Operating Systems: Internals and Design Principles

Chapter 2 Operating System Overview

Seventh Edition

By William Stallings

Operating Systems: Internals and Design Principles

Operating systems are those programs that interface the machine with the applications programs. The main function of these systems is to dynamically allocate the shared system resources to the executing programs. As such, research in this area is clearly concerned with the management and scheduling of memory, processes, and other devices. But the interface with adjacent levels continues to shift with time. Functions that were originally part of the operating system have migrated to the hardware. On the other side, programmed functions extraneous to the problems being solved by the application programs are included in the operating system.

—WHAT CAN BE AUTOMATED?: THE COMPUTER SCIENCE AND ENGINEERING RESEARCH STUDY, MIT Press, 1980

Operating System

- An interface between applications and computer
- A program that controls the execution of application programs and the allocation of system resources

Main objectives of an OS:

- Convenience
- Efficiency
- Ability to evolve

a User/Computer Interface

- The OS provides abstractions of the computer hardware, making it more convenient for applications to use the computer's capabilities
- It does this through a set of interfaces and services.

Computer Hardware and Software Infrastructure

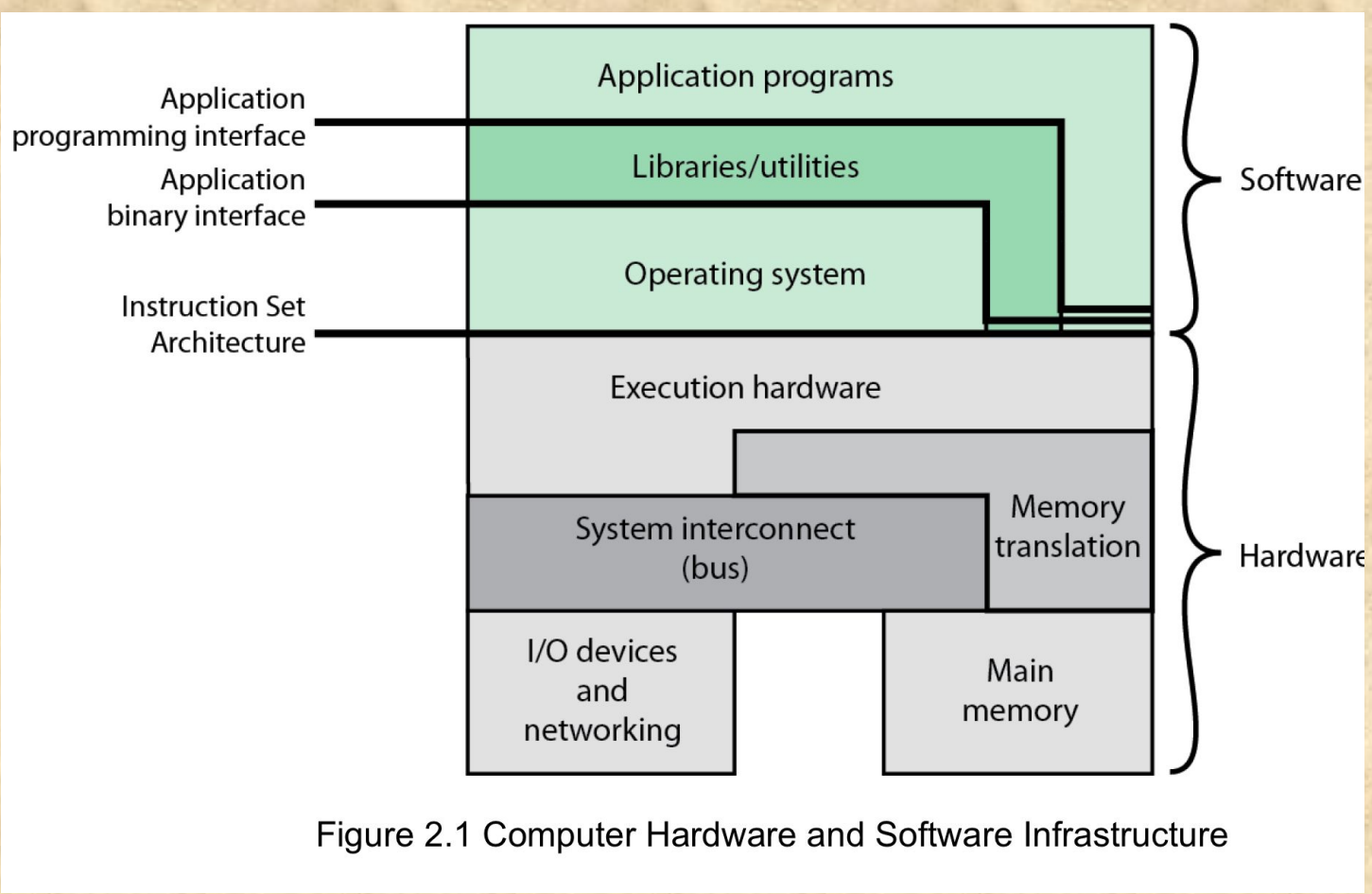


Figure 2.1 Computer Hardware and Software Infrastructure

Key Interfaces

- Instruction set architecture (ISA)
- Application binary interface (ABI)
  supports portability of applications in binary forms across different system platforms and environments
- Application programming interface (API) applications can interface to OS through system calls

Operating System Services

- Program development utilities – not strictly OS
- Program execution
- Access I/O devices
- Controlled access to files
- System access
- Error detection and response
- Accounting

Efficiency:The Operating System As a Resource Manager

- A computer is a set of resources for moving, storing, & processing data
- The OS is responsible for managing these resources
- The OS exercises its control through software

Operating System as Software

- Functions in the same way as ordinary computer software
- Program, or suite of programs, executed by the processor
- Frequently relinquishes control and must be able to regain control to decide on the next thing the processor should do.

Operating System as Resource Manager

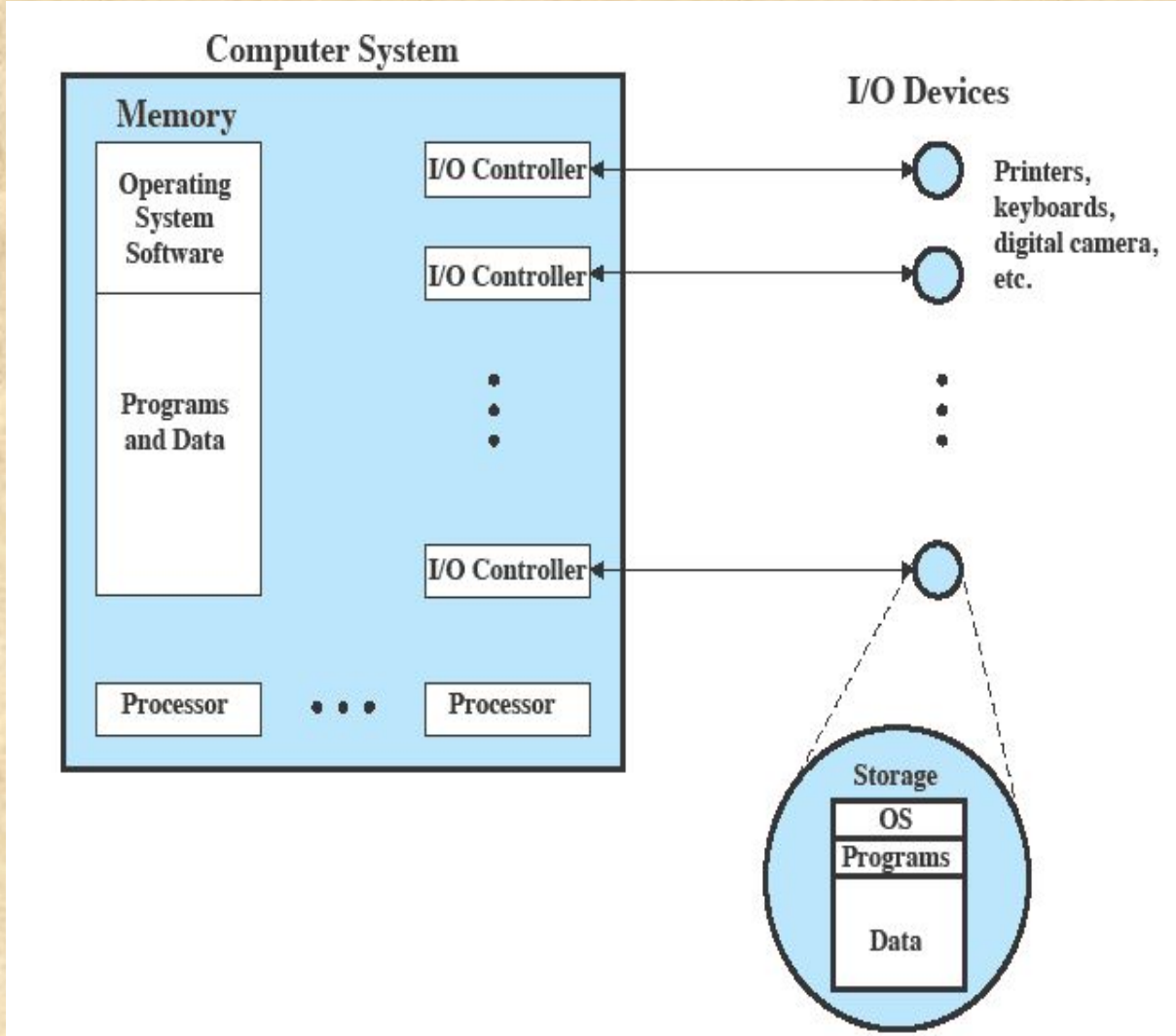


Figure 2.2 The Operating System as Resource Manager

Evolution of Operating Systems

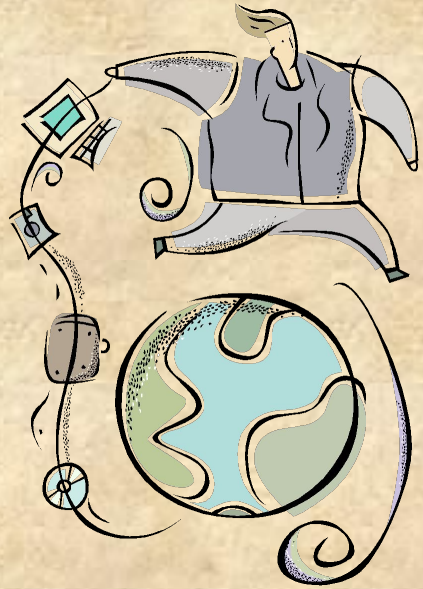
- A major OS will evolve over time for a number of reasons:

Hardware upgrades

New types of hardware

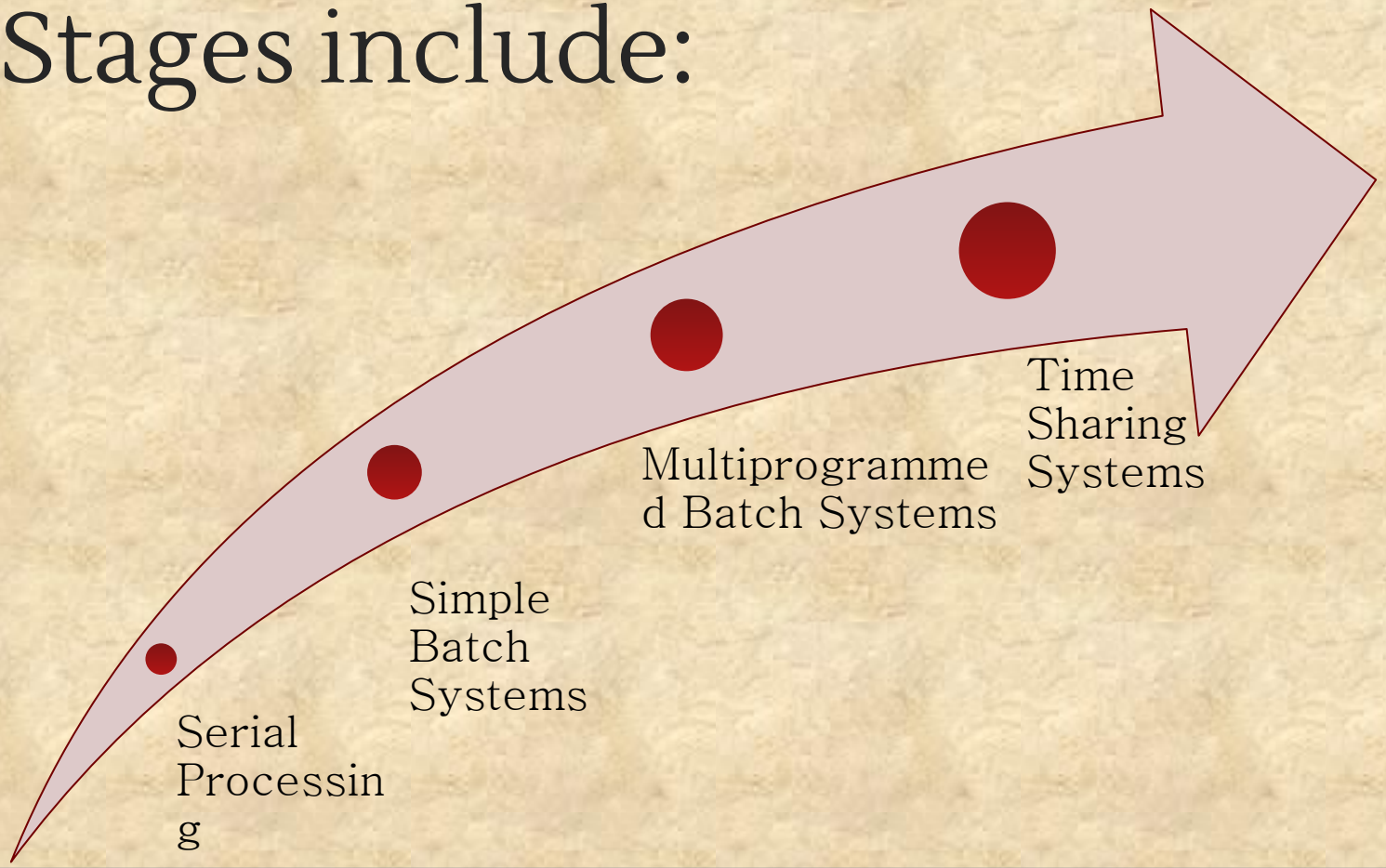
New services

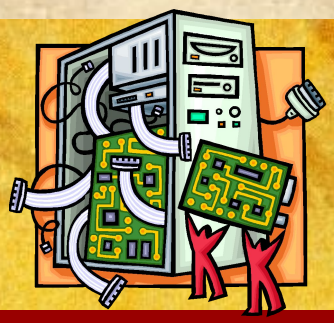
Fixes



Evolution of Operating Systems

- Stages include:





Serial Processing

Earliest Computers:

- No operating system
  - programmers interacted directly with the computer hardware
- Computers ran from a console with display lights, toggle switches, some form of input device, and a printer
- Users had access to the computer in “series”

Problems:

- Scheduling:
  - most installations used a hardcopy sign-up sheet to reserve computer time
    - time allocations could run short or long, resulting in wasted computer time
- Setup time
  - a considerable amount of time was spent just on setting up the program to

Simple Batch Systems

- Early computers were very expensive
  - important to maximize processor utilization
- Monitor (primitive operating system)
  - user no longer has direct access to processor
  - job is submitted to computer operator who batches them together and places them on an input device

Monitor Point of View

- Monitor controls the sequence of events
- *Resident Monitor* is software always in memory
- Monitor reads in job and gives control
- Job returns control to monitor

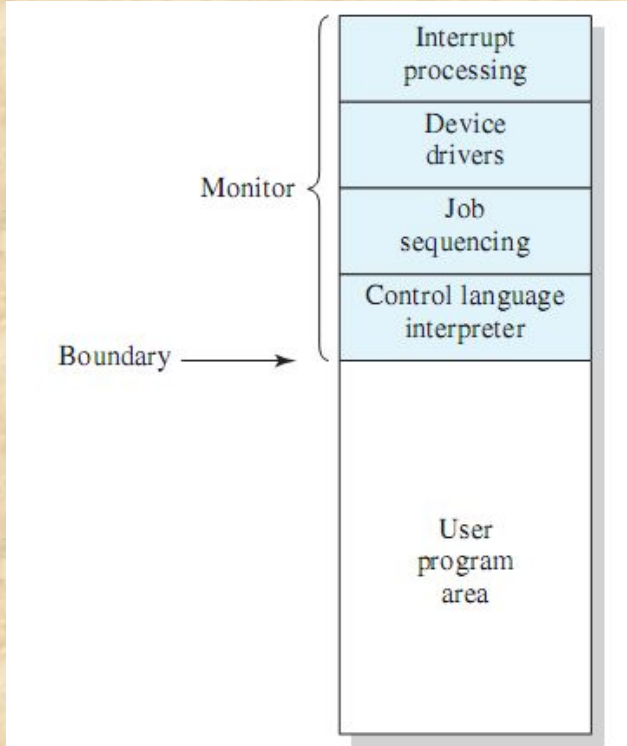


Figure 2.3 **Memory Layout for a Resident Monitor**

Processor Point of View

- Processor executes instruction from the memory containing the monitor
- Executes the instructions in the user program until it encounters an ending or error condition
- "*control is passed to a job*" means processor is fetching and executing instructions in a user program
- "*control is returned to the monitor*" means that the processor is fetching and executing instructions from the monitor program

Job Control Language (JCL)

Special type of programming language used to provide instructions to the monitor

what compiler to use

what data to use

Desirable Hardware Features

Memory protection for monitor

- while the user program is executing, it must not alter the memory area containing the monitor

Timer

- prevents a job from monopolizing the system

Privileged instructions

- can only be executed by the monitor

Interrupts

- gives OS more flexibility in controlling user programs

Modes of Operation

User Mode

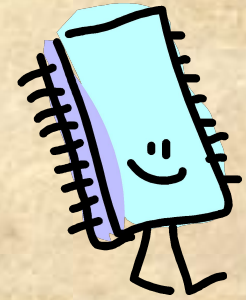
- user program executes in user mode
- certain areas of memory are protected from user access
- certain instructions may not be executed

Kernel Mode

- monitor executes in kernel mode
- privileged instructions may be executed
- protected areas of memory may be accessed

Simple Batch System Overhead

- Processor time alternates between execution of user programs and execution of the monitor
- Sacrifices:
  - some main memory is now given over to the monitor
  - some processor time is consumed by the monitor
- Despite overhead, the simple batch system improves utilization of the computer.



Multiprogrammed Batch Systems

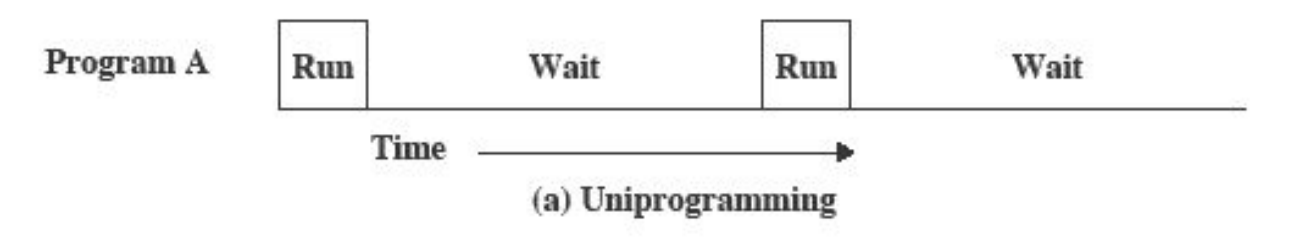
Read one record from file	15 μs
Execute 100 instructions	1 μs
Write one record to file	15 μs
TOTAL	31 μs

Percent CPU Utilization $= \frac{1}{31} = 0.032 = 3.2\%$

Figure 2.4 System Utilization Example

- Processor is often idle
  - even with automatic job sequencing
  - I/O devices are slow compared to processor

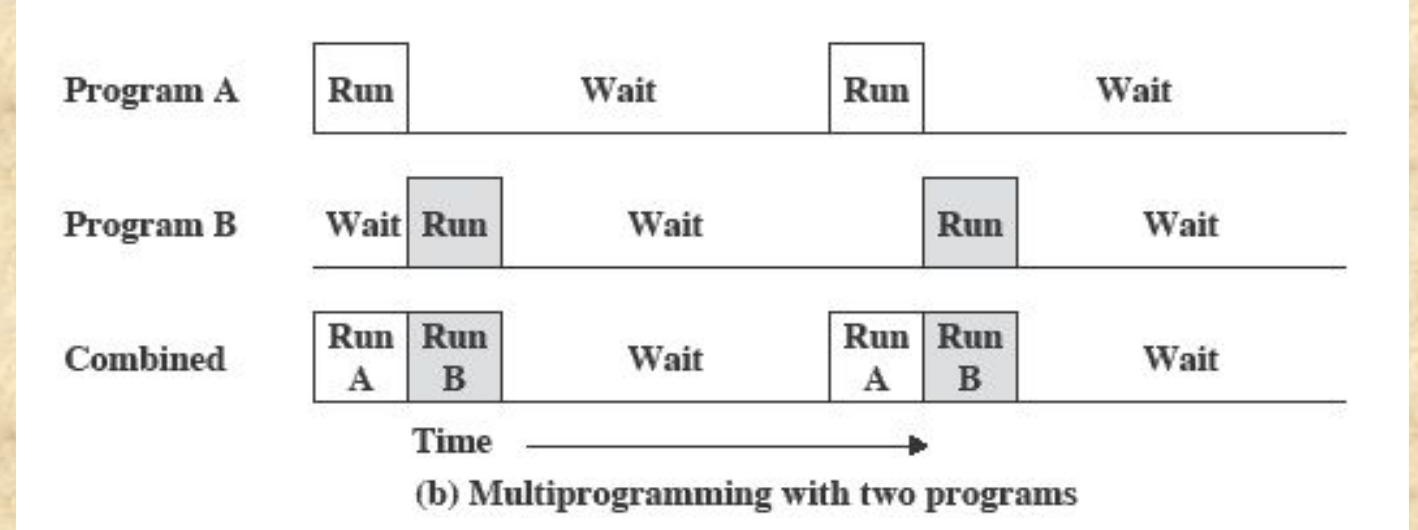
Uniprogramming



(a) Uniprogramming

- The processor spends a certain amount of time executing, until it reaches an I/O instruction; it must then wait until that I/O instruction concludes before proceeding

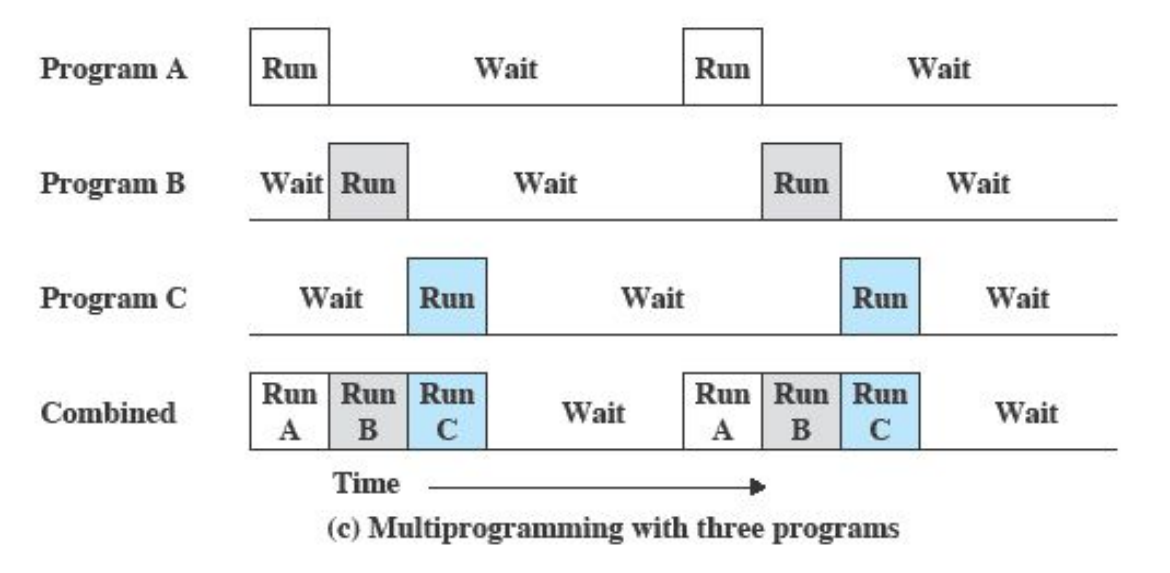
Multiprogramming



(b) Multiprogramming with two programs

- What if there's enough memory to hold the OS (resident monitor) and *two* user programs.
- When one job needs to wait for I/O, the processor can switch to the other job, which may not be waiting.

Multiprogramming



(c) Multiprogramming with three programs

- Multiprogramming
  - also known as multitasking
  - memory is expanded to hold three, four, or more programs and switch among all of them

Multiprogramming Example

Table 2.1 Sample Program Execution Attributes

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 M	100 M	75 M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

Effects on Resource Utilization

	Uniprogramming	Multiprogramming
Processor use	20%	40%
Memory use	33%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput	6 jobs/hr	12 jobs/hr
Mean response time	18 min	10 min

Table 2.2 Effects of Multiprogramming on Resource Utilization

Utilization Histograms

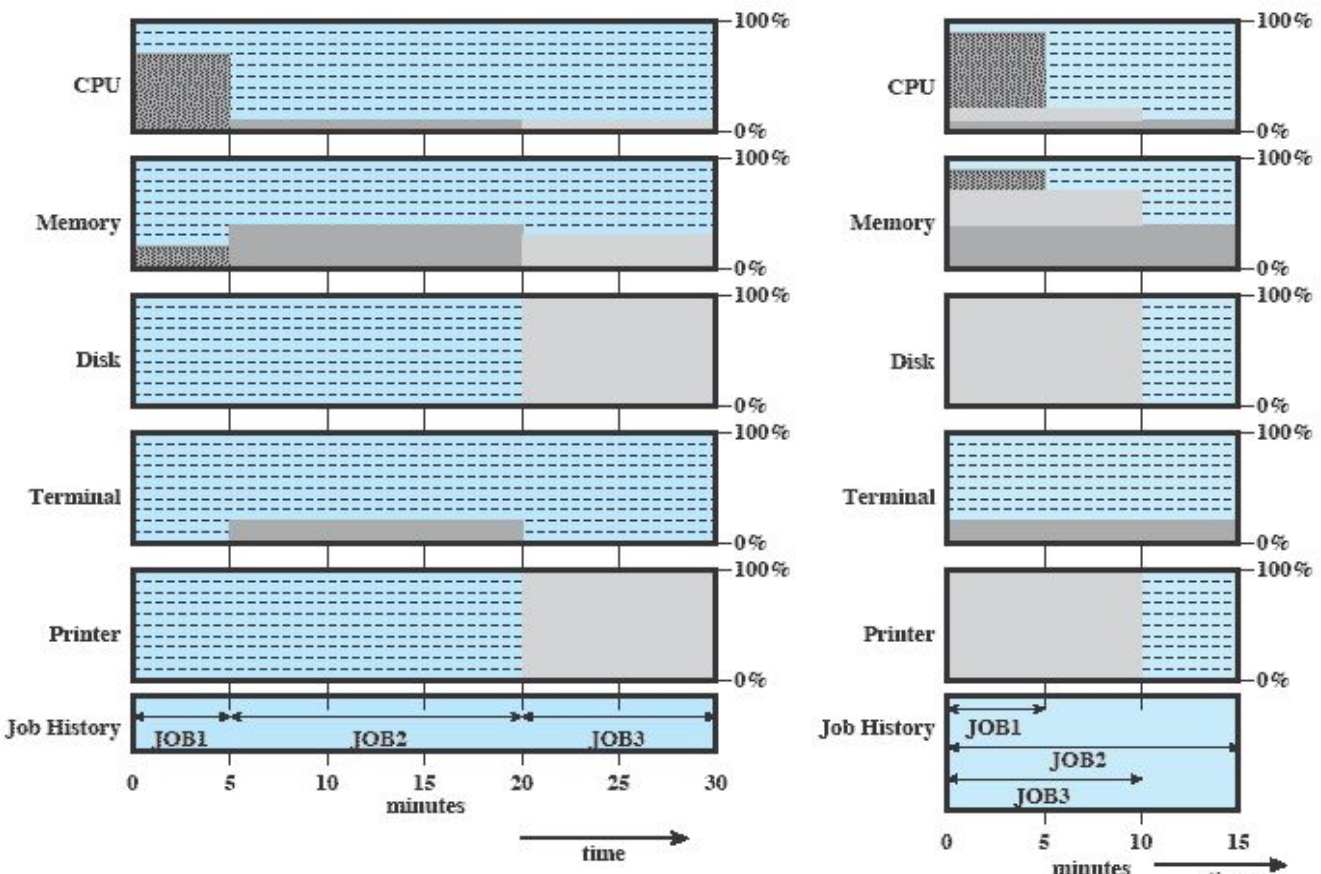


Figure 2.6 Utilization Histograms

Time-Sharing Systems

- Can be used to handle multiple *interactive* jobs
- Processor time is shared among multiple users
- Origin: multiple users simultaneously access the system through terminals, with the OS interleaving the execution of each user program in a short burst or quantum of computation

Batch Multiprogramming vs. Time Sharing

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

Table 2.3 Batch Multiprogramming versus Time Sharing

Compatible Time-Sharing Systems

CTSS

- One of the first time-sharing operating systems
- Developed at MIT by a group known as Project MAC for IBM 709/7094
- Ran on a computer with 32,000 *36*-bit words of main memory, with the resident monitor consuming 5000 of that
- To simplify both the monitor and memory management a program was always loaded to start at the location of the 5000th word

Time Slicing

- System clock generates interrupts at a rate of approximately one every 0.2 seconds
- At each interrupt OS regained control and could assign processor to another user
- At regular time intervals the current user would be preempted and another user loaded in
- Old user programs and data were written out to disk
- Old user program code and data were restored in main memory when that program was next

