操作系统

- 操作系统
  - 一、概述
    - 1、基本特征
      - 1.1并发
      - 1.2共享
      - 1.3虚拟
      - 1.4异步
    - 2、基本功能
      - 2.1进程管理
      - 2.2线程管理
      - 2.3内存管理
      - 2.4文件管理
      - 2.5设备管理
    - 3、系统调用
    - 4、大内核和微内核
      - 4.1大内核
      - 4.2微内核
    - 5、中断分类
      - 5.1外中断
      - 5.2异常
      - 5.3陷入
  - 二、进程管理
    - 1、进程与线程
      - 1.1进程
      - 1.2线程
      - 1.3区别
    - 2、进程状态的切换
    - 3、进程调度算法
      - 3.1批处理系统
        - 3.1.1先来先服务（FCFS）
        - 3.1.2短作业优先（SJF）
        - 3.1.3最短剩余时间优先（SRTN）
      - 3.2交互式系统
        - 3.2.1时间片轮转
        - 3.2.2优先级调度

        - 3.2.3多级反馈队列
    - 3.3实时系统
- 4、进程同步
    - 4.1临界区
    - 4.2同步与互斥
    - 4.3信号量
        - 4.3.1使用信号量实现生产者-消费者问题
    - 4.4自旋锁
    - 4.5管程
- 5、经典同步问题
    - 5.1读者-写者问题
    - 5.2哲学家进餐问题
- 6、进程通信
    - 6.1管道
    - 6.2FIFO
    - 6.3消息队列
    - 6.4信号量
    - 6.5共享存储
    - 6.6套接字
- 三、线程管理
    - 1、线程基本概念
    - 2、线程调度策略
    - 3、线程同步
        - 3.1互斥锁
        - 3.2条件变量
        - 3.3信号量
        - 3.4读写锁
    - 4、线程通信
        - 4.1全局变量
        - 4.2自定义消息
        - 4.3std::promise和std::future
        - 4.4std::packaged_task
        - 4.5std::async
- 四、死锁
    - 1、必要条件
    - 2、处理方法
    - 3、鸵鸟策略
    - 4、死锁检测与死锁恢复
    - 5、死锁预防
        - 5.1破坏互斥条件
        - 5.2破坏占有和等待条件
        - 5.3破坏不可抢占条件

        - 5.4破坏环路等待
    - 6、死锁避免
        - 6.1银行家算法
- 五、内存管理
    - 1、虚拟内存
    - 2、分页系统地址映射
    - 3、页面置换算法
        - 3.1最佳
        - 3.2最近最久未使用（LRU）
        - 3.3先进先出
        - 3.4最近最少使用（LFU）
    - 4、分段
        - 4.1段页式
        - 4.2分页与分段的比较
- 六、设备管理
    - 1、磁盘结构
    - 2、磁盘调度算法
        - 2.1先来先服务(FCFS)
        - 2.2最短寻道时间优先(SSTF)
        - 2.3电梯算法(SCAN)
- 七、编译和链接
    - 1、编译系统
        - 1.1预处理（产生.i文件）
        - 1.2编译（产生.s文件）
        - 1.3汇编（产生.o或.obj文件）
        - 1.4链接（产生.out或.exe文件）
    - 二、静态链接
    - 三、目标文件
    - 四、动态链接

一、概述

1、基本特征

1.1并发

并发是指宏观上在一段时间内能同时运行多个程序，而并行则指同一时刻能运行多个指令。

并行需要硬件支持，如多流水线、多核处理器或者分布式计算机系统。

操作系统通过引入进程和线程，使得程序能够并发运行。

1.2共享

共享是指系统中的资源可以被多个并发进程共同使用。

有两种共享方式：互斥共享和同时共享。

互斥共享的资源称为临界资源，如打印机等，，在同一时刻只允许一个进程访问，需要用同步机制来实现互斥访问。

1.3虚拟

虚拟技术把一个物理实体转换为多个逻辑实体。

主要有两种虚拟技术：时（时间）分复用技术和空（空间）分复用技术。

多个进程能在同一个处理器算上并发执行使用了时分复用技术，让每个进程轮流占用处理器，每次只执行一小个时间片并快速切换。

虚拟内存使用了空分复用技术，它将物理内存抽象为地址空间，每个进程都有各自的地址空间。地址空间的页被映射到物理内存，地址空间的页并不需要全部在物理内存中，当使用到一个内有在物理内存的页时，执行页面置换算法，将该页置换到内存中。

1.4异步

异步指进程不是一次性执行完毕，而是走走停停，以不可知的速度向前推进。

2、基本功能

2.1进程管理

进程控制、进程同步、进程通信、死锁处理、处理机调度等。

2.2线程管理

线程控制、线程同步、线程通信、线程调度等。

2.3内存管理

内存分配、地址映射、内存保护与共享、虚拟内存等。

2.4文件管理

文件存储空间的管理、目录管理、文件读写管理和保护等。

2.5设备管理

完成用户I/O请求，方便用户使用各种设备，并提高设备的利用率。

主要包括缓冲管理、设备分配、设备处理、虚拟设备等。

3、系统调用

如果一个进程在用户态需要使用内核态的功能，就进行系统调用从而陷入内核，由操作系统代为完成。

Linux的系统调用主要有以下这些：

Task	Commands
进程控制	fork(); exit(); wait();
进程通信	pipe(); shmget(); mmap();
文件操作	open(); read(); write();
设备操作	ioctl(); read(); write();
信息维护	getpid(); alarm(); sleep();
安全	chmod(); umask(); chown();

4、大内核和微内核

4.1大内核

大内核是将操作系统功能作为一个紧密结合的整体放倒内核。

由于各模块共享信息，因此有很高的新能。

4.2微内核

由于操作系统不断复杂，因此将一部分操作系统功能移出内核，从而降低内核的复杂性。移出的部分根据分层的原则划分成若干服务，相互独立。

在微内核结构下，操作系统被划分成小的、定义良好的模块，只有微内核这一个模块运行在内核态，其余模块运行在用户态。

因为需要频繁地在用户态和核心态之间切换，所以会有一定的性能损失。

5、中断分类

5.1外中断

由CPU执行指令以外的事件引起，如I/O完成中断，表示设备输入/输出处理已经完成，处理器能够发送下一个输入/输出请求。此外还有时钟中断、控制台中断等。

5.2异常

由CPU执行指令的内部事件引起，如非法操作码、地址越界、算术移出等。

5.3陷入

在用户程序中使用系统调用。

二、进程管理

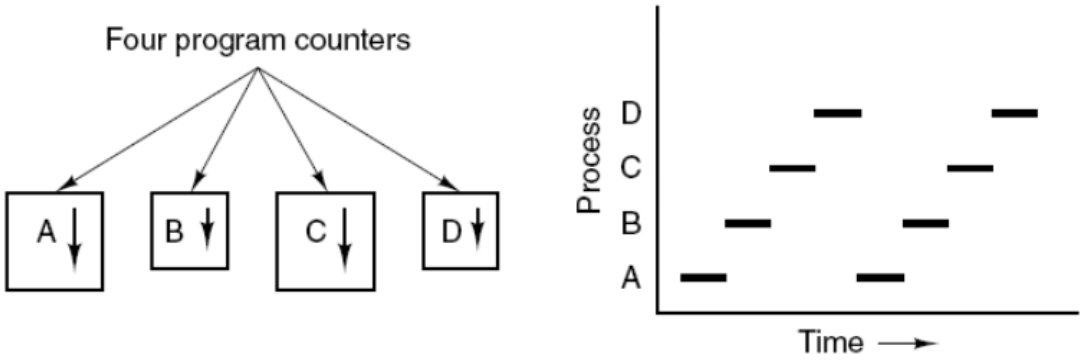
1、进程与线程

1.1进程

进程是资源分配的基本单位。

进程控制块（PCB）描述进程的基本信息和运行状态，所谓的创建进程和撤销进程，都是指对PCB的操作。

下图显示了4个程序创建了4个进程，这4个进程可以并发地执行。

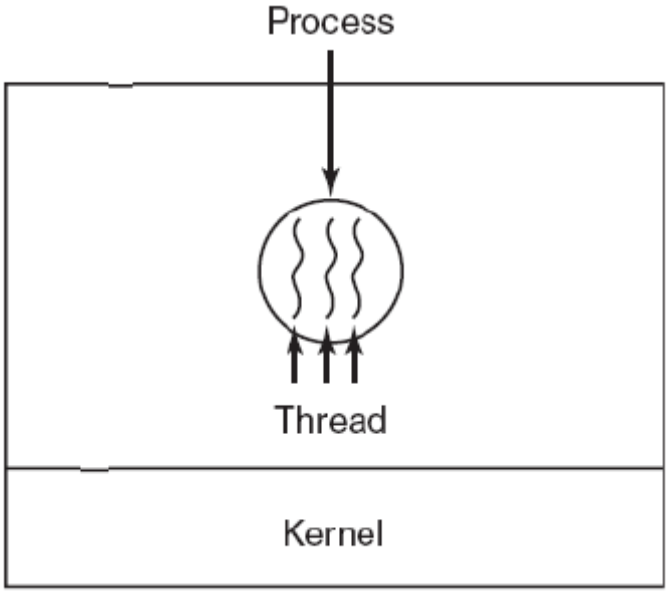


1.2线程

线程是独立调度的基本单位。

一个进程中可以有多个线程，它们共享进程资源。

QQ和浏览器是两个进程，浏览器进程里面有很多线程，例如HTTP请求线程、事件响应线程、渲染线程等，线程的并发执行使得在浏览器中点击一个新链接从而发起HTTP请求时，浏览器还可以响应用户的其他事件。



