操作系统的资源分配

1 进程与线程

简单的共享资源实现

在单一处理器的前提下，为了实现与多处理器类似的功能，我们需要进行时间多路复用

- 所有虚拟 CPU 共享相同的非 CPU 资源，如：I/O设备、相同的内存；
- 共享的结果相同：每个线程可以访问每个其他线程的数据（有利于共享，不利于保护）、线程可以共享指令（有利于共享，不利于保护）
- 这种（不受保护的）模型常见于：嵌入式应用程序的开发、同时具有产量和定时器的开关

进程 Process (Task)

- 进程是**执行中的程序**
- 进程是一个**程序及其数据**在处理机上顺序执行时所**发生的活动**
- 进程是程序在一个数据集合上运行的过程，它是**系统进行资源分配和调度的一个独立单位**

进程的基本属性

资源的拥有者：给每个进程分配一虚拟地址空间，保存进程映像；控制一些资源（文件，I/O设备）；有状态、优先级、调度等。

调度单位：进程是一个执行轨迹。

- 线程的引入：首先，不同应用中的并发任务要共享一个公共的地址空间和其他资源，因此只能将这些任务串行化，效率很低。其次，进程的创建、撤消和切换需要需要很大的开销，限制了并发度的提高。
- 线程的定义：线程是进程内一个相对独立的、可调度的执行单元。它是进程中的一个实体，有时也称轻量级进程。但资源的拥有者还是它的进程。线程将原来进程的不同属性分开处理。

进程的特征

- 进程之间具有**并发性**：在一个系统中，同时会存在多个进程。于是与它们对应的多个程序同时在系统中运行，轮流占用CPU和各种资源。
- 进程间会**相互制约**：由于进程是系统中资源分配和运行调度的单位，因此在对资源共享和竞争中，必然会相互制约，影响了各自向前推进的速度。

进程状态

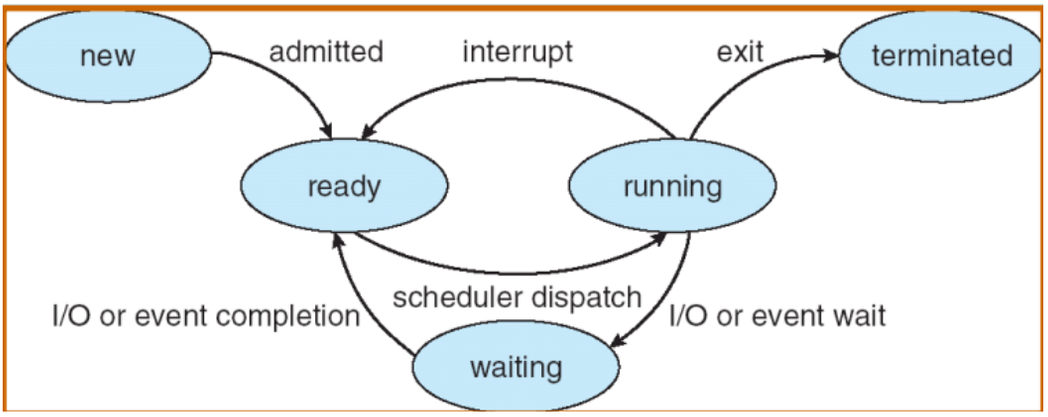
新的 （New）：进程正在创建

就绪 （Ready）：当进程已分配到除 CPU 以外的所有必要资源后，只要再获得 CPU，便可立即执行。（万事俱备，只欠 CPU）

执行 （Running）：进程已获得 CPU，其程序正在执行。（获得 CPU）

阻塞/等待 （Waiting）：正在执行的进程由于发生某事件而暂时无法继续执行时，便放弃处理机而处于暂停状态，把这种暂停状态称为阻塞状态，有时也称为等待状态。（无法继续执行，放弃处理机处于暂停状态）

终止 （Terminated）：进程已经完成执行



程序与进程之间的区别

- “ 进程 ” 是一个**动态的概念**：进程强调的是程序的一次 “ 执行 ” 过程，程序是一组有序指令的集合，在多道程序设计环境下，它不涉及 “ 执行 ”，因此,是一个静态的概念；
- 不同的进程可以**执行同一个程序**：即使多个进程执行同一个程序，只要它们运行在不同的数据集合上，它们就是不同的进程；
- 每一个进程都有自己的**生命期**：当系统要完成某一项工作时，它就 “ 创建 ” 一个进程，程序执行完毕，系统就 “ 撤销 ” 这个进程，收回它所占用的资源。

线程 Thread

有时也被称为轻量级进程 （Lightweight process, LWP）

- 线程是 CPU 使用的基本单元，是进程内的控制流
- 状态参数：在 OS 中的每一个线程都可以利用线程标识符和一组状态参数进行描述。这些参数包括：
    - 寄存器状态，它包括程序计数器PC和堆栈指针中的内容
    - 堆栈， 在堆栈中通常保存有局部变量和返回地址
    - 线程运行状态， 用于描述线程正处于何种运行状态
    - 优先级， 描述线程执行的优先程度
    - 线程专有存储器， 用于保存线程自己的局部变量拷贝
    - 信号屏蔽， 即对某些信号加以屏蔽
- 同一进程内的所有线程共享代码段、数据段和其它操作系统资源 （如文件、信号等）

> 传统进程或重量级进程等价于具有单一线程的作业。在多线程作业中，当一个服务器线程被阻塞并等待时，同一作业中的第二个线程可以继续运行。线程提供了一种机制，允许顺序进程进行阻塞系统调用，同时也实现了并行性。在同一作业中的多个线程的协作可以提供更高的吞吐量和改进的性能。部分需要共享公共缓冲区（即产品消费者）的应用程序也因此受益。

如同传统的进程一样，在各线程之间也存在着共享资源和相互合作的制约关系，致使线程在运行时也具有间断性。 相应地，线程在运行时，也具有下述三种基本状态：

① 执行状态，表示线程正获得处理机而运行；

② 就绪状态，指线程已具备执行条件，一旦获得CPU便可执行的状态；

③ 阻塞状态，指线程在执行中因某事件而受阻，处于暂停执行时的状态。

线程的基本属性

- 线程是进程内的一个相对独立的可执行单元。
- 线程是操作系统中的基本调度单元，它含有调度所需的必要信息。
- 每个进程在创建时，至少需要同时为该进程创建一个线程。也就是说进程中至少要有一个或一个以上线程，否则该进程无法被调度执行。
- 线程可以创建其他线程。
- 进程是资源分配的基本单元，同一进程内的多个线程共享该进程的资源。但线程并不拥有资源，只是使用它们。
- 由于共享资源(包括数据和文件)，所以，线程间需要通信和同步机制。
- 线程有生命期，在生命期中有状态的变化

内核线程与用户线程

- **内核线程（KLT）**，由操作系统直接支持，内核负责在内核空间执行线程创建、调度和管理 (如 Mach 和 OS/2)

  特点

  （1）所有线程管理由内核完成

  （2）没有线程库，但对内核线程工具提供 API

  （3）内核维护进程和线程的上下文

  （4）线程之间的切换需要内核支持

  （5）以线程为基础进行调度

  （6）创建和管理比用户线程慢

  内核线程优缺点

  （1）对多处理器，内核可以同时调度同一进程的多个线程，更好的支持多 CPU 体系结构

  （2）线程之间的阻塞相互独立（既一个线程执行阻塞系统调用时，内核可以调度进程里的其它线程执行）

  （3）内核例程是多线程的

  （4）在同一进程内的线程切换会调用内核，导致速度下降

- **用户线程（ULT）**，位于内核之上，无需内核支持，通过一组用户级的库调用 (例如来自 CMU 的 Andrew 项目)

  特点

  （1）由应用程序通过线程库完成所有线程的管理。线程库包括线程的创建、撤消、消息和数据传递、调度执行以及上下文保护和恢复

  （2）内核不知道线程的存在，但仍然管理线程的活动

  （3）线程切换不需要核心态特权

  （4）调度是应用程序特定的

  （5）当线程调用系统调用时，整个进程阻塞

  用户线程优缺点

  （1）线程切换不调用内核

  （2）调度是应用程序特定的，可以选择最好的算法

  （3）ULT 可运行在任何操作系统上（只需要线程库）

  （4）大多数系统调用是阻塞的，因此当内核阻塞进程时，进程中所有线程将被阻塞

（5）内核只将处理器分配给进程，所以同一进程中的两个线程不能同时运行于两个处理器上。

此外，部分混合方法同时实现了用户线程和内核线程 (如 Solaris2)

线程的优点

响应度高：如果对一个交互式应用程序采用多线程，即使其部分阻塞或执行冗长的操作，那么该程序仍能够继续执行，从而增加了对用户的响应度

资源共享： 线程默认共享它们所属进程的内存和资源

经济：创建和切换的代价小

多处理器体系结构的利用：无论系统中有多少个 CPU，单线程的进程只能运行在一个 CPU 上

二者的比较

1. **调度**
   在引入线程的 OS 中，线程是调度和分派的基本单位。当线程切换时，仅需保存和设置少量寄存器内容，切换代价远低于进程。
2. **并发性**
   在引入线程的 OS 中，不仅进程之间可以并发执行，而且在一个进程中的多个线程之间亦可并发执行，甚至还允许在一个进程中的所有线程都能并发执行。同样，不同进程中的线程也能并发执行。
3. **拥有资源**
   线程本身并不拥有系统资源，而是仅有一点必不可少的、能保证独立运行的资源。
   属于同一进程的所有线程都具有相同的地址空间。
4. **独立性**
   在同一进程中的不同线程之间的独立性要比不同进程之间的独立性低得多。
5. **系统开销**
   在一些 OS 中，线程的切换、同步和通信都无需操作系统内核的干预。
6. **支持多处理机系统**
   多线程进程可以将一个进程中的多个线程分配到多个处理机上，使它们并行执行，这无疑将加速进程的完成。

Q: 线程可以覆盖操作系统功能吗?