CTSS Operation

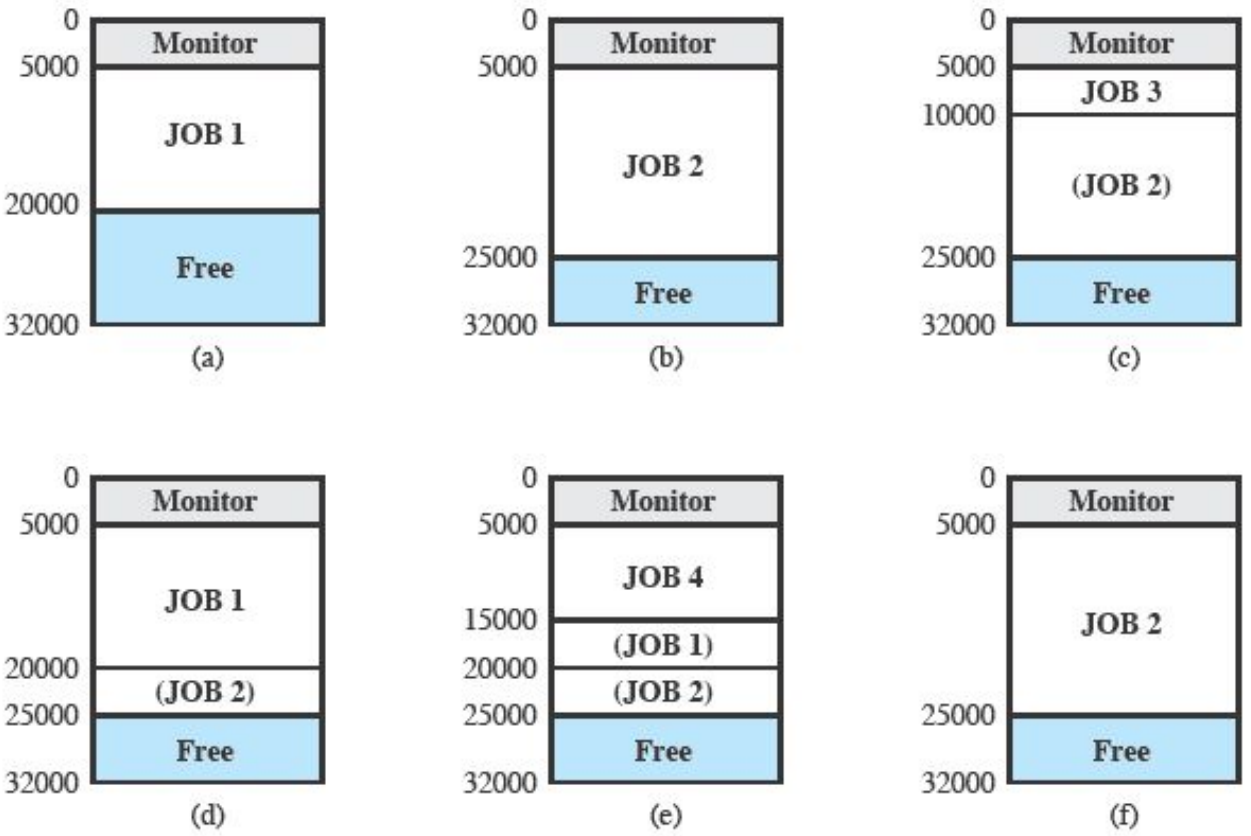


Figure 2.7 CTSS Operation

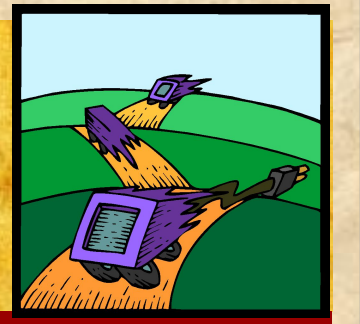
Major Advances

- Operating Systems are among the most complex pieces of software ever developed

Major advances in development include:

- Processes
- Memory management
- Information protection and security
- Scheduling and resource management
- System structure

Process



- Fundamental to the structure of operating systems

A *process* can be defined as:

a program in execution

an instance of a running program

the entity that can be assigned to, and executed on, a processor

a unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources

Development of the Process

- Three major lines of computer system development created problems in timing and synchronization that contributed to the development:

multiprogramming batch operation

- processor is switched among the various programs residing in main memory

time sharing

- be responsive to the individual user but be able to support many users simultaneously

real-time transaction systems

- a number of users are entering queries or updates against a database

Causes of Errors

- **Improper synchronization**
  - a program must wait until the data are available in a buffer
  - improper design of the signaling mechanism can result in loss or duplication
- **Failed mutual exclusion**
  - more than one user or program attempts to make use of a shared resource at the same time
  - only one routine at at time allowed to perform an update against a given file

- **Nondeterminate program operation**
  - program execution is interleaved by the processor when memory is shared
  - the order in which programs are scheduled may affect their outcome
- **Deadlocks**
  - it is possible for two or more programs to be hung up waiting for each other
  - may depend on the chance timing of resource allocation and release

Components of a Process

- A process contains three components:
  - an executable program
  - the associated data needed by the program (variables, work space, buffers, etc.)
  - the execution contex (or “process state”) the program

- The execution context is essential:
  - it is the internal data by which the OS is able to supervise and control the process
  - includes the contents of the various process registers
  - includes information such as the priority of the process and whether the process is waiting for the completion of a particular I/O event

Process Management

- The entire state of the process at any instant is contained in its context
- New features can be designed and incorporated into the OS by expanding the context to include any new information needed to support the feature

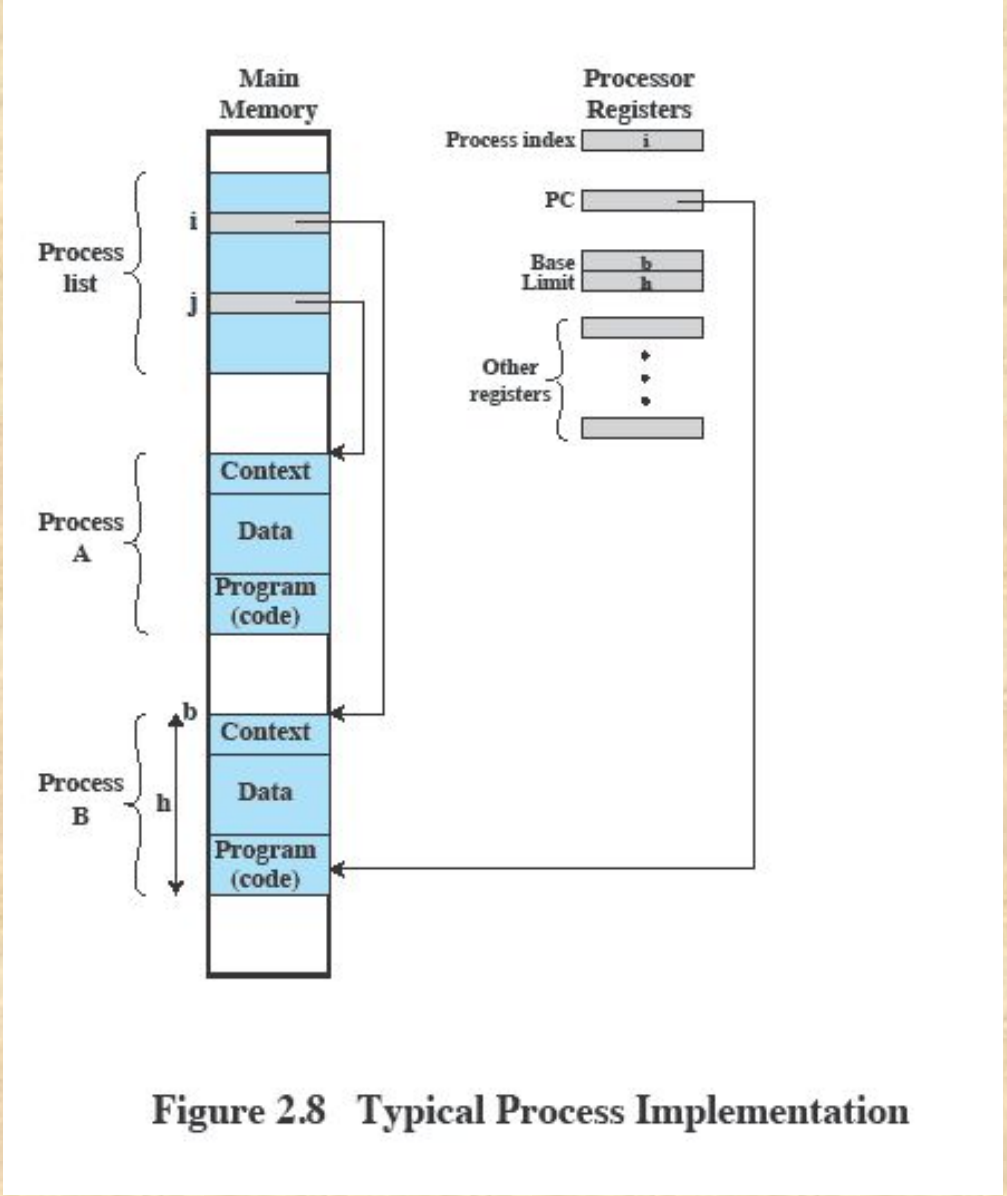
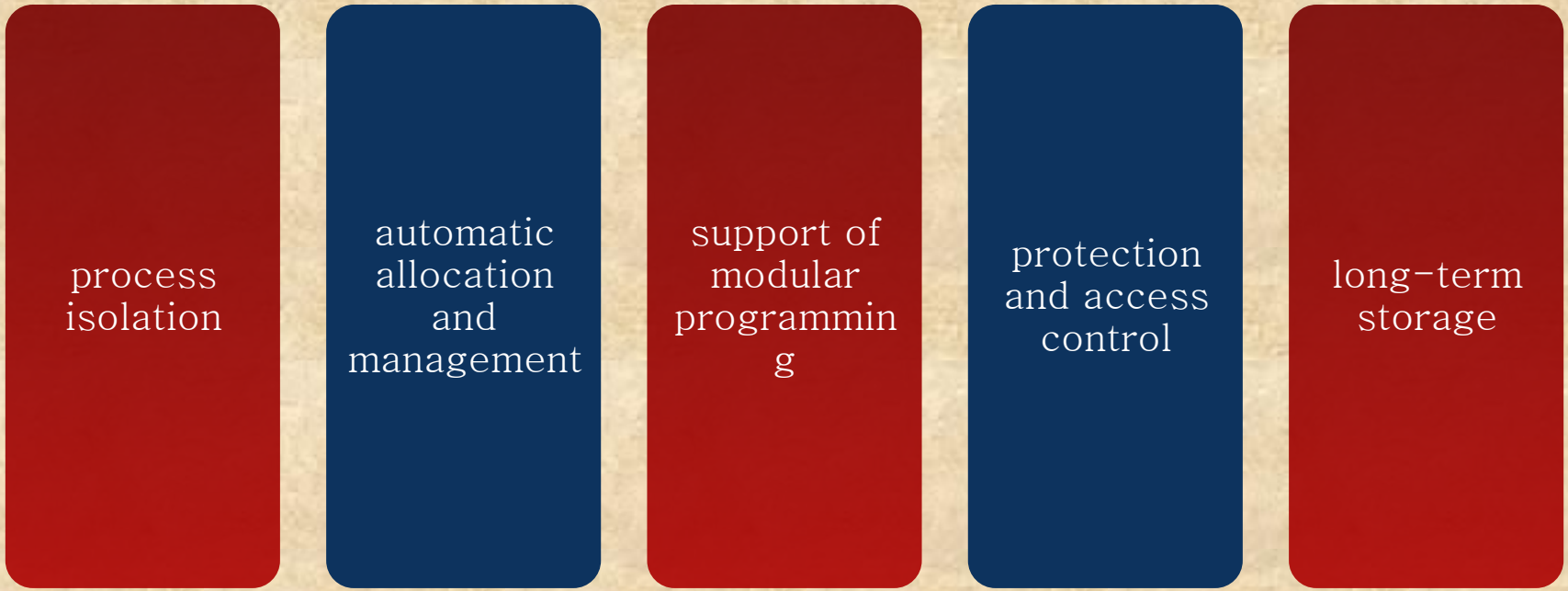


Figure 2.8 Typical Process Implementation

Memory Management

- The OS has five principal storage management responsibilities:



Virtual Memory

- A facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available
- Conceived to meet the requirement of having multiple user jobs reside in main memory concurrently

Paging

- Allows processes to be comprised of a number of fixed-size blocks, called pages
- Program references a word by means of a virtual address
  - consists of a page number and an offset within the page
  - each page may be located anywhere in main memory
- Provides for a dynamic mapping between the virtual address used in the program and a real (or physical) address in main memory

Virtual Memory

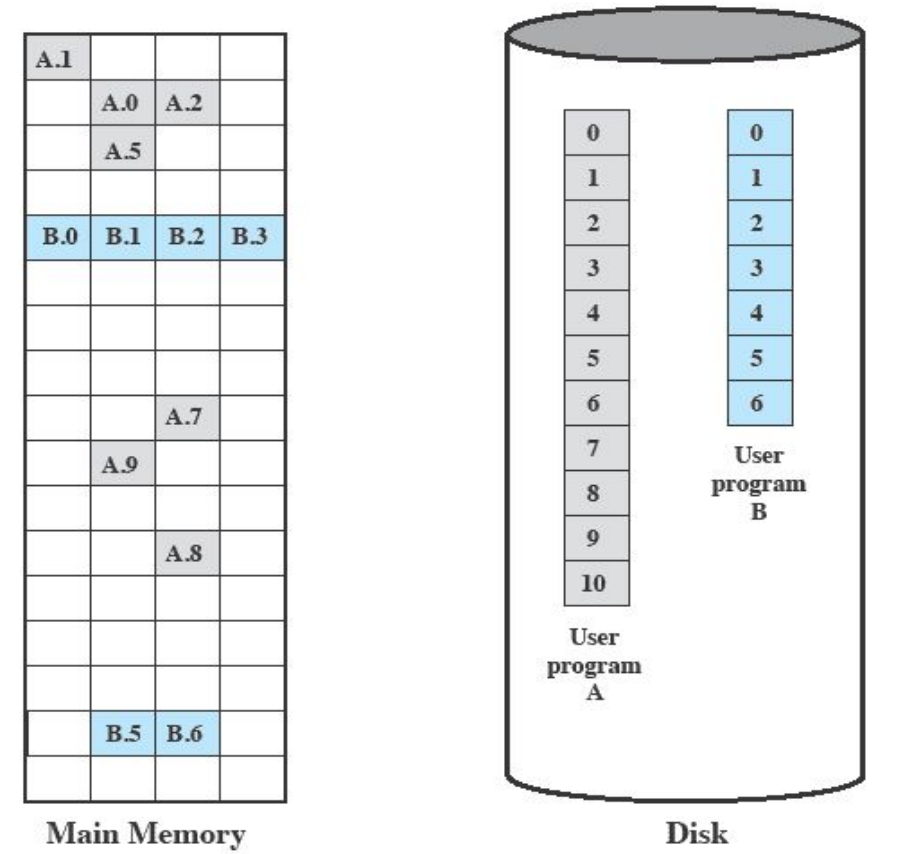


Figure 2.9 Virtual Memory Concepts

Virtual Memory Addressing

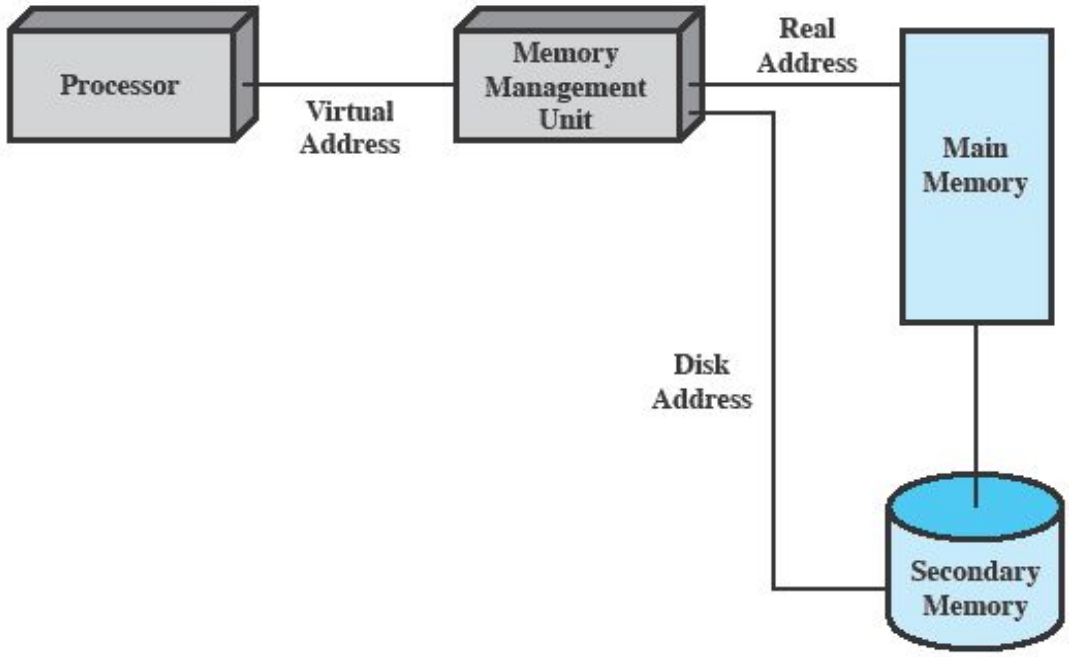
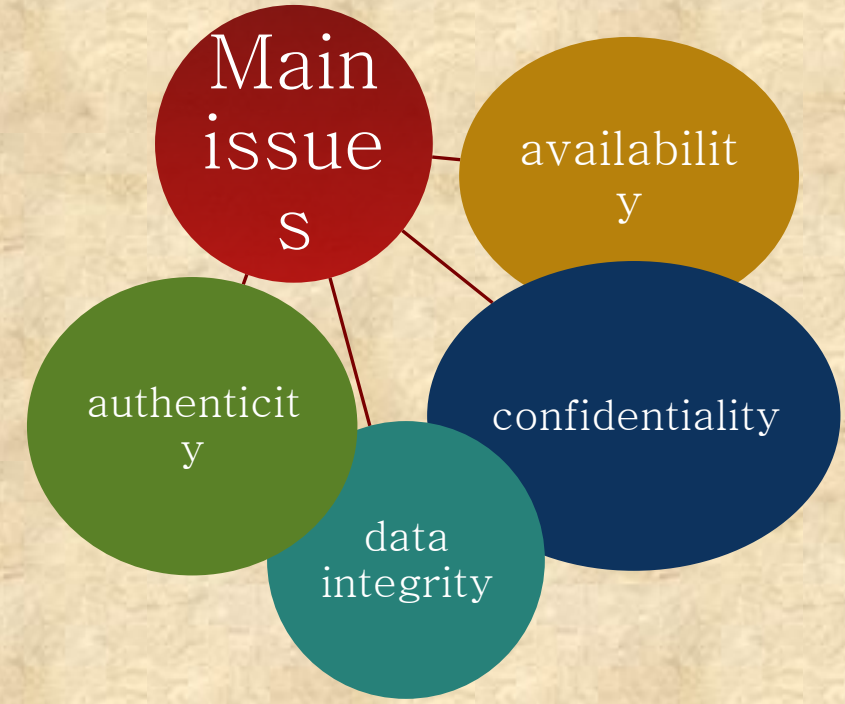


Figure 2.10 Virtual Memory Addressing

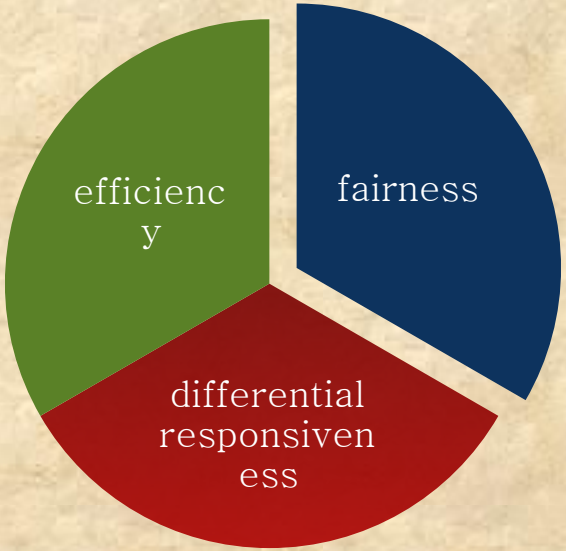
Information Protection and Security

- The nature of the threat that concerns an organization will vary greatly depending on the circumstances
- The problem involves controlling access to computer systems and the information stored in them



Scheduling and Resource Management

- Key responsibility of an OS is managing resources
- Resource allocation policies must consider:



Key Elements of an Operating System

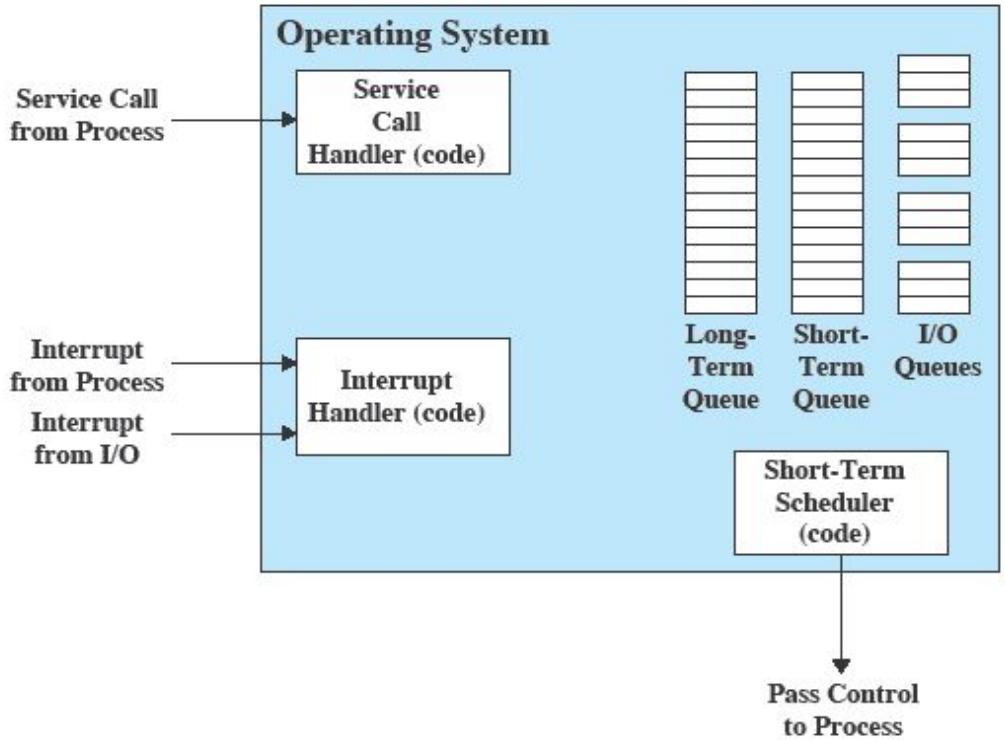


Figure 2.11 Key Elements of an Operating System for Multiprogramming

Processes

Process Concept

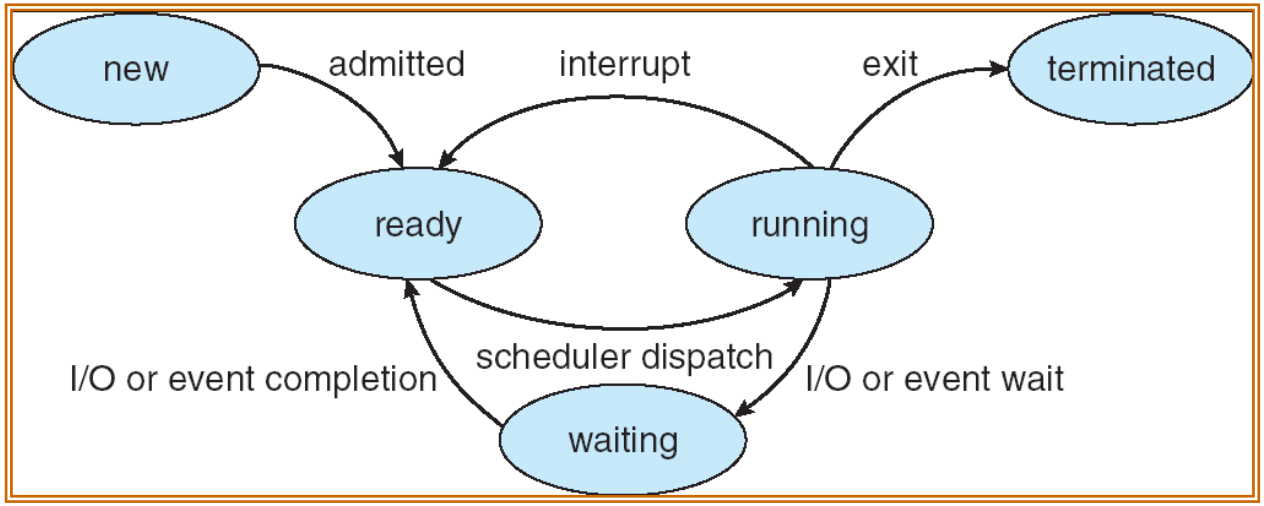
- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section
  - code
  - heap
  - allocated memory

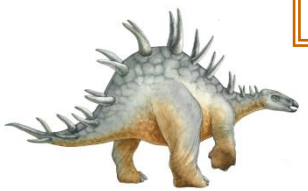
Processes and Scheduling

- Processes have *isolation* from each other
  - Address space
  - Security context
  - Termination protection
- Processes are scheduled separately from each other
- One process blocking or being pre-empted allows another to run
- On some systems, a process can be composed of several *threads* which share the process
- On a multiprocessor, processes can and do run simultaneously

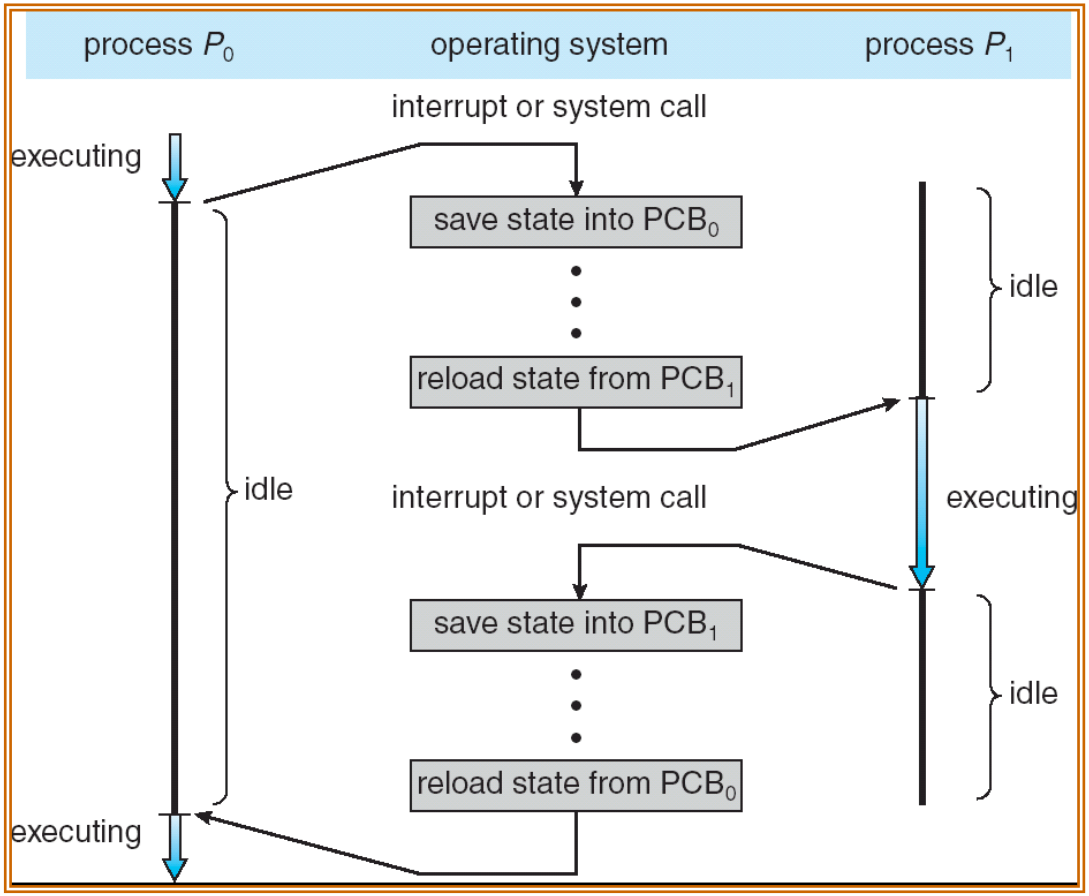
Diagram of Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a process
  - **terminated**: The process has finished execution





CPU Switch From Process to Process



Cooperating Processes

- *Independent* process cannot affect or be affected by the execution of another process.
- *Cooperating* process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity/Convenience

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
int *in = (int *) data++;
int *out = (int *) data++;
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;
item *buffer[BUFFER_SIZE];
buffer = (item *) data;
*in = *out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE - 1 elements

Bounded-Buffer – Producer Process

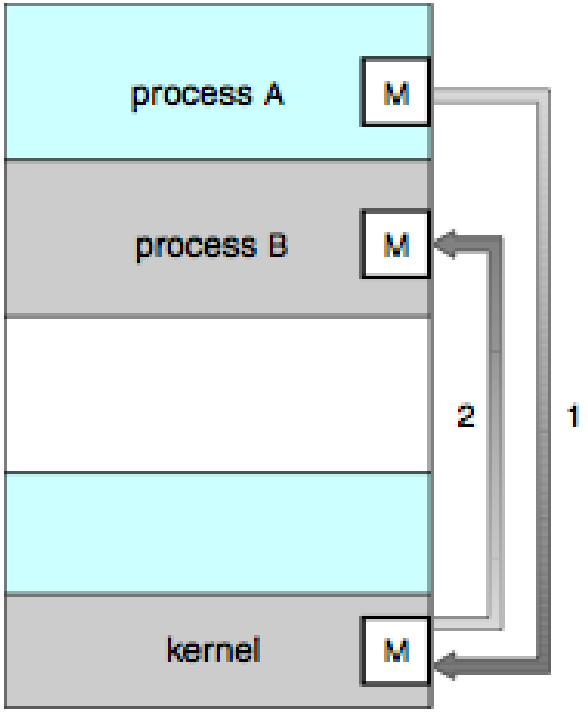
```
item nextProduced;

while (1) {
    while (((*in + 1) % BUFFER_SIZE) == *out)
          ; /* busy wait, do nothing */
    buffer[*in] = nextProduced;
    *in = (*in + 1) % BUFFER_SIZE;
}
```

Bounded-Buffer – Consumer Process

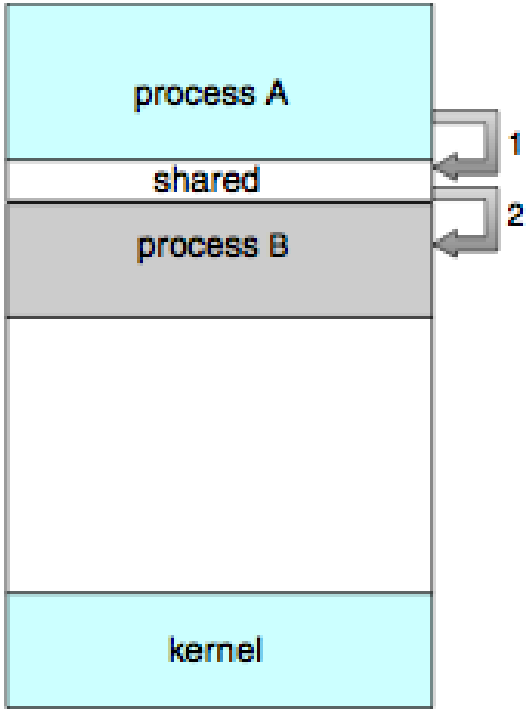
```
item nextConsumed;

while (1) {
    while (*in == *out)
        ; /* busy wait, do nothing */
    nextConsumed = buffer[*out];
    *out = (*out + 1) % BUFFER_SIZE;
}
```

