# 条件临界区域（conditional critical region）

条件临界区域（CCR, Brinch Hansen ）几乎与管程同时提出，特征：

- 一种有特定语法结构的临界区，封装一个**受保护**变量
- 线程只能在针对特定受保护变量的 region 语句里访问该变量
- 访问同一受保护变量的 region 语句属于同一 CCR，其执行互斥
- 每个 CCR 有一个布尔条件，任何到达相应 region 语句的线程都必须等到该条件为真且当时无其他线程在针对同一保护变量的 region 语句里
- CCR 可以嵌套，这时也要关心死锁问题（与管程的情况类似）

CCR 的概念

- 出现在并发语言 Edison 里
- 对 Ada 95 和 Java/C# 等语言的同步机制设计有很大影响

  这几种语言的并行机制实际上是管程和 CCR 的组合，不同语言采用的方式有一些不同



# 条件临界区域

基于条件临界区域（CCR）的有界缓冲区实现。region 语句可以有布尔条件，因此可以避免显式的条件变量

CCR 避免了 signal 的语义问题：这里用布尔条件而不是条件变量，且只能在 CCR 开始等条件

采用 CCR 的描述相对最简单清晰。但这种结构实际上明显低效：

常见实现中退出 CCR 的代码都立即唤醒所有等待线程，让它们在各自的环境里检查自己的条件

最坏情况：每当有线程退出 CCR，就需要对所有等待线程做上下文的切入切出，开销很大

```
buffer : record
    buf : array [1..SIZE] of data
    next_full, next_empty : integer := 1, 1
    full_slots : integer := 0

procedure insert(d : bdata)
    region buffer when full_slots < SIZE
        buf[next_empty] := d
        next_empty := next_empty mod SIZE + 1
        full_slots −:= 1

function remove : bdata
    region buffer when full_slots > 0
        d : bdata := buf[next_full]
        next_full := next_full mod SIZE + 1
        full_slots +:= 1
    return d
```

# Ada 95 的同步

**Ada 83** 的主要同步机制基于消息传递，**Ada 95** 增加了**受保护对象**

- 受保护对象可以有三类子程序成员：函数、过程和**入口**
  - 函数只能读对象数据，过程和入口可读可写。每个受保护对象有一个读者写者锁，保证所有可能冲突的操作互斥
  - 过程或入口用于获得对受保护对象的排他性访问，多个函数可以并发执行，但不能和过程或入口同时执行
  - 入口可带布尔**卫**，调用线程需要等卫成立（像 **CCR** 的布尔表达式）
- 三类特殊调用：**限时**调用，等待特定时长后作废；**条件**调用，不能立即执行就转去执行一段替代代码；**异步**调用，立即执行一段替代代码，如果替代代码结束前该调用可执行，就作废替代代码改执行调用
- 入口卫不在调用线程的上下文中求值，因此比 **CCR** 条件高效。所有卫都在受保护对象的定义里，可在目标代码中成组检测

  不能在入口执行中等待条件，但可调用另一入口而**入队**，效果相同



# Java 的同步

多线程共享的对象有一个互斥锁，通过 **synchronized** 语句获取（同步）

```
synchronized (my_shared_obj) {
    ...     // critical section
}
```

引用同一对象的所有同步语句在执行上互斥，引用不同对象的同步语句可同时执行。成员函数可为 **synchronized**，相当于方法体用 **synchronized(this)** 语句包围，对这种方法的调用将互斥

调用共享对象的非同步方法，或对其 **public** 数据成员的访问都不互斥，它们可以并发执行，也可以和 **synchronized** 语句或方法并发进行

线程通过调用预定义无参方法 **wait** 将自己挂起。虚拟机不能判断线程在给定对象上挂起的原因，这使线程可能由于并不合适的原因而被唤醒，因此通常必须把 **wait** 嵌在一个条件检测循环里

```
while (!condition) {
    wait();
}
```

