第一章

1.1 操作系统的功能

1.1.1 用户视角

- 用户使用方便，不在乎性能利用；
- 优化资源利用率；
- 工作站，与其他服务器相连，二者兼顾。

1.1.2 系统视角

- 操作系统看做**资源分配器**；
- **控制程序**管理用户程序执行；

1.1.3 定义

- 是一直运行在计算机上的程序（内核）；

1.2 计算机系统的组成

1.3 体系结构

1.3.2 多处理器系统

- 有优点：增加吞吐量、规模经济、增加可靠性
- 两类， 对称处理（每个处理器都参与完成系统的所有任务）和非对称处理（各个处理器处理各自任务）；
- 对称集群更为高效；

1.4 操作系统

- **多道程序设计**安排作业使得CPU总有一个执行作业，从而提高利用率；
- 同时在内存中保存多个作业，首先保存在磁盘的作业池上；
- **分时系统（多任务）**为多道程序设计额的延申，通过切换作业来执行多个作业，并且用户在运行时可与之交互；
- 该系统允许多个用户同时共享一台计算机；
- 作业调度，任务调度；

1.5 操作系统的执行

- 操作系统由**中断驱动**，事件总是由中断或陷阱引起的；

1.5.1 多重模式

- 至少需要用户模式和内核模式（系统模式，特权模式）；
- 模式位：内核0，用户1；
- 防止操作系统和用户程序收到错误用户程序的影响，具体为将可能引起损害的指令作为特权指令，只有在内核模式下才允许执行，如果在用户模式下试图执行，则认为非法，并以陷阱形式通知OS；

1.5.2 定时器

- 指定周期后产生中断；

1.6 进程管理

- **进程**是系统的工作单元，且并发执行；
- 操作系统负责管理：
    - CPU上调度进程线程；
    - 创建和删除进程；

- 挂起和重启进程；
- 提供同步机制；
- 提供通信机制。

1.7 内存管理

- 操作系统负责
  - 记录哪些部分在被使用以及被谁使用；
  - 决定那些进程会被调入或调出内存；
  - 根据需要分配和释放内存空间。

1.8 存储管理

1.8.1 文件系统管理

- 操作系统负责
  - 创建和删除文件
  - 创建和删除目录
  - 提供文件和目录的操作原语；
  - 映射文件到外存；
  - 备份文件到稳定的存储介质。

1.8.2 大容量存储管理

- 操作系统负责
  - 空闲空间管理
  - 存储空间分配；
  - 硬盘调度。

1.8.3 高速缓存

- 工作原理：信息通常保存在一个存储系统中，使用时会被复制到更快存储系统，即高速缓存；

1.9 保护与安全

- **保护**作为一种机制，用于控制进程和用户访问计算机系统的资源；
- **安全**防止系统不受内部或外部的攻击。
- 用户ID，组ID

第二章

2.1 操作系统服务

- 提供用户功能的服务：用户界面、程序执行、IO操作、文件系统操作、通信（共享内存或信息交换）、错误检测；
- 确保系统本身运行高效的服务：资源分配、记账、保护与安全。

2.2 用户与系统操作界面

2.2.1 命令解释程序

2.3 系统调用

- 应用开发人员根据应用编程接口（API）来设计程序，API为程序员调用实际的系统调用；
- 提供系统调用接口，截取API函数的调用，并据此调用系统调用。

2.4 系统调用的类型

2.4.1 进程控制

- 正常或异常停止执行；
- 加载和执行；
- 等待与响应；
- 提供系统调用允许一个进程锁定共享数据；

2.4.2 文件管理

- 创建和删除；
- 读写和重定位和关闭；
- 移动和复制；

2.4.3 设备管理

2.4.4 信息维护

2.4.5 通信

2.4.6 保护

2.5 系统程序

- 文件管理、状态信息、文件修改、程序语言支持、程序加载与执行、通信、后台服务。

2.6 操作系统的设计与实现

2.6.1 设计目标

- 用户目标和系统目标；
- 机制与策略相分离，机制决定如何做，策略决定做什么；

2.7 操作系统的结构

2.7.1 简单结构

- 优点：系统调用接口和内核通信开销非常小；
- 缺点：难以设计与实现；用户程序出错会导致整个系统崩溃。

2.7.2 分层方法

- 系统的模块化。
- 操作系统分成若干级，最低层为硬件，最高层为用户接口；
- 优点：简化了构造和调试，实现系统设计与实现的简化；
- 缺点：要合理定义各层；与其他方法相比效率较差。

2.7.3 微内核

- 从内核中删除不必要的部件，当作系统级与用户级的程序来实现；
- 主要功能是为客户端运行在用户控件中的各种服务提供通信（由消息传递实现）；
- 优点：便于扩展操作系统，易于移植，提供更好的安全性和可靠性；
- 缺点：增加系统功能带来的开销可能导致微内核自身性能受损。

2.7.4 模块

- 可加载的内核模块——内核有一组核心组件，无论启动或运行时，都可通过模块联入额外服务。
- 设计思想：内核提供核心服务，其他服务在内核运行时动态实现；
- 类似于上述，但更加灵活（任何模块可以相互调用）更加有效（无需调用消息传递进行通信）；