Interprocess Communication

Message Passing



Shared Memory



Message Passing

- Message system – processes communicate with each other without resorting to shared variables
- Message passing facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

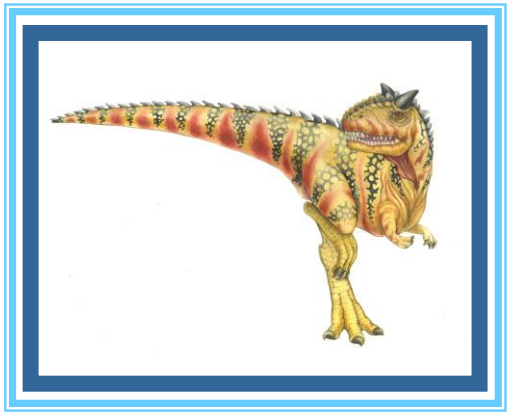
  send(*A, message*) – send a message to mailbox A

  receive(*A, message*) – receive a message from mailbox A

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** has the sender send the message and continue
  - **Non-blocking receive** has the receiver receive a valid message or null

Chapter 4: Threads



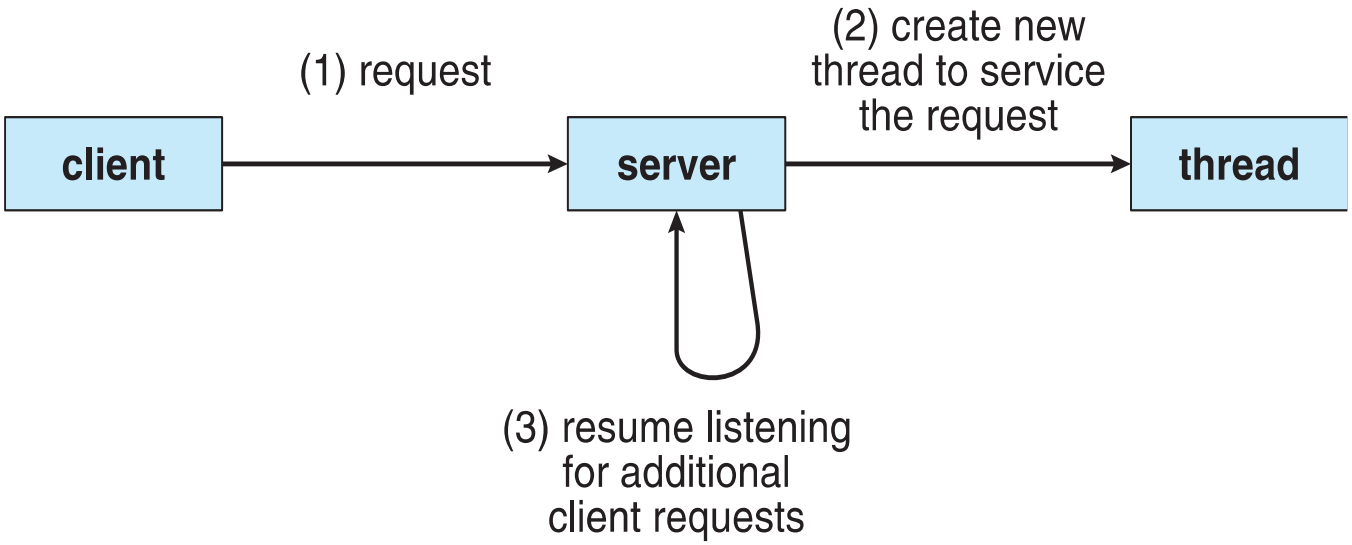
Objectives

- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Windows, and Java thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux

Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Multithreaded Server Architecture



Benefits

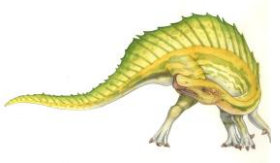
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures-each thread run parallel on diff processor

Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

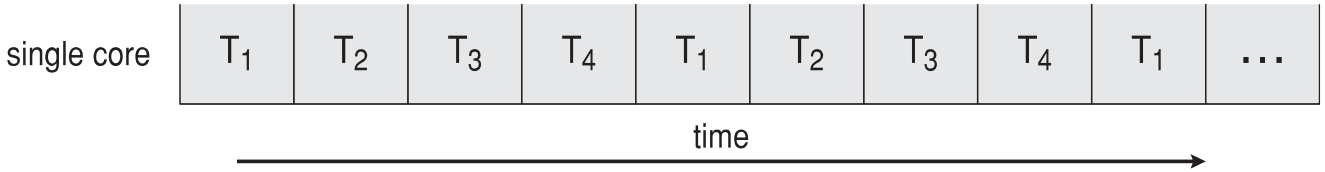
Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

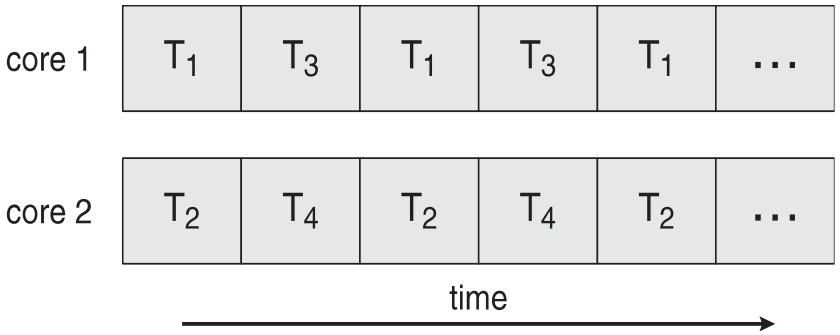


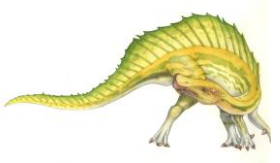
Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**

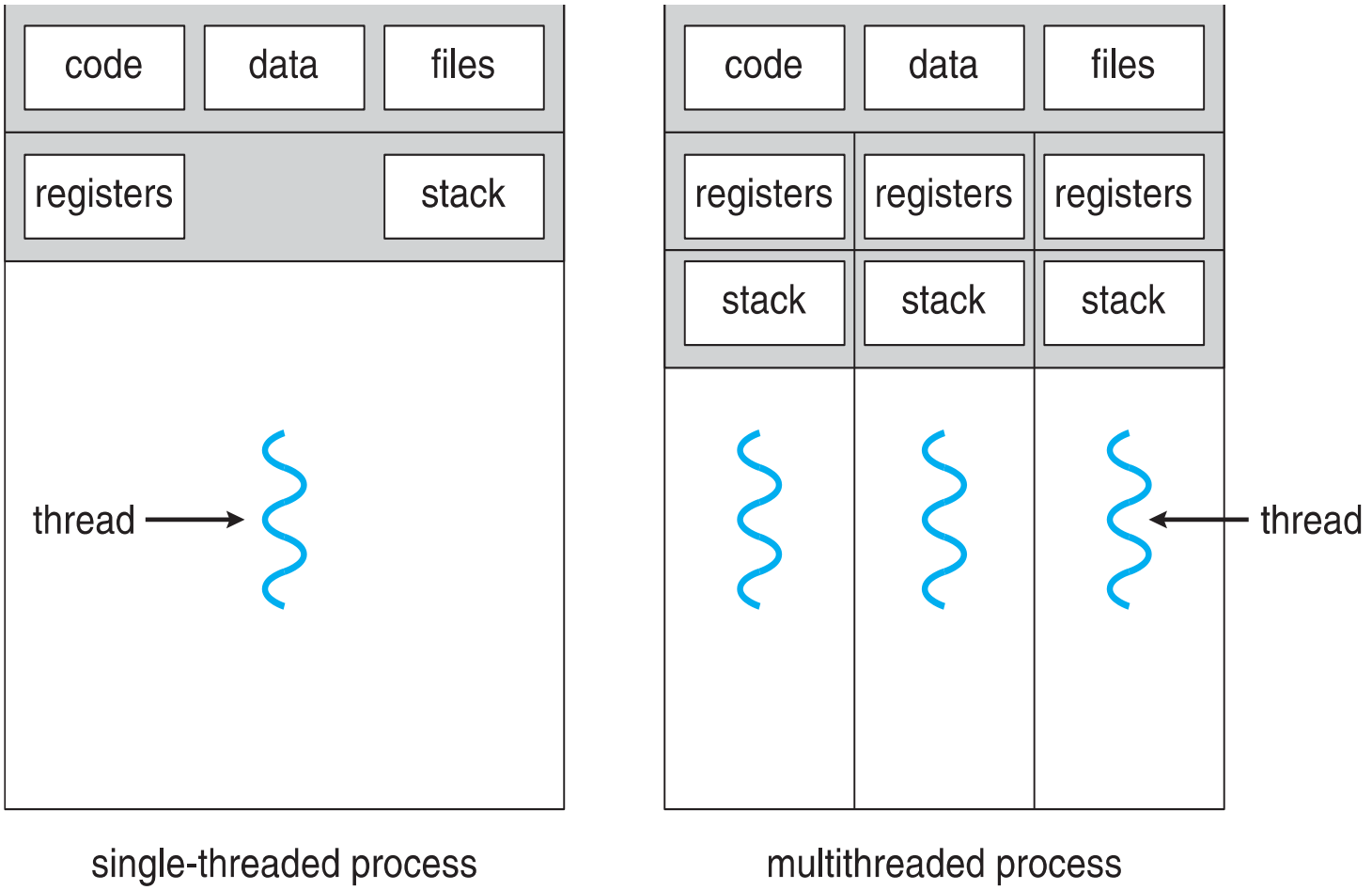


- **Parallelism on a multi-core system:**





Single and Multithreaded Processes



User Threads and Kernel Threads

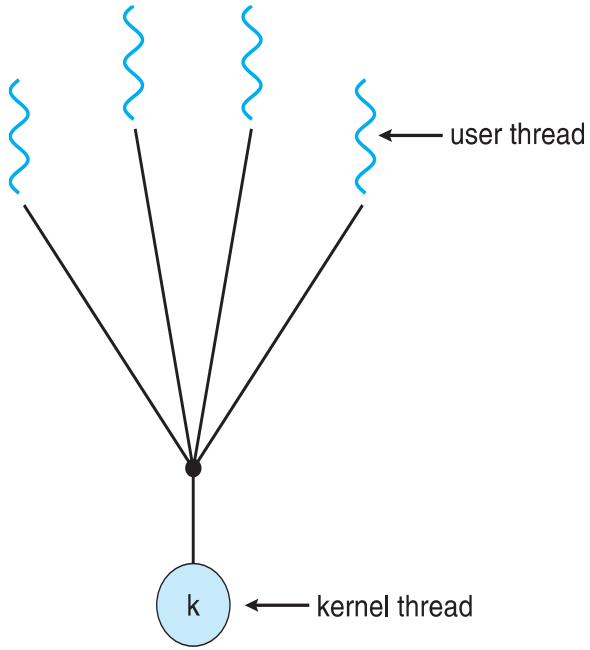
- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

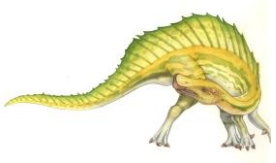
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

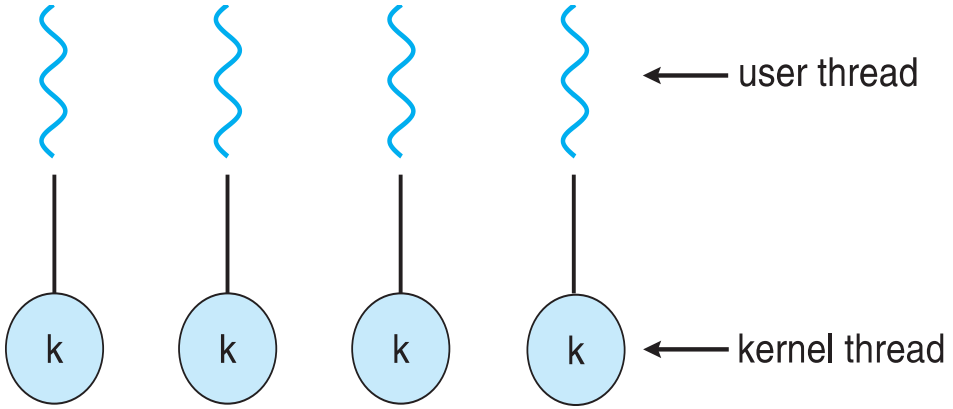
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**





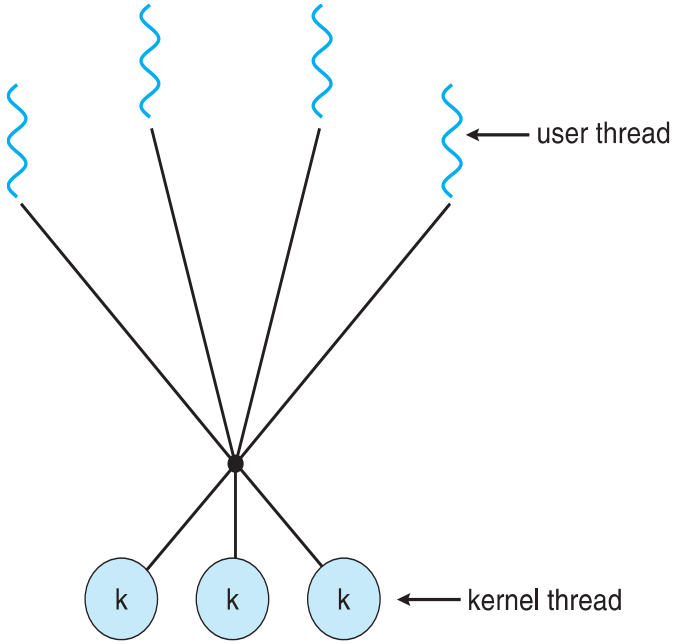
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Thread Libraries

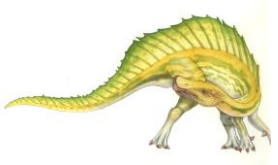
- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:

```
public interface Runnable
{
    public abstract void run();
}
```

  - Extending Thread class
  - Implementing the Runnable interface

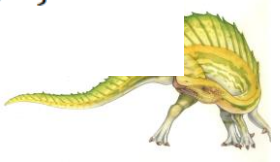


Java Multithreaded Program

```
class Sum
{
   private int sum;

   public int getSum() {
     return sum;
   }

   public void setSum(int sum) {
     this.sum = sum;
   }
}

class Summation implements Runnable
{
   private int upper;
   private Sum sumValue;

   public Summation(int upper, Sum sumValue) {
     this.upper = upper;
     this.sumValue = sumValue;
   }

   public void run() {
     int sum = 0;
     for (int i = 0; i <= upper; i++)
        sum += i;
     sumValue.setSum(sum);
   }
}
```

Java Multithreaded Program (Cont.)

```
public class Driver
{
   public static void main(String[] args) {
     if (args.length > 0) {
       if (Integer.parseInt(args[0]) < 0)
         System.err.println(args[0] + " must be >= 0.");
       else {
         Sum sumObject = new Sum();
         int upper = Integer.parseInt(args[0]);
         Thread thrd = new Thread(new Summation(upper, sumObject));
         thrd.start();
         try {
            thrd.join();
            System.out.println
                    ("The sum of "+upper+" is "+sumObject.getSum());
       } catch (InterruptedException ie) { }
       }
     }
     else
       System.err.println("Usage: Summation <integer value>"); }
}
```



Example -2

```
class ThreadA extends Thread{
    public void run( ) {
      for(int i = 1; i <= 5; i++) {
        System.out.println("From Thread A with i = "+ -1*i);
      }
      System.out.println("Exiting from Thread A ...");
    }
}
class ThreadB extends Thread {
   public void run( ) {
     for(int j = 1; j <= 5; j++) {
       System.out.println("From Thread B with j= "+2* j);
     }
     System.out.println("Exiting from Thread B ...");
      }
}
class ThreadC extends Thread{
    public void run( ) {
      for(int k = 1; k <= 5; k++) {
         System.out.println("From Thread C with k = "+ (2*k-1));
      }
      System.out.println("Exiting from Thread C ...");
    }
}
```

Example -2 (cont..)

```
public class Main {
    public static void main(String args[]) {
        ThreadA a = new ThreadA();
        ThreadB b = new ThreadB();
        ThreadC c = new ThreadC();
        a.start();
        b.start();
        c.start();
        System.out.println("... Multithreading is over ");
    }
}
```

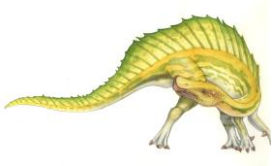
```
From Thread A with i = -1
From Thread A with i = -2
From Thread A with i = -3
From Thread B with j= 2
From Thread A with i = -4
From Thread A with i = -5
Exiting from Thread A ...
... Multithreading is over
From Thread C with k = 1
From Thread B with j= 4
From Thread B with j= 6
From Thread B with j= 8
From Thread B with j= 10
```

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), **`java.util.concurrent`** package

Operating System Examples

- Windows Threads
- Linux Threads



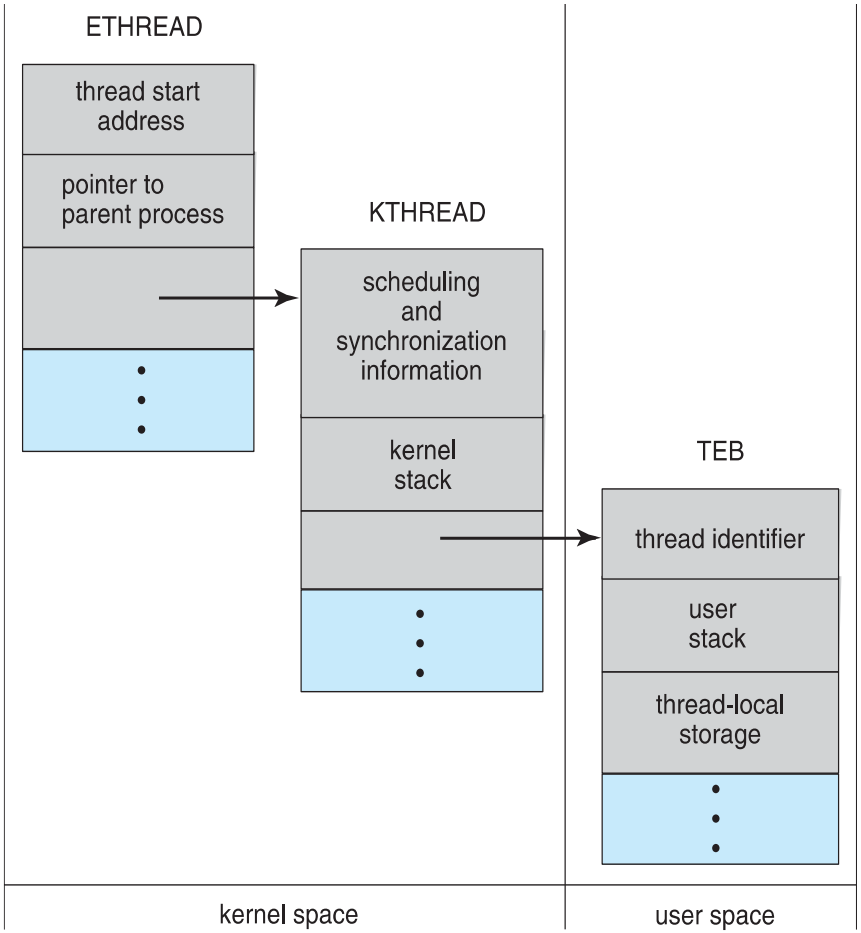
Windows Threads

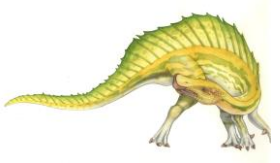
- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread

Windows Threads (Cont.)

- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

Windows Threads Data Structures



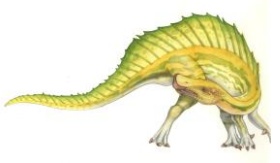


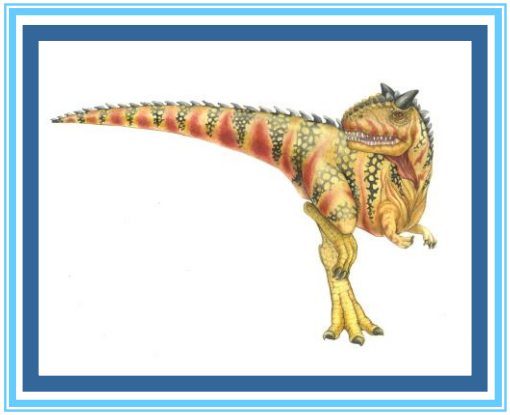
Linux Threads

- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
  - Flags control behavior

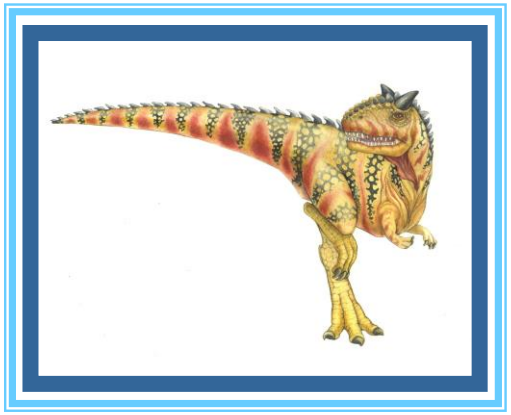
flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- **`struct task_struct`** points to process data structures (shared or unique)





Chapter 6: CPU Scheduling



Chapter 6: CPU Scheduling

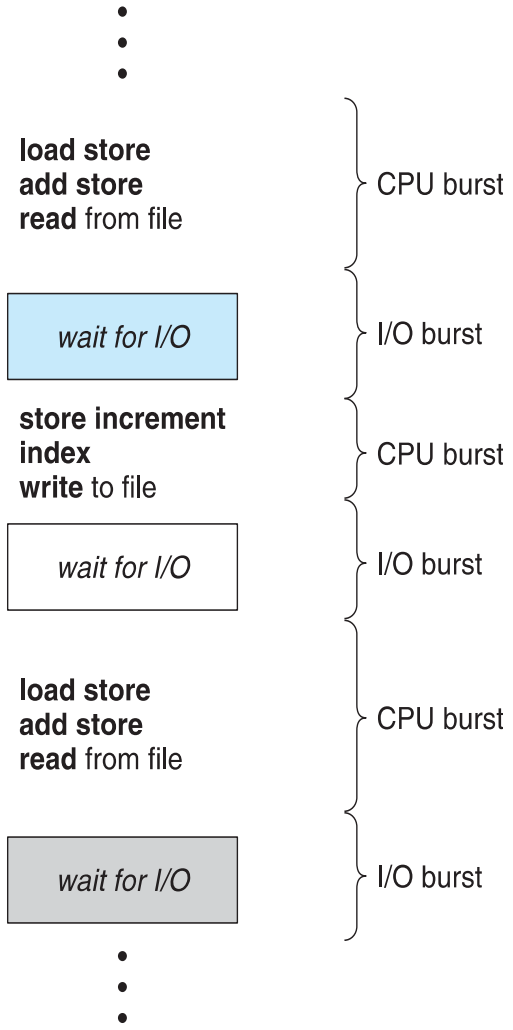
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Objectives

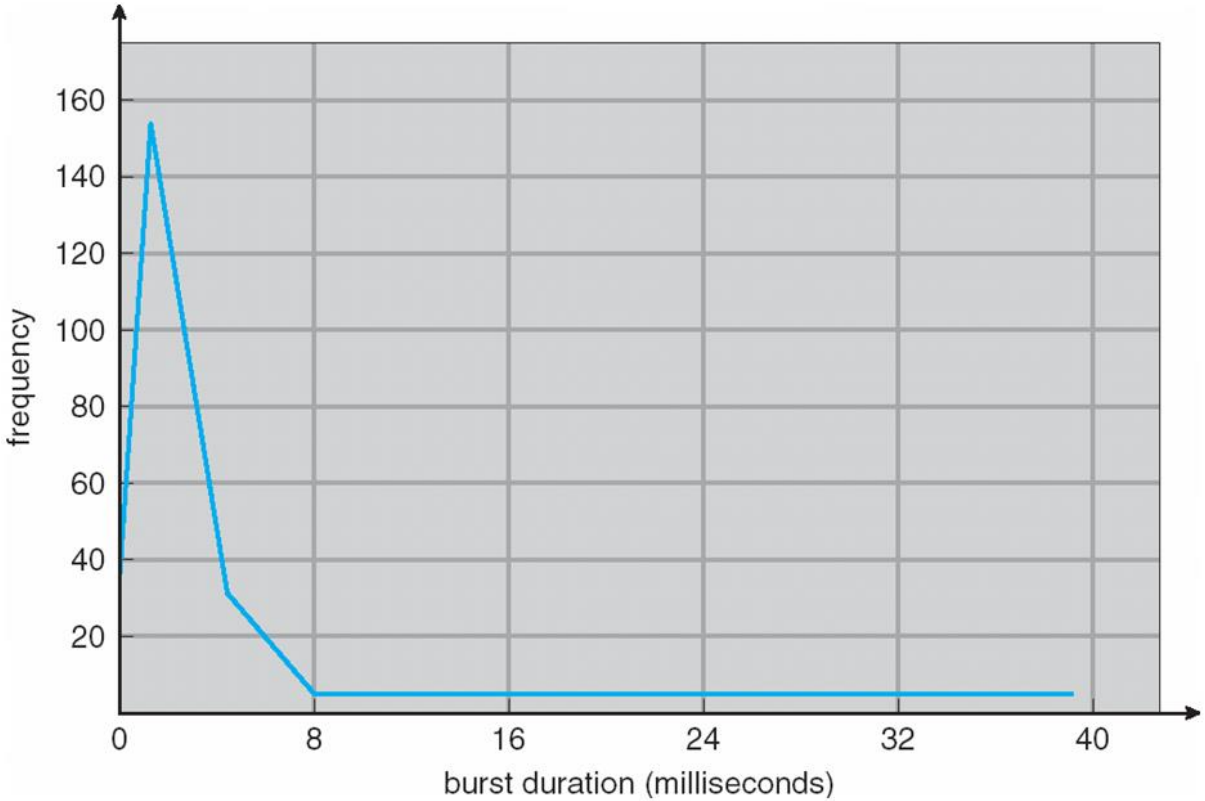
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



Histogram of CPU-burst Times



CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

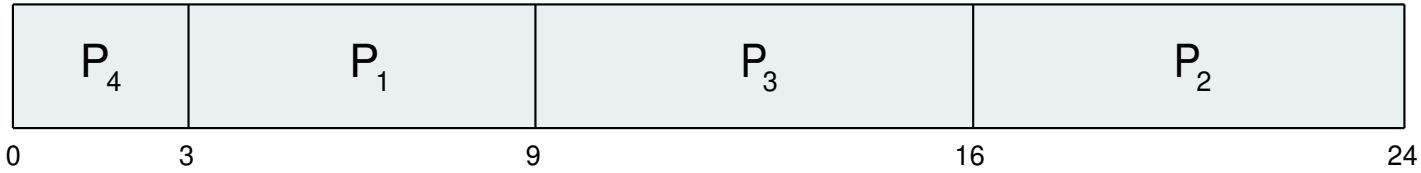
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Determining Length of Next CPU Burst

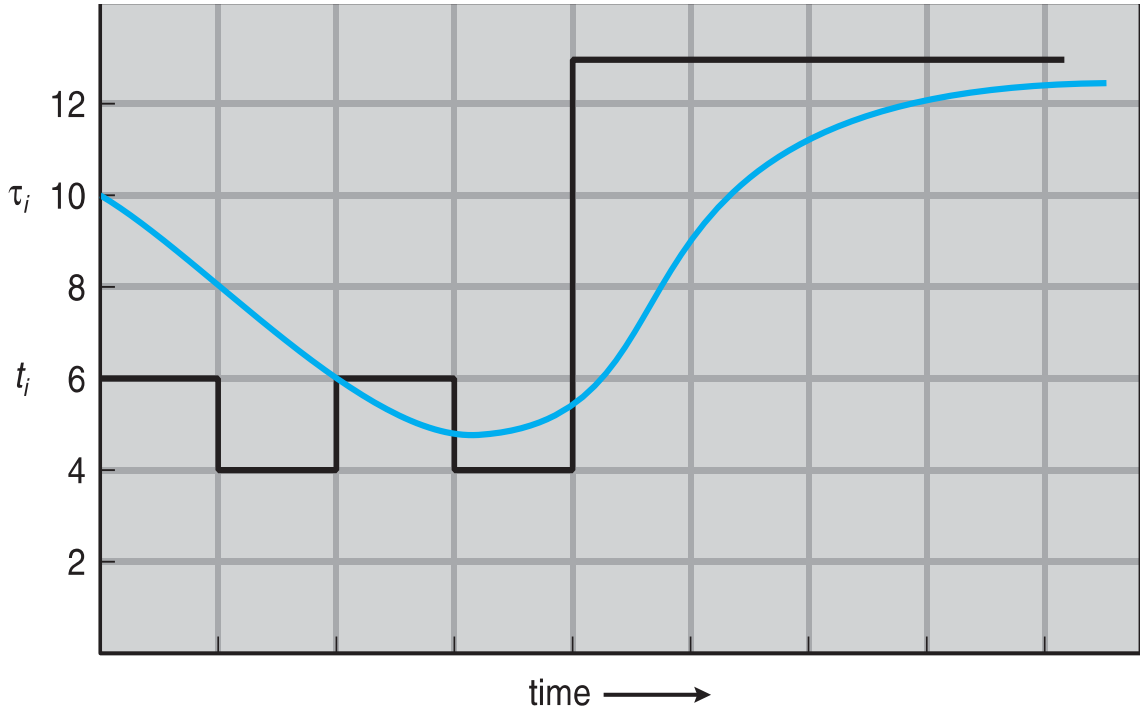
- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

- Can be done by using the length of previous CPU bursts, using exponential averaging

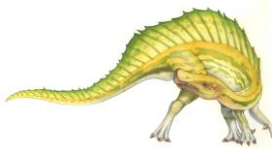
  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. $\alpha, 0 \le \alpha \le 1$
  4. Define : $\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n.$

- Commonly, α set to ½
- Preemptive version called **shortest-remaining-time-first**

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...



Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

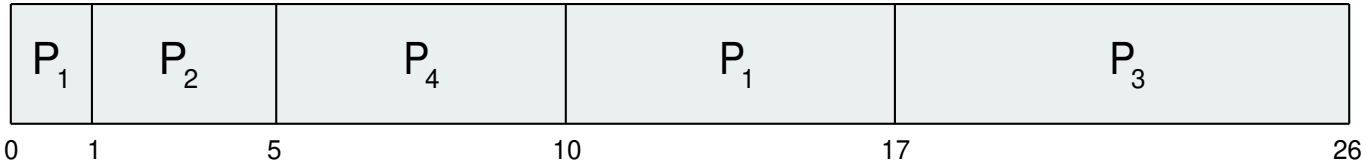
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

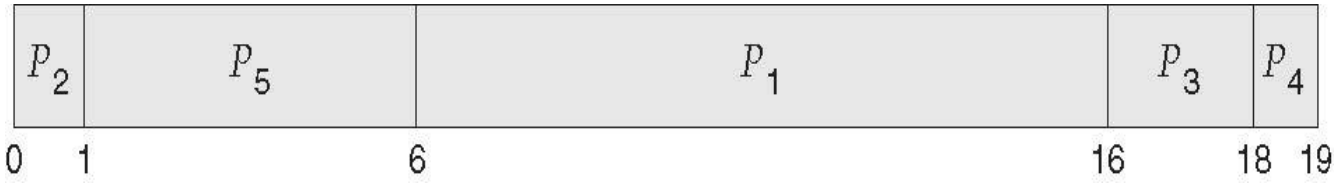
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

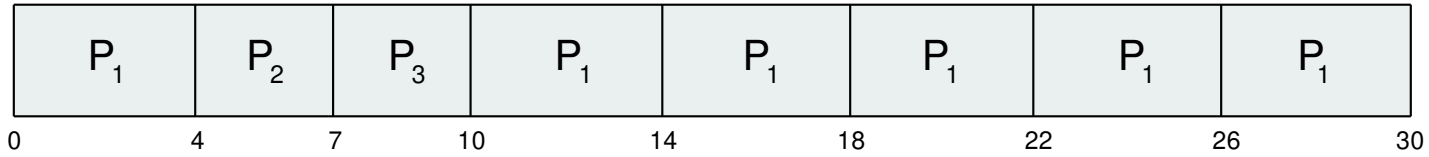
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

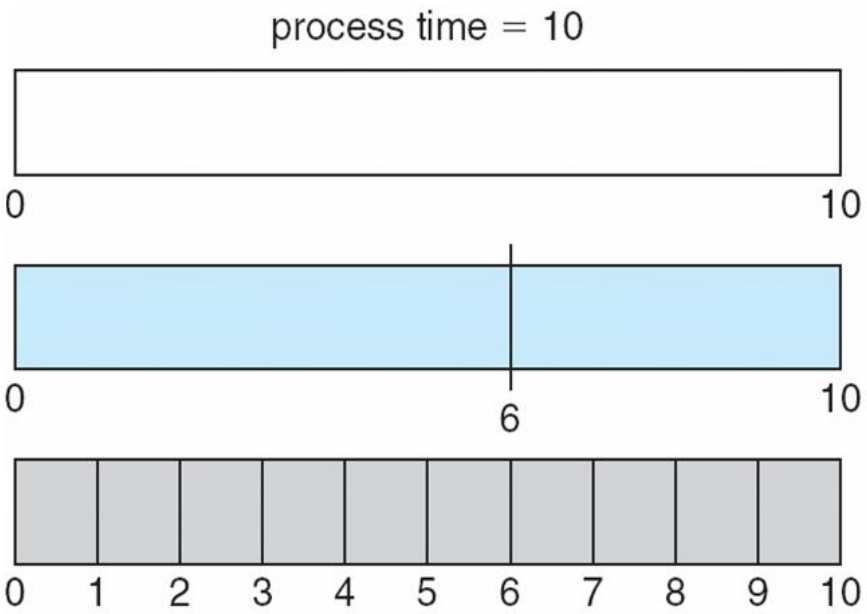
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



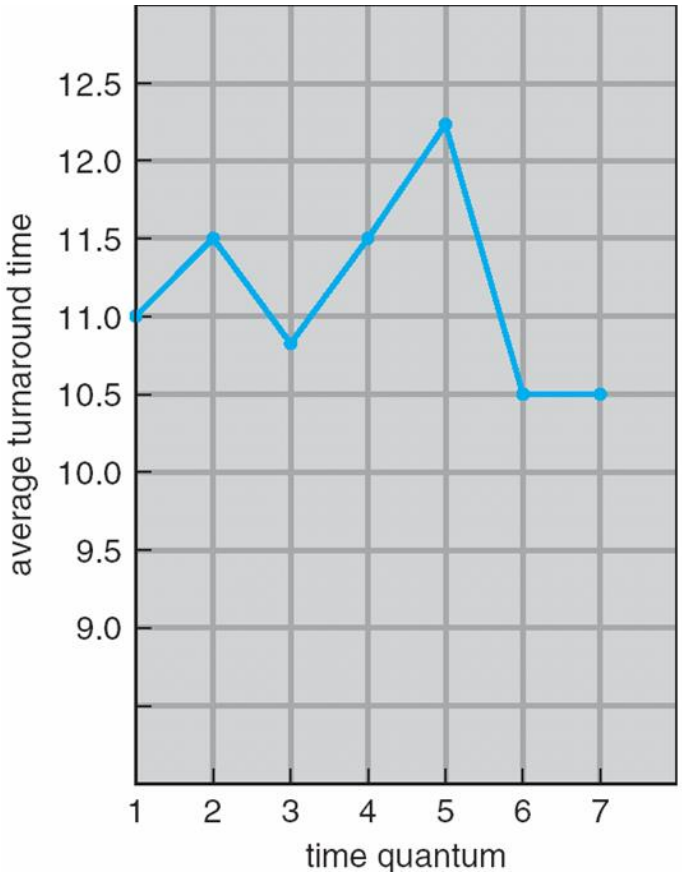
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum

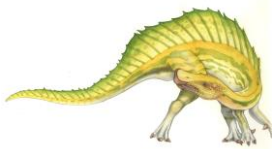


process	time
P_1	6
P_2	3
P_3	1
P_4	7

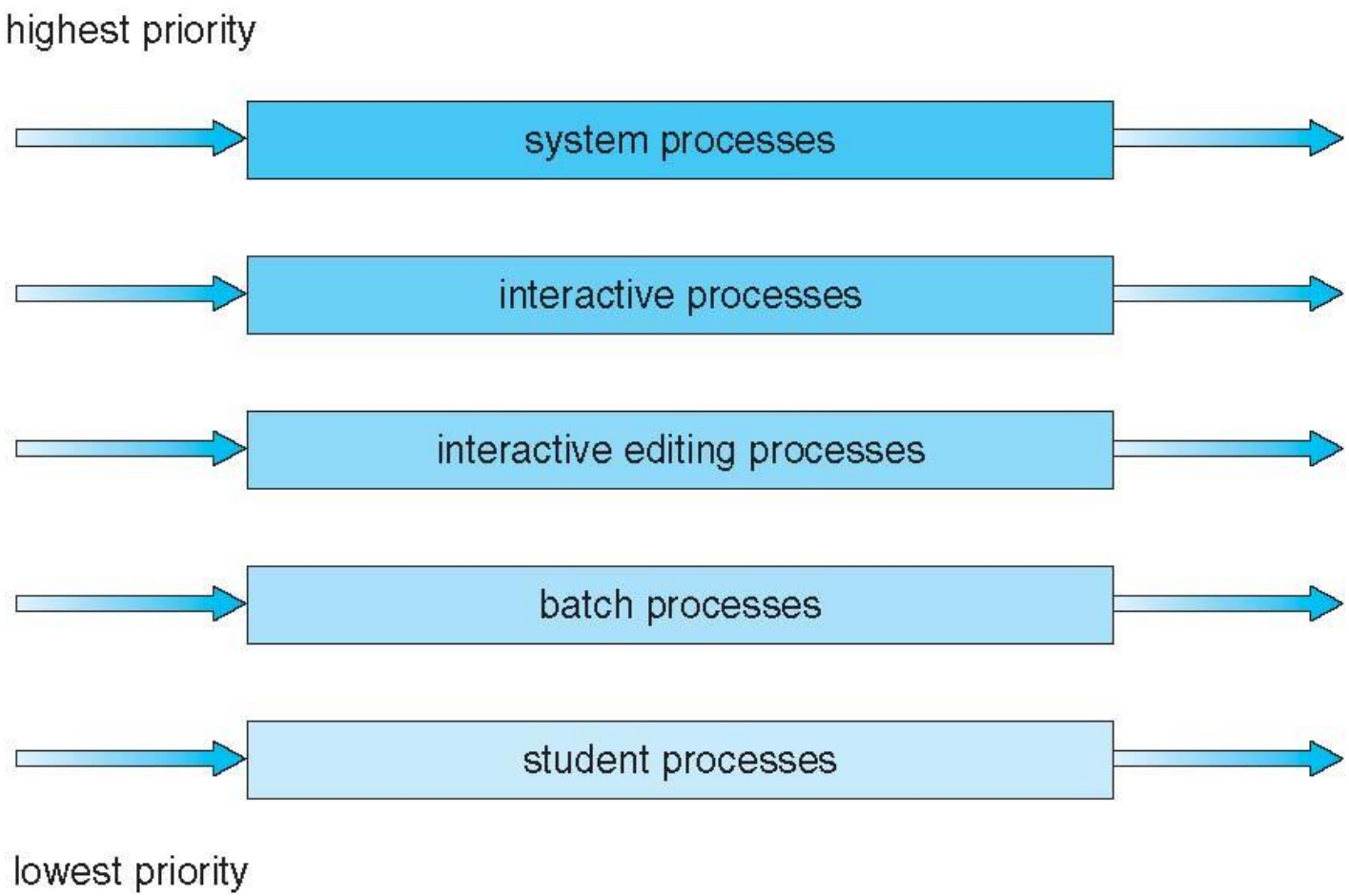
80% of CPU bursts
should be shorter than q

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS



Multilevel Queue Scheduling

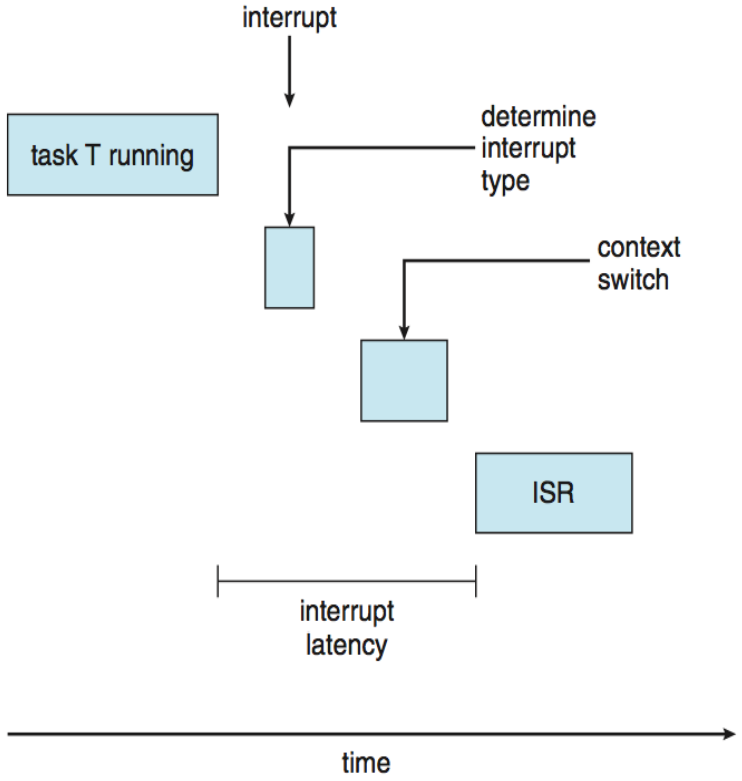


Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

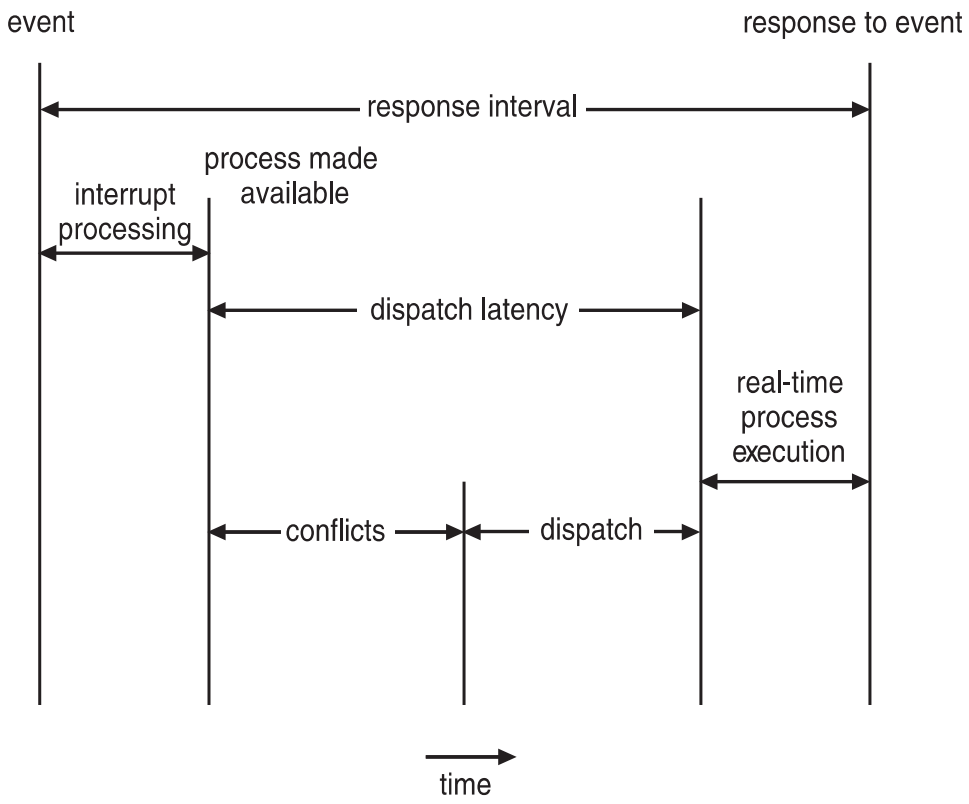
Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency – time for schedule to take current process off CPU and switch to another



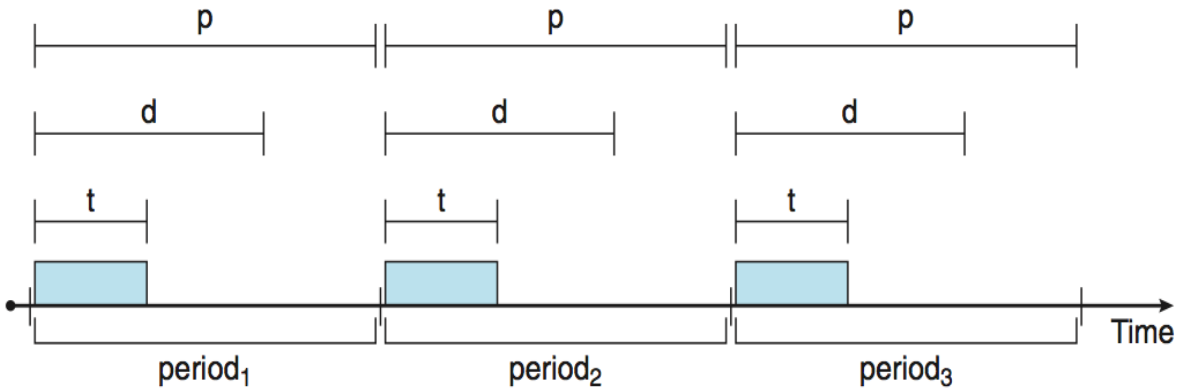
Real-Time CPU Scheduling (Cont.)

- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes

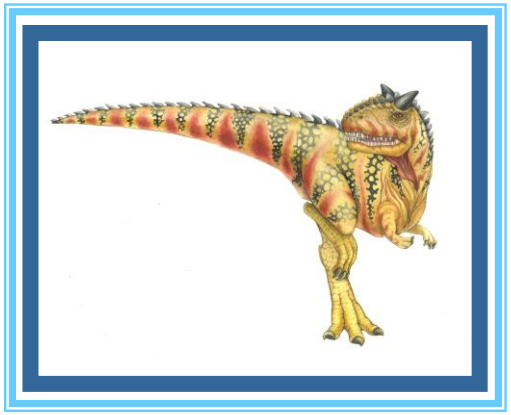


Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time *t*, deadline *d*, period *p*
  - $0 \le t \le d \le p$
  - **Rate** of periodic task is $1/p$



Chapter 5: Process Synchronization



Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

Objectives

- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (true) {
        while (counter == 0)
               ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
         counter--;
        /* consume the item in next consumed */
}
```

Race Condition

- **`counter++`** could be implemented as

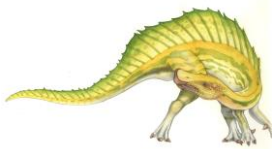
```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}



Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

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

Algorithm for Process P_i

```
do {

    while (turn == j);

            critical section

    turn = j;

            remainder section

} while (true);
```

Solution to Critical-Section Problem

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

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process $\mathbf{P_i}$ is ready!

