Distributed systems act as resource managers for the underlying hardware, allowing users access to memory, storage, CPUs, peripheral devices, and the network.

Much of this is accomplished by operating systems and network operating systems.
Virtual Machines

- Because multiple processes can run at the same time on a hardware device, an operating system provides a virtual machine, giving the user the impression of control of a system
  - This includes protection of each user process from interference by another process
- To accomplish this, operating systems typically use two levels of access to the system resources
  - The only way to switch from one mode to another is through system calls not accessible to users

Kernel Mode

- All instructions are available
- All memory is accessible
- All registers are accessible
**User Mode**

- Memory access is restricted
  - User not allowed to access memory locations that are outside a range of addresses
- Can’t access device registers

**Network Operating Systems**

- Autonomous nodes
  - Manage their own resources
  - One system image per node
  - Machines are connected via a network
- Processes only scheduled locally
  - User action required for distribution
- E.g., Windows XP, UNIX, LINUX
Distributed Operating System

- Single (global) system image
- Processes scheduled across nodes
  - E.g., for load balancing, optimization of communication
- No true distributed O/S in use
  - Legacy problems
  - User autonomy/integrity compromised
- Solution: middleware + O/S

System Layers

- Applications, services
- Middleware
- OS: kernel, libraries & servers
- OS1: Processes, threads, communication...
- OS2: Processes, threads, communication...
- Computer & network hardware
- Node 1
- Node 2
Requirements

- Main components: kernels & server processes
- Requirements:
  - Encapsulation
    - Transparency for clients
  - Protection
    - From illegitimate accesses
  - Concurrent processing

Requirements (cont)

- Remote invocations require:
  - Communication subsystem
  - Scheduling of requests
Core O/S Components

- **Process manager**
- **Communication manager**
- **Thread manager**
- **Memory manager**
- **Supervisor**

Core O/S Functions

- **Process management**
  - Creating, managing, and destroying processes
  - Every process has an address space and one or more threads
- **Thread management**
  - Creating, synchronizing, and scheduling threads
- **Communications management**
  - All communications between threads in the same computer and may include remote processes
- **Memory management**
  - Control of physical and virtual memory
The O/S Supervisor

- The supervisor
  - Dispatches interrupts, system call traps and exceptions
  - Control of the memory management unit and hardware caches
  - Manipulation of the processor and floating point registers

Protection

- All resources must be protected from interference
- This includes protection from access by malicious code, but it also includes protection from faults and other processes that may be running on the same computer
- Protection might also prevent the bypassing of required activities such as login authentication and authorization
Kernels and Protection

- Most processors have a hardware mode register that permits privileged instructions to be strictly controlled
  - **Supervisor mode** for privileged instructions
  - **User mode** for unprivileged instructions
- Separate **address spaces** are allocated
  - Only the privileged **kernel** can access the privileged spaces
  - Usually a **system call trap** is required to access privileged instructions from user space

Protection Overhead

- Protection comes at a price
  - Processor cycles to switch between address spaces
  - Supervision and protection of system call traps
  - Establishment, authentication, and authorization of privileged users and processes
- Since all overhead consumes expensive resources, it is always a key concern of IT managers
Kernel Types

- Microkernel
  - Contains code that must execute in kernel mode
  - Functions
    - Setting device registers
    - Switching the CPU between processes
    - Manipulating the MMU
    - Capturing hardware interrupts
  - Other OS functions in user mode
    - File systems, network communication

Kernel Types (cont)

- Monolithic kernel
  - Performs all O/S functions
  - Usually very large
Kernel vs. Microkernel

- Microkernel advantages
  - Testability
  - Extensibility
  - Modularity
  - Binary emulation

- Monolithic kernel advantages
  - The way most OSs operate
  - Performance

- Hybrid solutions may be ideal

Processes

- A typical process includes an execution environment and one or more threads
- An execution environment includes:
  - An address space
  - Thread synchronization
  - Communication resources such as semaphores
  - Communication interfaces such as sockets
  - Higher level resources such as open files and windows

- Many earlier versions of processes only allowed a single thread, thus the term multi-threaded process is often used for clarity
Address Spaces

- An address space is a management unit for the virtual memory of a process.
- It consists of non-overlapping regions accessible by the threads of the owning process.
- Each region has an extent (lowest virtual address and size), read/write/execute permissions for the processes threads, and whether it grows up or down.
- It is page oriented, and gaps are left between regions to allow for growth.