2 多线程模型与问题

多对一模型

- 多个用户线程映射到一个内核线程
- 线程的管理在用户空间内完成，效率比较高，通常用于不支持多内核线程的系统上
- **缺点**

  一个线程阻塞将导致整个进程阻塞

  多个线程不能并行运行在多处理器上
- 例如

Solaris 2 系统的 Green threads

Unix 的 Pthreads

一对一模型

- 每个用户线程都映射到一个内核线程上

  提供了更好的并发性（某个线程的阻塞不一定导致进程的阻塞、支持多处理器

- **缺点**

  创建用户线程造成的高开销

- 例如

  Windows NT

  OS/2

多对多模型

- 多路复用了许多用户线程到同样数量或者更小数量的内核线程上。 m 对 n (m >= n)
- 允许编程人员创建足够多的用户线程
- 避免线程阻塞引起进程阻塞、支持多处理器
- 例如

  Solaris 2

  IRIX

  HP-UX

  Tru64 UNIX

线程的取消

线程取消是指在线程完成之前终止线程的任务

例如：

- 多个线程并发执行以搜索数据库并且一个线程返回了结果
- 用户按下网页浏览器上的停止按钮，装入网页的线程就被取消

异步取消： 一个线程立即终止目标线程

延迟取消：目标线程不断的检查它是否应终止，允许一个线程检查它是否是在安全的点被取消

线程的特定数据

指同属一个进程的线程**共享进程数据**，每个线程都需要一定数据的拷贝

例如：

- 对于事务处理系统，需要通过独立线程以处理各个请求
- 每个事务都有一个唯一标识符，为了让每个线程与其唯一标识符相关联，可以使用线程特定数据

绝大多数线程库提供了对线程特定数据的支持，如

Win32, Pthreads, Java

3 进程调度

进程（Linux中称任务）的定义：进程是一个可并发执行的具有独立功能的程序关于某个数据集合的一次执行过程，也是操作系统进行资源分配和保护的基本单位。

描述进程的三个方面：

- 程序的一次运行活动；
- 进程的运行活动是建立在某个数据集合之上的；
- 进程在获得资源的基础上从事自己的运行活动。

需要解决的问题

对不同进程的区分与标记

引入进程控制块 (Process Control Block， PCB)

按什么原则分配 CPU 资源

进程调度算法

何时分配 CPU 资源

进程调度的时机

如何分配 CPU 资源

CPU调度过程（进程切换）

不同进程执行的时间控制（调度）

- 一次只有一个进程 “运行”
- 需要给重要进程更多的时间

不同进程彼此的资源保护

对非 CPU 资源设定控制访问机制。例如：

- 内存映射：给予每个进程自己的地址空间
- 内核/用户对偶性：通过系统调用进行 I/O 复用

进程控制块

- 是对于程序执行与其环境的 “快照”
- 任何时候只有一个 PCB 正在执行
- 是进程存在的唯一标志

PCB 内存储了：进程状态、程序计数器（IR）、CPU 寄存器、CPU 调度信息、内存管理信息、记账信息、I/O 状态信息等。这些信息根据进程的不同而不同。

上下文切换

中断

定义：程序执行过程中，当发生某个事件时，中止 CPU 上现行程序的运行，引出处理该事件的程序执行的过程。中断是由与现行指令无关的中断信号触发的(异步的)，且中断的发生与 CPU 处在用户模式或内核模式无关，在两条机器指令之间才可响应中断，一般来说，中断处理程序提供的服务不是为当前进程所需的，如时钟中断、硬盘读写服务请求中断。而异常是由处理器正在执行现行指令而引起的，一条指令执行期间允许响应异常，异常处理程序提供的服务是为当前进程所用的。异常包括很多方面，有出错(fault)，也有陷入(trap)。

优点：

- 能充分发挥处理机的使用效率
- 提高系统的实时处理能力

分类

从中断事件的性质和激活的手段来分

强迫中断：强迫性中断事件不是正在运行的程序所期待的，而是由于某种事故或外部请求信息所引起的，分为：机器故障中断事件、程序性中断事件、外部中断事件、输入输出中断事件。

自愿中断：自愿性中断事件是正在运行的程序所期待的事件。正在运行的程序对操作系统有某种需求，一旦机器执行到一条访管指令时，便自愿停止现行程序的执行而转入访管中断处理程序处理。

按照中断信号的来源分

外中断(又称中断)：指来自处理器和主存之外的中断。外中断包括电源故障中断、时钟中断、控制台中断、它机中断和 I/O 中断等。不同的中断具有不同的中断优先级，处理高一级中断时，往往会屏蔽部分或全部低级中断。

内中断(又称异常)：指来自处理器和主存内部的中断。内中断包括：通路校验错、主存奇偶错、非法操作码、地址越界、页面失效、调试指令、访管中断、算术操作溢出等各种程序性中断。异常是不能被屏蔽的，一旦出现应立即响应并加以处理。

按照是否由硬件产生信号分

硬中断：中断和异常要通过硬件设施来产生中断请求。

软中断：不必由硬件发信号而能引发的中断称软中断，软中断是利用硬件中断的概念，用软件方式进行模拟，实现宏观上的异步执行效果。

中断处理

机器故障中断事件的处理：事件是由硬件故障(例如，电源故障、主存储器故障)产生。中断处理能做的工作是：保护现场，防止故障蔓延，报告给操作员并提供故障信息以便维修和校正，对程序中所造成的破坏进行估价和恢复。

程序性中断事件的处理：采用中断续元处理来进行程序性中断事件的处理。

外部中断事件的处理：时钟是操作系统进行调度工作的重要工具，例如让分时进程作时间片轮转、让实时进程。

控制台中断事件的处理：操作员可以利用控制台开关请求操作系统工作，当使用控制台开关后，就产生一个控制台中断事件通知操作系统。操作系统处理这种中断就如同接受一条操作命令一样，转向处理操作命令的程序执行。

I/O中断的处理

中断的屏蔽：主机可允许或禁止某类中断的响应，如允许或禁止所有的 I/O 中断、外部中断、及某些程序性中断。有些中断是不能被禁止的，例如，计算机中的自愿性访管中断就不能被禁止。

多重中断事件的处理：中断正在进行处理期间，这时CPU又响应了新的中断事件，于是暂时停止正在运行的中断处理程序，转去执行新的中断处理程序。

进程切换的基本过程

1. 保存进程上下文环境
2. 更新当前运行进程的控制块内容，将其状态改为就绪或阻塞状态
3. 将进程控制块移到相应队列
4. 改变需投入运行进程的控制块内容，将其状态变为运行状态
5. 恢复需投入运行进程的上下文环境

进程控制的原语操作

Create( )

1. 申请空白 PCB
2. 为新进程分配资源
3. 初始化进程控制块
4.
    1. 初始化标识信息
    2. 初始化处理机状态信息
    3. 初始化处理机控制信息
5. 将新进程插入就绪队列

Stop( )

1. 根据进程 ID，检索进程 PCB，读取进程信息
2. 若被终止进程处于执行状态，则立即终止进程，并置调度指示符为真，用于指示该进程被终止后重新进行调度
3. 终止子孙进程
4. 归还资源
5. 移出已终止进程

block( )

1. 进程通过调用阻塞原语 block( ) 阻塞自己
2. 更改进程状态，并插入阻塞队列
3. 处理机重新调度，分配给另一就绪进程

wakeup( )

1. 将进程移出阻塞队列，并更改 PCB 中现行状态
2. 将 PCB 插入就绪队列

suspend( )

1. 首先检查被挂起进程的状态，若处于活动就绪状态，便将其改为静止就绪
2. 对于活动阻塞状态的进程，则将之改为静止阻塞状态

active( )

1. 将进程从外存调入内存，检查该进程的现行状态
2. 若是静止就绪，便将之改为活动就绪；若为静止阻塞，便将之改为活动阻塞

进程调度算法

进程调度算法的**任务**：控制协调进程对 CPU 的竞争，即按一定的调度算法从就绪队列中选中一个进程，把 CPU 的使用权交给被选中的进程。

进程调度算法的**原则**：

（1）具有**公平性**

（2）**资源利用率**高（特别是 CPU 利用率）

（3）在交互式系统情况下要追求**响应时间**

（4）在批处理系统情况下要追求**系统吞吐量**

（5）注重周转时间（ 从进程提交到进程完成的时间间隔）

（6）注重等待时间（在就绪队列中等待所花费的时间之和）

抢占与非抢占式

CPU调度决策(调度时机)可在如下四种环境下发生：

1. 当一个进程从运行状态切换到等待状态
2. 当一个进程从运行状态切换到就绪状态
3. 当一个进程从等待状态切换到就绪状态
4. 当一个进程终止时

可剥夺式（可抢占式 Preemptive）：当有比正在运行的进程优先级更高的进程就绪时，系统可强行剥夺正在运行进程的 CPU，提供给具有更高优先级的进程使用。如 UNIX, Linux, Windows NT

