操作系统概述

操作系统的概念

- 计算机系统从下往上分为：硬件，操作系统，应用程序，用户
  - **硬件：**提供基本的计算资源，如中央处理器，内存，输入/输出设备
  - **操作系统：**管理各种计算机硬件，为应用程序提供基础，并充当计算机硬件与用户之间的中介
  - **应用程序：**规定按何种方式使用这些资源来解决用户的计算问题，如字处理程序，电子制表软件，编译器，网络浏览器
- **操作系统(Operating System，OS)：**指**控制和管理**整个计算机系统的**硬件和软件资源**，合理地**组织，调度**计算机的**工作与资源的分配**，进而**为客户和其他软件**提供**方便接口与环境的程序集合**
- 操作系统是计算机系统中最基本的系统软件

操作系统的特征

- 操作系统的基本特征包括**并发，共享，虚拟，异步**
- **并发：**
  - 并发是指**两个或多个事件在同一时间间隔内发生**
  - 操作系统的并发性通过分时得以实现
  - 并发指的是一段时间间隔内多个程序交替执行，每个时刻实际上只有一个程序在执行
- **共享：**
  - 共享即资源共享，指系统中的资源可供内存中多个并发执行的进程共同使用，可分为**互斥共享方式和同时共享方式**
  - 互斥：同一个资源同一时间只有一个访问者可以进行访问，其他访问者需要等前一个访问者访问结束才可以开始访问该资源，但互斥无法限制访问者对资源的访问顺序，即访问是无序的
  - 同步：分布在不同进程之间的若干程序片断，它们的运行必须严格按照规定的某种先后次序来运行，这种先后次序依赖于要完成的特定的任务，所以同步就是在互斥的基础上，通过其它机制实现访问者对资源的有序访问
- **虚拟：**
  - 虚拟是指把一个物理上的实体变为若干逻辑上的对应物，可分为**时分复用技术和空分复用技术**
  - **虚拟处理器技术**是通过多道程序设计技术，采用让多道程序并发执行的方法，来分时使用一个处理器达到多个处理器的效果
  - 可以使用**虚拟存储器技术**将一台机器的物理存储器变为虚拟存储器，达到从逻辑上扩充存储器的容量
  - 可以使用**虚拟设备技术**将一台物理I/O设备虚拟成多台逻辑上的I/O设备，并允许每个用户占用一条逻辑上的I/O设备
- **异步：**
  - 异步是指当一个任务在执行时，系统不会等待它执行完成，而是直接执行下一个任务，可以让系统更有效率地利用资源，但不保证任务的执行顺序

操作系统的功能

- 操作系统的功能可分为：管理计算机系统资源，作为用户和计算机硬件系统之间的接口，实现对计算机资源的扩充
- **管理计算机系统资源**分为：处理机管理，存储器管理，设备管理，文件管理
  - 处理机管理：在多道程序环境下，处理机的分配和运行都以进程(或线程)为基本单位，对处理机的管理可归结为对进程的管理
    - 目的：处理**进程何时创建，何时撤销，如何管理，如何避免冲突，合理共享**的问题
    - 功能：进程控制，进程同步，进程通信，死锁处理，处理机调度
  - 存储器管理：
    - 目的：给多道程序的运行提供良好的环境，方便用户使用及提高内存的利用率
    - 功能：内存分配与回收，地址映射，内存保护与共享，内存扩充
  - 文件管理：
    - 操作系统中负责文件管理的部分称为文件系统
    - 功能：文件存储空间的管理，目录管理，文件读写管理和保护
  - 设备管理：
    - 目的：完成用户的I/O请求，方便用户使用各种设备，并提高设备的利用率
    - 功能：缓冲管理，设备分配，设备处理，虚拟设备
- **用户和计算机硬件系统之间的接口分为：**命令接口，程序接口
  - 命令接口进行作业控制的方式有两类：联机控制方式，脱机控制方式
  - 命令接口按照控制方式的不同可分为：联机命令接口和脱机命令接口
    - **联机命令接口：**又称交互式命令接口，适用于分时或实时系统的接口
      - 由一组键盘操作命令组成
      - 用户通过**控制台或终端**输入操作命令，向系统提出各种服务要求，控制权就从控制台或终端转移到操作系统的**命令解释程序**，然后由命令解释程序解释并执行输入的命令，完成指定的功能，之后控制权转回到控制台或终端
    - **脱机命令接口：**批处理命令接口，适用于批处理系统，由一组作业控制命令组成
      - 脱机用户事先用相应的作业控制命令写出一份作业操作说明书，连同作业一起提交给系统，系统调度到该作业时，由系统中的命令解释程序逐条解释执行作业说明书上的命令
  - 程序接口：由一组系统调用组成
    - 用户通过在程序中使用这些系统调用来请求操作系统为其提供服务，比如图形用户界面(GUI)，即图形接口，GUI最终通过调用程序接口来实现
- **扩充计算机资源：**
  - 没有任何软件支持的计算机称为**裸机**
  - 覆盖了软件的机器称为**扩充机器或虚拟机**

习题

- 源程序是一种计算机代码，是用应用程序设计语言编写的程序，经编译或解释后可形成具有一定功能的可执行文件，是直接面向用户的，不是操作系统的管理内容
- 系统调用是操作系统为应用程序使用内核功能所提供的接口
- 广义指令就是系统调用命令，命令解释器属于命令接口，shell是命令解释器，系统中的缓存全部由操作系统管理，操作系统不提供管理系统缓存的系统调用
- 多道程序设计没有封闭性和顺序性，**顺序性**是单道程序设计的基本特性
- 库函数属于用户程序而非系统调用，是系统调用的上层
- 计算机开机后，操作系统的程序会被自动加载到内存中的系统区，属于**RAM**
- 在多道程序系统中，进程数量越多，进程之间的资源竞争越激烈，可能因为资源竞争出现死锁现象，导致CPU利用率低
- 库函数与系统调用的区别：
    - 库函数是语言或应用程序的一部分，可以运行在用户空间中，许多库函数使用系统调用来实现功能，未使用系统调用的库函数的执行效率通常比系统调用高，因为使用系统调用需要上下文的切换及状态的转换(由用户态转向核心态)
    - 系统调用是操作系统的一部分，是内核为用户提供的程序接口，运行在内核空间中

操作系统发展历程

- 早期存在的主要问题：**人机矛盾和CPU与I/O之间速度不匹配**

单道批处理系统

- 单道批处理系统的特点：
  - 自动性：在顺利的情况下，作业自动完成
  - 顺序性：作业顺序进入内存，作业顺序完成
  - 单道性：内存中仅有一道程序运行
- 在运行期间发出输入/输出请求后，高速的CPU要等待低速的I/O完成，为了进一步提高资源的利用率和系统的吞吐量，提出多道程序技术

多道批处理系统

- 多道程序设计技术允许多个程序同时进入内存并允许他们在CPU中交替地进行，这些程序共享系统中的各种硬/软件资源
- 多道程序设计的特点：
  - 多道：内存中同时存放多道相互独立的程序
  - 宏观上并行：同时进入系统的多道程序都处于运行过程
  - 微观上串行：内存中的程序轮流占用CPU
- 多道程序设计技术解决的问题：
  - 如何分配处理器
  - 多道程序的内存分配问题
  - I/O设备如何分配
  - 如何组织和存放大量的程序和数据，以方便用户使用并保证其安全性和一致性
- 批处理系统中采用多道程序设计技术就形成了**多道批处理系统**
- 优点：
  - 资源利用率高
  - 系统吞吐量大
- 缺点：
  - 用户响应的时间长
  - 不提供人机交互能力

分时操作系统

- 为了解决人机交互问题，提出分时操作系统
- 分时技术：把处理器的运行时间分成很短的时间片，按时间片轮流把处理器分配给各联机作业使用
- 分时操作系统：多个用户通过终端同时共享一台主机而互不干扰
- 分时系统是实现人机交互的系统，具有与批处理系统不同的特征：

    - 同时性：也称多路性，指允许多个终端用户同时使用一台计算机
    - 交互性：用户通过终端采用人机对话的方式直接控制程序运行
    - 独立性：系统中多个用户之间独立操作互不干扰
    - 及时性：用户请求能在很短的时间内获得响应
- 但是在一些场合，需要系统能对外部的信息在规定时间内(短于时间片)完成，所以出现了实时操作系统

实时操作系统

- 实时操作系统分为**硬实时系统和软实时系统**
- 硬实时操作系统**必须**使任务在确定的时间内完成
- 软实时操作系统能让**绝大多数任务**在确定的时间内完成
- 特点：及时性和可靠性
- 实时操作系统追求的是最小的中断延时和线程切换延时

网络操作系统和分布式计算机系统

- 网络操作系统：把计算机网络中的计算机有机地结合起来，提供一种统一，经济而有效的使用各台计算机的方法，实现各台计算机之间数据的相互传送
- 网络操作系统的特点：网络中的各种资源的共享及各台计算机之间的通信
- 分布式计算机系统：
    - 系统中任意两台计算机通过通信方式交换信息
    - 系统中的每台计算机都具有同等的地位，没有主机和从机
    - 每台计算机上的资源为所有用户共享
    - 系统中的任意一台计算机都可以构成一个子系统，并且还能重构
    - 任何工作都可以分布在几台计算机上，由它们并行工作，协同完成
- 分布式计算机系统的特点：分布性和并行性
- 分布式操作系统和网络操作系统的不同：分布式操作系统中的若干计算机相互协同完成同一个任务

个人计算机操作系统

- 个人计算机操作系统是目前使用最广泛的操作系统，应用于文字处理，电子表格，游戏
- 常见的有Windows，Linux，MacOS，嵌入式操作系统，服务器操作系统，智能手机操作系统

习题

- 多道程序设计没有封闭性和顺序性，具有断续性，制约性，共享性
- 多道批处理系统的I/O设备可与CPU并行工作借助于中断技术的实现

操作系统运行环境

处理器运行模式

- CPU中运行两种程序：操作系统内核程序，用户自编程序(应用程序)
- 操作系统内核程序中执行**特权指令**，用户自编程序中执行**非特权指令**
- 特权指令：指不允许用户直接使用的指令，如I/O指令，置中断指令，存取用于内存保护的寄存器，送程序状态字到程序状态字寄存器的指令
- 非特权指令：指允许用户直接使用的指令，不能直接访问系统中的软硬件资源，仅限于访问用户的地址空间
- CPU的运行模式分为：用户态(目态)，内核态(管态)
- 应用程序向操作系统请求服务时通过使用访管指令，从而产生一个中断事件将操作系统转换为内核态
- 操作系统的内核分为：
  - 最底层的时钟管理，中断处理，设备驱动模块
  - 运行频率较高的程序，如进程管理，存储器管理，设备管理
- 大多数操作系统的内核包括四个内容：
  - **时钟管理**：负责计时，向用户提供系统时间，另外实现进程切换
  - **中断机制**：只有一小部分功能属于内核，负责保护和恢复中断现场的信息，转移控制权到相关的处理程序，减少中断的处理时间，提高系统的并行处理能力
  - **原语**：系统中的设备驱动，CPU切换，进程通信中的部分操作
    - 处于操作系统的最底层，是最接近硬件的部分
    - 这些程序的运行具有原子性，操作必须不间断执行完，关闭中断，使所有动作不可分割地完成后打开中断
    - 这些程序的运行时间都较短，而且频繁调用
  - **系统控制的数据结构及处理：**如作业控制块，进程控制块(PCB)，设备控制块，链表，消息队列，缓冲区，空闲区登记表
    - 进程管理：进程状态管理，进程调度，分派，创建和撤销进程控制块
    - 存储器管理：存储器的空间分配和回收，内存信息保护程序，代码对换程序
    - 设备管理：缓冲区管理，设备分配和回收

中断和异常

- 通过中断和异常，从用户态进入内核态，通过硬件实现
- 中断(Interruption)：也称外中断，指来自CPU执行指令外部的事件，通常用于信息输入/输出
  - 可屏蔽中断INTR：通过INTR线发出的中断请求，通过改变屏蔽字可以实现多重中断
  - 不可屏蔽中断NMI：通过NMI线发出的中断请求，通常是紧急的硬件故障
- 异常(Exception)：也称内中断，指来自CPU执行指令内部的事件，如程序的非法操作码，地址越界，运算溢出，虚存系统的缺页和专门的陷入指令，不能屏蔽，出现就要处理
  - 故障Fault：指令执行引起的异常，如非法操作码，缺页故障，除数为0，运算溢出
  - 自陷Trap：一种事先安排的“异常”事件，用于在用户态下调用操作系统内核程序，如条件陷阱指令
  - 终止Abort：出现了使CPU无法继续执行的硬件故障，如控制器出错，存储器校验错
- 故障异常和自陷异常属于软件中断(程序性异常)，终止异常和外部中断属于硬件中断

系统调用

- 系统调用：指用户在程序中调用操作系统提供的一些子功能
- 系统调用按功能可分为：
  - 设备管理：完成设备的请求或释放，以及设备启动等功能
  - 文件管理：完成文件的读，写，创建及删除等功能
  - 进程控制：完成进程的创建，撤销，阻塞及唤醒等功能
  - 进程通信：完成进程之间的消息传递或信号传递等功能
  - 内存管理：完成内存的分配，回收以及获取作业占用内存区大小及始址等功能
- 系统调用的处理需要由操作系统内核程序来完成，运行在内核态，用户程序可以执行**陷入指令(访管指令或trap指令)**来发起系统调用
- 操作系统的运行环境：
  - 用户通过操作系统运行上层程序
  - 上层程序的运行依赖于操作系统的底层管理程序提供服务支持
    - 当上层程序需要管理程序服务时，系统通过硬件中断机制进入内核态，运行管理程序
    - 或上层程序运行时出现异常情况，被动地需要管理程序的服务，通过异常处理来进入内核态
  - 管理程序执行结束后，用户程序需要继续执行，通过保存相应的程序现场退出中断处理程序或异常处理程序，返回断点继续执行
- 由用户态转换成内核态的例子：
  - 用户程序要求操作系统的服务，即系统调用
  - 发生一次中断
  - 用户程序中产生了一个错误状态
  - 用户程序中企图执行一条特权指令
  - 从内核态向用户态由一条特权指令实现，一般是中断返回指令
- 在用户态进入内核态时，不仅状态发生切换，所用的堆栈也需要由用户堆栈切换为系统堆栈，这个系统堆栈也属于该进程
- 程序由用户态转换为内核态使用**访管指令**，在用户态下使用，所以不是特权指令