Address Space

- Auxiliary regions
- Stack
- Heap
- Text
Linux Address Spaces

- Address spaces are a generalization of the (L)Unix model, which had three regions:
  - A fixed, unmodifiable **text region** containing program code
  - A **heap**, extensible toward higher virtual addresses
  - A **stack**, extensible toward lower virtual addresses
- An indefinite number of additional regions have since been added

Stack

- Generally there is a separate stack for each thread
- Whenever a thread or process is interrupted, status information is stored on the stack that permits the process or thread to continue from the point at which it was interrupted
- Usually, memory allocated to the stack is recovered when the process or thread retrieves the information and resumes, as interrupts occur and resumes in a last-in, first out (LIFO) manner
File Regions

- A file stored offline can be loaded into active memory
- A **mapped file** is accessed as an array of bytes in memory
  - Such a file can reduce access overhead dramatically, as it is orders of magnitude faster to access memory than disk files

Shared Memory Regions

- Sometimes it is desirable to share memory between processes, or between a process and the kernel
- Reasons for sharing memory include:
  - Libraries that might be large and would waste memory if each process loaded a copy
  - Kernel calls that access system calls and exceptions
  - Data sharing and communication between processes on shared tasks
Shared Memory – Problems

- **Bus contention**
  - Solution: per processor cache
    - Introduces cache consistency problems
- **Software cache consistency**
  - Solution approaches:
    - Write-through cache
    - Don’t cache updateable shared segments
    - Flush cache in critical sections
    - Prevent concurrent access

Process Creation

- An operating system creates processes as needed
- In a distributed environment, there are two independent aspects of the creation process:
  - Choice of a target host
  - Creation of an execution environment and an initial thread within it
Process Management

- Process creation
  - Parent process spawns child process
  - Choice of host
    - Performed by distributed system service
  - Creation & initialization of address space

- Process control
  - Create, suspend, resume, kill

- Process migration
  - Expensive operation!

Process Management - Creation

- Creating a new execution environment
  - Standard, statically pre-defined
  - From an existing environment
    - E.g., by fork()
    - Usually physically shared text region
    - Some data regions shared or copied (copy-on-write)
Choice of Target Host

- A transfer policy decides whether to situate the process locally or remotely
- A location policy determines which node should host the new process
- Location policies may be static or adaptive
  - Static policies ignore the current state of the system and are designed based on the expected long-term characteristics of the system
  - Adaptive policies are based on unpredictable runtime factors such as load on a node

Dynamic Location Policies

- Load sharing policies use a load manager to allocate processes to hosts
  - Sender-initiated policies require the node creating the process to specify the host
  - Receiver-initiated when load is below a certain threshold, advertises for more work
  - Migratory policies can shift processes between hosts at any time
Process Management - Migration

- Expensive operation
  - Not in widespread use for load balancing
  - May be useful for maintenance
    - Long-running tasks
- Choice of host (location policy)

Creating Execution Environment

- Once a host is selected, a new process requires an address space with initialized contents and default information such as files
  - The new address space can be defined statically, or copied from an existing execution environment
  - If copied, content may be shared and nothing written to the new environment until such time as a write instruction occurs for either process
  - Then the shared content is divided. This technique is called copy-on-write (next slide)
Copy-on-write

Threads

- A single process can have more than one activity going on at the same time
  - For example, it may be performing an activity while at the same time it needs to be aware of background events
  - There may also be background activities such as loading information into a buffer from a socket or file
- Servers may service many requests from different users at the same time
Client and Server Threads

- Thread 1 generates results
- Thread 2 makes requests to server
- Requests & queuing
- N threads
- Input-output

Multithreaded Server Architectures

- **Worker Pool**
  - A predetermined fixed number of threads is available for use as needed and returned to the pool after use

- **Thread-per-request**
  - A new thread is allocated for each new request, and discarded after use

- **Thread-per-connection**
  - A new thread is allocated for each new connection
  - Several requests can use that thread sequentially
  - The thread is discarded when the connection is closed
Multithreaded Server Architectures

- **Thread-per-object**
  - A new thread is allocated for each remote object
  - All requests for that object wait to use that thread
  - The thread is discarded when the connection to the object is destroyed

Server Threads

- **a. Thread-per-request**
  - I/O
  - Workers
  - Remote objects

- **b. Thread-per-connection**
  - Per-connection threads
  - Remote objects

- **c. Thread-per-object**
  - Per-object threads
  - I/O
  - Remote objects
Client Threads

- Clients block while awaiting response
  - E.g., blocking receive()
- Schedule other threads while waiting
- Multiple server connections
- E.g., web browsers
  - Access multiple pages concurrently
  - Multiple requests for same page

What is a Thread?

