操作系统概念 第九版
部分课后习题解答

Answers for

Operating System Concepts (9th Edition)

(part of)

2020-12

前言

本指南于 2020 软件工程期末考试周复习期间编写。在经历了一个学期操作系统基础课程的学习后，笔者愈发感到本课程的全英文教材和课上的中文授课存在一定程度上的脱节。课程教材《操作习系统概念》在编写上是非常有指导意义的，但此书的课后习题较为开放，知识点也琐碎繁杂，加之全英文描述，即使找到了一些外网环境下的习题解答，看懂答案也要费一番功夫。这使得对于课程学习来说，完成作业变成一件令人头疼的事情。

因此，笔者基于本书的作者在 2012 年出版时发布的部分练习解答（来源于 https://codex.cs.yale.edu/avi/os-book/OS9/practice-exer-dir/index.html）和在 SLADER 网站（https://www.slader.com/textbook/9781118063330-operating-system-concepts-9th-edition/）上寻找到的价值较高的英文习题解答，结合笔者和几位同学的翻译理解，整理出这篇《操作系统概念习题解答》，因学校课时安排较少，仅包括前十三章的部分习题，希望能给予后来学习者一些必要的帮助。

Nickbit

2020 年 12 月

目录

第一章 操作系统引入

实践练习部分 Practice Exercise

1.1 What are the three main purposes of an operating system?
简述操作系统的三个主要功能

答：三个主要功能为：
给用户提供一个在电脑硬件上方便且快捷地运行程序的环境。
根据实际情况合理，高效地分配资源。
作为控制程序，它需要完成：
监控用户程序的执行，防止错误和不当操作发生
管理用户程序和输入/输出设备的运行

1.2 We have stressed the need for an operating system to make efficient use of the computing hardware. When is it appropriate for the operating system to forsake this principle and to "waste" resources? Why is such a system not really wasteful?
我们总是强调操作系统要充分利用硬件的计算性能，但是为什么有的时候操作系统会放弃这一原则尝试去"浪费"资源？尝试给出一个例子，并解释为什么这种"浪费"是有用的

答：渲染 GUI 图形界面。这项工作虽然会"浪费"CPU 周期，但是可以优化用户与系统的的交互，使用户更加容易和操作系统交互，所以这种"浪费"是有用的。

1.3 What is the main difficulty that a programmer must overcome in writing an operating system for a real-time environment?
在编写实时环境下的操作系统时，最大的而且必须克服的困难是什么？

答：最大的困难是如何保证操作系统能在时间限制下工作。如果操作系统不能在一定的时间框架内完成某个任务，可能会导致整个系统的崩溃。所以在编写系统时要保证响应时间不能超过限制

1.4 Keeping in mind the various definitions of operating system, consider whether the operating system should include applications such as web browsers and mail programs. Argue both that it should and that it should not, and support your answers.
根据操作系统的多种定义，你是否认为操作系统应该预装诸如浏览器和邮件收发的流行软件？对应该和不应该的两种看法分别说明理由

答：为什么应该内置流行应用：
- 这些应用大概率能充分调用内核中的功能，从而达到比第三方应用更高的性能
- 方便用户，用户无需再花时间安装对应的软件

为什么不应该内置流行应用：
- 应用不应该是操作系统的一部分，多余的应用会影响操作系统的纯粹性，使操作系统变得臃肿

- 虽然可以通过调用内核中的功能实现的性能提升，但同时也带来了安全隐患
- 容易引发软件垄断等纠纷（参考网景和微软的故事）

1.5 How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) system?

从对单纯地对系统的保护角度来说，内核模式和用户模式的功能有何区别?

答：因为对硬件的访问权限，对中断的处理，和对某些指令的执行只能在内核模式中完成，所以 CPU 在用户模式下只有很小的权限，进而保护了关键资源的安全。

1.6 Which of the following instructions should be privileged?

以下哪些指令应该被视为特权指令?

a. Set value of timer.
b. Read the clock.
c. Clear memory.
d. Issue a trap instruction.
e. Turn off interrupts.
f. Modify entries in device-status table.
g. Switch from user to kernel mode.
h. Access I/O device.

答：a, c, e, f, h

1.7 Some early computers protected the operating system by placing it in a memory partition that could not be modified by either the user job or the operating system itself. Describe two difficulties that you think could arise with such a scheme.

一些早期的电脑为了保护操作系统，将系统存放于一块不论用户和操作系统本身都不能修改的内存分区中，简述这样做可能引起的两个问题。

答：

1. 某些敏感数据反而只能储存在未被保护的内存中，不安全
2. 系统不能进行更新迭代

习题部分 Exercise

1.12 In a multiprogramming and time-sharing environment, several users share the system simultaneously. This situation can result in various security problems.

a. What are two such problems?

b. Can we ensure the same degree of security in a time-shared machine as in a dedicated machine? Explain your answer.

在一个多用户的分时系统中，多个用户同时分享系统的使用，这种情况下可能会引发一些安全问题

a、简述其中两个可能发生的问题

b、可以做到在分时系统的计算机中达到与专用服务器一样的安全性吗？说明理由。

答：

a. 两个问题可能是：

1. 不同的用户可能同时尝试访问、修改内存同一位置的数据，造成冲突
2. 若资源不能在不同用户之间合理分配，会造成浪费甚至崩溃

b. 从某种程度上可以。我们可以使用虚拟机等技术，让分时系统机器模拟专用机器工作，便可达到同一等级的安全性。但同时，任何人设计的安全系统都有可能被人为破坏，所以并没有绝对的安全。

1.13 The issue of resource utilization shows up in different forms in different types of operating systems. List what resources must be managed carefully in the following settings:

a. Mainframe or minicomputer systems

b. Workstations connected to servers

c. Mobile computer

不同的操作系统资源利用率不同，对于下列系统来说，哪些资源是重要的、应该被严格管理的？

a. 主机或小型计算机

b. 需要连接至服务器的工作站

c. 便携式可移动计算机

答：

a. CPU，内存，外存

b. CPU，内存，外存，网络带宽

c. 功耗和用户体验

1.14 Under what circumstances would a user be better off using a time-sharing system than a PC or a single-user workstation?

在什么情况下用户更应该使用分时系统而不是 PC 或者个人电脑工作站？

答：当用户的预算有限，而且（尤其是）需要强大算力的时候。

1.19 What is the purpose of interrupts? How does an interrupt differ from a trap? Can traps be generated intentionally by a user program? If so, for what purpose?

中断的目的是什么？中断和陷阱的区别是什么？用户程序可以有意产生陷阱吗？如果可以，为什么要这么做？

答：中断是由软件或硬件向 CPU 发送的一个优先级非常高的指令。中断指令会中断当前的进程，然后执行一些新的指令。Trap（陷阱）是一种特殊的中断，由正在运行的软件遇到错误或者需要协助的时候发出。用户程序可以故意生成 trap 信号，比如在 debug 时来调用操作系统的功能判断运行时产生的错误。

1.21 Some computer systems do not provide a privileged mode of operation in hardware. Is

it possible to construct a secure operating system for these computer systems? Give arguments both that it is and that it is not possible.

一些计算机系统不提供能直接操作硬件的特权模式，可以为这种计算机实现一个完全安全的操作系统吗？对于可以和不可以两种看法，发表一些看法。

答：

可以，因为：

- 如果将特权指令全部删去，可以通过牺牲一定的功能性来换取安全性
- 如果所有的用户程序都用高级语言编写，运行时实时编译，可以在编译时对这些高级语言代码进行安全检查，确保有害代码不被执行

不可以，因为：

- 如果不牺牲功能性，任由用户程序调用特权指令，会对系统安全构成很大的威胁
- 任何人工设置的安全系统都可能被人为破坏

第二章 操作系统结构

实践练习部分 Practice Exercise

2.1 What is the purpose of system calls?

系统调用的功能是什么?

答：系统调用允许用户态的进程向操作系统请求服务。

2.5 What is the purpose of the command interpreter? Why is it usually separate from the kernel?

命令解释程序的作用是什么？为什么它要与内核程序分离?

答：命令解释程序从用户或文件读取并执行命令，通常需要将它们转化成若干个系统调用。它通常不属于内核（kernel）是因为命令解释程序是可以被修改的。

2.8 What is the main advantage of the layered approach to system design? What are the disadvantages of using the layered approach?

分层操作系统的设计有什么优点？相对的，其有什么缺点?

答：分层设计的优点:

1、系统更容易查错（debug）和修改，因为所有的变更都只影响到整个系统有限的一部分而不会涉及所有方面。同样其方便于系统的更新和优化。

2、分层系统相对于不分层的系统更加安全。

分层系统的缺点:

1、无法直接访问高层的数据，同样，软件想要直接访问硬件层级的数据也不如非分层系统方便。

2、分层系统的层数如果过多，容易影响操作系统的效率。

习题部分 Exercise

2.12 The services and functions provided by an operating system can be divided into two main categories. Briefly describe the two categories, and discuss how they differ.

操作系统提供的服务和功能可以分为两类，简述这两个分类，并说明它们的不同点

答：

第一类：加强并行进程间的保护。进程只能访问那些与它们的地址有关的内存位置。同时，进程不允许损坏与其他用户有关的文件，并且不允许在没有操作系统介入的情况下访问硬件设备。

第二类：提供不被底层硬件直接支持的新功能。比如虚拟内存和文件系统。

2.13 Describe three general methods for passing parameters to the operating system.

描述向操作系统传递参数的三个主要方法。

答：

a. 直接通过寄存器传参
b. 通过寄存器传递参数所处位置的地址
c. 参数可以被压入栈（stack）上，然后被操作系统从栈上取出（popped off）。

2.16 What are the advantages and disadvantages of using the same system- call interface for manipulating both files and devices?

采用系统调用接口来同时操作文件和设备的优点和缺点是什么?

答：

优点：

1、方便设备以驱动的形式连接到操作系统：所有设备都能被当做是一个文件系统中的文件被访问。鉴于大多数内核都通过这个文件接口控制设备，通过将设备独有的代码植入这个抽象的文件系统，驱动一个新的设备相对更简单。（it is relatively easy to add a new device driver by implementing the hardware-specific code to support this abstract file interface）

2、连接设备和访问文件时都使用系统调用接口可以使两种方式都使用相似的代码。因此，这能促进那些用 同样的方式来访问硬件设备和文件的用户程序代码，和那些 能被良好的 API 适配的设备驱动代码的发展。

缺点：在文件访问 API 的环境中捕获到某个设备的功能可能更加困难，因此导致性能或功能的损失。这个问题可以通过使用提供用来调用设备上的操作的本地通用接口来部分解决。

2.17 Would it be possible for the user to develop a new command interpreter using the system-call interface provided by the operating system?

用户可以使用操作系统提供的系统调用来开发一个自己的命令解释器吗？

答：
有可能。命令解释程序允许用户创建和管理进程，并决定他们之间的交流方式（比如说通过管道和文件）。因为所有的这些功能都是能被用户层程序通过系统调用使用的，所以用户理应可以开发一个新的命令解释程序。

2.19 Why is the separation of mechanism and policy desirable?
为什么操作系统的机制和策略需要分离编写？

答：
机制（mechanism）和策略（policy）必须要分开才能保证系统可以轻松地被修改。没有两个系统的安装实例（installation）是一样的，所以在某些安装实例下可能希望对操作系统微调来满足需要。机制和策略分开的情况下，可以在机制不变的情况下随意更改策略。这是一种更灵活的系统设计。

2.21 What is the main advantage of the microkernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?

采用微内核的系统架构有什么优点？在微内核架构下用户程序和系统软件是如何交流的？微内核架构有没有什么缺点？

答：
1 微内核设计的优势：
(a) 加入新服务时不需要修改内核（kernel）
(b) 因为更多的操作将在用户态而不是内核态下完成，所以更安全
(c) 更简单的内核设计和功能通常会使系统更稳定
2 用户程序和系统服务之间的通信可以通过消息传递（message passing）实现。这些消息通常由操作系统传递。
3 微内核设计的主要缺点：
进程间通讯会有间接损耗、为了实现用户进程和系统服务间的互动，需要频繁调用操作系统的消息传递功能，从而影响计算机的性能。

2.22 What are the advantages of using loadable kernel modules?
使用可装载内核模块的优点有哪些？

答：因为在设计系统时很难预测到未来需要什么功能。所以载入式内核模组的优势就在于可以在内核运行的过程中加入或移除某些功能。不需要重新编译或重启内核。

第三章 进程管理

实践练习部分 Practice Exercise

3.1 Using the program shown in Figure 3.30, explain what the output will be at LINE A.

阅读下图的代码程序，解释 LINE A 行会产生什么输出

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
pid_t pid;

   pid = fork();

   if (pid == 0) { /* child process */
      value += 15;
      return 0;
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE A */
      return 0;
   }
}
```

Figure 3.30 What output will be at Line A?

答：输出是 5. 因为子进程有自己的一份拷贝，所以尽管子进程里的 value 是 20，当回到母进程后，输出的 value 的值仍是母进程中 value 的值——5.

3.2 Including the initial parent process, how many processes are created by the program shown in Figure 3.31?

下图程序中一共创建了几个进程（包括初始主程序的父进程）？

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
```

Figure 3.31 How many processes are created?

答：一共有 8 个进程被创建。

3.5 When a process creates a new process using the fork() operation, which of the following states is shared between the parent process and the child process?

当一个进程使用 fork()创建子进程时，以下哪一个是父子进程间共享的部分？

a. Stack 栈

b. Heap 堆

c. Shared memory segments 共享内存段

答：只有 c. Shared memory segments（共享内存段）被共享，其它部分父子进程都有各自的拷贝。

习题部分 Exercise

3.8 Describe the differences among short-term, medium-term, and long- term scheduling.

辨析短期调度，中期调度和长期调度的区别。

答：

- 短期调度：使用 CPU 调度程序，调度程序选择已经准备好被执行的进程，把它们分配给 CPU
- 中期调度：一般在在分时系统中作中介调度层。使用交换（swapping）策略将运行中的程序占用的内存暂时存放到别的储存介质中，以便之后继续
- 长期调度：使用进程调度程序，决定哪些进程将被写入内存中执行

3.9 Describe the actions taken by a kernel to context-switch between processes.

描述在进程上下文切换时内核需要执行的操作。

答：一般来讲，操作系统需要将当前正在执行的进程的状态储存到对应进程的 PCB 中，然后从内存中读取下一个要被执行的指令的进程，恢复其 PCB 的状态。保存进程状态通常除了包括内存中的信息，也包括所有 CPU 寄存器里的值。上下文切换（context switch）也通常要执行很多基于处理器架构的操作，包括刷新数据和指令缓存（cache）。

3.11 Explain the role of the init process on UNIX and Linux systems in regard to process termination.

解释 UNIX 和 Linux 系统中 init 进程在处理其他进程终止上的作用。

答：如果父进程没有调用 wait()，子进程却已经终止了，子进程就变成了孤儿进程（orphan）。在 Linux 和 UNIX 系统中，会将 init 进程（也就是根进程）设置成孤儿进程的父进程，init 进程会阶段性地调用 wait()，以便收集任何进程的退出状态，并释放孤儿进程标识符(PID)和进程表条目(PCB)。

3.12 Including the initial parent process, how many processes are created by the program shown in Figure 3.32?

下图程序中一共创建了几个进程（包括初始主程序的父进程）？

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    int i;

    for (i = 0; i < 4; i++)
      fork();

    return 0;
}
```

Figure 3.32 How many processes are created?

答：一共有 16 个进程被创建。

3.13 Explain the circumstances under which the line of code marked printf("LINE J") in Figure 3.33 will be reached.

在什么情况下，下图程序中 printf("LINE J")的一行才会被执行？

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

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

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
       printf("LINE J");
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Figure 3.33 When will LINE J be reached?

答：execlp()系统调用会用 exec()的参数指向的程序取代当前程序所占用的内存空间。如果成功调用 execlp ()，一个新的程序将会被执行，而因为 execlp ()不会返回（return），printf("Line J");将不会执行。然而，如果 execlp ()调用失败，那 exec()会将控制权返回给原函数，进而执行 printf("Line J");，Line J 将被打印。

3.14 Using the program in Figure 3.34, identify the values of pid at lines A, B, C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

根据下图的程序，确定在 A,B,C,D 四个标记行中打印出的进程 PID 是多少（假设父进程的 PID 是 2600，其对应子进程的 PID 为 2603）

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid, pid1;

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

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       pid1 = getpid();
       printf("child: pid = %d",pid); /* A */
       printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* parent process */
       pid1 = getpid();
       printf("parent: pid = %d",pid); /* C */
       printf("parent: pid1 = %d",pid1); /* D */
       wait(NULL);
    }

    return 0;
}
```

Figure 3.34 What are the pid values?

答：PID 值分别为：A=0, B=2603, C=2603, D=2600

第四章 线程管理

实践练习部分 Practice Exercise

4.1 Provide two programming examples in which multithreading provides better performance than a single-threaded solution.