习题

- 通用操作系统使用时间片轮转调度算法
- 操作系统执行程序时，必须从起始地址开始执行
- 编译器是操作系统的上层软件，不是操作系统需要提供的功能
- 批处理操作系统的主要缺点是**缺少交互性**
- 输入/输出指令需要中断操作，中断必须在核心态下执行，所以输入/输出指令工作在核心态
- 多道性是为了提高系统利用率和吞吐量而提出的
- I/O通道实际上是一种特殊的处理器，具有执行I/O指令的能力，并通过执行通道程序来控制I/O操作
- 中断是操作系统必须提供的功能，计算机的各种错误和核心态到用户态到转换都需要中断处理
- 内核可以执行处理能执行的任何指令，用户程序只能执行除特权指令外的指令

- 时钟管理中使用硬件计数器保持时钟的运行
- 进程调度由调度算法决定CPU使用权，由操作系统实现，不需要硬件的支持
- 计算机通过硬件完成操作系统由用户态到核心态的转换，通过中断机制来实现的
- 运行操作系统代码的状态为核心态
- 广义指令又称系统调用命令，只能工作在核心态
- 调用广义指令的那条指令不一定是特权指令，但广义指令存在于核心态中，执行也在核心态
- 子程序调用只需保存程序断点，即该指令的下一条指令的地址
- 中断处理不仅要保存断点(PC程序计数器)，还要保存程序状态字寄存器(PSW)
- 当CPU检测到中断后，由硬件自动保存被中断程序的断点(程序计数器PC和程序状态字寄存器PSW)，而通用寄存器内容由操作系统来保存
- 执行系统调用的过程：
    - 正在运行的进程先传递系统调用参数
    - 由陷入(Trap)指令负责将用户态转换为内核态
    - 将返回地址压入堆栈中
    - CPU执行相应的内核态服务程序
    - 返回用户态
- 时钟中断的主要工作是处理和时间有关的信息及决定是否执行调度程序
- 操作系统不同，底层逻辑，实现方式都不同，为应用程序提供的系统调用接口也不同
- 通道技术和中断技术结合起来可实现CPU和I/O设备并行工作
    - 即CPU启动通道传输数据后便去执行其他程序的计算工作，而通道则进行输入/输出操作
    - 当通道工作结束时，再通过中断机构向CPU发出中断请求，CPU则暂停正在执行的操作，对出现的中断进行处理，处理完后再继续原来的工作

操作系统结构

分层法

- **分层法：**是将操作系统分为若干层，最底层为硬件，最高层为用户接口，每层只能调用紧邻它的低层的功能和服务
- 优点：
  - 便于系统的调试和验证，简化了系统的设计和实现
  - 易扩充和易维护，在系统中增加，修改或替换一层中的模块或整层时，只要不改变响应层间的接口，就不影响其他层
- 缺点：
  - 合理定义各层比较困难
  - 效率较差，执行一个功能需要穿过多层，增加了开销

模块化

- **模块化：**是将操作系统按功能划分为若干具有一定独立性的模块，各模块之间能够通过接口进行通信
- 模块太小：降低模块本身的复杂性，但使模块之间的联系过多，造成系统比较混乱
- 模块过大：增加模块内部的复杂性，但是模块之间的联系变少
- 衡量模块的独立性的标准：
  - 内聚性：模块内部各部分间联系的紧密程度，内聚性越高，模块独立性越好
  - 耦合性；模块间相互联系和相互影响的程度，耦合度越低，模块独立性越好
- 优点：
  - 提高了操作系统设计的正确性，可理解性和可维护性
  - 增强了操作系统的可适应性
  - 加速了操作系统的开发过程
- 缺点：
  - 模块间的接口规定很难满足对接口的实际需求

宏内核

- 从操作系统的内核架构来分：宏内核和微内核
- **宏内核**，也称大内核或单内核，是指将系统的主要模块都作为一个紧密联系的整体运行在**内核态**，从而为用户程序提供高性能的系统服务
- 各管理模块之间共享信息，能有效利用相互之间的有效特性
- 目前的桌面操作系统：Windows，Android，IOS，macOS，Linux都是基于宏内核的构架

微内核

- **微内核构架：**是指内核中最基本的功能保留在内核，而将那些不需要在内核态执行的功能移到用户态，从而降低内核的设计复杂性
- 那些在用户态运行的操作系统代码根据分层的原则被划分为若干服务程序
- 微内核结构将操作系统划分为：微内核和多个服务器
    - 微内核是指能实现操作系统最基本核心功能的小型内核，包括：
        - 与硬件处理紧密相关的部分
        - 一些较基本的功能
        - 客户和服务器之间的通信
    - 微内核外的服务器实现大多数的功能：(作为进程实现，工作在用户态，用户和服务器间借助微内核的消息传递机制来进行交互)
        - 提供对进程进行管理的进程服务器
        - 提供虚拟存储器管理功能的虚拟存储器服务器
- 微内核的功能：
    - **进程(线程)管理：**进程或线程之间的通信功能是微内核OS最基本的功能，还有进程切换，进程调度，多处理机之间的同步
    - **低级存储器管理：**微内核中只配置最基本的低级存储器管理机制，如用于实现将逻辑地址变换为物理地址等的页表机制和地址变换机制，都是依赖于硬件的部分
    - **中断和陷入处理：**捕获所发生的中断和陷入事件，并进行中断响应处理，识别事件后再发送给相关的服务器处理
- 微内核结构通常利用“机制与策略分离”的原理来构造OS结构，所以一般与硬件紧密联系的部分放入微内核，有关软件，算法的部分放在外部服务器中
- 微内核的特点：
    - **扩展性和灵活性**，许多功能在内核外部，方便扩展和修改
    - **可靠性和安全性**，一个模块的错误只会使模块崩溃而不会使整个系统崩溃
    - **可移植性**，CPU和I/O硬件的代码放在内核中，而服务器的实现与硬件无关，方便移植
    - **分布式计算**，客户和服务器之间，服务器与服务器之间的通信采用消息传递机制
- 缺点：
    - 性能问题，需要在内核态和用户态之间不断进行切换，导致开销增大
- 应用：
    - 实时，工业，航空及军事应用，都需要有高度的可靠性

外核

- 外核在内核态中运行，为虚拟机分配资源，检查使用这些资源的企图，以确保没有机器会使用他人的资源
- 优点：
    - 减少了映射层
    - 将多道程序与用户操作系统代码分离

操作系统加载过程

- 操作系统是一种程序，程序以数据的形式存放在硬盘中，而硬盘中通常分为多个区，并且一个计算机有多个或多种外部存储设备
- 怎么在正确的存储设备中加载正确的分区从而启动操作系统?
- **操作系统引导**：是指计算机利用CPU运行特定程序，通过程序识别硬盘，再识别硬盘分区，再识别硬盘分区上的操作系统，最后通过程序启动操作系统
- 常见的操作系统引导过程：
  - **激活CPU**：激活的CPU读取ROM中的boot程序，将指令寄存器置为BIOS(基本输入/输出系统)的第一条指令，即开始执行BIOS的指令，BIOS是存储在计算机主板上的固件firmware
  - **硬件自检**：启动BIOS程序后，BIOS先进行硬件自检，检查硬件是否出现故障
    - 有故障：主板发出不同含义的蜂鸣，启动中止
    - 无故障：屏幕显示CPU，内存，硬盘
  - **加载带有操作系统的硬盘**：硬件自检后，BIOS开始读取Boot Sequence(通过CMOS保存的启动顺序)，把控制权交给启动顺序排在第一位的引导设备(硬盘，固态硬盘，光盘)
  - **加载主引导记录MBR**：BIOS确定引导设备后开始读取MBR，MBR是一个特殊的扇区，位于硬盘的第一个扇区（通常是第0扇区），其中包含引导加载程序。引导加载程序负责加载操作系统的核心部分
  - **扫描硬盘分区表，并加载硬盘活动分区**：MBR包含硬盘分区表，标识**活动分区和非活动分区**，MBR扫描硬盘分区表，识别含有操作系统的硬盘分区(活动分区)，之后开始加载硬盘活动分区，将控制权交给活动分区
  - **加载分区引导记录PBR**：读取活动分区的第一个扇区，称为分区引导记录(PBR)，寻找并激活分区根目录下用于引导操作系统的程序(启动管理器)
  - **加载启动管理器**：分区引导记录搜索活动分区中的启动管理器，加载启动管理器
  - **加载操作系统**：控制权就交给了操作系统，操作系统内核是操作系统的核心组件，负责管理系统的资源和提供各种服务，操作系统内核被加载后，它会进行一系列初始化步骤，包括初始化设备驱动程序、建立内存管理、设置进程控制块等。这些步骤确保操作系统能够有效地运行，并为用户提供一个稳定的环境
  - **用户登录或图形界面启动**： 最终，操作系统会进入用户登录界面或直接启动图形用户界面（GUI），用户可以通过输入用户名和密码登录系统。

虚拟机

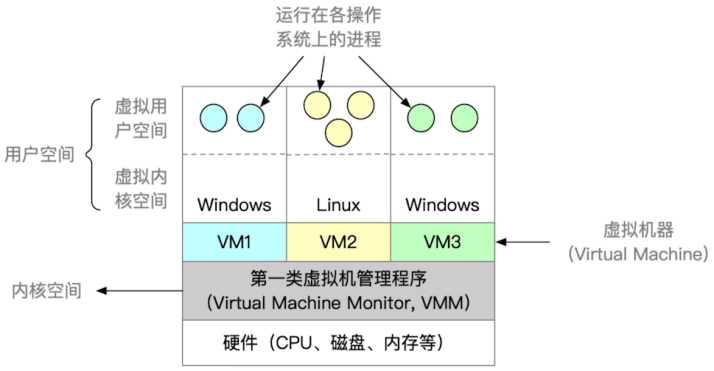
- 虚拟机是一台逻辑计算机，是指利用特殊的虚拟化技术，通过隐藏特定计算平台的实际物理特性，为用户提供抽象的，统一的，模拟的计算环境
- 有两种虚拟方法：
  - 第一类虚拟机管理程序(裸金属架构)
  - 第二类虚拟机管理程序(寄居架构)

第一类虚拟机管理程序

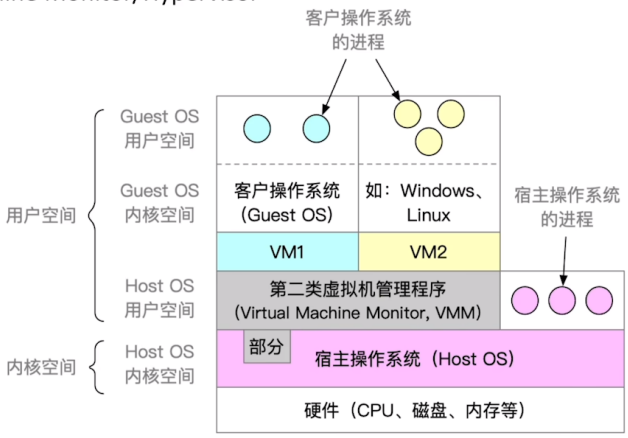
- 在裸机上运行并且具有多道程序功能，向上层提供若干台虚拟机，这些虚拟机是裸机硬件的精准复制品，所以不同的虚拟机上可以运行任何不同的操作系统
- 虚拟机作为用户态的一个进程运行，虚拟机上的操作系统认为自己运行在内核态(实际上运行在用户态)，这种称为**虚拟内核态**
  - 当虚拟机操作系统执行一条特权指令时，会陷入虚拟机管理程序，虚拟机管理程序将正确执行
  - 而如果是由虚拟机上的应用程序执行了特权指令，虚拟机管理程序将模拟真实硬件面对用户态执行特权指令时的行为

虚拟机：使用虚拟化技术，将一台物理机器虚拟化为多台虚拟机器（Virtual Machine, VM），每个虚拟机器都可以独立运行一个操作系统

同义术语：虚拟机管理程序/虚拟机监控程序/Virtual Machine Monitor/Hypervisor



第一类VMM，直接运行在硬件上



第二类VMM，运行在宿主操作系统上

第二类虚拟机管理程序

- 第二类虚拟机管理程序，依赖于Windows，Linux等操作系统分配和调度资源，像一个进程
- 运行在两种虚拟机管理程序上的操作系统称为**客户操作系统**
- 对于第二类虚拟机管理程序，运行在底层硬件上的操作系统称为**宿主操作系统**
- VMware Workstation是首个X86平台上的第二类虚拟机管理程序

