操作系统总结 by 白睿

第一章 操作系统 OS 简介

【操作系统概念】1.操作系统（通常称为内核 Kernel，其他程序则为系统程序和应用程序）是一直运行在计算机上的程序，运行在计算机用户与计算机硬件之间，负责进行**资源管理**（硬件资源和软件资源）和**提供服务**。

2.操作系统是资源管理平台；运行应用程序的平台；用户服务平台

**操作系统是一个软件！

**有时系统程序包括操作系统

**驱动程序也是操作系统

【操作系统功能】**资源管理**（硬件资源和软件资源）和**提供服务**

【操作系统目标】①有效使用计算机**硬件**②方便**用户**使用计算机③通过运行计算机**程序**方便用户解决问题

【计算机系统结构/组成】计算机系统由 4 部分**层次结构**构成：**硬件、操作系统、系统程序和应用程序、用户**

【计算机系统组织】启动、中断、I/O 结构、存储结构

【计算机系统操作】单个或多个中央处理器、设备控制器通过总线进行操作，包括启动、中断

【启动】计算机启动过程：

1）BIOS（Basic Input Output System）：确认每个设备是否工作正常，确认设备无误后，启动**引导程序**（Bootstrap）

2）Bootstrap 引导程序（位于 ROM 或 EEROM 中称为计算机硬件中的固件）：设备初始化；载入操作系统至内存；运行第一个进程 init()，等待事件发生

**以上两步均是操作系统执行的

**完整的过程是：电源→主板→BIOS→Bootstrap

【中断】通过硬件或软件中断实现事件的触发。现代操作系统是以**中断或事件**驱动的。

【软中断】系统调用（System Call）、异常抛出（软中断不可屏蔽）

【硬中断】硬件（I/O）向 CPU 发送中断信号触发中断，用户不可见，但可触发程序的运行，如：网卡接收数据包、点击热键。（硬中断可以屏蔽）

【中断服务程序】每个中断都有自己的中断服务程序

【中断向量表】操作系统拥有中断与中断服务程序之间的对应表，用于管理中断：即确定每个中断的具体操作

【中断操作流程】当发生中断时:保存当前进程的程序计数器 PC（目的：使中断运行完毕后能返回原进程继续执行）；跳到相应的中断服务程序执行中断。中断服务程序运行结束后：返回到被中断的程序并继续往下运行 或 返回到 OS 制定的程序中。

***同一个机器上，不同操作系统都有相同的机器指令（中断指令），但（不同热键对应的）中断操作可能不同

【I/O 结构】**I/O 设备**如何做到**与 CPU 同时运行**

1）每种类型的 I/O 设备有其对应的**设备控制器**，每个设备控制器拥有**本地缓冲器**和寄存器

2）CPU 只负责内存与本地缓冲器之间的数据传输，设备控制器负责其控制的 I/O 设备与本地缓冲器之间的数据传输。

3）I/O 操作结束后（**传输完一个字节**），设备控制器通过中断通知 CPU 表示 I/O 结束

【内存与设备进行 I/O 的两种方法】

1） 磁盘控制器每个字节传送完成后都触发中断通知 CPU 表示 I/O 结束。

2） DMA 直接访问内存

【DMA 方式】在专门的硬件控制下，实现高速外设和主存储器之间自动成批交换数据尽量减少 CPU 干预的输入输出操作方式

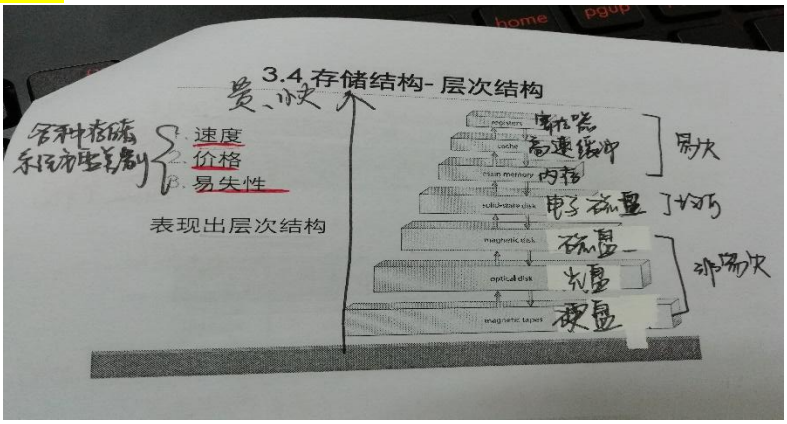
【DMA 特点】1.有利于高速 I/O 设备传送数据，接近于内存速度

2.DMA 控制器在没有 CPU 干涉的情况下，以块为单位，负责在设备缓冲器与主存之间的数据传送，减少了操作系统的负担

3.DMA 每完成块传送才触发中断

【I/O 方式/方法】**同步**（ I/O 过程中用户无控制权，只有 I/O 结束后，用户程序才能获得控制权）；**异步**（I/O 过程中用户有控制权）

【存储结构】根据各种存储系统在**速度、价格、易失性**三方面表现为**层次结构**



【主存】CPU 可以直接随机访问的唯一大容量存储设备，易失

【二级存储】不易失的存储设备，一般是磁盘类

【磁盘】磁盘表面可以逻辑划分为多个磁道（圈），而每个磁道再划分为扇区；磁盘控制器负责设备与计算机之间的逻辑交互

【闪存】芯片级别，移动存储设备；分为 Nand Flash 和 Nor Flash（二者存储方式不同）

【固态状态硬盘/固态硬盘】比磁盘速度快，更普及，但贵、读写次数少、寿命短，是基于 Nand Flash 的集成品。

***CPU 仅和内存打交道，所以所有数据均要通过内存来读写

***数据先存在内存中，关机后再存到硬盘中，可提高效率。所以点击保存后突然断电再开机文件是存储失败的，因为内存的断电则失了，而且是异常断电也没有存到硬盘中。

【计算机系统体系结构】1+3+2

计算机系统体系机构分为单处理器系统、多处理器/并联系统（非对称/异构多处理器、对称/同构多处理器、多核处理器）、集群系统（非对称集群、对称集群）

【单处理器系统】只有一个通用处理器处理用户指令，其他专用处理器（磁盘控制器、图形控制器）不直接接受用户指令，只能通过操作系统来接收用户指令。有的专用处理器与通用处理器集成在一起，通用处理器具有专用处理器功能。

【多处理器系统】有多个处理器，又称并联系统，多处理器共享一个内存。

特点：1.CPU 之间通过共享内存来进行通讯

2.操作系统可以运行在某一个 CPU 上或多个 CPU 上（运行在单个更方便）

优点：增加吞吐量，方便扩展（有多个 CPU 槽），增加可靠性（具有容错性）。

【非对称处理器】异构多处理器（AMP），各处理器结构不同（即速度可以不同）；一个处理器负责运行操作系统，其他处理器运行其他程序，处理器之间有主从关系，著处理器调度从处理器并安排工作。

【对称处理器】同构多处理器（SMP），各处理器结构相同；操作系统可运行在任一处理器（所以每个处理器都要能够完成操作系统中的所有任务）。（最常用对称多处理器）

【多核处理器】一个处理器有多个 CPU（即多个处理器集成到一个芯片上），可以是对称或非对称。

***不管是多处理还是多核，每个 CPU 都有自己的**寄存器和高速缓存。**

【集群系统】由相互连接的计算机组成的并行分布式系统，共享存储，可以作为单独、统一的计算资源来使用。

特点：多个自治系统（有自己的操作系统的计算机系统）协同工作；每个节点（计算机系统）都有自己的操作系统（但它们的工作目标相同）；一个集群系统可以看成是单个逻辑计算单元，它由多个节点通过网络连接组成。

优点：高性能

缺点：高维护费用。

【非对称集群】一个节点为主节点，即主机待机模式的节点，主节点控制其他子节点。集群之间想要通讯必须通过主节点。

【对称集群】没有主节点，所有节点地位相等，无主从关系。

【区别】集群系统是多处理器系统的一种，只是集群系统有多个计算机而多处理器系统只有一个计算机。所以集群系统也是分布式系统。

【操作系统结构&操作】3+2

操作系统结构有三种：批处理系统、多道程序系统、分析系统（多任务系统）

操作系统操作为双重模式操作，通过 CPU 的模式位这个硬件来区分内核模式和用户模式。

***计算机/系统/程序的功能：计算 & I/O

【批处理系统】I/O 时 CPU 空闲（因为 I/O 操作相对于 CPU 操作速度很慢），此时必须由操作员调度另一个程序运行，从而提高 CPU 的使用率。（空闲+人工任务调度）

【多道程序系统】I/O 时由调度器按一定机制自动选择另一个任务并运行。为保障 CPU 总有任务运行，多道程序系统必须把需要运行的多个任务都载入内存以备调度。

【分时系统(多任务系统)】为减少每个用户程序的响应时间，让每个任务公平使用 CPU，分时系统给每个用户一个给定的时间片，CPU 每隔一个**时间片**切换一次任务，在多用户之间相互切换。

【区别】

1.批处理系统：人工任务调度，程序直接插入电脑，分别载入内存运行（DOS）

多道程序系统：调度器自动调度任务，有选择机制，多任务全部载入内存

2.因为多道程序系统只能由调度器来选择哪个任务可以运行所以可能不公平，而且为了**确保**多用户交互系统每个用户**响应时间**小于一秒，所以采用分时/多任务系统，但分时/多任务系统也需要调度机制。

3.多道程序和多任务目的不同：
多道程序系统：为了提高 CPU 的使用率，让 CPU 忙，始终有任务运行
多任务系统：让每个任务能公平使用 CPU，体现公平性
4.现在的操作系统是多道程序和多任务混合（LINUX \ UNIX）
【操作系统操作】**双重模式操作**，以保护操作系统安全。硬件提供**模式位**来区分两种代码。
【用户模式 **1】**用户掌握计算机控制权
【内核模式 **0】**操作系统掌握计算机控制权，**系统调用**时由用户模式转为内核模式。
***用户发系统调用的指令，具体执行只能操作系统来。
***不同系统的系统调用不同， 但面向用户的应用程序接口 API 同，即不同系统的不同名函数通过相同接口让系统做相同操作
***为什么要保护操作系统安全?
1. 一个操作系统可以被多个用户、多个程序共享
2. 非法或不正确的操作会导致系统崩溃或破坏
3. 一个程序的运行可能会影响另一个程序的运行，如无限循环有可能会导致系统失灵或破坏
***内核模式又称：管理模式//系统模式//特权模式//监督程序模式
***内存空间中有用户代码&操作系统代码两区域，通过 CPU 模式位的数值来确定运行区域

【操作系统管理】

操作系统管理分为 6+：进程管理、内存管理、存储管理、I/O 管理、网络管理、安全管理……

【进程管理】进程是运行中的程序，是系统的**运行单元**。

进程管理包括（5）：创建和删除进程、挂起和重启进程，进程同步，进程通讯，死锁处理等。

【区别】
程序是被动实体，进程是活动实体。
一个程序从硬盘载入内存才是进程，然后赋予 CPU 运行，则变成正在进行的进程。
所以只要载入了内存就是进程。

【注意】
1. 在一个系统中有无数个进程在**同时**运行，运行在一个或多个 CPU 上（多道程序系统），进程之间通过**复用 CPU 并发**运行。(因为一个 CPU 只能赋予给一个进程，所以如果有多个进程就只能复用 CPU)
2. 运行进程需要分配一定的资源，如：CPU，内存，I/O 设备，文件等
3. 进程结束时，应收回已分配的资源，即让进程有效使用这些资源。

【内存管理】目的是提高**内存使用率**，从而有效使用内存。管理内存中的**数据存储**&**指令运行**（因为内存中存储的就是程序和数据）

内存管理包括（3）：记录内存使用信息，决定进程的载入，分配和释放内存空间等。

【存储管理】

【文件】文件是操作系统对存储设备物理属性的抽象定义，它是存储的逻辑单元

1）文件系统管理：创建删除文件和目录，提供操作的原语，文件映射到二级存储设备，在稳定存储设备上备份文件等。

2）大容量存储系统管理：一般为二级存储设备如硬盘，

管理活动：空闲空间的管理，分配，调度。

3） 高速缓存（Cache）：临时存储设备，设置在高速设备与低速设备之间（CPU 与内存间）。当高速设备从低速设备上读取数据时，会把数据临时复制到高速缓存上。I/O 操作开始进行的时候，操作系统先到高速缓存里查找，没有再去磁盘中找。

