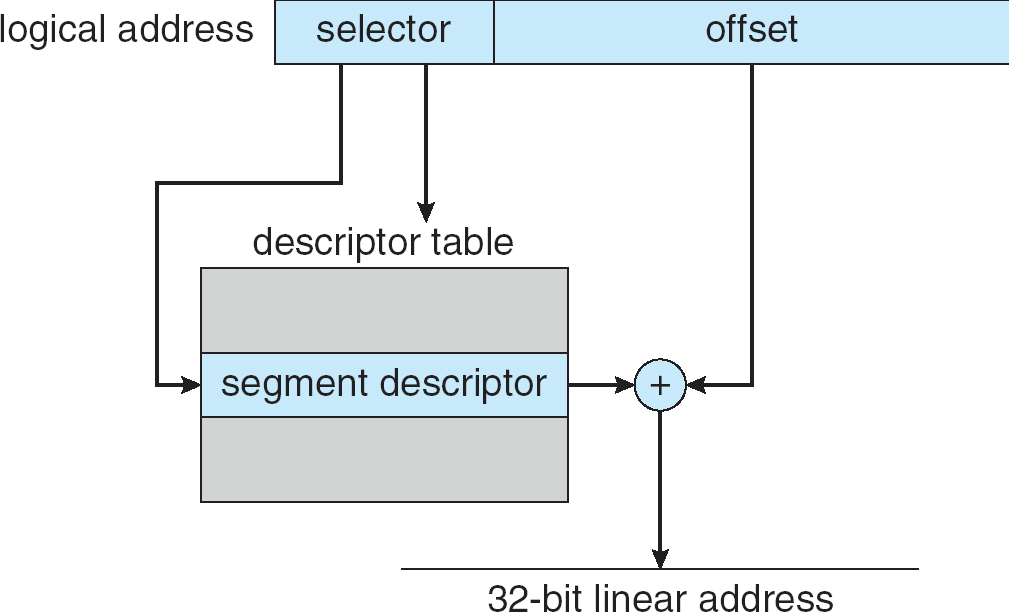
Which generates physical address in main memory Paging units form equivalent of MMU

