

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Dr. Rajashree Dash
Associate Professor

Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Dr. Rajashree Dash
Associate Professor, CSE**

Contents

0 Course Overview	1
0.1 Course Description	1
0.2 Course Objectives	1
0.3 Course Details	1
0.4 Text Book(s) and Reference(s)	2
0.5 Lesson Plan	2
0.6 Course Outcomes	5
0.7 Program Outcomes & Program Specific Outcomes	5
0.7.1 Program Outcomes (POs)	5
0.7.2 Program Specifis Outcomes (PSOs)	7

Lecture 0

Course Overview

0.1 Course Description

The course introduces the concepts relating to operating systems. The course covers the topics: Introduction, Operating System structures, Processes, Threads, Process Synchronisation, CPU Scheduling, Deadlocks, Main memory, virtual memory, file system interface, file system implementation, I/O systems, Introduction to LINUX, Memory addressing, Processes, threads, Interrupts, exceptions, kernel synchronization, Timing measurements, process scheduling, memory management, Process address space, system calls, signals, virtual file system, I/O, page cache, accessing files.. After the completion of this course the student should have the thorough knowledge of various operating system services along with a case study of LINUX operating system.

0.2 Course Objectives

1. To discuss the different components of Operating System and its structures.
2. To understand and analyze the mechanisms involved in handling, scheduling, synchronizing processes and threads.
3. To identify the issues involved due to deadlock and the strategies used to resolve them.
4. To demonstrate the different schemes available in operating system for managing resources such as memory, file and peripheral devices.
5. To provide an insight view of Linux kernel with respect to functional aspects of its different components.

0.3 Course Details

Subject Code : CSE 4049

Subject Name : "Design of Operating system"

Offered in : 5th Semester, B.Tech (CSE & CSIT)

Credits : 4

Grading pattern : 1

Contact Hours : 5 hours/Week (3 Classes/Week, 1hr/Class, 1 Lab/Week, 2hr/Lab)

0.4 Text Book(s) and Reference(s)

Text Book:

- (1) Abraham Silberschatz, Peter B. Galvin & Greg Gagne, *Operating System Concepts*, Wiley.
- (2) Daniel P. bovet & Marco Cesati, *Understanding the Linux kernel*, SPD.

0.5 Lesson Plan

<i>Contact hour</i>	<i>Topics to be covered</i>	<i>Remarks (if any)</i>
L-0	Introduction to the course and its motivation. Course description, objective, credit, grading pattern, class and Lab session of the course. NBA provided program outcomes and departmental specific program specific outcomes.	(Class: w.e.f from 23-09-2020), Silberschatz, Galvin & Gagne
L-1	Introduction to Operating Systems, Computer system organization	Silberschatz, Galvin & Gagne
L-2	Computer system architecture, Operating system structure	Silberschatz, Galvin & Gagne
L-3	Operating system operations, Computing Environments	Silberschatz, Galvin & Gagne
L-4	Operating system services, user and operating system interface	Silberschatz, Galvin & Gagne
L-5	System calls, types of system calls, system programs,	Silberschatz, Galvin & Gagne

L-6	Operating system structure: Simple structure, Monolithic kernel, layered approach, micro kernel, Modular approach	Silberschatz, Galvin & Gagne
L-7	Process concepts, Process state, Process control block	Silberschatz, Galvin & Gagne
L-8	Process scheduling: LTS,MTS,STS,Context switching	Silberschatz, Galvin & Gagne
L-9	Operations on processes, problem solving	Silberschatz, Galvin & Gagne
L-10	Co-Operating process and independent process , Inter process Communications, Shared memory solution to bounded buffer problem	Silberschatz, Galvin & Gagne
L-11	Message passing ,direct and indirect communication, synchronous and asynchronous message passing , Buffering	Silberschatz, Galvin & Gagne
L-12	Thread concept overview, Multicore programming	Silberschatz, Galvin & Gagne
L-13	Multithreading models,	Silberschatz, Galvin & Gagne
L-14	CPU scheduling : Basic concepts, scheduling criteria, scheduling algorithm:FCFS with same Arrival time	Silberschatz, Galvin & Gagne
L-15	FCFS with different Arrival time,SJF,SRTF	Silberschatz, Galvin & Gagne
L-16	Priority Scheduling, Round Robin Scheduling,	Silberschatz, Galvin & Gagne
L-17	Multilevel queue scheduling, Multilevel feedback queue scheduling,	Silberschatz, Galvin & Gagne
L-18	Process Synchronization: Background, critical section problem	Silberschatz, Galvin & Gagne
L-19	Peterson solution to critical section problem	Silberschatz, Galvin & Gagne
L-20	Synchronization hardware,test-and-set and compare-and-swap, Mutex lock	Silberschatz, Galvin & Gagne

L-21	Semaphore,types of semaphore,semaphore implementation	Silberschatz, Galvin & Gagne
L-22	Classical problems of synchronization, Bounded buffer and first Readers writers problem with semaphore	Silberschatz, Galvin & Gagne
L-23	2nd Readers writers problem, Dinning philosophers problem with semaphore solution,	Silberschatz, Galvin & Gagne
L-24	Monitor, Monitor solutions to Dinning philosophers problem	Silberschatz, Galvin & Gagne
L-25	Deadlocks, Necessary condition, Deadlock prevention	Silberschatz, Galvin & Gagne
L-26	Deadlock avoidance, Banker's algorithm	Silberschatz, Galvin & Gagne
L-27	Deadlock detection and recovery	Silberschatz, Galvin & Gagne
L-28	Memory management strategy,background, basic hardware	Silberschatz, Galvin & Gagne
L-29	Dynamic memory allocation, segmentation ,compaction	Silberschatz, Galvin & Gagne
L-30	Paging	Silberschatz, Galvin & Gagne
L-31	Hardware support for paging ,protection structure for page table	Silberschatz, Galvin & Gagne
L-32	Multilevel paging , hash page table ,inverted page table	Silberschatz, Galvin & Gagne
L-33	Segmentation	Silberschatz, Galvin & Gagne
L-34	Virtual memory management	Silberschatz, Galvin & Gagne
L-35	Demand paging, copy on write	Silberschatz, Galvin & Gagne

L-36	Page replacement,FIFO,OPTIMAL,LRU	Silberschatz, Galvin & Gagne
L-37	LRU approximation page replacement , Counting based page replacement	Silberschatz, Galvin & Gagne
L-38	Page buffering , Frame allocation algorithm	Silberschatz, Galvin & Gagne
L-39	Global vs local allocation , Thrashing	Silberschatz, Galvin & Gagne

0.6 Course Outcomes

By the end of course through lectures, laboratory works, and exams students will be able to:

- CO 1. Understand the different components of Operating System and various ways of structuring an operating system .
- CO 2. Analyze the mechanisms involved in handling, scheduling, synchronizing processes and threads
- CO 3. Learn the different methods used to prevent and deal with deadlock
- CO 4. Explore various memory management, file handling and input output schemes, analyzing their effectiveness in different scenario
- CO 5. Gain knowledge about various data structures and functions used for process management, scheduling, synchronization, memory and file management in Linux operating system

0.7 Program Outcomes & Program Specific Outcomes

0.7.1 Program Outcomes (POs)

There are twelve program outcomes (1-12) for the Computer science & Engineering B. Tech program:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

0.7.2 Program Specific Outcomes (PSOs)

- PSO 1. The ability to understand, analyze and develop computer programs in the areas related to business intelligence, web design and networking for efficient design of computer-based systems of varying complexities.
- PSO 2. The ability to apply standard practices and strategies in software development using open-ended programming environments to deliver a quality product for business success.



image/logosoa.png

Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Dr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

1	Introduction to Operating Systems, Computer system organization	1
1.1	Introduction	1
1.1.1	Computer System Structure	1
1.1.2	Role of Operating System	2
1.2	Computer System Organization	3
1.2.1	Computer Startup	4
1.2.2	Computer System Operation	4
1.2.3	Common functions of Interrupts	5
1.2.4	Interrupt Handling	5
1.2.5	Storage Structure	6
1.2.6	I/O structure	6

Lecture 1

Introduction to Operating Systems, Computer system organization

1.1 Introduction

An operating system (OS) is the software component of a computer system that is responsible for the management and coordination of activities and the sharing of the resources of the computer. It acts as an intermediary between a user of a computer and the computer hardware. It acts as a host for application programs that are run on the machine. As a host, one of the purposes of an OS is to handle the details of the operation of the hardware. This relieves application programs from having to manage these details and makes it easier to write applications. Almost all computers use an OS of some type. OSs offer a number of services to application programs and users. Applications access these services through application programming interfaces (APIs) or system calls. By using these interfaces, the application can request a service from the OS, pass parameters, and receive the results of the operation. Users may also interact with the OS by typing commands or using a graphical user interface (GUI).

The goals of operating system are:

- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

1.1.1 Computer System Structure

Computer system can be divided into four components:

- Hardware – provides basic computing resources
CPU, memory, I/O devices
- Operating system - Controls and coordinates use of hardware among various applications and users

- Application programs – define the ways in which the system resources are used to solve the computing problems of the users
Word processors, compilers, web browsers, database systems, video games
- Users
People, machines, other computers

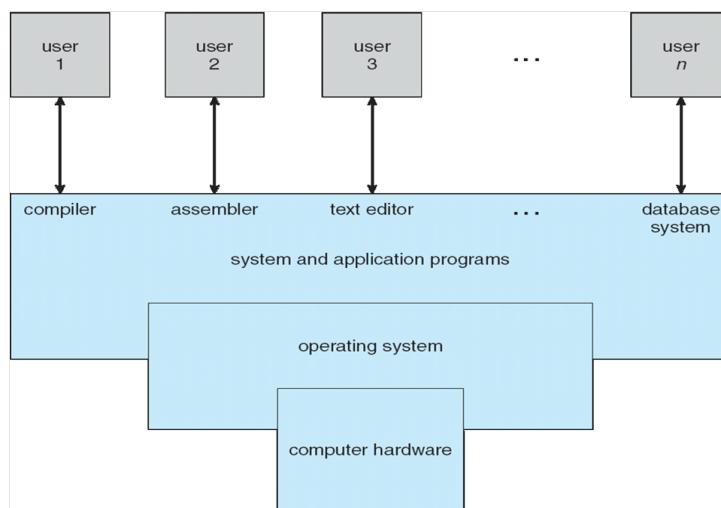


Figure 1.1: Abstract view of components of a computer system

1.1.2 Role of Operating System

Role of the operating system depends on the point of view.

- User view
The user's view of the computer varies according to the interface being used.
 - In **personal computers** the goal is to maximize the work that the user is performing. In this case, the operating system is designed mostly for ease of use, with some attention paid to performance and none paid to resource utilization—how various hardware and software resources are shared. Performance is, of course, important to the user; but such systems are optimized for the single-user experience rather than the requirements of multiple users.
 - In **mainframe and minicomputers**, a user sits at a terminal connected to a mainframe or a minicomputer and other users are accessing the same computer through other terminals. These users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization—to assure that all available CPU time, memory, and I/O are used efficiently and that no individual user takes more than her fair share.
 - In **workstations**, users sit at workstations connected to networks of other workstations and servers. These users have dedicated resources at their disposal, but they also share resources such as networking and servers, including file,

compute, and print servers. Therefore, their operating system is designed to compromise between individual usability and resource utilization.

- Recently, many varieties of [mobile computers](#), such as smartphones and tablets, are the standalone units used by individual users. Quite often, they are connected to networks through cellular or other wireless technologies. The user interface for mobile computers generally features a touch screen, where the user interacts with the system by pressing and swiping fingers across the screen rather than using a physical keyboard and mouse. Mobile operating systems often include not only a core kernel but also middleware a set of software frameworks that provide additional services to application developers. For example, each of the two most prominent mobile operating systems — Apple's iOS and Google's Android features a core kernel along with middleware that supports databases, multimedia, and graphics.
- Some computers have little or no user view. For example, [embedded computers](#) in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems are designed primarily to run without user intervention.

- System view

From the computer's point of view, the operating system is the program most intimately involved with the hardware.

- OS is a [resource allocator](#), that manages all resources and decides between conflicting requests for efficient and fair resource use
- OS is a [control program](#), that controls execution of programs to prevent errors and improper use of the computer

So operating system is the one program running at all times on the computer usually called the kernel. (Along with the kernel, there are two other types of programs: system programs, which are associated with the operating system but are not necessarily part of the kernel, and application programs, which include all programs not associated with the operation of the system.)

1.2 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. Each device controller is in charge of a specific type of device . The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

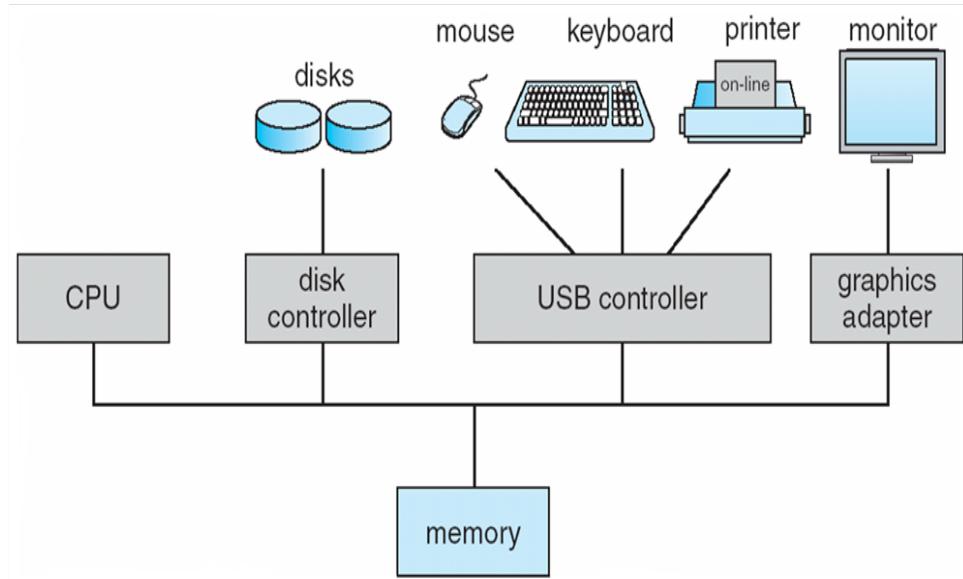


Figure 1.2: Modern computer system

1.2.1 Computer Startup

- When a computer is powered up an initial program (bootstrap program stored in ROM or EEPROM) runs.
- It initializes all aspects of the system, from CPU registers to device controllers to memory contents.
- It locates the operating-system kernel and load it into memory.
- Once the kernel is loaded and executing, it can start providing services to the system and its users.
- Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become system processes, or system daemons that run the entire time the kernel is running. On UNIX, the first system process is “init,” and it starts many other daemons.
- Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

1.2.2 Computer System Operation

- I/O devices and the CPU can execute concurrently.
- Each device controller is in charge of a particular device type.
- Each device controller has a local buffer and some registers.
- CPU moves data from/to main memory to/from local buffers.
- I/O is from the device to local buffer of controller.

- Device controller informs CPU that it has finished its operation by causing an interrupt

1.2.3 Common functions of Interrupts

- Occurrence of an event is usually signaled by an interrupt either from hardware or software.
- H/w may trigger interrupt at any time by sending signal to the CPU through system bus. S/w may trigger interrupt through system call.
- When CPU is interrupted it stops what it is doing and immediately transfers the control to a fixed location.
- The fixed location contains the starting address where the service routine for the interrupt is located.
- The interrupt service routine executes and on completion , CPU resumes the interrupted computation

1.2.4 Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter.
- A generic routine is invoked to examine the interrupt information ie. determines which type of interrupt has occurred.
- The routine would call the interrupt specific handler.
- A separate segments of code ie. Interrupt service routine determines what action should be taken for each type of interrupt.
- To speedup the interrupt handling, a table of pointers to interrupt routines is used. The table of pointers is stored in low memory.These locations hold the addresses of the interrupt service routines for the various devices.
- This array or interrupt vector of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device.
- The interrupt architecture also save the address of the interrupted instruction.
- After the interrupt is serviced the saved return address is loaded into the program counter and the interrupted computation resumes.

1.2.5 Storage Structure

All forms of memory provide an array of bytes. Each byte has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses. In a typical instruction–execution cycle, a system with a von Neumann architecture, first fetches an instruction from memory and stores that instruction in the instruction register. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory. The memory unit sees only a stream of memory addresses without knowing how they are generated or what they are for. The wide variety of storage systems can be organized in a hierarchy according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases.

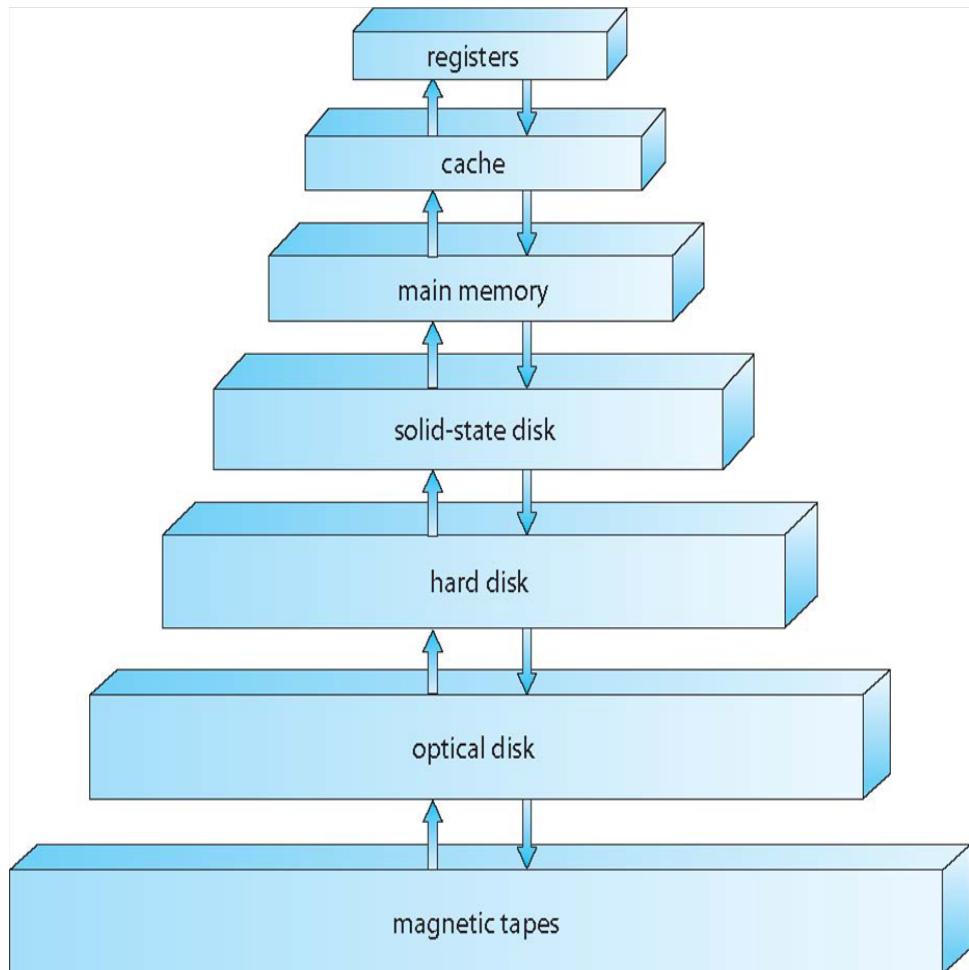


Figure 1.3: Storage Device Hierarchy

1.2.6 I/O structure

A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus. Each device controller is in charge of a specific type of device. Depending on the controller, more than one device may be attached. A device

controller maintains some local buffer storage and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage. Typically, operating systems have a device driver for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device.

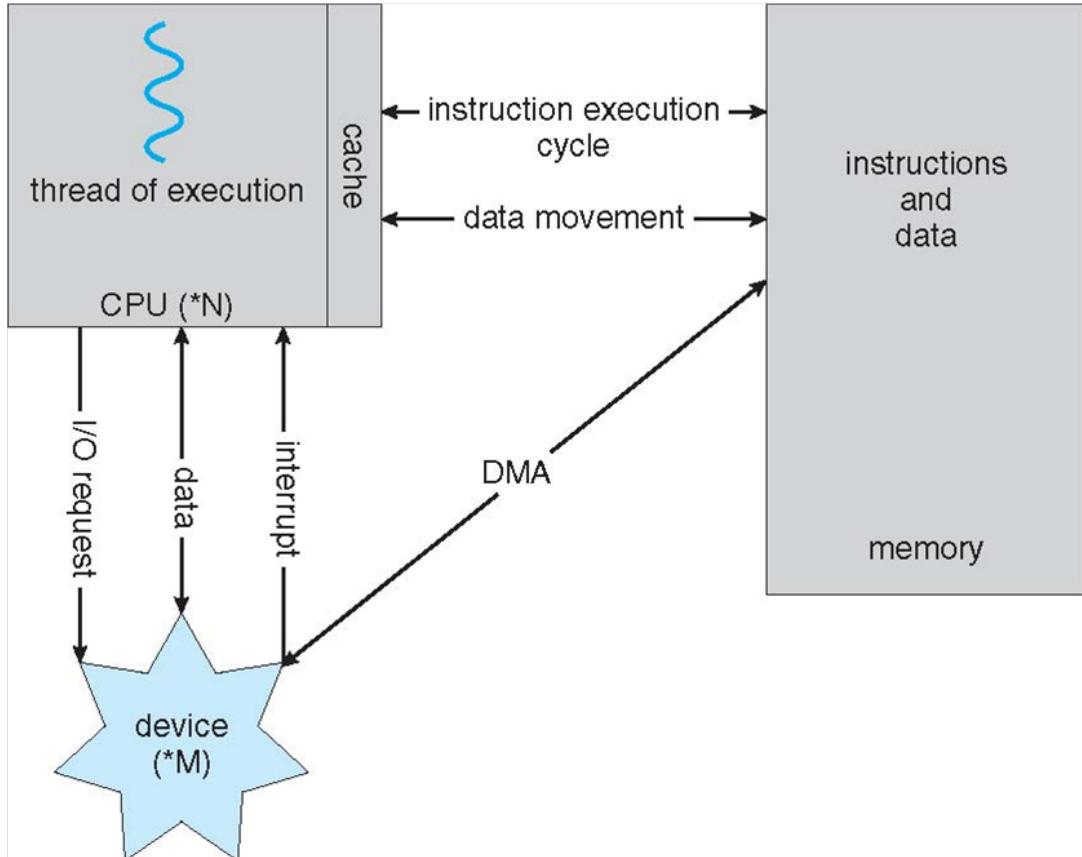


Figure 1.4: How modern Computer works

- Interrupt driven I/O
 - To start an I/O operation, the device driver loads the appropriate registers within the device controller.
 - Device controller examines the content of these registers to determine what action to take.
 - The controller starts transfer of data from device to local buffer.
 - Once data transfer is complete, the device controller informs the device driver that it has finished operation.
 - Device driver then returns control to OS possibly returning the data or pointer to the data or status information.
- Direct Memory Access

- This form of interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O. To solve this problem, direct memory access (DMA) is used.
- After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU.
- Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices.
- While the device controller is performing these operations, the CPU is available to accomplish other work.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

2 Computer system architecture, Operating system structure	1
2.1 Computer system architecture	1
2.1.1 Single Processor System	1
2.1.2 Multiprocessor or Parallel System	1
2.1.3 Clustered System	3
2.2 Operating System Structure	4
2.2.1 Simple Batch System	4
2.2.2 Multiprogramming	4
2.2.3 Time sharing or Multitasking	5

Lecture 2

Computer system architecture, Operating system structure

2.1 Computer system architecture

A computer system can be organized in a number of different ways according to the number of general-purpose processors used.

2.1.1 Single Processor System

- On a single processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes.
- These systems have other special-purpose processors such as device-specific processors, graphics controllers and so on.
- All of these special-purpose processors run a limited instruction set and do not run user processes.
- Sometimes, they are managed by the operating system or they do their jobs autonomously.

2.1.2 Multiprocessor or Parallel System

- In Multiprocessors systems (also known as parallel systems, tightly-coupled systems) more than one processors remain in close communication sharing computer bus, clock, sometimes memory and peripheral devices.
- Advantages:
 - Increased throughput : More work is done in less time but the speed-up ratio with N processors is not N , however; rather, it is less than N .
 - Economy of scale : Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage,

and power supplies. If several programs operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them than to have many computers with local disks and many copies of the data.

- Increased reliability : If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. The ability to continue providing service proportional to the level of surviving hardware is called [graceful degradation](#). Some systems go beyond graceful degradation and are called [fault tolerant](#), because they can suffer a failure of any single component and still continue operation.
- Types of Multiprocessors:
 - [Asymmetric Multiprocessing](#) – Each processor is assigned a specific task. A boss processor controls the system; the other processors either look to the boss for instruction or have predefined tasks. This scheme defines a boss-worker relationship. The boss processor schedules and allocates work to the worker processors.
 - [Symmetric Multiprocessing](#) – Each processor performs all tasks within operating system. All systems are peer. No boss-worker relationship exists between processors.

• Multicore

Recent trend in CPU design is to include multiple computing cores in a single chip (i.e. Multiprocessor chips). These are more efficient than multiple chips with single core.

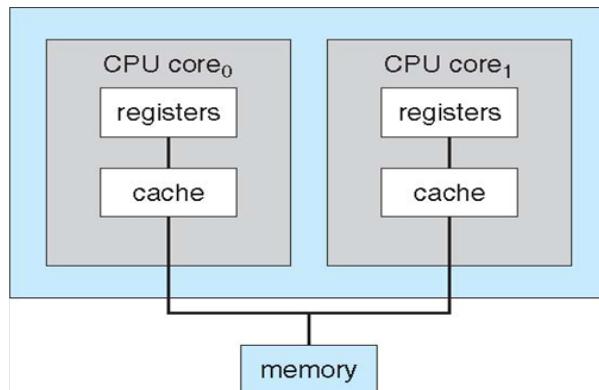


Figure 2.1: Dual core design

Advantage:

- On chip communication is faster than between chip communication.
- Uses significantly less power.

Well suited for server systems such as database, web servers.

2.1.3 Clustered System

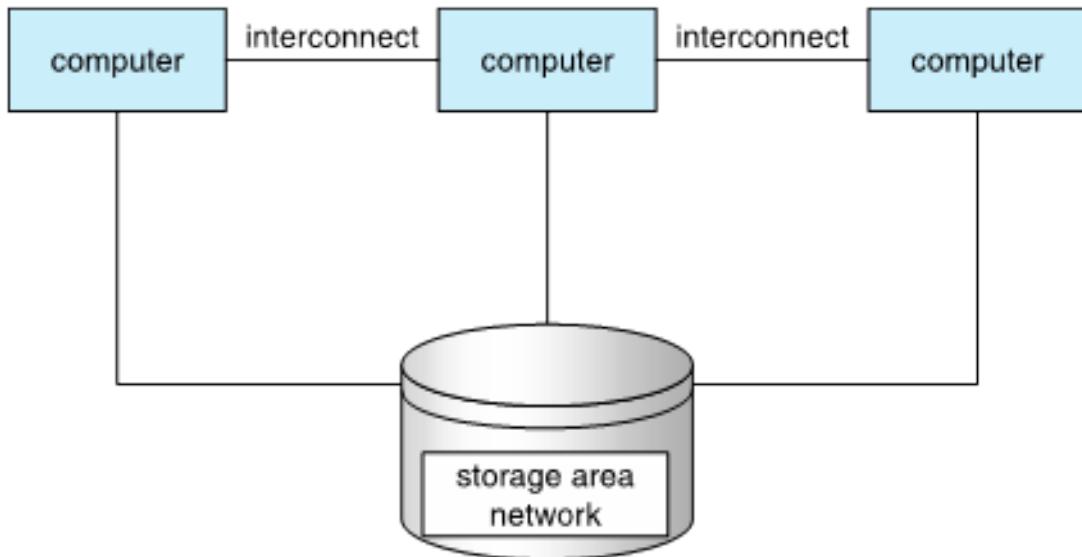


Figure 2.2: Structure of Clustered system

- It allows multiple machines to perform computations on data contained on shared storage and let computing continue in the case of failure of some subset of cluster members.
- Like multiprocessor systems, but multiple systems working together.
- Usually share storage and are closely linked via LAN.
- Provides a high-availability service which survives failures.
- Asymmetric clustering has one machine in hot-standby mode, while others are running the applications. The hot standby host monitors the active server. If the server fails it becomes the active server.
- Symmetric clustering has multiple nodes running applications, monitoring each other.
- Clusters can also be used to provide high-performance computing environments. Such systems can supply significantly greater computational power than single-processor or even SMP systems because they can run an application concurrently on all computers in the cluster. The application uses parallelization so that it is divided into separate components that run in parallel on individual computers in the cluster. Typically, these applications are designed so that once each computing node in the cluster has solved its portion of the problem, the results from all the nodes are combined into a final solution.

- Other forms of clusters include parallel clusters and clustering over a WAN. Parallel clusters allow multiple hosts to access the same data on the shared storage. To provide this shared access, the system uses distributed lock manager (DLM) that supplies access control and locking to ensure that no conflicting operation occurs.
- Cluster technology are also using storage area networks (SANs) which allow many systems to attach to a pool of storage. If the applications and their data are stored on the SAN, then the cluster software can assign the application to run on any host that is attached to the SAN. If the host fails, then any other host can take over. In a database cluster, dozens of hosts can share the same database, greatly increasing performance and reliability.

2.2 Operating System Structure

Internally, operating systems vary greatly in their makeup, since they are organized along many different lines.

2.2.1 Simple Batch System

- Enormously large machines run from a console. No direct interaction of user with system. User prepared a job consisting of program , data and some control information about nature of the job and submitted it to the computer operator. At some later time the o/p is appeared consisting result of the program as well as a dump of memory and registers in case of error.
- OS is simple and always in memory. Main task of OS is to transfer control automatically from one job to next. To speed up processing, jobs with similar needs were batched together and when computer became available would run each batch. Jobs are processed in order of their submission.
- Disadvantage:
Lack of interaction of user and job while it is executing.
CPU is often idle.

2.2.2 Multiprogramming

- Multiprogramming is a technique to execute no. of programs simultaneously by a single processor.
- Many processes are in state of execution at the same time.
- Multiprogramming organizes jobs so that CPU always has one to execute.
- All jobs are kept in job pool. OS selects a set of job and keeps in main memory. One job is selected and run via CPU scheduling. When it has to wait (for I/O for example), OS switches to another job.

- It increases CPU utilization and decreases total time needed to execute the jobs.
- Multiprogrammed systems provide an environment in which the various system resources are utilized effectively, but they do not provide for user interaction with the computer system.

2.2.3 Time sharing or Multitasking

- Timesharing (multitasking) is logical extension of multi programming, in which CPU switches jobs so frequently that users can interact with each job while it is running, creating interactive computing.
- Response time is very short. As system switches rapidly from one user to the next, each user is given the impression of having his own computer, but actually one computer is being shared among many users.
- Each user has at least one program executing in memory.
- If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them (Job scheduling).
- if several jobs are ready to run at the same time, the system must choose which job will run first (CPU scheduling).
- If processes don't fit in memory, swapping moves them in and out to run. Virtual memory allows execution of processes not completely in memory.
- A time-sharing system must also provide a file system. The file system resides on a collection of disks; hence, disk management must be provided. In addition, a time-sharing system provides a mechanism for protecting resources from inappropriate use.
- To ensure orderly execution, the system must provide mechanisms for job synchronization and communication, and it may ensure that jobs do not get stuck in a deadlock, forever waiting for one another



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by
Mr. Rakesh Kumar
Assistant Professor

Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

3 Operating System Operations, Computing Environments	1
3.1 Operating-System Operations	1
3.1.1 Dual-Mode and Multimode Operation	1
3.1.2 Timer	2
3.2 Process Management	2
3.3 Memory Management	3
3.4 Storage Management	3
3.4.1 File-System Management	3
3.4.2 Mass-Storage Management	3
3.4.3 Caching	3
3.4.4 I/O Systems	4
3.5 Protection and Security	5
3.6 Computing Environments	5
3.6.1 Traditional Computing	5
3.6.2 Mobile Computing	5
3.6.3 Distributed Systems	6
3.6.4 Client-Server Computing	6
3.6.5 Peer-to-Peer Computing	7
3.6.6 Virtualization	8
3.6.7 Cloud Computing	9
3.6.8 Real-Time Embedded Systems	10

Lecture 3

Operating System Operations, Computing Environments

3.1 Operating-System Operations

Interrupt-driven nature of modern OSes requires that erroneous processes not be able to disturb anything else.

3.1.1 Dual-Mode and Multimode Operation

- User mode when executing harmless code in user applications.
- Kernel mode (a.k.a. system mode, supervisor mode, privileged mode) when executing potentially dangerous code in the system kernel.
- Certain machine instructions (privileged instructions) can only be executed in kernel mode.
- Kernel mode can only be entered by making system calls. User code cannot flip the mode switch.
- Modern computers support dual-mode operation in hardware, and therefore most modern OSes support dual-mode operation.
- The concept of modes can be extended beyond two, requiring more than a single mode bit.
- CPUs that support virtualization use one of these extra bits to indicate when the virtual machine manager, VMM, is in control of the system. The VMM has more privileges than ordinary user programs, but not so many as the full kernel.
- System calls are typically implemented in the form of software interrupts, which causes the hardware's interrupt handler to transfer control over to an appropriate interrupt handler, which is part of the operating system, switching the mode bit to kernel mode in the process. The interrupt handler checks exactly which interrupt

was generated, checks additional parameters (generally passed through registers) if appropriate, and then calls the appropriate kernel service routine to handle the service requested by the system call.

- User programs' attempts to execute illegal instructions (privileged or non-existent instructions), or to access forbidden memory areas, also generate software interrupts, which are trapped by the interrupt handler and control is transferred to the OS, which issues an appropriate error message, possibly dumps data to a log (core) file for later analysis, and then terminates the offending program.

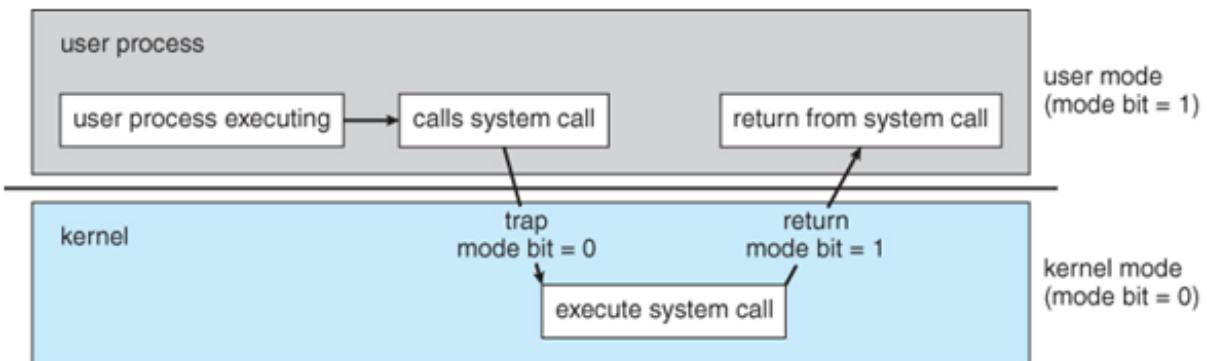


Figure 3.1: Transition from user to kernel mode

3.1.2 Timer

- Before the kernel begins executing user code, a timer is set to generate an interrupt.
- The timer interrupt handler reverts control back to the kernel.
- This assures that no user process can take over the system.
- Timer control is a privileged instruction, (requiring kernel mode.)

3.2 Process Management

An OS is responsible for the following tasks with regards to process management:

- Creating and deleting both user and system processes.
- Ensuring that each process receives its necessary resources, without interfering with other processes.
- Suspending and resuming processes.
- Process synchronization and communication.
- Deadlock handling.

3.3 Memory Management

An OS is responsible for the following tasks with regards to memory management:

- Keeping track of which blocks of memory are currently in use, and by which processes.
- Determining which blocks of code and data to move into and out of memory, and when.
- Allocating and deallocating memory as needed. (E.g. new, malloc)

3.4 Storage Management

3.4.1 File-System Management

An OS is responsible for the following tasks with regards to file system management:

- Creating and deleting files and directories.
- Supporting primitives for manipulating files and directories. (open, flush, etc.)
- Mapping files onto secondary storage.
- Backing up files onto stable permanent storage media.