Algorithm for Process P_i

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j);
                critical section
        flag[i] = false;
                remainder section
 } while (true);
```

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved
     P_i enters CS only if:
     either `flag[j] = false` or `turn = i`
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

OPERATING SYSTEM-SCHEDULING ALGORITHMS

WHY SCHEDULING?

- **Why do we need scheduling?**
  A typical process involves both I/O time and CPU time.
- In a uni programming system like MS-DOS, time spent waiting for I/O is wasted and CPU is free during this time.
- In multi programming systems, one process can use CPU while another is waiting for I/O. This is possible only with process scheduling.

SCHEDULING CRITERIA

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

SCHEDULING ALGORITHMS

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms.

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

- These algorithms are either **non-preemptive or preemptive**.
- Non-preemptive algorithms - once a process enters the running state, it cannot be preempted until it completes its allotted time.
- Preemptive scheduling -based on priority

CPU SCHEDULER

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **non-preemptive**
- All other scheduling is **preemptive – implications for data sharing between threads/processes**

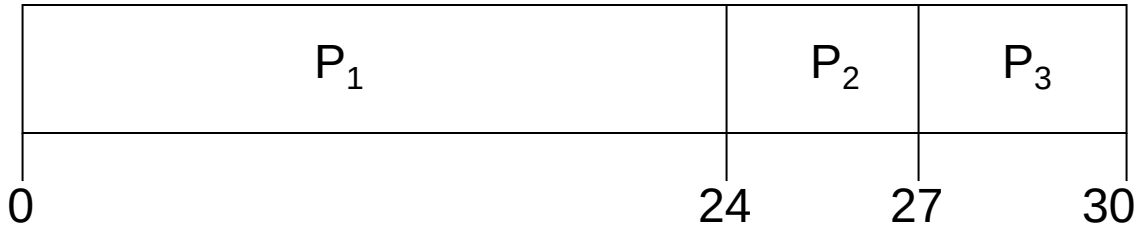
DISPATCHER

- Dispatcher module gives control of the CPU to the process selected by the scheduler;
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

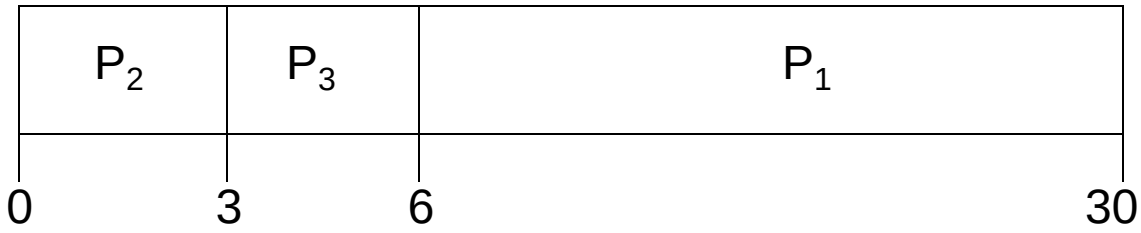


- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$, $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case

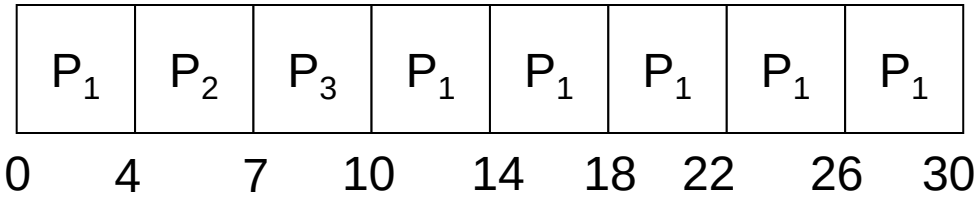
ROUND ROBIN (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- We can predict wait time: If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Performance
  - *Time quantum(q)* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ too many process /context switching

EXAMPLE OF RR WITH TIME QUANTUM = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Typically, higher average turnaround than few other, but better *response*.

Scheduling Algorithms

Shortest Job First and Sortest Remaining Time First

Non-preemptive vs. Preemptive Scheduling

Under *non-preemptive scheduling*, each running process keeps the CPU until it completes or it switches to the waiting (blocked) state

Under *preemptive scheduling,* a running process may be also forced to release the CPU even though it is neither completed nor blocked.

- In time-sharing systems, when the running process reaches the end of its time *quantum*

Algorithm 3: Non-Preemptive Shortest Job First (SJF)

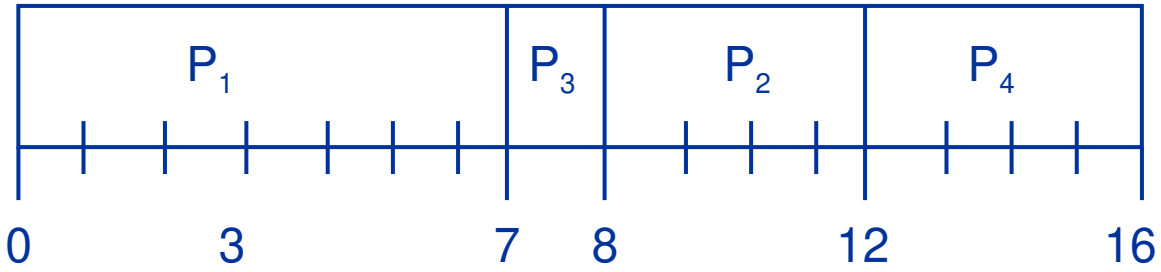
Characteristics:

- Always assign job with shortest next CPU burst to CPU

Example:

Process	Arrival	Burst
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Gantt Chart

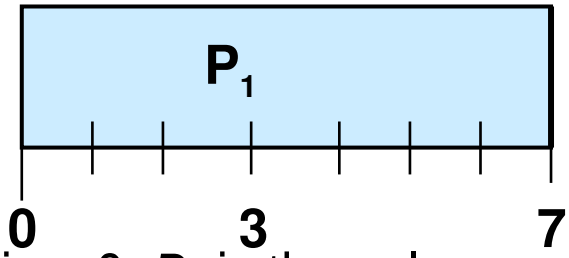


Avg wait time: (0 + 6 + 3 + 7) / 4 = 4 ms

Example for Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

SJF (non-preemptive)

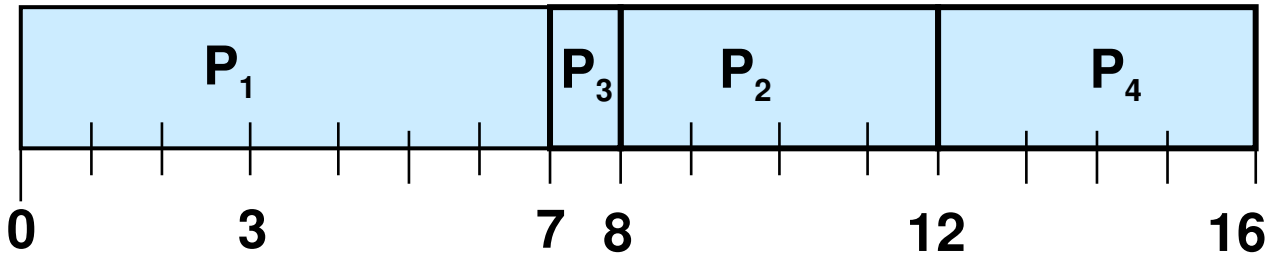


At time 0, P_1 is the only process, so it gets the CPU and runs to completion

Example for Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

Once P_1 has completed the queue now holds P_2, P_3 and P_4



P_3 gets the CPU first since it is the shortest. P_2 then P_4 get the CPU in turn (based on arrival time)

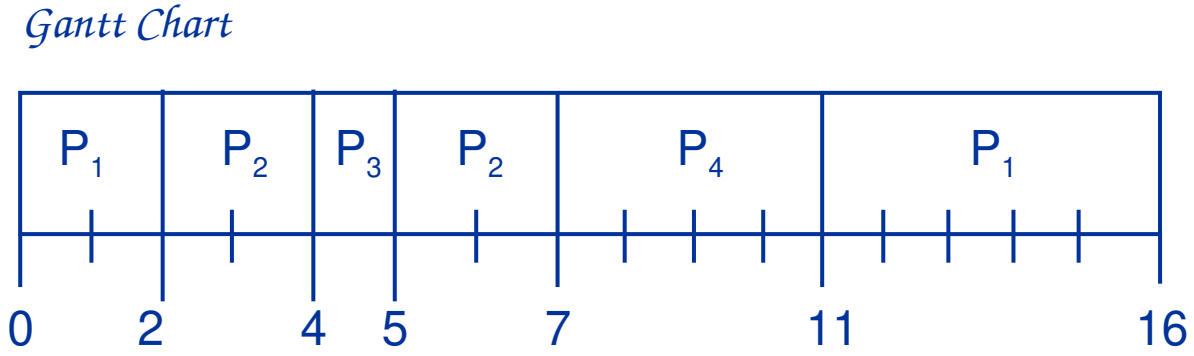
Algorithm 4: Preemptive Shortest Job First (SJF)- Shortest-Remaining-Time-First (SRTF)

Characteristics:

- Like Shortest Job First, but running job can be preempted.

Example:

Process	Arrival	Burst
P1	0	7
P2	2	4
P3	4	1
P4	5	4



Average waiting time = (9 + 1 + 0 +2)/4 = 3 ms

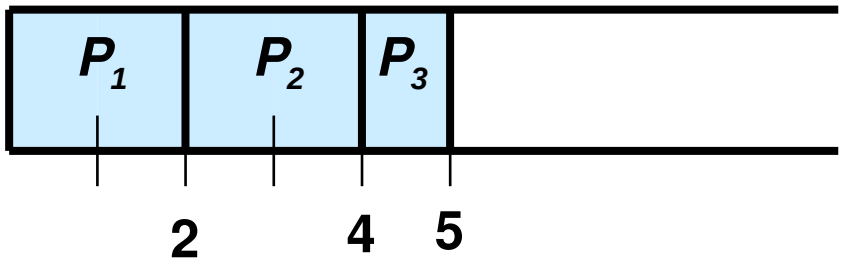
Example for Preemptive SJF (SRTF)

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

Time 0 – P_1 gets the CPU Ready = [(P_1,7)]

Time 2 – P_2 arrives – CPU has P_1 with time=5, Ready = [(P_2,4)] – P_2 gets the CPU

Time 4 – P_3 arrives – CPU has P_2 with time = 2, Ready = [(P_1,5),(P_3,1)] – P_3 gets the CPU



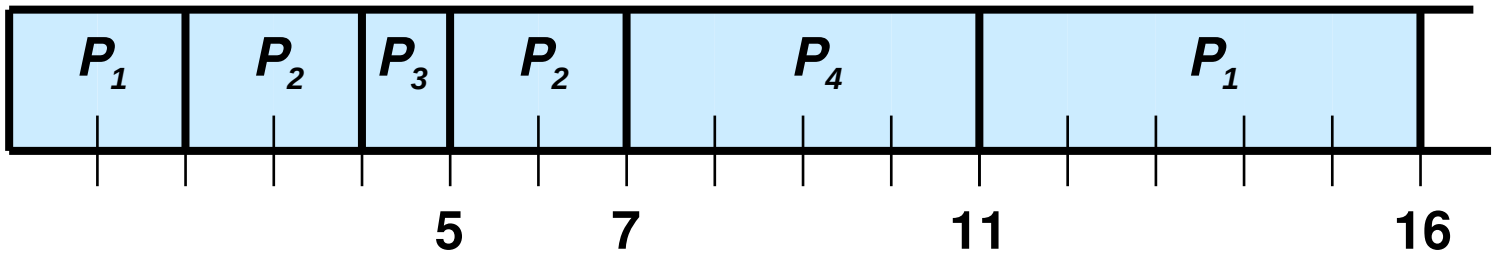
Example for Preemptive SJF (SRTF)

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

Time 5 – P_3 completes and P_4 arrives - Ready = [(P_1,5),(P_2,2),(P_4,4)] – P_2 gets the CPU

Time 7 – P_2 completes – Ready = [(P_1,5),(P_4,4)] – P_4 gets the CPU

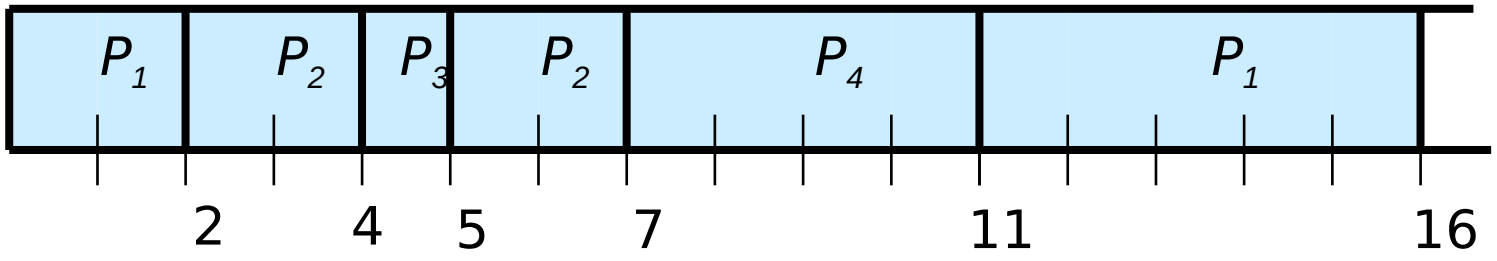
Time 11 – P_4 completes, P_1 gets the CPU



Example for Preemptive SJF (SRTF)

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

SJF (preemptive)



Average waiting time = (9 + 1 + 0 +2)/4 = 3

Implementing SJF

SJF is great, but how do we implement it?

- We don’t know a priori how long a job’s burst time is
- We have to try to *predict* the burst time

Priority-Based Scheduling

A priority number (integer) is associated with each process

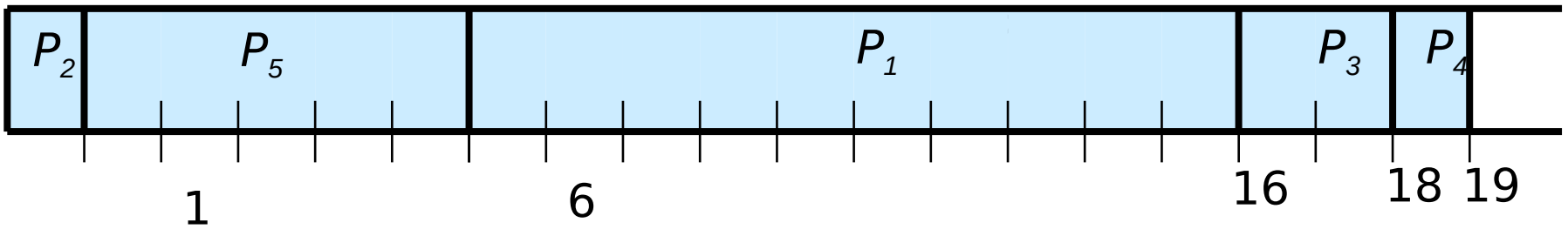
The CPU is allocated to the process with the highest priority (smallest integer $\equiv$ highest priority).

- Preemptive
- Non-preemptive

SJF is a priority scheme with the priority the remaining time.

Example for Non Preemptive Priority-based Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



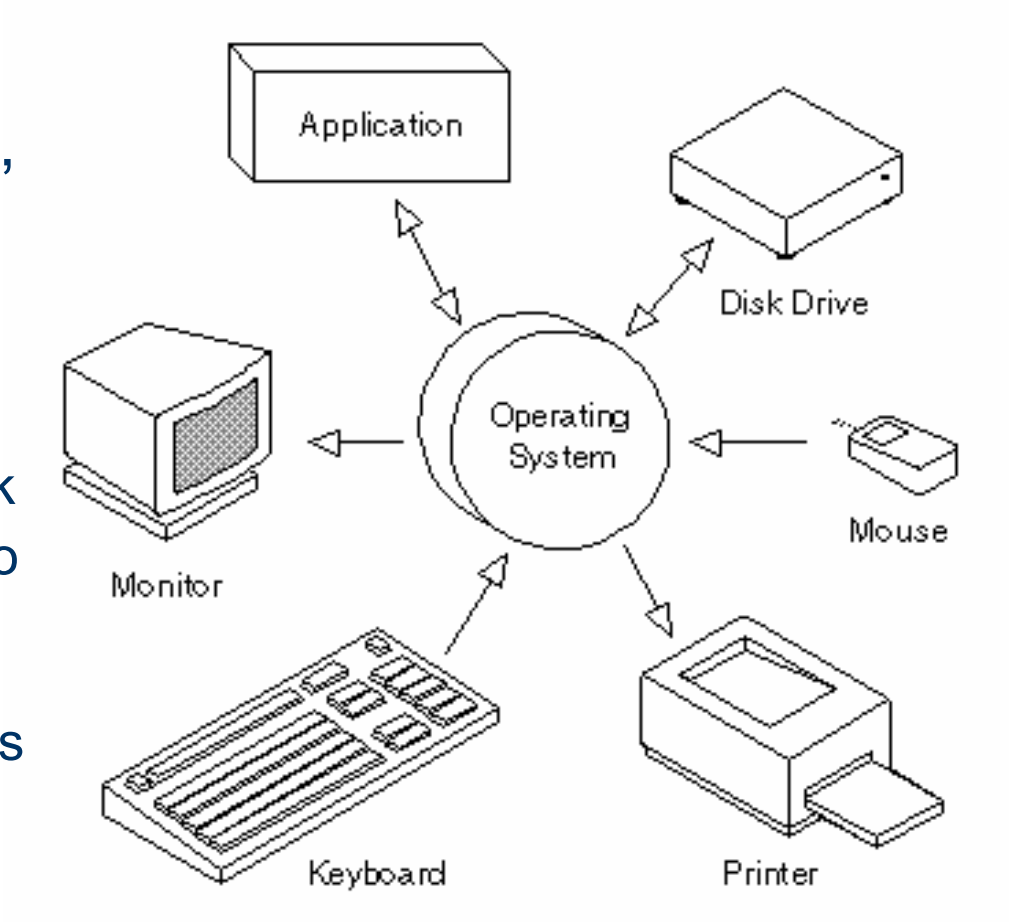
2. Operating System Case Study: Linux

Reference

- S.M. Sarwar, R. Koretsky and S.A. Sarwar, *Linux – The Textbook*, Addison Wesley, 1st ed, 2002

Features of modern OS

- To facilitate easy, efficient, fair, orderly, and secure use of resources
  - Provide a user interface
  - Organize files on disk
  - Allocating resource to different users with security control
  - Co-ordinate programs to work with devices and other programs



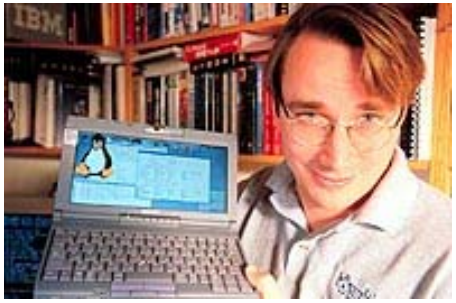
Case study: Linux

A. Development of Linux

- Before Linux
  - In 80's, Microsoft's DOS was the dominated OS for PC
    - single-user, single-process system
  - Apple MAC is better, but expensive
  - UNIX is much better, but much much expensive. Only for minicomputer for commercial applications
  - People was looking for a UNIX based system, which is cheaper and can run on PC
  - Both DOS, MAC and UNIX are proprietary, i.e., the source code of their kernel is protected
    - No modification is possible without paying high license fees

- GNU project
  - Established in 1984 by Richard Stallman, who believes that software should be free from restrictions against copying or modification in order to make better and efficient computer programs
  - GNU is a recursive acronym for "GNU's Not Unix"
  - Aim at developing a complete Unix-like operating system which is free for copying and modification
  - Companies make their money by maintaining and distributing the software, e.g. optimally packaging the software with different tools (Redhat, Slackware, Mandrake, SuSE, etc)
  - Stallman built the first free GNU C Compiler in 1991. But still, an OS was yet to be developed

- Beginning of Linux
  - A famous professor Andrew Tanenbaum developed Minix, a simplified version of UNIX that runs on PC
  - Minix is for class teaching only. No intention for commercial use
  - In Sept 1991, Linus Torvalds, a second year student of Computer Science at the University of Helsinki, developed the preliminary kernel of Linux, known as Linux version 0.0.1
  - It was put to the Internet and received enormous response from worldwide software developers
  - By December came version 0.10. Still Linux was little more than in skeletal form.



- Confrontation and Development
  - Message from Professor Andrew Tanenbaum
    - " I still maintain the point that designing a monolithic kernel in 1991 is a fundamental error. Be thankful you are not my student. You would not get a high grade for such a design :-)" (Andrew Tanenbaum to Linus Torvalds)
    - "Linux is obsolete".

      (Remark made by Andrew Tanenbaum)
  - But work went on. Soon more than a hundred people joined the Linux camp. Then thousands. Then hundreds of thousands
  - It was licensed under GNU General Public License, thus ensuring that the source codes will be free for all to copy, study and to change.

- Linux Today
  - Linux has been used for many computing platforms
    - PC, PDA, Supercomputer,…
  - Current kernel version 2.6.13
  - Not only character user interface but graphical user interface, thanks to the X-Window technology
  - Commercial vendors moved in Linux itself to provide freely distributed code. They make their money by compiling up various software and gathering them in a distributable format
    - Red Hat, Slackware, etc
  - Chinese distribution of Linux also appeared in Taiwan and China - CLE, Red Flag Linux

Linux Pros and Cons

- Advantages over Windows
  - It's almost free to relatively inexpensive
  - Source code is included
  - Bugs are fixed quickly and help is readily available through the vast support in Internet
  - Linux is more stable than Windows
  - Linux is truly multi-user and multi-tasking
    - multiuser: OS that can simultaneously serve a number of users
    - multitasking: OS that can simultaneously execute a number of programs
  - Linux runs on equipment that other operating systems consider too underpowered, e.g. 386 systems, PDA, etc

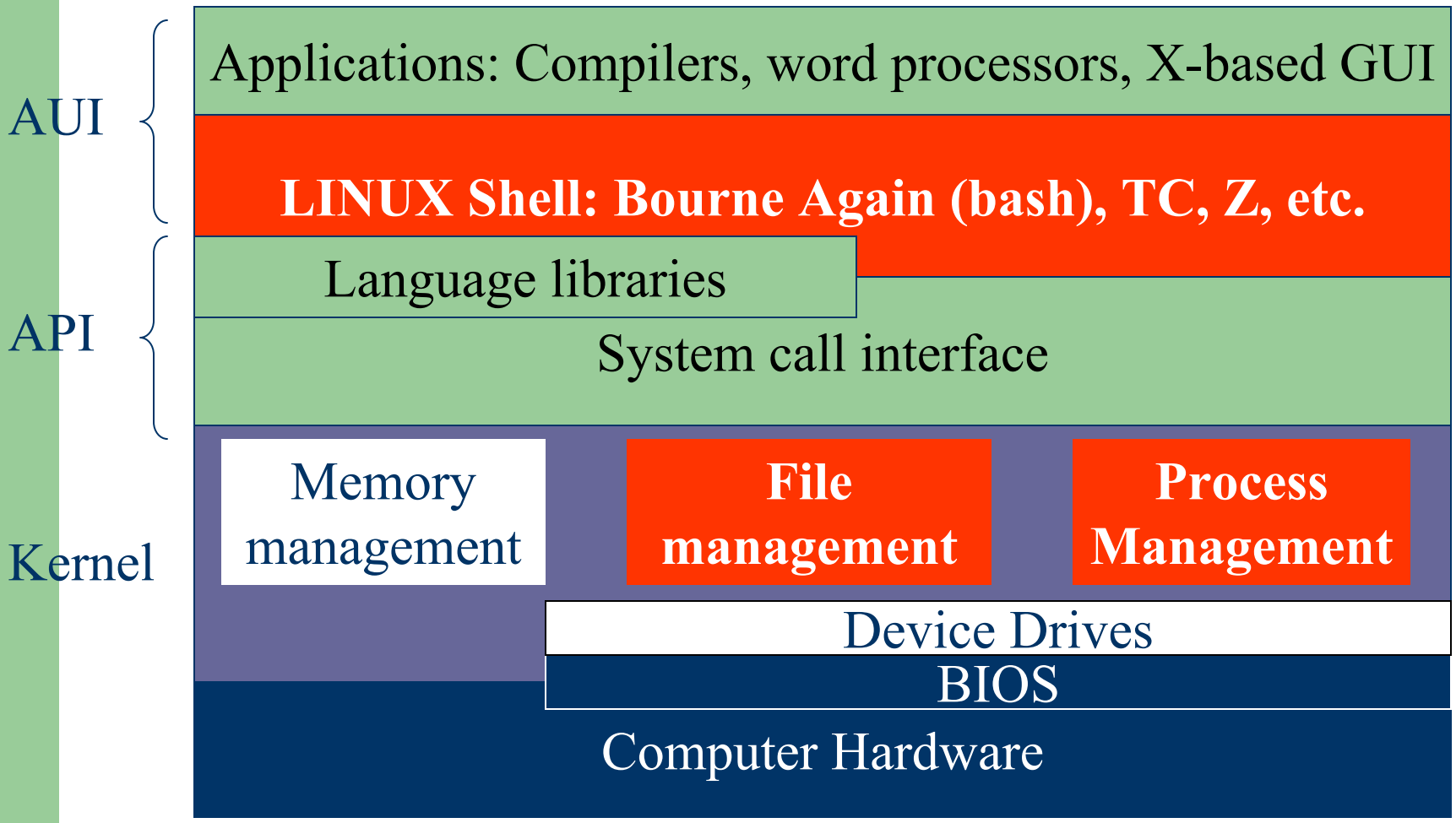
Linux Pros and Cons (Cont)

- Disadvantages compared with Windows
  - Isn't as popular as Windows
  - No one commercial company is responsible for Linux
  - Linux is relatively hard to install, learn and use

- Hence currently, Linux is mainly used in commercial applications, server implementation
- More than 75% current network servers are developed based on Linux or Unix systems
  - Due to the relatively high reliability

Case study: Linux

B. Linux System Architecture


AUI
API
Kernel
Applications: Compilers, word processors, X-based GUI
LINUX Shell: Bourne Again (bash), TC, Z, etc.
Language libraries
System call interface
Memory management
File management
Process Management
Device Drives
BIOS
Computer Hardware

- Kernel
  - The part of an OS where the real work is done
- System call interface
  - Comprise a set of functions (often known as Application Progarmmer's Interface API) that can be used by the applications and library routines to use the services provided by the kernel
- Application User's Interface
  - Interface between the kernel and user
  - Allow user to make commands to the system
  - Divided into text based and graphical based

- File Management
  - Control the creation, removal of files and provide directory maintenance
  - For a multiuser system, every user should have its own right to access files and directories
- Process Management
  - For a multitask system, multiple programs can be executed simultaneously in the system
  - When a program starts to execute, it becomes a process
  - The same program executing at two different times will become two different processes
  - Kernel manages processes in terms of creating, suspending, and terminating them
  - A process is protected from other processes and can communicate with the others

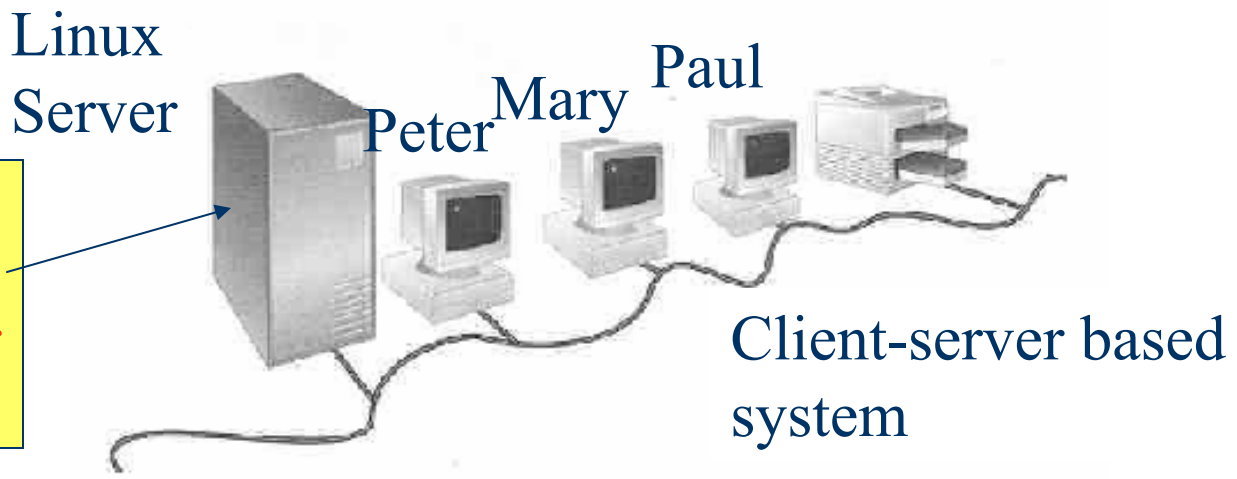
- Memory management
  - Memory in a computer is divided into main memory (RAM) and secondary storage (usually refer to hard disk)
  - Memory is small in capacity but fast in speed, and hard disk is vice versa