2.8 调试

- 大多数操作系系统将错误信息写道日志文件，或进行核心转储；

2.10 系统引导

- 加载内核以启动计算机的过程陈伟系统引导，**引导程序或引导加载程序**能够定位内核，并加载到内存已开始执行；
- 引导程序加载后就可遍历文件系统以寻找操作系统内核，将其加载到内存中开始执行，此时系统开始**运行。**

第三章

3.1 进程概念

- 所有CPU活动统一称为进程，包括批处理系统执行作业，分时系统使用用户程序或任务；

3.1.1 进程

- 进程包括：程序代码、当前活动（*程序计数器*、*处理器寄存器*）、堆栈（临时数据）、数据段（全局变量）、堆；
- 程序是**被动实体**，进程是**活动实体**；

3.1.2 进程状态

- new：正在创建；
- running：正在执行；
- waiting：等待某个事件；
- ready：等待分配处理器；
- terminated：进程已完成执行。

3.1.3 进程控制块（PCB）

- 包含：进程状态、程序计数器、CPU寄存器、CPU调度信息、内存管理信息、记账信息、IO状态信息；

3.1.4 线程

- 每个进程是只能进行单个执行线程的程序；

3.2 进程调度

- 为满足分时系统的目标，进程调度器选择一个可用进程到CPU上执行；

3.2.1 调度队列

- 等待运行的进程保存在就绪队列，这个队列的头结点由两个指针指向第一个和最后一个PCB块；

3.2.2 调度程序

- 对于批处理系统，提交的进程多余可以立即执行的，这些进程会被保存到大容量存储设备的缓冲池，以便以后执行；
- 长期调度程序或者作业调度程序从池中选择进程，短期或CPU则从准备执行的进程中选择；
- 短期调度程序执行频率高且速度快，长期则不频繁，控制*多道程序调度*；
- 长期选择*IO密集型和COU密集型*的合理组合——如果都是IO密集型，短期则无事可做，就绪队列基本为空；
- 中期调度程序将进程从内存中移除，从而降低多道程序程度，之后可被重新调入内存并在中断处继续执行，成为**交换**；
- 通过中期调度，进程可以换出再换入。

3.2.3 上下文切换

- 切换CPU到另一个进程需要保存当前进程状态和回复另一个进程状态，这个任务称为上下文切换。
- 进程上下文通常通过PCB表示，执行**状态保存**和**状态恢复**；

3.3 进程运行

3.3.1 进程创建

- 进程init的pid为1，作为所有进程的父进程；
- 进程创建新进程时有两种可能：
  - 父子并发执行；

    - 父进程等待子进程执行完；
- 新进程地址空间：
    - 子进程时父进程的复制品；
    - 子进程加载一个新程序
- fork()：子进程返回值为0，父进程返回值为子进程的进程标识符；
- 在fork之后，有个进程使用系统调用exec()，用新程序取代进程的内存空间，加载二进制文件到内存中，破坏原有内存内容；
- exec()覆盖进程地址空间，除非出现错误否则不会返回控制；
- 父进程可以调用wait()将自己移出就绪队列直到子进程终止；

3.3.2 进程终止

- **级联终止**：一个进程终止，则所有子进程也应该终止；
- **僵尸进程**：进程已经终止，其父尚未调用wait。所有进程终止时都会过渡到这种状态，但一旦wait被调用，僵尸进程的进程标识符和他在进程表中的条目就会被释放；
- **孤儿进程**：没有调用wait就终止。此时将init作为孤儿的父进程。init定期调用wait以便收集任何孤儿进程的退出状态，并释放孤儿进程标识符和进程表条目；

3.4 进程间通信(IPC)

- 一个进程能影响其他进程或被影响，称之为*协作的*；
- 允许协作的理由：
    - 信息共享
    - 计算加速
    - 模块化
    - 方便
- 两种模型
    - 共享内存：建立一块共享的内存区域，更快，更难实现；
    - 消息传递：在写作进程间交换消息（消息队列），对交换较少数量的数据很有用，但经常采用系统调用；
- 两种缓冲区：
    - 无界缓冲区：生产者无限产生；
    - 有界：满时生产者等待，空时消费者等待；

3.4.1 消息传递

- 直接通信：send receive 对称与非对称；
- 间接通信：使用邮箱，拥有共享邮箱时才可通信；
- send receive的实现：
    - 阻塞发送：持续发送直到被接受；
    - 非阻塞发送：发送消息立刻恢复；
    - 阻塞接收：接收进程阻塞直到有消息可用；
    - 非阻塞接收：接收进程收到一个有效消息或空消息；

3.6.2 套接字（socket）

- 通信的端点，通过网络通信的每对进程需要使用一堆套接字，即每个进程各有一个；
- 每个套接字由一个IP地址和一个端口号组成；
- 采用客户机-服务器架构，客户进程发出链接请求时主机为其分配一个端口（大于1024）；
- 虽然高效常用，但是较为低级，因为只允许在通信线程之间交换无结构的字节流，客户机服务器需要自己加上数据结构；
- IP地址127.0.0.1为特殊地址，称为回送（loopback），采用这个地址时引用自己，允许同一主机的客户机和服务器通过TCP/IP协议通信；