习题

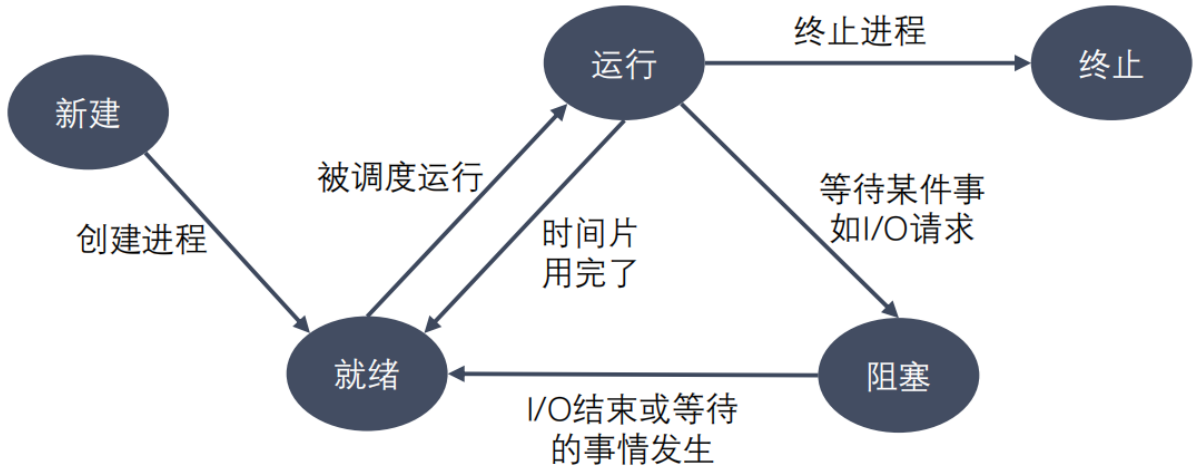
- 微内核OS：
  - 内核足够小
  - 基于客户/服务器模式(C/S模式)
  - 应用**机制与策略分离**原理
  - 采用面向对象技术
- 采用微内核，因为处于用户态的程序交互都借助于微内核进行通信，影响了系统的效率，并没有提高系统的高校
- 内核的服务越少，内核越稳定，所以微内核比宏内核稳定
- 操作系统的内核部分加载到内存中，其他部分仅在需要时加载到内存中
- 操作系统的引导程序：
  - 在传统BIOS系统中，它通常在**硬盘的MBR**中
  - 在UEFI系统中，它位于**EFI系统分区的特定目录**下
- 虚拟机可以使用软件实现，也可以使用硬件实现，并没有实现并行，因为本质上还是应用程序

进程和线程

进程

- **进程：**相当于一个任务，Task
- 进程的实体(也称进程映像)包括：
  - 程序段
  - 相关数据段
  - PCB(进程控制块)
- 进程的基本特征：
  - 动态性：进程有着创建，运行，就绪，阻塞，终止等状态，具有一定的生命周期，是动态地产生，变化和消亡的
  - 并发性：多个进程在一段时间内同时运行
  - 独立性：一个进程是独立运行，独立获得资源，独立接受调度的
  - 异步性：由于进程间的相互制约，各进程间独立运行，间断运行，执行的顺序不可知
- 进程的状态：
  - 运行态：进程正在CPU上运行，在单核中，一个时刻只能有一个进程在运行
  - 就绪态：进程获得了除CPU外的所有需要的资源，放入就绪队列中，等待调度
  - 阻塞态：又称等待态，进程在等待某一个事件的发生或某个资源，放入阻塞队列中
  - 创建态：进程正在被创建，创建的过程中首先申请一个PCB，再向PCB中填写用于控制和管理进程的信息，然后为该进程分配运行所需要的资源，如果资源全部满足则进入就绪态，否则一直处于创建态
  - 终止态：进程正常结束或因其他原因退出运行，设置为终止态之后处理资源释放和回收的工作
- 进程状态间的切换：
  - 就绪态->运行态：处于就绪态的进程得到调度，获得CPU资源
  - 运行态->就绪态：处于运行态的进程时间片用完，或在抢占式操作系统中有更高优先级的进程就绪

- 运行态->阻塞态：进程请求某一个资源的使用和分配或等待某一个事件的发生，进程以系统调用的方式请求操作系统提供服务
- 阻塞态->就绪态：进程等待的事件到来，中断处理程序将进程的状态从阻塞态转为就绪态



进程组成

- 进程的组成：
  - PCB(进程控制块)
  - 程序段
  - 数据段
- PCB用于保存处理机的状态信息，设置该进程恢复运行的现场，并根据PCB中的程序和数据的内存始址，找到其程序和数据
- **PCB中主要包括：**
  - 进程描述信息
    - 进程标识符PID：标志各个进程，每个进程都有一个唯一的进程标识符
    - 用户标识符UID：进程归属的用户，主要为共享和保护服务
  - 进程控制和管理信息
    - 进程当前状态：描述进程的状态信息
    - 进程优先级：进程抢占CPU的优先级
    - 代码运行入口地址
    - 程序的外存地址
    - 进入内存时间
    - 处理机占用时间
    - 信号量使用
  - 资源分配清单
    - 代码段指针
    - 数据段指针
    - 堆栈段指针
    - 文件描述符
    - 键盘

        - 鼠标
    - 处理机相关信息
        - 通用寄存器值
        - 地址寄存器值
        - 控制寄存器值
        - 标志寄存器值
        - 状态字
- **程序段：**
    - 能被CPU执行的程序代码
    - 程序可被多个进程共享，即多个进程可运行同一个程序
- **数据段：**
    - 进程运行过程中的数据变量
- **FreeRtos中的PCB**：(FreeRtos中以任务为基本单位，类似于传统操作系统中的线程，并没有进程，这里使用FreeRtos来进行理解)

```
/*
 * Task control block.  A task control block (TCB) is allocated for each task,
 * and stores task state information, including a pointer to the task's context
 * (the task's run time environment, including register values)
 * 任务状态，任务上下文，任务运行环境，寄存器的值
 */
typedef struct tskTaskControlBlock       /* The old naming convention is used to prevent breaking kernel aware debuggers. */
{
    volatile StackType_t * pxTopOfStack; /*< Points to the location of the last item placed on the tasks stack.  THIS MUST BE THE FIRST MEMBER OF THE TCB STRUCT. */堆栈指针，栈顶，栈用来保存局部变量和任务相关的信息

    ListItem_t xStateListItem;     /*< The list that the state list item of a task is reference from denotes the state of that task (Ready, Blocked, Suspended ). */保存任务状态的双向链表

    ListItem_t xEventListItem;     /*< Used to reference a task from an event list. */保存事件的链表

    UBaseType_t uxPriority;        /*< The priority of the task.  0 is the lowest priority. */优先级

    StackType_t * pxStack;         /*< Points to the start of the stack. */栈尾

    char pcTaskName[ configMAX_TASK_NAME_LEN ]; /*< Descriptive name given to the task when created.  Facilitates debugging only. */ /*lint !e971 Unqualified char types are allowed for strings and single characters only. */任务名字

    ......          /*还有许多条件编译*/
} tskTCB;
```

进程控制

- 进程控制主要对系统中的所有进程实施有效的管理，具有创建新进程，撤销已有进程，实现进程状态转换的功能
- 进程控制的程序段称为**原语**，原语在执行过程中不允许中断，是一个不可分割的基本单元
- **进程创建：**
  - 允许一个进程创建另一个进程，创建者称为父进程，被创建的进程称为子进程
  - 子进程可以继承父进程的所有资源，子进程被撤销时，将资源全部归还给父进程，撤销父进程时，同时撤销所有子进程
  - 创建一个新进程的过程：(创建原语)
    - 为新进程分配一个唯一的进程标识号，并申请一个空白PCB，若PCB申请失败，则创建失败
    - 为进程分配其运行所需的资源，如内存，文件，I/O设备和CPU时间，如果资源不足，则一直处于创建态等待资源
    - 初始化PCB，包括初始化标志信息，初始化处理机状态信息，初始化处理机控制信息，设置进程的优先级
    - 若进程就绪队列能接纳新进程，则将新进程插入就绪队列，等待被调度执行

```
BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,                指向任务函数
                        const char * const pcName,                任务的名称
                        const configSTACK_DEPTH_TYPE usStackDepth, 堆栈大小
                        void * const pvParameters,                传入参数
                        UBaseType_t uxPriority,                   任务优先级
                        TaskHandle_t * const pxCreatedTask )      任务句柄，也就是指向tcb的指针
{
    TCB_t * pxNewTCB;   //创建TCB
    BaseType_t xReturn; //设置函数返回值从而判断是否创建成功

    /* If the stack grows down then allocate the stack then the TCB so the stack does not grow into the TCB.  Likewise if the stack grows up then allocate the TCB then the stack. */
    向下增长(低地址向高地址)则先分配堆栈再分配TCB，向上增长则先分配TCB再分配堆栈
    #if ( portSTACK_GROWTH > 0 ) 判断堆栈增长方向
    {
        /* Allocate space for the TCB.  Where the memory comes from depends on
         * the implementation of the port malloc function and whether or not static
         * allocation is being used. */
        pxNewTCB = ( TCB_t * ) pvPortMalloc( sizeof( TCB_t ) );  //为TCB分配内存大小

        if( pxNewTCB != NULL ) //创建成功
        {
            memset( ( void * ) pxNewTCB, 0x00, sizeof( TCB_t ) );  //清0

            /* Allocate space for the stack used by the task being created.
             * The base of the stack memory stored in the TCB so the task can
             * be deleted later if required. */
            pxNewTCB->pxStack = ( StackType_t * ) pvPortMallocStack( ( ( ( size_t ) usStackDepth ) * sizeof( StackType_t ) ) );  //分配堆栈空间
            /*lint !e961 MISRA exception as the casts are only redundant for some ports. */

            if( pxNewTCB->pxStack == NULL ) //分配堆栈失败则删除TCB
            {
                /* Could not allocate the stack.  Delete the allocated TCB. */
                vPortFree( pxNewTCB );
                pxNewTCB = NULL;
            }
```

```
        }
    }
    #else /* portSTACK_GROWTH */
    {
        StackType_t * pxStack;

        /* Allocate space for the stack used by the task being created. */
        pxStack = pvPortMallocStack( ( ( ( size_t ) usStackDepth ) * sizeof( StackType_t ) ) ); /*lint !e9079 All values returned by pvPortMalloc() have at least the alignment required by the MCU's stack and this allocation is the stack. */

        if( pxStack != NULL )
        {
            /* Allocate space for the TCB. */
            pxNewTCB = ( TCB_t * ) pvPortMalloc( sizeof( TCB_t ) ); /*lint !e9087 !e9079 All values returned by pvPortMalloc() have at least the alignment required by the MCU's stack, and the first member of TCB_t is always a pointer to the task's stack. */

            if( pxNewTCB != NULL )
            {
                memset( ( void * ) pxNewTCB, 0x00, sizeof( TCB_t ) );

                /* Store the stack location in the TCB. */
                pxNewTCB->pxStack = pxStack;
            }
            else
            {
                /* The stack cannot be used as the TCB was not created.  Free
                 * it again. */
                vPortFreeStack( pxStack );
            }
        }
        else
        {
            pxNewTCB = NULL;
        }
    }
    #endif /* portSTACK_GROWTH */

    if( pxNewTCB != NULL )  //创建成功
    {
        #if ( tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE != 0 ) //判断是动态分配还是静态分配
        {
            /* Tasks can be created statically or dynamically, so note this
             * task was created dynamically in case it is later deleted. */
            pxNewTCB->ucStaticallyAllocated = tskDYNAMICALLY_ALLOCATED_STACK_AND_TCB;
        }
        #endif /* tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE */

        prvInitialiseNewTask( pxTaskCode, pcName, ( uint32_t ) usStackDepth, pvParameters, uxPriority, pxCreatedTask, pxNewTCB, NULL ); //实际创建函数
        prvAddNewTaskToReadyList( pxNewTCB ); //将TCB放入就绪链表中
        xReturn = pdPASS;
    }
    else
    {
        xReturn = errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY;
```

```
    }

    return xReturn;
}
```

- **进程终止：**
  - 引起进程终止的事件：
    - 正常结束：进程完成
    - 异常结束：进程运行时发生了某种异常事件，比如存储区越界，保护错，非法指令，特权指令错，运行超时，算术运算错
    - 外界干预：进程响应外界的请求终止运行
  - 进程终止的过程：
    - 根据被终止进程的标识符，检索出该进程的PCB，从中读出该进程的状态
    - 若被终止进程处于运行态，立即终止运行，将CPU分配给其他进程
    - 若该进程有子进程，则将子进程也同时终止
    - 将该进程所拥有的全部资源，还给父进程或还给操作系统
    - 将该PCB从所在队列(链表)中删除

```
void vTaskDelete( TaskHandle_t xTaskToDelete )
{
    TCB_t * pxTCB;

    taskENTER_CRITICAL(); 进入临界区
    {
        /* If null is passed in here then it is the calling task that is
         * being deleted. */
        pxTCB = prvGetTCBFromHandle( xTaskToDelete );  获取被删除任务的TCB

        /* Remove task from the ready/delayed list. */ uxListRemove将任务从就绪/等待链表中移除
        if( uxListRemove( &( pxTCB->xStateListItem ) ) == ( UBaseType_t ) 0 )
        {
            taskRESET_READY_PRIORITY( pxTCB->uxPriority );  清除相应的就绪标志位
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }

        /* Is the task waiting on an event also? */ 或将任务从阻塞链表中移除
        if( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem ) ) != NULL )
        {
            ( void ) uxListRemove( &( pxTCB->xEventListItem ) );
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }

        /* Increment the uxTaskNumber also so kernel aware debuggers can
         * detect that the task lists need re-generating.  This is done before
         * portPRE_TASK_DELETE_HOOK() as in the Windows port that macro will
         * not return. */
```

```
        uxTaskNumber++;

        if( pxTCB == pxCurrentTCB )    这个任务是当前正在运行的任务
        {
            /* A task is deleting itself.  This cannot complete within the
             * task itself, as a context switch to another task is required.
             * Place the task in the termination list.  The idle task will
             * check the termination list and free up any memory allocated by
             * the scheduler for the TCB and stack of the deleted task. */
            vListInsertEnd( &xTasksWaitingTermination, &( pxTCB->xStateListItem ) );  将该任务加入删除链表中

            /* Increment the ucTasksDeleted variable so the idle task knows
             * there is a task that has been deleted and that it should therefore
             * check the xTasksWaitingTermination list. */
            ++uxDeletedTasksWaitingCleanUp;

            /* Call the delete hook before portPRE_TASK_DELETE_HOOK() as
             * portPRE_TASK_DELETE_HOOK() does not return in the Win32 port. */
            traceTASK_DELETE( pxTCB );    任务删除后的跟踪函数，用户自己实现，可用于获取被删除任务的信息

            /* The pre-delete hook is primarily for the Windows simulator,
             * in which Windows specific clean up operations are performed,
             * after which it is not possible to yield away from this task -
             * hence xYieldPending is used to latch that a context switch is
             * required. */
            portPRE_TASK_DELETE_HOOK( pxTCB, &xYieldPending );
        }
        else
        {
            --uxCurrentNumberOfTasks;
            traceTASK_DELETE( pxTCB );

            /* Reset the next expected unblock time in case it referred to
             * the task that has just been deleted. */
            prvResetNextTaskUnblockTime();
        }
    }
    taskEXIT_CRITICAL();  退出临界区

    /* If the task is not deleting itself, call prvDeleteTCB from outside of
     * critical section. If a task deletes itself, prvDeleteTCB is called
     * from prvCheckTasksWaitingTermination which is called from Idle task. */
    if( pxTCB != pxCurrentTCB )    删除的任务不是自身，则在临界区外将其删除
    {
        prvDeleteTCB( pxTCB );
    }

    /* Force a reschedule if it is the currently running task that has just
     * been deleted. */
    if( xSchedulerRunning != pdFALSE )
    {
        if( pxTCB == pxCurrentTCB )
        {
            configASSERT( uxSchedulerSuspended == 0 );
            portYIELD_WITHIN_API();    因为正在运行的任务被删除，需要立即进行调度
        }
```

```
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
}
```

