

# 计算机操作系统

## 一、概述

### 操作系统基本特征

#### 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，如多流水线或者多处理器。

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。例如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，每次只执行一小段时间片并快速切换。

## 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。

# 操作系统基本功能

## 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。

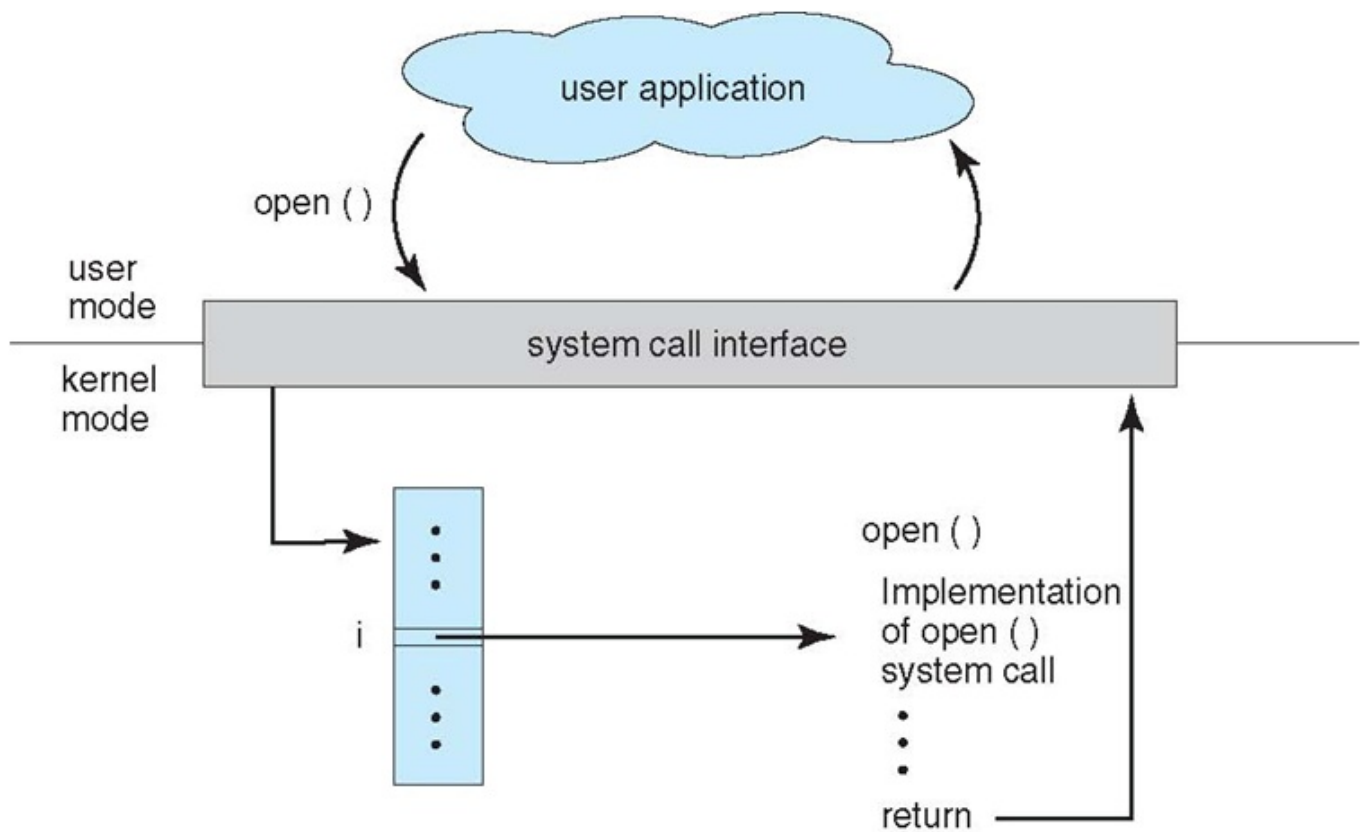
## 4. 设备管理

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。



## 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。

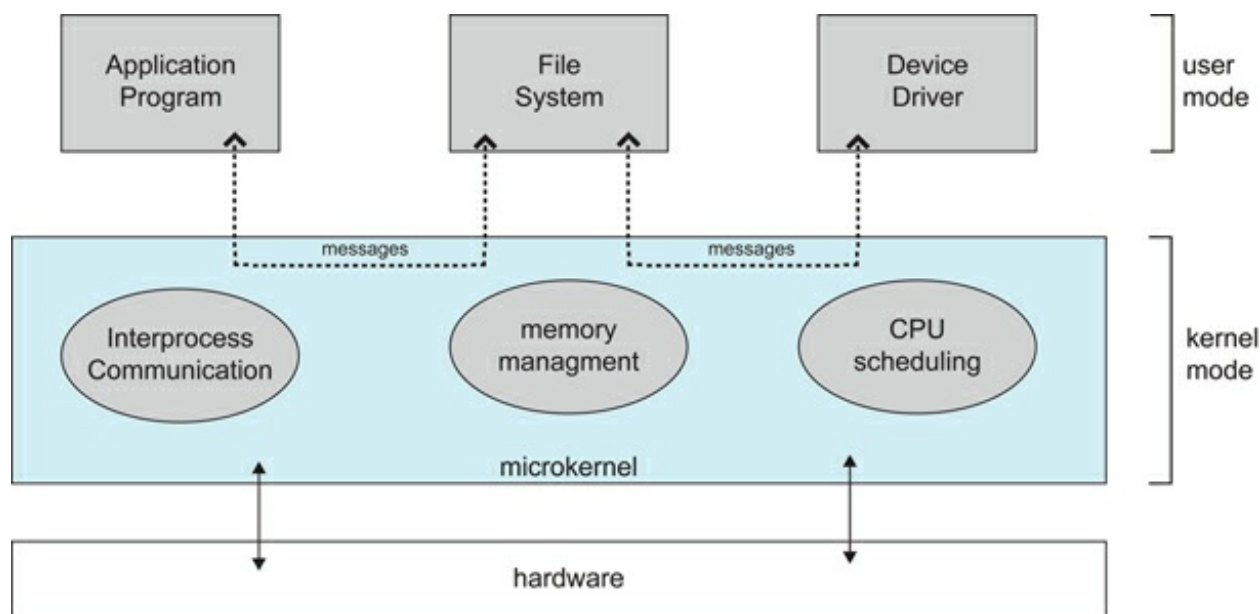
由于各模块共享信息，因此有很高的性能。

### 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

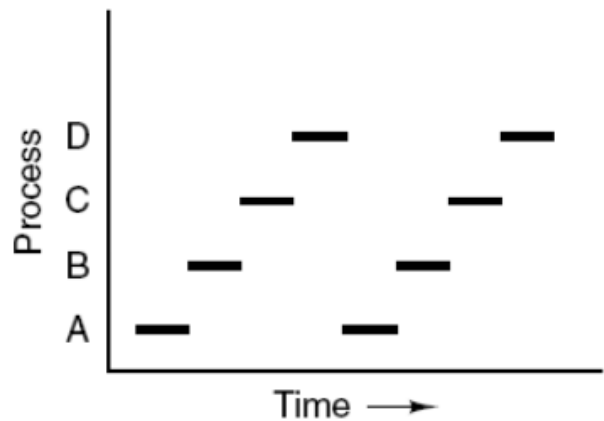
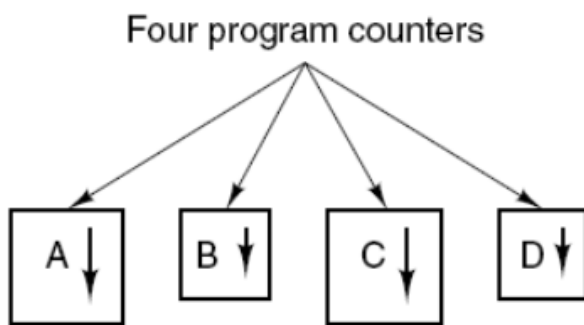
# 进程与线程