  - Data that are not currently used should be saved to hard disk first, while data that are urgently needed should be retrieved and stored in RAM
  - The mechanism is referred as memory management
- Device drivers
  - Interfaces between the kernel and the BIOS
  - Different device has different driver

Case study: Linux

B.1 User interface

Linux User Login

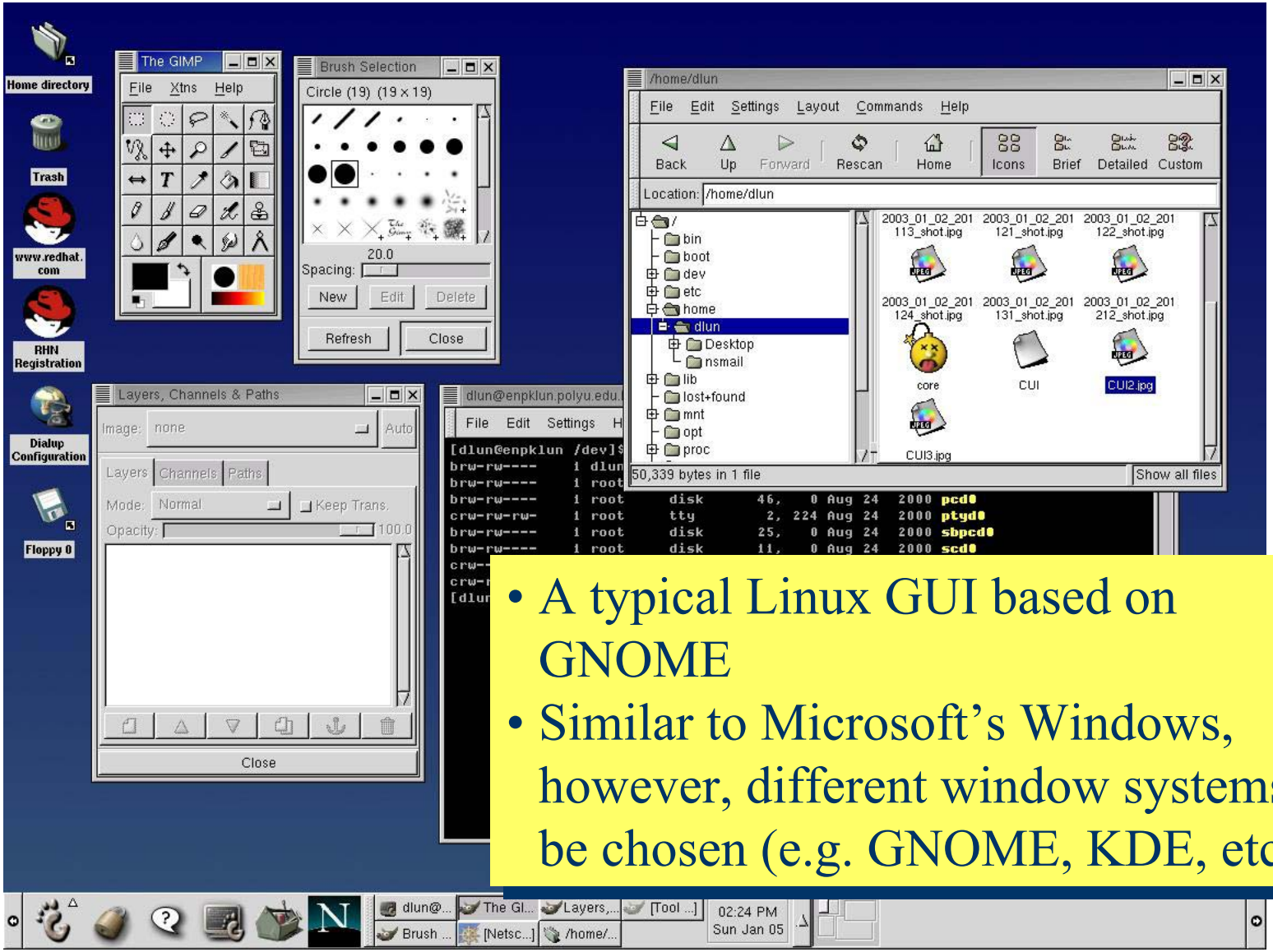
- Linux is a multiuser OS
- Allow multiple users to use the resource of a computer at the same time
- Every user needs to login the system with the password provided to identify their right in using the resource
- Require for both client-server based system or desktop



Linux User Interface

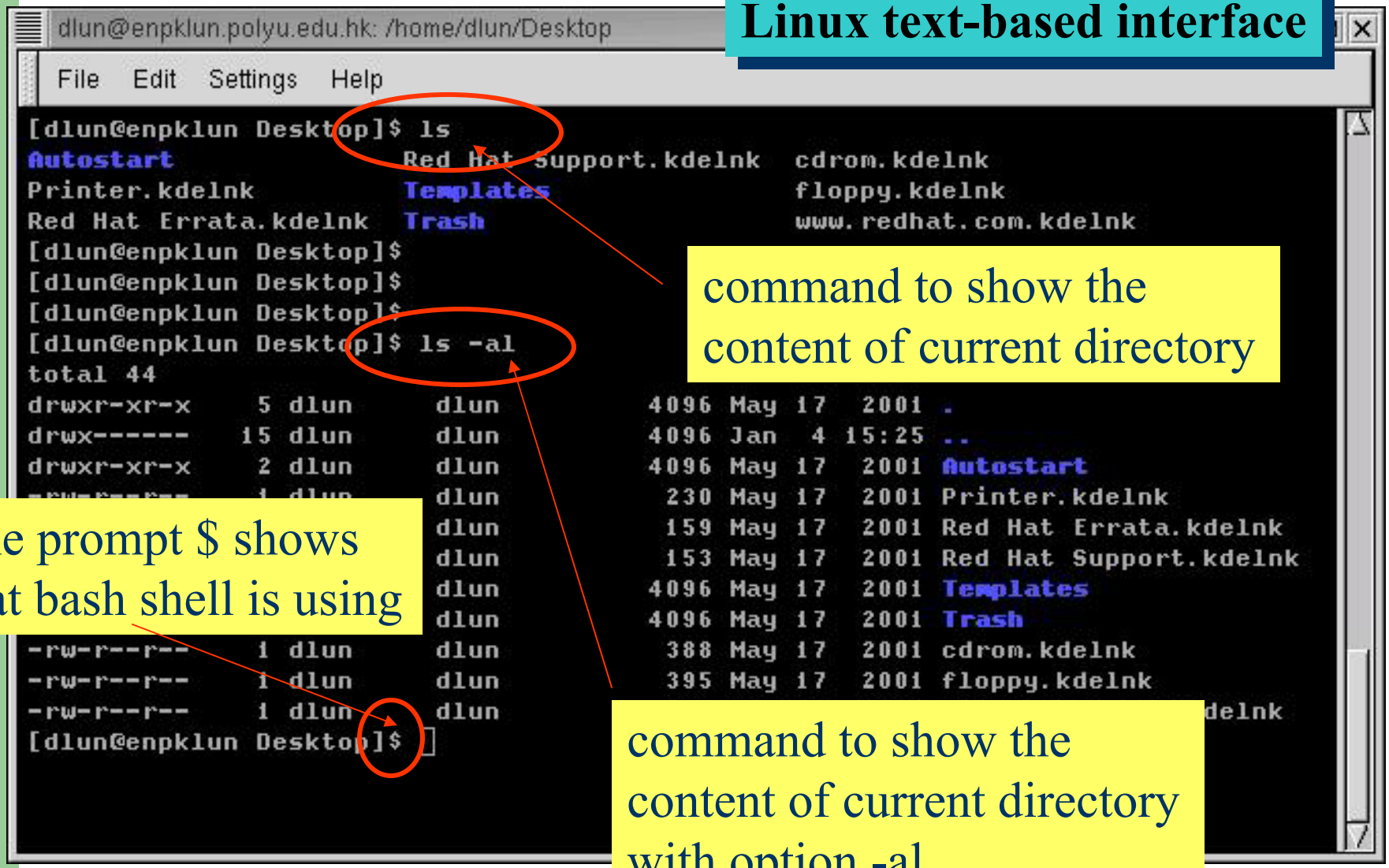
- Traditional Linux (Unix also) uses command-driven interface (or text-based interface)
  - User needs to type lines of command to instruct the computer to work, similar to DOS
  - Advantage: fast in speed. Very few resource is required for its implementation
  - Disadvantages: user needs to type, hence can easily make error. Besides, user needs to memorize all commands
  - Suitable for expert users and for the systems that interaction with user is not frequent, such as servers

- By adopting the X-Window technology, graphical user interface (GUI) is available for Linux:
  - Uses pointing devices (e.g. mouse) to control the system, similar to Microsoft's Windows
  - Provide menu-driven and/or icon-driven interfaces
    - menu-driven: user is provided with a menu of choices. Each choice refers to a particular task
    - icon-driven: tasks are represented by pictures (icon) and shown to user. Click on an icon invokes one task
  - Advantages: No need to memorize commands. Always select task from menus or icons
  - Disadvantages: Slow and require certain resource for its implementation
  - Suitable for general users and systems, such as PC



- A typical Linux GUI based on GNOME
- Similar to Microsoft's Windows, however, different window systems can be chosen (e.g. GNOME, KDE, etc)

Linux text-based interface

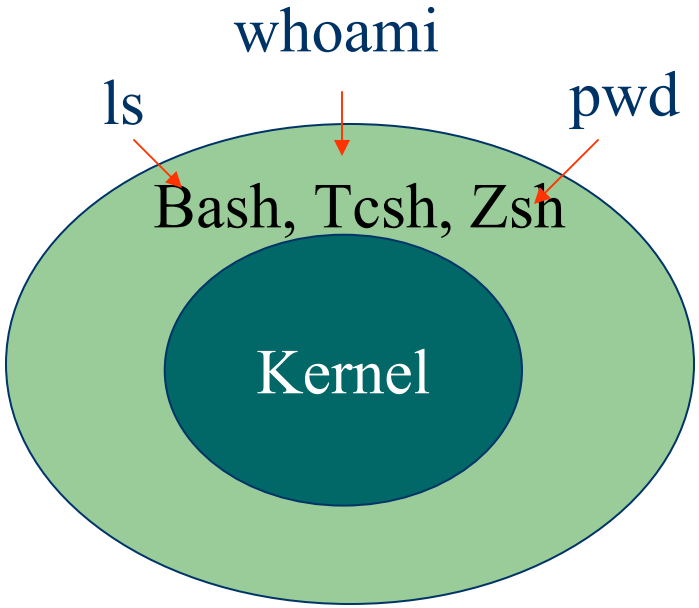


command to show the content of current directory

The prompt $ shows that bash shell is using

command to show the content of current directory with option -al

Linux Shell



- Shell interprets the command and request service from kernel
- Similar to DOS but DOS has only one set of interface while Linux can select different shell
  - Bourne Again shell (Bash), TC shell (Tcsh), Z shell (Zsh)
- Different shell has similar but different functionality
- Bash is the default for Linux
- Graphical user interface of Linux is in fact an application program work on the shell

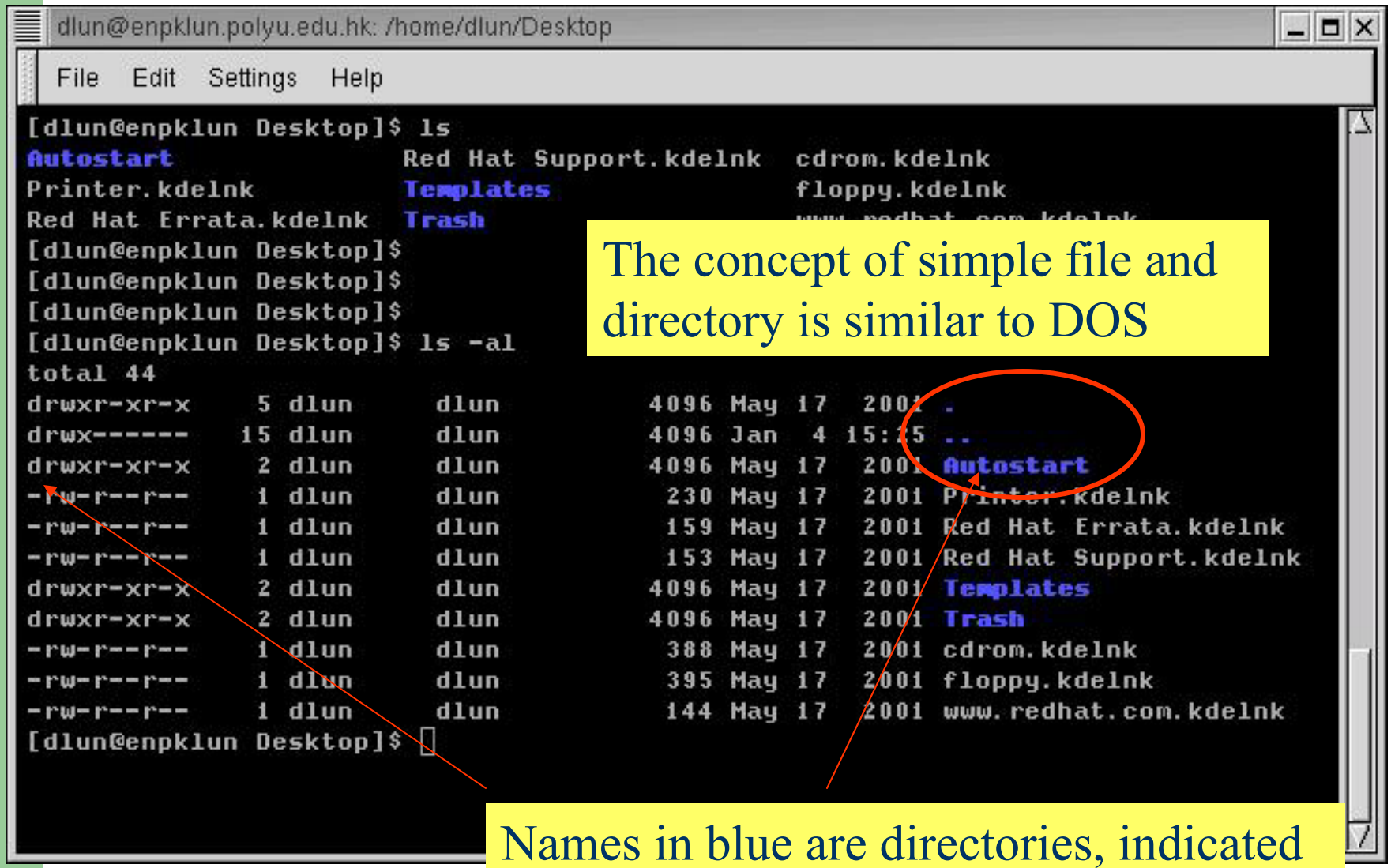
- Frequently used commands available in most shells:
  - ls : to show (list) the names of the file in the current directory
  - cd : change directory,
    - e.g. cd / change to the root directory
      cd .. change to the parent of that directory
  - cp : copy one file to another
    - e.g. cp abc.txt xyz.txt copy abc.txt to xyz.txt
  - rm : remove a file
  - man : ask for the manual (or help) of a command
    - e.g. man cd ask for the manual of the command cd
  - pwd : show the name of the present working directory
  - cat : to show the content of a text file
    - e.g. cat abc.txt show the content of abc.txt
  - whoami : to show the username of the current user

Case study: Linux

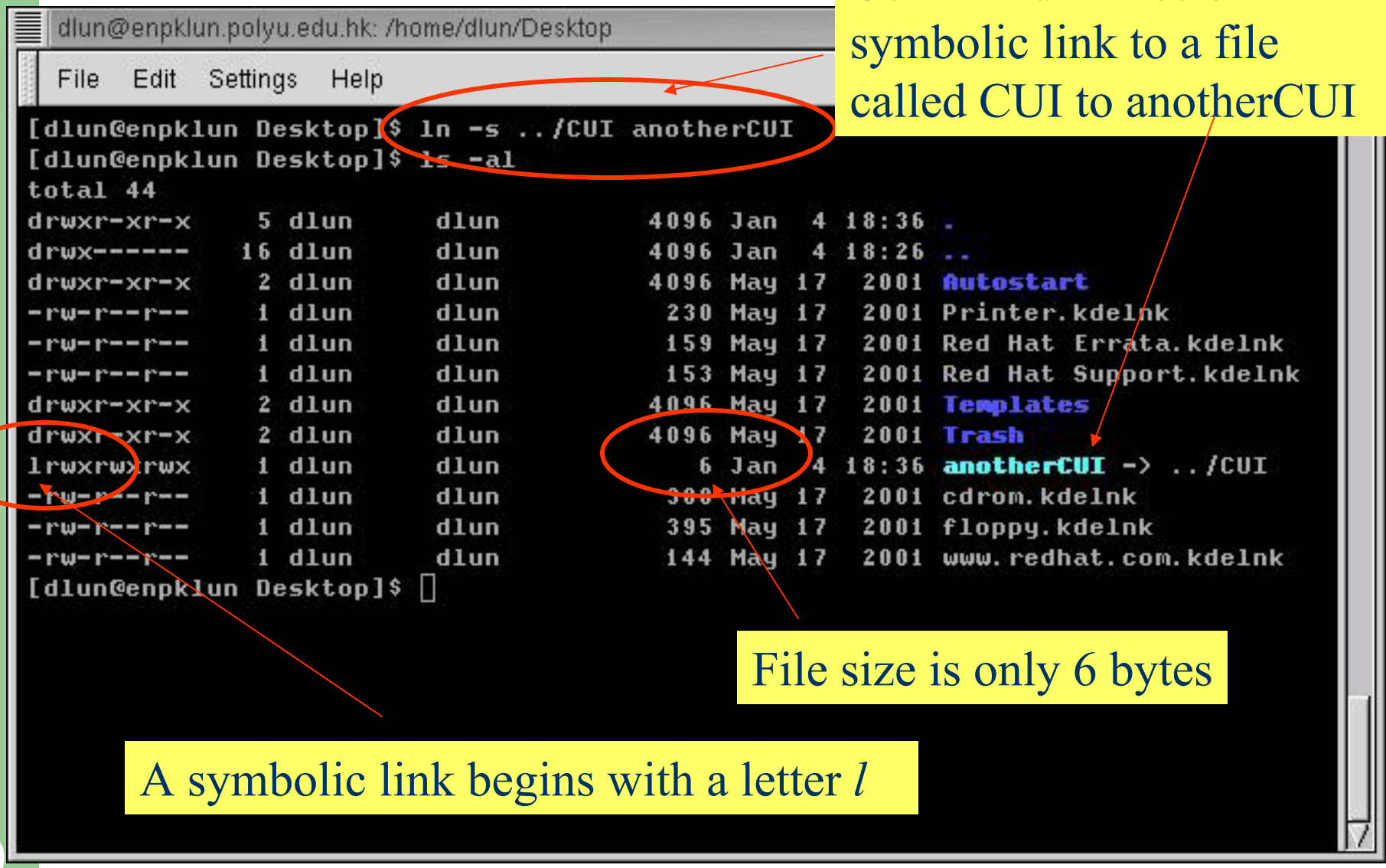
B.2 File management

Linux File Management

- In Linux, file is defined as simply the thing that deals with a sequence of bytes
- Hence everything are files
  - An ordinary file is a file; a directory is also file; a network card, a hard disk, any device are also files since they deal with a sequence of bytes
- Linux supports five types of files
  - simple/ordinary file (text file, c++ file, etc)
  - directory
  - symbolic (soft) link
  - special file (device)
  - named pipe (FIFO)


dlun@enpklun.polyu.edu.hk: /home/dlun/Desktop
File Edit Settings Help
[dlun@enpklun Desktop]$ ls
Autostart Red Hat Support.kdelnk cdrom.kdelnk
Printer.kdelnk Templates floppy.kdelnk
Red Hat Errata.kdelnk Trash
[dlun@enpklun Desktop]$
[dlun@enpklun Desktop]$
[dlun@enpklun Desktop]$
[dlun@enpklun Desktop]$ ls -al
total 44
drwxr-xr-x 5 dlun dlun 4096 May 17 2001 .
drwx------ 15 dlun dlun 4096 Jan 4 15:15 ..
drwxr-xr-x 2 dlun dlun 4096 May 17 2001 Autostart
-rw-r--r-- 1 dlun dlun 230 May 17 2001 Printer.kdelnk
-rw-r--r-- 1 dlun dlun 159 May 17 2001 Red Hat Errata.kdelnk
-rw-r--r-- 1 dlun dlun 153 May 17 2001 Red Hat Support.kdelnk
drwxr-xr-x 2 dlun dlun 4096 May 17 2001 Templates
drwxr-xr-x 2 dlun dlun 4096 May 17 2001 Trash
-rw-r--r-- 1 dlun dlun 388 May 17 2001 cdrom.kdelnk
-rw-r--r-- 1 dlun dlun 395 May 17 2001 floppy.kdelnk
-rw-r--r-- 1 dlun dlun 144 May 17 2001 www.redhat.com.kdelnk
[dlun@enpklun Desktop]$
The concept of simple file and directory is similar to DOS
Names in blue are directories, indicated by a letter d at the beginning of the line

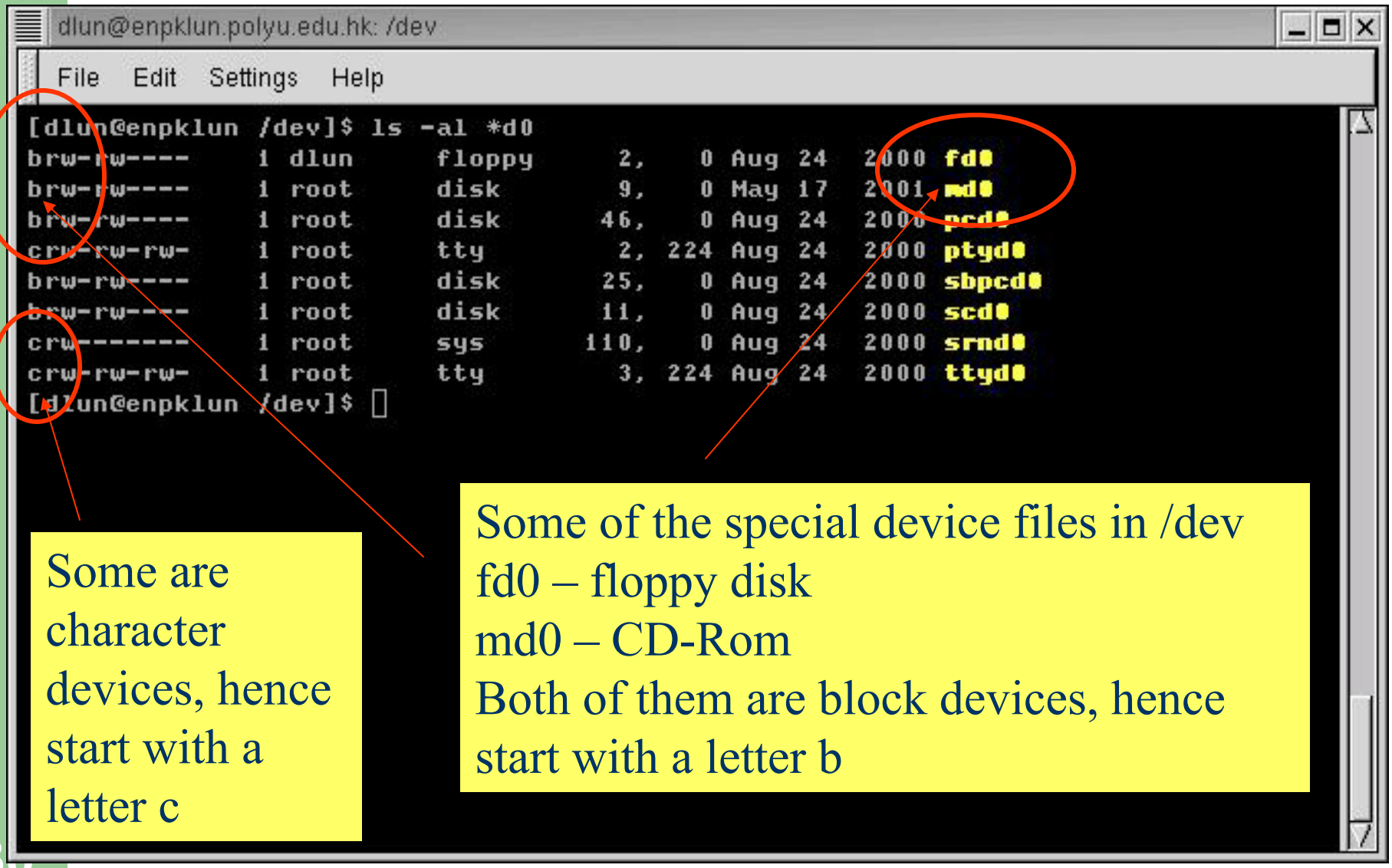
- Symbolic (soft) link
  - Not a real file, just a link to another file
  - Allow giving another name to a file without actually duplicates it – hence save memory space
- Special file (device)
  - Each hardware device, e.g. keyboard, hard disk, CD-ROM, etc is associated with at least one file
  - Usually store in /dev directory
  - Applications can read and write any devices by reading and writing their associate file – hence the access method is known as device independent
  - Divide into two types: character special files, e.g. keyboard, and block special files, e.g. disk



Command that sets a symbolic link to a file called CUI to anotherCUI

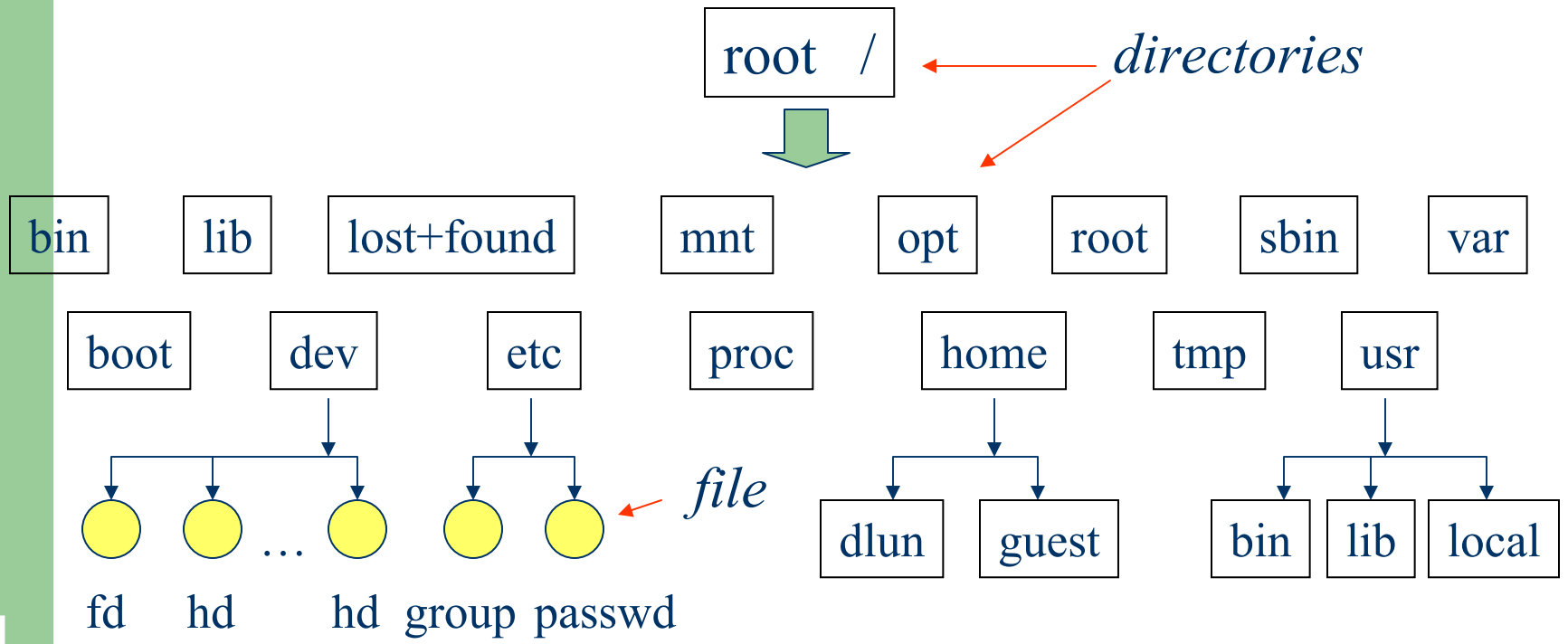
File size is only 6 bytes

A symbolic link begins with a letter *l*


dlun@enpklun.polyu.edu.hk: /dev
File Edit Settings Help
[dlun@enpklun /dev]$ ls -al *d0
brw-rw---- 1 dlun floppy 2, 0 Aug 24 2000 fd0
brw-rw---- 1 root disk 9, 0 May 17 2001 md0
brw-rw---- 1 root disk 46, 0 Aug 24 2000 pcd0
crw-rw-rw- 1 root tty 2, 224 Aug 24 2000 ptyd0
brw-rw---- 1 root disk 25, 0 Aug 24 2000 sbpcd0
brw-rw---- 1 root disk 11, 0 Aug 24 2000 scd0
crw------- 1 root sys 110, 0 Aug 24 2000 srnd0
crw-rw-rw- 1 root tty 3, 224 Aug 24 2000 ttyd0
[dlun@enpklun /dev]$
Some are character devices, hence start with a letter c
Some of the special device files in /dev
fd0 – floppy disk
md0 – CD-Rom
Both of them are block devices, hence start with a letter b

Linux File System Structure

- According to the File System Standard (FSSTND) proposed in 1994, every LINUX system should contain a set of standard files and directories



- Root Directory (/)
  - Top of the file system. Similar to \ in DOS
- /bin
  - Contain the binary (executable code) of most essential Linux commands, e.g. bash, cat, cp, ln, ls, etc.
- /boot
  - Contain all the files needed to boot the Linux system, including the binary of the Linux kernel. E.g., on Red Hat Linux 6.1, the kernel is in /boot/vmlinux-2.2.5-15 file
- /dev
  - Contain the special files for devices, e.g. fd0, hd0, etc.

- /etc
  - Contain host-specific files and directories, e.g. information about system configuration
  - /etc/passwd
    - This file contains login information of users in the system
    - For every user, one line of record is stored in the following format:

login_name : dummy_or_encrypted_password : user_ID : group_ID : user_info : home_directory : login_shell

- E.g. davis:x:134:105:James A Davis:/home/davis:/bin/bash
  - davis : login name
  - x : means that it is a dummy password. The encrypted password is stored in /etc/shadow. This field can also be used to store the actual encrypted password. In any case, the original (unencrypted) password cannot be seen by anyone, including the administrator
  - 134 : a user id given to that user. Range from 0 to 65535. 0 is assigned to super-user. 1 to 99 are reserved
  - 105 : a group id given to that user to indicate which group he belongs to. Range from 0 to 65535. 0 to 99 reserved
  - James A Davis : user info, usually user's full name
  - /home/davis : home directory of the user
  - /bin/bash : the location of the shell the user is using

- /home
  - Contain the home directories of every user in the system, e.g. dlun, guest, etc
- /lib
  - Store all essential libraries for different language compilers
- /lost+found
  - Contain all the files on the system not connected to any directory.
  - System administrator should determine the fate of the files in this directory

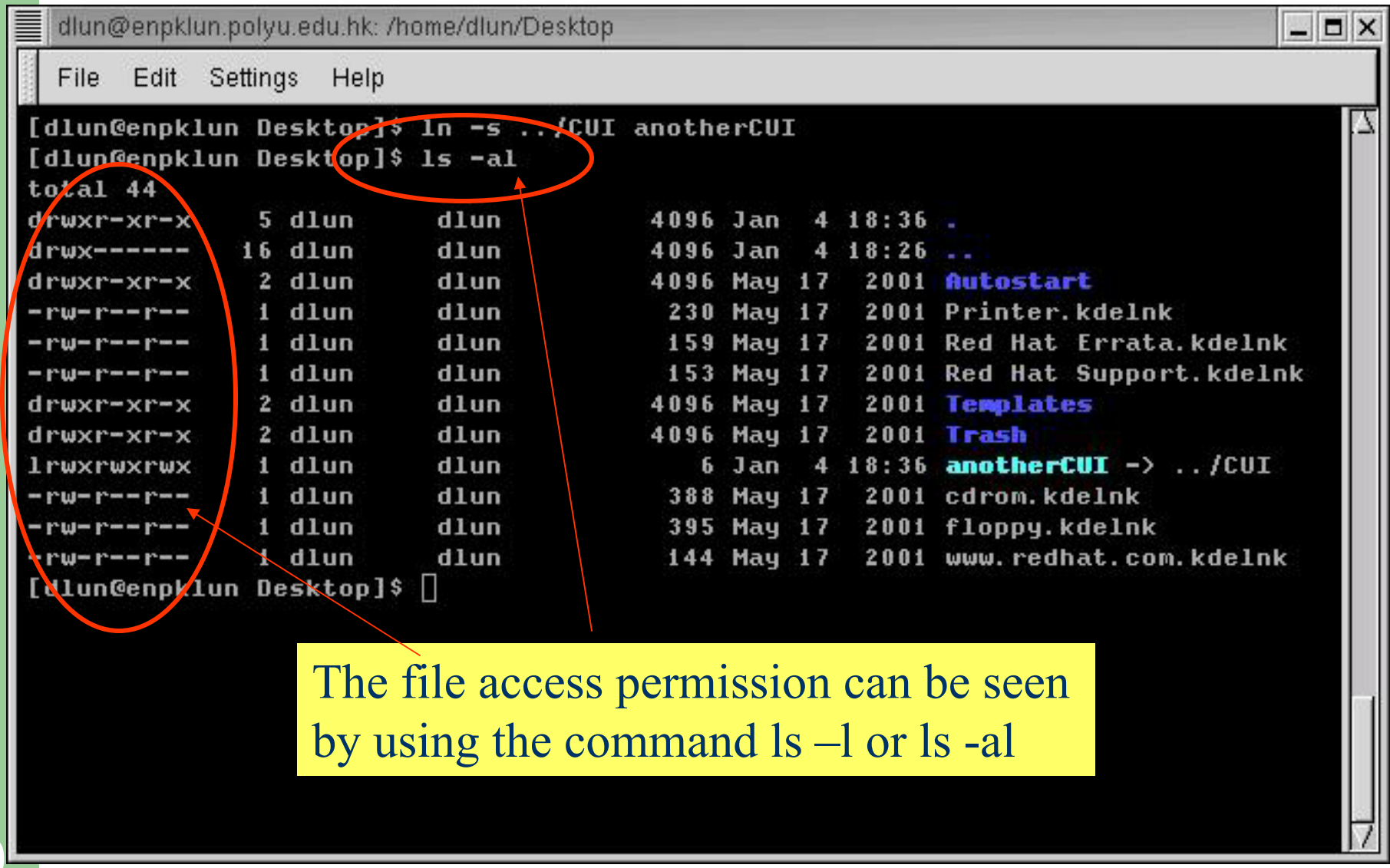
- /mnt
  - Use by system administrator to mount file systems temporarily by using the mount command
  - Before using any devices, they have to be mounted to the system for registration
  - For example, after mounting a CD-ROM, the file system in it will be mapped to /mnt/cdrom directory
  - User can then read and write files in the CD-ROM by accessing this directory
  - Similar to mapping a drive letter to a CD-ROM in Windows
  - Different from the special file in /dev. Special file is only a place where data of the CD-ROM is transferred or stored. No file system concept

- /opt
  - Use to install add-on software packages, e.g. star office, etc.
- /proc
  - Contain process and system information
- /root
  - Home directory of the user root, usually the administrator
- /sbin
  - The directories /sbin, /usr/sbin, and /usr/local/sbin contain system administration tools, utilities and general root only commands, such as halt, reboot and shutdown

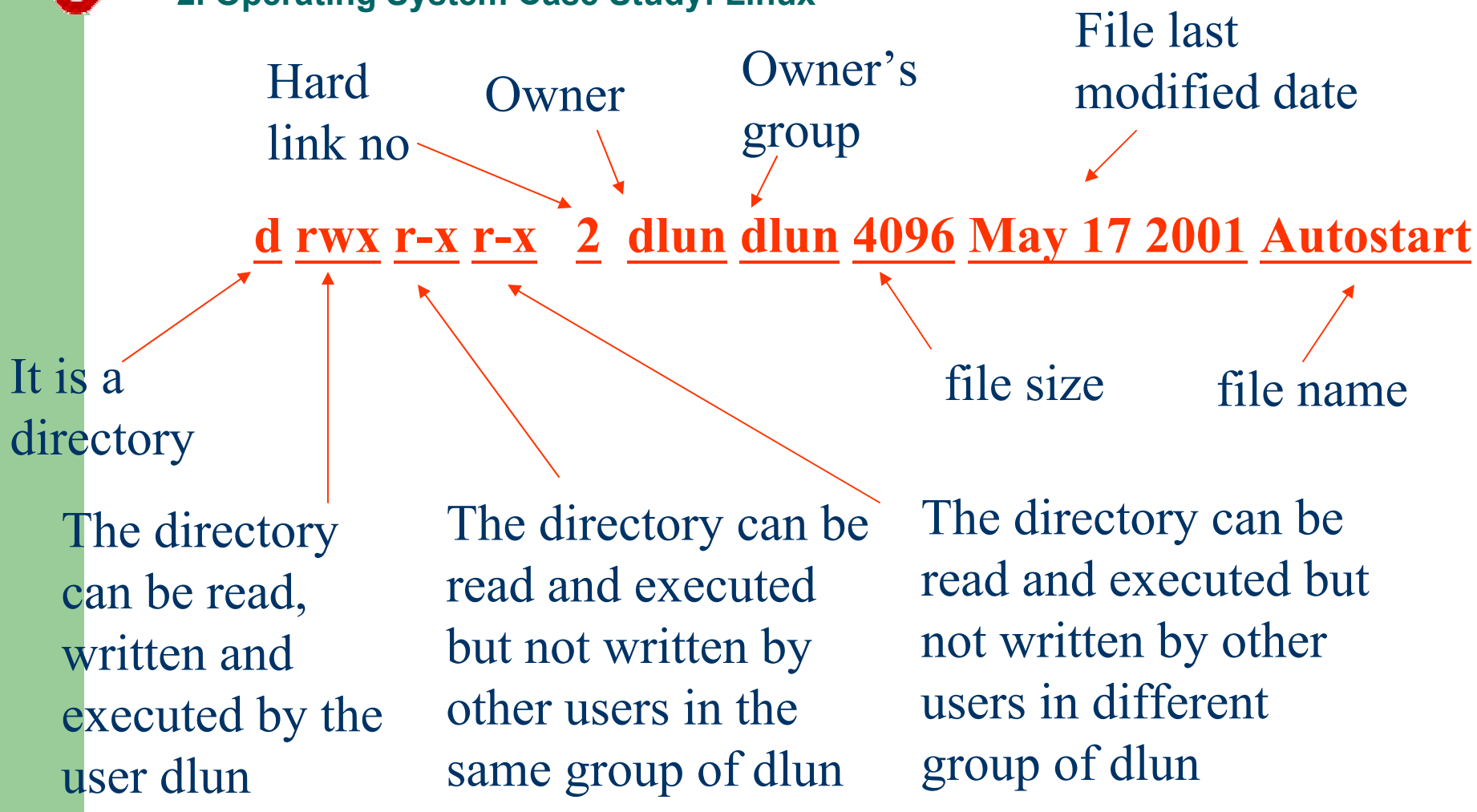
- /tmp
  - Contain temporary files. Usually files in this directory will be deleted from time to time to avoid the system fills with temp files
- /usr
  - One of the largest sections of the Linux file system
  - Contain read-only data that are shared between various users, e.g. the manual pages needed for the command man. Stored in /usr/man direcrtory
- /var
  - Contain data that keeps on changing as the system is running. E.g. /var/spool/mail directory keeps the mail of user

Linux File Access Privilege

- Linux is a multiuser system, the files of all users are stored in a single file structure
- Mechanism is required to restrict one user to access the files of another user, if he is not supposed to
- User can impose access permission to each file to restrict its access
- The term “access permission” refers to
  - read permission
  - write permission
  - execute permission



The file access permission can be seen by using the command ls –l or ls -al



The group of a user is assigned by the administrator when a user is added to the system

- Access permission can also be assigned to a directory
- Directory is also a file that contains the attributes of the files inside it
- If read permission is not given to a directory
  - cannot show the structure of this directory
  - e.g. cannot use ls
- If write permission is not given to a directory
  - cannot modify anything of the directory structure
  - e.g. cannot copy a file into this directory since it will modify the directory structure by adding one more file
- If execute permission is not given to a directory
  - nearly nothing can be done with this directory, even cd

- The access permission of a file or directory can be changed by using the command

chmod xyz filename/directory name

- xyz refers 3 digit in octal form
- E.g.

