

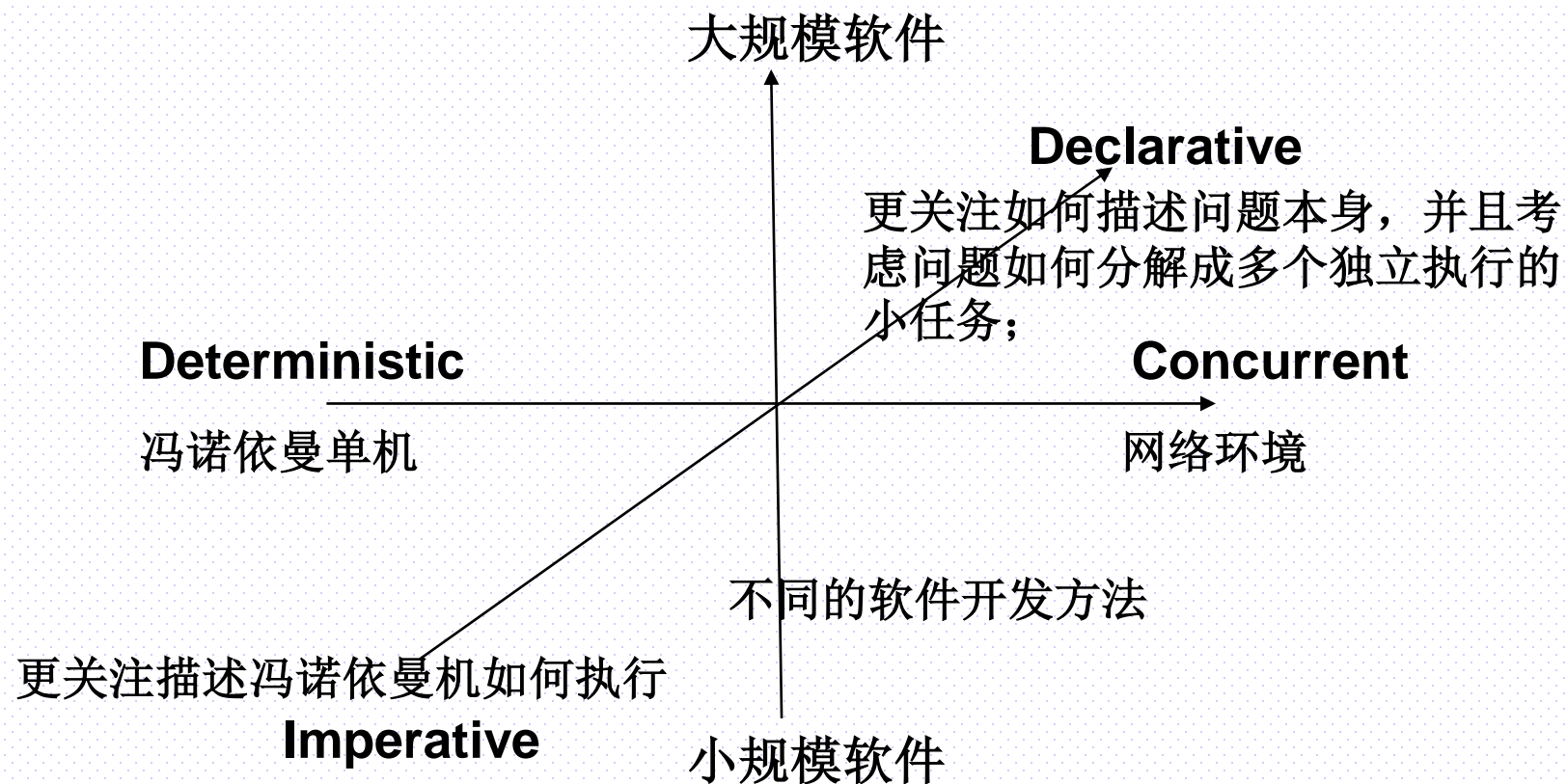


# 第七章 并发程序设计语言

- 多CPU/GPU

- 网络

➡ **Concurrent**



## 主要内容:

- 并发程序设计的基本概念
- 并发程序带来的问题
- 需要解决的基本问题
- 程序语言示例

## 并发 (concurrency) 和并行 (parallelism) 的概念

- 并发性(concurrency), 又称共行性, 是指能处理多个同时性活动的的能力, 并发事件之间不一定要同一时刻发生。
- 并行(parallelism)是指同时发生的两个并发事件, 具有并发的含义, 而并发则不一定并行。

另一种理解:

- 并行强调的是多个执行活动同时处于运行状态之中, 强调其相互独立性。这里关心并行算法、并行系统、并行体系结构等等
- 并发强调的是多个执行活动之间的关系和相互作用, 这里关心的是互斥、同步、通讯、资源的共享和竞争等等

第三种解读:

并发和并行的区别就是一个处理器同时处理多个任务和多个处理器或者是多核的处理器同时处理多个不同的任务。

前者是逻辑上的同时发生 (simultaneous), 而后者是物理上的同时发生。

# 并行与并发

- 程序设计语言中并发性和底层硬件的并行性是独立的概念。
- 硬件操作如果在时间上重叠就会并行地发生。
- 能够并行执行（但又非必需）的操作称作并发操作。
- 没有并行性的硬件，语言中同样可以有并发性。没有并发性的语言可以有并行执行。
  - 物理并发
  - 逻辑并发
  - 物理并发和逻辑并发允许作为程序设计方法学中的概念。

# 基本概念

- 并发程序设计模型
- 并行计算的硬件环境
- 程序与进程，线程与进程
- 原子动作
- 进程交互

# 基本概念

- 并发程序设计中最基本的概念是进程。
- 这里是指具有唯一控制线程的顺序计算。
- 顺序计算的线程就是程序控制流所能达到的所有程序点的序列。
- 进程间的交互：通信和同步。
  - 通信：进程间的数据交换
    - 显式消息
    - 共享变量
  - 同步：在一个进程的线程与另一个进程的线程间建立联系，涉及进程间交换控制信息
- 通信和同步用来描述进程间的竞争与合作关系

# 并发程序设计的模型：基本概念

- 1.共享变量模型
- 2.消息传递模型
- 3.数据并行模型
- 4.面向对象模型

是目前最灵活，最常用的模型

新的并发模型



# 基本概念

## I.共享变量模型

- 共享变量模型用限定作用范围和访问权限的办法，对进程寻址空间实行共享或限制，即利用共享变量实现并行进程间的通信。
- 为了保证能有序地进行IPC(Inter-Process Communication )，可利用互斥特性保证数据一致性与同步。

# 基本概念

- 共享变量模型与传统的顺序程序设计有许多相似之处。程序员只需关心程序中的可并发进程，而无需关心进程间的数据交换问题。
- 共享变量的数据一致性、临界区的保护性访问由编译器与并行系统来维护。
- 共享变量模型具有编程简单、易于控制的特点，但若在多处理机、机群系统上实现时则会导致系统开销增大，因此共享变量模型常用于共享存储多处理机，而不用于多处理机、机群系统。

# 基本概念

## 2.消息传递模型

- 消息传递模型是指程序中不同进程之间通过显式方法(如函数调用、运算符等)传递消息来相互通信，实现进程之间的数据交换、同步控制等。
- 消息包括指令、数据、同步信号等。
- 程序员不仅要关心程序中可并行成分的划分，而且还需关心进程间的数据交换。
- 消息的发送、接收处理将增加并行程序开发的复杂度。
- 但是它适用于多种并行系统，如多处理机、可扩展机群系统等，且具有灵活、高效的特点。

# 基本概念

## 3.数据并行模型

- 数据并行模型是指将数据分布于不同的处理单元，这些处理单元对分布数据执行相同的操作。
- 数据并行程序使用预先分布好的数据集。运算操作之间进行数据交换操作。
- 数据并行操作的同步是在编译而不是在运行时完成的。
- 数据并行模型中的SIMD模型可用于向量机、多处理机等并行计算机，而SPMD模型则可用于多处理机、机群系统。

# 基本概念

## 4.面向对象模型

- 面向对象模型是近几年随着面向对象技术的发展而提出的。它基于消息传递，但并行处理单位却是对象。
- 在这种模型中，对象是动态建立和控制的。处理是通过对象间发送和接收消息来完成。
- 面向对象模型具有简洁灵活的特点，适合多种平台，但系统开销较大。

# 基本概念

- 并行程序设计语言就是在这些并行程序设计模型的基础上，提出对并行性的描述方法、并行单元间协同的描述方法，以及该语言适用的并行计算环境。
- 无论是哪种模型，那个语言，都需要解决两个问题：进程管理和通讯。
- 事实上，模型之间的区别主要体现在对通讯实现方式的不一样上。

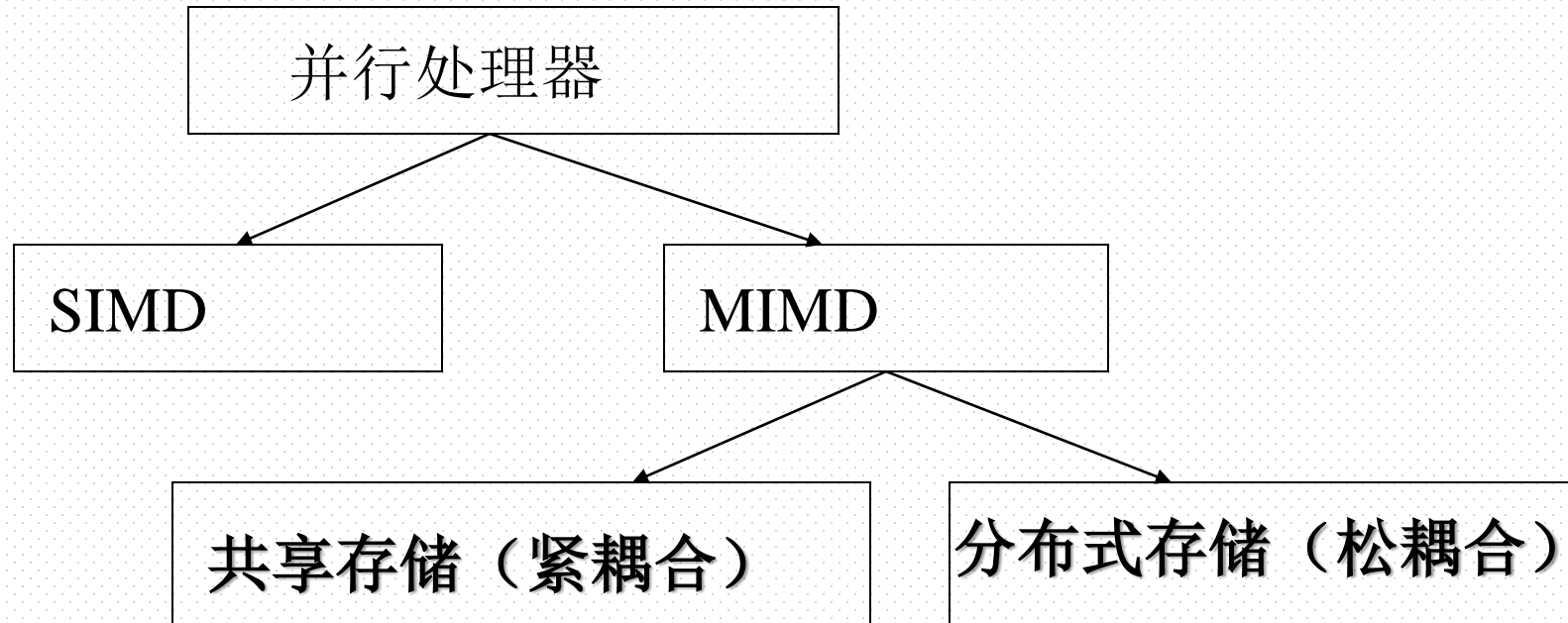
# 基本概念

## ■ 硬件环境

### ■ 根据Flynn的分类法:

- SISD: 一组可执行代码装入一个机器内存后, 以一个CPU, 一组数据执行一次。它属于SISD执行模式(即单指令流单数据流的简写)。物理上对应为单处理器的顺序程序。
- SIMD: 一组可执行代码装入后, 可以依次执行多个进程, 它属于SIMD, 单指令流多数数据流。对应为单机多处理器的主机或单CPU的分时系统、阵列机组。
- MISD: 在多机或多处理器上各有自己的可执行代码。协同完成一组数据的计算, 是MISD多指令流单数据流系统。对应为分布数据流机。
- MIMD: MIMD则为多指令流多数数据流系统, 对应为一般分布式系统(有多个不同的处理机, 运行各不相同的进程)。局域网和广域网就属于此列。如果网上协同运行一个程序作业, 则为以MIMD系统实现的并发程序。

# 并行处理器谱系





# 硬件的基本并行方案

- 输入/输出的并行性
  - 对于资源的占用处理：同步
  - 简单粗暴的处理方式：轮询（忙等待）
- 中断和分时
  - 看上去在并行，同一时间只有一个程序运行
  - 中断与时钟协同
- 多处理器器组织机构
  - 同步或异步的通信
  - 机器分离与消息，不可控的延迟
- 反应式系统
  - 对键盘、鼠标、窗口等使用独立的进程