3.4.2 Mass-Storage Management

An OS is responsible for the following tasks with regards to mass-storage management:

- Free disk space management.
- Storage allocation.
- Disk scheduling.

Note the trade-offs regarding size, speed, longevity, security, and re-writability between different mass storage devices, including floppy disks, hard disks, tape drives, CDs, DVDs, etc.

3.4.3 Caching

- There are many cases in which a smaller higher-speed storage space serves as a cache, or temporary storage, for some of the most frequently needed portions of larger slower storage areas.
- The hierarchy of memory storage ranges from CPU registers to hard drives and external storage. (See table below.)

- The OS is responsible for determining what information to store in what level of cache, and when to transfer data from one level to another.
- The proper choice of cache management can have a profound impact on system performance.
- Data read in from disk follows a migration path from the hard drive to main memory, then to the CPU cache, and finally to the registers before it can be used, while data being written follows the reverse path. Each step (other than the registers) will typically fetch more data than is immediately needed, and cache the excess in order to satisfy future requests faster. For writing, small amounts of data are frequently buffered until there is enough to fill an entire "block" on the next output device in the chain.
- The issues get more complicated when multiple processes (or worse multiple computers) access common data, as it is important to ensure that every access reaches the most up-to-date copy of the cached data (amongst several copies in different cache levels.)

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Figure 3.2: Performance of various levels of storage

3.4.4 I/O Systems

The I/O subsystem consists of several components:

- A memory-management component that includes buffering, caching, and spooling.
- A general device-driver interface.
- (UNIX implements multiple device interfaces for many types of devices, one for accessing the device character by character and one for accessing the device block by block. These can be seen by doing a long listing of /dev, and looking for a "c" or "b" in the first position. You will also note that the "size" field contains two

numbers, known as the major and minor device numbers, instead of the normal one. The major number signifies which device driver handles I/O for this device, and the minor number is a parameter passed to the driver to let it know which specific device is being accessed. Where a device can be accessed as either a block or character device, the minor numbers for the two options usually differ by a single bit.)

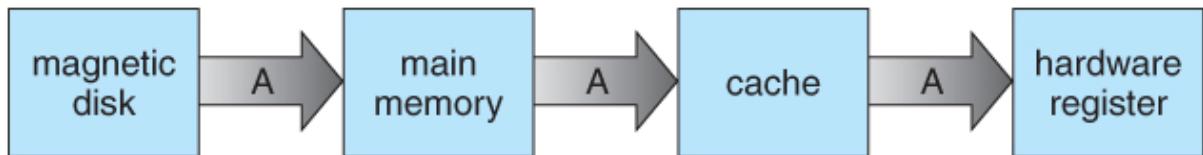


Figure 3.3: Migration of integer A from disk to register

3.5 Protection and Security

- **Protection** involves ensuring that no process access or interfere with resources to which they are not entitled, either by design or by accident. (E.g. "protection faults" when pointer variables are misused.)
- **Security** involves protecting the system from deliberate attacks, either from legitimate users of the system attempting to gain unauthorized access and privileges, or external attackers attempting to access or damage the system.

3.6 Computing Environments

3.6.1 Traditional Computing

- Stand-alone general purpose machines.
- But blurred as most systems interconnect with others (i.e., the Internet).
- Portals provide web access to internal systems.
- Network computers (thin clients) are like Web terminals.
- Mobile computers interconnect via wireless networks.
- Networking becoming ubiquitous – even home systems use firewalls to protect home computers from Internet attacks.

3.6.2 Mobile Computing

- Computing on small handheld devices such as smart phones or tablets. (As opposed to laptops, which still fall under traditional computing.)

- May take advantage of additional built-in sensors, such as GPS, tilt, compass, and inertial movement.
- Typically connect to the Internet using wireless networking (IEEE 802.11) or cellular telephone technology.
- Limited in storage capacity, memory capacity, and computing power relative to a PC.
- Generally uses slower processors, that consume less battery power and produce less heat.
- The two dominant OSes today are Google Android and Apple iOS.

3.6.3 Distributed Systems

- Distributed Systems consist of multiple, possibly heterogeneous, computers connected together via a network and cooperating in some way, form, or fashion.
- Networks may range from small tight LANs to broad reaching WANs.
 - **WAN = Wide Area Network**, such as an international corporation
 - **MAN =Metropolitan Area Network**, covering a region the size of a city for example.
 - **LAN =Local Area Network**, typical of a home, business, single-site corporation, or university campus.
 - **PAN = Personal Area Network**, such as the bluetooth connection between your PC, phone, headset, car, etc.
- Network access speeds, throughputs, reliabilities, are all important issues.
- OS view of the network may range from just a special form of file access to complex well-coordinated network operating systems.
- Shared resources may include files, CPU cycles, RAM, printers, and other resources.

3.6.4 Client-Server Computing

- A defined server provides services (HW or SW) to other systems which serve as clients. (Technically clients and servers are processes, not HW, and may co-exist on the same physical computer.)
- A process may act as both client and server of either the same or different resources.
- Served resources may include disk space, CPU cycles, time of day, IP name information, graphical displays (X Servers), or other resources.

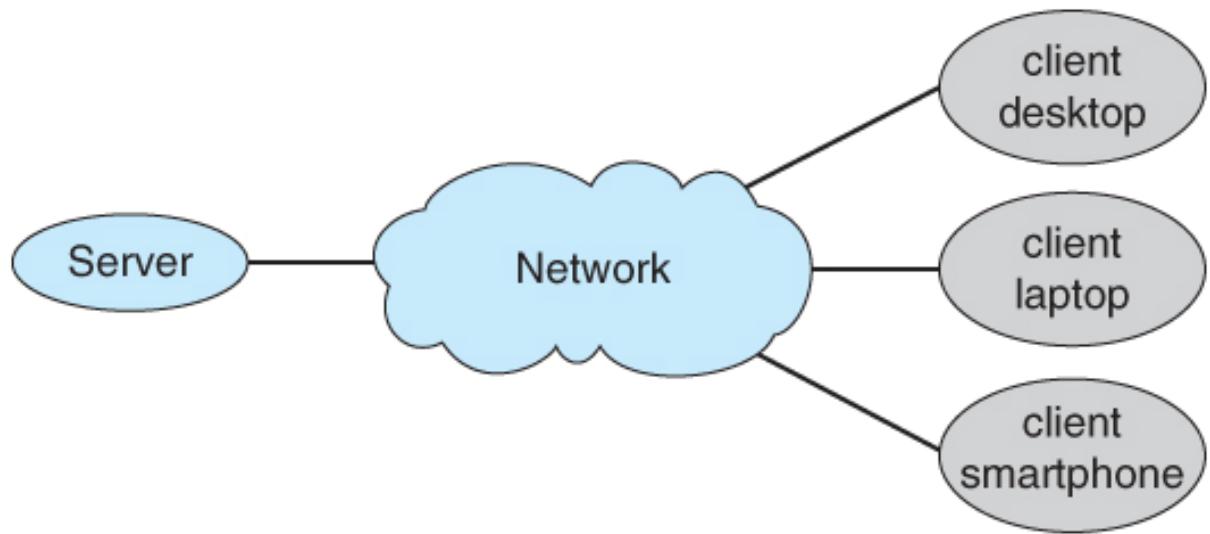


Figure 3.4: General structure of a client-server system

3.6.5 Peer-to-Peer Computing

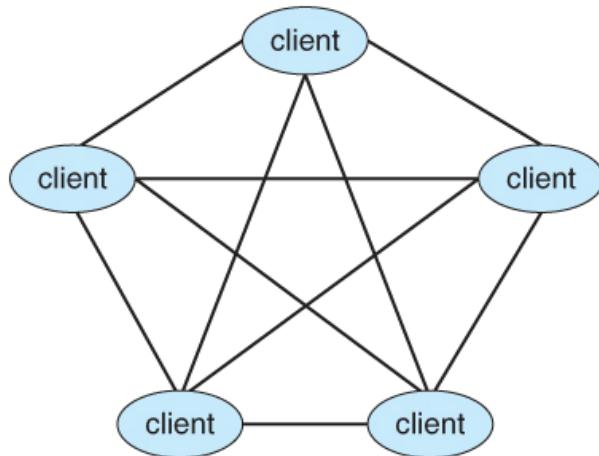


Figure 3.5: Peer-to-peer system with no centralized service

- Any computer or process on the network may provide services to any other which requests it. There is no clear "leader" or overall organization.
- May employ a central "directory" server for looking up the location of resources, or may use peer-to-peer searching to find resources.
- E.g. Skype uses a central server to locate a desired peer, and then further communication is peer to peer.

3.6.6 Virtualization

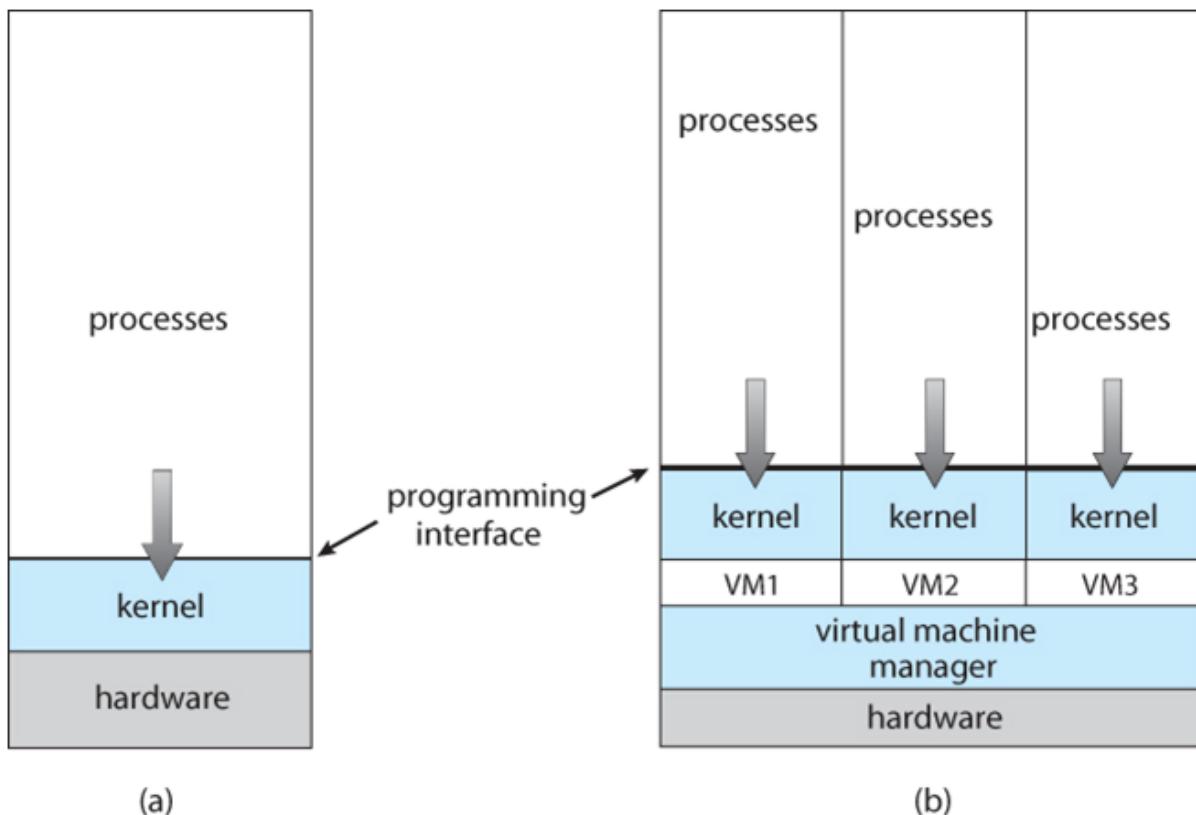


Figure 3.6: VMWare

- Allows one or more "guest" operating systems to run on virtual machines hosted by a single physical machine and the virtual machine manager.
- Useful for cross-platform development and support.
- For example, a student could run UNIX on a virtual machine, hosted by a virtual machine manager on a Windows based personal computer. The student would have full root access to the virtual machine, and if it crashed, the underlying Windows machine should be unaffected.
- System calls have to be caught by the VMM and translated into (different) system calls made to the real underlying OS.
- Virtualization can slow down programs that have to run through the VMM, but can also speed up some things if virtual hardware can be accessed through a cache instead of a physical device.

3.6.7 Cloud Computing

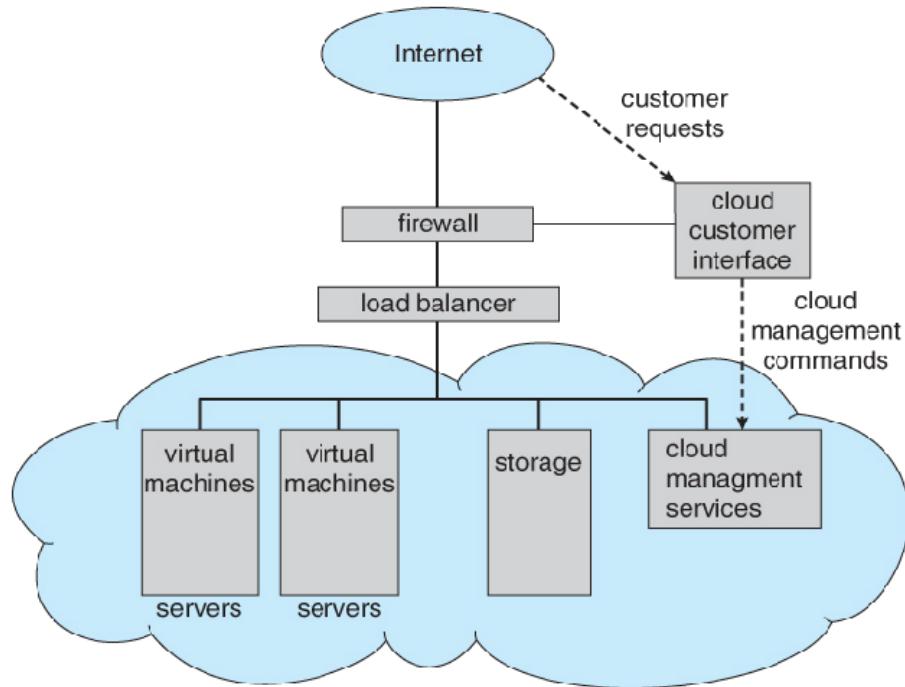


Figure 3.7: Cloud computing

- Delivers computing, storage, and applications as a service over a network.
- Types of cloud computing:
 - **Public cloud** - Available to anyone willing to pay for the service.
 - **Private cloud** - Run by a company for internal use only.
 - **Private cloud** - Run by a company for internal use only.
 - **Hybrid cloud** - A cloud with both public and private components.
 - **Software as a Service - SaaS** - Applications such as word processors available via the Internet.
 - **Platform as a Service - PaaS** - A software stack available for application use, such as a database server.
 - **Infrastructure as a Service - IaaS** - Servers or storage available on the Internet, such as backup servers, photo storage, or file storage.
 - Service providers may provide more than one type of service.
- Clouds may contain thousands of physical computers, millions of virtual ones, and petabytes of total storage.
- Web hosting services may offer (one or more) virtual machine(s) to each of their clients.

3.6.8 Real-Time Embedded Systems

- Embedded into devices such as automobiles, climate control systems, process control, and even toasters and refrigerators.
- May involve specialized chips, or generic CPUs applied to a particular task. (Consider the current price of 80286 or even 8086 or 8088 chips, which are still plenty powerful enough for simple electronic devices such as kids toys.)
- Process control devices require real-time (interrupt driven) OSes. Response time can be critical for many such devices.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by
Mr.Rakesh Kumar
Assistant Professor

Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

4 Operating System Services, User and Operating System Interface	1
4.1 Operating System Services	1
4.2 User Operating-System Interface	2
4.2.1 Command Interpreter	2
4.2.2 Graphical User Interface, GUI	4
4.2.3 Choice of Interface	5

Lecture 4

Operating System Services, User and Operating System Interface

4.1 Operating System Services

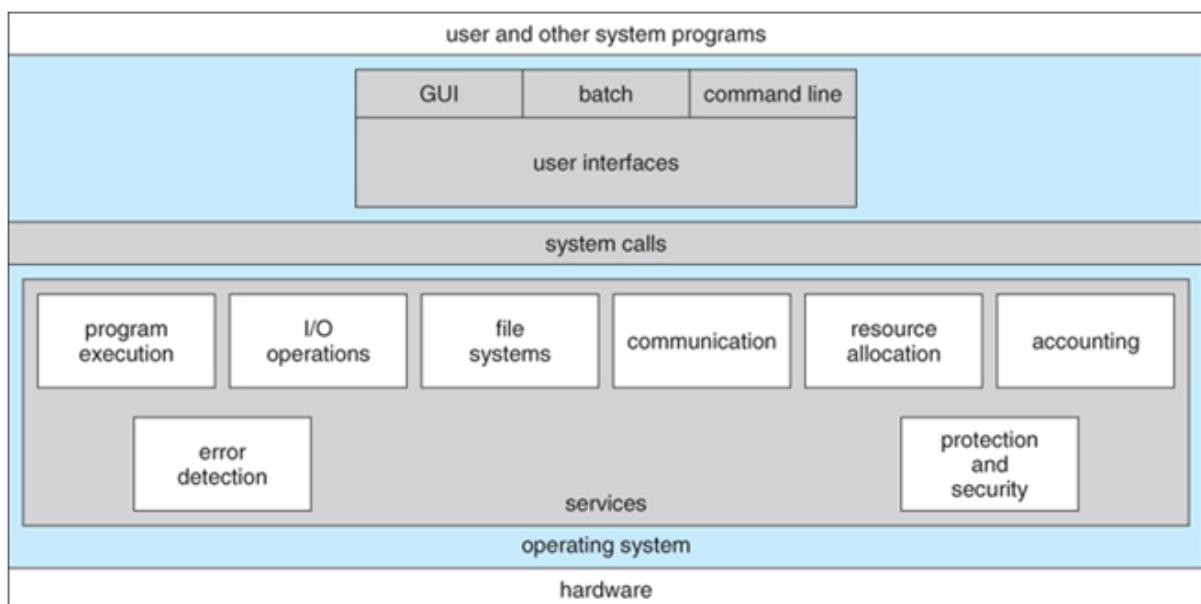


Figure 4.1: A view of operating system services

OSes provide environments in which programs run, and services for the users of the system, including:

- **User Interfaces** - Means by which users can issue commands to the system. Depending on the system these may be a command-line interface (e.g. sh, csh, ksh, tcsh, etc.), a GUI interface (e.g. Windows, X-Windows, KDE, Gnome, etc.), or a batch command systems. The latter are generally older systems using punch cards of job-control language, JCL, but may still be used today for specialty systems designed for a single purpose.

- **Program Execution** - The OS must be able to load a program into RAM, run the program, and terminate the program, either normally or abnormally.
- **I/O Operations** - The OS is responsible for transferring data to and from I/O devices, including keyboards, terminals, printers, and storage devices.
- **File-System Manipulation** - In addition to raw data storage, the OS is also responsible for maintaining directory and subdirectory structures, mapping file names to specific blocks of data storage, and providing tools for navigating and utilizing the file system.
- **Communications** - Inter-process communications, IPC, either between processes running on the same processor, or between processes running on separate processors or separate machines. May be implemented as either shared memory or message passing, (or some systems may offer both.)
- **Error Detection** - Both hardware and software errors must be detected and handled appropriately, with a minimum of harmful repercussions. Some systems may include complex error avoidance or recovery systems, including backups, RAID drives, and other redundant systems. Debugging and diagnostic tools aid users and administrators in tracing down the cause of problems.

Other systems aid in the efficient operation of the OS itself:

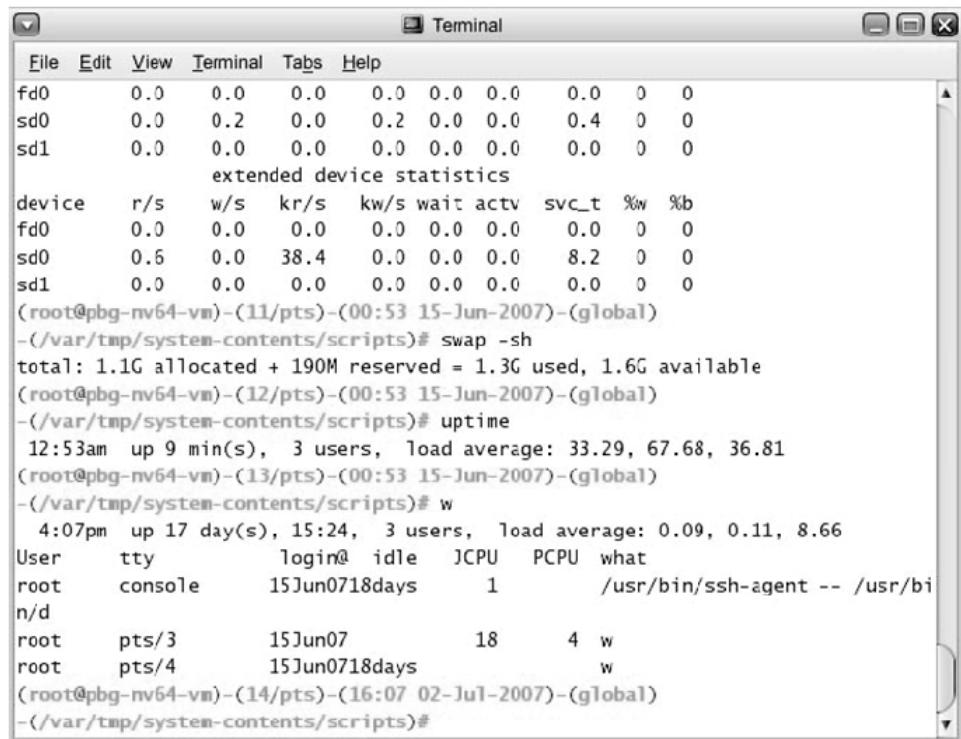
- **Resource Allocation** - E.g. CPU cycles, main memory, storage space, and peripheral devices. Some resources are managed with generic systems and others with very carefully designed and specially tuned systems, customized for a particular resource and operating environment.
- **Accounting** - Keeping track of system activity and resource usage, either for billing purposes or for statistical record keeping that can be used to optimize future performance.
- **Protection and Security** - Preventing harm to the system and to resources, either through wayward internal processes or malicious outsiders. Authentication, ownership, and restricted access are obvious parts of this system. Highly secure systems may log all process activity down to excruciating detail, and security regulation dictate the storage of those records on permanent non-erasable medium for extended times in secure (off-site) facilities.

4.2 User Operating-System Interface

4.2.1 Command Interpreter

- Gets and processes the next user request, and launches the requested programs.

- In some systems the CI may be incorporated directly into the kernel.
- More commonly the CI is a separate program that launches once the user logs in or otherwise accesses the system.
- UNIX, for example, provides the user with a choice of different shells, which may either be configured to launch automatically at login, or which may be changed on the fly. (Each of these shells uses a different configuration file of initial settings and commands that are executed upon startup.)
- Different shells provide different functionality, in terms of certain commands that are implemented directly by the shell without launching any external programs. Most provide at least a rudimentary command interpretation structure for use in shell script programming (loops, decision constructs, variables, etc.)
- An interesting distinction is the processing of wild card file naming and I/O redirection. On UNIX systems those details are handled by the shell, and the program which is launched sees only a list of filenames generated by the shell from the wild cards. On a DOS system, the wild cards are passed along to the programs, which can interpret the wild cards as the program sees fit.



The screenshot shows a terminal window titled "Terminal". The menu bar includes "File", "Edit", "View", "Terminal", "Tabs", and "Help". The window displays a series of system status commands:

```

fd0      0.0    0.0    0.0    0.0  0.0  0.0    0.0    0  0
sd0      0.0    0.2    0.0    0.2  0.0  0.0    0.4    0  0
sd1      0.0    0.0    0.0    0.0  0.0  0.0    0.0    0  0
                                         extended device statistics
device   r/s     w/s    kr/s   kw/s wait  actv  svc_t %w  %b
fd0      0.0    0.0    0.0    0.0  0.0  0.0    0.0    0  0
sd0      0.6    0.0   38.4    0.0  0.0  0.0    8.2    0  0
sd1      0.0    0.0    0.0    0.0  0.0  0.0    0.0    0  0
(root@pbg-nv64-vm)-(11/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)#
total: 1.1G allocated + 190M reserved = 1.3G used, 1.6G available
(root@pbg-nv64-vm)-(12/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)#
uptime
12:53am up 9 min(s), 3 users, load average: 33.29, 67.68, 36.81
(root@pbg-nv64-vm)-(13/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)#
w
 4:07pm up 17 day(s), 15:24, 3 users, load average: 0.09, 0.11, 8.66
User      tty          login@  idle   JCPU   PCPU what
root      console      15Jun0718days   1      /usr/bin/ssh-agent -- /usr/bi
n/d
root      pts/3        15Jun07           18      4  w
root      pts/4        15Jun0718days           w
(root@pbg-nv64-vm)-(14/pts)-(16:07 02-Jul-2007)-(global)
-/var/tmp/system-contents/scripts)#

```

Figure 4.2: The Bourne shell command interpreter in Solaris 10

4.2.2 Graphical User Interface, GUI

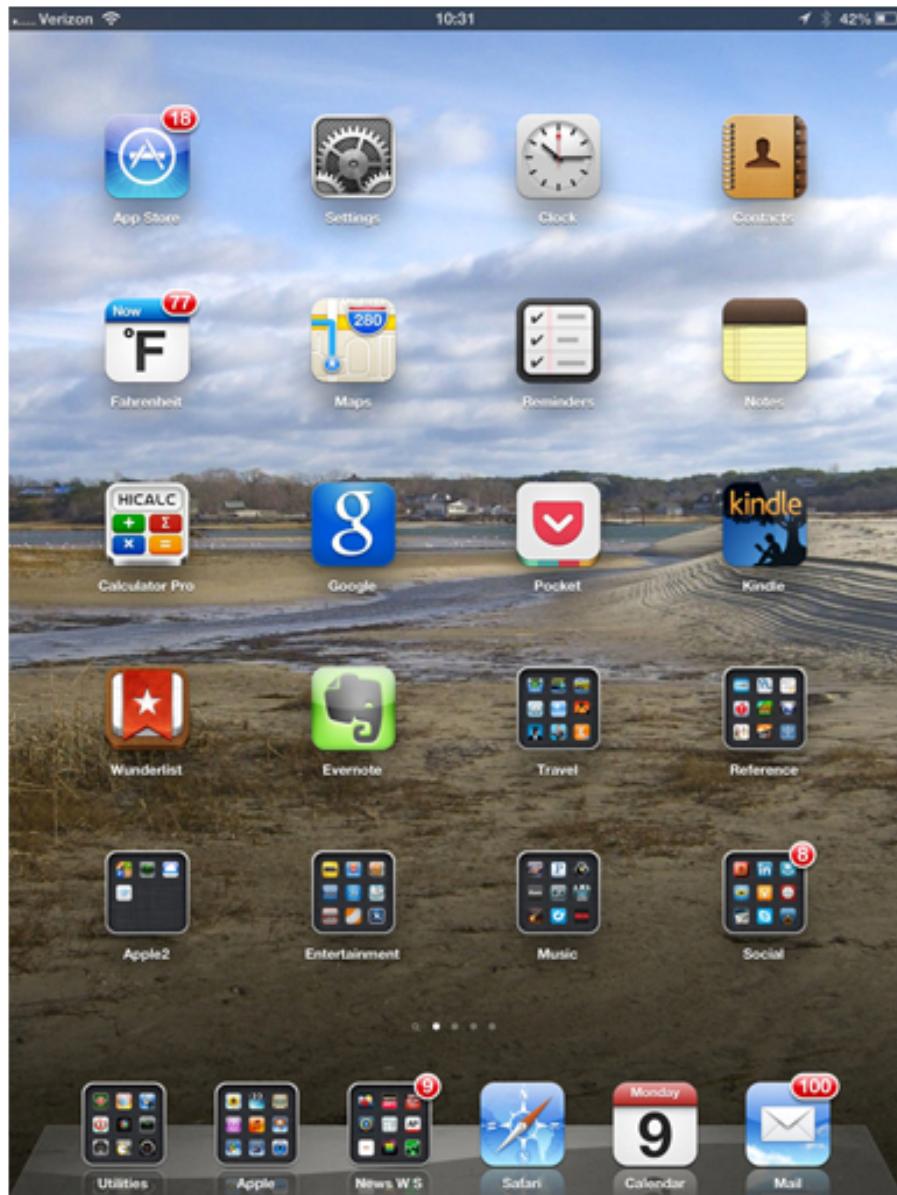


Figure 4.3: The iPad touchscreen

- Generally implemented as a desktop metaphor, with file folders, trash cans, and resource icons.
- Icons represent some item on the system, and respond accordingly when the icon is activated.
- This assures that no user process can take over the system. First developed in the early 1970's at Xerox PARC research facility.
- In some systems the GUI is just a front end for activating a traditional command line interpreter running in the background. In others the GUI is a true graphical shell in its own right.

- Mac has traditionally provided ONLY the GUI interface. With the advent of OSX (based partially on UNIX), a command line interface has also become available.
- Because mice and keyboards are impractical for small mobile devices, these normally use a touch-screen interface today, that responds to various patterns of swipes or "gestures". When these first came out they often had a physical keyboard and/or a trackball of some kind built in, but today a virtual keyboard is more commonly implemented on the touch screen.

4.2.3 Choice of Interface

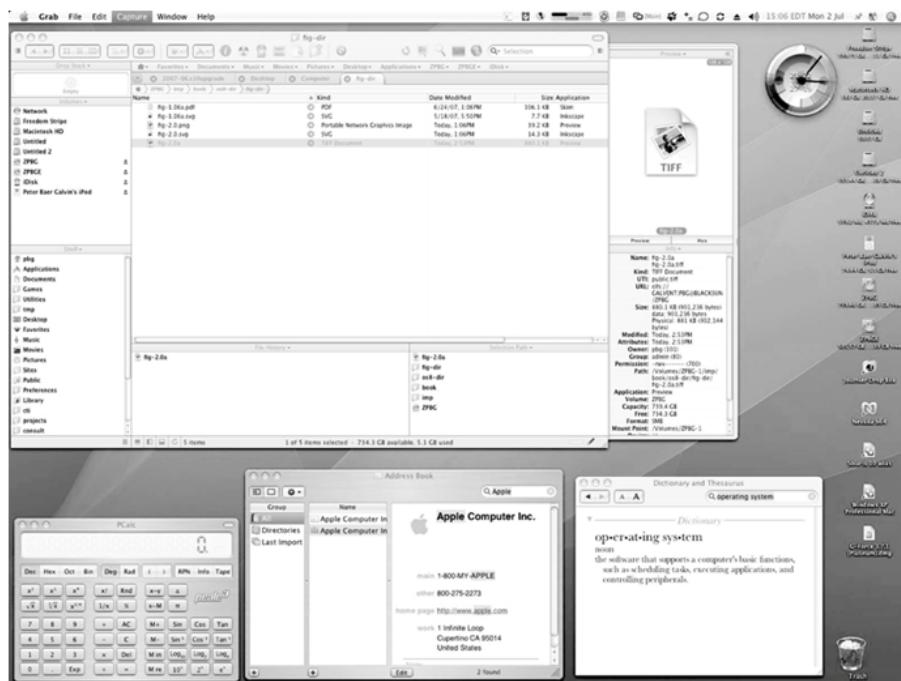


Figure 4.4: The Mac OS X GUI

- Most modern systems allow individual users to select their desired interface, and to customize its operation, as well as the ability to switch between different interfaces as needed. System administrators generally determine which interface a user starts with when they first log in.
- GUI interfaces usually provide an option for a terminal emulator window for entering command-line commands.
- Command-line commands can also be entered into **shell scripts**, which can then be run like any other programs.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

5 System Calls, Types of System Calls, System Programs	1
5.1 System Calls	1
5.2 Types of System Calls	3
5.2.1 Process Control	6
5.2.2 File Management	8
5.2.3 Device Management	8
5.2.4 Information Maintenance	8
5.2.5 Communication	8
5.2.6 Protection	9
5.3 System Programs	9
5.4 Operating-System Design and Implementation	10
5.4.1 Design Goals	10
5.4.2 Mechanisms and Policies	11
5.4.3 Implementation	11

Lecture 5

System Calls, Types of System Calls, System Programs

5.1 System Calls

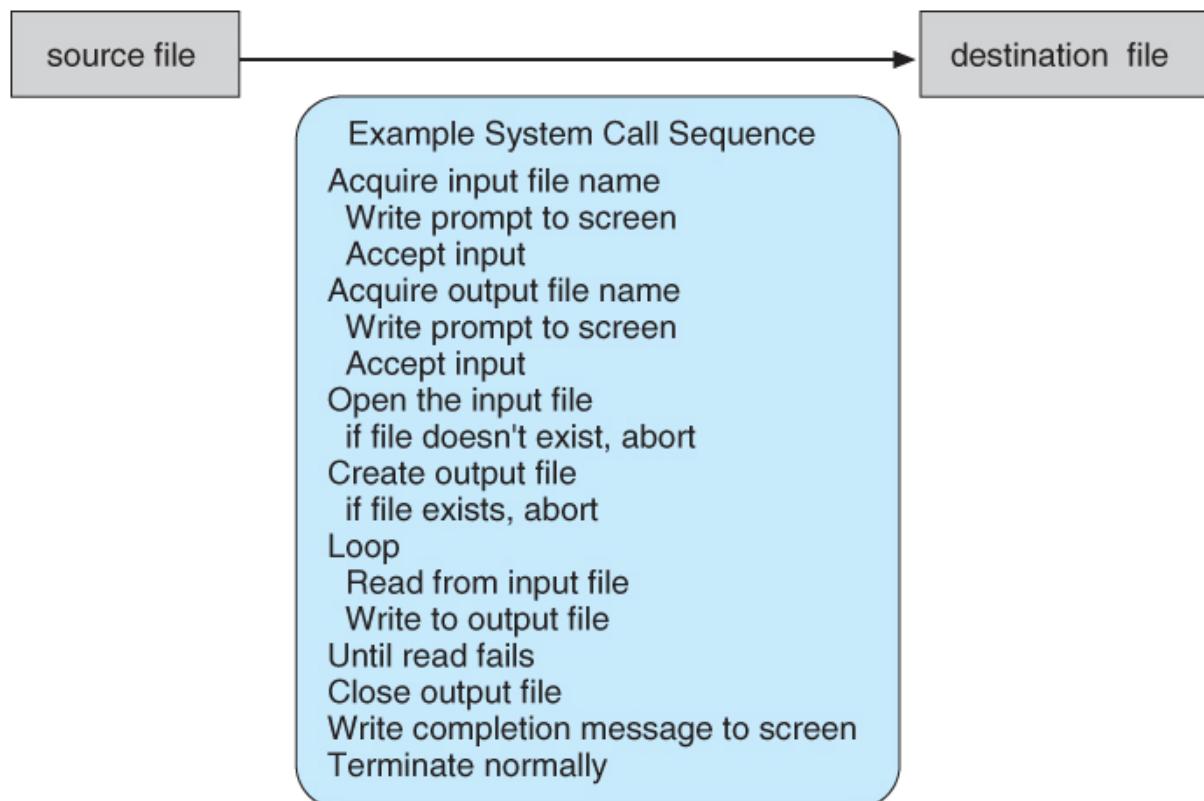


Figure 5.1: Example of how system calls are used

- System calls provide a means for user or application programs to call upon the services of the operating system.
- Generally written in C or C++, although some are written in assembly for optimal performance.

- Figure 5.1 illustrates the sequence of system calls required to copy a file.
- You can use "strace" to see more examples of the large number of system calls invoked by a single simple command. Read the man page for strace, and try some simple examples. (strace mkdir temp, strace cd temp, strace date >t.t, strace cp t.t t.2, etc.)

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the man page by invoking the command

`man read`

on the command line. A description of this API appears below:

<code>#include <unistd.h></code>		
return value	function name	parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things.) The parameters passed to `read()` are as follows:

- `int fd` - The file descriptor to be read
- `void *buf` - A buffer where the data will be read into
- `size_t count` - The maximum number of bytes to be read into the buffer.

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns -1.

Figure 5.2: Example of standard API

- Most programmers do not use the low-level system calls directly, but instead use an "Application Programming Interface", API. The 5.2 sidebar shows the `read()` call available in the API on UNIX based systems.

