

# Operating Systems

---

Dr. Coral Yan Huang

Email: [Yan.Huang@cs.cf.ac.uk](mailto:Yan.Huang@cs.cf.ac.uk)

Office: T/2.09

---

# Text Book

## Operating Systems

Deitel Deitel Choffnes, 2004 (Third Edition), Pearson Prentice Hall, ISBN 0-13-124696-8

# Contents

(1)

1. Introduction to operating systems
2. Hardware and software concepts
3. Process concepts
4. Thread concepts
5. Asynchronous concurrent execution
6. Concurrent programming
7. Deadlock
8. Processor scheduling

# Contents

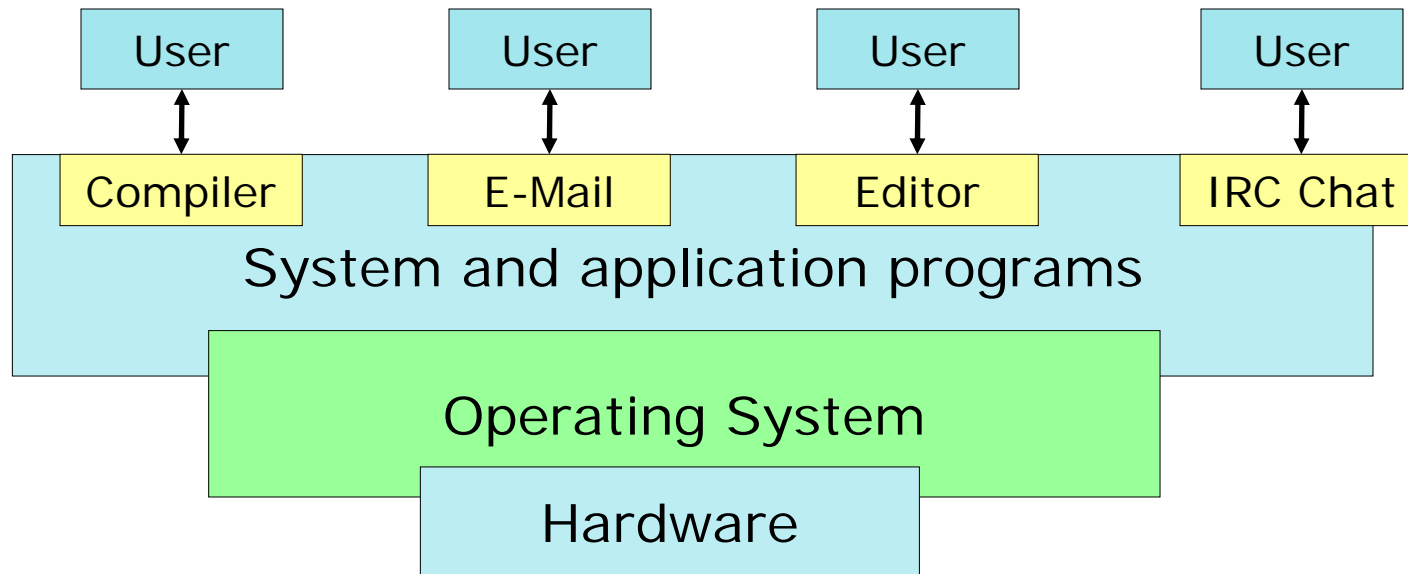
(2)

- 9. Memory management
- 10. Virtual memory management
- 11. Disk performance optimization
- 12. File and Database systems
- 13. Performance and processor design
- 14. Case study: UNIX
- 15. Case study: Windows XP

# Chapter 1

## Introduction to Operating Systems

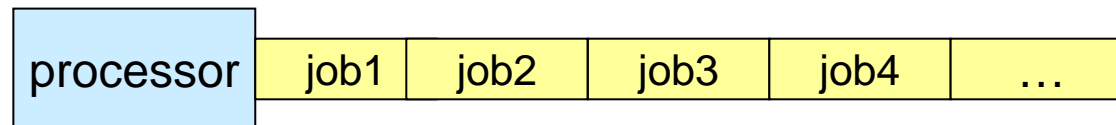
# What is an Operating System?



An *operating system* is software that enables applications to interact with a computer's hardware.

# History of Operating Systems (1)

- 1940s: no operating systems.
  - Assembly language developed
- 1950s:
  - First OS for IBM 701 computer by General Motors Research Labs
  - One job a time—Single-stream batch-processing systems.



# History of Operating Systems (2)

## ■ 1960s

### □ Multiprogramming Systems

- More peripheral devices, such as card readers, disk drives, tape drives, printers.
- Process-bound jobs and IO-bound jobs.
- Running multiple jobs at a time — optimize resource utilization.

### □ Timesharing Systems

- Interactive users communicate with their jobs during execution. (via “dumb terminals”)
- The term *process* is used to describe a program in execution.
- Virtual memory



# History of Operating Systems (3)

## □ 1970s

- Multimodel multiprogramming systems that support batch processing, timesharing and real-time applications.
  - Multimodel systems support both batch-processing and real-time applications
  - Real-time applications require the operating system to supply a response within a certain bounded time period.
- Commercial products of Operating Systems.
- TCP/IP communications standard became widely used.
- Security was encompassed

# History of Operating Systems (4)

## ■ 1980s

- The decade of the PC and the workstation—
  - Individuals and small businesses can have their own powerful computer.
- Graphical User Interface (GUI) with mouse provides easy-to-use interface
- Distributed computing (under Client/Server model) is widespread

# History of Operating Systems (5)

- 1990s
  - Hardware performance continued to improve exponentially.
    - Speed — several hundred MIPS (normal PC) , over one TIPS (supercomputer).
    - Storage — one GB a hard disk
  - With the creation of World Wide Web (WWW) and fast internet connections, distributed computing is commonly used between PCs.
    - OS support for networking tasks became standard.
    - Security threats.

To be continued

# History of Operating Systems (6)

- 1990s
  - Microsoft Corporation became dominant.

| Year | Operating Systems             |                 |
|------|-------------------------------|-----------------|
| 1981 | DOS                           |                 |
| 1985 | Win 1.0                       |                 |
| 1990 | Windows 3.0                   |                 |
| 1993 | Windows 3.1                   | Windows NT 3.1  |
| 1994 | Windows NT 3.5                |                 |
| 1995 | Windows 95                    | Windows NT 3.51 |
| 1996 | Windows NT 4.0                |                 |
| 1997 | Windows NT Enterprise Edition |                 |

To be Continued

# History of Operating Systems

(7)

| Year  | Operating Systems   |
|-------|---|
| 1998  | Windows 98    Windows NT 4.0<br>Terminal Server Edition   |
| 1999  |   |
| 2000  | Windows ME    Windows 2000 Family   |
| 2001  | Windows XP (Home, Professional)   |
| 2002  | (Windows XP Service Pack 1)   |
| 2003  | Windows Server 2003   |
| 2004+ | (Windows XP Service Pack 2)<br>Windows XP Media Center Edition 2005<br>Windows XP Professional x64Edition |

To be Continued

# History of Operating Systems (8)

- 1990s

- Object Technology

- The appearance of object-oriented languages such as C++ and Java
    - An Object includes both data and functions. In addition, programmers can create relationships between one object and another such as inheritance.
    - Object-Oriented modular operating systems are easier to maintain and extend.

# History of Operating Systems (9)

- 2000 and Beyond
  - Middleware is software that links web applications over a network.
    - CORBA, RMI, JXTA, Globus, Web service technologies
    - GRID
  - Next Operating systems
    - Improved security,
    - High-level standard support for parallelism
    - Enhanced 3D user interface
    - More compatible with other operating systems.

# Operating System Components

- *Shell, or command interpreter* — allows user to enter a command.
- *Kernel* — the software that contains the core components of the system.
  - Process scheduler
  - Memory manager
  - I/O manager
  - Interprocess communication (IPC) manager
  - File system manager



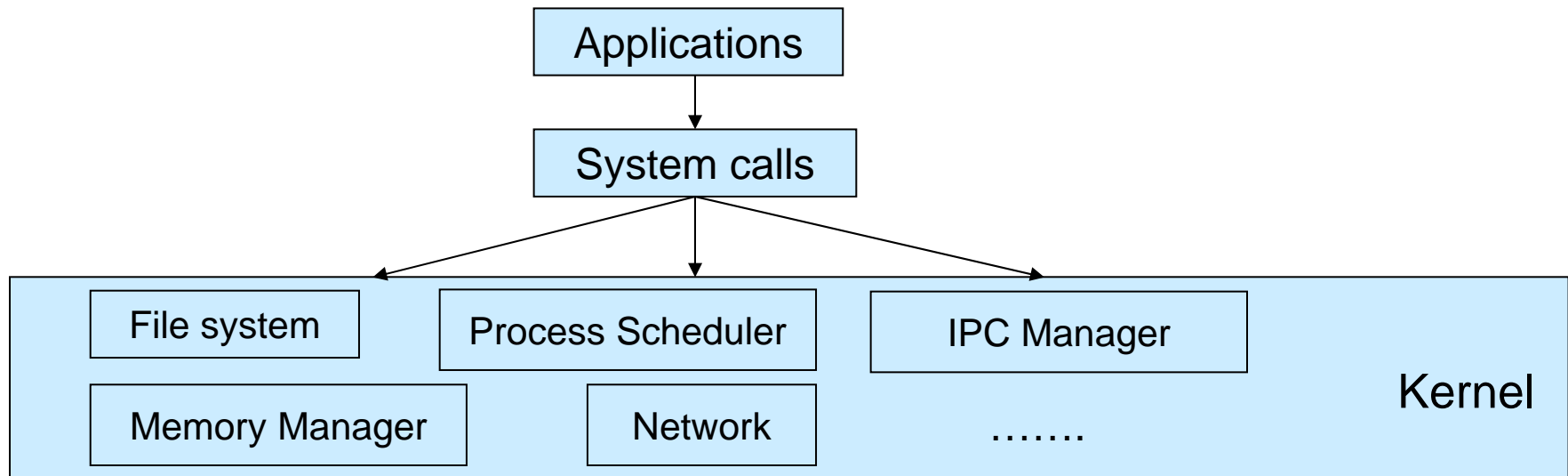
# Operating System Goals

- Efficiency
- Robustness
- Scalability
- Extensibility
- Portability
- Security
- Interactivity
- Usability

# Operating System Architectures (1)

## ■ Monolithic Architecture

- The earliest and most common
- Every component is contained in the kernel and can directly communicate with other.



# Operating System Architectures (2)

## ■ Layered Architecture

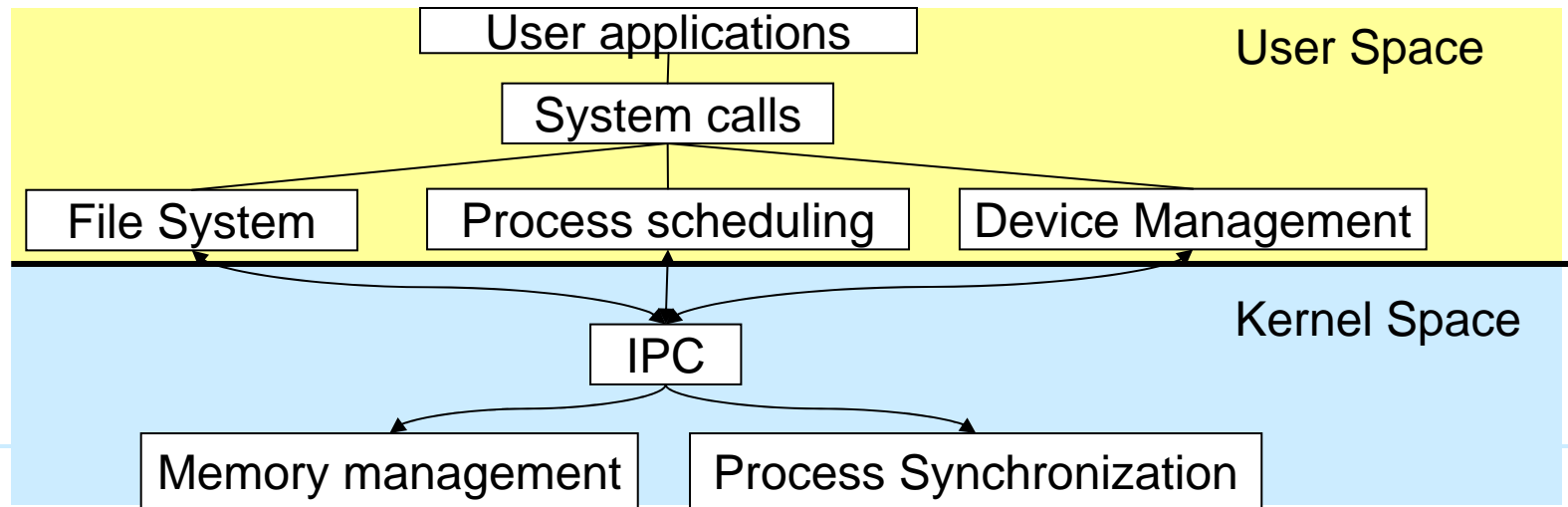
- Group components that perform similar functions into layers. Each layer communicates only with neighbour layers

|   |         |                     |              |
|---|---------|---------------------|--------------|
| □ | Layer 4 | User applications   | User Space   |
|   | Layer 3 | I/O Management      | Kernel Space |
|   | Layer 2 | Message interpreter |              |
|   | Layer 1 | Memory management   |              |
|   | Layer 0 | Process management  |              |

# Operating System Architectures (3)

## ■ Microkernel Architecture

- Only a small number of services (Typically IPC, Memory management and process synchronization) are provided to keep the kernel small and scalable. Most OS components such as the file system, process scheduler and device manager execute outside the kernel.



---

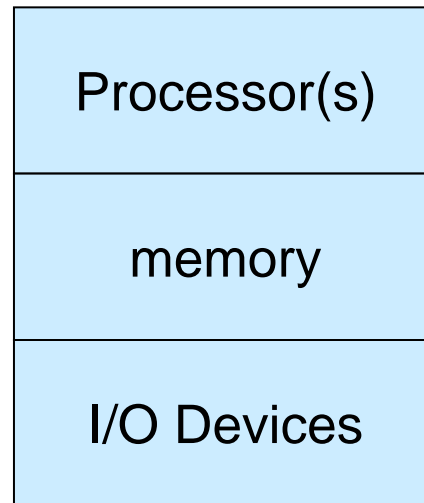
# Summary

- What is an Operating System?
- History of Operating Systems.
- Core operating system components
- Operation system goals
- Operating system architecture

# Chapter 2

## Hardware and Software Concepts

# Hardware Components



# Mainboards

- The central PCB (printed circuit board)
  - Slots – for processors, memory and other devices
  - Bus – provides high speed communication.
  - Chips, such as
    - BIOS (basic I/O system) chip
    - Controllers

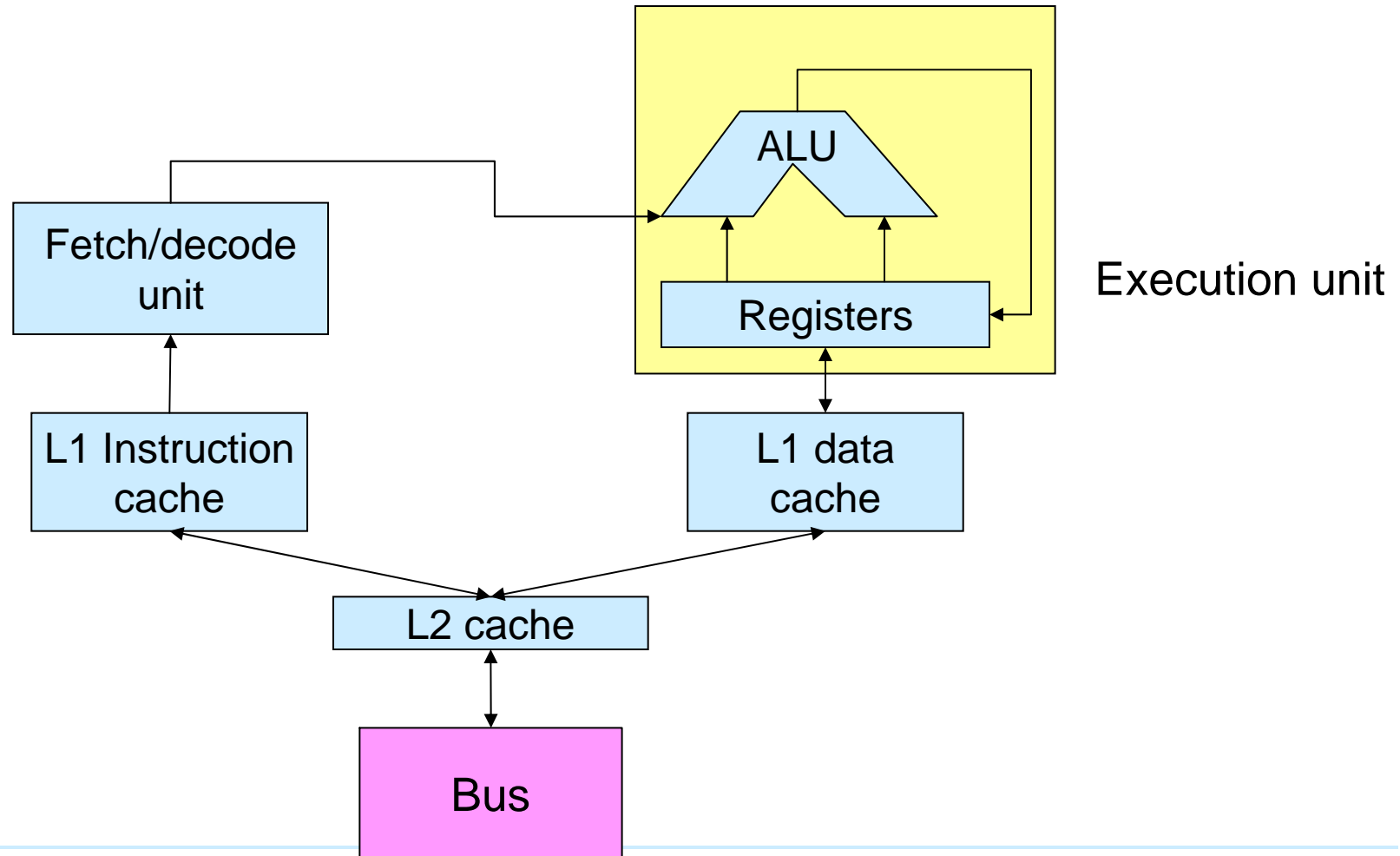


# Processor

- A processor is a hardware component that executes a stream of machine language.
- Processors can be
  - CPU (Central Processing Unit)
  - Graphics coprocessor
  - DSP (Digital Signal Processor)
- Instruction set defines the set of instructions a processor can execute.

# Processor Components

(1)



# Processor Components

(2)

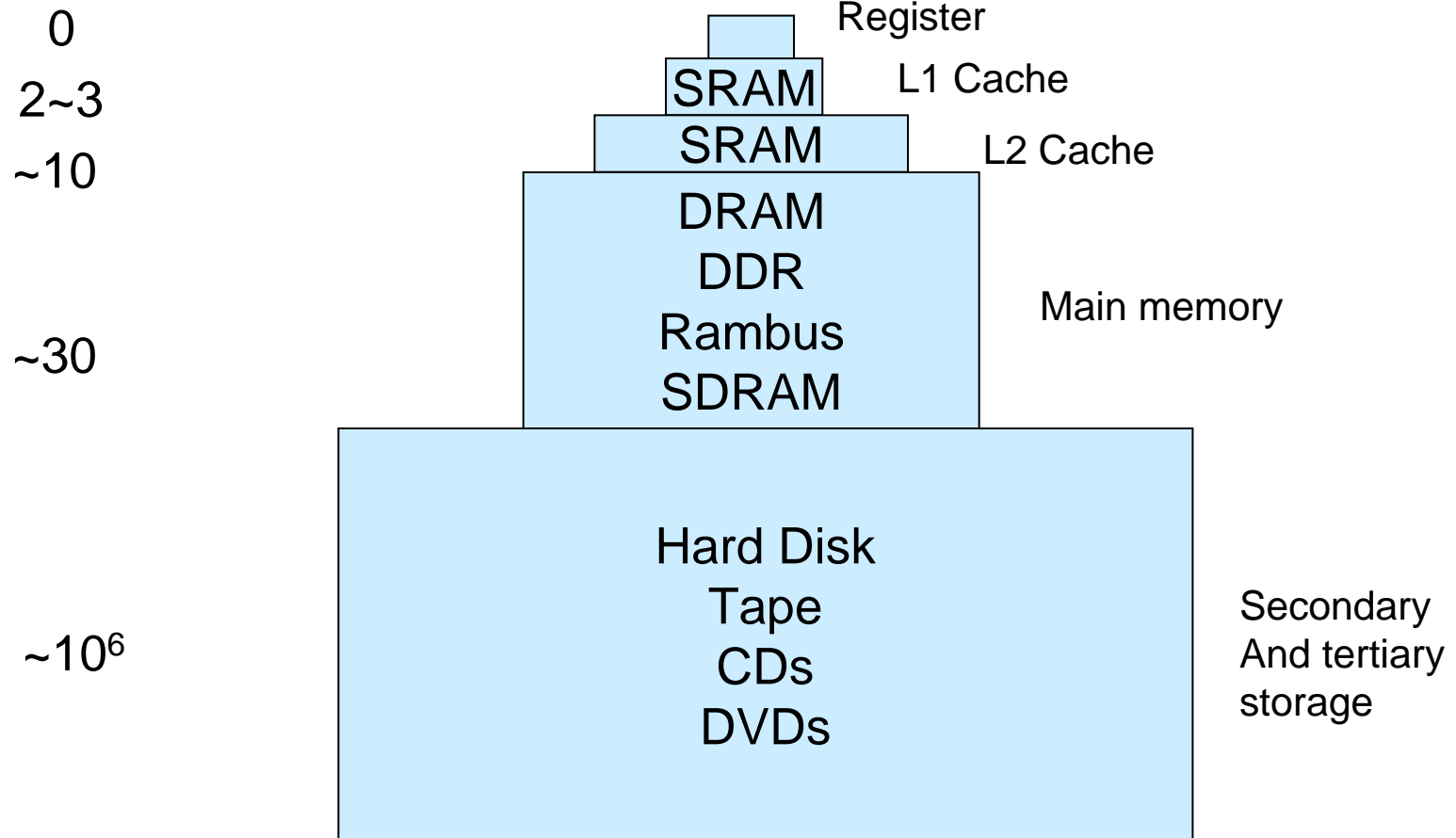
- ALU – Arithmetic and logic unit
- Registers are high-speed memories located on a processor that hold data for immediate use by the processor.

# Clocks

- Computer time is measured in cycles. A cycle refers to one complete oscillation of an electrical signal provided by the system clock generator.
- System clock generator decides the frequency at which buses transfer data, measured in cycles per second or Hz

# Memory Hierarchy

Latency (in processor cycles)



# Memory

- RAM – Random Access Memory
- DRAM – Dynamic RAM. Requires that a refresh circuit periodically (a few times every millisecond) reads the contents or the data will be lost.
- SRAM – Static RAM. It doesn't need to be refreshed to maintain the data.
  - Much faster but very expensive.

---

# Secondary Storage

- Hard Disk: 1TB
- Compact Disk (CD). 700MB per side
- Digital versatile disk (DVD). Store data in thinner tracks on up to two layers per side. Each layer can store up to 5.6 GB.

# Buses

- Data bus
- Address bus
- I/O channel.
- Peripheral component interconnect (PCI) bus
- Accelerated Graphics Port (AGP)



# Peripheral Devices

- Serial ports – transfer data one bit a time.
  - Mice, keyboards
- Parallel ports – transfer data several bits a time.
  - Printers
- USB ports – Transfer data at a fast speed.
  - 480Mbit per second (USB 2.0)
  - External disk drives, digital cameral, printers

# Software Overview

- Programming languages can be
  - Machine and assembly languages
  - High-level languages
    - Java, C++, VB++..

# Machine Language and Assembly Language

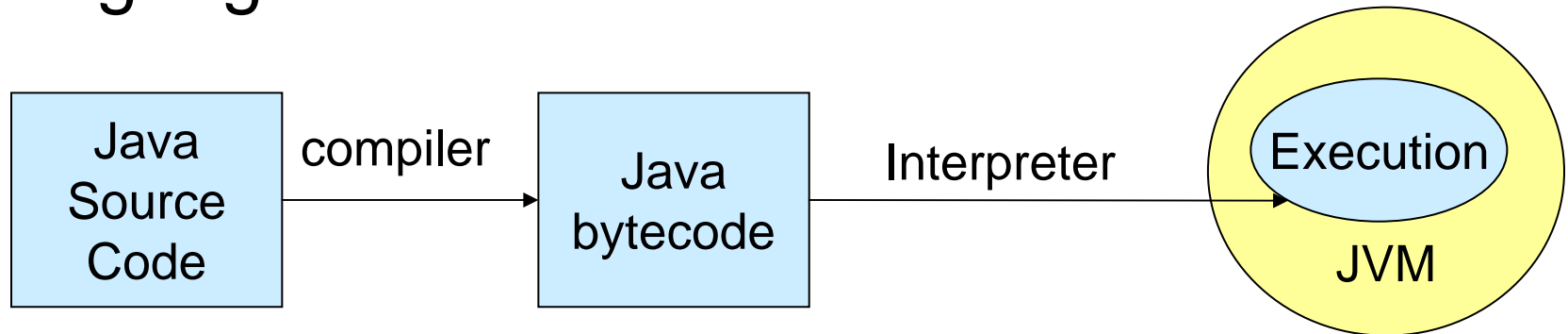
- A computer can only understand its own machine language
  - It consists a stream of numbers
  - Incomprehensible to humans
- Assembly language – use English-like abbreviations to represent the computer's basic operations.
  - A translator (called an assembler) converts assembly-language program to machine-language program.

# High-level Languages

- Fortran is the first high-level language (mid-1950s).
- Then COBOL (late 1950s)
- C (1970s)
- C++ (early 1980s) is object-oriented language.
- Java (1995)
- C# and .NET

# Interpreters and Compilers

- Compiler – covert high-level language program into machine language or low-level language.
- Interpreter – Directly execute source code or code that has been reduced to a low-level language.

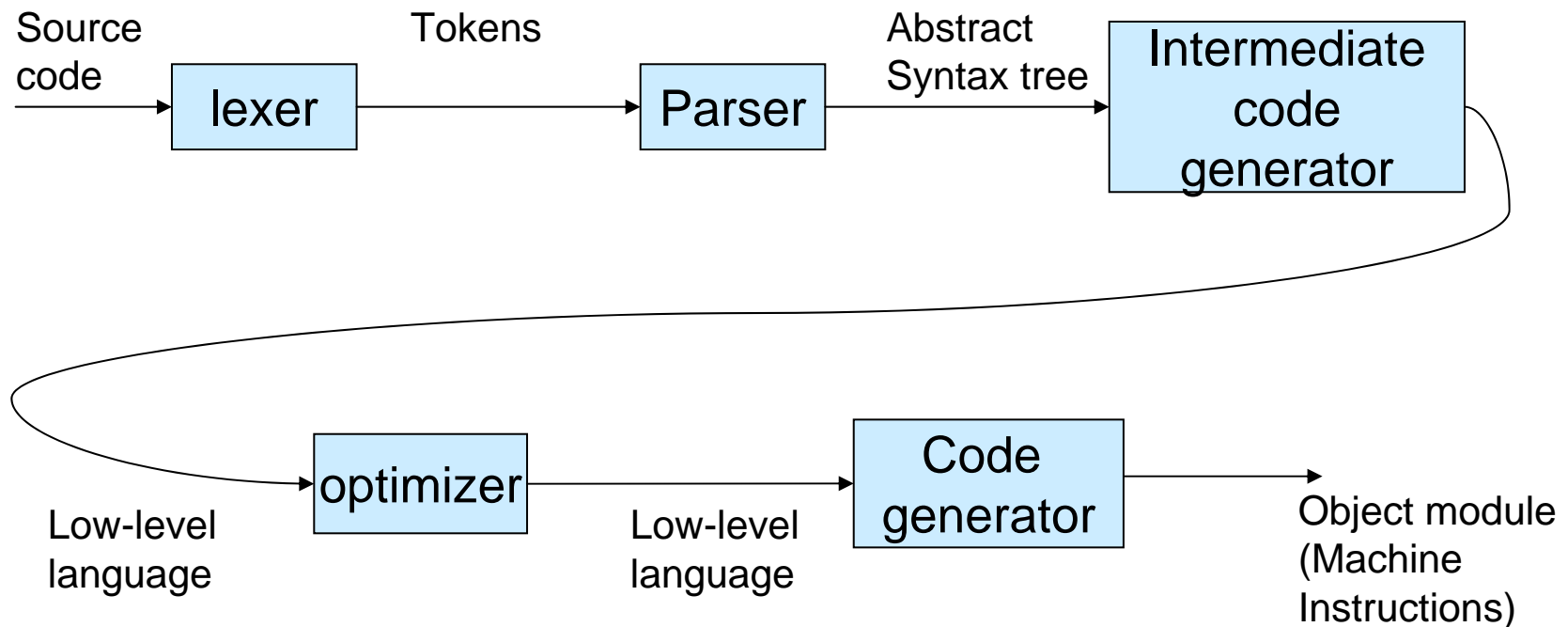


# Application Programming Interfaces (APIs)

- APIs provide a set of routines that programmers can use to request services from the system.
  - UNIX POSIX (Portable Operating System Interface)
  - Win32 API

# Compiling, Linking and Loading (1)

## ■ Compiling



# Compiling, Linking and Loading (2)

## ■ Linking

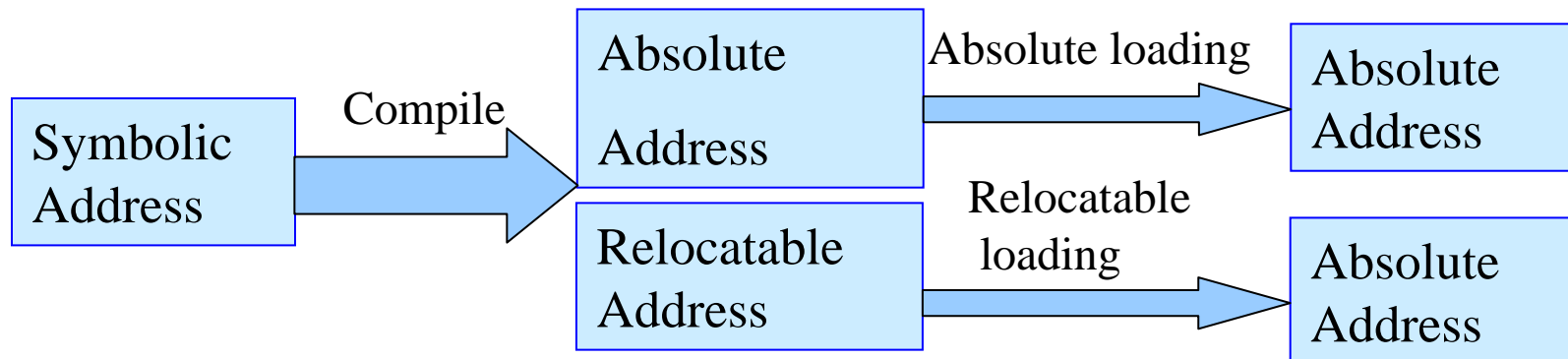
- Integrate the various modules referenced by a program into a single executable routine.
- Shared libraries – collection of functions that can be shared between different processes.
- Dynamic linking – The linking is postponed until execution time.



# Compiling, linking and loading (3)

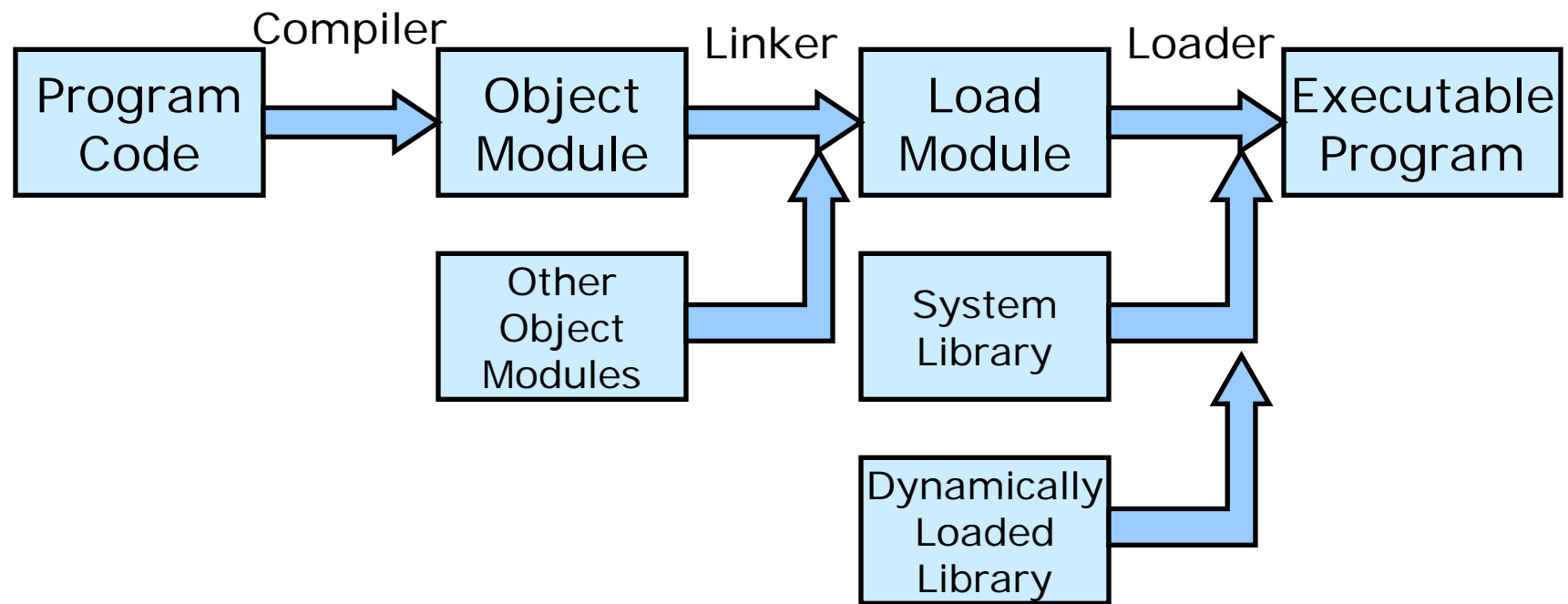
## ■ Loading

- Load program load modules into memory.
  - Absolute loading
  - Relocatable loading



- Dynamic loading – a program load module is not loaded until it is called.

# Compiling, linking and loading (4)



---

# Summary

- Hardware components
  - Processor, memory, bus, secondary storage, clocks, peripheral devices
- Software overview
  - Machine, assembly and high-level languages
  - Interpreters and Compilers
  - Application Programming Interfaces
  - Compiling, linking and loading

# Chapter 3:

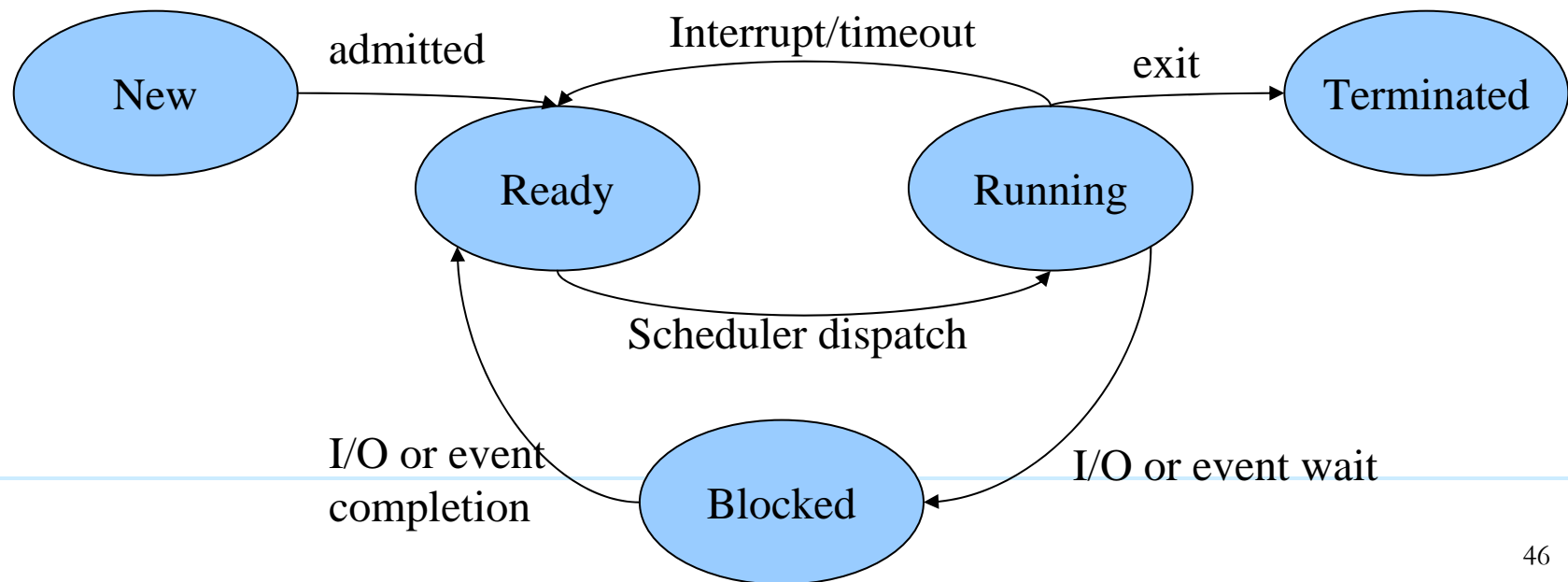
# Process Concepts

# What is a process?

- A process is an entity with its own address space. The address space typically consists of
  - text region: code
  - data region: variables and dynamically allocated memory
  - stack region: instructions and local variables
- A process is a program in execution

# Process States

- ❑ **Running State:** the process is executing on a processor
- ❑ **Ready State:** the process could execute on a processor when one is available
- ❑ **Blocked State:** the process is waiting for some event to happen before it can proceed.



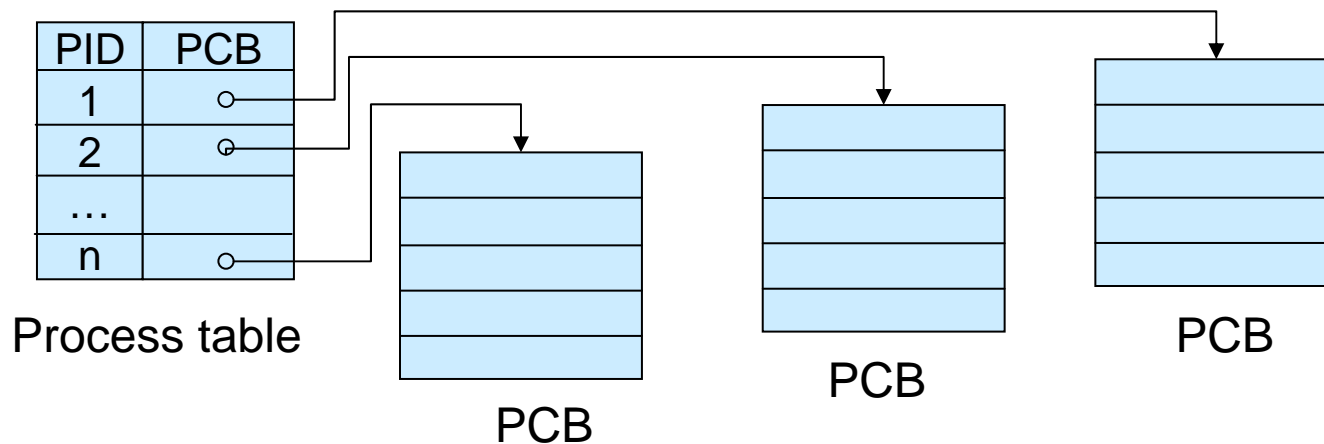
# Process Control Block (PCB) (1)

- Maintains information about the process.
  - Process counter – a value that determines which instruction the process should execute next.
  - Execution Context of a processor– the register content when the process was last running. It enables a process execution context to be restored when the process returns to the running state.

|                 |
|-----------------|
| Process counter |
| registers       |
| state           |
| priority        |
| Address space   |
| parent          |
| children        |
| Open files      |
| .....           |

# Process Control Block (PCB) (2)

- A process table is used to allow PCB to be accessed quickly.



- PID – process identification number



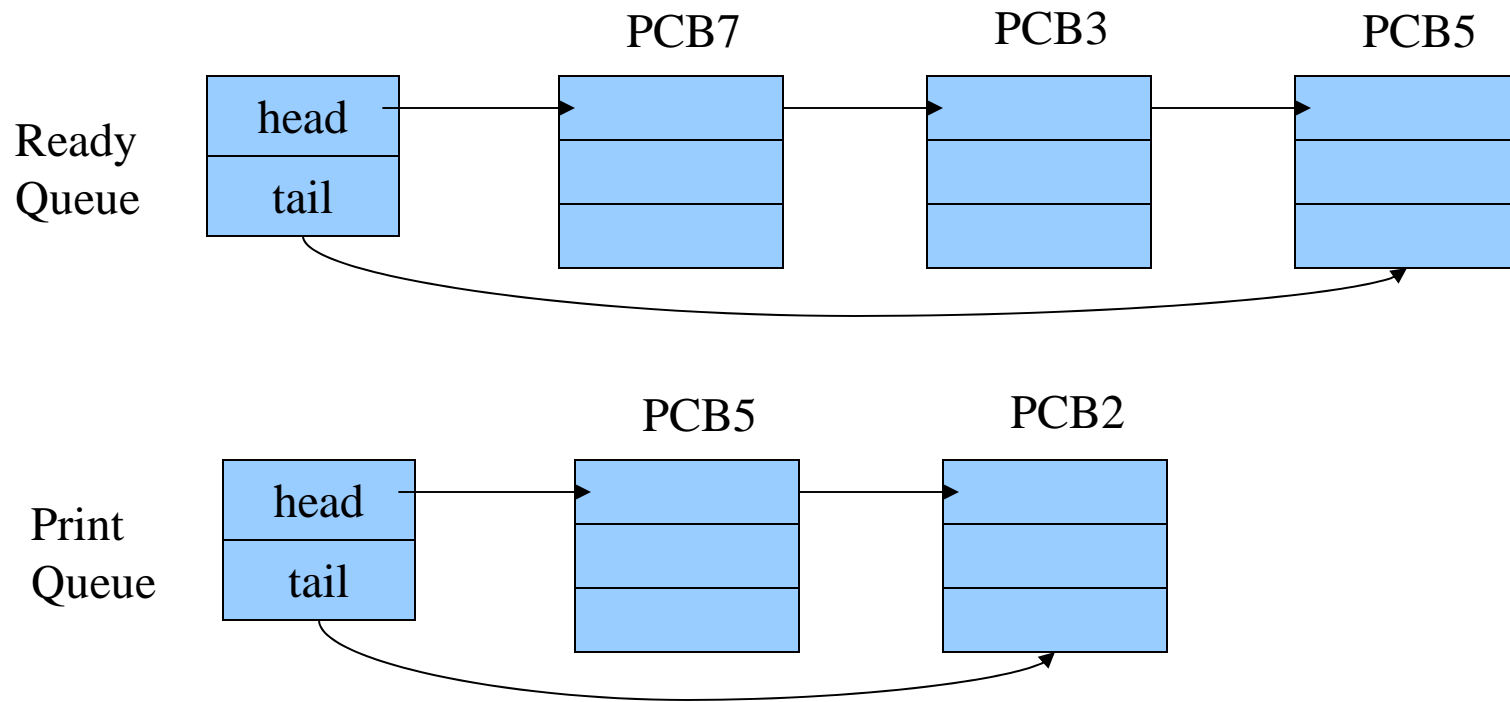
# Process Queues

(1)

- Job queue: a list of all the processes in the system
- Ready queue: a list of processes that are residing in memory and are ready and waiting to execute
- Device queue: a list of processes waiting for a particular I/O device. Each device has its own device queue

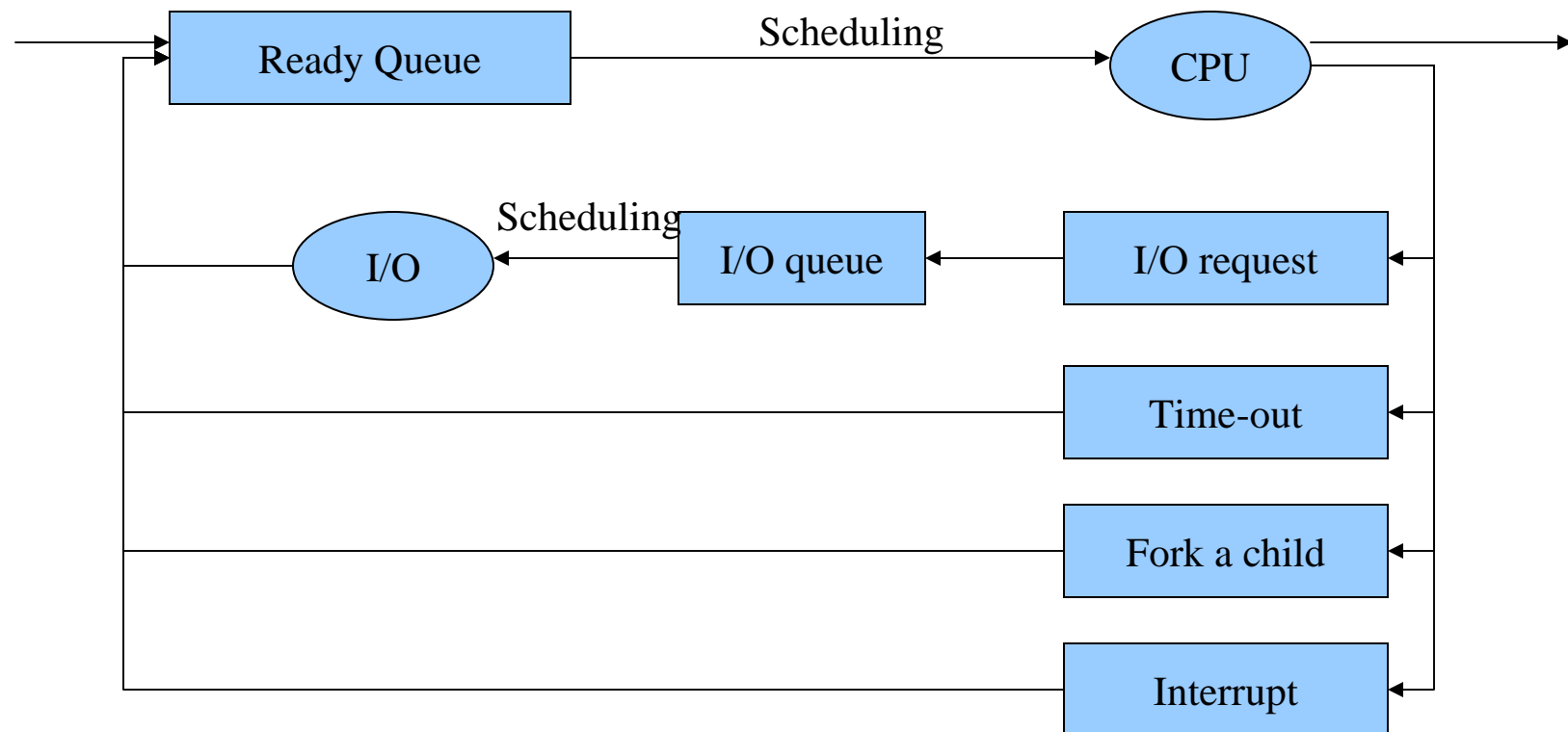
# Process queues

(2)



# Process queues

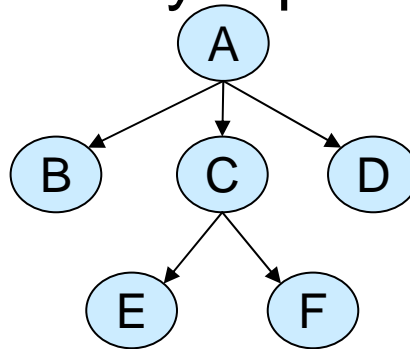
(3)



# Process Operations

(1)

- Create and destroy a process

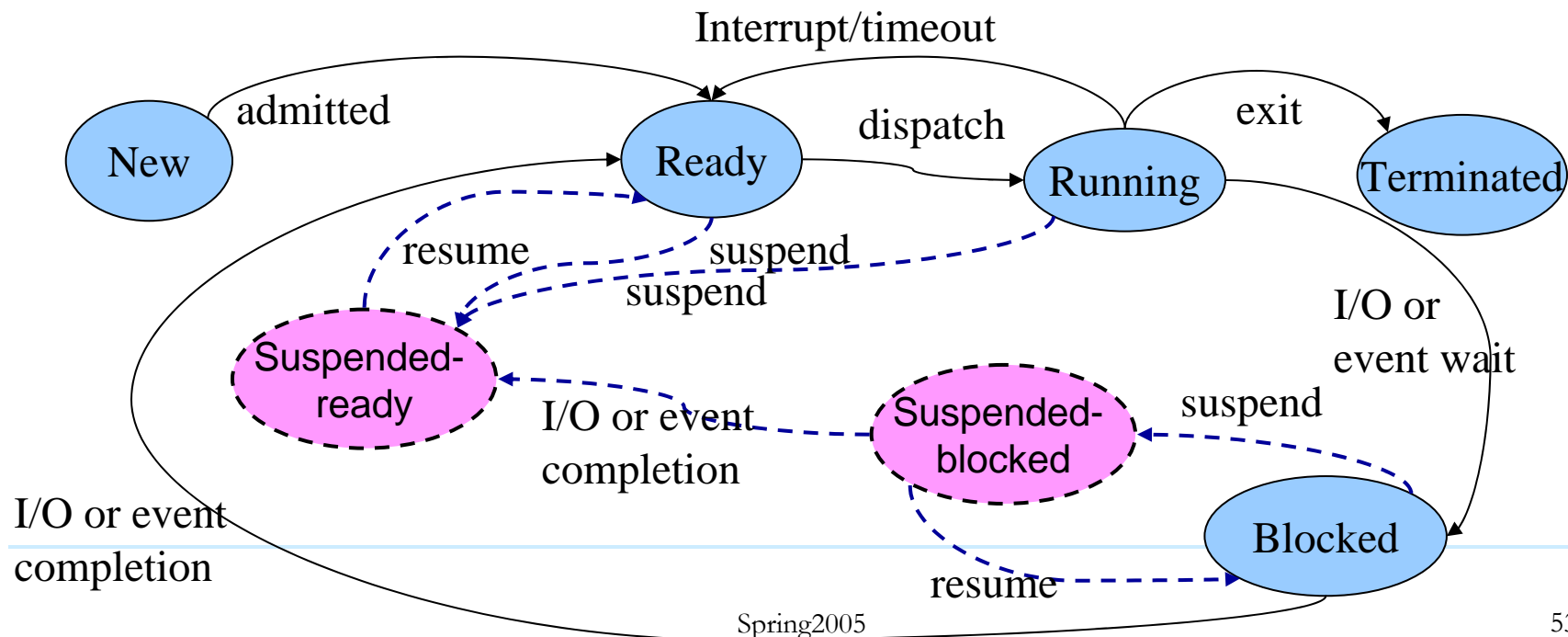


- Block and unblock a process
- Dispatch a process
- Change priority
- Operations for interprocess communication.