## 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。

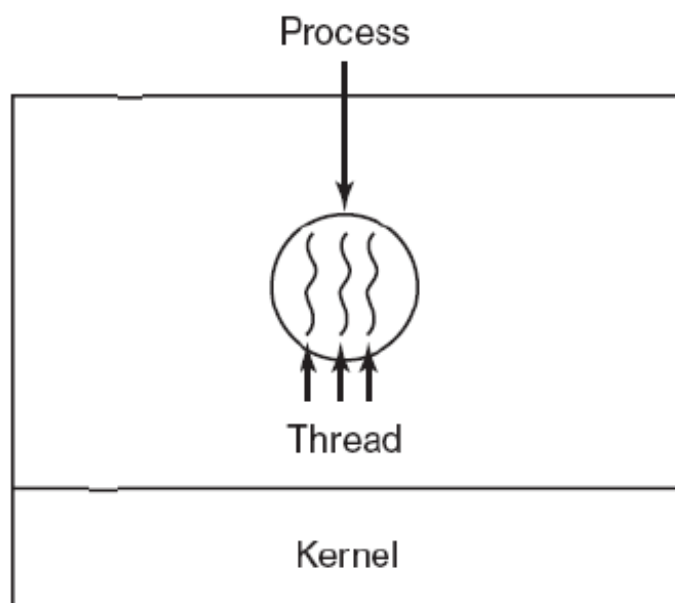
下图显示了 4 个程序创建了 4 个进程，这 4 个进程可以并发地执行。



## 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。



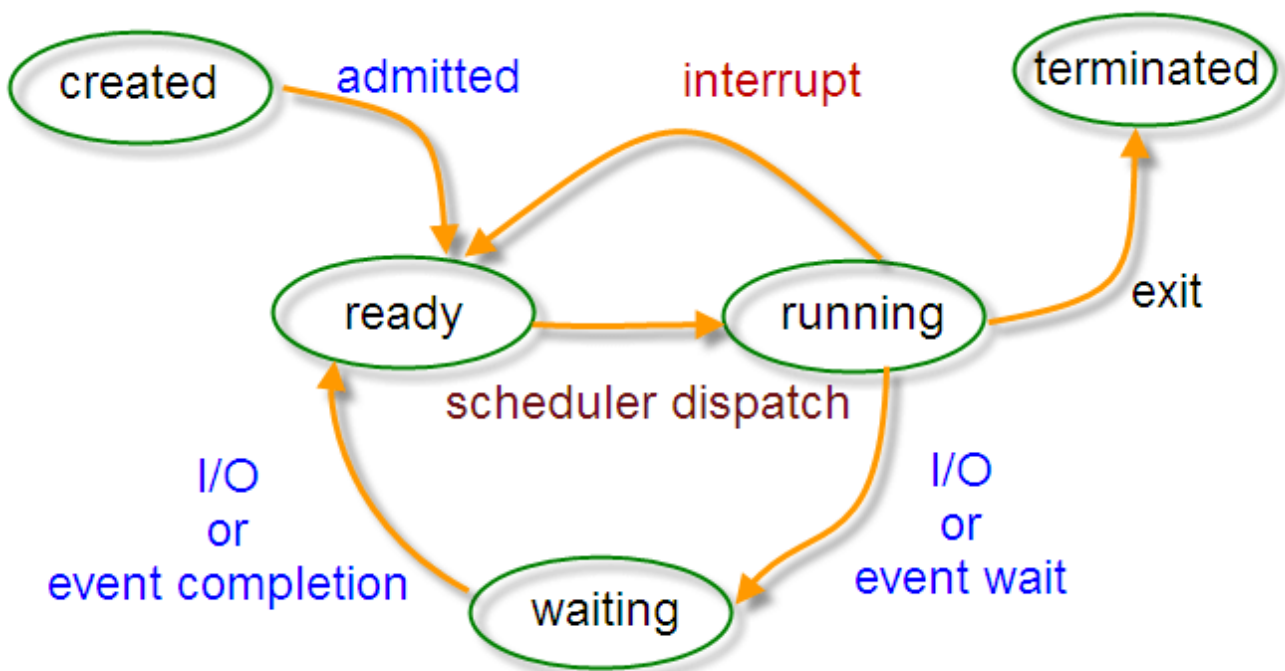
### 3. 区别

- 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。
- 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，从一个进程内的线程切换到另一个进程中的线程时，会引起进程切换。
- 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。
- 通信方面：进程间通信 (IPC) 需要进程同步和互斥手段的辅助，以保证数据的一致性。而线程间可以通过直接读/写同一进程中的数据段（如全局变量）来进行通信。

举例：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。

## 进程状态的切换

# Process State



## 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

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 first-come first-serverd ( 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 shortest job first ( 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。

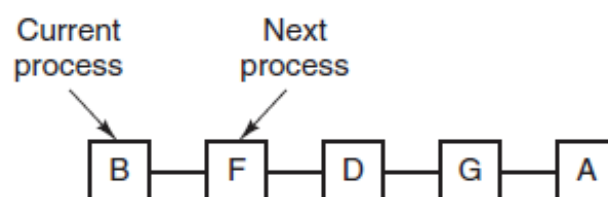
# 2. 交互式系统

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。

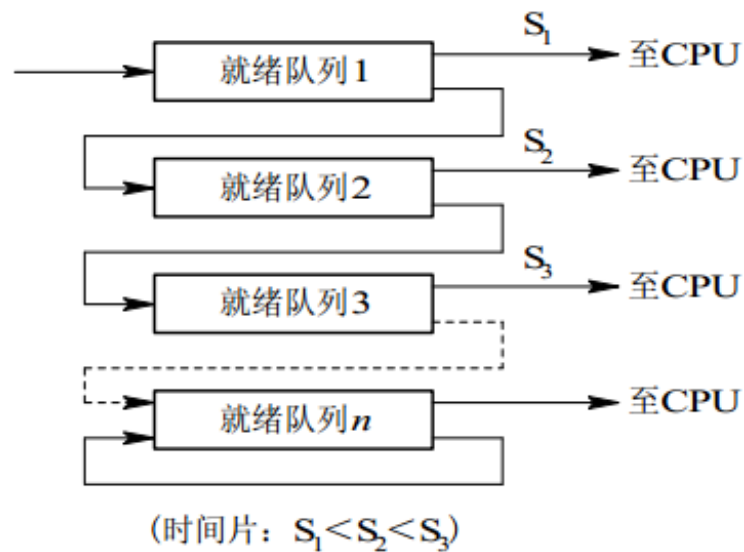
## 2.1 时间片轮转

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。







### 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。

## 进程调度算法实现

以下只是假象系统上的调度算法实现。源代码：[Scheduling](#)

### 1. FCFS

首先创建进程数据结构：

```
1.  public class Process {
2.      private String name;
3.      private long totalTime;
4.      private long remainTime;
5.      private long comeInTime;
6.
7.      public Process(String name, long totalTime, long comeInTime) {
8.          this.name = name;
9.          this.totalTime = totalTime;
```

```

10.         this.remainTime = totalTime;
11.         this.comeInTime = comeInTime;
12.     }
13.
14.     public void run(long runTime) {
15.         System.out.println("process " + name + " is running...");
16.         System.out.println("come in time : " + comeInTime);
17.         System.out.println("total time : " + totalTime);
18.         System.out.println("remain time : " + remainTime);
19.         System.out.println();
20.         remainTime -= runTime;
21.         try {
22.             Thread.sleep(runTime);
23.         } catch (InterruptedException e) {
24.             e.printStackTrace();
25.         }
26.     }
27.
28.     public long getTotalTime() {
29.         return totalTime;
30.     }
31.
32.     public long getRemainTime() {
33.         return remainTime;
34.     }
35.
36.     public long getComeInTime() {
37.         return comeInTime;
38.     }
39. }