*但！在多任务、多处理器、分布式环境下必须要保证**数据一致性**，用硬件解决。

***计算机可以在多种类型的物理介质上存储信息

***每种介质通过一个设备来控制，如磁盘驱动器。每个设备都有自己的设备控制器/驱动器，但是用户不能直接控制设备，必须间接通过操作系统来控制设备。

***文件通常组成目录以方便使用

***多用户访问文件时，需要控制权限问题。

*****缓存不能代替存储设备：因为其容量小，价格贵

缓存满了则替换，有一定的替换策略，最常用的是根据命中率替换。

【I/O 管理】

I/O 子系统目的：针对用户隐藏具体硬件设备的特性。

I/O 子系统组成：1）一个包括缓冲（buffer）、高速缓存（cache）和假脱机（spooling）的内存管理部分

2）通用设备驱动器接口

3）特定硬件设备的驱动程序

***LINUX 系统中有统一的 I/O 接口，均是通过文件管理系统接口来操作。

【区别】

缓冲：为**传输数据**暂时存储数据

缓存：为**性能提高**暂时存储数据

假脱机：是关于**低速字符设备**与计算机**主机交换信息**的一种技术，又称**外部设备联机并行操作**。

***假脱机也是在高低速设备之间用于提升性能的，但其以“字符”为单位传输，而硬盘以“块”为单位传输

***假脱机技术优点：提高 I/O 速度；设备并没有单独分配给任何一个任务；实现了虚拟设备功能从而可以共享设备

【其他计算机系统】

【分布式系统】将**物理上分开**的、各种**可能异构**的计算机系统通过**网络**连接在一起，为用户提供系统所维护的各种资源的计算机集合（集群系统是分布式系统的一种）

【实时系统】系统中的任务有时间节点(截止时间)的系统，如军事、医疗

【嵌入式系统】嵌入式系统资源有限，只保留最必要的内容，所以与一般系统相比较小

【其他计算环境】

【客户机/服务器计算（C/S)】分为两类：计算服务器系统 & 文件服务器系统。存在瓶颈问题，即服务器负担太重，因为所有消息传递均经过服务器

【点对点计算 P2P】每个计算机又时客户机又是服务器，复杂

【基于 Web 计算（B/S)】负载平衡器/负载均衡（利用分布式，将业务服务器与数据库服务器分开，每次将请求分派给不同的服务器）

第二章 操作系统结构

【操作系统服务】操作系统以不同形式向用户和程序提供服务：**6+3**

1.系统服务提供对用户有用函数：**基本服务**

【用户界面】命令行界面（CLI），图形用户界面（GUI）

【CLI】允许用户直接输入操作系统完成的命令

*有时在**内核**中**实现，**有时通过**系统程序实现**

*有**多种实现方式**的时候称之为**外壳**

***主要作用**是获取并执行用户指定的命令，有些命令式内置的，有些只是程序名

【程序执行】将程序装入内存并运行

【I/O 操作】I/O 可能涉及文件 or 设备

【文件系统操作】程序需要读写文件 & 目录

【通信】进程之间信息交换

【错误检测】OS 需要知道可能出现的错误

2.通过共享计算机资源来提高效率：**增值服务**

【资源分配】多用户或多作业并发运行时分配资源

【统计】资源使用记录

【保护和安全】保护：确保所有对系统资源的访问受控，安全：确保系统不受外界侵犯，延伸到外部 I/O 设备不受非法访问

【系统调用】1.**操作系统服务的面向应用程序的编程接口（API）**。

2.通常用高级语言编写(C or C++)。

3.程序通过应用程序接口 API 访问，而不是直接使用系统调用。

【三种常用 API】Win32 API、POSIX API（包括所有 UNIX,LINUX,MAC OS X），Java API。

【简答】

Q：为什么不直接调用系统调用而要通过 API 来调用系统服务？

A：系统调用依赖于系统，且昂贵；应用程序接口 API 有兼容性，使程序可移植且操作简易

【系统调用实现】每个系统调用都有一个固有番号（system call number）

操作系统通过一张系统调用番号表来管理系统调用接口

开发者无需关心系统调用的实现，只需调用运行库提供的 API

一般系统调用**运行库**来管理，**运行库提供 API**

【系统调用参数传递方式】**（3）寄存器、块**（参数保存在内存的一个块中，把块地址用寄存器传递给系统调用函数）、**栈**（以栈的形式保存在内存中，用户程序向栈 push 参数，操作系统 pop off 参数）

【对比】

C 语言中传递参数的方式有 2 种：传值（value）&传地址（reference）

***不是一个操作系统这三种调用参数传递方式均有，而是不同操作系统总共有这三种

【系统调用分类】**（6）**进程控制、文件管理、设备管理、信息维护、通信、保护。

【系统程序】**系统程序给用户提供一个方便的环境，以开发程序和执行程序**。分为**文件管理、**

状态信息、文件修改、程序语言支持、程序装入和执行、通信。

***系统程序不属于内核，但属于操作系统的一部分。为用户使用操作系统服务，

【对比】

应用程序的服务对象：用户

系统程序的服务对象：另一个程序

【操作系统的设计与实现】

【设计目标】**用户目标**（易用易学可靠安全快速）和**开发目标**（易射击实现维护、灵活可靠高效无错）

【策略和机制】**策略**（做什么）；**机制**（如何做）

【对比】

机制决定如何做，是实现细节。策略决定做什么，是策略决定。

***策略与机制的区分对于灵活性来说很重要

***策略可能会随时间或位置而有所改变

***每次策略改变都可能需要底层机制的改变

Eg：定时器结构是一种 CPU 保护的机制，但是对于特定用户将定时器设置成多长时间是个策略问题。。。。因为不同用户对定时器设置的决定使策略决定，为了实现这个决定的时间所采用的具体结构是机制。

【实现】早期：汇编语言，现在：高级语言（C、C++），实际混合多种语言：底层汇编，主体 C，系统程序 C/C++等

【对比】

汇编语言：运行高效、编程耗时、不易移植

高级语言：运行效率差，编程高效，易移植

【操作系统结构】

操作系统结构分为五类：简单结构、层次结构、微内核、模块结构、混合结构

【简单结构】功能不划分成模块（MS-DOS、UNIX），接口和功能层没划分清楚

【层次结构】将操作系统划分为若干层，底层 0 为硬件，顶层 N 为用户层。在底层上构建高层。每层只使用低层次的功能和服务。

优点：简化了系统设计和实现；便于调试和升级维护

缺点：层定义较难；效率差

【微内核】内核微型化：从核内移出尽可能多功能到用户空间

***发生在 用户模块间的通信 使用 消息传递 形式

***将非基本部分移出内核，变为系统程序或用户程序（Mash、WinNT 2000 3000）

【模块结构】现代操作系统，面向对象，每个核心部件分开，每个与其他组件的会话被称为接口，每个组件在需要时被加载到内核

***模块结构类似于分层方法，但更灵活

【混合结构】（Mac OS X）

【虚拟机】通过软件模拟的具有完整硬件系统功能的、运行在一个完全隔离环境中的完整计算机系统（JVM）

***物理计算机的资源被共享，以创建虚拟机，虚拟机是虚拟的物理环境

***虚拟机概念提供对系统资源的完全保护，因为每个虚拟机同其他虚拟机隔离

***虚拟机是研发操作系统的完美载体

***由于需要对物理机器进行精确复制，虚拟机实现困难

***Java 虚拟机有自己的硬件架构：处理器、堆栈、寄存器；其具有相应的指令系统；JVM 屏蔽了于具体操作系统平台相关的信息，使得 Java 程序只需生成在 java 虚拟机上运行的目标代码就可以在多种平台上不加修改地运行。(只要 Java 带着它自己的 JVM，他们整体就可以在多种 windos/linux 等系统上运行，因为 java 程序的运行不取决于具体系统，其就是在 JVM 上运行的)

第三章 进程

***编译四过程：预处理、编译、汇编、链接

【预处理】把程序中用到的头文件库中的代码引入到程序中（inline），c.→i.

【编译】转换为汇编语言文件，i. →s.

【汇编】得到机器语言，s. →o.

【链接】得到可执行的目标文件，ELF 格式（Executable/Linkable File）

*【进程的概念】*进程是运行中的程序。当程序被载入内存，就会变成进程。

进程的组成（4）:程序代码（**文本段/代码段 code section**）；通过程序计数器和处理器寄存器的内容来表示当前活动（**数据段 data section**：包含全局变量）；**栈**（包含临时数据：函数参数，返回地址，局部变量）；**堆**（在进程运行期间动态分配的内存

*****【注意】*****

局部变量保存在栈中，全局变量保存在数据段中

***操作系统会执行程序、进程、任务、作业

***每时每刻只有一个进程可被赋予 CPU，其他进程仍在内存中等待

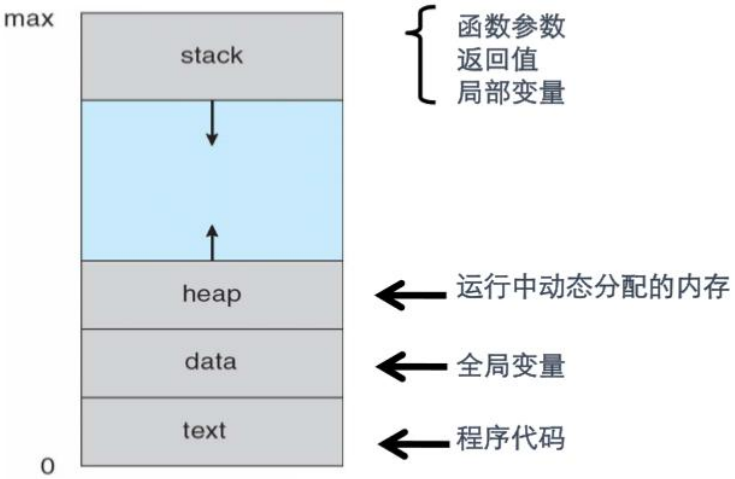
*****【区别】*****

作业是任务的实例。(同一人物在不同时间段执行不同作业，作业是在运行的实体)

进程在内存中的表现形式：进程控制块 PCB（Process Control Block）

***进程控制块又称任务控制块 TCB，是一种数据结构

***每个进程在操作系统内都用 PCB 表示，它包含许多于一个特定进程相关的信息（8）



***操作系统管理物理和逻辑资源：

1. 物理资源：处理器、内存、I/O 设备
2. 逻辑资源：进程、虚拟内存、数据结构
3. ***操作系统是通过"**控制表**"对系统中的每个资源进行控制（表就是资源与其编号的对应）

【*进程状态*】**新的、运行、等待（阻塞）、就绪（等待分配处理器运行）、停止**：

【进程状态图】

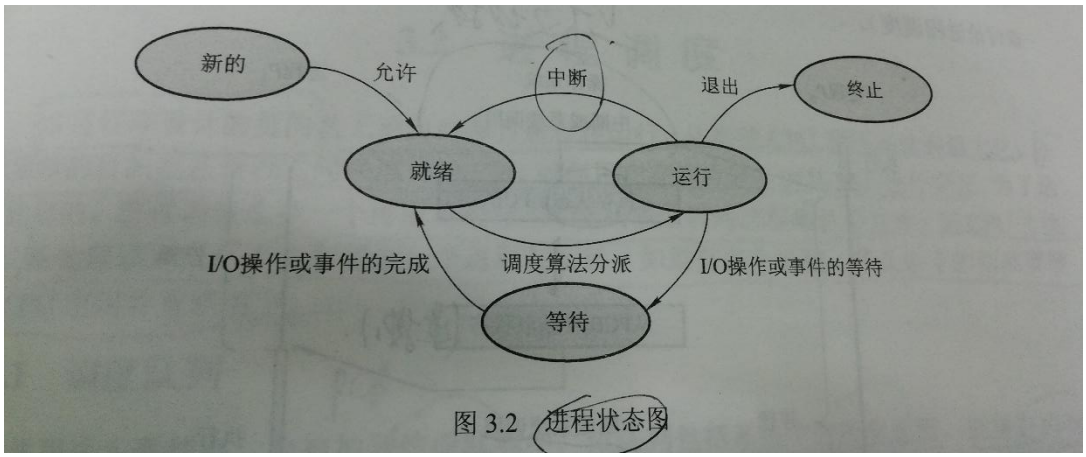


图 3.2 进程状态图

【*进程控制块 PCB 组成*】**每个进程对应一个 PCB（8）**

【进程 ID】PID，操作系统通过这个唯一的进程标识符来识别进程

【进程状态（5）】新的、运行、等待、就绪、停止

【新的 new】进程正在被创建

【运行 running】指令正在被执行

【等待（阻塞）waiting（blocking）】进程等待某个事件的发生

【就绪 ready】进程等待分配处理器

【终止 terminated】进程完成执行

【程序计数器】记录进程要执行的下一条指令地址

【CPU 寄存器】累加器、索引寄存器、堆栈指针、通用寄存器、其他条件码信息寄存器

【CPU 调度信息】进程的优先级、调度队列的指针和其他调度参数

【内存管理信息】基地址、界限地址、页表、段表

【记账信息】CPU 使用时间、时间界限等

【I/O 状态信息】I/O 设备进程分配状态、打开文件表

***进程之间是树形结构，父进程生成子进程

【*进程调度*】

【进程调度队列】多道程序的目的是无论何时都有进程在运行，从而使 CPU 的利用率达到最大。为此，CPU 需要在多个可用进程之间进行快速切换，**调度程序**从多个可用进程中选择一个进程运行。CPU 在多个进程之间切换，需要从调度队列中选取进程（PCB）运行。

