操作系统

1. 概述

1.1 目标和使用

操作系统（Operating System，简称OS）是计算机系统中的一种系统软件，它是控制和管理计算机硬件与应用软件之间交互的核心程序。是硬件和软件的中间层。

操作系统能合理组织计算机工作流程，提高计算机工作效率，是管理计算机硬件资源与软件资源的程序集合。

常见操作系统：

- Windows
- Ios：苹果公司的移动操作系统
- Macos：苹果公司的桌面操作系统
- Linux
- Android：谷歌公司开发的手机操作系统，工作范围已经超出了手机的范围，内核是 Linux
- Unix：操作的东西是计算机文化性的东西
- VxWorks：实时嵌入式操作系统，具有速度快、体积小的特点
- FreeBSD：主要运行服务器，具有速度快、稳定度高的特点
- DOS：只能操作一个程序

操作系统的目标：

- 方便性
- 有效性
- 可扩充性
- 开放性

操作系统的作用：

- 用户与计算机硬件系统之间的接口
    - 命令方式（UNIX、DOS命令）
    - 系统调用方式（API）
    - GUI方式（Windows、LINUX）
- 计算机系统资源的管理者
    - 处理机管理、存储器管理、I/O设别管理、文件管理
- 实现对计算机资源的抽象
    - 裸机：无软件的计算机系统
    - 虚拟机：覆盖了软件的机器，向用户提供一个对硬件操作的抽象模型

1.2 操作系统的发展过程

推动OS发展的主要动力：

- 不断提高计算机资源利用率
- 方便用户
- 器件的不断更新换代
- 计算机体系结构的不断发展
- 不断提出的新的应用要求

1. 无操作系统的计算机系统
    - 一次只能执行一个程序
2. 单道批处理系统
3. 多道批处理系统
4. 分时系统
    - 在一台主机上连接了多个嗲有显示器和键盘的终端，同时允许多个用户共享主机中的资源，每个用户都可以通过自己的终端以交互方式使用计算机。
    - 特征：多路性、独立性、及时性、交互性
5. 实时系统
    - 系统能及时响应外部事件的请求，在规定的时间内完成对该事件的处理，并控制所有实时任务协调一致地运行。
6. 微机操作系统
    1. 单用户单任务
    2. 单用户多任务
    3. 多用户多任务
7. 嵌入式操作系统
    - 嵌入式系统：为了完成某个特定功能而设计地系统，或是有附加机制的系统，或是其他部分的计算机硬件与软件的结合体。
    - 嵌入式 OS：用于嵌入式系统的OS
8. 网络操作系统
    - 在计算机网络环境下对网络资源进行管理和控制，实现数据通信及对网络资源的共享，为用户提供与网络资源接口的一组软件和规程的集合。
    - 特征：硬件独立性、接口一致性、资源透明性、系统可靠性、执行并行性
9. 分布式操作系统
    - 分布式系统：基于软件实现的一种多处理机系统，是多个处理及通过通信线路互连而构成的松耦合系统。
    - 分布式 OS：配置在分布式系上的公用 OS

1.3 操作系统的基本特征

- 并发（Concurrence）

  并行性：两个或多个事件在同一时刻发生（食堂最多并行 3 人打餐）

  并发性：两个或多个事件在同一时间间隔内发生（食堂并发用餐人数为 12 人）

  引入进程（任务）：动态、并发
- 共享（Sharing）

  系统中的资源可供内存中多个并发执行的进程共同使用
- 虚拟（Virtual）

  时分复用技术：虚拟处理机、虚拟设备

  空分复用技术：虚拟存储
- 异步（Asynchronism）

  进程的异步性：进程是以人们不可预知的速度向前推进的

1.4 操作系统的硬件运行环境

- 引导程序：位于固件
- 指令：CPU执行
- 事件：硬件中断或软件中断引起
- 执行程序：位于内存
- 程序：位于外存

操作系统内核：

- 常驻内存，通常与硬件紧密相关
- 支撑功能：中断处理、时钟管理、原语操作
- 资源管理功能：进程管理、存储器管理、设备管理

处理机的双重工作模式：

- 内核态（管态、系统态）：执行包括特权指令在内的一切指令
- 用户态（目态）：不能执行特权指令

特权指令：在内核态下运行的指令

- 不仅能访问用户空间，还能访问系统空间

非特权指令：在用户态下运行的指令

- 仅能访问用户空间
- 应用程序所使用的都是非特权指令
- 防止应用成型鼓的运行异常对系统造成破坏

用户态到内核态的切换：

状态位（Mode bit）指示正确的状态：

- 内核态：0

- 用户态：1

当中断或错误出现，硬件切换至内核态。

中断与异常

操作系统是中断驱动的，OS 总在等待某个事件的发生，事件总是由中断或异常引起的。

- 中断（interrupt）：由硬件引起
- 异常/陷阱（trap）：由软件引起

1.5 操作系统的主要功能

处理机管理功能

1. 进程控制

   创建进程、撤销（终止）进程、状态转换
2. 进程同步

   信号量机制
3. 进程通信

   直接通信、间接通信
4. 调度

   作业调度、进程调度

内存管理功能

1. 内存分配和回收
2. 内存保护

   确保每个用户程序仅在自己的内存空间运行

   绝不允许用户程序访问操作系统的程序和数据
3. 地址映射

   逻辑地址转换为物理地址
4. 内存扩充（虚拟存储技术）

   请求调入功能、置换功能

设备管理功能

主要任务：完成 I/O 请求，提高 CPU 和 I/O 设备的利用率。

1. 缓冲管理
2. 设备分配
3. 设备处理

文件管理功能

文件存储空间的管理、目录管理、文件的读/写和保护

操作系统与用户之间的接口

用户接口：

- 联机用户接口：命令行方式，各种系统都有命令行界面的位置，对于小型的嵌入式系统，命令行界面会和内核绑定，通过内核实现

  常用实现命令行的方式：shell

  两个流行的 shell：bash、zsh

  shell 命令：

  - 内置命令（Shell 内置）
  - 外部命令（是个程序）

  在命令行之下，我们可以自己写一个程序，来组合很多命令一起完成任务，这个程序叫做脚本。
- 脱机用户接口
- 图形用户接口：GUI，图形化界面（1995年Windows图形化界面被接受）

程序接口：

- 系统调用：能完成特定功能的子程序

现代 OS 的新功能

系统安全、网络功能和服务、支持多媒体

1.6 操作系统的结构设计

简单结构

也称整体系统结构，是无结构的，是为数众多的一组过程的集合，内部复杂、混乱。

模块化结构

将 OS 按功能划分为若干个模块，并规定号各模块间的接口，称为“模块-接口法”。

优点：提高 OS 设计的正确性、可理解性和易维护性，增强 OS 的可适应性，加速 OS 的开发过程。

大部分现代 OS 采用可加载的内核模块来设计。

分层式结构

操作系统划为若干层，在低层上构建高层。高层仅依赖于紧邻它的低层。

底层（0层）为硬件；最高层（N层）为用户层。

优点：易保证系统的准确性、系统的易维护性和可扩充性。

缺点：效率低

微内核结构

足够小的内核、应用“机制与策略分离”原理、基于客户/服务器模式、采用面向对象技术。

基本功能：进程管理、低级存储器管理、中断和陷入处理

外核结构

内核不提供传统 OS 中的进程、虚拟存储器等抽象，而是专注于物理资源的隔离（保护）与复用。

1.7 系统调用

System Calls（系统调用），又叫 API 接口，计算机与设计人员的接口，一个系统的编程接口。应用程序请求 OS 内核完成某功能时的一种过程调用。

目的：使应用程序可以通过它间接调用 OS 内核中的相关过程，取得相应的服务。

与一般过程调用的区别：

- 运行在不同的系统状态
- 状态的转换
- 返回问题
- 嵌套调用

三个重要的 API

- Win32 API for Windows
- POSIX API for POSIX-based systems
- Java API for the Java virtual machine

系统调用类型：

- 文件操作类
- 进程通信类
- 进程控制类
- 设备管理类
- 信息维护类

进程管理

一、进程描述与进程控制

1.1 前趋图和程序执行

程序顺序执行：一个较大的程序通常都由若干个程序段组成。程序在执行时，必须按照某种先后次序逐个执行。

前趋图：有向无循环图，用于描述进程之间执行的先后顺序。节点表示进程或程序段，有向边表示前趋关系。

多道程序设计：同一时刻内存中存放了多个作业，处理器交替运行不同的作业。提高了系统的效率，尤其是资源利用率。

程序并发执行：采用多道程序技术，将多个程序同时装入内存，使之并发运行。

- 特征：间断性、失去封闭性、不可再现性

结论：程序的并发执行使得程序的执行情况不可预见，其结果不再唯一，成为一个动态的过程。而程序是一个静态的概念，不再能切实反映程序执行的各种特征，（独立性、并发性、动态性）。

1.2 进程的描述

定义：进程是一个具有独立功能的程序，关于某个数据集合的一次运行活动，是系统进行**资源分配**和**调度**的一个独立单位。

要点：

- 进程是程序的一次执行过程
- 进程是一个程序及其数据在处理机上顺序执行时所发生的活动。
- 进程是程序在一个数据集合上运行的一次过程，它是系统进行资源分配和调度的一个独立单位。

进程控制块（Process Control Block，PCB）：专门的数据结构，与进程一一对应。

进程的组成

- PCB（Process Control Block ）：灵魂，进程存在的唯一标志
- *实体*：代码 + 数据
  - 程序：描述了进程要完成的功能，是进程执行时不可修改的部分。
  - 数据：进程执行时用到的数据（用户输入的数据、常量、静态变量）。
- 工作区：参数传递、系统调用时使用的动态区域（堆栈区）。

进程的特征

- 动态性：最基本的特征，生命期
- 并发性：一段时间内同时运行
- 独立性：进程实体是一个能独立运行的基本单位，是系统中独立获得资源和独立调度的基本单位
- 异步性：按各自独立的、不可预知的速度向前推进
- 结构性、制约性、共享性等

进程和程序的区别

- 生命周期：进程是程序的一个实例，是程序的一次执行过程。进程是活动的，程序是静态的，是代码的集合。
- 结构：程序是进程的代码部分。对应关系不同：可1：1，1：N，N：1
- 位置：进程在内存中，程序在外存中。

进程控制块PCB

PCB是进程的一部分，是操作系统为每个进程定义的一个记录型数据结构，是进程存在的唯一标志，常驻内存。

PCB中记录了用于描述进程的当前状态以及OS控制进程运行所需的全部信息。

PCB的作用是使一个在多道程序环境下不能独立运行的程序，成为一个能与其它进程并发执行的进程。

PCB的信息

- 进程标识符
  - 内部标识符PID：一个唯一的正整数标识符
  - 外部标识符UID：它由创建者提供，由用户(进程)在访问该进程时使用。
  - 父进程标识符PPID
- 处理机状态
  - 当处理机被中断时，所有这些信息都必须保存在PCB中，以便在该进程重新执行时，能从断点继续执行。
- 进程调度信息
  - 进程状态：当前是运行、就绪还是阻塞
  - 进程优先级：优先级高的进程先被调度
  - 调度所需的其它信息：比如，已等待CPU的时间
  - 事件：进程等待发生的事件，即阻塞原因
- 进程控制信息
  - 代码和数据的地址：内存或外存地(首)址
  - 进程同步和通信机制：如消息队列指针、信号量等
  - 资源清单：列出了所需资源及已经分配到的资源
  - 链接指针：指向本进程PCB所在队列中的下一个进程的PCB

PCB的组织管理

- 线性方式
- 链接方式
- 索引方式

PCB管理方式

操作系统对PCB的管理：PCB表常驻内存，集中统一管理；PCB中部分常驻内存（proc），部分（user）存于辅存。

1.3 进程的状态与控制

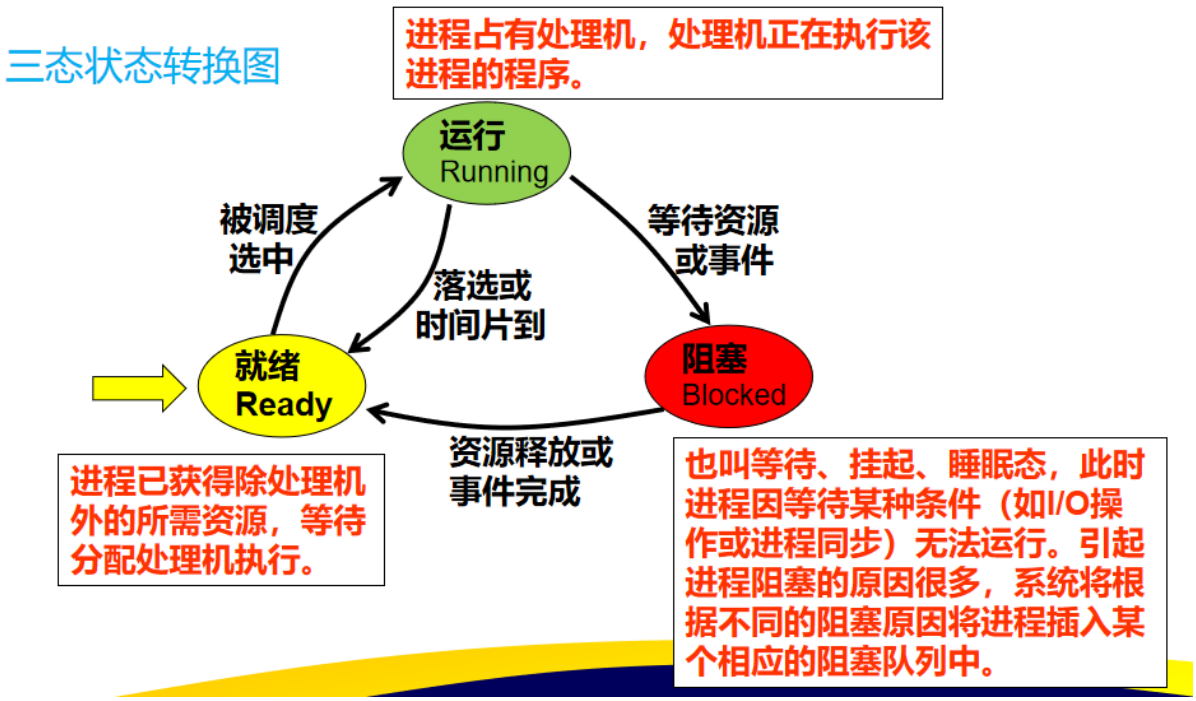
就绪状态：已分配到除CPU以外的所有必要资源，只缺CPU；处于就绪状态的进程可能有多个，在就绪队列中排队

