OS1-6

第三章概念

- 并发：指一组在逻辑上互相独立的程序或程序段在执行过程中，其执行时间在宏观上互相重叠，一个程序段的执行尚未结束，另一个程序段的执行已经开始的这种执行方式。
- 程序并发的特征
  - 间断性
  - 失去封闭性
  - 不可再见性
- 进程的特性
  - 动态性：有一定的生命周期
  - 并发行：多个进程实体，同时存在于内存中，能在一段时间内（不一定是同一时刻）同时运行
  - 独立性：进程实体是一个能独立运行的基本单位，同时也是系统中独立获得资源和独立调度的基本单位
  - 异步性：进程按各自独立的、不可预知的速度向前推进，即进程按异步方式运行
  - 结构特性：进程实体是由程序段、数据段及进程控制块等部分组成——进程映像(process image)
- 进程控制块
  - 作用
    - 用来描述进程
    - 用于进程的管理与调度
  - 构成
    - 标识符（identifier）——唯一标识进程
    - 状态（state）——进程的当前状态（运行/就绪/等待）
    - 优先级（priority）——相对于其他进程的优先级别
    - 程序计数器（PC = Program Counter）——即将被执行的下一条程序指令的地址
    - 内存指针（memory pointers）——包括指向程序代码、相关数据和共享内存的指针
    - 上下文数据（context data）——进程被中断时处理器寄存器中的数据
    - I/O状态信息（I/O status information）——包括显式I/O请求、分配给进程的

I/O设备、被解除使用的文件列表等
        - 记帐信息（accounting information）——包括占用处理器时间、时钟数总和、时间限制、账号等

- 进程状态
    - 轨迹：进程执行的指令序列
    - 进程的创建
        - 进程创建的过程
            - 给新进程分配一个唯一的进程标识号
            - 给进程分配空间
            - 初始化进程控制块
            - 设置正确的链接
            - 创建或扩充其他数据结构（如审计文件）
        - 进程创建的原因
            - 新的批处理作业——运行新程序
            - 交互登录——终端用户登录到系统（多用户操作系统）
            - OS提供服务——操作系统为用户创建进程以提供服务（如控制打印机）
            - 进程派生——由现有进程创建若干新进程（可提高模块化和开发并发性）
        - 进程派生（process spawning，产卵/大量生产）——指OS为一进程的显式请求创建新进程。如打印服务器进程为每一个打印请求产生一个新进程
    - 进程的终止
        - 批作业Halt指令或终止服务调用
        - 交互式用户退出系统，关闭终端
        - 用户结束一个应用程序
        - 错误和故障：（运行/等待）超时、内存不足、越界、保护错误、算术错误、I/O失败、无效指令、特权指令、数据误用
        - 进程撤销的过程
            - 撤销该进程的所有子进程
            - 收回进程所占用的资源
            - 撤销该进程的PCB
    - 挂起态
        - 交换：进程映像整体地或部分地从主存转移到辅存中（换出），或者从辅存中转移到内存中。
        - 引入交换的原因；
            - 实存不足：没有使用虚存的系统中，多个进程完全进入主存
            - CPU时间浪费：I/O 速度比计算速度慢很多 => 可能出现主存中的多个进程

全部阻塞等待 I/O
        - 调度策略：其他作业因没有主存空间不能投入运行
    - 挂起态的特点
        - 处于挂起态的进程不能立即执行
        - 被挂起的进程可能正在等待一个事件
        - 可通过代理来挂起进程（以阻止进程执行）
        - 被代理挂起的进程，只能通过代理显式地命令系统，才能进行状态转换
    - 引起挂起的原因
        - 交换（释放内存空间）
        - 交互式用户请求（为了调试等）
        - 定时（如周期性执行的进程）
        - 父进程请求（为检查、修改、协调等）
        - 其他OS原因（如后台进程、问题进程）
- 进程的描述
    - 进程的物理表示：进程映像（Process Image）
        - 用户数据
        - 用户程序
        - 系统栈（跟踪过程调用和过程间参数传递）
        - 进程控制块（Process Control Block，PCB)——由OS维护的用于记录和控制进程属性的集合
    - 进程的属性（在PCB中）
        - 进程标识信息
            - 进程标识符（process ID）
            - 父进程标识符
            - 用户标识符（user ID）
        - 处理器状态信息
            - 用户可见寄存器
            - 控制和状态寄存器
            - 栈指针（指向栈顶）
        - 进程控制信息
            - 调度和状态信息（进程状态，优先级，相关调度信息，等待的事件）
            - 数据结构（链接到队列、环或其他结构的信息）
            - 进程间通信
            - 进程特权
            - 存储管理（该进程虚存空间的指针）
            - 资源所有权和使用情况

- 进程控制块的组织
    - 链表
        - 同一状态的进程其PCB成一链表，多个状态对应多个不同的链表
        - 各状态的进程形成不同的链表：就绪链表、阻塞链表
    - 索引（index）表
        - 同一状态的进程归入一个索引表（由index指向PCB），多个状态对应多个不同的索引表
        - 各状态的进程形成不同的索引表：如就绪索引表、阻塞索引表等
- 进程控制块的作用
    - PCB = Process Control Block（进程控制块）
    - PCB是OS中最重要的数据结构，涉及进程调度、资源分配、中断处理、性能监控和分析等
    - PCB的访问与保护：通过专门的例程访问PCB
    - 资源控制块的集合定义了OS的状态
- 进程控制
    - 进程控制的功能：完成进程的创建、撤销以及进程的状态转换（进程切换/调度）
    - 进程控制由原语（primitive）完成
- 原语
    - 原语（primitive）：由若干条指令构成、在系统模式下执行，用于完成一定功能的一个过程
    - 原语是一种广义指令，相当于扩充了的机器指令集
    - 原语是原子操作(atomic operation)，即
    - 一个操作中的所有动作，要么全做，要么全不做
    - 原子操作是一个不可分割的操作!
- 执行模式
    - 两类指令
        - 特权指令（privileged instruction）：允许操作系统使用，不允许一般用户使用
        - 非特权指令（nonprivileged instruction）：操作系统和用户均可用的

