UNIT V

Design principle

Goals

Based on market requirements and Microsoft's development strategy, the original Microsoft NT design team established a set of prioritized goals. Note that from the outset, the priority design objectives of the Windows NT operating system were *robustness* and *extensibility*:

Robustness. The operating system must actively protect itself from internal malfunction and external damage (whether accidental or deliberate), and must respond predictably to software and hardware errors. The system must be straightforward in its architecture and coding practices, and interfaces and behavior must be well- specified.

Extensibility and maintainability. Windows NT must be designed with the future in mind. It must grow to meet the future needs of original equipment manufacturers (OEMs) and Microsoft. And the system must be designed for maintainability, it must accommodate changes and additions to the API sets it supports and the APIs should not employ flags or other devices that drastically alter their functionality.

Portability. The system architecture must be able to function on a number of platforms with minimal recoding.

Performance. Algorithms and data structures that lead to a high level of performance and that provide the flexibility needed to achieve our other goals must be incorporated into the design.

POSIX compliance and government certifiable C2 security. The POSIX standard calls for operating system vendors to implement UNIX-style interfaces so that applications can be moved easily from one system to another. U.S. government security guidelines specify certain protections, such as auditing capabilities, access detection, per-user resource quotas, and resource protection. Inclusion of these features would allow Windows NT to be used in government operations.

Mechanisms and polices

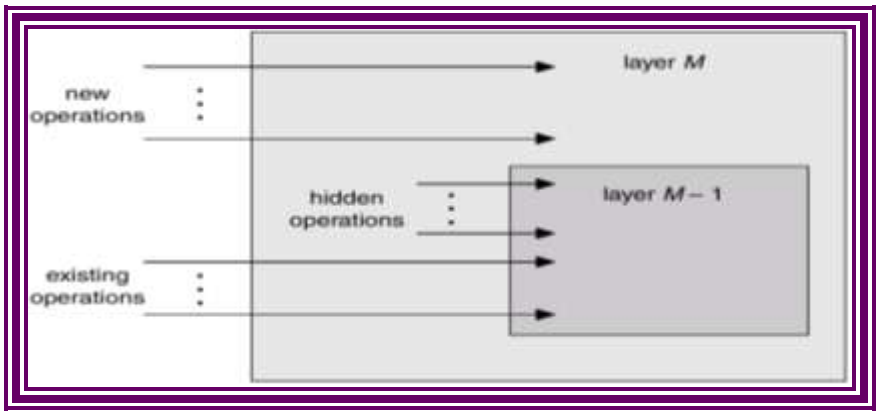
A **policy** is a plan of action to guide decisions and actions. The term may apply to government, private sector organizations and groups, and individuals. The policy process includes the identification of different alternatives, such as programs or spending priorities, and choosing among them on the basis of the impact they will have. Policies can be understood as political, management, financial, and administrative mechanisms arranged to reach explicit goals.

The separation of policy and mechanism is very important for flexibility. Policies are likely to change from place to place or time to time. A general mechanism would be more desirable.

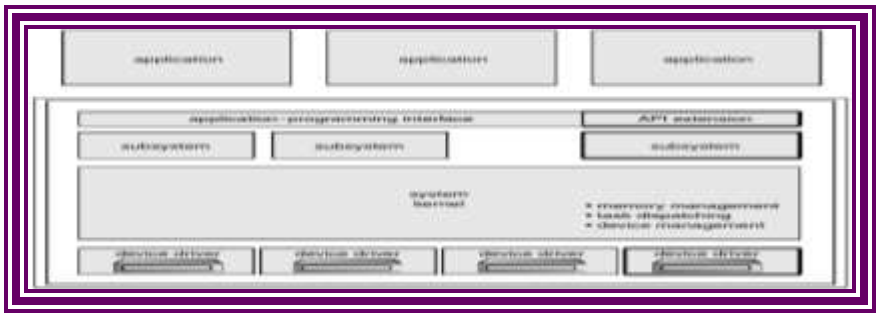
Layered approach

- 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.

An Operating System Layer



OS/2 Layer Structure

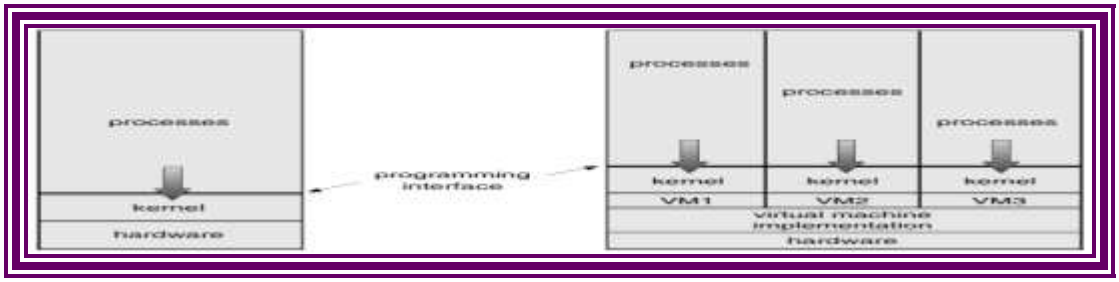


Virtual Machines

- 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.