1.3区别

Ⅰ 拥有资源

进程是资源分配的基本单位，但是线程不拥有资源，线程可以访问隶属进程的资源。

Ⅱ 调度

线程是独立调度的基本单位，在同一进程中，线程的切换不会引起进程切换，从一个进程中的线程切换到另一个进程中的线程时，会引起进程切换。

Ⅲ 系统开销

由于创建或撤销进程时，系统都要为之分配或回收资源，如内存空间、I/O 设备等，所付出的开销远大于创建或撤销线程时的开销。类似地，在进行进程切换时，涉及当前执行进程 CPU 环境的保存及新调度进程 CPU 环境的设置，而线程切换时只需保存和设置少量寄存器内容，开销很小。

Ⅳ 通信方面

线程间可以通过直接读写同一进程中的数据进行通信，但是进程通信需要借助 IPC。

Ⅴ 内核态用户态

进程的实现只能由操作系统内核来实现，而不存在用户态实现的情况。而线程可以分为内核态和用户态。

内核态创建线程比较浪费系统空间资源，因为系统需要维护线程列表，而线程的数量要远远大于进程的数量，过多的线程创建会使系统资源耗尽而瘫痪。其次内核态实现会修改操作系统。

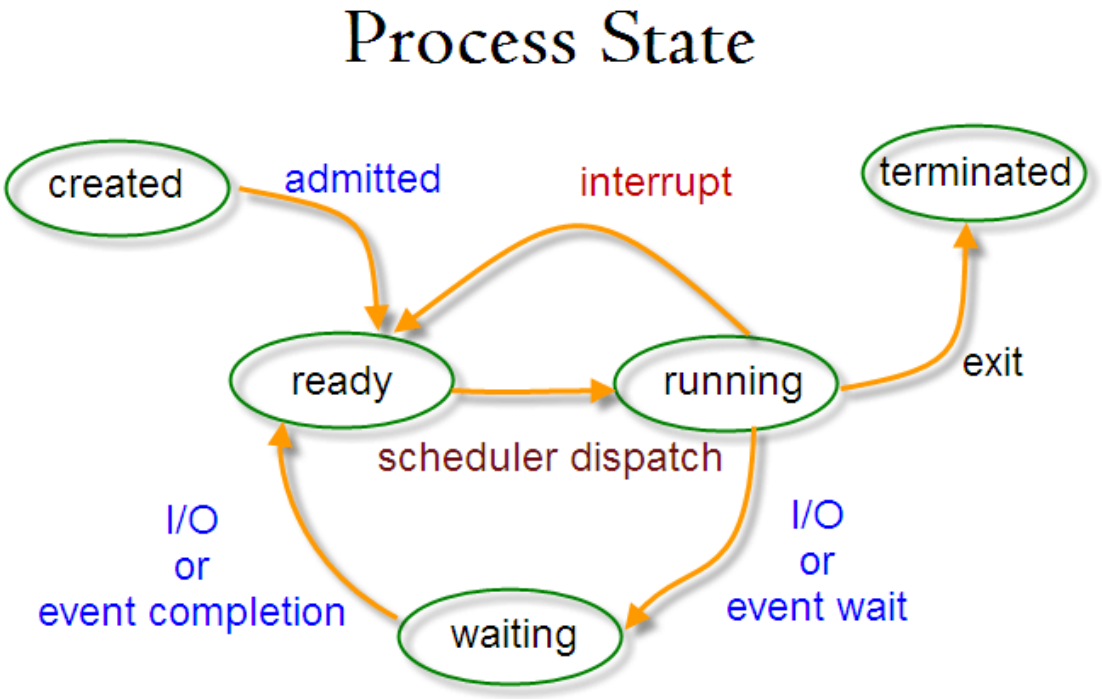
使用用户态创建线程就不必太担心系统资源耗尽的问题，内核不需要知道有多少线程创建。用户创建方便。

缺点：

如果在执行过程中一个线程受阻，它将无法将控制权交出来，这样整个进程都无法推进。操作系统随即把CPU控制权交给另外一个进程。

这样，一个线程受阻造成整个进程受阻，我们期望的通过线程对进程实施分身的计划就失败了。这是用户态线程致命的缺点。

2、进程状态的切换



s

- 就绪状态（ready）：等待被调度
- 运行状态（runing）
- 阻塞状态（waiting）：等待资源

1）就绪——执行：对就绪状态的进程，当进程调度程序按一种选定的策略从中选中一个就绪进程，为之分配了处理机后，该进程便由就绪状态变为执行状态；

2）执行——阻塞：正在执行的进程因发生某等待事件而无法执行，则进程由执行状态变为阻塞状态，如进程提出输入/输出请求而变成等待外部设备传输信息的状态，进程申请资源（主存空间或外部设备）得不到满足时变成等待资源状态，进程运行中出现了故障（程序出错或主存储器读写错等）变成等待干预状态等等；

3）阻塞——就绪：处于阻塞状态的进程，在其等待的事件已经发生，如输入/输出完成，资源得到满足或错误处理完毕时，处于等待状态的进程并不马上转入执行状态，而是先转入就绪状态，然后再由系统进程调度程序在适当的时候将该进程转为执行状态；

4）执行——就绪：正在执行的进程，因时间片用完而被暂停执行，或在采用抢先式优先级调度算法的系统中,当有更高优先级的进程要运行而被迫让出处理机时，该进程便由执行状态转变为就绪状态。

应注意：

- 只有就绪态和运行态可以相互转换，其他都是单向转换。就绪状态的进程通过调度算法从而获得CPU时间，转为运行状态；而运行状态的进程，在分配给它的CPU时间片用完之后就转换为就绪状态，等待下一次调度。
- 阻塞状态是缺少需要的资源从而由运行状态转换而来，但是该资源不包括CPU资源，缺少CPU时间会从运行态转换为就绪态。

3、进程调度算法

3.1批处理系统

批处理系统没有太多的用户操作，在该系统中，调度算法目标是保证吞吐量和周转时间（从提交到终止的时间）。

3.1.1先来先服务（FCFS）

非抢占式的调度算法，按照请求的顺序进行调度。

有利于长作业，但不利于短作业，因为短作业必须一直等待前面的长作业执行完毕才能执行，而长作业又需要执行很长时间，造成了短作业等待时间过长。

3.1.2短作业优先（SJF）

非抢占式的调度算法，按估计运行时间最短的顺序进行调度。

长作业可能会被饿死，处于一直等待短作业执行完毕的状态。因为如果一直有短作业到来，那么长作业永远得不到调度。

3.1.3最短剩余时间优先（SRTN）

最短作业优先的抢占式版本，按剩余运行时间的顺序进行调度。当一个新作业到达时，其整个运行时间与当前进程的剩余时间做比较。如果进的进程需要的时间更少，则挂起当前进程，运行新的进程。否则新的进程等待。

3.2交互式系统

交互式系统有大量的用户交互操作，在该系统中调度算法的目标是快速地进行响应。

3.2.1时间片轮转

将所有就绪进程按 FCFS 的原则排成一个队列，每次调度时，把 CPU 时间分配给队首进程，该进程可以执行一个时间片。当时间片用完时，由计时器发出时钟中断，调度程序便停止该进程的执行，并将它送往就绪队列的末尾，同时继续把 CPU 时间分配给队首的进程。

时间片轮转算法的效率和时间片的大小有很大关系：

- 因为进程切换都要保存进程的信息并且载入新进程的信息，如果时间片太小，会导致进程切换得太频繁，在进程切换上就会花过多时间。
- 而如果时间片过长，那么实时性就不能得到保证。