- 两种执行模式（CPU状态）
  - 用户模式(user mode)/用户态（user state）/目态（target state，目标状态）：只能执行非特权指令；用户程序在用户模式下运行（在Intel x86 CPU中对应于保护模式下的特权级[Privilege Level]1~3）
  - 系统模式(system mode)/系统态（system state）/内核模式（kernal mode）/内核态（kernal state）/特权模式（privileged mode）/管态（supervisory state，监管状态）：能执行指令全集，具有改变CPU执行状态的能力；操作系统（内核）在系统模式下运行（在Intel x86 CPU中对应于保护模式下的特权级0）
- 内核
  - OS中包含重要系统功能的部分，通常驻留主存，在系统模式下运行，响应来自进程的调用（系统功能调用）和来自设备的中断
  - 内核的典型功能
    - 进程管理：进程的创建、撤销、调度、切换、同步和通信以及PCB的管理等
    - 存储管理：给进程分配空间、交换、管理页和段
    - I/O管理：缓冲区管理、给进程分配I/O通道和设备
    - 支持功能：中断处理、审计、监视
- 两种模式的相互切换
  - 用户模式→系统模式：唯一途径是通过中断机制（在x86 CPU 中，具体可通过使用调用门指令CALL进行代码转移来实现）
  - 系统模式→用户模式：可通过修改PSW实现，如指令CHM（x86 CPU中没有此指令，但可通过远程返回指令RETF进行代码转移来实现）
- 进程切换
  - 操作系统指定一个进程为运行态，并将CPU控制权交给该进程
  - 进程切换的时机
    - 当OS从正在运行的进程那里获得控制权时，可能进行进程切换
- 导致os获得控制权的事件
  - 中断
    - 时钟中断：时间片到
    - I/O中断：I/O完成，高优先级进程就绪
    - 内存失效：调页时阻塞（所需内存地址不在主存）
  - 陷阱（trap）/异常（exception）：当前执行的指令出现错误（主要指在处理器和内存内部产生的软中断，一般称为内中断）
  - 系统调用（如申请I/O操作）：用户进程会被置为阻塞态（系统调用一般是通过[由操作系统规定的]特定中断来实现，如DOS的21h号中断）

        - 总之，只有通过[软/硬或内/外]中断，操作系统才能获得控制权
    - 进程切换
        - cpu所做的工作
            - 保存当前正在执行的程序的上下文环境
            - 把程序计数器置成中断处理程序的开始地址
            - 从用户模式切换到内核模式，使得中断处理代码可以执行特权指令
        - 进程切换的步骤
            - 保护处理器上下文环境 (到哪里去?)
            - 更新当前处于运行态进程PCB的控制信息（修改状态）
            - 该进程PCB挂入相应队列 (会是哪个队列?)
            - 选择一个就绪进程（涉及多种不同算法）
            - 更新所选进程的PCB（包括状态）
            - 更新存储管理数据结构（涉及地址转换）
            - 恢复被选中进程的处理器上下文环境 (从哪里来?)
    - os的执行
        - 非进程内核(Non-process kernel)——传统方法
            - 进程概念仅适用于用户程序
            - OS代码是在特权模式下工作的独立实体
        - 在用户进程中执行(Execution Within User Processes)
            - OS是用户进程调用的一组例程，OS代码为所有进程映像共享
            - 执行OS代码时切换到系统模式（不需进程切换）
        - 基于进程的OS(Process-Based Operating System)
            - 主要内核功能被组织成独立进程
            - 适合多处理器和多机环境

              # 第四章概念

- 引入线程的原因
    - 进程切换的开销大——每次切换都要保存和恢复进程所拥有的全部信息(PCB、有关程序段和相应的数据集等)
    - 进程占用的资源多——多个同类进程需占用多份资源，而一个进程中的多个同类线程则共享一份资源
- 线程：一个进程内的基本调度单位
- 多线程：是指OS支持在一个进程中执行多个线程的能力
- 进程/任务：资源分配和保护的单位

    - 拥有用于保存进程映像的虚地址空间
    - 受保护地访问处理器、其他进程、文件和I/O资源

- 线程：分派的执行单位

    - 执行状态（运行、就绪等）
    - 保存的线程上下文（非运行时）
    - 一个执行栈
    - 独立的用来存储局部变量的静态存储空间
    - 对进程的内存和其他资源的访问（与同一进程内的其他线程共享这些资源）

- 线程的优点

    - 创建速度快（在已有进程内）
    - 终止所用的时间少
    - 切换时间少（保存和恢复工作量小）
    - 通信效率高（在同一进程内，无需调用OS内核，可利用共享的存储空间）

- 线程的应用

    - 若应用程序可按功能划分成不同的小段，或可划分成一组相关的执行实体，则用一组线程（比用一组进程）可提高执行效率（尤其是在多处理器和多核系统中）
    - 单用户多核/多处理器系统的典型应用：
    - 服务器中的文件管理或通信控制
    - 前台和后台操作（如前台与用户交互、后台更新数据）
    - 异步处理（如字处理与周期性备份）
    - 加速执行（在多核/多处理器系统中的并行）
    - 模块化程序结构（涉及多种活动或多个I/O源和目的地的程序）

- 线程的功能特性

    - 线程状态
        - 运行
        - 就绪
        - 阻塞
    - 线程不拥有资源，且进程与其所有线程共享代码和地址空间。但是线程拥有自己的寄存器上下文和栈空间（用来存储局部变量和调用参数）
    - 挂起状态、终止状态是进程级的概念
        - 挂起一个进程，则该进程的所有线程也挂起（共享地址空间）

