Beni-Suif University
College of Computers and AI
Department of Computer Science

Lab Manual

CS322

Operating Systems

Preparing the scientific material

A.Prof. Ibrahim Al-Desouki

T.A. Muhammad Gamal Eldin

What We'll Build in This Course

Course Objectives

- **Process Management System**
- **CPU Scheduling Simulator**
- **Inter-Process Communication (IPC) Tools**
- **Deadlock Detection System**
- **Memory Management System**
- **File Management System**
- **Linux Command Toolkit**

Intended Learning Outcomes (ILOs)

By the end of this course, students will be able to:

1. **Explain fundamental concepts of Operating Systems**
2. **Analyze CPU scheduling algorithms**
3. **Implement process management techniques**
4. **Design and apply inter-process communication (IPC) mechanisms**
5. **Evaluate and prevent deadlocks**
6. **Apply memory management techniques**
7. **Perform file handling operations and understand file system structure**
8. **Use Linux commands and shell scripting**
9. **Demonstrate problem-solving skills**

Weekly Breakdown

Week 1: Introduction to operating systems

- **What is operating system?**
- **Operating System Goals.**
- **What is operating system Do ?**

Week 2: Operating Systems Structures

- **Operating System Services**
- **User Operating System Interface**
- **System Calls**
- **Types of System Calls**
- **System Programs**
- **Operating System Design and Implementation**
- **Operating System Structure**

Week 3: Linux Commands

- **Users and group, files , searching.**
- **Networking , Process , System Management.**

Week 4: Threads

- **Multicore Programming**
- **Multithreading Models**
- **Benefits of Threads**
- **Thread Libraries**

Week 5,6: cpu scheduling

- **Basic Concepts**
- **Scheduling Criteria**
- **Scheduling Algorithms**

Week 7: Process Synchronization

- **Introduction.**
- **The Critical Section Problem.**

Week 8,9,10: DeadLocks

- **Introduction to Deadlock.**
- **Deadlock Characterization.**
- **Methods for Handling Deadlocks.**
- **Deadlocks Avoidance.**
- **Deadlocks Detection and Recovery.**

Contents

9	Deadlocks
10	Deadlocks

Beni-Suif University
College of Computers and AI
Department of Computer Science

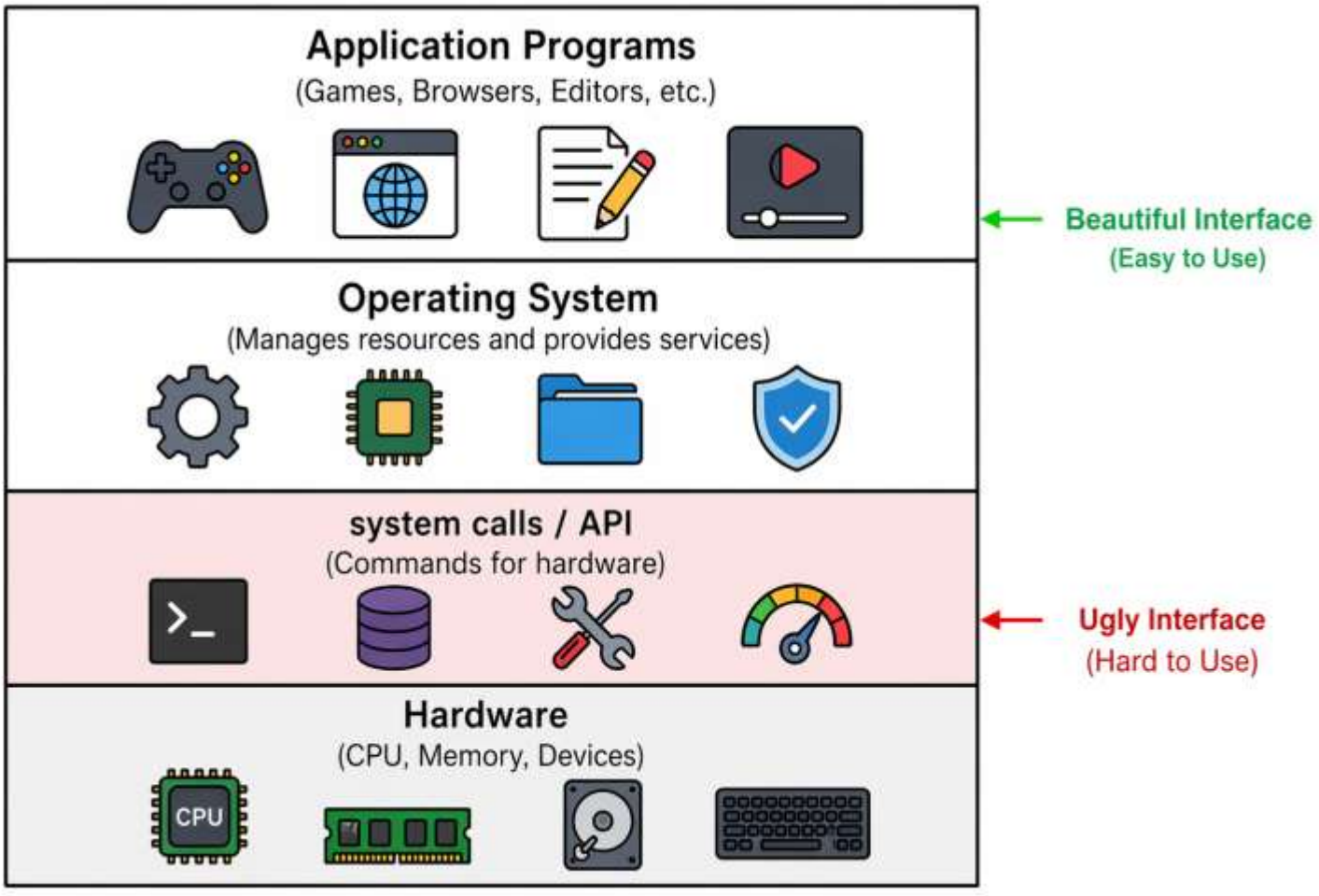
Lab Manual

Introduction to Operating Systems

Lab - 1
Introduction to Operating Systems

What is Operating Systems?

is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.



Operating system goals:

• Execute user programs and make solving user problems easier.

• Make the computer system convenient to use.

• Use the computer hardware in an efficient manner

What Operating Systems Do?

User View

• Users want convenience, ease of use and good performance.

➢Don't care about resource utilization.

• But shared computer such as mainframe or minicomputer must keep all users happy

System View

• From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a resource allocator.

• A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources.

OS is a resource allocator

➢ Manages all resources.

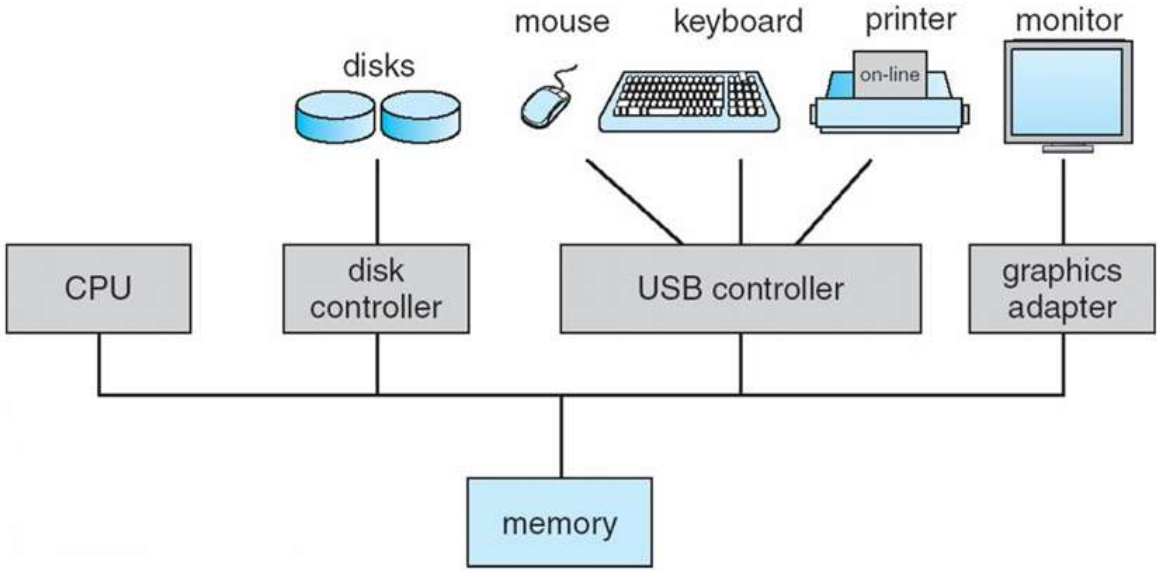
➢Decides between conflicting requests for efficient and fair resource use.

OS is a control program

➢ Controls execution of programs to prevent errors and improper use of the computer

Computer System Organization

• A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory



Computer Startup

• bootstrap program is loaded at power-up or reboot

➢Typically stored in ROM or electrically erasable programmable read-only memory (EPROM), generally known as firmware.

➢Initializes all aspects of system.

➢Loads operating system kernel and starts execution

Interrupts

• The occurrence of an event is usually signaled by an interrupt from either the hardware or the software.

➢Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus.

➢Software may trigger an interrupt by executing a special operation called a system call (also called a monitor call).

• Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common.

• Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines.

• Interrupt architecture must save the address of the interrupted instruction.

• A trap or exception is a software-generated interrupt caused either by an error or a user request.

• An operating system is interrupt driven.

Review The basic unit of computer storage is the bit.

A bit can contain one of two values, 0 and 1.

All other storage in a computer is based on collections of bits.

Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few.

A byte is 8 bits, and on most computers it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte.

A less common term is word, which is a given computer architecture's native unit of data.

A word is made up of one or more bytes. For example, a computer that has 64-bit registers and 64 bit memory addressing typically has 64-bit (8-byte) words.

A computer executes many operations in its native word size rather than a byte at a time.

Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes.

A kilobyte, or KB, is 1,024 bytes a megabyte, or MB, is 1,0242 bytes a gigabyte, or GB, is 1,0243 bytes a terabyte, or TB, is 1,0244 bytes a petabyte, or PB, is 1,0245 bytes Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes.

Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).

• Main memory– only large storage media that the CPU can access directly.

➢Random access

➢Typically volatile

• Secondary storage – extension of main memory that provides large nonvolatile storage capacity.

• Hard disks – rigid metal or glass platters covered with magnetic recording material.

➢Disk surface is logically divided into tracks, which are subdivided into sectors.

➢The disk controller determines the logical interaction between the device and the computer.

• Solid-state disks– faster than hard disks, nonvolatile.

➢Various technologies.

➢Becoming more popular

Beni-Suif University
College of Computers and AI
Department of Computer Science

Lab Manual

Operating Structures System

Lab - 2
Operating Structures System

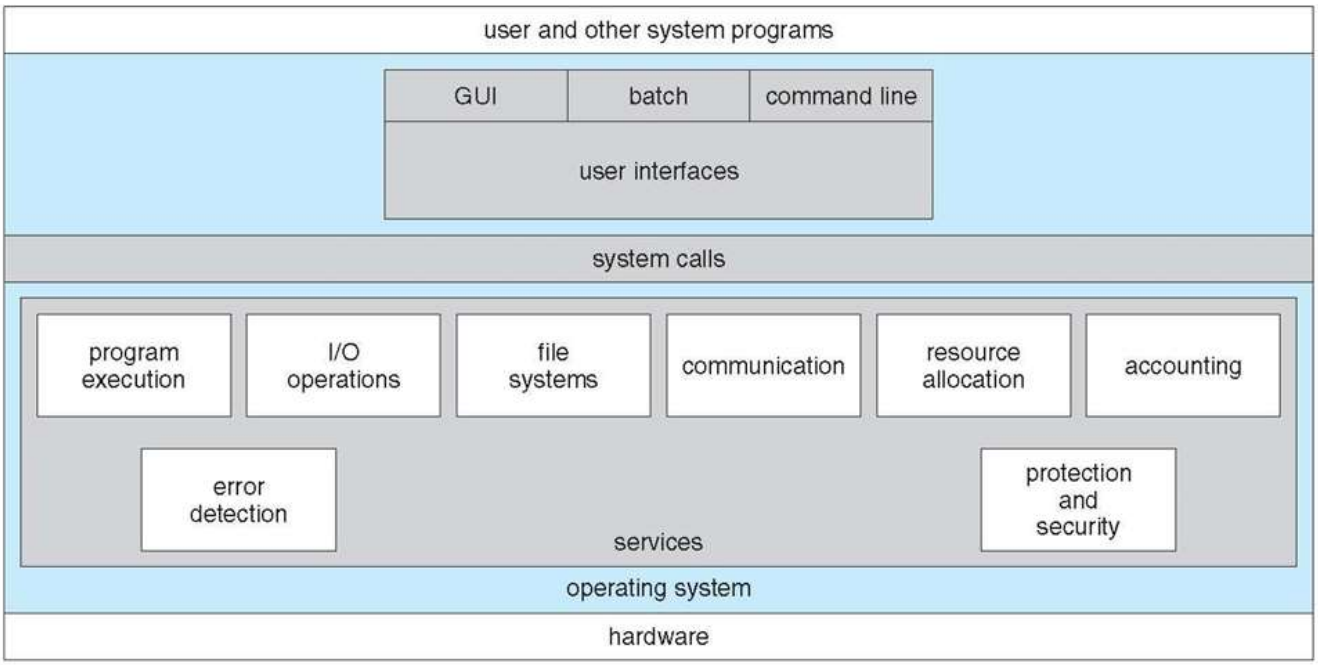
Lab - 2
Operating Structures System

1. Operating System Structures

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure

Operating System Services

Operating systems provide an environment for execution of programs and services to programs and users



• One set of operating-system services provides functions that are helpful to the user

➢ User interface - Almost all operating systems have a user interface (UI).
• Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch.

➢ Program execution - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error).

➢ I/O operations - A running program may require I/O, which may involve a file or an I/O device.

One set of operating-system services provides functions that are helpful to the user

➢File-system manipulation – The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

➢Communications– Processes may exchange information, on the same computer or between computers over a network.
• Communications may be via shared memory or through message passing (packets moved by the OS).