# Process Operations

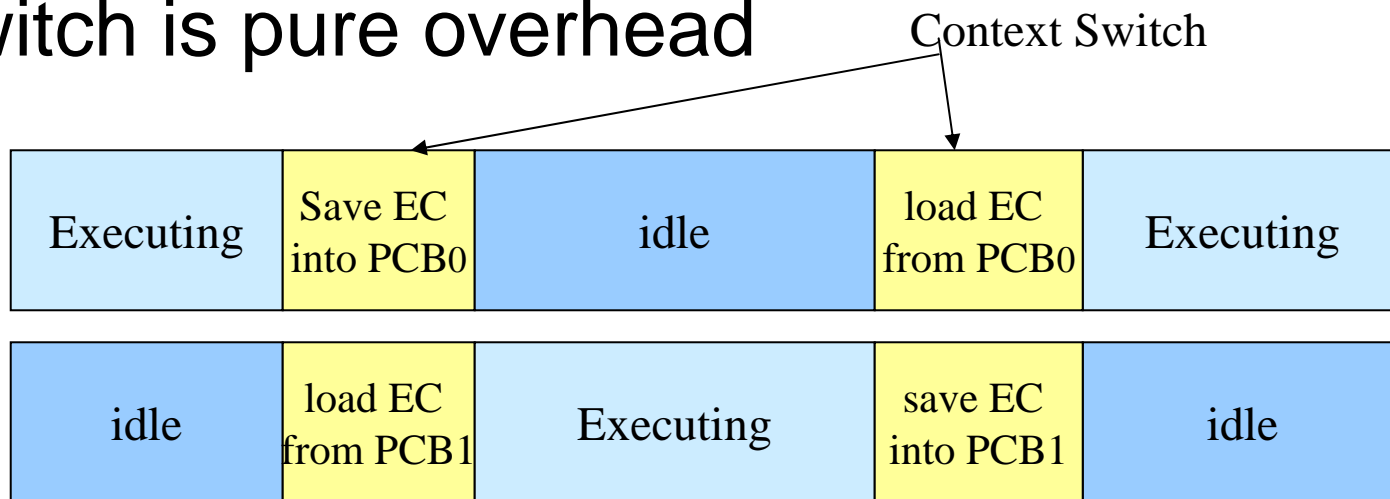
(2)

- Suspend and resume a process
  - A suspended process is indefinitely removed without being destroyed
  - Normally for detecting security threats and debugging purposes



# Context Switching

- Switching the CPU to another process requires saving the execution context of the old process into its PCB and loading the execution context of the new process.
- Switch is pure overhead



# Interrupts

(1)

- Interrupt is an event generated by the hardware that alters the sequence in which a processor executes instructions
  - Synchronous interrupts are caused by an event related to executing a current process's instructions.
  - Asynchronous interrupts are caused by an event unrelated with the execution of a current process's instructions

# Interrupts

(2)

- An interrupt handler is a set of instructions to be executed in response to each type of interrupt.
- Benefits of Interrupts
  - Interrupt provides a low-overhead means of gaining the attention of the CPU. This eliminates the need for the CPU to remain busy polling to see if devices require its attention
  - Polling vs. interrupt



# Interrupt Processing

- ❑ An interrupt occurs
- ❑ OS gains control. It saves the state of the interrupted process to its PCB and puts the process into the ready queue
- ❑ OS passes control to the appropriate interrupt handler to handle the interrupt
- ❑ The interrupt handler performs appropriate actions based on the type of interrupt.
- ❑ The state of the interrupted process (or some “next” process) is restored, and this next process executes

# Interrupt Classes

- Hardware-generated interrupts
  - I/O
  - Timer
  - Interprocessor interrupts
- Software-generated interrupts – Exceptions
  - Fault
  - Trap
  - Abort

# Interprocess Communication (IPC)

- Essential for processes that must coordinate activities to achieve a common goal.
- Implementation
  - Signals
  - Message passing

# Signals

- Signals are software interrupts that notify a process that an event has occurred.
  - It does not allow processes to exchange data with other processes.
- A process may catch, ignore or mask a signal.
  - Catch – A routine specified by the process is called by the system when it delivers the signal.
  - Ignore – the process relies on the system's default action to handle the signal.
    - Abort
    - Memory dump.
  - Mask – used to block a particular signal.

# Message passing

*Send (receiverProcess, message)*

*Receive (senderProcess, message)*

- Synchronous communication – blocking send
- Asynchronous communication – non-blocking send
- Broadcast

# Summery

- What is a process?
- Process states
- Process Control Block (PCB)
- Process queues and scheduler
- Process operations
- Context switching
- Interrupts
- Interprocess communication (IPC)

# Chapter 4

## Thread Concepts

# What is a thread?

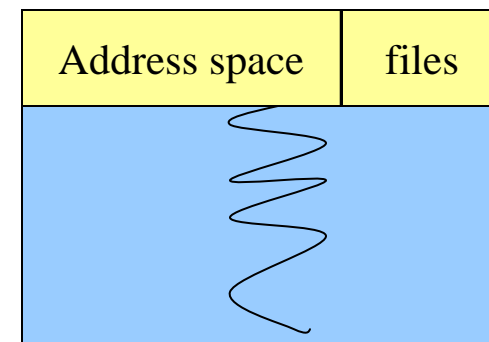
- A thread, also called a lightweight process, is a basic unit of CPU utilization that executes using the program and other resources of its associated process
  - Threads within a process share many of the process's resources such as its address space and open files
  - Several threads can be associated with one process
  - Threads allow a process to do more than one task at a time



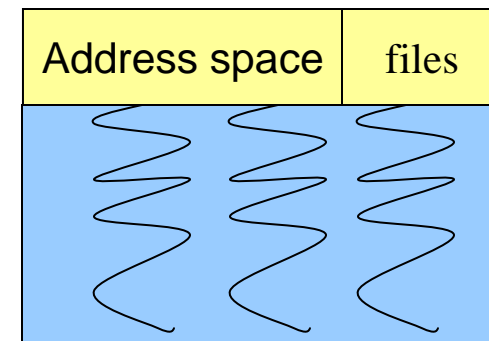
# Thread relationship to process

- A thread has its own
  - Thread ID
  - registers
  - Stack
  - Signal masks
  - TSD(Thread-Specific Data)
- It shares with other threads associated with the same process
  - Address space
  - Open files

Process of one thread



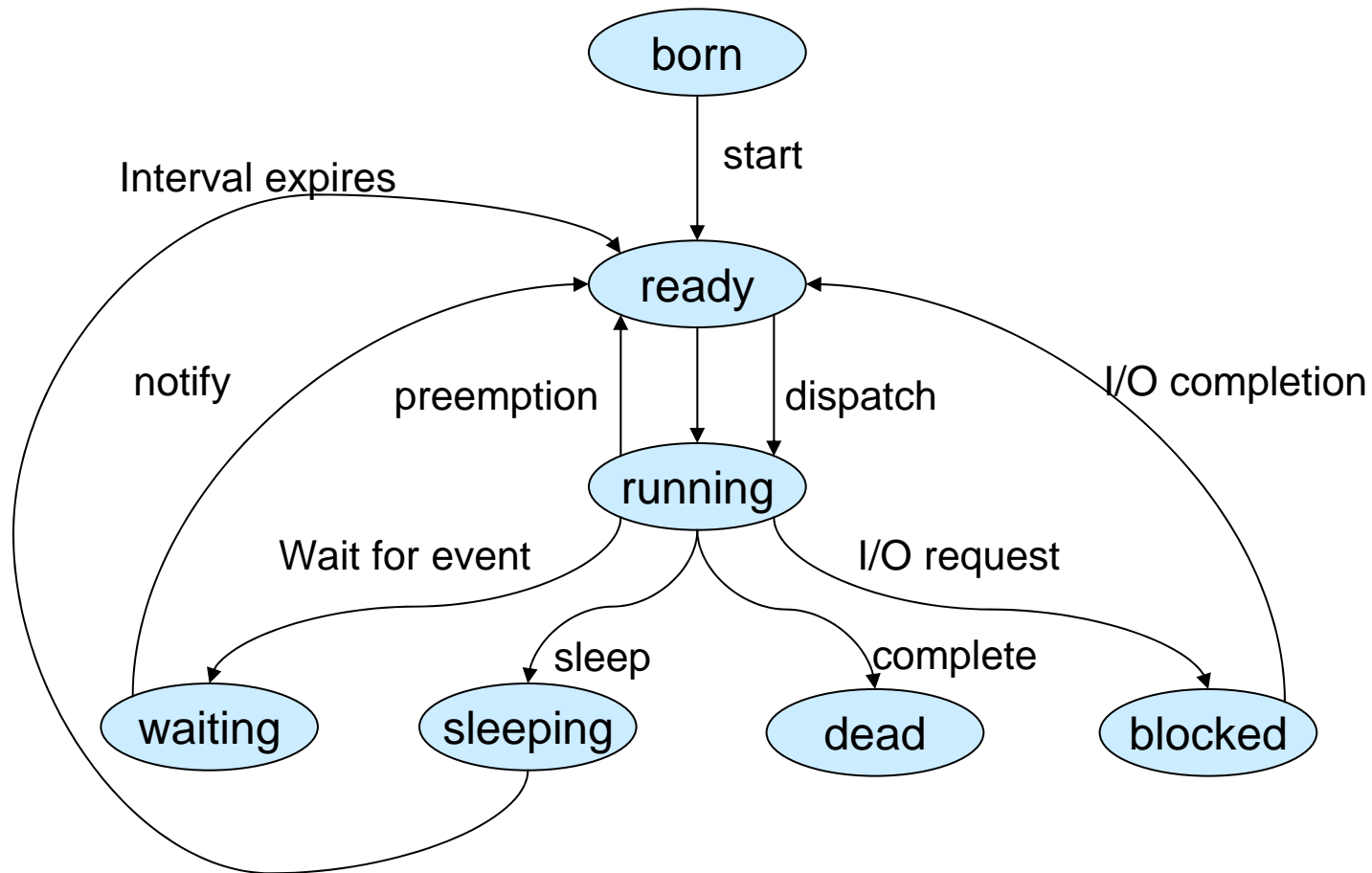
Process of multiple threads



# Motivation for Threads

- Software design – Separating independent code segments into individual threads can improve performance
- Performance – Multiple threads can share a processor (or a set of processors) so that tasks are performed in parallel.
- Cooperation – Threads can communicate using their shared address space.

# Thread States



# Threads Operations

- Common operations
  - Create
  - Exit
  - Suspend and resume
  - Sleep and wake
- Some OS also has
  - Cancel
  - Join

# Threading Models

- Thread implementations vary among operating systems.
- Three primary threading models
  - User-level threads
  - Kernel-level threads
  - A combination of user- and kernel-level threads

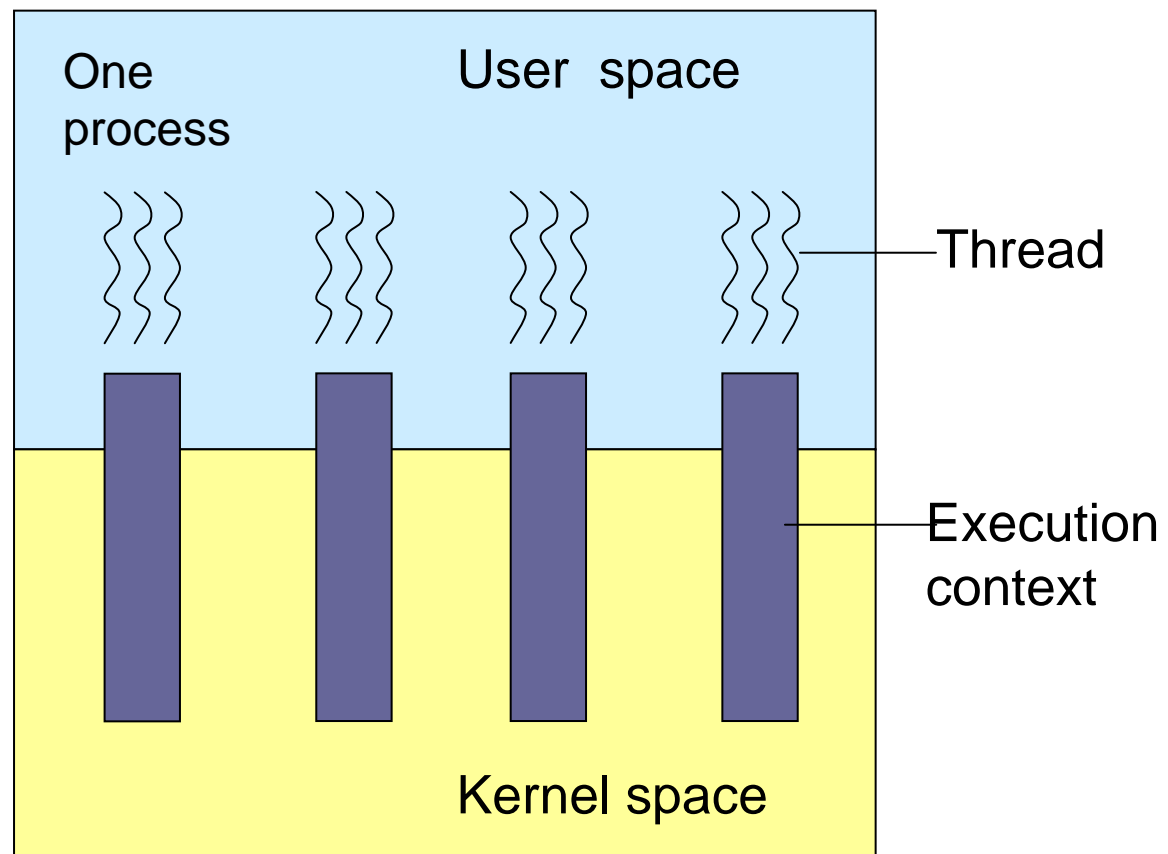
# User-level Threads

(1)

- This is also called many-to-one thread mapping.
- The user-level threads perform threading operations in user space.
  - They are created by runtime libraries that cannot execute privileged instructions or access kernel primitives directly.
  - Each multithreaded process is responsible for maintaining its threads.
  - The system treats each multithreaded process as a single execution unit.

# User-level Threads

(2)



# User-level Threads

(3)

## ■ Benefits

- Better portability.
- It is easier for a developer to control the scheduling to meet a specific requirements of an application
- Low overhead

## ■ Drawbacks

- Performance is not guaranteed.
  - The multithreaded process is viewed as a single thread of control by the system
  - The entire multithreaded process blocks when any of its threads request a blocking I/O operation.
  - Thread scheduling priority is not supported systemwide.



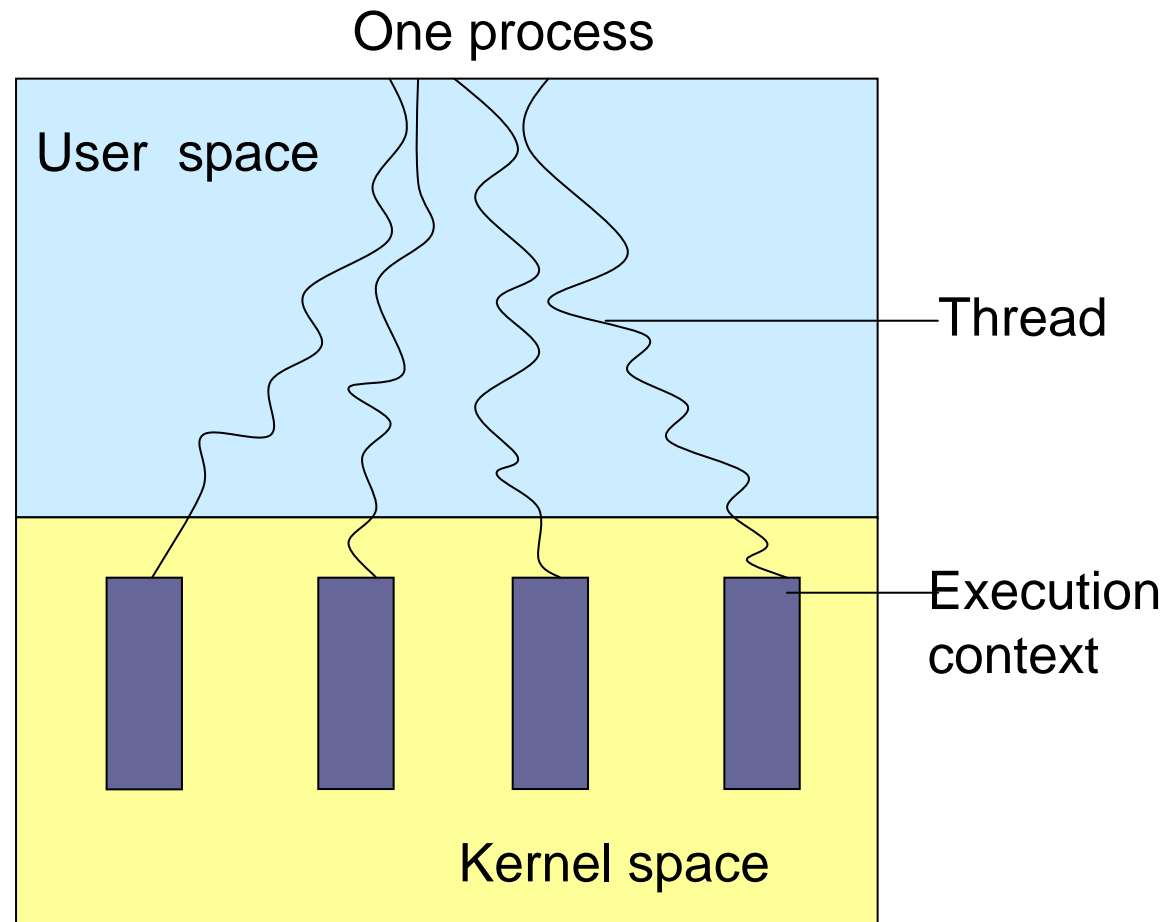
# Kernel-level Threads

(1)

- This is also called one-to-one thread mapping
- The operating system provides each user thread with a kernel thread that the system can dispatch
  - Each kernel thread has its own execution context
  - Kernel thread is managed by the system

# Kernel-level Threads

(2)



# Kernel-level Threads

(3)

## ■ Benefits

- Better performance
- Each thread can be managed individually.
  - Other threads can be dispatched when one thread is blocked.
  - Better interactivity
- Systemwide priority is supported

## ■ Drawbacks

- More overhead
- Less portability
- Uses more resources

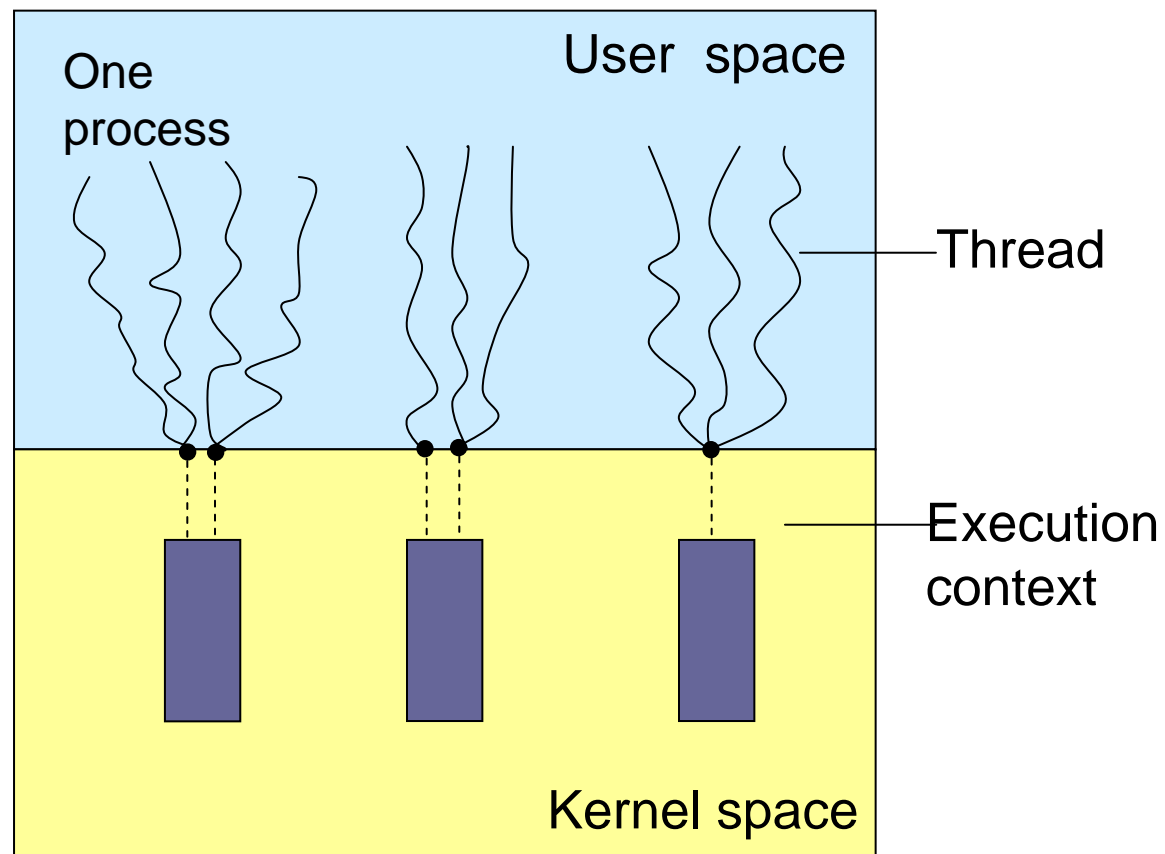
# Combining User- and Kernel-level Threads (1)

- This is also called many-to-many thread mapping.
  - It maps many user-level threads to a set of kernel threads.
  - The number of user threads and the number of the kernel threads need not be equal.
- Implementation
  - Thread pooling
    - One-to-one mapping requires that one data structure is allocated to represent each kernel thread – Too much overhead

# Combining User- and Kernel-level Threads (2)

- Thread pooling allows an application to specify the number of kernel threads it requires. Many-to-one mapping can be used for threads that exhibit a low degree of parallelism.
- It allow the kernel threads to remain in the system after a user thread dies. The kernel thread can then be allocated to a new user thread.
- Worker threads – the persistent kernel threads that typically performs several different functions.

# Combining User- and Kernel-level Threads (3)



# Thread Implementation Issues (1)

## ■ Thread Signal Delivery

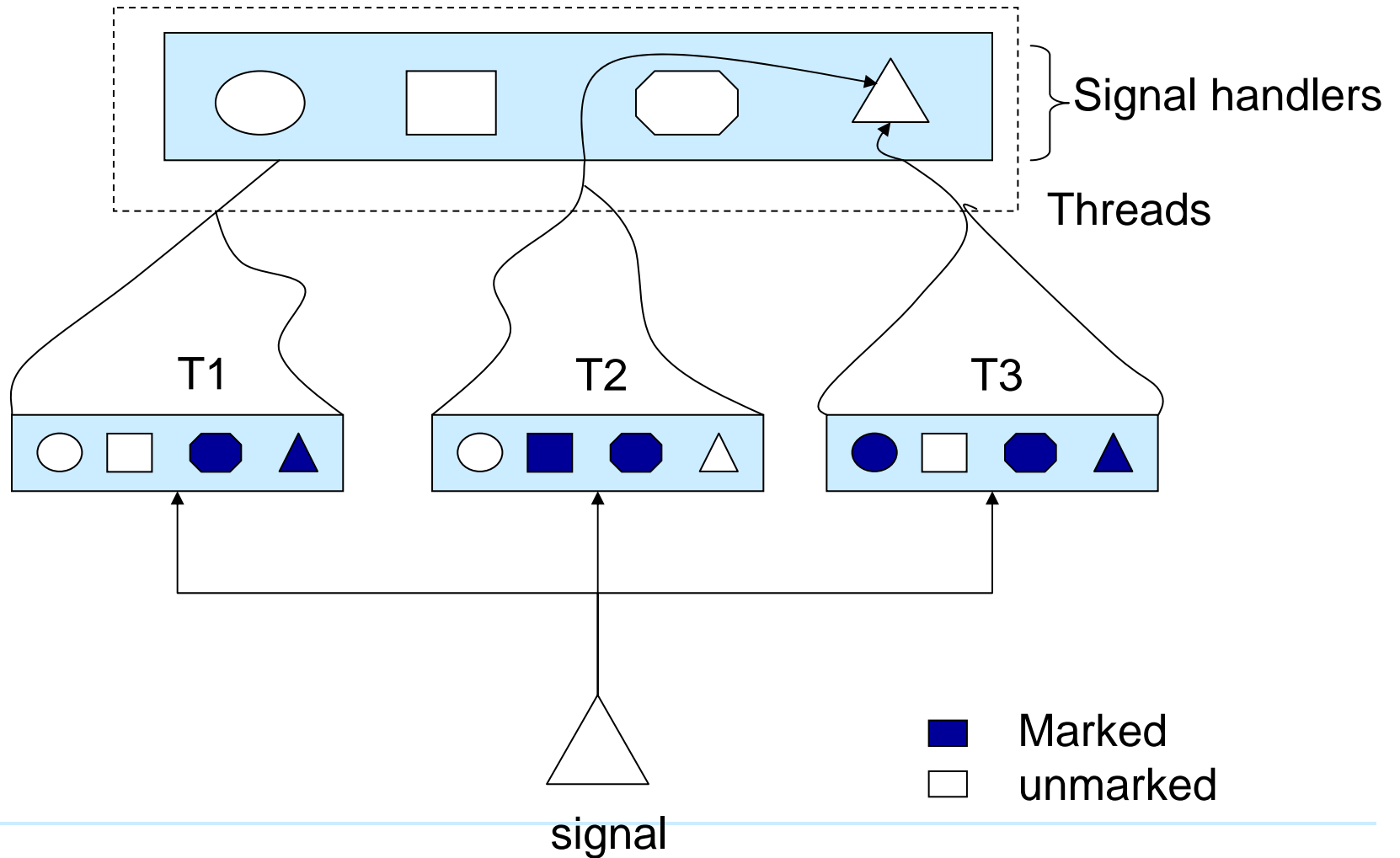
### □ Signal types

- A synchronous signal occurs as the direct result of an instruction executed by the current process or thread.
- An asynchronous signal occurs due to an event unrelated to the current instruction.

### □ Each signal must specify a process ID to indicate the signal recipient.

### □ A signal mask allows a thread to disable signals of particular types.

# Thread Implementation Issues (2)





# Thread Implementation Issues (2)

- Thread termination
  - Normal termination:
    - the operating system can immediately remove the thread from the system
  - Premature termination
    - Threading libraries determine how and when to remove the thread from the system.

# Java Threads

(1)

- There are two ways to create a new thread of execution.
  1. Declare a class to be a subclass of Thread.
    - This subclass should override the run method of class Thread.
    - An instance of the subclass can then be allocated and started.
  2. Declare a class that implements the Runnable interface.
    - This class then implements the run method.
    - An instance of the class can then be allocated, passed as an argument when creating Thread, and started.

**Note:** See <http://java.sun.com/j2se/1.4.2/docs/api/index.html> for more details.

# Java Class Thread

- `java.lang.Thread`
  - Has several constructors, such as
    - `public Thread();`
    - `public Thread(String threadName);`
    - `Public Thread(Runnable target);`
    - `Public Thread(Runnable target, String threadName);`(Automatically generated names are of the form "Thread-"+ $n$ , where  $n$  is an integer. )
  - Methods
    - `run()` – The code placed in the run method does the real work of the thread.
    - `start()` – Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
    - `sleep(long millis)` – Causes the currently executing thread to sleep for the specified number of milliseconds.

# Java Threads --Example 1

```
class PrimeThread1 extends Thread {  
    long minprime;  
    PrimeThread1(long num) {  
        this.minprime=num;  
    }  
    public void run() {  
        //calculate the next prime larger than minprime  
    }  
}
```

```
PrimeThread1 p = new PrimeThread1(143);  
p.start();
```

# Java Interface Runnable

- **Java.lang.Runnable**
  - should be implemented by any class whose instances are intended to be executed by a thread.
  - The class must define a run method.

# Java Threads --Example 2

```
class PrimeThread2 implements Runnable {  
    long minprime;  
    PrimeThread2(long num) {  
        this.minprime=num;  
    }  
    public void run() {  
        //calculate the next prime larger than minprime  
    }  
}
```

```
PrimeThread p = new PrimeThread(143);  
new Thread(p).start();
```

# Summery

- What are threads?
- Motivation
- Thread states
- Thread operations
- Thread Models
  - Kernel-level threads, user-level threads and combination of user- and kernel-level threads
- Thread implementation issues
- Java Threads (two methods)

# Chapter 5:

# Asynchronous Concurrent

# Execution



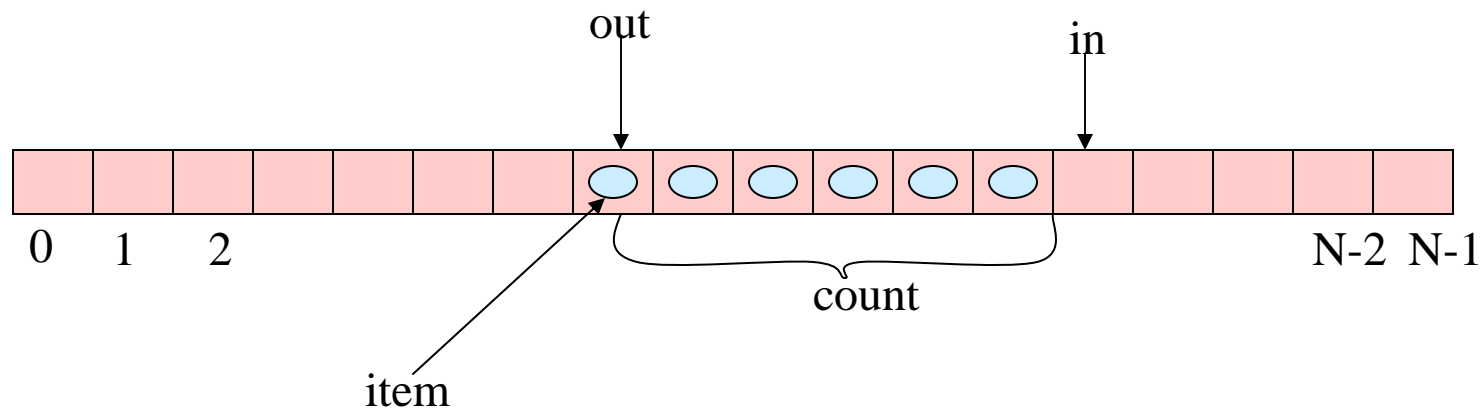
---

# Introduction

- **Concurrent:** threads that co-exist in a system at the same time
- **Asynchronous:** threads that operate independently of one another but must occasionally communicate and synchronize to perform cooperative tasks.

# Bounded-Buffer Producer and Consumer Problem

- Assume we have a bounded-buffer with fixed size of  $N$ . Producer adds items to the buffer and consumer takes items from the buffer



# Bounded-Buffer Producer and Consumer Problem – A solution

## Producer thread

```
While (count < N){  
    ....  
    item = produce an item;  
    ..  
    count++;  
    buffer[in]=item;  
    in = (in+1)%N;  
}
```

## Consumer thread

```
While (count >0){  
    count--;  
    item =buffer[out];  
    out = (out+1)%N;  
    ...  
    consume item  
    ...  
}
```

Note that both threads shares variable *count*. What happens if both routines are executed concurrently?

# Bounded-Buffer Producer and Consumer Problem – A solution

Low-level Language:

Count++;

```
R1 = count;  
R1 = R1 + 1;  
count = R1;
```

Count --;

```
R2 = count;  
R2 = R1 - 1;  
count = R2;
```

Assume count=5. Here is one possible interleaving:

|    |          |              |             |
|----|----------|--------------|-------------|
| s1 | producer | R1 = count;  | { R1=5 }    |
| s2 | producer | R1 = R1 + 1; | { R1=6 }    |
| s3 | consumer | R2 = count;  | { R2=5 }    |
| s4 | consumer | R2 = R1 - 1; | { R2=4 }    |
| s5 | producer | count = R1;  | { count=6 } |
| s6 | consumer | count = R2;  | { count=4 } |

Incorrect!

# Race Conditions

- Race condition – this situation occurs when several threads access and manipulate the same data concurrently and where the outcome of the execution depends on the particular order in which the access takes place.
- Solution: Only one thread at a time can manipulate the shared modifiable variables. Thread synchronization and coordination are needed.

# Critical-Section

- Declare a section of code to be **critical**, in which a set of threads may be changing common variables, updating a table, writing a file.
- Regulate access to the critical section to guarantee that the execution of critical section by the threads is mutually exclusive in time.

# Solution for Critical Section Problems

- ❑ Mutual exclusion: Only one process is allowed to be executing in the critical section at one time
- ❑ Progress: If no process is executing in its CS, then one of the processes that wish to enter the critical section, and is not executing in the remainder section, should be selected to enter the critical section. This selection cannot be indefinitely postponed.
- ❑ Bounded waiting: A process should not wait indefinitely to enter its CS

# Mutual Exclusion Primitives

```
While(true){
```

```
    Enter critical section;
```

```
    Critical section;
```

```
    Exit critical section;
```

```
    Non-critical section;
```

```
}
```



# Dekker's Algorithm : Version 1 (1)

- Set a shared variable *turn* (initialized to 1). If  $turn=i$  then thread *i* is allowed to enter the critical section

Thread 1

```
While(true){  
    while(turn!=1) { wait;}  
    critical section;  
    turn = 2;  
    non-critical section;  
}
```

Thread 2

```
While(true){  
    while(turn!=2) { wait;}  
    critical section;  
    turn = 1;  
    non-critical section;  
}
```

# Dekker's Algorithm : Version 1 (2)

- Busy wait – waste CPU utilization.
- More drawbacks:
  - What happens if thread 2 needs to enter the critical section first?
  - After thread 1 exits critical section, it sets turn=2, which assumes that thread 2 is ready to enter the critical section. But thread 2 may still be busy doing something else.
  - What happens if one thread needs to enter the critical section more frequently than the other? (**lockstep synchronization**)

# Dekker's Algorithm : Version 2 (1)

- Eliminate the lockstep synchronization
- Replace the variable *turn* with the two variables:

boolean t1Inside = false;

boolean t2Inside = false;

*t1Inside* = true indicates that thread 1 is inside the critical section. *t2Inside* = true indicates that thread 2 is inside the critical section.

# Dekker's Algorithm : Version 2 (2)

## Thread 1

```
While(true){  
    while(t2Inside) {wait;}  
    t1Inside=true;  
    critical section;  
    t1Inside=false;  
    non-critical section;  
}
```

## Thread 2

```
While(true){  
    while(t1Inside) {wait;}  
    t2Inside=true;  
    critical section;  
    t2Inside=false;  
    non-critical section;  
}
```

# Dekker's Algorithm : Version 2 (3)

## ■ Drawbacks

- Doesn't guarantee mutual exclusion
  - What happens if both *t1Inside* and *t2Inside* are false and both threads attempt to enter their critical section at the same time?
- Busy wait

# Dekker's Algorithm : Version 3 (1)

□ Replace the variables *t1Inside* and *t2Inside* with the two variables:

boolean *t1WantToEnter* = false;

boolean *t2WantToEnter* = false;

*t1WantToEnter* = true indicates that thread 1 desires to enter the critical section. *t2WantToEnter* = true indicates that thread 2 desires to enter the critical section.

# Dekker's Algorithm : Version 3 (2)

## Thread 1

```
While(true){  
    t1WantToEnter=true;  
    while(t2WantToEnter) {wait;}  
    critical section;  
    t1WantToEnter=false;  
    non-critical section;  
}
```

## Thread 2

```
While(true){  
    t2WantToEnter=true;  
    while(t1WantToEnter) {wait;}  
    critical section;  
    t2WantToEnter=false;  
    non-critical section;  
}
```

## Dekker's Algorithm : Version 3 (2)

It is possible for the following situation to happen:

thread 1: set t1WantToEnter = true;

thread 2: set t2WantToEnter = true;

Thread 1 and 2 will loop forever in the while statement.

- Satisfies the mutual-exclusive requirement, but introduces deadlock.



# Dekker's Algorithm : Version 4 (1)

## Thread 1

```
while(true){  
    t1WantToEnter=true;  
    while(t2WantToEnter) {  
        t1WantToEnter=false;  
        wait for a certain time;  
        t1WantToEnter=true;  
    }  
    critical section;  
    t1WantToEnter=false;  
    non-critical section;  
}
```

## Thread 2

```
while(true){  
    t2WantToEnter=true;  
    while(t1WantToEnter) {  
        t2WantToEnter=false;  
        wait for a certain time;  
        t2WantToEnter=true;  
    }  
    critical section;  
    t2WantToEnter=false;  
    non-critical section;  
}
```

# Dekker's Algorithm : Version 4 (1)

- Satisfies mutual exclusion requirement and prevents deadlock, but it allows indefinite postponement.

It is possible for the following situation to happen:

thread 1: set t1WantToEnter = true;

thread 2: set t2WantToEnter = true;

Thread 1 and 2 will loop forever in the while statement.

# Dekker's Algorithm : Version 5 (1)

## — A proper solution

- A boolean variable for each thread to indicate its desire to enter the critical section:

`boolean t1WantToEnter = false;`

`boolean t2WantToEnter = false;`

- An int variable to indicate the favoured thread that will enter the critical section.

`int favouredThread = 1;`

# Dekker's Algorithm : Version 5 (2)

## Thread 1 – A proper solution

```
while(true){
    t1WantToEnter=true;
    while(t2WantToEnter) {
        if(favouredThread==2){
            t1WantToEnter=false;
            while(favouredThread==2){wait;}
            t1WantToEnter=true;
        }
        critical section;
        favouredThread=2;
        t1WantToEnter=false;
        non-critical section;
    }
}
```

## Thread 2

```
while(true){
    t2WantToEnter=true;
    while(t1WantToEnter) {
        if(favouredThread==1)
            t2WantToEnter=false;
        while(favouredThread==1){wait;}
        t2WantToEnter=true;
    }
    critical section;
    favouredThread=1;
    t2WantToEnter=false;
    non-critical section;
}
```

## Dekker's Algorithm : Version 5 (2)

### – A proper solution

- Satisfies mutual exclusion requirement and prevents deadlock and indefinite postponement.
- But, what happens if the system pre-empt thread 1 when thread 1 has exited the inner wait loop and **JUST AFTER** it sets *t1WantToEnter* to *true*?

# Peterson's Algorithm

(1)

- A simpler algorithm for enforcing two-process mutual exclusion with busy wait.
- Has the same global variables as Dekker's algorithm.

```
boolean t1WantToEnter = false;  
boolean t2WantToEnter = false;  
int favouredThread = 1;
```

# Peterson's Algorithm

(2)

Thread 1

```
while(true){
    t1WantToEnter=true;
    favouredThread=2;
    while(t2WantToEnter &&
        favouredThread==2 ) {
        wait;
    }
    critical section;
    t1WantToEnter=false;
    non-critical section;
}
```

Thread 2

```
while(true){
    t2WantToEnter=true;
    favouredThread=1;
    while(t1WantToEnter &&
        favouredThread==1 ) {
        wait;
    }
    critical section;
    t2WantToEnter=false;
    non-critical section;
}
```

# Peterson's Algorithm

(3)

- Satisfies mutual exclusion requirement and prevents deadlock and indefinite postponement.
- Busy wait



## Hardware solutions

### – Disable Interrupts

- The reason mutual exclusion is needed is largely because pre-emption allows multiple threads to work on shared data asynchronously.
- Threads are typically pre-empted by interrupts.
- Not practical. (what if a thread enters an indefinite loop in its critical section?)

## Hardware solutions

### – Test-and-Set Instruction

- A special hardware instruction – Test-and-Set instruction enables a thread to perform some operations atomically.
- To allow a thread to enter a critical section, the Test-and-Set instruction will read a flag to determine that no other thread is executing in the critical section. Then it sets a lock to indicate that the thread is executing in the critical section.

# Mutual Exclusion with Semaphores (1)

- A semaphore is an integer variable that, apart from initialization, is accessed only through two standard operations: P and V
- Definitions of P(S) and V (S):

```
P(S){  
    if (S>0)  S=S-1  
    else    put the calling thread into the semaphore S's waiting queue  
}
```

```
V(S){  
    if any threads are waiting in the semaphore S's waiting queue  
        Resume the "next" waiting thread from the queue  
    else  S=S+1  
}
```

## Mutual Exclusion with Semaphores (2)

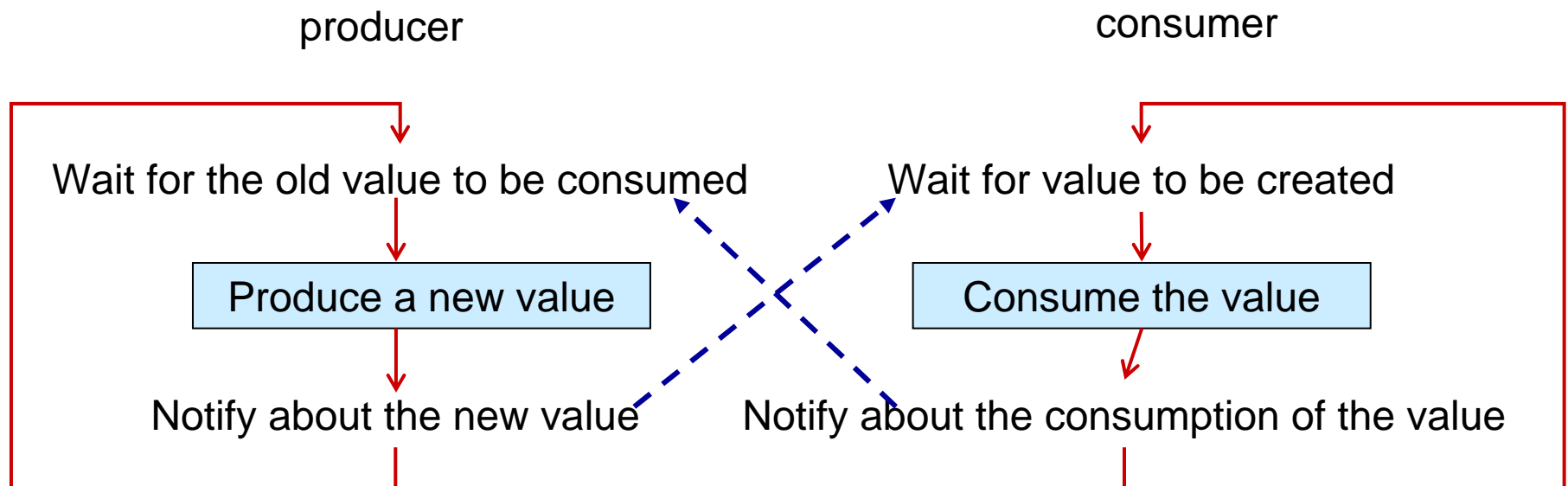
- Usage:

```
Semaphore S;  
P(S);  
Critical section;  
V(S)
```

- Mutual exclusion on a semaphore is enforced within P and V
- P and V are atomic operations which means they must be executed indivisibly

# Thread Synchronization with Semaphores

- Semaphores can be used to synchronize two or more concurrent threads.
- Producer/Consumer problems



# Thread Synchronization with Semaphores

- Use semaphore to notify the occurrence of an event:

Thread T1 wants to be notified about the occurrence of an event.