        - 终止一个进程，则该进程的所有线程也终止（共享代码段）
    - 与线程状态变化有关的操作
        - 派生（spawn，产卵）——线程可派生新线程
        - 阻塞（block）——等待事件发生
        - 唤醒/解除阻塞（unblock）——事件发生后被唤醒，转换到就绪态
        - 调度（schedule）——由操作系统将就绪线程调度到处理器（核）中执行
        - 结束（finish）——释放寄存器上下文和栈
    - 线程同步
        - 需要对各个线程的活动进行同步，以便它们互不干涉且不破坏数据结构
        - 线程同步机制与进程同步机制相同
    - 线程分类
        - 用户级线程（ULT）
            - 线程管理均由应用程序完成
            - 内核不知道线程的存在
            - 优点
                - 线程切换不需要模式切换
                - 调度算法可应用程序专用
                - ULT不需内核支持，线程库可在任何OS上运行
            - 缺点
                - 一个线程阻塞会导致整个进程阻塞
                - 不能利用多核和多处理器技术
            - 实例
                - GNU Portable Threads
        - 内核级线程
            - 线程管理由内核完成
            - 调度基于线程完成
            - 优点
                - 线程阻塞不会导致进程阻塞
                - 可以利用多核和多处理器技术
                - 内核例程本身也可以使用多线程
            - 缺点
                - 线程切换需要模式切换
        - 组合方法
            - 线程创建在用户空间完成
            - 线程调度和同步在用户空间进行

- SMP
    - 多个处理器可以执行相同功能（故称“对称”），内核可运行在任一处理器上
    - 每个处理器可从可用进程和线程池完成自身的调度工作
    - 对称的多核是SMP的特例，是片上的SMP
- 分层内核（宏内核/单体内核）
    - 所有功能按层次组织
    - 只在相邻层之间发生交互
    - 缺点:
        - 某一层中的主要变化可能会对相邻层中的代码产生巨大影响
        - 相邻层之间有很多交互，很难保证安全性
- 微内核
    - 基本思想
        - 只有最基本OS功能放在内核中, 运行在内核模式
        - 不是最基本服务和应用在内核之外, 运行在用户模式
    - 特点
        - 一致接口
            - 所有服务都以消息的形式提供
        - 可扩展性(Extensibility)
            - 允许增加新的服务
        - 灵活性(Flexibility)
            - 可以增加新的功能、删除现有功能
        - 可移植性(Portability)
            - 把系统移植到新处理器上只需要对内核而不需要对其他服务修改
        - 可靠性(Reliability)
            - 模块化设计
            - 小的微内核可以被严格地测试
        - 分布式系统支持
            - 消息传送不需要知道目标机器的位置
        - 对面向对象操作系统(OOOS)的支持
            - 组件是具有明确定义接口的对象，可互连构造软件
        - 存储管理
            - 微内核控制硬件概念上的地址空间，使得操作系统可以在进程级实现保护
            - 内核只负责将每个虚页映射到一个物理页框上

        - 内核外实现页替换算法等分页工作
            - 虚页调入
            - 页换出

第五章 并发性：互斥和同步

- 并发->共享->RC->互斥
- **竞态条件**
    - 在并发环境中发生
    - 多个进程共享数据
    - 多个进程读取且至少一个进程写入
    - 共享数据的最终结果取决于**进程执行的相对速度**
- 竞争引发的控制问题
    - 互斥
    - 死锁
    - 饥饿
- 进程的互斥机制
    - 软件方法(Dekker方法，Peterson算法)
    - 硬件方法(关中断，专用指令)
    - 操作系统或者程序设计语言中提供某种级别的支持（信号量、管程、消息机制）
- 互斥相关的概念
    - 互斥（mutual exclusion）
        - 多个进程需要访问一个不可共享的资源时，任何时候只能有一个访问这个资源
    - 临界资源（critical resource）
        - 不可共享的资源
    - 临界区（critical section）
        - 访问临界资源的那部分代码
    - 死锁(deadlock)
        - 一组进程中，每个进程都无限等待该组进程中另一进程所占用的资源
    - 饥饿(starvation)
        - 一组进程中，某个或某些进程无限等待该组进程中其他进程所占用的资源
- 进程的交互

    - 进程之间互相不知道对方的存在
        - 竞争关系
        - 多个独立进程的多道程序设计，尽管不会一起工作，但是他们可能都想访问同一个磁盘、文件或打印机。
    - 进程间接知道对方的存在
        - 通过共享合作
        - 彼此之间不知道对方的进程ID，但可以共享某些对象，比如I/O缓冲区。
    - 进程直接知道对方的存在
        - 通过通信合作
        - 彼此之间知道对方的进程ID

- 进程间的竞争现象

    - 进程在不知道其他进程存在情况下共享资源

- 进程间通过共享的合作

    - 特点
        - 没有意识到其他进程的存在，但知道要维护数据的完整性
        - 共享变量、文件或数据库等
    - 相互间产生的影响：
        - 执行结果可能会受影响
        - 执行时间受影响
    - 共享引发的控制问题
        - 互斥
        - 死锁
        - 饥饿
        - 数据的一致性

- 进程间通过通信的合作

    - 特点
        - 进程直接知道合作伙伴
        - 采用消息传送的方式通信（发送/接收消息）
    - 相互间产生的影响：同“通过共享的合作”
    - 引发的控制问题:
        - 死锁
        - 饥饿

