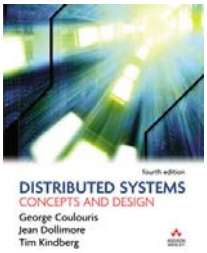
Chapter 6:
Operating System support



From Coulouris, Dollimore and Kindberg
Distributed Systems:
Concepts and Design

Edition 4, © Pearson Education 2005

Introduction

- What is an operating system (OS)?
  - To provide problem-oriented abstractions of the underlying physical resources on a single node and present them through the system call interface
- Network OS
  - Built-in networking capability
  - Multiple system images, one per node
  - Retain autonomy in managing resources on its own node
  - Users have to be involved to schedule processes across the nodes
  - E.g. UNIX, Windows NT

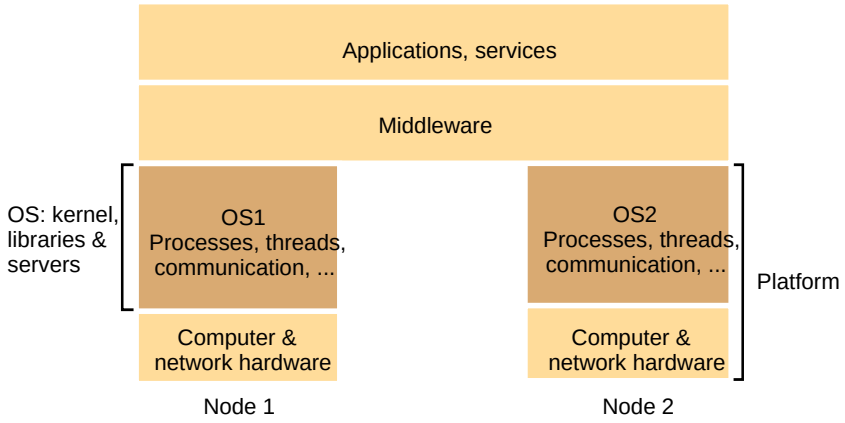
Introduction (con't)

- Distributed OS
  - One single system image
  - OS has control over all the nodes in the system
  - May transparently locate new process at whatever node suits its scheduling policies such as load balancing
  - Not available in general use
    - Users have much invested in application software
    - Users prefer to have a degree of autonomy for their machines
- Middleware + OS
  - Provide balance between autonomy and network-transparent resource access on the other

The OS Layer

- OS: runs at a node
  - Has a kernel and associated user-level services, e.g. libraries, for abstraction of local hardware resources
- Middleware: runs on a variety of OS-hardware combinations at nodes of a distributed system
  - Utilizes a combination of local resources to implement its mechanisms for remote invocations between objects or processes at the nodes
- Kernel, server, and client
  - A kernel provides services for local processes to access local resources
  - A server provides services for processes to access resources, locally and remotely
  - A client calls for services to access resources local and remote resources

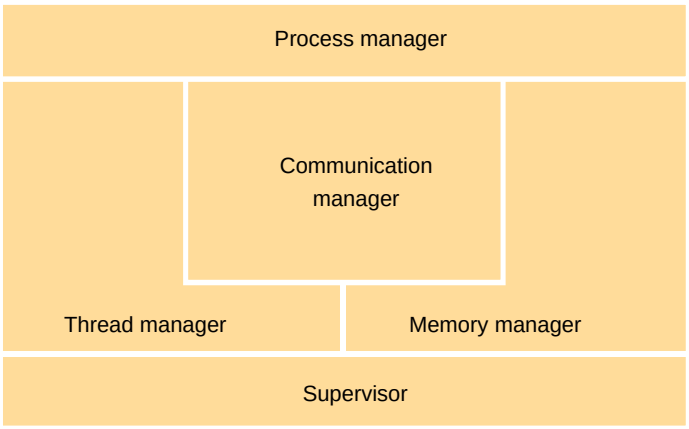
Figure 6.1
System layers



The OS Layer (con't)

- Requirements for kernels and servers
  - Encapsulation: provide service interface to resources
  - Protection: provide protection to keep resources from illegitimate accesses
  - Concurrent processing: achieve concurrency transparency, such as concurrently sharing and accessing resources
- Tasks needed for clients calling for services
  - Communication: requests and results have to be passed over a network or within a computer
  - Scheduling: When an operation is invoked, its processing must be scheduled within the kernel or server