举出两个使用多线程比单线程效率更高的程序的例子。

答：

1. 需要同时满足多个功能的程序，如支持拼写检查的文本编辑器，支持语法高亮的 IDE 等
2. 需要对多个用户同时进行的操作做出回应的程序，如服务端程序。

4.2 What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?

用户级线程和内核级线程的不同的是什么？在什么情况下前者比后者更好，反过来呢？

答：

用户级线程和内核级线程的主要区别为：

a. 用户级线程对内核不可见，内核只监控内核级线程；而内核级线程之间按可以互相通信
b. 在多对一或多对多映射的系统中，用户线程被线程库（thread library）调度（schedule），而内核则负责调度内核线程
c. 内核线程不一定属于一个进程，但用户线程一定是某个进程的一部分。通常来说，内核线程比用户线程更难维护（maintain），因为它们必须要用内核数据结构进行表示

在线程需要处理系统内核的进程时，使用内核级线程更好；而当需要节省空间和 CPU 开销时，使用用户级线程更佳。

4.3 Describe the actions taken by a kernel to context-switch between kernel- level threads.

描述内核在进行内核级别的线程切换时的主要步骤。

答：内核线程间的上下文切换通常需要先保存即将被移出 CPU 寄存器的值，然后再将需要用到的新值载入 CPU 寄存器中。

4.4 What resources are used when a thread is created? How do they differ from those used when a process is created?

一个线程被创建时需要使用哪些资源？这与创建进程的时候有什么不同？

答：线程是进程的一部分，因此通常来说，创建线程比创建进程消耗资源更少。创建进程需要分配一个进程控制块（PCB），一个相对较大的数据结构。PCB 中包括内存映射（memory map），文件列表，和环境变量。分配和管理内存映射将消耗大量时间。而创建线程只需要分配一个较小的数据结构来存储寄存器中的值，堆栈，和优先级。

4.5 Assume that an operating system maps user-level threads to the kernel using the many-to-many model and that the mapping is done through LWPs. Furthermore, the system allows developers to create real-time threads for use in real-time systems. Is it necessary to bind a real-time thread to an LWP? Explain.

假设有一个操作系统采用“多对多”的模型来映射用户线程和内核线程，其线程实现采用的是轻量级进程（LWP），并且该系统允许开发者在实时系统里创建实时线程。那么，是否有必要将实时线程于轻量级线程进行一一绑定？给出理由。

答：是的。对于实时应用程序来说，对时间的把握（timing）是至关重要的。如果一个线程是实时的但没有被 LWP 控制，那这个线程可能要等待与一个 LWP 连接上了之后才能运行。试想，如果一个连接上 LWP 的实时线程在运行遇到阻塞（block）了，而与它连接的 LWP 又被调配给了另一个线程，那么当此进程再次被调度等待执行时，它就得先等待与一

个 LWP 建立连接。如果将 LWP 和实时线程绑定，就可以保证避免重新分配 LWP 可能带来的延迟。

习题部分 Exercise

4.6 Provide two programming examples in which multithreading does ***not*** provide better performance than a single-threaded solution.

举出两个在编程上多线程的效率**不如**单线程的例子。

答：

a. 一个需要递增计数（也就是含有较多形如 var++语句的）的程序
b. 一个需要监控某个特定工作空间（working space）的程序，或者是一个需要不断接收来自同一用户（如 shell 命令解释器）的指令的程序。

4.7 Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

对于单核操作系统来说，什么情况下使用多个内核线程比使用单个内核线程效率更高?

答：当一个内核线程遇到缺页错误（page fault）时，可以切换执行另一个内核线程来利用中间的空闲时间。相反，一个单线程进程则无法在出现缺页错误时执行有效工作。因此，当一个程序有可能频繁出现缺页错误，和需要等待系统事件时，多线程解决方案哪怕在单处理器的系统上也有更好的表现。

4.8 Which of the following components of program state are shared across threads in a multithreaded process?

下列哪些程序的组成部分在多线程的进程中是被共享的?

a. Register values 寄存器值
b. Heap memory 堆内存
c. Global variables 全局变量
d. Stack memory 栈内存

答：b, c

4.9 Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.

在多核处理器的系统上，使用多个用户线程能使得系统的性能表现比在单核处理器上更高吗？解释原因。

答：拥有多个用户级线程的多线程系统无法同时使用多处理器系统中的多个处理器——这些线程都将被操作系统视作一个进程，不会被操作系统分配到不同的处理器上。因此，在多处理器系统中运行多个用户级线程并不会带来性能提升。

4.11 Is it possible to have concurrency but not parallelism? Explain.

有可能实现程序并发但不并行吗？解释原因。

答：
有可能。并发（concurrency）指的是通过调度进程或线程，让多个程序可以宏观上同时运行，而并行（parallelism）则要求线程或进程在微观上也是同时执行的，这只能在多核处理器上实现。因此在单核处理器上就可以实现并发但不并行的程序
解释：*可以理解为：并发是很多个人在同一条跑道（一个 CPU 核心）上跑步，虽然看起来是很多人同时在跑步，但总有一个人跑在最前面；而并行是很多人同时在不同的跑道上跑步，那么他们就可以并肩而行。*

4.15 Consider the following code segment:

```
pid t pid;

pid = fork();
if (pid == 0) { /* child process */

fork();

thread create( . . .);

}

fork();
```

阅读上面的代码块，然后回答：

a. How many unique processes are created? 有多少个不同的进程被创建？
b. How many unique threads are created? 有多少个不同的线程被创建？

答：

a. 6 个进程
b. 2 个线程

4.16 As described in Section 4.7.2, Linux does not distinguish between processes and threads. Instead, Linux treats both in the same way, allowing a task to be more akin to a process or a thread depending on the set of flags passed to the clone() system call. However, other operating systems, such as Windows, treat processes and threads differently. Typically, such systems use a notation in which the data structure for a process contains pointers to the separate threads belonging to the process. Contrast these two approaches for modeling processes and threads within the kernel.

参考书本 4.7.2 节的描述，Linux 系统并不区分线程和进程，相反，Linux 将两者统一对待。一个任务是更接近于进程还是更接近于线程却决于创建任务的时候使用 clone()系统调用时的参数。然而，其他操作系统诸如 Windows，就将线程和进程区别对待。一般来说，采用这种方式时，将线程的指针加入其所属进程的数据结构中。对于这两种方法，对比一下它们在内核中创建进程和线程的优缺点。

答：一方面，在那些不区分线程和进程的系统中，某些操作系统代码更精简。比如说，调度器可以对线程和进程一视同仁，不用额外的代码在每次调度时检查线程所属的进程。另一方面，这种统一性可能会让实施进程资源限制更难，更不直接——需要额外的代码去识别哪些线程对应着哪个进程。

第五章 进程同步

实践练习部分 Practice Exercise

5.3 What is the meaning of the term ***busy waiting***? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

忙碌等待是什么意思？在操作系统中还有哪些类型的等待？忙碌等待可以被完全避免吗？解释你的答案。

答：忙碌等待（busy waiting）是指一个进程在一直占用 CPU 资源的情况下循环等待某个条件的达成而不释放处理器占用。
除此之外，进程还可以在释放处理器后进入等待：在某种条件中阻塞，并在未来的某个时刻被唤醒。
忙碌等待可以被避免，但是会导致一定的开销——让一个进程休眠并在合适的时候将它唤醒会让内核产生开销。

5.5 Show that, if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated.

解释一下如果 wait() 和 signal()操作不是原子操作，为什么就会违反互斥锁的互斥原则？

答：wait()函数会将信号量递减 1。而信号量属于临界区段（critical section），所以如果 wait()不是原子操作，那么就可以有两个 wait 操作同时进行，那么两个进程会同时进入临界区段，这就违反了互斥原则（mutual exclusion）。signal() 操作的原理类似。

5.6 Illustrate how a binary semaphore can be used to implement mutual exclusion among *n* processes.

如何用二进制信号量来实现 n 个进程之间的互斥？用绘图或伪代码解释。

答：假设 n 个进程共享同一个信号量 mutex，将 mutex 初始化为 1，进程如下编写即可：

```
do {
    wait(mutex);
    /* critical section */
    signal(mutex);
    /* remainder section */
} while (true);
```

习题部分 Exercise

5.10 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synhronization primitives are to be used in user-level programs.

为什么使用取消中断来实现同步的方法对于单核系统不太适用？这里的同步是指在用户级别的程序之间实现同步。

答：

1、如果一个用户级程序被赋予了禁用中断的权限，那么它就可以禁用时钟中断和阻止上下文切换，这样一来就阻止了其它进程的运行，甚至导致重要的系统进程被阻塞而无法处理。用户程序不应该拥有内核程序一样高的优先级。

2、同时，因为用户程序比内核程序更容易发生错误，如果用户程序在禁用中断后发生异常退出，中断便不会被重新启用。

5.11 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

解释一下为什么中断的方法也不适合实现在多核系统上的同步。

答：

1、中断一个处理器核心是没有用的。如果仅仅在发起中断的进程所在的处理器上禁用中断，那么并不会影响到其它处理器——在其它处理器上执行的进程仍然可以访问临界区段，违反互斥原则。

2、如果在要在所有处理器上禁用中断，会因为要在多个处理器之间通信而十分耗时。这样每次进入临界区段时都会有延迟，系统的运行效率会下降。

5.12 The Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore. Explain why this policy is in place.

Linux 内核中有一条规则：进程不能在持有自旋锁的时候尝试获得信号量。解释一下为什么要这么做。

答：因为进程在尝试获得信号量的时候，有可能会被系统强制休眠（sleep）或者阻塞（block），但是如果一个进程持有自旋锁（spinlock），它则不应该被休眠（因为它在忙碌等待）。因此这一条规则防止了进程因为持有自旋锁的时候被中断而产生的饥饿现象。

5.13 Describe two kernel data structures in which race conditions are possible. Be sure to include a description of how a race condition can occur.

描述两种可能产生竞争状态（race condition）的内核数据结构，并且解释为什么这种结构会引起竞争状态。

答：

1. 树状数据结构，如 PID（process ID）分配系统和进程表（process table）：如果多个进程被同时创建，内核在分配 PID 和将它们加入进程表时会进入竞争状态
2. 双向循环链表，比如调度队列（schedule queue）：假设一个进程正在等待 I/O 输入，另一个进程正在被上下文切换出 CPU，如果 I/O 输入和上下文切换同时完成，那内核在将它们加入等待队列（waiting queue）时会抢夺链表的同一个指针位置，从而进入竞争状态。

5.23 Show how to implement the wait() and signal() semaphore operations in multiprocessor environments using the test_and_set() instruction. The solution should exhibit minimal busy waiting.

如何在多核环境使用 test_and_set()命令来实现 wait() 和 signal()的信号量操作？给出你的代码框架，尽量使忙碌等待降到最低。

答：

```
int guard = 0; // guard == 0 meaning critical section available
int semaphore_value = 0; // number of resources available

wait()
{
    while (test_and_set(&guard) == 1);
    if (semaphore_value == 0)
    {
        /* atomically add process to a queue of processes
        waiting for the semaphore */
      guard = 0;
      block();
    }
    else
    {
        semaphore_value--;
       guard = 0;
    }
}

signal()
{
    while (test_and_set(&guard) == 1);
    if (semaphore_value == 0 && /* there is a process on the wait queue */)
    {
        /* wake up the first process in the queue */
    }
    else
    {
        semaphore_value++;
```

```
    }
    guard = 0;
}
/* busy waiting is minimized by limiting it to the critical section
of the wait() and signal() operations, which are short */
```

5.25 Demonstrate that monitors and semaphores are equivalent insofar as they can be used to implement solutions to the same types of synchronization problems.

证明管程和信号量在解决同步问题时是等价的。

答：因为管程（monitor）信号量（semaphore）都是使用一个值作为信号，外加一个等待队列的形式来对进行同步进行控制，所以理论上二者可以互相转换。接下来用伪代码来实现这种转换：

```
//使用管程（monitor）实现信号量（semaphore）：
monitor Semaphore
{
    int value = 0;
    condition c;

    V()
    {
        value++;
        c.signal();
    }

    P()
    {
        while (value == 0)
            c.wait();
        value--;
    }
}
```

反之，可以使用信号量实现管程（参见课本代码）。因此，信号量和管程可以用来解决同样的问题。

5.26 Design an algorithm for a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

设计一个算法实现一个带有“有界缓冲区”的管程，其中缓冲区（buffers）放在管程的内部。

答：管程的结构如下，并举出该管程在生产者-消费者问题上的实现：

```
monitor bounded_buffer
{
    int items[MAX_ITEMS];
```

```
    int num = 0;
    condition full, empty;

    void produce(int v)
    {
        while (num == MAX_ITEMS)
            full.wait();
        items[num++] = v;
        empty.signal();
    }

    int consume()
    {
        int retVal;
        while (num  == 0)
            empty.wait();
        retVal = items[--num];
        full.signal();
        return retVal;
    }
}
```

5.32 A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: the sum of all unique numbers associated with all the processes currently accessing the file must be less than *n*. Write a monitor to coordinate access to the file.

有若干进程共享一个文件，每个进程都有一个特征码。文件可以在同一时刻被多个进程访问，但要满足如下规则：同时访问该文件的所有进程的特征码之和必须小于 n。

试着编写一个管程来实现对这些进程的同步。

答：

```
monitor file_access
{
    int sum = 0;
    int n;
    condition c;

    void access_file (int my_num)
    {
        while (sum + my_num >= n)
            c.wait();
        sum += my_num;
    }
```

```
    void finish_access (int my_num)
    {
        sum -= my_num;
        c.broadcast(); // here we use broadcast() instead of signal()
(which only signals the first process in the waiting queue) because
there might be more than one process that can be allowed access after
the current process is finished
    }
}
```

5.37 Assume that a finite number of resources of a single resource type must be managed. Processes may ask for a number of these resources and will return them once finished. As an example, many commercial software packages provide a given number of ***licenses***, indicating the number of applications that may run concurrently. When the application is started, the license count is decremented. When the application is terminated, the license count is incremented. If all licenses are in use, requests to start the application are denied. Such requests will only be granted when an existing license holder terminates the application and a license is returned. The following program segment is used to manage a finite number of instances of an available resource. The maximum number of resources and the number of available resources are declared as follows:

假设现在要对一种有特定数量的资源类型进行管理。进程可能需求一定数量的该种资源，并在运行完毕后释放它们。例如：很多商业软件都提供一定数量的***使用授权***，表示用户在同一时间可以同时运行该软件的实例数量。当有一个软件的实例启动时，授权数量就减一；当实例终止时，授权数量再加一。如果所有授权都已经被使用，则不允许开启新的实例，其必须等待其他实例终止并退回一个授权。下面的代码展示了一个处理这类问题的模型，其中最大资源的数量和可用资源的数量声明如下：

```
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
```

When a process wishes to obtain a number of resources, it invokes the decrease_count() function:

当一个进程想要获取一定数量的资源时，它调用decrease_count()函数：

```
/* decrease available resources by count resources */
/* return 0 if sufficient resources available, */
/* otherwise return -1 */
int decrease_count(int count) {
      if (available_resources < count)
            return -1;
      else {
           available_resources -= count;
           return 0;
      }
```

```
}
```

When a process wants to return a number of resources, it calls the increase_count() function:
当一个进程要还回资源时，它调用increase_count()函数：

```
/* increase available resources by count */
int increase_count(int count) {
      available_resources += count;
      return 0;
}
```

The preceding program segment produces a race condition. Do the following:
以上所述的代码段会产生竞争条件，回答下列问题：

a. Identify the data involved in the race condition.
b. Identify the location (or locations) in the code where the race condition occurs.
c. Using a semaphore or mutex lock, fix the race condition. It is permissible to modify the decrease_count() function so that the calling process is blocked until sufficient resources are available.

a、判断哪一个变量处在竞争条件状态。
b、判断在代码的何处位置开始发生竞争条件
c、使用信号量和互斥锁，避免竞争条件的产生。你可以修改decrease_count()使得进程在获得足够的资源前保持阻塞状态。

答：

a. 处在竞争状态的数据是：available_resources
b. increase_count()和decrease_count()函数中涉及到available_resources的代码段存在竞争状态
c. 解决：将available_resources声明为带等待队列的信号量，代码：

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

void initialize(Semaphore *S)
{
    S->value = MAX_RESOURCES;
}

void decrease_count(Semaphore *S, int count) // this is a P()
{
    S->value -= count;
    if (S->value <= 0)
        /* add process to S->list */
        /* block on semaphore */;
```

