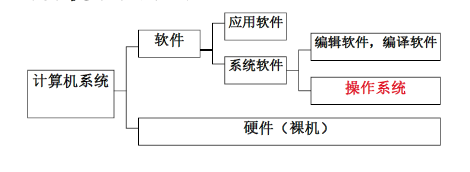
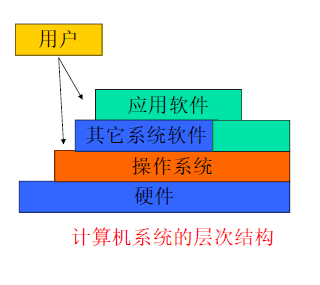
操作系统

1. 绪论
2. 操作系统的目标和作用
3. 计算机系统组成



1. 计算机系统的层次结构



1. 计算机系统中的OS

\*是在硬件基础上的第一层软件

\*是其他软件和硬件之间的接口

1. 配置OS的目标

\*方便用户使用：提供接口

\*有效性：提高系统的资源利用率和吞吐量

\*可扩展性：适应硬件和体系结构的发展和功能需求的变化

\*开放性：应用在不同系统中具有可移植性和互操作性，通过网络有效集成，系统应遵循标准规范

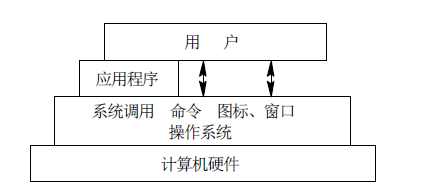
1. OS定义

有效地控制和管理计算机软硬件资源，合理地组织计算机工作流程，以提高资源利用率，并为用户提供强有力的使用功能和灵活方便的使用环境的程序集合。

1. OS的功能

① OS作为用户与计算机硬件系统之间的接口

用户可使用命令方式、系统调用、GUI三种方式通过OS利用计算机软硬件资源运行自己的程序。



② OS作为计算机系统资源管理者

\*资源：硬件资源（处理器、存储器、I/O设备）

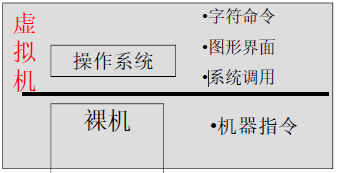
软件资源（数据、程序）

\*管理：处理机管理、存储器管理（内存的分配与回收）、I/O设备管理、文件管理

③ OS对计算机资源的抽象

虚拟机：裸机（直接面对硬件）+ 管理软件

1. OS层次结构



1. 推动OS发展的动力

\*提高计算机资源利用率

\*方便用户

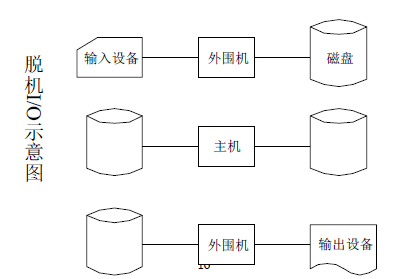
\*器件的不断更新换代

\*计算机系统结构的不断发展

\*新用户，新的服务、工具和功能

1. 操作系统的发展过程
2. 无操作系统的计算机系统（1945-1955）

特点：使用机器语言，通过控制台控制，用户独占资源，使用人工操作的方式，使用脱机输入输出的方式



\*\*脱机输入/输出方式优点：

1. 减少CPU的空闲时间
2. 提高I/O速度
3. 单道批处理系统（1955-1965）

\*特点：使用高级语言，出现监控程序

作业控制（用户提交作业 ——> 操作员把一批作业存入磁盘 ——> 常驻监控程序自动从磁盘装入程序、运行、撤出作业）

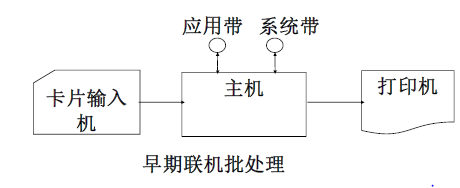
自动性、顺序性、单道性

\*批处理方式

1. 联机批处理（CPU直接处理输入输出）

硬件控制方式：作业的输入、计算和输出在CPU直接控制下进行

模型：

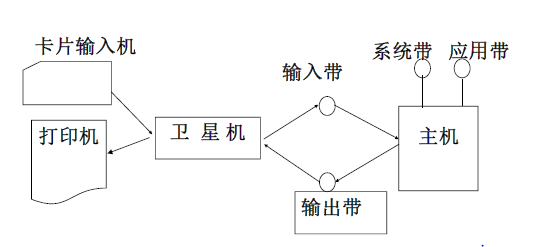


问题：输入输出时，CPU处于等待状态

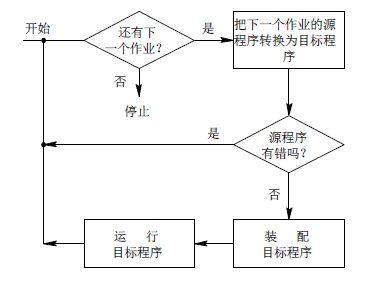
1. 脱机批处理

硬件控制方式：专门的卫星机负责输入输出，缓冲技术的一种，主机和卫星机可并行工作

模型：



\*处理过程



1. 集成电路时代的操作系统

（1）多道批处理系统（1965-1980）——> 出现标志操作系统的形成

\*特点：内存驻留多道作业，各道程序轮流占用CPU，交替执行

\*目的：提高CPU利用率，充分发挥计算机系统部件的并行性（CPU和I/O设备）

\*优点：提高CPU、内存、I/O设备利用率；提高资源利用率；增大作业吞吐量

\*缺点：作业周转时间长，不能交互计算，调试程序困难

\*需要解决的问题：处理机、内存、I/O设备、文件、作业管理问题，用户接口问题

1. 分时系统

\*工作方式：用户在终端控制程序运行，把系统资源进行时间上的分割，每个时间段称为一个时间片，每个用户依次轮流使用时间片（时间片的划分是影响系统的关键因素）

\*特点：多路性（多个用户共享系统资源，提高资源利用率）、独立性（用户操作互不干扰）、及时性、交互性

1. 实时系统

\*工作方式：系统能及时响应外部事件请求，在规定时间内完成对该事件的处理，控制所有实时任务协调一致的进行

\*应用需求：实时控制、实时信息处理

\*实时任务：

1. 周期性实时任务
2. 非周期性实时任务（都必须联系一个截止时间）

开始截止时间 or 完成截止时间

1）硬实时任务（必须满足截止时间要求）

2）软实时任务（截止时间要求不严格）

1. 网络操作系统（1980）

特点：多用户多任务，界面友好，软件丰富；统一管理网络中共享资源的使用和协调处理任务

功能：网络通讯、提供网络接口、提高网络服务质量

1. 分布式操作系统

特点：资源进一步共享、透明性、自治性

优点：处理能力增强、速度更快、可靠性增强

1. 嵌入式操作系统
2. 个人机操作系统

DOS：命令行、批处理程序、int 21h

Windows、Mac OS、Linux、Unix

1. 操作系统的基本特性
2. 并发性

\*定义 ：两个或两个以上的事件在同一时间间隔内发生，指操作系统应该具有处理和调度多个程序同时执行的能力

\*区别：并发与并行，进程与线程

1. 共享性

\*定义：系统中的资源可供内存中多个并发执行的进程共同使用

\*共享方式：

互斥共享方式：一段时间内只允许一个进程访问该资源（打印机）

同时访问方式：允许在一段时间内由多个进程“同时”对它们进行访问（磁盘、文件）

\*\*并发与共享是基本特征，互相依赖、互为存在

1. 异步性

\*定义：各任务执行任务不可预知

1. 虚拟性

\*定义：把物理上的一个实体变成逻辑上的多个对应物，或相反

\*分类：时分、空分

1. 操作系统的主要功能
2. 主要任务

为多道程序提供良好的运行环境，保证程序高效运行，最大程度利用系统资源，方便用户使用

2、主要功能

（1）处理机管理功能：进程控制、进程同步、进程通信、调度

（2）存储器管理功能：内存分配 、内存保护、地址映射、内存扩充

（3）设备管理功能：缓冲、设备分配、设备处理

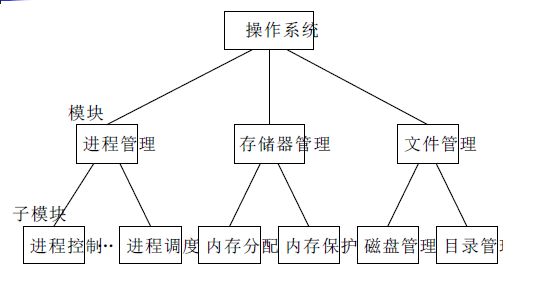
（4）文件管理功能：文件存储空间的管理、目录管理、文件的读/写管理

和保护

1. 用户接口：命令接口 、程序接口
2. 新功能：安全、网络功能、支持多媒体
3. 操作系统的结构设计

\*\*设计原则与目标：有效、系统的高效率、资源的高利用率、公平合理、易于维护和使用升级

1. 无结构操作系统
2. 模块化OS结构



\*优点：提高了OS设计的正确性、可理解性和可维护性

增强了OS的可适应性

加速了OS的开发过程

\*缺点：对模块的划分和对接口的规定不精确，存在错误

模块间存在着复杂的依赖关系，OS结构不清晰

1. 分层式OS结构

\*特性：每一步设计都建立在可靠的基础上

\*优点：易保证系统正确性，易扩充和易维护

\*缺点：系统效率会降低

1. 客户、服务器模式

\*特征：

1. 客户：具有一定的处理能力、向服务器请求服务
2. 服务器：为客户提供服务
3. 通信、网络
4. 交互：客户机发送请求消息 —> 服务器接受信息 —> 服务器回送消息 —> 客户机接收消息

\*优点：数据分布存储和处理

便于集中管理

灵活可扩充

\*缺点：可靠性，瓶颈

1. 面向对象技术

\*优点：可重用性好、易于修改和扩展、易于保证系统正确性和可靠性

1. 微内核技术

\*基本思想

1. 内核足够小：实现核心基本功能、硬件紧密相关、客开与服务器通信
2. 基于用户/服务器模式：基本功能放入内核、大部分功能放在微内核外的服务器实现（进程服务器、存储服务器、设备管理服务器），通过微内核提供的消息传递机制实现信息交互
3. 机制与策略分离
4. 采用面向对象技术

