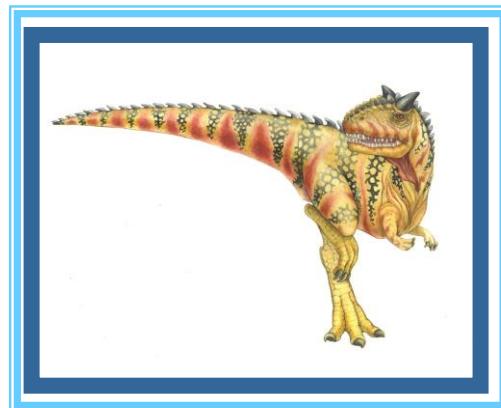


Chapter 6: Synchronization



Synchronization

Background

The Critical-Section Problem

Peterson's Solution

Synchronization Hardware

Semaphores

Classic Problems of Synchronization

Monitors

Synchronization Examples

Atomic Transactions

Objectives

- 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

Background

Concurrent access to **shared data** may result in data inconsistency

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

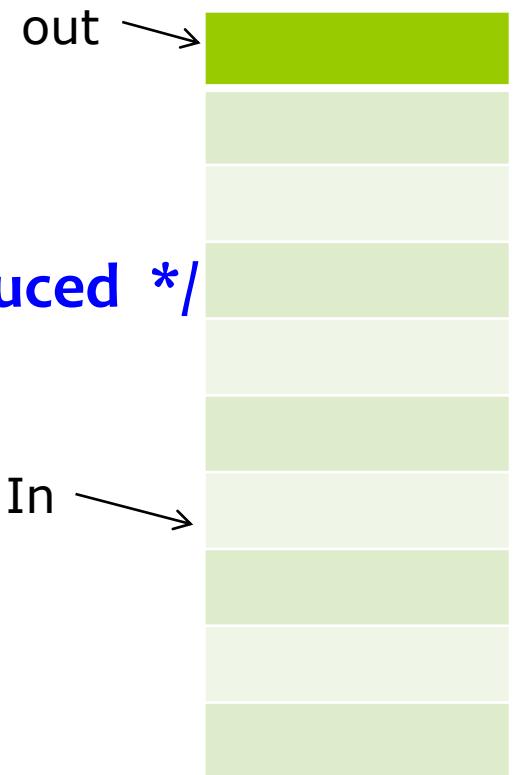
Suppose that we want to provide a solution to the consumer-producer problem that fills **all** the buffers.

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.

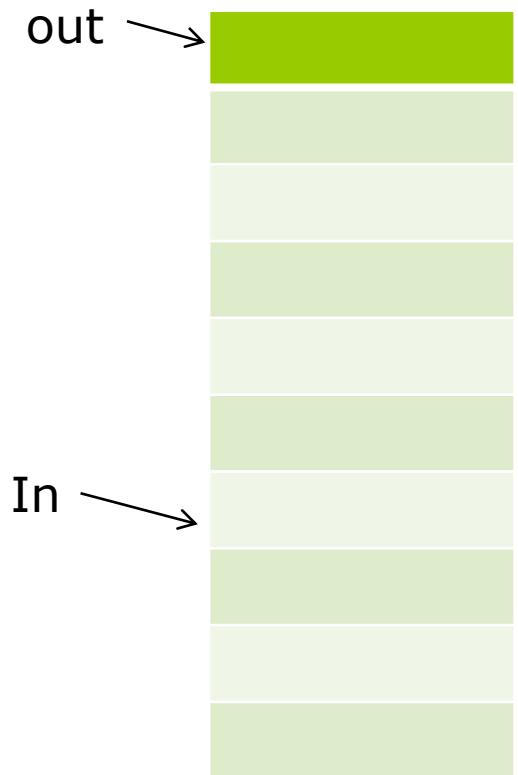
Producer

```
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
```

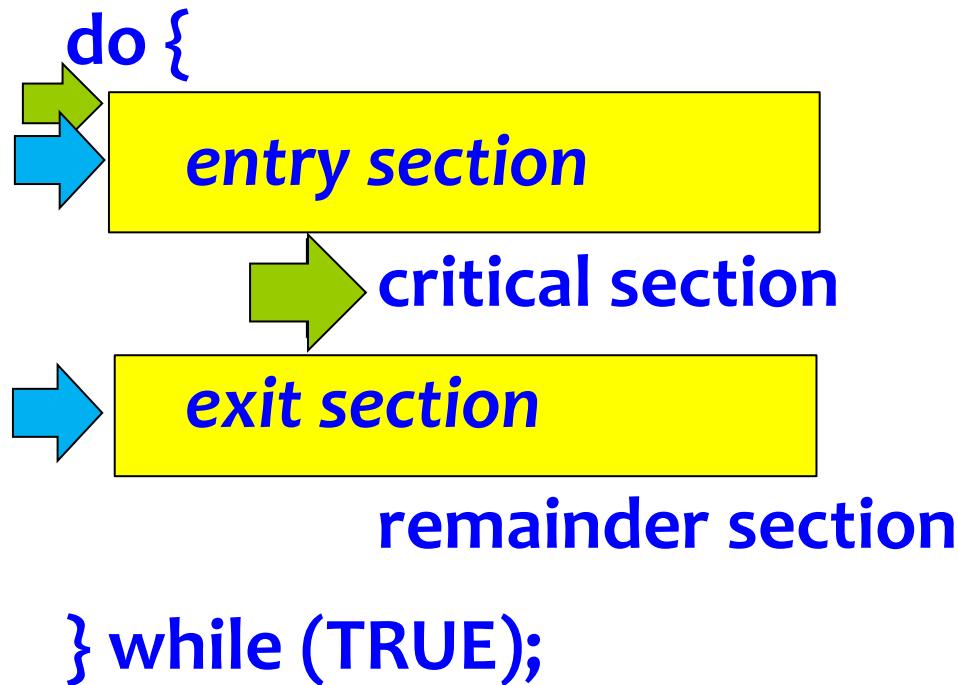
```
register2 = register2 - 1
```

```
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}

Critical-Section Problem



General structure of a typical Process P_i

Solution to Critical-Section Problem

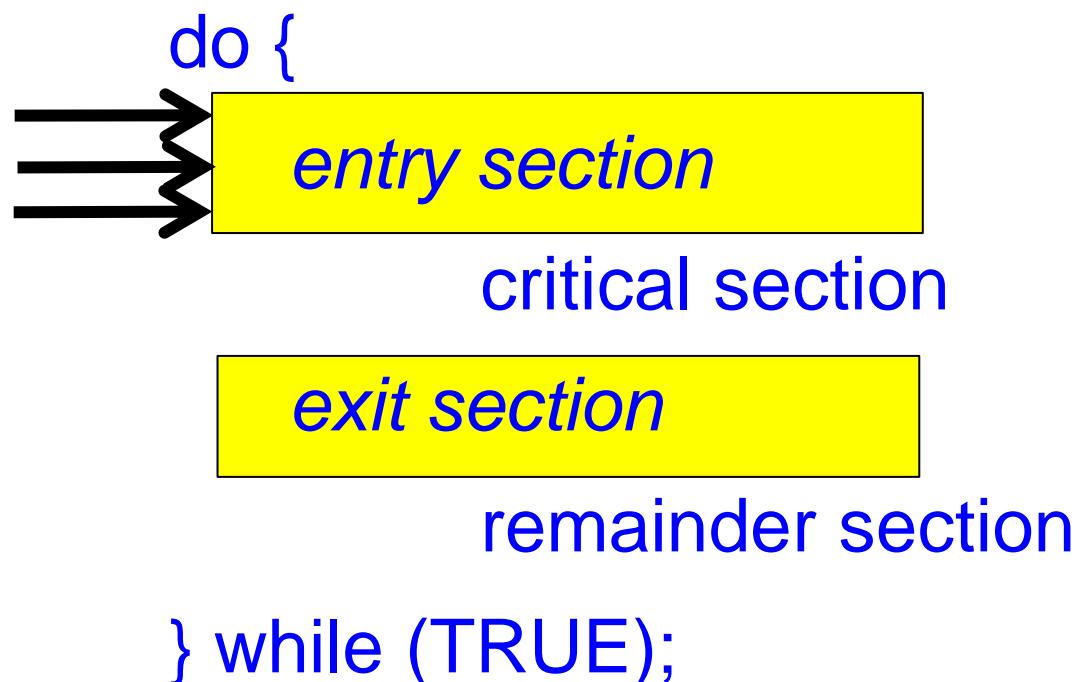
1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

```
do {  
    entry section  
    → critical section  
    exit section  
    remainder section  
} while (TRUE);
```

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (TRUE);
```

Solution to Critical-Section Problem

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**



Solution to Critical-Section Problem

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 P_i is ready!

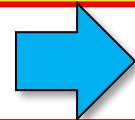
Algorithm for Process P_i

do {

 flag[i] = TRUE;

 turn = j;

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



 critical section

 flag[i] = FALSE;

 remainder section

} while (TRUE);

Prove this algorithm is correct

1. Mutual exclusion is preserved
2. The progress requirement is satisfied.
3. The bounded waiting requirement is met

Prove this algorithm is correct

1. Mutual exclusion is preserved

do {

flag[i] = TRUE;

turn = j;

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

→ critical section

flag[i] = FALSE;

remainder section

} while (TRUE);

do {

flag[j] = TRUE;

turn = i;

while (flag[i] && turn == i);

critical section

flag[j] = FALSE;

remainder section

} while (TRUE);

Prove this algorithm is correct

2. The progress requirement is satisfied.

```
do {
```

```
    flag[i] = TRUE;
```

```
    turn = j;
```

```
    → while (flag[j] && turn == j);
```

```
        → critical section
```

```
    → flag[i] = FALSE;
```

```
        remainder section
```

```
} while (TRUE);
```

```
do {
```

```
    flag[j] = TRUE;
```

```
    turn = i;
```

```
    while (flag[i] && turn == i);
```

```
        → critical section
```

```
    flag[j] = FALSE;
```

```
        remainder section
```

```
} while (TRUE);
```

Prove this algorithm is correct

3. The bounded waiting requirement is met

```
do {
```

```
    flag[i] = TRUE;
```

```
    turn = j;
```

```
    while (flag[j] && turn == j);
```

→ critical section

```
    flag[i] = FALSE;
```

remainder section

```
} while (TRUE);
```

```
do {
```

```
    flag[j] = TRUE;
```

```
    turn = i;
```

```
    while (flag[i] && turn == i);
```

→ critical section

```
    flag[j] = FALSE;
```

remainder section

```
} while (TRUE);
```

Synchronization Hardware

Any solution to the critical-section problem requires a simple tool – a **lock**.

Race conditions are prevented by requiring that critical regions be protected by locks

```
do {  
    → acquire lock  
        → critical section  
        release lock  
    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

TestAndSet Instruction

Definition:

```
boolean TestAndSet (boolean *target)
{
    → boolean rv = *target; /* Test */
    → *target = TRUE;      /* Set */
    return rv;
}
```



Solution using TestAndSet

Shared boolean variable **lock**., initialized to false.

Solution (Mutual-Exclusion):

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**

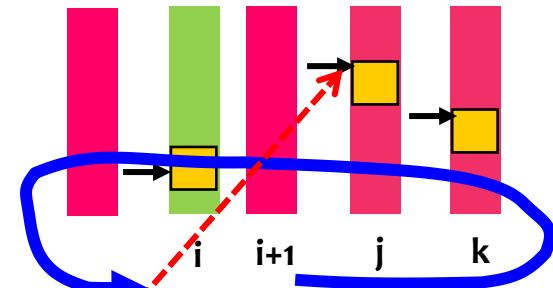
Solution(Mutual-Exclusion):

```
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]) (Find next waiting process)
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE; (No one is waiting)
    else
        waiting[j] = FALSE; (process j enters next)
    // remainder section
} while (TRUE);
```



Prove this algorithm is correct

1. Mutual exclusion is preserved
2. The progress requirement is satisfied.
3. The bounded waiting requirement is met

Semaphores

The hardware-based solutions for the CS problem are complicated for application programmers to use.

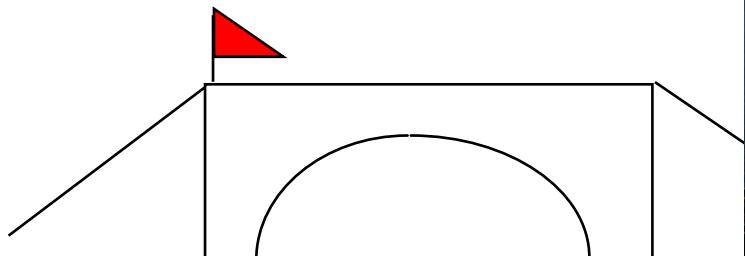
To overcome this difficulty, we use a synchronization tool called a **semaphore**.

Semaphore S – integer variable

Two standard operations modify **S**: **wait()** and **signal()**

Originally called **P()** and **V()**

Less complicated



Semaphores

Can only be accessed via two indivisible (atomic) operations

```
wait (S) {  
    while S <= 0      /* Semaphore S is occupied */  
        ; // no-op  
    S--;                /* Semaphore S is available, get it */  
}  
  
signal (S) {  
    S++;                /* Release the semaphore S */  
}
```

Semaphore Usage

Counting semaphore – integer 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 locks** as they are locks that provide **mutual exclusion**.

We can use binary semaphore to deal with the CS problem for multiple processes.

The n processes share a semaphore, **mutex**, initialized to 1

Mutual-Exclusion Implementation with semaphores

Provides mutual exclusion (for Process Pi)

```
Semaphore mutex; // initialized to 1  
do {  
    wait (mutex);  
    // Critical Section  
    signal (mutex);  
    // remainder section  
} while (TRUE);
```

Semaphore Usage

Counting semaphore can be used to control access to a given resource consisting of a finite number of instances.

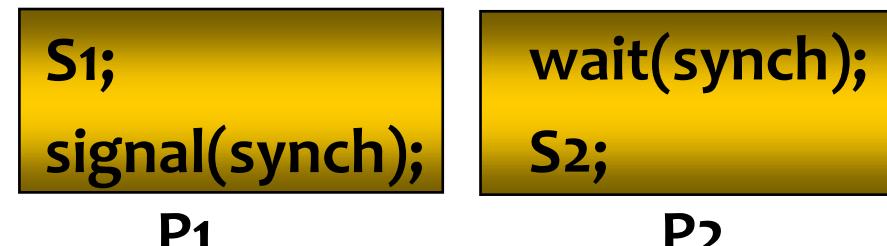
The semaphore is initialized to the number of resources available.

To use a resource, **wait()**

To release a resource, **signal()**

Semaphores can be used to solve various synchronization problem.

For example, we have two processes P1 and P2. Execute S1 and then S2: Synch = 0



Semaphore Implementation

The main disadvantage of previous mutual-exclusion solution is the **busy waiting (CPU is wasting)**.

This type of semaphore is called a **spinlock**.

To overcome this, we can use the concept of **block** and **wakeup** operations.

```
Typedef struct {  
    int value;  
    struct process *list;  
} semaphore
```

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)

- ▶ Value > 0 indicates semaphore is still available
- ▶ Value = 0 indicates semaphore is just occupied and no waiting process
- ▶ Value < 0 indicates the number of waiting processes pointer to next record in the list /* waiting 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.

Semaphore Implementation with no Busy waiting

Implementation of wait:

```
wait(semaphore *S) {  
    S.value --;  
    if (S.value < 0) {  
        add this process to S.list;  
        block();  
    }  
}
```

Semaphore Implementation with no Busy waiting

Implementation of signal:

```
signal(semaphore *S) {  
    S.value ++;  
    if (S.value <= 0) {  
        remove a process P from S.list;  
        wakeup(P);  
    }  
}
```

Semaphore Implementation with no Busy waiting

Note that the semaphore value may be **negative**.

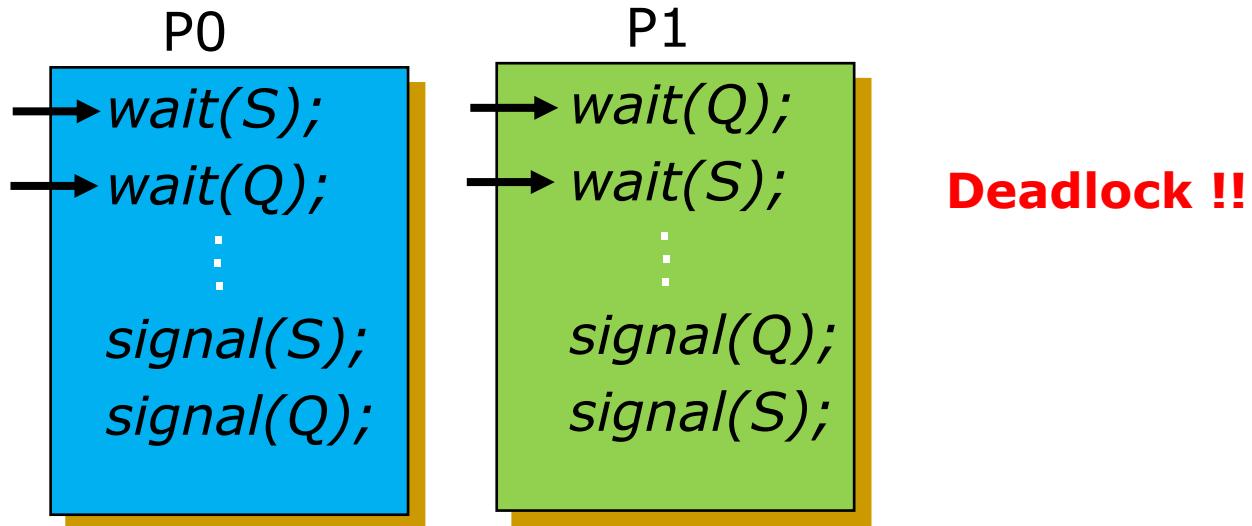
Its magnitude is the **number of processes waiting on that semaphore**.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB).

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



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

Priority Inversion

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

Three processes L, M, H with priority L < M < H

Assume process H requires resource R, which is using by process L. **Process H waits.**

Assume process M becomes runnable, thereby preempting process L.

Indirectly, a process with lower priority – M – has affected how long H must wait for L to release R.

Priority-inheritance protocol – all processes that are accessing resources needed by a higher priority process **inherit the higher priority** until they are finished with the resources.

Classical Problems of Synchronization

Bounded-Buffer Problem

Readers and Writers Problem

Dining-Philosophers Problem

Bounded-Buffer Problem

Used to illustrate the **power of synchronization primitives.**

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.

Bounded Buffer Problem (Cont.)

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);
```

Initial, Empty = N, Full = 0

The Reader and Writers Problem

A data object, such as a file or record, is to be shared among several concurrent processes.

The writers are required to have **exclusive access** to the shared object.

The readers-writers problem has several variations, all involving priorities.

The First problem -- require no reader will be kept waiting unless a writer has already obtained permission to use the shared object. Thus, no reader should wait for other readers to finish even a writer is waiting.

The Reader and Writers Problem

The Second problem -- require once a writer is ready, that writer performs its write as soon as possible, after old readers (or writer) are completed. Thus, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in **starvation**.

The first problem : Writers

- ▶ Writers wait, but readers come in one after one

The second problem : Readers

- ▶ Readers wait, but writers come in one after one

A solution for the first problem

Shared Data

Semaphore **mutex** initialized to 1

Semaphore **wrt** initialized to 1

Integer **readcount** initialized to 0

The **mutex semaphore** is used to ensure mutual exclusion when the variable **readcount** is updated.

Readcount keeps track of how many processes are currently reading the object.

The **wrt semaphore** functions as a mutual exclusion semaphore for the writers.

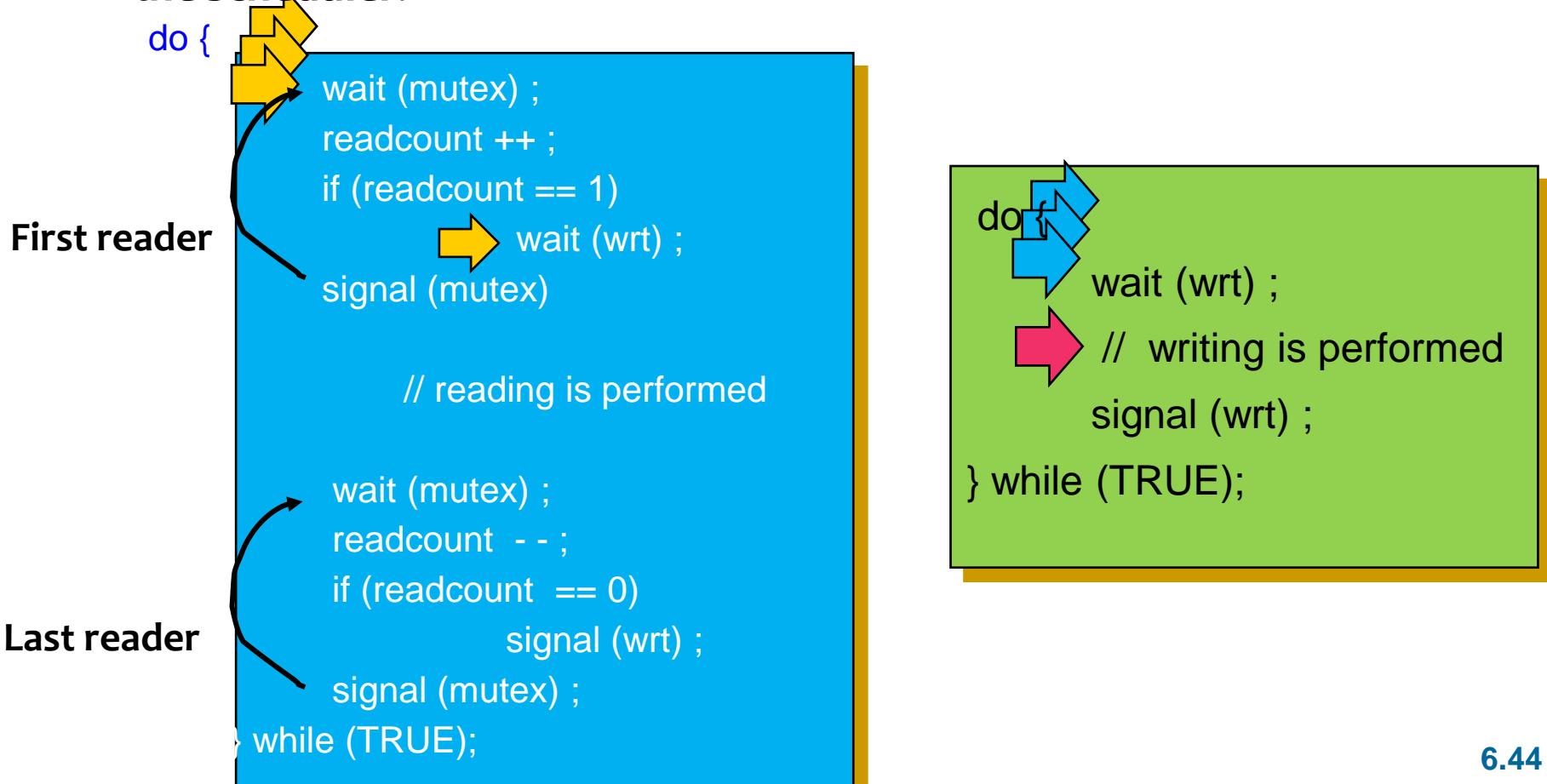
It also is used by **the first or last reader** that enters or exits the critical section.

It is not used by the readers who enter or exit while other processes are in their critical sections.

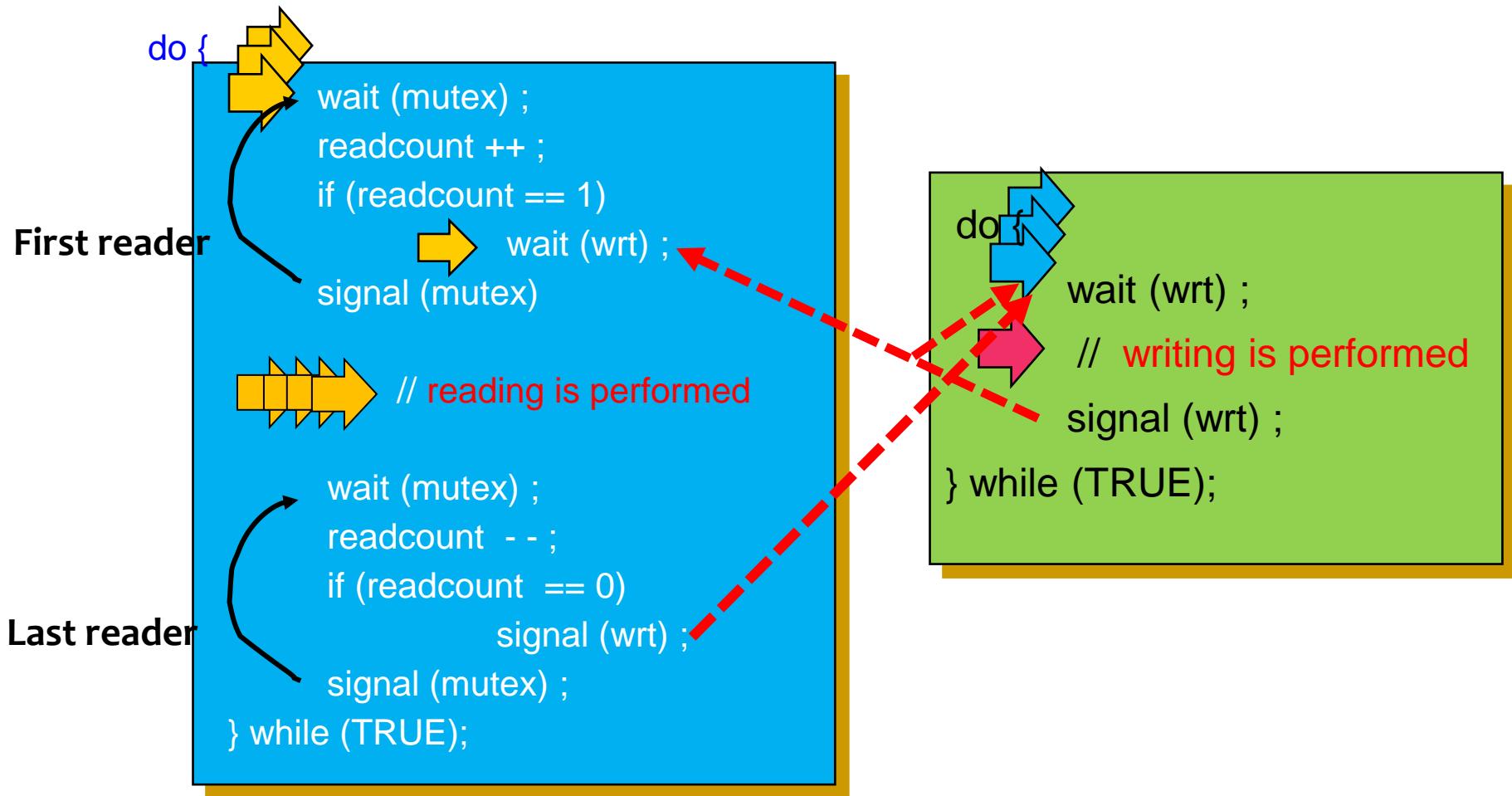
A solution for the first problem

If a writer is in the CS and n readers are waiting, then **one** reader is queued on **wrt** and **$n-1$** readers are queued on **mutex**.

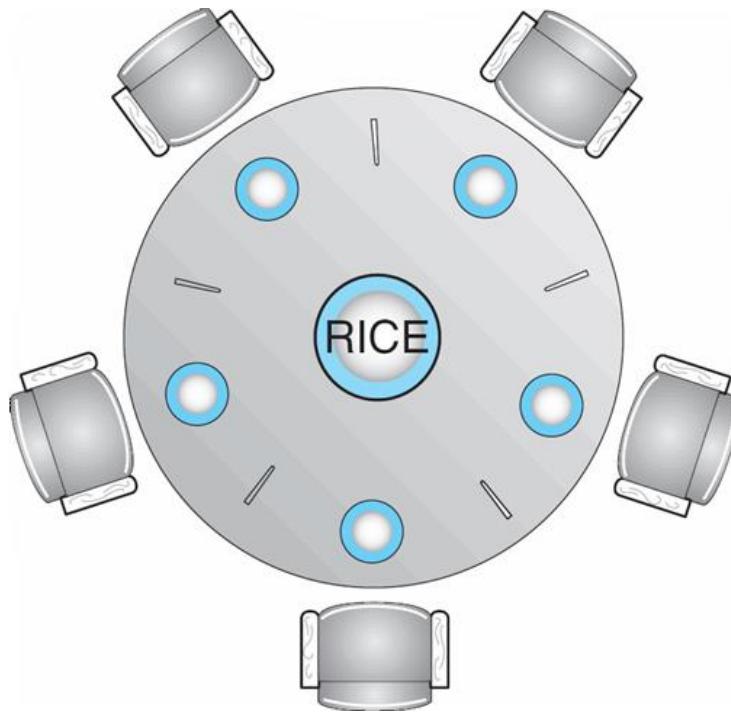
When a writer executes `signal(wrt)`, we may assume the execution of either the waiting writers or a single reader. The selection is made by the scheduler.



A solution for the first problem



6.6.3 Dining-Philosophers Problem



Shared data

Bowl of rice (data set)

Semaphore **chopstick [5] initialized to 1**

Dining-Philosophers Problem (Cont.)

Represent each chopstick by a semaphore.

Wait and Signal on the semaphores.

Var chopstick: array [0..4] of semaphores;

The structure of Philosopher *i*:

```
do {  
    wait ( chopstick[i] );  
    wait ( chopstick[ (i + 1) % 5 ] );
```

Deadlock !!

// eat

signal (chopstick[i]);

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

// think

} while (TRUE);

Several possible solutions to the deadlock problem

Allow at most **four** philosophers to be sitting simultaneously at the table.

Allow a philosopher to pick up her chopsticks only if **both chopsticks are available** (note that she must pick them up in a critical section).

Use an asymmetric solution. Thus,

an **odd philosopher** picks up first her left chopstick and then her right chopstick, whereas

an **even philosopher** picks up her right chopstick and then her left chopstick.

Problems with Semaphores

Correct use of semaphore operations: Otherwise, some problems may happen

signal (mutex) wait (mutex)

wait (mutex) ... wait (mutex)

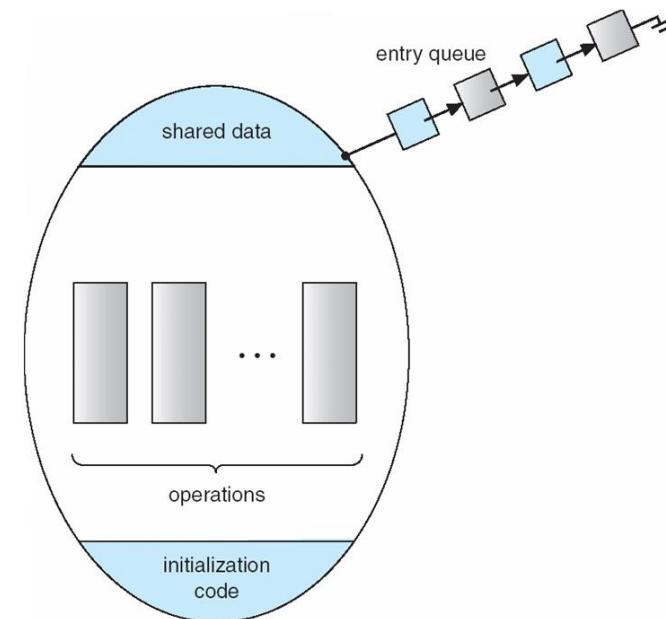
Omitting of wait (mutex) or signal (mutex) (or both)

6.7 Monitors

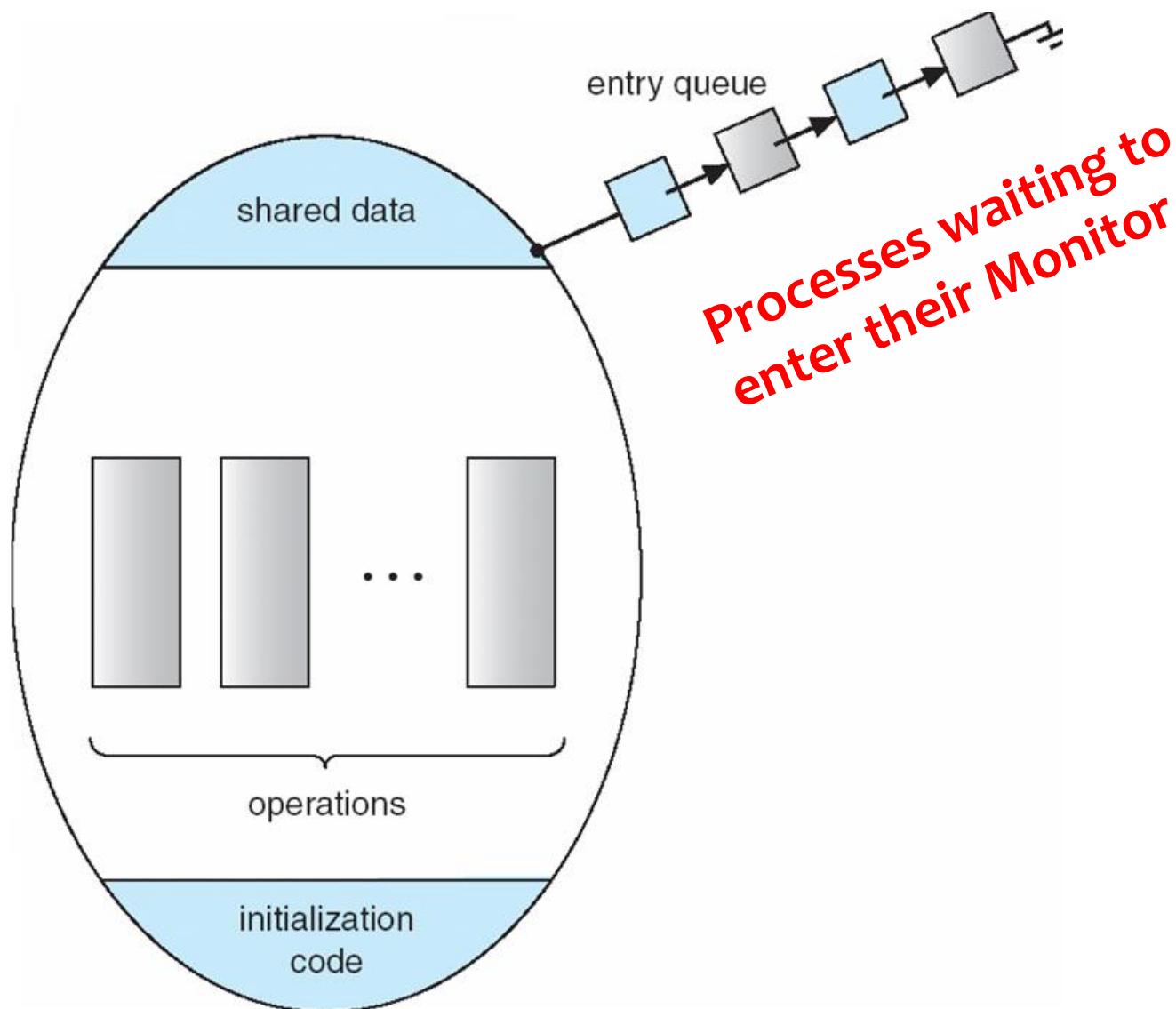
A **high-level abstraction** that provides a convenient and effective mechanism for **process synchronization**

Only one process may be active within the monitor at a time

```
monitor monitor-name
{ // shared variable declarations
  procedure P1 (...) { .... }
  ...
  procedure Pn (...) {.....}
  ...
  Initialization code ( ...) { ... }
  ...
}
```



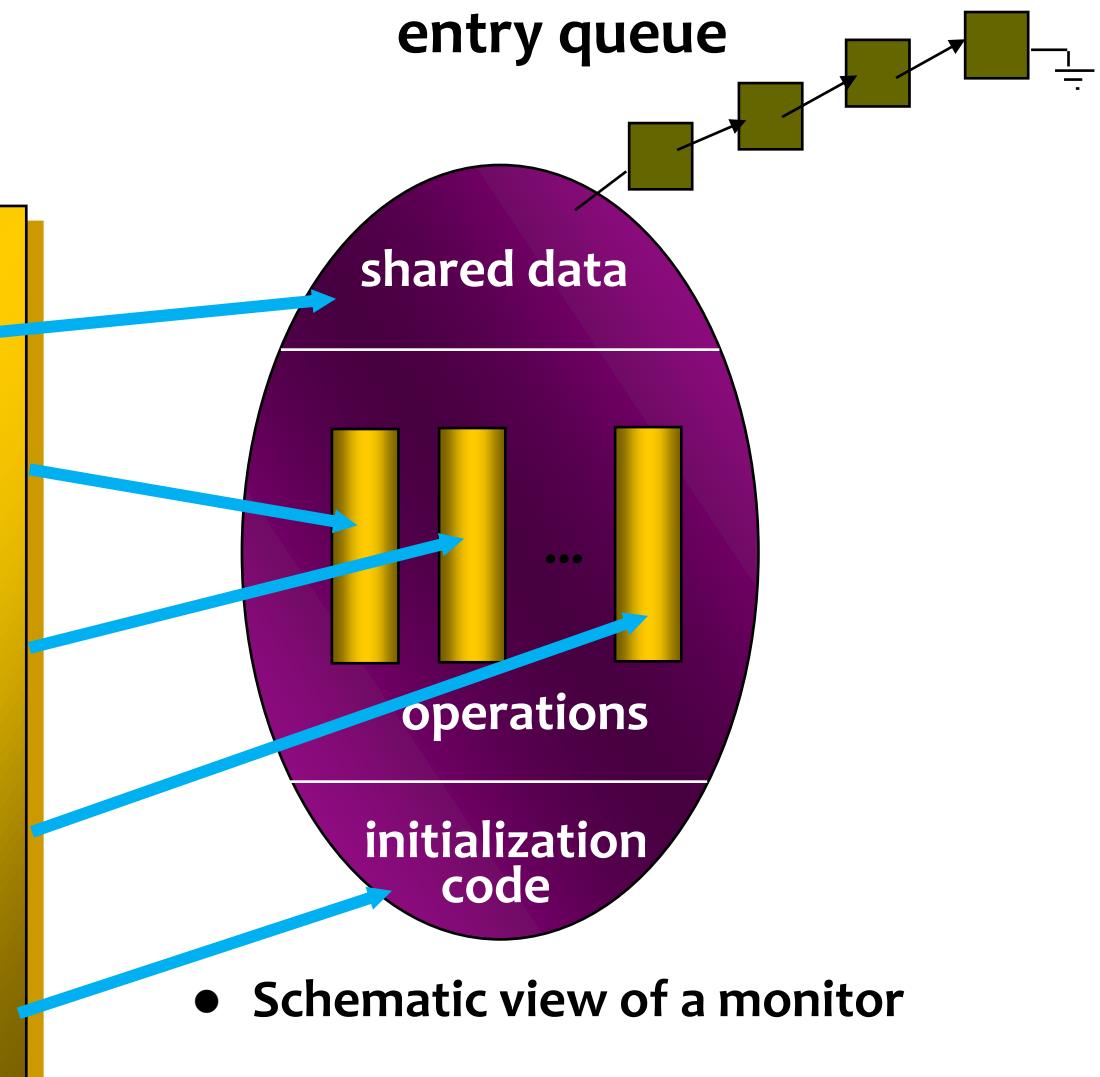
Schematic view of a Monitor



Monitors

- Syntax of a monitor

```
type monitor-name = monitor  
    variable declarations  
  
    procedure entry P1(...);  
        begin ... end;  
  
    procedure entry P2(...);  
        begin ... end;  
  
    procedure entry Pn(...);  
        begin ... end;  
begin  
    initialization code  
end.
```



Condition Construct

A programmer who needs to write **her own tailor-made synchronization scheme** can define one or more variables of type **condition**.

Var x,y : condition;

The only operations that can be invoked on a condition variable are **wait** and **signal**. For example, **x.wait**, **x.signal**.

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

Suppose P invokes **x.signal** and Q is suspended with **x**. Two possibilities exist:

P either waits Q leaves or another condition

Q either waits P leaves or another condition

Condition Variables

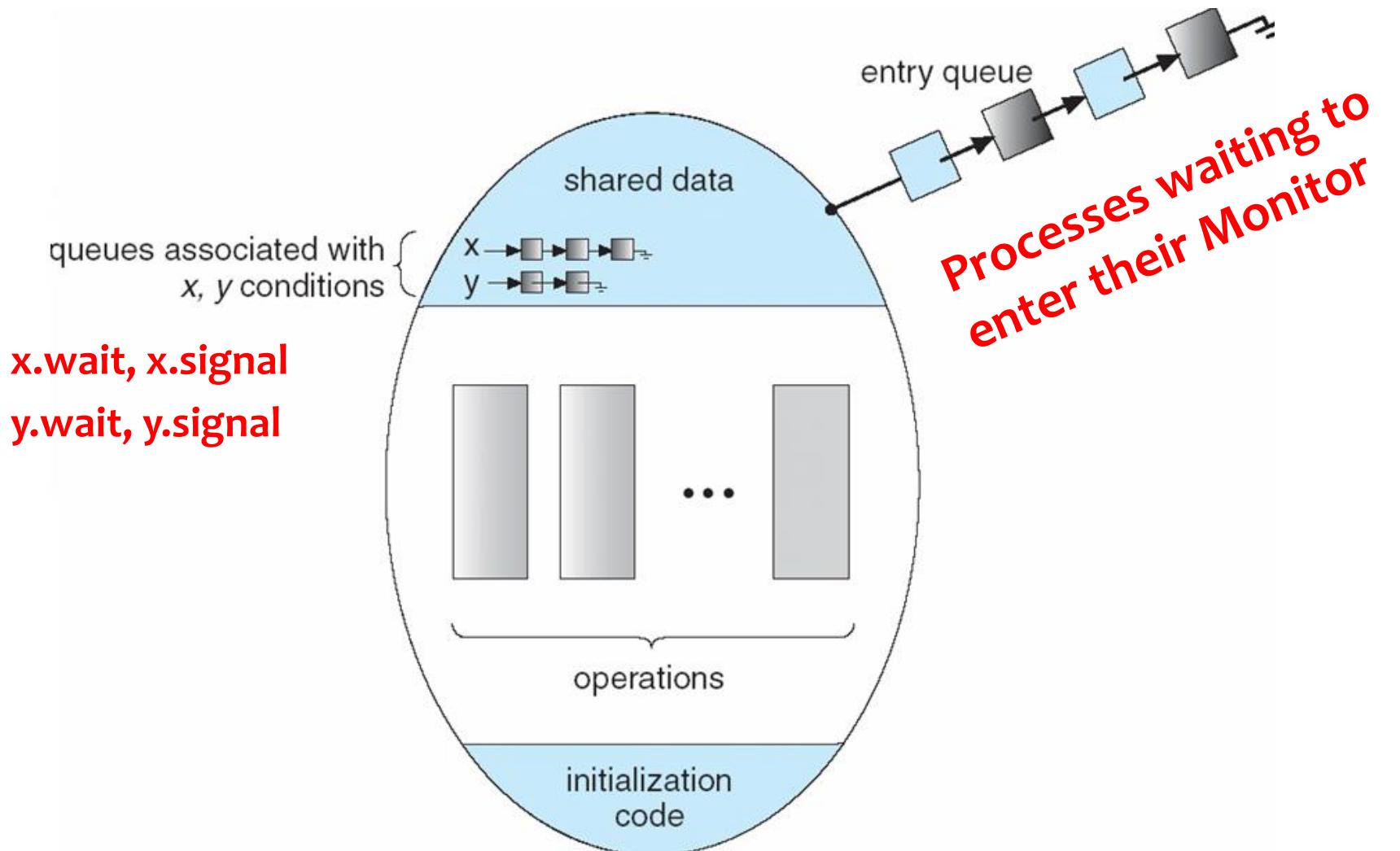
condition x, y;

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



A Deadlock-free Monitor Solution for the Dining-Philosophers Problem

A philosopher is allowed to pick up her chopsticks only if both of them are available.

Data structure:

Var state: array [0..4] of (thinking, hungry, eating);

Var self: array [0..4] of condition;

Philosopher i can **delay herself** when she is hungry, but is unable to obtain the chopsticks she needs.

Operations:

pickup and **putdown** on the instance **dp** of the **dining-philosophers monitor**

Solution to Dining Philosophers (cont)

Each philosopher i must invoke the operations **pickup()** and **putdown()** in the following sequence:

```
var dining-philosophers: dp  
  
dining-philosophers.pickup( $i$ );  
  
...  
eat  
  
...  
  
dining-philosophers.putdown( $i$ );
```

Process i

A Deadlock-free Monitor Solution for the Dining-Philosophers Problem

```
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);
}
```

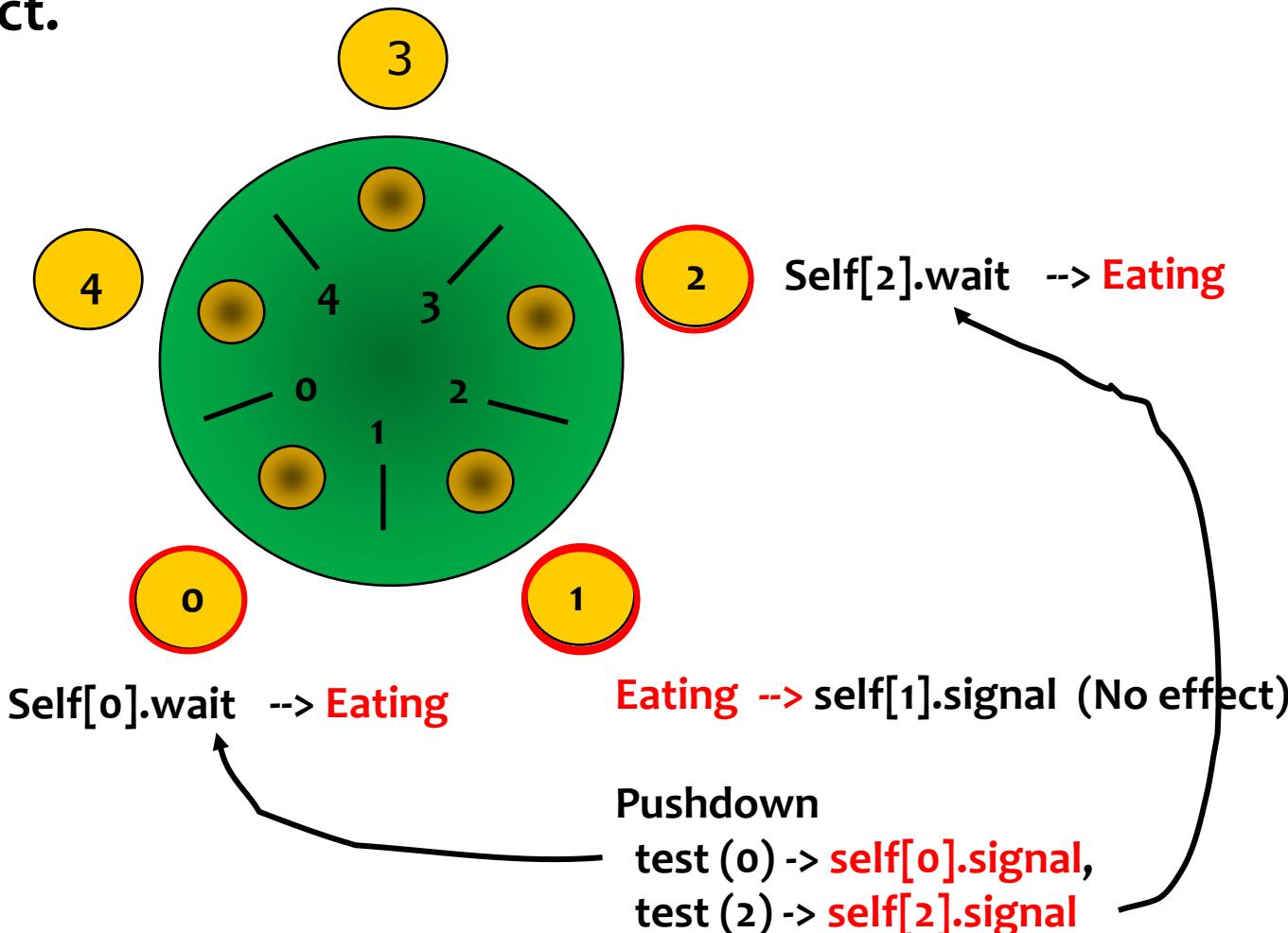
Test if both chopsticks are available

```
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;
}
```

Illustration of the algorithm

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



Monitor Implementation Using Semaphores

A possible implementation of the monitor mechanism using semaphores.

For each monitor, a semaphore mutex (init to 1) is provided.

A process must execute wait (mutex) before entering the monitor and must execute signal (mutex) after leaving the monitor

Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced (init to 0).

The signaling processes can use **next** to suspend themselves.

An integer variable **next_count** is also provided to count the number of processes suspended on next.

Monitor Implementation Using Semaphores

Variables

```
semaphore mutex; // (initially = 1)  
semaphore next; // (initially = 0)  
int next-count = 0;
```

Each external procedure F will be replaced by

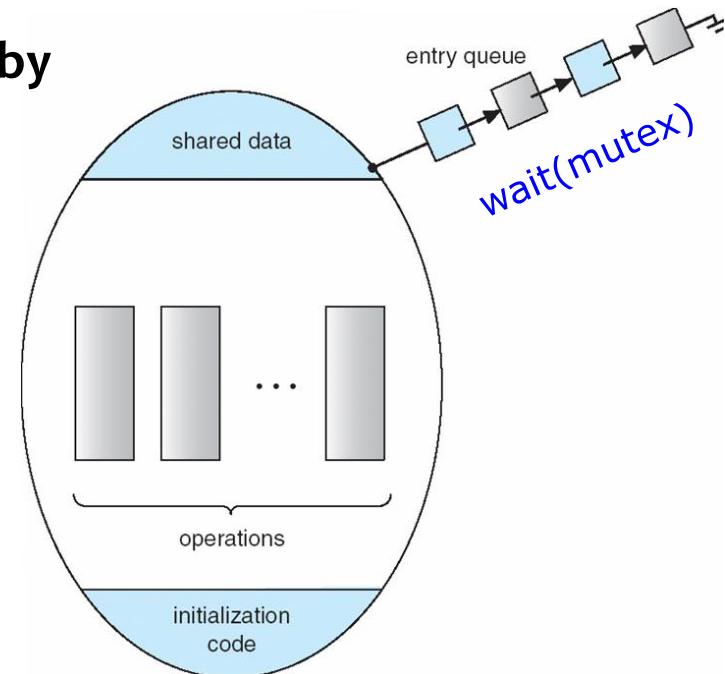
wait(mutex);

...

body of F

...

```
if (next_count > 0)  
    signal(next)  
else  
    signal(mutex);
```



Mutual exclusion within a monitor is ensured.

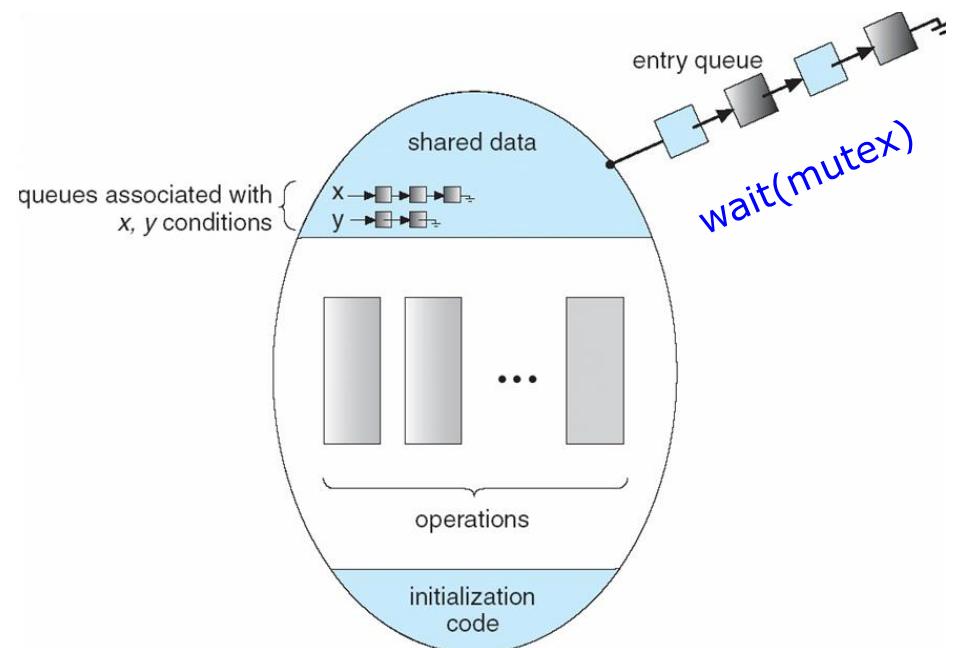
Monitor (Condition Variable) Implementation Using Semaphores

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

```
semaphore x_sem; // (initially = 0)  
int x-count = 0;
```

The operation **x.wait** can be implemented as:

```
x-count++;  
if (next_count > 0)  
    signal(next);  
else  
    signal(mutex);  
wait(x_sem);  
x-count--;
```



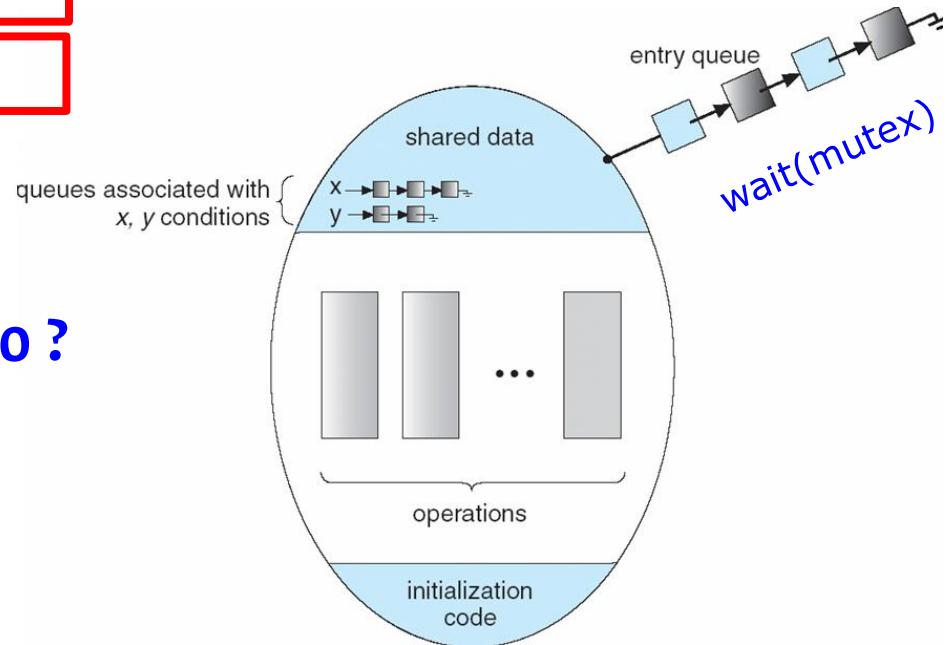
Monitor Implementation Using Semaphores

The operation **x.signal** can be implemented as:

```
if (x-count > 0) {  
    next_count++;  
    signal(x_sem);  
    wait(next);  
    next_count--;  
}
```

What happen if **x-count <= 0** ?

Nothing will happen !!



Resuming Processes within a Monitor

If several processes are suspended on **condition x**, and an **x.signal()** operation is executed by some process, how do we determine **which of the suspended processes should be resumed next**?

FCFS ordering is simple, but may not adequate

Conditional-wait construct

x.wait (c)

c is an integer expression that is evaluated when the wait() operation is executed.

c is called a priority number.

When x.signal () is executed, the process with **smallest priority number** is resumed next.