3.2.2优先级调度

为每个进程分配一个优先级，按优先级进行调度。

为了防止低优先级的进程永远等不到调度，可以随着时间的推移增加等待进程的优先级。

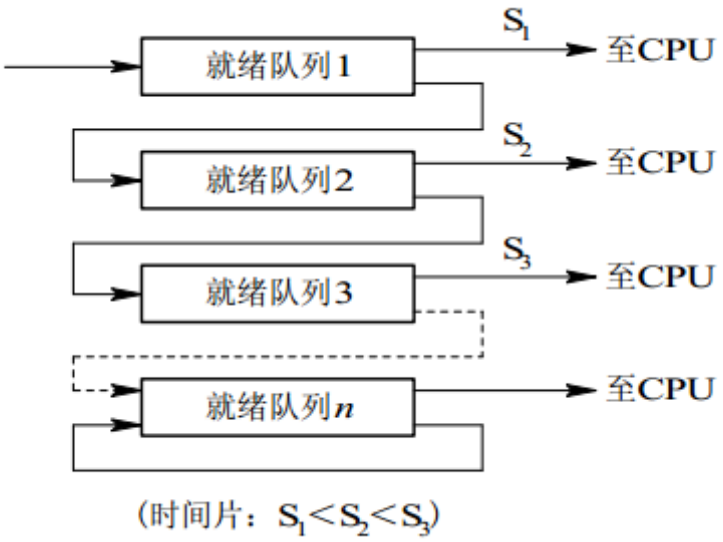
3.2.3多级反馈队列

一个进程需要执行100个时间片，如果采用时间片轮转调度算法，那么需要交换100次。

多级队列是为了这种需要执行多个时间片的进程考虑，它设置了多个队列，每个队列时间片大小都不同，例如1,2,4,8...进程在第一个队列没执行完，就会被移到下一个队列。这种方式下，之前的进程只需要交换7次。

每个队列优先权也不同，最上面的优先权最高。因此只有上一个队列没有进程在排队，才能调度当前队列上的进程。

可以将这种调度算法看成是时间片轮转调度算法和优先级调度算法的结合。



3.3实时系统

实时系统要求一个请求在一个确定时间内得到响应。

分为硬实时和软实时，前者必须满足绝对的截止时间，后者可以容忍一定的超时。

4、进程同步

为什么要引入进程间的同步？由于操作系统中的进程是并发的，因此当协同进程对共享数据进行访问时，可能会造成数据的不一致性问题。为了保证数据的一致性，那么我们就需要一种有效地机制，这就是被我们称之为的进程同步机制。

4.1临界区

对临界资源进行访问的那段代码称为临界区。

为了互斥访问临界资源，每个进程在进入临界区之前，需要先进行检查。

4.2同步与互斥

- 同步：多个进程因为合作产生的直接制约关系，使得进程有一定的先后执行关系。
- 互斥：多个进程在同一时刻只有一个进程能够进入临界区。

4.3信号量

信号量是一个整型变量，可以对其进行down和up操作，也就是常见的P和V操作。

- down：如果信号量大于0，执行-1操作；如果信号量等于0，进程睡眠，等待信号量大于0；
- up：对信号量执行+1操作，唤醒睡眠的进程让其完成down操作。

down和up操作需要被设计成原语，不可分割，通常的做法是在执行这些操作的时候屏蔽中断。

如果信号量的取值只能为0或者1，那么就成为了**互斥量**（mutex），0表示临界区已经加锁，1表示临界区解锁。

```
typedef int semaphore;
semaphore mutex = 1;
void P1() {
    down(&mutex);
    // 临界区
    up(&mutex);
}

void P2() {
    down(&mutex);
    // 临界区
    up(&mutex);
}
```

4.3.1使用信号量实现生产者-消费者问题

问题描述：使用一个缓冲区来保存物品，只有缓冲区没有满，生产者才可以放入物品；只有缓冲区不为空，消费者才可以拿走物品。

因为缓冲区属于临界资源，因此需要使用一个互斥量mutex来控制对缓冲区的互斥访问。

为了同步生产者和消费者的行为，需要记录缓冲区中物品的数量。数量可以使用信号量来进行统计，这里需要使用两个信号量：empty记录缓冲区还能放的数量，full记录缓冲区能取走的数量。其中，empty信号量是在生产者进程中使用，当empty不为0时，生产者才可以放入物品；full信号量是在消费者进程中使用，当full信号量不为0时，消费者才可以取走物品。

注意，不能先对缓冲区进行加锁，再测试信号量。也就是说不能先执行down(mutex)再执行down(empty)。如果这么做，那么可能出现：生产者对缓冲区加锁之后，执行down(empty)操作，发现empty为0，表明缓冲区已满，此时生产者进程睡眠，但是并没有释放缓冲区，那么此时消费者不能进入临界区，也就无法执行up(empty)操作，empty永远为0，导致死锁。

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer() {
    while(TRUE) {
        int item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer() {
    while(TRUE) {
        down(&full);
        down(&mutex);
        int item = remove_item();
        consume_item(item);
        up(&mutex);
        up(&empty);
    }
}
```

4.4自旋锁

自旋锁是为了保护共享资源提出的一种锁机制。

调用者申请的资源如果被占用，即自旋锁被已经被别的执行单元保持，则调用者一直循环在那里看是否该自旋锁的保持着已经释放了锁。

自旋锁是一种比较低级的保护数据结构和代码片段的原始方式，可能会引起以下两个问题;

- 死锁
- 过多地占用CPU资源

4.5管程

使用信号量机制实现的生产者消费者问题需要客户端代码做很多控制，而管程把控制的代码独立出来，不仅不容易出错，也使得客户端代码调用更容易。

管程有一个重要特性：在一个时刻只能有一个进程使用管程。进程在无法继续执行的时候不能一直占用管程，否则其它进程永远不能使用管程。

管程引入了**条件变量**以及相关的操作：wait() 和 signal() 来实现同步操作。对条件变量执行 wait() 操作会导致调用进程阻塞，把管程让出来给另一个进程持有。signal() 操作用于唤醒被阻塞的进程。

5、经典同步问题

5.1读者-写者问题

允许多个进程同时对数据进行读操作，但是不允许读和写以及写和写操作同时发生。

一个整型变量count记录在对数据进行读操作的进程数量，一个互斥量count_mutex用于对count加锁，一个互斥量data_mutex用于对读写的数据加锁。

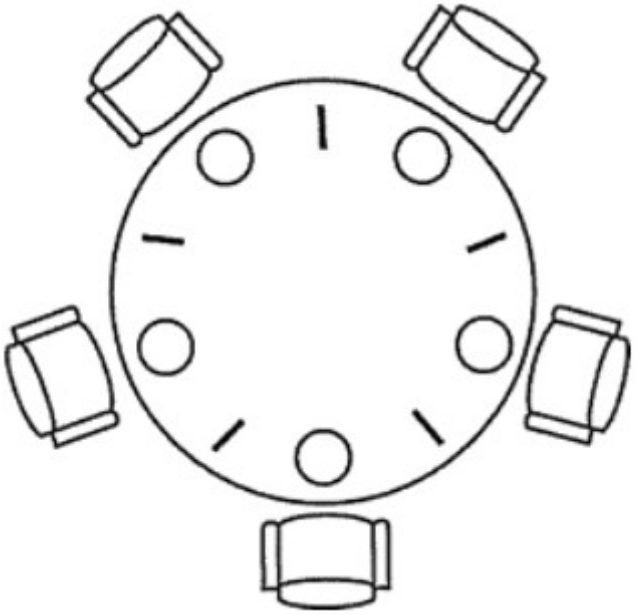
```
typedef int semaphore;
semaphore count_mutex = 1;
semaphore data_mutex = 1;
int count = 0;

void reader() {
    while(TRUE) {
        down(&count_mutex);
        count++;
        if(count == 1) down(&data_mutex); // 第一个读者需要对数据进行加锁，防止写进程访问
        up(&count_mutex);
        read();
        down(&count_mutex);
        count--;
```

```
        if(count == 0) up(&data_mutex);
        up(&count_mutex);
    }
}

void writer() {
    while(TRUE) {
        down(&data_mutex);
        write();
        up(&data_mutex);
    }
}
```

5.2哲学家进餐问题



五个哲学家围着一张圆桌，每个哲学家面前放着食物。哲学家的生活有两种交替活动：吃饭以及思考。当一个哲学家吃饭时，需要先拿起自己左右两边的两根筷子，并且一次只能拿起一根筷子。

下面是错误示例，会造成死锁：

```
semaphore chopstick[5] = {1,1,1,1,1};
#define N 5
void philosopher(int i) {
    while(TRUE) {
        think();
        //当哲学家饥饿时，总是先拿左边的筷子，再拿右边的筷子
        take(chopstick[i]);
        take(chopstick[(i+1)%N]);
        // 吃饭
```