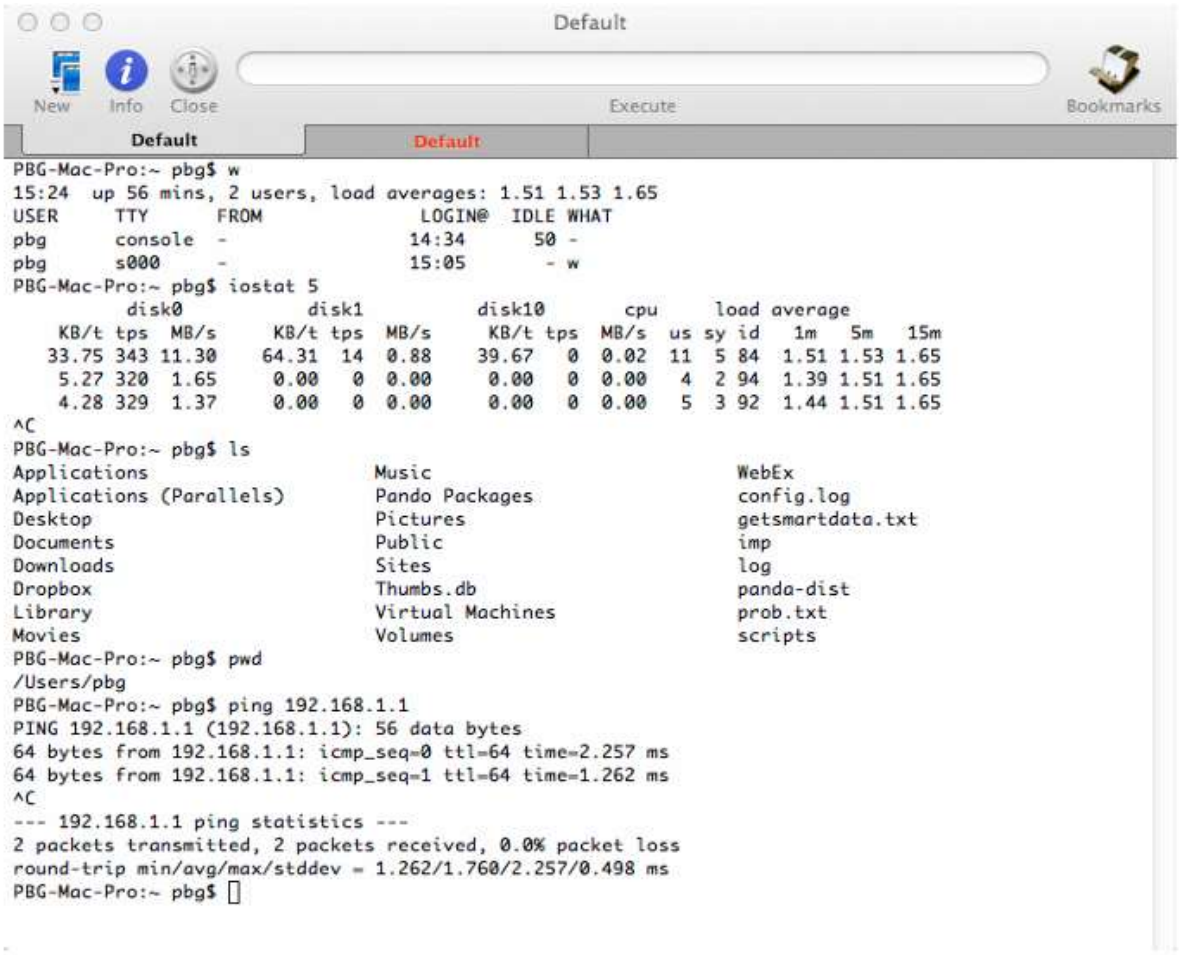
One set of operating-system services provides functions that are helpful to the user: (3/3)
➢Error detection – OS needs to be constantly aware of possible errors.
• May occur in the CPU and memory hardware, in I/O devices, in user program.
• For each type of error, OS should take the appropriate action to ensure correct and consistent computing.
• Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system.

User Operating System Interface

CLI or command interpreter allows direct command entry.

- ➢ Sometimes implemented in kernel, sometimes by systems program.
- ➢ Sometimes multiple flavors implemented – shells
- ➢ Primarily fetches a command from user and executes it.
- ➢ Sometimes commands built-in, sometimes just names of programs.

• Bourne Shell Command Interpreter



• Graphics User Interface (GUI)

• User-friendly desktop metaphor interface

➢ Usually mouse, keyboard, and monitor.

➢ Icons represent files, programs, actions, etc.

➢ Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder).

• Many systems now include both CLI and GUI interfaces.

System Calls

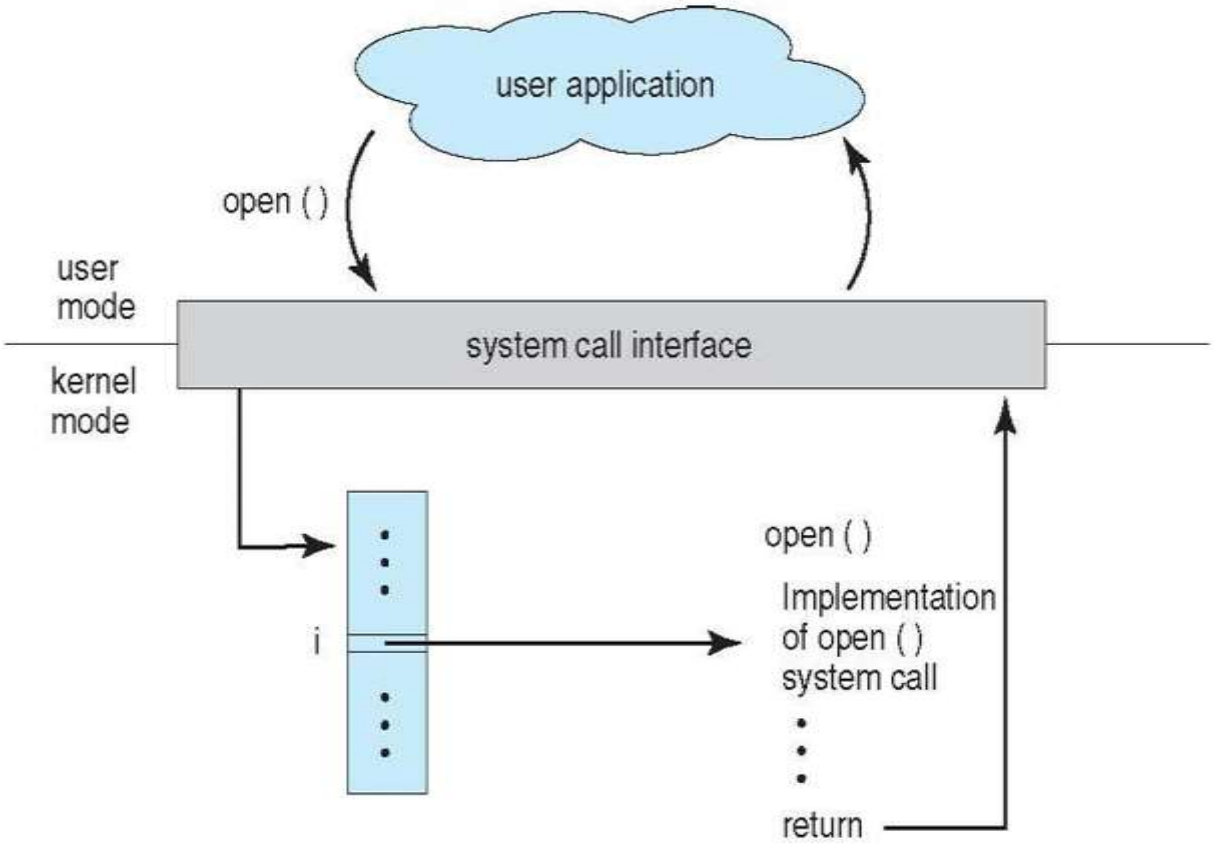
• Programming interface to the services provided by the OS.

• Typically written in a high-level language (C or C++).

• Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use.

• Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM).

Example : System call sequence to copy the contents of one file to another file.



Example System Call Sequence

Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally

System Programs

• System programs provide a convenient environment for program development and execution.

• They can be divided into:

➢ File manipulation

➢ Status information sometimes stored in a File modification

➢ Programming language support

➢ Program loading and execution

➢ Communications

➢ Background services

➢ Application programs

• File management - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.

• Status information

➢ Some ask the system for info - date, time, amount of available memory, disk space, number of users.

➢ Others provide detailed performance, logging, and debugging information.

➢ Typically, these programs format and print the output to the terminal or other output devices.

• File modification

➢ Text editors to create and modify files.

➢ Special commands to search contents of files or perform transformations of the text.

• Programming-language support - Compilers, assemblers, debuggers and interpreters sometimes provided.

• Communications - Provide the mechanism for creating virtual connections among processes, users, and computer systems.

➢ Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another.

OS Design and Implementation

• Design and Implementation of OS not "solvable", but some approaches have proven successful.

• Internal structure of different Operating Systems can vary widely.

• Start the design by defining goals and specifications.

• Affected by choice of hardware, type of system.

• User goals and System goals

➢User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast.

➢System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error free, and efficient.

• Specifying and designing an OS is highly creative task of software engineering

• Much variation

➢Early OSes in assembly language.

➢Then system programming languages like Algol, PL/1

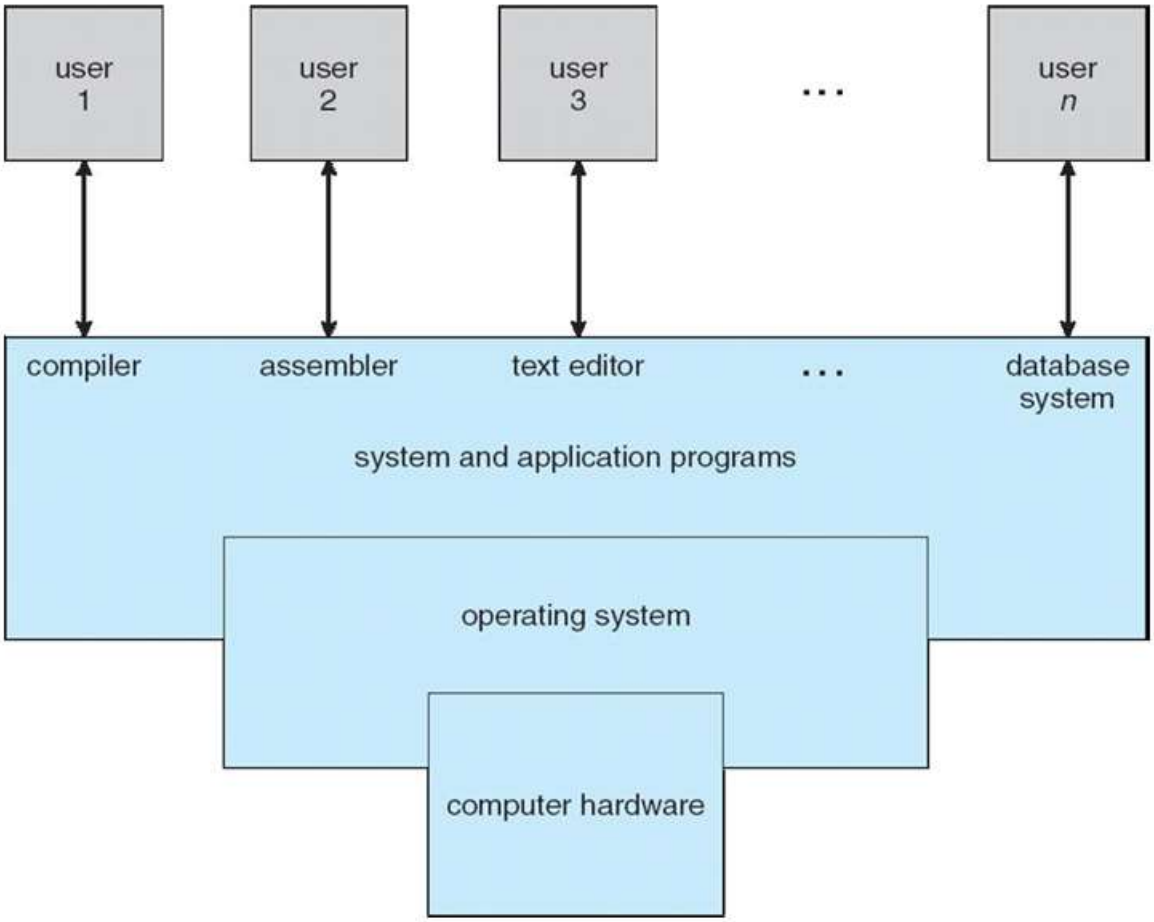
➢Now C, C++

• Actually usually a mix of languages

➢Lowest levels in assembly

➢Main body in C

➢Systems programs in C, C++, scripting languages like PERL, Python, shell scripts .



- General-purpose OS is very large program.
- Various ways to structure ones:
  - ➢ Simple structure – MS-DOS
  - ➢ More complex – UNIX
  - ➢ Layered
  - ➢ Microkernel – Mach

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Linux Commands

Lab - 3
Linux Commands

Keyboard Shortcuts

Ctrl + C	Kill process running in the terminal.
Ctrl + Z	Stop the current process. The process can be resumed in the foreground with **fg** or in the background with **bg**.
Ctrl + W	Cut one word before the cursor and add it to the clipboard.
Ctrl + U	Cut part of the line before the cursor and add it to the clipboard.
Ctrl + K	Cut part of the line after the cursor and add it to the clipboard.
Ctrl + Y	Paste from clipboard.
Ctrl + R	Recall the last command that matches the provided characters.
Ctrl + O	Run the previously recalled command.
Ctrl + G	Exit command history without running a command.
clear	Clear the terminal screen.
!!	Run the last command again.
exit	Log out of the current session.

Searching

find [path] -name	Find files and

Users and Groups

id	See details about the active users.
last	Show the last system logins.
who	Display who is currently logged into the system.
w	Show which users are logged in and their activity.
finger [user_name]	Show user information.
sudo useradd [user_name]	Create a new user account.
sudo adduser [user_name]	Create a new user account through the adduser command interface.
sudo userdel [user_name]	Delete a user account.
sudo usermod -aG [group_name] [user_name]	Modify user information (add a user to a group).
passwd sudo passwd [user_name]	Change the current user's or another user's password.
sudo groupadd [group_name]	Add a new group.
sudo groupdel [group_name]	Delete a group.
sudo groupmod -n [new_name] [old_name]	Modify a user group (change group name).
sudo [command]	Temporarily elevate user privileges to superuser or root.
su - [user_name]	Switch the user account or

Files

mkdir [directory_name]	Create a new directory.
rm [file_name]	Remove a file.
rm -r [directory_name]	Remove a directory recursively.
rm -rf [directory_name]	Recursively remove a directory without requiring confirmation.
cp [source_file] [destination_file]	Copy the contents of one file to another file.
cp -r [source_directory] [destination_directory]	Recursively copy a directory to a second directory.
mv [source_file] [destination_file]	Move or rename files or directories.
ln -s [path]/[file_name] [link_name]	Create a symbolic link to a file.
touch [file_name]	Create a new file.
cat [file_name]	Show the contents of a file.
cat [source_file] >> [destination_file]	Append file contents to another file.
head [file_name]	Show the first ten lines of a file.
tail [file_name]	Show the last ten lines of a file.
more [file_name]	Display contents of a file page by page.
less [file_name]	Show the contents of a file with navigation.
nano [file_name]	Open or create a file using the nano text editor.