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;
```

```
}
```

```
}
```

The process with smallest priority number is resumed next

Resuming Processes within a Monitor

The monitor allocates the resource that has the shortest time-allocation request.

A process that needs to access the resource in question must observe the following sequence:

R.acquire (t); ← Get the resource, or wait for it !!

.....

access the resource

.....

R.release();

Where R is an instance of type ResourceAllocator.

6.8 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:

Prior 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 variables

Non-portable extensions include:

read-write locks

spin locks

6.9 Atomic Transactions

Make sure that a critical section forms a single logical unit of work that either is performed in its entirety or is not performed at all.

Consistency of data, along with storage and retrieval of data, is a concern often associated with database systems.

System Model

Log-based Recovery

Checkpoints

Concurrent Atomic Transactions

6.9.1 System Model

Assures that operations (a collection of instructions) 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

6.9.2 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

$\langle T_i \text{ starts} \rangle$ written to log when transaction T_i starts

$\langle T_i \text{ commits} \rangle$ written when T_i 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

$\text{Undo}(T_i)$ restores value of all data updated by T_i

$\text{Redo}(T_i)$ sets values of all data in transaction T_i to new values

$\text{Undo}(T_i)$ and $\text{redo}(T_i)$ 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 $\langle T_i \text{ starts} \rangle$ without $\langle T_i \text{ commits} \rangle$,
 $\text{undo}(T_i)$

If log contains $\langle T_i \text{ starts} \rangle$ and $\langle T_i \text{ commits} \rangle$, $\text{redo}(T_i)$

6.9.3 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
3. Output **a log record <checkpoint>** to the log on stable storage

Now recovery only includes T_i , such that T_i started executing before the most recent checkpoint, and all transactions after T_i

All other transactions already on stable storage

6.9.4 Concurrent Atomic 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 T_0 and T_1

Execute T_0 , T_1 atomically

Execution sequence called **schedule**

Atomically executed transaction order called
serial schedule

For N transactions, there are $N!$ valid serial schedules

Schedule 1: T_0 then T_1

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Nonserial Schedule

Nonserial schedule allows overlapped execute

Resulting execution not necessarily incorrect

Consider schedule S , operations O_i, O_j of Transactions T_i and T_j ,

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

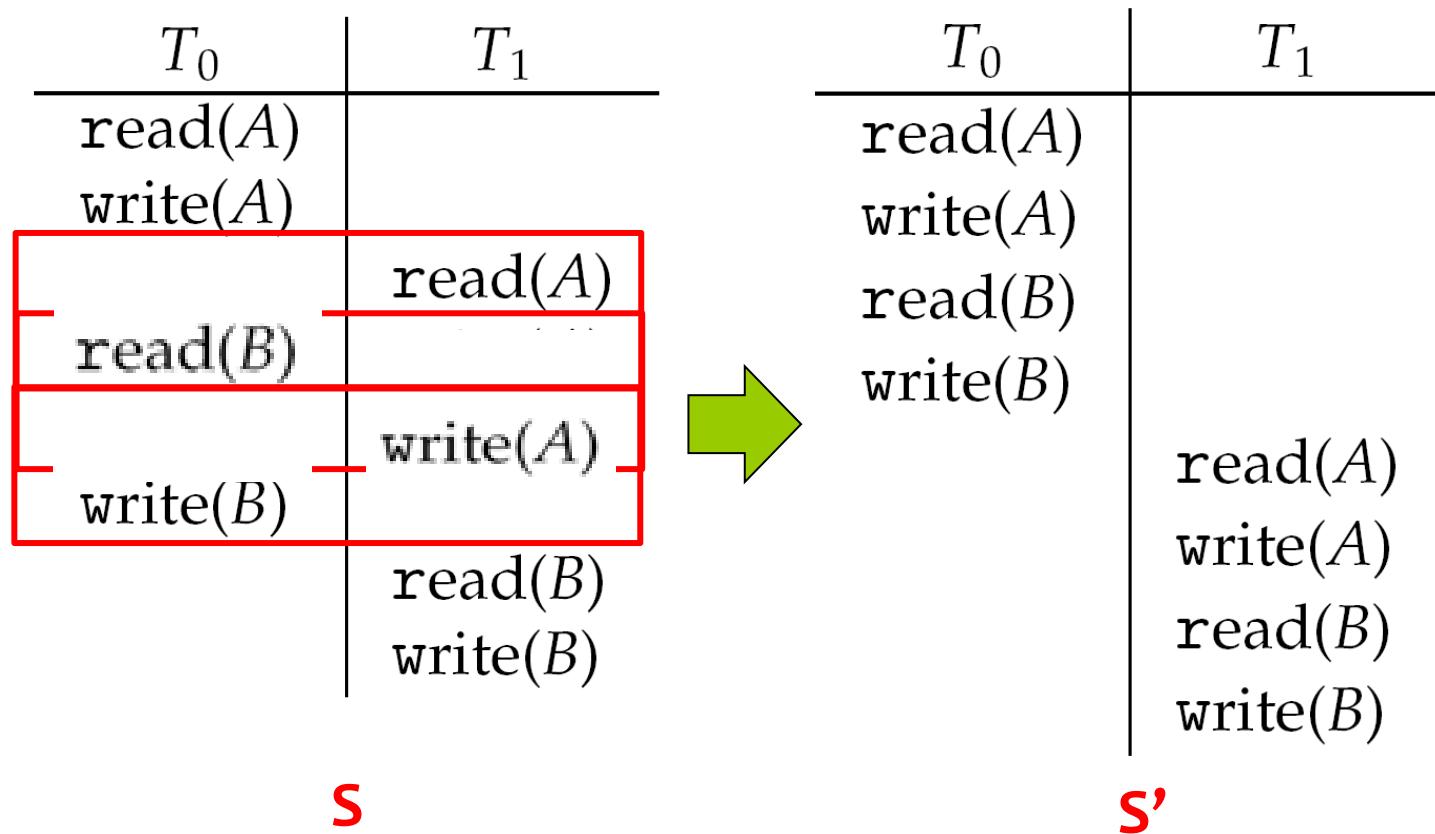
If O_i, O_j are consecutive and operations of different transactions & O_i and O_j don't conflict

Then S' with swapped order O_j, O_i equivalent to S

T_0	T_1
read(A)	
write(A)	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Nonserial Schedule

We say that S is **conflict serializable**, if it can be transformed into a serial schedule S' by a series of swaps of nonconflicting operations.



Locking Protocol

One way to ensure serializability is to associate **a lock with each data item** and each transaction follows locking protocol for access control.

Locks

Shared – T_i has shared-mode lock (S) on item Q, T_i can read Q but not write Q

Exclusive – T_i has exclusive-mode lock (X) on Q, T_i 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

The two-phase locking protocol ensures conflict serializability

Each transaction issues lock and unlock requests in two phases

Growing – A transaction may obtain locks but may not release any locks

Shrinking – A transaction may release locks but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and no more lock requests can be issued

Does not prevent deadlock

Timestamp-based Protocols

Select order among transactions **in advance – timestamp-ordering**

Transaction T_i associated with **timestamp $TS(T_i)$** before T_i starts

$TS(T_i) < TS(T_j)$ if T_i entered system before T_j

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

Timestamps determine serializability order

If $TS(T_i) < TS(T_j)$, system must ensure produced schedule equivalent to serial schedule where T_i appears before T_j

Timestamp-based Protocol Implementation

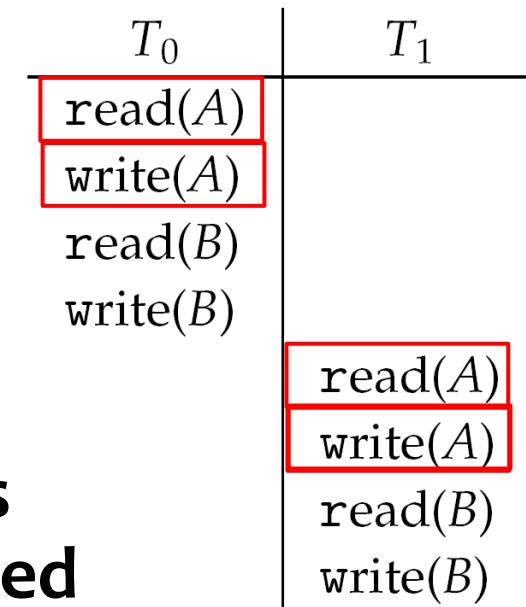
Data item Q gets two timestamps

W-timestamp(Q) – largest timestamp of any transaction that executed $\text{write}(Q)$ successfully

R-timestamp(Q) – largest timestamp of successful $\text{read}(Q)$

Updated whenever $\text{read}(Q)$ or $\text{write}(Q)$ executed

Timestamp-ordering protocol assures any conflicting read and write executed in timestamp order



Timestamp-based Protocol Implementation

Suppose T_i executes $\text{read}(Q)$

If $TS(T_i) < W\text{-timestamp}(Q)$, T_i needs to read value of Q that was already overwritten

- ▶ **read operation rejected and T_i rolled back**

If $TS(T_i) \geq W\text{-timestamp}(Q)$

- ▶ **read executed, $R\text{-timestamp}(Q)$ set to $\max(R\text{-timestamp}(Q), TS(T_i))$**

Timestamp-ordering Protocol

Suppose T_i executes `write(Q)`

If $TS(T_i) < R\text{-timestamp}(Q)$, value Q produced by T_i was needed previously and T_i assumed it would never be produced

- ▶ `Write` operation rejected, T_i rolled back

If $TS(T_i) < W\text{-timestamp}(Q)$, T_i attempting to write obsolete value of Q

- ▶ `Write` operation rejected and T_i rolled back

Otherwise, `write` executed

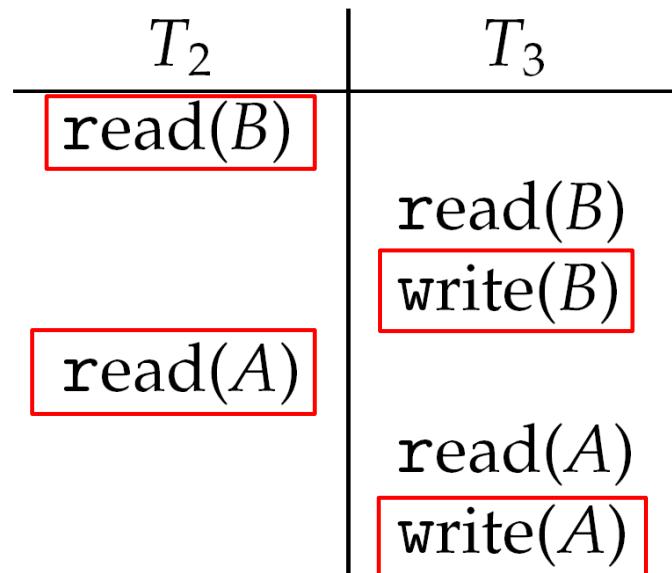
A transaction T_i is rolled back as a result of either a read or write operation is assigned a new timestamp and is restarted

Timestamp-ordering Protocol Example

Assume a transaction is assigned a timestamp immediately before its first instruction.

Thus, $\text{TS}(T_2) < \text{TS}(T_3)$

The following schedule is possible under Timestamp Protocol



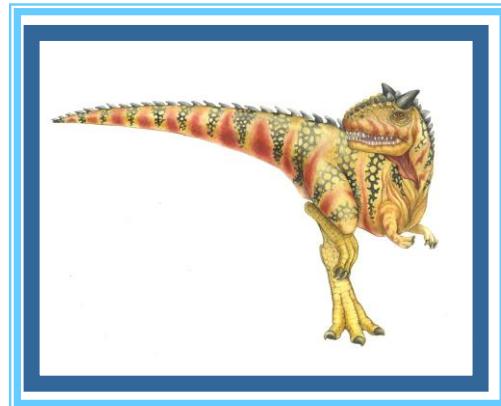
Timestamp-ordering Protocol

This algorithm **ensures**

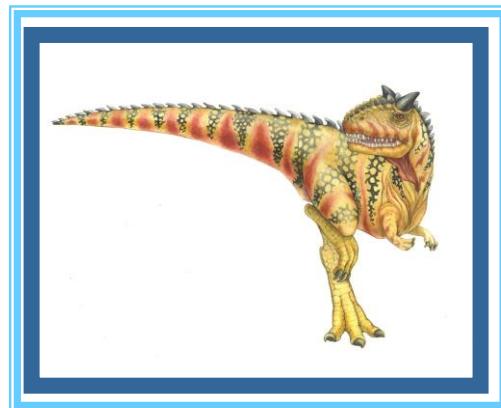
conflict serializability – conflicting operations
are processed in timestamp order, and

freedom from deadlock – no transactions ever
waits

End of Chapter 6



Chapter 7: Deadlocks



Chapter 7: Deadlocks

The Deadlock Problem

System Model

Deadlock Characterization

Methods for Handling Deadlocks

Deadlock Prevention

Deadlock Avoidance

Deadlock Detection

Recovery from Deadlock

Chapter Objectives

- 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 holds one disk drive and each needs another one

Example

semaphores A and B, initialized to 1

P_0

wait (A);

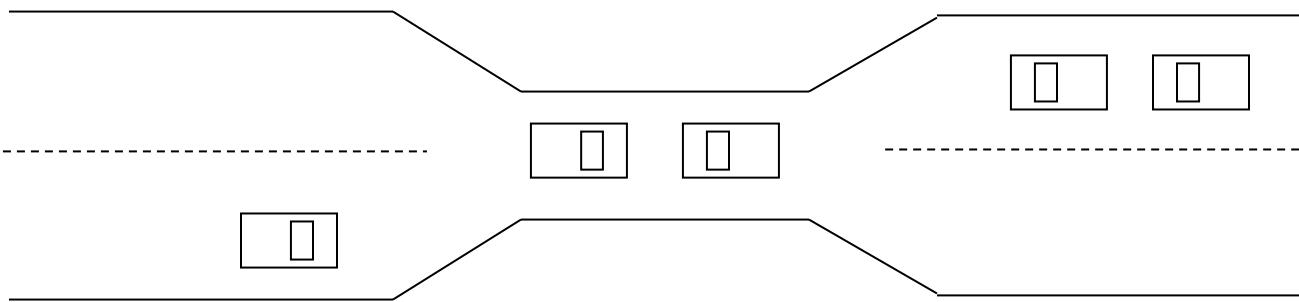
wait (B);

P_1

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, \dots, 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 $\{P_0, P_1, \dots, P_n\}$ 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, \dots, 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 .

Resource-Allocation Graph

A set of vertices V and a set of edges E .

V is partitioned into two types:

$P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system

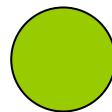
$R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system

request edge – directed edge $P_i \rightarrow R_j$

assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

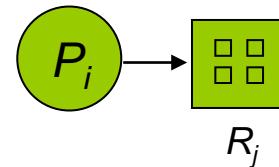
Process



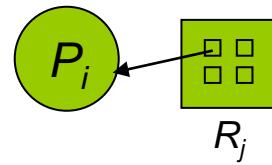
Resource Type with 4 instances



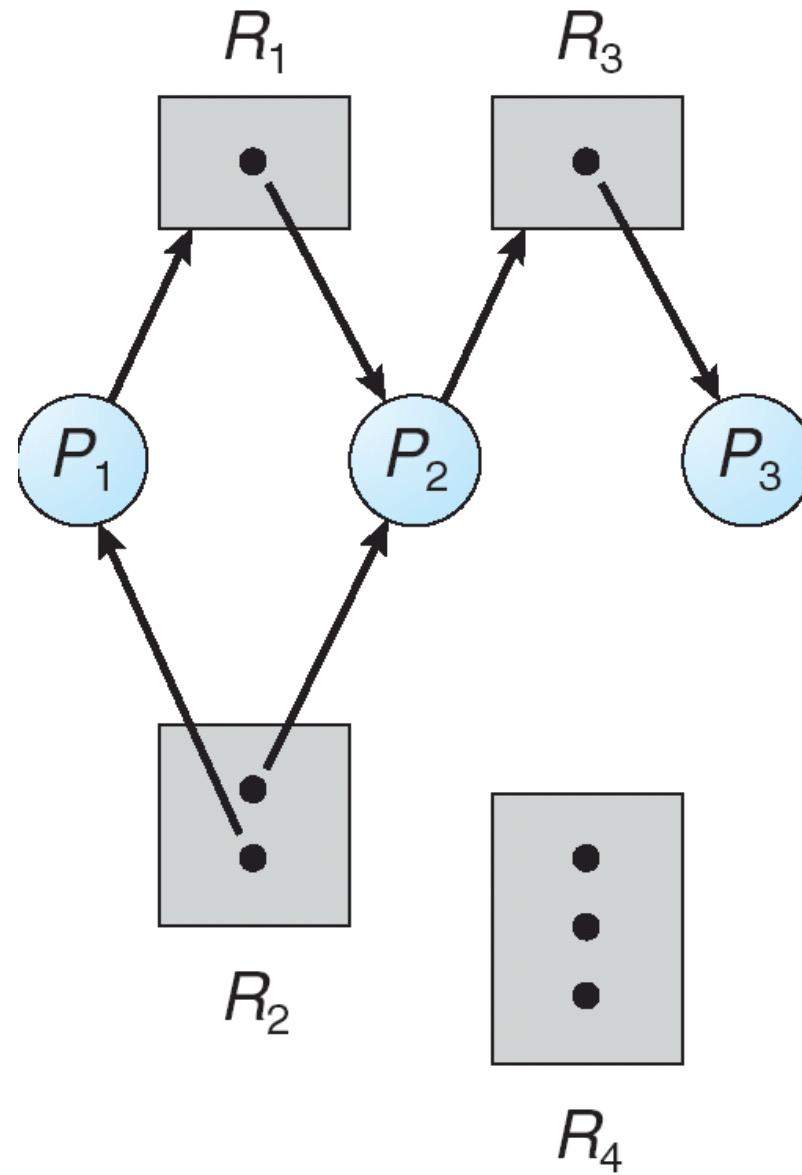
P_i requests instance of R_j



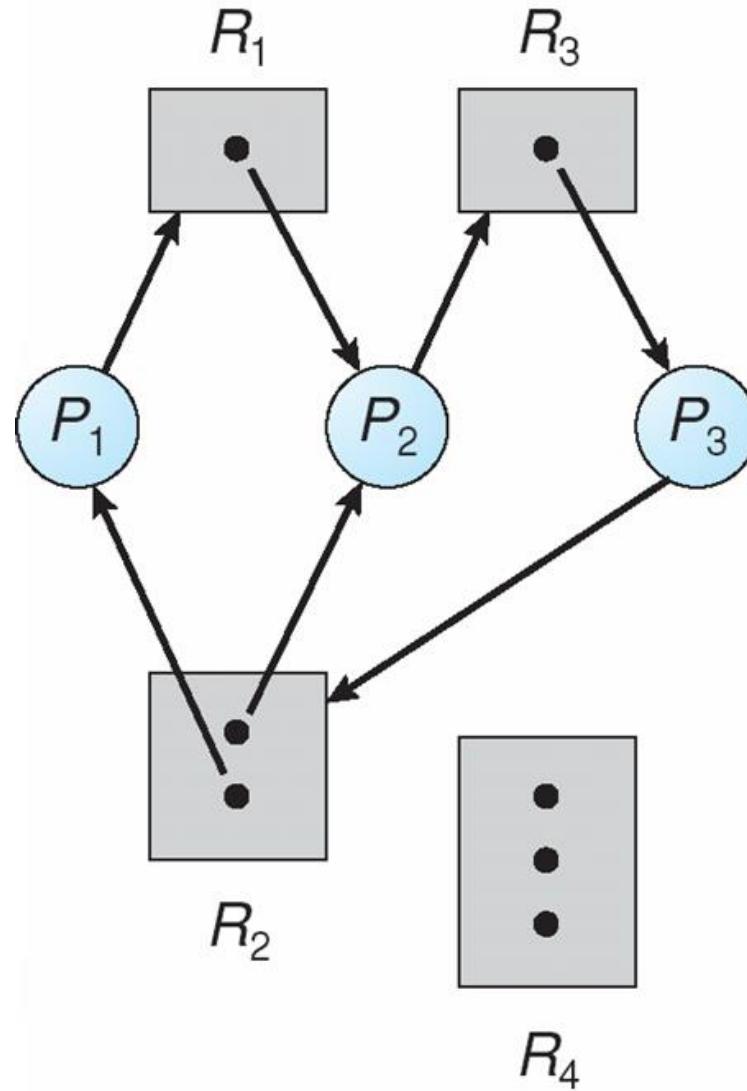
P_i is holding an instance of R_j



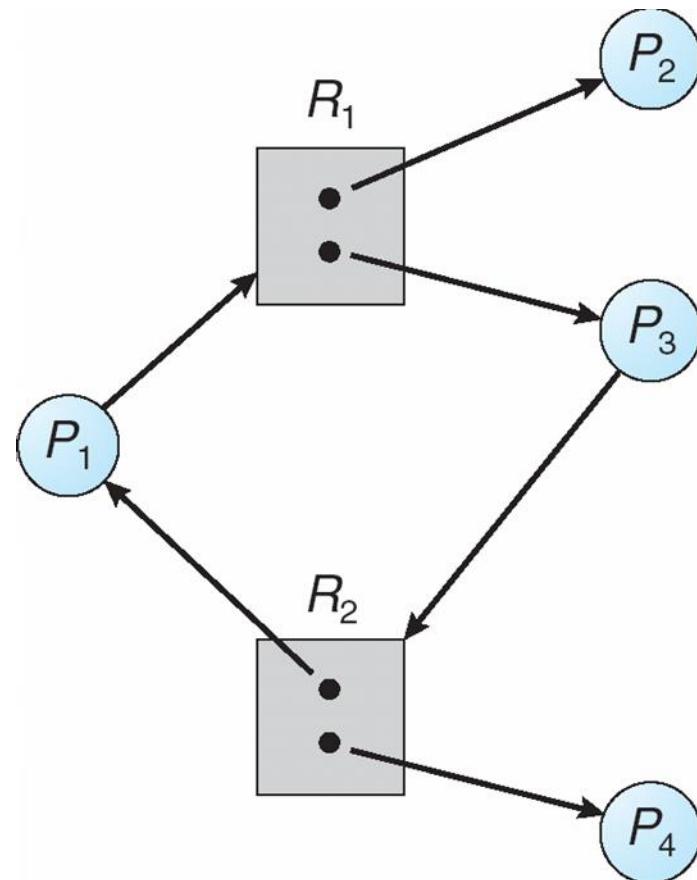
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

If graph contains **no cycles** \Rightarrow **no deadlock**

If graph contains a cycle \Rightarrow

if only one instance per resource type, then
deadlock

if several instances per resource type,
possibility of deadlock

Methods for Handling Deadlocks

Ensure that the system will **never** enter a deadlock state

Allow the system to enter a deadlock state and then **recover**

Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Restrain the ways request can be made

Mutual Exclusion – not required for sharable resources; must hold for nonsharable resources

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

Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

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

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 declares 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 $\langle P_1, P_2, \dots, P_n \rangle$ 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 $j < i$**

Safe State

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

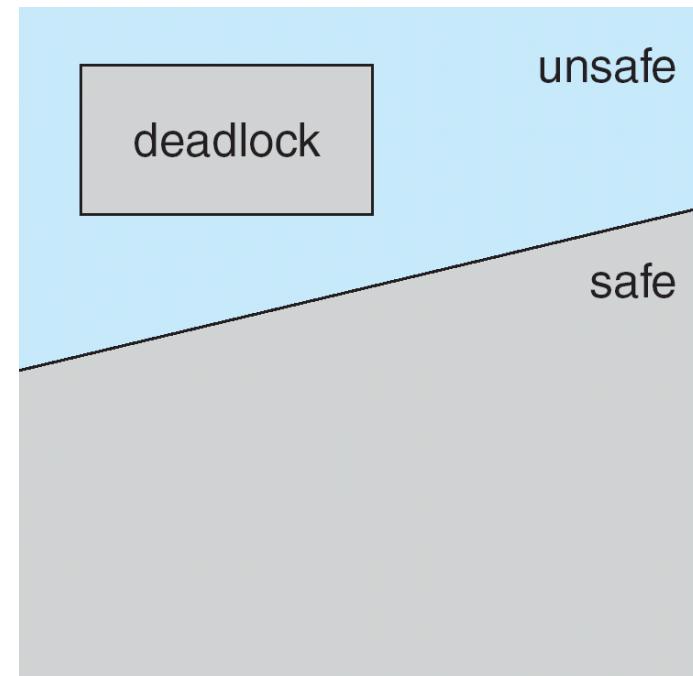
When P_i terminates, P_{i+1} can obtain its needed resources, and so on

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**.



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_i may request resource R_j ; represented by a dashed line

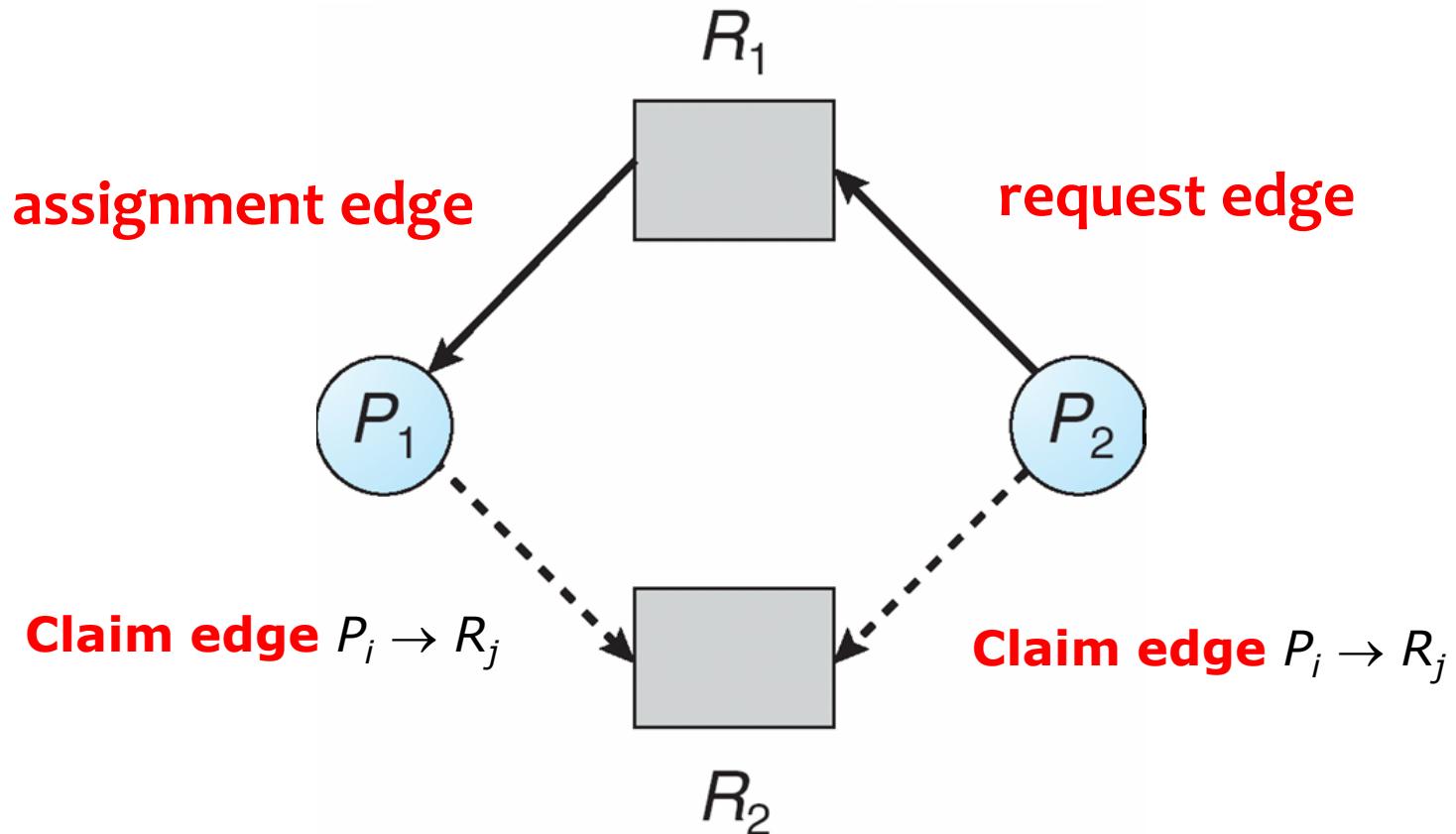
Claim edge converts to **request edge** when a process requests a resource

Request edge converted to an **assignment edge** when the resource is allocated to the process

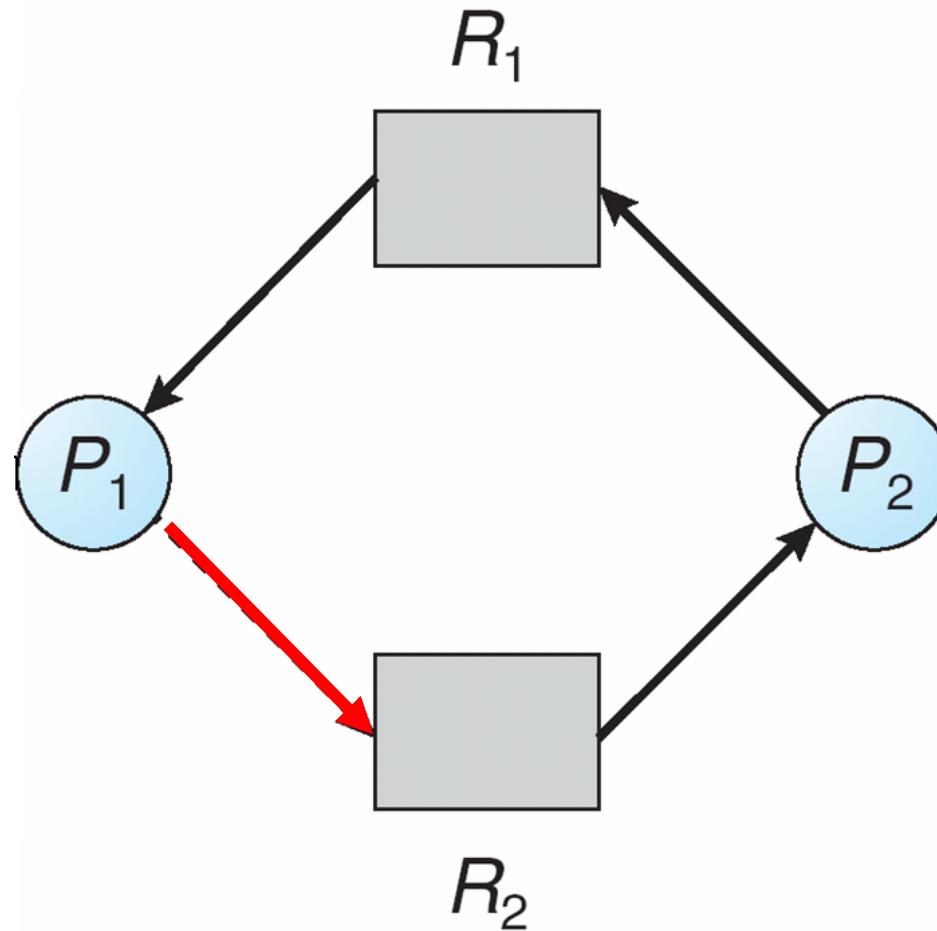
When a resource is released by a process, assignment edge reconverts to a claim edge

Resources must be claimed *a priori* in the system

Resource-Allocation Graph



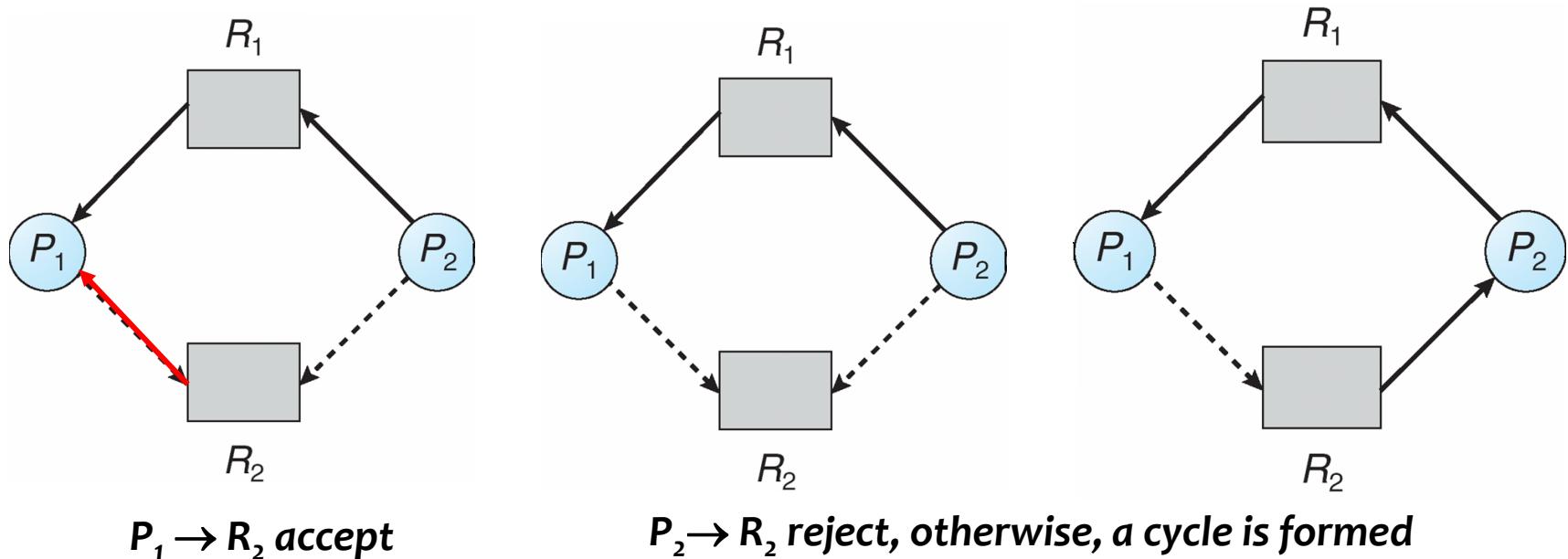
Unsafe State In Resource-Allocation Graph



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 $\text{available}[j] = k$, there are k instances of resource type R_j available

Max: $n \times m$ matrix. If $\text{Max}[i,j] = k$, then process P_i may request at most k instances of resource type R_j

Allocation: $n \times m$ matrix. If $\text{Allocation}[i,j] = k$ then P_i is currently allocated k instances of R_j

Need: $n \times m$ matrix. If $\text{Need}[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$

Safety Algorithm

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

$\text{Work} = \text{Available}$

$\text{Finish}[i] = \text{false}$ for $i = 0, 1, \dots, n-1$

2. Find any i such that both:

(a) $\text{Finish}[i] = \text{false}$

(b) $\text{Need}_i \leq \text{Work}$

If no such i exists, go to step 4

3. $\text{Work} = \text{Work} + \text{Allocation}_i$

$\text{Finish}[i] = \text{true}$

go to step 2

4. If $\text{Finish}[i] == \text{true}$ 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 $\text{Request}_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $\text{Request}_i \leq \text{Need}_i$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\text{Request}_i \leq \text{Available}$, 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:

$$\text{Available} = \text{Available} - \text{Request};$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i;$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i;$$

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 :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
	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 (Cont.)

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

	<u>Need</u>			<u>Available</u>		
	A	B	C	A	B	C
P_0	7	4	3	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
 $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria

Example: P_1 Request (1,0,2)

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

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	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 1	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 $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety requirement

Can request for (3,3,0) by P_4 be granted?

Can request for (0,2,0) by P_0 be granted?

Deadlock Detection

Allow system to enter deadlock state

Detection algorithm

Recovery scheme

Single Instance of Each Resource Type

Maintain **wait-for** graph

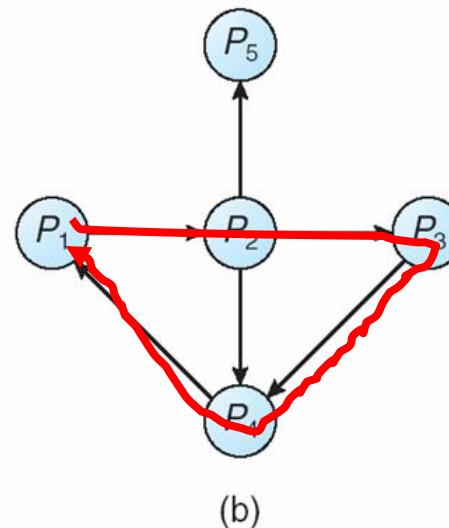
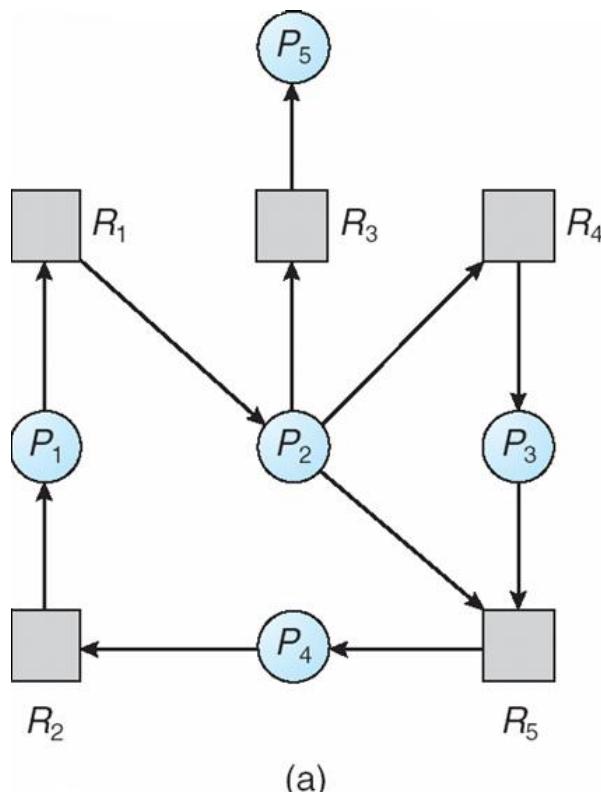
Nodes are processes

$P_i \rightarrow P_j$ 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 $n \times m$ matrix defines the number of resources of each type currently allocated to each process.

Request: An $n \times m$ matrix indicates the current request of each process. If $\text{Request}[i, j] = k$, 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) $\text{Work} = \text{Available}$
 - (b) For $i = 1, 2, \dots, n$, if $\text{Allocation}_i \neq 0$, then $\text{Finish}[i] = \text{false}$; otherwise, $\text{Finish}[i] = \text{true}$
2. Find an index i such that both:
 - (a) $\text{Finish}[i] == \text{false}$
 - (b) $\text{Request}_i \leq \text{Work}$

If no such i exists, go to step 4

