# 1. 进程

## 1.1. 进程描述

### 1.1.1. 进程描述与组成

进程：一个具有一定独立功能的**程序**在一个数据集合上的一次**动态**执行过程。

一个进程包括：程序的代码；程序处理的数据；程序计数器中的值，指示下一条指令；一组通用寄存器；一组系统资源；

### 1.1.2. 进程与程序的关系与区别

进程与程序的关系：

1. 程序是进程产生的基础；
2. 程序每次运行构成不同的进程；
3. 进程是程序功能的体现；
4. 通过多次执行，一个程序可对应多个进程；通过调用关系，一个进程可包括多个程序。

进程与程序的区别：

1. 进程是动态的，程序是静态的；程序是有序代码的集合，进程是程序的执行，进程有核心态、用户态。
2. 进程时暂时的，程序是永久的；
3. 进程与程序的组成不同：进程的组成包括程序、数据和进程控制块。

### 1.1.3. 进程的特点

动态性：动态创建、结束进程

并发性：进程可以被独立的调度并占用CPU运行，并发运行；

独立性：不同的进程的工作不互相影响（独立的地址空间、页表）；

制约性：因访问共享资源或进程间同步而产生制约。

### 1.1.4. 进程控制结构TCB

进程控制块：OS管理控制进程所用的信息集合。PCB是进程存在的唯一标识。

进程控制块包括：

1. 进程标识信息：如本进程的标识，本进程的父进程；用户标识；
2. 处理器状态信息：保存程序的运行现场信息（地址等寄存器；控制和状态寄存器；栈指针）
3. 进程控制信息：调度和状态信息；进程间通信信息；存储管理信息；进程所用资源。

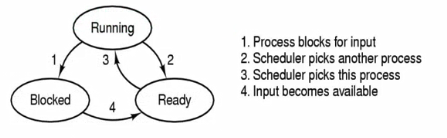
## 1.2. 进程的生命期管理

### 1.2.1. 进程状态

引起进程创建的3个主要事件：系统初始化、用户请求创建一个新进程、正在运行的进程调用了一个创建进程的系统调用。

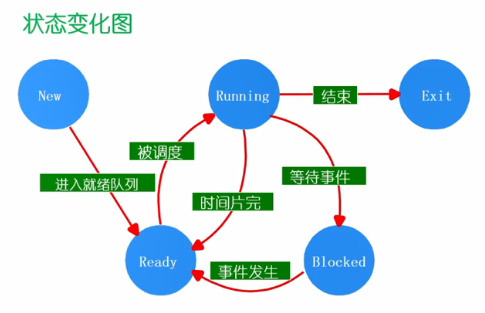
**三态模型**：

1. 运行态：当一个进程正在处理机上运行
2. 就绪态：一个进程获得了除CPU之外的一切所需资源，一旦得到CPU即可运行
3. 阻塞态：一个进程正在等待某一事件而暂时停止运行。如等待输入输出完成。



进程的其他状态：

* 创建状态（new）：一个进程正在被创建，还没转到就绪态之前的状态；
* 结束状态（exit）：一个进程正在从系统中消失时的状态，这是因为进程结束或由于其他原因导致。



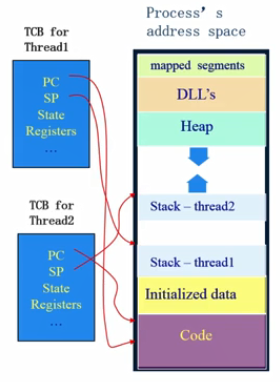
### 1.2.2. 进程挂起

进程在挂起状态时，意味着进程没有占用内存空间。处在挂起状态的进程映像在磁盘上。

挂起状态：

* 阻塞挂起：进程在外存并等待某个事件的出现；
* 就绪挂起：进程在外存，但只要进入内存，即可运行；

挂起：把一个进程从内存转到外存，可能有以下几种情况：

1. 阻塞到阻塞挂起：没有进程处于就绪状态或者就绪进程要求更多的内存资源时，会进行这种转换，以提交新进程或运行就绪进程；
2. 就绪到就绪挂起：当有高优先权阻塞进程和低优先级就绪进程时，系统会选择挂起低优先级就绪进程；
3. 运行到就绪挂起：对抢占式分时系统，当有高优先级阻塞挂起进程因事件出现而进入就绪挂起时，系统可能会把运行进程转到就绪挂起状态。

