Figure 6.1  System layers

Applications, services

Middleware

OS: kernel, libraries & servers

OS1
Processes, threads, communication, ...

Computer & network hardware

Node 1

OS2
Processes, threads, communication, ...

Computer & network hardware

Node 2

Platform
Figure 6.2  Core OS functionality

- Process manager
- Communication manager
- Thread manager
- Memory manager
- Supervisor
Figure 6.3  Address space

Stack

Auxiliary regions

Heap

Text

$2^N$
Figure 6.4  Copy-on-write

Process A's address space

RA

Process B's address space

RB

Kernel

A's page table

Shared frame

B's page table

a) Before write

b) After write

RB copied from RA
Figure 6.5  Client and server with threads

Thread 1 generates results

Thread 2 makes requests to server

Receipt & queuing

Requests

N threads

Input-output
Figure 6.6 Alternative server threading architectures (see also Figure 6.5)

- a. Thread-per-request
- b. Thread-per-connection
- c. Thread-per-object
**Figure 6.7  State associated with execution environments and threads**

<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 <em>BLOCKED</em>)</td>
</tr>
<tr>
<td>Semaphores, other synchronization objects</td>
<td>Software interrupt handling information</td>
</tr>
<tr>
<td>List of thread identifiers</td>
<td>Execution environment identifier</td>
</tr>
<tr>
<td>Pages of address space resident in memory; hardware cache entries</td>
<td></td>
</tr>
</tbody>
</table>
Figure 6.8  Java thread constructor and management methods

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.

destroy()
  Destroy the thread.
Figure 6.9  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, one or all of any threads that have called *wait()* on *object*.
Figure 6.10  Scheduler activations

A. Assignment of virtual processors to processes

B. Events between user-level scheduler & kernel

Key: P = processor; SA = scheduler activation
Figure 6.11 Invocations between address spaces (cont’d on next slide)

(a) System call

Control transfer via trap instruction

(b) RPC/RMI (within one computer)

Protection domain boundary
Figure 6.11 (cont’d) Invocations between address spaces

(c) RPC/RMI (between computers)
Figure 6.12  RPC delay against parameter size

- **RPC delay**
- **Requested data size (bytes)**
- **Packet size**
- **2000**
- **1000**
- **0**
Figure 6.13  A lightweight remote procedure call

1. Copy args
2. Trap to Kernel
3. Upcall
4. Execute procedure and copy results
5. Return (trap)
Figure 6.14  Times for serialised and concurrent invocations

Serialised invocations

Client

Receive unmarshal
process results

Receive unmarshal
execute request

marshal

Send

Server

Receive unmarshal
process results

Receive unmarshal
execute request

marshal

Send

Concurrent invocations

Client

Receive unmarshal
process results

Receive unmarshal
execute request

marshal

Send

Server

Receive unmarshal
process results

Receive unmarshal
execute request

marshal

Send

transmission
Figure 6.15  Monolithic kernel and microkernel

Monolithic Kernel

Key:
Server:  Kernel code and data:

Microkernel

Dynamically loaded server program:
Figure 6.16 The role of the microkernel

The microkernel supports middleware via subsystems

<table>
<thead>
<tr>
<th>Middleware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language support subsystem</td>
</tr>
<tr>
<td>Language support subsystem</td>
</tr>
<tr>
<td>OS emulation subsystem</td>
</tr>
<tr>
<td>....</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Microkernel</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Hardware</th>
</tr>
</thead>
</table>