3.6.3 管道

3.6.3.1 普通管道（又称匿名管道）

- fd[0]读，fd[1]写；
- 普通管道需要进程有父子关系，意味着只可用于同一机器的进程间通信；
- 进程完成同行并终止时就不存在了；

3.6.3.2 命名管道

- 通信可以是双向的，父子关系不是必须的；

第四章

- 线程之间拥有相同全局变量和动态分配空间以及代码段，有自己的局部变量空间

4.1 优点

- 响应性：即使部分阻塞或者执行冗长操作，仍然可以继续执行；
- 资源共享：默认共享，允许一个应用程序在同一地址空间内有多个不同活动；
- 经济：创建和切换更加经济；
- 可伸缩性：多线程可在多处理核上并行运行。

4.3 多核编程

- 并行：同时执行多个任务；
- 并发：支持多个任务，不强调同时；
- 加速比：1 / (S + (1 - S) / N)，其中S为串行

4.3.1 编程挑战

- 识别任务、平衡、数据分割、数据依赖

4.3.2 并行类型

- 数据并行、任务并行

4.2 多线程模型

- 用户线程和内核线程之间的关系

4.2.1 多对一模型

- 映射多个用户级线程到一个内核线程；
- 线程管理由用户线程完成，效率更高，但如果系统调用阻塞则整体阻塞，并不能并行运行在多处理核上（**同一时间只有一个线程可以访问内核**）

4.2.2 一对一模型

- 提供比多对一更好的并发功能，也允许多个线程并行运行在多处理核上。
- 创建一个用户线程就要创建一个内核线程，开销过大从而限制了系统支持的线程数量；

4.2.3 多对多模型

- 开发人员可以创建任意多的用户线程，并且相应内核线程能在多处理器系统上并发执行，当一个线程阻塞系统调用时，内核可以调度另一个线程来执行；
- 双层模型：也允许某个用户线程绑定到一个内核线程。

4.4 线程库

- 提供创建和管理线程的API
- 两种方法
  - 在用户空间提供一个没有内核支持的库；
  - 实现由操作系统直接支持的内核级的一个库；
- 两种线程
  - 异步线程：父进程创建一个子进程后父线程就回复自身执行，这样二者会并发执行；
  - 同步线程：父进程恢复之前等待子进程的终止。
- pthread_create(), pthread_join(), 通过for循环加入多个线程;
- pthread_create()产生的线程可以改变主线程的值，因为所给的地址其实就在主线程的地址空间中

```
pthread_t workers[10];
for(int i = 0; i < 10; i++){
    pthread_join(workers[i], NULL);
}
```

4.5 隐式多线程

4.5.1 线程池

- 多线程服务器仍有以下问题：
  - 创建线程所需要的时间以及完成工作后线程被丢弃；
  - 无限制的线程可能耗尽系统资源。
- 主要思想：进程开始时创建一定数量的线程，加入池中等待工作，收到请求时唤醒池内一个线程并将服务请求传递。完成服务后回到线程池等待工作，若池中无可用线程，服务器将等待直到有空线程为止。
- 优点
  - 使用现有线程服务请求比等待创建一个更快；
  - 限制了任何时候可用线程的数量；
  - 允许采用不同策略运行任务；
- OpenMP遇到 `#pragma omp parallel for` 指令时会创建和系统处理器核一样多的线程；
- GCD将代码进行块的分割，将块加入调度队列（串行和并发）；

4.6 多线程问题

- fork()有两种，复制所有线程或者只复制调用了系统调用fork()的线程——取决于exec()，分叉之后立刻调用则不必全部复制；

4.6.2 信号处理

- 所有信号遵循相同模式
  - 信号是由特定事件的发生而产生的；
  - 信号被传递给某个进程；
  - 信号一旦收到就应处理。
- 信号处理程序分为两种：
  - 缺省的信号处理程序，每个信号都有一个；
  - 用户定义的信号处理程序，缺省动作由用户定义信号处理程序来改写；

4.6.3 线程撤销

- 在线程完成前终止线程，需要撤销的线程称作**目标线程**；
- 目标线程的撤销有两种情况
  - 异步撤销：一个进程立即终止目标进程，可能不会释放必要的系统资源；
  - 延迟撤销：目标自身不断检查自己是否应该终止，允许其有机会有序终止自己； pthread_cancel(), pthread_testcancel()（建立撤销点）
- TLS 线程局部存储，是一种变量的存储方法，这个变量在它所在的线程内是全局可访问的，但是不能被其他线程访问到，这样就保持了数据的线程独立性。

4.6.5 调度程序激活

- 在多对多或双层模型中，增加一个中间数据结构**轻量级进程LWP**，表现为虚拟处理器；
- 通常，每个并发阻塞的系统调用需要一个LWP。
- **调度器激活**，内核提供一组LWP给应用程序，可以被任意调度。此外，内核将有关特定事件通知应用程序，称为*回调*，由线程库通过*回调处理程序*处理。