Detection Algorithm (Cont.)

3. $\text{Work} = \text{Work} + \text{Allocation}$;

$\text{Finish}[i] = \text{true}$

go to step 2

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

Algorithm requires an order of $O(m \times n^2)$ 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 :

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	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 $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = \text{true}$ for all i

Example (Cont.)

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?

Priority of the process

How long process has computed, and how much longer to complete

Resources the process has used

Resources process needs to complete

How many processes will need to be terminated

Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

Selecting a victim – minimize cost

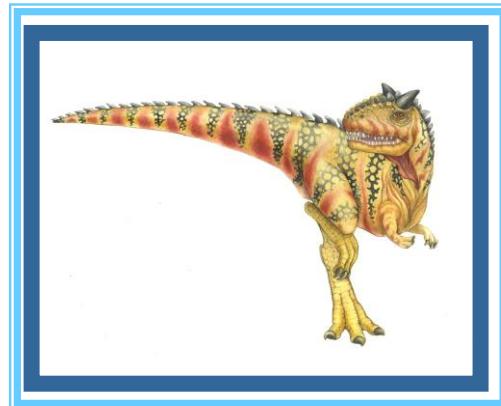
Rollback

**return to some safe state, restart process for
that state**

Starvation

**same process may always be picked as victim,
include number of rollbacks in cost factor**

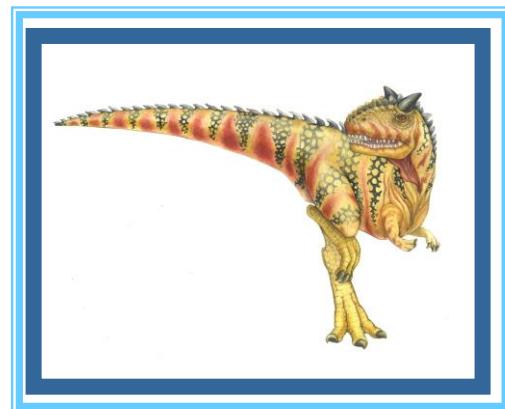
End of Chapter 7



Chapter 8:

Memory-Management

Strategies



Chapter 8: Memory Management Strategies

Background

Swapping

Contiguous Memory Allocation

Paging

Structure of the Page Table

Segmentation

Example: The Intel Pentium

Objectives

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**

Background

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)

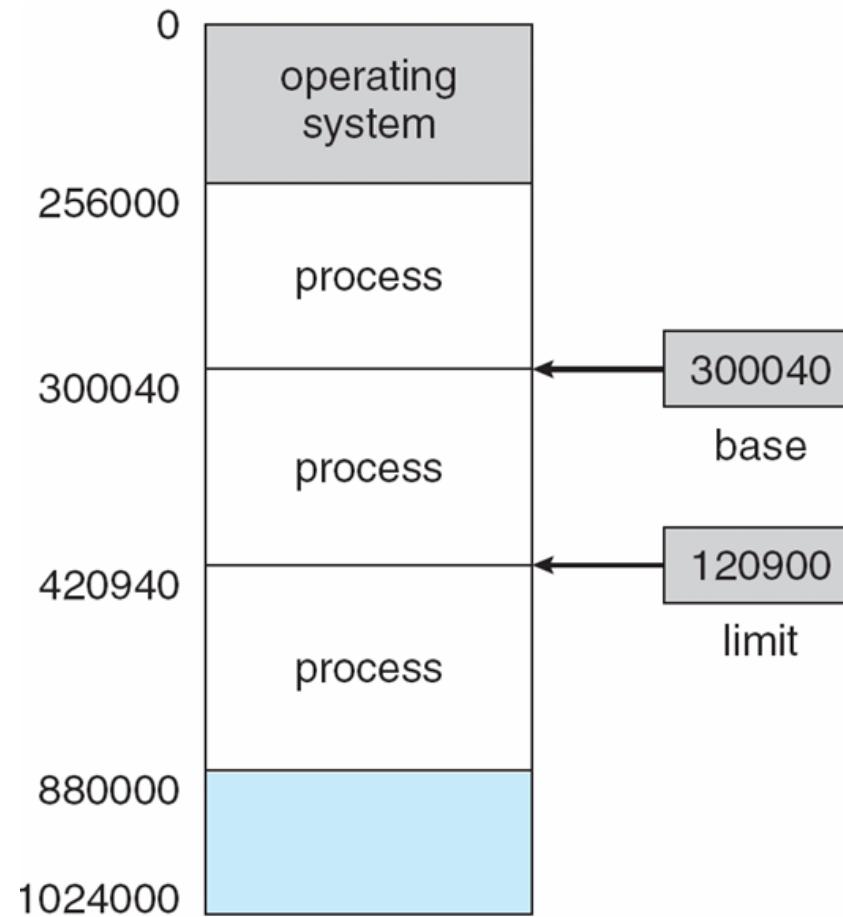
Main memory can take many cycles

Cache sits between main memory and CPU registers

Protection of memory is 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

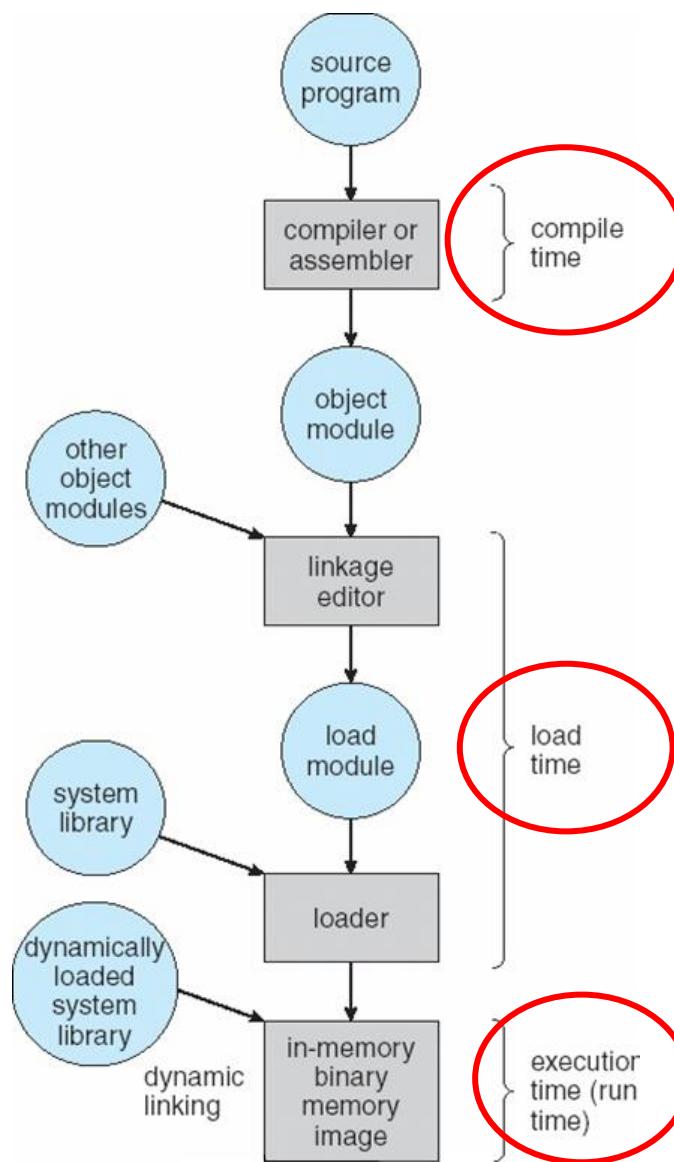
Address binding of instructions and data to memory addresses can happen at three different stages

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**

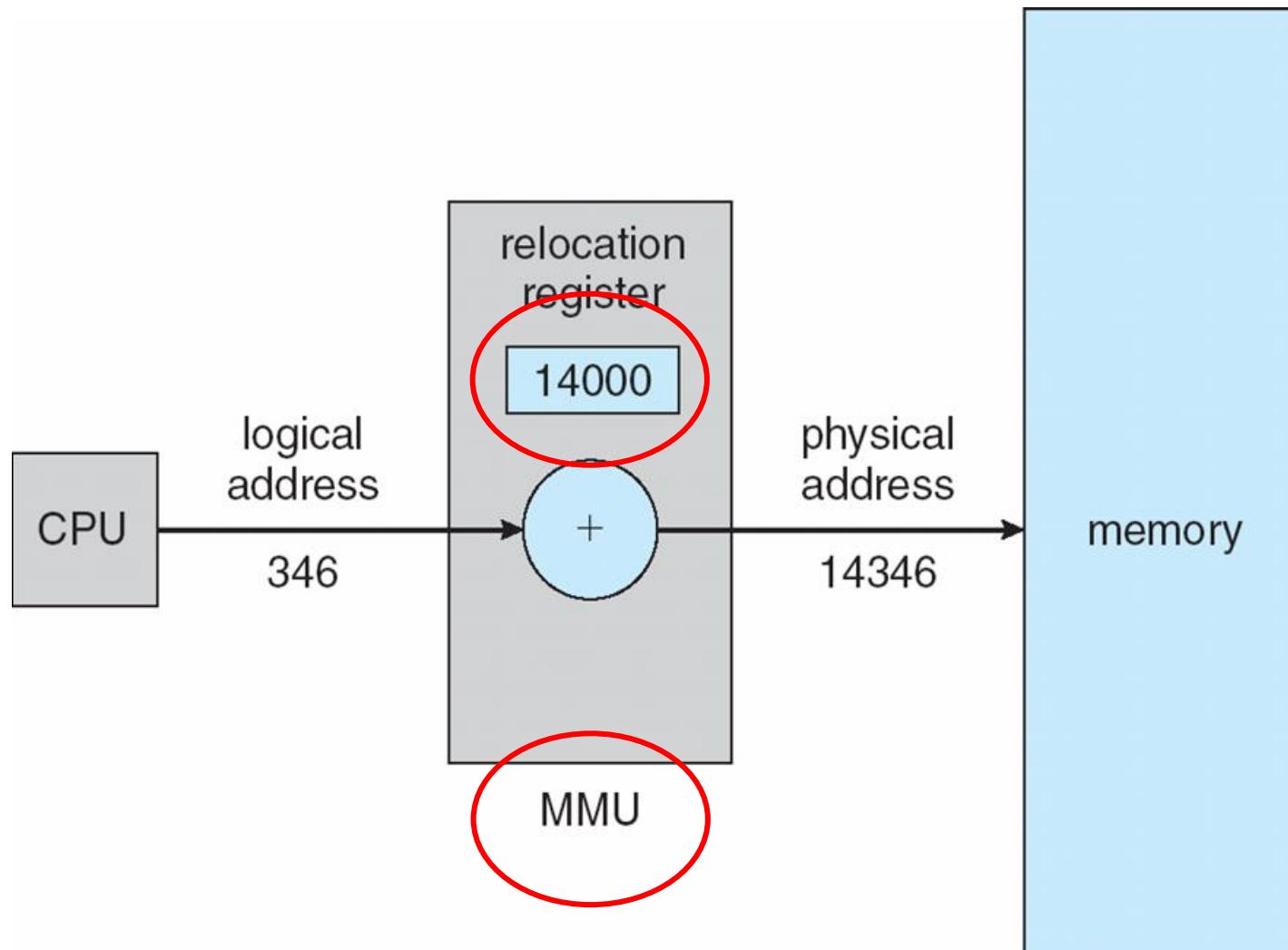
Memory-Management Unit (MMU)

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

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 execution

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

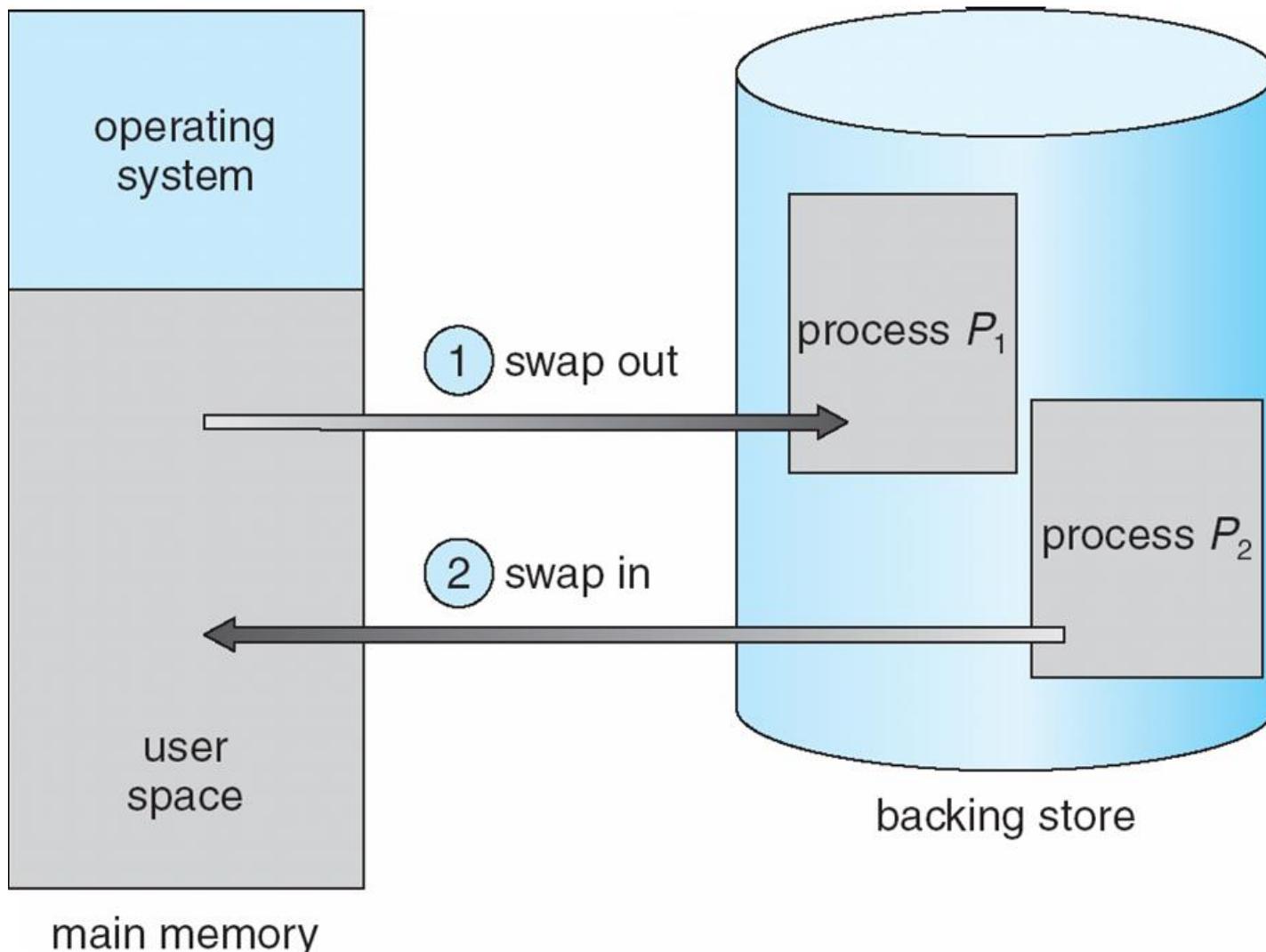
Swapping

Major part of swap time is **transfer time**; total transfer time is directly proportional to the amount of memory swapped

Modified 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 divides into two partitions:

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

User processes then held in high memory

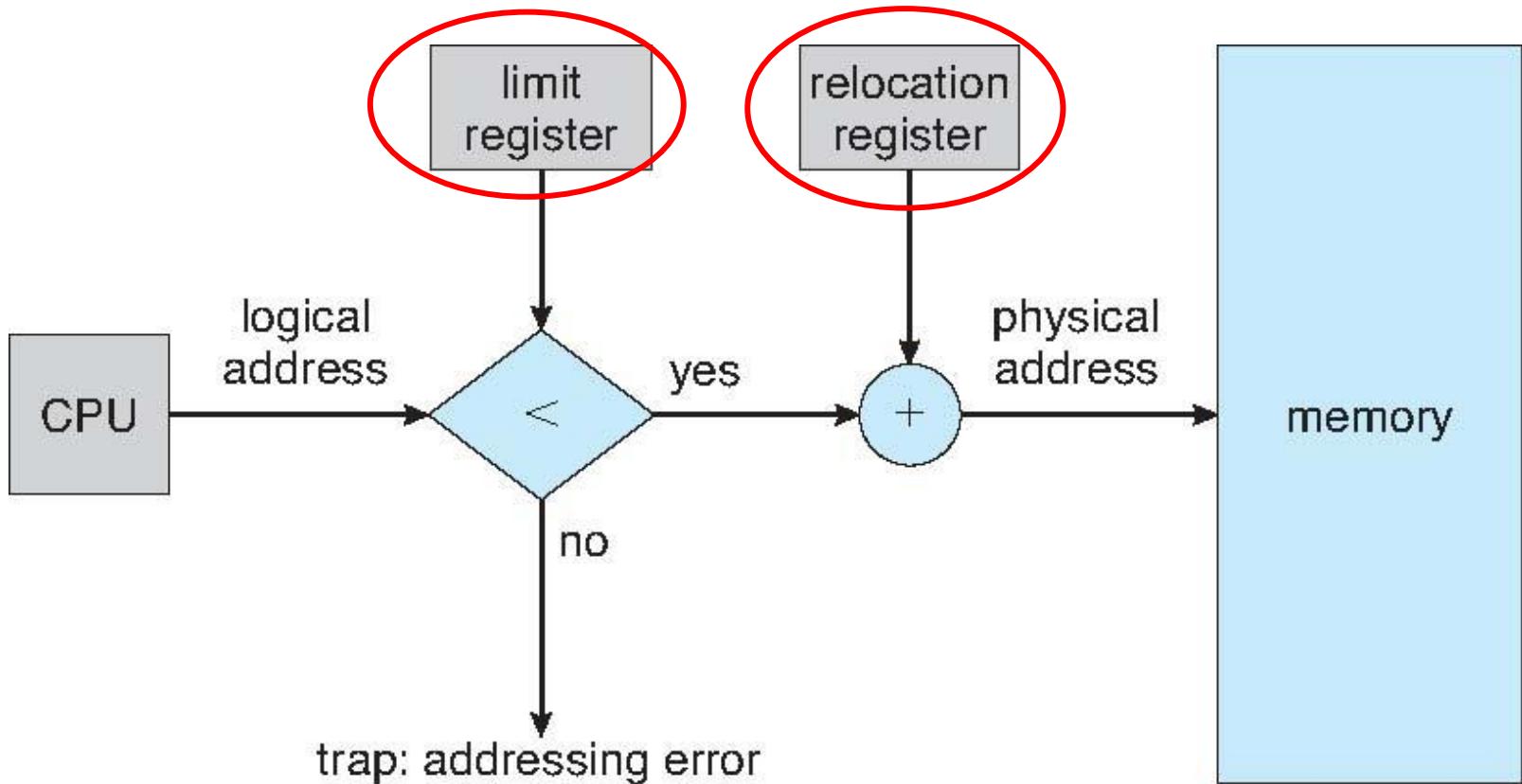
Relocation registers used to protect user processes from each other, and from changing operating-system code and data

Base register contains value of smallest physical address

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

MMU maps logical address ***dynamically***

Hardware Support for Relocation and Limit Registers



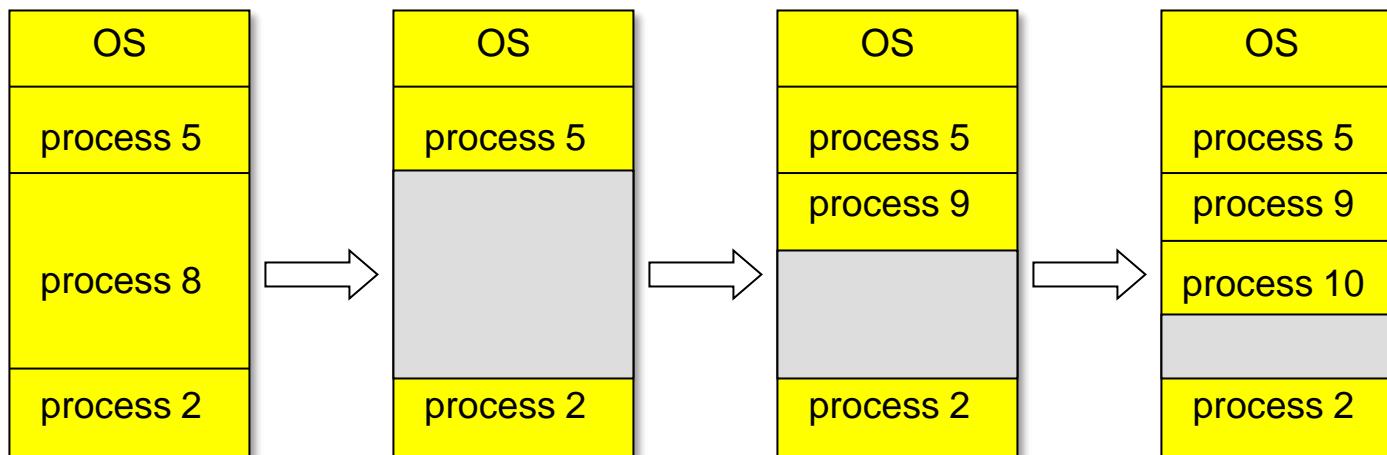
Contiguous Allocation (Cont)

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)



Dynamic Storage-Allocation Problem

How to satisfy a request of size n from a list of free holes ?

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

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

Paging avoids external fragmentation and the needs for compaction.

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

Divide logical memory into blocks of same size called pages

Keep 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

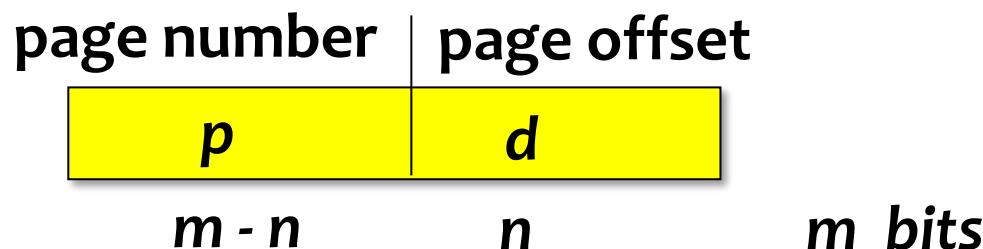
Internal fragmentation

Address Translation Scheme

Address generated by CPU is divided into:

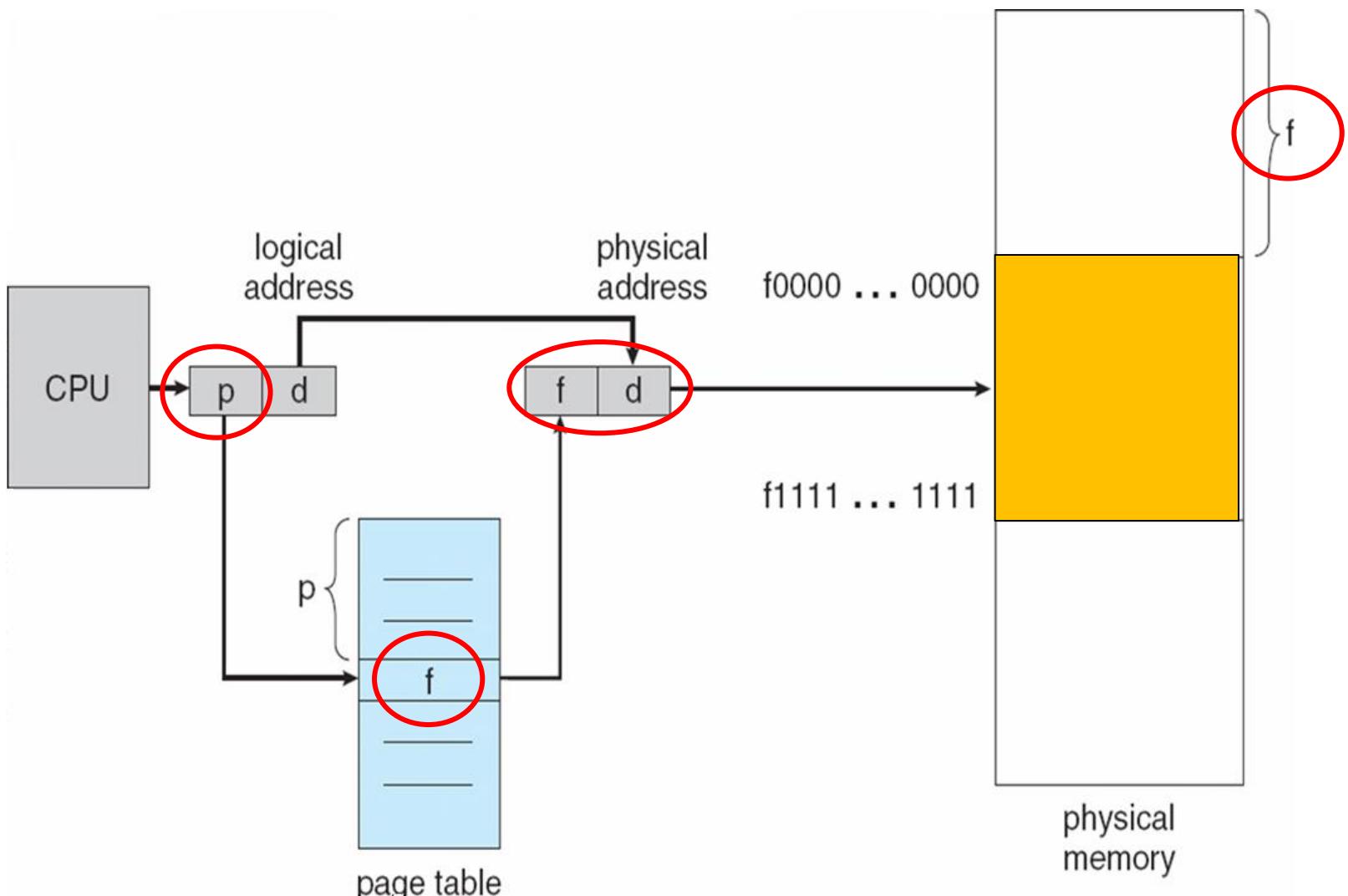
Page number (p) – used as an index into a *page table* which contains base address of each page in physical memory

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

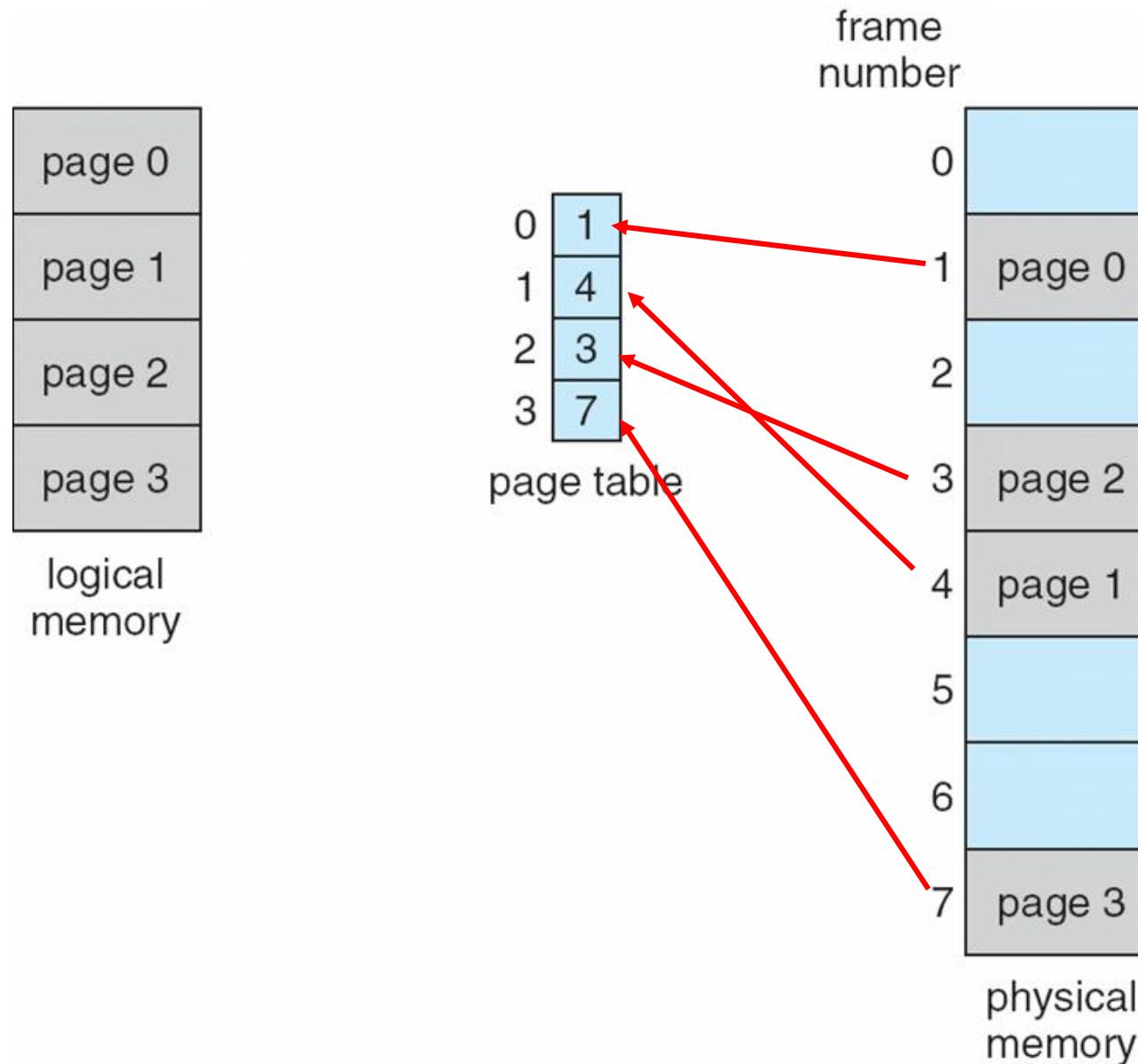


For given **logical address space 2^m** and **page size 2^n**

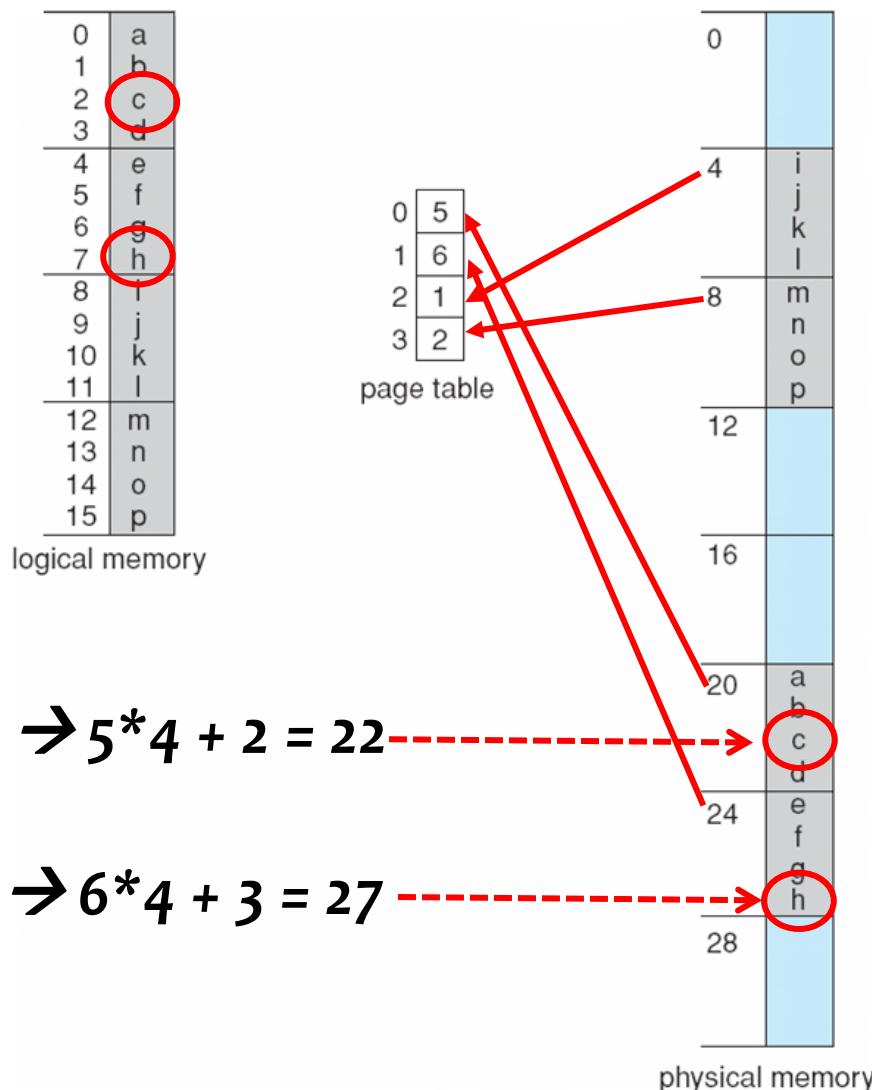
Paging Hardware



Paging Model of Logical and Physical Memory

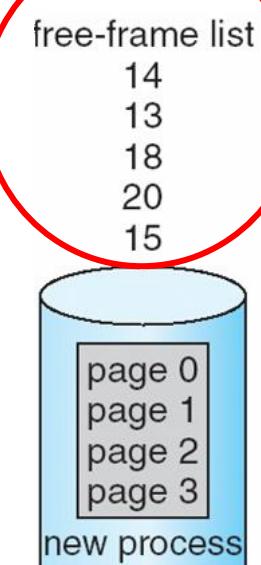


Paging Example



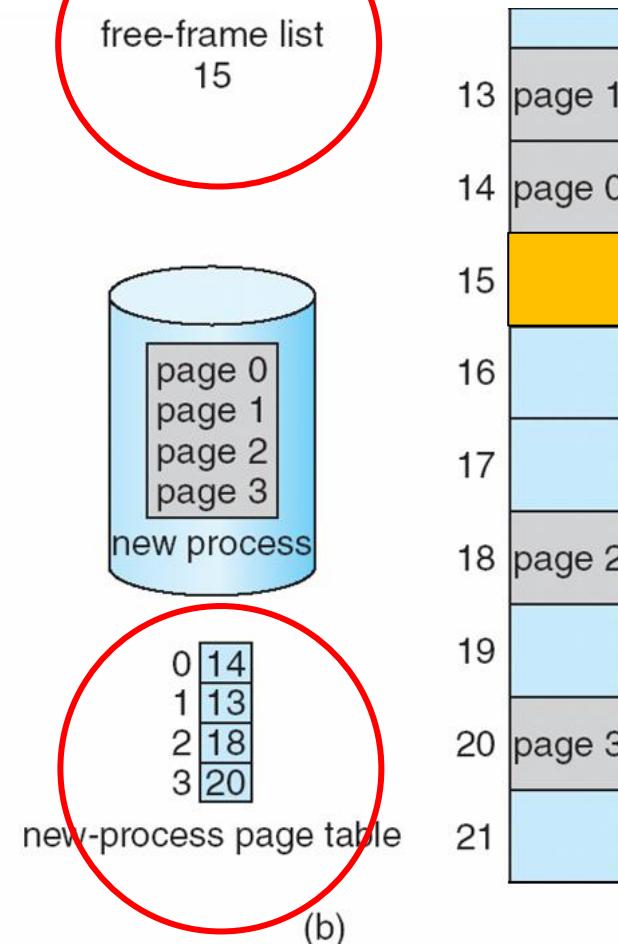
32-byte memory and 4-byte pages

Free Frames



(a)

Before allocation



(b)

After allocation

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 – provides **parallel search**

Page #	Frame #
3	8
4	10
6	5
10	7

$p = 6$

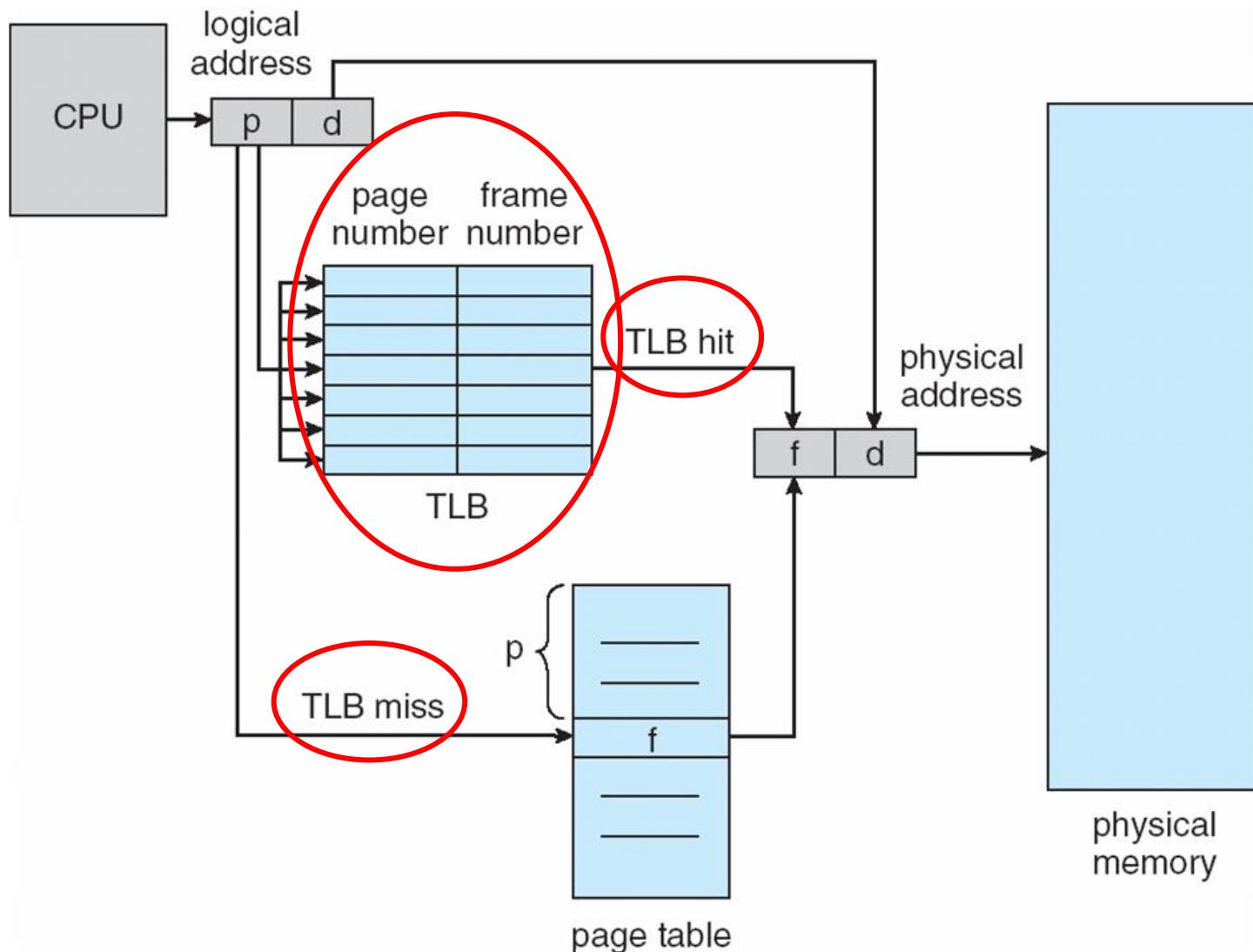
$F\# = 5$

■ Address translation (p, d)

If p is in associative register, get frame # out

Otherwise get frame # from page table in memory

Paging Hardware With TLB



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)

$$\begin{aligned} \text{EAT} &= (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

Memory Protection

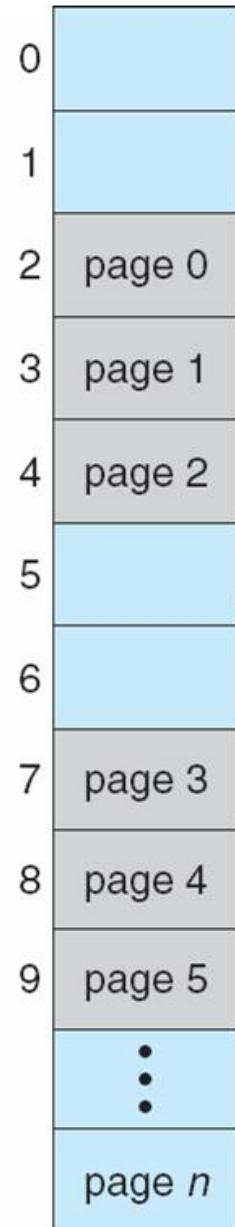
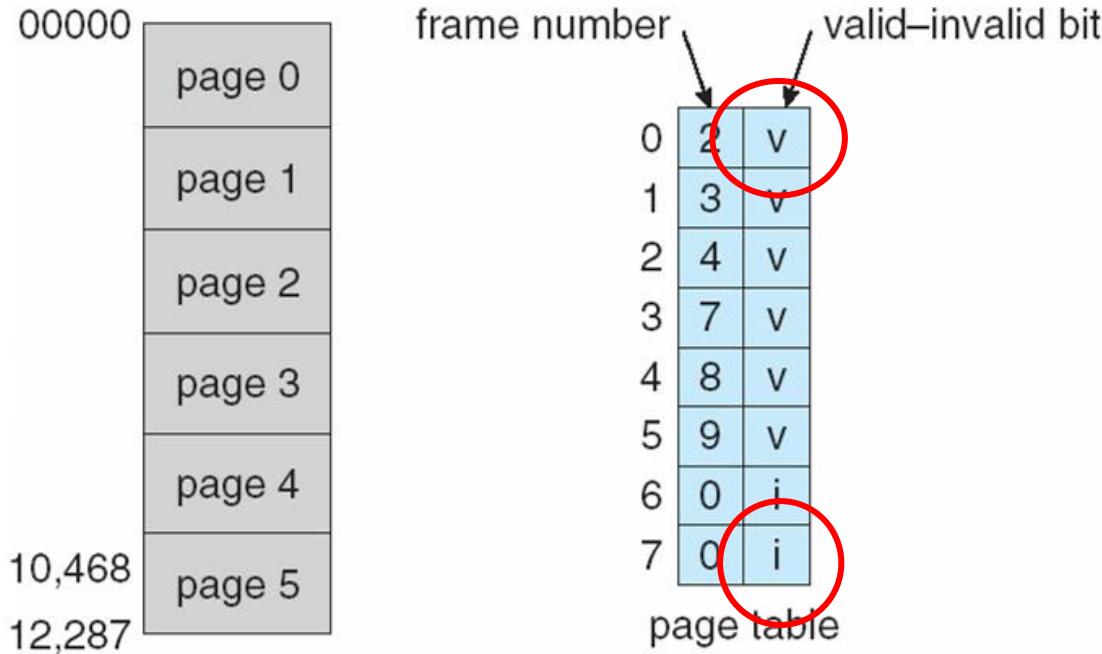
Memory protection implemented by associating **protection bit** with each frame

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).

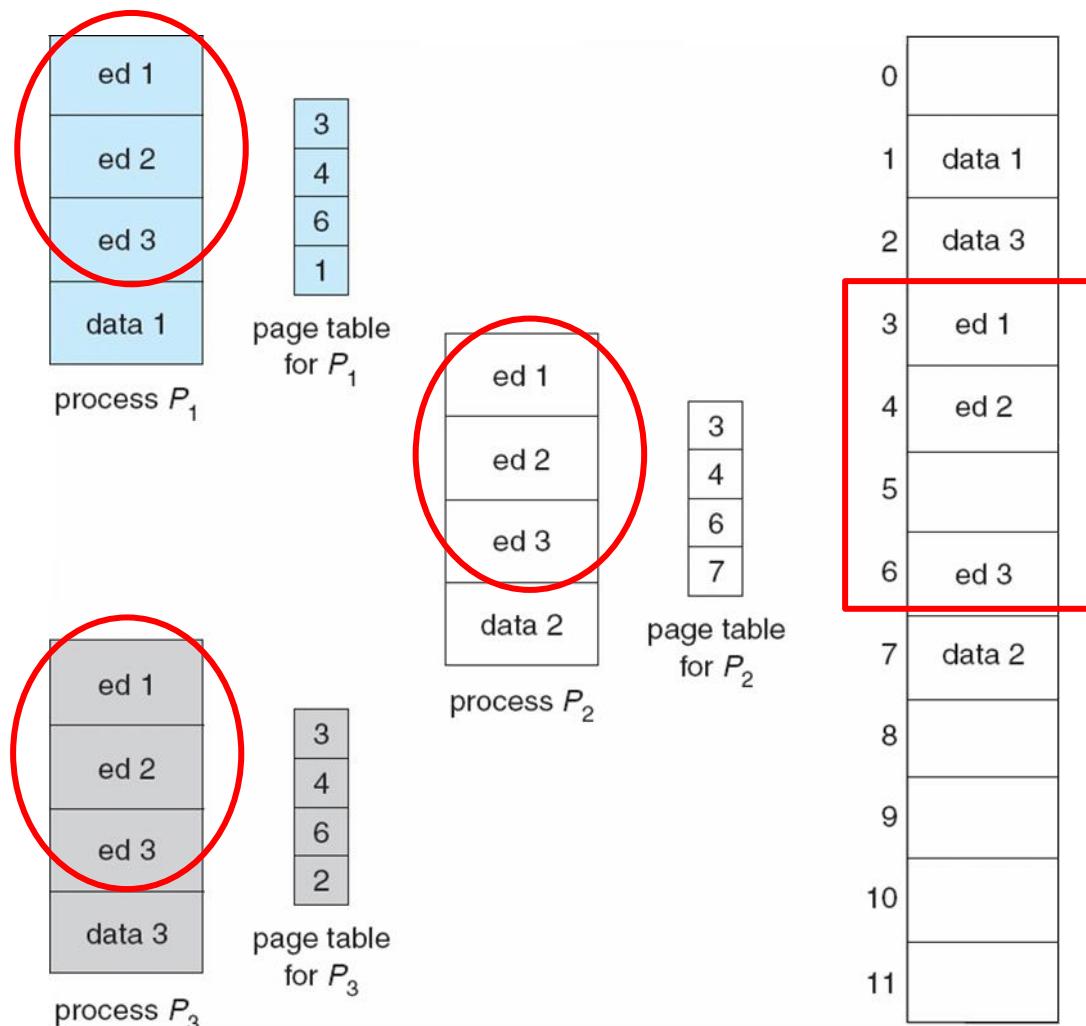
Reentrant code is non-self-modifying code: it never changes during execution.

Private code and data

Each process keeps a separate copy of the code and data

Some operating systems implement shared memory using shared pages.

Shared Pages Example



Structure of the Page Table

Hierarchical Paging

Hashed Page Tables

Inverted Page Tables

Hierarchical Page Tables

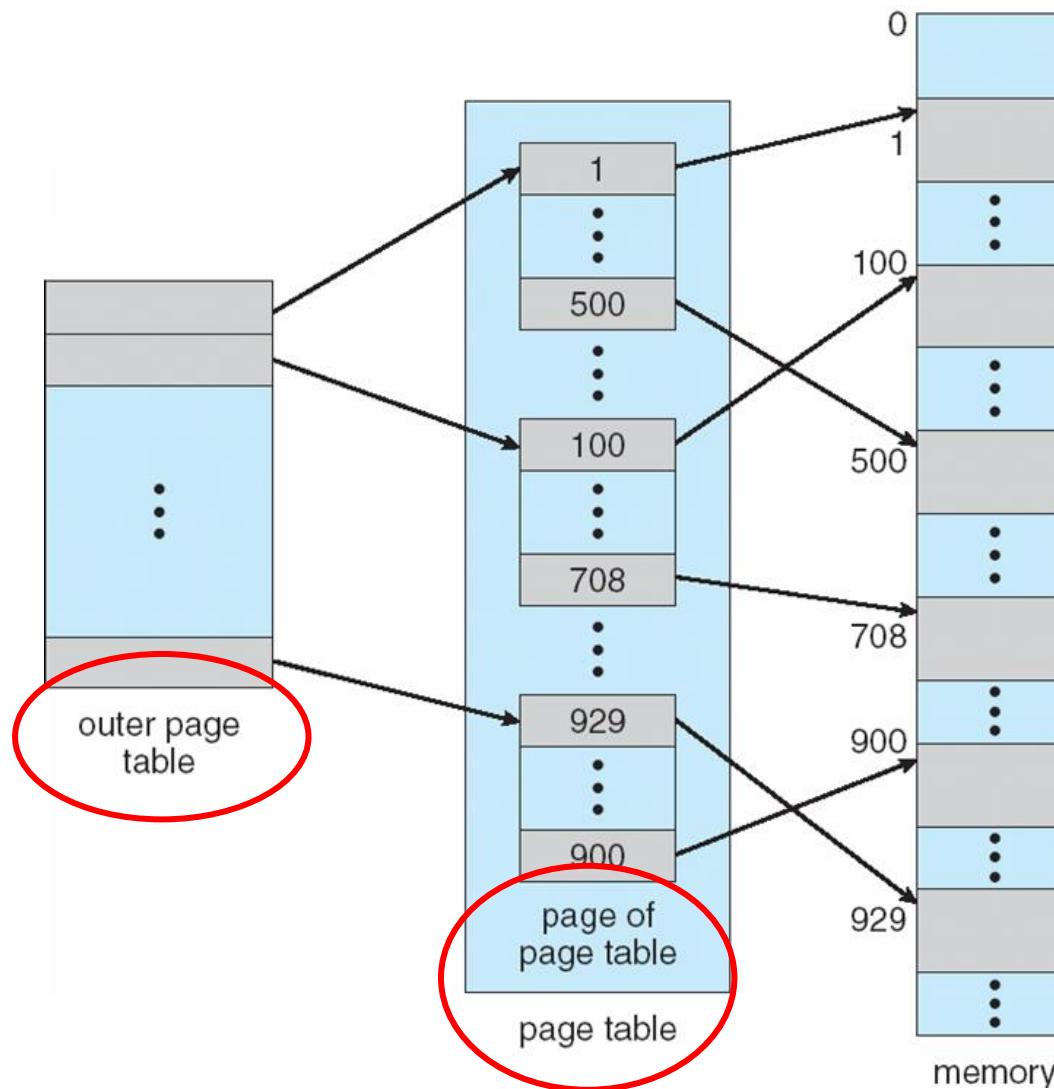
Modern computer systems support a large logical address space 2^{32} to 2^{64} . The page table itself becomes excessively large.

For 32-bit logical address space, and page size of 4K, then a page table consists of 1 million entries ($2^{32} / 2^{12} = 2^{20} = 1 \text{ million}$).

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

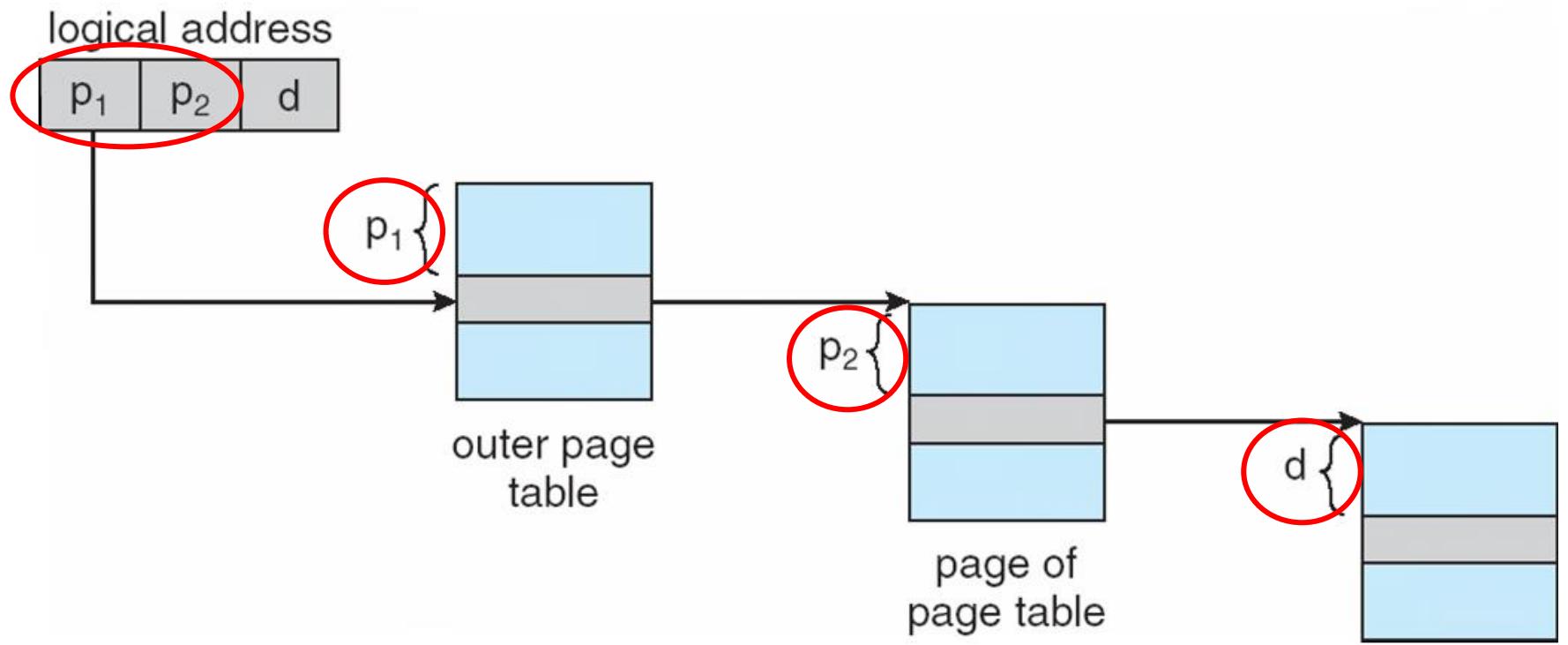
page number	page offset
p_1	p_2

12 10 10

Thus, a logical address is as

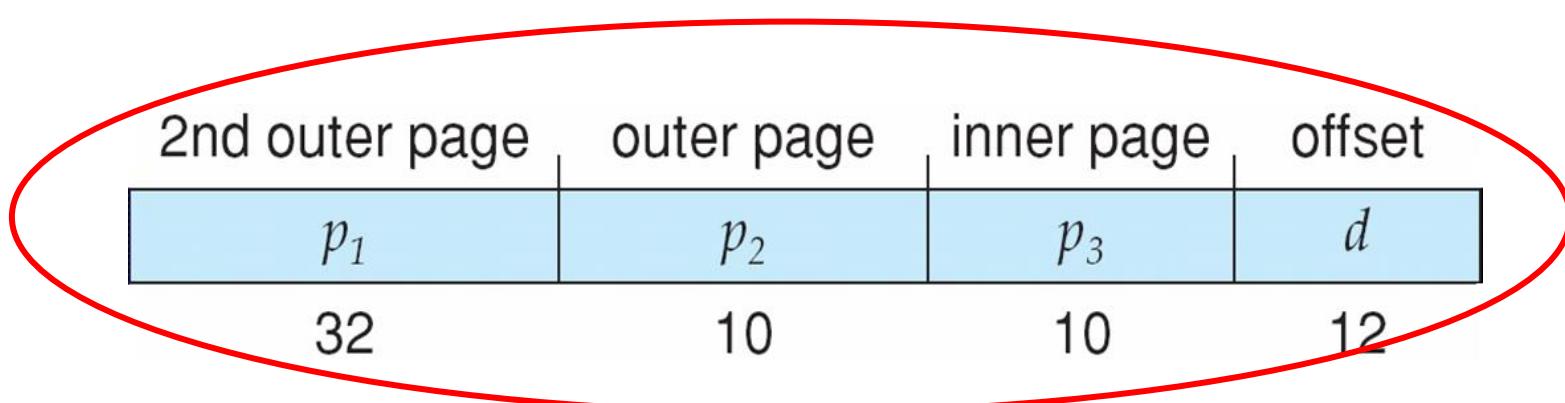
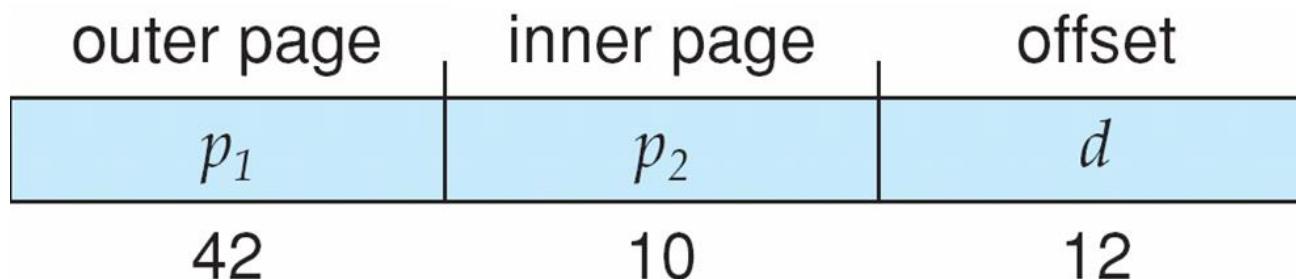
where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Address-Translation Scheme



Physical memory

Three-level Paging Scheme



64-bit machine with 4K page

Hashed Page Tables

Common in address spaces > 32 bits

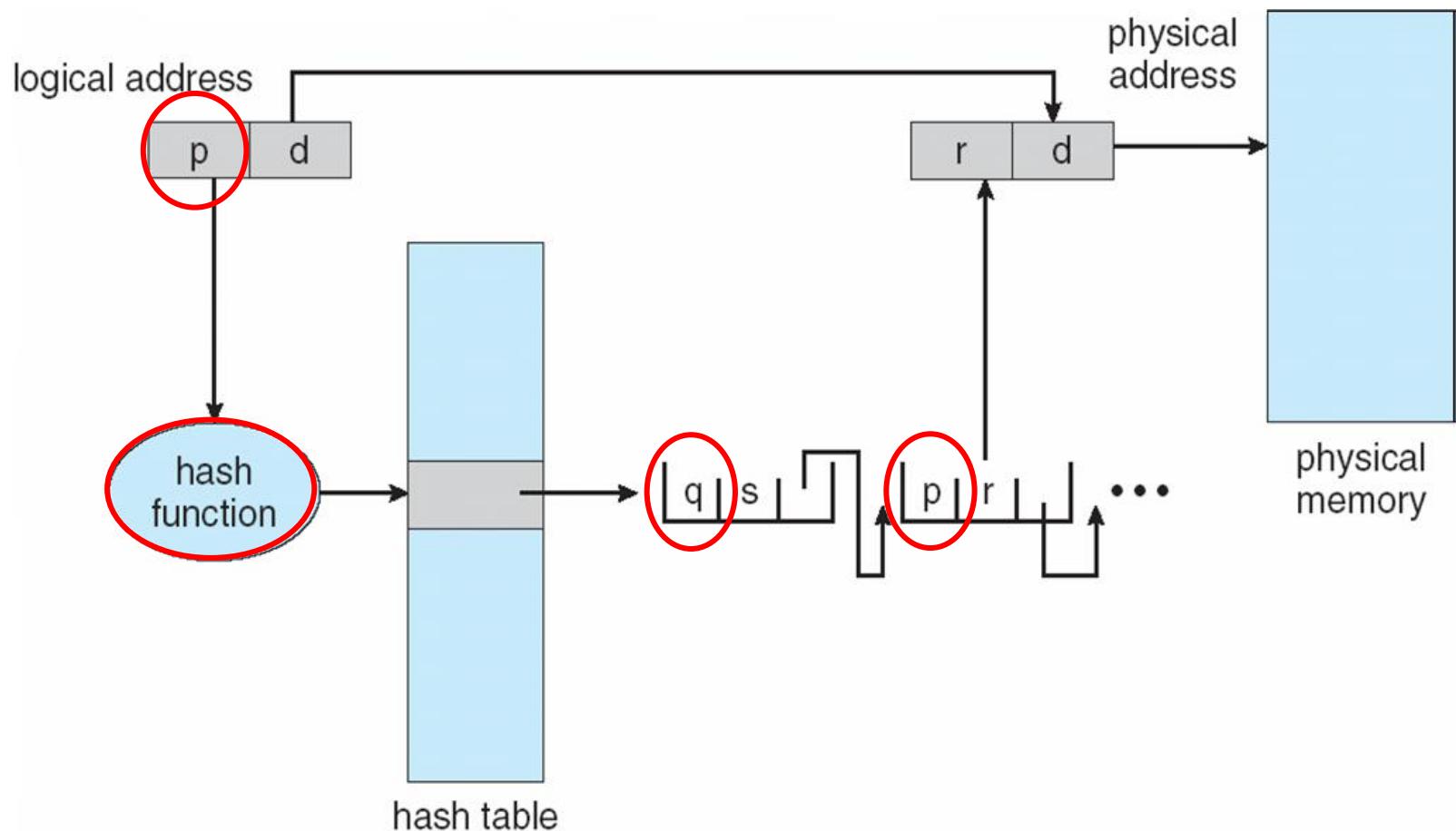
The **virtual page number is hashed into a page table**

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

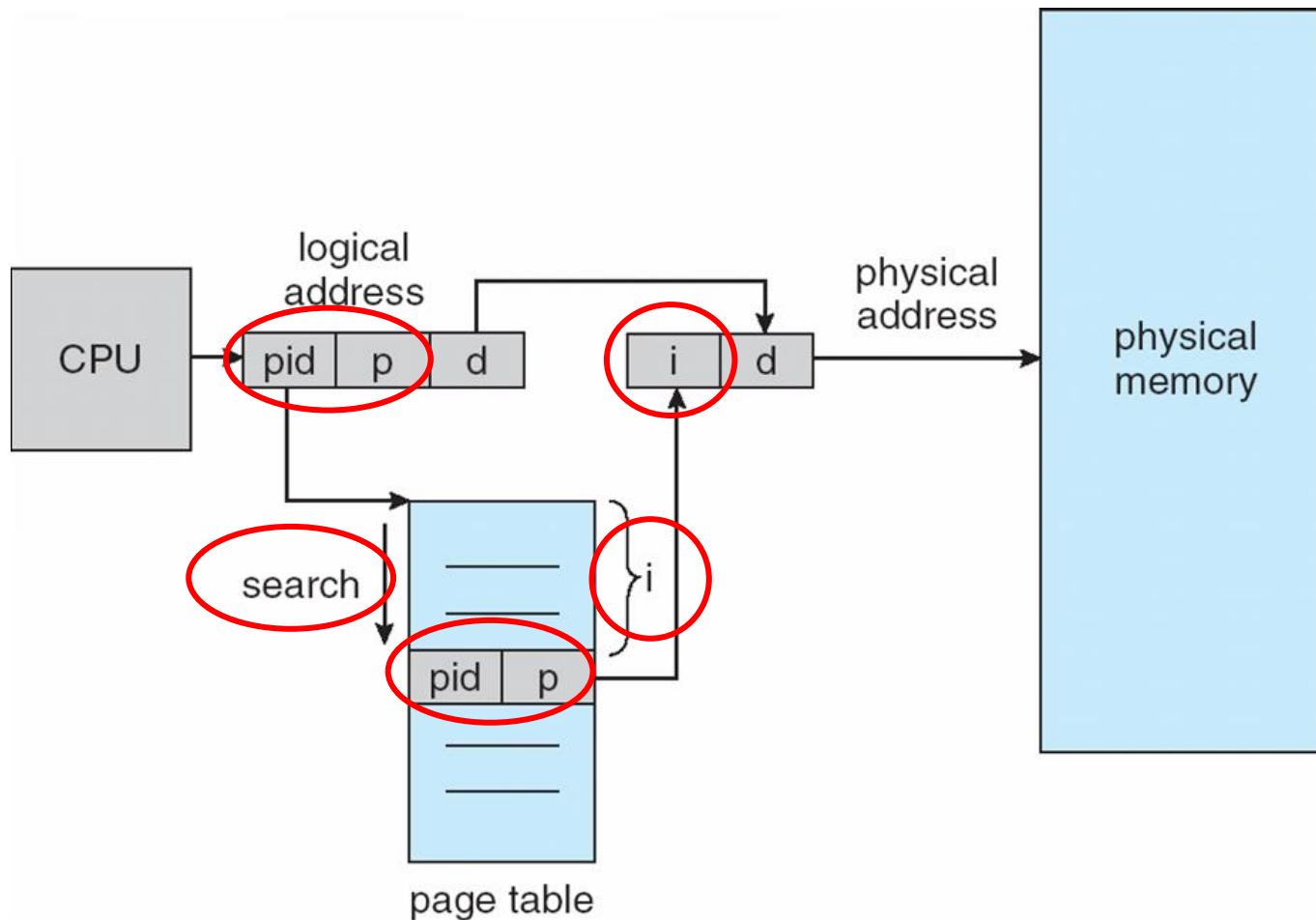
The page table is shared by all processes

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



**The search can be done sequentially, or
by hash function, or
by associative memory**

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

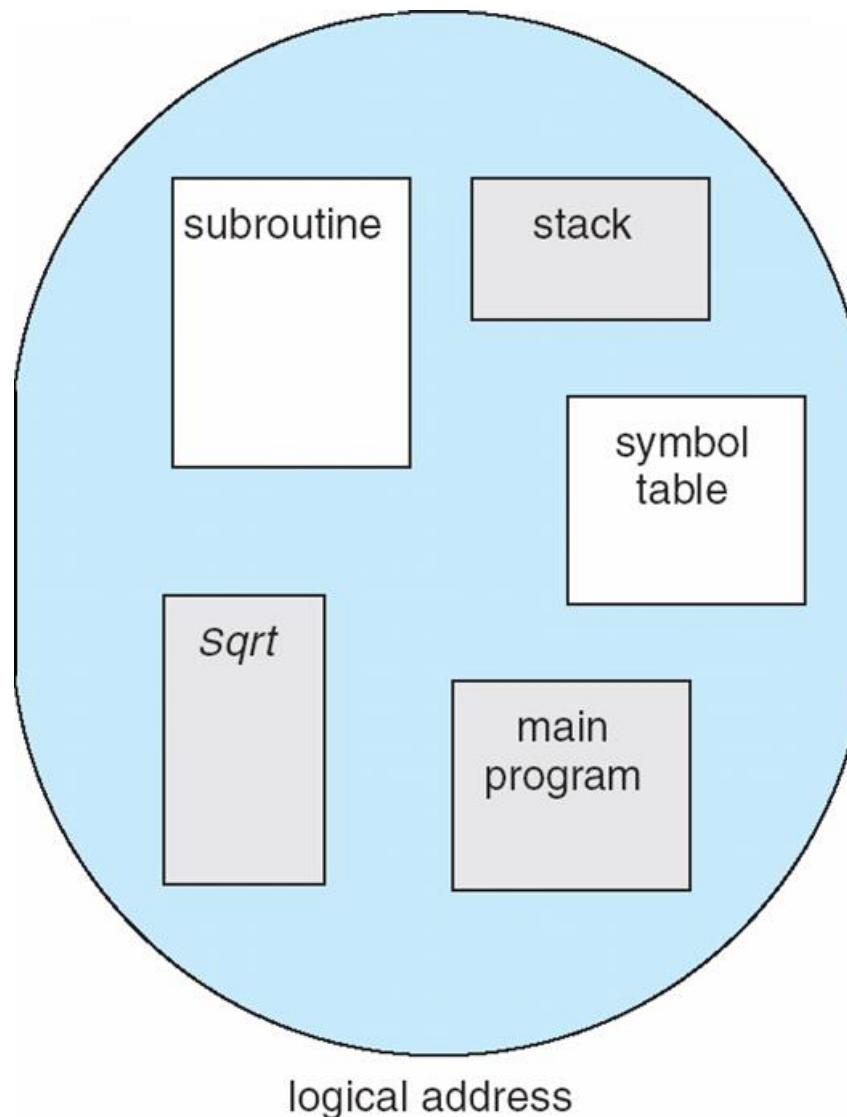
common block

stack

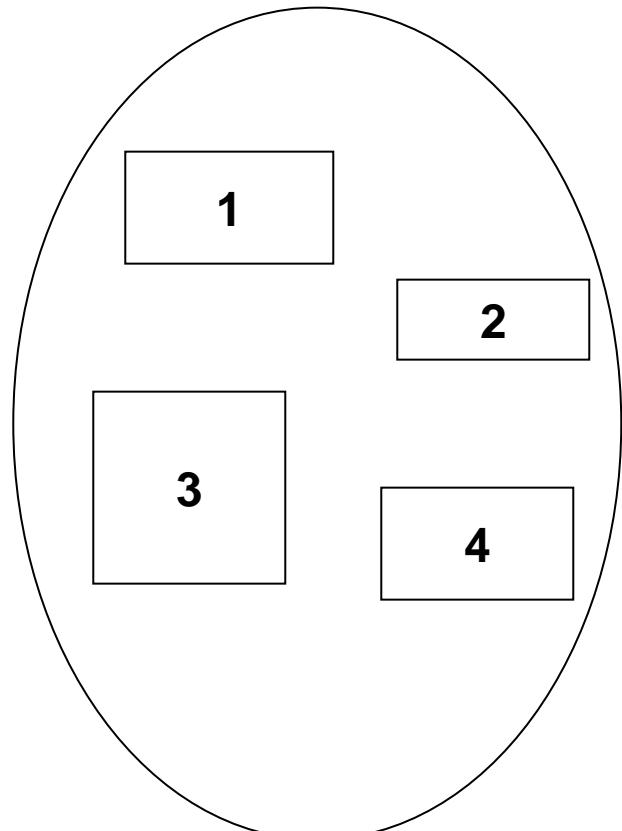
symbol table

arrays

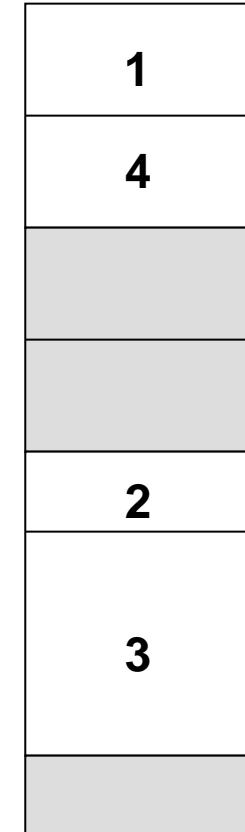
User's View of a Program



Logical View of Segmentation



user space



physical memory space

Segmentation Architecture

Logical address consists of a two-tuple:

<segment-number, offset>,

Segment table – maps two-dimensional physical addresses; each table entry has:

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

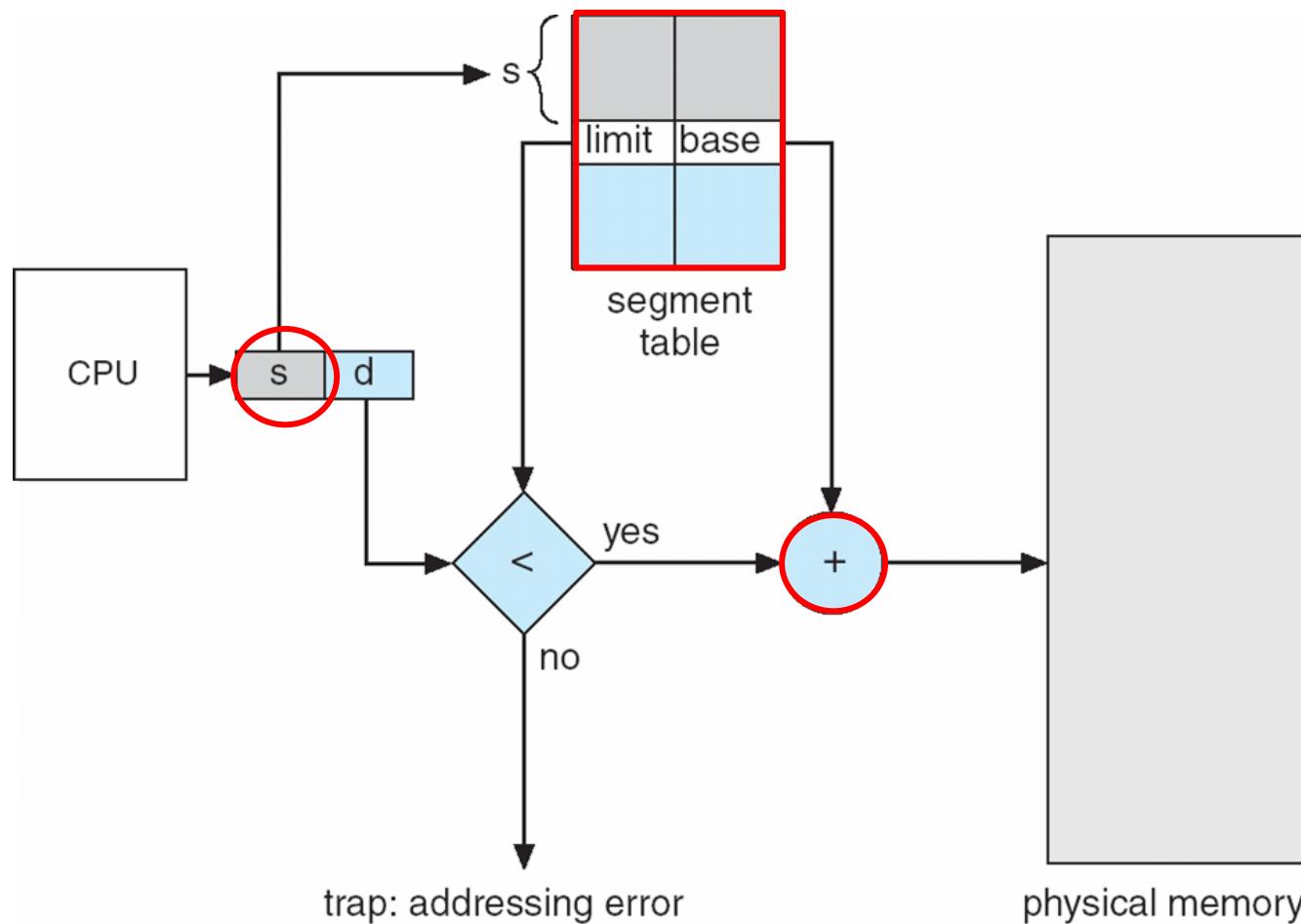
limit – specifies the length of the segment

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 $s < STLR$

Segmentation Hardware



Segmentation Architecture (Cont.)

Protection

With each entry in segment table associate:

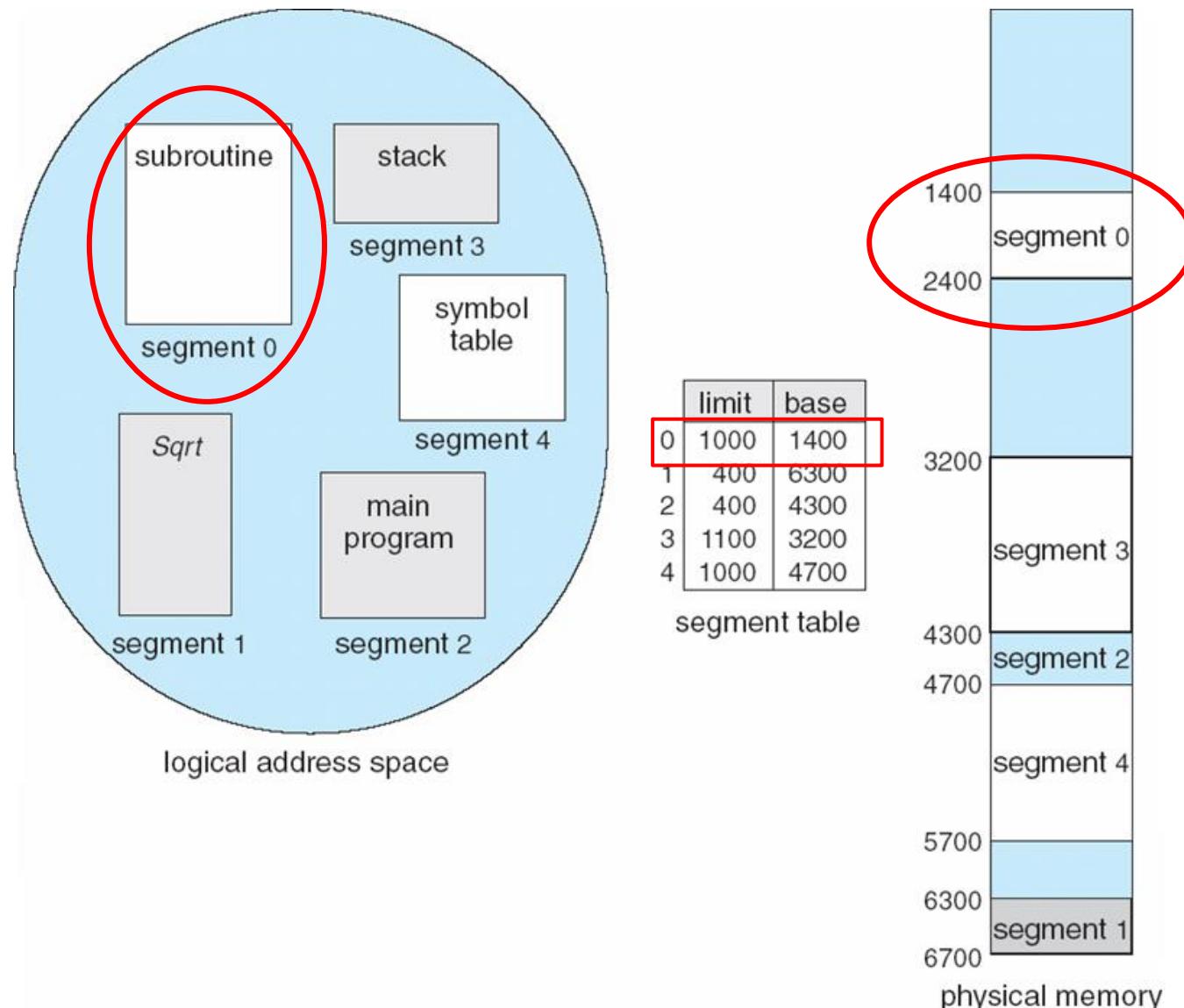
- ▶ validation bit = 0 \Rightarrow illegal segment
- ▶ read/write/execute privileges

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

Example of Segmentation



Example: The Intel Pentium

Supports both **segmentation** and **segmentation with paging**

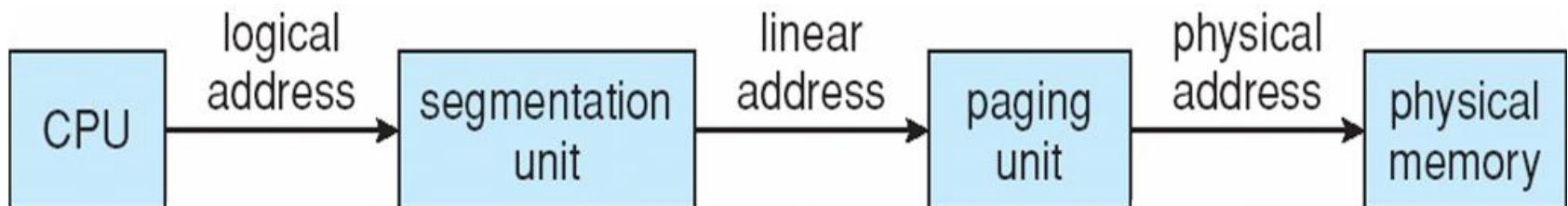
CPU generates **logical address**

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



Pentium Segmentation

A segment is allowed to be **as large as 4GB (32 bits)**, and the maximum number of segments per process is **16K**.

The logical address space is divided into **two partitions**:

The first partition up to **8K segments** that are private to that process.

The 2nd partition up to **8K segments** that are shared among all processes.

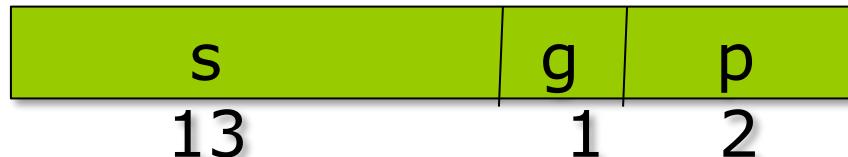
Local Descriptor Table (LDT): Information about the 1st partition

Global Descriptor Table (GDT): Information about the 2nd partition.

Each entry of LDT and GDT is an **8-byte descriptor** of a particular segment.

Pentium Segmentation

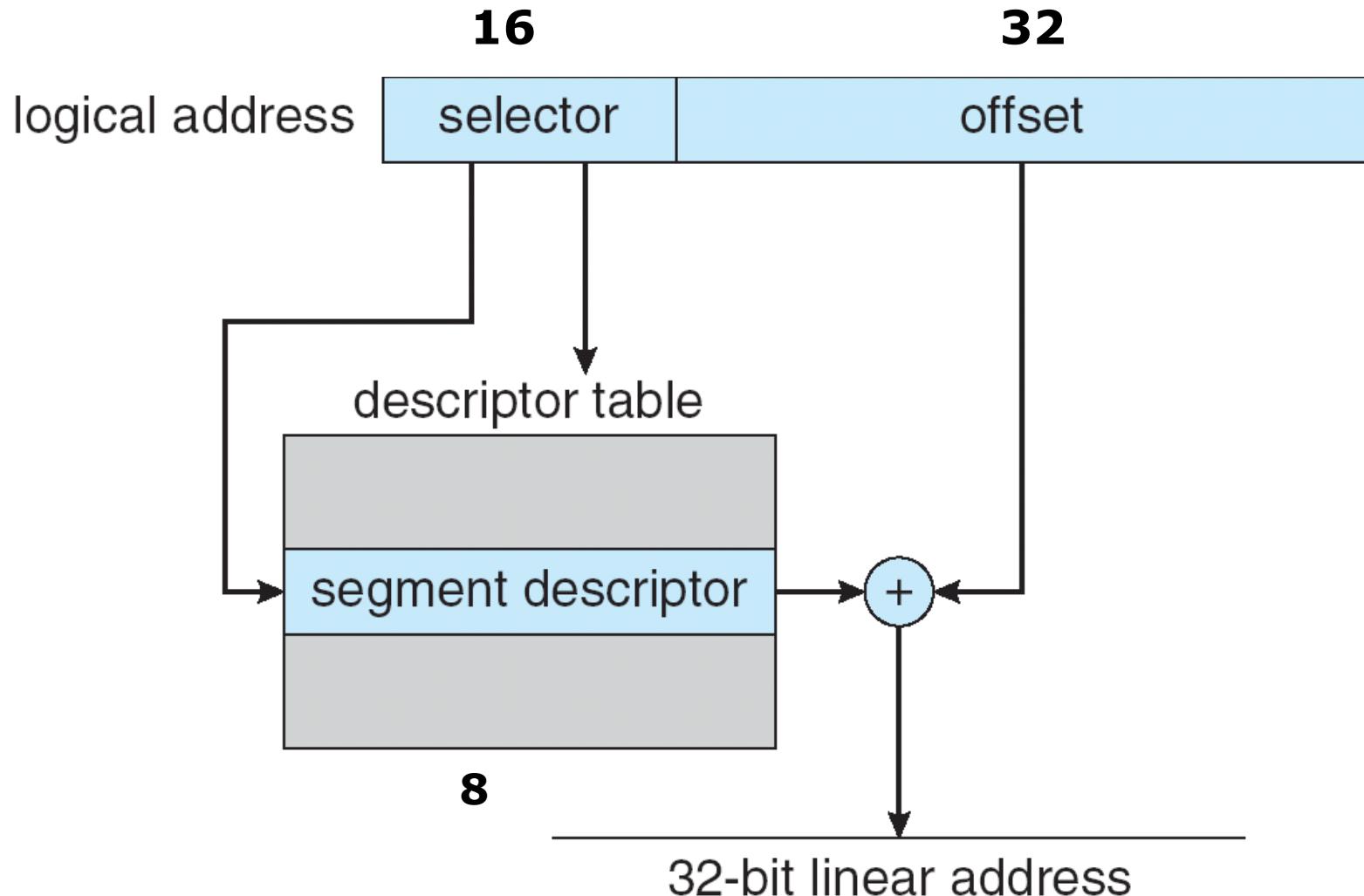
The logical address is a pair (**selector**, **offset**), where the **selector** is a 16-bit number and offset is a 32-bit number:



The machine has **six segment registers**, allowing six segments to be addressed at any one time by a process.

The **linear address is 32 bits long** and is formed as follows.

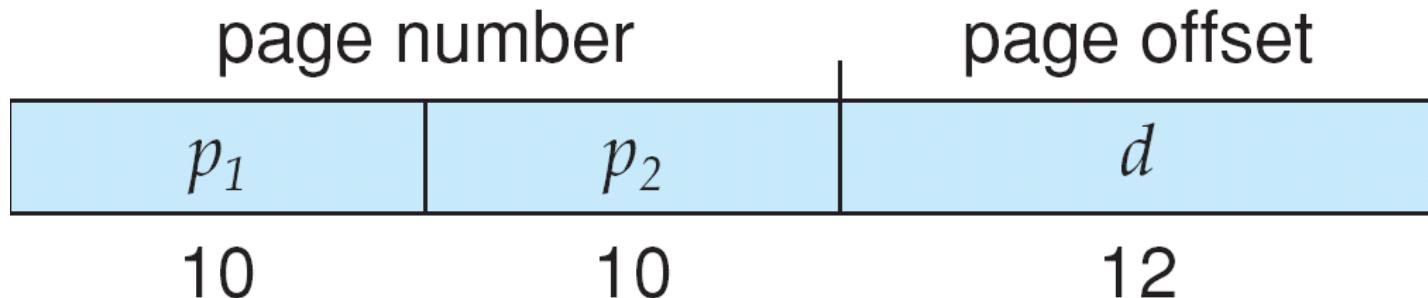
Intel Pentium Segmentation



Pentium Paging

A page is allowed to be as 4kB or 4MB.

For 4-KB pages, a two level paging scheme is used
(32-bit linear address)

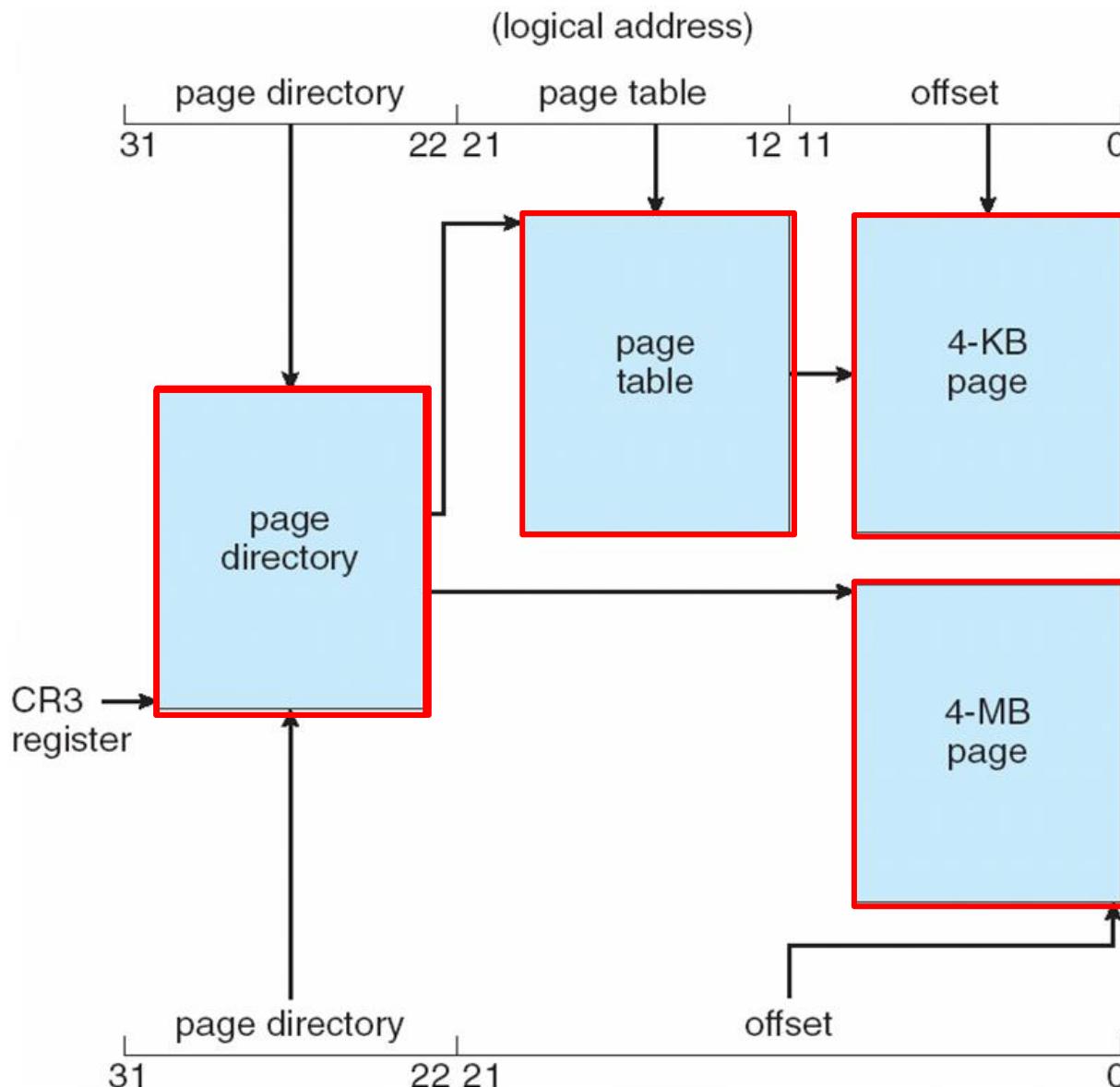


The address-translation scheme is shown as follows.

Page directory

Page size flag

Pentium Paging Architecture



Linux on Pentium Systems

Linux does not rely on segmentations and used it minimally.

On the Pentium, Linux uses only six segments:

A segment for kernel code

A segment for kernel data

A segment for user code

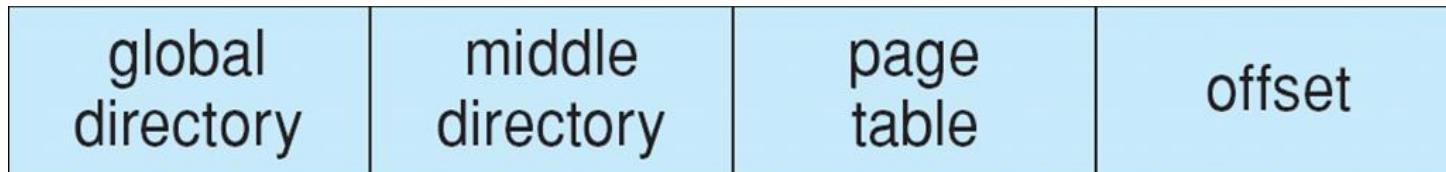
A segment for user data

A task-state segment (TSS)

A default LDT segment

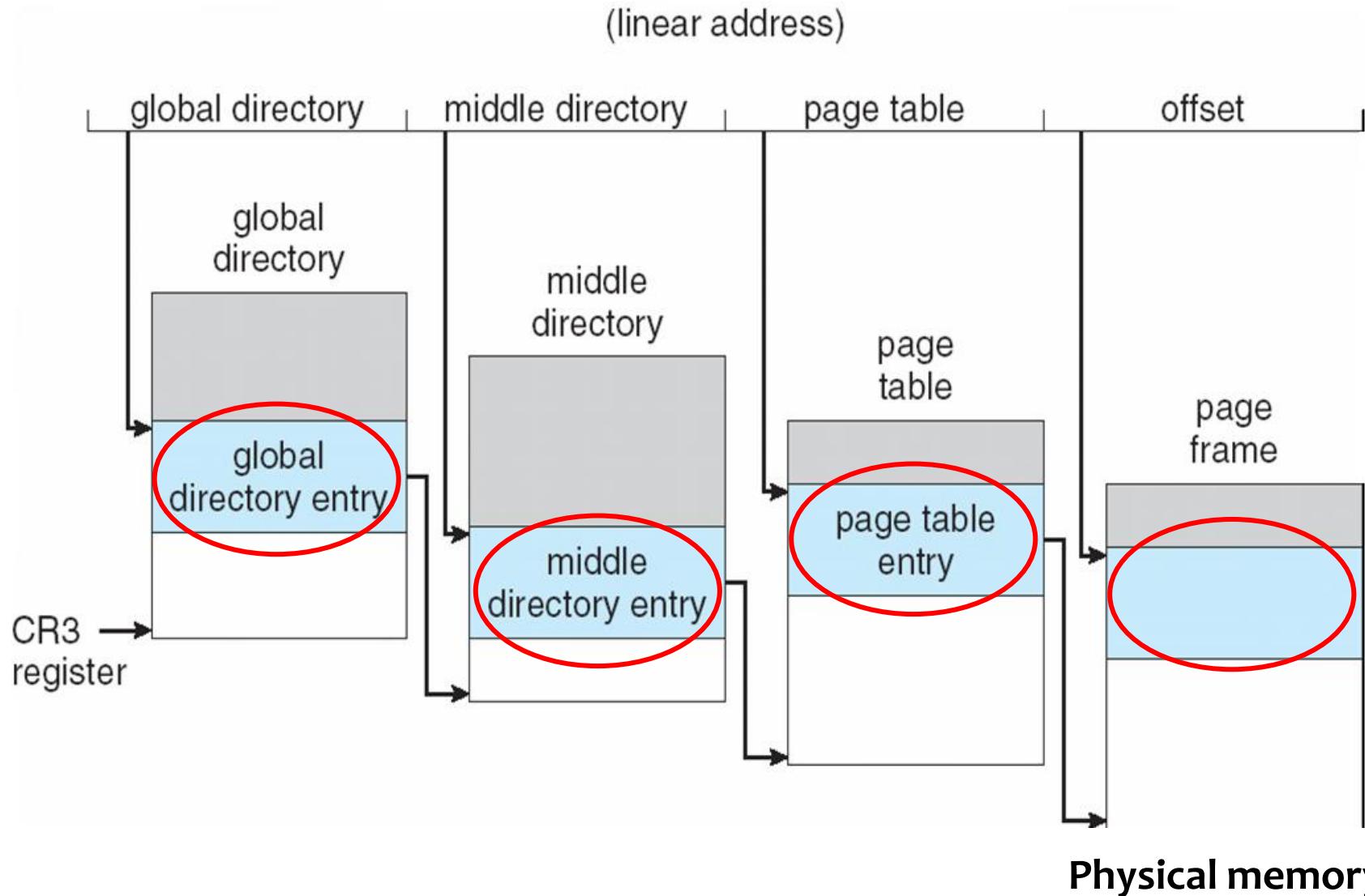
Linear Address in Linux

- The linear address in Linux is broken **into four parts:**

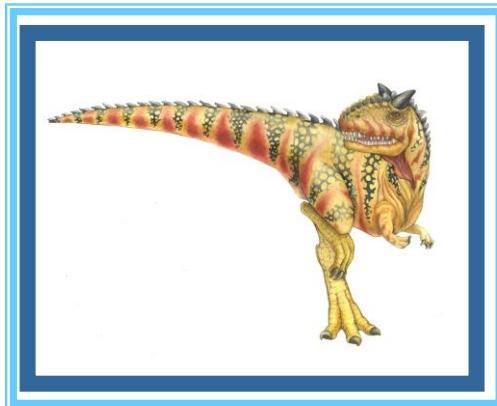


- Each task in Linux has its own set of page tables and the **CR3 register points to the global directory** for the task currently executing.
- During a **context switch**, the value of CR3 register is restored in the **TSS segments of the tasks involved in the context switch**.

Three-level Paging in Linux



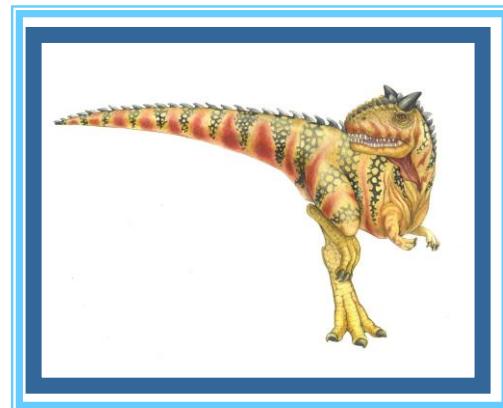
End of Chapter 8



Chapter 9:

Virtual-Memory

Management



Chapter 9: Virtual-Memory Management

Background

Demand Paging

Copy-on-Write

Page Replacement

Allocation of Frames

Thrashing

Memory-Mapped Files

Allocating Kernel Memory

Other Considerations

Operating-System Examples

Objectives

- 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**

Background

Virtual memory – separation of user logical memory from physical memory.

Only part of the program needs to be in memory for execution

Logical address space can therefore be much larger than physical address space

Allows address spaces to be shared by several processes

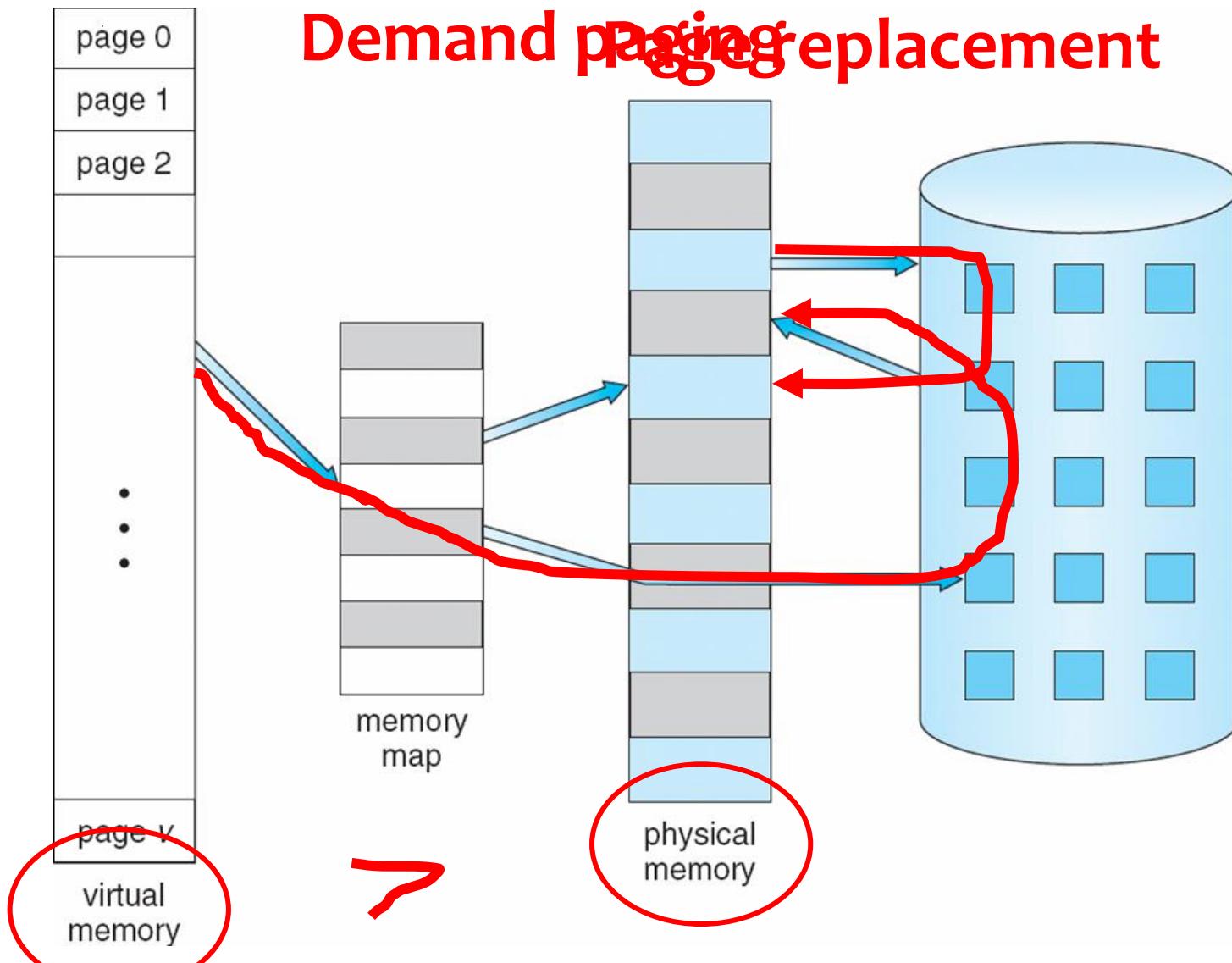
Allows for more efficient process creation

Virtual memory can be implemented via:

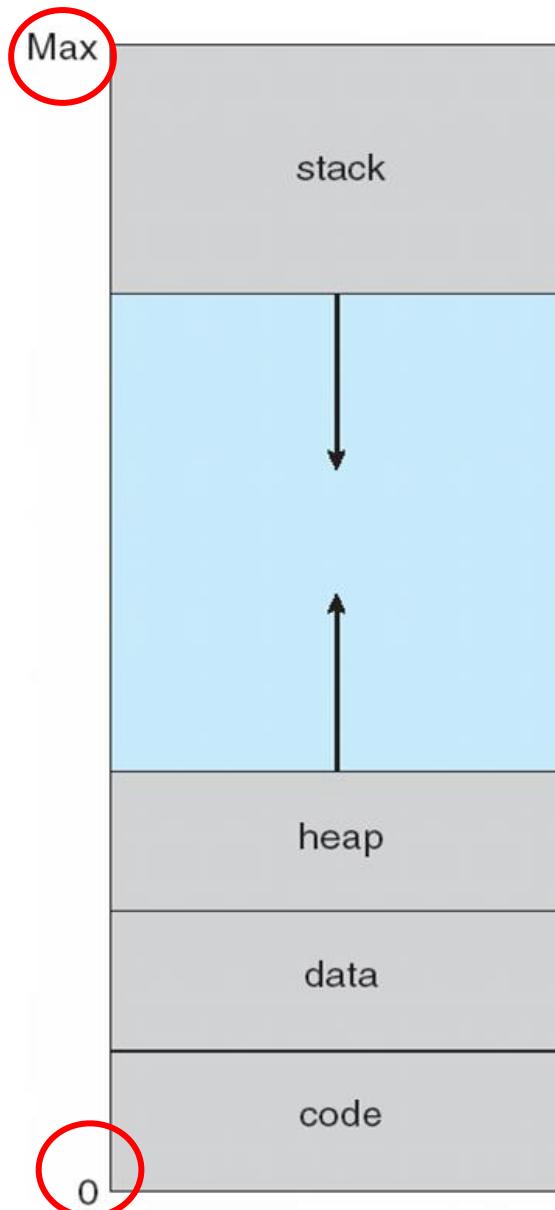
Demand paging

Demand segmentation

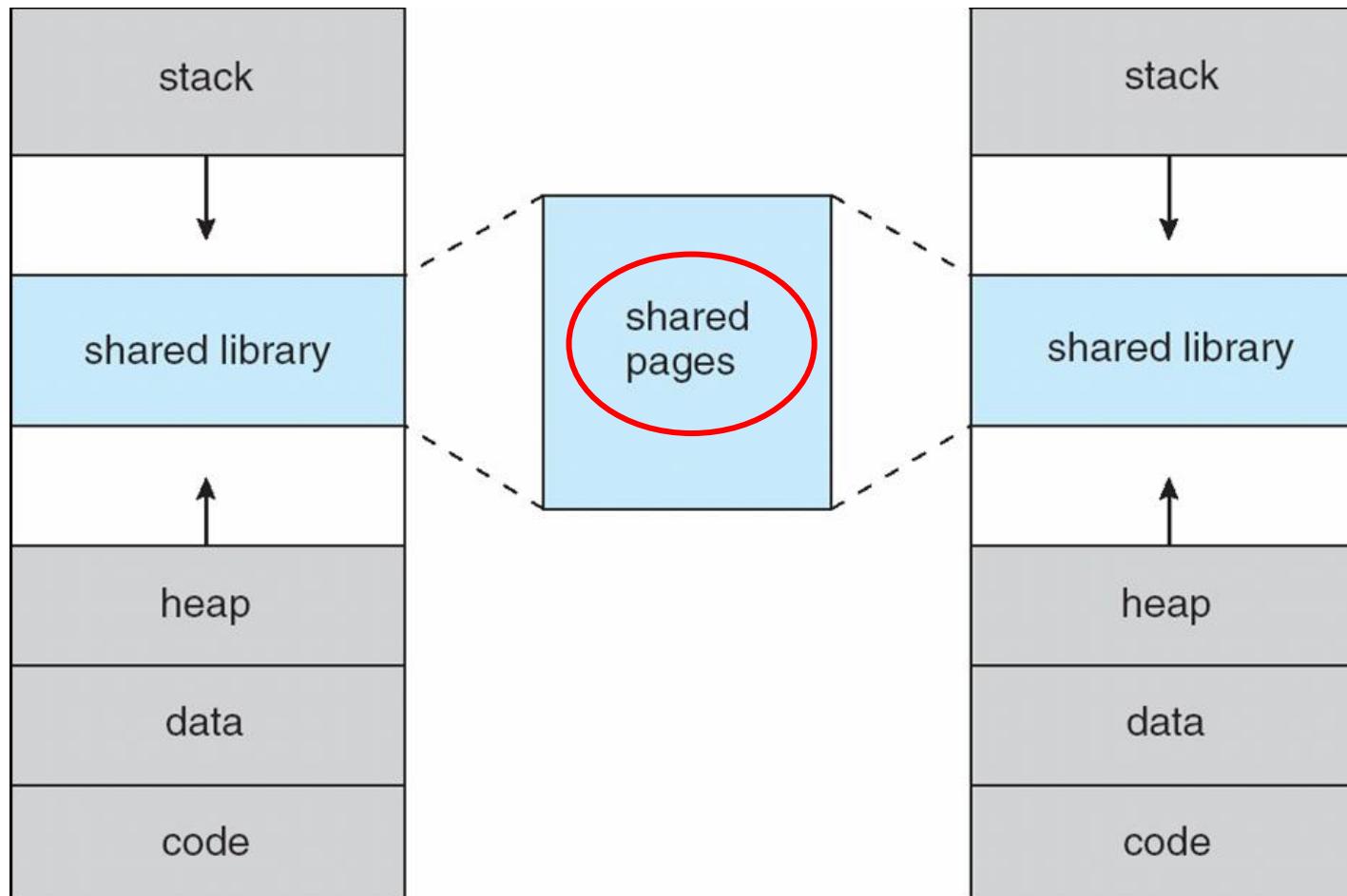
Virtual Memory > Physical Memory



Virtual-address Space



Shared Library Using Virtual Memory



Demand Paging

Bring a page into memory only when it is needed

- Less I/O needed

- Less memory needed

- Faster response

- More users

Page is needed \Rightarrow reference to it

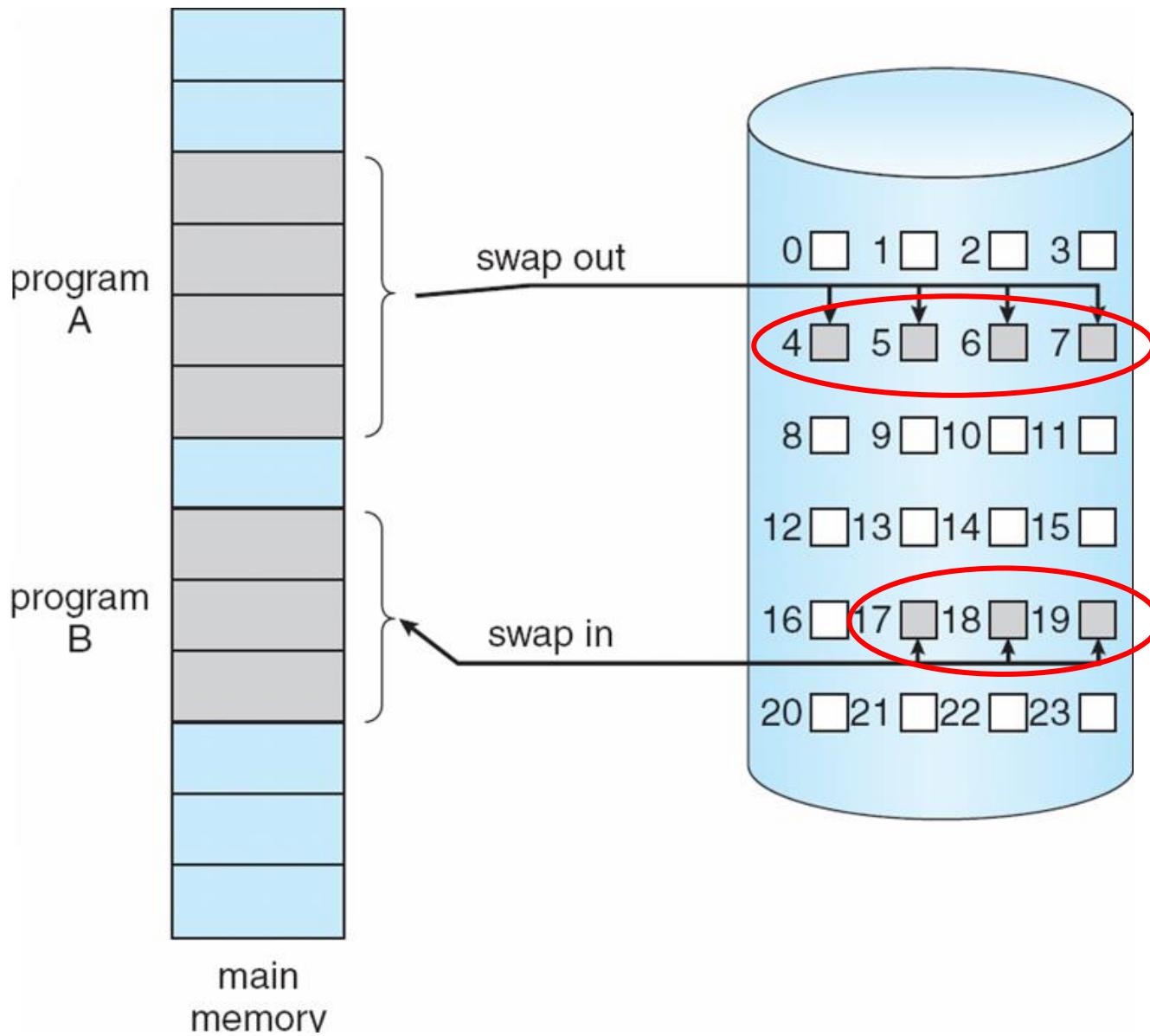
- invalid reference \Rightarrow abort

- not-in-memory \Rightarrow bring to memory

Lazy swapper – never swaps a page into memory unless page will be needed

Swapper that deals with pages is a **pager**

Transfer of a Paged Memory to Contiguous Disk Space



Valid-Invalid Bit

With each page table entry a valid-invalid bit is associated
(**v** ⇒ in-memory, **i** ⇒ not-in-memory)

Initially valid-invalid bit is set to **i** on all entries

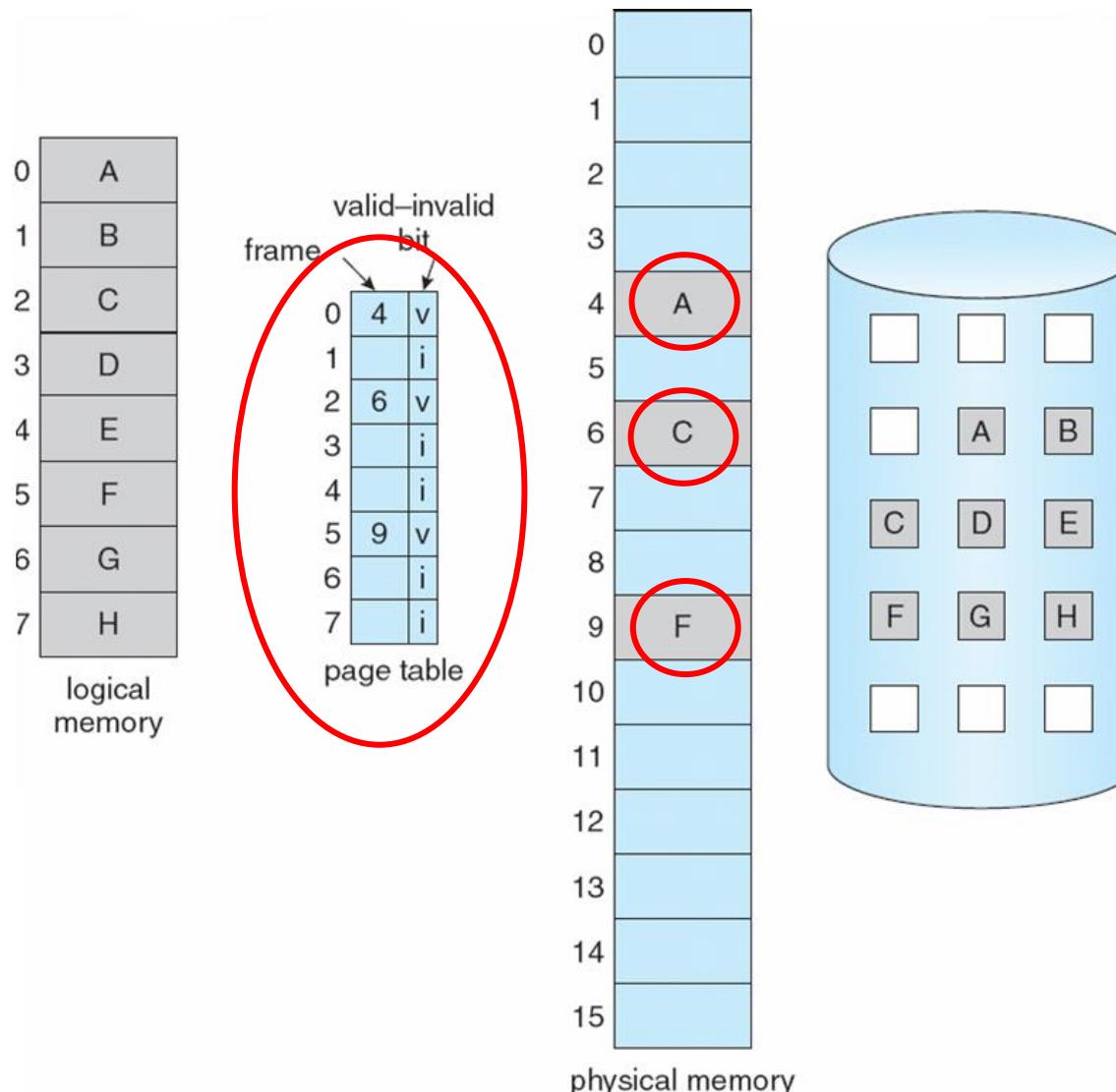
Example of a page table snapshot:

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

page table

During address translation, if valid-invalid bit in page table entry is **i** ⇒ **page fault**

Page Table When Some Pages Are Not in Main Memory



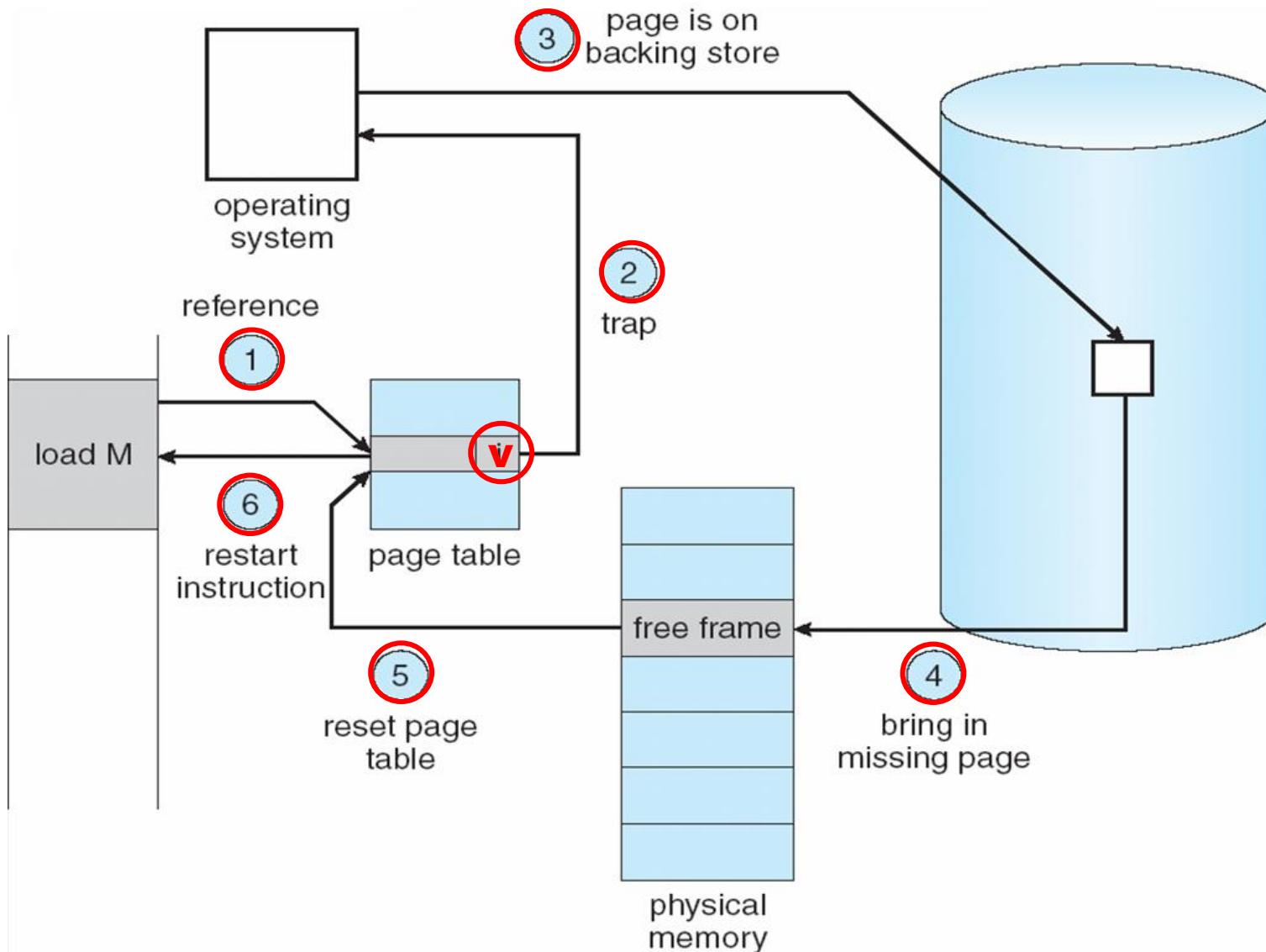
Page Fault

If there is a reference to a page, **first reference** to that page will trap to operating system:

page fault

1. Operating system looks at another table to decide:
 - Invalid reference \Rightarrow abort
 - Just not in memory
2. Get empty frame from physical memory
3. Swap page into frame from disk
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Performance of Demand Paging

Page Fault Rate $0 \leq p \leq 1.0$

if $p = 0$ no page faults

if $p = 1$, every reference is a fault

Effective Access Time (EAT)

$EAT = (1 - p) \times \text{memory access}$

+ p (page fault overhead

+ swap page out

+ swap page in

+ restart overhead)

Demand Paging Example

Memory access time = 200 nanoseconds

Average page-fault service time = 8 milliseconds

$$\text{EAT} = (1 - p) \times 200 + p \text{ (8 milliseconds)}$$

$$= (1 - p) \times 200 + p \times 8,000,000$$