- **进程阻塞和唤醒：**
  - 运行态的进程，请求系统资源失败，等待某种操作的完成，新数据尚未到达，无新任务可做，进程则通过调用阻塞原语(Block)使自己从运行态变为阻塞态，**只有运行态的进程才能转为阻塞态**
  - 进程阻塞的过程：
    - 找到将要被阻塞进程的标识符对应的PCB
    - 若该进程为运行态，则保护现场，将其状态转为阻塞态，停止运行
    - 把该PCB插入相应事件的等待队列，将处理机资源调度给其他就绪进程

```
void vTaskSuspend( TaskHandle_t xTaskToSuspend )
{
    TCB_t * pxTCB;

    taskENTER_CRITICAL();     进入临界区
    {
        /* If null is passed in here then it is the running task that is
         * being suspended. */
        pxTCB = prvGetTCBFromHandle( xTaskToSuspend );    从队列中获取被阻塞的任务的TCB

        traceTASK_SUSPEND( pxTCB );  将任务阻塞

        /* Remove task from the ready/delayed list and place in the
         * suspended list. */  将任务从就绪/等待链表中移除并且放入阻塞链表中
        if( uxListRemove( &( pxTCB->xStateListItem ) ) == ( UBaseType_t ) 0 )
        {
            taskRESET_READY_PRIORITY( pxTCB->uxPriority );
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }

        /* Is the task waiting on an event also? */ 如果任务在等待事件，则将其从事件列表中移除
        if( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem ) ) != NULL )
        {
            ( void ) uxListRemove( &( pxTCB->xEventListItem ) );
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }

        vListInsertEnd( &xSuspendedTaskList, &( pxTCB->xStateListItem ) );  将任务插入到阻塞链表中

        #if ( configUSE_TASK_NOTIFICATIONS == 1 )
```

```
        {
            BaseType_t x;
            如果任务在等待通知，则将通知状态清除
            for( x = 0; x < configTASK_NOTIFICATION_ARRAY_ENTRIES; x++ )
            {
                if( pxTCB->ucNotifyState[ x ] == taskWAITING_NOTIFICATION )
                {
                    /* The task was blocked to wait for a notification, but is
                     * now suspended, so no notification was received. */
                    pxTCB->ucNotifyState[ x ] = taskNOT_WAITING_NOTIFICATION;
                }
            }
        }
        #endif /* if ( configUSE_TASK_NOTIFICATIONS == 1 ) */
    }
    taskEXIT_CRITICAL();

    if( xSchedulerRunning != pdFALSE )
    {
        /* Reset the next expected unblock time in case it referred to the
         * task that is now in the Suspended state. */
        taskENTER_CRITICAL();
        {
            prvResetNextTaskUnblockTime();
        }
        taskEXIT_CRITICAL();
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    if( pxTCB == pxCurrentTCB )
    {
        if( xSchedulerRunning != pdFALSE )
        {
            /* The current task has just been suspended. */
            configASSERT( uxSchedulerSuspended == 0 );
            portYIELD_WITHIN_API();
        }
        else
        {
            /* The scheduler is not running, but the task that was pointed
             * to by pxCurrentTCB has just been suspended and pxCurrentTCB
             * must be adjusted to point to a different task. */
            if( listCURRENT_LIST_LENGTH( &xSuspendedTaskList ) == uxCurrentNumberOfTasks
) /*lint !e931 Right has no side effect, just volatile. */
            {
                /* No other tasks are ready, so set pxCurrentTCB back to
                 * NULL so when the next task is created pxCurrentTCB will
                 * be set to point to it no matter what its relative priority
                 * is. */
                pxCurrentTCB = NULL;
            }
            else
            {
                vTaskSwitchContext();
```

```
            }
        }
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
```

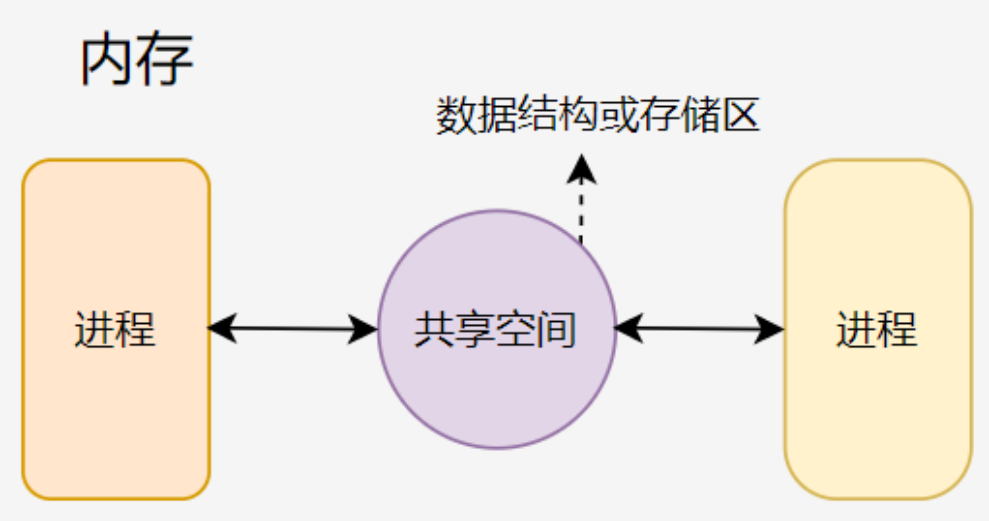
- 当被阻塞的进程等待的资源或事件到来时，由**其他进程**调用唤醒原语(Wakeup)将该进程唤醒
  - 在该事件的等待队列中找到相应进程的PCB
  - 将其从阻塞态转为就绪态
  - 等待调度程序调度

```
void vTaskResume( TaskHandle_t xTaskToResume )
{
    TCB_t * const pxTCB = xTaskToResume;

    /* It does not make sense to resume the calling task. */
    configASSERT( xTaskToResume );

    /* The parameter cannot be NULL as it is impossible to resume the
     * currently executing task. */
    if( ( pxTCB != pxCurrentTCB ) && ( pxTCB != NULL ) )
    {
        taskENTER_CRITICAL();
        {   检查任务是否被阻塞
            if( prvTaskIsTaskSuspended( pxTCB ) != pdFALSE )
            {
                traceTASK_RESUME( pxTCB );     唤醒任务

                /* The ready list can be accessed even if the scheduler is
                 * suspended because this is inside a critical section. */
                ( void ) uxListRemove( &( pxTCB->xStateListItem ) );
                prvAddTaskToReadyList( pxTCB ); 移除任务的阻塞状态，加入就绪链表中

                /* A higher priority task may have just been resumed. */
                if( pxTCB->uxPriority >= pxCurrentTCB->uxPriority )
                {      如果被唤醒的任务的优先级更高，则进行任务调度
                    /* This yield may not cause the task just resumed to run,
                     * but will leave the lists in the correct state for the
                     * next yield. */
                    taskYIELD_IF_USING_PREEMPTION();
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
```

```
41          taskEXIT_CRITICAL();
42      }
43      else
44      {
45          mtCOVERAGE_TEST_MARKER();
46      }
47  }
```

进程通信

- 进程通信是指进程之间的信息交换
  - 低级通信方式有：PV操作
  - 高级通信方式有：以较高速率传输大量数据的通信方式
    - 共享存储
    - 消息传递
    - 管道通信
- **共享存储：**
  - 通信的两个进程之间存在一块可直接访问的共享空间
  - 共享存储又可分为基于数据结构的低级共享方式和基于存储区的高级共享方式
  - 操作系统只负责为通信进程提供可共享使用的存储空间和同步互斥工具，数据交换由用户自己安排读/写指令完成



- **消息传递：**
  - 进程间的数据交换以格式化的信息(Message)为单位
  - 利用操作系统提供的消息传递方式来实现，即通过系统提供的**发送信息和接受信息**两个原语进行数据交换
  - **直接通信方式：**
    - 发送进程直接把消息发送给接收进程，并将它挂在接收进程的消息缓冲队列上，接收进程从消息缓冲队列中取消息
  - **间接通信方式：**
    - 发送进程把消息发送到某个中间实体，如**信箱**，接收进程从中间实体中取消息
- **管道通信：**
  - 发送进程往管道的一端写，接收进程从管道的一端读

- 数据在管道中是先进先出
- 只要管道非空，读进程就能从管道中读出数据，若管道为空，则读进程阻塞，直到写进程往管道中写数据
- 只要管道非满，写进程就能往管道中写入数据，若管道为满， 则写进程阻塞，知道读进程从管道中读数据
- 管道机制必须提供三方面的协调能力：同步，互斥，确定对方的存在
- Linux中的管道是一种特殊的文件，能克服使用文件的两个问题：
  - 限制管道的大小
  - 读进程也可能工作得比写进程快
- 管道只能由创建进程所访问，子进程可以通过父进程创建的管道来与父进程进行通信
- 普通管道只允许单向通信，若要双向通信则需要两个管道

线程

- 引入进程的目的是使多道程序并发执行，提高资源利用率和系统吞吐量
- 引入线程的目的是减小程序在并发执行时所付出的时空开销，提高操作系统的并发性能
- 线程是进程中的一个实体，由**线程ID，程序计数器，寄存器集合，堆栈**组成
- 线程与同属一个进程中的其他线程共享进程所拥有的全部资源
- 一个进程中有多个线程，线程的切换发生在同一个进程内部，只需要很少的时空开销
- 进程和线程的区别：
  - **定义：**
    - 进程是程序在执行过程中的一个实例。它是计算机中运行的程序的一个独立执行单元，有自己的地址空间、内存、文件描述符、以及其他系统资源
    - 线程是进程内的一个独立执行流。一个进程可以包含多个线程，它们共享进程的地址空间和系统资源
  - **资源分配：**
    - 进程拥有独立的资源，如内存空间、文件句柄等。进程之间的通信需要使用进程间通信（Inter-Process Communication，IPC）机制
    - 线程共享进程的资源，包括地址空间、文件描述符等。线程之间可以通过共享内存等方式进行通信
  - **创建和销毁：**
    - 进程的创建和销毁相对较慢，因为需要为其分配和释放独立的资源
    - 线程的创建和销毁相对较快，因为它们共享进程的资源，无需分配独立的内存空间
  - **独立性：**
    - 进程是相对独立的，一个进程的崩溃通常不会影响其他进程
    - 线程是进程的一部分，线程之间的错误可能会影响整个进程的稳定性
  - **切换开销：**
    - 进程切换的开销较大，涉及到上下文切换，需要保存和恢复整个进程的状态
    - 线程切换的开销相对较小，因为线程共享相同的地址空间，上下文切换时只需保存和恢复寄存器等少量状态
  - **通信方式：**
    - 进程间通信通常采用消息传递、管道、共享内存等方式
    - 线程间通信可以通过共享内存、互斥锁、信号量等机制实现
  - **适用场景：**

        - 进程适用于需要独立执行、相对独立的任务，例如运行一个应用程序
        - 线程适用于需要共享数据和相互协作的任务，例如图形界面应用程序
- **线程的特点：**
    - 不同进程之间可以并发执行，一个进程中的多个线程也能并发执行，甚至不同进程中的线程也能并发执行
    - 支持多处理机系统，进程只能运行在一个CPU上，但对于多线程进程，可以将进程中的多个线程分配到多个CPU上
    - 每个线程都有唯一的标识符和一个线程控制块，线程控制块记录了线程执行的寄存器和栈等现场状态
    - 不同的线程可以执行相同的程序，即同一个服务程序被不同的用户调用时，操作系统把它们创建成不同的线程
    - 同一个进程中的所有线程共享该进程所拥有的资源
    - 在单核系统中，各线程可交替使用CPU，在多核系统中，各线程可同时占用了不同的CPU
- **线程也具有三种基本状态：**
    - 执行状态
    - 就绪状态
    - 阻塞状态
- **线程控制块：TCB**
    - 线程标识符
    - 一组寄存器，包括程序计数器，状态寄存器，通用寄存器
    - 线程运行状态，用于描述线程正处于何种状态
    - 优先级
    - 线程专用存储区，线程切换时用于保存现场
    - 堆栈指针，用于过程调用时保存局部变量和返回地址
- 同一个进程中的所有线程都完全共享进程中的地址空间和全局变量，各个线程都可以访问进程地址空间的每个单元，所以一个线程可以读，写或清除另一个线程的堆栈
- **线程的终止：**
    - 线程被终止后并不立即释放它所占用的资源，只有当进程中的其他线程执行了分离函数后，被终止线程才与资源分离，此时的资源才能被其他线程利用
    - 被终止但未释放资源的线程仍可被其他线程调用，以使被终止线程重新恢复执行
- **线程的实现方式：**用户级线程，内核级线程和组合方式
    - 用户级线程(User-Level Thread,ULT)
        - 有关线程管理的工作都由应用程序在用户空间中完成，应用程序通过使用线程库设计成多线程程序
        - 优点：
            - 线程切换不需要转换到内核空间，节省了模式切换的开销
            - 调度算法可以是进程专用的，不同的进程可根据自身的需要，对自己的线程选择不同的调度算法
            - 用户级线程的实现与操作系统平台无关，对线程管理的代码是属于用户程序的一部分
        - 缺点：
            - 系统调用的阻塞问题，当线程执行一个系统调用时，不仅该线程被阻塞，而且进程内的所有线程都被阻塞
            - 不能发挥多处理机的优势，内核每次分配给一个进程的仅有一个CPU，因此进程中仅有一个线程能执行
    - 内核级线程(Kernel-Level Thread,KLT)
        - 线程管理的所有工作都在内核空间中实现
        - 优点：

