目录

第七章.虚拟内存

1. 分页
    1.1 页表
    1.2 一级分页系统中的地址转换
    1.3 两级分页系统中的地址转换
    1.4 倒排页表
    1.5 转换检测缓冲区(TLB)
2. 分段
    2.1 分段系统中的地址转换
    2.2 保护和共享
3. 段页式
    3.1 段页式系统中的地址转换
4. 内存管理中的相关策略
    4.1 读取策略
    4.2 放置策略
    4.3 置换策略
    4.4 驻留集管理
    4.5 清除策略
    4.6 加载控制

内存管理补充和总结

虚拟内存
分页系统地址映射
页面置换算法（P180)
    出现抖动的原因
    影响F（缺页中断率）的因素：
    1. 最佳
    2. 最近最久未使用
    3. 最近未使用
    4. 先进先出
    5. 第二次机会算法
    6. 时钟
分段
段页式
分页与分段的比较

替换算法实验

① 先进先出的算法（FIFO） 要求用数组或链表方法实现
        FIFO缓存淘汰算法的实现(Go语言实现)
② 最近最少使用算法（LRU） 要求用计数器或堆栈方法实现

第八章.单处理器调度

1.进程调度类型
2.调度算法
    2.1 短程调度准则
    2.2 优先级调度
    2.3 选择调度策略
    2.4 调度实例分析

第九章.I/O管理与磁盘调度

1.I/O缓冲
    1.1 单缓冲
    1.2 双缓冲(缓冲交换)
    1.3 循环缓冲
    1.4 I/O缓冲的作用
2.磁盘调度
    2.1 磁盘性能参数

计算机操作系统 - 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，在同一时刻只允许一个进程访问，需要用同步机制来实现互斥访问。

> **这个同时是宏观上的，而在微观上，这些进程可能是交替的对资源镜像访问的（即分时共享）**

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

多道程序下，允许多个程序并发执行，但是由于资源有限，所以：

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。

5. 总结

> **总结：**
>
> 1. 并发和共享互为存在条件
> 2. 没有并发和共享，就谈不上虚拟和异步，因此并发和共享就是操作系统的两个基本特征。

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。

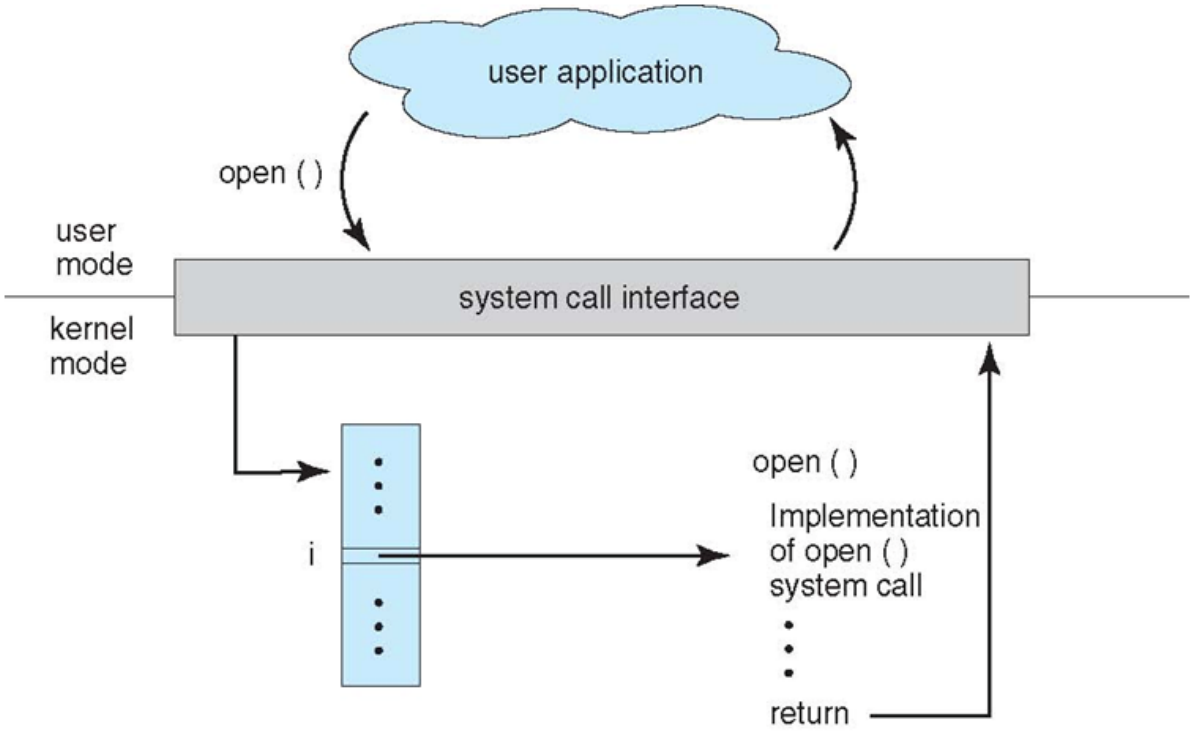
4. 设备管理

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();

宏内核和微内核

1. 宏内核

宏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。

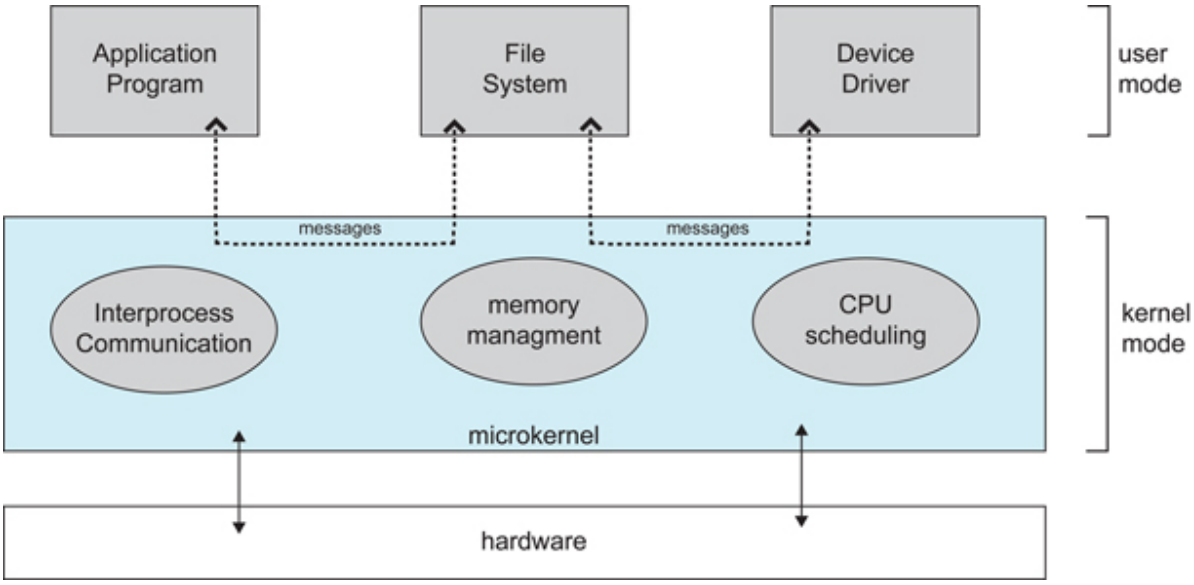
由于各模块共享信息，因此有很高的性能。

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。

第一章.计算机系统概述

1.基本构成

计算机的四个主要组件

- 处理器
- 内存
- I/O模块
- 系统总线

2.指令的执行

基本指令周期，指令处理包括2步：

- 处理器从存储器一次读一条指令
- 执行每条指令

处理器中的PC保存下一条指令的地址，IR保存当前即将执行的指令

3.中断

允许“其他模块”（I/O、存储器）中断“处理器”正常处理过程的机制

3.1 目的

提高CPU利用率，防止一个程序垄断CPU资源

3.2 类型

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

3.3 中断控制流

I/O程序：

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

中断：短I/O等待

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

中断：长I/O等待

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

3.4 中断处理

中断激活了很多事件，包括处理器硬件中的事件及软件中的事件

被中断程序的信息保存与恢复：

3.5 多个中断

在处理一个中断的过程中，可能会发生另一个中断，处理多个中断有2种方法

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

4.存储器的层次结构

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

4.1 高速缓存

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

上图中高速缓存通常分为多级：L1、L2、L3

5.直接内存存取(DMA)

针对I/O操作有3种可能的技术 * 可编程(程序控制)I/O（需处理器干预） * 中断驱动I/O（需处理器干预） * 直接内存存取

当处理器正在执行程序并遇到一个I/O相关的指令时，它通过给相应的I/O模块发命令来执行这个指令：

1）使用可编程I/O时，I/O模块执行请求的动作并设置I/O状态寄存器中相应的位，**但它并不进一步通 知处理器，尤其是它并不中断处理器**，因此处理器在执行I/O指令后，还需定期检查I/O模块的状态。为了确定I/O模块是否做好了接收或发送更多数据的准备，处理器等待期间必须不断询问I/O模块的状态，这会严重降低整个系统的性能

2）如果是中断驱动I/O，在给I/O模块发送I/O命令后，处理器可以继续做其它事。当I/O模块准备好与处理器交换数据时，会中断处理器并请求服务，处理器接着响应中断，完成后再恢复以前的执行过程

尽管中断驱动I/O比可编程I/O更有效，但是**处理器仍需要主动干预在存储器和I/O模块直接的数据传送，并且任何数据传送都必须完全通过处理器。**由于需要处理器干预，这两种I/O存在下列缺陷：

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

3）因此，当需要移动大量数据时，需要使用一种更有效的技术：直接内存存取。DMA功能可以由系统总线中一个独立的模块完成，也可以并入到一个I/O模块中。

DMA的工作方式如下，当处理器需要读写一块数据时，它给DMA模块产生一条命令，发送下列信息：

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

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

第二章.操作系统概述

1.操作系统的目标和功能

操作系统是控制应用程序执行的程序，并充当应用程序和计算机硬件之间的接口

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

- 易扩展性

2.操作系统的发展

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

以下为多道批处理系统与分时系统的比较

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

3.现代操作系统

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

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

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

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

第三章.进程

1.进程的定义

进程有以下定义：

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

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

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

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

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

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

1.1 gdb编译程序

编译c语言源程序

```
$gcc -o test -g test.c
```

没有错误提示，表示编译成功；否则返回第3步进行修改。

调试运行

```
$ gdb test
  :
  :<出现提示信息>
  :
  (gdb) 输入r un <enter> ,程序即可运行，输出运行结果。
  ./test
```

注意：用gcc编译时会出现：“在函数‘main’中：警告：隐式声明与内建函数‘execl’不兼容”的提示，可以忽略，不影响程序的执行。

1.2 进程系统调用

1.进程的创建 fork()

格式：`pid=fork()`

功能：创建一个新进程，新进程与父进程具有相同的代码，父子进程都从fork()之后的那条语句开始执行。

- 对于父进程，pid 的值>0;
- 对于子进程，pid的值=0;
- 创建失败，pid的值<0。

2.进程的终止 exit()

格式：`exit(status)`

功能：终止当前进程的执行，status是一个整数，其值可以返回父进程。

3.进程的同步 wait()

格式：`wait()`

功能：父进程进入睡眠态，当子进程终止时被唤醒。

4．进程的延迟 sleep()

格式：`sleep(n)`

功能：当前进程**延迟n秒执行。**

5．进程执行另一程序 execl()

1.3 进程创建

```c
#include <stdio.h>
#include <stdlib.h>
  main()
  {  int i;
     printf("just 1 process.\n"); //1    -- 过程
     i=fork();
     if (i==0)   //true
        printf("I am child.\n");  //4    -- 子进程
     else
        if (i>0)
           printf("I am parent.\n");   //2    -- 父进程
        else
           printf("fork() failed.\n");  //
     printf("program end.\n");   //3,5     -- 程序结束
  }
```

编译：

```
root@ubuntu:/c# ./a
just 1 process.
I am parent.
program end.
I am child.
program end.
```

> - **newproc：建立子进程映像；保护现场—>u区,返回`0`**
> - **swtch：选择子进程运行；从u区—>恢复现场返回`1`**
> - 子进程和父进程结束都调用`printf("program end.\n");`
> - `p110`页

调试（后面省略）

```
root@ubuntu:/c# gdb a
```

```
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a...
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands.
breakpoints -- Making program stop at certain points.
data -- Examining data.
files -- Specifying and examining files.
internals -- Maintenance commands.
obscure -- Obscure features.
running -- Running the program.
stack -- Examining the stack.
status -- Status inquiries.
support -- Support facilities.
tracepoints -- Tracing of program execution without stopping the program.
user-defined -- User-defined commands.

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Type "apropos -v word" for full documentation of commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb) run
Starting program: /c/a
just 1 process.
[Detaching after fork from child process 72855]
I am parent.
program end.
I am child.
program end.
[Inferior 1 (process 72851) exited normally]
```

1.4 进程的同步

```
#include <stdio.h>
#include <stdlib.h>
main()
{  int i;
   printf("just 1 process.\n"); //1
   i=fork();
   if (i>0)
      {   printf("I am parent.\n");  //2
          wait();
      }
   else
      if (i==0)
          { printf("I am child.\n"); //3
            exit(1);
           }
   printf("program end.\n");   //4
}
```

编译:

```
root@ubuntu:/c# ./a
just 1 process.
I am parent.
I am child.
program end.
```

1.5 进程的延迟

```
#include<stdio.h>
#include<stdlib.h>
int main()
{  int i,j,k;
   i=fork();
   if (i>0)
     for (j=1;j<=5;j++)  {
        sleep(1);
        printf("I am parent.\n"); }   //1
   else
      if (i==0)
         for(k=1;k<=5;k++) {
            sleep(1);
            printf("I am child.\n"); }   //2
   return 0;
}
```

编译:

```
root@ubuntu:/c# ./c
I am parent.
I am child.
I am parent.
I am child.
I am parent.
I am child.
I am parent.
I am child.
I am parent.
I am child.
```

> 进程在请求资源得不到满足或等待某一事件发生时，都要用sleep进入睡眠状态，等到资源可以满足，通过wakeup唤醒

1.6 进程执行另一程序

execl函数

Linux下头文件include <unistd.h>

函数定义

```
int execl(const char *path, const char *arg, ...);
```

函数说明：
execl()其中后缀"l"代表list也就是参数列表的意思，第一参数path字符指针所指向要执行的文件路径，接下来的参数代表执行该文件时传递的参数列表：argv[0],argv[1]... **最后一个参数须用空指针NULL作结束。**