Figure 6.2
Core OS functionality



Core OS Components

- Process manager: handles the creation and operations upon processes
- Thread manager: thread creation, synchronization and scheduling
- Communication manager: communication between threads attached to different processes on the same computer
- Memory manager: management of physical and virtual memory
- Supervisor: dispatching of interrupts, system call traps and other exceptions

Protection

- Illegitimate access
  - examples
    - Unauthorized operations on resources
    - To sidestep the exported operations
  - Need hardware support and a kernel to protect modules from one another at the level of individual invocations, regardless of the language in which they are written
- Kernel
  - A program that is distinguished by the facts that it is always runs and its code is executed with complete access privileges for the physical resources on its host computer

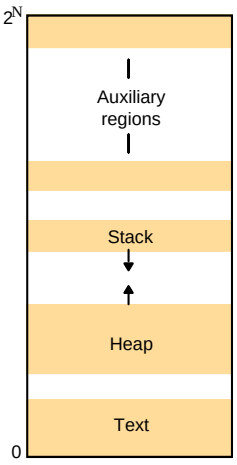
Kernel

- Executes in privileged mode; arranges other processes execute in user mode
- Set up address spaces to protect itself and other processes from accesses of an aberrant process
- A process can transfer from a user-level address space to the kernel's address space via an exception such as an interrupt or system call trap
  - When the TRAP instruction is executed, as with any type of exception, the hardware forces the processor to execute a kernel-supplied handler function, in order that no process may gain illicit control of the

Process and Thread

- A process consists of an execution environment, the unit of resource management
  - An address space
  - Activity (thread) synchronization and communication resources such as semaphores and communication interfaces
  - Higher-level resources such as open files and windows
- Thread is an activity, execution
  - Threads within same process can share resources of the execution environment
  - An execution environment provides protection from threads outside it

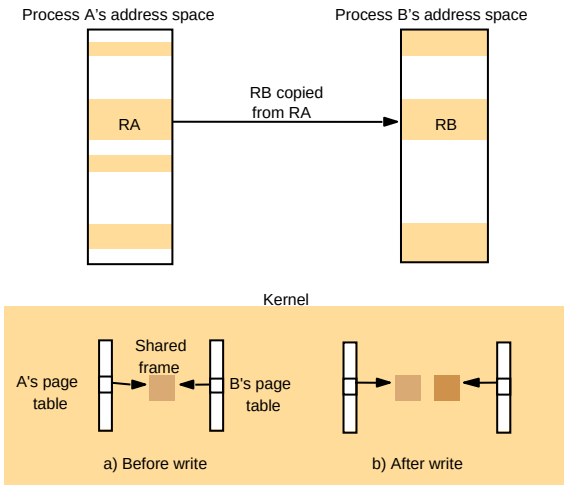
Figure 6.3
Address space



Creation of Processes

- Choice of process host
  - Policy: loads, architectures, resources, etc.
- Creation of a new execution environment
  - Initialize the address space with statically defined format
  - Initialize the address space with respect to an existing execution environment, e.g. fork in UNIX
    - Copy-on-write
    - Initially, page frames are shared between page tables and write-protected at hardware level
    - Any process's attempt to write will issue page fault
    - Kernel's page handler will replicate the frame with a new frame
    - Update page table and these two frames are writable at hardware level for two processes respectively

Figure 6.4
Copy-on-write



Threads

- Why multi-threads?
  - Performance
    - Throughput example: 2 ms processing time, 8 ms I/O access
    - Single thread and two threads on one CPU, two threads on two CPUs, and three or four threads on two CPUs
- Server threading architectures
  - Fixed threads
    - Too few or many threads, high level switching between I/O and worker
  - Thread-per-request
    - Threads do not contend for a shared queue, one thread per request
    - Overhead of thread creation and destruction
  - Thread-per-connection or thread-per-object
    - Low overhead
    - Load unbalanced; some workers have many requests while others have none

