2_进程管理

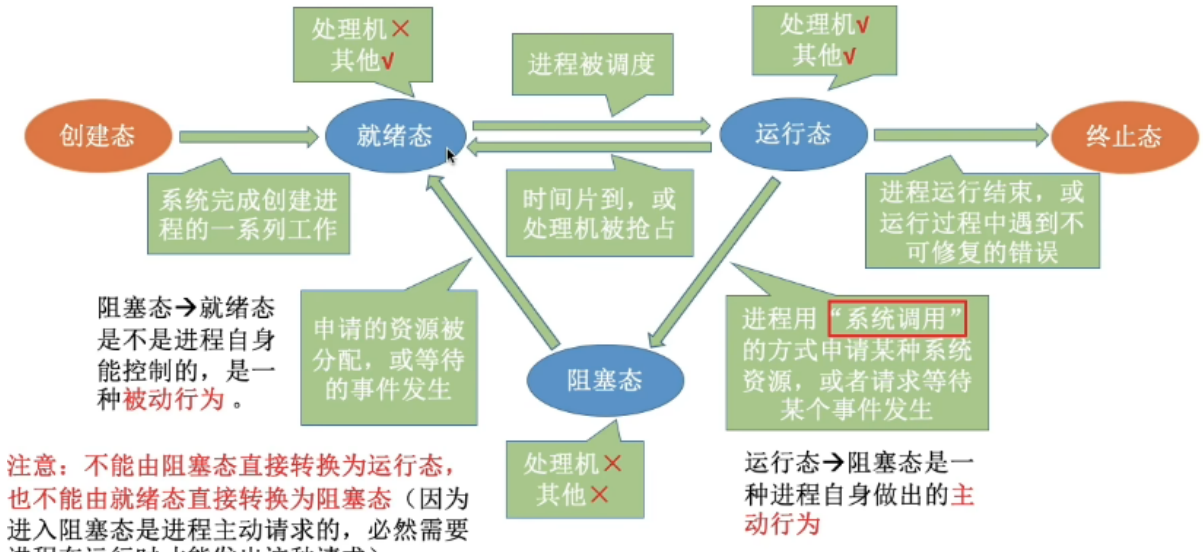
2.1_1进程的概念、组成、特征

- 程序|进程
  - 程序：是静态的，是可执行文件
  - 进程：是动态的，是执行过程
- PID：进程的身份证，用于区分不同的进程，不同的进程用有不同的PID
  - 唯一的，不重复的
- 进程实体（进程映像）：
  - 三部分组成：
    - 进程控制块（PCB）：
      - 是进程存在的唯一标志
      - 包括：
        - 进程描述信息：PID，UID
        - 进程控制和管理信息
        - 资源分配清单
        - 处理机相关信息
    - 程序段
      - 程序的代码（指令序列）
    - 数据段
      - 运行过程中产生的各种数据（如程序中定义的变量）
- 进程定义：
  - 是进程实体的运行过程，传统的进程！是系统进行资源分配和调度的一个独立单位
    - 进程是动态的
    - 进程实体是静态的
- 进程特征

- 动态性、并发性、独立性、异步性、结构性
    - 动态性：进程最基本的特性
    - 独立性：进程是能独立运行、独立获得资源、独立接收调度的基本单位
    - 异步性：各进程以不可预料的速度向前推进，可能导致运行结果的不确定性

2.1_2 进程的状态与转换

- 进程的状态
    - 创建态：创建PCB，程序段，数据段
    - 就绪状态：已经具备运行条件，但是没有空闲的CPU，暂时不能运行
        - CPU不能使用，其它资源已就绪
    - 运行状态：占有CPU，并在CPU上运行，单核只能一个进程（双核两个）
        - CPU可以使用，其它资源已就绪
    - 阻塞状态：等在某个事件的发生，暂时不能运行
        - CPU不能使用，其它资源未就绪
    - 终止状态：回收内存，程序段，数据段，撤销PCB
- 进程状态的转换



- 运行态->阻塞态：等待系统资源分配，或等待某时间的发生（进程主动的行为）
- 就绪态->运行态：进程被调度
- 阻塞态->就绪态：资源分配到位，等待某事件的发生（进程本身无法控制，是一种被动的行为）
- 运行态->就绪态：时间片到，或CPU被其他该优先级的进程抢占

- 在PCB中会使用一个变量state来记录当前的状态

- 进程的组织
  - 链式方式：指针指向不同的队列
  - 索引方式：构建索引表

2.1_3 进程控制

- 进程控制
  - 实现各种进程状态的相互转化
- 实现进程控制
  - 原语
    - 具有原子性期间不允许被中断
      - 可以用“关中断指令”和“开中断指令”这两个特权指令实现原

子性

- 原语做的事情
  - 更新PCD中的信息
  - 将PCD插入合适的队列
  - 分配/回收资源

- 进程控制相关的原语
  - 进程的创建
    - 创建原语：
      - 申请空白PCB
      - 为新进程分配所需资源
      - 初始化PCB
      - 将PCB插入就绪队列
      - 事件：
        - 用户登录、作业调度、提供服务、应用请求
  - 进程的终止
    - 撤销原语：
      - 从PCB集合中找到终止进程的PCB
      - 若进程正在执行，立即剥夺CPU，将CPU分配给其他进程
      - 终止其所有子进程、将该进程拥有的所有资源还给父进程或者操作系统
      - 删除PCB
      - 事件：
        - 正常结束、异常结束、外界干预
  - 进程的阻塞
    - 阻塞原语：
      - 找到要阻塞的进程对应的PCB
      - 保护进程运行的现场，PCB设置为“阻塞态”
      - 将PCB插入相应事件的等待队列
      - 事件：

            - 需要等待系统分配某种资源、需要等待相互合作的其他进程完成工作
    - 进程的唤醒
        - 唤醒原语
            - 在事件等待队列中找到对应的PCB
            - 将PCB从等待队列中移除，并将PCB设置为“就绪态”
            - 将PCB插入就绪队列，等待被调度
            - 事件：
                - 等待的事件发生
        - 唤醒原语与阻塞原语成对使用，==进程被什么指令阻塞就要被什么指令唤醒
    - 进程的切换
        - 切换原语
            - 将运行环境信息存入PCB
            - PCB移入相应队列
            - 选择另一个进程执行，并更新其PCB
            - 根据PCB恢复进程所需的运行环境
            - 事件：
                - 当前进程事件片到、有更高优先级的进程到达、当前进程主动阻塞、当前进程终止

2.1_4 进程通信

- IPC：两个进程之间产生数据交互
- 共享存储
    - 设置一个共享内存区域，并映射到进程的虚拟地址空间
    - 各进程对共享空间的访问是互斥的
        - P、V操作
    - 两种方式：
        - 基于存储区的共享：

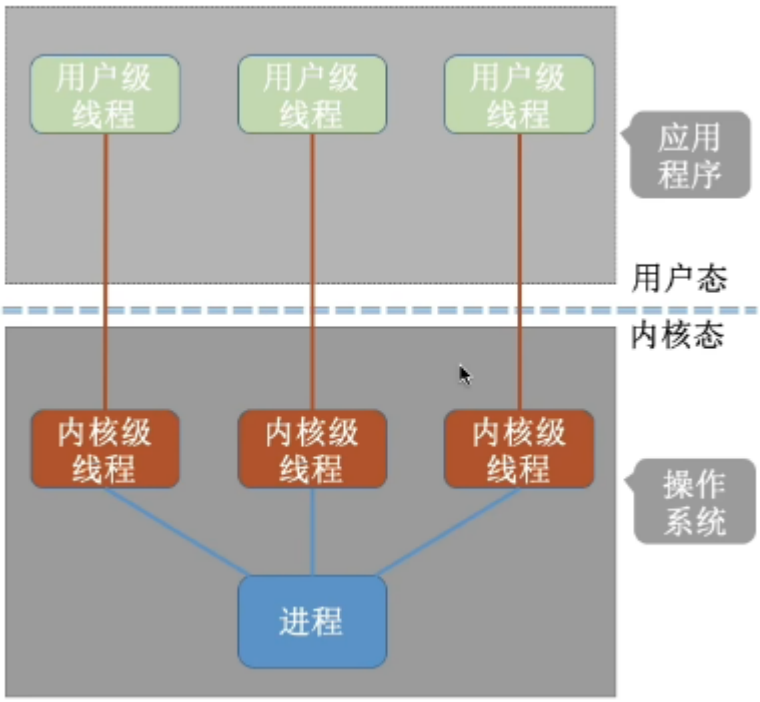
        - 划分存储区，是一种高级通信方式
    - 基于数据结构的共享：
        - 固定分配，是一种低级通信方式
- 消息传递
    - 以格式化的消息为单位，通过“发送消息/接收消息”两个原语进行数据交换
        - 消息：消息头（格式化的信息）、消息体
    - 两种方式：
        - 直接通信方式
            - 直接挂载消息
                - 直接点明进程发送给哪个进程，进程接收哪个进程发送的信息
        - 间接通信方式
            - 间接利用信箱发送消息
                - 指明发送给哪个信箱，进程指明接收哪个信箱中的消息
- 管道通信
    - ‘’管道‘’是特殊的共享文件（pipe文件）
    - 数据的读写遵循先进先出，数据流的形式
    - 只能支持半双工通信（单向传输），若要实现全双工通信，则需要设置两个管道
    - 各进程互斥的访问管道
    - 管道写满时，写进程阻塞
    - 管道读空时，读进程阻塞