System Management

uname -r	Show system information via uname command.
uname -a	See kernel release information.
uptime	Display how long the system has been running, including the load average.
hostname	View system hostname.
hostname -i	Show the IP address of the system.
last reboot	List system reboot history.
date	See current time and date.
timedatectl	Query and change the system clock.
cal	Show current calendar (month and day).
w	List logged-in users.
whoami	See which user you are using.
finger [user_name]	Show information about a particular user.
ulimit [flags] [limit]	View or limit system resource amounts.
shutdown [hh:mm]	Schedule a system shutdown.
shutdown now	Shut down the system immediately.
modprobe [module_name]	Add a new kernel module.
dmesg	Show bootup messages.

Processes

ps	List active processes.
pstree	Show processes in a tree-like diagram.
pmap	Display a memory usage map of processes.
top	See all running processes.
htop	Interactive and colorful process viewer.
kill [process_id]	Terminate a Linux process under a given ID.
pkill [process_name]	Terminate a process under a specific name.
killall [label]	Terminate all processes with a given label.
pgrep [keyword]	List processes based on the provided keyword.
pidof [process_name]	Show the PID of a process.
bg	List and resume stopped jobs in the background.
fg	Bring the most recently suspended job to the foreground.
fg [job]	Bring a particular job to the foreground.
lsof	List files opened by running processes.
trap "[commands]" [signal]	Catch a system error signal in a shell script. Executes provided commands when the signal is caught.

specified location.	
find [path] -size [+100M]	See files and directories larger than a specified size in a directory.
grep [search_pattern] [file_name]	Search for a specific pattern in a file with grep.
grep -r [search_pattern] [directory_name]	Recursively search for a pattern in a directory.
locate [name]	Locate all files and directories related to a particular name.
which [command]	Search the command path in the $PATH environment variable.
whereis [command]	Find the source, binary, and manual page for a command.
awk '[search_pattern] {print $0}' [file_name]	Print all lines matching a pattern in a file. See also the gawk command, the GNU version of awk.
sed 's/[old_text]/[new_text]/' [file_name]	Find and replace text in a specified file.

Directory Navigation

ls	List files and directories in the current directory.
ls -a	List all files and directories in the current directory (shows hidden files).
ls -l	List files and directories in long format.
pwd	Show the directory you are currently working in.
cd cd ~	Change directory to $HOME.

directory]	group.

SSH Login

ssh [user_name]@[host]	Connect to a remote host as a user via SSH.
ssh [host]	Securely connect to a host via SSH default port 22.
ssh -p [port] [user_name]@[host]	Connect to the host using a particular port.
ssh-keygen	Generate SSH key pairs.
sudo service sshd start	Start SSH server daemon.
scp [file_name] [user_name]@[host]:[remote_path]	Securely copy files between local and remote systems via SSH.
sftp [user_name]@[host]	Interactive file transfer over encrypted SSH session using SFTP protocol.
telnet [host]	Connect to the host via Telnet default port 23.

File Permissions

chmod 777 [file_name]	Assign read, write, and execute file permission to everyone (rwxrwxrwx).
chmod 755 [file_name]	Give read, write, and execute permission to owner, and read and execute permission to group and others (rwxr-xr-x).
chmod 766 [file_name]	Assign full permission to the owner, and read and write permission to the

gpg -c [file_name]	Encrypt a file.
gpg [file_name].gpg	Decrypt an encrypted .gpg file.
wc -w [file_name]	Show the number of words, lines, and bytes in a file.
ls | xargs wc	List the number of lines/words/characters in each file in a directory.
cut -d [delimiter] [file_name]	Cut a section of a file and print the result to standard output.
[data] | cut -d [delimiter]	Cut a section of piped data and print the result to standard output.
shred -u [file_name]	Overwrite a file to prevent its recovery, then delete it.
diff [first_file] [second_file]	Compare two files and display differences.
source [file_name]	Read and execute the file content in the current shell.
[command] | tee [file_name] >/dev/null	Store the command output in a file and skip the terminal output.

Disk Usage

df -h	Check free and used space on mounted systems.
df -i	Show free inodes on mounted file systems.
fdisk -l	Display disk partitions, sizes, and types with the command.
du -ah	See disk usage for all files and directories.

Network

ip addr show	List IP addresses and network interfaces.
ip address add [IP_address]	Assign an IP address to interface eth0.
ifconfig	Display IP addresses of all network interfaces.
ping [remote_host]	Ping remote host.
netstat -pnltu	See active (listening) ports with the netstat command.
netstat -tuln	Show TCP and UDP ports and their programs.
whois [domain_name]	Display more information about a domain.
dig [domain_name]	Show DNS information about a domain using the dig command.
dig -x [domain_name]	Do a reverse DNS lookup on the domain.
dig -x [IP_address]	Do a reverse DNS lookup of an IP address.
host [domain_name]	Perform an IP lookup for a domain.
hostname -I	Show the local IP address.
nslookup [domain_name]	Receive information about an internet domain.

File Compression

tar cf [archive.tar] [file/directory]	Archive an existing file or directory.
tar xf [archive.tar]	Extract an archived file.
tar czf [archive.tar.gz]	Create a .gz compressed tar archive.
gzip [file_name]	Compress or

	process is completed.
nohup [command] &	Run a Linux process in the background.

Hardware Information

lscpu	See CPU information.
lsblk	See information about block devices.
lspci -tv	Show PCI devices in a tree-like diagram.
lsusb -tv	Display USB devices in a tree-like diagram.
lshw	List hardware configuration information.
cat /proc/cpuinfo	Show detailed CPU information.
cat /proc/meminfo	View detailed system memory information.
cat /proc/mounts	See mounted file systems.
free -h	Display free and used memory.
sudo dmidecode	Show hardware information from the BIOS.
hdparm -i /dev/[device_name]	Display disk data information.
hdparm -tT /dev/[device_name]	Conduct a read speed test on the device/disk.
badblocks -s /dev/[device_name]	Test for unreadable blocks on the device/disk.
fsck /dev/[device_name]	Run a disk check on an unmounted disk or partition.

Shell Commands

pwd	Show the directory you are currently working in
cd cd ~	Change directory to $HOME
cd ..	Move up one directory level
cd -	Change to the previous directory
cd [directory_path]	Change location to a specified directory
dirs	Show current directory stack

Packages (Debian/Ubuntu)

sudo apt-get install [package_name]	Install an APT package using the apt-get package utility
sudo apt install [package_name]	Install an APT package using a newer APT package manager
apt search [keyword]	Search for a package in the APT repositories
apt list	List packages installed with APT
apt show [package_name]	Show information about a package
sudo dpkg -i [package_name.deb]	Install a .deb package with the Debian package manager
sudo dpkg -l	List packages installed with dpkg

	(rwxr-xr-x)
chmod 766 [file_name]	Assign full permission to the owner, and read and write permission to the group and others (rwxrw-rw-)
chown [user_name] [file_name]	Change the ownership of a file with chown command
chown [user_name]:[group_name] [file_name]	Change the owner and group ownership of a file

Packages (Red Hat, CentOS, Fedora)

sudo yum install [package_name]	Install a package using the YUM package manager
yum search [keyword]	Find a package in the YUM repositories based on the provided keyword
yum list installed	List all packages installed with YUM
yum info [package_name]	Show package information for a package
sudo dnf install [package_name]	Install a package using the DNF package manager
sudo rpm -i [package_name.rpm]	Install a .rpm package from a local file

	command
du -ah	See disk usage for all files and directories
du -sh	Show disk usage of the current directory
mount	Show currently mounted file systems
findmnt	Display target mount point for all file systems
mount [device_path] [mount_point]	Mount a device

Packages (Universal)

tar zxvf [file_name.tar.gz] cd [extracted_directory] ./configure make make install	Install software from source code
sudo snap install [package_name]	Install a Snap package
sudo snap find [keyword]	Search for a package in the Snap store
sudo snap list	List installed Snap packages
flatpak install [package_name]	Install a Flatpak package
flatpak search [keyword]	Search for a Flatpak application in repositories
flatpak list	List installed Flatpack packages

tar czf [archive.tar.gz]	Create a .gz compressed tar archive
gzip [file_name] gunzip [file_name.gz]	Compress or decompress .gz files
bzip2 [file_name] bunzip2 [file_name.bz2]	Compress or decompress .bz2 files

File Transfer

scp [source_file] [user]@[remote_host]:[destination_path]	Copy a file to a server directory securely
rsync -a [source_directory] [user]@[remote_host]:[destination_directory]	Synchronize the contents of a directory with a backup directory
wget [link]	Download files from FTP or web servers
curl -O [link]	Transfer data to or from a server with various protocols
ftp [remote_host]	Transfer files between local and remote systems interactively using FTP
sftp [user]@[remote_host]	Securely transfer between local and remote hosts using SFTP

	partition

Shell Commands

alias [alias-name]='[command]'	Create an alias for a command
watch -n [interval-in-seconds] [command]	Set a custom interval to run a user-defined command
sleep [time-interval] && [command]	Postpone the execution of a command
at [hh:mm]	Create a job to be executed at a certain time (Ctrl+D to exit prompt after command)
man [command]	Display a built-in manual for a command
history	Print the command history used in the terminal

Variables

let "[variable_name]=[value]"	Assign an integer value to a variable
export [variable_name]	Export a Bash variable
declare [variable-name]="[value]"	Declare a Bash variable
set	List the names of all the shell variables and functions
unset [variable_name]	Remove an environment variable
echo $[variable-name]	Display the value of a variable

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Threads

Multicore Programming ,Multithreading Models , Benefits of Threads , Thread Libraries

Lab - 4

Threads

Multicore Programming ,Multithreading Models , Benefits of Threads , Thread Libraries

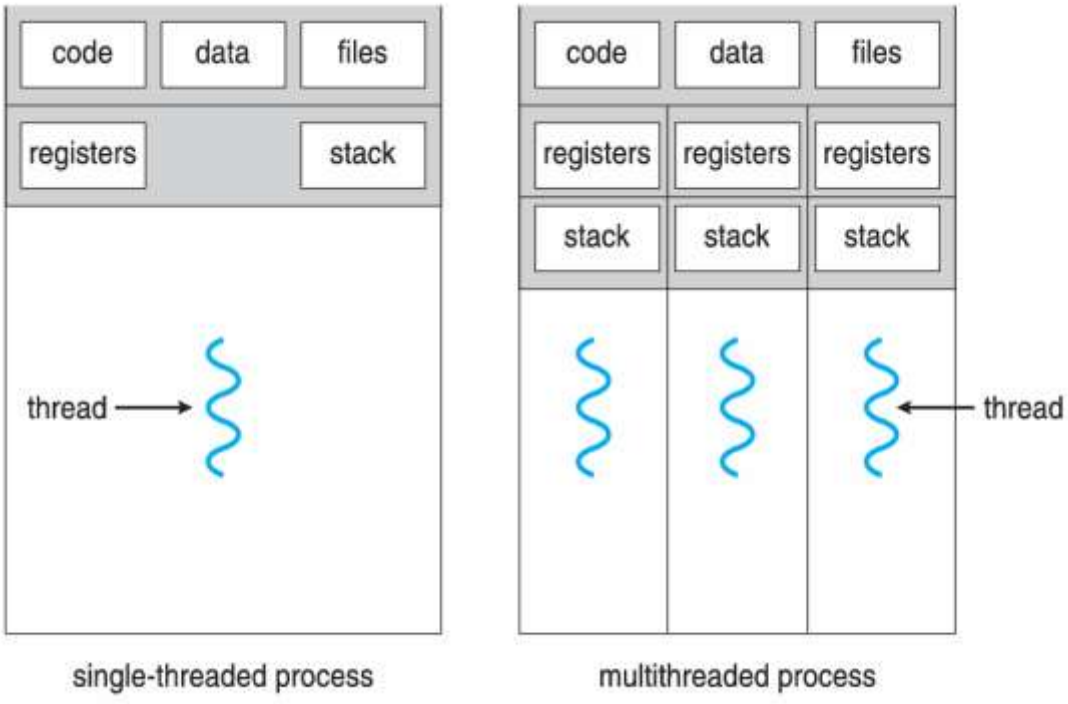
Introduction

• A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack.

• It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals

• Let's say, for example, a program is not capable of drawing pictures while reading keystrokes. The program must give its full attention to the keyboard input lacking the ability to handle more than one event at a time.

• The ideal solution to this problem is the seamless execution of two or more sections of a program at the same time. Threads allows us to do this



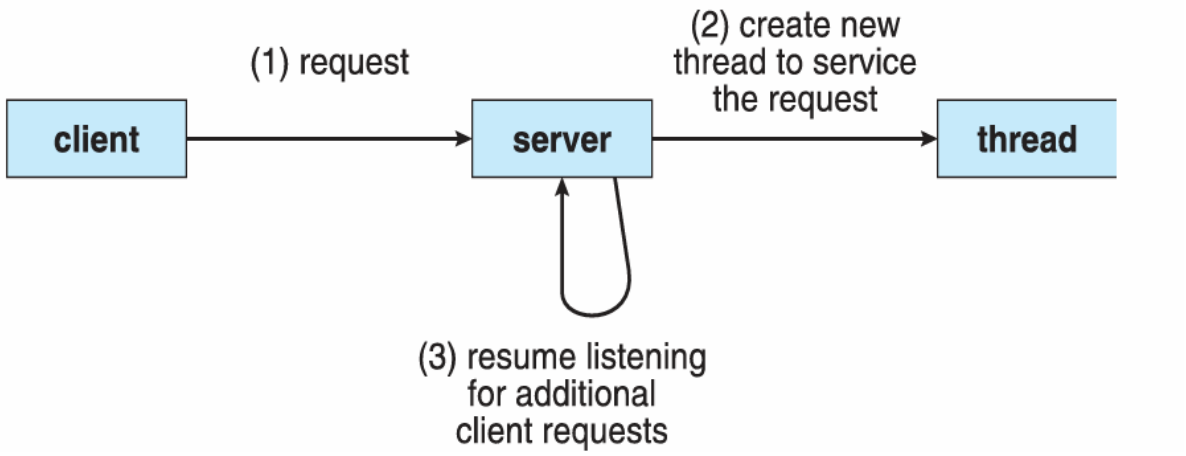
• Most software applications that run on modern computers are multithreaded.

• An application typically is implemented as a separate process with several threads of control.

➢ A web browser might have one thread display images or text while another thread retrieves data from the network, for example.