The use of APIs instead of direct system calls provides for greater program portability between different systems. The API then makes the appropriate system calls through the system call interface, using a table lookup to access specific numbered system calls, as shown in Figure 5.3.

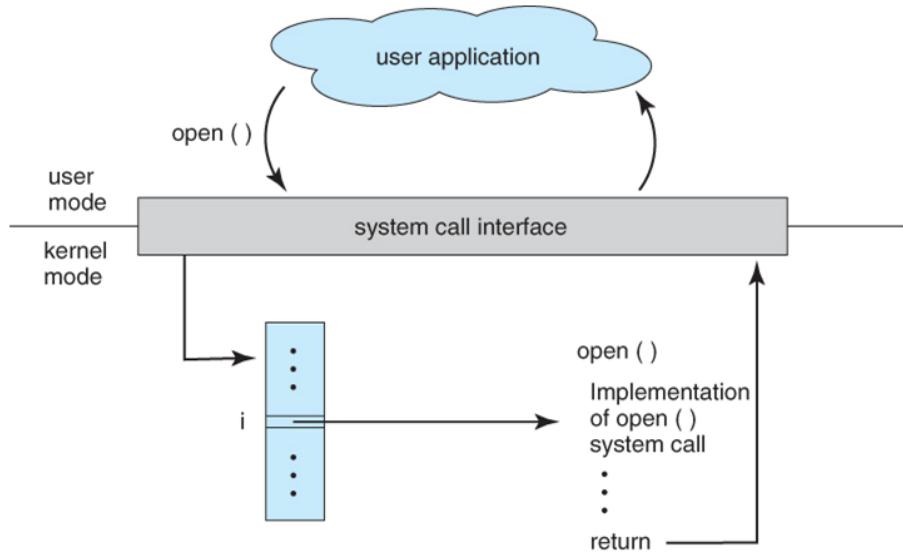


Figure 5.3: The handling of a user application invoking the `open()` system call

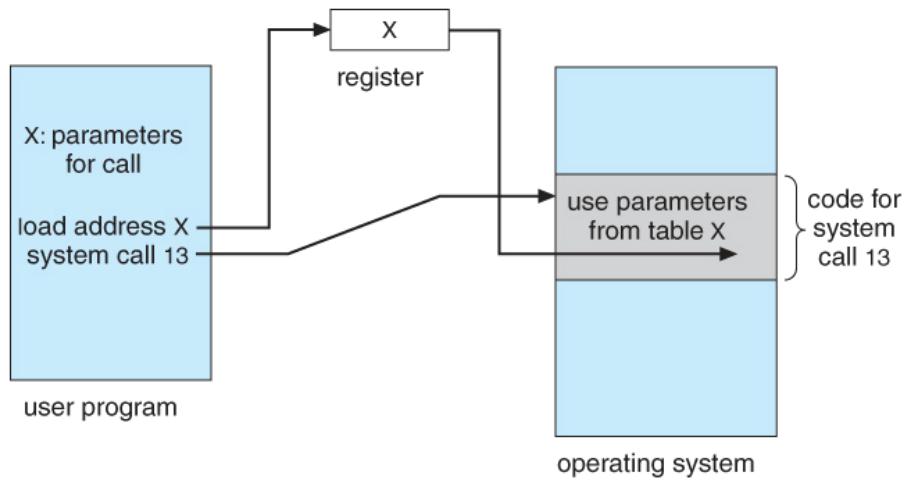


Figure 5.4: Passing of parameters as a table

- Parameters are generally passed to system calls via registers, or less commonly, by values pushed onto the stack. Large blocks of data are generally accessed indirectly, through a memory address passed in a register or on the stack, as shown in Figure 5.4.

5.2 Types of System Calls

Six major categories, as outlined in Figure 5.5 each with the mentioned six subsections:

- Process control
 - end, abort
 - load, execute
 - create process, terminate process
 - get process attributes, set process attributes
 - wait for time
 - wait event, signal event
 - allocate and free memory
- File management
 - create file, delete file
 - open, close
 - read, write, reposition
 - get file attributes, set file attributes
- Device management
 - request device, release device
 - read, write, reposition
 - get device attributes, set device attributes
 - logically attach or detach devices
- Information maintenance
 - get time or date, set time or date
 - get system data, set system data
 - get process, file, or device attributes
 - set process, file, or device attributes
- Communications
 - create, delete communication connection
 - send, receive messages
 - transfer status information
 - attach or detach remote devices

Figure 5.5: Types of system calls

Sixth type, protection, not shown in Figure 5.5 but described in Figure 5.6.):

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Figure 5.6: Examples of Windows and UNIX system calls

EXAMPLE OF STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. For example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call(s) in the operating system - in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program. This is shown below:

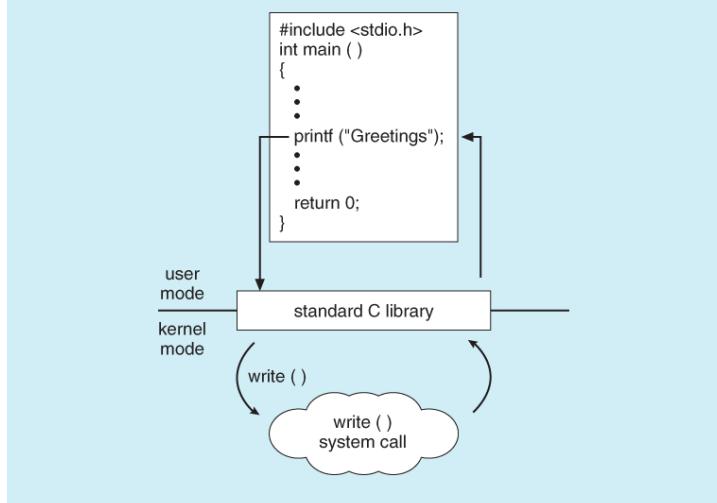


Figure 5.7: Examples of standard C library

Standard library calls may also generate system calls, as shown in 5.7

5.2.1 Process Control

- Process control system calls include end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, and allocate and free memory.
- Processes must be created, launched, monitored, paused, resumed and eventually stopped.
- When one process pauses or stops, then another must be launched or resumed.
- When processes stop abnormally it may be necessary to provide core dumps and/or other diagnostic or recovery tools.
- Compare DOS (a single-tasking system) with UNIX (a multi-tasking system).

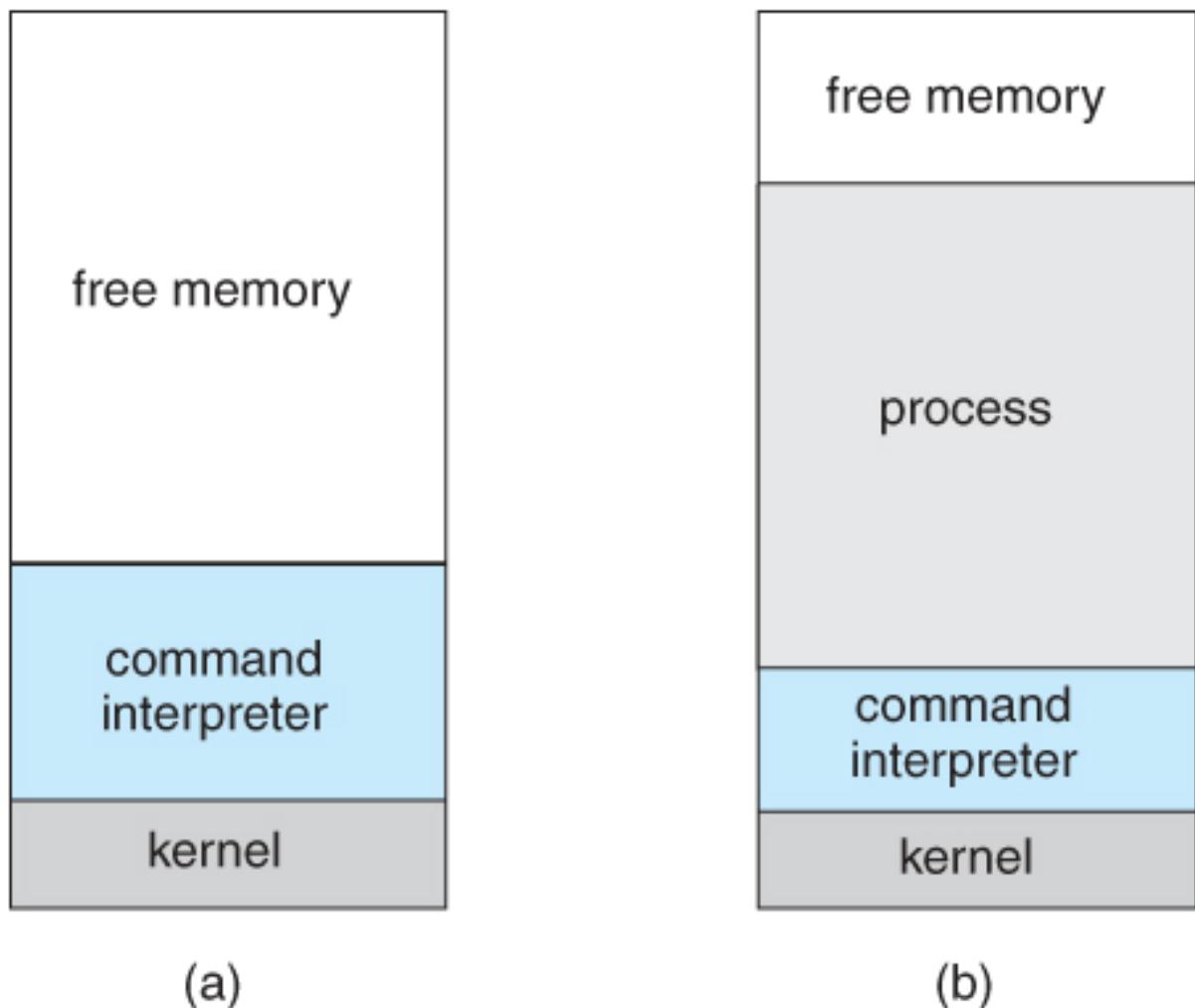


Figure 5.8: MS-DOS execution. (a) At system startup. (b) Running a program

- When a process is launched in DOS, the command interpreter first unloads as much of itself as it can to free up memory, then loads the process and transfers

control to it. The interpreter does not resume until the process has completed, as shown in Figure 5.8.

- Because UNIX(FreeBSD) is a multi-tasking system, the command interpreter remains completely resident when executing a process, as shown in Figure 5.9.

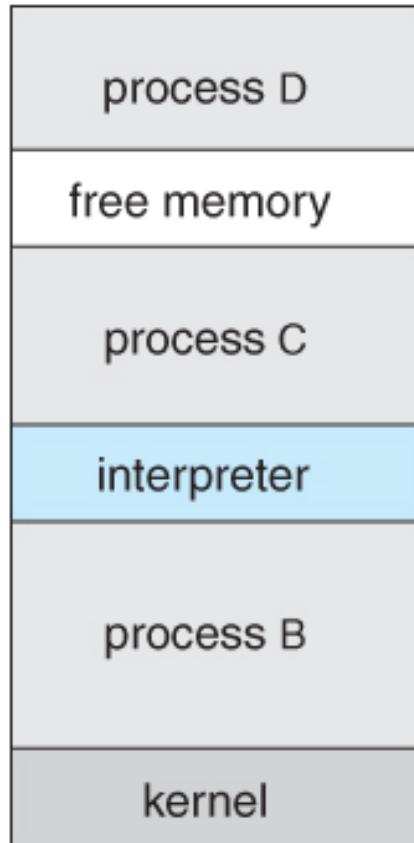


Figure 5.9: FreeBSD running multiple programs

- * The user can switch back to the command interpreter at any time, and can place the running process in the background even if it was not originally launched as a background process.
- * In order to do this, the command interpreter first executes a "fork" system call, which creates a second process which is an exact duplicate (clone) of the original command interpreter. The original process is known as the parent, and the cloned process is known as the child, with its own unique process ID and parent ID.
- * The child process then executes an "exec" system call, which replaces its code with that of the desired process.
- * The parent (command interpreter) normally waits for the child to complete before issuing a new command prompt, but in some cases it can also issue a new prompt right away, without waiting for the child process to complete. (The child is then said to be running "in the background", or "as a background process".)

5.2.2 File Management

- File management system calls include create file, delete file, open, close, read, write, reposition, get file attributes, and set file attributes.
- These operations may also be supported for directories as well as ordinary files.
- The actual directory structure may be implemented using ordinary files on the file system, or through other means.

5.2.3 Device Management

- Device management system calls include request device, release device, read, write, reposition, get/set device attributes, and logically attach or detach devices.
- Devices may be physical (e.g. disk drives), or virtual / abstract (e.g. files, partitions, and RAM disks).
- Some systems represent devices as special files in the file system, so that accessing the "file" calls upon the appropriate device drivers in the OS. See for example the /dev directory on any UNIX system.

5.2.4 Information Maintenance

- Information maintenance system calls include calls to get/set the time, date, system data, and process, file, or device attributes.
- Systems may also provide the ability to dump memory at any time, single step programs pausing execution after each instruction, and tracing the operation of programs, all of which can help to debug programs.

5.2.5 Communication

- Communication system calls create/delete communication connection, send/receive messages, transfer status information, and attach/detach remote devices.
- The **message passing model** must support calls to:
 - Identify a remote process and/or host with which to communicate.
 - Establish a connection between the two processes.
 - Open and close the connection as needed.
 - Transmit messages along the connection.
 - Wait for incoming messages, in either a blocking or non-blocking state.
 - Delete the connection when no longer needed.
- The **shared memory model** must support calls to:

- Create and access memory that is shared amongst processes (and threads).
- Provide locking mechanisms restricting simultaneous access.
- Free up shared memory and/or dynamically allocate it as needed.
- Message passing is simpler and easier, (particularly for inter-computer communications), and is generally appropriate for small amounts of data.
- Shared memory is faster, and is generally the better approach where large amounts of data are to be shared, (particularly when most processes are reading the data rather than writing it, or at least when only one or a small number of processes need to change any given data item.)

5.2.6 Protection

- Protection provides mechanisms for controlling which users / processes have access to which system resources.
- System calls allow the access mechanisms to be adjusted as needed, and for non-privileged users to be granted elevated access permissions under carefully controlled temporary circumstances.
- Once only of concern on multi-user systems, protection is now important on all systems, in the age of ubiquitous network connectivity.

5.3 System Programs

- System programs provide OS functionality through separate applications, which are not part of the kernel or command interpreters. They are also known as system utilities or system applications.
- Most systems also ship with useful applications such as calculators and simple editors, (e.g. Notepad). Some debate arises as to the border between system and non-system applications.
- System programs may be divided into these categories:
 - **File management** - programs to create, delete, copy, rename, print, list, and generally manipulate files and directories.
 - **Status information** - Utilities to check on the date, time, number of users, processes running, data logging, etc. System registries are used to store and recall configuration information for particular applications.
 - **File modification** - e.g. text editors and other tools which can change file contents.

- **Programming language support** - E.g. Compilers, linkers, debuggers, profilers, assemblers, library archive management, interpreters for common languages, and support for make.
- **Program loading and execution** - loaders, dynamic loaders, overlay loaders, etc., as well as interactive debuggers.
- **Communications** - Programs for providing connectivity between processes and users, including mail, web browsers, remote logins, file transfers, and remote command execution.
- **Background services** - System daemons are commonly started when the system is booted, and run for as long as the system is running, handling necessary services. Examples include network daemons, print servers, process schedulers, and system error monitoring services.
- Most operating systems today also come complete with a set of application programs to provide additional services, such as copying files or checking the time and date.
- Most users' views of the system is determined by their command interpreter and the application programs. Most never make system calls, even through the API, (with the exception of simple (file) I/O in user-written programs.)

5.4 Operating-System Design and Implementation

5.4.1 Design Goals

- **Requirements** define properties which the finished system must have, and are a necessary first step in designing any large complex system.
 - **User requirements** are features that users care about and understand, and are written in commonly understood vernacular. They generally do not include any implementation details, and are written similar to the product description one might find on a sales brochure or the outside of a shrink-wrapped box.
 - **System requirements** are written for the developers, and include more details about implementation specifics, performance requirements, compatibility constraints, standards compliance, etc. These requirements serve as a "contract" between the customer and the developers, (and between developers and subcontractors), and can get quite detailed.
- Requirements for operating systems can vary greatly depending on the planned scope and usage of the system. (Single user / multi-user, specialized system / general purpose, high/low security, performance needs, operating environment, etc.)

5.4.2 Mechanisms and Policies

- Policies determine what is to be done. Mechanisms determine how it is to be implemented.
- If properly separated and implemented, policy changes can be easily adjusted without re-writing the code, just by adjusting parameters or possibly loading new data / configuration files. For example the relative priority of background versus foreground tasks.

5.4.3 Implementation

- Traditionally OSes were written in assembly language. This provided direct control over hardware-related issues, but inextricably tied a particular OS to a particular HW platform.
- Recent advances in compiler efficiencies mean that most modern OSes are written in C, or more recently, C++. Critical sections of code are still written in assembly language, (or written in C, compiled to assembly, and then fine-tuned and optimized by hand from there.).
- Operating systems may be developed using emulators of the target hardware, particularly if the real hardware is unavailable (e.g. not built yet), or not a suitable platform for development, (e.g. smart phones, game consoles, or other similar devices.)



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

6 Operating System Structures	1
6.1 Operating-System Structures	1
6.1.1 Simple structure	1
6.1.2 Monolithic Structure	1
6.1.3 Layered Structure	2
6.1.4 Microkernel System Structure	4
6.2 Solaris Modular Approach	5

Lecture 6

Operating System Structures

6.1 Operating-System Structures

OS is a very large and complex system. A common approach is to partition the task into small components, or modules, rather than have one monolithic system. Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions. There are various structure discussed as follows.

6.1.1 Simple structure

Many operating systems do not have well-defined structures. Frequently, such systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system.

- MS-DOS written to provide the most functionality in the least space. It is not divided into modules.
- Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated.
- All layered have access to the base hardware. It is not well protected, structured, and not well defined.
- Application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to malicious programs, causing entire system crashes when user programs fail.
- It provides no dual mode and no hardware protection.

6.1.2 Monolithic Structure

- The UNIX OS consists of two separable parts that is systems programs and the kernel.

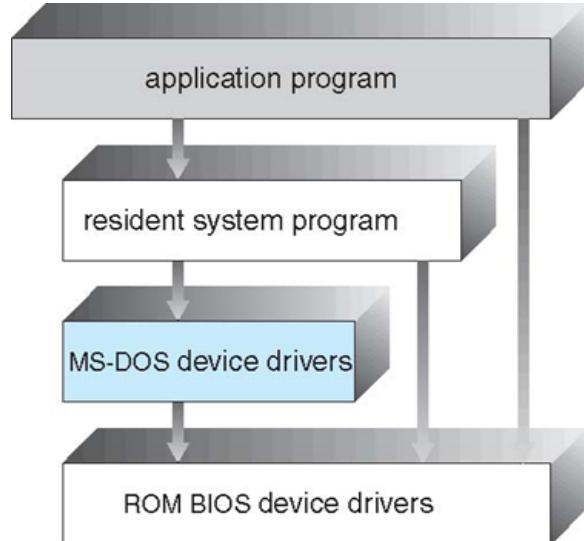


Figure 6.1: MS-DOS layer structure

- It consists of everything below the system-call interface and above the physical hardware
- It provides the file system, CPU scheduling, memory management, and other operating-system functions.
- There is very little overhead in the system call interface or in communication within the kernel
- The main disadvantages of this monolithic structure was difficult to implement and maintain.

6.1.3 Layered Structure

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers.
- A typical operating-system layer consists of data structures and a set of routines that can be invoked by higher-level layers.

Advantages

- The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers.

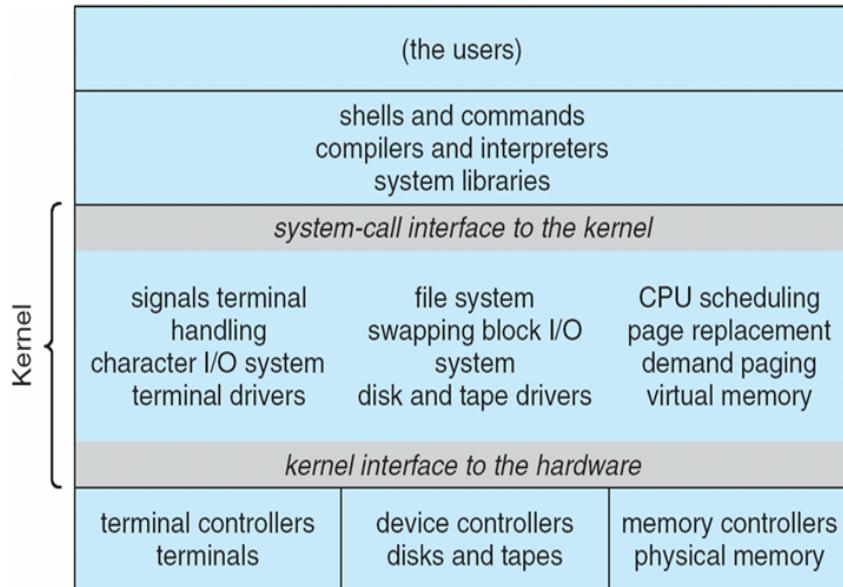


Figure 6.2: Traditional UNIX system structure

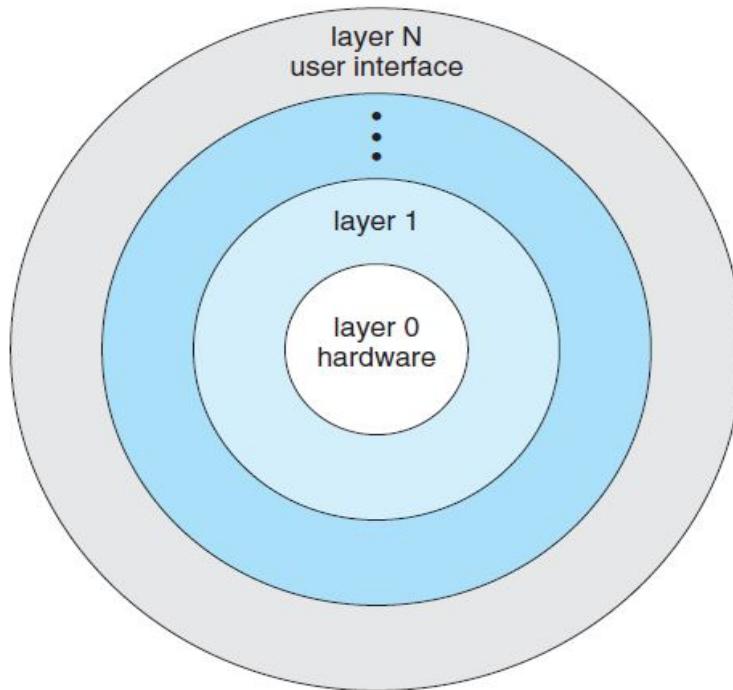


Figure 6.3: Layered Structure

- This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, it uses only the basic hardware to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on.
- If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design

and implementation of the system are simplified.

Disadvantages

- The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary. For example, the device driver for the backing store (disk space used by virtual-memory algorithms) must be at a lower level than the memory-management routines, because memory management requires the ability to use the backing store.
- It is not very efficient as it pass through all the layer

6.1.4 Microkernel System Structure

- Microkernel provides the core functionalities of the kernel such as memory management, cpu scheduling etc.
- The other functionalities like device driver, file server, virtual memory are implemented as the system program in user level
- The main function of the microkernel is to provide the communication between the client program and the services. Communication takes place between user modules using message passing. Example-Mach

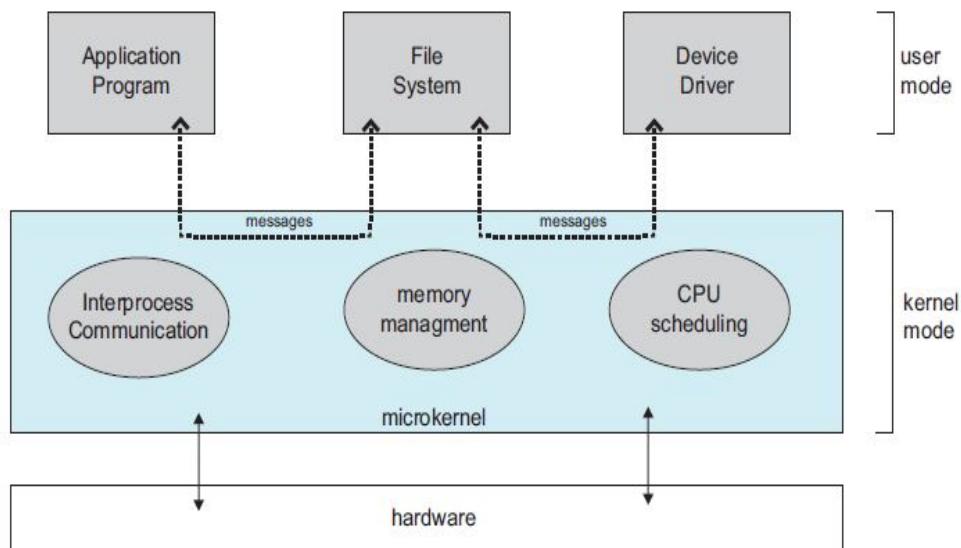


Figure 6.4: Microkernel Structure

Advantages

- Easier to extend a microkernel
- Easier to port the operating system to new architectures

- More reliable (less code is running in kernel mode)
- More secure

Disdvantages

Performance decrease due to the increase system function overhead

6.2 Solaris Modular Approach

- The kernel has a set of core components and links in additional services via modules
- It is loaded either at boot time or during runtime.
- Linking services dynamically is preferable to adding new features directly to the kernel
- Many modern operating systems implement loadable kernel modules

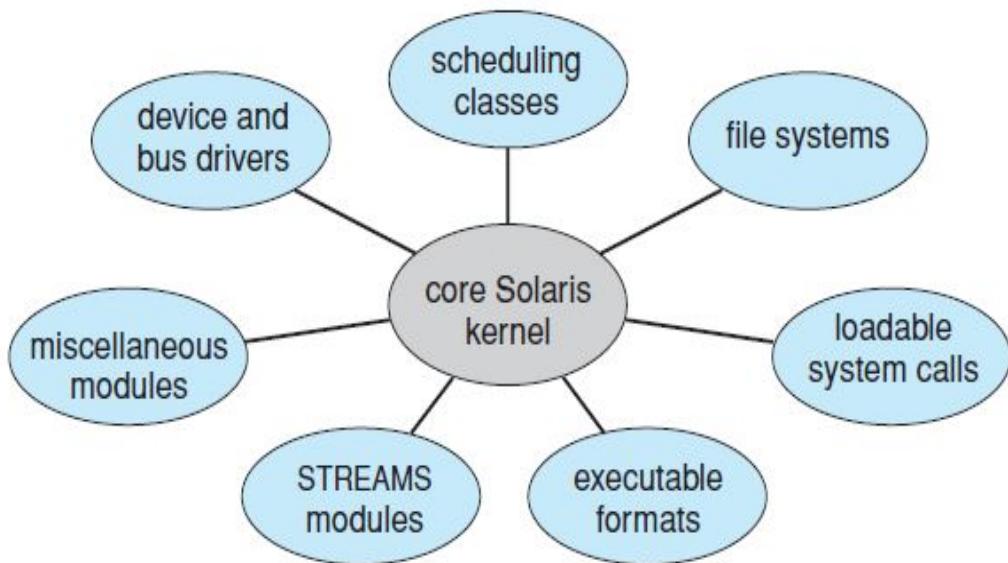


Figure 6.5: Solaris loadable modules

Advantages

- The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system, because any module can call any other module. Not pass through the all layer
- The approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

7 Process Management	1
7.1 Process Concept	1
7.2 Process State	1
7.3 Process Control Block	2

Lecture 7

Process Management

7.1 Process Concept

A process is an instance of a program running in a computer. On a single-user system, a user may be able to run several programs at one time such as a word processor, a web browser, and an e-mail package. A user can execute only one program at a time, such as on an embedded device that does not support multitasking, the operating system may need to support its own internal programmed activities, such as memory management. All these activities are called as processes. A process includes

- text: A process is more than the program code, which is sometimes known as the text section. It includes the current activity, as represented by the value of the program counter and the contents of the processor's registers.
- stack : It contains temporary data (such as function parameters, return addresses, and local variables), and a data section, which contains global variables.
- data section: It contains global variables
- heap: It is a memory that is dynamically allocated during process run time.

7.2 Process State

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

- New- The process is being created.
- Running- Instructions are being executed.
- Waiting- The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- Ready- The process is waiting to be assigned to a processor.
- Terminated- The process has finished execution.

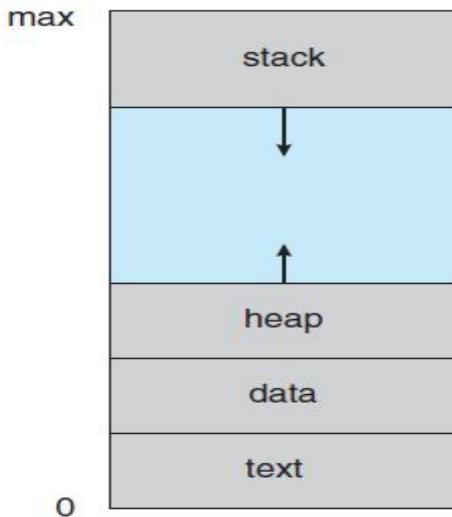


Figure 7.1: Process in memory

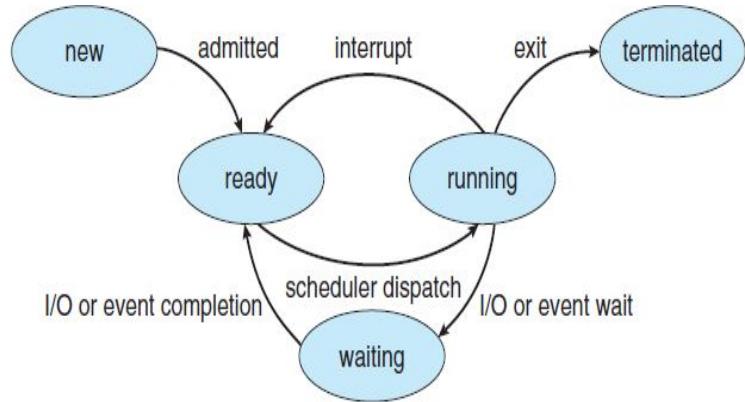


Figure 7.2: Process state diagram

7.3 Process Control Block

Each process is represented in the operating system by a process control block (PCB)—also called a task control block. It simply serves as the repository for any information that may vary from process to process. It contains many pieces of information associated with a specific process.

- Process state-The state may be new, ready, running, waiting, halted, and so on.
- Program counter. The counter indicates the address of the next instruction to be executed for this process.
- CPU registers- The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

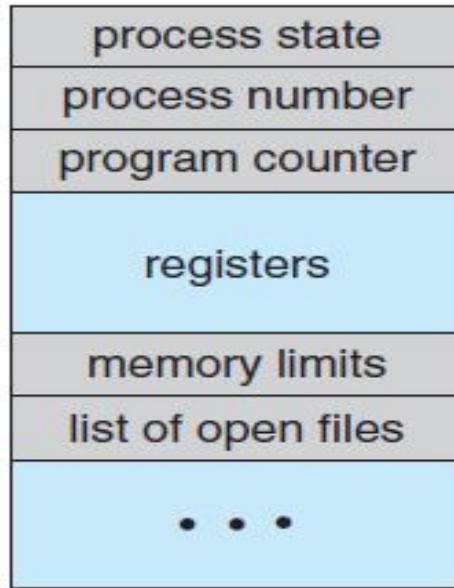


Figure 7.3: Process Control Block

- CPU-scheduling information. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- Memory-management information. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system .
- Accounting information- This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers etc.
- I/O status information- This information includes the list of I/O devices allocated to the process, a list of open files etc.

The PCB simply serves as the repository for any information that may vary from process to process.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

8 Process Management	1
8.1 Process Scheduling	1
8.1.1 Scheduler	3
8.1.2 Context Switch	4

Lecture 8

Process Management

8.1 Process Scheduling

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

Process Scheduling Queues

- Job queue – set of all processes in the system
- Ready queue – set of all processes residing in main memory, ready and waiting to execute
- Device queues – set of processes waiting for an I/O device Processes migrate among the various queues

When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. The process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue.

A common representation of process scheduling is a queueing diagram shown in Figure 8.2. A new process is initially put in the ready queue. It waits there until it is selected for execution, or dispatched. Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new child process and wait for the child's termination.

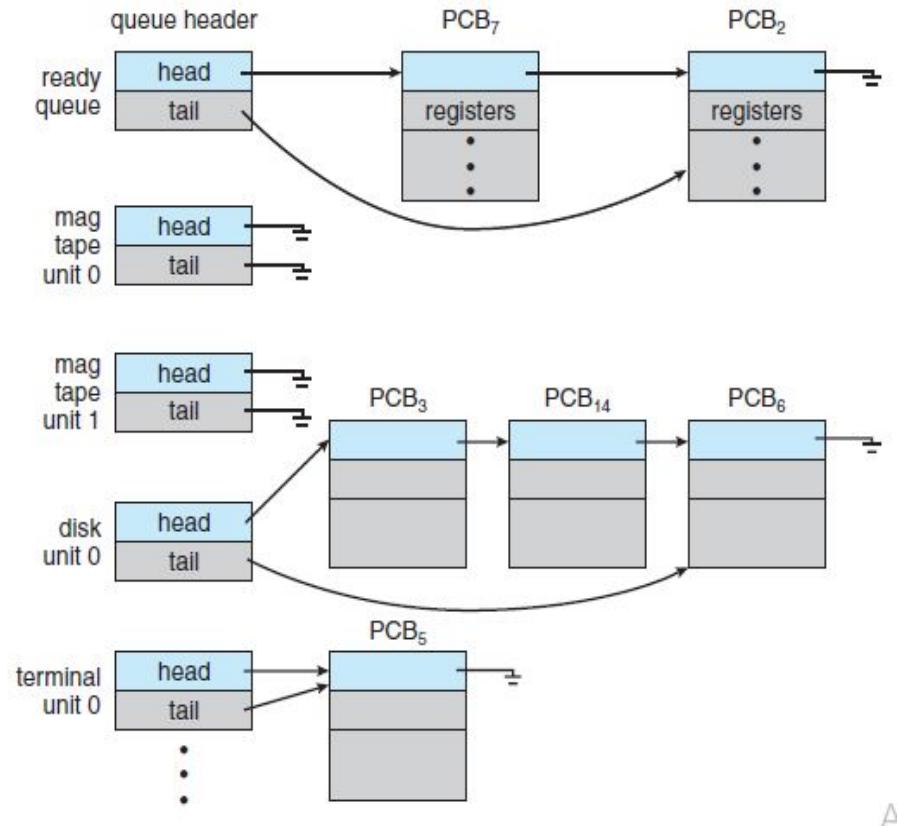


Figure 8.1: The ready queue and various I/O device queues

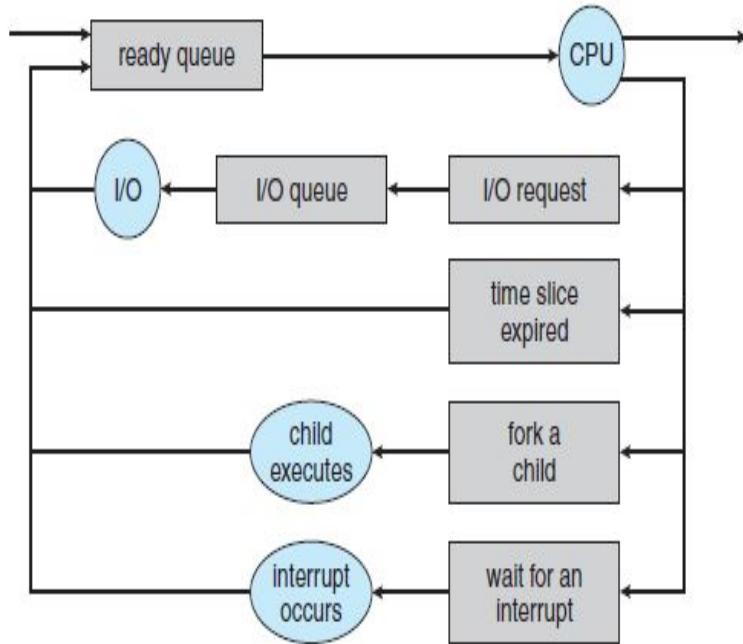


Figure 8.2: Queueing-diagram representation of process scheduling

- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the

ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

8.1.1 Scheduler

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the scheduler. There are three types of scheduler.