2.1_5 线程概念和多线程模型

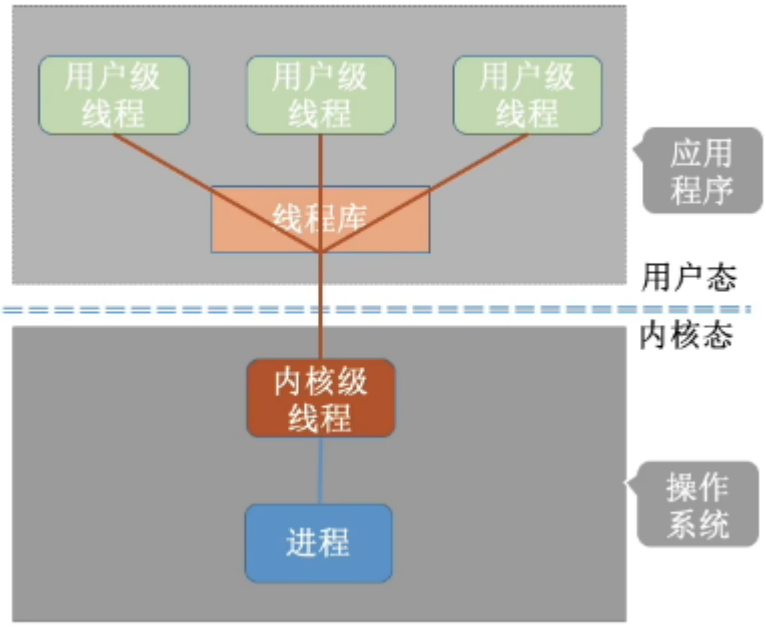
- 线程是一个基本的CPU执行单元，也是程序执行流的最小单位，进一步提高了系统的并发度
- 引入线程后

    - 资源分配、调度
        - 进程是资源分配的基本单位
        - 线程是调度的基本单位
    - 并发性
        - 各线程间也能并发，提升了并发度
    - 系统开销
        - 可以只在进程中切换，减小了CPU切换环境的系统开销
- 线程有哪些重要的属性
    - 线程是处理机调度的基本单位
    - 多CPU计算机中，各个线程可占用不同的CPU
    - 每个线程都有一个线程ID、线程控制块（TCB）
    - 线程也有就绪、阻塞、运行三种基本状态
    - 线程几乎不拥有系统资源
    - 同一进程的不同线程间共享进程的资源
    - 由于共享内存地址空间，统一进程中的线程间通信甚至无需系统干预
    - 同一进程中的线程切换，不会引起进程切换
    - 不同进程中的线程切换，会引起进程切换
    - 切换同进程内的线程，系统开销很小
    - 切换进程，系统开销较大
- 线程的实现方式
    - 用户级线程（ULT）
        - 由应用程序管理，从用户的视角看能看到的线程，操作系统感知不到线程的存在
        - 线程切换不需要变态
        - 优点：
            - 开销小，效率高
        - 缺点：
            - 并发度不高

        - 一个线程阻塞，则整个进程阻塞
    - 内核级线程（KLT）
        - 由操作系统管理，从操作系统内核视角看能看到的线程
        - 线程切换需要变态
        - 优点：
            - 并发能力强
        - 缺点：
            - 管理成本高
            - 开销大
- 多线程模型
    - 操作系统只“看得见”内核级线程，因此只有内核级线程才是处理分配的单位
    - 一对一模型
        - n个ULT映射到n个KLT
        - 优点：
            - 并发能力很强
            - 可在多核处理机上并行运行
        - 缺点：
            - 占用成本高，开销大