        - 能发挥多处理机的优势，内核能同时调度统一进程中的多个线程并行执行
        - 如果进程中的一个线程被阻塞，内核可以调度该进程中的其他线程占用处理机，也可以运行其他进程中的线程
        - 内核支持线程具有很小的数据结构和堆栈，线程切换比较快，开销小
        - 内核本身也可以采用多线程技术，可以提高系统的执行速度和小路
    - 缺点：
        - 同一个进程中的线程切换，需要从用户态转到内核态中进行，系统开销较大，因为用户进程的线程在用户态运行，而线程调度和管理在内核态中进行
  - 组合方式
    - 内核支持多个内核级线程的建立，调度和管理，同时允许用户程序建立，调度和管理用户级线程
    - 一些内核级线程对应多个用户级线程，这是用户级线程通过**时分多路复用**内核级线程实现的
    - 同一个进程中的多个线程可以同时在多处理机上并行运行，且阻塞一个线程时不需要将整个进程阻塞
- **线程库：**为程序员提供创建和管理线程的API，实现方法有两种：
  - 在用户空间中提供一个没有内核支持的库，这种库的所有数据结构和代码都位于用户空间中
  - 由操作系统直接支持的内核级的一个库，库内的代码和数据结构都位于内核空间中，调用库会导致对内核的系统调用
- 目前有三种线程库：POSIX Pthreads，Windows API，Java
  - POSIX Pthreads作为POSIX标准的扩展，可以提供用户级或内核级的库
  - Windows API是用于Windosw系统的内核级线程库
  - Java线程API允许线程在Java程序中直接创建和管理
- **多线程模型：**
  - 多对一模型：将多个用户级线程映射到一个内核级线程，线程的调度和管理在用户空间中完成
    - 优点：线程管理是在用户空间中进行，因而效率比较高
    - 缺点：如果一个线程在访问内核时发生阻塞，则整个进程都被阻塞，任何时刻只有一个线程能访问内核，多个线程不能同时在多个处理机上运行
  - 一对一模型：将每个用户级线程映射到一个内核级线程上
    - 优点：当一个线程被阻塞后，允许调度到另一个线程运行，并发能力强
    - 缺点：没创建一个用户线程，就要创建一个内核线程，开销较大
  - 多对对模型：将n个用户线程映射到m个内核线程中，n大于等于m
    - 克服了上面两种的缺点又合并了优点

习题

- 进程获得CPU运行是通过调度得到的
- 操作系统是根据PCB(进程控制块)来对并发执行的进程进行控制和管理的
- 进程之间可能是无关的，也可能是有交互性的
- PCB常驻内存中，进程的个数取决于内存的大小
- 打印机是独占资源同一时间只能有一个进程(线程)使用
- I/O操作完成之前进程在等待结果，状态为阻塞态
- I/O操作完成后进程等待的事件已经就绪，变为就绪态

- 程序封闭性是指进程执行的结果只取决于进程本身，不受外界的影响，进程的执行速度不会改变进程的执行结果
- 高级语言编写的程序在使用内存时分为三个段：
  - 正文段：存放二进制代码和常量
  - 数据堆段：动态分配的存储区
  - 数据栈段：临时使用的局部变量
- 进程的就绪数目越多，争夺CPU的进程就越多，但只要就绪队列不空，CPU就总是在执行，所以CPU的效率不会因为就绪进程的数目变化而变化
- 对进程的管理和控制功能是通过执行各种原语来实现的
- 使用一个线程来处理整个系统的键盘输入
- 全局变量是对同一进程而言的，在不同的进程中是不同的变量，没有联系，进程间不能通过全局变量来通信
- **阻塞态进程等待的事件：**
  - I/O完成：进程在进行输入/输出操作时处于阻塞，等待数据的读取或写入完成
  - 等待信号：进程在等待接受一个信号被阻塞
  - 锁或互斥体：进程可能在试图获取一个已被其他进程占用的锁或互斥体时被阻塞
  - 条件变量：进程可能在等待某个条件变量为真时被阻塞
  - 消息队列或缓冲区非空：进程可能在等待消息队列中有新的消息或某个缓冲区不再为空时被阻塞
  - 定时器超时：进程可能在等待某个定时器超时时被阻塞
  - 子进程结束：父进程可能在等待子进程结束时被阻塞
  - 资源可用：进程可能在等待某个资源变为可用时被阻
- **进程之间的通信方式：**
  - **管道（Pipe）：**
    - 管道是一种半双工通信机制，通常用于具有亲缘关系的父子进程之间。管道有两种类型：匿名管道和命名管道。匿名管道只能在有亲缘关系的进程之间使用，而命名管道可以在没有亲缘关系的进程之间使用。
  - **消息队列（Message Queue）：**
    - 消息队列是一种通过消息进行通信的机制，适用于不同进程之间的通信。进程可以向消息队列发送消息，另一个进程则可以从中接收。消息队列允许异步通信，发送和接收进程之间的关系相对松散。
  - **信号量（Semaphore）：**
    - 信号量是一种用于控制对共享资源的访问的同步机制。进程可以通过信号量来进行互斥访问，避免对共享资源的竞争。信号量可用于进程同步和进程互斥。
  - **共享内存（Shared Memory）：**
    - 共享内存允许多个进程共享同一块内存区域。进程可以通过在共享内存中写入和读取数据来进行通信。共享内存通常是最快速的 IPC 方法之一，但需要谨慎处理同步和互斥问题。
  - **套接字（Socket）：**
    - 套接字是一种用于在不同主机或同一主机上的不同进程之间进行通信的机制。套接字可以用于实现网络通信，也可以用于本地进程间通信。套接字通常用于 TCP 或 UDP 协议。
  - **文件映射（File Mapping）：**
    - 文件映射允许一个或多个进程将同一个文件映射到它们的地址空间中，从而实现共享内存的效果。进程可以通过在文件映射中写入和读取数据来进行通信。
  - **消息传递（Message Passing）：**
    - 消息传递是一种通过发送和接收消息进行通信的机制。进程可以通过消息传递来进行异步通信，其中消息的发送和接收是独立的。

    - **信号（Signal）：**
        - 信号是一种进程间通信的简单方式，用于通知接收进程发生了某个事件。信号通常用于处理异步事件，例如进程的终止或某个条件的发生。
- 内核级线程的程序实体可以在内核态运行
- **导致创建进程的事件：**
    - **程序启动：**
        - 当用户或系统启动一个程序时，操作系统需要创建一个新的进程来执行该程序。这是最常见的创建进程的事件之一。
    - **系统初始化：**
        - 在操作系统启动时，通常会创建一些初始化进程，用于执行系统初始化和设置任务。这些进程在系统启动时就被创建。
    - **用户登录：**
        - 当用户通过登录认证成功登录到计算机系统时，操作系统通常会创建一个新的进程以支持用户的会话。
    - **后台任务启动：**
        - 当系统需要执行一些后台任务，如定时任务、系统维护任务等，操作系统可能会创建相应的进程来执行这些任务。
    - **处理外部事件：**
        - 操作系统可能在响应外部事件时创建新的进程，例如硬件中断、网络数据到达等。
    - **程序调用其他程序：**
        - 一个程序在执行过程中调用了另一个程序，可能会导致创建新的进程来执行被调用的程序。这种情况通常发生在进程间通信和协作的场景中。
    - **并行处理：**
        - 在需要并行处理任务的情况下，操作系统可能会创建多个进程以充分利用多核处理器的性能。
    - **服务请求：**
        - 当系统接收到服务请求时，例如网络请求、文件系统请求等，操作系统可能会创建新的进程来处理这些请求。
    - **用户交互：**
        - 当用户在图形用户界面（GUI）中启动应用程序、打开文件等操作时，操作系统可能会创建新的进程来处理用户请求。
- 进程是资源分配的基本单位，线程是处理机调度的基本单位
- 进程的代码段，进程打开的文件，进程的全局变量等都是进程的资源，只有进程中某线程的栈指针(包含在TCB中)属于线程
- 属于进程的资源线程之间可以共享，但是属于线程的栈指针是独享的，其他线程无法访问
- 管道是一种固定大小的缓冲区，管道的大小通常为内存的一页
- 多对一线程模型中，操作系统可能并不为每一个用户级线程建立一个TCB，而是映射到同一个内核级线程中共享同一个TCB
- 父进程可与子进程共享一部分资源，但不共享虚拟地址空间，创建子进程时为子进程分配虚拟地址空间
- 临界资源一次只能为一个进程所用
- 每个进程有自己独立的地址空间，在操作系统和硬件的地址保护机制下，进程无法访问其他进程的地址空间，必须借助于系统调用函数实现进程之间的通信

调度

概念

- **CPU调度：**从就绪队列中按照一定的算法选择一个进程并将处理机分配给它运行来实现进程并发地执行
- **调度的层次：**
  - 高级调度(作业调度)：从外存中处于后备队列中的作业挑选一个或多个，给它们分配内存，输入/输出设备等必要的资源，建立相应的进程，也就是**创建进程并将进程加入就绪队列中**，常用于多道批处理系统中，其他系统通常不需要
  - 中级调度(内存调度)：主要为了提高内存利用率和系统吞吐量，将那些暂时不能运行的进程调至外存等待，此时进程的状态称为挂起态，当它们具备运行条件且内存有空闲时再重新调入内存的就绪队列中，也就是**将暂时不能运行的进程挂起，能运行再加入到就绪队列中**
  - 低级调度(进程调度)：从就绪队列中选取一个进程，将CPU分配给它，也就是**将就绪队列中的一个进程切换为运行态**
- **三级调度的关系：**
  - 作业调度为进程活动做准备，进程调度使进程正常活动起来
  - 中级调度将暂时不能运行的进程挂起，中级调度处于作业调度和进程调度之间
  - 作业调度次数最少，中级调度次数略多，进程调度频率最高
  - 进程调度是最基本的，不可或缺的
- **调度的性能标准：**
  - CPU利用率：
    - $cpu的利用率 = \frac{cpu有效工作时间}{cpu有效工作时间+cpu空闲等待时间}$
  - 系统吞吐量：
    - 表示单位时间内CPU完成作业的数量
    - 长作业需要消耗较长的处理机时间，降低系统的吞吐量
    - 短作业需要消耗较短的处理机时间，提高系统的吞吐量
  - 周转时间：
    - 从作业提交到作业完成所经历的时间，
      $$周转时间 = 作业等待时间 + 在就绪队列中排队时间 + 处理机上运行时间 + 输入/输出操作所花费时间$$
      $$= 作业完成时间 - 作业提交时间$$
    - 平均周转时间 = 所有作业的周转时间/作业数量
    - $带权周转时间 = \frac{作业周转时间}{作业实际运行时间}$
    - 平均带权周转时间 = 所有作业周转时间/作业数量
  - 等待时间：
    - 进程处于等处理机的时间之和，等待时间越长，用户满意度越低，**衡量调度算法的优劣只需看等待时间**
  - 响应时间：
    - 从用户提交请求到系统首次产生响应所用的时间，在交互式系统中，一般采用响应时间作为衡量调度算法的准则

调度的实现

- 用于调度和分派CPU的组件称为**调度程序**，通常由三部分组成：排队器，分派器，上下文切换器
  - 排队器：
    - 将系统中的所有就绪进程按照一定的策略排成一个或多个队列
  - 分派器：
    - 依据调度程序所选的进程将其从就绪队列中取出，分配CPU，切换为运行态
  - 上下文切换器：
    - 对CPU进行进程切换时，有两对上下文的切换操作
    - 第一对：将当前进程的上下文保存到其PCB中，再装入待运行程序的上下文，以便待运行程序运行
    - 第二对：移出运行程序的上下文，将所选进程的CPU现场信息装入处理机的各个对应寄存器
    - 现在采用两组寄存器，一组供内核使用，一组供用户使用，上下文切换时只需要改变指针
- **不能进行调度的情况：**
  - 在处理中断的过程中
  - 进程在操作系统内核临界区中
  - 其他需要完全屏蔽中断的原子操作过程中
- **应该进行进程调度和切换的情况：**
  - 发生引起调度条件且当前进程无法继续运行下去时，可以马上进行调度与切换，也就是非剥夺调度
  - 中断处理结束或自陷处理结束后，返回被中断进程的用户态程序执行现场前，若请求调度标志置一，则马上进行进程调度与切换，也就是剥夺调度
- 进程切换往往在调度后立刻发生，现场切换时，操作系统内核将原进程的现场信息推入内核堆栈中来保存它们，并更新堆栈指针，内核从新进程的内核栈中装入新进程的现场信息，更新当前运行进程空间指针，重设PC寄存器，开始新的进程
- **进程调度方式：**
  - 非抢占调度方式，也称非剥夺方式，实现简单，系统开销小，适用于大多数的批处理系统，但不能用于分时系统和大多数的实时系统
  - 抢占调度方式，也称剥夺方式，允许调度程序根据某种原则去暂停正在执行的进程，将处理机分配给更为重要的进程，提高系统吞吐率和响应效率
- **空闲进程：**
  - 进程切换时，就绪队列中没有进程则运行空闲进程，空闲进程的优先级最低，空闲进程不需要CPU之外的资源，不会被阻塞
- **两种线程调度：**
  - 用户级线程调度：由进程中调度程序决定哪个线程执行
  - 内核级线程调度：内核对选中的线程赋予一个时间片，超过时间片，强制挂起该线程
  - 用户级线程切换在同一进程中进行，仅需少量的机器指令，内核级线程切换需要完成的上下文切换，修改内存映像，使高速缓存失效，导致延迟