\*基本功能

微内核：进程调度

（1）进程管理

进程服务器：策略调度

微内核：地址映射

（2）低级存储器管理

存储服务器：虚拟存储管理

微内核：响应中断

（3）中断和陷入服务

进程服务器：处理中断

\*优点：正确性、灵活性、易于维护、可扩充

1. 提高系统可扩展性
2. 增强系统可靠性
3. 增强可移植性
4. 支持分布式
5. 采用面向对象技术

\*缺点

1. 效率降低

\*传统OS：执行程序调用（用户态进入系统态）—> 系统完成用户的请求（系统态进入用户态）

\*微内核OS：客户给内核发消息请求服务器服务——> 内核把请求发送给服务器 ——> 服务器完成请求发消息给内核 ——> 内核将消息发给客户

1. 服务器功能转让内核（提高了设计代价）
2. 操作系统的硬件环境（操作系统直接依赖于硬件）

\* OS需要的硬件特性：受保护的指令、系统调用、内存保护、中断机制、I/O系统、时钟操作

1. 受保护的指令（特权指令）

\*处理器的状态（根据运行程序对资源和机器指令的使用权限，把虚拟机设置为不同状态）

1. 管态（系统态、内核态）：操作系统的管理程序运行时的状态，特权级别较高，可移植性所有指令，使用所有资源，并具有改变处理器状态的能力。
2. 目态（用户态）：用户程序运行时的状态，特权级别较低，禁止使用特权指令，不能直接使用系统资源与改变CPU状态，只能访问用户程序所在的存储空间

\*特权级别划分：从R0到R3特权能力依次降低

1. R0：管态，运行操作系统核心代码
2. R3：目态，运行各种用户程序
3. R1：运行关键设备驱动程序和I/O处理例程
4. R2：运行其他受保护共享代码

\*各级别指令集关系：图片 2

\*CPU如何判断当前运行程序是系统程序还是用户程序

答：使用程序状态字PSW，用来做示处理器的状态，包括CPU的工作状态码（指明管态 or 目态）、条件码（指令执行后的结果特征）、中断屏蔽码（是否允许中断）

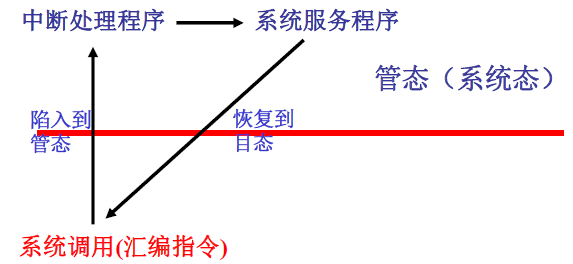
\*状态之间如何转换？

1. 管态—>目态：修改PSW实现
2. 目态—>管态：通过系统调用

2、系统调用

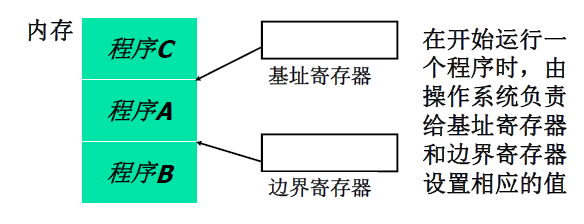
\*定义：通过特殊的访管指令，来请求操作系统为其提供某种功能的服务

\*实现过程：CPU执行访管指令（引起访管中断）—> 处理机保存中断点的程序执行上下文环境（CPU切换到管态）—> 中断处理程序工作，调用相应系统服务—> 中断处理结束，恢复中断上下文环境（CPU恢复到目态，回到中断点继续执行）



1. 内存保护

\*作用：防止一个用户程序去访问其他用户程序数据；保护操作系统免受用户程序的破坏 \*硬件支持：基址寄存器和边界寄存器（只需一对）



\*虚拟存储技术：把内存和外存结合起来使用，硬件提供虚、实地址映射机制

1. 中断机制

\*定义：由于某个事件的发生，改变了正在CPU上执行的指令的顺序，这种事件对应于CPU芯片内部或外部的硬件电路所产生的电信号

\*中断处理过程：CPU暂停正在执行的程序—>保留现场，转去执行相应事件的处理程序——> 处理完成返回断点，继续执行被打断程序

\*中断类型

1. 同步中断（异常）：CPU控制单元发出的中断

种类：CPU检测到的异常（溢出等）、程序设定的异常（int指令，软中断）

1. 异步中断（中断）：由其他的硬件设备在任意时刻发出的中断

种类：可屏蔽中断（I/O中断，例：打印机打印完成）、不可屏蔽中断（掉电、硬件故障）

\*\*系统根据中断向量来为每一个中断或异常指定相应的处理程序

1. I/O系统

功能：完成计算机系统中信息的输入输出功能

1. 时钟

功能：（1）分时系统：进程按时间片轮转

（2）实时系统：按要求的间隔输出正确的时间信号给实时的控制设备

（3）记录用户和系统所需的绝对时间

1. 进程的描述与控制
2. 进程的相关概念
3. 程序：指令或语句序列

\* 执行方式：顺序执行和并发执行

1. 顺序执行：

①定义：程序各部分依次顺序执行，当前操作完成后才能执行后继操作

②特性：顺序性、封闭性（独占全集资源，执行结果不受外界影响）、可再现性

1. 并发执行

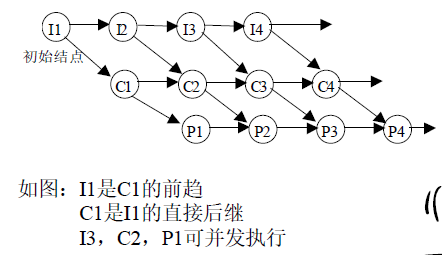
①定义：一组逻辑上相互独立的程序在执行过程中，执行时间在客观上相互重叠。

②区分：并发、并行

③引入目的：提高资源利用率，从而提高系统效率

④表示：前趋图

例：



⑤特征：间断性（共享资源间的制约，合作完成任务时的制约）、失去封闭性、不可再现性（并发执行在任何地方都可以中断）

1. 前驱图

定义：是有向无环图，描述进程之间执行的前后关系。结点表示一个程序段、进程或语句，有向边表示两个结点之间的前驱关系。（Pi—>Pj： Pi是Pj的直接前趋，Pj是Pi的直接后继）

初始结点：没有前趋的结点

终止结点：没有后继的结点

3、进程

1. 概念：进程是一个具有独立功能的程序对某数据集在处理机上的执行过程，也是操作系统进行资源分配的基本单位
2. 特征：结构性（组成：程序、数据集和进程控制块）、动态性、并发性（多进程实体同存在与内存中，在一段时间内同时执行，执行过程中可能被打断）、独立性（进程是资源分配的基本单元也是独立运行的基本单位）、异步性（各进程按异步方式运行）

4、程序与进程

1. 程序： 顺序性、静态性、孤立性

无法描述执行过程的随机性、并发性

无法反映资源共享

1. 进程： 动态性、共享性

是描述程序的动态执行过程和使用共享资源的基本单位

1. 区别：

① 程序是静态的，进程是动态的

② 程序是有序代码的集合，进程是代码的执行

③ 进程有生命周期，程序相对可以长久保存

④ 进程可以描述并发

⑤ 组成不同

⑥ 一个程序可以对应多个进程

⑦ 进程是竞争计算机系统资源的基本单位

1. 进程状态及转换
2. 三种基本状态

\* 运行态：进程占有CPU，并在CPU上运行

\* 就绪态：一个进程已经具备运行条件，但由于无CPU暂时不能运行（有了CPU立即可以运行）

\* 等待态（阻塞、睡眠）：进程因等待某种事情的发生而暂时不能运行的状态（即使CPU空闲，该进程也不可运行）

1. 状态转换

①就绪—>运行：调度程序选择一个新的进程运行

②运行—>就绪：运行进程用完了时间片/运行进程被中断，因为一高优先级进程处于就绪状态

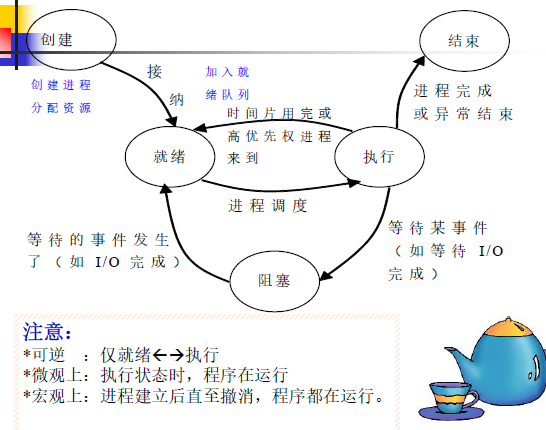
③运行—>等待：一进程所需事件未发生/os尚未完成服务/对一资源的访问尚不能进行/等待IO结果/等待某一进程提供输入

④等待—>就绪：当所等待的事件发生

1. 其他状态

① 创建状态（new）：创建PCB，填写必要信息，将进程转入就绪队列，创建完成后才可参与调度

② 中止状态（exit）：进程结束运行，回收除PCB之外的其他资源，并让其他进程从PCB中收集有关信息。数据不再需要后，进程被删除。



（4）挂起进程模型（静止状态）

①引入目的：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

②新增进程状态：活动就绪、静止就绪、活动阻塞（挂起）、静止阻塞（挂起）

③新增状态转换：

\*活动就绪——>挂起——>静止就绪：优先级高的就绪进程竞争时，系统会挂起优先级低的就绪进程

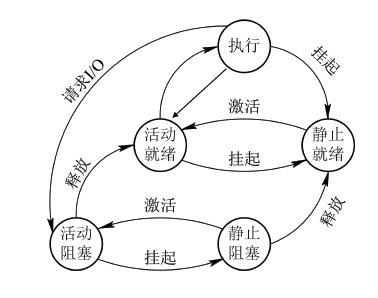
\*活动阻塞——>挂起——>禁止阻塞：就绪进程要求更多内存资源

\*静止阻塞——>静止就绪：等待的事件发生时

\*静止就绪——>激活——>活动就绪

\*静止阻塞——>激活——>活动阻塞

\*运行——>静止就绪：高优先级抢占CPU且内存不足，自我挂起



1. 进程管理中的数据结构

（1）进程的组成：有关程序段、数据集、进程控制块（PCB）

\*进程控制块：管理进程的数据结构，记录进程的相关信息，描述进程运动变化过程，识别和控制进程，包含了进程的描述信息、控制信息和资源信息。PCB是系统感知进程存在的唯一标识，进程与PCB一一对应，处于核心段，访问需要经过系统调用。

\*PCB组成：进程标志符、处理机状态、进程调度信息、进程控制信息

\*PCB组织方式：

PCB表——把所有PCB组织在一起，并把它们放在内存的指定位置，大小决定了系统中最多可同时存在的进程的个数，称为系统并发度。

进程队列——同一状态的所有PCB链接在一起

链接方式

索引方式

二、进程控制

1、目的：处理机管理，协调执行并发程序，实现资源共享

2、内容：创建、撤销进程，完成进程各状态之间的转换

（1）进程创建

\*形式：进程图——描述进程家族关系的有向树，父进程指向子进程，子进程可以继承父进程的资源，子进程撤销时应归还父进程资源，撤销父进程要求撤销子进程

\*引发事件：用户登录、作业调度、提供服务、应用请求（系统进程或父进程可以创建新进程）、创建原语(申请PCB，赋予一个统一进程标识符—>分配资源—>初始化进程控制块—>设置相应链接

