UNIT-1 INTRODUCTIONS TO OPERATING SYSTEM

1.1WHAT IS AN OS?

- The software that controls the hardware
- The layer of software.
- An operating system is software that enables applications to interact with a computer's hardware.
- The operating system is a "black box" between the applications and the hardware they run on that ensures the proper result, given appropriate inputs.
- Operating system are primarily resource managers-they manage hardware, including processor, memory, input/output devices and communication devices.
- They manage applications and other software abstractions

OPERATING SYSTEM COMPONENTS AND GOALS

1.2.1 CORE OPERATING SYSTEM COMPONENTS

- A user interface with the operating system via one or more user applications. A special application called a **SHELL OR COMMAND INTERPRETER.**
- SHELL are implemented as
    i. **Text interface-enable user to issue command from a keyboard**
    ii. **GUI (Graphical user interface)-allow user to point & click, drag and drop.**
    iii. **Command prompt window-enable user through commands.**

The software that contains the core components of the operating system is referred to as the KERNEL.

Os core components include:

1. **Process scheduler: which determines when and for how long a process executes on a processor.**
2. **Memory manager: which determines when and how memory is allocated to processes and what to do when main memory becomes fail.**
3. **I/O manager: which services input and output requests fromand to hardware devices respectively.**
4. **Interprocess communication(IPC) manager: which allows processes to communicate with one another.**

5. **File system manager: which organizes named collections of data on storage devices and provides and interface for accessing data on those devices.**
6. **Multiprogrammed environment: which multiple applications can execute concurrently. It determines which processor executes a process and for how long that process executes.**
7. **Program components which executes independently but perform their work in a common memory space are called THREADS.**
8. **Disk scheduler: responsible for reordering disk I/O request to maximize performance and minimize the amount of time a process wait for disk I/O. Redundant Array of Independent (RAID) system attempts to reduce the time a process waits for disk I/O by using multiple disk at once to service I/Orequest.**

OPERATING SYSTEM GOALS

- Efficiency-high throughput and low turnaround time
- Robustness-fault tolerance & reliable
- Scalability- adding resources in order to improve degree of multiprogramming
- Extensibility-adapt to new technonogies
- Portability- operate on many hardware configurations
- Security-prevent software and user from accessing services & resources without authorization.
- Interactivity- application to respond quickly to user
- Usability-potential to serve a user

OPERATING SYSTEM ARCHITECTURE

(a) Monolithic architecture
(b) Layered architecture
(c) Microkernel architecture
(d) Networked and Distributed architecture

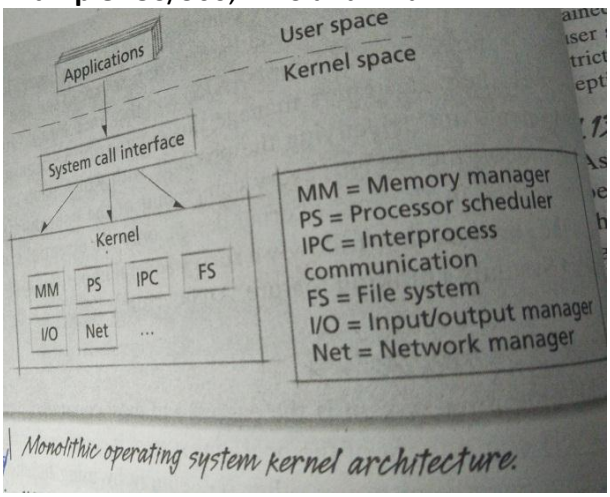
a) Monolithic architecture

- It is the earliest and most common operating system architecture.
- Every component of the operating system is contained in the kernel and can directly communicate with any other.

Advantage: Highly efficient (Direct intercommunication between components)

Disadvantage: it is difficult to isolate the source of bugs and other errors.

Example: OS/360, VMS and Linux



Monolithic operating system kernel architecture.

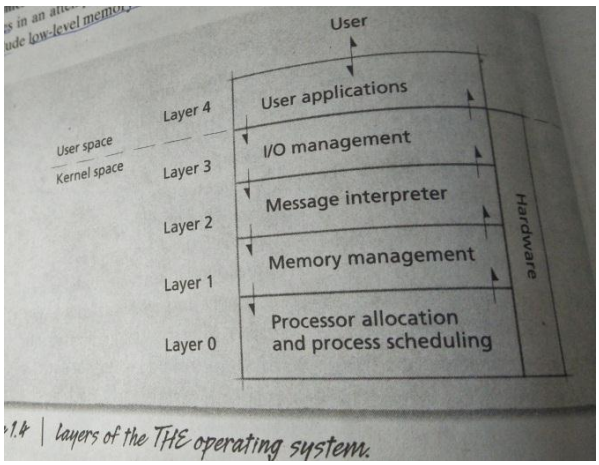
b) Layered architecture

- The layered approach to operating system attempts to address this issue by grouping components that perform similar functions into layers.
- Each layer communicates exclusively with those immediately above and below it.
- Lower level layers provide serices to higher level ones using an interface

  <u>Advantage:</u>
- More modular
- Components reused throughout the system
- Modified without requiring any modification to other layers.
- Simplify validation, debugging and modification.

  Disadvantage: performance degrades.

  Example: THE, (Windows XP, Linux implement some level of layering.



layers of the THE operating system.

c) Microkernel architecture

- Provides only a small number of services in an attempt to keep the kernel **small and scalable.**
- **The services include low level memory management, interprocess communication and basic process synchronization to enable processes to cooperate.**
- **Most operating system components such as process management, networking, file system interaction and device management execute outside the kernel with a lower privilege level.**

  Advantage:

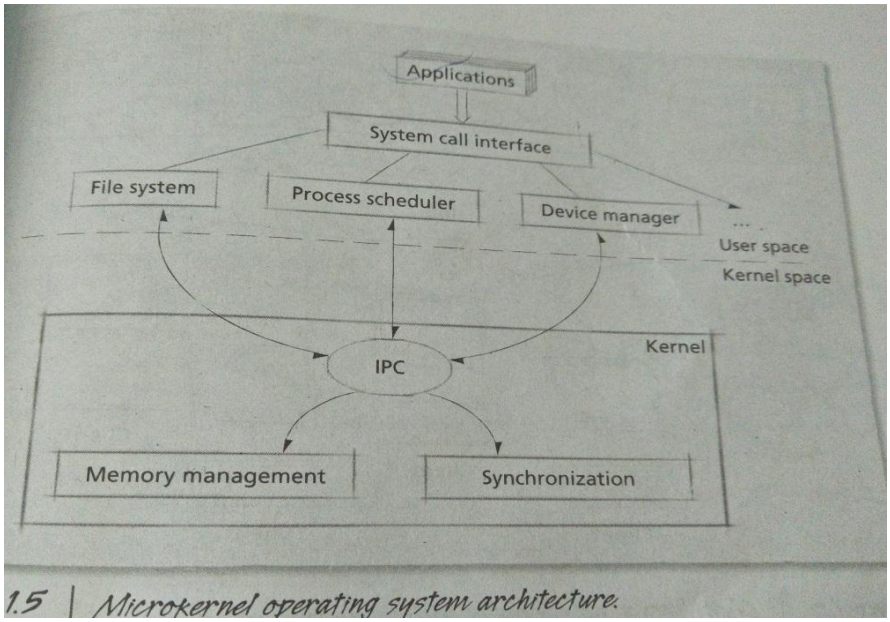
  High degree of modularity

  Making them extensible, Portable and scalable.

  One or more components can fail without causing the operating system to fail

  Disadvantage: performance degrade

  Example: Linux and Windows XP



1.5 | Microkernel operating system architecture.

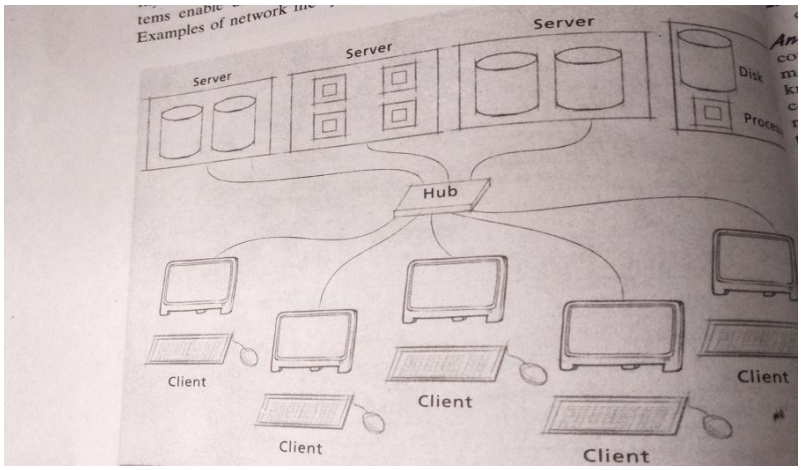
d) <u>NETWORKED AND DISTRIBUTED OPERATING SYSTEM</u>

- It enables its processes to access resources that reside on other independent computers on a network.
- The structure of the networked and distributed operating system is often based on the client/server model.
- In a networked environment, a process can execute on the computer on which it is created or on another computer on the network.
- Networked file system are an important component of networked operating system.

- In lower level network file system, user acquires resources on another machine by explicitly connecting to that machine and retrieving files.
- Higher-level network file systems enable user to access remote files as if they were on the local system.
  Example: Sun's Network File System (NFS) and CMU's Andrew and coda file system.

Distributed operating system

- It is a single operating system that manages resources on more than one computer system.
- Distributed operating systems are often difficult to implement and require complicated algorithms to enable processes to communicate and share data.
  Example: MIT's Chord operating system and Amoeba operating system



CHAPTER-2
PROCESS CONCEPTS

INTRODUCTION

DEFINITION OF PROCESS:

- **A Program in execution**
- **A asynchronous activity**
- **The "animated spirit" of a procedure**
- **The "locus of control" of a procedure in execution**

The data structure of the process called a "process descriptor" or a "process control block".

There are two key concepts

1. **A process is an "entity" – each process has its own address space, which consists of a text region, data region and stack region.**

Text region: stores the code that the processor execute.
Data region: stores variable and dynamically allocated memory that the process uses during execution.
Stack region: stores instruction and local variables for active procedure calls.

2. **A process is a "program in execution".**

1.6 PROCESS STATES: LIFE CYCLE OF A PROCESS

- **A Process is said to be RUNNING STATE, if it is executing on a processor**
- **A Process is said to be READY STATE, if it is execute on a processor**
- **A Process is said to be BLOCKED STATE, if it is waiting for some event to happen**

The operating system maintains a

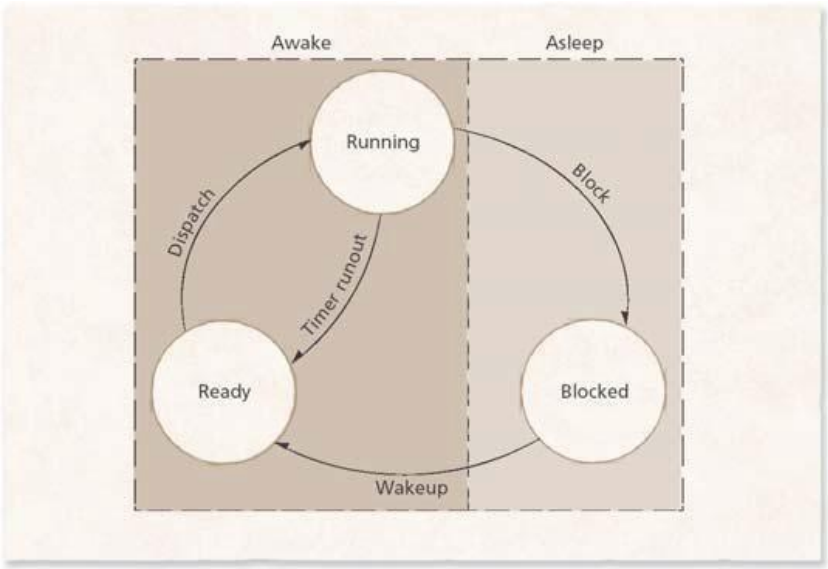
- **Ready list for ready process**
- **Blocked list for blocked process**

1.7 PROCESS MANAGEMENT

Operating systems provide fundamental services to processes including

–Creating processes
– Destroying processes
– Suspending processes
– Resuming processes
– Changing a process's priority
– Blocking processes
– Waking up processes
– Dispatching processes
– Interprocess communication (IPC)

Process States and State Transitions



Process Control Blocks (Pcbs)/Process Descriptors

PCBs maintain information that the OS needs to manage the process
– Typically include information such as
• Process identification number (PID)
• Process state

• Program counter
• Scheduling priority
• Credentials
• A pointer to the process's parent process
• Pointers to the process's child processes
• Pointers to locate the process's data and instructions in memory
• Pointers to allocated resources

Process table

– The OS maintains pointers to each process's PCB in a systemwide or per-user process table

– Allows for quick access to PCBs

– When a process is terminated, the OS removes the process from the process table and frees all of the process's resources

Process Operations

A process may spawn a new process

– The creating process is called the parent process

– The created process is called the child process

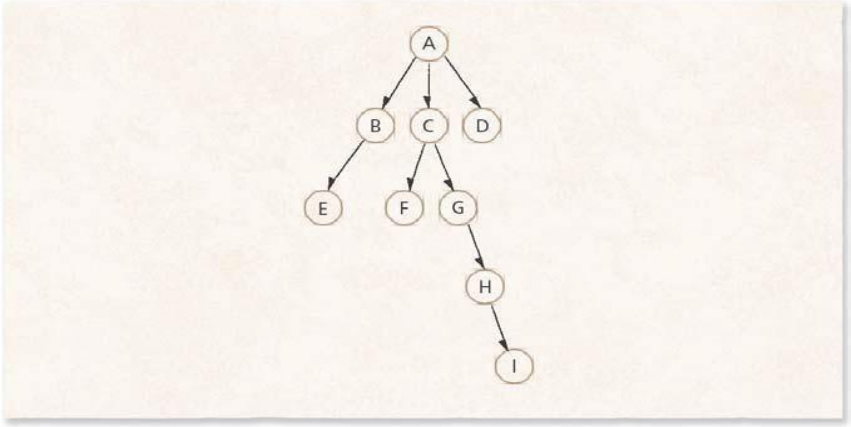
– Exactly one parent process creates a child

– When a parent process is destroyed, operating systems typically respond in one of two ways:

• Destroy all child processes of that parent

• Allow child processes to proceed independently of their parents

Fig. Process creation hierarchy.



Suspend and Resume

• Suspending a process

– Indefinitely removes it from contention for time on a processor without being destroyed

– Useful for detecting security threats and for software debugging purposes

– A suspension may be initiated by the process being suspended or by another process

– A suspended process must be resumed by another process

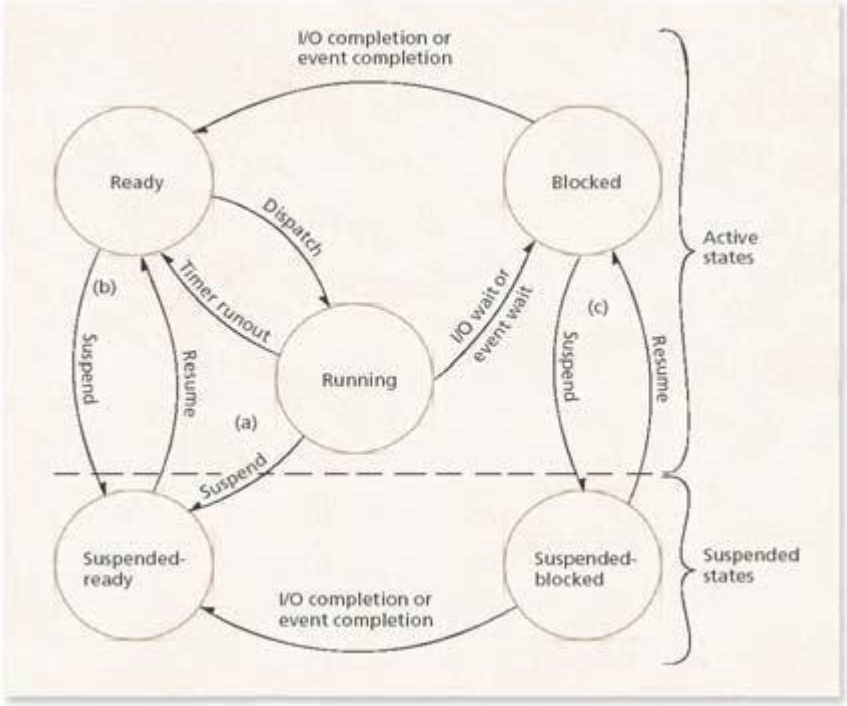
– Two suspended states:

• *suspendedready*

• *suspendedblocked*

.

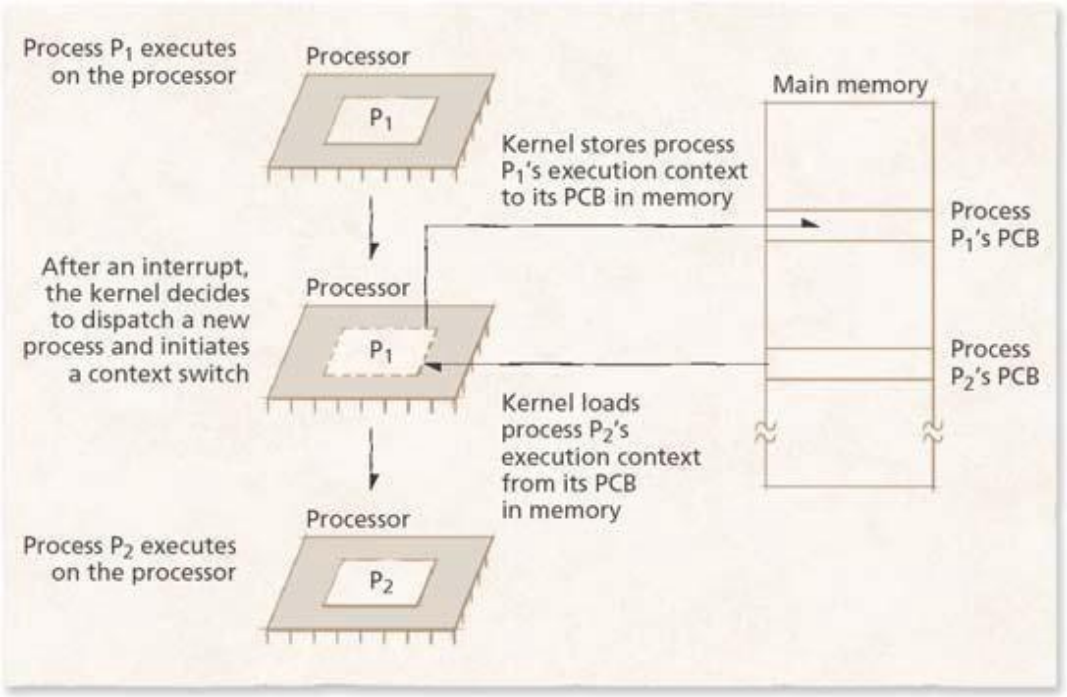
Process state transitions with suspend and resume.



Context Switching

– Performed by the OS to stop executing a *running* process and begin executing a previously *ready* process
– Save the execution context of the *running* process to its PCB
– Load the *ready* process's execution context from its PCB
– Must be transparent to processes
– Require the processor to not perform any "useful" computation

• OS must therefore minimize context-switching time
– Performed in hardware by some architectures



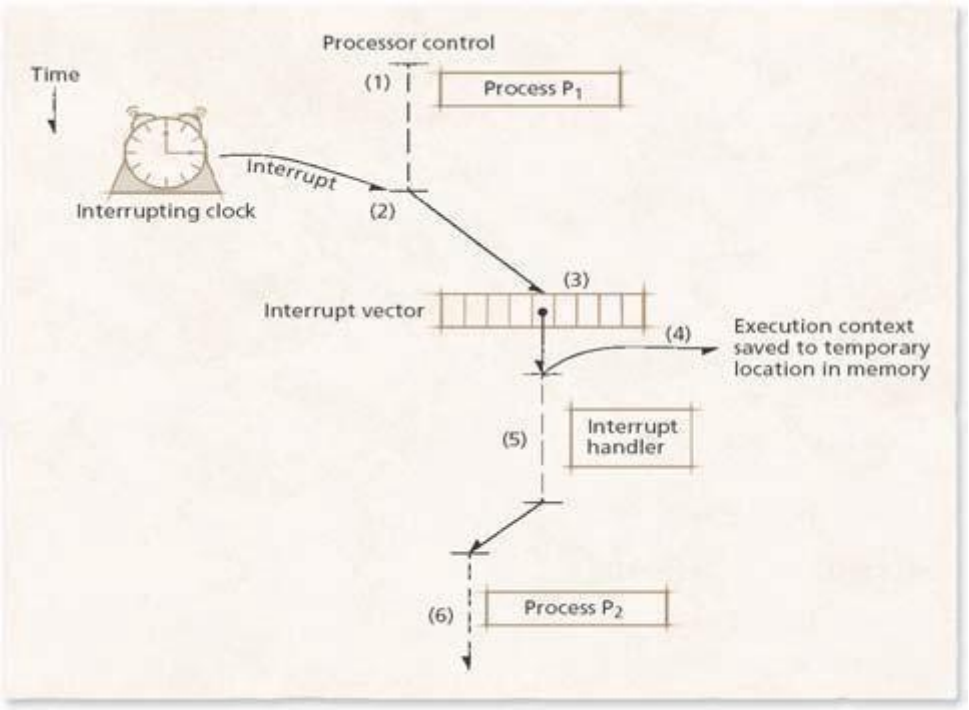
Interrupts

• Interrupts enable software to respond to signals from hardware
  – May be initiated by a running process
• Interrupt is called a trap
• Synchronous with the operation of the process
• For example, dividing by zero or referencing protected memory
  – May be initiated by some event that may or may not be related to the running process
• Asynchronous with the operation of the process
• For example, a key is pressed on a keyboard or a mouse is moved
  – Low overhead
• Polling is an alternative approach
  – Processor repeatedly requests the status of each device
  – Increases in overhead as the complexity of the system increases

Interrupt Processing

• Handling interrupts
  – After receiving an interrupt, the processor completes execution of the current instruction, then pauses the current process
  – The processor will then execute one of the kernel's interrupthandling functions
  – The interrupt handler determines how the system should respond
  – Interrupt handlers are stored in an array of pointers called the interrupt vector
  – After the interrupt handler completes, the interrupted process is restored and executed or the next process is executed

Figure Handling interrupts



Interrupt Classes

• Supported interrupts depend on a system's architecture
  – The IA-32 specification distinguishes between two types of signals a processor may receive:

• Interrupts
  – Notify the processor that an event has occurred or that an external device's status has changed
  – Generated by devices external to a processor
• Exceptions
  – Indicate that an error has occurred, either in hardware or as a result of a software instruction
  – Classified as faults, traps or aborts

Figure Common interrupt types recognized in the Intel IA-32 architecture.

Interrupt Type	Description of Interrupts in Each Type
I/O	These are initiated by the input/output hardware. They notify a processor that the status of a channel or device has changed. I/O interrupts are caused when an I/O operation completes, for example.
Timer	A system may contain devices that generate interrupts periodically. These interrupts can be used for tasks such as timekeeping and performance monitoring. Timers also enable the operating system to determine if a process's quantum has expired.
Interprocessor interrupts	These interrupts allow one processor to send a message to another in a multiprocessor system.

Exception Class	Description of Exceptions in Each Class
Fault	These are caused by a wide range of problems that may occur as a program's machine-language instructions are executed. These problems include division by zero, data (being operated upon) in the wrong format, attempt to execute an invalid operation code, attempt to reference a memory location beyond the limits of real memory, attempt by a user process to execute a privileged instruction and attempt to reference a protected resource.
Trap	These are generated by exceptions such as overflow (when the value stored by a register exceeds the capacity of the register) and when program control reaches a breakpoint in code.
Abort	This occurs when the processor detects an error from which a process cannot recover. For example, when an exception-handling routine itself causes an exception, the processor may not be able to handle both errors sequentially. This is called a double-fault exception, which terminates the process that initiated it.

