No.

Date

Week 2 H.W.

1. Describe what virtual Machine is. Then list advantages (benefits) using VM.
   请描述虚拟机是什么，使用虚拟机有什么好处。

答：1. 虚拟机是分层逻辑的延伸，其基本思想是将单个计算机的硬件抽象为几个不同的执行部件，从而造成一种每个独立的执行环境都在自己的计算机上运行的幻觉（进程认为有自己的处理器和内存）

2. 虚拟机提供了到其底层裸硬件（基本硬件）的接口。其上每个进程都有一个与基本计算机一样的（虚拟）副本

3. 用户拥有自己的虚拟机后，他们能运行原来机器所具有的任何操作系统或软件包

4. 虚拟机的实现困难，提供与底层机器完全一样的副本需要做大量工作

非虚拟机：

虚拟机：

虚拟机优点：

1. 不同的系统资源具有完全的保护，每个虚拟机完全独立于其他虚拟机，因此没有安全问题
2. 是用于研究和开发操作的好工具。系统在虚拟机中开发而不在物理机中开发，可以不中断正常系统操作

2. Describe about API and system call. Why we usually use API rather than System Call?
描述API和系统调用，为什么我们用API多过系统调用？

系统调用：

1. 提供了操作系统提供的有效服务界面
2. 通常是用高级程序设计语言编写的（C或C++）
3. 但程序大多通过高级应用程序接口而不是通过直接系统调用来访问OS提供的服务
4. 实现方式：
   - 4-1. 对于每次典型的系统调用都会与一个数相联系，系统调用接口会根据这些数字维护一个列表序列
   - 4-2. 系统调用接口调用所需的操作系统内核中的系统调用，并返回系统调用状态及其它返回值
   - 4-3. 调用者无需知道系统调用是怎样实现的
     只需遵从API并明白操作系统行为
     对于程序员来说，操作系统的绝大多数细节被隐藏起来
     通过运行支持库进行管理（包含编译器等一整套函数）

5. 参数传递
   - 5-1. 通常，系统调用需要提供比所需系统调用识别符更多的信息。
     这些附加信息的种类和数目因操作系统和调用的不同而不同
   - 5-2. 一般有三种方法向操作系统传递参数
     - 最简单的方法：在寄存器中传递参数
       但有些情况下参数比寄存器还多
     - 参数存在内存的块和列表中，并将块的地址通过寄存器传递
     - 参数通过程序压入堆栈，并通过操作系统弹出

   （后两种方法对参数的数目与长度均不做限制）
6. 调用种类
   - 6-1. 进程控制
   - 6-2. 文件管理
   - 6-3. 设备管理
   - 6-4. 信息维护
   - 6-5. 通信

API：

1. 为应用程序接口。程序大多通过高级应用程序接口，而不是通过直接系统调用来访问OS提供的服务
2. 三个最常用的API为Windows的Win32 API，以POSIX为基础的系统(包括UNIX几乎所有版本，Linux，Mac OS X)的POSIX API，与Java虚拟机(JVM)的Java API

为什么通过API而不通过直接系统调用实现：

1. 系统调用效率低，API将系统调用包装起来，使用效率更高
2. 调用API编出程序更具可移植性，不用注重太多细节

直接系统调用：

源文件 ⟶ 目标文件

获取输入文件名
屏幕输入提示
接收输入
获取输出文件名
屏幕输入提示
接收输入
打开输入文件
  如果文件不存在，中止
创建输出文件
  如果文件存在，中止
循环
  读取输入文件
  写入输出文件
直到读取失败
关闭输出文件
特定成信息输出至屏幕
正常结束

标准API的样例：

考虑Win 32 API中的ReadFile()函数——从文档读取信息的函数

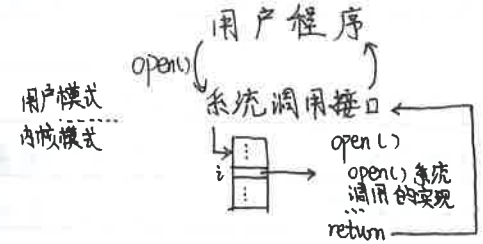
返回值
↓
BOOL ReadFile c (HANDLE file,
　　LPVOLD buffer,
　　DWORD bytestoread,
　　LPWORD bytesread,
　　LPOVERLAPPED ovl);

函数名 ↑　　参数

ReadFile()函数的参数描述如下：

HANDLE file：所要读取的文件
LPVOLD buffer：读进写出的数据缓冲
DWORD bytestoread：需特要读入缓冲区中节点数
LPWORD bytesread：上次读操作读的字节数
LPOVERLAPPED ovl：指示是否使用重叠I/O

API——系统调用——OS之间的关系图



C标准库样例

C程序调用库中printf()函数，而这个函数会进行对write()的系统调用

```
#include <stdio.h>
int main() {
...
printf("Greetings");
...
}
```

用户模式
内核模式
C 标 准 库
write()
write()系统调用

3. What is the microkernel? What are the advantages and disadvantages of using the microkernel approach?
什么是微内核？使用微内核有什么好处和坏处？

答：1. 微内核是将所有非基本部分从内核中移走，并将它们实现为系统程序或用户程序，这样得到了更小的内核

2. 一个例子：

假如一开始内核中有文件管理模块、内存管理模块、多媒体模块等等，我们将多媒体等这些移出内核，剩下的就形成了微内核。

3. 好处和劣势

好处：易于扩展（内核空间加大）
更易将操作系统移植到新结构中（更少代码）
更可靠（更少代码在内核模式中运行 → 更少故障点）
更安全

劣势：用户模式到内核模式的通信性能开销
（例如：多媒体就存在大量访问I/O行为，我们需要将其频繁切换于两模式间，存在大量开销，影响系统性能）

100/R. 图画得不错

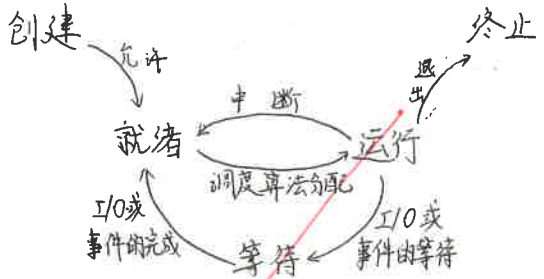
No.

Date

Week 3. H.W.