- 共享存储多用于同类多CPU的单机上，所有CPU处理的进程都共享公共的数据。
- 分布式存储是松耦合的计算机群体，它对应为多计算机的簇（cluster）。
- 和一组计算机的集合不同之处在于：它们各自的存储是被大家共享的，它们互连。每个计算机只是“整个”计算机中的一个节点，是今后高性能、可伸缩、高可靠性计算机的发展方向。

# 基本概念

- 并行程序设计语言都要解决：
  - 并行进程的描述与管理
  - 进程间数据分布、传递的描述与管理
  - 以及进程间同步协同的描述与管理问题。

# 基本概念

- 一个程序的一次执行叫做一个进程(process)
- 狭义定义：进程是正在运行的程序的实例
  - an instance of a computer program that is being executed
- 广义定义：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。

# 基本概念

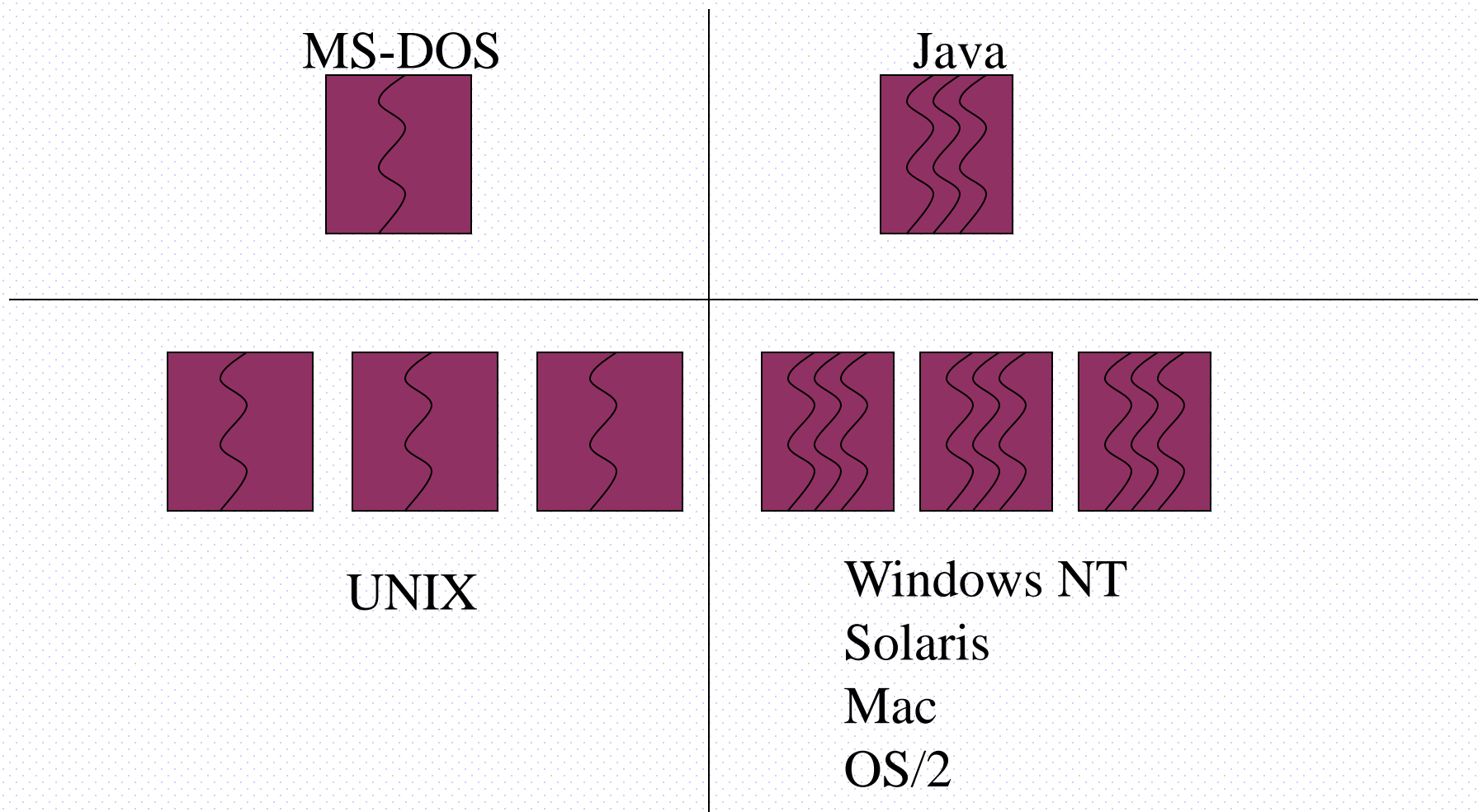
- 进程的状态
  - 运行：实际占用CPU
  - 就绪：可运行，因等待而暂时停止
  - 阻塞：等待外部召唤
  - 新生态和死亡态
- 从一个进程流向另一个进程的值的序列称为流。
  - 通过流连接起来的进程称为协程。包含0或多个输入或输出流。
- 管道：用管道可以将传统的顺序程序组合成为一个并发程序。一系列输入输出相衔接的管道称为管道线。

# 基本概念

线程是共享资源的轻量级进程（lightweight process）,它也有线程执行状态，也有其静态存储和局部变量。

- 进程用于把资源集中到一起，线程是在CPU上被调度的实体。
- 传统的OS支持单线程的计算模式。单用户的MS-DOS和多用户的UNIX就是例子，即使UNIX是多线程交互，每一进程之中只有一线程。

右侧上图，一个进程多个线程对应为**Java**虚机的计算模式。而当今所有**OS**均发展为右下角的多进程和多线程的计算模式。



# 基本概念

原子动作是一次“立即”执行完的“顺序”动作。至于是否真正不再分就不一定了。原子动作一般定义在语句级的事件(event)上

事件：是本程序表示的状态有了变化。



# 任务 (TASK)

- 任务是一个可以与同一程序的其它单位一起被并发执行的程序单位（类似子程序）
- 特性：
  - 可以隐式地启动
  - 当一个程序单位调用一个任务时，它在继续自己的工作之前并不需要等待任务执行的完成。
  - 当完成一个任务的执行时，控制可能会也可能不会返回到启动任务执行的程序单位。
- 重任务和轻任务
- 任务的相关性

例：PL/I的多任务

PL/I的并发进程是任务TASK，它可以定义语句级的事件。

P是一个进程，它并行执行Q进程，则P进程的正文可以写：

```
DECLARE X EVENT
```

```
:
```

```
CALL Q(APT) TASK (X) //激活Q
```

# 进程交互

## (I) 独立进程 两进程并行但不相关

设进程为事件序列, 若有C,K两并行进程, 可表达为 $C||K$ . 其中:

$$C=\{C_1, C_2 \dots C_n\}$$

$$K=\{K_1, K_2, \dots K_m\}$$

独立进程是两进程内任何事件 $C_i, K_j$ 的执行都不依赖对方

# 进程交互

## (2) 竞争进程 两进程竞争同一资源

临界段(Critical section): 具有并发处理资源的代码段

竞争进程一般形式是:

C : loop	K : loop
入口协议	入口协议
临界段	临界段
出口协议	出口协议
非临界段	非临界段
end loop	end loop

- 入口协议一般是按所设共享变量判断能否进入。出口协议则改置条件值。
- 为确保进程的确定性, 利用共享变量“通信”协调
- 如: 设C,K定义如前,  $C_i$ 必须先于 $K_j$ ( $K_j$ 要用到 $C_i$ 的结果)的执行, 即其它事件先后无所谓。

# 进程交互

## (3) 通信进程 两进程有协议的信息交换

- 同步(synchronous)通信 指两进程进度各不相同,但必须同步到达通信点。若一方未到,另一方等待,直至完成信息交换。
  - 交换后各自执行各自进程则为单向同步通信。
  - 如果交换后,发送方一直等待接受方执行的结果,拿回结果后再各自执行自己的进程为双向同步通信。
  - 合作同步与竞争同步
- 异步(asynchronous)通信 一般要借助相当大的邮箱。两进程以各自速度执行,发送方有了信息投入邮箱,并继续执行自己进程。接受方在认为合适时从邮箱获取信息。
  - 一般不竞争邮箱且为单向通信,当然也可做成双向的。
- 定向/广播式通信 所谓定向是发送方指明接受方,而广播式通信发送方只向公共信道发送信息,任何共享该信道的成员均可接受,所以是异步通信、单向的。

# 主要内容

- 并发程序设计的基本概念
- 并发程序带来的问题
- **需要解决的基本问题**
- 程序语言示例

# 带来的问题

- (1) 速度依赖
- (2) 输入值依赖
- (3) 不确定性
- (4) 死锁
- (5) 死等

# 带来的问题

## (I) 速度依赖

- 并发程序执行结果，取决于顺序成分进程执行的相对速度。
- 对于并发且有实时(real time)要求的程序，执行结果还取决于绝对速度。
- 并发程序调整相对速度的办法是延迟快进程。把进程挂起来(进入悬置态)待到指定条件满足才唤醒该进程。
- 其基本原语是:  
await<表达式> do <语句 | 进程>

# 带来的问题

## (2) 输入值依赖

同一并发程序两组数据输入可能会有很大差别。



# 带来的问题

## (3) 不确定性

- 顺序程序两次同样值的测试,一般情况下都是一致的。即所说的再现。
- 并发程序因上述原因往往没有确定的结果值。对于有副作用的函数或表达式这种先后次序的差异影响则更大

## 带来的问题

(4) 死锁 (deadlock) 是一种状态，由于进程对资源有互相不兼容的要求而使进程无法进展。

- 受到排斥 进程永远访问不到所需资源
- 循环等待 进程资源分配链形成一封闭回路
- 无占先(no preemption) 进程无法放弃所占的、其它进程需要的资源。所谓占先，只要所据资源的进程未处于使用状态，另一优先级高的进程有了要求，则此资源被后者占去
- 把持(wait and hold) 相互以占有对方资源为放弃已占资源的先决条件
- 活锁：无法取得进展

# 带来的问题

- 解决死锁的方法:
  - 利用工具作静态死锁检测，可以避免或减少死锁出现的可能
  - 或事前，让进程同时提出所有需要的资源，消除把持条件，或强行给资源排序，按此顺序满足要求，消除循环等待条件
  - 或事前，为调度程序声明最大的资源需求
  - 一旦出现，最笨的办法是重新启动，试换数据，找出原因改正之。事后重试解决
  - 一旦出现，找出死锁地点，夭折某些事件或进程或设置检测点

# 带来的问题

## (5) 死等 (starvation)

- 相互竞争的进程如果都满足进入某一资源条件,一般采用排队的先来先服务原则。相对最公平,但有的进程占用一种资源时间过长,致使其它资源长期闲置。
- 适当地让它等待可以解放很多占时少而重要的进程,这样更公平。因此,除了先来先服务而外,在调度例程中约定或在条件中加入优先级表来达到此目的。
- 调度程序则按此优先级和先来后到统一调度。如果优先级不当就会造成某些进程永远处于阻塞态,死等(但不是死锁)。死等是不公平调度引起的。
- 解决的办法是在改变某些进程的优先级,在公平性和合理性上作某种折衷



# 主要内容:

- 并发程序设计的基本概念
- 并发程序带来的问题
- **需要解决的基本问题**
- 程序语言示例

# 并发语言需要解决的基本问题

- 安全性(safety)是程序在执行期间不会出现异常的结果。对于顺序程序指其最终状态是正确的。对于并发程序指保证共享变量的互斥访问和无死锁出现
- 活性(liveness)是程序能按预期完成它的工作。对于顺序程序指程序能正常终止。对于并发程序指每个进程能得到它所要求的服务;或进程总能进入临界段;或送出的消息总能到达目的进程,活性深深受到执行机构调度策略的影响
- 公平性(fairness)指在有限进展的假设下没有一个进程处于死等状态。
  - 无条件公平性:调度策略如能保证每个无条件的原子功能均能执行
  - 弱公平性:在具有条件原子动作时,若条件原子动作能执行并依然保持无条件公平性,则为弱公平性
  - 强公平性:条件原子动作一定能执行,则为强公平性

# 并发语言需要解决的基本问题

- 进程管理：创建；起动（或恢复）执行；阻塞(或叫冻结)；停止执行；阻塞父进程创建子进程；撤销进程等六种操作。
- 通讯机制（信息交流和同步）：基于共享变量的；基于消息传递的

难点



- “我并不认为我们已经找到了适合并行计算机的正确程序设计概念。如果将来真能找到，几乎可以肯定它们与我们今天所了解的概念很不一样。”

— Brinch Hansen[1993]

- 我不相信有并行计算机的一组所谓“正确的”程序设计概念。

— Hoare[1993]

# 基于共享变量的通讯机制

## 一、忙等待(busy wait)

我们把指示变量叫做lock(锁)，每次测试临界段是否锁定。竞争进程以测定进入条件(锁)保持协调地进入临界段, 我们说它在**语义上**保证了条件同步。

锁就是条件，协调就是同步。

请注意，此时未设同步原语。

程序员也无法阻塞停止某个进程。如果有多个进程竞争进入临界段，则每个进程都要轮流测试锁。这就是著名的自旋锁 (spin lock)，其算法如下：

## program SPIN\_\_LOCK:

```
var Lock := false;
```

```
process P1::
```

```
  loop
```

```
    when not lock do  // 条件同步
```

```
      lock := true;    // 入口协议
```

```
      临界段;
```

```
      lock := false;   // 出口协议
```

```
      P1的非临界段;
```

```
    end do;
```

```
  end loop;
```

```
end p1;
```

```
process pn
```

```
:
```

```
end SPIN__LOCK
```

```
process P2::
```

```
  loop
```

```
    when not lock do
```

```
      lock := true;
```

```
      临界段;
```

```
      lock := false;
```

```
      P2的非临界段;
```

```
    end do;
```

```
  end loop;
```

```
end P2;
```