System Models



Advantages/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 to the underlying machine.

Multiprocessor

A **multiprocessor** computer is one with more than one CPU. The category of multiprocessor computers can be divided into the following sub-categories:

- **shared memory multiprocessors** have multiple CPUs, all with access to the same memory. Communication between the the processors is easy to implement, but care must be taken so that memory accesses are synchronized.
- **distributed memory multiprocessors** also have multiple CPUs, but each CPU has it's own associated memory. Here, memory access synchronization is not a problem, but communication between the processors is often slow and complicated.

Related to multiprocessors are the following:

- **networked systems** consist of multiple computers that are networked together, usually with a common operating system and shared resources. Users, however, are aware of the different computers that make up the system.
- **distributed systems** also consist of multiple computers but differ from networked systems in that the multiple computers are transparent to the user. Often there are

redundant resources and a sharing of the workload among the different computers, but this is all transparent to the user.

System Implementation

1. Traditionally written in assembly language, operating systems can now be written in higher-level languages.
2. Code written in a high-level language:

    - can be written faster.
    - is more compact.
    - is easier to understand and debug.

3. 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.

Distributed system

Motivation

1. **Distributed system** is collection of loosely coupled processors interconnected by a communications network
2. Processors variously called *nodes, computers, machines, hosts*

    - *Site* is location of the processor

3. Reasons for distributed systems
    - Resource sharing
        - sharing and printing files at remote sites
        - processing information in a distributed database
        - using remote specialized hardware devices
    - Computation speedup – **load sharing**
    - Reliability – detect and recover from site failure, function transfer, reintegrate failed site
    - Communication – message passing

Network Topology

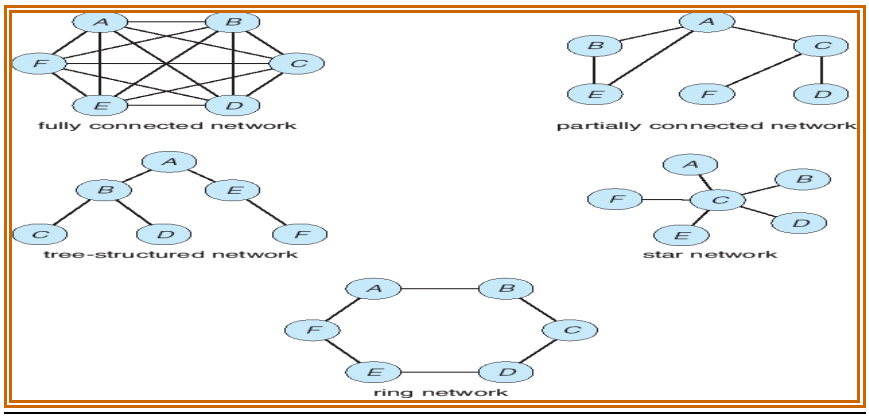
1. Sites in the system can be physically connected in a variety of ways; they are compared with respect to the following criteria:

    - **Basic cost** - How expensive is it to link the various sites in the system?
    - **Communication cost** - How long does it take to send a message from site *A* to site *B*?
    - **Reliability** - If a link or a site in the system fails, can the remaining sites still communicate with each other?

2. The various topologies are depicted as graphs whose nodes correspond to sites

    - An edge from node *A* to node *B* corresponds to a direct connection between the two sites

3. The following six items depict various network topologies



Communication Structure

The design of a *communication* network must address four basic issues:

- **Naming and name resolution** - How do two processes locate each other to communicate?
- **Routing strategies** - How are messages sent through the network?
- **Connection strategies** - How do two processes send a sequence of messages?
- **Contention** - The network is a shared resource, so how do we resolve conflicting demands for its use?

Naming and Name Resolution

- Name systems in the network
- Address messages with the process-id
- Identify processes on remote systems by

<host-name, identifier> pair

- *Domain name service* (DNS) – specifies the naming structure of the hosts, as well as name to address resolution (Internet)

Routing Strategies

1. **Fixed routing** - A path from *A* to *B* is specified in advance; path changes only if a hardware failure disables it

    - Since the shortest path is usually chosen, communication costs are minimized
    - Fixed routing cannot adapt to load changes
    - Ensures that messages will be delivered in the order in which they were sent

2. **Virtual circuit** - A path from *A* to *B* is fixed for the duration of one session. Different sessions involving messages from *A* to *B* may have different paths
    - Partial remedy to adapting to load changes
    - Ensures that messages will be delivered in the order in which they were sent

Dynamic routing - The path used to send a message form site *A* to site *B* is chosen only when a message is sent

- Usually a site sends a message to another site on the link least used at that particular time
- Adapts to load changes by avoiding routing messages on heavily used path
- Messages may arrive out of order
- This problem can be remedied by appending a sequence number to each message