```
660 :  110 110 000
    ⇒  rw- rw- ---
545 :  101 100 101
    ⇒  r-x r-- r-x
```

dlun@enpklun.polyu.edu.hk: /home/dlun/Desktop/test/temp

File Edit Settings Help

temp does not have execution right

```
[dlun@enpklun test]$ ls -l
total 12
-rw-r--r--    1 dlun     dlun          395 Jan  7 16:36 floppy.kdelnk
drw-------    2 dlun     dlun         4096 Jan  9 11:06 temp
-rw-rw-r--    1 dlun     dlun           16 Jan  7 16:05 test1.txt
[dlun@enpklun test]$
[dlun@enpklun test]$
[dlun@enpklun test]$ cd temp
bash: cd: temp: Permission denied
[dlun@enpklun test]$
[dlun@enpklun test]$
[dlun@enpklun test]$ chmod 700 temp
[dlun@enpklun test]$
[dlun@enpklun test]$ ls -l
total 12
-rw-r--r--    1 dlun     dlun          395 Jan  7 16:36 floppy.kdelnk
drwx------    2 dlun     dlun         4096 Jan  9 11:06 temp
-rw-rw-r--    1 dlun     dlun           16 Jan  7 16:05 test1.txt
[dlun@enpklun test]$ cd temp
[dlun@enpklun temp]$
```

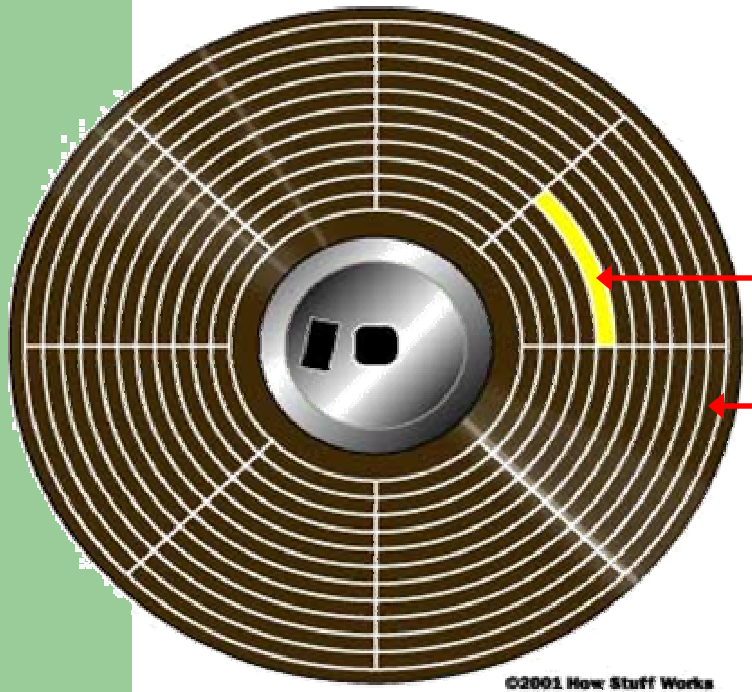
even cd is not workable

execution right is added

now we can change the directory to temp

File Storage in Linux

- Data storage on hard disk
  - Data in a hard disk are stored on a magnetic flat plate
  - Disk's surface needs to be partitioned and labeled so that computer can go directly to a specific point on it
  - Achieve by low level formatting the disk
    - Create magnetic concentric circles called tracks
    - Each track is split into smaller parts called sectors and numbered
- Each sector: hold 512 bytes data
  - E.g. 80 tracks (from outer to inner 0 .. 79), 18 sectors disk can store 80x18x512 bytes data.

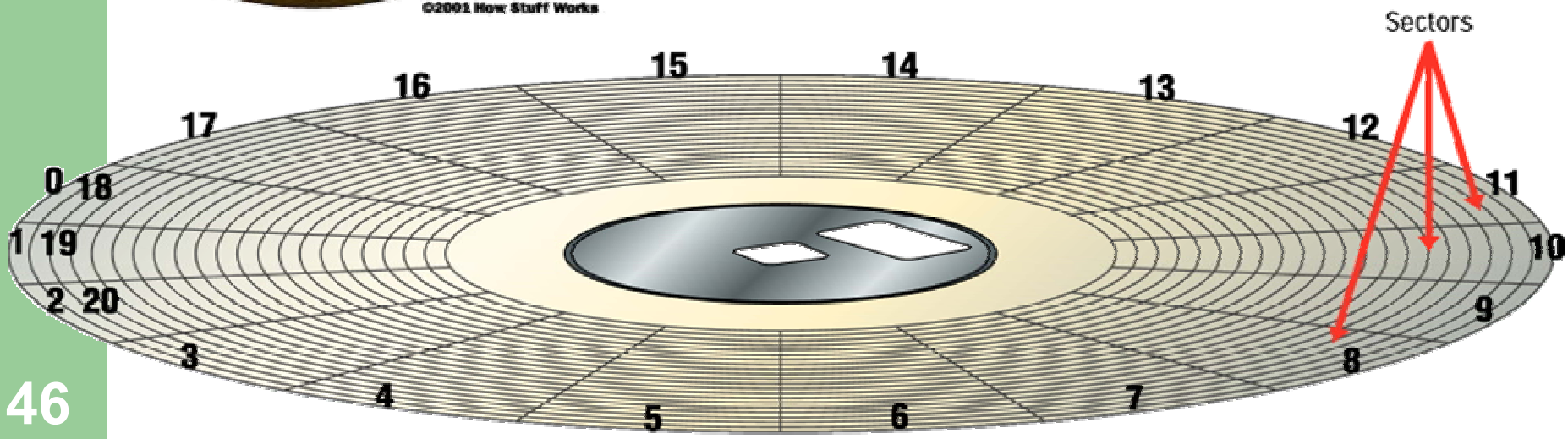


Formatted Disk

Sector

Track

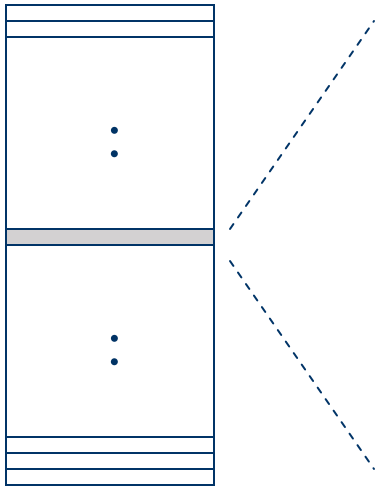
Density of data is higher for inner tracks than outer tracks



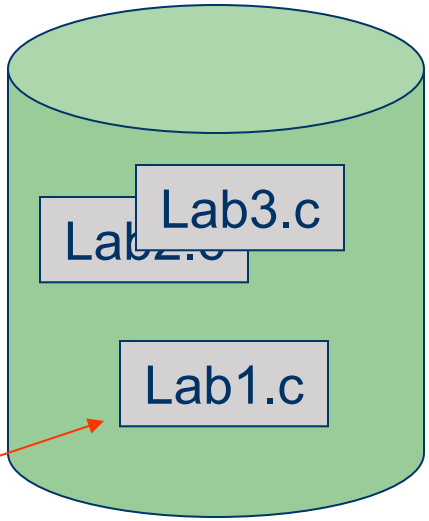
- Must read or write whole sector at a time
- OS allocates groups of sectors called cluster to files
- Files smaller than the cluster will still be allocated the whole cluster, but the rest left unused
- In Linux, every file is associated with an inode that records its location in the disk
- The inode of all files are put together in a data structure called inode table
- In the directory, every file is associated with a inode number that points to an entry of the inode table

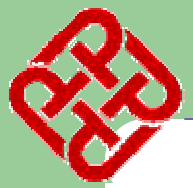
Contents of the directory /home/dlun

1076	…
2083	…
13059	lab1.c
17488	lab2.c
18995	lab3.c



Number of links
File mode
User ID
Time created
Time last updated
:
Location on disk





Case study: Linux

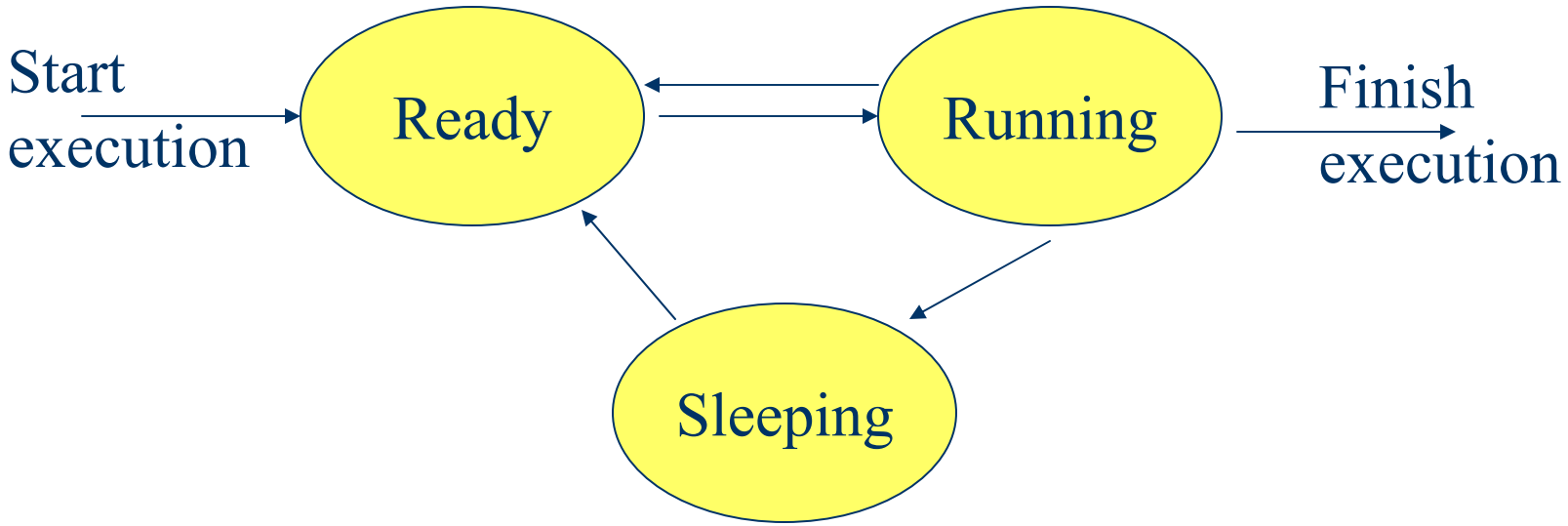
B.3 Process management

Linux Process Management

- Linux is a multitasking system
- Multiple programs can be executed at the same time
- Ultimately, a program needs to be executed by a CPU
- If there is only one CPU, how multiple programs can be executed at the same time?

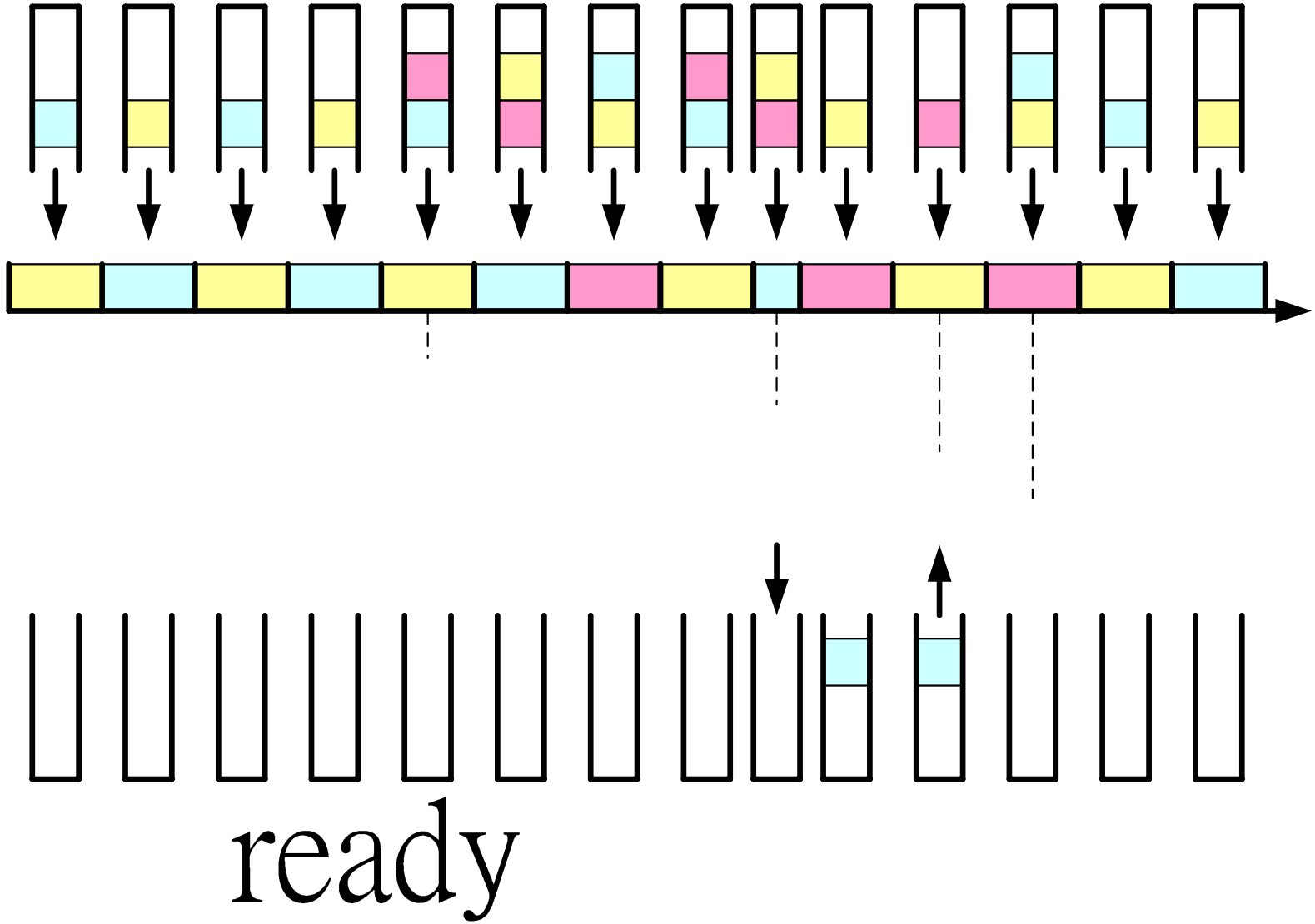
  ⇒ By time sharing
- That is, all programs are claimed to be executing. In fact, most of them are waiting for the CPU

- A program that is claimed to be executing is called a process
- For a multitasking system, a process has at least the following three states:



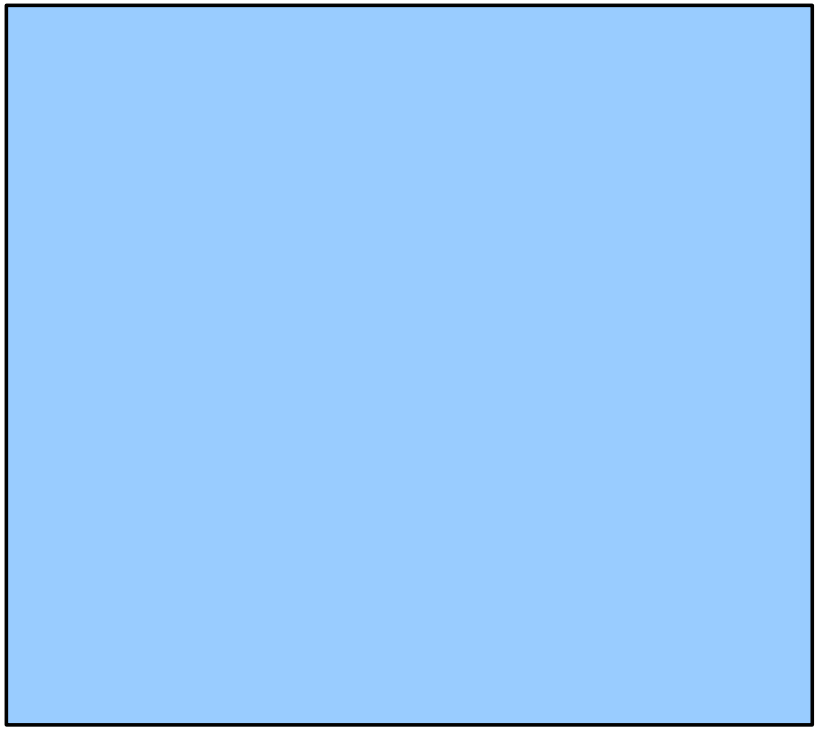
- Ready state
  - All processes that are ready to execute but without the CPU are at the ready state
  - If there is only 1 CPU in the system, all processes except one are at the ready state
- Running state
  - The process that actually possesses the CPU is at the running state
  - If there is only 1 CPU in the system, at most there is only one process is at the running state
- Sleeping state
  - The process that is waiting for other resources, e.g. I/O, is at the sleeping state

- Processes will alternatively get into the CPU one after the other (called the round robin scheme)
- A process will be “in” a CPU for a very short time (quantum)
  - For Linux, each quantum is about 100msec
- At the time that a process is selected to be “in” the CPU
  - It goes from ready state to running state
- After that, it will be swapped out
  - It goes from running state back to ready state
- Or it may due to the waiting of an I/O device, e.g. mouse
  - It goes from running state to sleeping state
- When obtaining the required resource
  - It goes from sleeping state to ready state

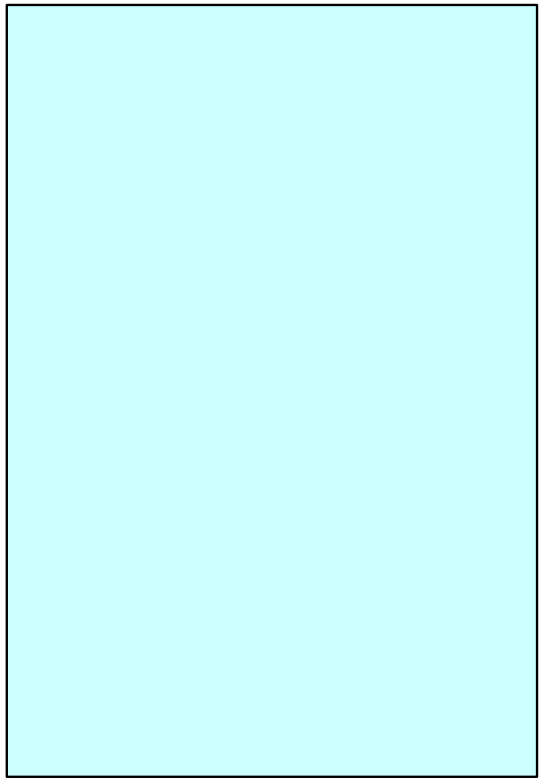

ready

- The mechanism to determine which process should "get into" the CPU is called Process scheduling
- For example,

Program A

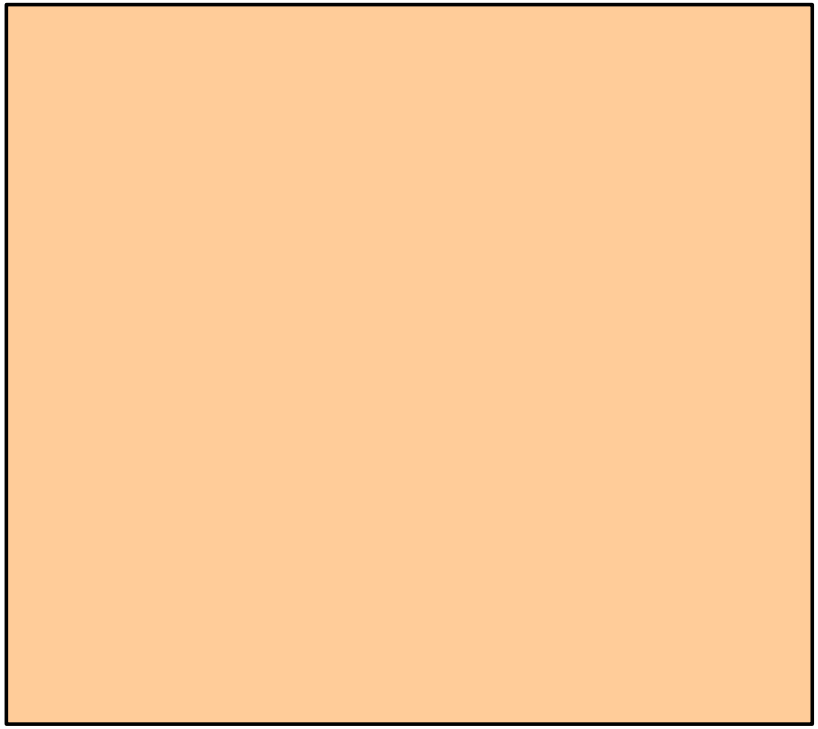


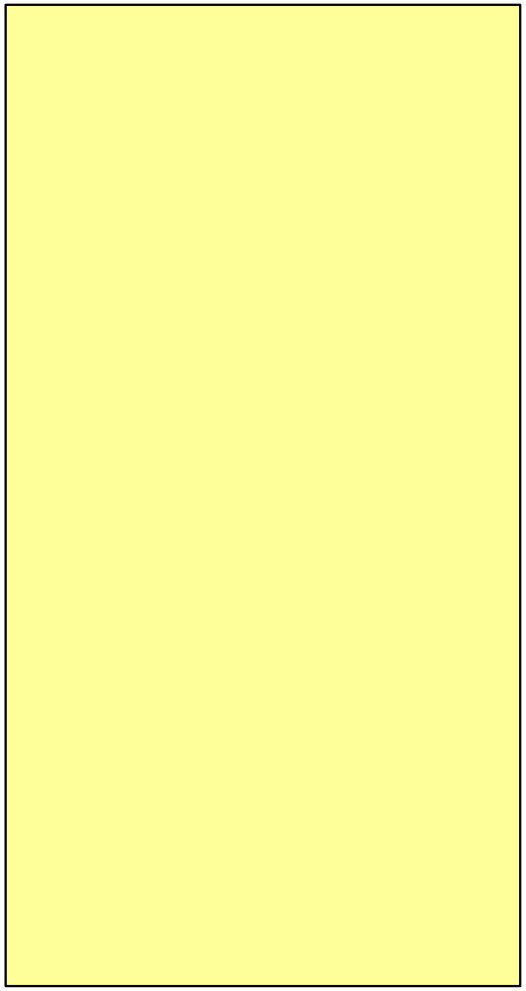
Actual sequence of operations



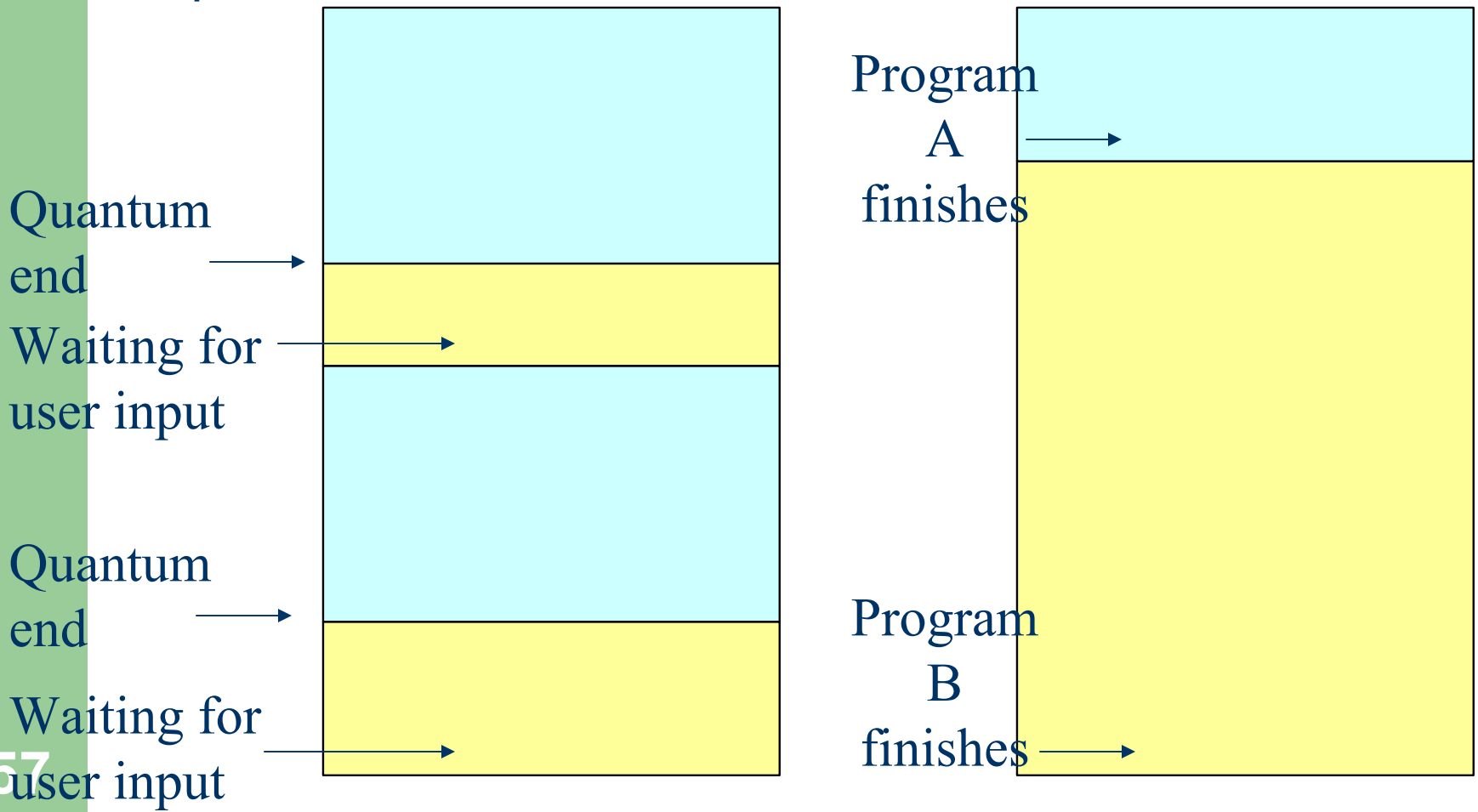
Actual sequence of operations

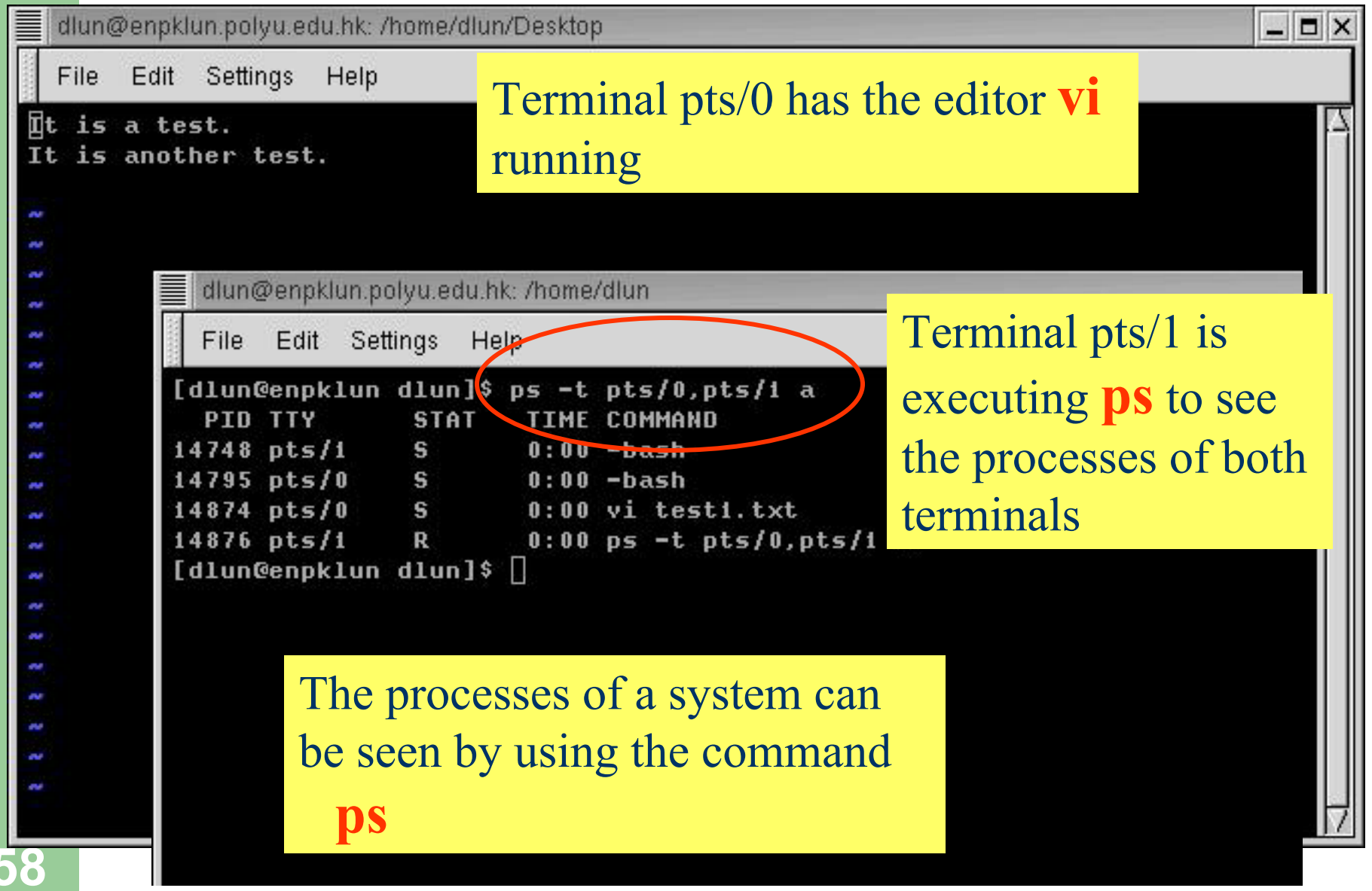
Program B





- Program A and B will be at the running state alternatively, depends on the quantum size and the availability of the required resource





Terminal pts/0 has the editor **vi** running

Terminal pts/1 is executing **ps** to see the processes of both terminals

The processes of a system can be seen by using the command

ps

```
  PID   TTY      STAT   TIME COMMAND
14748   pts/1    S      0:00 -bash
14795   pts/0    S      0:00 -bash
14974   pts/0    S      0:00 vi test1.txt
14876   pts/1    R      0:00 ps …
```

Process ID

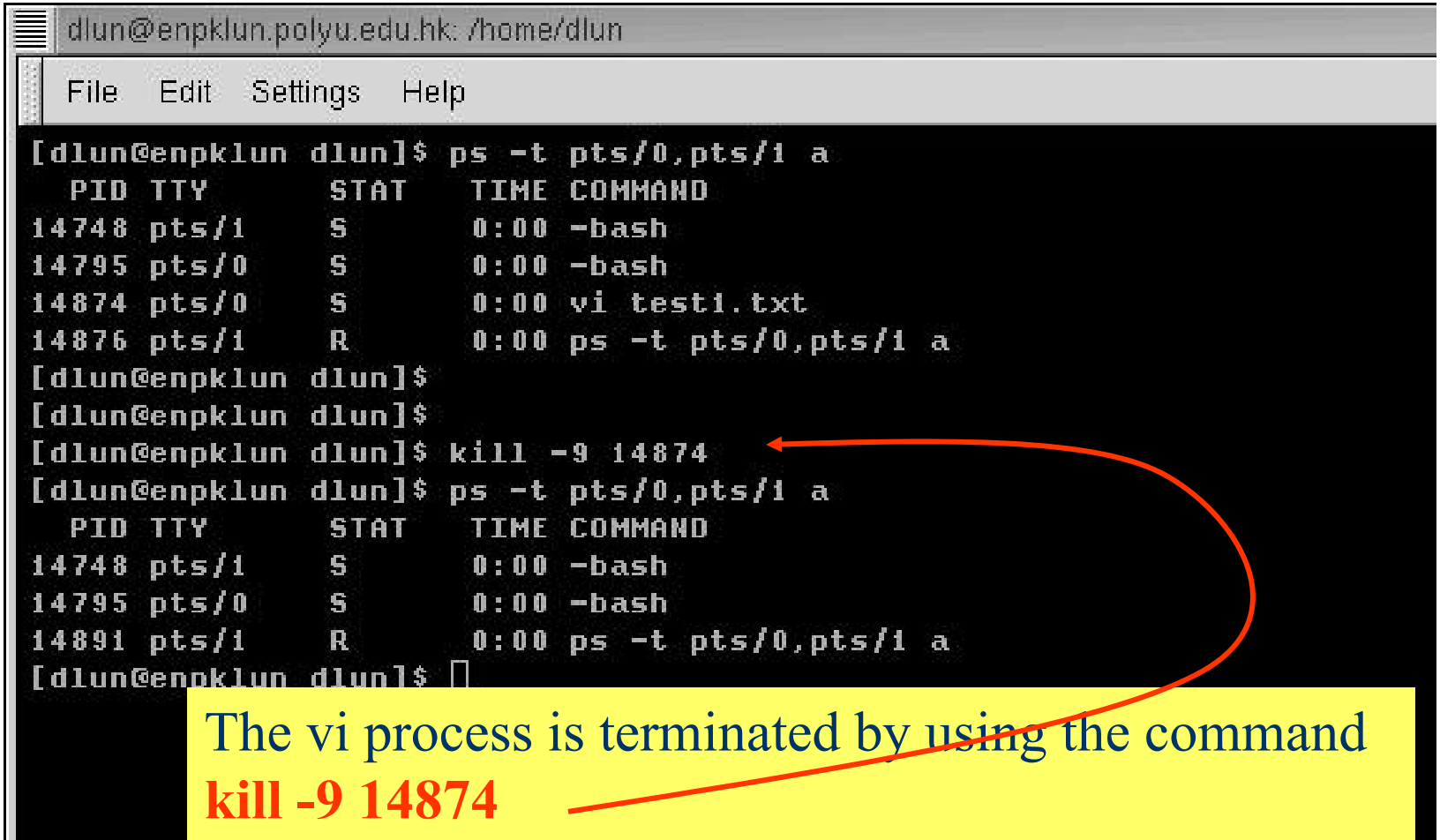
Terminal name

State:
S – Sleeping (waiting for input)
R – Running

How much time the process is continuously executing

- For the example above, both bash processes, which are the shell of both terminals, are waiting for the input of user. They must be in the sleeping state
- The vi process, which is an editor, is also waiting for the input of user. Hence it is also in sleeping state
- When ps reporting the processes in the system, it is the only process that is running. Hence it is in running state

- A process can be forced to terminate by using the command kill -9 *PID*



Basic Linux Commands

Learn basic commands for Linux, a free and open-source operating system that you can make changes to and redistribute.

What Is Linux?

Linux is an operating system's kernel. You might have heard of UNIX. Well, Linux is a UNIX clone. But it was actually created by Linus Torvalds from Scratch. Linux is free and open-source, that means that you can simply change anything in Linux and redistribute it in your own name! There are several Linux Distributions, commonly called “distros”.

1) Ubuntu Linux

2) Red Hat Enterprise Linux

3) Linux Mint

4) Debian

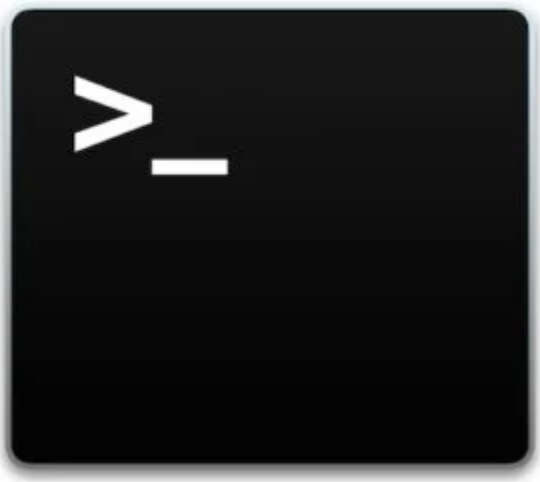
5) Fedora

Linux is Mainly used in servers. About 90% of the internet is powered by Linux servers. This is because Linux is fast, secure, and free! The main problem of using Windows servers are their cost. This is solved by using Linux servers. The OS that runs in about 80% of the smartphones in the world, Android, is also made from the Linux kernel. Most of the viruses in the world run on Windows, but not on Linux!

Linux Shell or “Terminal”

So, basically, a shell is a program that receives commands from the user and gives it to the OS to process, and it shows the output. Linux's shell is its main part. Its

distros come in GUI (graphical user interface), but basically, Linux has a CLI (command line interface). In this tutorial, we are going to cover the basic commands that we use in the shell of Linux.



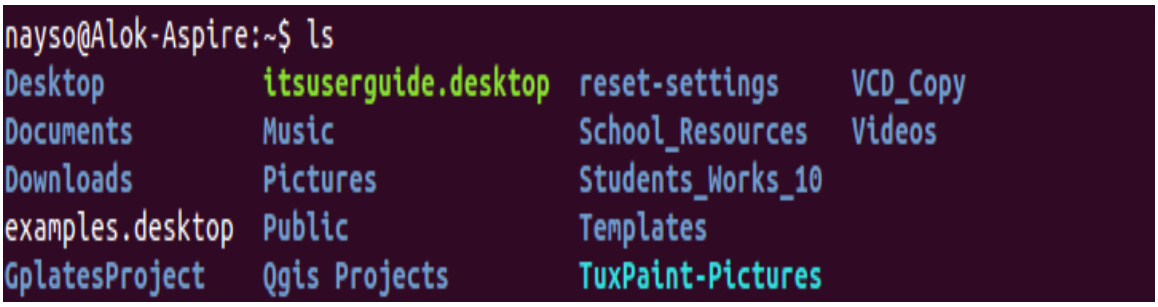
To open the terminal, press Ctrl+Alt+T in Ubuntu, or press Alt+F2, type in gnome-terminal, and press enter. In Raspberry Pi, type in lxterminal. There is also a GUI way of taking it, but this is better!

Basic Commands

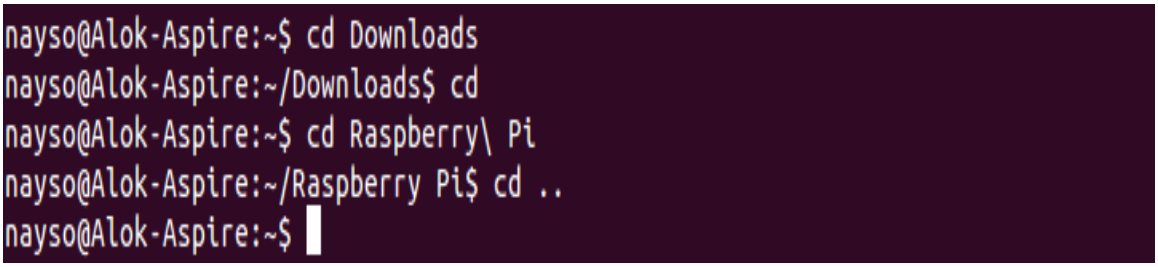
1. **pwd** — When you first open the terminal, you are in the home directory of your user. To know which directory you are in, you can use the “pwd” command. It gives us the absolute path, which means the path that starts from the root. The root is the base of the Linux file system. It is denoted by a forward slash(/). The user directory is usually something like "/home/username"

```
nayso@Alok-Aspire:~$ pwd
/home/nayso
```

2. **ls** — Use the "ls" command to know what files are in the directory you are in. You can see all the hidden files by using the command “ls -a”.



3. **cd** — Use the "cd" command to go to a directory. For example, if you are in the home folder, and you want to go to the downloads folder, then you can type in “cd Downloads”. Remember, this command is case sensitive, and you have to type in the name of the folder exactly as it is. But there is a problem with these commands. Imagine you have a folder named “Raspberry Pi”. In this case, when you type in “cd Raspberry Pi”, the shell will take the second argument of the command as a different one, so you will get an error saying that the directory does not exist. Here, you can use a backward slash. That is, you can use “cd Raspberry\ Pi” in this case. Spaces are denoted like this: If you just type “cd” and press enter, it takes you to the home directory. To go back from a folder to the folder before that, you can type “cd ..” . The two dots represent back.



4. **mkdir & rmdir** — Use the mkdir command when you need to create a folder or a directory. For example, if you want to make a directory called “DIY”, then you can type “mkdir DIY”. Remember, as told before, if you want to create a directory named “DIY Hacking”, then you can type “mkdir DIY\ Hacking”. Use rmdir to delete a directory. But rmdir can only be used to delete an empty directory. To delete a directory containing files, use rm.