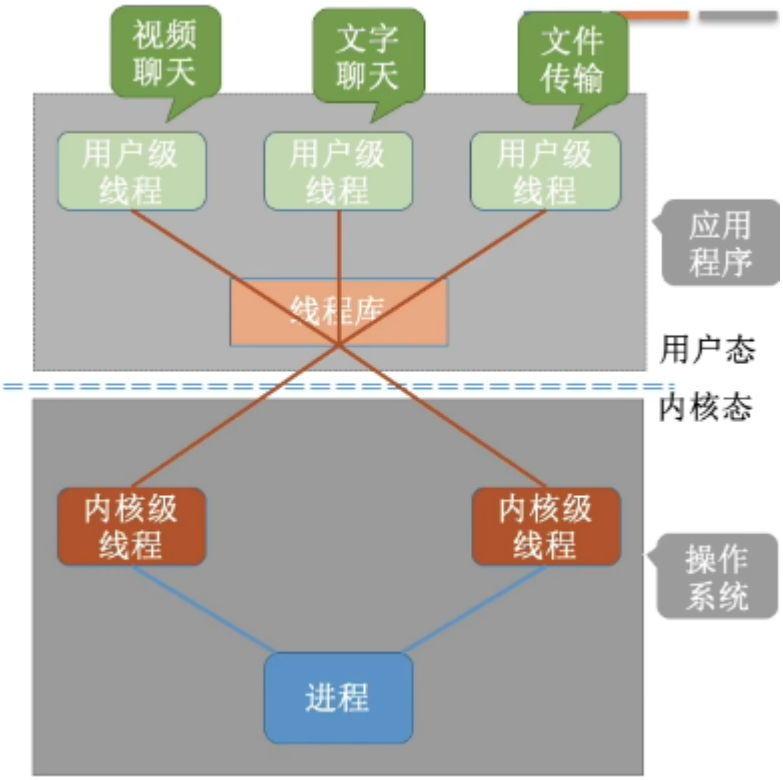


- 多对一模型
  - n个ULT映射到1个KLT
  - 优点：
    - 开销小，效率高
  - 缺点
    - 容易阻塞，并发度不高

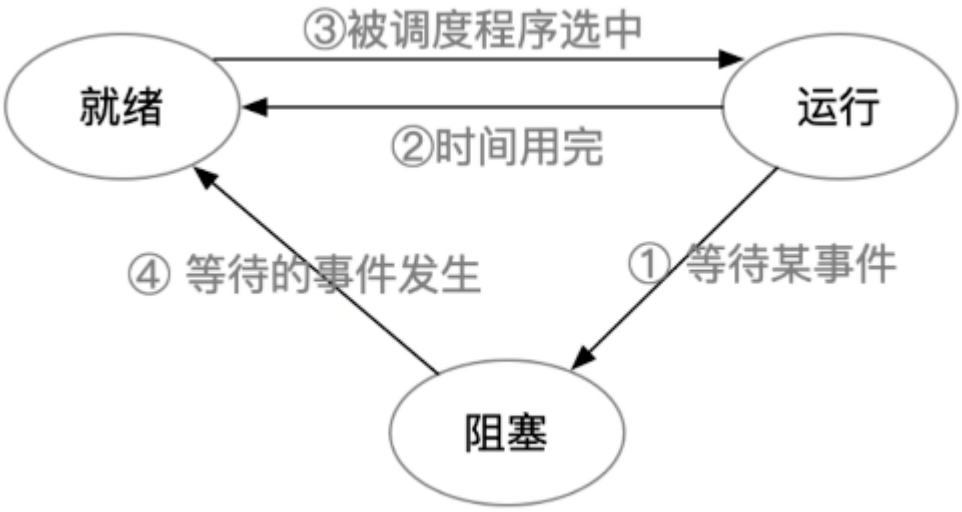


- 多对多模型

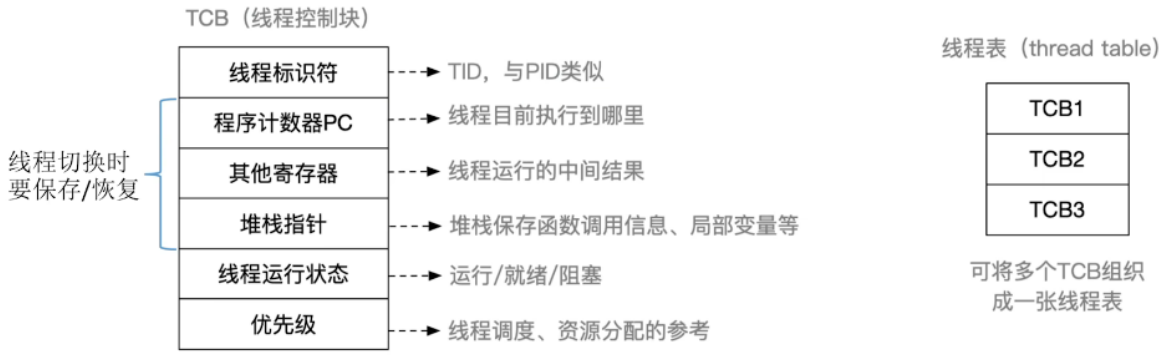
- n个ULT映射到m个KLT上（n>=m）
- 中和以上两种优缺点



- 线程的状态与转换



- 线程的组织与控制
  - 线程控制块（TCB）---->类别进程控制块（PCB）

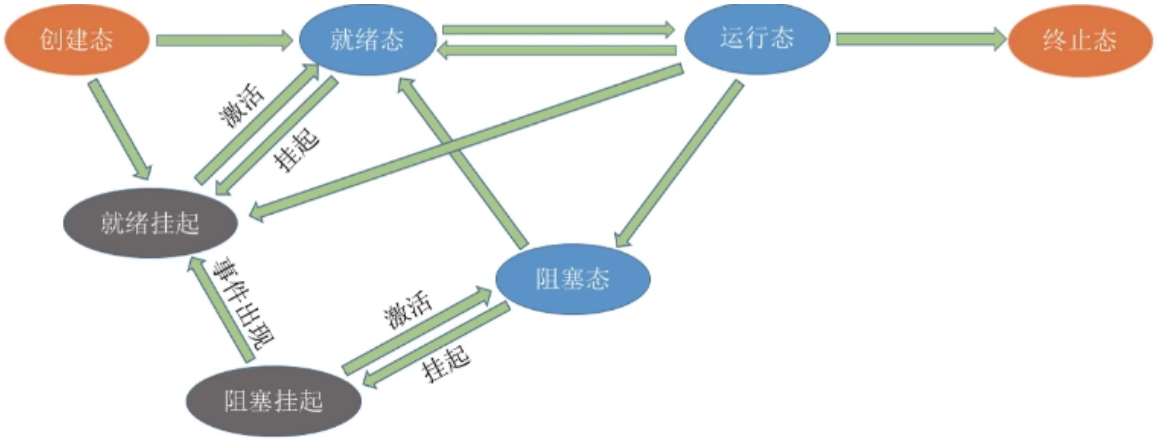


2.2_1 处理机调度的概念、层次

- 基本概念
  - 通常进程数量大于处理机数量，所以要按照一定的算法选择一个进程，并将处理机分配给它运行，以实现进程的并发执行
- 三个层次
  - 高级调度（作业调度）
    - 作业：一个具体的任务
    - 每个作业只调入一次，调出一次，作业调入时会建立相应的PCB，作业调出时才撤销PCB
  - 中级调度（内存调度）
    - 将暂时不用的进程放到外存（PCB不外放），提高内存利用率和系统吞吐量，进程状态为挂起状态，形成挂起队列
  - 低级调度（进程调度/处理机调度）
    - 是最基本的一种调度
    - 调度的频率很高

	要做什么	调度发生在..	发生频率	对进程状态的影响
高级调度（作业调度）	按照某种规则，从后备队列中选择合适的作业将其调入内存，并为其创建进程	外存→内存（面向作业）	最低	无→创建态→就绪态
中级调度（内存调度）	按照某种规则，从挂起队列中选择合适的进程将其数据调回内存	外存→内存（面向进程）	中等	挂起态→就绪态（阻塞挂起→阻塞态）
低级调度（进程调度）	按照某种规则，从就绪队列中选择一个进程为其分配处理机	内存→CPU	最高	就绪态→运行态

- 七状态模型



  - 挂起态与阻塞态
    - 挂起态和阻塞态都是不能获得CPU的服务
    - 挂起态将进程映像调到外存去了
    - 阻塞态进程映像还在内存中

2.2_2 进程调度的时机、切换与过程调度方式

- 时机
  - 可以进行进程调度
    - 主动放弃
      - 进程正常终止、运行过程中发生异常而终止、进程主动请求阻塞
    - 被动放弃

        - 分给进程的时间片用完、有更紧急的事需要处理、有更高优先级的进程进入就绪队列
    - 不能进行进程调度
        - 在处理中断的过程中
        - 在操作系统内核程序临界区中
            - 临界资源：一个时段段内各进程互斥地访问临界资源
            - 临界区：访问临界资源的那段代码
            - 内核程序临界区会访问就绪队列，导致其上锁
        - 在原子操作过程中（原语）
- 方式
    - 非剥夺调度方式（非抢占式）
        - 只允许进程主动放弃处理机
    - 剥夺调度方式（抢占式）
        - 进程被动放弃，可以优先处理紧急任务，适合分时操作系统、实时操作系统
- 切换与过程
    - “狭义的调度”与“进程切换”的区别
        - 狭义的调度：在就绪队列中选中一个要运行的进程
        - 广义的调度：狭义+进程切换
    - 进程切换的过程
        - 对原来运行进程各种数据的保存
        - 对新的进程各种数据的恢复
    - 进程切换、调度是有代价的，如果过于频繁的进程进程调度、切换，会使整个系统的效率降低，并不能提高系统的并发度

2.2_3调度器、闲逛进程

- 调度程序
    - 调度器/调度程序：管理调度
    - 不支持线程的擦着操作：调度程序的处理对象是进程

    - 支持线程的擦着操作：调度程序的处理对象是内核级线程
    - 触发“调度程序”
        - 创建新进程
        - 进程退出
        - 运行进程阻塞
        - I/O中断发生
    - 非抢占式调度策略：只有运行进程阻塞或退出才会触发调度程序
    - 抢占式调度策略：每个时钟中断或K个时钟中断都会触发调度程序
- 闲逛进程
    - 没有其他就绪进程时，运行闲逛进程（idle）
    - 优先级最低
    - 能耗低

2.2_4 调度算法的评价指标

- CPU利用率
    - $CPU利用率 = \frac{CPU忙碌的时间}{总时间}$
- 系统吞吐量（单位时间内完成的作业数量）
    - $系统吞吐量 = \frac{总共完成了多少道作业}{总共花了多少时间}$
- 周转时间（作业被提交给系统到完成这段时间间隔）
    - $(作业)周转时间 = 作业完成时间 - 作业提交时间$
    - $平均周转时间 = \frac{各作业周转时间之和}{作业数}$
    - $带权周转时间 = \frac{作业周转时间}{作业实际运行时间} = \frac{作业完成时间-作业提交时间}{作业实际运行时间}$
        - 必然大于等于1
        - 带权周转时间和周转时间都是越小越好
    - $平均带权周转时间 = \frac{各作业带权周转时间之和}{作业数}$
- 等待时间（进程或作业等待处理机状态时间的和）
    - 进程：等待被服务的时间之和
    - 作业：建立后的等待时间+作业在外存后备队列中等待的时间

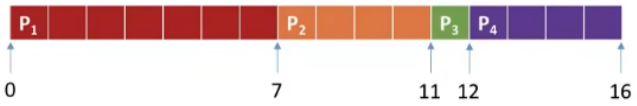
    - 等待时间 = 周转时间 – 运行时间（– I/O操作的时间）
- 响应时间
    - 从用户提交请求到首次产生响应所用的时间

2.2_5 FCFS、SJF、HRRN调度算法（适合早期批处理系统）

- 饥饿：某进程/作业长期得不到服务
- 关注进程或作业的平均周转时间，平均带权周转时间，平均等待时间
- 先来先服务（FCFS）
    - 算法思想
        - “公平”
    - 算法规则
        - 按照作业/进程到达的先后顺序进行服务
    - 用于作业/进程调度
        - 作业调度
            - 考虑哪个作业先到达后备队列
        - 进程调度
            - 考虑哪个进程先到达就绪队列
    - 是否可抢占
        - 先来先服务调度算法：到达时间越早的越优先得到服务
            - 非抢占式算法