Connection Strategies

1. **Circuit switching** - A permanent physical link is established for the duration of the communication (i.e., telephone system)
2. **Message switching** - A temporary link is established for the duration of one message transfer (i.e., post-office mailing system)
3. **Packet switching** - Messages of variable length are divided into fixed-length packets which are sent to the destination

- Each packet may take a different path through the network
- The packets must be reassembled into messages as they arrive

4. Circuit switching requires setup time, but incurs less overhead for shipping each message, and may waste network bandwidth

    - Message and packet switching require less setup time, but incur more overhead per message

Contention

Several sites may want to transmit information over a link simultaneously. Techniques to avoid repeated collisions include:

1. **CSMA/CD** - Carrier sense with multiple access (CSMA); collision detection (CD)

    - A site determines whether another message is currently being transmitted over that link. If two or more sites begin transmitting at exactly the same time, then they will register a CD and will stop transmitting
    - When the system is very busy, many collisions may occur, and thus performance may be degraded

2. CSMA/CD is used successfully in the Ethernet system, the most common network system
3. **Token passing** - A unique message type, known as a token, continuously circulates in the system (usually a ring structure)

    - A site that wants to transmit information must wait until the token arrives
    - When the site completes its round of message passing, it retransmits the token
    - A token-passing scheme is used by some IBM and HP/Apollo systems

4. **Message slots** - A number of fixed-length message slots continuously circulate in the system (usually a ring structure)

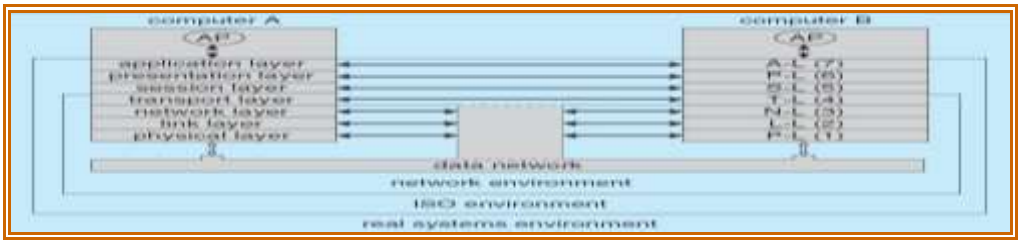
    - Since a slot can contain only fixed-sized messages, a single logical message may have to be broken down into a number of smaller packets, each of which is sent in a separate slot
    - This scheme has been adopted in the experimental Cambridge Digital Communication Ring

Communication Protocol

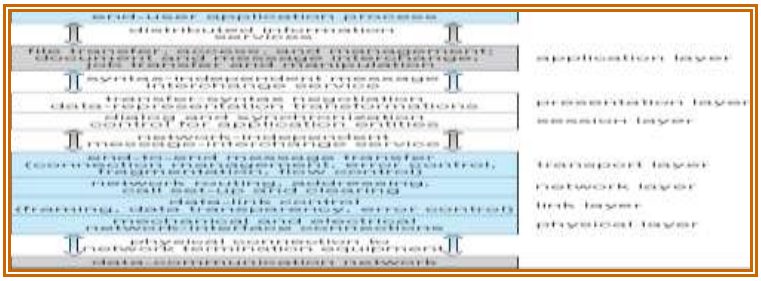
The communication network is partitioned into the following multiple layers:

- **Physical layer** – handles the mechanical and electrical details of the physical transmission of a bit stream
- **Data-link layer** – handles the *frames*, or fixed-length parts of packets, including any error detection and recovery that occurred in the physical layer
- **Network layer** – provides connections and routes packets in the communication network, including handling the address of outgoing packets, decoding the address of incoming packets, and maintaining routing information for proper response to changing load levels
- **Transport layer** – responsible for low-level network access and for message transfer between clients, including partitioning messages into packets, maintaining packet order, controlling flow, and generating physical addresses
- **Session layer** – implements sessions, or process-to-process communications protocols
- **Presentation layer** – resolves the differences in formats among the various sites in the network, including character conversions, and half duplex/full duplex (echoing)
- **Application layer** – interacts directly with the users' deals with file transfer, remote-login protocols and electronic mail, as well as schemas for distributed databases

Communication Via ISO Network Model

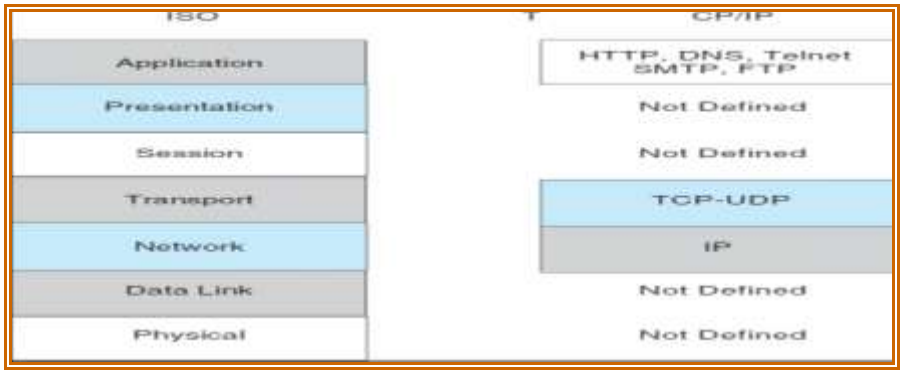


The ISO Protocol Layer



The ISO Network Message

The TCP/IP Protocol Layers



File Concept

1. Contiguous logical address space
2. Types:
    - Data
        - numeric
        - character
        - binary
    - Program