Interprocess Communication

• Many operating systems provide mechanisms for interprocess communication (IPC)
  – Processes must communicate with one another in multiprogrammed and networked environments
• For example, a Web browser retrieving data from a distant server
  – Essential for processes that must coordinate activities to achieve a common goal

Signals

• Software interrupts that notify a process that an event has occurred
  – Do not allow processes to specify data to exchange with other processes
  – Processes may catch, ignore or mask a signal
• Catching a signal involves specifying a routine that the OS calls when it delivers the signal
• Ignoring a signal relies on the operating system's default action to handle the signal
• Masking a signal instructs the OS to not deliver signals of that type until the process clears the signal mask

Message Passing

• IPC in distributed systems
  – Transmitted messages can be flawed or lost
• Acknowledgement protocols confirm that transmissions have been properly received
• Timeout mechanisms retransmit messages if acknowledgements are not received
  – Ambiguously named processes lead to incorrect message referencing
• Messages are passed between computers using numbered ports on which processes listen, avoiding this problem
  – Security is a significant problem
• Ensuring authentication

UNIT-II
ASYNCHRONOUS CONCURRENT EXECUTION

2.1 Introduction

• Concurrent execution
- – More than one thread exists in system at once
- – Can execute independently or in cooperation
- – Asynchronous execution
  - • Threads generally independent
  - • Must occasionally communicate or synchronize
  - • Complex and difficult to manage such interactions

2.2 Mutual Exclusion

Problem of two threads accessing data simultaneously
- – Data can be put in inconsistent state
  - • Context switch can occur at anytime, such as before a thread finishes modifying value
- – Such data must be accessed in mutually exclusive way
  - • Only one thread allowed access at one time
  - • Others must wait until resource is unlocked
  - • Serialized access
  - • Must be managed such that wait time not unreasonable

2.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

- – One thread creates data to store in shared object
- – Second thread reads data from that object
  - • Large potential for data corruption if unsynchronized

Figure 5.1 Buffer interface used in producer/consumer examples

```
1  // Fig. 5.1: Buffer.java
2  // Buffer interface specifies methods to access buffer data.
3
4  public interface Buffer
5  {
6     public void set( int value );  // place value into Buffer
7     public int get();              // return value from Buffer
8  }
```

Producer class represents the producer thread in a producer/consumer relationship.

```
1  // Fig. 5.2: Producer.java
2  // Producer's run method controls a producer thread that
3  // stores values from 1 to 4 in Buffer sharedLocation.
4
5  public class Producer extends Thread
6  {
7     private Buffer sharedLocation; // reference to shared object
8
9     // Producer constructor
10    public Producer( Buffer shared )
11    {
12       super( "Producer" ); // create thread named "Producer"
13       sharedLocation = shared; // initialize sharedLocation
14    } // end Producer constructor
15
16    // Producer method run stores values from
17    // 1 to 4 in Buffer sharedLocation
18    public void run()
19    {
20       for ( int count = 1; count <= 4; count++ )
21       {
22          // sleep 0 to 3 seconds, then place value in Buffer
23          try
24          {
25             Thread.sleep( ( int ) ( Math.random() * 3001 ) );
26             sharedLocation.set( count ); // write to the buffer
27          } // end try
28
29          // if sleeping thread interrupted, print stack trace
30          catch ( InterruptedException exception )
31          {
32             exception.printStackTrace();
33          } // end catch
34
35       } // end for
36
37       System.err.println( getName() + " done producing." +
38          "\nTerminating " + getName() + "." );
39
40    } // end method run
41
42 } // end class Producer
```

Consumer class represents the consumer thread in a producer/consumer relationship

```
1  // Fig. 5.3: Consumer.java
2  // Consumer's run method controls a thread that loops four
3  // times and reads a value from sharedLocation each time.
4
5  public class Consumer extends Thread
6  {
7     private Buffer sharedLocation; // reference to shared object
8
9     // Consumer constructor
10    public Consumer( Buffer shared )
11    {
12       super( "Consumer" ); // create thread named "Consumer"
13       sharedLocation = shared; // initialize sharedLocation
14    } // end Consumer constructor
15
16    // read sharedLocation's value four times and sum the values
17    public void run()
18    {
19       int sum = 0;
20
21       // alternate between sleeping and getting Buffer value
22       for ( int count = 1; count <= 4; ++count )
23       {
24          // sleep 0-3 seconds, read Buffer value and add to sum
25          try
26          {
27             Thread.sleep( ( int ) ( Math.random() * 3001 ) );
28             sum += sharedLocation.get();
29          }
30
31          // if sleeping thread interrupted, print stack trace
32          catch ( InterruptedException exception )
33          {
34             exception.printStackTrace();
35          }
36       } // end for
37
38       System.err.println( getName() + " read values totaling: "
39          + sum + ".\nTerminating " + getName() + "." );
40
41    } // end method run
42
43 } // end class Consumer
```

UnsynchronizedBuffer class maintains the shared integer that is accessed by a producer thread and a consumer thread via methods set and get.

```
1  // Fig. 5.4: UnsynchronizedBuffer.java
2  // UnsynchronizedBuffer represents a single shared integer.
3
4  public class UnsynchronizedBuffer implements Buffer
5  {
6     private int buffer = -1; // shared by Producer and Consumer
7
8     // place value into buffer
9     public void set( int value )
10    {
11       System.err.println( Thread.currentThread().getName() +
12          " writes " + value );
13
14       buffer = value;
15    } // end method set
16
17    // return value from buffer
18    public int get()
19    {
20       System.err.println( Thread.currentThread().getName() +
21          " reads " + buffer );
22
23       return buffer;
24    } // end method get
25
26 } // end class UnsynchronizedBuffer
```

SharedBuffer class enables threads to modify a shared object without synchronization.

```
1  // Fig. 5.5: SharedBufferTest.java
2  // SharedBufferTest creates producer and consumer threads.
3
4  public class SharedBufferTest
5  {
6     public static void main( String [] args )
7     {
8        // create shared object used by threads
9        Buffer sharedLocation = new UnsynchronizedBuffer();
10
11       // create producer and consumer objects
12       Producer producer = new Producer( sharedLocation );
13       Consumer consumer = new Consumer( sharedLocation );
14
15       producer.start();  // start producer thread
16       consumer.start();  // start consumer thread
17
18    } // end main
19
20 } // end class SharedCell
```

Sample Output 1:

```
Consumer reads -1
Producer writes 1
Consumer reads 1
Consumer reads 1
Consumer reads 1
Consumer read values totaling: 2.
Terminating Consumer.
Producer writes 2
Producer writes 3
Producer writes 4
Producer done producing.
Terminating Producer.
```

Critical Sections

- Most code is safe to run concurrently
- Sections where shared data is modified must be protected

– Known as critical sections
– Only one thread can be in its critical section at once

- Must be careful to avoid infinite loops and blocking inside a critical Section

2.3 Implementing Mutual Exclusion Primitives

• Indicate when critical data is about to be accessed
  – Mechanisms are normally provided by programming language or libraries
  – Delimit beginning and end of critical section
    - enterMutualExclusion
    - exitMutualExclusion
    - Common properties of mutual exclusion primitives
  – Each mutual exclusion machine language instruction is executed indivisibly
  – Cannot make assumptions about relative speed of thread execution
  – Thread not in its critical section cannot block other threads from entering their critical sections
  – Thread may not be indefinitely postponed from entering its critical section.

2.4 Software solutions to the Mutual Exlusion problem

2.4.1 Dekker's Algorithm(First version)

• First version of Dekker's algorithm
  – Succeeds in enforcing mutual exclusion
  – Uses variable to control which thread can execute
  – Constantly tests whether critical section is available
    - Busy waiting
    - Wastes significant processor time
  – Problem known as lockstep synchronization
    - Each thread can execute only in strict alternation

Mutual exclusion implementation – version 1

```
1  System:
2
3  int threadNumber = 1;
4
5  startThreads(); // initialize and launch both threads
6
7  Thread T1:
8
9  void main() {
10
11    while ( !done )
12    {
13       while ( threadNumber == 2 ); // enterMutualExclusion
14
15       // critical section code
16
17       threadNumber = 2; // exitMutualExclusion
18
19       // code outside critical section
20
21    } // end outer while
22
23 } // end Thread T1
24
25 Thread T2:
26
27 void main() {
28
29    while ( !done )
30     {
31       while ( threadNumber == 1 );  // enterMutualExclusion
32
33       // critical section code
34
35       threadNumber = 1; // exitMutualExclusion
36
37       // code outside critical section
38
39    } // end outer while
40
41 } // end Thread T2
```

2.4.2 Dekker's Algorithm(Second version)

– Removes lockstep synchronization
– Violates mutual exclusion
    • Thread could be preempted while updating flag variable
– Not an appropriate solution

```
1   System:
2
3   boolean t1Inside = false;
4   boolean t2Inside = false;
5
6   startThreads(); // initialize and launch both threads
7
8   Thread T1:
9
10  void main() {
11
12     while ( !done )
13      {
14        while ( t2Inside ); // enterMutualExclusion
15
16        t1Inside = true; // enterMutualExclusion
17
18        // critical section code
19
20        t1Inside = false; // exitMutualExclusion
21
22        // code outside critical section
23
24     } // end outer while
25
26  } // end Thread T1
27
28  Thread T2:
29
30  void main() {
31
32     while ( !done )
33     {
34        while ( t1Inside ); // enterMutualExclusion
35
36        t2Inside = true; // enterMutualExclusion
37
38        // critical section code
39
40        t2Inside = false; // exitMutualExclusion
41
42        // code outside critical section
43
44     } // end outer while
45
46  } // end Thread T2
```

2.4.3 Dekker's Algorithm(Third version)

– Set critical section flag before entering critical section test
  • Once again guarantees mutual exclusion
– Introduces possibility of deadlock
  • Both threads could set flag simultaneously
  • Neither would ever be able to break out of loop
– Not a solution to the mutual exclusion problem

```
1  System:
2
3  boolean t1WantsToEnter = false;
4  boolean t2WantsToEnter = false;
5
6  startThreads(); // initialize and launch both threads
7
8  Thread T1:
9
10 void main()
11 {
12    while ( !done )
13    {
14       t1WantsToEnter = true; // enterMutualExclusion
15
16       while ( t2WantsToEnter ); // enterMutualExclusion
17
18       // critical section code
19
20       t1WantsToEnter = false; // exitMutualExclusion
21
22       // code outside critical section
23
24    } // end outer while
25
26 } // end Thread T1
27
28 Thread T2:
29
30 void main()
31 {
32    while ( !done )
33    {
34       t2WantsToEnter = true; // enterMutualExclusion
35
36       while ( t1WantsToEnter ); // enterMutualExclusion
37
38       // critical section code
39
40       t2WantsToEnter = false; // exitMutualExclusion
41
42       // code outside critical section
43
44    } // end outer while
45
46 } // end Thread T2
```

2.4.4 Dekker's Algorithm- Fourth version

– Sets flag to false for small periods of time to yield control
– Solves previous problems, introduces indefinite postponement
  • Both threads could set flags to same values at same time
  • Would require both threads to execute in tandem (unlikely but possible)
– Unacceptable in mission- or business-critical systems

```
1   System:
2
3   boolean t1WantsToEnter = false;
4   boolean t2WantsToEnter = false;
5
6   startThreads(); // initialize and launch both threads
7
8   Thread T1:
9
10  void main()
11  {
12     while ( !done )
13     {
14        t1WantsToEnter = true;  // enterMutualExclusion
15
16        while ( t2WantsToEnter )  // enterMutualExclusion
17        {
18           t1WantsToEnter = false;  // enterMutualExclusion
19
20           // wait for small, random amount of time
21
22           t1WantsToEnter = true;
23        } // end while
24
25        // critical section code
26
27        t1WantsToEnter = false; // exitMutualExclusion
28
29        // code outside critical section
30
31     } // end outer while
32
```

```
33  } // end Thread T1
34
35  Thread T2:
36
37  void main()
38  {
39     while ( !done )
40     {
41        t2WantsToEnter = true;  // enterMutualExclusion
42
43        while ( t1WantsToEnter )  // enterMutualExclusion
44        {
45           t2WantsToEnter = false;  // enterMutualExclusion
46
47           // wait for small, random amount of time
48
49           t2WantsToEnter = true;
50        } // end while
51
52        // critical section code
53
54        t2WantsToEnter = false; // exitMutualExclusion
55
56        // code outside critical section
57
58     } // end outer while
59
60  } // end Thread T2
```

2.4.5 Dekker's Algorithm(A proper solution)

– Uses notion of favoured threads to determine entry into critical sections
  - Resolves conflict over which thread should execute first
  - Each thread temporarily unsets critical section request flag
  - Favoured status alternates between threads

– Guarantees mutual exclusion

– Avoids previous problems of deadlock, indefinite postponement

Dekker's Algorithm for mutual exclusion

```
1   System:
2
3   int favoredThread = 1;
4   boolean t1WantsToEnter = false;
5   boolean t2WantsToEnter = false;
6
7   startThreads(); // initialize and launch both threads
8
9   Thread T1:
10
11  void main()
12  {
13     while ( !done )
14     {
15        t1WantsToEnter = true;
16
17        while ( t2WantsToEnter )
18        {
19           if ( favoredThread == 2 )
20           {
21              t1WantsToEnter = false;
22              while ( favoredThread == 2 ); // busy wait
23              t1WantsToEnter = true;
24           } // end if
25
26        } // end while
27
28        // critical section code
29
30        favoredThread = 2;
31        t1WantsToEnter = false;
32
33        // code outside critical section
34
35     } // end outer while
36
37  } // end Thread T1
```

```
38
39  Thread T2:
40
41  void main()
42  {
43     while ( !done )
44     {
45        t2WantsToEnter = true;
46
47        while ( t1WantsToEnter )
48        {
49           if ( favoredThread == 1 )
50           {
51              t2WantsToEnter = false;
52              while ( favoredThread == 1 ); // busy wait
53              t2WantsToEnter = true;
54           } // end if
55
56        } // end while
57
58        // critical section code
59
60        favoredThread = 1;
61        t2WantsToEnter = false;
62
63        // code outside critical section
64
65     } // end outer while
66
67  } // end Thread T2
```

2.4.6 Peterson's Algorithm

• Less complicated than Dekker's Algorithm
  – Still uses busy waiting, favored threads
  – Requires fewer steps to perform mutual exclusion primitives
  – Easier to demonstrate its correctness
  – Does not exhibit indefinite postponement or deadlock

Peterson's Algorithm for mutual exclusion

```
1  System:
2
3  int favoredThread = 1;
4  boolean t1WantsToEnter = false;
5  boolean t2WantsToEnter = false;
6
7  startThreads(); // initialize and launch both threads
```

```
8
9   Thread T1:
10
11  void main()
12  {
13     while ( !done )
14     {
15        t1WantsToEnter = true;
16        favoredThread = 2;
17
18        while ( t2WantsToEnter && favoredThread == 2 );
19
20        // critical section code
21
22        t1WantsToEnter = false;
23
24        // code outside critical section
25
26     } // end while
27
28  } // end Thread T1
29
30  Thread T2:
31
32  void main()
33  {
34     while ( !done )
35     {
36        t2WantsToEnter = true;
37        favoredThread = 1;
38
39        while ( t1WantsToEnter && favoredThread == 1 );
40
41        // critical section code
42
43        t2WantsToEnter = false;
44
45        // code outside critical section
46
47     } // end while
48
49  } // end Thread T2
```

2.4.7 N-Thread Mutual Exclusion:Lamport's Bakery Algorithm

• Applicable to any number of threads

– Creates a queue of waiting threads by distributing numbered "tickets"

– Each thread executes when its ticket's number is the lowest of all threads

– Unlike Dekker's and Peterson's Algorithms, the Bakery Algorithm works in multiprocessor systems and for *n* threads

– Relatively simple to understand due to its real-world analog

Lamport's Bakery Algorithm

```
1   System:
2
3   // array that records which threads are taking a ticket
4   boolean choosing[n];
5
6   // value of the ticket for each thread initialized to 0
7   int ticket[n];
8
9   startThreads(); // initialize and launch all threads
10
11  Thread Tx:
12
13  void main()
14  {
15     x = threadNumber(); // store current thread number
16
17     while ( !done )
18     {
19        // take a ticket
20        choosing[x] = true; // begin ticket selection process
21        ticket[x] = maxValue( ticket ) + 1;
22        choosing[x] = false; // end ticket selection process
23
24        // wait for number to be called by comparing current
25        // ticket  value to other thread's ticket value
26        for ( int i = 0; i < n; i++)
27        {
28           if ( i == x )
29           {
30              continue; // no need to check own ticket
31           } // end if
32
```

```
33          // busy wait while thread[i] is choosing
34          while ( choosing[i] != false );
35
36          // busy wait until current ticket value is lowest
37          while ( ticket[i] != 0 && ticket[i] < ticket[x] );
38
39          // tie-breaker code favors smaller thread number
40          if ( ticket[i] == ticket[x] && i < x )
41
42             // loop until thread[i] leaves its critical section
43             while ( ticket[i] != 0 ); // busy wait
44       } // end for
45
46       // critical section code
47
48       ticket[x] = 0; // exitMutualExclusion
49
50       // code outside critical section
51
52    } // end while
53
54 } // end Thread TX
```

2.5 Hardware Solutions to the Mutual Exclusion Problem

• Implementing mutual exclusion in hardware
  – Can improve performance
  – Can decreased development time
• No need to implement complex software mutual exclusion solutions like Lamport's Algorithm.

2.5.1 Disabling Interrupts

• Disabling interrupts
  – Works only on uniprocessor systems
  – Prevents the currently executing thread from being preempted
  – Could result in deadlock
• For example, thread waiting for I/O event in critical section
  – Technique is used rarely

2.5.2 Test-and-Set Instruction

• Use a machine-language instruction to ensure that mutual exclusion primitives are performed indivisibly
  – Such instructions are called atomic
  – Machine-language instructions do not ensure mutual exclusion alone
• For example, programmers must incorporate favoured threads to avoid indefinite postponement
  – Used to simplify software algorithms rather than replace them
• Test-and-set instruction
  – testAndSet(a, b) copies the value of b to a, then sets b to true
  – Example of an atomic read-modify-write (RMW) cycle

```
1   System:
2
3   boolean occupied = false;
4
5   startThreads(); // initialize and launch both threads
6
7   Thread T1:
8
9   void main()
10  {
11     boolean p1MustWait = true;
12
13     while ( !done )
14     {
15        while ( p1MustWait )
16        {
17           testAndSet( p1MustWait, occupied );
18        }
19
20        // critical section code
21
22        p1MustWait = true;
23        occupied = false;
24
25        // code outside critical section
26
27     } // end while
28
29  } // end Thread T1
30
31  Thread T2:
32
```

```
33  void main()
34  {
35     boolean p2MustWait = true;
36
37     while ( !done )
38     {
39        while ( p2MustWait )
40        {
41           testAndSet( p2MustWait, occupied );
42        }
43
44        // critical section code
45
46        p2MustWait = true;
47        occupied = false;
48
49        // code outside critical section
50
51     } // end while
52
53  } // end Thread T2
```

2.5.3 Swap Instruction

- swap(a, b) exchanges the values of a and b atomically
- Similar in functionality to test-and-set
  - swap is more commonly implemented on multiple architectures

```
1   System:
2
3   boolean occupied = false;
4
5   startThreads(); // initialize and launch both threads
6
7   Thread T1:
8
9   void main()
10  {
11     boolean p1MustWait = true;
12
```

```
13      while ( !done )
14      {
15         do
16         {
17            swap( p1MustWait, occupied );
18         } while ( p1MustWait );
19
20         // critical section code
21
22         p1MustWait = true;
23         occupied = false;
24
25         // code outside critical section
26
27      } // end while
28
29   } // end Thread T1
30
31   Thread T2:
32
33   void main()
34   {
35      boolean p2MustWait = true;
36
37      while ( !done )
38      {
39         do
40         {
41            swap( p2MustWait, occupied );
42         } while ( p2MustWait );
43
44         // critical section code
45
46         p2MustWait = true;
47         occupied = false;
48
49         // code outside critical section
50
51      } // end while
52
53   } // end Thread T2
```

2.6 Semaphores

• Semaphores
  – Software construct that can be used to enforce mutual exclusion
  – Contains a protected variable
• Can be accessed only via wait and signal commands
• Also called *P* and *V* operations, respectively

2.6.1 Mutual Exclusion with Semaphores

• Binary semaphore: allow only one thread in its critical section at once
  – Wait operation
• If no threads are waiting, allow thread into its critical section
• Decrement protected variable (to 0 in this case)

• Otherwise place in waiting queue
  – Signal operation
• Indicate that thread is outside its critical section
• Increment protected variable (from 0 to 1)
• A waiting thread (if there is one) may now enter

2.6.2 Thread Synchronization with Semaphores

• Semaphores can be used to notify other threads that events have occurred
  – Producer-consumer relationship
• Producer enters its critical section to produce value
• Consumer is blocked until producer finishes
• Consumer enters its critical section to read value
• Producer cannot update value until it is consumed
  – Semaphores offer a clear, easy-to-implement solution to this problem

Producer/consumer relationship implemented with semaphores.

```
1  System:
2  // semaphores that synchronize access to sharedValue
3  Semaphore valueProduced = new Semaphore(0);
4  Semaphore valueConsumed = new Semaphore(1);
5  int sharedValue; // variable shared by producer and consumer
6
7  startThreads(); // initialize and launch both threads
8
9  Producer Thread:
10
11 void main()
12 {
13    int nextValueProduced; // variable to store value produced
14
15    while ( !done )
16    {
17       nextValueProduced = generateTheValue(); // produce value
18       P( valueConsumed ); // wait until value is consumed
19       sharedValue = nextValueProduced; // critical section
20      V( valueProduced );  // signal that value has been produced
21
22    } // end while
23
24 } // end producer thread
```

```
25
26  Consumer Thread:
27
28  void main()
29  {
30     int nextValue; // variable to store value consumed
31
32     while ( !done )
33     {
34        P( valueProduced ); // wait until value is produced
35        nextValueConsumed = sharedValue; // critical section
36        V( valueConsumed ); // signal that value has been consumed
37        processTheValue( nextValueConsumed ); // process the value
38
39     } // end while
40
41  } // end consumer thread
```

2.6.3 Counting Semaphores

• Counting semaphores
  – Initialized with values greater than one
  – Can be used to control access to a pool of identical resources
• Decrement the semaphore's counter when taking resource from pool
• Increment the semaphore's counter when returning it to pool
• If no resources are available, thread is blocked until a resource becomes available

2.6.4 Implementing Semaphores

• Semaphores can be implemented at application or kernel level
  – Application level: typically implemented by busy waiting
• Inefficient– Kernel implementations can avoid busy waiting
• Block waiting threads until they are ready– Kernel implementations can disable interrupts
• Guarantee exclusive semaphore access
• Must be careful to avoid poor performance and deadlock
• Implementations for multiprocessor systems must use a more sophisticated approach

2.7 Monitors

2.7.1 Introduction

• Recent interest in concurrent programming languages
  – Naturally express solutions to inherently parallel problems
  – Due to proliferation of multiprocessing systems, distributed systems and massively parallel architectures
  – More complex than standard programs
• More time required to write, test and debug

-Monitor– Contains data and procedures needed to allocate shared resources

• Accessible only within the monitor
• No way for threads outside monitor to access monitor data

Threads return resources through monitors as well

  – Monitor entry routine calls signal
• Alerts one waiting thread to acquire resource and enter monitor
  – Higher priority given to waiting threads than ones newly arrived
• Avoids indefinite postponement

2.7.2 Condition Variables

• Before a thread can reenter the monitor, the thread calling signal must first exit monitor
  – Signal-and-exit monitor
• Requires thread to exit the monitor immediately upon signalling
• Signal-and-continue monitor
  – Allows thread inside monitor to signal that the monitor will soon become available
  – Still maintain lock on the monitor until thread exits monitor
  – Thread can exit monitor by waiting on a condition variable or by completing execution of code protected by monitor

2.7.3 Simple Resource Allocation with Monitors

• Thread inside monitor may need to wait outside until another thread performs an action inside monitor
• Monitor associates separate condition variable with distinct situation that might cause thread to wait
  – Every condition variable has an associated queue

```
1  // Fig. 6.1: Resource allocator monitor
2
3  // monitor initialization (performed only once)
4  boolean inUse = false; // simple state variable
5  Condition available; // condition variable
6
7  // request resource
8  monitorEntry void getResource()
9  {
10    if ( inUse ) // is resource in use?
11    {
12       wait( available ); // wait until available is signaled
13    } // end if
14
15    inUse = true; // indicate resource is now in use
16
17 } // end getResource
18
19 // return resource
20 monitorEntry void returnResource()
21 {
22    inUse = false; // indicate resource is not in use
23    signal( available ); // signal a waiting thread to proceed
24
25 } // end returnResource
```

2.7.4 Monitor Example: Circular Buffer

• Circular buffer implementation of the solution to producer/consumer problem
  – Producer deposits data in successive elements of array
  – Consumer removes the elements in the order in which they were deposited (FIFO)
  – Producer can be several items ahead of consumer
  – If the producer fills last element of array, it must “wrap around” and begin depositing data in the first element of array

Due to the fixed size of a circular buffer
  – Producer will occasionally find all array elements full, in which case the producer must wait until consumer empties an array element
  – Consumer will occasionally find all array elements empty, in which case the consumer must wait until producer deposits data into an array element

```
3  char circularBuffer[] = new char[ BUFFER_SIZE ]; // buffer
4  int writerPosition = 0; // next slot to write to
5  int readerPosition = 0; // next slot to read from
6  int occupiedSlots = 0;  // number of slots with data
7  Condition hasData;      // condition variable
8  Condition hasSpace;     // condition variable
9
10 // monitor entry called by producer to write data
11 monitorEntry void putChar( char slotData )
12 {
13    // wait on condition variable hasSpace if buffer is full
14    if ( occupiedSlots == BUFFER_SIZE )
15    {
16       wait( hasSpace ); // wait until hasSpace is signaled
17    } // end if
18
19    // write character to buffer
20    circularBuffer[ writerPosition ] = slotData;
21    ++occupiedSlots; // one more slot has data
22    writerPosition = (writerPosition + 1) % BUFFER_SIZE;
23    signal( hasData ); // signal that data is available
24 } // end putChar
25
26 // monitor entry called by consumer to read data
27 monitorEntry void getChar( outputParameter slotData )
28 {
29    // wait on condition variable hasData if the buffer is empty
30    if ( occupiedSlots == 0 )
31    {
32       wait( hasData ); // wait until hasData is signaled
33    } // end if
34
35    // read character from buffer into output parameter slotData
36    slotData = circularBuffer[ readPosition ];
37    occupiedSlots--; // one fewer slots has data
38    readerPosition = (readerPosition + 1) % BUFFER_SIZE;
39    signal( hasSpace ); // signal that character has been read
40 } // end getChar
```

2.7.5 Monitor Example: Readers and Writers

• Readers
  – Read data, but do not change contents of data
  – Multiple readers can access the shared data at once
  – When a new reader calls a monitor entry routine, it is allowed to proceed as long as no thread is writing and no writer thread is waiting to write
  – After the reader finishes reading, it calls a monitor entry routine

to signal the next waiting reader to proceed, causing a “chain reaction”

• Writers
  – Can modify data
  – Must have exclusive access

```
// Fig. 6.3: Readers/writers problem

int readers = 0; // number of readers
boolean writeLock = false; // true if a writer is writing
Condition canWrite; // condition variable
Condition canRead; // condition variable

// monitor entry called before performing read
monitorEntry void beginRead()
{
   // wait outside monitor if writer is currently writing or if
   // writers are currently waiting to write
   if ( writeLock || queue( canWrite ) )
   {
      wait( canRead ); // wait until reading is allowed
   } // end if

   ++readers; // there is another reader

   signal( canRead ); // allow waiting readers to proceed
} // end beginRead

// monitor entry called after reading
monitorEntry void endRead()
{
   --readers; // there are one fewer readers

   // if no more readers are reading, allow a writer to write
   if ( readers == 0 )
   {
      signal ( canWrite ); // allow a writer to proceed
   } // end if

} // end endRead

// monitor entry called before performing write
monitorEntry void beginWrite()
{
   // wait if readers are reading or if a writer is writing
   if ( readers > 0 || writeLock )
   {
      wait( canWrite ); // wait until writing is allowed
   } // end if

```

```
45       writeLock = true; // lock out all readers and writers
46    } // end beginWrite
47
48    // monitor entry called after performing write
49    monitorEntry void endWrite()
50    {
51       writeLock = false; // release lock
52
53       // if a reader is waiting to enter, signal a reader
54       if ( queue( canRead ) )
55       {
56          signal( canRead ); // cascade in waiting readers
57       } // end if
58       else // signal a writer if no readers are waiting
59       {
60          signal( canWrite ); // one waiting writer can proceed
61       } // end else
62
63    } // end endWrite
```

UNIT III

DEADLOCK AND INDEFINITE POSTPONEMENT

OUTLINE

DEADLOCK AND INDEFINITE POSTPONEMENT

7.1 INTRODUCTION

1. A process or thread is waiting for a particular event that will not occur is called Deadlock.
2. In System deadlock one or more processes are deadlocked.