执行状态：已获得CPU，正在执行的状态

- 单处理机：一个进程处于执行状态
- 多处理机：多个进程处于执行状态

阻塞状态：正在执行的进程由于发生某事件而暂时无法继续执行的状态

三种状态的转换：

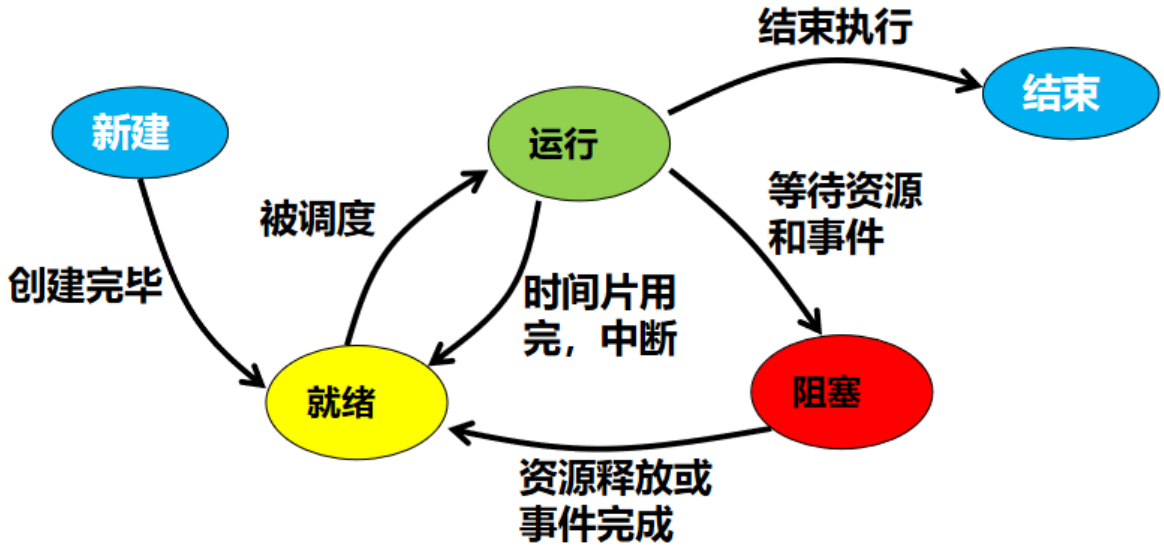


创建状态和终止状态

创建状态：申请一个空白PCB；填写PCB；分配资源；设置就绪状态插入就绪队列

终止状态：等待OS善后；收回PCB。

五种状态的相互转换：



五态进程状态转换

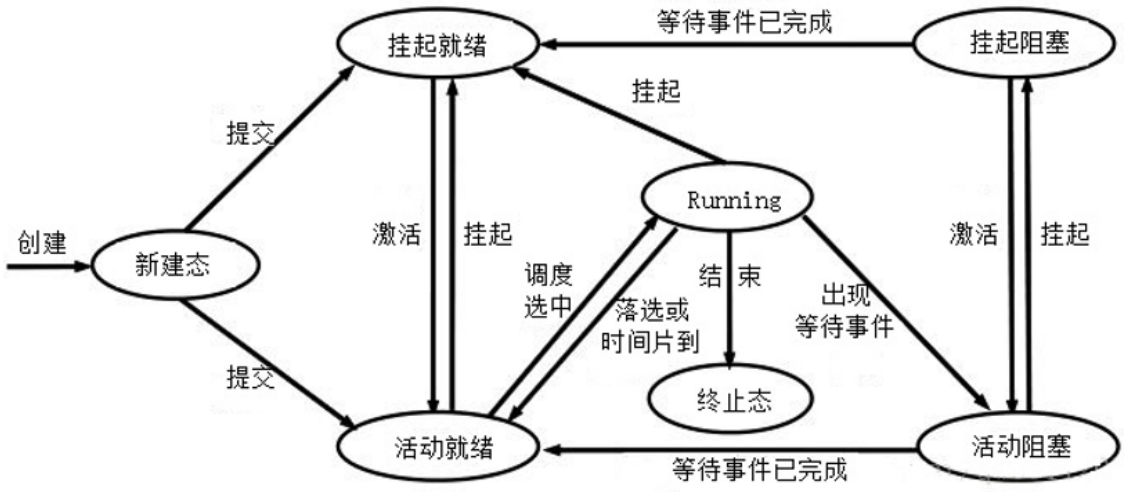
挂起状态

暂停正在执行的进程，或暂不调度正处于就绪状态的进程，使其处于静止的一种状态。（从内存调至交换区）

导致挂起状态的原因：

- 终端用户的请求
- 父进程请求
- 负荷调节的需要
- 操作系统的需要

七种状态的相互转换：



七态进程状态转换

1.4 进程控制

进程控制的任务：进程控制是对系统中所有进程从产生、存在到消亡的全过程实行有效的管理和控制。进程控制一般是由操作系统的内核来实现，内核在执行操作时，往往是通过执行各种原语操作来实现的。

进程控制的机构：OS的内核中的原语(Primitive) 。

内核、原语、进程图

内核：加在硬件上的第一层软件，通过执行各种原语操作来实现各种控制和管理功能，具有创建、撤消、进程通信、资源管理的功能。基本功能：

- 支撑功能：中断处理、时钟管理、原语操作
- 资源管理功能：进程管理、进程通信、存贮管理、设备管理

原语：由若干条机器指令构成的可完成特定功能的程序段，它是一个 “原子操作(atomic operation)” 过程，作为一个整体而不可分割 - - 要么全都完成，要么全都不做（类似数据库中的“事务”）。原语主要是通过屏蔽各种中断和固化技术保证其原子性的。

原语分类

- 进程控制原语
- 进程通信原语
- 进程管理原语
- 其他方面的原语

进程图：用于描述一个进程的家族关系的有向树。结点代表进程。一棵树表示一个家族，根结点为该家族的祖先(Ancestor)。

OS加载到内存后首先创建一个根进程，再由这个根进程通过系统调用创建一系列子进程和子孙进程，从而形成一颗进程树。子进程能继承父进程得到的系统资源。创建子进程时，有两种执行可能：

- 父子并发执行
- 父进程等待，直到某个或全部子进程运行结束

进程管理最基本的功能，一般由OS内核中的原语实现，包括：

- 进程创建
  - 通过执行fork()原语创建进程，执行create( )原语创建线程
  - 过程：申请空白PCB；分配所需资源；初始化PCB；插入就绪队列。
- 进程终止

  引起进程终止的事件
  - 正常结束（主动）
  - 异常结束（主动）
  - 异常结束（被动）
  - 外界干预
- 进程阻塞与唤醒

  引起进程阻塞的事件
  - 请求系统服务、启动某种操作、新数据尚未到达、无新工作可做

  阻塞可认为是进程自身的一种主动行为：通过调用阻塞原语block，将自己阻塞，放弃CPU的使用。

进程唤醒是一种被动行为——由别人唤醒：其他进程调用唤醒原语wakeup，将等待该事件的进程唤醒。

- 进程挂起与激活

  挂起功能（suspend）：自身挂起、挂起具有指定标识符的进程、将其进程及其全部或部分“子孙”挂起。

  激活功能（activate）：使处于静止状态的进程变为活动。

  挂起过程：检查要被挂起进程的状态，若处于活动就绪态就修改为挂起就绪态，若处于活动阻塞态，则修改为挂起阻塞态。被挂起的进程要交换到磁盘交换区。

  当系统资源尤其是内存资源充裕或进程请求激活指定进程时，系统或有关进程会调用激活（解挂）原语把指定进程激活。

  激活过程：把进程调入内存，然后修改它的状态，挂起阻塞态改为阻塞态，挂起就绪态改为就绪态，并分别插入相应队列中。

1.5 进程通信

进程通信是指进程之间的信息交换。

- 低级进程通信：进程的同步和互斥
  - 效率低，通信对用户不透明
- 高级进程通信：
  - 使用方便，高效地传送大量数据

1.5.1 进程通信的类型

共享存储器系统

- 基于共享数据结构的通信方式（效率低）
- 基于共享存储区的通信方式（高级）

管道通信

- 管道：用于连接一个读进程和一个写进程以实现它们之间通信的一个共享文件，又名pipe文件。
- 管道机制的协调能力：互斥、同步、对方是否存在

消息传递系统

- 直接通信方式
- 间接通信方式（通过邮箱）

客户机-服务器系统

- 套接字（Socket）
- 远程过程调用（RPC）和远程方法调用（RMI，Java）

1.5.2 消息传递通信的实现方式

直接通信方式

- 发送原语：send(receiver, message)
- 接收原语：receive(sender, message)

间接通信方式：通过信箱来完成

- 发送原语：send(mailbox, message)
- 接收原语：receive(mailbox, message)
- 信箱类型：私用、公共、共享邮箱

1.5.3 Linux进程通信方式

管道、信号、消息队列、共享内存、信号量、套接字。

1.6 线程的基本概念

60年代中期：提出进程概念

80年代中期：提出线程概念

提出线程的目的：

- 减少程序在并发执行时所付出的时空开销
- 使OS具有更好的并发性
- 适用于SMP结构的计算机系统

进程是拥有资源的基本单位（传统进程称为重型进程）；线程作为调度和分派的基本单位（又称为轻型进程）。线程依附于进程存在。

1.6.1 进程和线程的比较

调度的基本单位：

- 在传统的OS中，拥有资源、独立调度和分派的基本单位都是进程。
- 在引入线程的OS中，线程作为调度和分派的基本单位，进程作为资源拥有的基本单位。
- 在同一进程中，线程的切换不会引起进程切换，在由一个进程中的线程切换到另一个进程中的线程时，将会引起进程切换。

并发性：

- 在引入线程的OS中，不仅进程之间可以并发执行，而且在一个进程中的多个线程之间，也可以并发执行。

拥有资源：

- 进程是系统中拥有资源的一个基本单位，它可以拥有资源
- 线程本身不拥有系统资源，仅有一点保证独立运行的资源
- 允许多个线程共享其隶属进程所拥有的资源

独立性

- 同一进程中的不同线程之间的独立性要比不同进程之间的独立性低得多。

系统开销：

- 在创建或撤销进程时，OS所付出的开销将显著大于创建或撤销线程时的开销
- 线程切换的代价远低于进程切换的代价
- 同一进程中的多个线程之间的同步和通信也比进程的简单

支持多处理系统

1.6.2 线程的状态和线程控制块

线程状态

- 执行态、就绪态、阻塞态
- 线程状态转换与进程状态转换一样

线程控制块（thread control block，TCB）

- 包含了：线程标识符、一组寄存器、线程运行状态、优先级、线程专有存储区、信号屏蔽、堆栈指针

1.7 线程的实现

实现方式：

- 内核支持线程KST：利用系统调用
- 用户级线程ULT：借助中间系统
- 组合方式

1.7.1 内核支持线程KST

在内核空间实现。

优点：

- 在多处理机系统中，内核可同时调度同一进程的多个线程
- 线程的切换比较快，开销小
- 内核本身可采用多线程技术，提高执行速度和效率

缺点：

- 对用户线程切换，开销较大

1.7.2 用户级线程ULT

在用户空间实现。

优点：

- 线程切换不需要转换到内核空间
- 调度算法可以是进程专用
- 线程的实现与OS平台无关

缺点：

- 系统调用的阻塞问题
- 多线程应用不能利用多处理机进行多重处理的优点

1.7.3 ULT和KST组合方式

多对一模型、一对一模型、多对多模型

多对一模型

多个用户级线程映射到一个内核线程。

多个线程不能并行运行在多个处理器上。

线程管理在用户态执行，因此是高效的，但一个线程的阻塞系统调用会导致整个进程的阻塞。

用于不支持内核线程的系统中。

一对一模型

每个用户级线程映射到一个内核线程。

比多对一模型有更好的并发性。

允许多个线程并行运行在多个处理器上。

创建一个ULT需要创建一个KST，效率较差。

多对多模型

多个用户级线程映射为相等或小于数目的内核线程

允许操作系统创建足够多的KST。

3. 处理机调度与死锁

3.1 处理机调度概述

3.1.1 处理机调度层次

- 高级调度：长程调度/作业调度

  把后备作业调入内存，只调入一次，调出一次
- 中级调度：中程调度/内存调度

  将进程调至外存，条件适合再调入内存，在内、外存对换区进行进程对换
- 低级调度：短程调度/进程调度/处理机调度

  从就绪队列选取进程分配给处理机

  最基本的调度，频率非常高（相当于一个时间片完成）

高级调度

调度对象：作业

根据某种算法，选中在外存后备队列中的作业，调入内存，并为其创建进程和分配必要的资源，再将新创建的进程排在就绪队列上等待调度。

主要用于多道批处理系统中。

中级调度

即对换功能：

将暂不运行的进程，调至外存等待；将处于外存上的急需运行的进程，调入内存运行。

低级调度

调度对象：进程

根据某种调度算法，决定就绪队列中的哪个进程应获得处理机。应用在于多道批处理、分时和实时OS。

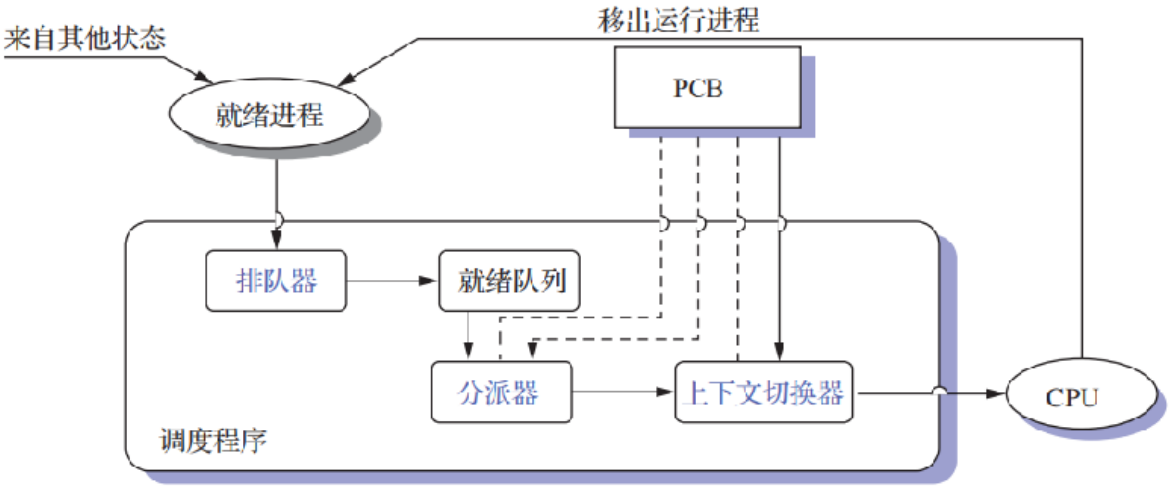
3.1.2 进程调度的任务和方式

进程调度的任务：

- 保存处理机的现场信息
- 按某种算法选取进程
- 把处理器分配给进程

进程调度机制（调度程序分为三部分）

- 排队器：用于将就绪进程插入相应的就绪队列
- 分派器：用于将选定的进程移出就绪队列
- 上下文切换器：进行新旧进程之间的上下文切换



进程调度的方式：非抢占方式、抢占方式。

非抢占方式：一旦把处理机分配给某进程后，便让该进程一直执行，直至该进程完成或发生某事件而被阻塞时，才再把处理机分配给其他进程，决不允许某进程抢占已经分配出去的处理机。

抢占方式：允许调度程序根据某种原则，去暂停某个正在执行的进程，将已分配给该进程的处理机重新分配给另一进程。(现代OS广泛采用)

- 优先权原则：允许优先权高的新到进程抢占当前进程的处理机
- 短作业优先原则：短作业可以抢占当前较长作业的处理机
- 时间片原则：各进程按时间片运行，当一个时间片用完后，便停止该进程的执行而重新进行调度

3.1.3 处理机调度算法的目标

共同目标：资源利用率、系统效率

批处理系统的目标：平均周转时间(Turnaround Time)短、系统吞吐量(Throughout)高、处理机利用率(CPU Utilization)高

分时系统的目标：响应时间快、均衡性

实时系统的目标：截止时间的保证、可预测性

$$CPU利用率 = \frac{CPU有效工作时间}{CPU有效工作时间+CPU空闲等待时间}$$

3.1.4 评价指标

周转时间：从作业提交给系统开始，到作业完成为止的这段时间间隔。

- 平均周转时间：$T = \frac{1}{n}(\sum_{i=1}^{n} T_i)$
- 带权周转时间：权值为作业周转时间T与系统为之服务时间TS之比。$W = \frac{1}{n}(\sum_{i=1}^{n} \frac{T_i}{T_{S_i}})$
- 平均带权周转时间

响应时间：从用户提交请求开始，直到系统首次显示出处理结果为止的一段时间。

等待时间（进程调度）：进程在就绪队列中等待调度的所有时间之和。

3.2 调度算法

3.2.1 作业调度算法

先来先服务调度算法（FCFS）、短作业优先调度算法（SJF）、优先级调度算法（PSA）、高响应比优先级调度算法（HRRN）。

3.2.2 进程调度算法

先来先服务调度算法（FCFS）、短作业优先调度算法（SJF）、优先级调度算法（PSA）、高响应比优先级调度算法（HRRN）、多级队列调度算法(MQ)、多级反馈队列调度算法(MFQ)、基于公平原则的调度算法(FS)、时间片轮转调度算法(RR)。