1. Long term scheduler
 - The long-term scheduler, or job scheduler, selects processes from this pool and loads them into memory for execution.
 - The processes can be described as either I/O bound or CPU bound.
 - An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations. In contrast, a CPU-bound process generates I/O requests infrequently, using more of its time doing computations.
 - It is important that the long-term scheduler select a good process mix of I/O-bound and CPU-bound processes. If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do.
 - If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced.
 - The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.
2. Medium term scheduler: The medium-term scheduler removes a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium-term scheduler.
3. Short term scheduler: The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute and allocates the CPU to one of them.

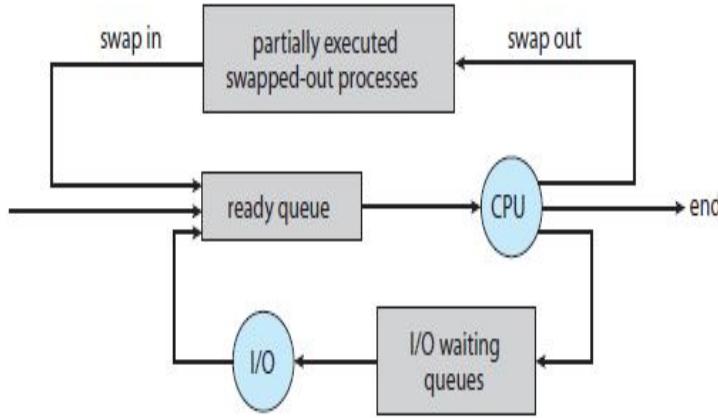


Figure 8.3: Addition of medium-term scheduling to the queueing diagram

8.1.2 Context Switch

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch.

Figure 8.4: CPU switches from one process to other process

- When an interrupt occurs, the system needs to save the current context of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.
- Context-switch time is pure overhead, because the system does no useful work while switching.
- Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions.
- Context-switch times are highly dependent on hardware support. For instance, some processors provide multiple sets of registers.
- A context switch here simply requires changing the pointer to the current register set. If there are more active processes than there are register sets, the system resorts to copying register data to and from memory.
- The more complex the operating system, the greater the amount of work that must be done during a context switch.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

9 Process Management	1
9.1 Operation on Process	1
9.1.1 Process Creation	1
9.1.2 Process creation using the fork() system call	2
9.1.3 Process termination	3
9.1.4 Problem using fork() system call	4

Lecture 9

Process Management

9.1 Operation on Process

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination. The following mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

9.1.1 Process Creation

A process may create several new process via a create process system call during the execution. The creating process is called a parent process and the new process are called the children of that process. Each of these new process may in turn create other processes, forming a tree of processes. Generally, process identified and managed via a process identifier (pid). It provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.

A process tree for the Linux operating system, showing the name of each process and its pid is shown in Figure 9.1. The init process (which always has a pid of 1) serves as the root parent process for all user processes. Once the system has booted, the init process can also create various user processes, such as a web or print server, an ssh server, and the like. The two children of init—kthreadd and sshd. The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel (in this situation, khelper and pdflush). The sshd process is responsible for managing clients that connect to the system by using ssh (which is short for secure shell). The login process is responsible for managing clients that directly log onto the system. In this example, a client has logged on and is using the bash shell, which has been assigned pid 8416. Using the bash command-line interface, this user has created the process ps as well as the emacs editor.

When a process creates a new process, two possibilities for execution exist:

1. Parent and children execute concurrently
2. Parent waits until children terminate

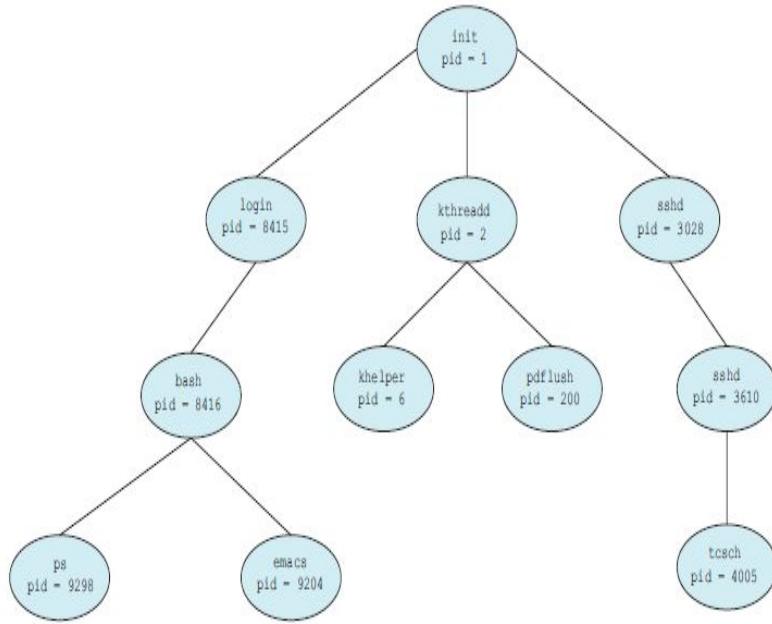


Figure 9.1: A tree of processes on a typical Linux system

Resource sharing options

1. Parent and children share all resources
2. Children share subset of parent's resources

There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

9.1.2 Process creation using the fork() system call

- A new process is created by the fork() system call.
- After a fork() system call, one of the two processes typically uses the exec() system call to replace the process's memory space with a new program.
- The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts its execution.
- The two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait() system call to move itself off the ready queue until the termination of the child.

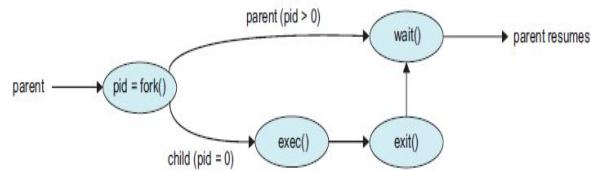


Figure 9.2: Process creation using the `fork()` system call

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

/* fork a child process */
pid = fork();

if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
}
else if (pid == 0) { /* child process */
    execvp("/bin/ls","ls",NULL);
}
else { /* parent process */
    /* parent will wait for the child to complete */
    wait(NULL);
    printf("Child Complete");
}

return 0;
}
  
```

Figure 9.3: C Program Forking Separate Process

9.1.3 Process termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the `exit()` system call. The process may

return a status value (typically an integer) to its parent process (via the wait() system call).

- All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.
- Parent may terminate the execution of children processes using the abort() system call for a variety of reasons, such as.
 - 1) Child has exceeded allocated resources
 - 2) Task assigned to child is no longer required
 - 3) The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.
- Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated. This phenomenon, referred to as **cascading termination**.
- The termination is initiated by the operating system. The parent process may wait for termination of a child process by using the wait() system call. The call returns status information and the pid of the terminated process `pid = wait(&status);`
- When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls `wait()`, because the process table contains the process's exit status. A process that has terminated, but whose parent has not yet called `wait()`, is known as a **zombie process**.
- If a parent did not invoke `wait()` and instead terminated, thereby leaving its child processes as **orphans**. Linux and UNIX address this scenario by assigning the init process as the new parent to orphan processes.

9.1.4 Problem using fork() system call

1. What is the output?

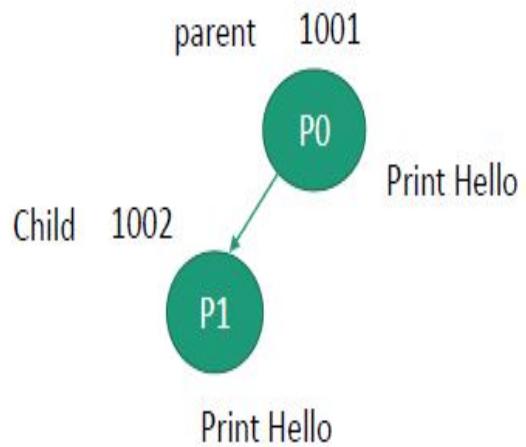
```
int main()
{
    fork();
    printf("Hello");
}
```

Output

Hello

Hello

2. How many processes are created?



```

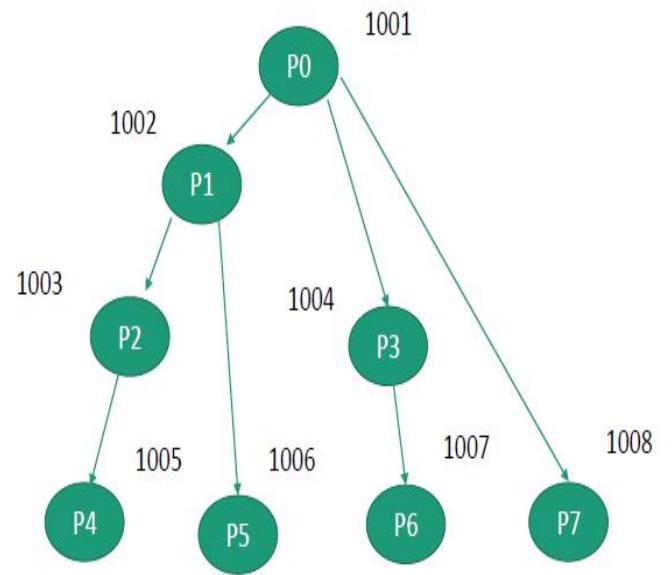
#include <stdio.h>
#include <unistd.h>

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
  
```



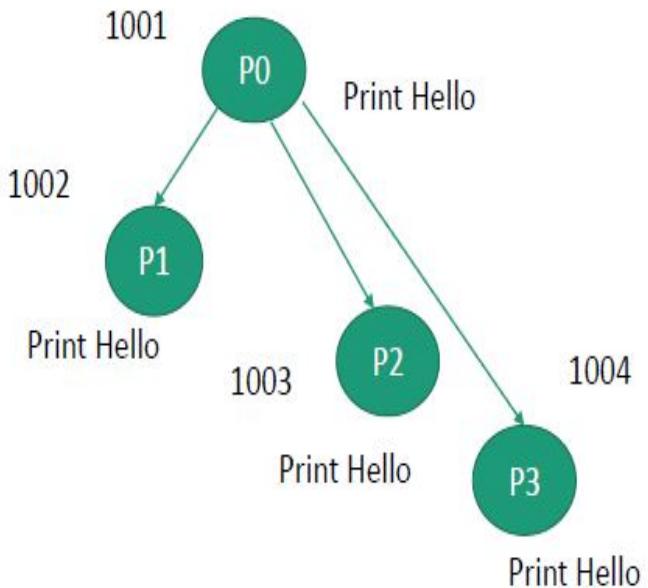
Output-8 3. How many times Hello will print?

Output-4 times

```

main()
{
If(fork() && fork())
{
    fork()
}
Printf("Hello")
}

```



4. What output will be at Line A?

Output-5

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
pid_t pid;

pid = fork();

if (pid == 0) { /* child process */
    value += 15;
    return 0;
}
else if (pid > 0) { /* parent process */
    wait(NULL);
    printf("PARENT: value = %d",value); /* LINE A */
    return 0;
}
}

```

Activate Window



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by
Mr. Rakesh Kumar
Assistant Professor

Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

10 Inter Process Communication	1
10.0.1 Co-Operating Process and Independent Process	1
10.1 Interprocess Communication	1
10.1.1 Shared-Memory Systems	2
10.1.2 Producer-Consumer Problem	2

Lecture 10

Inter Process Communication

10.0.1 Co-Operating Process and Independent Process

- Processes executing concurrently in a system may be either **independent processes or cooperating processes**.
- A process is **independent** if it cannot affect or be affected by the other processes executing in the system.
- A process is **cooperating** if it can affect or be affected by the other processes executing in the system.
- Several reasons for providing an environment for process cooperation includes **Information sharing, Computation speedup, Modularity and Convenience**.
- Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data and information.

10.1 Interprocess Communication

Two fundamental models of interprocess communication are: 1. Shared memory 2. Message passing

- In **shared-memory model**, a region of memory that is shared by cooperating processes is established. Processes then can exchange information by reading and writing data to the shared region. Shared memory allows maximum speed and convenience of communication and is faster than message passing.
- In the **message passing model**, communication takes place by means of messages exchanged between the cooperating processes. Message passing is useful for exchanging smaller amounts of data, is also easier to implement.

10.1.1 Shared-Memory Systems

- It requires communicating processes to establish a region of shared memory which typically resides in the address space of the process creating the sharedmemory segment.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.
- They can exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

10.1.2 Producer-Consumer Problem

- Here a producer process produces information that is consumed by a consumer process.
Examples: compiler-assembler, assembler-loader, client-server pardism etc.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- Two types of buffers can be used
 - Unbounded buffer which places no practical limit on the size of the buffer.
 - Bounded buffer which assumes a fixed buffer size.
- The following variables reside in a region of memory shared by the producer and consumer processes:
 - The shared buffer is implemented as a circular array with two logical pointers: in and out.
 - The variable in points to the next free position in the buffer and out points to the first full position in the buffer.
 - The buffer is empty when in == out and the buffer is full when ((in + 1) % BUFFER SIZE) == out. The code for the producer and consumer processes is shown below
 - The producer process has a local variable **nextProduced** in which the new item to be produced is stored. The consumer process has a local variable **nextConsumed** in which the item to be consumed is stored.

```

#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;

```

Figure 10.1: Communications models. (a) Message passing. (b) Shared memory.

- This scheme allows at most $BUFFER\ SIZE - 1$ items in the buffer at the same time.

<u>Producer Process</u>	<u>Consumer Process</u>
<pre> item nextProduced; while (true) { /* produce an item in nextProduced */ while (((in + 1) % BUFFER_SIZE) == out) ; /* do nothing */ buffer[in] = nextProduced; in = (in + 1) % BUFFER_SIZE; } </pre>	<pre> item nextConsumed; while (true) { while (in == out) ; // do nothing nextConsumed = buffer[out]; out = (out + 1) % BUFFER_SIZE; /* consume the item in nextConsumed */ } </pre>

Figure 10.2: Communications models. (a) Message passing. (b) Shared memory.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

11 Message-Passing Systems	1
11.1 Message Passing	1
11.1.1 Direct or Indirect Communication	1
11.1.2 Synchronous or Asynchronous Communication	3
11.1.3 Automatic or explicit buffering	3

Lecture 11

Message-Passing Systems

11.1 Message Passing

- OS provides the means for cooperating processes to communicate with each other via a message-passing facility.
- Message passing allows processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment. E.g. chat program
- A message-passing facility provides at least two operations: send(message) and receive(message).
- Messages sent by a process can be of either fixed or variable size.
- If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them.
- There are several methods for logically implementing a link and the send()/receive() operations:
 1. Direct or indirect communication
 2. Synchronous or asynchronous communication
 3. Automatic or explicit buffering

11.1.1 Direct or Indirect Communication

In **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:

— **send(P, message)**—Send a message to process P — **receive(Q, message)**—Receive a message from process Q

- A communication link in this scheme has the following properties:
 - A link is established automatically between every pair of processes that want to communicate.
 - A link is associated with exactly two processes.
 - Between each pair of processes, there exists exactly one link.
- This scheme exhibits **symmetry in addressing**; that is, both the sender process and the receiver process must name the other to communicate.
- Some employs **asymmetry in addressing** where the sender names the recipient; the recipient is not required to name the sender. Here, the send() and receive() primitives are defined as follows
 - **send(P, message)**—Send a message to process P.
 - **receive(id, message)**—Receive a message from any process; the variable id is set to the name of the process with which communication has taken place.
- Disadvantage in both symmetric and asymmetric schemes is changing the identifier of a process may necessitate examining all references to the old identifier so that they can be modified to the new identifier. With indirect communication, the messages are sent to and received from mailboxes, or ports into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. The send() and receive() primitives are defined as follows:
 - **send(A, message)**—Send a message to mailbox A.
 - **receive(A, message)**— Receive a message from mailbox A
- In this scheme, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if the processes have a shared mailbox.
- Here, a communication link has the following properties:
 - A link is established between a pair of processes only if both members of the pair have a shared mailbox.
 - A link may be associated with more than two processes.
 - Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.
- If process P1 sends a message to A, while both P2 and P3 execute a receive() from A, then which process will receive the message will depend on which of the following methods we choose:
 - Allow a link to be associated with two processes at most.
 - Allow at most one process at a time to execute a receive() operation.
 - Allow the system to select arbitrarily which process will receive the message.
- A mailbox may be owned either by a process or by the operating system.

- If the mailbox is owned by a process, then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox).
- When a process that owns a mailbox terminates, the mailbox disappears.
- A mailbox that is owned by the operating system has an existence of its own. OS then must provide mechanism that allows a process to do the following:
 - Create a new mailbox.
 - Send and receive messages through the mailbox.
 - Delete a mailbox.
- The process that creates a new mailbox is that mailbox's owner by default.
- The ownership and receiving privilege may be passed to other processes through appropriate system calls.

11.1.2 Synchronous or Asynchronous Communication

- Message passing may be either blocking or nonblocking- also known as synchronous and asynchronous.
 - **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.
 - **Nonblocking send:** The sending process sends the message and resumes operation.
 - **Blocking receive:** The receiver blocks until a message is available.
 - **Nonblocking receive:** The receiver retrieves either a valid message or a null.
- Different combinations of send() and receive() are possible.
- When both blocking send and receive are used, this is a form of **synchronization** or **synchronous communication**.
- When non-blocking send and receive are used, this is a form of **asynchronous communication**.

11.1.3 Automatic or explicit buffering

- Messages exchanged by communicating processes reside in a temporary queue which is implemented in three ways:
 - **Zero capacity:** The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it.
 - **Bounded capacity:** The queue has finite length n; thus, at most n messages can reside in it.
 - **Unbounded capacity:** The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

- The **zero-capacity** case is sometimes referred to as a message system with **no buffering**; the other cases are referred to as systems with **automatic buffering**.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSIT**

Contents

12 Threads	1
12.1 Overview	1
12.1.1 Benefits	2
12.1.2 Multicore Programming	2

Lecture 12

Threads

12.1 Overview

- 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.
- A traditional (or ***heavyweight***) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Figure 12.1 illustrates the difference between a traditional single-threaded process and a multithreaded process.
- Operating system kernels can also be multithreaded: several threads operate in the kernel, and each thread performs a specific task.
- One popular example of multithreaded application is the ***Web server***.

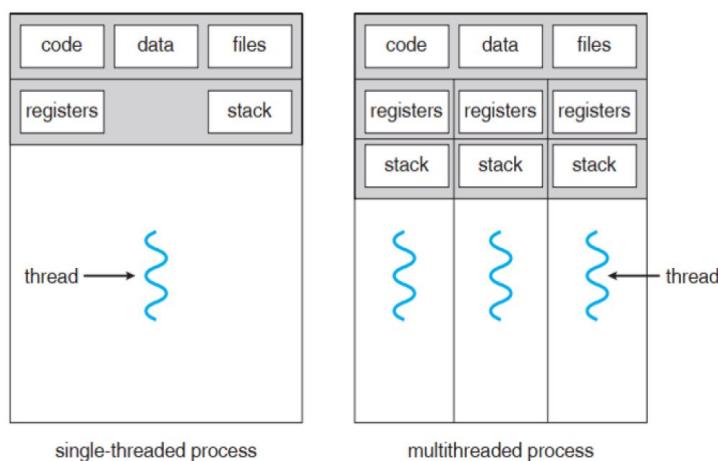


Figure 12.1: Single-threaded and Multi-threaded Processes

Web server may have to accept several requests from clients concurrently accessing it. If the Web server ran as a traditional single-threaded process, it would be able to service

only one client at a time, and a client might have to wait a very long time for its request to be serviced.

One solution is to create a separate process to service each request. However, process creation is time consuming and resource intensive. As the new process will perform the same task as the existing process, it is more efficient to use one process that contains multiple threads.

If the Web-server process is multithreaded instead, the server will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server will create a new thread to service the request and resume listening for additional requests. This is illustrated in Figure 12.2.

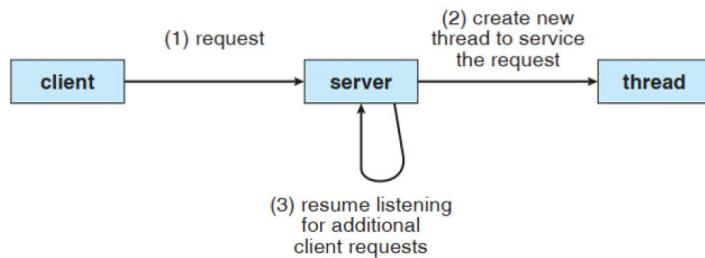


Figure 12.2: Multi-threaded Server Architecture

12.1.1 Benefits

The benefits of multithreaded programming can be

- **Responsiveness:** It may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- **Resource sharing:** Threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
- **Economy:** As threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
- **Scalability:** The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors.

12.1.2 Multicore Programming

- Multithreaded programming provides a mechanism for more efficient use of multiple cores and improved concurrency.
- On a single core system, concurrency merely means that the execution of the threads will be interleaved over time as shown in Figure 12.3.

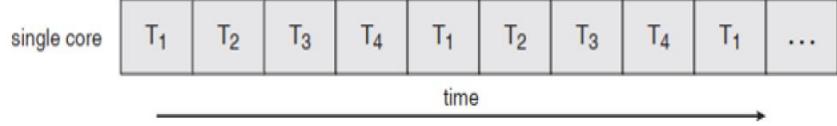


Figure 12.3: Concurrent Execution on a Single-core System

- On a system with multiple cores, however, concurrency means that threads can run in parallel as shown in Figure 12.4.

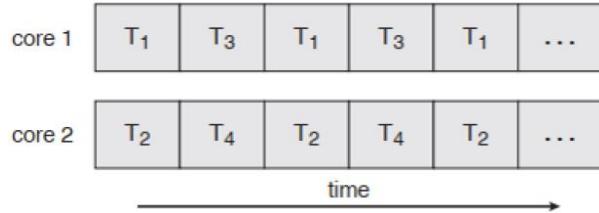


Figure 12.4: Concurrent Execution on a Single-core System

- Five areas present challenges in programming for multicore systems:
 - **Dividing activities:** Examine applications to find areas that can be divided into separate, concurrent tasks.
 - **Balance:** Programmers must also ensure that the tasks perform equal work of equal value.
 - **Data Splitting:** Just as applications divided into tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.
 - **Data Dependency:** In instances where one task depends on data from another, the execution of the tasks must be synchronized to accommodate the data dependency. **Testing and Debugging:** When a program is running in parallel on multiple cores, there are many different execution paths. Testing and debugging such concurrent programs is more difficult.
- There are two different ways to parallelize the workload of a process in a multicore system.
 - **Data Parallelism:** It focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core. **Example:** Summing the contents of an array of size N . On a single-core system, one thread would simply sum the elements $[0] \dots [N - 1]$. On a dualcore system, however, thread A, running on *core0*, could sum the elements $[0] \dots [N/2 - 1]$ while thread B, running on *core1*, could sum the elements $[N/2] \dots [N - 1]$. The two threads would be running in parallel on separate computing cores.

- **Task Parallelism:** It involves distributing not data but tasks (threads) across multiple computing cores. In contrast to above situation, an example of task parallelism might involve two threads, each performing a unique statistical operation on the array of elements. The threads again are operating in parallel on separate computing cores, but each is performing a unique operation.
- Fundamentally, data parallelism involves the distribution of data across multiple cores, and task parallelism involves the distribution of tasks across multiple cores, as shown in Figure 12.5.

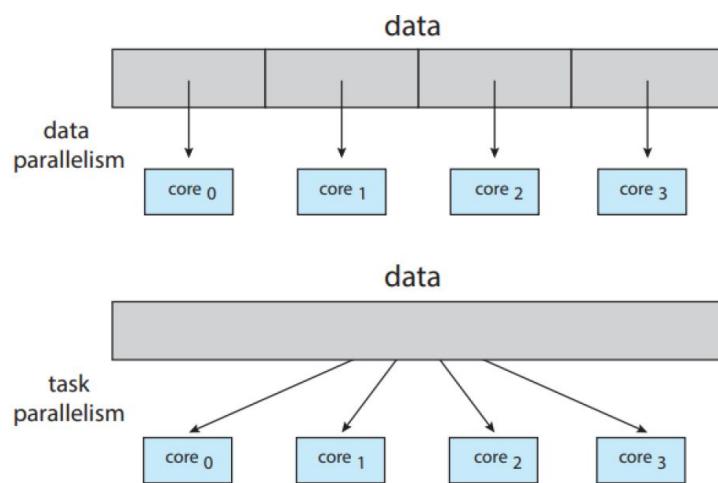


Figure 12.5: Data and Task Parallelism



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by
Mr. Rakesh Kumar
Assistant Professor

Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

13 Multithreading Models	1
13.1 Types of Thread	1
13.2 Many-to-One Model	1
13.3 One-to-One Model	2
13.4 Many-to-Many Model	3

Lecture 13

Multithreading Models

13.1 Types of Thread

Threads are implemented in following two ways

- *User Threads*

- These threads are implemented by users and the kernel is not aware of the existence of these threads.
- Kernel handles them as if they were single-threaded processes.
- User-level threads are small and much faster than kernel level threads.
- The entire process is blocked if one user-level thread performs blocking operation.

- *Kernel Threads*

- Kernel-level threads are handled by the operating system directly and the thread management is done by the kernel.
- The context information for the process as well as the process threads is all managed by the kernel. So these are slower than user-level threads.
- If a kernel-level thread is blocked, another thread of the same process can be scheduled by the kernel.

A relationship must exist between user threads and kernel threads and in below sections, we look at three common ways of establishing such a relationship.

13.2 Many-to-One Model

The benefits of multithreaded programming can be

- It maps many user-level threads to one kernel thread as shown in Figure 13.1.
- Thread management is done by the thread library in user, space, so it is efficient.

- The entire process will block if a thread makes a blocking system call.
- As only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.
- e.g. Green Threads in Solaris

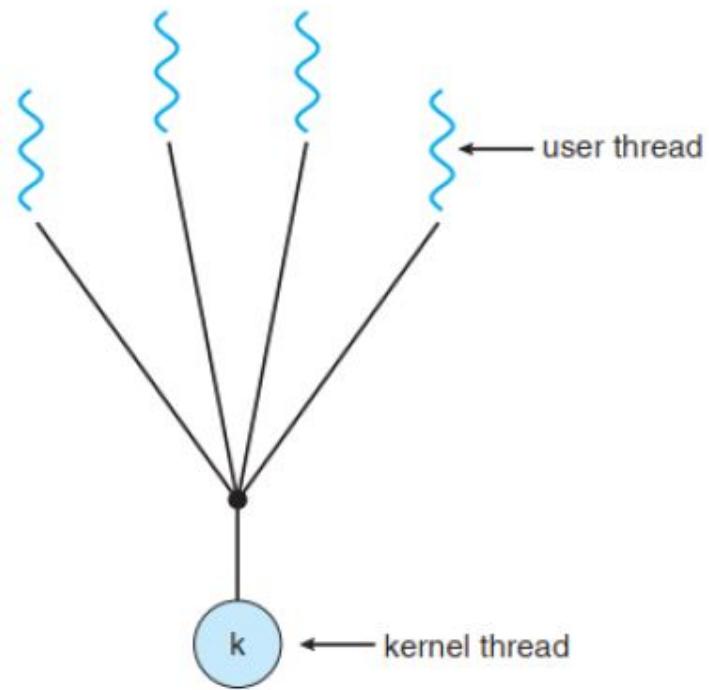


Figure 13.1: Many-to-One Model

13.3 One-to-One Model

- It maps each user thread to a kernel thread as shown in Figure 13.2
- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- It also allows multiple threads to run in parallel on multiprocessors.
- However, creating a user thread requires creating the corresponding kernel thread which can burden the performance of an application.
- Most implementations of this model restrict the number of threads supported by the system.
- e.g. Linux

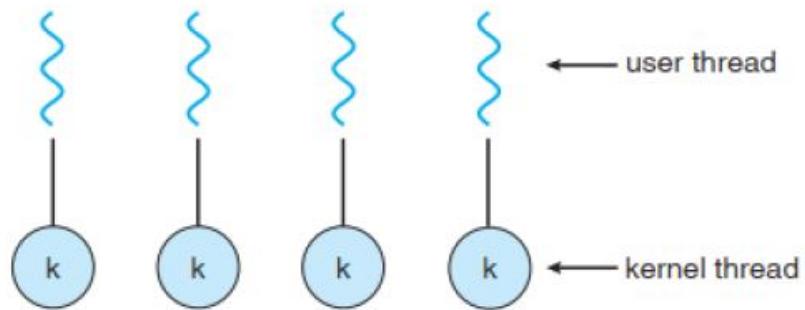


Figure 13.2: One-to-one Model

13.4 Many-to-Many Model

- It multiplexes many user-level threads to a smaller or equal number of kernel threads as shown in Figure 13.3.
- Here, developers can create as many user threads as necessary, and corresponding kernel threads can run in parallel on a multiprocessor.
- Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.
- One variation still multiplexes many user-level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread, which is referred to as a two-level model.

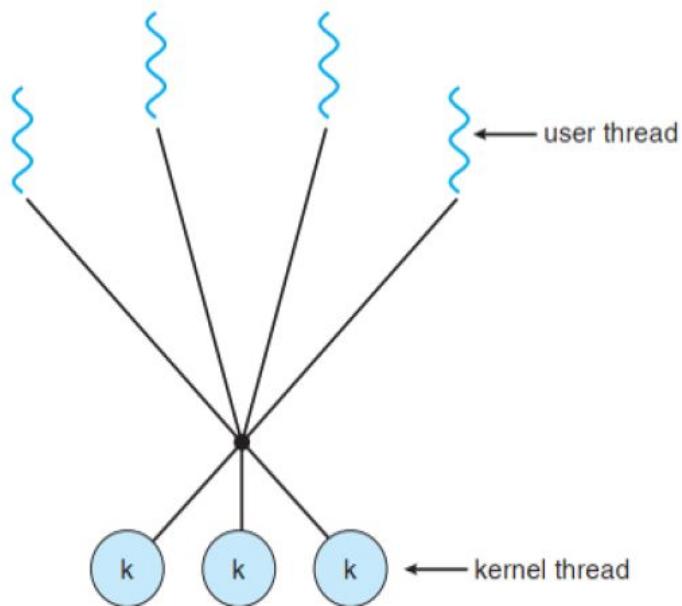


Figure 13.3: Many-to-Many Model



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

14 CPU Scheduling	1
14.1 Basic Concepts	1
14.1.1 Dispatcher	2
14.1.2 Dispatch latency	2
14.1.3 Types of CPU Scheduling	2
14.1.4 The key difference between Preemptive and Non-Preemptive Scheduling	3
14.2 Scheduling Criteria	3
14.3 FCFS(First Come First Serve) Scheduling Algorithm	5

Lecture 14

CPU Scheduling

14.1 Basic Concepts

CPU Scheduling is a process of determining which process will own CPU for execution while another process is on hold.

The main task of CPU scheduling is to make sure that whenever the CPU remains idle, the OS at least select one of the processes available in the ready queue for execution.

The selection process will be carried out by the CPU scheduler. It selects one of the processes in memory that are ready for execution.

In the uniprogrammming systems like MS DOS, when a process waits for any I/O operation to be done, the CPU remains idol. This is an overhead since it wastes the time and causes the problem of starvation. However, In Multiprogramming systems, the CPU doesn't remain idle during the waiting time of the Process and it starts executing other processes. Operating System has to define which process the CPU will be given.

In Multiprogramming systems, the Operating system schedules the processes on the CPU to have the maximum utilization of it and this procedure is called CPU scheduling. The Operating System uses various scheduling algorithm to schedule the processes.

This is a task of the short term scheduler to schedule the CPU for the number of processes present in the Job Pool. Whenever the running process requests some IO operation then the short term scheduler saves the current context of the process (also called PCB) and changes its state from running to waiting.

During the time, process is in waiting state; the Short term scheduler picks another process from the ready queue and assigns the CPU to this process. This procedure is called context switching.

Why do we need Scheduling?

In Multiprogramming, if the long term scheduler picks more I/O bound processes then most of the time, the CPU remains idle. The task of Operating system is to optimize the utilization of resources.

If most of the running processes change their state from running to waiting then there may always be a possibility of deadlock in the system. Hence to reduce this overhead, the OS needs to schedule the jobs to get the optimal utilization of CPU and to avoid the possibility to deadlock.

14.1.1 Dispatcher

Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:

1. switching context
2. switching to user mode
3. jumping to the proper location in the user program to restart that program

14.1.2 Dispatch latency

The time it takes for the dispatcher to stop one process and start another running

14.1.3 Types of CPU Scheduling

1. Preemptive Scheduling

In Preemptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.

2. Non-Preemptive Scheduling

In this type of scheduling method, the CPU has been allocated to a specific process. The process that keeps the CPU busy will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. That's because it doesn't need special hardware (for example, a timer) like preemptive scheduling

When scheduling is Preemptive or Non-Preemptive?

To determine if scheduling is preemptive or non-preemptive, consider these four parameters:

1. A process switches from the running to the waiting state.

2. Specific process switches from the running state to the ready state.
3. Specific process switches from the waiting state to the ready state.
4. Process finished its execution and terminated.

Only conditions 1 and 4 apply, the scheduling is called **non-preemptive**. All other scheduling are **preemptive**

14.1.4 The key difference between Preemptive and Non-Preemptive Scheduling

1. In preemptive scheduling the CPU is allocated to the processes for the limited time whereas in Non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to waiting state.
2. The executing process in preemptive scheduling is interrupted in the middle of execution when higher priority one comes whereas, the executing process in non-preemptive scheduling is not interrupted in the middle of execution and wait till its execution.
3. In Preemptive Scheduling, there is the overhead of switching the process from ready state to running state, vice-versa, and maintaining the ready queue. Whereas in case of non-preemptive scheduling has no overhead of switching the process from running state to ready state.
4. In preemptive scheduling, if a high priority process frequently arrives in the ready queue then the process with low priority has to wait for a long, and it may have to starve. On the other hands, in the non-preemptive scheduling, if CPU is allocated to the process having larger burst time then the processes with small burst time may have to starve.
5. Preemptive scheduling attain flexible by allowing the critical processes to access CPU as they arrive into the ready queue, no matter what process is executing currently. Non-preemptive scheduling is called rigid as even if a critical process enters the ready queue the process running CPU is not disturbed.
6. The Preemptive Scheduling has to maintain the integrity of shared data that's why it is cost associative as it which is not the case with Non-preemptive Scheduling.

14.2 Scheduling Criteria

Many criteria have been suggested for comparing CPU scheduling algorithms.

The criteria include the following:

The main objective of any CPU scheduling algorithm is to keep the CPU as busy as possible. Theoretically CPU utilisation can range from 0 to 100 but in a real time system it varies from 40 to 90 percent depending on the load upon the system.

1. Throughput

- A measure of the work done by CPU is the number of processes being executed and completed per unit time. This is called throughput. The throughput may vary depending upon the length or duration of processes.
- Example: let's say, the process P1 takes 3 seconds for execution, P2 takes 5 seconds, and P3 takes 10 seconds. So, throughput, in this case, the throughput will be $(3+5+10)/3 = 18/3 = 6$ seconds.

2. Turnaround time(TAT)

- For a particular process, an important criteria is how long it takes to execute that process. The time elapsed from the time of submission of a process to the time of completion is known as turnaround time. Turn-around time is the sum of times spent waiting to get into memory, waiting in ready queue, executing in CPU and waiting for I/O.

$$\text{Turnaround time} = \text{Burst time} + \text{Waiting time} \quad \text{OR}$$

$$\text{Turnaround time} = \text{Exit time} - \text{Arrival time}$$

- **Arrival time(AT)** is the time when a process enters into the ready state and is ready for its execution
- **Exit time** is the time when a process completes its execution and exit from the system
- **Burst time(BT)** Every process in a computer system requires some amount of time for its execution such as CPU time and IO time. Generally by ignoring the I/O time, the burst time is the total time taken by the process for its execution on the CPU.

