Operating system concepts (7. ed.).

Book · January 2005
Source: DBLP

3 authors, including:

Abraham Silberschatz
Yale University

OPERATING SYSTEM CONCEPTS

Avi Silberschatz
Department of Computer Sciences
University of Texas at Austin

Peter Galvin
Department of Computer Science
Brown University

CHAPTER 1: INTRODUCTION

- What is an operating system?
- Early Systems
- Simple Batch Systems
- Multiprogramming Batched Systems
- Time-Sharing Systems
- Personal-Computer Systems
- Parallel Systems
- Distributed Systems
- Real-Time Systems

Operating system – a program that acts as an intermediary between a user of a computer and the computer hardware.

Operating system goals:

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

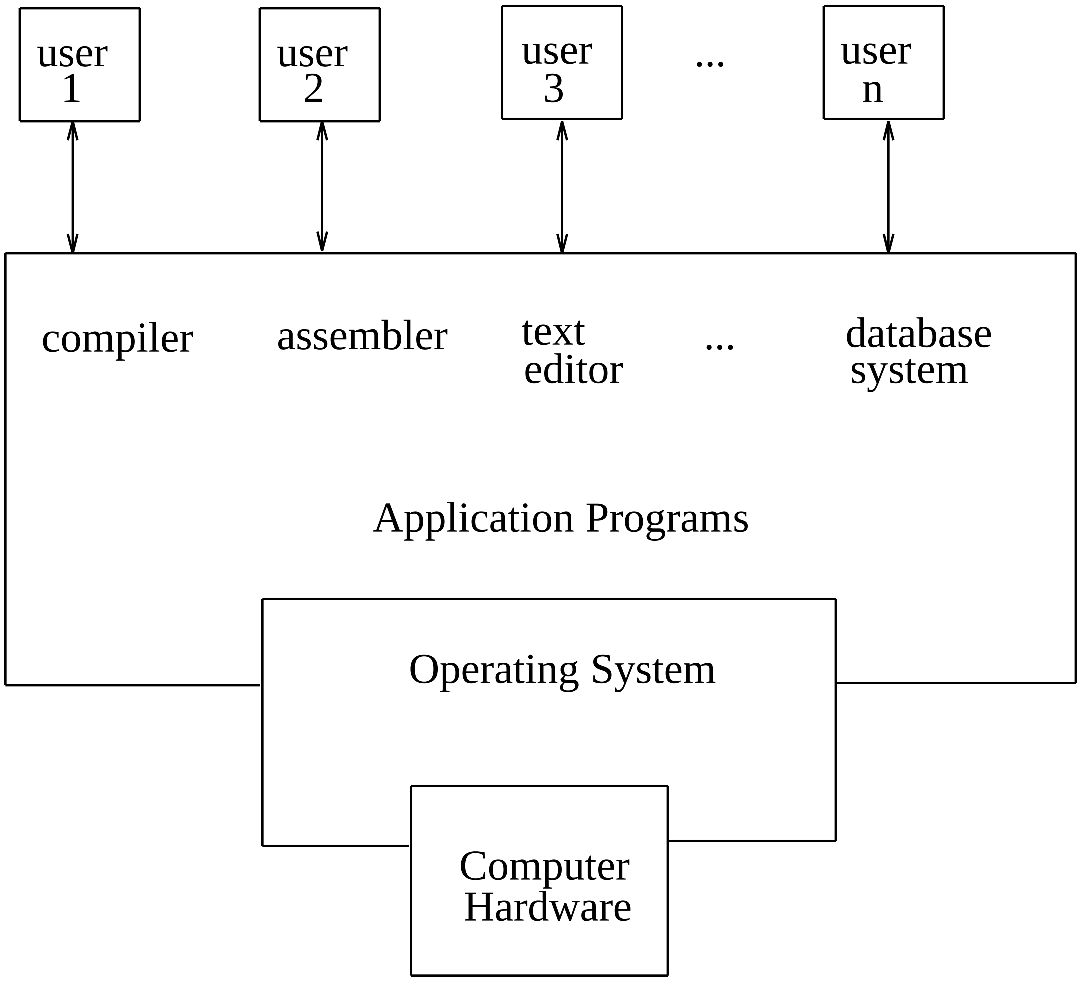
Computer System Components

1. Hardware – provides basic computing resources (CPU, memory, I/O devices).

2. Operating system – controls and coordinates the use of the hardware among the various application programs for the various users.

3. Applications programs – define the ways in which the system resources are used to solve the computing problems of the users (compilers, database systems, video games, business programs).

4. Users (people, machines, other computers).


user
1
user
2
user
3
...
user
n
compiler
assembler
text
editor
...
database
system
Application Programs
Operating System
Computer
Hardware

Operating System Definitions

- Resource allocator – manages and allocates resources.
- Control program – controls the execution of user programs and operation of I/O devices.
- Kernel – the one program running at all times (all else being application programs).

Early Systems – bare machine (early 1950s)

- Structure
  - Large machines run from console
  - Single user system
  - Programmer/User as operator
  - Paper tape or punched cards
- Early Software
  - Assemblers
  - Loaders
  - Linkers
  - Libraries of common subroutines
  - Compilers
  - Device drivers
- Secure
- Inefficient use of expensive resources
  - Low CPU utilization
  - Significant amount of setup time

Simple Batch Systems

- Hire an operator
- User $\neq$ operator
- Add a card reader
- Reduce setup time by batching similar jobs
- Automatic job sequencing – automatically transfers control from one job to another. First rudimentary operating system.
- Resident monitor
  - initial control in monitor
  - control transfers to job
  - when job completes control transfers back to monitor

Problems:

1) How does the monitor know about the nature of the job (e.g., Fortran versus Assembly) or which program to execute?

2) How does the monitor distinguish
   a) job from job?
   b) data from program?

Solution: introduce control cards

Control Cards

- Special cards that tell the resident monitor which programs to run.