函数返回值：
成功则不返回值， 失败返回-1， 失败原因存于errno中，可通过perror()打印

```
#include <unistd.h>/*** File: execl.c**/
#include <iostream>
using namespace std;
int main()
{
  // 执行/bin目录下的ls, 第一参数为程序名ls, 第二个参数为"-al", 第三个参数为"/etc/passwd"
  if(execl("/bin/ls", "ls", "-al", "/etc/passwd", (char *) 0) < 0)
  {
      cout<<"execl error"<<endl;
  }
   else
  {
     cout<<"success"<<endl;
  }
  return 0;
}
```

execl函数特点:

当进程调用一种exec函数时，该进程完全由新程序代换，而新程序则从其main函数开始执行。因为调用exec并不创建新进程，所以前后的进程ID并未改变。exec只是用另一个新程序替换了当前进程的正文、数据、堆和栈段。

用另一个新程序替换了当前进程的正文、数据、堆和栈段。

当前进程的正文都被替换了，那么execl后的语句，即便execl退出了，都不会被执行。

程序

```
/*************************************************************************
    > File Name: d.c
    > Author: smile
    > Mail: 3293172751nss@gmail.com
    > Created Time: Thu 26 May 2022 04:44:39 AM PDT
 ************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <iostream>
using namespace std;
int main(){
    int  i;
    char name[20];  //存放名称
    printf("Please input a directory name:\n");
    scanf("%s",name);
    i = fork();  //创建进程
    if (i==0)
       execl("/bin/ls","ls","-l",name,(char *) 0);
    return 0;
   }
```

编译：

```
root@ubuntu:/c# g++ -o d d.cpp
root@ubuntu:/c# ./d
Please input a directory name:
/etc/passwd
root@ubuntu:/c# -rw-r--r-- 1 root root 2939 May 11 03:13 /etc/passwd
ls
a  abcd.tar  a.c  b.c  c  c.c  d  d.cpp  e  e.c
```

案例

编写一段程序，使用系统调用fork()创建两个子进程。当此程序运行时，在系统中有一个父进程和两个子进程活动。让每一个进程在屏幕上显示一个字符：父进程显示字符“a”；子进程分别显示字符“b”和字符“c”。试观察记录屏幕上的显示结果，并分析原因。

```c
#include <stdio.h>
#include <stdlib.h>
  main()
  { int i,j;
   if (i=fork() == 0 ){printf(" I am child c\n");exit(0);}
   if (j=fork() == 0 ){printf(" I am child b\n");exit(0);}
   printf("I am parent a\n");
  }
```

编译：

```
root@ubuntu:/c# ./e
 I am parent a
 I am child c
 I am child b
```

原因：

1. 开始创建进程的时候`newproc()`判断，`fork()==1`，那么`if`不满足，所以先输出`I am parent a`
2. 进程被创建后需要返回现场，`if`作用域是当前语句之中

我们可以把每个fork当成一个二叉分支，执行fork的时候，一个程序变成了两个，在同一个起点，但一个走父程序，一个走子程序，两个程序的代码是一样的，只是代码执行路径不一样（被fork返回值控制），父程序由于fork返回>0不进if {}，子程序fork =0，所以会进if {}

> Fork就是叉子的意思，就是说主进程到这里一份变三分，怎么知道哪个是主呢，就看fork的返回值(-1，1，0)

而为啥这个`I am parent a`在前面，就是当你`fock()`创建的那一瞬间，顺序并非一种而是随机的。

也不能说完全随机，会根据执行分叉点父子各自要执行的代码量来定，你把程序复制两份，在fork点一个走i=0结果，一个走i>0结果，然后输出到相同stdout。后续同样，i>0那个过程继续复制一份程序，一个走j=0, 另一个走j>0，两次分叉出来三个程序。

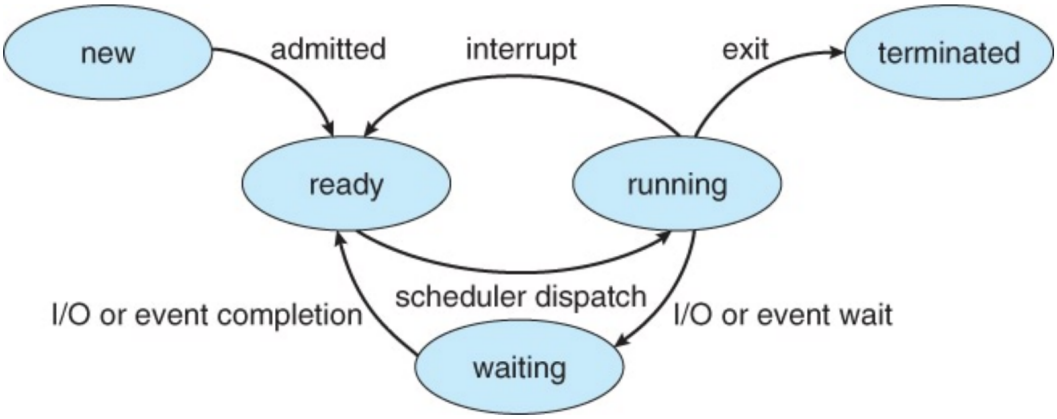
推荐文章，尤其是后面的例子很好

https://www.section.io/engineering-education/fork-in-c-programming-language/#:~:text=In%20the%20computing%20field%2C%20fork,also%20kills%20the%20child%20process.

1.7 什么是 Fork()?

在计算领域，`fork()`是在类 Unix 操作系统上创建进程的主要方法。该函数在原始进程之外创建一个称为*子进程的新副本，即*父进程。当父进程由于某种原因关闭或崩溃时，它也会杀死子进程。

让我们从流程的生命周期开始：



操作系统为每个进程使用唯一的 id 来跟踪所有进程。`fork()` 不接受任何参数并返回一个 int 值，如下所示：

- 零：如果是子进程（创建的进程）。
- 正值：如果是父进程。
- 负值：如果发生错误。

注意：以下代码仅在基于 Linux 和 UNIX 的操作系统中运行。如果您运行的是 Windows，那么我建议您使用Cygwin。

让我们跳到实践部分，我们将创建从简单级别到高级级别的示例。

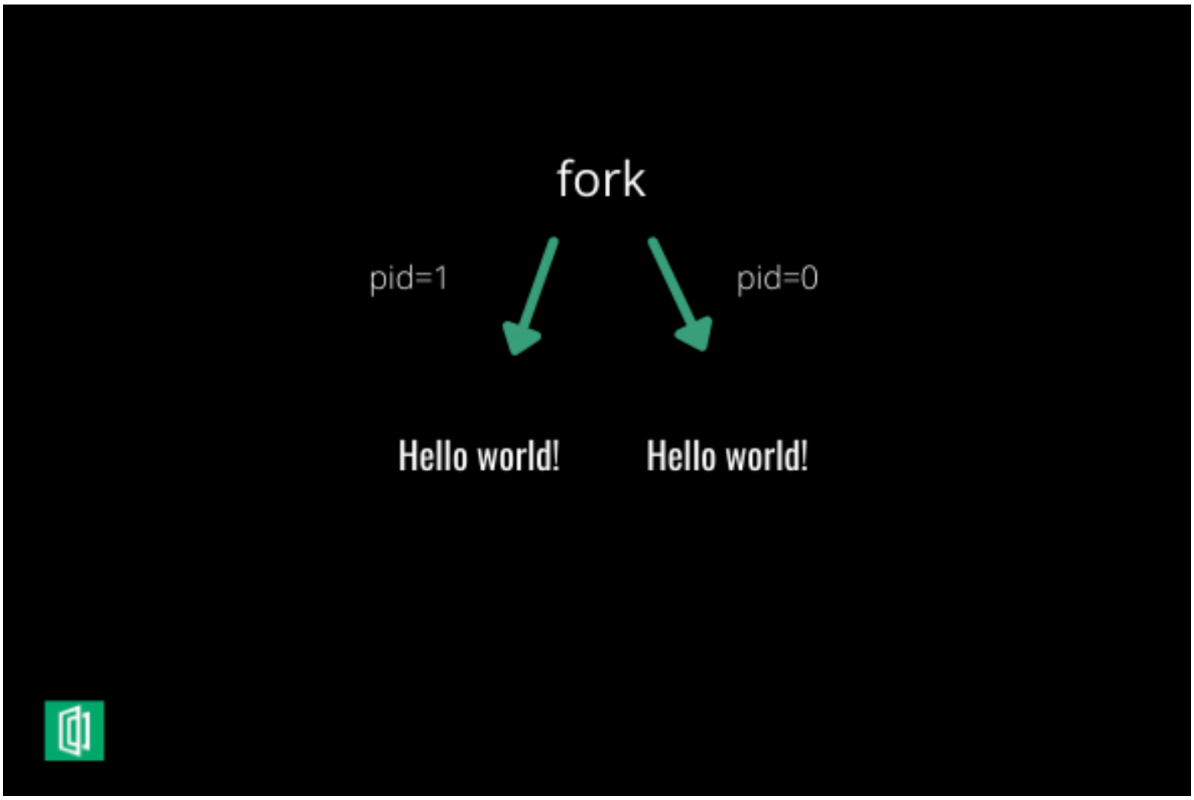
你好世界！

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    /* fork a process */
    fork();
    /* the child and parent will execute every line of code after the fork (each separately)*/
    printf("Hello world!\n");
    return 0;
}
```

输出将是：

```
Hello world!
Hello world!
```

其中一个输出来自父进程，另一个来自子进程。



简单地说，**我们可以知道结果是 n 的 2 次方，其中 n 是 fork() 系统调用的数量。**

例如：

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
    fork();
    fork();
    fork();
    printf("Hello world!\n");
    return 0;
}
```

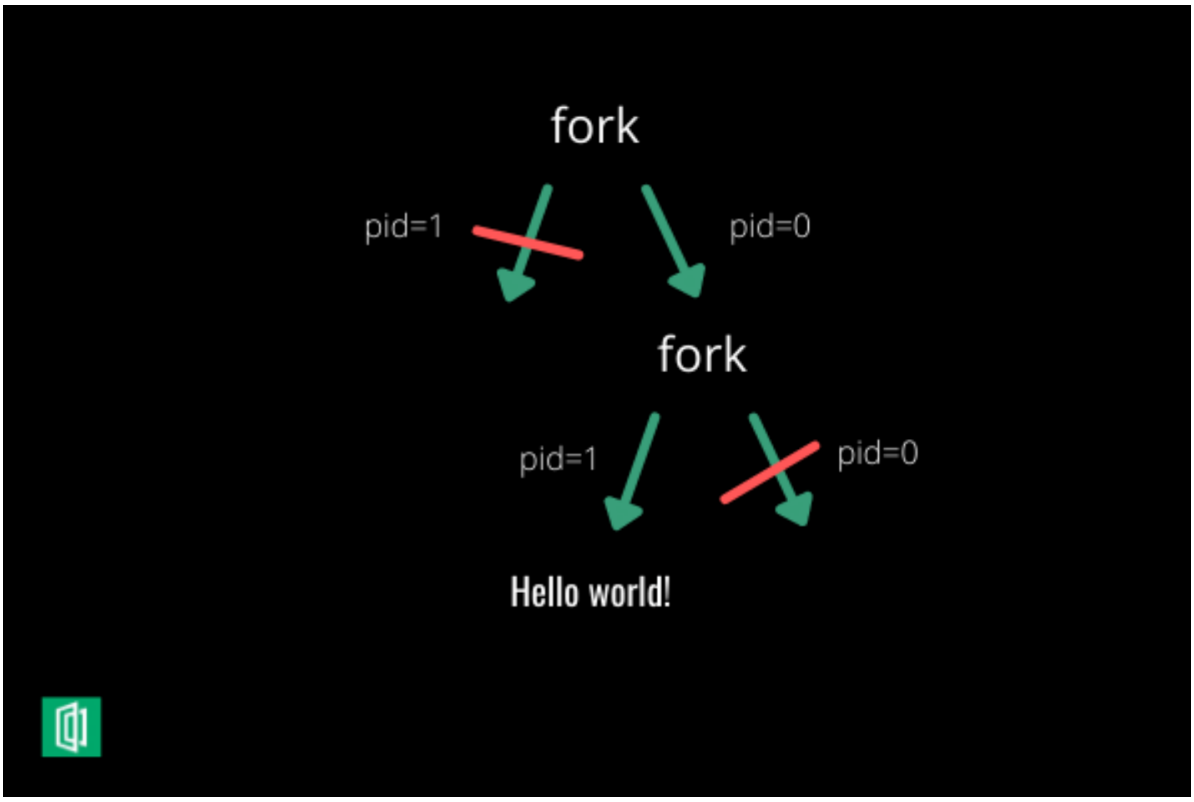
结果是：

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

另一个例子是：

```c
int main() {
  if(fork() == 0)    //子程序才能往下走
    if(fork()) //主程序才能往下走
      printf("Hello world!!\n");
  exit(0);
}
```

我画了一个简短的草图来帮助你理解这个想法：



在第一个`if`条件中发生了一个分叉，它正在检查它是否是子进程，然后它继续执行它的代码。否则（如果它是父进程）它不会通过那个`if`. 然后，在第二个`if`中，它将只接受持有积极 id 的父进程。