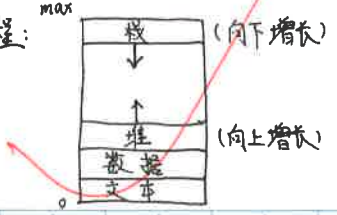
1. According to Fig 3.2, describe what process is and five possible states clearly.
请根据图3.2清晰地描述什么是进程并阐释进程五状态.

答: 图3.2:



答: 进程

1. 进程是正在执行的程序, 或者说进程是按照我们要求所编译的指令序列
2. 进程不只是程序代码(代码段), 还包括当前活动(程序计数器, 处理器寄存器), 通常还有堆栈段(临时数据: 函数参数、返回地址、局部变量)、数据段(全局变量), 还可能包括堆(运行时动态分配的内存)
3. 进程是活动实体, 有一个程序计数器来表示下一个要执行的命令和相关资源集合, 当一个程序(可执行文件)被载入内存时, 一个程序才能成为进程
4. 内存中的进程:



No.

Date

进程五状态:

在进程处理的过程中发生着状态的切换, 共有五个可能的状态.

1. 创建: 进程正在被创建
2. 运行: 正在执行指令(进程进入到CPU里面了)
3. 等待: 进程正在等待事件发生, 并不占有CPU
4. 就绪: 进程等待分配处理器(万事俱备, 只欠东风, 就差进到CPU里了)
5. 终止: 进程终止执行

进程五状态详解:

1: 创建: 创建进程(并不具备执行能力)
↘(载入内存, 分配内存空间等资源)
就绪: 进程已具备执行能力, 万事俱备, 只欠东风

2: 就绪
↓调度算法分配, 从就绪队列中选取进程进入CPU中
运行

3: 终止: 执行完毕, 释放I/O、内存等资源
↑(在给定时间内执行完毕)

运行
等待I/O或其它事件发生 → 等待 (不占有CPU)
中断: 运行是有时间限制的, 当时间一到进程就会被硬生生地抓到就绪状态中(即使进程仍有继续执行的能力) → 就绪

4: 等待
↓ I/O或其它事件的完成
就绪

(注: 特别注意 I/O或其它事件完成后是回到就绪状态, 并不能回到运行状态)

No

Date

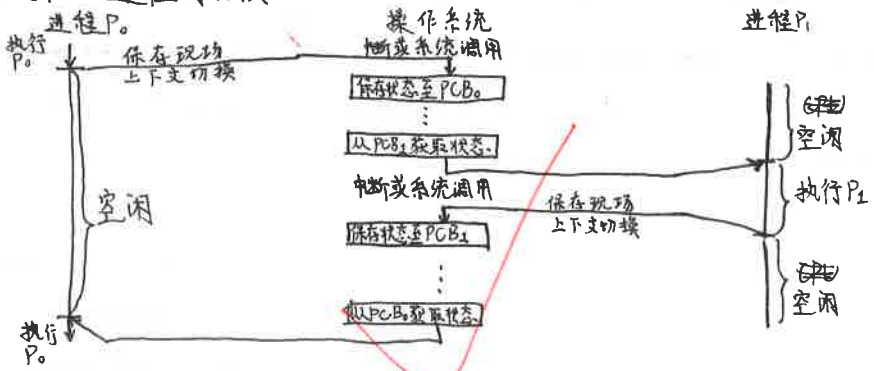
补充：

进程控制块（PCB）：相当于进程的身份证，包含5-个特定进程相关的信息

1. 进程状态：说明到了五状态中的哪个状态
2. 程序计数器：告诉我们执行到哪个语句了
3. CPU寄存器：进行上下文切换时储存CPU状态
4. CPU调度信息：用了多少CPU
5. 内存管理信息：用了多少内存
6. 审计信息
7. I/O管理信息

……

CPU在进程间切换：



从此也可以看出频繁的上下文切换会使系统性能下降。

No

Date

进程调度队列

1. 作业队列：系统中所有进程的集合（注意是"所有进程"）
2. 就绪队列：驻留在内存中就绪执行或等待执行的所有进程的集合
3. 设备队列：等待I/O设备的进程的集合

注：1. 就绪队列只有一条，为处在就绪状态的进程的队列

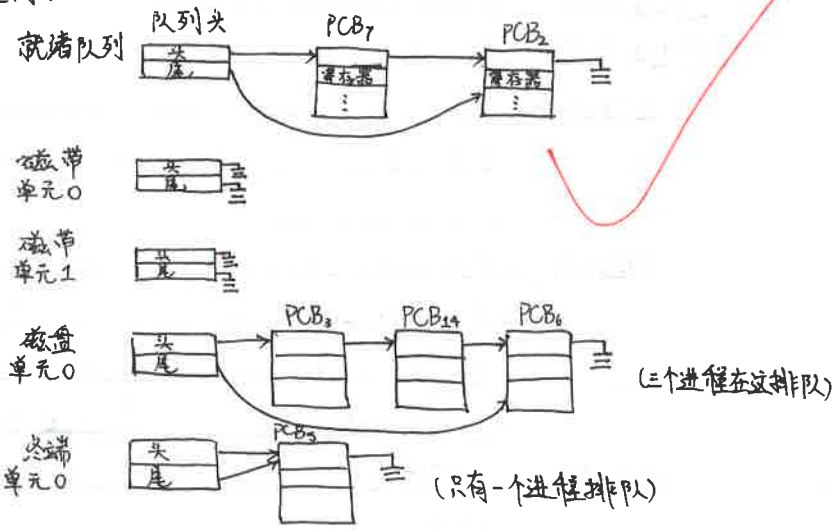
2. 设备队列对于不同类型的设备分了几条队，为处在等待状态的进程的队列

3. 运行状态中只有一个进程，没有"队列"一说

4. 进程在各个队列中迁移：

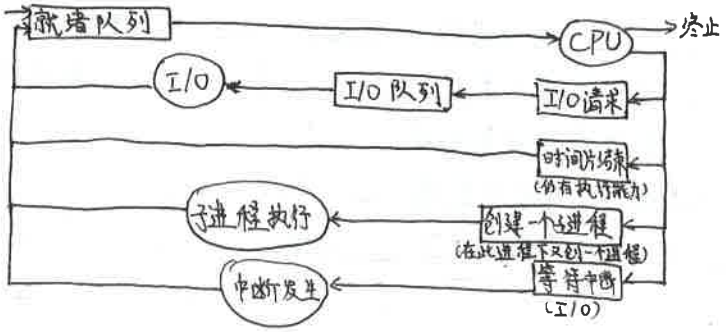
就绪队列和各种设备队列

图示：



注：每次会有一个就绪队列中的进程进入CPU中运行，当此进程要求I/O时会到对应设备队列中排队，完成后会再回到就绪队列中排队，或者直接回到就绪队列中排队（时间片结束）