不可剥夺式（非可抢占式 Nonpreemptive ）：某一进程被调度运行后，除非由于它自身的原因不能运行，否则一直运行下去。如 Apple 曾经的 Macintosh

经典的调度算法

1. 先进先出（FIFO、FCFS）

按照进程就绪的先后次序来调度进程。

- 优点：实现简单
- 缺点：没考虑进程的优先级

2. 短作业优先（SJF）【最优】

以最短的时间成本来调度进程

可剥夺式与非可剥夺式

SJF 算法是最优的——给出了给定进程集的最小平均等待时间

3. 最短剩余时间优先 （SRTF）

该算法是 SJF 调度的抢先版本。 在 SRTF 中，进程的执行可以在一段时间后停止。 在每个进程到来时，短期调度程序在可用进程列表和正在运行的进程中以最少的剩余执行时间安排进程。

SRTF 至少和 SJF 调度算法一样好

4. 优先级调度（PF）

优先选择就绪队列中优先级最高的进程投入运行。优先级根据优先数来决定

静态优先数法：在进程创建时指定优先数，在进程运行时优先数不变。

动态优先数法：在进程创建时创立一个优先数，但在其生命周期内优先数可以动态变化。如等待时间长优先数可改变。

优先级的定义原则

外部因素

- 进程重要性
- 使用计算机支付的费用
- 赞助工作的单位
- 其它因素

内部因素

- 时间极限
- 内存要求
- 打开文件的数量
- 平均I/O时间区间与平均CPU区间之比

潜在的问题：低优先级的进程一直无法执行

解决方案：随着时间的推移，进程的优先级也会增加

5. 时间片轮转调度（RR）

专门为分时系统而设计，类似于 FCFS 调度，但增加了抢占以在进程间切换

每个进程获得一小段CPU时间 (时间量, 或时间片) ，通常是 10-100 ms。时间片用完后，CPU被抢占，进程排在就绪队列末尾。就绪队列可看作循环的 FIFO 队列，新进程放在就绪队列末尾。如果就绪队列中有 n 个进程，那么每个进程会得到 1/n 的CPU时间；若每个时间片长度不超过 q 时间单元，则每个进程等待 CPU 的时间不会超过 (n-1)*q 个时间单元。

性能很大程度上取决于时间片的大小：

- 如果时间片非常大，则接近于 FIFO（FCFS）
- 如果时间片很小，则响应时间短；但如果时间片过小，则上下文切换开销过大

与**时间片大小**有关的因素：系统响应时间、就绪进程个数、 CPU 能力。

6. 多级队列反馈调度

系统中设置多个就绪队列，每个就绪队列分配给不同大小的时间片，优先级高的为第一级队列，时间片最小，随着队列级别的降低，时间片加大。 各队列按照先进先出调度算法，一个新进程就绪后进入第一级队列。进程由于等待而放弃 CPU 后，进入等待队列，直到等待的事件发生，则回到原来的就绪队列。 当有一个优先级更高的进程就绪时，可以抢占 CPU，被抢占进程回到原来一级就绪队列末尾。当第一级队列空时，就去调度第二级队列，如此类推。对于每个进程，当分配的时间片用完后进程仍然没有执行完毕，则该进程放弃 CPU，回到下一级队列队尾。

多级反馈队列调度定义了如下的参数：

- 队列数量
- 每个队列的调度算法
- 用以确定进程何时升级到较高优先权队列的方法
- 用以确定进程何时降级到较低优先权队列的方法
- 用以确定进程在需要服务时应进入那个队列的方法

如何选择 CPU 进程调度算法?

定义准则

- CPU利用率、响应时间、吞吐量等

- 定义这些度量值的相对重要性
- 基于权重的平均值

基于准则评估性能

分析和选择算法

- 确定性模型
- 排队模型
- 模拟
- 实现

4 同步

问题的引入

并发线程在访问共享数据时会引入问题：

- 乱序：程序必须对顺序不敏感
- 一致性：对共享数据的并发访问会导致数据的不一致性，需要特别设计以确保共享变量一致

为维护数据的一致性和执行结果的正确性，进程间必须同步，协作按一定次序执行

原子操作

原子操作是指不会被线程调度机制打断的操作。这种操作一旦开始，就一直运行到结束，中间不会有任何 上下文切换。如何保护一个关键的部分，只有原子操作和存储，这将十分困难。

竞争、互斥与同步

当两个进程竞相访问同一数据时，就会发生**竞争**。由于时间片的原因，执行结果可能会被破坏或者被错误地解释。共享数据的值取决于哪个进程最后完成。为防止竞争，并发进程必须同步。

互斥是进程间相互排斥的使用临界资源。因此又称**间接制约关系**，即指系统中的某些共享资源，一次只允许一个线程访问。当一个线程正在访问该临界资源时，其它线程必须等待。

同步不是相互排斥的使用临界资源的关系，而是相互依赖，相互配合的关系。因此又称**直接制约关系**，即多个线程（或进程）为合作完成任务，必须严格按照规定的某种先后次序来运行。

临界资源

指一次只允许一个进程使用(访问)的资源。如：硬件打印机、磁带机等，软件的消息缓冲队列、变量、数组、缓冲区等。该类硬件或软件（如外设、共享代码段、共享数据结构），当多个进程在对其进行访问时（关键是进行写入或修改），必须**互斥**地进行。而有些共享资源可以同时访问，如只读数据等。

进程间资源访问冲突

- 共享变量的修改冲突
- 操作顺序冲突

进程间的制约关系

- 间接制约：进行竞争——独占分配到的部分或全部共享资源，“ 互斥 ”
- 直接制约：进行协作——等待来自其他进程的信息，“ 同步 “

临界区问题

临界区(critical section)： 进程中访问临界资源的一段代码

进入区(entry section)： 在进入临界区之前，进程用来检查可否进入临界区的一段代码。如果可以进入临界区，通常设置相应 "正在访问临界区" 标志

退出区(exit section)： 用于将 "正在访问临界区" 标志清除

剩余区(remainder section)： 代码中的其余部分

问题：需要保证当一个进程正在临界区执行时，没有另外的进程进入临界区执行（**互斥使用**）。

经典的进程同步问题

生产者-消费者问题（有界缓冲问题）

生产者-消费者 (producer-consumer) 问题是一个著名的进程同步问题。

1. 生产者进程将它所生产的产品放入一个缓冲区中
2. 消费者进程可从一个缓冲区中取走产品去消费
3. 生产者进程与消费者进程能并发执行，所以需要在两者之间设置一个具有n个缓冲区的缓冲池
4. 所有的生产者进程和消费者进程都是以异步方式运行的，但它们之间必须保持同步：即不允许消费者进程到一个空缓冲区去取产品；也不允许生产者进程向一个已装满产品且尚未被取走的缓冲区中投放产品
5. 生产者在缓冲区满时必须等待，直到缓冲区有空间才继续生产；消费者在缓冲区空时必须等待，直到缓冲区中有产品才能继续读取

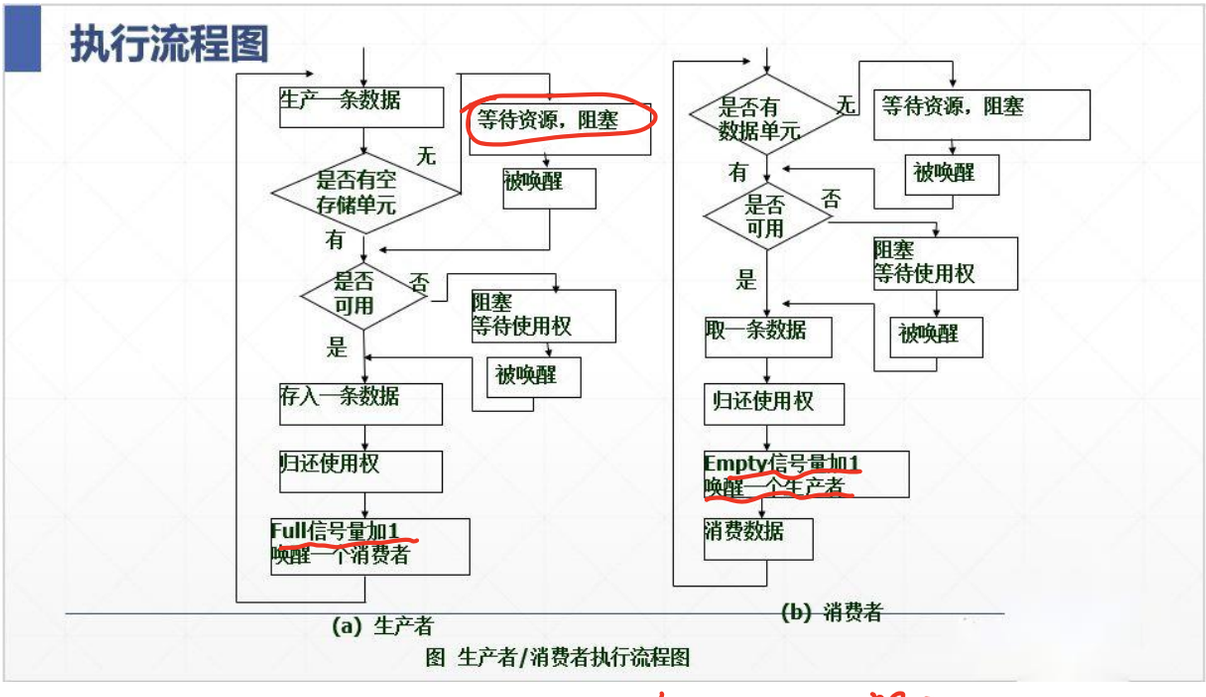


图 生产者/消费者执行流程图

哲学家就餐问题

同时拿左边的筷子.