7.2 EXAMPLES OF DEADLOCK

Deadlocks can develop in many ways. We have a one process deadlock. Such deadlocks are extremely difficult to detect. Deadlocks in real systems involve multiple processes for multiple resources of multiple types.

7.2.1 TRAFFIC DEADLOCK

This kind of deadlocks is developed in cities .A number of Automobiles is attempting to drive through a busy neighborhood and the traffic is snarled.

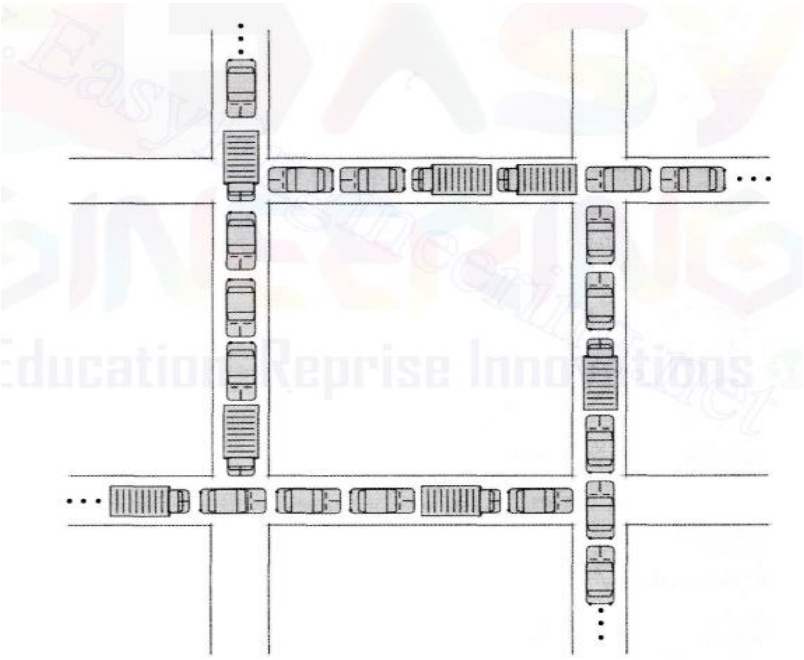
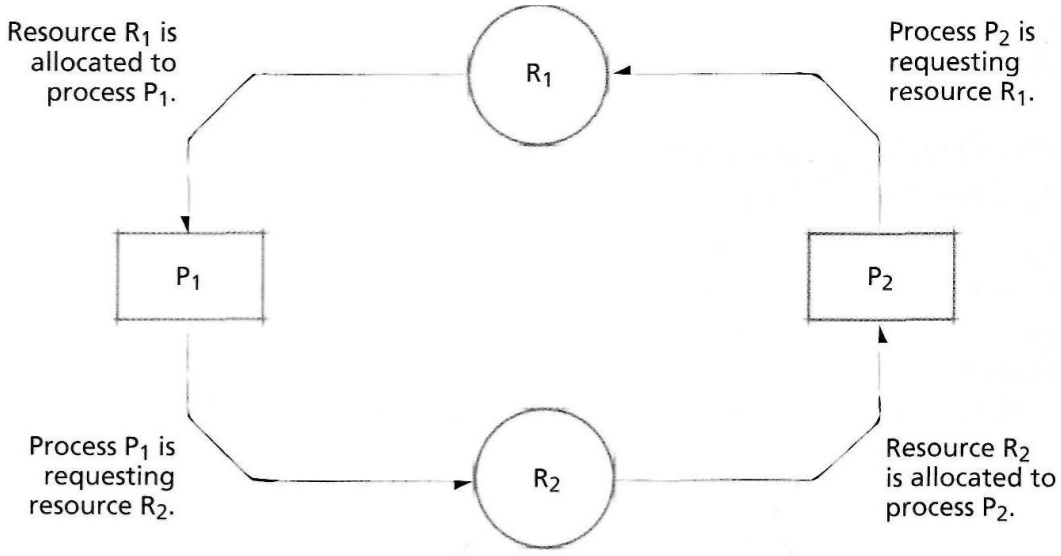


Figure 7.1 | Traffic deadlock example.[4]

7.2.2 SIMPLE RESOURCE DEADLOCK

- An example of simple resource deadlock is resource allocation graph.
- It shows two processes as rectangles and two resources as circles.
- An arrow from a resource to a process indicates that the resource is allocated to the process
- . An arrow from a process to a resource indicates that the process is requesting, but has not yet been allocated, the resource.



- Process P1 holds resource R1 and needs resource R2 to continue.
- Process P2 holds resource R2 and needs resource R1 to continue.
- Each process is waiting for the other to free. This circular wait is characteristic of deadlocked systems.

7.2.3 DEADLOCK IN SPOOLING SYSTEMS

- Spooling systems are prone to deadlock.

Common solution

- Restrain input spoolers so that when the spooling file begins to reach some saturation threshold, the spoolers do not read in more print jobs.

Today's systems

- Printing begins before the job is completed so that a full spooling file can be emptied even while a job is still executing.

- Same concept has been applied to streaming audio and video.

7.2.4 EXAMPLE: DINING PHILOSOPHERS

Problem statement:

Five philosophers sit around a circular table. Each leads a simple life alternating between thinking and eating spaghetti. In front of each philosopher is a dish of spaghetti that is constantly replenished by a dedicated wait staff. There are exactly five forks on the table, one between each adjacent pair of philosophers. Eating spaghetti (in the most proper manner) requires that a philosopher use both adjacent forks (simultaneously). Develop a concurrent program free of deadlock and indefinite postponement that models the activities of the philosophers.

```
void typicalPhilosopher()
{
        while ( true )
        {
                think();
                eat();
        } // end while

} // end typicalPhilospher
```

Dining philosopher behavior

Constraints:

– To prevent philosophers from starving:

- Free of deadlock
- Free of indefinite postponement

– Enforce mutual exclusion

- Two philosophers cannot use the same fork at once

The problems of mutual exclusion, deadlock and indefinite postponement lie in the implementation of method eat.

```
void eat()
{
        pickUpLeftFork();
        pickUpRightFork();
        eatForSomeTime();
        putDownRightFork();
        putDownLeftFork();
} // eat
```

Implementation of method eat

7.3 RELATED PROBLEM: INDEFINITE POSTPONEMENT

- A process may be delayed indefinitely while other processes receive the system's attention.
- This situation, called indefinite postponement, indefinite blocking, or starvation.
- Indefinite postponement may occur due to biases in a system's resource scheduling policies.
- Some systems prevent indefinite postponement by increasing a process's priority gradually as it waits for a resource —this technique is called **aging.**

7.4 RESOURCE CONCEPTS

- Resources that are **preemptible,** such as processors and main memory. Resources can be removed from a process without loss of work.
- Certain resources are **nonpreemptible;** they cannot be removed from the processes to which they are assigned until the processes voluntarily release them.
- For example, tape drives and optical scanners.

Reentrant code

– Cannot be changed while in use.

– May be shared by several processes simulta

neously.

Serially reusable code

– May be changed but is reinitialized each time it is used.

– May be used by only one process at a time.

7.5 FOUR NECESSARY CONDITIONS FOR DEADLOCK

1. Mutual exclusion condition

– Resource may be acquired exclusively by only one process at a time.

2. Wait-for condition (hold-and-wait condition)

– Process that has acquired an exclusive resource may hold that resource while the process waits to obtain other resources

3. No-preemption condition

– Once a process has obtained a resource, the system cannot remove it from the process's control until the process has finished using the resource.

4. Circular-wait condition

– Two or more processes are locked in a **"circular chain"** in which each process is waiting for one or more resources that the next process in the chain is holding.

7.6 DEADLOCK SOLUTIONS

There are four major areas of interest in deadlock research.

1. **Deadlock prevention**

   In deadlock prevention is to condition a system to remove any possibility of deadlocks occurring, but prevention methods can often result in poor resource utilization.

2. **Deadlock avoidance**

   Avoidance methods do not precondition the system to remove all possibility of deadlock.

3. **Deadlock detection**

   If a deadlock has occurred, and to identify the processes and resources that are involved.

4. **Deadlock recovery**

   Deadlock recovery methods are used to clear deadlocks from a system so that it may operate free them, and so that the deadlocked processes may complete their execution and free their resources.

7.7 DEADLOCK PREVENTION

A deadlock cannot occur if a system denies any of the four necessary conditions, suggested the following deadlock prevention strategies:

• Each process must request all its required resources at once and cannot proceed until all have been granted.

• If a process holding certain resources is denied a further request, it must release its original resources and, if necessary, request them again together with the additional resources.

• A linear ordering of resources must be imposed on all processes; i.e., if a process has been allocated certain resources, it may subsequently request only those resources later in the ordering.

7.7.1 DENYING THE "WAIT-FOR" CONDITION

- All of the resources a process needs to complete its task must be requested at once.

- If all the resources needed by a process are available, then the system may grant them all to the process at once. If they are not all available, then the process must wait until they are.

- This leads to inefficient resource allocation.

- One approach to getting better resource utilization in these circumstances is to divide a program into several threads that run relatively independently of one another.

- Another way to avoid this is to handle the needs of the waiting processes in first-come-first served order.

7.7.2 DENYING THE "NO-PREEMPTION" CONDITION

- The second strategy denies the "no preemption" condition.
- Suppose a system does allow processes to hold resources while requesting additional resources.
- As long as sufficient resources remain available to satisfy all requests, the system cannot deadlock.
- When a request for additional resources cannot be satisfied. Now a process holds resources that a second process may need in order to proceed, while the second process may hold resources needed by the first process—a two-process deadlock.
- Processes may lose work when resources are preempted. This can lead to substantial overhead as processes must be restarted.

7.7.3 DENYING THE "CIRCULAR-WAIT" CONDITION

- In this strategy, we assign a unique number to each resource (e.g., a disk drive, printer, scanner, and file) that the system manages and we create a **linear ordering** of resources.
- One disadvantage of this strategy is that it is not as flexible or dynamic as we might desire.
- Resources must be requested in ascending order by resource number.
- Resource numbers are assigned for the computer system and must be "lived with" for long periods (i.e.,months or even years).

- If new resources are added or old ones removed at an installation, existing programs and systems may have to be rewritten.
- Requires the programmer to determine the ordering or resources for each system.

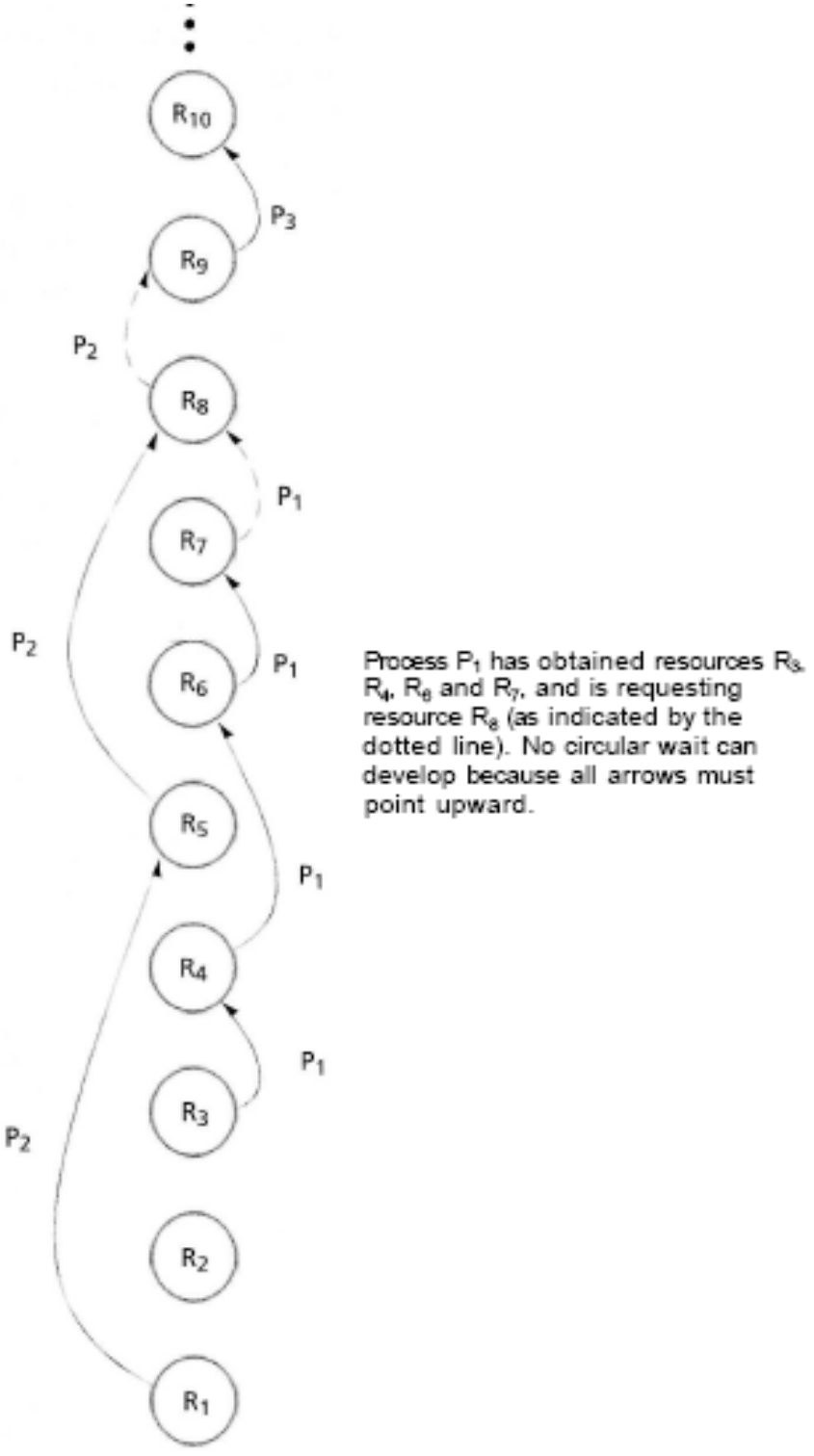


Figure 7.5 | Havender's linear ordering of of resources for preventing deadlock.

7.8 DEADLOCK AVOIDANCE WITH DIJKSTRA'S BANKER'S ALGORITHM

The Banker's Algorithm defines how a particular system can prevent deadlock by controlling how resources are distributed to users. The Banker's Algorithm prevents deadlock in operating systems that exhibit the following properties:

- The operating system shares a fixed number of resources, *t,* among a fixed number of processes, *n.*
- Each process specifies in advance the maximum number of resources that it requires to complete its work.
- The operating system accepts a process's request if that process's **maximum need** does not exceed the total number of resources available in the system, *t* (i.e., the process cannot request more than the total number of resources available in the system).
- Sometimes, a process may have to wait to obtain an additional resource, but the operating system guarantees a finite wait time.
- If the operating system is able to satisfy a process's maximum need for resources, then the process guarantees that the resource will be used and released to the operating system within a finite time.

The system is said to be in a safe **state** if the operating system can guarantee that all current processes can complete their work within a finite time. If not, then the system is said to be in an **unsafe state.**

We also define four terms that describe the distribution of resources among processes.

- Let *max(Pi)* be the maximum number of resources that process P*i* requires during its execution. For example, if process P3 never requires more than two resources, then *max(P3) = 2.*
- Let *loan(Pi)* represent process P*i*'s current **loan** of a resource, where its loan is the number of resources the process has already obtained from the system. For example, if the system has allocated four resources to process P5, then *loan(P5)* = 4.
- Let *claim(Pi)* be the current **claim** of a process, where a process's claim is equal to its maximum need minus its current loan. For example, if process P7 has a maximum need of six resources and a current loan of four resources, then we have *claim(P7) = max(P7) - loan(P7)* = 6 - 4 = 2
- Let *a* be the number of resources still available for allocation. This is equivalent to the total number of resources *(t)* minus the sum of the loans to all the processes in the system, i.e.,

$$a = t - \sum_{i=1}^{n} loan(P_i)$$

7.8.1 EXAMPLE OF A SAFE STATE

Process	max(P_i) (maximum need)	loan (P_i) (current loan)	claim(P_i) (current claim)
P_1	4	1	3
P_2	6	4	2
P_3	8	5	3
Total resources, X, = 12		Available resources, a, = 2	

Figure 7.6 | Safe State.

- This state is "safe" because process P2 currently has a loan of four resources and will eventually need a maximum of six, or two additional resources.
- The system has 12 resources, of which 10 are currently in use and two are available. If the system allocates these two available resources to P2, fulfilling P2's maximum need, then P2 can run to completion.
- After P2 finishes, it will release six resources, enabling the system to immediately fulfill the maximum needs of P1 (3) and P3 (3), enabling both of those processes to finish.

7.8.2 EXAMPLE OF AN UNSAFE STATE

Process	max(P_i) (maximum need)	loan(P_i) (current loan)	claim (P_i) (current claim)
P_1	10	8	2
p_2	5	2	3
p_3	3	1	2
Total resources, t, = 12		Available resources, a, = 1	

Figure 7.7 | Unsafe state.

- We sum the values of the third column and subtract from 12 to obtain a value of one for a.
- At this point, no matter which process requests the available resource, we cannot guarantee that all three processes will finish.
- In fact, suppose process P1 requests and is granted the last available resource.
- A three-way deadlock could occur if indeed each process needs to request at least one more resource before releasing any resources to the pool.
- It is important to note here that an unsafe state does not imply the existence of deadlock, nor even that deadlock will eventually occur.

7.8.3 EXAMPLE OF SAFE-STATE-TO-UNSAFE-STATE TRANSITION

- The current value of *a* is 2.
- Now suppose that process P3 requests an additional resource.
- If the system were to grant this request, then the new state would be as in Fig. 7.8.
- Now, the current value of *a* is 1, which is not enough to satisfy the current claim of any process, so the state is now unsafe.

Process	$max(P_i)$ (maximum need)	$loan(P_i)$ (current loan)	$claim(P_i)$ (current claim)
P_1	4	1	3
P_2	6	4	2
P_3	8	6	2
Total resources, t, = *12*		*Available resources*, a, = 1	

Figure 7.8 | *Safe-state-to-unsafe-state transition.*

7.8.4 BANKER'S ALGORITHM RESOURCE ALLOCATION

- The "mutual exclusion," "wait-for," and "no-preemption" conditions are allowed—processes are indeed allowed to hold resources while requesting and waiting for additional resources, and resources may not be preempted from a process holding those resources.

- The system may either grant or deny each request.
- If a request is denied, that process holds any allocated resources and waits for a finite time until that request is eventually granted.
- The system grants only requests that result in safe states.
- Resource requests that would result in unsafe states are repeatedly denied until they can eventually be satisfied.

7.8.5 WEAKNESSES IN THE BANKER'S ALGORITHM

The algorithm has a number of weaknesses.

- It requires that there be a fixed number of resources to allocate.
- The algorithm requires that the population of processes remains fixed. In today's interactive and multiprogrammed systems the process population is constantly changing.
- The algorithm requires that the banker (i.e., the system) grant all requests within a "finite time."
- Similarly, the algorithm requires that clients (i.e., processes) repay all loans (i.e., return all resources) within a "finite time."
- The algorithm requires that processes state their maximum needs in advance. With resource allocation becoming increasingly dynamic, it is becoming more difficult to know a process's maximum needs.

For the reasons stated above, Dijkstra's Banker's Algorithm is not implemented in today's operating systems.

7.9 DEADLOCK DETECTION

Deadlock detection is the process of determining that a deadlock exists and identifying the processes and resources involved in t h e deadlock.

Deadlock detection algorithms can incur significant runtime overhead.

7.9.1 RESOURCE-ALLOCATION GRAPHS

- In Fig. 7.10(a), process P1 is requesting a resource of type R1. The arrow from P1 indicating that the resource request is under consideration.

- In Fig. 7.10(b), process P2 has been allocated a resource of type R2 (of which there are two).The arrow is drawn from the small circle within the large circle R2 to the square P2, to indicate that the system has allocated a specific resource.
- Figure 7.10(c) indicates a situation somewhat closer to a potential deadlock.
- Process P3 is requesting a resource of type R3, but the system has allocated the only R3 resource to process P4.

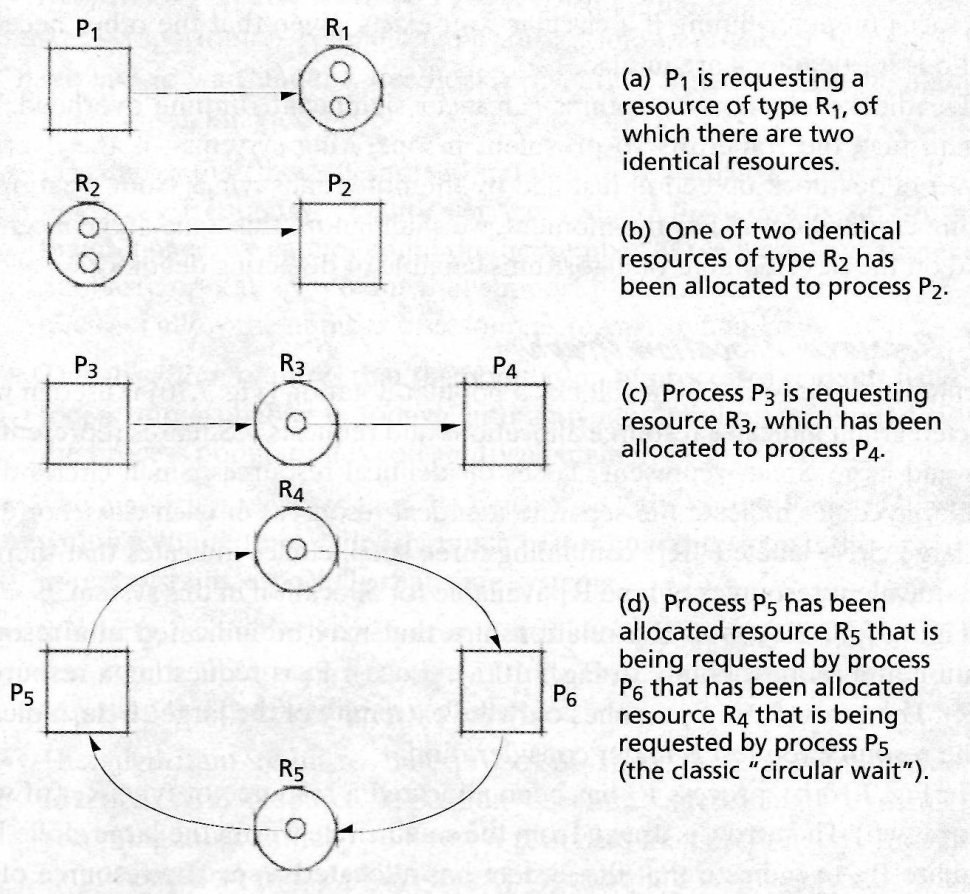


Figure 7.10 | Resource-allocation and request graphs

- Figure 7.10(d) indicates a deadlocked system in which process P5 is requesting a resource of type R4, the only one of which the system has allocated to process P6.
- Process P6, is requesting a resource of type R5, the only one of which the system has allocated to process P5.
- This is an example of the "circular wait" necessary for a deadlocked system.

