一、概述

基本特征

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**。

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**，例如打印机等，在同一时间只允许一个进程访问，需要用同步机制来实现对临界资源的访问。

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：**时分复用技术和空分复用技术**。

多个进程能在同一个处理器上并发执行使用了**时分复用技术**，让每个进程轮流占有处理器，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**，当使用到一个没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**。

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。

基本功能

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. 内存管理

内存分配、地址映射、内存保护与共享、虚拟内存等。

3. 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。

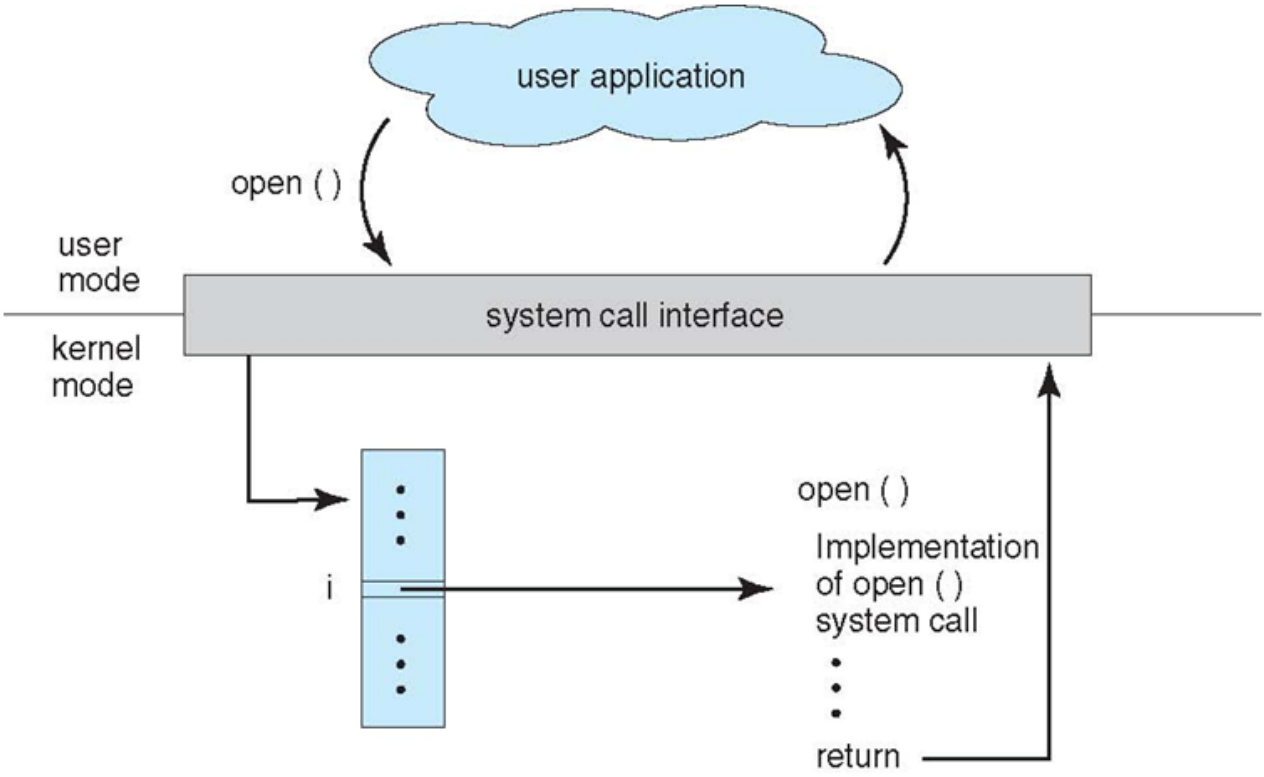
4. 设备管理

完成用户的 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。

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();

大内核和微内核

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到内核。

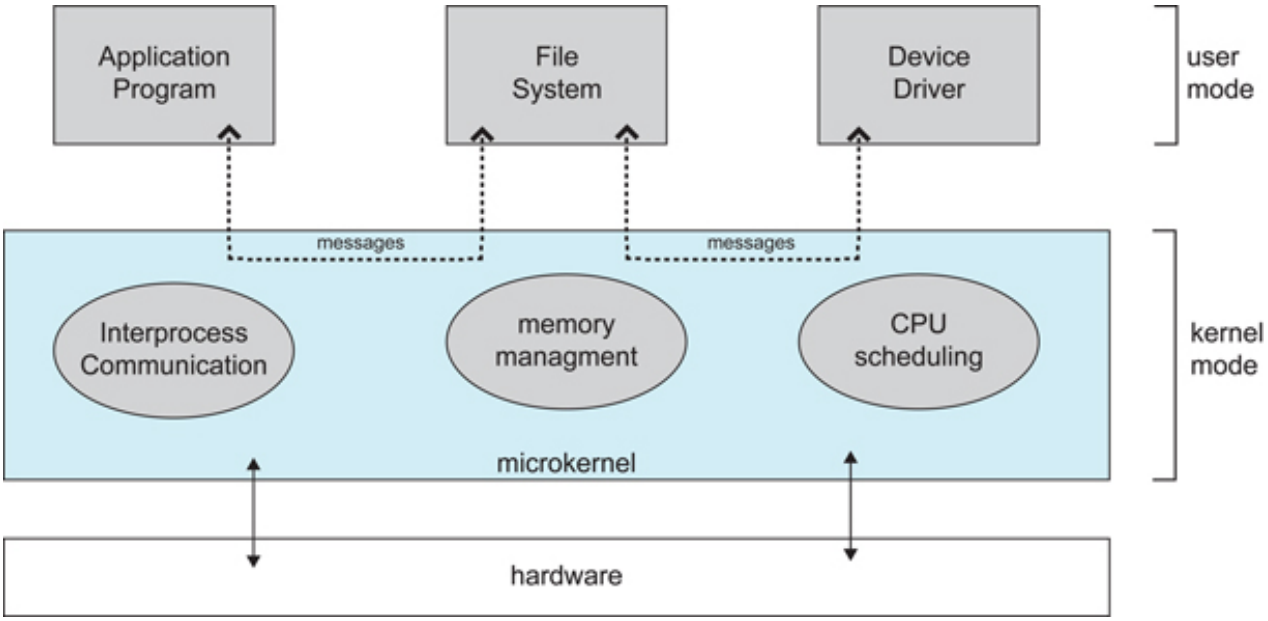
由于**各模块共享信息，因此有很高的性能**。

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进行切换**，所以会有一定的性能损失。



中断分类

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。

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、算术溢出等。

3. 陷入

在用户程序中使用系统调用。

二、进程管理

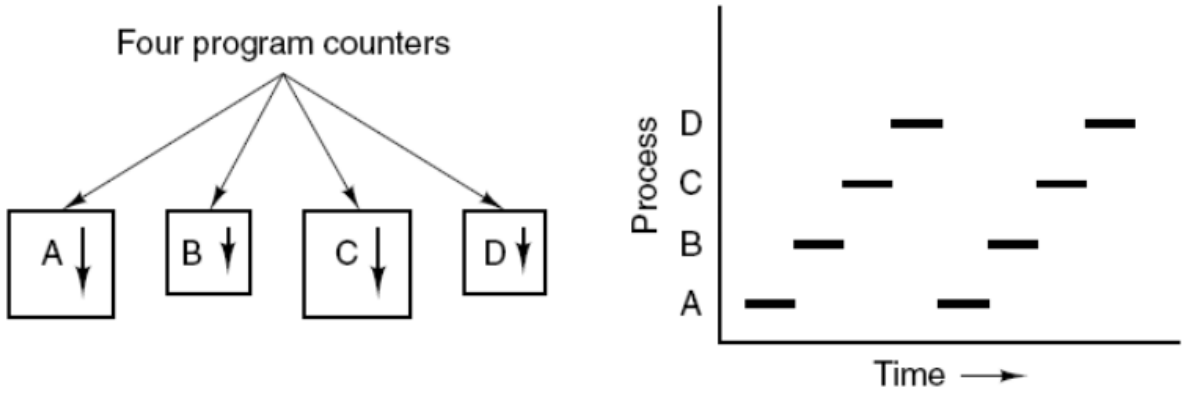
进程与线程

1. 进程

进程是**资源分配的基本单位**。

进程控制块 (Process Control Block, 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。