- 上述算法如果在多处理器的条件下, 进程严格同时到达, 对资源的竞争变成对指示变量查询和更改的竞争, 要取决于操作系统对公用主存储器的存取访问的排序。
- 如果某进程进入循环且正在更新lock为true期间, 第二个进程又访问了lock(为false), 那么它也进入临界段。互斥得不到保证。
- 为此, 寻找以忙等待实现互斥同步的算法, 从65年到81年有许多名家写了上百篇论文, 最后peterson的算法(1981)获得满意的解。算法如下:

# program MUTUAL\_EXCLUSION

```
Var enter1 : Boolean := false;  
    enter2 : Boolean := false;  
    turn   : String := "P1";    // 或赋初值 "P2"
```

```
process P1::
```

```
    loop
```

```
        enter1 := true;
```

```
        // 以下三行入口协议
```

```
        turn := "P2";
```

```
        while enter2 and turn = "P2" do skip;
```

```
        // 跳至循环末端
```

```
        临界段;
```

```
        enter1 := false;    // 出口协议
```

```
        P1的非临界段;
```

```
    end loop;
```

```
end;
```

```
end.
```

```
process P2::
```

```
    loop
```

```
        enter2 := true;
```

```
        turn := "P1";
```

```
        while enter1 and  
            turn = "P1" do skip;
```

```
        临界段;
```

```
        enter2 := false;
```

```
        p2的非临界段;
```

```
    end loop;
```

```
end;
```

## 二、信号量 (semaphore)

Dijkstra 首先理解到忙等待的低级和设计麻烦, 提出了完整的信号量(semaphores)理论(1968)。

信号量是一种数据结构, 包含一个非负整值变量  $s$ , 和两个操作  $P(\text{passeren})$ ,  $V(\text{vrygeren})$ , 即  $\text{wait}(\text{等待})$  和  $\text{signal}(\text{示信})$ 。

信号量是守卫(guard)机制的实现。

$V$ 操作发信号指示一个事件可以出现,  $P$ 操作延迟所在进程直至某个事件已经出现:

**$P(s) : \text{await } s > 0 \text{ do } s := s - 1;$**  // ‘await’ 表达延迟的原语

**$V(s) : s := s + 1;$**

## Program MUTEX\_\_EXAMPLE;

例  
以  
信  
号  
量  
实  
现  
的  
两  
进  
程  
互  
斥

```
var mutex : Semaphore := 1;
process P1::
    loop
        P(mutex);    // 入口协议
        临界段;
        V(mutex);    // 出口协议
        P1的非临界段;
    end loop;
end p1;
process p2::
    loop
        P (mutex);
        临界段;
        V (mutex);
        P2的非临界段;
    end loop;
end;
end.
```

例  
以  
信  
号  
灯  
实  
现  
的  
两  
进  
程  
互  
斥

wait(aSemaphore)

if aSemaphore的计数器  $> 0$  then

    aSemaphore 的计数器减一；

else

    将调用任务置于aSemaphore的队列中

    尝试将控制传递给一个就绪任务

    （如果就绪任务队列为空，？）

end if

release(aSemaphore)

if aSemaphore的队列为空（没有任务等待） then

    aSemaphore 的计数器加一；

else

    将调用任务置于就绪队列之中

    将控制传递给aSemaphore队列中的一个任务

end if





信号灯变量s, 当只有一个资源时取值{0,1}就够了, 此时称为二值信号灯。


P, V操作很容易扩大到n个进程竞争一个临界段。也可以将二值信号灯劈开分别实施同步警卫功能。以下是生产者/消费者著名问题的同步解。

# program PRODUCER\_\_CONSUMER

```
var buf : TYPE;    // 任意类型TYPE
var empty : sem := 1, full : sem := 0; // 两信号变量初始化
process PRODUCER [i:1.. J]::
  loop
    PRODUCER[i] 产生一条消息m;
    deposit : P(empty); // 存入消息m的三个操作
                buf := m;
                V(full);
  end loop;
end;
```

```
process CONSUMER [j:1..N]::
  loop
    fetch : P(full); // 从buf取出消息m的三条操作
    m := buf;
    V (empty);
    CONSUME R[j]取出这条消息m;
  end loop;
end;
```

```
end PRODUCER__CONSUMER.
```



将信号变量s一分为二(empty, full)简化了传递方向。称劈分二值信号灯(split binary semaphore)。

这个算法保证了互斥，无死锁。

将P,V操作扩充到多进程, 多资源(多个临界段)也是很容易的。

例如, 我们可实现q个缓冲区的多生产、消费者问题。其算法如下:

```
semaphore access, fullspots, emptyspots;
```

```
Access.count = 1;
```

```
fullspots.count = 0;
```

```
emptysports.count = BUFLen;
```

```
task producer;
```

```
loop
```

```
  //产生value
```

```
  wait(emptysports); //等待一个位置
```

```
  wait(access);
```

```
  DEPOSIT(VALUE);
```

```
  release( access);
```

```
  release(fullspots); //增加占满了位置
```

```
end loop
```

```
end producer;
```

```
task consumer;
```

```
loop
```

```
  wait(fullspots); //确保它不是空
```

```
  wait(access); //等待访问
```

```
  FETCH(VALUE);
```

```
  release(access);
```

```
  release(emptysports); //增加为空的位罝
```

```
  //消费VALUE
```

```
end loop
```

```
end consumer;
```

一个信号量本身是一个共享数据对象，对于信号量的操作也会遇到竞争问题。

选择互斥是更为复杂的同步机制。用P, V操作也可以实现选择互斥。信号量的操作不能中断。

经典的例子是哲学家就餐问题。

一张圆桌坐了五位哲学家，桌上放有一大盆通心粉，但只有五把叉子。而吃通心粉必须有两把叉子。一旦据有两把叉子的哲学家就可以吃，否则它只好利用等待的时间思考。如果每人拿一把叉子等待其它人放下叉子，就产生死锁。

通心粉是共享资源，但叉子是只有两个哲学家共享的资源。每个哲学家的行为过程即为一进程。第 $i$ 个进程只和第 $i$ 和 $i+1$ 个叉子有关。故算法如下：

program DINING\_\_PHILO:

var forks [1..5]: sem:= (5\*1);

process PHILOSOPHER [i: 1..4]:

loop

P(forks [i]); P(forks[i+1]);

吃通芯粉;

V(forks[i]); V (forks [i+1]);

思考问题;

end loop;

end;

end DINING-PHILO.

process PHILOSOPHER [5]:

loop

P(forks[1]); P(forks[5]);

吃通芯粉;

V(forks[1]); V(forks[5]);

思考问题;

end loop;

end;

以上以信号灯实现互斥同步均隐含使用了await等待原语。事实上多数分时处理的机器，单主机多处理器的机器是用系统调用并行核(或叫管理程序)实现的。

进程创建就绪后将该进程的描述子(descriptor)入队，即就绪进程表，等待P操作的完成。这样，进程处于就绪或阻塞状态且未运行。此时P, V操作的语义解释是：

P(s): if  $s = 1$  then  $s := 0$   
      else 置进程于queue中等候

V(s): if queue  $\neq$  empty then 从queue中消除一进程,令运行程序执行它  
      else  $s := 1$

## ■ 合作同步的程序设计安全问题

- 不存在静态的方式来检测信号量使用的正确性；这种正确性完全依赖于信号量所出现的程序的语义。
- 如果producer不包括wait(emptyspots)，会造成缓冲区上溢，反之亦然。
- 遗漏任何释放操作，会造成死锁。

## ■ 竞争同步的可靠性问题

- 没有wait(access)，引起缓冲区的非安全访问。
- 任意任务不包含release(access)，也会造成死锁



# 基于共享变量的通讯机制

## 三、条件临界区

- 条件临界区(Condition Critical Region简称CCR)将共享变量显式地置于叫做资源的区域内
- 每个进程在自己的进程体内指明要访问的条件临界区，而同一临界区可出现在不同进程之中(谁进谁用)
- 首先在资源中声明共享变量: **resource** r(共享变量声明)，例: **resource** sema( s:int :=n )

使用共享变量: **region** r **when** B **do** S **end**

**region** sema **when** s>0 **do** s:= s-1 **end** // P操作

**region** sema **do** s:= s+1 **end** //V操作

## 例：用CCR实现的生产者与消费者

```
program PRODUCER_CONSUMER_CCR
  var buf:TYPE;
  var empty:sema,full:sema;
  resource sema:(empty := 1,full := 0);
  process PRODUCER [i:1..M]::
    loop
      PRODUCER[i] 产生一条消息m;
      deposit:
        region sema when empty>0 do empty := empty - 1 end;
        buf := m;
        region sema do full := full + 1 end;
      end loop;
    end;
  process CONSUMER [j:1..N]::
    loop
      fetch:
        region sema when full>0 do full:= full - 1 end;
        m=buf;
        region sema do empty := empty + 1 end;
        CONSUMER[j] 消费者取出的这条消息m;
      end loop;
    end;
end PRODUCER_CONSUMER_CCR.
```

## 条件临界区评价

条件临界区最主要的优点是概念清晰。此外：

- 无需辅助标志和变量即可描述共享变量的任何进程交互
- 程序编译时即可保证互斥
- 一个进程创建一个条件不需顾及其它条件是否与此条件有关
- 易于程序正确性证明
- 体现了共享数据传递的方便

它的致命缺点是低效(和信号灯相比)。此外：

- 进程和共享变量耦合太紧
- 临界区利写不利读，一多了就太散，因而也难修改

## 四、监控器

Dijkstra建议是把分散在整个程序中的**region**语句进一步集中成为一个模块叫做监控器(**monitor**)。

```
program monitor
```

```
    monitor Mname::
```

```
        共享数据声明并初始化;
```

```
        proc op1 (<形参表1>) is <op1 体> end;
```

```
        ...
```

```
        proc opn (<形参表n>) is <opn体> end;
```

```
    end;
```

```
    process Pname [i:1..N]::
```

```
        局部数据声明并初始化
```

```
    begin
```

```
        :
```

```
        call Mname.opi (实参表);
```

```
        :
```

```
    end
```

```
begin
```

```
    初始化, 激活进程
```

```
end monitor.
```

## 例：有界缓冲区的监控器实现算法

```
monitor BOUNDED_BUFFER::  
  var buf[1..q]: TYPE;  
  var front := 1, rear := 1, count := 0;  
  var not_full:cond;           //当count < q示信为真  
  var not_empty:cond;          //当count > 0 示信为真  
  proc deposit (data :TYPE) is  
    while count = q do wait (not_full) end;  
    buf [rear] := data;  
    rear := (rear mod q) + 1;  
    count := count + 1;  
    signal (not_empty);  
  end;  
  proc fetch (var result :TYPE) is  
    while count = 0 do wait (not_empty) end;  
    result := buf [front];  
    front := (front mod q) + 1;  
    count := count - 1;  
    signal (not_full);  
  end;  
end BOUNDED_BUFFER.
```

- 管程：由一个共享变量集合和一些过程组成，在任何时刻它只能允许一个进程执行管程里的过程。
- 问题：进程可能被阻塞在管程里。（缓冲区满了，生产者占据管程又无法生产）
- 解决方案：保持一个阻塞进程的队列，让另一个队列中的进程执行V操作，将被阻塞的进程从队列中取出。

# 以监控器实现条件同步的技术

## (1) 复盖条件变量

进程等待列表过长问题。

两级条件，上级发布，下级自行匹配

## (2) 传递条件

有无占先对竞争的并发进程影响是很大的, 由于不占先在被唤醒进程执行之前, 监控器不能拒绝另一进程进入它. (见下例\*)

为了防止条件信号被偷, 发信号的进程直接将条件传入被唤醒的进程。(见下例\*\*)

## (3) 会合同步

进程交互是客户/服务器(C/S)关系时, 为此两交互进程必须会合(rendezvous)才能得到服务。如不能到达会合的同步点则要相互等待。(见下例\*\*\*)

例\* 以监控器作信号灯

monitor SEMAPHORE::

var s:= 0;

var pos:cond; //当s>0, pos示信为真

proc P( ) is

while s=0 do wait (pos) end;

s:= s-1;

end;

proc V( ) is

s := s+1;

signal (pos);

end;

end SEMAPHORE.



例\*\*： 以监控器实现的FIFO信号灯

```
monitor SEMAPHORE::  
  var s=0;  
  var pos :cond;    //当V中pos队列非空示真  
  proc P( ) is  
    if s>0 then s:= s-1 endif;  
    □ if s=0 then wait(pos) endif;  
  end;  
  proc V( ) is  
    if empty(pos) then s:= s+1 endif;  
    □ if not empty(pos) then signal(pos) endif;  
  end;  
end SEMAPHORE.
```

注:本例中“□”号表示和前一个语句并行执行的语句,以下同.