  $JOB
  $FTN
  $RUN
  $DATA
  $END

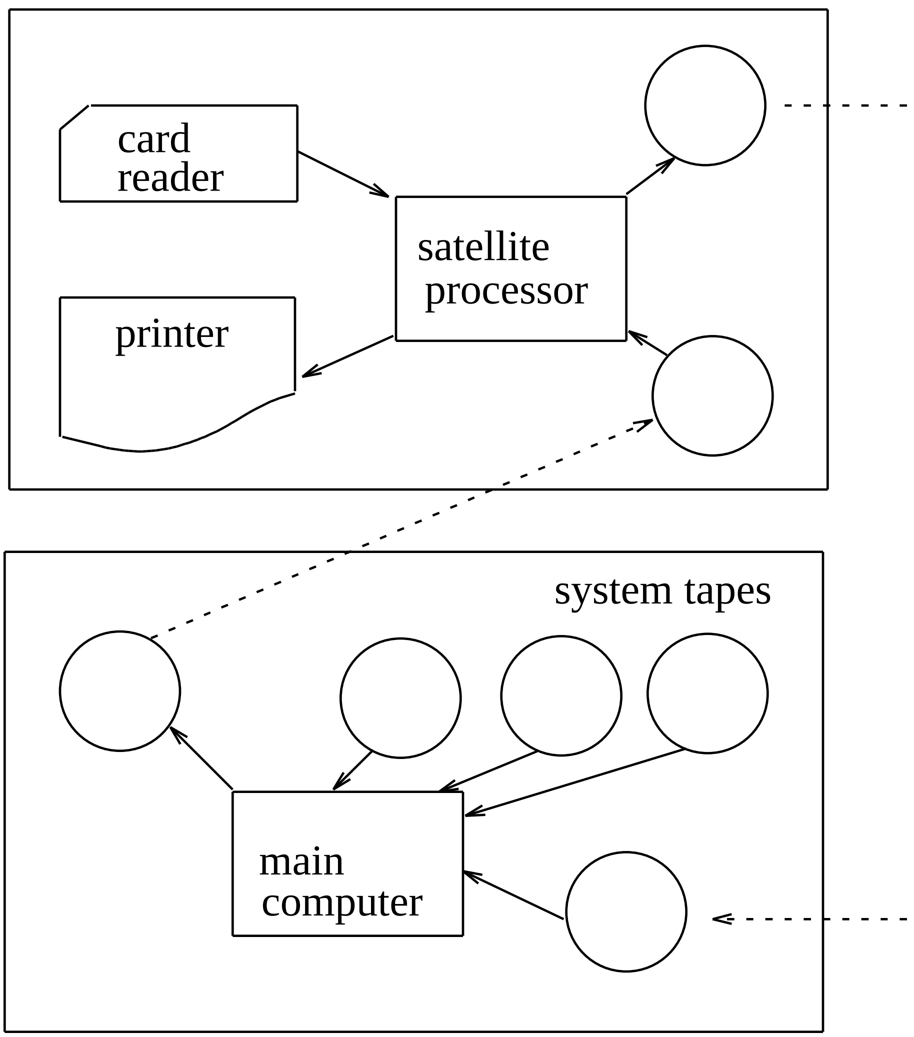
- Special characters distinguish control cards from data or program cards:

  $ in column 1
  // in column 1 and 2
  7-9 in column 1

- Parts of resident monitor
  - Control card interpreter – responsible for reading and carrying out instructions on the cards.
  - Loader – loads systems programs and applications programs into memory.
  - Device drivers – know special characteristics and properties for each of the system's I/O devices.

- Problem: Slow Performance – since I/O and CPU could not overlap, and card reader very slow.

- Solution: Off-line operation – speed up computation by loading jobs into memory from tapes and card reading and line printing done off-line.

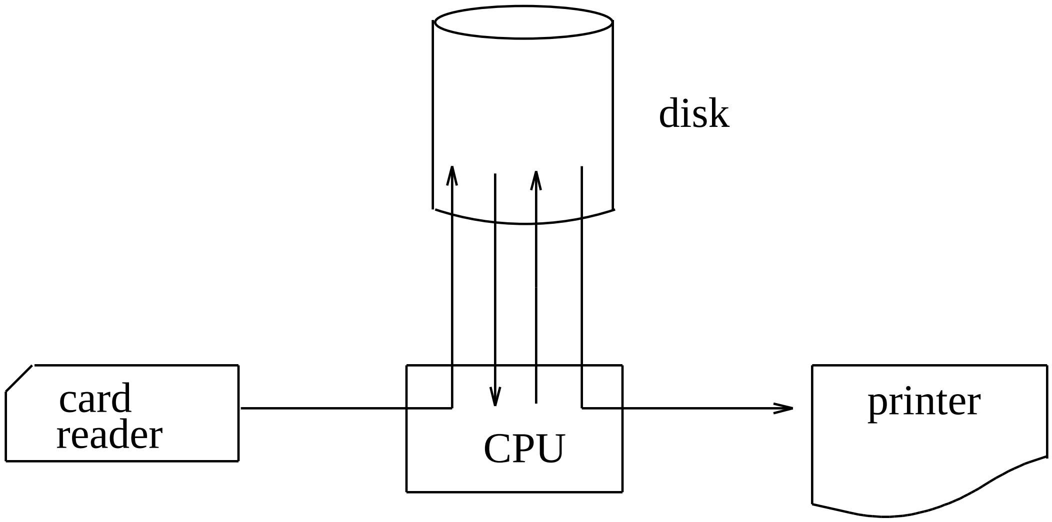


- Advantage of off-line operation – main computer not constrained by the speed of the card readers and line printers, but only by the speed of faster magnetic tape units.

- No changes need to be made to the application programs to change from direct to off-line I/O operation.

- Real gain – possibility of using multiple reader-to-tape and tape-to-printer systems for one CPU.

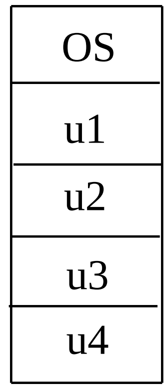
Spooling – overlap the I/O of one job with the computation of another job.

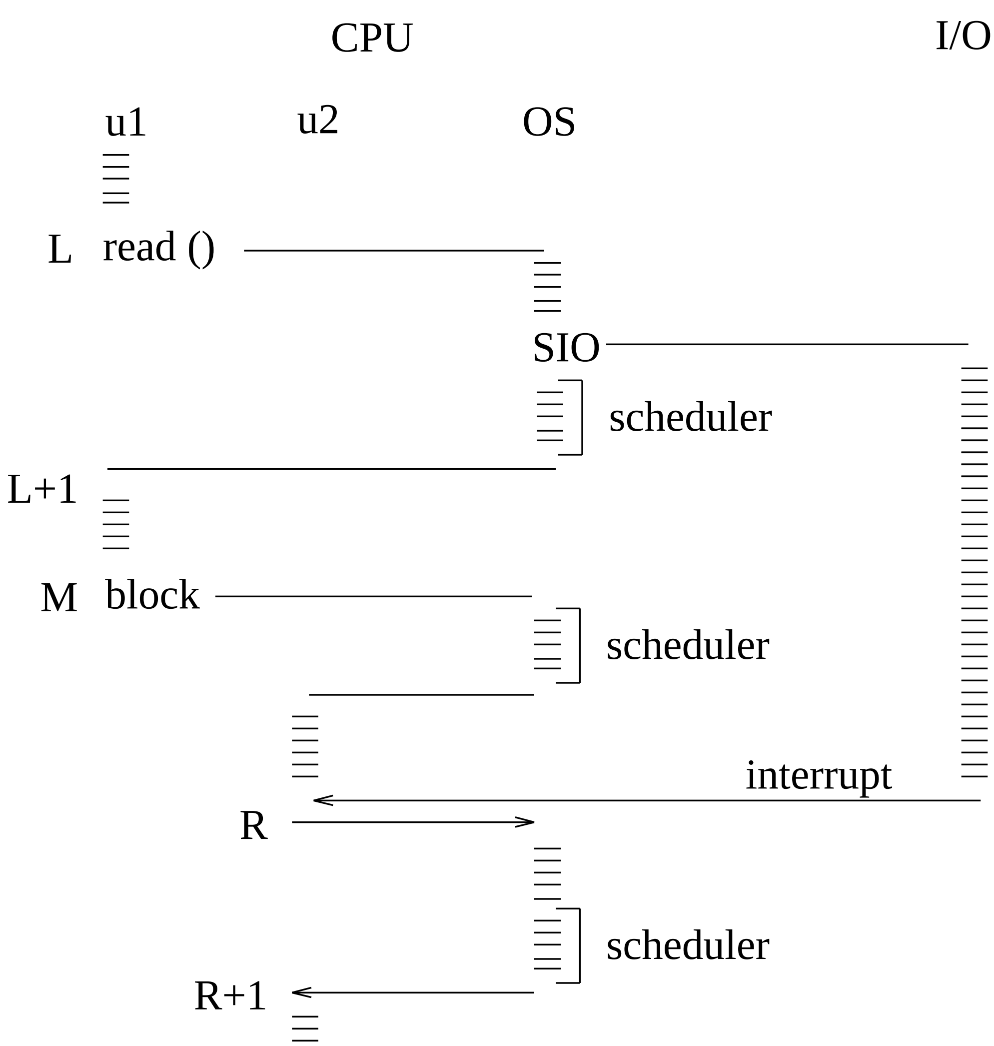


- While executing one job, the operating system:
  - reads the next job from the card reader into a storage area on the disk (job queue).
  - outputs the printout of previous job from disk to the line printer.