Figure 6.5
Client and server with threads

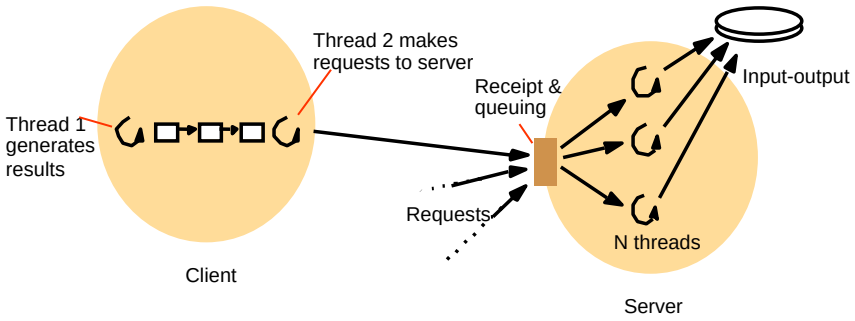
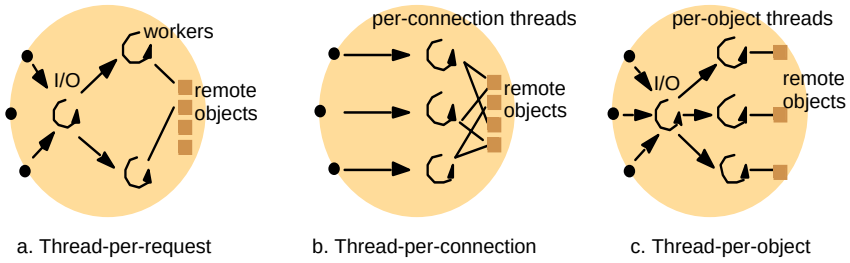


Figure 6.6
Alternative server threading architectures (see also Figure 6.5)



a. Thread-per-request b. Thread-per-connection c. Thread-per-object

Multi-Threaded vs. Multiple Processes

- Thread creation and management within an existing process are cheaper than processes
- Switching between different threads within the same process is cheaper than switching between different processes
  - Context switching
- Resources sharing can be achieved more efficiently between threads than between different processes
- But, threads within a process are not protected from one another
  - synchronization

Figure 6.7
State associated with execution environments and threads

Execution environment	*Thread*
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

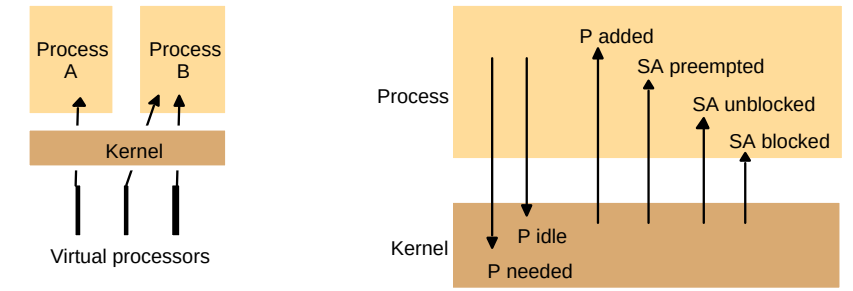
Threads Implementation

- Kernel threads
  - Managed by kernel
- User-level threads
  - Implemented as libraries and linked to applications
  - Only processes, not threads, can be recognized by kernel
  - Con:
    - Threads within a process cannot take advantage of a multiprocessor
    - A thread blocked may block entire process and all threads within it
    - Threads within different processes cannot scheduled by a single scheme
  - Pro:
    - Thread operations are less costly
    - May have application-specific scheduling policies
    - More threads can be supported at user level than kernel level

Hierarchical Scheduling

- Enable user-level code to provide scheduling hints to kernel's thread scheduler
  - Thread operations occur at user-level
  - Can take advantage of a multiprocessor
  - Application-specific scheduling policies
  - But the entire process may be blocked if a thread within it is blocked
- Example: FastThreads
  - Processes notify the kernel about processor requirements
  - The kernel notify the process about its thread status and let the process to make a scheduling decision

Figure 6.10
Scheduler activations



A. Assignment of virtual processors to processes

B. Events between user-level scheduler & kerne
Key: P = processor; SA = scheduler activatior

Performance

- Invocation costs
  - Domain transition
    - Several system calls (context switch)
  - Communication across networks
    - Fixed overhead: network latency, measured by null procedure
    - Data
      - Data transmission across network
      - Marshalling
      - Copying (across user-kernel, protocol layer, kernel buffer and network interface)
      - Packet initialization (add headers, trailers, and checksum)
    - Wait for ack
  - Thread scheduling and switching
    - Server thread is scheduled