➢ A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

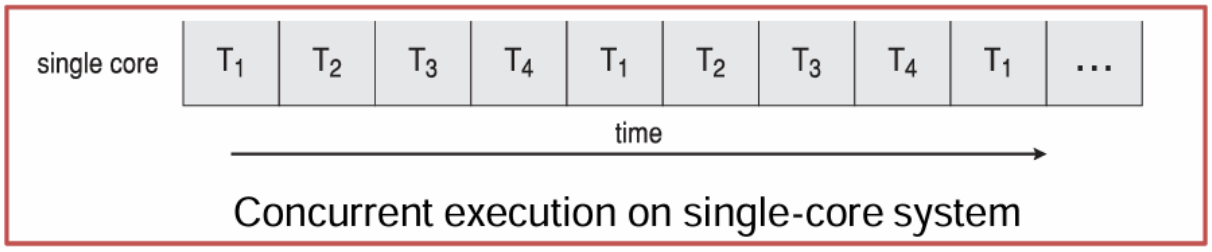
• Multithreaded server architecture

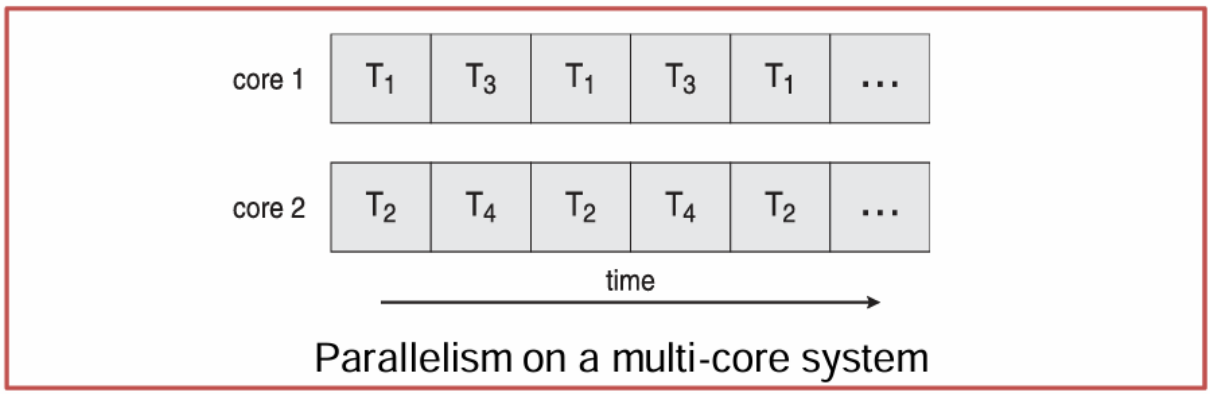


• Finally, most operating-system kernels are now multithreaded. Several threads operate in the kernel, and each thread performs a specific task, such as managing devices, managing memory, or interrupt handling.

Multicore Programming

•Multithreaded programming provides a mechanism for more efficient use of these multicore or multiprocessor systems.





Multithreading Models

• Support for threads may be provided either at the user level, for user threads, or by the kernel, for kernel threads.

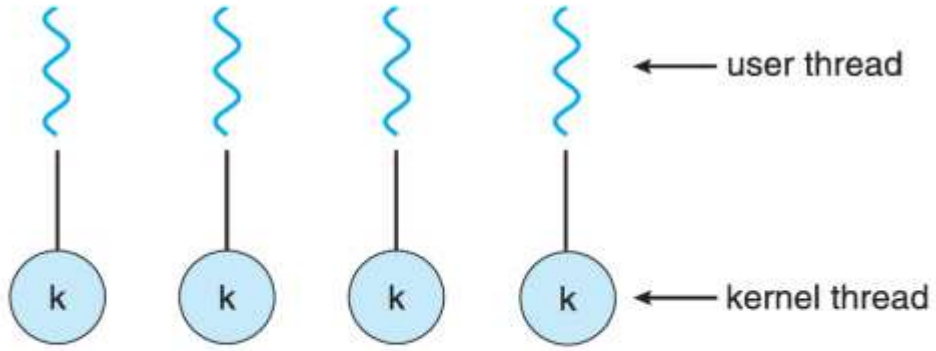
• User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.

• Ultimately, a relationship must exist between user threads and kernel threads.

• We look at three common models:

- ➢the one-to-one model,
- ➢the many-to-one model, and
- ➢the many-to-many model

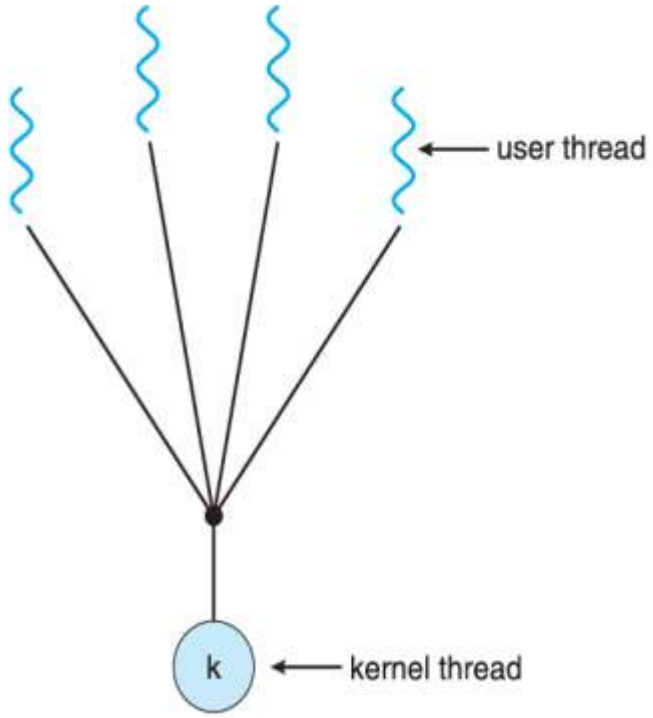
One-to-One model (1/3)



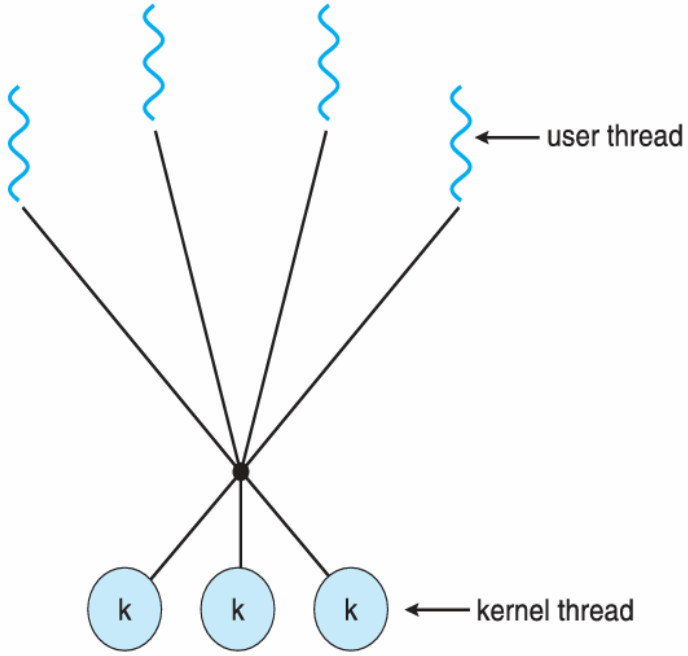
The one-to-one model maps each user thread to a kernel thread. It also allows multiple threads to run in parallel on multiprocessors.

The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system. Linux, along with the family of Windows operating systems, implement the one-to-one model.

Many-to-One model



Many-to-Many model (1/2)

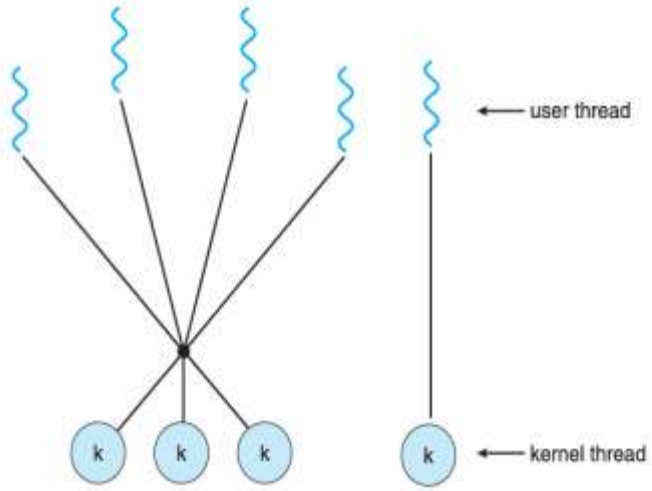


Many-to-Many model

• Allows many user level threads to be mapped to many kernel threads.
• Allows the operating system to create a sufficient number of kernel threads.
• Allows the developer to create as many user threads as she wishes, it does not result in true concurrency, because the kernel can schedule only one thread at a time.

Two-level model

• Similar to Many-to-Many, except that it allows a user thread to be **bound** to kernel thread.



Benefits of Threads

• 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.

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

CPU Scheduling

FCFS, SJF Preemptive , SJF NonPreemptive , Priority , Round Robin

Lab - 5

CPU Scheduling

FCFS , SJF NonPreemptive

Lab - 5

cpu scheduling

FCFS, SJF NonPreemptive

1. An overview

• CPU scheduling is the central in multi-programming system.

• Maximum CPU utilization obtained multiprogramming (prevent CPU from being idle). with

• Processes residing in the main memory is selected by the Scheduler that is:

- ➢ Concerned with deciding a policy about which process is to be selected.
- ➢ Process selection based on a scheduling algorithm.

• Process execution consists of a cycle of CPU execution and I/O wait.

• Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.

• CPU bursts vary greatly from process to process and from computer to computer.

Scheduling can be

• Non-preemptive

- ➢ Once a process is allocated the CPU, it leave until terminate.

• Preemptive does not

- ➢ OS can force (preempt) a process from CPU at any time.
- ✓ Say, to allocate CPU to another higher-priority process.

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. (time from submission to termination)
- 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.

Scheduling Algorithm Optimization Criteria

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

There are many different CPU-scheduling algorithms:

1. First Come, First Served (FCFS).
2. Shortest Job First (SJF).
    - ➢Preemptive SJF.
    - ➢Non-Preemptive SJF.
3. Priority.
4. Round Robin.
5. Multilevel queues.

1. First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order: P1 , P2 , P3 The Gantt Chart for the schedule is: Note: A process may have many CPU bursts, but in the following examples we show only one for simplicity.

Waiting Time		
P_1	P_2	P_3
0	24	27

Average waiting time: (0 + 24 + 27)/3 = 17

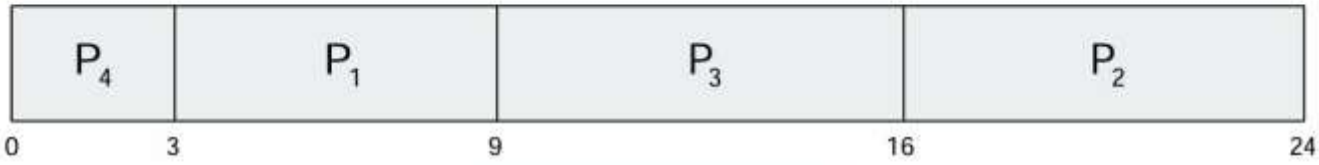
Turnaround Time		
P_1	P_2	P_3
24	27	30

Average turnaround time: (24 + 27 + 30)/3 = 27

2.1 Shortest-Job-First (SJF) Scheduling

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



Waiting Time			
P_1	P_2	P_3	P_4
3	16	9	0

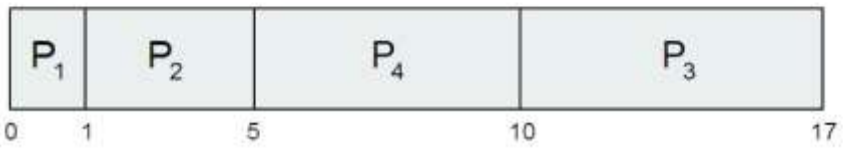
Average waiting time: (3 + 16 + 9 + 0)/4 = 7

2.1 Shortest-Job-First (SJF) (Non-Preemptive SJF)

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

Process	*Arrival* Time	Burst Time
P_1	0	1
P_2	1	4
P_3	2	7
P_4	3	5

- *Non-Preemptive* SJF Gantt Chart



Waiting Time			
P_1	P_2	P_3	P_4
(0 – 0)	(1 – 1)	(10 – 2)	(5 – 3)

```
# Python3 program for implementation
# of FCFS scheduling
# Function to find the waiting
# time for all processes
def findWaitingTime(processes, n, bt, wt):

    # waiting time for
    # first process is 0
    wt[0] = 0

    # calculating waiting time
    for i in range(1, n ):
        wt[i] = bt[i - 1] + wt[i - 1]

# Function to calculate turn
# around time
def findTurnAroundTime(processes, n,
                       bt, wt, tat):

    # calculating turnaround
    # time by adding bt[i] + wt[i]
    for i in range(n):
        tat[i] = bt[i] + wt[i]

# Function to calculate
# average time
def findavgTime( processes, n, bt):

    wt = [0] * n
    tat = [0] * n
    total_wt = 0
    total_tat = 0

    # Function to find waiting
    # time of all processes
    findWaitingTime(processes, n, bt, wt)

    # Function to find turn around
    # time for all processes
    findTurnAroundTime(processes, n,
                       bt, wt, tat)

    # Display processes along
    # with all details
    print( "Processes Burst time " +
                " Waiting time " +
               " Turn around time")
```

```
    # Calculate total waiting time
    # and total turn around time
    for i in range(n):

        total_wt = total_wt + wt[i]
        total_tat = total_tat + tat[i]
        print(" " + str(i + 1) + "\t\t" +
                    str(bt[i]) + "\t " +
                    str(wt[i]) + "\t\t " +
                    str(tat[i]))

    print( "Average waiting time = "+
                str(total_wt / n))
    print("Average turn around time = "+
                 str(total_tat / n))

# Driver code
if __name__ =="__main__":

    # process id's
    processes = [ 1, 2, 3]
    n = len(processes)

    # Burst time of all processes
    burst_time = [10, 5, 8]

    findavgTime(processes, n, burst_time)

# This code is contributed
# by ChitraNayal
```

```
class Process:
    def __init__(self, pid, burst_time):
        self.pid = pid
        self.burst_time = burst_time

def sjf_scheduling(processes):
    processes.sort(key=lambda x: x.burst_time)  # Sort processes by burst time
    total_processes = len(processes)
    current_time = 0
    waiting_time = 0

    print("Process ID\tBurst Time\tWaiting Time")
    for p in processes:
        print(f"{p.pid}\t\t{p.burst_time}\t\t{waiting_time}")
        waiting_time += current_time
        current_time += p.burst_time

    average_waiting_time = waiting_time / total_processes
    print("\nAverage Waiting Time:", average_waiting_time)

# Example usage
if __name__ == "__main__":
    processes = [Process(1, 6), Process(2, 8), Process(3, 7), Process(4, 3)]
    sjf_scheduling(processes)
```

Lab - 6

CPU Scheduling

SJF Preemptive , Priority , Round Robin

2.2 Shortest-remaining-time-first (Preemptive SJF)

- 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

Process	***Arrival* Time**	Burst Time
P_1	0	~~8~~ ~~7~~
P_2	1	~~4~~ ~~3~~ ~~2~~
P_3	2	~~9~~
P_4	3	~~5~~

26 ms

- ***Preemptive*** SJF Gantt Chart

Waiting Time			
P_1	P_2	P_3	P_4
10 – 1	1 – 1	17 – 2	5 – 3
= 9	= 0	= 15	= 2

Average waiting time = [9+0+15+2]/4 = 26/4 = 6.5 msec

3. 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

3. 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

Waiting Time				
P_1	P_2	P_3	P_4	P_5
6	0	16	18	1

Average Waiting time = [6+0+16+18+1]/5 = 8.2 msec

4. Round Robin (RR) Scheduling

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.