- 互斥的要求
    - 在具有相同资源或共享对象的临界区的所有进程中，一次只允许一个进程进入临界区（强制排它）
    - 一个在非临界区停止的进程必须不干涉其他进程（充分并发）
    - 没有进程在临界区中时，任何需要访问临界区的进程必须能够立即进入（空闲让进）
    - 决不允许出现一个需要访问临界区的进程被无限延迟（有限等待）
    - 相关进程的执行速度和处理机数目没有任何要求或限制（满足异步）
    - 当进程不能进入临界区，应该立即释放处理机，防止进程忙等待（让权等待）
- 信号量
    - 作用：由OS提供的用于并发控制的一种特殊数据结构
    - 组成
        - 一个整数变量
        - P操作和V操作
    - P操作
        - 信号量的值减1
        - 若信号量的值变成负数，则执行P操作的进程被阻塞
    - V操作
        - 信号量的值加1
        - 如果信号量的值不是正数（其绝对值=现被阻塞的进程数[等待队列的长度]），则使一个因执行P操作被阻塞的进程解除阻塞（唤醒）
    - 信号量的实际含义
        - 信号量的初值表示的是某类资源的数目
        - 如果信号量的值为负数的话，表示的是在队列上等待的进程数。
    - 优缺点
        - 优点
            - 简单
            - 可以解决多种类型的同步、互斥问题
        - 缺点
            - 不够安全，可能会产生死锁
            - 遇到复杂同步互斥问题时实现复杂。
- 为什么信号能能够实现互斥问题?
    - n个进程访问同一个共享资源（临界资源）

- 将信号量s初始化为1
- 每个进程进入临界区之前执行P操作
  - s--
  - 若s>=0，则进程进入临界区
  - 若s<0，则进程被阻塞不能进入临界区，加入等待队列
- 进程离开临界区时执行V操作
  - s++
  - 若s<=0，则唤醒一个被阻塞的进程，将其移出等待队列，置为就绪状态，使其在下次操作系统调度时可进入临界区
  - 这样可以保证最多只有一个进程在临界区，从而实现了共享资源的互斥访问

- 消息传递的同步
  - 阻塞send阻塞receive：发送者和接受者都被阻塞，直到完成信息的传递
  - 无阻塞send阻塞receive
  - 无阻塞send无阻塞receive
    # 附页--互斥的代码
    ##1.硬件语言实现互斥

- TS指令
  即比较M的某一个标志位，如果该标志位为0，则表示没有使用这个锁，可以使用，如果为1，则不能访问CS。

```
boolean testset (int i)
{
    if (i == 0) {
            i = l;
            return true;
    } else
            return false;
}
```

```
M:byte 0
lock:
    TS [M],0//M的第零位,M[0]为1则循环
    jz lock
    ret
unlock:
    mov M[0],0
```

```
Version 2
enter_region:
TSL REGISTER, LOCK                    /*复制锁到寄存器并将锁置1*/
CMP REGISTER, #0                      /*判断寄存器内容是否为0*/
JNE enter_region                      /*若不是0跳转到enter_region*/
RET                                   /*返回调用者，进入临界区*/

leave_region:
MOVE LOCK, #0                         /*在锁中置0*/
RET                                   /*返回调用者*/
```

- XCHG

```
[M] word 0
lock:
    move ax,1
    XCHG [M],ax
    cmp ax,1
    jz lock
    ret
unlock:
    mov [M],0
```

```
Version 2
enter_region:
MOVE REGISTER, #1                     /*给寄存器中置1*/
XCHG REGISTER, LOCK                   /*交换寄存器与锁变量的内容*/
CMP REGISTER, #0                      /*判断寄存器内容是否为0? */
JNE enter_region                      /*若不是0跳转到enter_region*/
RET                                   /*返回调用者，进入临界区*/
leave_region:
MOVE LOCK, #0                         /*在锁中置0*/
RET                                   /*返回调用者*/
```

缺点:
忙等待
饥饿
死锁

2.软件方法--四种尝试

- 第一种尝试--turn来轮流

```
P0:
    while(turn != 0);


    critical section
    turn = 1;
P1:
    while(turn != 1);


    critical section
    turn = 0;
```

问题：虽然保证了互斥访问资源，但是硬性规定进入的顺序。不能保证“空闲让进”准则。

- 第二种尝试

```
flag[]={0,0}
P0:
while(flag[1]);

flag[0]=1;
critical section
flag[0]=0;

P1:
while(flag[0]);

flag[1]=1;
critical section
flag[1]=0;
```

问题：连最基本的互斥都不能满足

- 第三种尝试

```
flag[]={0,0}
P0:
    flag[0]=1;
    while(flag[1]);
    critical section
    flag[0]=0;
P1:
    flag[1]=1;
    while(flag[0]);
    critical section
    flag[1]=0;
```

问题：可能会导致死锁。（会画时序图！）

- 第四种尝试

```
flag[]={0,0}
P0:
    flag[0]=1;
    while(flag[1])
    {
        flag[0]=0;
        delay();
        flag[0]=1;
    }
    critical section
    flag[0]=0;
P1:
    flag[1]=1;
    while(flag[0])
    {
        flag[1]=0;
        delay();
        flag[1]=1;
    }
    critical section
    flag[1]=0;
```

问题：可能会导致活锁。

3.软件方法--正确方法

- Dekker方法

```
boolean flag[2];
int turn;
void P0() {
    while (true) {
        flag[0] = true; // 自己想进临界区
        // flag[1]==false时进入临界区
        while (flag[1]) { // flag[1]==true时等待
            if (turn == 1) { // turn==1时礼让
                flag[0] = false;
                while (turn == 1) /* do nothing */;
                flag[0] = true;
            }
        }
        /* critical section */
        turn = 1;
        flag[0] = false;
        /* remainder */
    }
}
```

- 存在的问题：
  - 逻辑复杂
  - 正确性难证明
  - 存在轮流问题
  - 存在忙等待

- Peterson算法

```
boolean flag [2];
int turn;
void P0() {
while (true) {
    flag [0] = true;
    turn = 1;
    while (flag [1] && turn == 1)
```

```
        /* do nothing */;
    /* critical section */
    flag [0] = false;
    /* remainder */
}
}
```