7.9.2 REDUCTION OF RESOURCE-ALLOCATION GRAPHS

One technique useful for detecting deadlocks involves graph reductions. If a process's resource requests may be granted, then we say that a graph may be **reduced** by that process. This reduction is equivalent to showing how the graph would look if the process was allowed to complete its execution and return its resources to the system.

If a graph can be reduced by all its processes, then there is no deadlock. If a graph cannot be reduced by all its processes, then the irreducible processes constitute the set of deadlocked processes in the graph.

Figure 7.11 shows a series of graph reductions demonstrating that a particular set of processes is not deadlocked.

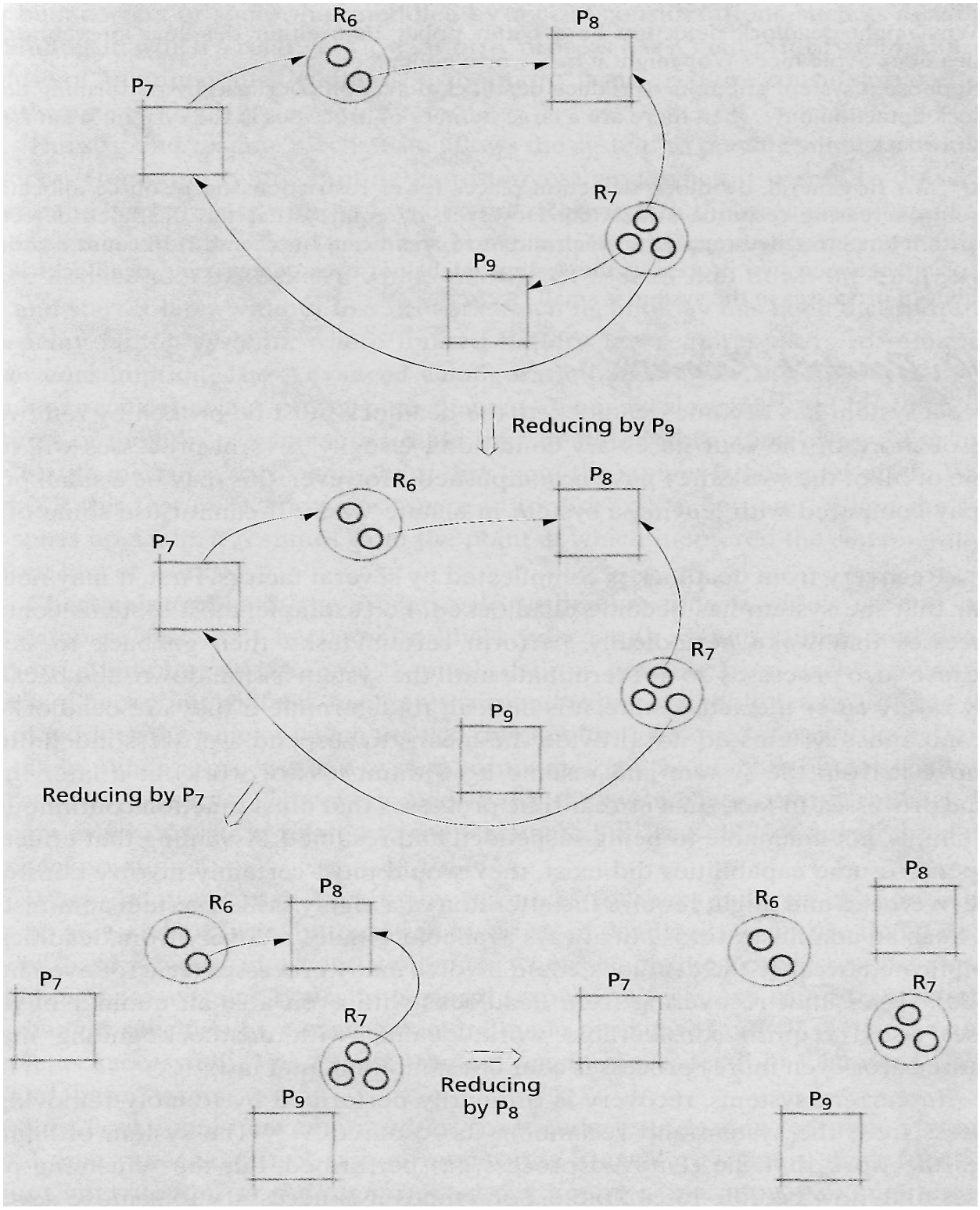


Figure 7.11 | *Graph reductions determining that no deadlock exists*

7.10 DEADLOCK RECOVERY

Recovery from deadlock is complicated by several factors. First, it may not be clear that the system has become deadlocked.

Second, most systems do not provide the means to suspend a process indefinitely, remove it from the system and resume it (without loss of work) at a later time. Some processes, in fact, such as real-time processes that must function continuously, are simply not amenable to being suspended and resumed.

Finally, recovery from deadlock is complicated because the deadlock could involve many processes (tens, or even hundreds).

Suspend/resume mechanism

The suspend/resume mechanism allows the system to put a temporary hold on a process (temporarily preempting its resources), and, when it is safe to do so, to resume the held process without loss of work.

Checkpoint/rollback

Checkpoint/rollback copes with system failures and deadlocks by attempting to preserve as much data as possible from each terminated process. Checkpoint/rollback facilitates suspend/resume capabilities by limiting the loss of work to the time at which the last **checkpoint** (i.e., saved state of the system) was taken.

PROCESSOR SCHEDULING

OUTLINE

PROCESSOR SCHEDULING

8.1 INTRODUCTION

A system has a choice of processes to execute, it must have a strategy —called a **processor scheduling policy** (or **discipline)** —for deciding which process to run at a given time.

A scheduling policy should attempt to satisfy certain performance criteria, such as maximizing the number of processes that complete per unit time (i.e., throughput), minimizing the time each process waits before executing (i.e., latency), preventing indefinite postponement of processes, ensuring that each process completes before its stated deadline, or maximizing processor utilization.

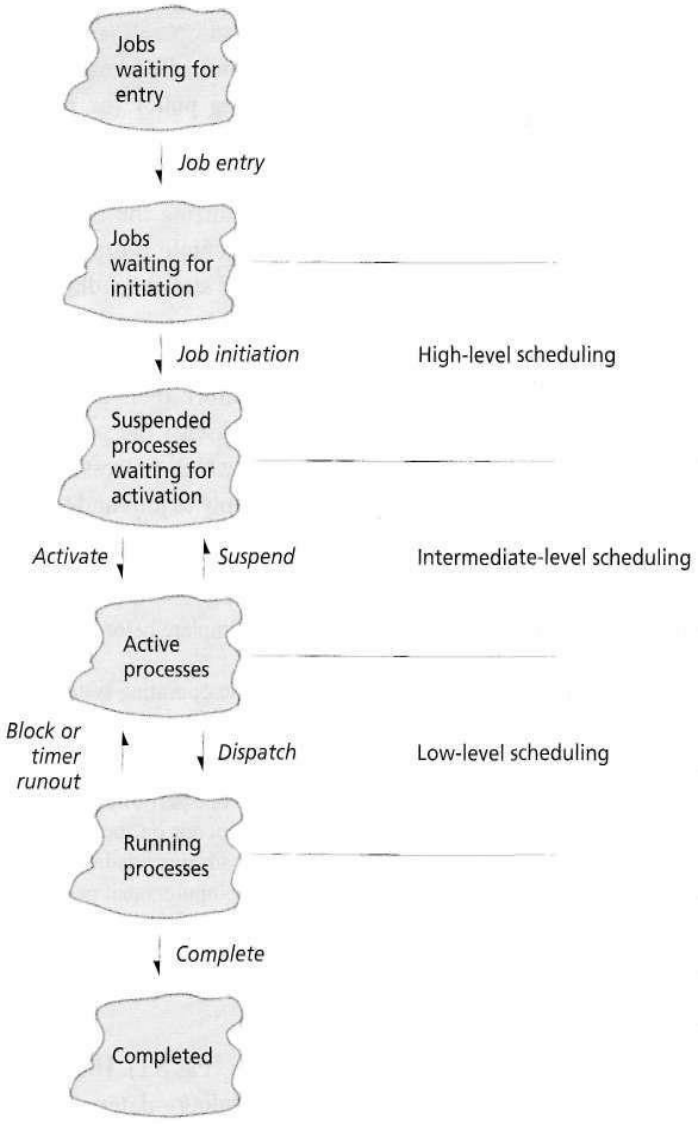
8.2 SCHEDULING LEVELS

We consider three levels of scheduling. **High-level scheduling**—also called **job scheduling or long-term scheduling**—determines which jobs the system allows to compete actively for system resources. This level is sometimes called **admission scheduling**, because it determines which jobs gain admission to the system. Once admitted, jobs are initiated and become processes or groups of processes

The **intermediate-level scheduling** policy determines which processes shall be allowed to compete for processors. This policy responds to short-term fluctuations in system load.

A system's **low-level scheduling policy** determines which active process the system will assign to a processor when one next becomes available. Low-level scheduling policies often assign a priority to each process. The low-level scheduler (also called the dispatcher) also assigns (i.e., dispatches) a processor to the selected process.

Scheduling Levels



8.3 PREEMPTIVE VS. NONPREEMTPIVE SCHEDULING

A scheduling discipline is **nonpreemptive** if, once the system has assigned a processor to a process, the system cannot remove that processor from that process. A scheduling discipline is **preemptive** if the system can remove the processor from the process it is running.

Under a nonpreemptive scheduling discipline, each process, once given a processor, runs to completion or until it voluntarily relinquishes its processor. Under a preemptive scheduling discipline, the processor may execute a portion of a process's code and then perform a context switch.

8.4 PRIORITIES

Static priorities remain fixed, so static-priority-based mechanisms are relatively

easy to implement and incur relatively low overhead. Such mechanisms are not, however, responsive to changes in environment, even those that could increase throughput and reduce latency.

Dynamic priority mechanisms are responsive to change. For example, the system may want to increase the priority of a process that holds a key resource needed by a higher-priority process. After the first process relinquishes the resource, the system lowers the priority, so that the higher-priority process may execute. Dynamic priority schemes are more complex to implement and have greater overhead than static schemes. The overhead is justified by the increased responsiveness of the system.

8.5 SCHEDULING OBJECTIVES

Depending on the system, the user and designer might expect the scheduler to:

- *Maximize throughput. A* scheduling discipline should attempt to service the maximum number of processes per unit time.
- *Maximize the number of interactive processes receiving "acceptable" response times.*
- *Maximize resource utilization.* The scheduling mechanisms should keep the resources of the system busy.
- *Avoid indefinite postponement. A* process should not experience an unbounded wait time before or while receiving service.
- *Enforce priorities.* If the system assigns priorities to processes, the scheduling mechanism should favor the higher-priority processes.
- *Minimize overhead.* Interestingly, this is not generally considered to be one of the most important objectives. Overhead often results in wasted resources. Overhead can greatly improve overall system performance.
- *Ensure predictability.* By minimizing the statistical variance in process response times, a system can guarantee that processes will receive predictable service levels (see the Operating Systems Thinking feature, Predictability).

Despite the differences in goals among systems, many scheduling disciplines exhibit similar properties:

- **Fairness.** A scheduling discipline is fair if all similar processes are treated the same, and no process can suffer indefinite postponement due to scheduling issues (see the Operating Systems Thinking feature, Fairness).
- **Predictability.** A given process always should run in about the same amount of time under similar system loads.

• **Scalability.** System performance should degrade gracefully (i.e., it should not immediately collapse) under heavy loads.

8.6 SCHEDULING CRITERIA

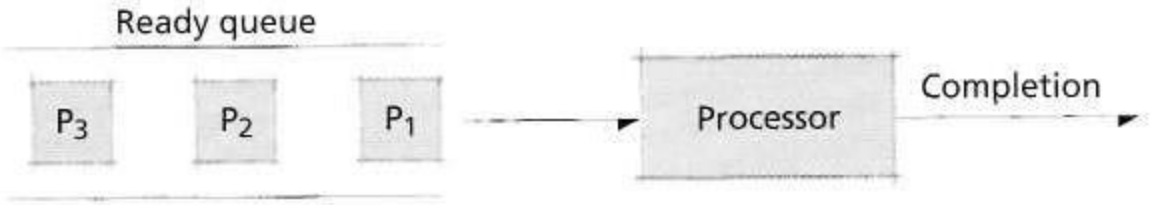
A **processor-bound** process tends to use all the processor time that the system allocates for it. An **I/O-bound** process tends to use the processor only briefly before generating an I/O request and relinquishing it. Processor-bound processes spend most of their time using the processor; I/O-bound processes spend most of their time waiting for external resources (e.g., printers, disk drives, network connections, etc.) to service their requests

A **batch process** contains work for the system to perform without interacting with the user. An **interactive process** requires frequent inputs from the user. The system should provide good response times to an interactive process, whereas a batch process generally can suffer reasonable delays.

8.7 SCHEDULING ALGORITHMS

These algorithms decide when and for how long each process runs; they make choices about preemptibility, priorities, running time, time-to-completion, fairness and other process characteristics.

8.7.1 FIRST-IN-FIRST-OUT (FIFO) SCHEDULING

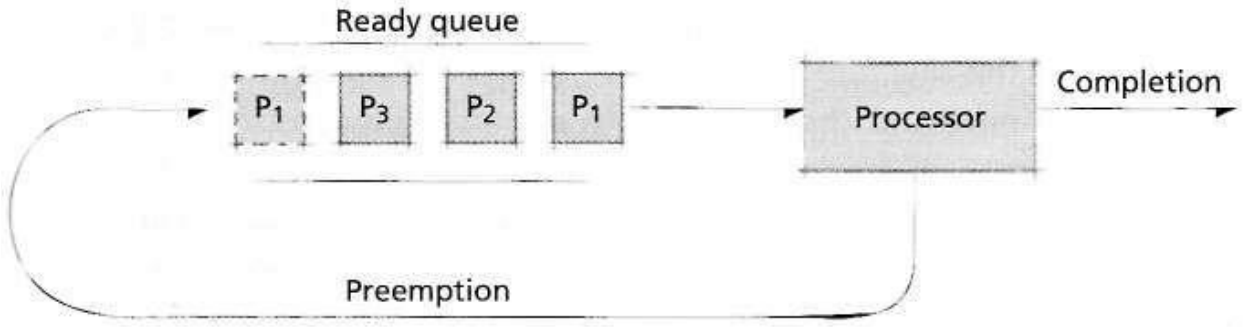


FIFO is nonpreemptive — once a process has a processor, the process runs to completion. FIFO is fair in that it schedules processes according to their arrival times, so all processes are treated equally, but somewhat unfair because long processes make short processes wait, and unimportant processes make important processes wait. FIFO is rarely used as a master scheme in today's systems.

8.7.2 ROUND-ROBIN (RR) SCHEDULING

Processes are dispatched FIFO but are given a limited amount of processor time called a **time slice or** a **quantum.** If a process does not complete before its quantum expires, the system preempts it and gives the processor to the next waiting process. The system then places the preempted process at the back of the ready queue.

Process P1 is dispatched to a processor, where it executes either until completion, in which case it exits the system, or until its time slice expires, at which point it is preempted and placed at the tail of the ready queue.



Selfish Round-Robin

Kleinrock discussed a variant of round-robin called **selfish round-robin (SRR)** that uses aging to gradually increase process priorities over time. In this scheme, as each process enters the system, it first resides in a holding queue, where it ages until the its priority reaches the level of processes in the active queue. At this point, it is placed in the active queue and scheduled round-robin with other processes in the queue. The scheduler dispatches only processes in the active queue, meaning that older processes are favored over those that have just entered the system.

Quantum Size

The system as the quantum gets either extremely large or extremely small. As the quantum gets large, processes tend to receive as much time as they need to complete, so the round-robin scheme degenerates to FIFO. As the quantum gets small, context-switching overhead dominates.

This maximizes I/O utilization and provides relatively rapid response times for interactive processes. It does so with minimal impact to processor-bound processes, which continue to get the lion's share of processor time because I/O-bound processes block soon after executing.

8.7.3 SHORTEST-PROCESS-FIRST (SPF) SCHEDULING

Shortest-process-first (SPF) is a nonpreemptive scheduling discipline in which the scheduler selects the waiting process with the smallest estimated run-time-to-completion. SPF reduces average waiting time over FIFO.

A key problem with SPF is that it requires precise knowledge of how long a process will run, and this information usually is not available. Therefore, SPF must rely on user- or system-supplied run-time estimates.

Another problem with relying on user process duration estimates is that users may supply small (perhaps inaccurate) estimates so that the system will give their programs higher priority.

SPF derives from a discipline called short job first (SJF), which might have worked well scheduling jobs in factories but clearly is inappropriate for low-level scheduling in operating systems. SPF, like FIFO, is nonpreemptive and thus not suitable for environments in which reasonable response times must be guaranteed.

8.7.4 HIGHEST-RESPONSE-RATIO-NEXT (HRRN) SCHEDULING

Brinch Hansen developed the **highest-response-ratio-next (HRRN)** policy that corrects some of the weaknesses in SPF. HRRN is a nonpreemptive scheduling discipline in which each process's priority is a function not only of its service time but also of its time spent waiting for service. Once a process obtains it, the process runs to completion. HRRN calculates dynamic priorities according to the formula

$$\text{priority} = \frac{\text{time waiting} + \text{service time}}{\text{service time}}$$

Because the service time appears in the denominator, shorter processes receive preference. However, because the waiting time appears in the numerator, longer processes that have been waiting will also be given favorable treatment. This technique is similar to aging.

8.7.5 SHORTEST-REMAINING-TIME (SRT) SCHEDULING

Shortest-remaining-time (SRT) scheduling is the preemptive counterpart of SPF that attempts to increase throughput by servicing small arriving processes.

In SRT, the scheduler selects the process with the smallest estimated run-time-to-completion. In SPF, once a process begins executing, it runs to completion. In SRT, a newly arriving process with a shorter estimated run-time preempts a running process with a longer run-time-to-completion.

Again, SRT requires estimates of future process behavior to be effective, and the designer must account for potential user abuse of this system scheduling strategy.

The SRT algorithm offers minimum wait times in theory, but in certain situations, due to preemption overhead, SPF might perform better. The SRT discipline would perform the preemption, but this may not be the optimal choice. One solution is to guarantee that a running process is no longer preemptible when its remaining run time reaches a low-end threshold.

8.7.6 MULTILEVEL FEEDBACK QUEUES

A new process enters the queuing network at the tail of the highest queue. The process progresses through that queue in FIFO order until the process obtains a processor. If the process completes its execution, or if it relinquishes the processor to wait for I/O completion or the completion of some other event, exits the queuing network. If a process's quantum expires before the process voluntarily relinquishes the processor, the system places the process at the tail of the next lower-level queue. As long as the process uses the full quantum provided at each level, it continues to move to the tail of the next lower queue.

In many multilevel feedback schemes, the scheduler increases a process's quantum size as the process moves to each lower-level queue. Thus, the longer a process has been in the queuing network, the larger the quantum it receives each time it obtains the processor.

One common variation of the multilevel feedback queuing mechanism is to have a process circulate round-robin several times through each queue before it moves to the next lower queue. Also, the number of cycles through each queue may be increased as the process moves to the next lower queue.

Multilevel feedback queuing is a good example of an **adaptive mechanism,** i.e., one that responds to the changing behavior of the system it controls. Adaptive mechanisms generally require more overhead than nonadaptive ones, but the resulting sensitivity to changes in the system makes the system more responsive and helps justify the increased overhead.

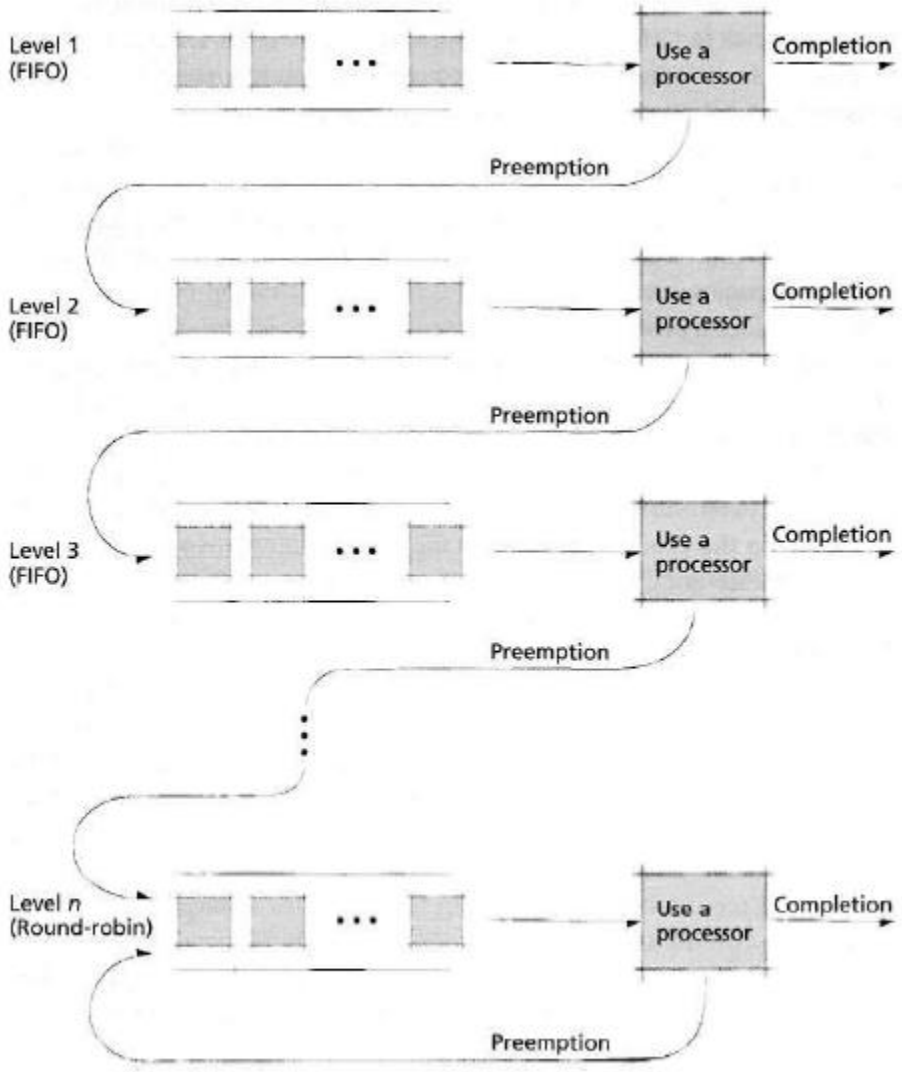


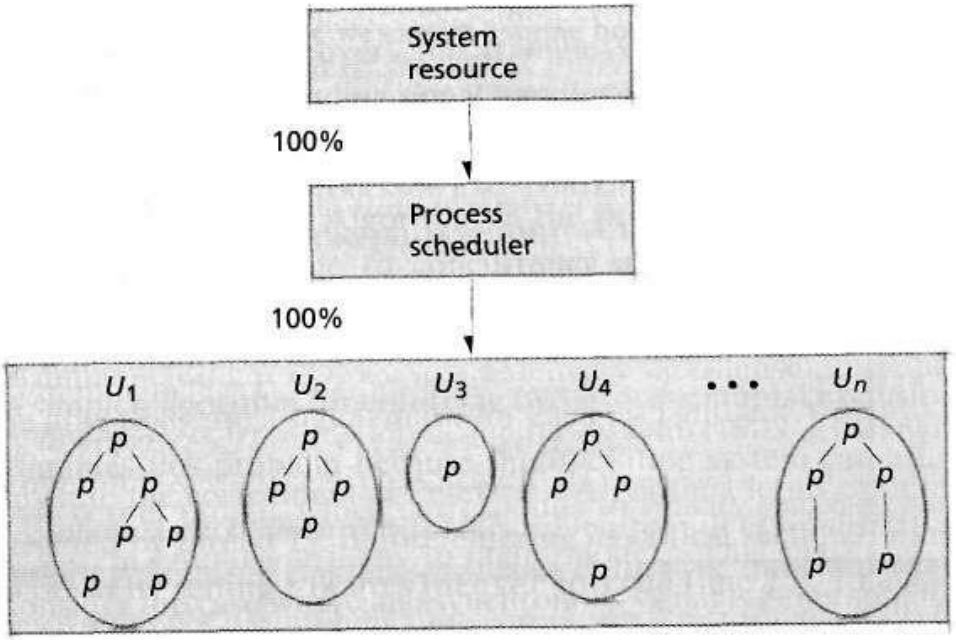
Figure 8.4 | Multilevel feedback queues.