## 1.3. 线程

进程中的一条执行流程，即线程。

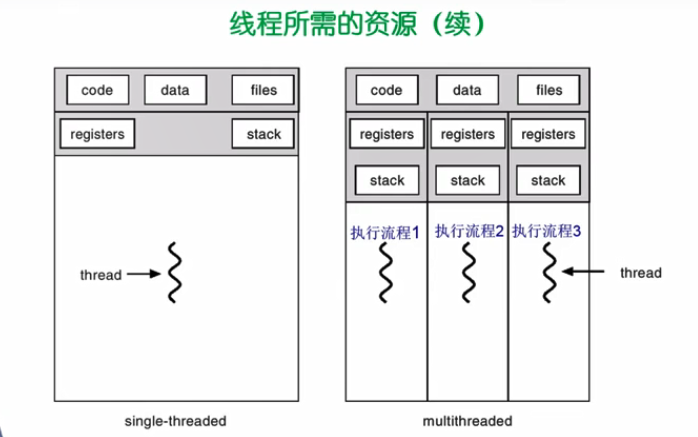
重新理解进程：

* 从资源组合的角度：进程把一组相关的资源组合起来，构成了一个资源平台，包括地址空间（代码段、数据段）、打开的文件等各种资源；
* 从运行的角度：代码在这个资源平台上的一条执行流程。

线程的优点：

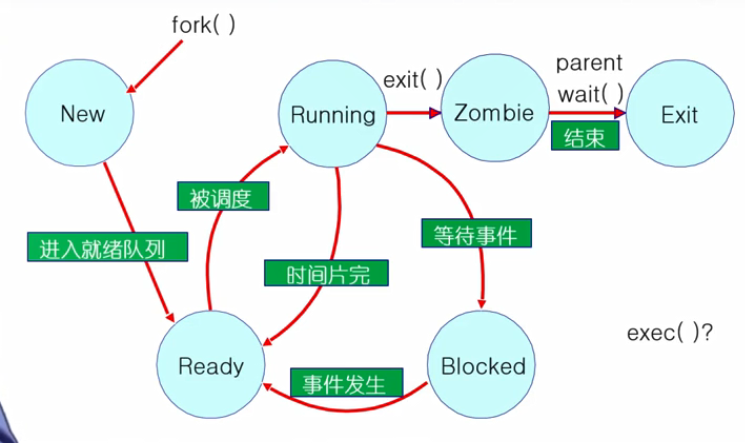
1. 一个进程中可以同时存在多个线程；
2. 各个线程之间可以并发执行；
3. 各个线程之间可以共享地址空间和文件等资源。

缺点：一个线程崩溃，会导致其所属进程的所有线程崩溃。



线程和进程的比较：

1. 进程时资源分配的单位，线程是CPU调度单位；
2. 进程拥有一个完整的资源平台，线程只独享必不可少的资源，如寄存器和栈；
3. 线程同样具有就绪、阻塞和执行三种状态，同样具有状态间的转换关系；
4. 线程能够减少并发执行的时间和空间开销：
   1. 线程的创建时间比进程短；
   2. 线程的终止时间比进程短；（不需要释放资源）
   3. 同一进程内的线程切换时间比进程短；（不需要切换地址空间和页表）
   4. 由于同一进程的各线程间共享内存和文件资源，可直接进行不通过内核的通信。