Thread T2 is capable of detecting the occurrence of the event and then notify its occurrence to the threads who are interested in the event.

Use a semaphore S, initialized to 0.

T1: P(S), this causes T1 to block

T2: V(S), this signals the occurrence of the event and allows T1 to proceed.

# Producer/Consumer relationship implemented with semaphores (1)

## Global variables:

- ❑ Semaphore *valueProduced* for the “value has been produced” event. It is initialised to 0.
- ❑ Semaphore *valueConsumed* for the “value has been consumed” event. It is initialised to 1.
- ❑ Int *sharedValue* is a variable shared by both producer and consumer threads.

# Producer/Consumer relationship implemented with semaphores (2)

```
While(true){  
  int nextValueProduced  
  While(true){  
    nextValueProduced=create a new value  
    P(valueConsumed)  
    sharedValue=nextValueProduced;  
    V(valueProduced)  
  }  
}
```

Producer thread

Consumer thread

```
while(true){  
  int nextValueConsumed  
  While(true){  
    nextValueProduced=create a new value  
    P(valueProduced)  
    nextValueConsumed=sharedValue  
    V(valueConsumed);  
    consume the value  
  }  
}}
```



# Bounded-Buffer Problem

- *mutex* semaphore provides mutual exclusion for accesses to the buffer pool. It is initialized to 0.
- *empty* and *full* semaphores count the number of empty and full buffers respectively. Initially,  $empty=N$ ,  $full=0$ . ( $N$  is the buffer size.)

## Producer thread

```
While (true) {  
    .. Produce an item ...  
    P(empty); P(mutex);  
    .. Add the item to buffer ...  
    V(mutex); V(full);  
}
```

## Consumer thread

```
While (true) {  
    P(full); P(mutex);  
    .. remove an item from buffer;  
    V(mutex); V(empty);  
    ...Consume the item ..  
}
```

# The Readers and Writers Problem

- A file is to be shared among several concurrent threads. Some of those threads (readers) may want only to read the file, whereas others (writers) may want to update the file.
- Two variations
  - No reader will be kept waiting unless a writer has already obtained permission to use the file.
  - A writer can always perform its write as soon as possible.

# The Readers and Writers Problem – Data Structure

*Semaphore mutex=1, wrt=1;*

*int readcount = 0;*


*mutex* semaphore is used to ensure mutual exclusion when the variable *readcount* is updated; *wrt* semaphore functions as a mutually exclusive semaphore for the writers. It is also used by the first and last reader that enters or exits the critical section. The *readcount* variable keeps track of how many readers currently are reading the file.

# The Readers and Writers Problem – Reader and Writer Processes

A yellow cloud-shaped callout bubble with a black outline, containing the text "Reader Thread". It is connected to the Reader Thread code block by three small circles of increasing size.

Reader Thread

```
P(mutex);  
if(readcount<=0){  
    P(wrt);  
}  
readcount ++;  
V(mutex);  
...  
reading is performed;  
....  
P(mutex);  
readcount--;  
if(readcount<=0) V(wrt);  
V(mutex);
```

A yellow cloud-shaped callout bubble with a black outline, containing the text "Writer Thread". It is connected to the Writer Thread code block by three small circles of increasing size.

```
P(wrt);  
...  
writing is performed;  
....  
V(wrt);
```

Writer Thread

# Summary

- Race conditions, Critical Section and Mutual exclusion
- Solutions for Critical Section problems
  - Dekker's algorithm (version 1-5)
  - Peterson's algorithm
  - Hardware solutions
  - Semaphores
- Thread synchronization with semaphores
  - Consumer and producer
  - Bounded-buffer problem
  - The readers and writers problem

# Chapter 6

## Concurrent Programming

# Deadlock and Starvation

- Deadlock is the situation where two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Starvation is a situation where processes wait indefinitely within the semaphore.

# The Dining Philosophers Problem (1)

- Consider five philosophers sitting around a circular table. There is a bowl of rice in the center of the table and 5 single chopsticks lay on the table. When a philosopher gets hungry, he tries to pick up the pair of chopsticks closest to him, one chopstick at a time. When he gets two chopsticks in his hands, he eats. He puts back the chopsticks after he has finished eating.



# The Dining Philosophers Problem (2)

- A simple solution is to represent each chopstick by a semaphore.
  - A philosopher tries to grab a chopstick by executing a P operation on that semaphore
  - a philosopher releases his chopstick by executing a V operation on the semaphore

# The Dining-Philosopher Problem (3)

```
Semaphore chopsticks[] = new Semaphore[5];

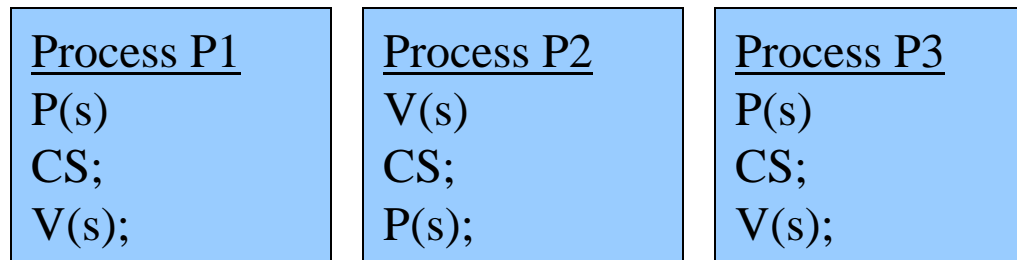
//process i
while(true) {
    P(chopsticks[i]);
    P(chopsticks[(i+1)mod5]);
    ...
    eat;
    ...
    V(chopsticks[i]);
    V (chopsticks[(i+1)mod5]);
    ...
    think;
    .....
}
```

# The Dining Philosophers Problem (4)

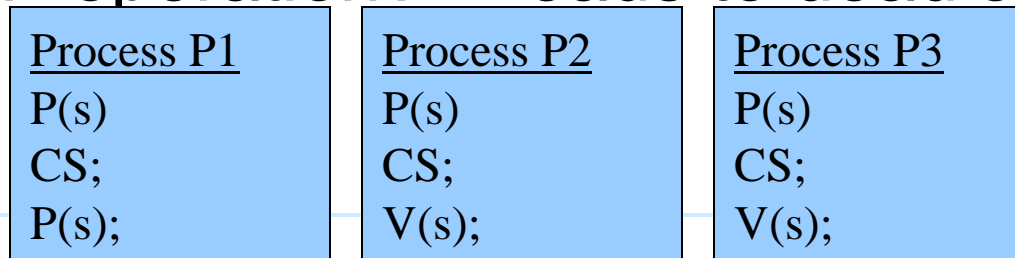
- Possibility of deadlock
  - What happens if all the philosophers are hungry at the same time, and they all grab the chopstick on their left, and then wait for the chopstick on their right?

# Semaphore – problems (1)

- What happens if a process interchanges the order of P and V operations? – not mutually exclusive any more.

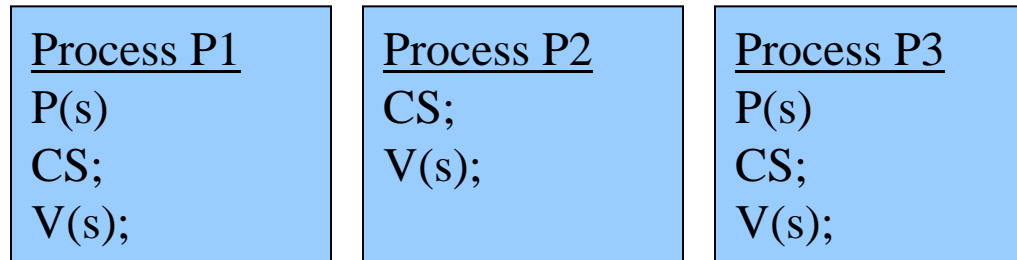


- What happen if a process replaces V operation with P operation? – leads to deadlock.

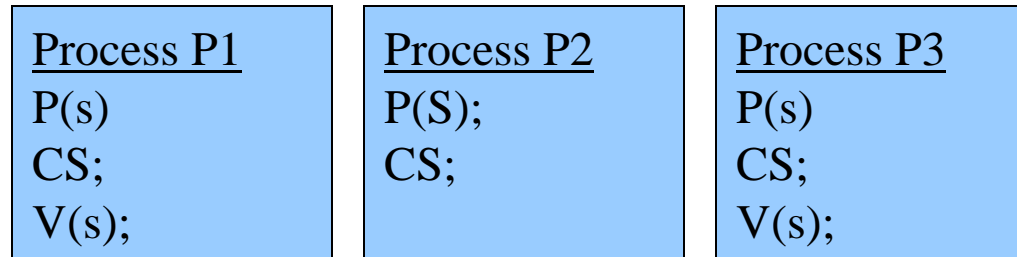


## Semaphore – problems (2)

- What happens if a process omits the P operation? – not mutually exclusive any more.



- What happens if a process omits the V operation? – leads to deadlock.



# Monitors

- A high-level synchronization construct in which mutual exclusion is rigidly enforced.
- The representation of a monitor type consists of
  - Declarations of variables
  - Declarations of functions that implement operations on the variables

# Monitors

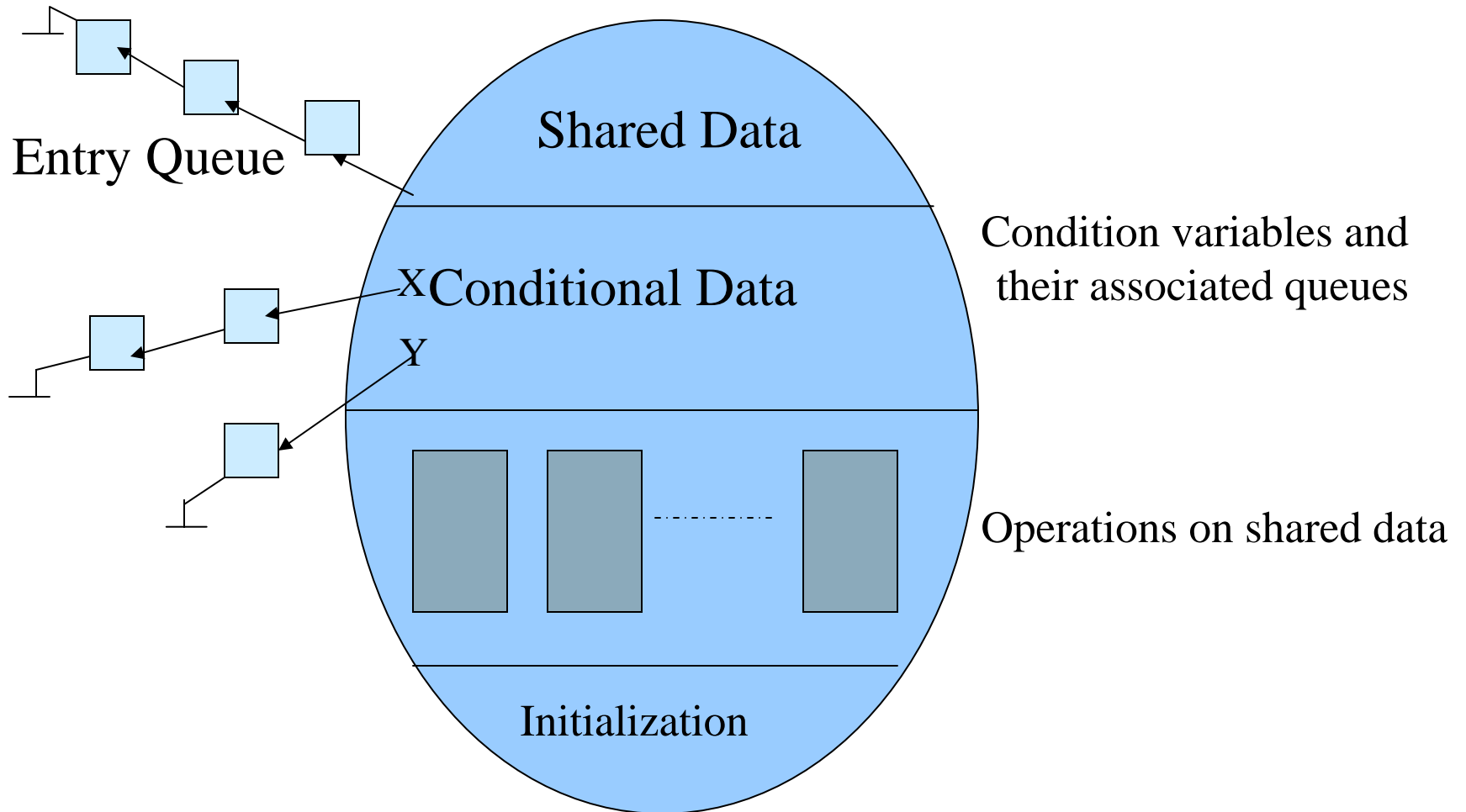
- Mutual exclusion is rigidly enforced at the monitor boundary:
  - A thread must call a monitor entry routine to enter the monitor.
  - If no other threads are executing inside the monitor, the monitor entry routine acquires a lock on the monitor for the calling thread and lets it enter the monitor.
  - If another thread is executing inside the monitor, the monitor is locked and the calling thread has to wait outside the monitor.
- A thread inside a monitor uses a conditional variable to wait on a condition outside the monitor.

## Monitors – Conditional Variables

- Each condition variable has a queue.
- If the condition is not right, the process calls *wait()* to exit the monitor and put itself into the queue associated with the condition.
- Processes calling *signal()* cause a waiting process to be removed from the queue and be ready to enter the monitor again.



# Monitors



# Bounded-Buffer Problem

(1)

- Variable declaration and initialization:

```
//Shared Data
```

```
int size = 100;
```

```
byte[] buffer = new byte[size];
```

```
int in=0, out=0, count=0;
```

```
//Conditional variables
```

```
ConditionalVariable notFull, notEmpty;
```

# Bounded-Buffer Problem

(2)

- Operation *deposit (byte item)* is used to put an item into the buffer.

```
void deposit(byte item){  
    if(count == size) wait (notFull);  
    buffer[in] = item;  
    in = (in+1) mod size;  
    count++;  
    if(count == 1) notify(notEmpty);  
}
```

## Bounded-Buffer Problem (3)

- Operation *fetch* () is used to fetch an item from the buffer.

```
byte fetch() {  
    if(count==0) wait(noEmpty);  
    byte item = buffer[out];  
    out = (out+1)% size;  
    count--;  
    if(count == size-1) notify(notFull);  
    return item;  
}
```

# Bounded-Buffer Problem (4)

- Assuming the instance of the monitor is called BFMonitor.
- The producer process:
  - ...produce a new item ...
  - BFMonitor.deposit(item);
- The consumer process:
  - item = BFMonitor.fetch();
  - ...consume the item...

# Dining Philosophers Problem (1)

- Variable declaration and initialization:
  - `static final int THINKING=0, HUNGRY=1, EATING=2;`
  - `//Shared data`
  - `int[] state = new int[5];`
  - `//Conditional variable`
  - `ConditionalVariable[] self = new ConditionalVariable[5];`

# Dining Philosophers Problem (2)

- Operation *pickup* (*int i*) is used by philosopher *i* to pick up the chopsticks on both sides so that he can eat.
  - `public void pickup( int i ) {`
  - `state[i] = HUNGRY;`
  - `test(i);`
  - `if( state[i] != eating )`
  - `wait( self[i] );`
  - `}`

# Dining Philosophers Problem (3)

- Operation *test* (*int i*) is a private function. It allows philosopher *i* to eat only when both his neighbors are not eating.

```
void test(int i){  
    if( state[left(i)] != EATING && state[i] == HUNGRY  
        && state[right(i)] != EATING){  
        state[i] = EATING;  
        notify(self[i]);  
    }  
}
```



# Dining Philosophers Problem (4)

- When a philosopher finishes eating, *putdown(int i)* is called to change his state and notify his two neighbors.

```
public void putdown(i){  
    state[i]=THINKING;  
    test(left(i));  
    test(right(i));  
}
```

# Dining Philosophers Problem (5)

- Assuming instance of the monitor is called DPMonitor.
- The processes for philosopher  $i$ :  
    DPMonitor.pickup(i);  
    ...Eating is performed ...  
    DPMonitor.putdown(i);

---

# Summary

- Deadlock and Starvation
- Semaphores (continued)
  - Dining-philosopher problem.
  - Problems
- Monitors
  - Definition and structure.
  - Bounded-buffer problem.
  - Dining philosophers problem.

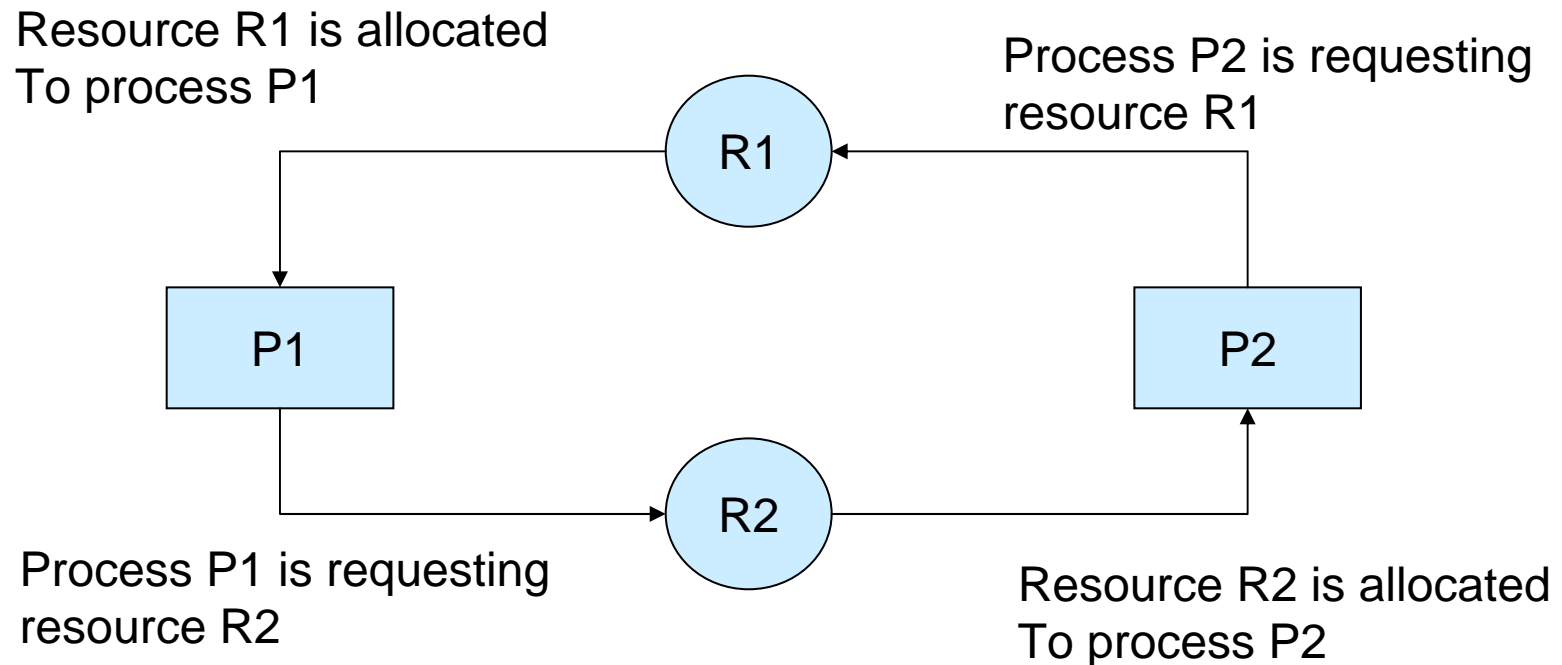
# Chapter 7

## Deadlock

# What is Deadlock?

- When several processes compete for a finite number of resources, a deadlock situation may arise where a process requests a resource and the resource is held by one of the waiting processes.
- A process is said to be deadlocked if it requests a resource that will not become available.

# Deadlock Example



Each process holds a resource being requested by the other process and neither process is willing to release the resource it holds

## Another example

- Assume there are resources R1 and R2 and processes P1 and P2. P1 and P2 execute concurrently.

```
P(1){  
    acquire R1;  
    acquire R2;  
    operate on R1 and R2;  
    release R1;  
    release R2;  
}
```

```
P(2){  
    acquire R2;  
    acquire R1;  
    operate on R1 and R2;  
    release R2;  
    release R1;  
}
```

# Deadlock in Spooling System

- A spooling system improves system throughput by dissociating a program from a slow device. It can be prone to deadlock.
  - For example, a spooling system sends pages to a faster device, such as a hard disk, before they are printed on a printer.
  - If the spooling system requires that the complete output of a program be available before printing can begin, several partially completed jobs can become deadlocked if the pages they generated have filled the available space for spooling.



---

## Related Problem: Starvation

- Starvation (or indefinite postponement) is a situation where processes wait indefinitely for a resource.

# Resource Concepts

- Resources can be
  - Preemptible – Can be removed from a process. Such as processors and main memory
  - Nonpreemptible – Cannot be removed from a process until the process voluntarily releases them. Such as printers, tape drives and scanners.
  - Shared – Can be shared by multiple processors.

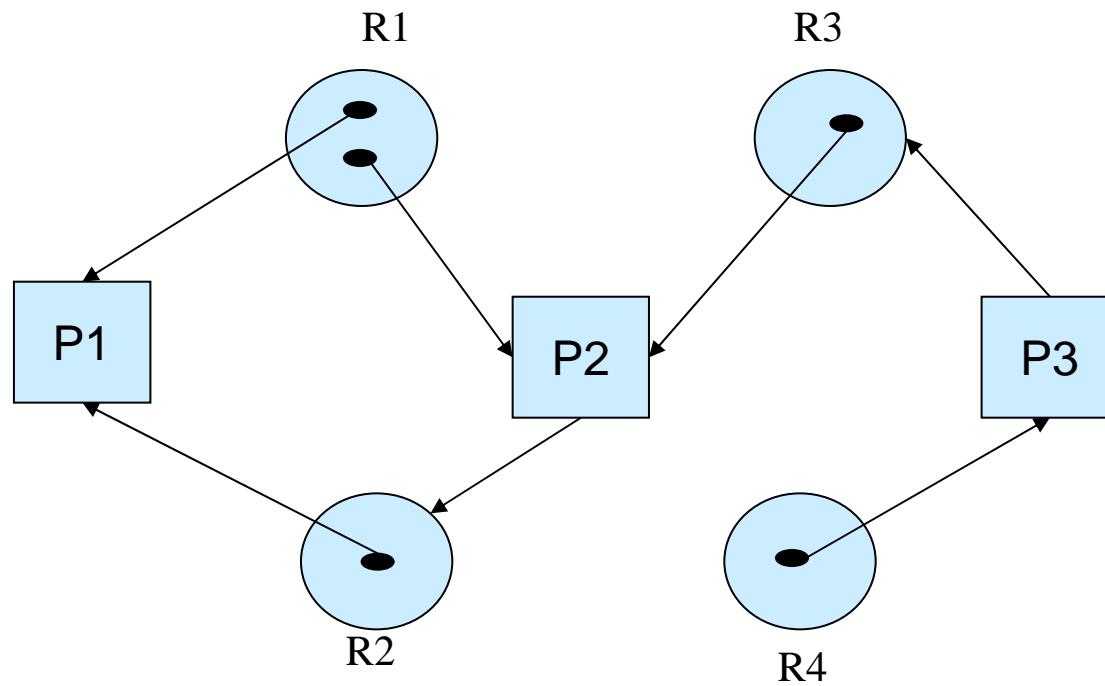
# Necessary Conditions

- Necessary conditions for deadlock
  - Mutual Exclusion
  - Hold and wait
  - No pre-emption
  - Circular wait
- All four conditions must hold for a deadlock to occur

# Resource Allocation Graph

- Vertices  $V = \{P, R\}$ 
  - $P = \{P_1, P_2, \dots, P_n\}$ : active processes in the system
  - $P = \{R_1, R_2, \dots, R_m\}$ : resource types in the system
- Directed edges
  - Request edges:  $P_i \rightarrow R_j$  signifies process  $P_i$  requested an instance of resource type  $R_j$  and is currently waiting for the resource
  - Assignment edges:  $R_j \rightarrow P_i$  signifies an instance of  $R_j$  has been allocated to process  $P_i$

# An example



# Detecting Deadlock in a Resource allocation Graph (1)

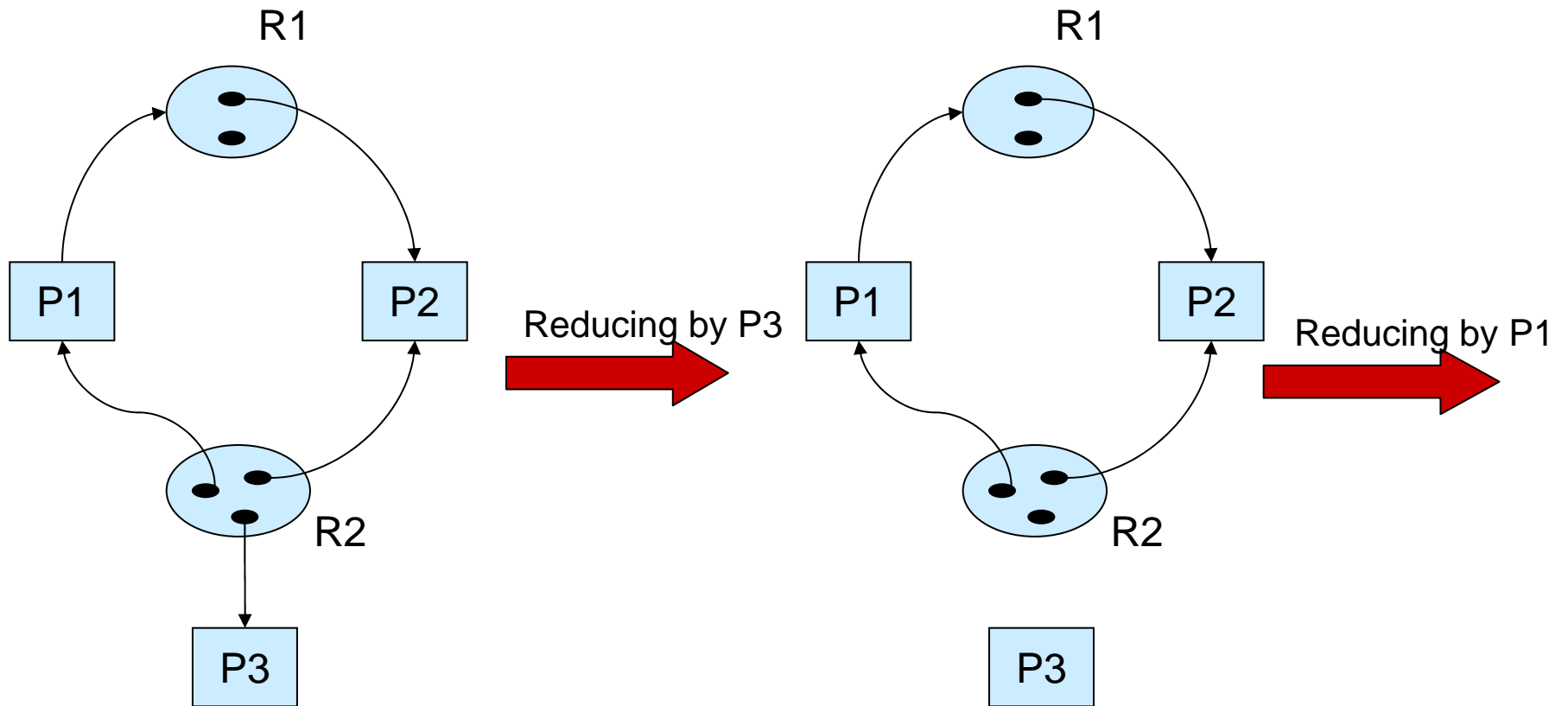
- In a resource-allocation graph in which each resource type only has one instance, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.
  - If the graph contains no cycles, then no process in the system is deadlocked.
  - If the graph contains a cycle, then deadlock exists.

## Detecting Deadlock in a Resource allocation Graph (2)

- Reduction of resource allocation graph
  - If a process's resource requests may be granted, we reduce the graph by the process by removing all the arrows to and from that process .
  - If a graph can be reduced by all its processes, then there is no deadlock, otherwise, the irreducible processes constitute the set of deadlock processes in the graph.

# Reduction of resource allocation graph

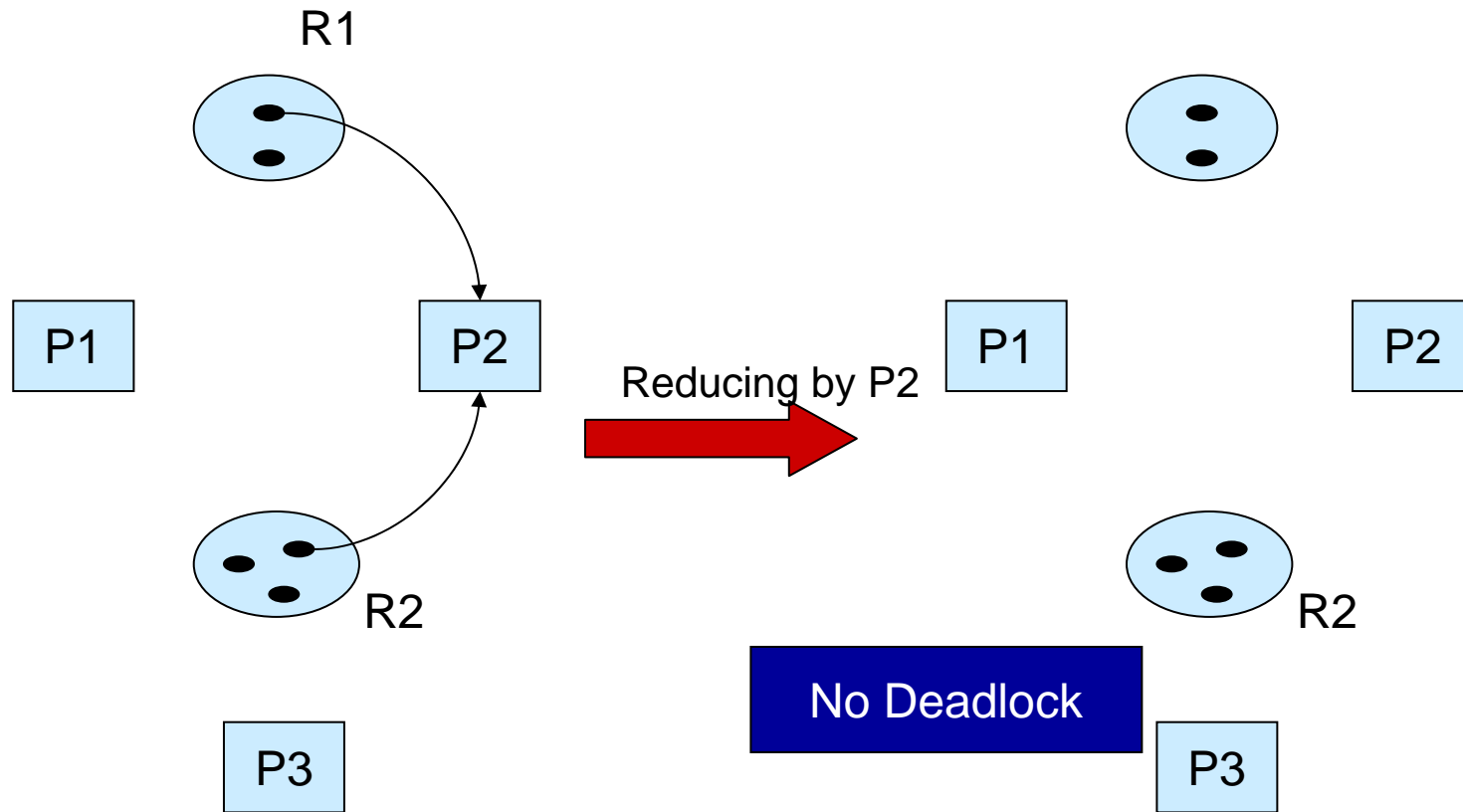
## – An Example (1)



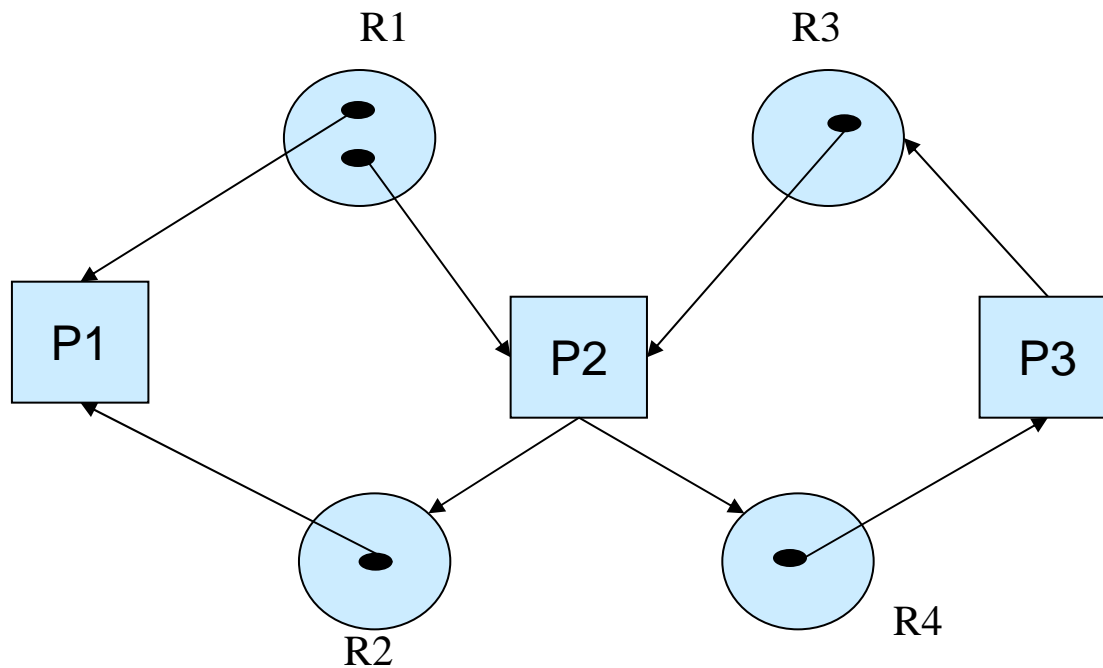


# Reduction of resource allocation graph

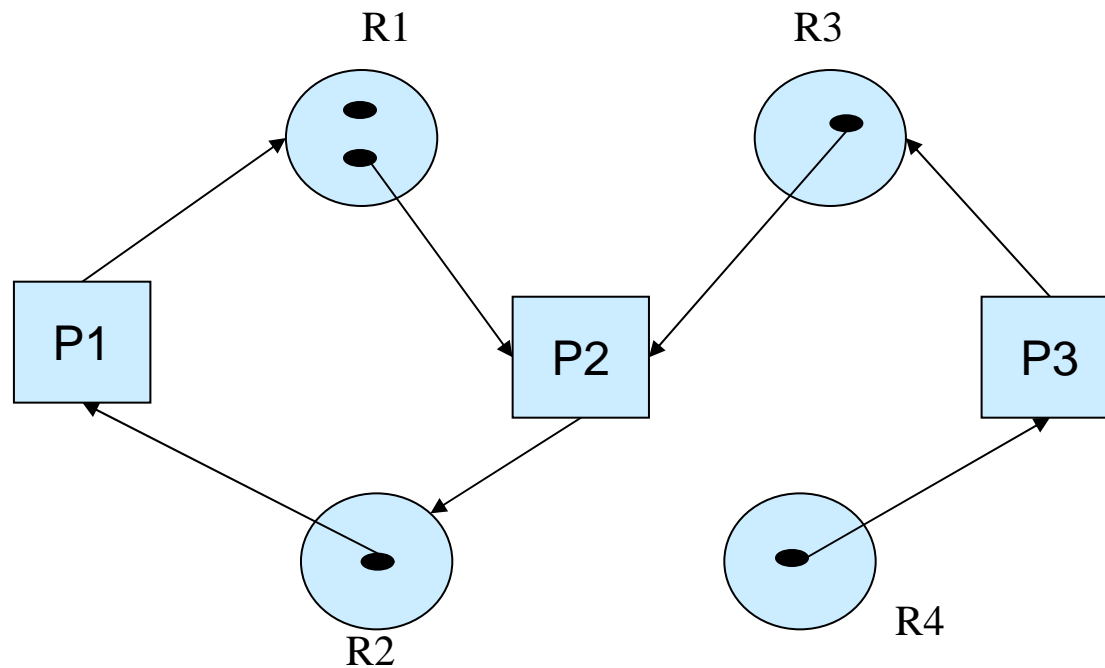
## – An Example (2)



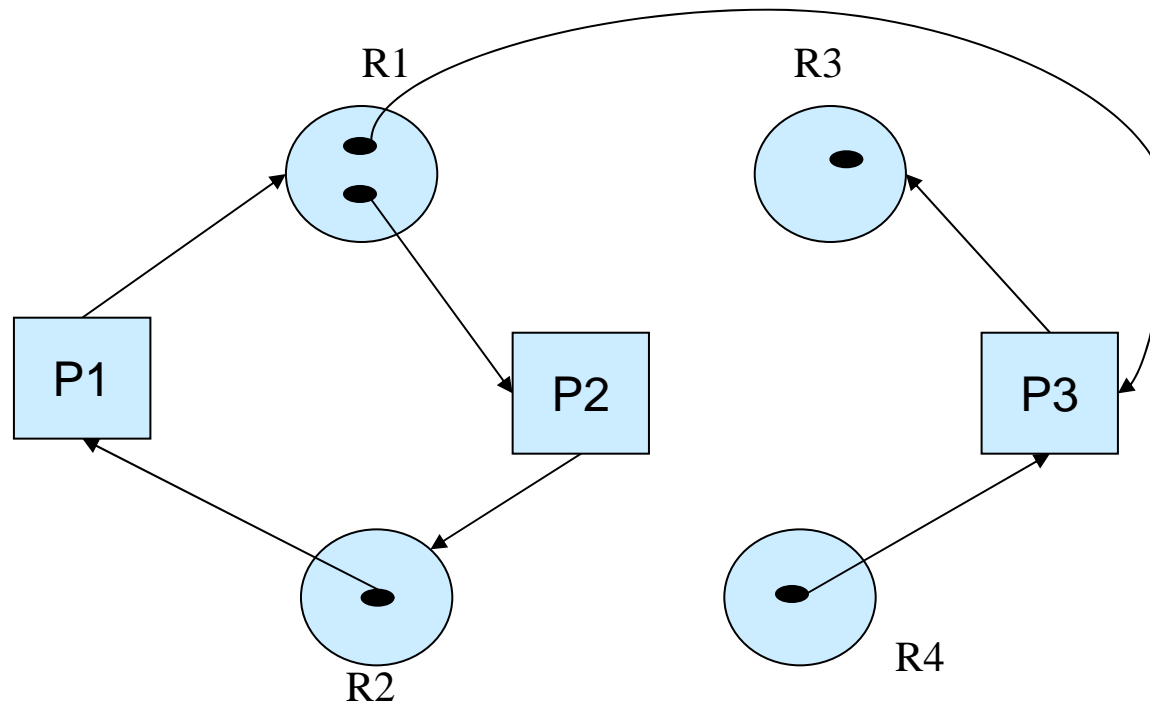
# Another Example



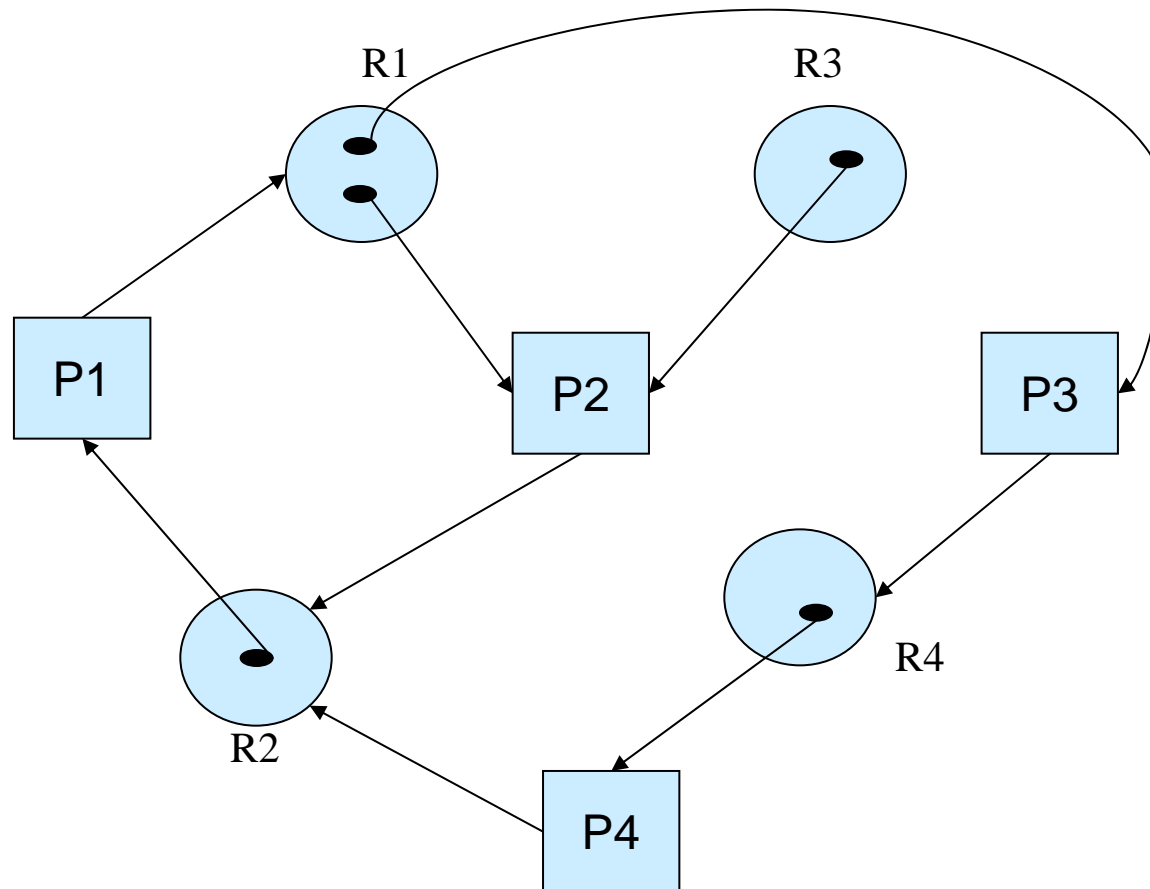
# Another Example again



And again



...again



# Handling Deadlock (1)

- Deadlock prevention
  - Prevent deadlocks by restraining how requests can be made.
- Deadlock Avoidance
  - Avoid deadlock by deciding whether a request for a resource can be satisfied or must wait to avoid a possible future deadlock. This method requires additional information about how resources are to be requested.

# Handling Deadlocks

(2)

- Deadlock detection
  - Identify the processes and resources that are involved if a deadlock has occurred.
- Deadlock Recovery
  - Clear deadlocks from a system so that the deadlocked processes may complete their execution and free their resources.

# Deadlock Prevention (1)

- Mutual exclusion: No mutually exclusive access to the sharable resources – Not reasonable.
- Hold and wait (two protocols):
  - Each process requests and is allocated all its resources before the start of its execution.
  - A process is allowed to request a resource only when it has none.
  - Disadvantages:
    - Resource utilization is low
    - Starvation



# Deadlock Prevention (2)

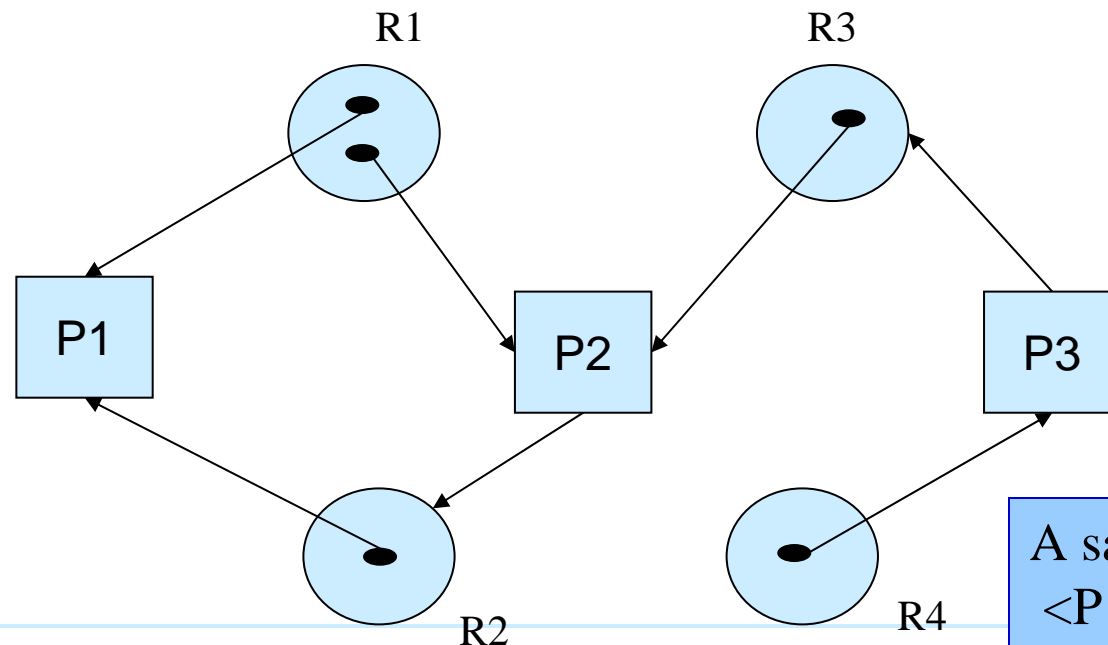
- No Pre-emption
  - If a process that is holding some resources requests another resource that cannot immediately be allocated to it, then all resources currently being held are pre-empted.
- Circular Wait
  - Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

## Safe state

- 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.

# Safe sequence

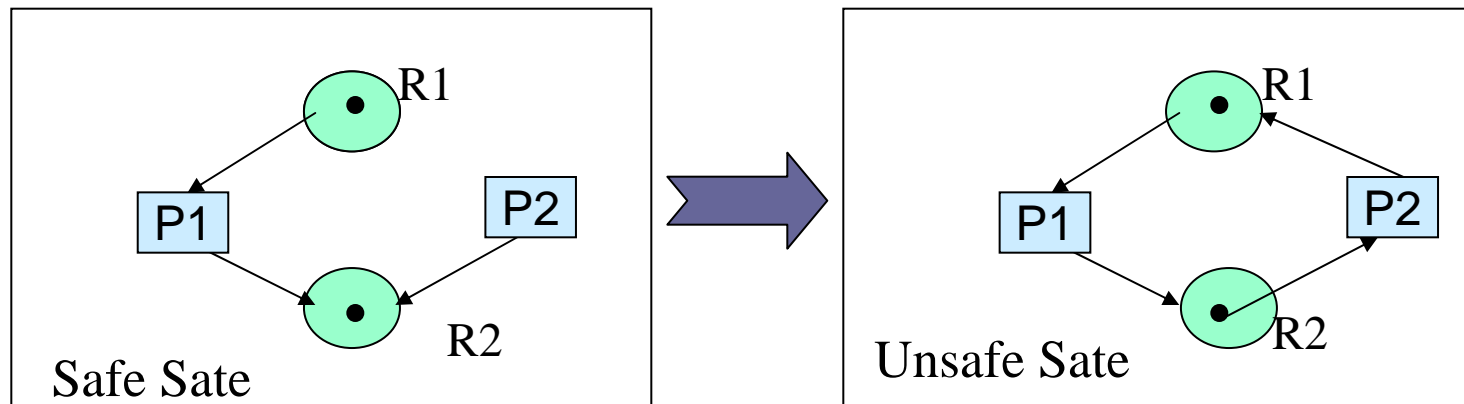
- A **safe sequence** is a sequence of processes  $\langle P_1, P_2, \dots, P_n \rangle$ , such that for each  $P_i$ , the resources that  $P_i$  can request can be satisfied by the current available resources plus the resources held by all the  $P_j$ , with  $j < i$ .