File Structure

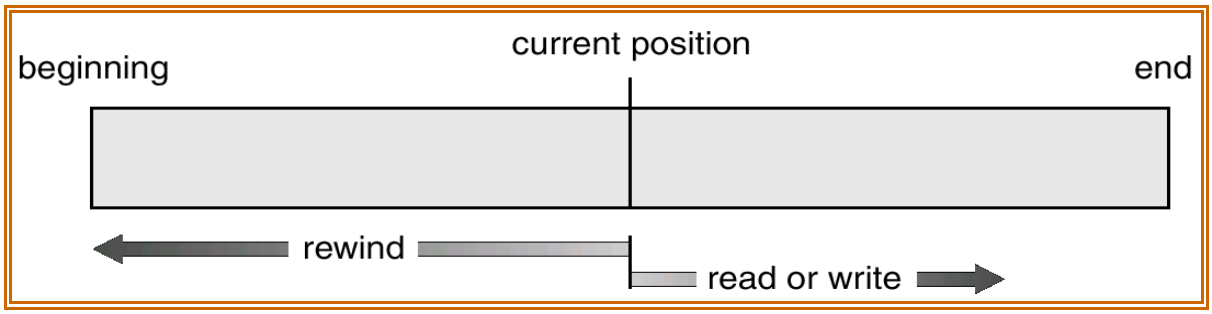
1. None - sequence of words, bytes
2. Simple record structure
    - Lines
    - Fixed length
    - Variable length
3. Complex Structures

    - Formatted document
    - Relocatable load file
4. Can simulate last two with first method by inserting appropriate control characters
5. Who decides:
    - Operating system
    - Program

Modes of computation

- Sequential Access
    - read next
    - write next
    - reset
    - no read after last write
        - (rewrite)
- Direct Access
    - read *n*
    - write *n*
    - position to *n*
        - read next
        - write next
    - rewrite *n*
- *n* = relative block number

Sequential-access File

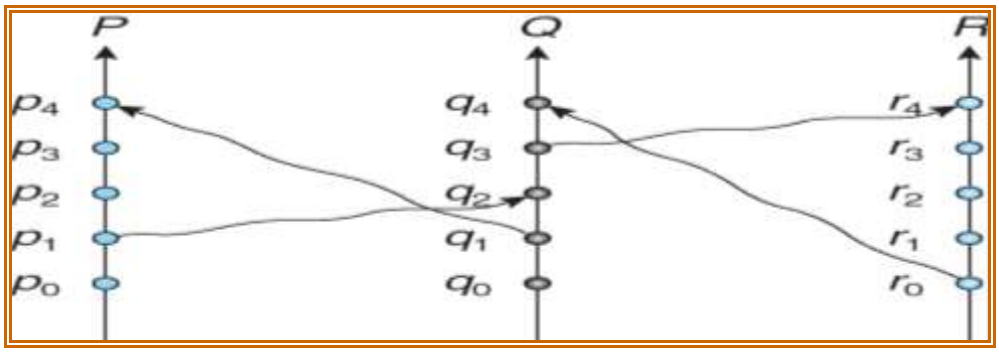


Event Ordering

1. *Happened-before* relation (denoted by $\rightarrow$)

- If A and B are events in the same process, and A was executed before B, then $A \rightarrow B$
- If A is the event of sending a message by one process and B is the event of receiving that message by another process, then $A \rightarrow B$
- If $A \rightarrow B$ and B $\rightarrow C$ then $A \rightarrow C$

Relative Time for Three Concurrent Processes



Implementation of →

1. Associate a timestamp with each system event
    - Require that for every pair of events A and B, if A → B, then the timestamp of A is less than the timestamp of B
2. Within each process Pi a logical clock, LCi is associated
    - The logical clock can be implemented as a simple counter that is incremented between any two successive events executed within a process
        - Logical clock is monotonically increasing
3. A process advances its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock
4. If the timestamps of two events A and B are the same, then the events are concurrent
    - We may use the process identity numbers to break ties and to create a total ordering

Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
- Uses *adaptive mutexes* for efficiency when protecting data from short code segments.
- Uses *condition variables* and *readers-writers* locks when longer sections of code need access to data.
- Uses *turnstiles* to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock.