```

接着创建一个进程等待队列数据结构：

```

1.     public interface ProcessQueue {
2.
3.         void add(Process process);
4.
5.         Process get();
6.
7.         boolean isEmpty();
8.     }

```

FCFS 算法使用普通的先进先出队列即可实现该进程等待队列：

```

1.  public class FCFSProcessQueue implements ProcessQueue {
2.
3.      private Queue<Process> queue = new LinkedList<>();
4.
5.      @Override
6.      public void add(Process process) {
7.          queue.add(process);
8.      }
9.
10.     @Override
11.     public Process get() {
12.         return queue.poll();
13.     }
14.
15.     @Override
16.     public boolean isEmpty() {
17.         return queue.isEmpty();
18.     }
19. }

```

接下来是调度器的实现，把它设计成可独立运行的 Thread，它对进程等待队列进行轮询，如果有进程的话就从队列中取出一个任务来执行。批处理系统中，调度器调度一个进程都直接让进程执行完毕，也就是给进程运行时间为进程的总时间。

```

1.  public class Scheduler extends Thread {
2.
3.      protected ProcessQueue processQueue;
4.
5.      public Scheduler(ProcessQueue processQueue) {
6.          this.processQueue = processQueue;
7.      }
8.  }

```

```

1.  public class BatchScheduler extends Scheduler {
2.      public BatchScheduler(ProcessQueue processQueue) {
3.          super(processQueue);
4.      }
5.
6.      @Override
7.      public void run() {
8.          while (true) {
9.              if (!processQueue.isEmpty()) {

```

```

10.         Process process = processQueue.get();
11.         process.run(process.getTotalTime());
12.     }
13.     try {
14.         Thread.sleep(10);
15.     } catch (InterruptedException e) {
16.         e.printStackTrace();
17.     }
18. }
19. }
20. }

```

使用一个模拟器来模拟实际场景下进程的不定时到达，每 1s 到达一个进程，并且进程的运行时间再 0s ~ 3s 之间，使得多个进程会发生竞争关系。

```

1.  public class ProcessComeEmulator extends Thread {
2.
3.      private ProcessQueue processQueue;
4.
5.      public ProcessComeEmulator(ProcessQueue processQueue) {
6.          this.processQueue = processQueue;
7.      }
8.
9.      @Override
10.     public void run() {
11.         int processId = 0;
12.         while (true) {
13.             System.out.println("process " + processId + " is coming..."
14. );
15.             System.out.println();
16.             Process process = new Process((processId++) + "", getRandom
17. Time(), System.currentTimeMillis());
18.             processQueue.add(process);
19.             try {
20.                 Thread.sleep(1000);
21.             } catch (InterruptedException e) {
22.                 e.printStackTrace();
23.             }
24.         }
25.     }
26.
27.     private long getRandomTime() {
28.         return (long) (Math.random() * 3000);
29.     }
30. }

```

```
28.     }
```

最后创建一个客户端来运行整个调度程序：

```
1.  public class FCFSClient {
2.      public static void main(String[] args) {
3.          ProcessQueue processQueue = new FCFSProcessQueue();
4.          ProcessComeEmulator processComeEmulator = new
ProcessComeEmulator(processQueue);
5.          processComeEmulator.start();
6.          Scheduler scheduler = new FCFSScheduler(processQueue);
7.          scheduler.start();
8.      }
9.  }
```

执行结果如下：

```
1.  process 0 is coming...
2.
3.  process 0 is running...
4.  come in time   : 1528964425691
5.  total time    : 1807
6.  remain time   : 1807
7.
8.  process 1 is coming...
9.
10. process 1 is running...
11. come in time   : 1528964426691
12. total time    : 2311
13. remain time   : 2311
14.
15. process 2 is coming...
16.
17. process 3 is coming...
18.
19. process 4 is coming...
20.
21. process 2 is running...
22. come in time   : 1528964427692
23. total time    : 2651
24. remain time   : 2651
25.
26. process 5 is coming...
```

```
27.
28. process 6 is coming...
29.
30. process 3 is running...
31. come in time : 1528964428692
32. total time : 137
33. remain time : 137
34.
35. process 4 is running...
36. come in time : 1528964429692
37. total time : 2513
38. remain time : 2513
```

## 2. SJF

与 FCFS 不同的是，SJF 的进程等待队列需要使用优先级队列来实现：

```
1. public class SJFProcessQueue implements ProcessQueue {
2.
3.     private PriorityQueue<Process> processesQueue = new PriorityQueue<>
4.     (
5.         (o1, o2) -> (int) (o1.getTotalTime() - o2.getTotalTime()));
6.
7.     @Override
8.     public void add(Process process) {
9.         processesQueue.add(process);
10.    }
11.
12.    @Override
13.    public Process get() {
14.        return processesQueue.poll();
15.    }
16.
17.    @Override
18.    public boolean isEmpty() {
19.        return processesQueue.isEmpty();
20.    }
21. }
```

运行客户端：

```
1. public class SJFClient {
```

```
2.     public static void main(String[] args) {
3.         ProcessQueue processQueue = new SJFProcessQueue();
4.         ProcessComeEmulator processComeEmulator = new
ProcessComeEmulator(processQueue);
5.         processComeEmulator.start();
6.         Scheduler scheduler = new BatchScheduler(processQueue);
7.         scheduler.start();
8.     }
9. }
```

```
1.  process 0 is coming...
2.
3.  process 0 is running...
4.  come in time   : 1528964250005
5.  total time    : 2496
6.  remain time   : 2496
7.
8.  process 1 is coming...
9.
10. process 2 is coming...
11.
12. process 1 is running...
13. come in time   : 1528964251006
14. total time    : 903
15. remain time   : 903
16.
17. process 3 is coming...
18.
19. process 2 is running...
20. come in time   : 1528964252006
21. total time    : 1641
22. remain time   : 1641
23.
24. process 4 is coming...
25.
26. process 5 is coming...
27.
28. process 4 is running...
29. come in time   : 1528964254007
30. total time    : 243
31. remain time   : 243
32.
33. process 5 is running...
34. come in time   : 1528964255007
```

```
35.    total time      : 646
36.    remain time    : 646
37.
38.    process 3 is running...
39.    come in time    : 1528964253006
40.    total time      : 2772
41.    remain time     : 2772
```

## 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。

```
1.    // entry section
2.    // critical section;
3.    // 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。

```
1.  typedef int semaphore;
2.  semaphore mutex = 1;
3.  void P1() {
4.      down(&mutex);
5.      // 临界区
6.      up(&mutex);
7.  }
8.
9.  void P2() {
10.     down(&mutex);
11.     // 临界区
12.     up(&mutex);
13. }
```

## 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，也就无法执行 up(empty) 操作，empty 永远都为 0，那么生产者和消费者就会一直等待下去，造成死锁。

```
1.  #define N 100
2.  typedef int semaphore;
3.  semaphore mutex = 1;
4.  semaphore empty = N;
```

```

5.  semaphore full = 0;
6.
7.  void producer() {
8.      while(TRUE) {
9.          int item = produce_item();
10.         down(&empty);
11.         down(&mutex);
12.         insert_item(item);
13.         up(&mutex);
14.         up(&full);
15.     }
16. }
17.
18. void consumer() {
19.     while(TRUE) {
20.         down(&full);
21.         down(&mutex);
22.         int item = remove_item();
23.         up(&mutex);
24.         up(&empty);
25.         consume_item(item);
26.     }
27. }

```

## 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。

```

1.  monitor ProducerConsumer
2.      integer i;
3.      condition c;
4.
5.      procedure insert();
6.      begin
7.          // ...
8.      end;
9.
10.     procedure remove();

```