\*创建进程的方法：在一个已经存在的进程当中，通过系统调用来创建一个新进程

（2）进程中止

\*引发事件：进程运行结束、异常结束、外界干预

\*进程中止过程：

（3）进程的阻塞和唤醒

\*引发事件：请求系统服务未果、启动某操作，操作未完成前、新数据尚未到达、无新工作可做

\*阻塞过程：调用阻塞原语阻塞自己（停止执行—>修改PCB状态—>插入阻塞队列—>调度程序重新调度—>切换处理机）

\*唤醒过程：移出阻塞队列—>修改PCB标志—>插入就绪队列

（4）挂起与激活

\*引起挂起事件：用户进程挂起自己，父进程挂起子进程，系统挂起指定进程

过程：检查被挂起进程状态（活动就绪—>静止就绪、活动阻塞—>静止阻塞），拷贝PCB到指定内存区域

\*引起激活事件：父进程或用户进程激活指定进程且内存空间足够

过程：外存调入内存，检查进程状态（静止就绪—>活动就绪、静止阻塞—>活动阻塞），检查优先级

3、手段：进程采用原语的形式控制程序段

\*原语：系统态下运行，完成系统特定功能的过程

\*原子操作过程：一个整体不可分割执行

\*机器指令级：执行期间不可中断

\*功能级：作为原语的程序段不允许并发执行

三、进程同步

1、基本概念

（1）间接相互制约关系：有共享资源

（2）直接相互制约关系：进程间合作

（3）临界资源：一次只允许一个进程使用的资源（互斥方式使用）

\*\* 生产者-消费者问题：分别执行正确，并发执行错误，必须对共享资源加以管理

（4）临界区：进程中访问临界资源的那段代码。进程互斥的进入临界区去访问临界资源。进入之前检查临界资源是否被访问，退出临界区把临界资源的访问标志修改为未访问。

（5）同步机制遵循的规则：空闲让进、忙则等待、有限等待、让权等待

2、硬件同步机制

（1）基于关闭中断的互斥实现

\*定义：当一个进程进入临界区后，关闭所有中断；退出临界区再打开中断

\*特点：关闭中断后，CPU不会被分配给其他进程，进程的切换是由中断引发的，这样其他进程获得不了资源也就无法运行；操作系统内核常使用该方法更新内部数据结构。

（2）基于繁忙等待的互斥实现

\*缺点：浪费CPU事件、可能导致预料之外的结果（高优先级分配了CPU，但无法进入临界区，低优先级无法获得CPU，造成死锁）

方法一：加锁标志位法

\*当lock值为0时才可进入临界区，若为1说明有进程在临界区内，需循环等待。

方法二：TSL指令

\*把查询lock变量与修改lock变量捆绑在一起，使它们不会被打断。TSL是一个原子操作

\*使用TSL指令实现进程间互斥：使用lock作为加锁标志位，协调各个进程对共享资源的访问。当lock为0时，任何进程均可使用TSL指令把它设置为非0，进而访问共享资源，当lock为非0时，循环等待，退出临界区时，把lock置为0.

方法三：SWAP指令

\*当mutex为0时，任何进程均使用swap指令把它设置为非0，进而访问共享资源；mutex非0时，等待。退出临界区时，把mutex置0.

3、信号量机制 1965 Dijkstra

（1）整型信号量

\*通过两个标准的原子操作 wait(S)、signal(S)来访问（P、V操作），该机制使进程处于忙等状态。 wait(S){ while S≤0 do no-op；S∶=S-1;} signal(S){ S∶=S+1; }

（2）记录型信号量

\*基本思想：一个整型变量代表资源数目，进程链表链接所有等待进程，采用了记录型的数据结构。wait操作请求一个单位的该类资源，成功则资源数减1，资源数小于0说明无资源可用，自我阻塞，插入信号量链表。signal释放一个资源，资源数加1，等待该资源的进程被唤醒。

typedef struct {

value:integer;

struct PCB \*list;

}semaphore

wait( semaphore S){

S.value--;

if (S.value＜0) block(S->list)

}

signal(semaphore S){

S.value++;

if (S.value≤0)wakeup(S.->list);

}

（3）AND型信号量

\*一次性分配原则：将进程在整个运行过程中需要的所有资源，一次性全部分配给进程，进程使用完后再一起释放。对临界资源的分配采用原子操作的方式：要么全部分配给进程，要么一个也不分配。

Swait(S1, S2, …, Sn){

While(TRUE){

if (Si≥1 && … && Sn≥1){

for (i =1;i<=n;i++) si--;

break;

} else{

block(Si🡪list);

// where Si is the first

place the process in the waiting queue associated with the first Si found with Si＜1, and set the program counter 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）信号量集

\*基本思想：一次分配多个某类资源，每次分配前测试该资源的数量是否大于下限值（S：信号量，d：需求量，t：下限量）

Swait(S1, t1, d1, …, Sn, tn, dn)

signal(S1, d1, …, Sn, dn)

\*特殊情况：

1）Swait(S,d,d)：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，现有资源少于d时不予分配。

2）Swait(S,1,1)：S>1时退化为记录型信号量

S=1时退化为互斥信号量

3）Swait(S,1,0)：S=0后，阻止任何进程进入特定区，相当于一个可控开关。

4、信号量的应用

（1）实现互斥：semaphore mutex =1;

（2）实现前驱关系：语句S1和S2分属不同进程，若希望执行S1后再执行S2，则可使二者共享公用信号量s。

进程P1： S1; signal(s);

进程P2： wait(s); S2;

示例：

semaphore a,b,c,d,e,f,g;

P1(){ S1; signal(a); signal(b); }

P2(){ wait(a); S2; signal(c); signal(d); }

P3(){ wait(b); S3; signal(e); }

P4(){ wait(c); S4; signal(f); }

P5(){ wait(d); S5; signal(g); }

P6(){ wait(e); wait(f); wait(g); S6; }

main(){

a.value=b.value=c.value=d.value=e.value=f.value=g.value=0;

cobegin

p2();p2();p3();p4();p5();p6();

coend

}

5、管程机制

（1）定义：代表共享资源的数据结构，对该数据结构实施操作的一组过程，共同构成操作系统的资源管理模块

（2）作用：请求和释放资源通过管程完成

（3）组成：名称、内部数据结构、过程、初始化语句

（4）条件变量：

* + **cwait**，**csignal**原语同步
  + 显式说明条件变量：形如 **condition x, y;**
  + 只能由管程访问
  + 抽象数据类型，由链表记录因该条件变量而阻塞的进程
  + **cwait(x):** 调用管程的进程因**x**而阻塞，则调用该过程讲其插入到**x**条件的等待队列。释放管程。
  + **csignal(x):** 调用管程的进程发现**x**条件变化，调用该过程重新启动阻塞进程。

1. 经典进程同步问题
2. 生产者-消费者问题

\*并发进程互斥和同步的一般模型

①问题描述

\*消费者：使用某一资源的进程

\*生产者：释放某一类资源

\*两者满足的条件：通过有n个缓冲区的公共缓冲池交换信息，消费者想接受信息，则缓冲区至少有一个单元是满的；生产者想发送数据则缓冲区中至少一个单元是空的。

②解决方法

\*记录型信号量

semaphore mutex=1//临界区占用情况，1时表示空闲, empty=n//空闲的资源数, full=0//被占用的资源数;

item buffer[n];

int in=0, out=0;

producer(){ //生产者

do{

…

producer an item nextp;

…

wait(empty);//empty-1

wait(mutex);//判断是否有其他进程占用临界区，没有的话mutex置0，程序继续运行。（当n=1时可省略）

buffer(in)=nextp;//生产

in =(in+1) mod n;

signal(mutex);//出临界区

signal(full);//full+1

} while(true);

}

consumer(){

do{

wait(full);//full-1

wait(mutex);（当n=1时可省略）

nextc=buffer(out);//消费

out =(out+1) mod n;

signal(mutex);

signal(empty);//empty+1

consumer the item in nextc;

}while(true);

}

main(){

cobegin

producer(); consumer();

coend

}

\*\*注意：wait和signal成对出现，次序不能颠倒，若wait(empty)和wait(mutex)次序颠倒，当占用临界区后，发现缓冲池已满，则无法生产，消费者获得不了临界区资源，无法消费，最终造成死锁。

\*利用AND信号量

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

item buffer[n];

int in=0, out=0;

producer(){

do{

produce an item in nextp;

…

Swait(empty, mutex);//资源数目和临界区使用情况同时设置，避免中途中断

buffer(in) =nextp;

in =(in+1)mod n;

Ssignal(mutex, full);

} while(true);

}

consumer(){

do{ 

Swait(full, mutex);

nextc =buffer(out);

out =(out+1) mod n;

Ssignal(mutex, empty);

consumer the item in nextc;

} while(true);

}

\*利用管程解决

monitor producer-consumer{

int in=0,out=0,count=0;

item buffer[N];

condition notfull, notempty;}

void put(item x){

if count≥N then cwait(notfull);

buffer[in] = x;

in=(in+1) % N;

count++;

csignal(notempty);

}

void get(item x){

if (count≤0)cwait(notempty);

x =buffer[out];

out =(out+1) % N;

count--;

csignal(notfull);

}

}

Producer(){

item x;

while(1){

produce an item in x;

PC.put(x);

}

}

Consumer(){

item x

while(1){

PC.get(x);

consume the item in x;

}

}

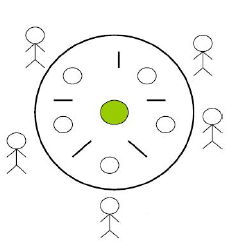
1）put(item)过程：生产者将生产的产品投放到缓冲池中，当count>=n时表示缓冲池已满，生产者需等待。

2）get(item)过程：消费者从缓冲池取出一个产品，count<=0，无产品可取用，消费者等待。

1. 哲学家进餐问题

①问题描述

欲吃面，每个哲学家必须获得两把叉子，且每人只能直接从自己左边或右边去取叉子。



②解决方法

\*每把叉子都要互斥使用，应把每把叉子设置成一个互斥的信号量，初值均为1（空闲）。

\*隐患：死锁

\*相关思路：

1）只允许4个人同时取左边筷子，仅当左右叉子都可用时才进餐（设置信号量）

2）规定奇数编号哲学家先左后右；偶数编号相反。

3）1、 2号哲学家竞争1号筷子；3、4号哲学家竞争3号筷子。即五位哲学家都先竞争奇数号筷子，获得后，再去竞争偶数号筷子，最后总会有一位哲学家能获得两只筷子而进餐。

\*记录型信号量解决（初始先判断人数信号量）

semaphore chopstick[5] ={1,1,1,1,1};

do{

Wait(people);//people--;

wait(chopstick［i］);

wait(chopstick［(i+1) % 5］);

eat;

signal(chopstick［i］);

signal(chopstick［(i+1) % 5］);

think;