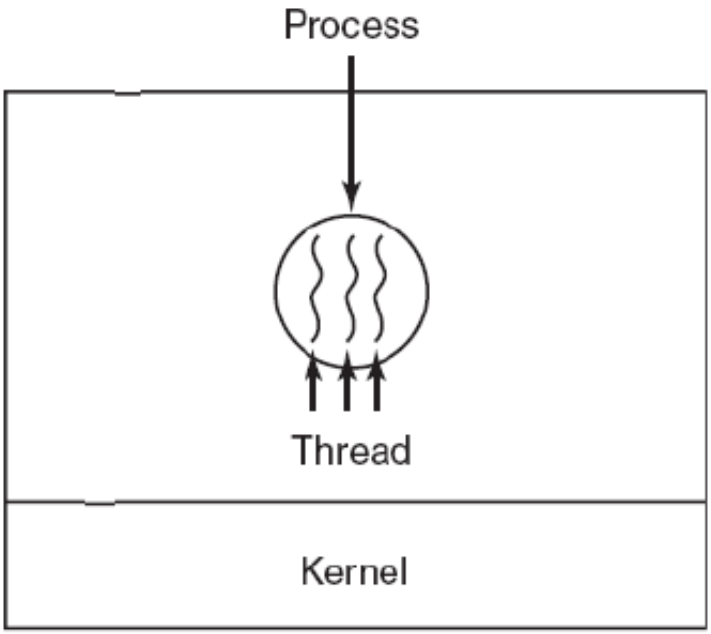


2. 线程

线程是**CPU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它事件。



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。

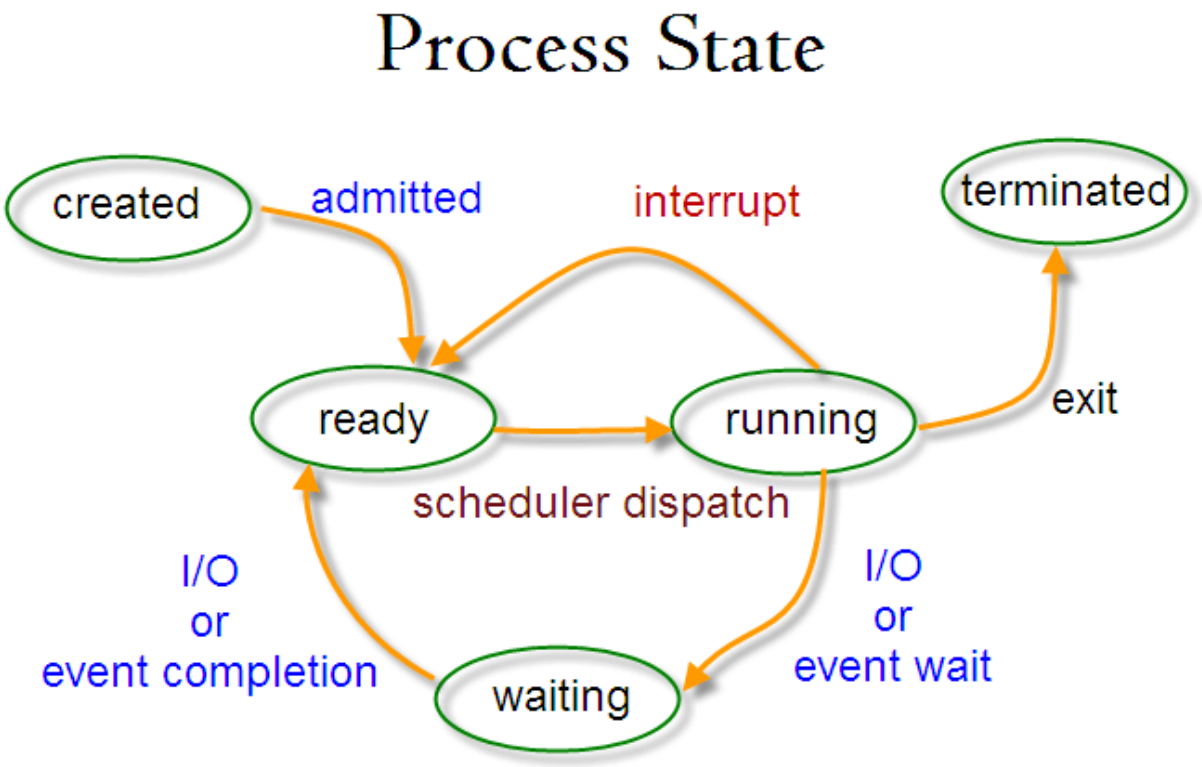
III 系统开销

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，Inter-Process Communication。

进程状态的切换



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

应该注意以下内容：

- **只有就绪态和运行态可以相互转换，其它的都是单向转换**。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 时间片用完之后就会转为就绪状态，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 时间**，缺少 CPU 时间会从运行态转换为就绪态。

进程调度算法

不同环境的调度算法目标不同，因此需要针对不同环境来讨论调度算法。

1. 批处理系统（追求吞吐量）

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）**。

1.1 先来先服务（FCFS）

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**。

1.2 短作业优先（SJF）

按估计运行时间最短的顺序进行调度。

长作业有可能会饿死，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。

1.3 最短剩余时间优先shortest remaining time next（SRTN）

按估计剩余时间最短的顺序进行调度。

2. 交互式系统（要求快速响应）

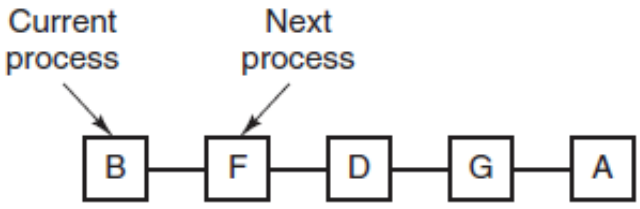
交互式系统有大量的用户交互操作，在该系统中调度算法的**目标是快速地进行响应**。

2.1 时间片轮转（每个进程排队，定时执行一个时间片）

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。



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（防止低优先级饿死）**。

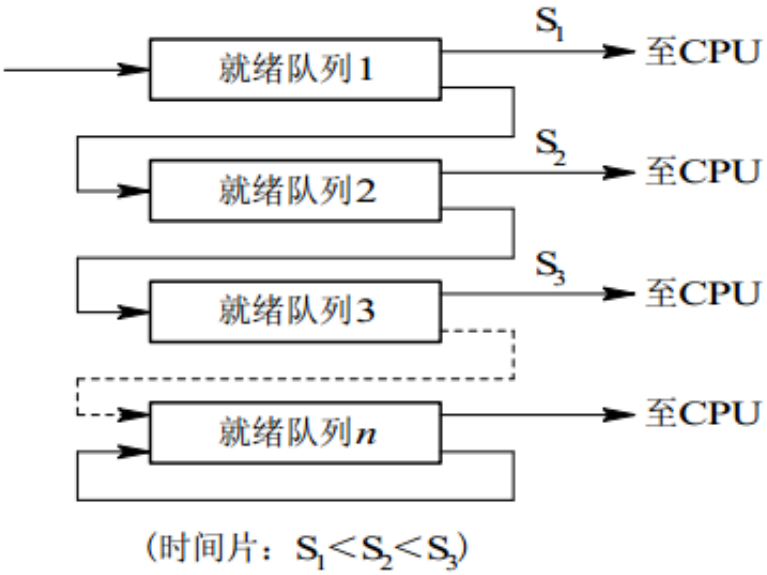
2.3 多级反馈队列（优先级高先被选择，但低优先级时间片多）

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。

多级队列是为这种需要连续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. 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**。

进程同步

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**。

```
// entry section
// critical section;
// exit section
```

2. 同步与互斥

- 同步：**多个进程按一定顺序执行**；
- 互斥：**多个进程在同一时刻只有一个进程能进入临界区**。

3. 信号量

信号量（Semaphore）是一个**整型变量**，可以对其执行 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);
}
```

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 记录空缓冲区的数量，full 记录满缓冲区的数量**。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)。如果这么做了，那么可能会出现这种情况：生产者对缓冲区加锁后，执行 down(empty) 操作，发现 empty = 0，此时生产者睡眠。消费者不能进入临界区，因为生产者对缓冲区加锁了，消费者就无法执行 up(empty) 操作，empty 永远都为 0，导致生产者永远等待下，不会释放锁，消费者因此也会永远等待下去。

```
#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. 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。

c 语言不支持管程，下面的示例代码使用了类 Pascal 语言来描述管程。示例代码的管程提供了 insert() 和 remove() 方法，客户端代码通过调用这两个方法来解决生产者-消费者问题。

```
monitor ProducerConsumer
    integer i;
    condition c;

    procedure insert();
    begin
        // ...
    end;

    procedure remove();
    begin
        // ...
    end;
end monitor;
```

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，否者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。

```
// 管程
monitor ProducerConsumer
    condition full, empty;
    integer count := 0;
    condition c;

    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty);
    end;

    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N -1 then signal(full);
    end;
end monitor;

// 生产者客户端
procedure producer
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item);
    end
end;

// 消费者客户端
procedure consumer
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item);
    end
end;
```

经典同步问题

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。

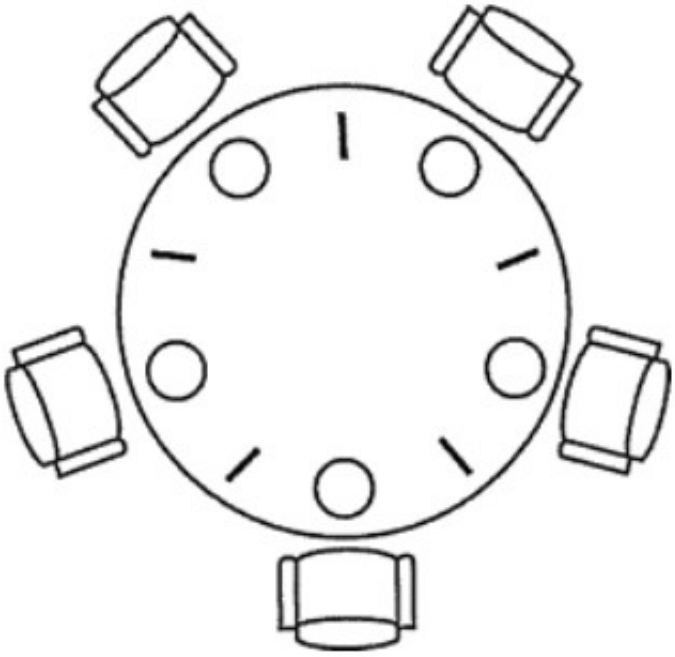
一个整型变量 count 记录在对数据进行读操作的进程数量，一个互斥量 count_mutex 用于对 count 加锁，一个互斥量 data_mutex 用于对读写的数据加锁。

```
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); //最后一个读者退出前释放数据锁，让写者可以获取锁
        up(&count_mutex);
    }
}

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

2. 哲学家就餐问题



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。

下面是一种错误的解法，考虑到如果所有哲学家同时拿起左手边的筷子，那么就无法拿起右手边的筷子，造成死锁。