### 例\*\*\* 贪睡的理发师的模拟解

```
monitor BARBER_SHOP::  
  var barber := 0, chair := 0, open = 0;  
  var barber_available :cond           //当barber>0 示真  
  var chair_occupied :cond             //当chair>0示真  
  var door_open:cond                   //当open=0示真  
  proc get_haircut( ) is                //顾客调用  
    while barber=0 do wait (barber_available) end;  
    barber := barber+1;  
    chair := chair+1;  
    signal (chair_occupied);  
    while open=0 do wait (door_open) end;  
    open := open+1;  
    signal (customer_left);  
  end;  
  proc get_next_customer( ) is  
  proc finished_cut ( ) is  
end BARBER_SHOP.
```

} 见下一页

```
proc get_next_customer( ) is      //理发师调用
    barber := barber+1;
    signal (barber_available);
    while chair=0 do wait (chair_occupied) end;
    chair := chair-1;
end;
proc finished_cut ( ) is          //理发师调用
    open := open+1;
    signal (door_open);
    while open>0 do wait (customer_left) end;
end;
```

## 各种语言实现监控器时的原语语义差异

监控器有三个特征：

- 第一，监控器封装了共享变量，共享变量仅能由监控器内的过程访问。
- 第二，监控器内的过程都是互斥地执行的。因而共享变量不能并发访问。
- 第三，条件同步由wait和signal操作实现

监控器有时不一定必须互斥。也可以采用其它办法实现条件同步。

## (1) 实现条件同步的各种信号机制

- 自动信号AS: 只要wait加上条件就可以不用signal原语了. 即省去检查signal是否执行的开销, 程序员也不必操心是否用错。
- 信号和继续SC: 当无占先时发信号的进程继续执行。直至它进入等待或返回之前, 其它进程是不许进入监控器的。modula3即采用此种机制。
- 信号和出口SX: 既然被占了先, 发信号的进程也就不等了。立即从监控器出口或从过程返回。并发Pascal即采用此种机制。
- 信号和等待SW: 发信号的进程被人占先之后处于监控器内等待, 直到它能再次获得互斥访问, 恢复执行。Modula和并发Euclid采用这种机制。
- 信号和急等SU: 发信号进程被人占先之后也要等待, 但保证在监控器有新的进程进入之前先使它得到恢复。Pascal\_plus即采用这样的机制。

## 各种语言实现监控器时的原语语义差异

以上五种信号机制语义略有不同，但可从理论上证明它们是等价的。  
即以一种机制可模拟另一种机制。

实现费用不同，对某些类型问题表达的方便性不同。也正是不同语言各自钟爱它们的原因。

## (2) 嵌套监控器中的互斥

在磁盘调度器之类的应用中，一个进程首先要争取进入磁盘去寻址，找到地址后读/写，这样就要设计两个监控器：

一个管理粗的磁盘资源，进程进入或释放。另一个管理读/写区，进程互斥地读写。这两个监控器是嵌套的

每一时刻只有一个进程进入监控器，调用某个过程，我们称它是闭式调用。在嵌套监控器之中，这种方式容易引起死锁。

开式调用是若有嵌套调用发生时上层互斥自动解开，待调用返回后上层监控器又重新闭合(获得)互斥。

# 路径表达式

- 1974年Campbell和Habermann提出以路径表达式直接控制进程顺序的建议——**监控器中派生出来的一个重要分枝。**
- 路径表达式是就每一资源在其开始声明时，就在其上定义操作的约束。
- 当使用路径表达式实现同步要求时，不必将同步代码(或同步协议)编程到相应的过程处，这些同步代码可由编译程序根据所写出的路径表达式自动地生成。

进程A的一个执行由两个事件a和b组成，另一个进程X的一个执行可以由x、y、z组成，那么A和X的并发执行可能为：

a	b	x	y	z
a	x	b	y	z
.....				
x	y	z	a	b



## • 路径表达式

- `path deposit, fetch end` //deposit和fetch并发执行
- `path deposit; fetch end` //deposit必须先于fetch执行
- `path 1: (deposit; fetch) end`

//只能有一条路径(但可多次执行此路径), 两操作交替互斥执行.

- `path N: (1: (deposit); 1: (fetch)) end`

//deposit和fetch是一一对应地互斥激活, 先执行deposit, 完成的deposit个数不超过N次, 且可多于fetch完成的个数。

由路径表达式指明的同步约束, 编译时即可保证.

优点是程序员可直接控制过程的执行, 正文清晰。但当同步化依赖过程参数或监控器的状态时, 表达能力差。

# 基于消息传递的通讯机制

- 消息是信息传递的单元，按shannon的模型，信息源借助信道(channel)向信息目标发送消息。
- 信道成了并发进程共享的资源。信道是通信网的抽象，泛指进程间通信的路径。
- 信道由两个原语访问：send，receive。当某进程向信道发送一消息，通信就开始了。
- 需要该消息的进程，从信道上接受(获取)这条消息.数据流也随发送者传递到接受者。

## 基于消息传递的通讯机制

- 由于信道本身不能存储，变量只能存放在各个进程中，因而不能共享地访问，所以也用不着互斥机制。
- 由于只有所在进程能考察变量情况，条件同步编程与基于共享变量的大不相同。程序也不一定非要在一个处理器上执行，可以分布在多个处理器上，分布式程序因而得名。
- 反过来，分布式程序却可在单主机或多路处理器的(分时)系统上执行。此时把信道改成共享存储就可以了。

# 两类通信模式

- 基于消息传递的进程，通信虽然不靠共享存储间接实现，然而通信时也有同步异步之分。
- 同步消息传递：两进程都要执行到同步点直接交换数据。判定是否同步只看含send的进程是否有阻塞，等待接受进程的到來。如果是这样，一定是同步的。
- 异步消息传递：两进程谁也不等谁按各自速度执行自己进程，就完成了数据通信。其实现可以看作在接受进程中附带一很大的邮箱(缓冲时间差)。发送者按自己进程执行，将一条条消息发到邮箱。而接受者在自己合适的时候处理一条条消息各不相扰。

## 异步消息传递

**chan**<信道名> (<标识符1>:<类型1>, ..., <标识符n>: <类型n>)

其中<标识符i>:<类型i>为传到信道上的数据域名(可缺省)和类型。例如:

```
chan input (char);  
chan disk_access(cylinder, block, count : int,  
                 buffer: char*);  
char output [1...N] ([1...M]: char);
```

# 过滤器

- 根据消息中信息实现不同功能
- 从一个信道input接受字符流的过滤器进程char\_to\_Line。每当它接受满一行或见到CR符号就向输出信道output送出该行。CR为行结束符。

## •异步通信的过滤器

```
chan input (char), output ([1..MAXLINE]:char);
  char_to_Line::
    var line [1..MAXLINE]:char, i:int :=1;
    loop
      receive input (line [i]);
      while line [i] =CR and i<MAXLINE do
        i := i+1;
        receive input (line [i]);
      end
      send output (line); i:= 1;
    end loop;
    ...
```

# 文件服务器

- 操作系统中常用的文件服务器。它保持客户与服务器交谈的连续性。
- 为了访问外存上的文件，客户首先要打开文件，文件存在表示成功，客户才能发出一系列读/写要求。最后由客户关闭文件。
- 打开之后作读、写、关闭等操作，但打开的和操作的是同一文件(即连续交谈)。
- 为了简化，文件都是一种类型，差别只在空不空，所以文件服务器也是N个进程，一个管一个。每一个对应一个访问信道。但打开文件的信道是全局的。为了简化，客户号、服务器号均以整数表示。



## • 客户/服务器异步通信实现

```
type kind = enum (READ, WRITE, CLOSE);
chan open(fname:string *, clientid:int);
chan access[1..N] ( kind, TYPE);           //可增细节信息
chan open_reply[1..N] (int);
chan access_reply [1..N] (RTYPE, FLAG);
File_Server [i:1..N]::
  var fname:string*, clientid:int;
  var k:kind, args:TYPE;
  var more:bool := false;
  var 局部缓冲区, 快速缓存, 磁盘地址等;
  loop
    receive open (fname, clientid);
    //打开文件fname, 不成则等, 若成:
    send open_reply[clientid](i);
    more := true;
    while more do
      receive access [i] (k, args);
      if k=READ then 执行读操作;
      □if k=WRITE then 执行写操作;
      □if k=CLOSE then关闭文件;
      more := false;
      endif;
      send access_reply [clientid] (results);
    end;
  end loop
```

# 基于消息传递的通讯机制

client [j:1..M]::

**send** open (“foo”, j);           //第j个客户要打开foo文件

**receive** open\_reply[j](serverid)   //取server的号

    //准备访问数据k.args

**send** access [serverid] (k, args)

**receive** access\_reply[j] (results)   //取结果信息

# 同步消息传递

## •通信者的顺序进程CSP

### (1) 通信语句

$A::\dots B!e\dots$       $//B!e$  是输出语句，目标是B

$B::\dots A?x\dots$       $//A?x$  是输入语句，源自A进程

$\langle\text{目标进程}\rangle!$  信道名 ( $\langle\text{输出表达式表}\rangle$ )

$\langle\text{源进程}\rangle?$  信道名 ( $\langle\text{输入参数表}\rangle$ )

我们把求最大公约数做成一服务器进程GCD。客户进程client每向它发送一对整数，它求出结果发回客户。

```
GCD :: var x, y: int;  
loop  
  client ? args( x, y)  
  while x/=y do  
    if x>y then x:= x-y endif  
    □ if y>x then y:=y-x endif;  
  end;  
  client ! result (x);  
end loop;
```

client进程在准备好数据v1, v2和结果存储空间r之后，应有以下语句：

```
... GCD! args(v1, v2);  
GCD ? result (r)...
```

其中args, result是信道端口名。通信进程双方要指明对方写传递消息的输入/出语句，它的最大的限制是不能一到多地通信。

## (2) 选择通信

Dijkstra 1975年提出的(警)卫式命令。即设一布尔条件B, 仅当B为真才执行消息传递语句。所以, 也叫卫式通信。卫式通信语句的一般形式是:

B; C $\rightarrow$ S;

选择通信语句一般形式是:

if G1 $\rightarrow$ S1

□G2 $\rightarrow$ S2

...

□Gn $\rightarrow$ Sn

endif

- 过滤器的同步实现

```
sieve [1] ::var p:= 2, i:int;  
  print (p); sieve [2]! p;  
    for i := 3 to N by 2 do          //发送候选奇数  
      sieve [2] !i  
    end.  
sieve [j: 2..L]::var p:int, next:int;  
  sieve [j-1]? p; print (p) ; //只打印“第一个”收到的  
  loop  
    sieve[j-1] ? next;  
    if next mod p /=0 then  
      sieve [i+1] ! next endif;  
  end loop;
```

## • 客户/服务器的同步通信实现

### 文件服务器的同步实现算法

```
File_Server [i:1..N]::  
  var fname:string, args:TYPE;  
  var more :bool;  
  var 局部缓冲区, 快速缓存, 磁盘地址等;  
  while (c:1..M) client [c]? open (fname) do //如名为fname的文件已打开:  
    client ! open_reply( );  
    more := true;  
    while more = true do  
      if client [c]? read(args) then  
        调处理读入数据的过程;  
        client [c]! read_reply(results);  
        □if client [c]? write(args) then  
          调处理写的过程;  
          client [c] ! write_reply(results);  
          □if client [c]? close() then  
            调关闭文件操作;  
            more := false  
          endif;  
        endif;  
      end;  
    end.  
  end.
```

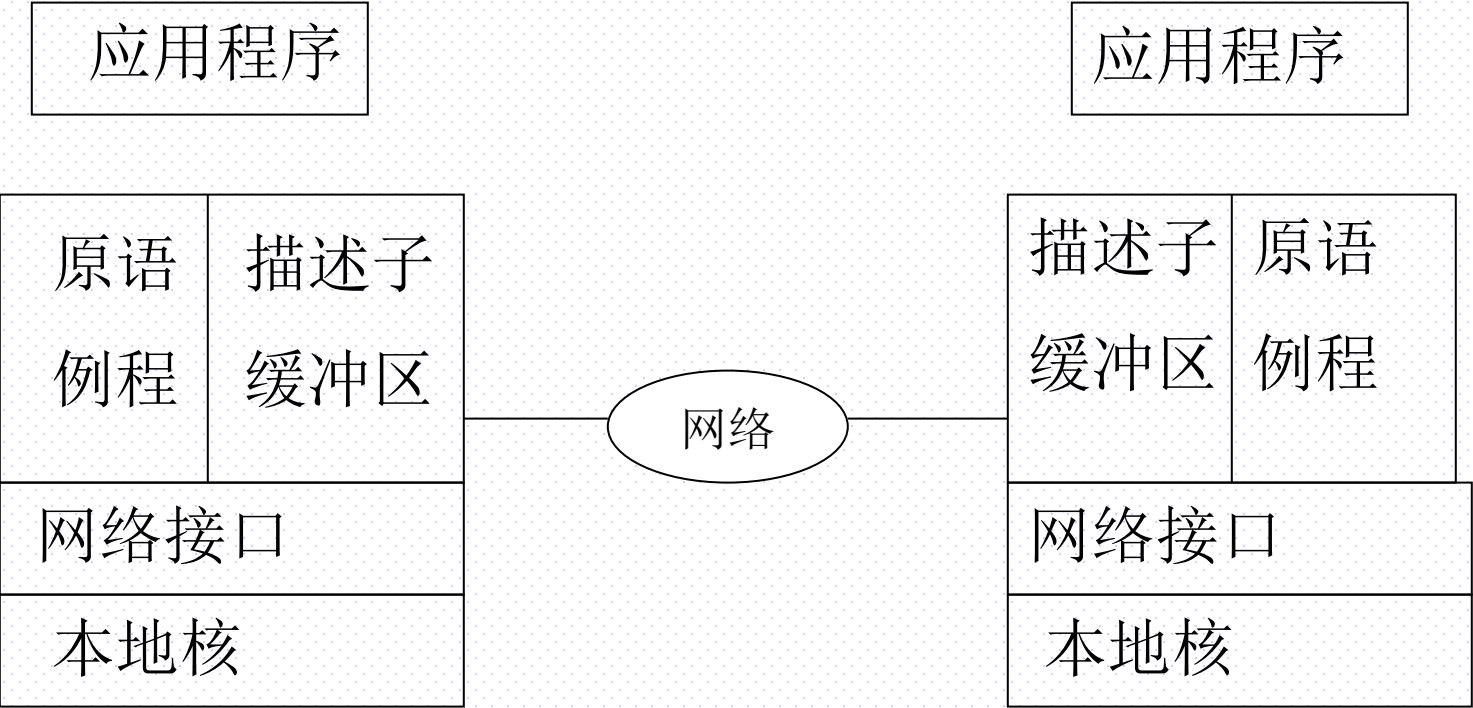
```
client [j:1..M] ::  
    var serverid : int;  
    while (i:1..N) File_Server[i]! open("foo") do  
        serverid := i;  
        File_server[i] ? open_reply(serverid);  
    end.  
    //以下写使用该进程的代码。例如，读可写：  
    File_Server[serverid] ! read(访问参数);  
    File_Server[serverid] ? read_reply(results);  
    ...  
    //最后要有关闭文件的消息
```