- *Job pool* – data structure that allows the operating system to select which job to run next, in order to increase CPU utilization.

Multiprogrammed Batch Systems – several jobs are kept in main memory at the same time, and the CPU is multiplied among them.





OS Features Needed for Multiprogramming

- I/O routine supplied by the system.
- Memory management – the system must allocate the memory to several jobs.
- CPU scheduling – the system must choose among several jobs ready to run.
- Allocation of devices.

Time-Sharing Systems– Interactive Computing

- The CPU is multiplied among several jobs that are kept in memory and on disk (the CPU is allocated to a job only if the job is in memory).

- A job is swapped in and out of memory to the disk.

- On-line communication between the user and the system is provided; when the operating system finishes the execution of one command, it seeks the next ‘‘control statement’’ not from a card reader, but rather from the user’s keyboard.

- On-line file system must be available for users to access data and code.

Personal-Computer Systems

- *Personal computers* – computer system dedicated to a single user.

- I/O devices – keyboards, mice, display screens, small printers.

- User convenience and responsiveness.

- Can adopt technology developed for larger operating systems; often individuals have sole use of computer and do not need advanced CPU utilization or protection features.

Parallel Systems – multiprocessor systems with more than one CPU in close communication.

- *Tightly coupled* system – processors share memory and a clock; communication usually takes place through the shared memory.

- Advantages of parallel systems:

  - Increased *throughput*

  - Economical

  - Increased reliability
    - *graceful degradation*
    - *fail-soft* systems

- *Symmetric multiprocessing*
  - Each processor runs an identical copy of the operating system.
  - Many processes can run at once without performance deterioration.

- *Asymmetric multiprocessing*
  - Each processor is assigned a specific task; master processor schedules and allocates work to slave processors.
  - More common in extremely large systems.

Distributed Systems – distribute the computation among several physical processors.

- *Loosely coupled* system – each processor has its own local memory; processors communicate with one another through various communication lines, such as high-speed buses or telephone lines.

- Advantages of distributed systems:
  - Resource sharing
  - Computation speed up – load sharing
  - Reliability
  - Communication

Real-Time Systems

- Often used as a control device in a dedicated application such as controlling scientific experiments, medical imaging systems, industrial control systems, and some display systems.

- Well-defined fixed-time constraints.

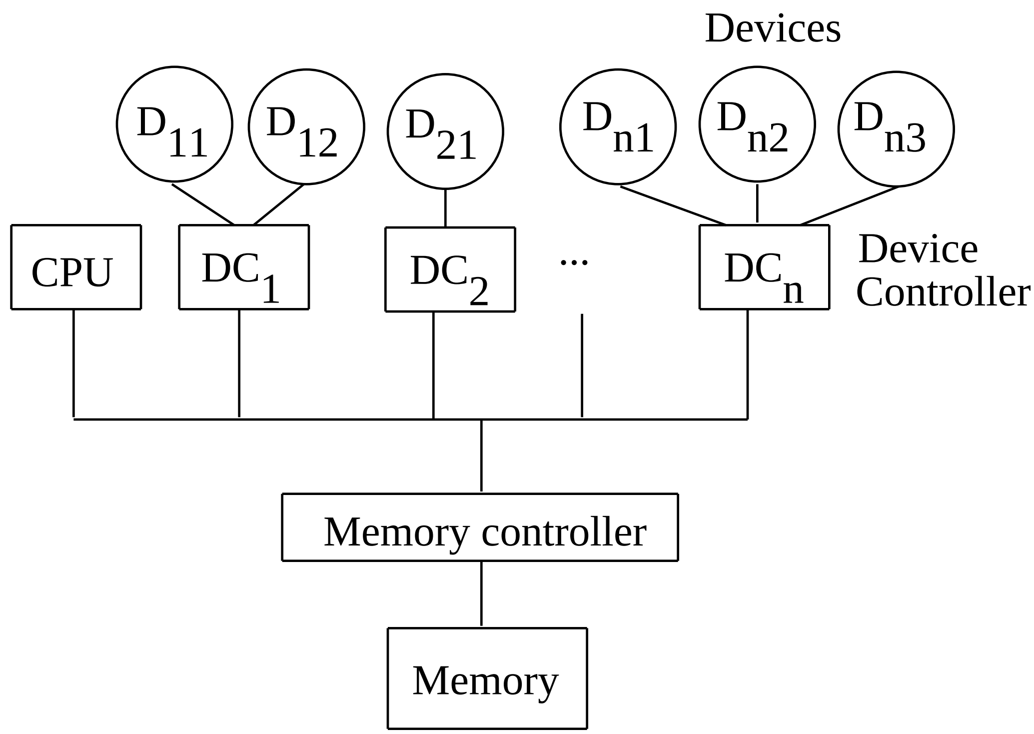
- *Hard real-time* system.
  - Secondary storage limited or absent; data stored in short-term memory, or read-only memory (ROM).
  - Conflicts with time-sharing systems; not supported by general-purpose operating systems.

- *Soft real-time* system.
  - Limited utility in industrial control or robotics.
  - Useful in applications (multimedia, virtual reality) requiring advanced operating-system features.

CHAPTER 2: COMPUTER-SYSTEM STRUCTURES

- Computer-System Operation
- I/O Structure
- Storage Structure
- Storage Hierarchy
- Hardware Protection
- General System Architecture

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.
- CPU moves data from/to main memory to/from the 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*.

Common Functions of Interrupts

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

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

- Incoming interrupts are *disabled* while another interrupt is being processed to prevent a *lost interrupt*.

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

- An operating system is *interrupt driven*.

Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter.

- Determines which type of interrupt has occurred:

  - *polling*

  - vectored interrupt system

- Separate segments of code determine what action should be taken for each type of interrupt.

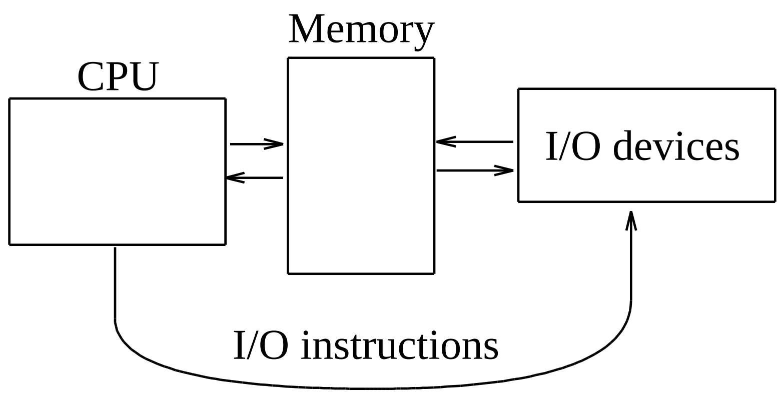
I/O Structure

- After I/O starts, control returns to user program only upon I/O completion.
  - **wait** instruction idles the CPU until the next interrupt.
  - wait loop (contention for memory access).
  - at most one I/O request is outstanding at a time; no simultaneous I/O processing.