```
    return;
}

void increase_count(Semaphore *S, int count) // this is a V()
{
    S->value += count;
    if (S->value < MAX_RESOURCES)
        /* notify all processes waiting in S->list */
        /* remove i processes from S->list */
        /* wakeup those i processes */;
    return;
}
```

第六章 处理器调度

实践练习部分 Practice Exercise

6.1 A CPU-scheduling algorithm determines an order for the execution of its scheduled processes. Given *n* processes to be scheduled on one processor, how many different schedules are possible? Give a formula in terms of *n*.

n 个进程进行进程调度，一共有多少种不同的调度方案？用含 n 的表达式表示。

答：共有 n!种不同的调度可能

6.2 Explain the difference between preemptive and nonpreemptive scheduling.

解释抢占式调度和非抢占式调度的不同的点

答：抢占式（preemptive）调度允许进程执行中途被打断，将 CPU 分配给另一个进程。非抢占式（nonpreemptive）调度则不允许，系统必须等待进程执行完后释放 CPU。

6.3 Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

Process	Arrival Time	Burst Time
P_1	0.0	8
P_2	0.4	4
P_3	1.0	1

a. What is the average turnaround time for these processes with the FCFS scheduling algorithm?

b. What is the average turnaround time for these processes with the SJF scheduling algorithm?

c. The SJF algorithm is supposed to improve performance, but notice that we chose to run process P_1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes P_1 and P_2 are waiting during this idle time, so their waiting time may increase. This algorithm could be called future-knowledge scheduling.

上表列出了一些进程的到达时间（Arrival Time）和运行时间（Burst Time），回答以下问题：

a、使用 FCFS 调度，平均周转时间是多少？

b、使用 SJF 调度，平均周转时间是多少？

c、SJF 调度理应能提高性能，但是注意到上面的调度算法中，P1 总是从时间 0 开始，因为我们不知道还有两个执行时间较短的进程很快就会加入等待队列。假设 CPU 先等待 1 秒，然后再执行 SJF 调度，这种算法被称为预先知晓调度，计算此时的平均周转时间。注意：因为在 0-1 秒这一段时间，所有进入的进程都处于等待状态，因此计算过程中它们的等待时间会增加。

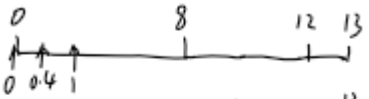
答：

a. (0 + 8 + (8 – 0.4) + 4 + (12 – 1) + 1) / 3 = 10.53

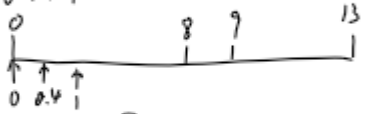
b. (8 + (8 – 1) + 1 + (9 – 0.4) + 4) / 3 = 9.53

c. (0 + 1 + (1 + 1 – 0.4) + 4 + (1 + 5) + 8) / 3 = 6.87

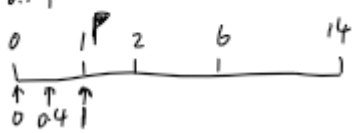
a.



b.



c.



6.4 What advantage is there in having different time-quantum sizes at different levels of a multilevel queueing system?

对于多级队列的调度，对不同级别的队列采用不同的时间片大小，有什么优势？

答：需要比较高响应度（responsiveness）的进程可以在较小时间片（time-quantum）的队列上执行以获得更高的响应度。如果在这个队列中无法完成，可以把它转移到较大时间片的队列上，和那些不需要高响应度的进程一起排队执行，节省下上下文切换的时间，提高计算机整体的运行效率。

6.5 Many CPU-scheduling algorithms are parameterized. For example, the RR algorithm requires a parameter to indicate the time slice. Multilevel feedback queues require parameters to define the number of queues, the scheduling algorithm for each queue, the criteria used to move processes between queues, and so on.

These algorithms are thus really sets of algorithms (for example, the set of RR algorithms for all time slices, and so on). One set of algorithms may include another (for example, the FCFS algorithm is the RR algorithm with an infinite time quantum). What (if any) relation holds between the following pairs of algorithm sets?

a. Priority and SJF
b. Multilevel feedback queues and FCFS
c. Priority and FCFS
d. RR and SJF

对于以上四个项中提到的调度算法，分析每一项中的两个算法在逻辑实现上有没有相互联系，如果有，简述一下这种联系。

答：

a. SJF 中 CPU 执行时间最短的进程拥有最高的优先级。
b. FCFS 是多级队列调度的一个特例，当只有最低级的队列（时间片无限长）时就是 FCFS 调度。
c. FCFS 也是一种优先级调度，FCFS 中发起时间最早的进程拥有最高的优先级。
d. 没有关联。

6.6 Suppose that a scheduling algorithm (at the level of short-term CPU scheduling) favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O-bound programs and yet not permanently starve CPU-bound programs?

假设有一种调度算法倾向于最近一段时间使用最少 CPU 时间的进程。解释为什么这种调度算法会优先执行 I/O 密集型进程，同时也不会导致 CPU 密集型进程产生饥饿？

答：I/O 密集型（I/O-bound）进程会被优先执行是因为它们的 CPU 执行区间通常较短。而 CPU 密集型（CPU-bound）进程不会饥饿（starvation）是因为 I/O 密集型进程在 I/O 执行区间通常会释放 CPU 占用，使得 CPU 有时间来处理 CPU 密集型的进程。

6.7 Distinguish between PCS and SCS scheduling.

辨析 PCS 和 SCS 调度的区别。

答：线程库（thread library）将线程分配给轻量进程（LWP），这属于 PCS 调度。而操作系统调度内核线程属于 SCS 调度。因此在多对一（many-to-one）和多对多（many-to-many）系统上，这两者是不同的，但在一对一（one-to-one）系统上，这两者相同。

6.9 The traditional UNIX scheduler enforces an inverse relationship between priority numbers and priorities: the higher the number, the lower the priority. The scheduler recalculates process priorities once per second using the following function:

Priority = (recent CPU usage / 2) + base

where base = 60 and *recent CPU usage* refers to a value indicating how often a process has used the CPU since priorities were last recalculated.

Assume that recent CPU usage is 40 for process *P*1, 18 for process *P*2, and 10 for process *P*3. What will be the new priorities for these three processes when priorities are recalculated? Based on this information, does the traditional UNIX scheduler raise or lower the relative priority of a CPU-bound process?

在早期的 UNIX 系统中，进程的优先级与它们的优先级数字相反，数字越大，优先级反而越低。调度系统每 1 秒就重新计算进程的优先级，计算公式如下：

优先级数字 = (最近的 CPU 使用率 / 2) + 基准

将基准设置为 60，最近的 CPU 使用率标志自上一次计算优先级以来，该进程使用 CPU 的频率。

假设进程 P1 的最近 CPU 使用率为 40，P2 的为 18，P3 的为 10，则下一次计算时，三个进程的优先级分别时多少？根据这一准则，UNIX 调度系统是提高了还是降低了 CPU 密集型进程的相对优先级？

答：再次计算时，P1, P2, P3 的优先级分别为 80，69 和 65。因为基准 60 总是不变，而每次调整优先级时，CPU 使用率的比重都会减半，因此调度器会降低 CPU 密集型进程的相对优先级。

习题部分 Exercise

6.10 Why is it important for the scheduler to distinguish I/O-bound programs from CPU-bound programs?

为什么调度器一定要区别 I/O 密集型进程和 CPU 密集型进程？

答：因为 CPU 密集型进程通常占用 CPU 的时间较长，而 I/O 密集型进程较短。同一种调度方法往往不能适用两种进程混合出现的情况，因此应该根据进程类型来采用不同的调度方法。

6.11 Discuss how the following pairs of scheduling criteria conflict in certain settings.

a. CPU utilization and response time
b. Average turnaround time and maximum waiting time
c. I/O device utilization and CPU utilization

讨论在确定调度方法的时候，以下每一组的两个性能标准之间为什么会相互冲突（为什么鱼和熊掌不可兼得）？

a. CPU 利用率和响应时间
b. 平均周转时间和最长等待时间
c. I/O 设备利用率和 CPU 利用率

答：

a. 若要响应时间低，通常需要更频繁的上下文切换，这样 CPU 利用率就低了。
b. 要最小化平均周转时间（average turnaround time）的话，CPU 执行区间较长的进程等待时间会更长。
c. 若要增加 I/O 利用率，应该先执行 I/O 密集型（I/O-bound）进程，而若要增加 CPU 利用率，则应该先执行 CPU 密集型进程（因为这样可以尽量减少上下文切换的次数）。

6.14 Consider the exponential average formula used to predict the length of the next CPU burst. What are the implications of assigning the following values to the parameters used by the algorithm?

a. $\alpha = 0$ and $\tau_0 = 100$ milliseconds

b. $\alpha = 0.99$ and $\tau_0 = 10$ milliseconds

参考预测下一次 CPU 执行时间的指数平均公式算法，判断根据以上两种不同的参数，预测的结果会分别受哪一个变量的影响？

答：

a. 这样做因为$\alpha = 0$，所有预测结果与t_n无关，每次预测的τ_n均为 100ms。

b. 这样做的话$\tau_{n+1} = 0.99t_n + 0.1$，上次执行的实际时间$t_n$在预测时占绝大部分比重，10ms 的延迟无关紧要。

6.15 A variation of the round-robin scheduler is the **regressive round-robin** scheduler. This scheduler assigns each process a time quantum and a priority. The initial value of a time quantum is 50 milliseconds. However, every time a process has been allocated the CPU and uses its entire time quantum (does not block for I/O), 10 milliseconds is added to its time quantum, and its priority level is boosted. (The time quantum for a process can be increased to a maximum of 100 milliseconds.) When a process blocks before using its entire time quantum, its time quantum is reduced by 5 milliseconds, but its priority remains the

same. What type of process (CPU-bound or I/O-bound) does the regressive round-robin scheduler favor? Explain.

一种 RR 调度算法的变种是***累退 RR*** 调度。这种调度算法为每个进程赋予了一个时间片和优先级。假设初始时间片为 50ms，但当每一次进程被分配给 CPU 执行一个时间片（不包括 I/O 阻塞的时间）后，其时间片就增加 10ms，然后将其优先级增加（时间片最大只能增加到 100ms）。每当一个进程没有执行完其全部的时间片就阻塞（中断）了，它的时间片会被减少 5ms，但优先级不变。

试问这种调度对 CPU 密集型进程更为友好还是对 I/O 密集型进程更为友好，解释理由。

答：这调度对 CPU 密集型更友好。因为每次一个进程在时间片内没执行完，它还能被给予更多的时间，而且它的优先级还会被提升。相反，对于 I/O 密集型进程，这种机制不会带来什么提升。

6.16 Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5, all at time 0.

a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

c. What is the waiting time of each process for each of these scheduling algorithms?

d. Which of the algorithms results in the minimum average waiting time (over all processes)?

参考上表给出的 5 个进程的运行时间和优先级，假设它们都在时间 0 时刻到达。

a. 分别画出采用 FCFS、SJF、非抢占式优先级调度和 RR 调度(时间片=2)时调度顺序的甘特（Gantt）图。

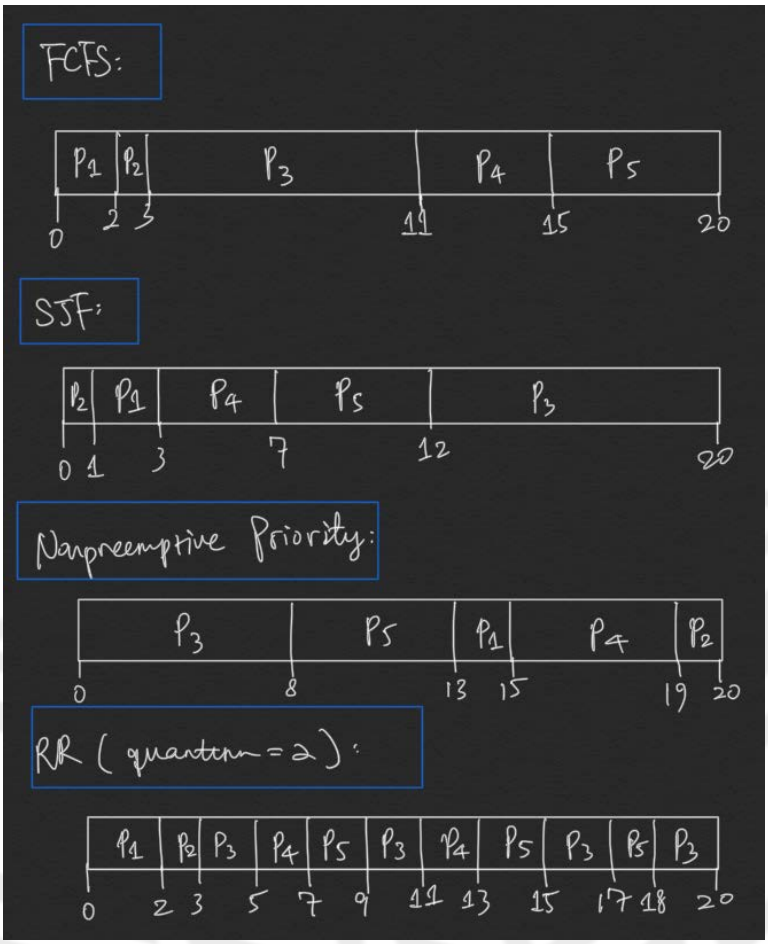
b. 对 a 中的各种调度算法，每个进程的平均周转时间是多少？

c. 对 a 中的各种调度算法，每个进程的等待时间是多少？

d. 对 a 中的各种调度算法，哪个算法拥有最短的平均等待时间？

答：

a. 如下图所示



b. 如下表

	P1	P2	P3	P4	P5
FCFS	2	3	11	15	20
SJF	3	1	20	7	12
Non-pre. Pri.	15	20	8	19	13
RR (qtm = 2)	2	3	20	13	18

c. 如下表

	P1	P2	P3	P4	P5
FCFS	0	2	3	11	15
SJF	1	0	12	3	7
Non-pre. Pri.	13	19	0	15	8
RR (qtm = 2)	0	2	12	9	13

d. SJF 调度平均等待时间最短(4.6)。

6.17 The following processes are being scheduled using a preemptive, round-robin scheduling algorithm. Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an ***idle***

task (which consumes no CPU resources and is identified as P_{idle}). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a time quantum is 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

Thread	Priority	Burst	Arrival
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105

a. Show the scheduling order of the processes using a Gantt chart.

b. What is the turnaround time for each process?

c. What is the waiting time for each process?

d. What is the CPU utilization rate?

一些进程的优先级、运行时间和到达时间如上表。调度算法采用抢占式的 RR 算法，优先级的数字越大，对应进程的优先级越高。除此之外系统中还有一个不占用任何 CPU 资源的“空闲”进程，记为P_{idle}。每当调度算法没有任何其他的进程可以调度的时候，其就调度P_{idle}运行，P_{idle}的优先级为 0。

现在假设时间片为 10 个单位，如果有一个进程因为另一个更高优先级进程的出现而被抢占，它默认进入等待队列的尾部。回答下列问题：

a. 画出进程调度顺序的甘特图。

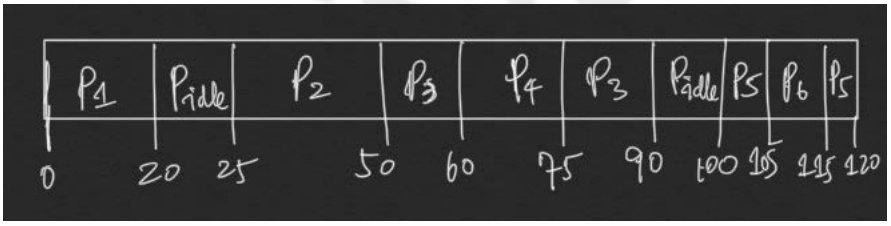
b. 计算每个进程的平均周转时间。

c. 计算每个进程的等待时间

d. 计算 CPU 利用率。

答：

a. 如下图



b. 如下表

P1	P2	P3	P4	P5	P6
20	25	60	15	20	10

c. 如下表

P1	P2	P3	P4	P5	P6
0	0	35	0	10	0

d. CPU utilization rate = (120-10-5)/120 = 87.5%

6.19 Which of the following scheduling algorithms could result in starvation?
以下哪些调度算法可能导致饥饿?

a. First-come, first-served
b. Shortest job first
c. Round robin
d. Priority

答：b（短时优先）, d（优先级调度）会导致饥饿。

6.20 Consider a variant of the RR scheduling algorithm in which the entries in the ready queue are pointers to the PCBs.

a. What would be the effect of putting two pointers to the same process in the ready queue?
b. What would be two major advantages and two disadvantages of this scheme?
c. How would you modify the basic RR algorithm to achieve the same effect without the duplicate pointers?

考虑一种 RR 调度的变种，这种调度算法中等待队列里的每一个条目都是一个指向进程 PCB 的指针。回答下列问题:

a. 如果队列里有两个指针执行同一个进程，会对这个进程的调度有什么影响?
b. 这种调度算法有什么优势和劣势，各举出两点。
C 能否通过一种方式改进基本的 RR 调度算法，使其不使用拷贝指针的方式也能达到和这种算法一样的效果?

答：

a. 那个进程将会获得两倍的执行时间（前提是其 PCB 也要有一份拷贝）。
b. 优点：1）可以给需要更多时间的进程更多的时间，2）不一定要等时调度进程，可以按照进程的工作量调度进程，利用更少的上下文切换带来更高的效率。
缺点：1）我们不一定能准确预测每个进程需要多少时间，可能会造成浪费，2）存储多个重复的指针会浪费内存空间。
c. 可以，在每个 PCB 中加一个数据域，用来存储建议分配给这个进程的时间片的数目。