Deadlock handling

Deadlock Prevention

1. Resource-ordering deadlock-prevention – define a *global* ordering among the system resources
    - Assign a unique number to all system resources

- A process may request a resource with unique number *i* only if it is not holding a resource with a unique number grater than *i*
- Simple to implement; requires little overhead

2. Banker's algorithm – designate one of the processes in the system as the process that maintains the information necessary to carry out the Banker's algorithm
    - Also implemented easily, but may require too much overhead

Timestamped Deadlock-Prevention Scheme

1. Each process *Pi* is assigned a unique priority number
2. Priority numbers are used to decide whether a process *Pi* should wait for a process *Pj*; otherwise *Pi* is rolled back
3. The scheme prevents deadlocks
    - For every edge $Pi \rightarrow Pj$ in the wait-for graph, *Pi* has a higher priority than *Pj*
    - Thus a cycle cannot exist

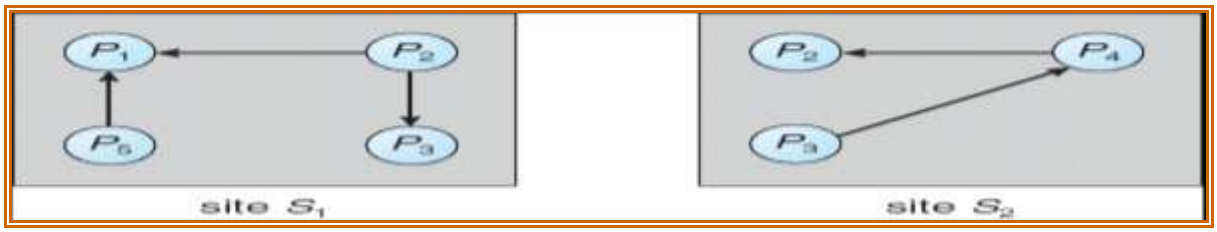
Wait-Die Scheme

1. Based on a nonpreemptive technique
2. If *Pi* requests a resource currently held by *Pj*, *Pi* is allowed to wait only if it has a smaller timestamp than does *Pj* (*Pi* is older than *Pj*)
    a. Otherwise, P*i* is rolled back (dies)
3. Example: Suppose that processes *P*1, *P*2, and *P*3 have timestamps t, 10, and 15 respectively
    a. if *P*1 request a resource held by *P*2, then *P*1 will wait
    b. If *P*3 requests a resource held by *P*2, then *P*3 will be rolled back

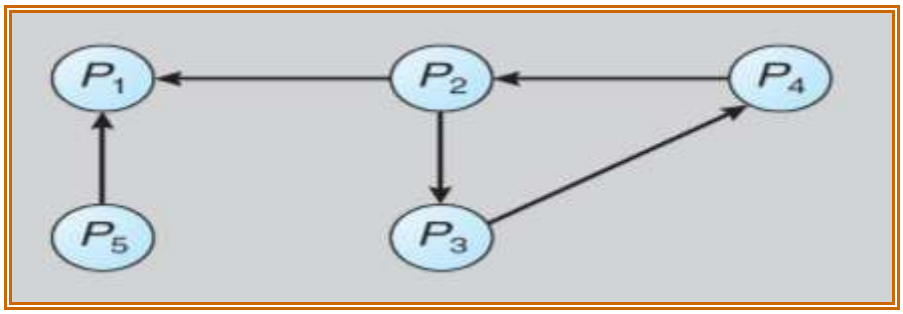
Would-Wait Scheme

1) Based on a preemptive technique; counterpart to the wait-die system
2) If *Pi* requests a resource currently held by *Pj*, *Pi* is allowed to wait only if it has a larger timestamp than does *Pj* (*Pi* is younger than *Pj*). Otherwise *Pj* is rolled back (*Pj* is wounded by *Pi*)
3) Example: Suppose that processes *P*1, *P*2, and *P*3 have timestamps 5, 10, and 15 respectively
    a) If *P*1 requests a resource held by *P*2, then the resource will be preempted from *P*2 and *P*2 will be rolled back
    b) If *P*3 requests a resource held by *P*2, then *P*3 will wait

Two Local Wait-For Graphs



Global Wait-For Graph



Deadlock Detection – Centralized Approach

1) Each site keeps a local wait-for graph
   a) The nodes of the graph correspond to all the processes that are currently either holding or requesting any of the resources local to that site
2) A global wait-for graph is maintained in a single coordination process; this graph is the union of all local wait-for graphs
3) There are three different options (points in time) when the wait-for graph may be constructed:

1. Whenever a new edge is inserted or removed in one of the local wait-for graphs
2. Periodically, when a number of changes have occurred in a wait-for graph
3. Whenever the coordinator needs to invoke the cycle-detection algorithm
   - Unnecessary rollbacks may occur as a result of false cycles
   - Append unique identifiers (timestamps) to requests form different sites
   - When process *Pi*, at site *A*, requests a resource from process *Pj*, at site *B*, a request message with timestamp *TS* is sent
   - The edge $Pi \rightarrow Pj$ with the label *TS* is inserted in the local wait-for of *A*. The edge is inserted in the local wait-for graph of *B* only if *B* has received the request message and cannot immediately grant the requested resource