8.7.7 FAIR SHARE SCHEDULING

Fair share scheduling (FSS) supports scheduling across such sets of processes. Fair share scheduling enables a system to ensure fairness across groups of processes by restricting each group to a certain subset of the system resources.

UNIX considers resource-consumption rates across all processes. Under FSS, however, the system apportions the resources to various **fair share groups** . It distributes resources not used by one fair share group to other fair share groups in proportion to their relative needs.

Each process has a priority, and the scheduler associates processes of a given priority with a priority queue for that value. The process scheduler selects the ready process at the head of the highest-priority queue.



Kernel priorities are high and apply to processes executing in the kernel; user priorities are lower. Disk events receive higher priority than terminal events. The scheduler assigns the user priority as the ratio of recent processor usage to elapsed real time; the lower the elapsed time, the higher the priority.

The fair share groups are prioritized by how close they are to achieving their specified resource-utilization goals. Groups doing poorly receive higher priority; groups doing well, lower priority.

UNIT V

DISK PERFORMANCE OPTIMIZATION

OUTLINE

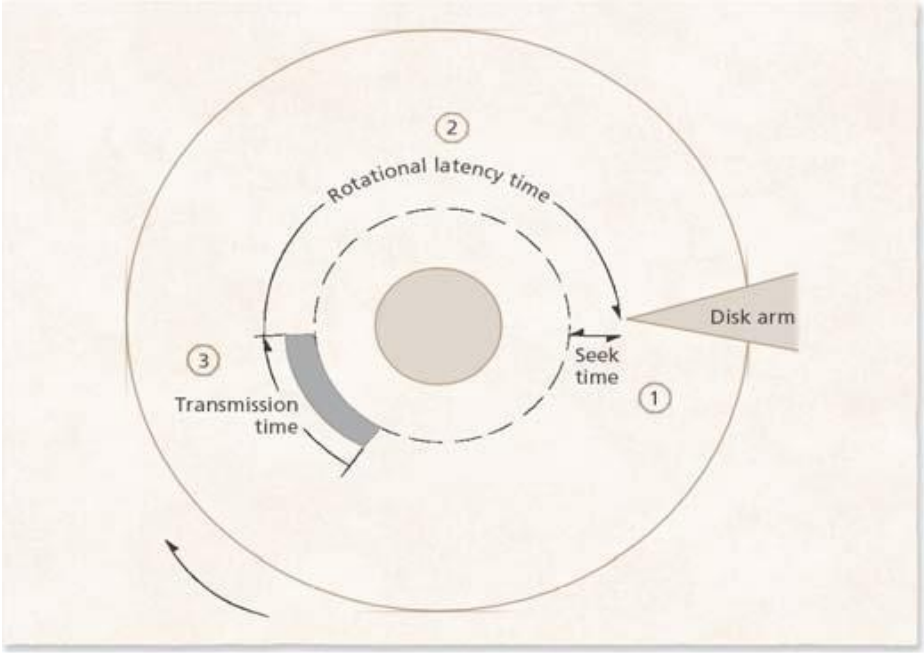
DISK PERFORMANCE OPTIMIZATION

12.1 INTRODUCTION

- Secondary storage is one common bottleneck
  - Improvements in secondary storage performance significantly boost overall system performance
  - Solutions can be both software- and hardware-based.

12.4 WHY DISK SCHEDULING IS NECESSARY

- First-come-first-served (FCFS) scheduling has major drawbacks
  - Seeking to randomly distributed locations results in long waiting times
  - Under heavy loads, system can become overwhelmed
- Requests must be serviced in logical order to minimize delays
  - Service requests with least mechanical motion
- The first disk scheduling algorithms concentrated on minimizing seek times, the component of disk access that had the highest latency
- Modern systems perform rotational optimization as well

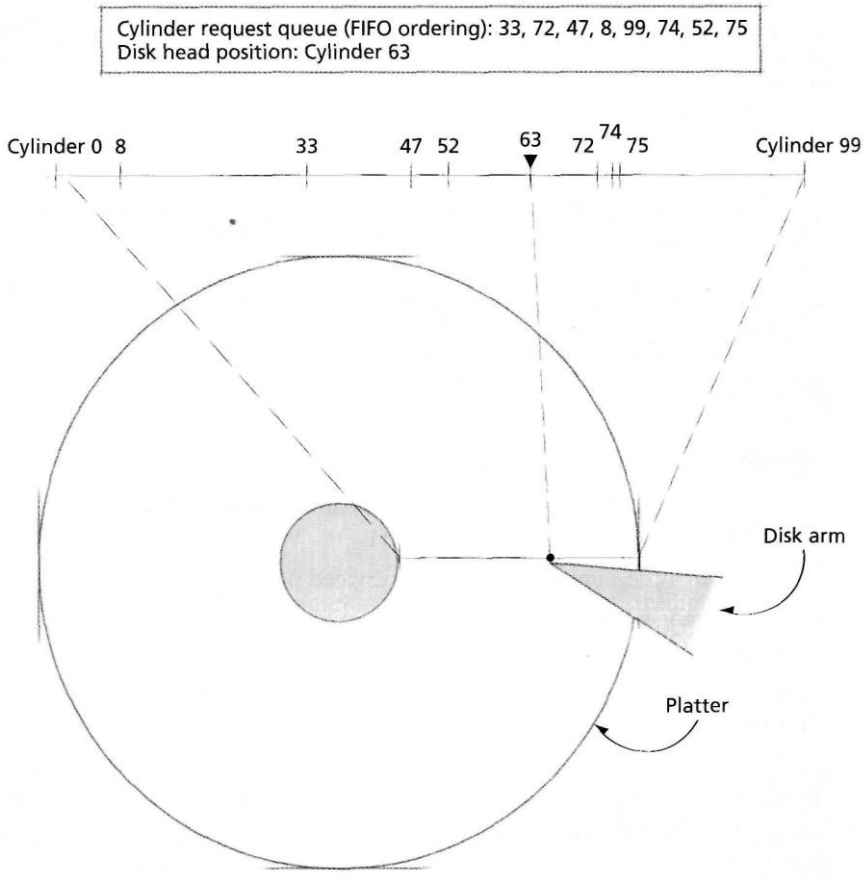


12.5 DISK SCHEDULING STRATEGIES

- A system's disk scheduling strategy depends on the system objectives, but most strategies are evaluated by the following criteria:

  - *Throughput*—the number of requests serviced per unit time
  - ***Mean response time—***the average time spent waiting for a request to be serviced
  - *Variance of response times—&* measure of the predictability of response times.

- Each disk request should be serviced within an acceptable time period.
- To demonstrate the result of each policy on an arbitrary series of requests.
- The arbitrary series of requests is intended to demonstrate how each policy orders disk requests; it does not necessarily indicate the relative performance of each policy in a real system.



In the examples that follow, we assume that the disk contains 100 cylinders, numbered 0-99, and that the read/write head is initially located at cylinder 63, unless stated otherwise.

12.5.1 FIRST-COME-FIRST-SERVED (FCFS) DISK SCHEDULING

FCFS scheduling uses a FIFO queue so that requests are serviced in the order in which they arrive.

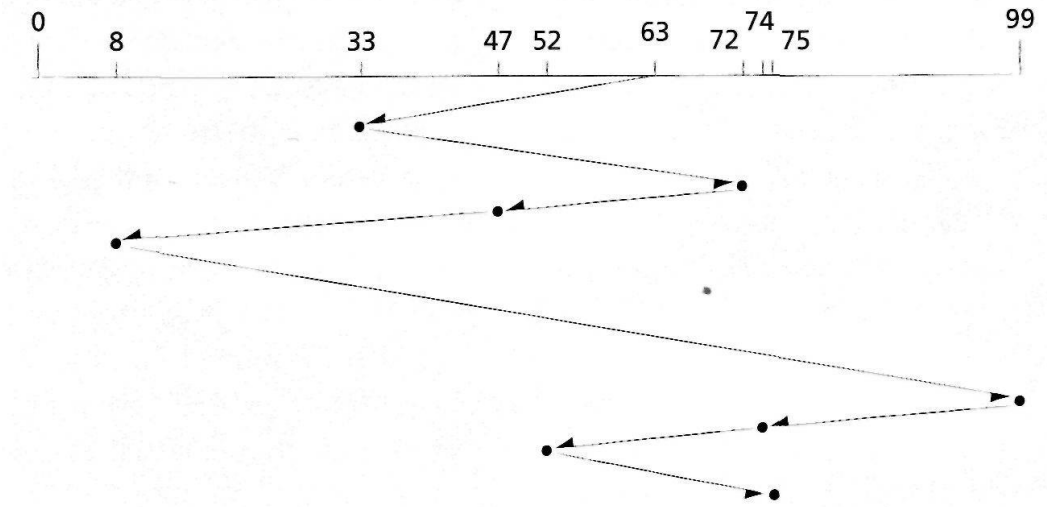
– Advantages

- Fair
- Prevents indefinite postponement
- Low overhead

– Disadvantages

- Potential for extremely low throughput

– FCFS typically results in a random seek pattern because it does not reorder requests to reduce service delays



12.5.2 SHORTEST-SEEK-TIME-FIRST (SSTF) DISK SCHEDULING

Shortest-seek-time-first (SSTF) scheduling next services the request that is closest to the read-write head's current cylinder.

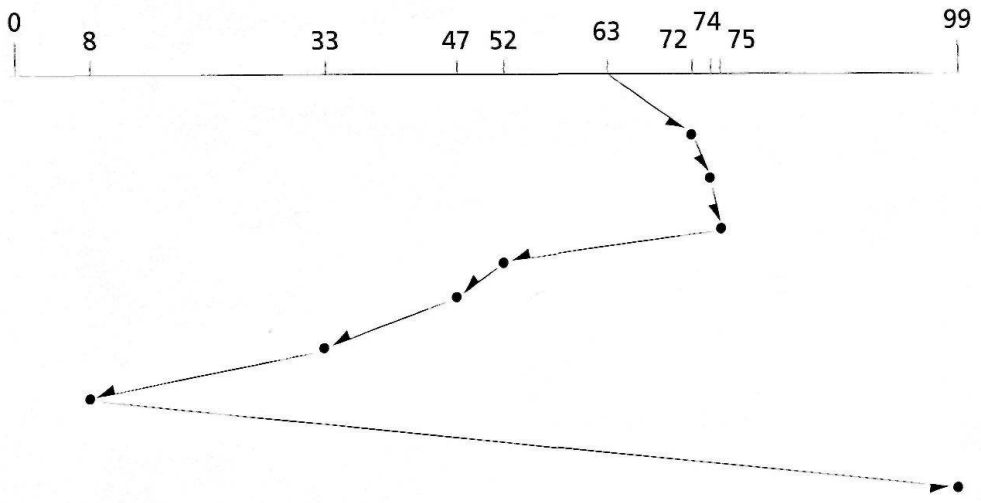
– Advantages

- Higher throughput and lower response times than FCFS
- Reasonable solution for batch processing systems

– Disadvantages

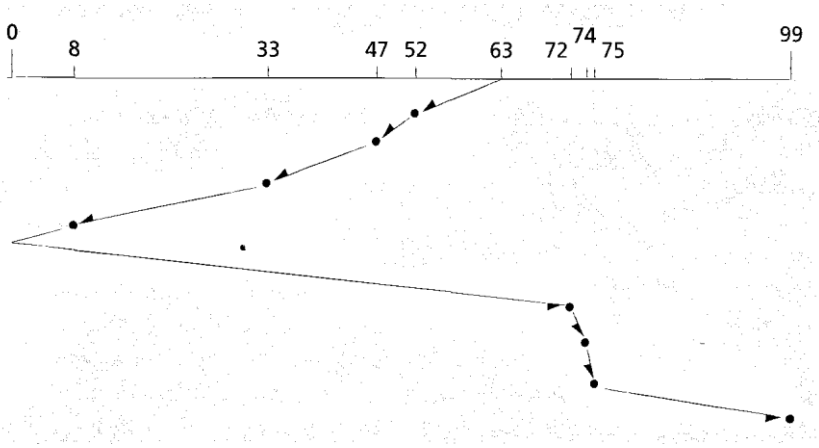
- Does not ensure fairness
- Possibility of indefinite postponement

- High variance of response times
- Response time generally unacceptable for interactive systems



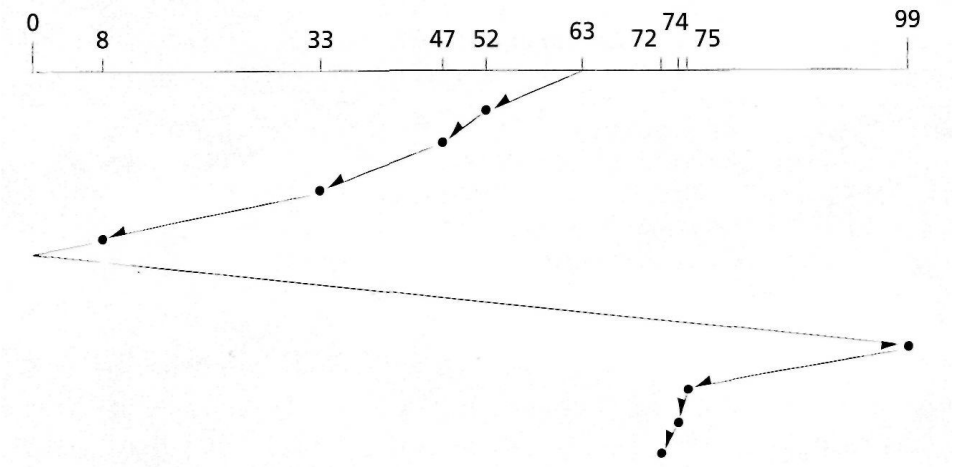
12.5.3 SCAN DISK SCHEDULING

- SCAN chooses the request that requires the shortest seek distance in a **preferred direction.**
- If the preferred direction is currently outward, the SCAN strategy chooses the shortest seek distance in the outward direction.
- SCAN does not change its preferred direction until it reaches the outermost cylinder or the innermost cylinder. In this sense, it is called the **elevator algorithm.**
- SCAN behaves much like SSTF in terms of high throughput and good mean response times. arriving requests can be serviced before waiting requests, both SSTF and SCAN can suffer indefinite postponement.



12.5.4 C-SCAN DISK SCHEDULING

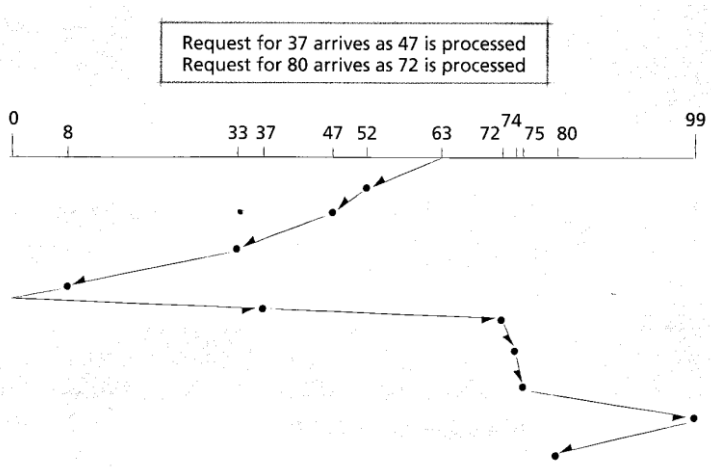
- When the arm has completed its inward sweep, it jumps (without servicing requests) to the outermost cylinder, then resumes its inward sweep, processing requests.
- C-SCAN maintains high levels of throughput while further limiting variance of response times by avoiding discrimination against the innermost and outermost cylinders.



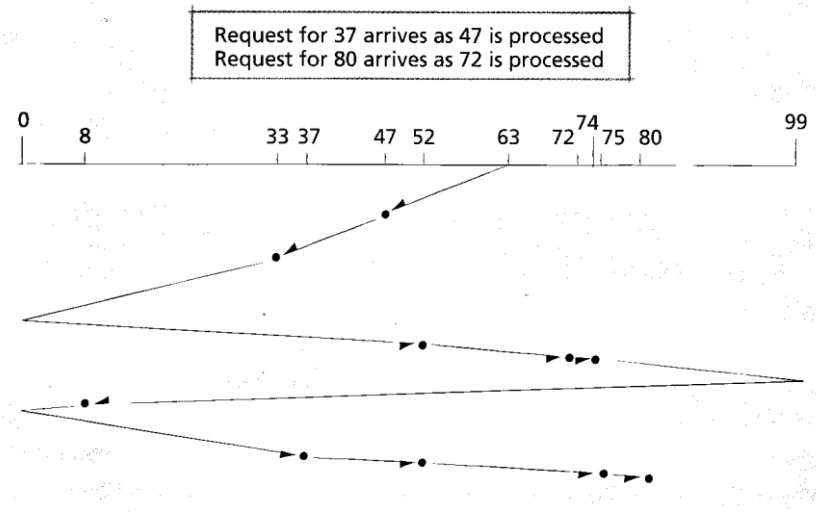
- The best disk scheduling policy might operate in two stages. Under a light load, the SCAN policy is best. Under medium to heavy loads, C-SCAN.
- C-SCAN with rotational optimization handles heavy loading conditions effectively.

12.5.5 FSCAN AND N-STEP SCAN DISK SCHEDULING

- FSCAN uses the SCAN strategy to service only those requests waiting when a particular sweep begins (the "F" stands for "freezing" the request queue at a certain time).
- Requests arriving during a sweep are grouped together and ordered for optimum service during the return sweep.



- N-Step SCAN services the first *n* requests in the queue using the SCAN strategy.
- When *n-1, N*-Step SCAN degenerates to FCFS.
- As *n* approaches infinity, N-Step SCAN degenerates to SCAN.



- FSCAN and N-Step SCAN offer good performance due to high throughput and low mean response times.

12.5.6 LOOK AND C-LOOK DISK SCHEDULING

• **LOOK**: Improvement on SCAN scheduling

– Only performs sweeps large enough to service all requests

• Does move the disk arm to the outer edges of the disk if no requests for those regions are pending

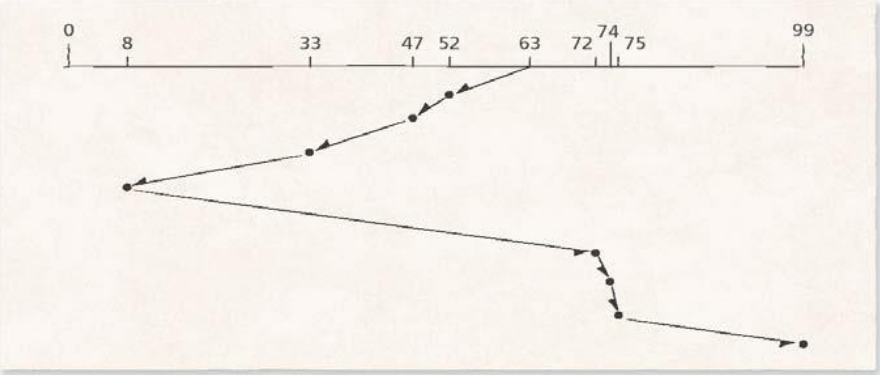
• Improves efficiency by avoiding unnecessary seek operations

• High throughput

• **C-LOOK** improves C-SCAN scheduling

– Combination of LOOK and C-SCAN

– Lower variance of response times than LOOK, at the expense of throughput.



SEEK OPTIMIZATION STRATEGIES SUMMARY

Strategy	*Description*
FCFS	Services requests in the order in which they arrive.
SSTF	Services the request that results in the shortest seek distance first.
SCAN	Head sweeps back and forth across the disk, servicing requests according to SSTF in a preferred direction.
C-SCAN	Head sweeps inward across the disk, servicing requests according to SSTF in the preferred (inward) direction. Upon reaching the innermost track, the head jumps to the outermost track and resumes servicing requests on the next inward pass.
FSCAN	Requests are serviced the same as SCAN, except newly arriving requests are postponed until the next sweep. Avoids indefinite postponement.
N-Step SCAN	Services requests as in FSCAN, but services only *n* requests per sweep. Avoids indefinite postponement.
LOOK	Same as SCAN except the head changes direction upon reaching the last request in the preferred direction.
C-LOOK	Same as C-SCAN except the head stops after servicing the last request in the preferred direction, then services the request to the cylinder nearest the opposite side of the disk.

12.6 ROTATIONAL OPTIMIZATION

• Seek time formerly dominated performance concerns

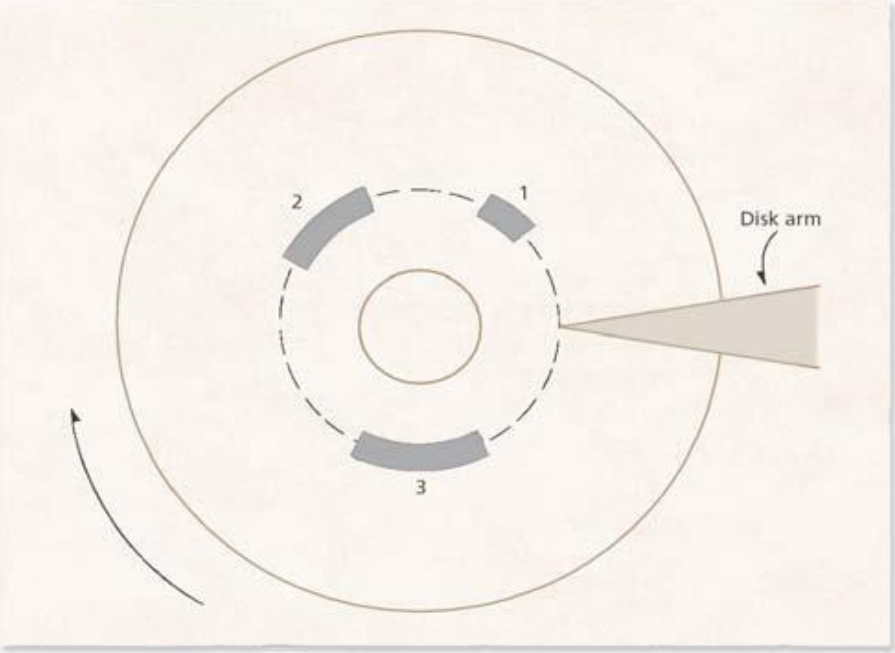
– Today, seek times and rotational latency are the same order of magnitude

• Recently developed strategies attempt to optimization disk performance by reducing rotational latency

• Important when accessing small pieces of data distributed throughout the disk surfaces

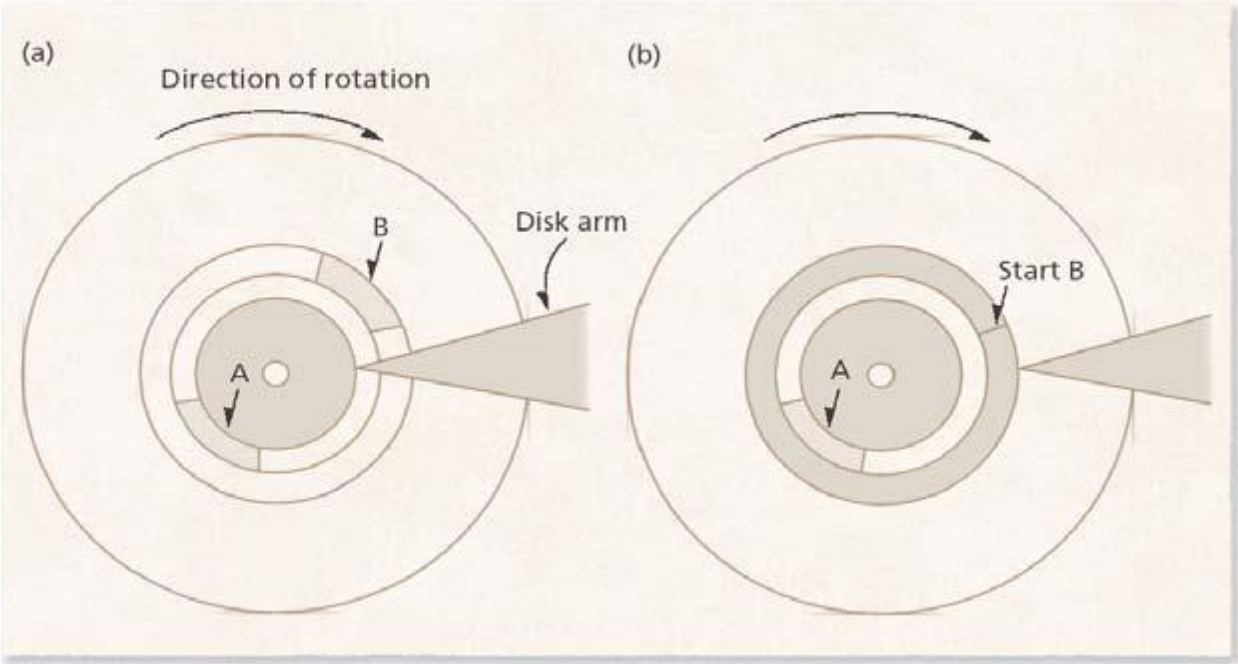
12.6.1 SLTF SCHEDULING

- Shortest-latency-time-first scheduling
  - On a given cylinder, service request with shortest rotational latency first
  - Easy to implement
  - Achieves near-optimal performance for rotational latency



12.6.2 SPTF AND SATF SCHEDULING

- Shortest-positioning-time-first scheduling
  - Positioning time: Sum of seek time and rotational latency
  - SPTF first services the request with the shortest positioning time
  - Yields good performance
  - Can indefinitely postpone requests



- Shortest-access-time-first scheduling
  - Access time: positioning time plus transmission time
  - High throughput
  - Again, possible to indefinitely postpone requests
  - Both SPTF and SATF can implement LOOK to improve performance
  - Weakness
    - Both SPTF and SATF require knowledge of disk performance characteristics which might not be readily available due to error correcting data and transparent reassignment of bad sectors

FILE AND DATABASE SYSTEMS

OUTLINE

FILE AND DATABASE SYSTEMS

13.1 INTRODUCTION

- Files
  - – Named collection of data that is manipulated as a unit
  - – Reside on secondary storage devices
- Operating systems can create an interface that facilitates navigation of a user's files
  - – File systems can protect such data from corruption or total loss from disasters
  - – Systems that manage large amounts of shared data can benefit from databases as an alternative to files.