进程	到达时间	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4

先来先服务调度算法：按照到达的先后顺序调度，事实上就是等待时间越久的越优先得到服务。
因此，调度顺序为：P1 → P2 → P3 → P4



周转时间 = 完成时间 - 到达时间

P1=7-0=7；P2=11-2=9；P3=12-4=8；P4=16-5=11

带权周转时间 = 周转时间/运行时间

P1=7/7=1；P2=9/4=2.25；P3=8/1=8；P4=11/4=2.75

等待时间 = 周转时间 – 运行时间

P1=7-7=0；P2=9-4=5；P3=8-1=7；P4=11-4=7

平均周转时间 = (7+9+8+11)/4 = 8.75
平均带权周转时间 = (1+2.25+8+2.75)/4 = 3.5
平均等待时间 = (0+5+7+7)/4 = 4.75

注意：本例中的进程都是纯计算型的进程，一个进程到达后要么在等待，要么在运行。如果是又有计算、又有I/O操作的进程，其等待时间就是周转时间 - 运行时间 - I/O操作的时间

        - 优点

        - 公平、算法简单
    - 缺点
        - 对长作业有利、对短作业不利
    - 是否会饥饿
        - 不会导致饥饿
- 短作业优先（SJF，shortest job first）
    - 算法思想
        - 追求最短的平均等待时间，最少的平均周转时间，最少的平均平均带权时间
    - 算法规则
        - 最短（服务时间最短）的作业/进程优先得到服务，时间相同，先到达的先被服务
    - 用于作业/进程调度
        - 可用于作业调度
        - 用于进程调度时称为“短进程优先算法（SPF）”
    - 是否可抢占
        - 非抢占式（SJF/SPF）
            - 选当前已经到达的且运行时间最短的作业/进程

进程	到达时间	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4

短作业/进程优先调度算法：每次调度时选择当前已到达且运行时间最短的作业/进程。
因此，调度顺序为：P1 → P3 → P2 → P4

周转时间 = 完成时间 - 到达时间　　P1=7-0=7；P3=8-4=4；P2=12-2=10；P4=16-5=11

带权周转时间 = 周转时间/运行时间　　P1=7/7=1；P3=4/1=4；P2=10/4=2.5；P4=11/4=2.75

等待时间 = 周转时间 – 运行时间　　P1=7-7=0；P3=4-1=3；P2=10-4=6；P4=11-4=7

平均周转时间 = (7+4+10+11)/4 = 8
平均带权周转时间 = (1+4+2.5+2.75)/4 = 2.56
平均等待时间 = (0+3+6+7)/4 = 4

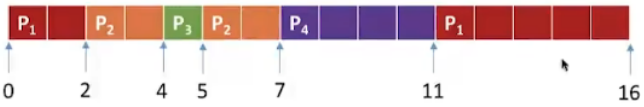
8.75
3.5
4.75

对比FCFS算法的结果，显然SPF算法的平均等待/周转/带权周转时间都要更低

        - 抢占式（最短剩余时间优先算法 SRTN）
            - 有新作业进入就绪队列或有作业完成了，考察队列中的最小需要时间的作业

进程	到达时间	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4

最短剩余时间优先算法：每当有进程加入就绪队列改变时就需要调度，如果新到达的进程剩余时间比当前运行的进程剩余时间更短，则由新进程抢占处理机，当前运行进程重新回到就绪队列。另外，当一个进程完成时也需要调度



需要注意的是，当有新进程到达时就绪队列就会改变，就要按照上述规则进行检查。以下 P_n（m）表示当前 P_n 进程剩余时间为 m。各个时刻的情况如下：

0时刻（P1到达）： P_1（7）
2时刻（P2到达）： P_1（5）、P_2（4）
4时刻（P3到达）： P_1（5）、P_2（2）、P_3（1）
5时刻（P3完成且P4刚好到达）： P_1（5）、P_2（2）、P_4（4）
7时刻（P2完成）： P_1（5）、P_4（4）
11时刻（P4完成）： P_1（5）

周转时间 = 完成时间 - 到达时间　　P1=16-0=16；P2=7-2=5；P3=5-4=1；P4=11-5=6

带权周转时间 = 周转时间/运行时间　　P1=16/7=2.28；P2=5/4=1.25；P3=1/1=1；P4=6/4=1.5

等待时间 = 周转时间 – 运行时间　　P1=16-7=9；P2=5-4=1；P3=1-1=0；P4=6-4=2

平均周转时间 = (16+5+1+6)/4 = 7
平均带权周转时间 = (2.28+1.25+1+1.5)/4 = 1.50
平均等待时间 = (9+1+0+2)/4 = 3

- 如果题目中未特别说明，默认短作业优先算法为非抢占式
- 在所有进程都几乎同时到达时，采用SJP调度算法的平均等待时间、平均周转时间最少
- 若无上述前提，抢占式的短作业/进程的平均等待时间、平均周转时间最少

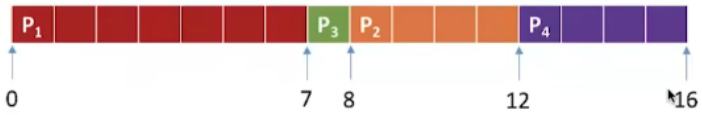
- 优点
  - “最短的”平均等待时间，平均周转时间
- 缺点
  - 对短作业有利，对长作业不利
- 是否会饥饿
  - 可能产生饥饿现象

- 高响应比优先（HRRN）
  - 算法思想
    - 要综合考虑作业/进程的等待时间和要求服务的时间
  - 算法规则
    - 在每次调度时先计算各个作业/进程的响应比，选择响应比最高的作业/进程为其服务

- 响应比 $= \frac{(等待时间+要求服务时间)}{要求服务时间}$
    - 大于等于1
- 用于作业/进程调度
    - 都可使用
- 是否可抢占
    - 非抢占式

进程	到达时间	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4

高响应比优先算法：非抢占式的调度算法，只有当前运行的进程主动放弃CPU时（正常/异常完成，或主动阻塞），才需要进行调度，调度时计算所有就绪进程的响应比，选响应比最高的进程上处理机。



响应比 = $\frac{等待时间+要求服务时间}{要求服务时间}$

P2和P4要求服务时间一样，但P2等待时间长，所以必然是P2响应比更大

0时刻：只有 P_1 到达就绪队列，P_1 上处理机
7时刻（P1主动放弃CPU）：就绪队列中有 P_2(响应比=(5+4)/4=2.25)、P_3((3+1)/1=4)、P_4((2+4)/4=1.5)，
8时刻（P3完成）：P_2(2.5)、P_4(1.75)
12时刻（P2完成）：就绪队列中只剩下 P_4

- 是否会饥饿
    - 不会导致饥饿

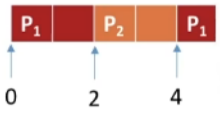
2.2_6 时间片轮转、优先级调度、多级反馈队列（适合交互式系统）

- 关注进程的响应时间
    - 响应比 $= \frac{(等待时间+要求服务时间)}{要求服务时间}$
- 时间片轮转算法（RR）
    - 算法思想
        - 公平轮流地位各个进程服务，让每个进程在一定时间间隔内都可以得到响应
    - 算法规则
        - 按照各进程到达就绪队列的顺序，轮流让各个进程执行一个时间片（如100ms）。若进程未在一个时间片内执行完，则剥夺处理机，将进程重新放到就绪队列对位重新排队。

- 用于作业/进程调度
  - 只能用于进程调度
- 是否可抢占
  - 抢占式：有时钟装置发出的时钟中断实现

进程	到达时间	运行时间
P1	0	5
P2	2	4
P3	4	1
P4	5	6

时间片轮转调度算法：轮流让就绪队列中的进程依次执行一个时间片（每次选择的都是排在就绪队列队头的进程）

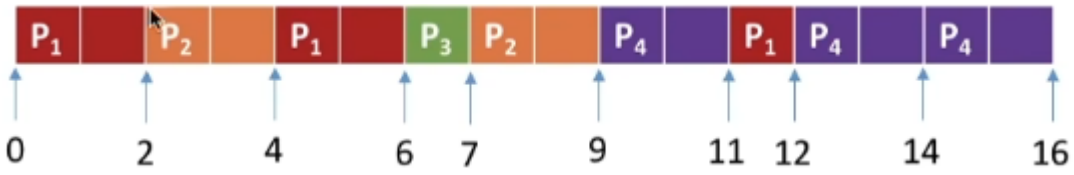


时间片大小为 2（注：以下括号内表示当前时刻就绪队列中的进程、进程的剩余运行时间）

0时刻（P1(5)）:0时刻只有P1到达就绪队列，让P1上处理机运行一个时间片
2时刻（P2(4) → P1(3)）:2时刻P2到达就绪队列，P1运行完一个时间片，被剥夺处理机，重新放到队尾。此时P2排在队头，因此让P2上处理机。（注意：2时刻，P1下处理机，同一时刻新进程P2到达，如果在题目中遇到这种情况，默认新到达的进程先进入就绪队列）
4时刻（P1(3) → P3(1) → P2(2)）:4时刻，P3到达，先插到就绪队尾，紧接着，P2下处理机也插到队尾
5时刻（P3(1) → P2(2) → P4(6)）:5时刻，P4到达插到就绪队尾（注意：由于P1的时间片还没用完，因此暂时不调度。另外，此时P1处于运行态，并不在就绪队列中）