## Logical to Physical Address Translation in Pentium

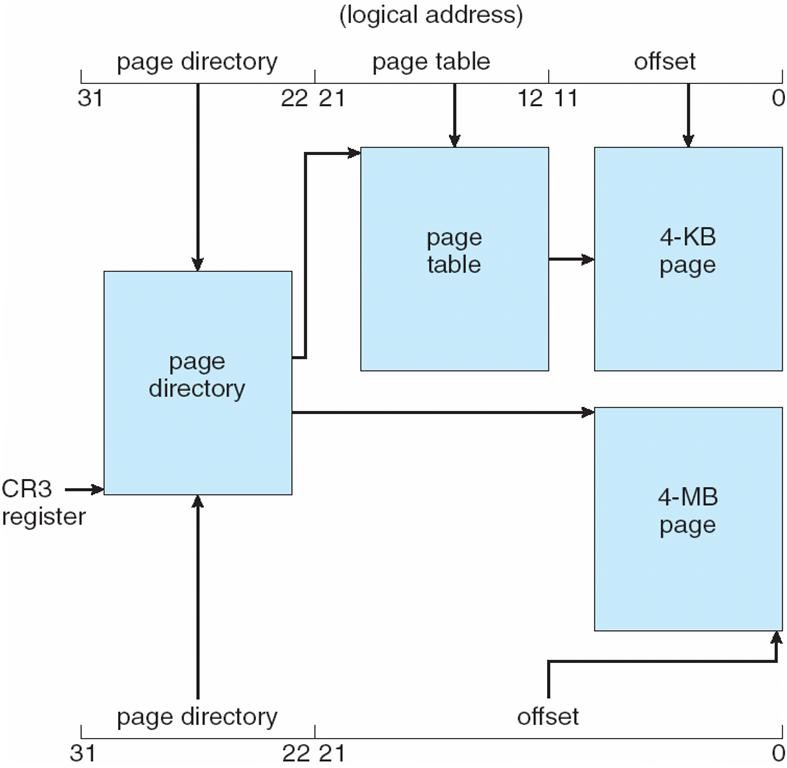




**Intel Pentium Segmentation**



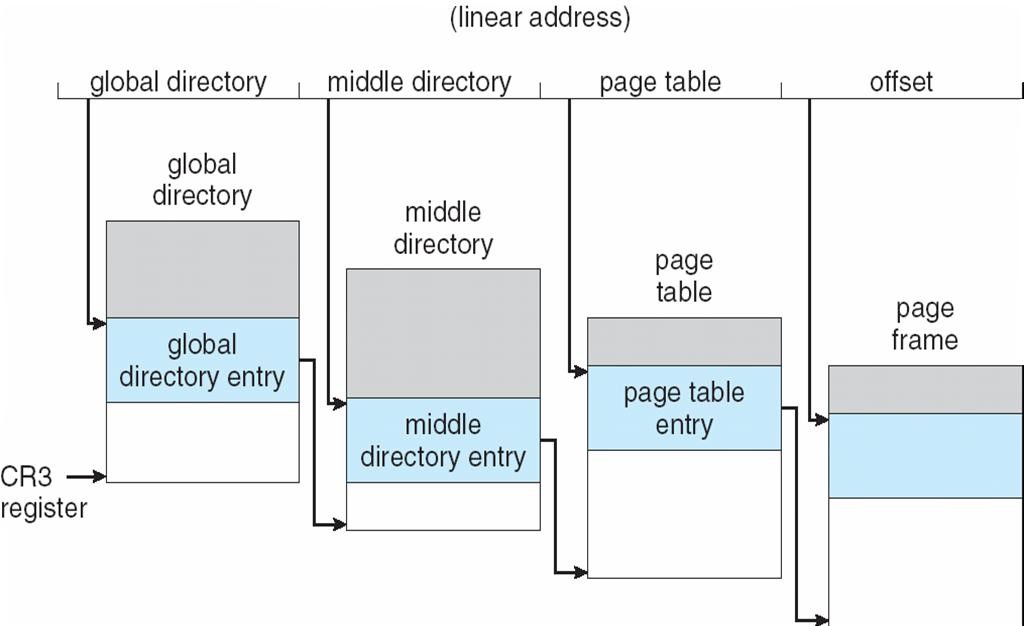
**Pentium Paging Architecture**

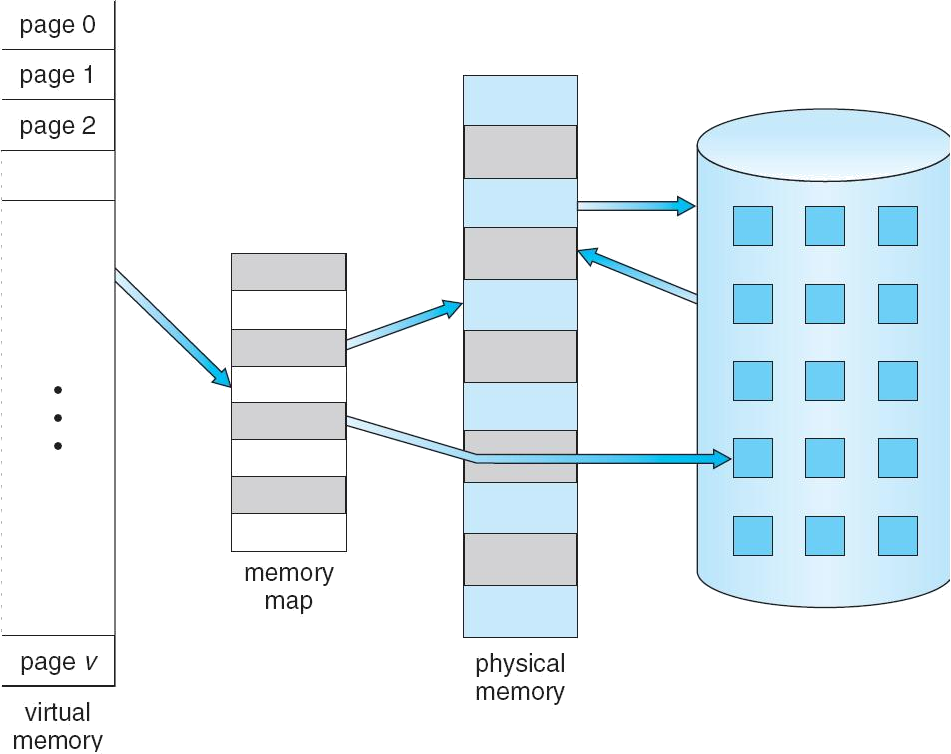


**Linear Address in Linux**



**Three-level Paging in Linux**



**UNIT – 5**

**VIRTUAL MEMORY**

**Objective**

 To describe the benefits of a virtual memory system.

 To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames.

 To discuss the principle of the working-set model.

## Virtual Memory

 Virtual memory is a technique that allows the execution of process that may not be completely in memory. The main visible advantage of this scheme is that programs can be larger than physical memory.

 Virtual memory is the separation of user logical memory from physical memory this separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available ( Fig ).