13.2 DATA HIERARCHY

- Information is stored in computers according to a data hierarchy.
- Lowest level of data hierarchy is composed of bits
  - – Bit patterns represent all data items of interest in computer systems
- Next level in the data hierarchy is fixed-length patterns of bits such as bytes, characters and words
  - – Byte: typically 8 bits
  - – Word: the number of bits a processor can operate on at once
  - – Characters map bytes (or groups of bytes) to symbols such as letters, numbers, punctuation and new lines
- Three most popular character sets in use today: ASCII, EBCDIC and Unicode
  - – Field: a group of characters
  - – Record: a group of fields
  - – File: a group of related records
- Highest level of the data hierarchy is a file system or database
- A volume is a unit of data storage that may hold multiple files.

13.3 FILES

A file is a named collection of data that may be manipulated as a unit by operations such as

- **open** —Prepare a file to be referenced.

- **close** —Prevent further reference to a file until it is reopened.
- **create** — Create a new file.
- **destroy** —Remove a file.
- **copy—**Copy the contents of one file to another.
- **rename**—Change the name of a file.
- **list—**Print or display the contents of a file.

Individual data items within the file may be manipulated by operations like

- **read—**Copy data from a file to a process's memory.
- **write** —Copy data from a process's memory to a file.
- **update—**Modify an existing data item in a file.
- **insert—**Add a new data item to a file.
- **delete** —Remove a data item from a file.

Files may be characterized by attributes such as

- size —the amount of data stored in the file.
- **location—**the location of the file (in a storage device or in the system's logical file organization).
- **accessibility—**restrictions placed on access to file data.
- **type—**how the file data is used. For example, an executable file contains machine instructions for a process. A data file may specify the application that is used to access its data.
- **volatility—**the frequency with which additions and deletions are made to a file.
- **activity—**the percentage of a file's records accessed during a given period of time.

13.4 FILE SYSTEMS

A file system organizes files and manages access to data.3 File systems are responsible for:

- **File management—**providing the mechanisms for files to be stored, referenced, shared, and secured.
- **Auxiliary storage management** —allocating space for files on secondary or tertiary storage devices.

• **File integrity mechanisms** —ensuring that the information stored in a file is uncorrupted. When file integrity is assured, files contain only the information that they are intended to have.

• **Access methods—**how the stored data can be accessed.

The mechanism tor sharing files should provide various types of controlled access such as **read access, write access, execute access** or various combinations of these.

• **File system characteristics**

– Should exhibit device independence:

• Users should be able to refer to their files by symbolic names rather than having to use physical device names

– Should also provide backup and recovery capabilities to prevent either accidental loss or malicious destruction of information

– May also provide encryption and decryption capabilities to make information useful only to its intended audience

13.4.1 DIRECTORIES

• Directories:

– Files containing the names and locations of other files in the file system, to organize and quickly locate files

• Directory entry stores information such as:

– File name

– Location

– Size

– Type

– Accessed

– Modified and creation times

Directory Field	*Description*
Name	Character string representing the file's name.
Location	Physical block or logical location of the file in the file system (i.e., a pathname).
Size	Number of bytes consumed by the file.
Type	Description of the file's purpose (e.g., data file or directory file).
Access time	Time the file was last accessed.
Modified time	Time the file was last modified.
Creation time	Time the file was created.

• Single-level (or flat) file system:

– Simplest file system organization

– Stores all of its files using one directory

– No two files can have the same name

– File system must perform a linear search of the directory contents to locate each file, which can lead to poor performance

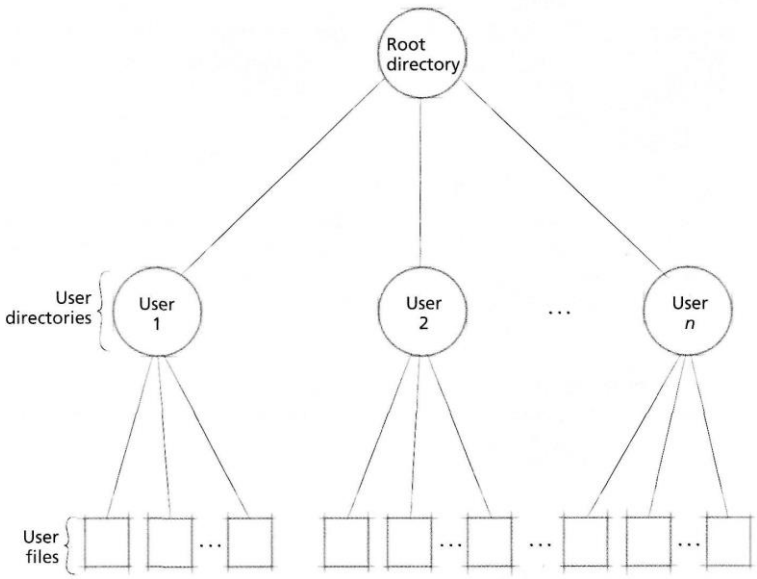
• Hierarchical file system:

– A root indicates where on the storage device the root directory begins

– The root directory points to the various directories, each of which contains an entry for each of its files

– File names need be unique only within a given user directory

– The name of a file is usually formed as the pathname from the root directory to the file

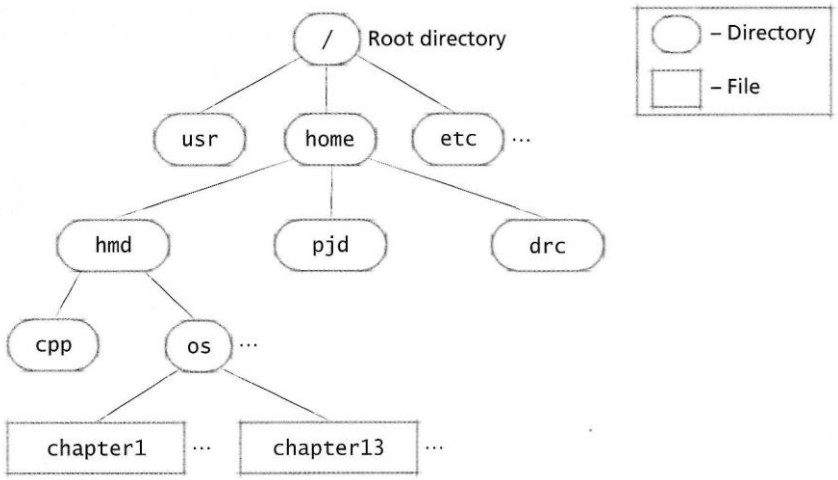


• Working directory

– Simplifies navigation using pathnames

– Enables users to specify a pathname that does not begin at the root directory (i.e., a relative path)

– Absolute path (i.e., the path beginning at the root) = working directory + relative path



• Link: a directory entry that references a data file or directory located in a different directory

– Facilitates data sharing and can make it easier for users to access files located throughout a file system's directory structure

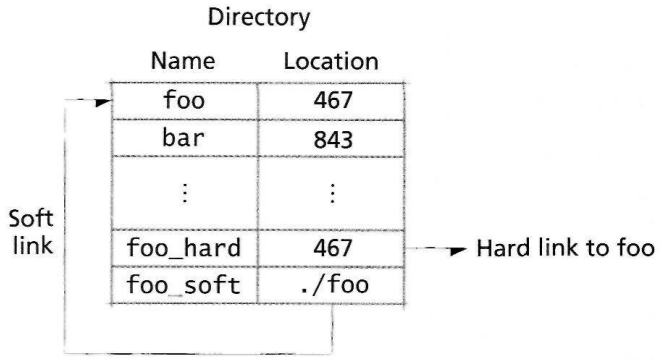
– Soft link: directory entry containing the pathname for another file

– Hard link: directory entry that specifies the location of the file (typically a block number) on the storage device

– Because a hard link specifies a physical location of a file, it references invalid data when the physical location of its corresponding file changes

– Because soft links store the logical location of the file in the file system, they do not require updating when file data is moved

– However, if a user moves a file to different directory or renames the file, any soft links to that file are no longer valid

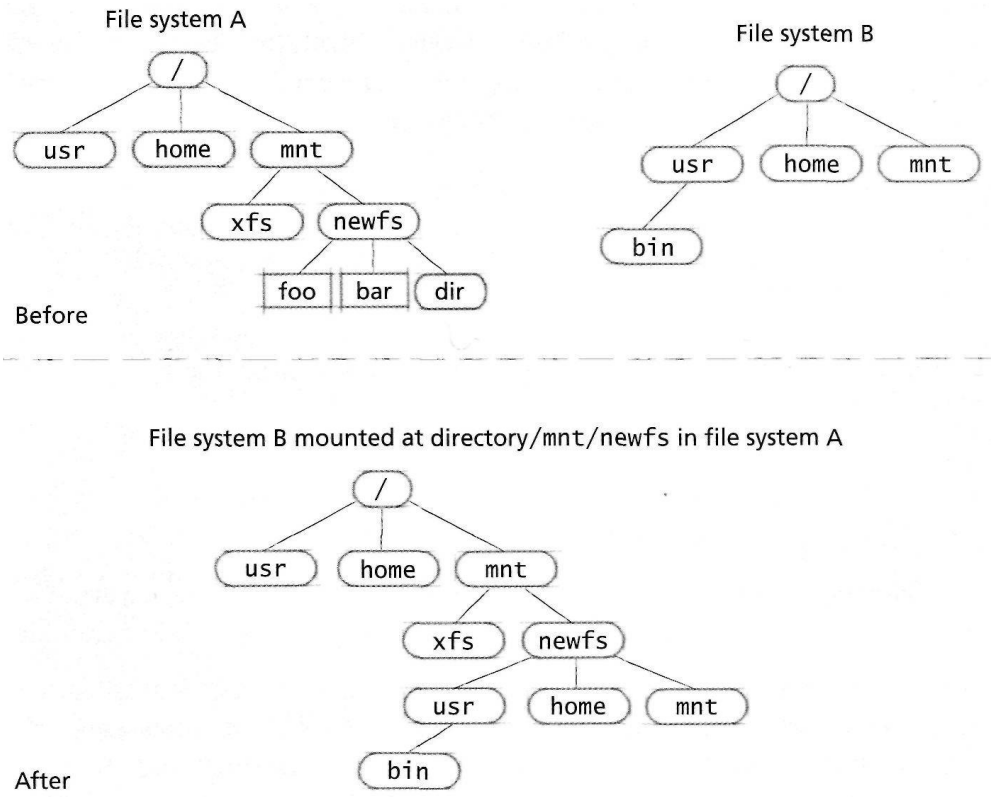


13.4.2 METADATA

- Metadata
  - Information that protects the integrity of the file system
  - Cannot be modified directly by users
- Many file systems create a superblock to store critical information that protects the integrity of the file system
  - A superblock might contain:
- The file system identifier
- The location of the storage device's free blocks
  - To reduce the risk of data loss, most file systems distribute redundant copies of the superblock throughout the storage device
- File open operation returns a file descriptor
  - A non-negative integer index into the open-file table
- From this point on, access to the file is directed through the file descriptor
- To enable fast access to file-specific information such as permissions, the open-file table often contains file control blocks, also called file attributes:
  - Highly system-dependent structures that might include the file's symbolic name, location in secondary storage, access control data and so on

13.4.3 MOUNTING

- Mount operation
  - Combines multiple file systems into one namespace so that they can be referenced from a single root directory
  - Assigns a directory, called the mount point, in the native file system to the root of the mounted file system
- File systems manage mounted directories with mount tables:
  - Contain information about the location of mount points and the devices to which they point
- When the native file system encounters a mount point, it uses the mount table to determine the device and type of the mounted file system
- Users can create soft links to files in mounted file systems but cannot create hard links between file systems



13.5 FILE ORGANIZATION:

• **Sequential—**Records are placed in physical order. The "next" record is the one that physically follows the previous record. This organization is natural for files stored on magnetic tape, an inherently sequential medium.

• **Direct—**Records are directly (randomly) accessed by their physical addresses on a direct access storage device (DASD).

• **Indexed sequential—**Records on disk are arranged in logical sequence according to a key contained in each record.

• **Partitioned—**This is essentially a file of sequential sub files. Each sequential subfile is called a **member.** The starting address of each member is stored in the file's directory. Partitioned files have been used to store program libraries or macro libraries.

13.6 FILE ALLOCATION

• File allocation

– Problem of allocating and freeing space on secondary storage is somewhat like that experienced in primary storage allocation under variable-partition multiprogramming

– Contiguous allocation systems have generally been replaced by more dynamic noncontiguous allocation systems

• Files tend to grow or shrink over time

• Users rarely know in advance how large their files will be used.

13.6.1 CONTIGUOUS FILE ALLOCATION

• Contiguous allocation

– Place file data at contiguous addresses on the storage device

– Advantages

• Successive logical records typically are physically adjacent to one another

– Disadvantages

• External fragmentation

• Poor performance can result if files grow and shrink over time

• If a file grows beyond the size originally specified and no contiguous free blocks are available, it must be transferred to a new area of adequate size, leading to additional I/O operations.

13.6.2 LINKED-LIST NONCONTIGUOUS FILE ALLOCATION

• Sector-based linked-list noncontiguous file allocation scheme:

– A directory entry points to the first sector of a file

• The data portion of a sector stores the contents of the file

• The pointer portion points to the file's next sector

– Sectors belonging to a common file form a linked list

• When performing block allocation, the system allocates blocks of contiguous sectors (sometimes called extents)

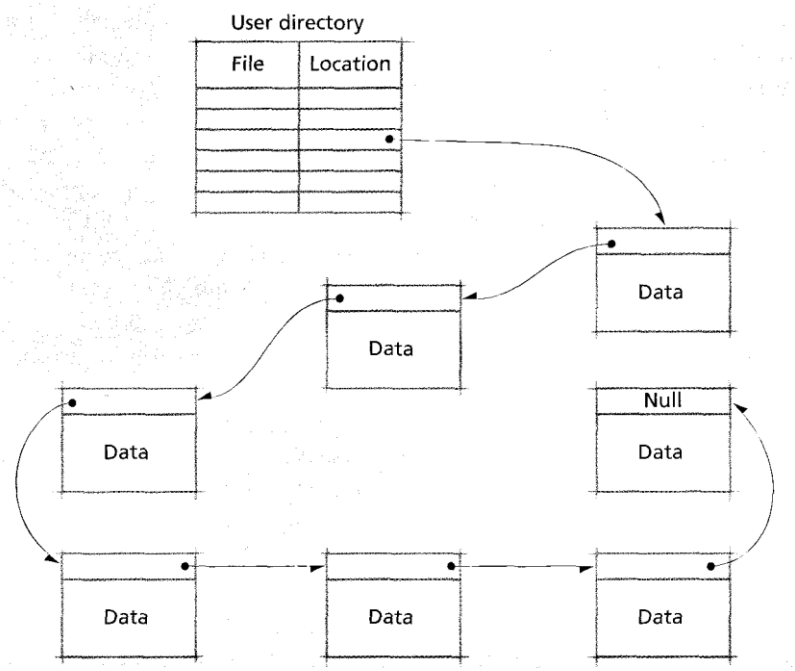
• Block chaining

– Entries in the user directory point to the first block of each file

– File blocks contain:

• A data block

• A pointer to the next block

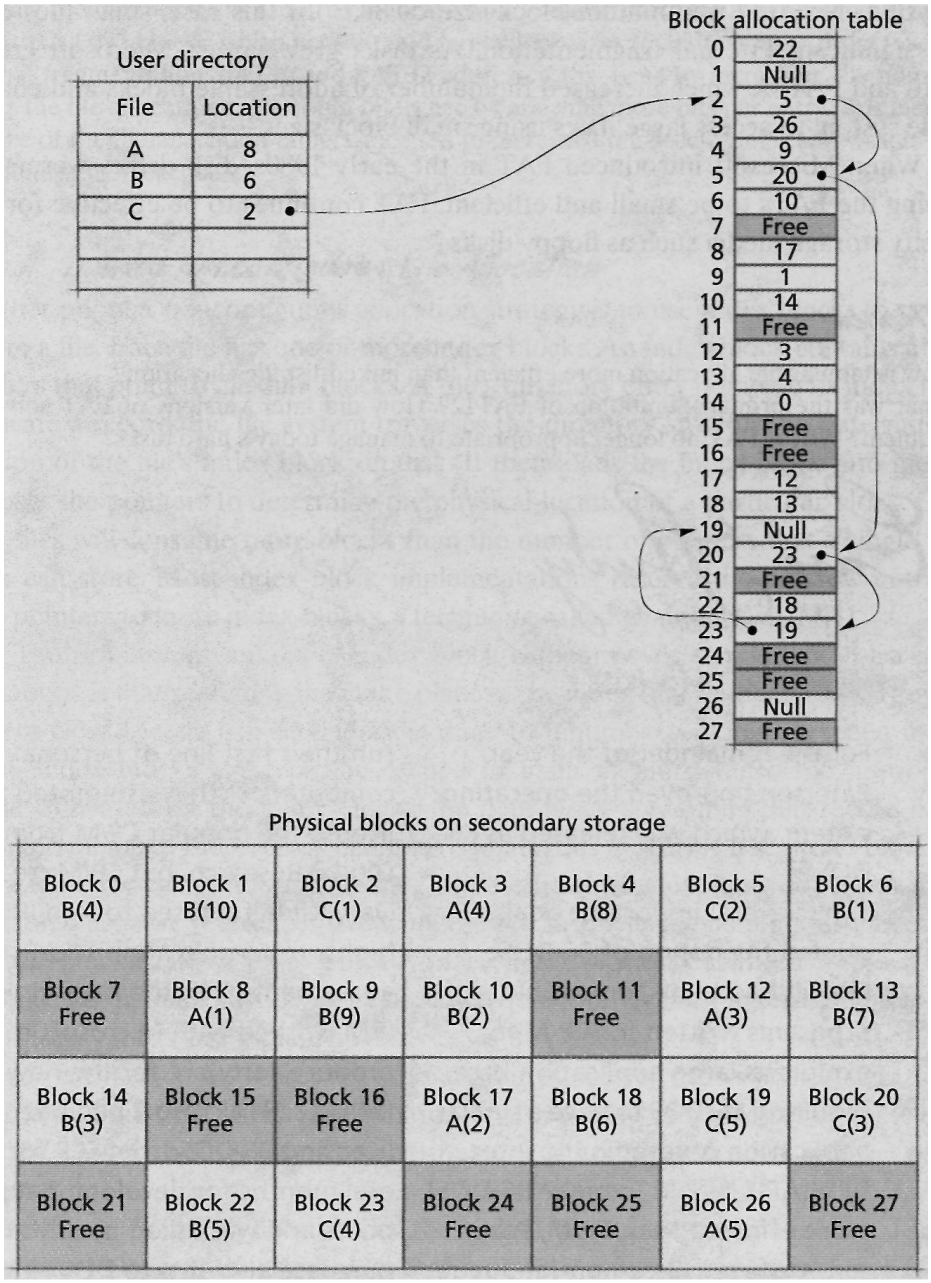


- When locating a record
  - The chain must be searched from the beginning
  - If the blocks are dispersed throughout the storage device (which is normal), the search process can be slow as block-to-block seeks occur
- Insertion and deletion are done by modifying the pointer in the previous block
- Large block sizes
  - Can result in significant internal fragmentation
- Small block sizes
  - May cause file data to be spread across multiple blocks dispersed throughout the storage device
  - Poor performance as the storage device performs many seeks to access all the records of a file

13.6.3 TABULAR NONCONTIGUOUS FILE ALLOCATION

- Uses tables storing pointers to file blocks
- Reduces the number of lengthy seeks required to access a particular record
  - Directory entries indicate the first block of a file
  - Current block number is used as an index into the block allocation table to determine the location of the next block.
- If the current block is the file's last block, then its block allocation

table entry is null



- Pointers that locate file data are stored in a central location
  - The table can be cached so that the chain of blocks that compose a file can be traversed quickly
  - Improves access times
- To locate the last record of a file, however:
  - The file system might need to follow many pointers in the block allocation table
  - Could take significant time
- When a storage device contains many blocks:

– The block allocation table can become large and fragmented

– Reduces file system performance

• A popular implementation of tabular noncontiguous file allocation is Microsoft's FAT file system.

13.6.4 INDEXED NONCONTIGUOUS FILE ALLOCATION

• Indexed noncontiguous file allocation:

– Each file has an index block or several index blocks

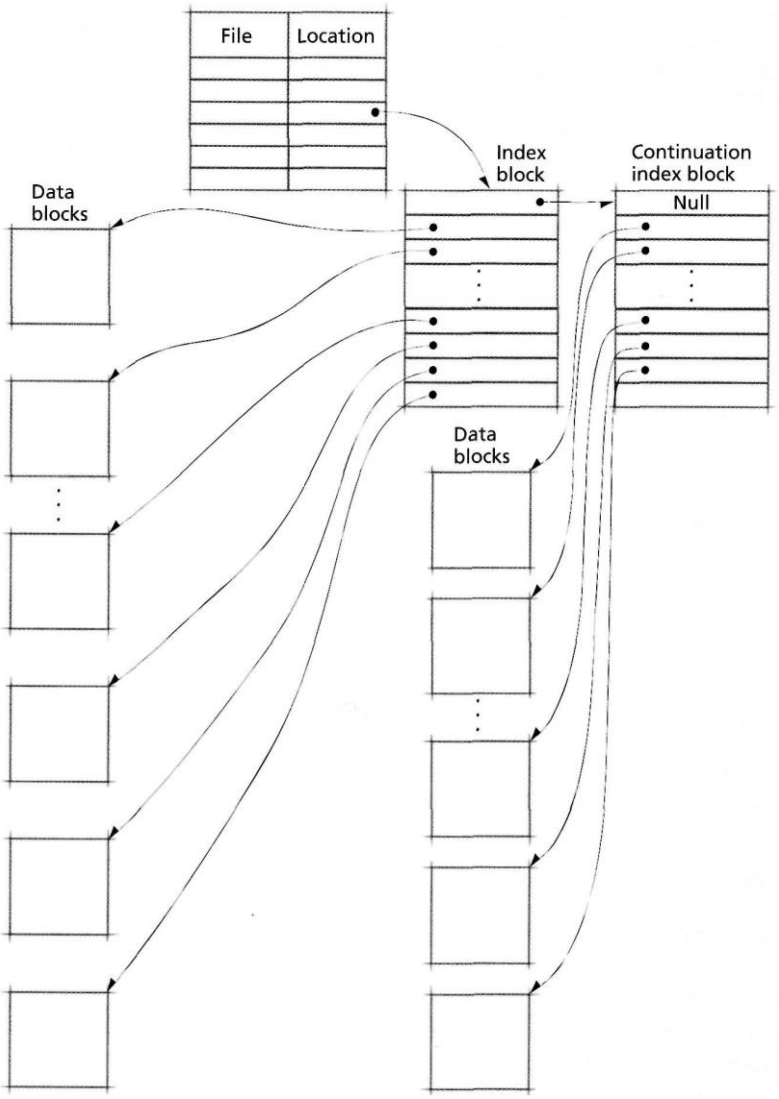
– Index blocks contain a list of pointers that point to file data blocks

– A file's directory entry points to its index block, which may reserve the last few entries to store pointers to more index blocks, a technique called chaining

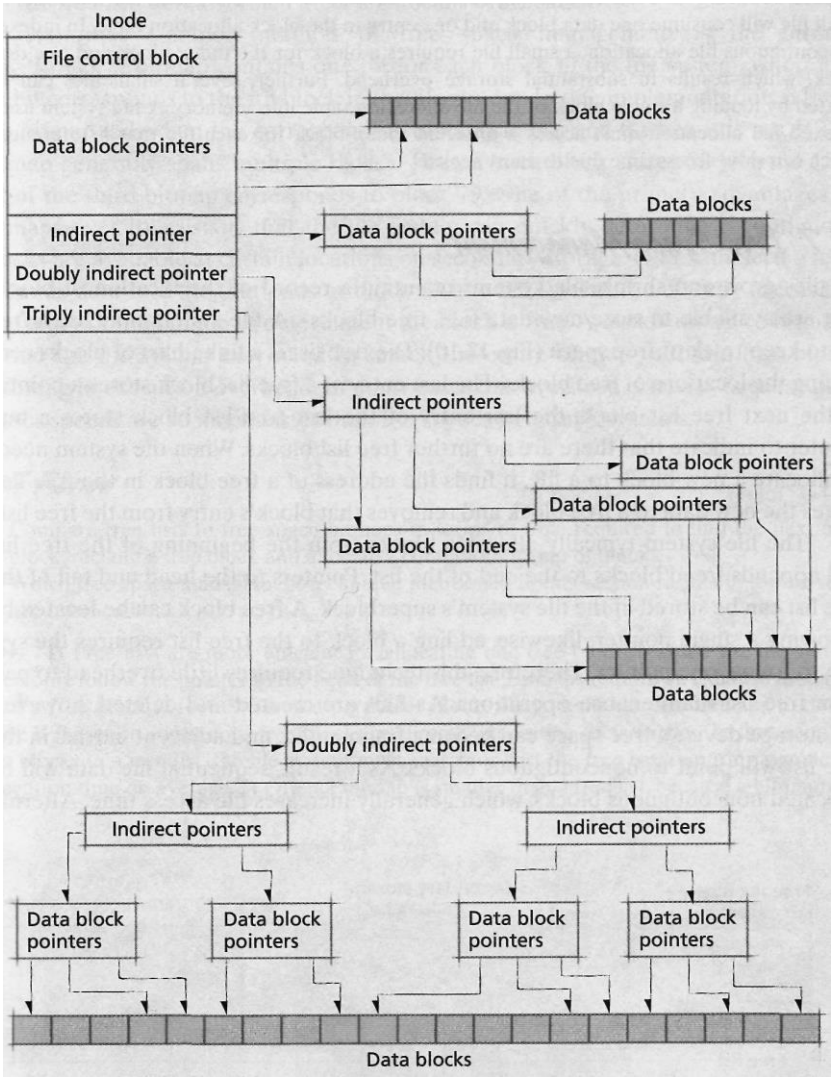
• Primary advantage of index block chaining over simple linked-list implementations:

– Searching may take place in the index blocks themselves.