A safe sequence:  
 $\langle P_1, P_2, P_3 \rangle$

# Safe Sequence and Safe State

- A system is in a safe state only if there exists a **safe** sequence.
- If no safe sequence exists, the system state is **unsafe**. An unsafe state may lead to a deadlock.
- It is possible to go from a safe state to a unsafe state.



# Deadlock-avoidance Algorithms

- Deadlock-avoidance algorithms: ensure that the system is always in a safe state.

# Dijkstra's Banker's Algorithm (1)

- A system groups all the resources it manages into resource types. The resources of the same type provide identical functionality.
- Dijkstra's banker's algorithm for a system with only one type of resource. (It is easy to be extended to systems with multiple resource types.)

# Dijkstra's Banker's Algorithm (1)

- ❑ The system shares a fixed number of resources,  $t$ , and a fixed number of processes,  $n$
- ❑ Each process specifies in advance the maximum number of resources that it requires to complete its work
- ❑ The system accepts a process's request if that process's maximum need does not exceed the total number of resources available in the system,  $t$ .

# Example of a Safe State

- Suppose a system contains 12 equivalent resources and three processes sharing the resources. The resources are allocated as following

| Process | Maximum need | Current loan | Current claim |
|---------|--------------|--------------|---------------|
| P1      | 4            | 1            | 3             |
| P2      | 6            | 4            | 2             |
| P3      | 8            | 5            | 3             |



# Example of an Unsafe State

- Suppose a system contains 12 equivalent resources and three processes sharing the resources. The resources are allocated as following

| Process | Maximum need | Current loan | Current claim |
|---------|--------------|--------------|---------------|
| P1      | 10           | 8            | 2             |
| P2      | 5            | 2            | 3             |
| P3      | 3            | 1            | 2             |

# Summary

- What is deadlock?
- Deadlock's four necessary conditions
- Resource Allocation Graph
- Handling deadlock
  - Deadlock detection – Reduce RAG
  - Deadlock prevention and avoidance
  - Deadlock recovery
- Safe State
- Safe sequence
- Dijkstra's banker's algorithm

# Chapter 8

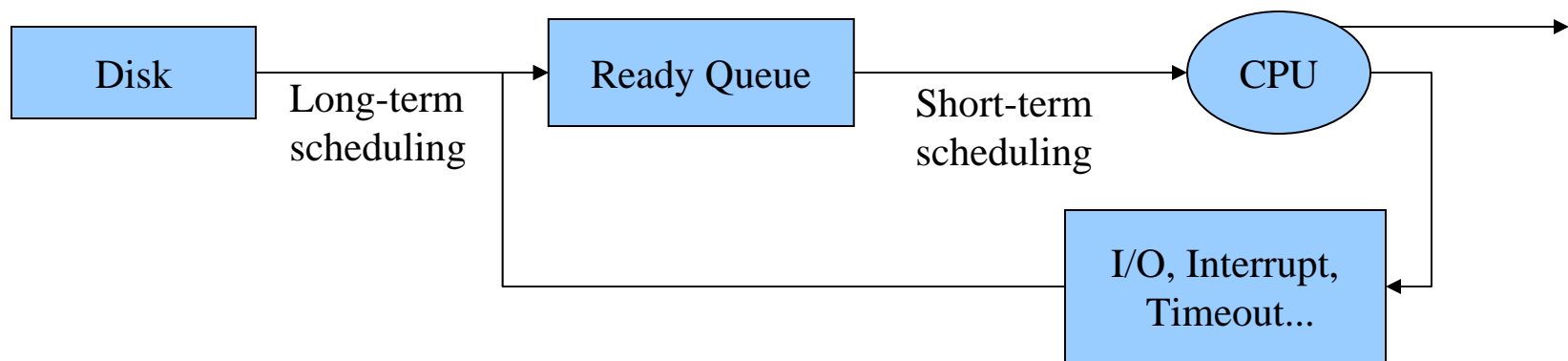
## Processor Scheduling

# Processor Scheduling

- Processor scheduling is the task of selecting a waiting process from the ready queue and allocating the processor to it.

# Scheduling Levels

- Long-term scheduler: selects processes from the process pool (usually on disk) and loads them into memory
- Short-term scheduler: selects processes from the ready queue



# CPU-I/O Burst Cycle

- Process execution consists of a cycle of processor execution and I/O wait.
- When one process does I/O, a scheduler will typically switch the processor to another process.

# When do scheduling decisions take place?

- When a process switches from running to waiting.
- When a process switches from running to ready.
- When a process switches from waiting to ready.
- When a process terminates.

# Preemptive vs. Non-preemptive Scheduling

- Non-preemptive Scheduling: Once the system has assigned a processor to a process, the system cannot remove that processor from that process
- Preemptive Scheduling: The system can remove the processor from the process it is running.



# Priorities

- Priorities are often used to determine how to schedule and dispatch processes.
  - Static priorities
  - Dynamic priorities

# Scheduling objectives

- Maximize throughput
- Minimize the time each process waits before executing
- Maximize resource utilization
- Avoid indefinite postponement
- Minimize overhead
- Ensure predictability
- Enforce priorities

# Scheduling Criteria

- Processor Utilization – as high as possible.
- Throughput – number of processes completed per time unit.
- Turnaround time – interval from submission to completion of process.
- Waiting time – time spent ready to run but not running.

# First-Come, First-Served (FCFS)

- Use FIFO queue.
- Non-preemptive – A process doesn't give up processor until it either terminates or performs I/O.
- Question: What if we have 3 processes: P1 (takes 24s), P2 (3), P3 (6). If arrive in order P1->P2->P3, What is waiting time? Turnaround time? Throughout? What if arrive in order P2->P3->P1?

# Answer

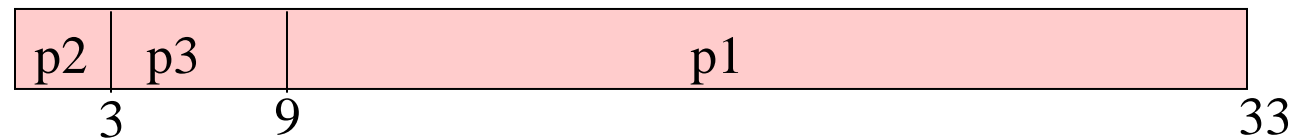
Gantt Chart



$$\text{Average waiting Time} = (0+24+27)/3 = 17$$

$$\text{Turnaround Time} = (24+27+33)/3 = 28$$

Gantt Chart



$$\text{Average waiting Time} = (0+3+9)/3 = 4$$

$$\text{Turnaround Time} = (3+9+33)/3 = 15$$

# Shortest-Job-First(SJF)

- Assign processor to the process that has the smallest next processor burst
- May be either preemptive or non-preemptive
- Optimal with respect to average waiting time
- How does scheduler know the length of the next processor burst?

---

# Priority Scheduling

- A priority is associated with each process, and the processor is allocated to the process with the highest priority. Normally low numbers represent high priority.
- May be preemptive or non-preemptive.
- May leave some low-priority processes waiting indefinitely for processor – Indefinite Blocking.
- Aging – a technique of gradually increasing the priority of processes that wait for a long time.

# Round-Robin(RR) Scheduling

- Similar to FCFS but with preemption.
- Have a time quantum (time slice). Let the first process in the queue run until it exceeds the time quantum, then run the next process.



## Round-Robin(RR) Scheduling

- Very small quantum – Context switch overhead.
- Very big quantum – turns into FCFS.
- Question: time quantum=4;

| Process | burst time |
|---------|------------|
| P1      | 8          |
| P2      | 3          |
| P3      | 10         |

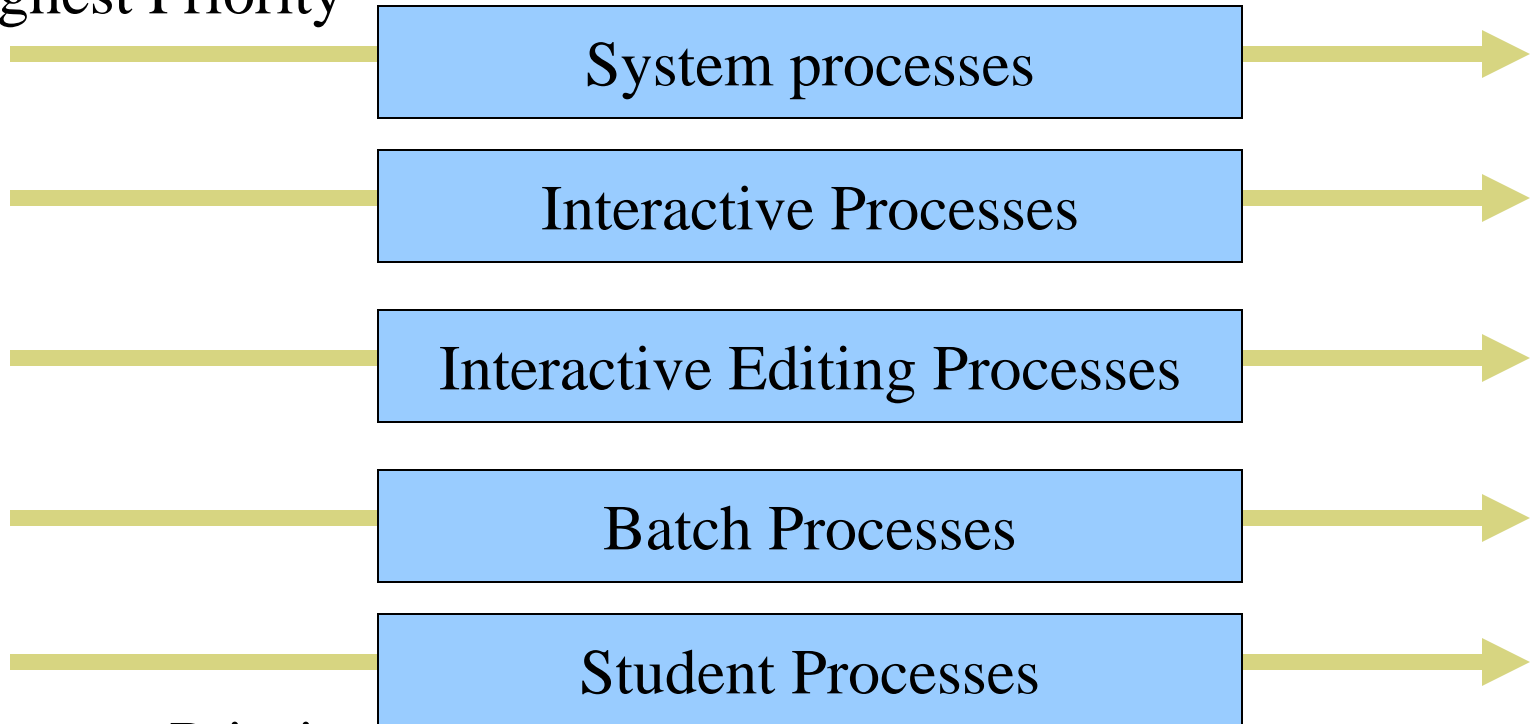
**What happens if time quantum = 1 and 20?**

# Multilevel Queue Scheduling

- Like Round-Robin except have multiple queues.
- Partitions the ready queue into several queues. Processes are permanently assigned to one queue. Each queue has its own scheduling algorithm. Fixed-priority preemptive scheduling between queues.

# Multilevel Queue scheduling

Highest Priority



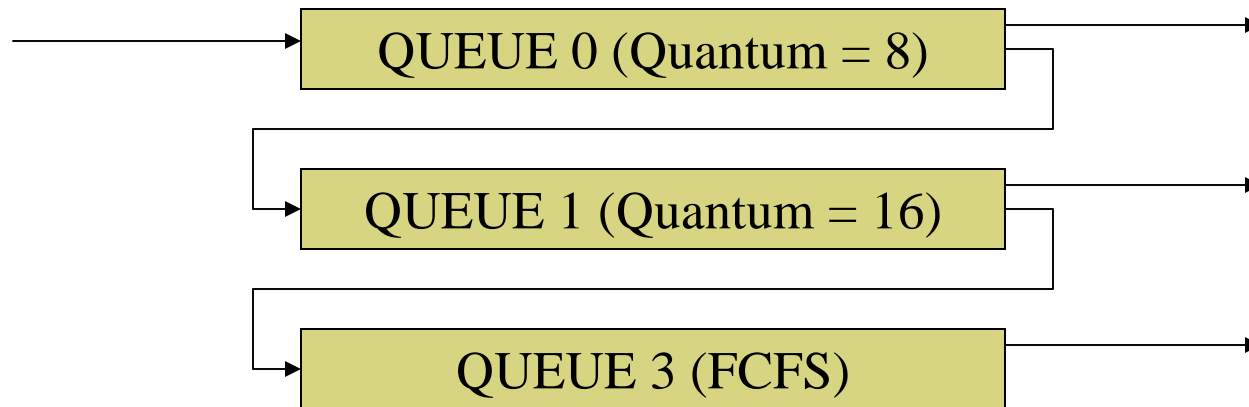
Lowest Priority

# Multilevel Feedback Queue Scheduling

- Similar to multilevel queue scheduling except processes can move between queues as their priority changes.
- Separate processes with different processor burst characteristics. A process with too much processor time will be moved into a lower-priority queue.

# Multilevel Feedback Queue Scheduling

## – Example



- A process entering the ready queue is put in queue 0.
- Only when queue  $i-1$  is empty, will it execute processes in queue  $i$ .
- A process that arrives for queue  $i$  will preempt a process in queue  $i+1$ .
- If a process in queue  $i$  does not finish in a time quantum, it is moved to the tail of queue  $i+1$ .

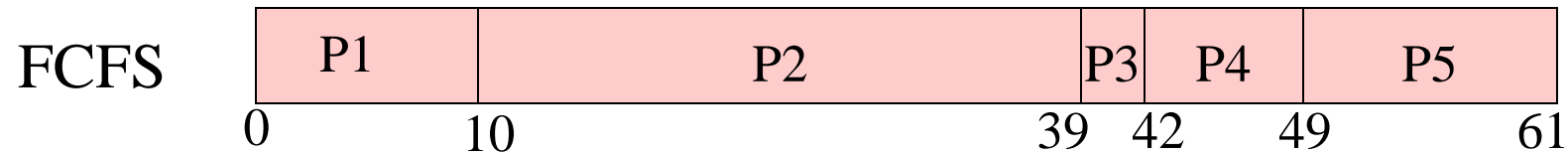
# Algorithm Evaluation – Example

- Five processes arrive at time 0, in the order given, with the length of the CPU burst time given in milliseconds:

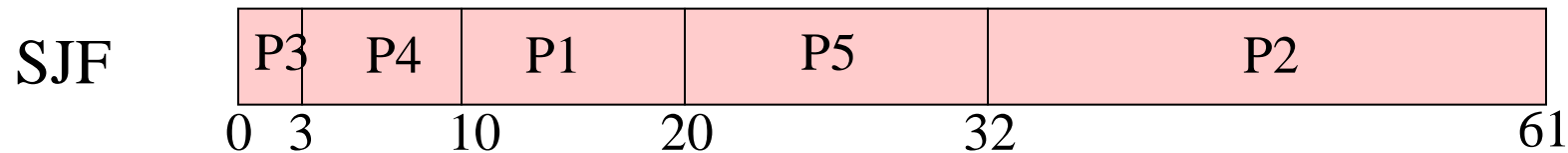
| Process | Burst Time |
|---------|------------|
| P1      | 10         |
| P2      | 29         |
| P3      | 3          |
| P4      | 7          |
| P5      | 12         |

Consider the FCFS, SJF and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes. Which algorithm would give the minimum average waiting time?

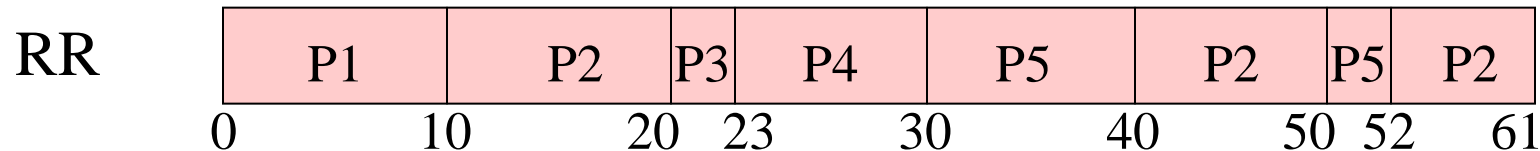
# Algorithm Evaluation – Example



$$\text{Average waiting time} = (0+10+39+42+49)/5=28$$



$$\text{Average waiting time} = (10+32+0+3+20)/5=13$$



$$\text{Average waiting time} = (0+32+20+23+40)/5=23$$

# Summary

- What is CPU scheduling?
- Scheduling criteria: CPU utilization, waiting time, turnaround time...
- Scheduling algorithms: FCFS, SJF, RR, Multilevel queue ...



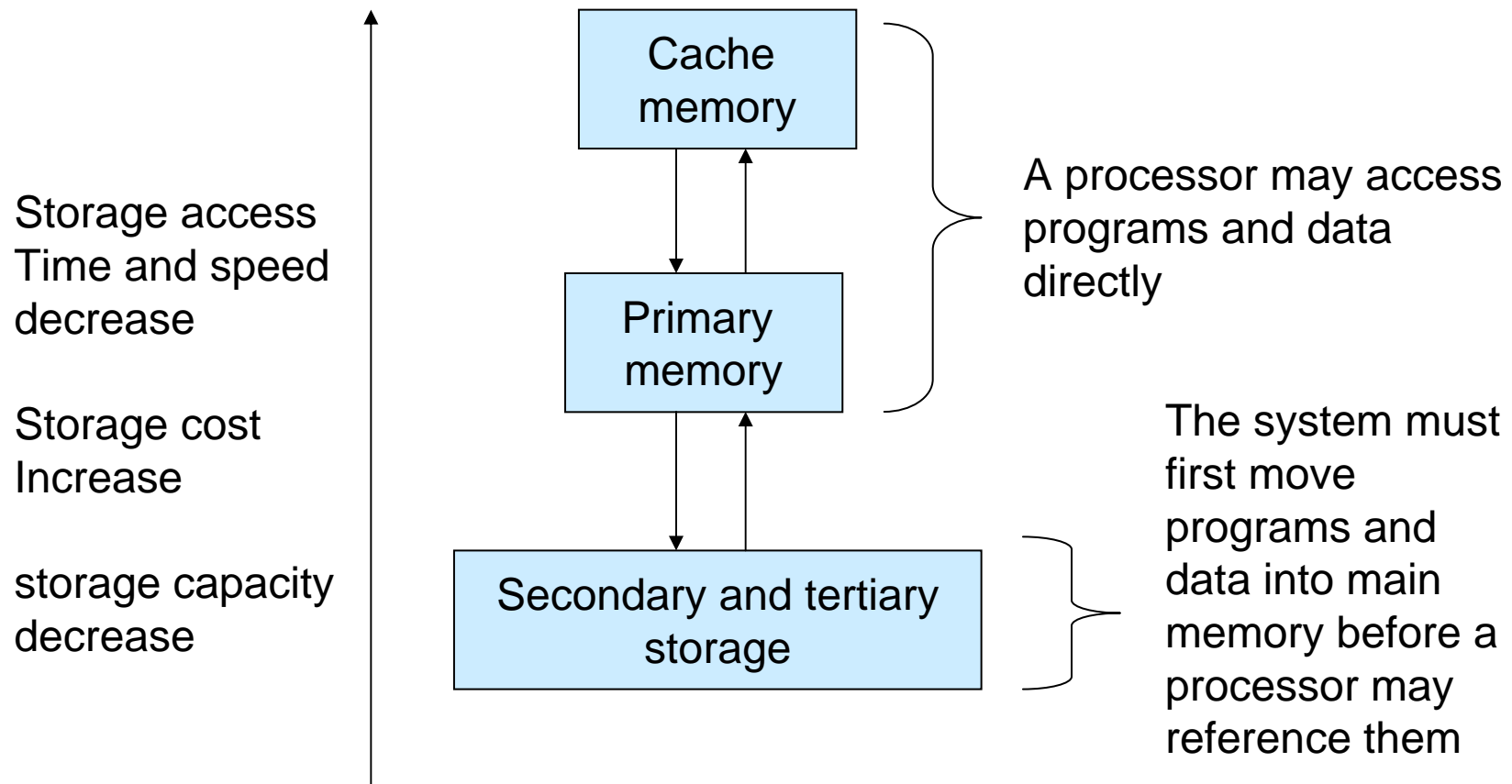
# Chapter 9

## Memory Management

# Background

- Memory is a large array of bytes, each with its own address.
- The program must be loaded into memory to be executed.
- CPU fetches instructions from memory according to the program counter. The instruction may cause loading from and storing to memory.

# Memory Hierarchy



# Memory Management Strategies

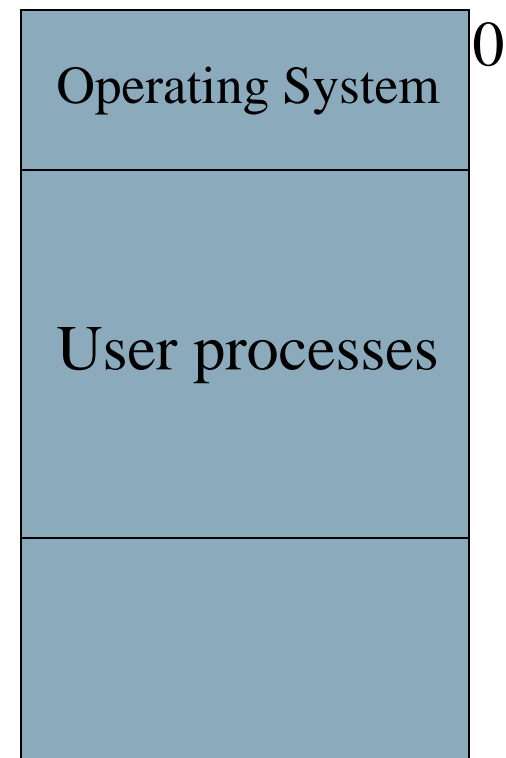
- Fetch strategies – determines when to move the next piece of a program or data to main memory from secondary storage
  - Demand fetch strategies
  - Anticipatory fetch strategies
- Placement strategies – determines where in main memory should place incoming program or data pieces.
  - First-fit, best-fit, worst-fit
- Replacement strategies – determines which piece to remove from the main memory

# Contiguous vs. Non-contiguous Memory Allocation

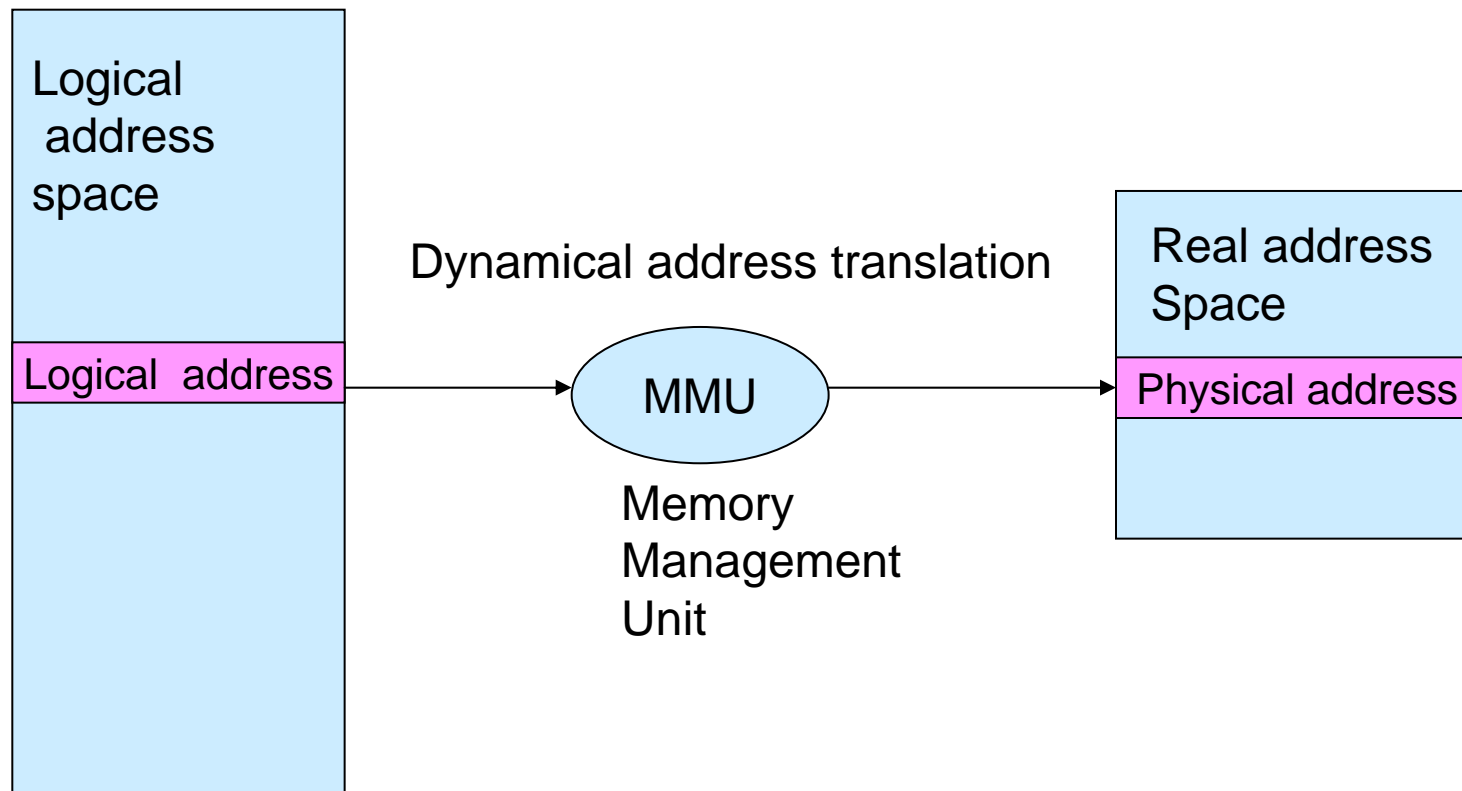
- Contiguous memory allocation.
  - To run a program, the system has to find enough contiguous memory to accommodate the entire program.
- Non-contiguous memory allocation.
  - A program is divided into blocks or segments that the system may place in nonadjacent slots in main memory.

# Memory Allocation

- The memory is usually divided into two partitions, one for the resident operating system, and one for the user processes. It is common for the operating system to occupy low memory.

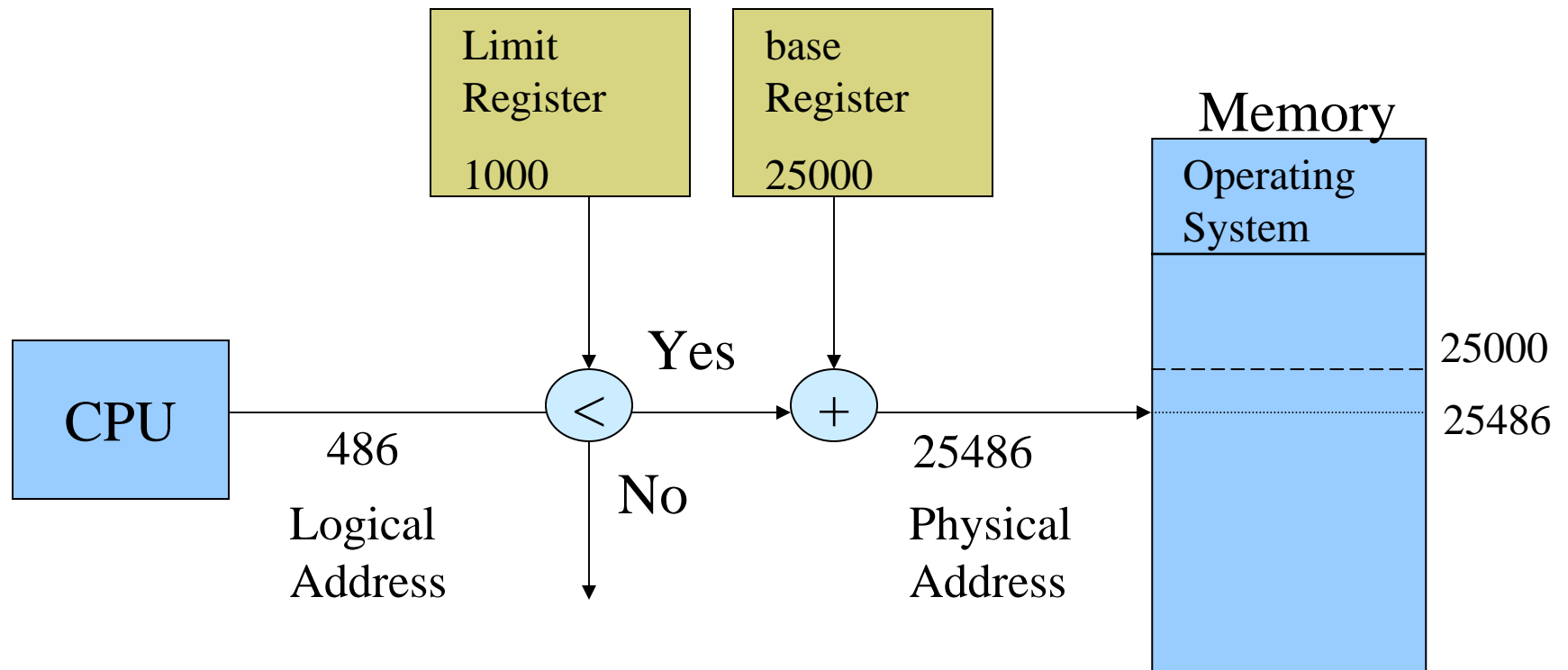


# Logical and Physical Addresses



# Contiguous Single-Partition Allocation

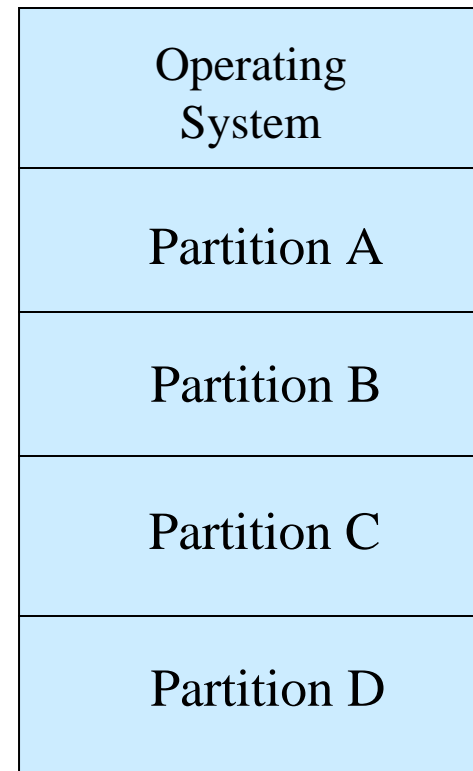
- A **base register** contains the memory base value of a process.
- A **limit register** contains the range of the logical addresses.





# Contiguous Fixed-Partition Allocation:

- Divide the memory into fixed-length partitions.
- The degree of multiprogramming is constrained.
- The size of each process is bounded.
- Suffers internal fragmentation



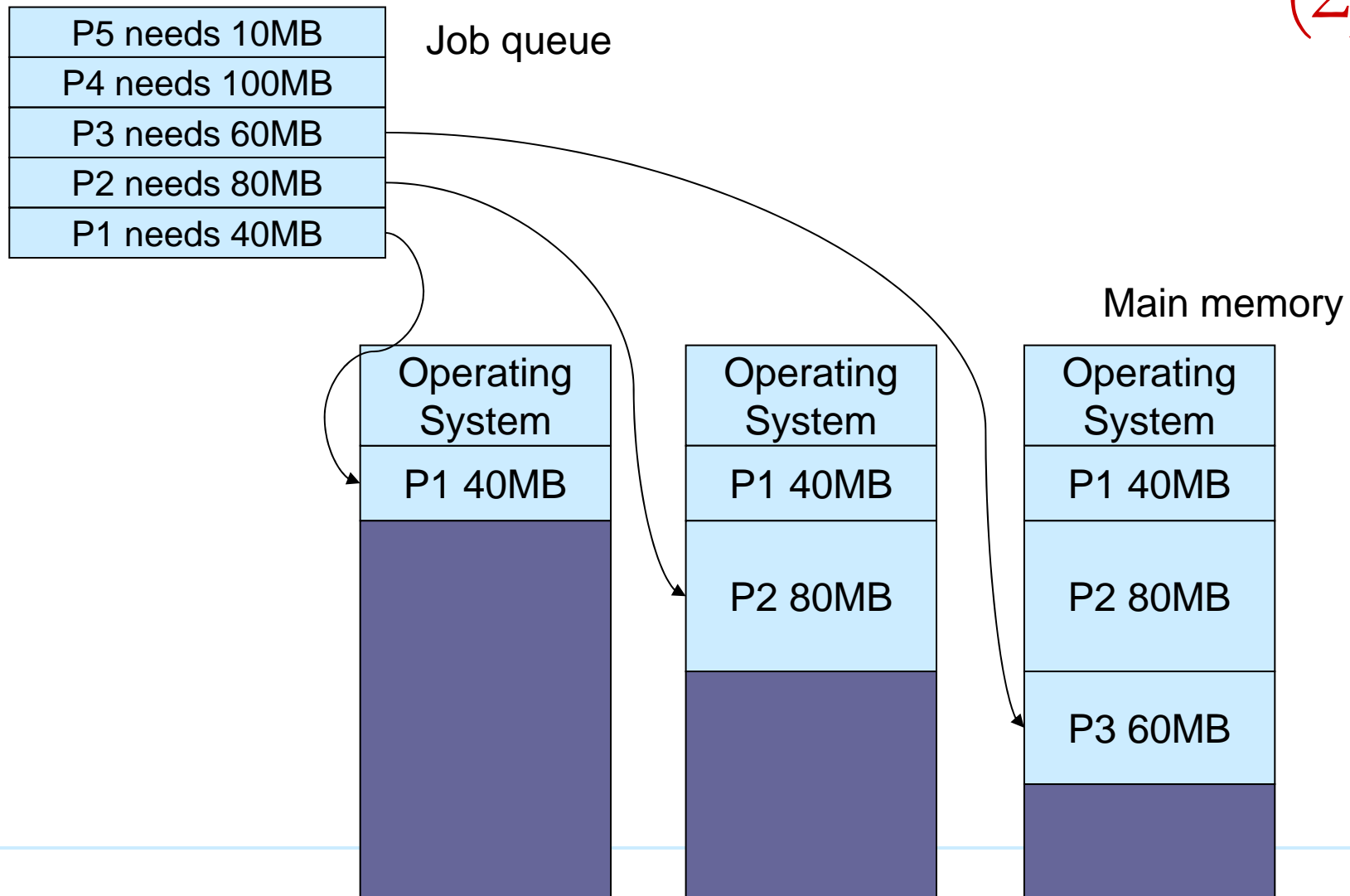
# Fragmentation

- Fragmentation – the system cannot use certain areas of available memory.
  - External fragmentation – the available memory space is broken into chunks, and the largest contiguous chunk is insufficient for a request.
  - Internal fragment – memory that is internal to a partition, but cannot be used.

# Contiguous Variable-Partition Allocation (1)

- Initially, all the memory is considered as one large block of available memory, a hole.
- When a process needs memory, a hole large enough for the process is allocated for it.
- A free memory list is used to track available memory.

# Contiguous Variable-Partition Allocation (2)

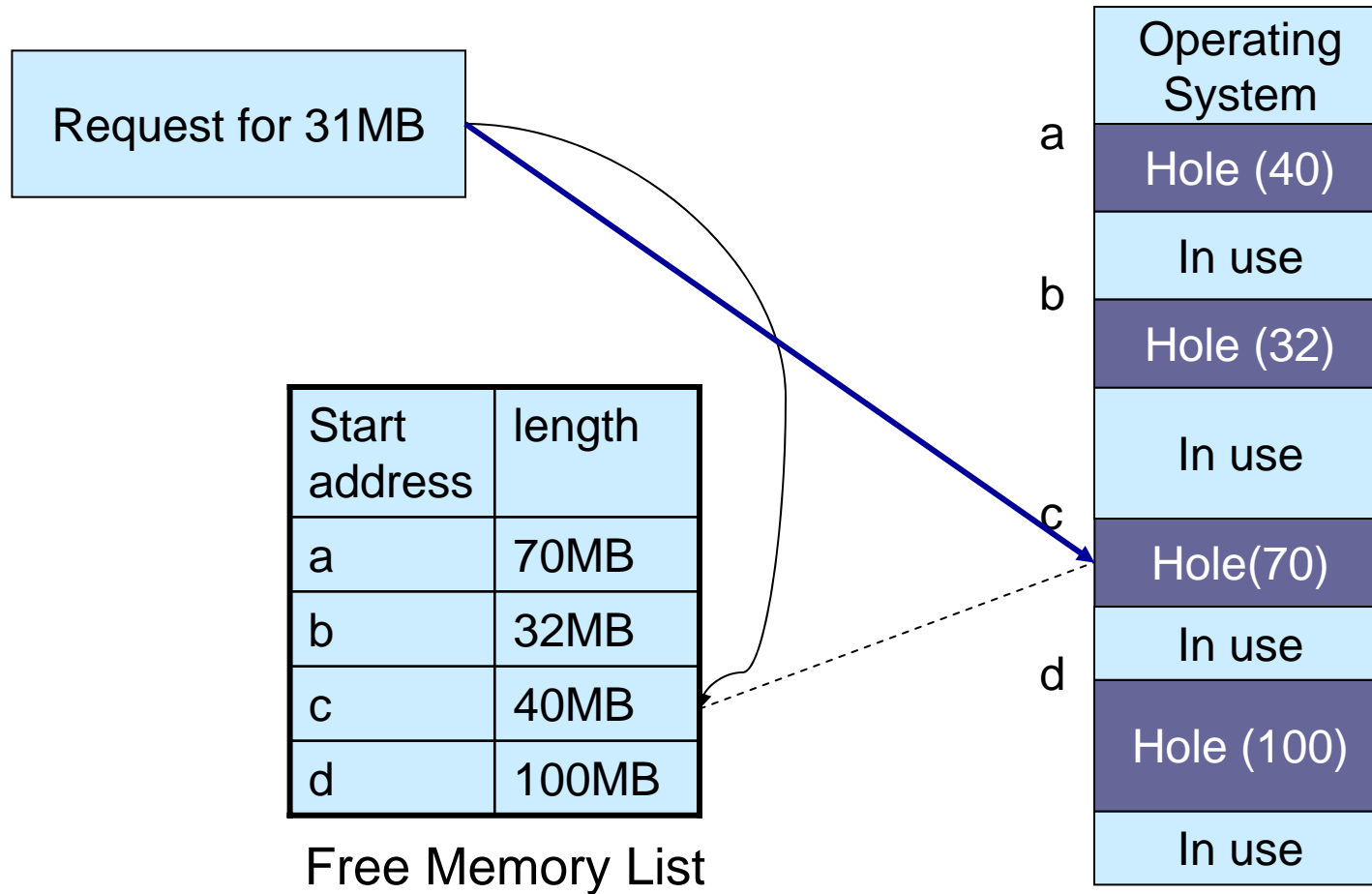


# Contiguous Variable-Partition Allocation

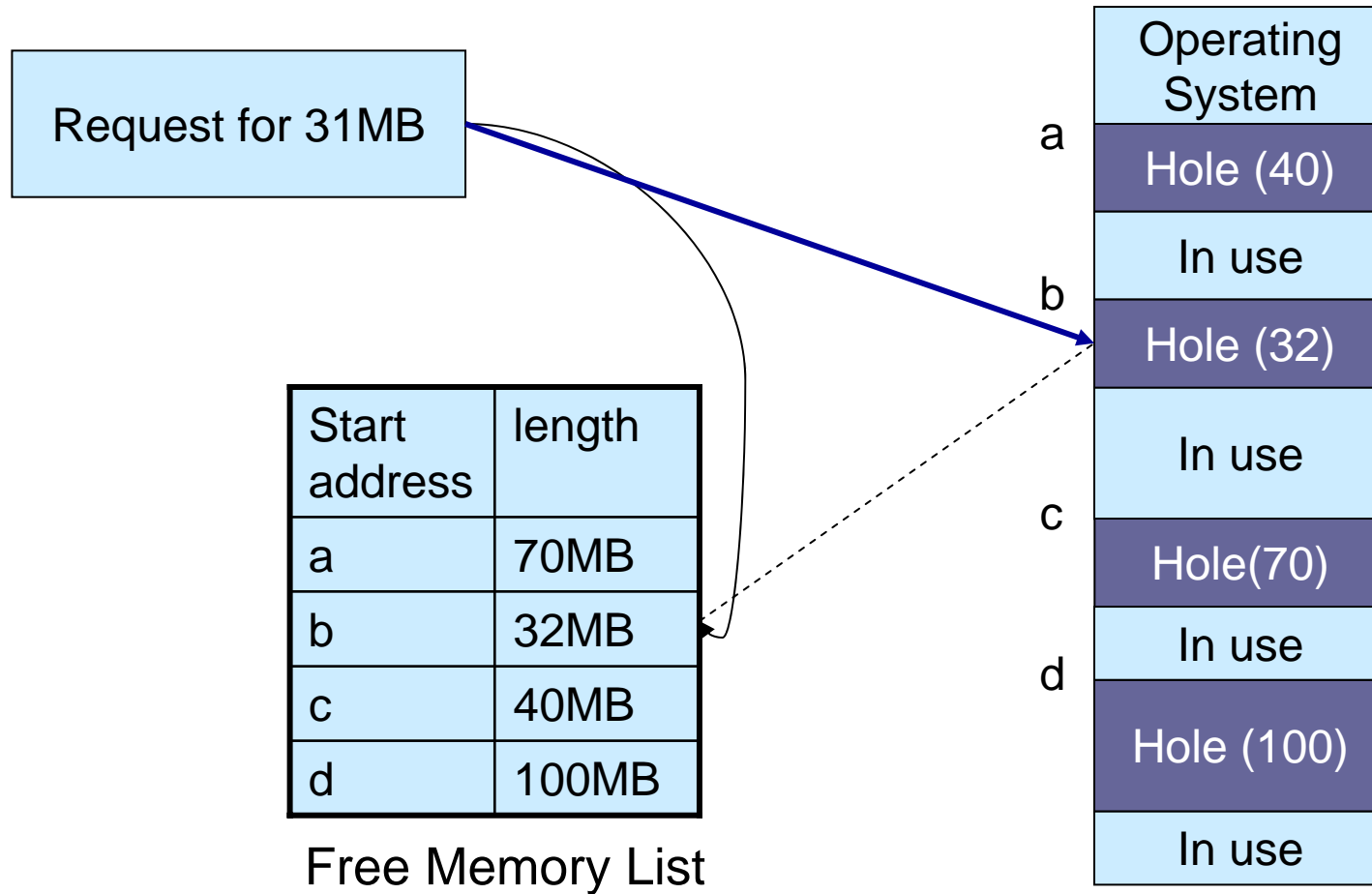
## Memory Placement Strategies

- First-fit: allocate the first hole large enough.
- Best-fit: allocate hole with the smallest leftover.
- Worst-fit: allocate the largest hole.