哲学家就餐问题是由 Dijkstra 提出并解决的典型的同步问题。该问题描述的是五个哲学家共用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有五个碗和五只筷子，他们的生活方式是交替的进行思考和进餐。平时，一个哲学家进行思考，饥饿时便试图取用其左右最靠近他的筷子，只有在他拿到两只筷子时才能进餐。进餐完毕，放下筷子继续思考。这个问题可以用来解释死锁和资源耗尽。

读者-写者问题

读者写者问题是并发程序设计中的经典问题。问题描述为：有读者和写者两组并发进程，共享一个文件，当两个或以上的读进程同时访问共享数据时不会产生副作用，但若某个写进程和其他进程（读进程或写进程）同时访问共享数据时则可能导致数据不一致的错误。因此要求：

1. 允许多个读者可以同时对文件执行读操作；
2. 只允许同时一个写者往文件中写信息；
3. 任一写者在完成写操作之前不允许其他读者或写者工作；
4. 写者执行写操作前，应让已有的读者和写者全部退出。

可多个读
不可多写.

多副本一致性问题

主从同步方式

写请求首先发送给主副本，主副本同步更新到其它副本后返回。这种方式可以保证副本之间数据的强一致性，写成功返回之后从任意副本读到的数据都是一致的。但是可用性很差，只要任意一个副本写失败，写请求将执行失败。

主从异步方式

采用异步复制的方式，主副本写成功后立即返回，然后在后台异步的更新其它副本。这种方式可用性较好，只要主副本写成功，写请求就执行成功。但是不能保证副本之间数据的强一致性，写成功返回之后从各个副本读取到的数据不保证一致，只有主副本上是最新的数据，其它副本上的数据落后，只提供最终一致性。

解决同步问题遵循的原则

- **空闲让进**：当无进程进入临界区时，相应的临界资源处于空闲状态，因而允许一个请求进入临界区的进程立即进入自己的临界区。
- **忙则等待(互斥)**：当已有进程进入自己的临界区时，即相应的临界资源正被访问，因而其它试图进入临界区的进程必须等待，以保证进程互斥地访问临界资源。
- **有限等待**：对要求访问临界资源的进程，应保证进程能在有限时间进入临界区，以免陷入 “饥饿” 状态。
- **让权等待**：当进程不能进入自己的临界区时，应立即释放临界资源，以免进程陷入忙等。

信号量机制

1965年，荷兰学者 Dijkstra 提出的信号量机制是一种卓有成效的进程同步工具，在长期广泛的应用中，信号量机制又得到了很大的发展，它从整型信号量机制发展到记录型信号量机制，进而发展为 “信号集”机制。现在信号量机制已广泛应用于 OS 中。一种不需要忙等待 (Busy-waiting) 的同步工具

数据结构与基本原语

信号量 S 除一个整数值 S.value（计数）外，还有一个进程等待队列 S.L，其中存放着阻塞在该信号量的各个进程的标识。

- 信号量只能通过初始化和两个标准的原语来访问——作为 OS 核心代码执行，不受进程调度的打断。
- 初始化指定一个非负整数值，表示空闲资源总数（又称为"资源信号量"）——若为非负值表示当前的空闲资源数，若为负值其绝对值表示当前等待临界区的进程数。

1. 整型信号量

最初 Dijkstra 把整型信号量定义为一个整型量，除初始化外，仅能通过两个标准的原子操作 (Atomic Operation) **wait(S)** 和 **signal(S)** 来访问。这两个操作一直被分别称为P、V操作。 wait 和 signal 操作可描述为：

```
def wait (S):
    while S <= 0:
        wait
    S = S - 1

def signal (S):
    S = S + 1
```

为使多个进程能互斥地访问某临界资源，只需为该资源设置一个互斥信号量 mutex（即之前定义的S），

- 该互斥信号量的初值设为 1
- 各进程的临界区操作居于 wait(mutex) 和 signal(mutex) 操作之间的区域即可

```
semaphore mutex; //初始 mutex = 1
Process Pi:
do {
    wait(mutex);
    //critical section
    signal(mutex);
    //remainder section
} while (1);
```

问题：影响效率，资源利用率不高

2. 记录型信号量

在整型信号量机制中的 wait 操作，只要是信号量 S≤0， 就会不断地测试。因此，该机制并未遵循“让权等待”的准则， 而是使进程处于 “忙等” 的状态。记录型信号量是一种不存在 “忙等” 现象的进程同步机制，通过引入“让权等待”，解决整型信号量忙等的问题。其在信号量机制中，除了需要一个用于代表资源数目的整型变量 value 外，又增加一个进程链表 L，用于链接上述的所有等待进程。

```
def wait(S):
    S.value = S.value - 1;
    if S.value < 0:
        block(S,L)


def signal(S):
    S.value = S.value + 1
    if S.value <= 0:
        wakeup(S,L)
 end
```

在记录型信号量机制中，S.value 的初值表示系统中某类资源的数目，因而又称为资源信号量，对它的每次 wait 操作，意味着进程请求一个单位的该类资源。当 S.value < 0 时，表示该类资源已分配完毕，因此进程应调用 block 原语，进行自我阻塞，放弃处理机，并插入到信号量链表 S.L 中。

记录型信号量机制遵循了 “让权等待” 准则。 此时 S.value 的绝对值表示在该信号量链表中已阻塞进程的数目。对信号量的每次 signal 操作，表示执行进程释放一个单位资源。若加 1 后仍是 S.value≤0，则表示在该信号量链表中，仍有等待该资源的进程被阻塞，故还应调用 wakeup 原语，将 S.L 链表中的第一个等待进程唤醒。

如果 S.value 的初值为1，表示只允许一个进程访问临界资源，此时的信号量转化为互斥信号量。

3. AND 型信号量

基本思想：将进程在整个运行过程中需要的所有资源，一次性全部地分配给进程，待进程使用完后再一起释放。只要尚有一个资源未能分配给进程，其它所有可能为之分配的资源，也不分配给他。亦即，对若干个临界资源的分配，采取原子操作方式：要么全部分配到进程，要么一个也不分配。

由死锁理论可知，这样就可避免上述死锁情况的发生。为此，在 wait 操作中，增加了一个 “AND” 条件，故称为 AND 同步，或称为同时 wait 操作， 即 Swait(Simultaneous wait)。

```
def Swait(S1, S2, …, Sn):
    if Si ≥ 1 and … and Sn ≥ 1:
        for i from 1 to n:
            Si = Si - 1
    else:
        Place the process in the waiting queue associated with the first Si
found with         Si<1, and set the program count of this process to the
beginning of Swait               operation

def Ssignal(S1, S2, …, Sn):
    for i from 1 to n:
        Si = Si + 1
    Remove all the process waiting in the queue associated with Si into the
ready       queue
```

4. 信号量集

```
def Swait(S1, t1, d1, …, Sn, tn, dn):
    if Si ≥ t1 and … and Sn ≥ tn:
        for i from 1 to n:
            Si = Si - di
    else:
        Place the process in the waiting queue associated with the first Si
found with         Si<ti, and set the program count of this process to the
beginning of Swait             operation

def Ssignal(S1, d1, S2, …, Sn, dn):
    for i from 1 to n:
        Si = Si + di
    Remove all the process waiting in the queue associated with Si into the
ready       queue
```

一般 “信号量集” 的几种特殊情况：

(1) Swait(S, d, d)。 此时在信号量集中只有一个信号量 S， 但允许它每次申请 d 个资源，当现有资源数少于 d时，不予分配。

(2) Swait(S, 1, 1)。 此时的信号量集已蜕化为一般的记录型信号量 (S > 1时) 或互斥信号量 (S=1时)。

(3) Swait(S, 1, 0)。这是一种很特殊且很有用的信号量操作。当 S ≥ 1 时，允许多个进程进入某特定区；当 S 变为 0 后，将阻止任何进程进入特定区。换言之，它相当于一个可控开关。

信号量的应用

1. 实现进程互斥

```
semaphore mutex = 1;
process 1:
while (1){
    wait(mutex);
```

```
    // critical section
    signal(mutex);
    // remainder seetion
}
process 2:
while (1){
    wait(mutex);
    // critical section
    signal(mutex);
    // remainder seetion
}
```

注意： wait、signal 操作必须成对出现，有一个 wait 操作就一定有一个 signal 操作

- 当为互斥操作时，它们在同一进程中出现
- 当为同步操作时，则不在同一进程中出现

如果两个 wait 操作相邻，那么它们的顺序至关重要，而两个相邻的 signal 操作的顺序无关紧要。一个同步 wait 操作与一个互斥 wait 操作在一起时，同步 wait 操作在互斥 wait 操作前。