表示进程调度的队列图



2. Please describe the differences among short-term, medium-term, and long-term scheduling.

清阐述短程调度，中程调度与长程调度的差别

1. 作用上有差别（根本差别）

短程调度：由于只能有一个进程能分配到CPU处于运行状态，短程调度的作用就在于选取哪个在就绪状态中的进程进入运行状态

中程调度：当某进程在CPU执行的过程中产生了巨大变故（比如：CPU利用率极低，过分执行或并不适合在本机上执行），会将该进程（已执行了一部分）移出CPU，将信息从内存移至磁盘中（等待下一次进入进程五状态的"就绪"状态中）

长程调度：由于内存等资源有限，进入就绪状态（此时已得到分配到的内存等资源，具备了执行能力）的进程不能太多。长程调度在于从新创建的进程中选取几个、选取哪些进程给它分配资源，进入就绪状态。也就是说，长程调度控制着多道程序的程度（因为其控制了内存中的进程数）

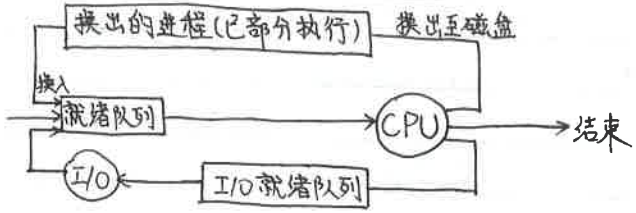
2. 调用频率有差别

短程调度：频繁进行（毫秒级）⇒ 要求必须要快

长程调度：不那么频繁进行（秒级或分钟级）⇒ 可能速度会慢

中程调度介于二者之间

注：1. 中程调度图示



注：2. 长程调度必须仔细选择，这是因为有两类进程

I/O相关的进程：用更多时间在I/O上（如：磁盘碎片整理程序）

CPU相关的进程：用更多时间在运算上（如：测试计算机性能的程序）

长程调度应平衡于二者，若

I/O相关的进程过多：等待队列很满，就绪队列很空

CPU相关的进程过多：等待队列很空，就绪队列很满

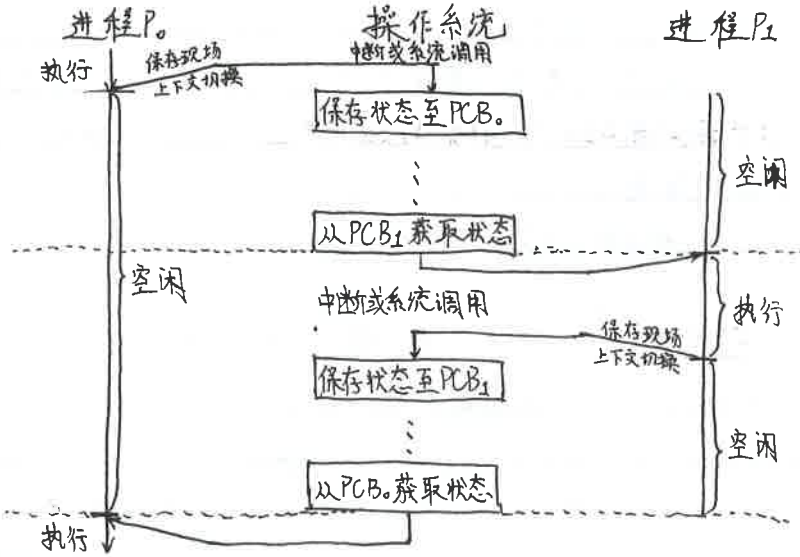
为达最好性能，需平衡好这二者

3. Please describe the actions taken by a kernel to context-switch between processes.

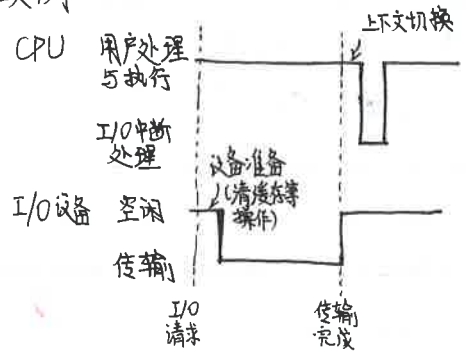
清阐述在上下文切换的过程中内核的行为

答：1. 当CPU需要从当前进程切换到另一个进程时，系统必须保存上一进程状态，载入下一进程状态。（内核会将旧进程的状态保存在进程控制块（PCB）中，然后载入经调度要执行的新进程的信息（从PCB中获取））

2. 过程图示



3. 实例：



注：1. 大量上下文切换会极大影响系统性能

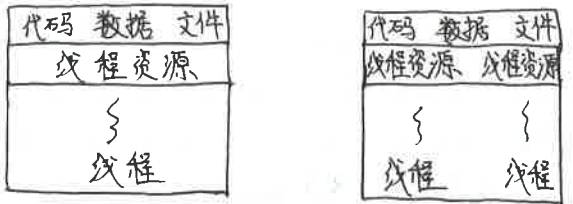
2. 系统越复杂，上下文切换工作越多

4. Please describe the differences between process and thread.

请阐释进程与线程间的差别

答：1. 线程是进程的一部分（有时线程称为轻权进程或轻量级进程，为CPU调度和分派的基本单元）；而传统意义上的进程被称为重量级进程，从现代的角度来看，是只拥有一个线程的进程。如果进程有多个控制线程，那么它就能同时执行多个任务

关系图：



2. 调度上：

1. 传统操作系统中，进程是CPU调度基本单位
2. 现代操作系统中，线程是CPU调度基本单位，进程为资源拥有的基本单位

3. 并发性：

进程间可并发执行，一个进程的多个线程间亦可并发执行

4. 拥有资源：

进程为拥有系统资源的独立单位，它可拥有自己的资源

线程不拥有系统资源，但可访问其隶属进程的资源

5. 系统开销：

1. 对于创建/删除进程操作系统所付出的开销远大于创建/删除线程时操作系统所作的开销
2. 对于线程的同步和通信较为简单

A+ good (V^_^)V

No.

Date

Week 4. H.W.