操作系统持有就绪队列和一组设备队列，进程可以在多个调度队列之间移动

【就绪队列】等待分配 CPU 运行的进程。不一定是 FIFO 队列。一般用链表来实现。当进程创建时就被放到该队列，是驻留在内存中的就绪并等待运行的进程。

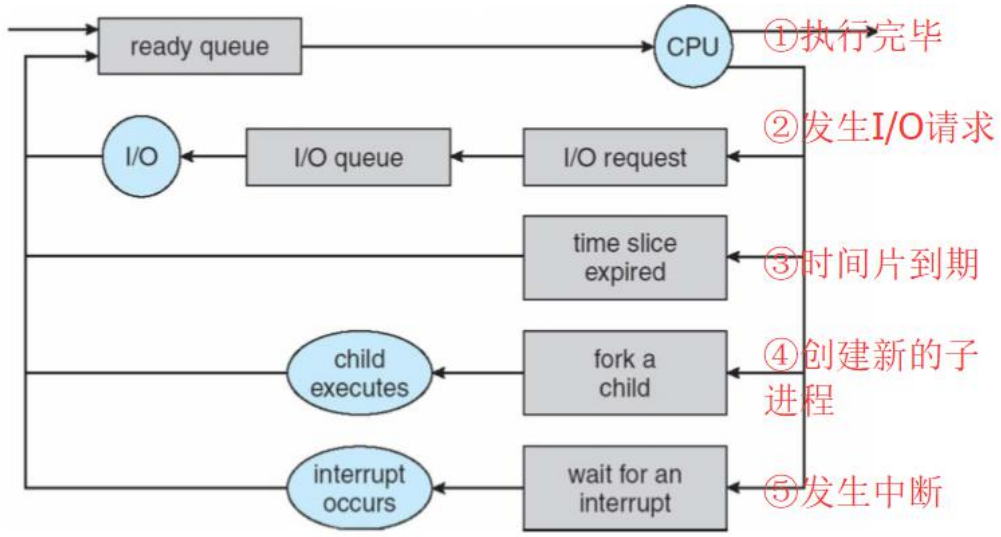
【设备队列】等待特定 I/O 设备的进程，**每个 I/O 设备都有自己的设备队列**。

***调度队列分为 2 部分：就绪队列&设备队列

【进程调度】

进程分配到 CPU 并执行时（运行状态），可能发生如下事件，最终进入就绪状态，被放回就绪队列：**（4）**

1. 进程发出一个 I/O 请求，并被放到 I/O 队列（即设备队列）
2. 进程创建一个新的子进程，并等待该子进程结束
3. 进程可能会由于中断而强制释放 CPU，并被放回就绪队列
4. 用完时间片，并被放到就绪队列



【调度程序】从调度队列中选择进程的程序。**区别**在于**执行的频率**（短期执行更频繁）。

【长期调度程序（作业调度程序）】从存储设备的**缓冲池**中选择进程，并装入就绪队列（装入内存中）等待执行。(I/O 调度)

【短期调度程序（CPU 调度程序）】从准备执行（就绪）队列内存中选择进程，分配 CPU 执行。

【中期调度程序】用于分时(多任务)系统。核心思想是将进程从内存中移出（从 CPU 竞争中移出），从而降低多道程序设计的程度。之后重新调入内存从断点处执行（Swap in/Swap out）。

***分时系统，增加了中期调度程序（长、中、短均有）

***设备→内存就绪队列（长期），内存就绪队列→CPU（短期）,分时系统（中期）

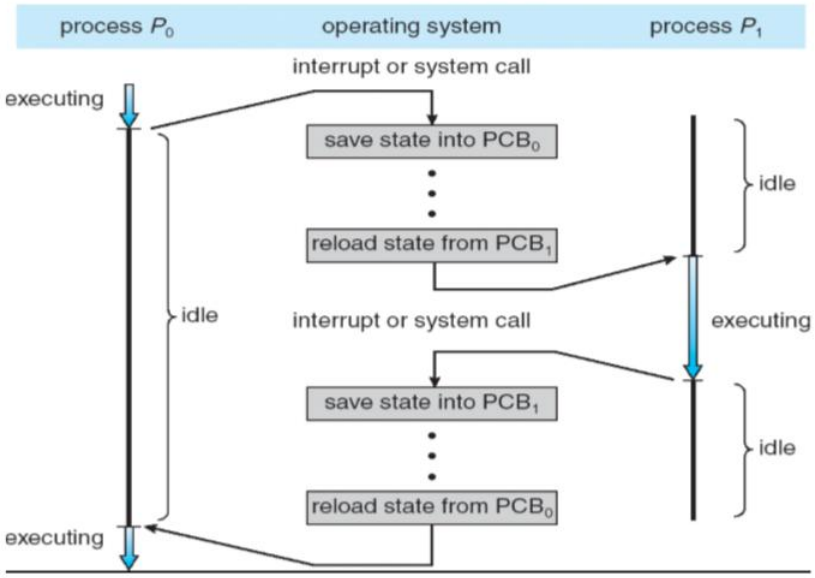
【上下文切换】将 CPU 切换到另一个进程时，系统需要保存当前进程的状态（以备被切换的进程恢复运行）并恢复另一个进程 状态（PCB），这一**任务**称为上下文切换。

***上下文切换时间额外开销。（因为在保存和载入的过程中 CPU 空闲，系统什么都不能做）

***上下文切换速度依赖**硬件**（内存速度、寄存器数量、指令运行时间、载入/保存时间）

***为提高上下文切换速度，有些处理器提供多组寄存器集合，上下文切换只需要简单地改变当前寄存器组地指针

***上下文切换的次数是衡量一个调度算法性能好坏的标准之一。（因为如果切换时 CPU 啥也干不了，所以如果切换次数过多，则耗时太长）



【进程操作】分为进程创建、进程载入和执行、进程终止

【进程创建】

【进程树】系统中进程的结构是树形结构，进程之间存在父子关系，父进程创建子进程，形成进程树。（操作系统是根据一个唯一的进程标识符 PID 来识别系统中的进程。一个进程可以创建另一个进程，创建的进成为父进程，被创建的新进程为子进程）

【进程创建选项】资源共享选项、执行选项、地址空间选项

【资源共享选项】（父子进程）共享资源、共享部分资源、不共享资源

【执行选项】（父子进程）同时执行、父进程等待子进程结束

【地址空间选项】子进程完全复制父进程内容、子进程覆盖父进程内存空间

【进程创建步骤】

1）在系统内创建 PCB（复制父进程 PCB）

2）分配内存空间

3）载入可执行文件（程序）

4）初始化并运行程序

-***【注意】***-

子进程可能有两种情况：

1. 子进程与父进程执行相同任务，这样则无需调用 exec（），完全复制父进程即可。
2. 子进程与父进程执行不同任务，这样则需要调用 exec（），覆盖父进程。

***覆盖父进程的意思是覆盖子进程复制过来的父进程，而原来的那个父进程不会改变

***fork（）进行的是进程创建步骤的 1)2),exec()进行的是 3)4)，二者均是系统调用，exec（）用于重载运行程序

***没有 exec（）时，父进程和子进程除了 PID 和内存空间位置不同外完全一样

***fork（）的返回值：

1. 子进程：返回 0
2. 父进程：返回子进程的进程标识符 PID>0

***如果父进程创建了子进程之后没有等待子进程，那么二者的执行顺序是随机的。如果等待则子进程运行结束后再运行父进程。Wait（NULL）

【进程终止】进程终止有两种情况：

1. 子进程完成最后语句并使用 exit()系统调用，请求操作系统删除自身并终止运行：
   这时，子进程可以通过父进程的 wait()返回状态值给父进程。
2. 父进程可以用 abort()系统调用终止子进程的执行。父进程终止子进程有三大原因：
   1） 子进程使用了超过它所分配到的资源。
   2） 分配给子进程的任务不再需要
   3） 父进程终止导致子进程终止，即父进程终止，操作系统不允许子进程继续运行

【级联终止】如果一个进程终止，它的所有子进程都将终止。

【僵尸进程】进程终止的特殊情况。一个子进程结束，但它的父进程还没有等待它（调用 wait（）/waitpid（）），即没有父进程的进程。(子进程比父进程先结束，而父进程又没有回收子进程, 释放子进程占用的资源, 此时子进程将成为一个僵尸进程。如果父进程先退出 ，子进程被 init 接管，子进程退出后 init 会回收其占用的相关资源）

***不管父进程有无 wait（），只要是子进程进行完了但是父进程还没进行完，那么从子进程结束到父进程结束的这段时间子进程就是僵尸进程。

***如果父进程先进行完但是子进程还没完，那么子进程会被交给 init（）上帝进程，此时子进程叫孤儿进程

【进程间通信 **IPC**】协作进程间需要共享数据，需要进程间通信机制（Inter-Process Communication）

PS：并发运行的进程可能出现的状态：相互独立 或 协作工作

【独立进程】一个进程不能影响其他进程或不被其他进程所影响, 那么该进程是独立的, 进程之间没有共享数据

【协作进程】一个进程影响其他进程或被其他进程影响，那么该进程是协作的，进程之间有数据共享

【进程协作的理由】(4）信息共享、提高运算速度、模块化、方便

【进程间的通信模型】(2）消息传递 & 共享内存

【消息传递】需要内核干涉（经过内核），适合传递少量数据，实现简单。

消息传递提供两种操作：发送（消息） & 接收（消息）,消息可以定长或变长

***如果进程 P 和 Q 需要通信，首先需要建立通信线路（communication link），并相互发送消息和接收消息

【通信线路的逻辑实现方法】(3）直接或间接通信；同步或异步通信；自动或显式缓冲

【直接通信】通信的每个进程必须明确地命名发送者和接收者（**对称寻址→一对一**）；只命名接收者，接收者不需要命名发送者（任何进程发送的数据均可被接收者接收）（**非对称寻址→多对一**）

【间接通信】通过共享同一个**端口**（port）或**邮箱**（email）实现通信（通过端口或邮箱来发送或接收消息，每个邮箱都有一个唯一的标识符，进程之间可以通过共享端口来进行通信）

***操作系统拥有的邮箱（端口）是独立存在的，操作系统提供机制以允许进程进行如下操作：1.创建或删除邮箱（端口）2.通过邮箱（端口）发送和接收消息

***世界标准邮箱有：web（80），ftp（21），telnet（23）

***端口即数字，是内存的一部分空间，用于网络交互的空间

【同步方式】发送者与接收者是否阻塞

【同步发送】发送进程阻塞，直到消息被接收

【异步发送】发送进程发送消息并继续操作
【同步接收】接收进程阻塞，直到收到消息
【异步接收】接收进程不阻塞，收到一个有效消息或空消息
***TCP 协议：同步，传递消息的双方语言一致
***UDP 协议：异步，传递消息的双方语言不一致
【缓冲方式】通信进程所交换的信息都驻留在临时队列中，消息队列实现缓冲（message queue 可多发送者多接收者）
消息队列的实现方式有三种：零容量、有限容量、无限容量，其中零容量认为没有缓冲，另外两个均是自动缓冲
【零容量】无缓冲，队列的最大长度为 0，线路中没有等待的消息（阻塞/同步）
【有限容量】队列长度有限，当缓冲满，发送者等待，直到队列中的空间可用
【无限容量】队列长度无限。发送者从不等待
【对比】
1. **直接通信的通信线路的属性：**
   1) **线路是自动建立的**
   2) **一个线路只与两个进程相关**
   3) **两个进程之间只有一个线路**
2. **间接通信的通信线路的属性：**
   1) **两个进程共享一个邮箱（端口）就可以建立通信线路**
   2) **一个线路（端口）可以与多个进程相关联**
   3) **两个通信进程之间可有多个通信线路，每个线路对应一个邮箱（端口）**

（3. 直接通信的对称寻址可以实现一对一，非对称寻址可以实现多对一，间接通信可以实现以上两种+多对多）

【共享内存】无需内核干涉（经过内核），允许以最快的速度进行方便的通信，速度更快（效率高、性能好、实现复杂<因为内核不干涉，所以同步通讯的机制均要人来设置>，内存中某一部分空间作为交互媒介）
【安全性】每个进程都有自己受保护的内存地址空间，通常一个进程不能访问另一个进程的内存空间（操作系统阻止）
***所以为了实现共享内存、便于两个或多个进程可以访问内存，共享区域应该取消安全性限制；并且，必须保障；不能两个以上进程同时向共享区域写数据（写的同时读也不行，但可以同时读）。

【生产者-消费者问题】共享缓冲区域（**此缓冲即为共享内存区域**），该共享内存区域有两种实现方式：无限缓冲、有限缓冲。
【无限缓冲】对缓冲大小没有限制。如果缓冲为空，消费者必须等待，而生产者总是可以产生新的信息。**（空不可读无限写）**
【有限缓冲】缓冲大小固定。如果缓冲为空，消费者必须等待，如果缓冲为满，生产者不能产生新的信息。**（空不可读满不可写）**
***

有限缓冲

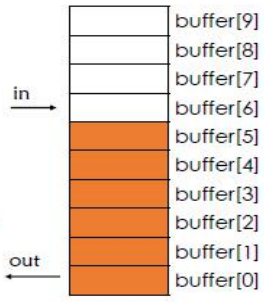
- 驻留在内存中的变量 buffer，由生产者和消费者共享

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

in（写）： 指向缓冲中的下一个空位
out（读）： 指向缓冲中的第一个满位
in == out： 表示缓冲为空
(in + 1)% BUFFER_SIZE == out
： 表示缓冲为满