第五章

5.1 基本概念

- 进程执行包括周期进行CPU执行和IO等待；
- 进程选择采用**短期调度程序**，从就绪队列中选择一个来执行。队列内的记录通常为**进程控制块（PCB）**；

5.1.3 抢占调度

- 需要调度的情况有四种：
  - 从运行切换到等待；
  - 从运行切换到就绪；
  - 从等待切换到就绪；
  - 终止；
- 如果调度只能发生在14情况下，则称为*非抢占的或协作的*，在该情况下，进程将会一致使用CPU直到终止或者切换状态；

5.1.4 调度程序

- 是一个模块，用来将CPU控制交给由短期调度程序选择的进程，包括
  - 切换上下文；
  - 切换到用户模式；
  - 跳转到用户程序的合适位置以便重新启动。
- 调度程序应尽可能快，停止一个进程而启动两一个进程所需的时间称为**调度延迟**；

5.2 调度准则

- CPU使用率：应该使CPU尽可能忙碌，即最大化；
- 吞吐量：一个时间单元内进程完成的数量，应最大化；
- 周转时间：从该进程提交到进程完成的时间段，包括等待进入内存、在就绪队列中等待、在CPU上执行和IO执行，应最小化；
- 等待时间：在就绪队列中等待所花的时间，CPU调度算法只影响此因素，应最小化；
- 响应时间：从提交请求到产生第一响应的时间，应最小化；

5.3 调度算法

5.3.1 FCFS 先到先得

- 可能发生**护航效果**：CPU密集型进程得到CPU，此时IO设备空闲；完成后IO进程很快执行完并移回到IO队列，CPU密集型进程重新进入CPU——即所有进程都在等待一个大进程释放CPU，这会导致CPU和设备使用率降低；
- 非抢占

5.3.2 SJF 最短作业优先调度

- 最优算法，但困难在于如何得知下次CPU执行的长度；
- 在批处理系统的长期调度中，可将用户提交作业时指定的进程时限作为长度，故SJF常用于长期调度；
- 抢占的SJF：**最短剩余时间优先**

5.3.3 优先级调度

- SJF为一个特例；
- 主要问题是**无穷阻塞或饥饿**：
  - 让某个低优先级进程无限等待CPU，采用**老化**——逐渐增加在系统中等待很长时间的进程的优先级解决；

5.3.4 RR 轮转调度

- 专门为分时系统设计；
- 类似FCFS，但使用抢占：定义一个较小时间单元为时间片；
- 时间片太大会演变成FCFS，时间片太小上下文切换开销太大；

5.3.5 多级调度队列

- 进程分为**前台进程**和**后台进程**，多级队列根据进程属性将就绪队列分成多个单独队列，每个队列有自己的调度算法。
- 最常用的包括RR前台FCFS后台

5.3.6 多级反馈调度队列

- 进程可以在队列之间切换；

5.4 线程调度

5.4.1 竞争范围

- 对于多对一或者多对多的模型系统线程库会调度用户级线程以便在LWP上运行，这种方案称为**进程竞争范围（PCS）**，因为竞争范围发生在同一进程的线程之间；
- 为决定哪个内核线程调度到一个处理器上，内核采用**系统竞争范围（SCS）**，一对一只采用这种；

5.5 多处理器调度

5.5.1 多处理器调度方法

- 非对称多处理：让一个处理器处理所有系统活动，其他处理器只执行用户代码；
- **对称多处理（SMP）**：每个处理器自我调度，都检查共同就绪队列；

5.5.2 处理器亲和性

- **处理器亲和性**：尽量避免对缓存的失效访问和重新填充，让一个进程运行在同一个处理器上；
- 当这个进程也可以迁移到其他处理器时，称为**硬亲和性**；

5.5.3 负载平衡

- 负载平衡设法将负载平分配到SMP系统的所有处理器（对于公共队列不必须，对于私有队列必须）
  - 推迁移：将超载处理器上的负载推到空闲或不太忙的处理器；
  - 拉迁移：空闲处理器从一个忙处理器上拉一个等待任务。
- 会影响亲和性；

5.5.4 多喝处理器

- 将多个处理器放置在一个物理芯片上；
- **内存停顿**：当一个处理器访问内存时，花费大量时间等待所需数据；
- *粗粒度*和*细粒度*多线程实现；

5.6 实时CPU调度

- **软实时系统**不保证会调度关键实时进程，只保证这类进程会更优先，**硬实时系统**确保应该在截止期限内得到服务；

5.6.1 最小化延迟

- **事件延迟**：从事件发生到事件得到服务这段时间；
  - 中断延迟：从CPU收到中断再到中断处理程序开始的时间；
  - 调度延迟：调度程序从停止一个进程到启动另一个进程所需的时间量，使其尽可能低的最有效计数是提供抢占式内核；

5.6.2 优先权调度

- 调度进成一般是周期性的，由固定的处理时间t，CPU应处理的截止期限d和周期p，`0<=t<=d<=p`，速率为1/p。
- 然后使用**准入控制**：承认进程，保证完成，如果不能保证，拒绝请求；

