操作系统笔记

目录

第一章.计算机系统概述

1.基本构成

计算机的四个主要组件

- 处理器
- 内存
- I/O 模块
- 系统总线

2.指令的执行

基本指令周期，指令处理包括 2 步：

- 处理器从存储器一次读一条指令
- 执行每条指令

处理器中的 PC 保存下一条指令的地址，IR 保存当前即将执行的指令

3.中断

允许“其他模块”（I/O、存储器）中断“处理器”正常处理过程的机制

3.1 目的

提高 CPU 利用率，防止一个程序垄断 CPU 资源

3.2 类型

- 1）程序中断
- 2）时钟中断
- 3）I/O 中断
- 4）硬件失效中断

3.3 中断控制流

I/O 程序：

- 指令序列 4：为实际 I/O 作准备
- I/O 命令：如果不使用中断，执行命令时，程序必须等待 I/O 设备执行请求的函数（或周期性地检测 I/O 设备的状态或轮询 I/O 设备）。程序可能通过简单地重复执行一个测试操作的方式进行等待，以确定 I/O 操作是否完成
- 指令序列 5：操作完成，包括设置成功或失败标签

中断：短 I/O 等待

- 利用中断功能，处理器可以在 I/O 操作的执行过程中执行其它指令：用户程序到达系统调用 WRITE 处，但涉及的 I/O 程序仅包括准备代码和真正的 I/O 命令。在这些为数不多的几条指令执行后，控制返回到用户程序。在这期间，外部设备忙于从计算机存储器接收数据并打印。这种 I/O 操作和用户程序中指令的执行是并发的
- 当外部设备做好服务的准备时，也就是说，当它准备好从处理器接收更多的数据时，该外部设备的 I/O 模块给处理器发送一个中断请求信号。这时处理器会做出响应，暂停当前程序的处理，转去处理服务于特定 I/O 设备的程序，这个程序称为中断处理程序。在对该设备的服务响应完成后，处理器恢复原先的执行

中断：长 I/O 等待

- 对于如打印机等较慢的设备来说，I/O 操作比执行一系列用户指令的时间长得多，因此在下一次 I/O 操作时，前一次 I/O 可能还为执行完。在上图 c)中，第二次 WRITE 调用时，第一次 WRITE 的 I/O 还为执行完，结果是用户程序会在这挂起，当前面 I/O 完成后，才能继续新的 WRITE 调用

3.4 中断处理

中断激活了很多事件，包括处理器硬件中的事件及软件中的事件
被中断程序的信息保存与恢复：

3.5 多个中断

在处理一个中断的过程中，可能会发生另一个中断，处理多个中断有 2 种方法

- **当正在处理一个中断时，禁止再发生中断**：如果有新的中断请求信号，处理器不予理睬。通常在处理中断期间发生的中断会被挂起，当处理器再次允许中断时再处理
- **定义中断优先级**：允许高优先级的中断处理打断低优先级的中断处理程序的允许

4.存储器的层次结构

从上往下看，会出现以下情况：*每“位”的价格递减 * 容量递增 * 存取时间递增 * 处理器访问存储器的频率递减（有效的基础是访问的局部性原理）

4.1 高速缓存

内存的存储周期跟不上处理器周期，因此，利用局部性原理在处理器和内存间提供一个容量小而速度快的存储器，称为高速缓存

上图中高速缓存通常分为多级：L1、L2、L3

5.直接内存存取(DMA)

针对 I/O 操作有 3 种可能的技术 * 可编程(程序控制)I/O（需处理器干预） * 中断驱动 I/O（需处理器干预） * 直接内存存取
当处理器正在执行程序并遇到一个 I/O 相关的指令时，它通过给相应的 I/O 模块发命令来执行这个指令：
1）使用可编程 I/O 时，I/O 模块执行请求的动作并设置 I/O 状态寄存器中相应的位，**但它并不进一步通知处理器，尤其是它并不中断处理器**，因此处理器在执行 I/O 指令后，还需定期检查 I/O 模块的状态。为了确定 I/O 模块是否做好了接收或发送更多数据的准备，处理器等待期间必须不断询问 I/O 模块的状态，这会严重降低整个系统的性能
2）如果是中断驱动 I/O，在给 I/O 模块发送 I/O 命令后，处理器可以继续做其它事。当 I/O 模块准备好与处理器交换数据时，会中断处理器并请求服务，处理器接着响应中断，完成后再恢复以前的执行过程
尽管中断驱动 I/O 比可编程 I/O 更有效，但是**处理器仍需要主动干预在存储器和 I/O 模块直接的数据传送，并且任何数据传送都必须完全通过处理器**。由于需要处理器干预，这两种 I/O 存在下列缺陷：

- I/O 传送速度受限于处理器测试设备和提供服务的速度（数据传送受限于处理器）
- 处理器忙于管理 I/O 传送工作，必须执行很多指令以完成 I/O 传送（处理器为数据传送需要做很多事）

3）因此，当需要移动大量数据时，需要使用一种更有效的技术：直接内存存取。DMA 功能可以由系统总线中一个独立的模块完成，也可以并入到一个 I/O 模块中。
DMA 的工作方式如下，当处理器需要读写一块数据时，它给 DMA 模块产生一条命令，发送下列信息：

- 是否请求一次读或写
- 涉及的 I/O 设备的地址
- 开始读或写的存储器单元
- 需要读或写的字数

之后处理器继续其它工作。处理器将这个操作委托给 DMA 模块，DMA 模块直接与存储器交互，这个过程不需要处理器参与。当传送完成后，DMA 模块发送一个中断信号给处理器。因此只有在开始和结束时，处理器才会参与

第二章.操作系统概述

1.操作系统的目标和功能

操作系统是控制应用程序执行的程序，并充当应用程序和计算机硬件之间的接口

- 作为用户/计算机接口
- 作为资源管理器（操作系统控制处理器使用其他系统资源，并控制其他程序的执行时机）
- 易扩展性

2.操作系统的发展

1. **串行处理**：程序员直接与计算机硬件打交道，因为当时还没操作系统。这些机器在一个控制台上运行，用机器代码编写的程序通过输入设备载入计算机。如果发生错误使得程序停止，错误原因由显示灯指示。如果程序正常完成，输出结果出现在打印机中
2. **简单批处理系统**：中心思想是使用一个称为监控程序的软件。通过使用这类操作系统，用户不再直接访问机器，相反，用户把卡片或磁带中的作业提交给计算机操作员，由他把这些作业按顺序组织成一批，并将整个批作业放在输入设备上，供监控程序使用。每个程序完成处理后返回到监控程序，同时，监控程序自动加载下一个程序
3. **多道批处理系统**：简单批处理系统提供了自动作业序列，但是处理器仍经常空闲，因为对于 I/O 指令，处理器必须等到其执行完才能继续。内存空间可以保持操作系统和一个用户程序，假设内存空间容得下操作系统和两个用户程序，那么当一个作业需要等到 I/O 时，处理器可以切换到另一个可能不需要等到 I/O 的作业。进一步还可以扩展存储器保存三个、四个或更多的程序，并且在他们之间进行切换。这种处理称为多道程序设计或多任务处理，是现代操作系统的主要方案
4. **分时系统**：正如多道程序设计允许处理器同时处理多个批作业一样，它还可以用于处理多个交互作业。对于后一种情况，由于多个用户分享处理器时间，因而该技术称为分时。在分时系统中，多个用户可以通过终端同时访问系统，由操作系统控制每个用户程序以很短的时间为单位交替执行

以下为多道批处理系统与分时系统的比较

	批处理多道程序设计	分时
主要目标	充分使用处理器	减小响应时间
操作系统指令源	作业控制语言；作业提供的命令	终端输入的命令

3.现代操作系统

对操作系统要求上的变化速度之快不仅需要修改和增强现有的操作系统体系结构，而且需要有新的操作系统组织方法。在实验用和商用操作系统中有很多不同的方法和设计要素，大致分为以下几类：

- 微内核体系结构
- 多线程
- 对称多处理
- 分布式操作系统
- 面向对象设计

大内核：至今为止大多数操作系统都有一个单体内核，操作系统应该提供的大多数功能由这些大内核提供，包括调度、文件系统、网络、设备管理器、存储管理等。典型情况下，这个大内核是作为一个进程实现的，所有元素共享相同的地址空间

微内核：微内核体系结构只给内核分配一些最基本的功能，包括地址空间，进程间通信和基本的调度。其它操作系统服务都是由运行在用户态下且与其他应用程序类似的进程提供，这些进程可以根据特定应用和环境定制。这种方法把内核和服务程序的开发分离开，可以为特定的应用程序或环境要求定制服务程序。可以使系统结构的设计更简单、灵活，很适合于分布式环境