优点：简单（用 wait、signal 操作可解决任何同步互斥问题）

缺点：不够安全；wait、signal 操作使用不当会出现死锁；实现复杂

2. 利用记录型信号量解决生产者消费者问题

假定在生产者和消费者之间的公用缓冲池中，具有 n 个缓冲区，这时可利用**互斥信号量 mutex** 实现诸进程对缓冲池的互斥使用；利用**资源信号量 empty 和 full** 分别表示缓冲池中空缓冲区和满缓冲区的数量。又假定这些生产者和消费者相互等效，只要缓冲池未满，生产者便可将消息送入缓冲池；只要缓冲池未空，消费者便可从缓冲池中取走一个消息。对生产者—消费者问题可描述如下：

```
semaphore (mutex, empty, full) = (1,n,0);
item buffer[n];
int in, out = 0;

proceducer:
while (1) {
    //produce an item nextp;
    //…
    wait(empty);
    wait(mutex);
    buffer[in] = nextp;
    in = (in + 1) mod n;
    signal(mutex);
    signal(full);
}

consumer:
while (1) {
    wait(full);
    wait(mutex);
    nextc = buffer[out];
    out = (out + 1) mod n;
    signal(mutex);
    signal(empty);
    //consume the item in nextc;
}
```

在生产者—消费者问题中应注意：

1.首先，在每个程序中用于实现互斥的 wait(mutex) 和 signal(mutex) 必须成对地出现；

2.其次，对资源信号量 empty 和 full 的 wait 和 signal 操作，同样需要成对地出现，但它们分别处于不同的程序中。例如，wait(empty) 在计算进程中，而 signal(empty) 则在打印进程中，计算进程若因执行 wait(empty) 而阻塞， 则以后将由打印进程将它唤醒；

3.最后，在每个程序中的多个 wait 操作顺序不能颠倒。应**先执行对资源信号量的 wait 操作，然后再执行对互斥信号量的 wait 操作**，否则可能引起进程死锁。

3. 利用 AND 信号量解决生产者消费者问题

```
semaphore (mutex, empty, full) = (1,n,0);
item buffer[n];
int in, out = 0;

proceducer:
while (1) {
    //produce an item nextp;
    //…
    Swait(empty, mutex);
    buffer[in] = nextp;
    in = (in + 1) mod n;
    Ssignal(mutex, full);
}

consumer:
while (1) {
    Swait(full, mutex);
    nextc = buffer[out];
    out = (out + 1) mod n;
    Ssignal(mutex, empty);
    //consume the item in nextc;
}
```

4. 利用记录型信号量解决哲学家进餐问题

经分析可知，放在桌子上的筷子是临界资源，在一段时间内只允许一位哲学家使用。为了实现对筷子的互斥使用，可以用一个信号量表示一只筷子，由这五个信号量构成信号量数组。其描述如下：

```
semaphore chopstick[5];

philosopher i:
while (1) {
    wait(chopstick [i] );
    wait(chopstick [(i + 1) mod 5] );
    //…
    //eat;
    //…
    signal(chopstick [i] );
    signal(chopstick [(i + 1) mod 5] );
    // …
    //think;
}
```

可采取以下几种解决方法：

(1) 至多只允许有四位哲学家同时去拿左边的筷子，最终能保证至少有一位哲学家能够进餐，并在用毕时能释放出他用过的两只筷子，从而使更多的哲学家能够进餐。

(2) 仅当哲学家的左、右两只筷子均可用时，才允许他拿起筷子进餐。

(3) 规定奇数号哲学家先拿他左边的筷子，然后再去拿右边的筷子；而偶数号哲学家则相反。按此规定，将是1、2 号哲学家竞争 1 号筷子；3、4号哲学家竞争 3 号筷子。即五位哲学家都先竞争奇数号筷子，获得后，再去竞争偶数号筷子，最后总会有一位哲学家能获得两只筷子而进餐。

5. 利用 AND 信号量机制解决哲学家进餐问题

在哲学家进餐问题中，要求每个哲学家先获得两个临界资源 (筷子) 后方能进餐，这在本质上就是前面所介绍的 AND 同步问题，故用 AND 信号量机制可获得最简洁的解法。

```
semaphore chopstick[5];

philosopher i:
while (1) {
    think;
    Sswait(chopstick [(i+1) mod 5] , chopstick  [i] );
    eat;
    Ssignat(chopstick  [(i+1) mod 5] , chopstick  [i] );
}
```

6. 利用记录型信号量解决读者-写者问题

操作如下：为保证 Reader 与 Writer 进程在读或写时互斥，设置一个互斥信号量 Wmutex。 再设置一个整型变量 Readcount 表示正在读的进程数目。由于只要有一个 Reader 进程在读，便不允许 Writer 进程去写。因此，仅当 Readcount=0 , 表示尚无 Reader 进程在读时，Reader 进程才需要执行 Wait(Wmutex) 操作。若wait(Wmutex) 操作成功，则 Reader 进程去读，相应地做 Readcount+1 操作。当 Reader 进程在执行了Readcount 减 1 操作后其值为0时，才须执行 signal(Wmutex) 操作，以便让 Writer 进程写。因为 Readcount 是一个可被多个 Reader 进程访问的临界资源，因此，应该为它设置一个互斥信号量 rmutex 。

读者-写者问题可描述如下：

```
semaphore (rmutex, wmutex) = (1, 1);
int Readcount = 0;

Reader:
while (1) {
    wait(rmutex);
    if (readcount == 0)
        wait(wmutex);
    Readcount = Readcount + 1;
    signal(rmutex);
    // if I am the first reader tell all others
    // that the database is being read
    //…
    //    Perform read operation;
    //…
    wait(rmutex);
    readcount = readcount - 1;
    if (readcount == 0)
        signal(wmutex);
    signal(rmutex);
}
```

```
Writer:
while (1) {
    wait(wmutex);
    // Perform write operation;
    signal(wmutex);
}
```

7. 利用信号量集机制解决读者-写者问题

```
int RN;
semaphore (L, mx) = (RN, 1);

Reader:
while (1) {
    Swait(L,1,1);
    Swait(mx,1,0);
    //…
    //perform read operation;
    //…
    Ssignal(L,1);
}

Writer:
while (1) {
    Swait(mx,1,1; L,RN,0);
    //perform write operation;
    Ssignal(mx,1);
}
```

管程（Monitor）

管程的提出

（1）PV 操作的缺点

1. 易读性差：因为要了解对于一组共享变量及信号量的操作是否正确，则必须通读整个系统或者并发程序。

2. 不利于修改和维护：因为程序的局部性很差，所以任一组变量或一段代码的修改都可能影响全局。

3. 正确性难以保证：因为操作系统或并发程序通常很大，而 PV 操作代码都是由用户编写的，系统无法有效地控制和管理这些P，V操作，要保证这样一个复杂的系统没有逻辑错误是很难的，它将导致死锁现象的产生。

（2）管程的引入

1. 把分散在各进程中的临界区集中起来进行管理；
2. 防止进程有意或无意的违法同步操作；
3. 便于用高级语言来书写程序，也便于程序正确性验证

管程的属性

（1）共享性：

（2）安全性：

（3）互斥性：

管程的特征

（1）**模块化：**一个管程是一个基本程序单位，可以单独编译；

（2）**抽象数据类型：**管程是一种特殊的数据类型，其中不仅有数据，而且有对数据进行操作的代码；

（3）**信息掩蔽**：管程是半透明的，管程中的外部过程（函数）实现了某些功能，而这些功能是怎样实现的，在其外部则是不可见的。

管程的组成部分

（1）名称

（2）数据结构说明

（3）对该数据结构进行操作的一组过程/函数

（4）初始化语句

管程的要素

（1）管程中的共享变量在管程外部是不可见的，外部只能通过调用管程中所说明的外部过程（函数）来间接地访问管程中的共享变量；

（2）为了保证管程共享变量的数据完整性，规定管程**互斥进入**；

（3）管程通常是用来管理资源的，因而在管程中应当设有进程等待队列以及相应的等待及唤醒操作

管程与进程的异同

（1）管程定义的是**公用数据结构**，而进程定义的是**私有数据结构**；

（2）管程把共享变量上的同步操作集中起来，而临界区却分散在每个进程中；

（3）管程是为管理共享资源而建立的，进程主要是为占有系统资源和实现系统并发性而引入的；

管程的条件变量：

（1）条件变量：当调用管程过程的进程无法运行时，用于阻塞进程的一种信号量。

（2）同步原语 wait： 当一个管程过程发现无法继续时，它在某些条件变量 condition 上执行 wait，这个动作引起调用进程阻塞。另一个进程可以通过对其伙伴在等待的同一个条件变量 condition 上执行同步原语 signal 操作来唤醒等待进程。

（3）条件变量与 PV 操作中信号量的区别：使用 signal 释放等待进程时，可能出现两个进程同时停留在管程内。解决方法：

1. 执行 signal 的进程等待，直到被释放进程退出管程或等待另一个条件。
2. 被释放进程等待，直到执行 signal 的进程退出管程或等待另一个条件。
3. 霍尔采用了第一种办法，汉森选择了两者的折衷，规定管程中的过程所执行的 signal 操作是过程体的最后一个操作。