```
        eat();
        //当哲学家进餐完成后，总是先放下左边的筷子，再放下右边的筷子
        put(chopstick[i]);
        put(chopstick[(i+1)%N]);
    }
}
```

为了防止死锁的发生，可以使用以下三种策略：

策略一：至多只允许四个哲学家同时进餐，以保证只有有一个哲学家能够进餐，最终总会释放出他所使用过的两根筷子，从而可使更多的哲学家进餐。定义信号量count，只允许4个哲学家同时进餐。

```
#define N 5
semaphore chopstick[5]={1,1,1,1,1};
semaphore count=4; // 设置一个count，最多有四个哲学家可以进来
void philosopher(int i)
{
    while(true)
    {
        think();
        down(&count); //请求进入房间进餐 当count为0时 不能允许哲学家再进来了
        take(chopstick[i]); //请求左手边的筷子
        take(chopstick[(i+1)%N]); //请求右手边的筷子
        eat();
        put(chopstick[i]); //释放左手边的筷子
        put(chopstick[(i+1)%N]); //释放右手边的筷子
        up(&count); //退出房间释放信号量
    }
}
```

策略二：可以利用AND 型信号量机制实现，也可以利用信号量的保护机制实现。利用信号量的保护机制实现的思想是通过记录型信号量mutex对取左侧和右侧筷子的操作进行保护，使之成为一个原子操作，这样可以防止死锁的出现。描述如下：

```
#define N 5
semaphore chopstick[5]={1,1,1,1,1};
semaphore mutex = 1; // 这个过程需要判断两根筷子是否可用，并保护起来
void philosopher(int i)
{
    while(true)
    {
        // 这个过程中可能只能由一个人在吃饭
        think();
```

```
        down(&mutex);  // 保护信号量
        take(chopstick[(i+1)%N]); //请求右手边的筷子
        take(chopstick[i]); //请求左手边的筷子
        up(&mutex); //释放保护信号量
        eat();
        put(chopstick[(i+1)%N]); //释放右手边的筷子
        put(chopstick[i]); //释放左手边的筷子
    }
}
```

策略三：规定**奇数号**的哲学家先拿起他**左边**的筷子，然后再去拿他**右边**的筷子；而**偶数号**的哲学家则先拿起他**右边**的筷子，然后再去拿他**左边**的筷子。按此规定，将是1、2号哲学家竞争1号筷子，3、4号哲学家竞争3号筷子。即五个哲学家都竞争奇数号筷子，获得后，再去竞争偶数号筷子，最后总会有一个哲学家能获得两支筷子而进餐。

```
#define N 5
semaphore chopstick[5]={1,1,1,1,1};
void philosopher(int i)
{
    while(true)
    {
        think();
        if(i%2 == 0) //偶数哲学家，先右后左。
        {
            take (chopstick[(i + 1)%N]) ;
            take (chopstick[i]) ;
            eat();
            put (chopstick[(i + 1)%N]) ;
            put (chopstick[i]) ;
        }
        else //奇数哲学家，先左后右。
        {
            take (chopstick[i]) ;
            take (chopstick[(i + 1)%N]) ;
            eat();
            put (chopstick[i]) ;
            put (chopstick[(i + 1)%N]) ;
        }
    }
}
```

6、进程通信

进程同步与进程通信很容易混淆，它们的区别在于：

- 进程同步：控制多个进程按一定顺序执行；
- 进程通信：进程间传输信息。

进程通信是一种手段，而进程同步是一种目的。也可以说，为了能够达到进程同步的目的，需要让进程进行通信，传输一些进程同步所需要的信息。

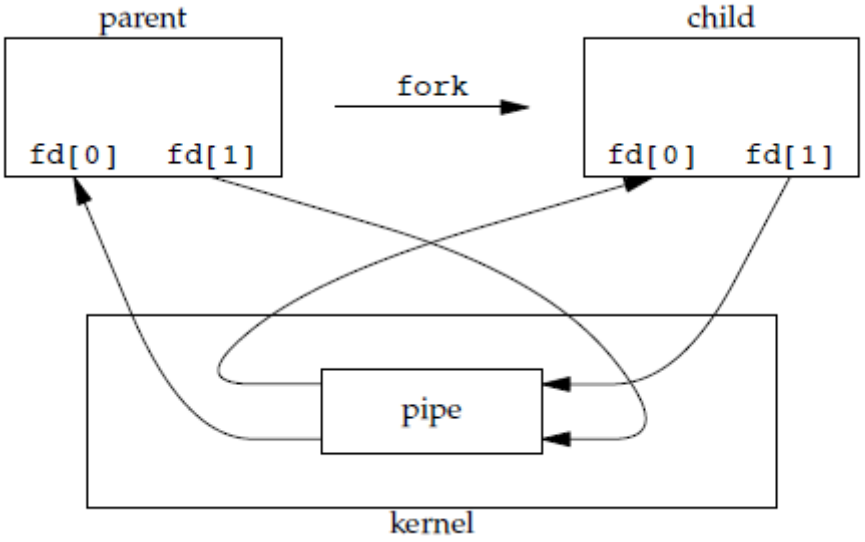
6.1管道

管道是通过调用 pipe 函数创建的，fd[0] 用于读，fd[1] 用于写。