第三章.进程

1.进程的定义

进程有以下定义：

- 一个正在执行中的程序
- 一个正在计算机上执行的程序实例
- 能分配给处理器并由处理器执行的实体
- 一个具有以下特征的活动单元：一组指令序列的执行、一个当前状态和相关的系统资源集

也可以把进程视为由**程序代码、和代码相关联的数据集、进程控制块**组成的实体

进程控制块：由操作系统创建和管理。进程控制块包含了充分的信息，这样就可以中断一个进程的执行，并且在后来恢复执行进程时就好像进程未被中断过一样。进程控制块是操作系统能够支持多进程和提供多重处理技术的关键，**进程控制块是操作系统中最重要的数据结构，每个进程控制块包含操作系统所需要的关于进程的所有信息**

- 内存指针：包括程序代码和进程相关数据的指针，还有和其他进程共享内存块的指针
- 上下文数据：进程执行时处理器寄存器中的数据

进程被中断时，操作系统会把程序计数器和上下文数据保存到进程控制块中的相应位置

程序状态字(PSW)：所有处理器设计都包括一个或一组通常称为程序状态字的寄存器，包含有进程的状态信息

2.进程的状态

2.1 进程的创建与终止

进程按以下步骤创建：

1. 给新进程分配一个唯一的进程标识符
2. 给新进程分配空间（包括进程映像中的所有元素）
3. 初始化进程控制块
4. 设置正确的连接（保存到相应队列）

会导致创建进程的事件：

会导致终止进程的事件：

2.2 两状态进程模型

2.3 五状态进程模型

运行态->就绪态：1）超时：即正在运行的进程到达了”允许不中断执行“的最大时间段（所有多道程序操作系统都实现了这类时间限定）2）优先级低的进程被优先级高进程抢占（并不是所有操作系统都实现了）

图 b)中一个事件对应一个队列。当事件发生时，相应队列中的所有进程都转换到就绪态

除此之外，就绪队列也可以按照优先级组织成多个队列

2.4 引入”挂起态“的进程模型

为何引入？

考虑一个没有使用虚拟内存的系统，每个被执行的进程必须完全载入内存，因此，2.3 图 b)中，所有队列中的所有进程必须驻留在内存中

所有这些设计机制的原因都是由于 I/O 活动比计算速度慢得多，因此在单道程序系统中的处理器大多数时候是空闲的。但是 2.3 图 b)的方案并未完全解决这个问题。在这种情况下，内存保存有多个进程，当一个进程正在等待时，处理器可以转移到另一个进程，但是处理器比 I/O 要快的多，以至于内存中所有的进程都在等待 I/O 的情况很常见。因此，即使是多道程序设计，大多数时候处理器仍然处于空闲

因此，**可以把内存中某个进程的一部分或全部移出到磁盘中**。当内存中没有处于就绪状态的进程时，操作系统就把被阻塞的进程换出到磁盘中的”挂起队列“。操作系统在此之后取出挂起队列中的另一个进程，或者接受一个新进程的请求，将其纳入内存运行
“交换”是一个 I/O 操作，因而也可能使问题更加恶化。但是由于磁盘 I/O 一般是系统中最快的 I/O(相对于磁带或打印机 I/O)，所以交换通常会提高性能

进程模型

- **就绪/挂起->就绪**：1）内存中没有就绪态进程，需要调入一个进程继续执行；2）处于就绪/挂起的进程具有更高优先级
- **就绪->就绪/挂起**：1）如果释放空间以得到足够空间的唯一方法是挂起一个就绪态的进程；2）如果操作系统确信高优先级的阻塞态进程很快将会就绪，那么可能会挂起一个低优先级的就绪态进程而不是一个高优先级的阻塞态进程
- **新建->就绪/挂起**：进程创建需要为其分配内存空间，如果内存中没有足够的空间分配给新进程，会使用”新建->就绪/挂起“转换
- **阻塞/挂起->阻塞**：比较少见。如果一个进程终止，释放了一些内存空间，阻塞/挂起队列中有一个进程比就绪/挂起队列中任何进程的优先级都要高，并且操作系统有理由相信阻塞进程的事件很快就会发生
- **运行->就绪/挂起**：如果位于阻塞/挂起队列中的具有较高优先级的进程变得不再阻塞，操作系统抢占这个进程，也可以直接把这个进程转换到就绪/挂起队列中，并释放一些内存

导致进程挂起的原因

3.进程的描述

操作系统为了管理进程和资源，必须掌握关于每个进程和资源当前状态的信息。普遍使用的方法是：操作系统构造并维护它所管理的每个实体的信息表：

内存表用于跟踪内(实)存和外存(虚拟内存)
使用**进程映像**来描述一个进程，进程镜像包括：**程序、数据、栈和进程控制块(属性的集合)**：

下图为一个典型的**进程映像**结构：

4.进程控制

4.1 执行模式

大多数处理器至少支持两种执行模式：

- **用户态**
- **内核态(系统态、控制态)**：软件具有对处理器及所有指令、寄存器和内存的控制能力

使用两种模式的原因是很显然的，它可以保护操作系统和重要的操作系统表(如进程控制块)不受用户程序的干涉

处理器如何知道它正在什么模式下执行及如何改变模式?

程序状态字(PSW)中有一位表示执行模式，这一位应某些事件的要求而改变。在典型情况下，

- 当用户调用一个操作系统服务或中断触发系统例程的执行时，执行模式被设置为内核态
- 当从系统服务返回到用户进程时，执行模式被设为用户态

4.2 进程切换

在下列事件中，进程可能把控制权交给操作系统：

- **系统中断**：

  - **中断**：与当前正在运行的进程无关的某种类型的外部事件相关。控制首先转移给中断处理器，做一些基本的辅助工作后，转到与已经发生的特定类型的中断相关的操作系统例程
  - **陷阱**：与当前正在运行的进程所产生的错误或异常条件相关。操作系统首先确定错误或异常条件是否是致命的。1）如果是，当前进程被换到退出态，发生进程转换；2）如果不是，动作取决于错误的种类或操作系统的设计，可能会进行一次进程切换或者继续执行当前进程
- **系统调用**：转移到作为操作系统代码一部分的一个例程上执行。通常，使用系统调用会把用户进程置为阻塞态

进程切换步骤如下： 1. 保存处理器上下文环境（包括程序计数器和其它寄存器） 2. 更新当前处于运行态进程的进程控制块（状态和其它信息） 3. 将进程控制块移到相应队列 4. 选择另一个进程执行 5. 更新所选择进程的进程控制块（包括将状态变为运行态） 6. 更新内存管理的数据结构 7. 恢复处理器在被选择的进程最近一次切换出运行状态时的上下文环境

进程切换一定有模式切换；模式切换不一定有进程切换（中断会发生模式切换，但是在大多数操作系统中，中断的发生并不是必须伴随着进程的切换的。可能是中断处理器执行之后，当前正在运行的程序继续执行）；

第四章.线程

1.进程与线程

- **进程**是操作系统进行资源分配的基本单位
- **线程**是调度的基本单位

进程中的所有线程共享该进程的状态和资源，进程和线程的关系如下图：

从性能上比较，线程具有如下优点：

1. 在一个已有进程中创建一个新线程比创建一个全新进程所需的时间要少许多
2. 终止一个线程比终止一个进程花费的时间少
3. 同一进程内线程间切换比进程间切换花费的时间少
4. 线程提高了不同的执行程序间通信的效率（在大多数操作系统中，独立进程间的通信需要内核的介入，以提供保护和通信所需要的机制。但是，由于在同一个进程中的线程共享内存和文件，它们无须调用内核就可以互相通信）

2.线程状态

和进程一样，线程的关键状态有运行态、就绪态和阻塞态。一般来说，挂起态对线程没有什么意义。这是由于此类状态是一个进程级的概念。特别地，如果一个进程被换出，由于它的所有线程都共享该进程的地址空间，因此它们必须都被换出

有 4 种与线程相关的基本操作：

- **派生**：在典型情况下，当派生一个新进程时，同时也为该进程派生了一个线程。随后，进程中的线程可以在同一进程中派生另一个线程，并为新线程提供指令指针和参数；新线程拥有自己的寄存器上下文和栈空间，且被放置在就绪队列中
- **阻塞**：当线程需要等待一个事件时，它将被阻塞（保存它的用户寄存器、程序计数器和栈指针），此时处理器转而执行另一个处于同一进程中或不同进程中的就绪线程
- **解除阻塞**：当阻塞一个线程的事件发生时，该线程被转移到就绪队列中
- **结束**：当一个线程完成时，其寄存器上下文和栈都被释放