# 通讯机制的实现

- 显然上述的同步异步并行是依靠通讯原语实现的，原语的具体语义依赖于原语在具体机器上的实现（并发核）。

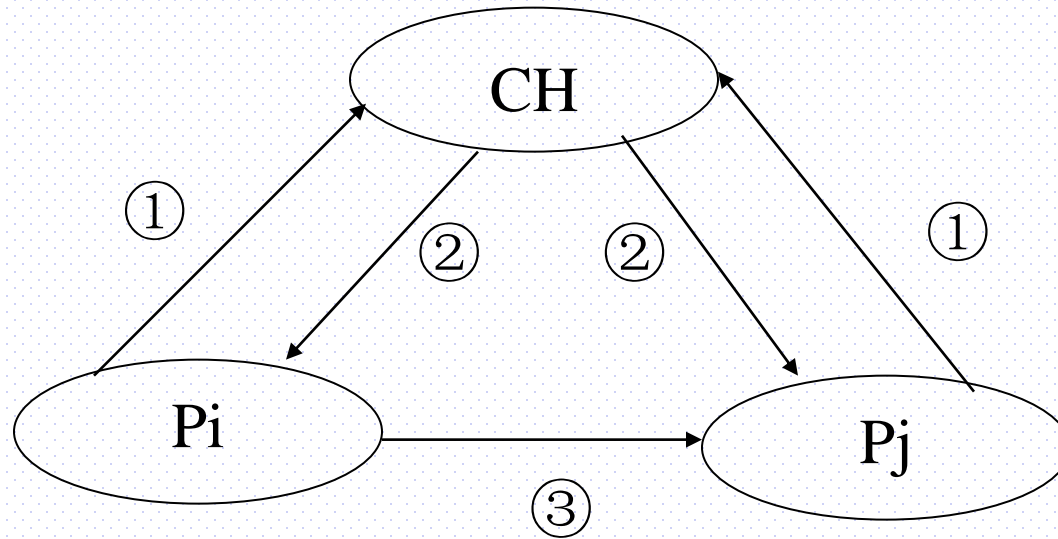
# 异步通信的底层实现



分布式并发核示意图

同步通信在最基本的操作上和异步是一样的。当前实现同步有两种策略：

(1) 集中式的同步实现



集中管理，它易于检查进程匹配情况。然而，在分布式系统中，这样多次来回通信增加了系统通信开销，往往成为并行核的瓶颈。

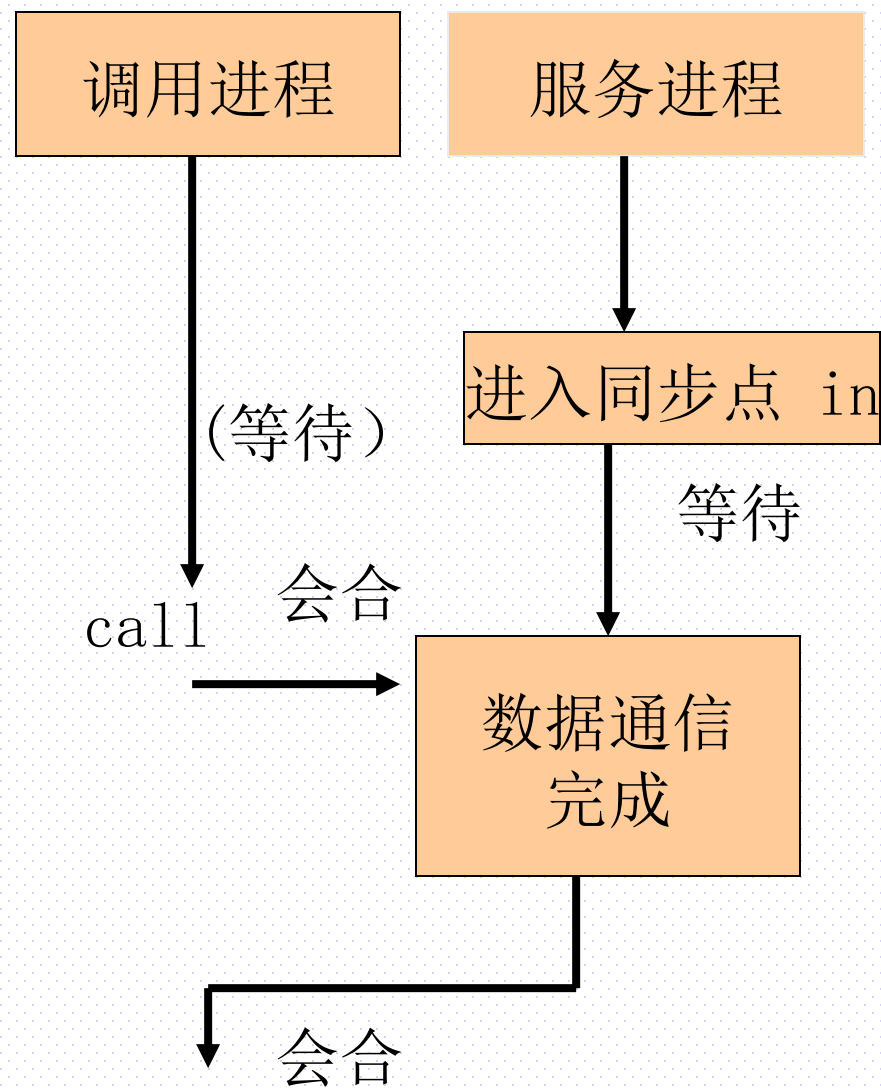
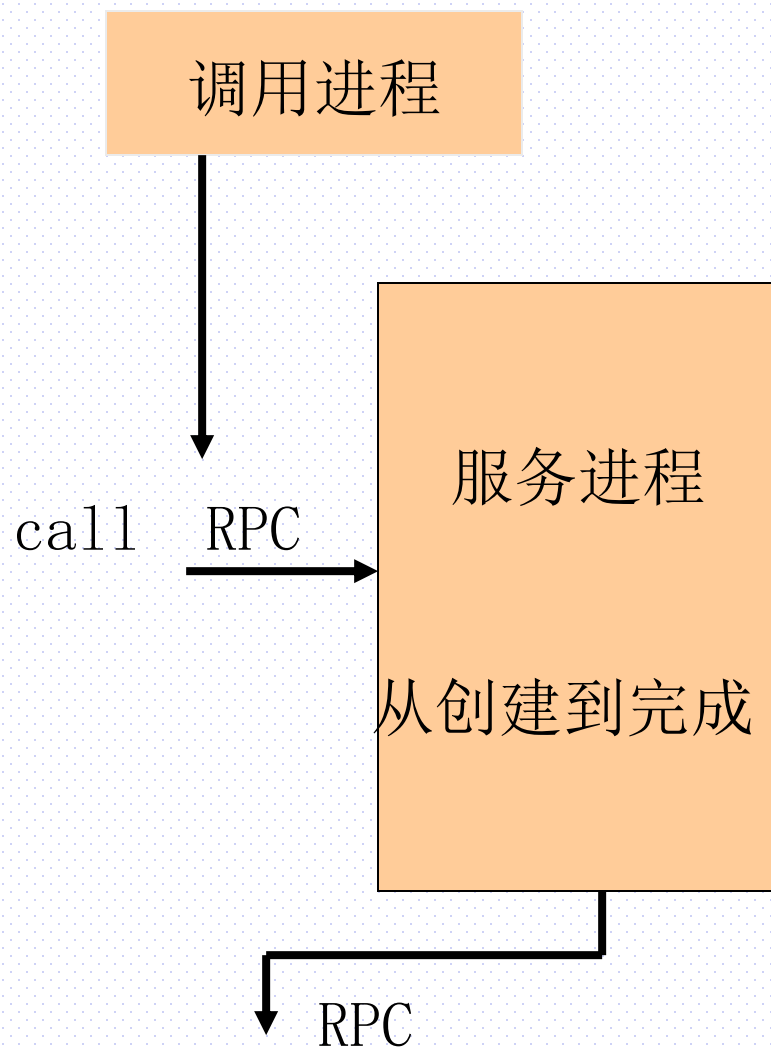
## (2) 分散式同步实现

- 用异步通信原语实现分散式同步通信，即为每个要求通信的进程设立一个匹配信道(输出语句为一端，输入语句为另一端)和一个应答信道。有输入语句的进程设阻塞等候队列。
- 如果输出/入语句不出现在警卫子句内处理程序要简单得多，输出语句出现在警卫子句中情况会变得更复杂。因此CSP和Occam的最初版本均不允许在警卫子句中有输出语句。

### (3) 远程过程调用和会合

- 以上异步和同步通信是以嵌入有通信语句或原语的进程为基本的通信单元。
- 70年代末，结构化程序发展为模块的高级形式，并发程序研究中也提出将监控器模块与同步消息传送结合的机制，即远程过程调用(Remote Procedure Call 简称RPC)。

# 远程过程调用和会合



# 远程过程调用RPC

**module** Mname

输出(外部可见)过程型构;

**body**

本模块内共享变量及初始化代码;

输出过程的体;

局部于本模块的过程和进程;

**end.**

调用方式:

**call** Mname.opname(AP\_list);

**call** opname(AP\_list);

**module** File\_Server

**op** readblk(field, offset:int; **res** blk [1..1024]:char);

**op** writeblk (field, offset:in; blk(1..1024):char);

**body**

**var** cache\_block:TYPE;           //磁盘缓存块

**var** queue :QTYPE;           //等待访问磁盘的队列

**var** 按需要的信号灯变量

**proc** readblk (field, offset, blk) **is**

**if** 需要的块不在cache\_block之中**then**

        将读要求存入磁盘的等待队列;

        等待处理读;

**endif**;

    blk := cache\_block;

**end**;

**proc** writeblk (field, offset, blk)

    选取一可写cache\_block;

    如必要, 将写要求入等待队;

    等待至磁盘可写;

    cache\_block := blk;

**end**;

Disk\_Driver ::loop

    等待磁盘访问要求;

    启动磁盘操作; 等待中断;

    唤醒等待此要求的进程并执行之;

**end loop**;

**end** File\_Server.



## 会合

- 会合由进程输出操作，因而，每当有远程调用时进程已经是激活了的。这和模块输出操作，远程调用时再激活进程不一样，故而得名，即以两活动进程相会合以求得同步。服务进程一般定义为：

Pname::各操作Opi的声明;

局部变量声明;

**in/accept** 语句和条件;     //接受RPC的入口点

opi(FRG\_list);

其它局部操作;

调用时和RPC一样:

**call** Mname.Pname.Opi(ARG\_list)

有界缓冲区是一典型过滤器，一些进程向缓冲区存入(**deposit**)数据，另一些进程从缓冲区取出(**fetch**)，缓冲区(**buffer**)本身是一个进程。会合机制的实现算法如下：

Buffer ::

**op** deposit(data:TYPE), fetch(**var** result:TYPE);

**var** buf[1..q]:TYPE

**var** front := 1, rear := 1, count := 0;

**loop**

**in** deposit (data) **and** count < n **do**

buf[rear] := data;

rear := rear mod n + 1;

count := count + 1; **end**;

**in** fetch (result) **and** count > 0 **do**

result := buf[front];

front := front mod n + 1;