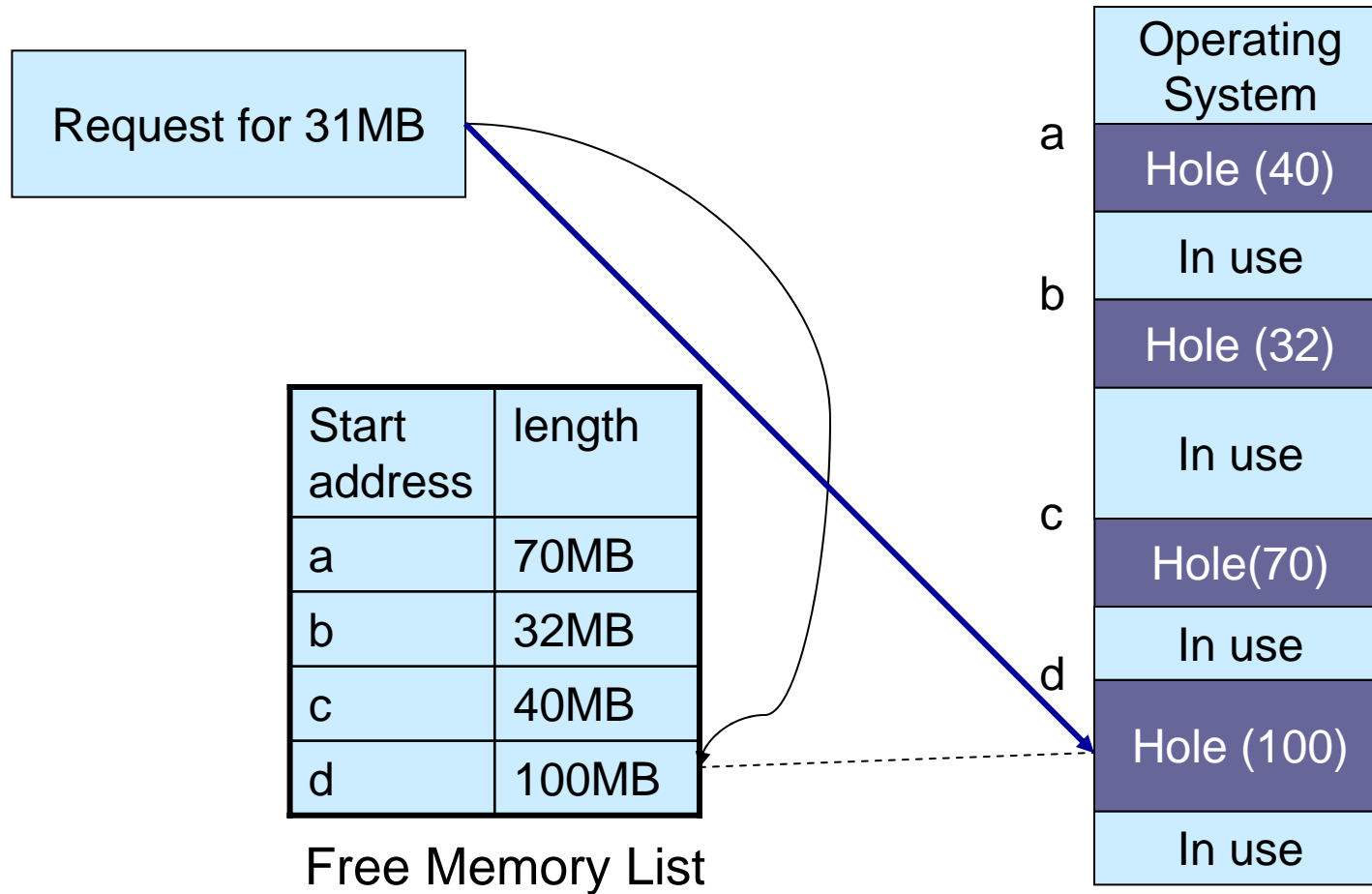
# First-Fit Strategy



# Best-Fit Strategy



# Worst-Fit Strategy





# An Example

(1)

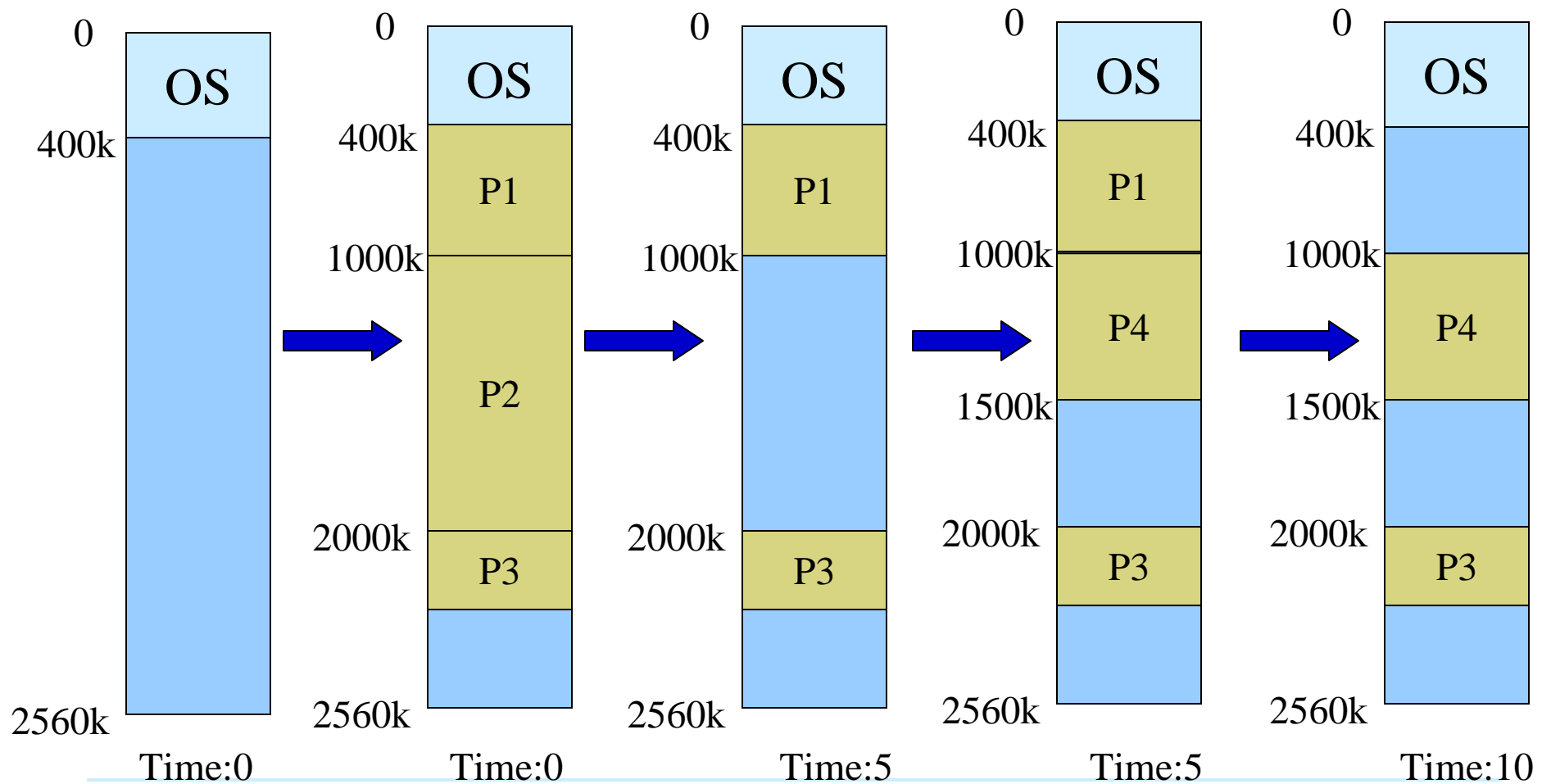
## Question:

Assume that we have 2560k of memory available and a resident OS of 400k. Given the input queue in the table, using FCFS, how would the First-Fit algorithm place the processes in the input queue (The newly freed memory hole is appended to the end of the free memory list)?

| Process | Memory | Burst Time |
|---------|--------|------------|
| P1      | 600k   | 10         |
| P2      | 1000k  | 5          |
| P3      | 300k   | 40         |
| P4      | 500k   | 35         |

# An Example

(2)



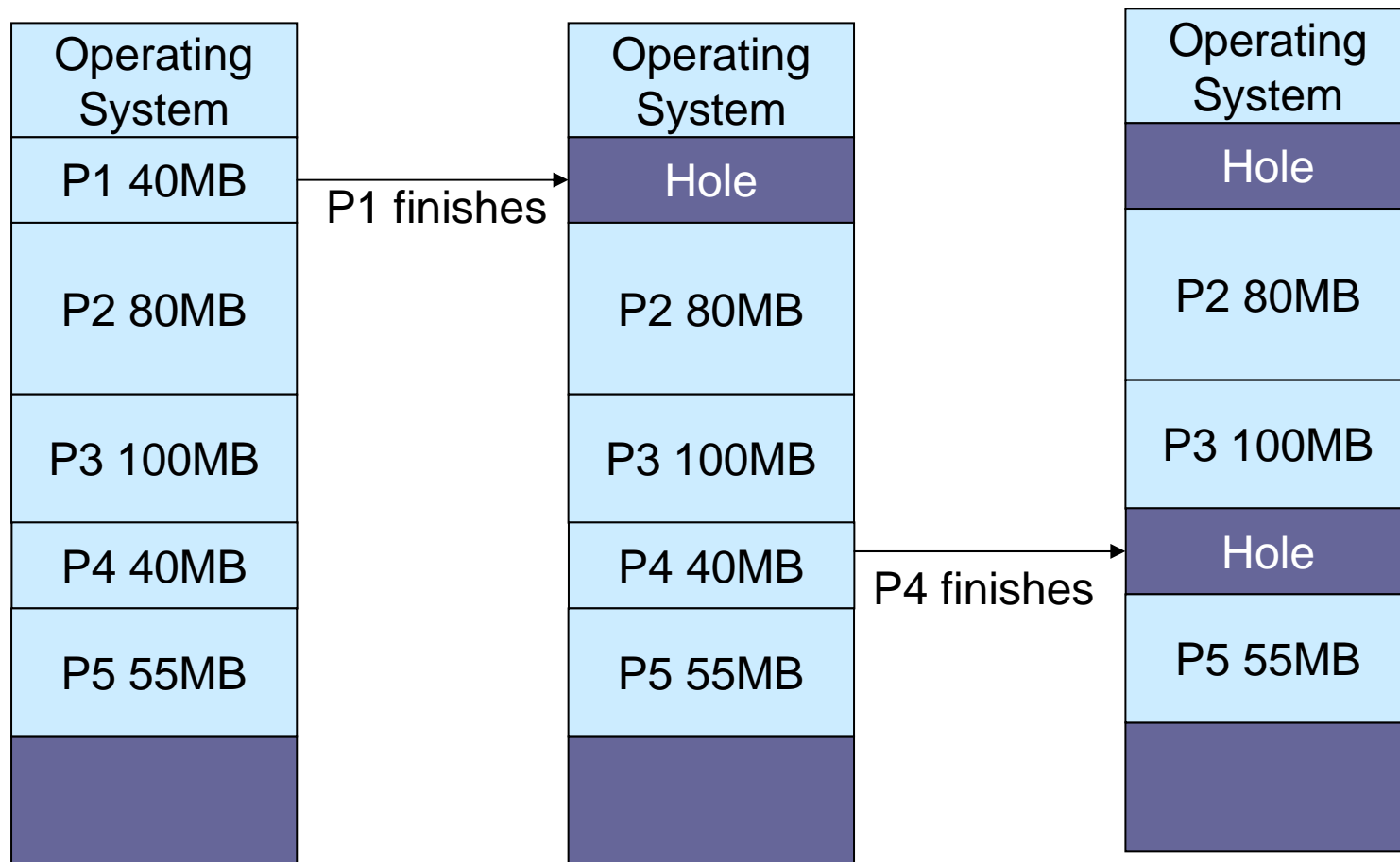
# Contiguous Variable-Partition Allocation

## External Fragmentation

- As processes are loaded and removed from memory, the free memory space is broken into little pieces, creating an external fragmentation problem.

# Contiguous Variable-Partition Allocation

## External Fragmentation – Example



# Contiguous Variable-Partition Allocation

## Reducing External Fragmentation

- Coalescing – merge adjacent holes to form a single, larger hole.
- Memory compaction (also referred to as garbage collection) – relocate all occupied areas of memory to one end of main memory. This leaves a single large free memory hole.
- Non-contiguous memory allocation:
  - Paging
  - Segmentation

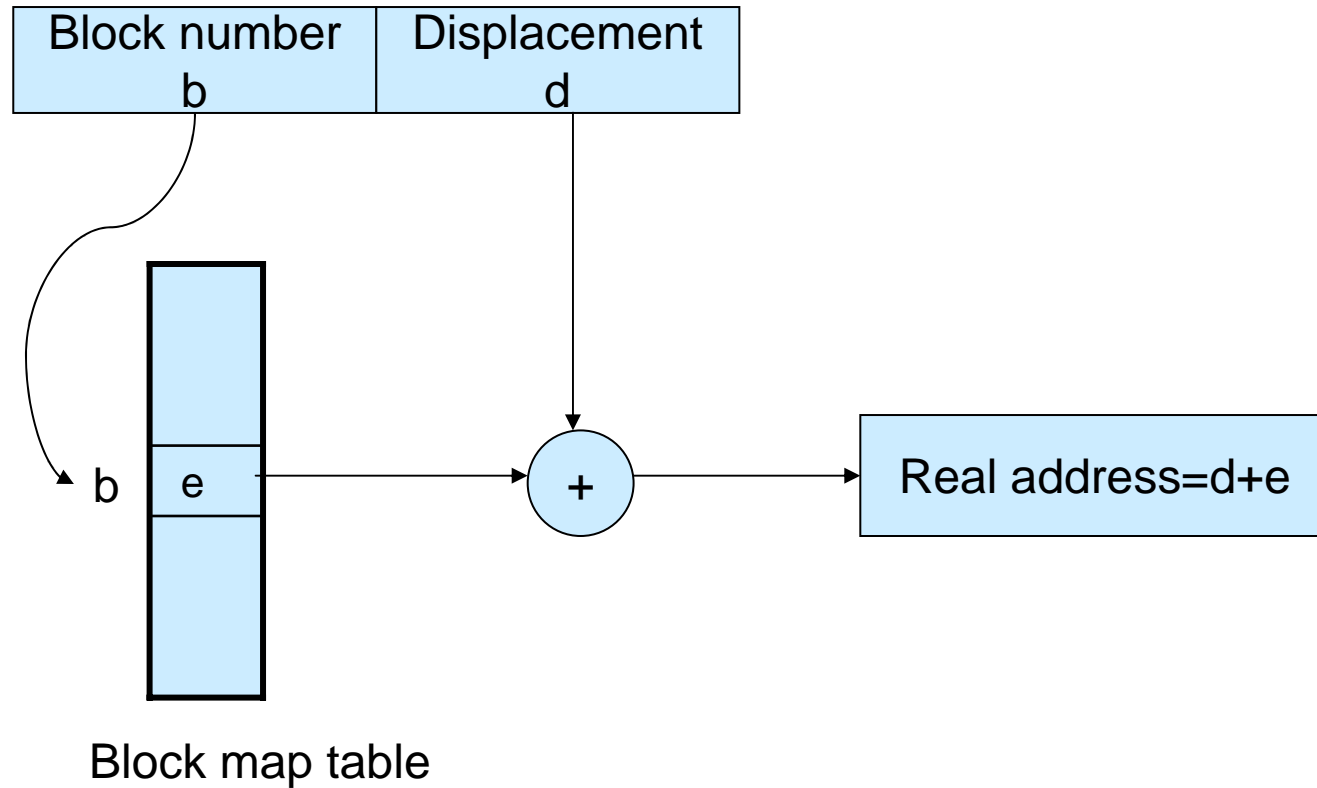
# Block Mapping

- Group information (program and data of processes) into blocks – the system only needs to keep track of where in main memory each virtual memory block has been placed.
  - Pages – blocks that are a fixed size (paging)
  - Segments – blocks that are of different size. (segmentation)
- An address is represented as a ordered pair

|                   |                   |
|-------------------|-------------------|
| Block number<br>b | Displacement<br>d |
|-------------------|-------------------|

Virtual address: (b, d)

# Logical Address Translation



# Paging

- Break physical memory into fixed-sized blocks called **frames**.
- Break logical memory into fixed-sized blocks called **pages**.
- Logical address is an ordered pair  $(p, d)$ , where  $p$  is the page number in logical memory and  $d$  is the page offset.
- A **page table** is indexed by the page number, and each item contains the physical frame number that contains the page.



# Representation of Memory Addresses

- Consider a system using  $n$  bits to represent both physical and logical addresses.
  - The most-significant  $n-m$  bits is used for the page or page frame number
  - The remaining  $m$  bits are used for the displacement.

Consider a 32-bit system using 4k pages. ( $n=32$ ,  $m=12$ ).

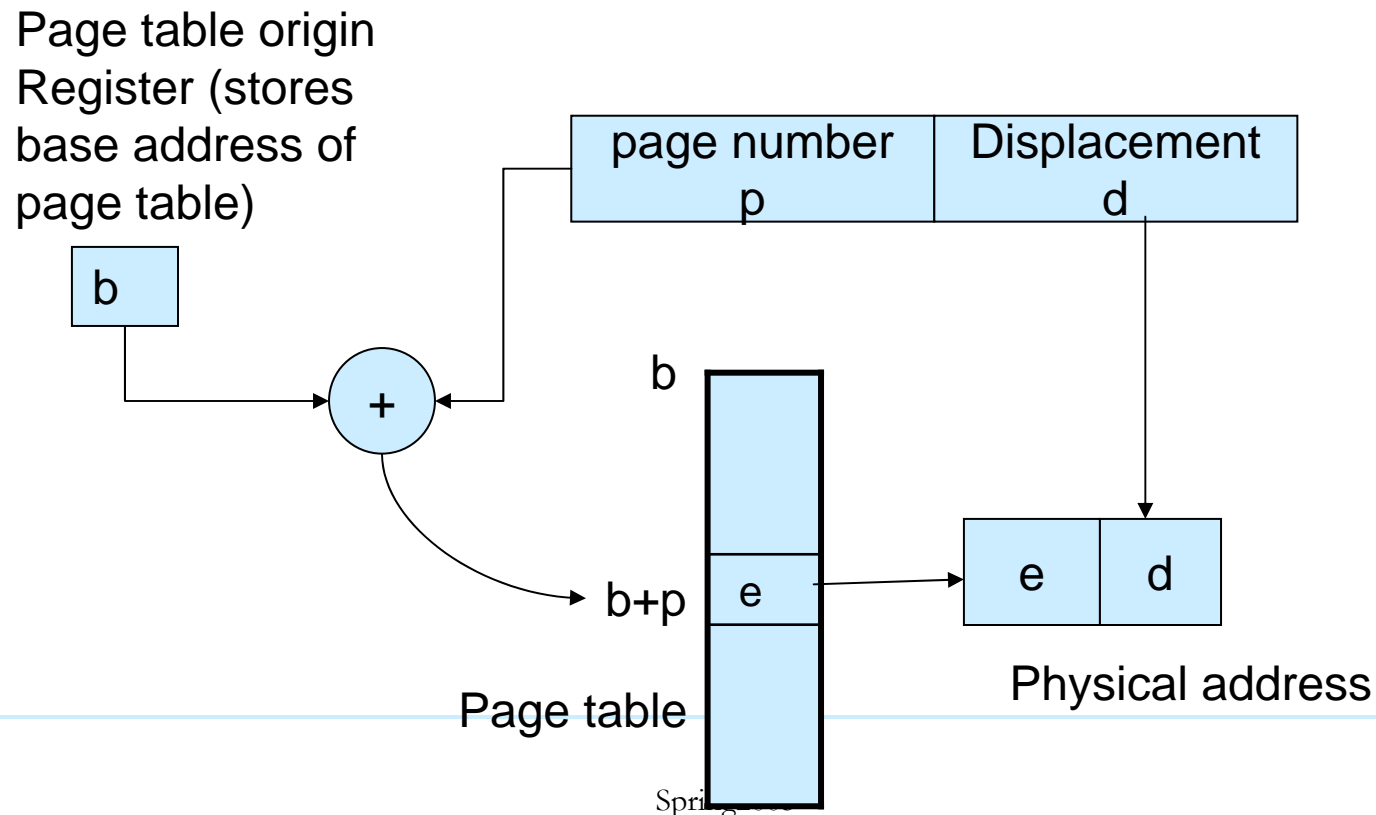
Each page or frame number is represented using 20 bits.

A logical address (15, 200) is represented as the binary string

000000000000000001111000011001000

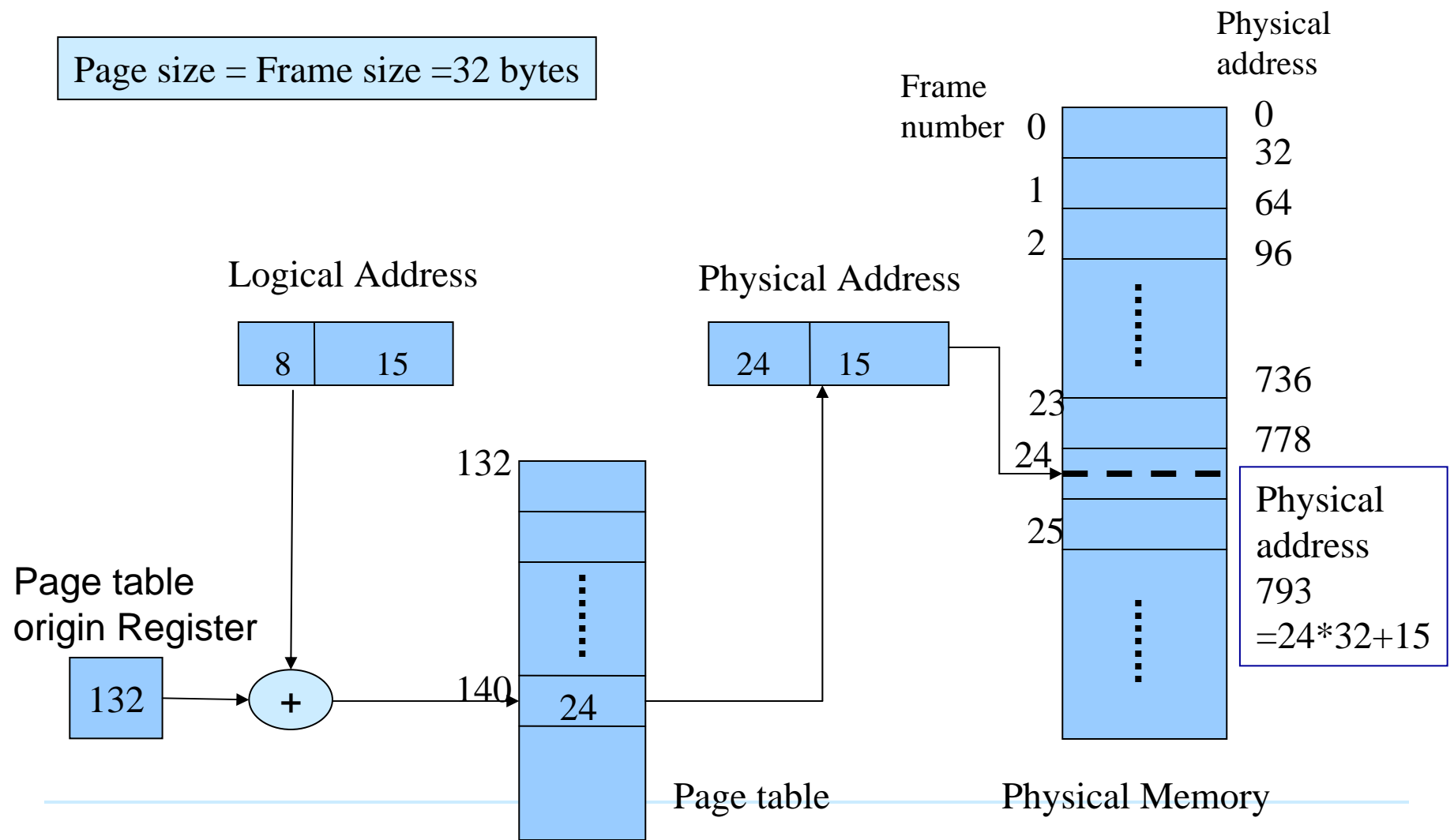
# Paging Address Translation by Direct Mapping (1)

The system can locate directly any entry in the page table with a single access to the table



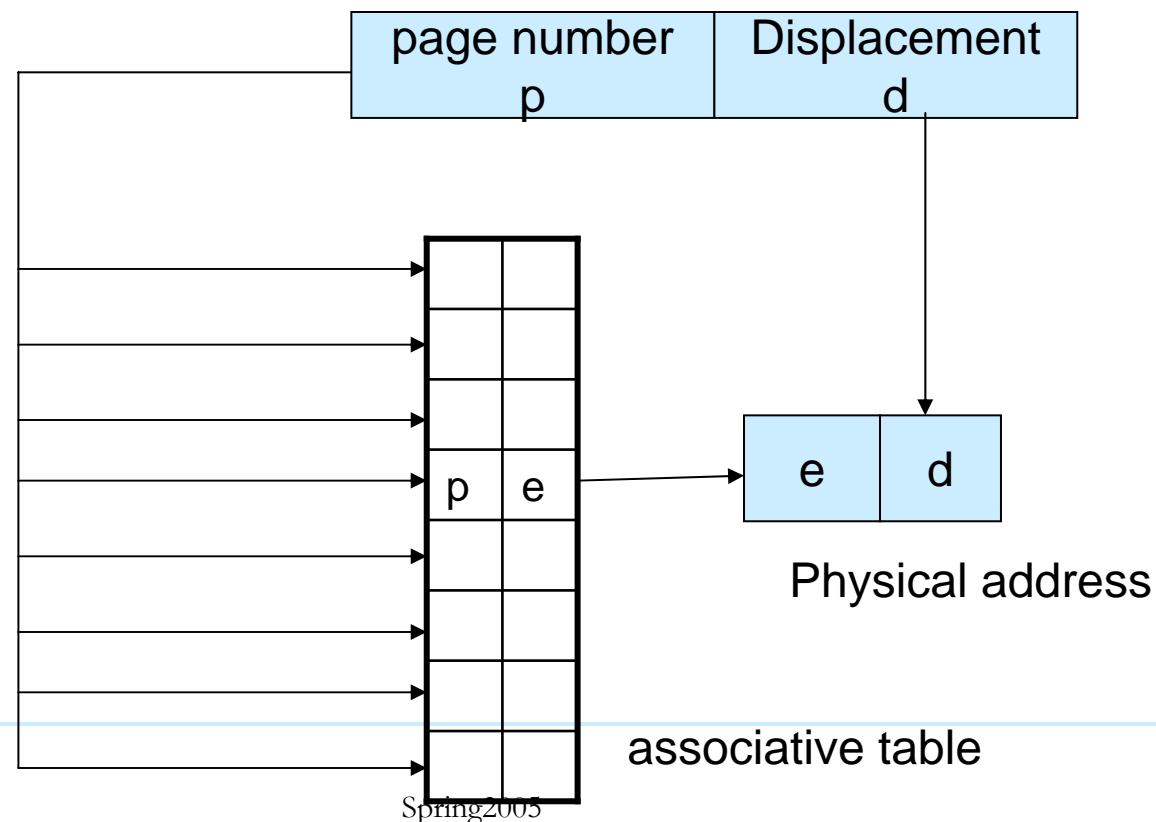
# Paging Address Translation by Direct Mapping (2)

Page size = Frame size = 32 bytes



# Paging Address Translation by Associative Mapping (1)

The page table is placed into a content-addressed Associative memory. Every entry in the associative memory is searched simultaneously for a page.



## Paging Address Translation by Associative Mapping (2)

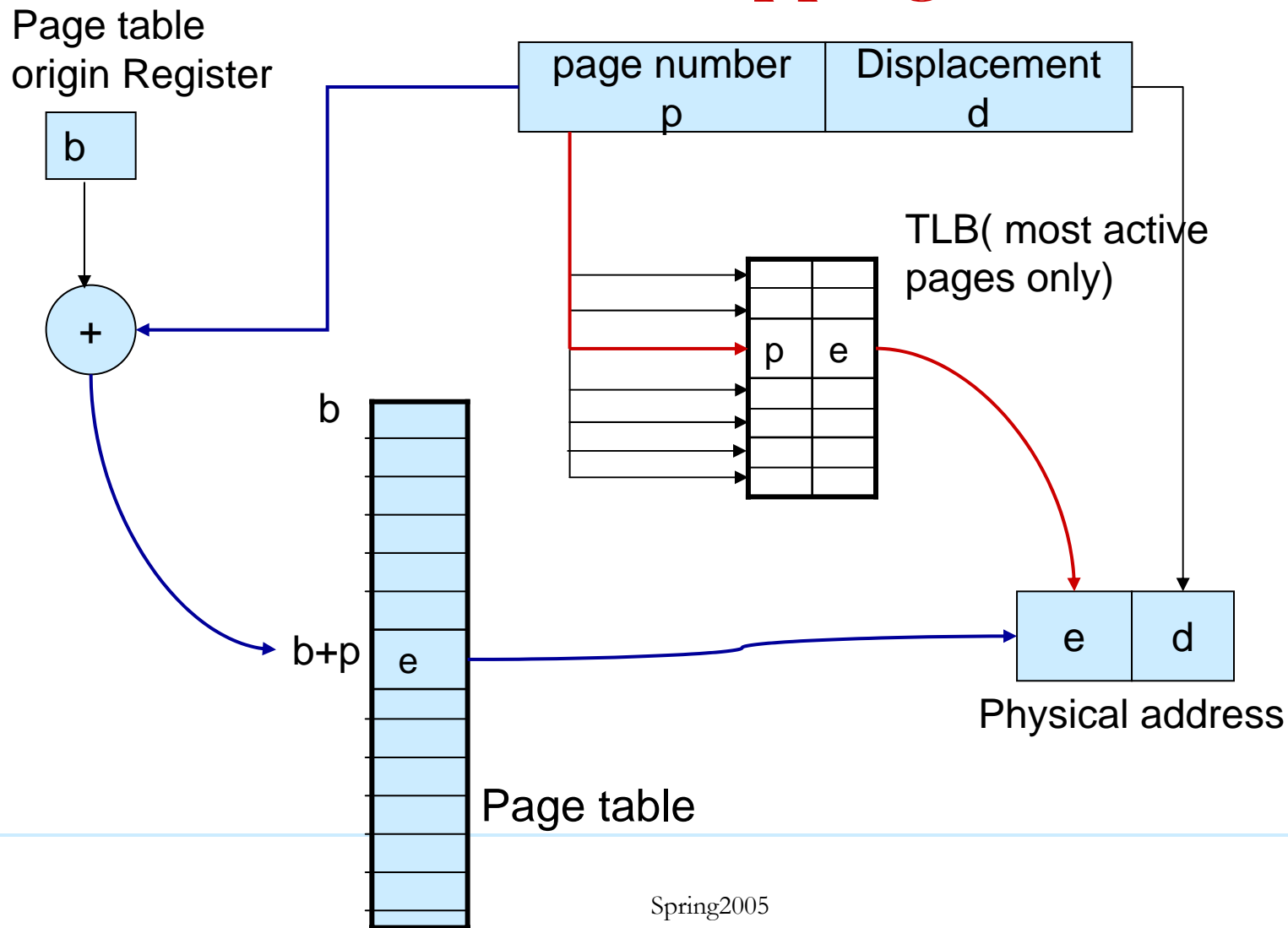
- Associative memory has a cycle time much greater than main memory.
- Pure associative is fast but not often used.
  - High speed cache and associate memory are far too expensive to hold the complete page mapping table.

## Paging Address Translation by Direct/Associative Mapping (1)

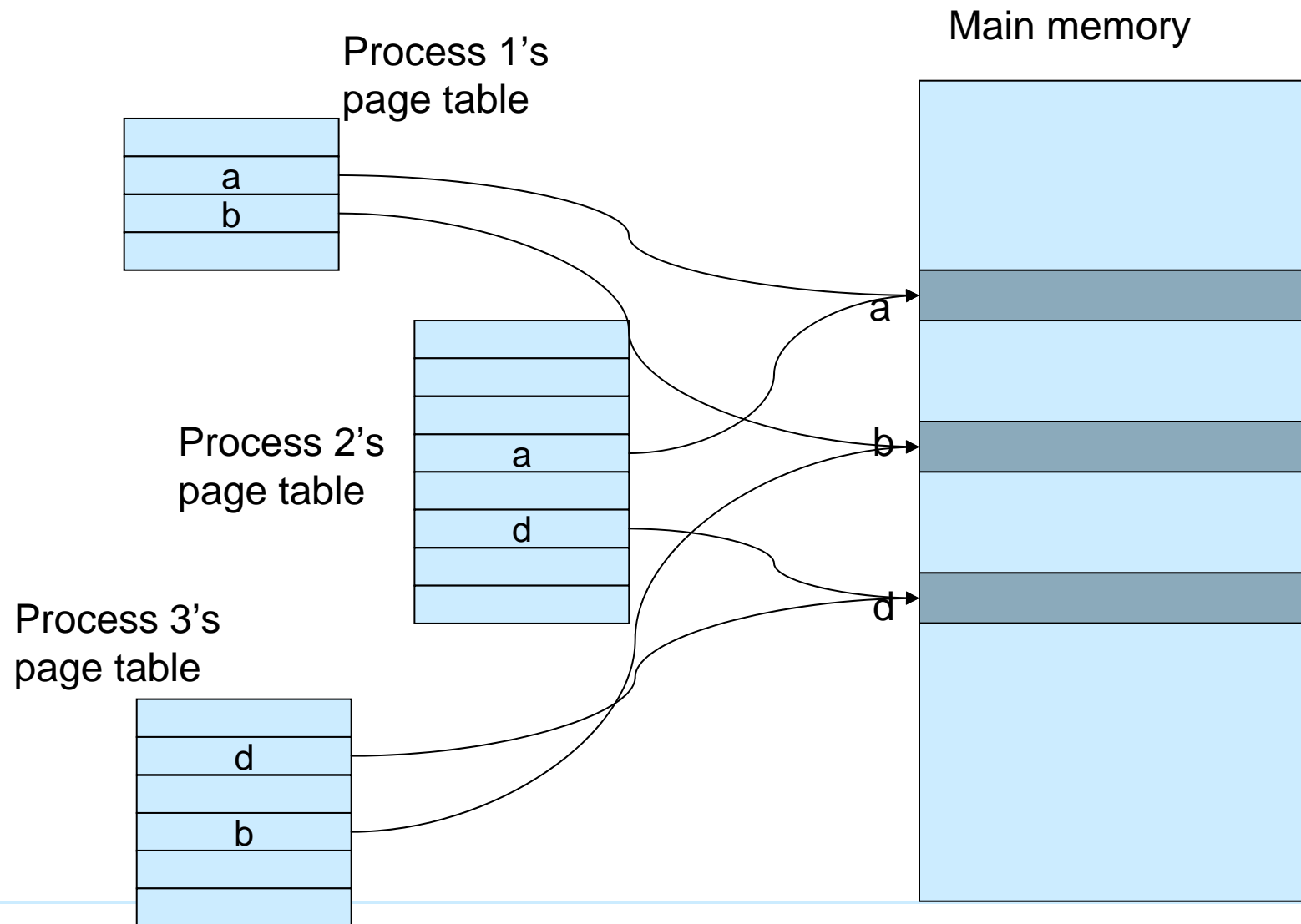
- An associate memory, called translation lookaside buffer (TLB) is used to hold a small part of page table (most active pages only).
- When a process references a logical address, the system first tries to find the page entry in TLB. If the page entry cannot be found, the system then locates the page entry in the conventional page table.

# Paging Address Translation by Direct/Associative Mapping

(2)



# Sharing in a Paging System





# Segmentation

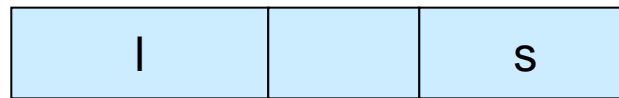
- Paging separates user's view of memory from the actual physical memory.
- The user views memory as a collection of variable-sized segments, with no necessary ordering among segments.
- Segmentation supports this user view of memory.

# Segmentation

- A program's data and instructions are divided into blocks called segments. Each segment consists of contiguous locations. The segments can be of different size.
- Logical address is an ordered pair  $(s, d)$ , where  $s$  is the segment number in logical memory and  $d$  is the segment offset.

# Segment Map Table

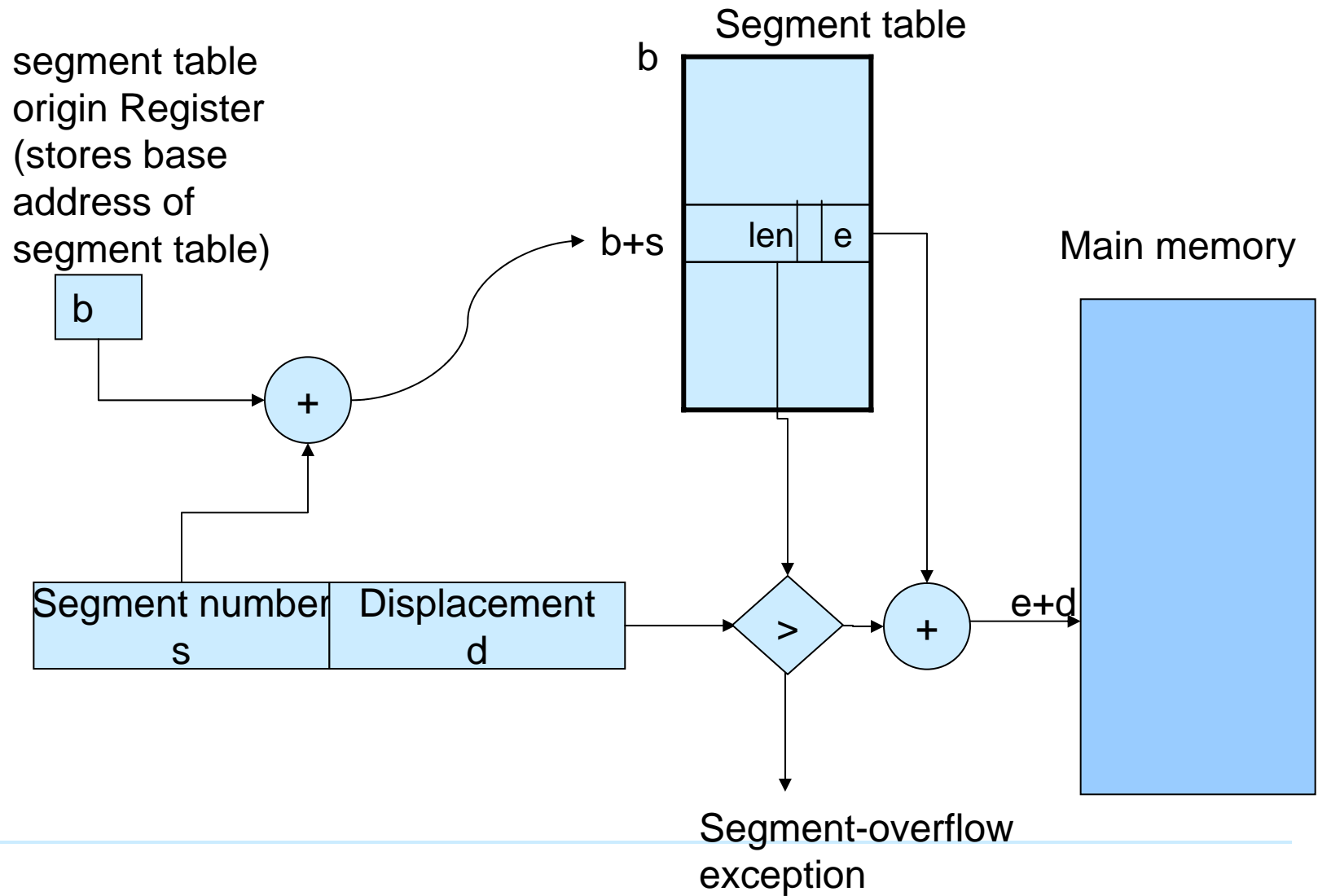
- The segment map table entry



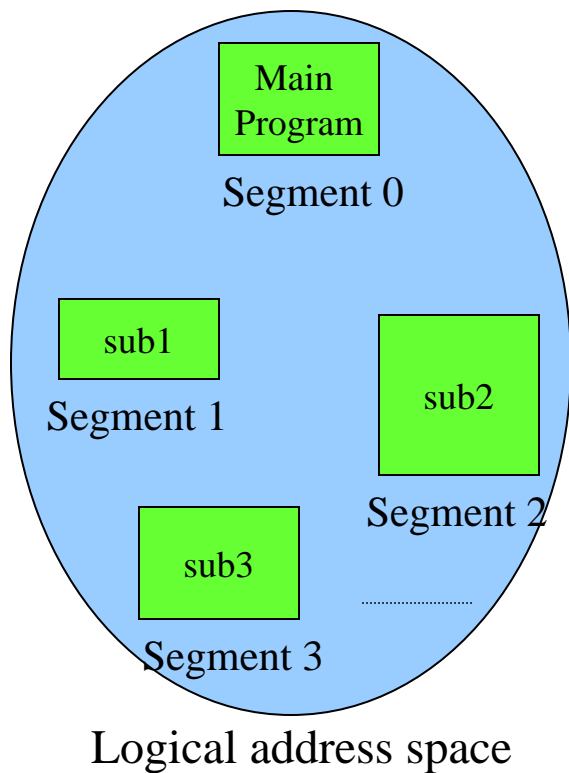
|                   |                    |  |
|-------------------|--------------------|--|
| segment<br>length | Protection<br>bits | Base address<br>of segment in<br>Main memory |
|-------------------|--------------------|--|

- Protection bits indicates whether some operations on the segment are allowed. For example, set a segment read-only.

# Logical address translation

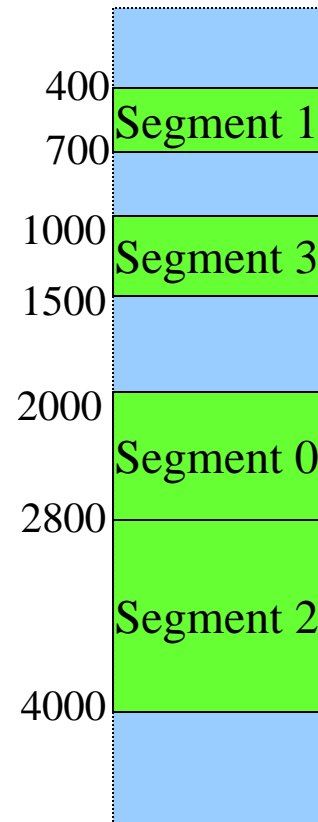


# Segmentation: An Example



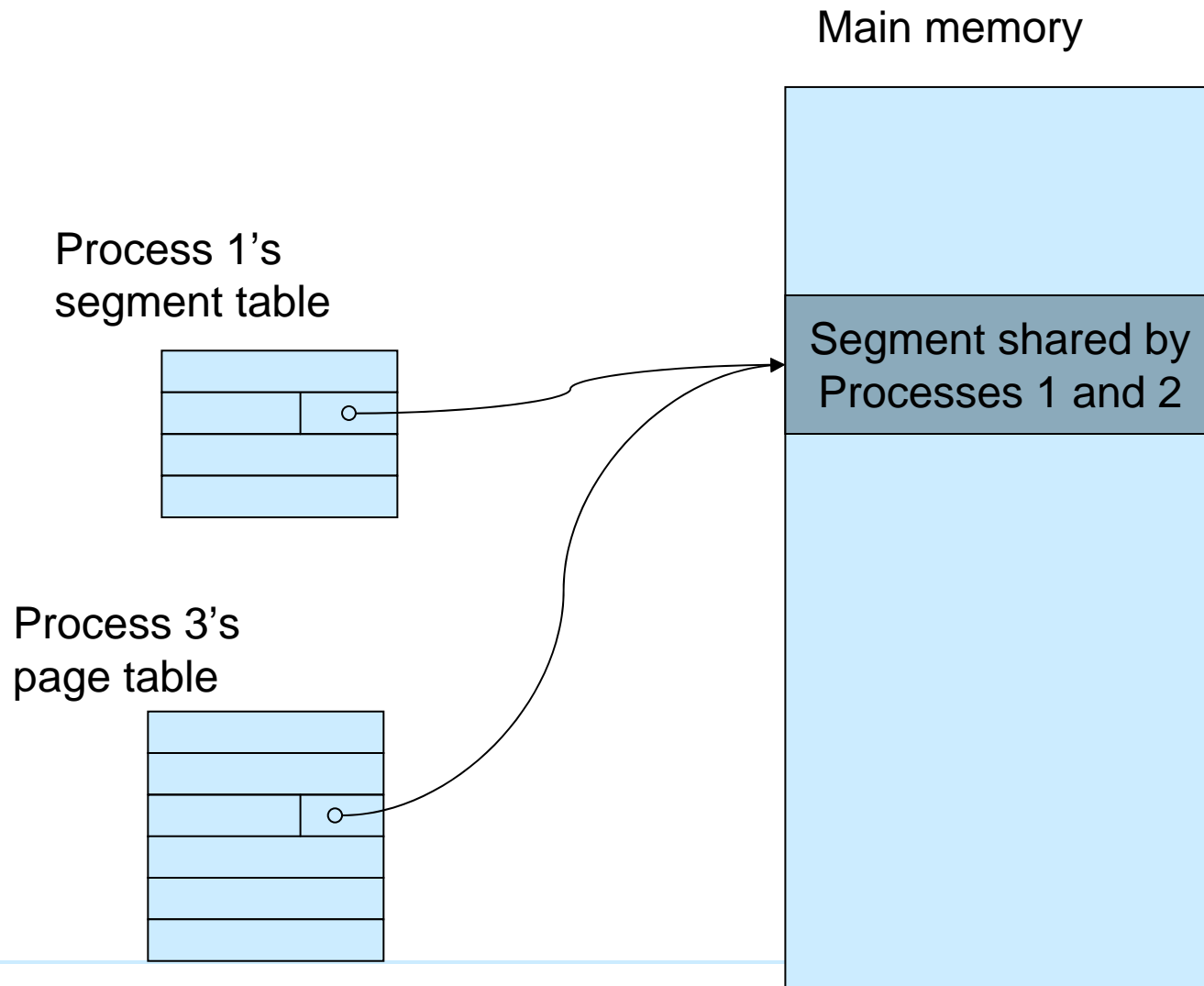
|   | length | base |
|---|--------|------|
| 0 | 800    | 2000 |
| 1 | 300    | 400  |
| 2 | 1200   | 2800 |
| 3 | 500    | 1000 |

Segment table



Physical memory

# Sharing in a Paging System



---

# Fragmentation in Segmentation

- Segmentation can cause external fragmentation.
- Solution:
  - Page the segments – Segmentation with paging

# Segmentation with Paging

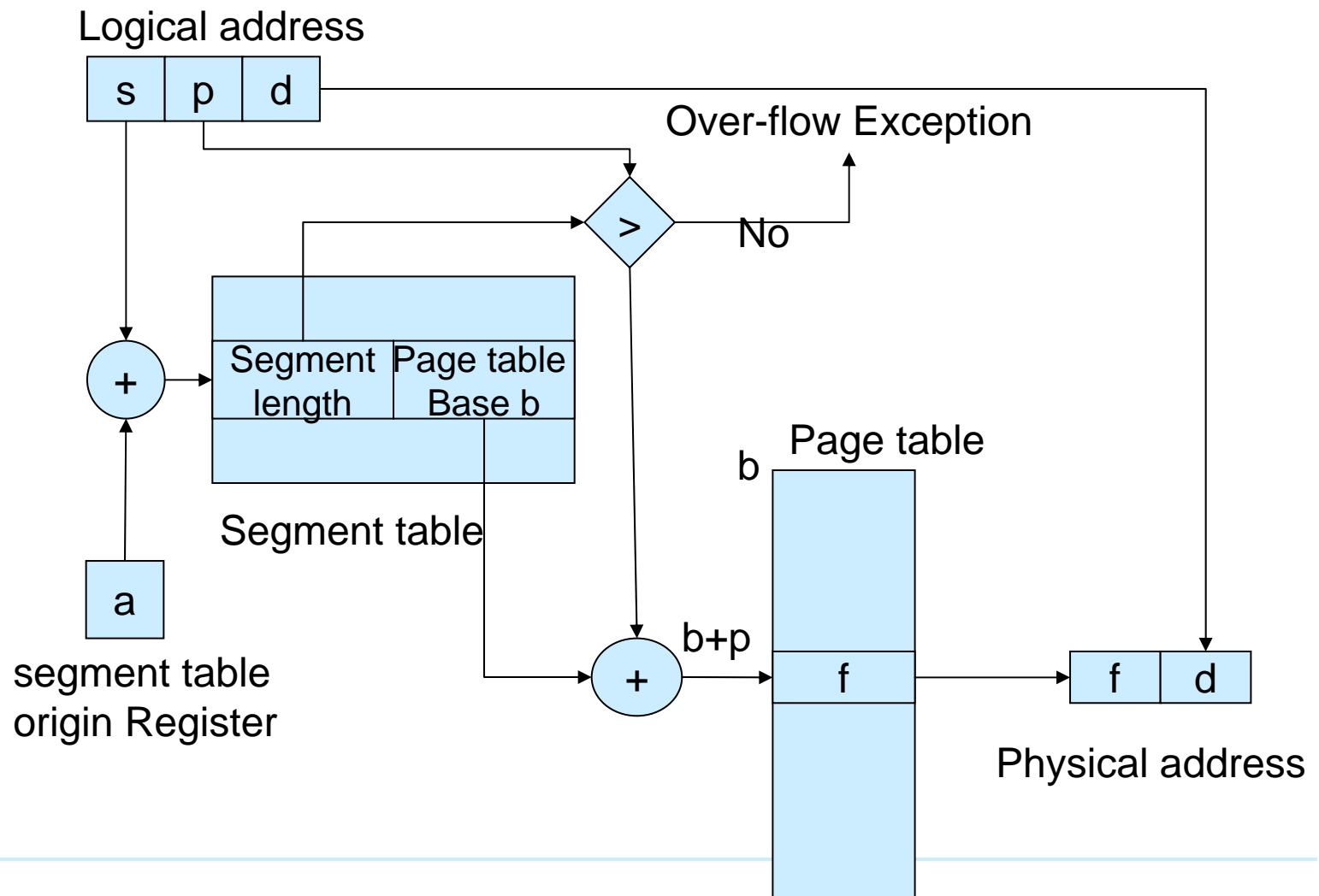
(1)

- Segments are arranged across multiple pages.
- A Logical memory address is presented as a ordered triple  $(s, p, d)$ , where  $s$  is the segment number,  $p$  is the page number, and  $d$  is the displacement within the page



# Segmentation with Paging

(2)



# Summary

- Contiguous vs. non-contiguous memory allocation
- Contiguous memory allocation:
  - Single partition.
  - Multiple partition: Fixed and variable partition.
- Blocking memory systems
  - Paging
  - Segmentation
  - Segmentation with paging

# Chapter 10

## Virtual Memory

# Why Virtual Memory?

- Previous memory allocation strategies require the entire process to be in memory before its execution.
- Benefits of the ability to execute a program that is only partially in memory:
  - A program is no longer constrained by the amount of physical memory that is available.
  - More programs can be run at the same time.
  - Less I/O would be needed.