Figure 6.11
Invocations between address spaces

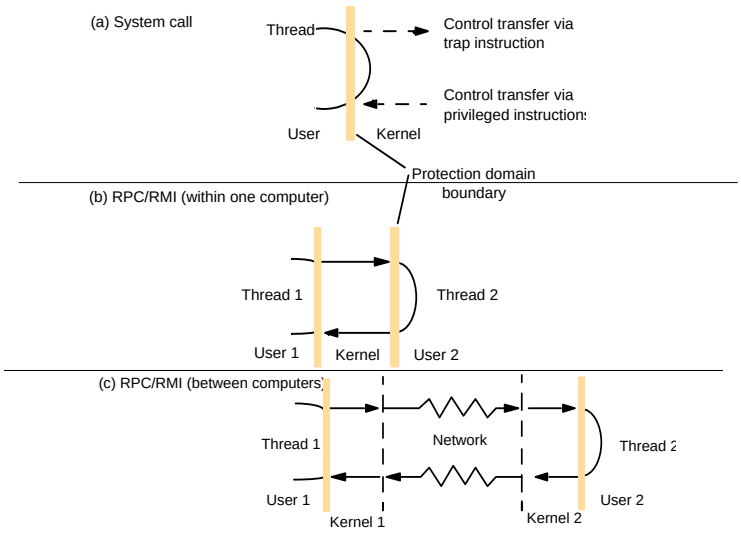
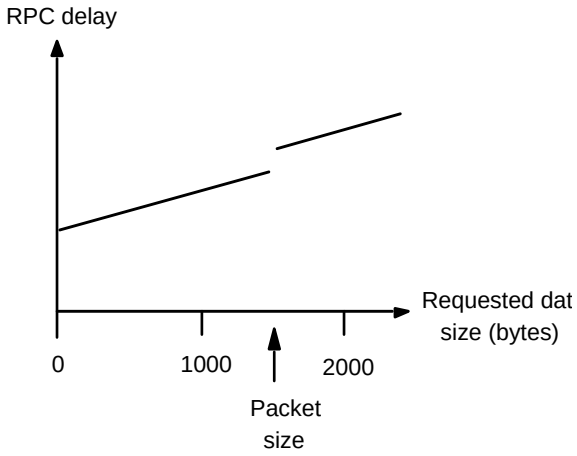


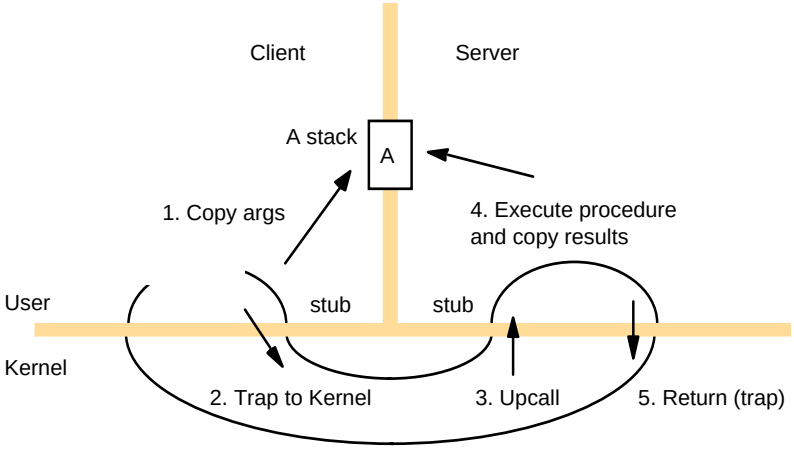
Figure 6.12
RPC delay against parameter size



Lightweight RPC (LRPC)

- To improve invocation within a computer
- Use shared memory to reduce data copying
- Let client to enter server's execution environment and call the procedure
  - Client trap to kernel via server's interface
  - Kernel check its validity to find valid server procedure
  - Kernel issues a context switch to server's execution environment
  - Execute the server procedure
  - Thus no need to block client thread and create a server thread to handle, which is needed for remote invocation

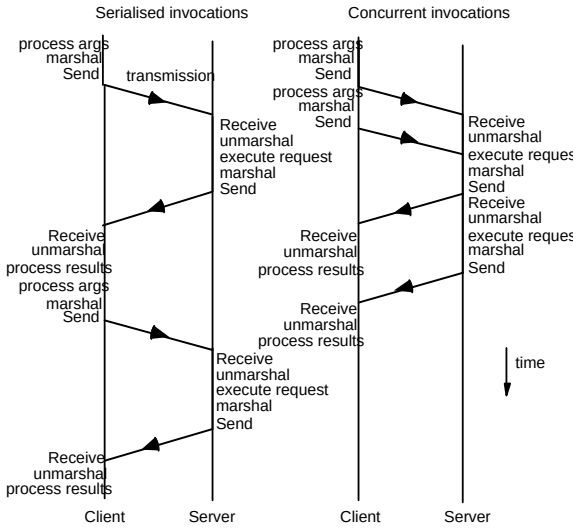
Figure 6.13
A lightweight remote procedure call