}while(true);

\*AND信号量解决（要求每个哲学家先获得两个临界资源后方能进餐）

semaphore chopstick[5] ={1,1,1,1,1};

do{

think;

Sswait(chopstick［(i+1) % 5］, chopstick ［i］);

eat;

Ssignal(chopstick ［(i+1) % 5］, chopstick ［i］);

}while(true);

1. 读者-写者问题（读者优先）

①问题描述

允许多个进程同时读共享数据，读写不能同时，写写不能同时

②解决方案

\*读写进程互斥（设置信号量）

\*记录读进程数目，无读进程才可写（设置全局变量）

\*记录型信号量解决

semaphore rmutex=1, wmutex=1;

int readcount =0;

Void reader(){

do{

wait(rmutex);//rmutex-1

if (readcount=0) wait(wmutex);//wmutex-1，第一个读进程，设置不能再写

readcount++;//为该互斥资源设立临界区（wait、signal限制）

signal(rmutex);//rmutex+1

perform read operation;

wait(rmutex);//rmutex-1

readcount--;

if (readcount=0) signal(wmutex);//wmutex+1，最后一个读进程，设置可以写

signal(rmutex);}

while(true);

}

void writer(){

do{

wait(wmutex);

perform write operation;

signal(wmutex);}

while(true);

}

\*信号量集机制解决（最多允许N个读者同时读）

Swait(s,t,d);//s（资源总量）>=t时，-d（进程需要的资源数）

#define N 88

semaphore L=N, mx=1(mx=1可读，mx=0可写）;

void reader()

do{

swait(L,1,1);//控制读者数目，每次有一个读者进入，L-1，当L减为0后在此阻塞

swait(mx,1,0);//无写进程可读，起开关作用，无writer进入写操作，mx=1，可以进入读操作

perform read operation;

ssignal(L,1);

}while(true);

}

void writer(){

do{

swait(mx,1,1; L,N,0);//起开关作用

//无写进程mx=1，

//无读进程，即L＝N；

perform write operation;

ssignal(mx,1);

}while(true)

}

1. 进程通信
2. 进程间交换的信息

①控制信息的传送：低级通信，传递状态或值，由程序员实现，目的是控制执行速度

②大批量数据的传送：高级通信，目的是交换信息，隐藏进程通信细节，降低编程复杂度

1. 进程通信的类型

①共享存储器系统

\*基于共享数据结构的通信方式：程序员负责同步。效率低

\*基于共享存储区的通信方式：高级通信，向系统申请共享存储器的一个分区，对此分区进行读写

②消息传递系统

\*以格式化的消息为单位进行消息交换

\*程序员直接利用系统提供的一组通信命令（原语）进行通信

\*高级通信，操作系统隐藏了通信的实现细节，简化了编程复杂度

③管道通信

\*读写进程通过称为管道的共享文件进行通信

\*管道机制能够协调进程间的互斥、同步，并能确定对方的存在

1. 消息传递通信的实现

①直接通信的方式：发送进程利用OS提供的发送命令，直接把消息发送给目标进程

要求发送进程和接收进程都以显式方式提供对方标识符

\*发送消息给接收进程：Send(Receiver,message);

\*接收Sender发来的消息：Receive(Sender,message);

\*若接收进程可与多个发送进程通信，无法事先指定发送进程可设置成接收任何进程的消息

②间接通信方式：通信双方共享数据结构（邮箱）暂存消息

发送进程发消息到邮箱，接收进程从邮箱取消息

\*系统提供原语：信箱的创建和撤销、消息的发送和接收

\*信箱类型：私用信箱——进程的一部分

公用信箱——操作系统创建，供核准的进程使用

共享信箱——创建者指明共享的进程的名字

\*发送进程和接收进城的关系：1:n，n:1,1:1，n:n

1. 消息传递系统实现中的若干问题

①建立通信链路

\*计算机网络中：发送进程在通信之前显式建立，使用完成后显式拆除

\*单机系统中：发送进程无需明确提出建立链路的请求，利用系统提供的发送命令系统会自动为之建立

②消息的格式：变长、定长

③进程同步方式

\*发送进程阻碍，接收进程阻碍：用于无缓冲时

\*发送进程不阻碍，接收进程阻碍：用于有缓冲时，广泛使用

\*均不阻塞

1. 线程

\*进程的两个基本属性：资源分配的基本单位、执行过程的基本单位

\*缺点：创建进程——撤销进程——进程切换过程中涉及到资源的管理和分配

处理机调度时系统时间空间开销大，并发度有限

1. 线程的基本概念

\*引入线程的目的：减少处理机空转时间和调度切换时间

提高系统执行效率

适应对称多处理机

\*线程与进程比较

①进程拥有若干个线程（至少一个）

②线程仅作为调度的基本单位，同一进程内线程切换不会引起进程切换，减少了切换开销，提高了速度

③线程具有并发性

④线程不拥有资源，访问所属进程的资源；进程的资源对其所属的所有线程是共享的

⑤线程系统开销少，切换仅涉及寄存器，不涉及存储器，同不通信容易

\*线程的属性

①独立轻型实体：不拥有系统资源，只有很少的必要资源（TCB、程序计数器、寄存器和堆栈）

②调度和分派的基本单位

③可并发执行

④共享进程资源

\*线程的状态

①状态参数：寄存器状态、堆栈、线程运行状态、优先级、线程专用存储器、信号屏蔽

②线程运行状态：执行状态（线程获得处理机正在运行）

就绪状态（线程具备执行条件，一旦获得CPU便可执行）

阻塞状态（执行时受阻）

\*线程的创建和终止

①创建：程序启动时有一个初始化线程在执行。利用线程创建函数创建，提供参数：指向线程主程序的入口指针、堆栈大小、调度优先级。执行完成返回线程标识符

②终止：线程完成工作后退出，线程运行出错或被其他线程终止

\*多线程OS中的进程

①系统资源分配的单位

②可包括多个线程

③进程不再是一个可执行的实体（进程为独立运行单位）

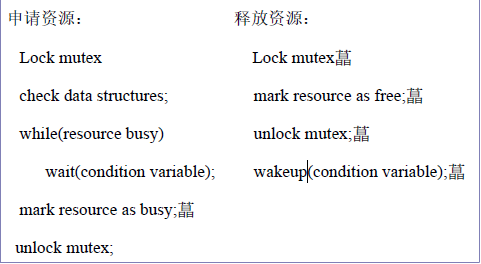
1. 线程的同步和通信

\*互斥锁：实现对资源互斥访问的机制

优点：时间空间开销低、适合高频度使用的共享资源

缺点：可能引发死锁

\*条件变量：创建一个互斥锁时便联系着一个条件变量，用于线程的长期等待，直至资源可用



\*信号量机制

①私用信号量：实现同一进程中各线程之间的同步，数据结构存放在应用程序的地址空间中；属于特定进程所有，OS并不知道私用信号量的存在

②公用（系统）信号量：实现不同进程间或不同进程各线程间的同步，有公开的名字供所有进程使用；数据结构存储在受保护的系统存储区中，OS分配空间并管理，是一种比较安全的同步机制

1. 线程的实现方式

①内核支持线程：线程的创建、撤销、切换依靠内核

②用户级线程：无需系统调用实现，无需内核支持，进程切换速度快

\*优点：减少管理开销，各进程可选择适合自己的线程调度方法，与操作系统平台无关

\*缺点：一个线程进行系统调用时阻塞，则其所属进程即阻塞，进程内所有线程均阻塞；内核每次分配给进程一个CPU，进程中只有一个线程执行，效率低

③组合方式

1. 线程的实现

①内核支持线程的实现：创建进程时分配任务数据区——>创建线程时为线程分配TCB——>撤销进程回收TCB

②用户级线程的实现

1. 用户级线程与内核控制线程的关系

\*一对一：并性能力强，系统开销大

\*多对一：管理开销小，线程阻塞会引起进程阻塞

\*多对多：根据情况变化

1. 处理机调度与死锁
2. 处理机调度的层次和目标

\*多道批处理系统（进过如下两个过程才能获得处理机）

①作业调度（高级调度、长程调度）

②进程调度（低级调度、短程调度）

\*终端型作业：进程调度

\*中度调度：提高内存利用率

1. 处理机调度的层次

\*作业调度：宏观、高级，调入后备作业进入内存，建立进程，结束时回收资源。

\*交换调度：中级，进程在内存和外存的交换

\*进程调度：微观、低级，选取进程占用处理机，进程上下文切换

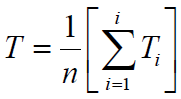
\*线程调度

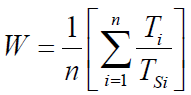
1. 调度算法目标

\*面向系统的准则：处理机利用率好、公平、各类资源平衡利用、强制执行机制

①批处理系统

1. 周转时间短

\*\*平均周转时间：

\*\*带权周转时间：

\*\*Ti:作业提交到作业完成

\*\*Tsi：作业在外存后备队列等待作业调度的时间＋在就绪队列等待进程调度的时间＋CPU执行时间＋等待I/O操作完成时间

（2）系统吞吐量高

（3）处理机利用率高

②分时系统

1. 响应时间快（输入时间+处理时间+响应时间）
2. 均衡性

③实时系统

1. 截止时间的保证（开始截止时间、完成截止时间）
2. 作业与作业调度
3. 作业

\*一个作业是指在一次应用计算或者事务处理过程中，从输入开始到输出结束，用户要求计算机所做的有关该次任务处理的全部工作的总称。

1. 作业步

\*作业由不同的顺序相连的作业步组成。

作业步是在一个作业的处理过程中，计算机所做的相对独立的工作。(编译, 链接,运行)

3、作业控制块 Job Control Block

作业在系统中存在的标志，保存系统对作业进行管理和调度所需信息

作业标志，用户名称，作业类型，状态，调度信息，资源需求，时间信息等。

4、作业运行三个阶段与三种状态

收容阶段 后备状态

运行阶段 运行状态

完成阶段 完成状态

1. 作业调度

（1）基本任务：根据作业控制块的内容，检查系统是否满足作业的资源要求，并按一定算法选取作业调入内存，为其创建进程，分配必要的资源。将新创建的进程插入就绪队列，准备运行。