- After I/O starts, control returns to user program without waiting for I/O completion.
  - *System call* – request to the operating system to allow user to wait for I/O completion.
  - *Device-status table* contains entry for each I/O device indicating its type, address, and state.
  - Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt.

Direct Memory Access (DMA) Structure

- Schema



- Used for high-speed I/O devices able to transmit information at close to memory speeds.

- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention.

- Only one interrupt is generated per block, rather than the one interrupt per byte.

Storage Structure

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

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

- Magnetic disks – rigid metal or glass platters covered with magnetic recording material.
  - Disk surface is logically divided into *tracks*, which are subdivided into *sectors*.
  - The *disk controller* determines the logical interaction between the device and the computer.

Storage Hierarchy

- Storage systems organized in hierarchy:
  - speed
  - cost
  - volatility

- *Caching* – copying information into faster storage system; main memory can be viewed as a fast *cache* for secondary memory.

Hardware Protection

- Dual-Mode Operation

- I/O Protection

- Memory Protection

- CPU Protection

Dual-Mode Operation

- Sharing system resources requires operating system to ensure that an incorrect program cannot cause other programs to execute incorrectly.

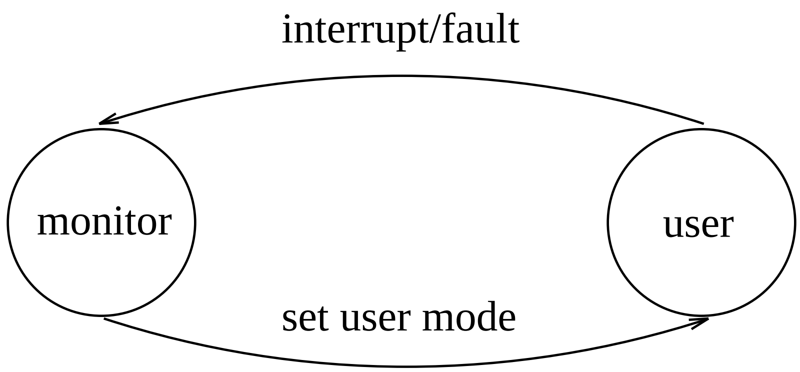
- Provide hardware support to differentiate between at least two modes of operations.

  1. *User mode* – execution done on behalf of a user.

  2. *Monitor mode* (also *supervisor mode* or *system mode*) – execution done on behalf of operating system.

- *Mode bit* added to computer hardware to indicate the current mode: monitor (0) or user (1).

- When an interrupt or fault occurs hardware switches to monitor mode



- *Privileged instructions* can be issued only in monitor mode.

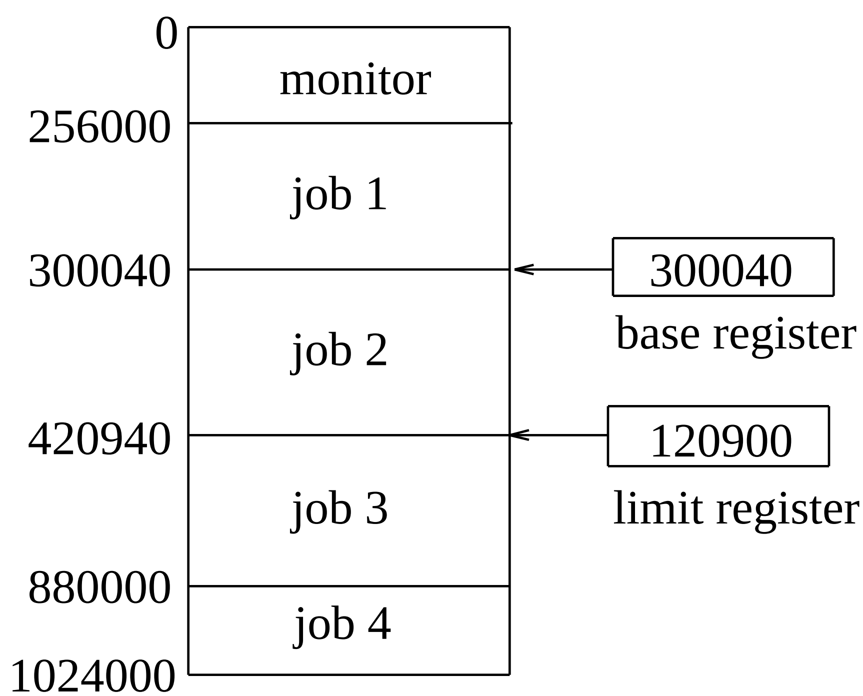
I/O Protection

- All I/O instructions are privileged instructions.

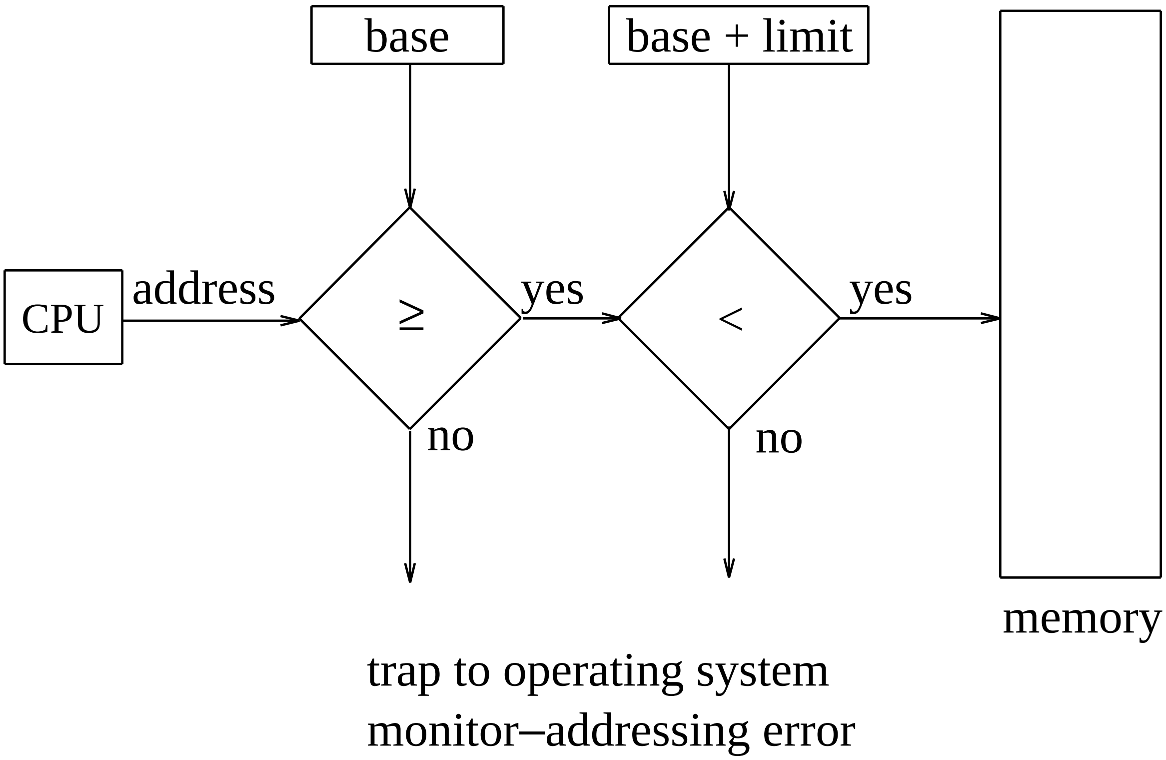
- Must ensure that a user program could never gain control of the computer in monitor mode (i.e., a user program that, as part of its execution, stores a new address in the interrupt vector).