```
#define N 5

void philosopher(int i) {
    while(TRUE) {
        think();
        take(i);        // 拿起左边的筷子
        take((i+1)%N); // 拿起右边的筷子
        eat();
        put(i);
        put((i+1)%N);
    }
}
```

为了防止死锁的发生，可以设置两个条件：

- 必须同时拿起左右两根筷子；
- 只有在两个邻居都没有进餐的情况下才允许进餐。

```
#define N 5
#define LEFT (i + N - 1) % N // 左邻居
#define RIGHT (i + 1) % N    // 右邻居
#define THINKING 0
#define HUNGRY   1
#define EATING   2
typedef int semaphore;
int state[N];                // 跟踪每个哲学家的状态
```

```
semaphore mutex = 1;          // 临界区的互斥
semaphore s[N];               // 每个哲学家一个信号量

void philosopher(int i) {
    while(TRUE) {
        think();
        take_two(i);
        eat();
        put_two(i);
    }
}

void take_two(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_two(i) {
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test(i) {          // 尝试拿起两把筷子
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

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。

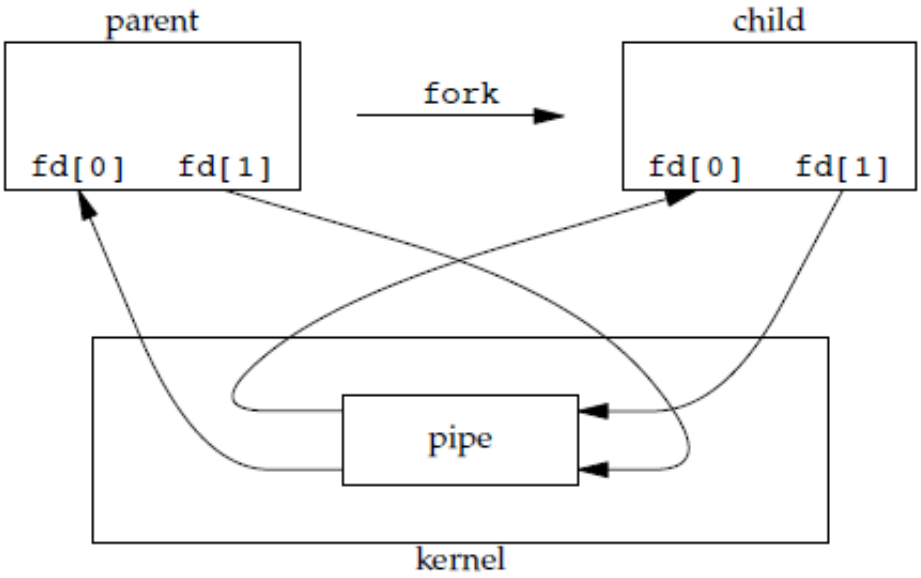
1. 管道

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）；
- **只能在父子进程中使用。**

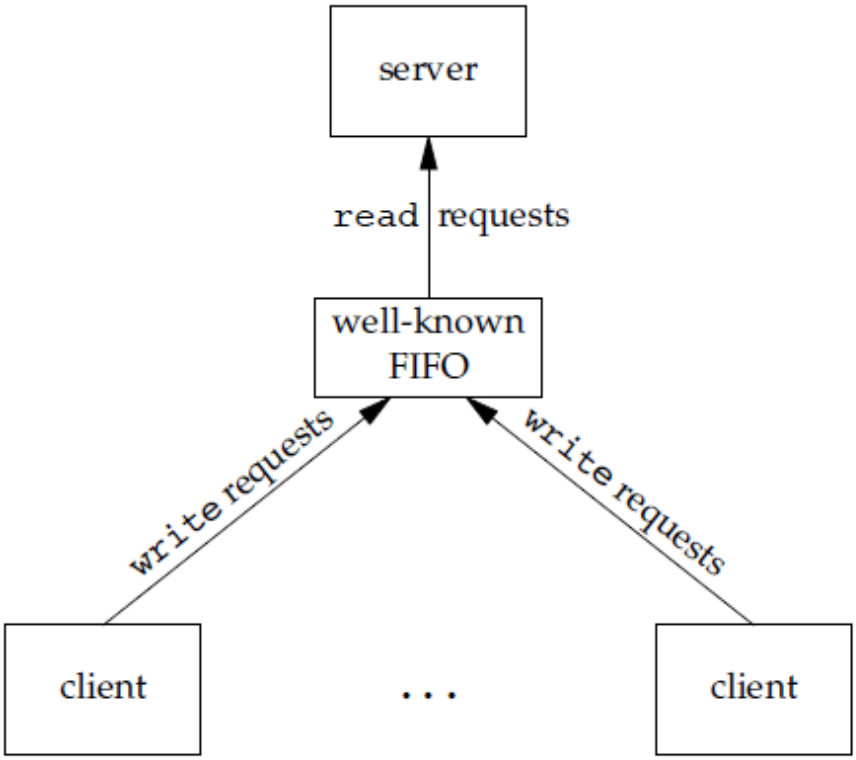


2. FIFO，命名管道

也称为**命名管道，去除了管道只能在父子进程中使用的限制。**

```
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
int mkfifoat(int fd, const char *path, mode_t mode);
```

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。**



3. 消息队列

相比于 FIFO，消息队列具有以下优点：

- 消息队列**可以独立于读写进程存在**，从而避免了 FIFO 中同步管道的**打开和关闭时可能产生的困难**；
- **避免了 FIFO 的同步阻塞问题，不需要进程自己提供同步方法**；
- 读进程可以**根据消息类型有选择地接收消息，而不像 FIFO 那样只能默认地接收**。

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**。

5. 共享存储（最快的）

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. 套接字（唯一可用于不同机器间进程通信）

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、死锁

必要条件

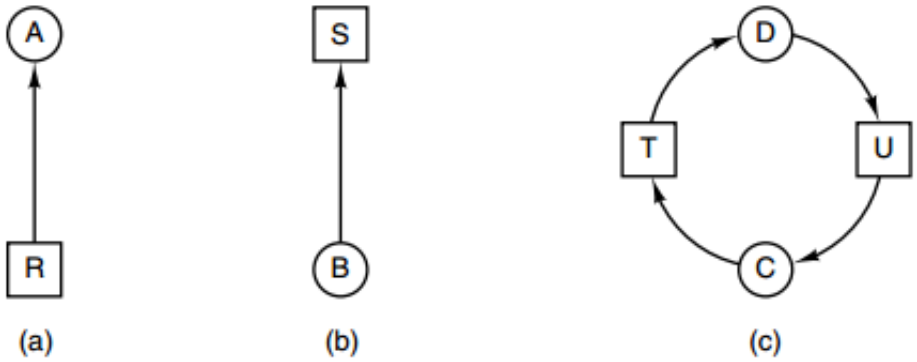


Figure 6-3. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

- **互斥**：每个资源要么已经分配给了一个进程，要么就是可用的。
- **占有和等待**：已经得到了某个资源的进程可以再请求新的资源。
- **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。

处理方法

主要有以下四种方法：

- **鸵鸟策略**
- **死锁检测与死锁恢复**
- **死锁预防**
- **死锁避免**

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。

死锁检测与恢复（检测当前状态是否死锁，会找出死锁进程并尝试恢复）

不试图阻止死锁，而是当检测到死锁发生时，采取措施进行恢复。

1. 每种类型一个资源的死锁检测

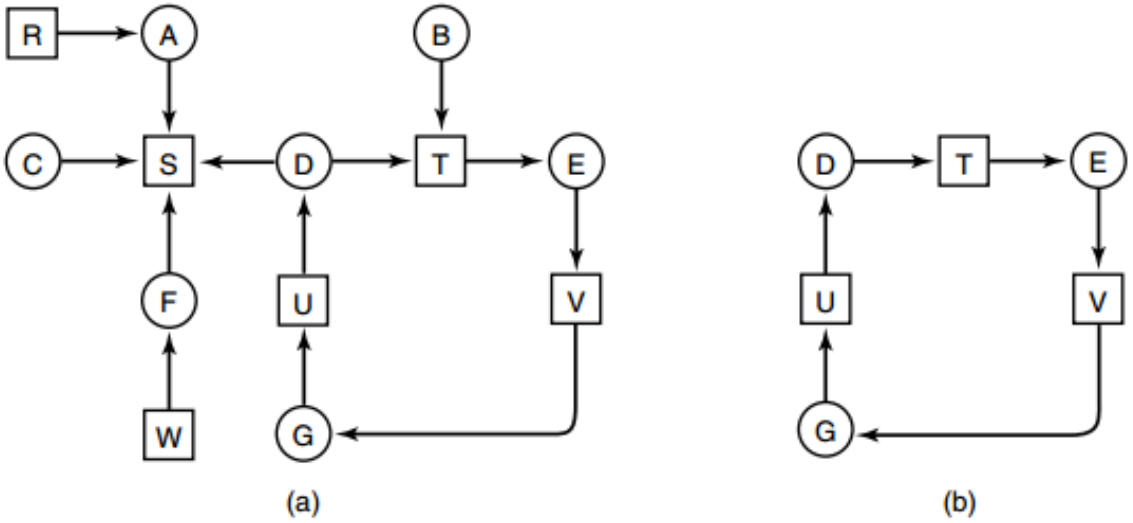


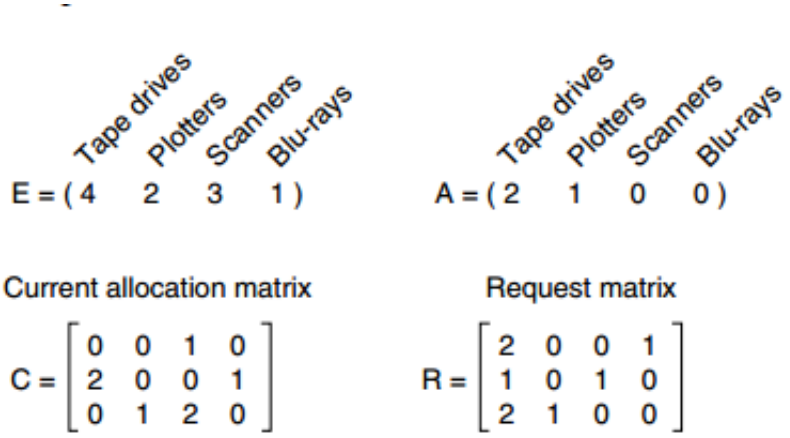
Figure 6-5. (a) A resource graph. (b) A cycle extracted from (a).