## 1.4. 进程与线程的区别

根本区别：进程是操作系统资源分配的基本单位，而线程是任务调度和执行的基本单位

1. **在开销方面**：每个进程都有独立的代码和数据空间（程序上下文），程序之间的切换会有较大的开销；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小。
2. **所处环境：**在操作系统中能同时运行多个进程（程序）；而在同一个进程（程序）中有多个线程同时执行（通过CPU调度，在每个时间片中只有一个线程执行）
3. **内存分配方面：**系统在运行的时候会为每个进程分配不同的内存空间；而对线程而言，除了CPU外，系统不会为线程分配内存（线程所使用的资源来自其所属进程的资源），线程组之间只能共享资源。
4. **包含关系：**没有线程的进程可以看做是单线程的，如果一个进程内有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的；线程是进程的一部分，所以线程也被称为轻权进程或者轻量级进程。

## 1.5. 协程

以多线程实现生产者/消费者模式为例，却并不是一个高性能的实现。为什么性能不高呢？原因如下：

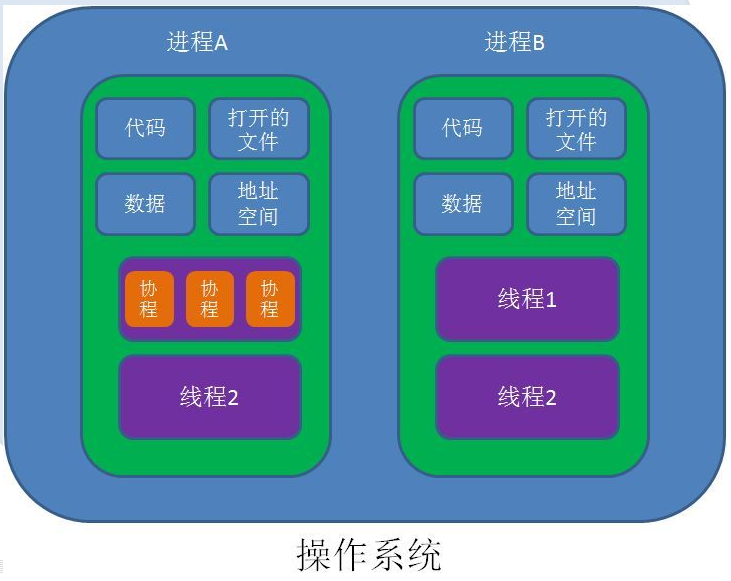
1.涉及到同步锁。

2.涉及到线程阻塞状态和可运行状态之间的切换。

3.涉及到线程上下文的切换。

以上涉及到的任何一点，都是非常耗费性能的操作。

协程，英文Coroutines，是**一种比线程更加轻量级的存在**。正如一个进程可以拥有多个线程一样，**一个线程也可以拥有多个协程**。



最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。**协程的暂停完全由程序控制**，线程的阻塞状态是由操作系统内核来进行切换。因此，协程的开销远远小于线程的开销。

# 2. 进程调度

## 2.1. 先来先服务

先来先服务(FCFS)调度算法是一种最简单的调度算法，该算法既可用于作业调度，也可用于进程调度。当在作业调度中采用该算法时，每次调度都是从后备作业队列中选择一个或多个最先进入该队列的作业，将它们调入内存，为它们分配资源、创建进程，然后放入就绪队列。在进程调度中采用FCFS算法时，则每次调度是从就绪队列中选择一个最先进入该队列的进程，为之分配处理机，使之投入运行。该进程一直运行到完成或发生某事件而阻塞后才放弃处理机。

## 短作业(进程)优先调度算法

短作业(进程)优先调度算法SJ(P)F，是指对短作业或短进程优先调度的算法。它们可以分别用于作业调度和进程调度。短作业优先(SJF)的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。而短进程优先(SPF)调度算法则是从就绪队列中选出一个估计运行时间最短的进程，将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时再重新调度。