Memory Protection

- Must provide memory protection at least for the interrupt vector and the interrupt service routines.
- In order to have memory protection, add two registers that determine the range of legal addresses a program may access:
  - **base register** – holds the smallest legal physical memory address.
  - **limit register** – contains the size of the range.
- Memory outside the defined range is protected.



- Protection hardware



- When executing in monitor mode, the operating system has unrestricted access to both monitor and users' memory.
- The load instructions for the *base* and *limit* registers are privileged instructions.

CPU Protection

- *Timer* – interrupts computer after specified period to ensure operating system maintains control.
  - Timer is decremented every clock tick.
  - When timer reaches the value 0, an interrupt occurs.
- Timer commonly used to implement time sharing.
- Timer also used to compute the current time.
- Load-timer is a privileged instruction.

General-System Architecture

- Given that I/O instructions are privileged, how does the user program perform I/O?

- System call – the method used by a process to request action by the operating system.
  - Usually takes the form of a trap to a specific location in the interrupt vector.
  - Control passes through the interrupt vector to a service routine in the OS, and the mode bit is set to monitor mode.
  - The monitor verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call.

CHAPTER 3: OPERATING-SYSTEM STRUCTURES

- System Components
- Operating-System Services
- System Calls
- System Programs
- System Structure
- Virtual Machines
- System Design and Implementation
- System Generation

Most operating systems support the following types of system components:

- Process Management
- Main-Memory Management
- Secondary-Storage Management
- I/O System Management
- File Management
- Protection System
- Networking
- Command-Interpreter System

Process Management

- A *process* is a program in execution. A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task.

- The operating system is responsible for the following activities in connection with process management:

  - process creation and deletion.

  - process suspension and resumption.

  - provision of mechanisms for:
    - ○ process synchronization
    - ○ process communication

Main-Memory Management

- Memory is a large array of words or bytes, each with its own address. It is a repository of quickly accessible data shared by the CPU and I/O devices.

- Main memory is a volatile storage device. It loses its contents in the case of system failure.

- The operating system is responsible for the following activities in connection with memory management:

  - Keep track of which parts of memory are currently being used and by whom.

  - Decide which processes to load when memory space becomes available.

  - Allocate and deallocate memory space as needed.

Secondary-Storage Management

- Since main memory (*primary storage*) is volatile and too small to accommodate all data and programs permanently, the computer system must provide *secondary storage* to back up main memory.

- Most modern computer systems use disks as the principle on-line storage medium, for both programs and data.

- The operating system is responsible for the following activities in connection with disk management:

  - Free-space management

  - Storage allocation

  - Disk scheduling

I/O System Management

- The I/O system consists of:
  - A buffer-caching system
  - A general device-driver interface
  - Drivers for specific hardware devices

File Management

- A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data.

- The operating system is responsible for the following activities in connection with file management:

  - File creation and deletion.
  - Directory creation and deletion.
  - Support of primitives for manipulating files and directories.
  - Mapping files onto secondary storage.
  - File backup on stable (nonvolatile) storage media.

Protection System

- *Protection* refers to a mechanism for controlling access by programs, processes, or users to both system and user resources.

- The protection mechanism must:
  - distinguish between authorized and unauthorized usage.
  - specify the controls to be imposed.
  - provide a means of enforcement.

Networking (Distributed Systems)

- A *distributed* system is a collection of processors that do not share memory or a clock. Each processor has its own local memory.

- The processors in the system are connected through a *communication network.*

- A distributed system provides user access to various system resources.

- Access to a shared resource allows:
  - Computation speed-up
  - Increased data availability
  - Enhanced reliability

Command-Interpreter System

- Many commands are given to the operating system by *control statements* which deal with:
  - process creation and management
  - I/O handling
  - secondary-storage management
  - main-memory management
  - file-system access
  - protection
  - networking

- The program that reads and interprets control statements is called variously:
  - control-card interpreter
  - command-line interpreter
  - shell (in UNIX)

  Its function is to get and execute the next command statement.

Operating-System Services

- Program execution – system capability to load a program into memory and to run it.

- I/O operations – since user programs cannot execute I/O operations directly, the operating system must provide some means to perform I/O.

- File-system manipulation – program capability to read, write, create, and delete files.

- Communications – exchange of information between processes executing either on the same computer or on different systems tied together by a network. Implemented via *shared memory* or *message passing*.

- Error detection – ensure correct computing by detecting errors in the CPU and memory hardware, in I/O devices, or in user programs.

Additional operating-system functions exist not for helping the user, but rather for ensuring efficient system operation.

- Resource allocation – allocating resources to multiple users or multiple jobs running at the same time.

- Accounting – keep track of and record which users use how much and what kinds of computer resources for account billing or for accumulating usage statistics.

- Protection – ensuring that all access to system resources is controlled.

System Calls

- System calls provide the interface between a running program and the operating system.
  - Generally available as assembly-language instructions.
  - Languages defined to replace assembly language for systems programming allow system calls to be made directly (e.g., C, Bliss, PL/360).

- Three general methods are used to pass parameters between a running program and the operating system:
  - Pass parameters in *registers*.
  - Store the parameters in a table in memory, and the table address is passed as a parameter in a register.
  - *Push* (store) the parameters onto the *stack* by the program, and *pop* off the stack by the operating system.

System Programs

- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File manipulation
  - Status information
  - File modification
  - Programming-language support
  - Program loading and execution
  - Communications
  - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls.

System Structure – Simple Approach

- MS-DOS – written to provide the most functionality in the least space; it was not divided into modules. MS-DOS has some structure, but its interfaces and levels of functionality are not well separated.

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts:
  - the systems programs.
  - the kernel, which consists of everything below the system-call interface and above the physical hardware. Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level.

System Structure – Layered Approach

- 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 layered design was first used in the THE operating system. Its six layers are as follows:

Level 5:	user programs
Level 4:	buffering for input and output devices
Level 3:	operator-console device driver
Level 2:	memory management
Level 1:	CPU scheduling
Level 0:	hardware

Virtual Machines

- A *virtual machine* takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware.
- A virtual machine provides an interface *identical* to the underlying bare hardware.
- The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory.
- The resources of the physical computer are shared to create the virtual machines.
  - CPU scheduling can create the appearance that users have their own processor.
  - Spooling and a file system can provide virtual card readers and virtual line printers.
  - A normal user time-sharing terminal serves as the virtual machine operator's console.

Advantages and Disadvantages of Virtual Machines

- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.

- A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.

- The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate of the underlying machine.

System Design Goals

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

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

Mechanisms and Policies

- Mechanisms determine *how* to do something; policies decide *what* will be done.

- The separation of *policy* from *mechanism* is a very important principle; it allows maximum flexibility if policy decisions are to be changed later.

System Implementation

- Traditionally written in assembly language, operating systems can now be written in higher-level languages.

- Code written in a high-level language:
  - can be written faster.
  - is more compact.
  - is easier to understand and debug.