1. What is a thread pool? What problem does it solve and what benefits does it make.

线程池是什么？它能解决什么问题，有什么优点？

解：1. 由来背景：在线程池概念出现前，每当服务器收到请求，就会创建一个(问题)独立线程来处理请求。这会带来一些问题，其一是每次创建线程都会存在时间上的开销；二是没办法限制系统中并发执行的线程的数量，而无限制的线程可能会耗尽系统资源。

2. 由此我们提出了线程池的思想。我们在进程开始~~的~~时先创建好一定数量的线程，让它们先处于"睡眠"状态(等待工作)，之后每来一个请求就唤醒一个线程给它服务，服务完了以后又回到池中处于"睡眠"状态，而这些"睡眠"状态的线程的集合即为线程池。

3. 优点：针对问题一：免去了每来一个请求就要创建一个线程的时间，应答速度更快，性能更优

针对问题二：由于线程池中处于"睡眠"状态的线程有限，因此当全部线程都处于服务状态时，再有~~的~~请求过来，就必须要等待某一线程服务完了以后才能应答该请求。此举控制了系统并发的程度，避免了对系统资源的过度利用(尤其对不支持大量并发的系统非常重要)

2. Consider a multiprocessor system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be more than the number of processors in the system. Discuss the performance implications of the following ~~scen~~ scenarios:

a. The number of kernel threads allocated to the program is less than the number of processors

b. The number of kernel threads allocated to the program is equal to the number of processors

c. The number of kernel threads allocated to the program is greater than the number of processors but less than the ~~num~~ number of user-level threads.

现有一个多处理器系统与通过多对多线程模型所编写出的多线程程序，同时这个多线程程序中的用户级线程数比处理器数目多，讨论下列情况对性能的影响：

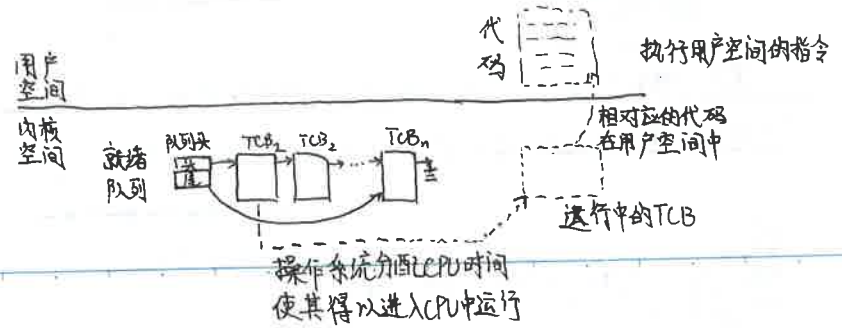
a. 分配给这个程序的内核线程数比处理器数目少

b. 分配给这个程序的内核线程数和处理器数目一样多

c. 分配给这个程序的内核线程数比处理器数目多，但比用户~~线~~级线程数少

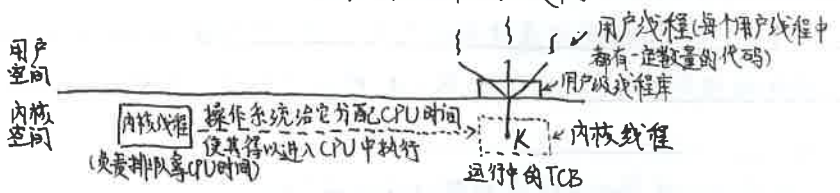
解：在解答本题之前先解释一下多线程和单线程的区别，以更好地理解用户级线程和内核线程的概念

1. 在原本单线程中内核空间由就绪队列中有一些线程控制块以链表的形式在排队，待CPU空闲时，操作系统会给它分配CPU时间，让它进入CPU运行。之后系统就会根据TCB(线程控制块)，执行其对应用户空间上的指令。如下图



2、由于拿CPU时间属于内核空间的工作，执行的代码在用户空间，因而我们将这两者分开，也就有了内核线程及用户级线程的概念，内核线程负责排队拿CPU时间。拿到CPU时间，处于运行状态后会映射到对应的用户级线程上，告诉每个对应的用户级线程优先级（运行时间长短）（注：这是由用户级线程库所管理的）。而每个用户级线程都有一定数量的代码，具体过程如下图：

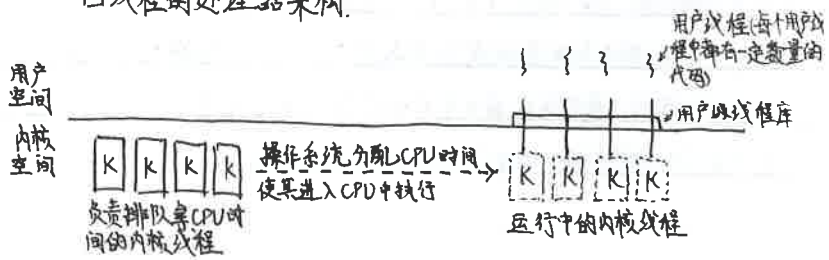
①多对一模型（多个用户级线程对应一个内核线程）



特点：一个排队的内核线程映射多个用户线程

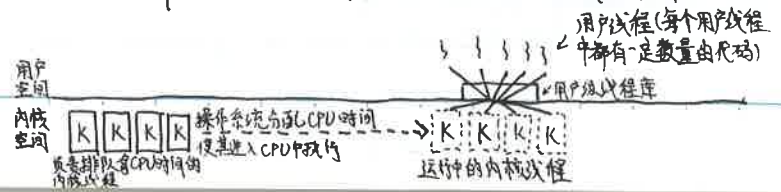
②一对一模型（一个用户级线程对应一个内核线程）

注：一对一模型一般用于多处理器架构中，因此我们假定一个双核四线程的处理器架构



特点：多个排队的内核线程，每个内核线程映射到单个用户线程上

③多对多模型即为上述二者的结合（仍假定双核四线程架构）



之后我们再看看多对一模型、一对一模型、多对多模型的特点及优劣

①多对一模型：

定义：多个用户级线程对应一个内核线程

特点：一个排队的内核线程映射多个用户线程

优点：1. 只有一个排队的内核线程负责拿CPU时间，如果想要让某个（或某些）用户线程优先级高，让某个（或某些）用户线程优先级低，我们通过修改用户级线程库就可以做到，而不需修改内核（内核修改难度极大）

