Learning Objectives

Know what a modern operating system does to support distributed applications and middleware
- Definition of network OS
- Definition of distributed OS

Understand the relevant abstractions and techniques, focusing on:
- processes, threads, and support for invocation mechanisms.

Understand the options for operating system architecture
- monolithic and micro-kernels

If time:
- Lightweight RPC
System Layers

What is the impact of particular OS mechanisms on middleware’s ability to deliver distributed resource sharing to the users?

Applications, services

Middleware

OS: kernel, libraries & servers

OS1
Processes, threads, communication, ...

Computer & network hardware

Node 1

Middleware and the Operating System

Middleware implements abstractions that support network-wide programming. Examples:

- RPC and RMI (Sun RPC, Corba, Java RMI)
- event distribution and filtering (Corba Event Notification)
- resource discovery for mobile and ubiquitous computing
- support for multimedia streaming

Traditional OS’s (e.g. early Unix, Windows 3.0)
  - simplify, protect and optimize the use of local resources

Network OS’s (e.g. Mach, modern UNIX, Windows NT)
  - do the same but they also support a wide range of communication standards and enable remote processes to access (some) local resources (e.g. files); telnet, rlogin.
Distributed OS vs. Network OS

What is a distributed OS?
– Presents users (and applications) with an integrated computing platform (a single system image) that hides the individual computers.
– Has control over all of the nodes (computers) in the network and allocates their resources to tasks without user involvement.
  • In a distributed OS, the user doesn’t know (or care) where his programs are running – scheduling is transparent
– Examples:
  • Cluster computer systems
  • V system, Sprite, Globe OS

Network OS's vs. Distributed OS
– The nodes retain autonomy in managing their own processing resources (multiple system images, one per node)
– It does not schedule processes across the nodes, requiring user involvement; e.g., UNIX, Win NT

The Support Required by OS

• OS manages the basic resources of computer systems:
  – processing, memory, persistent storage and communication.

• It is the task of an operating system to:
  – raise the programming interface for the resources to a more useful level:
    • Encapsulation: by providing abstractions of the basic resources such as: processes, unlimited virtual memory, files, communication channels
    • Protection of the resources used by applications
    • Concurrent processing to enable applications to complete their work with minimum interference from other applications
  – provide the resources needed for (distributed) services and applications to complete their task:
    • Communication – within a computer or network access provided
    • Scheduling - processors scheduled at the relevant computer(s)
Core OS functionality

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

Supervisor (hardware abstraction):
- Dispatching of interrupts, system call traps, and other exceptions.
- Control of memory management unit and hardware caches.

Kernels and Protection

Kernel:
- A program that always runs and its code is executed with complete access privileges for the physical resources on its host computer
- Sets up address space for protection

Protection
- A process cannot access memory outside its address space
- A process executes kernel code in the kernel’s address space
  - User-level address space to kernel by Interrupt or a system-call trap
  - Cost: kernel-user switching takes many clock cycles; a system-call trap is more expensive than a simple procedure call or method call (user space – user space)
Processes and Threads

Concurrency:
- Multi-threading, a thread is an OS abstraction of a schedulable activity.
- Overlapping computation with input and output, and enabling concurrent processing on multi-processors

Process
- An execution environment with one or more threads
- Process abstraction serves as a protection domain
- Execution environments is also a resource allocation unit
  - An address space
  - Thread synchronization and communication resources, semaphores, communication interface like sockets
  - Higher-level resource such as open files and windows

Process Address Space

What is a region?
- An area of contiguous virtual memory accessible by the threads of the owning process
- Region: (extent, permission, grown or not)
  - A separate stack for each thread

Regions can be shared
- kernel code (mapped at the same location)
- Libraries; mapped as a region in address spaces
- shared data & communication

UNIX fork() is expensive
- must copy process’s address space
- copy-on-write
Creation of a New Process – Choice of Process Host

Transfer policy:
- Situate a new process locally or remotely

Location policy:
- Which node should host a new process selected for transfer
  - *Sprite*: commands for executing an idle workstation in a NOW system
- Load sharing:
  - Static vs. dynamic process location
  - Centralized, hierarchical, or decentralized load managing
  - Sender-initiated vs. receiver-initiated process transferring
  - Process migration: benefit vs. cost
  - Round-Robin (RR), WRR, shortest-queue-first, least-remaining-first, etc.
  - The simplicity principle

Creation of a New Execution Environment: Copy-on-write

Process A’s address space

RA

Process B’s address space

RB

Kernel

A’s page table

b) After write

B’s page table

a) Before write

RB copied from RA
(not physically)