The Algorithm

The controller sends an initiating message to each site in the system
2. On receiving this message, a site sends its local wait-for graph to the coordinator
3. When the controller has received a reply from each site, it constructs a graph as follows:

(a) The constructed graph contains a vertex for every process in the system

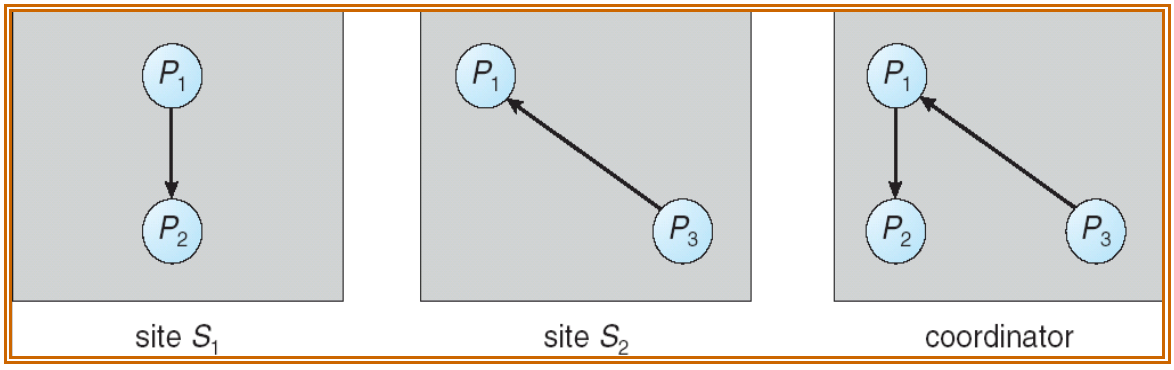
(b) The graph has an edge Pi → Pj if and only if

(1) there is an edge Pi → Pj in one of the wait-for graphs, or

(2) an edge Pi → Pj with some label TS appears in more than one wait-for graph

If the constructed graph contains a cycle ⇒ deadlock

Local and Global Wait-For Graphs



Election Algorithms

- Determine where a new copy of the coordinator should be restarted
- Assume that a unique priority number is associated with each active process in the system, and assume that the priority number of process *Pi* is *i*
- Assume a one-to-one correspondence between processes and sites
- The coordinator is always the process with the largest priority number. When a coordinator fails, the algorithm must elect that active process with the largest priority number
- Two algorithms, the bully algorithm and a ring algorithm, can be used to elect a new coordinator in case of failures

Bully Algorithm

- Applicable to systems where every process can send a message to every other process in the system
- If process *Pi* sends a request that is not answered by the coordinator within a time interval *T*, assume that the coordinator has failed; *Pi* tries to elect itself as the new coordinator
- *P*i sends an election message to every process with a higher priority number, *Pi* then waits for any of these processes to answer within *T*
- If no response within *T*, assume that all processes with numbers greater than i have failed; *Pi* elects itself the new coordinator
- If answer is received, *Pi* begins time interval *T´*, waiting to receive a message that a process with a higher priority number has been elected

- If no message is sent within *T´*, assume the process with a higher number has failed; *Pi* should restart the algorithm
- If *Pi* is not the coordinator, then, at any time during execution, *Pi* may receive one of the following two messages from process *Pj*
- *Pj* is the new coordinator (*j* > *i*). *Pi*, in turn, records this information
- *Pj* started an election (*j* > *i*). *Pi*, sends a response to *Pj* and begins its own election algorithm, provided that *Pi* has not already initiated such an election
- After a failed process recovers, it immediately begins execution of the same algorithm
- If there are no active processes with higher numbers, the recovered process forces all processes with lower number to let it become the coordinator process, even if there is a currently active coordinator with a lower number

Ring Algorithm

- Applicable to systems organized as a ring (logically or physically)
- Assumes that the links are unidirectional, and that processes send their messages to their right neighbors
- Each process maintains an active list, consisting of all the priority numbers of all active processes in the system when the algorithm ends
- If process Pi detects a coordinator failure, I creates a new active list that is initially empty. It then sends a message elect(i) to its right neighbor, and adds the number i to its active list
- If *Pi* receives a message elect(*j*) from the process on the left, it must respond in one of three ways:
    - If this is the first *elect* message it has seen or sent, *Pi* creates a new active list with the numbers *i* and *j*
        - It then sends the message *elect(i)*, followed by the message *elect(j)*
    - If $i \neq j$, then the active list for *Pi* now contains the numbers of all the active processes in the system
        - *Pi* can now determine the largest number in the active list to identify the new coordinator process
    - If *i* = *j*, then *Pi* receives the message *elect(i)*
        - The active list for *Pi* contains all the active processes in the system
            - *Pi* can now determine the new coordinator process.