3.2.3 FCFS

先来先服务，First Come First Served，FCFS。

算法内容：调度作业/就绪队列中最先入队者，等待操作完成或阻塞。

算法原则：按作业/进程到达顺序服务（执行）。

调度方式：非抢占式调度

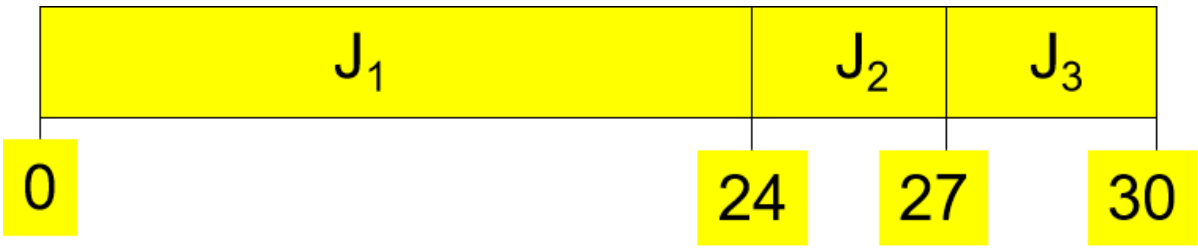
适用场景：作业/进程调度

优点：有利于CPU繁忙型作业，充分利用CPU资源

缺点：不利于I/O繁忙型作业，操作耗时

【FCFS举例】

情况1：假设作业到达顺序如下：J1、J2、J3，分别的运行时间为：24、3、3，则该调度的甘特图（Gantt）为：



那么

- 平均等待时间 = (0 + 24 + 27)/3 = 17
- 平均周转时间 = (24 + 27 + 30)/3 = 27

- 平均带权周转时间 = (24/24 + 27/3 + 30/3)/3 = 6.7

情况2：假设作业到达顺序如下：J2、J3、J1，则该调度的甘特图（Gantt）为：

那么

- 平均等待时间 = (6 + 0 + 3)/3 = 3
- 平均周转时间 = (30+ 3 + 6)/3 = 13
- 平均带权周转时间 = (30/24 + 3/3 + 6/3)/3 = 1.4

此结果产生是由于短进程先于长进程到达。

3.2.4 SJF

短作业优先，Shortest Job First，SJF。

算法内容：所需服务时间最短的作业/进程优先服务（执行）

算法原则：追求最少的平均（带权）周转时间

调度方式：非抢占式

适用场景：作业/进程调度

优点：平均等待/周转时间最少

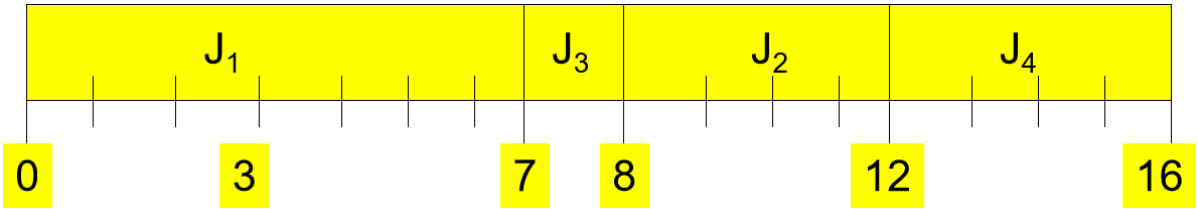
缺点：长作业周转时间会增加、估计时间不准确，不能保证紧迫任务及时处理。

对于进程调度，SJF 有两种模式：

- 非抢占式
- 抢占式——抢占发生在有比当前进程剩余时间片更短的进程到达时，也称为最短剩余时间优先调度(SRTN)

【举例】非抢占式 SJF

进程	到达时间	运行时间
J_1	0.0	7
J_2	2.0	4
J_3	4.0	1
J_4	5.0	4

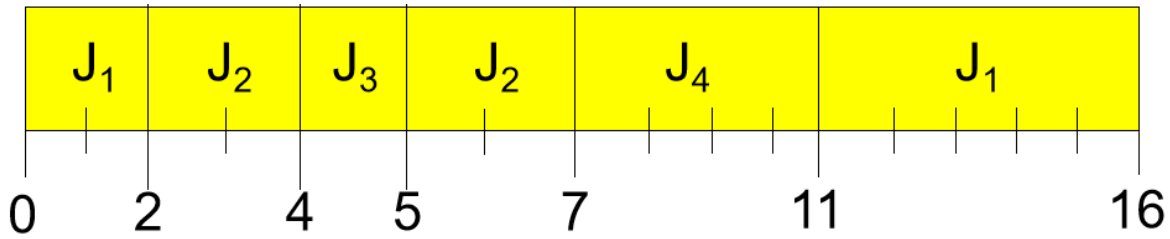


那么

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

【举例】抢占式 SJF

进程	到达时间	运行时间
J_1	0.0	7
J_2	2.0	4
J_3	4.0	1
J_4	5.0	4



那么

- 平均等待时间 = (9 + 1 + 0 +2)/4 = 3
- 平均周转时间 = (16+ 5 +1+ 6)/4 = 7

3.2.5 PSA/PR

优先级调度，Priority-Scheduling Algorithm，PSA。

算法内容：又叫优先权调度，按作业/进程的优先级（紧迫程度）进行调度

算法原则：优先级最高（最紧迫）的作业/进程先调度

调度方式：抢占/非抢占式（并不能获得及时执行）

适用场景：作业/进程调度

优点：实现简单，灵活

缺点：饥饿——低优先级的进程可能永远得不到运行

解决办法：老化——视进程等待时间的延长提高其优先数

优先级设置原则：

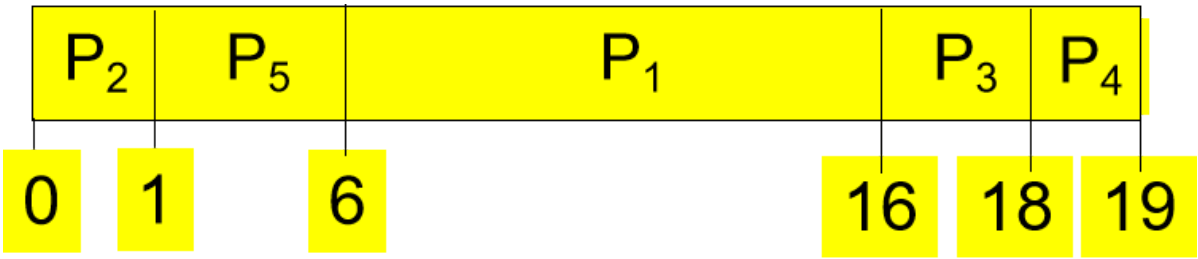
- 静态/动态优先级
  - 静态：创建进程时确定优先数(整数)，在进程的整个运行期间保持不变
  - 动态：创建进程时先赋予其一个优先级，然后其值随进程的推进或等待时间的增加而改变
- 系统 > 用户；交互型 > 非交互型；I/O型 > 计算型

- 低优先级进程可能会产生“饥饿”

高响应比优先调度算法是一种优先级调度算法，用于作业调度。

【举例】非抢占式

进程	优先级	运行时间
P1	3	10
P2	1	1
P3	3	2
P4	4	1
P5	2	5



那么

- 平均等待时间 = (6 + 0 + 16+18+1)/5 =8.2
- 平均周转时间 = (16 + 1 + 18+19+6)/5 = 12

HRRN

高响应比优先调度算法，Highest Response Ratio Next，HRRN，是PSA的特例。

算法内容：结合FCFS和SJF，综合考虑等待时间和服务时间计算响应比，高的优先调度

算法原则：综合考虑作业/进程的等待时间和服务时间

调度方式：非抢占式

适用场景：作业/进程调度

响应比计算：

- 响应比 = （等待时间 + 服务时间）/ 服务时间
- 只有当前进程放弃执行权（完成/阻塞）时，重新计算所有进程响应比
- 长作业等待越久响应比越高，更容易获得处理机

HRRN 既考虑作业的等待时间，又考虑作业的运行时间

- 如等待时间相同，运行时间越短，类似于SJF
- 如运行时间相同，取决于等待时间，类似于FCFS

- 长作业可随其等待时间的增加而提高响应比，从而得到调度

优先级：优先级 $=\frac{\text{等待时间}+\text{要求服务时间}}{\text{要求服务时间}}$

响应比：$R_p=\frac{\text{等待时间}+\text{要求服务时间}}{\text{要求服务时间}}=\frac{\text{响应时间}}{\text{要求服务时间}}$

3.2.6 RR

时间片轮转，Round-Robin，RR。

算法内容：按进程到达就绪队列的顺序，轮流分配一个时间片去执行，时间用完则剥夺

算法原则：公平、轮流为每个进程服务，进程在一定时间内能得到响应

调度方式：抢占式，由时钟中断确定时间到

适用场景：进程调度

优点：公平、响应快，适用于分时系统

缺点：时间片太大，相当于FCFS；太小，处理机切换频繁，开销增大

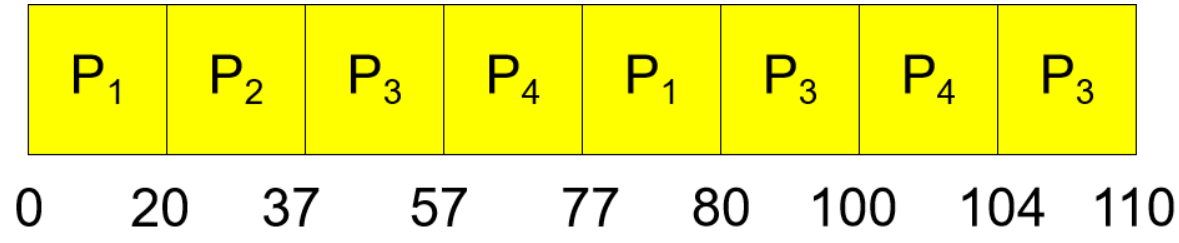
- 时间片：小单位的CPU时间，通常为10~100毫秒
- 时间片决定因素：系统响应时间、就绪队列进程数量、系统处理能力

执行原理：为每个进程分配不超过一个时间片的CPU。时间片用完后，该进程将被抢占并插入就绪队列末尾，循环执行。

假定就绪队列中有n个进程、时间片为q, 则每个进程每次得到1/n的、不超过q单位的成块CPU时间，没有任何一个进程的等待时间会超过(n-1) q单位。

【举例】时间片为20

进程	运行时间
P1	23
P2	17
P3	46
P4	24



那么

- 平均等待时间：(57+20+64+80)/4 = 55.25
- 平均响应时间: (0+20+37+57)/4= 28.5
- 通常，RR的平均周转时间比SJF长，但响应时间要优于SJF.

时间片大小的确定

一般准则：时间片 / 10 > 进程上下文切换的时间。

3.2.7 MQ

多级队列调度，Multileveled Queue，MQ。

就绪队列从一个分为多个，每个队列有自己的调度算法，调度须在队列间进行。

- 固定优先级调度，即前台运行完后再运行后台，有可能产生饥饿。
- 给定时间片调度，即每个队列得到一定的CPU时间，进程在给定时间内执行；如80%的时间执行前台的RR调度，20%的时间执行后台的FCFS调度。

3.2.8 MFQ

多级反馈队列调度，Multileveled Feedback Queue，MFQ。

算法内容：设置多个优先级排序的就绪队列，优先级从高到低，时间片从小到大，前面队列不为空，不执行后续队列进程。新进程采用队列降级法

- 进入第一级队列，按FCFS分时间片，没有执行完，移到第二级，第三级.....

算法原则：集前几种算法优点

调度方式：抢占式

适用场景：进程调度

3.2.9 基于公平原则的调度算法

主要考虑调度的公平性。

公平分享调度算法，Fare-Share，FS。

- 调度的公平性主要针对用户而言
- 所有用户能获得相同的处理机时间或时间比例

保证调度算法（Guarantee）

- 性能保证，而非优先运行
- 如保证处理机分配的公平性（处理机时间为1/n）

3.3 实时调度

实时调度是针对实时任务的调度。实时任务，都联系着一个截止时间。

根据实时任务性质分为

- 硬实时HRT任务
- 软实时SRT任务

根据调度方式分为

- 非抢占式调度算法：响应时间为数秒至数十秒，可用于要求不太严格的实时控制系统
- 抢占式调度算法：响应时间为数秒至数百毫秒，可用于有一定要求的实时控制系统
    - 基于时钟中断的抢占式优先级调度
    - 立即抢占的优先级调度