# Overlays

- Overlays allow the system to execute a program that is larger than main memory.
  - The program is divided into logical sections. When the memory for a section is not needed, the system can replace it with the memory for a needed section.

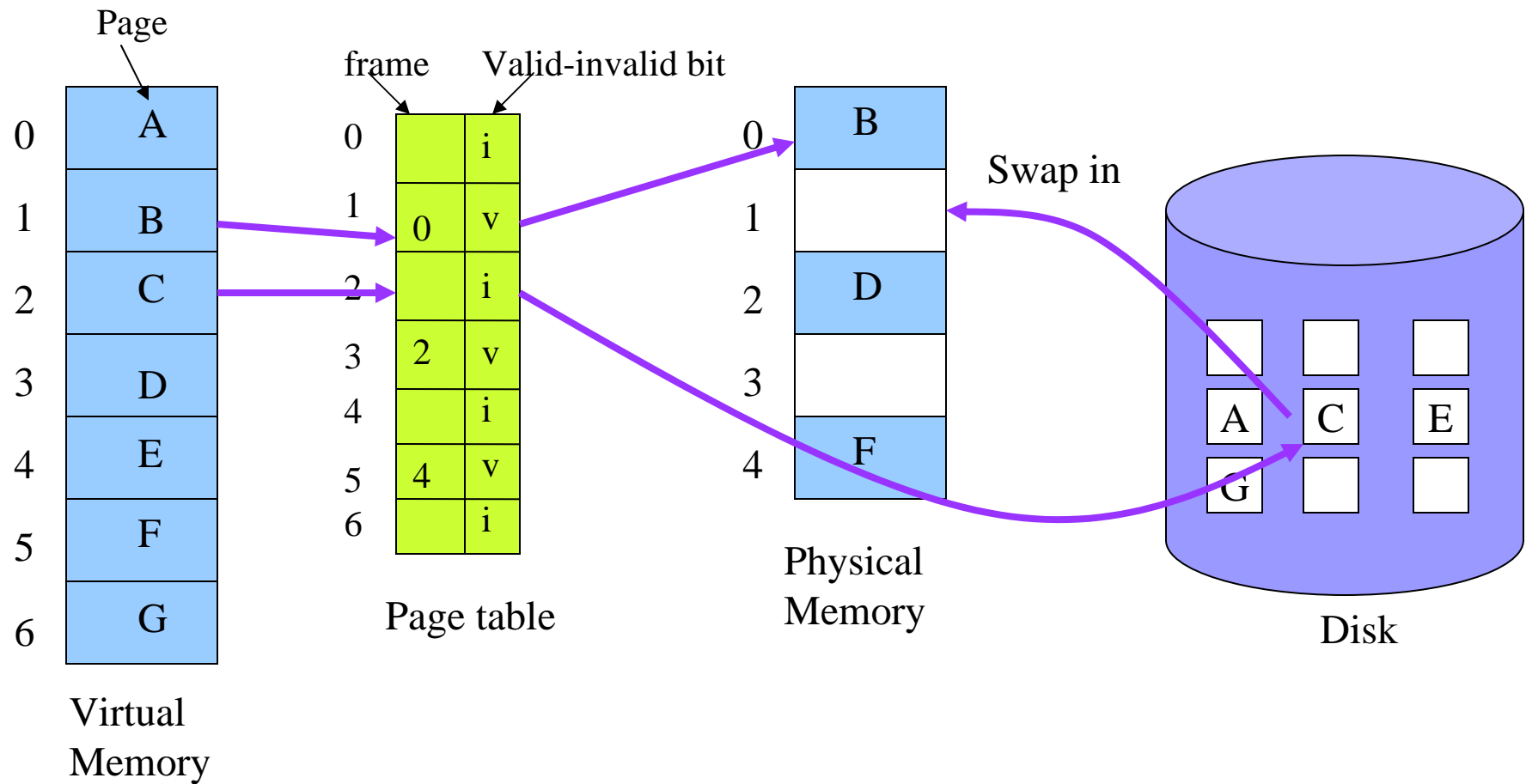
# Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory to continue execution.
- The backing store is commonly a fast disk.
- The context switch time is very high.

# Virtual Memory

- Virtual memory is a technique that allows the execution of processes that may not be completely in memory.

# Diagram of Virtual Memory





# Implementation: Demand paging scheme

- A process is viewed as a set of sequential pages. It resides on secondary memory (disk).
- A page is never brought into memory unless it will be needed.

# Implementation

- Add valid-invalid bit into page table.
  - “valid” indicates that the associated page is both legal and in memory.
  - “invalid” indicates that the associate page is not in memory.
- Access to a page marked invalid causes a page-fault trap to the OS.

# Handling Page Faults

- Find a free frame.
- Read the desired page into the frame.
- Reset the page table to indicate that the page is in memory.
- Restart the instruction that was interrupted by the page-fault trap. The process now can access the page as though it had always been in memory.

# Performance of Demand Paging (1)

- Assume
  - $m$  : memory access time.
  - $p$  : page-fault rate ( $0 \leq p \leq 1$ ).
  - $f$  : page-fault service time.
- The effective access time for a demand-paged memory:
  - $\text{effective access time} = (1-p) \times m + p \times f$

## Performance of Demand Paging (2)

- If we take an average page-fault service time of 25 milliseconds and a memory access time of 100 nanoseconds, then
  - $\text{effective access time} = (1-p) \times 100 + p \times 25,000,000$
  - $= 100 + 24,999,900 \times p \text{ (nanoseconds)}$
- If one access out of 1000 causes a page fault:
- $\text{effective access time} = 25,099.900 \text{ (nanoseconds)}$
- **Conclusion:** The effective access time is directly proportional to the page-fault rate. It is important to keep the page-fault rate low.

# Page Replacement

(1)

- The page-fault rate is not a problem if each page is faulted at most once.
- What happens if we are over-allocating memory?
- Page replacement: if no frame is free, find one that is not currently being used and free it. The page is freed by writing its contents to swap space, and changing the page table.

# Page Replacement

(2)

- Page replacement may double the page-fault service time and increase the effective access time accordingly.
- Reduce the overhead by using a dirty bit:
  - A dirty bit is associated with each page.
  - The dirty bit is set whenever there is any modification in the page.
  - When a page is selected for replacement, the page is written to the disk only if its dirty bit is set.

# Page-Replacement Algorithms

(1)

- Random Page Replacement
  - Easy-to-implement and low-overhead
  - It may accidentally select the next page to replace the page that will be referenced next.
- First-In-First-Out Page Replacement
  - Replace the one that has been in memory the longest
  - It can replace heavily used pages
- Least-Recently-Used (LRU) Page Replacement
  - Replace the one that has spent the longest time in memory without being referenced.
  - Better performance at the cost of system overhead



# Page Replacement Algorithms (2)

- Least-Frequently-Used Page Replacement
  - Replace the one that is least frequently used.
  - Substantial overhead
- Not-Used-Recently Page Replacement
  - Replace the one that has not been used recently.
- Optimal Algorithm
  - Replace the page that will not be used for the longest period of time.

# Page-Replacement Algorithms:

## Evaluation method

- The lower the page-fault rate the better.
- Evaluation: Run on a string of memory references and count the number of page faults. For example
  - 7, 8, 0, 6, 5, 4, 6, 9, 2, 3, 3
  - Each number is a reference to a page.
  - The more frames available, the lower the number of page faults.

# Page-Replacement Algorithms:

## First-In First-Out (FIFO) Algorithm

- Create a FIFO queue to hold all pages in memory. Always replace the page at the head of the queue, and insert a new page at the tail of the queue.
- Assume there are 3 frames in memory. If run on a reference string
  - 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
  - using FIFO algorithm, how many page faults?

# Page-Replacement Algorithms:

## First-In First-Out (FIFO) Algorithm

Reference strings

|             |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|-------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|             | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
|             | 7 | 7 | 7 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7 | 7 | 7 |
|             |   | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| Page frames |   |   | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 1 |

number of page faults = 15

Bad replacement choices increase the page-fault rate:

The page replaced may be an active page in constant use.  
After paging out this page to bring a new one, a page fault may occur immediately.

---

# Page-Replacement Algorithms: Optimal Algorithm

- Replace the page that will not be used for the longest period of time.
- It has the lowest page-fault rate.
- Assume there are 3 frames in memory. If run on a reference string  
7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1  
using optimal algorithm, how many page faults?

# Page-Replacement Algorithms: Optimal Algorithm

Reference strings

|             |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|-------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|             | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
|             | 7 | 7 | 7 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 7 | 7 | 7 |
|             |   | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Page frames |   |   | 1 | 1 | 1 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

number of page faults = 9

Optimal algorithm is difficult to implement, because it requires future knowledge of the reference string.

---

# Page-Replacement Algorithms:

## Least Recently Used (LRU) Algorithm

- Replace the page that has not been used for the longest period of time.
- An approximation to the optimal algorithm.
- Assume there are 3 frames in memory. If run on a reference string  
7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1  
using LRU algorithm, how many page faults?

# Page-Replacement Algorithms:

## Least Recently Used(LRU) Algorithm

Reference strings

|             |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|-------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|             | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
|             | 7 | 7 | 7 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|             |   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 3 | 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| Page frames |   |   | 1 | 1 | 1 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 7 | 7 | 7 |

number of page faults = 12



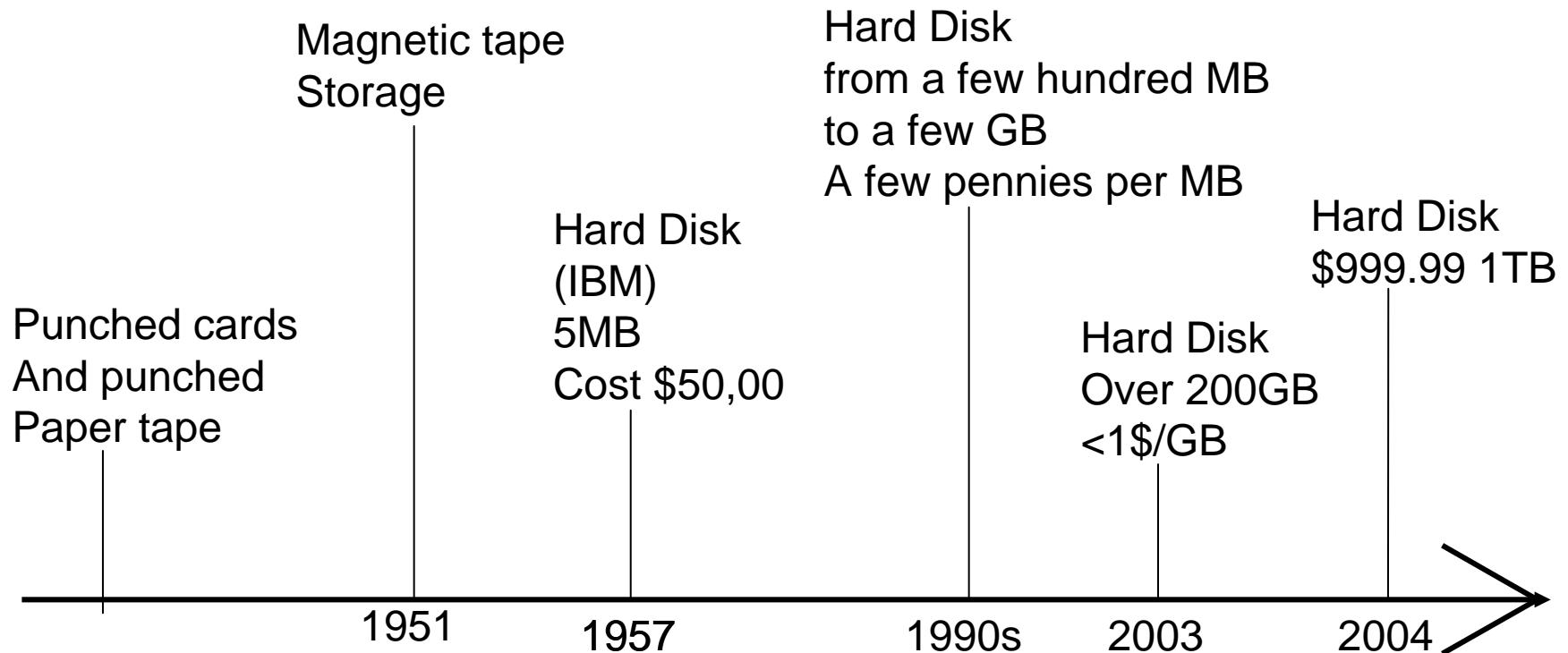
# Summary

- Why is virtual memory needed?
- Overlay and swapping.
- Implementation of Virtual Memory
  - Demand paging scheme.
  - Page-fault handling.
- Performance evaluation
  - effective access time and page-fault rate.
- Page-replacement algorithms:
  - FIFO, Optimal, and LRU.

# Chapter 11

## Disk Performance Optimization

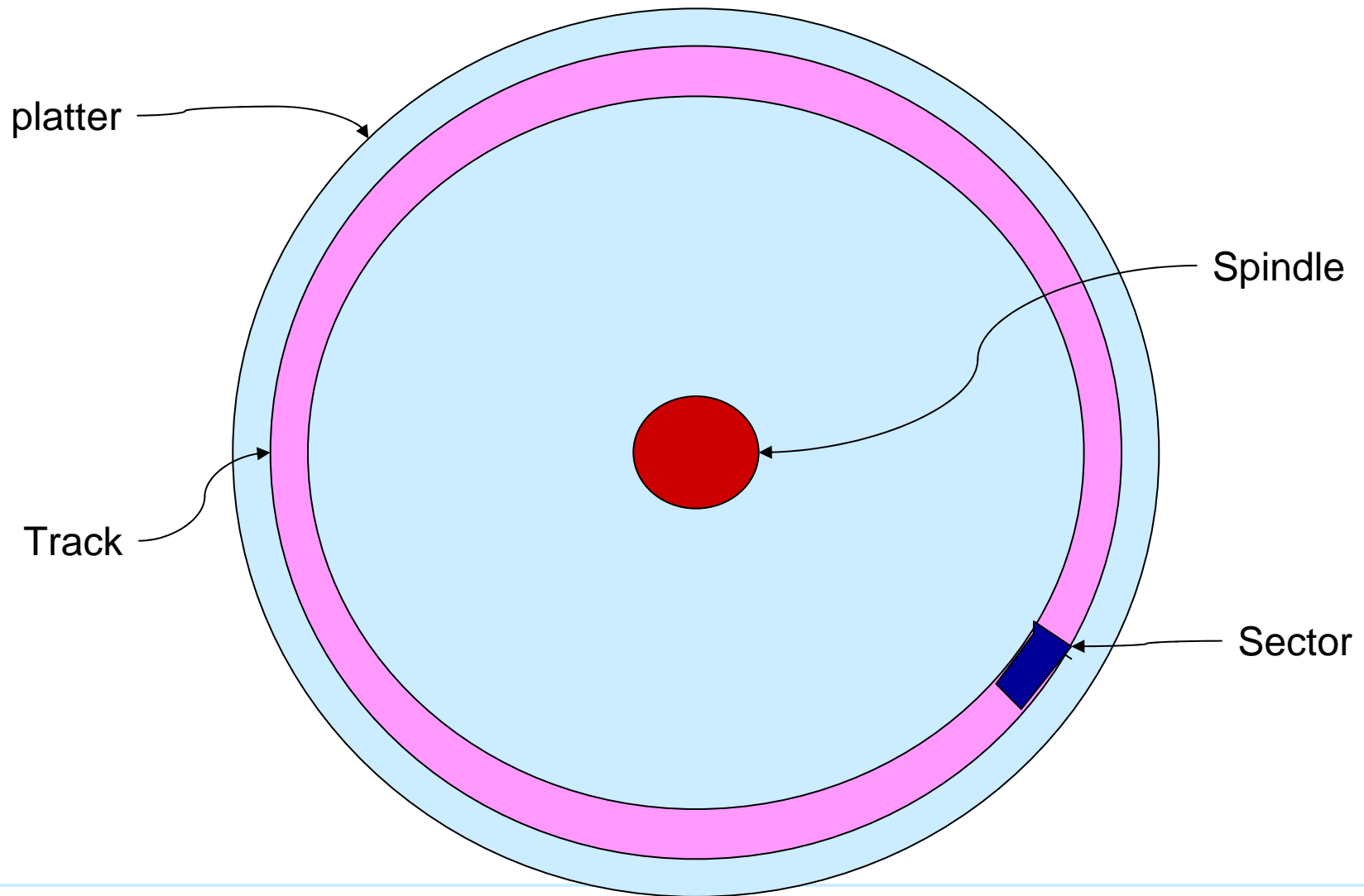
# Evolution of Secondary Storage



# Characteristics of Moving-Head Disk Storage

- Data is recorded on a series of magnetic disks, or **platters**, connected to a **spindle** that rotates at high speed. Each platter has two surfaces.
- The data on each disk surface is accessed by a read-write **head**
- As the platters spin, each head sketches out a circular **track** of data on a disk surface. A vertical set of circular tracks is called **cylinder**. The track is divided into **sectors**, often 512bytes.

# Picture of a Platter



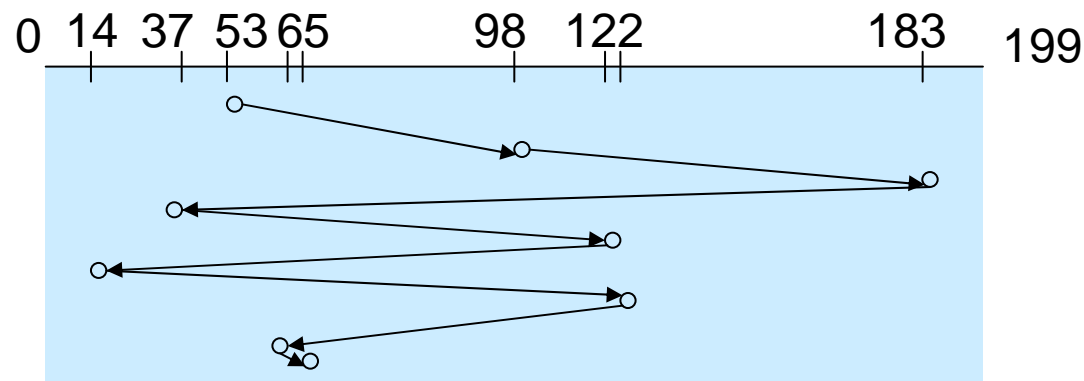
# Disk Access Time and bandwidth

- The Disk access time has two major components
  - The seek time is the time for the disk arm to move the heads to the cylinder containing the desired sector.
  - The rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head.
- The disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.
- Both the access time and bandwidth can be improved by scheduling the servicing of disk I/O requests.

# Disk Scheduling:

## First-Come First-Serve Scheduling

For example, a disk queue with requests for I/O to sectors on cylinders  
98,183,37,122,14,124,65,67  
The head is initially at cylinder 53.

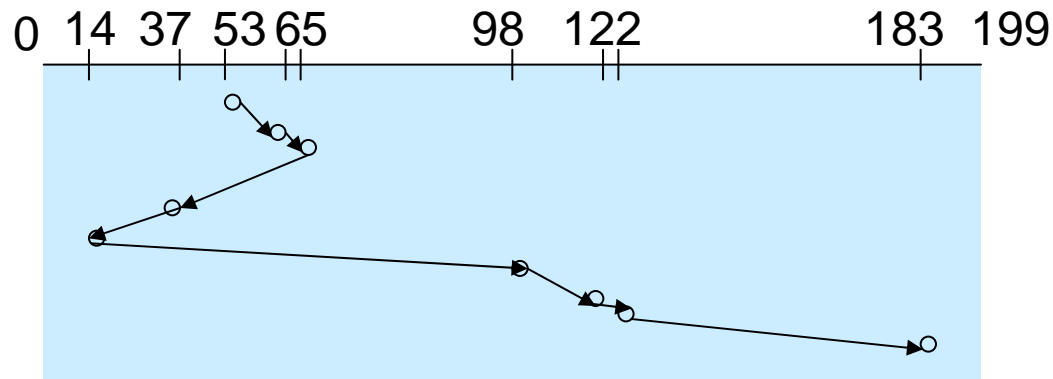


The total movement is 640 cylinders

# Disk Scheduling:

## Shortest-Seek-Time-First Scheduling

For example, a disk queue with requests for I/O to sectors on cylinders  
98,183,37,122,14,124,65,67  
The head is initially at cylinder 53.



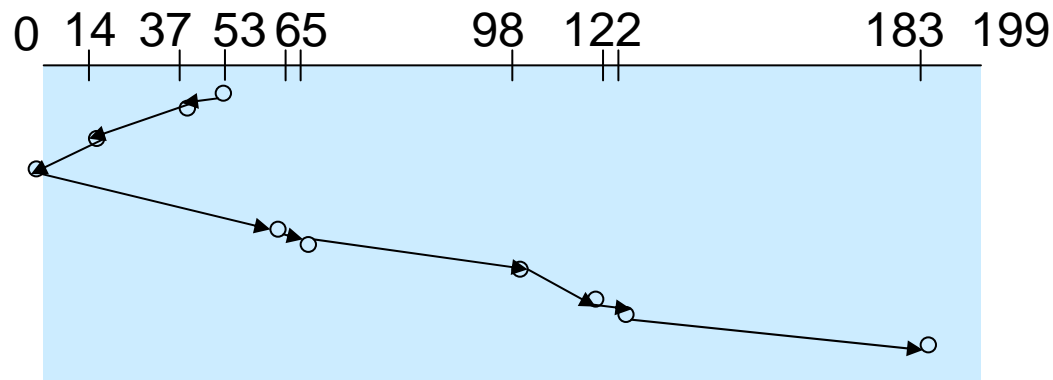
The total movement is 208 cylinders



# Disk Scheduling: Scan Scheduling

The disk arm starts at one end of the disk and moves toward the other end.

For example, a disk queue with requests for I/O to sectors on cylinders  
98,183,37,122,14,124,65,67  
The head is initially at cylinder 53.

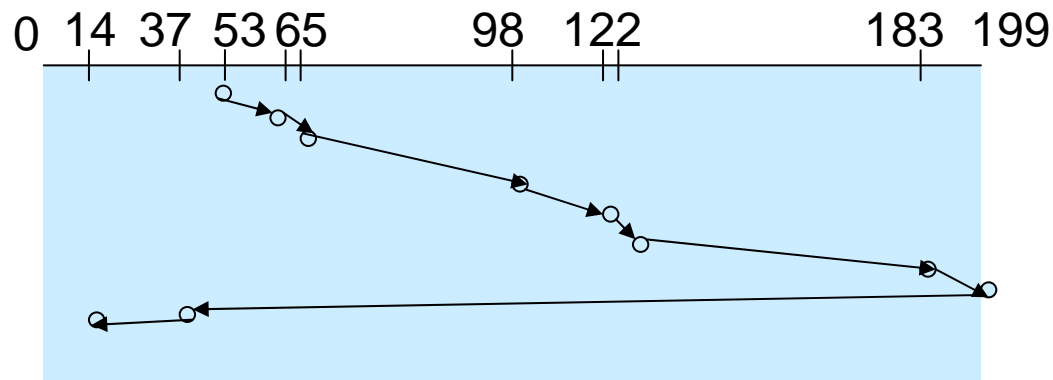


The total movement is 236 cylinders

# Disk Scheduling: C-Scan Scheduling

It is designed to provide a more uniform wait time. The disk arm moves the head from one end to the other end, serving requests along the way.

For example, a disk queue with requests for I/O to sectors on cylinders 98, 183, 37, 122, 14, 124, 65, 67. The head is initially at cylinder 53.

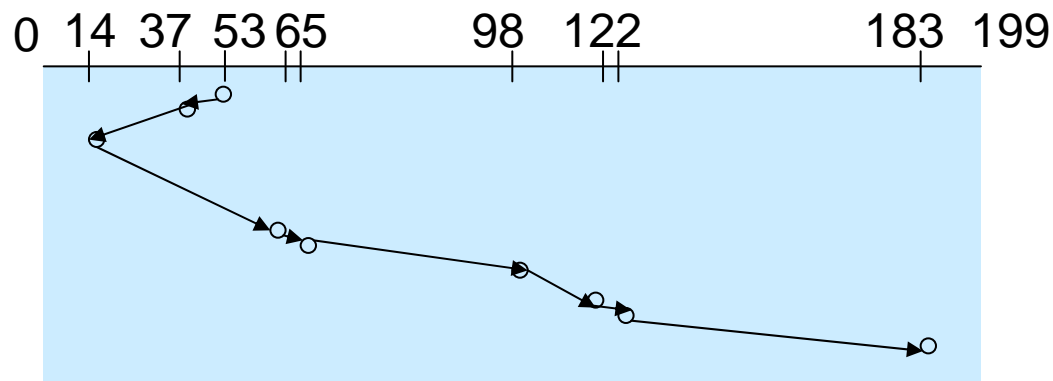


The total movement is 331 cylinders

# Disk Scheduling: Look Scheduling

Similar with Scan scheduling, but the disk arm only goes as far as the final request in each direction.

For example, a disk queue with requests for I/O to sectors on cylinders  
98,183,37,122,14,124,65,67  
The head is initially at cylinder 53.

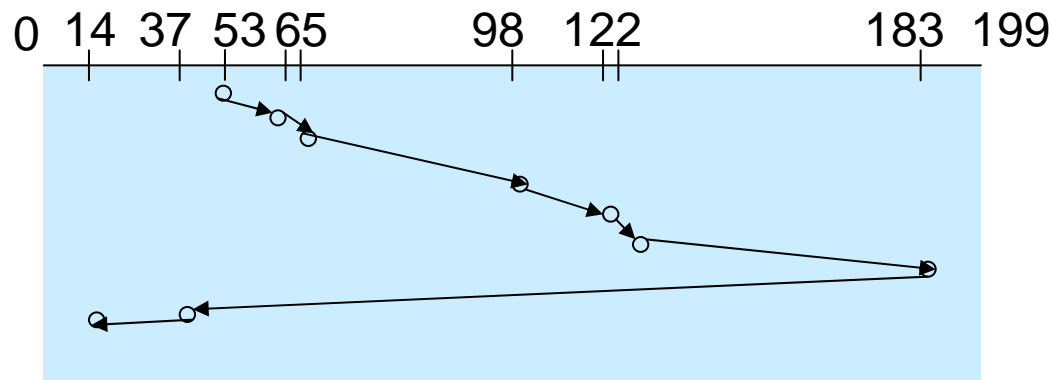


The total movement is 208 cylinders

# Disk Scheduling: C-Look Scheduling

Similar with C-Scan scheduling, but the disk arm only goes as far as the final request in each direction.

For example, a disk queue with requests for I/O to sectors on cylinders  
98,183,37,122,14,124,65,67  
The head is initially at cylinder 53.



The total movement is 299 cylinders

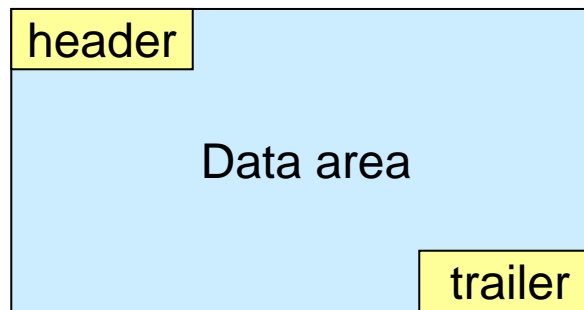
# Caching and buffering

- A disk cache buffer is a region of main memory reserved for disk data. It allows the system to quickly access data that would otherwise need to be fetched from disk.
  - Write-back caching: the system writes the modified data to disk periodically, enabling the system to batch multiple I/O requests.
  - Write-through caching: the system writes the modified data to disk immediately.
- Challenges:
  - The size of the buffer.
  - Replacement strategies

# Disk Formatting

(1)

- Low-level formatting: before a disk can store data, it must be divided into sectors that the disk controller can read and write.
  - The data structure of a sector



- The header and trailer store information used by the disk controller, such as a sector number and an error-correcting code (ECC).

# Disk Formatting

(2)

- To use a disk to hold files, the system needs to record its own data structures on the disk. It does this in two steps.
  - Partition the disk into one or more groups of cylinders. The disk treats each partition as a separate disk.
  - Do a logical formatting on each partition. This allows the system to store the initial file-system data structures onto the disk.

# Boot Block

(1)

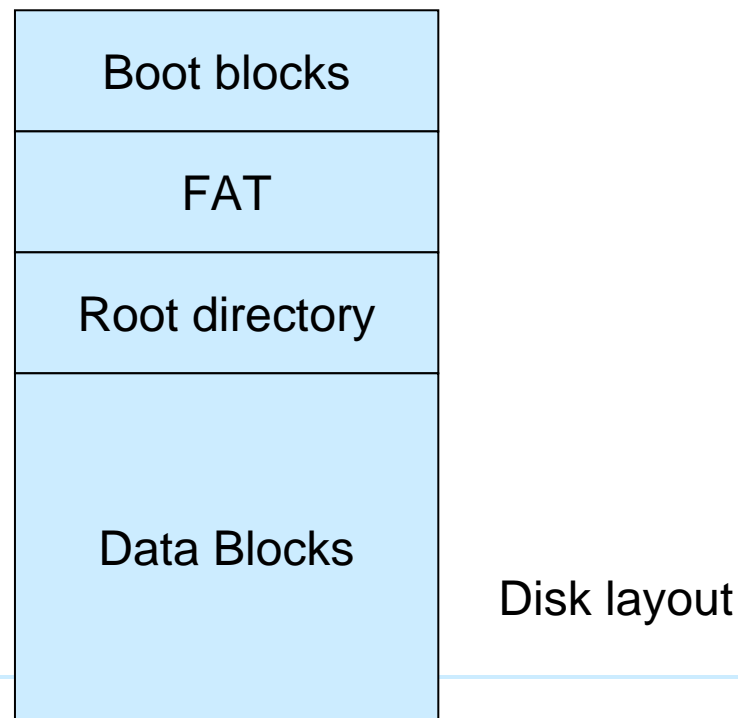
- When a computer is powered up, it needs to have an initial program to run, which is the bootstrap program.
- The bootstrap program is usually stored in ROM (Read-Only Memory)
  - It is read only
  - It is difficult to update.
    - Solution: Store a small bootstrap loader program in the boot ROM, whose job is to bring in a full bootstrap program from disk.



# Boot Block

(2)

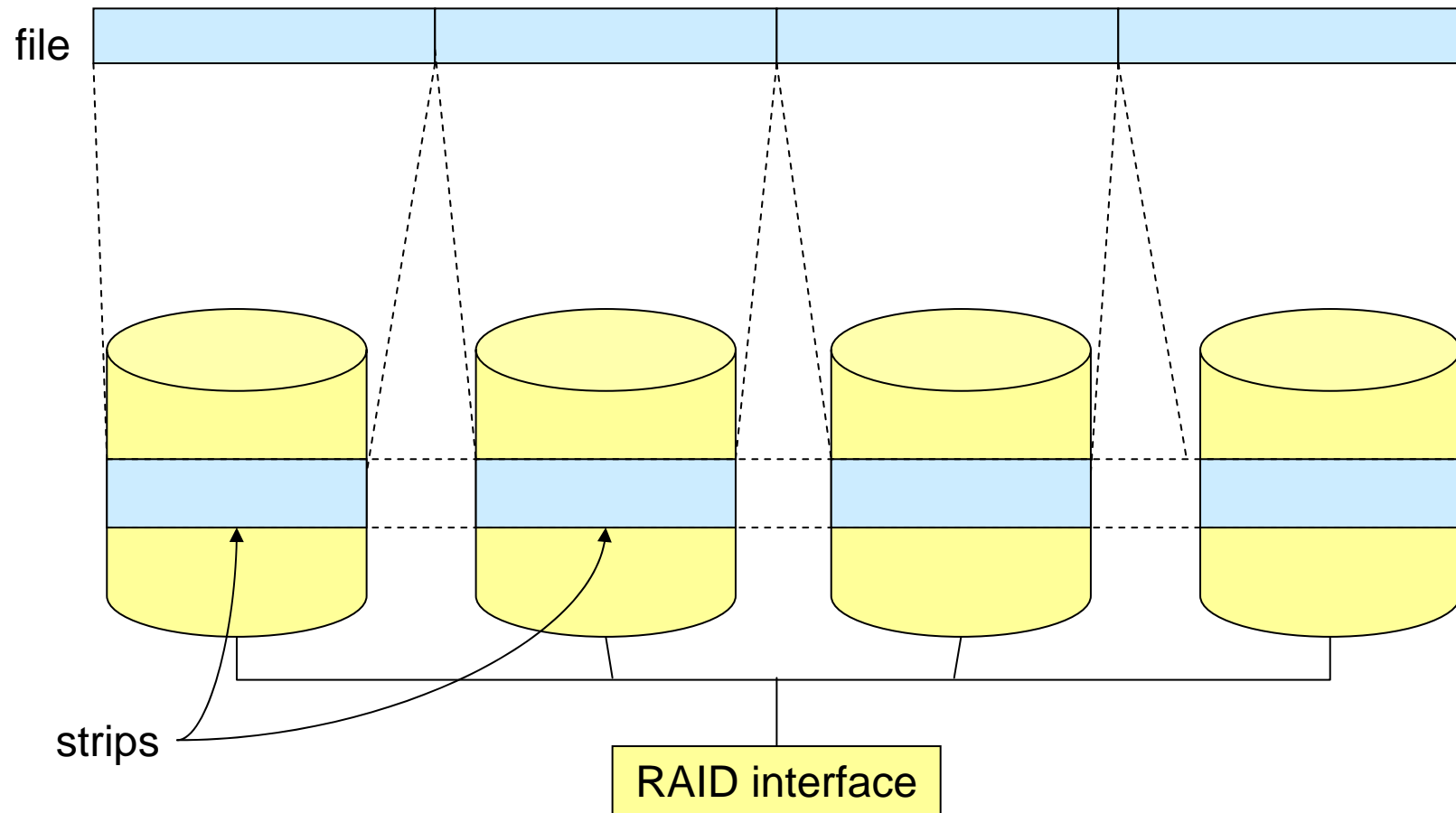
- The full bootstrap program is stored in a set of data blocks called the boot blocks, at a fixed location on the disk.
- A disk that has the boot blocks is called boot disk or system disk.



# Redundant Arrays of Independent Disks (RAID) (1)

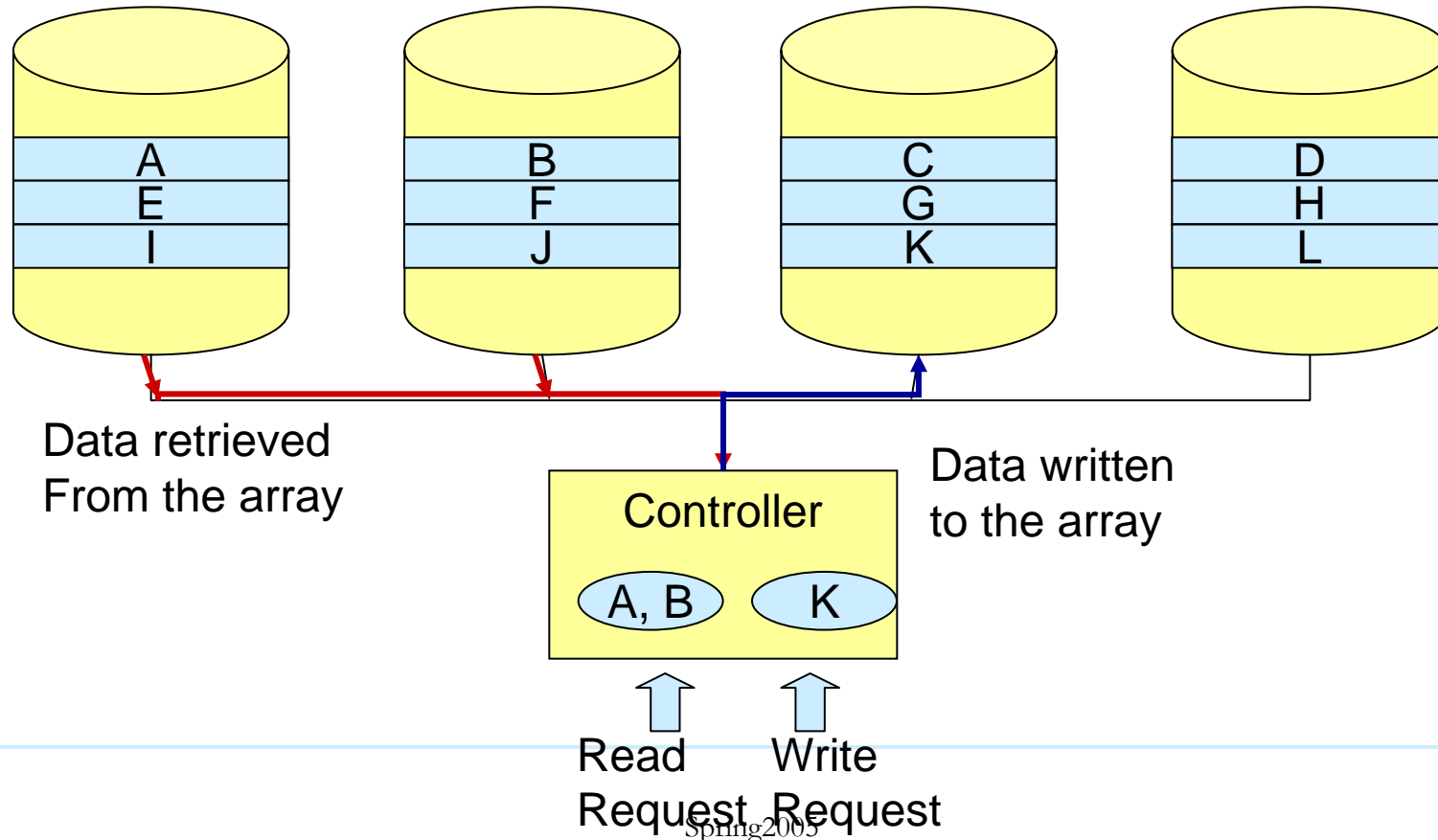
- RAID is a family of techniques that use multiple disks that are organized to provide high performance and reliability.
- The storage of each disk is divided into fixed-size blocks called strips. Contiguous strips of a file are typically placed on separate disks so that a request for file data can be serviced using multiple disks at once.
- There are over five different organizations, or levels, of disk arrays.

# Redundant Arrays of Independent Disks (RAID) (2)



# Level 0 (Striping)

- It uses a striped disk array with no redundancy.
- With  $n$  disks, it is  $n$  times faster than a single disk.



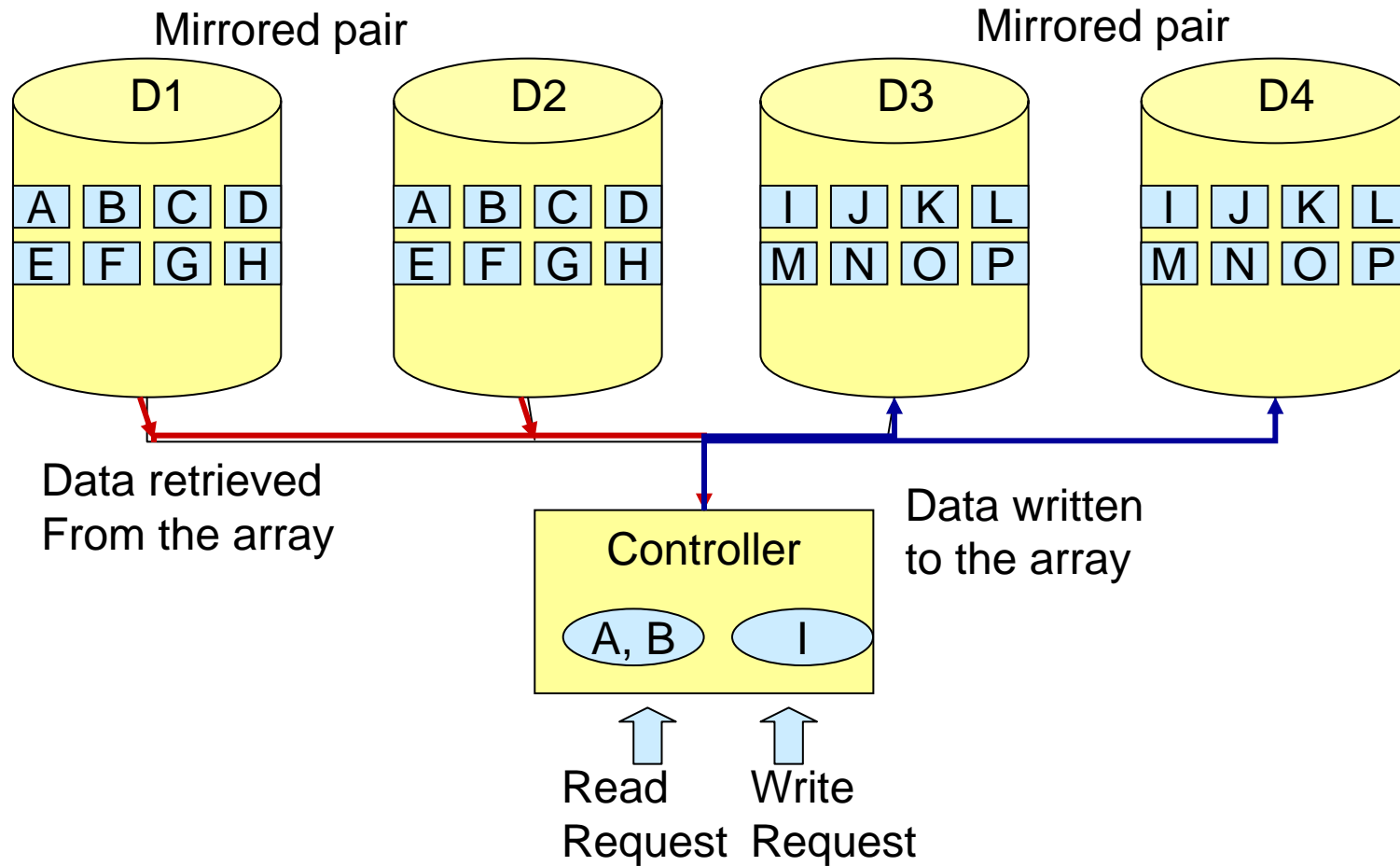
# Level 1 (Mirroring)

(1)

- It uses disk mirroring to provide redundancy. So each disk in the array is duplicated.
- Strips are not provided in this level.
- High fault tolerance – It provides the highest degree of fault tolerance among the most popular RAID levels.
- High cost – Only half of the array's disk capacity can be used to store unique data.

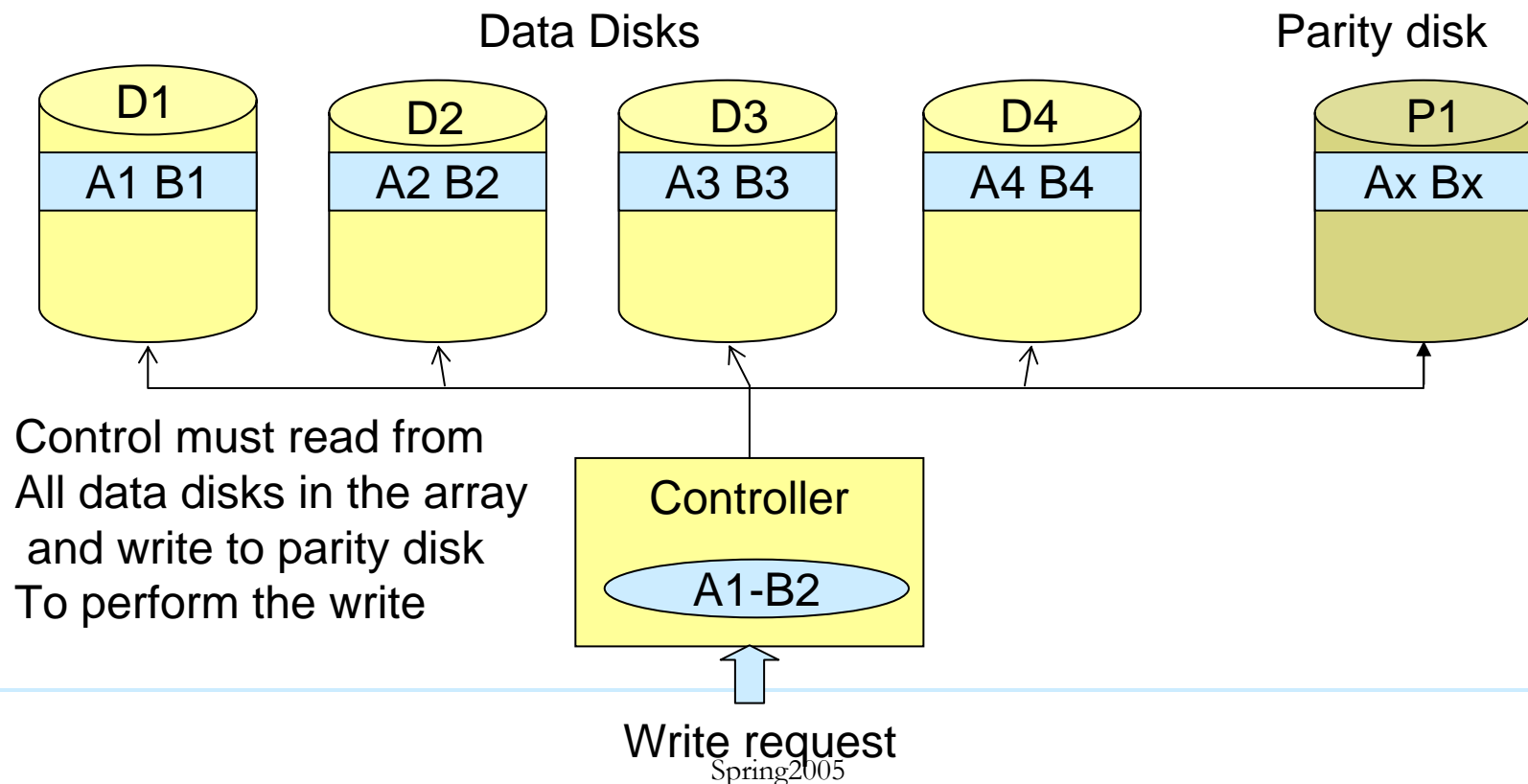
# Level 1 (Mirroring)

(2)



## Level 2 (Bit-level Hamming ECC Parity)

- Its arrays are striped at the bit level.
- Hamming Error-Correcting Codes (ECCs) is used to check errors in data transmission and correct them.

