Chapter 6: Operating System Support

- Introduction
- The operating system layer
- Protection
- Processes and Threads
- Communication and invocation
- Operating system architecture
- Summary

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, Elvin)
  - 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).
Middleware and the Operating System

○ What is a distributed OS?
  ▪ Presents users (and applications) with an integrated computing platform 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.
  ▪ Examples:
    ◆ Cluster computer systems
    ◆ V system, Sprite

Combination of middleware and network OS

○ No distributed OS in general use
  ▪ Users have much invested in their application software
  ▪ Users tend to prefer to have a degree of autonomy for their machines

○ Network OS provides autonomy

○ Middleware provides network-transparent access resource
Chapter 5: Operating System Support

- Introduction
- The operating system layer
- Protection
- Processes and Threads
- Communication and invocation
- Operating system architecture
- Summary

The relationship between OS and Middleware

- Operating System
  - Tasks: processing, storage and communication
  - Components: kernel, library, user-level services
- Middleware
  - runs on a variety of OS-hardware combinations
  - remote invocations
- Architecture
Functions that OS should provide for middleware

- **Encapsulation**
  - provide a set of operations that meet their clients’ needs
- **Protection**
  - protect resource from illegitimate access
- **Concurrent processing**
  - support clients access resource concurrently
- **Invocation mechanism: a means of accessing an encapsulated resource**
  - Communication
    - Pass operation parameters and results between resource managers
  - Scheduling
    - Schedule the processing of the invoked operation

The core OS components

- **Process manager**
  - Handles the creation of 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
  - control of memory management unit and hardware caches
  - processor and floating point unit register manipulations
- **Figure**
Chapter 5: Operating System Support

- **Introduction**
- **The operating system layer**
- **Protection**
- **Processes and Threads**
- **Communication and invocation**
- **Operating system architecture**
- **Summary**

---

Illegitimate access

- **Maliciously contrived code**
- **Benign code**
  - contains a bug
  - have unanticipated behavior
- **Example: read and write in File System**
  - Illegal user vs. access right control
  - Access the file pointer variable directly
    - (setFilePointerRandomly) vs. type-safe language
      - Type-safe language, e.g. Java or Modula-3
      - Non-type-safe language, e.g. C or C++
Kernel and Protection

- **Kernel**
  - always runs
  - complete access privileges for the physical resources

- **Different execution mode**
  - *supervisor mode (kernel process) / user mode (user process)*
  - Interface between kernel and user processes: system call trap
  - *An address space*: a collection of ranges of virtual memory locations, in each of which a specified combination of memory access rights applies, e.g.: read only or read-write

- **The cost for protection**
  - switching between different processes take many processor cycles
  - a system call trap is a more expensive operation than a simple method call

Chapter 5: Operating System Support

- **Introduction**
- **The operating system layer**
- **Protection**
- **Processes and Threads**
- **Communication and invocation**
- **Operating system architecture**
- **Summary**
Process and thread

- **Process**
  - A program in execution
  - Problem: sharing between related activities are awkward and expensive
  - Nowadays, a process consists of an *execution environment* together with one or more *threads*

- **Thread**
  - Abstraction of a single activity
  - Benefits
    - Responsiveness
    - Resource sharing
    - Economy
    - Utilization of MP architectures

Execution environment

- **the unit of resource management**
- **Consist of**
  - An address space
  - Thread synchronization and communication resources such as semaphores and communication interfaces (e.g. sockets)
  - Higher-level resources such as open files and windows
- **Shared by threads within a process**
Address space

- **Address space**
  - a unit of management of a *process*’s virtual memory
  - Up to $2^{32}$ bytes and sometimes up to $2^{64}$ bytes
  - consists of one or more regions

- **Region**
  - an area of continuous virtual memory that is accessible by the threads of the owning process