5.6.3 单调速率调度

- 采用抢占，静态优先的策略调度周期性任务，优先级与周期成反比——更频繁的需要CPU的任务应有更高优先级；
- 假定每次CPU执行，周期性进程处理时间相同；
- CPU利用率有限：`N * ( 2 ^ (1 / N) - 1)`

5.6.4 最早截止期限优先调度 EDF

第六章 同步

6.1 背景

- **竞争条件**：多个进程并发访问和操作统一数据并且执行结果与特定访问顺序有关；

- 因此需要写作进程进行**进程同步**以及**进程协调**；

6.2 临界区问题

- 每个进程有一段代码称为**临界区**，进程在该区执行可能更改公共变量，某个进程执行时不允许其他进程在他们的临界区执行；
  - 临界区问题：设计一个协议以便在进入临界区前，每个进程请求许可；
  - 实现这一请求的代码区段称为**进入区**，临界区之后为**退出区**，其他代码为**剩余区。**
- 临界区问题结局方案应满足：
  - 互斥；
  - 进步：如果没有进程在其临界区内且有其他进程需要进入临界区，则只有不在剩余区执行的进程可以参加选择，且这种选择不能被无限推迟；
  - 有限等待：从一个进程做出请求到这个请求被允许为止，其他进程允许进入临界区的次数有限；
  - 对n个进程的相对速度不作任何假设，每个进程执行速度不为0；
- 抢占式内核的临界区问题解决方案；

6.3 Peterson解决方案

- 适用于两个进程交错执行临界区和剩余区；
- 要求共享两个数据项

```
int turn;       //turn = i means pi can go in
boolean flag[2];    // flag[i] = true means pi is ready to go
do {
    /*enter*/
    flag[i] = true;
    turn = j;
    while(flag[j] && turn == j);
    /*quit*/
    flag[i] = false;
}
```

6.4 硬件同步

- 另一种基于 `test_and_set()` 的算法：

```
boolean waiting[n];
boolean lock;
do{
    waiting[i] = true;
    key = true;
    while(waiting[i] && key){
        key = test_and_set(&lock);  //set lock to true
    }
    waiting[i] = false;
    /*quit*/
    j = (j + 1) % n;
    while((j != 1) && !waiting[j])
        j = (j + 1) % n;
    if(j == i)
        lock = false;
    else
        waiting[j] = false;
}while(true);
```

6.5 互斥锁

```
acquire() {
    while(!available);
    available = true;
}
realease(){
    available = true;
}
```

- 缺点是会busy waiting（浪费，这种类型也被称为自旋锁；

6.6 信号量

- 信号量S是一个整型变量，只能通过wait()P, signal()V来访问；

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

- 一个进程修改信号量值时，没有其他进程能够修改同一信号量的值；

6.6.1 信号量的使用

- 分为计数信号量（不受限制）和二进制信号量（只有01，类似互斥锁）；
- 使用资源时wait，释放资源时signal；

6.6.2 信号量的实现

- 不再忙等待而是阻塞自己，将自己放到与信号量相关的等待队列中，并将状态切换为等待状态，最后利用wakeup()切换到就绪队列；

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

wait(semaphore *S){
    S->value--;
    if(S->value < 0){
        add this to S->list;
        block();
    }
}

signal(semaphore *S){
    S-->value++;
    if(S->value <= 0){
        remove a P from S->list;
        wakeup(P);
    }
}

void down(semaphore *s) {
    disable_interrupt();
    while ( *s == 0 ) {
        enable_interrupt();
        special_sleep();
        disable_interrupt();
    }
    *s = *s - 1;
    enable_interrupt();
}

void up(semaphore *s) {
    disable_interrupt();
    if ( *s == 0 )
        special_wakeup();
    *s = *s + 1;
    enable_interrupt();
}
```

- 这里的wait和signal并未真正取消忙等待，而是将忙等待从进入区移动至临界区；

6.6.3 死锁与饥饿

- 两个或多个进程无限等待一个事件（执行操作signal），而该事件只能由等待进程之一产生——**死锁**；

6.6.4 优先级的反转

- 指当一个较高优先级的进程需要读取或修改内核数据而这个数据当前整备较低优先级进程访问时出现的调度挑战（内核数据通常用锁保护，导致高优先级不得不等待）；
- **优先级继承协议**：正在访问的进程临时获得更高优先级进程的优先级；

6.7 经典同步问题

6.7.1 有界缓冲问题

- 生产者消费者公用以下结构

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
//producer
do{
    //produce an item in next_produced
    wait(empty);
    wait(mutex);
    //add next_produced to the buffer
    signal(mutex);
    signal(full);
} while(true);
//consumer
do{
    wait(full);
    wait(mutex);
    //remove an item from buffer to next_consumed
    signal(mutex);
    signal(empty);
    //consume the item
} while(true);
```

6.7.2 读者作者问题

- **第一读者作者问题**：要求读者不应该等待，除非作者已获得使用共享对象的权限；

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
//writer
do{
    wait(rw_mutex);
    //writing
    signal(rw_mutex);
} while(true);
//reader
do{
    wait(mutex);
    read_count++;
    if(read_count == 1)
        wait(rw_mutex);
    signal(mutex);
    //reading
    wait(mutex);
    read_count --;
    if(read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while(true);
```

- **第二读者作者问题**：一旦作者就绪，作者应尽快执行——即如果有一个作者等带访问对象，此时不会有新读者开始读；

- 读写锁：
  - 容易识别那些进程只读共享数据和哪些进程只写共享数据；
  - 读者进程数比作者进程数多。

6.7.3 哲学家进餐问题

- 没根筷子代表一个信号量