线程的生命周期

3.线程分类

线程的实现可以分为两大类：

- **用户级线程**：有关线程管理的所有工作都由应用程序完成(使用线程库)，内核意识不到线程的存在
- **内核级线程**：有关线程管理的所有工作都由内核完成，应用程序部分没有进行线程管理的代码

3.1 用户级线程

在用户级线程中，进程和线程的状态可能有如下转换：

- a)->b)：**线程 2 中执行的应用程序代码进行系统调用，阻塞了进程 B**。例如，进行一次 I/O 调用。这导致控制转移到内核，内核启动 I/O 操作，把进程 B 置于阻塞状态，并切换到另一个进程。在此期间，根据线程库维护的数据结构，进程 B 的线程 2 仍处于运行状态。值得注意的是，从处理器上执行的角度看，线程 2 实际上并不处于运行态，但是在线程库看来，它处于运行态
- a)->c)：**时钟中断把控制传递给内核**，内核确定当前正在运行的进程 B 已经用完了它的时间片。内核把进程 B 置于就绪态并切换到另一个进程。同时，根据线程库维护的数据结构，进程 B 的线程 2 仍处于运行态
- a)->d)：**线程 2 运行到需要进程 B 的线程 1 执行某些动作的一个点**。此时，线程 2 进入阻塞态，而线程 1 从就绪态转换到运行态。进程自身保留在运行态

在前两种情况中，当内核把控制切换回进程 B 时，线程 2 会恢复执行

还需注意，**进程在执行线程库中的代码时可以被中断**，或者是由于它的时间片用完了，或者是由于被一个更高优先级的进程所抢占。因此在中断时，进程可能处于线程切换的中间时刻。当该进程被恢复时，线程库得以继续运行，并完成线程切换和把控制转移给另一个线程

用户级线程的优点

1. 由于所有线程管理数据结构都在一个进程的用户地址空间中，线程切换不需要内核态特权，节省了两次状态转换的开销
2. 调度可以是应用程序相关的（一个应用程序可能更适合简单的轮转调度，另一个可能更适合基于优先级的调度），可以为应用量身定做调度算法而不扰乱底层操作系统调度程序
3. 可以在任何操作系统中运行，不需要对底层内核进行修改以支持用户级线程

用户级线程的缺点

1. 当用户级线程执行一个系统调用时，不仅这个线程会被阻塞，进程中的所有线程都会被阻塞
2. 一个多线程应用程序不能利用多处理技术。内核一次只把一个进程分配给一个处理器，因此一次进程中只有一个线程可以执行（事实上，在一个进程内，相当于实现了应用程序级别的多道程序）

3.2 内核级线程

内核能意识到线程的存在

内核级线程的优点

1. 内核可以同时把同一进程中的多个线程调度到多个处理器中同时运行
2. 如果进程中一个线程被阻塞，内核可以调度其它线程
3. 内核例程自身也可以使用多线程

内核级线程的缺点

1. 把控制从一个线程转移到用一进程的另一线程时，需要到内核的状态切换

3.3 混合方案

可以混合使用用户级和内核级线程。在混合方案中，同一应用程序中的多个线程可以在多个处理器上并行地运行，某个会引起阻塞的系统调用不会阻塞整个进程。
如果设计正确，该方法将会结合纯粹用户级线程和内核级线程方法的优点，同时克服它们的缺点

第五章.并发

并发相关的术语：

1.互斥

可以根据进程相互之间知道对方是否存在的程度，对**进程间的交互**进行分类：

- **进程间的资源竞争**：每个进程不影响它所使用的资源，这类资源包括 I/O 设备、存储器、处理器时间和时钟。首先需要提供互斥要求（比方说，如果不提供对打印机的互斥访问，打印结果会穿插）。实施互斥又产生了两个额外的控制问题：死锁和饥饿
- **进程间通过共享的合作**：进程可能使用并修改共享变量而不涉及其他进程，但却知道其他进程也可能访问同一数据。因此，进程必须合作，以确保共享的数据得到正确管理。由于数据保存在资源中（设备或存储器），因此再次涉及有关互斥、死锁、饥饿等控制问题，除此之外，还有一个新要求：数据的一致性
- **进程间通过通信的合作**：由于在传递消息的过程中，进程间未共享任何对象，因而这类合作不需要互斥，但是仍然存在死锁和饥饿问题（死锁举例：两个进程可能都被阻塞，每个都在等待来自对方的通信；饥饿举例：P1,P2,P3，P1 不断试图与 P2，P3 通信，P2 和 P3 都试图与 P1 通信，如果 P1 和 P2 不断交换信息，而 P3 一直被阻塞，等待与 P1 通信，由于 P1 一直是活跃的，P3 处于饥饿状态）

1.1 互斥的硬件支持

1) 中断禁用（只对单处理器有效）：为保证互斥，只需保证一个进程不被中断即可

```
while(true){
    /* 禁用中断 */
    /*  临界区  */
    /* 启用中断 */
    /* 其余部分 */
}
```

问题：

- 处理器被限制于只能交替执行程序，因此执行的效率将会有明显的降低
- 该方法不能用于多处理器结构中

2) 专用机器指令

- **比较和交换指令**
- **交换指令**

在硬件级别上，对存储单元的访问排斥对相同单元的其它访问。基于这一点，处理器的设计者提出了一些机器指令，用于保证两个动作的原子性。在指令执行的过程中，任何其它指令访问内存将被阻止

```
/*比较和交换指令*/
int bolt;
void P(int i)
{
    while(true){
        while(compare_and_swap(&bolt,0,1) == 1)
            /*不做任何事*/;
        /*临界区*/
        bolt = 0;
        /*其余部分*/
    }
}
```

```
int compare_and_swap(int *word,int testval,int newval)
{
    int oldval;
    oldval = *word;
    if(oldval == testval) *word = newval;
    return oldval;
}

/*交换指令*/
int bolt;
void P(int i)
{
    int keyi = 1;
    while(true){
        do exchange (&keyi,&bolt);
        while(keyi != 0);
        /*临界区*/
        bolt = 0;
        /*其余部分*/
    }
}

void exchange (int *register,int *memory)
{
    int temp;
    temp = *memory;
    *memory = *register;
    *register = temp;
}
```

优点

- 适用于单处理器或共享内存的多处理上的任何数目的进程
- 简单且易于证明
- 可用于支持多个临界区（每个临界区可以用它自己的变量定义）

缺点

- 使用了忙等待（进入临界区前会一直循环检测，会销毁处理器时间）
- 可能饥饿（忙等的进程中可能存在一些进程一直无法进入临界区）
- 可能死锁（P1 在临街区中时被更高优先级的 P2 抢占，P2 请求相同的资源）

1.2 互斥的软件支持

软件支持包括操作系统和用于提供并发性的程序设计语言机制，常见如下表：

1）信号量

通常称为计数信号量或一般信号量

可把信号量视为一个具有整数值的变量，在它之上定义三个操作：

1. 一个信号量可以初始化为非负数（表示发出 semWait 操作后可立即执行的进程数量）
2. semWait 操作使信号量减 1。若值为负数，执行该操作进程被阻塞。否则进程继续执行
3. semSignal 操作使信号量加 1。若值小于或等于 0，则被 semWait 阻塞的进程被解除阻塞

信号量原语的定义：

```
struct semaphore{
    int count;
    queueType queue;
};

void semWait(semaphore s)
{
    s.count--;
    if(s.count < 0){
        /*把当前进程插入到队列当中*/;
        /*阻塞当前进程*/;
    }
}

void semSignal(semaphore s)
{
    s.count++;
    if(s.count <= 0){
        /*把进程P 从队列中移除*/;
        /*把进程P 插入到就绪队列*/;
    }
}
```

2）二元信号量

二元信号量是一种更特殊的信号量，它的值只能是 0 或 1

可以使用下面 3 种操作：

1. 可以初始化为 0 或 1
2. semWaitB 操作检查信号的值，如果为 0，该操作会阻塞进程。如果值为 1，将其改为 0 后进程继续执行
3. semSignalB 操作检查是否有任何进程在信号上阻塞。有则通过 semSignalB 操作，受阻进程会被唤醒，如果没有，那么设置值为 1

二元信号量的原语定义：

```
struct binary_semaphore{
    enum {zero,one} value;
    queueType queue;
};

void semWaitB(binary_semaphore s)
{
    if(s.value == one)
        s.value = zero;
    else{
        /*把当前进程插入到队列当中*/;
        /*阻塞当前进程*/;
    }
}

void semSignalB(binary_semaphore s)
{
    if(s.queue is empty())
        s.value = one;
```

```
    else{
        /*把进程P 从等待队列中移除*/;
        /*把进程P 插入到就绪队列*/;
    }
}
```