```
nayso@Alok-Aspire:~/Desktop$ ls
nayso@Alok-Aspire:~/Desktop$ mkdir DIY
nayso@Alok-Aspire:~/Desktop$ ls
DIY
nayso@Alok-Aspire:~/Desktop$ rmdir DIY
nayso@Alok-Aspire:~/Desktop$ ls
nayso@Alok-Aspire:~/Desktop$
```

5. **rm** - Use the rm command to delete files and directories. Use "rm -r" to delete just the directory. It deletes both the folder and the files it contains when using only the rm command.

```
nayso@Alok-Aspire:~/Desktop$ ls
newer.py  New Folder
nayso@Alok-Aspire:~/Desktop$ rm newer.py
nayso@Alok-Aspire:~/Desktop$ ls
New Folder
nayso@Alok-Aspire:~/Desktop$ rm -r New\ Folder
nayso@Alok-Aspire:~/Desktop$ ls
nayso@Alok-Aspire:~/Desktop$
```

6. **touch** — The touch command is used to create a file. It can be anything, from an empty txt file to an empty zip file. For example, “touch new.txt”.

```
nayso@Alok-Aspire:~/Desktop$ ls
nayso@Alok-Aspire:~/Desktop$ touch new.txt
nayso@Alok-Aspire:~/Desktop$ ls
new.txt
```

7. **man & --help** — To know more about a command and how to use it, use the man command. It shows the manual pages of the command. For example, “man cd” shows the manual pages of the cd command. Typing in the command name and the argument helps it show which ways the command can be used (e.g., cd –help).

8. **cp** — Use the cp command to copy files through the command line. It takes two arguments: The first is the location of the file to be copied, the second is where to copy.

```
nayso@Alok-Aspire:~/Desktop$ ls /home/nayso/Music/
nayso@Alok-Aspire:~/Desktop$ cp new.txt /home/nayso/Music/
nayso@Alok-Aspire:~/Desktop$ ls /home/nayso/Music/
new.txt
```

9. **mv** — Use the mv command to move files through the command line. We can also use the mv command to rename a file. For example, if we want to rename the file “text” to “new”, we can use “mv text new”. It takes the two arguments, just like the cp command.

```
nayso@Alok-Aspire:~/Desktop$ ls
new.txt
nayso@Alok-Aspire:~/Desktop$ mv new.txt newer.txt
nayso@Alok-Aspire:~/Desktop$ ls
newer.txt
```

10. **locate** — The locate command is used to locate a file in a Linux system, just like the search command in Windows. This command is useful when you don't know where a file is saved or the actual name of the file. Using the -i argument with the command helps to ignore the case (it doesn't matter if it is uppercase or lowercase). So, if you want a file that has the word “hello”, it gives the list of all the files in your Linux system containing the word "hello" when you type in “locate -i hello”. If you remember two words, you can separate them using an asterisk (*). For example, to locate a file containing the words "hello" and "this", you can use the command “locate -i *hello*this”.

```
nayso@Alok-Aspire:~$ locate newer.txt
/home/nayso/Desktop/newer.txt
nayso@Alok-Aspire:~$ locate *DIY*Hacking*
/home/nayso/DIY Hacking
```

Intermediate Commands

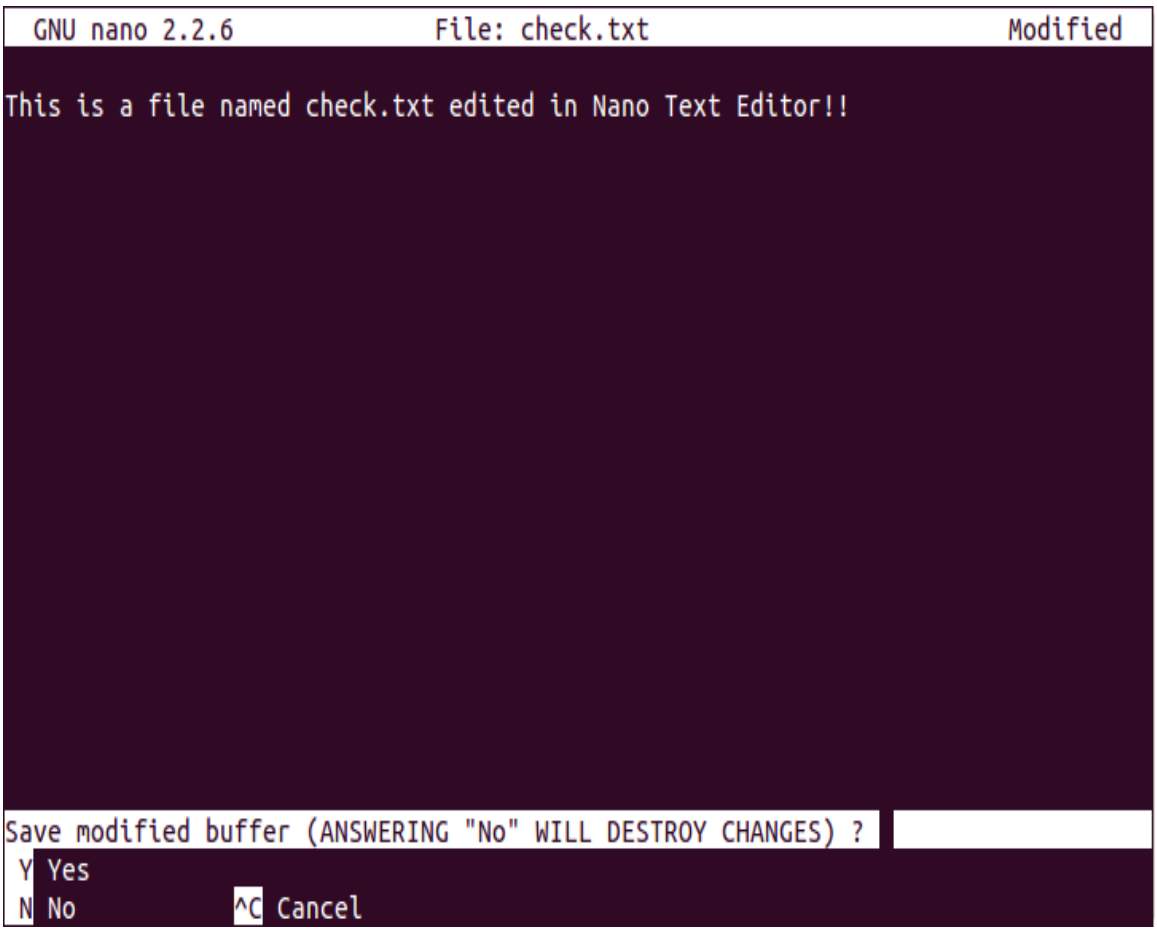
1. **echo** — The "echo" command helps us move some data, usually text into a file. For example, if you want to create a new text file or add to an already made text file, you just need to type in, “echo hello, my name is alok >> new.txt”. You do not need to separate the spaces by using the backward slash here, because we put in two triangular brackets when we finish what we need to write.

2. **cat** — Use the cat command to display the contents of a file. It is usually used to easily view programs.

```
nayso@Alok-Aspire:~/Desktop$ echo hello, my name is alok >> new.txt
nayso@Alok-Aspire:~/Desktop$ cat new.txt
hello, my name is alok
nayso@Alok-Aspire:~/Desktop$ echo this is another line >> new.txt
nayso@Alok-Aspire:~/Desktop$ cat new.txt
hello, my name is alok
this is another line
```

3. **nano, vi, jed** — nano and vi are already installed text editors in the Linux command line. The nano command is a good text editor that denotes keywords with color and can recognize most languages. And vi is simpler than nano. You

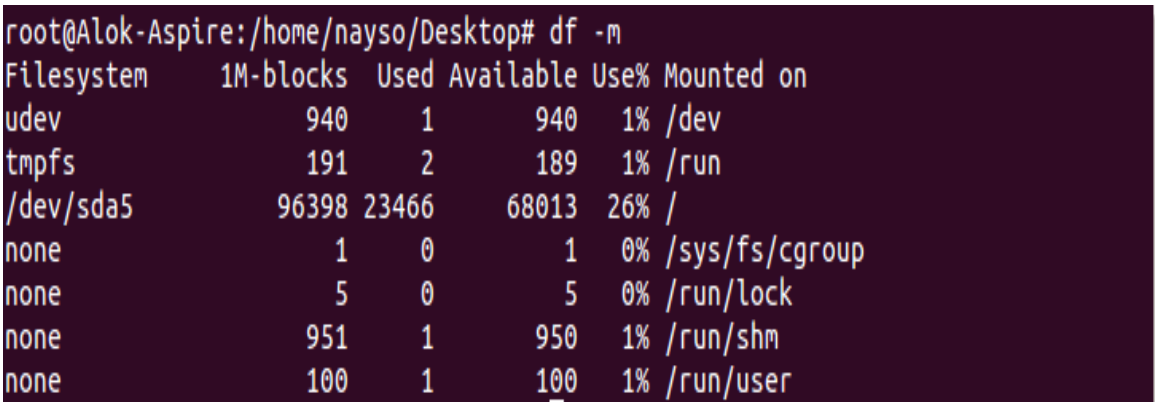
can create a new file or modify a file using this editor. For example, if you need to make a new file named "check.txt", you can create it by using the command “nano check.txt”. You can save your files after editing by using the sequence Ctrl+X, then Y (or N for no). In my experience, using nano for HTML editing doesn't seem as good, because of its color, so I recommend jed text editor. We will come to installing packages soon.



4. **sudo** — A widely used command in the Linux command line, sudo stands for "SuperUser Do". So, if you want any command to be done with administrative or root privileges, you can use the sudo command. For example, if you want to edit a file like viz. alsa-base.conf, which needs root permissions, you can use the command – sudo nano alsa-base.conf. You can enter the root command line using the command “sudo bash”, then type in your user password. You can also use the

command “su” to do this, but you need to set a root password before that. For that, you can use the command “sudo passwd”(not misspelled, it is passwd). Then type in the new root password.

5. **df** — Use the df command to see the available disk space in each of the partitions in your system. You can just type in df in the command line and you can see each mounted partition and their used/available space in % and in KBs. If you want it shown in megabytes, you can use the command “df -m”.



6. **du** — Use du to know the disk usage of a file in your system. If you want to know the disk usage for a particular folder or file in Linux, you can type in the command df and the name of the folder or file. For example, if you want to know

the disk space used by the documents folder in Linux, you can use the command “du Documents”. You can also use the command “ls -lah” to view the file sizes of all the files in a folder.

```
nayso@Alok-Aspire:~$ du Documents
516     Documents/DIYHacking
548     Documents
```

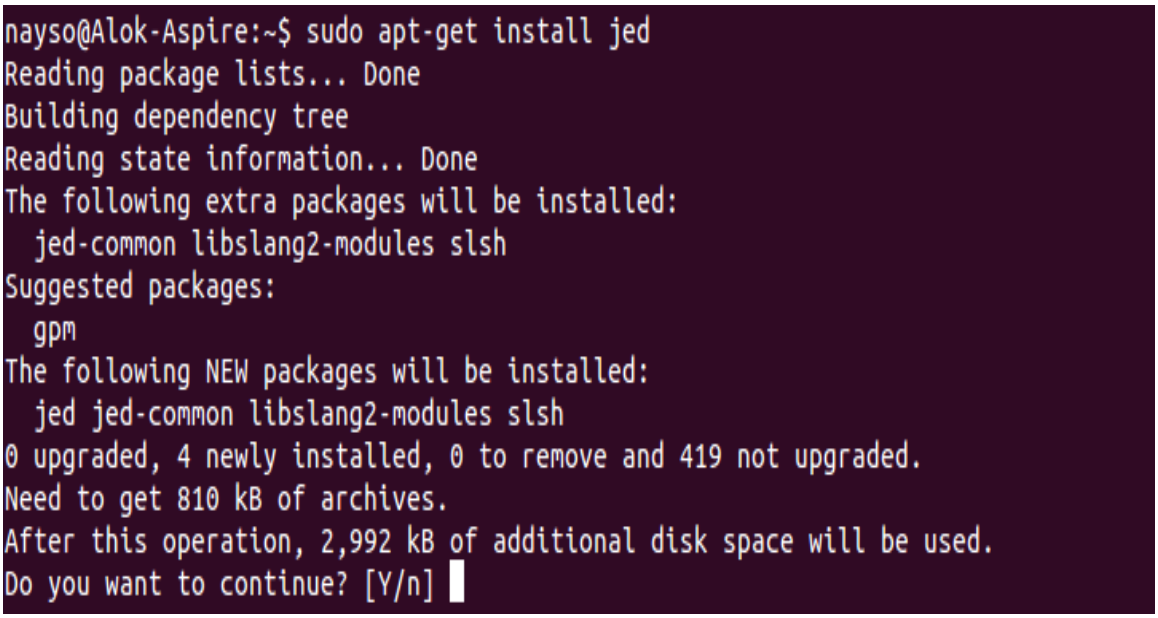
7. **tar** — Use tar to work with tarballs (or files compressed in a tarball archive) in the Linux command line. It has a long list of uses. It can be used to compress and uncompress different types of tar archives like .tar, .tar.gz, .tar.bz2,etc. It works on the basis of the arguments given to it. For example, "tar -cvf" for creating a .tar archive, -xvf to untar a tar archive, -tvf to list the contents of the archive, etc. Since it is a wide topic, here are some examples of tar commands.

8. **zip, unzip** — Use zip to compress files into a zip archive, and unzip to extract files from a zip archive.

9. **uname** — Use uname to show the information about the system your Linux distro is running. Using the command “uname -a” prints most of the information about the system. This prints the kernel release date, version, processor type, etc.

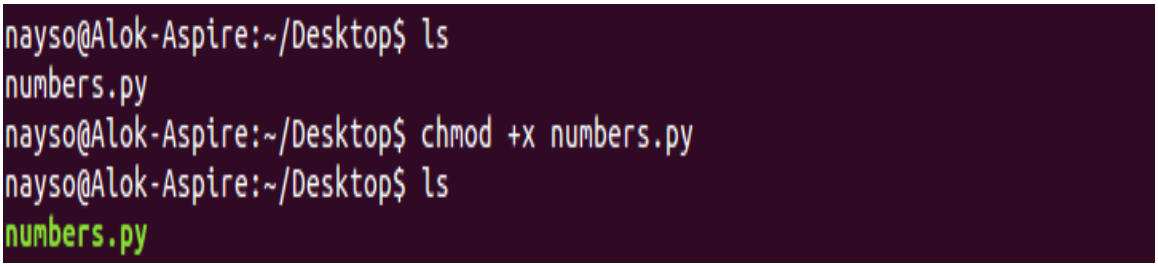
```
nayso@Alok-Aspire:~$ uname -a
Linux Alok-Aspire 4.4.0-22-generic #40~14.04.1-Ubuntu SMP Fri May 13 17:27:18 UT
C 2016 i686 i686 i686 GNU/Linux
```

10. **apt-get** — Use apt to work with packages in the Linux command line. Use apt-get to install packages. This requires root privileges, so use the sudo command with it. For example, if you want to install the text editor jed (as I mentioned earlier), we can type in the command “sudo apt-get install jed”. Similarly, any packages can be installed like this. It is good to update your repository each time you try to install a new package. You can do that by typing “sudo apt-get update”. You can upgrade the system by typing “sudo apt-get upgrade”. We can also upgrade the distro by typing “sudo apt-get dist-upgrade”. The command “apt-cache search” is used to search for a package. If you want to search for one, you can type in “apt-cache search jed”(this doesn't require root).

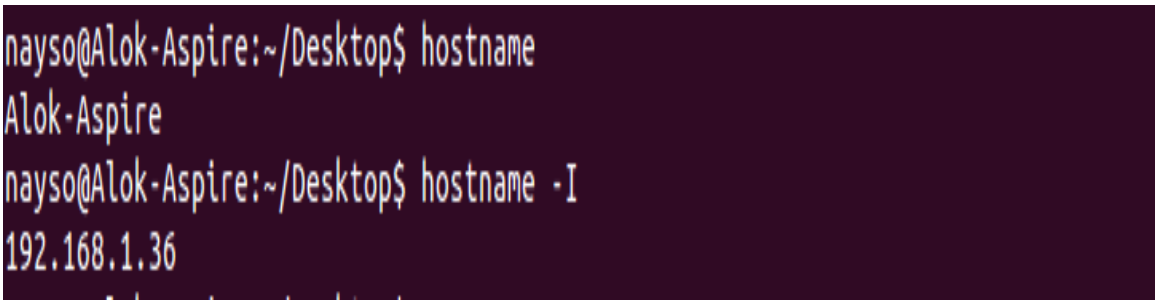


11. **chmod** — Use chmod to make a file executable and to change the permissions granted to it in Linux. Imagine you have a python code named numbers.py in your computer. You'll need to run “python numbers.py” every time you need to run it. Instead of that, when you make it executable, you'll just need to run “numbers.py” in the terminal to run the file. To make a file executable, you can use the command “chmod +x numbers.py” in this case. You can use “chmod 755 numbers.py” to give it root permissions or “sudo chmod +x numbers.py” for root executable. Here

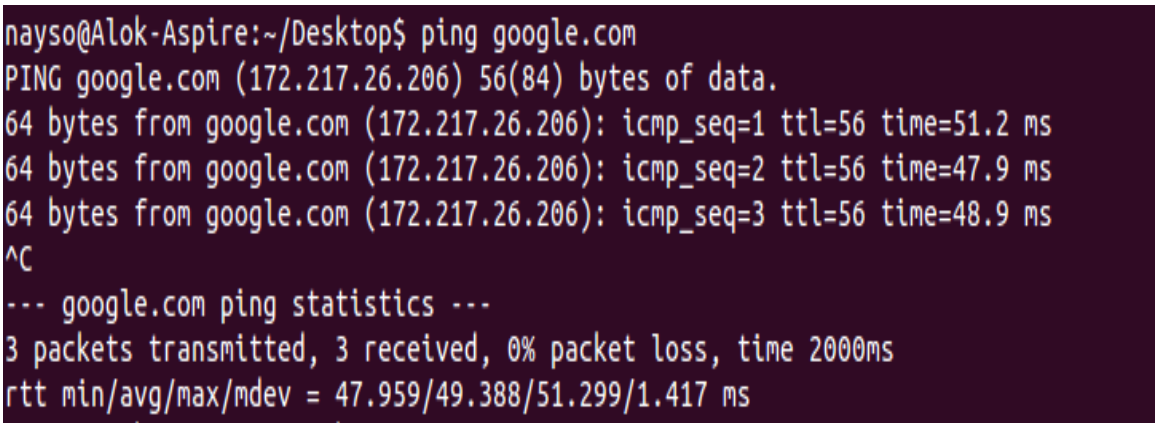
is some more information about the chmod command.



12. **hostname** — Use hostname to know your name in your host or network. Basically, it displays your hostname and IP address. Just typing “hostname” gives the output. Typing in “hostname -I” gives you your IP address in your network.



13. **ping** — Use ping to check your connection to a server. Wikipedia says, "Ping is a computer network administration software utility used to test the reachability of a host on an Internet Protocol (IP) network". Simply, when you type in, for example, “ping google.com”, it checks if it can connect to the server and come back. It measures this round-trip time and gives you the details about it. The use of this command for simple users like us is to check your internet connection. If it pings the Google server (in this case), you can confirm that your internet connection is active!



Tips and Tricks for Using Linux Command Line

1) You can use the clear command to clear the terminal if it gets filled up with too many commands.

2) TAB can be used to fill up in terminal. For example, You just need to type “cd Doc” and then TAB and the terminal fills the rest up and makes it “cd Documents”.

3) Ctrl+C can be used to stop any command in terminal safely. If it doesn't stop with that, then Ctrl+Z can be used to force stop it.

4) You can exit from the terminal by using the exit command.

5) You can power off or reboot the computer by using the command sudo halt and sudo reboot.

An Introduction to Device Drivers

Sarah Diesburg

COP 5641 / CIS 4930

Introduction

- Device drivers
  - Black boxes to hide details of hardware devices
  - Use standardized calls
    - Independent of the specific driver
  - Main role
    - Map standard calls to device-specific operations
  - Can be developed separately from the rest of the kernel
    - Plugged in at runtime when needed

The Role of the Device Driver

- Implements the *mechanisms* to access the hardware
  - E.g., show a disk as an array of data blocks
- Does not force particular *policies* on the user
  - Examples
    - Who many access the drive
    - Whether the drive is accessed via a file system
    - Whether users may mount file systems on the drive

Policy-Free Drivers

- A common practice
  - Support for synchronous/asynchronous operation
  - Be opened multiple times
  - Exploit the full capabilities of the hardware
- Easier user model
- Easier to write and maintain
- To assist users with policies, release device drivers with user programs

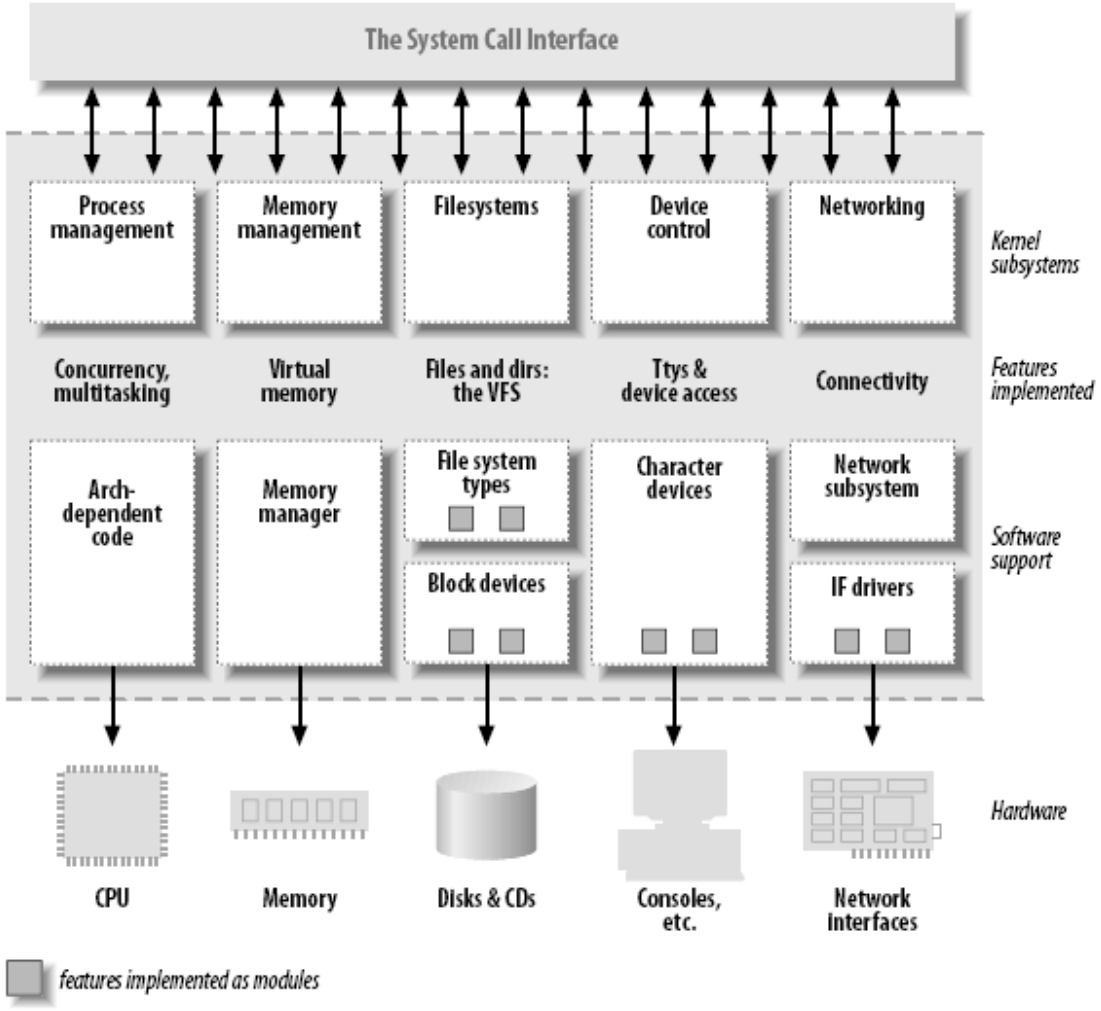
Splitting the Kernel

- Process management
  - Creates, destroys processes
  - Supports communication among processes
    - Signals, pipes, etc.
  - Schedules how processes share the CPU
- Memory management
  - Virtual addressing

Splitting the Kernel

- File systems
  - Everything in UNIX can be treated as a file
  - Linux supports multiple file systems
- Device control
  - Every system operation maps to a physical device
    - Few exceptions: CPU, memory, etc.
- Networking
  - Handles packets
  - Handles routing and network address resolution issues

Splitting the Kernel



Loadable Modules

- The ability to add and remove kernel features at runtime
- Each unit of extension is called a *module*
- Use **`insmod`** program to add a kernel module
- Use **`rmmod`** program to remove a kernel module

Classes of Devices and Modules

- Character devices
- Block devices
- Network devices
- Others

Character Devices

- Abstraction: a stream of bytes
  - Examples
    - Text console (**`/dev/console`**)
    - Serial ports (**`/dev/ttyS0`**)
  - Usually supports **`open`**, **`close`**, **`read`**, **`write`**
  - Accessed sequentially (in most cases)
  - Might not support file seeks
  - Exception: frame grabbers
    - Can access acquired image using **`mmap`** or **`lseek`**

Block Devices

- Abstraction: array of storage blocks
- However, applications can access a block device in bytes
  - Block and char devices differ only at the kernel level
  - A block device can host a file system

Network Devices

- Abstraction: data packets
- Send and receive packets
  - Do not know about individual connections
- Have unique names (e.g., **`eth0`**)
  - Not in the file system
  - Support protocols and streams related to packet transmission (i.e., no **`read`** and **`write`**)

Other Classes of Devices

- Examples that do not fit to previous categories:
  - USB
  - SCSI
  - FireWire
  - MTD

File System Modules

- Software drivers, not device drivers
- Serve as a layer between user API and block devices
- Intended to be device-independent

Security Issues

- Deliberate vs. incidental damage
- Kernel modules present possibilities for both
- System does only rudimentary checks at module load time
- Relies on limiting privilege to load modules
  - And trusts the driver writers
- Driver writer must be on guard for security problems

Security Issues

- Do not define security policies
  - Provide mechanisms to enforce policies
- Be aware of operations that affect global resources
  - Setting up an interrupt line
    - Could damage hardware
  - Setting up a default block size
    - Could affect other users

Security Issues

- Beware of bugs
  - Buffer overrun
    - Overwriting unrelated data
  - Treat input/parameters with utmost suspicion
  - Uninitialized memory
    - Kernel memory should be zeroed before being made available to a user
    - Otherwise, information leakage could result
      - Passwords

Security Issues

- Avoid running kernels compiled by an untrusted friend
  - Modified kernel could allow anyone to load a module

Version Numbering

- Every software package used in Linux has a release number
  - You need a particular version of one package to run a particular version of another package
  - Prepackaged distribution contains matching versions of various packages

Version Numbering

- Different throughout the years
- After version 1.0 but before 3.0
  - <major>.<minor>.<release>.<bugfix>
  - Time based releases (after two to three months)
- 3.x
  - Moved to 3.0 to commemorate 20th anniversary of Linux
  - <version>.<release>.<bugfix>
  - https://lkml.org/lkml/2011/5/29/204

License Terms

- GNU General Public License (GPL2)
  - GPL allows anybody to redistribute and sell a product covered by GPL
    - As long as the recipient has access to the source
    - And is able to exercise the same rights
  - Any software product derived from a product covered by the GPL be released under GPL

License Terms

- If you want your code to go into the mainline kernel
  - Must use a GPL-compatible license

Joining the Kernel Development Community

- The central gathering point
  - Linux-kernel mailing list
  - http://www.tux.org/lkml
- Chapter 20 of LKM further discusses the community and accepted coding style