对对象执行 **wait** 方法将释放该对象的锁。在嵌套的同步语句或同步方法里执行 **wait** 时，不释放线程掌握的任何其他对象的锁

# Java 的同步

- 要唤醒在一个对象上等待的线程，就必须在引用该对象的同步语句或方法里执行 **notify** 方法（无参。显然要由另一线程执行）
  - 虚拟机响应 **notify** 时取出在该对象上挂起的某线程并将它变成可运行。如果当时不存在挂起进程，**notify** 相当于空操作
  - 调用 **notifyAll** 方法唤醒在一个对象上等待的所有线程
- 如果一个线程等待的条件多于一个（其 **wait** 嵌套在几个不同循环里），这里无法保证被唤醒的确实是应该唤醒的线程
  - 为保证确实能唤醒一个线程，人们通常用 **notifyAll** 而不用 **notify**
  - 为保证只有**一个被**唤醒线程进入执行，最先看到自己等待的条件已满足的线程必须去修改对象状态，以保证其他被唤醒线程在得到运行机会的时候立刻转回去休眠
  - 所有等待线程每次得到运行机会都将重新求值自己的条件。如果问题中需要等待多个条件，这种“解决方法”的代价可能很高



# Java 的同步

在 **Java 5** 版本之前，要实现高效程序，通常需要设法设计一种算法，保证其中的线程在任何时刻都不同时等待多个条件

**2004** 年发布的 **java.util.concurrent** 包提供一种更通用的解决方案。遇到上述情况时可以定义一个 **Lock** 变量，代替 **synchronized** 语句和方法

老代码

```
synchronized (my_shared_obj) {
    ...     // critical section
}
```

现在可以写成

```
Lock l = new ReentrantLock();
l.lock();
try {
    ...     // critical section
} finally {
    l.unlock();
}
```

**Lock** 变量在作用域结束不自动释放锁，需要写语句明确释放

显式释放有可能写错

但是这种机制也使人可以实现一些有用的算法，其中锁获取和释放不按后进先出的顺序进行

注意：用 **synchronized** 语句或方法时，锁的获取和释放是隐式的，而且总是后获得者先释放

# Java 的同步

Java 还有 tryLock 方法，其行为与 Ada 95 的限时入口调用类似，只在立即可用或可选描述的时间内可用时得到锁（用一个布尔值报告成功与否）

一个 Lock 变量可以有任意多个关联的 Condition 变量，利用这种机制很容易描述线程需要等待多个条件的算法，其中不需要用 notifyAll

```
Condition c1 = l.newCondition();
Condition c2 = l.newCondition();
...
c1.await();
...
c2.signal();
```

只用 synchronized 方法（不用锁或同步语句）的 Java 对象的行为像一个管程，但这个“管程”里只允许有一个条件变量

如果 synchronized 语句是以 wait 开始的循环，其功能与每次显式检查条件的 CCR 类似。由于需要显式写 notify，因此不必在退出临界区时都重新求值条件，只需要在 notify 时做



# Java 的同步

Java 和其他一些语言的设计说明，管程和 CCR 的差异实际上很模糊

一些相关情况：

- 存在一种统一方法解决所有同步问题：对每个保护对象用一个布尔条件，让线程在条件上忙等待
  - 这种方案不完美，因为需要在程序里到处使用低级的信号量，基于同步语句或者方法的隐式互斥描述更清晰
- 有些问题可以很自然地用多重条件表述，如果需要用 Java 的基本同步机制描述，就必须在优美和高效之间做某种选择，很难兼得
- Java 5 增强的并发机制是企图缓解这种两难境况的一个尝试
  - 这里的 Lock 变量既保留了管程和 CCR 互斥和条件同步之间的差异，又使人能把等待线程划分为一些等价类，可以按类分别唤醒
  - 类划分的粒度可以适当选择和改变，因此可以权衡设计，在 CCR 的简单性到 Hoare 风格管程的效率之间任意选择适当的位置

# 消息传递模型

小型多处理器系统里常见共享存储器的并发程序设计

在大型多机系统和网络上，目前的大多数并发程序设计都基于消息模型

下面讨论基于消息的计算中最主要的三个问题：

- 通讯方命名
- 发送
- 接收

最后仔细讨论发送和接收语义的一种特殊组合，远程过程调用

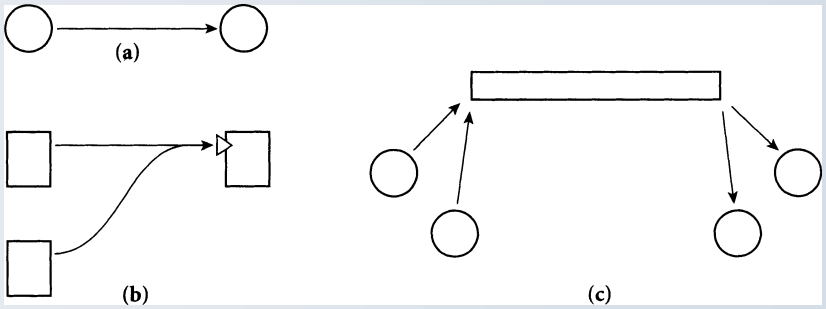
这里的大部分例子来自 **Ada**、**Occam** 和 **SR** 程序设计语言，**Java** 的网络库和**PVM/MPI** 库程序包