4.信号量

- 整型信号量

```
struct semaphore{ //定义结构
int count;
queueType *queue;
} s;
void  P(semaphore s) { // P操作
s.count--;
if (s.count<0)
   Block(CurruntProcess, s.queue);
}
}
void  V(semaphore s) {// V操作
s.count++;
if (s.count<=0)
   WakeUp(s.queue);
}
```

- 二元信号量

```
struct semaphore{
    bool count;
    queueType *queue;
 } s;

 void P(semaphore s) {
    if (s.count==1)  s.count=0;
    else  Block(CurruntProcess, s.queue);
 }

void  semSignalB(semaphore s) {
```

```
    if (s.queue is empty) s.count==1;
    else WakeUp(s.queue);
}
```

5.生产消费者问题

- Version 1

```
int n; /*缓冲区中产品数*/
Binary_semaphore s=1; /*互斥*/
Binary_semaphore delay=0; /*等待*/
void producer()  {
  while (true)  {
    produce();
    semWaitB(s);
    append();
    n++;
    if (n==1)  semSignalB(delay);
    semSignalB(s);
  }
}

void consumer()  {
  semWaitB(delay);
  while (true)  {
    semWaitB(s);
    take();
    n--;
    semSignalB(s);
    consume();
    if (n==0)  semWaitB(delay);
  }          少执行一次
}
void main()  {
  n=0;
  parbegin(producer, consumer) ;
}
```

备注：这个方案存在超前消费，比如消费者在consume（）之后进行切换。

- Version 2

```
int n;
Binary_semaphore s=1;
Binary_semaphore delay=0;
void producer()  {
  while (true)  {
    produce();
    semWaitB(s);
    append();
    n++;
    if (n==1)  semSignalB(delay);
    semSignalB(s);
   }
}
void consumer()  {
  int m;
  semWaitB(delay);
  while (true)  {
    semWaitB(s);
    take();
    n--;
    m=n;
    semSignalB(s);
    consume();
    if (m==0)  semWaitB(delay);
  }
}
```

- Version 3

```
semaphore n=0; /*缓冲区中的产品数*/
semaphore s=1; /*互斥*/
void producer()  {
  while (true)  {
    produce();
    semWait(s);
    append();
    semSignal(s);
    semSignal(n);
  }
}
```

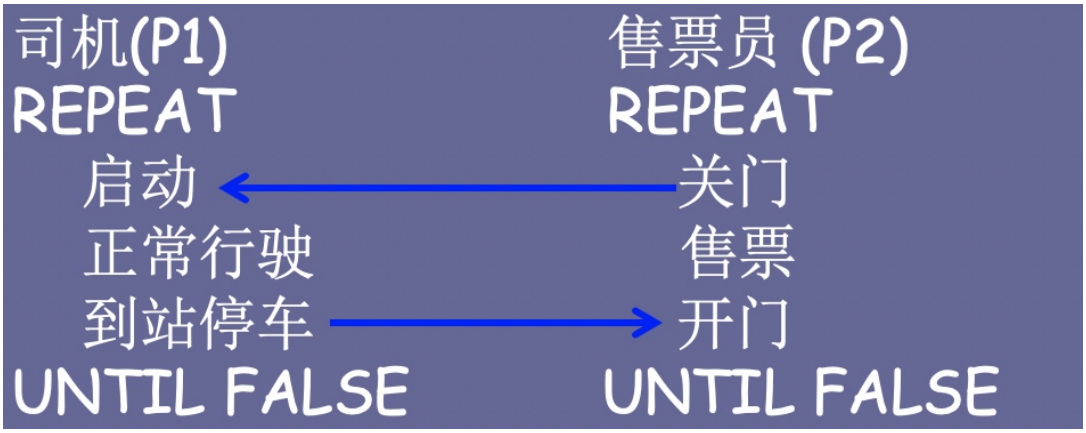
```
void consumer()  {
  while (true)  {
    semWait(n);
    semWait(s);
    take();
    semSignal(s);
    consume();
  }
}
void main()  {
   parbegin(producer, consumer);
}
```

如果把V(n)和V(s)之间颠倒顺序，就会产生死锁。

- 信号量的实现
    - 用TestSet指令实现

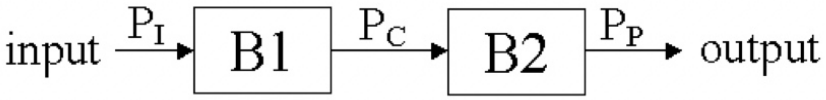
```
testset是1的话就返回0，否则置1返回1
void  semWaitT(semaphore s) {
   while (!testset(s.flag));
   s.count--;
   if (s.count<0)  Block(CurruntProcess, s.queue);
   s.flag=0;
 }
void  semSignalT(semaphore s) {
   while (!testset(s.flag));
   s.count++;
   if (s.count<=0)  WakeUp(s.queue);
   s.flag=0;
}
```

- 用信号量实现同步的两个例子
    - 司机售票员




```
semaphore s1=0,s2=0
P1:
{
    P(s1);
    启动
    正常行驶
    到站停车
    V(s2)
}
P2:
{
    关门
    V(S1)
    售票
    P(S2)
    开门
}
```

- 打印区的问题
  - 进程PI将输入的数据写入缓冲区B1，
  - 进程PC读出B1中的数据，完成计算，把结果写入缓冲区B2
  - 进程PP读出B2中的结果，打印输出
  - (1)先写后读（不能读空缓冲区）
  - (2)未读完不能写（不能写非空缓冲区）




```
    P1 :
while ( 1 ) {
P(empty1) ;
输入数据写到B1;
V(full1);
}
PC :
while ( 1 ) {
P(full1);
从B1中读取数据;
V(empty1);
计算;
P(empty2);
结果写到B2;
V(full2);
}
PP:
while ( 1 ) {
P(full2);
读取B2中的结果并输出到打印机;
V(empty2);
}
```