上图为资源分配图，其中**方框表示资源，圆圈表示进程**。资源指向进程表示该资源已经分配给该进程，**进程指向资源表示进程请求获取该资源**。

图 a 可以抽取出环，如图 b，它满足了环路等待条件，因此会发生死锁。

每种类型一个资源的死锁检测算法是**通过检测有向图是否存在环来实现**，从一个节点出发进行**深度优先搜索**，对访问过的节点进行标记，如果访问了已经标记的节点，就表示有向图存在环，也就是检测到死锁的发生。

2. 每种类型多个资源的死锁检测



上图中，有**三个进程四个资源**，每个数据代表的含义如下：

- E 向量：资源总量
- A 向量：资源剩余量
- C 矩阵：每个进程所拥有的资源数量，每一行都代表一个进程拥有资源的数量
- R 矩阵：每个进程请求的资源数量

进程 P1 和 P2 所请求的资源都得不到满足，只有进程 P3 可以，让 P3 执行，之后释放 P3 拥有的资源，此时 A = (2 2 2 0)。P2 可以执行，执行后释放 P2 拥有的资源，A = (4 2 2 1) 。P1 也可以执行。所有进程都可以顺利执行，没有死锁。

算法总结如下：

每个进程最开始时都不被标记，执行过程有可能被标记。当算法结束时，任何没有被标记的进程都是死锁进程。

1. 寻找一个没有标记的进程 Pi，它所请求的资源小于等于 A。
2. 如果找到了这样一个进程，那么将 C 矩阵的第 i 行向量加到 A 中，**标记该进程**，并转回 1。
3. **如果没有这样一个进程，算法终止。**

3. 死锁恢复

- 利用**抢占**恢复
- 利用**回滚**恢复
- 通过**杀死进程**恢复

死锁预防（在运行前）

在程序运行之前预防发生死锁。

1. 破坏互斥条件

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。**

2. 破坏占有和等待条件

一种实现方式是**规定所有进程在开始执行前请求所需要的全部资源。**

3. 破坏不可抢占条件

4. 破坏环路等待条件

给资源统一编号，进程只能按编号顺序来请求资源。

死锁避免（在运行时去试探一个请求到达的状态是否安全，不去安全就拒绝请求）

在程序运行时避免发生死锁。

1. 安全状态

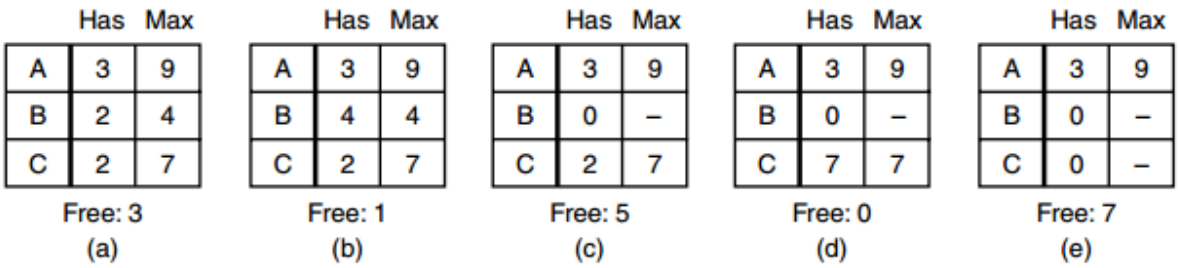


Figure 6-9. Demonstration that the state in (a) is safe.

图 a 的第二列 Has 表示已拥有的资源数，第三列 Max 表示总共需要的资源数，Free 表示还有可以使用的资源数。从图 a 开始出发，先让 B 拥有所需的所有资源（图 b），运行结束后释放 B，此时 Free 变为 5（图 c）；接着以同样的方式运行 C 和 A，使得所有进程都能成功运行，**因此可以称图 a 所示的状态是安全的。**

定义：如果没有死锁发生，并且即使所有进程突然请求对资源的最大需求，也仍然存在某种调度次序能够使得每一个进程运行完毕，则称该状态是安全的。

安全状态的检测与死锁的检测类似，因为安全状态必须要求不能发生死锁。下面的银行家算法与死锁检测算法非常类似，可以结合着做参考对比。

安全状态检测和死锁检测区别

- 安全状态是去试探请求会导致的状态是否安全（**试探的状态还没进入**）
- 死锁检测是检测当前的状态是否已经死锁（**检测的状态已经进入**）

2. 单个资源的银行家算法

一个小城镇的银行家，他向一群客户分别承诺了一定的贷款额度，**算法要做的是判断对请求的满足是否会进入不安全状态**，如果是，就拒绝请求；否则予以分配。

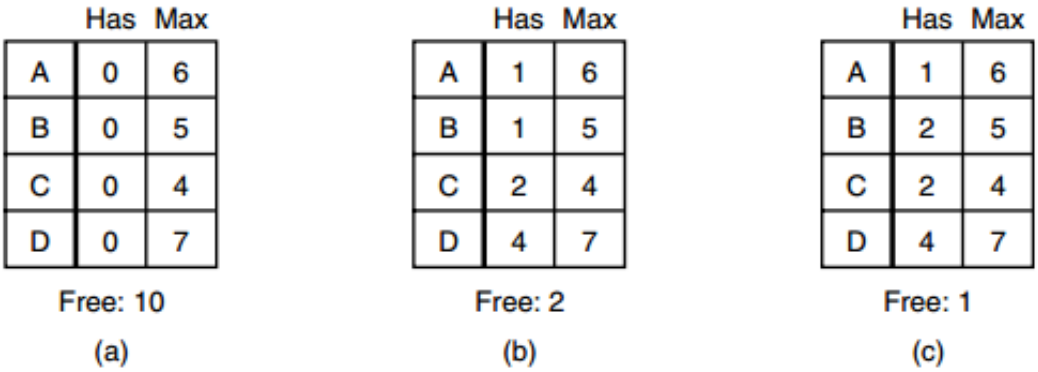


Figure 6-11. Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

上图 c 为不安全状态，**因此算法会拒绝之前的请求**，从而避免进入图 c 中的状态。

3. 多个资源的银行家算法

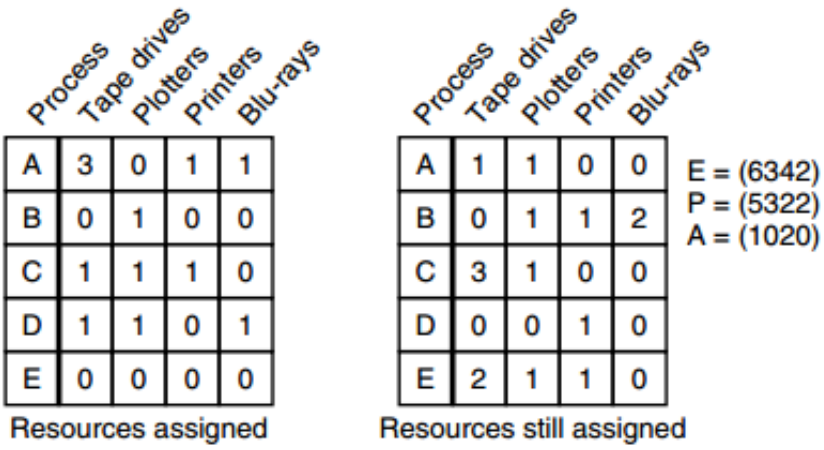


Figure 6-12. The banker's algorithm with multiple resources.