调度算法

- **先来先服务调度算法(FCFS)：**适用于作业调度和进程调度，选择**最先**进入队列的进程
  - 属于不可剥夺算法，不能作为分时系统或实时系统
  - 算法简单，效率低
  - 对长作业有利，对短作业不利
  - 有利于CPU繁忙型作业，不利于I/O繁忙型作业
- **短作业优先调度算法(SJF)：**对短作业优先调度的算法，选择**运行时间最短**的进程
  - 对长作业不利，可能导致长作业饥饿
  - 不能保证紧迫性作业被及时处理
  - 作业多长短是根据用户所提供的估计执行时间而定
  - SJF调度算法的平均等待时间和平均周转时间最少
- **优先级调度算法：**适用于作业调度和进程调度，选择**优先级最高**的进程
  - 非抢占式优先级调度算法：即使更高优先级的进程就绪了也要等待现在的进程执行完毕
  - 抢占式优先级调度算法：有更高优先级的进程就绪立即暂停现在的进程转而让高优先级进程执行
  - 进程的优先级可分为：
    - 静态优先级：在创建进程时确定，取决于进程类型，进程对资源的要求，用户要求
    - 动态优先级：根据进程情况的变化动态调整优先级，取决于进程占用CPU时间的长短，就绪进程等待CPU时间的长短
  - 优先级的参考原则：
    - 系统进程>用户进程
    - 交互型进程>非交互型进程
    - I/O型进程>计算型进程，I/O型进程指的是那些频繁使用I/O设备的进程，计算型进程指的是频繁使用CPU的进程(很少使用I/O设备)
- **高响应比优先调度算法：**主要用于作业调度，是FCFS和SJF的平衡，选择**响应比最高**的进程，
  响应比 $=\frac{\text{已等待时间}+\text{要求服务时间}}{\text{要求服务时间}}$
  - 作业的等待时间相同时，要求服务时间越短，响应比越高，有利于短作业
  - 要求服务时间相同时，作业的响应比由已等待时间决定，等待时间越长，响应比越高，类似于FCFS
  - 响应比随等待时间变大，有利于长作业
- **时间片轮转调度算法：**适用于分时系统，按FCFS排列，每个进程分配一个时间片
  - 时间片过大使所有进程能在一个时间片中完成，则时间片轮转调度算法就变成FCFS
  - 时间片过小导致进程间频繁切换，使处理机的开销增大
  - 时间片取决于系统的响应时间，就绪队列中的进程数目和系统的处理能力
- **多级队列调度算法：**
  - 设置多个就绪队列，不同类型或性质的进程固定分配到不同的就绪队列中
  - 每个队列实施不同的调度算法
  - 同一队列中的进程可以设置不同的优先级
  - 不同队列本身可以设置不同的优先级
- **多级反馈队列调度算法：**结合了时间片轮转调度算法和优先级调度算法，动态调整**进程优先级和时间片大小**
  - **算法思想：**

        - 设置多个就绪队列，每个队列的优先级不同，第一级队列的优先级最高
        - 每个队列的进程时间片不同，优先级越高，时间片越小
        - 每个队列采用FCFS，新进程都是从末尾插入，如果一个进程在一个时间片内不能完成，则插入下一级队列的末尾
        - 按队列优先级调度，只有前一级队列为空时才能执行本队列，如果在进程执行过程中高优先级队列中插入进程则立即暂停将正在执行的进程插入末尾再让高优先级的进程先执行
    - **优点：**
        - 终端型作业用户：短作业优先
        - 短批处理作业用户：周转时间较短
        - 长批处理作业用户：经过前面队列的部分执行，不会长期得不到执行

进程切换

- 创建，撤销及要求由系统设备完成的I/O操作，都是利用系统调用进入内核，再由内核中的相应处理程序来完成的
- **上下文切换：**切换CPU到另一个进程需要保存当前进程状态并恢复另一个进程的状态
- 上下文：某一个时刻CPU寄存器和程序计数器的内容
- 上下文切换的流程：
    - 挂起一个进程，保存CPU上下文，包括程序计数器和其他寄存器
    - 更新PCB信息
    - 把进程的PCB移入相应的队列，如就绪，在某事件阻塞等队列
    - 选择另一个进程执行，并更新其PCB
    - 跳转到新进程PCB中的程序计数器所指向的位置执行
    - 恢复处理机上下文
- 每次上下文切换都需要纳秒量级的时间，上下文切换对系统来说意味着消耗大量的CPU时间，有些处理器提供多个寄存器组，这样上下文切换就只需要简单改变当前寄存器组的指针
- **模式切换：**用户态和内核态之间的切换，没有改变当前的进程
- 上下文切换只发生在内核态

习题

- 时间片轮转的主要目的：使多个交互的用户能够得到及时响应
- 时间片轮转增加了系统开销，吞吐量和周转时间都不如批处理
- 交互式系统为改善用户的响应时间大多数采用时间片轮转调度算法
- CPU繁忙型作业需要使用很长时间CPU时间，类似于长作业，I/O型作业需要频繁请求I/O操作，CPU时间使用短，使用完成就要重新调度，类似于短作业
- 作业是从用户角度出发，由用户提交，以用户任务为单位
- 进程是从操作系统的角度出发，由系统生成，是操作系统的资源分配和独立运行的基本单位
- 分时系统中，响应时间与时间片和用户数成正比
- 中断向量：存放中断服务程序的入口地址

- 当进程处于临界区时，说明进程正在占用CPU，只要不破坏临界资源的使用规则，就不会影响CPU的调度，所以可以调度
- 计算响应比时注意等待时间 $=$ 调度的时刻 $-$ 到达就绪队列的时刻
- 响应比越高，优先级越高
- 具有两道作业的批处理系统代表能够将内存中能同时放入2个作业

同步与互斥

- **临界资源：**一次仅允许一个进程使用的资源
- **临界区：**进程中访问临界资源的那段代码
- **临界资源的访问过程：**
  - 进入区：在进入区检查可否进入临界区，若能，则设置正在访问临界区的标志
  - 临界区：进程中访问临界资源的那段代码，也称临界段
  - 退出区：将正在访问临界区的标志清除
  - 剩余区：进程中代码的其他部分
- **同步：**也称直接制约关系，指协调多个进程之间的**工作次序**而等待，传递信息所产生的制约关系
- **同步遵循的准则：**
  - 空闲让进：临界区空闲时，可以允许一个请求进入临界区的进程立即进入临界区
  - 忙则等待：已有进程进入临界区后，其他请求进入的进程必须等待
  - 有限等待：对请求访问的进程，保证在有限时间内进入临界区
  - 让权等待：当进程不能进入临界区时，必须立即释放CPU，防止进程忙等待
- **互斥：**也称间接制约关系，当一个进程进入临界区使用临界资源时，另一个进程必须等待，占用临界资源的进程退出临界区后，另一个进程才允许访问此临界资源

实现互斥的方法

- **软件实现方法：Peterson's Algorithm**

```
P1进程：                                                P2进程：
flag[i] = true;                                         flag[j] = true;
turn    = j;                                            turn    = i;
while (flag[j] == true && turn == j){};                 while (flag[i] == true && turn == i){};  进入区
critical section;                                       critical section;                        临界区
flag[i] = false;                                        flag[j] = false;                         退出区
remainder section;                                      remainder section                        剩余区
```

- 用`flag`来表示其他进程是否想要进入临界区，用`turn`来保证要想进入临界区必须对方的`flag`为false
- **硬件实现方法：**

- 中断屏蔽方法：CPU只在发生中断时引起进程切换，所以屏蔽中断能够保证当前运行的进程让临界区代码顺利执行完

```
taskENTER_CRITICAL();
临界区代码;
taskEXIT_CRITICAL();
```

  - 硬件方法限制了CPU交替执行程序的能力，执行的效率会明显降低
- 硬件指令方法：
  - TestAndSet指令：这条指令是原子操作，即执行该代码时不允许被中断，功能为读出指定标志并设置为true

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

  - 为每个临界资源设置一个共享布尔变量lock，true表示正在被使用，进程进入临界区前先用TestAndSet检查

```
while (TestAndSet(&lock) == true){}
临界区代码
lock = false;
剩余区代码
```

  - Swap指令：交换两个字(或字节)的内容

```
void Swap(bool *a, bool *b)
{
    bool temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

  - 为每个临界资源设置一个全局共享布尔变量lock，在每个进程中设置一个局部变量key，进入临界区前先利用Swap交换key与lock的值，然后在检查key的值

```
key = true;
while(key == true)
{
    Swap(&lock, &key);
}
临界区代码;
lock = false;
剩余区代码;
```

- **以上代码仅表示该硬件方法的实现思想，实际上是由硬件逻辑直接实现的，不会被中断**
- 硬件方法的优点：
  - 适用于任意数目的进程，不管是单处理机还是多处理机

        - 简单，容易验证其正确性
        - 可以支持进程内有多个临界区，只需为每个临界区设置一个布尔变量
    - 硬件方法的缺点：
        - 进程等待进入临界区时要消耗CPU时间，不能实现**让权等待**
        - 从等待进程中随机选择一个进入临界区，有的进程可能导致“饥饿”现象

互斥锁

- 互斥锁通常使用**硬件机制**来实现，且获得锁和释放锁操作都是原子操作
- 互斥锁缺点：忙等待，常用于多处理器系统

```
acquire()
{
    while (!available){};
    available = false;
}

release()
{
    available = true;
}
```

信号量

- 信号量可用来解决互斥和同步的问题，只能被两个标准的原语wait(S)和signal(S)访问，记为“P操作”和“V操作”
- **整形信号量**：使用一个表示资源数目的整型量S

```
wait(S)
{
    while(S<=0){};
    S = S - 1;
}
signal(S)
{
    S = S + 1;
}
```

- 只要S<=0，就会一直循环等待，不遵循“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**：使用一个表示资源数量的整型变量value和一个进程链表L

```
typedef struct{
    int value;
    struct process *L;
}semaphore;

void wait(semaphore S)
{
    S.value --;
```

```
    if (S.value < 0)                    表示资源已分配完
    {
        add this process to S.L;        将进程放入等待队列
        block(S.L);                     进程自我阻塞
    }
}

void signal(semaphore S)
{
    S.value ++;
    if (S.value <= 0)                   表示有进程在等待
    {
        remove a process P from S.L;    将等待的进程移出等待队列
        wakeup(P);                      唤醒等待的进程
    }
}
```

- **利用信号量实现同步：**

```
semaphore S = 0;
P1()
{
    代码段X;
    V(S);   代码段X已经执行完成，可以进行下一步
}

P2()
{
    P(s);   请求信号量，等待X执行完再执行Y
    代码段Y;
}
```

- **利用信号量实现互斥：**

```
semaphore S = 1;
P1()
{
    其他代码;
    P(S);
    进程P1的临界区;
    V(S);
    其他代码;
}

P2()
{
    其他代码;
    P(S);
    进程P2的临界区;
    V(S);
    其他代码;
}
```

- **利用信号量实现前驱关系：**比如S2和S3的进行需要S1完成，S4的进行需要S的完成

```
semaphore a1=a2=b1=0;
S1()
{
    其他代码;
    V(a1);
    V(a2);      S1完成释放a1和a2
}

S2()
{
    其他代码;
    P(a1);      等待S1完成
}

S3()
{
    其他代码;
    P(a2);      等待S1完成
    其他代码;
    V(b1);
}

S4()
{
    其他代码;
    P(b1);      等待S2完成
    其他代码;
}
```

- **FreeRtos中实现的数据结构：**

```
typedef struct QueueDefinition
{
    int8_t * pcHead;           /*头指针*/
    int8_t * pcWriteTo;        /*尾指针*/

    union                      /*联合，共享内存，可以用作队列，也可以用作信号量*/
    {
        QueuePointers_t xQueue;
        SemaphoreData_t xSemaphore;
    } u;

    List_t xTasksWaitingToSend;                /*一个链表用来存放等待往队列或信号量里放数据的阻塞的任务*/
    List_t xTasksWaitingToReceive;             /*一个链表用来存放等待往队列或信号量里读数据的阻塞的任务*/

    volatile UBaseType_t uxMessagesWaiting;    /*当前队列中的任务数量*/
    UBaseType_t uxLength;                      /*能够存放的任务个数*/
    UBaseType_t uxItemSize;                    /*存放任务的大小*/

    volatile int8_t cRxLock;                   /*互斥锁*/
    /*< Stores the number of items received from the queue (removed from the queue) while the queue was locked.  Set to queueUNLOCKED when the queue is not locked. */
    volatile int8_t cTxLock;                   /*互斥锁*/
```

```
    /*< Stores the number of items transmitted to the queue (added to the queue) while the queue was locked.  Set to queueUNLOCKED when the queue is not locked. */
} xQUEUE;
```

- **FreeRtos中请求信号量(部分)P操作：**

```
taskENTER_CRITICAL();
const UBaseType_t uxSemaphoreCount = pxQueue->uxMessagesWaiting;
if( uxSemaphoreCount > ( UBaseType_t ) 0 )
{
    traceQUEUE_RECEIVE( pxQueue );

    /* Semaphores are queues with a data size of zero and where the
     * messages waiting is the semaphore's count.  Reduce the count. */
    pxQueue->uxMessagesWaiting = uxSemaphoreCount - ( UBaseType_t ) 1;  //请求信号量