2. 线程间切换只用在用户空间中进行，效率高

3. 几乎不限制线程数量（因为只有一个内核线程在排队，操作系统就认为只有一个线程）

缺点：1. 只要有一个用户线程需要进行系统调用（比如进行I/O操作），内核线程立马就会退出运行状态至等待状态（对应五状态模型）。那么由于用户线程是在内核线程拿到CPU时间，处于运行状态时才能执行的，因而其它的用户线程均无法往下执行（一句话：如果其中一个用户线程阻塞，所有线程均无法执行，此时内核中线程也随之阻塞）

2. 在多处理器系统上，处理器增多对多对一线程编写的程序的性能提升没有帮助（因为只有一个内核线程在排队，窗口再多也没有用）

② 一对一模型：

定义：每个用户级线程对应一个内核线程

特点：多个排队的内核线程，每个内核线程映射到单个用户线程上

优点：1. 用户线程和内核线程具有一致性，线程之间的并发是真正的并发，一个线程的阻塞不影响其他线程的正常执行

2. 在多处理器的系统上有更好的表现（随着处理器的增多，性能越优）

缺点：1. 系统资源有限，因而很多系统限制了内核线程的数量，由于每个用户线程必须对应到一个内核线程上，因此用户线程的数量同样会受到限制

2. 由于操作系统负责每个内核线程的时间分配，而每个内核线程又只与某个用户线程相对应，因此每个用户线程的优先级相同，且难以更改其优先级

3. 线程间切换涉及内核，需要进行模式切换，需要保存上下文，使得用户线程的执行效率下降

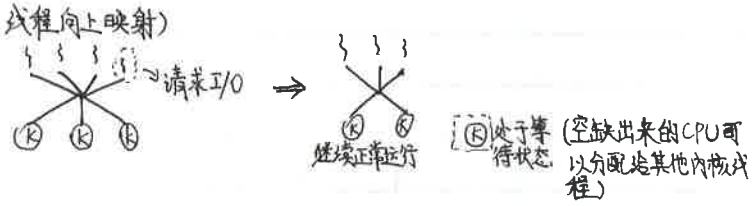
③ 多对多模型：

定义：允许多个用户级线程映射到多个内核线程

特点：多个排队的内核线程，允许每个内核线程映射到多个用户线程上

优点：多对多模型融合了上述两模型的优点

1. 一个线程的阻塞不影响其他线程的正常执行（因为可以由其他的内核线程向上映射）



2. 在多处理器系统上同样有更好表现

3. 几乎不限制用户线程数量（极端情况就像多对一模型那样，一个内核线程对应多个用户线程）

4. 也可以在用户空间中进行线程切换

5. 可以通过用户级线程库进行一定程度上的优先级处理

缺点：1. 实现难度很大

2. 不能通过用户级线程库完全进行优先级处理

3. 随着处理器增多，性能提升效果不及一对一模型那么好

通过上述分析，我们就可以很容易地回答出这道题目了

a. 当内核线程数比处理器数目少时，肯定有处理器处于空闲状态或者服务于其他程序的线程

b. 当内核线程数和处理器一样多时，可能所有处理器都在服务于这个程序。但一旦某个线程请求系统调用，它所在的内核线程中的某一个就要处于等待状态，因此就会有处理器处于空闲状态或服务于其他程序

c. 当内核线程数比处理器多时，肯定有内核线程不处于运行状态，此时一旦某个线程请求系统调用，它所在的内核线程中的某一个就要处于等待状态，因此处理器就可以服务于本程序的其处于等待队列中的线程

3. What are the benefits of multithreaded programming?

多线程程序设计的好处是什么？

解：1. 响应度高：如果对一个交互程序采用多线程，那么即使其部分阻塞或执行较冗长的操作，该程序仍能继续进行

2. 资源共享：可以共享它们所属进程的内存和资源，即能允许一个程序在同一地址空间有多个线程。这也使线程间通信比原来要简单

3. 经济：创建进程所需的内存和资源较多，由于线程可共享它们所属进程的资源，因而创建线程更为经济

4. MP（多处理器）架构的用率高：增强了并发特性（每个进程可并行运行在多处理器上）

No.

Date

Week 5 H.W.

1. Please describe the differences between preemptive scheduling and nonpreemptive scheduling.
请叙述抢占式调度与非抢占式调度有什么区别.

答：在此之前先来看看CPU的4个可能的调度决策

1. 当一个进程从运行状态切换至等待状态.(I/O请求,等待事件发生)
2. 当一个进程从运行状态切换至就绪状态.(出现由时间片用完所产生的中断)
3. 当一个进程从等待状态切换至就绪状态.(完成I/O,事件已发生)
4. 当一个进程终止时(进程执行完了)

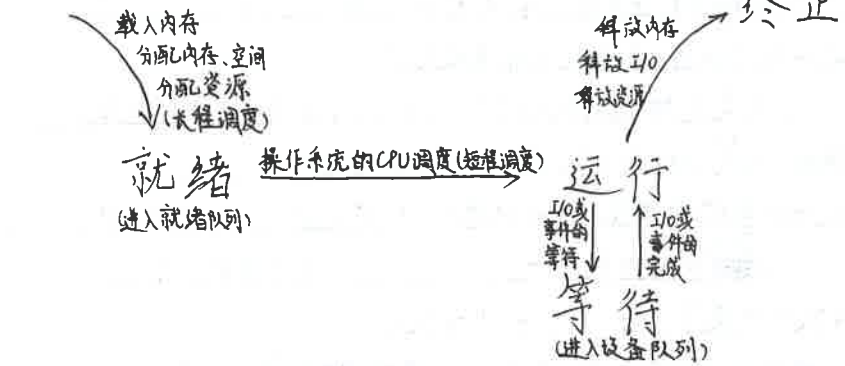
1. 当一个进程从运行状态切换至等待状态是非抢占下也会发生的,因为它在等待事件发生或者I/O输入的过程中已经失去了运行的能力。因此操作系统夺走了它的运行权利,让它回到了等待状态
2. 当一个进程从运行状态切换至就绪状态是只有在抢占下才会发生的,因为当时进程还有继续运行的能力,只是由于时间片用完了.也就是说操作系统将它的运行权利夺走了是非必需的,完全可以等它运行完,因此这是抢占的.
3. 当一个进程从等待状态切换至就绪状态也是只有在抢占下才会发生的,因为当事件发生完或者I/O输入完后进程已经具备了运行的能力,完全可以换入CPU中继续运行(进入运行状态),而我们强令其回到就绪状态是非必需的,因此这也是抢占的.
4. 当进程运行终止时也是在非抢占下也会发生的,因为它已不需要运行了,因此操作系统将它的运行权利夺走了,终止了进程.