6时刻（P3(1) → P2(2) → P4(6) → P1(1)）：6时刻，P1时间片用完，下处理机，重新放回就绪队尾，发生调度
7时刻（P2(2) → P4(6) → P1(1)）：虽然P3的时间片没用完，但是由于P3只需运行1个单位的时间，运行完了会主动放弃处理机，因此也会发生调度。队头进程P2上处理机。
9时刻（P4(6) → P1(1)）：进程P2时间片用完，并刚好运行完，发生调度，P4上处理机
11时刻（P1(1) → P4(4)）：P4时间片用完，重新回到就绪队列。P1上处理机

12时刻（P4(4)）：P1运行完，主动放弃处理机，此时就绪队列中只剩P4，P4上处理机
14时刻（）：就绪队列为空，因此让P4接着运行一个时间片。
16时刻：所有进程运行结束

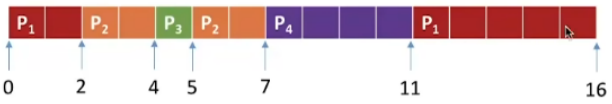


  - 若时间片设置的太大，则会退化成先来先服务的调度算法
  - 若时间片设置的太小，则会频繁切换进程
- 优点
  - 响应快，适用于分时操作系统
- 缺点
  - 由于高频率的进程切换，因此有一定的开销；不区分任务的紧急程度
- 是否会饥饿
  - 不会导致饥饿化

- 优先级调度算法
  - 算法思想
    - 根据任务的紧急程度来决定处理顺序
  - 算法规则
    - 每个进程/作业有各自的优先级，调度时选择优先级最高的作业/进程
  - 用于作业/进程调度
    - 作业/进程/IO
  - 是否可抢占
    - 抢占式

进程	到达时间	运行时间	优先数
P1	0	7	1
P2	2	4	2
P3	4	1	3
P4	5	4	2

抢占式的优先级调度算法：每次调度时选择当前已到达且优先级最高的进程。当前进程主动放弃处理机时发生调度。另外，当就绪队列发生改变时也需要检查是会发生抢占。



注：以下括号内表示当前处于就绪队列的进程
0时刻（P1）：只有P1到达，P1上处理机。
2时刻（P2）：P2到达就绪队列，优先级比P1更高，发生抢占。P1回到就绪队列，P2上处理机。
4时刻（P1、P3）：P3到达，优先级比P2更高，P2回到就绪队列，P3抢占处理机。
5时刻（P1、P2、P4）：P3完成，主动释放处理机，同时，P4也到达，由于P2比P4更先进入就绪队列，因此选择P2上处理机
7时刻（P1、P4）：P2完成，就绪队列只剩P1、P4，P4上处理机。
11时刻（P1）：P4完成，P1上处理机

    - 非抢占式
      - 优先级相同时，先到达的先运行

进程	到达时间	运行时间	优先数
P1	0	7	1
P2	2	4	2
P3	4	1	3
P4	5	4	2

非抢占式的优先级调度算法：每次调度时选择当前已到达且优先级最高的进程。当前进程主动放弃处理机时发生调度。

注：以下括号内表示当前处于就绪队列的进程
0时刻（P1）：只有P1到达，P1上处理机。
7时刻（P2、P3、P4）：P1运行完成主动放弃处理机，其余进程都已到达，P3优先级最高，P3上处理机。
8时刻（P2、P4）：P3完成，P2、P4优先级相同，由于P2先到达，因此P2优先上处理机
12时刻（P4）：P2完成，就绪队列只剩P4，P4上处理机。
16时刻（）：P4完成，所有进程都结束

  - 补充
    - 静态优先级：不变
    - 动态优先级：可以变
    - 通常：系统进程优先级高于用户进程，前台进程优先级高于后台进程，操作系统更偏好I/O进程

        - I/O进程：I/O繁忙进程
        - 计算型进程：CPU繁忙进程
    - 可以从追求公平、提升资源利用率等角度考虑改变优先级
  - 优点
    - 适用于实时操作系统
    - 可区分紧急程度
  - 缺点
    - 可能会导致饥饿
  - 是否会饥饿
    - 可能会导致饥饿
- 多级反馈队列调度算法
  - 算法思想
    - 对其它算法调度的这种权衡
  - 算法规则
    - 设置多级就绪队列，各级队列优先级从高到低，时间片从小到大。
    - 新进程到达时先进入第一级队列，按照FCFS原则排队等待被分配时间片。若用完时间片进程还未结束，则进程进入下一级队列队尾。如果此时已经在最下级的队列，则重新放回最下级队列末尾。
    - 只有第K级队头的进程为空时，才会为K+1级对头的进程分配时间片，被抢占处理机的进程重新放回原队列队尾。
  - 用于作业/进程调度
    - 用于进程调度
  - 是否可抢占
    - 可抢占式
  - 优点
    - 对各个进程相对公平（FCFS的优点），每个新到达的进程都可以很快就得到响应（RR的优点）
    - 短进程只用较少的时间就可以完成（SPF的优点）

        - 不必实现估计进程的运行时间（避免用户作假）
        - 可灵活地调整对各类进程的偏好程度，比如CPU密集型进程、IO密集型进程
    - 缺点
        - 会导致饥饿
    - 是否会饥饿
        - 会导致饥饿

2.3_1 进程同步、进程互斥

- 进程同步
    - 指为了完成某种任务而建立的两个或多个进程，这些进程因为需要在某些位置上协调他们的工作次序而产生的制约关系。进程间的直接制约关系就是源于它们之间的相互合作
- 进程互斥
    - 临界资源：一个时间段内只允许一个进程使用的资源
    - 对临界资源的互斥访问，可以在逻辑上分为四个部分:(见代码)
        - 临界区是进程中访问临界资源的代码段
        - 进入区和退出区是负责实现互斥的代码段
    - 原则：
        - 空闲让进
            - 空的可以直接进去
        - 忙则等待
            - 当有进程进入临界区后，则繁忙不能进去
        - 有限等待
            - 不能让进程等待无限长时间（保证不会饥饿）
        - 让权等待
            - 不能进去，应立即释放处理机，防止进程忙等待

```
do{
   entry section;  //进入区   对访问的资源检查或进行上锁
```

```
    critical section; //临界区(段) 访问临界资源的那部分代码
    exit section;    //退出区   负责解锁
    remainder section; //剩余区   其它处理
} while(true)
```

2.3_2 进程互斥的软件实现方法

- 单标志法
  - 算法思想
    - 两个进程在访问完临界区后会把使用临界区的权限教给另一个进程。也就是说每个进程进入临界区的权限只能被另一个进程赋予
  - 可以实现互斥
  - 存在的问题：
    - 每个进程只能轮流的使用，需要在上一个进程的退出区进行“谦让”
  - 违背：
    - 空闲让进原则

```
int turn =0;  //true 表示当前允许进入的临界区进程号

//p0进程
while(turn!=0); //进入区
critical section;//临界区
turn = 1;//退出区
remainder section;//剩余区

//p1进程
while(turn!=1);  //进入区
critical section;//临界区
turn = 0;//退出区
remainder section;//剩余区
```

- 双标志先检查

- 算法思想
  - 设置一个bool数组flag[ ]来标记自己是否想要进入临界区的意愿
- 存在的问题：
  - 由于进程是并发进行的，可能两个进程同时进入临界区
- 违背：
  - 忙则等待的原则

```
bool flag[2]={false,false};//初始都不想进入临界区

//p1进程
while(flag[1]);//如果flag[1]想进入临界区，则p1就一直循环等待
flag[0]=true;//标记为p1想要进入临界区
critical section;//访问临界区
flag[0]=false;//将p1重置为不想访问
remainder section;//剩余区

//p2进程
while(flag[0]);//如果flag[0]想进入临界区，则p2就一直循环等待
flag[0]=true;//标记为p2想要进入临界区
critical section;//访问临界区
flag[1]=false;//将p2重置为不想访问
remainder section;//剩余区
```