Adapted from Distributed Systems: Concepts and Design 4th Edn.
© Addison-Wesley Publishers 2005
Threads Concept and Implementation

Process

Thread activations

Activation stacks (parameters, local variables)

Heap (dynamic storage, objects, global variables)

'text' (program code)

System-provided resources (sockets, windows, open files)

Client and Server with Threads: Work Pool

Thread 1 generates results

Buffer

Client

Thread 2 makes requests to server (blocking RMI typically)

Receipt & queuing

Input-output

Requests

Server

N threads

The 'worker pool' architecture (fixed usually); shared queue

See Figure 4.6 for an example of this architecture programmed in Java (using TCP sockets).
Alternative Server Threading Architectures

- Implemented by the server-side ORB in CORBA
  - (a) would be useful for UDP-based service, e.g. NTP;
  - (b) is the most commonly used - matches the TCP connection model
  - (c) is used where the service is encapsulated as an object. Each object has only one thread at a time, avoiding the need for thread synchronization within objects.

How about queueing? Cost? Load sharing?

Threads versus Multiple Processes

Creating a thread is (much) cheaper than a process (~10-20 times)

Switching to a different thread in same process is (much) cheaper (5-50 times)
  - Scheduling and context switching (state saving + possible domain transition)

Threads within same process can share data and other resources more conveniently and efficiently (without copying or message passing)

But, threads within a process are not protected from each other

What key factors decide cost of thread operations?

<table>
<thead>
<tr>
<th>Execution environment</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space tables</td>
<td>Saved processor registers</td>
</tr>
<tr>
<td>Communication interfaces, open files</td>
<td>Priority and execution state (such as BLOCKED, RUNABLE, SUSPENDED)</td>
</tr>
<tr>
<td>Semaphores, other synchronization objects</td>
<td>Software interrupt handling information (e.g., UNIX Signals)</td>
</tr>
<tr>
<td>List of thread identifiers</td>
<td>Execution environment identifier</td>
</tr>
</tbody>
</table>
Java Thread Constructor and Management Methods

Methods of objects that inherit from class Thread

• **Thread(ThreadGroup group, Runnable target, String name)**
  - Creates a new thread in the SUSPENDED state, which will belong to group and be identified as name; the thread will execute the run() method of target.

• **setPriority(int newPriority), getPriority()**
  - Set and return the thread’s priority.

• **run()**
  - A thread executes the run() method of its target object, if it has one, and otherwise its own run() method (Thread implements Runnable).

• **start()**
  - Change the state of the thread from SUSPENDED to RUNNABLE.

• **sleep(int millisecs)**
  - Cause the thread to enter the SUSPENDED state for the specified time.

• **yield()**
  - Enter the READY state and invoke the scheduler. // for non-preemptive scheduling

• **destroy()**: Destroy the thread.

Java Thread Synchronization Calls

• **thread.join(int millisecs)**
  - Blocks the calling thread for up to the specified time until thread has terminated.

• **thread.interrupt()**
  - Interrupts thread, causes it to return from a blocking method call such as sleep().

• **object.wait(long millisecs, int nanosecs)**
  - Blocks the calling thread until a call made to notify() or notifyAll() on object wakes the thread, or the thread is interrupted, or the specified time has elapsed.

• **object.notify(), object.notifyAll()**
  - Wakes (respectively all or all of any) thread(s) of object.
  - Similar to the semaphore operations. E.g. a worker thread would use queue.wait() to wait for incoming requests.
Threads Implementation

Threads can be implemented:
- in the OS kernel (Win NT, Solaris, Mach)
- at user level (e.g. by a thread library: C threads, pthreads), or in the language (Ada, Java).
  + lightweight - no system calls
  + modifiable scheduler
  + low cost enables more threads to be employed
- not pre-emptive
- cannot exploit multiple processors
- page fault blocks all threads
- Java can be implemented either way
- hybrid approaches can gain some advantages of both
  • user-level hints to kernel scheduler
  • hierarchical threads (Solaris)
  • event-based (SPIN, FastThreads)

Support for Communication and Invocation

The performance of RPC and RMI mechanisms is critical for effective distributed systems.
- Typical times for 'null procedure call' (w/o parameters):
  - Local procedure call < 1 microseconds \[\text{10,000 times slower!}\]
  - Remote procedure call \[\text{~ 10 milliseconds}\]
- 'network time' (involving about 100 bytes transferred, at 100 megabits/sec.) accounts for only .01 millisecond; the remaining delays must be in OS and middleware - latency, not communication time.