- An operating system is far easier to *port* (move to some other hardware) if it is written in a high-level language.

System Generation (SYSGEN)

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site.

- SYSGEN program obtains information concerning the specific configuration of the hardware system.

- *Booting* – starting a computer by loading the kernel.

- *Bootstrap program* – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution.

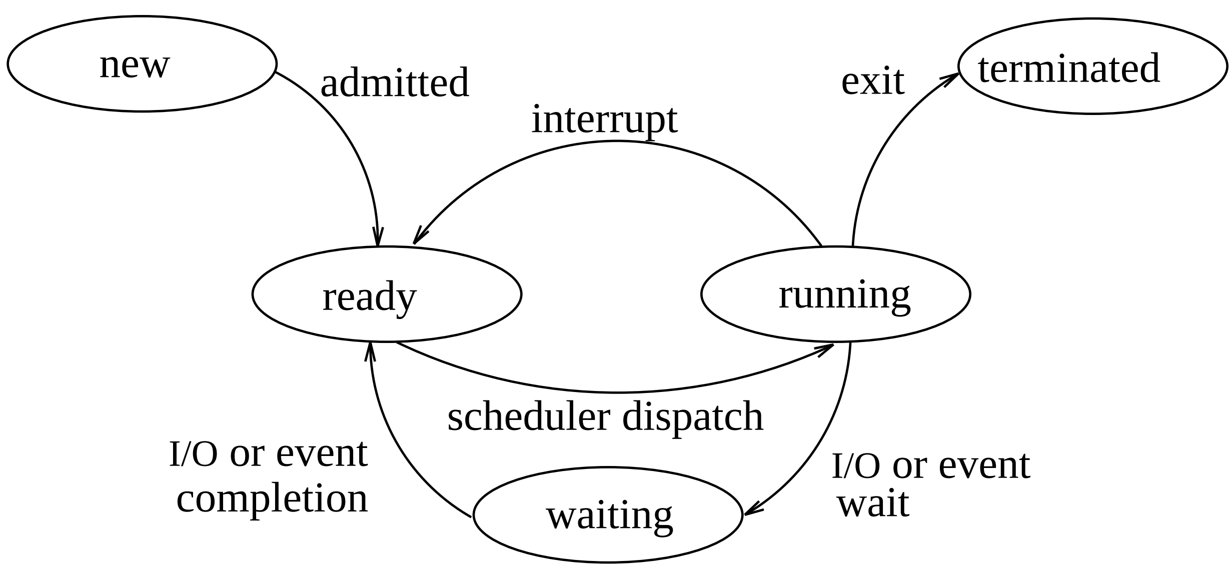
CHAPTER 4: PROCESSES

- Process Concept
- Process Scheduling
- Operation on Processes
- Cooperating Processes
- Threads
- Interprocess Communication

Process Concept

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

- As a process executes, it changes *state*.
  - **New:** The process is being created.
  - **Running:** Instructions are being executed.
  - **Waiting:** The process is waiting for some event to occur.
  - **Ready:** The process is waiting to be assigned to a processor.
  - **Terminated:** The process has finished execution.
- Diagram of process state:

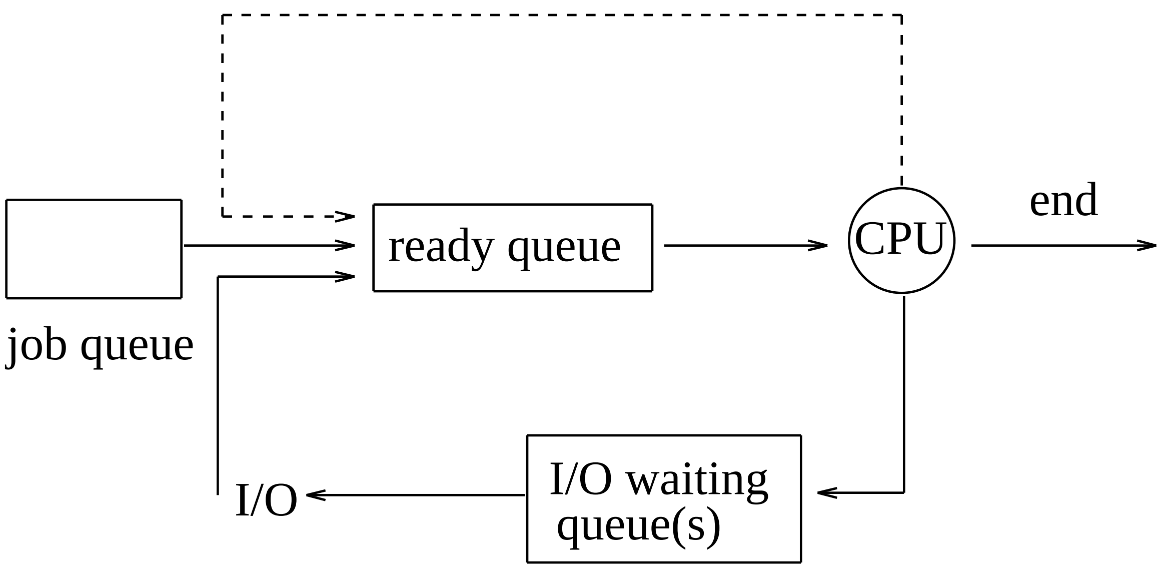


- Process Control Block (PCB) – Information associated with each process.

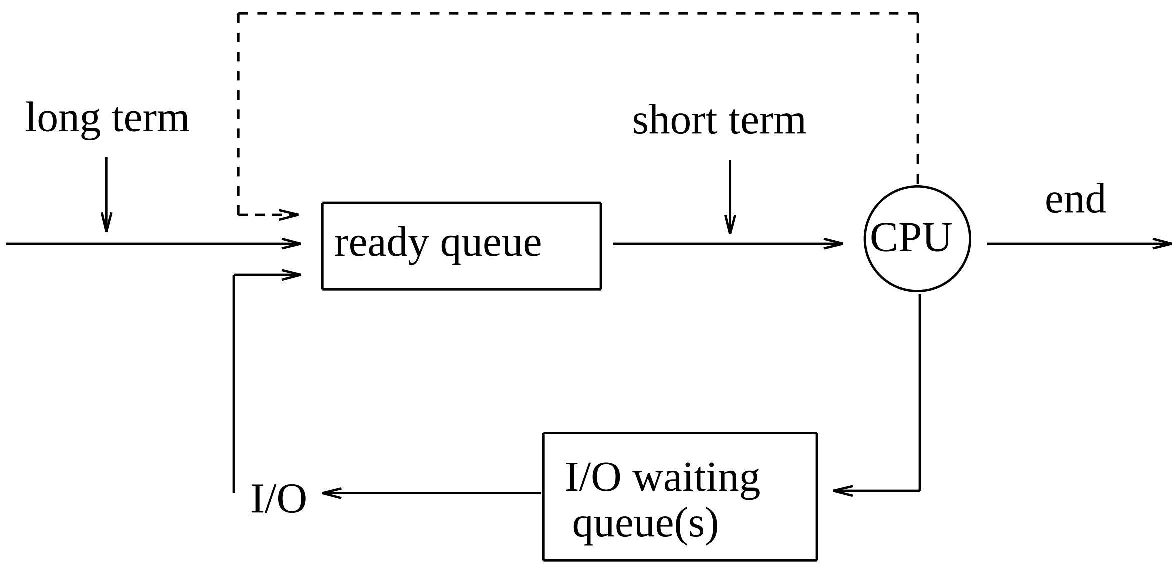
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information

- 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 a particular I/O device.