# 通讯对方命名（Naming Communication Partner）

发送或接收消息时，通常必须描述把它送到哪里去，或从哪里收到它

参与方需要相互指名（引用）通讯的对方。指名方式有几种：直接引用线程或进程；引用特定模块的**入口**或**端口**；引用某种**套接字**或**通道**抽象




**(a)** 两进程明确地互相指名对方（按进程/线程给消息定址）

**(b)** 发送方指名接收方的一个输入端口，端口也常称为入口或操作，接收方一般为内部有一个或几个线程的模块（按端口/入口/操作）

**(c)** 发送/接收方指名某独立的通道抽象，也称为连接或信箱（按通道）

# 通讯方命名：实例

- 按进程给消息定址（Hoare 最早的 CSP 建议，PVM 和 MPI）
  - 每个 PVM 和 MPI 进程有唯一 id（一个整数），send/receive 里写明通信对方的id。MPI 要求其实现可重入，一个进程可以安全地划分为多个线程，每个线程可以以进程的名义发送/接收消息
- 按照端口给消息定址（Ada）：
  - Ada的**入口调用** t.foo(args) 把消息送到作业 t 的 foo 入口（t 可为作业名或指向作业的指针）。作业与模块类似，入口就像内部子程序的头部，作业里通过 accept 语句接收送给入口的消息。每个入口属于唯一一个作业，所有送给同一入口的消息都由这个作业接收
- 按通道给消息定址（Occam）：通道由内部 CHAN 和 CALL 类型支持

```
CHAN OF BYTE stream :
CALL lookup(RESULT [36]BYTE name, VAL INT ssn) :
```

stream 定义为单向通道，传输 BYTE 消息；lookup 是双向通道，送 INT 消息，返回 36 个 BYTE 的字符串回复



# 通讯方命名：Java

Java 的互联网库采用了多种指名方式

- java.net 库提供两种消息传递风格，分别对应互联网 UDP 和 TCP 协议
- UDP 是一种**数据报**协议，比较简单。消息采用端口指名方式，消息送到某一特定的**互联网地址**和**端口号**
- TCP协议也采用基于端口指名，但只用它建立**连接**，建立好的连接用于随后的通信。连接能保证可靠地按顺序传递消息

从通讯方命名的角度看，TCP 协议采用了两种不同的方式：

- 先通过指名端口的方式建立连接（建立通道）
- 而后通过通道进行通讯

Java 建立的一个 TCP 连接在每一端有一个 OS 进程，内部可能有许多线程，每个线程都可以使用其进程的连接端

# 通讯方命名：Java

服务器端一直在端口上“监听”，accept 操作阻塞到接收客户端连接请求，收到请求时它立即分支（fork）出一个新线程与客户端通信：

```
ServerSocket my_server_socket = new ServerSocket(port_id);
Socket client_connection = my_server_socket.accept();
```

客户端发连接方式是将服务器端符号名和端口号传给 Socket的构造函数

```
Socket server_connection = new Socket(host_name, port_id);
```

建立连接后客户端和服务器端将调用 Socket 类的方法创建输入和输出流，它们支持所有标准的 Java 正文 I/O 功能

```
BufferedReader in = new BufferedReader(
    new InputStreamReader(client_connection.getInputStream()));
PrintStream out =
    new PrintStream(client_connection.getOutputStream());
// This is in the server; the client would make streams out
// of server_connection.
...
String s = in.readLine();
out.println("Hi, Mom\n");
```



# 发送（sending）

send 操作的一个最重要设计问题是调用者阻塞的时间，即线程做了 send 之后什么时刻继续执行？常见阻塞方式有三种：

- **资源管理**：基础系统没把外送数据复制到安全位置之前，发送线程不应修改该数据。大多数系统中发送方阻塞到某个时刻，此后线程可以安全地修改数据而不会危及外送消息（阻塞到消息已另行保存）
- **失败语义**：通过远距离网络通信时消息传递出错的可能性很大。许多系统阻塞发送方直到得知消息正确送达（阻塞到消息已安全送达）
- **返回参数**：许多情况下消息里包含一个请求，期望得到回复。许多系统阻塞消息发送方直到收到回复（阻塞到得到了回复）