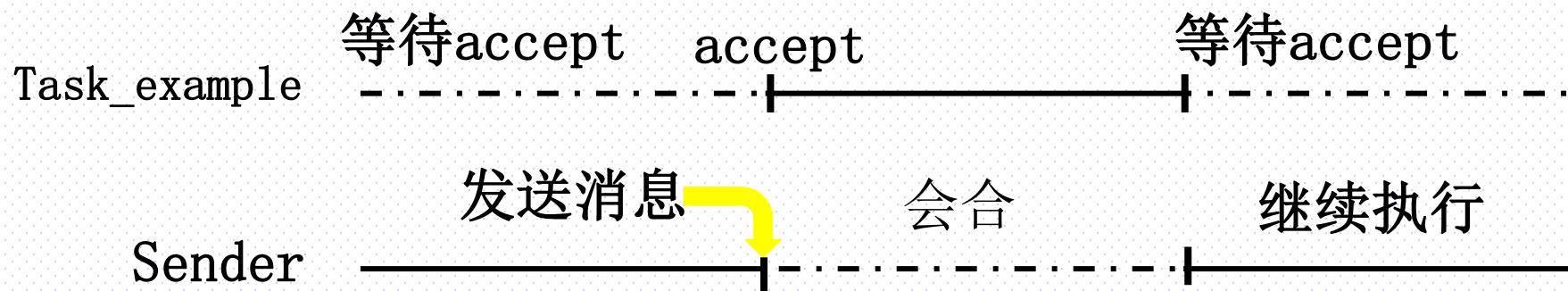
count := count - 1;

**end**;

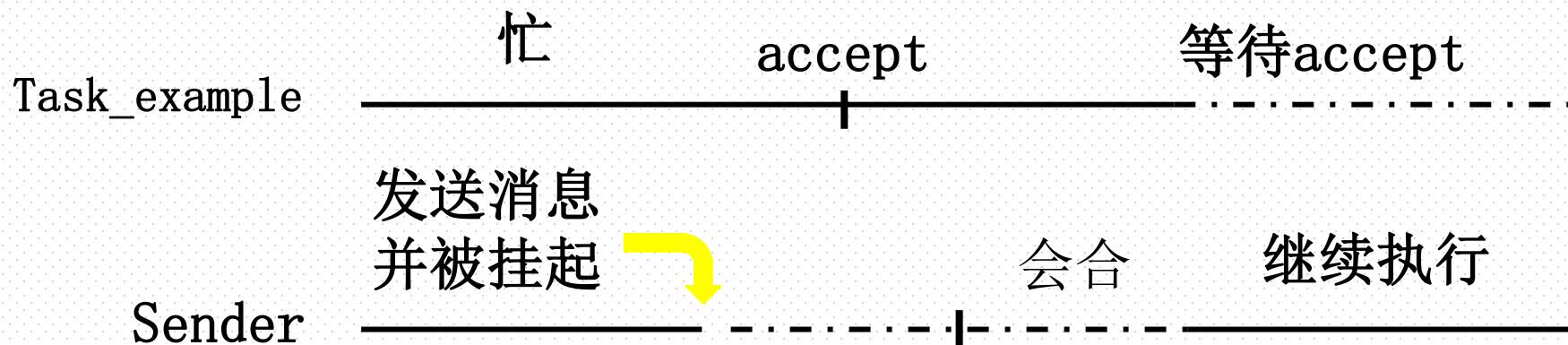
**end loop.**

# ADA会合时的两种方式

Task\_example等待Sender

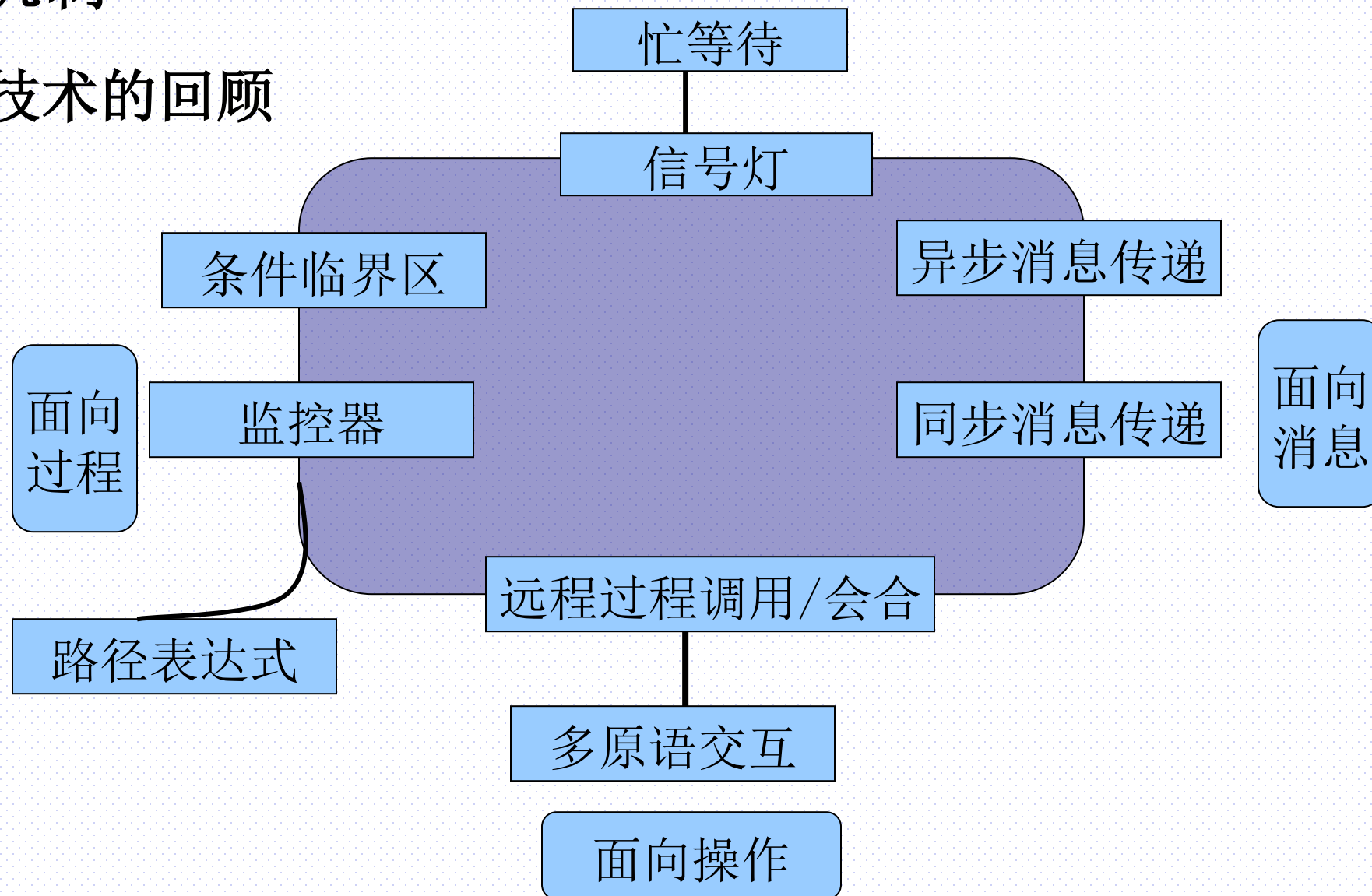


Sender 等待Task\_example



# 多原语的并发机制

- 进程交互主要技术的回顾



## 主要内容:

- 并发程序设计的基本概念
- 并发程序带来的问题
- 需要解决的基本问题
- 程序语言示例

## 并发语言需要解决的基本问题

- 开发并行程序设计语言一般有三种方法:

①设计一种新语言。新语言有比较强的并行性描述能力。但是兼容性不好，不易推广，因此现在采用该方法的系统较少。

②对现有的顺序语言加以扩展，提供并行性描述机制。该方法具有兼容性好、编程简便等特点，现在常常被采用。

③不改变现有顺序语言，而由用户提供的函数库、类库或并行化编译系统等方法实现并行程序设计。该方法简单灵活，易于推广。

# 并发程序设计语言

高级程序设计语言扩充并发机制，最早可溯源至PL/1和Algol-68。20多年以来，随着各种并发机制的研究推出了数十种具有并发机制的高级程序设计语言。

直到目前，还没有一种并发程序设计语言能统治并发程序设计整个领域。其原因是硬件背景、应用领域和程序设计模型的多样性。

## 主要的并发程序设计语言

语言	年代	并发机制	备注
DP	1978	CCR+RPC	
Edison	1978	CCR	
Argus	1982	CCR+RPC+原子事务	
Lynx	1991	RPC+会合+CCR	
Concurrent Euclid		监控器(SW)	
Concurrent Pascal	1975	监控器(SW)	
Modula-3	1985	监控器(SC)+协例程和锁的包	
Path Pascal	1979	监控器+路径表达式	
Pascal Plus	1979	监控器(SU)	
Turing Plus	1983	监控器(SC+SW)	
Mesa	1979	监控器(SC)+RPC	
Emerald		监控器+RPC(面向对象)	



语言	年代	并发机制	备注
Actor		异步消息传递(基于对象)	
PLITS	1979	异步消息传递	
NIL	1982	异步消息传递+会合	
Gypsy	1979	异步消息传递	
CONIC		异步消息传递	
CSP	1980	同步消息传递	
Joyce		同步消息传递	
Occam	1987	同步消息传递	
Concurrent C	1985	会合+异步发送	
Concurrent C++	1987	会合	
Ada	1983	会合	
SR	1982	多共享+消息原语	92年SR2.0
Star Mod	1980	多消息原语	
Linda	1986	带共享元组空间的消息原语	
Java	1996	类库支持多原语	

# Ada的任务

- Ada的任务结构

```
task TNAME is
    entry ENAME (FP_list);           //可以多个
end TNAME;
task body TNAME is
    局部声明;
begin
    accept ENAME (FP_list) is       //对应多个
        语句序列;
    end ENAME;
end TNAME;
```

任务激活后就是一个进程。其它进程通过

```
call TNAME.ENAME (AP_list);
```

## •Ada的通信与同步

- (1) 简单选择
- (2) 否则选择
- (3) 卫式选择
- (4) 延时选择

# SR语言

- 设计于70年代末，从多消息传递原语扩充了共享内存而来。
- 程序结构（有**resources** (资源)和**globals**(全局共享)两种模块）：

**resource** name

输入规格说明;

输出操作和类型声明;

**body** name(FP\_list)

变量和局部声明;

初始化码;

过程;

进程;

终止化代码;

**end** name.

**global** name

输入规格说明;

输出操作和类型声明;

**body** name

变量和局部声明;

初始化码;

过程;

终止化代码;

**end** name

## SR语言

- 在SR中，同一资源内的进程可以共享变量，可以使用各种原语实现通信和同步：信号灯、异步消息传递、RPC、会合。
- SR既适合分布式也适于共享内存的多处理器机器。
- 调用操作用**call**则为同步，**send**为异步。

## 基于消息的并行编程模型

- 由于消息传递并行编程模型的广泛应用，目前已经出现了许多基于该模型的并行编程语言，其中最流行的是PVM（Parallel Virtual Machine）和MPI（Message Process Interface）。
- 第一个将工作站集群作为并行计算平台并被广泛接受的并行编程语言是PVM。它由美国的Oak Ridge国家实验室、Tennessee大学、Emorg大学、CMU大学等联合开发而成，能够将异构的Unix计算机通过异构网络连接成一个“虚拟”的并行计算系统，为其上运行的应用程序提供分布式并行计算环境。

# PVM

- PVM是一种基于局域网的并行计算环境。它通过将多个异构的计算机有机地组织起来，形成一个易于管理、易于编程、并且具有很好的可扩展性的并行计算环境。目前PVM支持C和 Fortran语言。
- 程序员首先参照消息传递模型编写好并行程序，然后将编译后的程序以任务为单位在网络中特定类型的计算机上运行。
- PVM能够在虚拟机中自动加载任务并运行，并且还提供了任务间相互通信和同步的手段。由于所有的计算任务都被分配到合适的计算节点上，多个节点并行运算，从而实现了任务一级的并行。
- PVM为利用现有计算资源进行并行程序的开发与研究提供了一种有效的解决方案。由于PVM免费、开放和易于使用的特性，使得它成为一个被广泛接受的并行程序开发环境，目前几乎所有的并行计算系统都支持PVM。

# PVM

- PVM--Parallel Virtual Machine

PVM3的原代码可以在

<http://www.emp.ornl.gov/pvm>

或<ftp://netlib2.cs.utk.edu/pvm>下找到

- XPVM是PVM的图形化控制台，用Tcl/Tk写成。可以在  
<http://www.netlib.org/utk/icl/xpvm/xpvm.html>下找到



# MPI

- MPI是为开发基于消息传递模型的并行程序而制定的工业标准，其目的是为了**提高并行程序的可移植性和易用性**。
- 参与MPI标准制定的人员来自欧美40多个组织，大部分主要的并行计算机制造商、大学研究所、政府实验室、工业组织等都投入到MPI标准化工作。
- 有了统一的并行编程语言标准，并行计算环境下的应用软件及软件工具就都能够实现透明的移植，各个厂商就可以依据标准提供独具特色和优势的软件实现和软件支持，从而提高了并行处理的能力。

# MPI

- MPI是一种基于消息传递模型的并行编程接口，目前已经发展成为消息传递模型的代表和事实上的工业标准，而不是一门具体的语言。迄今为止，所有的并行计算机制造商都提供对MPI的支持，因而从理论上说任何一个正确的MPI程序可以不加修改地在所有并行计算机上运行。
- MPI只是一个并行编程语言标准，要编写基于MPI的并程序，还必须借助某一MPI具体实现。
- MPICH是Linux平台下最重要的一种MPI实现，是一个与MPI规范同步发展的版本。每当MPI标准推出新的版本时，MPICH就会有相应的实现版本。
- LAM（Local Area Multicomputer）是Linux平台下另一免费的MPI实现。它由Ohio州立大学开发，主要用于异构的网格计算并行系统。