3. Waiting time(WT)

- A scheduling algorithm does not affect the time required to complete the process once it starts execution. It only affects the waiting time of a process i.e. time spent by a process waiting in the ready queue.
- **Waiting time** = Turnaround time - Burst time

4. Response time(RT)

- In an interactive system turn-around time is not the best criteria. A process may produce some output fairly early and continue computing new results while previous results are being output to user. Thus another criteria is the time taken from submission of process of request until the first response is produced. This measure is called response time.
- **Response time** = Time at which the process gets the CPU for the first time - Arrival time

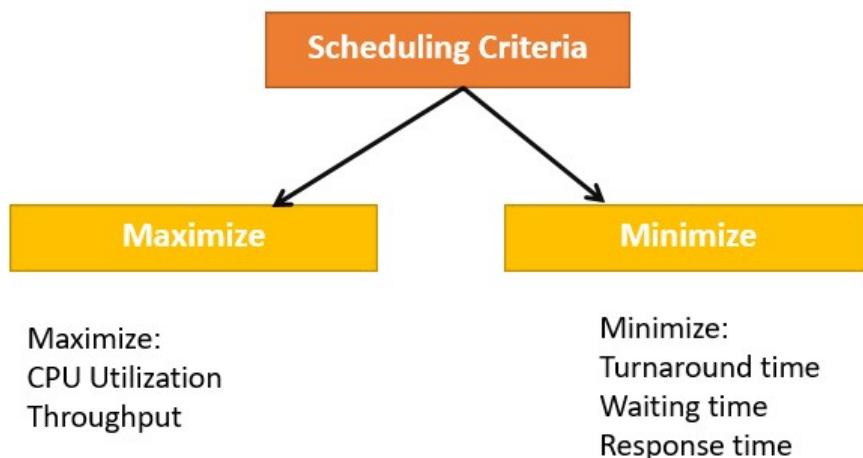


Figure 14.1: CPU scheduling criteria

14.3 FCFS(First Come First Serve) Scheduling Algorithm

In the "First come first serve" scheduling algorithm, as the name suggests, the process which arrives first, gets executed first, or we can say that the process which requests the CPU first, gets the CPU allocated first.

First Come First Serve, is just like FIFO(First in First out) Queue data structure, where the data element which is added to the queue first, is the one who leaves the queue first.

This is used in Batch Systems.

It's easy to understand and implement programmatically, using a Queue data structure, where a new process enters through the tail of the queue, and the scheduler selects process from the head of the queue. A perfect real life example of FCFS scheduling is buying tickets at ticket counter

It is a non-preemptive, pre-emptive scheduling algorithm

AWT or Average waiting time is the average of the waiting times of the processes in the queue, waiting for the scheduler to pick them for execution.

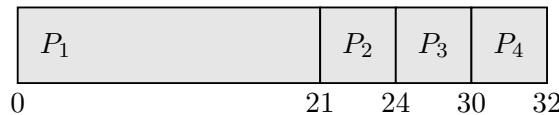
[Lower the Average Waiting Time, better the scheduling algorithm]

Example

Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, and given Burst Time, let's find the average waiting time using the FCFS scheduling algorithm.

Process	Burst time
P1	21
P2	3
P3	6
P4	2

Gantt chart for FCFS



Calculation of Waiting Time, Turn around time, and Response time for each process

The average waiting time will be 18.75 ms. For the above given processes, first P1 will be provided with the CPU resources,

Hence, waiting time for P1 will be 0

P1 requires 21 ms for completion, hence waiting time for P2 will be 21 ms

Similarly, waiting time for process P3 will be execution time of P1 + execution time for P2, which will be $(21 + 3)$ ms = 24 ms.

For process P4 it will be the sum of execution times of P1, P2 and P3.

The GANTT chart above perfectly represents the waiting time for each process.

Process	BT	WT	TAT	RT
P1	21	0	21	0
P2	3	21	24	21
P3	6	24	30	24
P4	2	30	32	30



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

15 CPU Scheduling	1
15.1 FCFS with different Arrival time	1
15.1.1 Advantages of FCFS	2
15.1.2 Disadvantages of FCFS	2
15.2 Shortest Job Next (SJN) Or Shortest Job first (SJF)	2
15.3 Advantages of SJF	5
15.4 Disadvantages of SJF	5

Lecture 15

CPU Scheduling

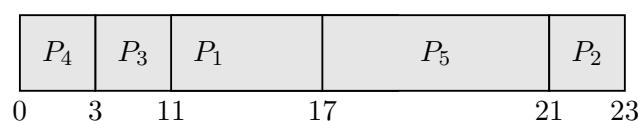
15.1 FCFS with different Arrival time

Here is an example of five processes arriving at different times. Each process has a different burst time.

Example

Process	Burst time	Arrival time
P1	6	2
P2	3	5
P3	8	1
P4	3	0
P5	4	4

Gantt chart for FCFS



Calculation of waiting time, turn around time, and Response time of each process

Process	AT	BT	WT	TAT	RT
P1	2	6	9	15	9
P2	5	3	16	19	16
P3	1	8	2	10	2
P4	0	3	0	3	0
P5	4	4	13	17	13

$$\text{Average Waiting Time} = 40/5 = 8$$

Average Turn around Time = $54/5 = 10.8$

15.1.1 Advantages of FCFS

1. The simplest form of a CPU scheduling algorithm
2. Easy to program
3. First come first served

15.1.2 Disadvantages of FCFS

1. It is a Non-Preemptive CPU scheduling algorithm, so after the process has been allocated to the CPU, it will never release the CPU until it finishes executing.
2. The Average Waiting Time is high.
3. Short processes that are at the back of the queue have to wait for the long process at the front to finish.
4. Not an ideal technique for time-sharing systems
5. Because of its simplicity, FCFS is not very efficient
6. (**What is Convoy Effect?**) Convoy Effect is a situation where many processes, who need to use a resource for short time are blocked by one process holding that resource for a long time. This essentially leads to poor utilization of resources and hence poor performance

15.2 Shortest Job Next (SJN) Or Shortest Job first (SJF)

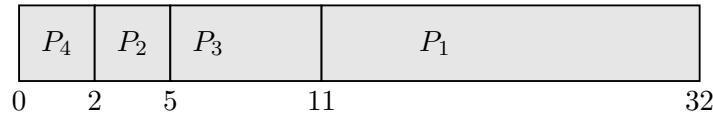
1. It is an algorithm in which the process having the smallest execution time is chosen for the next execution.
2. This is also known as shortest job first, or SJF
3. Two schemes:
 - i. Non-preemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
 - ii. Preemptive(**SRJF-shortest remaining job first**) – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF)

4. Best approach to minimize waiting time.
5. Easy to implement in Batch systems where required CPU time is known in advance.
6. Impossible to implement in interactive systems where required CPU time is not known.

Example

Processes	Execution time
P1	21
P2	3
P3	6
P4	2

Gantt chart for SJF



Calculation of waiting time, turn around time, and Response time of each process

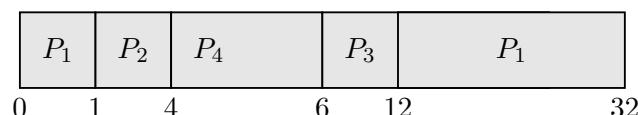
Process	BT	WT	TAT	RT
P1	21	11	32	11
P2	3	2	5	2
P3	6	5	11	5
P4	2	0	2	0

$$\text{Average Wait Time: } (0 + 4 + 12 + 5) / 4 = 21 / 4 = 5.25$$

Example of SRTF

Processes	arrival time	Execution time
P1	0	21
P2	1	3
P3	2	6
P4	3	2

Gantt chart for SRTF



Calculation of waiting time, turn around time, and Response time of each process

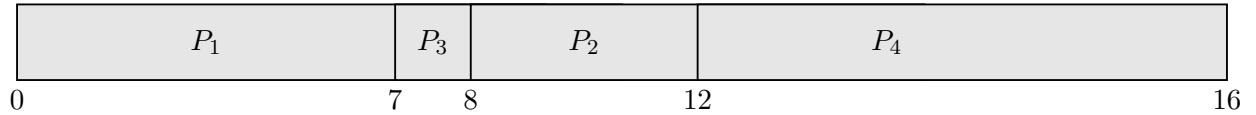
Process	AT	BT	WT	TAT	RT
P1	0	21	11	32	0
P2	1	3	0	3	0
P3	2	6	4	10	4
P4	3	2	1	3	1

Example

Calculate average waiting time and turnaround time using Non-preemptive and preemptive SJF of below list of processes with their arrival time and CPU burst time.

Processes	arrival time	Execution time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

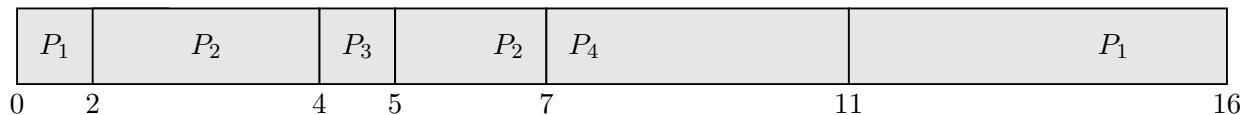
Gantt chart for Nonpreemptive SJF



Calculation of waiting time, turn around time, and Response time of each process

Process	AT	BT	WT	TAT	RT
P1	0.0	7	0	7	0
P2	2.0	4	6	10	6
P3	4.0	1	3	4	3
P4	5.0	4	7	11	7

Gantt chart for Preemptive SJF



Calculation of waiting time, turn around time, and Response time of each process

Process	AT	BT	WT	TAT	RT
P1	0.0	7	9	16	0
P2	2.0	4	1	5	0
P3	4.0	1	0	1	0
P4	5.0	4	2	6	2

15.3 Advantages of SJF

1. SJF is frequently used for long term scheduling.
2. It reduces the average waiting time over FIFO (First in First Out) algorithm.
3. SJF method gives the lowest average waiting time for a specific set of processes.
4. It is appropriate for the jobs running in batch, where run times are known in advance.
5. Probably optimal with regard to average turnaround time.

15.4 Disadvantages of SJF

1. Job completion time must be known earlier, but it is hard to predict.
2. It is often used in a batch system for long term scheduling.
3. SJF can't be implemented for CPU scheduling for the short term. It is because there is no specific method to predict the length of the upcoming CPU burst.
4. This algorithm may cause very long turnaround times or starvation.
5. It is hard to know the length of the upcoming CPU request.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSIT**

Contents

16 CPU Scheduling	1
16.1 Priority Scheduling	1
16.1.1 Problem with Priority Scheduling Algorithm	2
16.1.2 Using Aging Technique with Priority Scheduling	3
16.2 Round robin Scheduling	3
16.2.1 Advantage of Round-robin Scheduling	4
16.2.2 Disadvantages of Round-robin Scheduling	4

Lecture 16

CPU Scheduling

16.1 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).

SJF is a priority scheduling where priority is the predicted next CPU burst time.

Priority scheduling can be of two types:

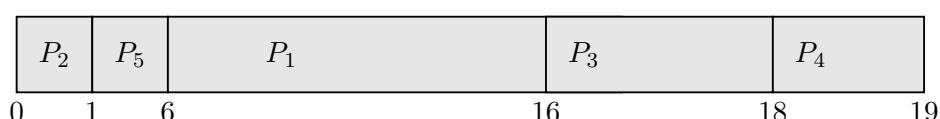
Preemptive Priority Scheduling: If the new process arrived at the ready queue has a higher priority than the currently running process, the CPU is preempted, which means the processing of the current process is stopped and the incoming new process with higher priority gets the CPU for its execution.

Non-Preemptive Priority Scheduling: In case of non-preemptive priority scheduling algorithm if a new process arrives with a higher priority than the current running process, the incoming process is put at the head of the ready queue, which means after the execution of the current process it will be processed.

Example

Processes	Burst time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

Gantt chart for Priority Scheduling



Calculation of waiting time, turn around time, and Response time of each process

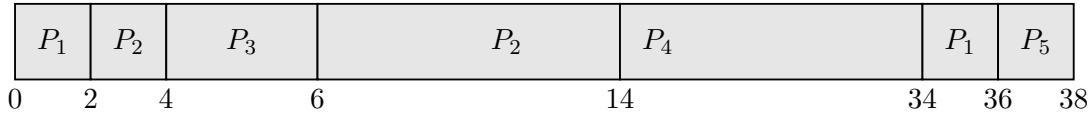
Process	Burst time	Priority	WT	TAT	RT
P1	10	3	6	16	6
P2	1	1	0	1	0
P3	2	3	16	18	16
P4	1	4	18	19	18
P5	5	2	1	6	1

Average waiting time = $(0 + 1 + 6 + 16 + 18)/5 = 8.2$

Example

Processes	Burst time	Priority	arrival time
P1	4	4	0
P2	10	2	2
P3	2	1	4
P4	20	3	6
P5	2	5	8

Gantt chart for Preemptive Priority Scheduling



Calculation of waiting time, turn around time, and Response time of each process

Process	AT	Burst time	Priority	WT	TAT	RT
P1	0	4	4	32	36	0
P2	2	10	2	2	12	0
P3	4	2	1	0	2	0
P4	6	20	3	8	28	8
P5	8	2	5	28	30	28

Average waiting time = $32 + 2 + 0 + 8 + 28/5 = 14$

16.1.1 Problem with Priority Scheduling Algorithm

1. In priority scheduling algorithm, the chances of indefinite blocking or starvation
2. A process is considered blocked when it is ready to run but has to wait for the CPU as some other process is running currently.

3. But in case of priority scheduling if new higher priority processes keeps coming in the ready queue then the processes waiting in the ready queue with lower priority may have to wait for long durations before getting the CPU for execution.
- example::In 1973, when the IBM 7904 machine was shut down at MIT, a low-priority process was found which was submitted in 1967 and had not yet been run.

16.1.2 Using Aging Technique with Priority Scheduling

To prevent starvation of any process, we can use the concept of aging where we keep on increasing the priority of low-priority process based on the its waiting time.

16.2 Round robin Scheduling

Round Robin is the preemptive process scheduling algorithm.

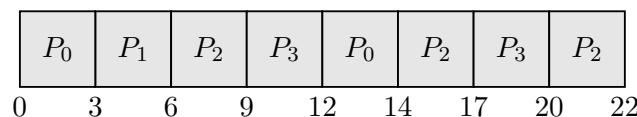
Each process is provided a fix time to execute, it is called a **quantum**.

Once a process is executed for a given time period, it is preempted and other process executes for a given time period.

Context switching is used to save states of preempted processes.

Processes	arrival time	Burst time
P0	0	5
P1	1	3
P2	2	8
P3	3	6

Gantt chart for Round robin Scheduling



Calculation of waiting time, turn around time, and Response time of each process

Process	AT	BT	WT	TAT	RT
P0	0	5	9	14	0
P1	1	3	2	5	2
P2	2	8	12	20	4
P3	3	6	11	17	6

Average Wait Time: $(9 + 2 + 12 + 11)/4 = 8.5$

16.2.1 Advantage of Round-robin Scheduling

1. It doesn't face the issues of starvation or convoy effect.
2. All the jobs get a fair allocation of CPU.
3. It deals with all process without any priority
4. It gives the best performance in terms of average response time.

16.2.2 Disadvantages of Round-robin Scheduling

1. If slicing time of OS is low, the processor output will be reduced.
2. This method spends more time on context switching
3. Lower time quantum results in higher the context switching overhead in the system.
4. Finding a correct time quantum is a quite difficult task in this system.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

17 CPU Scheduling	1
17.1 Multiple-Level Queues Scheduling	1
17.1.1 Advantages	2
17.1.2 Disadvantages	2
17.2 Multiple-Level feedback Queues Scheduling(MLFQ)	2
17.2.1 Advantages	5
17.2.2 Disadvantages	5

Lecture 17

CPU Scheduling

17.1 Multiple-Level Queues Scheduling

Multiple-level queues are not an independent scheduling algorithm.

They make use of other existing algorithms to group and schedule jobs with common characteristics.

Multiple queues are maintained for processes with common characteristics.

Each queue can have its own scheduling algorithms. Priorities are assigned to each queue.

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

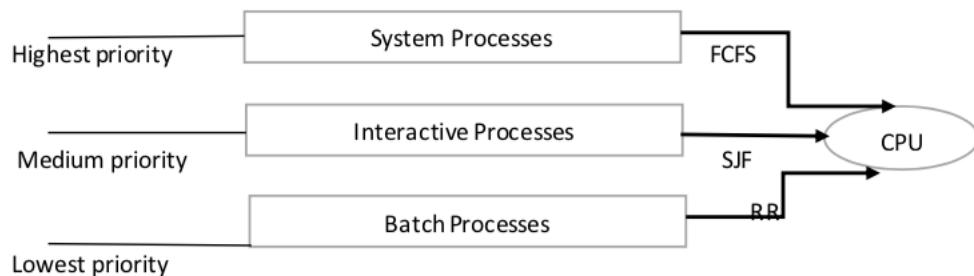


Figure 17.1: Multi-level Queue Scheduling

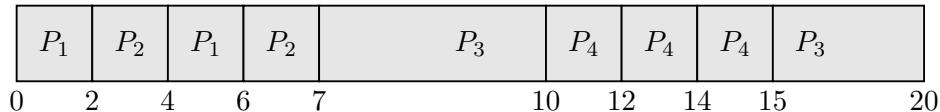
Example

Consider below table of 4 processes under Multi level queue scheduling; Queue number denotes the queue of the process.

Processes	Arrival time	Burst time	Q_number
P1	0	4	1
P2	0	3	1
P3	0	8	2
P4	10	5	1

priority of queue1 is greater than queue2. Queue1 uses Round robin (Time Quantum=2) and queue 2 uses FCFS.

Gantt chart for Multi-level queue Scheduling



Calculation of waiting time, turn around time, and Response time of each process

Process	AT	BT	WT	TAT	RT
P1	0	4	2	6	0
P2	0	3	4	7	2
P3	0	8	12	20	7
P4	10	5	0	5	0

17.1.1 Advantages

The major advantage of this algorithm is that we can use various algorithms such as FCFS, SJF, Ljf, etc. At the same time in different queues.

17.1.2 Disadvantages

The lowest level processes suffer from starvation problem.

17.2 Multiple-Level feedback Queues Scheduling(MLFQ)

In a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system. Processes do not move between queues. This setup has the advantage of low scheduling overhead, but the disadvantage of being inflexible.

Multilevel feedback queue scheduling, however, allows a process to move between queues. The idea is to separate processes with different CPU-burst characteristics. If a process uses too much CPU time, it will be moved to a lower-priority queue. Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

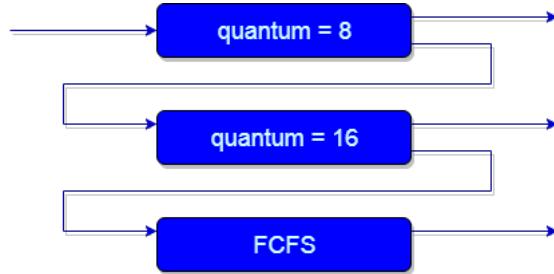


Figure 17.2: Example of Multilevel feedback scheduling

Let us suppose that in ?? queue 1 and 2 follow round robin with time quantum 4 and 8 respectively and queue 3 follow FCFS. Then in MLFQ,

1. When a process starts executing then it first enters queue 1.
2. In queue 1 process executes for 4 unit and if it completes in this 4 unit or it gives CPU for I/O operation in this 4 unit than the priority of this process does not change and if it again comes in the ready queue than it again starts its execution in Queue 1.
3. If a process in queue 1 does not complete in 4 unit then its priority gets reduced and it shifted to queue 2. Above points 2 and 3 are also true for queue 2 processes but the time quantum is 8 unit. In a general case if a process does not complete in a time quantum than it is shifted to the lower priority queue.
4. In the last queue, processes are scheduled in FCFS manner.
5. A process in lower priority queue can only execute only when higher priority queues are empty.
6. A process running in the lower priority queue is interrupted by a process arriving in the higher priority queue.

Example

Q-Consider a system which has a CPU bound process, which requires the burst time of 40 seconds. The multilevel Feed Back Queue scheduling algorithm is used and the queue time quantum '2' seconds and in each level it is incremented by '5' seconds. Then how many times the process will be interrupted and on which queue the process will terminate the execution?

Solution –

Process P needs 40 Seconds for total execution.

At Queue_1 it is executed for 2 seconds and then interrupted and shifted to queue 2.

At Queue_2 it is executed for 7 seconds and then interrupted and shifted to queue_3.

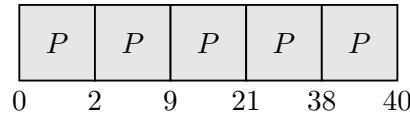
At Queue_3 it is executed for 12 seconds and then interrupted and shifted to queue_4.

At Queue_4 it is executed for 17 seconds and then interrupted and shifted to queue_5.

At Queue_5 it executes for 2 seconds and then it completes.

Hence the process is interrupted 4 times and completes on queue_5.

Gantt chart for Process P in Multi-level feedback queue Scheduling



Example

Consider a Multi-level feedback queue scheduler with three queues, numbered from 0 to 2. Show how to schedule the following processes.

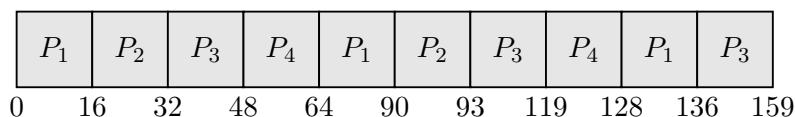
Q0-RR with quantum 16

Q1-RR with quantum 26

Q2-FCFS

Processes	Arrival time	Burst time
P1	0	50
P2	0	19
P3	0	65
P4	0	25

Gantt chart for MLFQ



Calculation of waiting time, turn around time, and Response time of each process

Process	AT	BT	WT	TAT	RT
P1	0	50	86	136	0
P2	0	19	74	93	16
P3	0	65	94	159	32
P4	0	25	103	128	48



Figure 17.3: Multi-level feedback Queue Scheduler with three queues

17.2.1 Advantages

1. It is more flexible.
2. It allows different processes to move between different queues.
3. It prevents starvation by moving a process that waits too long for lower priority queue to the higher priority queue.

17.2.2 Disadvantages

1. For the selection of the best scheduler, it require some other means to select the values.
2. It produces more CPU overheads.
3. It is most complex algorithm.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

18 Synchronisation	2
18.1 Introduction	2
18.2 Background	2
18.3 Critical section problem	5
18.4 Solution to Critical-Section Problem	6
18.5 Critical section handling in OS	6

Lecture 18

Synchronisation

18.1 Introduction

- We know that, the processes are categorized as the following two types:
 - Independent Process
 - Cooperative Process
- *Independent Process* : A Independent Process is one that can not affect or not be affected by other processes executing in the system.
- *Cooperative Process* : A cooperating process is one that can affect or be affected by other processes executing in the system.
- Process synchronization problem arises in the case of Cooperative processes because 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.
- But concurrent access to shared data may result in data inconsistency.
- Here we will discuss several mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency can be maintained.

18.2 Background

- Processes can execute concurrently or in parallel.
- The concurrent process execution means that one process may only partially complete execution before another process is scheduled. In fact, a process may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instructions of another process.
- In parallel execution, in which two instruction streams (representing different processes) execute simultaneously on separate processing cores.

- Here, we discuss how concurrent or parallel execution can contribute to issues involving the integrity of data shared by several processes.
- In Inter-process communication (In Chapter 3), we have discussed on producer-consumer problem using bounded buffer which could be used to enable processes to share memory and at most `BUFFERSIZE-1` items are allowed in the buffer at the same time.
- The discussed code for the shared data definition

```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- The discussed code for the producer process

```
item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)           //buffer full
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

- The discussed code for the consumer process

```
item next_consumed;
while (true) {
    while (in == out)           //buffer empty
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```

- To overcome his deficiency, we want to modify the algorithm.
 - hence, we can add an integer variable counter, initialized to 0.

- When the producer process produces an item, counter gets incremented every time and is decremented every time when the consumer process consumes an item.
- The code for the producer process can be modified as follows:

```

while (true) {
    /* produce an item in next_produced */

    while (counter == BUFFER_SIZE)
        ; /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}

```

- The code for the consumer process can be modified as follows:

```

while (true) {
    while (counter == 0)
        ; /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;

    /* consume the item in next_consumed */
}

```

- Both producer and consumer routines are correct separately but they may not function correctly when executed concurrently.
 - Suppose currently counter value is 5.
 - If both the producer and consumer processes execute separately the statements, “counter++” and “counter–”, then the counter value will be 5 which is the correct result.
 - But if the producer and consumer processes concurrently execute the statements, “counter++” and “counter–”, then the counter value may be either 4, 5 or 6.
- We can visualize the value of counter may be incorrect as follows:
- In machine language counter++ could be implemented as

```

register1 = counter
register1 = register1 + 1
counter = register1

```

here, *register₁* is one of the local CPU registers.

- similarly, In machine language counter– could be implemented as

```

register2 = counter
register2 = register2 - 1
counter = register2

```

here, *register₂* is one of the local CPU registers.

- The concurrent execution of counter++ and counter– is equivalent to a sequential execution in which the machine-level statements are interleaved in some arbitrary order.
- Consider this execution interleaving with "count = 5" initially:

T_0 :	producer	execute	$register_1 = \text{counter}$	{ $register_1 = 5$ }
T_1 :	producer	execute	$register_1 = register_1 + 1$	{ $register_1 = 6$ }
T_2 :	consumer	execute	$register_2 = \text{counter}$	{ $register_2 = 5$ }
T_3 :	consumer	execute	$register_2 = register_2 - 1$	{ $register_2 = 4$ }
T_4 :	producer	execute	$\text{counter} = register_1$	{ $\text{counter} = 6$ }
T_5 :	consumer	execute	$\text{counter} = register_2$	{ $\text{counter} = 4$ }

- Here we have got the incorrect value of counter i.e 4
- If we reverse the orders of S4 and S5 then also we would get incorrect value of counter i.e 6, because we allowed both processes to manipulate the shared variable counter concurrently. This is called race condition.
- A race condition is a situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.
- So we need to ensure that only one process at a time can be manipulating the shared variable.

18.3 Critical section problem

- Consider a system of n processes P_0, P_1, \dots, P_{n-1} .
- Each process has a segment of code called a critical section, in which process may be changing common variables, updating table, writing file, etc.
- When one process is executing in critical section, no other process is to be allowed to execute in its critical section. i.e No two processes are executing in their critical sections at the same time.
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section.
- The general structure of process P_i is shown in figure below.

```

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

```

18.4 Solution to Critical-Section Problem

- A solution to critical-section problem must satisfy the following three requirements:
 - *Mutual Exclusion* : If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
 - *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.
 - *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.

18.5 Critical section handling in OS

- Two general approaches are used to handle critical sections in operating systems
 - *Preemptive kernel*
 - *Non-preemptive kernel*
- *Preemptive kernels* : it allows a process in execution in the kernel mode to be interrupted by some other process
- *Non-preemptive kernels* : it does not allow a process in execution in the kernel mode to be interrupted by some other process. It runs until exits kernel mode, blocks, or voluntarily yields CPU.
- It is difficult to design preemptive kernels where It is easier to design non-preemptive kernels.
- But a non-preemptive kernel is essentially free from race condition on kernel data structures because only one process is active in the kernel at a time.
- But this is not same in preemptive case, so they must be carefully designed to ensure that shared kernel data are free from race conditions.

- A preemptive kernel is more suitable for real-time programming because it will allow a real-time process to preempt a process currently running in the kernel.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

19 Peterson's Solution	2
19.1 Peterson's Solution	2
19.2 Correctness proof of Peterson's Solution	3

Lecture 19

Peterson's Solution

19.1 Peterson's Solution

- It is a classic software-based solution to the critical-section problem.
- Though there is no guarantee that this Peterson's solution will work correctly on modern computer architecture it provides a good algorithm description of solving the critical-section problems.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- Those two processes share two variables:
 - int turn;
 - boolean flag[2];
- The variable turn indicates whose turn it is to enter the critical section.
- Suppose two processes are P_i and P_j and if $\text{turn} == i$, then process P_i is allowed to execute in its critical section.
- The flag array is used to indicate if a process is ready to enter the critical section, i.e, if $\text{flag}[i] == \text{true}$ then process P_i is ready to enter its critical section.
- Lets describe the structure of process P_i in Peterson's solution in the following way:

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

- To enter the critical section, P_i first sets $\text{flag}[i]$ to be true.
- Then it sets turn to the value j means that if the other process wishes to enter the critical section, it can do so.
- If both processes try to enter the critical section at the same time, turn will be set to both i and j at roughly the same time.
- Only one of these assignments will last, the other will occur but will be overwritten immediately.

19.2 Correctness proof of Peterson's Solution

- In order to prove that this solution is correct, we need to show that:
 - Mutual exclusion is preserved.
 - The progress requirements is satisfied.
 - The bounded-waiting requirement is satisfied.
- *Mutual exclusion*
 - Each process P_i enters its critical section only if either $\text{flag}[j] == \text{false}$ or $\text{turn} == i$.
 - If both processes can be executing in their critical sections at the same time, then $\text{flag}[i] == \text{flag}[j] == \text{true}$.
 - These two observations indicate that P_i and P_j could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both.
 - As a result, for only one of the processes this turn variable will be set to true.
 - So, if both the processes have raised their request to enter into critical section, this flag variables will be true, but turn can have only one value at a time.
 - So, whichever process has got that thing satisfied that process will enter into the critical section.
 - So mutual exclusion is preserved.
- *Progress Requirement*
 - Progress requirement means that if we have got a number of processes waiting outside their critical section and they want to enter into the critical section, then the decision who will enter next cannot be postponed indefinitely.
 - It is decided solely by the turn variable that which process will enter into the critical section.

- So, only for one of the processes that while loop condition will be true and for the other processes it is definitely false.
 - whichever process this while loop condition is false, it will enter the critical section.
 - Then after finishing the critical section it will set this flag to be equal to false.
 - So the decision like who will enter into the critical section that is never getting postponed indefinitely.
 - So progress requirement is preserved.
- *Bounded waiting Requirement*
 - When a process comes and it finds that another process is in critical section, it cannot enter into the critical section.
 - Then when the other process will finish off the critical section the flag variable will be reset to false.
 - And, the next time this process will check the while loop condition it will find that flag is false
 - So it will enter into the critical section.
 - It is bounded by at most one execution of the other process.
 - Hence bounded wait is also satisfied.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

20 Synchronization Hardware	1
20.1 Synchronization Hardware	1
20.1.1 The test and set() Instruction	2
20.1.1.1 Solution using the test and set() instruction	2
20.1.2 The swap() Instruction	3
20.1.2.1 Solution using swap() instruction	3
20.1.3 Bounded-waiting Mutual Exclusion with test and set	3
20.1.4 Mutex locks	5
20.1.4.1 acquire() and release()	5

Lecture 20

Synchronization Hardware

20.1 Synchronization Hardware

1. In the last lecture we have discussed software-based Peterson's solution to the critical-section problem.
2. But this solution does not provide guarantee on modern computer architecture.
3. Here We explore several more solutions to the critical-section problem using techniques ranging from hardware to software based APIs available to application programmers.
4. Any solution to the critical-section problem requires a simple tool called a lock.
5. A process must acquire a lock before entering the critical-section and it must release the lock when it exits the critical-section.
6. This is shown in Figure 20.1 for the process P_i

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

Figure 20.1: General structure of process P_i

7. This solution is feasible in a uniprocessor environment where we could disable interrupts from occurring while a shared variable was being modified.
8. Currently running code would execute without preemption.

9. But this solution is too inefficient on multiprocessor systems.
10. It can be time consuming to disable interrupts on a multiprocessor system as the message is passed to all the processors.
11. Therefore modern machines provide special atomic hardware instructions.
 - (a) Atomic = non-interruptible
 - Either test memory word and set value
 - Or swap contents of two memory words

20.1.1 The test and set() Instruction

- The textbf{test and set()} instruction can be defined as

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

1. It is executed atomically.
2. It returns the original value of passed parameter.
3. It sets the new value of passed parameter to “TRUE”.

20.1.1.1 Solution using the test and set() instruction

- We can implement mutual exclusion by declaring a shared Boolean variable **lock**, initialized to false.
- **Solution:**

```
do
{
    while (test_and_set(&lock))
        ; /* do nothing */
    /* critical section */
    lock = false;
    /* remainder section */
} while(true);
```

```

void Swap( boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}

```

20.1.2 The swap() Instruction

- The swap() instruction operates on the contents of two words.
- It can be defined as:
 1. It is executed atomically.
 2. It returns the original value of passed parameter “value”.
 3. it sets the variable “value” the value of the passed parameter “new value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

20.1.2.1 Solution using swap() instruction

- Like the test and set() instruction, here also we can implement mutual exclusion by declaring a shared global Boolean variable **lock**, initialized to false.
- In addition, each process has a local variable ”key”.
- **Solution:**

```

do {
    key=TRUE;
    while (key==TRUE)
        swap(&lock, &key )
    /* critical section */
    lock = FALSE;
    /* remainder section */
} while (true);

```

20.1.3 Bounded-waiting Mutual Exclusion with test and set

- Both the test and set() and swap() instructions satisfy the mutual-exclusion requirement, but they do not satisfy the bounded-waiting requirement and hence progress requirement is also not satisfied.
- So another algorithm is described here using the test and set() instruction that satisfies all the three critical-section requirements.
- The common data structures are:
 - boolean waiting[n];
 - boolean lock;

- Both waiting[n] and lock are initialized to false.

```

do{
    waiting[i] = true;
    key=true;
    while (waiting[i] && key) /*lock=false
        key = test_and_set(&lock);
    waiting[i] = false;

    /* critical section */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
    j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;

    /* remainder section */

} while (true);

```

- Mutual exclusion is met because a process P_i can enter the critical section only if either waiting[i]=false or key=false.

-The key value can become false only if the test and set() is executed and the first process P_i wishing to enter the critical-section executes it and finds key value as false. So other processes must wait.

-The waiting[i] value can become false only if another process leaves its critical section. Only one waiting[i] is set to false.

-So mutual exclusion is satisfied.

- To prove whether progress requirement is met, we can note that when a process exiting the critical-section either sets **lock** to false or sets **waiting[j]** to false.
- It means both allow a process that is waiting to enter the critical-section to proceed.
 - Hence progress requirement is also met.
- To prove bounded-waiting requirement, we can note that when a process leaves its critical-section, it checks the waiting array whether any other process is interested to enter in critical-section in the cyclic ordering i.e (i+1), (i+2), ... (n-1), 0, 1, ... (i-1).
- Thus any process waiting to enter its critical-section will do so within n-1 turns.
 - So it is also satisfying the bounded-waiting requirement.

20.1.4 Mutex locks

1. The hardware-based solutions are complicated and generally inaccessible to application programmers.
2. OS designers build software tools to solve critical section problem.
3. Simplest is **mutex lock**.
4. It protects a critical section by first acquire() a lock then release() the lock.
5. Calls to acquire() and release() must be atomic Usually implemented via hardware atomic instructions.
6. But this solution requires busy waiting.
7. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This wastes CPU cycles that some other process might be able to use productively.
8. This lock therefore called a spinlock.

20.1.4.1 acquire() and release()

```
acquire()
{
    while (!available)
        /* busy wait */
    available=false;
}
```

```
release()
{
    available = true;
}
```

```
do{
    acquire lock
    critical section
    release lock
    remainder section
}while(true);
```



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by
Mr. Rakesh Kumar
Assistant Professor

Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

21 Semaphores	1
21.1 Semaphores	1
21.2 Semaphore Usage	2
21.3 Semaphore Implementation	3
21.3.1 Semaphore Implementation with no Busy waiting	3
21.3.2 Deadlock and Starvation	5
21.3.3 Deadlock and Starvation	6

Lecture 21

Semaphores

21.1 Semaphores

- A Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities is a **semaphore**.
- A semaphore S is an integer variable that is accessed only through two standard atomic operations: **wait()** and **signal()**.
- The **wait()** operation was originally termed P (from the Dutch proberen, "to test");
- The **signal()** was originally called V (from verhogen, "to increment").
- The definition of **wait()** is as follows:

```
wait(S) {  
    while (S <= 0)  
        ; // no-op  
    S- -;  
}
```

- The wait operation decrements the semaphore value S , if it is positive. If S is negative or zero, then no operation is performed.
- The definition of **signal()** is as follows:

```
signal(S) {  
    S++;  
}
```

- The signal operation increments the semaphore value S .
- when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

- These two operations are atomic operations.

21.2 Semaphore Usage

- There are two types of semaphores.
 1. Counting Semaphore.
 2. Binary Semaphore.
- **Counting Semaphore -**
 - These are integer value semaphores and have an unrestricted value domain
 - These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.
- **Binary Semaphore -**
 - The binary semaphores are like counting semaphores but their value is restricted to 0 and 1.
 - The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0.
 - It is sometimes easier to implement binary semaphores than counting semaphores.
 - The binary semaphores are known as mutex locks as they are locks that provide mutual exclusion.
 - The binary semaphores can be used to deal with the critical-section problem for multiple processes.
- We can also use semaphores to solve various synchronization problems.
- For example, consider two concurrently running processes: P_1 with a statement S_1 and P_2 with a statement S_2 and suppose we require that S_2 be executed only after S_1 has completed.
- **Solution:**
 - Let P_1 and P_2 share a common semaphore variable “synch” initialized to 0.
 - Then the steps are as follows:
 - P_2 will execute S_2 only after P_1 has invoked signal(synch), which is after statement S_1 has been executed.