【客户机-服务器系统通信】(3)

【套接字 Socket】套接字被定义为通信的断电，即端口。套接字由 IP 地址和端口号连接组成。连接由一堆套接字组成，二者通过套接字通讯。

【套接字分类（2)】**面向连接套接字**（TCP）（用于可靠传输）、**无连接套接字**（UDP）（用于广播、多媒体传送，一直发消息）

***套接字是与另一台计算机进程进行通信时的通讯工具，应用通过网络交互式必用套接字。其是最底层的，eg：HTTP 协议就是封装套接字后形成的端口号为 80 的协议

【远程过程调用（RPC)】RPC 抽象化了通过网络连接的进程之间过程调用，

【Stubs 存根】远程过程的代理，隐蔽了通信发生的细节，每个独立的远程过程都有一个存根。

【存根的主要工作】编组参数、解析参数

【远程方法调用（RMI)】RPC 的 Java 特性

【区别】

RMI 与 RPC 的不同：

1. RPC 调用远程**子程序或函数**，RMI 调用远程**对象的方法**
2. RPC 参数传递方式是**普通数据结构**，RMI 参数传递方式可以是**对象**

（他们都是调用函数只是传参不同）

【管道 Pipe】管道是进程之间进行通信的另一种方式。管道通信方式的中间介质是文件，通常称这种文件为管道文件

***两个进程利用管道文件进行通讯时，一个进程为写进程，另一个进程为读进程。

【管道分类】匿名管道（单向，用于亲缘进程)、命名管道（FIFO & 多入口的多元输入，但多元输入不能保证 FIFO）

【匿名管道】管道是半双工的，数据只能单向通信；需要双方通信时，需要建立起两个管道；只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）

【命名管道】可在同一台计算机的不同进程之间或在跨越一个网络的不同计算机的不同进程之间，支持可靠的、单向或双向的数据通信。

***命名管道谁都可以用，只要叫名字就行。匿名管道只有亲属可以用。

***mknod/mkfifo 可以给管道起名，是用于管理管道的文件，该文件的读写必须遵从 FIFO

***进程间由依赖关系，eg：A 的 output 是 B 的 input，可以用消息传递或者共享内存，但这种有依赖关系的往往用管道，现在管道不是很常用了

***一些命令行意思：

ls-l 即 dir：显示当前目录下所有列表

Ls-l>>pipe 1：即把输出的列表输入到管道 1
Cat myfile：显示文件内容
Who：显示用户名称和访问记录
Cat :显示
Spell：检查文件拼写错误：有错则输出错词，无错则无信息输出
Man XXXX：查 XXX 的意思

第四章 线程

【线程的概念】线程是运行在进程里的，比进程更小的运行单位。
***一个应用程序大部分有多种任务，即由多个线程组成
Eg：客户端：一个浏览器软件由显示内容、处理数据、拼写检查、网络处理等多个任务组成
网页服务器程序：由监听用户请求（注意：每个用户的请求不同）、负责处理用户请求的任务来组成
【进程与线程的关系】
1. 一个进程可以拥有单个或多个线程，线程之间共享**代码段、全局变量、打开的文件标识符、工作环境（包括当前目录、用户权限等）**。（线程共享该进程的内存空间）
2. 但是有关 CPU 的信息不可共享，包含 PC 值，寄存器和栈。
3. 每个线程是独立的调度对象。
4. 线程的运行是进程内部的执行轨迹，是进程内部执行指令的跟踪

【线程的创建 pthread_create()】相对创建进程（fork()整个复制父进程），创建线程只需要复制**栈和寄存器**
*****创建进程是重量级的操作过程，而且进程之间的上下文切换也是一个代价比较高的操作，因为 PCB 以及很多数据需要交换。**
*****创建线程是轻量级操作，只需复制栈和寄存器的内容，数据段和代码段的内容是该进程的所有线程共享的，且无需上下文切换**
【多线程的优点】**(4)**
1. **响应速度快**:即使部分线程发生阻塞，其他线程也可以继续运行
2. **资源共享**:共享所属进程的内存和资源
3. **经济**:创建和切换线程更经济
4. **充分利用多处理器体系结构**:多线程可以在多处理器上并行运行

【用户线程】用户线程受内核的支持。用户线程是由用户层的线程库来管理，无需内核管理。当前主要线程库有 3：POSIX Pthreads(LINUX)，Win32 threads(Windows)，Java threads(Java)
【内核线程】内核线程由内核管理，由内核进行维护和调度，当前通用操作系统都支持内核线程。
【映射】用户线程和内核线程之间的关系
***注意讨论该关系有四种模型，4 模型均是建立在 3 假设之下的：
1. 操作系统支持内核线程
2. 把内核线程看成是一个进程
3. 用户线程基于内核线程运行，即用户线程阻塞则内核阻塞，内核线程阻塞则用户线程也阻塞

【多线程模型】用户线程与内核线程之间的映射关系
【一对一模型】一个用户线程映射到一个内核线程的模型（Linux，Windows）

【一对一优点】1. 可以提供并发功能，可以在多个处理器上并行执行
2. 在一个线程执行阻塞系统调用时，可以调度另一个线程继续执行

【多对一模型】多个用户线程映射一个内核线程。线程由用户空间管理。一个用户线程阻塞，其他线程也阻塞。

【多对多模型】多个用户线程映射多个内核线程。一个用户线程阻塞，内核调度机制调度另一个线程执行。

【二级模型/混合模型】多对多模型的变种，允许用户线程绑定一个特定的内核线程（多对多+一对一）

【模型分析】

1. 一对一模型的唯一缺点是创建一个用户线程就需要创建一个内核线程，OS 负担重，占资源多。
2. 多对一模型的缺点是一个用户线程一旦被阻塞，其他线程也会阻塞，既无法提供并发功能（因为假设 3，用户进程阻塞内核也阻塞，而只有一个内核，所以内核阻塞其余用户进程也全阻塞，所有都阻塞了）
3. 多对多模型没有这两个缺点，但当一个线程执行阻塞系统调用时，需要提供调度机制来让内核调度另一个线程的执行。

【注意】

多对多中内核线程是有限的多个，而非无限多。所以要用少量内核线程来为多个用户线程服务就需要调度机制

【线程库】为程序员提供的、创建和管理线程的 API（3 方法+2 大类）

【线程库 2 种实现方法】：非嵌入到内核的方式 & 嵌入到内核的方式

1 .【非嵌入到内核的方式】是在用户空间中提供一个没有内核支持的库，此库的所有代码和数据结构都存在于用户空间中。

2.【嵌入到内核的方式】是操作系统直接支持的内核库，此时所有代码和数据结构都存在于内核中。

【3 种主要线程库】POSIX Pthreads(LINUX)，Win32 threads(Windos)，Java threads(Java)

【POSIX Pthreads】提供用户级或内核级的库

【Win32 threads】内核级库，嵌入到内核的方式

【Java threads】Java 线程 API 通常采用宿主系统上的线程库来实现

***JVM 和宿主操作系统：

**JVM 一般在操作系统之上实现，并隐藏了基本的操作系统实现细节。

**JVM 提供了一种一致的、抽象的环境以允许 Java 程序能在任何支持 JVM 的平台上运行

**JVM 的规范没有指明 Java 线程如何被映射到底层的操作系统，而是让特定的 JVM 实现来决定。操作系统是什么模型，JVM 就支持什么模型。

【创建进程的线程复制问题】线程的复制取决于创建新进程的目的。

1. 若 fork()的子进程与父进程做不同的工作，则需要调用 exec()，此时只复制调用 fork()的那个线程（因为 exec()要完全覆盖原线程，所以这样负担更小，更快）
2. 若 fork()的子进程与父进程做相同的工作，则不需调用 exec()，此时需要复制全部线程。（因为要复制原进程的所有内存空间，线程在内存中所以全都要复制）

【线程取消】指在目标线程完成任务之前终止之。

【目标线程】线程完成任务之前终止现成的任务，需要取消的线程一般称为目标线程。
***目标线程的取消可能发生“要取消的线程正在更新与其他线程所共享的数据”，所以有两种取消方式：

【异步取消】一个线程立即终止目标线程

【延迟取消】目标线程不断检查它是否应该终止，即自身是否处于安全**取消点**后取消。

【取消点】当一个线程认定为可以安全取消时，可以安全取消的这个点为取消点，即 pthread_testcancel()会被调用。

***线程取消请求的实际取消的行为依赖于线程的状态（创建线程时会为线程设置模式（3），否则默认延迟取消）：

1. Deferred Mode：默认状态为延迟取消
2. Off Mode：如果系统设置为禁用线程取消的话，线程取消会待定直到可以取消线程为止
3. Asynchronous Mode：异步模式，立即终止目标线程

【信号处理】信号是用来通知进程某个特定事件的发生，这需要操作系统提供一种内核和进程之间的通信机制。信号用于通知进程某个事件的发生。分为同步信号（内部）、异步信号（外部）两类型。

（信号两类型四发送方法，程序两类型三处理步骤）

【同步信号（内部信号）】进程本身事件产生的信号，如非法访问内存，除零等软中断

【异步信号（外部信号）】进程之外事件产生的信号，如按 CTRL+C 键等硬中断

【信号处理程序】信号处理程序是用来处理发生的事件。有默认信号处理程序和用户定义信号处理程序两类。对于多线程进程，需要决定信号传给哪些线程。

【信号处理程序三步骤】：

1. 由特定事件**产生**信号
2. 这个信号**传送**给进程
3. 信号处理程序**处理**相应信号

***事件产生信号，传送给进程需要考虑的问题：

1. 对于单线程无问题
2. 对于多线程要确定需要传送给哪个线程：【发送信号的方法】
   （1） 到信号所应用的线程
   （2） 到进程内每个线程
   （3） 到进程某个固定线程
   （4） 规定一个特定的线程来接收进程的所有信号

***发送信号的方法依赖于产生信号的类型：比如按下热键需要把信号发送给进程内的每个线程。比如 pthread_kill(pthread_t tid,int signal)函数可以把信号传送给指定的线程。

【线程池】在进程开始时创建一定数量的线程并放入线程池中等待唤醒。

【线程池优点】1.加速创建线程的过程。2.限制使用线程的数量，防止大量创建并发线程，有助于管理。

【调度程序激活 Upcall】一种内核向用户线程的通信机制。在多对多模型中的用户线程和内核线程之间设一种中间数据结构——**轻量级进程（LWP）**，程序可调度用户线程至 LWP。当某线程阻塞，内核通过 Upcall 告知用户进程，用户进程将该线程的 LMP 和内核线程分配给其他线程使用。

*多对多模型和二层模型中的用户线程和内核线程之间需要通信（即调度：少量内核线程为

多量用户线程服务），以便维持内核线程的适当数量。
*用户和内核线程之间设置一种中间数据结构——轻量级进程（LWP）（因为他是对线程调度，如果是对进程调度则是重量级操作 HWP）
*对于用户线程库，LWP 表现为一种应用程序可以调度用户线程的虚拟处理器
*每个 LWP 与一个内核线程相连，该内核线程被操作系统调度到物理处理器上运行
*如果内核线程阻塞，LWP 也阻塞，相连用户线程也会阻塞
【调度程序激活 Upcall】一种解决用户线程库和内核间通信的方法。
*内核提供一组 LWP 给应用程序，应用程序可调度用户线程到一个可用的 LWP 上
*调度程序激活提供内核和线程库之间的通信机制
*UPCALL 由具有 UPCALL 处理句柄的线程库处理，而且 UPCALL 处理句柄必须在虚拟处理器上运行
*这种通信机制允许应用程序保持适当的内核线程数量
【调度程序激活过程】当一个用户线程被阻塞，对应的内核线程也会被阻塞，该内核线程对应的 LWP 也阻塞，此时 UPCALL 通知应用程序阻塞情况，调度程序会把该阻塞的 LWP 取消，再建一个新的 LWP 来让该内核线程为新的用户线程服务。既保持了内核线程的数量，又只让少量的内核线程可以服务多个用户线程。
【总】**多对多模型和二层模型因为要用少量内核线程为多量用户线程服务，所以需要通信，因此在二者之间设置了一个中间数据结构叫轻量级进程 LWP，用于在内核线程和用户线程之间服务，该服务需要一种调度机制，即调度程序激活。**
*****【创建进程和线程的区别】*****
1. **创建进程是重量级的操作，因为要把父进程的全部内存都复制过来，而且进程之间的上下文切换也是一个代价比较高的操作，因为 PCB 以及很多数据需要交换。**
2. **创建线程是轻量级的操作，只需要复制 PC 和寄存器的内容，无需上下文切换。**