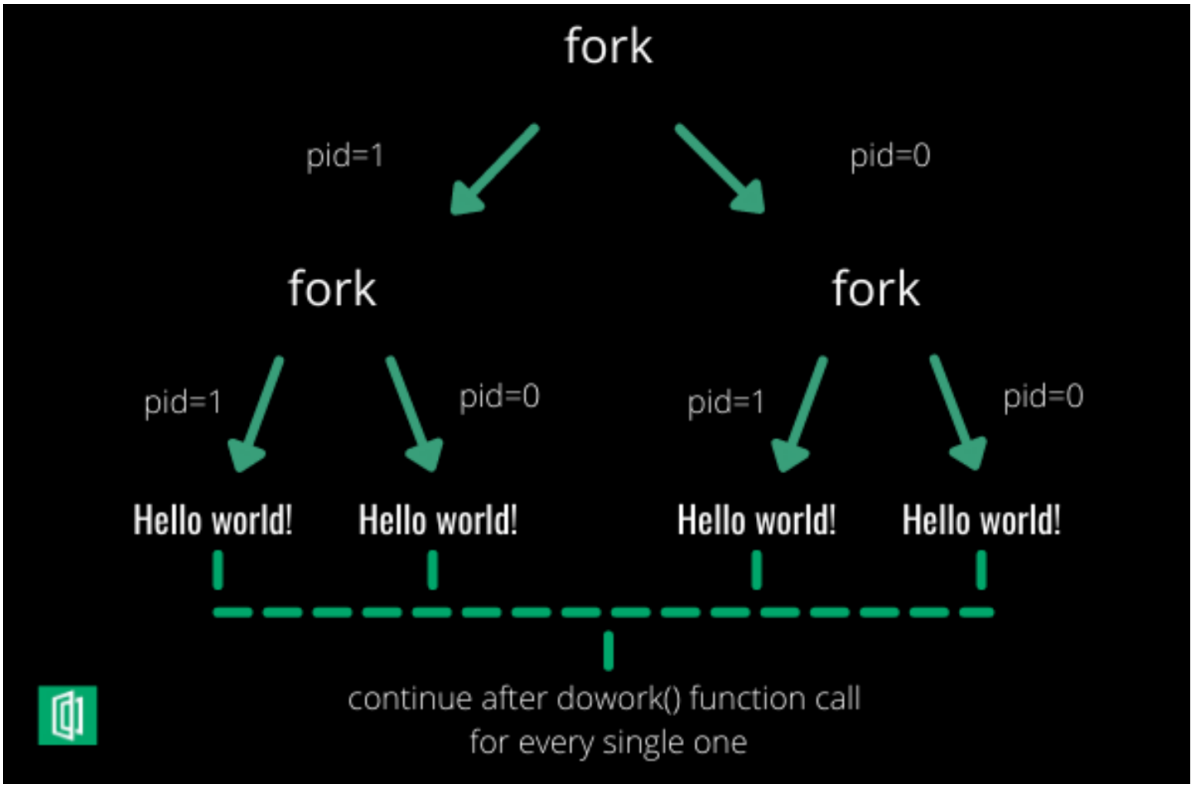
结果，它只会打印一个“Hello world！”。

现在尝试执行以下代码并将您的结果与我们的结果进行比较：

```c
int doWork(){
    fork();
    fork();   //2**2
    printf("Hello world!\n");
}
int main() {
    doWork();
    printf("Hello world!\n");
    exit(0);
}
```

结果将是：

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```



因为当内部分叉的进程`dowork()`打印`Hello World!`时，它将在函数调用之后继续主代码并打印`Hello World!`然后退出。

高级示例

当一个进程创建一个新进程时，那么执行退出有两种可能：

- 父级继续与其子级同时执行。
- 父级等待直到其部分或所有子级终止。

```c
#include <sys/types.h>
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main(int argc, char *argv[]) {

   /* fork a child process */
   pid_t pid = fork();

   if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
   }
```

```
15
16      else if (pid == 0) { /* child process */
17          printf("I'm the child \n"); /* you can execute some commands here */
18      }
19
20      else { /* parent process */
21          /* parent will wait for the child to complete */
22            wait(NULL);
23          /* When the child is ended, then the parent will continue to execute
    its code */
24            printf("Child Complete \n");
25      }
26  }
```

等待调用系统 `wait(NULL)` 将使父进程等待，直到子进程执行完所有命令。

结果将是：

```
1  I'm the child
2  Child Complete
```

另一个例子：

```
1  #include <sys/types.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5  #include <stdlib.h>
6  int main(int argc, char *argv[])
7  {
8      printf("I am: %d\n", (int) getpid());
9
10     pid_t pid = fork();
11     printf("fork returned: %d\n", (int) pid);
12
13     if (pid < 0) { /* error occurred */
14         perror("Fork failed");
15     }
16     if (pid == 0) { /* child process */
17         printf("I am the child with pid %d\n", (int) getpid());
18                 printf("Child process is exiting\n");
19                 exit(0);
20         }
21     /* parent process */
22     printf("I am the parent waiting for the child process to end\n");
23         wait(NULL);
24         printf("parent process is exiting\n");
25         return(0);
26 }
```

结果将类似于：

```
I am: 2337
fork returned: 2338
I am the parent waiting for the child process to end
fork returned: 0
I am the child with pid 2338
Child process is exiting
parent process is exiting
```

1.8 进程通信

操作系统提供了一种称为进程通信（IPC）的机制让协作进程之间实现彼此之间的通信。IPC是利用消息（message），明确地将信息从一个进程地地址拷贝到另一个进程的地址空间，而不适用共享存储器的一种通信机制。

send和receive原语

> 在信息通信过程中，接收方和发送方之间有一个协议，是双方都认可其中的信息格式。
>
> 在信箱通信中，除了定义信箱结构外，还需要定义原语和接收原语。发送原语和接收原语一般有两种选项，这两个原语都有同步和异步两种语义。

- **send原语：send原语可以是同步或者异步的**
- **receive原语可以是阻塞或非阻塞的**

一、进程的信号通信系统调用

1. `Signal(sig,function)`

功能：允许调用进程控制软中断信号的处理。`Sig`表示信号，如`SIGINT`表示按ctrl+c键，`SIGQUIT`表示按ctrl+\键。

2. `Kill(pid,sig)`

功能：向进程号为pid的进程发送信号sig。

二、进程的管道通信系统调用

1. `pipe()`系统调用

格式：`status=pipe(fd)`

```
int fd[2];
```

功能：fd用来存放标识管道的两个文件描述符，如果调用成功，fd[0]用于从管道读，fd[1]用于向管道写。**如果管道为空，读管道的进程等待；如果管道已满，写管道的进程等待。**

2. `close()`系统调用

格式：`close(fd)`

功能：fd为欲关闭文件的文件描述符，用于关闭文件。

3. `read/write`系统调用

格式：`read(fd,buffer,size)`

```
write(fd,buffer,size)
```

功能：fd是读/写的文件描述符，buffer是字节指针，存放读/写的字节流的地址，size是要求读/写的字节数。

4. `Lockf(files,function,size)`

功能：用作锁定文件的某些段或整个文件。其中：files是文件描述符，0表示标准输入，1表示标准输出；function是锁定和解锁，1表示锁定，0表示解锁；size是锁定或解锁的字节数，若用0，表示从当前的位置到文件尾。

三、进程之间的信号通信

```
/*************************************************************************
    > File Name: 1.c
    > Author: smile
    > Mail: 3293172751nss@gmail.com
    > Created Time: Thu 26 May 2022 05:33:17 AM PDT
 ************************************************************************/
#include <stdio.h>
#include <signal.h>
//status=pipe(fd)
void waiting(),stop();
int wait_mark;
main() {
    int p1,p2;
    while ((p1=fork())==-1);   //what空语句号，创建
    if (p1>0) {    //父进程进入
        while ((p2=fork())==-1);  //创建一个进程p2
        if (p2>0) {
            wait_mark=1;
            signal(SIGINT,stop);         //此时表示中断信号
//1. `Signal(sig,function)`功能：允许调用进程控制软中断信号的处理。`Sig`表示信号，如`SIGINT`表示按ctrl+c键，`SIGQUIT`表示按ctrl+\键。
            alarm(1);
            signal(SIGALRM,stop);
            waiting();
            kill(p1,16);
            kill(p2,17);
            wait(0);
            wait(0);
            printf("parent process is killed!\n");
            exit(0);
        } else {  //子进程
            wait_mark=1;
            signal(17,stop); //进程之间进行通信，
```

```
            waiting();
            lockf(1,1,0);
//功能：用作锁定文件的某些段或整个文件。其中：files是文件描述符，0表示标准输入，1表示标准输出；function是锁定和解锁，1表示锁定，0表示解锁；size是锁定或解锁的字节数，若用0，表示从当前的位置到文件尾。
            printf("child process 2 is killed by parent!\n");
            lockf(1,0,0);
            exit(0);
        }
    } else {
        wait_mark=1;
        signal(16,stop);
        waiting();
        lockf(1,1,0);
        printf("child process 1 is killed by parent!\n");
        lockf(1,0,0);
        exit(0);
    }
}
void waiting() {
    while (wait_mark !=0);
}
void stop() {
    wait_mark=0;
}

```

编译：

```
root@ubuntu:/c# ./1
child process 2 is killed by parent!
child process 1 is killed by parent!
parent process is killed!
```

四、进程之间的管道通信

当执行pipe()系统调用时，系统将在内存中建立一个大小为10kb的无名管道文件，当执行fork()调用时，父子进程共享管道文件，父子进程都有两个文件描述符与管道文件对应。以下程序是父子进程通过管道进行通信。

```
#include <stdio.h>
#define MSGSIZE 16
char *msg1="hello,#1";
char *msg2="hello,#2";
char *msg3="hello,#3";
main() {
    char inbuf[MSGSIZE]; //定义一个数组，
    int p[2],j,pid;
    if(pipe(p)<0) {
//status=pipe(fd)
//fd用来存放标识管道的两个文件描述符，如果调用成功，fd[0]用于从管道读，fd[1]用于向管道写。如果管道为空，读管道的进程等待；如果管道已满，写管道的进程等待。
```

```
        perror("pipe call err");
        exit(1); //结束
    }
    if((pid=fork())<0) {   //fock() 2**2 不走返回err
        perror("fork call err");
        exit(2);
    }
    if (pid>0) {      //父进程走分支
        close(p[0]);
        /*关闭父进程读*/
        write(p[1],msg1,MSGSIZE);   //写入管道
        write(p[1],msg2,MSGSIZE);
        write(p[1],msg3,MSGSIZE);
        wait(0); //等待
    }
    if(pid==0) {   //子进程走分支
        close(p[1]);
        /*关闭子进程写*/
        for (j=0;j<3;j++) {
            read(p[0],inbuf,MSGSIZE);
            printf("------in child--------%s\n",inbuf);
        }
        exit(0);
    }
    exit(0);
}
```

编译：

```
root@ubuntu:/c# ./2
------in child--------hello,#1
------in child--------hello,#2
------in child--------hello,#3
```

上述程序中，子进程是否可以在父进程之前执行？为什么？

答：推测要是子程序在父程序之前执行，会进入死锁，程序会一直等待下去。

2.进程的状态

2.1 进程的创建与终止

进程按以下步骤创建：

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

会导致创建进程的事件：

会导致终止进程的事件：

2.2 两状态进程模型

2.3 五状态进程模型

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

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

除此之外，就绪队列也可以按照优先级组织成多个队列

2.4 引入"挂起态"的进程模型

为何引入?

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

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

因此，**可以把内存中某个进程的一部分或全部移出到磁盘中。**当内存中没有处于就绪状态的进程时，操作系统就把被阻塞的进程换出到磁盘中的"挂起队列"。操作系统在此之后取出挂起队列中的另一个进程，或者接受一个新进程的请求，将其纳入内存运行

"交换"是一个I/O操作，因而也可能使问题更加恶化。但是由于磁盘I/O一般是系统中最快的I/O(相对于磁带或打印机I/O)，所以交换通常会提高性能

进程模型

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

导致进程挂起的原因

3.进程的描述

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

内存表用于跟踪内(实)存和外存(虚拟内存)

使用**进程映像**来描述一个进程，进程镜像包括：**程序、数据、栈和进程控制块(属性的集合)**：

下图为一个典型的**进程映像**结构：

4.进程控制

4.1 执行模式

大多数处理器至少支持两种执行模式：

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

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

处理器如何知道它正在什么模式下执行及如何改变模式?

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

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

4.2 进程切换

在下列事件中，进程可能把控制权交给操作系统：

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

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

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

计算机操作系统 - 进程管理

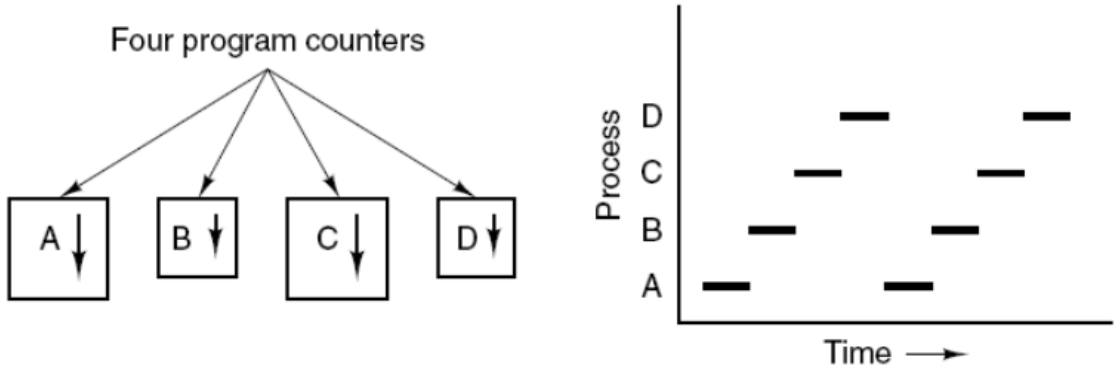
进程与线程

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。

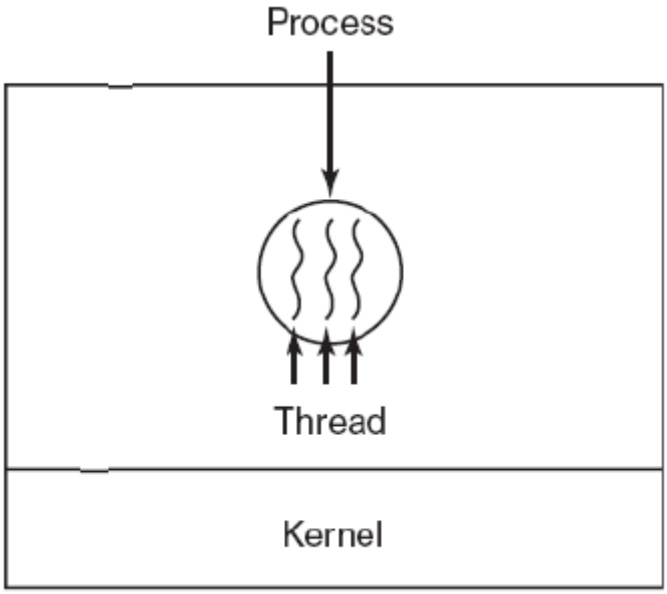


2. 线程

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。

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。

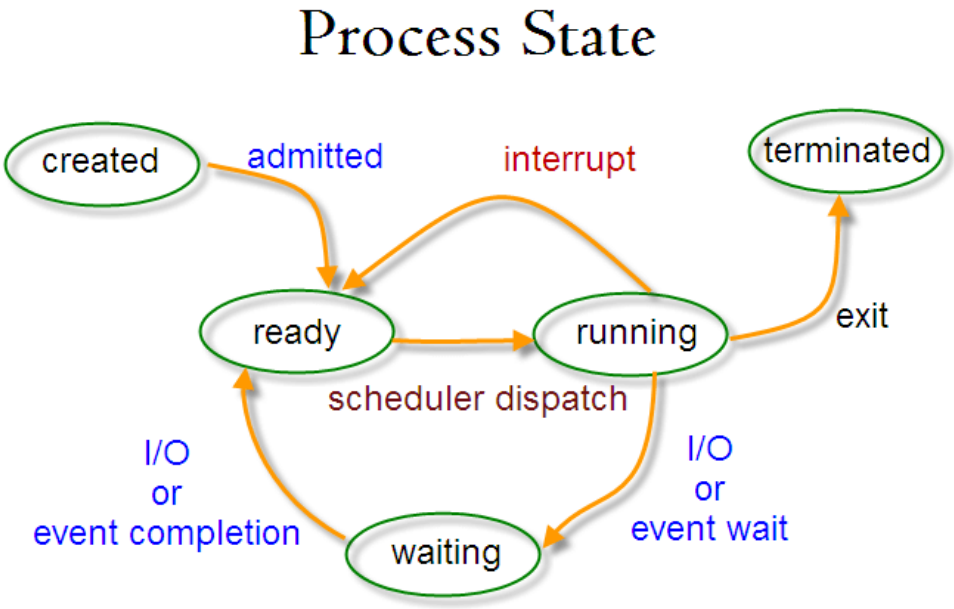
Ⅲ 系统开销

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。

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

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）

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。

1.2 短作业优先 shortest job first（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）