从上述分析就可以很容易地看出两者的区别了:

⇒非抢占式调度只会出现2个CPU调度决策

① 进程从运行状态切换至等待状态
② 当一个进程终止时

也就是说对于非抢占式调度来说对应的进程五状态图为

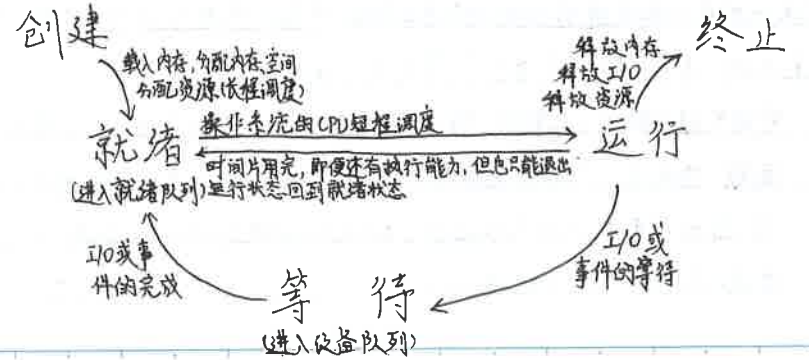


对于非抢占式调度来说一旦进入运行状态将不再回到就绪状态.

⇒抢占式调度会出现全部4个CPU调度决策

① 进程从运行状态切换至等待状态
② 进程从运行状态切换至就绪状态
③ 进程从等待状态切换至就绪状态
④ 当一个进程终止时

也就是说对于抢占式调度来说对应的进程五状态图为



对于抢占式调度来说进入运行状态后仍可能回到就绪状态

2、In the multilevel queue-scheduling algorithm, what is the advantages of using different size of time quantum in different layers?

在多级队列调度算法中，在不同层级使用不同长度时间片的好处是什么？

答：在解答本题之前先来看一下多级队列调度是什么

1. 多级队列调度是将就绪队列分成多个独立队列，而每个队列选择适合自己的调度算法（同时在队列与队列之间也必须有调度）

2. 一个例子：我们将原先的一条就绪队列分成"前台"和"后台"两条队列，并且假设有两个进程，一是游戏进程，二是杀毒软件进程

在多级队列调度算法前，只有一条就绪队列

* 众所周知，游戏的响应速度是比杀毒软件高是得多的，交互性也更强（这一点我们很好理解，例如在玩《红色警戒》、《帝国时代》什么的时候对手已经兵临城下了，要是我们系统的响应速度很慢，按个键都要等几秒才做出反应，估计离 game over 就不远了；但在杀毒软件中，反应慢点就慢点，只要起到应有功能就行）。而我们对游戏做的每一个操作占用CPU时间都不大，即游戏中进程基本都是I/O密集型的，将会用比计算更多的时间在I/O上，具有很多短的CPU周期。但对于杀毒软件进程来说占用CPU时间会长得多，即杀毒软件中进程很多都是CPU密集型的，计算的时间很长。

* 假设平均每个游戏进程的时间片为1单位，杀毒软件进程的时间片为50单位

* 如果只有一条就绪队列，各自分得时间片相同

  ◦ 时间片短（假设为1单位），那么游戏进程的响应速度很快，周转时间很短，也没有CPU资源的浪费，但跑一个杀毒软件进程却要进出CPU 50次，这会带来极大的分派延迟。由于每次都要进行上下文切换，也使性能急剧下降，时间开销很大

  ◦ 时间片长（假设为25单位），那么杀毒软件进程进出CPU两次就好了，上下文切换次数不多，系统的性能还是能得到保证的，时间开销不明显，但是对于一个只用1时间片的游戏进程来说，每次可能需要等上它25倍的时间才轮到它完成一个小小的任务，响应速度急剧下降，周转时间急剧上升

* 那么，如果我们分成两条队列，比如说"前台"和"后台"，一个负责交互，一个负责批处理；给"前台"更高优先级或更大的时间片，给"后台"较低优先级或较少时间片；对于"前台"中每个进程的时间片很小，"后台"中每个进程的时间片很大，就可以两全齐美了。

  ◦ 对于这个例子，我们可以将游戏进程放到"前台"，将杀毒软件进程放到"后台"，令"前台"中每个进程的时间片为1，"后台"中每个进程的时间片为25，这样游戏进程就可以拿到更快的响应速度和更少的周转时间，杀毒软件进程也避免了频繁的上下文切换，系统性能也得以提升。

由此，我们就可以很简单地回答出这个问题了

好处是：调度程序可以将运行时间短或交互性强或I/O密集或优先级高的进程安排在时间片小的队列，这样可以提高系统的响应速度和减少周转时间；而对于需要连续占用（占用长时间）处理器的低优先级进程可安排在时间片长的队列中，可有效减少上下文切换次数，避免过分影响系统性能

注：从这里又引申出了多级反馈队列调度的方法。多级反馈队列调度是允许进程在队列之间进行移动的。在多级反馈队列中，一个进程处于哪一个时间片的队列，可以在运行时根据它先前运行状况加以调整。假定一个进程开始处于高优先级、时间片小的队列中，如该进程运行完时间片后还未结束或未进入阻塞（等待）状态，那么可将其转入下一个时间片较长的低优先级队列，这样短进程可较快地获得CPU，长进程一旦进入运行状态也可运行较长时间，以实现调度。

No.

Date

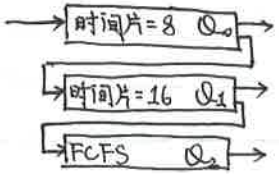
*例子：

三个队列：

Q_0：时间片为8毫秒的轮转调度

Q_1：时间片为16毫秒的轮转调度

Q_2：FCFS（先到先服务调度）



①：一个时间片小于8毫秒的进程进入到CPU后未运行完就终止

②：一个时间片介于8毫秒到24毫秒之间的进程进入到CPU 8毫秒后被中断（时间片用完），被调度出来进入Q_1. Q_1的优先级比Q_0低，等到Q_0没有进程的时候，先进入Q_1的进程可进入CPU中最多连续运行16毫秒

③：如果还没运行完将被调度出来进入Q_2. Q_2的优先级比Q_1低，等到Q_1，Q_0没有进程的时候，先进入Q_2的进程可连续进入CPU中运行更长的时间

注：1. 为避免饿死在一个队列中，随着排队时间的加长，优先级会不断增大

2. 实际情况下可以多划分几级

3. 要是某个进程占用了很长时间CPU，中程调度会把它从CPU中强制移出

4. 进行这种调度方法需要解决下列问题

- 队列数量
- 每个队列的调度算法
- 用以确定何时升级到最高优先级队列的方法
- 用以确定何时降级到最低优先级队列的方法
- 用以确定进程在需要服务时应进入哪个队列的方法

No.