第五章 CPU 调度

【CPU 调度基本概念】一个进程的执行由 **CPU 区间**（CPU 执行）和 **I/O 区间**（I/O 等待）组成。调度的目的是最大化 CPU 使用率。
***进程在执行的过程中，不断在这两个状态之间进行切换。
***进程一般由大量的短 CPU 区间（<8ms）和少量的长 CPU 区间组成。（I/O 为主的程序短 CPU 区间多，CPU 为主的程序长 CPU 区间少，所以怎么说都是短多长少）、

每当 CPU 空闲时，调度程序（**短期调度程序**）从**就绪队列**中选择一个进程，并为之分配 CPU：
【CPU 调度的决策可能发生/CPU 调度程序可能运行的环节】当一个进程：
1. 从运行状态转换成等待状态（如发生 I/O）
2. 从运行状态转换成就绪状态（如发生中断）
3. 从等待状态转换成就绪状态（I/O 完成）
4. 进程终止
5. 进程进入就绪队列

   ***当进程发生这五种变化时，会激活调度程序，然后该程序会在就绪队列中选择一个进程位置分配 CPU**

【非抢占调度】一旦把 CPU 分配给一个进程，直到该进程结束之前，不能把 CPU 分配给其他进程。调度时机：运行中的进程转变为等待或终止状态。

【抢占调度】进程在执行过程中可以被其他进程抢占 CPU 使用权。调度时机（包含上述时机）：一个进程（高优先级）从等待转变为就绪状态、运行中的进程发生中断转变为就绪态。

【分派程序】分派程序是一个模块，用来将 CPU 的控制权交给由短期调度程序选择的进程。

【分派程序功能】

1. 切换上下文
2. 切换到用户模式
3. 跳转到用户程序的合适位置，以重新启动程序

【分派延迟】指分派程序（调度程序）停止一个进程而启动另一个进程所花时间。分派延迟包含上下文切换时间。

*****分派延迟包括上下文切换，因为上下文切换只是进程的切换，而模式的切换等也属于分派延迟**

【调度准则】衡量调度算法考虑的要素：

1）CPU 使用率：需要使 CPU 尽可能忙

2）吞吐量：单位时间内完成的**进程**数（表示的公平性：比如 CPU 使用率很高但是一直只是为一个进程服务，这样是不好的，应该为更多的进程服务，也就是公平对待每个进程）

3）周转时间：从进程提交至进程完成的总时间。周转时间为所有时间段之和，包括等待进入内存，在就绪队列中等待，在 CPU 上执行和 I/O 执行。

【进程提交/到达 arrive/release】进程进入 **ready 状态（是状态，而不是进入就绪队列）**

4）等待时间：进程在就绪队列中等待的时间

5）响应时间：从提交请求（即进入就绪队列）到**产生第一响应（即第一次分配 CPU 给它）**的时间（**注意！**是开始响应所需要的时间，而不是**输出**响应所需要的时间）（交互系统）

【调度算法】

【先到先服务 FCFS】非抢占调度。就绪队列为 FIFO。

【护航效果】所有其他小进程都等待一个大进程释放 CPU，即短进程跟在长进程后面。（多个进程等待大进程执行完毕）使平均等待时间较长。

【大进程】CPU 区间长，I/O 区间短

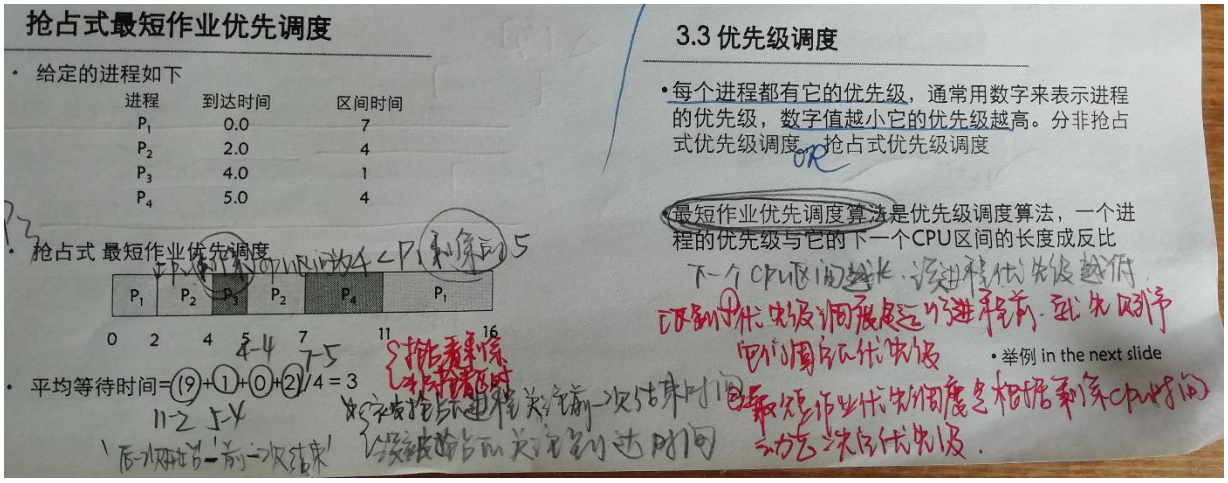
【小进程】I/O 区间短，CPU 区间长

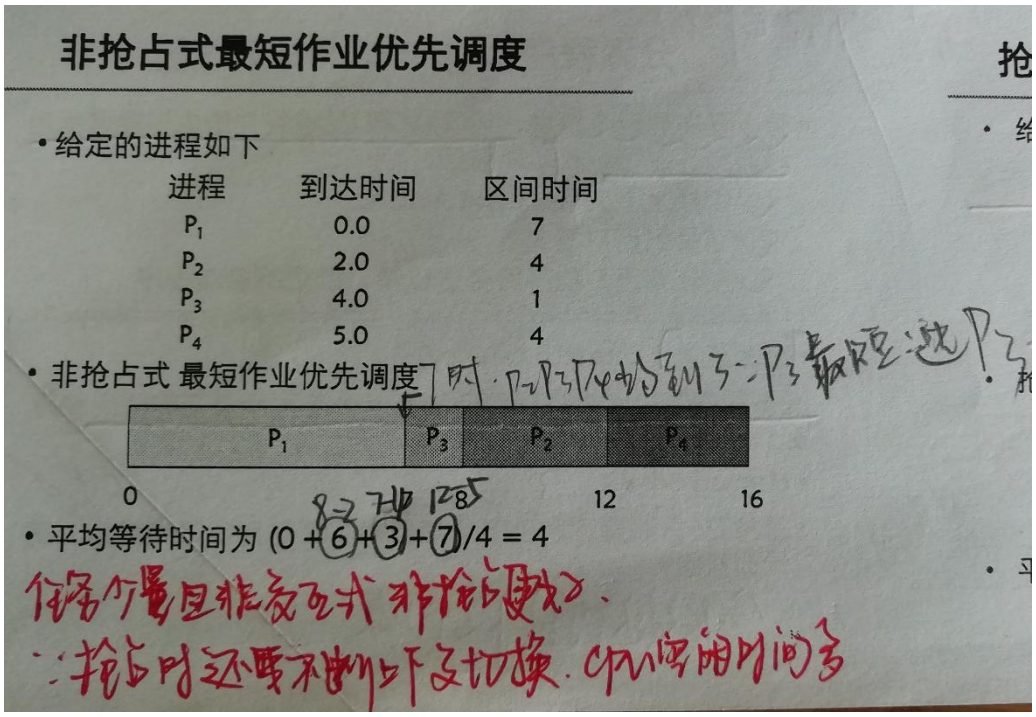
***大进程运行 CPU 区间时，其他短进程很快完成 I/O 操作并移到就绪队列等待分配 CPU（I/O 设备空闲）

***大进程释放 CPU 后，移到 I/O 队列，其他短进程很快完成 CPU 操作并移到 I/O 队列（CPU 空闲）

【最短作业优先调度 SJF **（最优）**】抢占&非抢占调度。选择下一个 CPU 区间（即剩余 CPU 区间）长度最短的进程，而不是整个进程区间长度。对于就绪队列中的进程，下一个 CPU 区间越短，优先级越高。此算法平均等待时间最低。（抢占调度时称最短剩余时间优先调度）

【对比】





【优先级调度】抢占&非抢占调度。每个进程都有它的优先级，用数字表示，数字值越小优先级越高。按照规定优先级调度

【无穷阻塞（**饥饿**）】低优先级进程无穷等待，无法运行。

【**老化**】用于解决无穷阻塞：逐渐增加系统中等待时间长的进程的优先级

【注意】老化过程中，低优先级的进程被运行后下一次就不会再老化它，其再次到达（release）的时候，优先级不是老化后的，而仍然保持原来的优先级。

【轮转法调度 RR】抢占调度。就绪队列为 FIFO。用于**分时系统**。是 FCFS 调度算法的变种，给每个进程分配一个时间片，时间片一般为 10-100ms。系统每隔一个时间片发出一个时钟中断，并调度**另一个**进程执行。每个进程只能运行给定的时间片并释放 CPU，并被放入到就绪队列。如果进程在给定的时间片内提前结束，就会发生中断并调度另一个进程。每次给每个进程分配不超过一个**时间片**。

【对比】较 FCFS 周转时间长，响应时间短。若时间片过大，则等同于 FCFS；

若时间片过小，则为**处理器共享**（但上下文切换开销太高）。

【周转时间&时间片】周转时间依赖于时间片的大小。一般时间片增加，平均周转时间会减少。但会有周转时间并未随时间片大小的增加而改善的不正常现象，但时间片大于 80%的 CPU 区间往往不会出现该不正常现象。

【多级队列调度】将就绪队列拆分成多个独立队列。队列内采用自己的调度算法，队列之间也有调度：固定优先级抢占调度：前台队列比后台队列优先级高//队列之间划分时间片：。每个队列都有给定的 CPU 时间，这个时间可用于调度队列内的进程。（最常用：前台 RR 轮转法，后台 FCFS 先到先服务）

【多级反馈队列调度】多级队列，每个队列都有自己的调度机制。**允许进程在队列中移动。**占用过多 CPU 的进程会被移至低优先级队列。（过程：高优先级进程在指定时间片内未完成，则会被放到低一级的队列末尾运行，以此类推，到最后一个队列还无法完成，则按 FCFS 调度）（**反馈**体现在**时间片**上）

【调度程序参数】：队列数量、每个队列调度算法、确定何时升级降级的方法、确定进程在需要服务时该进入哪个队列的方法

*****【对比】***多级队列调度&多级反馈队列调度都是多级队列，每个队列都有自己的调度机制。但是多级队列调度不允许进程在队列中移动，而多级反馈允许。**

【线程调度】操作系统调度内核线程，线程库调度用户线程。

【进程竞争范围 PCS】线程库调度**用户线程**到一个有效的 LWP，发生在进程内部。

【系统竞争范围 SCS】操作系统调度**内核线程**到 CPU 上允许，发生在整个系统所有线程之间。（一对一模型只有 SCS：Win XP）

【多处理器调度】

【多处理器分类】同构处理器：处理器功能相同。异构处理器：处理器功能不同。

【多处理器系统分类】非对称+对称+多核

【非对称多处理】只有一个处理器进行调度，允许处理器之间的移动。

【对称多处理 SMP】每个处理器有自己的调度算法，不允许处理器之间的移动。

【处理器亲和性】对称多处理器系统避免进程在处理器之间移动（一个进程要有一种对其运行所在处理器的亲和性）

【软亲和性】允许进程在处理器之间的移动

【硬亲和性】不允许进程在处理器之间的移动

【负载平衡】设法将工作负载平均地分配到 SMP 系统中的所有处理器上。

【推 Push】负担重地 CPU 把进程推给闲 CPU

【拉 Pull】闲 CPU 从其他 CPU 抢进程运行

【注意】推和拉只是对于 CPU 监控的策略不同。如果监控的是闲 CPU 则拉，监控的是忙 CPU 则推，所以二者不相互排斥。**（闲拉忙推）**

【竞争范围】

【进程竞争范围 PCS（Local scheduling）】在采用多对多或多对一模型的系统上，线程库调度用户线程到一个有效的 LWP（即用户线程映射内核线程）。根据优先级完成，一般是程序员给定。竞争发生在相同进程的线程之间。

【系统竞争范围 SCS（Global scheduling）】在采用一对一模型的系统上，系统将内核线程调度到有效的物理处理器上。竞争发生在系统的所有线程之间。（Windows XP）

*****而这是根据竞争范围划分，在进程内竞争就是 PCS，在系统内竞争就是 SCS。而二者的**

竞争均发生在线程之间，只不过一个是相同进程的所有线程，另一个是系统的所有线程。
*****Linux 不支持进程竞争范围 PCS**

【实时任务】有截止时间的进程
***实时系统中的进程具有截止时间要求，进程必须在给定的截止时间内完成执行。
***实时系统必须满足两个条件：1.计算正确 2.有截止时间限制
【软实时系统】任务具有软截止时间。可以适当地超过截止时间结束任务。（音频、视频系统等）
【硬实时系统】具有硬截止时间。必须在截止时间内结束任务。（军用设备、医疗设备等）
【实时调度任务模型】
【Periodic Tasks】周期性任务：任务周期性到达
【Aperiodic Tasks】非周期性任务：任务到达时间不确定（即任务随时可能到达），无截止时间
【Sporadic Tasks】不定时任务：任务到达时间不确定，但有截止时间，该时间为给定的时间片
【实时调度算法】
【Rate Monotonic】频率优先（最常用）
【Earliest Deadline First】最短作业优先（最优）
【Earliest Deadline Zero Laxity】EDZL 零拖延时间作业优先（再不运行就死定了！）