```
#include <unistd.h>
int pipe(int fd[2]);
```

它具有以下限制：

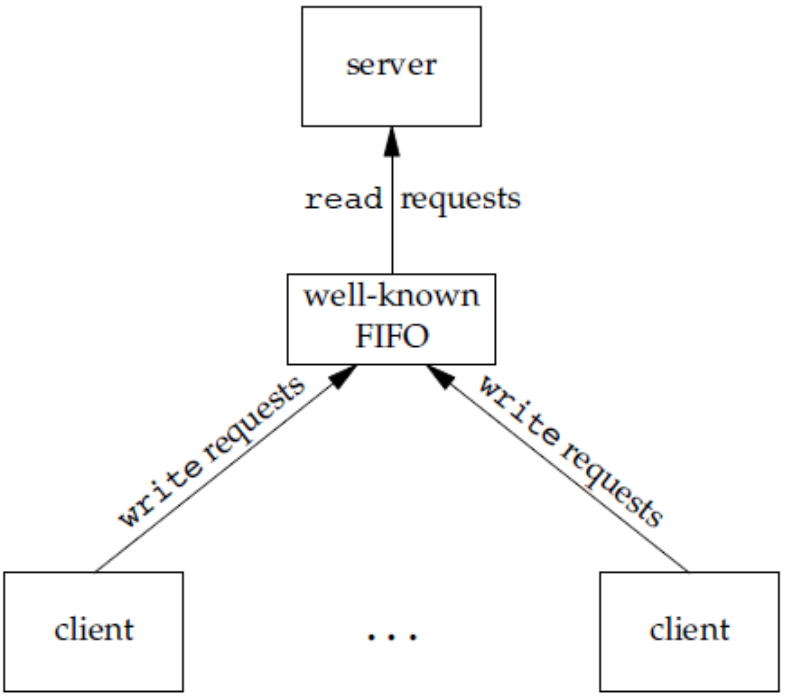
- 只支持半双工通信（单向交替传输）；
- 只能在父子进程或者兄弟进程中使用。



6.2FIFO

也称为命名管道，在内核中申请一块固定大小的缓冲区，程序拥有写入和读取的权利，没有血缘关系的进程也可以进程间通信。

FIFO 常用于客户-服务器应用程序中，FIFO 用作汇聚点，在客户进程和服务器进程之间传递数据。



6.3消息队列

消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列ID）来标识。

- 消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级。
- 消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除。
- 消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取。

6.4信号量

它是一个计数器，用于为多个进程提供对共享数据对象的访问。

6.5共享存储

允许多个进程共享一个给定的存储区。因为数据不需要在进程之间复制，所以这是最快的一种 IPC。

需要使用信号量用来同步对共享存储的访问。

多个进程可以将同一个文件映射到它们的地址空间从而实现共享内存。另外 XSI 共享内存不是使用文件，而是使用内存的匿名段。

6.6套接字

与其它通信机制不同的是，它可用于不同机器间的进程通信。

三、线程管理

1、线程基本概念

线程是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

线程共享资源

1. 进程代码段
2. 进程的公有数据(利用这些共享的数据，线程很容易的实现相互之间的通讯)
3. 进程打开的文件描述符、信号的处理器、进程的当前目录和进程用户ID与进程组ID。

线程独立资源包括

1. **线程ID**
   每个线程都有自己的线程ID，这个ID在本进程中是唯一的。进程用此来标识线程。
2. **寄存器组的值**
   由于线程间是并发运行的，每个线程有自己不同的运行线索，当从一个线程切换到另一个线程上 时，必须将原有的线程的寄存器集合的状态保存，以便将来该线程在被重新切换到时能得以恢复。
3. **线程的堆栈**
   堆栈是保证线程独立运行所必须的。线程函数可以调用函数，而被调用函数中又是可以层层嵌套的，所以线程必须拥有自己的函数堆栈， 使得函数调用可以正常执行，不受其他线程的影响。
4. **错误返回码**
   由于同一个进程中有很多个线程在同时运行，可能某个线程进行系统调用后设置了errno值，而在该 线程还没有处理这个错误，另外一个线程就在此时被调度器投入运行，这样错误值就有可能被修改。所以，不同的线程应该拥有自己的错误返回码变量。

5. **线程的信号屏蔽码**
   由于每个线程所感兴趣的信号不同，所以线程的信号屏蔽码应该由线程自己管理。但所有的线程都 共享同样的信号处理器。
6. 线程的优先级
   由于线程需要像进程那样能够被调度，那么就必须要有可供调度使用的参数，这个参数就是线程的优先级。

2、线程调度策略

linux内核的三种调度方法：

- SCHED_OTHER 分时调度策略。注意的是这排队跟上WC一样，前面的人占用了位置，它不出来，第二个人是轮不上的。
- SCHED_FIFO实时调度策略，先到先服务。一旦占用cpu则一直运行，直到有更高优先级任务到达或自己放弃。
- SCHED_RR实时调度策略，时间片轮转 。当任务的时间片用完，系统将重新分配时间片，并置于就绪队列尾。放在队列尾保证了所有具有相同优先级的RR任务的调度公平。

3、线程同步

3.1互斥锁

互斥锁机制是同一时刻只允许一个线程执行一个关键部分的代码。

3.2条件变量

条件变量是利用线程间共享全局变量进行同步的一种机制。条件变量上的基本操作有：触发条件(当条件变为 true 时)；等待条件，挂起线程直到其他线程触发条件。

3.3信号量

如同进程一样，线程也可以通过信号量来实现通信，虽然是轻量级的。

3.4读写锁

读写锁和互斥量类似，区别在于：互斥量只有两个状态，即锁住状态和不加锁状态，而且一次只有一个线程可以对其加锁。而读写锁可以有三个状态，读模式下

加锁状态，写模式下加锁状态，不加锁状态。一次只能有一个线程占有写模式的读写锁，但可以由多个线程同时占有读模式的读写锁。

读写锁非常适用于对数据结构读次数远大于写的情况。

4、线程通信

4.1全局变量

由于属于同一个进程的各个线程共享操作系统分配该进程的资源，故解决线程间通信最简单的一种方法是使用全局变量。对于标准类型的全局变量，我们建议使用volatile 修饰符，它告诉编译器无需对该变量作任何的优化，即无需将它放到一个寄存器中，并且该值可被外部改变。如果线程间所需传递的信息较复杂，我们可以定义一个结构，通过传递指向该结构的指针进行传递信息。

4.2自定义消息

我们可以在一个线程的执行函数中向另一个线程发送自定义的消息来达到通信的目的。一个线程向另外一个线程发送消息是通过操作系统实现的。利用Windows操作系统的消息驱动机制，当一个线程发出一条消息时，操作系统首先接收到该消息，然后把该消息转发给目标线程，接收消息的线程必须已经建立了消息循环。

4.3std::promise和std::future

std::promise 可以用来在线程间提供数据传递。

std::future = std::promise.get_future()。

线程中可以对promise赋值std::promise.set_value。

赋值之后std::future.get()就会返回其他线程中设置的值

4.4std::packaged_task