6.21 Consider a system running ten I/O-bound tasks and one CPU-bound task. Assume that the I/O-bound tasks issue an I/O operation once for every millisecond of CPU computing and that each I/O operation takes 10 milliseconds to complete. Also assume that the context-switching overhead is 0.1 millisecond and that all processes are long-running tasks. Describe the CPU utilization for a round-robin scheduler when:

a. The time quantum is 1 millisecond
b. The time quatum is 10 milliseconds

考虑一个系统准备运行 10 个 I/O 密集型进程和 1 个 CPU 密集型进程。假设 I/O 密集型进程请求一个 I/O 操作，需要 CPU 运行时间 1 ms，I/O 设备数据传输 10ms 。并且一次进程上下文切换的时间开销时 0.1ms，所有的进程都长时间运行，不会中途退出。

计算根据 RR 调度在时间片分别为 1ms 和 10ms 的情况下，CPU 的利用率。

答：

计算公式是$CPU利用率 = \frac{IO进程数 * p + q}{IO进程数 * p + q + IO进程数 * 上下文切换时间} * 100\%$，其中 p 是 I/O 操作时间（1ms），q 是时间片大小，所以计算结果是：

a. CPU utilization rate = (10 * 1 + 1 * 1) / (10 * 1 + 1 * 1 + 10 * 0.1) = 91%
b. CPU utilization rate = (10 * 1 + 1 * 10) / (10 * 1 + 1 * 10 + 10 * 0.1) = 95%

6.24 Explain the differences in how much the following scheduling algorithms discriminate in favor of short processes:

a. FCFS
b. RR
c. Multilevel feedback queues

分析以上调度算法对运行时间短的进程是否有偏袒?

答：

a. FCFS 不偏袒短进程。因为无论长短，它们都要在等待队列里等待前面的进程完成。事实上，FCFS 甚至对短进程不利，因为如果它们前面有长进程的话，它们会等很久。
b. RR 是否偏袒短进程要根据时间片的设定决定。当短进程的长度小于等于时间片的长度的时候，RR 对短进程就没有什么优势了。
c. MFQ（多级队列）偏袒短进程。因为 MFQ 中短进程的优先级更高。

第七章 死锁

实践练习部分 Practice Exercise

7.3 Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix *Need*?

b. Is the system in a safe state?

c. If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately?

分析上图一个系统在某个时刻的资源快照，根据银行家算法回答：

a. 作出 Need 矩阵

b. 系统是否处于安全状态？

c. 如果一个进程 P1 请求资源(0,4,2,0)，这个请求可以被立刻允许吗？

答：

a.

Need	A	B	C	D
P_0	0	0	0	0
P_1	0	7	5	0
P_2	1	0	0	2
P_3	0	0	2	0
P_4	0	6	4	2

b. 是的。空闲（available）资源能够满足 P3 和 P0 运行。等待 P3/P0 运行完后，存在安全序列（safe sequence）。

c. 可以。满足 P1 的请求（request）后，空闲资源矩阵为(1, 1, 0, 0)，存在安全序列 <P0, P2, P3, P1, P4>。

7.6 Consider a computer system that runs 5,000 jobs per month with no deadlock-prevention or deadlock-avoidance scheme. Deadlocks occur about twice per month, and the operator must terminate and rerun about 10 jobs per deadlock. Each job is worth about $2 (in CPU time), and the jobs terminated tend to be about half-done when they are aborted.

A systems programmer has estimated that a deadlock-avoidance algorithm (like the banker's algorithm) could be installed in the system with an increase in the average execution time per job of about 10 percent. Since the machine currently has 30-percent idle time, all 5,000 jobs per month could still be run, although turnaround time would increase by about 20 percent on average.

a. What are the arguments for installing the deadlock-avoidance algorithm?

b. What are the arguments against installing the deadlock-avoidance algorithm?

考虑一个系统每个月要运行 5000 个工作进程，而且没有死锁保护机制。死锁平均每个月发生两次，每次发送死锁的时候系统就不得不重启，重新运行 10 个工作进程。假设每个进程占用的 CPU 时间价值 2 元。

假设现在程序员想要改进系统，如采用银行家算法防止死锁，但是会导致每个工作进程的运行时间提高 10%，如果现在整个系统有 30%的时间处于空闲（在 5000 个工作进程都都能正常运行的情况下），防止死锁会使平均周转时间增加 20%

试问该系统应不应该采取这种防死锁改进，分别说明理由。

答：

a. 应该使用的理由：可以确定死锁不会出现，并且根据数据，仍然能保证 5000 个任务能够完成。

b. 不应该使用的理由：死锁不经常出现，而且出现时带来的影响也不大。如果加入防死锁，反而会降低系统在其他大部分时间内的效率。

7.7 Can a system detect that some of its processes are starving? If you answer "yes," explain how it can. If you answer "no," explain how the system can deal with the starvation problem.

操作系统可以检测是否有进程处于饥饿状态吗？如果是，解释如何实现，否则解释系统应该如何处理可能的饥饿问题。

答：

可以。可以设置一个运行时长 T，当进程在等待队列中的时长超过 T 之后就默认其陷入饥饿状态。

此外操作系统也可以直接定义当前等待时间最长的进程处于饥饿状态，然后调高他的优先级。

7.8 Consider the following resource-allocation policy. Requests for and releases of resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any processes that are blocked waiting for resources. If a blocked process has the desired resources, then these resources are taken away from it and are given to the requesting process. The vector of resources for which the blocked process is waiting is increased to include the resources that were taken away.
For example, consider a system with three resource types and the vector *Available* initialized to (4,2,2). If process *P*0 asks for (2,2,1), it gets them. If *P*1 asks for (1,0,1), it gets them. Then, if *P*0 asks for (0,0,1), it is blocked (resource not available). If *P*2 now asks for (2,0,0), it gets

the available one (1,0,0) and one that was allocated to *P*0 (since *P*0 is blocked). *P*0's *Allocation* vector goes down to (1,2,1), and its *Need* vector goes up to (1,0,1).

1. Can deadlock occur? If you answer "yes," give an example. If you answer "no," specify which necessary condition cannot occur.
2. Can indefinite blocking occur? Explain your answer.

考虑如下的资源分配方案：在任何时候都可以申请或释放资源，如果请求的资源因为不可用而无法被分配，则检查所有正在阻塞（等待资源）的进程，如果某个这样的进程恰好有这种资源，则从该进程中拿走资源给需要的进程。被拿走资源的进程，其目前 Need 向量中这个资源+1。

例如：假设资源向量有三种，初始 *Available* 为(4,2,2)，进程 P_0 请求资源(2,2,1)并获得了资源，然后进程 P_1 请求资源(1,0,1)并获得了资源，然后 P_0 请求资源(0,0,1)，这时资源不足，其进入阻塞。然后如果此时进程 P_2 请求资源(2,0,0)，它会从可用资源获得一个(1,0,0)以及从 P_0 中获得一个(1,0,0)，然后 P_0 的 *Allocation 向量变成*(1,2,1)，*Need* 向量变成(1,0,1)。回答如下问题：

1、可能会发生死锁吗？如果回答是，举出一个例子，否则，解释什么条件使得死锁不会发生。

2、有进程可能会被无限阻塞吗，解释原因。

答：

1. 不会。因为有资源的抢占机制作为条件。
2. 有可能。因为某些进程的资源可能会不断地被抢占，陷入饥饿状态。

7.10 Is it possible to have a deadlock involving only one single-threaded process? Explain your answer.

是否有可能出现死锁中只包含一个单线程进程的情况？说明理由。

答：不可能。因为单线程最多只能被阻塞，这不符合死锁出现的条件之一——不可能出现持有并等待（hold-and-wait）。

习题部分 Exercise

7.17 Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock free.

考虑一个由四个同一种资源组成的系统，资源正在被三个进程使用，且每个进程至多需要两种资源。证明这个系统是没有死锁的。

答：因为共有 4 个单位资源，3 个进程，根据鸽巢原理，一定会有一个进程能够拥有两个单位的资源，因此不会陷入死锁。

7.19 Consider the version of the dining-philosophers problem in which the chopsticks are placed at the center of the table and any two of them can be used by a philosopher. Assume that requests for chopsticks are made one at a time. Describe a simple rule for

determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

考虑一种哲学家进餐问题：把筷子都放在餐桌的中央，每个哲学家每次都可以从中央拿走一只筷子。假设一次只有一个哲学家发出需要筷子的申请。确定一种规则来判断能否允许一个哲学家的筷子请求，而且不能发生死锁。

答：可以使用这种规则：当一个哲学家要去拿他的第一根筷子的时候，进行判断：当且仅当所有其他哲学家都只有一根筷子或没有筷子，且只剩最后一根筷子的时候，拒绝这个请求。否则就可以允许该请求 。

7.21 We can obtain the banker's algorithm for a single resource type from the general banker's algorithm simply by reducing the dimensionality of the various arrays by 1. Show through an example that we cannot implement the multiple-resource-type banker's scheme by applying the single-resource-type scheme to each resource type individually.

如果将普通的银行家算法的矩阵降低一个维度，我们就可以得到一个对于每一种特定的资源而进行判断的"单类资源的银行家算法"。举出一个例子来证明，不能通过对每一种资源分别实施"单类资源的银行家算法"，判断每一种资源不会死锁，从而得到整个系统不会产生死锁的结论。

答：

Answer: Consider a system with resources A, B, and C and processes P_0, P_1, P_2, P_3, and P_4 with the following values of *Allocation*:

Allocation			
	A	B	C
P_0	0	1	0
P_1	3	0	2
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

And the following value of *Need*:

Need			
	A	B	C
P_0	7	4	3
P_1	0	2	0
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

If the value of *Available* is (2 3 0), we can see that a request from process P_0 for (0 2 0) cannot be satisfied as this lowers *Available* to (2 1 0) and no process could safely finish.

However, if we were to treat the three resources as three single-resource types of the banker's algorithm, we get the following:
For resource A (which we have 2 available),

	Allocated	Need
P_0	0	7
P_1	3	0
P_2	3	6
P_3	2	0
P_4	0	4

Processes could safely finish in the order of P_1, P_3, P_4, P_2, P_0.
For resource B (which we now have 1 available as 2 were assumed assigned to process P_0),

	Allocated	Need
P_0	3	2
P_1	0	2
P_2	0	0
P_3	1	1
P_4	0	3

Processes could safely finish in the order of P_2, P_3, P_1, P_0, P_4.
And finally, for For resource C (which we have 0 available),

	Allocated	Need
P_0	0	3
P_1	2	0
P_2	2	0
P_3	1	1
P_4	2	1

Processes could safely finish in the order of P_1, P_2, P_0, P_3, P_4.
As we can see, if we use the banker's algorithm for multiple resource types, the request for resources (0 2 0) from process P_0 is denied as it leaves the system in an unsafe state. However, if we consider the banker's algorithm for the three separate resources where we use a single resource type, the request is granted. Therefore, if we have multiple resource types, we must use the banker's algorithm for multiple resource types.

解释：*考虑以上的 Allocation 和 Need 矩阵，在这种情况下，如果 P 增加一个(0,2,0)的请求，按照矩阵维度的银行家算法来计算，这个请求不能满足，因为剩下的 ABC 三个资源只有(2,1,0)，会使得 P1、P2、P3、P4 直接死锁。但是如果分别考虑 A,B,C 的一维银行家算法，则这个需求应该可以被满足，这就得到了一个反例。*

7.22 Consider the following snapshot of a system:

	Allocation	*Max*
	A B C D	*A B C D*
P_0	3 0 1 4	5 1 1 7
P_1	2 2 1 0	3 2 1 1
P_2	3 1 2 1	3 3 2 1
P_3	0 5 1 0	4 6 1 2
P_4	4 2 1 2	6 3 2 5

Using the banker's algorithm, determine whether or not each of the following states is unsafe. If the state is safe, illustrate the order in which the processes may complete. Otherwise, illustrate why the state is unsafe.

a. ***Available*** $= (0, 3, 0, 1)$

b. ***Available*** $= (1, 0, 0, 2)$

使用银行家算法判断在 a 和 b 两种可用资源的情况下，系统是否是安全的，如果是安全的，给出进程获得资源的顺序；如果不是安全状态，说明理由。

答：

a. 此时系统是不安全的，P_0 和 P_4 会陷入死锁

b. 此时系统是安全的，资源获得顺序为：<P1, P2, P0, P3, P4>

7.23 Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	2 0 0 1	4 2 1 2	3 3 2 1
P_1	3 1 2 1	5 2 5 2	
P_2	2 1 0 3	2 3 1 6	
P_3	1 3 1 2	1 4 2 4	
P_4	1 4 3 2	3 6 6 5	

Answer the following questions using the banker's algorithm:

a. Illustrate that the system is in a safe state by demonstrating an order in which the processes may complete.

b. If a request from process P_1 arrives for $(1, 1, 0, 0)$, can the request be granted immediately?

c. If a request from process P_4 arrives for $(0, 0, 2, 0)$, can the request be granted immediately?

根据银行家算法回答以下问题：

a. 证明系统处于安全状态，写出进程资源获得的顺序

b. 如果进程 P1 请求资源(1,1,0,0)，该请求可以被立刻满足吗?

c. 如果进程 P4 请求资源(0,0,2,0)，该请求可以被立刻满足吗?

答：

a. <P0, P3, P1, P2, P4>

b. 可以。将资源分配给 P1 后，系统仍处在安全状态（safe state）。可以按照<P0, P3, …>的顺序执行进程。

c. 不可以。将资源分配给 P4 后，系统将进入不安全状态（unsafe state），可能会陷入死锁。

7.25 A single-lane bridge connects the two Vermont villages of North Tunbridge and South Tunbridge. Farmers in the two villages use this bridge to deliver their produce to the neighboring town. The bridge can become deadlocked if a northbound and a southbound farmer get on the bridge at the same time. (Vermont farmers are stubborn and are unable to back up.) Using semaphores and/or mutex locks, design an algorithm in pseudocode that prevents deadlock. Initially, do not be concerned about starvation (the situation in which northbound farmers prevent southbound farmers from using the bridge, or vice versa).

在南蛤蟆村和北蛤蟆村之间有一条独木桥，两个村的农民每天使用这座桥来向邻村运送农产品。如果同时又一个南村的人和一个北村的人想上桥，就会产生死锁的情况（蛤蟆村的人脑子都不太灵光，不晓得后退）。使用互斥锁/信号量/信号量和互斥锁结合的方法，设计一个算法，来防止死锁。（使用伪代码编写，你无需考虑饥饿，即北村的人一直占着桥不让南村人经过的情况，或者反过来）。

答：使用一个互斥的信号量即可，伪代码如下：