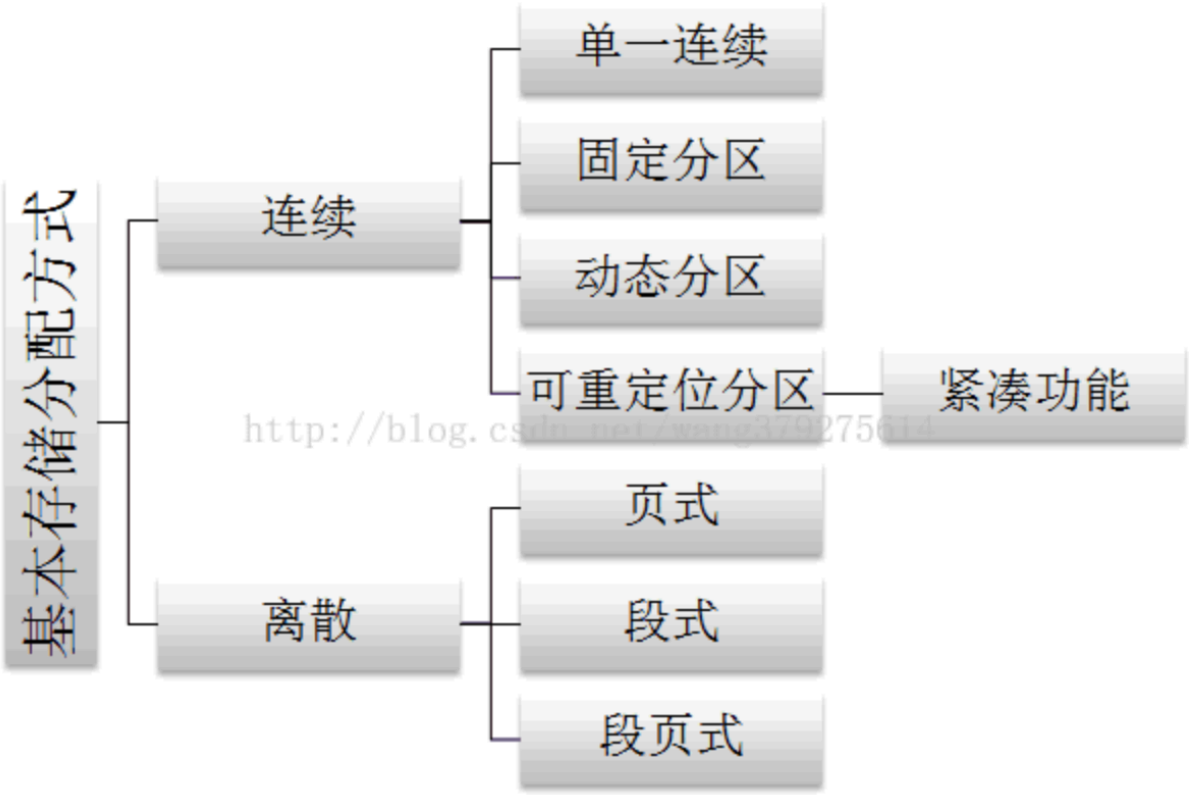
上图中有五个进程，四个资源。左边的图表示已经分配的资源，右边的图表示还需要分配的资源。最右边的 E、P 以及 A 分别表示：**总资源、已分配资源以及可用资源**，注意这三个为向量，而不是具体数值，例如 A=(1020)，表示 4 个资源分别还剩下 1/0/2/0。

检查一个状态是否安全的算法如下：

- 查找右边的矩阵是否存在一行小于等于向量 A。**如果不存在这样的行，那么系统将会发生死锁，状态是不安全的。**
- **假若找到这样一行，将该进程标记为终止，并将其已分配资源加到 A 中。**
- **重复以上两步，直到所有进程都标记为终止，则状态是安全的。**

如果一个状态不是安全的，需要拒绝进入这个状态。

四、内存管理



内部碎片

内部碎片是已经被分配出去的的内存空间，**但是没被进程装满**。例如，由于进程的最后一页经常装不满一块物理内存块而形成了**不可利用的碎片**，称之为“页内碎片”。

外部碎片

外部碎片是指还没有分配出去，**但是由于大小太小而无法分配给申请空间的新进程的内存空间空闲块。**

技术	简要说明	优点	缺点
固定分区	物理内存被分为大小固定的分区，进程可以装载到**大于等于**自身的分区	实现简单	1. 有**内部碎片** 2. 活动进程数目是固定的
动态分区	分区是被动态创建的，进程可以装载到正好**等于**自身大小的分区	没有内部碎片，内存使用更完全	有**外部碎片**，需要压缩外部碎片
简单分页	主存被分为很多大小相同的**页框**，进程被分为同样大小的很多**页**。要装入一个进程需要将**所有的页**装入，**可以装入不连续的页框**	没有外部碎片	有很少的内部锁片，在进程最后一页（页内碎片）
简单分段	进程被分为很多段，要装入一个进程，需要将进程装**所有段**入内存中**不一定连续的动态分区**	没有内部碎片，比较于动态分区，内存利用率更高，开销小	有外部碎片，需要压缩外部碎片
虚拟内存分页	与简单分页相比，不需要将所有页装入内存	1.没有外部碎片 2.巨大的虚拟内存空间 3.更高程度的多道程序设计	复杂的内存管理开销
虚拟内存分段	与简单分段相比，不需要将所有段装入内存	具有虚拟内存分页的三个优点，**并且支持保护和共享**	复杂的内存管理开销

程序直接使用「连续」物理内存地址

地址空间（可寻址的一片空间）

早期计算机，程序中访问的内存地址都是实际的物理内存地址，同时运行的程序内存总量必须小于实际物理内存

程序把物理内存地址写死在程序里了

存在的问题

问题1：进程地址空间不隔离。一个进程错误使用内存地址，可能造成多个进程崩溃。

问题2：内存利用率低。需要运行一个大程序，必须要腾出相应的**「连续」**内存空间，会导致频繁的换入换出（内存和外存）

问题3：程序运行地址不确定。当满足某个程序的运行内存要求后，**分配给其的地址不一定是其程序中写死的内存地址**。

虚拟内存

把一个进程的一部分调入内存中运行，当内存没有空闲空间时，将新的覆盖旧的页，同时将旧页写入磁盘。虚拟内存**主要使用分页存储管理模式**。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。

内存管理单元（MMU）管理着**程序虚拟地址空间和物理内存**的转换

分页存储管理

1.基本思想

用户程序的逻辑地址空间（虚拟地址）被划分成若干固定大小的区域，**称为"页"**，相应地，**内存空间分成若干个相等的物理块（页框）**。可将用户程序的任一页放在内存的任一块中，**实现了离散分配**。

2.等分物理内存

分页存储管理将内存空间划分成等长的若干物理块，称为物理页面也成为物理块，每个物理块的大小一般取2的整数幂。**内存的所有物理块从0开始编号，称作物理页号**。

3.程序逻辑地址

程序的各个**逻辑页面从0开始依次编号**，称作**逻辑页号或相对页号**。**每个页面内从0开始编址，称为页内地址**。程序中的**逻辑地址（虚拟地址）**由两部分组成：**页号P和页内位移量W**。

4.页表

系统为**每个进程建立一张页表**，用于记录**进程逻辑页面与内存物理页面之间的对应关系**。

页表的作用是实现**从页号到物理块号（页框号）的地址映射**，**程序逻辑地址空间有多少页，该页表里就登记多少行**，且按逻辑页的顺序排列。

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）。

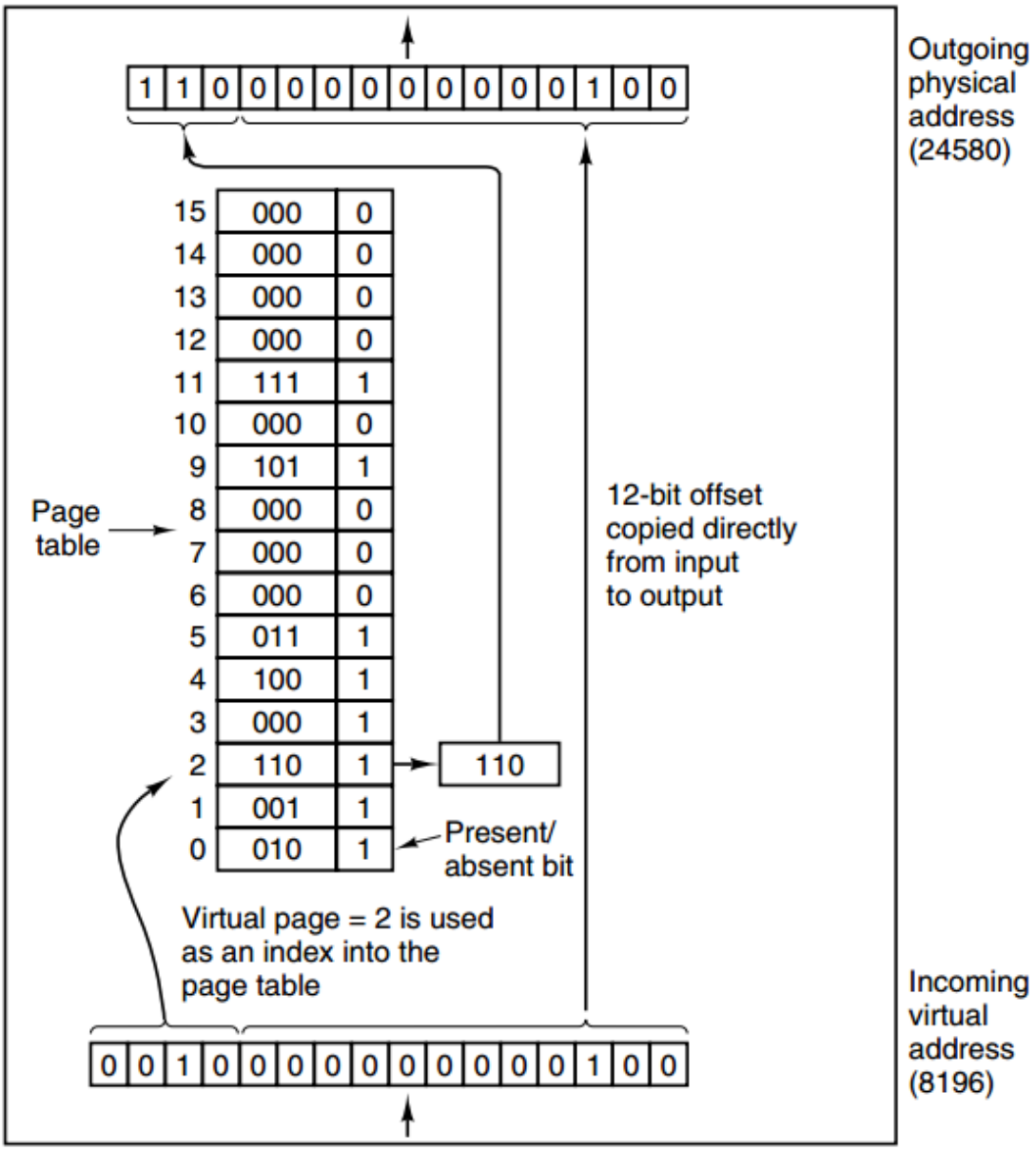


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

5.页面共享与保护

当**多个不同进程中需要有相同页面信息时**，可以在主存中只保留一个副本，只要让这些**进程各自的有关项中指向内存同一块号即可。同时在页表中设置相应的"存取权限"**，对不同进程的访问权限进行各种必要的限制。