 Following are the situations, when entire program is not required to load fully.

* 1. User written error handling routines are used only when an error occurs in the data or computation.
  2. Certain options and features of a program may be used rarely.
  3. Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.

 The ability to execute a program that is only partially in memory would counter many benefits.

1. Less number of I/O would be needed to load or swap each user program into memory.
2. A program would no longer be constrained by the amount of physical memory that is available.
3. Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

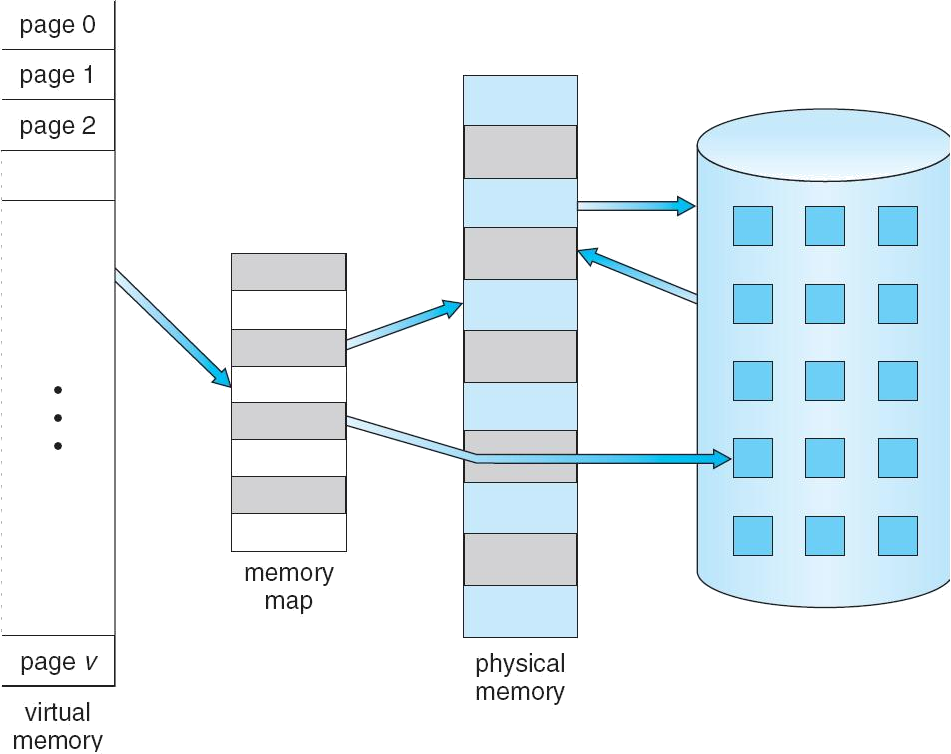


Fig. Diagram showing virtual memory that is larger than physical memory.

Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

## Demand Paging

A demand paging is similar to a paging system with swapping(Fig 5.2). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory.

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used in anyway, decreasing the swap time and the amount of physical memory needed.

Hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk using the valid-invalid bit scheme. Where valid and invalid pages can be checked checking the bit and marking a page will have no effect if the process never attempts to access the pages. While the process executes and accesses pages that are memory resident, execution proceeds normally.

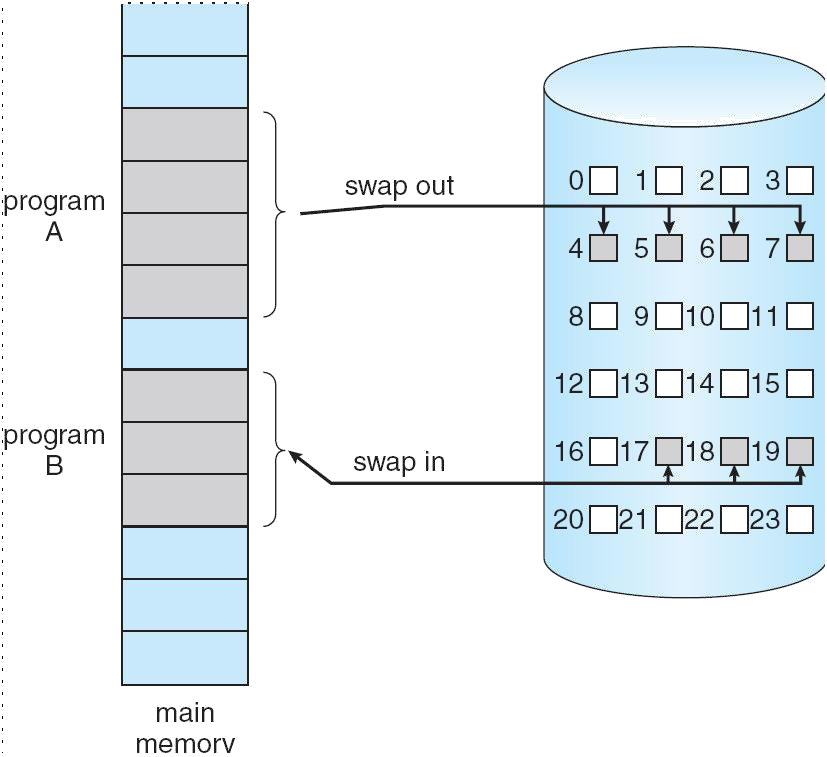


Fig. Transfer of a paged memory to continuous disk space

Access to a page marked invalid causes a page-fault trap. This trap is the result of the operating system's failure to bring the desired page into memory. But page fault can be handled as following (Fig 5.3):