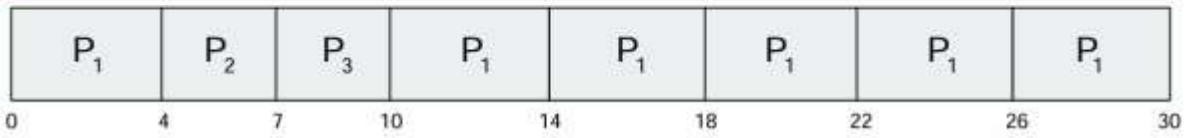
If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once.

No process waits more than $(n-1)q$ time units.

4. Round Robin (RR) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- All the processes **arrive** at the same time **0**.
- Round Robin (RR) scheduling of quantum: **4** ms



Waiting Time		
P_1	P_2	P_3
$0 + (10 - 4)$	4	7

Average waiting time: $(6 + 4 + 7)/3 = 5.667$ ms

Priority Scheduling Code

```
# Python3 program for implementation of
# Priority Scheduling

# Function to find the waiting time
# for all processes


def findWaitingTime(processes, n, wt):
    wt[0] = 0

    # calculating waiting time
    for i in range(1, n):
        wt[i] = processes[i - 1][1] + wt[i - 1]

# Function to calculate turn around time


def findTurnAroundTime(processes, n, wt, tat):

    # Calculating turnaround time by
    # adding bt[i] + wt[i]
    for i in range(n):
        tat[i] = processes[i][1] + wt[i]

# Function to calculate average waiting
# and turn-around times.


def findavgTime(processes, n):
    wt = [0] * n
    tat = [0] * n

    # Function to find waiting time
    # of all processes
    findWaitingTime(processes, n, wt)

    # Function to find turn around time
    # for all processes
    findTurnAroundTime(processes, n, wt, tat)

    # Display processes along with all details
    print("\nProcesses    Burst Time    Waiting",
          "Time    Turn-Around Time")
    total_wt = 0
    total_tat = 0
    for i in range(n):

        total_wt = total_wt + wt[i]
        total_tat = total_tat + tat[i]
        print(" ", processes[i][0], "\t\t",
              processes[i][1], "\t\t",
              wt[i], "\t\t", tat[i])
```

```
    print("\nAverage waiting time = %.5f " % (total_wt / n))
    print("Average turn around time = ", total_tat / n)


def priorityScheduling(proc, n):

    # Sort processes by priority
    proc = sorted(proc, key=lambda proc: proc[2],
                  reverse=True)

    print("Order in which processes gets executed")
    for i in proc:
        print(i[0], end=" ")
    findavgTime(proc, n)


# Driver code
if __name__ == "__main__":

    # Process id's
    proc = [[1, 10, 1],
            [2, 5, 0],
            [3, 8, 1]]
    n = 3
    priorityScheduling(proc, n)

# This code is contributed
# Shubham Singh(SHUBHAMSINGH10)
```

Round Robin Code

```
<script>
    // JavaScript program for implementation of RR scheduling

    // Function to find the waiting time for all
    // processes
    const findWaitingTime = (processes, n, bt, wt, quantum) => {
        // Make a copy of burst times bt[] to store remaining
        // burst times.
        let rem_bt = new Array(n).fill(0);
        for (let i = 0; i < n; i++)
            rem_bt[i] = bt[i];

        let t = 0; // Current time

        // Keep traversing processes in round robin manner
        // until all of them are not done.
        while (1) {
            let done = true;

            // Traverse all processes one by one repeatedly
            for (let i = 0; i < n; i++) {
                // If burst time of a process is greater than 0
```

```
            // then only need to process further
            if (rem_bt[i] > 0) {
                done = false; // There is a pending process

                if (rem_bt[i] > quantum) {
                    // Increase the value of t i.e. shows
                    // how much time a process has been processed
                    t += quantum;

                    // Decrease the burst_time of current process
                    // by quantum
                    rem_bt[i] -= quantum;
                }

                // If burst time is smaller than or equal to
                // quantum. Last cycle for this process
                else {
                    // Increase the value of t i.e. shows
                    // how much time a process has been processed
                    t = t + rem_bt[i];

                    // Waiting time is current time minus time
                    // used by this process
                    wt[i] = t - bt[i];

                    // As the process gets fully executed
                    // make its remaining burst time = 0
                    rem_bt[i] = 0;
                }
            }
        }

        // If all processes are done
        if (done == true)
            break;
    }
}

// Function to calculate turn around time
const findTurnAroundTime = (processes, n, bt, wt, tat) => {
    // calculating turnaround time by adding
    // bt[i] + wt[i]
    for (let i = 0; i < n; i++)
        tat[i] = bt[i] + wt[i];
}

// Function to calculate average time
const findavgTime = (processes, n, bt, quantum) => {
    let wt = new Array(n).fill(0), tat = new Array(n).fill(0);
```

```
    let total_wt = 0, total_tat = 0;

    // Function to find waiting time of all processes
    findWaitingTime(processes, n, bt, wt, quantum);

    // Function to find turn around time for all processes
    findTurnAroundTime(processes, n, bt, wt, tat);

    // Display processes along with all details
    document.write(`Processes Burst time Waiting time Turn around time<br/>`);

    // Calculate total waiting time and total turn
    // around time
    for (let i = 0; i < n; i++) {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];

        document.write(`${i + 1} ${bt[i]} ${wt[i]} ${tat[i]}<br/>`);
    }

    document.write(`Average waiting time = ${total_wt / n}`);
    document.write(`<br/>Average turn around time = ${total_tat / n}`);
}

// Driver code
// process id's
processes = [1, 2, 3];
let n = processes.length;

// Burst time of all processes
let burst_time = [10, 5, 8];

// Time quantum
let quantum = 2;
findavgTime(processes, n, burst_time, quantum);

# This code is contributed by
# Shubham Singh(SHUBHAMSINGH10)
```

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Process Synchronization

Introduction , The Critical Section Problem ,Solution to The Critical Section Problem.

Lab - 7

Process Synchronization

Introduction , The Critical Section Problem ,Solution to The Critical Section Problem.

Lab - 7

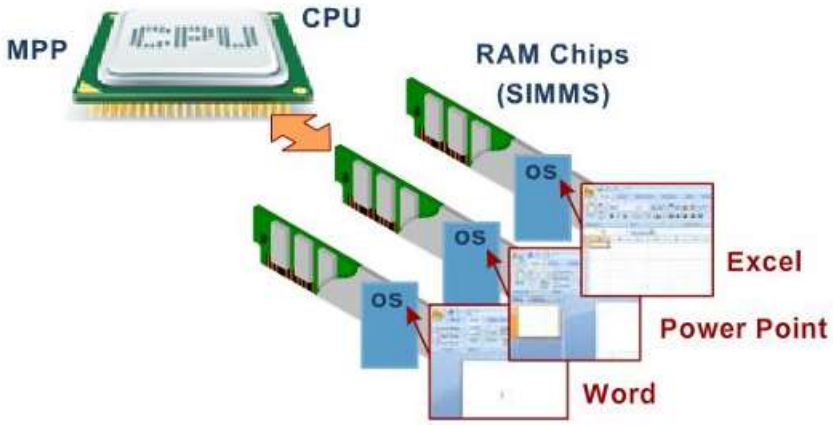
Reasoning & Inference

Forward Chaining, Backward Chaining, Expert System

1. An overview

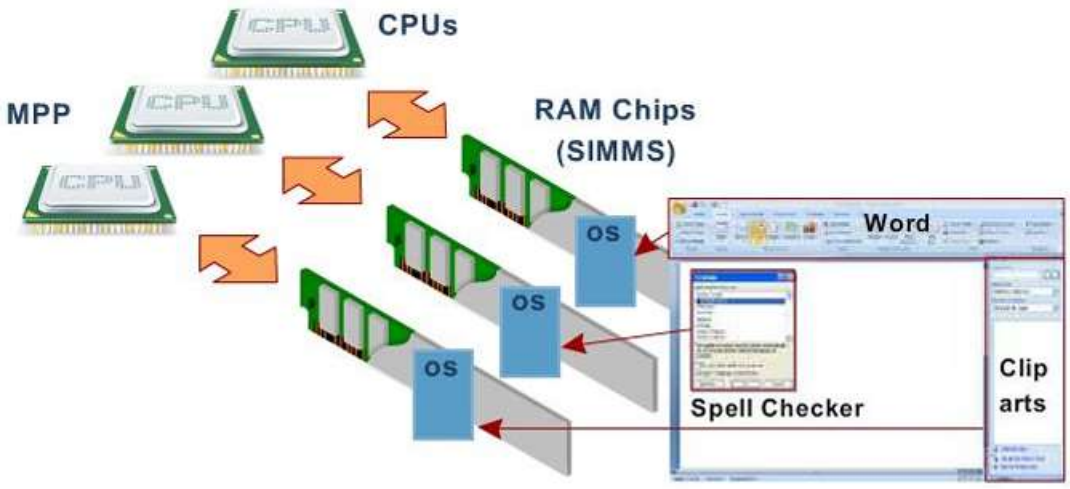
• The operating system supports multiple processes running in parallel that are the results from the following:

➢ Multiple applications: Multi-programming allows processor time to be shared among a number of active applications



• The operating system supports multiple processes running in parallel that are the results from the following:

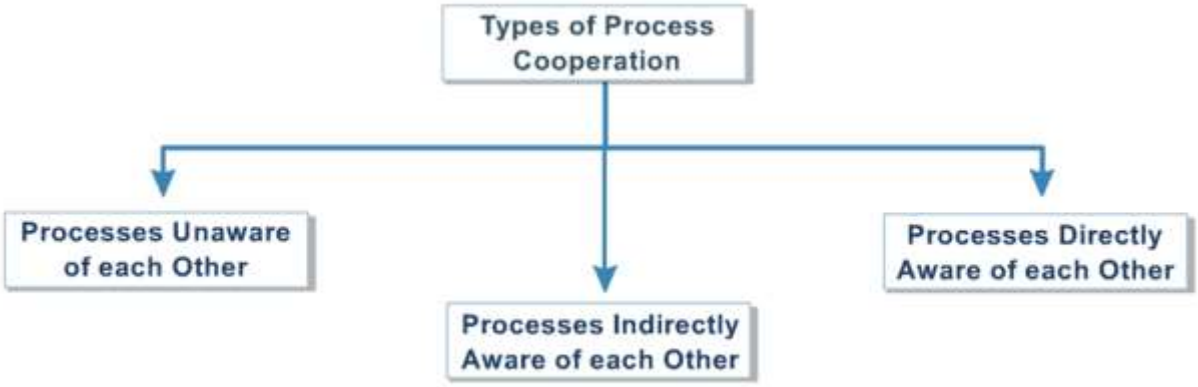
➢ Structured application: some applications can be implemented as a set of concurrent processes.



• A cooperating process is one that can affected by affect or be other processes executing in the system.
• Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. The former case is achieved through the use of threads, discussed in Chapter 4.
• Concurrent access to shared data may result in data inconsistency, however. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Types of Process Cooperation

• Processes interact with each other can be:



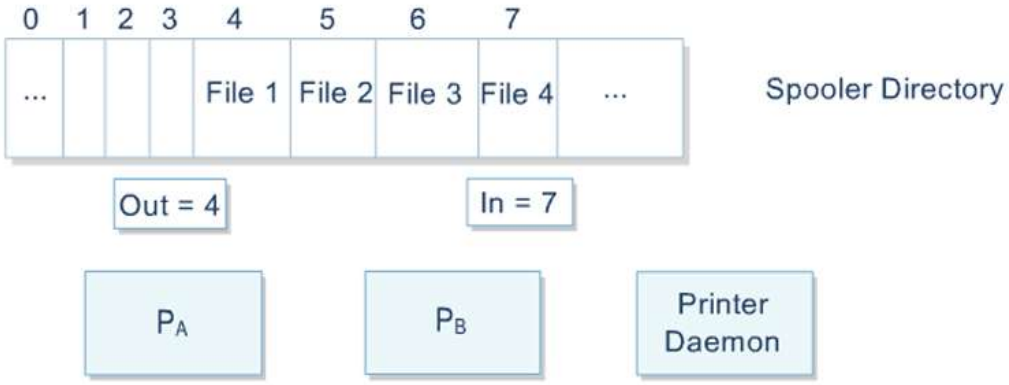
Processes unaware of each other

• Those are independent processes that are not intended to work together.
• For example, in a single user environment, two independent applications may both want to access the same disk, file or printer.

• In a multi-user environment, users are concurrent, use the same web server and need access to the same web page. A person can handle several tasks at once (have you every listened to music while doing other work and heard the phone ring?).

Critical Section Problem

• The problem has a Spooler directory with slots; index variable; two processes attempt concurrent access.
• According to the figure, at a certain time, slots 0 to 3 are empty and slot 4 to 7 are full. Process A and B simultaneously want to queue a file for printing



Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

DeadLocks

Introduction to Deadlock., Deadlock Characterization. , Methods for Handling Deadlocks.