$$= 200 + p \times 7,999,800$$

If one access out of 1,000 causes a page fault, then

$$\text{EAT} = 8.2 \text{ microseconds.}$$

This is a slowdown by a factor of 40!!

Process Creation

Virtual memory allows other benefits during process creation:

- **Copy-on-Write**
- **Memory-Mapped Files (later)**

Copy-on-Write

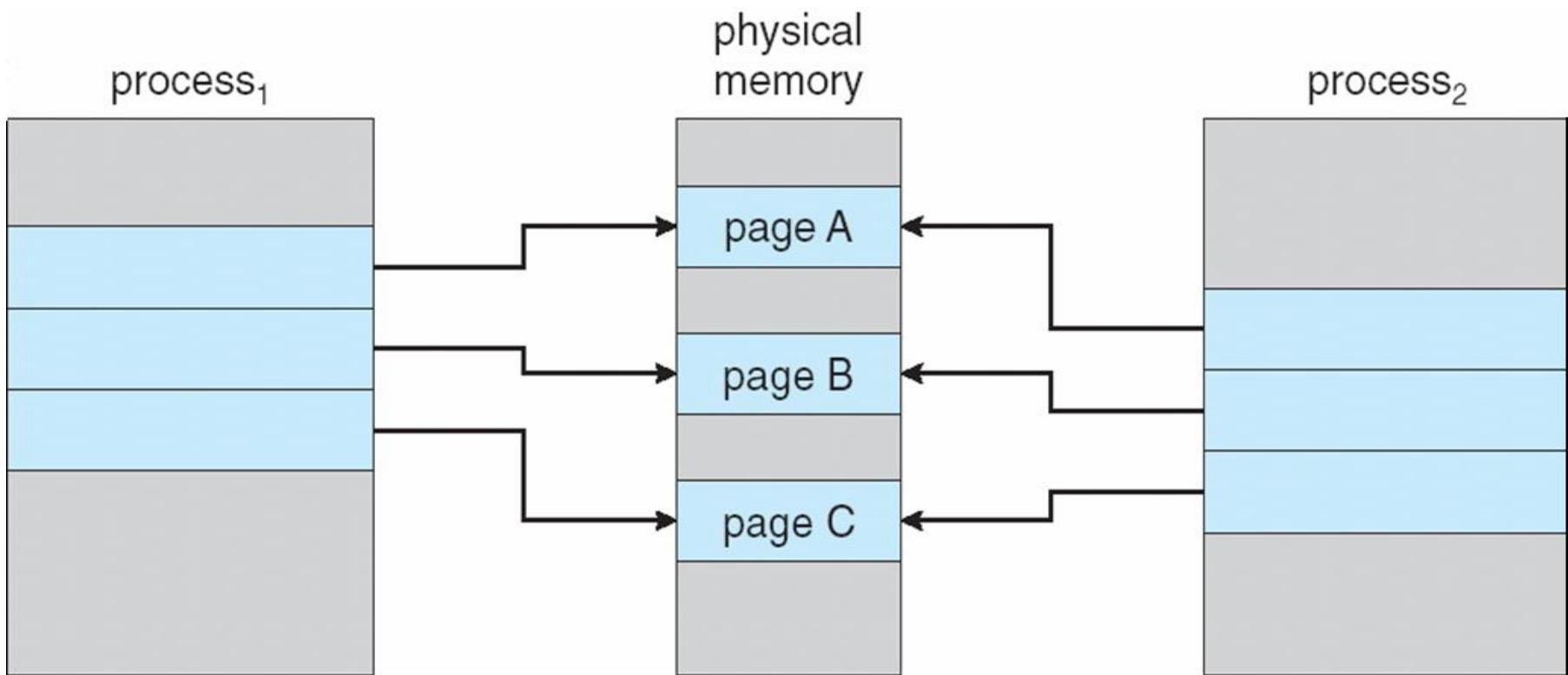
Copy-on-Write (COW) allows both parent and child processes to initially *share the same pages in memory*

If either process modifies a shared page, only then is the page copied

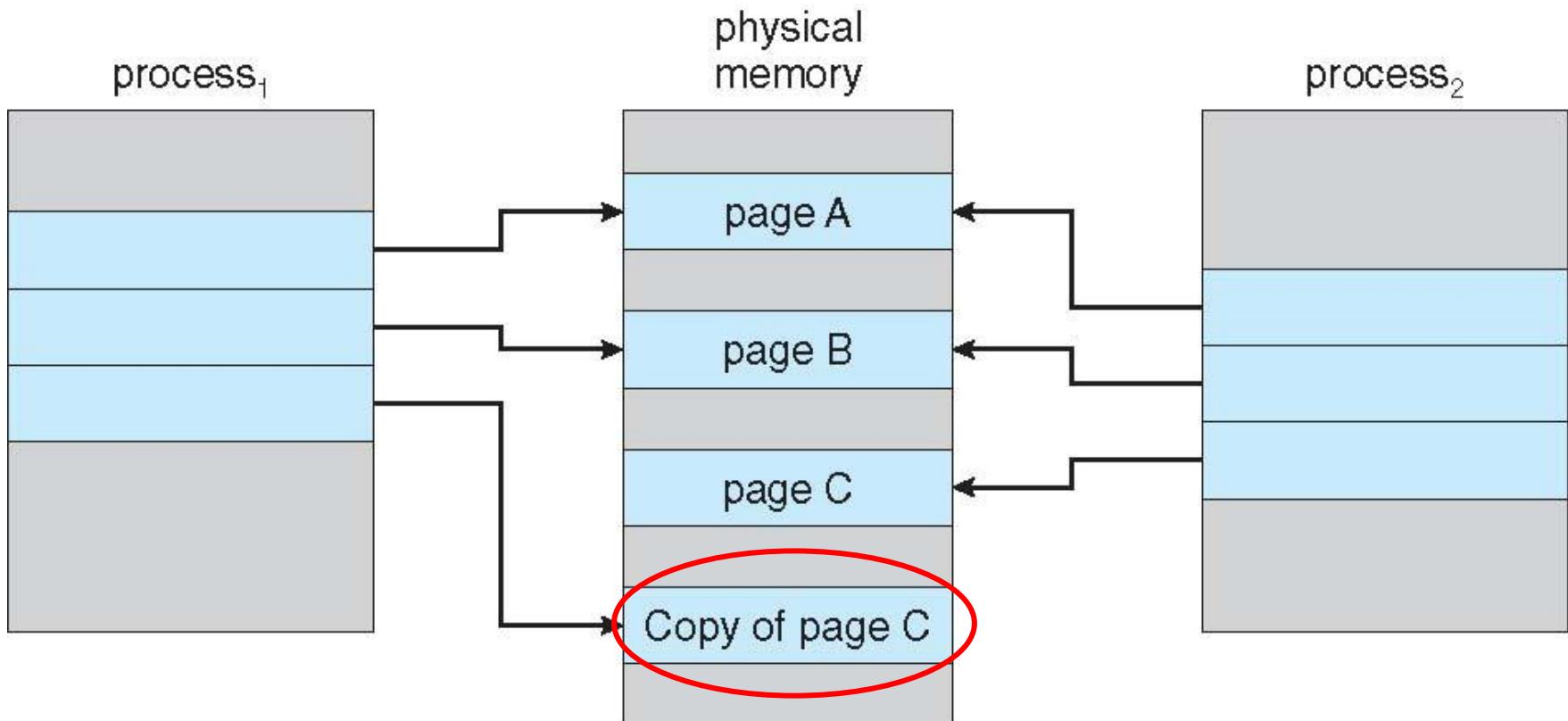
COW allows more efficient process creation as **only modified pages are copied**

Free pages are allocated from a pool of zeroed-out pages

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



What happens if there is no free frame?

Page replacement – find some page in memory, but not really in use, swap it out

algorithm

performance – want an algorithm which will result in **minimum number of page faults**

Same page may be brought into memory several times

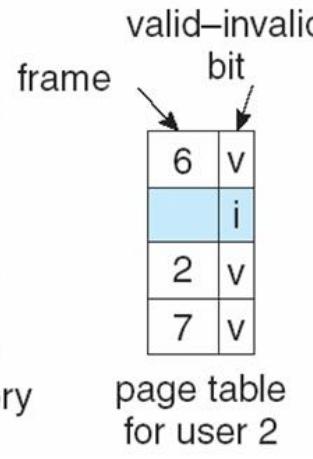
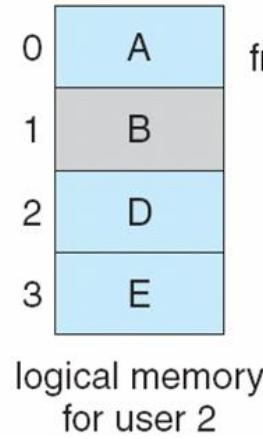
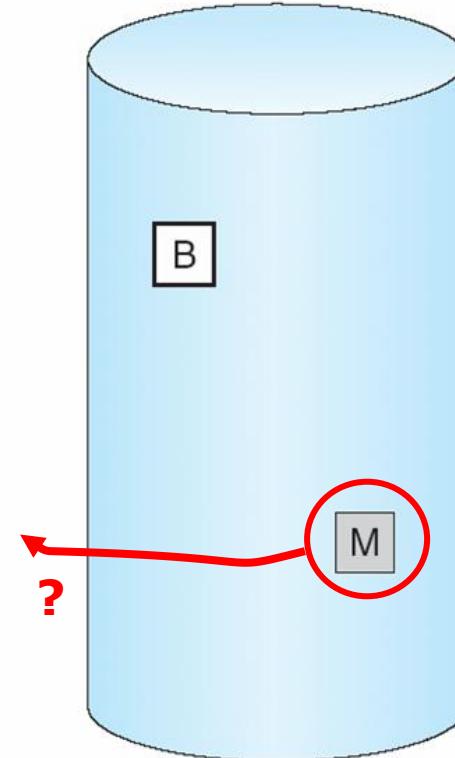
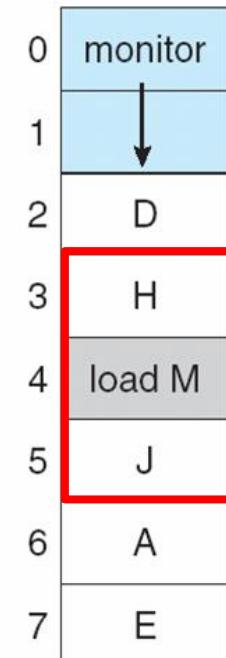
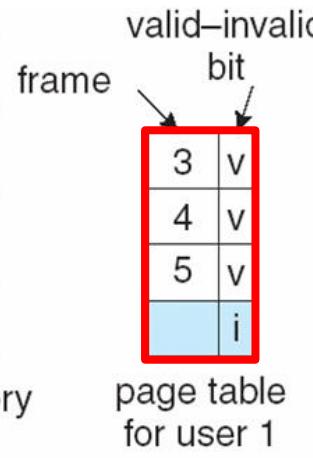
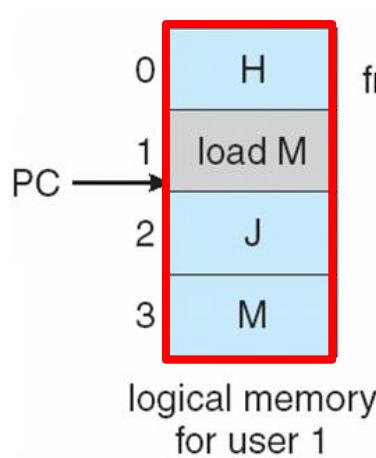
Page Replacement

Prevent over-allocation of memory by modifying **page-fault service routine** to include page replacement

Use **modify (dirty) bit** to reduce overhead of page transfers – **only modified pages are written to disk**

Page replacement completes separation between logical memory and physical memory – **large virtual memory can be provided on a smaller physical memory**

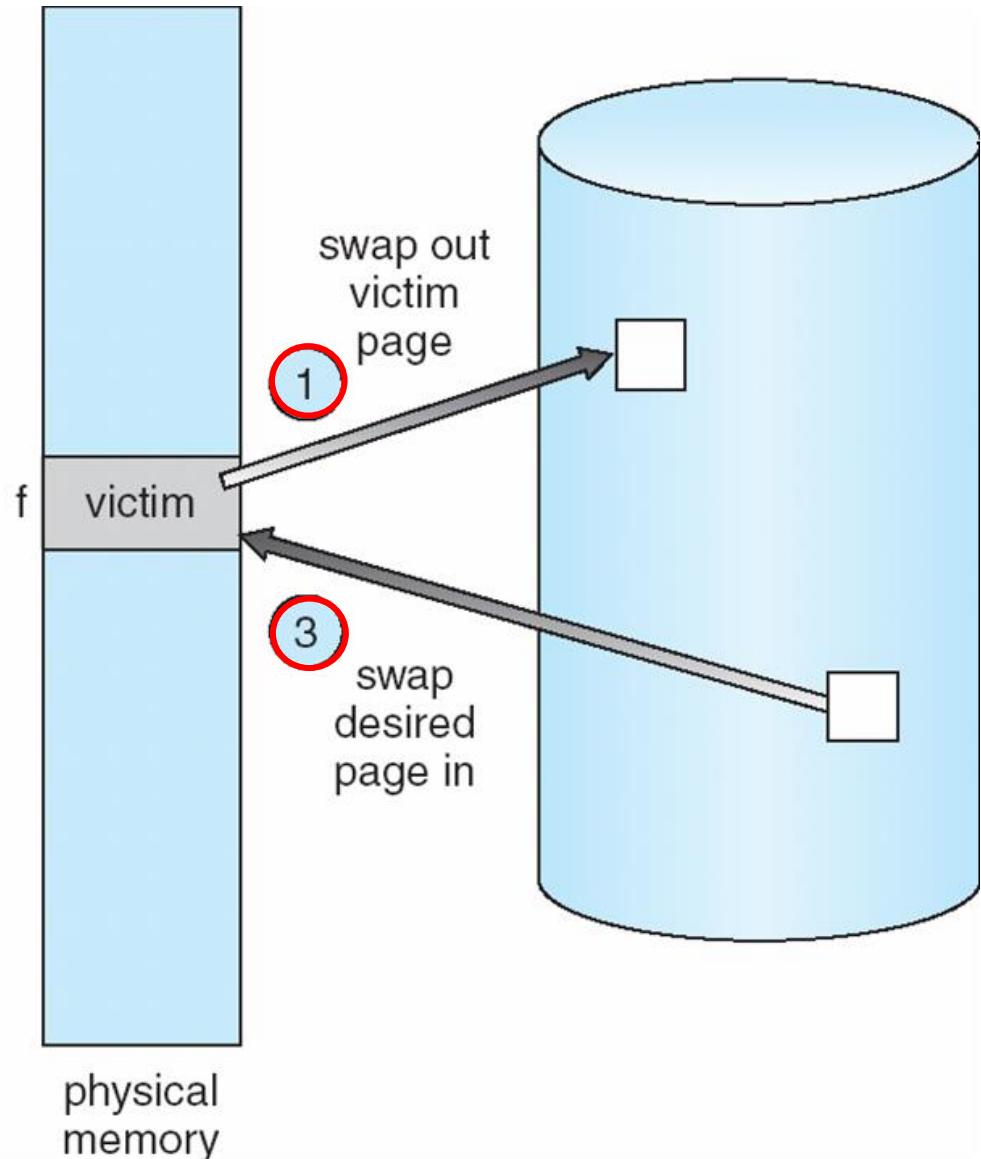
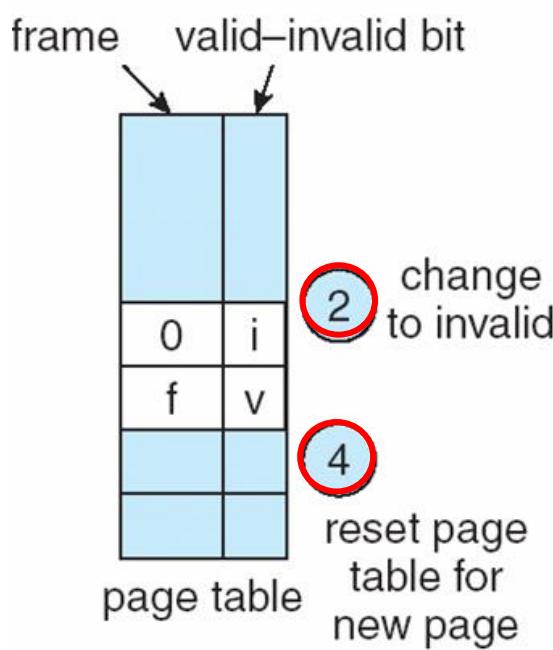
Need For Page Replacement



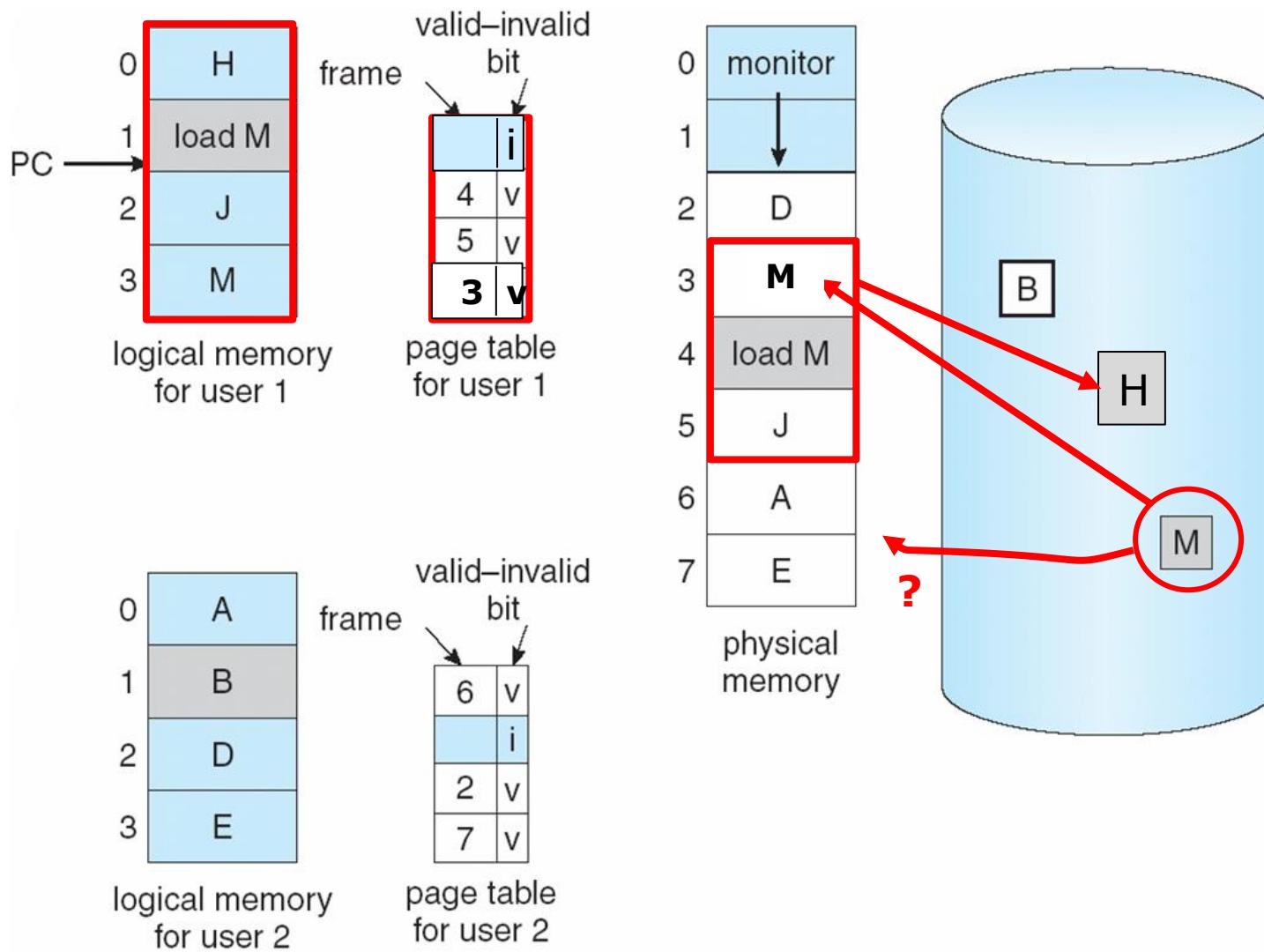
Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to **select a victim frame**
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Restart the process

Page Replacement



Page Replacement Example



Page Replacement Algorithms

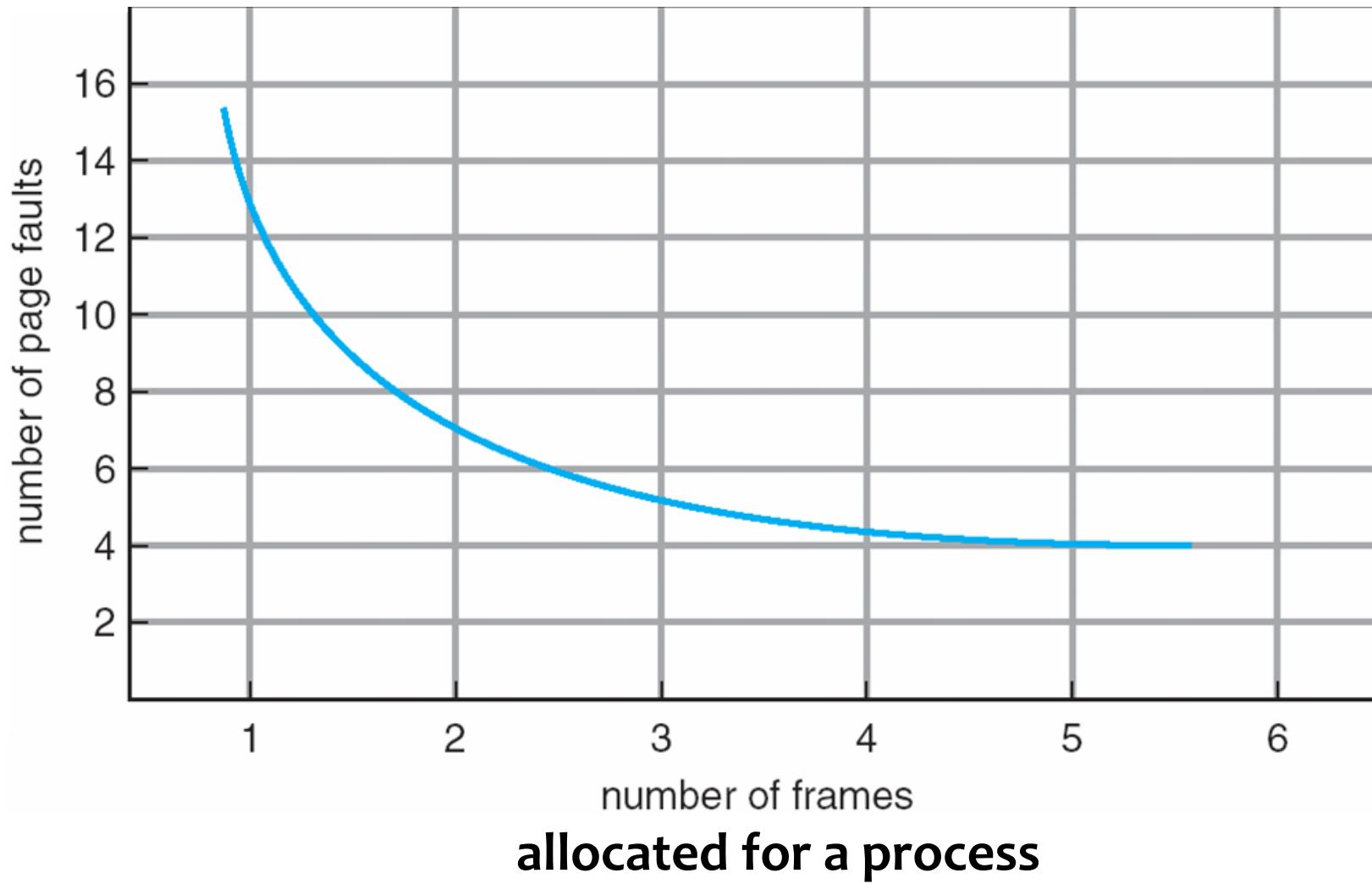
Want lowest **page-fault rate**

Evaluate algorithm by running it on a particular string of memory references (**reference string**) and computing the number of page faults on that string

In all our examples, the reference string is

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

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 frames (3 pages can be in memory at a time per process)

1	1	4	5
2	2	1	3 9 page faults
3	3	2	4

1	1	5	4
2	2	1	5 10 page faults
3	3	2	
4	4	3	

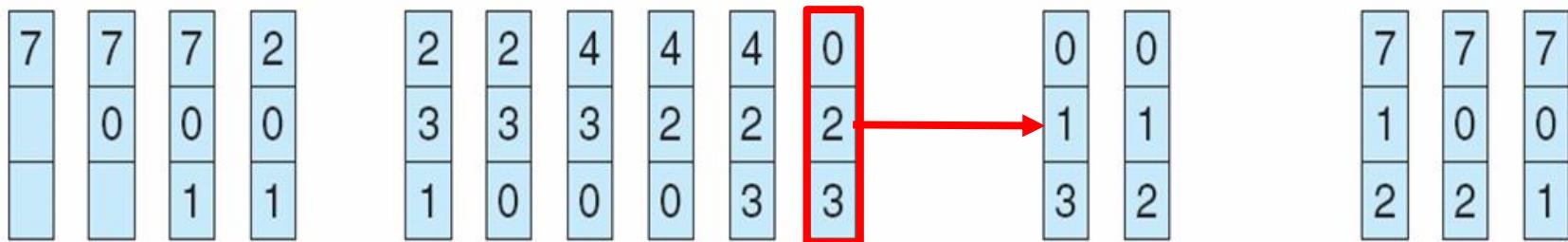
4 frames

Belady's Anomaly: more frames \Rightarrow more page faults

FIFO Page Replacement

reference string

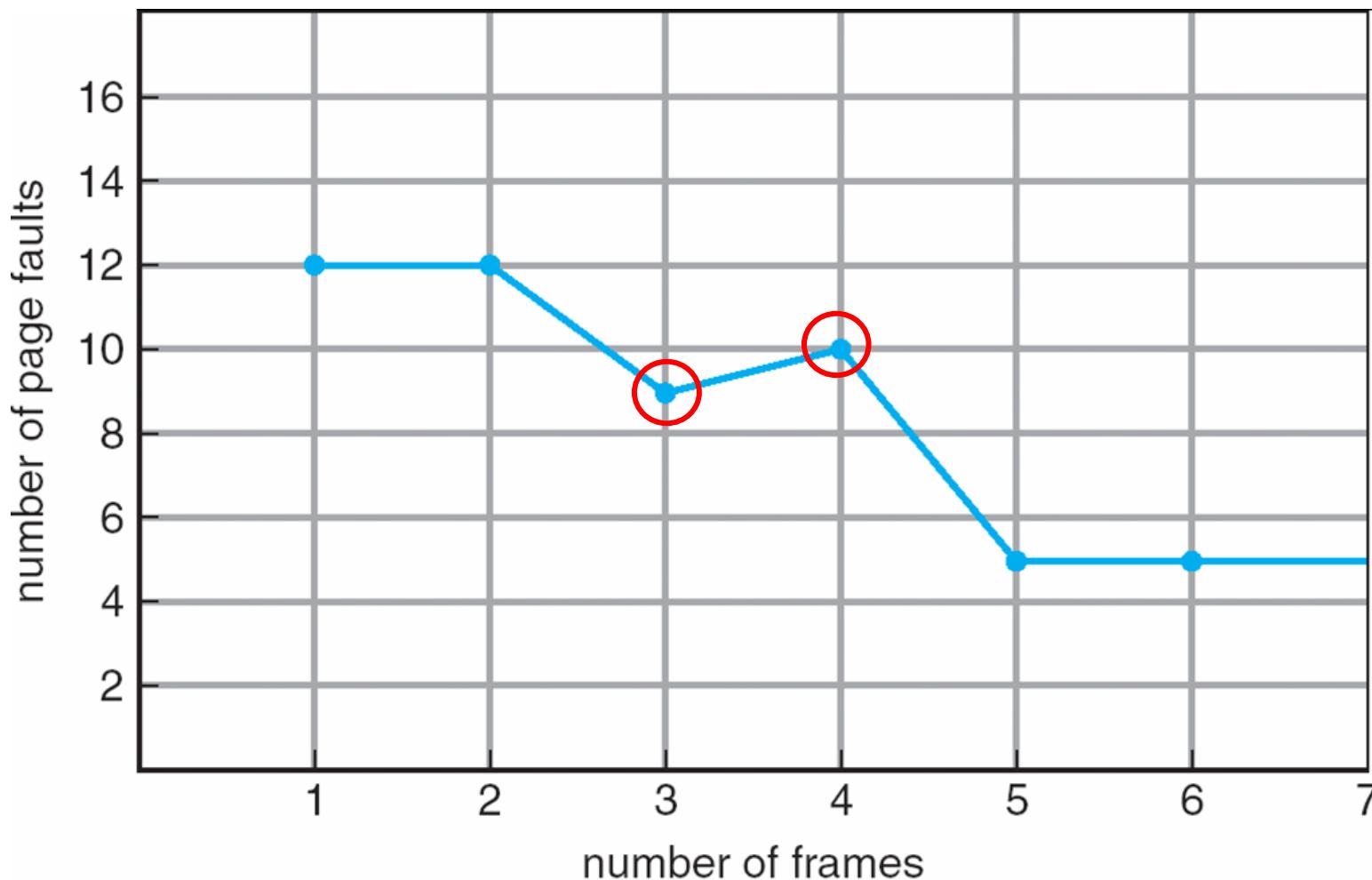
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

Total number of page faults = 15

FIFO Illustrating Belady's Anomaly

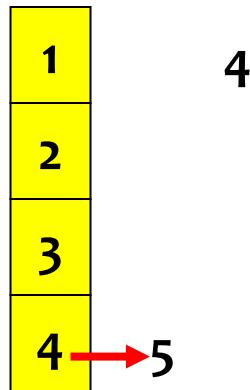


Optimal Algorithm

Replace page that will not be used for **longest period of time** (最久之後用到)

4 frames example

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



6 page
faults

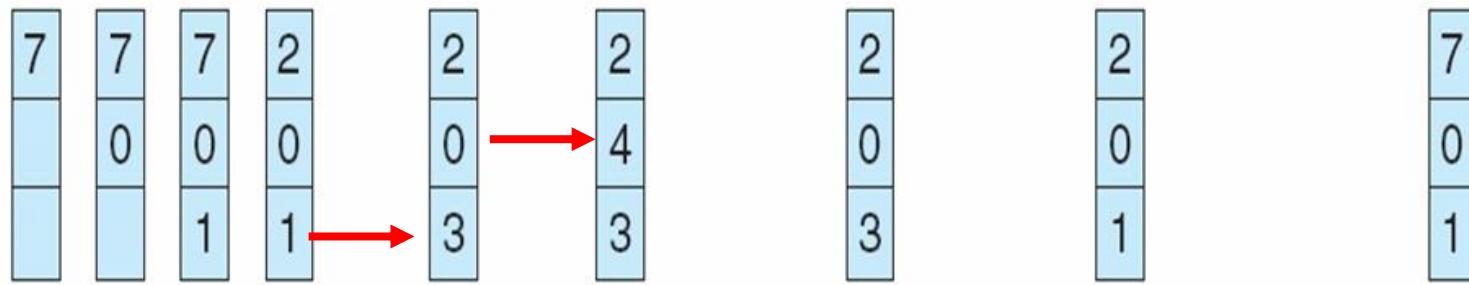
How do you know this?

Used for measuring how well your algorithm performs (lower bound)

Optimal Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

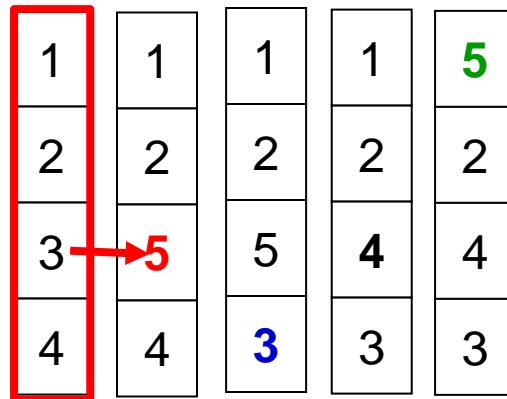


page frames

Total number of page faults = 9

Least Recently Used (LRU) Algorithm

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



(最早之前用過)

Counter implementation

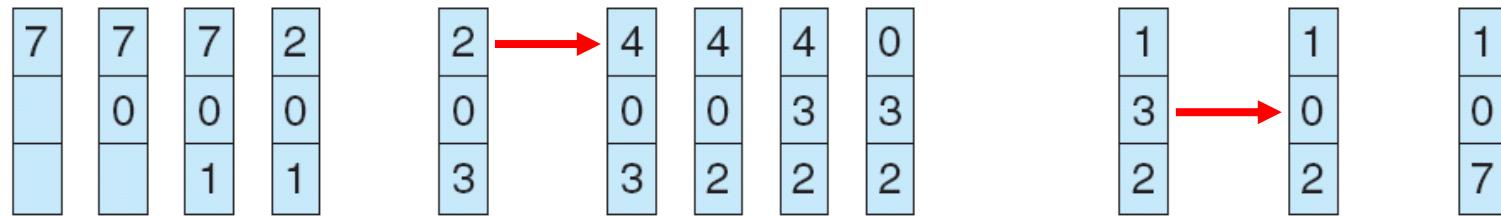
Every page entry has a counter; every time page is referenced through this entry, copy the **clock** into the counter

When a page needs to be changed, look at the counters to determine which is to change

LRU Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 3 0 3 2 1 2 0 1 7 0 1



page frames

Total number of page faults = 12

LRU Algorithm (Cont.)

Stack implementation – keep a stack of page numbers in a double link form:

Page referenced:

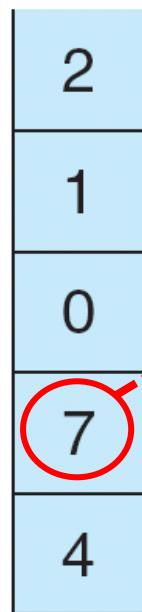
- ▶ move it to the top
- ▶ requires 6 pointers to be changed

No search for replacement – **Replace the bottom page of the stack**

Stack implementation

reference string

4 7 0 7 1 0 1 2 1 2 7 1 2



stack
before
a



stack
after
b

7

2

1

0

4

2 7 1 2 1 2

a b



LRU Approximation Algorithms

Reference bit

With each page associate a bit, initially = 0

When page is referenced bit set to 1

Replace the one which is 0 (if one exists)

- ▶ We do not know the order, however

Second chance

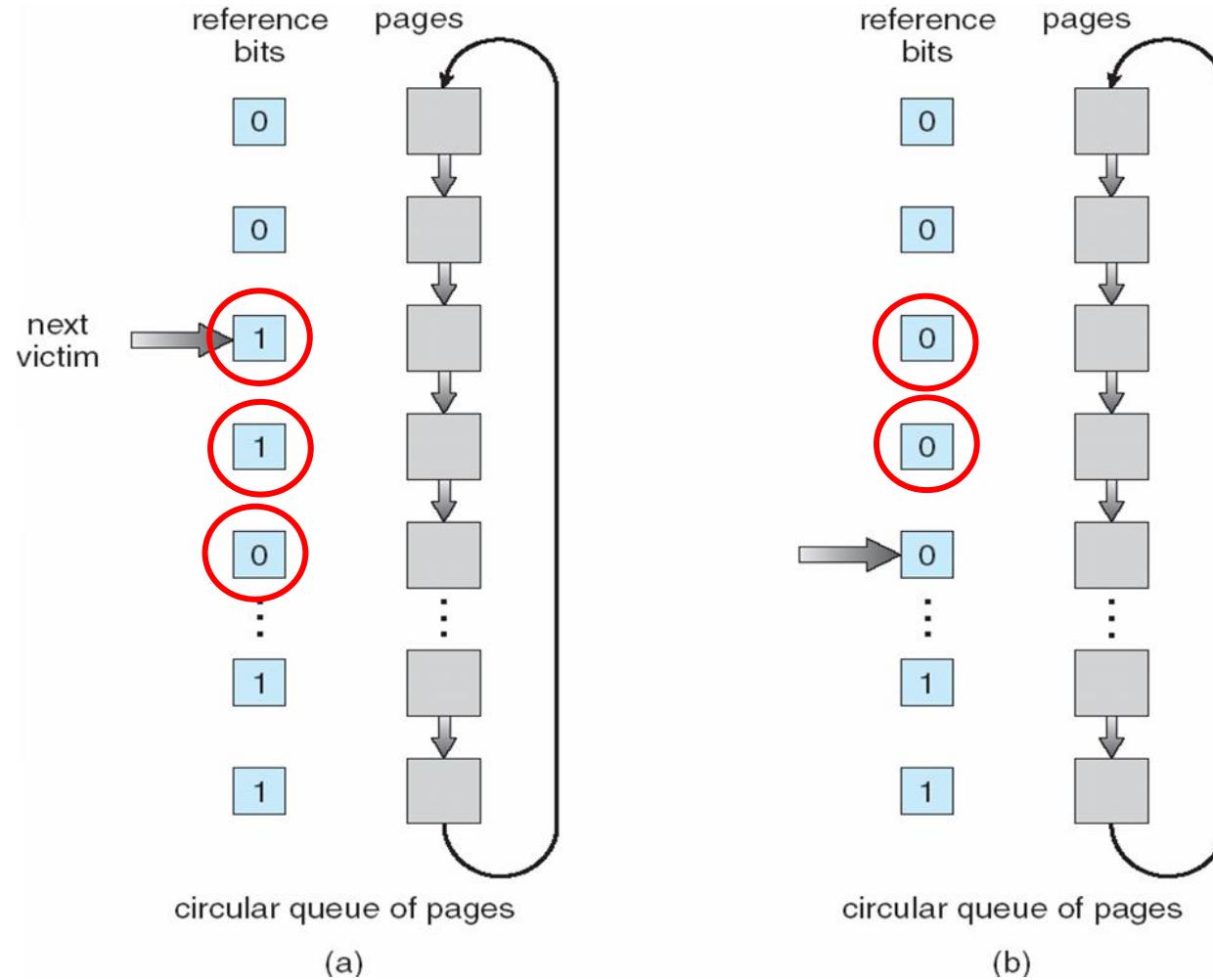
Need reference bit

Clock replacement

If page to be replaced (in clock order) has reference bit = 1 then:

- ▶ set reference bit 0
- ▶ leave page in memory
- ▶ replace next page (in clock order), subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



Counting Algorithms

Keep a **counter** of the number of references that have been made to each page

LFU Algorithm: replaces page with smallest count

MFU Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Allocation of Frames

Each process needs **minimum** number of pages

Example: IBM 370 – 6 pages to handle SS MOVE instruction:

instruction is 6 bytes, might span 2 pages

2 pages to handle from

2 pages to handle to

Two major allocation schemes

fixed allocation

priority allocation

Fixed Allocation

Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.

Proportional allocation – Allocate according to the size of process

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation

Use a proportional allocation scheme using **priorities** rather than size

If process P_i generates a page fault,

select for replacement one of its frames

select for replacement a frame from a process **with lower priority number**

Global vs. Local Allocation

Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another

Local replacement – each process selects from only its own set of allocated frames

Thrashing

If a process does not have “enough” pages, the page-fault rate is very high. This leads to:

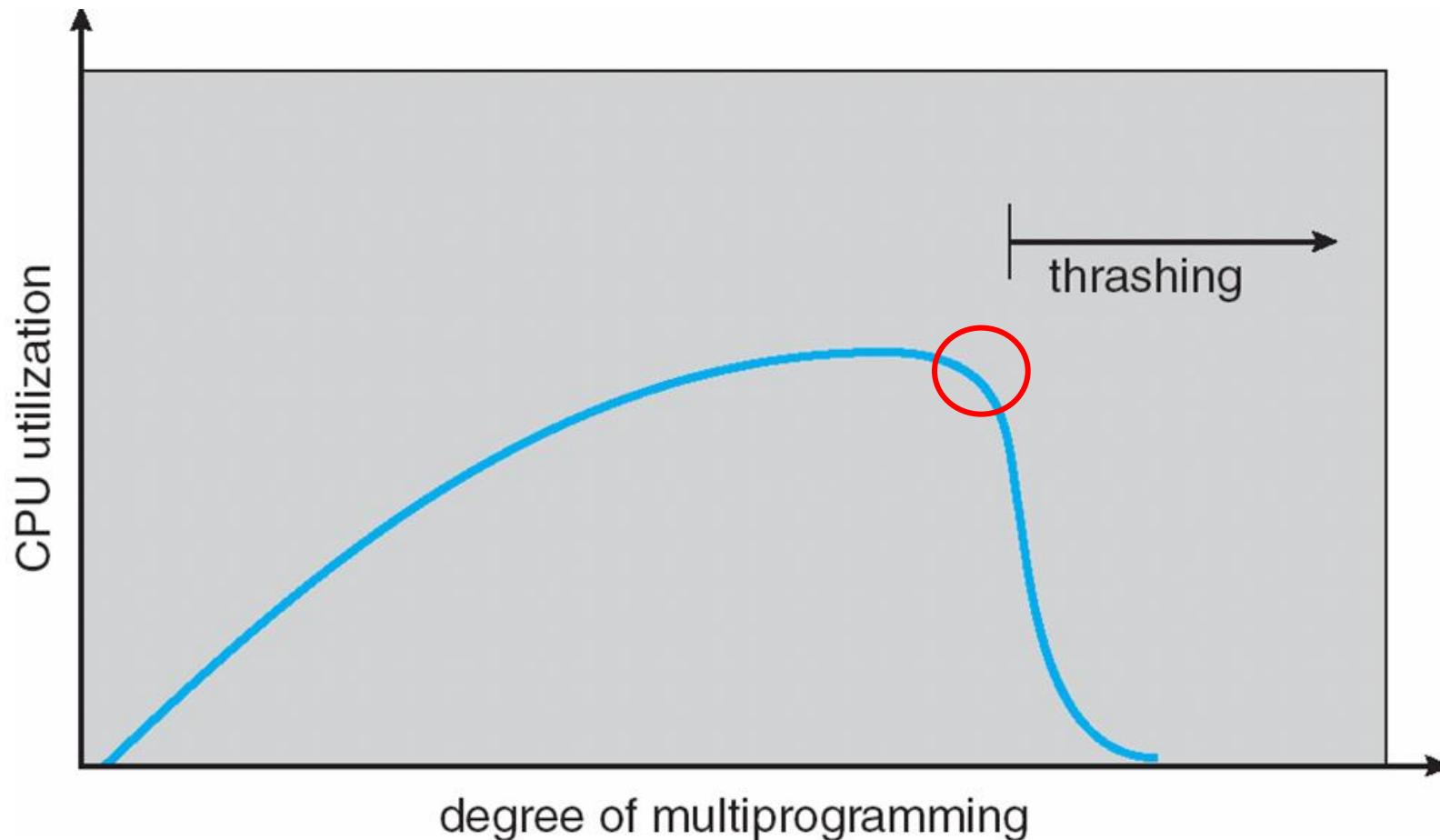
low CPU utilization

operating system thinks that it needs to increase the degree of multiprogramming

another process added to the system

Thrashing ≡ a process is busy swapping pages in and out

Thrashing (Cont.)



Demand Paging and Thrashing

Why does demand paging work?

Locality model

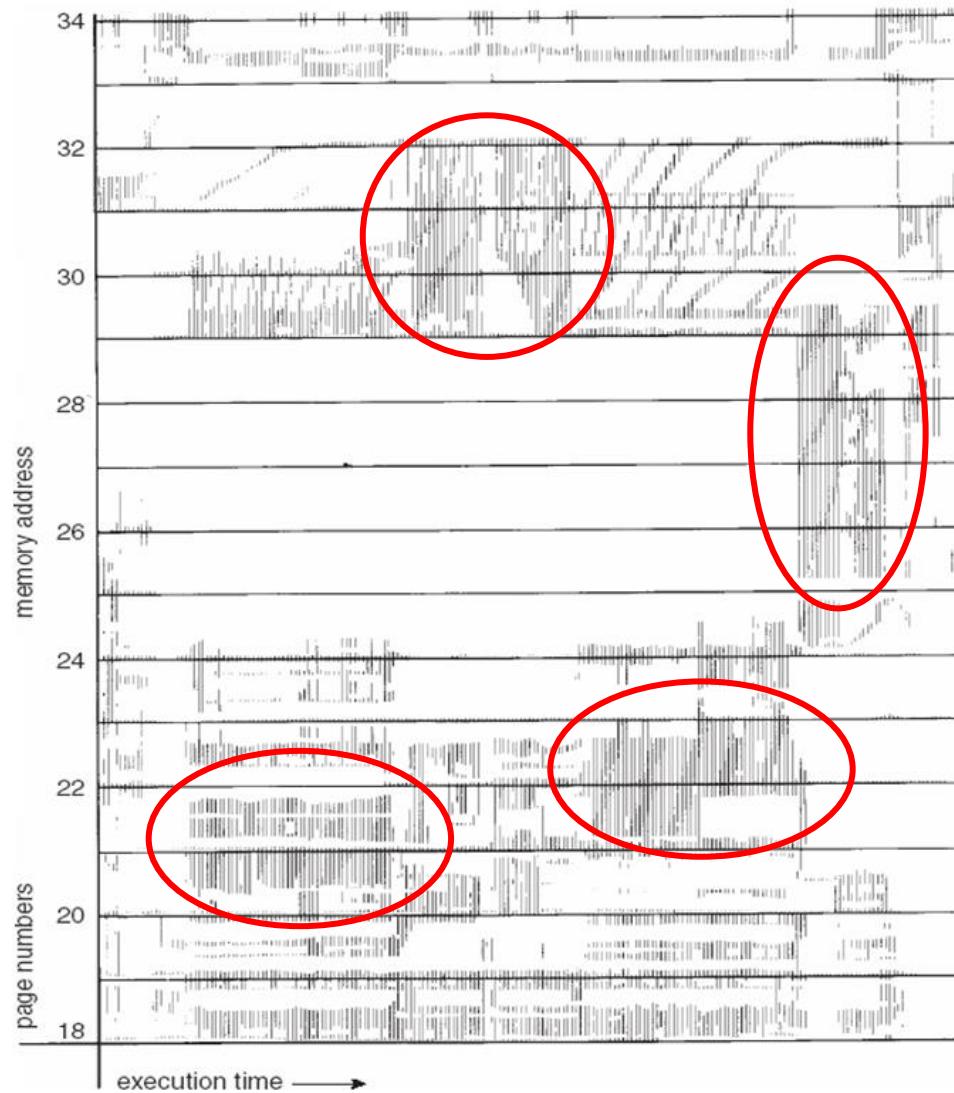
Process migrates from one locality to another

Localities may overlap

Why does thrashing occur?

Σ size of locality > total memory size

Locality In A Memory-Reference Pattern



Working-Set Model

$\Delta \equiv$ working-set window \equiv a fixed number of page references, Example: 10,000 instruction

WSS_i (working set of Process P_i) =
total number of pages referenced in the most recent Δ
(varies in time)

- if Δ too small will not encompass entire locality
- if Δ too large will encompass several localities
- if $\Delta = \infty \Rightarrow$ will encompass entire program

$D = \sum WSS_i \equiv$ total demand frames

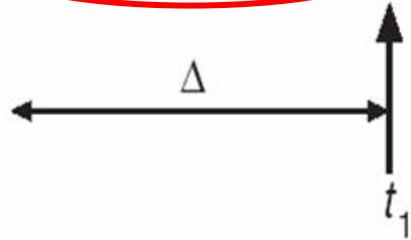
if $D > m \Rightarrow$ Thrashing

Policy if $D > m$, then suspend one of the processes

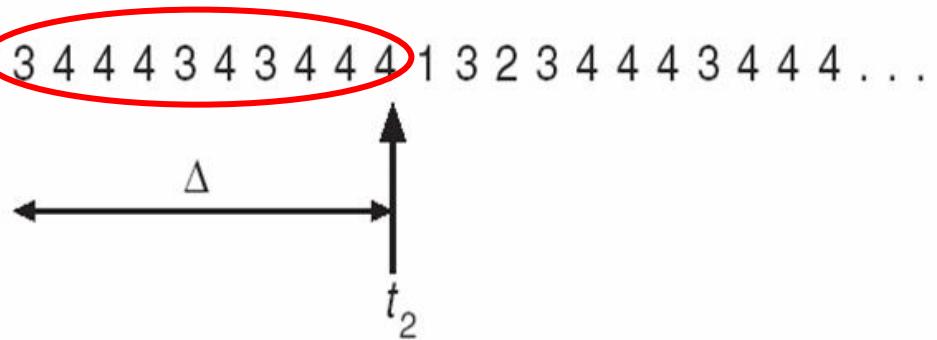
Working-set model

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



$$WS(t_1) = \{1, 2, 5, 6, 7\}$$



$$WS(t_2) = \{3, 4\}$$

Keeping Track of the Working Set

Approximate with interval timer + a reference bit

Example: $\Delta = 10,000$

Timer interrupts after every 5000 time units

Keep in memory 2 bits for each page

Whenever a timer interrupts copy and set the values of all reference bits to 0

If one of the bits in memory = 1 \Rightarrow page in working set

Why is this not completely accurate?

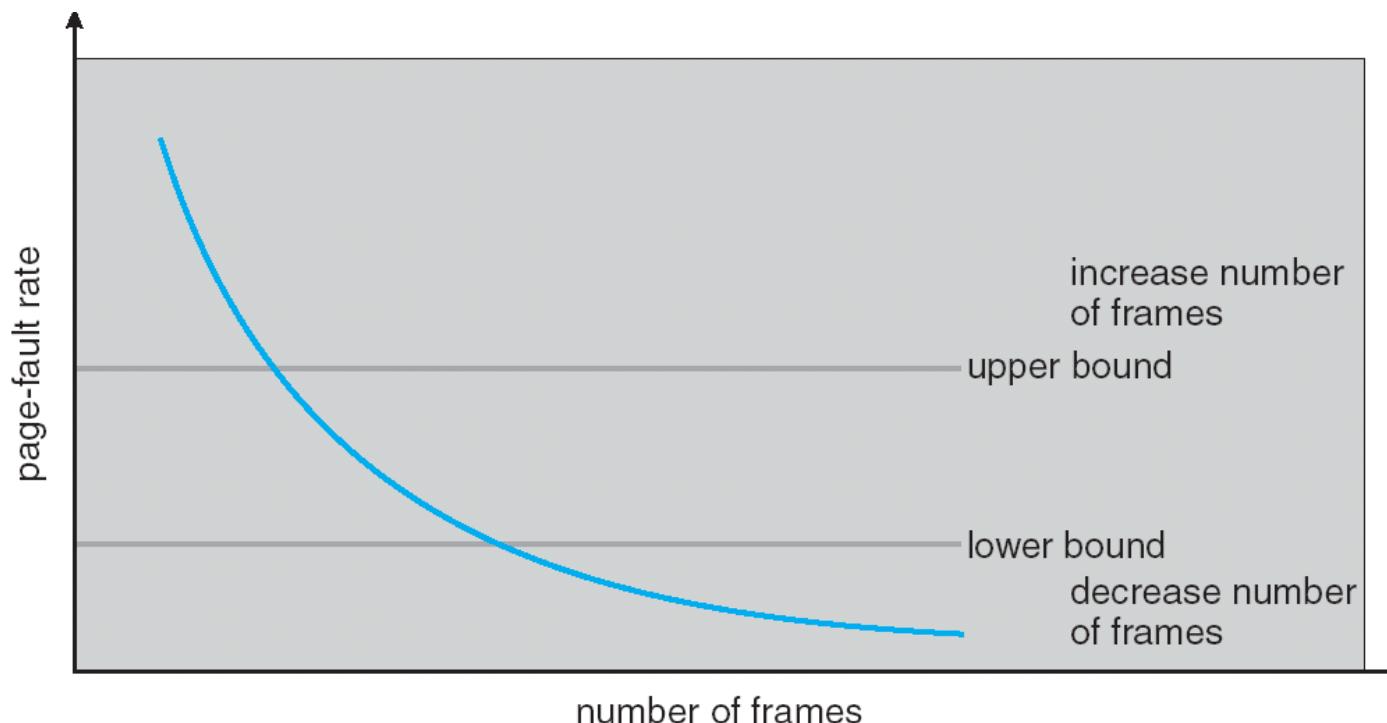
Improvement = 10 bits and interrupt every 1000 time units

Page-Fault Frequency Scheme

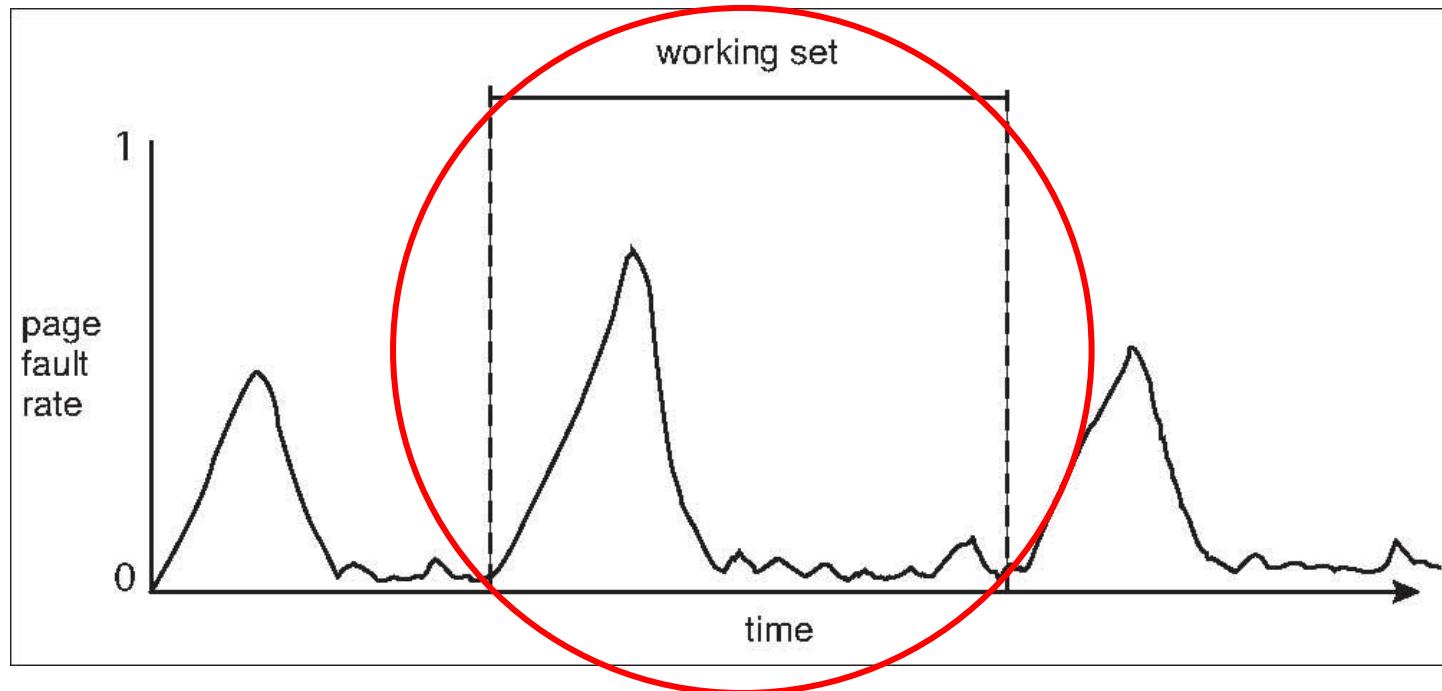
Establish “acceptable” page-fault rate

If actual rate too low, process loses frame

If actual rate too high, process gains frame



Working Sets and Page Fault Rates



Memory-Mapped Files

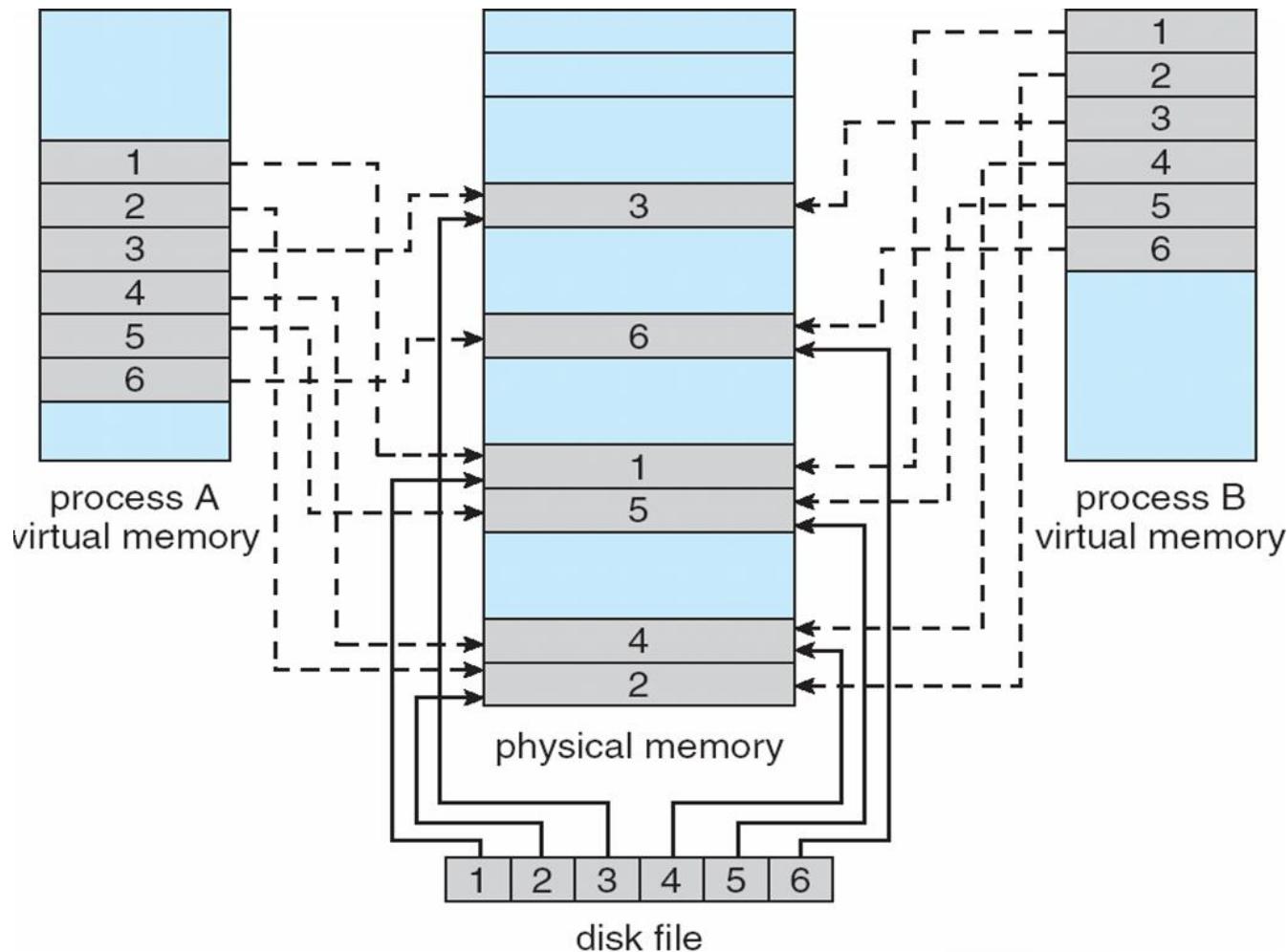
Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping a disk block to a page in memory**

A file is initially read using **demand paging**. A page-sized portion of the file is read from the file system into a physical page. **Subsequent reads/writes from/to the file are treated as ordinary memory accesses.**

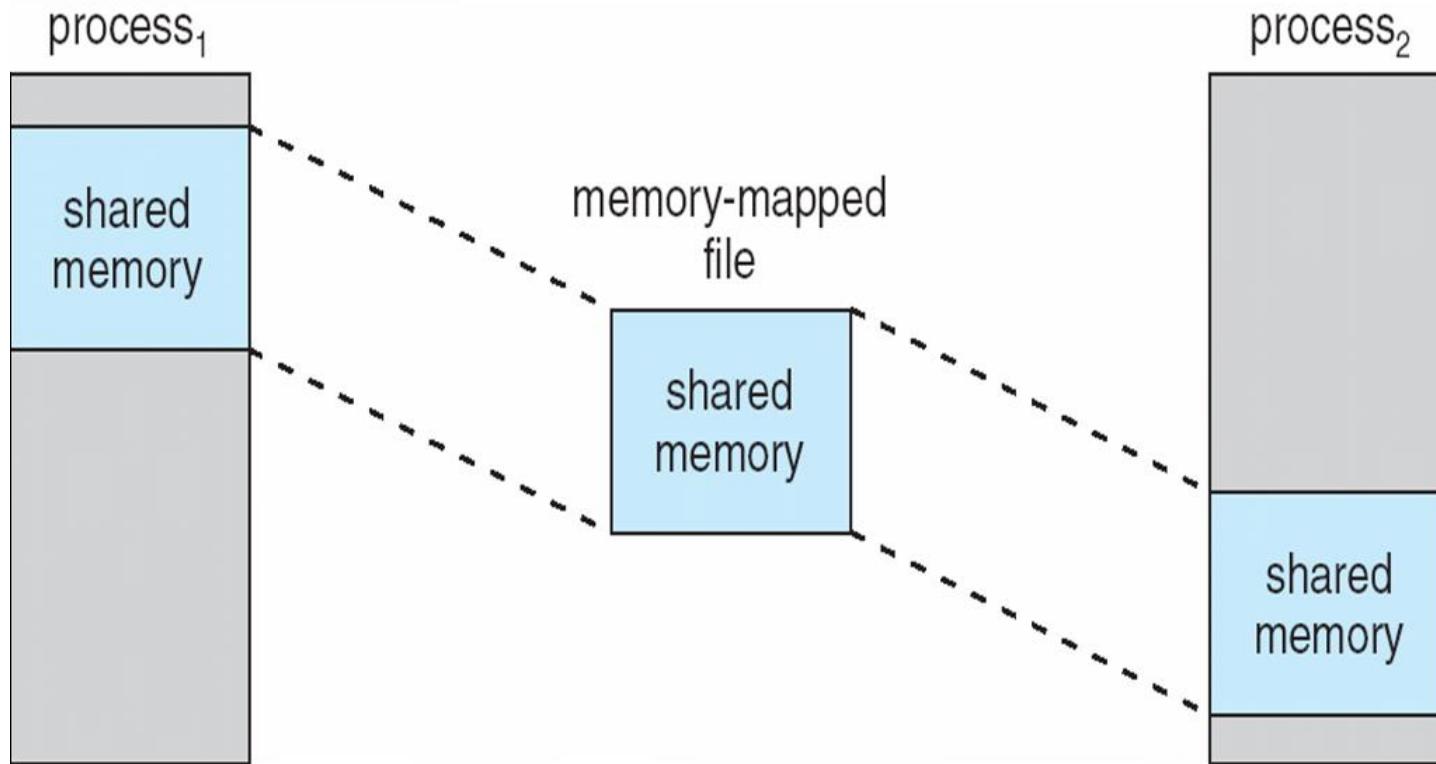
Simplifies file access by treating file I/O through memory rather than `read()` , `write()` system calls

Also allows several processes to map the same file allowing the pages in memory to be shared

Memory Mapped Files



Memory-Mapped Shared Memory in Windows



Allocating Kernel Memory

Treated differently from user mode memory (list of free..)

Often allocated from a **free-memory pool**

Kernel requests memory for structures of varying sizes, some of which are less than a page in size.

The kernel must use memory conservatively and attempt to minimize waste due to fragmentation.

Many OS do not subject kernel code or data to the paging system.

Some kernel memory needs to be contiguous due to certain hardware devices interact directly with physical memory – without the benefit of a virtual memory interface.

Two strategies: **Buddy System** and **Slab Allocation**

Buddy System

Allocates memory from fixed-size segment consisting of physically-contiguous pages

Memory is allocated from this segment using a power-of-2 allocator

Satisfies requests in units sized as power of 2

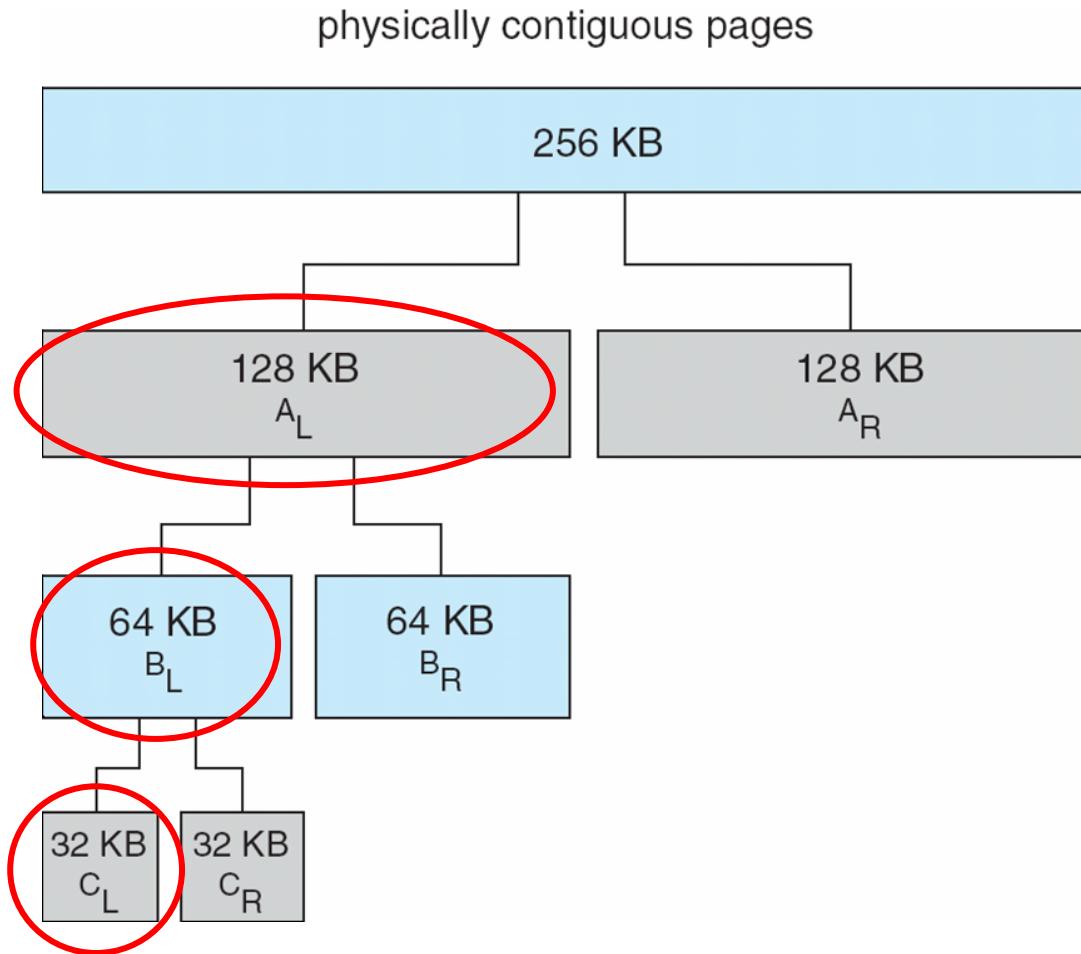
Request rounded up to next highest power of 2, for 11kB, it is satisfies with a 16-KB segment

When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2

- ▶ Continue until appropriate sized chunk available

Buddy System Allocator

Assume a size of a memory segment is initially 256KB and the kernel requests 21 KB of memory. C_L is the segment allocated to this request.



Buddy System Allocator

An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as **coalescing**.

When kernel releases CL,

$$C_L + C_R \rightarrow B_L,$$

$$B_L + B_R \rightarrow A_L,$$

$$A_L + A_R \rightarrow 256\text{KB segment.}$$

- Drawback: cause fragmentation within allocated segments.
- The next one is a memory allocation scheme where no space is lost due to fragmentation

Slab Allocation

Slab is one or more physically contiguous pages

Cache consists of one or more slabs

Single cache for each unique kernel data structure

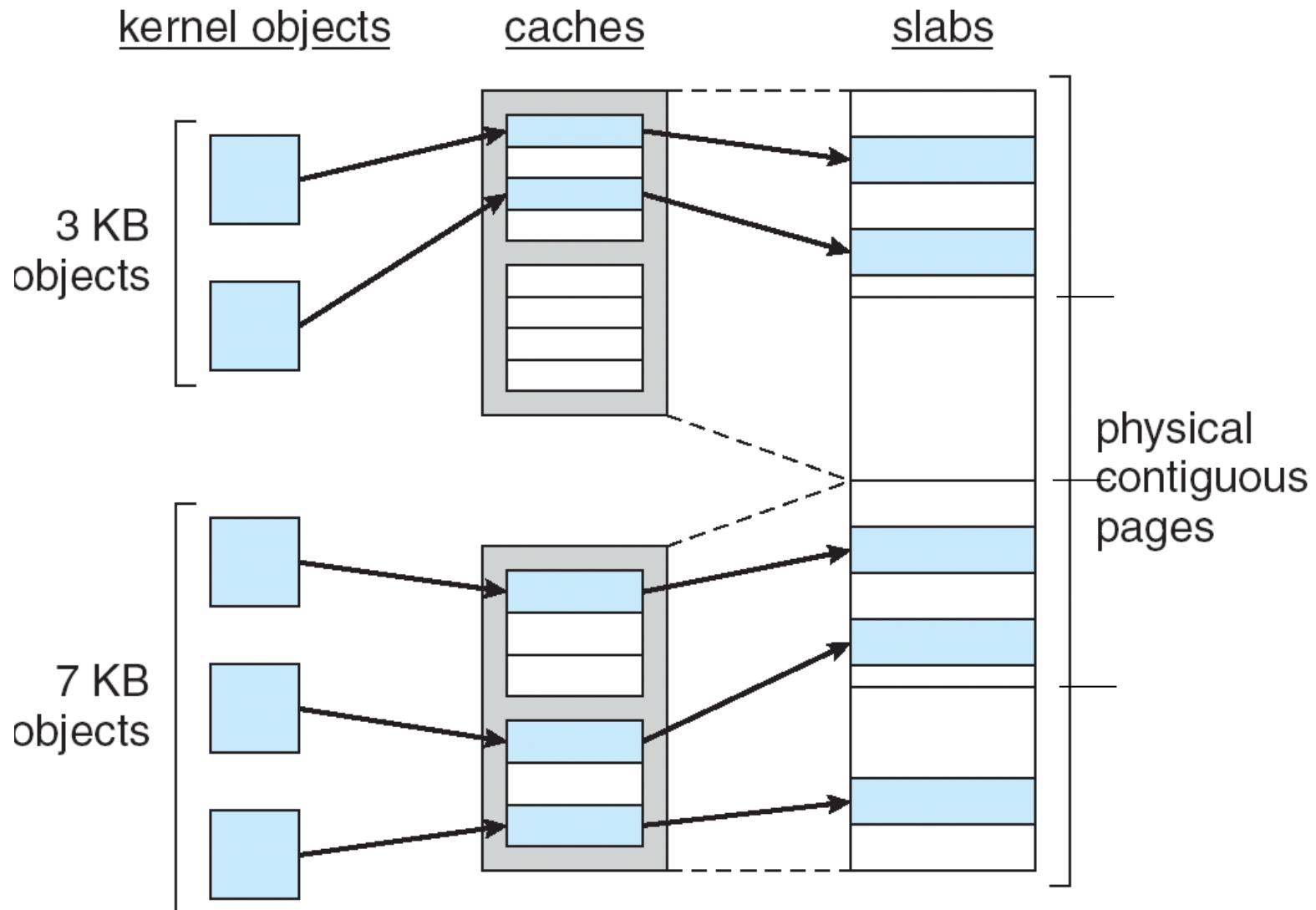
A separate cache for the data structure
representing process descriptor

A separate cache for file objects

A separate cache for semaphores

Each cache filled with **objects – instantiations of the kernel data structure** the cache represents.

Slab Allocation



Slab Allocation

The slab-allocation algorithm **uses caches to store kernel objects**

When a cache is created, a number of objects are allocated to the cache (initially marked as **free**).

The number of objects in the cache depends on the size of the associated slab. A 12-KB slab can store six 2-KB objects.

Slab Allocation

When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request.

The object assigned from the cache is marked as **used**

If slab is full of used objects, next object allocated from empty slab

If no empty slabs, new slab allocated

Two main benefits

No memory is wasted due to fragmentation.

Memory request can be satisfied quickly. (Objects are created in advance and thus can be quickly allocated from cache)

Other Issues -- Prepaging

Prepaging

To reduce the large number of page faults that occurs at process startup

Prepage all or some of the pages a process will need, before they are referenced

But if prepaged pages are unused, I/O and memory was wasted

Assume s pages are prepaged and α of the pages is used

- ▶ Is cost of $s * \alpha$ saved pages faults > or < than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages?
- ▶ α near zero \Rightarrow prepaging loses

Other Issues – Page Size

Page size selection must take into consideration:

fragmentation

table size

I/O overhead

locality

Other Issues – TLB Reach

TLB Reach - The amount of memory accessible from the TLB ([translation look-aside buffers, Chapter 8](#))

$$\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$$

Ideally, the working set of each process is stored in the TLB, otherwise there is a high degree of page faults

Increase the Page Size

This may lead to an increase in fragmentation as not all applications require a large page size

Provide Multiple Page Sizes

This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Other Issues – Program Structure

Program structure