- **UNIX address space**

Address space … *continued*

- **The number of regions is indefinite**
  - Support a separate stack for each thread
  - access *mapped file*
    - `CreateFileMapping(win32),mmap(linux)`
  - Share memory between processes

- **Region can be shared**
  - Libraries
  - Kernel
  - Shared data and communication
  - Copy-on-write
Creation of new process in distributed system

- **Creating process by the operation system**
  - *Fork, exec* in UNIX

- **Process creation in distributed system**
  - The choice of a target host
  - The creation of an execution environment, an initial thread

- **Choice of process host**
  - running new processes at their originator’s computer
  - sharing processing load between a set of computers

- **Load sharing system**
  - Centralized, Decentralized, Hierarchical
  - sender-initiated, receiver-initiated

- **Load sharing policy**
  - Transfer policy: situate a new process locally or remotely?
  - Location policy: which node should host the new process?
    - Static policy without regard to the current state of the system
    - Adaptive policy applies heuristics to make their allocation decision
  - Migration policy: when & where should migrate the running process?
Creation of a new execution environment

- Initializing the address space
  - Statically defined format
  - With respect to an existing execution environment, e.g. `fork()`, `exec()` (Linux, Unix)

- Copy-on-write scheme
  - A technique that is used to reduce a mount of data copy.

---

Creation of a new execution environment

- **Linux**下典型的fork语义是这样的，
  - 新进程会把父进程的虚拟内存影射表(VM)都拷过来，也就是说创建的时候，新进程和父进程的虚拟地址空间一样，指向的虚存和物理内存都一样：这样是为了加快进程的创建速度，

  - 当然，当新进程改变内存的数据时，会采用**copy-write**的方式，去创建新的空间，不会污染父进程的地址空间内的数据
Single and Multithreaded Processes

Heavyweight process

Lightweight process

Threads concept and implementation

Process

Activation stacks
(parameters, local variables)

Heap (dynamic storage, objects, global variables)

'text' (program code)

System-provided resources
(sockets, windows, open files)
Example: client and server with threads

Given: $t_p = 2 \text{ ms}$, $t_{io} = 8 \text{ ms}$, Quest.: $T$ (maximum server throughput)?

Threads – an example

For single thread, server can handle 100 client requests per second

Client and server with threads

If all the disk request are serialized and take 8 ms, the maximum throughput is $1000/8 = 125$
Different server implementations (1, 2)

1. A single thread: $T = \frac{1000}{2+8} = 100$
   - requests are handled one by one
2. Two threads: $T = \frac{1000}{8} = 125$
   - processing on one request can overlap the disk IO of another request

Different server implementations (3, 4)

3. Two threads, disk cache (75% hit): $T = \frac{1000}{2.5} = 400$
   - $t_{io} = 0.75*0 + 0.25*8 = 2 \text{ms}$; due to search in cache, processing delay increase, say $2.5 \text{ms}$, $t_p = 2.5 \text{ms}$
4. Up to two threads, disk cache, two processors: $T = \frac{1000}{2} = 500$
   - Processing on different requests can be overlapped
**Alternative server threading architectures**

- **a. Thread-per-request**
  - (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. E.g., could have multiple shared whiteboards with one thread each. Each object has only one thread, avoiding the need for thread synchronization within objects.

---

**Architectures for multi-threaded servers**

- **Worker pool**
  - Server creates a fixed pool of “worker” threads to process the requests when it starts up
  - **Pro:** simple
  - **Cons**
    - Inflexibility: worker threads number unequal current request number
    - High level of switching between the I/O and worker thread

- **Thread-per-request**
  - Server spawn a new worker thread for each new request, destroy it when the request processing finish
  - **Pro:** throughput is potentially maximized
  - **Con:** overhead of the thread creation and destruction
Thread scheduling

- **Preemptive scheduling**
  - A thread may be suspended at any point to make way for another thread