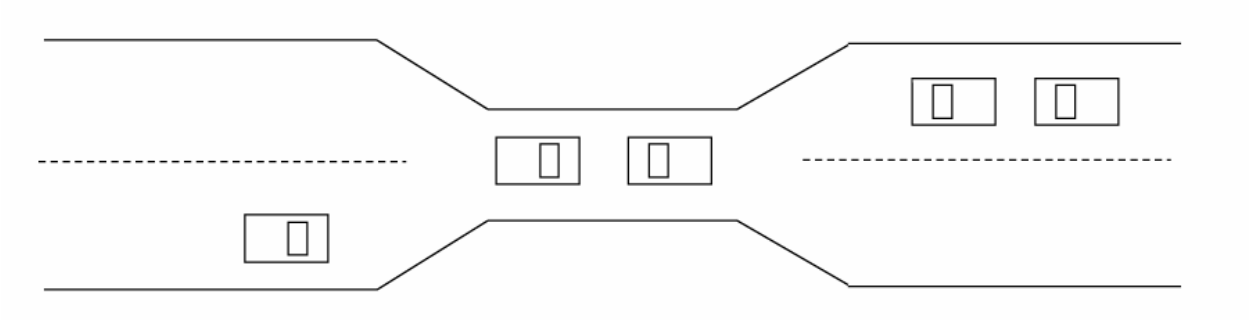
Lab - 8

DeadLocks

Introduction to Deadlock., Deadlock Characterization. , Methods for Handling Deadlocks.

• Generally speaking, deadlock, involves conflicting needs for resources by two or more request orders. A common example is a traffic deadlock.

- ➢ If deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- ➢ Several cars may have to back up if deadlock occurs.
- ➢ Starvation is possible.



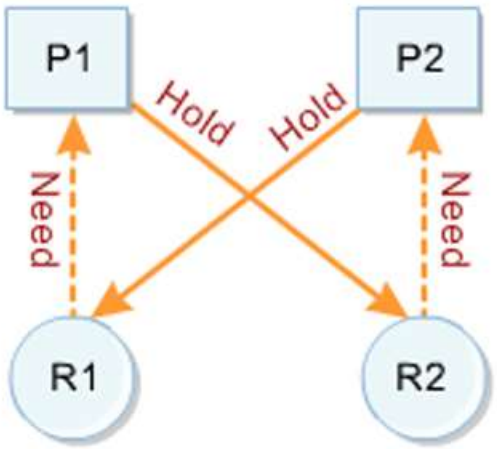
Deadlock in computer system

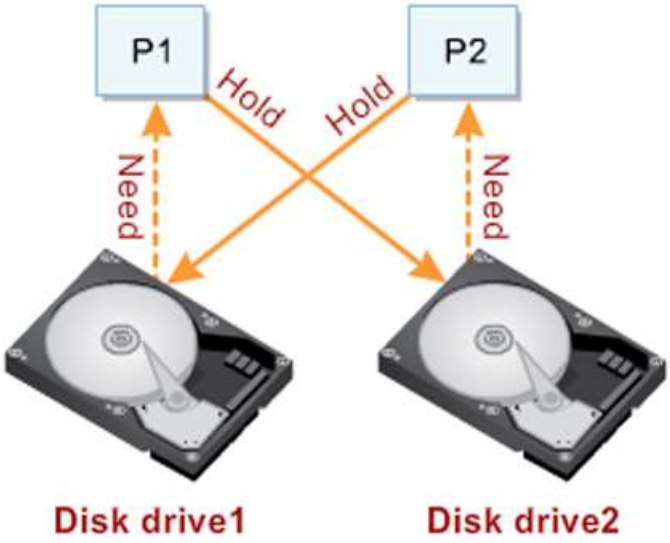
• A computer system consists of a finite number of resources to be distributed among a number of competing processes.

• An operating system is a resource allocator i.e., there are many resources that can be allocated to only one process at a time. • Each process utilizes a resource as follows:

- ➢ request
- ➢ use
- ➢ release

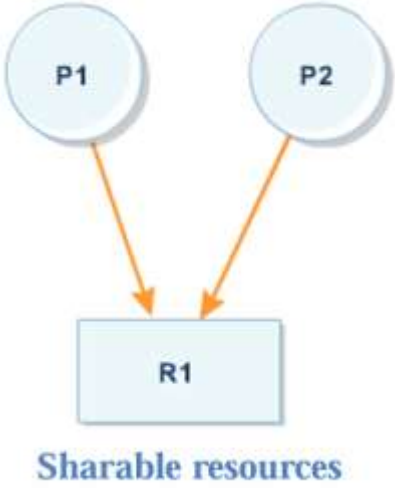
• General example of deadlock in a computer system:

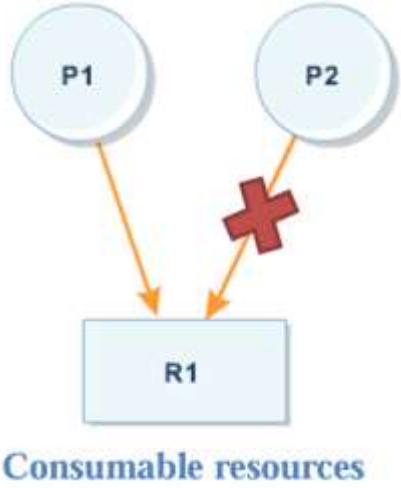




Types of resources in computer

• Sharable resources can be used by more than one process at a time. A consumable resource can only be used by one process.





• Resources can be pre-emptable or non pre-emptable.

➢ Memory is an example of a pre-emptable resource, but

➢ A printer is a non-preemptable one.

• Reusable: used by one process at a time and then returned.

➢ Processors, I/O channels, main and secondary memory, files, databases, and semaphores.

➢ Deadlock may occur if each process holds one resource and requests another.

• Consumable: created (produced) and destroyed (consumed) by a process.

➢ Interrupts, signals, messages, data in I/O buffers.

➢ Deadlock may occur if receive_message() is blocking.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

• Mutual exclusion.

• Hold and wait.

• No preemption.

• Circular wait

Deadlock can arise if four conditions hold simultaneously.

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

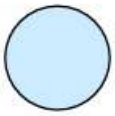
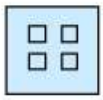
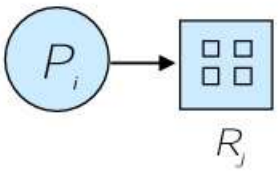
2. Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes.

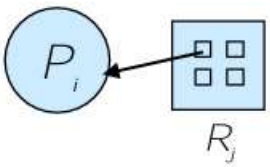
3. No preemption: a resource can be released only willingly by the process holding it, after that process has completed its task.

4. Circular wait: there exists a set {P0, P1, …, Pn} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.

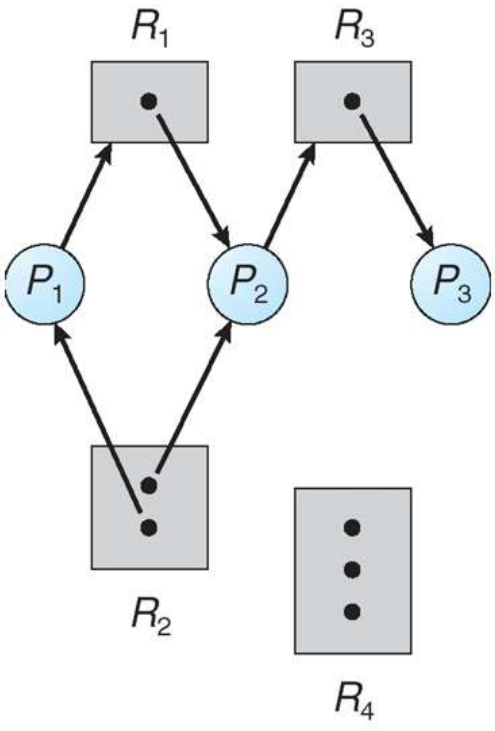
©Ahmed Hagag Operating Systems 13 Deadlock Characterization (4/11) Resource Allocation Graph (1 / 2) • A set of vertices V and a set of edges E. • Vis partitioned into two types: ➢P= {P1, P2, …, Pn}, the set consisting of all processes in the system. ➢R= {R1, R2, …, Rm}, the set consisting of all resource types in the system. • Request edge – directed edge Pi→ Rj • Assignment edge – directed edge Rj → Pi

Resource-Allocation Graph

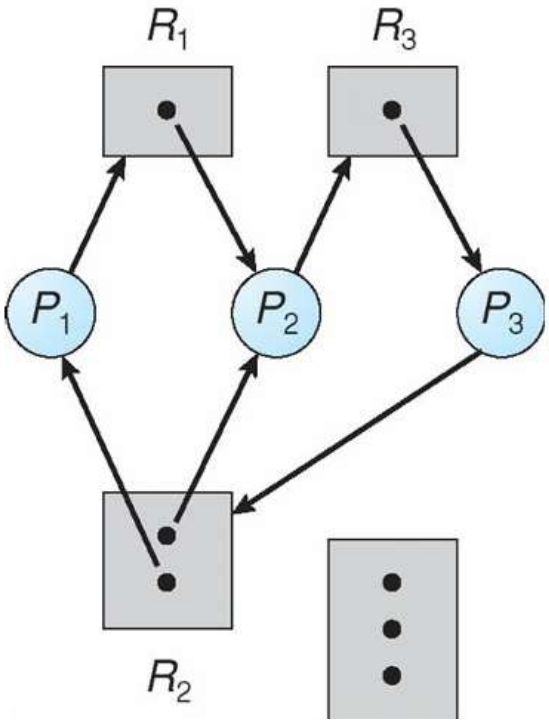
- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

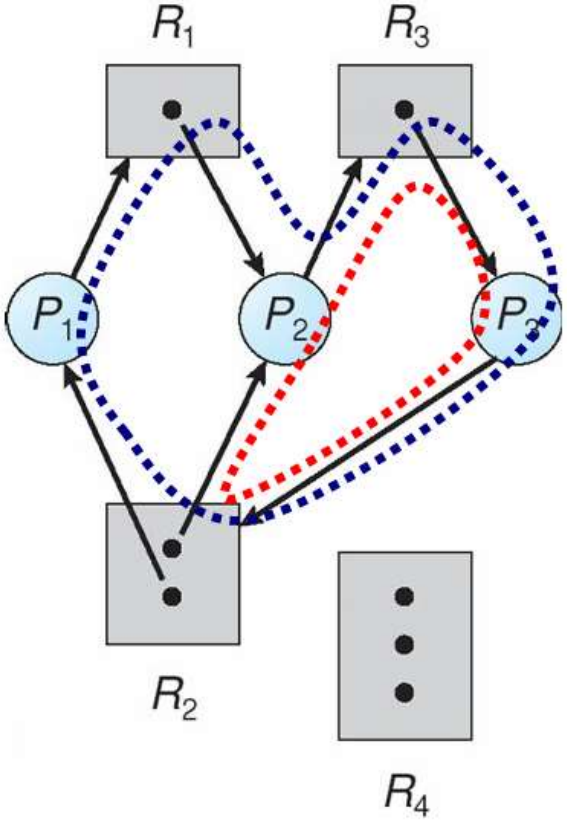
Example of a Resource Allocation Graph



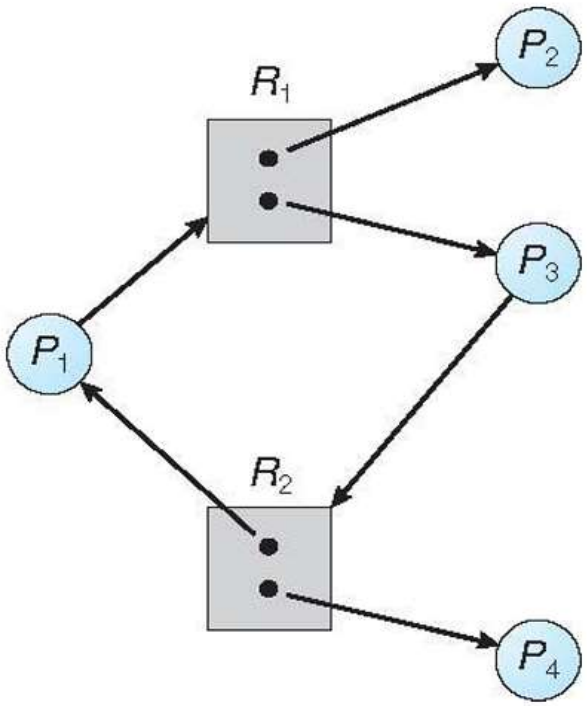
Resource Allocation Graph With A Deadlock



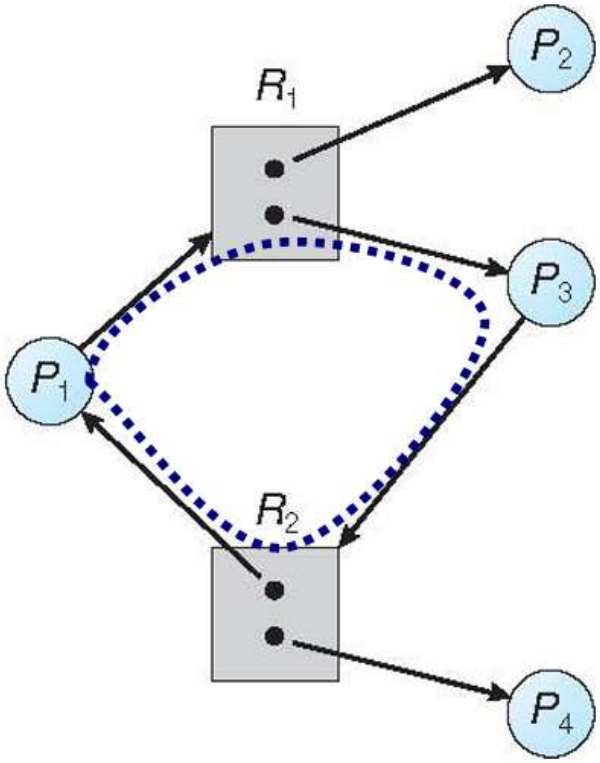
Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Graph With A Cycle But No Deadlock



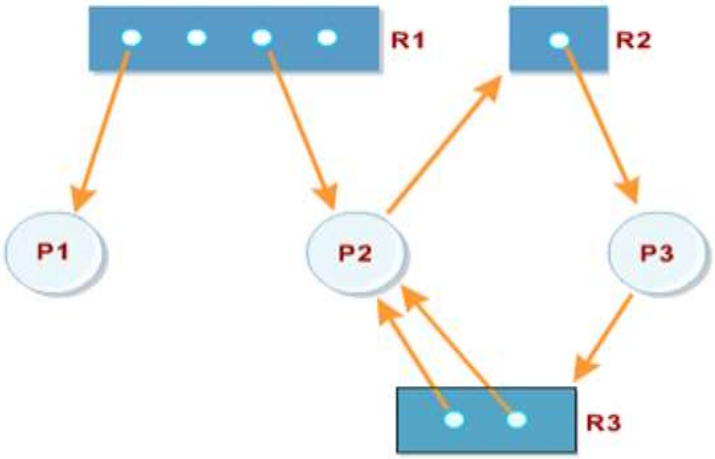
Basic Facts

If graph contains no cycles ⇒ no deadlock

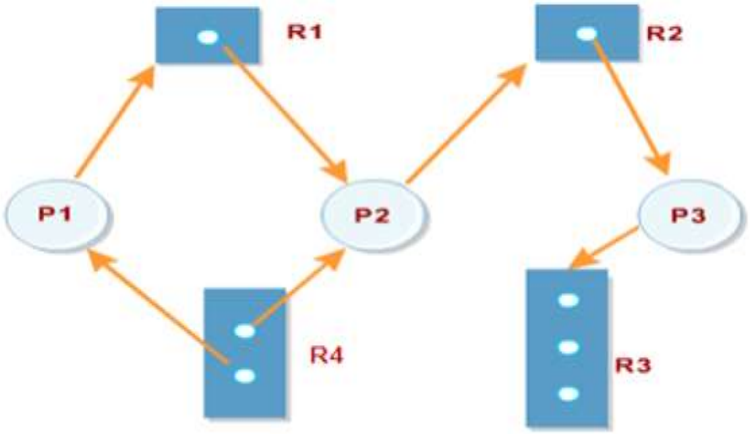
.If graph contains a cycle ⇒

- if only one instance per resource type, then deadlock.
- if several instances per resource type, possibility of deadlock.