- 双标志后检查
  - 算法思想
    - 设置一个bool数组flag[]来标记自己是否想要进入临界区的意愿，不过是先上锁后检查
  - 可以实现互斥
  - 存在的问题：
    - 由于进程是并发进行的，可能会两个同时上锁，都进不去
  - 违背：
    - 空闲让进和有限等待原则

```
bool flag[2]={false,false};

//p1进程
flag[0]=true;
while(flag[1]);
critical section;
flag[0]=false;
remainder section;

//p2进程
flag[1]=true;//标记p2进程想要进入临界区
while(flag[0]);//如果p1也想进入，则p1循环等待
critical section;//访问临界区
flag[1]=false;//p2重置为不想访问
remainder section;//剩余区
```

- Peterson 算法
  - 算法思想
    - 主动让对方先使用处理器
  - 可以实现互斥
  - 遵循空闲让进、忙则等待、有限等待三个原则
  - 违背：
    - 让权等待原则

```
bool flag[2]={false,false};
int turn=0;

//p1进程
flag[0]=true;
turn=1;
while(flag[1]&&turn==1);
critical section;
flag[0]=false;
remainder section;

```

```
//p2进程
flag[1]=true;//表示自己想进入临界区
turn=0;//表示可以让对方先进入临界区
while(flag[0]&&turn==0);//如果对方想进入且自己谦让，则自己循环等待
critical section;//访问临界区
flag[1]=false;//p2重置为不想访问
remainder section;//剩余区
```

2.3_3 进程互斥的硬件实现方法

- 中断屏蔽方法
  - 原则
    - 利用“开/关中断指令”实现
      - 结构：关中断->临界区->开中断
  - 优点、
    - 简单
    - 高校
  - 缺点
    - 不适用于多处理机
    - 只适用于操作系统内核进行，不适用于用户进程（==“开/关”中断指令）运行在内核态
- TestAndSet（TS/TSL指令）
  - 原则
    - TSL是用硬件实现的，上锁、检查一气呵成
  - 优点
    - 实现简单
    - 适用于多处理机环境
  - 缺点
    - 不满足让权等待
    - 会盲等

```
//true表示已经上锁
```

```
bool TestAndSet(bool *lock){
  bool old;
  old=*lock;
  *lock=true;
  return old;
}

//以下是使用TSL指令实现互斥的算法逻辑
while(TestAndSet (&lock));//上锁并检查
···//临界区代码段
lock=false; //解锁
```

- Swap指令（Exchange指令、XCHG指令）
  - 原理
    - Swap指令是用硬件实现的，中间不允许被中断
  - 优点
    - 简单
    - 适用多处理机
  - 缺点
    - 不能让权等待

```
//true表示已经上锁
void Swap(bool *a,bool *b){
  bool temp;
  temp=*a;
  *a=*b;
  *b=temp;
}

//以下是使用Swap指令实现互斥的算法逻辑
bool old=true;
while(old=true)
  Swap(&lock,&old);
临界区代码段
lock=false; //解锁
```

```
15  //剩余代码段
```

2.3_4互斥锁

- 解决临界区的最简单的工具就是互斥锁
- 锁就是一个bool变量
- 适用于多处理器的系统
- 一个进程在进入临界区时获得锁，退出临界区时释放锁（均为原子操作）
  - acquire()获得锁
  - release()释放锁
  - 每个互斥锁有个bool变量available，表示是否可用
    - 可用则调用acquire()会成功，且上锁
    - 不可用则阻塞，直到锁释放
- 缺点
  - 忙等待（进程时间片用完才会下处理机）
  - 违反让权等待原则

```
1  aquire(){
2      while(!available)
3          ;//忙等待
4      available = false;//获得锁
5  }
6  release(){
7      available = true;//释放锁
8  }
```

2.3_5 信号量机制

- 信号量：是一种变量，表示系统中某种资源的数量
- 操作系统提供一对原语对信号量进行操纵
  - 一对原语：

        - wait（S）原语和signal（S）原语，分别简写为P（S）、V（S），简称P、V操作
- 整形信号量
    - 用一个整数表示系统资源的变量，用来表示系统中某种资源的数量
        - 该变量只能进行初始化、P操作、V操作
    - 可能出现忙等
        - 不满足让权等待原则

```
int S=1;
void wait(int S){ //wait原语，相当于：进入区
  while(S<=0); //如果资源数不够，就意志循环等待
  S=S-1;    //如果资源数够，则占用一个资源
}

void signal(int S){//signal原语，相当于“退出区”
  S=S+1;    //使用完资源后，在退出区释放资源
}
```

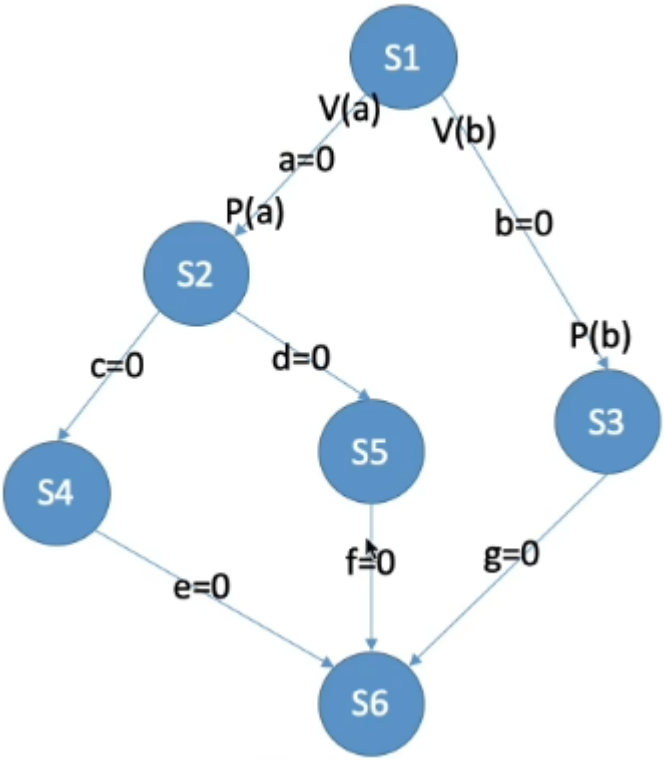
- 记录型信号量
    - 记录型数据结构表示的信号量
    - 除非特别说明，否则默认S为记录型信号量
    - 不会出现忙等

```
//记录型信号量的定义
typedef struct{
  int value;//剩余资源数
  struct process *L;//等待队列
} semaphore;

//某进程需要使用资源时，通过wait原语申请
void wait (semaphore S){
  S.value--;
  if(S.value<0){
    block (S.L);//将该进程加入到消息队列中
```

```
 }
}

//进程使用完资源后，通过signal原语释放
void signal (semaphore S){
  S.value++;
  if(S.valie<=0){
    wakeup(S.L);
 }
}
```

2.3_6 用信号量机制实现进程互斥、同步、前驱关系

- 实现进程互斥
  - 划定临界区
  - 设置互斥信号量mutex，初值为1
  - 对不同的临界资源需要设置不同的互斥信号量
  - 进入区 P（mutex）----申请资源
  - 退出区 V（mutex）----释放资源
  - P、V操作必须成对出现
- 实现进程同步
  - 保证一前一后的操作顺序
  - 设置同步信号量S，初始为0
    - semaphore S=0
  - 在“前操作”之后执行V（S）
  - 在“后操作”之后执行P（S）
    - 前V后P
- 实现进程的前驱关系
  - 要为每一对前驱关系各设置一个同步变量
  - 在“前操作”之后对相应的同步变量执行V操作
  - 在“后操作”之前对相应的同步变量执行P操作




```
P1() {
  ...
  S1;
  V(a);
  V(b);
  ...
}
```

```
P2() {
  ...
  P(a);
  S2;
  V(c);
  V(d);
  ...
}
```

```
P3() {
  ...
  P(b);
  S3;
  V(g);
  ...
}
```