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。 当一个新的作业到达时，其整个运行时间与当前进程的剩余时间作比较。如果新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。

2. 交互式系统

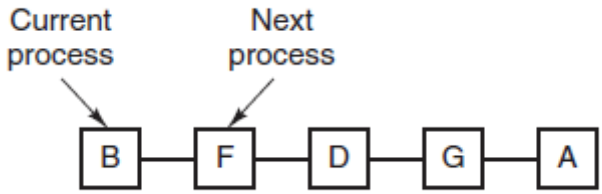
交互式系统有大量的用户交互操作，在该系统中调度算法的目标是快速地进行响应。

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。
- 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。

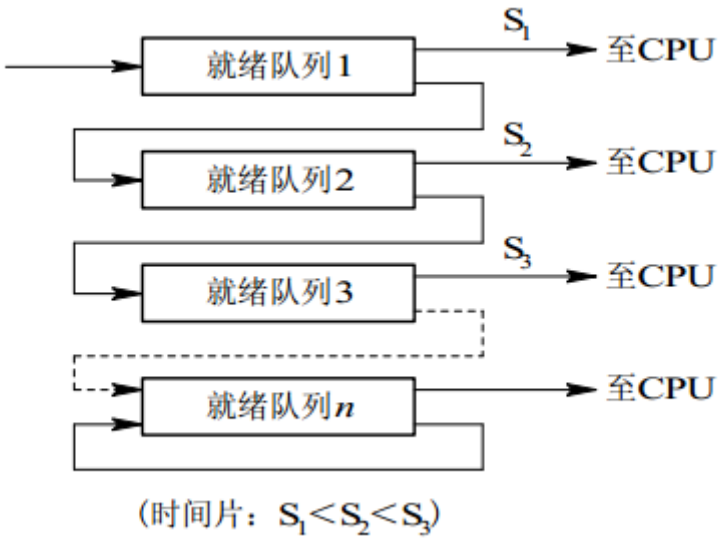
2.3 多级反馈队列

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 </br>

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。

使用管程实现生产者-消费者问题

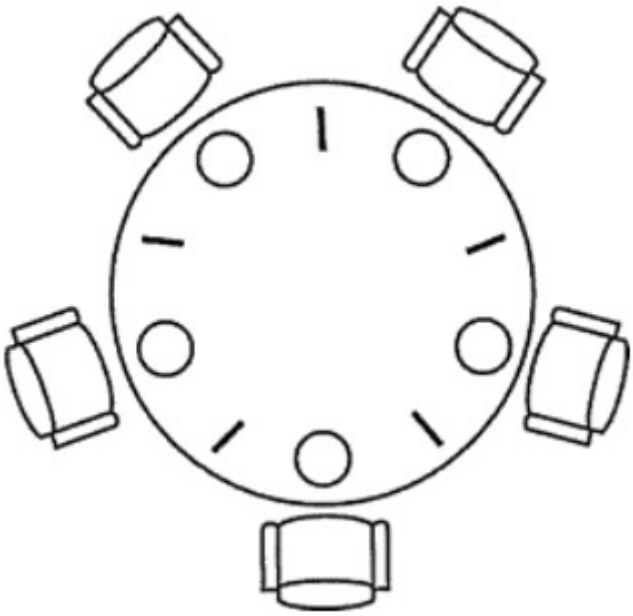
```
// 管程
monitor ProducerConsumer
    condition full, empty;
    integer count := 0;
    condition c;

    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

生产者和消费者问题前面已经讨论过了。

1. 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。

下面是一种错误的解法，如果所有哲学家同时拿起左手边的筷子，那么所有哲学家都在等待其它哲学家吃完并释放自己手中的筷子，导致死锁。

```c
#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。

```c
#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，临界区是 state 数组，对其修改需要互斥
semaphore s[N];              // 每个哲学家一个信号量

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

void take_two(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    check(i);
    up(&mutex);
    down(&s[i]); // 只有收到通知之后才可以开始吃，否则会一直等下去
}

void put_two(i) {
    down(&mutex);
    state[i] = THINKING;
    check(LEFT); // 尝试通知左右邻居，自己吃完了，你们可以开始吃了
    check(RIGHT);
    up(&mutex);
}

void eat(int i) {
```

```
    down(&mutex);
    state[i] = EATING;
    up(&mutex);
}

// 检查两个邻居是否都没有用餐，如果是的话，就 up(&s[i])，使得 down(&s[i]) 能够得到通知并继续执行
void check(i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=EATING)
{
        state[i] = EATING;
        up(&s[i]);
    }
}
```

2. 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);
    }
}
```

以下内容由 @Bandi Yugandhar 提供。

The first case may result Writer to starve. This case favous Writers i.e no writer, once added to the queue, shall be kept waiting longer than absolutely necessary(only when there are readers that entered the queue before the writer).

```
int readcount, writecount;                    //(initial value = 0)
semaphore rmutex, wmutex, readLock, resource; //(initial value = 1)

//READER
void reader() {
<ENTRY Section>
 down(&readLock);                 //  reader is trying to enter
 down(&rmutex);                   //   lock to increase readcount
  readcount++;
  if (readcount == 1)
   down(&resource);               //if you are the first reader then lock the resource
 up(&rmutex);                    //release  for other readers
 up(&readLock);                   //Done with trying to access the resource

<CRITICAL Section>
//reading is performed

<EXIT Section>
 down(&rmutex);                   //reserve exit section - avoids race condition with readers
 readcount--;                        //indicate you're leaving
  if (readcount == 0)             //checks if you are last reader leaving
   up(&resource);                 //if last, you must release the locked resource
 up(&rmutex);                    //release exit section for other readers
}

//WRITER
void writer() {
  <ENTRY Section>
  down(&wmutex);                   //reserve entry section for writers - avoids race conditions
  writecount++;                   //report yourself as a writer entering
  if (writecount == 1)            //checks if you're first writer
   down(&readLock);                  //if you're first, then you must lock the readers out. Prevent them from trying to enter CS
  up(&wmutex);                     //release entry section

<CRITICAL Section>
 down(&resource);                 //reserve the resource for yourself - prevents other writers from simultaneously editing the shared resource
  //writing is performed
 up(&resource);                  //release file

<EXIT Section>
  down(&wmutex);                   //reserve exit section
  writecount--;                   //indicate you're leaving
  if (writecount == 0)            //checks if you're the last writer
   up(&readLock);                 //if you're last writer, you must unlock the readers. Allows them to try enter CS for reading
  up(&wmutex);                     //release exit section
}
```

We can observe that every reader is forced to acquire ReadLock. On the otherhand, writers doesn't need to lock individually. Once the first writer locks the ReadLock, it will be released only when there is no writer left in the queue.

From the both cases we observed that either reader or writer has to starve. Below solutionadds the constraint that no thread shall be allowed to starve; that is, the operation of obtaining a lock on the shared data will always terminate in a bounded amount of time.

```
int readCount;                  // init to 0; number of readers currently accessing resource

// all semaphores initialised to 1
Semaphore resourceAccess;       // controls access (read/write) to the resource
Semaphore readCountAccess;      // for syncing changes to shared variable readCount
Semaphore serviceQueue;         // FAIRNESS: preserves ordering of requests (signaling must be FIFO)

void writer()
{
    down(&serviceQueue);           // wait in line to be servicexs
    // <ENTER>
    down(&resourceAccess);         // request exclusive access to resource
    // </ENTER>
    up(&serviceQueue);           // let next in line be serviced

    // <WRITE>
    writeResource();            // writing is performed
    // </WRITE>

    // <EXIT>
    up(&resourceAccess);         // release resource access for next reader/writer
    // </EXIT>
}

void reader()
{
    down(&serviceQueue);           // wait in line to be serviced
    down(&readCountAccess);        // request exclusive access to readCount
    // <ENTER>
    if (readCount == 0)         // if there are no readers already reading:
        down(&resourceAccess);     // request resource access for readers (writers blocked)
    readCount++;                // update count of active readers
    // </ENTER>
    up(&serviceQueue);           // let next in line be serviced
    up(&readCountAccess);        // release access to readCount

    // <READ>
    readResource();             // reading is performed
    // </READ>

    down(&readCountAccess);        // request exclusive access to readCount
```

```
    // <EXIT>
    readCount--;                // update count of active readers
    if (readCount == 0)         // if there are no readers left:
        up(&resourceAccess);     // release resource access for all
    // </EXIT>
    up(&readCountAccess);        // release access to readCount
}

```

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。

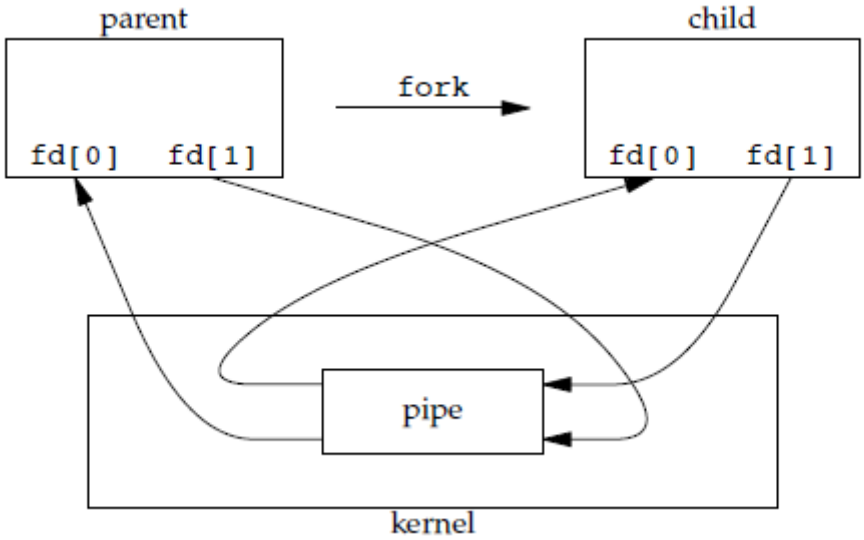
1. 管道

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）；
- 只能在父子进程或者兄弟进程中使用。

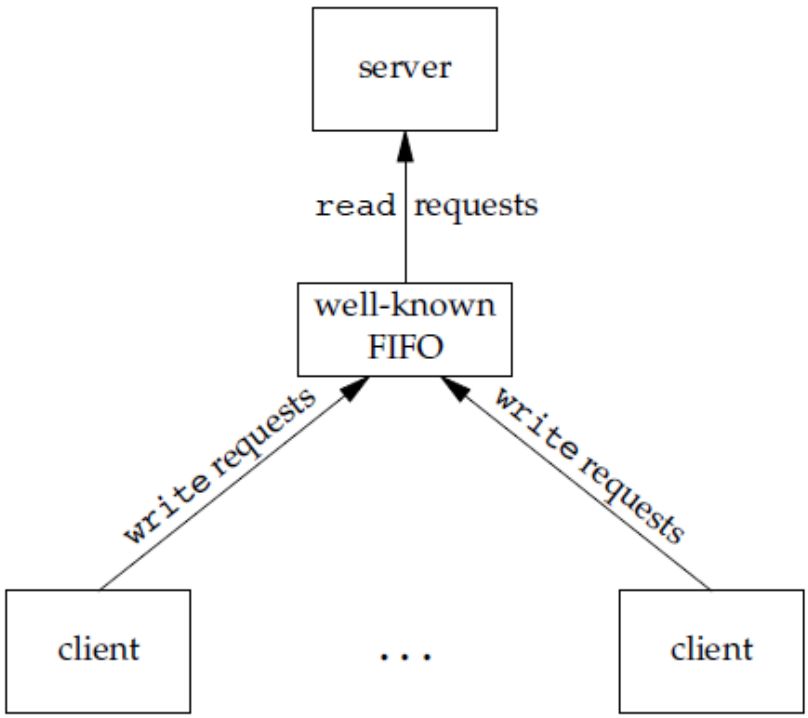


2. FIFO

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。

```
#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

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

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。

第四章.线程

1.进程与线程

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

进程中的所有线程共享该进程的状态和资源，进程和线程的关系如下图：

从性能上比较，线程具有如下优点：

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

2.线程状态

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

有4种与线程相关的基本操作：

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

线程的生命周期

3.线程分类

线程的实现可以分为两大类：

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

3.1 用户级线程

在用户级线程中，进程和线程的状态可能有如下转换：

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

在前两种情况中，当内核把控制切换回进程B时，线程2会恢复执行

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

用户级线程的优点

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

用户级线程的缺点

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

3.2 内核级线程

内核能意识到线程的存在

内核级线程的优点

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

内核级线程的缺点

1. 把控制从一个线程转移到用一进程的另一线程时，需要到内核的状态切换

3.3 混合方案

可以混合使用用户级和内核级线程。在混合方案中，同一应用程序中的多个线程可以在多个处理器上并行地运行，某个会引起阻塞的系统调用不会阻塞整个进程。

如果设计正确，该方法将会结合纯粹用户级线程和内核级线程方法的优点，同时克服它们的缺点

第五章.并发

并发相关的术语：

1.互斥

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

- **进程间的资源竞争**：每个进程不影响它所使用的资源，这类资源包括I/O设备、存储器、处理器时间和时钟。首先需要提供互斥要求（比方说，如果不提供对打印机的互斥访问，打印结果会穿插）。实施互斥又产生了两个额外的控制问题：死锁和饥饿
- **进程间通过共享的合作**：进程可能使用并修改共享变量而不涉及其他进程，但却知道其他进程也可能访问同一数据。因此，进程必须合作，以确保共享的数据得到正确管理。由于数据保存在资源中