```
semaphore chop[5] = {1, 1, 1, 1, 1};
do{
    wait(chop[i]);
    wait(chop[(i+1)%5]);
    //eating
    signal(chop[i]);
    signal(chop[(i+1)%5]);
    //thinking
} while(true);
```

- 仍然导致死锁，以下方式补救
  - 允许最多4名哲学家坐在座子上；
  - 只有一个哲学家两根筷子都可用时，才能拿起筷子；
  - 使用非对称解决方案；

6.8 管程——高级同步工具

- 用于解决信号量解决临界区问题时的各种错误；

6.8.1 使用方法

- 管程属于**抽象数据类型（ADT）**，封装了数据以及对其操作的一组函数；
- 管程内定义的函数才能访问管程内的局部声明的变量和形式参数，确保只有一个进程在管程内处于活动状态
- P调用x.signal()时，x上有一个挂机进程Q：
  - **唤醒并等待**：进程P等待直到Q离开管程，或等待另一个条件；
  - **唤醒并继续**：进程Q等待直到P离开管程，或等待另一个条件；

6.8.2 哲学家问题的管程解决

- 加入限制：只有两根筷子都可用才可拿起筷子；

```
#define N 5
#define LEFT ((i+N-1) % N)
#define RIGHT ((i+1) % N)
int state[N];
semaphore mutex = 1;
semaphore s[N];
void take(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}
void put(int i) {
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}
void test(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
    state[i] = EATING;
    up(&s[i]);
    }
}
void philosopher(int i) {
    think();
```