```

P1:
    S1;
    Signal(synch);
P2:
    wait(Synch)
    S2;

```

21.3 Semaphore Implementation

- It must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time.
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section and how these operations can be implemented shown in figure 21.1.

```

do {
    wait(mutex);

    // critical section

    signal(mutex);

    // remainder section
} while (TRUE);

```

Figure 21.1: Mutual exclusion implementation with semaphore

- But this critical section implementation could now have busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of semaphore is also called **Spinlock** because the process "spins" while waiting for the lock.
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

21.3.1 Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
 - value (of type integer)
 - pointer to next record in the list

- Two operations:
 - block** – It places the process invoking the operation on the appropriate waiting queue.
 - wakeup** – It removes one of processes in the waiting queue and place it in the ready queue.
- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation
- The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.
- The semaphore can be defined as

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

- Each semaphore has an integer value and a list of processes list.
- When a process must wait on a semaphore, it is added to the list of processes.
- A signal() operation removes one process from the list of waiting processes and awakens that process.
- The wait() semaphore operation can be defined as

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

- The signal () semaphore operation can now be defined as

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

- The block() operation suspends the process that invokes it.
- The wakeup(P) operation resumes the execution of a blocked process P.
- These two operations are provided by the operating system as basic system calls.

- It is critical that semaphore operations be executed atomically. We must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time.
- This is a critical-section problem and in a single-processor environment, we can solve it by simply inhibiting interrupts during the time the wait() and signal() operations are executing.
- In a multiprocessor environment, interrupts must be disabled on every processor. Otherwise, instructions from different processes (running on different processors) may be interleaved in some arbitrary way.
- Therefore, SMP systems must provide alternative locking techniques—such as compare and swap() or spinlocks—to ensure that wait() and signal() are performed atomically.
- Actually Spinlocks have an advantage that no context switching is required when a process must wait on a lock.

21.3.2 Deadlock and Starvation

- The implementation of a semaphore with a waiting queue may result
- Two operations:

Deadlock –A situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes and these processes are said to be deadlocked.

Starvation – Starvation(Indefinite blocking)- A situation in which processes wait indefinitely within the semaphore.

Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

- Let a system consisting of two processes P_0 and P_1 , each accessing two semaphores S and Q initialized to 1

```

P_{0}
wait(S);    P_{1}
            wait(Q);
            wait(Q);    wait(S);

.....
signal(S);  signal(Q);
signal(Q);  signal(S);

```

- Suppose P_0 executes wait(S) and then P_1 executes wait(Q).
- When P_0 executes wait(Q), it must wait until P_1 executes signal(Q).
- Similarly, when P_1 executes wait(S), it must wait until P_0 executes signal(S).
- Since these signal() operations can not be executed, P_0 and P_1 are deadlocked.

21.3.3 Deadlock and Starvation

- Priority Inversion – Scheduling problem arises when lower-priority process holds a lock needed by higher-priority process. this is called priority inversion.
- Suppose a system consisting of three processes, X, Y and Z whose priorities follow the order $X \downarrow Y \downarrow Z$.
- Assume that process Z requires resource R, which is currently being accessed by process X.
- So process Z would wait for X to finish using resource R.
- However, now suppose that process Y becomes runnable, thereby preempting process X.
- So a process with a lower priority(process Y) has affected how long process Z must wait for X to relinquish resource R. This is Priority inversion
- This problem can be solved by implementing a **priority-inheritance protocol**.
- According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources.
- When they are finished, their priorities revert to their original values
- In the above example a priority-inheritance protocol allows the process X to temporarily inherit the priority of process Z, and is preventing the process Y from preempting its execution.
- When the process X has finished using resource R, it relinquishes its inherited priority from Z and assumes its original priority
- Because now the resource R is available, process Z will run next but not Y.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

22 Process synchronization	1
22.1 Classical Problems of Synchronization:	1
22.2 Bounded Buffer Problem	1
22.3 Reader-Writer problem	4

Lecture 22

Process synchronization

22.1 Classical Problems of Synchronization:

Classical problems used to test newly-proposed synchronization schemes, In our solutions to the problems, we use semaphores for synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

22.2 Bounded Buffer Problem

Bounded buffer problem, which is also called producer consumer problem, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

What is the Problem Statement?

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, producer and consumer, which are operating on the buffer. A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't

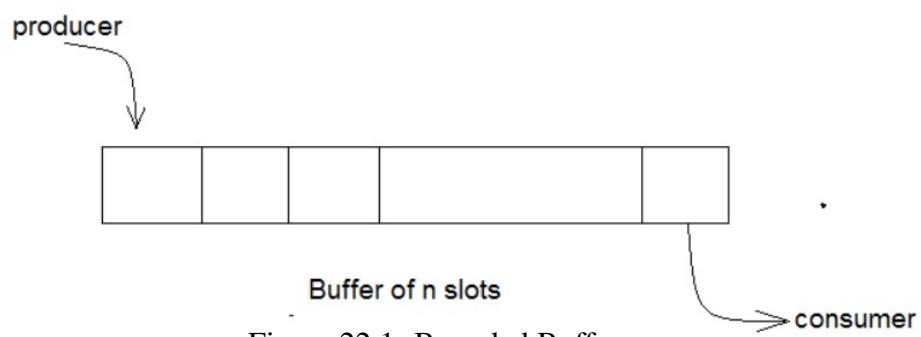


Figure 22.1: Bounded Buffer

produce the expected output if they are being executed concurrently. There needs to be a way to make the producer and consumer work in an independent manner.

- N buffers, each can hold one item
- **Semaphore mutex initialized to the value 1** (a binary semaphore which is used to acquire and release the lock.)
- **Semaphore full initialized to the value 0** (a counting semaphore whose initial value is 0.)
- **Semaphore empty initialized to the value N** (a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.)

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

The pseudocode of the producer function looks like this:

```
do
{
    // wait until empty > 0 and then decrement 'empty'
    wait(empty);

    // acquire lock
    wait(mutex);

    /* perform the insert operation in a slot */

    // release lock
    signal(mutex);

    // increment 'full'
    signal(full);
}

while(TRUE);
```

How producer code work?

- Looking at the above code for a producer, we can see that a producer first waits until there is atleast one empty slot.
- Then it decrements the empty semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
- Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
- After performing the insert operation, the lock is released and the value of full is incremented because the producer has just filled a slot in the buffer.

The pseudocode for the consumer function looks like this:

```

do

{

    // wait until full > 0 and then decrement 'full'

    wait(full);

    // acquire the lock

    wait(mutex);

    /* perform the remove operation in a slot */

    // release the lock

    signal(mutex);

    // increment 'empty'

    signal(empty);

}

while(TRUE);

```

How consumer code work

- The consumer waits until there is atleast one full slot in the buffer.
- Then it decrements the full semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.

- After that, the consumer acquires lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
- Then, the consumer releases the lock.
- Finally, the empty semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

22.3 Reader-Writer problem

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

The Problem Statement

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are reader and writer. Any number of readers can read from the shared resource simultaneously, but only one writer can write to the shared resource. When a writer is writing data to the resource, no other process can access the resource. A writer cannot write to the resource if there are non zero number of readers accessing the resource at that time.

The Solution From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

The reader-writer problem has several variation all involving priorities.

case1: The Reader Priority : Reader-writer problem a waiting reader process get more priority than a waiting writer process.

In other word, no reader should wait for other readers to finish simply because a writer is waiting.

Case 2: In Writer's Priority : Reader-writer problem a waiting writer process get more priority than a waiting reader process. In other word, If writer is waiting to access the object , no new reader may start reading. A solution to reader writer problem may result in starvation.

In first case, Writer process may starve.

In Second case, Reader process may starve.

Shared Data

- Data set
- Semaphore mutex initialized to 1 (we use the mutex to make the process to acquire and release lock whenever it is updating the read_count variable.)
- Semaphore wrt initialized to 1 (wrt is used to achieve mutual exclusion for writer process.)

- Integer readcount initialized to 0 (An integer variable read_count is used to maintain the number of readers currently accessing the resource.)

The structure of a writer process

```

do {

    wait(wrt);

    // writing is performed

    signal(wrt);

} while(TRUE);

```

The structure of a reader process

```

do {

    wait(mutex) ;

    readcount ++ ;

    if(readcount == 1)

        wait(wrt) ;

    signal(mutex)

    // reading is performed

    wait(mutex) ;

    readcount -- ;

    if(readcount == 0)

        signal(wrt) ;

    signal(mutex) ;

} while(TRUE);

```

How above code work

- The code for the writer, the writer just waits on the wrt semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments wrt so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the read_count is updated by a process.
- When a reader wants to access the resource, first it increments the read_count value, then accesses the resource and then decrements the read_count value.
- The semaphore wrt is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the wrt semaphore because there are zero readers now and a writer can have the chance to access the resource.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

23 Process synchronization	1
23.1 2nd Variation of Reader-Writer problem	1
23.2 Dining Philosophers Problem (DPP)	3

Lecture 23

Process synchronization

23.1 2nd Variation of Reader-Writer problem

The second readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the shared data, no new readers may start reading.

A solution to either the first readers-writers problem or the second readers-writers problem may result in starvation. In the first case, writers might starve; in the second case, readers might starve. The basic structure of the solution to the second readers-writers problem is as follows:

Reader

wait until no active or waiting writers

access database

check out -- wake up waiting writer

Writer

wait until no active readers or writers

access database

check out wake up waiting readers or writer

Similar to the producer-consumer problem, more than one locks are needed. Use a separate semaphore for each constraint;

- **Semaphore okToDelete;** // reader's constraint, if 0, no reader can access database

- **Semaphore okToWrite;** // writer's constraint, if 0, no writer can access database
- **Semaphore mutex;** // mutual exclusion
- **int AR = 0** //number of active readers
- **int AW = 0** // number of active writers
- **int WR = 0** // number of waiting readers
- **int WW = 0** //number of waiting writers

Code for readers:

Reader()

```

{
  wait(mutex);
  if ((AW + WW) == 0) // if no any writers
  {
    signal(okToRead);
    AR = AR + 1;
  }else
    WR = WR + 1;
  signal(mutex);
  wait(okToRead);
  — Read DB—
  wait(mutex);
  AR = AR - 1;
  if (AR == 0 && WW > 0) //if there are no other active readers but waiting
writers
{
  signal(okToWrite);
  AW = AW + 1;
  WW = WW - 1;
}
signal(mutex);
}
```

Code for writers:

Writer() {

```

{
  wait(mutex);
  if ((AW + AR+WW) ==0) // if no active writer or active readers, term WW
redundant
{
  signal(okToWrite);
```

```

AW = AW + 1;
} else WW = WW + 1;
signal(mutex);
wait(okToWrite);
—Write Data —
wait(mutex);
AW = AW - 1;
if (WW > 0) // give priority to other writers
{
    signal(okToWrite);
    AW = AW + 1;
    WW = WW - 1;
}
else while (WR > 0)
{
    signal(okToRead);
    AR = AR + 1;
    WR = WR - 1;
}
signal(mutex);
}

```

23.2 Dining Philosophers Problem (DPP)

The dining philosopher is a classic synchronization problem as it demonstrates a large class of concurrency control problems.

Problem statement

The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available. Otherwise a philosopher puts down their chopstick and begin thinking again.

Solution of Dining Philosophers Problem

A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.

The structure of the chopstick is shown below



Figure 23.1: Dining Philosophers Problem

semaphore chopstick [5];

Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher.

The structure of a random philosopher i is given as follows

```

do {
    wait( chopstick[i] );
    wait( chopstick[ (i+1) % 5 ] );
    ..
    EATING THE RICE
    signal( chopstick[i] );
    signal( chopstick[ (i+1) % 5 ] );
    THINKING
} while(1);

```

In the above structure, first wait operation is performed on $\text{chopstick}[i]$ and $\text{chopstick}[(i+1) \% 5]$. This means that the philosopher i has picked up the chopsticks on his sides. Then the eating function is performed.

After that, signal operation is performed on $\text{chopstick}[i]$ and $\text{chopstick}[(i+1) \% 5]$. This means that the philosopher i has eaten and put down the chopsticks on his sides. Then the philosopher goes back to thinking.

Difficulty with the solution

The above solution makes sure that no two neighboring philosophers can eat at the same time. But this solution can lead to a deadlock. This may happen if all the philosophers pick their left chopstick simultaneously. Then none of them can eat and deadlock occurs.

Some of the ways to avoid deadlock are as follows

- There should be at most four philosophers on the table.
- An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.
- A philosopher should only be allowed to pick their chopstick if both are available at the same time.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

24 Process synchronization	1
24.1 Monitors	1
24.1.1 why use monitors ?	1
24.1.2 Definition of Monitor	2
24.1.3 Syntax of monitor	3
24.1.4 Schematic view of a monitor	3
24.1.5 Monitor with Condition Variables	4
24.2 Dining-Philosophers Solution Using Monitors	4

Lecture 24

Process synchronization

24.1 Monitors

Monitor is one of the way to achieve process synchronization. Monitor is supported by programming language to achieve mutual exclusion between processes.

24.1.1 why use monitors ?

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors.

Unfortunately, such timing errors can still occur when semaphores are used. To illustrate how, we review the semaphore solution to the critical-section problem. All processes share a semaphore variable mutex, which is initialized to 1.

Each process must execute `wait(mutex)` before entering the critical section and `signal(mutex)` afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously. Next, we examine the various difficulties that may result. Note that these difficulties will arise even if a single process is not well behaved. This situation may be caused by an honest programming error or an uncooperative programmer.

- Suppose that a process interchanges the order in which the `wait()` and `signal()` operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);  
...  
critical section  
...  
wait(mutex);
```

In this situation, several processes maybe executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active.

- Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes

```

wait(mutex);
...
critical section
...
wait(mutex);

```

In this case, a deadlock will occur.

- Suppose that a process omits the wait(mutex), or the signal(mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem. To deal with such errors, researchers have developed high-level language constructs. In this section, we describe one fundamental high-level synchronization construct—the monitor type.

24.1.2 Definition of Monitor

- Monitors are abstract data type for defining shared objects (shared resources) and are used for scheduling the shared object in a multiprogramming environment.
- A monitor definition consists of the shared resources (object), predefined set of procedures representing various possible operations to access the resources and a set of local administrative data.
- A monitor ensures that at a given time only one process is active within the monitor. Any other process requesting an access are blocked on monitors entry queue.
- Procedures within a monitor are the gateway to access the shared resources and hence any process that needs an access to the shared resource must call the appropriate procedure.
- To achieve synchronization among multiple concurrently executing processes and requesting an access to shared resource, monitors make use of two special operations mainly wait and signal which are executed within monitor procedures.
- Monitors also make use of one or more condition variables on which the wait and signal operation acts.
- Condition variable represent different condition on which requesting processes may either be blocked or unblocked using wait and signal operations.

24.1.3 Syntax of monitor

```
monitor monitor name  
{ /* shared variable declarations */  
function P1 ( . . . ) { . . .  
}  
function P2 ( . . . ) { . . .  
}  
. . .  
. . .  
function Pn ( . . . ) { . . .  
}  
initialization code ( . . . ) { . . .  
}  
}
```

24.1.4 Schematic view of a monitor

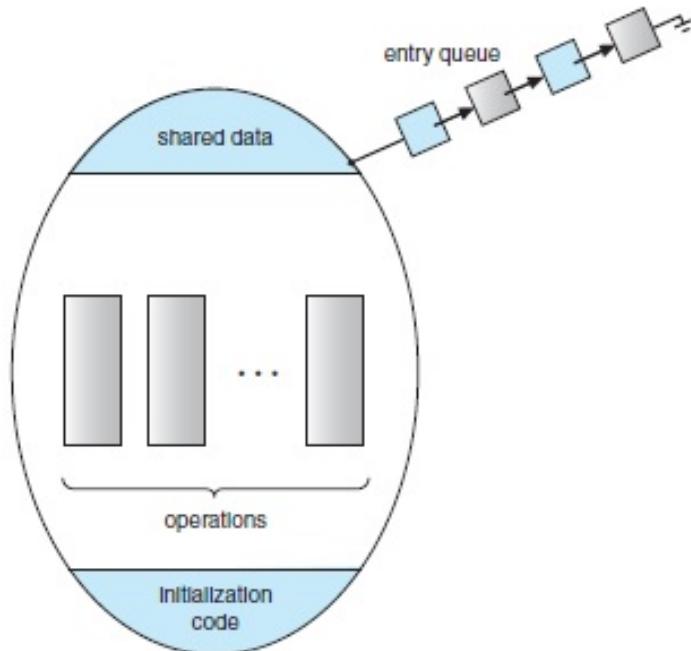


Figure 24.1: Schematic view of a monitor.

24.1.5 Monitor with Condition Variables

- `<condition variable>.wait` : Blocks the caller process on the queue associated with the condition variables.
Example: `X.wait`
- `<condition variable>.signal` : unblock exactly one process waiting on the queue of the condition variable which regains control of the monitor.
Example: `X.signal`
- `<condition variable>.queue` : is true if the queue is not empty and false if queue is empty.

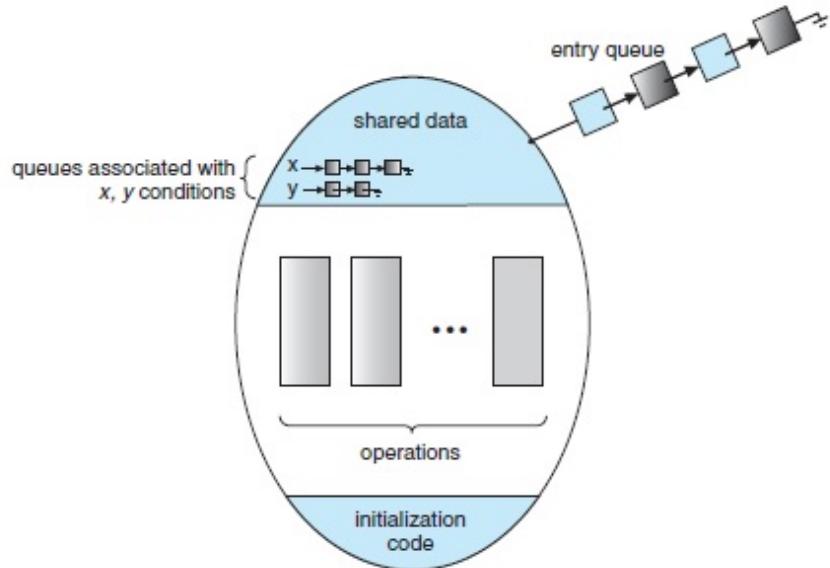


Figure 24.2: Monitor with Condition Variables.

24.2 Dining-Philosophers Solution Using Monitors

We illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

```
enum {THINKING, HUNGRY, EATING} state[5];
```

Philosopher i can set the variable $\text{state}[i] = \text{EATING}$ only if her two neighbors are not eating: $(\text{state}[(i + 4) \% 5] \neq \text{EATING}) \text{ and } (\text{state}[(i + 1) \% 5] \neq \text{EATING})$.

We also need to declare

```
condition self[5];
```

This allows philosopher i to delay herself when she is hungry but is unable to obtain the chopsticks she needs. Each philosopher, before starting to eat, must invoke the operation `pickup()`. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the `putdown()` operation.

```
monitor DiningPhilosophers
```

```
{ enum {THINKING, HUNGRY, EATING} state[5];
```

```
condition self[5];
```

```
void pickup(int i)
```

```
{
```

```
state[i] = HUNGRY;
```

```
test(i);
```

```
if (state[i] != EATING)
```

```
self[i].wait();
```

```
}
```

```
void putdown(int i)
```

```
{
```

```
state[i] = THINKING;
```

```
test((i + 4) % 5);
```

```
test((i + 1) % 5);
```

```
}
```

```
void test(int i)
```

```
{
```

```
if ((state[(i + 4)%5] != EATING) && (state[i] == HUNGRY) && (state[(i + 1)%5]!= EATING))
```

```
{
```

```
state[i] = EATING;
```

```
self[i].signal();
```

```
}
```

```
}
```

```
initialization code()
```

```
{
```

```
for (int i = 0; i < 5; i++)
```

```
state[i]= THINKING;
```

```
}
```

```
}
```

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

```
DiningPhilosophers.pickup(i);  
...  
eat  
...  
DiningPhilosophers.putdown(i);
```



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

25 Deadlocks	1
25.1 Deadlocks	1
25.1.1 System model	2
25.1.2 Necessary Conditions	2
25.1.3 Resource-Allocation Graph	3
25.1.4 Methods for Handling Deadlocks	6
25.2 Deadlock Prevention	6

Lecture 25

Deadlocks

25.1 Deadlocks

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.

or

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. A similar situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s).

For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource

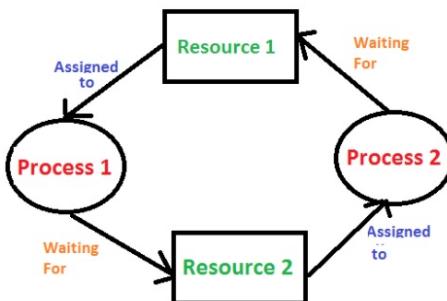


Figure 25.1: process 1 and process 2 in deadlocks

2 which is acquired by process 2, and process 2 is waiting for resource 1.

25.1.1 System model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources may be partitioned into several types (or classes), each consisting of some number of identical instances. CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

- **Request.** The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
- **Use.** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
- **Release.** The process releases the resource.

Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process P_i is holding the DVD and process P_j is holding the printer. If P_i requests the printer and P_j requests the DVD drive, a deadlock occurs.

25.1.2 Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

- **Mutual exclusion.** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
- **Hold and wait.** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
- **No preemption.** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait.** A set P_0, P_1, \dots, P_n of waiting processes must exist such that P_0 is waiting for a resource held by P_1 , P_1 is waiting for a resource held by P_2 , ..., P_{n-1} is waiting for a resource held by P_n , and P_n is waiting for a resource held by P_0 .

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

25.1.3 Resource-Allocation Graph

A **resource-allocation graph** tracks which resources is held by which process and which process is waiting for a resource of a particular type. It is very powerful and simple tool to illustrate how interacting processes can deadlock.

This graph consists of a set of vertices V and a set of edges E . The set of vertices V is partitioned into two different types of nodes: $P = P_1, P_2, \dots, P_n$, the set consisting of all the active processes in the system, and $R = R_1, R_2, \dots, R_m$, the set consisting of all resource types in the system.

The vertices in a resource allocation graph(RAG) can be of two types:

- A set of all active processes represented as **circle**.
- Set of available resources type each represented as **rectangles**.
- A resource can have more than one instance. Each instance will be represented by a **dot** inside the rectangle.

Edges in a resource allocation graph(RAG)can also be two types:

- **Request edges**
- **Allocation edge/ Assignment edge**

A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$; it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource. This directed edge $P_i \rightarrow R_j$ is called a **request edge**;

A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i . This directed edge $R_j \rightarrow P_i$ is called an **assignment edge**.

Note: If there is a cycle in the resource allocation graph and each resource in the cycle provide only one instance than the processes will deadlock.

Example 1:

The sets P , R , and E :

- $P = P_1, P_2, P_3$
- $R = R_1, R_2, R_3, R_4$
- $E = P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3$

Resource instances:

- One instance of resource type R1
- Two instances of resource type R2
- One instance of resource type R3
- Three instances of resource type R4

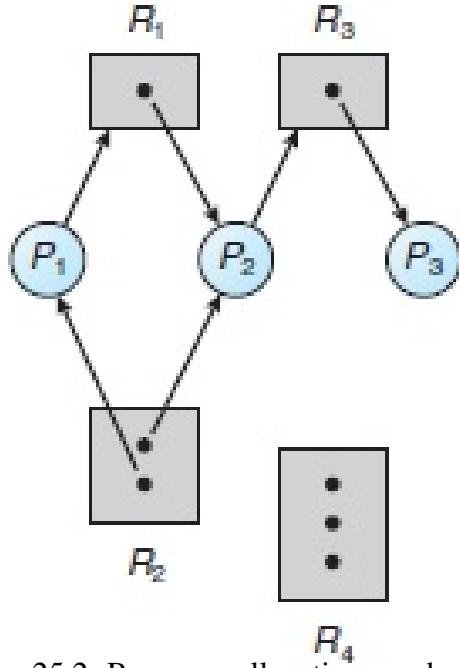


Figure 25.2: Resource-allocation graph.

Process states:

- Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1 .
- Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3 .
- Process P_3 is holding an instance of R_3 .

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient

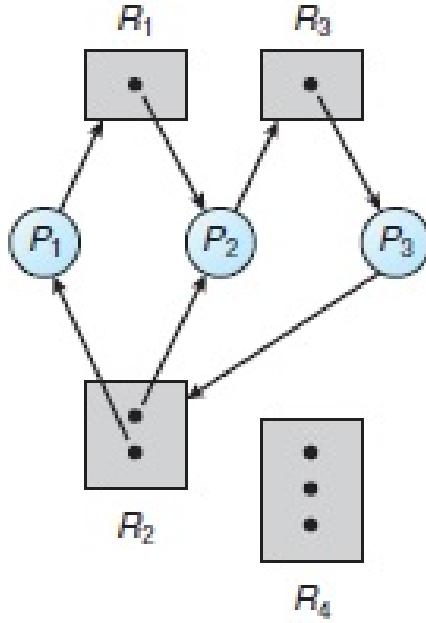


Figure 25.3: Resource-allocation graph with a deadlock.

condition for the existence of deadlock.

To illustrate this concept, we return to the resource-allocation graph depicted in (Figure 7.2). Suppose that process P3 requests an instance of resource type R2. Since no resource instance is currently available, we add a request edge $P_3 \rightarrow R_2$ to the graph (Figure 7.3). At this point, two minimal cycles exist in the system:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is

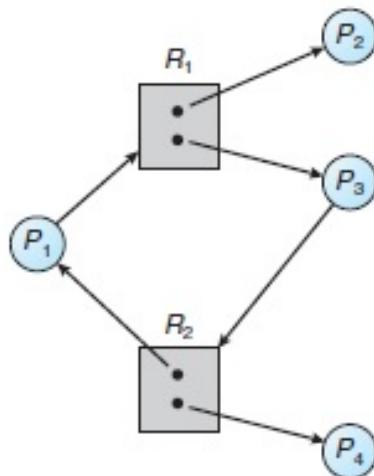


Figure 25.4: Resource-allocation graph with a cycle but no deadlock.

held by process P3. Process P3 is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1. Now consider the

resource-allocation graph in Figure 7.4. In this example, we also have a cycle:

$$P1 \rightarrow R1 \rightarrow P3 \rightarrow R2 \rightarrow P1$$

However, there is no deadlock. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle. In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.

25.1.4 Methods for Handling Deadlocks

The following are the 3 possible ways to deal with or handle the possible ways of dealing with deadlock situations.

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

25.2 Deadlock Prevention

We can prevent Deadlock by eliminating any of the above four conditions.

Eliminate Mutual Exclusion

It is not possible to dis-satisfy the mutual exclusion because some resources, such as the tape drive and printer, are inherently non-shareable.

Eliminate Hold and wait

Allocate all required resources to the process before the start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization. for example, if a process requires printer at a later time and we have allocated printer before the start of its execution printer will remain blocked till it has completed its execution.

The process will make a new request for resources after releasing the current set of resources. This solution may lead to starvation.

Eliminate No Preemption

Preempt resources from the process when resources required by other high priority processes.

Eliminate Circular Wait

Each resource will be assigned with a numerical number. A process can request the resources increasing/decreasing order of numbering. For Example, if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by
Mr. Rakesh Kumar
Assistant Professor

Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

26 Deadlock avoidance algorithm	1
26.1 Deadlock Avoidance	1
26.1.1 Safe State	1
26.2 Types of Deadlock avoidance algorithm	3
26.2.1 Resource allocation graph algorithm	3
26.2.2 Banker's Algorithm	6
26.2.2.1 Safety Algorithm	7
26.2.2.2 Resource Request Algorithm	8

Lecture 26

Deadlock avoidance algorithm

26.1 Deadlock Avoidance

- Deadlock-prevention algorithms, prevent deadlocks by limiting how requests can be made. The limits ensure that at least one of the necessary conditions for deadlock cannot occur. Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.
- An alternative method for avoiding deadlocks is to require additional information about which resources a process will request and use during its life time. With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.
- Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.
- The various algorithms that use this approach differ in the amount and type of information required.
- 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

26.1.1 Safe State

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

- System is in safe state if there exists a sequence P1, P2, ..., Pn 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 plus resources held by all the Pj, with j < i
- 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, P_{i+1} can obtain its needed resources, and so on.

- Example:

Let initially a resource R1 has 12 instances. Let he current resource allocation state is as follows:

Process	Max need	current need	resources allocated
P0	10	5	5
P1	4	2	2
P2	9	7	2

Now the current available is 3. So P1 can immediately be allocated all its resources and after completing its task can release them. Then P0 can get all its resources and after completing its task can release them. Finally P2 can get all its resources and return them. So here P1,P0 and P2 is the safe sequence. Hence the given resource allocation state is safe.

Suppose P2 requests and is allocated one more resource. Then the new resource allocation state is as follows:

Process	Max need	current need	resources allocated
P0	10	5	5
P1	4	2	2
P2	9	6	3

Now the current available is 2. So P1 can only be allocated all its resources and after completing its task it can release them. Now the number of available instances of R1 is 4, which cann't satisfy the requirement of neither P0 nor P2. So P0 and P2 will wait for each other. No safe sequence is possible here, that implies the resource allocation state is unsafe.

- The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.

26.2 Types of Deadlock avoidance algorithm

- Single instance of a resource type
Use a resource-allocation graph
- Multiple instances of a resource type
Use the banker's algorithm

26.2.1 Resource allocation graph algorithm

- A variant of resource allocation graph is used for deadlock avoidance. Along with request and assignment edge a claim edge from process to resource vertex is introduced in this graph.
- Claim edge P_i to R_j indicates that process P_j may request resource R_j . It is represented by a dashed line.
- Claim edge is converted to request edge when a process requests a resource.
- Request edge is converted to an assignment edge when the resource is allocated to the process.
- When a resource is released by a process, assignment edge is reconverted to a claim edge.
- Resources must be claimed a priori in the system. i.e. before process P_i starts executing, all its claim edges must already appear in the resource-allocation graph.
- Suppose that process P_i requests a resource R_j . The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph. No cycle implies safe state, that implies no deadlock.
- Example:

Let there are two processes P_1 and P_2 . There are two resources R_1, R_2 each one having a single instance. P_1 and P_2 both needs R_1 and R_2 to complete its task. Initially the maximum need of both processes are represented in resource allocation graph as follows:

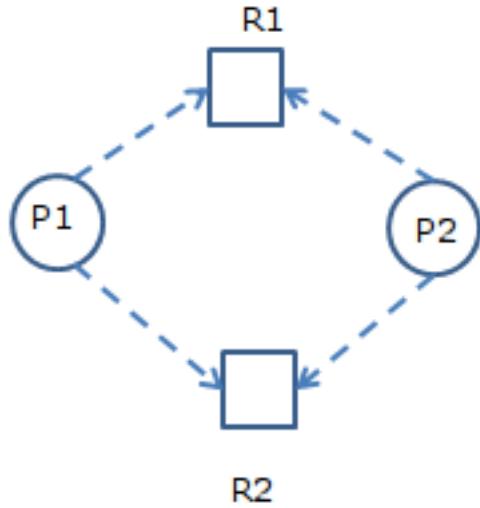


Figure 26.1: Initial Resource allocation graph

Now the P1 requests for resource R1. So resource allocation graph will be as follows:

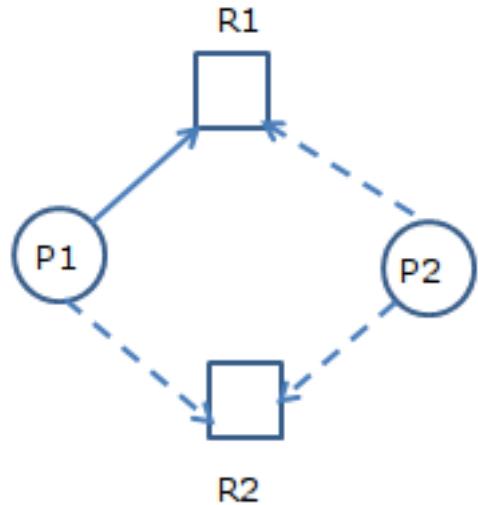


Figure 26.2: P1 requests R1

Converting Request edge to assignment edge, there is no cycle in the graph. So R1 is allocated to P1. The new resource allocation graph will be as follows:

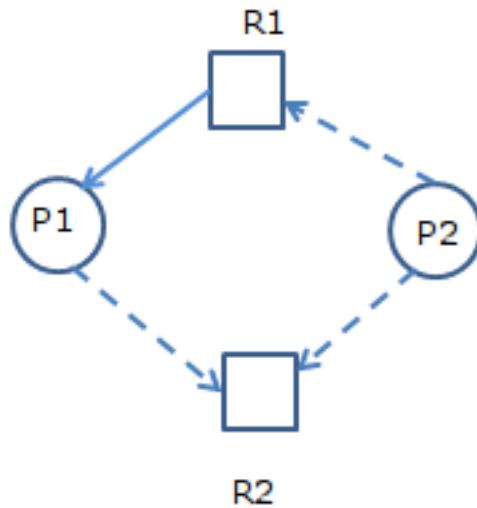


Figure 26.3: R1 is assigned to P1

Let P2 requests for R2. So resource allocation graph will be as follows:

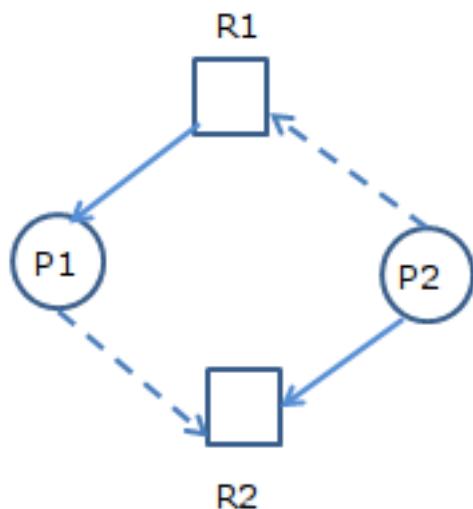


Figure 26.4: P2 requests R2

Converting the request edge to assignment edge, it is found that there is a cycle in the graph as follows:

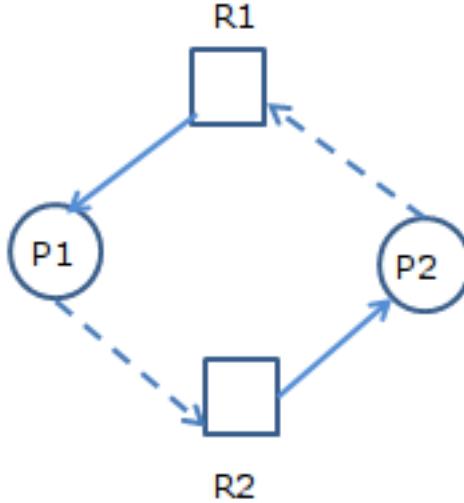


Figure 26.5: R2 is assigned to P2

So the request of P2 for R2 will not be granted immediately, though R2 is currently available.