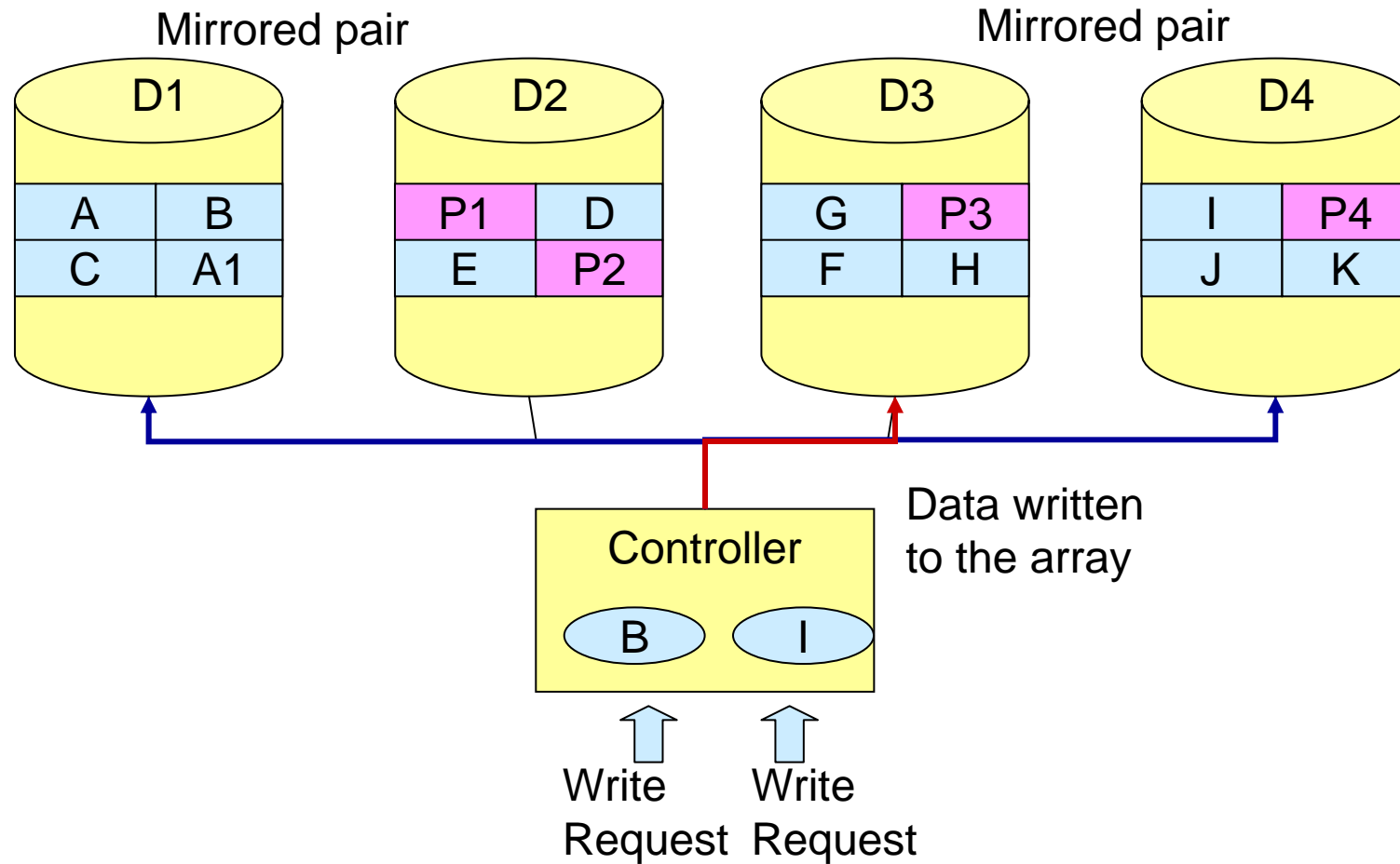


## Level 3, 4 and 5

- Level 3 (Bit-level XOR ECC parity)
- Level 4 (Block-level XOR ECC parity)
  - It is striped using fixed-size blocks.
- Level 5 (Block-level Distributed XOR ECC parity)
  - The parity blocks are distributed throughout the array of disks.



# Level 5 (Block-level Distributed XOR ECC parity)



---

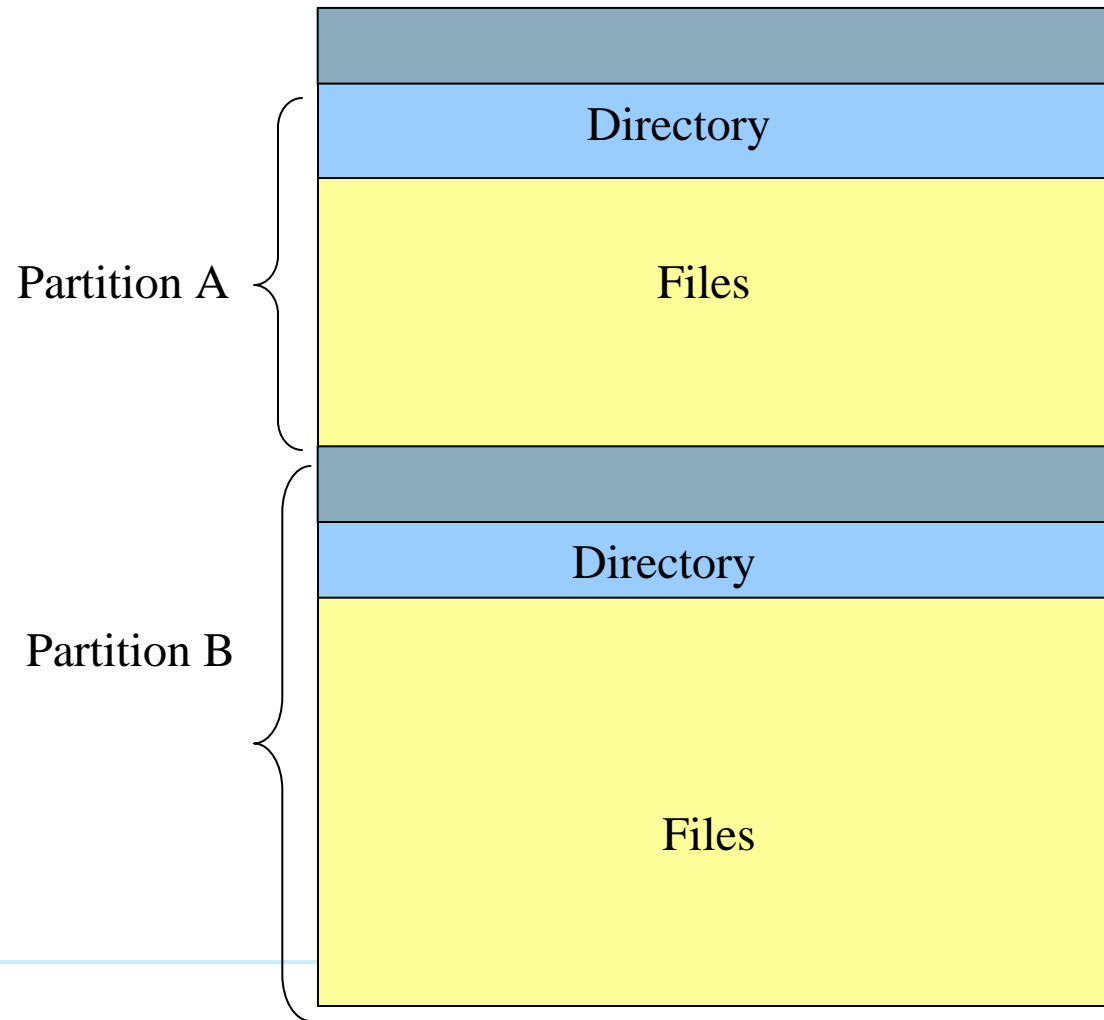
# Summary

- Evolution of secondary storage
- Disk structure
- Disk scheduling
- Caching and buffering
- Disk management
  - Disk formatting
  - Boot block
- RAID organizations: levels 0-5

# Chapter 12

## File and Database Systems

# Typical File System



# File Concepts

- File attributes: name, type, location, size, protection, time, date, and user identification. The information about all files is kept in the directory structure.
- File operations: creating, writing, reading, reposition, deleting, truncating, appending, and renaming.

# File systems

- A file system organizes files and manages access to data. It is responsible for:
  - File management – provides mechanisms for files to be stored, referenced, shared and secured.
  - Auxiliary storage management – allocates space for files on secondary and tertiary storage devices.
  - File integrity mechanisms – ensures that the information stored in a file is not corrupted.
  - Access methods – how the stored data can be accessed.

---

# Directory Structure

## Tree-structured Directories

- The tree has a root directory.
- Each file in the system has a unique path name.
- A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way.
- Relative and absolute paths

# Directories – Links

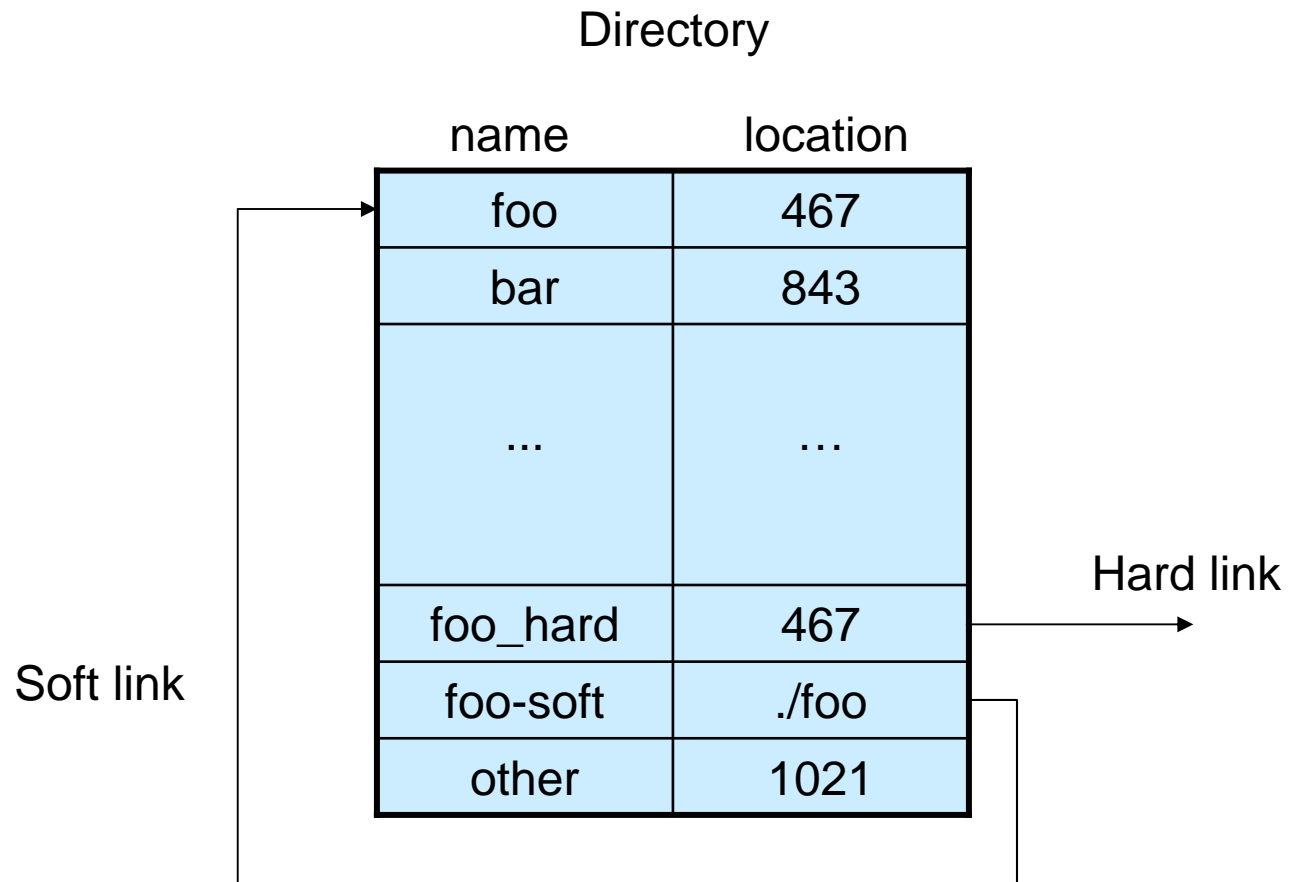
(1)

- A link is a directory entry that references a data file that is typically located in a different directory. It is often used to simplify file system navigation and to share files.
  - A soft link( “shortcut”, “alias”) is a directory entry containing the pathname for another file.
  - A hard link is a directory entry that specifies the location of the file on the storage device.



# Directories – Links

(2)



# Metadata

- File systems need to store data (Metadata) other than user data and directories, such as
  - A file system identifier
  - The number of blocks in the file system
  - The locations of the storage device's free blocks
  - The location of the root directory
  - The date and time at which the file system was last modified
- Metadata is used to protect the integrity of the file system and cannot be modified directly by users.

# File Descriptor

(1)

- A open-file table that keeps track of open files is maintained by the OS to avoid repeated traversals over the directory.
  - It contains a file control block for each open file, which includes
    - File name
    - Location in secondary storage
    - Data access method
    - Device type
    - Access control data

# File Descriptor

(2)

- Type
  - Disposition (permanent or temporary)
  - Creation data and time
  - Date and time last modified
  - Access activity counts
- 
- A file descriptor is a non-negative integer that indexes into the open-file table. It is returned by a file open operation.

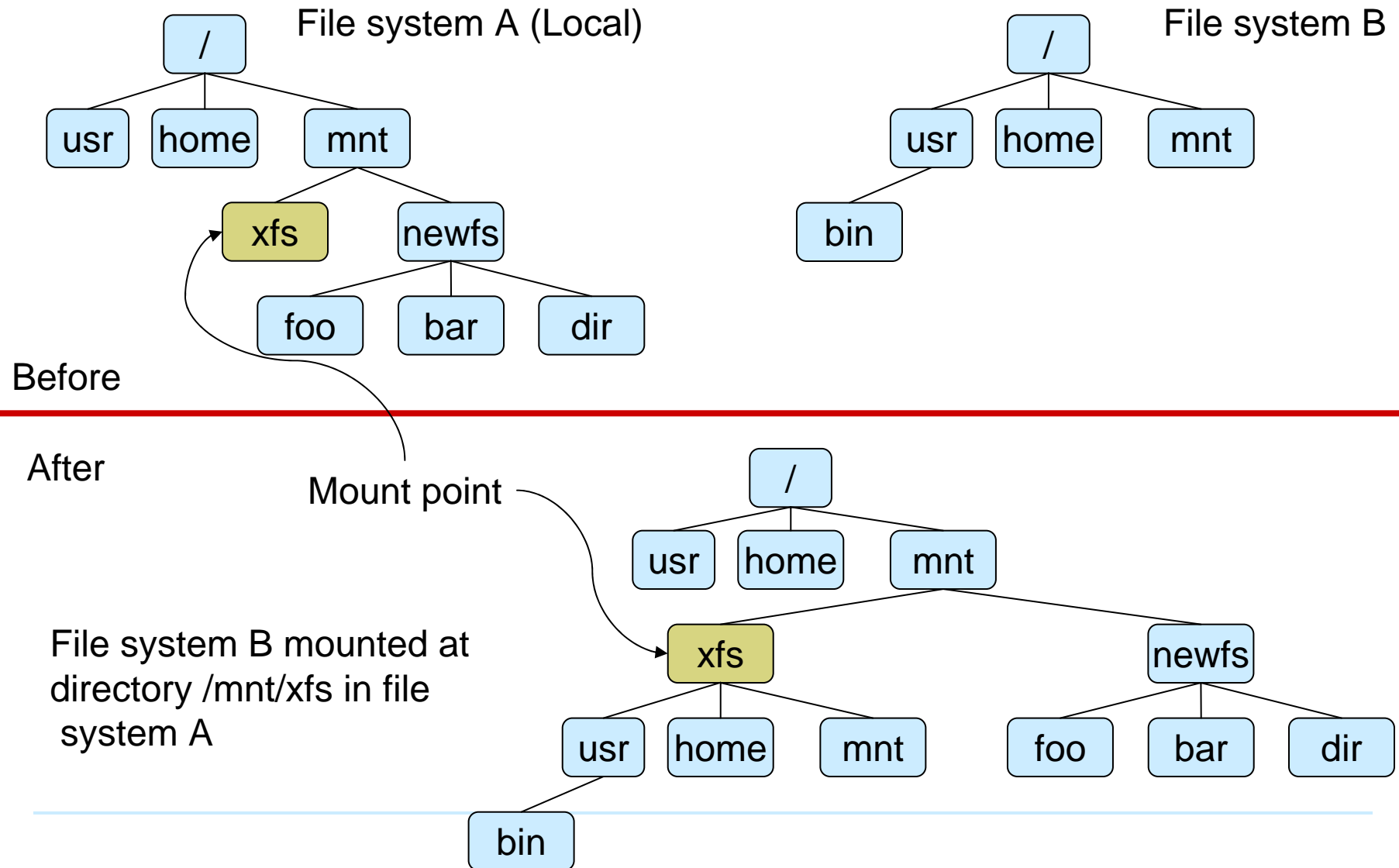
# Mounting

(1)

- Users often require access to information that is not part of the local file system.
- Mounting combines multiple file systems into one namespace, so that the files in the multiple file systems can be identified by a single file system.
  - It assigns a directory, called the mount point, in the local file system to the root of the mounted file system.

# Mounting

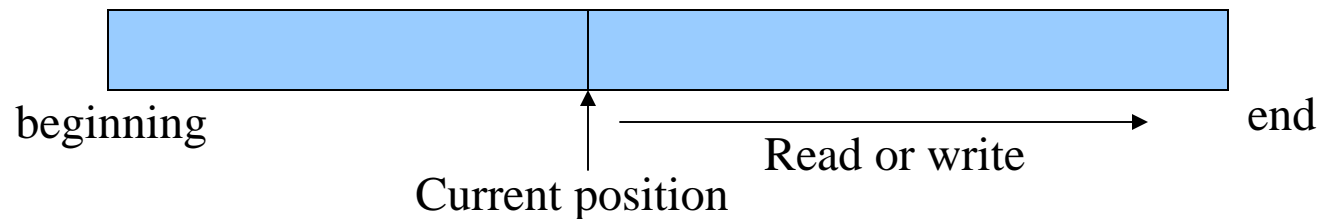
(2)



# File Access Methods:

## Sequential Access

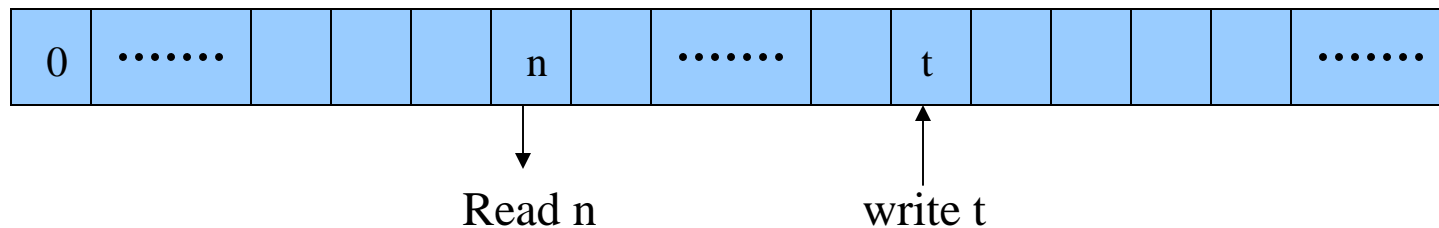
- Information in the file is processed in order.



# File Access Methods:

## Direct Access

- A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.





# File Organization

- Refers to the manner in which the records of a file are arranged on secondary storage.
  - Sequential – records are placed in physical order.
  - Direct – records are directly (randomly) accessed by their physical addresses on a direct access storage device.
  - Indexed sequential – Records on disk are arranged in logical sequence according to a key contained in each record.
  - Partitioned – This is a file of sequential subfiles. Each sequential subfile is called a member. The starting address of each member is stored in the file's directory.

---

# File Allocation Methods:

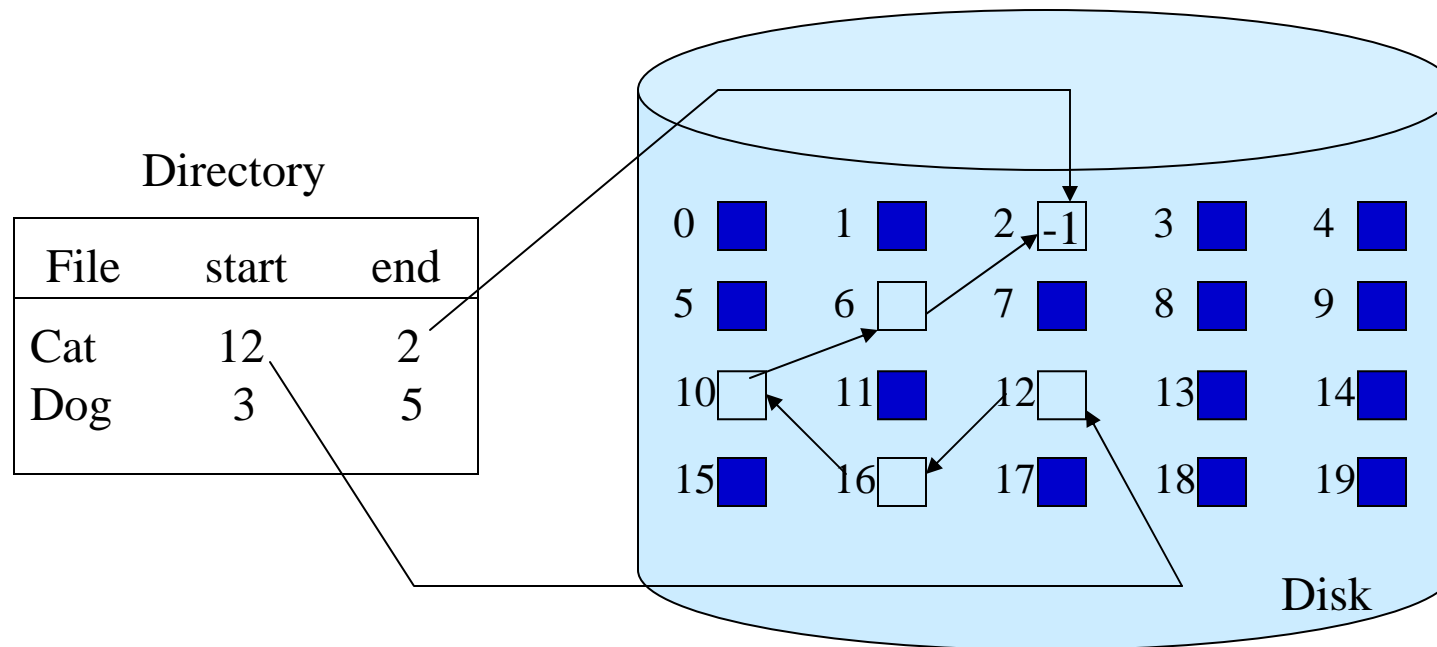
## Contiguous Allocation

- Each file occupies a set of contiguous blocks of disk.
- Benefits:
  - Quick and easy calculation of block holding data.
  - Supports both sequential and direct access.
- Disadvantages:
  - Where is the best place to put a new file?
  - Problems when the file gets bigger.
  - Fragmentation problem.

# File Allocation Methods:

## Linked Allocation (1)

- Each file is a linked list of disk blocks which may be scattered anywhere in the disk.



# File Allocation Methods:

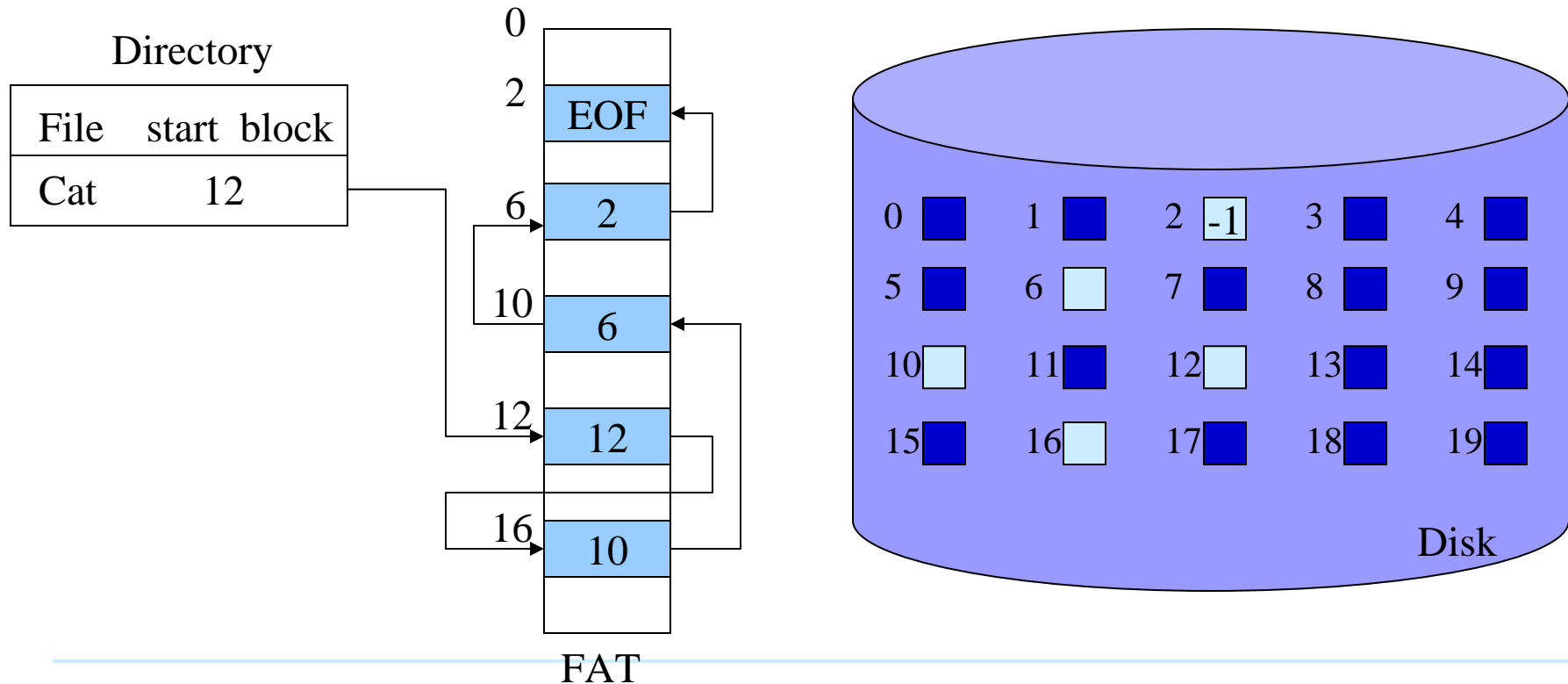
## Linked Allocation (2)

- Benefits: solves all the problems of contiguous allocation.
- Disadvantages:
  - Potentially terrible performance for direct access.
  - Each file needs more space because each block contains a pointer to the next block.
  - Reliability problem – what if one pointer is lost?

# File Allocation Methods

## – FAT Allocation (1)

- Use a file allocation table (FAT) – A variation on Linked Allocation.



---

# File Allocation Methods

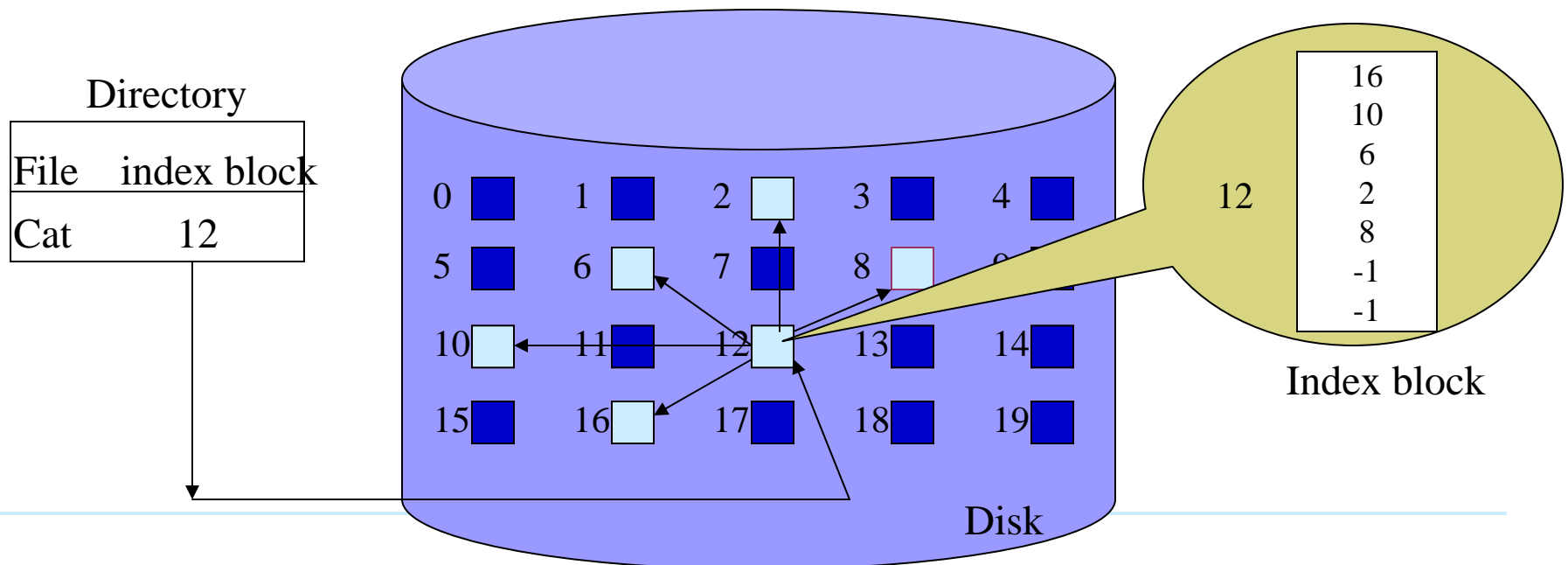
## – FAT Allocation (2)

- Benefit: direct access time is improved.
- Disadvantage: Results in a significant number of disk head seeks.
- What happens if FAT is cached?

# File Allocation Methods

## – Indexed Allocation (1)

- Each file has its own index block, which is an array of disk-block addresses. The  $i$ th entry in the index block points to the  $i$ th block of the file.



# File Allocation Methods

## – Indexed Allocation (2)

- Benefits

- Solves fragmentation and size-declaration problems of contiguous allocation.
- Support direct access, but not as efficient as FAT.

- Disadvantages

- Wasted space due to overhead of index block



# Free Space Management

- A file system needs to maintain a record of the location of blocks that are available to store new data.
  - Use a linked-list (free list) of free blocks.
  - Use a bit map. A bit map contains one bit for each block in the file system, where the  $i$ th bit corresponds to the  $i$ th block in the file system.

# File Access Control

(1)

- Use access control matrix.
  - The entry  $a_{ij}$  is 1 if user  $i$  is allowed access to file  $j$

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

# File Access Control

(2)

- Access control by user classes.

- Owner
- Specified user
- Group
- Public

The access control data is usually stored as part of the file control block

# Data Integrity Protection – Backup

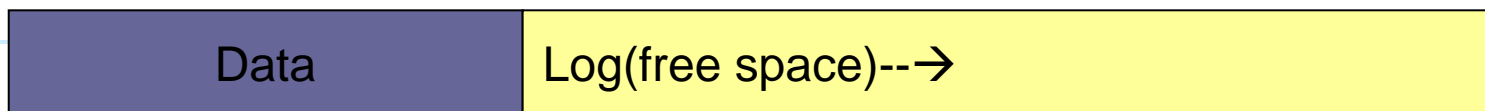
- A physical backup duplicates a storage device's data at the bit level.
  - It stores no information about the logical structure of the file system.
  - It is difficult to be restored on computers with different architecture.
  - The whole file system has to be recorded and restored.
- A logical backup stores data in a common format using a directory structure, allowing systems with different native file formats to read and restore the backup data.
  - An incremental backup is a logical backup that only stores file system data that has changed since the previous backup.

# Data Integrity and Log-Structured File System (LSF) (1)

- If a system failure occurs during a write operation, the file system data may be left in an inconsistent state.
- Transaction-based logging reduces the risk of data loss by using atomic transaction.
  - A group of operations are performed in their entirety, or not at all. If an error occurs before completion of the transaction, the system returns back to the state before the transaction began.
  - It is usually implemented by recording the result of each operation in a log file. Once the transaction is completed, the log is transferred to permanent storage.

# Data Integrity and Log-Structured File System (LSF) (2)

- With transaction logging, the file system may still get in a inconsistent state.
- An LSF performs file system operations as logged transactions.
  - The entire disk serves as a log file to record transactions.
  - Modified directories and metadata are always written to the end of the log.



# Database Systems

- A database is a centrally controlled, integrated collection of data
- A database system involves
  - the data
  - the hardware on which the data resides
  - the software that controls access to data (called a database management system or DBMS)
- Databases are commonly implemented on web servers and shared by multiple processes.

---

# Advantages of Database Systems

- Database system organizes data according to its content.
- It incorporates a querying mechanism that allows applications to specify which information to retrieve.
- It uses standardized data organization techniques and its structure cannot be altered by applications.



# Data Access in Database Systems

- Database systems are data-independent systems enabling multiple applications to access the same data with different logical views.
- Database languages facilitate data independence by providing a standard way to access information. A database language consists of
  - A data definition language (DDL)
  - A data manipulation language (DML)
  - A query language.

# Relational Database Model

(1)

- A relational database is composed of **relations** (tables). A row of the relation (table) is called a **tuple**. One of the attributes (column) in each tuple is used as the **primary key** for referencing data in the relation. The tuples of the relation are uniquely identifiable by the primary key.

| tuple | Number | Name       | Department | Salary | Location |
|-------|--------|------------|------------|--------|----------|
|       | 1234   | Jones, A   | 712        | 1100   | London   |
|       | 2347   | Kerwin, T  | 003        | 1456   | Cardiff  |
|       | 1009   | Myers, B   | 323        | 3200   | London   |
|       | 789    | Stevens, L | 198        | 2001   | Cardiff  |

Primary  
key

attribute

# Relational Database Model

(2)

- Project operation – Creates a subset of a database which contains only certain subsets of the table's rows and columns.
- Join operation – Combines small database tables into a larger one.

# Relational Database Model

(3)

## ■ Advantages

- It is easy to be implemented in a physical database system.
- It is relatively easy to convert any other database structure into the relational model.
- The project and join operations are easy to implement and make the creation of new relations required for particular applications easy to perform.
- Access control to sensitive data is straightforward to implement.

---

# Summary

- File concepts.
- File systems: directories, metadata and mounting.
- File access methods – sequential and direct access.
- File organization
- File allocation methods: contiguous, linked, linked with a FAT, and indexed.
- Free space management
- File access control
- Data integrity protection
- Database systems

# Chapter 13

## I/O Systems

# I/O Hardware (1)

- Computers operate many kinds of devices:
  - Storage devices: disks, tape.
  - Transmission devices: network cards, modems.
  - Human-interface devices: screen, keyboard, mouse.
- A device communicates with a computer system by sending signals over a cable or bus via an I/O port.
- A controller is a collection of electronics that can operate a port, a bus, or a device.

# I/O Hardware

(2)

- An I/O port typically consists of four registers:
  - Status register is read by the host to get the status of a device.
  - Control register is written by the host to start a command.
  - Data-in register is read by the host to get input.
  - Data-out register is written by the host to send output.



# Polling

- The CPU repeatedly reads the **busy** bit in the status register until that bit becomes clear.
- It is efficient if the controller and device are fast and always ready for service.
- It is inefficient if they are not.
- It may be more efficient to have the device controller to notify the CPU that a device is ready for service – Interrupt.

# Interrupts

(1)

- The CPU senses the interrupt request line after executing every instruction.
- The I/O interrupt cycle:
  - The device controller raises an interrupt by asserting a signal on the interrupt request line.
  - the CPU detects the interrupt, saves its state, and then jumps to the interrupt-handler routine at a fixed memory address.

# Interrupts

(2)

- The interrupt handler determines the cause of the interrupt, performs the necessary processing.
- The interrupt handler returns the CPU to the execution state prior to the interrupt.
- Enables the CPU to respond to an asynchronous event.
- More sophisticated interrupt features needed for a modern operating system.

# Direct Memory Access (DMA)

- PIO – A process is used to transfer data between the CPU and a device by watching status bits and feeding data into a data register 1 byte at a time.
- It is expensive to use a general-purpose processor for PIO.
- Use a special-purpose processor called direct-memory-access controller for PIO.

# DMA Data transfer

- CPU writes a DMA command block into memory. The block contains:
  - A pointer to the source of the transfer.
  - A pointer to the destination of the transfer.
  - The number of bytes to be transferred.
- CPU writes the address of the DMA command block to the DMA controller.

# Kernel I/O Subsystem

- The services provided by the kernel's I/O subsystem:
  - I/O scheduling.
  - Buffering.
  - Catching.
  - Spooling.
  - Device reservation.
  - Error handling.

# I/O Scheduling

- Scheduling can improve the overall system performance, can share a device fairly among processes, and can reduce the average waiting time.
- A queue of requests is maintained for each device. The I/O system can rearrange the order of the queue to improve the overall system efficiency.

# Buffering

- A buffer is a memory area that stores data while they are transferred between two devices or between a device and an application.
- Why buffer?
  - To cope with speed mismatch between the producer and consumer of a data stream.
  - To adapt between devices that have different data-transfer size.
  - To support copy semantics for application I/O.



# Caching

- A cache is region of fast memory that holds copies of data.
- Access to the cached copy is more efficient than access to the original.
- The difference between a buffer and a cache:
  - A cache only holds a copy of a data item on faster storage, whereas a buffer may hold the only existing copy of the data item.

# Spooling

- A spool is a buffer that holds output for a device.
- An example: several applications wish to print their output concurrently.
  - Each output is spooled to a separate disk file, and added into a queue.
  - The spooling system copies the queued spooling file to the printer one at a time.

# Error Handling

- Generally, an I/O system call will return a bit of information about the status of the call, signifying either success or failure
- A failed call results in recovery processing by the operating system

# Summary

- The basic I/O hardware elements: buses, controllers, devices.
- Mechanisms for interaction between device controller and CPU: polling, interrupts, and DMA.
- I/O subsystem provides services such as I/O scheduling, buffering, caching, spooling, and error handling.

# Chapter 14

## Performance and Processor Design

# Introduce

- It is important to determine how effectively a particular system manages its resources.
- The performance of a system depends heavily on its hardware, operating system and the interaction between the two.
- Evaluation techniques required to measure the performance of the system.

## Common reasons for performance evaluation

- Selection evaluation – determines whether a system from a vendor is appropriate
- Performance projection – estimates a new system that doesn't exist.
- Performance monitoring – accumulates performance data on an existing system to ensure that it is meeting its performance goals.

# Performance Measures

(1)

- ❑ Turnaround time – interval from submission to completion of a job.
- ❑ Response time – interval from a user's pressing Enter key or clicking a mouse until the system displays its response.
- ❑ Reaction time – interval from a user's pressing an enter key or clicking a mouse until the first time slice of service is given to that user's request.



# Performance Measures

(2)

- ❑ Throughput – work-per-unit-time
- ❑ Workload – the amount of work that has been submitted to the system
- ❑ Capacity – the maximum throughput a system can attain
- ❑ Utilization – The fraction of time that a resource is in use.

# Performance Evaluation Techniques (1)

- Tracing and profiling
  - Traces are records of system activity executing in user mode.
  - Profiles are records of system activity executing in kernel mode.
- Timing and micro-benchmarks
  - Timing is used to evaluate the performance of computer hardware.
    - MIPS – Millions of instructions per second
    - BIPS – Billions of instructions per second
    - TIPS – Trillions of instructions per second
  - Micro-benchmarks measure the time required to perform an operating system operation.

# Performance Evaluation Techniques (2)

## ■ Benchmarks

- A benchmark is a program that is executed to evaluate a machine.
  - Such as SPECmarks, SYSmark benchmark, WebMark, MobileMark ..

## ■ Synthetic programmes

- They are similar to benchmarks, except that they focus on a specific component of the system.
  - Such as WinBen 99, IOStone, STREAM

# Performance Evaluation Techniques (3)

- Simulation

- A computerized model of the system is used to evaluate the system.

- Performance Monitoring

- Collect and analyse the information regarding the performance of an existing system.

# Performance Techniques in Processor Design

- Instruction set architecture (ISA) is an interface that describes the processor, including its instruction set, number of registers and memory size.
  - It is hardware equivalent to an OS API.
  - The ISA design decisions directly affect a processor's performance.

# Complex Instruction Set Computing (CISC)

- Incorporate frequently used sections of code into single machine-language instructions (CISC instructions) to get better performance.
- The complex instructions are implemented via microprogramming.
- It moves most of the complexity from the software to the hardware.
- It also reduces the size of programs.
- Examples: Pentium, Athlon

# Reduced Instruction Set Computing (RISC)

- Provides fewer instructions, most of which can be performed quickly.
- The programming complexity is moved from hardware to software.
- RISC control units are implemented in hardware.
- Supports only fixed-length instructions, and makes better use of pipelines.
- Examples: MIPS, SPARC, G5

# Post-RISC Processors

- Superscalar architecture
- Out-of-Order Execution (OOO)
- Branch prediction
- On-chip floating and vector processing support
- Additional infrequently-used instructions
- CISC convergence to RISC



---

# Summary

- Common purposes for performance evaluation
- Performance measures
- Performance Evaluation Techniques
- Performance Techniques in processor design

# Chapter 15

## UNIX System

# UNIX Layer Structure

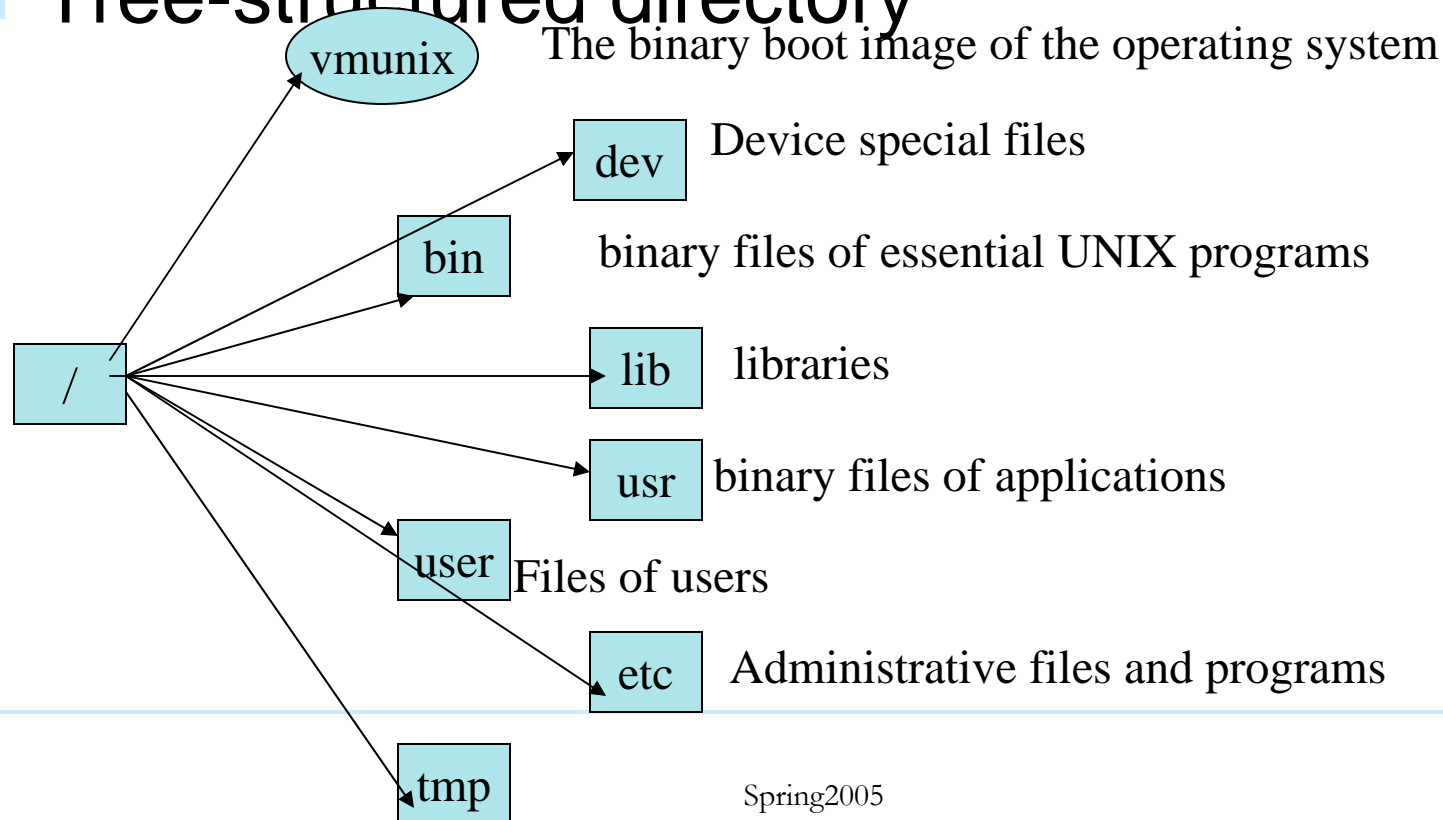
|   |                                       |                                       |
|---|---------------------------------------|---------------------------------------|
| users   |                                       |                                       |
| Shells and commands<br>compilers and interpreters<br>system libraries |                                       |                                       |
| System-call interface to the kernel                                   |                                       |                                       |
| CPU scheduling<br>virtual memory                                      | file system<br>I/O systems            | memory management<br>.....            |
| Kernel interface to the hardware                                      |                                       |                                       |
| Terminal controllers<br>terminals                                     | device controllers<br>disks and tapes | memory controllers<br>physical memory |

# Programmer Interface

- System calls define the *programmer interface* to UNIX.
- Three main types of system call:
  - File manipulation
  - Process control
  - Information manipulation

# File Manipulation

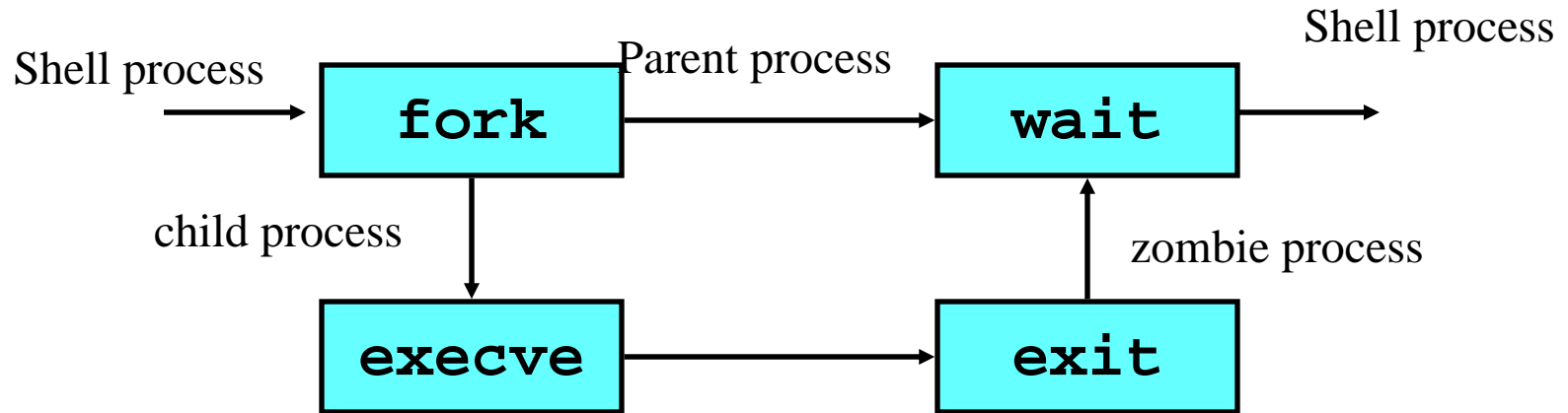
- Files in UNIX are sequences of bytes
  - Kernel doesn't impose a structure on files.
- Tree-structured directory



# File manipulation

- System calls
  - **creat, open, read, write, close, link, unlink, symlink, trunc, fcntl**
  - **mkdir, rmdir, opendir, readdir, closedir**
  - **stat, rename, chmod, chown, ioctl**

# Process Control



- **fork**: create a child process
- **execve**: load a new program into memory
- **exit**: complete execution
- **wait**: suspend execution until child calls **exit**

# Signals

- Signals are a facility for handling exceptional conditions.
- There are 20 signals, each responding to a distinct condition. This may be
  - a keyboard interrupt such as ^C.
  - an bad memory reference.
  - ..