管程的形式

```
TYPE monitor_name = MONITOR;
共享变量说明
define 本管程内所定义、本管程外可调用的过程（函数）名字表；
use 本管程外所定义、本管程内将调用的过程（函数）名字表；

PROCEDURE 过程名（形参表）；
```

```
    过程局部变量说明;
    BEGIN
    语句序列;
    END;
......

FUNCTION 函数名（形参表）：值类型;
    函数局部变量说明;
    BEGIN
    语句序列;
    END;

......

BEGIN
 共享变量初始化语句序列;
END;
```

实例

```
TYPE SSU = MONITOR
var busy : boolean;
    nobusy : semaphore;
define require, return;
use wait, signal;

procedure require;
begin
    if busy then wait(nobusy); /*调用进程加入等待队列*/
        busy := true;
end;

procedure return;
begin
    busy := false;
    signal(nobusy); /*从等待队列中释放进程*/
end;

begin /*管程变量初始化*/
    busy := false;
end;
```

利用管程解决生产者-消费者问题

在利用管程方法来解决生产者-消费者问题时， 首先便是为它们建立一个管程，并命名为 Producer-Consumer , 或简称为 PC 。其中包括两个过程：

(1) put(item) 过程。生产者利用该过程将自己生产的产品投放到缓冲池中，并用整型变量 count 来表示在缓冲池中已有的产品数目，当 count ≥ n 时， 表示缓冲池已满， 生产者须等待。

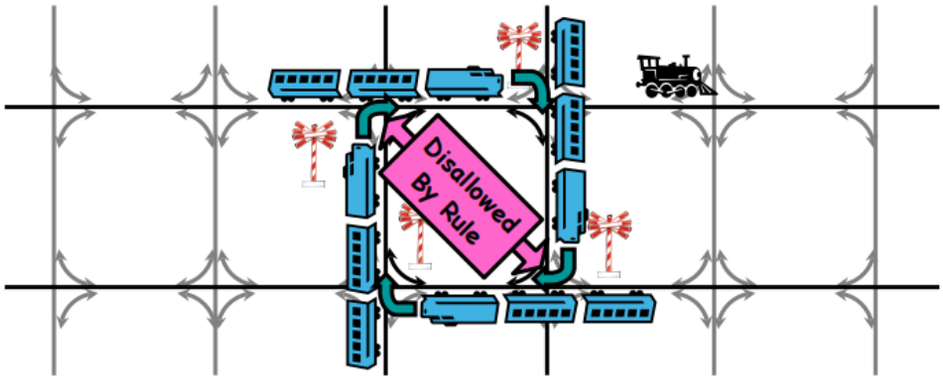
(2) get(item) 过程。消费者利用该过程从缓冲池中取出一个产品，当 count ≤ 0 时，表示缓冲池中已无可取用的产品， 消费者应等待。

```
type producer-consumer = monitor
Var in,out,count:integer;
    buffer:array [0,…,n-1]  of item;
    notfull, notempty:condition;
```

```
procedure entry put(item)
begin
    if count >= n then notfull.wait;
    buffer(in) := nextp;
    in := (in + 1) mod n;
    count := count+1;
    if notempty.queue then notempty.signal;
end

procedure entry get(item)
begin
    if count <= 0 then notempty.wait;
    nextc := buffer(out);
    out := (out+1) mod n;
    count := count-1;
    if notfull.quene then notfull.signal;
end

begin
    in := out := 0;
    count := 0
end
```

```
producer:
begin
    repeat
        produce an item in nextp;
        PC.put(item);
    until false;
end

consumer:
begin
    repeat
        PC.get(item);
        consume the item in nextc;
    until false;
end
```

5 死锁



死锁不仅与系统拥有的资源数量有关，而且与资源分配策略、进程对资源的使用要求以及并发进程的推进顺序有关。

死锁产生的四个必要条件

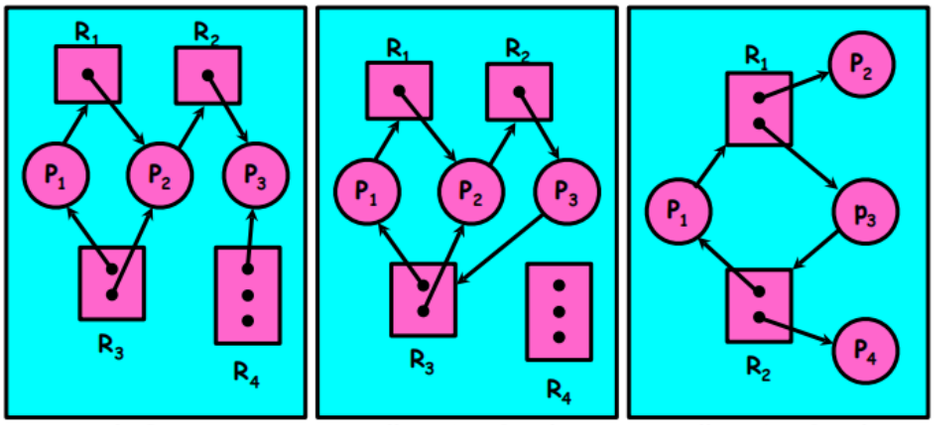
1. **互斥条件(Mutual exclusion)：进程互斥使用资源。**
2. **部分分配条件(Hold and wait)：申请新资源时不释放已占有资源。**
3. **不剥夺条件(No preemption)：一个进程不能抢夺其他进程占有的资源。**
4. **环路条件(Circular wait)：存在一组进程循环等待资源**

系统资源分配模型

1. 节点集合，分为两类:

- P = {P1, P2, ..., Pn}, 由系统中的所有进程组成的集合
- R = {R1, R2, ..., Rm}, 由系统中的所有资源类型组成的集合。每个资源类型 Ri 都有 Wi 个实例

2. 资源申请边：有向边 Pi → Rj
3. 资源分配边：有向边 Rj → Pi



死锁判断

如果图没有环，那么不会有死锁

如果图有环

- 如果每一种资源类型只有一个实例，那么死锁发生
- 如果一种资源类型有多个实例，**可能**死锁

系统为死锁状态的**充要**条件：当且仅当该状态的进程-资源分配图是不可完全简化的。该充分条件称为死锁定理

资源分配图化简

1. 找一个非孤立点进程结点且只有分配边，去掉分配边，将其变为孤立结点。
2. 再把相应的资源分配给一个等待该资源的进程，即将某进程的申请边变为分配边。

如果进程-资源分配图中有环路，且涉及的资源类中有多个资源，则环路的存在只是产生死锁的必要条件而不是充分条件。如果能在进程-资源分配图中消去此进程的所有请求边和分配边，成为孤立结点。经一系列简化，使所有进程成为孤立结点，则该图是可完全简化的；否则则称该图是不可完全简化的

操作系统如何对待死锁

允许死锁的发生，然后恢复

- 需要死锁检测算法，操作系统不断监视系统进展情况，判断死锁是否发生
- 需要一些技术来强制抢占资源和/或终止任务，一旦死锁发生则采取专门的措施，解除死锁并以最小的代价恢复操作系统运行

确保系统永远不会进入死锁

- 需要监控所有可能产生的死锁
- 有选择地终止那些可能导致死锁的情况

忽略问题，假装死锁不会发生

- 大多数操作系统使用，包括 UNIX

死锁检测算法

（1）每个进程和资源指定唯一编号；

（2）设置一张资源分配表；

（3）记录各进程与其占用资源之间的关系；

（4）设置一张进程等待表；

（5）记录各进程与要申请资源之间的关系

死锁发生后的处理:

1）立即结束所有进程的执行，并重新启动操作系统。方法简单，但以前工作全部作废，损失可能很大。

2）撤销陷于死锁的所有进程,解除死锁继续运行。

3）逐个撤销陷于死锁的进程,回收其资源,直至死锁解除。

4）剥夺陷于死锁的进程占用的资源，但并不撤销它, 直至死锁解除。

5）根据系统保存的 checkpoint，让所有进程回退，直到足以解除死锁。

6）当检测到死锁时，如果存在某些未卷入死锁的进程，而这些进程随着建立一些新的抑制进程能执行到结束，则它们可能释放足够的资源来解除死锁。

死锁的预防（Prevention）

- **破坏第一个条件**：使资源可**同时**访问而不是互斥使用，这是个简单的办法，磁盘可用这种办法管理，但有许多资源往往是不能同时访问，所以这种做法许多场合行不通。
- **破坏第三个条件**：**采用剥夺式调度方法**可破坏第三个条件，但只适用于对主存资源和处理器资源的分配，当进程在申请资源未获准许的情况下,如果主动释放资源(一种剥夺式),然后才去等待，以后再一起向系统提出申请,也能防止死锁。
- **破坏第二个条件或第四个条件**：种种死锁防止办法施加于资源的限制条件太严格,会造成资源利用率和吞吐率低。两种比较实用的死锁防止方法，它们能破坏第二个条件或第四个条件：**执行前一次申请全部资源、没有占有资源时才能分配资源**

预防措施

（1）**静态分配**：一个进程必须在执行前就申请它所要的全部资源，并且直到它所要的资源都得到满足后才开始执行。

（2）**层次分配策略：**资源被分成多个层次，当进程得到某一层的一个资源后，它只能再申请较高层次的资源。当进程要释放某层的一个资源时，必须先释放占有的较高层次的资源。当进程得到某一层的一个资源后，它想申请该层的另一个资源时，必须先释放该层中的已占资源。