```
11.     begin
12.         // ...
13.     end;
14. 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。

## 使用管程实现生成者-消费者问题

```
1.  // 管程
2.  monitor ProducerConsumer
3.      condition full, empty;
4.      integer count := 0;
5.      condition c;
6.
7.      procedure insert(item: integer);
8.      begin
9.          if count = N then wait(full);
10.         insert_item(item);
11.         count := count + 1;
12.         if count = 1 then signal(empty);
13.     end;
14.
15.     function remove: integer;
16.     begin
17.         if count = 0 then wait(empty);
18.         remove = remove_item;
19.         count := count - 1;
20.         if count = N - 1 then signal(full);
21.     end;
22. end monitor;
23.
24. // 生产者客户端
25. procedure producer
26. begin
27.     while true do
28.         begin
```

```

29.         item = produce_item;
30.         ProducerConsumer.insert(item);
31.     end
32. end;
33.
34. // 消费者客户端
35. procedure consumer
36. begin
37.     while true do
38.     begin
39.         item = ProducerConsumer.remove;
40.         consume_item(item);
41.     end
42. end;

```

## 经典同步问题

生产者和消费者问题前面已经讨论过了。

### 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。

```

1.  typedef int semaphore;
2.  semaphore count_mutex = 1;
3.  semaphore data_mutex = 1;
4.  int count = 0;
5.
6.  void reader() {
7.      while(TRUE) {
8.          down(&count_mutex);
9.          count++;
10.         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
11.         up(&count_mutex);
12.         read();
13.         down(&count_mutex);

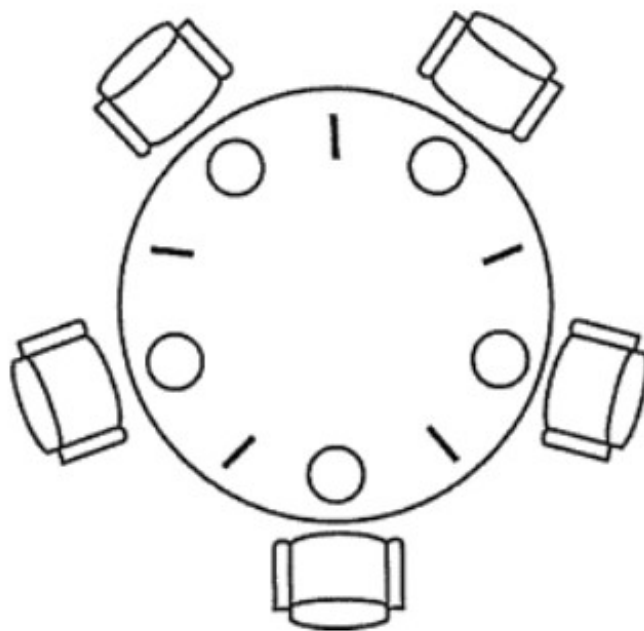
```

```

14.         count--;
15.         if(count == 0) up(&data_mutex);
16.         up(&count_mutex);
17.     }
18. }
19.
20. void writer() {
21.     while(TRUE) {
22.         down(&data_mutex);
23.         write();
24.         up(&data_mutex);
25.     }
26. }

```

## 2. 哲学家进餐问题



```

1.  #define N 5
2.
3.  void philosopher(int i) {
4.      while(TRUE) {
5.          think();
6.          take(i);          // 拿起左边的筷子
7.          take((i+1)%N);    // 拿起右边的筷子
8.          eat();
9.          put(i);
10.         put((i+1)%N);

```

```
11.     }
12. }
```

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。

```
1.  #define N 5
2.  #define LEFT (i + N - 1) % N // 左邻居
3.  #define RIGHT (i + 1) % N    // 右邻居
4.  #define THINKING 0
5.  #define HUNGRY 1
6.  #define EATING 2
7.  typedef int semaphore;
8.  int state[N];                // 跟踪每个哲学家的状态
9.  semaphore mutex = 1;         // 临界区的互斥
10. semaphore s[N];              // 每个哲学家一个信号量
11.
12. void philosopher(int i) {
13.     while(TRUE) {
14.         think();
15.         take_two(i);
16.         eat();
17.         put_tow(i);
18.     }
19. }
20.
21. void take_two(int i) {
22.     down(&mutex);
23.     state[i] = HUNGRY;
24.     test(i);
25.     up(&mutex);
26.     down(&s[i]);
27. }
28.
29. void put_tow(i) {
30.     down(&mutex);
31.     state[i] = THINKING;
32.     test(LEFT);
33.     test(RIGHT);
34.     up(&mutex);
35. }
36.
```

```
37. void test(i) { // 尝试拿起两把筷子
38.     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=EATING) {
39.         state[i] = EATING;
40.         up(&s[i]);
41.     }
42. }
```