- 生产消费者问题
  - 无限缓冲区
    - 第一种尝试，基于二元信号量
      问题：如果在consume之后进行切换，会出现超前消费的情况。

```
int n; /*缓冲区中产品数*/
Binary_semaphore s=1; /*互斥*/
Binary_semaphore delay=0; /*等待*/
void producer()  {
while (true)  {
```

```
produce();
semWaitB(s);
append();
n++;
if (n==1)  semSignalB(delay);
semSignalB(s);
}
}
void consumer()  {
semWaitB(delay);
while (true)  {
semWaitB(s);
take();
n--;
semSignalB(s);
consume();
if (n==0)  semWaitB(delay);
}          少执行一次
}
void main()  {
n=0;
parbegin(producer, consumer) ;
}
```

- 正确的解法，基于二元信号量

```
        int n;
Binary_semaphore s=1;
Binary_semaphore delay=0;
void producer()  {
while (true)  {
produce();
semWaitB(s);
append();
n++;
if (n==1)  semSignalB(delay);
semSignalB(s);
}
}
void consumer()  {
  int m;
  semWaitB(delay);
```

```
  while (true)  {
    semWaitB(s);
    take();
    n--;
    m=n;
    semSignalB(s);
    consume();
    if (m==0)  semWaitB(delay);
  }
}
```

- 正确解法--基于计数信号量
  consumer的`semWait(n); semWait(s);`不能颠倒顺序，否则会产生死锁

```
semaphore n=0; /*缓冲区中的产品数*/
semaphore s=1; /*互斥*/
void producer()  {
  while (true)  {
    produce();
    semWait(s);
    append();
    semSignal(s);
    semSignal(n);
  }
}
      void consumer()  {
  while (true)  {
    semWait(n);
    semWait(s);
    take();
    semSignal(s);
    consume();
  }
}
void main()  {
   parbegin(producer, consumer);
}
```

- 有限缓冲区

```
const int sizebuffer=N
```

```
semaphore n=0; /*产品数*/
semaphore s=1;  /*互斥*/
semaphore e=N; /*空闲数*/
void producer()  {
  while (true)  {
    produce();
    semWait(e);
    semWait(s);
    append();
    semSignal(s);
    semSignal(n);
  }
}
void consumer()  {
  while (true)  {
    semWait(n);
    semWait(s);
    take();
    semSignal(s);
    semSignal(e);
    consume();
  }
}
void main()  {
   parbegin(producer, consumer);
}
```

- 管程
  - 管程中的数据每次只能被一个进程访问
  - 可将共享数据结构放入管程以得到保护
  - 可用这些数据代表临界资源
  - 利用管程解决生产者消费者问题

```
monitor boundedbuffer  { /*管程体*/
char buffer[N];
int nextin, nextout;          /*局部数据*/
int count; /*产品数*/
cond notfull, notempty; /*条件变量*/
void append(char x)  { /*添加过程*/
```

```
    if (count==N)  cwait(notfull);
    buffer[nextin]=x;
    nextin=(nextin+1) % N;
    count++;
    csignal(notempty);
}
void take(char x)  {/*取出过程*/
    if (count==0)  cwait(notempty);
    x=buffer[nextout];
    nextout=(nextout+1) % N;
count--;
csignal(notfull);
}
{nextin=0; nextout=0; count=0;} /*初始化*/
}
void producer() {
char x; /*产品=字符*/
while(true) { produce(x);  append(x); }
}
void comsumer() {      管程过程调用
char x;
while(true) { take(x);  comsume(x); }
}
void main() {
parbegin(producer, consumer);
}
```

- 要求在条件队列中至少有一个进程
- 当另一进程为该条件产生csignal信号时，则产生此信号的进程必须立即退出管程（或阻塞在管程上），以便让队列中被唤醒的进程能够立即进入管程运行
- 如果产生csignal信号的进程在管程内的运行还未结束，则需要两次额外的进程切换（阻塞以让被唤醒的进程运行，等其运行结束后再恢复运行）
- 进程调度程序必须保证在激活被唤醒的进程前没有其他进程进入管程，否则可能造成永久阻塞

- 通知类的管程

```
monitor boundedbuffer {
char buffer[N];
int nextin, nextout;
int count;
```

```
cond notfull, notempty;
void append(char x)  {
while (count==N)  cwait(notfull);
buffer[nextin]=x;
nextin=(nextin+1) % N;
count++;
cnotify(notempty);
}
void take(char x)  {
while (count==0)  cwait(notempty);
x=buffer[nextout];
nextout=(nextout+1) % N;
count--;
cnotify(notfull);
}
{nextin=0; nextout=0; count=0;}
}
void producer() {
char x;
while(true) { produce(x);  append(x); }
}
void comsumer() {
char x;
while(true) { take(x);  comsume(x); }
}
void main() {
parbegin(producer, consumer);
}
```

  - 用cnotify原语代替原来的csignal操作（发通知的进程不需立即退出管程，接到通知的进程也不立即被唤醒，只是转为就绪，等待合适的时候再进入管程运行）
  - 用while循环代替if判断（有额外的条件变量检查，但可避免额外的两次进程切换）等待条件增加计时器，等待超时的进程将被转为就绪
  - cnotify原语.通知特定等待条件队列中的第一个等待进程，但当前执行cnotify原语的进程继续执行被通知的等待进程转为就绪，但必须重新检查条件。
  - cbroadcast原语.通知特定等待条件队列中的所有等待进程，但当前执行cnotify原语的进程继续执行