- Process migration between the various queues.



- Schedulers
  - *Long-term scheduler* (*job scheduler*) – selects which processes should be brought into the ready queue.
  - *Short-term scheduler* (*CPU scheduler*) – selects which process should be executed next and allocates CPU.



- Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast).

- Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow).

- The long-term scheduler controls the *degree of multiprogramming*.

- Processes can be described as either:

  - *I/O-bound process* – spends more time doing I/O than computations; many short CPU bursts.

  - *CPU-bound process* – spends more time doing computations; few very long CPU bursts.

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.

- Context-switch time is overhead; the system does no useful work while switching.

- Time dependent on hardware support.

Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.

- Resource sharing
  - Parent and children share all resources.
  - Children share subset of parent’s resources.
  - Parent and child share no resources.

- Execution
  - Parent and children execute concurrently.
  - Parent waits until children terminate.

- Address space
  - Child duplicate of parent.
  - Child has a program loaded into it.

- UNIX examples
  - **fork** system call creates new process.
  - **execve** system call used after a **fork** to replace the process’ memory space with a new program.

Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**).
  - Output data from child to parent (via **fork**).
  - Process' resources are deallocated by operating system.

- Parent may terminate execution of children processes (**abort**).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.
    - Operating system does not allow child to continue if its parent terminates.
    - *Cascading termination.*

Cooperating Processes

- *Independent* process cannot affect or be affected by the execution of another process.

- *Cooperating* process can affect or be affected by the execution of another process.

- Advantages of process cooperation:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Producer-Consumer Problem

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

- Shared-memory solution:
  - Shared data

    var *n;*
    type *item* = ... ;
    var *buffer*: **array** [0..*n*–1] **of** *item*;
    in, *out*: 0..*n*–1;
    in := 0;
    out := 0;

- Producer process

```
repeat
    ...
  produce an item in nextp
    ...
  while in+1 mod n = out do no-op;
  buffer[in] := nextp;
  in := in+1 mod n;
until false;
```

- Consumer process

```
repeat
  while in = out do no-op;
  nextc := buffer[out];
  out := out+1 mod n;
      ...
  consume the item in nextc
      ...
until false;
```

- Solution is correct, but can only fill up $n-1$ buffer.

Threads

- A *thread* (or *lightweight process*) is a basic unit of CPU utilization; it consists of:
  - program counter
  - register set
  - stack space

- A thread shares with its peer threads its:
  - code section
  - data section
  - operating-system resources

  collectively known as a *task*.

- A traditional or *heavyweight* process is equal to a task with one thread.

- In a task containing multiple threads, while one server thread is blocked and waiting, a second thread in the same task could run.

    - Cooperation of multiple threads in same job confers higher throughput and improved performance.

    - Applications that require sharing a common buffer (producer–consumer problem) benefit from thread utilization.

- Threads provide a mechanism that allows sequential processes to make blocking system calls while also achieving parallelism.

- Kernel-supported threads (Mach and OS/2).

- User-level threads; supported above the kernel, via a set of library calls at the user level (Project Andrew from CMU).

- Hybrid approach implements both user-level and kernel-supported threads (Solaris 2).

Solaris 2 – version of UNIX with support for threads at the kernel and user levels, symmetric multiprocessing, and real-time scheduling.

- LWP – intermediate level between user-level threads and kernel-level threads.
- Resource needs of thread types:
  - Kernel thread – small data structure and a stack; thread switching does not require changing memory access information, and therefore is relatively fast.
  - LWP – PCB with register data, accounting information, and memory information; switching between LWPs is relatively slow.
  - User-level thread – needs only a stack and a program counter. Switching is fast since kernel is not involved. Kernel only sees the LWPs in the process that support user-level threads.

Interprocess Communication (IPC) – provides a mechanism to allow processes to communicate and to synchronize their actions.

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

Implementation questions:

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bidirectional?

Direct Communication

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

Indirect Communication

- Messages are directed and received from *mailboxes* (also referred to as *ports*).
  - Each mailbox has a unique *id*.
  - Processes can communicate only if they share a mailbox.
- Properties of communication link
  - Link established only if the two processes share a mailbox in common.
  - A link may be associated with many processes.
  - Each pair of processes may share several communication links.
  - Link may be unidirectional or bidirectional.
- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox

Indirect Communication (Continued)

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A.
  - P_1 sends; P_2 and P_3 receive.
  - Who gets the message?

- Solutions
  - Allow a link to be associated with at most two processes.
  - Allow only one process at a time to execute a receive operation.
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Buffering – queue of messages attached to the link; implemented in one of three ways.

- Zero capacity – 0 messages
  Sender must wait for receiver (*rendezvous*).

- Bounded capacity – finite length of *n* messages
  Sender must wait if link full.

- Unbounded capacity – infinite length
  Sender never waits.

Exception Conditions – error recovery

- Process terminates
- Lost messages
- Scrambled Messages

CHAPTER 5: CPU SCHEDULING

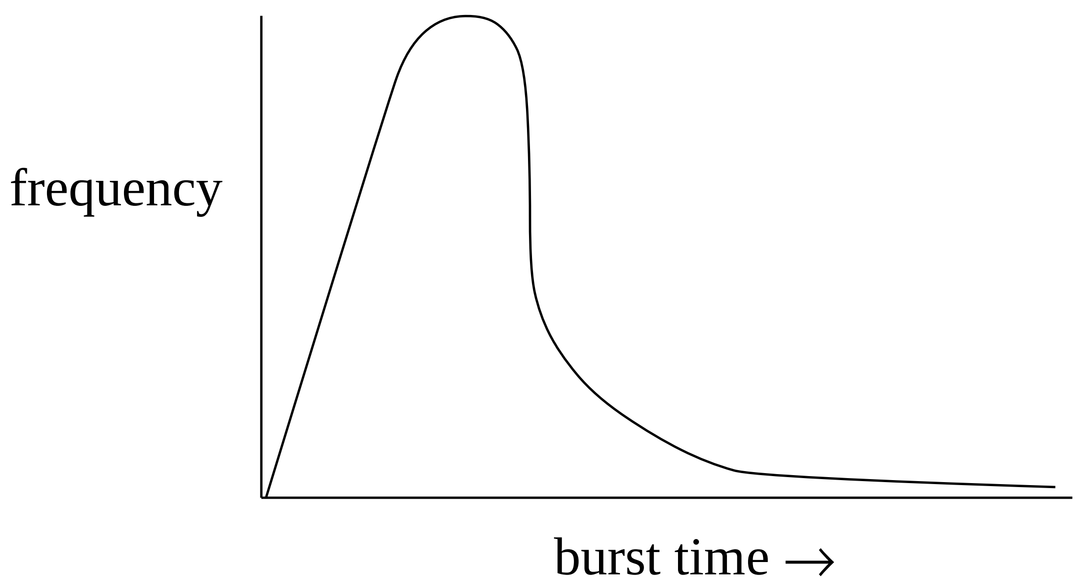
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time Scheduling
- Algorithm Evaluation

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming.

- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait.

- CPU burst distribution



- *Short-term scheduler* –selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

- CPU scheduling decisions may take place when a process:
    1. switches from running to waiting state.
    2. switches from running to ready state.
    3. switches from waiting to ready.
    4. terminates.

- Scheduling under 1 and 4 is *nonpreemptive*.