Asynchronous Operation

- To defeat high latencies
- Concurrent invocations
  - To make concurrent invocations (asynchronous), instead of serialized invocations (synchronous)
- Asynchronous invocations: CORBA *oneway*
  - Servers place status of invocation in *promise*
  - Clients make separate to collect the results
- Persistent asynchronous invocations: QRPC
  - Network may be disconnected
  - QRPC queues invocation requests when disconnected and dispatch them to servers when connected
  - It may queues results in servers until the client reconnects and collects them

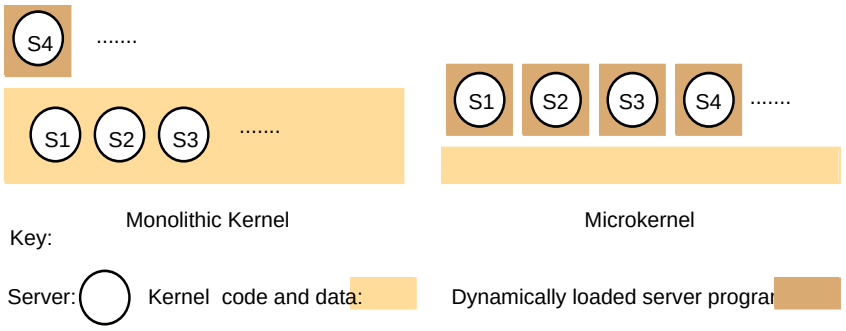
Figure 6.14
Times for serialized and concurrent invocations



OS Architecture

- OS = kernel + servers running on user-level
- Monolithic kernels and microkernels
  - The differences are primarily in the decision as to what functionality belongs in the kernel and what is left to server processes that can be dynamically loaded to run on top of it

Figure 6.15
Monolithic kernel and microkernel



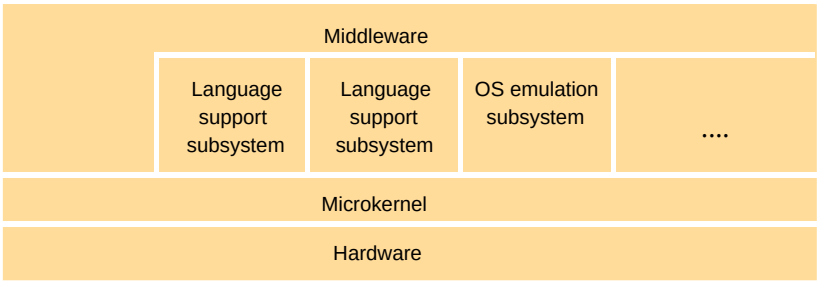
Monolithic Kernel

- Massive and in the order of megabytes of code
- Coded in a non-modular way
- Altering any individual software component to adapt it to changing requirements is difficult
- It can contain some server processes running in its address space, such as file servers and networking
- Service invocation is efficient

Microkernel

- Provides only the most basic abstraction
- System services are provided by user-level servers that are dynamically loaded when needed
- Clients access these services via the kernel
- Portability:
  - Can provide more than one system call interface–more than one *operating system*; able to provide the binary emulation of other OS
- The micorkernel lies between hardware layer and a layer consisting major system components called subsystems

Figure 6.16
The role of the microkernel



The microkernel supports middleware via subsystems

Microkernel (con't)

- To improve performance, middleware may use microkernel directly; otherwise, it uses the subsystems
- Easy to develop
  - Kernel is small, and easy to keep out of bugs
  - Modules can be easily debugged since their crashes will not hurt kernel

Some Hybrid Approaches

- Mach and Chorus
  - Allow to dynamically load servers into kernel or user address space
  - Develop at user-level, execute at kernel-level for performance
- SPIN
  - The kernel and all dynamically loaded modules execute in a single address space
  - All are written in a type-safe language (Modula-3), so they are mutually protected; i.e. protection domain within the kernel space are established using protected name spaces

Some Hybrid Approaches (con't)

- L4
  - Dynamically loaded modules execute at user-level
  - It optimizes the interprocess communication to improve the performance
- Exokernel
  - Employ user-level libraries instead of user-level servers to supply functional extensions
  - All these functionality – even a file system– to be linked into applications