*****【补充】*****
【多核编程】
【编程挑战】
【识别任务】这涉及分析应用程序，查找区域以便分为独立的、并发的任务。在理想情况下，任务是相互独立的，因此可以在多核上并行运行。
【平衡】在识别可以并行运行任务时，程序员还应确保任务执行同等价值的工作。在有些情况下，有的任务与其他任务相比，可能对整个任务的贡献并不多；采用单独核来执行这个任务就不值得了。
【数据分割】正如应用程序要分为单独任务，由任务访问和操作的数据也应划分以便运行在单独的核上。
【数据依赖】任务访问的数据必须分析多个任务之间的依赖关系。当一个任务依赖于另一个任务的数据时，程序员必须确保任务执行时同步的，以适应数据依赖性。
【测试与调试】当一个程序并行运行于多核时，许多不同的执行路径时可能的。测试与调试这样的并发程序比测试与调试单线程的应用程序更加困难。
【并行类型】
【数据并行】将数据分布于多个计算核上，并在每个核上执行相同操作。（比如求和，可以一个核求 1~N/2 的和，另一个核求 N/2~N 的和，这样更快）
【任务并行】设计将任务（线程）分配到多个计算核。每个线程都执行一个独特的操作。不同线程可以操作相同的数据或不同的数据。
*****【区别】*****
【并发性】单处理器时，CPU 调度器通过快速切换系统内的进程，以便允许每个进程取得进展，从而提供并行假象，这些进程是并发运行，而非并行运行。
【并行性】并行系统可以同时执行多个任务，比如：多核（多个计算核在单个 CPU 芯片上）

&多处理器（多个计算核在多个 CPU 芯片上）均可达到并行。
【对比】
***并行系统可以同时执行多个任务；并发系统支持多个任务，允许所有任务都能取得进展。
***没有并行，并发也是可能的。

第六章 进程同步

【竞争条件】多个进程并发访问和操作同一数据，且执行结果与访问发生的特定顺序有关。竞争条件的结果是不可预测的。

避免发生竞争条件的关键问题：确保操作共享数据的代码段的执行同步（互斥运行），不能让两个以上的进程同时运行**操作共享数据的代码段（即临界区）**

【临界区 Critical Section】多个进程同时操作共享数据时，每个进程拥有操作共享数据的代码段(程序段)，这个代码段称为临界区（比如：共享变量、共享表、共享文件等）当一个进程进入临界区，没有其他进程可被允许在临界区内执行。

【临界区问题/竞争问题】指一个进程协作协议，【临界区问题的通用结构】分为进入区、临界区、退出区、剩余区

【解决临界区问题需要满足的条件】

【互斥】当一个进程进入临界区，其他进程不能进入临界区。

【前进】若没有进程执行临界区，则必须确保一个进程进入临界区

【有限等待】一个进程从请求进入临界区，直到该请求被允许，必须有限等待

【Peterson 算法】基于软件的适用于**两个进程**的临界区问题解决方案。（前提加载和存储为原子指令）

【变量】1）int turn：表示哪个进程可以进入临界区

2）boolean flag[2]：表示哪个进程想进入临界区（true）

【算法描述】i=当前进程编号；j=另一个进程编号

```
do {
    flag[i]=true;
    turn=j;
    while (flag[j] && turn==j);
        //临界区
    flag[i]=false;
        //剩余区
}while(true);
```

【硬件同步】现代计算机提供原子指令（硬件指令，以**原子地**执行某些指令）：

***原子指令即在执行该指令时绝对不会被中断的指令，其用于解决临界区问题。

【TestAndSet()】检查和设置字的内容。将参数 target 置 true，并将原值作为返回值返回。

【Swap()】交换两个字的内容参数的值。

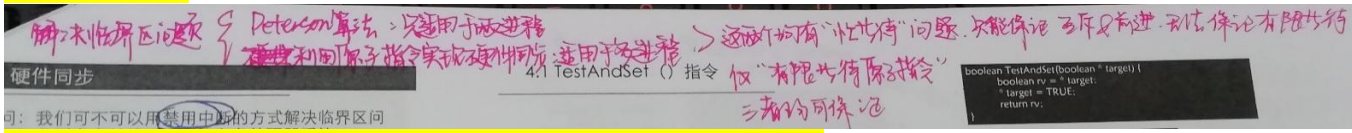
【互斥实现】

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

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

【有限等待原子指令】为实现有限等待，声明布尔变量 waiting[i]来表示是否想进入临界区。当某个进程退出临界区时，向下遍历一次 waiting[i]来引导下一个进程进入临界区。

【总结】



***有限等待原子指令的过程非常重要会出大题，所以见纸质版 PPT

【对比】

临界区问题的三个解决方案（Peterson 算法，硬件同步，有限等待原子指令）均是单资源的同步操作（因为只有一个临界区），如果想解决多资源的同步操作只能通过信号量

【信号量】用于多资源共享。利用一个共享信号量和**标准原子操作** wait（-1）和 signal（+1）来管理共享资源的数量。

【二进制信号量/互斥锁】适用于单资源的共享（资源数量 mutex=1）（acquire-1、release+1），信号量的值 S 只能为 0 获 1.

【计数信号量】适用于多资源共享，共享资源数量为 n。wait(n)为减，signal(n)为加，n=0 时表示所有资源都被占用。**可以适用于优先约束（即必须按优先级顺序来运行的非抢占的）。**

***信号量的实现关键**是保障 wait()和 signal()操作的原子执行，即必须保障没有两个进程能同时对同一信号量执行 waiy()和 signal()操作。

【保障方法】

1. 单处理器环境下，禁止中断
2. 多处理器环境下，禁止每个处理器的中断。（困难又危险）

【忙等待】当一个进程位于临界区，其他进程必须执行不必要的循环操作，浪费 CPU。这种类型的信号量称为**【自旋锁】**。

【无忙等待的信号量实现】让忙等待的进程挂起（blocking），可以进入临界区时，让进程重新启动（wakeup）将等待中的进程加入就绪队列，并**挂起**（block），当可以进入临界区时**重新执行**（wakeup）。

***选择忙等待 or 无忙等待信号量的机制取决于 **上下文切换&临界区长度**，综合二者折中选择。

【挂起】进程从运行状态转换成等待状态

【重启】进程从等待状态转换成就绪状态

***无忙等待信号量在代表资源数量的整数值的基础上又多了一个保存等待状态进程的链表

【死锁】多个进程共享多个资源（使用多个信号量）时可能发生。（贪）

【饥饿】无限期等待，即进程在信号量内无限期等待。**当就绪队列（信号量的链表）LIFO**

（last in first out）实现时可能发生。

【有限缓冲问题】即生产者-消费者问题中的**有限缓冲、读写互斥、写写互斥、读读互斥**问题。

【有限缓冲】1）信号量 empty 表示空缓冲项个数（为 0 时不能生产）

2）信号量 full 表示满缓冲项个数（为 0 时不能消费）

【读写互斥】3）二进制信号量 mutex 实现生产与消费互斥

【读者-写者问题】多写者、多读者的**写写互斥、读写互斥、记录读者数量**问题。

【写写、读写互斥】1）信号量 wrt（读者写者共享，初始化 1）实现写操作与其他操作互斥

【记录读者数量】2）信号量 mutex 实现 readcount（初始化 0）更新的互斥

【哲学家进餐问题】多进程多资源共享问题，出现死锁。

【管程】封装数据和一组互斥操作，并保证同时最多只有一个线程执行管程代码。在此问题中，哲学家 i（thinking，hungry，eating）只有在其相邻两个邻居都不进餐时才能拿起筷子。

【管程的实现 2 方法】1.利用抽象化概念 2.利用信号量

【管程的概念】管程是一种用于多线程互斥访问共享资源的程序结构。采用面向对象方法，简化了进程间的同步控制；任意时刻最多只有一个线程执行管程代码；正在管程中的线程可临时放弃管程的互斥访问，等待事件出现时恢复。

【引入管程的原因】1. 把分散在各个进程中的临界区集中起来进行管理

2.防止进程有意或无意的违法同步操作（交换 wait()和 signal（）操作顺序；用 wait()替代 signal()操作；省略了 wait()或 signal()操作）

3.便于用高级语言来书写程序，便于程序正确性验证。

【管程的使用】1.在对象/模块中，收集相关共享数据 2.定义访问共享数据的方法

【管程的组成】1. 一个锁：控制管程代码的互斥访问

2.零或多个条件变量：管理共享数据的并发访问（条件变量的数目即共享资源/数据的数量）

3.一个条件变量对应于一个等待队列，每个条件变量仅有 wait()和 signal()操作

【条件变量】当调用管程过程的进程无法运行时，用于阻塞进程的一种信号量

***当一个管程过程发现无法继续时，它在某些条件变量 condition 上执行 wait 操作，这个动作引起调用进程阻塞，即挂起进程

***另一个进程可以通过对其伙伴在等待的同一个条件变量 condition 上执行同步 signal 操作来唤醒等待的进程，即重启进程

******进程请求资源后两种情况：1.若无资源可用，则把其挂起，放到该资源对应的等待队列 2.若有资源可用，则把该对应等待队列的资源分配给该进程

【哲学家问题的实现】【利用信号量实现管程机制】见 PPT

第七章 死锁

【死锁】当一组进程中每个进程都在等待一个事件，而这一事件只能由这一组进程的另一个进程引起，则这组进程就处于死锁状态。

【死锁的必要条件】以下四个条件**全部**满足即死锁

1）互斥：一个资源同时只能被一个进程占有

2）占有并等待：一个进程必须占有至少一个资源，并请求/等待另一资源

3）非抢占：资源不能被抢占

4）循环等待：Pi 等待的资源被 Pi+1 占用

【资源分配图】**申请边**：进程→资源；**分配边**：资源→进程

资源分配图

资源分配图由节点集合 V 和一个边集合 E 组成

1. 节点集合V 分为（1）进程集合P 和（2）资源集合 R
   - 进程节点集合： $P = \{P_1, P_2, \cdots, P_n\}$
   - 资源节点集合： $R = \{R_1, R_2, \cdots, R_m\}$

2. 边集合E

- $P_i \rightarrow R_j$

：表示进程 Pi已经申请使用资源类型 R_j 的一个实例

- $R_j \rightarrow P_i$

：表示资源类型 R_j 的一个实例已经分配给进程P_i

【环与死锁】若资源分配图无环，则一定不会死锁。若有环，可能发生死锁（若每个资源只有一个实例，则一定发生死锁）。

【预防死锁】即打破四个必要条件中的至少一个

【互斥】无法打破

【占有并等待】进程执行前提前占有所有资源；进程在没有资源时才可以申请资源

缺点：资源利用率低、可能发生饥饿

【非抢占】若一个进程占有并等待，则其现有资源可被抢占。

【循环等待】为每个资源类型分配一个唯一的整数，每个进程按递增顺序申请资源。确定一个资源申请顺序，即给每种资源编号。任何进程在申请资源时，必须先释放号码比该资源大的所有资源。

【避免死锁】确保系统不进入安全状态。这是一种动态的方法，它根据进程申请资源的**附加信息**决定是否申请资源。即通过进程资源申请信息和一定的算法，以保证死锁不发生

【要掌握的附加信息】1.当前可用资源 2.已分配给每个进程的资源 3.每个进程将来要申请获释放的资源 4.每个进程可能申请的每种资源类型实例的需求

【避免死锁算法】

【资源分配图算法】适用于每种资源只有**单个实例**。资源分配图新增**需求边**：进程**-->**资源，表示进程可能将在未来某时刻申请此资源。当该进程申请该资源时，将需求边变成申请边（虚线变成实线）

【资源分配图算法规则】当一个进程申请资源时，将申请边变为分配边后若无环（包含需求边），则允许申请；否则拒绝申请。（因为单实例只要有环必死锁）

【银行家算法】适用于每种资源有**多个实例**。

【数据结构】1）available[i]表示资源 i **可分配的个数**

2）max[i][j]表示进程 i 需要的资源 j 的总数

3）allocation[i][j]表示进程 i **已经占有的资源** j 的个数

4）need[i][j]表示进程 i **仍需要的资源** j 的个数

【安全性算法】确定计算机系统是否处于安全状态的算法。对于当前可用资源 available，若能找到一个进程排列，使得所有进程能依次在前一个进程释放完所占资源（available+=allocation）后能够取得足够数量的资源（need≤available），则为安全状态；若找不到这样的排列，则为不安全状态。

***【注意】这个排列要从上往下，直到到底了再从头开始找。这样的顺序一定不会出错，否则老师可能扣分。

【资源请求算法（先）】判断是否可安全允许请求的算法。当一个进程申请资源时，若申请合法（≤need）且有足够的资源可供分配，则对分配后的新状态进行安全性检测，若安全则允许；否则拒绝申请。