– File systems typically place index blocks near the data blocks they reference, so the data blocks can be accessed quickly after their index block is loaded.

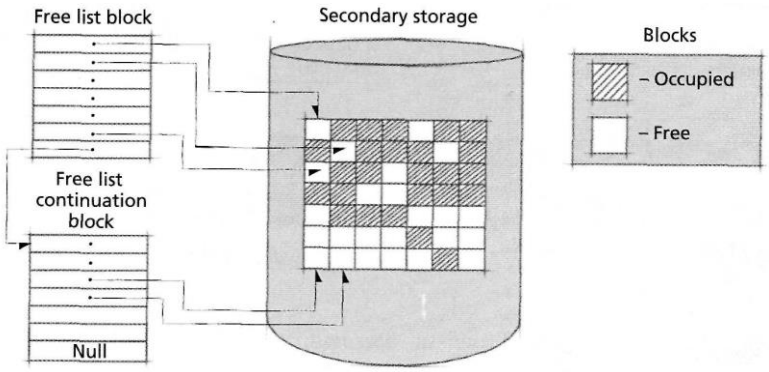


- Index blocks are called inodes (i.e., index nodes) in UNIX-based operating systems

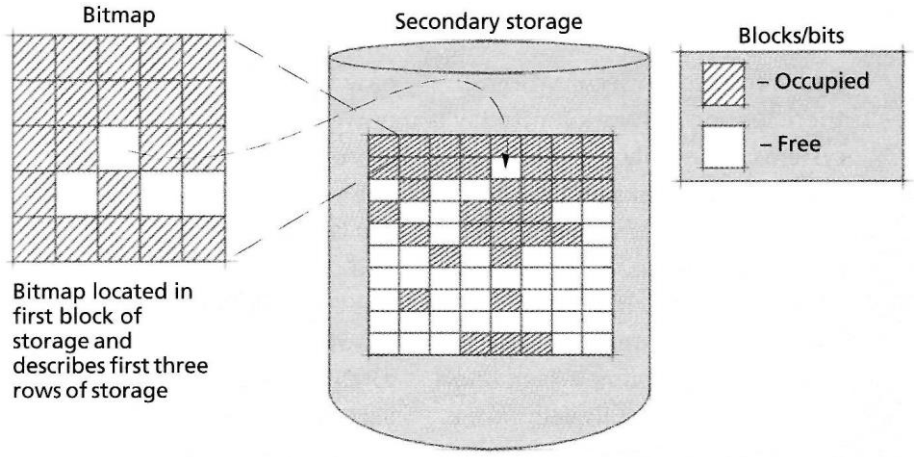


13.7 FREE SPACE MANAGEMENT

- Some systems use a free list to manage the storage device's free space
  - Free list: Linked list of blocks containing the locations of free blocks
  - Blocks are allocated from the beginning of the free list
  - Newly freed blocks are appended to the end of the list
- Low overhead to perform free list maintenance operations
- Files are likely to be allocated in noncontiguous blocks
  - Increases file access time



- A bitmap contains one bit for each block in memory
  - – *i*th bit corresponds to the *i*th block on the storage device
- Advantage of bitmaps over free lists:
  - – The file system can quickly determine if contiguous blocks are available at certain locations on secondary storage
- Disadvantage of bitmaps:
  - – The file system may need to search the entire bitmap to find a free block, resulting in substantial execution overhead



13.8 FILE ACCESS CONTROL

- Files are often used to store sensitive data such as:
  - – Credit card numbers
  - – Passwords
  - – Social security numbers
- Therefore, they should include mechanisms to control user access to data.
  - – Access control matrix
  - – Access control by user classes

13.8.1 ACCESS CONTROL MATRIX

- Two-dimensional access control matrix:
  - Entry a*ij* is 1 if user *i* is allowed access to file *j*
  - Otherwise a*ij* = 0
- In an installation with a large number of users and a large number of files, this matrix generally would be large and sparse
- Inappropriate for most systems

User \ File	1	2	3	4	5	6	7	8	9	10
1	1	1	0	0	0	0	0	0	0	0
2	0	0	1	0	1	0	0	0	0	0
3	0	1	0	1	0	1	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0
5	1	1	1	1	1	1	1	1	1	1
6	0	0	0	0	0	1	1	0	0	0
7	1	0	0	0	0	0	0	0	0	1
8	1	0	0	0	0	0	0	0	0	0
9	1	1	1	1	0	0	0	0	1	1
10	1	1	0	0	1	1	0	0	0	1

13.8.2 ACCESS CONTROL BY USER CLASSES

- A technique that requires considerably less space is to control access to various user classes
- User classes can include:
  - The file owner
  - A specified user
  - Group
  - Project
  - Public
- Access control data
  - Can be stored as part of the file control block
  - Often consumes an insignificant amount of space

UNIT IV

REAL MEMORY ORGANIZATION AND MANAGEMENT

OUTLINE

REAL MEMORY ORGANIZATION AND MANAGEMENT

9.1 INTRODUCTION

- The organization and management of the real memory (also called main memory, physical memory or primary memory) of a computer system has been a major influence on operating systems design.
- Secondary storage—most commonly disk and tape - provides massive, inexpensive capacity for the abundance of programs and data that must be kept readily available for processing. Main memory requires careful management.

9.2 MEMORY ORGANIZATION

- Main memory is still relatively expensive compared to secondary storage.
- Also, today's operating systems and applications require ever more substantial quantities (Fig. 9.1).
- For example, Microsoft recommends 256MB of main memory to efficiently run Windows XP Professional.

Operating System	*Release Date*	*Minimum Memory Requirement*	***Recommended Memory***
Windows 1.0	November 1985	256KB	
Windows 2.03	November 1987	320KB	
Windows 3.0	March 1990	896KB	1MB
Winndows 3.1	April 1992	2.6MB	4MB
Windows 95	August 1995	8MB	16MB
Windows NT 4.0	August 1996	32MB	96MB
Windows 98	June 1998	24MB	64MB
Windows ME	September 2000	32MB	128MB
Windows 2000 Professional	February 2000	64MB	128MB
Windows XP Home	October 2001	64MB	128MB
Windows XP Professional	October 2001	128MB	256MB

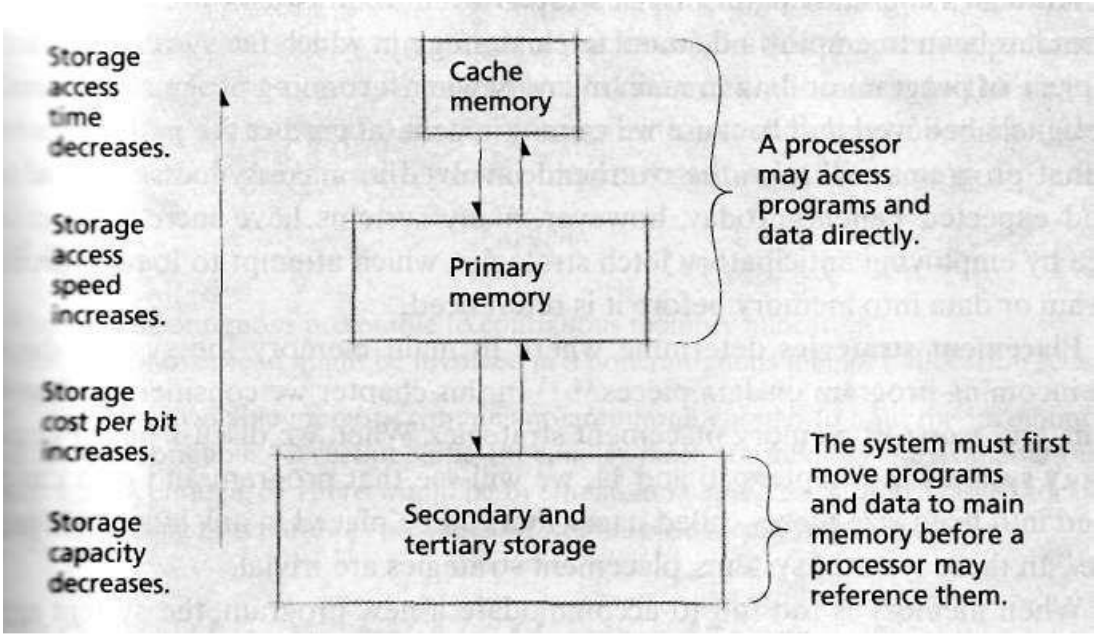
9.3 MEMORY MANAGEMENT

- **Memory management strategies** determine how a particular memory Organization performs under various loads.
- Memory management is typically performed by both software and special-purpose hardware.

- The memory manager determines how available memory space is allocated to processes and how to respond to changes in a process's memory usage.
- It also interacts with special-purpose memory management hardware (if any is available) to improve performance.
    - Performed by memory manager
        - Which process will stay in memory?
        - How much memory will each process have access to?
        - Where in memory will each process go?

9.4 MEMORY HIERARCHY

- Main memory
    - Should store currently needed program instructions and data only
- Secondary storage
    - Stores data and programs that are not actively needed
- Cache memory
    - Extremely high speed
    - Usually located on processor itself
    - Most-commonly-used data copied to cache for faster access
    - Small amount of cache still effective for boosting performance
- Due to temporal locality



9.5 MEMORY MANAGEMENT STRATEGIES

They are divided into:

1. Fetch strategies
2. Placement strategies
3. Replacement strategies

Fetch strategies determine when to move the next piece of a program or data to main memory from secondary storage. We divide them into two types.

i. **Demand fetch strategies**
ii. **Anticipatory fetch strategies.**

Placement strategies determine where in main memory the system should place incoming program or data pieces. We consider the **first-fit**, **best-fit,** and **worst-fit** memory placement strategies.

Replacement strategies

- When memory is too full to accommodate a new program, the system must remove some (or all) of a program or data that currently resides in memory.
- The system's **replacement strategy** determines which piece to remove.

9.6 CONTIGUOUS VS. NONCONTIGUOUS MEMORY ALLOCATION

• Ways of organizing programs in memory

– Contiguous allocation

- • Program must exist as a single block of contiguous addresses
- • Sometimes it is impossible to find a large enough block
- • Low overhead

– Noncontiguous allocation

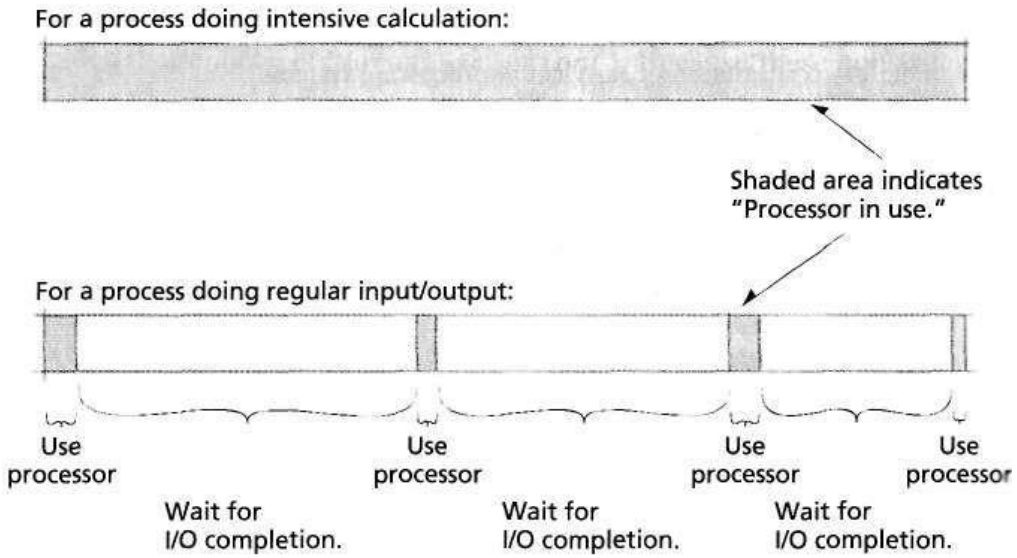
- • Program divided into chunks called segments
- • Each segment can be placed in different part of memory
- • Easier to find “holes” in which a segment will fit
- • Increased number of processes that can exist simultaneously in

Memory offsets the overhead incurred by this technique.

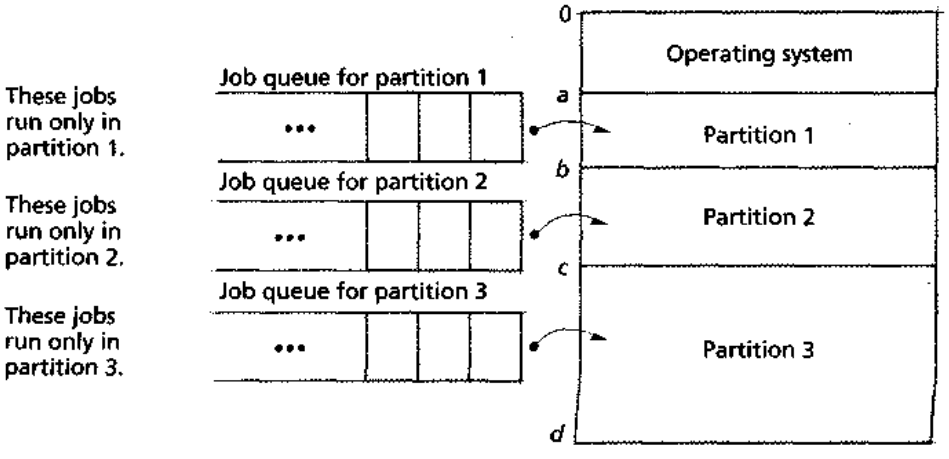
9.8 FIXED-PARTITION MULTIPROGRAMMING

- A typical process would consume the processor time it needed to generate an input/output request; the process could not continue until the I/O finished.

- To take maximum advantage of multiprogramming, several processes must reside in the computer's main memory at the same time.
- Thus, when one process requests input/output, the processor may switch to another process and continue to perform calculations without the delay.

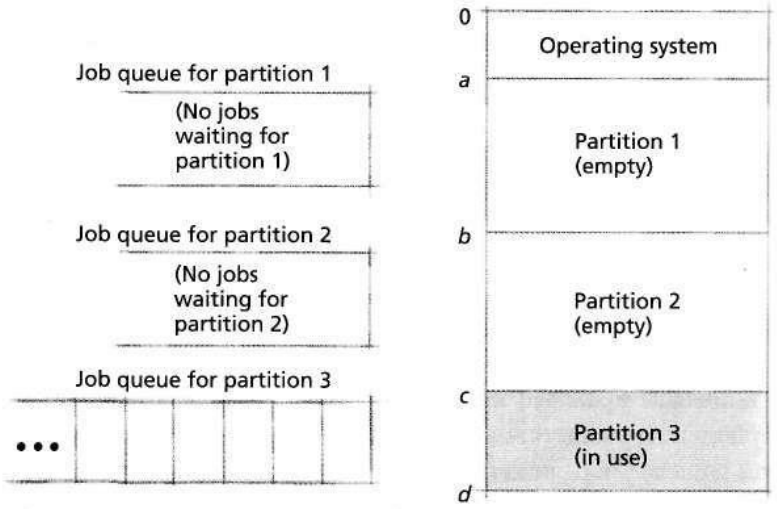


- The system divides main memory into a number of fixed size partitions.
- Each partition holds a single job, and the system switches the processor rapidly between jobs to create the illusion of simultaneity.
- This technique enables the system to provide simple multiprogramming capabilities.

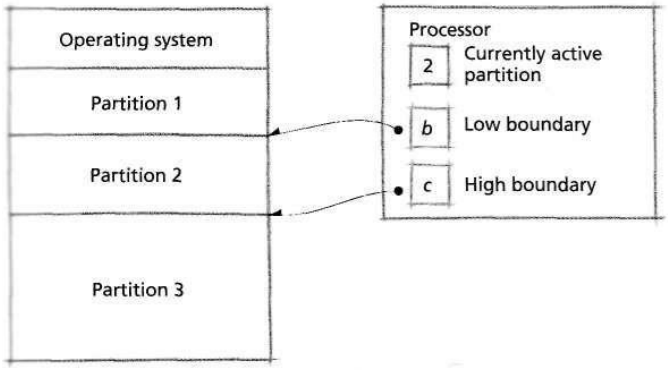


- This restriction led to wasted memory.

- If a job was ready to run and the program's partition was occupied, then that job had to wait, even if other partitions were available.



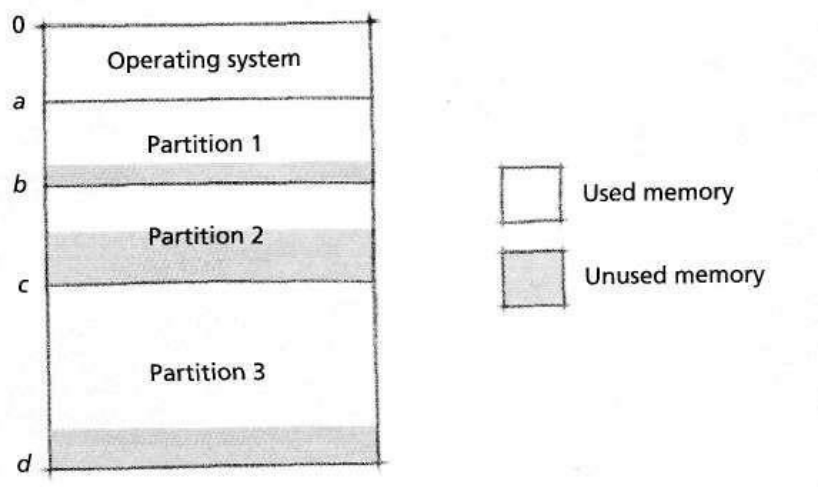
- All the jobs in the system must run in partition 3 (i.e., the programs' instructions all begin at address c).
- Because this partition currently is in use. all other jobs are forced to wait, even though the system has two other partitions in which the jobs could run (if they had been compiled for these partitions).
- This scheme eliminates some of the memory waste inherent in multipro- operating system from the user process.
- In a multiprogramming system, the system must protect the operating system from all user processes and protect each process from all the others.



- The system can delimit each partition with two boundary registers low and high, also called the **base** and **limit** registers.
- When a process issues a memory request, the system checks whether the requested address is greater than or equal to the process's low boundary register value and less than the process's high boundary register value (see the Anecdote, Compartmentalization).
- If so, the system honors the request; otherwise, the system terminates the program with an error message.

Fixed-partition multiprogramming suffers from **internal frag**mentation, which occurs when the size of a process's memory and data is smaller than that of the partition in which the process executes.

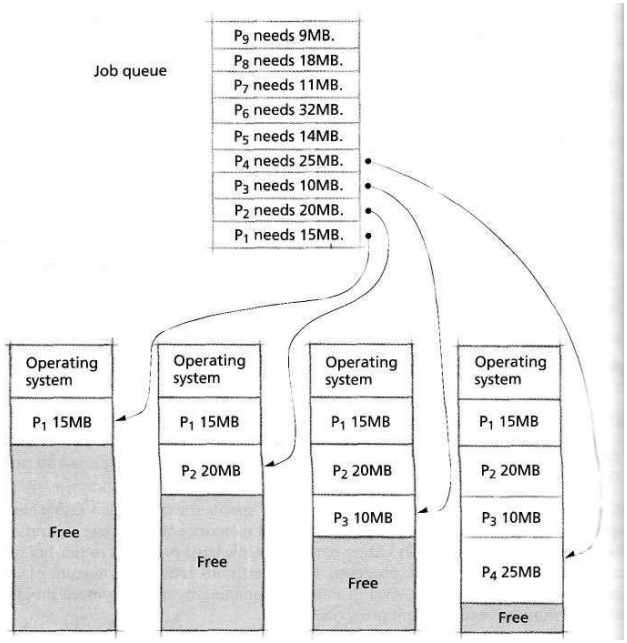
INTERNAL FRAGMENTATION



9.9 VARIABLE-PARTITION MULTIPROGRAMMING

- The operating system designers decided, would be to allow a process to occupy only as much space as needed (up to the amount of available main memory).
- This scheme is called **variable-partition multiprogramming**.

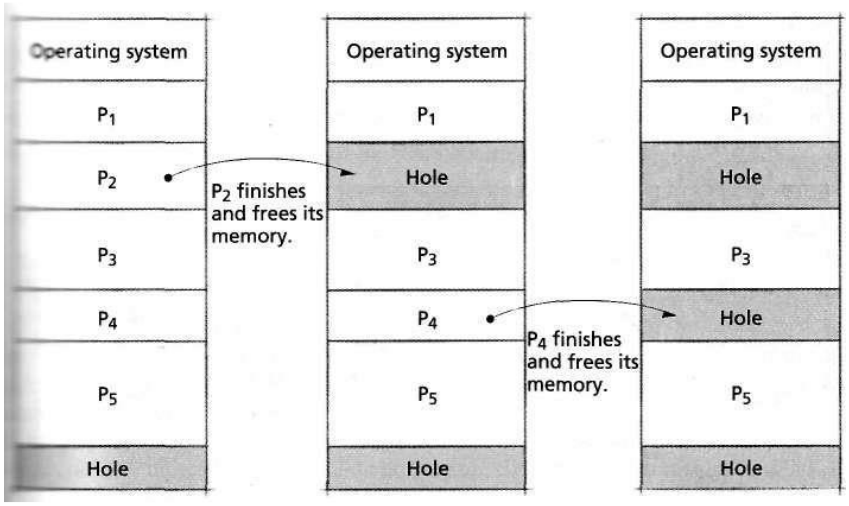
9.9.1 VARIABLE-PARTITION CHARACTERISTICS



- The queue at the top contains available jobs and information about their memory requirements.
- The operating system makes no assumption about the size of a job.

Holes

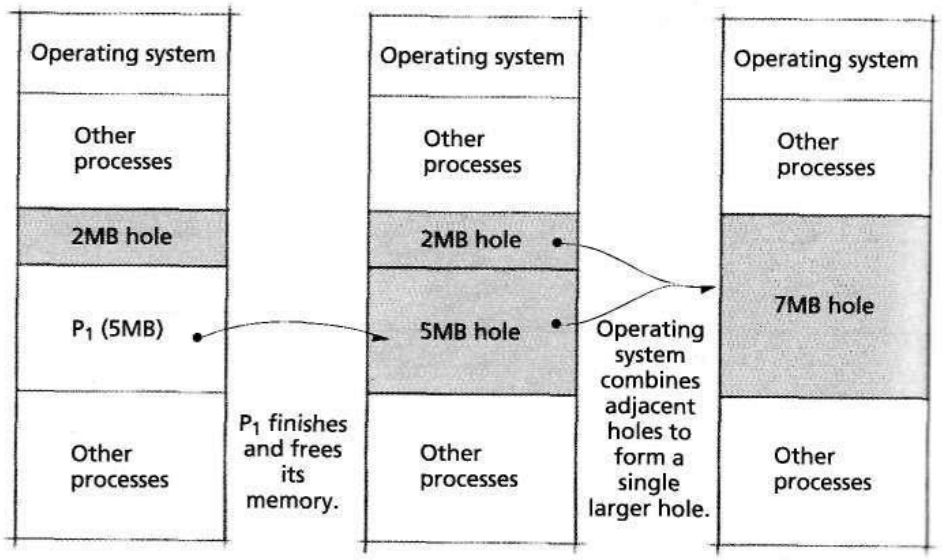
- Variable-partition multiprogramming organizations do not suffer from internal fragmentation.
- In variable partition multiprogramming, the waste does not become obvious until processes finish and leave **holes** in main memory.