```
Int[128,128] data;
```

Each row is stored in one page (need 128 pages to store)

Program 1

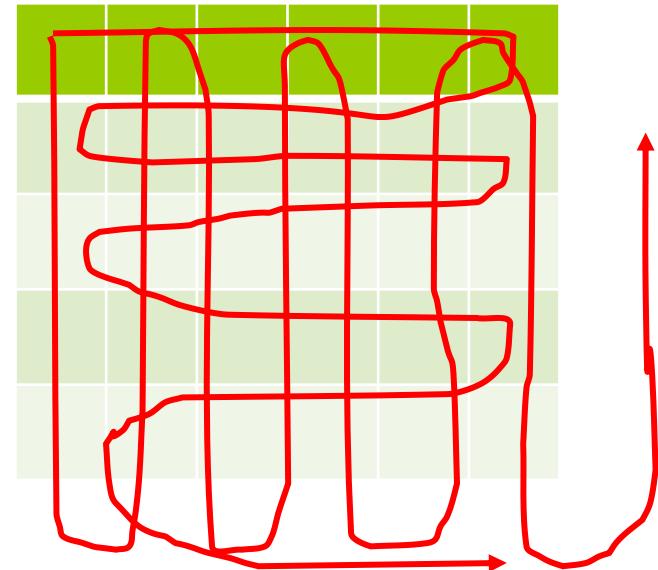
```
for (j = 0; j < 128; j++)  
    for (i = 0; i < 128; i++)  
        data[i,j] = 0;
```

128 x 128 = 16,384 page faults

Program 2