## 2.3. 高优先权优先调度算法

为了照顾紧迫型作业，使之在进入系统后便获得优先处理，引入了最高优先权优先(FPF)调度算法。此算法常被用于批处理系统中，作为作业调度算法，也作为多种操作系统中的进程调度算法，还可用于实时系统中。当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程，这时，又可进一步把该算法分成如下两种。

### 2.3.1. 非抢占式优先权算法

  在这种方式下，系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程。这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。

### 2.3.2. 抢占式优先权调度算法

在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但在其执行期间，只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程)的执行，重新将处理机分配给新到的优先权最高的进程。因此，在采用这种调度算法时，是每当系统中出现一个新的就绪进程i 时，就将其优先权Pi与正在执行的进程j 的优先权Pj进行比较。如果Pi≤Pj，原进程Pj便继续执行；但如果是Pi>Pj，则立即停止Pj的执行，做进程切换，使i进程投入执行。显然，这种抢占式的优先权调度算法能更好地满足紧迫作业的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。

### 2.3.3. 优先级反转

优先级反转是指一个低优先级的任务持有一个被高优先级任务所需要的共享资源。高优先任务由于因资源缺乏而处于受阻状态，一直等到低优先级任务释放资源为止。而低优先级获得的CPU时间少，如果此时有优先级处于两者之间的任务，并且不需要那个共享资源，则该中优先级的任务反而超过这两个任务而获得CPU时间。如果高优先级等待资源时不是阻塞等待，而是忙循环，则可能永远无法获得资源，因为此时低优先级进程无法与高优先级进程争夺CPU时间，从而无法执行，进而无法释放资源，造成的后果就是高优先级任务无法获得资源而继续推进。

### 2.3.4. 优先级反转案例解释

不同优先级线程对共享资源的访问的同步机制。优先级为高和低的线程tall和线程low需要访问共享资源，优先级为中等的线程mid不访问该共享资源。当low正在访问共享资源时，tall等待该共享资源的互斥锁，但是此时low被mid抢先了，导致mid运行tall阻塞。即优先级低的线程mid运行，优先级高的tall被阻塞。

### 2.3.5. 优先级倒置解决方案

       (2.3.5.1）**设置优先级上限**：给临界区一个高优先级，进入临界区的进程都将获得这个高优先级，如果其他试图进入临界区的进程的优先级都低于这个高优先级，那么优先级反转就不会发生。

     （2.3.5.2）**优先级继承**：当一个高优先级进程等待一个低优先级进程持有的资源时，低优先级进程将暂时获得高优先级进程的优先级别，在释放共享资源后，低优先级进程回到原来的优先级别。嵌入式系统VxWorks就是采用这种策略。

## 2.4. 时间片轮转法

在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU分配给队首进程，并令其执行一个时间片。时间片的大小从几ms 到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。

## 2.5. 高响应比优先调度算法HRRN

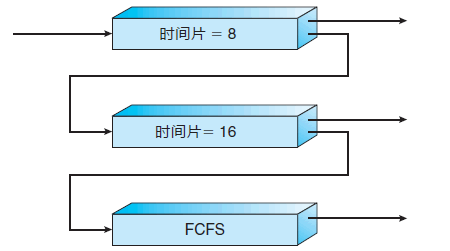
高响应比优先调度算法（Highest Response Ratio Next）是一种对CPU中央控制器响应比的分配的一种算法。HRRN是介于FCFS（先来先服务算法）与SJF（短作业优先算法）之间的折中算法，既考虑作业等待时间又考虑作业运行时间，既照顾短作业又不使长作业等待时间过长，改进了调度性能。