- The system can continue to place new processes in these holes.
- However, as processes continue to complete, the holes get smaller, until every hole eventually becomes too small to hold a new process.
- This is called **external fragmentation.**

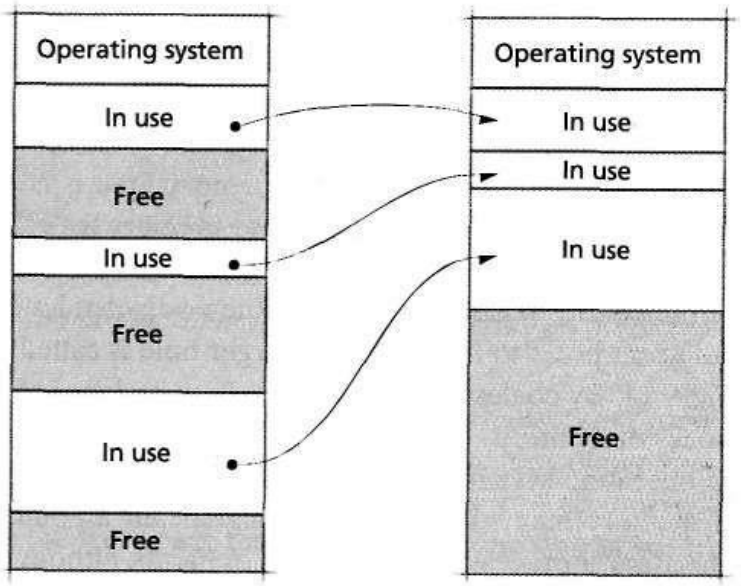
The system then records in a **free memory list** either (1) that the system now has an additional hole or (2) that an existing hole has been enlarged.

By coalescing holes, the system reclaims the largest possible contiguous blocks of memory.



- Another technique for reducing external fragmentation is called **memory compaction,** which relocates all occupied areas of memory to one end or the other of main memory.

- Now all of the available free memory is contiguous, so that an available process can run if its memory requirement is met by the single hole that results from compaction.
- Sometimes memory compaction is colorfully referred to as **burping the memory.**
- More conventionally, it is called **garbage collection.**



Operating system places all "in use" blocks together leaving free memory as a single large hole.

9.9.2 MEMORY PLACEMENT STRATEGIES

There are three strategies are used.

– First-fit strategy

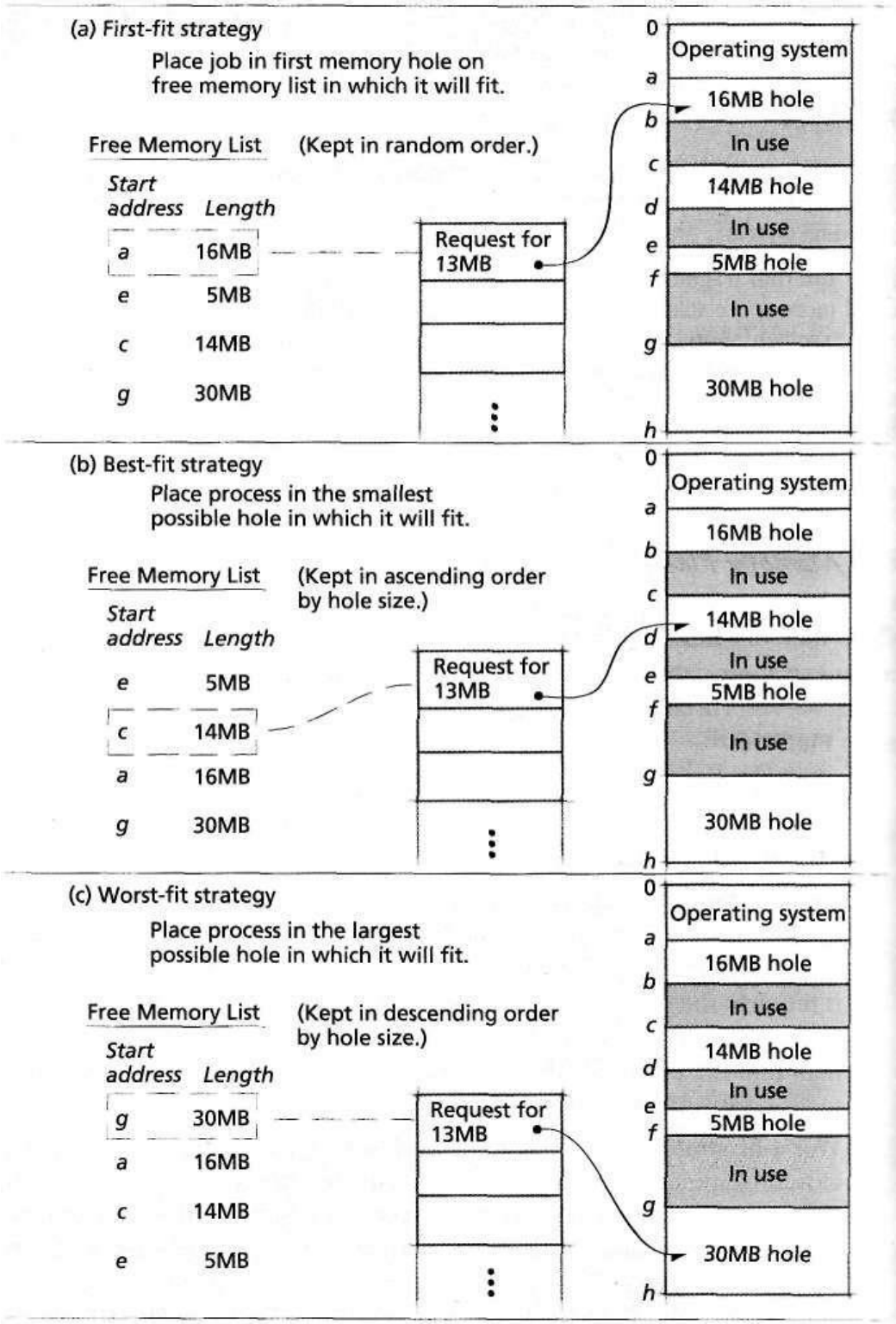
- Process placed in first hole of sufficient size found
- Simple, low execution-time overhead

– Best-fit strategy

- Process placed in hole that leaves least unused space around it
- More execution-time overhead

– Worst-fit strategy

- Process placed in hole that leaves most unused space around it
- Leaves another large hole, making it more likely that another process can fit in the hole.


(a) First-fit strategy
Place job in first memory hole on free memory list in which it will fit.
Free Memory List
(Kept in random order.)
Start address
Length
a 16MB
e 5MB
c 14MB
g 30MB
Request for 13MB
0
Operating system
a
16MB hole
b
In use
c
14MB hole
d
In use
e
5MB hole
f
In use
g
30MB hole
h
(b) Best-fit strategy
Place process in the smallest possible hole in which it will fit.
Free Memory List
(Kept in ascending order by hole size.)
Start address
Length
e 5MB
c 14MB
a 16MB
g 30MB
Request for 13MB
0
Operating system
a
16MB hole
b
In use
c
14MB hole
d
In use
e
5MB hole
f
In use
g
30MB hole
h
(c) Worst-fit strategy
Place process in the largest possible hole in which it will fit.
Free Memory List
(Kept in descending order by hole size.)
Start address
Length
g 30MB
a 16MB
c 14MB
e 5MB
Request for 13MB
0
Operating system
a
16MB hole
b
In use
c
14MB hole
d
In use
e
5MB hole
f
In use
g
30MB hole
h

VIRTUAL MEMORY MANAGEMENT

OUTLINE

VIRTUAL MEMORY MANAGEMENT

11.1 INTRODUCTION

- Replacement strategy
  - Technique a system employs to select pages for replacement when memory is full
  - Determines where in main memory to place an incoming page or segment
- Fetch strategy
  - Determines when pages or segments should be loaded into main memory
  - Anticipatory fetch strategies
- Use heuristics to predict which pages a process will soon reference and load those pages or segments.

11.5 PAGE REPLACEMENT

- When a process generates a page fault, the memory manager must locate referenced page in secondary storage, load it into page frame in main memory and update corresponding page table entry.
- Modified (dirty) bit
  - Set to 1 if page has been modified; 0 otherwise
  - Help systems quickly determine which pages have been modified
- Optimal page replacement strategy (OPT or MIN)
  - Obtains optimal performance, replaces the page that will not be referenced again until furthest into the future.

11.6 PAGE-REPLACEMENT STRATEGIES

Each strategy is characterized by the heuristic it uses to select a page for replacement and the overhead it incurs. Some replacement strategies are intuitively appealing but lead to poor performance.

11.6.1 RANDOM PAGE REPLACEMENT

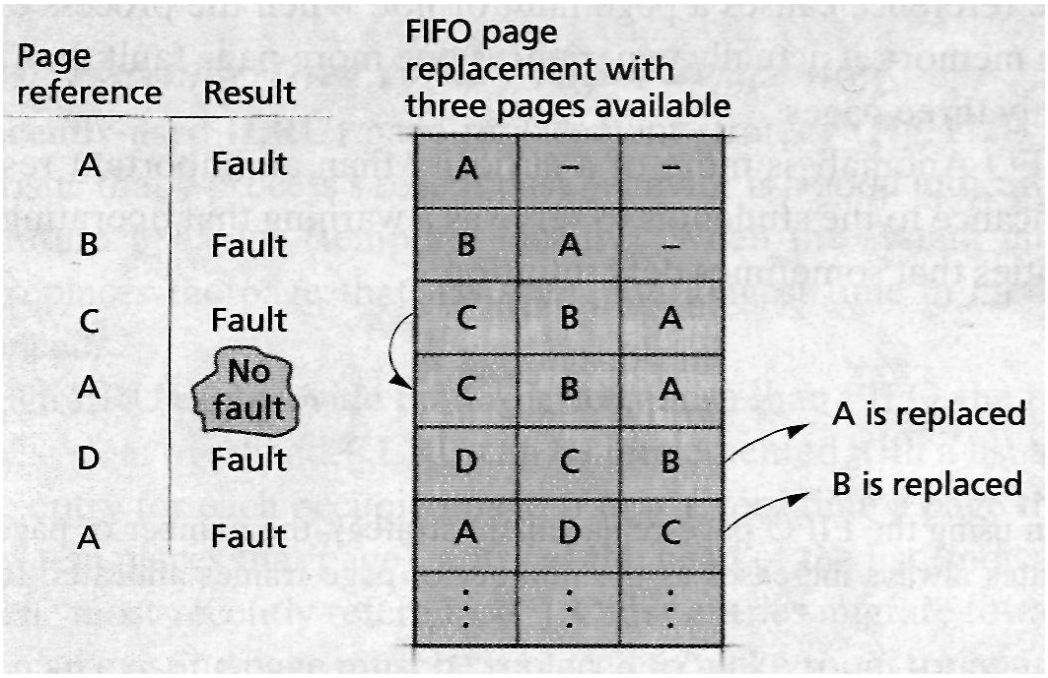
- Random page replacement
  - Low-overhead page-replacement strategy that does not discriminate against particular processes.

– Each page in main memory has an equal likelihood of being selected for replacement.

– Could easily select as the next page to replace the page that will be referenced next.

11.6.2 FIRST-IN-FIRST-OUT (FIFO) PAGE REPLACEMENT

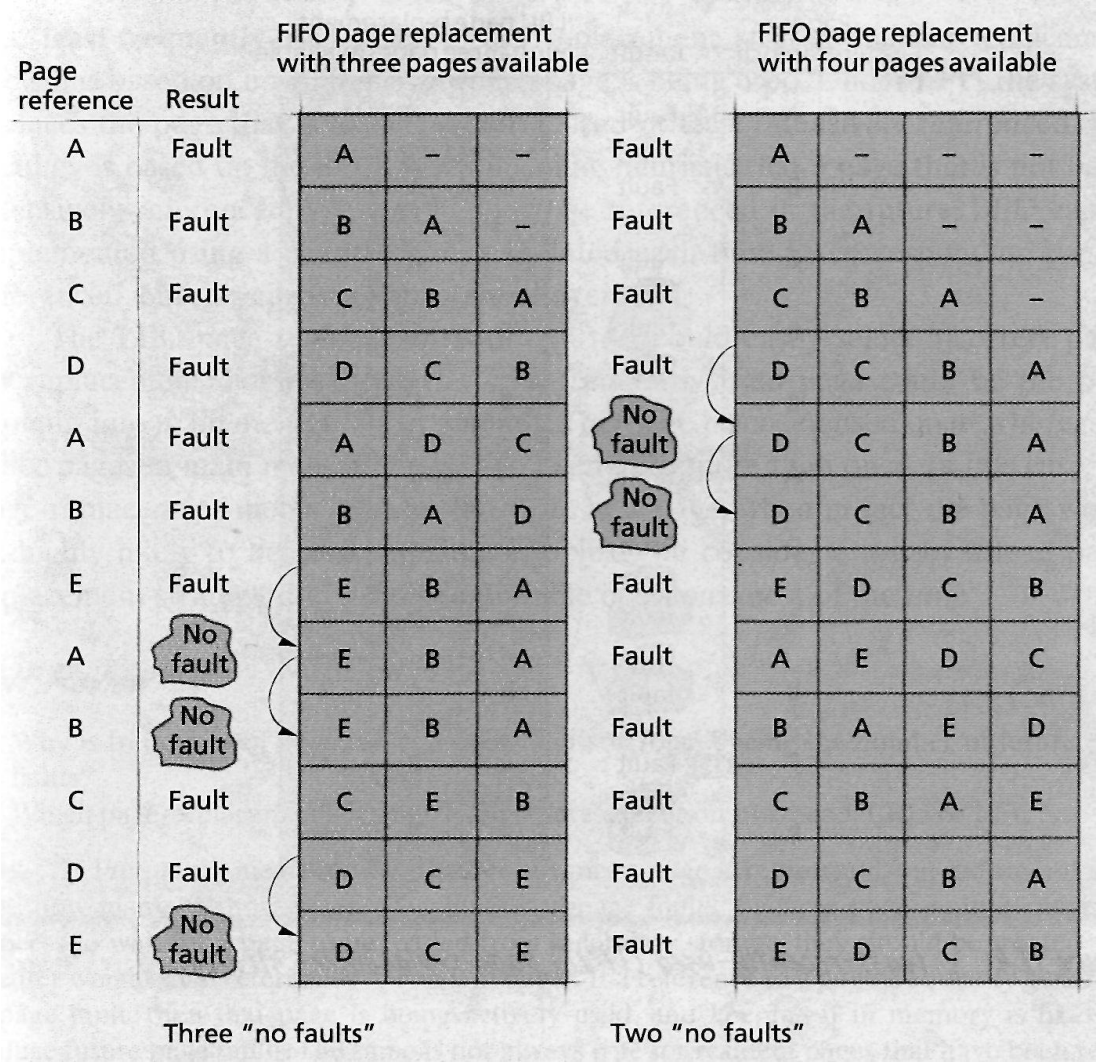
- **A** simple example of the FIFO strategy for a process which has been allocated three page frames.
- The leftmost column contains the process's page-reference pattern.
- Each row shows the state of the FIFO queue after each new page arrives; pages enter the tail of the queue on the left and exit the head on the right.
- This would be a poor choice, because the page would be recalled to main memory
- almost immediately, resulting in an increased page-fault rate. Modifications to FIFO: Second- Chance and Clock Page Replacement, FIFO forms the basis of various implemented page-replacement schemes.



11.6.3 FIFO ANOMALY

- The first table demonstrates how the reference pattern causes the system to load and replace pages (using FIFO) when the system allocates three page frames to the process.

- The second table shows how the system behaves in response to the same reference pattern, but when four page frames have been allocated.
- When the process executes with four pages in memory, it actually experiences one more page fault than when it executes with only three pages.



11.6.4 LEAST-RECENTLY USED (LRU) PAGE REPLACEMENT

– Exploits temporal locality by replacing the page that has spent the longest time in memory without being referenced

– Can provide better performance than FIFO

– Increased system overhead

– LRU can perform poorly if the least-recently used page is the next page to be referenced by a program that is iterating inside a loop that references several pages

Page reference	Result	LRU page replacement with three pages available		
A	Fault	A	–	–
B	Fault	B	A	–
C	Fault	C	B	A
B	No fault	B	C	A
B	No fault	B	C	A
A	No fault	A	B	C
D	Fault	D	A	B
A	No fault	A	D	B
B	No fault	B	A	D
F	Fault	F	B	A
B	No fault	B	F	A

11.6.5 LEAST-FREQUENTLY-USED (LFU) PAGE REPLACEMENT

– Replaces page that is least intensively referenced

– Based on the heuristic that a page not referenced often is not likely to be referenced in the future

– Could easily select wrong page for replacement

– A page that was referenced heavily in the past may never be referenced again, but will stay in memory while newer, active pages are replaced.

11.6.6 NOT-USED-RECENTLY (NUR) PAGE REPLACEMENT

The NUR strategy is implemented using the following two hardware bits per page table entry:

• **referenced bit—**set to 0 if the page has not been referenced and set to one if the page has been referenced.

• **modified bit—**set to 0 if the page has not been modified and set to 1 if the page has been modified.

- The referenced bit is sometimes called the **accessed bit.**
- The pages in the lowest-numbered groups should be replaced first, and those in the highest-numbered groups last. Pages within a group are selected randomly for replacement.
- Note that Group 2 seems to describe an unrealistic situation—namely, pages that have been modified but not referenced.
- This occurs because of the periodic resetting of the referenced bits.

Group	*Referenced*	*Modified*	*Description*
Group 1	0	0	Best choice to replace
Group 2	0	1	[Seems unrealistic]
Group 3	1	0	
Group 4	1	1	Worst choice to replace

11.6.7 MODIFICATION TO FIFO: SECOND-CHANCE AND CLOCK PAGE REPLACEMENT

• **Second chance page replacement**

– Examines referenced bit of the oldest page

• If it's off

– The strategy selects that page for replacement

• If it's on

– The strategy turns off the bit and moves the page to tail of FIFO queue

– Ensures that active pages are the least likely to be replaced

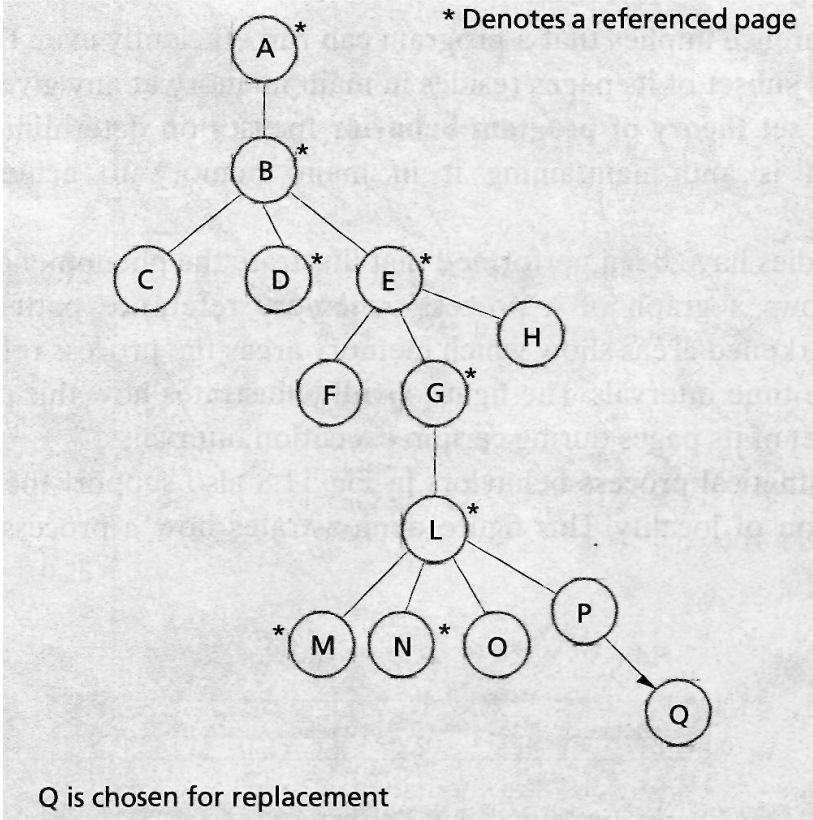
• **Clock page replacement**

– Similar to second chance, but arranges the pages in circular list instead of linear list.

11.6.8 FAR PAGE REPLACEMENT

- Each vertex in the access graph represents one of the process's pages.
- An edge from vertex v to vertex *w* means that the process can reference page *w* after it has referenced page v.
- For example, if an instruction on page *v* references data on page *w,* there will be a directed edge from vertex *v* to vertex *w.*
- Similarly, if a function call to page *x* returns to page y, there will be an edge from vertex *x* to vertex *y.*

- The access graph indicates that, after the process references page B, it will next reference either page A, C, D or E, but it will not reference page G before it has referenced
- page E.



- The field of graph theory provides algorithms for building and searching the
- kinds of graphs in the far strategy.
- However, largely due t its complexity and execution-time overhead, far has not been implemented in real systems.

11.8 PAGE-FAULT-FREQUENCY (PFF) PAGE REPLACEMENT

- The **page-fault-frequency (PFF)** algorithm adjusts a process's **resident page set**
- based on the frequency at which the process is faulting.
- Alternatively, PFF may adjust a process's resident page set based on the time between page faults, called the process's **interfault time.**

- PFF has a lower overhead than working set page replacement because it adjusts the resident page set only after each page fault; a working set mechanism must operate after each memory reference.
- A benefit of PFF is that it adjusts a process's resident page set dynamically, in
- response to the process's changing behavior.
- If a process is switching to a larger working set, then it will fault frequently, and PFF will allocate more page frames.

11.9 PAGE RELEASE

• Inactive pages can remain in main memory for a long time until the management strategy detects that the process no longer needs them

– One way to solve the problem

• Process issues a voluntary page release to free a page frame that it knows it no longer needs

• Eliminate the delay period caused by letting process gradually pass the page from its working set

• The real hope is in compiler and operating system support.

11.10 PAGE SIZE

• Some systems improve performance and memory utilization by providing multiple page sizes

– Small page sizes

• Reduce internal fragmentation

• Can reduce the amount of memory required to contain a process's working set

• More memory available to other processes

– Large page size

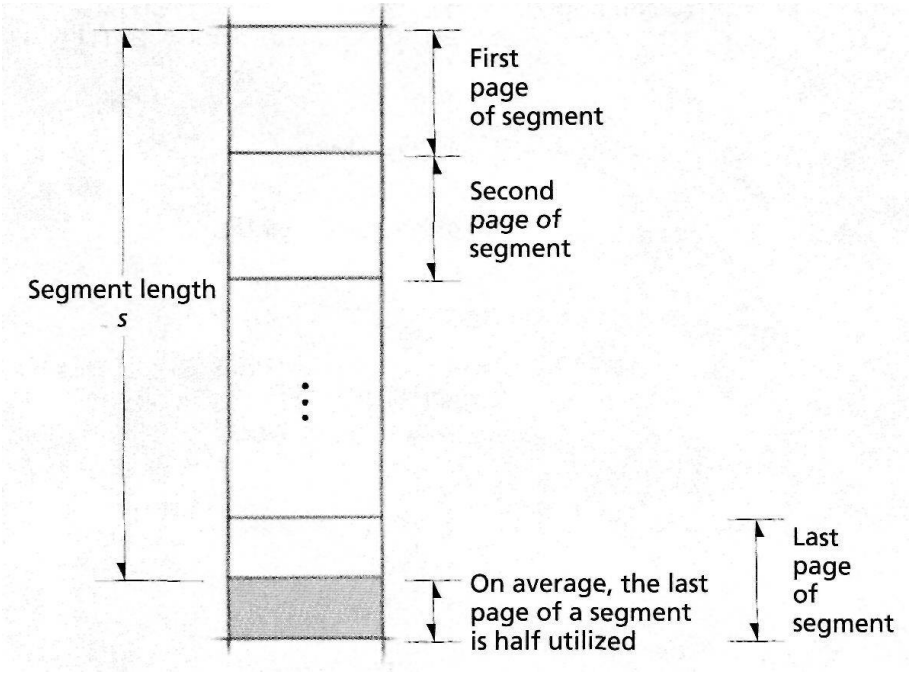
• Reduce wasted memory from table fragmentation

• Enable each TLB entry to map larger region of memory, improving performance

• Reduce number of I/O operations the system performs to load a process's working set into memory

– Multiple page size

• Possibility of external fragmentation



Manufacturer	*Model*	*Page Size*	*Real adress size*
Honeywell	Multics	1KB	36 bits
IBM	370/168	4KB	32 bits
DEC	PDP-10 and PDP-20	512 bytes	36 bits
DEC	VAX 8800	512 bytes	32 bits
Intel	80386	4KB	32 bits
Intel/AMD	Pentium 4 / Athlon XP	4KB or 4MB	32- or 36 bits
Sun	UltraSparc II	8KB, 64KB, 512KB, 4MB	44 bits
AMD	Opteron / Athlon 64	4KB, 2MB and 4MB	32, 40, or 52 bits
Intel-HP	Itanium, Itanium 2	4KB, 8KB, 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, 64MB, 256MB	Between 32 and 63 bits
IBM	PowerPC 970	4KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB	32 or 64 bits