- **Non-preemptive scheduling**
  - A thread runs until it makes a call to the threading system, when the system may de-schedule it and schedule another thread to run
    - Avoid race condition
    - Can’t take advantage of a multiprocessor since it run exclusively
    - The programmer need to insert yield() calls

Architectures for multi-threaded servers … continued

- **Thread-per-connection**
  - Server creates a new worker thread when client creates a connection, destroys the thread when the client closes the connection
  - Pro: lower thread management overheads compared with the thread-per-request
  - Con: client may be delayed while a worker thread has several outstanding requests but another thread has no work to perform

- **Thread-per-object**
  - Associate a thread with each remote object
  - Pro & Con are similar to thread-per-connection
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)
- Threads within same process can share data and other resources more conveniently and efficiently (without copying or messages)
- Threads within a process are not protected from each other

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 BLOCKED)</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>
多线程提高效率的机理

- 在只有一个处理器的情况，如果两个或多个线程并发执行，都没有阻塞，微观上是多个线程轮流使用CPU，那么线程的并发和串行执行的效果一样。
- 线程并发执行只有在阻塞的情况下才能起到并发执行的效果。
- 上述情况在IO处理时比较常见。
- 如果有多个处理器，则可以提高效率。
- 如果操作系统提供足够的支持，即使一个处理器，能够在硬件流水的模式下，让线程并发地在不同的硬件资源上执行，那就能够真正提高效率。

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 preemptive
    - can exploit multiple processors
    - page fault blocks all threads
  - hybrid approaches can gain some advantages of both
    - user-level hints to kernel scheduler
    - hierarchic threads (Solaris 2)
    - event-based (SPIN, FastThreads)
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.

Java Thread constructor and management methods

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.
创建线程的方式

1. public class mythread extends Applet
   implements Runnable
   (小应用或已经是某个类的子类时)
2. 继承类Thread
   public class mythread extends Thread
3. 上述两种方法中都可用类Thread产生线程的对象Thread newthread;
4. 创建并启动线程
   newthread=new Thread(this);
   newthread.start();

创建线程的方式

5. run方法是运行线程的主体,启动线程时,由java直接调用public void run()
6. 停止线程,由小应用程序的stop调用线程的stop
   newthread.stop()
7. sleep方法的作用,暂停线程的执行,让其它线程得到机会,sleep要丢出异常,必须抓住.
   Try{
       sleep(100);
   }catch(InterruptedException e){。。。。}
### 线程的状态

```
new Thread()

Runnable

Not Runnable

Dead
```

- `new Thread()`
- `start()`
- `yield()`
- `sleep()`
- `wait()`
- `suspend()`
- `run()`
- `exit`
- `stop()`
- `resume()`

### 多线程问题---执行的顺序

- 多个线程运行时，调度策略为固定优先级调度，级别相同时，由操作系统按时间片来分配。
- 下面给出的例子中，共运行三个线程，它们做同样的事，每次打印循环次数和自己的序列号，运行结果表明，它们并不是连续运行的。
- 在上例中如果给某个线程赋予较高的优先权，则发现这个进程垄断控制权。
  ```java
  thread.setPriority(Thread.MAX_PRIORITY)
  ```
多线程问题---如何写多线程

1. 分别定义不同的线程类，在各自的run方法中定义线程的工作
   class mythread1 extends Thread
   { public void run{....}  }
   class mythread2 extends Thread
   { public void run{....}  }

2. 在主类中实例化各线程类并启动线程。
   public class demo extends Applet
   { public void init()
     { mythread t1=new mythread1();
       mythread t2=new mythread2();
       t1.start(); t2.start();}  }
多线程问题---线程间的通信

1. 线程间的通信可以用管道流。

创建管道流:
```
PipedInputStream pis=new PipedInputStream();
PipedOutputStream pos=new PipedOutputStream(pis);
```
或:
```
PipedOutputStream pos=new PipedOutputStream();
PipedInputStream pis=new PipedInputStream(pos);
```