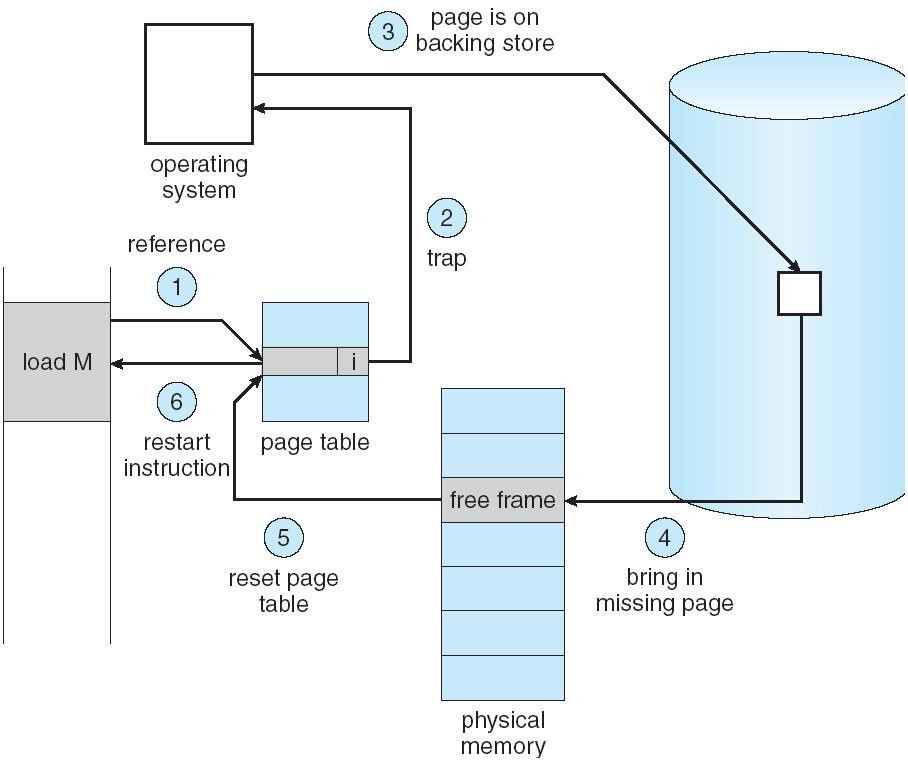


Fig. Steps in handling a page fault

1. We check an internal table for this process to determine whether the reference was a valid or invalid memory access.
2. If the reference was invalid, we terminate the process. If .it was valid, but we have not yet brought in that page, we now page in the latter.
3. We find a free frame.
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been memory.

Therefore, the operating system reads the desired page into memory and restarts the process as though the page had always been in memory.

The page replacement is used to make the frame free if they are not in used. If no frame is free then other process is called in.

## Advantages of Demand Paging:

1. Large virtual memory.
2. More efficient use of memory.
3. Unconstrained multiprogramming. There is no limit on degree of multiprogramming.

## Disadvantages of Demand Paging:

1. Number of tables and amount of processor over head for handling page interrupts are greater than in the case of the simple paged management techniques.
2. due to the lack of an explicit constraints on a jobs address space size.

## Page Replacement Algorithm

There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults. The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data.

1. For a given page size we need to consider only the page number, not the entire address.
2. if we have a reference to a page p, then any immediately following references to page p will never cause a page fault. Page p will be in memory after the

first reference; the immediately following references will not fault.

Eg:- consider the address sequence

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102,

0103, 0104, 0104, 0101, 0609, 0102, 0105

and reduce to 1, 4, 1, 6,1, 6, 1, 6, 1, 6, 1

To determine the number of page faults for a particular reference string and page replacement algorithm, we also need to know the number of page frames available. As the number of frames available increase, the number of page faults will decrease.

## FIFO Algorithm

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement 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. We can create a FIFO queue to hold all pages in memory.

The first three references (7, 0, 1) cause page faults, and are brought into these empty eg. 7, 0, 1, 2, 0, 3,

0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1 and consider 3 frames. This replacement means that the next reference to 0 will fault. Page 1 is then replaced by page 0.

## Optimal Algorithm

An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN. It is simply

Replace the page that will not be used for the longest period of time.

Now consider the same string with 3 empty frames.

The reference to page 2 replaces page 7, because 7 will not be used until reference 15, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. Optimal replacement is much better than a FIFO.

The optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

## LRU Algorithm

The FIFO algorithm uses the time when a page was brought into memory; the OPT algorithm uses the time when a page is to be used. In LRU replace the page that has not been used for the longest period of time.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time.

Let SR be the reverse of a reference string S, then the page-fault rate for the OPT algorithm on S is the same as the page-fault rate for the OPT algorithm on SR.

## LRU Approximation Algorithms

Some systems provide no hardware support, and other page-replacement algorithm. Many systems provide some help, however, in the form of a reference bit. The reference bit for a page is set, by the hardware, whenever that page is referenced. Reference bits are associated with each entry in the page table Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware.