```
    take(i);
    eat();
    put(i);
}
```

6.8.4 管程内的进程重启

- **条件等待**结构：

```
x.wait(c); //c is int type
```

- c称为优先值，优先值最小的进程会在x.signal()执行时重新启动；
- 管程概念仍可能引起如下问题：
  - 进程可能在没有先获得资源访问权限时访问资源；
  - 进程可能在获得访问权限后不再释放资源；
  - 进程可能在没有请求之前试图释放资源；
  - 进程可能请求统一资源两次。

6.9 实例

- Linux对于SMP机器提供基本加锁机制为自旋锁，内核则只有短时间操作才会使用自旋锁；
- 对于单处理器，获得自旋锁改为禁止抢占，释放自旋锁改为启用抢占；

6.10 替代方法

- **事务内存**源自数据库理论，提供一种进程同步策略；
- **内存事务**为一个内存读写操作的序列，是原子的；
- **软件事务内存（STM）**和**硬件事务内存（HTM）**；
- `#pragma omp critical` 可以指定后面的代码为临界区；
- **原子操作**：独立而不可分割的操作，要么全部执行完，要么不执行或者回滚；

第七章 死锁

- 锁申请的资源被其他进程所占有，那么该进程可能再也无法改变状态；

7.2 死锁特征

7.2.1 必要条件

- 互斥：至少有一个资源一次只有一个进程可使用；
- 占有并等待：一个进程应占有至少一个资源并等待另一个资源，而这个资源为其他进程所占有；
- 非抢占：资源不能被抢占；
- 循环等待：登对队列头尾相接。
- 四个条件同时成立则产生死锁。

7.2.2 资源分配图

- 有环不一定死锁；

7.3 死锁处理

- 三种基本方法
  - 通过协议预防或者避免死锁；
  - 可以允许进入死锁状态，但是检测并恢复他；
  - 可以忽视这个问题，认为死锁不可能在系统中发生。
- 两种方案：
  - 死锁预防：确保至少有一个必要条件不成立；
  - 死锁避免：要求操作系统事先得到有关进程申请资源和使用资源的额外信息；

7.4 死锁预防

- 互斥必须成立，无法否定；
- 持有并等待：
  - 方法一：每个进程执行前申请并获得所有资源；
  - 方法二：允许进程仅在没有资源时才可以申请资源，申请更多资源时应释放现已分配的资源；
  - 资源利用率较低，可能发生饥饿；
- 无抢占：如果一个进程持有资源并且申请另一个不能被立即分配的资源，则它现在拥有的资源都可以被抢占；
- 循环等待：对所有资源类型进行排序，要求每个进程按照递增顺序来申请资源；

7.5 死锁避免

- 死锁预防策略会导致设备使用率低和系统吞吐率低；
- 银行家算法：分配之前判断是否安全，安全，才分配；

7.6 死锁检测

- 等待图（单个实例）；
- 银行家检测（多个实例）；
- 死锁检测算法；

7.7 死锁恢复

7.7.1 进程终止

- 终止所有死锁进程；
- 一次种植一个进程，直到消除思索循环；

7.7.2 资源抢占

- 选择牺牲进程；
- 回滚进程直到足够打破死锁；
- 饥饿：同i进程可能总是被选为牺牲品，可以添加回滚次数作为限制；

第八章 内存管理

8.1 背景

8.1.1 基本硬件

- **基地址寄存器**含有最小的合法的物理内存地址；
- **界限地址寄存器**指定了范围大小；
- 这两个寄存器允许操作系统修改而不允许用户修改；

8.1.2 地址绑定

- 源程序中的地址通常由符号表示，编译器将符号地址绑定到可重定位的地址，连接程序或加载程序将可重定位地址绑定到绝对地址；
- 编译时生成绝对代码，加载时生成可重定位代码；

8.1.3

- CPU生成的地址称为**逻辑地址**，内存单元看到的地址（加载到内存地址寄存器）的地址称为物理地址；
- 执行时会产生不同的逻辑地址和物理地址，这时称逻辑地址为虚拟地址；
- **内存管理单元MMU**的硬件设备负责虚拟地址到物理地址的运行时映射；
- 真实物理地址将逻辑地址通过基地址寄存器进行重定位；

8.1.4 动态加载

- 程序只有调用时才会加载，都已重定位加载格式保存在磁盘上；

8.1.5 动态链接和共享库

- dll为系统库，可链接到用户程序以便运行；
- 使用dll则在二进制映像内每个库程序的引用都有一个**存根**，用来指出如何定位适当的内存驻留库程序或者如何加载库；
  - 存根会用程序地址替换自己，这样下次执行时不会因dll产生任何开销；
- 也可用于库的更新。一个库的多个版本都可以加载到内存，通过版本信息确定使用哪个库的副本，称为**共享库**；

8.2 交换

8.2.1 标准交换

- 在内存与备份存储之间进；
- 交换时间的主要部分是传输时间；
- 常用变种：正常情况下禁止交换，空闲内存低于某个阈值时启动交换；或是只交换进程的部分；

8.3 连续内存分配

- *暂时*的操作系统代码：根据需要调入或者调出，不必在内存中保留它的代码和数据；

8.3.2 内存分配

- 最简单的方法：分区。每个分区一个进程。
  - 可变分区：OS用一个表记录那些可用那些不可用。最开始的所有内存均可用于用户进程，因此可以作为一大块的**孔**；
- 首次适应：分配首个足够大的块；
- 最有适应：最小的足够大的块；
- 最差适应：分配最大的快；

8.3.3 碎片

- 首次适应和最佳适应都有**外部碎片**的问题：空闲浪费的空间被分成小的片段，可以满足请求但是不连续；
  - 允许逻辑地址空间不连续；
- **内部碎片**分配的内存减去所需的内存；
  - 采用紧缩解决，但是只有动态时才能采用；

8.4 分段

- 一种支持分段的用户视图的内存管理方案；
- 一个C编译器可能产生如下的段：
  - 代码；
  - 全局变量；
  - 堆；
  - 每个线程使用的栈；
  - 标准C库；

8.4.2 分段硬件

- 定义一个实现方式以便映射用户定义的二位地址到一维物理地址
  - 通过**段表**实现；
  - 段表的每个条目都有段基地址以及段界限；

8.5 分页

8.5.1 基本方法

- 将物理内存分为固定大小的块，称为**帧**；逻辑内存分为同样大小的块，称为**页**；
- CPU生成的地址分为**页码**和**页偏移**，页码作为页表的索引，包含每页所在物理内存的基地址；基地址与页偏移组合成物理内存地址；
- 分页方案不会产生外部碎片，但是有内部碎片；
- 物理内存的分配细节：哪些帧已分配，哪些空闲，总共多少帧等，保存在**帧表**中；
- 分页增加了上下文切换的时间；

8.5.2 硬件

- 使用**PTBR页表基地址寄存器**指向页表，但是要两次内存访问；