3.3.1 实时调度的基本条件

1、提供必要的信息

就绪时间、开始截止时间和完成截止时间、处理时间、资源要求、优先级。

2、系统处理能力强

单处理机系统、多处理机系统都满足。

3、采用抢占式调度机制

4、 采用快速切换机制

对中断具有快速响应能力，快速的任务分派能力。

3.3.2 EDF

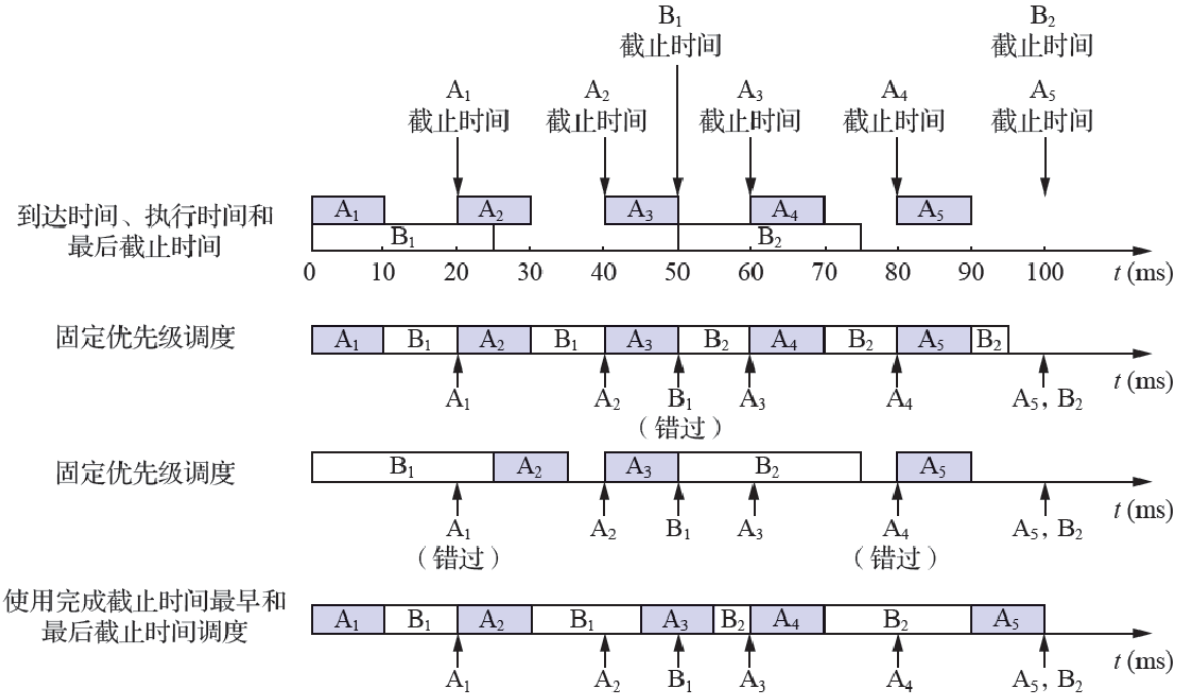
最早截止时间优先调度，Earliest Deadline First，EDF。

EDF根据任务的截止时间确定优先级，截止时间越早，优先级越高。

既可用于抢占式调度，也可用于非抢占式调度

- 非抢占式调度用于非周期实时任务
- 抢占式调度用于周期实时任务

【举例】两个周期性实时任务A和B，任务A要求每20 ms执行一次，执行时间为10 ms，任务B要求每50 ms执行一次，执行时间为25 ms。



3.3.3 LLF

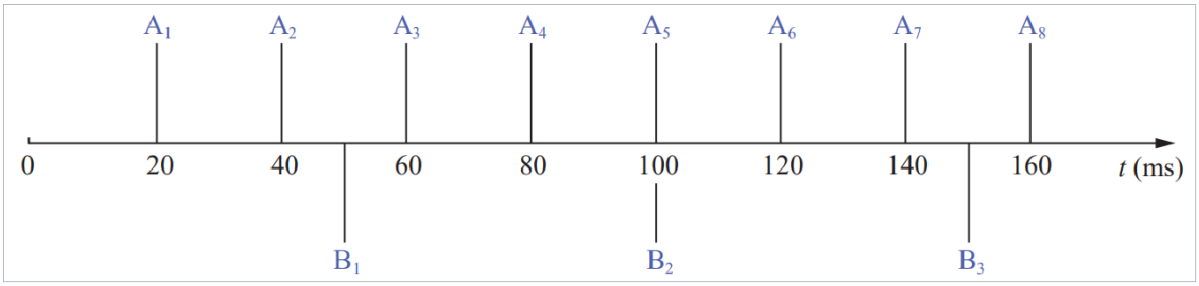
最低松弛度优先调度，LLF。

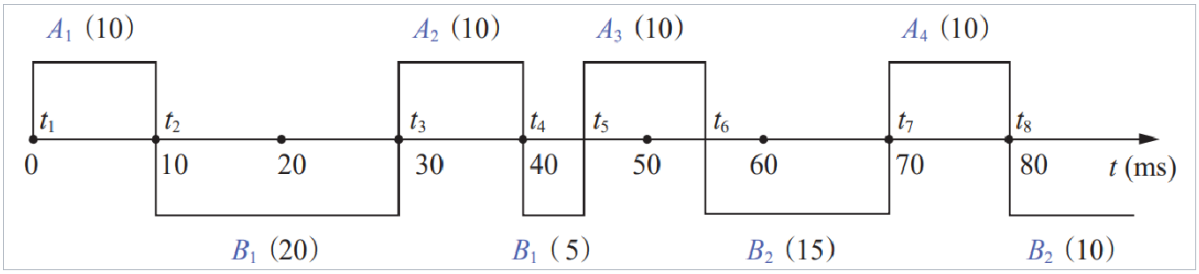
根据任务的紧急程度（松弛度）确定任务优先级

- 紧急程度越高（松弛度越低），优先级越高
- 松弛度 = 必须完成时间 −其本身的运行时间 −当前时间

主要用在抢占式调度方式中。

【举例】两个周期性实时任务A和B，任务A要求每20 ms执行一次，执行时间为10 ms，任务B要求每50 ms执行一次，执行时间为25 ms。





3.3.4 优先级倒置现象

采用优先级调度和抢占方式，可能产生优先级倒置。

现象：高优先级进程被低优先级进程延迟或阻塞。

解决办法：

- 制定一些规定，如规定低优先级进程执行后，其所占用的处理机不允许被抢占；
- 建立动态优先级继承。

3.4 Linux进程调度

默认调度算法：完全公平调度CFS算法。

基于调度器类：允许不同的可动态添加的调度算法并存，每个类都有一个特定的优先级。

总调度器：根据调度器类的优先顺序，依次对调度器类中的进程进行调度。

调度器类：使用所选的调度器类算法（调度策略）进行内部的调度。

调度器类的默认优先级顺序为：Stop_Task > Real_Time > Fair > Idle_Task

普通进程调度：

- 采用SCHED_NORMAL调度策略。
- 分配优先级、挑选进程并允许、计算使其运行多久
- CPU运行时间与友好值（-20~+19）有关，数值越低优先级越高。

实时进程调度：

- 实时调度的进程比普通进程具有更高的优先级。
- SCHED_FIFO：进程若处于可执行的状态，就会一直执行，直到它自己被阻塞或者主动放弃CPU。
- SCHED_RR：与SCHED_FIFO大致相同，只是进程在耗尽其时间片后，不能再执行，而是需要接受CPU的调度。

3.5 死锁概述

3.5.1 死锁概念

死锁（Deadlock）：指多个进程在运行过程中因争夺资源而造成的一种僵局，当进程处于这种僵持状态时，若无外力作用，这些进程都将永远不能再向前推进。

3.5.2 资源问题

可重用性资源和可消耗性资源

- 可重用性资源：一次只能分配给一个进程，不允许多个进程共享，遵循：请求资源、使用资源、释放资源（大部分资源）
- 可消耗性资源：由进程动态创建和消耗（进程间通信的信息）

可抢占性和不可抢占性资源

- 可抢占性资源：某进程在获得这类资源后，该资源可以再被其他进程或系统抢占，**CPU和主存区**
- 不可抢占资源：当系统把这类资源分配个某进程后，再不能强行收回，只能在进程用完后自行释放，**打印机、磁带机**

3.5.3 死锁原因

竞争不可抢占性资源：系统中的不可抢占性资源，由于它们的数量不能满足诸进程运行的需要，会使进程在运行过程中，因争夺这些资源而陷入僵局。

竞争可消耗性资源

进程推进顺序不当

3.5.4 产生死锁的必要条件

互斥：一段时间内某资源只能被一个进程占用。

请求和保持：一个至少持有一个资源的进程等待获得额外的由其他进程所持有的资源。

不可抢占：一个资源只有当持有它的进程完成任务后，自由的释放。

循环等待：等待资源的进程之间存在环

- P0等待P1的资源，P1等待P2的资源，……，Pn-1等待Pn的资源，Pn等待P0的资源

3.5.5 处理死锁的方法

确保系统永远不会进入死锁状态：

- 死锁预防：破坏死锁的四个必要条件中一个或几个
- 死锁避免：在资源动态分配时，防止系统进入不安全状态

允许系统进入死锁状态，然后恢复系统

- 死锁检测：事先不采取任何措施，允许死锁发生，但及时检测死锁发生
- 死锁恢复：检测到死锁发生时，采取相应措施，将进程从死锁状态总解脱出来

忽略这个问题，假装系统中从未出现过死锁。这个方法被大部分操作系统采用，包括UNIX。

3.6 预防死锁

破坏死锁的四个必要条件中一个或几个

- 互斥：共享资源必须的，不仅不能改变，还应加以保证
- 请求和保持：必须保证进程申请资源的时候没有占有其他资源
- 非抢占：如果一个进程的申请没有事先，它要释放所有占有的资源
- 循环等待：对所有的资源类型排序进行线性排序，并赋予不同的序号，要求进程按照递增顺序申请资源

3.7 避免死锁

死锁避免算法动态检查资源分配状态以确保不会出现循环等待的情况。

资源分配状态定义为可用的与已分配的资源数，和进程所需的最大资源量。

3.7.1 安全状态

当进程申请一个有效的资源的时候，系统必须确定分配后是安全的。

如果存在一个安全序列，则系统处于安全状态

进程序列 <P1, P2, ..., Pn> 是安全的，如果每一个进程 Pi 所申请的可以被满足的资源数加上其他进程所持有的该资源数小于系统总数

- 如果 Pi 需要的资源不能马上获得，那么 Pi 等待直到所有的 Pi-1 进程结束
- 当 Pi-1 结束后，Pi 获得所需资源，执行、返回资源、结束
- 当 Pi 结束后，Pi+1 获得资源，执行.....

基本事实

- 如果一个系统在安全状态，就没有死锁；否则，可能死锁。

死锁避免：确保系统永远不会进入不安全状态。

3.7.2 资源分配图

系统模型

资源类型 R1、R2、.....、Rm：CPU周期、内存空间、I/O设备

每一种资源Ri有Wi种实例

每一个进程通过如下方法来使用资源：申请、使用、释放

组成

一个顶点的集合V和边的集合E

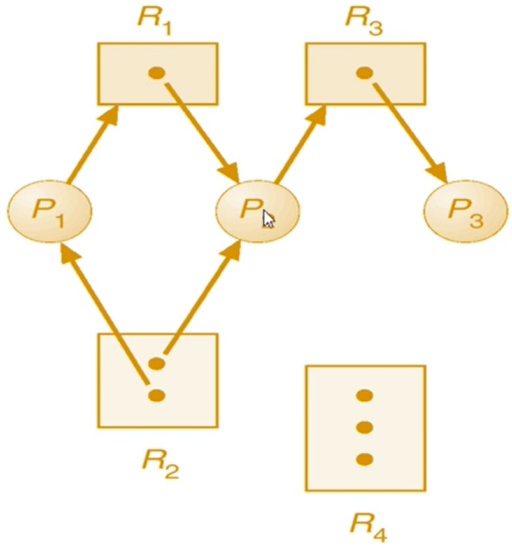
V被分为两个部分

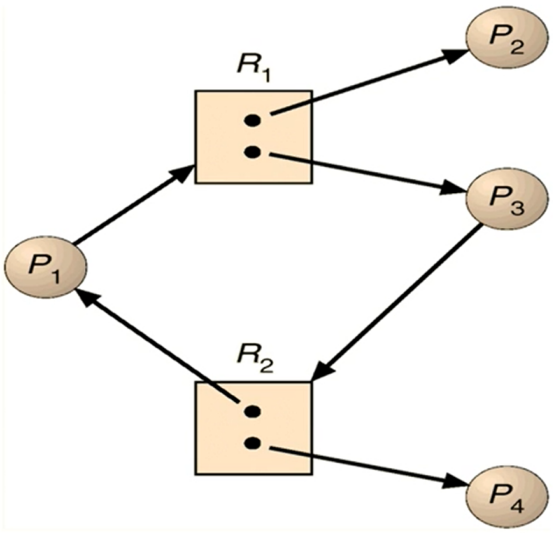
- P={P1，P2，……，Pn}，含有系统中全部的进程
- R={R1，R2，……，Rn}，含有系统中全部的资源

请求边：有向边Pi --> Rj

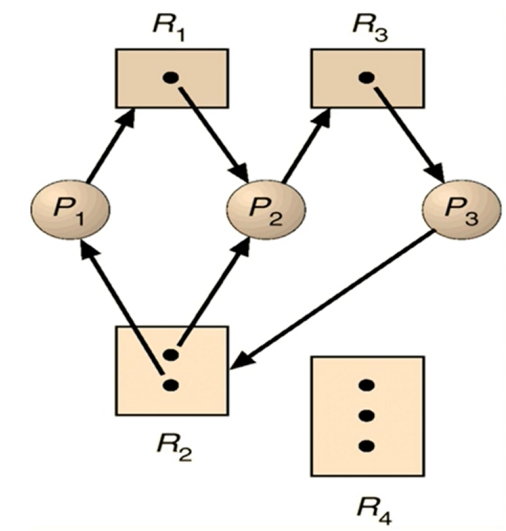
分配边：有向边Rj --> Pi

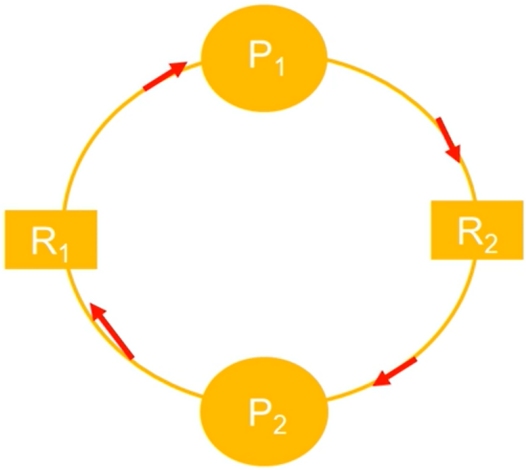
无死锁的资源分配图：





有死锁的资源分配图：





基本事实

- 如果图没有环，那么不会有死锁
- 如果图有环，那么
  - 如果每一种资源类型只有一个实例，那么死锁发生
  - 如果一种资源类型有多个实例，那么可能死锁

3.8 死锁的检测与解除

当系统为进程分配资源时，若未采取任何限制性措施，则系统必须提供检测和解除死锁的手段。为此，系统必须：

- 保存有关资源的请求和分配信息
- 提供一种算法，以利用这些信息来检测系统是否已进入死锁状态

3.8.1 资源分配图的简化

在资源分配图中，找出一个既不阻塞又非独立的进程节点Pi。在顺利的情况下，Pi可获得所需资源而继续运行，直至运行完毕，再释放其所占有的全部资源，这相当于消去Pi所有的请求边和分配边，使之成为孤立的节点

P1释放资源后，便可使P2获得资源而继续运行，直到P2完成后又释放出它所占有的全部资源；

在进行一系列的简化后，若能消去图中所有的边，使所有进程都成为孤立节点，则称该图是可完全简化的；若不能通过任何进程使该图完全简化，则称该图不可完全简化。

3.8.2 死锁定理

所有的简化顺序，都将得到相同的不可简化图。

S为死锁状态的充分条件是：当且仅当S状态的资源分配图是不可完全简化的。该充分条件称为死锁定理。

3.8.3 死锁的解除

常用解除死锁的两种方法

- 抢占资源：从一个或多个进程中抢占足够数量的资源给死锁进程
- 终止或撤销进程：终止系统中一个或多个死锁进程，直到打破循环环路，使死锁状态消除为止
    - 终止所有死锁进程（最简单方法）
    - 逐个终止进程（稍温和方法）

4. 进程同步

4.1 概念

主要任务：使并发执行的诸进程之间能有效地共享资源和相互合作，从而使程序的执行具有可再现性。

进程间的制约关系：

- 互斥关系：间接相互制约（进程互斥适用临界资源）
- 同步关系：直接相互制约（进程间相互合作）

临界资源：系统中某些资源一次只允许一个进程使用，称这样的资源为临界资源或互斥资源或共享变量。诸进程间应采取互斥方式，实现对这种资源的共享。

4.1.1 进程同步（协作关系）

也称同步关系。某些进程为完成同一任务需要分工协作，即一个进程的执行依赖于另一个进程的消息，当没有消息时要等待，直到消息到达被唤醒。由于合作的每一个进程都是独立地以不可预知的速度推进，这就需要相互协作的进程在某些协调点上协调各自的工作。当合作进程中的一个到达协调点后，在尚未得到其伙伴进程发来的消息或信号之前应阻塞自己，直到其他合作进程发来协调信号或消息后才被唤醒并继续执行。这种协作进程之间相互等待对方消息或信号的协调关系称为**进程同步**。

进程的同步（Synchronization）是解决进程间协作关系(直接制约关系)的手段。

4.1.2 进程互斥（竞争关系）

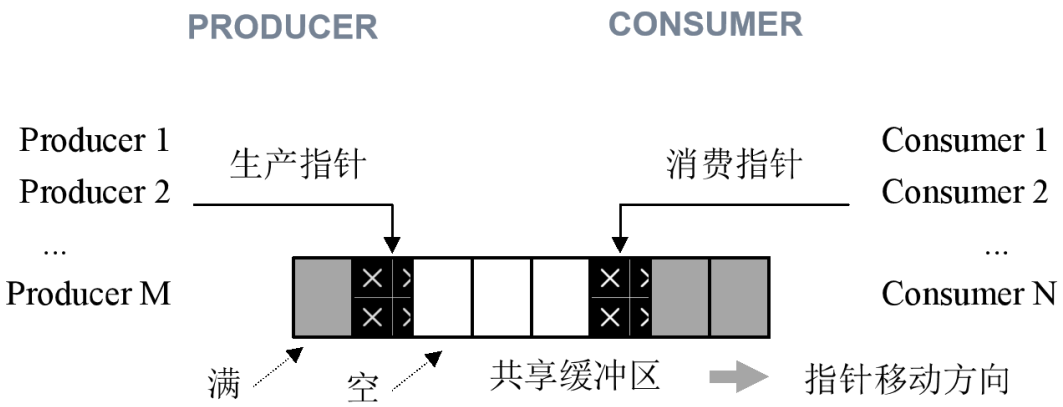
一组并发进程中的一个或多个程序段，因共享某一公有资源而导致它们必须以不允许交叉执行的单位执行。即不允许两个以上的共享该资源的并发进程同时进入临界区称为**互斥**。

进程的互斥（Mutual Exclusion）是解决进程间竞争关系(间接制约关系)的手段。进程互斥指若干个进程要使用同一共享资源时，任何时刻最多允许一个进程去使用，其他要使用该资源的进程必须等待，直到占有资源的进程释放该资源。

4.1.3 同步与互斥

- 进程的互斥是进程同步的特例，可视为由临界资源的特性而形成的进程之间的共享等待同步。
- 两者都涉及到并发进程访问共享资源的问题。
- 进程的互斥是指只要无进程在使用共享资源时，就允许任意一个参与竞争的进程去使用该资源。
- 进程同步不同：当涉及到共享资源的进程必须同步时，即使无进程在使用共享资源，那么尚未得到同步消息的进程仍不能去使用这个资源。