可以包裹一个函数, 有点类似std::function，不同之处在于这个可以通过get_future返回std::future对象来获取异步执行的函数结果。

4.5std::async

std::async提供异步执行的方法，std::future = std::async(...), 函数执行完成后可以通过std::future.get()获取到执行函数的返回值。

四、死锁

1、必要条件

死锁是指两个或两个以上的进程（线程）在运行过程中因争夺资源而造成的一种僵局（Deadly-Embrace)），若无外力作用，这些进程（线程）都将无法向前推进。

互斥：某种资源一次只允许一个进程访问，即该资源一旦分配给某个进程，其他进程就不能再访问，直到该进程访问结束。
占有和等待：一个进程本身占有资源（一种或多种），同时还有资源未得到满足，正在等待其他进程释放该资源。
不可抢占：已经分配给一个进程的资源不能强制性地被抢占，它只能被占有它的进程显式地释放。
环路等待：有两个或者两个以上的进程组成一条环路，该环路中的每个进程都在等待下一个进程所占有的资源。

2、处理方法

1. 鸵鸟策略
2. 死锁检测与死锁恢复
3. 死锁预防
4. 死锁避免

3、鸵鸟策略

把头埋在沙子里，假装根本没发生问题。

因为解决死锁问题的代价很高，因此鸵鸟策略这种不采取任务措施的方案会获得更高的性能。

当发生死锁时不会对用户造成多大影响，或发生死锁的概率很低，可以采用鸵鸟策略。

大多数操作系统，包括 Unix，Linux 和 Windows，处理死锁问题的办法仅仅是忽略它。

4、死锁检测与死锁恢复

允许系统在运行过程中发生死锁，但可设置检测机构及时检测死锁的发生，并采取适当措施加以清除。

主要有两种死锁检测方法：超时机制和检测是否存在环路。

在检测到发生死锁之后，可以使用进程回退或者事务回滚等机制，释放获取的资源，之后再重新执行。

InnoDB 存储引擎使用检测是否存在环路的方式，并且选择将回滚操作代价小的事务进行回滚。

5、死锁预防

死锁预防是在程序运行之前通过某些手段从而保证不发生死锁。

5.1破坏互斥条件

临界资源需要互斥访问，所以基本不能破坏互斥条件。但也可以使用一些手段来使得多个进程可以用共享的方式去使用临界资源。例如假脱机打印机技术允许若干个进程同时输出，唯一真正请求物理打印机的进程是打印机守护进程。

5.2破坏占有和等待条件

方法1：所有的进程在开始运行之前，必须一次性地申请其在整个运行过程中所需要的全部资源。

- 优点：简单易实施且安全。
- 缺点：因为某项资源不满足，进程无法启动，而其他已经满足了的资源也不会得到利用，严重降低了资源的利用率，造成资源浪费。使进程经常发生饥饿现象。

方法2：该方法是对第一种方法的改进，允许进程只获得运行初期需要的资源，便开始运行，在运行过程中逐步释放掉分配到的已经使用完毕的资源，然后再去请求新的资源。这样的话，资源的利用率会得到提高，也会减少进程的饥饿问题。

5.3破坏不可抢占条件

当一个已经持有了一些资源的进程在提出新的资源请求没有得到满足时，它必须释放已经保持的所有资源，待以后需要使用的时候再重新申请。这就意味着进程已占有的资源会被短暂地释放或者说是被抢占了。

该种方法实现起来比较复杂，且代价也比较大。释放已经保持的资源很有可能会导致进程之前的工作实效等，反复的申请和释放资源会导致进程的执行被无限的推迟，这不仅会延长进程的周转周期，还会影响系统的吞吐量。

5.4破坏环路等待

给资源统一编号，进程只能按编号顺序来请求资源。

但有些资源本身并不具有编号属性，如果加上编号的话，那么会让程序逻辑变得复杂。

6、死锁避免

在使用前进行判断，只允许不会产生死锁的进程申请资源。

死锁避免是利用额外的检验信息，在分配资源时判断是否会出现死锁，只在不会出现死锁的情况下才分配资源。
两种避免办法：
1、如果一个进程的请求会导致死锁，则不启动该进程
2、如果一个进程的增加资源请求会导致死锁 ，则拒绝该申请。

6.1银行家算法

银行家算法是从当前状态出发，按照系统各类资源剩余量逐个检查各进程需要申请的资源量，找到一个各类资源申请量均小于等于系统剩余资源量的进程P1。然后分配给该P1进程所请求的资源，假定P1完成工作后归还其占有的所有资源，更新系统剩余资源状态并且移除进程列表中的P1，进而检查下一个能完成工作的客

户，……。如果所有客户都能完成工作，则找到一个安全序列，银行家才是安全的。若找不到这样的安全序列，则当前状态不安全。

五、内存管理

1、虚拟内存

虚拟内存的目的是为了让物理内存扩充成更大的逻辑内存，从而让程序获得更多的可用内存。

为了更好的管理内存，操作系统将内存抽象成地址空间。每个程序拥有自己的地址空间，这个地址空间被分割成多个块，每一块称为一页。这些页被映射到物理内存，但不需要映射到连续的物理内存，也不需要所有页都必须在物理内存中。当程序引用到不在物理内存中的页时，由硬件执行必要的映射，将缺失的部分装入物理内存并重新执行失败的指令。

从上面的描述中可以看出，虚拟内存允许程序不用将地址空间中的每一页都映射到物理内存，也就是说一个程序不需要全部调入内存就可以运行，这使得有限的内存运行大程序成为可能。例如有一台计算机可以产生 16 位地址，那么一个程序的地址空间范围是 0~64K。该计算机只有 32KB 的物理内存，虚拟内存技术允许该计算机运行一个 64K 大小的程序。

2、分页系统地址映射

内存管理单元（MMU）管理着地址空间和物理内存的转换，其中的页表（Page table）存储着页（程序地址空间）和页框（物理内存空间）的映射表。

一个虚拟地址分成两个部分，一部分存储页面号，一部分存储偏移量。

下图的页表存放着 16 个页，这 16 个页需要用 4 个比特位来进行索引定位。例如对于虚拟地址（0010 000000000100），前 4 位是存储页面号 2，读取表项内容为（110 1），页表项最后一位表示是否存在于内存中，1 表示存在。后 12 位存

储偏移量。这个页对应的页框的地址为 （110 000000000100）。