6.页面置换

当进程在物理内存中运行时，调用到不在物理内存中的虚拟页面时，**MMU**注意到该页面没有被映射到物理内存，于是cpu陷入到操作系统，这个陷阱称为**缺页中断**。操作系统找到一个很少使用的页框且把他的内容写入磁盘备份。随后把需要访问的虚拟页面读到刚才回收的页框中，修改映射关系，然后重新启动引起陷阱的指令。

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）**。

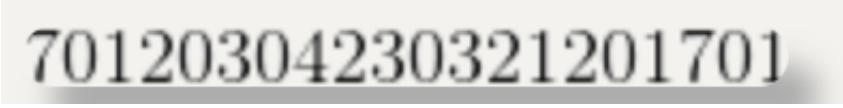
1. 最佳

OPT, Optimal replacement algorithm

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。

举例：一个系统为某进程**分配了三个物理块**，并有如下页面引用序列：



开始运行时，先将 7, 0, 1 三个页面装入内存。当进程要访问页面 2 时，产生缺页中断，会将页面 7 换出，因为页面 7 再次被访问的时间最长。

2. 最近最久未使用

LRU, Least Recently Used

虽然无法知道将来要使用的页面情况，但是可以知道过去使用页面的情况。LRU 将最近最久未使用的页面换出。

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。

因为每次访问都需要更新链表，因此这种方式实现的 LRU 代价很高。

3. 最近未使用

NRU, Not Recently Used

每个页面都有两个状态位：R 与 M，**当页面被访问时设置页面的 R=1，当页面被修改时设置 M=1。其中 R 位会定时被清零**。可以将页面分成以下四类：

- R=0，M=0
- R=0，M=1
- R=1，M=0
- R=1，M=1

当发生缺页中断时，NRU 算法随机**地从类编号最小的非空类中挑选一个页面将它换出**。

NRU 优先换出已经被修改的脏页面（R=0，M=1），而不是被频繁使用的干净页面（R=1，M=0）。

4. 先进先出

FIFO, First In First Out

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。

5. 第二次机会算法（FIFO的改进）

FIFO 算法可能会把经常使用的页面置换出去，为了避免这一问题，对该算法做一个简单的修改：

当页面被**访问 (读或写)** 时设置**该页面的 R 位为 1**。需要替换的时候，检查最老页面的 R 位。如果 R 位是 0，那么这个页面既老又没有被使用，可以立刻置换掉；如果是 1，就将 R 位清 0，**并把该页面放到链表的尾端，修改它的装入时间使它就像刚装入的一样，然后继续从链表的头部开始搜索。**

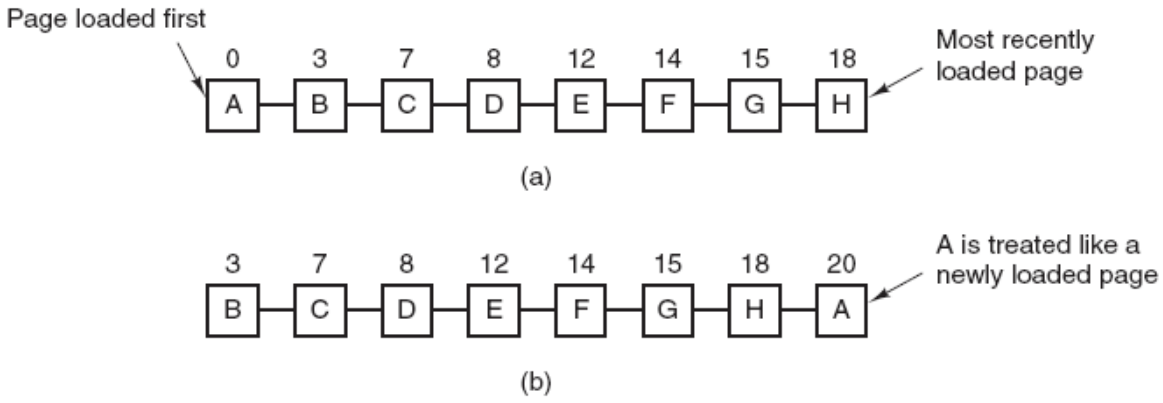


Figure 3-15. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set. The numbers above the pages are their load times.

6. 时钟（第二次机会算法的改进）

Clock

第二次机会算法需要在链表中移动页面，降低了效率。时钟算法使用环形链表将页面连接起来，再使用一个指针指向最老的页面。

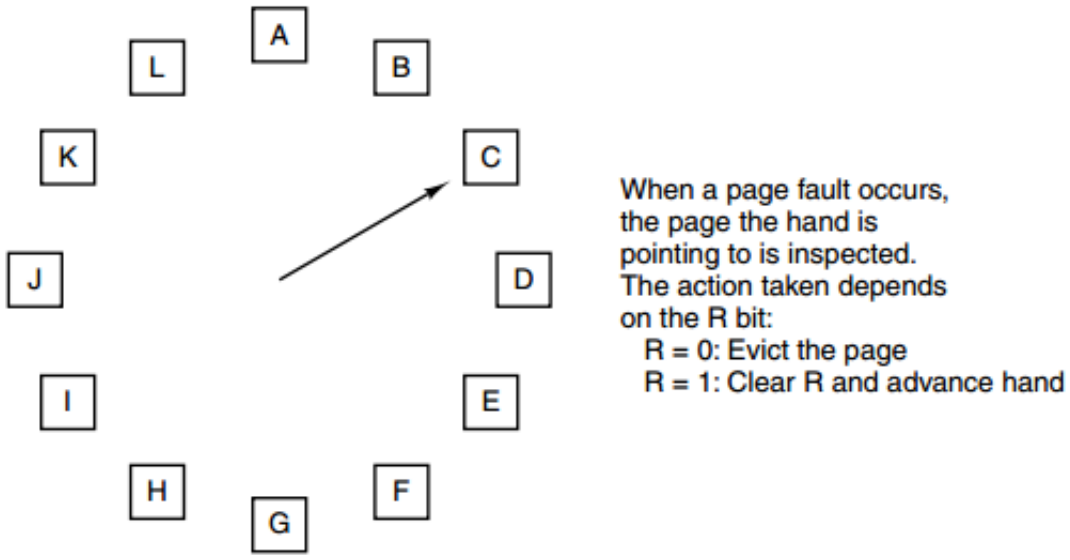


Figure 3-16. The clock page replacement algorithm.

分段存储管理

1.基本思想

页面是主存物理空间中划分出来的**等长的固定区域。分页方式的优点是页长固定，因而便于构造页表、易于管理，且不存在外碎片**。但分页方式的**缺点**是**页长与程序的逻辑大小不相关**。例如，某个时刻一个子程序可能有一部分在主存中，另一部分则在辅存中。这不利于编程时的独立性，并给换入换出处理、存储保护和存储共享等操作造成麻烦。

分段是**按照程序的自然分界划分的长度可以动态改变的区域**。通常，程序员把子程序、操作数和常数等不同类型的数据划分到不同的段中（写c程序时会用到），并且每个程序可以有多个相同类型的段。

段表本身也是一个段，可以存在辅存中，但一般是驻留在主存中。

将用户程序地址空间分成**若干个大小不等的段**，每段可以定义一组相对完整的逻辑信息。存储分配时，以段为单位，段与段在内存中可以不相邻接，也实现**了离散分配**。