## 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。

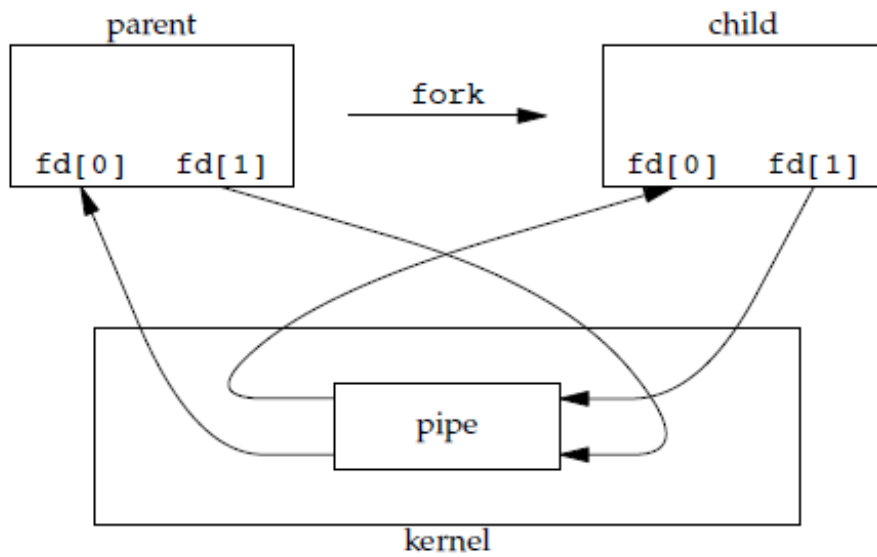
### 1. 管道

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

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

它具有以下限制：

- 只支持半双工通信（单向传输）；
- 只能在父子进程中使用。



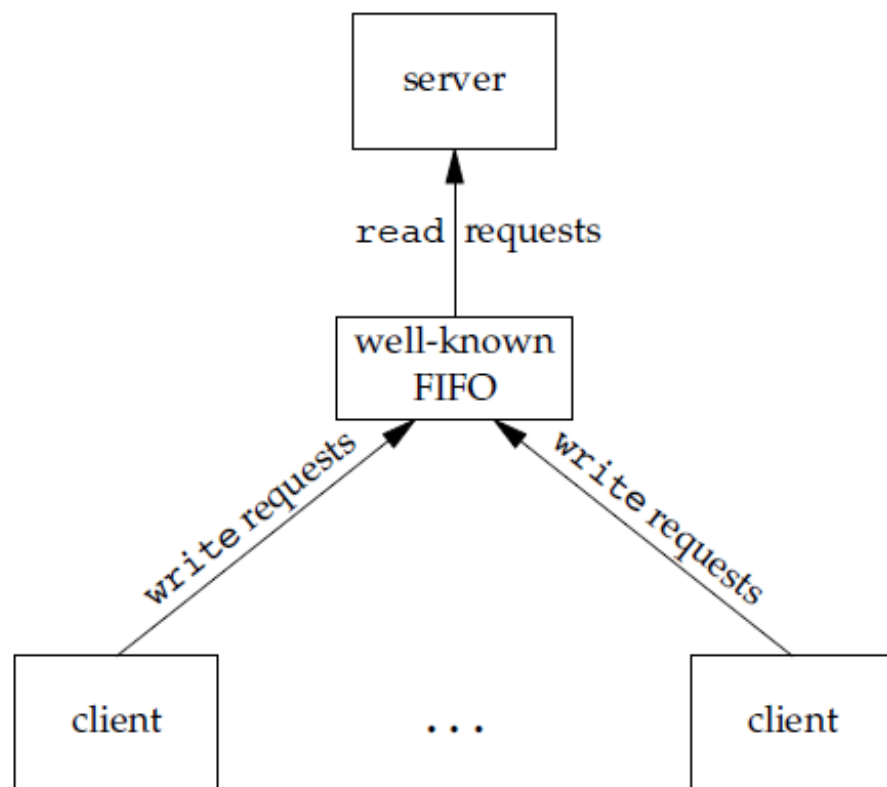
## 2. FIFO

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。

```
1. #include <sys/stat.h>
2. int mkfifo(const char *path, mode_t mode);
3. 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

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。

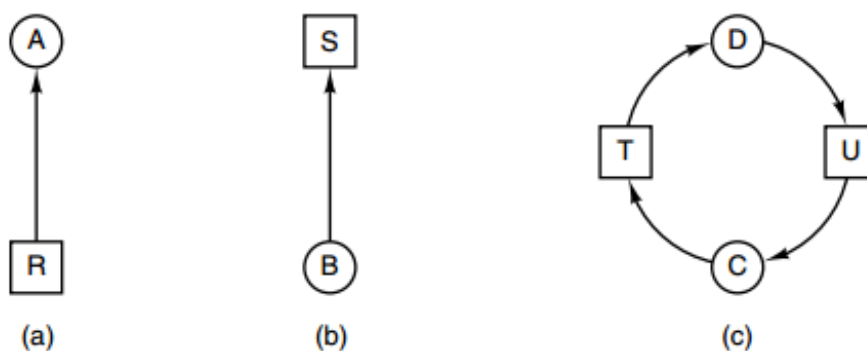
多个进程可以将同一个文件映射到它们的地址空间从而实现共享内存。另外 XSI 共享内存不是使用文件，而是使用使用内存的匿名段。

## 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、死锁

## 死锁的必要条件



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

## 死锁的处理方法

### 1. 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。

## 2. 死锁检测与死锁恢复

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。

(一) 每种类型一个资源的死锁检测

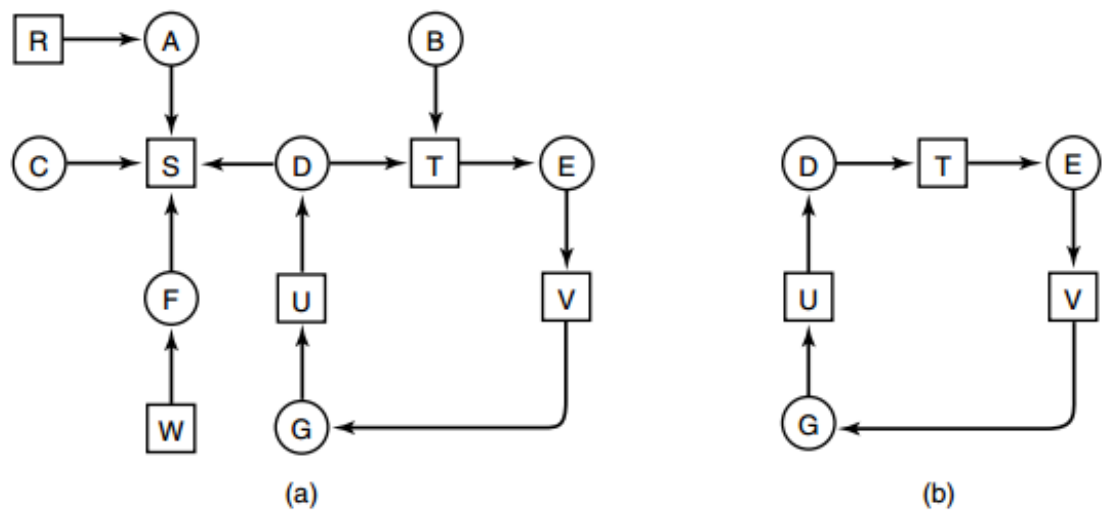


Figure 6-5. (a) A resource graph. (b) A cycle extracted from (a).

	Tape drives	Plotters	Scanners	Blu-rays
E =	( 4	2	3	1 )
A =	( 2	1	0	0 )
Current allocation matrix				
C =	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$			
Request matrix				
R =	$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$			

### 3. 死锁预防

在程序运行之前预防发生死锁。

#### (一) 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。

#### (二) 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。

#### (三) 破坏不可抢占条件

#### (四) 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。

### 4. 死锁避免

在程序运行时避免发生死锁。

#### (一) 安全状态

Has Max			Has Max			Has Max			Has Max			Has Max		
A	3	9	A	3	9	A	3	9	A	3	9	A	3	9
B	2	4	B	4	4	B	0	—	B	0	—	B	0	—
C	2	7	C	2	7	C	2	7	C	7	7	C	0	—
Free: 3			Free: 1			Free: 5			Free: 0			Free: 7		
(a)			(b)			(c)			(d)			(e)		

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