## Additional-Reference-Bits Algorithm

The operating system shifts the reference bit for each page into the high-order or of its 5-bit byte, shifting the other bits right 1 bit, discarding the low-order bit.

These 5-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, then the page has not been

used for eight time periods; a page that is used at least once each period would have a shift register value of 11111111.

## Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page gets a second chance, its reference bit is cleared and its arrival e is reset to the current time.

## Enhanced Second-Chance Algorithm

The second-chance algorithm described above can be enhanced by considering troth the reference bit and the modify bit as an ordered pair.

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

## Counting Algorithms

There are many other algorithms that can be used for page replacement.

* **LFU Algorithm:** The least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. 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.
* **MFU Algorithm:** The most frequently used (MFU) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

## Page Buffering Algorithm

When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out.

This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

When the FIFO replacement algorithm mistakenly replaces a page mistakenly replaces a page that is still in active use, that page is quickly retrieved from the free-frame buffer, and no I/O is necessary. The free-frame buffer provides protection against the relatively poor, but simple, FIFO replacement algorithm.

**UNIT VI**

**Principles of deadlock**

To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks.To present a number of different methods for preventing or avoiding deadlocks in a computer system

## The Deadlock Problem

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set

Example

System has 2 disk drives

*P*1 and *P*2 each hold one disk drive and each needs another one Example

semaphores *A* and *B*, initialized to 1

*P*0 *P1*

wait (A); wait(B)

wait (B); wait(A)

## Bridge Crossing Example

Traffic only in one direction

Each section of a bridge can be viewed as a resource

If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback) Several cars may have to be backed up if a deadlock occurs

Starvation is possible

Note – Most OSes do not prevent or deal with deadlocks

## System Model

Resource types *R*1, *R*2, . . ., *R*m

*CPU cycles, memory space, I/O devices* Each resource type *R*i has *W*i instances. Each process utilizes a resource as follows:

#### request use release

**Deadlock Characterization**

Deadlock can arise if four conditions hold simultaneously

**Mutual exclusion:** only one process at a time can use a resource

**Hold and wait:** 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

**Circular wait:** there exists a set {*P*0, *P*1, …, *P*0} 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, …, *Pn*–1 is waiting for a resource that is held by

*P*n, and *P*0 is waiting for a resource that is held by *P*0.

n

## Resource-Allocation Graph

### A set of vertices *V* and a set of edges *E*

V is partitioned into two types:

*P* = {*P*1, *P*2, …, *Pn*}, the set consisting of all the processes in the system *R* = {*R*1, *R*2, …, *Rm*}, the set consisting of all resource types in the system request edge – directed edge *P*1 ® *Rj*

assignment edge – directed edge *Rj* ® *Pi*

Process

Resource Type with 4 instances

*Pi* requests instance of *Rj*n

*Pi*

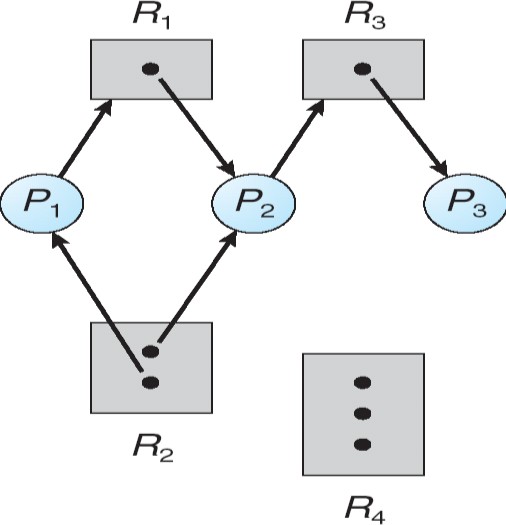
*Rj*

*Pi* is holding an instance of *Rj*

*Pi*

*Rj*

**Example of a Resource Allocation Graph**



**Resource Allocation Graph With A Deadlock**