在决定应阻塞多长时间时，必须考虑许多问题，如

- 同步语义
- 缓冲区需求
- 运行时的错误报告

# 发送：同步语义

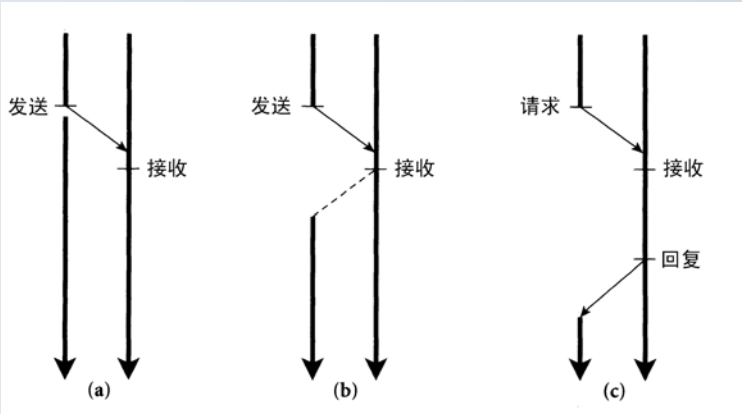
消息传送可能经过许多中间步骤。先在发送方软件中下降若干层次，而后可能经过很多中间机器，最后在接收方上升若干层次

可在任何一步解除发送方阻塞，但多数情况在用户层面无法区分。如果消息系统的缓冲区足够大，三种延迟就对应着发送的三种主要语义：

**(a)** 无等待发送：发送方阻塞很短时间，消息传递机制把消息复制到安全位置后负责送达

**(b)** 同步发送：发送方一直等到它的消息被接收

**(c)** 远程调用发送：发送方一直等到收到回复




无等待 **send**：**SR** 和 **Java** 的互联网包；同步 **send**：**Occam**；远程调用 **send**：**SR**、**Occam** 和 **Ada**。**PVM**和**MPI**：无等待 **send** 和同步 **send** 的混合



# 发送：缓冲区

消息系统不可能提供完全无等待的 **send** 版本（除非直接丢掉消息）

设想：不断把消息发给一个根本不接收的线程，需要无限的缓冲区存储。线程实现必须能阻塞过分活跃的发送方，防止它破坏整个系统

对任意固定大小的缓冲区，都能存在需要更大空间才能正确运行的程序

设某消息系统可以在任何通信路径上缓冲 ***n*** 个消息。考虑下面程序

***A*** 要发 ***n*** + 1 个消息给 ***B***，而后发一个消息给 ***C***。***C*** 随之由另一路径发一个消息给 ***B***。***B*** 要求在接收 ***A*** 的消息之前先接收 ***C*** 的消息。***A*** 发了***n*** 个消息之后就会阻塞和死锁。这是“与实现有关的死锁”

最好的实现方式是提供足够大的空间，使应用不会发现“规模的限制”

对同步 **send** 和远程调用 **send**，缓冲区空间通常不是问题：

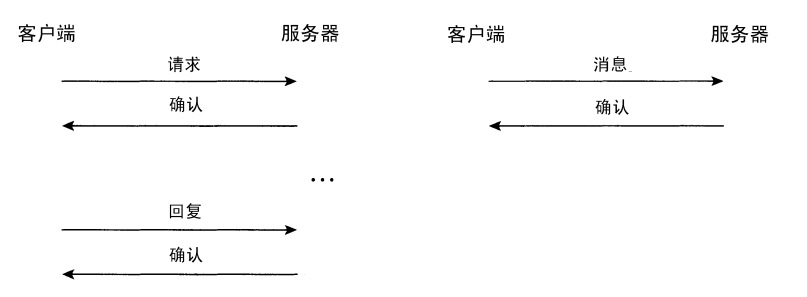
消息所需空间受到线程数的限制，程序能创建的线程数已有限制。回复线程总应能继续，请求线程正等着这个回复，并能重用相关缓冲区空间