What key factors determines invocation cost?
Factors Affecting RPC/RMI Performance

- marshalling/unmarshalling + operation despatch at the server
- data copying: application -> kernel space -> communication buffers
- protocol processing: for each protocol layer
- network access delays: connection setup, network latency
- thread scheduling and context switching: including kernel entry

Implementation of Invocation Mechanisms

Most invocation middleware (Corba, Java RMI, HTTP) is implemented over TCP
- For universal availability, unlimited message size and reliable transfer; see section 4.4 for further discussion of the reasons.
- Sun RPC (used in NFS) is implemented over both UDP and TCP and generally works faster over UDP

Research-based systems have implemented much more efficient invocation protocols, e.g.
- Firefly RPC (see www.cdk3.net/oss)
- Amoeba’s doOperation, getRequest, sendReply primitives (www.cdk3.net/oss)
- LRPC [Bershad et. al. 1990], described on pp. 237-9).

Concurrent and asynchronous invocations
- middleware or application doesn’t block waiting for reply to each invocation
Invocations between Address Spaces

(a) System call

Control transfer via trap instruction
Control transfer via privileged instructions

Protection domain (an execution environment)

(b) RPC/RMI (within one computer)

User 1
Thread 1
Kernel 1
User 2
Thread 2

User 1
Thread 1
Kernel
User 2
Thread 2

(c) RPC/RMI (between computers)

User 1
Thread 1
Network
Kernel 1
User 2
Thread 2
Kernel 2

1/14/2008

RPC Delay against Parameter Size

An RPC fetches a specified amount of data from a server
- Roughly proportional to the size until meeting a threshold; multiple packets will be involved

1/14/2008
**Invocation within a Computer**

Uses shared memory for inter-process communication
- Many cross-address-space invocations take place with a computer
- Arguments four times for conventional RPC (could be copied only once)

1. Copy args
2. Trap to Kernel
3. Upcall
4. Execute procedure and copy results
5. Return (trap)

**Bershad's LRPC: A Lightweight Remote Procedure Call**

LRPC: underlying communication happens to be local
Asynchronous RPC

(a) The interconnection between client and server in a traditional RPC
(b) The interaction using asynchronous RPC

Deferred Synchronous RPC

A client and server interacting through two asynchronous RPCs
One-Way RPC

Request-reply communication:
- Synchronous (blocking doOperation) vs. asynchronous communication
  - CORBA one-way RPC invocations have maybe semantics

Times for Serialized and Concurrent Invocations

Figure 6.14

Serialised invocations
- process and marshal
- Send
- transmission
- Receive
- unmarshal
- execute request
- marshal
- Send
- Receive
- unmarshal
- process results
- process args
- marshal
- Send
- Receive
- unmarshal
- process results

Concurrent invocations
- process and marshal
- Send
- Receive
- unmarshal
- execute request
- marshal
- Send
- Receive
- unmarshal
- process results
- process args
- marshal
- Send
- Receive
- unmarshal
- process results

1/14/2008

Adapted from Distributed Systems: Concepts and Design Edn. 4 © Addison Wesley Publishers 2005
Monolithic Kernel and Microkernel

The microkernel supports middleware via subsystems

Advantages and Disadvantages of Microkernel

+ flexibility and extensibility
  • services can be added, modified and debugged
  • small kernel -> fewer bugs
  • protection of services and resources is still maintained

- service invocation expensive
  - • unless LRPC is used
  - • extra system calls by services for access to protected resources
Relevant Topics Not Covered

Mach OS case study (Chapter 18)
- Halfway to a distributed OS
- The communication model is of particular interest. It includes:
  - ports that can migrate between computers and processes
  - support for RPC
  - typed messages
  - access control to ports
  - open implementation of network communication (drivers outside the kernel)

Thread programming (Section 6.4, and many other sources)
- e.g. Doug Lea's book *Concurrent Programming in Java*, (Addison Wesley 1996)

Summary

The OS provides local support for the implementation of distributed applications and middleware:
- Manages and protects system resources (memory, processing, communication)
- Provides relevant local abstractions:
  - files, processes
  - threads, communication ports

Middleware provides general-purpose distributed abstractions
- RPC, RMI, event notification

Invocation performance is important
- it can be optimized, E.g. Firefly RPC, LRPC

Microkernel architecture for flexibility
- The KISS principle ('Keep it simple – stupid!')
  - has resulted in migration of many functions out of the OS