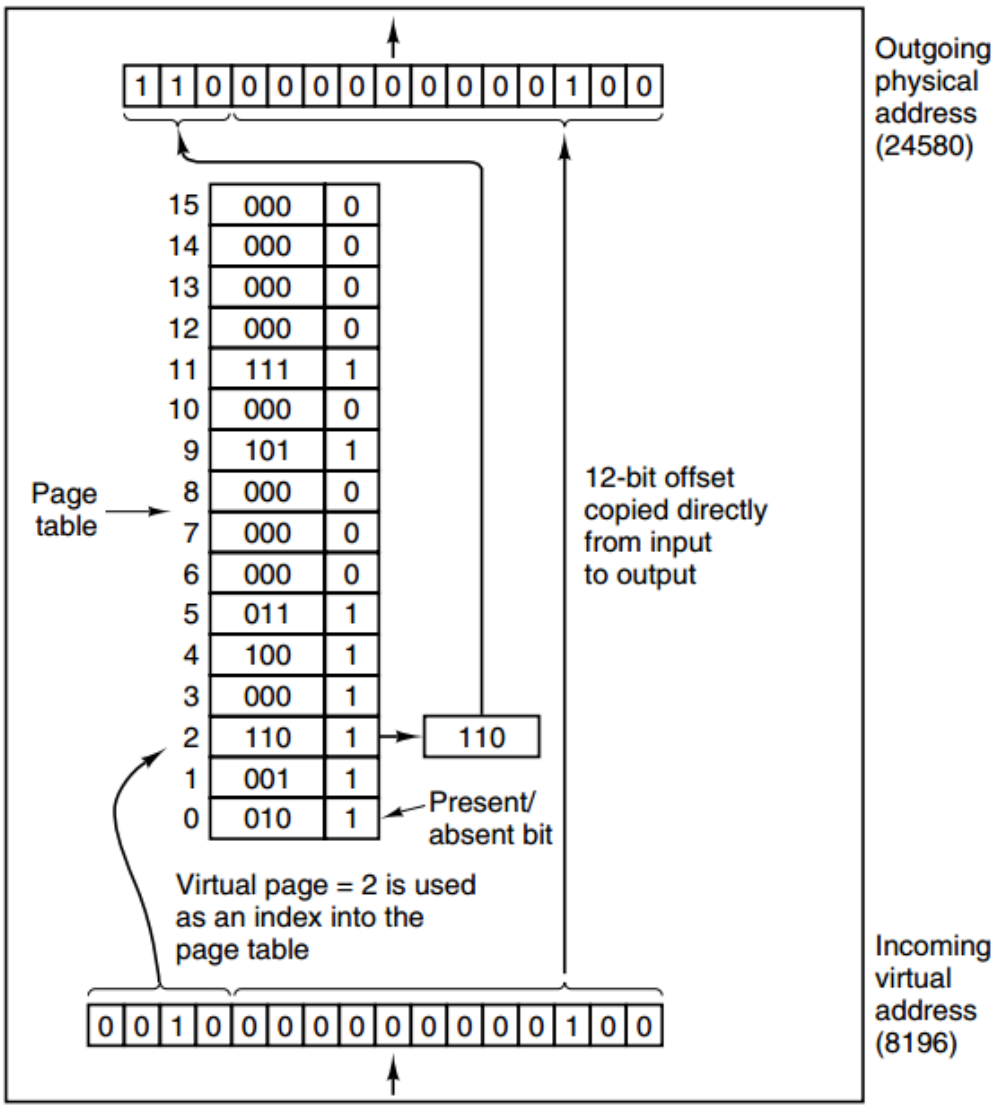


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

3、页面置换算法

在程序运行过程中，如果要访问的页面不在内存中，就发生缺页中断从而将该页调入内存中。此时如果内存已无空闲空间，系统必须从内存中调出一个页面到磁盘对换区中来腾出空间。

页面置换算法和缓存淘汰策略类似，可以将内存看成磁盘的缓存。在缓存系统中，缓存的大小有限，当有新的缓存到达时，需要淘汰一部分已经存在的缓存，这样才有空间存放新的缓存数据。

页面置换算法的主要目标是使页面置换频率最低（也可以说缺页率最低）。

3.1最佳

所选择的被换出的页面将是最长时间内不再被访问，通常可以保证获得最低的缺页率。

是一种理论上的算法，因为无法知道一个页面多长时间不再被访问。

3.2最近最久未使用（LRU）

为了实现 LRU，需要在内存中维护一个所有页面的链表。当一个页面被访问时，将这个页面移到链表表头。这样就能保证链表表尾的页面是最近最久未访问的。

LRU算法的描述： 设计一种缓存结构，该结构在构造时确定大小，假设大小为K，并有两个功能：

- set(key,value)：将记录(key,value)插入该结构。当缓存满时，将最久未使用的数据置换掉。
- get(key)：返回key对应的value值。

实现：最朴素的思想就是用数组+时间戳的方式，不过这样做效率较低。因此，我们可以用双向链表（LinkedList）+哈希表（HashMap）实现（链表用来表示位置，哈希表用来存储和查找）

3.3先进先出

选择换出的页面是最先进入的页面。

该算法会将那些经常被访问的页面也被换出，从而使缺页率升高。

3.4最近最少使用（LFU）

如果一个数据在最近一段时间很少被访问到，那么可以认为在将来它被访问的可能性也很小。因此，当空间满时，最小频率访问的数据最先被淘汰。

设计一种缓存结构，该结构在构造时确定大小，假设大小为 K，并有两个功能：

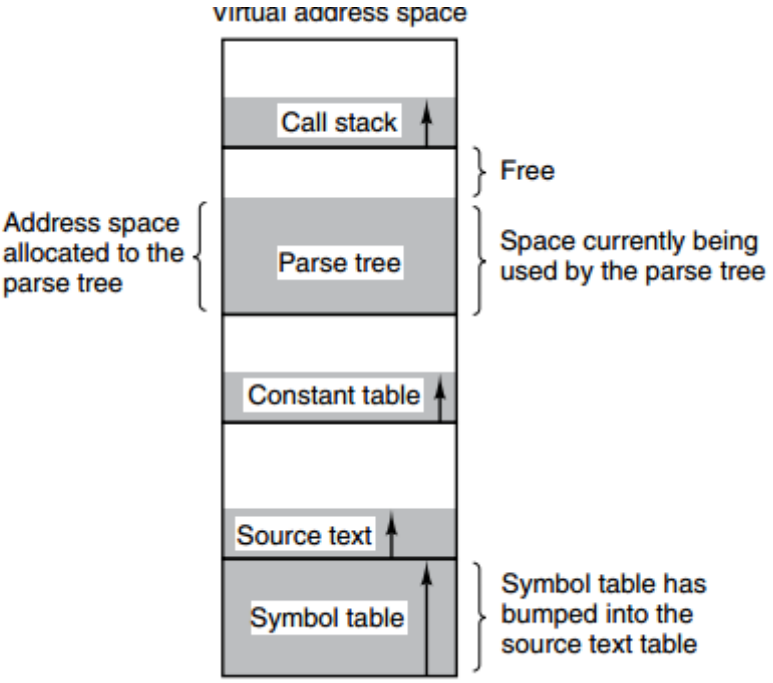
- set(key,value)：将记录(key,value)插入该结构。当缓存满时，将访问频率最低的数据置换掉。
- get(key)：返回key对应的value值。

算法实现策略：考虑到 LFU 会淘汰访问频率最小的数据，我们需要一种合适的方法按大小顺序维护数据访问的频率。LFU 算法本质上可以看做是一个 top K 问题 (K = 1)，即选出频率最小的元素，因此我们很容易想到可以用二项堆来选择频率最小的元素，这样的实现比较高效。最终实现策略为**小顶堆+哈希表**。

4、分段

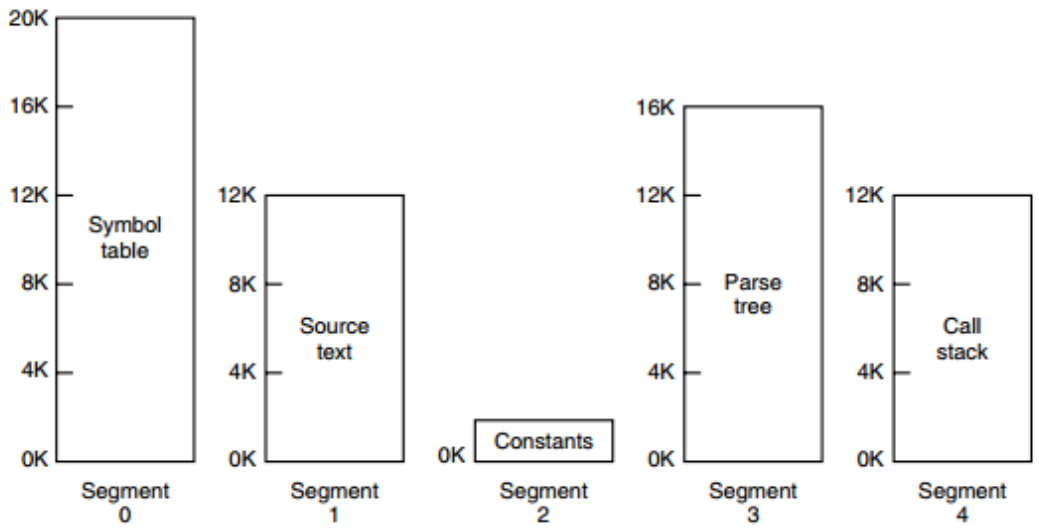
虚拟内存采用的是分页技术，也就是将地址空间划分成固定大小的页，每一页再与内存进行映射。

下图为一个编译器在编译过程中建立的多个表，有 4 个表是动态增长的，如果使用分页系统的一维地址空间，动态增长的特点会导致覆盖问题的出现。



分段的做法是把每个表分成段，一个段构成一个独立的地址空间。每个段的长度

可以不同，并且可以动态增长。



4.1段页式

程序的地址空间划分成多个拥有独立地址空间的段，每个段上的地址空间划分成大小相同的页。这样既拥有分段系统的共享和保护，又拥有分页系统的虚拟内存功能。

4.2分页与分段的比较

对程序员的透明性：分页透明，但是分段需要程序员显式划分每个段。

地址空间的维度：分页是一维地址空间，分段是二维的。

大小是否可以改变：页的大小不可变，段的大小可以动态改变。

出现的原因：分页主要用于实现虚拟内存，从而获得更大的地址空间；分段主要是为了使程序和数据可以被划分为逻辑上独立的地址空间并且有助于共享和保护。

六、设备管理

1、磁盘结构

盘面（Platter）：一个磁盘有多个盘面；

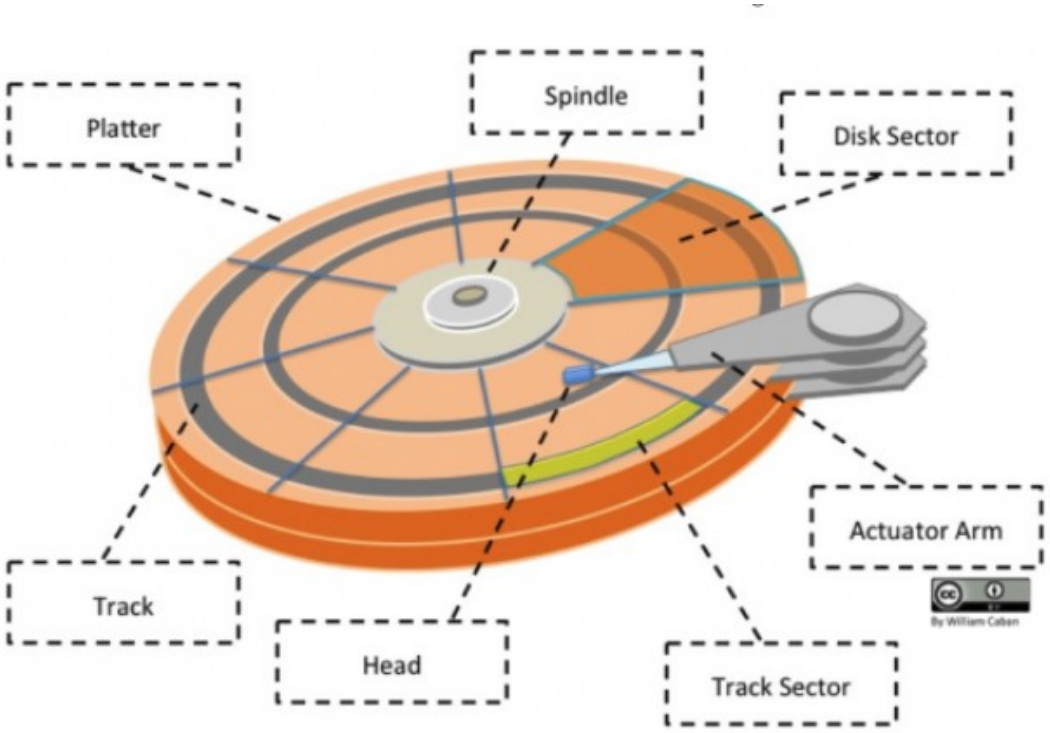
磁道（Track）：盘面上的圆形带状区域，一个盘面可以有多个磁道；

扇区（Track Sector）：磁道上的一个弧段，一个磁道可以有多个扇区，它是最小的物理储存单位，目前主要有 512 bytes 与 4 K 两种大小；

磁头（Head）：与盘面非常接近，能够将盘面上的磁场转换为电信号（读），或者将电信号转换为盘面的磁场（写）；

制动手臂（Actuator arm）：用于在磁道之间移动磁头；

主轴（Spindle）：使整个盘面转动。



2、磁盘调度算法

读写一个磁盘块的时间的影响因素有：

旋转时间（主轴转动盘面，使得磁头移动到适当的扇区上）

寻道时间（制动手臂移动，使得磁头移动到适当的磁道上）

实际的数据传输时间

其中，寻道时间最长，因此磁盘调度的主要目标是使磁盘的平均寻道时间最短。

2.1先来先服务(FCFS)

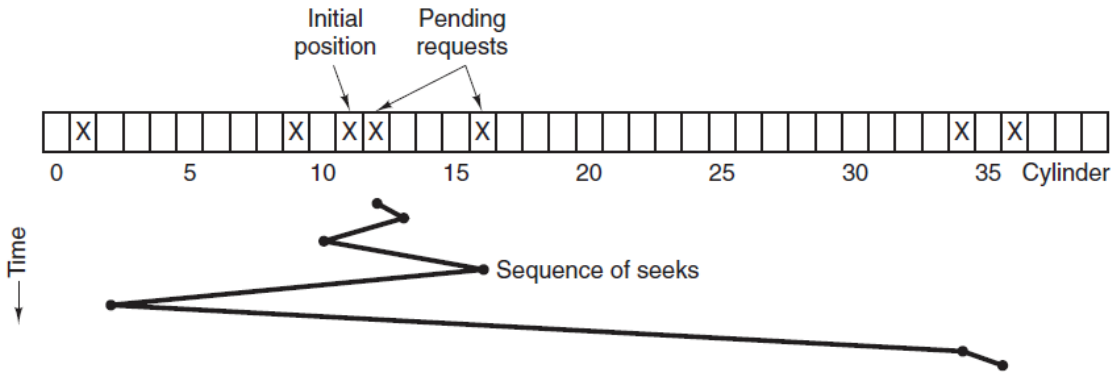
按照磁盘请求的顺序进行调度。

优点是公平和简单。缺点也很明显，因为未对寻道做任何优化，使平均寻道时间可能较长。

2.2最短寻道时间优先(SSTF)

优先调度与当前磁头所在磁道距离最近的磁道。

虽然平均寻道时间比较低，但是不够公平。如果新到达的磁道请求总是比一个在等待的磁道请求近，那么在等待的磁道请求会一直等待下去，也就是出现饥饿现象。具体来说，两端的磁道请求更容易出现饥饿现象。

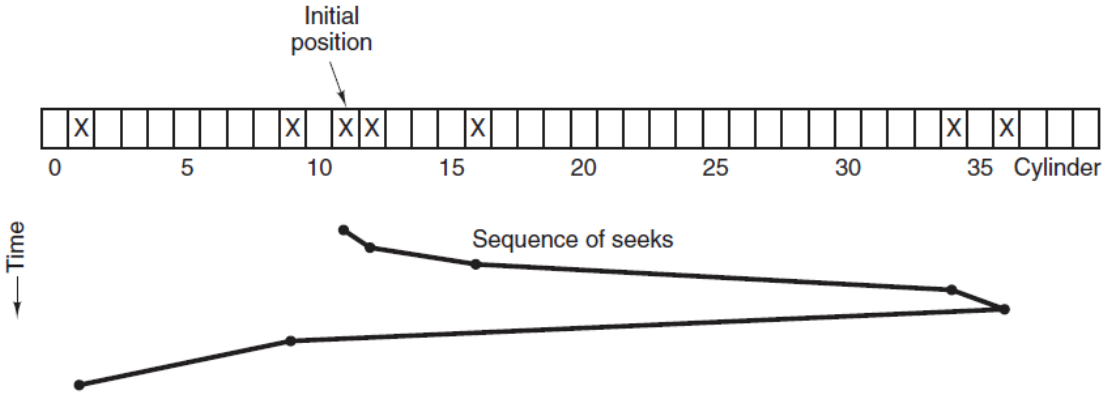


2.3电梯算法(SCAN)

电梯总是保持一个方向运行，直到该方向没有请求为止，然后改变运行方向。

电梯算法（扫描算法）和电梯的运行过程类似，总是按一个方向来进行磁盘调度，直到该方向上没有未完成的磁盘请求，然后改变方向。

因为考虑了移动方向，因此所有的磁盘请求都会被满足，解决了 SSTF 的饥饿问题。



七、编译和链接

1、编译系统

1.1预处理（产生.i文件）