- 强信号量：队列设计为 FIFO，被阻塞最久的进程最先从队列中释放（保证不会饥饿）
- 弱信号量：没有规定进程从队列中移出顺序

使用信号量的互斥（这里是一般信号量，不是二元信号量）

```
const int n = /*进程数*/
semaphore s = 1;

void P(int i)
{
    while(true){
        semWait(s);
        /*临界区*/;
        semSignal(s);
        /*其它部分*/;
    }
}

void main()
{
    parbegin(P(1),P(2),...,P(n));
}
```

下图为三个进程使用了上述互斥协议后，一种可能的执行顺序：

信号量为实施互斥及进程间合作提供了一种原始但功能强大且灵活的工具，但是，使用信号量设计一个正确的程序是很困难的，其难点在于 semWait 和 semSignal 操作可能分布在整个程序中，却很难看出这些在信号量上的操作所产生的整体效果（详见 1.3 经典互斥问题中的“生产者/消费者”问题）

3）互斥量

互斥量和二元信号量关键的区别在于：互斥量加锁的进程和解锁的进程必须是同一进程

4）管程

管程是一个程序设计语言结构，它提供了与信号量同样的功能，但更易于控制。它是由**一个或多个过程**，**一个初始化序列**和**局部数据**组成的软件模块，主要特点如下：

1. 局部数据变量只能被管程的过程访问，任何外部过程都不能访问
2. 一个进程通过调用管程的一个过程进入管程
3. 在任何时候，只能有一个进程在管程中执行，调用管程的其它进程都被阻塞，等待管程可用

为进行并发处理，管程必须包含同步工具（例如：一个进程调用了管程，并且当它在管程中时必须被阻塞，直到满足某些条件。这就需要一种机制，使得该进程在管程内被阻塞时，能释放管程，以便其它进程可以进入。以后，当条件满足且管程在此可用时，需要恢复进程并允许它在阻塞点重新进入管程）

管程通过使用**条件变量**提供对同步的支持，这些条件变量包含在管程中，并且只有在管程中才能被访问。有 2 个操作：

- cwait(c)：调用进程的执行在条件 c 上阻塞，管程现在可被另一个进程使用
- csignal(c)：恢复执行在 cwait 后因某些条件被阻塞的进程。如果有多个则选择其一；如果没有则什么也不做

管程的结构如下：

管程优于信号量之处在于，所有的同步机制都被限制在管程内部，因此，不但易于验证同步的正确性，而且易于检查出错误。此外，如果一个管程被正确编写，则所有进程对保护资源的访问都是正确的；而对于信号量，只有当所有访问资源的进程都被正确地编写时，资源访问才是正确的

5）消息传递

最小操作集：

- send(destination,message)
- receive(source,message)

阻塞：

- 当一个进程执行 send 原语时，有 2 种可能：
  - 发送进程被阻塞直到这个消息被目标进程接收
  - 不阻塞
- 当一个进程执行 receive 原语后，也有 2 种可能：
  - 如果一个消息在此之前被发送，该消息被正确接收并继续执行
  - 没有正在等待的消息，则 a)进程阻塞直到等待的消息到达，b)继续执行，放弃接收的努力

消息传递过程中需要识别消息的源或目的地，这个过程称为**寻址**，可分为两类： 1. 直接寻址 * 对于 send: 包含目标进程的标识号 * 对于 receive：1）进程显示指定源进程；2）不可能指定所希望的源进程时，通过 source 参数保存相应信息 2. 间接寻址（解除了发送者/接收者的耦合性，更灵活） * 消息发送到一个共享数据结构，称为"信箱"。发送者和接收者直接有"一对一"、"多对一"、"一对多"和"多对多"的对应关系（典型的"多对一"如客户端/服务器，此时"信箱"就是端口）

消息传递实现互斥(消息函数可视为在进程直接传递的一个令牌)：

```
const int n = /*进程数*/;
void P(int i)
{
    message msg;
    while(true){
        receive(box,msg);
        /*临界区*/;
        send(box,msg);
        /*其它部分*/;
    }
}

void main()
{
    create mailbox (box);
    send(box,null);
    parbegin(P(1),P(2),...,P(n));
}
```

可以使用消息传递处理"生产者/消费者问题"，可以有多个消费者和生产者，系统甚至可以是分布式系统，代码见 1.3

1.3 经典问题

在设计同步和并发机制时，可以与一些经典问题联系起来，以检测该问题的解决方案对原问题是否有效

1）生成者/消费者问题

有一个或多个生产者生产某种类型的数据，并放置在缓冲区中；有一个消费者从缓冲区中取数据，每次取一项；

任何时候只有一个主体（生产者或消费者）可以访问缓冲区。要确保缓存满时，生产者不会继续添加，缓存为空时，消费者不会从中取数据

实现代码：

- **当缓冲无限大时**（二元信号量，对应图 5.10；信号量，对应图 5.11）

- **当缓冲有限时**（信号量，对应图 5.13；管程，对应图 5.16；消息传递，对应图 5.21）

2）读者/写者问题

有一个由多个进程共享的数据区，一些进程只读取这个数据区中的数据，一些进程只往数据区中写数据；此外还满足以下条件：

- 任意多的读进程可以同时读
- 一次只有一个进程可以写
- 如果一个进程正在写，禁止所有读；

实现代码：

- **读优先**：只要至少有一个读进程正在读，就为进程保留对这个数据区的控制权（信号量，对应图 5.22）
- **写优先**：保证当有一个写进程声明想写时，不允许新的读进程访问该数据区（信号量，对应图 5.23）

2.死锁

死锁定义：一组进程中的每个进程都在等待某个事件，而只有在这种进程中的其他被阻塞的进程才可以触发该事件，这时就称这组进程发生死锁

假设两个进程的资源请求和释放序列如下：

下图是相应的**联合进程图**，显示了进程竞争资源的进展情况：

敏感区域：路径 3，4 进入的区域。敏感区域的存在依赖于两个进程的逻辑关系。然而，如果另个进程的交互过程创建了能够进入敏感区的执行路径，那么死锁就必然发生

死锁问题中的资源分类

- **可重用资源**：一次只能供一个进程安全地使用，并且不会由于使用而耗尽的资源（包括处理器、I/O 通道、内外存、设备等）
- **可消耗资源**：可以被进程创建和消耗的资源。通常对某种类型可消耗资源的数目没有限制，一个无阻塞的生产进程可以创建任意数目的这类资源（包括中断、信号、消息和 I/O 缓冲中的信息）

资源分配图

- 进程到资源：进程请求资源但还没得到授权
- 资源到进程：请求资源已被授权
- 资源中的“点”：表示该类资源的一个实例

2.1 死锁的条件

死锁条件：

1. **互斥**：一次只有一个进程可以使用一个资源
2. **占有且等待**：当一个进程等待其他进程时，继续占有已经分配的资源
3. **不可抢占**：不能强行抢占进程已占有的资源
4. **循环等待**：存在一个封闭的进程链，使得每个进程至少占有此链中下一个进程所需的一个资源

条件 1~3 是死锁的必要条件，条件 4 是前 3 个条件的潜在结果，即假设前 3 个条件存在，可能发生的一系列事件会导致不可解的循环等待。这个不可解的循环等待实际上就是死锁的定义。之所以不可解是因为有前 3 个条件的存在。因此，4 个条件连在一起构成了死锁的充分必要条件

2.2 死锁预防

死锁预防是通过约束资源请求，使得 4 个死锁条件中的至少 1 个被破坏，从而防止死锁发生

- **间接的死锁预防（防止死锁条件 1~3）**
  - **预防互斥**：一般来说，不可能禁止

  - **预防占有且等待**：可以要求进程一次性地请求所有需要的资源，并且阻塞进程直到所有请求都同时满足。这种方法在两个方面是低效的：1）为了等待满足其所有请求的资源，进程可能被阻塞很长时间。但实际上只要有一部分资源，就可以继续执行；2）分配的资源有可能有相当长的一段时间不会被使用，且在此期间，这些资源不能被其它进程使用；除此之外，一个进程可能事先并不会知道它所需要的所有资源
  - **预防不可抢占**：有几种方法：1）如果占用某些资源的进程进一步申请资源时被拒，则释放其占用的资源；2）如果一个进程请求当前被另一个进程占有的一个资源，操作系统可以抢占另一个进程，要求它释放资源(方法 2 只有在任意两个进程优先级不同时，才能预防死锁)；此外，通过预防不可抢占来预防死锁的方法，只有在资源状态可以很容易保存和恢复的情况下才实用