高响应比优先调度算法既考虑作业的执行时间也考虑作业的等待时间，综合了先来先服务和最短作业优先两种算法的特点。该算法中的响应比是指作业等待时间与运行比值：

响应比 =（等待时间+要求服务时间）/ 要求服务时间

## 2.6. **多级反馈队列调度算法**

前面介绍的各种用作进程调度的算法都有一定的局限性。如短进程优先的调度算法，仅照顾了短进程而忽略了长进程，而且如果并未指明进程的长度，则短进程优先和基于进程长度的抢占式调度算法都将无法使用。而多级反馈队列调度算法则不必事先知道各种进程所需的执行时间，而且还可以满足各种类型进程的需要，因而它是目前被公认的一种较好的进程调度算法。在采用多级反馈队列调度算法的系统中，调度算法的实施过程如下所述。



1. 设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第i+1个队列的时间片要比第i个队列的时间片长一倍。
2. 当一个新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按FCFS原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第n队列后，在第n 队列便采取按时间片轮转的方式运行。
3. 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第1～(i-1)队列均空时，才会调度第i队列中的进程运行。如果处理机正在第i队列中为某进程服务时，又有新进程进入优先权较高的队列(第1～(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第i队列的末尾，把处理机分配给新到的高优先权进程。

# 3. 并发性：互斥与同步

## 3.1. 并发的原理

多道程序设计系统中，进程被交替执行，表现出一种并发执行的外部特征。多道程序设计系统中，进程的相对执行速度不可预测，它取决于其他进程的活动、操作系统处理中断的方式及操作系统的调度策略。在全局资源共享、操作系统资源最优分配和定位程序设计错误方面引入了困难。

### 竞争条件(Race Condition)

**多个进程或者线程读写数据时，执行结果依赖于多个进程的指令执行顺序**。

### 原子操作

一次不存在任何中断或失败的执行。该执行成功结束或者根本没被执行，并且不应该发现任何部分执行的状态。

### 临界区

进程中的一段需要访问共享资源并且当另一个进程处于相应代码区域时不会被执行的代码区域。

### 互斥

当一个进程处于临界区并访问共享资源时，没有其他进程会处于临界区并且访问任何相同的共享资源。

### 死锁

两个或以上的进程，在相互等待完成特定任务，而最终没法将自身任务进行下去。

### 饥饿

一个可执行的进程被调度器持续忽略，以至于虽处于可执行状态却不被执行。

## 3.2. 临界区

特点：

1. 互斥：同一时间中，临界区中最多存在一个线程。
2. 前进：如果一个线程想进入临界区，那么它最终会成功。
3. 有限等待：如果一个线程i处于入口区，那么在i的请求被接受之前，其他线程进入临界区的时间时有限的。
4. 无忙等待：如果一个线程在等待进入临界区，那么它可以在进入之前被挂起。

### 3.2.1. 方法一：禁用硬件中断

为保证互斥，只需要保证一个进程不被中断即可。没有中断，即没有上下文切换，因此没有并发，硬件将中断处理延迟到中断被启用之后。进入临界区时关闭中断，离开临界区时开启中断。

### 3.2.2. 方法二：专用机器指令

硬件级别上，对存储单元的访问排斥对相同单元的其他访问。基于此点，处理器的设计者提供了一些机器指令，用于保证两个动作的原子性。在该指令执行过程中，任何其他指令访问内存将被阻止，而且这些动作在一个指令周期中完成。

1. **比较和交换指令：compare\_and\_swap**， 使用一个测试值testval检查一个内存单元，如果该内存单元的当前值是testval，就用newval取代该值；否则保持不变。该指令总是返回旧内存值，因此，如果返回值与测试值相同，则表示该内存单元已被更新。
2. **交换指令：exchange**，交换一个寄存器和一个存储器的内容。