（2）目标：用户—周转时间短，系统—平均周转时间短，系统吞吐量高

（3）中心问题：接纳多少作业（周转时间、利用率），接纳哪些作业

（4）作业调度的位置



1. 调度算法

①先来先服务（FCFS）

\*应用于作业调度or进程调度

\*有利于长作业，不利于短作业

②短作业优先（SJF）

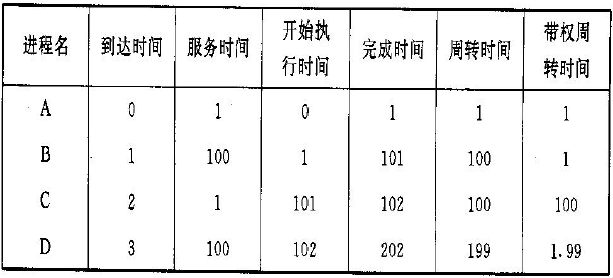
\*降低作业的平均等待时间，提高系统吞吐量

\*长作业长期得不到执行

\*未考虑作业紧迫程度

\*未估计运行时间，可能有用户作弊（自己实际的作业时间与提交的不符）

##例题：对于SJF来说，运行顺序仍为A、B、C、D（不允许中断的情况下，B先于C到达，此时C不可见）





③高优先权优先调度算法

\*算法分类：非抢占式优先权算法（批处理系统）、抢占式优先权算法（实时分时系统）

\*优先权分类：静态优先权（创建进程时确立优先级）、动态优先权（随进程等待时间增加优先权增大、随进程运行时间增加优先权减少）

④高响应比优先调度算法



(1) 如果作业的等待时间相同，则要求服务的时间愈短，其优先权愈高，因而该算法有利于短作业。

(2) 当要求服务的时间相同时，作业的优先权决定于其等待时间，等待时间愈长，其优先权愈高，因而它实现的是先来先服务。

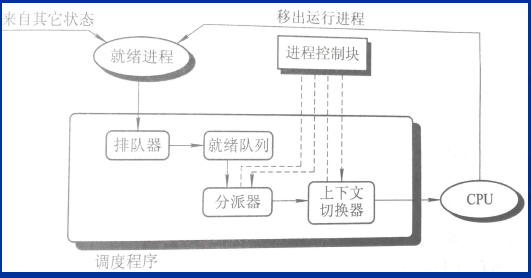
(3) 对于长作业，作业的优先级可以随等待时间的增加而提高，当其等待时间足够长时，其优先级便可升到很高， 从而也可获得处理机。

1. 进程调度（短程、低级）
2. 功能

\*选取就绪队列中的进程并分配处理机

1. 进程调度中的三个基本机制

\*排队器、分排器（调度器）、上下文切换



1. 调度方式

\*非抢占式：简单，无法满足紧急任务

\*抢占式：公平，满足紧急进程

1. 算法

①基于时间片的轮转调度算法

\*简单时间轮转法

1. 基本原理：进程排成队列、先来先服务。时间片用完时，计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，送往就绪队列的末尾；再把处理机分配给就绪队列中新的队首进程。

（2）时间片大小确定：短时间片利于短作业，切换进程系统开销大；长时间片，响应时间长，可能退化为FCFS，一般取时间片略大于要求的交互时间。

\*多级反馈队列调度算法

（1）基本原理：设置多个就绪队列；依次为各个队列赋予不同的优先级；优先权愈高的队列中，为每个进程所规定的执行时间片就愈短。新进程进入内存，首先放入第一队列的末尾，按FCFS原则待调度。轮到该进程执行时，如能在一个时间片内完成，便可准备撤离系统；如果未能完成，便将该进程转入第二队列的末尾，再同样地按FCFS原则等待调度执行, 依此类推;当一个长作业(进程)从第一队列依次降到第n队列后，在第n队列中便采取按时间片轮转的方式运行。仅当第一队列空闲时，调度程序才调度第二队列中的进程运行； 仅当第1~(i-1) 队列均空时，才会调度第i队列中的进程运行。如果处理机正在第i队列中为某进程服务时，又有新进程进入优先权较高的队列(第1~(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第i队列的末尾，把处理机分配给新到的高优先权进程。



1. 性能分析

#终端型作业用户：交互短作业 ，通常可在第一队列完成

#短批处理作业用户：可在前几个队列完成，周转时间较短

#长批处理作业用户：依次在1，2，…. 队列运行，最后到n队列按轮转方式运行

②基于公平原则的调度算法

1. 基本原理

\*调度公平：实际占用处理器时间与公平分配占用处理器期望时间的比率作为参数；比率最低的优先调度，达到公平比率；

\*分享公平：以用户为单位进行公平调度，避免多进程用户过多占用处理机

③多队列调度

1. 基本原理

\*多个就绪队列

\*不同类型和性质的进程分配在不同队列中

\*每个处理机不同就绪队列

\*不同队列可以不同优先级和调度原则

1. 实时调度
2. 实现实时调度的基本条件

\*提供必要的信息（就绪时间、开始截止时间、完成截止时间、处理时间、资源要求、优先级）

\*系统处理能力强

\*采用抢占样式调度机制

\*具有快速切换机制

1. 实时调度算法分类

\*根据任务性质：硬实时调度、软实时调度

\*根据调度方式：非抢占式、抢占式

①非抢占式调度算法

特点：简单、适用小型系统和不严格实时系统

（1）非抢占式轮转法：实时性要求不强

（2）非抢占式优先调度算法：一般性实时要求

②抢占式调度算法

特点：实时性要求严格

（1）基于时钟中断的抢占式优先调度算法

（2）立即抢占式的优先权调度算法

\*调度时间：静态调度算法、动态调度算法

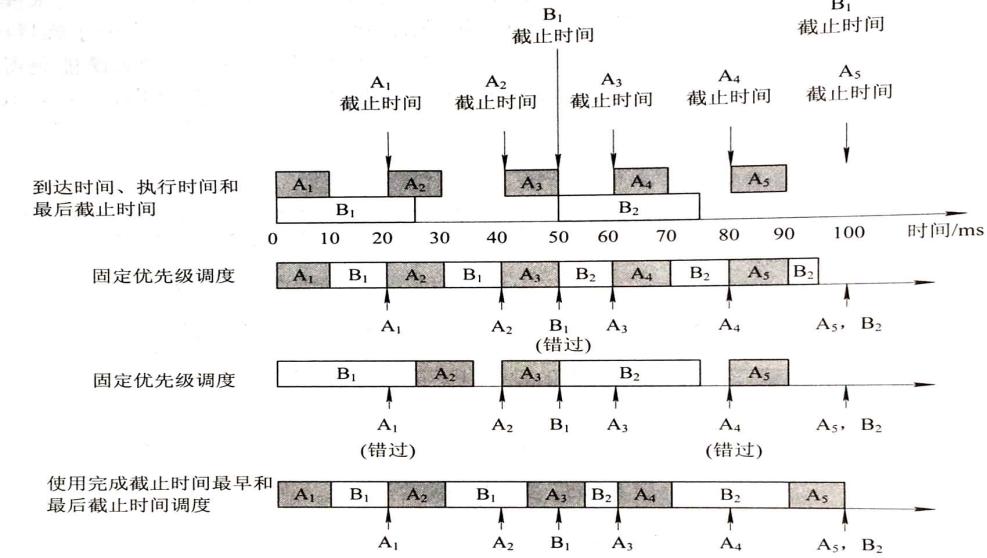


1. 常用实时调度算法

①最早截止时间优先

\*截止时间越早，优先级越高





②最低松弛度优先

\*根据任务的紧急程度确定优先级，主要用于可抢占式调度方式

松弛度的计算方法如下：

任务的松弛度=必须完成的时间-其本身的运行时间-当前时间

其中其本身运行的时间指任务运行结束还需多少时间，如果任务已经运行了一部分，则：

任务松弛度=任务的处理时间-任务已经运行的时间 – 当前时间

几个注意点：

1. 该算法主要用于可抢占调度方式中，当一任务的最低松弛度减为0时，它必须立即抢占CPU，以保证按截止时间的要求完成任务。

2. 计算关键时间点的各进程周期的松弛度，当进程在当前周期截止时间前完成了任务，则在该进程进入下个周期前，无需计算它的松弛度。

3. 当出现多个进程松弛度相同且为最小时，按照“最近最久未调度”的原则进行进程调度。

4、优先级倒置问题（不希望出现的情况）

\*高优先级进程被低优先级进程延迟或阻塞

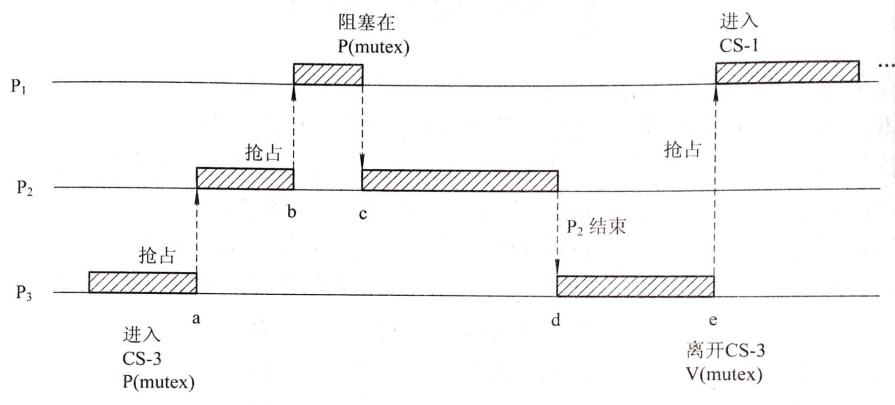
例：

P1 ….P(mutex); CS-1;V(mutex);…

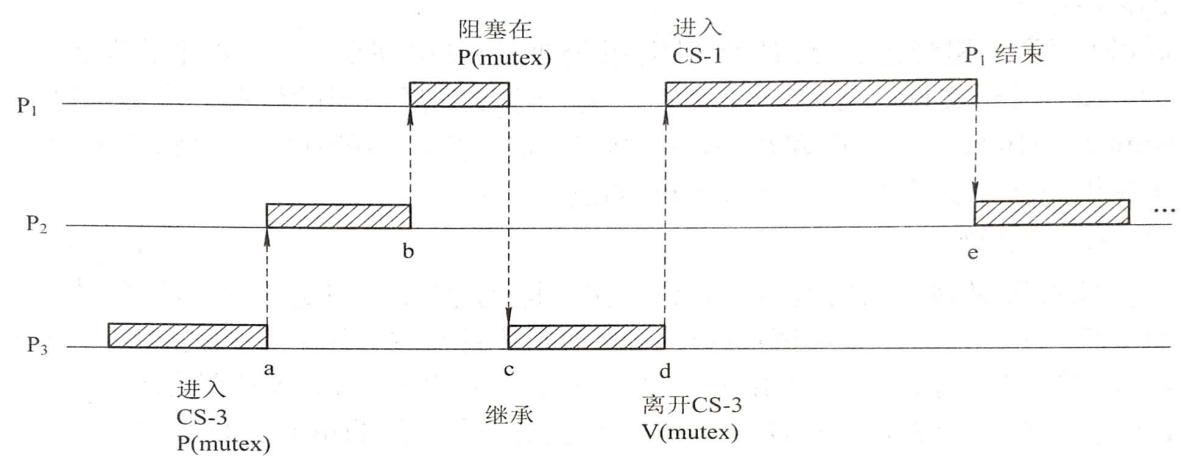
P2 …..;CS-2;…….