```
semaphore bridge_available = 1; /* initialize semaphore */
void enter_bridge() {
    bridge_available.wait();
    /* acquire the lock when entering the bridge */
}

void leave_bridge() {
    bridge_available.signal();
  /* yield the lock when leaving the bridge*/
}
```

第八章 主要内存

实践练习部分 Practice Exercise

8.1 Name two differences between logical and physical addresses.
说出两个物理地址和逻辑地址之间的区别。

答：逻辑地址不指向实际的内存地址，物理地址指向实际内存中的地址。
逻辑地址是由 CPU 发出的，而物理地址是由内存管理单元(MMU)根据逻辑地址生成的。

8.2 Consider a system in which a program can be separated into two parts: code and data. The CPU knows whether it wants an instruction (instruction fetch) or data (data fetch or store). Therefore, two base– limit register pairs are provided: one for instructions and one for data. The instruction base–limit register pair is automatically read-only, so programs can be shared among different users. Discuss the advantages and disadvantages of this scheme.

在一个系统中程序可以被分为代码段和数据段两个部分，代码段负责存储指令码，数据段负责存储要操作的数据。可以采取设置两个基址-边界寄存器对的方式来实现 CPU 对程序的执行：一对寄存器存储代码段，另一对存储数据段。存储代码段的寄存器对应该是原子只读的，这样程序可以方便地在多个用户间共享。讨论以下这个方案的优点和缺点。

答：
优势：不同的进程之间可以更好地共享代码和数据。
缺点：代码和数据必须分开存储。但在汇编代码中，它们往往相互依附在一起。

8.3 Why are page sizes always powers of 2?
为什么页的大小一定是 2 的幂?

答：因为内存地址是由二进制比特位表示的。如果分页大小为 2 的次方，计算地址时只需要进行比特位的左移和右移操作，不需要进行算数运算。

8.4 Consider a logical address space of 64 pages of 1024 words each, mapped onto a physical memory of 32 frames.

a. How many bits are there in the logical address?
b. How many bits are there in the physical address?

假设逻辑地址空间有 64 页，每页 1024 个字，将其映射到有 32 个帧的物理内存。
a. 逻辑地址应该有几位?
b. 物理地址应该有几位?

答：

a. log64 + log1024 = 16 bits
b. log32 + log1024 = 15 bits

8.5 What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how this effect could be used to decrease the amount of time needed to copy a large amount of memory from one place to another. What effect would updating some byte on the one page have on the other page?

如果允许一个页表中的两个不同条目指向同一个页帧，会有什么影响？解释一下为什么这种影响可以被用于降低在内存之间拷贝大量数据的时间。如果更新一个页面上的某个字节，对拷贝过去的页面的相应字节有什么影响?

答：允许一个页表中的两个不同条目指向同一个页帧，则不同用户之间可以方便地共享数据。在复制内存中的数据时，只需要复制分页表上的词条（entry）即可，节约时间。但是，如果访问数据的进程是不可重入的（non-reentrant），一个进程修改了某个页面上的字节，另一个拥有指针拷贝的进程页面也会受影响。

8.6 Describe a mechanism by which one segment could belong to the address space of two different processes.

如何使一个分段的地址空间可以被两个不同的进程共享？简述你的方案。

答：让两个进程的分段表（segmentation table）指向同一块物理地址，同时，不同进程中对应的分段号也要相等。

8.7 Sharing segments among processes without requiring that they have the same segment number is possible in a dynamically linked segmentation system.

a. Define a system that allows static linking and sharing of segments without requiring that the segment numbers be the same.
b. Describe a paging scheme that allows pages to be shared without requiring that the page numbers be the same.

在动态链接的分段系统中，段号不同的进程之间也可以共享一个分段，回答问题：

a. 描述一下如何通过静态链接的方式，使段号不同的进程之间也可以共享一个分段？

b. 描述一下在分页系统中，如何使页码不同的的页之间实现共享？

答：这两种机制都需要一个进程在不知晓其分段号或分页号的情况下访问资源。解决方案是给每个进程分配四个寄存器，分别存储当前代码段基址、栈基址、全局变量段基址，以此类推。做法就是让所有的内存访问都通过这四个寄存器间接地映射到物理内存上。只需要调整这四个寄存器中的值即可做到让不同的进程用不同的进程号或分段号访问同一物理地址。

习题部分 Exercise

8.9 Explain the difference between internal and external fragmentation.

解释内部碎片和外部碎片之间的区别。

答：

内部碎片（internal fragmentation）是在一段正在被某个进程占据的内存区域内的碎片，而外部碎片是在不同进程占用的区域之间的碎片。

8.12 Most systems allow a program to allocate more memory to its address space during execution. Allocation of data in the heap segments of programs is an example of such allocated memory. What is required to support dynamic memory allocation in the following schemes?

a. Contiguous memory allocation
b. Pure segmentation

c. Pure paging

许多系统允许程序在运行期间分配更多的内存到其的地址空间中，比如允许堆分配的方式获得内存。在以下几种内存分配方式中，要支持这种动态内存分配，需要什么条件？

a. 连续内存分配　　b. 纯分段分配　　c. 纯分页分配

答：

a. 需要重新分配整个程序所占的内存空间。因为大概率当前的地址没有足够的空间进行扩展。
b. 需要重新分配需要扩展的段的内存空间，同上理。
c. 递增式（incremental）地分配新页（page）即可。程序的逻辑地址需要更新，但无需重新分配内存空间。

8.13 Compare the memory organization schemes of contiguous memory allocation, pure segmentation, and pure paging with respect to the following issues:

a. External fragmentation
b. Internal fragmentation
c. Ability to share code across processes

考虑对于连续内存分配、纯分段分配和纯分页分配，是否会有以下问题的产生：

a. 产生外部碎片　　b. 产生内部碎片　　c. 能否允许进程共享代码

答：

连续内存分配（contiguous memory allocation）和纯分段（pure segmentation）都会导致外部碎片，而纯分页（pure paging）会导致内部碎片。除了连续内存分配方式以外，其余二者都支持进程间共享代码。

	产生外部碎片	产生内部碎片	允许进程共享代码
连续内存分配	是	否	否
纯分段	是	否	是
纯分页	否	是	是

8.14 On a system with paging, a process cannot access memory that it does not own. Why? How could the operating system allow access to other memory? Why should it or should it not?

在一个支持分页的系统上，进程不能访问不属于其的内存。解释原因？

操作系统要怎么做才能允许其访问其他位置的内存？应不应该允许这么做？

答：

进程需要根据分页号（page number）和位移量（offset）在分页表（page table）中找到物理内存内存地址，进而访问物理内存。而操作系统控制着分页表，保证了不属于某个进程的物理内存地址不会出现在它的分页表中。这样做是为了保证数据安全。

要允许这种访问，操作系统只需将不属于某个进程的物理内存地址加入其分页表中即可。在安全性要求不高的情况下可以允许这种操作，这样做可以提高进程间通讯的效率。

8.17 Compare paging with segmentation with respect to how much memory the address translation structures require to convert virtual addresses to physical addresses.

从内存地址转换程序（从逻辑地址转换到物理地址）需要花费的内存开销角度，比较分页和分段分配的特点。

答：分页法需要更多的内存空间进行转换。
分段法对于每段内存只需要两个寄存器：一个记录段基址，另一个记录段长度。
而分页法需要给每一个页建立一个条目。

8.19 Program binaries in many systems are typically structured as follows. Code is stored starting with a small, fixed virtual address, such as 0. The code segment is followed by the data segment that is used for storing the program variables. When the program starts executing, the stack is allocated at the other end of the virtual address space and is allowed to grow toward lower virtual addresses. What is the significance of this structure for the following schemes?

a. Contiguous memory allocation
b. Pure segmentation
c. Pure paging

很多系统的二进制程序文件通常有如下的结构：代码段被存储在以一个固定的逻辑地址开头的位置，比如 0。代码段后面紧跟着数据段，存储要操作的数据。当程序开始执行时，栈从程序逻辑地址的另一头（较大的那头）开始分配内存给程序，直到较小的一头为止。对于以下内存分配方案，这种结构有什么意义？

a. 连续内存分配　　b. 纯分段分配　　c. 纯分页分配

答：

a. 因为在连续内存分配方法中，在程序开始执行时系统就要为整个程序分配它的内存地址，所以分配的内存大小可能远大于实际需要的。
b. 纯分段技术可以在程序启动时分配一些小段的内存空间，随着运行的深入递增式地分配。
c. 纯分页技术不需要在程序启动时分配一大块内存空间，但是需要分配一个映射了全部物理内存地址的分页表。而当要分配新页时，这个分页表可能还需要更新。

8.20 Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

a. 3085
b. 42095
c. 215201
d. 650000
e. 2000001

假设页大小为 1KB，求下列逻辑地址对应的页号和偏移量（地址为十进制数字）

答：

Page number	Offset
3085/1024 = 3	3085 mod 1024 = 13
42095/1024 = 41	42095 mod 1024 = 111
215201/1024 = 210	215201 mod 1024 =161
650000/1024 = 634	650000 mod 1024 =784
2000001/1024 = 1953	2000001 mod 1024 = 129

8.22 What is the maximum amount of physical memory?

分析物理内存的最大值受什么因素影响?

答:

假设一台电脑的物理地址长度为 n 比特，那么它最多有2^n个可址单元。所以，物理内存最大有 $(2^n * 字长)$ 字节。

8.23 Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames.

a. How many bits are required in the logical address?

b. How many bits are required in the physical address?

考虑一个拥有 256 个页，每个页大小为 4KB 的逻辑地址空间，其映射到有 64 个帧的物理内存。

a. 逻辑地址至少需要多少位?

b. 物理地址至少需要多少位?

答:

a. log256 + log(4K) = 20 bits

b. log64 + log(4K) = 18 bits (Note: page size = frame size)

8.24 Consider a computer system with a 32-bit logical address and 4-KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?

考虑一个有 32 位逻辑地址和 4KB 大小页表的系统，系统的最大物理内存是 512MB。对于下列页表实现方式，页表中各有多少个条目?

a. 采用单层页表

b. 采用倒置页表

答：

a.采用单层页表（按照逻辑地址的最大值计算）：

$$\text{Size of logical address space} = 2^m = \#\text{ of pages} \times \text{page size}$$

$$2^{32} = \#\text{ of pages} \times 2^{12}$$

$$\#\text{ of pages} = 2^{32} / 2^{12} = 2^{20}\text{ pages}$$

b. 采用倒置页表（按照物理地址的最大值计算）：

$$\text{Size of physical address space} = \#\text{ of frames} \times \text{frame size}$$

$$2^{29} = \#\text{ of frames} \times 2^{12}$$

$$\#\text{ of frames} = 2^{29} / 2^{12} = 2^{17}\text{ pages}$$

8.25 Consider a paging system with the page table stored in memory.

a. If a memory reference takes 50 nanoseconds, how long does a paged memory reference take?

b. If we add TLBs, and 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes 2 nanoseconds, if the entry is present.)

考虑一个页表存储在内存中的分页系统：

a. 如果访问一个内存引用要 50ns，则引用一个页表要多长时间？

b. 如果使用 TLB，TLB 命中率（能在 TLB 中找到页表引用的概率）为 75%，从 TLB 中查找页表条目只需 2ns，则此时有效访问时间是多少？

答：

a. 访问时间 = 50 + 50 = 100 ns （50ns 访问内存中的分页表，50ns 访问实际地址）

b. 有效访问时间 = 0.75 * (50 + 2) + 0.25 * (50 + 2 + 50) = 64.5 ns

（+2是在TLB中查找页面的时间，无论是否查找到，都需要2ns）

8.26 Why are segmentation and paging sometimes combined into one scheme?

为什么分段经常和分页组合使用?

答：它们可以互相促进。比如，当分页表过大时，我们可以对分页表分段，将分页表上的一段连续空闲入口词条（entry）当作一段。类似的，我们也可以先分段再分页，这样做不仅提高了分配的效率，更加解决了分段法产生的外部碎片问题。

8.27 Explain why sharing a reentrant module is easier when segmentation is used than when pure paging is used.

解释一下为什么分享一个允许进程重入的内存块，分段分配时要比分页分配时简单。

答

共享段（segment）时，无论段的大小，只需要共享一个段基址即可。而共享页时，需要共享所有（要共享的）页在页表上的条目。

8.28 Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

a. 0,430
b. 1,10
c. 2,500
d. 3,400
e. 4,112

参考上图中的分段表，计算 a-e 五个逻辑地址对应的物理地址。

答：

a. 219 + 430 = 649
b. 2300 + 10 = 2310
c. 非法访问，操作系统进入陷阱（trap）
d. 1327 + 400 = 1727
e. 非法访问，操作系统进入陷阱（trap）

8.29 What is the purpose of paging the page tables?
为什么要对页表也进行分页处理?

答：当单个的页表过大时，对页表分页可以简化内存分配问题（因为这样可以按页分配页表，而不是长度不定的数据块）。同时，也可以交换（swap）部分暂时不用的页表。总而言之，可以大幅提高效率。

8.31 Compare the segmented paging scheme with the hashed page table scheme for handling large address spaces. Under what circumstances is one scheme preferable to the other?

比较分段分配和使用哈希页表进行分页分配，在处理较大的地址空间时的优缺点。什么情况下应该使用前者，什么情况下应该使用后者?

答：如果程序占用虚拟内存空间不大，用哈希页表更好，因为它的大小更小。但哈希页表也要用链表解决冲突问题。当页表过大时，遍历每个哈希值的链表会产生巨大的开销。

第九章 虚拟内存

实践练习部分 Practice Exercise

9.1 Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.

缺页错误时如何产生的？操作系统在缺页错误时会如何处理？

答：当某个指令访问到当前不在内存中的页时，会发生缺页错误（page fault）。发生缺页错误之后，操作系统会检查这次访问是否合法：如果不合法，终止程序；如果合法，则将要访问的页读入到内存中的空闲页（free frame）中。随后，操作系统更新进程表和页表，然后重新执行这个指令。

9.2 Assume that you have a page-reference string for a process with *m* frames (initially all empty). The page-reference string has length *p*; *n* distinct page numbers occur in it. Answer these questions for any page-replacement algorithms:

a. What is a lower bound on the number of page faults?

b. What is an upper bound on the number of page faults?

假设你有一串需要 m 个帧的页面引用（初始都为空）。页面引用串的长度为 p，其中不同的页面个数有 n 个。回答下列问题：

a. 产生缺页错误数目的最小值是多少？

b. 产生缺页错误数目的最大值是多少？

答：

a. *n*

b. *p*

9.3 Consider the page table shown in Figure 9.30 for a system with 12-bit virtual and physical addresses and with 256-byte pages. The list of free page frames is *D*, *E*, *F* (that is, *D* is at the head of the list, *E* is second, and *F* is last).

Convert the following virtual addresses to their equivalent physical addresses in hexadecimal. All numbers are given in hexadecimal. (A dash for a page frame indicates that the page is not in memory.)

考虑下图展示的页表，逻辑地址的长度为 12 位，页表大小为 256B。当前的空闲帧有 D,E,F（在空闲帧表中即按照这个顺序排列）。将以下逻辑地址（十六进制）转化为物理地址（以十六进制的形式）。

- 9EF
- 111
- 700
- 0FF

Page	Page Frame
0	–
1	2
2	C
3	A
4	–
5	4
6	3
7	–
8	B
9	0

Figure 9.30 Page table for Exercise 9.3.

答：

- 9EF -> 0EF
- 111 -> 211
- 700 -> D00（需要先调入页 D）
- 0FF -> EFF （需要先调入页 E）

9.5 Discuss the hardware support required to support demand paging.
分析请求调页需要硬件的哪些支持?

答：请求调页（demand paging）和分页（paging）需要的硬件支持是一样的：我们需要一个页表，和二级存储（secondary memory）（用来存储不在内存中的页）。每次访问内存时，都需要访问页表以检查对应的页是否在内存中，以及该程序是否具有这一页的读写权限。这些检查都必须在硬件层面上实现。我们也可以使用 TLB 来提升查找的性能。

9.6 An operating system supports a paged virtual memory. The central processor has a cycle time of 1 microsecond. It costs an additional 1 microsecond to access a page other than the current one. Pages have 1,000 words, and the paging device is a drum that rotates at 3,000 revolutions per minute and transfers 1 million words per second. The following statistical measurements were obtained from the system:

- One percent of all instructions executed accessed a page other than the current page.
- Of the instructions that accessed another page, 80 percent accessed a page already in memory.
- When a new page was required, the replaced page was modified 50 percent of the time.

Calculate the effective instruction time on this system, assuming that the system is running one process only and that the processor is idle during drum transfers.

一个操作系统支持分页形式的虚拟内存。CPU 时钟周期是 1ms，访问一个非当前的页面需要 1ms。页面大小为 1000 字长。被分页的存储设备是一个转速 3000r/min，每秒能传输 100 万字长的磁鼓。统计测量得到以下数据：

- 有 1%的指令需要访问一个非当前的页面
- 80%的对非当前页面的访问，其访问的是内存中已经有的页面
- 有 50%的概率在更换新页面的时候，被换走的页面会被修改。

根据以上数据，计算系统的有效指令执行时间（假设系统只有一个 CPU 核，在磁鼓传输数据时，CPU 处于空闲状态）

答：

鼓（drum）的遍历速度 = 3000 rev/min = 50 rev/s
遍历一圈所需时间 = 1 s / 50 rev/s = 0.02 s = 20,000 μs
访问一个不在内存中的页所需平均时间 = 20,000 μs / 2 = 10,000 μs
读写一个页所需时间 = 1,000 words / 10^6 words/s = 1,000 μs
访问当前页所需时间 = 1 μs
访问在内存中的非当前页所需时间 = 1 μs + 1 μs = 2 μs

有效访问时间 = 0.99 * 1 μs // 访问当前页
+ 0.008 * 2 μs // 访问内存中的其它页
+ 0.001 * (10,000 μs + 1,000 μs) // 访问不在内存中的页（不用置换）
+ 0.001 * (20,000 μs + 2,000 μs) // 访问不在内存中的页（需要置换）
= 34.006 μs

9.7 Consider the two-dimensional array A:

```
int A[][] = new int[100][100];
```

where A[0][0] is at location 200 in a paged memory system with pages of size 200. A small process that manipulates the matrix resides in page 0 (locations 0 to 199). Thus, every instruction fetch will be from page 0.

For three page frames, how many page faults are generated by the following array-initialization loops? Use LRU replacement, and assume that page frame 1 contains the process and the other two are initially empty.

考虑这样一个二维数组 A：int A[ ][ ] = new int[100][100]; 假设 A[0][0]位于分页系统的第 200 页，页面大小也为 200。一个简单的计算该矩阵的小程序储存在第 0 页中，这意味着，每次取指令时都会从页面 0 开始。

假设有 3 个物理帧，使用下面的两种数组初始化算法，分别会产生多少缺页错误？页面置换算法使用 LRU，并且物理帧 1 中已经存储了程序，而另外两个物理帧中是空的。

a.
```
for(intj=0;j<100;j++)
    for (int i = 0; i < 100; i++)
        A[i][j] = 0;
```

b.
```
for(inti=0;i<100;i++)
    for (int j = 0; j < 100; j++)
        A[i][j] = 0;
```

答：

a. 100 * 50 = 5000
b. 100 / 2 = 50

解释：对于b来说，j每次直接访问200个页面
所以i说是从0到99 其实只要加载50次缺页中断
反之，对于a，在每次j循环的中i要产生50次缺页错误，所以a要缺页中断100*50次

9.8 Consider the following page reference string:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, and seven frames? Remember that all frames are initially empty, so your first unique pages will cost one fault each.

根据上面写出的页面引用串，计算在帧数分别为 1-7 的情况下缺页错误发生的次数，所有的帧在一开始都是空的，调页算法分别采取下列的计算：

- LRU replacement
- FIFO replacement
- Optimal replacement

答：

页帧数		1	2	3	4	5	6	7
缺页错误数	LRU	20	18	15	10	8	7	7
	FIFO	20	18	16	14	10	10	7
	Opt.	20	15	11	8	7	7	7

习题部分 Exercise

9.14 Assume that a program has just referenced an address in virtual memory. Describe a scenario in which each of the following can occur. (If no such scenario can occur, explain why.)

假设一个程序刚刚引用了在虚拟内存中的一个地址。判断下面的情况是否可能发生，如果不可能，解释原因。

- TLB miss with no page fault
- TLB miss and page fault
- TLB hit and no page fault
- TLB hit and page fault

答：

前三者都有可能发生，后者不可能。因为无论要访问的页是否在内存中，都有可能发生 TLB 缺失（TLB miss），而如果 TLB 命中（TLB hit），就说明要访问的页就一定在内存中，那就不可能发生缺页中断（page fault）。

9.15 A simplified view of thread states is *Ready*, *Running*, and *Blocked*, where a thread is either ready and waiting to be scheduled, is running on the processor, or is blocked (for example, waiting for I/O). This is illustrated in Figure 9.31. Assuming a thread is in the Running state, answer the following questions, and explain your answer:

线程的三个基本状态是 ***准备***、***运行中***和***阻塞***，其分别代表一个线程等待被调度、正在 CPU 里运行和被如 I/O 操作等的情况阻塞。线程状态之间的转换如下图。假设一个线程正处于运行中的状态，回答下列问题：

a. Will the thread change state if it incurs a page fault? If so, to what state will it change?
b. Will the thread change state if it generates a TLB miss that is resolved in the page table? If so, to what state will it change?
c. Will the thread change state if an address reference is resolved in the page table? If so, to what state will it change?

a. 如果一个线程引发了缺页错误，会导致线程状态改变吗？如果会，它的状态会变成什么？
b. 如果一个线程遇到了 TLB 未命中的情况，会导致线程状态改变吗？如果会，它的状态会变成什么？
c. 如果一个线程引用的地址已经在页表中，会导致线程状态改变吗？如果会，它的状态会变成什么？

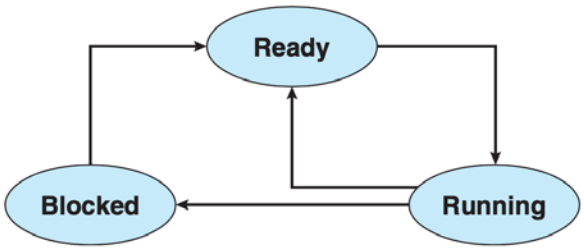


Figure 9.31 Thread state diagram for Exercise 9.15.

答：

a. 它会进入阻塞（blocked）状态，因为需要等待 I/O 操作将要访问的页读入内存。
b. 它不会改变状态。只要能在页表中找到，它就可以继续运行。
c. 同上。

9.17 What is the copy-on-write feature, and under what circumstances is its use beneficial? What hardware support is required to implement this feature?

什么是写时复制？在什么时候写时复制会是有益的？实现写时复制需要什么硬件支持？

答：当两个进程访问同一块数据时（比如说代码段和可执行文件）时，可以将要访问的地址以写保护（write protected）的方式映射到两个进程的虚拟内存中。当进程需要写入时，我们则必须将那些数据复制一份，这样才能保证两个进程访问的数据相互独立。
它所需要的硬件支持很简单：每次要写入内存时，都要访问页表来确定某一页是否是写保护的。如果是，则发出系统中断（trap），等待这一页被复制好了之后再写入。

9.18 A certain computer provides its users with a virtual memory space of 2^32 bytes. The computer has 2^22 bytes of physical memory. The virtual memory is implemented by

paging, and the page size is 4,096 bytes. A user process generates the virtual address 11123456. Explain how the system establishes the corresponding physical location. Distinguish between software and hardware operations.

有一个计算机提供2^{32}B 的虚拟内存，物理内存有2^{22}B，虚拟内存采取分页的方式，页大小为 4096B，假设一个用户需要访问逻辑地址 11123456，解释系统怎么确定对应的物理位置，在这个过程中注意区分硬件和软件部分的不同侧重点。

答：

在虚拟内存中：

页大小 = 2^12 字节

页数 = 2^20

将十进制地址 11123456 转化为二进制：

0001 0001 0001 0010 0011 0100 0101 0110

一共有 32 位。显然，其中前 20 位是指向某一页的（页码），后 12 位则指向某一页中具体的某一字节（偏移量）。硬件负责根据页表实时将虚拟地址翻译成物理地址；软件负责处理错误页，还有将虚拟的内存页按需读入物理内存中等操作。

9.19 Assume that we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified and 20 milliseconds if the replaced page is modified. Memory-access time is 100 nanoseconds.

Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?

假设现在有一个需要调页的内存，页表存储于寄存器中。当发生缺页错误要换页时，如果当前有空的物理帧或者被换走页面没有被修改，处理缺页错误要 8ms，如果换走的页面被修改过，那么处理缺页错误要 20ms。访问一次内存空间需要 100ns。

假设换走的页面中有 70%时被修改过的，现在要求有效访问时间不超过 200ns，则能够接收的最大缺页错误率是多少？

答：

与 9.3 同理，我们假设缺页中断率（page fault rate）为 p，那么，

有效访问时间 = (1 – p) * 0.1 μs

+ p * (1 – 0.7) + (8,000 μs)

+ p * 0.7 * 20,000 μs

= 32800p μs

由 32800p μs < 0.2 μs 得

p < 0.000006

9.21 Consider the following page reference string:

7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0 , 1.

Assuming demand paging with three frames, how many page faults would occur for the following replacement algorithms?

对以下页面置换算法，按照上面的页面引用串，一共会发生多少缺页错误？假设物理帧的数量为 3。

- LRU replacement
- FIFO replacement
- Optimal replacement

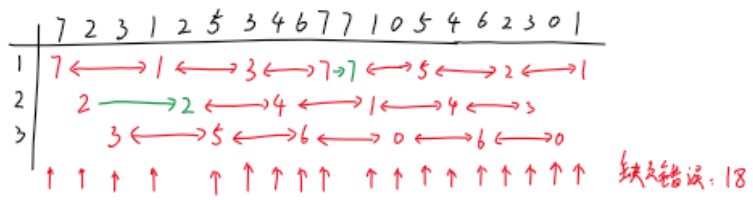
答：

LRU 置换：18

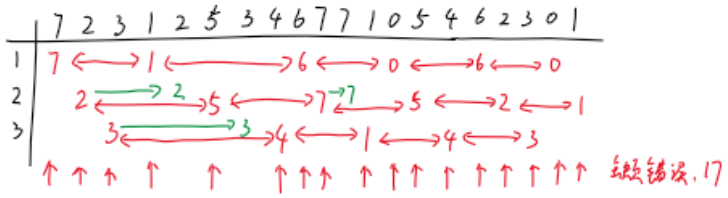
FIFO 置换：17

最优置换：13

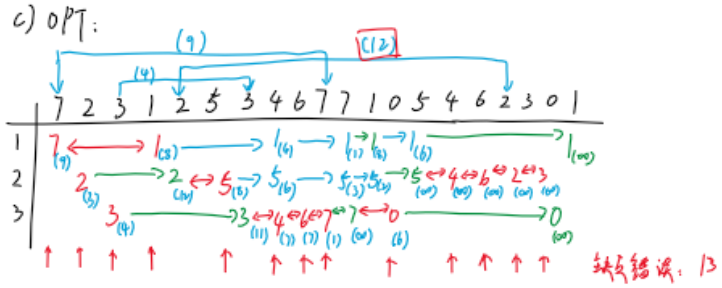
a) LRU：



b) FIFO



c) OPT：



9.22 The page table shown in Figure 9.32 is for a system with 16-bit virtual and physical addresses and with 4,096-byte pages. The reference bit is set to 1 when the page has been referenced. Periodically, a thread zeroes out all values of the reference bit. A dash for a page frame indicates the page is not in memory. The page-replacement algorithm is localized LRU, and all numbers are provided in decimal.

a. Convert the following virtual addresses (in hexadecimal) to the equivalent physical addresses. You may provide answers in either

根据下图所示的页表，页面大小为 4096B，逻辑地址为 16 位，当页面被引用时，其对应引用位（reference bit）会被设置为 1。现在使用 LRU 页面置换算法，所有的数字使用十六进制，回答下列问题：

a. 将下列逻辑地址转化为物理地址，结果可以用十六进制或者十进制来表示，并且改变对应页表条目中引用位的值

Page	Page Frame	Reference Bit
0	9	0
1	1	0
2	14	0
3	10	0
4	–	0
5	13	0
6	8	0
7	15	0
8	–	0
9	0	0
10	5	0
11	4	0
12	–	0
13	–	0
14	3	0
15	2	0

Figure 9.32 Page table for Exercise 9.22.

hexadecimal or decimal. Also set the reference bit for the appropriate entry in the page table.

- 0xE12C
- 0x3A9D
- 0xA9D9
- 0x7001
- 0xACA1

b. Using the above addresses as a guide, provide an example of a logical address (in hexadecimal) that results in a page fault.

c. From what set of page frames will the LRU page-replacement algorithm choose in resolving a page fault?

b. 使用上面问题的结果作为参考，给出一个会产生缺页错误的逻辑地址（用十六进制表示）

c. 如果产生缺页错误，LRU 算法会从哪些帧上选择一帧来进行调页?

答：

a. 页大小 = 2^12 字节
页数 = 2^16 – 2^12 = 2^4 = 16
因此，十六进制地址第一位为页号。所以，
- 0xE12C -> 0x312C

- 0x3A9D -> 0xAA9D
- 0xA9D9 -> 0x59D9
- 0x7001 -> 0xF001
- 0xACA1 -> 0x5CA1

页面 E,3,A,7 （对应页帧3，10，5，15 ）的引用位会被设置为 1

b. 当引用页表中的 4,8,C,D 四个页面时就会发生缺页错误，比如地址 0x4001
c. 由 a.知，页帧 3，10，5，15 的访问位（referenced bit）为 1.所以，LRU 会从 9，1，14，13，8，0，4，2 这几个页帧中选一个进行置换。

9.24 Discuss situations in which the least frequently used (LFU) page-replacement algorithm generates fewer page faults than the least recently used (LRU) page-replacement algorithm. Also discuss under what circumstances the opposite holds.

分析在什么情况下最不经常使用（LFU）置换算法比最近最少使用（LRU）算法有更少的缺页错误，并分析什么时候相反。

答：

假设实际内存大小为 4，那么，

- LFU 更好的访问序列：1，1，2，3，4，5，1
- LRU 更好的访问序列：1，1，2，3，4，5，2

9.25 Discuss situations in which the most frequently used (MFU) page-replacement algorithm generates fewer page faults than the least recently used (LRU) page-replacement algorithm. Also discuss under what circumstances the opposite holds.

分析在什么情况下最经常使用（MFU）置换算法比最近最少使用（LRU）算法有更少的缺页错误，并分析什么时候相反。

答：

假设实际内存大小为 4，那么，

- MFU 更好的访问序列：1，2，2，3，4，5，1
- LRU 更好的访问序列：1，2，2，3，4，5，2

9.27 Consider a demand-paging system with the following time-measured utilizations:

CPU utilization 20%
Paging disk 97.7%
Other I/O devices 5%

For each of the following, indicate whether it will (or is likely to) improve CPU utilization. Explain your answers.

a. Install a faster CPU.
b. Install a bigger paging disk.
c. Increase the degree of multiprogramming.
d. Decrease the degree of multiprogramming.
e. Install more main memory.
f. Install a faster hard disk or multiple controllers with multiple hard

disks.

g. Add prepaging to the page-fetch algorithms.

h. Increase the page size.

假设一个请求调页的系统各个部分的利用率如下：

CPU 利用率　20%

分页的磁盘　97.7%

其他 I/O 设备　5%

对下面的几种情况，判断其是否能提升该系统的 CPU 利用率，解释你的答案。

a. 更换速度更快的 CPU

b. 增大磁盘的大小

c. 增大多道程序程度

d. 减少多道程序程度

e. 增大主要内存

f. 更换更快的磁盘或使用多个磁盘

g. 在请求调页算法中加入预分页算法。

h. 增加页面大小

答：

会提升性能的有：d，e，f，g

不会提升性能的有：a，b，c，h

a. 因为现在限制性能的，不是 CPU（因为它的占用率很低），而是虚拟内存和实际内存大小的不平衡所导致的交换（swapping）压力过大（系统已经接近抖动状态）。因此，提升 CPU 速率并不可行。

b. 增大分页磁碟（paging disk）大小，增加的是二级存储空间的大小，可以帮助提升多任务处理度（degree of multiprogramming），对解决抖动只能起反作用。

c. 增大多任务处理度只会加重交换（swapping）的压力，对解决抖动并无帮助。

d. 同上理，减少多道程序程度可以缓解抖动。

e. 加大内存大小也可以减轻交换的压力，有助于提升 CPU 利用率。

f. 更快的磁碟读写速度意味着更快的交换，因此有助于提升 CPU 利用率。

g. 提前完成交换，有助于提升 CPU 利用率。

h. 对于顺序访问（sequential access）的进程，更大的页面可以提升性能。但是鉴于目前交换压力较大，很可能是因为进程访问的地址比较分散，这种情况下，更大的页意味着物理内存中能容纳的页帧数更少，反而不利于提升性能。

9.30 A page-replacement algorithm should minimize the number of page faults. We can achieve this minimization by distributing heavily used pages evenly over all of memory, rather than having them compete for a small number of page frames. We can associate with each page frame a counter of the number of pages associated with that frame. Then, to replace a page, we can search for the page frame with the smallest counter.

a. Define a page-replacement algorithm using this basic idea. Specifically address these problems:

i. What is the initial value of the counters?

ii. When are counters increased?

iii. When are counters decreased?

iv. How is the page to be replaced selected?

b. How many page faults occur for your algorithm for the following reference string with four page frames?

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

c. What is the minimum number of page faults for an optimal pagereplacement strategy for the reference string in part b with four page frames?

换页算法的目的时尽可能地减少缺页错误。为了达到这一目的，我们可以将经常使用的页面平均分布到内存中，而不是让它们不停的竞争换页。将每个帧与一个计数器进行绑定，计数器计算映射到这个帧的页面数量，这样在需要换页的时候，我们只需搜索帧表，然后找到页面计数最小的页替换即可。

a. 使用上面的思路设计一个换页算法，并详述下面的几个问题该如何解决：

i. 计数器的初始值应该是多少？

ii. 什么时候计数器+1？

iii. 什么时候计数器-1？

iv. 如何选择被替换的页？

b. 对于下面的页面引用串，你设计的算法缺页错误数是多少？

c. 对于 b 中的引用串，使用 OPT 算法的最小缺页错误数是多少？

答：

a. i. 初始值为 0

ii. 当有一个新页跟当前页建立关系时

iii. 当有一个跟当前页有关系的页不再被需要时

iv. 选择 counter 最小的页。如果 counter 相同，使用 FIFO 算法。

b. 14

c. 11

9.31 Consider a demand-paging system with a paging disk that has an average access and transfer time of 20 milliseconds. Addresses are translated through a page table in main memory, with an access time of 1 microsecond per memory access. Thus, each memory reference through the page table takes two accesses. To improve this time, we have added an associative memory that reduces access time to one memory reference if the page-table entry is in the associative memory.

Assume that 80 percent of the accesses are in the associative memory and that, of those remaining, 10 percent (or 2 percent of the total) cause page faults. What is the effective memory access time?

假设一个请求调页的系统，磁盘访问和数据传输时间为 20ms。页表存储在主要内存中，一次内存访问需要 1ms。因此一次通过查找页表的内存引用需要两次访问内存。为了提高效率，我们增加一个辅助内存，如果要查找的页表的条目已经在辅助内存里了，那么只需要访问一次内存。

假设有 80%的访问请求都能在辅助内存中找到，在剩下的部分中有 10%会造成缺页错误（也就是总缺页错误率 2%），求有效访问时间。

答：

有效访问时间 = 0.8 * 1 μs + 0.18 * 2 μs + 0.02 * (20,000 μs + 2 μs) = 401.2 μs

***解释**：“辅助内存”的机制即和 TLB 是相同的，只不过这里辅助内存并没有提升访问速度，只是减少了访问内存的次数。*

9.32 What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?

抖动产生的原因是什么？系统如何检查抖动？如果检测到抖动，系统如何解决？

答：
抖动产生的原因，是因为没有给进程分配足够的内存页帧。
系统可以通过监控缺页中断率，或者通过比较多道程序程度和 CPU 占用率，来侦测抖动。 抖动可以通过降低多道程序程度来解决。

9.33 Is it possible for a process to have two working sets, one representing data and another representing code? Explain.

一个进程可能拥有两个工作集吗？一个用来监控代码段，另一个用来监控数据段。解释理由。

答：
当然可能。事实上，很多处理器为了实现这个功能会提供两个 TLB。有时，一个程序的代码工作集不变，但是数据工作集会时常改变。

第十章 大容量存储

实践练习部分 Practice Exercise

10.1 Is disk scheduling, other than FCFS scheduling, useful in a single-user environment? Explain your answer.

非 FCFS 的磁盘调度在单用户环境下会更好吗？解释理由。

答：在单用户系统中，一般 I/O 请求队列不会积压，所以使用 FCFS 调度是很划算的。但 LOOK 调度的实现并不比 FCFS 复杂多少，并且多任务处理的表现更好。比如当浏览器正在后台下载文件，操作系统在置换页，并且还有其它应用程序在前台展现时。

10.2 Explain why SSTF scheduling tends to favor middle cylinders over the innermost and outermost cylinders.

解释为什么 SSTF 调度会跟倾向于磁盘中部的磁道而非两端的磁道。

答：因为磁盘中部距离任意其它位置的距离之和的平均是最短的。换句话说，假设当前磁头位置不在磁盘中部，对于一个随机分布的新请求，其位于在磁头之包含磁盘中部的一侧的可能性更大:

```
[          ^                    |                              ]
[          |                    |                              ]
         head                 center
```

10.3 Why is rotational latency usually not considered in disk scheduling? How would you modify SSTF, SCAN, and C-SCAN to include latency optimization?

为什么在磁盘调度算法中通常不考虑考虑旋转延迟？如果考虑旋转延迟，如何优化 SSTF,SCAN 和 C-SCAN 算法？

答：因为大多数磁盘不会将旋转速度返回给系统，而且就算将这个信息返回了，考虑旋转延迟的调度器进行计算的时间复杂度会很大，待调度器计算完成磁盘状态已经改变了，这就导致考虑旋转延迟没有作用。如果要考虑针对旋转延迟的优化，就必须在控制磁盘的硬件层面上实现这些调度算法，而不是用软件算法来控制调度。

10.4 Why is it important to balance file-system I/O among the disks and controllers on a system in a multitasking environment?

为什么要平衡磁盘控制器和文件系统的 I/O 请求，特别是当系统处于多任务状态时？

答：根据木桶原理，磁盘和磁盘控制器较慢的访问速度是限制计算机整体效率的瓶颈。所以均衡地调度 I/O 请求可以避免产生这种瓶颈。

习题部分 Exercise

10.9 None of the disk-scheduling disciplines, except FCFS, is truly fair (starvation may occur).

有一种说法称：磁盘调度算法中，只有 FCFS 是绝对公平的，尽管可能发生饥饿。

a. Explain why this assertion is true.
b. Describe a way to modify algorithms such as SCAN to ensure fairness.
c. Explain why fairness is an important goal in a time-sharing system.
d. Give three or more examples of circumstances in which it is important that the operating system be unfair in serving I/O requests.

a. 为什么这个说法是正确的
b. 提出一种改进 SCAN 算法的方法，让其更加公平
c. 解释为何在分时系统中公平的算法至关重要
d. 举出其他三个例子说明什么时候操作系统需要对 I/O 请求进行不公平调度。

答：

a. 显然成立。
b. C-SCAN（Circular SCAN）调度比 SCAN 更公平，并且不会产生饥饿。
c. 在分时系统中，每个请求都应该在一定的时限内被处理。
d. 1）换页（paging）和请求调页（swapping）时，
   2）写入系统元数据（metadata）时，

3）处理实时进程时

10.11 Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4,999. The drive is currently serving a request at cylinder 2,150, and the previous request was at cylinder 1,805. The queue of pending requests, in FIFO order, is:

2,069, 1,212, 2,296, 2,800, 544, 1,618, 356, 1,523, 4,965, 3681

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?

假设一个硬盘驱动器有 5000 个磁道，编号 0 到 4999。磁头目前处于 2150 磁道，上一次请求的位置在 1805 磁道，等待队列里的后续请求位置如下，队列采用先进先出的顺序。

从当前的位置开始，使用以下磁盘调度算法完成所有队列中的请求，求得的磁头的总移动长度（以磁道数为单位）是多少？

a. FCFS
b. SSTF
c. SCAN
d. LOOK
e. C-SCAN
f. C-LOOK

答：

算法	FCFS	SSTF	SCAN	LOOK	C-SCAN	C-LOOK
移动总长	13011	7586	7492	7424	9917	9137

具体计算过程和解释如下图：

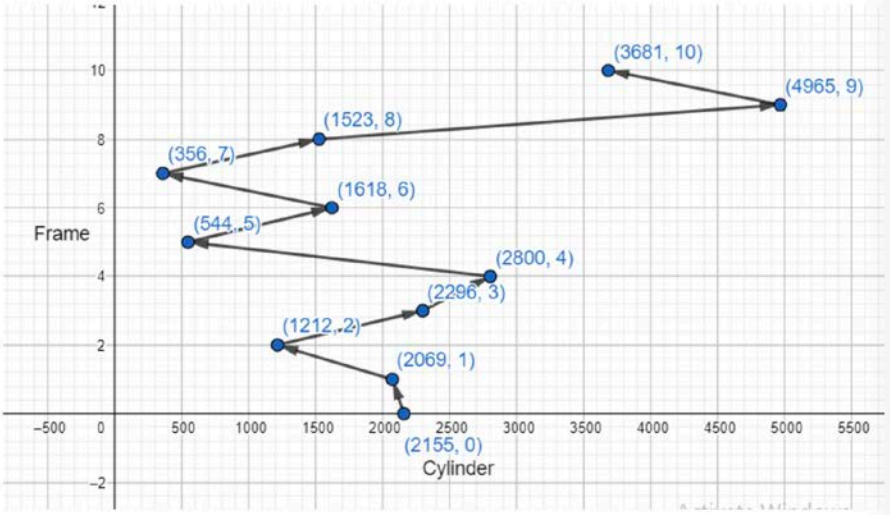
Part (a) FCFS

FCFS stands for First come First Serve. This is a very simple scheduling algorithm. It performs according to the request order.

$$\text{FCFS} = \left\{ \begin{array}{l} |2150-2069| + |2069-1212| + |1212-2296| \\ +|2296-2800| + |2800-544| + |544-1618| \\ +|1618-356| + |356-1523| + |1523-4965| + |4965-3681| \end{array} \right\}$$

$$\text{FCFS} = 81 + 857 + 1084 + 504 + 2256 + 1074 + 1262 + 1167 + 3442 + 1284$$

$$\boxed{\text{FCFS} = 13011}$$



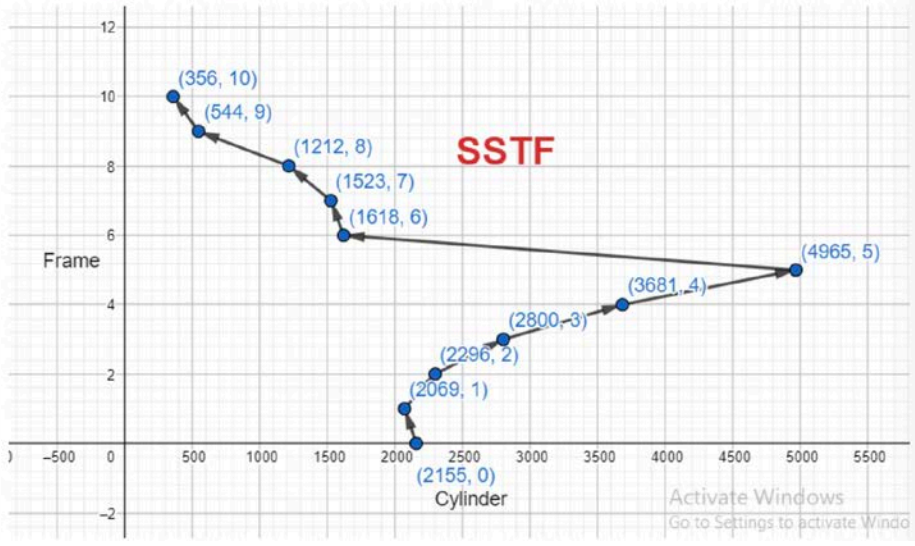
Part (b) SSTF

In SSTF disk scheduling algorithm, request is selected with the minimum seek time from the current position of head.

$$\text{SSTF} = \left\{ \begin{array}{l} |2150-2069| + |2069-2296| + |2296-2800| + |2800-3681| \\ +|3681-4965| + |4965-1618| + |1618-1523| \\ +|1523-1212| + |1212-544 + |544-356|| \end{array} \right\}$$

$$\text{SSTF} = 81 + 227 + 504 + 881 + 1284 + 3347 + 95 + 311 + 668 + 188$$

$$\boxed{\text{SSTF} = 7586}$$



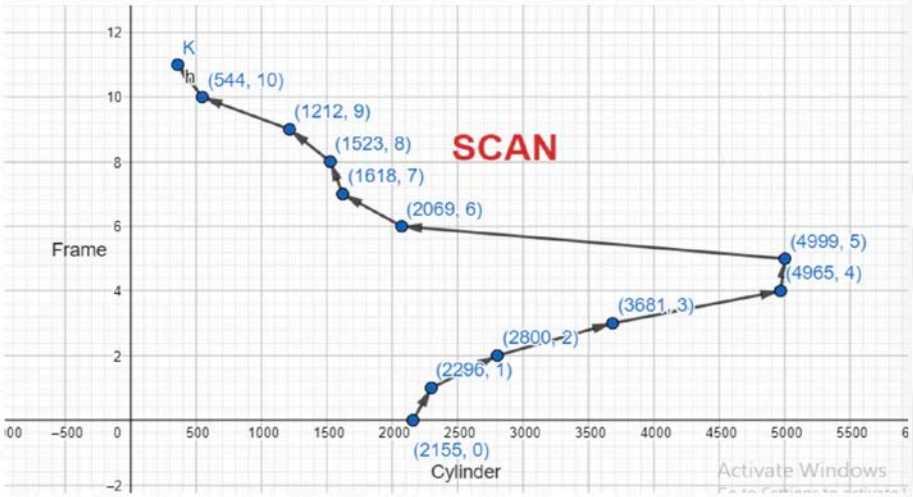
Part (c) SCAN

The algorithm is known as Elevator algorithm. Its head goes from one end to another and then reverses.

$$\text{SCAN} = \left\{ \begin{array}{l} |2150-2296| + |2296-2800| + |2800-3681| + |3681-4965| + |4965-4999| \\ +|4999-2069| + |2069-1618| + |1618-1523| + |1523-1212| + |1212-544| \\ +|544-356| \end{array} \right\}$$

$$\text{SCAN} = 146 + 504 + 881 + 1284 + 34 + 2930 + 451 + 95 + 311 + 668 + 188$$

$\boxed{\text{SCAN} = 7492}$



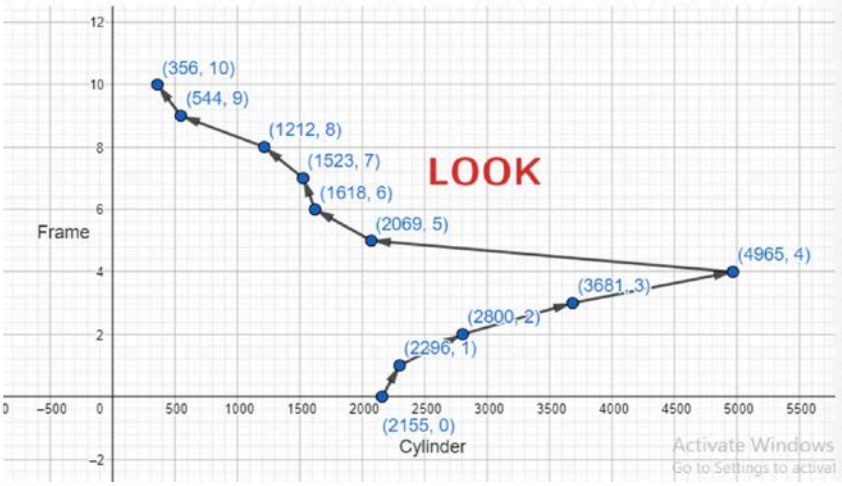
Part (d) LOOK

LOOK Algorithm is an improved version of the SCAN Algorithm. Head starts from the first request at one end of the disk and moves towards the last request at the other end servicing all the requests in between. After reaching the last request at the other end, head reverses its direction.

$$\text{LOOK} = \left\{ \begin{array}{l} |2150-2296| + |2296-2800| + |2800-3681| + |3681-4965| \\ +|4965-2069| + |2069-1618| + |1618-1523| + |1523-1212| + |1212-544| \\ +|544-356| \end{array} \right\}$$

$$\text{LOOK} = 146 + 504 + 881 + 1284 + 1284 + 2896 + 451 + 95 + 311 + 668 + 188$$

$\boxed{\text{LOOK} = 7424}$



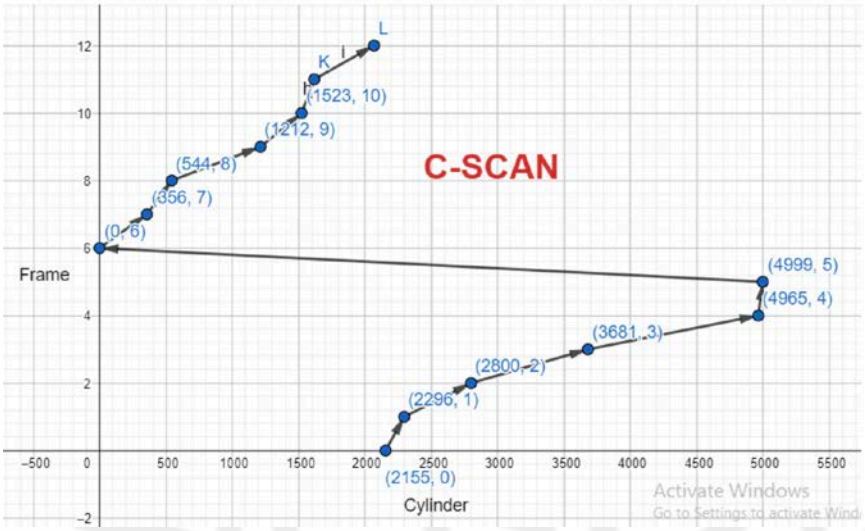
Part (e) C-SCAN

Circular-SCAN Algorithm is an improved version of the SCAN Algorithm. Head starts from one end of the disk and move towards the other end servicing all the requests in between. After reaching the other end, head reverses its direction. It then returns to the starting end without servicing any request in between. The same process repeats.

$$C-SCAN = \left\{ \begin{array}{l} |2150-2296|+|2296-2800|+|2800-3681|+|3681-4965| \\ +|4965-4999|+|4999-0|+|0-356|+|356-544|+|544-1212| \\ +|1212-1523|+|1523-1618|+|1618-2069| \end{array} \right\}$$

$$C-SCAN = 146+504+881+1284+34+4999+356+188+311+668+95+451$$

$$\boxed{C-SCAN = 9917}$$



第十一章 文件系统接口

实践练习部分 Practice Exercise

11.1 Some systems automatically delete all user files when a user logs off or a job terminates, unless the user explicitly requests that they be kept. Other systems keep all files unless the user explicitly deletes them. Discuss the relative merits of each approach.

有的系统会在用户任务全部结束后自动删除所有用户的缓存和日志文件，除非用户主动提出保留这些文件，而一些其他系统则一直保留这些文件，除非用户主动要求删除它们。分析这两种处理方法的优缺点。

答：删除文件有利于节省空间。保存文件可以防止用户因忘记保存文件而丢失文件。

11.2 Why do some systems keep track of the type of a file, while others leave it to the user and others simply do not implement multiple file types? Which system is "better"?

为什么有的系统为文件指定不同的类型并追踪，有的系统让用户确定文件类型，有的系统干脆就不实现多文件类型？这些方法中哪一个"更好"？

答：哪一种更好，取决于进程对文件的不同需求。

有些系统可以针对不同类型的文件进行不同的操作。而有些系统则把具体的文件操作交给进程和用户操作，在访问数据时不提供任何帮助。要根据用户的具体需求才能确定哪种操作“更好”。比如，对于提供数据库应用的操作系统来说，最好在操作系统层级上实现一些数据库文件的操作，这样就不需要每个应用程序自己（通常是用不同的方法）实现一遍重复的操作。而对于通用操作系统来说，提供一些最基本的文件类型操作就行了。这样做既节省空间，也为每个进程提供了较大的自由度。

11.3 Similarly, some systems support many types of structures for a file's data, while others simply support a stream of bytes. What are the advantages and disadvantages of each approach?

与文件类型的实现相似，有的系统支持很多不同的文件数据结构，而有的系统就只实现一个字节流数据结构。这两种方法又有何优缺点？

答：在系统层面提供不同文件类型的操作通常来说更高效，应用程序也因此变得更精简。这样做的缺点是它会使系统变得臃肿，而且需要使用到其它文件类型的应用程序可能就无法在这样的操作系统上运行了。
如果系统不定义文件类型，将所有文件都视为字节流（byte stream）的话（比如 UNIX 操作系统），系统将会变得更精简，也允许不同的应用程序实现自己的文件操作。

11.4 Could you simulate a multilevel directory structure with a single-level directory structure in which arbitrarily long names can be used? If your answer is yes, explain how you can do so, and contrast this scheme with the multilevel directory scheme. If your answer is no, explain what prevents your simulation's success. How would your answer change if file names were limited to seven characters?

可以在单一目录结构下实现多级目录结构的功能吗（文件名可以任意长）？如果可以，解释如何做到，如果不可以，解释理由。如果文件名限制在 7 个字符以内，还可以做到吗？

答：如果文件名的长度没有限制，那么使用单层目录（single directory）实现多级目录（multilevel directory）是很容易的。我们可以用“.”来作为“子目录”的分隔符。比如“用户.文件.作业一”就表示“作业一”是在“文件”这个子目录下的，而“文件”则在“用户”这个根目录下。

如果对文件名的长度有限制，那上述方法就行不通了。但是我们亦可以考虑使用长度固定的字符串作为索引，在另一个文件表中存储每个索引对应的（长度不限）的文件名。在系统中，文件名都以这种定长的索引的形式出现。（不过这种方式并不保证一直可行，特别是文件的数目超过限制以后）

11.5 Explain the purpose of the open() and close() operations.
解释 open()和 close()函数的作用。

答：

- open()系统调用告诉操作系统指定的文件即将被访问
- close()系统调用告诉操作系统之前由 open()操作打开的文件将不再被该进程访问

11.6 In some systems, a subdirectory can be read and written by an authorized user, just as ordinary files can be.

a. Describe the protection problems that could arise.

b. Suggest a scheme for dealing with each of these protection problems.

在一些系统中，子目录可以像普通的文件一样被有授权的用户读写。

a. 分析这种方式可能带来什么安全问题？

b. 提出一个方案来解决上述问题。

答：

a. 在每个目录词条（directory entry）中，都存储了该目录中文件的信息。如果用户可以像更改一个文件一样更改一个子目录的话，他就能修改子目录的目录词条，从而改变子目录中文件的访问权限，造成访问保护被破坏。

b. 禁止用户直接（像修改一个文件一样）修改子目录，而是提供相应的系统调用。

习题部分 Exercise

11.9 Consider a file system in which a file can be deleted and its disk space reclaimed while links to that file still exist. What problems may occur if a new file is created in the same storage area or with the same absolute path name? How can these problems be avoided?

考虑一种文件系统，其中文件被删除后其在磁盘中的数据被回收，而指向文件的链接仍然保留。这种机制在创建一个相同位置（相同绝对路径和文件名称）的文件时会引发什么问题？如何避免这种问题？

答：如果链接（link）指向的地方原来存储的是文件 A，文件 A 删除后不删除链接，后来那里被用来存储文件 B，那么我们就可以用文件 A 留下来的链接来访问文件 B，破坏了文件保护机制。

有两种方法可以解决这个问题：

a. 在删除一个文件时，找到与它相关的所有链接并删除。

b. 在打开一个链接时，先检查目标文件和链接创建时所应该指向的文件的信息是否相同，如果不相同，则不允许打开。

11.10 The open-file table is used to maintain information about files that are currently open. Should the operating system maintain a separate table for each user or maintain just one table that contains references to files that are currently being accessed by all users? If the same file is being accessed by two different programs or users, should there be separate entries in the open-file table? Explain.

打开文件表用来存储当前打开的所有文件的信息。操作系统应该为每个用户单独维护一个打开文件表，还是应该只维护一个打开文件表，其中包含所有用户的打开文件信息？如果一个文件被两个不同用户的进程同时访问，在打开文件表中需要创建两个不同的条目吗？解释原因。

答：给打开同一文件的不同的用户或进程在打开文件表（open file table）中创建不同的词条可以实现一些不这样做就无法实现的功能。比如说，如果多个进程正在访问同一个文