- **直接的死锁预防（防止死锁条件 4）**
  - **预防循环等待**：可以通过定义资源类型的线性顺序来预防，如果一个进程已经分配到了 R 类型的资源，那么它接下来请求的资源只能是那些排在 R 类型之后的资源；这种方法可能是低效的，会使进程执行速度变慢，并且可能在没有必要的情况下拒绝资源访问

都会导致低效的资源使用和低效的进程运行

2.3 死锁避免

死锁避免允许 3 个必要条件，但通过明智选择，确保永远不会到达死锁点
由于需要对是否会引起死锁进行判断，因此死锁避免需要知道将来的进程资源请求的情况
2 种死锁避免的方法：

1. **进程启动拒绝**：如果一个进程的请求会导致死锁，则不启动此进程
2. **资源分配拒绝**：如果一个进程增加的资源请求会导致死锁，则不允许此分配

1）进程启动拒绝

一个有 n 个进程，m 种不同类型资源的系统。定义如下向量和矩阵：

从中可以看出以下关系成立：

对于进程 n+1，仅当对所有 j，以下关系成立时，才启动进程 n+1：

2）资源分配拒绝(银行家算法)

当进程请求一组资源时，假设同意该请求，从而改变了系统的状态，然后确定其结果是否还处于安全状态。如果是，同意这个请求；如果不是，阻塞该进程直到同意该请求后系统状态仍然是安全的

- 安全状态：至少有一个资源分配序列不会导致死锁(即所有进程都能运行直到结束)
- 不安全状态：非安全的一个状态(所有分配序列都不可行)

下图为一个安全序列：

下图为一个不安全序列：

这个不安全序列并不是一个死锁状态，仅仅是有可能死锁。例如，如果 P1 从这个状态开始运行，先释放一个 R1 和 R3，后来又再次需要这些资源，一旦这样做，则系统将到达一个安全状态

优点

- 不需要死锁预防中的抢占和回滚进程，并且比死锁预防的限制少。比死锁预防允许更多的并发

缺点

- 必须事先声明每个进程请求的最大资源
- 所讨论的进程必须是无关的，也就是说，他们执行的顺序必须没有任何同步要求的限制
- 分配的资源数目必须是固定的

- 在占有资源时，进程不能退出

2.4 死锁检测

死锁检测不限制资源访问或约束进程行为。只要有可能，被请求的资源就被分配给进程。操作系统周期性地执行一个算法检测死锁条件 4(循环等待)

常见死锁检测算法

这种算法的策略是查找一个进程，使得可用资源可以满足该进程的资源请求，然后假设同意这些资源，让该进程运行直到结束，再释放它的所有资源。然后算法再寻找另一个可以满足资源请求的进程
这个算法并不能保证防止死锁，是否死锁要取决于将来同意请求的次序，它所做的一切是确定当前是否存在死锁

恢复

一旦检测到死锁，就需要某种策略以恢复死锁，有下列方法(复杂度递增)：

- 取消所有死锁进程（操作系统最常用）
- 回滚每个死锁进程到前面定义的某些检测点
- 连续取消死锁进程直到不再存在死锁（基于某种最小代价原则）
- 连续抢占资源直到不再存在死锁（基于代价选择，每次抢占后需重新调用算法检测，被抢占的进程需回滚）

2.5 死锁“预防/避免/检测”总结

2.6 经典问题(哲学家就餐问题)

就餐需要使用盘子和两侧的叉子，设计一套算法以允许哲学家吃饭。算法必须保证互斥（没有两位哲学家同时使用同一把叉子），同时还要避免死锁和饥饿

方法一(基于信号量，可能死锁)：每位哲学家首先拿起左边的叉子，然后拿起右边的叉子。吃完面后，把两把叉子放回。如果哲学家同时拿起左边的叉子，会死锁

方法二(基于信号量，不会死锁)：增加一位服务员，只允许 4 位哲学家同时就座，因而至少有一位哲学家可以拿到两把叉子

方法三(基于管程，不会死锁)：和方法一类似，但和信号量不同的是，因为同一时刻只有一个进程进入管程，所以不会发生死锁

3.UNIX 并发机制

UNIX 为进程间的通信和同步，提供了下列几种重要的通信机制：

- **提供进程间传递数据的方法**
  - 管道
  - 消息
  - 共享内存
- **触发其它进程的行为**
  - 信号量
  - 信号

3.1 管道

管道是一个环形缓冲区，允许两个进程以生产者/消费者的模型进程通信

- 写管道：当一个进程试图写管道时，如果有足够的空间，则写请求被立即执行，否则进程被阻塞
- 读管道：当一个进程试图读管道时，如果读取字节数多于当前管道中的字节数，进程被阻塞

操作系统强制实施互斥，即一次只能有一个进程可以访问管道

两类管道

- 命名管道：共享的进程可以不相关
- 匿名管道：只有父子关系的进程才能共享

3.2 消息

每个进程都有一个关联的消息队列，功能类似于信箱

- 发送消息：发送者指定发送消息的类型。试图给一个满队列发送时进程会被阻塞
- 接收消息：接收者可以按先进先出的顺序接收信息；也可以按类型接收；试图从空队列读消息时，进程会被阻塞，试图读取某一类型消息，但是该类型消息不存在时，不会阻塞进程

3.3 共享内存

共享内存是 UNIX 提供的进程间通信手段中速度最快的一种。共享内存是虚存中由多个进程共享的一个公共内存块。互斥约束不属于共享内存机制的一部分，但必须由使用共享内存的进程提供

3.4 信号量

UNIX System V 中的信号量**系统调用**是对 semWait 和 semSignal 原语的推广

3.5 信号

信号是用于向一个进程通知发生异步事件的机制。类似于硬件中断，但没有优先级，即内核平等地对待所有的信号。对于同时发送的信号，一次只给进程一个信号，而没有特定的次序

4.Linux 内核并发机制

Linux 包含了在其他 UNIX 系统中出现的所有并发机制，其中包括管道，消息，共享内存和信号。除此之外，还包括：

- 原子操作
- 自旋锁
- 信号量
- 屏障

4.1 原子操作

Linux 提供了一组操作以保证对变量的原子操作。这些操作能够用来避免简单的竞争条件。原子操作执行时不会被打断或被干涉

- 在单处理器上：线程一旦启动原子操作，则从操作开始到结束的这段时间内，线程不能被中断
- 在多处理器上：原子操作所针对的变量是被锁住的，以免被其他的进程访问，直到原子操作执行完毕

Linux 中定义了 2 种原子操作：

- 针对整数变量的整数操作：定义了一个特殊的数据类型 atomic_t，原子整数操作仅能用在这个数据类型上，其它操作不允许用在这个数据类型上
- 针对位图中某一位的位图操作：操作由指针变量指定任意一块内存区域的位序列中的某一位。因此没有和原子整数操作中 atomic_t 等同的数据类型

Linux 原子操作表：

4.2 自旋锁

自旋锁是 Linux 中包含临界区最常见的技术。同一时刻，只有一个线程能获得自旋锁。其它任何企图获得自旋锁的进程将一直进行尝试（忙等），直到获得了该锁

- 普通自旋锁
- 读者-写者自旋锁：允许多个线程同时以只读方式访问同一数据结构，只有当一个线程想要更新时，才会互斥访问

自旋锁操作表：

4.3 信号量

内核的信号量不能通过系统调用直接被用户程序访问。内核信号量是作为内核内部函数实现的，比用户可见的信号量更高效

- 二元信号量：在 Linux 中也称为互斥信号量 MUTEX
- 计数信号量
- 读者-写者信号量：允许多个并发的读者，仅允许一个写者。事实上，对于读者使用的是一个计数信号量，而对于写者使用的是一个二元信号量

Linux 提供 3 种版本的 down 操作：

1. down：对应于传统的 semWait 操作
2. down_interruptible：允许因 down 操作而被阻塞的线程在此期间接收并相应内核信号
3. down_trylock：可在不被阻塞的同时获得信号量，如果信号量不可用，返回非 0 值，不会阻塞