4.1.4 生产者-消费者问题



共享变量：

- 缓冲池 buffer：用数组来表示具有 n 个缓冲区的缓冲池。
- 输入指针 in：指示下一个可投放产品的缓冲区，每当生产者进程生产并投放一个产品后，输入指针加1，初值为0。
- 输出指针 out：指示下一个可获取产品的缓冲区，每当消费者进程取走一个产品后，输出指针加1，初值为0。
- 整型变量 count：初值为0，表示缓冲区中的产品个数。

生产者进程

```
void producer() {
    while(1){
        produce an item in nextp;
        …
        while (count == n)
            ; // do nothingx
        // add an item to the buffer
        buffer[in] = nextp;
        in = (in + 1) % n;
        count++;
    }
}
```

消费者进程

```
void consumer() {
    while(1){
        while(count == 0)
            ;  // do nothing
        // remove an item from the buffer
        nextc = buffer[out];
        out = (out + 1) % n;
        count--;
        consume the item in nextc;
        ...
    }
}
```

4.1.5 临界区

临界区：进程中涉及使用临界资源的代码段

进入区：用于检查是否可以进入临界区的代码段。

退出区：将临界区正被访问的标志恢复为未被访问标志。

剩余区：其他代码

一个访问临界资源的循环进程的描述：

```
while (TRUE) {
    进入区
    临界区
    退出区
    剩余区
}
```

```
item nextConsumed;while (1) {
    while (count == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;  // 临界区，count是临界资源
}
```

在并发系统中，程序执行结果的正确性不仅取决于自身的正确性，还与其在执行过程中能否正确的与其它进程实施同步或互斥有关。必须对临界资源的访问进行控制。

临界区设计的四大准则

空闲让进：当无进程处于临界区，应允许一个请求进入临界区的进程立即进入自己的临界区；

忙则等待：已有进程处于其临界区，其它试图进入临界区的进程必须等待；

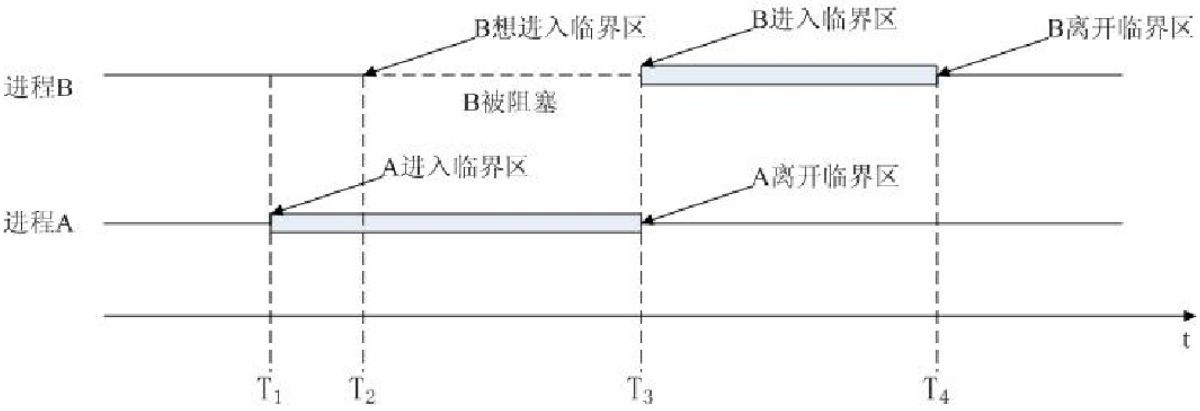
有限等待：等待进入临界区的进程不能"死等"；

让权等待：不能进入临界区的进程，应释放CPU（如转换到阻塞状态）。

其他说法：

- 任何两个进程不能同时处于临界区；
- 不对CPU的速度和数量做任何的假设；
- 临界区外的进程，不得阻塞临界区内的进程；
- 不得使进程在临界区外无休止等待。

临界区的互斥



4.1.6 进程同步机制

- 软件同步机制：使用编程方法解决临界区问题
- 硬件同步机制：使用特殊的硬件指令，可有效实现进程互斥
- 信号量机制：一种有效的进程同步机制，已被广泛应用
- 管程机制：新的进程同步机制

4.2 软件同步机制

4.2.1 单标志法

P0进程	P1进程

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

当P0进程使用完毕后，置 turn 为1。若P1进程暂时不使用该资源，因此不会将turn置0，因此P0无法再次使用该资源，违背了“空闲让进” 原则。

4.2.2 双标志先检查

P0进程	P1进程
bool flag[2];	
while (flag[1]) ;	while (flag[0]) ;
flag[0] = true;	flag[1] = true;
critical section;	critical section;
flag[0] = flase;	flag[1] = flase;
remainder section;	remainder section;

P0 或P1均可无序多次使用该资源。

若以某种次序访问，则出现两个进程同时进入临界区，违背“忙则等待”原则。

4.2.3 双标志后检查

P0进程	P1进程
bool flag[2];	
flag[0] = true;	flag[1] = true;
while (flag[1]) ;	while (flag[0]) ;
critical section;	critical section;
flag[0] = flase;	flag[1] = flase;
remainder section;	remainder section;

若以某种次序访问，则出现两个进程都无法进入临界区，违背“空闲让进”和“有限等待”原则。

4.2.4 Peterson's Algorithm

P0进程	P1进程
bool flag[2];	
int turn = 0;	
flag[0] = true;	flag[1] = true;
turn = 1;	turn = 0;
while (flag[1] && turn == 1) ;	while (flag[0] && turn == 0) ;
critical section;	critical section;
flag[0] = flase;	flag[1] = flase;
remainder section;	remainder section;

若进入while循环，则自旋忙等，违背"让权等待"原则。

4.3 硬件同步机制

4.3.1 屏蔽中断

开关中断指令。关闭中断，屏蔽其他请求，CPU执行完临界区之后才打开中断，可有效保证互斥。

缺点：关中断被滥用、长时间关中断影响效率、多处理机不适用，无法防止其他处理机调度其他进程访问临界区、仅适用于内核进程。

4.3.2 Test-and-Set 指令实现互斥

原子地检查和修改字的内容：

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

进程Pi：

```
while (TestAndSet(lock))    ;
critical section
*lock = false;
remainder section
```

违背"让权等待"，会发生忙等。

4.3.3 Swap指令实现互斥

原子地交换两个变量：

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

进程Pi:

```
do {
     key = true;
     do {
         Swap(lock,key);
     }while (key != false);
     //当key获取到False，则锁测试完成，将锁置为true完成，且条件不成立，退出循环。
     临界区
     lock = false;
     剩余区
}while(true);
```

4.4 信号量机制

1965年，由荷兰学者迪科斯彻Dijkstra提出（P、V分别代表荷兰语的Proberen (test)和Verhogen (increment)），是一种卓有成效的进程同步机制。

信号量-软件解决方案:

- 保证两个或多个代码段不被并发调用
- 在进入关键代码段前，进程必须获取一个信号量，否则不能运行
- 执行完该关键代码段，必须释放信号量
- 信号量有值，为正说明它空闲，为负说明其忙碌

类型

- 整型信号量
- 记录型信号量
- AND型信号量
- 信号量集

4.4.1 整型信号量

用一整数值s表示可用资源数目。

提供两个不可分割的[原子操作]访问信号量

```
wait(S) -----  P（S）:
    while s<=0  ;   /*do no-op*/
    s=s-1;
signal(S) -----  V（S）:
    s=s+1;
```

- Wait(s)又称为P(S)
- Signal(s)又称为V(S)

- 缺点：进程忙等

4.4.2 记录型信号量

数值标明当前可用资源数目；

每个信号量与一个队列关联；

其值只能通过初始化和P（proberen）、V（verhogen）操作来访问。

信号量的类型：

- 公用信号量：用于进程间的互斥，初值通常为1
- 私有信号量：用于进程间的同步，初值通常为0或n

其数据结构表示如下：

```
typedef struct
{
    int value; //表示该类资源可用的数量
    PCB *pointer; //等待使用该类资源的进程排成队列的队列头指针
} semaphore;
```

信号量只能通过初始化和两个标准的原语PV来访问

- 作为OS核心代码执行，不受进程调度的打断。初始化指定一个非负整数值，表示空闲资源总数（又称为"资源信号量"） - - 若为非负值表示当前的空闲资源数，若为负值其绝对值表示当前等待临界区的进程数

初始化指定一个非负整数值，表示空闲资源总数（又称为"资源信号量"）

- 若为非负值表示当前的空闲资源数，若为负值其绝对值表示当前等待临界区的进程数。

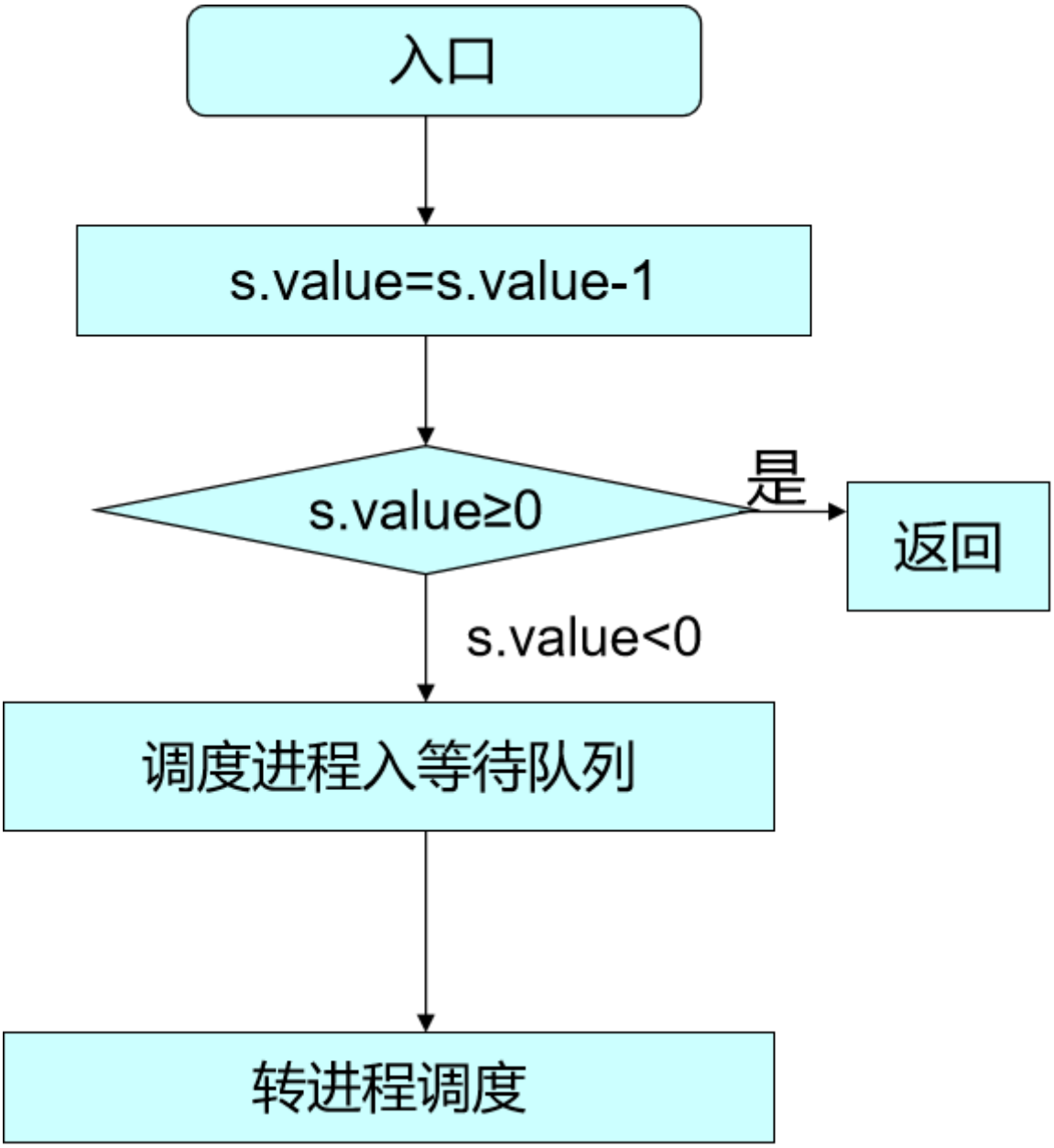
4.4.3 P() / wait() 操作

P操作：荷兰语“proberen”——“检测”之意。意味着请求分配一个单位资源。

P操作 P(s)

```
void P(semaphore *s)
{
    s.value = s.value - 1; //表示申请一个资源
    if ( s.value<0 )  //表示没有空闲资源
    {
        insert (CALLER, s.pointer);
        //将调用进程插入到信号量s的阻塞队列中
        block (CALLER); //自我阻塞
    }
}
```

P原语操作功能流程图：



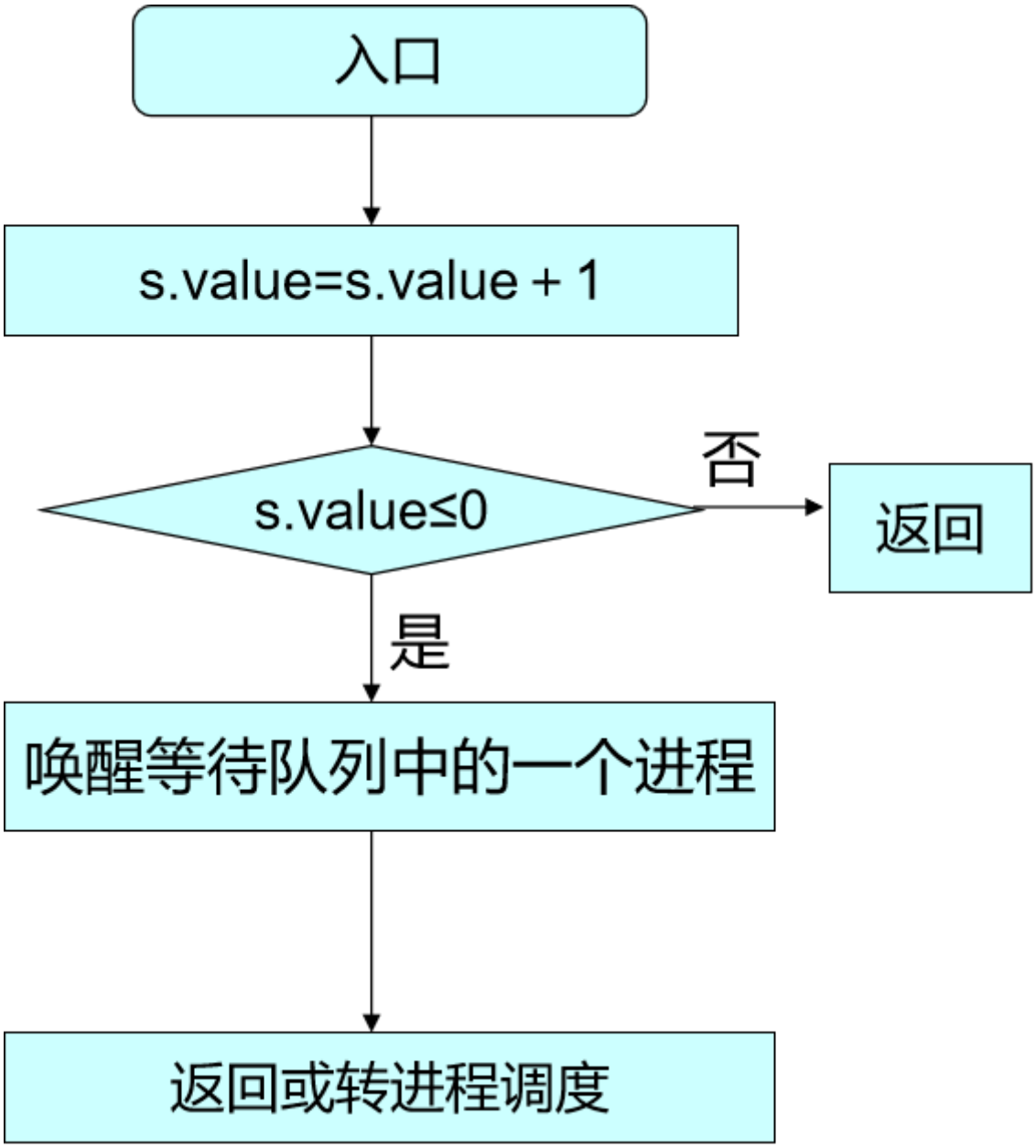
4.4.4 V() / signal() 操作

V操作：荷兰语“verhogen”——“增量”之意，意味着释放一个单位资源。

V操作 V(s)

```
void V(semaphore *s)
{
    PCB *proc_id;  //定义一个PCB指针变量
    s.value = s.value + 1; //表示释放一个资源
    if ( s.value<=0 )  //表示阻塞队列中还有进程
    {
        remove (s.pointer, proc_id );
        //从信号量s对应的阻塞队列首部摘除一个进程
        wakeup(proc_id);  //唤醒该进程，插入就绪队列
    }
}
```