机器指令方法优点：适用于在单处理器或共享内存的多处理器上的任何数目的进程；简单且易于证明；可用于支持多个临界区；

缺点：使用了忙等待，当一个进程等待进去临界区时，会耗费CPU资源；可能饥饿，因为选择哪个进程进入临界区是任意的；可能死锁。

### 3.2.3. 基于软件的解决方案

## 3.3. 信号量

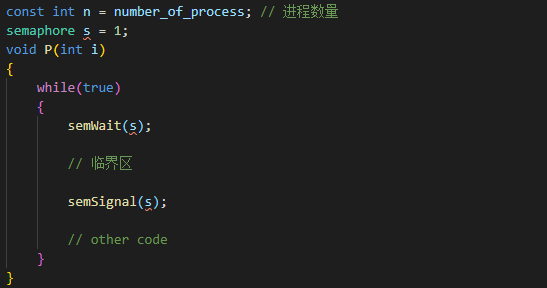
两个或多个进程可以通过简单的信号进行合作，一个进程可以被迫在某一位置停止，直到它接收到一个特定的信号。为了发信号，需要使用一个称为信号量的特殊变量。为通过信号量s传送信号，进程可执行原语semSignal(s)；为通过信号量s接收信号，进程可执行原语semWait(s)；如果相应的信号仍然没有发送，则进程被阻塞，直到发送完为止。

可把信号量视为一个具有整数值的变量，并定义三个操作：

1. 一个信号量可以初始化为非负数；
2. semWait操作使信号量减1，若值为负数，则执行semWait的进程被阻塞。否则进程继续执行；
3. semSignal操作使信号量加1，若值小于或等于0，则被semWait操作阻塞的进程被解除阻塞。

### 3.3.1. 互斥

使用二元信号量来实现互斥，信号量一般被初始化为1。进程进入临界区之前执行semWait(s)，若s为负，则进程被阻塞。



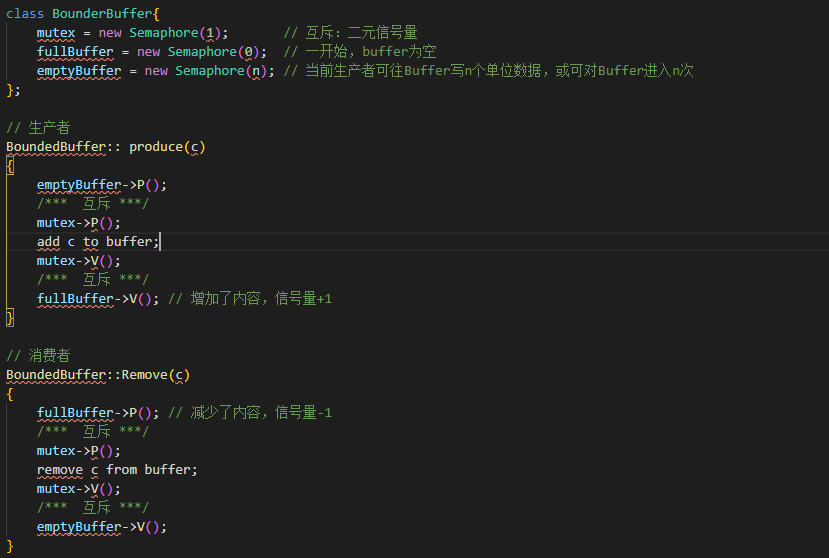
### 3.3.2. 生产者消费者

一个或多个生产者产生数据将数据放在一个缓冲区里；单个消费者每次从缓冲区中取出数据；在任何一个时间内只有一个生产者或小费制可以访问该缓冲区。

正确性要求：

1. 在任何一个时间内只有一个线程操作Buffer（互斥）； 🡪 二值信号量
2. 当Buffer为空，消费者必须等待生产者（同步）； 🡪 Buffer为空，一般信号量
3. 当Buffer为满，生产者必须等待消费者（同步）； 🡪 Buffer为满，一般信号量

针对上面每个约束，使用一个单独的信号量，如上所示。



## 3.4. 管程

管程是一个程序设计语言结构，它提供了与信号量同样的功能，但更易于控制。管程是一个或多个过程、一个初始化序列和局部数据组成的软件模块，其主要特点如下：

1. 局部数据变量只能被管程的过程访问，任何外部过程都不能访问；
2. 一个进程通过调用管程的一个过程进入管程；
3. 在任何时候，只有一个进程在管程中执行，调用管程的任何其他进程都被阻塞，以等待管程可用。