```
for (i = 0; i < 128; i++)  
    for (j = 0; j < 128; j++)  
        data[i,j] = 0;
```

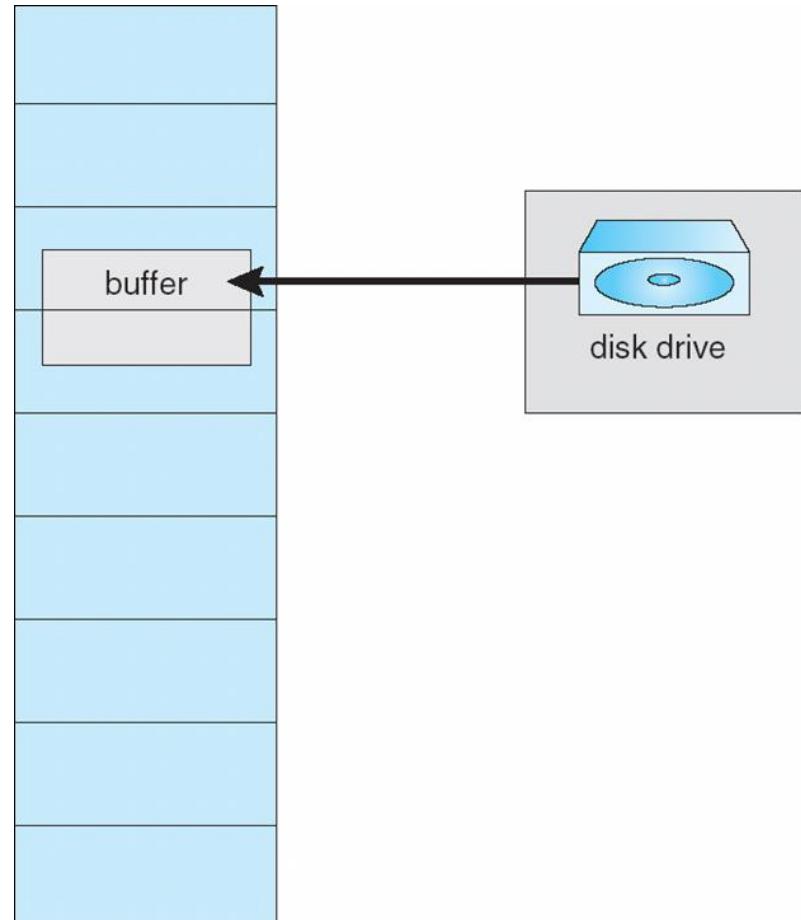
128 page faults



Other Issues – I/O interlock

I/O Interlock – Pages must sometimes be locked into memory

Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm



Operating System Examples

Windows XP

Solaris

Windows XP

Uses demand paging with clustering. Clustering brings in pages surrounding the faulting page

Processes are assigned working set minimum and working set maximum

Working set minimum is the minimum number of pages the process is guaranteed to have in memory

A process may be assigned as many pages up to its working set maximum

When the amount of free memory in the system falls below a threshold, automatic working set trimming is performed to restore the amount of free memory

Working set trimming removes pages from processes that have pages in excess of their working set minimum

Solaris

Maintains a list of free pages to assign faulting processes (threads)

Lotsfree – threshold parameter (amount of free memory, usually 1/64 of the physical memory) to begin **paging**

Desfree – threshold parameter to **increasing paging** (from 4 times to 100 times/sec)

Minfree – threshold parameter to begin **swapping** processes, thereby freeing all pages allocated to the swapped processes.

Paging is performed by **pageout** process

Pageout scans pages (four times per second) using modified clock algorithm (**two hands**)

Solaris

The front hand scans the pages and sets the ref bit to 0

The back hand checks and appends each page with ref bit still equals 0 to the free list.

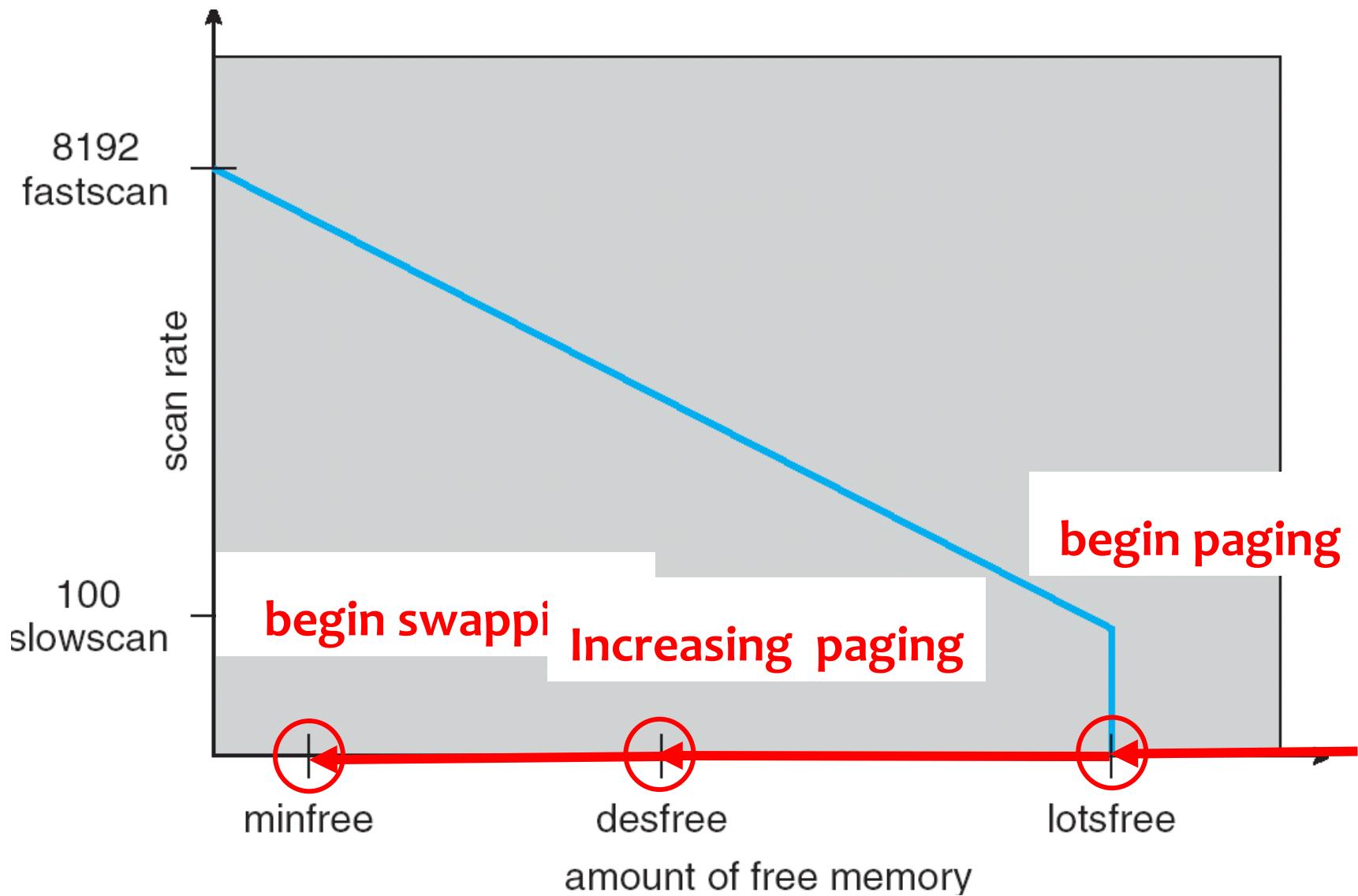
Handspread: the distance (in pages) between the two hands

Scanrate is the rate at which pages are scanned. This ranges from **slowscan** (100 pages/sec) to **fastscan** (up to 8192 pages/sec)

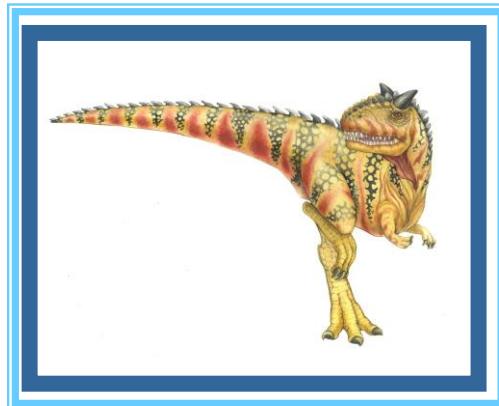
The amount of time between the two hands depends on scanrate and handspread. For scanrate = 100/sec, handspread = 1000 pages, we have 10 sec.

Pageout is called more frequently depending upon the amount of free memory available

Solaris 2 Page Scanner

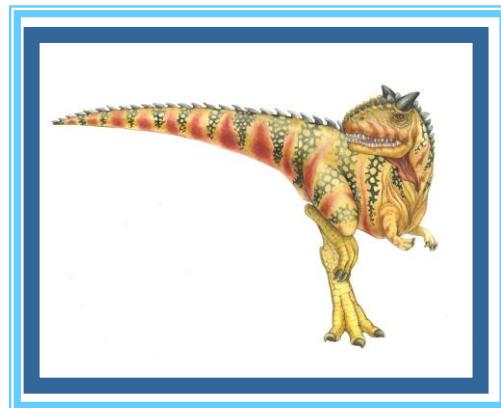


End of Chapter 9



Chapter 10

File-System



Chapter 10: File System

File Concept

Access Methods

Directory Structure

File-System Mounting

File Sharing

Protection

Objectives

To explain the **function of file systems**

To describe the **interfaces to file systems**

To discuss file-system **design tradeoffs**, including

access methods,

file sharing,

file locking, and

directory structures

To explore **file-system protection**

File Concept

Contiguous logical address space

Types:

Data

- ▶ numeric
- ▶ character
- ▶ binary

Program

File Structure

None - sequence of words, bytes

Simple record structure

Lines

Fixed length

Variable length

Complex Structures

Formatted document

Relocatable load file

Can simulate last two methods with first method by inserting appropriate **control characters (CR, LF)**

Who decides:

Operating system

Program

File Attributes

Name – only information kept in human-readable form

Identifier – unique tag (number) identifies file within file system

Type – needed for systems that support different types

Location – pointer to file location on device

Size – current file size

Protection – controls who can do reading, writing, executing

Time, date, and user identification – data for protection, security, and usage monitoring

Information about files are kept in the **directory structure**, which is maintained on the disk

File Operations

File is an abstract data type

Create

Write

Read

Reposition within file

Delete

Truncate

Open(F_i) – search the directory structure on disk for entry F_i , and move the content of entry to memory

Close (F_i) – move the content of entry F_i in memory to directory structure on disk

Open Files

Several pieces of data are needed to manage open files:

File pointer: pointer to last read/write location, per process that has the file open

File-open count: counter of number of times a file is open – to allow removal of data from open-file table when last process closes it

Disk location of the file: cache of data access information

Access rights: per-process access mode information

Open File Locking

Provided by some operating systems and file systems

Shared Lock: several processes can acquire the lock concurrently (like a reader lock)

Exclusive Lock: Only one process at a time can acquire such a lock (like a writer lock)

Mandatory or advisory file locking mechanisms:

Mandatory – Once a process acquires an exclusive lock, the OS will prevent any other process from accessing the locked file. (Windows)

Advisory – The OS will not prevent a process from acquiring access to a locked file. Rather, the process must be written so that it manually acquiring the lock before accessing the file. (UNIX)

File Locking Example – Java API

```
import java.io.*;
import java.nio.channels.*;
public class LockingExample {
    public static final boolean EXCLUSIVE = false;
    public static final boolean SHARED = true;
    public static void main(String args[]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;
        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");
            // get the channel for the file
            FileChannel ch = raf.getChannel();
            // this locks the first half of the file - exclusive
            exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);
            /**
             * Now modify the data ...
             */
            // release the lock
            exclusiveLock.release();
        }
    }
}
```

File Locking Example – Java API (cont)

```
// this locks the second half of the file - shared  
sharedLock = ch.lock(raf.length()/2+1, raf.length(), SHARED);  
  