Reaching Agreement

1) There are applications where a set of processes wish to agree on a common "value"
2) Such agreement may not take place due to:
    a) Faulty communication medium
    b) Faulty processes
        i) Processes may send garbled or incorrect messages to other processes
        ii) A subset of the processes may collaborate with each other in an attempt to defeat the scheme

Faulty Communications

1) Process *Pi* at site *A*, has sent a message to process *Pj* at site *B*; to proceed, *Pi* needs to know if *Pj* has received the message
2) Detect failures using a time-out scheme
   a) When *Pi* sends out a message, it also specifies a time interval during which it is willing to wait for an acknowledgment message form *Pj*
   b) When *Pj* receives the message, it immediately sends an acknowledgment to *Pi*
   c) If *Pi* receives the acknowledgment message within the specified time interval, it concludes that *Pj* has received its message
      i) If a time-out occurs, *Pj* needs to retransmit its message and wait for an acknowledgment
   d) Continue until *Pi* either receives an acknowledgment, or is notified by the system that *B* is down
3) Suppose that *Pj* also needs to know that *Pi* has received its acknowledgment message, in order to decide on how to proceed
   a) In the presence of failure, it is not possible to accomplish this task
   b) It is not possible in a distributed environment for processes *Pi* and *Pj* to agree completely on their respective states

Faulty Processes (Byzantine Generals Problem)

1) Communication medium is reliable, but processes can fail in unpredictable ways
2) Consider a system of n processes, of which no more than m are faulty
   a) Suppose that each process *Pi* has some private value of *Vi*
3) Devise an algorithm that allows each nonfaulty Pi to construct a vector $Xi = (Ai,1, Ai,2, \ldots, Ai,n)$ such that::
   a) If *Pj* is a nonfaulty process, then $Aij = Vj$.
   b) If *Pi* and *Pj* are both nonfaulty processes, then $Xi = Xj$.
4) Solutions share the following properties
   a) A correct algorithm can be devised only if $n \geq 3 \times m + 1$
   b) The worst-case delay for reaching agreement is proportionate to $m + 1$ message-passing delays

UNIX SYSTEM

History

First developed in 1969 by Ken Thompson and Dennis Ritchie of the Research Group at Bell Laboratories; incorporated features of other operating systems, especially MULTICS.The third version was written in C, which was developed at BellLabs specifically to support UNIX.The most influential of the non-Bell Labs and non-AT&T UNIX development groups — University of California at Berkeley (Berkeley Software Distributions).

– 4BSD UNIX resulted from DARPA funding to develop a standard UNIX system for government use.

– Developed for the VAX, 4.3BSD is one of the most influential versions, and has been ported to many other platforms.

UNIX Design Principles

- Designed to be a time-sharing system.
- Has a simple standard user interface (shell) that can be replaced.
- File system with multilevel tree-structured directories.
- Files are supported by the kernel as unstructured sequences of bytes.
- Supports multiple processes; a process can easily create new processes.
- High priority given to making system interactive, and providing facilities for program development.

Programmer Interface

Like most computer systems, UNIX consists of two separable parts:

1) Kernel: everything below the system-call interface and above the physical hardware.
2) Provides file system, CPU scheduling, memory management, and other OS functions through system calls.
3) System programs: use the kernel-supported system calls to provide useful functions, such as compilation and file manipulation.

User Interface

Programmers and users mainly deal with already existing systems programs: the needed system calls are embedded within the program and do not need to be obvious to the user.
The most common systems programs are file or directory

– Directory: mkdir, rmdir, cd, pwd
– File: ls, cp, mv, rm

Other programs relate to editors (e.g., emacs, vi) text formatters
(e.g., troff, TEX), and other activities.

File Manipulation

1) A file is a sequence of bytes; the kernel does not impose a structure on files.
2) Files are organized in tree-structured directories.
3) Directories are files that contain information on how to find other files.
4) Path name: identifies a file by specifying a path through the directory structure to the file.
5) Absolute path names start at root of file system
6) Relative path names start at the current directory
7) System calls for basic file manipulation: create, open, read, write, close, unlink, trunc.

8) The UNIX file system supports two main objects: files and directories.
9) Directories are just files with a special format, so the representation of a file is the basic UNIX concept.

Blocks and Fragments

Mos of the file system is taken up by data blocks.
4.2BSD uses two block sized for files which have no indirect blocks:
– All the blocks of a file are of a large block size (such as 8K), except the last.
– The last block is an appropriate multiple of a smaller fragment size (i.e., 1024) to fill out the file.
– Thus, a file of size 18,000 bytes would have two 8K blocks and one 2K fragment (which would not be filled completely).

The block and fragment sizes are set during file-system creation according to the ntended use of the file system:
– If many small files are expected, the fragment size should be small.
– If repeated transfers of large files are expected, the basic block size should be large.
The maximum block-to-fragment ratio is 8 : 1; the minimum block size is 4K (typical choices are 4096 : 512 and 8192 :
1024).