    #if ( configUSE_MUTEXES == 1 )
    {
        if( pxQueue->uxQueueType == queueQUEUE_IS_MUTEX )  //互斥
        {
            /* Record the information required to implement
             * priority inheritance should it become necessary. */
            pxQueue->u.xSemaphore.xMutexHolder = pvTaskIncrementMutexHeldCount();
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    #endif /* configUSE_MUTEXES */

    /* Check to see if other tasks are blocked waiting to give the 判断其他任务是否等待释放信号量而阻塞
     * semaphore, and if so, unblock the highest priority such task. */
    if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToSend ) ) == pdFALSE )
    {
        if( xTaskRemoveFromEventList( &( pxQueue->xTasksWaitingToSend ) ) != pdFALSE )
        {
            queueYIELD_IF_USING_PREEMPTION();
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    taskEXIT_CRITICAL();
    return pdPASS;
}
taskEXIT_CRITICAL();
```

- **FreeRtos中释放信号量(部分)V操作：**

```
taskENTER_CRITICAL();
pxQueue->uxMessagesWaiting = uxMessagesWaiting + ( UBaseType_t ) 1;
if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToReceive ) ) == pdFALSE )
{
    if( xTaskRemoveFromEventList( &( pxQueue->xTasksWaitingToReceive ) ) != pdFALSE )
    {
        /* The unblocked task has a priority higher than
         * our own so yield immediately.  Yes it is ok to do
         * this from within the critical section - the kernel
         * takes care of that. */
        queueYIELD_IF_USING_PREEMPTION();
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
else if( xYieldRequired != pdFALSE )
{
    /* This path is a special case that will only get
     * executed if the task was holding multiple mutexes and
     * the mutexes were given back in an order that is
     * different to that in which they were taken. */
    queueYIELD_IF_USING_PREEMPTION();
}
else
{
    mtCOVERAGE_TEST_MARKER();
}
taskEXIT_CRITICAL();
```

管程

- 信号量机制的缺点：进程自备同步操作，P(S)、V(S)操作大量分散在各个进程中，还要仔细安排P(S)的顺序，不易管理，易发生死锁
- 引入**管程**的目的：
  - **把分散在各进程中的临界区集中起来进行管理**
  - 防止进程的非法同步操作
  - 便于利用高级语言来编程，便于验证程序正确性
- 管程相当于一个类，包含：
  - **局部共享变量和条件变量**组成管程内的数据结构
  - 对数据结构进行操作的一组过程
  - 对数据结构进行初始化的语句
  - 一个互斥锁(由编译器添加)

```
monitor Demo  定义一个叫做Demo的管程
{
    共享数据结构S;

```

```
    condition x;  一系列条件变量
    ......

    init_code()
    {
       S = 5; 对共享数据结构进行初始化
    }

    take_away()
    {
        对共享数据结构s的一系列处理;
        S--;
        if(x的资源不满足)
        {
            x.wait();    资源不满足，进入等待队列，让出管程
        }
        其他代码;
    }

    give_back()
    {
        对共享数据结构s的一系列处理;
        S++;
        if(有进程在等待条件变量x并且条件变量x满足){
            x.signal();    释放因条件变量x而阻塞的进程
        }
        其他代码;
    }
}
```

- 管程内的共享数据结构只能被管程内的过程所访问
- 一个进程只有通过调用管程内的过程才能进程管程访问共享数据结构
- 每次仅允许一个进程进入管程，从而实现互斥
- 对于**条件变量**：
  - 当一个进程进入管程之后被阻塞，则如果该进程不释放管程会导致其他进程无法进入管程，所以需要使用条件变量来将阻塞的进程放入一个等待队列并释放管程
  - 将阻塞原因定义为条件变量condition，通常阻塞原因有多个，所以有多个条件变量
  - 每个条件变量都保存一个等待队列，用于记录因该条件变量而阻塞的所有进程
  - 对条件变量只能进行两种操作：wait和signal
  - **x.wait**：当x对应的条件不满足进入阻塞前，调用x.wait将自己插入x条件变量的等待队列，并释放管程
  - **x.signal**：x对应的条件发生变化，由其他进程调用x.signal唤醒一个因x条件而阻塞的进程

经典同步互斥问题

- **生产者-消费者问题**：
  - 生产者和消费者共享一个大小为n的缓存区
  - 缓冲区不满生产者就能放入消息，缓冲区不空消费者就能读取消息
  - 同时只能有一个生产者或一个消费者使用缓冲区

```
semaphore mutex = 1;
semaphore empty = n;
semaphore full  = 0;

producer()
{
    while(1)
    {
        produce an item in nextp;
        p(empty);
        p(mutex);
        add nextp to buffer;
        V(mutex);
        V(full);
    }
}

consumer()
{
    while(1)
    {
        P(full);
        P(mutex);
        remove an item from buffer;
        V(mutex);
        V(empty);
        consume the item;
    }
}
```

- **生产者消费者问题：**
  - 两个生产车间和一个装配车间，一个车间生产A，一个车间生产B
  - 两个车间每生产一个放在装配车间上的F1，F2装配线上，F1和F2上均可放10个(生产者和消费者共享大小为10的缓冲区)
  - 装配工人从F1和F2上分别取A和B装配成产品，不能同时放和取(同时只能有一个生产者或消费者访问缓冲区)

```
empty_A = 10; empty_B = 10; mutex_A = 1; mutex_B = 1; full_A = 0; full_B = 0;
process A()                 process B()
{                           {
    while(1)                    while(1)
    {                           {
        P(empty_A);                 P(empty_B);
        P(mutex_A);                 P(mutex_B);
        将A放在F1上;                 将B放在F2上;
        V(mutex_A);                 V(mutex_B);
        V(full_A);                  V(full_B);
    }                           }
}                           }

compose()
{
    while(1)
    {
        P(full_A);
        P(mutex_A);
```

```
        取A;
        V(mutex_A);
        V(empty_A);

        P(full_B);
        P(mutex_B);
        取B;
        V(mutex_B);
        V(empty_B);
        将A和B合成
    }
}
```

- **生产者消费者问题：**
  - 有若干小和尚和老和尚，小和尚负责从井里打水并放入水缸中，老和尚从水缸中取水喝
  - 共有三个桶，每次只能用一个桶从井里打水或从缸中打水，缸中一共能放10桶水

```
bucket = 3; empty_water = 10; full_water = 0; well = 1; water_tank = 1;
young monk()
{
    P(empty_water);
    P(bucket);
    P(well);
    从井里打水;
    V(well);
    P(water_tank);
    将水倒入缸中
    V(water_tank);
    V(bucket);
    V(full_water);
}

old monk()
{
    P(full_water);
    P(bucket);
    P(well);
    从水缸里打水
    V(well);
    V(bucket);
    V(empty_water);
}
```

- **生产者消费者问题：(资源不限量，两个生产者互斥，一个缓冲区互斥)**
  - 有一条路，可以从北到南，从南到北，中间有一个桥，只允许一辆车通过
  - 允许同方向多辆车一起通过

```
int countN_S = 0; int countS_N = 0;
semaphore mutexN_S = 1; semaphore mutexS_N = 1;
semaphore bridge = 1;
StoN()                              NtoS()
{                                   {
    P(mutexS_N);                        P(mutexN_S);
    if(countS_N == 0)                   if(countN_S == 0)
```

```
    {                                   {
        P(bridge);                          P(bridge);
    }                                   }
    countS_N ++;                        countN_S ++;
    V(mutexS_N);                        V(mutexN_S);
    through the bridge;                 through the bridge;
    P(mutexS_N);                        P(mutexN_S);
    countS_N --;                        countN_S --;
    if (countS_N == 0)                  if (countN_S == 0)
    {                                   {
        V(bridge);                          V(bridge);
    }                                   }
    V(mutexS_N);                        V(mutexN_S);
}                                   }
```

- **生产者消费者问题：(两种资源有最大值)**
  - 自行车装配线上有一个箱子，里面有N个位置，每个位置可以放一个车架或一个车轮
  - 一共三名工人，第一个生产车架放进箱子里，第二个生产车轮放进箱子里，第三个拿一个车架和两个车轮装配
  - 要求不死锁

```
frame_max_num = N-2; wheel_max_num = N-1; frame = 0; wheel = 0; empty = N;
worker1()                                           worker2()
{                                                   {
    while(1)                                            while(1)
    {                                                   {
        produce a frame;                                    produce a wheel;
        P(frame_max_num);                                   P(wheel_max_num);
        P(empty);                                           P(empty);
        put the frame in the box;                           put the whell in the box;
        V(frame);                                           V(wheel);
    }                                                   }
}                                                   }
worker3()
{
    while(1)
    {
        P(frame);
        get a frame;
        V(empty);
        V(frame_max_num);

        P(wheel);
        P(wheel);
        get two wheels;
        V(empty);
        V(empty);
        V(wheel_max_num);
        V(wheel_max_num);
        compose a bike;
    }
}
```

- **生产者消费者问题：(即能当生产者又能当消费者)**

```
produce_consumer()
{
    if (empty == 1)
    {
        P(empty);
        P(mutex);
        product one;
        V(mutex);
        V(full);
    }
    if (full == 1)
    {
        P(full);
        P(mutex);
        consume one;
        V(mutex);
        V(empty);
    }
}
```

- **生产者消费者问题2：**
  - 桌子上一个盘子，每次只能往其中放入一个水果
  - 爸爸放苹果，妈妈放橘子
  - 儿子等着吃盘子里的橘子，女儿等着吃盘子里的苹果
  - 只有盘子为空时，才能往盘子里放水果
  - 只有当盘子中有想要的水果时才从盘子中取出

```
semaphore plate = 1, apple = 0, orange = 0;
dad()
{
    while(1)
    {
        P(plate);
        put the apple on the plate;
        V(apple);
    }
}

mom()
{
    while(1)
    {
        P(plate);
        put the orange on the plate;
        V(orange);
    }
```

```
}

son()
{
    while(1)
    {
        P(orange);
        V(plate);
        eat the orange ;
    }
}

daughter()
{
    while(1)
    {
        P(apple);
        V(plate);
        eat the apple;
    }
}
```

- **读者写者问题：**
  - 读者和写者共享一个文件，允许多个读者读文件
  - 只允许一个写者写文件
  - 写者在写完前其他读者和写者不能进行读和写
  - 写者在写之前，其他的写者或读者应该退出

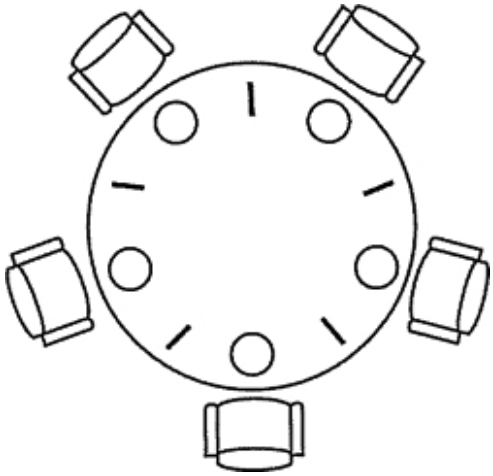
```
int count = 0;
semaphore mutex = 1;
semaphore rw = 1;
semaphore w = 1;

writer()
{
    while (1)
    {
        P(w);
        p(rw);
        writing;
        V(rw);
        V(w);
    }
}

reader()
{
    while(1)
    {
        P(w);            读者与写者之间的互斥
        P(mutex);        互斥访问count
        if (count == 0) 仅在第一次的时候请求rw
        {
```

```
            P(rw);       多个读者和写者之间的互斥
        }
        count ++;
        V(mutex);
        V(w);
        reading;
        P(mutex);
        count --;
        if (count == 0)
        {
            V(rw);
        }
        V(mutex);
    }
}
```

- **同步问题：**
  - 现有三个进程P1，P2，P3，需要轮流输入数据a，b，c，然后进行计算，输入设备互斥使用
  - P1:x=a+b; P2:y=a*b; P3:z=y+c-a;
  - 计算完后由P1进行打印

```
P1->P2->P3 设置三个信号量 S1 = 1; S2 = 0; S3 = 0;
x需要b，设置s_b = 0，y需要a，设置s_a = 0，z需要y，设置s_y = 0; 打印需要z，设置s_z = 0;
P1()
{
    P(S1);
    write a;  写完再释放
    V(S2);
    P(s_b);
    x = a + b;
    P(s_z); z有代表y有
    printf x,y,z;
}

P2()
{
    P(S2);
    write b;
    V(S3);
    V(s_b);
    y=a*b;
    V(s_y);
}

P3()
{
    P(S3);
    write c;
    P(s_y);
    z = y + c - a;
    V(s_z);
}
```

- **哲学家进餐问题：**
  - 5个哲学家围着圆桌思考，每个哲学家左边和右边都有一个筷子
  - 当哲学家饿时才试图拿起左右两根筷子，如果左右都有则进行进餐，否则进行等待
  - 进餐完毕后放下筷子继续思考



```
semaphore chopstick[5] = {1,1,1,1,1}; 哲学家分别为0-4，哲学家i左边筷子的编号为i，右边为i+1
Pi()
{
    while(1)
    {
        P(chopstick[i]);      拿起左边筷子
        P(chopstick[i+1]%5); 拿起右边筷子
        eat;
        V(chopstick[i+1]%5);
        V(chopstick[i]);
        think;
    }
}
```

  - 上述算法中，如果5位哲学家都想进餐并都执行到拿起左边筷子那一步，则会导致死锁
  - 加入一些限制条件：
    - 至多允许4位哲学家同时进餐
    - 仅当一位哲学家左右筷子都可用时才允许拿起筷子
    - 对哲学家进行编号，奇数的哲学家先拿左边的，偶数的哲学家先拿右边的
  - 对于第二种：

```
semaphore chopstick[5] = {1,1,1,1,1};
semaphore mutex = 1;
Pi()
{
    while(1)
    {
        P(mutex);
        P(chopstick[i]);
        P(chopstick[i+1]%5);
        V(mutex);
        eat;
        V(chopstick[i+1]%5);
```