（设备或存储器），因此再次涉及有关互斥、死锁、饥饿等控制问题，除此之外，还有一个新要求：数据的一致性

- **进程间通过通信的合作**：由于在传递消息的过程中，进程间未共享任何对象，因而这类合作不需要互斥，但是仍然存在死锁和饥饿问题（死锁举例：两个进程可能都被阻塞，每个都在等待来自对方的通信；饥饿举例：P1,P2,P3，P1不断试图与P2，P3通信，P2和P3都试图与P1通信，如果P1和P2不断交换信息，而P3一直被阻塞，等待与P1通信，由于P1一直是活跃的，P3处于饥饿状态）

1.1 互斥的硬件支持

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

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

问题：

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

2) 专用机器指令

- **比较和交换指令**
- **交换指令**

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

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

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

/*交换指令*/
int bolt;
void P(int i)
```

```
{
    int keyi = 1;
    while(true){
        do exchange (&keyi,&bolt);
        while(keyi != 0);
        /*临界区*/
        bolt = 0;
        /*其余部分*/
    }
}

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

优点

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

缺点

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

1.2 互斥的软件支持

软件支持包括操作系统和用于提供并发性的程序设计语言机制，常见如下表：

1）信号量

通常称为计数信号量或一般信号量

可把信号量视为一个具有整数值的变量，在它之上定义三个操作：

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

信号量原语的定义：

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

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

```
        /*阻塞当前进程*/;
    }
}

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

2）二元信号量

二元信号量是一种更特殊的信号量，它的值只能是0或1

可以使用下面3种操作：

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

二元信号量的原语定义：

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

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

void semSignalB(binary_semaphore s)
{
    if(s.queue is empty())
        s.value = one;
    else{
        /*把进程P从等待队列中移除*/;
        /*把进程P插入到就绪队列*/;
    }
}
```

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

使用信号量的互斥（这里是一般信号量，不是二元信号量）

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

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

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

下图为三个进程使用了上述互斥协议后，一种可能的执行顺序：

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

3）互斥量

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

4）管程

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

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

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

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

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

管程的结构如下：

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

5）消息传递

最小操作集：

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

阻塞：

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

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

消息传递实现互斥(消息函数可视为在进程直接传递的一个令牌)：

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

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

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

1.3 经典问题

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

1）生成者/消费者问题

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

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

实现代码：

- **当缓冲无限大时**（二元信号量，对应图5.10；信号量，对应图5.11）
- **当缓冲有限时**（信号量，对应图5.13；管程，对应图5.16；消息传递，对应图5.21）

2）读者/写者问题

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

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

实现代码：

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

2.死锁

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

假设两个进程的资源请求和释放序列如下：

下图是相应的**联合进程图**，显示了进程竞争资源的进展情况：

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

死锁问题中的资源分类

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

资源分配图

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

2.1 死锁的条件

死锁条件：

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

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

2.2 死锁预防

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

- **间接的死锁预防（防止死锁条件1~3）**
  - **预防互斥**：一般来说，不可能禁止
  - **预防占有且等待**：可以要求进程一次性地请求所有需要的资源，并且阻塞进程直到所有请求都同时满足。这种方法在两个方面是低效的：1）为了等待满足其所有请求的资源，进程可能被阻塞很长时间。但实际上只要有一部分资源，就可以继续执行；2）分配的资源有可能有相当长的一段时间不会被使用，且在此期间，这些资源不能被其它进程使用；除此之外，一个进程可能事先并不会知道它所需要的所有资源
  - **预防不可抢占**：有几种方法：1）如果占用某些资源的进程进一步申请资源时被拒，则释放其占用的资源；2）如果一个进程请求当前被另一个进程占有的一个资源，操作系统可以抢占另一个进程，要求它释放资源(方法2只有在任意两个进程优先级不同时，才能预防死锁)；此外，通过预防不可抢占来预防死锁的方法，只有在资源状态可以很容易保存和恢复的情况下才实用
- **直接的死锁预防（防止死锁条件4）**
  - **预防循环等待**：可以通过定义资源类型的线性顺序来预防，如果一个进程已经分配到了R类型的资源，那么它接下来请求的资源只能是那些排在R类型之后的资源；这种方法可能是低效的，会使进程执行速度变慢，并且可能在没有必要的情况下拒绝资源访问

> 都会导致低效的资源使用和低效的进程运行

2.3 死锁避免

死锁避免允许3个必要条件，但通过明智选择，确保永远不会到达死锁点

由于需要对是否会引起死锁进行判断，因此死锁避免需要知道将来的进程资源请求的情况

2种死锁避免的方法：

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

1）进程启动拒绝

一个有n个进程，m种不同类型资源的系统。定义如下向量和矩阵：

从中可以看出以下关系成立：

对于进程n+1，仅当对所有j，以下关系成立时，才启动进程n+1：

2）资源分配拒绝(银行家算法)

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

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

下图为一个安全序列：

下图为一个不安全序列：

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

优点

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

缺点

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

2.4 死锁检测

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

常见死锁检测算法

这种算法的策略是查找一个进程，使得可用资源可以满足该进程的资源请求，然后假设同意这些资源，让该进程运行直到结束，再释放它的所有资源。然后算法再寻找另一个可以满足资源请求的进程

这个算法并不能保证防止死锁，是否死锁要取决于将来同意请求的次序，它所做的一切是确定当前是否存在死锁

恢复

一旦检测到死锁，就需要某种策略以恢复死锁，有下列方法(复杂度递增)：

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

2.5 死锁“预防/避免/检测”总结

2.6 经典问题(哲学家就餐问题)

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

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

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

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

3.UNIX并发机制

UNIX为进程间的通信和同步，提供了下列几种重要的通信机制：

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

- 信号量
- 信号

3.1 管道

管道是一个环形缓冲区，允许两个进程以生产者/消费者的模型进程通信

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

操作系统强制实施互斥，即一次只能有一个进程可以访问管道

两类管道

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

3.2 消息

每个进程都有一个关联的消息队列，功能类似于信箱

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

3.3 共享内存

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

3.4 信号量

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

3.5 信号

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

4.Linux内核并发机制

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

- 原子操作
- 自旋锁
- 信号量
- 屏障

4.1 原子操作

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

- 在单处理器上：线程一旦启动原子操作，则从操作开始到结束的这段时间内，线程不能被中断

- 在多处理器上：原子操作所针对的变量是被锁住的，以免被其他的进程访问，直到原子操作执行完毕

Linux中定义了2种原子操作：

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

Linux原子操作表：

4.2 自旋锁

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

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

自旋锁操作表：

4.3 信号量

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

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

Linux提供3种版本的down操作：

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

信号量操作表：

4.4 屏障

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

对于屏障操作，需要注意2点：

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

barrier()操作是mb()操作的一个轻量版本，它仅仅控制编译器的行为

屏障操作表：

计算机操作系统 - 死锁

- 计算机操作系统 - 死锁

必要条件

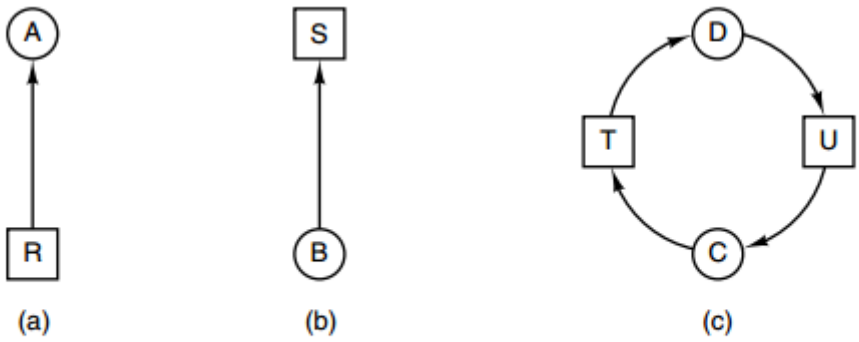


Figure 6-3. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

- 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。

1. 每种类型一个资源的死锁检测

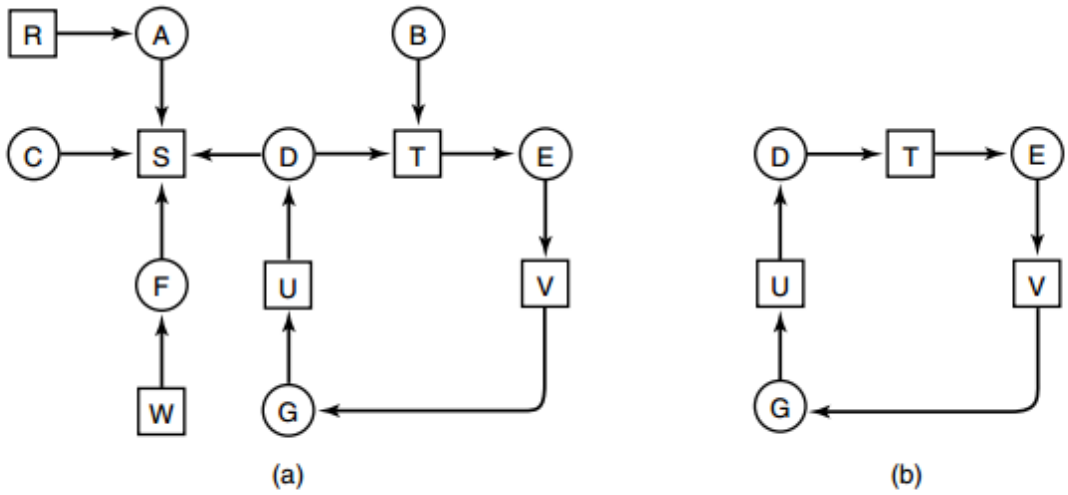


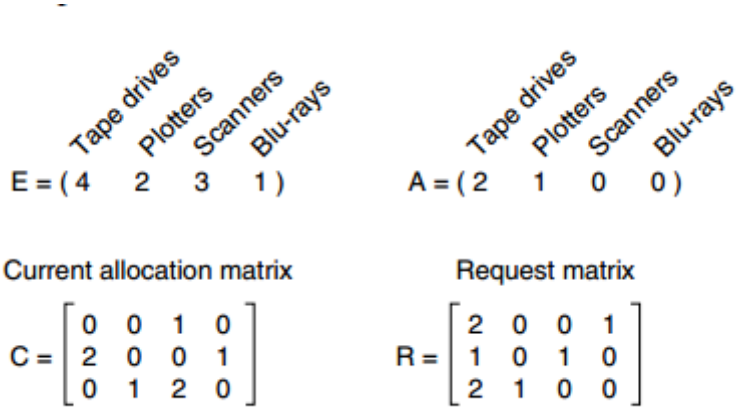
Figure 6-5. (a) A resource graph. (b) A cycle extracted from (a).

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 P_1 和 P_2 所请求的资源都得不到满足，只有进程 P_3 可以，让 P_3 执行，之后释放 P_3 拥有的资源，此时 A = (2 2 2 0)。P_2 可以执行，执行后释放 P_2 拥有的资源，A = (4 2 2 1) 。P_1 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 P_i，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

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

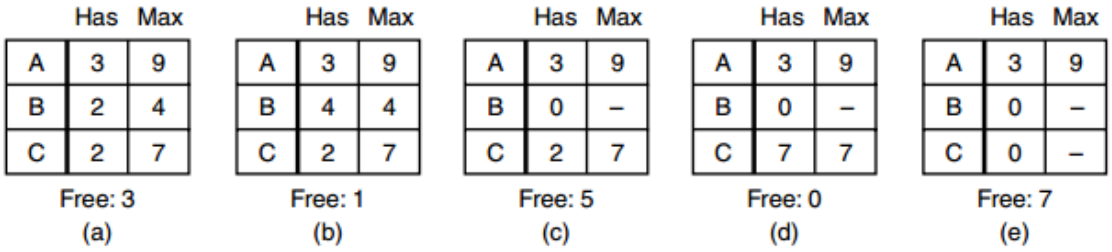


Figure 6-9. Demonstration that the state in (a) is safe.

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 所示的状态时安全的。

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。

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。

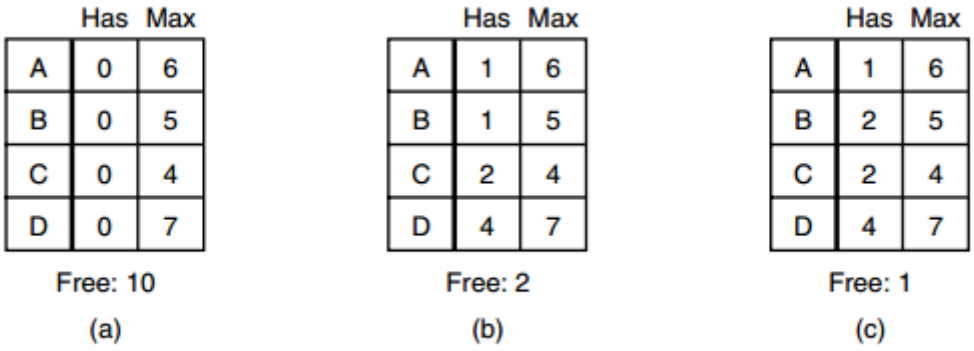


Figure 6-11. Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

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

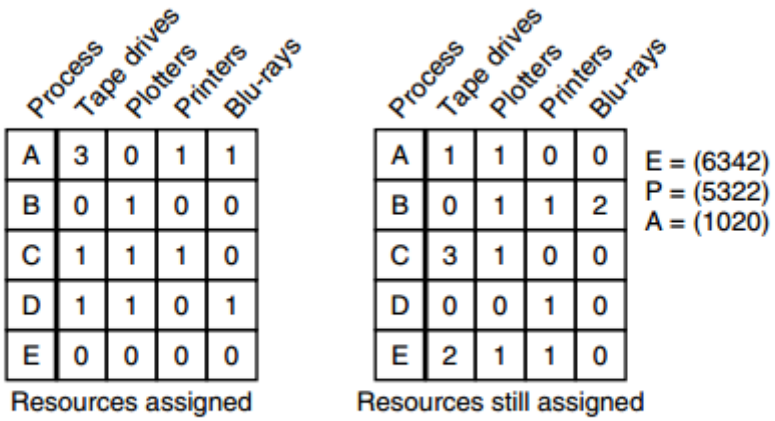


Figure 6-12. The banker's algorithm with multiple resources.