# MPI

- MPI--Message Passing Interface:

<http://www-c.mcs.anl.gov/mpi>

- 支持Xwindow的MPICH: <http://www.mcs.anl.gov/mpi/mpich>

- XMPI: XMPI - A Run/Debug GUI for MPI

<http://www.osc.edu/lam/lam/xmpi.html>

## 00800403

```
/*=====*
 * mpi_hello.c - demo program of mpich.*
 *=====*/

#include
#include "mpi.h"

int main(int argc, char **argv)
{
    int myrank, nprocs, namelen;
    char processor_name[mpi_max_processor_name];

    mpi_init(&argc, &argv);
    mpi_comm_size(mpi_comm_world, &nprocs);
    mpi_comm_rank(mpi_comm_world, &myrank);
    mpi_get_processor_name(processor_name, &namelen);

    printf("hello world! i'm rank %d of %d on %s\n", myrank, nprocs, processor_name);

    mpi_finalize();
    return 0;
}
```

# 并行程序示例

Master.c

```
#include "pvm3.h"
#define SLAVE1 "slave1"
#define SLAVE2 "slave2"
main()
{
    int mytid,tids1[2],tids2;...
    pvm_spawn(SLAVE1,char(**),0,0,"",nproc1,tids1);
    pvm_spawn(SLAVE2,char(**),0,0,"",nproc2,&tids2);...
    pvm_initsend(PvmDataRaw);
    pvm_pkint(n,1,1);
    pvm_mcast(tids1,nporc1,0);...
    pvm_send(tids2.0);
    pvm_rcv(-1,msgtype);...
    pvm_exit();
}
```

slave1.c, slave2.c

- Ada83支持同步消息传递，Ada95支持异步消息传递。
- Ada中通过任务/task来运行（并发执行的程序单位），消息传递是这种设计的基础。
- 任务的形式：说明部分和体部分。
- 任务的接口是它的入口点，可以接收来自其它任务的消息。

```
task Task_Example is
  entry Entry_1 (Item : in Integer)
end Task_Example
```

```
accept 入口_名称(形参) do
  .....
end 入口_名称;
```

# 建立通用任务库

- 施动者任务(actor task): 没有入口点的任务。
- 服务者任务(server task): 可以有accept子句, 但是在accept子句的外面没有其它代码, 仅仅用于对其它任务做出反映。
- 发送者任务必须直到接收者任务的入口点名称, 一个任务的入口点并不需要将发送消息给它的任务名称。

# 创建任务

- Ada中的任务具有类型，可以匿名或命名。
- 命名类型的Ada任务可以使用new操作动态创建。并且可以通过使用指针引用该任务。
- 将任务声明于包等的声明部分。

```
task type Buffer is
```

```
    entry Deposit (Value : in Integer);
```

```
    entry Fetch (Value: out Integer);
```

```
end;
```

```
type Buf_ptr is access Buffer;
```

```
.....
```

```
Buf: Buf_Ptr
```

```
Buf:= new Buffer;
```



# 任务的多入口

- 任务可以具有任意多个入口。
- 任务中相关的accept子句的出现顺序决定接收消息顺序。
- 通过select语句形成随机接收。

**task body Teller is**

**loop**

**select**

**accept** Drive\_Up(formal parameters) do

.....

end Drive\_Up

.....

or

**select**

.....

end select

**end loop;**

**end Teller;**

# ADA的合作同步

- 让收发任务在accept上同步

- 通过卫式 (guard) 延迟会合

- ```
When not Full(Buffer) =>
```

- ```
    accept Deposit(New_Value) do
```

- 通过卫式对accept开放与关闭

- 通过loop、select和卫式构造复杂的会合

- 正常情况：一个开放accept对应一条消息

- 多个开放accept对应较少的消息

- 应对accept都关闭的异常：增加else子句(是否可以有accept? )

- terminate子句

# ADA的竞争同步

- 使用任务控制存取一个数据结构，需要在任务中声明这个数据结构，即可互斥访问。
- 任务的执行语义：任意时刻只能有一条活跃的accept子句，由此保证对于受保护数据结构的互斥访问。
- 例外：被嵌套的任务可以访问上层任务的共享数据结构。一种破坏数据封装的行为。
- 例子：具有buffer的生产者消费者，在buffer处于空和满之间，生产者和消费者都可以操作buffer，不使用信号量将产生错误。

# 其它的控制机制

- 优先级: pragma Priority(表达式);
  - 通过表达式说明任务的相对优先级
- 二元信号量

```
task Binary_Semaphore
is
    entry Wait;
    entry Release;
end Binary_Semaphore;
```

```
task body Binary_Semaphore
is
    begin
    loop
        accept Wait;
        accept Release;
    end loop;
end Binary_Semaphore ;
```

# 其它的控制机制

## ■ 受保护的對象

- 使用任务入口隐式提供竞争同步的问题：很难实现高效率的会合机制。
- 将任务受保护改为受保护的對象，类似于管程。
  - 形式上受保护对象的入口与任务入口相似（entry）
  - 受保护对象依靠受保护的过程、函数和语句组来访问
  - 受保护的过程：给受保护对象的数据提供互斥的读写访问。
  - 受保护的函数提供并发的只读访问
  - 多任务使用受保护对象时，在受保护对象入口处提供同步通讯。

## 其它的控制机制

### ■ 异步消息传递

- 前述会合机制为严格同步，会合前需要消息发送者和接收者都必须就绪。
- 特殊select子句：异步select子句，具有触发两种机制，入口调用或delay子句（二选一）
- 异步select子句中包括可中止部分（abort子句）。

loop	select
select	delay 10.0
.....	.....
then abort	then abort
.....	.....
end select	end select
end loop	

## ADA并发机制小结

- 在不具有独立存储器的分布式处理器的情况下，管程和消息传递都用于实现共享数据的访问。
- Ada的使用受保护对象支持共享数据的访问代码更简单，更高效。
- 对于分布式系统，消息传递更适合。

# JAVA对并发的支持

- 以synchronized控制模块的同步
  - Java使用监控器 (Monitor) 作为线程同步的机制。每一个对象都有一个监控器与之相对应。可以用synchronized关键字请求得到某个对象的监控器。
  - 在任何时刻，只能有一个线程可以拥有某个对象的监控器。
  - 如果有多个线程申请某个对象的监控器，系统会阻塞这些线程。
  - 如果监控器的使用者释放了监控器的拥有权，系统就会唤醒某个申请该监控器的阻塞线程。
- 支持多线程编程

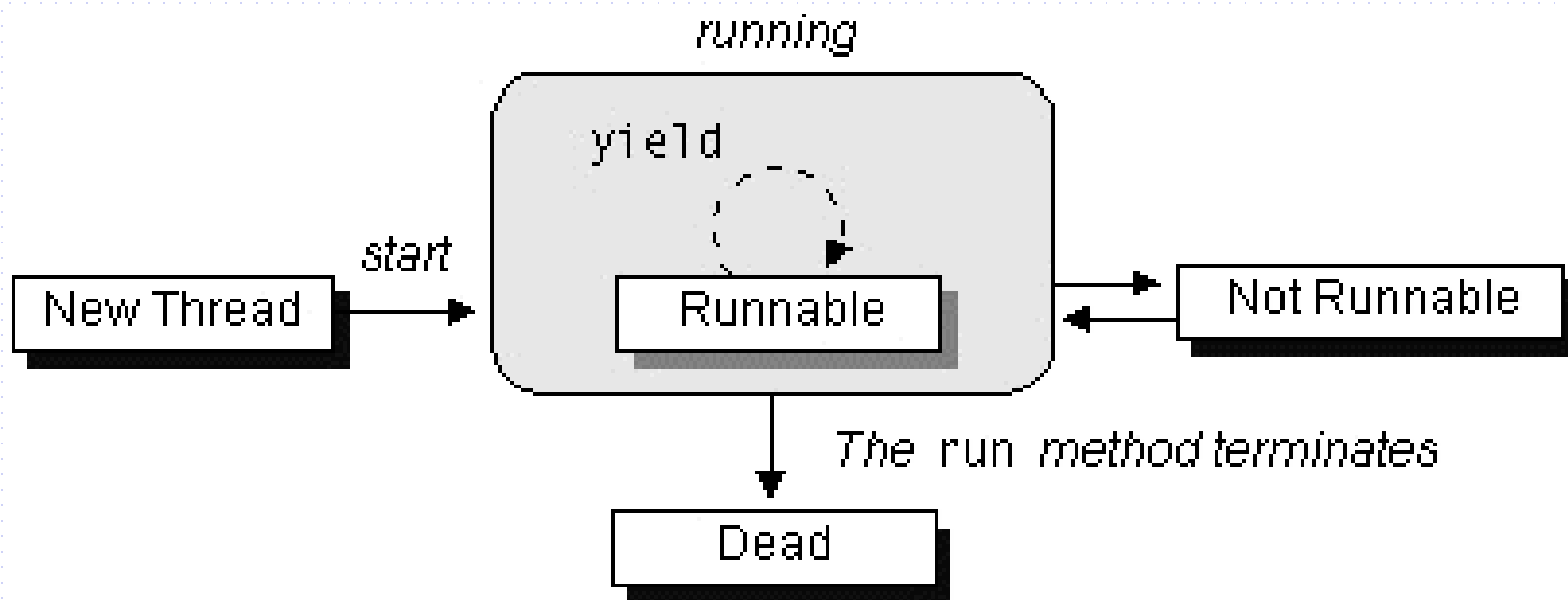


# ■ 支持多线程编程

## ■ 线程的状态

- 初始态：线程对象被创建但没有被运行时的状态；
- 运行态：线程对象被启动，进入CPU的调度队列，开始执行另外的运行线索；
- 阻塞态：线程对象由于某种原因，不能继续执行，CPU不再为它分配时间片。但是，这时的线程仍然是有生命的，一旦条件满足，它能够再次进入运行态，继续执行剩余的任务；
- 死亡态：线程执行完任务，或者被强行结束。进入死亡态后，该线程不能再次返回运行态。

## JAVA线程状态的转化:



Java的线程也是对象，由`java.lang.Thread`类继承而来。`Thread`类中有`run`方法，子类线程覆盖这个方法，将需要执行的语句填入`run`方法中。

这样，当该线程被启动时，它会自动执行`run`方法中的语句。

- 线程原语。java.lang.Thread类中有如下方法：

start / stop：启动和停止线程

suspend / resume：悬挂和恢复线程

sleep：使线程睡眠

join：延迟一个方法的执行，直到另一线程的run方法执行完毕

interrupt：唤醒被阻塞的或是睡眠的线程

yield：让出CPU时间，使其它线程执行

## ■ wait / notify （等待-唤醒） 原语

wait和notify方法是所有Java对象父类：`java.lang.Object`类的方法。当一个线程拥有一个对象的监控器（通过synchronized关键字）时，它可以调用该对象的wait或notify方法。

- 调用wait方法，该线程将放弃该对象的监控器，并且被阻塞，直到其它线程调用了该对象的notify方法唤醒它为止。
- 调用notify方法，该线程将唤醒某个由于等待（调用了wait方法）而被阻塞的线程，将该对象的监控器交给它，使其恢复执行。
- 通过synchronized实现竞争同步
- 通过wait/notify实现合作同步。

- 以下是利用Java实现的生产者-消费者问题:

```
import java.util.*;

public class CP{

    Vector pool; //Vector是可以存储任何对象的表.这里当作产品池.
    int product = 0;
    public static int EMPTY = 0;
    public static int FULL = 25;
}

public static void main(String[] args){
    CP cp = new CP();
    Consumer consumer = new Consumer(cp);
    Producer producer = new Producer(cp);
    consumer.start();//启动线程,使其运行它的run函数
    producer.start();
}
```

```
public CP() {
    pool = new Vector();
} //CP类的构造函数.
public synchronized void produce() {
    try{
        if(pool.size() == FULL)
            pool.wait();
        product++; //produce a product.
        pool.addElement(new Integer(product));
        //put the product into the pool.
        System.out.println("Produce: " + product);
        if(pool.size() == EMPTY + 1)
            pool.notify();
    } catch (InterruptedException e) {}
}
```

```
public synchronized void consume() {
    try{
        int i = pool.size();
        if(i == EMPTY)
            pool.wait();
        System.out.println("Consume: " +
            pool.firstElement().toString());
        pool.removeElementAt(0);
        if(pool.size() == FULL - 1)
            pool.notify();
    } catch (InterruptedException e) {}
}

} //End of class CP.
```

CP类以监控器的方式来管理产品池。它封装了数据对象——产品池，和操纵该对象的方法——生产(produce)和消费(consume)方法，并且通过Java的同步机制对这些方法的并发性进行控制。

以下是消费者类—Consumer

```
class Consumer extends Thread{
    CP cp;
    public Consumer(CP cp) {
        super();
        this.cp = cp;
    }
    public void run() { //线程的执行函数
        while(true) {
            cp.consume(); //调用监控器的consume方法
            Thread.sleep(1000); //模拟做其它工作的时间
        }
    }
} // End of class Consumer
```