V原语操作功能流程图：



4.4.5 AND信号量操作定义

AND信号量同步的基本思想：将进程再整个运行过程中需要的所有资源，一次性全部分配给进程，待进程使用完后再一起释放。

对若干个临界资源的分配，采用原子操作。

在wait(S)操作中增加了一个AND条件，故称AND同步，或同时wait(S)操作，即Swait（Simultaneous wait）。

```
Swait(S1, S2, …, Sn)  {
    while (TRUE)  {
        if (Si>=1 && … && Sn>=1) {
            for (i =1;i<=n;i++) Si--;;
                break;
        }
        else  {
            place the process in the waiting queue associated with the first Si found with Si<1, and set the program count of this process to the beginning of Swait operation
        }
    }
}
```

```
Ssignal(S1, S2, …, Sn)  {
    while (TRUE) {
        for (i=1 ; i<=n;i++) {
            Si++;
            remove all the process waiting in the queue associated with Si into the ready queue.
        }
    }
}
```

4.4.6 信号量集

在记录型信号量中，wait或signal仅能对某类临界资源进行一个单位的申请和释放，当需要对N个单位进行操作时，需要N次wait/signal操作，效率低下。

扩充AND信号量：对进程所申请的所有资源以及每类资源不同的资源需求量，在一次P、V原语操作中完成申请或释放。

- 进程对信号量Si的测试值是该资源的分配下限值ti，即要求Si≥ti，否则不予分配。一旦允许分配，进程对该资源的需求值为di，即表示资源占用量，进行Si= Si-di操作
- Swait(S1，t1，d1，…，Sn，tn，dn)
- Ssignal(S1，d1，…，Sn，dn)

几种特殊形式

- Swait(S，d，d)：此时在信号量集中只有一个信号量S，但允许它每次申请d个资源，当现有资源数少于d时，不予分配。
- Swait(S，1，1)：此时的信号量集已蜕化为一般的记录型信号量(S > 1时)或互斥信号量(S=1时)。
- wait(S，1，0)：这是一种很特殊且很有用的信号量操作。当S≥1时，允许多个进程进入某特定区；当S变为0后，将阻止任何进程进入特定区。换言之，它相当于一个可控开关。

4.4.7 信号量的作用

实现进程互斥：设置互斥信号量

实现前趋关系

实现进程同步：设置同步信号量

4.4.8 信号量解决同步与互斥的一般方法

1. 对问题进行分析。搞清问题属于哪类问题，即是互斥问题，还是同步问题，还是两者的混合问题；
2. 分清并发进程之间的互斥关系和同步关系。即搞清同步存在于哪些进程之间，互斥存在于哪些进程之间；
3. 搞清同步的基本方式(具体问题具体分析)，互斥所涉及的临界资源的数量；
4. 设定信号量的个数和初值。
   - 互斥信号量的个数等于临界资源的数目，且其初值均为1；
   - 同步信号量的个数及其初值则要根据具体问题而定，没有统一的方法。
   - 值得注意的是每个信号量必须置一次且只能置一次初值，初值不能为负数；
5. 设计出相应的管理方法，给出其并发算法。
   - 尽可能地利用现在已经发明的较为成熟的实现方法

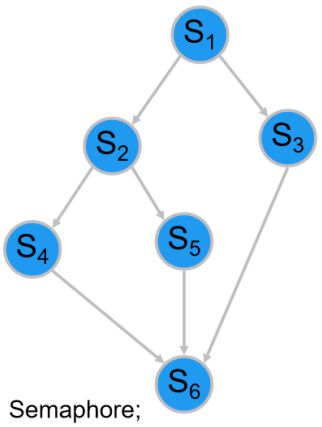
4.4.9 实现进程互斥

```
semaphore mutex;
mutex=1; // 初始化为 1

while(1)
{
    wait(mutex);
    临界区;
    signal(mutex);
    剩余区;
}
```

为了正确地解决一组并发进程对临界资源的竞争使用，我们引入一个互斥信号量，用mutex表示，对于互斥使用的资源，其信号量的初值就是系统中这个资源的数量。

4.4.10 实现前趋关系



S1()
{
.
.
.
.
发信号通知S2;
发信号通知S3
}

S2()
{
收到S1发出的信号
.
.
.
发信号通知S4;
发信号通知S5;
}

S3()
{
收到S1发出的信号
.
.
.
发信号通知S6;
}

S4()
{
收到S2发出的信号
.
.
.
发信号通知S6;
}

S5()
{
收到S2发出的信号
.
.
.
发信号通知S6;
}

S6()
{
收到S3发出的信号
收到S4发出的信号
收到S5发出的信号
.
.
}

```
main(){
   Semaphore a,b,c,d,e,f,g;
    a.value=0;b.value=0;c.value=0;
    d.value=0;e.value=0;f.value=0;g.value=0;
    cobegin
        { S1;signal(a);signal(b); }
        { wait(a);S2;signal(c) ;signal(d);}
        { wait(b);S3;signal(e); }
        { wait(c);S4;signal(f); }
        { wait(d);S5;signal(g); }
        { wait(e);wait(f);wait(g);S6; }
    coend
}
```

4.4.11 实现进程同步

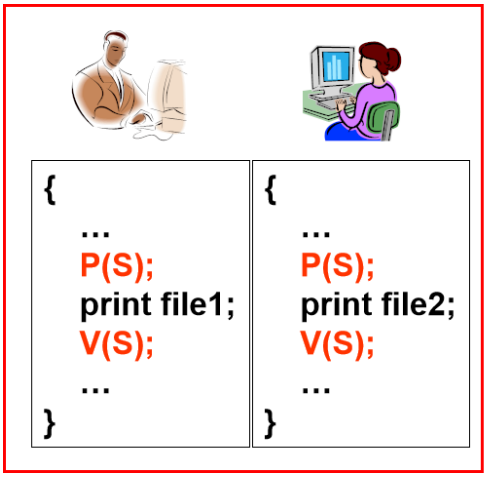
P1和 P2 需要 C1代码段比C2代码段先运行：

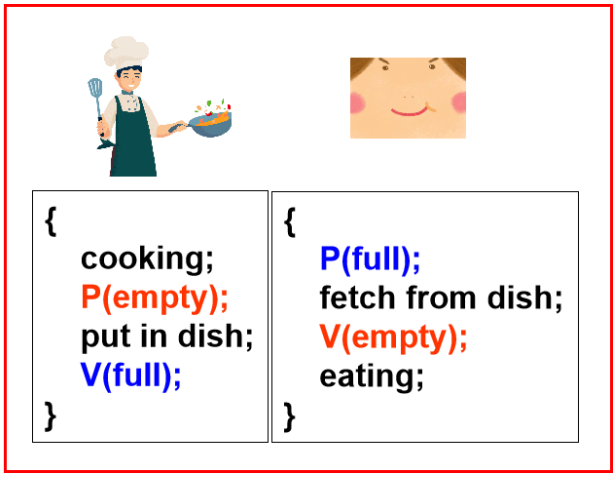
```
semaphroe s = 0;  // 主要用于传递信息
P1() {
    C1;
    V(s);
    ...
}
P2() {
    ...
    P(s);
    C2;
}
```

4.4.12 PV操作应注意的问题

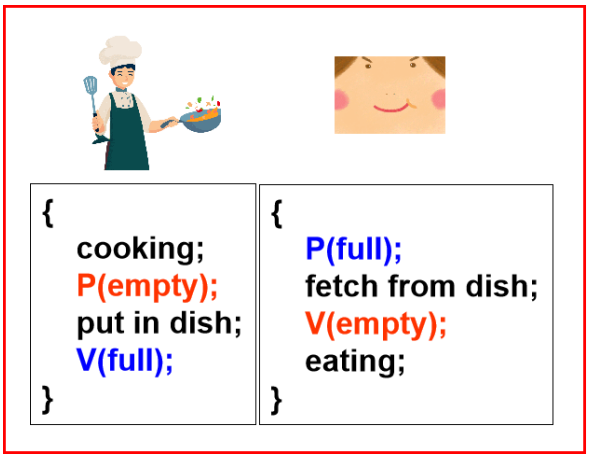
P、V操作总是成对出现的

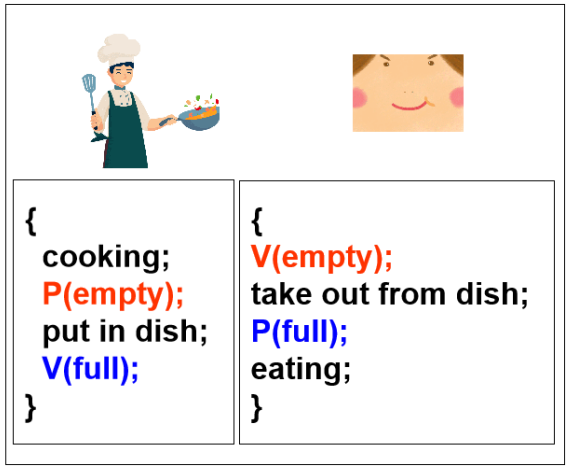
- 互斥操作时他们处于同一进程中；
- 同步操作时他们处于不同进程中。





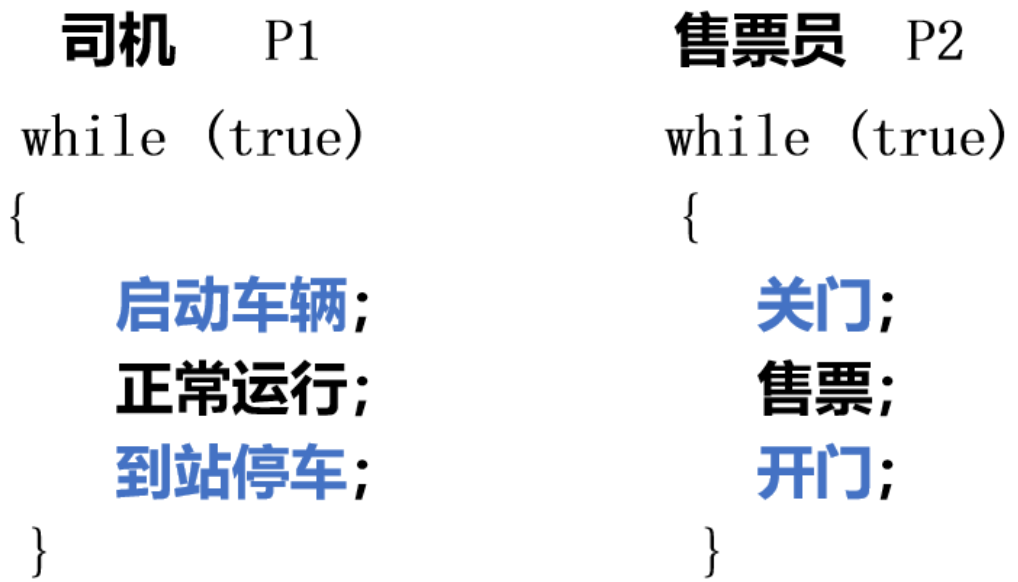
P、V操作的位置十分重要，放置不当会造成严重后果，注意逻辑关系。





4.4.13 练习

【1】、公共汽车中的司机和售票员




```
p1()
{
    while(1)
    {
        P(go);
        启动汽车;
        正常行驶;
        到站停车;
        V(stop);
    }
}
p2()
{
    while(1)
    {
        关门;
        V(go);
        售票;
        P(stop);
```

```
20          开门;
21      }
22  }
```

4.4.14 信号量与PV操作的总结

信号量 S 的物理含义

- S.value>0，表示有S.value个资源可用；
- S.value=0，表示无可用资源；
- S.value<0，则表示有| S.value |个进程在等待该资源的阻塞队列中排队；
- P(S)表示申请一个资源；
- V(S)表示释放一个资源；
- 信号量的初值应该大于等于0；
- 互斥信号量的初值一般是资源的数量；
- 同步信号量的初值一般为0。

P、V操作必须成对出现

- 对同一个信号量，有一个P操作就一定有一个V操作；
- 当为互斥操作时，PV操作处于同一进程中；
- 当为同步操作时，PV操作处于不同的进程中；
- 如果P(S1)和P(S2)两个操作在一起，那么P操作的顺序至关重要
    - 一个同步P操作与一个互斥P操作在一起时，同步P操作在互斥P操作前；
- 而两个V操作无关紧要；

P、V操作的优缺点

- 优点：简单，而且表达能力强（用P、V操作可解决任何同步互斥问题）；
- 缺点：不够安全，P、V操作使用不当会出现死锁；遇到复杂同步互斥问题时实现复杂。

4.5 管程机制

一个管程定义了一个数据结构和能为并发进程所执行（在该数据结构上）的一组操作，这组操作能同步进程和改变管程中的数据。

语法描述：

```
Monitor monitor_name {  // 管程名
    share variable declarations;  // 共享变量说明
    cond declarations;  // 条件变量说明
    public:  // 能被进程调用的过程
        void P1(...){...}  // 对数据结构操作的过程
        void P2(...){...}
        ...
    ...
    {  // 管程主体
        initialization code;  // 初始化代码
        ...
    }
}
```

4.5.1 管程功能

互斥：

- 管程中的变量只能被管程中的操作访问
- 任何时候只有一个进程在管程中操作
- 类似临界区
- 由编译器完成

同步：

- 条件变量
- 唤醒和阻塞操作

4.5.2 条件变量

```
condition x, y;
```

条件变量的操作

- 阻塞操作：wait
- 唤醒操作：signal

x.wait()：进程阻塞直到另外一个进程调用x.signal()

x.signal()：唤醒另一个进程

条件变量问题：

- 管程内可能存在不止1个进程

处理方式：

- P等待，直到Q离开管程或等待另一条件
- Q等待，直到P离开管程或等待另一条件

4.6 生产者消费者问题

生产者-消费者问题是相互合作进程关系的一种抽象。

利用记录型信号量实现：

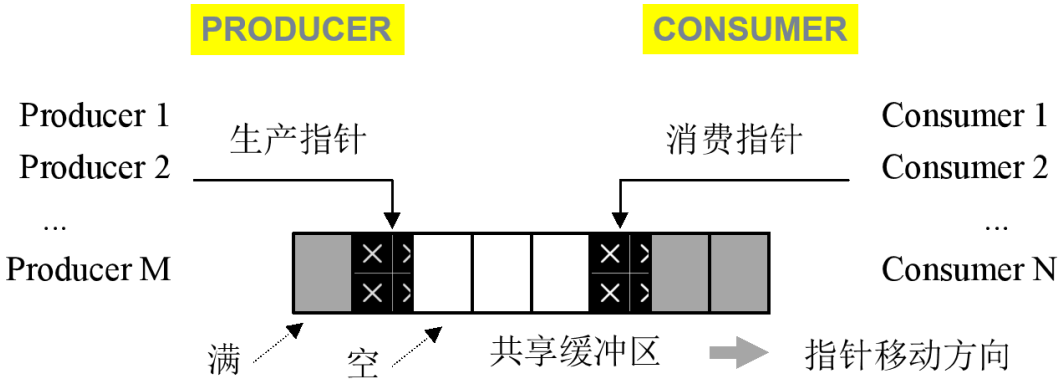
- 假定，在生产者和消费者之间的公用缓冲池中，具有n个缓冲区，可利用互斥信号量mutex使诸进程实现对缓冲池的互斥使用；
- 利用资源信号量empty和full分别表示缓冲池中空缓冲区和满缓冲区的数量。
- 又假定这些生产者和消费者相互等效，只要缓冲池未满，生产者便可将消息送入缓冲池；只要缓冲池未空，消费者便可从缓冲池中取走一个消息。

问题描述

生产者（M个）：生产产品，并放入缓冲区

消费者（N个）：从缓冲区取产品消费

问题：如何实现生产者和消费者之间的同步和互斥?