Date

3. Consider the following set of processes, with the length of the CPU burst given in milliseconds

假设有以下的进程，其CPU时间片长度如下图所示

Process（进程）	Burst Time（时间片）	优先级数
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

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 smaller priority number implies a higher priority), and RR (quantum = 1)

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 the scheduling algorithms in part a?

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

假设这些进程都在$t=0$时刻以P_1、P_2、P_3、P_4、P_5的顺序到来

a. 画4个Gantt图来分析在下列四种算法下这些进程的运行情况：FCFS（先到先服务），SJF（最短作业优先），非抢占式优先级调度（优先级数越小表优先级越高），RR（轮转法，时间片为1）

b. 每个进程在每种调度中的周转时间是多少

c. 每个进程在每种调度中的等待时间是多少

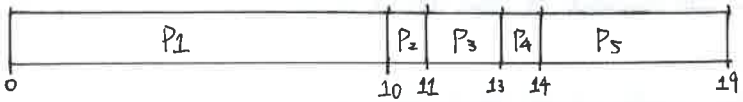
d. 哪种调度算法获得了最小平均等待时间

答：在解答本题之前先来大致看一下上面提到的四种调度算法

① FCFS（先到先服务）调度：

1. 先到先服务：先到的进程先运行
2. 非抢占
3. 受到来顺序的影响大
4. 会产生护航效果：由于长进程会占用大量时间，因此会有大量短进程排在后面

对于本题来说：



周转时间（从进程提交到完成的时间）

P1：10

P2：11

P3：13

P4：14

P5：19

等待时间（在就绪队列中等待所花费时间之和）

P1：0

P2：10

P3：11

P4：13

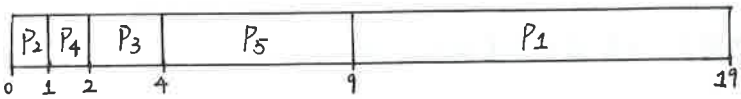
P5：14

平均等待时间：(0 + 10 + 11 + 13 + 14) / 5 = 9.6

② SJF（最短作业优先）调度：

1. 将每个进程与其下一个CPU区间段相关联，并用这些CPU区间段长度来调度最短时间的进程（时间片越短越优先）
2. 可以抢占也可以非抢占
   - 非抢占：一旦进程进入CPU，在进程没有完成CPU时间片之前不能被抢占
   - 抢占：如果一个新进程的CPU时间片长度比当前执行的进程的剩余执行时间更短，则抢占。这称为最短剩余时间优先调度（SRTF）
   - 由于这题进程同时到来，因而两者效果相同
3. 可能产生饥饿（大量短进程涌入导致长进程长时间等待）
4. 难以事先确定作业的时间片长度

对于本题来说：



周转时间（从进程提交到完成的时间）

P1：19　P2：1　P3：4　P4：2　P5：9

等待时间（在就绪队列中等待所花费时间之和）

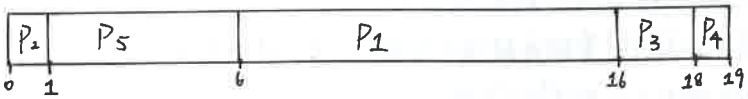
P1：9　P2：0　P3：2　P4：1　P5：4

平均等待时间：(9 + 0 + 2 + 1 + 4) / 5 = 3.2

③非抢占式优先级调度(优先级数越小优先级越高)

1. 每个进程都有一个优先级与其关联,具有最高优先级的进程会分配到CPU
2. 可以抢占也可以非抢占
   - 非抢占:一旦进程进入CPU,在进程没有完成CPU时间片之前不能被抢占
   - 抢占:如果一个新进程的优先级比当前执行进程的优先级高,则抢占
3. 亦可能产生饥饿,但可通过"老化"的办法解决(随等待时间变长,优先级升高)

对于本题来说:



周转时间(从题交进程到完成的时间)

$P_1:16$ $P_2:1$ $P_3:18$ $P_4:19$ $P_5:6$

等待时间(在就绪队列中等待所花费的时间之和)

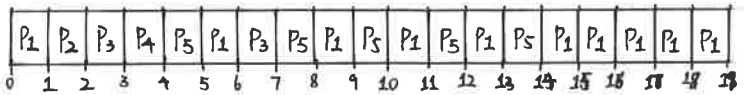
$P_1:6$ $P_2:0$ $P_3:16$ $P_4:18$ $P_5:1$

平均等待时间:$(6+0+16+18+1)/5=8.2$

④轮转法调度

1. 每个进程得到一小单元的CPU时间(时间片),当时间耗完,该进程被抢占并放到就绪队列尾部
2. 一定是抢占的
3. 如果有n个进程在就绪队列中,时间片为q.每个进程必须等待的CPU时间不会超过$(n-1)q$个时间片长度.
4. 当q很大时相当于FIFO(先到先服务),q很小时会带来极大的上下文切换代价,开销极大
5. 通常情况下具有比SJF(最短作业优先)更大的周转时间,但具更好的响应时间(响应时间最多$(n-1)q$)

对于本题来说:



周转时间(从题交进程到完成的时间)

$P_1:19$ $P_2:2$ $P_3:7$ $P_4:4$ $P_5:14$

等待时间(在就绪队列中等待所花费时间之和)

$P_1:9$ $P_2:1$ $P_3:5$ $P_4:3$ $P_5:9$

注意:别把运行时间算进去

平均等待时间:$(9+1+5+3+9)/5=5.4$

综上所述:SJF(最短作业优先算法)是最优的.

A+

No.

Date

Week 6 H.W.

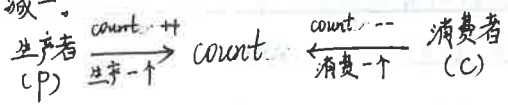
1. What is a critical section? What the three requirements does a solution to the critical-section problem satisfy?