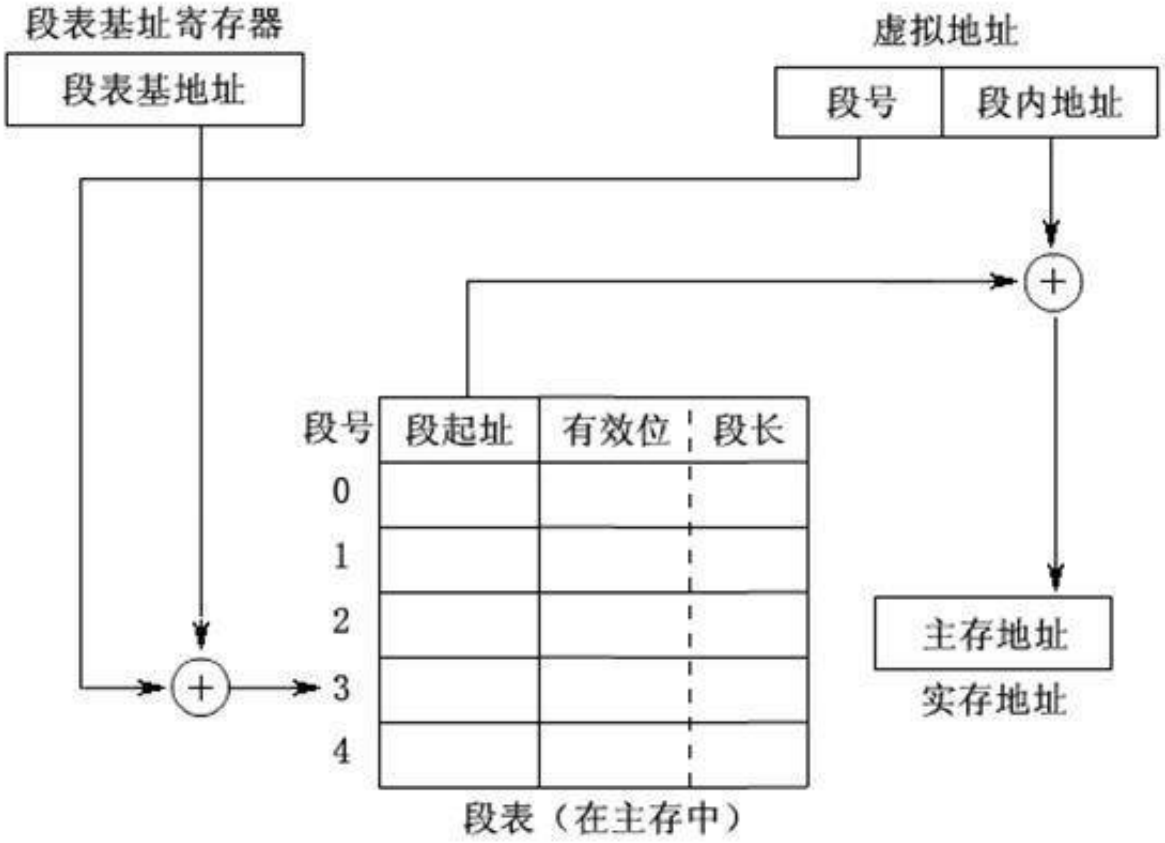
2.分段地址结构

进程的逻辑地址空间被划分为若干个段，每个段定义了一组逻辑信息。例程序段、数据段等。**每个段都从0开始编址，并采用一段「连续的」逻辑地址空间**。段的长度由相应的逻辑信息组的长度决定，因而**各段长度不等。整个进程的逻辑地址空间是「二维的」**（除了段的编号还有段长）。

在段式虚拟存储系统中，虚拟地址由**段号和段内地址**组成，虚拟地址到物理内存地址的变换通过**段表**来实现。每个进程设置一个段表，段表的每一个表项对应一个段，**每个表项至少包括三个字段**：**有效位**（指明该段是否已经调入主存）、**段起址**(该段在物理内存中的首地址)和**段长**（记录该段的实际长度）。

3.地址变换

针对每一个逻辑地址，存储管理部件首先**以段号S**为索引访问**段表的第S个表项**。若该表项的**有效位为1**，则将虚拟地址的**段内地址D与该表项的段长字段比较**；若段内地址较大则说明地址越界，将产生**地址越界中断**；否则，将该表项的**段起址与段内地址相加**，**求得主存实地址并访存**。如果该表项的有效位为0，则产生**缺页中断**，从辅存中调入该页，并修改段表。段式虚拟存储器虚实地址变换过程如图所示。



绝对物理内存地址=根据段号找到**段表中的起始地址+段内地址** (如果段内地址超过限长则产生“地址越界”程序性中断事件达到存储保护)

4.分段存储方式的优缺点

分页对程序员而言是不可见的，而分段通常对程序员而言是可见的，因而分段为组织程序和数据提供了方便。与页式虚拟存储器相比，**段式虚拟存储器有许多优点**：

(1) 段的逻辑独立性使其易于编译、管理、修改和保护，**也便于多道程序共享**。

(2) **段长可以根据需要动态改变，允许自由调度**，以便有效利用主存空间。

(3) **方便编程，分段共享，分段保护，动态链接，动态增长**

因为段的长度不固定，段式虚拟存储器也有一些缺点：

(1) 主存空间分配比较麻烦。

(2) **容易在段间留下许多碎片（外部碎片）**，造成存储空间利用率降低。

(3) 由于**段长不一定是2的整数次幂**，因而不能简单地像分页方式那样用虚拟地址和实存地址的最低若干二进制位作为段内地址，并与段号进行直接拼接，必须用**加法操作通过段起址与段内地址的求和运算得到物理地址**。因此，段式存储管理比页式存储管理方式**需要更多的硬件支持**。

段页式存储管理

在页式、段式存储管理中，为获得一条指令或数据，须两次访问内存；而段页式则须三次访问内存

1.段页式存储管理的基本思想

段页式存储组织是分段式和分页式结合的存储组织方法，这样可充分利用分段管理和分页管理的优点。

(1) 用**分段方法来分配和管理程序可见的「虚拟内存」**。程序的地址空间**按逻辑单位分成基本独立的段**，而每一段有自己的段名，**再把每段分成固定大小的若干页**。

(2) **用分页方法来分配和管理实际「物理内存」**。即把整个主存分成与上述页大小相等的存储块，可装入进程的任何一页。**程序对内存的调入或调出是按页进行的。但它又可按段实现共享和保护**。

2.段页式存储管理的优缺点

优点：

(1) 它提供了大量的虚拟存储空间。

(2) 能有效地利用主存，为组织多道程序运行提供了方便。

缺点：

(1) 增加了硬件成本、系统的复杂性和管理上的开消。

(2) 存在着系统发生抖动的危险。

(3) 存在着**内碎片**。

(4) 还有各种表格要占用主存空间。

段页式存储管理技术对当前的大、中型计算机系统来说，算是最通用、最灵活的一种方案。

分页与分段比较

- 对程序员的透明性：**分页透明**，但是分段需要程序员显示划分每个段。
- 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**。

后面这一段是我没太明白的一段

程序通过虚拟地址使用「连续」物理内存

通过添加一个虚拟地址，**让程序不再写死物理内存地址，操作系统负责完成 虚拟地址 -> 物理内存地址映射**

分段（可以解决问题1、3）

操作系统处理好虚拟地址到物理内存地址的映射，就可以保证进程地址空间之间的「隔离」

它的思想是在「连续」虚拟地址空间和「连续」物理地址空间之间做一一映射。比如说虚拟地址空间中某个 10M 大小的空间映射到物理地址空间中某个 10M 大小的空间。

无法解决问题2：因为问题2的根本原因是程序**需要「完整连续」的内存地址**，导致了频繁的换入换出

分段技术下，**每次换入换出的本质上还是一整个进程**，于是，根据程序的**局部性原理**，设计了一个**更细粒度**的内存分割和映射的方法，**分页**

分页

仍然是一种虚拟地址空间到物理地址空间映射的机制。但是，粒度更加的小了。**单位不是整个程序，而是某个“页”**。

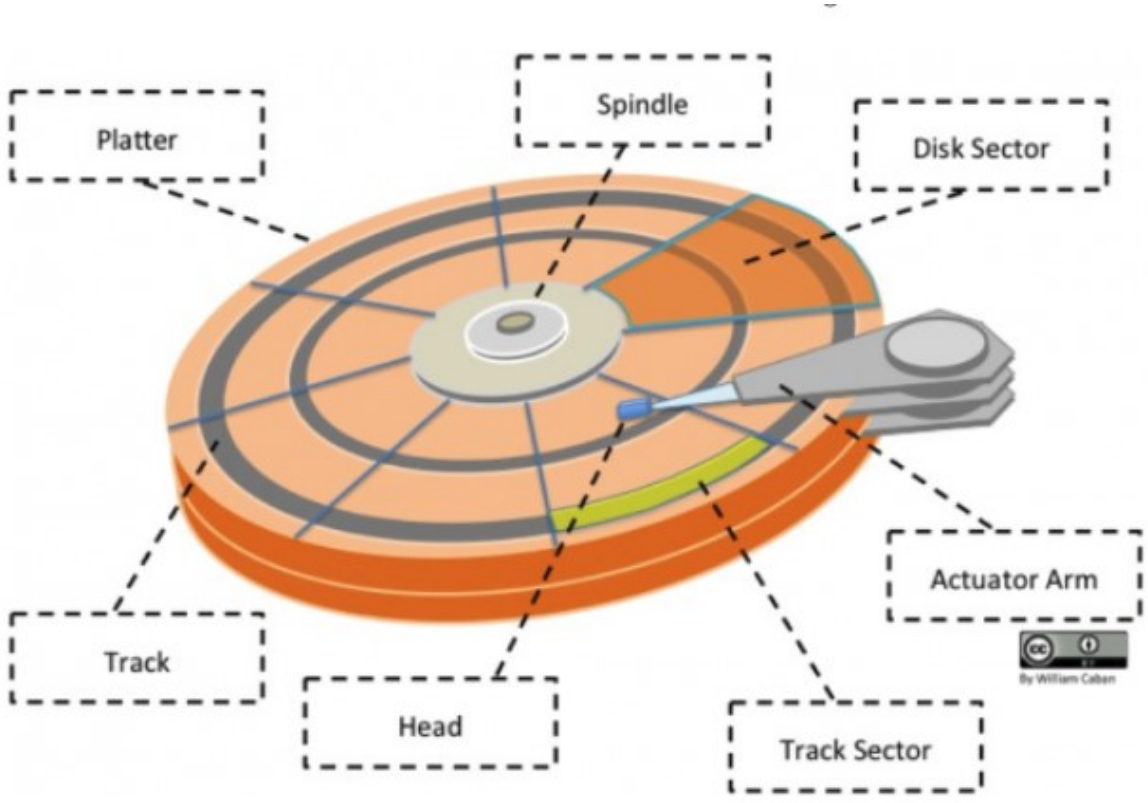
分页它的**虚拟地址空间仍然是「连续的」**，但是，**每一页映射后的物理地址就「不一定是连续的」**。

分页的思想是程序**运行时用到哪页就为哪页分配内存，没用到的页暂时保留在硬盘上**。并且把换入换出的**单位缩小到了一页**，而不是换入换出整个程序了，这里也体现了局部性原理。