- All other scheduling is *preemptive*.

Dispatcher

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

Scheduling Criteria

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

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

First-Come, First-Served (FCFS) Scheduling

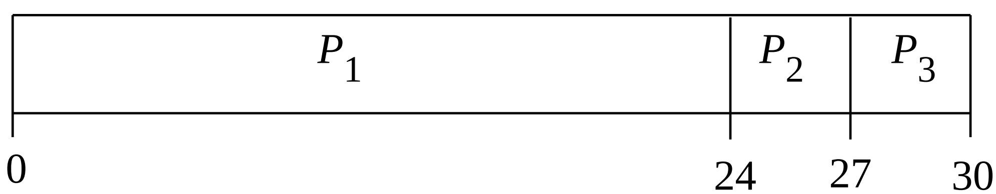
- Example:

Process	Burst time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order:

  P_1, P_2, P_3.

  The Gantt chart for the schedule is:



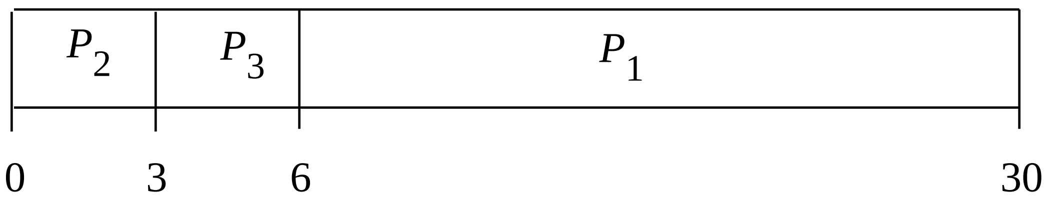
- Waiting time for: $P_1 = 0$
  $P_2 = 24$
  $P_3 = 27$

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

- Suppose that the processes arrive in the order:

  P_2, P_3, P_1.

  The Gantt chart for the schedule is:



- Waiting time for: $P_1 = 6$
  $P_2 = 0$
  $P_3 = 3$

- Average waiting time: $(6 + 0 + 3)/3 = 3$

- Much better than previous case.

- *Convoy effect*: short process behind long process

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

- Two schemes:

  a) nonpreemptive – once CPU given to the process it cannot be preempted until it completes its CPU burst.

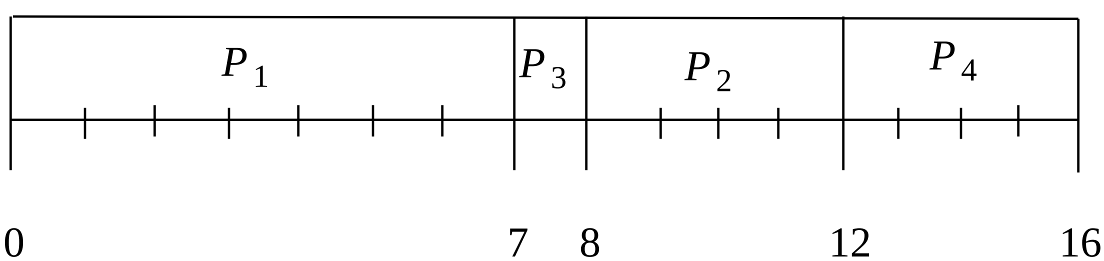
  b) preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF).

- SJF is optimal – gives minimum average waiting time for a given set of processes.

Example of SJF

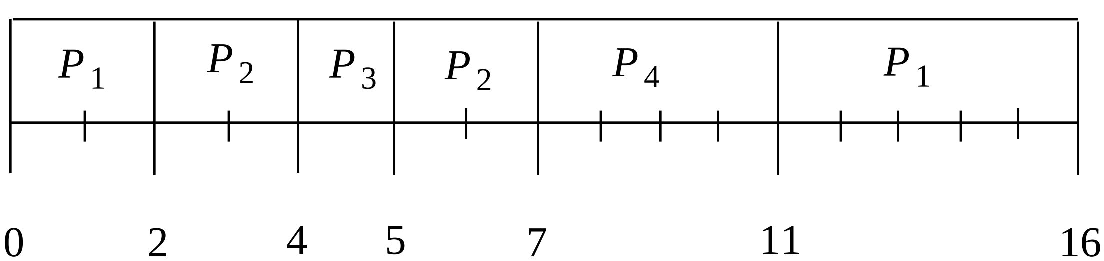
Process	Arrival time	CPU time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- SJF (non-preemptive)



Average waiting time = (0 + 6 + 3 + 7)/4 = 4

- SRTF (preemptive)



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

How do we know the length of the next CPU burst?

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

  1. T_n = actual length of n^{th} CPU burst
  2. ψ_n = predicted value of n^{th} CPU burst
  3. $0 \leq W \leq 1$
  4. Define:

$$\psi_{n+1} = W * T_n + (1 - W)\, \psi_n$$

Examples:

- $W = 0$

  $\psi_{n+1} = \psi_n$

  Recent history does not count.

- $W = 1$

  $\psi_{n+1} = T_n$

  Only the actual last CPU burst counts.

- If we expand the formula, we get:

$$\psi_{n+1} = W * T_n + (1 - W) * W * T_{n-1} + (1 - W)^2 * W * T_{n-2} + \ldots + (1 - W)^q * W * T_{n-q}$$

So if $W = 1/2 \Rightarrow$ each successive term has less and less weight.

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).

  a) preemptive

  b) nonpreemptive

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

- Problem ≡ Starvation – low priority processes may never execute.

  Solution ≡ Aging – as time progresses increase the priority of the process.

Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10–100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.

- Performance

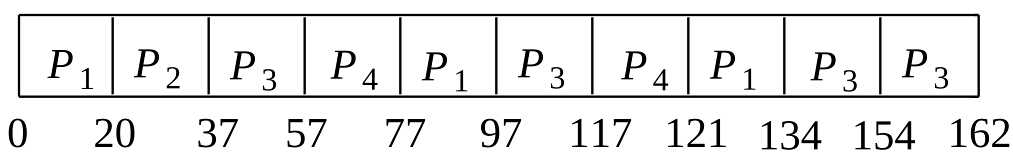
  q large $\Rightarrow$ FIFO

  q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high.

Example of RR with time quantum = 20

Process	CPU times
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:



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

Multilevel Queue

- Ready queue is partitioned into separate queues.

  Example: foreground (interactive)
  background (batch)

- Each queue has its own scheduling algorithm.

  Example: foreground – RR
  background – FCFS

- Scheduling must be done between the queues.
  - Fixed priority scheduling

    Example: serve all from foreground then from background. Possibility of starvation.

  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes.

    Example:

    80% to foreground in RR

    20% to background in FCFS

Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way.

- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithm for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of multilevel feedback queue

- Three queues:
  - Q_0 – time quantum 8 milliseconds
  - Q_1 – time quantum 16 milliseconds
  - Q_2 – FCFS

- Scheduling

  A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1. At Q_1, job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

- Multiple-Processor Scheduling
  - CPU scheduling more complex when multiple CPUs are available.
  - *Homogeneous* processors within a multiprocessor.
  - *Load sharing*
  - *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing.

- Real-Time Scheduling
  - *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time.
  - *Soft real-time* computing – requires that critical processes receive priority over less fortunate ones.

Algorithm Evaluation

- *Deterministic modeling* – takes a particular predetermined workload and defines the performance of each algorithm for that workload.
- Queueing models
- Implementation