流程

```
生产者:
{
  …
  生产一个产品
  …

  …
  把产品放入指定缓冲区
  …


}
```

```
消费者:
{
  …

  …
  从指定缓冲区取出产品
  …

  …
  消费取出的产品
  …
}
```

互斥分析

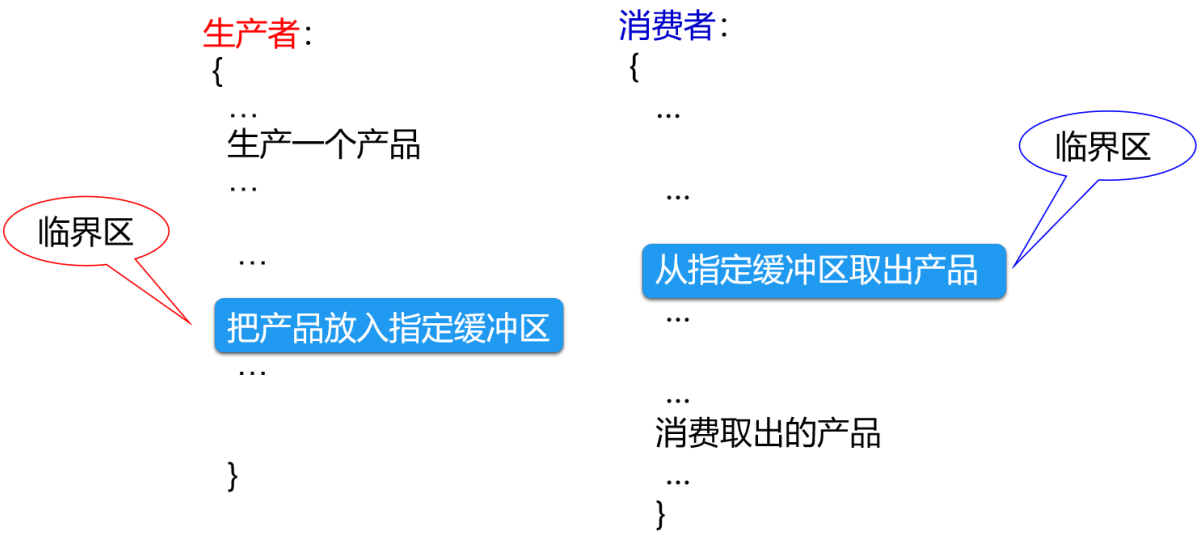
临界资源:

- 生产者
  - 把产品放入指定缓冲区
  - in:所有的生产者对in指针需要互斥
  - count：所有生产者消费者进程对count互斥

```
buffer[in] = nextp;
in = (in + 1) % N;
count++;
```

- 消费者
  - 从指定缓冲区取出产品
  - out:所有的消费者对out指针需要互斥
  - count：所有生产者消费者进程对count互斥
  - ```
    nextc = buffer[out];
    out = (out + 1) % N;
    count--;
    ```

划分临界区



增加互斥机制


    ```

同步分析

两者需要协同的部分

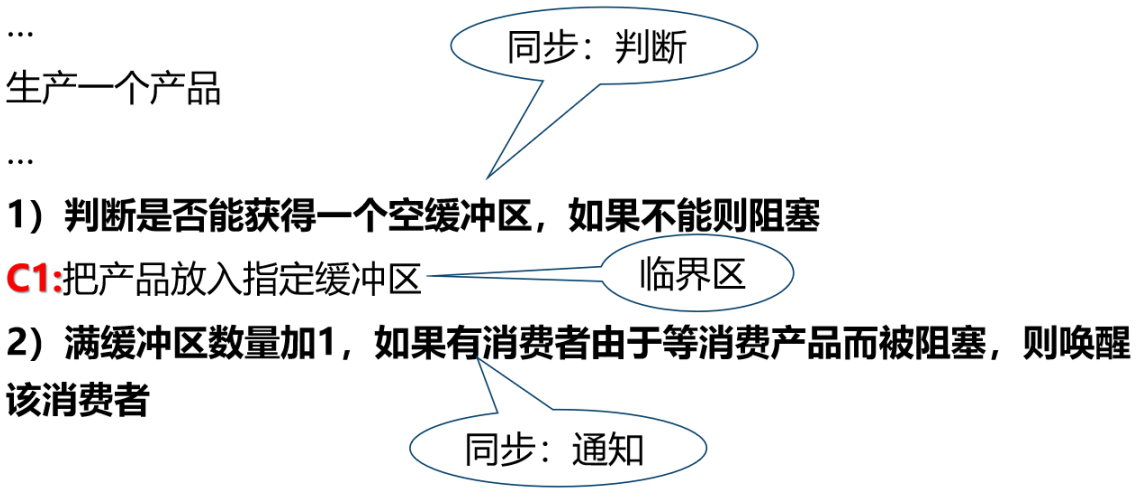
- 生产者：把产品放入指定缓冲区（关键代码C1）
- 消费者：从满缓冲区取出一个产品（关键代码C2）

三种运行次序（不同条件下不同运行次序）

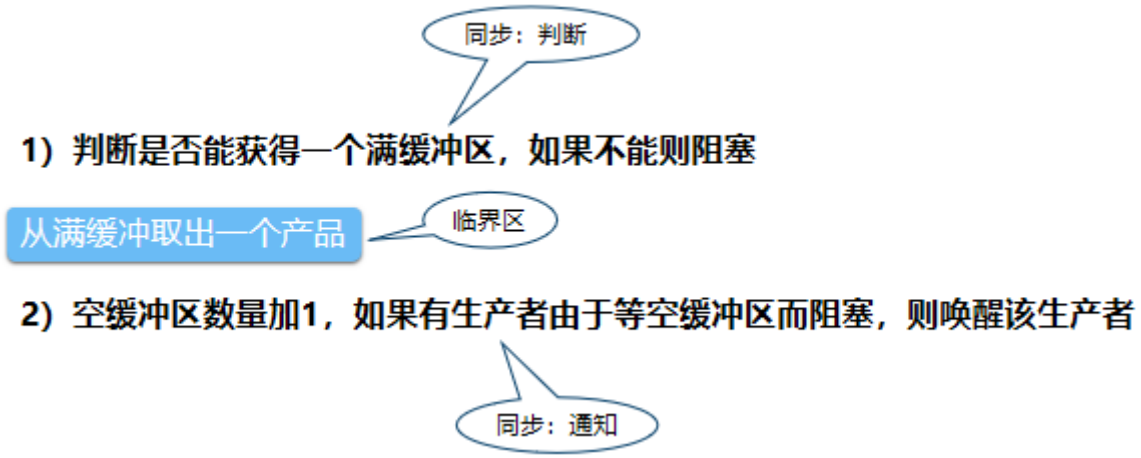
- 所有缓冲区空时：
- 所有缓冲区满时：
- 缓冲区有空也有满时：

算法描述

生产者



消费者



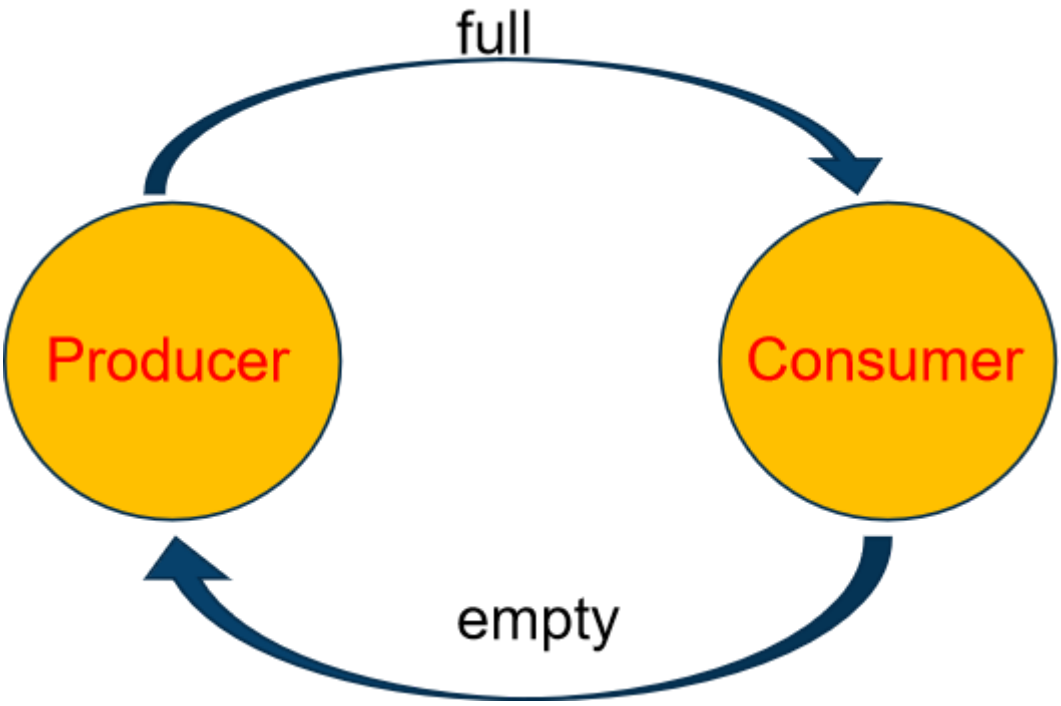
同步信号量定义

```
semaphore *full, *empty, *mutex;  //full:满缓冲区数量    empty：空缓冲区数量
```

初始化：

```
full->value = 0;
empty->value = n;
mutex ->value = 1;
```

解决方法




```
Semaphore empty=n,full=0;
//同步信号量

Semaphore Sin=1,Sout=1;
//资源in的互斥信号量，资源out的互斥信号量

生产者:
{
    …
    生产一个产品
    …
    wait(empty);  // 当empty大于0时，表示有空缓冲区，继续执行；否则，表示无空缓冲区，当前生产者阻塞
    wait(mutex);
    …
    C1：把产品放入指定缓冲区
    …
    signal(mutex);
    signal(full);  // 把full值加1，如果有消费者等在full的队列上，则唤醒该消费者
}
消费者:
{
    …
    wait(full);   // 当full大于0时，表示有满缓冲区，继续执行；否则表示无满缓冲区，当前消费者阻塞。
    wait(mutex);
    …
    C2：从指定缓冲区取出产品
```

```
    …
    signal(mutex);
    signal(empty);  // 把empty值加1，如果有生产者等在empty队列上，则唤醒该生产者。
    …
    消费取出的产品
    …
}
```

4.7 哲学家就餐问题

问题描述

五个哲学家的生活方式：交替思考和进餐共用一张圆桌，分别坐在五张椅子上在圆桌上有五个碗和五支筷子平时哲学家思考，饥饿时便试图取用其左、右最靠近他的筷子，只有在拿到两支筷子时才能进餐进餐毕，放下筷子又继续思考。

记录型信号量解决方案

```
Semaphore chopstick[5] = {1,1,1,1,1};
Philosopher i:
do {
    wait(chopStick[i]);  // get left chopstick
    wait(chopStick[(i + 1) % 5]);   // get right chopstick
    …
    // eat for awhile
    …
    signal(chopStick[i]); //return left chopstick
    signal(chopStick[(i + 1) % 5]); // return right chopstick
    …
    // think for awhile
    …
} while (true)
```

存在问题

- 可能引起死锁，如五个哲学家同时饥饿而各自拿起左筷子时，会使信号量chopstick均为0；因此他们试图去拿右筷子时，无法拿到而无限期等待。

解决方案

- 最多允许4个哲学家同时坐在桌子周围
- 仅当一个哲学家左右两边的筷子都可用时，才允许他拿筷子。
- 给所有哲学家编号，奇数号的哲学家必须首先拿左边的筷子，偶数号的哲学家则反之

4.8 读者-写者问题

有两组并发进程：读者和写者，共享一组数据区。

要求：

- 允许多个读者同时执行读操作；
- 不允许读者、写者同时操作；
- 不允许多个写者同时操作。

分类：

- 读者优先(第一类读者写者问题)
- 写者优先(第二类读者写者问题)

问题描述

一个数据对象(文件、记录)可以为多个并发进程共享。其中有的进程只需要读其中的内容，我们称为“读者”；有的进程负责更新其中内容(读/写)，我们称为“写者”。“读者”可以同时读取共享数据对象；“写者”不能和其它任何进程同时访问数据对象。如何实现？

读者优先

如果读者来：

- 无读者、写者，新读者可以读。
- 有写者等，但有其它读者正在读，则新读者也可以读。
- 有写者写，新读者等。

如果写者来：

- 无读者，新写者可以写。
- 有读者，新写者等待。
- 有其它写者，新写者等待。

分析

- “读 - 写”：互斥访问
- ”写 - 写”：互斥访问
- “读 - 读”：允许同时访问

第一类读者—写者问题：“读者”优先，只要有读进程在读，写进程被迫等待。

```
// 设置信号量：
semaphore rmutex, wmutex; //公用信号量，用于互斥
rmutex=1; wmutex=1; //设置初值
int  readcount; //计数，用于记录读者的数目

读者进程：
P(rmutex); //对readcount互斥访问
readcount ++; //读者数目加1
if (readcount==1)   //第一个读进程
P(wmutex); //申请读写锁
V(rmutex); //释放readcount
reading;
P(rmutex); //对readcount互斥
readcount --;
if (readcount==0)  //最后一个读进程
V(wmutex); //释放data资源
V(rmutex); //释放readcount

写者进程：
P(wmutex); //申请使用data资源
writing;
```

```
V(wmutex); //释放data资源
```

写者优先

多个读者可以同时进行读

写者必须互斥（只允许一个写者写，也不能读者写者同时进行）

写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）

```
Semaphore wmutex,rmutex,S;
wmutex=rmutex=1;
S=1;
readcount=0
writer(){
    while(1){
        P(S);
        P(wmutex);
        writing;
        V(wmutex);
        V(S);
    }
}
reader(){
    while(1){
        P(S);
        P(rmutex);
        readcount++;
        if(readcount==1)P(wmutex);
        V(rmutex);
        V(S);
        reading;
        P(rmutex);
        readcount--;
        if(readcount==0)V(wmutex);
        V(rmutex);
    }
}
```

4.9 Linux进程同步机制

4.9.1 关于信号量的讨论

信号量的使用：

- 信号量必须置一次且只能置一次初值，初值不能为负数
- 除了初始化，只能通过执行P、V操作来访问信号量

使用中存在问题：

- 死锁
- 饥饿

死锁：两个或多个进程无限期地等待一个事件的发生，而该事件正是由其中的一个等待进程引起的。