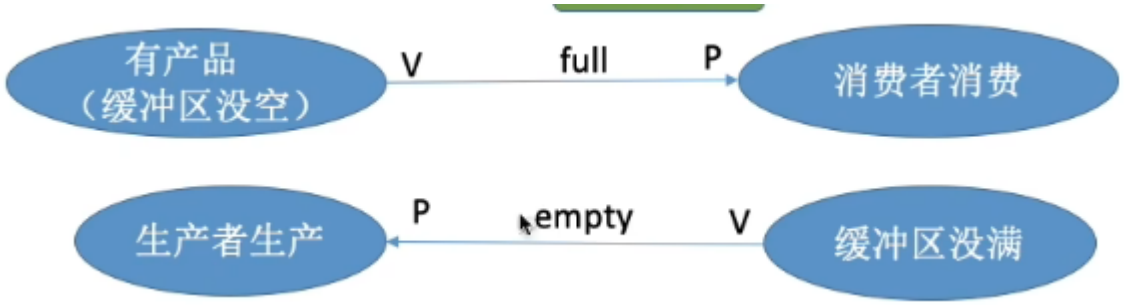
```
P4(){
  ...
  P(c);
  S4;
  V(e);
  ...
}
```

```
P5(){
  ...
  P(d);
  S5;
  V(f);
  ...
}
```

```
P6() {
  ...
  P(e);
  P(f);
  P(g);
  S6;
  ...
}
```

2.3_7 生产者-消费者问题

- 前提：
  - 只有缓冲区没满时，生产者才能把产品放入缓冲区，否则必须等待
  - 只有缓冲区不空时，消费者才能从中取出产品，否则必须等待
- 缓冲区是临界资源，各个进程互斥访问

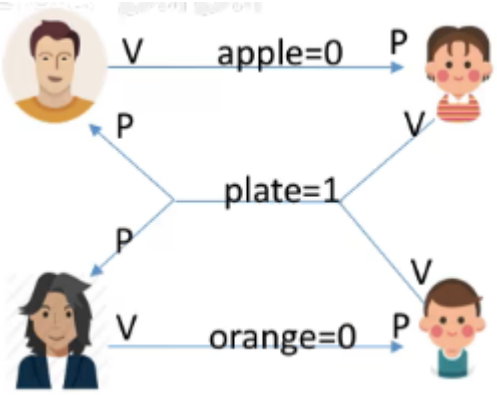


  - 实现互斥的P操作要放在实现同步的P操作之后，不然会发生死锁

- .V操作不会导致进程发生阻塞的状态，所以可以交换
- 使用操作不要放在临界区，不然并发度会降低

```
semaphore mutex = 1;//互斥信号量，实现对缓冲区的互斥访问
semaphore empty = n;//同步信号量，表示空闲缓冲区的数量
semaphore full = 0;//同步信号量，表示产品数量，也即非空缓冲区的数量

producter(){
    while(1){
        ...生产一个产品
        P(empty);
        P(mutex);
        ...把产品放入缓冲区
        V(mutex);
        V(full);
    }
}

consumer(){
    while(1){
        P(full);
        P(mutex);
        ...从缓冲区取出一个产品
        V(empty);
        V(mutex);
        ...使用产品
    }
}
```

2.3_8 多生产者-多消费者模型

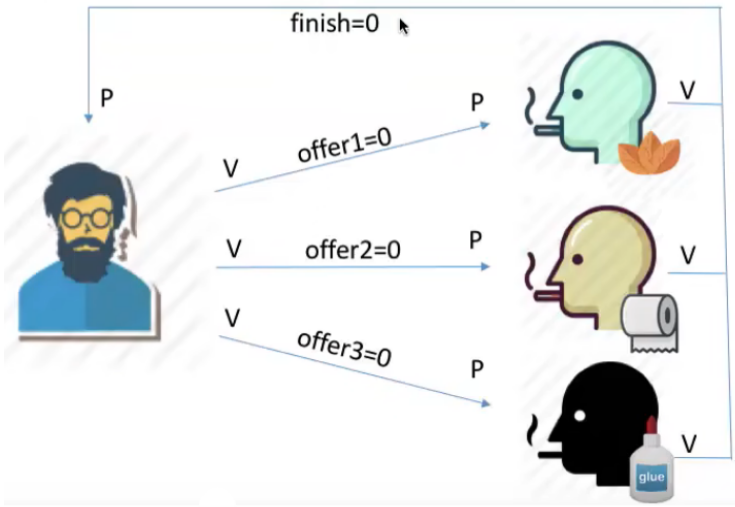



```
semahore mutex = 1;
semahore apple = 0;
semahore plate = 1;
semahore orange = 0;

dad(){
    while(1){
        ...准备一个苹果;
        P(palte);//检查盘子是否可用
        P(mutex);
        ...把苹果放入盘子;
        V(mutex);
        V(apple);//将苹果放入
    }
}

mom(){
    while(1){
        ...准备一个橘子;
        P(palte);//检查盘子是否可用
        P(mutex);
        ...把橘子放入盘子;
        V(mutex);
        V(orange);//将橘子放入
    }
}

daughter(){
```

```
        while(1){
            P(apple);//检查盘子中是否放有苹果
            P(mutex);
            ...从盘子中取出苹果;
            V(mutex);
            V(plate);
            ...吃掉苹果;
        }
    }

    son(){
        while(1){
            P(orange);//检查盘子中是否放有橘子
            P(mutex);
            ...从盘子中取出橘子;
            V(mutex);
            V(plate);
            ...吃掉橘子;
        }
    }
```

2.3_9 吸烟者问题

- 可生产多个产品的单生产者-多消费者



桌上有组合一 → 第一个抽烟者取走东西
桌上有组合二 → 第二个抽烟者取走东西
桌上有组合三 → 第三个抽烟者取走东西
发出完成信号 → 供应者将下一个组合放到桌上

```
semaphore offer1 = 0;
```

```
semaphore offer2 = 0;
semaphore offer3 = 0;
semaphore finish = 0;
int i = 0;//实现轮流抽烟

provider(){
    while(1){
        if(i==0){
            将组合一放到桌上
            V(offer1)
        }else if(i==1){
            将组合二放到桌上
            V(offer2)
        }else{
            将组合三放到桌上
            V(offer3)
        }
        i = (i+1)%3
        P(finish)
    }
}

smoker1(){
    while(1){
        P(offer1)
        从桌上拿走组合一；卷烟；抽掉
        V(finish)
    }
}

smoker2(){
    while(1){
        P(offer2)
        从桌上拿走组合二；卷烟；抽掉
        V(finish)
    }
}
```

```
39
40  smoker3(){
41      while(1){
42          P(offer3)
43          从桌上拿走组合三；卷烟；抽掉
44          V(finsh)
45      }
46  }
```

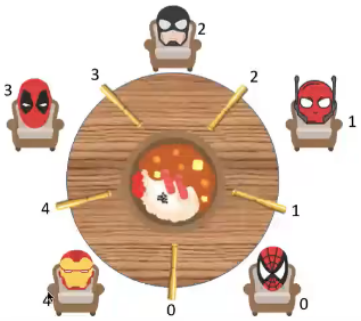
2.3_10 读者-写者问题

- 条件：
  - 允许多个读者同时对文件执行读操作
  - 只允许一个写者往文件中写信息
  - 任一写者在完成写操作之前不允许其他读者或写者工作
  - 写者执行写操作前，应让已有的读者和写者全部退出
- 互斥关系：
  - 写进程----写进程
  - 写进程----读进程

```
1   semaphore rw=1;//用于实现对文件的互斥访问。表示当前是否有进程在访问
    共享文件
2   int count=0;//记录当前有几个读进程在访问文件
3   semaphore mutex=1;//用于保证对count变量的互斥访问
4   semaphore w=1; //用于实现“写优先”
5
6   writer(){
7     while(1){
8       P（w）;
9       P(rw); //写之前“加锁”
10      写文件
11      V（rw);//写之后“解锁”
12      V(w);
13    }
14  }
```

```
15
16  reader(){
17    while(1){
18      P(w);
19      P(mutex); //各读进程互斥访问count
20      if(count==0)
21        P(rw); //第一个读进程的读进程数+1
22      count++; //访问文件的读进程数+1
23      V(mutex);
24      V(w);
25
26      读文件
27      P(mutex); //各读进程互斥访问count
28      count--; //访问文件的读进程数-1
29      if(count==0)
30        V(rw); //最后一个读进程负责“解锁”
31      V(mutex);
32    }
33  }
```

2.3_11 哲学家进餐问题

- 五个人，必须拿左右的筷子才能吃饭
  - 
- 同时避免死锁发生
  - 解决方案：
    - 可以对哲学家进程施加一些限制条件，比如最多允许四个哲学家同时进餐，这样可以保证至少有一个哲学家是可以拿到左右两只筷子的