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7
Free: 10		
(a)		

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7
Free: 2		
(b)		

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7
Free: 1		
(c)		

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

	Process	Tape drives	Plotters	Printers	Blu-rays
A	3	0	1	1	
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	
Resources assigned					

	Process	Tape drives	Plotters	Printers	Blu-rays
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	
Resources still assigned					

$E = (6342)$   
 $P = (5322)$   
 $A = (1020)$

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

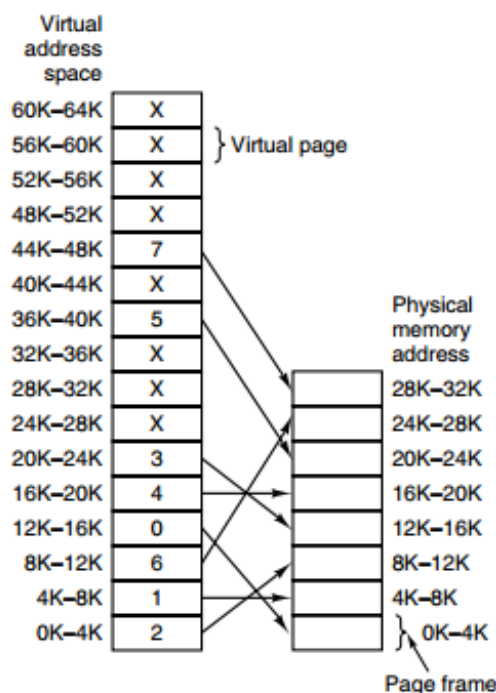
## 四、内存管理

### 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。当程序引用到一部分不在物理内存中的地址空间时，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称为可能。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。



**Figure 3-9.** The relation between virtual addresses and physical memory addresses is given by the **page table**. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K-8K really means 4096-8191 and 8K to 12K means 8192-12287.

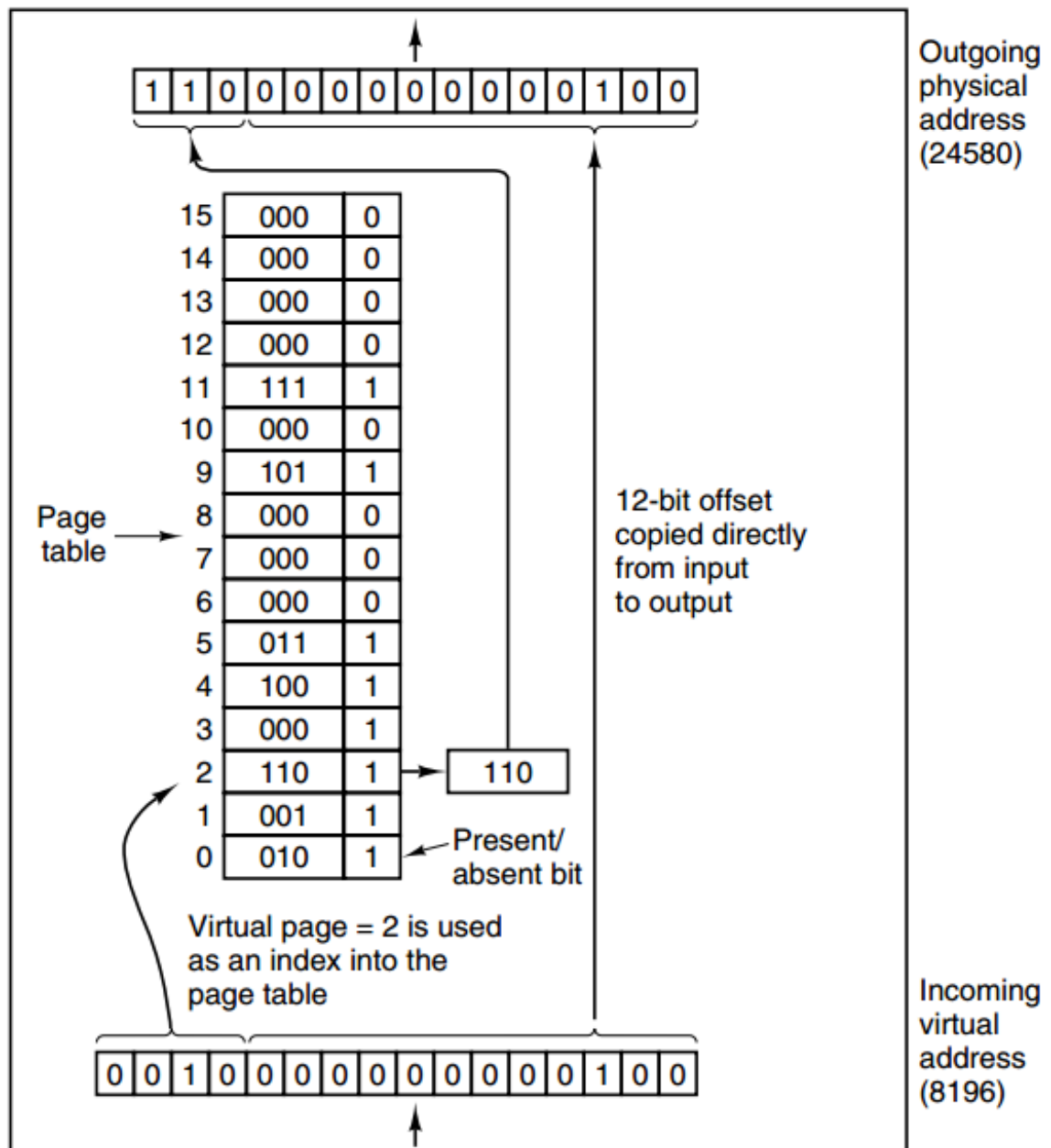
## 分页系统地址映射

- 内存管理单元（MMU）：管理着地址空间和物理内存的转换。
- 页表（Page table）：页（地址空间）和页框（物理内存空间）的映射表。例如下图中，页表的第 0 个表项为 010，表示第 0 个页映射到第 2 个页框。页表项的最后一位用来标记页是否在内存中。

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。因此对于虚拟地址（0010 000000000100），前 4 位是用来存储页面号，而后 12 位存储在页中的偏移量。

（0010 000000000100）根据前 4 位得到页号为 2，读取表项内容为（110 1），它的前 3

为为页框号，最后 1 位表示该页在内存中。最后映射得到物理内存地址为（11000000000100）。



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

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。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

### Optimal

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：

## 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时最近最久未访问的。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。



4	7	0	7	1	0	1	2	1	2	6