26.2.2 Banker's Algorithm

- It is used to a resource allocation system with multiple instances of each resource type.
- Each process must a priori claim maximum use. When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

- [Data Structures for the Banker's Algorithm](#)

Let m= number of resources and n= number of processes

- [Available](#): Vector of length m. If available [j] = k, there are k instances of resource type R_j available
- [Max](#): n x m matrix. If Max [i,j] = k, then process P_i may request at most k instances of resource type R_j

- **Allocation**: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj
- **Need**: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task.

$$\text{Need } [i,j] = \text{Max}[i,j] - \text{Allocation } [i,j]$$
- There are two parts of the Banker's algorithm:
 - Safety Algorithm
 - Resource Request Algorithm

26.2.2.1 Safety Algorithm

1. Let Work and Finish be vectors of length m and n, respectively. Initialize:
 $\text{Work} = \text{Available}$
 $\text{Finish } [i] = \text{false for } i = 0, 1, \dots, n-1$
2. Find an i such that both:
 - (a) $\text{Finish } [i] = \text{false}$
 - (b) $\text{Need}_i < \text{Work}$
 If no such i exists, go to step 4.
3. $\text{Work} = \text{Work} + \text{Allocation}_i$
 $\text{Finish}[i] = \text{true}$
 go to step 2.
4. If $\text{Finish } [i] == \text{true}$ for all i, then the system is in a safe state.

Example:

Let there are 5 processes P0 through P4 and 3 resource types A,B and C. Initially A has 10 instances, B has 5 instances and C has 7 instances available. At time T0 the resource allocation state is represented as follows:

Process	Max	Allocation	Need	Available
	A B C	A B C	A B C	A B C
P0	7 5 3	0 1 0	7 4 3	3 3 2
P1	3 2 2	2 0 0	1 2 2	
P2	9 0 2	3 0 2	6 0 0	
P3	2 2 2	2 1 1	0 1 1	
P4	4 3 3	0 0 2	4 3 1	

Now check the specified resource allocate state is safe or not!

Implementation of safety algorithm to check the specified resource allocation state is safe or not

Initially work= 3 3 2

Initially Finish	Process	Need _i <= Work	Updated Finish	Safe sequence
F	P0	743<=332(F)		
F	P1	122<=332 (T)		
		work= 332+200=532	T	P1
F	P2	600<=532(F)		
F	P3	011<=532(T)		
		work=532+211=743	T	P1,P3
F	P4	431<=743(T)		
		Work=743+002=745	T	P1,P3,P4
F	P0	743<=745(T)		
		work=745+010=755	T	P1,P3,P4,P0
F	P2	600<=745(T)		
		work=755+302=10 5 7	T	P1,P3,P4,P0,P2

As the safe sequence P1,P3,P4,P0,P2 exists here, so the given resource allocation state is safe.

26.2.2.2 Resource Request Algorithm

Let Request_i = request vector for process Pi.

If Request_i[j] = k then process Pi wants k instances of resource type R_j. When a request for resources is made by process Pi , the following actions are taken:

1. If Request_i <= Need_i go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If Request_i <= 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:

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

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

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

If the resultant resource allocation state is safe, then the resources are allocated to Pi otherwise if it is unsafe Pi must wait, and the old resource-allocation state is restored.

Example:

Let there are 5 processes P0 through P4 and 3 resource types A,B and C. At time T0 the resource allocation state is represented as follows:

Process	Max	Allocation			Need	Available
		A	B	C		
P0	7 5 3	0 1 0	7 4 3	3 3 2		
P1	3 2 2	2 0 0	1 2 2			
P2	9 0 2	3 0 2	6 0 0			
P3	2 2 2	2 1 1	0 1 1			
P4	4 3 3	0 0 2	4 3 1			

The specified resource allocation state is currently safe. Suppose now process P1 requests one additional instance of resource type A and two instances of resource type C, so Request1 = (1,0,2). Let's use the banker's algorithm, to decide whether this request can be immediately granted or not. Here first the resource request algorithm is called and then the resultant resource allocation state is checked using safety algorithm.

- Implementation of Resource Request algorithm

- Request1 ; Need1 102;122 (true) go to step 2
- Request1 <= Available (that is, (1,0,2) <= (3,3,2) (true) go to step 3
- Available=332-102=230
Allocation1=200+102=302
Need1=122-102=020

Updated Resource allocation state is:

Process	Max	Allocation			Need	Available
		A	B	C		
P0	7 5 3	0 1 0	7 4 3	2 3 0		
P1	3 2 2	3 0 2	0 2 0			
P2	9 0 2	3 0 2	6 0 0			
P3	2 2 2	2 1 1	0 1 1			
P4	4 3 3	0 0 2	4 3 1			

- Implementation of safety algorithm to check the resultant resource allocation state is safe or not

Initially work= 2 3 0

Initially Finish	Process	Need _i <= Work	Updated Finish	Safe sequence
F	P0	743<=230(F)		
F	P1	020<=230 (T) work= 230+302=532	T	P1
F	P2	600<=532(F)		
F	P3	011<=532(T) work=532+211=743	T	P1,P3
F	P4	431<=743(T) Work=743+002=745	T	P1,P3,P4
F	P0	743<=745(T) work=745+010=755	T	P1,P3,P4,P0
F	P2	600<=745(T) work=755+302=10 5 7	T	P1,P3,P4,P0,P2

As the safe sequence P1,P3,P4,P0,P2 exists here, so the resultant resource allocation state is safe. Hence the request made by P1 can be granted.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by
Mr. Rakesh Kumar
Assistant Professor

Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

27 Deadlock detection and recovery	1
27.1 Deadlock Detection	1
27.1.1 Single instance of each resource type	1
27.1.2 Multiple instances of each resource type	2
27.2 Deadlock Recovery	4
27.2.1 Process Termination	4
27.2.2 Resource Preemption	5

Lecture 27

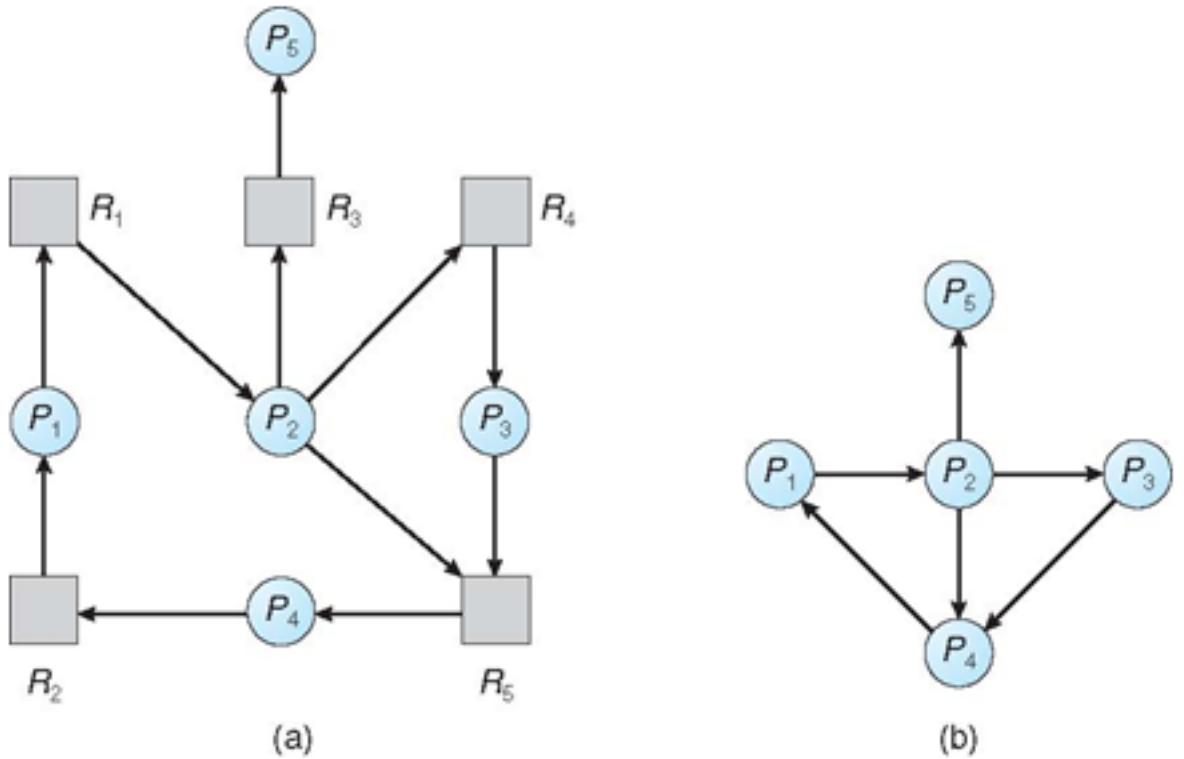
Deadlock detection and recovery

27.1 Deadlock 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 or not.
 - An algorithm to recover from the deadlock.
- There are two deadlock detection approach:
 - Single instance of each resource type
 - Multiple instances of each resource type

27.1.1 Single instance of each resource type

- Use a wait-for graph to detect deadlock.
- Wait for graph is created from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges. Here nodes represent processes. The edge $P_i \rightarrow P_j$, implies P_i is waiting for P_j .
- Periodically an algorithm is invoked that searches for a cycle in the wait for graph. If there is a cycle present, that implies there exists a deadlock.
- The algorithm to detect a cycle in the graph requires an order of n^2 operations, where n is the number of vertices in the graph.



Resource-Allocation Graph Corresponding wait-for graph

Figure 27.1: Resource allocation graph and its corresponding Wait for graph

27.1.2 Multiple instances of each resource type

- **Data Structures used** Let $m =$ number of resources and $n =$ number of processes
 - **Available:** A vector of length m indicates the number of available resources of each type.
 - **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
 - **Request:** An $n \times m$ matrix indicates the current request of each process. If Request $[i][j] = k$, then process P_i is requesting k more instances of resource type R_j .
- **Detection Algorithm**
 1. Let Work and Finish be vectors of length m and n , respectively. Initialize:
Work = Available
if Allocation $_i \neq 0$, then Finish[i] = false; otherwise, Finish[i] = true.
 2. Find an i such that both:
 - (a) Finish [i] = false

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

If no such i exists, go to step 4.

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

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

go to step 2.

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

- **Example:**

Suppose a system with five processes P0 through P4 and three resource types A, B, and C. Resource type A has seven instances, resource type B has two instances, and resource type C has six instances. At time T0 the resource allocation state is represented as follows:

Process	Allocation	Request	Available						
			A	B	C	A	B	C	
P0	0 1 0	0 0 0	0	0	0				
P1	2 0 0	2 0 2							
P2	3 0 3	0 0 0							
P3	2 1 1	1 0 0							
P4	0 0 2	0 0 2							

- **Implementation of deadlock detection algorithm**

Initially work= 0 0 0

Initially Finish	Process	$\text{Request}_i \leq \text{Work}$	Updated Work	Updated Finish	Process sequence
F	P0	$000 \leq 000(T)$		T	P0
		work= $000+010=010$			
F	P1	$202 \leq 010 (F)$			
F	P2	$000 \leq 010(T)$			
		work= $010+303=313$		T	P0,P2
F	P3	$100 \leq 313(T)$			
		work= $313+211=524$		T	P0,P2,P3
F	P4	$002 \leq 524(T)$			
		Work= $524+002=525$		T	P0,P2,P3,P4
F	P1	$202 \leq 525(T)$			
		work= $525+200=726$		T	P0,P2,P3,P4,P1

As sequence P0, P2, P3, P4, P1 will result in $\text{Finish}[i] = \text{true}$ for all i , so there is no deadlock here.

Suppose now process P 2 makes one additional request for an instance of type C.

The Request matrix is modified as follows:

Process	Request
	A B C
P0	0 0 0
P1	2 0 2
P2	0 0 1
P3	1 0 0
P4	0 0 2

Here again using the deadlock detection algorithm, it can be checked that P0 can only complete its task and the number of available resources is not sufficient to fulfill the requests of other processes. So Deadlock exists, consisting of processes P1, P2, P3, and P4.

27.2 Deadlock Recovery

When a detection algorithm determines that a deadlock exists, the following alternatives are used to recover from deadlock:

- Inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Let the system recover from the deadlock automatically. There are two options for breaking a deadlock.
 - Abort one or more processes to break the circular wait.
 - Preempt some resources from one or more of the deadlocked processes.

27.2.1 Process Termination

- **Abort all deadlocked processes:** This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.
- In case of partial termination method, a deadlocked process is selected for terminated based on the following factors:
 - What the priority of the process is
 - How long the process has computed and how much longer the process will compute before completing its designated task

- How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
- How many more resources the process needs in order to complete
- How many processes will need to be terminated
- Whether the process is interactive or batch

27.2.2 Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

- **Selecting a victim:** Which resources and which processes are to be preempted? The order of preemption must be determined to minimize the cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.
- **Rollback:** If we preempt a resource from a process, what should be done with that process? The process is rolled back to some safe state and restart it from that state. So it requires the system to keep more information about the state of all the running processes. As, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it.
- **Starvation:** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process? In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task. So it should be ensured that a process can be picked as a victim only a finite number of times. The most common solution is to include the number of rollbacks in the cost factor.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by
Mr. Rakesh Kumar
Assistant Professor

Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

38 Understanding Linux Kernel	1
38.1 Introduction	1
38.1.1 Linux Versus Other Unix-Like Kernels	1
38.2 Basic Operating System Concepts	3
38.2.1 Multiuser Systems	3
38.2.2 Users and Groups	3
38.2.3 Processes	4
38.2.4 Kernel Architecture	4
38.3 An Overview of the Unix Filesystem	5
38.3.1 Files	5
38.3.2 Hard and Soft Links	5
38.3.3 File Types	6
38.3.4 File Descriptor and Inode	7
38.3.5 Access Rights and File Mode	7
38.4 An Overview of Unix Kernels	7
38.4.1 The Process/Kernel Model	8
38.4.2 Process Implementation	8
38.4.3 Reentrant Kernels	8
38.4.4 Process Address Space	9
38.4.5 Synchronization and Critical Regions	9
38.4.6 Signals and Interprocess Communication	10
38.4.7 Process Management	11
38.4.8 Memory Management	11
38.4.9 Device Drivers	12

Lecture 38

Understanding Linux Kernel

38.1 Introduction

- Linux is a member of the large family of Unix-like operating systems.
- Linux was initially developed by Linus Torvalds in 1991 as an operating system for IBM-compatible personal computers based on the Intel 80386 microprocessor.
- It isn't a commercial operating system: its source code under the GNU General Public License (GPL) is open and available to anyone to study.
- Linux is a true Unix kernel, although it is not a full Unix operating system because it does not include all the Unix applications, such as filesystem utilities, windowing systems and graphical desktops, system administrator commands, text editors, compilers, and so on. However, because most of these programs are freely available under the GPL, they can be installed in every Linux-based system.

38.1.1 Linux Versus Other Unix-Like Kernels

The following list describes how Linux competes against some well-known commercial Unix kernels:

- Monolithic kernel: It is a large, complex do-it-yourself program, composed of several logically different components. In this, it is quite conventional; most commercial Unix variants are monolithic (Apple Mac OS X and the GNU Hurd are exception).
- Compiled and statically linked traditional Unix kernels: Most modern kernels can dynamically load and unload some portions of the kernel code (typically, device drivers), which are usually called modules. Linux's support for modules is very good, because it is able to automatically load and unload modules on demand (SVR4.2 and Solaris kernels have same feature). Kernel threading: Linux uses kernel threads in a very limited way to execute a few kernel functions periodically (However, Solaris and SVR4.2/MP, are organized as a set of kernel threads).

- Multithreaded application support: Linux defines its own version of lightweight processes, which is different from the types used on other systems such as SVR4 and Solaris. While all the commercial Unix variants of LWP are based on kernel threads, Linux regards lightweight processes as the basic execution context and handles them via the nonstandard clone() system call.
- Preemptive kernel: When compiled with the "Preemptible Kernel" option, Linux 2.6 can arbitrarily interleave execution flows while they are in privileged mode. Solaris and Mach 3.0 , are fully preemptive kernels while SVR4.2/MP introduces some fixed preemption points as a method to get limited preemption capability.
- Multiprocessor support: Linux 2.6 supports symmetric multiprocessing (SMP) where the system can use multiple processors and each processor can handle any task there is no discrimination among them.
- Filesystem: Linux filesystems come in many flavors. Ext2 filesystem if you don't have specific needs. ou might switch to Ext3 if you want to avoid lengthy filesystem checks after a system crash. It has object-oriented Virtual File System technology which makes porting a foreign filesystem to Linux is generally easier than porting to other kernels.
- STREAMS: Linux has no analog to the STREAMS I/O subsystem introduced in SVR4. Linux offers the following advantages over its commercial competitors:
 - Linux is cost-free.
 - Linux is fully customizable in all its components. - You can customize the kernel by selecting only the features really needed.
 - Linux runs on low-end, inexpensive hardware platforms
 - Linux is powerful. – Linux systems are very fast. Its main goal is its efficiency.
 - Linux developers are excellent programmers. – Linux systems are very stable; they have a very low failure rate and system maintenance time.
 - The Linux kernel can be very small and compact 2 – It is possible to fit a kernel image, including a few system programs, on just one 1.44 MB floppy disk.
 - Linux is highly compatible with many common operating systems.
 - Linux lets you directly mount filesystems for all versions of MS-DOS and Microsoft Windows , SVR4, OS/2 , Mac OS X , Solaris , SunOS , NEXTSTEP , many BSD variants, and so on.
 - By using suitable libraries, Linux systems are even able to directly run programs written for other operating systems.

- Linux is well supported
 - it may be a lot easier to get patches and updates for Linux than for any proprietary operating system.
 - Drivers for Linux are usually available a few weeks after new hardware products have been introduced on the market.

38.2 Basic Operating System Concepts

The most important program in the OS is called the kernel. It is loaded into RAM when the system boots and contains many critical procedures that are needed for the system to operate.

- The other programs can provide a wide variety of interactive experiences for the user as well as doing all the jobs the user bought the computer.
- However, the kernel provides key facilities to everything else on the system and determines many of the characteristics of higher software. Hence, we often use the term "operating system" as a synonym for "kernel."
- Unix-like operating system hides all low-level details concerning the physical organization of the computer from applications run by the user unlike MS-DOS.
- The hardware introduces at least two different execution modes for the CPU: a nonprivileged mode for user programs and a privileged mode for the kernel known as User Mode and Kernel Mode , respectively.

38.2.1 Multiuser Systems

- A multiuser system is a computer that is able to concurrently and independently execute several applications belonging to two or more users. Concurrently means that applications can be active at the same time and contend for the various resources such as CPU, memory, hard disks, and so on.
- Multiuser operating systems must include several features:
 - An authentication mechanism for verifying the user's identity.
 - A protection mechanism against buggy user programs that could block other applications running in the system.
 - A protection mechanism against malicious user programs that could interfere with or spy on the activity of other users.
 - An accounting mechanism that limits the amount of resource units assigned to each user

38.2.2 Users and Groups

- In a multiuser system, each user has a private space on the machine; typically, he owns some quota of the disk.

- All users are identified by a unique number called the User ID, or UID.
- To selectively share material with other users, each user is a member of one or more user groups , which are identified by a unique number called a user group ID.
- Any Unix-like operating system has a special user called root or superuser . The system administrator must log in as root to handle user accounts, perform maintenance tasks such as system backups and program upgrades, and so on.

38.2.3 Processes

- All operating systems use one fundamental abstraction: the process. A process can be defined either as "an instance of a program in execution" or as the "execution context" of a running program.
- Multiuser systems must enforce a multiprogramming or multiprocessing execution environment.
- Processes of a multiuser system must be preemptable; the operating system tracks how long each process holds the CPU and periodically.
- Unix-like operating systems adopt a process/kernel model. Each process has the illusion that it's the only process on the machine, and it has exclusive access to the operating system services.

38.2.4 Kernel Architecture

- Most Unix kernels are monolithic: each kernel layer is integrated into the whole kernel program and runs in Kernel Mode on behalf of the current process. 4
- In contrast, microkernel operating systems demand a very small set of functions from the kernel and several system processes that run on top of the microkernel implement other operating system-layer functions, like memory allocators, device drivers, and system call handlers.
- Microkernels are generally slower than monolithic ones, because the explicit message passing between the different layers of the operating system has a cost.
- However, Microkernels force the system programmers to adopt a modularized approach, because each operating system layer is a relatively independent program that must interact with the other layers through well-defined and clean software interfaces.
- To achieve many of the theoretical advantages of microkernels without introducing performance penalties, the Linux kernel offers modules. A module is an object file whose code can be linked to (and unlinked from) the kernel at runtime.

- The main advantages of using modules include: modularized approach, Platform independence, Frugal main memory usage and No performance penalty.

38.3 An Overview of the Unix Filesystem

The Unix operating system design is centered on its filesystem, which has several interesting characteristics.

38.3.1 Files

- A Unix file is an information container structured as a sequence of bytes; the kernel does not interpret the contents of a file.
- From the user's point of view, files are organized in a tree-structured namespace, as shown in Fig. 38.1.
- All the nodes of the tree, except the leaves, denote directory names. A directory node contains information about the files and directories just beneath it.
- A file or directory name consists of a sequence of arbitrary ASCII characters.
- The directory corresponding to the root of the tree is called the root directory. By convention, its name is a slash (/).
- To identify a specific file, the process uses a pathname, which consists of slashes alternating with a sequence of directory names that lead to the file.
- If the first item in the pathname is a slash, the pathname is said to be absolute, because its starting point is the root directory. Otherwise, if the first item is a directory name or filename, the pathname is said to be relative, because its starting point is the process's current directory.

38.3.2 Hard and Soft Links

- A filename included in a directory is called a file hard link, or more simply, a link. The same file may have several links included in the same directory or in different ones, so it may have several filenames.
- The Unix command: `ln p1 p2` is used to create a new hard link that has the pathname p2 for a file identified by the pathname p1.
- Hard links have two limitations:
 - It is not possible to create hard links for directories.
 - Links can be created only among files included in the same filesystem.

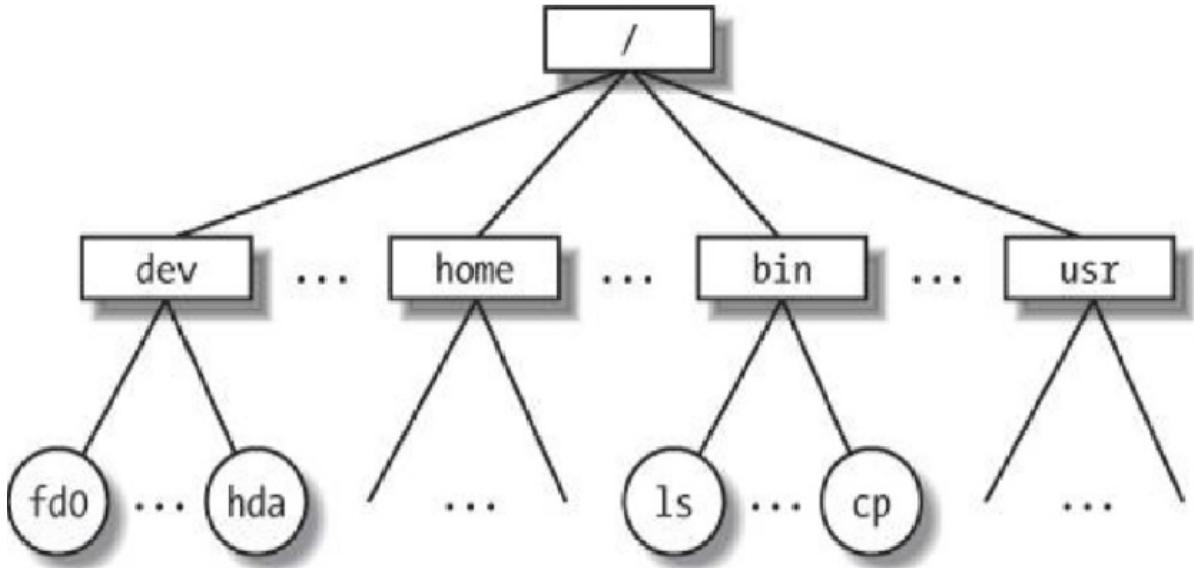


Figure 38.1: An example of a directory tree

- To overcome these limitations, soft links (also called symbolic links) were introduced.
- Symbolic links are short files that contain an arbitrary pathname of another file. The pathname may refer to any file or directory located in any filesystem; it may even refer to a non-existent file.
- The Unix command: `ln -s p1 p2` creates a new soft link with pathname p2 that refers to pathname p1.

38.3.3 File Types

Unix files may have one of the following types:

- Regular file
- Directory
- Symbolic link
- Block-oriented device file
- Character-oriented device file
- Pipe and named pipe (also called FIFO)
- Socket

38.3.4 File Descriptor and Inode

- Unix makes a clear distinction between the contents of a file and the information about a file.
- All information needed by the filesystem to handle a file is included in a data structure called an inode. Each file has its own inode, which the filesystem uses to identify the file.
- Inode of a file provide at least the following attributes
 - File type (see the previous section)
 - Number of hard links associated with the file
 - File length in bytes
 - Device ID (i.e., an identifier of the device containing the file)
 - Inode number that identifies the file within the filesystem
 - UID of the file owner
 - User group ID of the file
 - Several timestamps that specify the inode status change time, the last access time, and the last modify time
 - Access rights and file mode

38.3.5 Access Rights and File Mode

- The potential users of a file fall into three classes:
 - The user who is the owner of the file
 - The users who belong to the same group as the file, not including the owner
 - All remaining users (others)
- There are three types of access rights – read, write, and execute for each of these three classes.
- Three additional flags, called `suid` (Set User ID), `sgid` (Set Group ID), and `sticky`, define the file mode having the following meaning:
 - `suid`: If the executable file has the `suid` flag set, the process gets the UID of the file owner.
 - `sgid`: If the executable file has the `sgid` flag set, the process gets the user group ID of the file.
 - `sticky`: An executable file with the `sticky` flag set corresponds to a request to the kernel to keep the program in memory after its execution terminates

38.4 An Overview of Unix Kernels

Unix kernel implements a set of services and corresponding interfaces. Applications use those interfaces and do not usually interact directly with hardware resources.

38.4.1 The Process/Kernel Model

- A CPU can run in either User Mode or Kernel Mode.
- When a program is executed in User Mode, it cannot directly access the kernel data structures or the kernel programs.
- When an application executes in Kernel Mode, however, access to the kernel data structures and kernel programs possible.
- A program usually executes in User Mode and switches to Kernel Mode only when requesting a service provided by the kernel. When the kernel has satisfied the program's request, it puts the program back in User Mode.
- On a uniprocessor system, only one process is running at a time and it may run either in User or in Kernel Mode.
- Fig. 38.2 illustrates examples of transitions between User and Kernel Mode when a process switch occurs in different scenarios.

38.4.2 Process Implementation

- To let the kernel manage processes, each process is represented by a process descriptor that includes information about the current state of the process.
- When the kernel stops the execution of a process, it saves the current contents of several processor registers in the process descriptor. These include:
 - The program counter (PC) and stack pointer (SP) registers
 - The general purpose registers
 - The floating point registers
 - The processor control registers (Processor StatusWord) containing information about the CPU state
 - The memory management registers used to keep track of the RAM accessed by the process
- When the kernel decides to resume executing a process, it uses the proper process descriptor fields to load the CPU registers.

38.4.3 Reentrant Kernels

- All Unix kernels are reentrant.
- One way to provide reentrancy is to write functions so that they modify only local variables and do not alter global data structures. Such functions are called reentrant

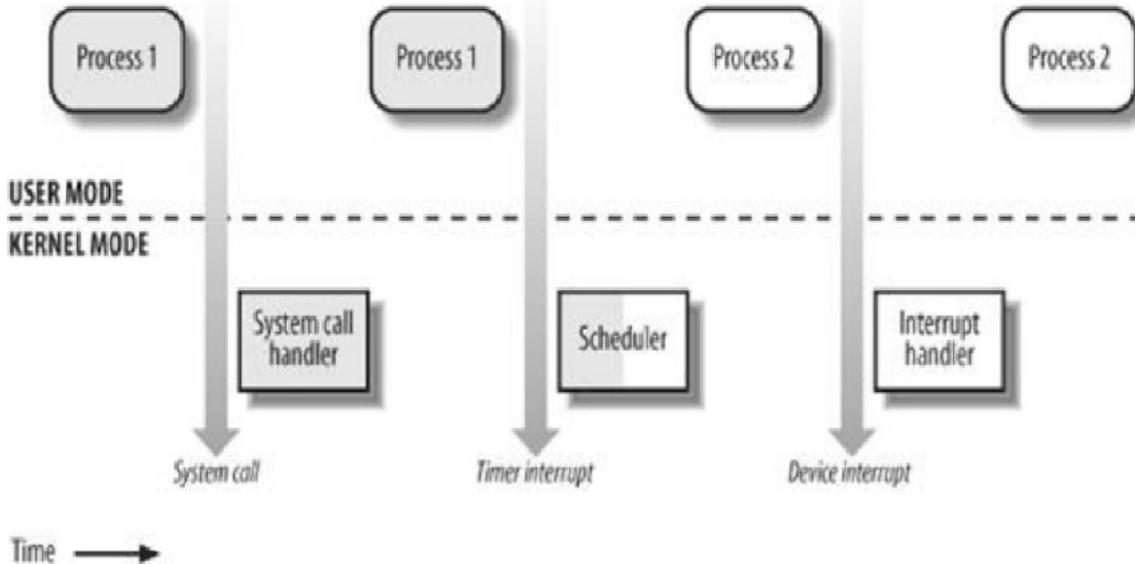


Figure 38.2: Transition Between User and Kernel mode

functions. But a reentrant kernel is not limited only to such reentrant functions, instead the kernel can include nonreentrant functions and use locking mechanisms to ensure that only one process can execute a nonreentrant function at a time.

- A kernel control path denotes the sequence of instructions executed by the kernel to handle a system call, an exception, or an interrupt.

38.4.4 Process Address Space

- Each process runs in its private address space. A process running in User Mode refers to private stack, data, and code areas.
- When running in Kernel Mode, the process addresses the kernel data and code areas and uses another private stack.
- There are times when part of the address space is shared among processes. In some cases, this sharing is explicitly requested by processes; in others, it is done automatically by the kernel to reduce memory usage.

38.4.5 Synchronization and Critical Regions

- Implementing a reentrant kernel requires the use of synchronization.
- If a kernel control path is suspended while acting on a kernel data structure, no other kernel control path should be allowed to act on the same data structure unless it has been reset to a consistent state.

- When the outcome of a computation depends on how two or more processes are scheduled, the code is incorrect. We say that there is a race condition.
- Safe access to a global variable is ensured by using atomic operations. Different ways of achieving process synchronization Unix includes
- Kernel preemption disabling: A synchronization mechanism applicable to preemptive kernels consists of disabling kernel preemption before entering a critical region and reenabling it right after leaving the region.
- Interrupt disabling: Another synchronization mechanism for uniprocessor systems consists of disabling all hardware interrupts before entering a critical region and reenabling them right after leaving it.
- Semaphores: A semaphore is simply a counter associated with a data structure; it is checked by all kernel threads before they try to access the data structure.
- Spin locks: A spin lock is very similar to a semaphore, but it has no process list; when a process finds the lock closed by another process, it "spins" around repeatedly, executing a tight instruction loop until the lock becomes open.
- Avoiding deadlocks: Several operating systems, including Linux, avoid this problem by requesting locks in a predefined order.

38.4.6 Signals and Interprocess Communication

- Unix signals provide a mechanism for notifying processes of system events.
- Each event has its own signal number, which is usually referred to by a symbolic constant such as SIGTERM.
- There are two kinds of system events:
 - Asynchronous notifications: For instance, a user can send the interrupt signal SIGINT to a foreground process by pressing the interrupt keycode (usually Ctrl-C) at the terminal.
 - Synchronous notifications: For instance, the kernel sends the signal SIGSEGV to a process when it accesses a memory location at an invalid address.
- Interprocess communication among processes in User Mode which have been adopted by many Unix kernels are semaphores, message queues, and shared memory.
- The kernel implements these constructs as IPC resources. A process acquires a resource by invoking a `shmget()`, `semget()`, or `msgget()` system call.
- Shared memory provides the fastest way for processes to exchange and share data.

38.4.7 Process Management

- The fork() and exit() system calls are used respectively to create a new process and to terminate it, while an exec()-like system call is invoked to load a new program.
- Naive implementation of the fork() would require both the parent's data and the parent's code to be duplicated and the copies assigned to the child.
- Current kernels that can rely on hardware paging units follow the Copy-On-Write approach, which defers page duplication until the last moment.
 - Zombie processes: A special zombie process state is introduced to represent terminated processes: a process remains in that state until its parent process executes a wait4() system call on it.
 - Process groups and login sessions: In order to execute the command line ls | sort | more a shell that supports process groups, such as bash, creates a new group for the three processes corresponding to ls, sort, and more. In this way, the shell acts on the three processes as if they were a single entity. Each process descriptor includes a field containing the process group ID . Each group of processes may have a group leader, which is the process whose PID coincides with the process group ID. A newly created process is initially inserted into the process group of its parent. Modern Unix kernels also introduce login sessions. a login session contains all processes that are descendants of the process that has started a working session on a specific terminal usually, the first command shell process created for the user.

38.4.8 Memory Management

- It is the most complex activity in a Unix kernel.
- Virtual memory: It acts as a logical layer between the application memory requests and the hardware Memory Management Unit (MMU).
- Virtual memory has many purposes and advantages:
 - Several processes can be executed concurrently.
 - It is possible to run applications whose memory needs are larger than the available physical memory.
 - Processes can execute a program whose code is only partially loaded in memory.
 - Each process is allowed to access a subset of the available physical memory.
 - Processes can share a single memory image of a library or program.
 - Programs can be relocatable that is, they can be placed anywhere in physical memory.
 - Programmers can write machine-independent code, because they do not need to be concerned about physical memory organization.
- Random access memory usage: All Unix operating systems clearly distinguish between two portions of the random access memory (RAM). A few megabytes are

dedicated to storing the kernel image. The remaining portion of RAM is usually handled by the virtual memory system and is used in three possible ways:

- To satisfy kernel requests for buffers, descriptors, and other dynamic kernel data structures
- To satisfy process requests for generic memory areas and for memory mapping of files
- To get better performance from disks and other buffered devices by means of caches.

- Kernel Memory Allocator: The Kernel Memory Allocator (KMA) is a subsystem that tries to satisfy the requests for memory areas from all parts of the system.
- A good KMA should have the following features:
 - It must be fast.
 - It should minimize the amount of wasted memory.
 - It should try to reduce the memory fragmentation problem.
 - It should be able to cooperate with the other memory management subsystems to borrow and release page frames from them.
- Process virtual address space handling: The address space of a process contains all the virtual memory addresses that the process is allowed to reference.
- All recent Unix operating systems adopt a memory allocation strategy called demand paging . With demand paging, a process can start program execution with none of its pages in physical memory.
- Virtual address spaces also allow other efficient strategies, such as the Copy On Write strategy.
- Caching: A good part of the available physical memory is used as cache for hard disks and other block devices.
- This strategy is based on the fact that there is a good chance that new processes will require data read from or written to disk by processes that no longer exist.
- The sync() system call forces disk synchronization by writing all of the "dirty" buffers into disk.