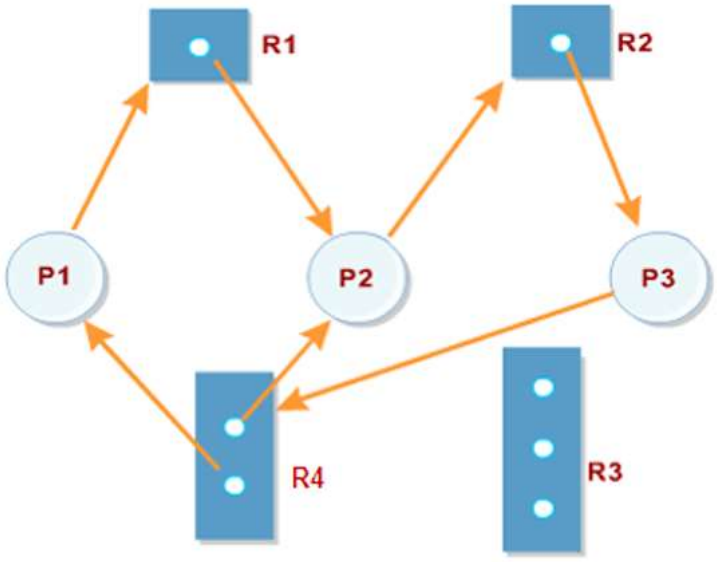
With a Deadlock or Without ??



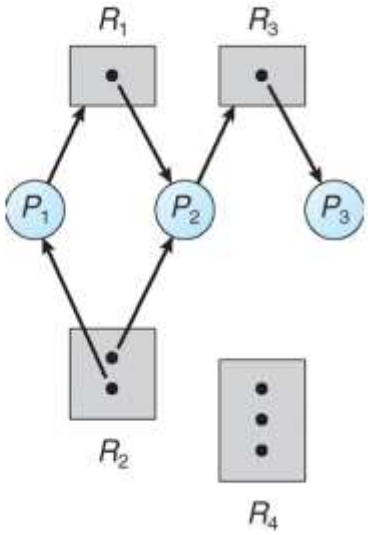
With a Deadlock or Without ??



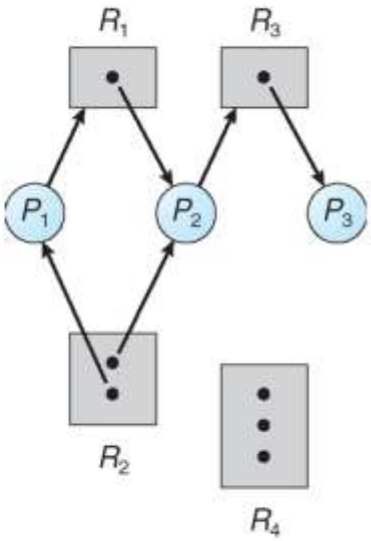
With a Deadlock or Without ??



1. Describe the following Resource-Allocation Graph.
2. Is this graph contain a cycle ?
3. Is there a deadlock? Why?



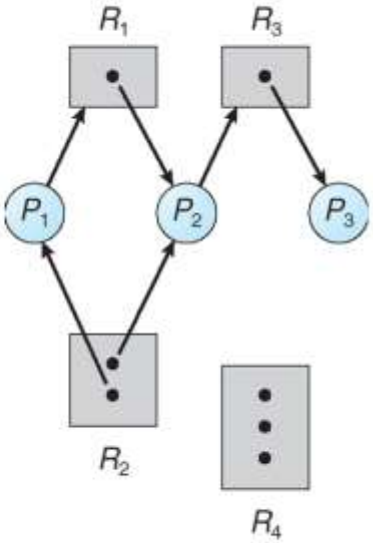
1. Describe the following Resource-Allocation Graph.



- There are 3 processes P_1, P_2, P_3
- There are 4 resources:
  - R_1(1 instance),
  - R_2(2 instances),
  - R_3(1 instance),
  - R_4(3 instances).
- P_1 holding 1 instance from R_2
- P_1 request 1 instance from R_1
- P_2 holding 1 instance from R_2
- P_2 holding 1 instance from R_1
- P_2 request 1 instance from R_3
- P_3 holding 1 instance from R_3

2. Is this graph contain a cycle ?

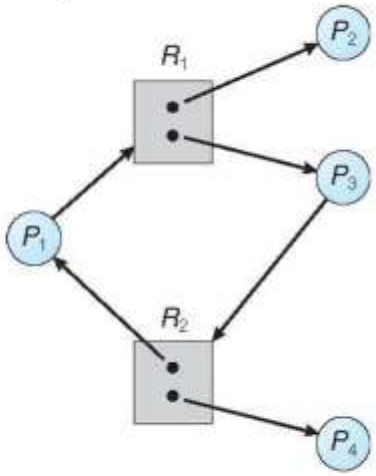
- No cycle.



3. Is there a deadlock? Why?

- No deadlock. Because there is no cycle.

1. Describe the following Resource-Allocation Graph.

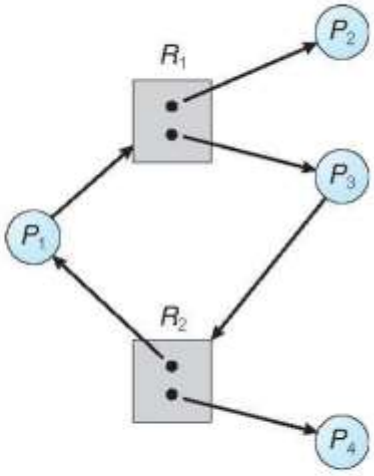


- There are 4 processes P_1, P_2, P_3, P_4
- There are 2 resources:
  - R_1(2 instances),
  - R_2(2 instances).
- P_1 holding 1 instance from R_2
- P_1 request 1 instance from R_1
- P_2 holding 1 instance from R_1
- P_3 holding 1 instance from R_1
- P_3 request 1 instance from R_2
- P_4 holding 1 instance from R_2

2. Is this graph contain a cycle ?

- There is one cycle $< R_2, P_1, R_1, P_3, R_2 >$.

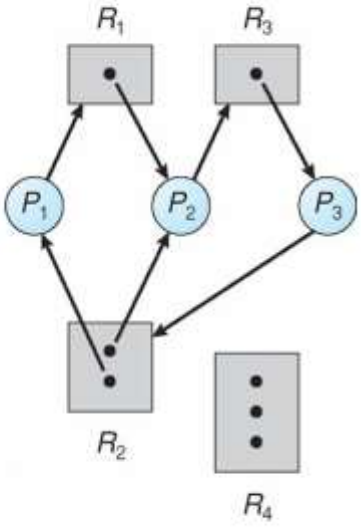
3. Is there a deadlock? Why?



- No deadlock.
- Because P_2 and P_4 will finish their jobs over time.
- Then, 1 instance from R_1 and 1 instance from R_2 will be free for P_1 and P_3.

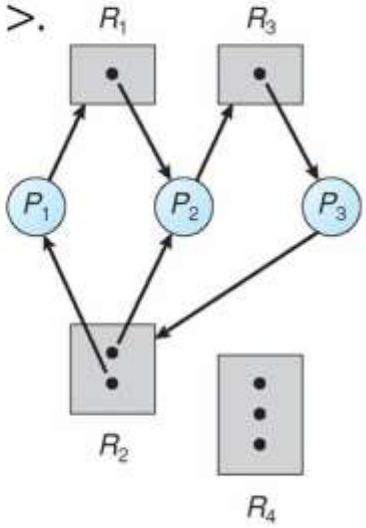
1. Describe the following Resource-Allocation Graph.

- There are 3 processes P_1, P_2, P_3
- There are 4 resources:
  - R_1(1 instance),
  - R_2(2 instances),
  - R_3(1 instance),
  - R_4(3 instances).
- P_1 holding 1 instance from R_2
- P_1 request 1 instance from R_1
- P_2 holding 1 instance from R_2
- P_2 holding 1 instance from R_1
- P_2 request 1 instance from R_3
- P_3 holding 1 instance from R_3
- P_3 request 1 instance from R_2



2. Is this graph contain a cycle ?

- There is one cycle $< R_2, P_1, R_1, P_2, R_3, P_3, R_2 >$.



3. Is there a deadlock? Why?

- Yes there is a deadlock.
- Because P_1 is waiting for P_2 and,
- P_2 is waiting for P_3 and,
- P_3 is waiting for P_1 and P_2. Over time.

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

DeadLocks

Methods for Handling Deadlocks, Deadlocks Avoidance. • Deadlocks Detection and Recovery.

Lab - 9

DeadLocks

Methods for Handling Deadlocks, Deadlocks Avoidance. • Deadlocks Detection and Recovery

Methods for Handling Deadlocks

• Ensure that the system will never enter a deadlock state:

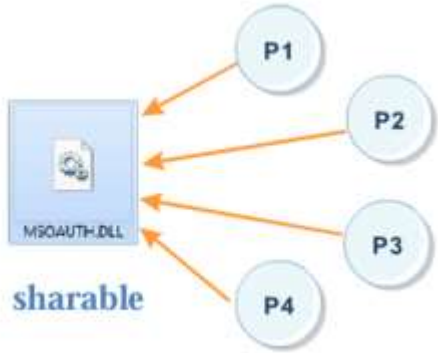
- ➢ Deadlock prevention.
- ➢ Deadlock avoidance.

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

• Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems. Deadlock Prevention

Mutual Exclusion– cannot be broken

- • We cannot prevent deadlocks by denying mutual exclusion because some resources are non-sharable.
- • Sharable resources (e.g., read-only files) accessed concurrently.



Hold and Wait – can be broken if

• A process requests a resource only if it does not hold any other resources.

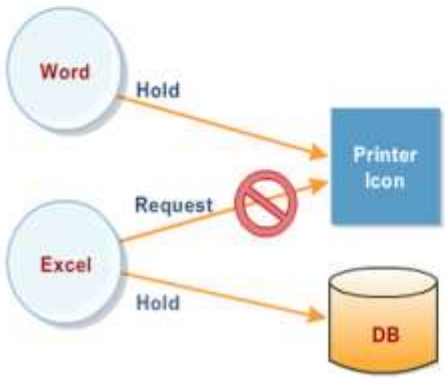
• A process requests and is allocated all its resources before it begins execution.

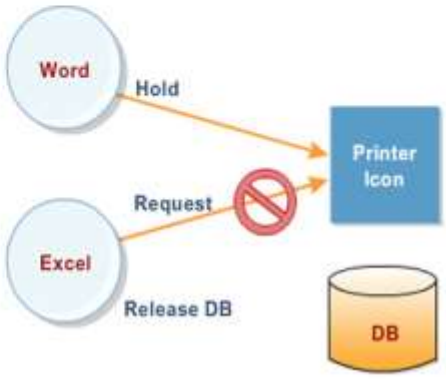
• Disadvantages:

➢ Low resource utilization; starvation possible.

No Preemption – can be broken if

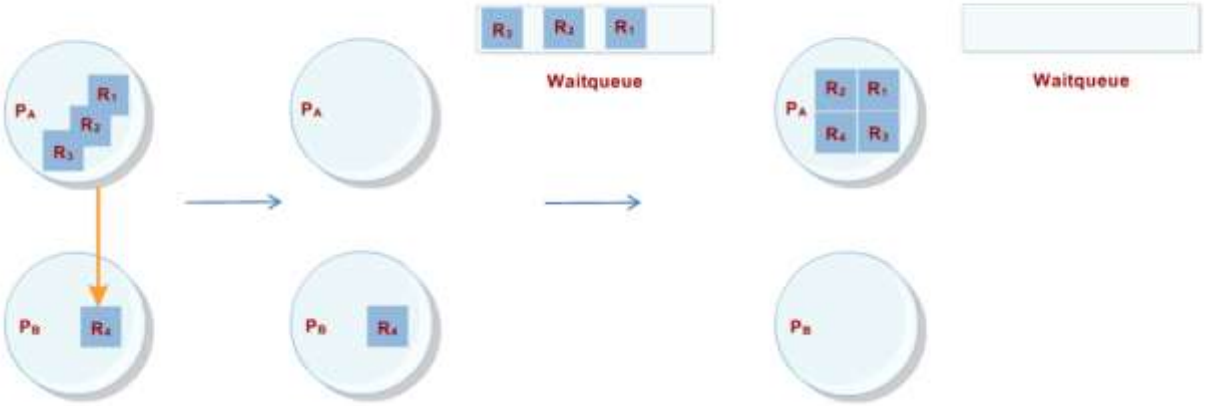
- If a process holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.





No Preemption – can be broken if

- Preempted resources are added to the list of resources for which the process is waiting.
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.



No Preemption – can be broken if

• Problems?

• Difficult to use with resources whose state are not easily saved, e.g., printers and tape drives. (In contrast to CPU registers and memory space).

Circular Wait – can be broken if

- Impose a total ordering on all resource types, and
- Require that each process requests resources in an increasing order of enumeration.

Deadlocks Avoidance

Requires that the system has some additional a priori information available.

• Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.

• The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

• Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

• When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

• System is in safe state if there exists a sequence of ALL the processes in the systems such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, with j < i.

That is:

• If Pi resource needs are not immediately available, then Pi can wait until all Pj have finished.

• When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate.

• When Pi terminates, Pi +1 can obtain its needed resources, and so on.

Basic Facts

• If a system is in safe state ⇒ no deadlocks.

• If a system is in unsafe state ⇒ possibility of deadlock.

• Avoidance ⇒ ensure that a system will never enter an unsafe state

Avoidance Algorithms:

• Single instance of a resource type ⇒

➢ Use a resource allocation graph.