```
g++ -E helloworld.cpp -o helloworld.i
```

1. 将#define删除，并将宏定义展开
2. 处理一些条件预编译指令，如#ifndef,#ifdef,#elif,#else,#endif（作用：防止重复包含头文件）等。将不必要的代码过滤掉。
3. 处理#include预编译指令，将被包含的文件插入到该预编译指令的位置。这个过程是递归进行的，因为被包含的文件也包含其他文件
4. 过滤掉所有注释里面的内容
5. 添加行号和文件名标识
6. 保留#program编译器指令，因为编译器需要使用他们

1.2编译（产生.s文件）

```
g++ -S helloworld.i -o helloworld.s
```

编译就是将预处理的文件进行一系列的词法分析、语法分析、语义分析，以及优化后产生相应的汇编代码文件。

1.3汇编（产生.o或.obj文件）

汇编过程实际上是把汇编语言代码翻译成目标机器指令的过程，即生成目标文件。目标文件中所存放的也就是与源程序等效的目标机器语言代码。目标文件由段组成，通常一个目标文件至少有两个段：

- 代码段：该段中所包含的主要是程序的指令，该段一般是可读和可执行的，但一般不可写。
- 数据段：主要存放程序中要用到的各种全局变量或静态的数据。一般数据段都是可读、可写、可执行的。

1.4链接（产生.out或.exe文件）

链接就是把每个源代码独立的编译，然后按照它们的要求将它们组装起来，链接主要解决的是源代码之间的相互依赖问题，链接的过程包括地址和空间的分配，符号决议，和重定位等这些步骤。

二、静态链接

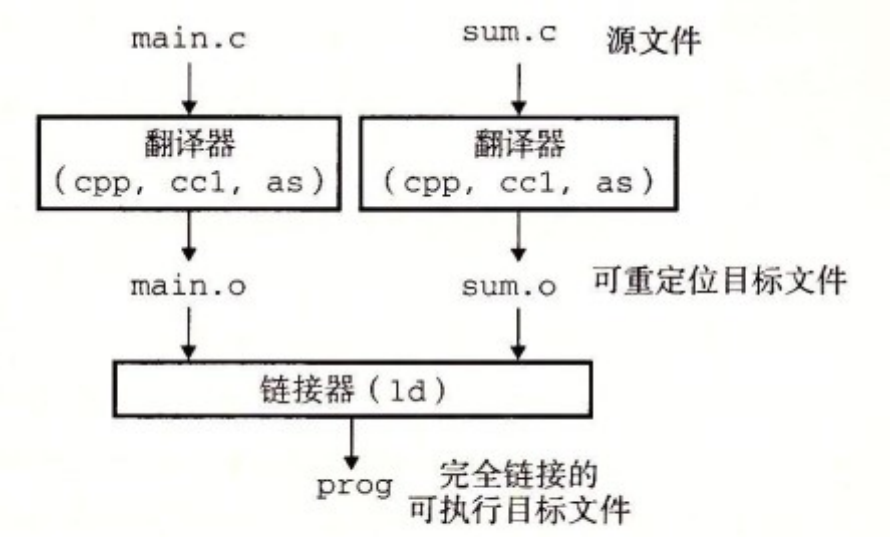


图 7-2 静态链接。链接器将可重定位目标文件组合起来，形成一个可执行目标文件 prog

静态链接器以一组可重定位目标文件为输入，生成一个完全链接的可执行目标文件作为输出。链接器主要完成以下两个任务：

- 符号解析：每个符号对应于一个函数、一个全局变量或一个静态变量，符号解析的目的是将每个符号引用与一个符号定义关联起来。
- 重定位：链接器通过把每个符号定义与一个内存位置关联起来，然后修改所有对这些符号的引用，使得它们指向这个内存位置。

三、目标文件

可执行目标文件：可以直接在内存中执行；

可重定位目标文件：可与其它可重定位目标文件在链接阶段合并，创建一个可执行目标文件；

共享目标文件：这是一种特殊的可重定位目标文件，可以在运行时被动态加载进内存并链接；

四、动态链接

静态库有以下两个问题：

- 当静态库更新时那么整个程序都要重新进行链接；
- 对于 printf 这种标准函数库，如果每个程序都要有代码，这会极大浪费资源。

共享库是为了解决静态库的这两个问题而设计的，在 Linux 系统中通常用 .so 后缀来表示，Windows 系统上它们被称为 DLL。它具有以下特点：

- 在给定的文件系统中一个库只有一个文件，所有引用该库的可执行目标文件都共享这个文件，它不会被复制到引用它的可执行文件中；
- 在内存中，一个共享库的 .text 节（已编译程序的机器代码）的一个副本可以被不同的正在运行的进程共享。

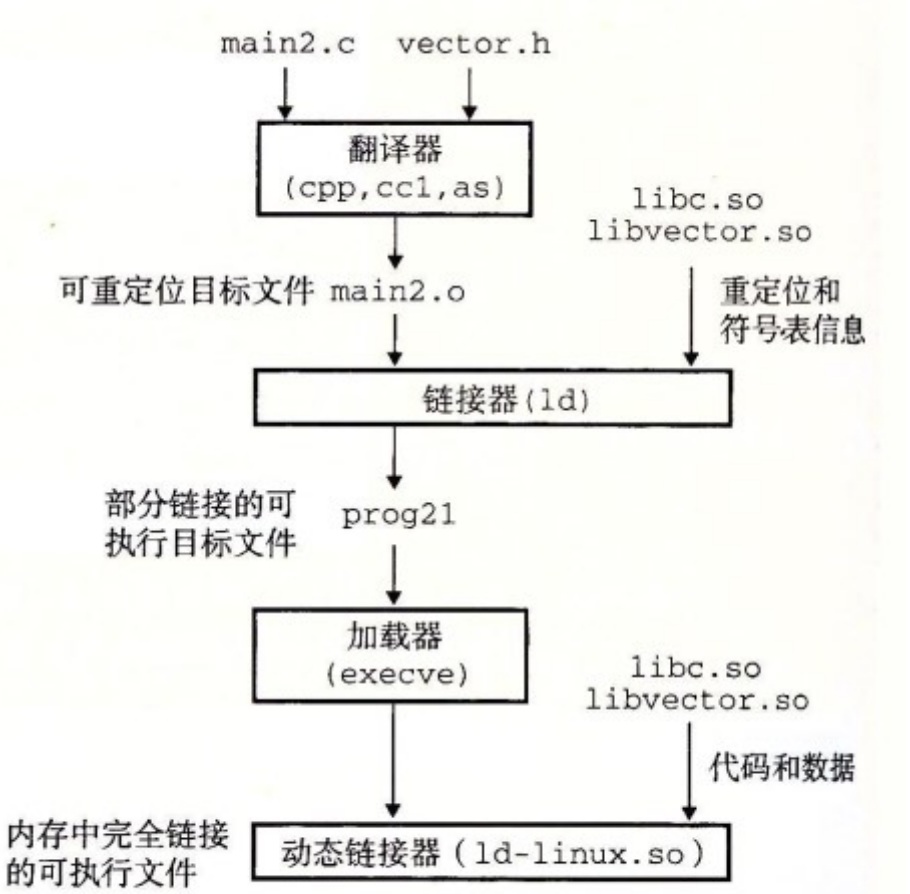


图 7-16　动态链接共享库