/** Now read the data ... */  
  
// release the lock  
sharedLock.release();  
}  
catch (java.io.IOException ioe) {  
    System.err.println(ioe);  
}  
finally {  
    if (exclusiveLock != null)  
        exclusiveLock.release();  
    if (sharedLock != null)  
        sharedLock.release();  
}  
}  
}
```

File Types – Name, Extension

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Access Methods

Sequential Access

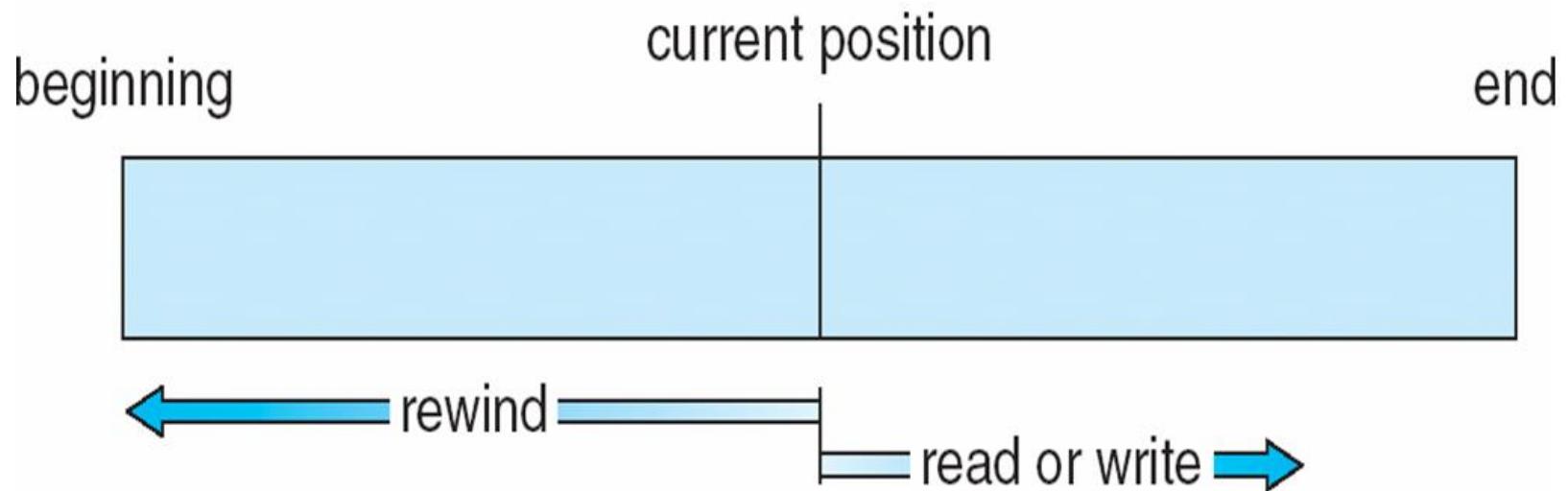
read next
write next
reset

Direct Access

read n
write n
position to n
read next
write next
rewrite n

n = relative block number

Sequential-access File



Simulation of Sequential Access on Direct-access File

sequential access	implementation for direct access
<i>reset</i>	$cp = 0;$
<i>read next</i>	$read cp;$ $cp = cp + 1;$
<i>write next</i>	$write cp;$ $cp = cp + 1;$

Example of Index and Relative Files

logical record
last name number

Adams	
Arthur	
Asher	
⋮	
Smith	

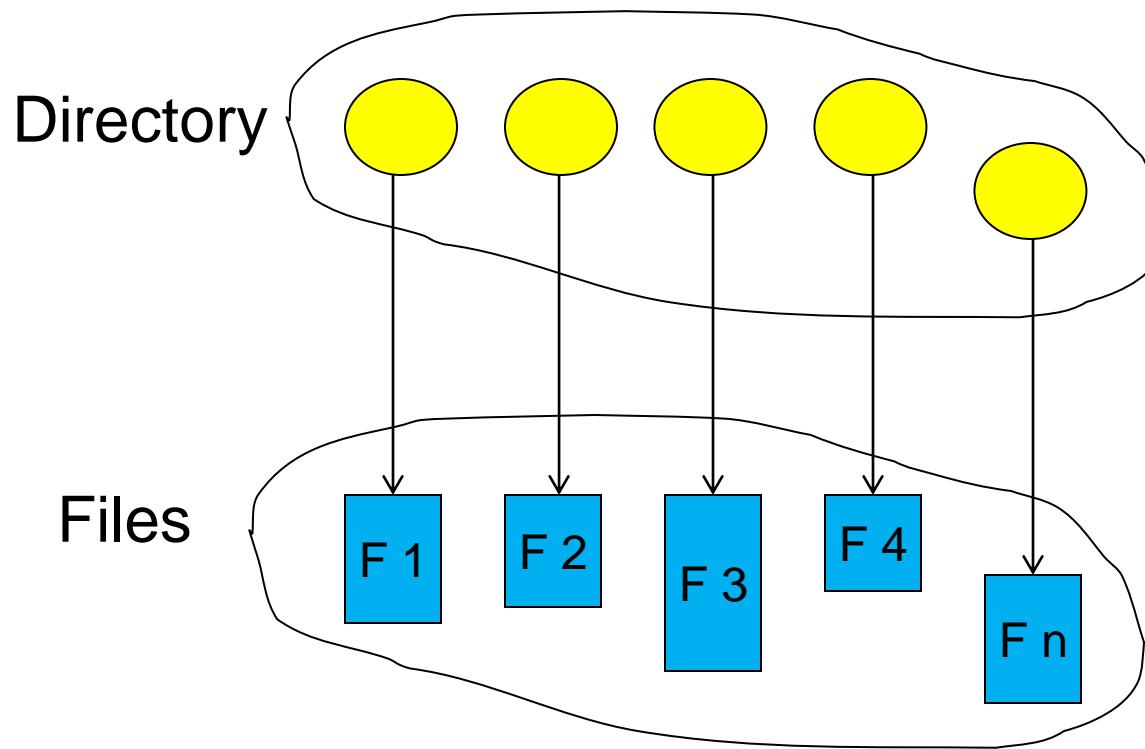
index file

smith, john	social-security	age

relative file

Directory Structure

A collection of nodes containing information about all files



Both the directory structure and the files reside on disk
Backups of these two structures are kept on tapes

Disk Structure

Disk can be subdivided into **partitions**

Disks or partitions can be **RAID** protected against failure

Disk or partition can be used **raw** – without a file system,
or **formatted** with a file system

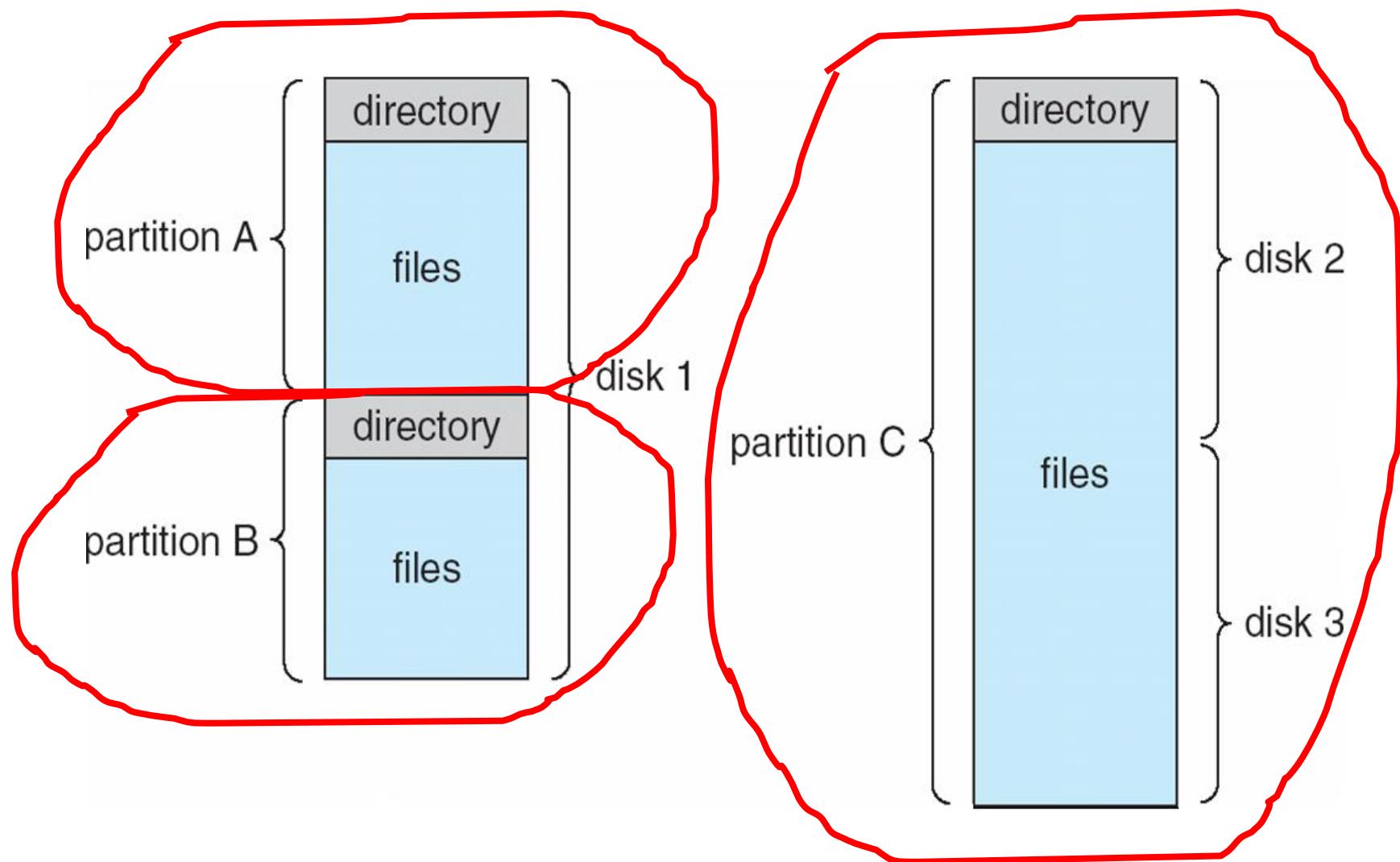
Partitions also known as minidisks, slices

Entity containing file system known as a **volume**

Each volume containing file system also tracks that file
system's info in **device directory** or **volume table of
contents**

As well as **general-purpose file systems** there are many
special-purpose file systems, frequently all within the
same operating system or computer (Solaris)

A Typical File-system Organization



Operations Performed on Directory

Search for a file

Create a file

Delete a file

List a directory

Rename a file

Traverse the file system

Organize the Directory (Logically) to Obtain

Efficiency – locating a file quickly

Naming – convenient to users

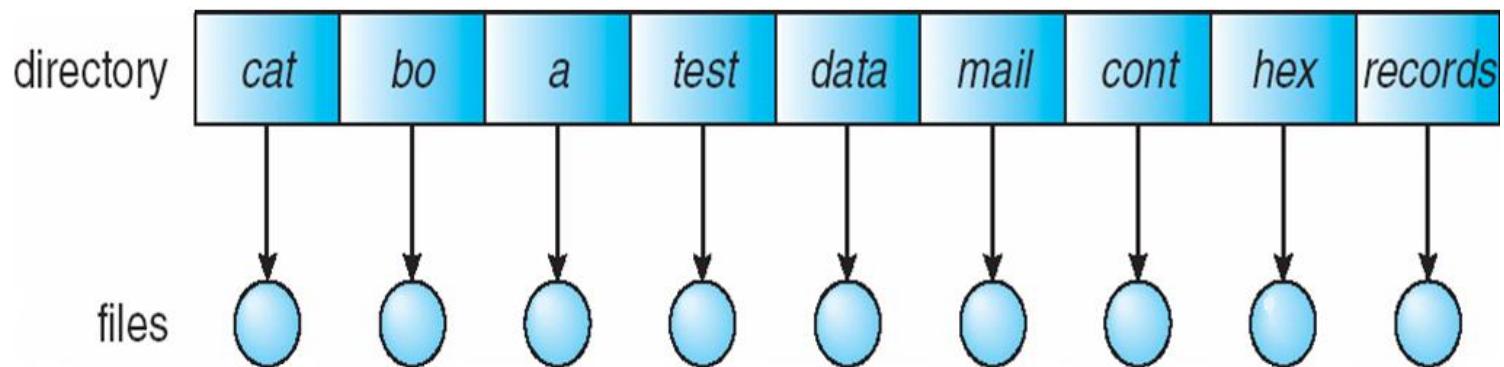
Two users can have same name for different files

The same file can have several different names

Grouping – logical grouping of files by properties,
(e.g., all Java programs, all games, ...)

Single-Level Directory

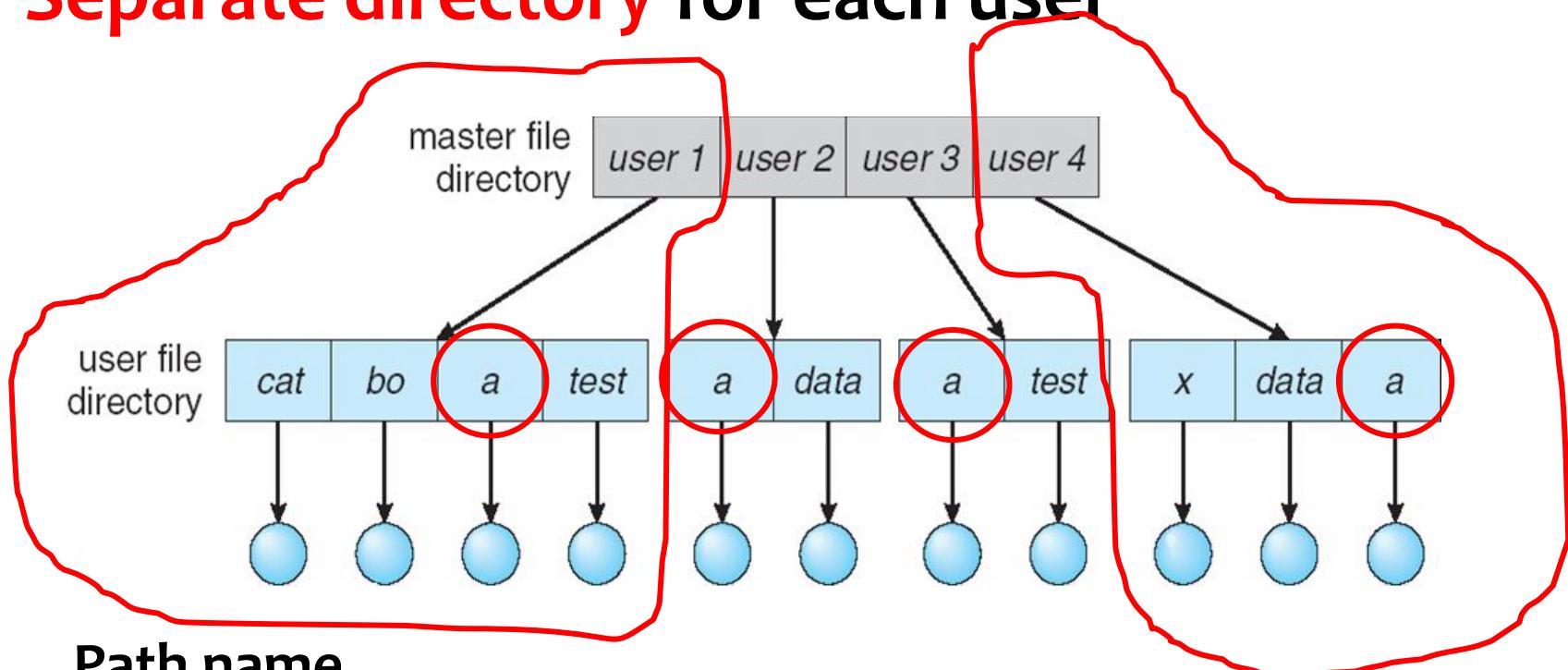
A **single directory** for all users



- Naming problem
- Grouping problem

Two-Level Directory

Separate directory for each user



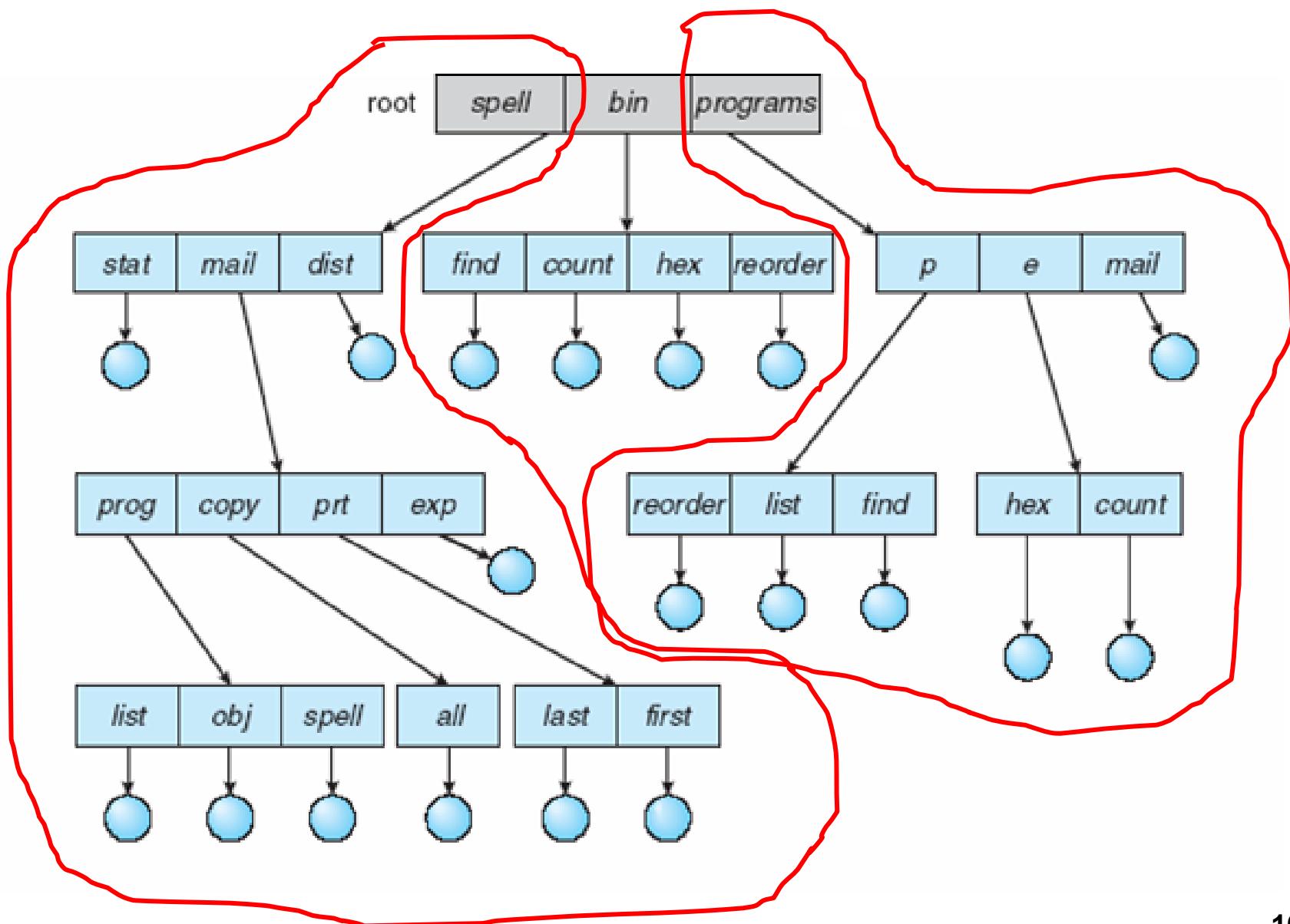
Path name

Can have the same file name for different user

Efficient searching

No grouping capability

Tree-Structured Directories



Tree-Structured Directories (Cont)

Efficient searching

Grouping Capability

Current directory (**working directory**)

`cd /spell/mail/prog`

`type list`

Tree-Structured Directories (Cont)

Absolute or relative path name

Creating a new file is done in **current directory**

Delete a file

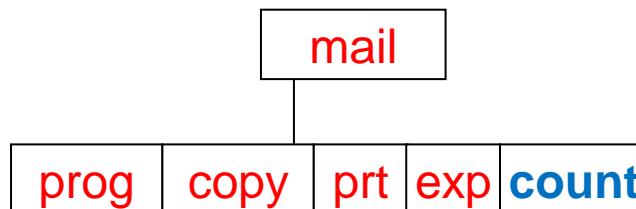
rm <file-name>

Creating a new subdirectory is done in current directory

mkdir <dir-name>

Example: if in current directory **/mail**

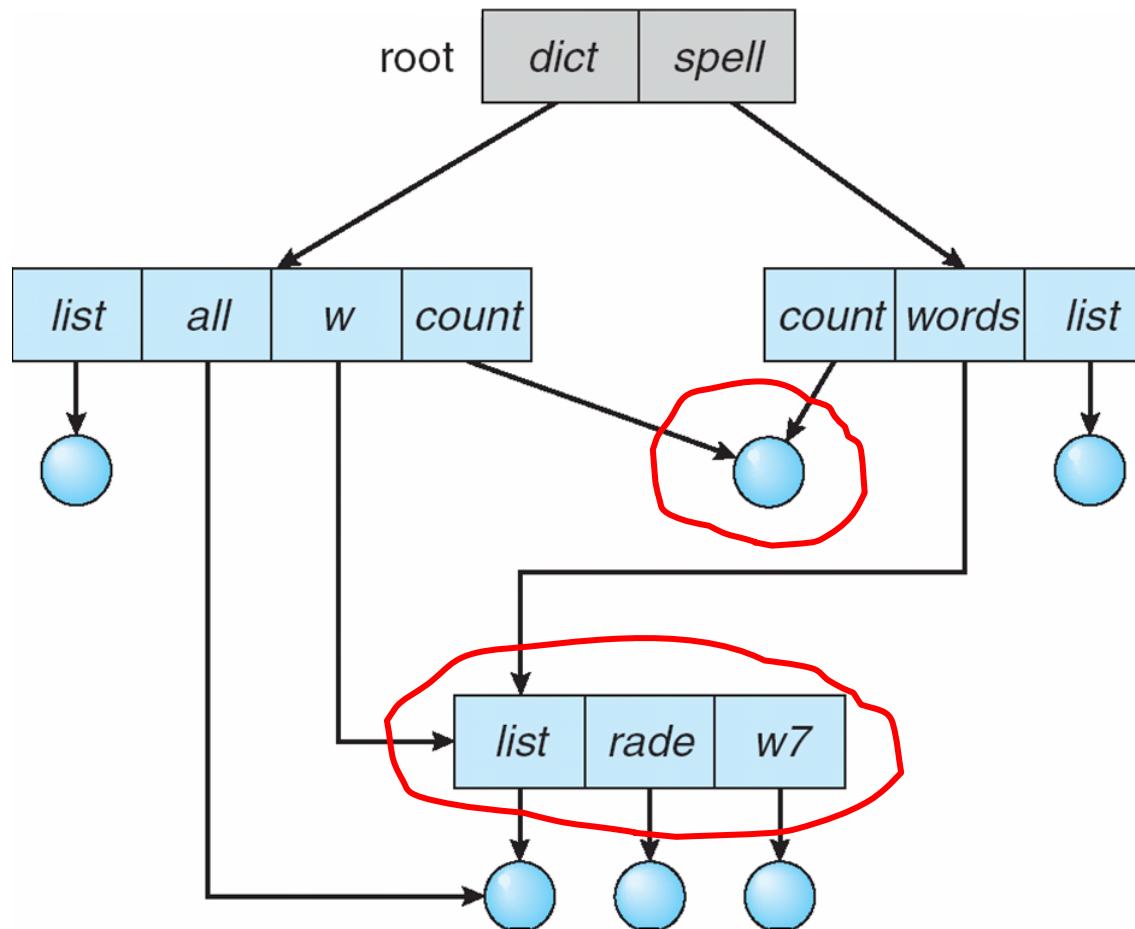
mkdir count



Deleting “mail” \Rightarrow deleting the entire subtree rooted by “mail”

Acyclic-Graph Directories

Have **shared subdirectories and files**, for joined project, for example



Acyclic-Graph Directories (Cont.)

Allows directories **to share subdirectories and files**. The same file or subdirectory may be in two different directories

Shared files and subdirectories can be implemented in several ways.

Create a new directory entry - Link

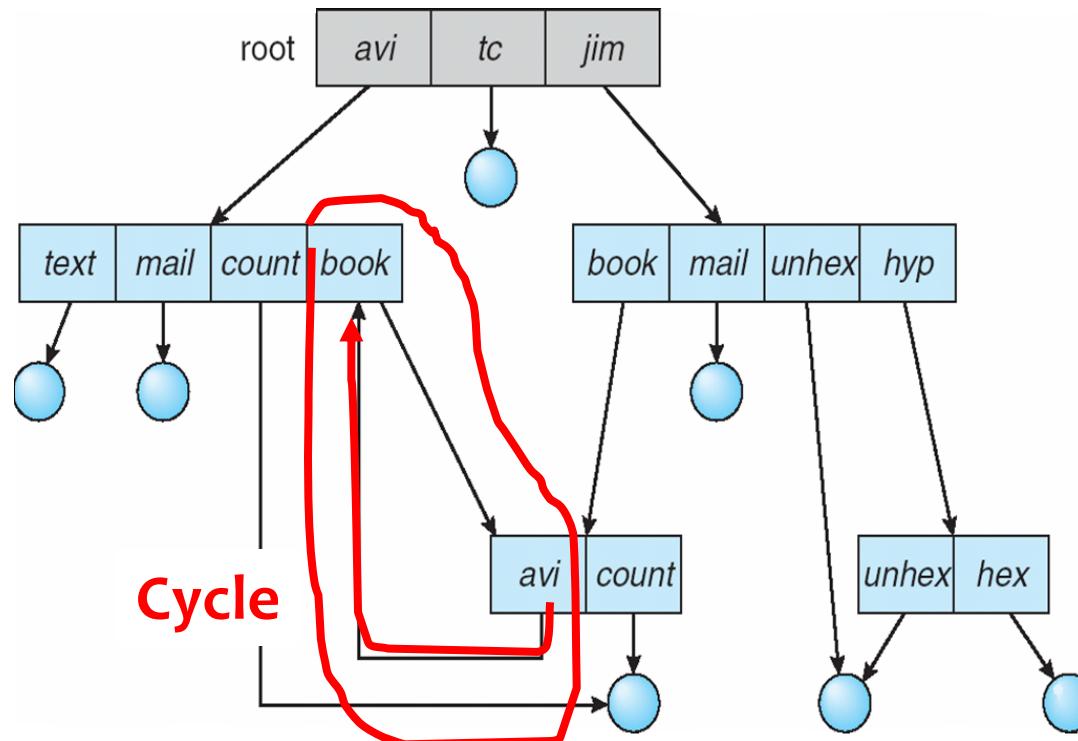
Link – a pointer to another file or subdirectory. A link may be implemented as an absolute or a relative **path name**.

Resolve the link – using that path name to locate the real file. Links are easily identified by their format in the directory entry and are effectively indirect pointers.

General Graph Directory

A serious problem with using acyclic-graph structure is ensuring that there is no cycles.

However, when we add links, the tree structure is destroyed, resulting in a simple graph structure.



General Graph Directory (Cont.)

If cycles are allowed to exist in the directory

An **infinite loop** continually searching through the cycle

When a file can be **deleted** ?

- ▶ A **Garbage collection scheme** is used to determine when the last reference has been deleted and the disk space can be reallocated.

Every time a new link is added use a **cycle detection algorithm** to determine whether it is OK

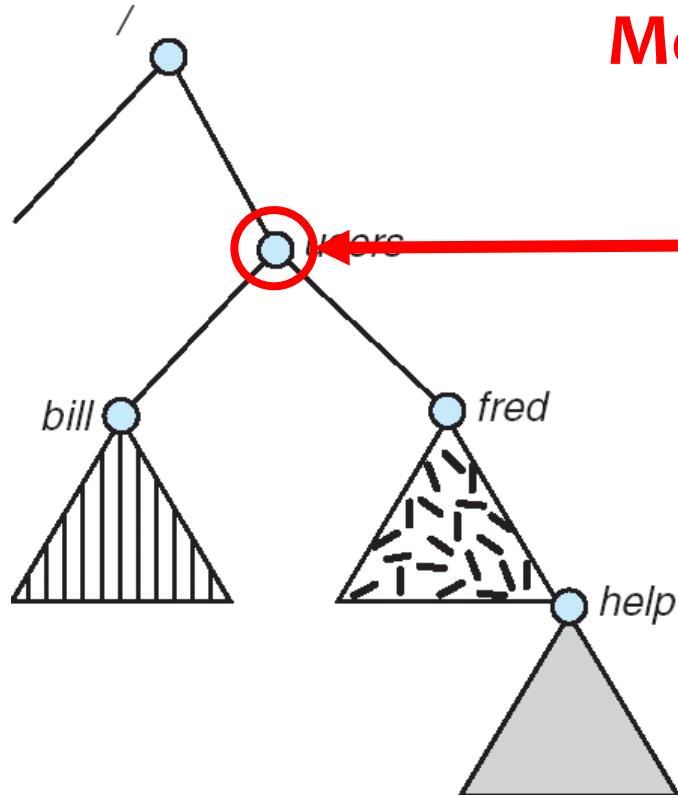
File System Mounting

A file system must be **mounted** before it can be accessed

A unmounted file system is mounted at a **mount point**

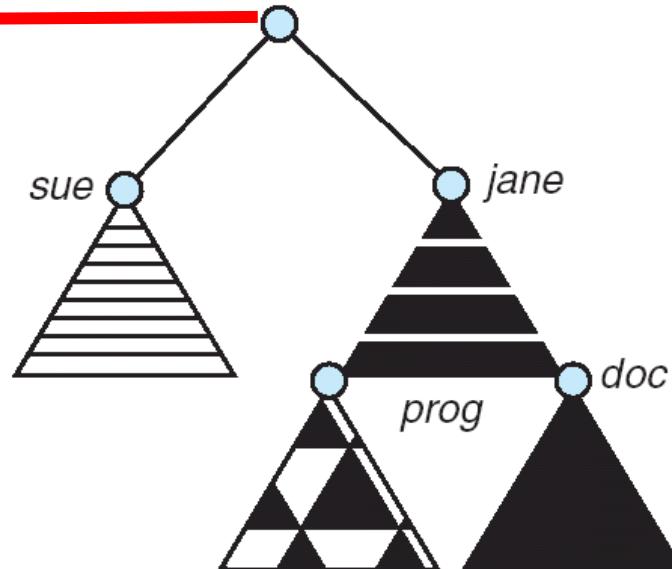
File System Mounting

Mounting the volume residing
on */device/dsk* over */users*



(a)

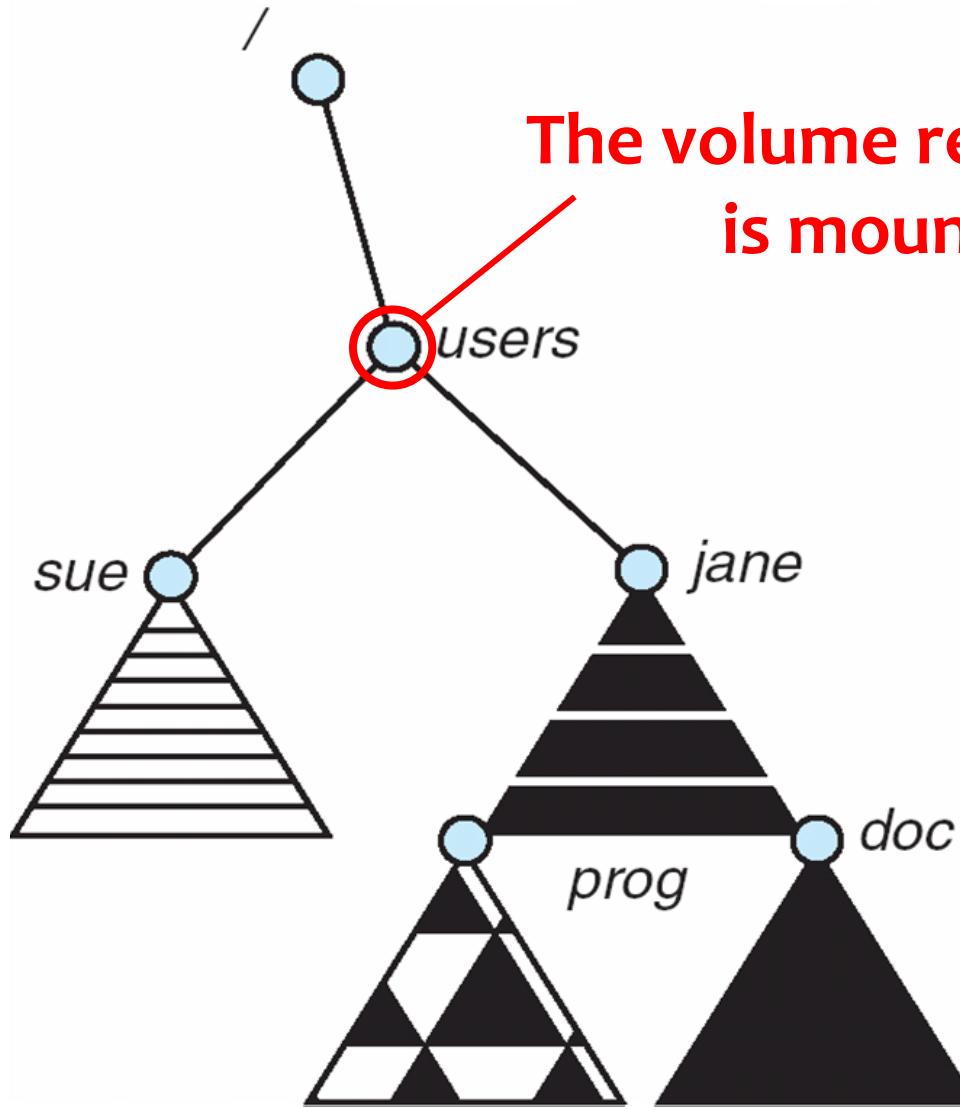
(a) Existing.



(b)

(b) Unmounted Partition

Mount Point



The volume residing on */device/dsk*
is mounted over */users*

File Sharing

Sharing of files on multi-user systems is desirable

Sharing may be done through a **protection scheme**

On distributed systems, files may be shared across a network

Network File System (NFS) is a common distributed file-sharing method

File Sharing – Multiple Users

User IDs identify users, allowing permissions and protections to be per-user

Group IDs allow users to be in groups, permitting group access rights

Uses networking to allow file system access between systems

Manually via programs like FTP

Automatically, seamlessly using distributed file systems

Semi automatically via the WWW

File Sharing – Remote File Systems

Client-server model allows **clients to mount remote file systems from servers**

Server can serve multiple clients

Client and user-on-client **identification** is insecure or complicated

NFS is standard UNIX client-server file sharing protocol

CIFS (Common Internet File System) is standard Windows protocol

Standard OS file calls are translated into remote calls

Distributed Information Systems (distributed naming services) such as LDAP (lightweight directory access protocol), DNS, NIS, Active Directory (Windows XP and Windows 2000) implement **unified access** to information needed for remote computing

File Sharing – Failure Modes

Remote file systems add **new failure modes**, due to network failure, server failure

Recovery from failure can involve **state information** about status of each remote request

Stateless protocols such as NFS include all information in each request, allowing easy recovery but less security

File Sharing – Consistency Semantics

Consistency semantics specify how multiple users are to access a shared file simultaneously

Similar to Ch 6 **process synchronization algorithms**

- ▶ Tend to be less complex due to disk I/O and network latency (for remote file systems)

Andrew File System (AFS, Chapter 17) implemented complex remote file sharing semantics

Unix file system (UFS, Chapter 17) implements:

- ▶ Writes to an open file visible immediately to other users of the same open file
- ▶ Sharing **file pointer** to allow multiple users to read and write concurrently

AFS has **session semantics**

- ▶ Writes only visible to sessions starting after the file is closed

Protection

File owner/creator should be able to control:

what can be done

by whom

Types of access

Read

Write

Execute

Append

Delete

List

Access Lists and Groups

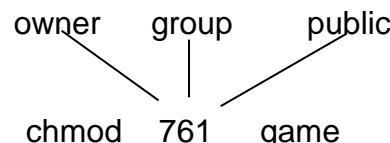
Mode of access: read, write, execute

Three classes of users

		RWX
a) owner access	7	\Rightarrow 1 1 1 RWX
b) group access	6	\Rightarrow 1 1 0 RWX
c) public access	1	\Rightarrow 0 0 1

Ask manager to create a group (unique name), say G, and add some users to the group.

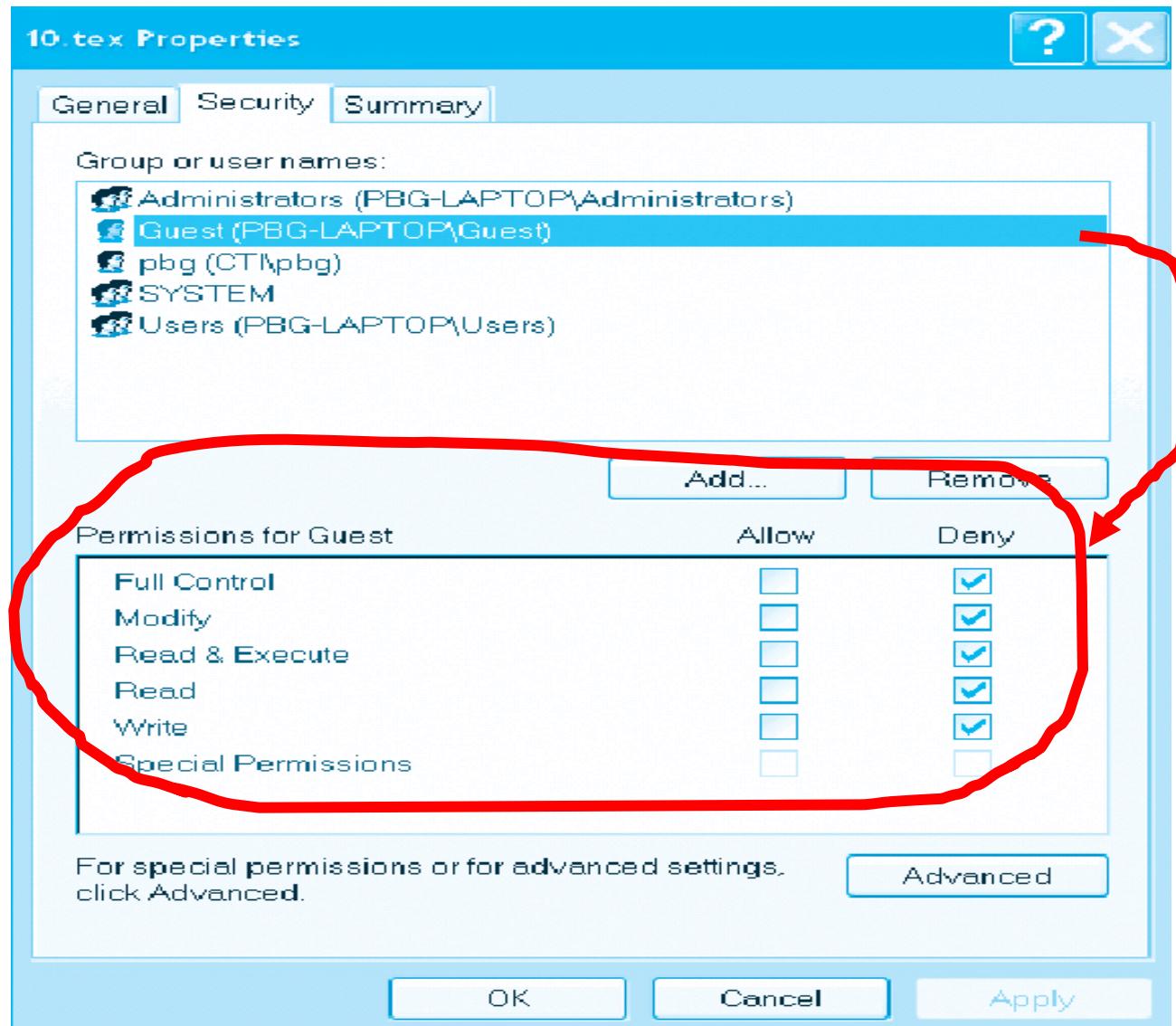
For a particular file (say *game*) or subdirectory, define an appropriate access.



Attach a group to a file

chgrp G game

Windows XP Access-control List Management



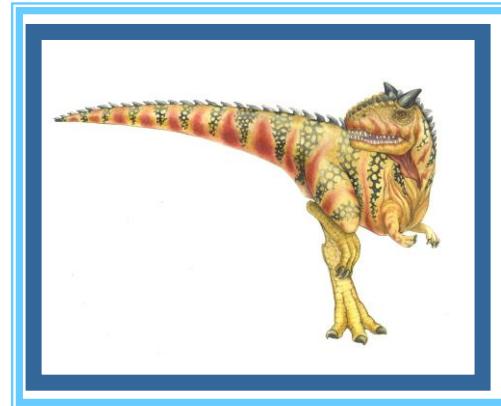
A Sample UNIX Directory Listing

subdirectory						
-rw-rw-r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx-----	5	pbg	staff	512	Jul 8 09:33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1	pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4	pbg	faculty	512	Jul 31 10:31	lib/
drwx-----	3	pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3	pbg	staff	512	Jul 8 09:35	test/

The number of links to the file

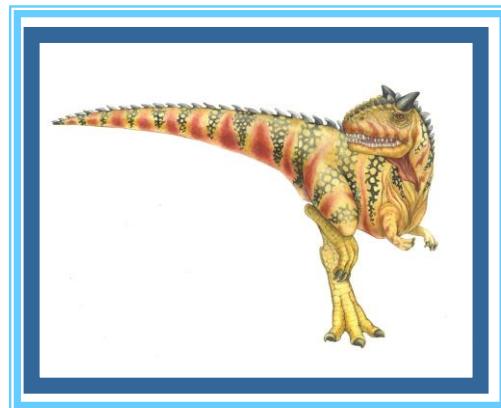
Owner, group Owner Group's name

End of Chapter 10



Chapter 11

Implementing File-Systems



Chapter 11: Implementing File Systems

File-System Structure

File-System Implementation

Directory Implementation

Allocation Methods

Free-Space Management

Efficiency and Performance

Recovery

Log-Structured File Systems

NFS

Example: WAFL File System

Objectives

To describe the details of **implementing local file systems and directory structures**

To describe the implementation of **remote file systems**

To discuss **block allocation and free-block algorithms and trade-offs**

File-System Structure

File structure

Logical storage unit

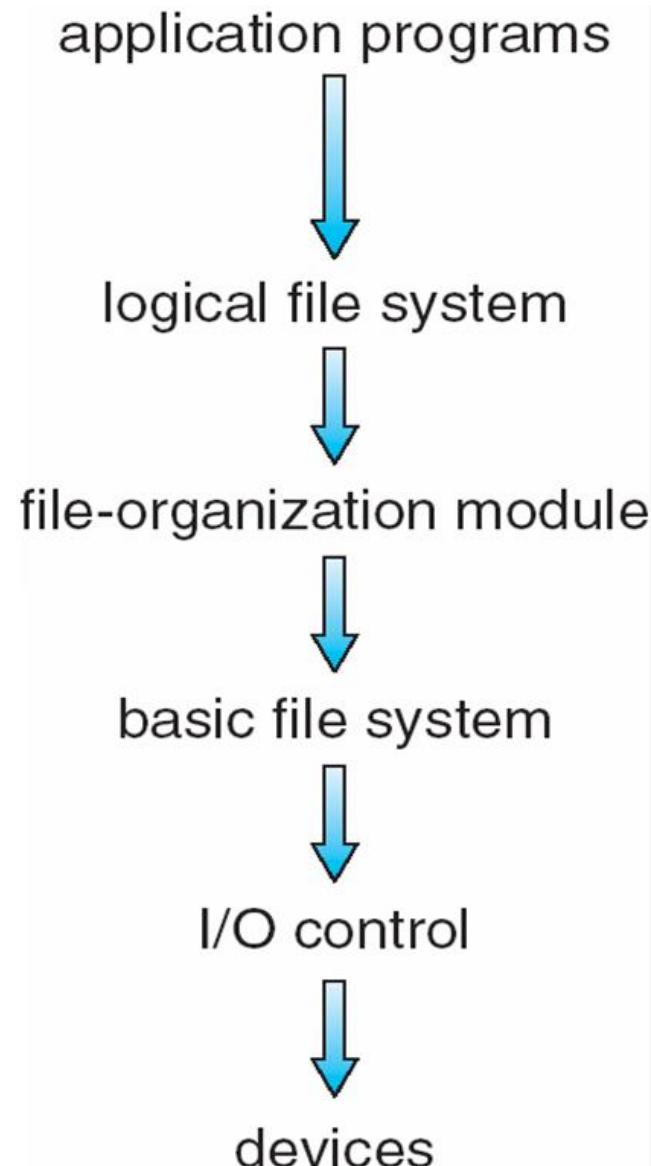
Collection of related information

File system resides on secondary storage (disks**)**

File system organized into **layers**

File control block – storage structure consisting of information about a file

Layered File System



A Typical File Control Block

file permissions

file dates (create, access, write)

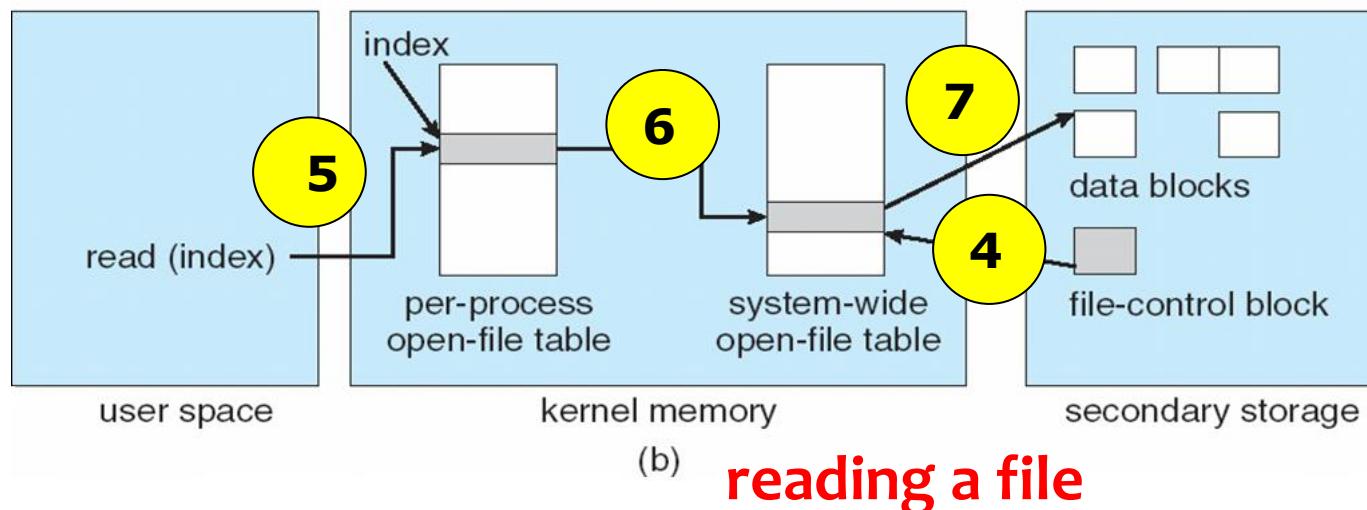
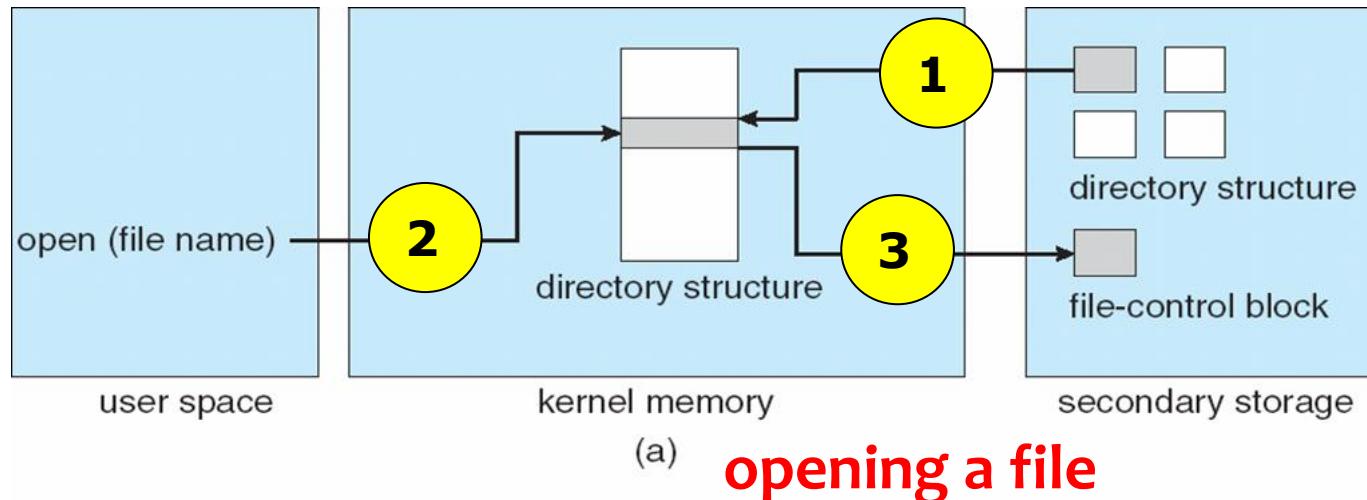
file owner, group, ACL

file size

file data blocks or pointers to file data blocks

In-Memory File System Structures

The necessary **file system structures** provided by the OS.



Virtual File Systems

Modern OS must concurrently **support multiple types of file systems.**

How does an OS allow multiple types of file systems to be integrated into a directory structure ?

To write directory and file routines for each type.

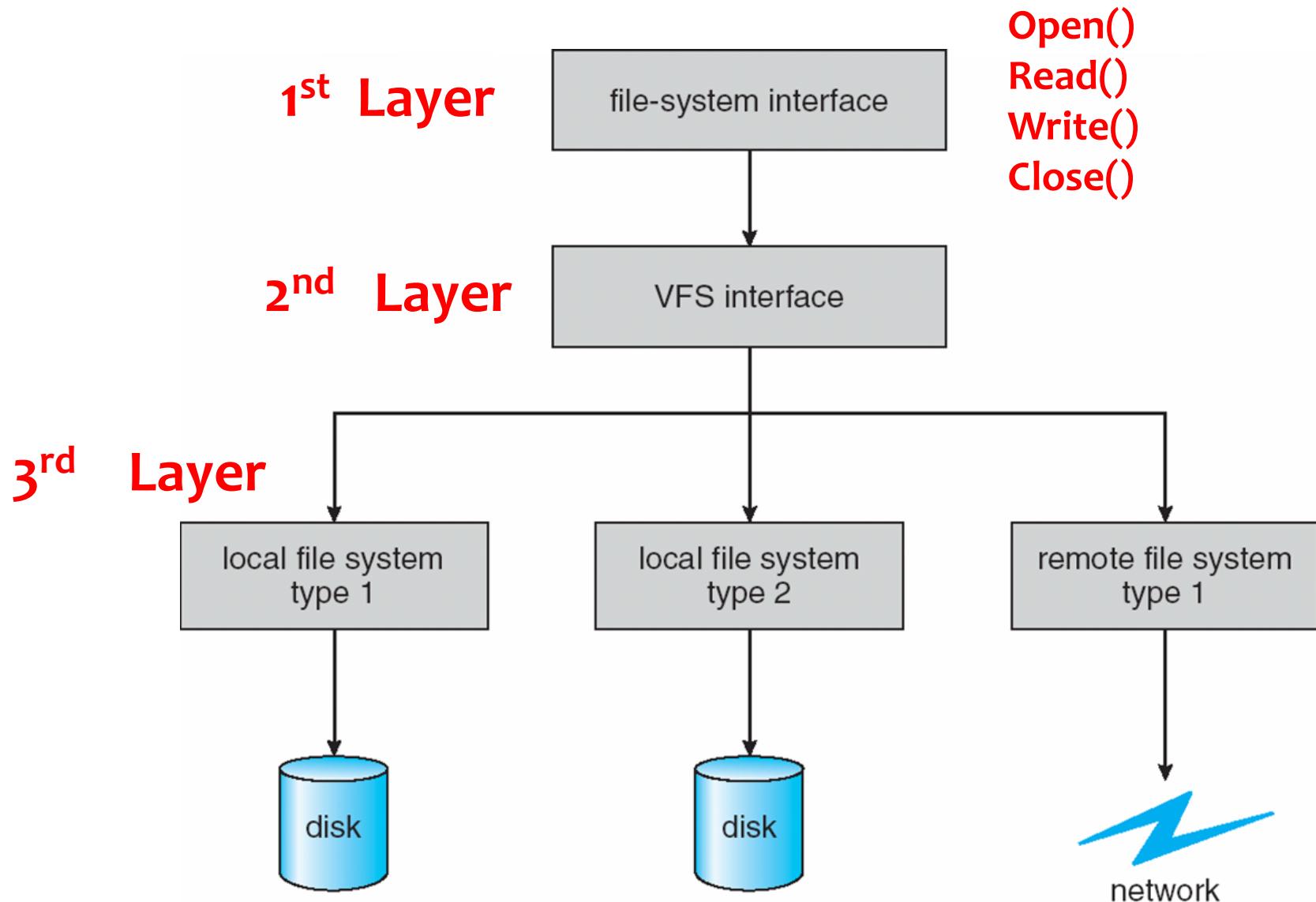
Most Operating systems provide an **object-oriented way of implementing file systems, including UNIX.**

The file-system implementation consists of **three major layers** (Figure 11.4)

The first layer is the **file-system interface**, based on open(), read(), write(), and close() calls

The 2nd layer is called the **virtual file system (VFS) layer**

Schematic View of a Virtual File System



Virtual File Systems

The VFS serves two important functions

It separates file-system-generic operations from their implementation by defining a clean VFS interface.

It provides a mechanism for uniquely representing a file throughout a network.

- ▶ The VFS is based on a file-representation structure, called a vnode, that contains numerical designator for a network-wide unique file.

Virtual File Systems

Unix **inodes** are unique only within a single file system

The kernel maintains one vnode structure for each active node (file or directory)

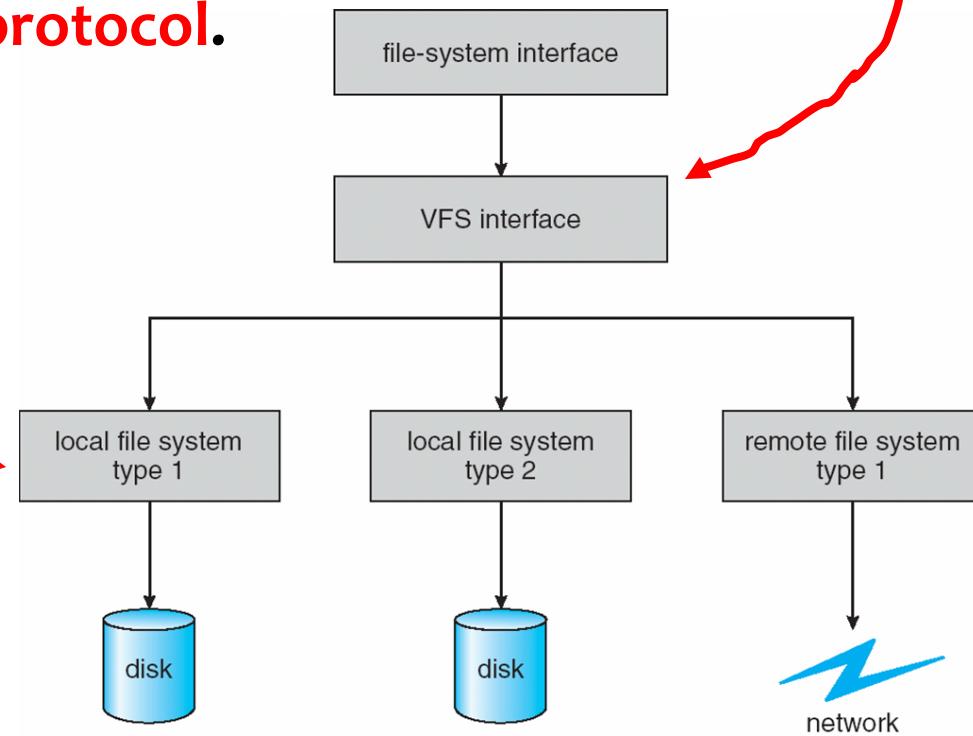
The VFS distinguishes local files from remote files, and local files are further distinguished according to their file-system types.

Virtual File Systems

VFS allows the **same system call interface (the API)** to be used for different types of file systems.

The API is to the **VFS interface**, rather than any specific type of file system.

The 3rd layer implements the **file-system type or the remote-file-system protocol**.



Virtual File Systems

The VFS architecture in **Linux** defines four major object types

The **inode object**, represents an individual file

The **file object**, represents an open file

The **Superblock object**, represents an entire file system

The **dentry object**, represents an individual directory entry.

Directory Implementation

The selection of **directory-allocation** and **directory-management algorithms** affects the efficiency, performance, and reliability of the file system

Linear list of file names with pointer to the data blocks.

simple to program

time-consuming to execute

Hash Table – linear list stores the directory entries, but a hash data structure is also used.

decreases directory search time

collisions – situations where two file names hash to the same location

fixed size

Allocation Methods

An allocation method refers to how disk blocks are allocated for files:

Contiguous allocation

Linked allocation

Indexed allocation

Contiguous Allocation

Each file occupies **a set of contiguous blocks** on the disk

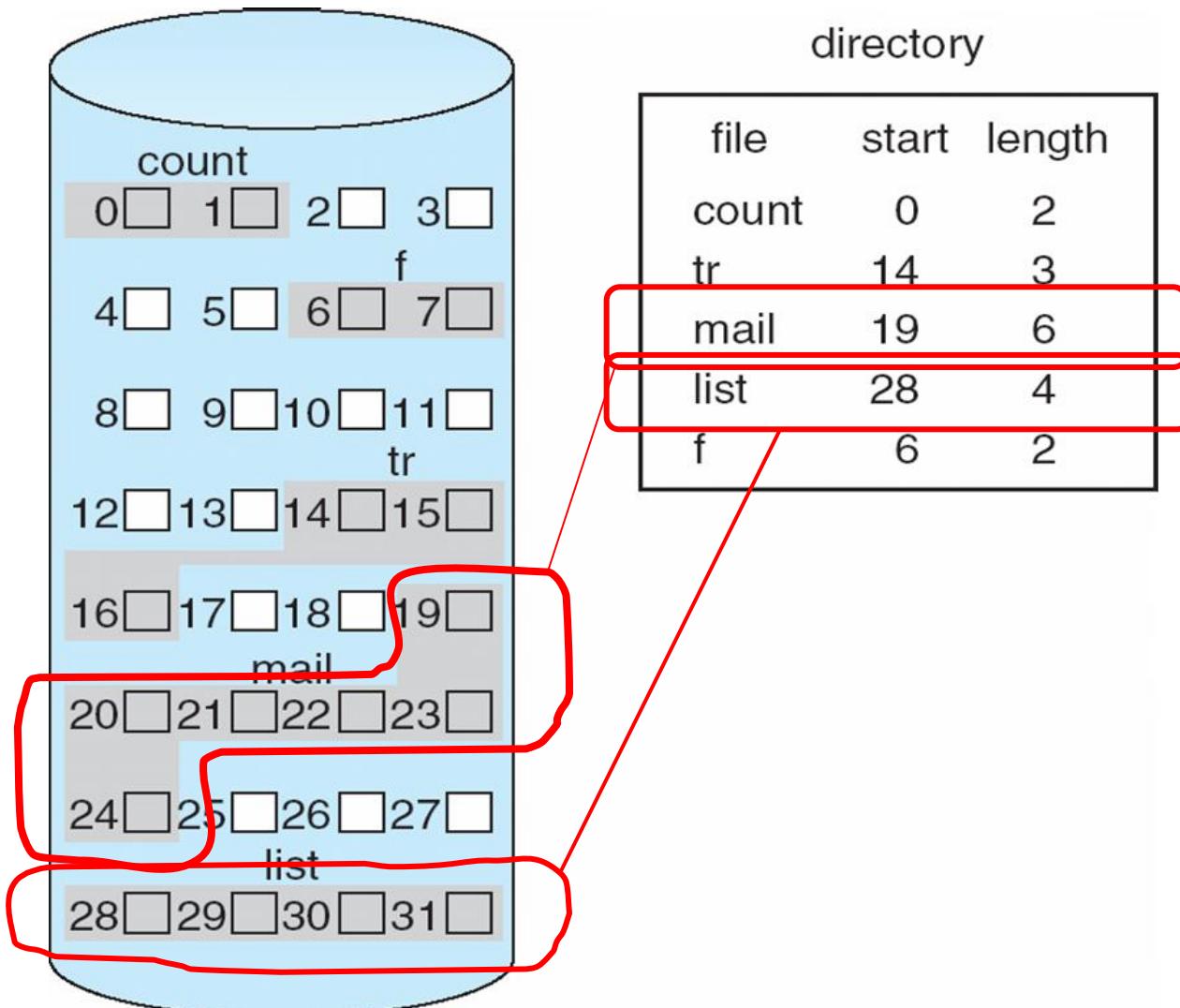
Simple – only starting location (block #) and length (number of blocks) are required

Random access

Wasteful of space (dynamic storage-allocation problem), **external fragmentation**

Files cannot grow

Contiguous Allocation of Disk Space



Extent-Based Systems

Many newer file systems use a **modified contiguous allocation scheme**

Extent-based file systems **allocate disk blocks in extents**

An extent is a contiguous block of disks

Extents are allocated for file allocation

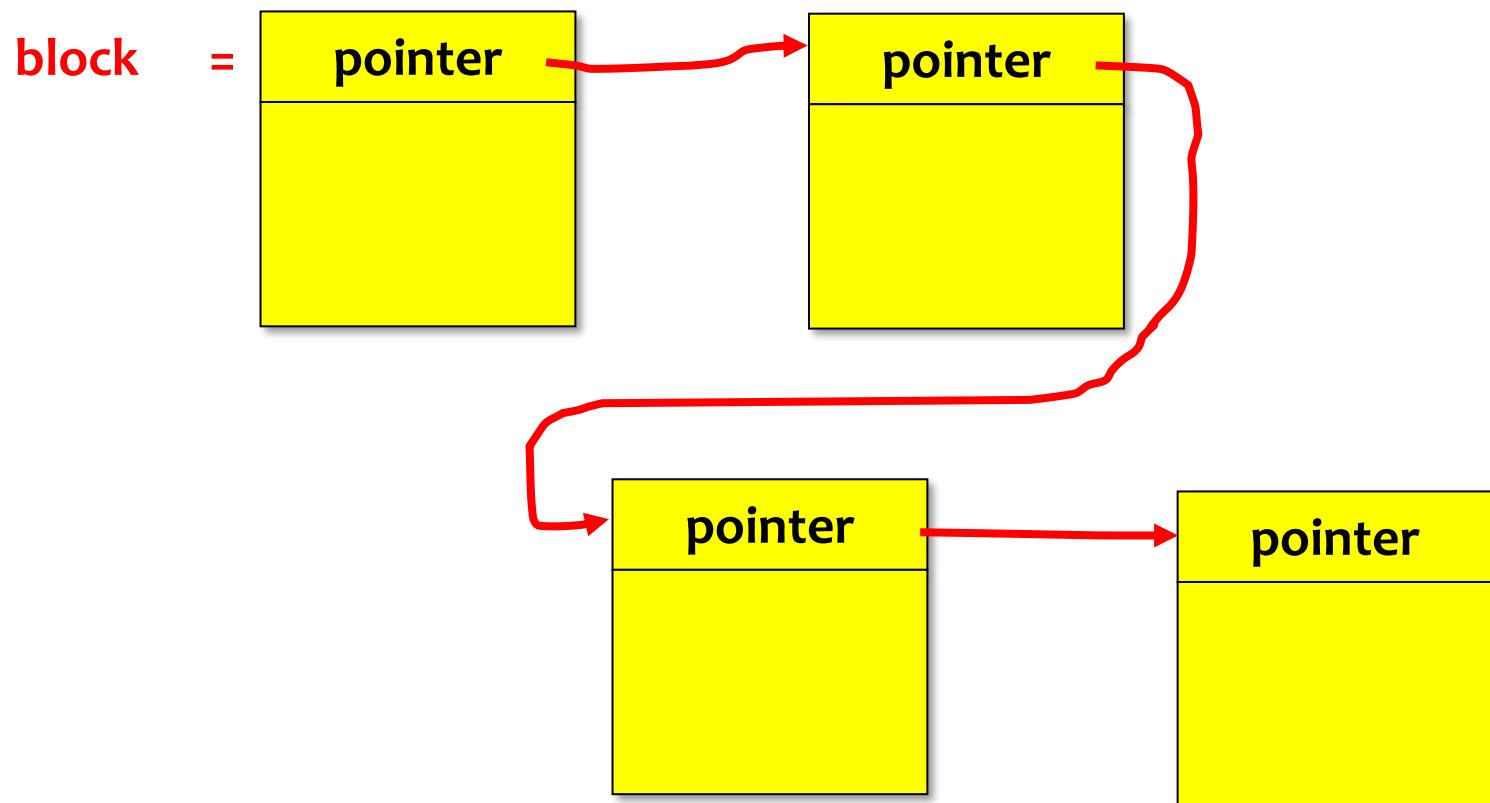
A file consists of one or more extents.

The location of a file's blocks is then recorded as **a location and a block count, plus a link to the first block of the next extent.**

The commercial Veritas File System uses extents to optimize performance. It is a high-performance replacement for standard UNIX UFS.

Linked Allocation

Each file is a linked list of disk blocks: **blocks may be scattered anywhere on the disk.**



Linked Allocation (Cont.)

Simple – need only starting address

Free-space management system – **no waste of space**

Some disadvantages

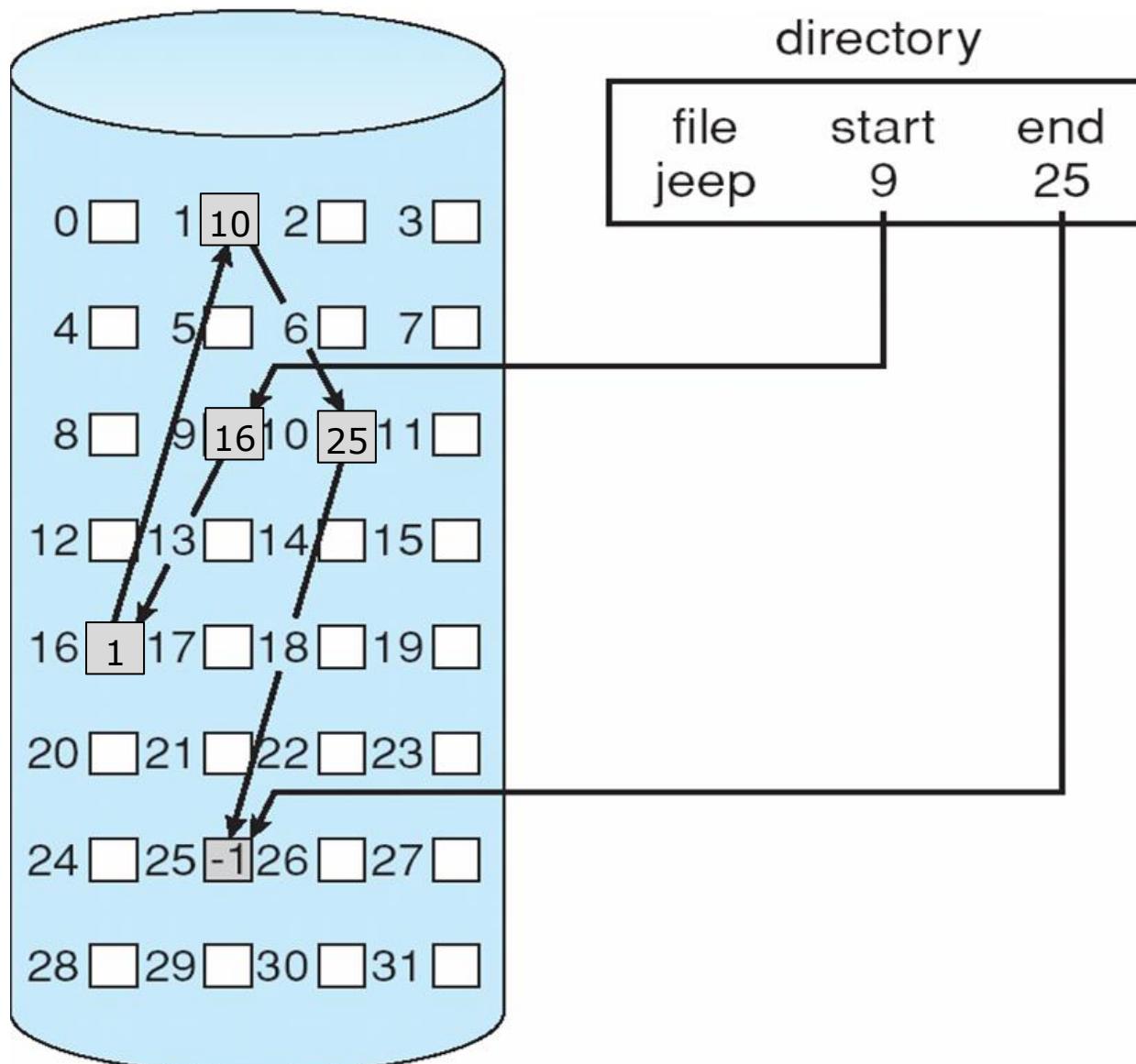
No random access (only for sequential access files)

Space required for pointers, If a pointer requires 4 bytes out of a 512 bytes block, 0.78 percent is used for pointers

- ▶ **Clusters (k blocks)**

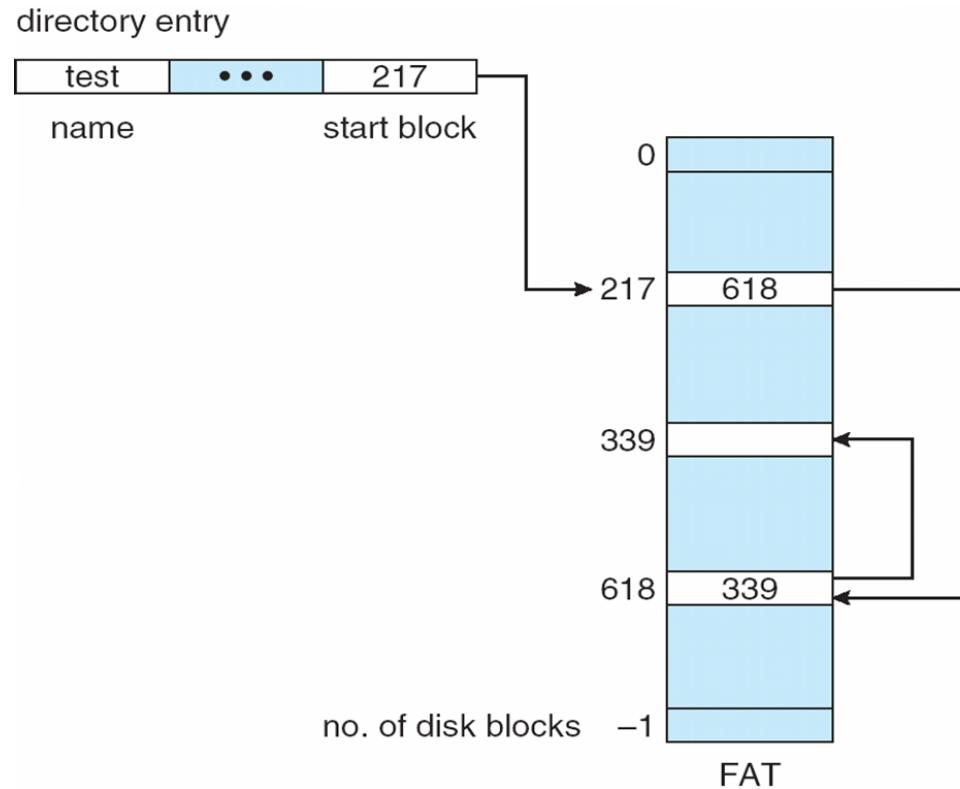
Reliability Issue

Linked Allocation



File-Allocation Table

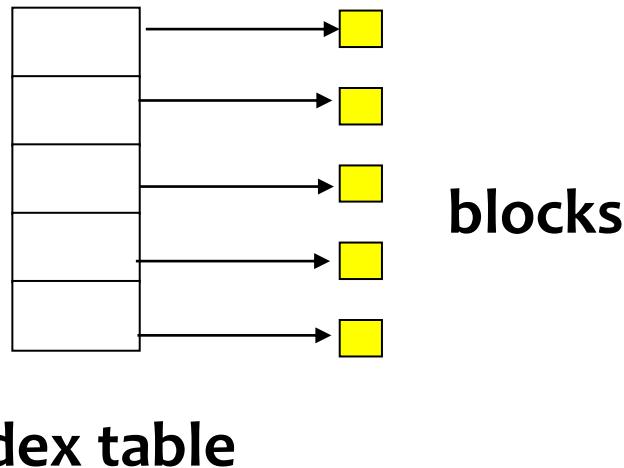
- An important variation on linked allocation is the use of a **file-allocation table (FAT)**
- The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached.



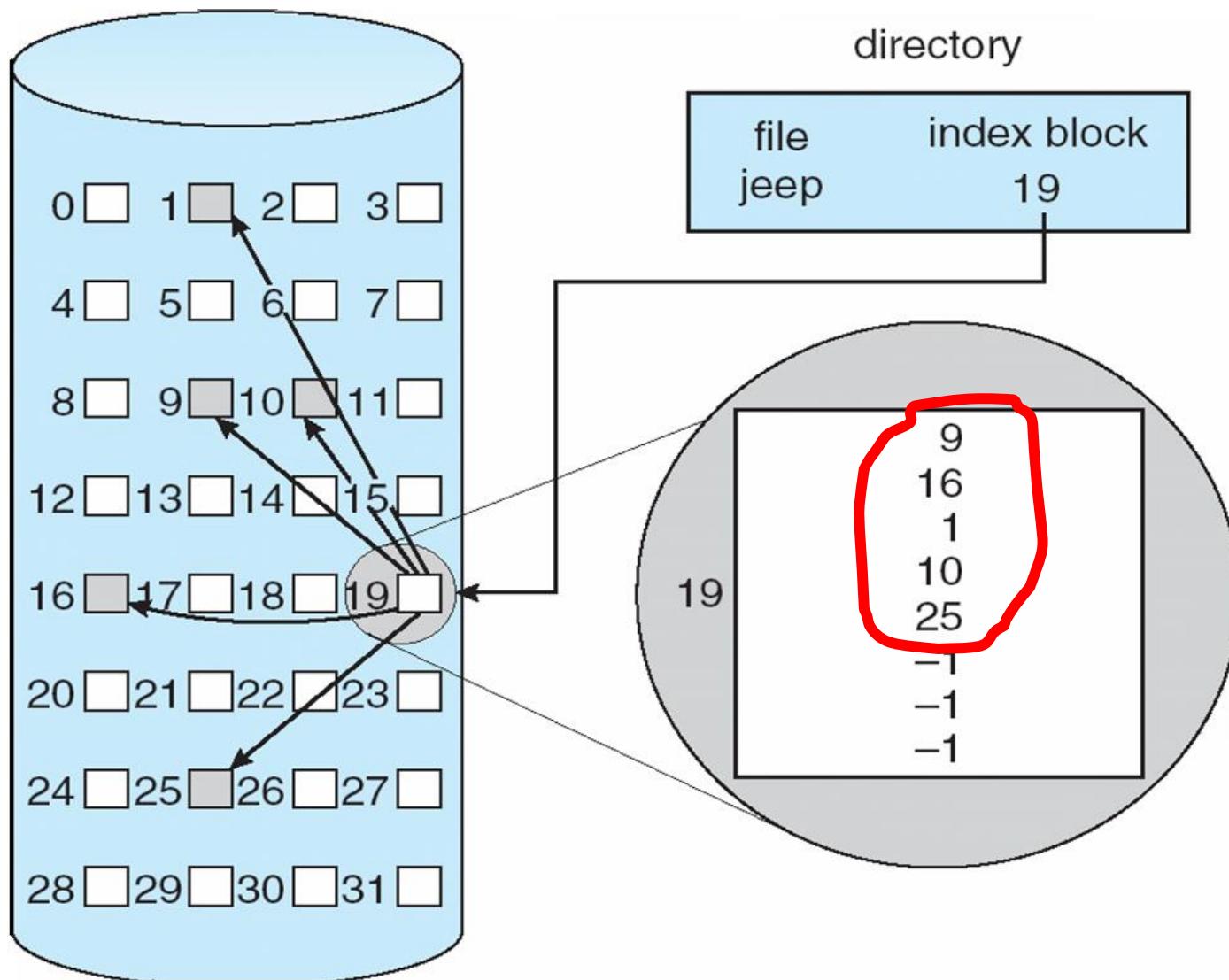
Indexed Allocation

Brings all pointers together into the **index block**.

Logical view.



Example of Indexed Allocation



Indexed Allocation (Cont.)

Need index table

Random access

Dynamic access without external fragmentation,
but have overhead of index block.

Mapping from logical to physical in a file of
maximum size of 256K words and block size of 512
words.

Only 1 block is needed for index table.

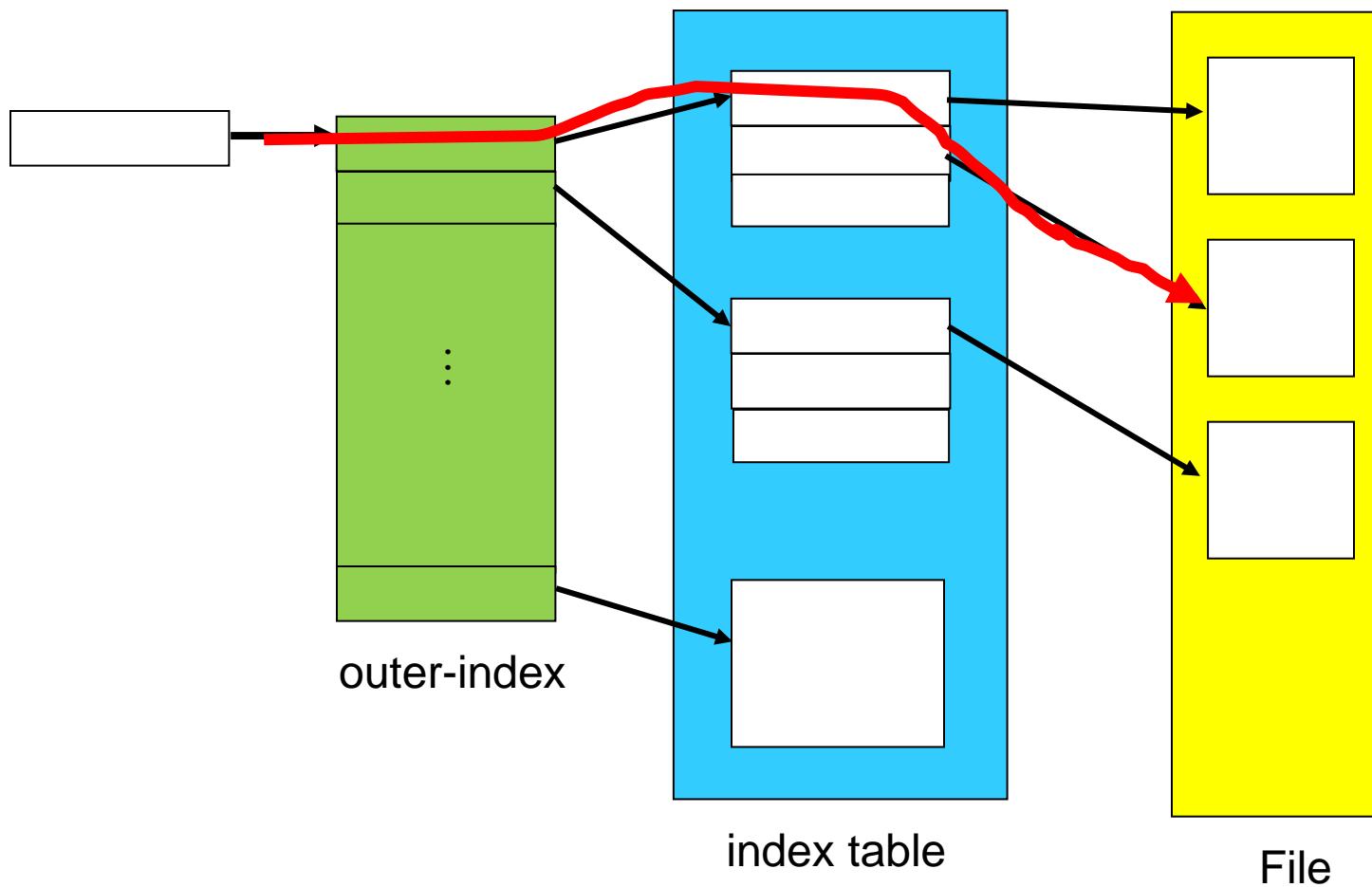
$$256K/0.5K = 512 \text{ blocks}$$

Indexed Allocation – Mapping (Cont.)

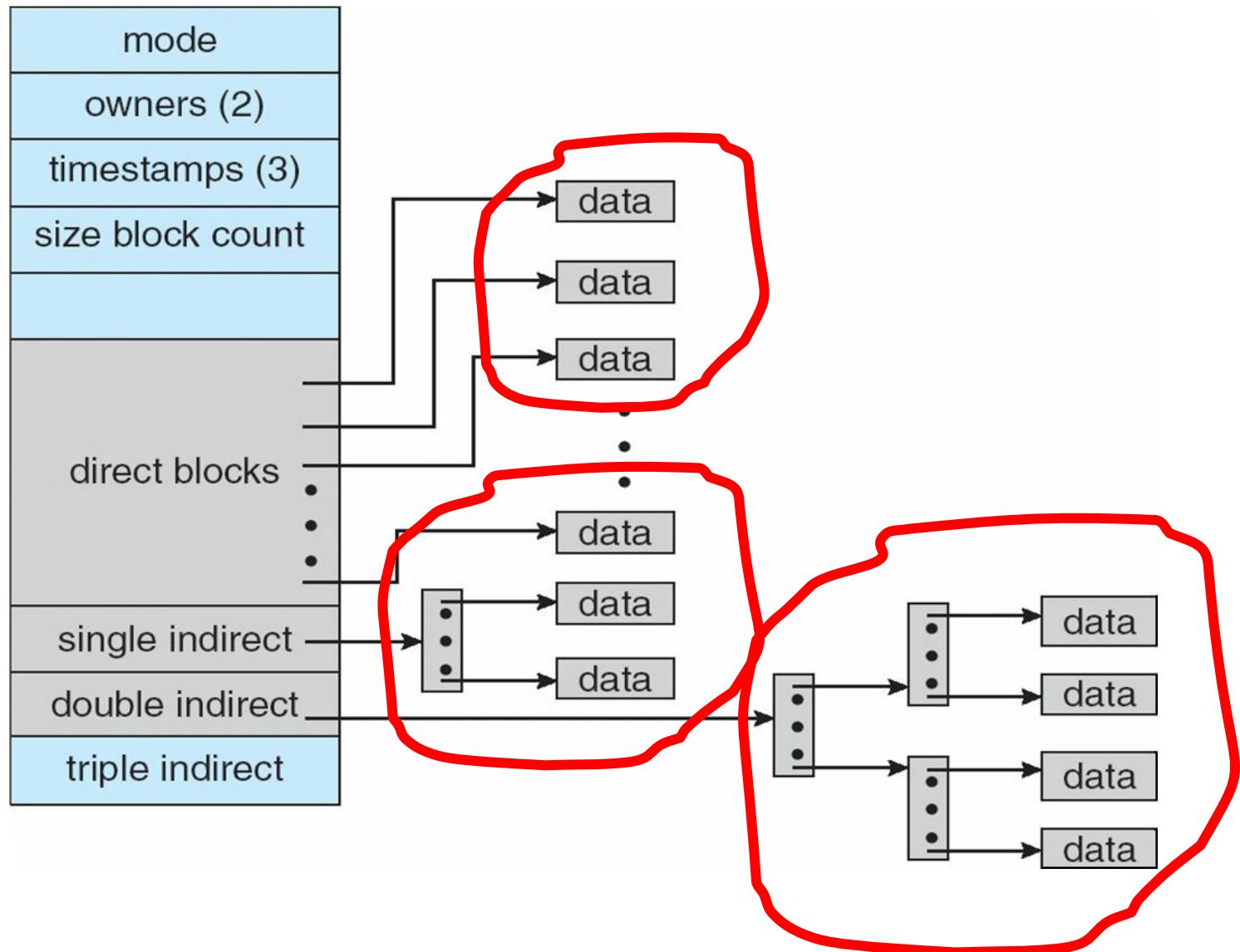
Mapping from logical to physical in a file of unbounded length (block size of 512 words).

Linked scheme – Link blocks of index table (no limit on size).

Indexed Allocation – Mapping (Cont.)

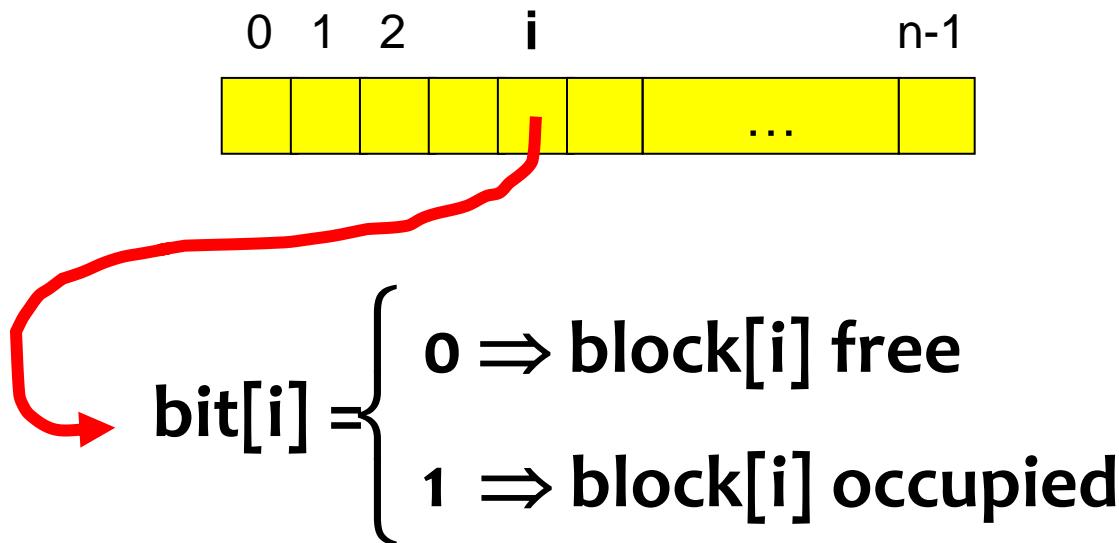


Combined Scheme: UNIX (4K bytes per block)



Free-Space Management

Bit vector (n blocks)



Free-Space Management (Cont.)

Bit map requires extra space

Example:

block size = 2^{12} bytes

disk size = 2^{30} bytes (1 gigabyte)

$n = 2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)

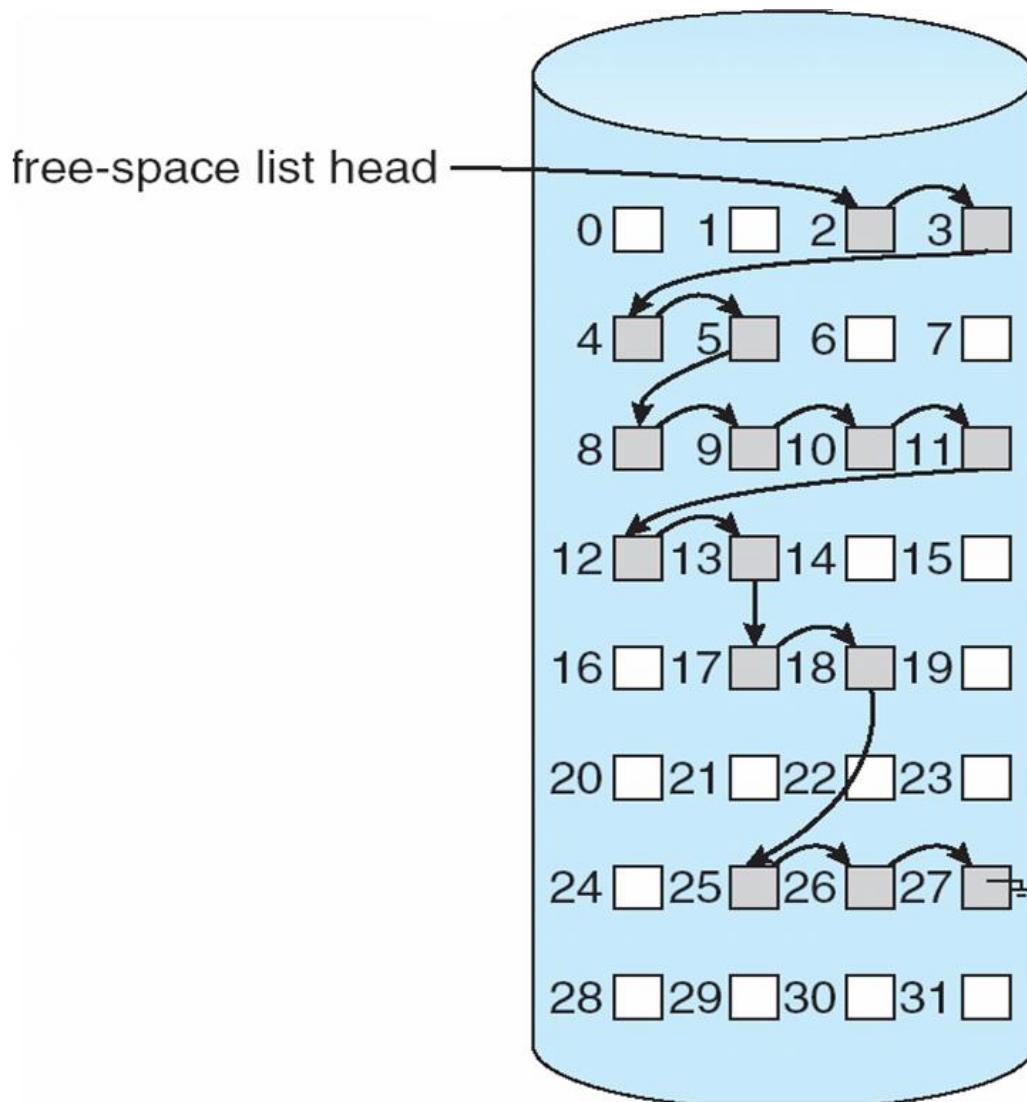
Easy to get contiguous files 000**1111111000**....

Linked list (free list)

Cannot get contiguous space easily

No waste of space

Linked Free Space List on Disk



Free-Space Management (Cont.)

Grouping: stores the **addresses of n free blocks** in the first free block. A block has **n** entries.

The first $n-1$ of these blocks are actually free.

The last block contains the addresses of another n free blocks, and so on.

Counting: keep the address of the first free block and the number (n) of free contiguous blocks that follow the first block.

Address 1, n_1 (2,3)

Address 2, n_2 , (8,5)

Address 3, n_3 , (17,2) ...

Free-Space Management (Cont.)

Need to protect:

Pointer to free list

Bit map

- ▶ Must be kept on disk
- ▶ Copy in memory and disk may differ
- ▶ Cannot allow for $\text{block}[i]$ to have a situation where **bit[i] = 1** (occupied) in memory and **bit[i] = 0** (free) on disk

Solution:

- ▶ Set $\text{bit}[i] = 1$ in disk
- ▶ Allocate $\text{block}[i]$
- ▶ Set $\text{bit}[i] = 1$ in memory

Efficiency and Performance

Efficiency

The Efficient use of disk space depends heavily on the **disk allocation and directory algorithms**

- ▶ UNIX inodes are **preallocated** on a volume. Even a “empty” disk has a percentage of it space lost to inodes.
- ▶ By preallocating, the inodes and spreading them across the volume, we improve the file system’s performance. --- Try to **keep a file’s data block near that file’s inode block to reduce the seek time**.

The type of data kept in file’s directory (or inode) entry also require consideration

- ▶ “Last write date”
- ▶ “Last access date”

Efficiency and Performance

Performance

disk cache – separate section of main memory for frequently used blocks

free-behind and read-ahead – techniques to optimize sequential access

- ▶ **Free-behind** removes a page from the buffer as soon as the next page is requested.
- ▶ With **read-ahead**, a requested page and several subsequent pages are read and cached.

improve PC performance by dedicating section of memory as **virtual disk, or RAM disk**

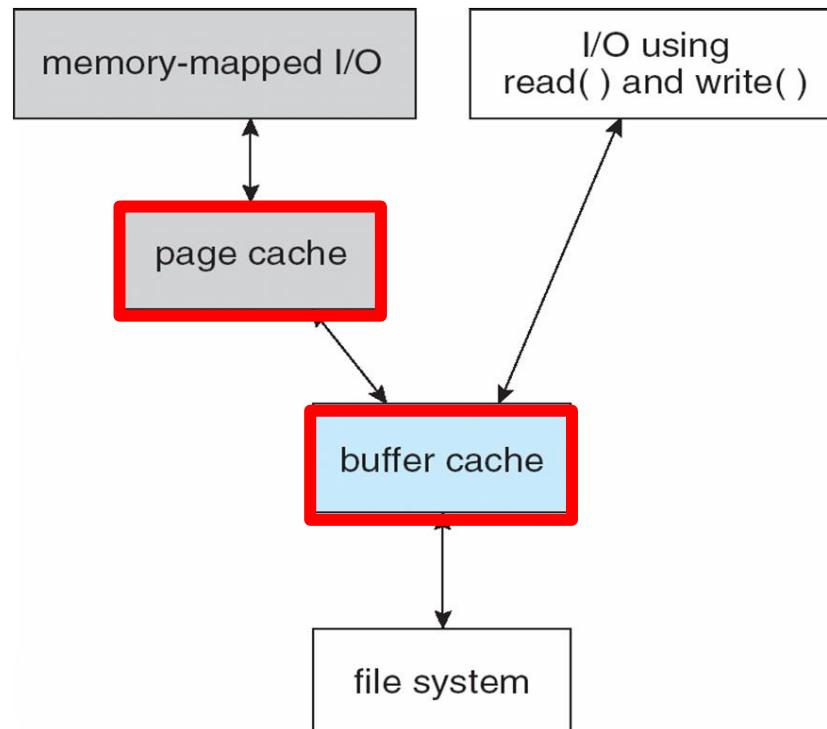
Page Cache

A **page cache** caches pages rather than disk blocks using virtual memory techniques

Memory-mapped I/O uses a **page cache**

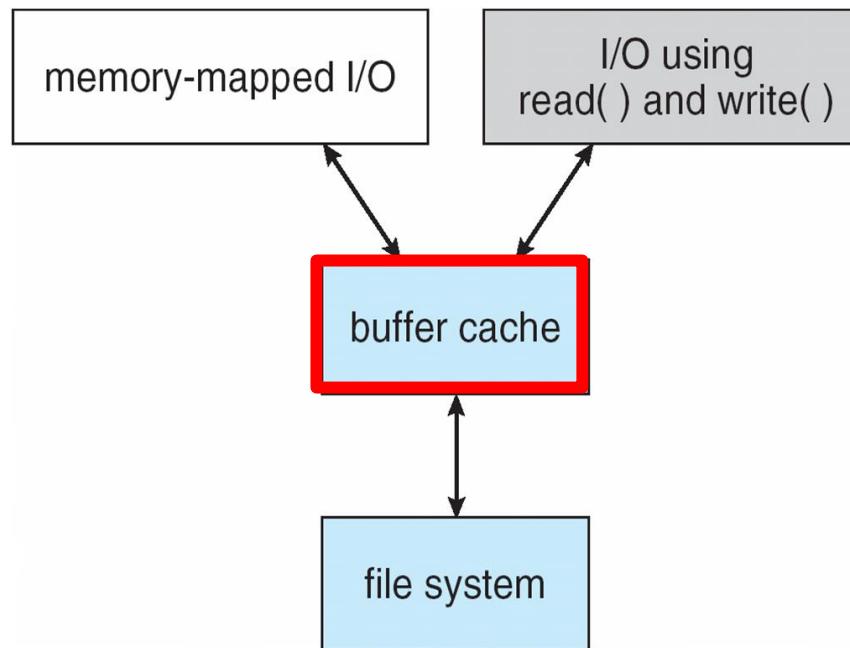
Routine I/O through the file system uses the buffer (disk) cache

Double caching



Unified Buffer Cache

A **unified buffer cache** uses the same page cache to cache both memory-mapped pages and ordinary file system I/O



I/O Using a Unified Buffer Cache

Recovery

Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies

Use system programs to **back up** data from disk to another storage device (floppy disk, magnetic tape, other magnetic disk, optical)

Recover lost file or disk by **restoring** data from backup

Log Structured File Systems

Log structured file systems record each update to the file system as a **transaction**

All transactions are written to a **log**

A transaction is considered **committed** once it is written to the log

However, the file system may not yet be updated

The transactions in the log are **asynchronously written to the file system**

When the file system is modified, the transaction is removed from the log

If the file system crashes, all remaining transactions in the log must still be performed

The Sun Network File System (NFS)

An implementation and a specification of a software system for **accessing remote files across LANs (or WANs)**

The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol and Ethernet)

Interconnected workstations viewed as **a set of independent machines with independent file systems**, which allows sharing among these file systems in a **transparent** manner

NFS (Cont.)

A **remote directory is mounted over a local file system directory**

- ▶ The **mounted directory** looks like **an integral subtree** of the local file system, replacing the subtree descending from the local directory

Specification of the remote directory for the mount operation is **nontransparent**; the **host name** of the remote directory has to be provided

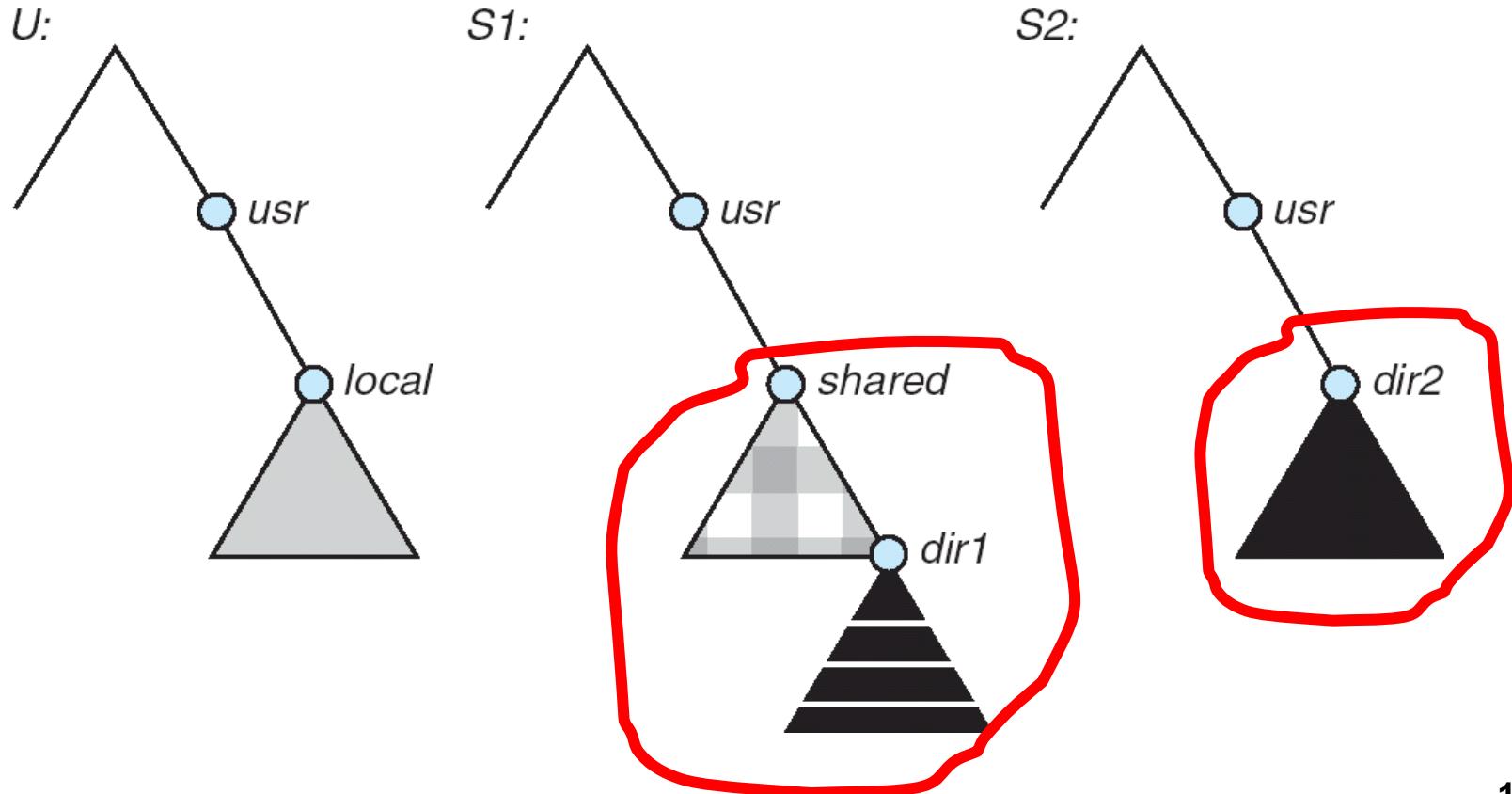
- ▶ Files in the remote directory can then be accessed in a **transparent manner**

Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory

NFS (Cont.)

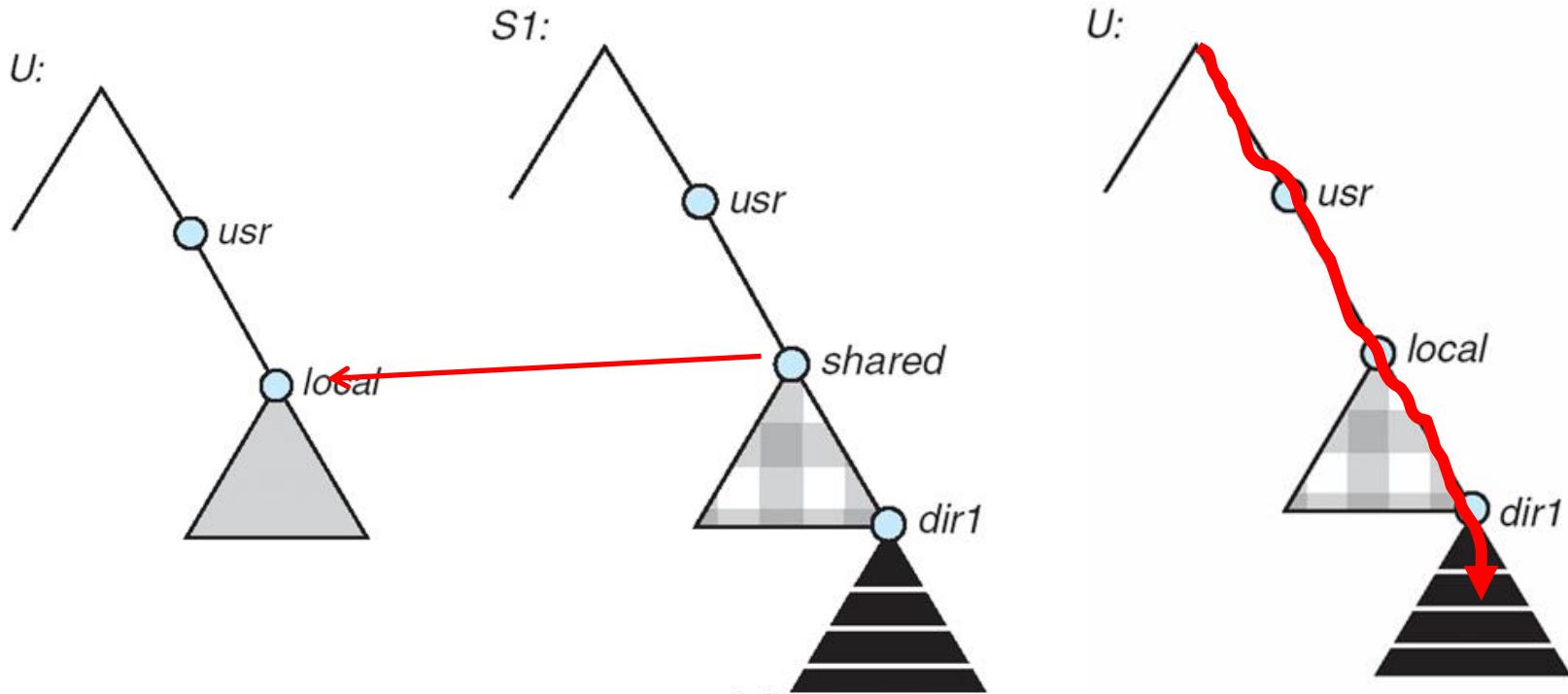
Consider the following file system where the **triangles** represent subtrees of directories that are of interest.

Three independent file systems of machines: **U, S1, and S2**



NFS (Cont.)

The effect of mounting **S1:/user/shared** over **U:/user/local**



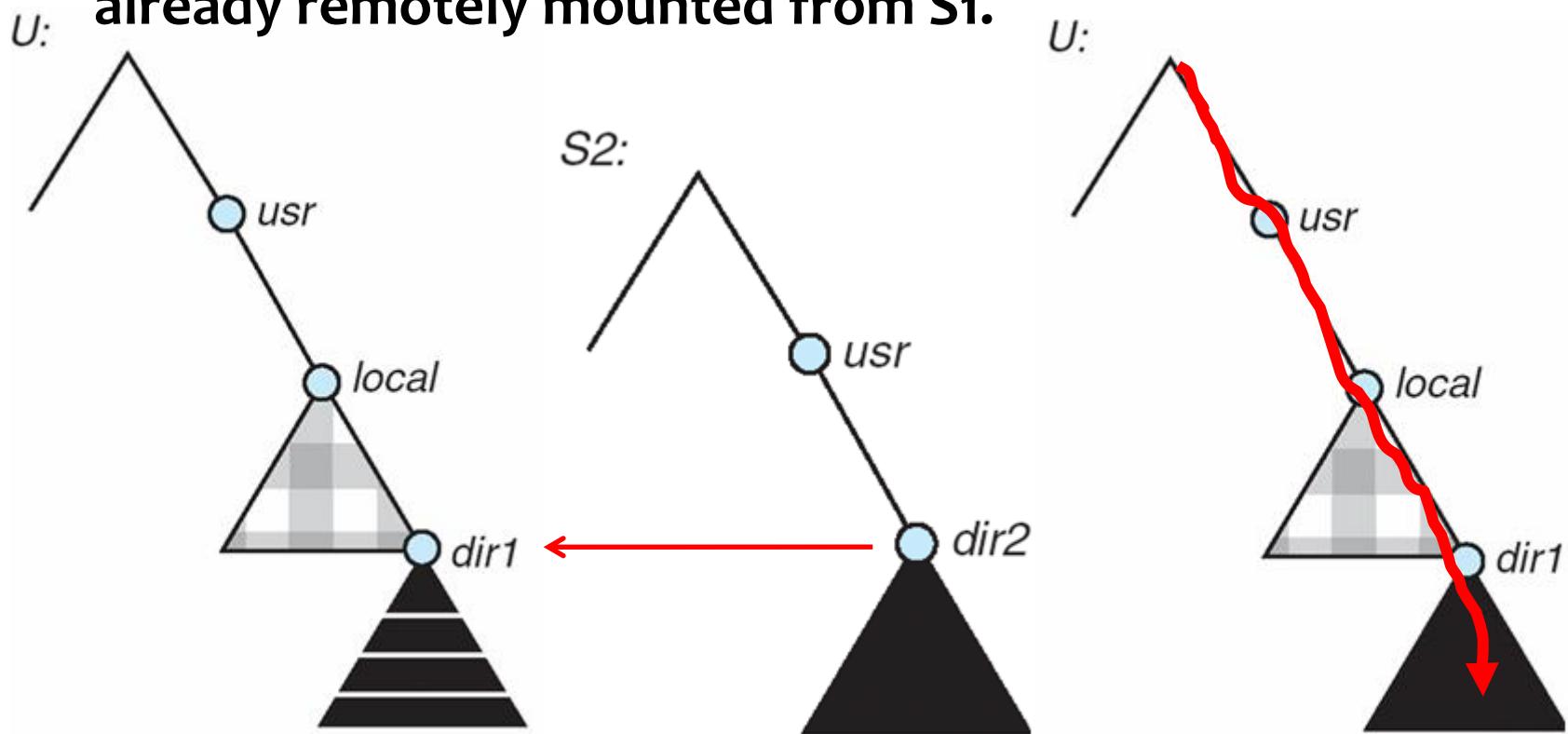
The machine U can access any file within the dir1 directory using prefix **/usr/local/dir1**.

The original **/usr/local** on that machine is no longer visible.

NFS (Cont.)

Cascading mounts.

mounting **S2:/user/dir2** over **U:/user/local/dir1** which is already remotely mounted from **S1**.



Users can access files within the dir2 on U using prefix /usr/local/dir1.

NFS (Cont.)

User mobility

If a shared file system is mounted over a user's home directories on all machines in a network, the user can log into any workstation and get his home environment.

NFS (Cont.)

One of the design goals of NFS was to operate in a heterogeneous environment of different machines, operating systems, and network architectures;

the NFS specifications independent of these media

This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces

NFS (Cont.)

The NFS specification distinguishes between the services provided by a **mount mechanism** and the actual **remote-file-access services**.

Accordingly, two separate protocols are specified for these services:

- a **mount protocol** and

- an **NFS protocol** for remote file accesses.

The protocols are specified as **sets of RPCs**.

These RPCs are the building blocks used to implement transparent remote file access.

NFS Mount Protocol

The **mount protocol** establishes initial logical connection between a server and a client.

A mount operation includes

name of remote directory to be mounted and

name of server machine storing it

Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine

The server maintains an **export list** – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them

NFS Mount Protocol

In Solaris, this list is the **/etc/dfs/dfstab**, which can be edited only by a supervisor.

Any **directory** within an exported file system can be mounted by an accredited machine.

A **component unit** is such a directory.

Following a mount request that conforms to its export list, the server returns a **file handle** — a key for further accesses

In UNIX terms, the file handle consists of a **file-system identifier**, and **an inode number** to identify the mounted directory within the exported file system

The mount operation changes only the user's view and does not affect the server side

NFS Protocol

The NFS protocol provides **a set of RPCs for remote file operations.**

The procedures support the following operations:

- searching for a file within a directory

- reading a set of directory entries

- manipulating links and directories

- accessing file attributes

- reading and writing files

These procedures can be invoked only after a file handle for the remotely mounted directory has been established.

NFS Protocol

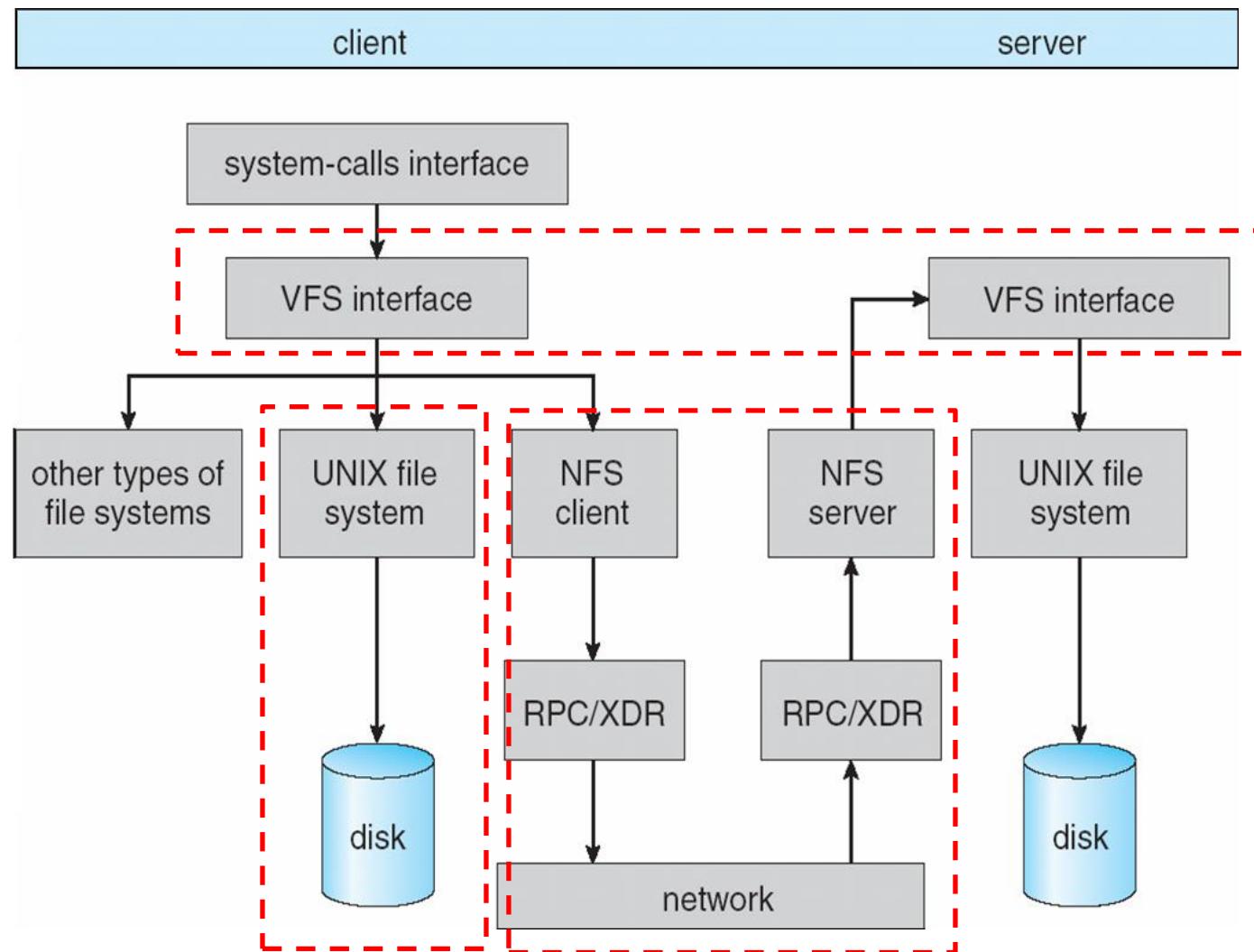
NFS servers are **stateless**; Servers do not maintain information about their clients from one access to another. Each request has to provide a full set of arguments.

Every NFS request has a **sequence number**, allowing the server to determine if a request is duplicated or missed.

Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching)

The NFS protocol **does not** provide concurrency-control mechanisms

Schematic View of NFS Architecture



Three Major Layers of NFS Architecture

UNIX file-system interface (based on the open, read, write, and close calls, and file descriptors)

Virtual File System (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types

The VFS activates file-system-specific operations to handle **local requests** according to their file-system types

Calls the NFS protocol procedures for **remote requests**

NFS service layer – bottom layer of the architecture

Implements the NFS protocol

An illustration of NFS Architecture

NFS is integrated into the OS via a **VFS**.

Let's trace how the operation on an already open remote file is handled.

The client initiates the operation with a regular system call.

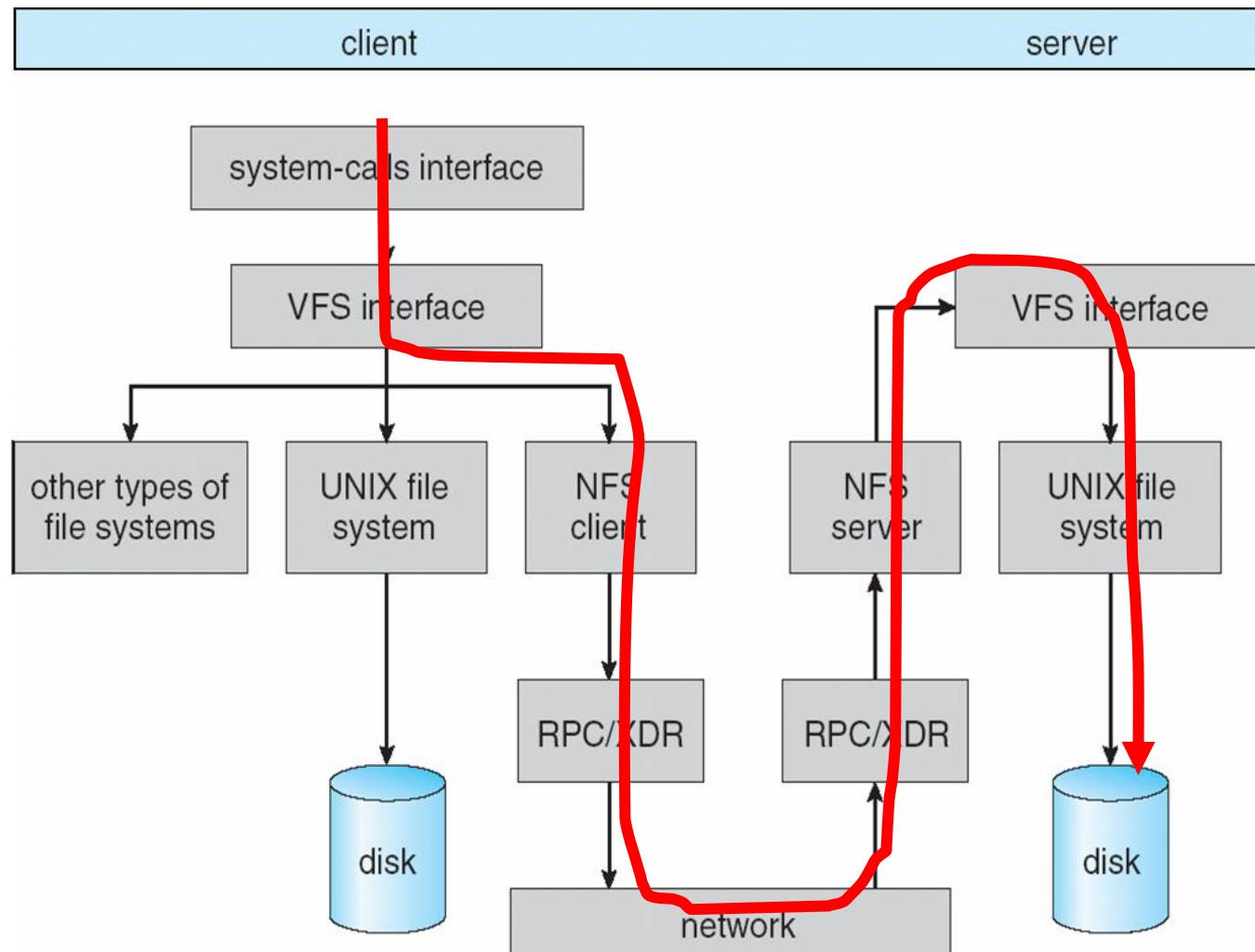