- 消息传递实现生产者-消费者问题

```
 /* 互斥程序 */
const int capacity=/*缓冲区容量*/;
message null=/*空消息*/;
void producer() {
message pmsg;
while(true)    {
  receive(mayproduce, pmsg);
  pmsg=produce();
  send(maycomsume, pmsg);
}
}
void comsumer() {
message cmsg;
message cmsg;
while(true)    {
  receive(mayconsume, cmsg);
  consume(cmsg);
  send(mayproduce, null);
}
}
void main() {
create_mailbox(mayproduce);
create_mailbox(mayconsume);
for(int i=1; i<capacity; i++)
 send(mayproduce, null);
parbegin(producer(), consumer());
} /*初始时：mayconsume信箱为空，mayproduce信箱则被null信号填满*/
```

- 读者-写者问题

- 读者优先

```
/* program reader_and_writer */
int readcount;
semaphore x=1, wsem=1;
void reader() {
while(true) {
  P(x);
  readcount++;
  if (readcount==1) P(wsem);
  V(x);
```

```
  READUNIT();
  P(x);
  readcount--;
  if (readcount==0) V(wsem);
  V(x);
}
}
void writer() {
while(true) {
  P(wsem);
  WRITEUNIT();
  V(wsem);
}
}
void main() {
readcount=0;
parbegin(reader(), writer());
}
```

信号量wsem用于互斥
信号量x用于保证readcount被正确更新
色块为临界区
写进程可能处于饥饿状态

- 写者优先

```
/* program reader_and_writer */
int readcount, writecount;
semaphore x=1, y=1, z=1, rsem=1, wsem=1;
void reader() {
while(true) {
  P(z); P(rsem);
 P(x);
  readcount++;
  if (readcount==1) P(wsem);
  V(x);
 V(rsem); V(z);
 READUNIT();
  P(x);
  readcount--;
  if (readcount==0) V(wsem);
  V(x);
}
```

```
void writer() {
while(true) {
  P(y);
  writecount++;
  if (writecount==1) P(rsem);
  V(y);
  P(wsem);
  WRITEUNIT();
  V(wsem);
  P(y);
  writecount--;
  if (writecount==0) V(rsem);
  V(y);
}
}
void main() {
readcount = writecount = 0;
parbegin(reader(), writer());
}
```

为保证写进程优先进入，只允许一个读进程在rsem上排队，信号量z用于其余读进程排队
信号量rsem用于在写数据区时禁止读进程进入
信号量y则用于保证writecount更新的正确性

- 读写公平

```
semophere file_access=1,x=1,queue=1
void reader()
{
while(1)
{
    p(queue);
    p(x);
    if(readcount==0)
    p(file_access)
    readcount++;
    v(x)
    v(queue)
    read();
    p(x);
    readcount--;
    if(readcount==0);
    v(file_access);
    v(x)
```

```
}
}
void writer()
{
while(1)
{
    p(queue)
    p(file_access)
    v(queue)
    write()
    v(file_access)
}
```

}

第六章

6.1死锁原理

- 死锁的概念：一个进程集合中的每个进程都在等待只能由该集合中的其他一个进程才能引发的事件（释放占有资源/进行某项操作）
- 联合进程图
- 资源分为两类
    - 可重用资源：一次仅供一个进程使用并且不因使用而耗尽的资源（处理器、I/O通道、主存和辅存、设备，文件、数据库、信号量等数据结构）。可重用资源会导致死锁。（内存，磁盘）
        - 例子：$p_0 p_1 q_0 q_1 p_2 q_2$

步骤 Step	进程P操作 Process P Action
p_0	Request (D) 请求
p_1	Lock (D) 加锁
p_2	Request (T) 请求
p_3	Lock (T) 加锁
p_4	Perform function
p_5	Unlock (D) 解锁
p_6	Unlock (T) 解锁

D：磁盘

T：磁带

执行功能

步骤 Step	进程Q操作 Process Q Action
q_0	Request (T) 请求
q_1	Lock (T) 加锁
q_2	Request (D) 请求
q_3	Lock (D) 加锁
q_4	Perform function
q_5	Unlock (T) 解锁
q_6	Unlock (D) 解锁

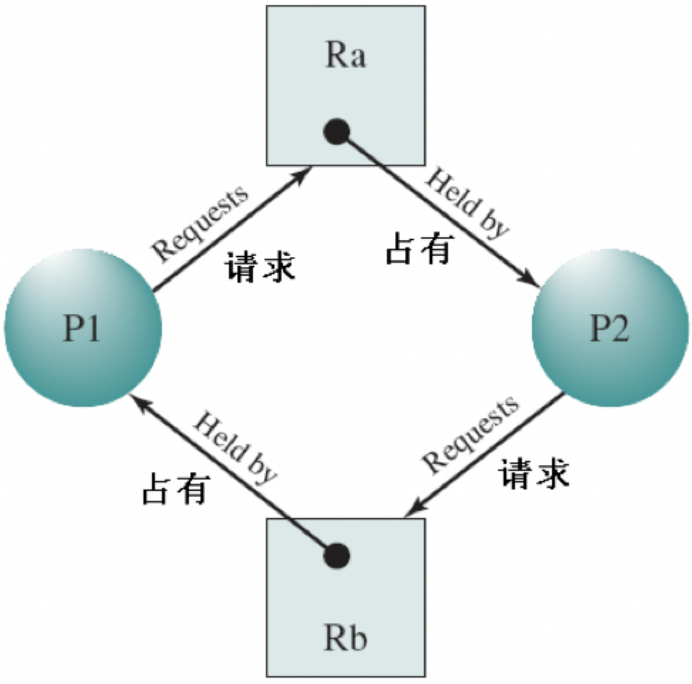
Example of Two Processes Competing for Reusable Resources

两个进程竞争可重用资源的例子

- 可消耗资源：可被创建和销毁的资源。（中断、信号、消息、I/O缓冲区中的信息）
  - 例子

```
P1:                                P2:
…                                  …
receive(P2);                       receive(P1);
…                                  …
send(P2, M1);                      send(P1, M2);
…                                  …
```