信号量操作表：

4.4 屏障

屏障用于保证指令执行的顺序。如，rmb()操作保证了之前和之后的代码都没有任何读操作会穿过屏障

对于屏障操作，需要注意 2 点：

1. 屏障和机器指令相关，也就是装载和存储指令（高级语言 a=b 会产生 2 个指令）
2. 编译方面，屏障操作指示编译器在编译期间不要重新排序指令；处理器方面，屏障操作指示流水线上任何屏障前的指令必须在屏障后的指令开始执行之前提交

barrier()操作是 mb()操作的一个轻量版本，它仅仅控制编译器的行为

屏障操作表：

第六章.内存管理

- **单道程序设计中**：内存被划分为两部分，一部分供操作系统使用(驻留监控程序、内核)，一部分供当前正在执行的程序使用
- **多道程序设计中**：必须在内存中进一步细分“用户”部分，以满足多个进程的要求，细分的任务由操作系统动态完成，称为内存管理

内存管理的需求

- **重定位**：程序在从磁盘换入内存时，可以被装载到内存中的不同区域
- **保护**：处理器必须保证进程以外的其它进程不能未经授权地访问该进程的内存单元
- **共享**：任何保护机制都必须具有一定灵活性，以允许多个进程访问内存的同一部分
- **逻辑组织**
- **物理组织**

内存管理中的地址

- **逻辑地址**：指与当前数据在内存中的物理分配地址无关的访问地址，执行对内存访问前必须转换成物理地址
- **相对地址**：逻辑地址的一个特例，是相对于某些已知点（通常是程序开始处）的存储单元
- **物理地址(绝对地址)**：数据在内存中的实际位置
- **虚拟地址**：虚拟内存中的逻辑地址

内存管理单元(MMU)：CPU 中的一个模块，将虚拟地址转换成实际物理地址

1.内存管理中的数据块

- **页框**：内存中一个固定长度的块
- **页**：二级存储(如磁盘)中一个固定长度的数据块
- **段**：二级存储中一个变长的数据块

2.内存分区

2.1 固定分区

系统生成阶段，**内存被划分成许多静态(大小，容量固定不变)分区**，两种固定分区：

- **分区大小相等**
- **分区大小不等**

放置策略：

- **对于分区大小相等的固定分区**
  - 只要存在可用分区，就可以分配给进程
- **对于分区大小不等的固定分区**
  - **每个进程分配到能容纳它的最小分区**：每个分区维护一个队列（较多小进程时，大分区会空闲）
  - **每个进程分配到能容纳它的最小可用分区**：只需一个队列

存在内部碎片；活动进程数固定

2.2 动态分区

并不进行预先分区，**在每次需要为进程分配时动态划分**

外部碎片（随着时间推移，内存中产生了越来越多”空洞“）：

可以使用压缩解决外部碎片，但是非常耗时

放置算法：由于压缩十分耗时，因而需要巧妙地把进程分配到内存中，塞住内存中的”洞“

- **最佳适配**：选择与要求大小最接近的块（通常性能最差，尽管每次浪费的空间最小，但结果却使得内存中很快产生许多碎片）
- **首次适配**：选择大小足够的第一个块（不仅最简单，通常也是最好、最快的；容易在首部产生碎片）

- **下次适配**：从上次放置的位置起，第一个大小足够的块（比首次适配差，常常会在尾部产生碎片）

维护复杂，且会产生外部碎片

2.3 伙伴系统

内存最小块和最大块的尺寸是 M 和 L。在为一个进程分配空间时，如果需要的内存大于 L/2，则分配 L 的内存，否则，将大小为 L 的块分成两个 L/2 的块，继续上述步骤；如果两个相邻的块（伙伴）都未分配出去（如前面的进程释放后），则将其合并

下图为一个伙伴系统的例子：

伙伴系统是一种折中方案，克服了固定分区和动态分区方案的缺陷。但在当前操作系统中，基于分页和分段机制的虚拟内存更好。伙伴系统在并行系统中有很多应用

2.4 分区中的地址转换

逻辑地址->物理地址的转换如下

- **基址寄存器**：被载入程序在内存中的起始地址
- **界限寄存器**：程序的终止位置

这种转换方式适用于程序运行时，被加载到内存中连续区域的情况。对于分页和分段，由于一个程序可以加载到内存的不同区域，所以需要使用另外的机制进行转换

3.分页

内存被划分为大小固定的块，且块相对比较小，每个进程也被分成同样大小的小块，那么进程中称为页的块可以指定到内存中称为页框的可用块。**和固定分区的不同在于：一个程序可以占据多个分区，这些分区不要求连续**

使用分页技术在内存中每个进程浪费的空间，仅仅是最后一页的一小部分（内部碎片）

3.1 分页中的地址转换

由于进程的页可能不连续，因此仅使用一个简单的基址寄存器是不够的，操作系统需要为每个进程维护一个**页表**。页表项是进程每一页与内存页框的映射

4.分段

段有一个最大长度限制，但不要求所有程序的所有段长度都相等。分段**类似于动态分区，区别在于：一个程序可以占据多个不连续的分区**

分段同样会产生外部碎片，但是进程被划分成多个小块，因此外部碎片也会很小

4.1 分段中的地址转换

由于进程的段可能不连续，因此也不能仅靠一个简单的基址寄存器，地址转换通过**段表**实现。由于段的大小不同，因此段表项中还包括段的大小

如果偏移大于段的长度，则这个地址无效

5.内存安全

5.1 缓冲区溢出

缓冲区溢出是指输入到一个缓冲区或者数据保存区域的数据量超过了其容量，从而导致覆盖了其它区域数据的状况。攻击者造成并利用这种状况使系统崩溃或者通过插入特制的代码来控制系统

被覆盖的区域可能存有其它程序的变量、参数、类似于返回地址或指向前一个栈帧的指针等程序控制流数据。缓冲区可以位于堆、栈或进程的数据段。这种错误可能产生如下后果：

1. **破坏程序的数据**
2. **改变程序的控制流，因此可能访问特权代码**

最终很有可能造成程序终止。当攻击者成功地攻击了一个系统之后，作为攻击的一部分，程序的控制流可能会跳转到攻击者选择的代码处，造成的结果是被攻击的进程可以执行任意的特权代码（比如通过判断输入是否和密码匹配来访问特权代码，如果存在缓冲区漏洞，非法输入导致存放“密码”的内存区被覆盖，从而使得“密码”被改写，因此判断为匹配进而获得了特权代码的访问权）

缓冲区溢出攻击是最普遍和最具危害性的计算机安全攻击类型之一

5.2 预防缓冲区溢出

广义上分为两类：

- 编译时防御系统，目的是强化系统以抵御潜伏于新程序中的恶意攻击
- 运行时预防系统，目的是检测并终止现有程序中的恶意攻击

尽管合适的防御系统已经出现几十年了，但是大量现有的脆弱的软件和系统阻碍了它们的部署。因此运行时防御有趣的地方是它能够部署在操作系统中，可以更新，并能为现有的易受攻击的程序提供保护

第七章.虚拟内存

一个进程只能在内存中执行，因此这个存储器称为实存储器，简称实存。但是程序员或用户感觉到的是一个更大的内存，通常它被分配在磁盘上，称为虚拟内存，简称虚存

虚存使得程序不必完全载入内存才能运行，每次可以只有部分驻留在内存中。如果处理器访问一个不在内存中的逻辑地址，则产生一个中断，说明产生了内存访问故障。操作系统把被中断的进程置于阻塞态。为了能继续执行这个进程，操作系统必须把包含引发访问故障的逻辑地址的进程块读入内存。为此，操作系统产生一个磁盘 I/O 读请求。在此期间，可以调度另一个进程运行。一旦需要的块被读入内存，则产生一个 I/O 中断，操作系统把由于缺少该块而被阻塞的进程置为就绪态

不必将程序完全载入即可运行使得程序可以比实际内存更大

系统抖动：如果一个块正好在将要被用到之前换出，操作系统就不得不很快把它取回来。太多这类操作会导致一种称为系统抖动的情况，处理器大部分时间都用于交换块，而不是执行指令

1. 分页

1.1 页表

- 每个进程都有自己的页表
- 由于进程某些页可能不在内存中，所以页表项中有一位表示该页是否在内存中
- 页表项有一位表示该页(从上次载入)是否已经被修改
- 页表的长度可以基于进程的长度而变化，因此不能在寄存器中保存它（对于占据大量虚存空间的程序，其页表很大，因此页表通常保存在虚存中，因此页表也服从分页管理）
- 一个程序正在运行时，页表至少有一部分必须在内存中

1.2 一级分页系统中的地址转换

一级分页系统中的虚拟地址和页表项：

地址转换：

1.3 两级分页系统中的地址转换

两级页表结构（假设页大小为 4KB，每个页表项大小为 4B）：

地址转换：

1.4 倒排页表

一级和两级分页系统中的页表存在一个缺陷：页表的大小与虚拟地址空间的大小成正比

一种替代方法是使用一个倒排页表，其机构如下：

页表结构之所以称为”倒排“，是因为它使用页框号而非虚拟页号来索引页表项

1.5 转换检测缓冲区(TLB)