							2	1	2	6
				1	0	1	1	2	1	2
		0	7	7	1	0	0	0	0	1
	7	7	0	0	7	7	7	7	7	0
4	4	4	4	4	4	4	4	4	4	7

### 3. 最近未使用

#### NRU, Not Recently Used

首先，系统为每一页面设置了两个状态位。当页面被访问（读或写）时设置 R 位；当页面（即修改页面）被写入时设置 M 位。当启动一个进程时，它的所有页面的两个位都由操作系统设置成 0，R 位被定期地（比如在每次时钟中断时）清零，以区别最近没有被访问的页面和被访问的页面。

当发生缺页中断时，操作系统检查所有的页面并根据它们当前的 R 位和 M 位的值，把它们分为 4 类：

- 第 0 类：没有被访问，没有被修改
- 第 1 类：没有被访问，已被修改
- 第 2 类：已被访问，没有被修改
- 第 3 类：已被访问，已被修改

NRU 算法随机地从类编号最小的非空类中挑选一个页面淘汰之。

算法隐含的意思是，在最近一个时钟滴答中（典型的时间是大约 20ms）淘汰一个没有被访问的已修改页面要比一个被频繁使用的“十净”页面好。NRU 主要优点是易于理解和能够有效地被实现，虽然它的性能不是最好的，但是已经够用了。

### 4. 先进先出

#### FIFO, First In First Out

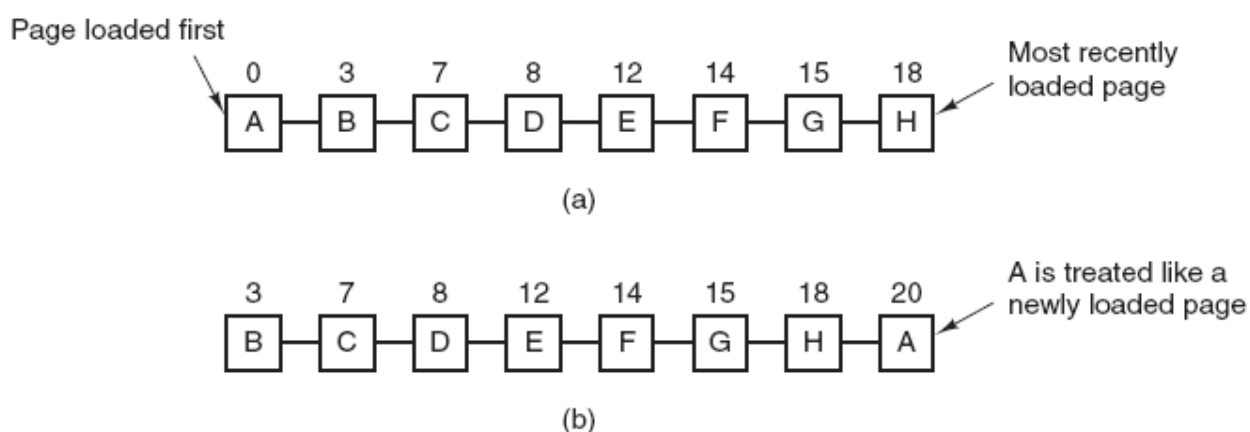
选择换出的页面是最先进入的页面。

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 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

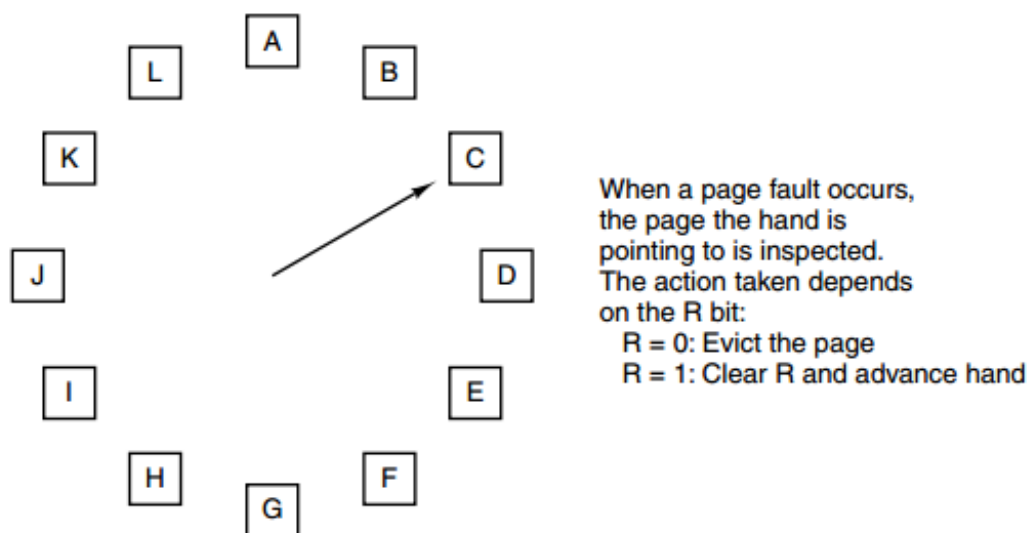


**Figure 3-15.** Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and A has its  $R$  bit set. The numbers above the pages are their load times.

## 6. 时钟

### 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链接起来，再使用一个指针指向最老的页面。

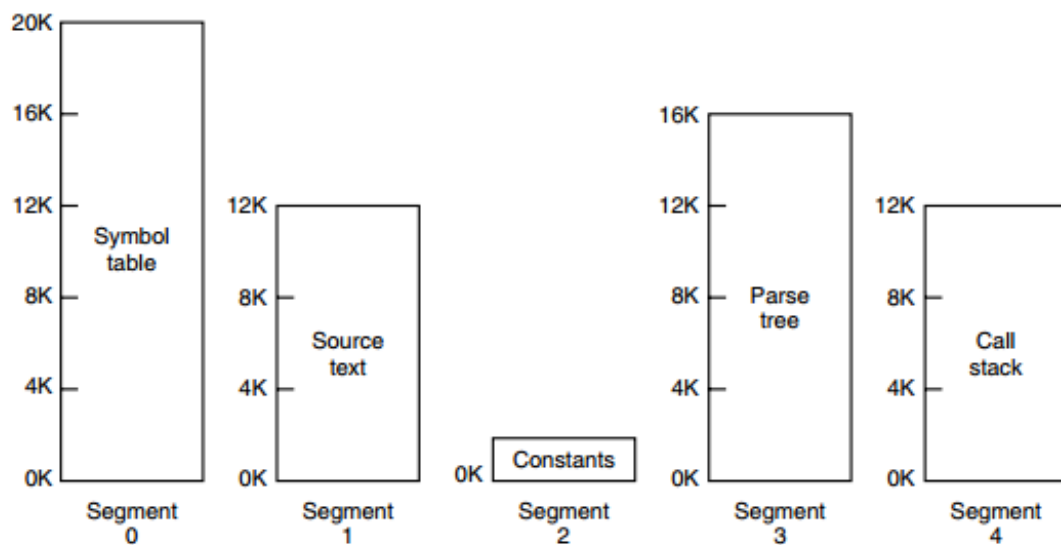
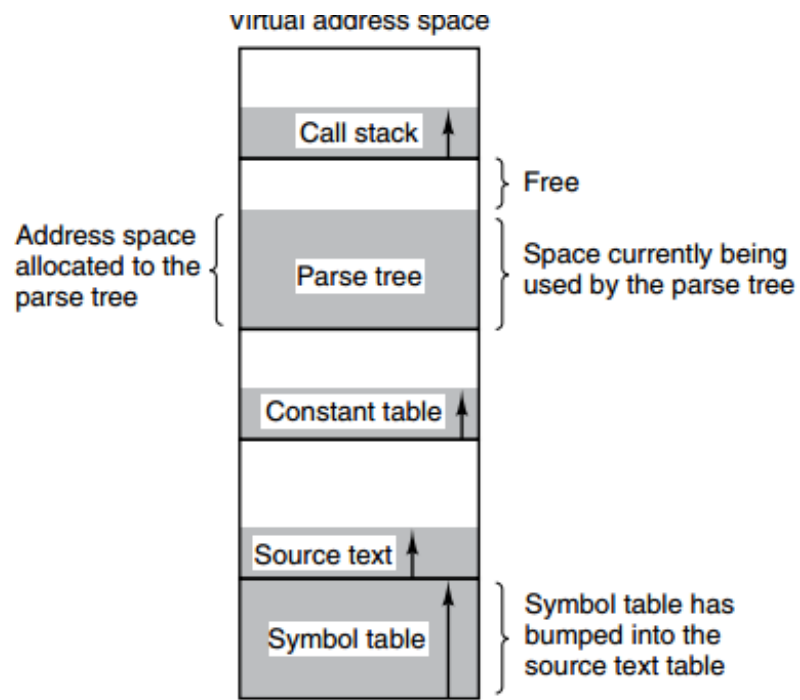


**Figure 3-16.** The clock page replacement algorithm.

## 分段

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。



## 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。

## 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。

## 五、设备管理

### 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（主轴旋转磁盘，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。

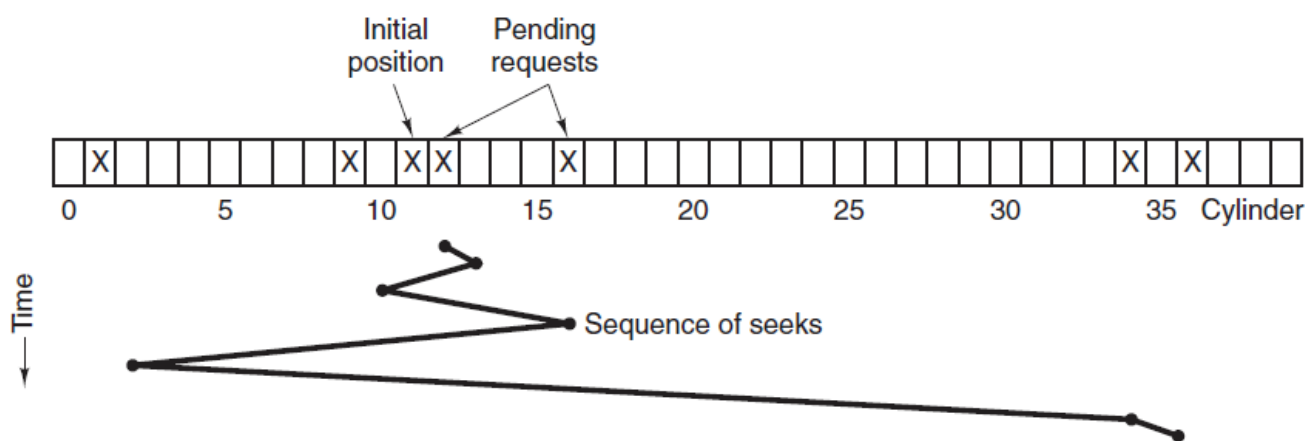
优点是公平和简单。缺点也很明显，因为未对寻道做任何优化，使平均寻道时间可能较长。

#### 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，两边的磁道请求更容易出现饥饿现象。



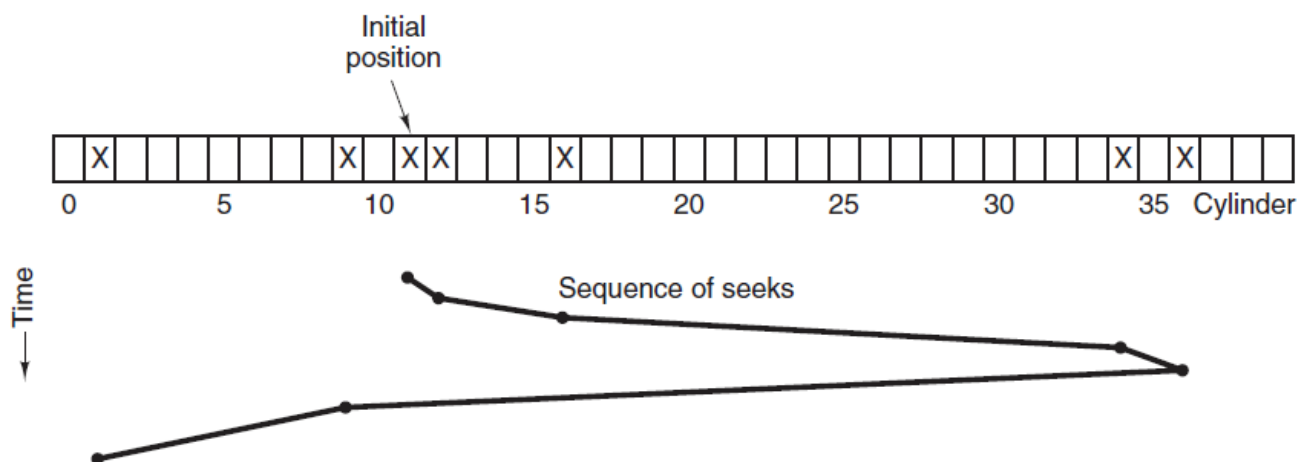
### 3. 电梯算法

#### 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。



## 六、链接

### 编译系统

以下是一个 hello.c 程序：