P3 …P(mutex);CS-3;V(mutex);…

假定P1, P2, P3优先级依次降低



\*动态优先级继承：高优先级进程P因低优先级进程Q 而阻塞，提升低优先级进程到高优先级进程级别，阻止优先级低于进程P优先级而高于Q优先级的进程抢占Q.



1. 死锁概述
2. 定义

\* 如果一组进程中每一个进程都在等待仅有该组进程中其他进程才能引发的事件，那么该组进程是死锁的。

2、产生死锁的原因

①竞争资源引起进程死锁（）

\*\*资源分类：资源问题：可重用性资源、消耗性资源、可抢占资源、不可抢占资源

1. 竞争非剥夺资源（打印机、磁带机）
2. 竞争临时资源

S1,s2,s3 为消息

P1: ..Release(s1);Request(s3);

P2: ..Release(s2);Request(s1);

P3: ..Release(s3);Request(s2);

safe

P1: .. Request(s3);Release(s1);

P2: .. Request(s1);Release(s2);

P3: .. Request(s2);Release(s3);

Deadlock

1. 并发进程资源竞争（申请--分配--使用--释放）

\* 系统提供的资源数量少于并发进程所要求的资源数量

②进程推进顺序和速度不当

3、死锁后果

\*参与死锁的所有进程都在等待资源

\*浪费大量系统资源，甚至导致系统崩溃

1. 产生死锁的**必要条件**
2. 互斥：资源不能同时被两个或以上进程共享
3. 不剥夺：进程已经获得的资源只能自己释放
4. 部分分配：等待新资源时占有已分配的资源
5. 循环等待：存在进程循环链，链中每一进程已获得的资源同时被下一个进程所请求。
6. 处理死锁的基本方法

\*\*对资源请求分配进行控制

1. 预防

\*定义：设置限制条件，破坏产生死锁的必要条件

\*特点：容易实现，降低资源利用率和系统吞吐量

1. 避免（动态预防）

\*定义：资源分配时，系统预测资源使用情况，防止系统进入不安全状态

\*特点：实现难度大

1. 检测和恢复

\*定义：检测死锁发生的位置和原因，通过外力破坏死锁发生的必要条件

\*特点：允许死锁发生

1. 预防死锁

1、摒弃“请求和保持”条件（打破部分分配条件）

\*定义：静态分配资源，进程运行前预先分配其运行所全部资源。

\*优点：简单，容易实现，安全

\*缺点： 进程资源需求可能无法事先确定；资源很难满足要求

进程生命周期中对资源一直占用，资源浪费

降低了并发性

\*改进: 细分进程任务为子任务, 为子任务请求一次性分配资源

2、摒弃“不可抢占”条件

\*定义：进程提出新的资源请求如不能满足则放弃已有资源

\*缺点：降低系统效率，增长周转时间，对某些类型的资源不适用

3、摒弃“环路等待”条件

\*定义：层次分配资源，资源分类按顺序排列，各进程有序申请资源

\*缺点：限制了进程对资源的请求，可能会降低系统效率；排序需系统开销。

1. 避免死锁

\*\*定义：允许进程动态申请资源，分配之前考察分配的安全性

1. 什么是安全状态

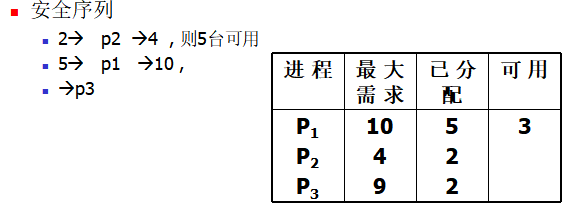
系统能按某种进程顺序(P1, P2, …，Pn)(称〈P1, P2, …, Pn〉序列为安全序列)，来为每个进程Pi分配其所需资源，直至满足每个进程对资源的最大需求，使每个进程都可顺利地完成。如果系统无法找到这样一个安全序列，则称系统处于不安全状态，不安全状态的系统可能会进入死锁。

2、安全状态实例

假定系统中有三个进程P1、 P2和P3；

共有12台磁带机；进程P1总共要求10台磁带机，P2和P3分别要求4台和9台。

假设在T0时刻，进程P1、P2和P3已分别获得5台、2台和2台磁带机，尚有3台空闲未分配。



1. 安全状态转换为不安全状态

\*\*无法找到一个安全序列

例：P3请求1台，为其分配，

此时，在P2完成后只能释放出4台，既不能满足P1尚需5台的要求，也不能满足P3尚需6台的要求，致使它们都无法推进到完成，彼此都在等待对方释放资源，即陷入僵局，结果导致死锁。

故：应该不满足P3的请求。



1. 银行家算法（思想：预请求与分配）
2. 使用的数据结构

假设n个进程，m个资源

\*可利用资源向量Available（Available[j]=K:系统中现有第j类资源K个）

\*最大需求矩阵Max（Max[i,j]=K:进程i需要第j类资源的最大数目为K）

\*分配矩阵Allocation（Allocation[i,j]=k:进程i当前已分得第j类资源的数目为K）

\*需求矩阵Need（Need[i,j]=Max[i,j]-Allocation[i,j]：进程i还需要第j类资源k个方能完成任务）

1. 算法描述

\*设Request是进程Pi的请求向量，如果Requesti［j］=K，表示进程Pi需要K个第j类型的资源。当Pi发出资源请求后，系统按下述步骤进行检查：

①如果Requesti［j］≤Need［i,j］，便转向步骤2；否则认为出错，因为它所需要的资源数已超过它所宣布的最大值。

②如果Requesti［j］≤Available［j］，便转向步骤(3)；否则， 表示尚无足够资源，Pi须等待。

③系统试探着把资源分配给进程Pi，并修改下面数据结构中的数值：

Available［j］∶=Available［j］- Requesti［j］（可用的=可用的-分配的） 

Allocation［i,j］∶=Allocation［i,j］+Requesti［j］;（占有的=占有的+请求的）

Need［i,j］ ∶= Need［i,j］- Requesti［j］ （需要的=需要的-请求的）

④系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态。若安全，才正式将资源分配给进程Pi，以完成本次分配；否则， 将本次的试探分配作废，恢复原来的资源分配状态，让进程Pi等待。

1. 安全性算法流程

\*step1：设置两个向量

① 工作向量Work: 它表示系统可提供给进程继续运行所需的各类资源数目，它含有m个元素，在执行安全算法开始时，Work∶=Available;

② Finish: 它表示系统是否有足够的资源分配给进程，使之运行完成。开始时先做Finish［i］∶=false; 当有足够资源分配给进程时， 再令Finish［i］∶=true。

\*step2：从进程集合中找到一个能满足下述条件的进程： 

① Finish［i］=false;

② Need［i,j］≤Work［j］； 若找到， 执行步骤(3)， 否则，执行步骤(4)。

\*step3：当进程Pi获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：

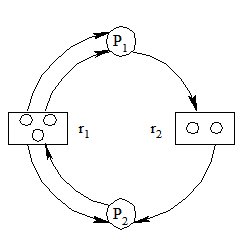
Work［j］∶=Work［j］+Allocation［i,j］;

Finish［i］∶=true;

go to step 2;

\*step4：如果所有进程的Finish［i］=true都满足， 则表示系统处于安全状态；否则，系统处于不安全状态。

1. 死锁的检测和解除
2. 死锁的检测
3. 资源分配图



P：进程结点

R：资源结点

e={pi,rj}：资源请求边，由进程pi指向资源rj，表示进程pi请求一个单位的rj资源。

e={rj,pi}：资源分配边，由资源rj指向进程pi，表示把一个单位的资源rj分配给进程pi。

1. 死锁定理

(1) 在资源分配图中，找到既不阻塞又非独立的进程结点pi。此进程可顺利结束，释放资源。即可消去pi相连的请求边和分配边，使其称为孤立结点。

(2)依次进行。

(3)若所有进程都能称为孤立结点，则称该图为可完全简化。否则为不可完全简化。

\*\* 死锁状态充分条件：当且仅当该状态的资源分配图是不可完全简化的

1. 死锁检测中的数据结构
2. Available：可利用资源向量，表示每一类资源的可用数目
3. Allocation：进程占用资源数
4. Work：资源工作向量
5. Request：进程请求资源向量

\*\*把不占用资源的进程记入L表中，若不能把所有进程都计入L表中，表明系统状态的资源分配图不可完全简化，将发生死锁。

Work ∶=Available;

L∶={Li|Allocationi=0∩Requesti=0}

for all Li ！∈ L do

begin

for all Requesti≤Work do

begin

Work∶=Work+Allocationi;

Li∪L;

end

end

deadlock∶ = (L={p1, p2, …, pn});

1. 死锁解除的方法
2. 剥夺资源
3. 撤销进程：尽可能使撤销的进程数目少或代价小
4. 存储器管理

一、存储器的结构层次

\*理想的存储器：容量大、速度快、价格低

1. 多级存储器结构

CPU寄存器（高速缓存）

主存（磁盘缓存）

磁盘（可移动存储介质）

1. 主存储器和寄存器

①主存储器

\*放置进程运行时的程序和数据

\*外部设备 <—> 主存 <—> 寄存器

②寄存器

\*速度与cpu同步

1. 高速缓存和磁盘缓存

①高速缓存：速度高于主存，用来存放主存中经常访问的信息

②磁盘缓存：主存中暂存对磁盘的读写信息

③硬盘缓存：硬盘与外部总线交换数据的场所

二、程序的装入和链接

1. 程序的装入
2. 绝对装入方式

\*程序中使用的绝对地址既可以在编译或汇编时给出，也可由程序员赋予

\*在程序中采用符号地址，在编译或汇编时，再将符号地址转化为绝对地址

\*装入事先指定的内存地址

\*只适用于单道程序系统

1. 可重定位装入方式

①基本思想

\*程序装入时对目标程序中指令和数据的修改过程称为重定位

\*程序执行之前进行地址重定位（os的装配程序实现从相对地址到绝对地址的映射）

\*装入时一次完成——静态重定位

②特点

\*容易实现

\*程序在主存中只能连续分配；程序在运行期间不能被再移动。无法实现虚拟存储；

\*用户之间难以共享主存的同一程序，若共享需要使用副本，浪费空间（？？？）

1. 动态运行时装入方式（基址寄存器+虚址寄存器）

①基本思想

在程序运行期间进行重定位，即在CPU访问存储单元时才进行程序相对地址到物理地址之间的映射。

②特点

\*程序装入时无需修改

\*地址空间动态分配，主存使用更灵活

\*进程可共享程序的单个副本

1. 程序的链接（目标模块——>装入模块）
2. 静态链接：对相对地址进行修改，变换外部调用符号（CALL—>JSR），预先链接成一个完整的可执行文件
3. 装入时动态链接：装入内存时再链接