2. 通过一个中间类来传递信息。
```
PrintStream p = new PrintStream( pos );
p.println(“hello”);
DataInputStream d=new DataInputStream(pis);
d.readLine();
```
多线程问题---线程间的通信

- 管道流可以连接两个线程间的通信
- 下面的例子中有一个两个线程在运行，一个往外输出信息，一个读入信息。
- 将一个写线程的输出通过管道定义为读线程的输入。

```java
outStream = new PipedOutputStream();
inStream = new PipedInputStream(outStream);
new Writer(outStream).start();
new Reader(inStream).start();
```

多线程问题---资源协调

1. 数据的完整性

线程1取过来
加1后送回去

资源

变量

withdrawal()透支

余额
对共享对象的访问必须同步，叫做条件变量。
Java语言允许通过监视器(有的参考书称其为管程)使用条件变量实现线程同步。
监视器阻止两个线程同时访问同一个条件变量，它的如同锁一样作用在数据上。
线程1进入withdrawal方法时，获得监视器（加锁）；当线程1的方法执行完毕返回时，释放监视器（开锁）,线程2的withdrawal方能进入。

用synchronized来标识的区域或方法即为监视器监视的部分。
一个类或一个对象有一个监视器，如果一个程序内有两个方法使用synchronized标志，则他们在一个监视器管理之下。

一般情况下，只在方法的层次上使用关键区。
此处给出的例子演示两个线程在同步限制下工作的情况。

```java
class Account {
    static int balance = 1000;  // 为什么用static?
    static int expense = 0;
    public synchronized void withdraw(int amount) {
        if (amount <= balance) {
            balance -= amount;
            expense += amount;
        } else {
            System.out.println("bounced: "+amount);
        }
    }
}
```

可能出现的问题:
- 生产者比消费者快时，消费者会漏掉一些数据没有取到。
- 消费者比生产者快时，消费者取相同的数据。

`notify()` 和 `wait()` 方法用来协调读取的关系。
- `notify()` 和 `wait()` 都只能从同步方法中的调用。
多线程问题---资源协调

- **notify** 的作用是唤醒正在等待同一个监视器的线程。
- **wait** 的作用是让当前线程等待
- 信息版例子
- **read()** 方法在读信息之前先等待，直到信息可读，读完后通知要写的线程。
- **write()** 方法在写信息之前先等待，直到信息被取走，写完后通知要读的进程。

```java
class WaitNotifyDemo {
    public static void main(String[] args) {
    MessageBoard m = new MessageBoard();
    Reader readfrom_m = new Reader(m);
    Reader writeto_m = new Writer(m);
    readfrom_m.start();
    writeto_m.start();
    }
}
```
class MessageBoard {
{   private String message;
    private boolean ready = false; //信号灯
public synchronized String read()
{  while (ready == false)
{  try { wait(); } catch (InterruptedException e) { } }
    ready = false;
    notify(); //起始状态先写后读
        return message;
}
public synchronized void write(String s)
{   while (ready == true)
{   try { wait(); } catch (InterruptedException e) { } }
    message = s; ready = true; notify();
}}

class Reader extends Thread
{   private MessageBoard mBoard;
public Reader(MessageBoard m)
{   mBoard = m;   }
public void run()
{   String s = " ";
            boolean reading = true;
            while( reading ){
                s = mBoard.read();
        System.out.println("Reader read: " + s);
        if( s.equals("logoff") ) reading = false;   }
        System.out.println("Finished: 10 seconds...");
        try{ sleep( 10000 ); } catch (InterruptedException e) { }
        catch (InterruptedException e) {   }   } }
class Writer extends Thread
{
    private MessageBoard mBoard;
    private String messages[] = {
        "Monday :------------------------",
        "……",
        "Sunday : ----------------------"};

    public Writer(MessageBoard m)
    {
        mBoard = m;
    }

    public void run()
    {
        for (int i = 0; i < messages.length; i++)
        {
            mBoard.write(messages[i]);
            System.out.println("Writer wrote:" + messages[i]);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) {
            }
        }
        mBoard.write("logoff");
    }
}

Chapter 5: Operating System Support

- Introduction
- The operating system layer
- Protection
- Processes and Threads
- Communication and invocation
- Operating system architecture
- Summary
Asking following questions about OS:

- 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 disconnection operation?