- 要求奇数号哲学家先拿左边的筷子，然后再拿右边的筷子，而偶数号哲学家刚好相反。用这种方法可以保证如果相邻的两个奇偶号哲学家都想吃饭，那么只会有其中一个可以拿起第一只筷子，另一个会直接阻塞。这就避免了占有一只后再等待另一只的情况。
- 仅当一个哲学家左右两只筷子都可用时才允许他抓起筷子

```
semaphore chopstick[5]={1,1,1,1,1};
semaphore mutex = 1; //互斥地取筷子
Pi(){     //i号哲学家的进程
  while(1){
    P(mutex);
    p(chopstick[i]);   //拿右
    p(chopstick[(i+1)%5]);//拿左
    V(mutex);
    吃饭...
    V(chopstick[i]);
    V(chopstick[(i+1)%5]);
    思考...
 }
}
```

2.3_12 管程

- 为什么要引入管程
  - 信号量机制存在问题：P、V操作容易出错、困难
  - 引入管程，实现进程的互斥和同步
- 管程的定义和基本特征
  - 定义
    - 局部于管程的共享数据结构说明
    - 对该数据结构进程操作的一组过程
    - 对局部于管程的共享数据设置初始值的语句
    - 管程有一个名字

    - 基本特征：
        - 局部于管程数据结构只能被局部于管程的过程所访问
        - 一个进程只有通过调用管程内的过程才能进入管程访问共享数据
        - 每次仅允许一个进程在管程内执行某个内部过程
- 拓展1：用管程解决生产者消费者问题：
    - 只有通过特定的“入口”才能访问共享数据
    - 每次只能开发其中的一个“入口”，并且只能让一个进程或线程进入
        - 这种互斥特性是由编译器负责的，程序员可以不用关系

```
monitor Producerconsumer
  condition full,empty;
  int count = 0;
  void insert(Item item){
    if(count == N)
      wait(full);
    count++;
    insert_item (item);
    if(count == 1)
      signal(empty);
 }
  Item remove(){
    if(count == 0)
      wait(empty);
    count--;
    if(count == N-1)
      signal(full);
    return remove_item();
 }
  end monitor;

//使用
producer(){
  while(1){
    item = 生产一个产品;
```

```
26        Producerconsumer.insert(item);
27     }
28   }
29
30   consumer(){
31     while(1){
32        item = Producerconsumer.remove();
33        消费产品 item;
34     }
35   }
```

- 拓展2：Java中类似于管程的机制
  - java中用synchronized来描述一个函数,这个函数同一时间只能被一个线程调用

2.4_1 死锁的概念

- 进程死锁|饥饿|死循环
  - 死锁：
    - 定义：各进程互相等待对方手里的资源，导致各进程都阻塞，无法向前推进的现象。
    - 区别：至少两个或两个以上的进程同时发生死锁
  - 饥饿：
    - 定义：由于长期得不到想要的资源，某进程无法向前推进的现象。
    - 区别：可能只有一个进程发生饥饿
  - 死循环：
    - 定义：某进程执行过程中一直跳不出某个循环的现象。
    - 区别：死循环是程序员的问题，死锁和饥饿是操作系统的问题
- 死锁产生的必要条件
  - 互斥条件：多个进程争夺资源发生死锁
  - 不剥夺条件：进程获得的资源不能由其它进程强行抢夺，只能主动释放

    - 请求和保持条件：某个进程有了资源，还在请求资源
    - 循环等待条件：存在资源的循环等待链
        - 发生死锁的时候一定有循环等待，但是发送循环等待时未必死锁
- 什么时候会发生死锁
    - 对系统资源的竞争
    - 进程推进顺序非法
    - 信号量的使用不当也会造成死锁
- 死锁的处理策略
    - 预防死锁
    - 避免死锁
    - 死锁的检测和解除

2.4_2 死锁的处理策略——预防死锁

- 破坏互斥条件
    - 把互斥的资源改造为共享资源（SPOOLing技术）
    - 缺点
        - 有些不能破坏
- 破坏不剥夺条件
    - 方案1：当请求得不到满足的时候，立即释放手里的资源
    - 方案2：由系统介入，强行帮助剥夺
    - 缺点
        - 复杂，造成之前工作失效，降低系统开销，会全部放弃、导致饥饿
- 破坏请求和保持条件
    - 采用静态分配方法，一次性全部申请，如果申请不到，不要运行
    - 缺点
        - 资源利用率极低，可能会导致某些进程饥饿)
- 破坏循环等待条件

- 顺序资源分配法：对资源编号，进程按编号递增顺序请求资源
- 缺点
  - 不方便增加新的设备，实际使用与递增顺序不一致，会导致资源的浪费，必须按规定次序申请资源

2.4_3 死锁的处理策略——避免死锁

- 安全序列
  - 如果系统按照这种序列分配资源，则每个进程都能顺利完成
  - 安全序列可能有很多个
  - 如果分配资源后，系统找不到任何一个安全序列，系统进入不安全状态
  - 如果系统处于安全状态，就一定不会发生死锁。如果系统进入不安全状态，就可能发生死锁（处于不安全状态未必就是发生了死锁，但发生死锁时一定时在不安全状态）
- 银行家算法：
  - 在资源分配前先预判这次分配是否会导致系统进入不安全状态

数据结构：
长度为 m 的一维数组 Available 表示还有多少可用资源
n*m 矩阵 Max 表示各进程对资源的最大需求数
n*m 矩阵 Allocation 表示已经给各进程分配了多少资源
Max – Allocation = Need 矩阵表示各进程最多还需要多少资源
用长度为 m 的一位数组 Request 表示进程此次申请的各种资源数

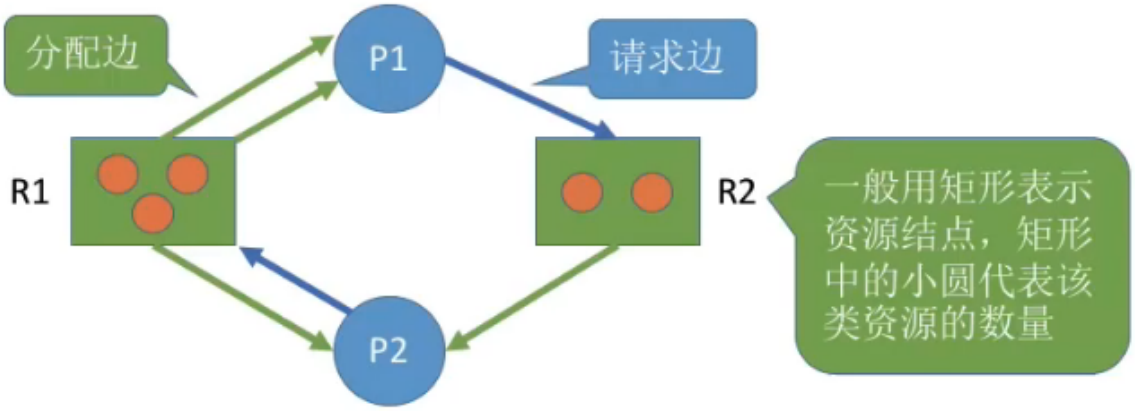
银行家算法步骤：
①检查此次申请是否超过了之前声明的最大需求数
②检查此时系统剩余的可用资源是否还能满足这次请求
③试探着分配，更改各数据结构
④用安全性算法检查此次分配是否会导致系统进入不安全状态

安全性算法步骤：
检查当前的剩余可用资源是否能满足某个进程的最大需求，如果可以，就把该进程加入安全序列，并把该进程持有的资源全部回收。
不断重复上述过程，看最终是否能让所有进程都加入安全序列。

2.4_4 死锁的处理策略——检测和解除

- 死锁的检测

- 用某种数据结构来保存资源的请求和分配信息
- 提供一种算法，利用上述信息来检测系统是否已进入死锁状态
  - 依次消除与不阻塞进程相连的边，直到无边可消



  - 如图，P1进程不会被阻塞，但P2会被阻塞
  - 如果这个图最终能消除所有的边，则这个图是可完全被简化的，相当于可以找到一个安全序列

- 死锁的解除
  - 资源剥夺法：挂起某些死锁进程，并抢占它的资源，将这些资源分配给其他的死锁进程。
  - 撤销进程法：强制撤销部分，甚至全部死锁进程，并剥夺这些进程的资源。
  - 进程回退法：让一个或多个死锁进程回退到足以避免死锁的地步。