```
例如：S和Q是两个初值为1的信号量
        P0   P1
        P(S);   P(Q);
        P(Q);   P(S);
        ...     ...
        V(S);   V(Q);
        V(Q)    V(S);
```

饥饿：无限期地阻塞，进程可能永远无法从它等待的信号量队列中移去（只涉及一个进程）。

4.9.2 互斥分析基本方法

查找临界资源-->划分临界区-->定义互斥信号量并赋初值-->在临界区前的进入区加wait操作；退出区加signal操作。

4.9.3 同步分析基本方法

①找出需要同步的代码片段（关键代码）--> ②分析这些代码片段的执行次序 --> ③增加同步信号量并赋初始值 --> ④在关键代码前后加wait和signal操作。

4.9.4 关于PV操作的讨论

信号量 S 的物理含义

- S.value>0，表示有S.value个资源可用；
- S.value=0，表示无可用资源；
- S.value<0，则表示有| S.value |个进程在等待该资源的阻塞队列中排队；
- P(S)表示申请一个资源；
- V(S)表示释放一个资源；
- 信号量的初值应该大于等于0；
- 互斥信号量的初值一般是资源的数量；
- 同步信号量的初值一般为0。

P、V操作必须成对出现

- 对同一个信号量，有一个P操作就一定有一个V操作；
- 当为互斥操作时，PV操作处于同一进程中；
- 当为同步操作时，PV操作处于不同的进程中；
- 如果P(S1)和P(S2)两个操作在一起，那么P操作的顺序至关重要
    - 一个同步P操作与一个互斥P操作在一起时，同步P操作在互斥P操作前；
- 而两个V操作无关紧要；

4.9.5 信号同步的缺点

同步操作分散：信号量机制中，同步操作分散在各个进程中，使用不当就可能导致各进程死锁（如P、V操作的次序错误、重复或遗漏）；

不利于修改和维护：各模块的独立性差，任一组变量或一段代码的修改都可能影响全局；

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

正确性难以保证：操作系统或并发程序通常很大，很难保证这样一个复杂的系统没有逻辑错误。

4.9.6 Linux同步机制

Linux并发的主要来源：中断处理、内核态抢占、多处理器的并发。

同步方法：

- 原子操作
- 自旋锁（spin lock）：不会引起调用者阻塞
- 信号量（Semaphore）
- 互斥锁（Mutex）
- 禁止中断（单处理器不可抢占系统）

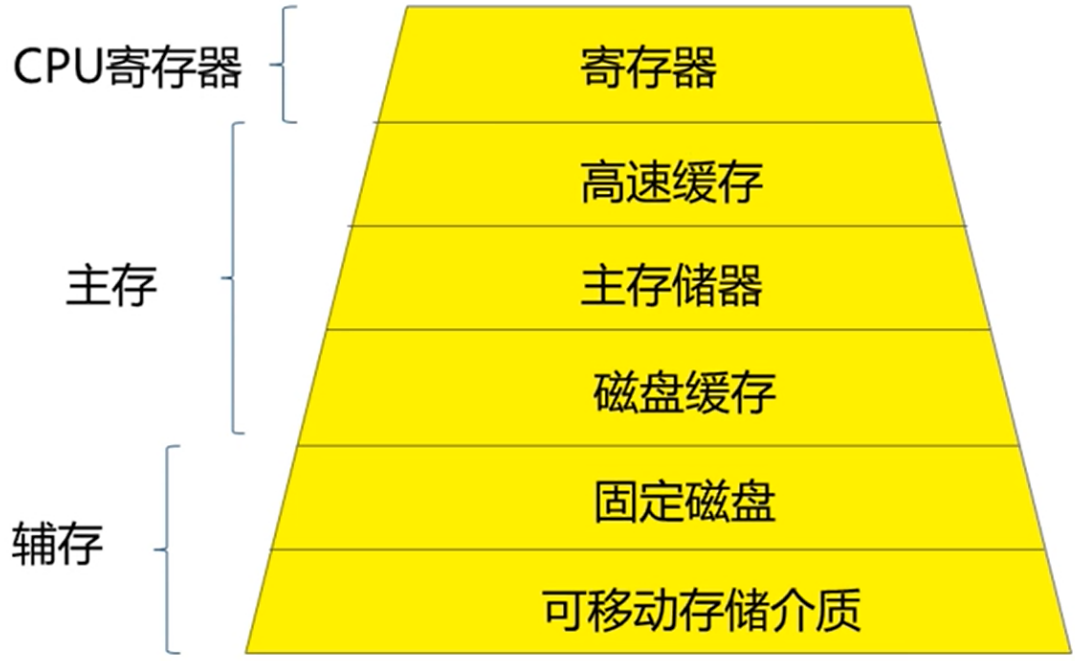
5. 存储器管理

5.1 存储器的层次结构

存储层次

- CPU寄存器
- 主存：高速缓存、主存储器、磁盘缓存
- 辅存：固定磁盘、可移动介质

层次越高，访问速度越快，价格也越高，存储容量也最小。



寄存器和主存掉电后，存储的信息不再存在；辅存的信息长期保存。

5.1.1 寄存器

寄存器访问速度最快，完全能与CPU协调工作，但价格却十分昂贵，因此容量不可能做得很大。

寄存器的长度一般以字(word)为单位。寄存器的数目，对于当前的微机系统和大中型机，可能有几十个甚至上百个；而嵌入式计算机系统一般仅有几个到几十个。

寄存器通常用于加速存储器的访问速度，如用寄存器存放操作数，或用作地址寄存器加快地址转换速度等。

5.1.2 主存储器

主存储器(简称内存或主存)是计算机系统中一个主要部件，用于保存进程运行时的程序和数据，也称可执行存储器，材质以DRAM为主。

由于主存储器的访问速度远低于CPU执行指令的速度，为缓和这一矛盾，在计算机系统中引入了寄存器和高速缓存。

5.1.3 高速缓存

高速缓存是现代计算机结构中的一个重要部件，其容量大于或远大于寄存器，而比内存约小两到三个数量级左右，从几十KB到几MB，访问速度快于主存储器。

根据程序执行的局部性原理(即程序在执行时将呈现出局部性规律，在一较短的时间内，程序的执行仅局限于某个部分)，将主存中一些经常访问的信息存放在高速缓存中，减少访问主存储器的次数，可大幅度提高程序执行速度。

5.1.4 磁盘缓存

由于目前磁盘的I/O速度远低于对主存的访问速度，因此将频繁使用的一部分磁盘数据和信息，暂时存放在磁盘缓存中，可减少访问磁盘的次数。

总结：

- 可执行存储器：寄存器和主存储器。
- 主存储器：内存或主存。
- 寄存器：访问速度最快，与CPU协调工作，价格贵。
- 高速缓存：速度介于寄存器和存储器之间。
    - 备份主存主常用数据，减少对主存储器的访问次数；
    - 缓和内存与处理机之间的矛盾。
- 磁盘缓存：
    - 暂时存放频繁使用的一部分磁盘数据和信息；
    - 缓和主存和I/O设备在速度上的不匹配；
    - 利用主存的部分空间，主存可看成辅存的高速缓存。

5.2 程序的装入和链接

在多道程序环境下，要使程序运行，必须先为之创建进程。

而创建进程的第一件事，便是将程序和数据装入内存。

程序的运行步骤：

- 编译：由编译程序(Compiler)对源程序进行编译，形成若干个目标模块
- 链接：由链接程序(Linker)将目标模块和它们所需要的库函数链接在一起，形成一个完整的装入模块
- 装入：由装入程序(Loader)将装入模块装入内存

相关概念

逻辑地址：目标代码的相对编址

物理地址：目标代码的绝对编址

地址空间：目标代码用逻辑地址编址所限定的区域

存储空间：内存若干存储单元用物理地址编址所限定的区域

地址重定位：当程序被装入内存时，程序的逻辑地址被转换成内存的物理地址的过程

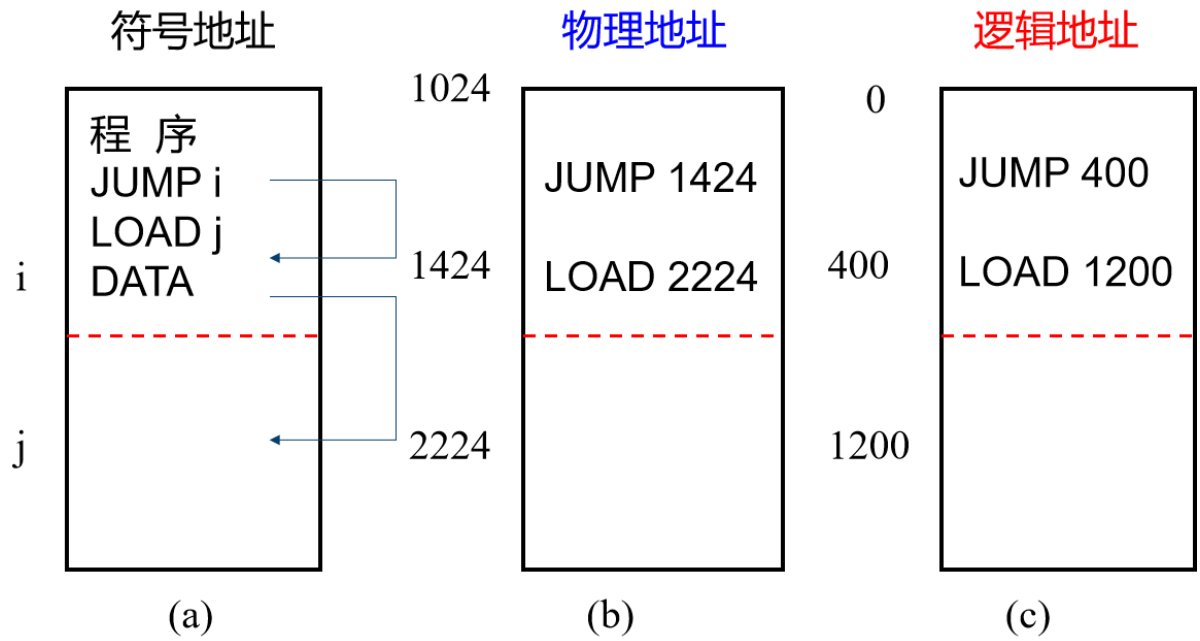
5.2.1 物理地址和逻辑地址

物理地址（绝对地址）：物理内存的地址，内存以字节为单位编址。

- 物理地址空间：所有物理地址的集合

逻辑地址（虚拟地址、相对地址）：由CPU产生的地址，即程序编译后使用的相对于0字节的地址。

- 逻辑地址空间：由程序所生成的所有逻辑地址的集合



5.2.2 重定位的方式

静态重定位：目标代码装入内存时，一次性进行地址转换。

动态重定位：目标代码装入内存时，先不进行地址转换（即原代码装入），在执行时，再实施地址转换。

5.2.3 内存保护

目的：

- 保护OS不被用户访问
- 保护用户进程不会相互影响

实现：硬件

- 基地址寄存器：保存最小的合法物理内存地址（基地址）
- 界限寄存器：保存合法的地址范围大小（界限地址）
- 内存空间保护的实现：判断"基地址≤物理地址 < （基地址+界限地址）"是否成立。

5.2.4 程序的装入

绝对装入方式

编译时产生的地址使用绝对地址（由编译器或程序员完成）。程序或数据被修改时，需要重新编译程序。

可重定位装入方式

编译后的目标模块使用相对地址。在装入时，由装入程序完成重定位（静态重定位）。要求连续空间，全部装入内存且位置不可移动。

- 逻辑地址转换为物理地址的过程，称为重定位，也称为地址交换。

动态运行时装入方式

编译后的目标模块使用相对地址。装入内存时并不地址转换，在真正执行时，完成动态重定位。

5.2.5 程序的链接

静态链接

在程序运行前，将各目标模块及它们所需的库函数链接成一个完整的装配模块，以后不再拆开。

对相对地址进行修改；变换外部调用符号。

装入时动态链接

在装入内存时，采用边装入边链接的链接方式。

便于修改和更新。

便于实现对目标模块的共享。

运行时动态链接

将某些目标模块的链接推迟到执行时才执行。即在执行过程中，若发现一个被调用模块尚未装入内存时，立即由OS去找到该模块并将它装入内存，并把它链接到调用者模块上。

加快装入过程，节省大量的内存空间。

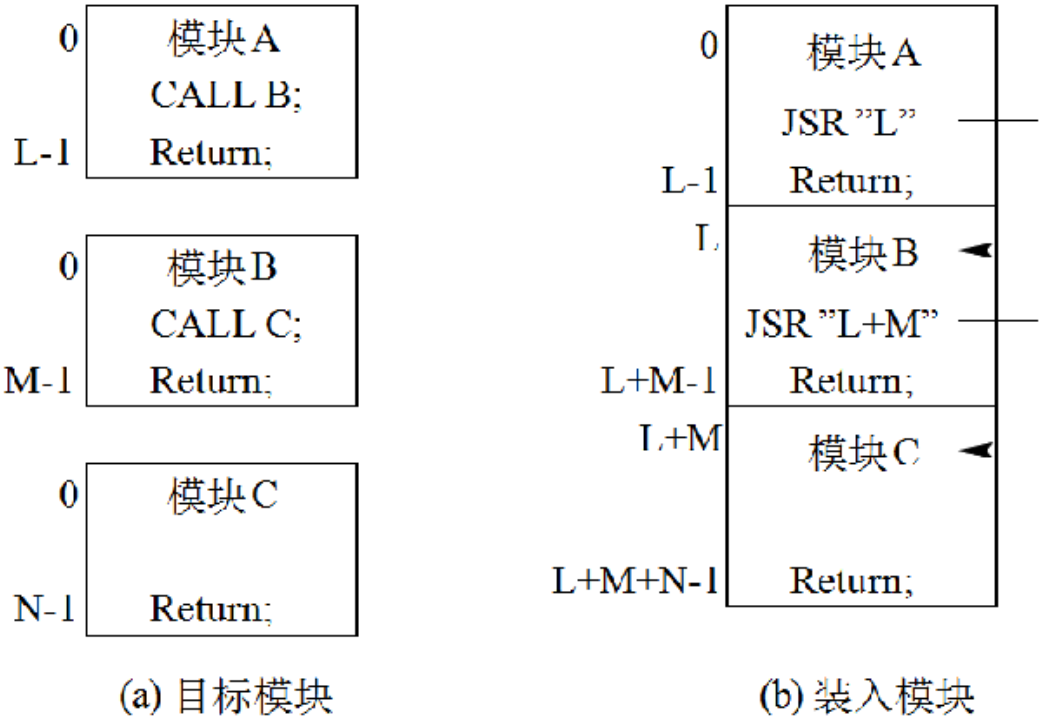
5.2.6 静态链接方式（Static Linking）

特点：

- 一次链接，n次运行
- 得到完整的可装入模块，不可再拆
- 不灵活：不管有用与否的模块都将链接到装入模块，同时导致内存利用率较低
- 不利于模块的修改和升级

5.2.7 装入时动态链接（Load-time Dynamic Linking）

用户源程序经编译后所得的目标模块，是在装入内存时边装入边链接的，即在装入一个目标模块时，若发生一个外部模块调用事件，将引起装入程序去找出相应的外部目标模块，并将它装入内存，还要按照图所示的方式来修改目标模块中的相对地址。



程序链接示意图

5.2.8 运行时动态链接(Run-time Dynamic Linking)

装入时动态链接是将所有可能要运行到的模块都全部链接在一起并装入内存，显然这是低效的，因为往往会有些目标模块根本就不运行。比较典型的例子是作为错误处理用的目标模块，如果程序在整个运行过程中都不出现错误，则显然就不会用到该模块。

运行时动态链接方式是对装入时链接方式的一种改进。这种链接方式是将对某些模块的链接推迟到程序执行时才进行链接，亦即，在程序执行过程中，当发现一个被调用模块尚未装入内存时，才立即由OS去找到该模块并将之装入内存，把它链接到调用者模块上。凡在本次执行过程中未被用到的目标模块，都不会被调入内存和被链接到装入模块上，这样不仅可加快程序的装入过程，而且可节省大量的内存空间。

优点：

- 便于共享：多个进程可共用一个DLL模块，节省了内存。
- 为部分装入提供了条件(运行时动态链接)：一个进程可由若干DLL模块构成，按需装入。
- 便于模块的修改和升级：只要被修改模块的接口不变，则该模块的修改不会引发其它模块的重新编译。
- 改善了程序的可移植性：可面向不同的应用环境开发同一功能模块的不同版本，根据当前的环境载入匹配的模块版本。

缺点：

- 增加了程序执行时的链接开销。(每次运行都需链接)
- 模块数量众多，增加了模块的管理开销。

5.3 对换与覆盖

5.3.1 对换（Swapping）

对换：把内存中暂时不能运行的进程或者暂时不用的程序和数据，调出到外存上，以便腾出足够的内存空间，再把已具备运行条件的进程或进程所需的程序或数据，调入内存。

对换是提高内存利用率的有效措施，广泛应用于OS中。

类型

- 整体对换：对换以整个进程为单位，也称为进程对换。
- 页面(分段)对换：对换是以“页”或“段”为单位进行的，又统称为“部分对换”

为了实现进程对换，系统必须能实现三方面的功能

- 对换空间的管理
- 进程的换出
- 进程的换入

5.3.2 对换区的管理

主要目标：

- 提高进程换入和换出的速度
- 提高文件存储空间的利用率次之
- 应采用连续分配方式，很少考虑碎片问题