- It is a program’s path of execution
  - Most programs run as a single thread, which could cause problems if a program needs multiple events or actions to occur at the same time
What is a Thread?

- Multi-threading literally means multiple lines of a single program can be executed at the same time
  - However, it is different from multi-processing because all of the threads share the same address space for both code and data, causing it to be less overhead
- So, by starting a thread an efficient path of execution is created while still sharing the original data area from the parent

Threads vs. Processes

- Both methods work
- Threads more efficient
  - Creation (no new execution environment)
  - Context switch
  - Initial page faults
  - Caching
  - Resource sharing
- Problem: concurrency control
  - Threads share an environment with one another; therefore we have the problems of a shared memory
Thread States

- **Running state**: A thread is said to be in the running state when it is being executed.
- **Ready state (Not Runnable)**: A thread in this state is ready for execution, but is not currently executed.
  - Once a thread gets access to the CPU, it gets moved to the Running state.

  ![Thread States Diagram](image)

- **Dead State**: A thread reaches the dead state when the run method has finished execution.
- **Waiting State (yielding)**:
  - In this state the thread is waiting for some action to happen.
  - Once that action happens, the thread goes into the ready state.
  - Threads in the waiting state could be sleeping, suspended, blocked, or waiting on a monitor.
Uses of Threads

- Threads are used for all sorts of applications, from general interactive drawing applications to games
  - For instance, a program is not capable of drawing pictures while reading keystrokes.
    - So the program either has to give full attention to listening to keystrokes or drawing pictures
    - Otherwise, one thread can listen to the keyboard while the other draws the pictures

Another good usage of threads is on a system with multiple CPUs or cores
- In this case, each thread runs on a separate CPU, resulting in true parallelism instead of time sharing
User Space Threads

- Kernel is unaware of threads
- Threads are managed by run-time system
- Thread information is in shared memory
- System calls have to be non-blocking
  - Extra overhead (additional system call)
- Cannot transfer control on page fault
- Thread scheduling part of the application
  - Inter-process thread scheduling impossible

Kernel Threads

- Managed by kernel
- Data structures in kernel space
- Create/destroy are system calls
- System calls and page faults not problematic
- Synchronization more expensive
Kernel vs. User Space Threads

- Kernel space threads better in I/O intensive applications
- User space threads better for fine grained parallel computing intensive applications
- Hybrid solutions possible
  - E.g., application provides scheduling hints

Example: Fast Threads

- FastThreads is a hierarchic, event-based thread scheduling system
  - It manages a kernel on a computer with one or more processors and a set of application processes
  - Each process has a user level scheduler, while the kernel allocates virtual processors to processes.
- Part a of the next slide shows a kernel allocation processes on a three-processor machine.
Scheduler Activations

A. Assignment of virtual processors to processes

B. Events between user-level scheduler & kernel

Key: P = processor; SA = scheduler activation

Kernel Notifications

- We see from previous slide:
  - The process notifies the kernel when a virtual processor is idle and no longer needed or when an extra virtual processor is needed.
  - The kernel notifies a process when a scheduler activation notifies that process’ scheduler of an event. There are four types of scheduler activation events, shown on the next slide.
User Level Schedule Notification

- A new virtual processor allocated
- Scheduler activation blocked
- Scheduler activation unblocked
- Scheduler activation preempted

Communication and Invocation

- What communication primitives does it supply?
- Which protocols does it support and how open is the communication implementation?
- What steps are taken to make communication as efficient as possible?
- What support is provided for high-latency and disconnected operation?
Invocation Overhead

- In a typical employment situation, a worker might be expected to complete a number of tasks each day that contribute toward the goals of the employer. To accomplish those tasks, the worker might also have to accomplish tasks, such as commuting to work, that do not contribute directly to the employer’s goals.
- A process also has tasks that are within the scope of the process and tasks that do not contribute directly to goals. Process invocation can be compared to commuting as an overhead expense.

Invocation

- The invocation of a local process takes place within local memory and may involve a few tens of instruction cycles.
- Invocation of a remote process involves network activities and possibly access to files, and may require billions of instruction cycles in processors running at speeds measured in gigahertz.
- These invocation activities are external to the desired processing activities and increase costs without adding value.
Delay Factors – Address Space Changes

Invocation Performance – Delay Factors

- Domain transitions
  - Address space changes
- Network communication
  - Bandwidth
  - Number of packets
- Thread scheduling
- Context switching
Latency

- Invocation costs are the delays required to set up communications and the non-goal performing overhead of invocations.
- These fixed overhead costs measure the latency of the connection.
- Substantial efforts go into minimizing and reducing latency costs in distributed applications.