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，则状态时安全的。

如果一个状态不是安全的，需要拒绝进入这个状态。

第六章.内存管理

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

内存管理的需求

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

内存管理中的地址

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

内存管理单元(MMU)：CPU中的一个模块，将虚拟地址转换成实际物理地址

1.内存管理中的数据块

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

2.内存分区

2.1 固定分区

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

- **分区大小相等**
- **分区大小不等**

放置策略：

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

> 存在内部碎片；活动进程数固定

2.2 动态分区

并不进行预先分区，**在每次需要为进程分配时动态划分**

外部碎片（随着时间推移，内存中产生了越来越多”空洞“）：

可以使用压缩解决外部碎片，但是非常耗时

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

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

> 维护复杂，且会产生外部碎片

2.3 伙伴系统

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

下图为一个伙伴系统的例子：

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

2.4 分区中的地址转换

逻辑地址->物理地址的转换如下

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

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

3.分页

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

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

3.1 分页中的地址转换

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

4.分段

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

分段同样会产生外部碎片，但是进程被划分成多个小块，因此外部碎片也会很小

4.1 分段中的地址转换

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

如果偏移大于段的长度，则这个地址无效

5.内存安全

5.1 缓冲区溢出

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

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

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

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

> 缓冲区溢出攻击是最普遍和最具危害性的计算机安全攻击类型之一

5.2 预防缓冲区溢出

广义上分为两类：

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

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

第七章.虚拟内存

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

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

不必将程序完全载入即可运行使得程序可以比实际内存更大

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

1. 分页

1.1 页表

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

1.2 一级分页系统中的地址转换

一级分页系统中的虚拟地址和页表项：

地址转换：

1.3 两级分页系统中的地址转换

两级页表结构（假设页大小为4KB，每个页表项大小为4B）：

地址转换：

1.4 倒排页表

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

一种替代方法是使用一个倒排页表，其机构如下：

页表结构之所以称为"倒排"，是因为它使用页框号而非虚拟页号来索引页表项

1.5 转换检测缓冲区(TLB)

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

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

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

2. 分段

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

2.1 分段系统中的地址转换

2.2 保护和共享

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

3. 段页式

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

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

可以将分页和分段结合，即段页式

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

3.1 段页式系统中的地址转换

4. 内存管理中的相关策略

操作系统的内存管理设计取决于三个基本方面的选择：

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

4.1 读取策略

读取策略确定一个页何时取入内存

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

4.2 放置策略

放置策略决定一个进程块驻留在实存中的什么地方

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

4.3 置换策略

置换策略决定在必须读取一个新页时，应该置换内存中的哪一页

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

基本置换算法

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

4.4 驻留集管理

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

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

基于上述因素，通常采用两种策略：

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

置换范围

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

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

4.5 清除策略

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

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

比较好的方法是结合页缓冲技术

4.6 加载控制

加载控制决定驻留在内存中的进程数目，称为系统并发度

并发度太低会导致处理器利用率不高，并发度太高会发生系统抖动

内存管理补充和总结

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。

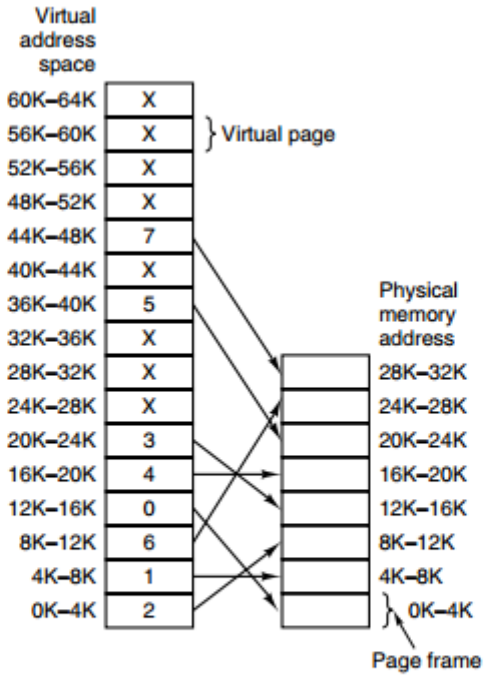


Figure 3-9. The relation between virtual addresses and physical memory addresses is given by the **page table**. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287.

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，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）。

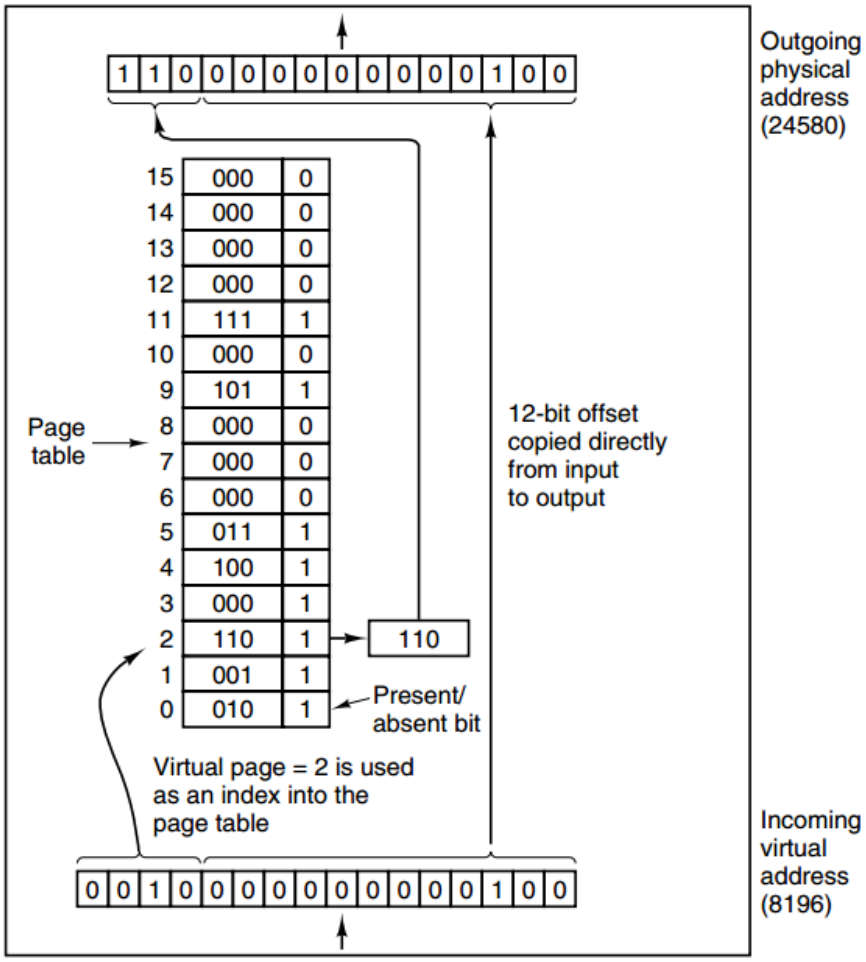


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

页面置换算法（P180）

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。(word)

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）。

此时需要注意的是，就是在置换算法或者策略的时候要注意减少抖动，此时就需要用到如下的几种置换算法了，根据算法的特性和程序的特性来选择合适的替换算法。

出现抖动的原因

- **算法本身（LRU或者FIFO算法）**
- **程序本身（goto无条件跳转，避免动态程序）**

影响F（缺页中断率）的因素：

```
缺页率 = (页面置换次数+分配给该进程的物理块数)/要访问的页面总数
```

计算公式：

$$缺页率(F) = \frac{(页面置换次数 + 分配给该进程的物理块数)}{要访问的页面总数} \tag{1}$$

- **块size大小： 块越大，F越小（但是程序会退化到原来的分区管理）**
- **增加分配给该进程的块数（但是程序会退化到页式管理）**
- **算法**

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：

```
7，0，1，2，0，3，0，4，2，3，0，3，2，1，2，0，1，7，0，1
```

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

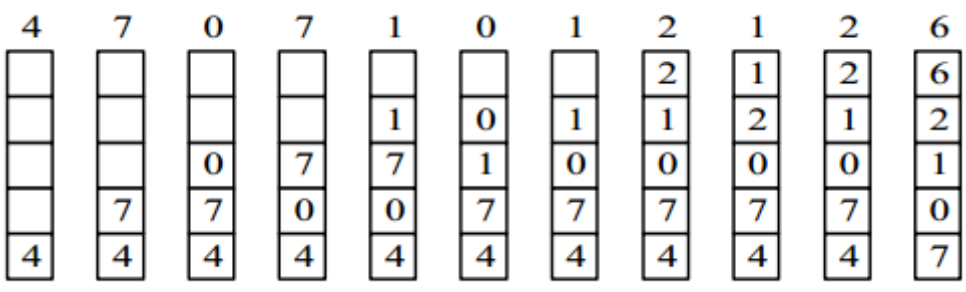
虽然无法知道将来要使用的页面情况，但是可以知道过去使用页面的情况。LRU 将最近最久未使用的页面换出。

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。

> 比如说按照组成原理来讲，一个指令如果指令周期是3个周期（例如LAD取数指令，是RS型指令）
>
> 那么还需要访问页表甚至是硬盘，最少就需要五个周期。

```
4，7，0，7，1，0，1，2，1，2，6
```



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换出，导致缺页率升高。

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。

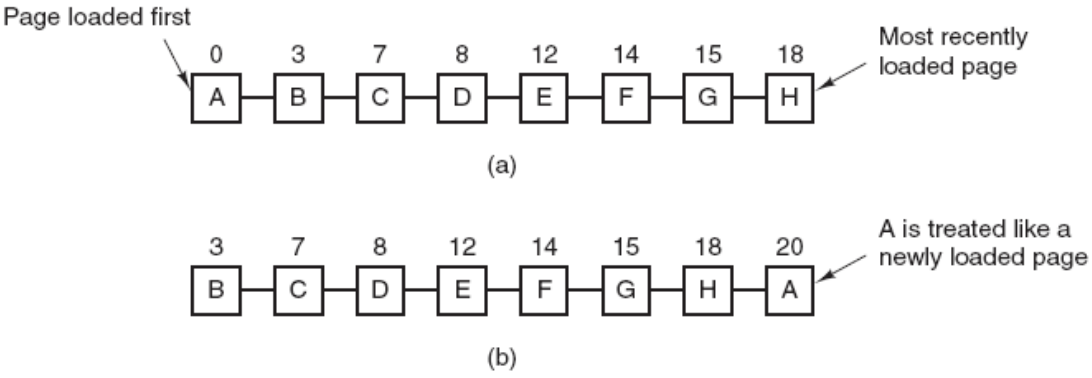


Figure 3-15. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set. The numbers above the pages are their load times.

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。

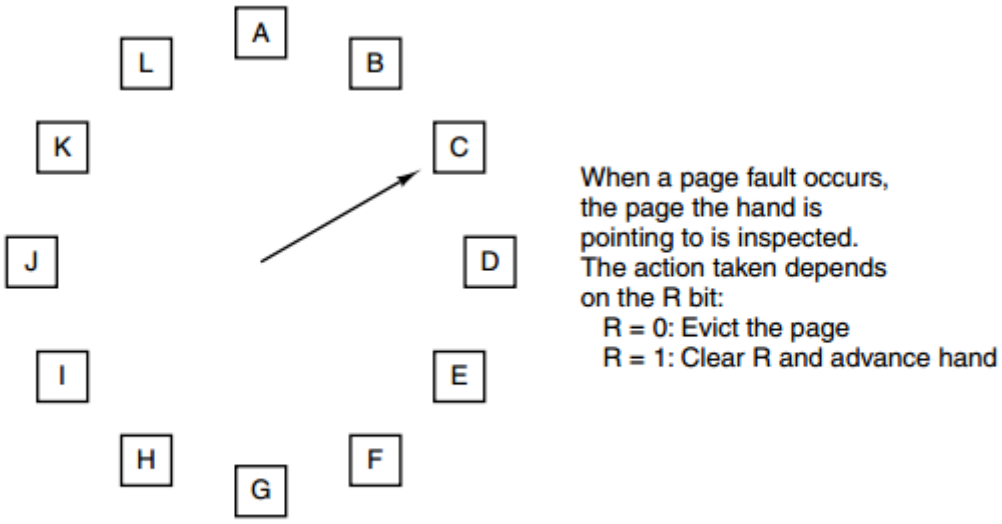
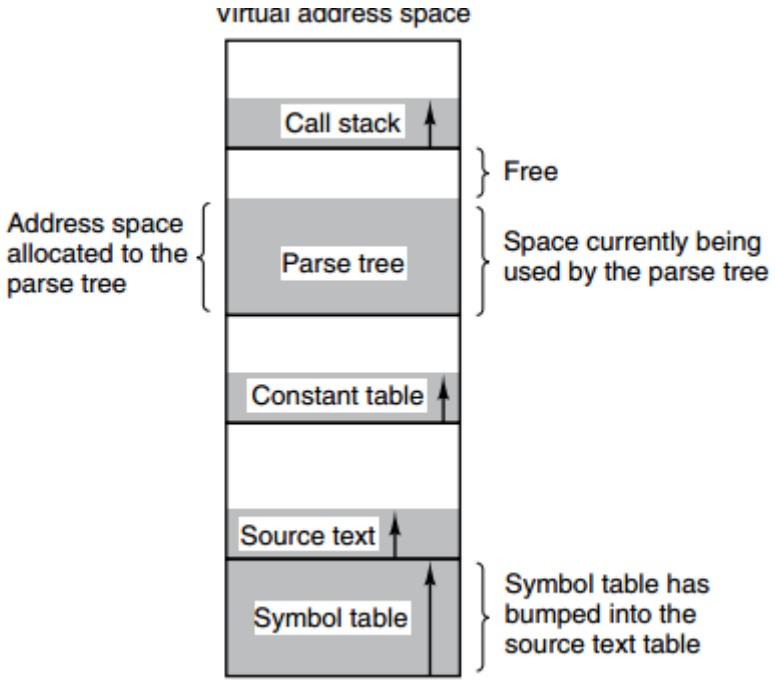


Figure 3-16. The clock page replacement algorithm.