三种 **send** 可以互相模拟（功能等价，简单，从略）

# 发送：错误报告

消息传递系统不一定可靠，语言或库常通过**确认**消息验证一次传输成功

在合理时间内没收到确认，系统就重发消息。尝试几次都没收到客户端确认，就生成一个错误报告




如果发送方仍在阻塞状态，发消息过程中的错误就可作为异常，或通过结果参数/全局变量报告。如果发送方已经继续执行，就很难报告错误

- **UDP** 认为消息都不可靠，出错就直接丢掉；**TCP** 保证只有出现“灾难性”错误时才会丢消息（如：连接已不可用，多次调用都立即失败）
- **MPI** 的方式更严厉：出现消息无法送达时认为整个程序失败
- **PVM** 提供了**通知**机制，出现失败时给一个事先指定的进程发消息，要求它处理（可执行清理工作：如废除相关进程，启动新进程接替工作）



# 发送：库或语言

上面实例都假定用库实现。其中 **send**、**receive**、**accept** 等都是常规子程序，参数个数固定，用两个参数描述发送地址和消息长度

如果要发送的消息内容保存在几个程序变量里，就必须自己把它们**汇集**或**整编**到一个记录里之后发送

接收端也必须把有关值**分解**（**解编**）后送回，以便存入指定的程序变量

并发语言可能以更灵活的方式提供消息传递操作

- “参数”表通常可以包含任意多个需要发送的值
- 编译器可以对值做类型检查，可采用一些类似于跨编译单元的子程序连接时所用的技术
- 并发语言可以采用非子程序调用的语法形式，如把远程调用的**accept**和**reply**结合起来，使**reply**不必明确说明对应哪个**accept** 等等

采用语言支持并发机制有不少优点

但改进慢（注意，并发领域很不成熟，变化比较大）

# 接收

消息接收机制的最主要情况是显式 **receive** 操作和前面说的**隐式**接收

- 采用显式 **receive** 时，外来的消息进入等待队列，直到某个现有线程有意接收它。任意时间点可能存在数量很大的一批待接收消息
- 采用隐式接收，到达特定端口的每个消息都创建一个新线程（受到资源限制，线程数增加过多时会暂停处理请求）

多数语言和库的线程都有某种**选择性**，可以表明希望接收的消息类型

- **PVM** 和 **MPI** 消息都包含发送进程 **id** 和它确定的一个整数**标志**

  **receive** 操作描述期望的发送方 **id** 和消息标志，只接收匹配的消息

- **Ada**、**Occam** 和 **SR** 是语言，采用特殊的非过程调用语法形式描述接收

  消息集成在命名系统和类型系统里，线程都能基于端口/通道名和参数做选择（而不是只能基于简单的标志）

  三种语言里的选择 **receive** 结构都是某种特殊形式的**卫式命令**



# 接收

**Ada 83** 的有界缓冲区代码。这里定义了一个主动的“管理器”线程，在循环里执行 **select** 语句

**accept** 语句接收远程调用请求的 **in** 和 **in out** 参数，在 **end** 处返回 **in out** 和 **out**参数作为回复消息

分支分两步：先求值各**when**表达式（卫），而后考虑真分支后面的 **accept** 语句，看消息是否可用

**accept** 的卫可选，没有时相当于 **when true =>**

如果多个卫都为真（如例中的缓冲区有部分填充）且消息都可用，系统可以选择执行任一分支，具体选择由实现决定

```
task buffer is
    entry insert(d : in bdata);
    entry remove(d : out bdata);
end buffer;

task body buffer is
    SIZE : constant integer := 10;
    subtype index is integer range 1..SIZE;
    buf : array (index) of bdata;
    next_empty, next_full : index := 1;
    full_slots : integer range 0..SIZE := 0;
begin
    loop
        select
          when full_slots < SIZE =>
            accept insert(d : in bdata) do
                buf(next_empty) := d;
            end;
            next_empty := next_empty mod SIZE + 1;
            full_slots := full_slots + 1;
        or
          when full_slots > 0 =>
            accept remove(d : out bdata) do
                d := buf(next_full);
            end;
            next_full := next_full mod SIZE + 1;
            full_slots := full_slots - 1;
        end select;
    end loop;
end buffer;
```

# 接收

客户端作业可通过**入口调用**与这个有界缓冲区通信

```
-- producer:                    -- consumer:
buffer.insert(3);               buffer.remove(x);
```

每个 **select** 语句至少有一个 **accept** 分支。还有另外三种分支：

```
when condition => delay how_long
    other_statements
...
or when condition => terminate
...
else ...
```