***资源请求算法中调用了安全性算法来判断每个资源请求是否会引起死锁

【死锁检测】

【等待图算法】适用于每种资源只有**单个实例**。资源分配图删除所有资源节点，得到**等待图**。**当且仅当**等待图中有环，系统中存在死锁。

【检测算法】适用于每种资源有**多个实例**。**过程同银行家算法中的安全性检测，但检测算法检测的是当前系统，安全性检测用于预测。**

***检测算法过程和银行家中的安全性算法过程相同，不过检测算法把 Need 变成了 Request，并且初始化不同，检测算法多了对 Finish[i]的选择性初始化，即如果已经分配资源 Allocation ≠0 则 Finish[i]=false，否则 Finish[i]=true。而安全性算法是所有 Finish[i]都初始化为 false。

【检测时机】取决于**死锁发生的频率**和**死锁发生时会受影响的进程数量**。

1. 若经常发生死锁，那么每次资源请求时都调用检测算法。
2. 若只有当某进程提出请求且得不到满足时才会死锁，那么每次资源请求不被允许时调用检测算法。

【死锁恢复】（2+2）

【进程终止】终止所有死锁进程；一次终止**一个**进程直到取消死锁循环为止（考虑 6 点）。

1）进程的优先级
2）进程已计算了多久，进程在完成指定任务之前还需要多久
3）进程使用了多少类型的资源（占类型多的先终止掉）
4）进程需要多少资源以完成（将来需要多的先终止）
5）多少资源需要被终止
6）进程是交互的还是批处理的（交互的不终止，先终止批处理的，因为交互的与用户有信息交流）

【资源抢占】通过抢占资源以取消死锁，逐步从进程中抢占资源给其他进程使用，直到死锁被打破为止。资源抢占有三个问题要处理：

1）选择一个牺牲品：抢占哪些资源和哪个进程
2）回滚：必须把不能正常运行的进程回滚到安全状态，以便重启进程
3）防止饥饿：避免同一个进程总成为牺牲品

第八章 内存管理

【进程使用内存空间的界定】

为确保进程只访问合法地址范围，一个进程使用的内存地址范围是由一对基地址寄存器（base register）和界限地址寄存器（limit register）来定义。

为了确保进程争产运行必须保护内存：1.确保系统区域不被用户进程访问 2.确保用户进程不被其他进程访问（用户进程若像访问系统区域只能通过接口从而调用系统调用来访问，而不能直接访问）

【基地址】最小的合法物理内存地址，存于**基地址寄存器**。

【界限地址】地址范围大小，存于**界限地址寄存器**。

【内存保护】使用基地址和界限地址保护内存，确保系统区域、用户进程不被非法访问。

【地址绑定】是逻辑地址到物理地址的映射

【逻辑地址/相对地址/符号地址】用户程序在经过编译后形成的目标代码，其首地址（基地址）一般为 0，其余指令中的地址都是相对于首地址来编址。不能用逻辑地址直接寻址。

【物理地址/内存地址/绝对地址】把内存分成很多个大小相等的存储单元，每个单元给一个编号，这个编号称为物理地址。物理地址可以直接寻址。

通常将**指令和数据**绑定到内存地址，绑定的时间有三种情况：

【编译时绑定】编译时就知道进程在内存中的地址，编译生成**绝对代码**。（Ms-dos 每个时刻只有一个进程运行，所以其在编译的时候就会发生地址绑定）

【加载时绑定】编译时不知道进程驻留内存地址，编译器必须生成**可重定位代码（relocatable code）**，最后绑定延迟到加载时才进行。

【运行时绑定】进程执行时允许在内存中移动（Swap in/Swap out），那么绑定必须延迟到执行时才进行，需要特定硬件支持（MMU：内存管理单元<里面包含重定位寄存器，提供重定位基地址，即最小的物理地址值>）

【内存管理单元】是映射虚拟地址为物理地址的硬件设备。用户进程所生成的地址在交送内存之前都会加上重定位寄存器的值。用户程序看不到真正的物理地址。

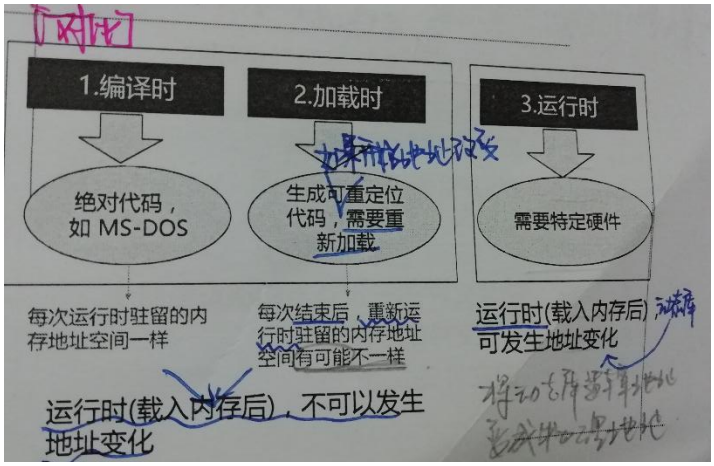
逻辑地址+重定位基地址=物理地址

【内存管理单元】进行地址绑定的硬件设备，使用户看不到真正的物理地址。逻辑地址+基地址**（重定位寄存器）**=物理地址。

*****【注意】***编译时绑定和加载时绑定都是针对于程序来说的。**

运行时绑定是针对动态库来说的。当该程序需要调用动态库时，调用动态库载入内存时会针对动态库发生地址绑定，而不是针对程序，因为程序要运行就已经载入内存了，所以程序依然是加载时地址绑定的。

*****【对比】*****



【动态加载】子程序只有在被调用时才被载入至内存，即加载延迟到运行时。

【优点】1. 内存使用率高：不使用的程序不会载入到内存

2.适合用户用大量代码来操作不常发生事件

3. 不需要操作系统的特别支持，由程序员来设计

【静态链接】加载程序合并到二进制程序镜像中，一直驻留在内存。链接时将库的内容加入至可执行程序

【动态链接】链接延迟到运行时。

***存根 stub，这个小程序，用来定位适当的内存驻留库程序，或如果该程序不在内存时应如何装入库。

***stub 首先检查所需子程序是否在内存中，如果不在就将子程序装入内存。Stub 会用子程序**地址**来**替换**自己，并开始子程序。

***动态链接适用于**系统库**（比如：语言库），由操作系统管理

可执行文件只存链接地址

*****【对比】***静态库&动态库**

1. **静态库：从程序开始到结束常驻在运行库中的代码（比如 printf）**
   动态库：程序运行时该代码并不一直在库中，需要时才载入，结束时释放
2. **静态库可移植性好：一个程序生成可执行文件后拿到另一个没有该运行库的环境中仍是可以运行的，因为该程序的静态库已经在程序中了。但是动态库就不可以。**

*****【对比】***动态链接&动态加载**

1. **动态链接和动态加载是载入动态库的两种方式，都需要时加载到内存**
2. **动态链接是程序启动时建立了链接（即只有链接地址），需要时载入内存。由操作系统决定**
   动态加载是通过程序的方法来控制加载（dlopen）。由程序员决定。

【交换 Swap】进程可以暂时从内存交换到备份存储上（通常是快速磁盘）（Swap out），当需要再次执行时再调回内存（Swap in）。交换时间与交换量成正比。

【优先级调度算法】低优先级交换出（滚出 roll out），高优先级交换进（滚入 roll in）

【交换时间/转移时间】转移时间与交换内存空间量成正比

Q1：交换出的进程调回时应调回到哪个内存空间根据动态绑定时间决定

A1：若地址为编译时或加载时绑定，则必须交换回原来的位置。若是运行时绑定，可以发生地址变化。

***编译时&加载时绑定，每次绑定的逻辑地址&物理地址都不变，都和以前绑定的逻辑地址&物理地址相同。而运行时绑定，每次的逻辑地址&物理地址都可能不同。（因为其运行的时候换了一块帧，那么该帧对应的页和原来的就不同，所以逻辑地址也会不同）

Q2：何时交换？

A2：1.内存空间不够时 swap out，把现在要执行的程序 swap in

2.内存空间足够，但为了有效使用进程而不把全部进程载入内存，而只载入在一个时间片内可以运行完毕的进程量。

PS：Unix 中，虚拟内存大小一般是实际物理内存大小的一半

【进程的内存分配方式】

【连续分配】每个进程位于一个连续的内存区域

【多分区方法】内存分成固定大小的分区，每个分区只能容纳一个进程（所以多道程序的程度会受分区数限制）

【划分区的方法】a)分区大小相同：只适合于多个相同程序的并发执行，否则会缺乏灵活性

b)分区大小不同：多个小分区，适量中分区，少量大分区。根据程序大小，分配当前空闲的，适当大小的分区。

【可变分区方法】内存空间视为**孔**，操作系统用**表**来记录已用和未用内存。当进程载入时：

【首次适应】分配足够运行进程的第一个找到的孔

【最佳适应】分配足够运行进程的最小的孔

【最差适应】分配最大的孔

【碎片问题】**内部碎片**（多分区方法，区外产生的碎片）；**外部碎片**（可变分区方法，区内产生的碎片）。

【碎片问题解决方法】

1.【紧缩】移动内存内容，把所有的空闲空间合并成一整块。只适用于**动态重定位**的时候。开销大。

2.允许物理地址空间为非连续：分页/分段

【分页分配】允许进程的物理空间非连续。产生内部碎片。

【页和帧】页——逻辑内存划分成固定大小的块；帧——物理内存划分成固定大小的块**（帧物页逻）**

*****以页为单位分配内存**

*****一个页对应一个帧，所以若程序大小为 n 页，则需要有 n 个帧来存放它。但是帧是不必连续的内存空间。**

*****需要知道两个信息：1.（哪可分配）掌握空闲空间（空闲帧）信息→空闲帧表（对空闲帧管理）**

2.（分配到哪）为了把逻辑地址转变为物理地址→页表（对已用帧管理）

【地址变换】**逻辑地址**分为**页号 p+页偏移 d**，页号是页表的索引，相应页表中存储着该页所在物理内存的基地址。页偏移结合基地址形成物理地址。页号通过**页表**映射为**帧号（页表每个条目是跟页号所对应的帧号，帧号即要映射到的那块内存的块号，该块内存的范围是基地址~基地址+d）**，页偏移不变。

【空闲帧表】用来记录管理空闲帧以便分配

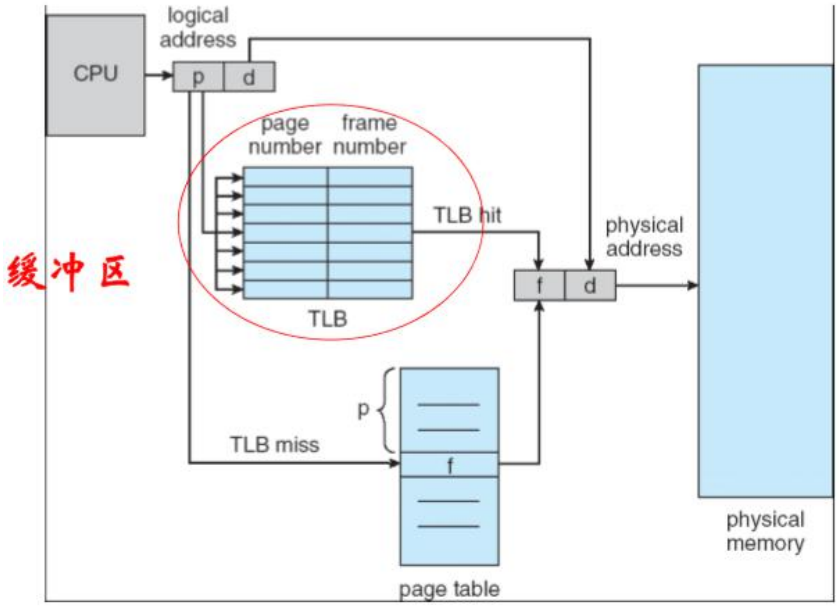
【页表的实现】1.用一组专用寄存器来保存（贵且容量小）

2.将页表放入内存（需要**页表基寄存器&页表长寄存器**）

【页表放入内存问题】每次有新进程待分配内存，需要两次访存以确定其放入内存的位置，第一次查页表，第二次访问地址（即确定它要放到内存的哪块地址范围）。这会导致数据访问速度慢。

【解决】**地址转换旁观缓冲 TLB**（键<标签>，值）。TLB 的工作原理同缓存：当进程要放入内存的时候，先查 TLB，若有则直接映射到物理地址。否则访问内存中的页表，读取数据并将其放到 TLB，方便下次快速查询。和缓存一样需要替换机制：最常用的是命中率(使用频率)高的留下，低的替换出去。