原则上，每次虚拟内存访问可能引起两次物理内存访问：一次取相应的页表项，一次取需要的数据。因此，简单的虚拟内存方案会导致内存访问时间加倍

TLB 保存在高速缓冲存储器中，它记录了最近用到过的页表项。给定一个虚拟地址，处理器首先检查 TLB： * 如果需要的页表项在其中，则检索页框号并形成实地址 * 如果未找到需要的页表项，则处理器用页号检索进程页表，并检查相应的页表项。 * 如果”存在位“置位，则页在内存中，处理器从页表项中检索页框号形成实地址。并更新 TLB * 如果”存在位”没置位，表示需要的页不在内存中，这时发生**缺页中断**，因此离开硬件作用范围，调用操作系统，操作系统负责载入所需要的页，并更新页表

虚拟机制必须与高速缓存系统进行交互，一个虚拟地址通常为页号、偏移量的形式。首先，内存系统查看 TLB 中是否存在匹配的页表项，如果存在，通过把页框号和偏移量组合起来产生实际地址（物理地址）；如果不存在，则从页表中读取页表项。一旦产生了一个由标记和其余部分组成的实地址，则查看高速缓存中是否存在包含这个字的块。如果有，把它返回给 CPU；如果没有，从内存中检索这个字

2. 分段

- 每个进程都有一个唯一的段表。
- 进程可能只有一部分段在内存中，所以段表项中有一位表明相应段是否在内存中
- 段表项有一位修改位表明相应段从上一次载入起是否被改变
- 根据进程大小，段表长度可变，而无法在寄存器中保存

2.1 分段系统中的地址转换

2.2 保护和共享

分段有助于实现保护与共享机制。由于**每个段表项包括一个长度和一个基地址**，因而程序不会不经意地访问超出该段的内存单元。为实现共享，一个段可能在多个进程的段表中被引用

3. 段页式

- 分页对程序员是透明的，它消除了外部碎片，从而可以更有效地使用内存
- 分段对程序员是可见的，它具有处理不断增长的数据结构的能力以及支持共享和保护的能力

分段通常对于程序员可见，并且作为组织程序和数据的一种方便手段提供给程序员。一般情况下，程序员或编译器会把程序和数据指定到不同的段。为了实现模块化程序设计的目的，程序或数据可能进一步分成多个段。这种方法最不方便的地方是程序员必须清楚段的最大长度限制

可以将分页和分段结合，即段页式

在段页式系统中，用户的地址空间被程序员划分成许多段。每个段依次划分成许多固定大小的页，页的长度等于内存中的页框大小。如果某一段的长度小于一页，则该段只占据一页。从程序员角度看，逻辑地址仍然由段号和段偏移量组成；从系统角度看，段偏移量可视为指定段中的一个页号和页偏移量

3.1 段页式系统中的地址转换

4. 内存管理中的相关策略

操作系统的内存管理设计取决于三个基本方面的选择：

1. 是否使用虚存技术
2. 是使用分页还是使用分段，或者是二者组合
3. **为各种存储管理特征采用的算法**

4.1 读取策略

读取策略确定一个页何时取入内存

- **请求分页**：只有当访问到某页中的一个单元时才将该页取入内存
- **预先分页**：读取的页并不是缺页中断请求的页，如果一个进程的页被连续存储在辅存中，则一次读取许多连续的页

4.2 放置策略

放置策略决定一个进程块驻留在实存中的什么地方

- 在一个纯粹的分段系统中，放置策略并不是重要的设计问题（诸如最佳适配、首次适配等都可供选择）
- 对于纯粹的分页系统或段页式系统，如何放置通常没有关系，因为地址转换硬件和内存访问硬件可以以相同的效率为任何页框组合执行它们的功能

4.3 置换策略

置换策略决定在必须读取一个新页时，应该置换内存中的哪一页

页框锁定：如果一个页框被锁定，当前保存在该页框中的页就不能被置换。大部分操作系统内核和重要的控制结构就保存在锁定的页框中。此外，I/O 缓存区和其它对时间要求严格的区域也可能锁定在内存的页框中

基本置换算法

- **最佳(OPT)**：置换下次访问距当前时间最长的那些页，该算法能导致最少的缺页中断（由于要求操作系统必须知道将来的事件，因此不可能实现，而是作为一种标准来衡量其它算法的性能）
- **最近最少使用(LRU)**：置换内存中上次使用距当前最远的页，LRU 性能接近于 OPT，但是难以实现（一种方法是为每一页添加一个最后一次访问的时间戳，但是开销较大）
- **先进先出(FIFO)**：把分配给进程的页框视为一个循环缓冲区，按循环的方式移动页。实现简单，但性能较差（隐含的逻辑是置换驻留在内存中时间最长的页，经常会出现部分程序或数据在整个程序的生命周期中使用频率都很高的情况，如果使用 FIFO 这些页会需要反复地被换入换出）。FIFO 还会产生当所分配的物理块数增大而页故障不减反增的异常现象，称为 **Belady 异常**
- **时钟(CLOCK)**：时钟策略是试图以较小的开销接近 LRU 性能的一种算法，最简单的时钟策略需要给每一页关联一个附加位，称为使用位。当某一页首次装入内存中时，该页的使用位设置为 1；当该页随后被访问到时，它的使用位也会被置为 1。当需要置换一页时，操作系统扫描缓冲区，以查找使用位被置为 0 的一个页框。每当遇到一个使用位为 1 的页框时，就将该位重新置为 0（**只有寻找置换页和发生置换时，指针会移动。如果当前需要访问的页在内存中，即使不是当前指针指向的页，也只是将被访问的页置为 1，而不发送指针移动，如下图右下角 CLOCK 策略中最后一次访问 2**）

4.4 驻留集管理

对于分页式的虚拟内存，在准备执行时，不需要也不可能把一个进程的所有页都读入内存。因此，操作系统必须决定读取多少页，即给特定的进程分配多大的内存空间。需要考虑以下几个因素：

- 分配给一个进程的内存越少，在任何时候驻留在内存中的进程数就越多（这就增加了操作系统至少找到一个就绪进程的可能）
- 如果一个进程在内存中的页数比较少，尽管有局部性原理，缺页率仍然相对较高
- 如果分配过多页，由于局部性原理，该进程的缺页率没有明显的变化

基于上述因素，通常采用两种策略：

1. **固定分配策略**：为一个进程在内存中分配固定数目的页框用于执行时使用，这个数目在最初加载时（创建进程时）决定（可以根据进程类型或程序员的需要确定）。一旦发生缺页中断，进程的一页必须被它所需要的页面置换
2. **可变分配策略**：允许分配给一个进程的页框在进程的生命周期中不断地变化。如果缺页中断多，则多分配一些；缺页中断少，适当减少分配。这种方法的难点在于要求操作系统评估活动进程的行为

置换范围

- **局部置换策略**：仅仅在产生缺页的进程的驻留页中选择
- **全局置换策略**：把内存中所有未被锁定的页都视为置换的候选页，而不管它们属于哪个进程

- **可变分配、全局范围**：发生缺页时，如果存在空闲页框，则使用空闲页框；否则在全局页框中选择置换
- **可变分配、局部范围**：不时评估进程的页框分配情况，增加或减少分配给它的页框

4.5 清除策略

清除策略与读取策略相反，它用于确定在何时将一个被修改过的页写回辅存，通常有 2 种选择

- **请求式清除**：只有当一页被选择用于置换时，才被写回辅存（可以减少写页，但意味着发生缺页中断的进程在解除阻塞之前必须等待两次页传送，这可能降低处理器的利用率）
- **预约试清除**：将被修改的多个页在需要用到它们占据的页框之前成批地写回辅存（并没有太大意义，因为这些页中大部分常常会在置换之前又被修改，辅存传送能力有限，不应该浪费在不太需要的清除操作上）

比较好的方法是结合页缓冲技术

4.6 加载控制

加载控制决定驻留在内存中的进程数目，称为系统并发度
并发度太低会导致处理器利用率不高，并发度太高会发生系统抖动

第八章.单处理器调度

所谓单处理器调度，指的是单个处理器上的调度。主要是单处理器上多道程序设计系统的进程调度，多道程序设计系统中，内存可以同时驻留多个进程

1.进程调度类型

调度类型和进程状态转换：

长程调度决定哪一个程序可以进入系统中处理，因此控制着系统的并发度
在批处理系统或者操作系统的批处理部分，新提交的作业被发生到磁盘，并保存在一个批处理队列中。
在长程调度程序运行的时候，从队列中创建相应的进程。这里涉及两个决策：