如果在 ***how_long*** 秒内无其他分支可选，就可以选 **delay** 分支（**Ada** 要求其实现支持最长至一天，最短至**20 ms**的延迟时间）

如果所有潜在通信对方都已终止或都停在带 **terminate** 分支的 **select** 里，就可以选 **terminate** 分支。这种分支导致执行这个 **select** 的作业终止

在其他分支的卫都不为真或没有立即可执行的 **accept** 时选中 **else**分支。在有 **else** 分支的 **select** 语句里不允许有 **delay** 分支（语义矛盾）



# 远程过程调用

三种 **send**（无等待/同步/远程调用）可与两种 **receive**（显式/隐式）分别配对

- 远程调用 **send** 与显式接收的组合有时称为**握手**（**Ada**）
- 远程调用 **send** 和隐式接收的组合称为**远程过程调用**（**RPC**）

存在几种支持 **RPC** 的并发语言（如 **SR**），许多顺序语言扩充带有**桩编译器**支持 **RPC**。桩编译器以被远程调用的子程序的形式化描述（与子程序头部参数声明类似）作为输入，基于输入生成**客户端桩**和**服务器端桩**

**RPC** 执行情况。客户端程序调用一个远程子程序的情况：

- 子程序的客户端桩把参数和期望操作的指示符整编为一个消息发给服务器，而后等待回复消息；在接到回复后将回复消息解编并赋给各结果参数
- 服务器端桩以消息缓冲区为参数，取得消息后分解出请求的各参数，调用适当局部过程，得到过程返回结果后将其整编为消息并发送给客户
- 客户端桩调用相对简单。服务器端桩的调用将在下面讨论实现时考察

# 远程过程调用：语义

**RPC** 的一个设计目标是尽可能使调用的远程情况**透明化**，使之看起来就像是一个做了本地调用

桩编译器系统生成的客户端桩接口应与其代理的远程过程接口一模一样，应能在不知道/不关心它究竟是本地还是远程的情况下调用它

然而，得到透明性有几个重要的实际困难：

- 参数模式受限：很难实现跨网络的引用调用参数，因为实际参数不在被调子程序的地址空间里。访问全局变量也存在类似的困难
- 性能约束：远程过程调用可能需要更长时间才能返回，这无法避免。使用时常常不得不考虑网络延迟的影响
- 失败语义：远程调用失败的可能性更大。本地调用可假设被调过程或正好运行一次，或者失败。这种假设未必能用到远程调用



# 远程过程调用：语义

一些可能的办法：

- 如果程序不依赖引用产生的别名，可用值/结果参数代替引用参数。**Ada** 说能区分 **in out** 参数的引用和值/结果实现的程序是**错误程序**
- 引用参数和全局变量也能实现：通过消息传一个大信息块。这种做法代价太高，而且给远程过程传链接数据结构时无法做深拷贝

本地与远程调用的性能差异只能通过减慢本地调用来掩盖，显然不合适

调用失败时令调用方夭折，可以提供一次调用的失败语义

在要求高可靠的系统里可以推迟调用方的执行，直到操作系统或运行时系统能用保存在磁盘的信息重新构造失败的计算

可以考虑用发失败通知的方式接受“至多失败一次”语义

如设法恢复丢失的消息时，根据需要重传远程调用。需要保证不出现调用的多次实际执行，但可以接受由于通信故障而没有实际调用。如果语言有异常机制，实现就可以利用它把通信故障做得像是运行时错误

# 远程过程调用：参数

Ada 的参数模型相对高级，同一组参数模式可以同时用于子程序和入口

- 编译器将尽可能通过引用方式把实参送给子程序，避免复制
- 如果所用作业位于多处理器系统或集群中的其他处理器，编译器将自动用值-结果的方式把同样的实参传给入口

有几个并发语言为支持远程调用而设计了特殊的参数模式

- **Emerald** 里每个参数都引用一个对象，对远程对象的引用通过消息传递的方式透明实现。为尽可能减少这种引用，远程调用中传递的对象通常随调用迁移，打包放入消息送到远程站点（这使远程程序可以局部访问它们），回复时送回调用方。**Emerald** 把这种方式称为移动调用
- **Hermes** 的参数传递是破坏性的：从调用方角度看，调用使实参变成未初始化状态，因此可以迁移到被调用方，不会出现远程引用