（3）**层次分配策略的变种——按序分配策略：**把系统的所有资源排一个顺序，例如，系统若共有 n 个进程,共有m 个资源，用 ri 表示第 i 个资源，于是这 m 个资源是：r1,r2……,rm。规定进程不得在占用资源 ri(1≤i≤m)后再申请 rj(j<i)。不难证明，按这种策略分配资源时系统不会发生死锁。

死锁的避免 (Avoidance)

每次进行资源分配时，通过判断系统状态来决定这次分配后，是否仍存在**不安全状态**，否则不予分配。

- 每个进程声明其所需每种资源的最大数目
- 动态检查当前资源分配状态
- 资源分配状态取决于当前可用资源，已分配资源，以及进程声明所需最大资源数

Safe State 安全状态

如果系统能按某个顺序为每个进程分配资源（不超过其最大值）并能避免死锁，那么系统状态就是安全的。

- 如果存在一个安全序列，那么系统处于安全态，
- 如果一个系统处于安全状态 ⇒ 就没有死锁
- 如果一个系统不是处于安全状态 ⇒ 就有可能死锁

银行家算法

（1）对每个请求进行检查，是否会导致不安全状态。若是，则不满足该请求；否则便满足。

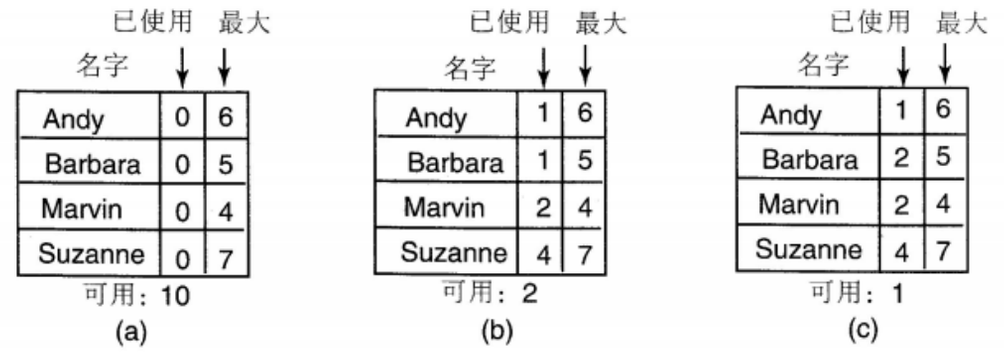
（2）检查状态是否安全的方法是看他是否有足够的资源满足一个距最大需求最近的客户，如此反复下去。如果所有投资最终都被收回，则该状态是安全的，最初的请求可以批准。

（3）4 个客户每个都有一个贷款额度：

（4）一个状态被称为是安全的，条件是**存在一个状态序列**能够使所有的客户**均得到其所有的贷款。**

（5）图示状态是安全的，以使 Marvin 运行结束，释放所有的 4 个单位资金。这样下去便可满足 Suzanne 或 Barbara 的请求。

（6）如果所有客户忽然都申请，希望得到最大贷款额，而银行家无法满足其中任何一个要求，则发生死锁。



银行家算法的基本思想

1. 系统中的所有进程进入进程集合,
2. 在安全状态下系统收到进程的资源请求后,先把资源试探性分配给它。

3. 系统用剩下的可用资源和进程集合中其他进程还要的资源数作比较，在进程集合中找到**剩余资源能满足最大需求量**的进程,从而，保证这个进程运行完毕并归还全部资源。
4. 把这个进程从集合中去掉, 系统的剩余资源更多了，反复执行上述步骤。
5. 最后,检查进程集合，若为空表明本次申请可行，系统处于安全状态，可实施本次分配；否则，有进程执行不完，系统处于不安全状态,本次资源分配暂不实施，让申请进程等待。

思考题

一个 OS 有 20 个进程，竞争使用 65 个同类资源，申请方式是逐个进行的，一旦某个进程获得它所需要的全部资源，则立即归还所有资源。每个进程最多使用三个资源。若仅考虑这类资源，该系统有无可能产生死锁，为什么?

在某系统中，三个进程共享四台同类型的设备资源，这些资源一次只能一台地为进程服务和释放，每个进程最多需要二台设备资源，试问在系统中是否会产生死锁?

某系统中有 n 个进程和 m 台打印机，系统约定：打印机只能一台一台地申请、一台一台地释放，每个进程需要同时使用的打印机台数不超过 m。如果 n 个进程同时使用打印机的总数小于 m+n ，试讨论，该系统可能发生死锁吗?并简述理由。

仅涉及一个进程的死锁有可能存在吗?为什么?

6 进程通信

基本方式

1. 管道 (Pipe) 通信

所谓 “管道” ，是指用于连接一个读进程和一个写进程以实现他们之间通信的一个**共享文件**，又名 pipe 文件。由于发送进程和接收进程是利用管道进行通信的，故又称为管道通信。

1. 发送进程(即写进程)：向管道(共享文件)提供输入，以字符流形式将大量的数据送入管道；
2. 接收进程(即读进程)：接受管道输出，从管道中接收(读)数据。

这种方式首创于 UNIX 系统，由于它能有效地传送大量数据，因而又被引入到许多其它操作系统中） 为了协调双方的通信，管道机制必须提供以下三方面的协调能力：

① **互斥**，即当一个进程正在对pipe执行读/写操作时，其它 (另一) 进程必须等待。

② **同步**，指当写 (输入) 进程把一定数量 (如 4 KB) 的数据写入 pipe ，便去睡眠等待， 直到读 (输出) 进程取走数据后，再把他唤醒。当读进程读一空 pipe 时，也应睡眠等待，直至写进程将数据写入管道后，才将之唤醒。

③ **确定对方是否存在**，只有确定了对方已存在时，才能进行通信。

2. 共享存储系统 (Shared-Memory System)

(1) 基于**共享数据结构**的通信方式。

(2) 基于**共享存储区**的通信方式

3. 消息传递系统 (Message passing system)

不论是单机系统、多机系统，还是计算机网络，消息传递机制都是用得最广泛的一种进程间通信的机制。

1. 消息传递系统中的进程间数据交换，以格式化消息 (message) 为单位的（在计算机网络中，message 称为报文。程序员直接利用系统提供的一组通信命令 (原语) 进行通信）
2. 操作系统隐藏了通信的实现细节，大大减化了通信程序编制的复杂性，而获得广泛的应用。
3. 消息传递系统的通信方式属于高级通信方式。又因其实现方式的不同而进一步分成直接通信方式和间接通信方式两种。

直接通信方式

这是指发送进程利用 OS 所提供的发送命令，直接把消息发送给目标进程。此时，要求发送进程和接收进程都以显式方式提供对方的标识符。通常，系统提供下述两条通信命令(原语)：

- Send (Receiver, message); 发送一个消息给接收进程；
- Receive (Sender, message); 接收 Sender 发来的消息；

例如，原语 Send(P2, m1) 表示将消息 m1 发送给接收进程 P2; 而原语 Receive(P1，m1) 则表示接收由 P1 发来的消息 m1。 在某些情况下，接收进程可与多个发送进程通信，因此，它不可能事先指定发送进程。例如，用于提供打印服务的进程，它可以接收来自任何一个进程的 “打印请求” 消息。对于这样的应用，在接收进程接收消息的原语中的源进程参数，是完成通信后的返回值，接收原语可表示为：Receive (id, message);

生产者-消费者问题：直接通信原语

1. 当生产者生产出一个产品(消息)后，便用Send原语将消息发送给消费者进程；
2. 而消费者进程则利用Receive原语来得到一个消息。如果消息尚未生产出来，消费者必须等待，直至生产者进程将消息发送过来。

生产者-消费者的通信过程描述如下：

```
repeat
   …
   repeat
       produce an item in nextp;
       …
       send(consumer, nextp);
   until false;
   repeat
       receive(producer, nextc);
       …
       consume the item in nextc;
   until false;
until false;
```

间接通信方式

(1) **信箱**的创建和撤消。进程可利用信箱创建原语来建立一个新信箱。创建者进程应给出信箱名字、信箱属性(公用、私用或共享)；对于共享信箱， 还应给出共享者的名字。当进程不再需要读信箱时，可用信箱撤消原语将之撤消。

(2) 消息的发送和接收。当进程之间要利用信箱进行通信时，必须使用共享信箱，并利用系统提供的下述通信原语进行通信：

- Send(mailbox, message); 将一个消息发送到指定信箱；
- Receive(mailbox, message); 从指定信箱中接收一个消息；

信箱

信箱可由操作系统创建，也可由用户进程创建，创建者是信箱的拥有者。据此，可把信箱分为以下三类。

私用信箱

用户进程可为自己建立一个新信箱，并作为该进程的一部分。信箱的拥有者有权从信箱中读取消息，其他用户则只能将自己构成的消息发送到该信箱中。这种私用信箱可采用单向通信链路的信箱来实现。 当拥有该信箱的进程结束时，信箱也随之消失。

公用信箱

它由操作系统创建，并提供给系统中的所有核准进程使用。核准进程既可把消息发送到该信箱中，也可从信箱中读取发送给自己的消息。显然，公用信箱应采用双向通信链路的信箱来实现。通常，公用信箱在系统运行期间始终存在。