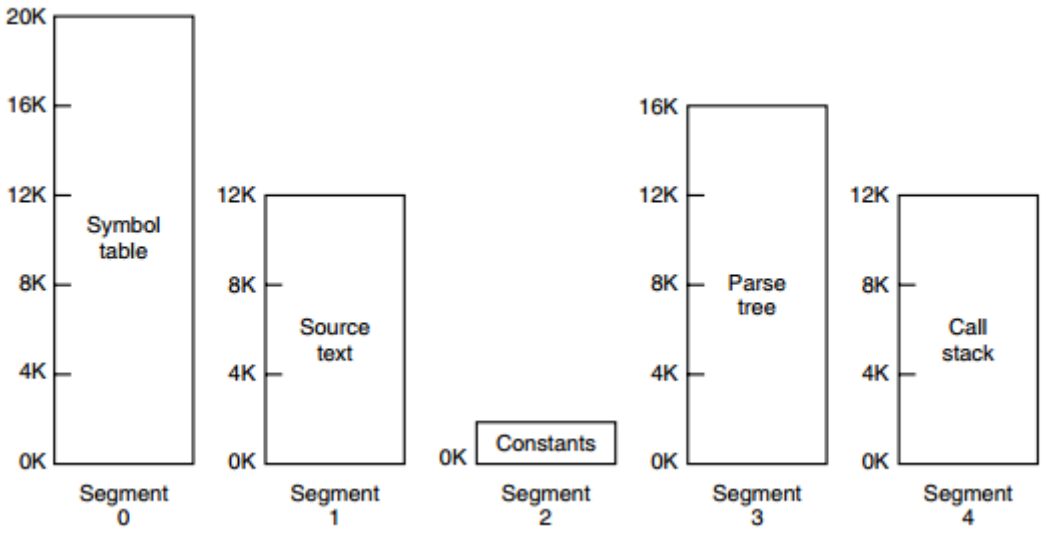
分段

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。



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。

替换算法实验

计算并输出下述各种算法在内存容量为3块、4块下的缺页率。

日期： /

存储管理 内存为3块，4块不能页号

① T270：

链表：段存页表

T270(块=1024)，分配3块

(736，1840，2300，1860，3210，2470，4200，2760)

计算页号：736/1024=0 ………

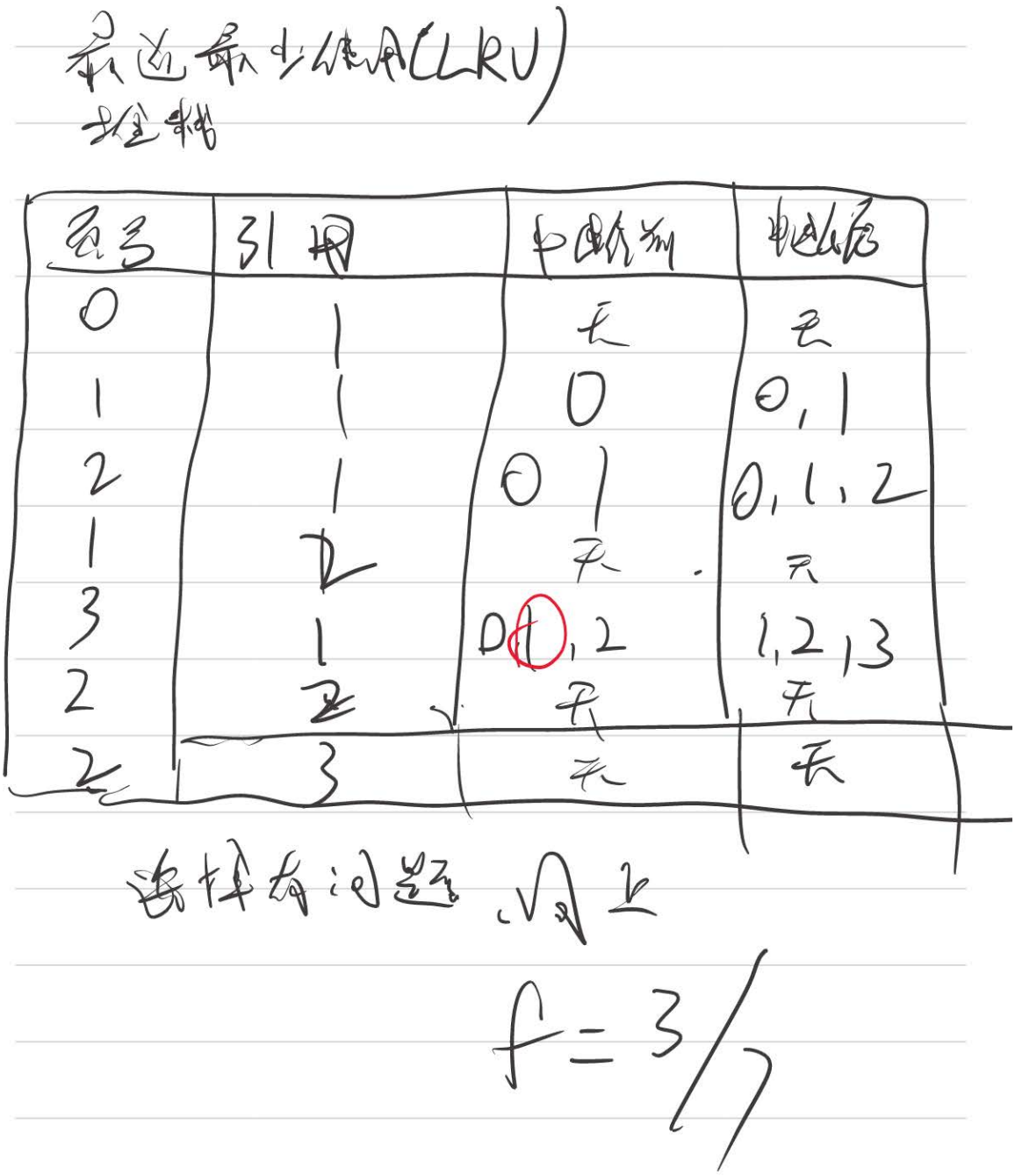
0 1 2 1 3 2 2

	页号	中断前	中断后
1	0	无	无
2	1	0	0，1
3	2	0，1	0，1，2
4	1	无	无
5	3	0，1，2	1，2，3
6	2	无	无
7	2	无	无

中断率

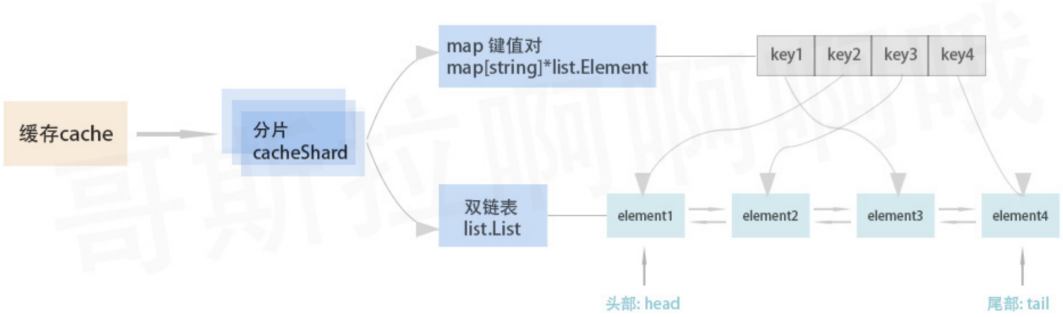
f=3/7

日期: /



① 先进先出的算法（FIFO） 要求用数组或链表方法实现

FIFO缓存淘汰算法的实现(Go语言实现)



如上图示，实现 fifo算法 的缓存架构图：

fifo 算法是淘汰缓存中最早添加的记录,即一个数据最先进入缓存，那么也应该最先被删除掉(队列先进先出)。
算法的实现比较简单：创建一个队列（一般通过双链表实现），新增记录添加到队首，淘汰队尾记录

1. map 用来存储键值对。这是实现缓存最简单直接的数据结构，因为它的查找记录和增加记录时间复杂度都是 O(1)
2. list.List 是go标准库提供的双链表。

> 通过这个双链表存放具体的值，移动任意记录到队首的时间复杂度都是 O(1)，
> 在队首增加记录的时间复杂度是 O(1)，删除任意一条记录的时间复杂度是 O(1)

FIFO代码实现如下：

需要用到go_package中的list包，介绍：

LISt概述¶

包列表实现了一个双向链表。

遍历一个列表（其中 l 是一个 *List）：

```
if  e := l.Front(); e != nil e = e.Next() {
    // 用 e.Value 做一些事情
}
```

还用到了runtime包，包的介绍：

> Package runtime contains operations that interact with Go's runtime system, such as functions to control goroutines. It also includes the low-level type information used by the reflect package; see reflect's documentation for the programmable interface to the run-time type system.
>
> 包运行时包含与Go运行时系统交互的操作，比如控制goroutines的函数。它还包括反射包使用的低级类型信息;有关运行时类型系统的可编程接口，请参阅reflect的文档。

runtime 调度器是个非常有用的东西，关于 runtime 包几个方法:

- **NumCPU**：返回当前系统的 CPU 核数量
- **GOMAXPROCS**：设置最大的可同时使用的 CPU 核数
  通过runtime.GOMAXPROCS函数，应用程序何以在运行期间设置运行时系统中得P最大数量。但这会引起“Stop the World”。所以，应在应用程序最早的调用。并且最好是在运行Go程序之前设置

好操作程序的环境变量GOMAXPROCS，而不是在程序中调用runtime.GOMAXPROCS函数。
无论我们传递给函数的整数值是什么值，运行时系统的P最大值总会在1~256之间。

> go1.8后，默认让程序运行在多个核上,可以不用设置了
> go1.8前，还是要设置一下，可以更高效的利益cpu

- **Gosched**：让当前线程让出 cpu 以让其它线程运行,它不会挂起当前线程，因此当前线程未来会继续执行
  这个函数的作用是让当前 goroutine 让出 CPU，当一个 goroutine 发生阻塞，Go 会自动地把与该 goroutine 处于同一系统线程的其他 goroutine 转移到另一个系统线程上去，以使这些 goroutine 不阻塞。
- **Goexit**：退出当前 goroutine(但是defer语句会照常执行)
- **NumGoroutine**：返回正在执行和排队的任务总数
  runtime.NumGoroutine函数在被调用后，会返回系统中的处于特定状态的Goroutine的数量。这里的特指是指Grunnable\Gruning\Gsyscall\Gwaition。处于这些状态的Groutine即被看做是活跃的或者说正在被调度。
  注意：垃圾回收所在Groutine的状态也处于这个范围内的话，也会被纳入该计数器。
- **GOOS**：目标操作系统
- **runtime.GC**：会让运行时系统进行一次强制性的垃圾收集
  1.强制的垃圾回收：不管怎样，都要进行的垃圾回收。2.非强制的垃圾回收：只会在一定条件下进行的垃圾回收（即运行时，系统自上次垃圾回收之后新申请的堆内存的单元（也成为单元增量）达到指定的数值）。
- **GOROOT**：获取goroot目录
- **GOOS**：查看目标操作系统 很多时候，我们会根据平台的不同实现不同的操作，就而已用GOOS了

为我们创建一个FIFO的队列有一个很好的帮助

```go
// TODO: 定义cache接口
type Cache interface {
    // 设置/添加一个缓存，如果key存在，则用新值覆盖旧值
    Set(key string, value interface{})
    // 通过key获取一个缓存值
    Get(key string) interface{}
    // 通过key删除一个缓存值
    Del(key string)
    // 删除 '最无用' 的一个缓存值
    DelOldest()
    // 获取缓存已存在的元素个数
    Len() int
    // 缓存中 元素 已经所占用内存的大小
    UseBytes() int
}

// TODO: 结构体，数组，切片，map,要求实现 Value 接口，该接口只有1个 Len 方法，返回占用内存的字节数
type Value interface {
    Len() int
}

// TODO: 定义fifo结构体
type fifo struct {
    // 缓存最大容量，单位字节
    // groupCache 使用的是最大存放 entry个数
    maxBytes int
```

```
27
28      // 已使用的字节数，只包括值， key不算
29      usedBytes int
30
31      // 双链表
32      ll *list.List
33      // map的key是字符串，value是双链表中对应节点的指针
34      cache map[string]*list.Element
35  }
36
37  // TODO: 定义key,value 结构
38  type entry struct {
39      key   string
40      value interface{}
41  }
42
43  // TODO: 计算出元素占用内存字节数
44  func (e *entry) Len() int {
45      return CalcLen(e.value)
46  }
47
48  // TODO: 计算value占用内存大小
49  func CalcLen(value interface{}) int {
50      var n int
51      switch v := value.(type) {
52      case Value: // 结构体，数组，切片，map,要求实现 Value 接口，该接口只有1个 Len
    方法，返回占用的内存字节数，如果没有实现该接口，则panic
53          n = v.Len()
54      case string:
55          if runtime.GOARCH == "amd64" {
56              n = 16 + len(v)
57          } else {
58              n = 8 + len(v)
59          }
60      case bool, int8, uint8:
61          n = 1
62      case int16, uint16:
63          n = 2
64      case int32, uint32, float32:
65          n = 4
66      case int64, uint64, float64:
67          n = 8
68      case int, uint:
69          if runtime.GOARCH == "amd64" {
70              n = 8
71          } else {
72              n = 4
73          }
74      case complex64:
75          n = 8
76      case complex128:
77          n = 16
78      default:
79          panic(fmt.Sprintf("%T is not implement cache.value", value))
80      }
```

```go

    return n
}

// TODO: 构造函数，创建一个新 Cache，如果 maxBytes 是0，则表示没有容量限制
func NewFifoCache(maxBytes int) Cache {
    return &fifo{
        maxBytes: maxBytes,
        ll:       list.New(),
        cache:    make(map[string]*list.Element),
    }
}

// TODO: 通过 Set 方法往 Cache 头部增加一个元素（如果已经存在，则移到头部，并修改值）
func (f *fifo) Set(key string, value interface{}) {
    if element, ok := f.cache[key]; ok {
        f.ll.MoveToFront(element)
        eVal := element.Value.(*entry)
        f.usedBytes = f.usedBytes - CalcLen(eVal.value) + CalcLen(value) // 更新占用内存大小
        element.Value = value
    } else {
        element := &entry{key, value}
        e := f.ll.PushFront(element) // 头部插入一个元素并返回该元素
        f.cache[key] = e

        f.usedBytes += element.Len()
    }

    // 如果超出内存长度，则删除队首的节点
    for f.maxBytes > 0 && f.maxBytes < f.usedBytes {
        f.DelOldest()
    }
}

// TODO: 获取指定元素
func (f *fifo) Get(key string) interface{} {
    if e, ok := f.cache[key]; ok {
        return e.Value.(*entry).value
    }

    return nil
}

// TODO: 删除指定元素
func (f *fifo) Del(key string) {
    if e, ok := f.cache[key]; ok {
        f.removeElement(e)
    }
}

// TODO: 删除最 '无用' 元素
func (f *fifo) DelOldest() {
    f.removeElement(f.ll.Back())
}
```

```
135
136 // TODO: 删除元素并更新内存占用大小
137 func (f *fifo) removeElement(e *list.Element) {
138     if e == nil {
139         return
140     }
141
142     f.ll.Remove(e)
143     en := e.Value.(*entry)
144     f.usedBytes -= en.Len()
145     delete(f.cache, en.key)
146 }
147
148 // TODO: 缓存中元素个数
149 func (f *fifo) Len() int {
150     return f.ll.Len()
151 }
152
153 // TODO: 缓存池占用内存大小
154 func (f *fifo) UseBytes() int {
155     return f.usedBytes
156 }
```