```

1.  #include <stdio.h>
2.
3.  int main()
4.  {
5.      printf("hello, world\n");
6.      return 0;
7.  }

```

在 Unix 系统上，由编译器把源文件转换为目标文件。

```

1.  gcc -o hello hello.c

```

这个过程大致如下：

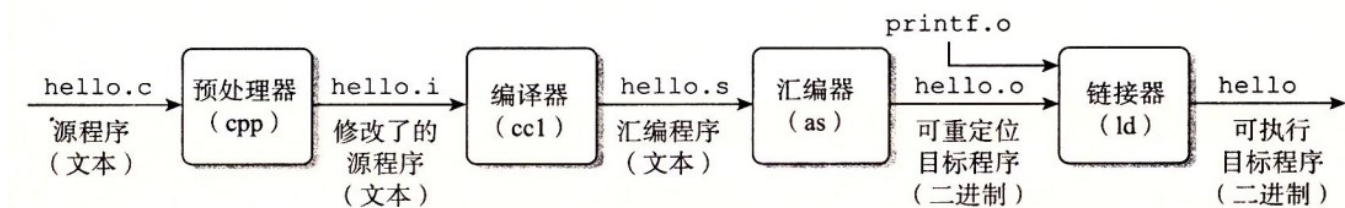


图 1-3 编译系统

## 静态链接

静态连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。

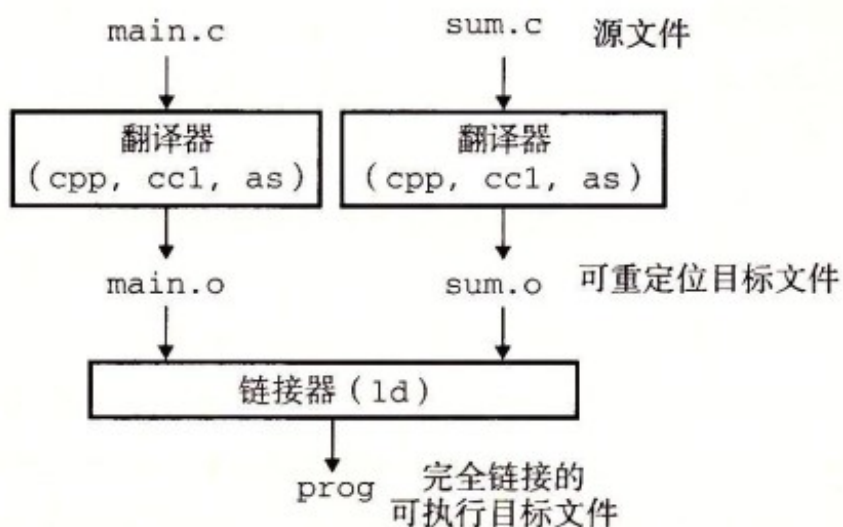


图 7-2 静态链接。链接器将可重定位目标文件组合起来，形成一个可执行目标文件 `prog`

## 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，可以在加载或者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。

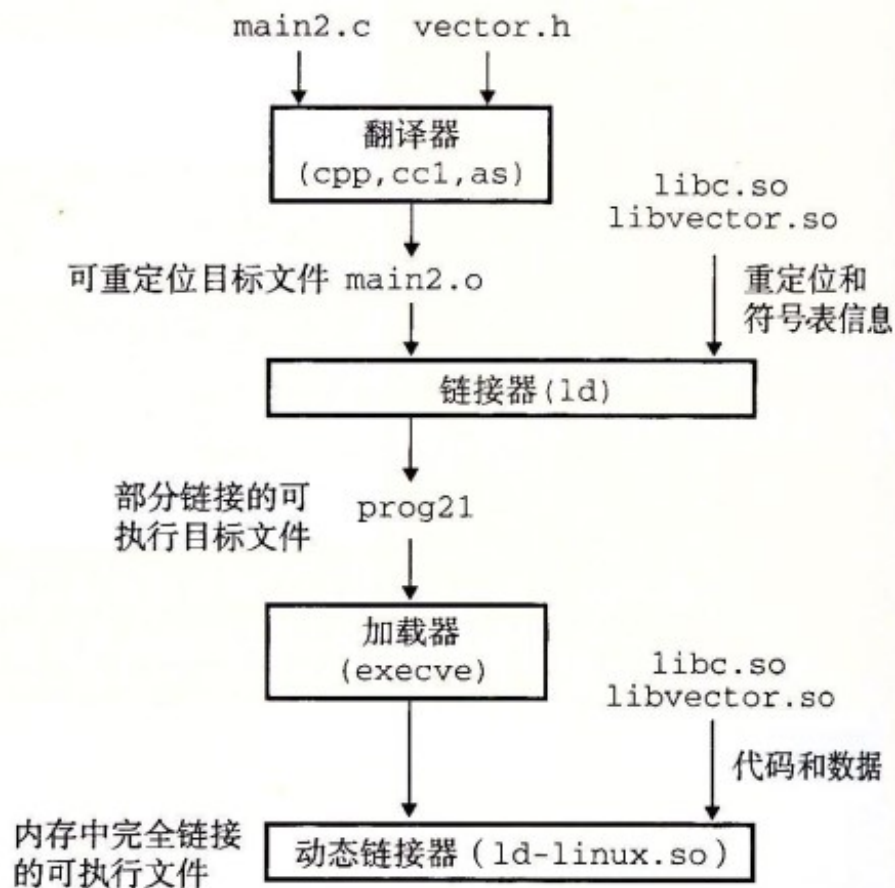


图 7-16 动态链接共享库

## 参考资料

- Tanenbaum A S, Bos H. Modern operating systems[M]. Prentice Hall Press, 2014.
- 汤子瀛, 哲凤屏, 汤小丹. 计算机操作系统[M]. 西安电子科技大学出版社, 2001.
- Bryant, R. E., & O' Hallaron, D. R. (2004). 深入理解计算机系统.
- [Operating System Notes](#)
- [进程间的几种通信方式](#)
- [Operating-System Structures](#)
- [Processes](#)
- [Inter Process Communication Presentation\[1\]](#)