共享信箱

它由某进程创建，在创建时或创建后，指明它是可共享的，同时须指出共享进程(用户)的名字。信箱的拥有者和共享者，都有权从信箱中取走发送给自己的消息。

在利用信箱通信时，在发送进程和接收进程之间，存在以下四种关系：

1. 一对一关系。这时可为发送进程和接收进程建立一条两者专用的通信链路，使两者之间的交互不受其他进程的干扰。
2. 多对一关系。允许提供服务的进程与多个用户进程之间进行交互，也称为客户/服务器交互(client/server interaction)。
3. 一对多关系。允许一个发送进程与多个接收进程进行交互，使发送进程可用广播方式，向接收者(多个)发送消息。
4. 多对多关系。允许建立一个公用信箱，让多个进程都能向信箱中投递消息；也可从信箱中取走属于自己的消息。

消息传递系统实现中的若干问题

1. 通信链路(communication link)

为使在发送进程和接收进程之间能进行通信，必须在两者之间建立一条**通信链路**。有两种方式建立通信链路。第一种方式是：由发送进程在通信之前，用显式的 “建立连接” 命令(原语)请求系统为之建立一条通信链路；在链路使用完后，也用显式方式拆除链路。第二种方式是：发送进程无须明确提出建立链路的请求，只须利用系统提供的发送命令(原语)，系统会**自动**地为之建立一条链路。这种方式主要用于单机系统中。 根据通信链路的连接方法，又可把通信链路分为两类：

① 点—点连接通信链路，这时的一条链路只连接两个结点(进程)；

② 多点连接链路，指用一条链路连接多个($n > 2$)结点(进程)。

而根据通信方式的不同，则又可把链路分成两种：

① 单向通信链路，只允许发送进程向接收进程发送消息；

② 双向链路，既允许由进程 A 向进程 B 发送消息，也允许进程B同时向进程 A 发送消息。

2. 消息的格式

在某些 OS 中，消息是采用比较短的定长消息格式，这减少了对消息的处理和存储开销。这种方式可用于办公自动化系统中，为用户提供快速的便笺式通信；但这对要发送较长消息的用户是不方便的。在有的 OS 中，采用另一种变长的消息格式，即进程所发送消息的长度是可变的。系统在处理和存储变长消息时，须付出更多的开销，但方便了用户。 这两种消息格式各有其优缺点，故在很多系统(包括计算机网络)中，是同时都用的。

消息缓冲队列通信机制

1. 消息缓冲队列通信机制中的数据结构

(1) 消息缓冲区。在消息缓冲队列通信方式中，主要利用的数据结构是消息缓冲区。它可描述如下：

```
type message buffer=record
     sender; 发送者进程标识符
     size; 消息长度
     text; 消息正文
     next; 指向下一个消息缓冲区的指针
end
```

(2) PCB 中有关通信的数据项。在利用消息缓冲队列通信机制时，在设置消息缓冲队列的同时，还应增加用于**对消息队列进行操作和实现同步的信号量**，并将它们置入进程的 PCB 中。在 PCB 中应增加的数据项可描述如下：

```
type processcontrol block=record
     …
     mq; 消息队列队首指针
     mutex; 消息队列互斥信号量
     sm; 消息队列资源信号量
     …
 end
```

2. 发送原语

发送进程在利用发送原语发送消息之前，应先在自己的内存空间，设置一发送区a，把待发送的消息正文、发送进程标识符、消息长度等信息填入其中，然后调用发送原语，把消息发送给目标(接收)进程。发送原语首先根据发送区 a 中所设置的消息长度 a.size 来申请一缓冲区 i ，接着，把发送区 a 中的信息复制到缓冲区i中。为了能将 i 挂在接收进程的消息队列 mq 上，应先获得接收进程的内部标识符 j ，然后将 i 挂在 j.mq 上。

由于该队列属于临界资源， 故在执行 insert 操作的前后，都要执行 wait 和 signal 操作。

```
procedure send(receiver, a)
begin
    getbuf(a.size,i); 根据 a.size 申请缓冲区
    将发送区 a 中的信息复制到消息缓冲区之中
    i.sender := a.sender;
    i.size := a.size;
    i.text := a.text;
    i.next := 0;
    getid(PCB set, receiver.j); 获得接收进程内部标识符
    wait(j.mutex);
    insert(j.mq, i); 将消息缓冲区插入消息队列;
    signal(j.mutex);
    signal(j.sm);
end
```

3. 接收原语

接收原语描述如下：

```
procedure receive(b)
begin
    j := internal name; j 为接收进程内部的标识符
    wait(j.sm);
    wait(j.mutex);
    remove(j.mq, i); 将消息队列中第一个消息移出
    signal(j.mutex);
    将消息缓冲区 i 中的信息复制到接收区 b
    b.sender := i.sender;
    b.size := i.size;
    b.text := i.text;
end
```

Linux 进程通信

类型

- 管道（pipe）
- 命名管道（FIFO）
- 信号（signal）
- 信号量
- 共享内存
- 内存映射
- 消息队列
- 套接字（socket）

类型	无连接	可靠	流控制	记录消息类型	优先级
普通PIPE	N	Y	Y		N
流PIPE	N	Y	Y		N
命名PIPE(FIFO)	N	Y	Y		N
消息队列	N	Y	Y		Y
信号量	N	Y	Y		Y
共享存储	N	Y	Y		Y
UNIX流SOCKET	N	Y	Y		N
UNIX数据包SOCKET	Y	Y	N		N

管道通信

管道这种通讯方式有两种限制，一是半双工的通信，数据只能单向流动；二是只能在具有亲缘关系的进程间使用。进程的**亲缘关系**通常是指父子进程关系，所以管道允许一个进程和另一个与它有共同祖先的进程之间进行通信。

- **流管道 s_pipe**：去除了第一种限制，可以双向传输

- **命名管道 name_pipe**：克服了管道没有名字的限制，因此除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信，即可以用于任何两个进程之间的通信

信号量通信

信号量用来控制多个进程对共享资源的访问。它常作为一种**锁机制**，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

Linux 除了支持 Unix 早期信号语义函数 signal 外，还支持语义符合 Posix.1 标准的信号函数 igaction（实际上，该函数是基于 BSD 的，BSD 为了实现可靠信号机制，又能够统一对外接口，用 sigaction 函数重新实现了 signal 函数）

消息队列通信

消息队列是消息的链表，存放在内核中并由消息队列标识符标识。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。

消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。

内存映射通信

内存映射允许**任何多个进程**间通信，每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现。

共享内存通信

共享内存就是映射一段能被其他进程所访问的内存，让多个进程可以访问同一块内存空间，这段共享内存由一个进程创建，但多个进程都可以访问。

共享内存是**最快的 IPC** 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。

Socket 通信

套接字/口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。

Socket 是更为一般的进程间通信机制，可用于不同机器之间的进程间通信。由 Unix 系统的 BSD 分支起源，但现在一般可以移植到其它类 Unix 和 Linux 系统上，应用十分广泛。

共享内存通信方式分析

共享内存

- 针对消息缓冲的缺点改而利用内存缓冲区直接交换信息，无须复制，快捷、信息量大是其优点
- 通信方式是通过将共享的内存缓冲区直接附加到进程的虚拟地址空间中来实现的
- 用户态进程之间的读写操作的同步问题操作系统无法实现。必须由各进程利用其他同步工具解决
- 由于内存实体存在于计算机系统中，所以只能由处于同一个计算机系统中的诸进程共享。不方便网络通信

共享内存块

- 提供了在任意数量的进程之间进行高效双向通信的机制。每个使用者都可以读取写入数据，但是所有程序之间必须达成并遵守一定的协议，以防止诸如在读取信息之前覆写内存空间等竞争状态的出现

Linux 无法严格保证提供对共享内存块的独占访问，甚至是在使用 IPC_PRIVATE 创建新的共享内存块的时候也不能保证访问的独占性。 同时，多个使用共享内存块的进程之间必须协调使用同一个键值。

不同通信方式的比较

管道：速度慢，容量有限，只有父子进程能通讯

命名管道（FIFO）：任何进程间都能通讯，但速度慢

消息队列：容量受到系统限制，且要注意第一次读的时候，要考虑上一次没有读完数据的问题

信号量：不能传递复杂消息，只能用来同步

共享内存区：容量可控，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全。当然，共享内存区同样可以用作线程间通讯，不过一般没这个必要，线程间本来就已经共享了同一进程内的一块内存。

若用户传递的**信息较少**，需要通过**信号**来触发某些行为，可以选择软中断信号/信号量：

若进程间要求传递的**信息量比较大**或者进程间**存在交换数据**的要求，可以选择：

- **无名管道。**简单方便．但局限于单向通信的工作方式．并且只能在创建它的进程及其子孙进程之间实现管道的共享：
- **有名管道。**虽然可以提供给任意关系的进程使用．但是由于其长期存在于系统之中，使用不当容易出错．所以普通用户一般不建议使用。
- **消息缓冲。**可以不再局限于父子进程，而允许任意进程通过共享消息队列来实现进程间通信，并由系统调用函数来实现消息发送和接收之间的同步，从而使得用户在使用消息缓冲进行通信时不再需要考虑同步问题，使用方便，但是信息的复制需要额外消耗 CPU 的时间，不适宜于信息量大或操作频繁的场合。