测试

```
1  测试:
2  func TestFifoCache(t *testing.T) {
3      cache := NewFifoCache(512)
4
5      key := "k1"
6      cache.Set(key, 1)
7      fmt.Printf("cache 元素个数: %d, 占用内存 %d 字节\n\n", cache.Len(),
   cache.UseBytes())
8
9      val := cache.Get(key)
10     fmt.Println(cmp.Equal(val, 1))
11     cache.DelOldest()
12     fmt.Printf("cache 元素个数: %d, 占用内存 %d 字节\n\n", cache.Len(),
   cache.UseBytes())
13 }
14 ------------------------------------------------------------------------
   ------------------------------
15 结果:
16 === RUN   TestFifoCache
17 cache 元素个数: 1, 占用内存 8 字节
18
19 true
20 cache 元素个数: 0, 占用内存 0 字节
21
22 --- PASS: TestFifoCache (0.00s)
23 PASS
```

② 最近最少使用算法（LRU） 要求用计数器或堆栈方法实现

第八章.单处理器调度

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

1.进程调度类型

调度类型和进程状态转换：

长程调度决定哪一个程序可以进入系统中处理，因此控制着系统的并发度

在批处理系统或者操作系统的批处理部分，新提交的作业被发生到磁盘，并保存在一个批处理队列中。在长程调度程序运行的时候，从队列中创建相应的进程。这里涉及两个决策：

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

长程调度执行频率较低，并且仅仅是粗略地决定是否接受新进程及接受哪一个

该章剩余内容主要关注短程调度，即处理器选择一个进程执行时的调度决策。短程调度执行得最频繁，并且精确地决定下一次执行哪个进程

2.调度算法

2.1 短程调度准则

调度算法的设计需要考虑如下方面（以下为从一种维度的划分）：

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

2.2 优先级调度

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

2.3 选择调度策略

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

1）先来先服务(FCFS)：

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

2）轮转(时间片)：

- 抢占
- 以时间片为周期产生时钟中断，切换运行

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

3）最短进程优先(SPN)：

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

4）最短剩余时间(SRT)：

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

5）最高响应比(HRRN)：

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

6）反馈法：

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

调度策略对比总结：

性能比较

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

2.4 调度实例分析

给出如下进程以及到达时间和服务时间：

使用各种调度策略：

第九章.I/O管理与磁盘调度

为了解决兼容性外围设备和OS的兼容性问题，提出了标准化

其二为了实现设备的**即插即用**

1.I/O缓冲

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

两类I/O设备：（buf)

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

> buf中也含有寄存器，用来进行数据交换的，同时存在的是生产者与消费者的问题。

1.1 单缓冲

对于面向块的I/O设备：

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

对于面向流的I/O设备：

单缓冲方案能以每次传送一行的方式或者每次传送一个字节的方式使用

1.2 双缓冲(缓冲交换)

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

1.3 循环缓冲

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

1.4 I/O缓冲的作用

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

2.磁盘调度

2.1 磁盘性能参数

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

2.2 磁盘调度算法

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

1）先进先出(FIFO)

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

2）优先级

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

3）最短服务时间优先(SSTF)

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

3）SCAN

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

4）C-SCAN

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

5）N-step-SCAN(N步扫描)

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

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

2.3 磁盘调度算法比较

假设有一些I/O请求，需问这些磁道：55、58、39、18、90、160、150、38、184

使用不同磁盘调度算法的结果如下：

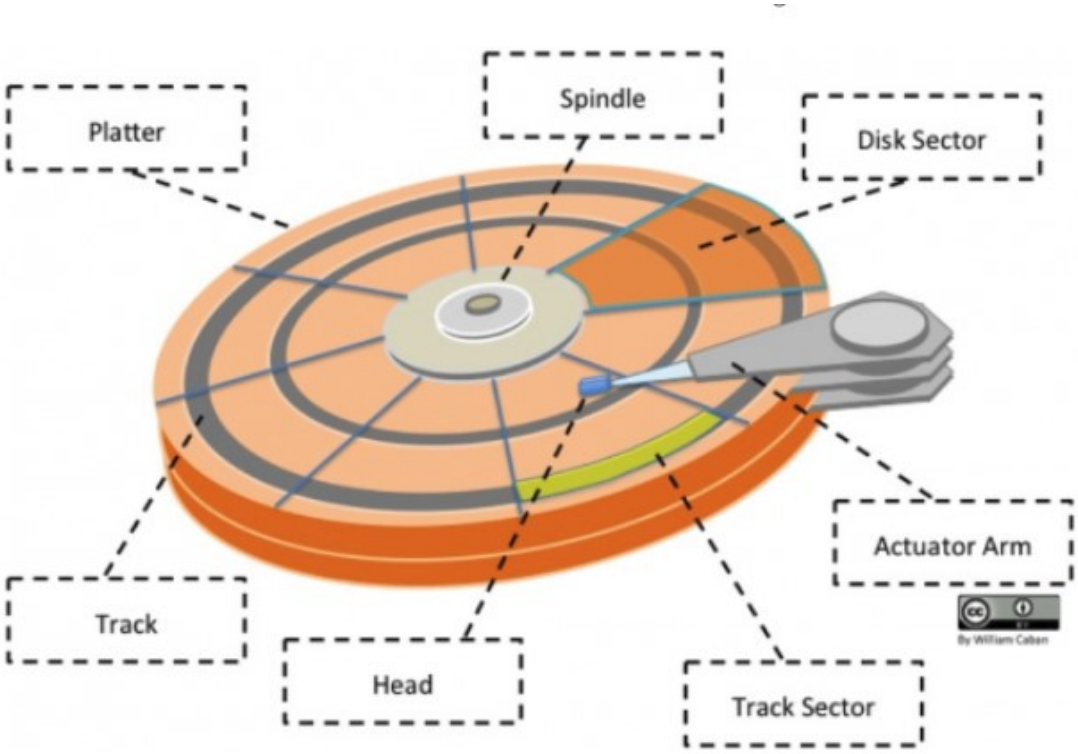
3.磁盘高速缓存

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

计算机操作系统 - 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。

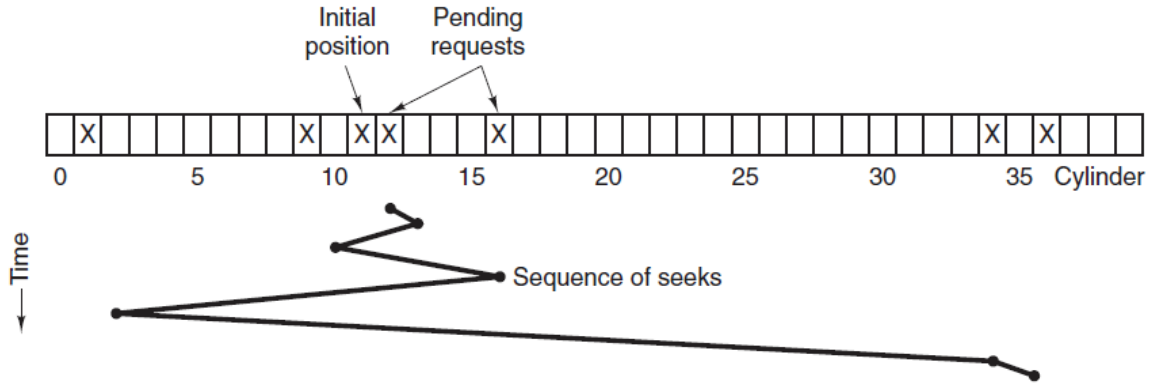
优点是公平和简单。缺点也很明显，因为未对寻道做任何优化，使平均寻道时间可能较长。

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。



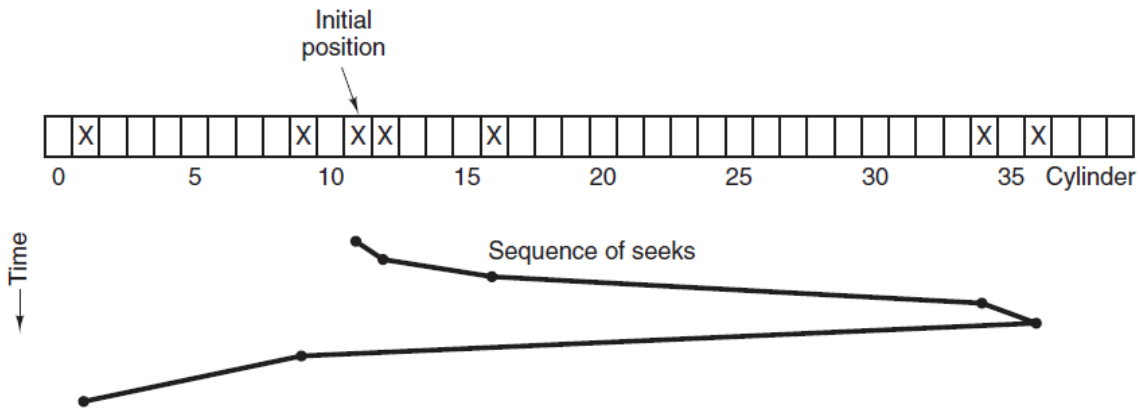
3. 电梯算法

> 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。



文件系统设计

一、文件的系统调用

1．文件描述符（fd）

UNIX的底层输入输出系统调用用一个整数来代表所打开的文件，这就文件描述符。

文件描述符与打开的文件名建立一一对应的关系。

2．Creat/link/unlink系统调用

```
Fd=creat(name,pmode);
```

Name为文件名，整数pmode为文件的许可机制。

Link建立链接， unlink删除链接。

3．Open/close系统调用

```
Fd=open(name,rwmode,[,pmode]);
```

Rwmode表示读写方式的整数，0—只读1—只写2—读写

```
Status=close(fd);
```

4．read/write系统调用

```
 n=read(fd,buffer,size);

 n=write(fd,buffer,size);
```

 buffer是字符指针，存放读/写字节流的地址。

5．随机存取的系统调用lseek和tell

```
newpos=lseek(fd,offset,origin);
```

offset为位移量，origin=0从文件头开始，1从当前位置开始，2从文件尾开始。

Pos=tell(fd);报告当前文件指针的位置。

6．记录的锁定

就是进程在对文件的某个部分进行某种操作期间，为这部分文件内容设立一个“正在使用”的标志，防止其它进程对文件的这个部分进行操作。

```
status=lockf(fd,func,size);
```

 func=0开锁，1锁定，2测试是否锁定，若已锁返回-1，若未锁则锁定，3测试是否锁定，若已锁返回-1，若未锁返回0。

分析下面程序的执行结果：

```
/*************************************************************************
```

```
    > File Name: file.c
    > Author: smile
    > Mail: 3293172751nss@gmail.com
    > Created Time: Thu 26 May 2022 08:57:03 AM PDT
 ************************************************************************/
#include <stdio.h>
#include <unistd.h>
int main() {
    int fd;
    int a[10],i;
    for (i=0;i<10;i++) a[i]=i+1;
    fd=creat("aaa",0755);
    printf("lockf 40 bytes in parent…\n");
    fflush(stdout);
    lockf(fd,1,40);
    printf("…locked.\n");
    fflush(stdout);
    if (fork()==0) {
        printf("Enter child,write 20 bytes in child…\n");
        fflush(stdout);
        write(fd,a,20);
        printf("…writeen.\n");
        fflush(stdout);
        printf("lockf 80 bytes in child…\n");
        fflush(stdout);
        lockf(fd,1,80);
        printf("…locked in child.\n");
        fflush(stdout);
        sleep(2);
        lockf(fd,0,80);
        printf("…child unlocked.\n");
        fflush(stdout);
        exit(0);
    }
    printf("Parent sleep now…\n");
    sleep(0);
    printf("…parent wakeup.\n");
    printf("Parent unlock now…\n");
    fflush(stdout);
    lockf(fd,0,40);
    printf("…Parent unlocked.\n");
    fflush(stdout);
    wait(0);
    printf("Program end.\n");
    return 0;
}
```

编译:

```
root@ubuntu:/c# ./file
lockf 40 bytes in parent…
…locked.
Parent sleep now…
Enter child,write 20 bytes in child…
…writeen.
```

```
lockf 80 bytes in child…
…parent wakeup.
Parent unlock now…
…Parent unlocked.
…locked in child.
…child unlocked.
Program end.
root@ubuntu:/c#
```

二、实验内容

为Linux系统设计一个简单的文件系统。要求做到以下几点：

1. **可以实现下列几条命令；**

- ls 列文件目录

后面的指令都可以在C语言代码中进行实现，结合execl调用终端。（可以实现数据库的登陆）

- create 创建文件 – mkdir
- delete 删除文件 – rm
- read 读文件 –cat or more
- write 写文件 –ipython(write) or echo >> >>

2. **列目录时要列出文件名、存取权限（八进制）、文件长度、时间（三种）；**
3. **源文件可以进行读写保护。**

```
root@ubuntu:/c# whereis ls
ls: /usr/bin/ls /usr/share/man/man1/ls.1.gz
root@ubuntu:/c# cd /usr/bin/
root@ubuntu:/usr/bin# ls -al |grep ls
-rwxr-xr-x  1 root root       47552 Oct 16  2020 alsabat
-rwxr-xr-x  1 root root       85296 Oct 16  2020 alsaloop
-rwxr-xr-x  1 root root       72432 Oct 16  2020 alsamixer
root@ubuntu:/usr/bin# touch mycmd && ll >> mycmd
root@ubuntu:/usr/bin# ls -al |grep mycmd
-rw-r--r--  1 root root      122606 May 27 01:54 mycmd
root@ubuntu:/usr/bin# cat  mycmd |head -n 5
total 619180
drwxr-xr-x  2 root root       69632 May 27 01:54 ./
drwxr-xr-x 14 root root        4096 Feb  9  2021 ../
-rwxr-xr-x  1 root root       59736 Sep  5  2019 [*
-rwxr-xr-x  1 root root          96 Mar  8  2021 2to3-2.7*
root@ubuntu:/usr/bin# echo "myname" >> mycmd
root@ubuntu:/usr/bin# echo "myname" > mycmd
root@ubuntu:/usr/bin# cat mycmd
myname
```