还可以有其他的考虑



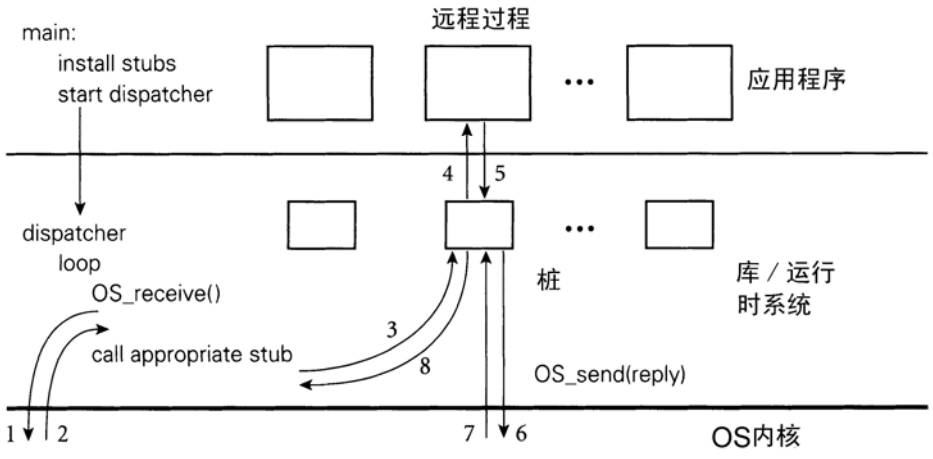
# 远程过程调用：实现

在（服务器端）内核接口层的 **receive** 通常是显式的。为使应用程序员将其当作隐式，桩编译器的代码（**SR** 等的运行系统）要填补其中空白

典型的层次结构：

上面是应用程序

中间包括库和 **RPC** 桩编译器产生的代码




初始化时应用程序两次调用进入运行时系统：

- 第一次是注册桩编译器产生的桩子程序代码指针
- 第二次调用启动**消息分派器**

# 远程过程调用：实现

最简单情况：一个 **OS** 进程上的单线程服务器

- 指派器一直在循环中，频繁地进入内核去接收消息
- 一旦收到消息，指派器就调用适当 **RPC** 桩子程序，分解出请求的参数并调用适当的应用层过程
- 过程返回时，桩子程序把返回参数整编为一个回复消息，进入内核把消息送回调用方后返回指派器

如远程请求都能及时处理，这种方式就很合适，甚至可能不出现阻塞

如果远程请求需要经常等用户层同步，服务器进程就需要一个线程就绪表，而且把指派器集成到普通线程调度器里

- 当前线程（在应用代码）阻塞时，调度器/指派器由就绪表取一个线程
- 就绪表为空时调度器/指派器进入内核去收一条消息，并分一个新线程去处理它，而后继续执行可运行线程，直到就绪表再次变空



# 远程过程调用：实现

对于多进程的服务器

- 在启动指派器时，通常首先要求内核安排一个为远程请求服务的进程池。其中的进程用于执行前面讨论的那些操作

如果使用的是线程与进程一一对应的语言或者库，其中的每个进程将反复从内核接收消息，而后调用适当的桩子程序

对更一般的线程包：

- 每个进程不断从就绪表取出线程并执行它，直至就绪表变空
- 遇到就绪表空的进程将通过系统调用进入内核获取下一条消息
- 只要可运行线程的数目大于等于进程数，进程就不接收新消息
- 一旦可运行线程的数目小于进程数，空闲的进程就会通过系统调用进入内核，如果有等待请求就将其取出；否则就在内核里阻塞，直至出现了新请求

# 总结

并发性是目前计算机科学技术领域最重要的理论和技术挑战

- 已经有了许多理论和技术成果，但还很不成熟
- 相关问题将长期成为计算机科学技术的研究热点
- 多核系统广泛存在，使我们对这方面的深入理解变得更为急迫

在理论和实践中，通常把并发系统分为基于共享存储和基于消息的两类

并发进程（线程）有许多创建方式，包括比较结构化的（如静态的，**cobegin**等）和更灵活的（如 **fork/join**）

目前的主要问题是缺乏程序员使用方便的并发性编程的高级抽象

- 写并发程序时需要考虑大量的控制细节
- 不合适的细节处理可能导致各种不当的运行行为，如死锁、活锁等

本章还讨论了许多并发机制的细节，包括调度器、各种同步/互斥机制等等