优点：便于修改和更新，便于实现对目标模块的共享

1. 运行时动态链接：加快程序的装入速度，节省大量内存空间

三、连续分配方式

\*定义：为一个用户程序分配一个连续的内存空间

1. 单一连续分配
2. 应用范围：单用户单任务的操作系统
3. 应用方式：系统区——os 用户区——用户
4. 特点：简单易实现、浪费存储资源
5. 固定分区分配
6. 应用方式：将内存用户区划分为若干个固定大小的区域，每个区域装入一道作业。为实现分区保护，系统硬件需设置基址和界限存储器
7. 动态分区分配
8. 使用的数据结构：空闲分区表（分区序号、始址、长度）

空闲分区链：空闲分区链成双向链表（按分区起始地址递增的谁序排列）

1. 顺序搜索分区法

①首次适应算法

\*实现：从分区链线性搜索，找到的第一个长度大于等于请求容量的空闲区即进行分配，分配时只划分出需要的长度

\*优点：释放分区时易于合并相邻为较大空闲分区，尽可能利用存储器低端地址，而在高端地址保留较多或者较大空闲分区以满足后继大作业存储要求。

\*缺点：低端地址难以利用的小空闲分区增多，分区分配时搜索次数增加。

②循环首次适应

\*实现：从上次找到的空闲分区的下一个空闲分区开始查找，循环查找

\*优点：空闲分区分配均匀，减少查找的时间开销

\*缺点：缺少大的空闲分区

③最佳适应算法

\*实现：从空闲分区中找到能满足要求容量的最小分区，空闲区的可用表或可用链表按分区从小到大递增顺序链接；

\*优点：可以保留大分区

④最坏适应算法

\*实现：从最大空闲分区中裁取块分配；然后为该分区剩下的存储空间建立空闲分区控制块，插入可用表中。空闲分区按大小递减顺序链接。

\*优点：查找时间少，分配后剩余较大空间可

\*缺点：空闲分区均匀减少，难以满足后到的大作业

1. 索引搜索分区分配算法

①快速适应算法

\*实现：根据空闲分区容量进行分类，每类设置一个空闲分区链表，记录下各类型空闲分区表的头指针。分配内存时，根据需求内存长度，查管理索引表，找到满足要求的最小空闲分区链表，取下第一块分配。

\*优点：查找效率高，不对分区进行分割（？？？）

\*缺点：分区回收复杂

②伙伴系统

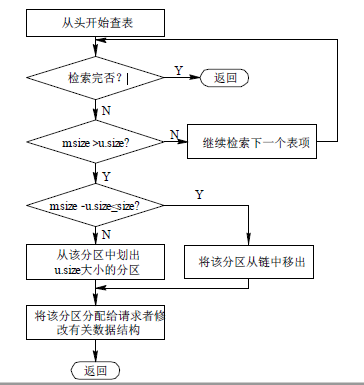
\*实现：启始地址x, 大小2k块的伙伴

③哈希算法

\*实现：以空闲分区大小为关键字构建哈希表

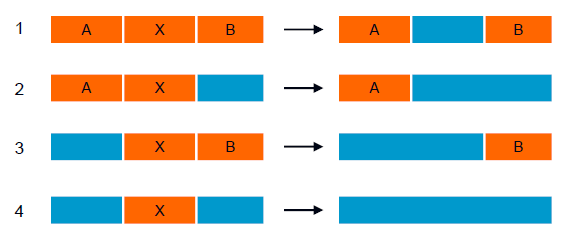
1. 如何分配和回收内存

①分配内存



②回收内存

\*分区合并

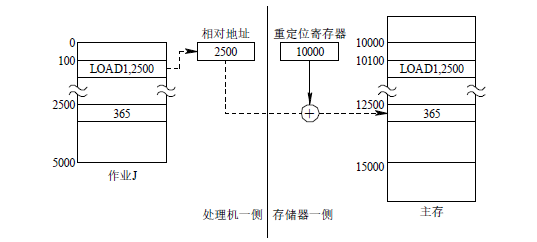


1. 可重定位分区分配
2. 动态重定位的引入

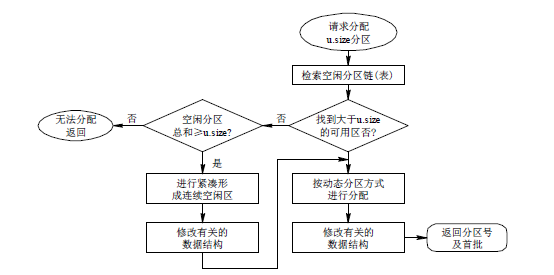
\*思路：进程连续分配需要连续的内存空间，要把分散的小分区拼接成大分区，紧凑后程序的内存地址发生了改变，必须修改地址（重定位）。

1. 动态重定位的实现

\*程序装入后仍为相对地址，在指令执行时再变换为物理地址，设定一重定位寄存器用来保存开始地址



1. 动态重定位分区分配算法



1. 对换
2. 对换的引入

\*思路：把内存中暂时不能运行的进程或暂时不用的程序和数据调出到外存上，再把已具备运行条件的进程或进程所需要的程序和数据调入内存

\*目的：提高内存利用率

1. 对换空间的管理

外存：文件区（离散分配）、对换区（连续分配）

1. 进程的换出与换入

①换出：

系统首先选择处于阻塞状态且优先级最低的进程作为换出进程，然后启动盘块，将该进程的程序和数据传送到磁盘的对换区上。若传送过程未出现错误，便可回收该进程所占用的内存空间，并对该进程的进程控制块做相应的修改。

②换入：

系统应定时地查看所有进程的状态，从中找出“就绪”状态但已换出的进程，将其中换出时间(换出到磁盘上)最久的进程作为换入进程，将之换入，直至已无可换入的进程或无可换出的进程为止。

四、基本分页存储管理方式

\*分区存储管理的问题：碎片较多、进程在分区中连续存放，大小受限制、不利于程序段和数据段共享

\*解决思路：避开连续性要求，一个作业可以存放在不连续的存储空间

1. 页面与页表

（1）页面

\*页面（页）：一个进程的逻辑地址空间分成若干个大小相等的片，并为各页编号，页面大小应为2的幂次

\*物理块（页框）：把内存空间分成与页面相同大小的若干个存储块，为它们编号

\*分配：在为进程分配内存时，以块为单位将进程中的若干个页分别装入到多个可以不相邻接的物理块中。进程的最后一页经常装不满一块而形成了不可利用的碎片，称之为“页内碎片”。

\*页面过小：内存碎片减小，有利于提高内存利用率。但每个进程占用页面较多，导致进程的页表过长，占用大量内存，降低页面换进换出的效率。

\*页面过大：页表减少，提高页面换进换出的速度，但页面碎片增大

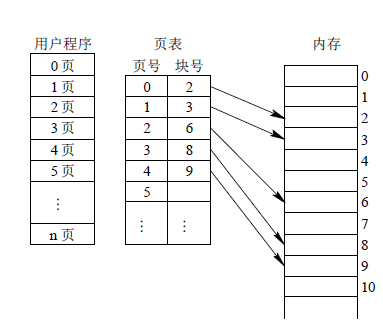
1. 地址结构

\*已知逻辑地址空间中的地址，页面大小，求页号和页内地址

例：页面大小1kB，如A ＝ 2170B；则p=2、d=122

1. 页表

页 <——> 物理块号



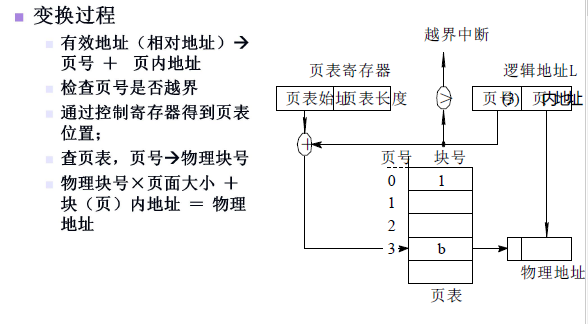
1. 地址变换机构

\*逻辑地址——>物理地址 页号——>物理块号

1. 基本地址变换机构

\*页表：专门的存储器，存储在内存中

\*页表存储器：存储页表在内存中的开始地址和长度



（2）具有快表的地址变换机构

页表指出逻辑地址中的页号与所占主存块号的对应关系。作用：页式存储管理在用动态重定位方式装入作业时，要利用页表做地址转换工作。快表就是存放在高速缓冲存储器的部分页表。它起页表相同的作用。由于采用页表做地址转换，读写内存数据时CPU要访问两次主存。有了快表，有时只要访问一次**高速缓冲存储器**，一次主存，这样可加速查找并提高指令执行速度。

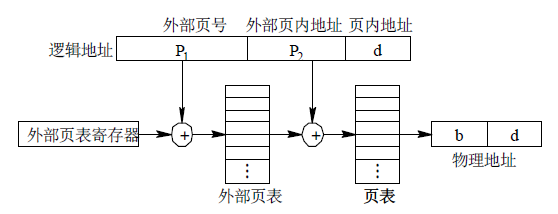
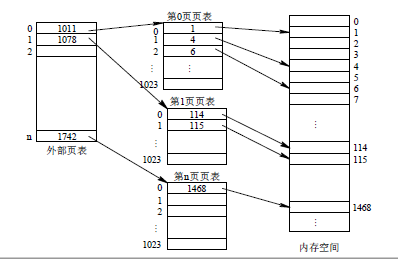
1. 两级和多级页表

\*问题：逻辑地址空间大导致页表非常大，要占用相当大的连续内存空间

\*解决办法：只将当前需要的部分页表调入内存，其余页表仍驻留在磁盘上，需要时再调入

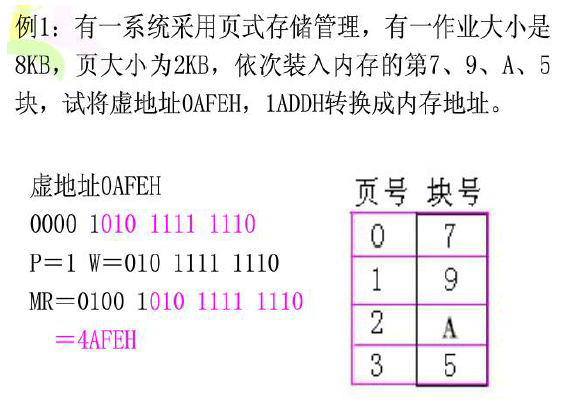
①两级页表

\*页表再分页，并将各页面放在不同的物理块为离散分配的页表建立页表，称为外层页表，外部页表放在内存，部分页表在内存，部分页表在外存



②多级列表

\*\*例题



五、基本分段存储管理方式（方便用户）

1、引入

（1）方便编程：作业按照逻辑分段

（2）信息共享：页无完整信息，段按逻辑划分具有独立意义，适合共享

（3）信息保护

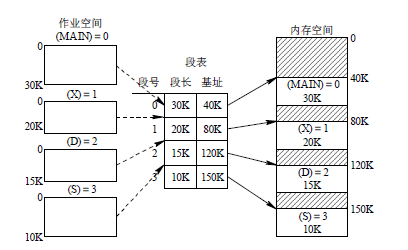
（4）动态增长

（5）动态链接

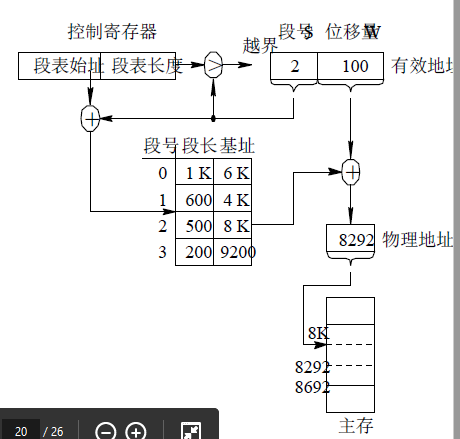
2、基本原理

（1）分段：把程序按内容或过程分成段，长度不固定。每个段有自己的名字或编号；每段从0编址，占用连续的地址空间。按段分配内存（段号+段内地址）

（2）段表：记录起始地址和长度



（3）地址变换机构



（4）分段和分页的主要区别

\*分页：

①页是信息的物理单位，分页是为实现离散分配方式，以消减内存的外零头， 提高内存的利用率。

②页的大小固定且由系统决定，由系统把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而在系统中只能有一种大小的页面