件，而其中一个进程提出删除文件的请求。这个时候我们可以检查是否有别的进程/用户仍在使用它，如果没有，则将其删除，如果有，我们可以拒绝这个请求，防止其它进程或用户出错。因此，操作系统应该为每个用户维护单独的打开文件表。

11.11 What are the advantages and disadvantages of providing mandatory locks instead of advisory locks whose use is left to users' discretion?

操作系统提供强制锁代替建议锁（可以根据用户的意愿决定是否锁住文件）有什么优缺点？

答：强制锁（mandatory lock）的优点是实现起来更简单，更安全。缺点是会导致当多个进程要访问同一文件时系统运行效率低下（甚至导致死锁）。

11.12 Provide examples of applications that typically access files according to the following methods:

- Sequential
- Random

举出采用顺序访问和随机访问文件的典型程序例子。

答：

- 顺序访问：打印文件内容
- 随机访问：访问数据库中某一指定位置的信息

11.13 Some systems automatically open a file when it is referenced for the first time and close the file when the job terminates. Discuss the advantages and disadvantages of this scheme compared with the more traditional one, where the user has to open and close the file explicitly.

一些系统当进程第一次调用文件时就打开文件，直到进程结束才关闭文件。而传统的做法是用户明确地指定进程打开文件系统才打开文件，指定关闭文件才关闭文件。比较这种自动的做法相对传统做法的优缺点。

答：这样做的优点是可以使开发者的工作更轻松。缺点是，每次进程访问一个文件的时候，操作系统都要顺着文件目录而不是直接通过一个指针找到该文件，然后访问它。并且每个文件被记录下来的使用时间也会边长，即使该文件在进程运行期间只实际使用了很短一段时间。

11.14 If the operating system knew that a certain application was going to access file data in a sequential manner, how could it exploit this information to improve performance?

如果操作系统提前得知一个确定的程序要通过顺序访问的方式访问一个文件的数据，它要如何做才能尽可能地提高访问数据的效率？

答：操作系统可以建立一个缓冲区，将接下来要访问的内容提前读入内存，节约时间。

11.15 Give an example of an application that could benefit from operating-system support for random access to indexed files.

操作系统支持通过索引随机访问文件有何优点，举出一个例子来证明。

答：在数据库应用中，通过键值访问对应的词条时，如果这个键值和操作系统提供的索引对应，那么使用操作系统提供的索引直接访问将大大减少搜索时间。

11.17 Some systems provide file sharing by maintaining a single copy of a file. Other systems maintain several copies, one for each of the users sharing the file. Discuss the relative merits of each approach.

有的系统通过维护一个文件拷贝来实现文件共享，而其他系统则维护多个文件拷贝，供每一个要共享该文件的用户使用。分析这两种做法的优缺点。

答：存放一份拷贝的优点是节约空间，而存放多份拷贝则可以允许多个用户同时读取和写入同一个文件，而不会造成数据冲突。

第十二章 文件系统实现

实践练习部分 Practice Exercise

12.2 What problems could occur if a system allowed a file system to be mounted simultaneously at more than one location?

如果操作系统允许一个文件系统安装在多个路径上，会导致什么后果?

答：那样一个文件可能会拥有多个路径，导致用户混淆和程序错误。

12.3 Why must the bit map for file allocation be kept on mass storage, rather than in main memory?

为什么记录文件分配的表都必须存储在大容量存储器里，而不是放在内存中?

答：如果系统崩溃，易失性存储设备上的数据将会丢失。如果将空闲空间列表（free-space list）丢失，整个主存内的数据都丢失了。

12.4 Consider a system that supports the strategies of contiguous, linked, and indexed allocation. What criteria should be used in deciding which strategy is best utilized for a particular file?

考虑系统支持三种不同形式的文件空间分配方式：连续、链式和索引。确定每种方式最适合什么类型文件的分配，给出你的标准。

答：

- 连续（contiguous）存储——文件将被顺序访问，并且大小相对较小
- 链式（linked）存储——文件较大，且文件通常被顺序访问
- 索引（indexed）存储——文件较大，且通常被随机访问

12.5 One problem with contiguous allocation is that the user must preallocate enough space for each file. If the file grows to be larger than the space allocated for it, special actions must be taken. One solution to this problem is to define a file structure consisting of an initial contiguous area (of a specified size). If this area is filled, the operating system automatically defines an overflow area that is linked to the initial contiguous area. If the overflow area is filled, another overflow area is allocated. Compare this implementation of a file with the standard contiguous and linked implementations.

连续分配文件空间的缺点是用户必须提前为每个文件分配出足够使用的空间。如果文件变得过大，就必须另采取措施。一种解决方案是分配文件时分配一个初始化好的连续空间，如果空间不够用，就另找一块连续的空间，两端空间之间用链表连接起来。对比这种方法与传统的连续分配及链接分配之间的优缺点。

答：这种方法比传统的连续存储开销更大，但是比链式存储开销更小。

12.6 How do caches help improve performance? Why do systems not use more or larger caches if they are so useful?

高速缓存如何提高文件系统的表现？既然缓存如此好用，为什么系统不使用更多的缓存？

答：缓存可以帮助解决由设备之间访问速度差异带来的等待。缓存的访问速度更快，就意味着它的价格也更昂贵。因此如果将缓存做大，系统的造价会更昂贵。

12.7 Why is it advantageous to the user for an operating system to dynamically allocate its internal tables? What are the penalties to the operating system for doing so?

如果操作系统使用动态分配表，对使用启动用户来说有什么好处？这对操作系统自身又有什么代价？

答：因为如果操作系统可以动态的更新它用到的表，那么操作系统将更大概率不会因为表的大小限制而产生错误，于是可以允许用户对文件的操作更加灵活。但因为内核结构和代码都更复杂，这样做也意味着更多潜在的 bug。

习题部分 Exercise

12.9 Consider a file system that uses a modifed contiguous-allocation scheme with support for extents. A file is a collection of extents, with each extent corresponding to a contiguous set of blocks. A key issue in such systems is the degree of variability in the size of the extents. What are the advantages and disadvantages of the following schemes?

a. All extents are of the same size, and the size is predetermined.
b. Extents can be of any size and are allocated dynamically.
c. Extents can be of a few fixed sizes, and these sizes are predetermined.

考虑一个使用改进版连续分配文件的文件系统：文件被分为一系列延伸（extent），每一个延伸绑定一个连续的空间块。这个方法的核心问题在于延伸块的大小，按照以下三种方案设计延伸块，分别有什么优点和缺点？

a. 所有延伸块的大小都相同，预先设定好
b. 动态分配每个延伸块的大小
c. 延伸块只能是几个特定大小中的一种，这几个特定大小也预先设定好。

答：

a. 若所有延伸（extent）的大小都一样，那么分配算法将会变得更简单。但同时也会产生内部碎片。
b. 这样做可以避免产生内部碎片，不过分配算法将会更复杂。
c. 这样做结合了以上两者的优点：即尽可能的减少了内部碎片，又简化了分配算法。

12.10 Contrast the performance of the three techniques for allocating disk blocks (contiguous, linked, and indexed) for both sequential and random file access.

对于顺序读写和随机读写，比较顺序存储 、链式存储和索引存储的效率。

答：

顺序读写：顺序存储 > 链式存储 > 索引存储
随机读写：顺序存储 > 索引存储 > 链式存储

12.11 What are the advantages of the variant of linked allocation that uses a FAT to chain together the blocks of a file?

使用 FAT 作为辅助的链接文件分配方法相比原本的链接分配有什么好处?

答：其优势在于，在随机读写时，我们可以直接访问缓存在内存中的 FAT 来获取块（block）的地址，而不需要通过顺序访问其前面所有的块来定位到它。

12.12 Consider a system where free space is kept in a free-space list.

a. Suppose that the pointer to the free-space list is lost. Can the system reconstruct the free-space list? Explain your answer.
b. Consider a file system similar to the one used by UNIX with indexed allocation. How many disk I/O operations might be required to read the contents of a small local file at /a/b/c? Assume that none of the disk blocks is currently being cached.
c. Suggest a scheme to ensure that the pointer is never lost as a result of memory failure.

考虑一个将空闲空间维护在空闲空间表中的系统：

a. 如果空闲空间表的头指针丢失了，系统能够重新生成一张表吗，解释理由。
b. 考虑类似 UNIX 的使用索引分配的文件系统，如果要读取位于"/a/b/c"位置的文件的内容，需要几次磁盘 I/O 操作？假设没有磁盘块位于缓存中。
c. 设计一个方案，确保空闲文件表指针在内存错误发生的时候不会丢失。

答：

a. 可以。我们需要遍历整个目录来确定那些目录是被进程使用的。那些没有被使用的块便构成了空闲空间表。
b. 一共需要 4 个 I/O 操作来完成。

1）读根目录所在块，
2）读/a 所在块，
3）读/a/b 所在块，
4）读/a/b/中的文件 c 的所在块。

c. 我们可以将指向空闲空间链表的指针储存多次，不仅放在内存中，也放在外存中。

12.14 Discuss how performance optimizations for file systems might result in difficulties in maintaining the consistency of the systems in the event of computer crashes.

分析文件系统的优化措施是如何导致在系统崩溃时难以维持操作系统的一致性的。

答：一个主要的问题在于数据和元数据（metadata）的更新延迟。因为当它们在缓存（cache）和缓冲区（buffer）中被更新后，系统还需要一段时间在硬盘中更新它们。同时，有些算法可能会认为这些更新只是暂时的或者它们马上就要迎来第二次更新，从而推迟它们的回写。这样一来，这种更新的延迟在系统崩溃时便会造成数据的不同步。

第十三章 输入输出系统

实践练习部分 Practice Exercise

13.1 State three advantages of placing functionality in a device controller, rather than in the kernel. State three disadvantages.

说出在设备控制器上实现特定功能而非在内核上实现的三个优点和三个缺点。

答：
优点：1）bug 不会导致系统崩溃，2）使用硬件层面实现的算法往往效率更高，3）内核更简化
缺点：1）bug 很难修复，2）很难更新，3）专门的硬件实现可能无法适配所有软件算法

13.3 Why might a system use interrupt-driven I/O to manage a single serial port and polling I/O to manage a front-end processor, such as a terminal concentrator?

解释为什么操作系统采用中断驱动式 I/O 来管理单一串行端口，而使用轮询 I/O 来管理处理器模块，比如终端集中器（terminal concentrator）？

答：轮询有时比中断 I/O（interrupt-driven I/O）更高效。如果每次 I/O 请求时间较短，I/O 请求较频繁时，轮询比较高效。如果只有一个端口，I/O 请求不频繁，不应该使用轮询。但终端集中器是很多个端口的集合，所以 I/O 请求会更加频繁，使用轮询I/O 更好。

习题部分 Exercise

13.5 How does DMA increase system concurrency? How does it complicate hardware design?

使用直接访问内存（DMA）如何提高系统的并发性？它又是如何使得硬件系统的设计变得复杂的？

答：DMA 通过让 CPU 一边处理它的工作，一边自己通过系统总线和内存总线传输数据来提高系统的并行性。DMA 控制器必须得内置到系统中，而系统也必须要允许 DMA 使用总线。有时为了让 CPU 和 DMA 可以共享内存总线，操作系统还需要支持周期窃取（cycle stealing）。

13.8 When multiple interrupts from different devices appear at about the same time, a priority scheme could be used to determine the order in which the interrupts would be serviced. Discuss what issues need to be considered in assigning priorities to different interrupts.

当多个来自不同设备的中断同时发生时，需要有一个优先处理决定机制来判断先处理谁的中断。讨论设计优先处理机制时需要考虑哪些因素？

答：首先，我们应该给由硬件发出的中断更高的优先级，由软件发出的陷阱（trap）更低的优先级。其次，控制设备的中断应该比执行某种任务的中断具有更高的优先级。比如在显示器上回显应该比从磁盘复制文件拥有更高的优先级。最后，拥有时间限制的中断应该拥有更高的优先级。另外，没有缓冲机制的中断也应该优先执行，因为这种中断处理的数据往往只存在很小的时间窗口。

13.9 What are the advantages and disadvantages of supporting memory-mapped I/O to device control registers?

使用内存映射 I/O 相对使用设备控制寄存器有什么优缺点？

答：优点：这种 I/O 不需要额外的指令集。缺点：设计者必须保证内存访问的合法性。

13.11 In most multiprogrammed systems, user programs access memory through virtual addresses, while the operating system uses raw physical addresses to access memory. What are the implications of this design for the initiation of I/O operations by the user program and their execution by the operating system?

现代操作系统中，物理地址和虚拟地址分开，这种方式对用户发送 I/O 操作和操作系统执行 I/O 各自有何影响？

答：这样的设计通常意味着由用户程序发出的 I/O 请求会指定一片（虚拟）内存中的空间用作缓冲区。在执行 I/O 操作前，内核首先得将数据从用户空间的缓冲区拷贝至内核空间的缓冲区。通过软件翻译地址，内核可以访问到用户空间的缓冲区。当然，如果涉及的虚拟内存页不在物理内存中，这个 I/O 操作还会有一定的延迟（请求调页的延迟）。

13.12 What are the various kinds of performance overhead associated with servicing an interrupt?

处理中断会带来哪些不同类型的开销？

答：处理中断显然会带来上下文切换的开销。除此之外，还有清空和恢复指令流水线（instruction pipeline）的开销。