# User Interface

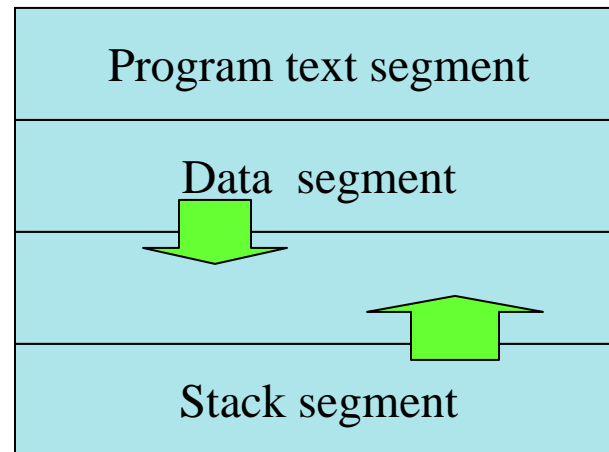
- Shell is a customized command line interpreter.
- When UNIX is booted, a process named *init* is active. *init* creates a logon process and opens stdin, stdout and stderr files to accept user input from the terminal and display output and error messages.
- In responding to a command, the shell forks a child process to carry out the command.

# Images and Processes (1)

- A user routine can be viewed as an *image* which contains:
  - program
  - process ID
  - scheduling information such as priority
  - data storage
  - the contents of registers
  - open files, current directory ...

# Images and Processes (2)

- An image consists of three segments: text segment, data segment, and stack segment.



- The execution of an image is a *process*.

---

# Process Management

- A process table contains an entry for each process. Each entry contains
  - process ID
  - process state
  - process priority
  - text table address
  - segment addresses ...
- The UNIX dispatcher uses this table to schedule processes.

# CPU Scheduling

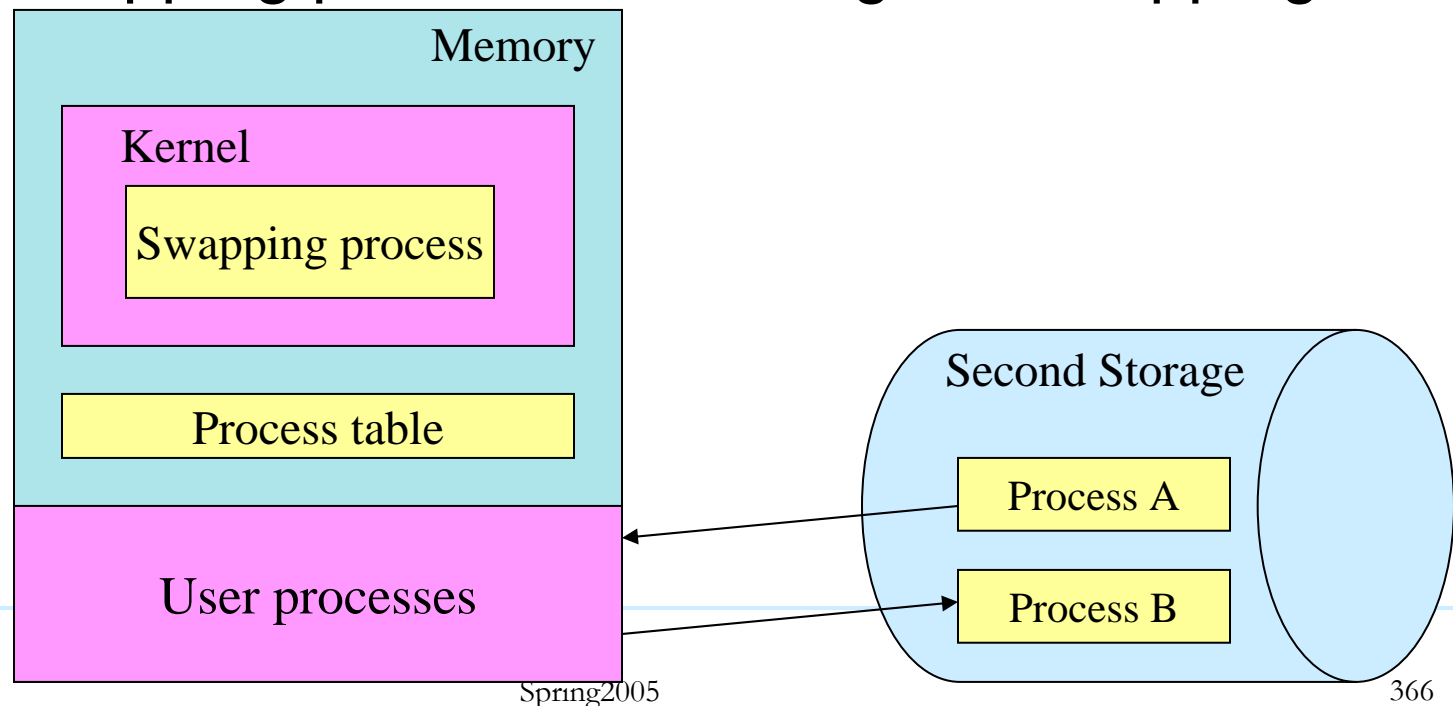
- Each process has a scheduling priority. A larger number indicates lower priority.
- The more CPU time a process accumulates, the more priority it gets, and vice versa.
- Round-Robin Scheduling.

# Memory Management

(1)

## ■ Swapping

- Some processes are swapped out if memory is not enough.
- A swapping process is in charge of swapping.



# Memory Management (2)

- Because several processes can physically share a text segment, the text segment cannot be released until all the processes using it have died.
- A text table is used to keep track of active text segments.

# File System

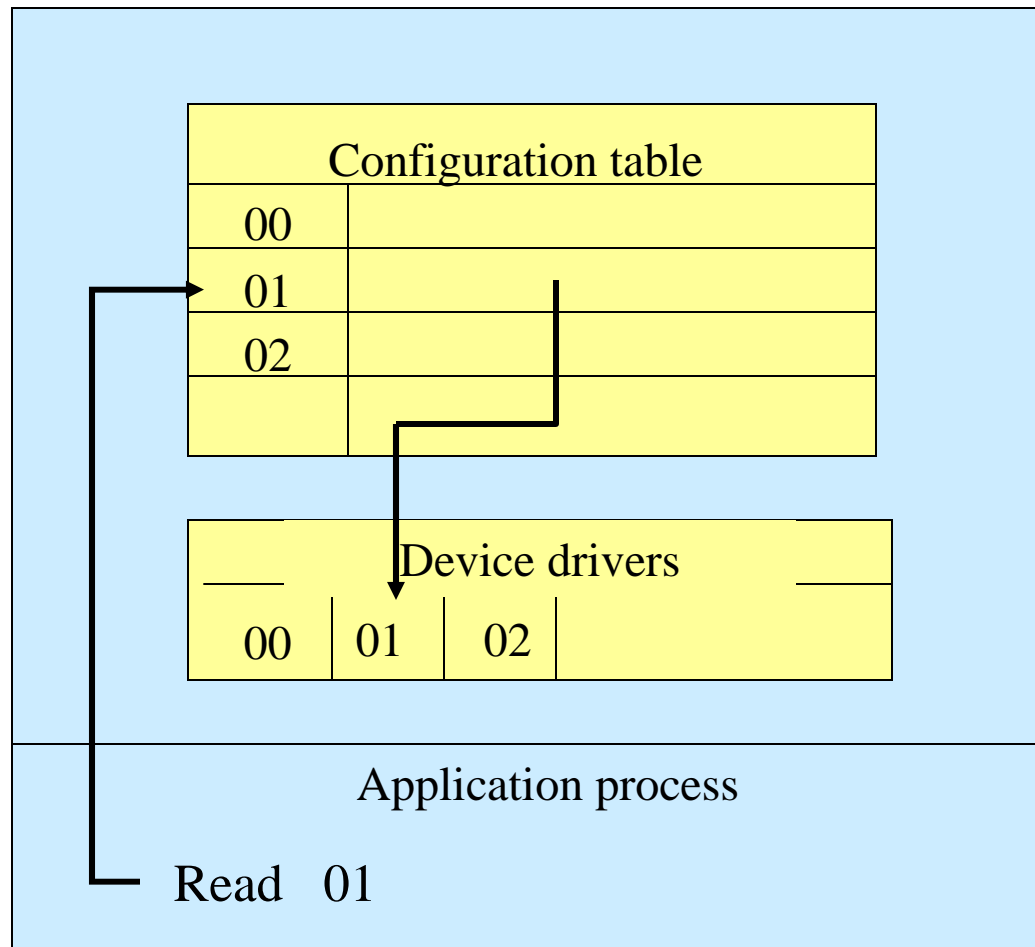
(1)

- All data are treated as a string of bytes and no physical structure is imposed by the system.
- This gives compatibility of file and I/O devices.
  - Ordinary files.
  - Special files.
    - Block or structured devices (normally a disk) hold files.
    - Character devices include printers, terminals and other nonblock peripherals.
- Each device has a device driver. All devices are listed in a configuration table and identified by a device number.



# File system

(2)



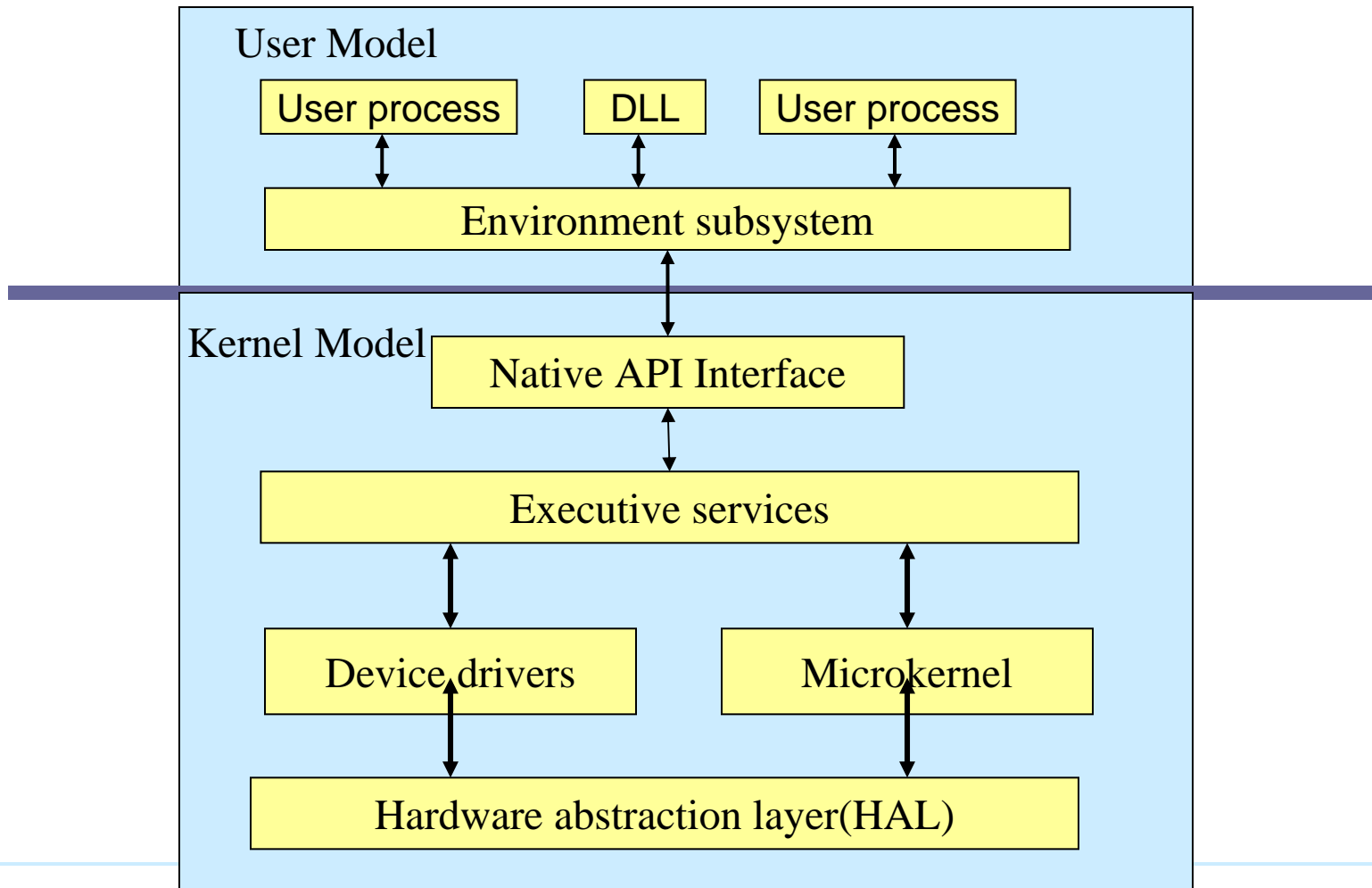
# Summary

- Layer structure.
- Programmer and User interface.
- Process management
  - Process image
  - CPU scheduling
- Memory management
  - swapping and segmentation
- File system
  - files
  - block and character devices

# Chapter 16

## Windows XP

# System Architecture



# Kernel Mode

- First layer
  - The **Hardware Abstraction Layer** hides the underlying hardware and provide a virtual machine interface to other processes.
- Second layer
  - **Device drivers** transfer logical I/O calls to specific physical I/O hardware primitives.
  - **Microkernel** coordinates dispatching and multi processor synchronization and handles asynchronous procedure calls.
- Top layer
  - **Executive services** such as I/O manager, security manager, power manager, process manager, object manager ...

---

# User Model

- Environment subsystems are user-mode processes interposed between executive services and the rest of user space. It exports an API for a specific environment.
  - Win32 subsystem
  - POSIX subsystem
  - 16-bit MS DOS
  - Win64 subsystem

# System Management Mechanisms

- How components and user processes store and access configuration data in the system
- How objects are implemented, managed and manipulated
- How interrupts are used

# Registry

- The registry is a database storing configuration information specific to users, applications and hardware.
- The registry is organized as a tree.
  - Each node in the tree represents a registry key.
  - A key stores subkeys and values
  - A value contains a value name and a value data.



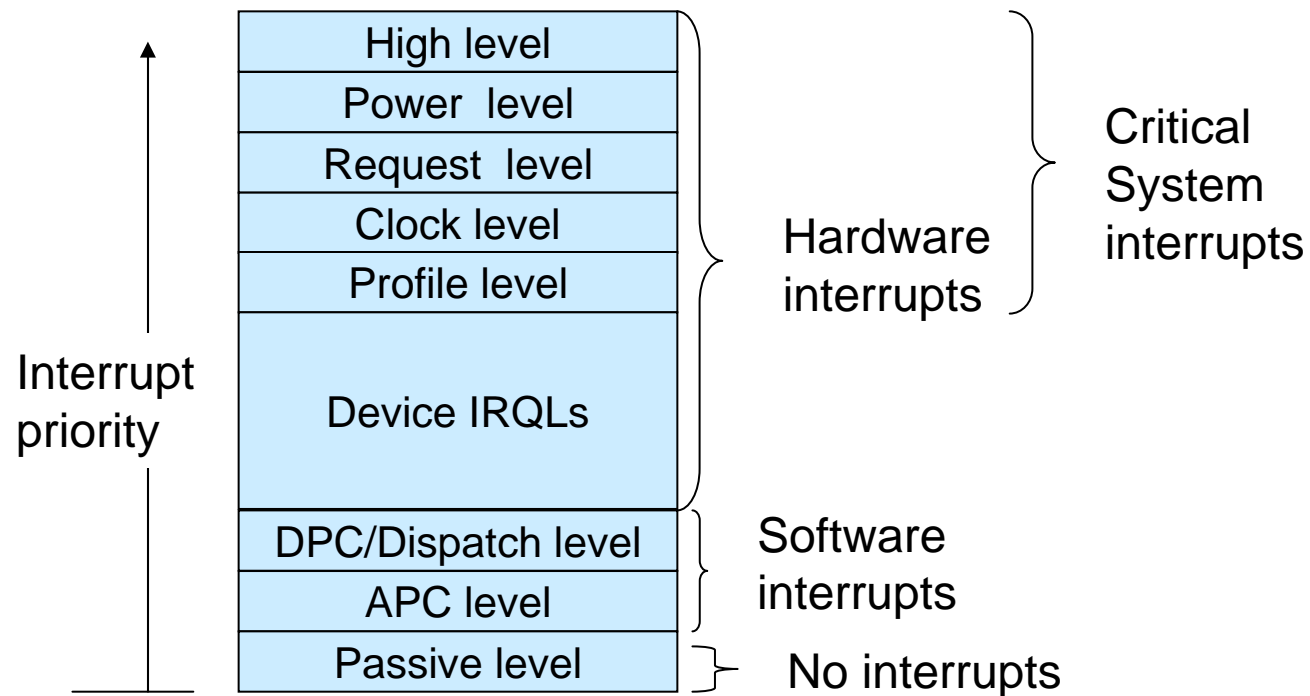
# Object Manager

- Windows XP represents physical resources and logical resources with objects.
  - The objects are constructs that Windows XP uses to represent resources( not the objects in OO programming).
  - Object types include files, devices, processes, threads, pipes, semaphores and many others.
  - The objects can be accessed through object handles.
- Object manager is responsible for creating, deleting objects and maintaining information about each object type.

# Interrupt Request Levels (IRQLs) (1)

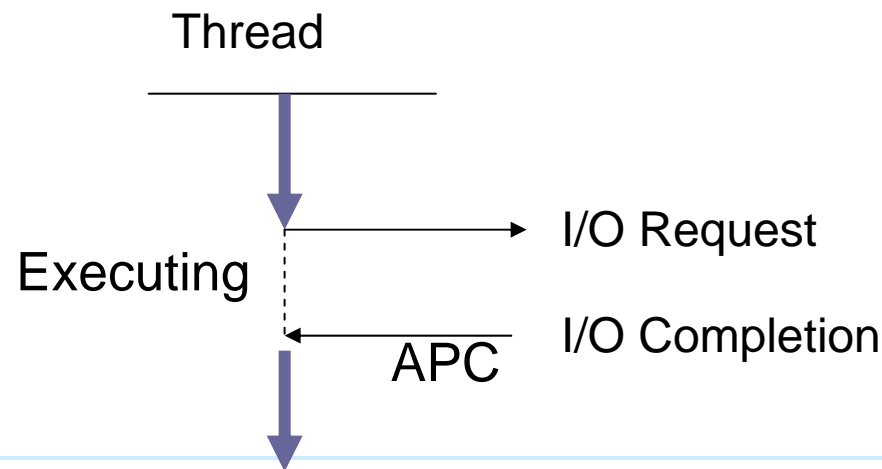
- IRQLs are a measure of interrupt priority.
  - An interrupt that executes at an IRQL higher than the current one can interrupt the current execution and obtain the processor.
  - Windows XP defines several IRQLs
    - Passive IRQL: threads
    - APC IRQL
    - DPC/dispatch IRQL
    - Device IRQLs
    - Profile IRQL
    - Clock IRQL
    - Request IRQL
    - Power IRQL
    - High IRQL

# Interrupt Request Levels (IRQLs) (2)



# Asynchronous Procedure Calls (APCs)

- APCs are procedure calls that threads or the system can queue for execution by a specific thread.
- This is used to complete various functions that must be done in a specific thread context.



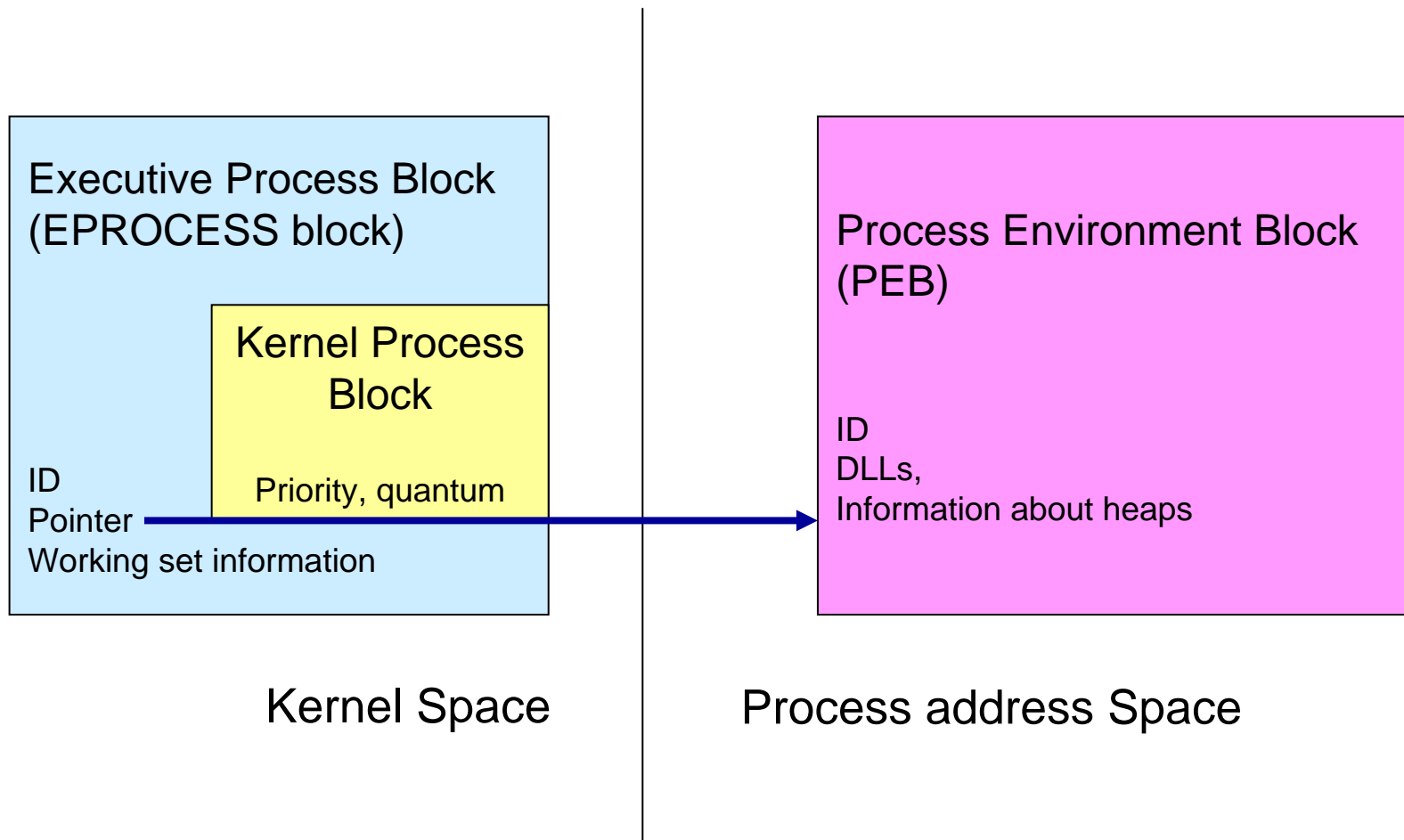
# Deferred Procedure Calls (DPCs)

- DPCs are software interrupts which run in the context of the currently executing thread.
  - Hardware interrupts are processed by calling the associated interrupt service routine, which is responsible for quickly acknowledging the interrupt and queuing a DPC for later execution.

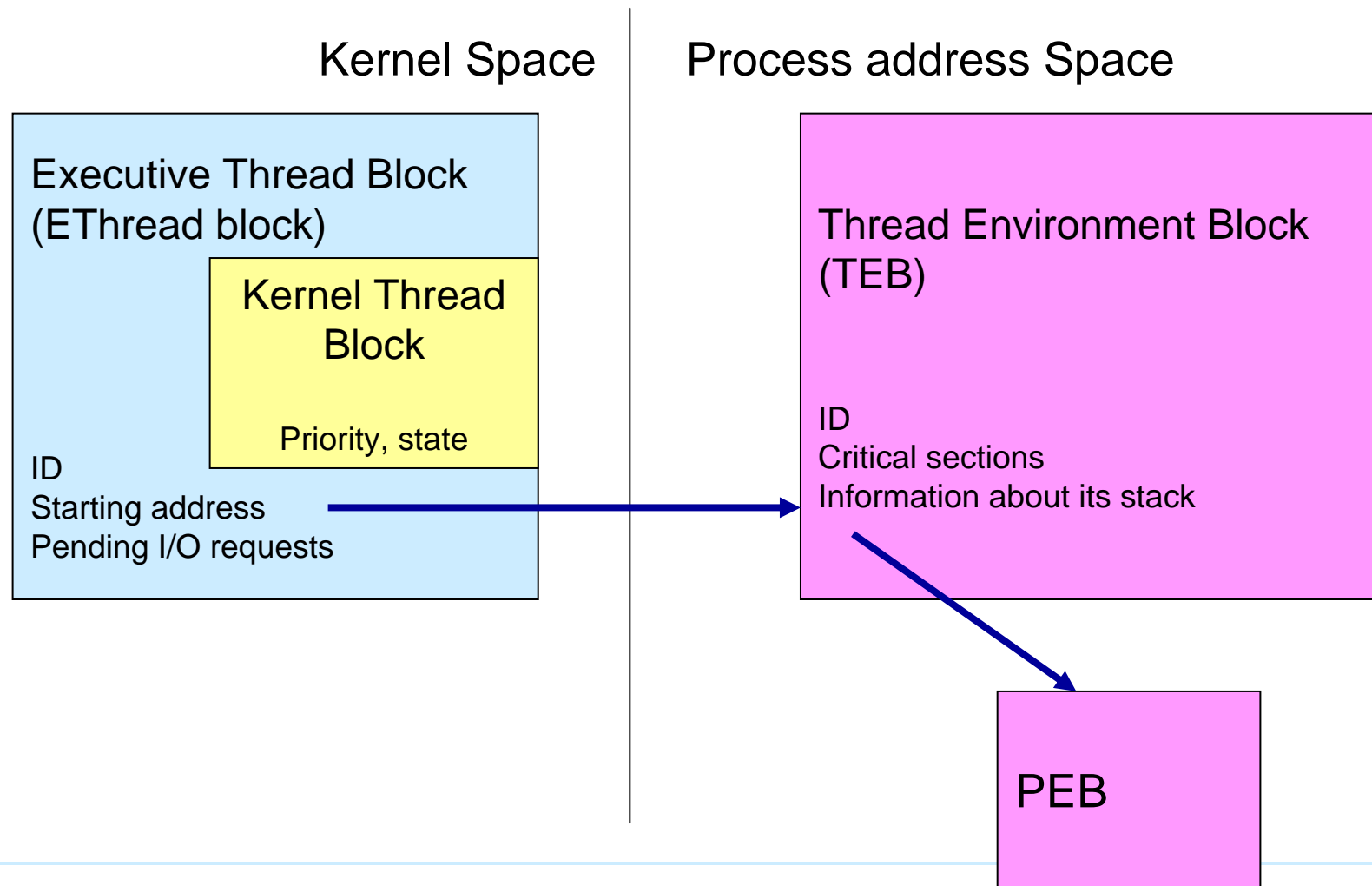
# System Threads

- There are two kinds of system threads:
  - Kernel thread – a thread which is created by a kernel component, executes in kernel mode and typically belongs to a system process.
  - System worker thread – a thread which is created at system initialization time or dynamically in response to a high volume of requests and sleeps until a work item is received. It also belongs to a system process.
    - Windows XP includes three types of worker threads: delayed, critical and hypercritical.

# Process Organization



# Thread Organization



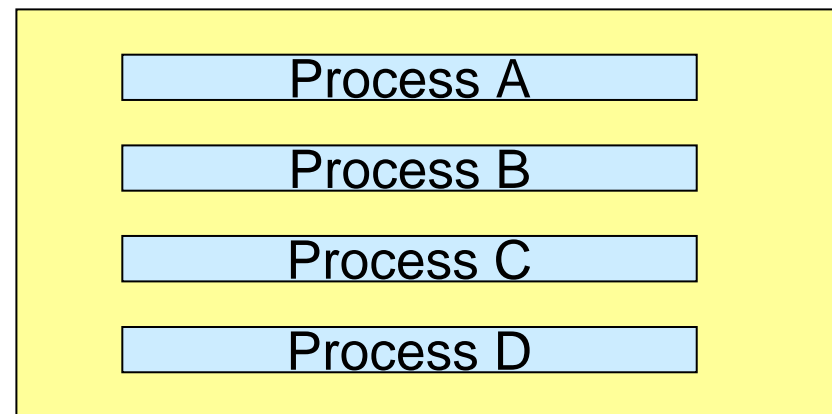


# Creating and Terminating Processes

- The parent and child processes are completely independent. (The child process receives a new address space when it is created)
- A primary thread is created when a process is initiated by the system.
- A process terminates when
  - All of its threads terminate
  - Any of its thread explicitly terminate the process
  - Its user logs off

# Jobs

- Several processes can be grouped together into a unit, called a job.
  - It allows developer to define rules and set limits on a number of processes.
  - It can be easily terminated



Job

Spring2005

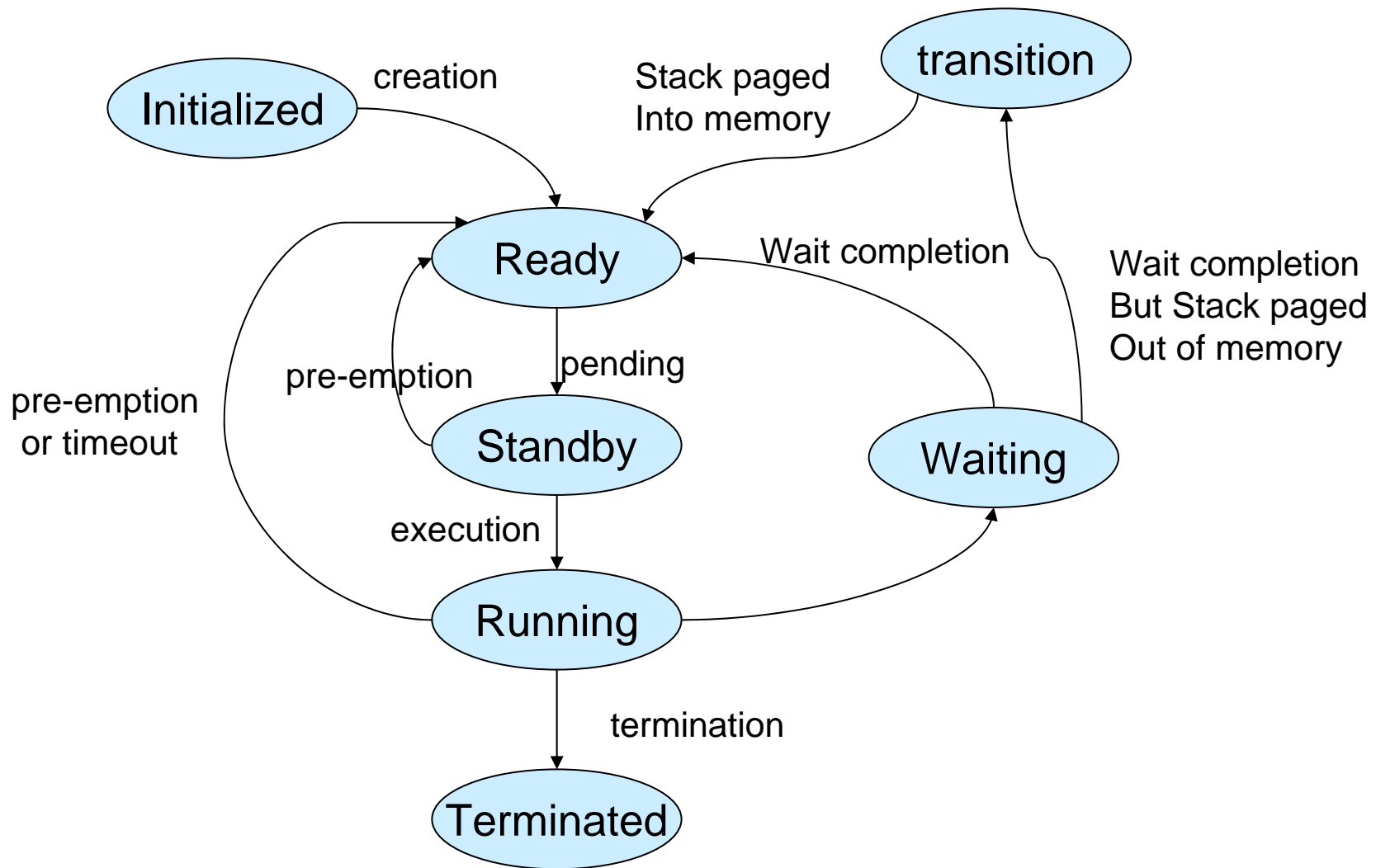
# Fibers

- Fibers are used to port code to and from other operating systems.
- A fiber is created by a thread. It is similar to a thread, except that it is scheduled for execution by the thread that created it, rather than the microkernel.
- Windows XP threads are implemented with a one-to-one mapping. Fibers allows Windows XP applications to write code executed using the many-to-many mapping.

# Thread Pools

- Each process has a thread pool that consists of a number of worker threads which execute functions queued by user threads.
  - The worker thread sleeps until a request is queued to the pool.
  - The thread that queues the request must specify the function to execute and must provide context information.

# Thread States



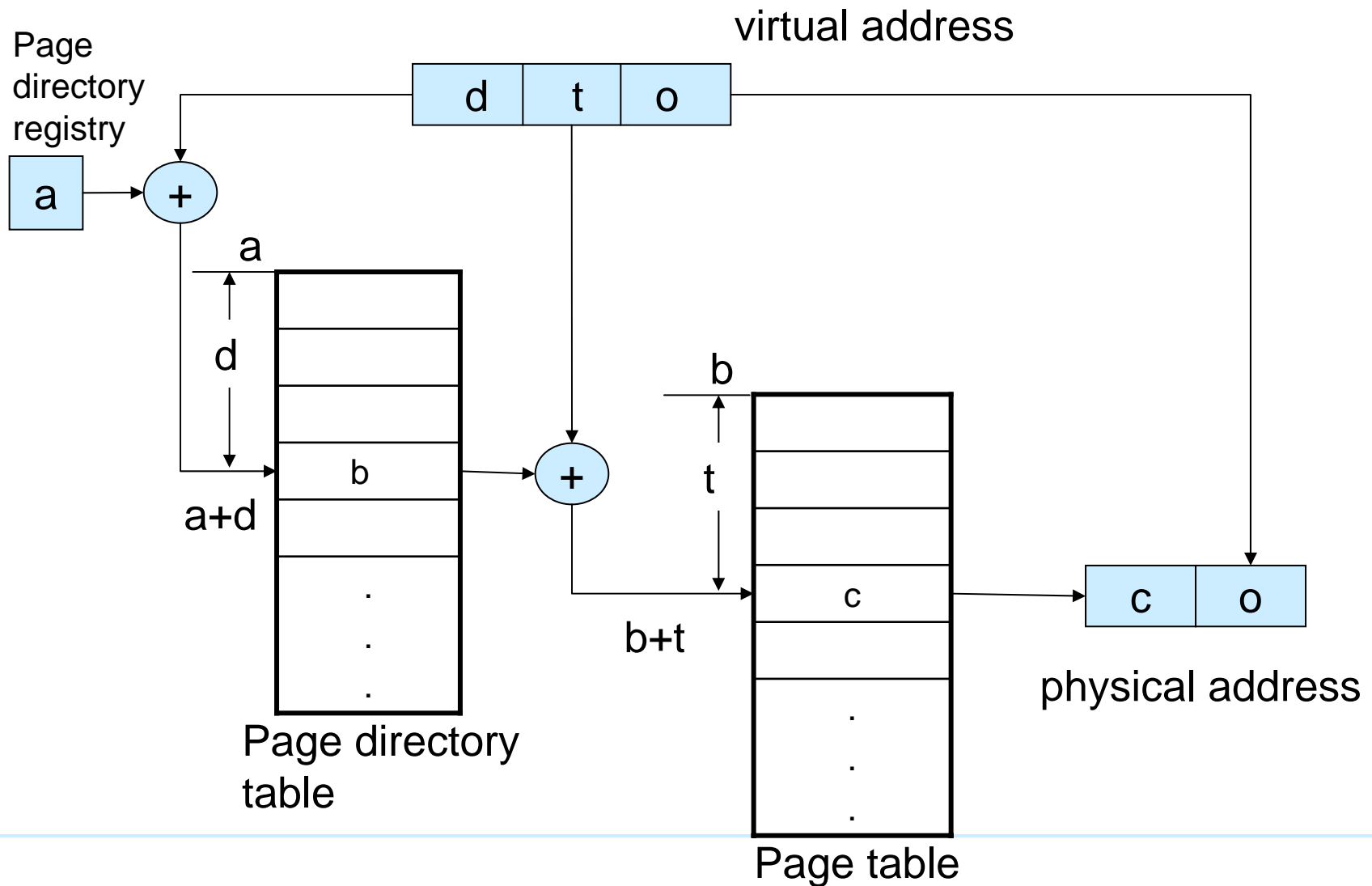
# Thread Scheduling Algorithm

- This is based on priority
  - Multi-level queue scheduling
  - Windows XP has 32 priority levels: 0-31.
  - A thread is placed into the ready queue corresponding to its priority.

# Memory Management

- Use 32-bit memory model. The biggest possible address is 4 GB.
- 4 GB of virtual address space is available to each process. Top 2GB for kernel mode threads and the bottom 2 GB for user mode threads.
- A Virtual Memory Manager (VMM) manages memory. It handles paging.

# Virtual Address Translation



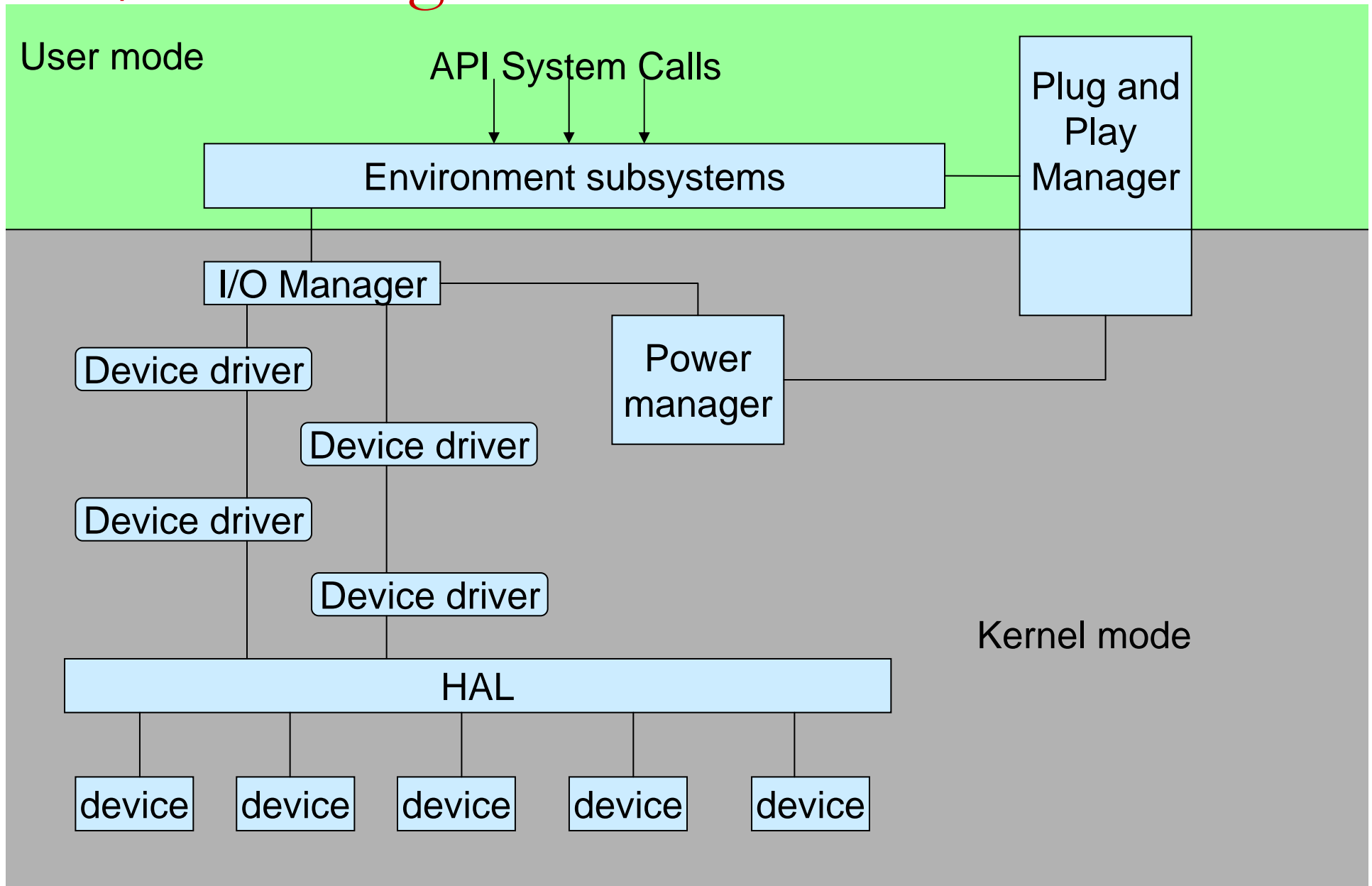


---

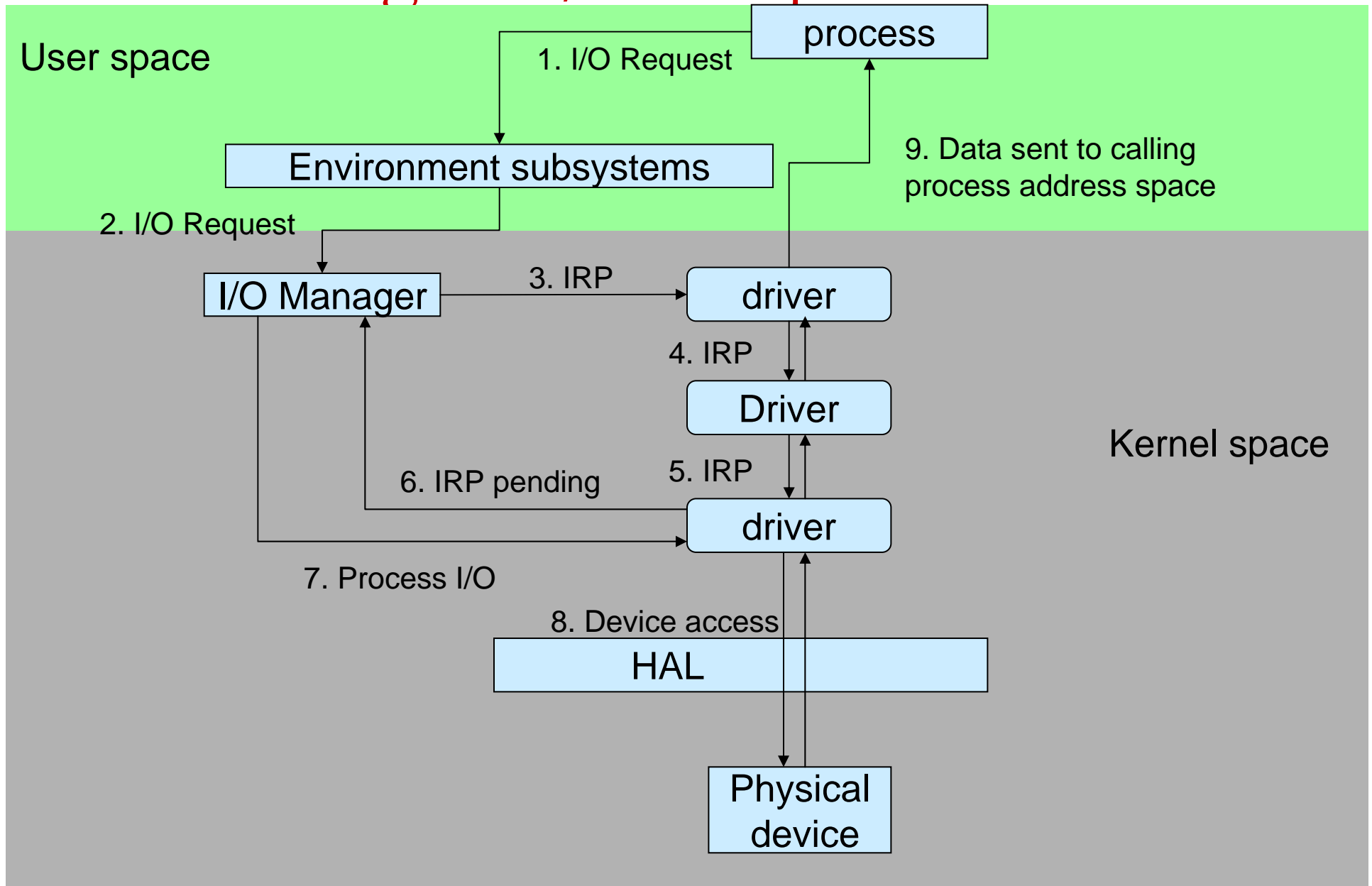
# File Management

- Supports
  - FAT, used in MS-DOS file system.
  - FAT32, allows long file names and larger disk drive.
  - NTFS, with more features such as file, folder-level security, encryption, compression and so on.

# I/O Management



# Processing an I/O Request



# NTFS

- NTFS works with disk volumes. Volumes are allocated in clusters. Logical cluster number is used as logical disk address.
- Each volume has 4 sections
  - Boot section
  - Master file table (MFT) section
  - System files
  - File area

---

# More about Windows XP

- Interprocess communication
- Component Object Model (COM)
- Networking
- .Net

---

# Summary

- System management mechanisms
  - Registry
  - Object manager
  - Interrupt request levels
  - Asynchronous and deferred procedure calls
- Process and thread management
- Memory management
- File system management
- Input/output management