The **OS layer** maps this call to a VFS operation on the appropriate vnode.

The **VFS layer** identifies the file as a remote one and invokes the appropriate NFS procedure.

An RPC call is made to the **NFS service layer** at the remote server. This call is reinjected to the VFS layer on the remote system, which finds that it is local and invokes the appropriate file system operation.

This path is retraced to return the results.

Schematic View of NFS Architecture



NFS Path-Name Translation

Path-name translation in NFS involves the parsing of a path name such as **/usr/local/dir1/file.txt** into separate directory entries, or components: **(1) usr, (2) local, and (3) dir1.**

Performed by breaking the path into **component names** and performing a separate NFS lookup call for every pair of component name and directory vnode.

Once a mount point is crossed, every component lookup causes a separate RPC to the server.

To make lookup faster, a directory name **lookup cache** on the client's side holds the vnodes for remote directory names

NFS Remote Operations

Nearly one-to-one correspondence between regular UNIX system calls for file operations and the NFS protocol RPCs (except opening and closing files).

A remote file operation can be translated directly to the corresponding RPC.

NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance.

There are two caches: the file-attribute cache and the file-blocks cache.

NFS Remote Operations

When a file is opened, the kernel checks with the remote server whether to fetch or revalidate the **cached attributes**.

The cached file blocks are used only if the corresponding cached attributes are up to date.

The attribute cache is updated whenever new attributes arrive from the server.

Cached attributes are discarded after 60 seconds.

Both **read-ahead** and **delayed-write** techniques are used between the server and the client.

Clients do not free delayed-write blocks until the server confirms that the data have been written to disk.

Example: WAFL File System

Disk I/O has a huge impact on system performance.

Some file systems are general purpose, others are optimized for specific tasks,

The WAFL (“Write-anywhere file layout”) file system from Network Appliance, is a powerful file system optimized for random writes.

Used on Network Appliance “Files” – distributed file system appliances

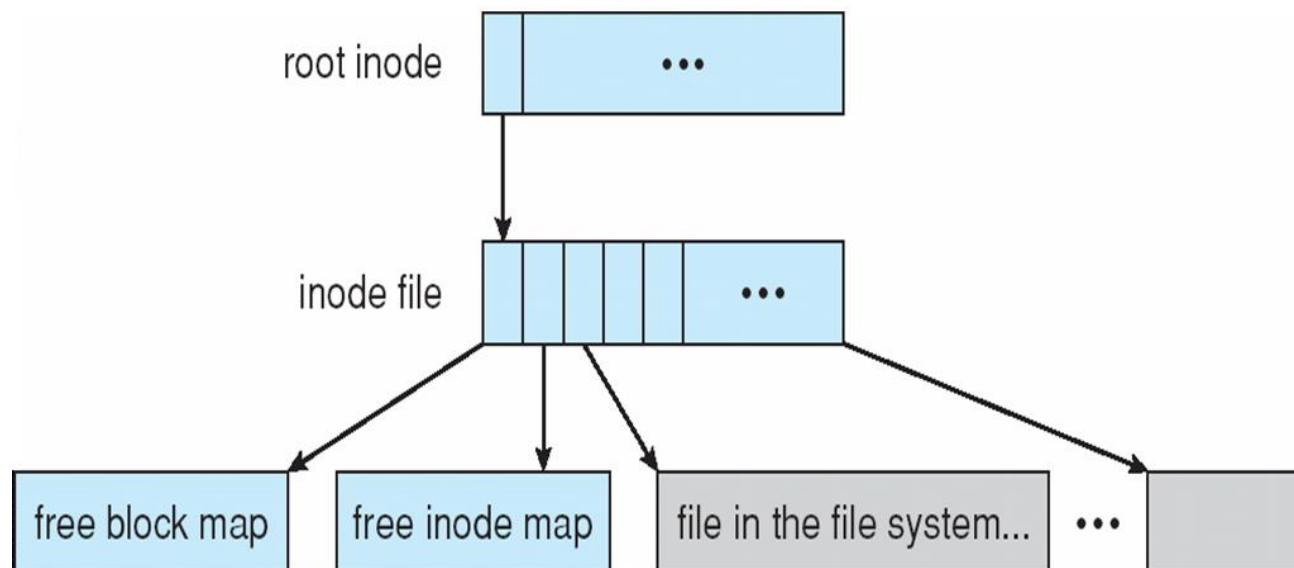
Serves up NFS, CIFS, http, ftp protocols

Random I/O optimized, write optimized

Example: WAFL File System

Similar to Berkeley Fast File System, with extensive modifications.

It is block-based and uses **inodes** to describe files. Each inode contains 16 pointers to blocks (or indirect blocks) belonging to the file described by the inode.



The WAFL File Layout

Example: WAFL File System

Each file system has a **root inode**.

A WAFL file system is **a tree of blocks** with the root inode as its base.

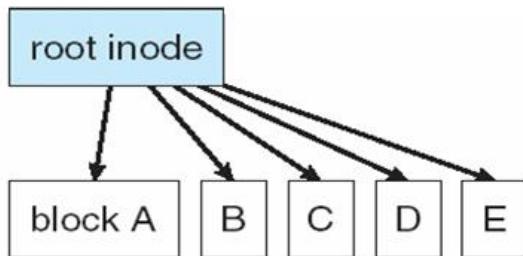
To take a **snapshot**, WAFL creates **a copy of the root inode**.

Any file or metadata updates after that go to new blocks rather than overwriting their existing blocks.

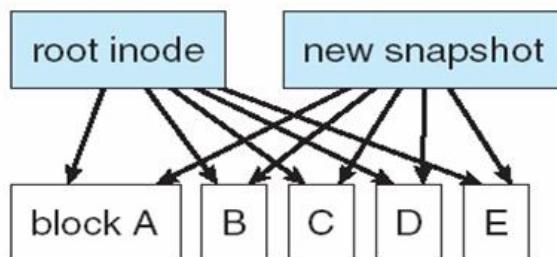
Used blocks are never overwritten, so writes are very fast, because **a write can occur at the free block nearest the current head location**.

The snapshot facility is also useful **for backups, testing, versioning**, and so on.

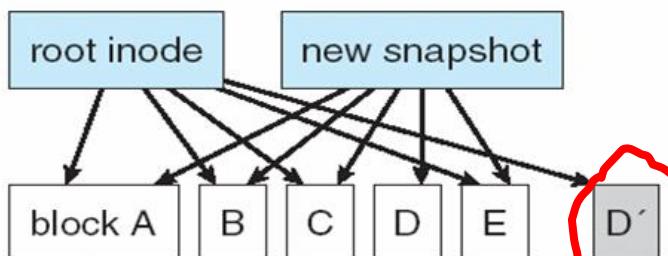
Snapshots in WAFL



(a) Before a snapshot.



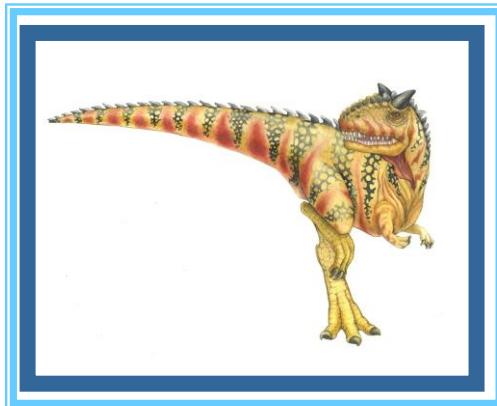
(b) After a snapshot, before any blocks change.



(c) After block D has changed to D'.

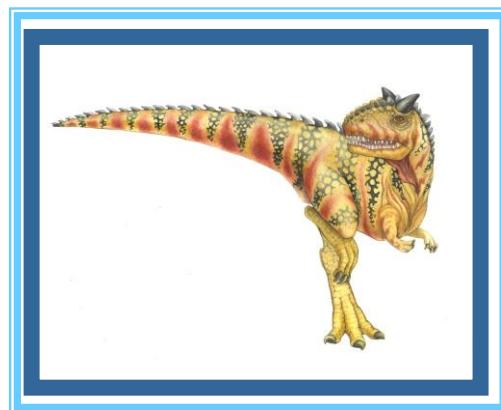
A write can occur at the free block nearest the current head location

End of Chapter 11



Chapter 12

Secondary-Storage Structure



Chapter 12: Secondary-Storage Structure

Overview of Mass Storage Structure

Disk Structure

Disk Attachment

Disk Scheduling

Disk Management

Swap-Space Management

RAID Structure

Stable-Storage Implementation

Tertiary Storage Devices

Objectives

Describe the **physical structure** of secondary and tertiary storage devices and the resulting effects on the uses of the devices

Explain the **performance characteristics** of mass-storage devices

Discuss operating-system services provided for mass storage, including **RAID** and **HSM**
(Hierarchical Storage Management)

12.1 Overview of Mass Storage Structure

Magnetic disks provide bulk of secondary storage of modern computers

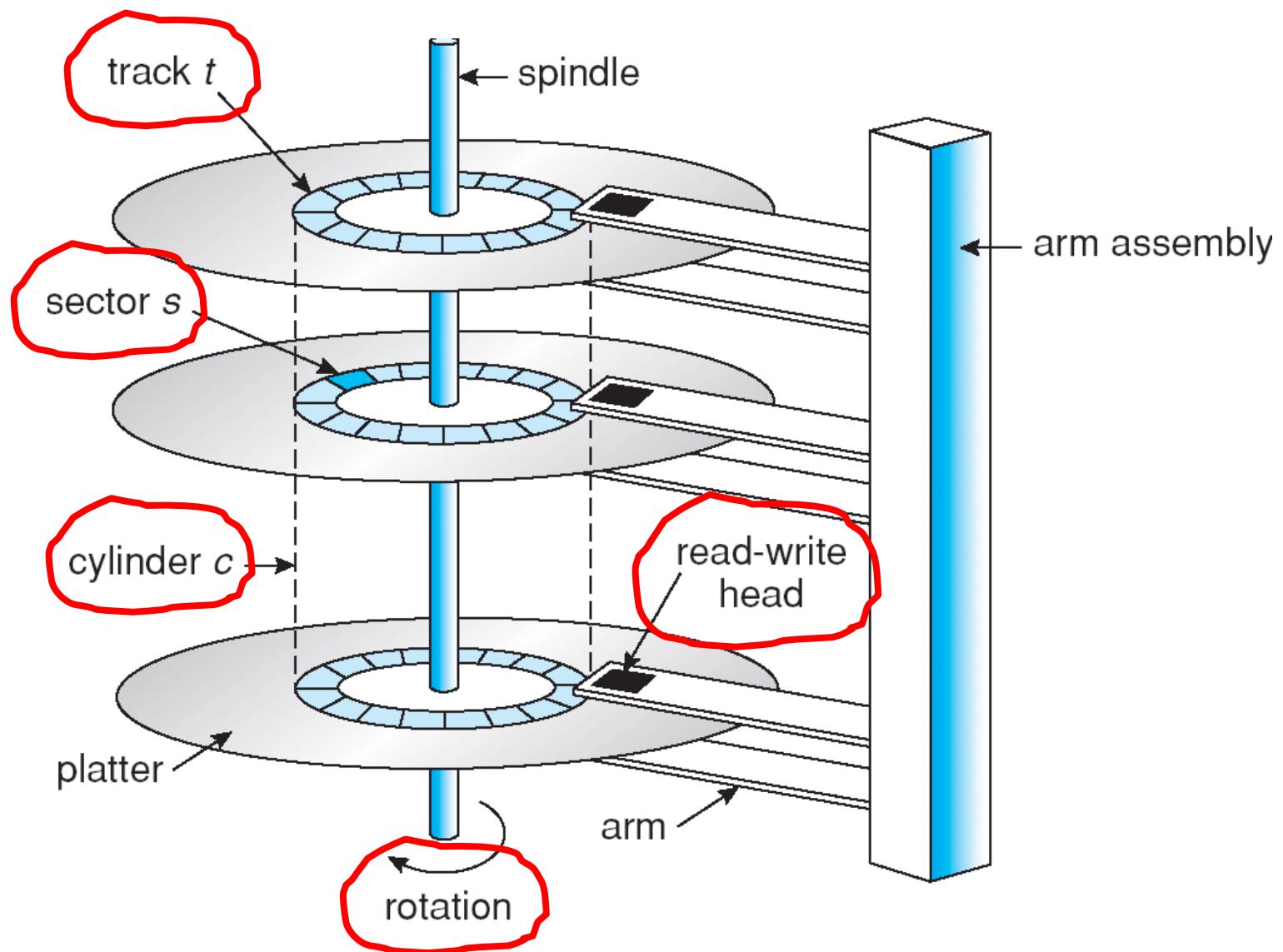
Drives rotate at 60 to 200 times per second

Transfer rate is rate at which data flow between drive and computer

Positioning time (random-access time) is time to move disk arm to desired cylinder (**seek time**) and time for desired sector to rotate under the disk head (**rotational latency**)

Head crash results from disk head making contact with the disk surface → That's bad

Moving-head Disk Mechanism



Overview of Mass Storage Structure

Disks can be removable

Drive attached to computer via **I/O bus**

Busses vary, including **EIDE, ATA, SATA, USB, Fibre Channel (FC), SCSI**

Host controller in computer uses bus to talk to **disk controller** built into drive or storage array

Overview of Mass Storage Structure (Cont.)

Magnetic tape

Was early secondary-storage medium

Relatively permanent and holds large quantities of data

Access time slow

Random access ~1000 times slower than disk

Mainly used for backup, storage of infrequently-used data, transfer medium between systems

Kept in spool and wound or rewound past read-write head

Once data under head, transfer rates comparable to disk

20-200GB typical storage

12.2 Disk Structure

Disk drives are **addressed** as large **1-dimensional arrays** of ***logical blocks***, where the logical block is the smallest unit of transfer.

The **1-dimensional array of logical blocks** is mapped into the **sectors** of the disk sequentially.

Sector 0 is the first sector of the first track on the outermost cylinder.

Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

12.3 Disk Attachment

Host-attached storage accessed through I/O ports talking to I/O busses

SCSI itself is a bus, up to 16 devices on one cable, **SCSI initiator** requests operation and **SCSI targets** perform tasks

Each target can have up to **8 logical units** (disks attached to device controller)

FC (Fiber Channel) is high-speed serial architecture

Can be **switched fabric** with 24-bit address space – the basis of **storage area networks (SANs)** in which many hosts attach to many storage units

Can be **arbitrated loop (FC-AL)** of 126 devices

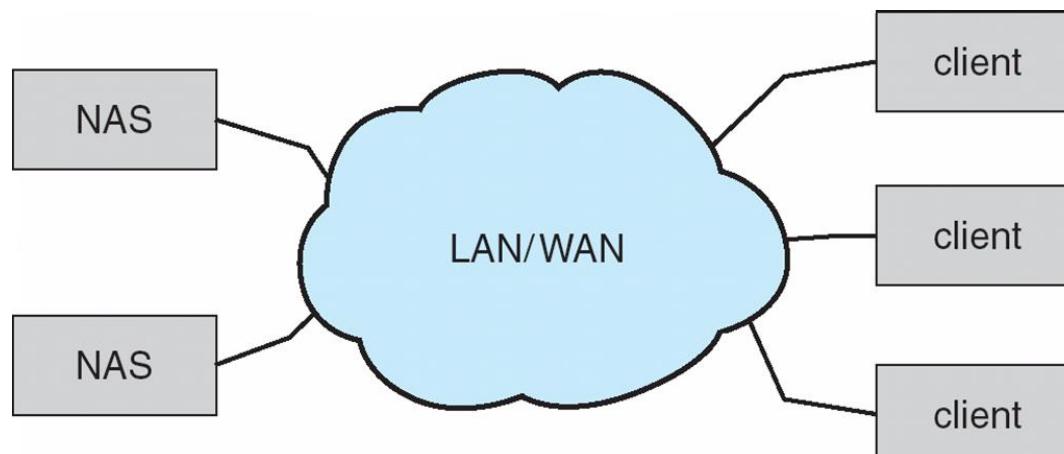
Network-Attached Storage (NAS)

Network-attached storage (NAS) is storage made available over a network rather than over a local connection (such as a bus)

NFS and CIFS are common protocols

Implemented via remote procedure calls (RPCs) between host and storage

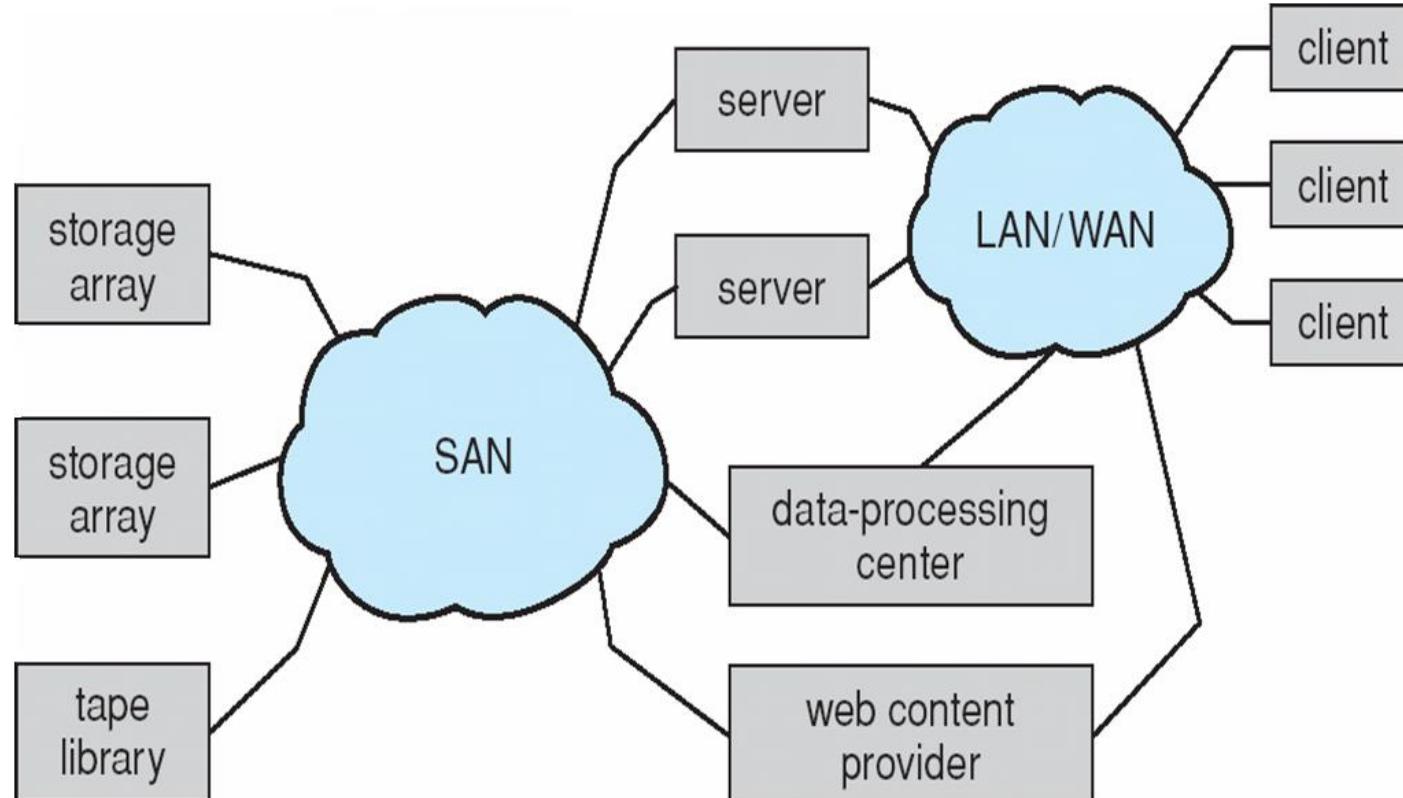
New iSCSI protocol uses IP network to carry the SCSI protocol



Storage Area Network (SAN)

Common in large storage environments (and becoming more common)

Multiple hosts attached to multiple storage arrays - flexible



12.4 Disk Scheduling

The operating system is responsible for using hardware efficiently — for the disk drives, this means having a **fast access time and disk bandwidth**.

Access time has two major components

Seek time is the time for the disk arm to move the heads to the cylinder containing the desired sector.

Rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head.

Minimize seek time

Seek time \approx seek distance

Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

Disk Scheduling (Cont.)

Several algorithms exist to schedule the servicing of disk I/O requests.

We illustrate them with a request queue (0-199 cylinders).

98, 183, 37, 122, 14, 124, 65, 67

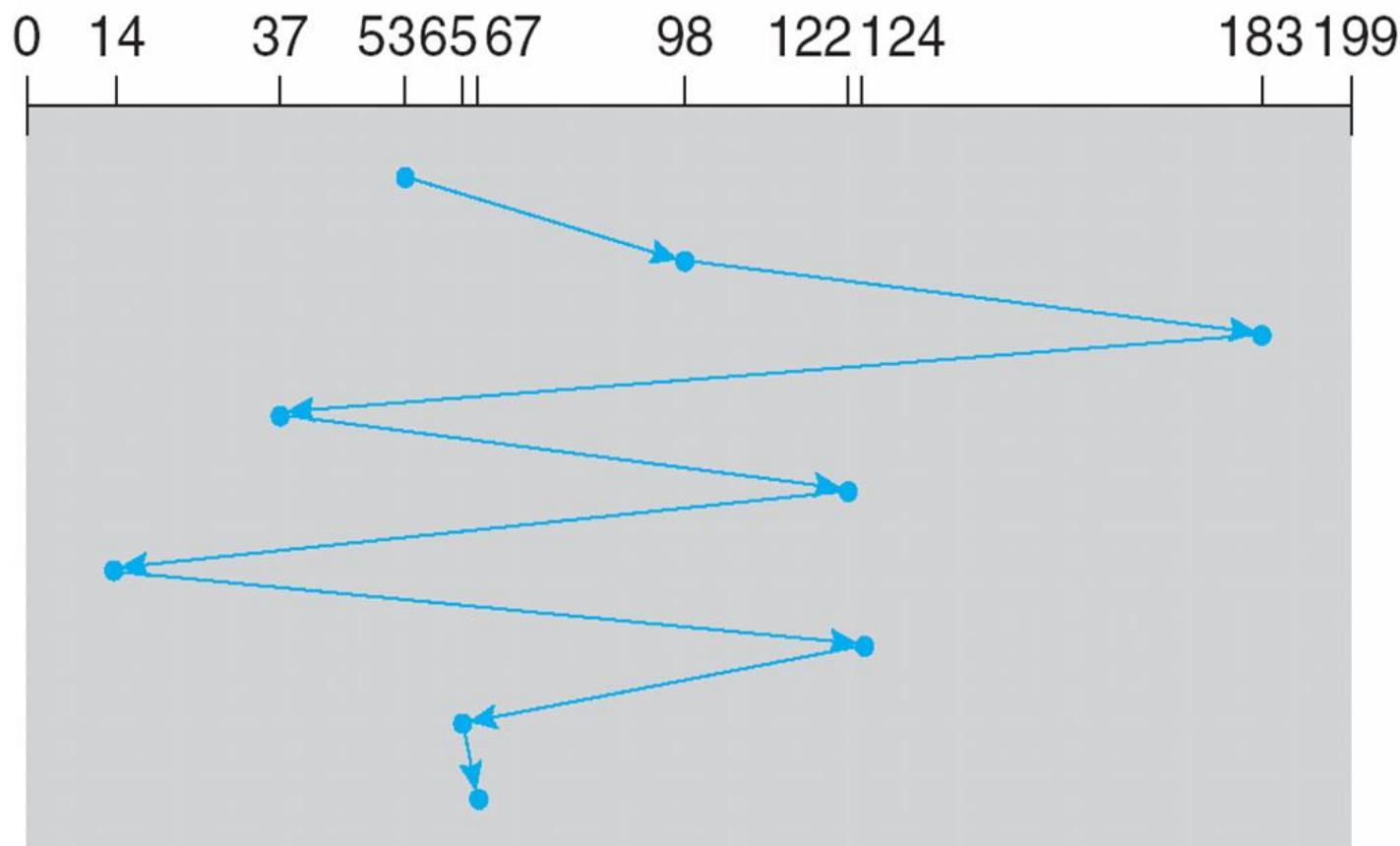
Head pointer 53

FCFS (First Come First Service)

Illustration shows total head movement of **640 cylinders**.

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



SSTF (Shortest Seek Time First)

Selects the request with the **minimum seek time** from the current head position.

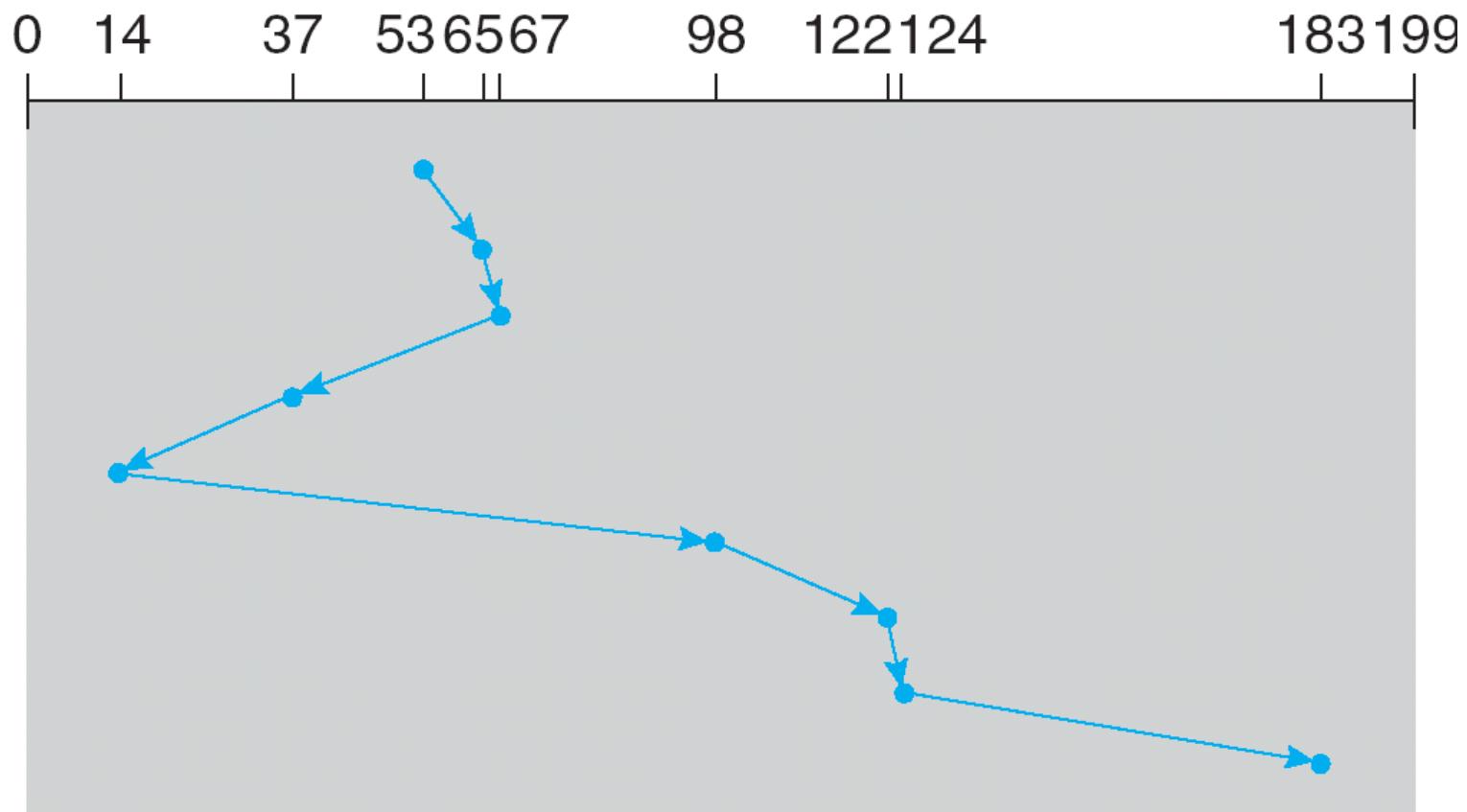
SSTF scheduling is a form of SJF scheduling; may cause **starvation** of some requests.

Illustration shows total head movement of **236 cylinders**.

SSTF (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



SCAN

The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where **the head movement is reversed and servicing continues.**

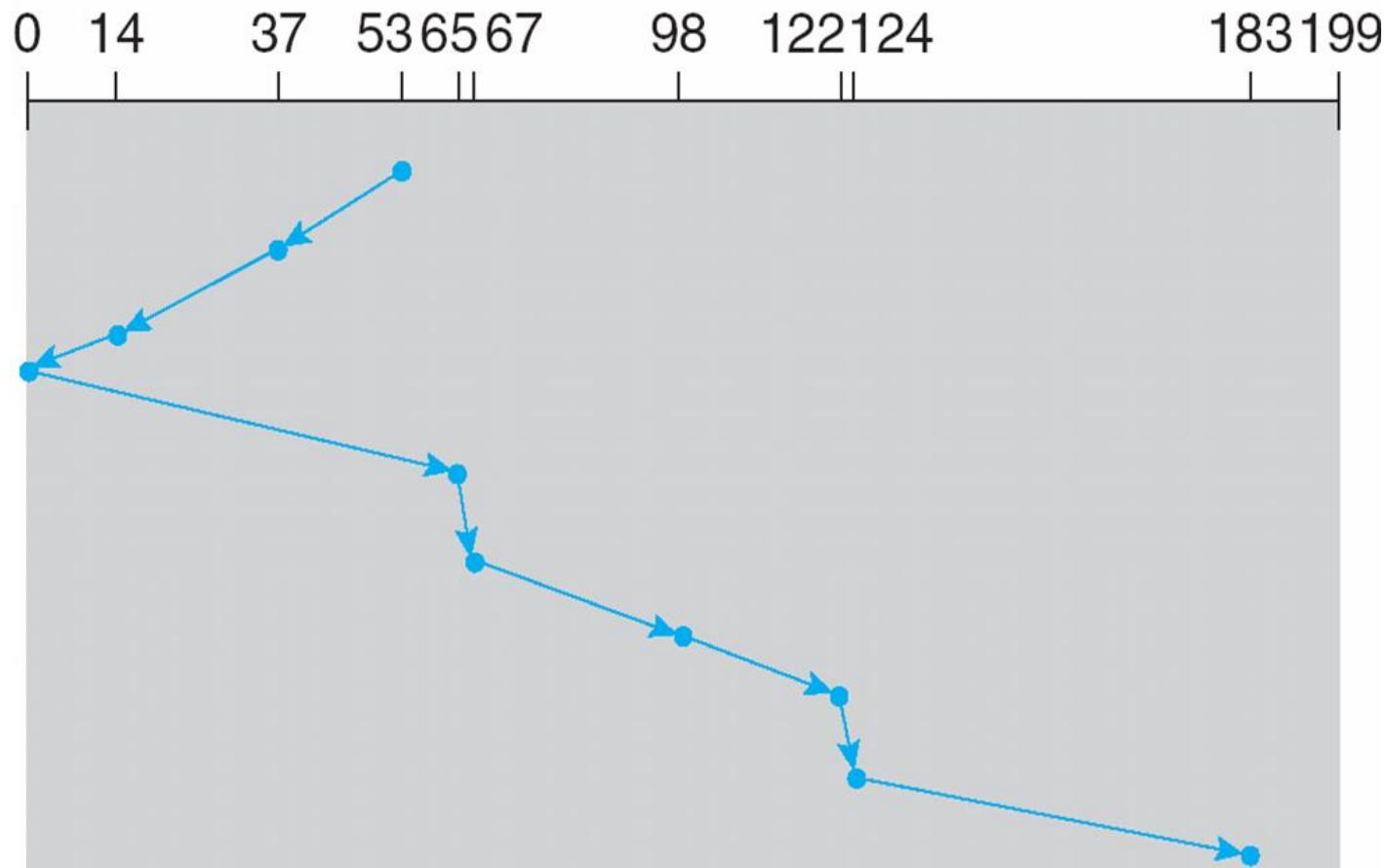
Sometimes called the **elevator algorithm**.

Illustration shows total head movement of **208 cylinders**.

SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



C-SCAN

Provides a **more uniform wait time than SCAN**.

The head moves from one end of the disk to the other.
servicing requests as it goes.

When it reaches the other end, however, it immediately
returns to the beginning of the disk, **without servicing**
any requests on the return trip.

Treats the cylinders as a circular list that wraps around
from the last cylinder to the first one.

C-SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



C-LOOK (or LOOK)

Versions of SACN and C-SCAN

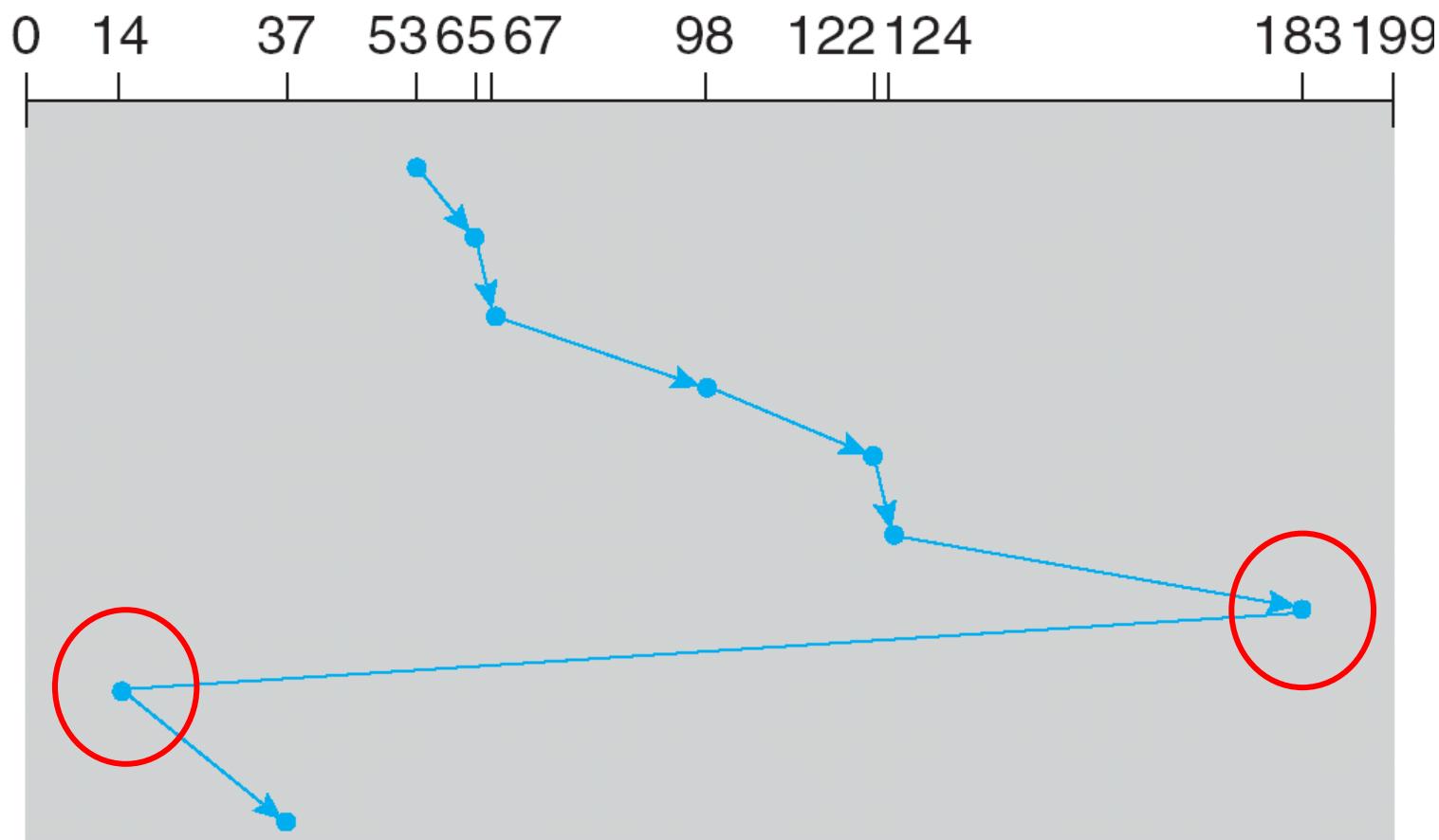
Arm only goes **as far as the last request in each direction**, then reverses direction immediately, without first going all the way to the end of the disk.

They **look** for a request before continuing to move in a given direction.

C-LOOK (Cont.)

queue 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



Selecting a Disk-Scheduling Algorithm

SSTF is common and has a natural appeal

SCAN and C-SCAN perform better for systems that place a heavy load on the disk.

Performance depends on the number and types of requests.

Requests for disk service can be influenced by the file-allocation method.

The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary.

Either SSTF or LOOK is a reasonable choice for the default algorithm.

12.5 Disk Management

Low-level formatting, or **physical formatting** — Dividing a disk into sectors that the disk controller can read and write.

To use a disk to hold files, the operating system still needs to record its own data structures on the disk.

Partition the disk into one or more groups of cylinders.

Logical formatting or “making a file system”.

Boot block initializes system.

The bootstrap is stored in ROM.

Bootstrap loader program.

Methods such as **sector sparing** used to handle bad blocks.

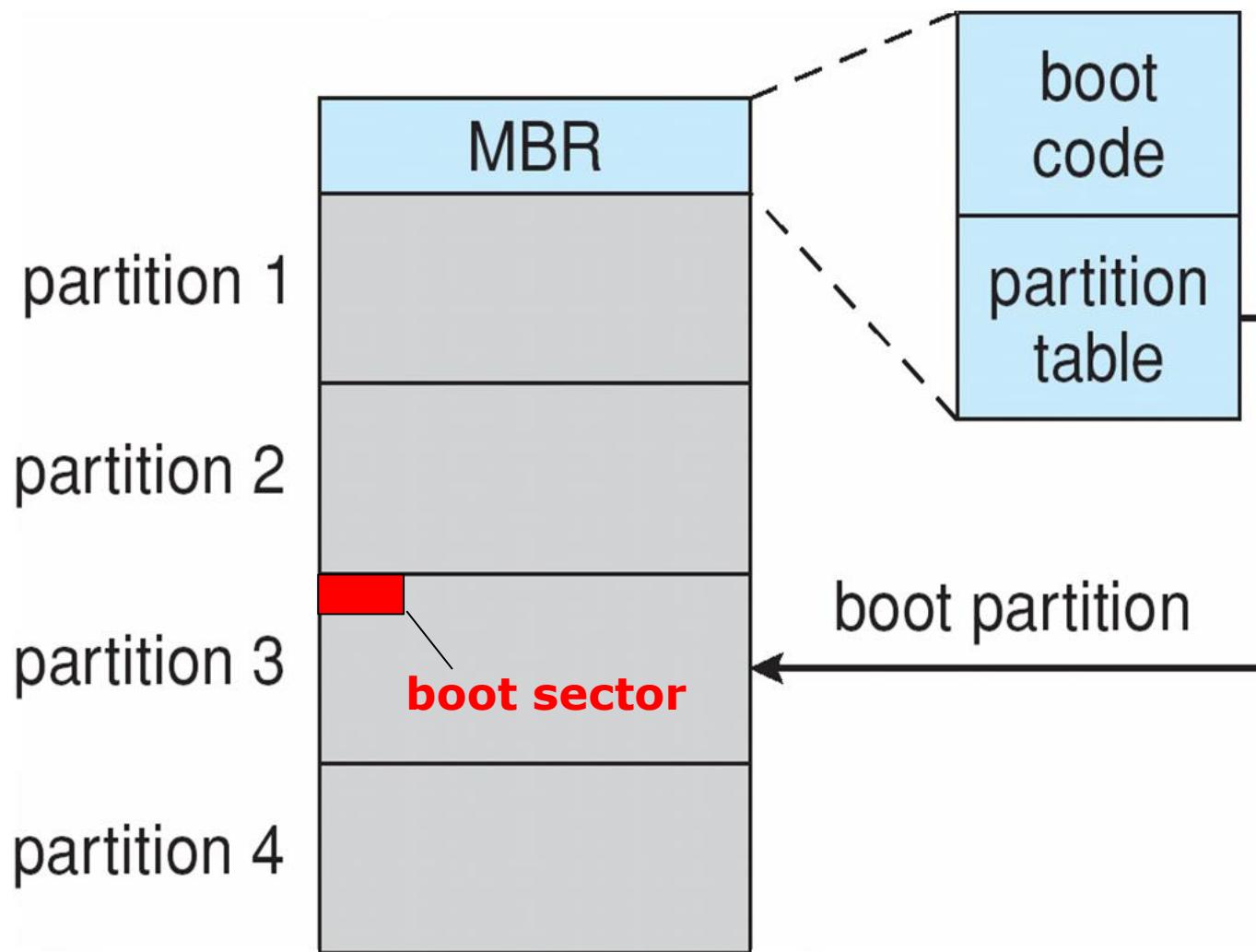
Booting from a Disk in Windows 2000

The Windows 2000 system places its boot code in the first sector on the hard disk (**Master boot record, MBR**).

Windows 2000 allows a hard disk to be divided into one or more partitions; one partition, identified as the **boot partition**, contains the OS and device drivers.

Once the system identifies the boot partition, it reads the first sector from that partition (which is called the **boot sector**) and continues with the remainder of the boot process.

Booting from a Disk in Windows 2000



12.6 Swap-Space Management

Swap-space — Virtual memory uses disk space as an extension of main memory.

Swap-space can be carved out of the normal file system or, more commonly, it can be in a separate disk partition.

Swap-space management

4.3BSD allocates swap space when process starts; holds **text segment** (the program) and **data segment**.

Kernel uses **swap maps** to track swap-space use.

Solaris 2 allocates swap space **only when a page is forced out of physical memory**, not when the virtual memory page is first created.

Swapping on Linux Systems

Linux allows one or more **swap areas** to be established.

A **swap area** may be either a swap file on a regular file system or a raw-swap-space partition.

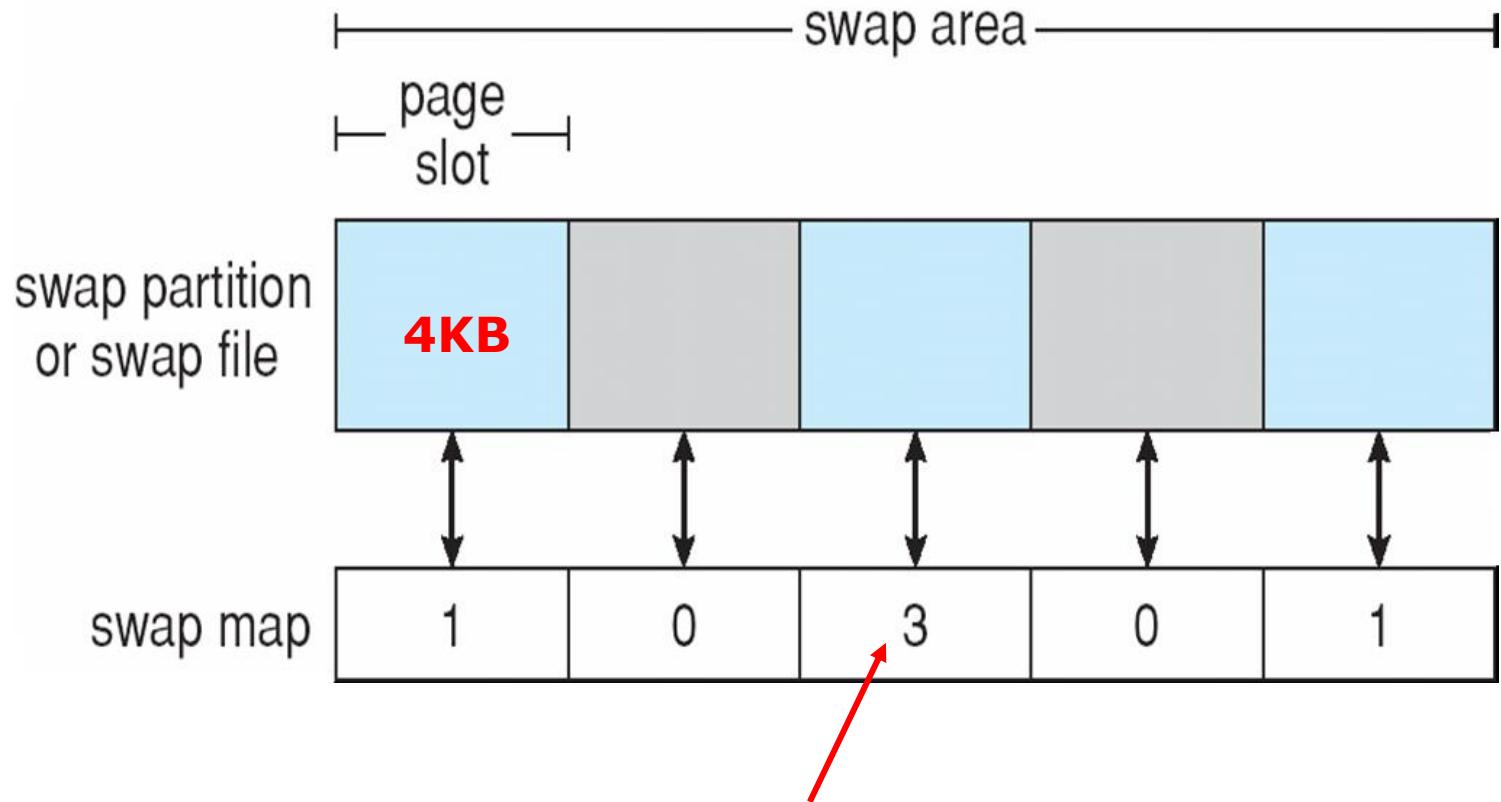
Each swap area consists of a series **of 4KB page slots**, which are used to hold swapped pages.

Each swap area is associated with a **swap map**.

Counter = 0, the corresponding page slot is available.

Counter >0, the page slot is occupied by a swapped page. The value of the counter indicates **the number of mappings to the swapped page**. Counter =3, the swapped page is storing a region of memory shared by three processes.

Data Structures for Swapping on Linux Systems



The value of the counter indicates the number of mappings to the swapped page

12.7 RAID Structure

RAID (Redundant Arrays of Independent Disks) – multiple disk drives provides **reliability** via **redundancy**.

RAID is arranged into seven different levels.

With multiple disks, we can improve the transfer rate by striping data across the disks.

Data striping consists of splitting the bits of each byte across multiple disks – **bit level striping**

Block-level striping consists of splitting the blocks of each file across multiple disks.

RAID (cont)

Several improvements in disk-use techniques involve the use of multiple disks working cooperatively.

Disk striping uses a group of disks as one storage unit.

RAID schemes improve performance and improve the reliability of the storage system by storing redundant data.

Mirroring or shadowing keeps duplicate of each disk.

Block interleaved parity uses much less redundancy.

RAID (cont)

RAID level 0: RAID level 0 refers to disk arrays with **striping at the level of blocks** but without any redundancy.



(a) RAID 0: non-redundant striping.

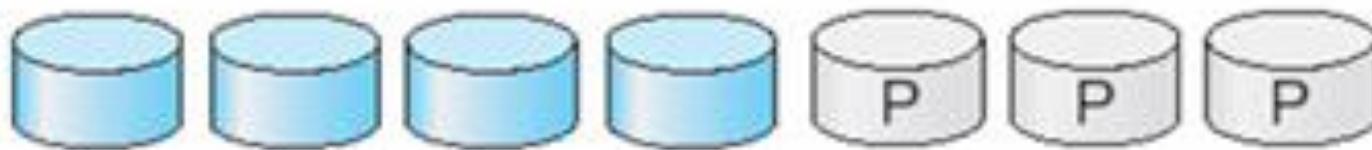
RAID level 1: RAID level 1 refers to **disk mirroring**.



(b) RAID 1: mirrored disks.

RAID (cont)

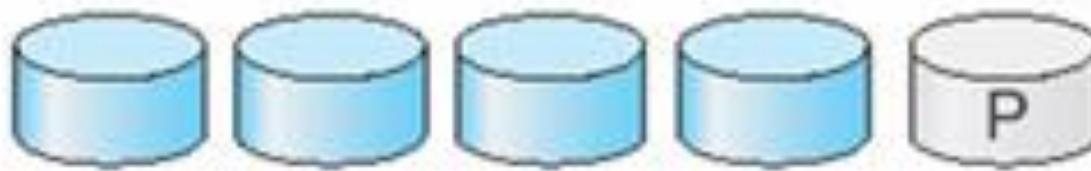
RAID level 2: RAID level 2 also known as **memory-style error-correcting-code (ECC) organization**. The **parity bits** are used. The disks labeled P store the error-correction bits.



(c) RAID 2: memory-style error-correcting codes.

RAID (cont)

RAID level 3: RAID level 3 refers to **bit-interleaved parity organization** improves on level 2 by taking into account the fact that, unlike memory systems, **disk controllers can detect whether a sector has been read correctly**, so a single parity bit can be used for error correction and for detection.



(d) RAID 3: bit-interleaved parity.

RAID (cont)

RAID level 4: RAID level 0 refers to **block-interleaved parity organization**, uses block-level striping, as in RAID 0, and also keeps a parity block on a separate disk for corresponding blocks from N other disks.



(e) RAID 4: block-interleaved parity.

RAID (cont)

RAID level 5: RAID level 5 refers to **block-interleaved distributed parity**, differs from level 4 by spreading data and parity among all $N+1$ disks, rather than storing data in N disks and parity in one disk.

For each block, one of the disks stores the parity, and the others store data.

With an array of five disks, the parity for the n th block is stored in disk $(n \bmod 5)+1$; the n th blocks of the other four disks store actual data for the block.

A parity block cannot store parity for the blocks in the same disk.

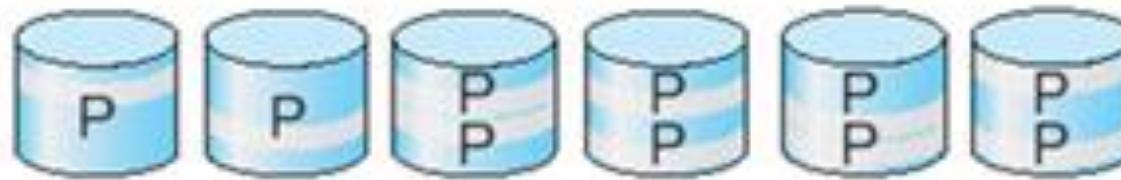


(f) RAID 5: block-interleaved distributed parity.

RAID (cont)

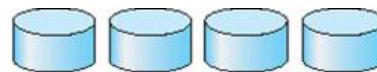
RAID level 6, **P+Q redundancy scheme**, is much like RAID 5 but stores **extra redundant information** to guard against multiple disk failures.

Instead of parity, **error-correcting codes** such as **Reed-Solomon codes** are used. 2 bits of redundant data are stored for every 4 bits of data – compared with 1 parity bit in level 5 – and the system can **tolerate two disk failures**.

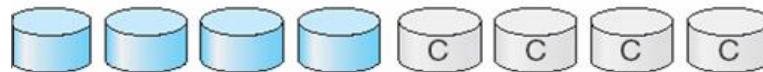


(g) RAID 6: P + Q redundancy.

RAID Levels



(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



(c) RAID 2: memory-style error-correcting codes.



(d) RAID 3: bit-interleaved parity.



(e) RAID 4: block-interleaved parity.



(f) RAID 5: block-interleaved distributed parity.



(g) RAID 6: P + Q redundancy.

RAID (cont)

RAID 0 provides the **performance**,

RAID 1 provides the **reliability**.

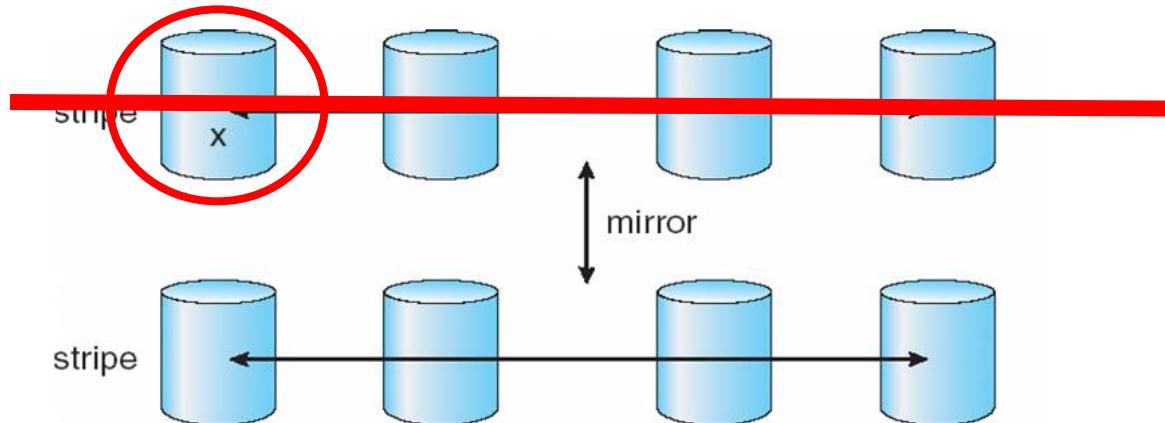
RAID 0+1: A set of disks are striped, and then the stripe is mirrored to another, equivalent strip

RAID 1+0: Disks are mirrored in pairs and then the resulting mirrored pairs are striped.

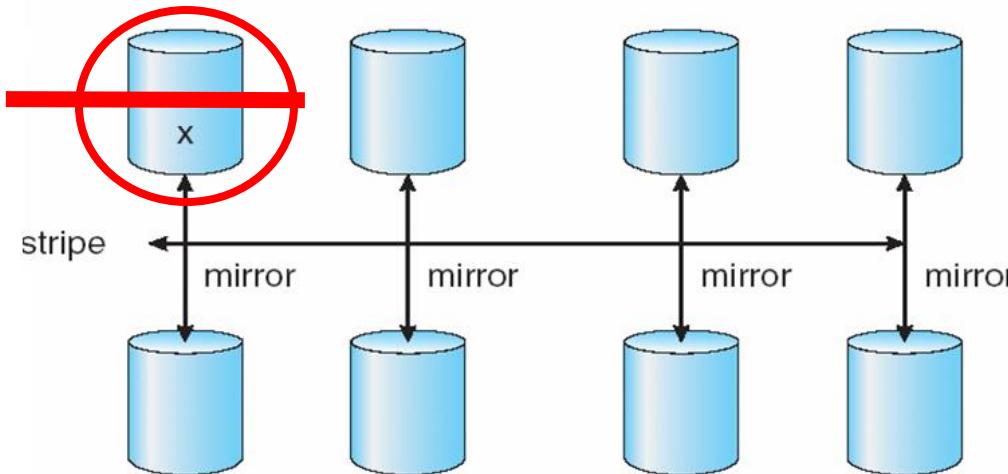
RAID 1+0 has some theoretical advantages over RAID 0+1.

For example, if a single disk fails in RAID 0+1, an entire strip is inaccessible, leaving only the other strip available. With a failure in RAID 1+0, a single disk is unavailable, but the disk that mirrors it is still available, as are all the rest of the disks.

RAID (0 + 1) and (1 + 0)



a) RAID 0 + 1 with a single disk failure.



b) RAID 1 + 0 with a single disk failure.

12.8 Stable-Storage Implementation

Write-ahead log scheme requires **stable storage**.

To implement stable storage:

Replicate information on more than one nonvolatile storage media with independent failure modes.

Update information in a **controlled manner** to ensure that we can recover the stable data after any failure during data transfer or recovery.

12.9 Tertiary Storage Structure

Low cost is the defining characteristic of tertiary storage.

Generally, tertiary storage is built using **removable media**. Common examples of removable media are floppy disks and CD-ROMs; other types are available.

Removable Disks

Floppy disk — thin flexible disk coated with magnetic material, enclosed in a protective plastic case.

Most floppies hold about 1 MB; similar technology is used for removable disks that hold more than 1 GB.

Removable magnetic disks can be nearly as fast as hard disks, but they are at a greater risk of damage from exposure.

Removable Disks (Cont.)

A **magneto-optic disk** records data on a rigid platter coated with magnetic material.

Laser heat is used to amplify a large, weak magnetic field to record a bit.

Laser light is also used to read data (Kerr effect).

The magneto-optic head flies much farther from the disk surface than a magnetic disk head, and the magnetic material is covered with a protective layer of plastic or glass; resistant to head crashes.

Optical disks do not use magnetism; they employ special materials that are altered by laser light.

WORM Disks

The data on read-write disks can be modified over and over.

WORM (“Write Once, Read Many Times”) disks can be written only once.

Thin aluminum film sandwiched between two glass or plastic platters.

To write a bit, the drive uses a **laser light to burn a small hole through the aluminum; information can be destroyed by not altered.**

Very durable and reliable.

Read Only disks, such as CD-ROM and DVD, come from the factory with the data pre-recorded.

Tapes

Compared to a disk, a tape is less expensive and holds more data, but **random access is much slower**.

Tape is an economical medium for purposes that **do not require fast random access**, e.g., backup copies of disk data, holding huge volumes of data.

Large tape installations typically use robotic tape changers that move tapes between tape drives and storage slots in a tape library.

stacker – library that holds a few tapes

silo – library that holds thousands of tapes

A disk-resident file can be *archived* to tape for low cost storage; the computer can *stage* it back into disk storage for active use.

Operating System Support

Major OS jobs are to manage physical devices and to present a virtual machine abstraction to applications

For hard disks, the OS provides two abstraction:

Raw device – an array of data blocks.

File system – the OS queues and schedules the interleaved requests from several applications.

Application Interface

Most OSs handle **removable disks almost exactly like fixed disks** — a new cartridge is formatted and **an empty file system is generated on the disk**.

Tapes are presented as a raw storage medium, i.e., and application does not open a file on the tape, it opens the whole tape drive as a raw device.

Usually the tape drive is reserved for the exclusive use of that application.

Since the OS does not provide file system services, the **application must decide how to use the array of blocks**.

Since every application makes up its own rules for how to organize a tape, a tape full of data can generally **only be used by the program that created it**.

Tape Drives

The basic operations for a tape drive differ from those of a disk drive.

locate positions the tape to a specific logical block, not an entire track (corresponds to seek).

The **read position** operation returns the logical block number where the tape head is.

The **space** operation enables relative motion.

Tape drives are “**append-only devices**”; updating a block in the middle of the tape also effectively erases everything beyond that block.

An EOT mark is placed after a block that is written.

File Naming

The issue of naming files on removable media is especially difficult when we want to write data on a removable cartridge on one computer, and then use the cartridge in another computer.

Contemporary OSs generally leave the name space problem unsolved for removable media, and depend on applications and users to figure out how to access and interpret the data.

Some kinds of removable media (e.g., CDs) are so well standardized that all computers use them the same way.

Hierarchical Storage Management (HSM)

A *hierarchical storage system* extends the storage hierarchy beyond primary memory and secondary storage to incorporate *tertiary storage* — usually implemented as a **jukebox** of tapes or removable disks.

Usually incorporate tertiary storage by extending the file system.

Small and frequently used files remain on disk.

Large, old, inactive files are archived to the jukebox.

HSM is usually found in supercomputing centers and other large installations that have enormous volumes of data.

Speed

Two aspects of speed in tertiary storage are **bandwidth** and **latency**.

Bandwidth is measured in bytes per second.

Sustained bandwidth – average data rate during a large transfer; **# of bytes/transfer time**.

Data rate when the data stream is actually flowing.

Effective bandwidth – average over the entire I/O time, including seek or locate, and cartridge switching.
Drive's overall data rate.

Speed (Cont.)

Access latency – amount of time needed to locate data.

Access time for a **disk** – move the arm to the selected cylinder and wait for the rotational latency; **< 35 milliseconds**.

Access on **tape** requires winding the tape reels until the selected block reaches the tape head; **tens or hundreds of seconds**.

Generally say that random access within a tape cartridge is about **a thousand times slower than random access on disk**.

The low cost of tertiary storage is a result of having **many cheap cartridges share a few expensive drives**.

A removable library is best devoted to the storage of infrequently used data, because the library can only satisfy a relatively small number of I/O requests per hour.

Reliability

A **fixed disk drive** is likely to be **more reliable** than a removable disk or tape drive.

An **optical cartridge** is likely to be more reliable than a magnetic disk or tape.

A **head crash in a fixed hard disk** generally destroys the data, whereas the failure of a tape drive or optical disk drive often leaves the data cartridge unharmed.

Cost

Main memory is much more expensive than disk storage

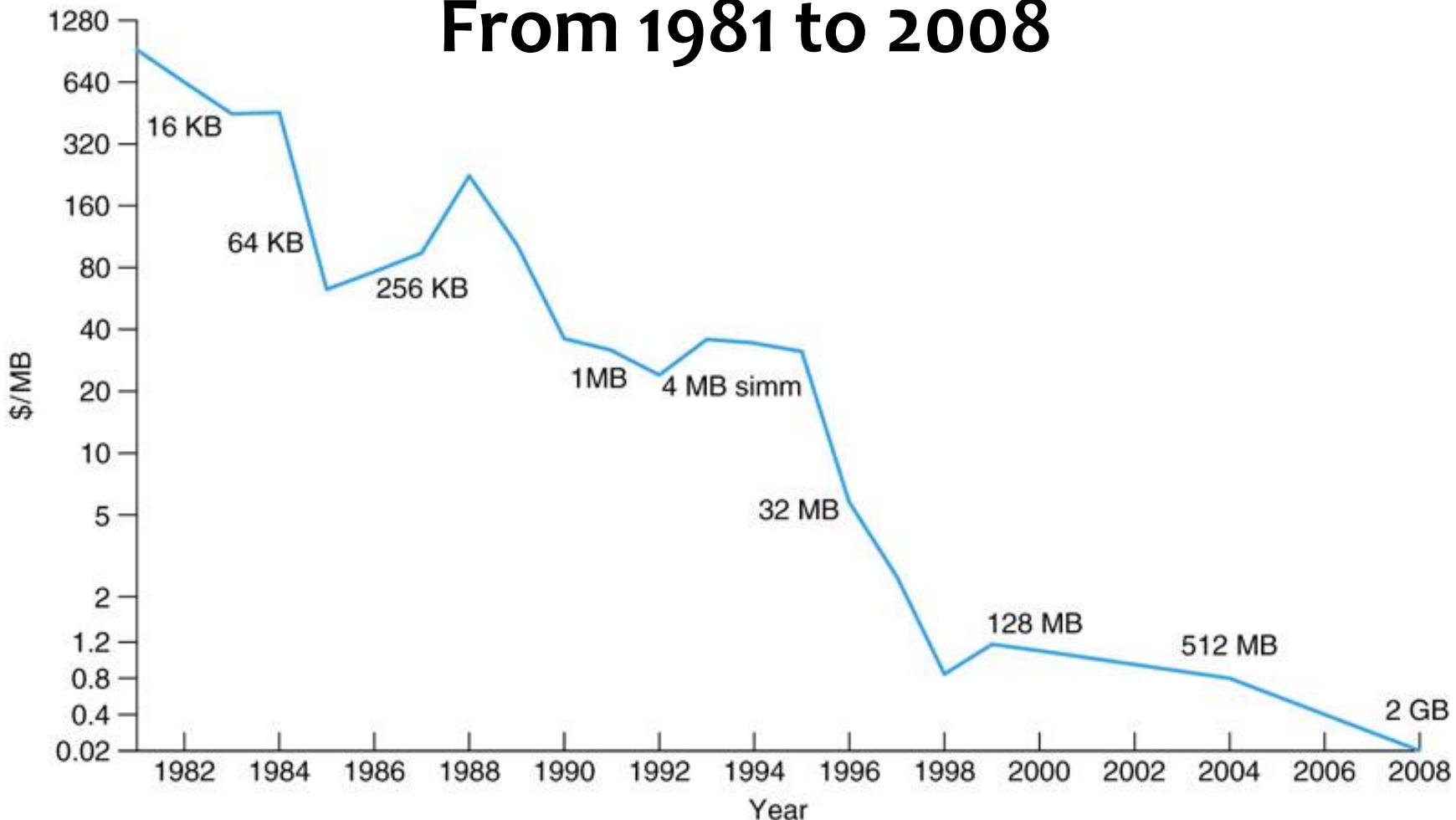
The cost per megabyte of hard disk storage is competitive with magnetic tape **if only one tape is used per drive.**

The cheapest tape drives and the cheapest disk drives have had about the same storage capacity over the years.

Tertiary storage gives a cost savings only when the number of cartridges is considerably larger than the number of drives.

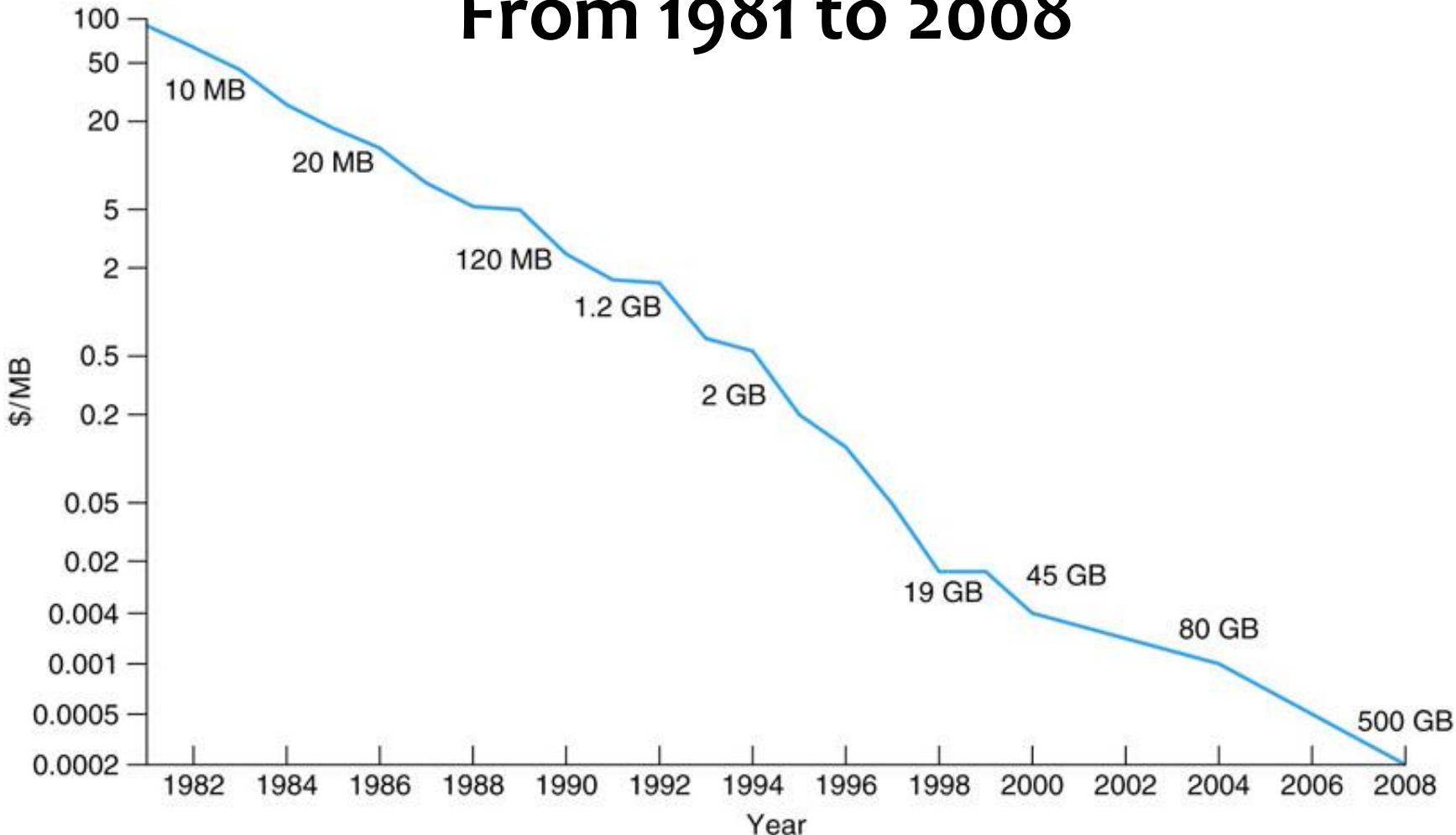
Price per Megabyte of DRAM

From 1981 to 2008



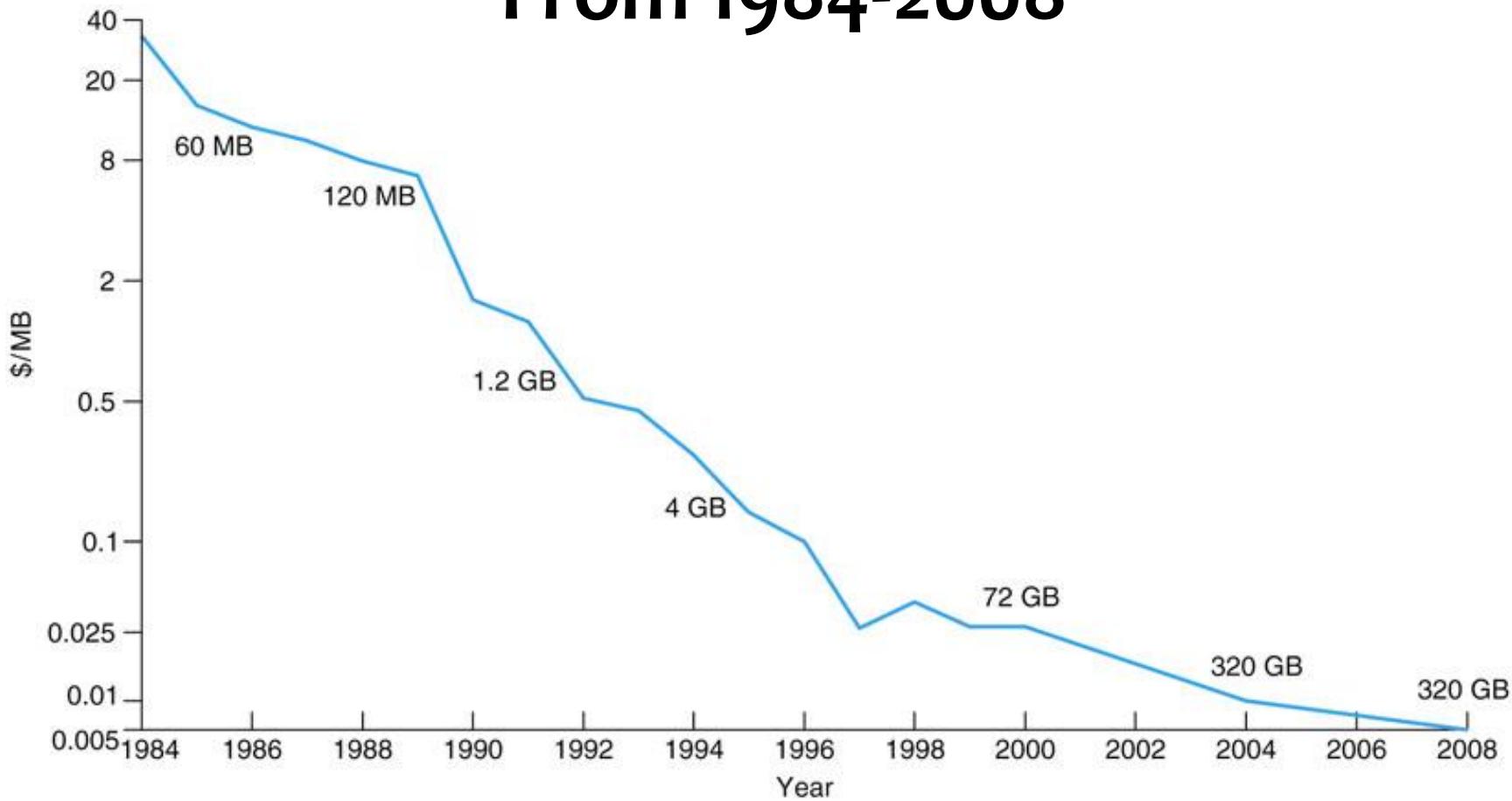
Price per Megabyte of Magnetic Hard Disk

From 1981 to 2008



Price per Megabyte of a Tape Drive

From 1984-2008



End of Chapter 12