- 调度程序决定何时操作系统接纳一个进程或者多个进程
- 调度程序决定接收哪个作业或哪些作业，并将其转变成进程

长程调度执行频率较低，并且仅仅是粗略地决定是否接受新进程及接受哪一个
该章剩余内容主要关注短程调度，即处理器选择一个进程执行时的调度决策。短程调度执行得最频繁，并且精确地决定下一次执行哪个进程

2.调度算法

2.1 短程调度准则

调度算法的设计需要考虑如下方面（以下为从一种维度的划分）：

- 面向用户的准则：延迟（侧重于用户）
- 面向系统的准则：效果、利用率、吞吐量（侧重于系统）

2.2 优先级调度

- 每个进程被指定一个优先级，调度程序总是优先选择具有较高优先级的进程
- 低优先级进程可能饥饿

2.3 选择调度策略

- **周转时间**：等待时间 + 服务时间
- **归一化周转时间**：周转时间/服务时间

1）先来先服务(FCFS)：

- 非抢占
- 对短进程不利（相对于 I/O 密集型的进程，更利于处 CPU 密集型的进程）；一种改进是与优先级结合，每个优先级一个队列，同一队列内部使用 FCFS

2）轮转(时间片)：

- 抢占
- 以时间片为周期产生时钟中断，切换运行

- 主要设计问题是时间片的长度，太短时间片会带来频繁的进程上下文切换开销。时间片过长(比最长进程还长)，算法就退化成了 FCFS

3）最短进程优先(SPN)：

- 非抢占
- 每次调度选择(所需总)处理时间最短的进程。可能饥饿长进程
- 难点在于需要估计每个进程所需要的处理时间

4）最短剩余时间(SRT)：

- 抢占
- 每次选择剩余处理时间最少的进程，可能饥饿长进程
- 也需要估计每个进程所需的处理时间。同时，维护过去的服务时间也会增加开销

5）最高响应比(HRRN)：

- 非抢占
- 调度选择归一化周转时间最大的进程，归一化时间越大说明进程“年龄”越大。当偏向短作业时（小分母产生大比值），长进程由于得不到服务，等待的时间不断增加，从而增大了比值，最终在竞争中可以胜出
- 同样需要预估每个进程所需的处理时间

6）反馈法：

- 抢占
- 反馈法为了解决 SPN、SPT 和 HRRN 必须预估进程所需处理时间的问题(不能获得剩余执行时间就关注已经执行了的时间)。通过处罚运行时间较长进程的方法来偏向短进程。进程每被抢占一次(说明进程还未运行完，可能是个长进程)，就移入更低优先级的队列。在这种机制下，短进程在降级过多前就能运行完，长进程会一直降级，如果已经处于最低级队列，则再次被抢占后返回该队列。
- 这种方法的问题是长进程的周转时间可能惊人的增加，导致饥饿，一种方法是可以增加低优先级队列中进程运行的时间片，但仍可能饥饿，还有一种方法是如果在低级队列中时间过长，提升到高优先级队列中

调度策略对比总结：

性能比较

调度策略的性能是选择调度策略的一个关键因素。但是由于相关的性能取决于各种各样的因素，包括各种进程的服务时间分布、调度的效率、上下文切换机制、I/O 请求的本质和 I/O 子系统的性能，因而不可能得到明确的比较结果

2.4 调度实例分析

给出如下进程以及到达时间和服务时间：

使用各种调度策略：

第九章.I/O 管理与磁盘调度

1.I/O 缓冲

缓冲技术：在输入请求发出之前就开始执行输入传送，并且在输出请求发出一段时间之后才开始执行输出传送，这项技术称为缓冲

两类 I/O 设备：

- **面向块的 I/O 设备**：将信息保存在块中，块的大小通常是固定的，传送过程中一次传送一块
- **面向流的 I/O 设备**：以字节流的方式输入/输出数据，没有块结构

1.1 单缓冲

对于面向块的 I/O 设备：

- 输入传送的数据被放到系统缓冲区中。当传送完成时，进程把该块移到用户空间，并立即请求另一块
- 相对于无缓冲的情况，这种方法通常会提高系统速度。用户进程可以在下一数据块读取的同时，处理已读入的数据块。由于输入发生在系统内存中而非用户进程内存中，因此操作系统可以将该进程换出

对于面向流的 I/O 设备：

单缓冲方案能以每次传送一行的方式或者每次传送一个字节的方式使用

1.2 双缓冲(缓冲交换)

分配 2 个缓冲区。在一个进程往一个缓冲区中传送数据(从这个缓冲区中取数据)的同时，操作系统正在清空(或者填充)另一个缓冲区

1.3 循环缓冲

双缓冲方案可以平滑 I/O 设备和进程之间的数据流。如果关注的焦点是某个特定进程的性能，那么常常会希望相关 I/O 操作能够跟得上这个进程。如果该进程需要爆发式地执行大量的 I/O 操作，仅有双缓冲就不够了，在这种情况下，通常使用多于两个的缓冲区方案来缓解不足

1.4 I/O 缓冲的作用

I/O 缓冲是用来平滑 I/O 需求的峰值的一种技术，但是当进程的平均需求大于 I/O 设备的服务能力时，缓冲再多也不能让 I/O 设备与这个进程一直并驾齐驱。即使有多个缓冲区，所有的缓冲区终将会被填满，进程在处理完每一大块数据后不得不等待。但是，在多道程序设计环境中，当存在多种 I/O 活动和多种进程活动时，缓冲是提高操作系统效率和单个进程性能的一种方法

2.磁盘调度

2.1 磁盘性能参数

- **寻道时间**：磁头定位到磁道所需的时间
- **旋转延迟**：选好磁道后，磁头到达扇区开始位置的时间
- **存取时间**：寻道时间+旋转延迟
- **传输时间**：磁头定位到扇区开始位置后，数据读写的时间
- **排队时间**

2.2 磁盘调度算法

在多道程序环境中，操作系统为每个 I/O 设备维护一个请求队列。因此对一个磁盘，队列中可能有来自多个进程的许多 I/O 请求。如果随机地从队列中选择请求，那么磁道完全是被随机访问的，这种情况下性能最差。**随机调度**可用于与其他调度算法进行对比

1）先进先出(FIFO)

- 按顺序处理队列中的请求
- 如果有大量进程竞争一个磁盘，这种算法在性能上往往接近于随机调度

2）优先级

- 这种方法不会优化磁盘利用率，但可以满足操作系统的其它目标
- 通常比较短的批作业和交互作业的优先级比较高。长作业可能饥饿
- 可能会导致部分用户采用对抗手段：把作业分成小块，以回应系统的这种策略。对于数据库系统，这种算法往往性能较差

3）最短服务时间优先(SSTF)

- 选择使磁头臂从当前位置开始移动最少(最小寻道时间)的磁盘 I/O 请求
- 但是，总是选择最小寻道时间并不能保证平均寻道时间最小，不过能提供比 FIFO 更好的性能
- 磁头臂可以沿两个方向移动

3）SCAN

- 运行类似电梯。磁头臂沿某一方向移动，并在途中满足所有未完成请求，直到到达最后一个磁道，或者该方向上没有更多请求。接着反转服务方向
- 偏向接近最靠里或最靠外的磁道的请求，并且偏向最近的请求，可能发生饥饿

4）C-SCAN

- 沿某个方向的扫描结束后，返回到相反方向的末端，再次扫描
- 减少了新请求的最大延迟
- 可能饥饿

5）N-step-SCAN(N 步扫描)

SSTF、SCAN 和 C-SCAN 可能在一段很长时间内磁头臂都不会移动(比如一个或多个进程对一个磁道有较高的访问速度，通过重复的请求这个磁道垄断整个设备)，从而饥饿其它请求

- 把请求队列分成长度为 N 的子队列，每一次用 SCAN 处理一个子队列。在处理一个子队列时，新请求必须添加到其它某个队列中
- 对于比较大的 N 值，性能接近 SCAN；当 N=1 时，实际上就是 FIFO

2.3 磁盘调度算法比较

假设有一些 I/O 请求，需问这些磁道：55、58、39、18、90、160、150、38、184
使用不同磁盘调度算法的结果如下：

3.磁盘高速缓存

一个磁盘高速缓存是内存中为磁盘扇区设置的一个缓冲区，它包含有磁盘中某些扇区的副本。当出现一个请求某一特定扇区的 I/O 请求时，首先进行检查，以确定该扇区是否在磁盘高速缓存中。如果在，则该请求可以通过这个高速缓存来满足；如果不在，则把请求的扇区从磁盘读到磁盘高速缓存中