```
        V(chopstick[i]);
        think
    }
}
```

- **吸烟者问题：**
  - 三个抽烟者和一个供应者
  - 抽烟需要三种东西：烟草，卷纸，胶水
  - 供应者每次将两种材料放在桌子上，后面放另外两种，一直重复
  - 三个抽烟者都有不同的材料
  - 只有抽烟者抽完了供应者才能放材料

```
int num = 0;
semaphore offer1 = 0;
semaphore offer2 = 0;
semaphore offer3 = 0;
semaphore finish = 0;
process P1()
{
    while(1)
    {
        num ++;
        num = num%3;
        if (num == 0)
        {
            V(offer1);
        }
        else if (num == 1)
        {
            V(offer2);
        }
        else
        {
            V(offer3);
        }
        P(finish);
    }
}

smoker1()
{
    while(1)
    {
        P(offer1);
        smoking;
        V(finish);
    }
}

smoker2()
{
    while(1)
    {
        P(offer2);
```

```
        smoking;
        V(finish);
    }
}

smoker3()
{
    while(1)
    {
        P(offer3);
        smoking;
        V(finish);
    }
}
```

- **数量差问题：**
  - 一个仓库中放入A和B
  - 每次只能往仓库中放一种
  - A-B>M-1
  - B-A>N-1

```
Sa:A剩余能放入的件数，Sb:B剩余能放入的件数，mutex:互斥访问仓库
produce A()
{
    P(Sa);
    P(mutex);
    将A放入仓库中
    V(mutex);
    V(Sb);
}

produce A()
{
    P(Sb);
    P(mutex);
    将B放入仓库中
    V(mutex);
    V(Sa);
}
```

- 访问临界资源的那段代码称为**临界区**，也就是P/V操作，加减锁
- 可重入的程序代码一次可供多个进程使用
- 执行P操作时的进程处于运行态
- 不允许修改的代码称为**可重入代码**，也称纯代码，即允许多个进程同时访问的代码
- PV操作是一种低级的进程通信原语，不是系统调用
- 银行家算法是避免死锁的算法
- 信箱通信是一种间接通信
- 只有一个进程在离开管程时才能调用signal()操作
- “让权等待”准则在互斥准则中不一定需要实现

死锁

- **死锁：**指多个进程因竞争资源而造成的一种互相等待，若无外力作用，这些进程都将无法向前推进
- **死锁产生的原因：**
  - 系统资源的竞争：对不可剥夺资源的竞争才会导致死锁，比如磁带机，打印机
  - 进程推进顺序非法：进程在运行过程中，请求和释放资源的顺序不当，双方都在等待对方的资源而进入死锁
- **死锁产生的必要条件：**只要其中任意一个条件不成立则死锁不会发生
  - 互斥条件：进程要求对所分配的资源进行排他性使用
  - 不剥夺条件：进程所获得的资源在未使用完之前，不能被其他进程强行夺走，只能由获得该资源的进程自己来释放
  - 请求并保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已被其他进程占用，请求进程被阻塞并不释放已获得的资源
  - 循环等待条件：存在一种进程资源的循环等待链，链中每个进程已获得的资源同时被链中下一个进程所请求，但只是必要条件，如果同类资源数大于1，则该圈可以被打破

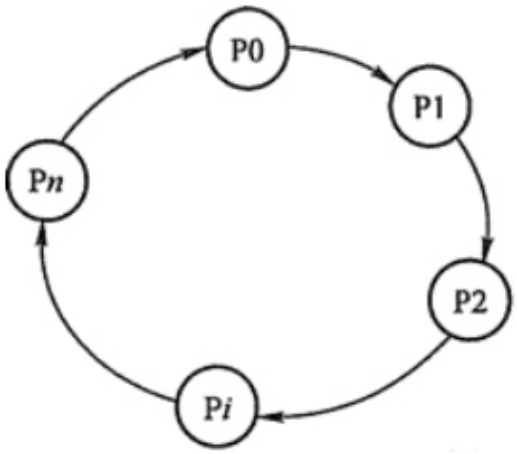


图2-15 循环等待

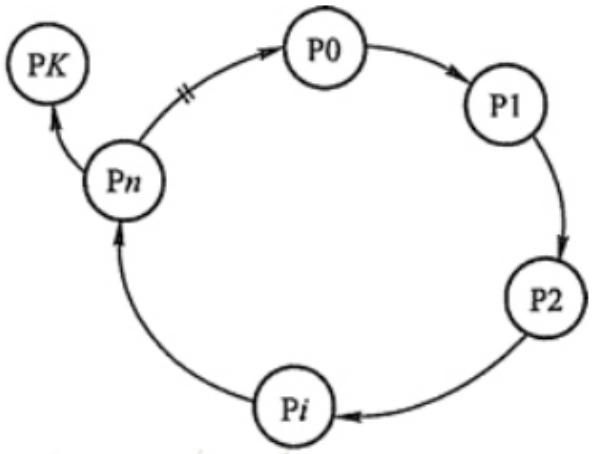


图2-16 满足条件但无死循环

- 死锁的处理策略：
  - 死锁预防：设置某些限制条件，破坏产生死锁的4个必要条件中的一个或多个
  - 避免死锁：在资源的动态分配过程中，用某种方法防止系统进入不安全状态
  - 死锁的检测及解除：不采取任何限制性措施，允许发生死锁，及时检测出死锁并解除死锁

	资源分配策略	各种可能模式	主要优点	主要缺点
死锁预防	保守，宁可资源闲置	一次请求所有资源，资源剥夺，资源按序分配	适用于突发式处理的进程，不必进行剥夺	效率低，进程初始化时间长，剥夺次数多，不便灵活申请新资源
死锁避免	中等，在运行时判断是否可能死锁	寻找可能的安全允许顺序	不必进行剥夺	必须知道将来的资源需求，进程不能被长时间阻塞
死锁检测	宽松，只要允许就分配资源	定期检查死锁是否已经发生	不延长进程初始化时间，允许对死锁进行现场处理	通过剥夺解除死锁，造成损失

死锁预防

- 通过破坏死锁产生的必要条件
- **破坏互斥条件：**
  - 若允许系统资源都能共享使用则不会出现死锁，但临界资源只能进行互斥访问，所以破坏互斥条件来预防死锁的方法不太可行
- **破坏不可剥夺条件：**
  - 当一个已保持了某些不可剥夺资源的进程请求新的资源被阻塞时，该进程必须释放已经保持的**所有资源**，之后再重新申请
  - 释放已获得的资源可能造成前一阶段工作的失效，反复地申请和释放资源会增加系统开销，降低系统吞吐率量
  - 这种方法常用于状态易于保存和恢复的资源，如CPU的寄存器及内存资源，一般不能用于打印机之类的资源
- **破坏请求并保持条件：**
  - 采用**预先静态分配方法**：即进程在运行前一次申请完它所需要的全部资源，在所有资源未满足前，不能投入运行
  - 一旦投入运行，代表所有资源都能满足，则不会再请求新的资源，也就不会进入死锁
  - 系统资源被严重浪费，其中有些资源可能仅在运行初期或运行快结束时才使用，甚至根本不使用
  - 会导致“饥饿”现象，由于个别资源长期被其他进程占用，导致进程一直不能运行
- **破坏循环等待条件：**
  - 采用**顺序资源分配法**，首先给系统中的资源编号，规定每个进程必须按编号递增的顺序请求资源，同类资源一次申请完(同编号的资源全部拿走)，所以进程申请的资源编号只能变大，不能向前申请，所以也就不会出现循环
  - 限制了新类型设备的增加
  - 作业使用资源的顺序和系统规定的顺序不同导致资源浪费
  - 给用户编程带来麻烦

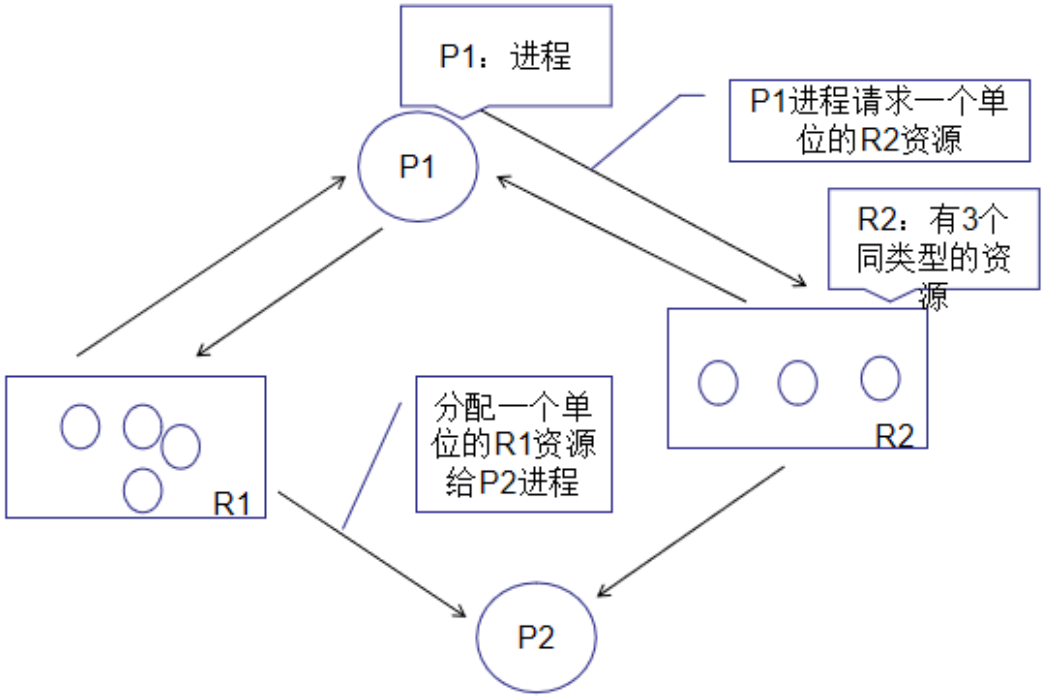
死锁避免

- 并不事先采取某种限制措施破坏必要条件，而是在资源动态分配过程中，防止系统进入不安全状态，以避免发生死锁

- **安全状态：**系统能按某种进程推进顺序为每个进程分配其所需的资源，直至满足每个进程对资源的最大需求，使每个进程都可顺序完成，若系统无法找到一个安全序列，则称系统处于不安全状态
- **系统安全状态：**
  - 允许进程动态地申请资源，但系统在进行资源分配之前，先计算此次分配的安全性
  - 若此次分配不会导致系统进入不安全状态，则允许分配，否则让进程等待
  - 系统进入不安全状态后，**可能**进入死锁状态，只要系统处于安全状态，系统便可避免进入死锁状态
- **银行家算法：**
  - 可利用资源向量Available：所有资源都可用数的矩阵
  - 最大需求矩阵Max：每一行代表一个进程对共m个资源的需求数，共有n个进程，则有n行，所以为n*m的矩阵
  - 分配矩阵Allocation：每一行代表一个进程对共m个资源已占用数，共有n个进程，则有n行，所以为n*m的矩阵
  - 需求矩阵Need：每一行代表一个进程对共m个资源还需要的个数，共有n个进程，则有n行，所以为n*m的矩阵
  - Need = Max - Allocation
- **算法描述：**
  - 设Request是进程P的请求向量
  - 若Request <= Need，则下一步，否则认为出错，因为所需要的数目已超过最大需求数
  - 若Request <= Available，则下一步，否则认为出错，因为资源不够，进行等待
  - 系统重新计算各个矩阵：
    - Available = Available - Request
    - Need = Need - Request
    - Allocation = Allocation + Request
  - 再执行系统**安全算法**，检查资源分配后，系统是否处于安全状态，若安全，则将资源分配给P，否则，恢复计算过的矩阵，P等待
- **安全算法：**
  - 设置工作向量Work，表示系统中可用资源数，初始时Work = Available
  - 初始时安全序列为空，从Need中寻找符合要求的进程P：
    - 该进程不在安全序列中
    - 该行小于或等于Work
  - 找到后将该进程P加入到安全序列，执行下一步，如果没找到则执行最后一步
  - 进程P进入安全序列后，可顺利执行， 直至完成，释放分配给它的所有资源，所以Work = Work + Allocation
  - 若此时安全序列中已有所有进程，则系统处于安全状态，否则系统处于不安全状态

死锁检测与解除

- **资源分配图：**
  - 圆圈代表一个进程
  - 方框代表一种资源
  - 方框内的圆圈代表一种资源有多少个
  - 由方框指向圆圈的有向线段代表已分配的资源数量
  - 由圆圈指向方框的有向线段代表进程请求的资源数量



- **死锁定理：**当且仅当某一状态的资源分配图是不可完全简化的，那么该状态进入死锁
- **简化资源分配图：**
  - 判断每个资源已分配的数量和剩余的数量，判断能不能满足某个进程对所有资源数量的请求
  - 如果能找到一个，则运行该进程并运行完成后进行释放寻找下一个满足条件的进程
- **死锁解除：**
  - 资源剥夺法：将某些死锁进程挂起，抢占它的资源，将这些资源分配给其他的死锁进程，但防止被挂起的进程长时间得不到资源而处于饥饿的状态
  - 撤销进程法：强制撤销部分甚至所有死锁进程并剥夺这些进程的资源，撤销的原则可以按进程优先级和撤销进程代价的高低进行
  - 进程回退法：让一个或多个进程回退到足以回避死锁的地步，进程回退时自愿释放资源而非被剥夺，要求系统保持进程的历史信息，设置还原点

习题

- 死锁检测方法可以获得最大的并发性，即次为死锁避免，死锁预防
- 进程是程序及其数据在计算机上的一次运行活动，是一个动态的概念
- 程序是一组有序的指令集合，是一种静态的概念
- 一个进程可以执行一个或多个程序，一个程序也可构成多个进程
- 进程可创建进程，而程序不能形成新的程序
- 发生“饥饿”的进程的状态可能是就绪态(长期得不到处理机)，也可能是阻塞态(如长期得不到所需的I/O设备)，而发生死锁的进程的状态则必定是阻塞态