因此，可以把共享数据结构放在管程中，由管程提供的互斥机制进行保护。为进行并发处理，管程通过使用条件变量提供对同步的支持，这些条件变量包含在管程中，并且只有在管程中才能被访问。有两个函数可以操作条件变量：

1. cwait(c)：调用进程的执行在条件c上阻塞，管程现在可被另一个进程使用；
2. csignal(c)：恢复执行在cwait后因为某些条件而阻塞的进程。如果有多个这样的进程，选择其中一个；如果没有这样的进程，什么也不做。🡪 如果没有进程在条件x上等待，则csignal(x)的执行不会产生任何效果。

## 3.5. 消息传递

# 4. 死锁

## 4.1. 死锁原理

可以把死锁定义为一组相互竞争系统资源或进行通信的进程间的永久阻塞。当一组进程中的每个进程都在等待某个事件（典型的情况是等待所请求的资源被释放），而只有在这组进程中的其他被阻塞的进程才可以触发该事件，这时就称这组进程发生死锁。

### 4.1.1. 死锁条件

1. **互斥：**依次只有一个进程可以使用一个资源，其他进程不能访问已分配给其他进程的资源；
2. **占有且等待：**当一个进程等待其他进程时，继续占有已经分配的资源；
3. **不可抢占：**不能强行抢占进程已占有的资源；
4. **循环等待：**存在一个封闭的进程链，使得每个进程至少占有此链中下一个进程所需要的一个资源。

前三个是死锁存在的**必要条件**，而不是充分条件，死锁产生后一定有前3个现象。4个条件一起，称为死锁的充分必要条件。

## 4.2. 死锁预防

死锁预防策略是试图设计一种系统来排除发生死锁的可能性，防止前面列出的4个条件中的任何一个发生。

### 4.2.1. 互斥

不可能禁止这个条件，如果要对资源进行互斥访问，则操作系统必须支持互斥。

### 4.2.2. 占有且等待

可以要求进程一次性地请求所有需要的资源，并且阻塞这个进程直到所有请求都同时满足。该方法是低效的：

1. 进程可能被阻塞很长时间，以等待所有请求被满足；
2. 分配给一个进程的资源可能有相当长的一段时间不会被使用，但在此期间它们又不能被其他进程使用；
3. 进程可能实现并不知道它所需要的所有资源。

### 4.2.3. 不可抢占

两种方法：

1. 如果一个进程请求当前被另一个进程占有的一个资源，则操作系统可以抢占另一个进程，要求它释放资源。
2. 如果占有某些资源的一个进程进一步申请资源时被拒绝，则该进程必须释放它最初占有的资源。

只有在资源状态可以很容易地保存和恢复的情况下，这种方法才是实用的。

### 3.2.4. 循环等待

可以通过定义资源类型的线性访问顺序来预防。如果一个进程已经分配到了R类型的资源，那么它接下来请求的资源只能是那些排在R类型之后的资源类型。

## 4.3. 死锁避免

通过约束资源请求，使得4个死锁条件中至少有一个被破坏。死锁避免需要知道将来的进程资源请求的情况。两种方法：

1. **进程启动拒绝：**如果一个进程的请求会导致死锁，则不启动进程；
2. **资源分配拒绝：**如果一个进程增加的资源请求会导致死锁，则不允许此分配。此即银行家算法

死锁避免的优点：不需要死锁预防中的抢占和回滚进程，并且比死锁预防的限制少。

限制：

1. 必须事先声明每个进程请求的最大资源；
2. 所讨论的进程必须是无关的，即他们的执行顺序没有任何同步的限制；
3. 分配的资源数目必须是固定的；
4. 在占有资源时，进程不能退出。

## 4.4. 死锁检测