Communication primitives & protocols

- Communication primitives
  - TCP(UDP) Socket in Unix and Windows
  - DoOperation, getRequest, sendReply in Amoeba
  - Group communication primitives in V system and Chorus
Communication primitives & protocols

- **Protocols and openness**
  - provide standard protocols that enable internetworking between middleware
  - integrate novel low-level protocols without upgrading their application
  - Static stack
    - new layer to be integrated permanently as a “driver”
  - Dynamic stack
    - protocol stack be composed on the fly
    - E.g. web browser utilize wide-area wireless link on the road and faster Ethernet connection in the office

Invocation performance

- **Invocation costs**
  - Different invitations
  - The factors that matter
    - synchronous/asynchronous, *domain transition*, communication across a network, thread scheduling and switching
Invocation performance

- Invocation over the network
  - Delay: the total RPC call time experienced by a client
  - Latency: the fixed overhead of an RPC, measured by null RPC
  - Throughput: the rate of data transfer between computers in a single RPC
  - An example
    - Threshold: one extra packet to be sent, might be an extra acknowledge packet is needed

Support for communication and invocation

- The performance of RPC and RMI mechanisms is critical for effective distributed systems.
  - Typical times for 'null procedure call':
    - Local procedure call  < 1 microseconds
    - Remote procedure call ~ 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.
Support for communication and invocation

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

Improve the performance of RPC

- Memory sharing
  - rapid communication between processes in the same computer
- Choice of protocol
  - TCP/UDP
    - E.g. Persistent connections: several invocations during one
  - OS’s buffer collect several small messages and send them together
- Invocation within a computer
  - Most cross-address-space invocation take place within a computer
  - LRPC (lightweight RPC)
Asynchronous operation

o **Performance characteristics of the Internet**
  - High latencies, low bandwidths and high server loads
  - Network disconnection and reconnection.
  - outweigh any benefits that the OS can provide

o **Asynchronous operation**
  - Concurrent invocations
    - E.g., the browser fetches multiple images in a home page by concurrent *GET* requests
  - Asynchronous invocation: non-blocking call
    - E.g., CORBA *oneway* invocation: maybe semantics, or collect result by a separate call

Asynchronous operation … *continued*

o **Persistent asynchronous invocations**
  - Designed for *disconnected operation*
  - Try indefinitely to perform the invocation, until it is known to have succeeded or failed, or until the application cancels the invocation
  - QRPC (Queued RPC)
    - Client queues outgoing invocation requests in a stable log
    - Server queues invocation results

o **The issues to programmers**
  - How user can continue while the results of invocations are still not known?
Chapter 5: Operating System Support

- **Introduction**
- **The operating system layer**
- **Protection**
- **Processes and Threads**
- **Communication and invocation**
- **Operating system architecture**
- **Summary**

Monolithic kernels and microkernels

- **Monolithic kernel**
  - Kernel is massive: perform all basic operating system functions, megabytes of code and data
  - Kernel is undifferentiated: coded in a non-modular way
  - E.g. Unix
  - Pros: efficiency
  - Cons: lack of structure
Monolithic kernels and microkernels

- **Microkernel**
  - Kernel provides only the most basic abstractions: address spaces, threads and local interprocess communication
  - All other system services are provided by servers that are dynamically loaded
  - E.g., VM of IBM 370
  - Pros: extensibility, modularity, free of bugs
  - Cons: relatively inefficiency