- **转换表缓冲区TLB**，有的TLB在每个条目中保存**地址空间标识符ASID**，唯一表示每个进程，并为进程提供地址空间的保护；

8.5.3 保护

- 通过与每个帧关联的保护位实现，用一个位定义一个页是可读可写或只可读；
- 有的系统提供硬件**页表长度寄存器PTLR**来表示页表的大小；

8.5.4 共享页

8.6 页表结构

- 向前映射页表；

PLUS CONTENT

- data segament包含初始化的变量（全局和局部和静态），BSS包括尚未被初始化的变量，开始执行时才真正分配内存；
- 32-bitlinux系统地址空间3GB左右；

第九章 虚拟内存

9.2 请求调页（demand paging）

- 仅在需要时才加载页面；
- malloc()只分配虚拟页面地址；

9.3 写是复制

- 通过允许父进程和子进程最初共享相同的页面来工作；这些共享页面标记为i而是复制，这意味着如果任何一个进程写入共享页面，那么就创建共享页面的副本；

9.4 页面置换算法

- FIFO
  - Belady异常：随着分配帧数量的增加，缺页错误率可能增加；
  -
- LRU
- OPT：置换最长时间不会使用的页面；
- 二者都属于堆栈算法，不会产生异常——帧数为n的内存页面集合时帧数为n+1内存页面集合的子集；
- 近似LRU；
- 二次机会算法；

9.5 帧分配算法

- 平均分配；
- 比例分配；

9.6 系统抖动

- 没有足够的帧，会很快产生缺页错误，此时必须置换某个页面，但由于所有页面都在使用中，又很快的将再次置换，如此高度的页面调度活动乘坐抖动——调页时间多余执行时间；
- 原因：全局置换算法，CPU利用率降低，CPU调度程序进而增加多道成都，导致更多的缺页错误；
- 解决：局部置换算法或者优先权置换算法，提供需要的帧数；工作集模型；PFF缺页错误频率——设置上下限；

9.8 分配内核内存

- 不同于用户内存分配的原因：
  - 内核需要为不同大小的数据结构请求内存，有的小于一页；
  - 用户模式进程分配的页面不必位于连续物理内存；

9.8.1 伙伴系统

- 从屋里连续的大小固定的段上进行分配，采用2的幂分配器；
- 可以通过合并快速将相邻伙伴组合已形成更大分段，但很容易造成碎片；

9.8.2 slab分配

- 每个slab由一个或多个物理连续的页面组成，每个cache由一个或多个slab组成；
- 没有因碎片而引起的浪费，可以快速满足内存请求。

第十二章 磁盘

- 盘片的表面分为圆形磁道，再细分为扇区，同一磁臂为止的刺刀集合形成柱面；
- **扇区备用或扇区转寄**：控制器采用备用快逻辑的替换坏块；
- **扇区滑动**扇区整体后移；
- 磁盘可以储存数据之前必须先**低级格式化或者物理格式化**，使用特殊的数据结构填充磁盘，包括头部尾部数据区域，扇区号和纠错代码（ECC）
- 使用文件系统写入或者直接将磁盘分区作为逻辑块的一个大的有序数组，称为原始磁盘（raw disk）；
- 磁盘调度算法：
  - FCFS
  - SSTF 最短寻道时间有先；可能引起饥饿；
  - SCAN
  - CSCAN 到达一段后立刻回到另一端；
  - CLOOK 会先查看前方是否由请求；
- SSD分类
  - SLC单层单元：每层1bit；
  - MLC多层单元：每层2bit；

12.7 RAID 磁盘冗余阵列

- RAID0:具有块分条但是没有冗余的磁盘阵列；
- RAID1:磁盘镜像；
- RAID01：0 + 1， 先分条，再镜像；
- RAID10:1 + 0, 先镜像再分条；
- RAID4:多一组p块，写入时采用RMW（读改写，四次磁盘访问，两次读入两次写入，需要读入校验盘和写数据盘）/RRW（读重构写，读入不写的数据盘）
- raid5:分散在所有磁盘上，n块的校验块在(n mod 5) + 1上；
- raid6:2位冗余数据，两个磁盘，和5一样交错分布；

第十章 文件系统

10.1 文件概念

- 文件属性
  - 名称；
  - 标识符；
  - 类型；
  - 位置；
  - 尺寸；
  - 保护；
  - 时间、日期、用户标识；
- FILE中的两个重要成员：文件描述符和一个缓冲；
- 六个文件基本操作：创建文件，写文件，读文件，重新定位文件，删除文件，截断文件；

- 上述的大多数文件操作设计搜索目录以得到命名文件的相关条目，为避免这种搜索，OS有一个**打开文件表**用于维护所有打开文件的信息。当强求文件操作时，可通过该表的索引指定文件而不需要搜索。通常为每个文件关联一个**打开计数**；
- 文件数据储存在内核固定大小的缓存中；

10.2 文件目录

- 目录也是文件；
- 创建文件等同于创建他的目录文件；
- FAT：文件配置表，一种文件系统的简称；
- FATX:X代表簇的大小；
- boot code: a set of instructions run by a computer when it is starting;
- MBR:a special type of boost sector which is located at the beginning of the partioned computer storage;
- FAT采用小端对齐，低地址字节放在低地址端；
- ext extent file system:indoe allocation;
  - 分为多个group，superblock全都相同，group0的称为primary superblock；
  - GDT储存初始块号；
  - 目录条目储存inode和文件名；
  - 硬链接是一个新的指向一个已存在文件的目录项，即创建一个偶两个路径名的文件；
  - link count用于计算有多少指向他的目录项；
    - 当一个文件被创建，这个值总是1；
    - 当一个目录被创建，这个值总是2（temp和.
    - unlink将该值-1，等于0时将被删除；
  - 软连接时一个文件，一个新的inode将被创建，储存目标路径名；
    - 少于60为12个d和3个id，多余则多一个数据块；
  - buffer cache:储存读写副本的块，由页cache 目录项cache inodecache三种；
  - TXB，TXE（包含TID）；
    - 写日志：写事务内容（txb，元数据和数据）；
    - 日志提交：元数据，数据，（txe）；
    - 检查点：写入磁盘；
  - 元数据日志：
    - 写数据和写日志元数据（可并行）；
    - 日志提交；
    - 元数据检查点；
  - 虚拟文件系统：
    - 物理文件系统和服务之间的一个接口；
  - 读4写2执行1；从左往右所有者，同组用户，其他用户；

第十三章 IO

-