【页缓冲 TLB】设专用 Cache 缓存页表（地址转换旁观缓冲），用于解决查页表访存慢的问题。



【内存保护】与每个帧关联一个保护位，建立与也表中的每一条目相关联的**有效-无效位**，确保访问合法。

【有效位 v】表示相关的**页**在进程的**逻辑地址**空间内（合法）

【无效位 i】表示相关的**页**不在进程的**逻辑地址**空间内（非法）

【分页优先】可以通过**共享页表**实现代码共享。

【可重入代码】可以共享的代码，每个进程共享代码段的（页）逻辑地址相同

【私有代码和数据】针对私有代码和数据的（页）逻辑地址不同

【页表结构/实现页表的方式】映射过程中，分页方式需要**系统**维护页表信息

【层次结构】逻辑地址编址设为多层次，进行多页表多级映射（二级页表索引）

【特点】1.可以实现页表的不连续存储 2.节约内存空间

【哈希结构】逻辑地址：【虚拟页码（哈希值），偏移量】，哈希页表每一条目用链表实现，链表的每个元素包括：【虚拟页码，所映射的帧号，下一个元素指针】。根据虚拟页码（哈希值）找到匹配的条目，并在链表里找到相对应的帧号。更具体地说是：通过页号转换到哈希表中，用链表指针找到相对应的帧号，并用帧号和页偏移形成物理地址。

【反向页表】整个系统只有一个页表，减少页表消耗的内存空间。页表对于每个真正的内存页或帧才建立一个条目。需要记录每页进程 PID。逻辑地址：【进程标识符 PID，页号 p，页偏移 d】（地址信息：p+d 确定，进程信息：pid 确定），反向页表每个条目：【PID，页号 p】。

【分段分配】以用户视角去管理内存，一个程序有多个逻辑段组成（主程序、函数、方法、对象、堆、栈、局部变量、全局变量等）将内存看作长度不一的段的集合。通过段表（段基地址、界限）实现索引。逻辑地址：【分段号，偏移】，段表条目：【界限地址，段基地址】

*****段必须连续分配，所以会产生碎片问题。Linux 用的是分页，分页是最常用的。**

*****【区别】*****

1. **分段最后映射到物理地址加的是段界限（界限地址，即段偏移），其逻辑地址的偏移是其在段内的偏移。**
   除了分段的所有结构最后映射到物理地址加的都是逻辑地址中的那个偏移量。
2. **分段的段表中是先界限地址后段基地址，而且他的都是先基地址后偏移量。**

*****【总结】*****

1. **固定大小分配内存单元（连续分配的多分区&单页）会产生内部碎片**

2. 可变大小分配单元（连续分配的可变分区&分段）会产生外部碎片。

*****页号=页码=虚拟页码，这几个说法意思相同，但在不同页表的实现方法中叫法不同。**

第九章 虚拟内存

【虚拟内存】允许程序执行时不必完全载入物理内存的一种内存管理技术。

【虚拟内存优点】1.扩大逻辑地址空间 2.允许多进程共享地址空间 3.提升进程创建效率。

***进程的虚拟地址空间是进程如何在内存中存放的逻辑视图

进程包括代码段、数据段、堆、栈以及堆和栈之间的孔。这个孔是虚拟内存的一部分，只有在堆和栈生长时，才需要实际的物理地址。所以堆和栈之间的区域是动态分配的。

虚拟地址空间连续存放，需要 MMU 将逻辑页映射到内存的物理页帧。

***虚拟内存允许共享，实现进程之间内存共享：比如两个进程需要相同的动态库，不用把这个库均放到两个进程的物理内存中，只需要通过页表来共享该区域。

【按需调页】即在需要的时候才将页调入内存。

***执行程序从磁盘载入内存的方式：全部载入（全部载入不需要按需调用）、部分载入

【按需调页优点】1.减少 I/O 2.减少内存使用 3. 应答时间快

【有效-无效位】页表内设有效位，区分页在物理内存还是磁盘。

【有效（v)】页在物理内存中

【无效（i)】也在虚拟内存中

【按需调页操作由两部分组成】1.交换程序（无空闲帧时） 2.调页程序（将页从虚拟内存换入 swap in 物理内存时）

【页错误】当访问无效页**（页在磁盘）**时，出现页错误，并陷入操作系统，即页错误触发操作系统的载入操作，该页需要从虚拟内存载入物理内存。

【页错误处理步骤】

1）确认该访问是否合法

2）找到一个空闲帧（若无空闲帧，则执行**页置换**）

3）将所需页调入该帧

4）修改页表（有效-无效位）

5）重新开始因页错误而中断的指令

【写时复制 COW】(创建进程时使用虚拟内存的长处）允许父进程和子进程共享物理内存中的页。只有当进程需要对页进行写操作时才创建一个共享页的副本。

【COW 优点】1.加速进程创建 2.最小化新创建进程的页数节省新进程内存空间。

*****【区别】*****

1. 分配空闲页（COW 时）→空闲缓冲池

2. 分配空闲帧（页置换时→加快换入页的执行）→空闲帧缓冲池

【无空闲帧可分配的解决方法】1.终止进程 2.交换出一个进程，即页置换。

【页置换算法】找出当前没有使用的帧，将其换出；需要使用的页将其换入。性能指标：最小化页错误次数（注意同一个页有可能多次被释放&载入）

***页置换是按需调页的基础，它分开了逻辑内存和物理内存，给程序员提供了巨大的内存空间。

【页置换的基本操作】

1. 查找所需页在磁盘上的位置
2. 查找一个空闲帧：

a) 如果有空闲帧就用

b) 如果没有空闲帧，就通过某种置换算法选择一个牺牲帧

3. 将所需要的页读入空闲帧，修改页表和帧表

4. 重启进程

【页置换问题】页置换需要两次页传输（换入和换出），导致页处理时间加倍，增加了内存访问时间。

【解决方法】1.换出：每个页关联一个修改位；通过修改位确认关联页是否被修改：如果被修改过，在换出时必须写入磁盘；如果没有修改过，换出时不需要写入磁盘，从而避免了写入磁盘操作。**（数据不变不写，变了才写）**

2.换入：**页缓冲**：利用**空闲帧缓冲池**，系统保留一个空闲帧缓冲池，当需要牺牲帧写出虚拟内存之前，从空闲帧缓冲池先得到内存。**（一部分内存空间为缓冲池，只有待换入的页才可以被分配到该缓冲池）（相当于牺牲内存空间来换取较快的响应速度）**

【页置换算法】

*置换算法**目标**：**最小化页错误发生**

*利用**引用串**来评估一个置换算法

*引用串：一系列页的序号（即要使用的页号的顺序）

*评估：检查发生的页错误次数

【FIFO】最先进入的帧先被替换（Belady 异常：帧数越多反而页错误更频繁）

【最优置换】替换掉在未来最长时间不使用的帧（理想下最优）

【最近最少使用 LRU】每个页关联该页上次使用的时间，替换掉历史最长时间没有使用的帧。

【近似 LRU 算法】

【附加引用位】LRU 算法改进。（方便比较）

I. 附加引用位算法

- 每个页都与引用位相关联
- 每当引用页时，相应页的引用位就被硬件置位
- 开始，引用位被初始化为0
- 页被引用，引用位被设置为1；没被引用，就设置为0（在规定的时间周期）
- 如8个字节的引用位表示对8个周期进行记录引用位
- 每次引用的记录，放到8位字节的最高位，而将其他位向右移一位，并抛弃最低位

0	0	0	0	0	0	0	0

1	1	1	1	1	1	1	1

【二次机会】在 LRU 基础上只保留 1 位引用位，替换时给刚使用的帧第二次机会

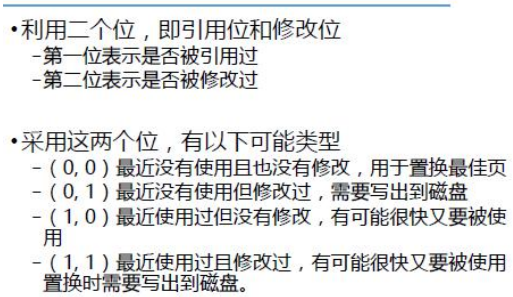
II. 二次机会算法

每当需要置换页时，检查每页相关联的引用位，

1. 如引用位为0就置换，并把引用位设置成 1
2. 如引用位为1就跳过（给第二次机会），并清零，然后跳到下一个FIFO 页

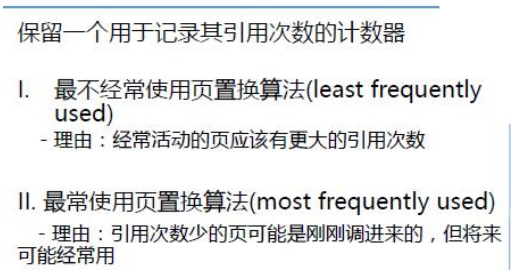
【增强型二次机会】增加引用位（表示最近是否有使用）和修改位（表示是否被修改过，决定置换时是否需要写出磁盘），最佳页是最近没有使用且没有修改的。

III. 增强型二次机会算法



【基于计数的页置换】增加计数器记录帧使用次数：1.最不经常使用页置换算法（**看过去**：经常活动的页应该有更大的引用次数）、最常使用页置换算法（**看未来**：引用次数少的页可能是刚刚调进来的，但将来可能经常用）

(4) 基于计数的页置换



【页置换加速】

【修改位】页表内增加修改位，当页内数据变更时，标记位修改。发生页置换时，仅当修改位为修改时，才写入磁盘；否则没有必要写回磁盘。

【页缓冲】使用**空闲帧缓冲池**，发生页置换时可以先取得空闲帧写入后再换出牺牲帧。

【物理帧分配】系统针对进程的分配，每个进程需要分配最小需要运行的页，即将部分进程内容载入到内存。

【分配方式】1.**平均分配（每个进程分配物理帧的大小相同）**

2.比例分配（根据进程大小比例）

3.优先级分配（根据进程优先级分配）

***Linux 采取进程固定优先级分配

【页置换分类（范围）】1.**全局置换**（从所有进程帧集中选择一个置换帧：该进程分配到的帧数量可能发生变化）

2.局部置换（进程仅从自己分配的帧中选择一个置换帧：该进程分配到的帧数量不会发生变化）

***全局置换有可能会影响别的进程的正常运行，但该进程本身不会受到影响；局部置换与之相反。所以二者各有利弊。

【系统颠簸】因一个进程没有分配到足够的页帧，页错误频繁发生，进程页置换时间高于执行时间。

【系统颠簸后果】

1. **CPU 使用率下降，操作系统会试图增加多道程序的程度**（增加多道程序程度即把更多的进程放入内存，这样一个进程 I/O 时就可以上下文切换去执行另一个进程从而提高 CPU 的使用率。但进程越多越易发生页错误，所以会恶性循环，所以页置换策略尤为重要）
2. **进程会试图去抢别的进程的帧**

***对磁盘文件进行操作/实现 open()，read()，write()等系统调用的两种方法：

1. 对磁盘上的文件进行直接的操作
2. 利用虚拟内存技术（把文件映射到内存，按内存管理的方式来管理文件，访问文件用访问内存的方式）→这叫做**内存映射文件**

【内存映射文件】利用**虚拟内存**技术，将文件当作内存访问。将磁盘块映射成内存的一页或多页：1.访问文件会发生页错误(因为文件在磁盘中)2.文件的读写就按通常的内存访问来处理 3.文件的写操作不会立即发生，而是定期写入磁盘或在文件关闭的时候写入。

【内存映射文件的优点】多个进程可以将同一个文件映射到各自的虚拟内存中，以允许数据共享（和通过共享页表来实现共享内存一样的意思）

【内核内存的分配】从空闲内存池中获取。内核需要为不同大小的内核数据结构分配内存，而且必须是连续分配。内核内存分配不同于用户内存分配，其不受分页系统的控制。

【Buddy 分配】从物理上连续的大小的固定的段上分配（2 的幂大小）

【优点】可通过合并而快速地形成更大的段。可把邻近的空闲空间合并为一个更大的内存连续空间，来给请求更大内存空间的数据结构块分配。

【缺点】可能产生内部碎片

【Slab 分配】slab 是一个或多个物理上连续的页，cache 含有一个或多个 slab。每个内核数据结构（信号量，文件对象，进程描述符等）都有它自己的 cache，每个 cache 含有内核数据结构的对象实例。当创建 cache 时，起初包含若干标记为空闲的对象，对象的数量于 slab 的大小有关（因为 cache 就是由 slab 组成的）。当需要内核数据结构的对象时，可从 cache 中获取，并标记为使用。

【优点】1.不会产生内部碎片从而没有内存浪费。（因为每个内核数据结构都有相应地 cache，而每个 cache 都由若干 slab 组成，而每个 slab 又分为若干个与对象大小相同的部分，所以内核请求对象内存时，slab 分配器可以返回刚好可以表示对象所需的内存。）

2.内存请求可以快速满足。对象预先创建，所以可以从 cache 上快速分配。对象用完释放时，只需要标记为空闲并返回给 cache，以便下次继续使用。

【区别】

1. 内核内存的分配（buddy & slab）是给**内核进程**分配内存，其不受分页系统的控制，**只有连续分配**，该内存从**空闲内存池**中获取。
2. 内存空间的分配（**连续**分配：多分区&可变分区，**不连续**分配：分页分配&分段分配）是给**用户进程**分配内存该内存从**内核所维护的空闲页帧链表**中获取。
3. 内存分配均是操作系统来分配。