五、设备管理

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。



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**。

1. 先来先服务

> FCFS, First Come First Served

按照磁盘请求的顺序进行调度。

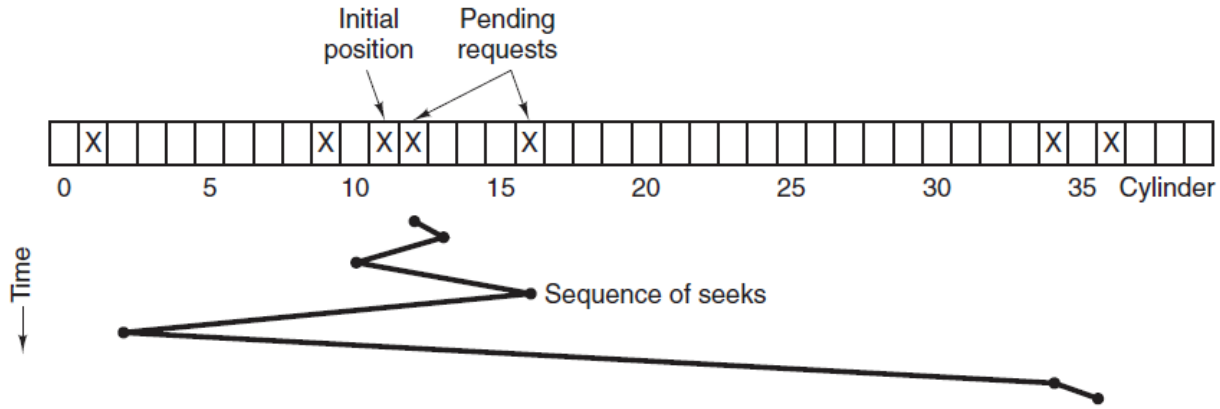
优点是公平和简单。缺点也很明显，因为未对寻道做任何优化，使平均寻道时间可能较长。

2. 最短寻道时间优先

> SSTF, Shortest Seek Time First

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**。

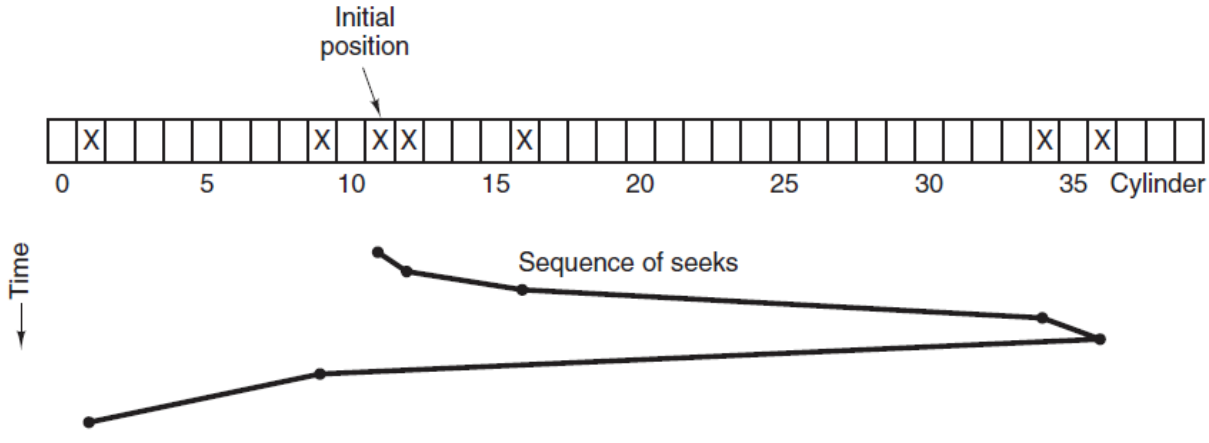


3. 电梯算法

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。



六、链接

编译系统

以下是一个 hello.c 程序：

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

在 Unix 系统上，由**编译器把源文件转换为目标文件**。

```
gcc -o hello hello.c
```

这个过程大致如下：

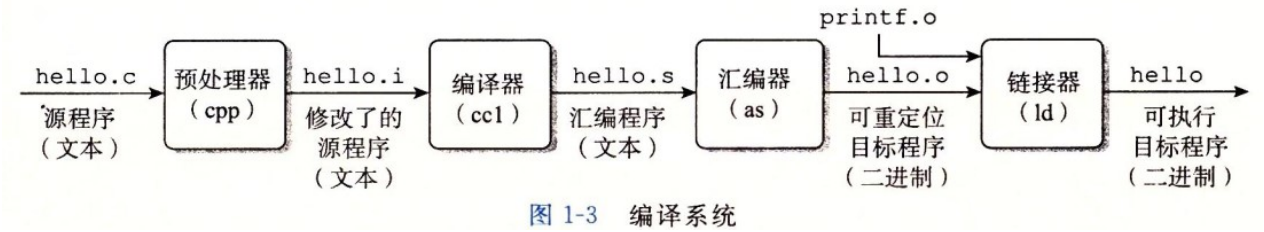


图 1-3　编译系统

- **预处理阶段**：处理以 # 开头的预处理命令；
- **编译阶段**：翻译成汇编文件；
- **汇编阶段**：将汇编文件翻译成可重定向目标文件；
- **链接阶段**：将可重定向目标文件和 printf.o 等**单独预编译好的目标文件进行合并**，得到最终的可执行目标文件。

静态链接

静态链接器以**一组可重定向目标文件为输入**，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。

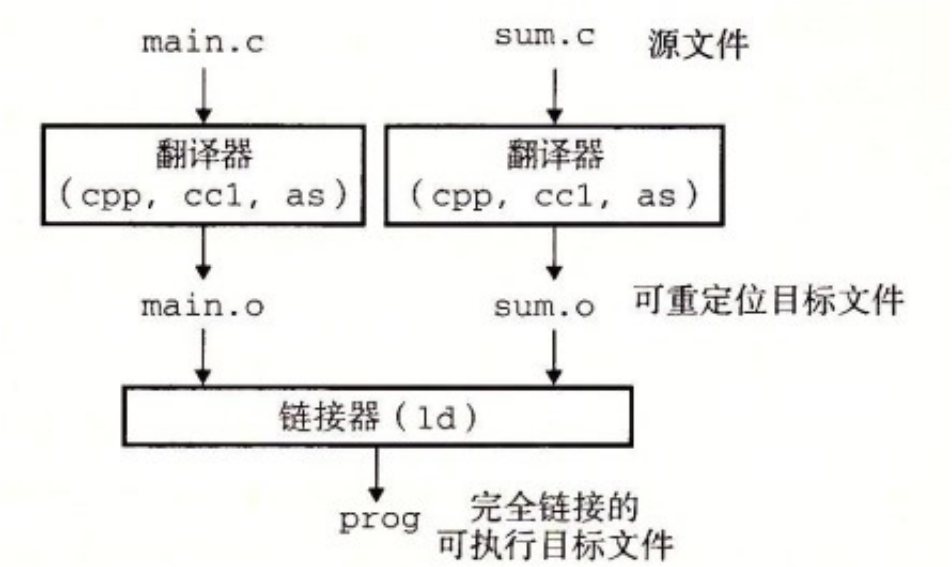


图 7-2 静态链接。链接器将可重定位目标文件组合起来，形成一个可执行目标文件 prog

目标文件

- 可执行目标文件：可以直接在内存中执行；
- 可重定向目标文件：可与其它可重定向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向目标文件，可以在运行时被动态加载进内存并链接；

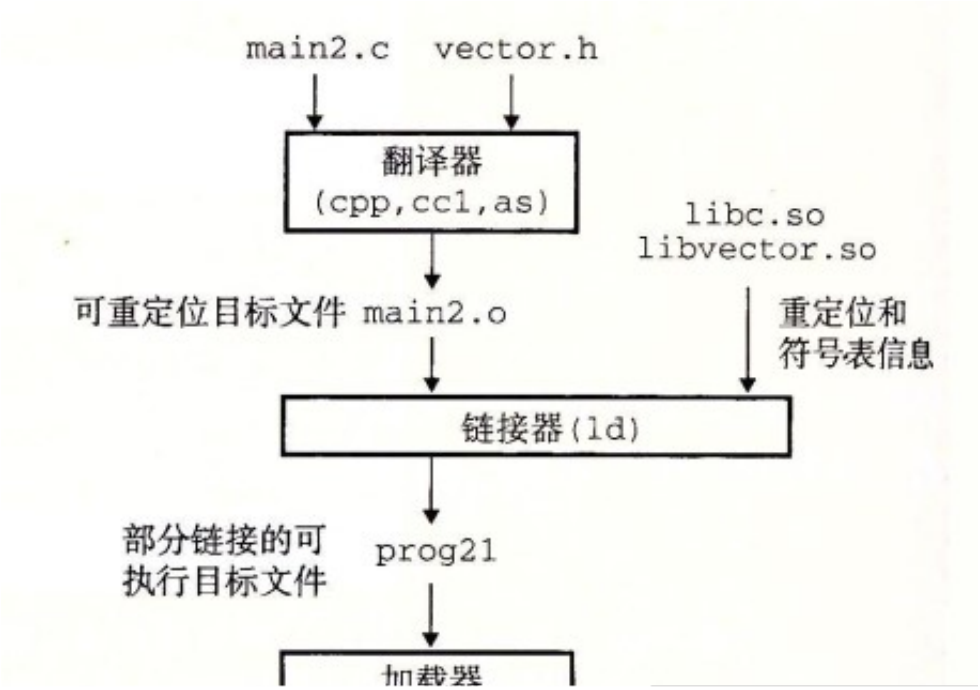
动态链接

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。


main2.c
vector.h
翻译器
(cpp,cc1,as)
libc.so
libvector.so
可重定位目标文件 main2.o
重定位和
符号表信息
链接器(ld)
部分链接的可
执行目标文件
prog21
加载器