38.4.9 Device Drivers

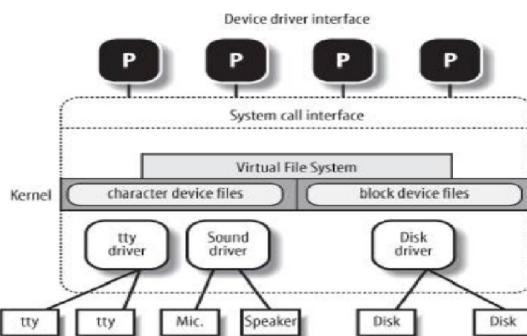


Figure 38.3: Device Driver Interface

- The kernel interacts with I/O devices by means of device drivers .
- Device drivers are included in the kernel and consist of data structures and functions that control one or more devices, such as hard disks, keyboards, mouses, monitors etc.
- Each driver interacts with the remaining part of the kernel (even with other drivers) through a specific interface. Fig 38.3 illustrates how device drivers interface with the rest of the kernel and with the processes.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

39 Process	1
39.1 Introduction	1
39.2 Processes, Lightweight Processes, and Threads	2
39.3 Process Descriptor	3
39.4 Process States	3
39.5 Identifying a Process	4
39.5.1 Thread Groups, tgid and pid	5
39.6 Process Descriptor Handling	5
39.7 Process List	7
39.7.1 Process 0 and Process 1	8
39.8 Process Switch	9
39.8.1 Hardware Context	9
39.8.2 Hardware/software context switch	10
39.9 Creating Processes	10
39.9.1 The clone(), fork(), and vfork() System Calls	11
39.9.2 Key Differences Between fork() and vfork()	12
39.10 Destroying Process	12
39.10.1 The do_group_exit() function	13
39.10.2 The do_exit() function	13

Lecture 39

Process

39.1 Introduction

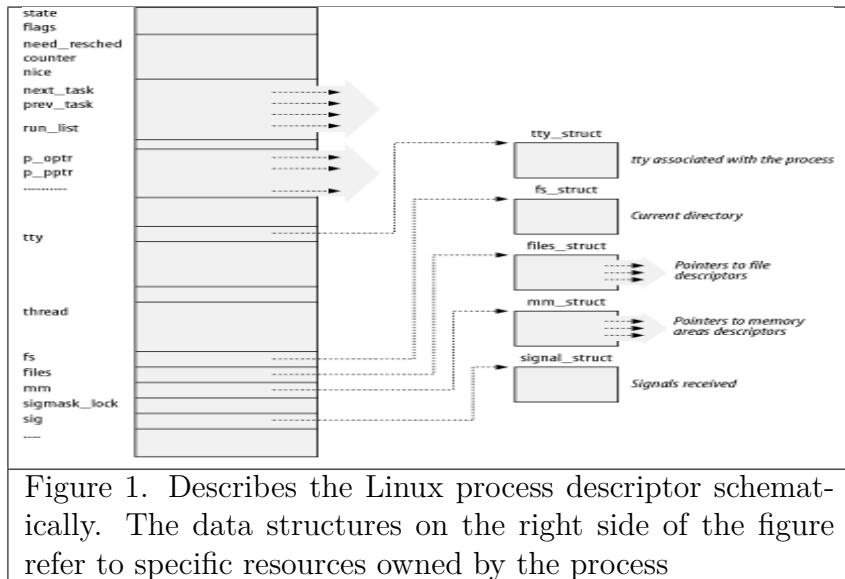
- The process is one of the fundamental abstractions in Unix operating systems, the other fundamental abstraction is files.
- Processes are, however, more than just the executing program code (often called the text section in Unix). They also include a set of resources such as open files and pending signals, internal kernel data, processor state, an address space, one or more threads of execution, and a data section containing global variables.
- Threads of execution, often shortened to threads, are the objects of activity within the process. Each thread includes a unique program counter, process stack, and set of processor registers. The kernel schedules individual threads, not processes.
- In traditional Unix systems, each process consists of one thread. In modern systems, however, multithreaded programs—those that consist of more than one thread—are common.
- Linux has a unique implementation of threads: It does not differentiate between threads and processes. To Linux, a thread is just a special kind of process.
- On modern operating systems, processes provide two virtualizations: a virtualized processor and virtual memory.
- The virtual processor gives the process the illusion that it alone monopolizes the system, despite possibly sharing the processor among dozens of other processes.
- Virtual memory lets the process allocate and manage memory as if it alone owned all the memory in the system. Interestingly, note that threads share the virtual memory abstraction while each receives its own virtualized processor. Two or more processes can exist that are executing the same program.
- In fact, two or more processes can exist that share various resources, such as open files or an address space.

39.2 Processes, Lightweight Processes, and Threads

- From the kernel's point of view, the purpose of a process is to act as an entity to which system resources (CPU time, memory, etc.) are allocated.
- (Traditionally) When a process is created, it is almost identical to its parent.
- It receives a (logical) copy of the parent's address space and executes the same code as the parent
- they have separate copies of the data (stack and heap), so that changes by the child to a memory location are invisible to the parent (and vice versa).
- While earlier Unix kernels employed this simple model, modern Unix systems do not.
- They support multithreaded applications user programs having many relatively independent execution flows sharing a large portion of the application data structures.
- Most multithreaded applications are written using standard sets of library functions called pthread (POSIX thread).
- Linux uses lightweight processes to offer better support for multithreaded applications.
- Basically, two lightweight processes may share some resources, like the address space, the open files, and so on. Whenever one of them modifies a shared resource, the other immediately sees the change.
- A straightforward way to implement multithreaded applications is to associate a lightweight process with each thread.
- Each thread can be scheduled independently by the kernel so that one may sleep while another remains runnable.
- Examples of POSIX-compliant pthread libraries that use Linux's lightweight processes are Linux Threads, Native POSIX Thread Library (NPTL), and IBM's Next Generation Posix Threading Package (NGPT).
- multithreaded applications represented by "thread groups"
- basically a set of lightweight processes that act as a whole with regards to some system calls such as `getpid()`, `kill()`, and `_exit()`.

39.3 Process Descriptor

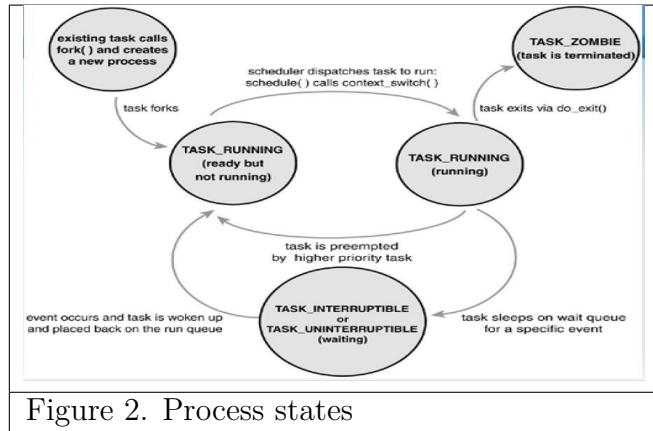
- To manage processes, the kernel must have a clear picture of what each process is doing.
- the process's priority, whether it is running on a CPU or blocked on an event, what address space has been assigned to it, which files it is allowed to address, etc
- This is the role of the process descriptor a task_struct type structure whose fields contain all the information related to a single process.
- The process descriptor is rather complex.
- In addition to a large number of fields containing process attributes, the process descriptor contains several pointers to other data structures that, in turn, contain pointers to other structures.



39.4 Process States

- **TASK_RUNNING** : The process is either executing on a CPU or waiting to be executed
- **TASK_INTERRUPTIBLE**: The process is suspended (sleeping) until some condition becomes true.
- **TASK_UNINTERRUPTIBLE**: Like **TASK_INTERRUPTIBLE**, except that delivering a signal to the sleeping process leaves its state unchanged.
- **TASK_STOPPED** : Process execution has been stopped; the process enters this state after receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal.

- **TASK_TRACED**: Process execution has been stopped by a debugger. When a process is being monitored by another (such as when a debugger executes a `ptrace()` system call to monitor a test program), each signal may put the process in the **TASK_TRACED** state.
- **EXIT_ZOMBIE** :Process execution is terminated, but the parent process has not yet issued a `wait4()` or `waitpid()` system call to return information about the dead process.
- **EXIT_DEAD** :The final state: the process is being removed by the system because the parent process has just issued a `wait4()` or `waitpid()` system call for it. Changing its state from **EXIT_ZOMBIE** to **EXIT_DEAD** avoids race conditions due to other threads of execution that execute `wait()`-like calls on the same process.



39.5 Identifying a Process

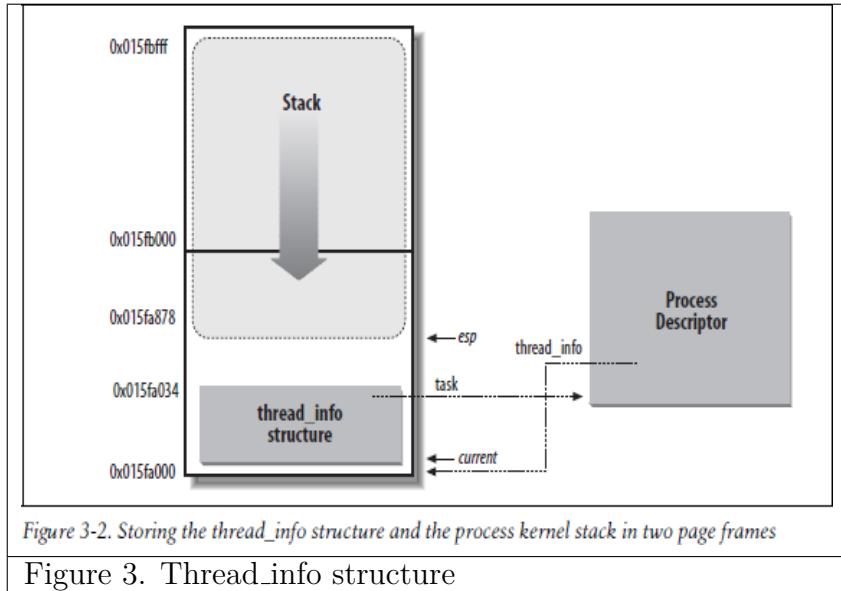
- As a general rule, each execution context that can be independently scheduled must have its own process descriptor;
- therefore, even lightweight processes, which share a large portion of their kernel data structures, have their own `task_struct` structures.
- The 32-bit addresses of the `task_struct` structure are referred to as process descriptor pointers. Most of the references to processes that the kernel makes are through process descriptor pointers.
- On the other hand, Unix-like operating systems allow users to identify processes by means of a number called the Process ID (or PID), which is stored in the `pid` field of the process descriptor.
- PIDs are numbered sequentially: the PID of a newly created process is normally the PID of the previously created process increased by one.

39.5.1 Thread Groups, tgid and pid

- Linux associates a different PID with each process or lightweight process in the system. This approach allows the maximum flexibility, because every execution context in the system can be uniquely identified.
- On the other hand, Unix programmers expect threads in the same group to have a common PID. POSIX 1003.1c standard states that all threads of a multithreaded application must have the same PID.
- For instance, it should be possible to send a signal specifying a PID that affects all threads in the group.
- Linux makes use of thread groups. The identifier shared by the threads is the PID of the thread group leader, that is, the PID of the first lightweight process in the group; it is stored in the tgid field of the process descriptors.
- The getpid() system call returns the value of tgid relative to the current process instead of the value of pid, so all the threads of a multithreaded application share the same identifier.
- Most processes belong to a thread group consisting of a single member; as thread group leaders, they have the tgid field equal to the pid field, thus the getpid() system call works as usual for this kind of process.

39.6 Process Descriptor Handling

- Process descriptors are stored in dynamic memory rather than in the memory area permanently assigned to the kernel. For each process, Linux packs two different data structures in a single per-process memory area: a small data structure linked to the process descriptor, namely the thread_infostructure, and the Kernel Mode process stack.
- The length of this memory area is usually 8,192 bytes (two page frames).
- For reasons of efficiency (avoiding fragmentation) the kernel can be configured at compilation time so that the memory area including stack and thread_info structure spans a single page frame (4,096 bytes).
- The figure shows that the thread_info structure and the task_struct structure are mutually linked by means of the fields task and thread_info, respectively.
- The esp register is the CPU stack pointer, which is used to address the stack's top location.



- On 80x86 systems, the stack starts at the end and grows toward the beginning of the memory area.
- Right after switching from User Mode to Kernel Mode, the kernel stack of a process is always empty, and therefore the `esp` register points to the byte immediately following the stack.
- The value of the `esp` is decreased as soon as data is written into the stack. Because the `thread_info` structure is 52 bytes long, the kernel stack can expand up to 8,140 bytes.

Kernel thread

- They are same as user space threads in many aspects, but one of the biggest difference is that they exist in the kernel space and execute in a privileged mode and have full access to the kernel data structures.
- These are basically used to implement background tasks inside the kernel. The task can be handling of asynchronous events or waiting for an event to occur.
- Device drivers utilize the services of kernel threads to handle such tasks. For example, the `ksoftirqd/0` thread is used to implement the Soft IRQs in kernel. The `khubd` kernel thread monitors the usb hubs and helps in configuring usb devices during hot-plugging.

Identifying the current process

- The kernel can easily obtain the address of the `thread_info` structure of the process currently running on a CPU from the value of the `esp` register.

- Advantage of storing the process descriptor with the stack emerges on multiprocessor systems: the correct current process for each hardware processor can be derived just by checking the stack
- Earlier versions of Linux forced to introduce a global static variable called `current` to identify the process descriptor of the running process. On multiprocessor systems, it was necessary to define `current` as an array—one element for each available CPU
- List of Data Structure
- Linux kernel defines the `list_head` data structure, whose only fields `next` and `prev` represent the forward and back pointers of a generic doubly linked list element, respectively.
- It is important to note, however, that the pointers in a `list_head` field store the addresses of other `list_head` fields rather than the addresses of the whole data structures in which the `list_head` structure is included; see Figure .
- A new list is created by using the
- `LIST_HEAD(list_name)` macro.
- It declares a new variable named `list_name` of type `list_head`, which is dummy first element that acts as a placeholder for the head of the new list, and initializes the `prev` and `next` fields of the `list_head` data structure so as to point to the `list_name` variable itself.

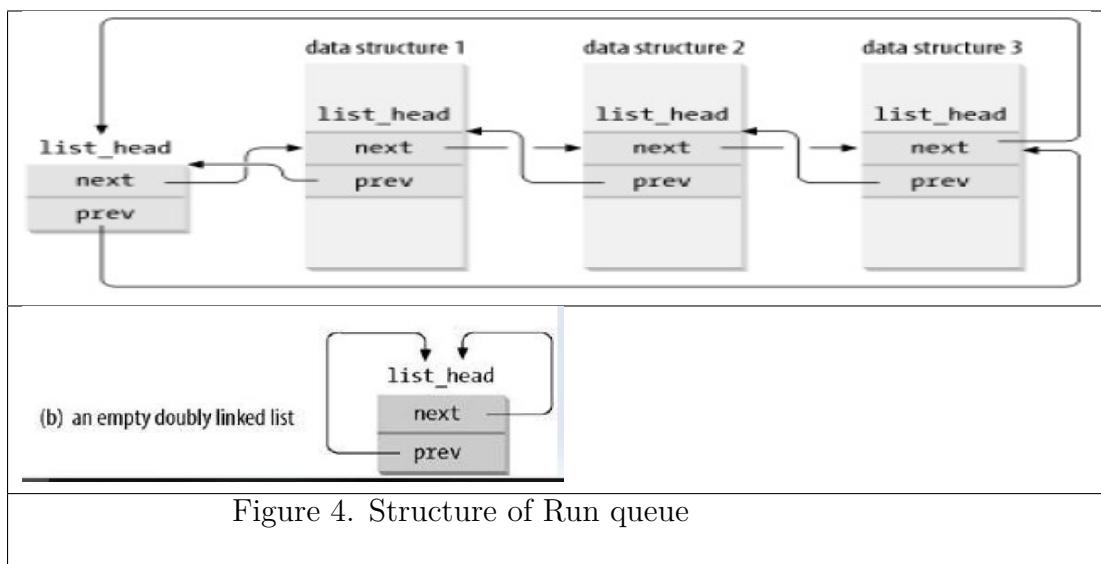


Figure 4. Structure of Run queue

39.7 Process List

- The first example of a doubly linked list we will examine is the *process list*, a list that links together all existing process descriptors.

- The head of the process list is the init_task task_struct descriptor; it is the process descriptor of the so-called *process 0* or *swapper*. [1.27](#)in The tasks->prev field of init_task points to the tasks field of the process descriptor inserted last in the list.
- The SET_LINKS and REMOVE_LINKSmacros are used to insert and to remove a process descriptor in the process list, respectively.

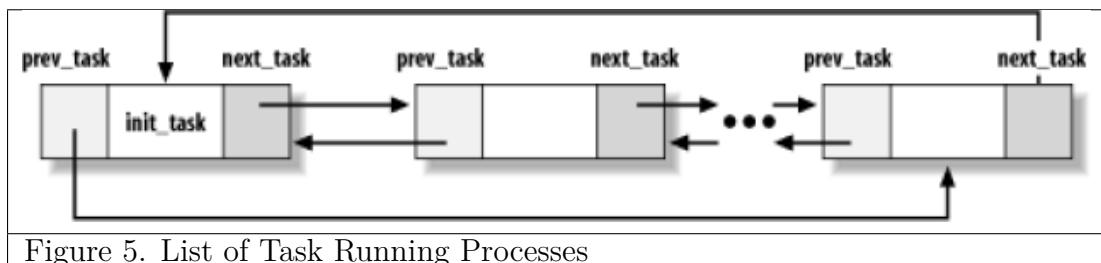


Figure 5. List of Task Running Processes

- When looking for a new process to run on a CPU, the kernel has to consider only the runnable processes.
- Earlier Linux versions put all runnable processes in the same list called runqueue. The earlier schedulers were compelled to scan the whole list in order to select the "best" runnable process, an O(n) operation.
- Linux 2.6 implements the runqueue differently. The aim is to allow the scheduler to select the best runnable process in constant time O(1), independently of the number of runnable processes.
- Trick consist of splitting the runqueue in many lists of runnable processes, one list per process priority.
- Each task_struct descriptor includes a run_list field of type list_head. If the process priority is equal to k (a value ranging between 0 and 139), the run_list field links the process descriptor into the list of runnable processes having priority k .
- Furthermore, on a multiprocessor system, each CPU has its own runqueue, that is, its own set of lists of processes.
- This is a classic example of making a data structures more complex to improve performance: to make scheduler operations more efficient, the runqueue list has been split into 140 different lists!

39.7.1 Process 0 and Process 1

- In Unix-like operating system, every process except process 0 (the swapper) is created when another process executes the fork() System call. The process that invoked fork is the parent process and the newly created process is the child process. Every process (except process 0) has one parent process, but can have many child processes.

- The operating system kernel identifies each process by its process identifier. Process 0 is a special process that is created when the system boots; after forking a child process (process 1), process 0 becomes the swapper process (sometimes also known as the Idle task). Process 1, known as init, is the ancestor of every other process in the system

Process Resource Limits

- Each process has an associated set of resource limits , which specify the amount of system resources it can use.
- These limits keep a user from overwhelming the system (its CPU, disk space, and so on).
- The resource limits for the current process are stored in the current->rlim field, that is, in a field of the process's signal descriptor. The field is an array of elements of type struct rlimit, one for each resource limit:

```
struct rlimit {
    unsigned long rlim_cur;
    unsigned long rlim_max;
};
```

39.8 Process Switch

- To control the execution of processes, the kernel must be able to suspend the execution of the process running on the CPU and resume the execution of some other process previously suspended.
- This activity goes variously by the names *process switch*, *task switch*, or *context switch*.

39.8.1 Hardware Context

- While each process can have its own address space, all processes have to share the CPU registers.
- So before resuming the execution of a process, the kernel must ensure that each such register is loaded with the value it had when the process was suspended.
- The set of data that must be loaded into the registers before the process resumes its execution on the CPU is called the hardware context.
- The hardware context is a subset of the process execution context, which includes all information needed for the process execution.

- The 80x86 architecture includes a specific segment type called the Task State Segment (TSS), to store hardware contexts.
- In Linux,
 - a part of the hardware context of a process is stored in the process descriptor,
 - while the remaining part is saved in the Kernel Mode stack.

39.8.2 Hardware/software context switch

- Process switching occurs only in Kernel Mode. The contents of all registers used by a process in User Mode have already been saved on the Kernel Mode stack before performing process switching.
- Essentially, every process switch consists of two steps:
 - Switching to install a new address space.
 - Switching the Kernel Mode stack and the hardware context, which provides all the information needed by the kernel to execute the new process, including the CPU registers. This is performed by the `switch_to` macro. It is one of the most hardware-dependent routines of the kernel.

39.9 Creating Processes

- Unix operating systems rely heavily on process creation to satisfy user requests. For example, the shell creates a new process that executes another copy of the shell whenever the user enters a command.
- Traditional Unix systems treat all processes in the same way: resources owned by the parent process are duplicated in the child process. This approach makes process creation very slow and inefficient.
- In many cases, the child issues an immediate `execve()` and wipes out the address space that was so carefully copied.
- Modern Unix kernels solve this problem by introducing three different mechanisms:
 - The Copy On Write technique allows both the parent and the child to read the same physical pages. Whenever either one tries to write on a physical page, the kernel copies its contents into a new physical page that is assigned to the writing process.
 - Lightweight processes allow both the parent and the child to share many per-process kernel data structures, such as the paging tables, the open file tables, and the signal dispositions.

- The `vfork()` system call creates a process that shares the memory address space of its parent. To prevent the parent from overwriting data needed by the child, the parent’s execution is blocked until the child exits or executes a new program.

39.9.1 The `clone()`, `fork()`, and `vfork()` System Calls

- Lightweight processes are created in Linux by using a function named `clone()`, which uses the following parameters:
 - `fn`: Specifies a function to be executed by the new process; when the function returns, the child terminates. The function returns an integer, which represents the exit code for the child process. Points to data passed to the `fn()` function.
 - `flags`: The low byte specifies the signal number to be sent to the parent process when the child terminates; the `SIGCHLD` signal is generally selected. The remaining three bytes encode a group of clone flags.
 - `child_stack`: Specifies the User Mode stack pointer to be assigned to the `esp` register of the child process. The invoking process (the parent) should always allocate a new stack for the child.
 - `tls`: Specifies the address of a data structure that defines a Thread Local Storage segment for the new lightweight process.
 - `ptid`: Specifies the address of a User Mode variable of the parent process that will hold the PID of the new lightweight process.
 - `ctid`: Specifies the address of a User Mode variable of the new lightweight process that will hold the PID of such process.

Linux implements `fork()` and `vfork()` with `clone()`

- The `fork()`, `vfork()`, and `clone()` library calls all invoke the `clone()` system call with the requisite flags. The `clone()` system call, in turn, calls `do_fork()`.
- The traditional `fork()` system call is implemented by Linux as a `clone()` system call whose flags parameter specifies both a `SIGCHLD` signal and all the clone flags cleared, and whose `child_stack` parameter is the current parent stack pointer.
- Therefore, the parent and child temporarily share the same User Mode stack. But thanks to the Copy On Write mechanism, they usually get separate copies of the User Mode stack as soon as one tries to change the stack.

- The `vfork()` system call, is implemented by Linux as a `clone()` system call whose `flags` parameter specifies both a `SIGCHLD` signal and the flags `CLONE_VM` and `CLONE_VFORK`, and whose `child_stack` parameter is equal to the current parent stack pointer.

39.9.2 Key Differences Between `fork()` and `vfork()`

- The primary difference between `fork` and `vfork` is that the child process created by the `fork()` has a separate memory space from the parent process. However, the child process created by the `vfork()` system call shares the same address space of its parent process.
- The child process created using `fork` execute simultaneously with the parent process. On the other hand, child process created using `vfork` suspend the execution of parent process till its execution is completed.
- As the memory space of parent and child process is separate modification done by any of the processes does not affect other's pages. However, as the parent and child process shares the same memory address modification done by any process reflects in the address space.
- The system call `fork()` uses copy-on-write as an alternative, which let child and parent process share the same address space until any one of them modifies the pages. On the other hand, the `vfork` does not use copy-on-write

39.10 Destroying Process

- The usual way for a process to terminate is to invoke the `exit()` library function, which releases the resources allocated by the C library, executes each function registered by the programmer, and ends up invoking a system call that evicts the process from the system.
- Alternatively, the kernel may force a whole thread group to die. This typically occurs when a process in the group has received a signal that it cannot handle or ignore or when an unrecoverable CPU exception has been raised in Kernel Mode while the kernel was running on behalf of the process
- In Linux 2.6 there are two system calls that terminate a User Mode application:
- The `exit_group()` system call, which terminates a full thread group, that is, a whole multithreaded application. The main kernel function that implements this system call is called `do_group_exit()`. This is the system call that should be invoked by the `exit()` C library function.

- The `_exit()` system call, which terminates a single process, regardless of any other process in the thread group of the victim. The main kernel function that implements this system call is called `do_exit()`. This is the system call invoked, for instance, by the `pthread_exit()` function of the LinuxThreads library.

39.10.1 The `do_group_exit()` function

- The `do_group_exit()` function kills all processes belonging to the thread group of current. It receives as a parameter the process termination code, which is either a value specified in the `exit_group()` system call (normal termination) or an error code supplied by the kernel (abnormal termination).

39.10.2 The `do_exit()` function

- All process terminations are handled by the `do_exit()` function, which removes most references to the terminating process from kernel data structures. The `do_exit()` function receives as a parameter the process termination code.



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA

LECTURE NOTES

Design of Operating System (CSE 4049)

Prepared by

Mr. Rakesh Kumar
Assistant Professor

**Faculty of Engineering and Technology (ITER)
SIKSHA ‘O’ ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA**

Declaration

This lecture note is prepared not to replace the prescribed text book(s). It is prepared to provide a quick reference to the study material and make the students flexible to go around the course in a systematic and elegant way. Students may use this lecture note for a brief and complete coverage about the basic stuffs of the course.

**Mr. Rakesh Kumar
Assistant Professor, CSE**

Contents

40 CPU Scheduling in Linux	2
40.1 Scheduling Policy	2
40.2 System calls related to scheduling	3
40.3 Scheduling Algorithm	4
40.4 Scheduling of Conventional Processes	5
40.5 Scheduling of Real-Time Processes	7
40.6 Functions Used by the Scheduler	8

Lecture 40

CPU Scheduling in Linux

The chapter consists of three parts:

- The section “Scheduling Policy” introduces the choices made by Linux in the abstract to schedule processes.
- The section “The Scheduling Algorithm” discusses the data structures used to implement scheduling and the corresponding algorithm.
- Finally, the section “System Calls Related to Scheduling” describes the system calls that affect process scheduling.

40.1 Scheduling Policy

- The scheduling algorithm of traditional Unix operating systems must fulfill several conflicting objectives: fast process response time, good throughput for background jobs, avoidance of process starvation, reconciliation of the needs of low- and high priority processes, and so on.
- Linux scheduling is based on the time sharing technique.
- The set of rules used to determine when and how to select a new process to run is called scheduling policy.
- The scheduling policy is based on ranking processes according to their priority.
- In Linux, process priority is dynamic:
 - The scheduler keeps track of what processes are doing and adjusts their priorities periodically.
 - Processes that have been denied the use of a CPU for a long time interval are boosted by dynamically increasing their priority.
 - Correspondingly, processes running for a long time are penalized by decreasing their priority.

- On the based on scheduling technique, processes are traditionally classified as I/O-bound or CPU-bound.
 - The I/O-bound make heavy use of I/O devices and spend much time waiting for I/O operations to complete.
 - The CPU-bound carry on number-crunching applications that require a lot of CPU time.
- An alternative classification distinguishes three classes of processes:
 - *Interactive processes*
 - * These interact constantly with their users, and therefore spend a lot of time waiting for keypresses and mouse operations.
 - * When input is received, the process must be woken up quickly, or the user will find the system to be unresponsive.
 - * Typically, the average delay must fall between 50 and 150 milliseconds.
 - * Typical interactive programs are command shells, text editors, and graphical applications.
 - *Batch processes*
 - * These do not need user interaction, and hence they often run in the background. Because such processes do not need to be very responsive, they are often penalized by the scheduler.
 - * Typical batch programs are programming language compilers, database search engines, and scientific computations.
 - *Real-time processes*
 - * These have very stringent scheduling requirements. Such processes should never be blocked by lower-priority processes and should have a short guaranteed response time with a minimum variance.
 - * Typical real-time programs are video and sound applications, robot controllers, and programs that collect data from physical sensors.

40.2 System calls related to scheduling

- Programmers may change the scheduling priorities by means of the system calls.
- The following table provides more details about System Calls Related to Scheduling.

System call	Description
<code>nice()</code>	Change the static priority of a conventional process
<code>getpriority()</code>	Get the maximum static priority of a group of conventional processes
<code>setpriority()</code>	Set the static priority of a group of conventional processes
<code>sched_getscheduler()</code>	Get the scheduling policy of a process
<code>sched_setscheduler()</code>	Set the scheduling policy and the real-time priority of a process
<code>sched_getparam()</code>	Get the real-time priority of a process
<code>sched_setparam()</code>	Set the real-time priority of a process
<code>sched_yield()</code>	Relinquish the processor voluntarily without blocking
<code>sched_get_priority_min()</code>	Get the minimum real-time priority value for a policy
<code>sched_get_priority_max()</code>	Get the maximum real-time priority value for a policy
<code>sched_rr_get_interval()</code>	Get the time quantum value for the Round Robin policy
<code>sched_setaffinity()</code>	Set the CPU affinity mask of a process
<code>sched_getaffinity()</code>	Get the CPU affinity mask of a process

40.3 Scheduling Algorithm

- The scheduling algorithm used in earlier versions of Linux was quite simple and straightforward.
- The scheduling algorithm of Linux 2.6 is much more sophisticated.
 - It scales well with the number of runnable processes.
 - It also scales well with the number of processors because each CPU has its own queue of runnable processes.
 - Furthermore, the new algorithm does a better job of distinguishing interactive processes and batch processes.
 - As a consequence, users of heavily loaded systems feel that interactive applications are much more responsive in Linux 2.6 than in earlier versions.
- The scheduler always succeeds in finding a process to be executed; in fact, there is always at least one runnable process: the swapper process, which has PID 0 and executes only when the CPU cannot execute other processes.
- **Note:** *The every CPU of a multiprocessor system has its own swapper process with PID equal to 0.*
- Every Linux process is always scheduled according to one of the following scheduling classes:
 - *SCHED_FIFO*
 - * *SCHED_FIFO* is a First-In, First-Out real-time process.
 - * When the scheduler assigns the CPU to the process, it leaves the process descriptor in its current position in the runqueue list.
 - * If no other higher-priority real-time process is runnable, the process continues to use the CPU as long as it wishes, even if other real-time processes that have the same priority are runnable.

- *SCHED_RR*
 - * *SCHED_RR* is a Round Robin real-time process.
 - * When the scheduler assigns the CPU to the process, it puts the process descriptor at the end of the runqueue list.
 - * This policy ensures a fair assignment of CPU time to all *SCHED_RR* real-time processes that have the same priority.
- *SCHED_NORMAL*
 - * *SCHED_NORMAL* is a conventional, time-shared process.
- The scheduling algorithm behaves quite differently depending on whether the process is conventional or real-time.

40.4 Scheduling of Conventional Processes

- Every conventional process has its own static priority, which is a value used by the scheduler to rate the process with respect to the other conventional processes in the system.
- The kernel represents the static priority of a conventional process with a number ranging from 100 (highest priority) to 139 (lowest priority); notice that static priority decreases as the values increase.
- A new process always inherits the static priority of its parent.
- However, a user can change the static priority of the processes that he owns by passing some “nice values” to the nice() and setpriority() system calls.

- *Base time quantum*
 - * The static priority essentially determines the base time quantum of a process, that is, the time quantum duration assigned to the process when it has exhausted its previous time quantum.
 - * Static priority and base time quantum are related by the following formula:

$$\text{base time quantum} = \begin{cases} (140 - \text{static priority}) \times 20 & \text{if static priority} < 120 \\ (\text{140} - \text{static priority}) \times 5 & \text{if static priority} \geq 120 \end{cases} \quad (1)$$

- * As you see, the higher the static priority (i.e., the lower its numerical value), the longer the base time quantum.
- * As a consequence, higher priority processes usually get longer slices of CPU time with respect to lower priority processes.
- * The following Table shows the static priority, the base time quantum values, and the corresponding nice values for a conventional process having highest static priority, default static priority, and lowest static priority.

Description	Static priority	Nice value	Base time quantum
Highest static priority	100	-20	800 ms
High static priority	110	-10	600 ms
Default static priority	120	0	100 ms
Low static priority	130	+10	50 ms
Lowest static priority	139	+19	5 ms

– *Dynamic priority and average sleep time*

- * A conventional process also has a dynamic priority, which is a value ranging from 100 (highest priority) to 139 (lowest priority).
 - * The dynamic priority is the number actually looked up by the scheduler when selecting the new process to run.
 - * It is related to the static priority by the following empirical formula:
- $$\text{dynamic priority} = \max(100, \min(\text{static priority} - \text{bonus} + 5, 139)) \quad (2)$$
- * The bonus is a value ranging from 0 to 10; a value less than 5 represents a penalty that lowers the dynamic priority, while a value greater than 5 is a premium that raises the dynamic priority.
 - * The value of the bonus, in turn, depends on the past history of the process; more precisely, it is related to the average sleep time of the process.
 - * Roughly, the average sleep time is the average number of nanoseconds that the process spent while sleeping. However, that this is not an average operation on the elapsed time.
 - * The average sleep time decreases while a process is running.
 - * The average sleep time can never become larger than 1 second.
 - * The correspondence between average sleep times and bonus values is shown in Table below.

Average sleep time	Bonus
Greater than or equal to 0 but smaller than 100 ms	0
Greater than or equal to 100 ms but smaller than 200 ms	1
Greater than or equal to 200 ms but smaller than 300 ms	2
Greater than or equal to 300 ms but smaller than 400 ms	3
Greater than or equal to 400 ms but smaller than 500 ms	4
Greater than or equal to 500 ms but smaller than 600 ms	5
Greater than or equal to 600 ms but smaller than 700 ms	6
Greater than or equal to 700 ms but smaller than 800 ms	7
Greater than or equal to 800 ms but smaller than 900 ms	8
Greater than or equal to 900 ms but smaller than 1000 ms	9
1 second	10

- * The average sleep time is also used by the scheduler to determine whether a given process should be considered interactive or batch.
- * More precisely, a process is considered “interactive” if it satisfies the following formula:

$$\text{dynamic priority} \leq 3 \times \text{static priority} / 4 + 28 \quad (3)$$

- * which is equivalent to the following:

$$\text{bonus} - 5 \geq \text{static priority} / 4 - 28$$

- * The expression $\text{static priority}/4 - 28$ is called the interactive delta.
- * It should be noted that it is far easier for high priority than for low priority processes to become interactive.
- * For instance, a process having highest static priority (100) is considered interactive when its bonus value exceeds 2, that is, when its average sleep time exceeds 200 ms.
- * Conversely, a process having lowest static priority (139) is never considered as interactive, because the bonus value is always smaller than the value 11 required to reach an interactive delta equal to 6.
- * A process having default static priority (120) becomes interactive as soon as its average sleep time exceeds 700 ms.
- *Active and expired processes*
 - * Even if conventional processes having higher static priorities get larger slices of the CPU time, they should not completely lock out the processes having lower static priority.
 - * To avoid process starvation, when a process finishes its time quantum, it can be replaced by a lower priority process whose time quantum has not yet been exhausted.
 - * To implement this mechanism, the scheduler keeps two disjoint sets of runnable processes:
 - *Active processes* : These runnable processes have not yet exhausted their time quantum and are thus allowed to run.
 - *Expired processes* : These runnable processes have exhausted their time quantum and are thus forbidden to run until all active processes expire.

40.5 Scheduling of Real-Time Processes

- Every real-time process is associated with a real-time priority, which is a value ranging from 1 (highest priority) to 99 (lowest priority).
- The scheduler always favors a higher priority runnable process over a lower priority one; in other words, a real-time process inhibits the execution of every lower-priority process while it remains runnable.
- Contrary to conventional processes, real-time processes are always considered active.

- The user can change the real-time priority of a process by means of the `sched_setparam()` and `sched_setscheduler()` system calls.
- If several real-time runnable processes have the same highest priority, the scheduler chooses the process that occurs first in the corresponding list of the local CPU's runqueue.
- A real-time process is replaced by another process only when one of the following events occurs:
 - The process is preempted by another process having higher real-time priority.
 - The process performs a blocking operation, and it is put to sleep (in state `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`).
 - The process is stopped (in state `TASK_STOPPED` or `TASK_TRACED`), or it is killed (in state `EXIT_ZOMBIE` or `EXIT_DEAD`).
 - The process voluntarily relinquishes the CPU by invoking the `sched_yield()` system call.
 - The process is Round Robin real-time (`SCHED_RR`), and it has exhausted its time quantum.

40.6 Functions Used by the Scheduler

- The scheduler relies on several functions in order to do its work; the most important are:
 - `scheduler_tick()` : Keeps the `time_slice` counter of current up-to-date.
 - `try_to_wake_up()` : Awakens a sleeping process.
 - `recalc_task_prio()` : Updates the dynamic priority of a process
 - `schedule()` : Selects a new process to be executed
 - `load_balance()` : Keeps the runqueues of a multiprocessor system balanced



Faculty of Engineering and Technology (ITER)
SIKSHA 'O' ANUSANDHAN (DEEMED TO BE) UNIVERSITY
Bhubaneswar, ODISHA