以下是生产者类—Producer

```
class Producer extends Thread{
    CP cp;
    public Producer(CP cp){
        super();
        this.cp = cp;
    } // 构造函数
    public void run(){
        while(true){
            cp.produce(); //调用监控器的produce方法
            Thread.sleep(200); //模拟做其它工作的时间
        }
    }
}
```

- 线程通信有以下几种方式：
  - 共享对象
  - 管道流
  - 异地线程之间可以通过socket或RMI进行通信

以下是一个利用管道流（PipedStream）进行线程之间通信的例子。

```
import java.io.*;

public class CommThreads{
    public static void main(String[] args){
        // 生成一个单向管道流
        PipedInputStream in = new PipedInputStream();
        PipedOutputStream out = new PipedOutputStream(in);

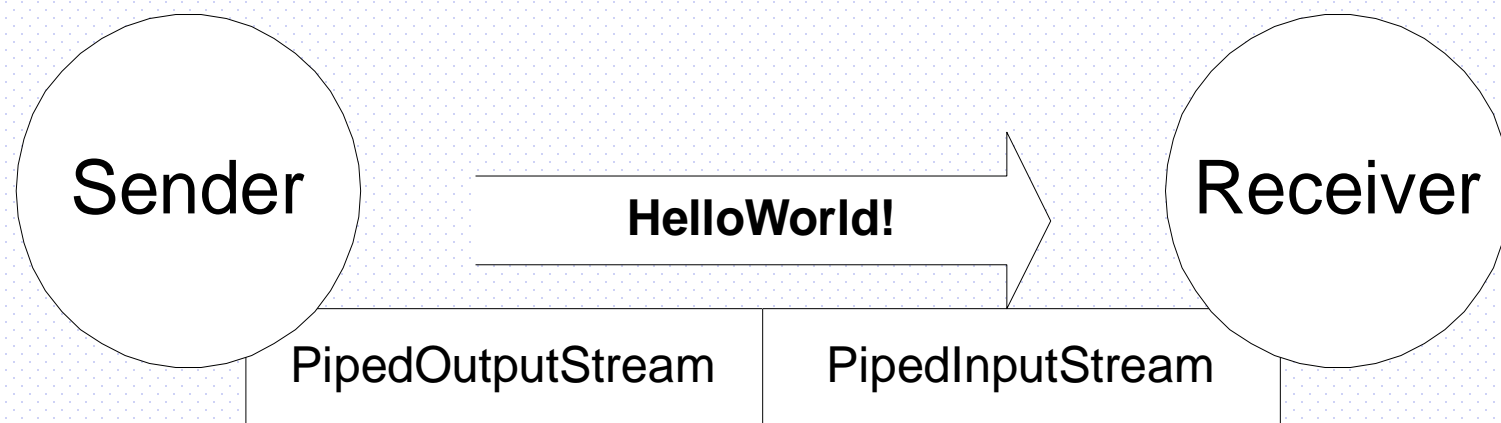
        Thread sender = new Sender(out);
        Thread receiver = new Receiver(in);
        sender.start(); // 启动发送者线程
        receiver.start(); // 启动接收者线程
    }
}
```

```
class Receiver extends Thread{
    PipedInputStream in;
    public Receiver(PipedInputStream in){
        super();
        this.in = in;// 得到管道流对象
    }
    public void run(){
        StringBuffer sBuffer = new StringBuffer();
        int b = 0;
        while(b != -1){//如果b==-1，表明已经读完
            b = in.read();//从管道中读取信息
            if(b != -1) sBuffer.append((char)b);
        }
        System.out.println("Receiver: " +
            sBuffer.toString());// 输出得到的消息
    }
}
```

```
class Sender extends Thread{
    PipedOutputStream out;
    public Sender(PipedOutputStream out) {
        super();
        this.out = out; // 得到管道流对象
    }
    public void run() {
        String sMessage = "HelloWorld!";
        // 发送的消息是 "HelloWorld!"。
        byte[] bMessage = sMessage.getBytes();
        // 将消息转化成字节流
        out.write(bMessage, 0, bMessage.length);
        out.flush(); // 发送消息
    }
}
```

在这个例子中，通过 java.io 包提供的 PipedInputStream 和 PipedOutputStream 为两个线程创建了一个单向管道。

Sender和Receiver线程可以通过这个管道传递消息。如果管道流建立在 socket 之上，线程之间就可以远程通信。



## Go语言特点

1. Go是类c的语言。编译型语言。
2. channel（通道）；
  - 支持通信顺序进程（Communicating Sequential Process）
  - 在不同的执行体之间传递值
3. 基于共享内存的多线程模型；
  - 类似其它语言中线程的使用

## I. goroutine

- 在go里，每一个并发执行的活动称为goroutine
- 程序启动时，只有一个goroutine来调用main函数，其被称为主goroutine。

- 创建：在普通的函数或者方法调用前加上go关键字前缀

```
f()      // 调用 f(), 等待它返回  
go f()   // 新建一个调用 f() 的goroutine, 立刻返回
```

- 终止：main函数返回时，所有正在运行的goroutine都直接中止。否则，只能通过与一个goroutine通信来让它自己停止



## 2. channel;

- channel是让一个goroutine发送特定值到另一个goroutine的通信机制。通道是并发执行体之间的连接。
- 每个通道具有一个特定的类型，称为元素类型，一个通道只能发送相应类型的值。
- int类型元素的通道写为chan int
- 通道的创建，通过内置的make函数来完成：

```
ch := make(chan int) // ch 的类型是'chan int'
```

- 通道是一个使用make创建的数据结构的引用。

## 2. channel;

- 通道复制或作为参数传递时，复制的是引用，因此调用者和被调用者引用同一个通道。
- 使用“==”比较，当二者是同一通道的引用时，返回true。
- 通道的两个主要操作：发送(send)和接收(receive)
- send语句从一个goroutine传输一个值到另一个在同一通道上执行receive操作的goroutine，两个操作都使用“<-”操作符。

```
ch <- x    // 将 x 发送到通道 ch
x = <- ch  // 使用 x 接收通道 ch 传递过来的值
<- ch     // 接收 ch 传递过来的值后，丢弃结果
```

## 2. channel;

- 通道的第三个操作：关闭(close)。

- 在关闭后的通道上执行发送操作，将导致宕机
- 在关闭后的通道上执行接收操作，当通道中有值时，将立即获取到其中的所有值；当通道为空时，会获取到通道元素类型对应的零值

```
close(ch) // 使用内置的close函数来关闭通道
```

- 无缓冲通道和缓冲通道

- 使用简单的make创建无缓冲通道
- make可以接收第二个可选参数，一个代表通道容量的整数。

```
ch = make(chan int) // 无缓冲通道  
ch = make(chan int, 0) // 无缓冲通道  
ch = make(chan int, 3) // 容量为 3 的缓冲通道
```

## 2. channel;

- 无缓冲通道

- 无缓冲通道上的发送（接收）操作将导致阻塞，直到另一个goroutine在该通道上执行接收操作（发送一个值）
- 无缓冲通道可以用来进行goroutine的强同步
- 消息的种类：
  - 带值的消息
  - 不携带值，强调同步效果

## 2. channel;

- 缓冲通道

```
ch = make(chan int, 3) // 容量为 3 的缓冲通道
```

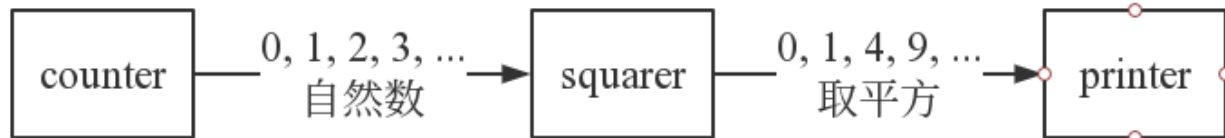
- 缓冲通道有一个元素队列，这个队列的最大长度即通道的容量
- 在缓冲通道上的发送操作将在队列的尾部添加一个元素，如果通道已满，则阻塞操作直到另一个goroutine对该通道执行接收操作
- 在缓冲通道上的接收操作将在队列的头部取出一个元素，如果通道为空，则阻塞操作直到另一个goroutine对该通道执行发送操作
- 缓冲通道的缓冲区将通道的发送和接收goroutine解耦

## 2. channel; 管道和单向通道

- 管道
  - 通道可以用来连接goroutine，这样一个goroutine的输入是另一个goroutine的输出，这称为管道。
- 单向通道
  - 因为将一个通道传递给函数时，它通常会限制为不能发送或不能接收。
  - go的类型系统提供了单向通道类型，仅仅导出发送或接收操作。
  - 类型‘chan<- int’是一个只能发送的通道，而类型‘-<chan int’是一个只能接收的通道

```
func makeThumbnails1( filenames []string) {
    for _, f := range filenames {
        go thumbnail.ImageFile(f)
    }
}
```

```
func makeThumbnails2( filenames []string) {
    ch := make(chan struct{})
    for _, f := range filenames {
        go func(f string) {
            thumbnail.ImageFile(f)
            ch <- struct{}{}
        }(f)
    }
    for range filenames {
        <-ch
    }
}
```



## 三个goroutine:

第一个goroutine叫做counter，产生一个自然数序列，发送到下一个

goroutine;

第二个goroutine叫做squarer，计算每个数的平方，发送到下一个

goroutine;

第三个goroutine叫做printer，接收值并输出它们。

```
func counter(out chan<- int) {
    for x := 0; x < 100; x++ {
        out <- x
    }
    close(out)
}

func squarer(out chan<- int, in <-chan int) {
    for v := range in {
        out <- v * v
    }
    close(out)
}

func printer(in <-chan int) {
    for v := range in {
        fmt.Println(v)
    }
}

func main() {
    naturals := make(chan int)
    squares := make(chan int)

    go counter(naturals)
    go squarer(squares, naturals)
    printer(squares)
}
```



- 假设生成一个自然数需要200毫秒的时间，对自然数取平方需要500毫秒的时间，而输出一个自然数需要300毫秒的时间，请修改上一页的程序，让输出所有数字所需时间最小。
- 提示一：可以用`time.Sleep(200 * time.Millisecond)`来模拟时间消耗了200毫秒；
- 要求一：在goroutine的阻塞时间最小同时，请尽量减少内存花费；
- 要求二：请说明你的程序输出每一个数字所需的时间是多少。

### 3. select语句；

- 类似switch语句，select语句具有一个case列表和一个可选的default分支。
- 每一个case指定一次通信（在某个通道上的发送或接收操作）和一个关联的代码块。
- select一直等待，直到一次通信来告知某个操作可以执行
- 当多个情况同时满足时，select随机选择一个，这样保证每一个通道有相同的机会被选择

```
select {  
  case <- ch1:  
    // ...  
  case x := <-ch2:  
    // ... use `x` ...  
    x = x * x  
  case ch3 <- y:  
    // ...  
  default:  
    // ...  
}
```

## 1. 互斥锁: `sync.Mutex`

- 一个互斥锁只能同时被一个 goroutine 锁定, 其它 goroutine 将阻塞直到互斥锁被解锁, 之后再重新争抢对互斥锁的锁定。

## 2. 读写互斥锁: `sync.RWMutex`

- 读写锁是针对读写操作的互斥锁, 读写锁与互斥锁最大的不同就是可以分别对 读、写进行锁定。一般用在大量读操作、少量写操作的情况:
  - 同时只能有一个 goroutine 能够获得写锁定。
  - 同时可以有任意多个 goroutine 获得读锁定。
  - 同时只能存在写锁定或读锁定 (读和写互斥)。

# 同步原语

## 1. 组等待：sync.Mutex

- 用于等待一组 goroutine 结束。
- 主goroutine调用Add来设置等待的goroutine的数量。然后每一个goroutines运行并在完成时调用Done。同时，可以使用Wait来阻塞，直到所有的goroutines完成。

## 2. 延迟初始化：sync.Once

- 让一个操作只执行一次。

1. goroutine是用户级的线程。
2. OS线程（操作系统线程）一般都有固定的栈内存（通常为2MB），一个goroutine的栈在其生命周期开始时只有很小的栈（典型情况下2KB），goroutine的栈不是固定的，他可以按需增大和缩小，goroutine的栈大小限制可以达到1GB。所以在Go语言中一次创建十万左右的goroutine也是可以的。
3. OS线程是由OS内核来调度的，goroutine则是由Go运行时（runtime）调度，这个调度器使用一个称为m:n调度的技术（复用/调度m个goroutine到n个OS线程）。goroutine的调度不需要切换内核语境，所以调用一个goroutine比调度一个线程成本低很多。
4. Go语言中可以通过runtime.GOMAXPROCS()函数设置当前程序并发时占用的CPU逻辑核心数。Go1.5版本之前，默认使用的是单核心执行。Go1.5版本之后，默认使用全部的CPU逻辑核心数。

## 总结

- 并行语言需要支持的两大内容：进程控制，通讯。
- 并行语言对不同并行程序设计模型的支持主要体现在对通讯方式的支持上。
- 并行语言现状：没有广泛使用的独立并行语言，基本都属于在传统语言如C，C++，Java上增加对进程/线程，共享/通讯的支持（两种扩展方式：语法直接扩展；并行库）。

## 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，多核系统的广泛出现是我们对这方面的深入理解变得更为急迫
- 这方面已经有了许多理论和技术成果，但还很不成熟。
- 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能方便程序员使用的并发性编程的高级抽象
- 在写并发程序时，需要去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，例如死锁等