• Multiple instances of a resource type ⇒

➢ Use the banker ’ s algorithm

Resource-Allocation Graph:

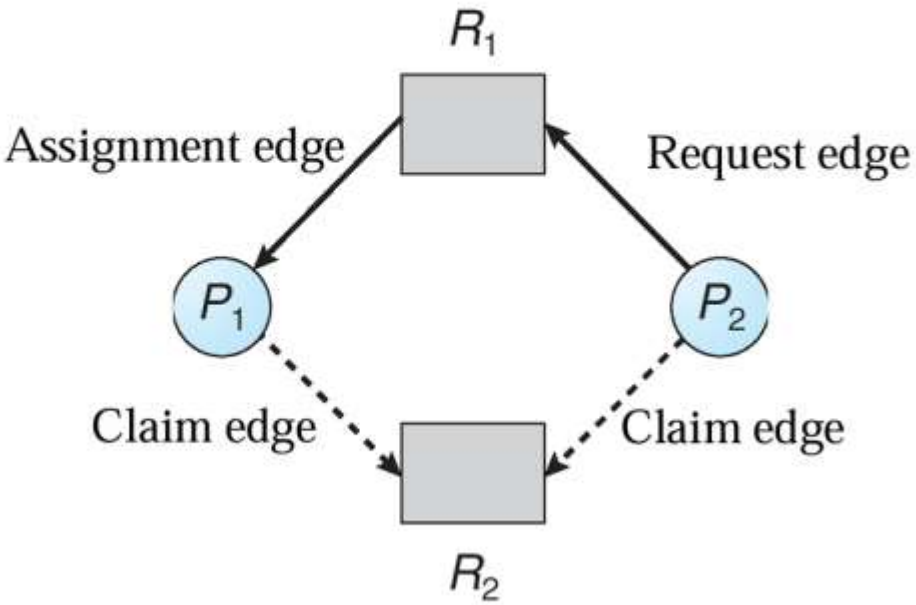
- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j; represented by a dashed line.

- Claim edge converts to request edge when a process requests a resource.

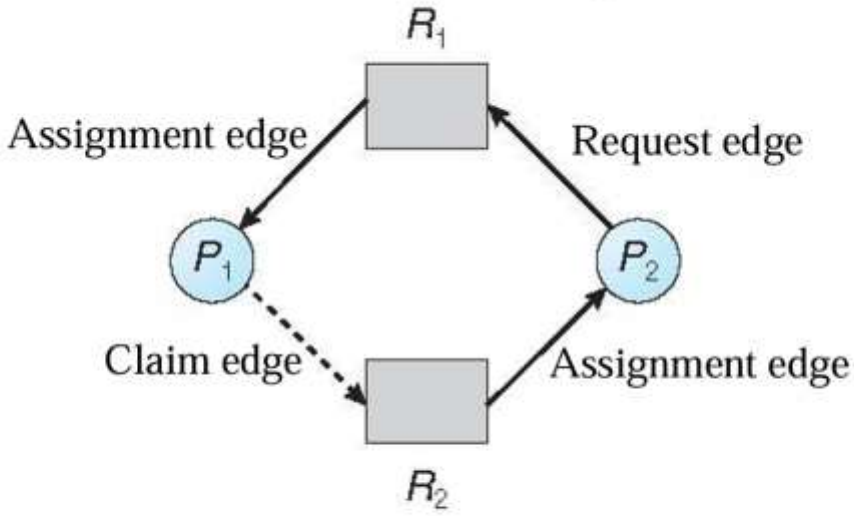
• Request edge converted to an assignment edge when the resource is allocated to the process.

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

Resource-Allocation Graph:



Unsafe State In Resource-Allocation Graph:



A cycle, as mentioned, indicates that the system is in an unsafe state.
If P_1 requests R_2, and P_2 holding R_2, then a deadlock will occur.

Resource-Allocation Graph Algorithm:

• Suppose that process Pi requests resource Rj

• The request can be granted only if converting the request edge to assignment edge does not create a cycle in the resource allocation graph.

Deadlocks Avoidance Banker ’s Algorithm:

• Multiple instances of resources.

• Each process must a priori claim maximum use.

• When a process requests a resource it may have to wait.

• When a process gets all its resources it must return them in a finite amount of time.

Deadlocks Avoidance Data Structures for the Banker ’s Algorithm:

Let n = number of processes, and m = number of resources types.

• Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available

• Max: n ×mmatrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj

• Allocation: n × m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj

• Need: n ×mmatrix. If Need[i,j] = k, then Pi may need k more instances of Rjto complete its task Need [i,j] = Max[i,j] – Allocation [i,j]

Deadlocks Avoidance Banker ’s Algorithm (Safety Algorithm): 1. Let Workand Finish be vectors of length m and n, respectively. Initialize: Work = Available Finish [i] = false for i = 0, 1, …, n- 1 2. Find an i such that both: (a) Finish [i] = false (b) Needi ≤ Work If no such i exists, go to step 4 3. Work= Work + Allocationi Finish[i] = true go to step 2 4. If Finish [i] == true for all i, then the system is in a safe state.

Deadlocks Avoidance Banker ’s Algorithm (Resource-Request Algorithm):

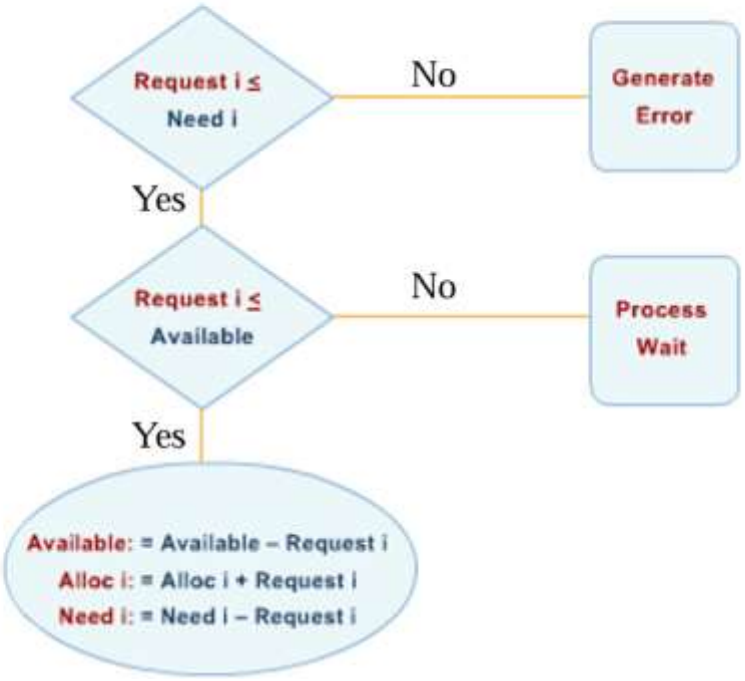
Requesti = request vector for process Pi. If Requesti [j] = k then process Pi wants k instances of resource type Rj

1. If Requesti ≤ Needi go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

2. If Requesti ≤ Available, go to step 3. Otherwise Pi must wait, since resources are not available.

3. Pretend to allocate requested resources to Pi by modifying the state as follows: Available = Available – Requesti; Allocationi= Allocationi + Requesti; Needi = Needi– Requesti; If safe ⇒ the resources are allocated to Pi. If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored.

Banker's Algorithm (Resource-Request Algorithm):



Banker's Algorithm (Example):

Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

a. What is the content of the matrix ***Need***?
b. Is the system in a safe state? Why?

Banker's Algorithm (Example):

b. Is the system in a safe state? Why?

		Allocation	*Max*	*Available*	*Need*
		A B C	*A B C*	*A B C*	*A B C*
3	P_0	0 1 0	7 5 3	10 5 7	7 4 3
1	P_1	2 0 0	3 2 2		1 2 2
4	P_2	3 0 2	9 0 2		6 0 0
2	P_3	2 1 1	2 2 2		0 1 1
5	P_4	0 0 2	4 3 3		4 3 1

The system is in a **safe state** since the sequence < P_1, P_3, P_0, P_2, P_4> **satisfies safety criteria**.

Lab - 10

DeadLocks

Deadlocks Detection and Recovery

Deadlocks Detection:

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

• An algorithm that examines the state of the system to determine whether a deadlock has occurred.

• An algorithm to recover from the deadlock.

Deadlocks Detection Single Instance of Each Resource Type:

• Maintain wait-for graph:

➢ Nodes are processes.

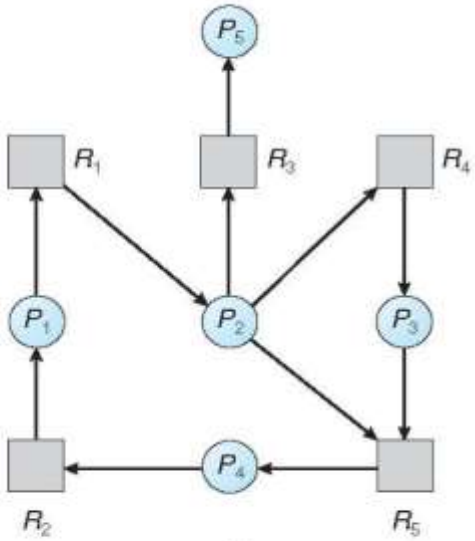
➢ Pi→Pj if Pi is waiting for Pj.

• Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.

• An algorithm to detect a cycle in a graph requires an order of n2 operations, where n is the number of vertices in the graph.

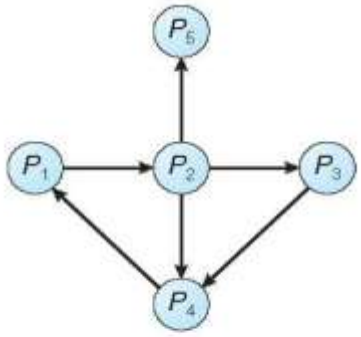
Deadlocks Detection Single Instance of Each Resource Type:

• Resource-Allocation Graph and Wait-for Graph



(a)

Resource-Allocation Graph



(b)

Corresponding wait-for graph

Several Instance of a Resource Type:

• Available: A vector of length m indicates the number of available resources of each type.

• Allocation: An n × m matrix defines the number of resources of each type currently allocated to each process.

• Request: An n × m matrix indicates the current request of each process. If Request [i][j] = k, then process Pi is requesting k more instances of resource type Rj.

Detection Algorithm:

1. Let Work and Finish be vectors of length m and n, respectively Initialize: (a) Work = Available (b) For i = 1,2, …, n, if Allocationi ≠ 0, then Finish[i] = false; otherwise, Finish[i] = true

2. Find an index i such that both: (a) Finish[i] == false (b)Requesti ≤ Work If no such i exists, go to step

3. Work = Work + Allocationi Finish[i] = true go to step 2

4. If Finish[i] == false, for some i, 1 ≤ i ≤ n, then the system is in deadlock state. Moreover, if Finish[i] == false, then Pi is deadlocked.

Algorithm requires an order of O(m × the system is in deadlocked state. n 2) operations to detect whether

Detection Algorithm (Example1):

a. Is the system is in deadlock state? Why?

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

	Allocation	*Request*	*Available*	*Work*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2		
P_2	3 0 3	0 0 0		
P_3	2 1 1	1 0 0		
P_4	0 0 2	0 0 2		

	Allocation	Request	Available	Work	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	0 0 0	0 0 0	0 0 0	false
P_1	2 0 0	2 0 2			false
P_2	3 0 3	0 0 0			false
P_3	2 1 1	1 0 0			false
P_4	0 0 2	0 0 2			false

	Allocation	Request	Available	Work	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	0 0 0	0 0 0	0 0 0	false
P_1	2 0 0	2 0 2			false
P_2	3 0 3	0 0 0			false
P_3	2 1 1	1 0 0			false
P_4	0 0 2	0 0 2			false

	Allocation	Request	Available	Work	Finish	
	A B C	A B C	A B C	A B C		
P_0	0 1 0	0 0 0	0 0 0	0 1 0	true	1
P_1	2 0 0	2 0 2			false	
P_2	3 0 3	0 0 0			false	
P_3	2 1 1	1 0 0			false	
P_4	0 0 2	0 0 2			false	

	Allocation	*Request*	*Available*	*Work*	*Finish*	
	A B C	*A B C*	*A B C*	*A B C*		
P_0	0 0 0	0 0 0	0 0 0	3 1 3	true	1
P_1	2 0 0	2 0 2			false	
P_2	3 0 3	0 0 0			true	2
P_3	2 1 1	1 0 0			false	
P_4	0 0 2	0 0 2			false	

	Allocation	*Request*	*Available*	*Work*	*Finish*	
	A B C	*A B C*	*A B C*	*A B C*		
P_0	0 0 0	0 0 0	0 0 0	5 2 4	true	1
P_1	2 0 0	2 0 2			false	
P_2	0 0 0	0 0 0			true	2
P_3	2 1 1	1 0 0			true	3
P_4	0 0 2	0 0 2			false	

	Allocation	*Request*	*Available*	*Work*	*Finish*	
	A B C	*A B C*	*A B C*	*A B C*		
P_0	0 0 0	0 0 0	0 0 0	5 2 6	true	1
P_1	2 0 0	2 0 2			false	
P_2	0 0 0	0 0 0			true	2
P_3	0 0 0	0 0 0			true	3
P_4	0 0 2	0 0 2			true	4

	Allocation	*Request*	*Available*	*Work*	*Finish*	
	A B C	*A B C*	*A B C*	*A B C*		
P_0	0 0 0	0 0 0	0 0 0	7 2 6	true	1
P_1	2 0 0	2 0 2			true	5
P_2	0 0 0	0 0 0			true	2
P_3	0 0 0	0 0 0			true	3
P_4	0 0 0	0 0 0			true	4

	Allocation	*Request*	*Available*	*Work*	*Finish*	
	A B C	*A B C*	*A B C*	*A B C*		
P_0	0 0 0	0 0 0	0 0 0	7 2 6	true	1
P_1	0 0 0	0 0 0			true	5
P_2	0 0 0	0 0 0			true	2
P_3	0 0 0	0 0 0			true	3
P_4	0 0 0	0 0 0			true	4

We claim that the system is **not** in a **deadlocked** state. Indeed, if we execute our algorithm, we will find that the sequence $< P_0, P_2, P_3, P_4, P_1 >$ results in *Finish*[*i*] == *true* for all *i*.

Recovery from Deadlock:

• Process Termination.

• Resource Preemption

Process Termination:

1. Abort all deadlocked processes.

2. Abort one process at a time until the deadlock cycle is eliminated.

3. In which order should we choose to abort?

- ➢ Priority of the process.
- ➢ How long process has computed, and how much longer to completion.
- ➢ Resources the process has used.
- ➢ Resources process needs to complete.
- ➢ How many processes will need to be terminated.
- ➢ Is process interactive or batch?