Process Management

- Representation of processes is a major design problem for operating system.
- UNIX is distinct from other systems in that multiple processes can be created and manipulated with ease.

These processes are represented in UNIX by various control blocks.
– Control blocks associated with a process are stored in the kernel.
– Information in these control blocks is used by the kernel for process control and CPU scheduling.

Memory Management

- The initial memory management schemes were constrained in size by the relatively small memory resources of the PDP machines on which UNIX was developed.
- Pre 3BSD system use swapping exclusively to handle memory contention among processes: If there is too much contention, processes are swapped out until enough memory is available.
- Allocation of both main memory and swap space is done first-fit.
- required for multiple processes using the same text segment.
- The scheduler process (or swapper) decides which processes to swap in or out, considering such factors as time idle, time in or out of main memory, size, etc.
- In f.3BSD, swap space is allocated in pieces that are multiples of power of 2 and minimum size, up to a maximum size determined by the size or the swap-space partition on the disk.

I/O System

The I/O system hides the peculiarities of I/O devices from the bulk of the kernel.
Consists of a buffer caching system, general device driver code, and drivers for specific hardware devices.
Only the device driver knows the peculiarities of a specific device.

Interprocess Communication

- Most UNIX systems have not permitted shared memory because the PDP-11 hardware did not encourage it.
- The pipe is the IPC mechanism most characteristic of UNIX.

– Permits a reliable unidirectional byte stream between two processes.
– A benefit of pipes small size is that pipe data are seldom written to disk; they usually are kept in memory by the normal block buffer cache.

- In 4.3BSD, pipes are implemented as a special case of the socket mechanism which provides a general interface not only to facilities such as pipes, which are local to one machine, but also to networking facilities.

Linux operating system

History

- Linux is a modern, free operating system based on UNIX standards
- First developed as a small but self-contained kernel in 1991 by Linus Torvalds, with the major design goal of UNIX compatibility
- Its history has been one of collaboration by many users from all around the world, corresponding almost exclusively over the Internet
- It has been designed to run efficiently and reliably on common PC hardware, but also runs on a variety of other platforms
- The core Linux operating system kernel is entirely original, but it can run much existing free UNIX software, resulting in an entire UNIX-compatible operating system free from proprietary code
- Many, varying Linux Distributions including the kernel, applications, and management tools

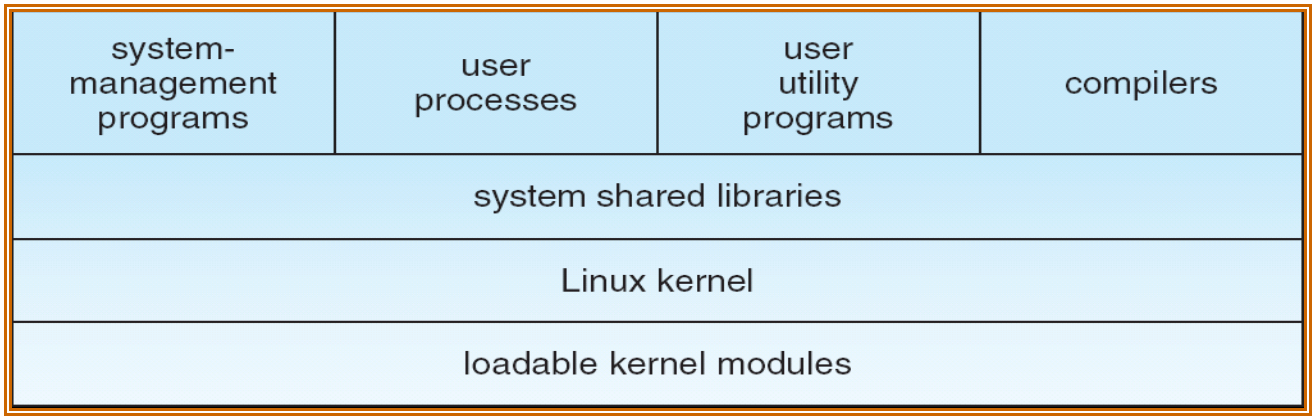
The Linux System

- Linux uses many tools developed as part of Berkeley's BSD operating system, MIT's X Window System, and the Free Software Foundation's GNU project
- The min system libraries were started by the GNU project, with improvements provided by the Linux community
- Linux networking-administration tools were derived from 4.3BSD code; recent BSD derivatives such as Free BSD have borrowed code from Linux in return
- The Linux system is maintained by a loose network of developers collaborating over the Internet, with a small number of public ftp sites acting as de facto standard repositories

Design Principles

- Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools
- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model
- Main design goals are speed, efficiency, and standardization
- Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification
- The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior

Components of a Linux System



- Like most UNIX implementations, Linux is composed of three main bodies of code; the most important distinction between the kernel and all other components
- The kernel is responsible for maintaining the important abstractions of the operating system
  - Kernel code executes in *kernel mode* with full access to all the physical resources of the computer
  - All kernel code and data structures are kept in the same single address space