③分页的作业地址空间是一维的，即单一的线性地址空间，程序员只需利用一个记忆符，即可表示一个地址；

\*分段

①段则是信息的逻辑单位，它含有一组其意义相对完整的信息。 分段的目的是为了能更好地满足用户的需要。

②而段的长度却不固定， 决定于用户所编写的程序

③而分段的作业地址空间则是二维的，程序员在标识一个地址时，既需给出段名， 又需给出段内地址。

3、信息共享

4、段页式存储管理方式

\*思路：分段技术管理用户的逻辑地址空间

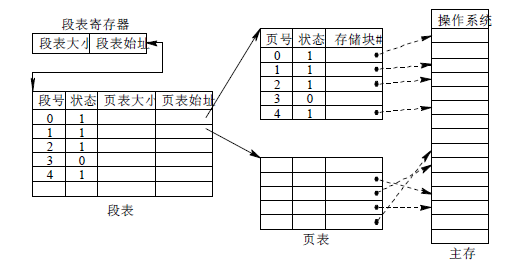
分页技术管理实际使用的主存空间

1. 基本原理

\*用户程序分为若干段，每段分成若干页

\*虚地址构成：段号+段内页号+页内地址

\*每个作业或进程一张段表，每个段一张页表

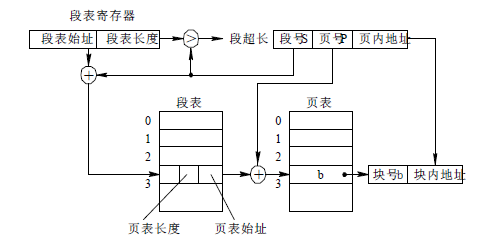


1. 地址变换过程

段表始址，段号 => 该段的页表始址

页表始址 、 段内页号 => 物理块号

物理块号 、 页内地址 => 物理地址



1. 虚拟存储器
2. 概述
3. 作业全部装入内存运行导致的问题：超过内存容量的作业无法运行

内存限制，大量作业在外存等待无法运行

1. 解决方法：增加物理内存、逻辑上扩充内存（虚拟内存管理）
2. 常规存储器管理方式的特征
3. 一次性装入内存
4. 装入后驻留在内存直至结束，即使阻塞也在内存，用过后不再使用的程序和数据仍驻留在内存
5. 局部性原理

（1）时间局限性：某条指令一旦执行，则不久以后该指令可能再次执行；如果某数据被访问过， 则不久以后该数据可能再次被访问。产生时间局限性的典型原因，是由于在程序中存在着大量的循环操作

（2）空间局限性：一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也将被访问，即程序在一段时间内所访问的地址，可能集中在一定的范围之内，其典型情况便是程序的顺序执行。

5、虚拟存储器定义

（1）基本原理：基于局部性原理，装入程序的核心和反复调用执行的部分到内存就开始执行，其他部分执行过程中动态调入。

（2）概念：虚拟存储器， 是指具有请求调入功能和置换功能， 能从逻辑上对内存容量加以扩充的一种存储器系统。其逻辑容量由内存容量和外存容量之和所决定，其行速度接近于内存速度，而每位的成本却又接近于外存。

6、虚拟存储器特征

（1）多次性（基本特征）：作业分多次调入内存

（2）对换性

（3）虚拟性

7、实现方法

（1）请求分页系统：页表机制、缺页中断机构、地址变换机构

（2）请求分段系统：段表机制、缺段中断机构、地址变换机构

1. 请求分页存储管理方式
2. 基本原理

\*只装入核心和反复调用执行的部分，其他部分在执行过程中动态调入

1. 主要问题
2. 虚页不在内存应如何找到对应页面
3. 程序从外存调入内存的存放位置：页面置换
4. 请求分页中的硬件支持
5. 页表机制

– 页号

– 物理块号

– 状态位：该页是否在内存

– 访问字段：访问情况，如次数，时间等信息

– 修改位：是否被修改过

– 外存地址

1. 缺页中断机制

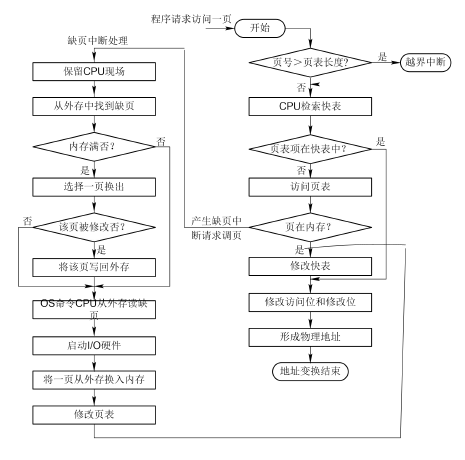
\* 访问的页不在内存中

\* 一条指令执行期间，可能产生多次缺页中断

1. 地址变换机构

\* 产生和处理缺页中断

\* 从内存中置换页



1. 请求分页系统中的内存分配
2. 最小物理块数的确定（保证进程正常运行的最小物理块数）

\*单地址指令且采用直接寻址方式：2块（存放指令&存放数据）

\*间接寻址：3块？？

\*指令长度可能是两个或多于两个字节，因而其指令本身有可能跨两个页面，且源地址和目标地址所涉及的区域也都可能跨两个页面。共6个页面。

1. 物理块的分配策略

\*固定分配（为进程分配固定数目的物理块）局部置换（缺页置换只能在该进程的页面中选取）

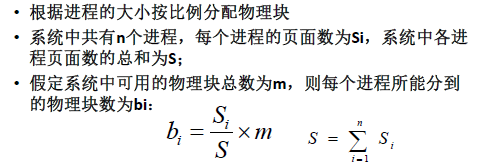
\*可变分配全局置换

\*可变分配局部置换

1. 物理块分配算法

\*平均分配算法：平均分给各个进程

\*按比例分配算法



\*考虑优先权的分配算法

1. 页面调入策略
2. 调入时机

\*预调页策略：预测

\*请求调页策略

1. 调入位置

\*请求分页系统的外存：文件区（离散分配方式）、对换区（连续分配方式）

对换区磁盘I/O速度比文件区高

\*系统拥有足够的对换区空间：全部从对换区调入所需页面，提高调页速度

\*系统缺少足够的对换区空间：不会被修改的文件调入调出不经过对换区，可能被修改的部分在将它们换出时便需调到对换区，下次直接从对换区调入

\*UNIX方式

1. 调入过程

– 访问的页面未在内存 - 缺页中断

– 中断处理程序首先保留CPU环境，分析中断原因后， 转入缺页中断处理

程序

– 查找页表，得到该页在外存的物理块

– 如果此时内存能容纳新页，则启动磁盘I/O将所缺之页调入内存，然后修

改页表

– 如果内存已满，则须先按照某种置换算法从内存中选出一页准备换出；

– 如果该换出页未被修改过，可不必将该页写回磁盘；但如果此页已被修

改， 则必须将它写回磁盘，

– 然后再把所缺的页调入内存， 并修改页表中的相应表项，置其存在位为

“1”

– 将此页表项写入快表中

– 在缺页调入内存后，利用修改后的页表， 去形成所要访问数据的物理

地址，再去访问内存数据。

1. 页面置换算法
2. 最佳置换算法（理想状态，可以作为评判标准）

\*基本原理：淘汰永不使用或最长时间不使用的页面，保证最低缺页率

1. 先进先出（FIFO）

\*基本原理：淘汰在内存最久的页面

1. 最近最久未使用（LRU）

\*基本原理：淘汰据上次被访问时间最长的页面

\*实现：

1. 寄存器：移位寄存器 R=Rn-1Rn-2Rn-3 … R2R1R0，访问该页面时，Rn-1置1。定时右移
2. 栈：最久未访问在栈底
3. 最少使用（LFU）

\*基本原理：记录各页面访问频率，将近期使用最少的页面淘汰

\*实现：移位寄存器，访问该页最高位置1；定期右移该寄存器。（数1的个数？？）

1. clock置换算法
2. 简单clock算法

\* 基本原理：所有页面链接成循环队列，页面设访问位（访问时置1），需要淘汰页面时，顺序检查各页面。访问位为0则淘汰；若为1则置0，暂不淘汰

1. 改进clock算法

\*基本原理：设访问位和修改位

• 1类(A=0, M=0): 表示该页最近既未被访问， 又未被

修改， 是最佳淘汰页。

• 2类(A=0, M=1)： 表示该页最近未被访问， 但已被修

改， 并不是很好的淘汰页。

• 3类(A=1, M=0)： 最近已被访问， 但未被修改， 该

页有可能再被访问。

• 4类(A=1, M=1): 最近已被访问且被修改， 该页可能

再被访问。

\*淘汰过程：

– (1) 从指针所指示的当前位置开始， 扫描循环队列， 寻

找A=0且M=0的第一类页面， 将所遇到的第一个页面作

为所选中的淘汰页。 在第一次扫描期间不改变访问位A。



– (2) 如果第一步失败，即查找一周后未遇到第一类页面，

则开始第二轮扫描，寻找A=0且M=1的第二类页面，将

所遇到的第一个这类页面作为淘汰页。在第二轮扫描

期间，将所有扫描过的页面的访问位都置0。

– (3) 如果第二步也失败，亦即未找到第二类页面，则将

指针返回到开始的位置。 然后重复第一步，如果仍失

败，必要时再重复第二步，此时就一定能找到被淘汰

的页。

1. 页面缓冲算法（PBA）
2. 影响页面换进换出效率的若干因素：页面置换算法、写回磁盘的频率、读入内存的频率

\*基本思想：无论页面是否被修改，淘汰时均放在内存，修改页面的数量达到一定值时再写回磁盘

1. 访问内存的有效时间