- **Hybrid approaches**

---

Chapter 5: Operating System Support

- **Introduction**
- **The operating system layer**
- **Protection**
- **Processes and Threads**
- **Communication and invocation**
- **Operating system architecture**
- **Summary**
Summary

- **Process & thread**
  - A Process consists of multiple threads and an execution environment
  - Multiple-threads: cheaper concurrency, take advantage of multiprocessors for parallelism
- **Remote invocation cost**
  - Marshalling & unmarshalling
  - Data copying
  - Packet initialization
  - Thread scheduling and context switching
  - Network transmission
- **OS architecture**
  - Monolithic kernel & microkernel

System layers

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

Figure 2.1 software and hardware service layers in distributed systems

Core OS functionality
Process address space

- **Regions can be shared**
  - kernel code
  - libraries
  - shared data & communication
  - copy-on-write

- **Files can be mapped**
  - Mach, some versions of UNIX

- **UNIX fork() is expensive**
  - must copy process's address space

---

Copy-on-write scheme

- **Process A's address space**
  - RA
  - RB copied from RA

- **Process B's address space**
  - RB

---

A's page table

- Shared frame

B's page table

- Kernel

a) Before write

b) After write
Invocations between address spaces

(a) System call

(b) RPC/RMI (within one computer)

(c) RPC/RMI (between computers)

Delay of an RPC operation when returned data size varies
A lightweight remote procedure call

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

User

Kernel

Client

Server

A stack

Times for serialized and concurrent invocations

<table>
<thead>
<tr>
<th>Serialised invocations</th>
<th>Concurrent invocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>process args</td>
<td>marshal</td>
</tr>
<tr>
<td>receive</td>
<td>unmarshal</td>
</tr>
<tr>
<td>transmission</td>
<td>receive</td>
</tr>
<tr>
<td>receive</td>
<td>unmarshal</td>
</tr>
<tr>
<td>receive</td>
<td>unmarshal</td>
</tr>
<tr>
<td>receive</td>
<td>unmarshal</td>
</tr>
<tr>
<td>client</td>
<td>server</td>
</tr>
</tbody>
</table>

Receive
unmarshal
process results

Receive
unmarshal
process results

Receive
unmarshal
process results

Receive
unmarshal
execute request

Send

Send

Send

Send

time
Monolithic kernel and Microkernel

<table>
<thead>
<tr>
<th>Server</th>
<th>Kernel code and data:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamically loaded server program:</td>
<td></td>
</tr>
</tbody>
</table>

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

| Microkernel |

<table>
<thead>
<tr>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>The microkernel supports middleware via subsystems</td>
</tr>
</tbody>
</table>

Threads versus multiple processes

- Main state components of Execution Environment and Thread
- The comparison of processes and threads
  - Creating a new thread within an existing process is cheaper than creating a process
    - 1ms vs. 11ms
  - Switching to a different thread within the same process is cheaper than switching between threads belonging to different processes
    - 0.4ms vs. 1.8ms
  - Threads within a process may share data and other resources conveniently and efficiently compared with separate process
  - But, threads within a process are not protected from one another
Threads programming

- Concurrent programming
  - Race condition, critical section, monitor, condition variable, semaphore
  - C Threads package or pthreads for C, Java
- The Java thread class

Threads within clients

- The client example
  - First thread: generates results to be passed to a server by remote method invocation, but does not need a reply
  - Second thread: perform the remote method and block while the first thread is able to continue computing further results
- Web browser
  - Multiple threads handle multiple concurrent requests for web pages
Thread synchronization

- Variable
  - each thread’s local variables in methods are private to it
  - no private copies of static variables or object instance variables

- Java synchronized methods

- Example: multiple threads manipulate a queue
  - Mutual exclusive
    - Synchronized method of addto() and removefrom() methods in the queue class
  - Producer-consumer
    - Wait(): block on waiting condition variables
    - notify(): unblock the waiting threads