- 资源分配图



(c) Circular wait

循环等待

    - 死锁的必要条件
    - 互斥条件
        - 指进程对所分配到的资源进行排他性使用，即在一段时间内某资源只由一个进程占有
        - 如果此时还有其他进程要求该资源，要求者只能阻塞，直至占有该资源的进程用毕释放
    - 占有且等待条件
        - 进程已经占有至少一个资源，但可以提出新的资源要求
        - 若该资源已被其它进程占有，则请求进程阻塞，同时对已经获得的其它资源保持不放
    - 不可剥夺/抢占（preemption）条件
        - 进程已获得的资源在未使用完之前不能被剥夺，只能在使用完时由自己释放

- 死锁的充要条件

    - 死锁的必要条件加上：
        - 环路等待条件
            - 资源分配图中存在环路，即进程集合{P0, P1, P2, …, Pn}(n>=2)中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源

6.2死锁预防

- 死锁预防：通过破坏产生死锁的四个条件中的一个或多个条件，保证不会发生死锁
- 破坏互斥条件
  - 方法：允许多个进程同时使用资源
  - 适用条件：
    - 资源固有特性允许多个进程同时使用
    - 借助特殊技术允许多个进程同时使用
  - 缺点
    - 不适合绝大多数资源
- 破坏占有且等待条件
  - 方法：禁止已拥有资源的进程再申请其他资源，如要求所有进程在开始时一次性地申请在整个运行过程所需的全部资源；或申请资源时要先释放其占有资源后，再一次性申请所需全部资源
  - 优点：简单、易于实现、安全
  - 缺点：进程延迟运行、资源严重浪费
- 破坏不可剥夺条件
  - 方法：一个已经占有了某些资源的进程，当它再提出新的资源请求而不能立即得到满足时，必须释放它已经占有的所有资源，待以后需要时再重新申请
  - 适用条件：资源的状态可以很容易的保护和恢复（例如CPU）
  - 缺点：实现复杂、代价大，反复申请/释放资源、系统开销大、降低系统吞吐量
- 破坏环路等待条件
  - 方法：对所有资源按类型进行线性排队，进程申请资源必须严格按资源序号递增的顺序（可避免循环等待）
  - 缺点：
    - 很难找到令每个人都满意的编号次序，类型序号的安排只能考虑一般作业的情况, 限制了用户简单、自主地编程
    - 易造成资源的浪费（会不必要地拒绝对资源的访问）
    - 可能低效（会使进程的执行速度变慢）

6.3死锁避免

- 死锁避免：不需事先采取限制措施破坏产生死锁的必要条件，而是在资源的动态分配过程中，采用某种策略防止系统进入不安全状态，从而避免发生死锁
- 死锁避免的两种方法：

- 不启动其资源请求会导致死锁的进程
- 不允许会导致死锁的进程资源请求

- 资源分配拒绝策略(银行家算法)

Resource = $\boldsymbol{R}=(R_1,R_2,\cdots,R_m)$	系统中每种资源的总量
Available = $\boldsymbol{V}=(V_1,V_2,\cdots,V_m)$	未分配给进程的每种资源的总量
Claim = $\boldsymbol{C}=\begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \cdots & \cdots & \ddots & \cdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{bmatrix}$	C_{ij} = 进程 i 对资源 j 的需求
Allocation = $\boldsymbol{A}=\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \cdots & \cdots & \ddots & \cdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{bmatrix}$	A_{ij} = 当前分配给进程 i 的资源 j

  - 安全状态：至少存在一个执行时序，使当前所有进程都能运行到结束状态。只要系统处于安全状态，必定不会进入死锁状态
  - 不安全状态：不存在一个执行时序，使当前所有进程都能运行到结束状态，是指仅仅有死锁的可能性，但是并一定就是死锁。

- 死锁避免的缺点

  - 必须事先声明每个进程请求的最大资源
  - 考虑的进程必须是无关的，即它们执行的顺序没有任何同步要求的限制
  - 分配的资源数目必须是固定的
  - 在占有资源时进程不能退出

6.4死锁检测

- 死锁检测的算法

1. 标记 Allocation 矩阵中一行全为零的进程。
2. 初始化一个临时向量 $\boldsymbol{W}$，令 $\boldsymbol{W}$ 等于 Available 向量。
3. 查找下标 i，使进程 i 当前未标记且 $\boldsymbol{Q}$ 的第 i 行小于等于 $\boldsymbol{W}$，即对所有的 $1\leqslant k\leqslant m$, $Q_{ik}\leqslant W_k$。若找不到这样的行，终止算法。
4. 若找到这样的行，标记进程 i，并把 Allocation 矩阵中的相应行加到 $\boldsymbol{W}$ 中，即对所有的 $1\leqslant k\leqslant m$，令 $W_k=W_k+A_{ik}$。返回步骤 3。

  第一步的原因是没有分配资源的进程是不会参与死锁的。
  - 检测时机
  - 资源申请时（早期检测，算法相对简单，但处理机时间消耗大）
  - 周期性检测，或死锁“好像”已经发生（如CPU使用率降低到一定阈值时）
- 恢复
  - 剥夺法

连续剥夺资源直到不再存在死锁

- 回退法
  把每个死锁进程回滚(rollback)到前面定义的某些检查(checkpoint), 并且重新启动所有进程（死锁可能重现）
- 杀死进程法
  - 杀死所有死锁进程（最常用）
  - 或连续杀死死锁进程直到不再存在死锁
  - 或杀死一个非死锁进程

6.6哲学家就餐问题

- 基于信号量的解决方案

```
semaphore fork[5]={1,1,1,1,1};
semaphore room=4;
int i;
void philosopher(int i) {
   while (true)  {
      think();
      P(room);
      P(fork[i]);
      P(fork[(i+1) mod 5]);
      eat();
      V(fork[(i+1) mod 5]);
      V(fork[i]);
      V(room); }
}
```

- 使用管程的解决方案