Invocation Costs as a Percentage of Throughput

- As more work is accomplished for a fixed amount of overhead, the overhead is less of a concern as a percentage of total costs.
- With very small data sizes, most of the system time may be spent in overhead activities.
- With large data transfers, the overhead costs may be negligible as a percentage of total costs.
- The next slide illustrates this as a graph of RPC delays against packet size in RPC transfers.
The RPC’s run on the same machine

One way to reduce the overhead of a remote invocation is to share some of the costs
- While process invocation is expensive, some activities can be done once and then reused, while others must be done for each communication

Lightweight RPC attempts to minimize overhead by sharing process activities within parent processes that are pre-established
- Overhead costs can be reduced by as much as two-thirds
Concurrent Invocation

- Another approach to reducing communication overhead is to reduce the number of messages sent to establish the connection and to continue processing while communicating instead of suspending threads until a message is received.
- Serialized and concurrent invocations are compared on the next slide.
Serialized and Concurrent Invocation Timing

Delay Factors – Network Communication

- **Bandwidth**
- **Packet size**
  - Delay increases almost linearly
- **Packet initialization**
  - E.g., headers, checksums
- **Marshalling & data copying**
  - Across address spaces
  - Across protocol layers
- **Synchronization**
Characteristics of an Open Distributed System

- Run only that system software at each computer that is necessary for it to carry out its particular role
- Allow the software implementing any particular service to be changed independently of other facilities
- Allow for alternatives of the same service to be provided, when this is required to suit different users or applications
- Introduce new services without harming the integrity of existing ones

Kernel Architecture

- Monolithic kernels that serve all the functions of the O/S may not be ideally suited for distributed applications
  - They are massive, undifferentiated and intractable
- An alternative is to have the kernel perform the most basic abstractions and allow microkernels that can be adapted for specialized functions to manage system resources
Monolithic Kernel and Microkernel

Roll of Microkernel

The microkernel supports middleware via subsystems
Multiuser O/S and Semaphores

- With many processes in a single hardware device, multiple processes need controls to enable the CPU (or CPUs) to switch between tasks
- A semaphore can be thought of as an integer with two operations, down and up
  - The down operation checks to see if the value is > 0
    - If it is, it decrements the value and continues
    - If it is, the calling process is blocked
  - The up operation does the opposite
    - It first checks to see if there are any now blocked processes that could not complete earlier

Semaphores

- If so, it unblocks one of them and continues
- Otherwise, it increments the semaphore value
- Once an up or down operation begins, no other operation can access the semaphore until the operation is complete or a process blocks
- Semaphores are not a complete solution to preventing processes from interfering with each other
- For that reason, we have monitors
Monitors

- A monitor is a construct similar to an object, containing variables and procedures
  - The variables can only be accessed by calling one of the procedures
  - The monitor allows only one process at a time to execute a procedure

What Can the O/S Do For Me?

- In a course on distributed systems, it is logical to look at operating systems and network operating systems from the perspective of the services we require for a distributed system
  - What does a server require?
  - What does a client require?
What Does a Server Do?

- Waits for client requests
- Serves many requests at the same time
- Gives priority to important tasks
- Manages background tasks
- Keeps on going like the Energizer bunny
- Gobbles up memory, CPU cycles, and disk space

Care and Feeding of a Server

- Servers tend to need a high level of concurrency
  - This requires task management, which is best done by an operating system
  - Using the stack concept, we separate support services from the application services that the server performs
  - Many services are provided by Operating Systems (OS) and their extension, Network Operating Systems (NOS)
Basic Services of an O/S

- Task Preemption
- Task Priority
- Semaphores
- Local/Remote Inter Process Communications (IPC)
- File System Management
- Security Features
- Threads
- Intertask Protection
- High Performance Multi-user Files
- Memory Management
- Dynamic Run-Time Extensions

Extended Services

- Extended services are a category of middleware that exploit the potential of networks to distribute applications
  - DBMS, TP monitors, and objects
  - Distributed computing environment
  - Network operating systems
  - Communications services
Service Needs

- Ubiquitous communications
- Access to network file and print services
- Binary large objects (BLOBS)
- Global directories and Network Yellow Pages
- Authentication and authorization services
- System management
- Network time
- Database and transaction services
- Internet services
- Object-oriented services

Summary

- Middleware and O/S
- Kernels
  - Monolithic vs. microkernels
- Processes and threads
- RPC
  - Delays
  - Lightweight RPC