什么是临界区？解决临界区问题的算法需满足什么条件？

答：① 临界区是共享数据被访问的代码段

* 这是因为共享数据的并发访问可能产生数据的不一致

比如说：我们想通过有限缓冲解决生产者——消费者问题时，我们增加一个整型变量 count，并初始化为0，每向缓冲区增加一项（即生产一个时），count 加一；每从缓冲区移除一项（即消费一个时）count 减一。



假如 count ++ 是通过如下方式实现的：

```
register 1 = count
register 1 = register 1 + 1
count = register 1
```

count -- 是通过如下方式实现的：

```
register 2 = count
register 2 = register 2 - 1
count = register 2
```

乍看之下没问题，但来看看下面这种情况（假设一开始 count 为5）

S0：生产者执行 register 1 = count (register 1 = 5)

S1：生产者执行 register 1 = register 1 + 1 (register 1 = 6)

不料这个时候时间片刚好用完，发生了切换，切换到消费者进程

S2：消费者执行 register 2 = count (register 2 = 5)

S3：消费者执行 register 2 = register 2 - 1 (register 2 = 4)

这时候时间片又用完了，又切换回了生产者进程

S4：生产者执行：count = register 1 (count = 6)

然后生产者进程结束了，退出CPU终止执行，又轮到消费者进程

S5：消费者执行：count = register 2 (count = 4)

最终结果，count = 4

但我们很快发现这里面的问题，按正确的道理来说应该是这样子的

首先生产者对 count 进行操作

S0：生产者执行 register 1 = count (register 1 = 5)

S1：生产者执行 register 1 = register 1 + 1 (register 1 = 6)

S2：生产者执行 count = register 1 (count = 6)

然后消费者再对 count 进行操作

S3：消费者执行 register 2 = count (register 2 = 6)

S4：消费者执行 register 2 = register 2 - 1 (register 2 = 5)

S5：消费者执行 count = register 2 (count = 5)

最终结果，count = 5

从此我们可以看出共享数据的并发访问可能会出现数据的不一致性

* 由此，我们需要确保共享同一逻辑地址空间的协作进程有序进行

比如说：在生产者进程对 count 的操作未完成前不会被调度走；同样地在消费者进程对 count 的操作未完成前也不会被调度走

② 由此，我们可对临界区下一个综合定义。

临界区是共享数据被访问的代码段，在临界区中（即对共享数据的访问过程中）只允许有一个进程进来（即只允许有一个进程访问此数据，对其进行操作），且操作完成以前这个进程不会被调度走，别的进程也不允许对此数据进行访问

③ 一种形象的理解：

临界区就好比是厕所，进程就像是一个又一个的人

* 一个人进了厕所关上门，别人都别想进来（不然会产生严重后果）
* 同时只能有一个人进去

解决临界区问题需满足以下三个条件：

1. 互斥：如果进程 P_i 在其临界区内执行，那么其他进程均不能在其临界区中执行。即临界区外进程不能干扰临界区内进程
   * 在具有关于相同资源或共享对象的临界区的所有进程中，一次只允许一个进程进入临界区（择一而入）
   * 好比当有人在上厕所时，别人都不能进来（无空等待）
2. 前进：如果没有进程在其临界区内执行且有进程需要进入临界区，那么只有那些不在剩余区内执行的进程可参加选择，以确定谁能下一个进入临界区，且这种选择不能被无限延迟
   * 简言之一句话：没有进程在临界区时，任何需要进入临界区的进程必须能够立即进入（有空让进）
   * 好比当没有人在厕所里面的时候，一个需要上厕所的人可以马上进去
   * 注意：当临界区内没有进程时，外面等的可以进一个（只允许最多一个进去），但未必是在外面排队的那个，有可能还是刚退出临界区的进程
   * 好比当没有人在厕所里面的时候，外面有几个人都想上，但只会有那个最快的进去，而不会几个人同时进去（择一而入）
   * 存在这样一种情况，一个人上完厕所刚把厕所门打开，然后突然又想上厕所，随即把厕所门关上了。
3. 有限等待：从一个进程做出进入临界区的请求，直到该请求允许为止，其他进程允许进入其临界区的次数有上限。假定每个进程的执行速度不为0。然而不能对n个进程的相对速度做任何假设。
   * 其实说了这么多也就一句话：一个进程阻留在临界区中的时间必须是有限的（算法可行）
   * 好比一个人上厕所不能一直呆在里面不出来

因此总结出四句话：

有空让进，无空等待，择一而入，算法可行

注：典型进程的通用结构

```
while (true) {
    进入区
    临界区
    退出区
    剩余区
}
```

进入区：请求允许进入临界区的代码段

退出区：临界区之后的代码段

2、What is the meaning of the term busy waiting? Can busy waiting be avoided altogether? Explain your answer.

术语"忙等待"的意思是什么？忙等待能否完全避免？为什么？

答：忙等待实际上是一种循环等待来解决临界区问题的方法。在忙等待的过程中CPU并没有做实际的工作。通过下面这个例子我们就可以比较清楚了。但在讲那个例子之前先来看一些关于原子操作的背景知识。

* 对于临界区问题有两种解决思想，一是通过软件的方法进行解决（例如：Peterson算法）；二是通过硬件的手段进行解决。前者对于两个进程以内的临界区问题解决是很好的，但对于多于2个进程的情况前者往往无能为力，因此很多系统都提供了对于临界区的硬件支持

  * 在单处理器中可以通过禁止中断的方法解决（因为临界区内不允许有中断出现）。但这样不仅临界区内没中断，进程在运行过程中也不会被抢占。因而这使得系统运行低效（因为一个长进程的运行会引发众多短进程的排队，产生护航效果），也不具有很好的扩展性（因为把中断屏蔽了）

  * 对于现代计算机系统来说，提供了专门用于处理临界区问题的硬件指令。而这些硬件指令是原子的（即不可被中断的，因为在当时看来原子是万物里最小的），对于原子操作来说只有成功执行和不成功执行两种情况，不可能执行一半中止。就像ATM机取钱和扣除账户一样，取100一定账面上被扣100.

* 由于有原子操作的出现，我们可以通过定义原子函数来解决临界区问题

比如 testset 函数（注意在原子函数中参数传递的效果相当于指针传递）

```
boolean testset (int i) {
    if (i == 0) {
        i = 1;
        return true;
    }
    else {
        return false;
    }
}
```

然后就可以利用这个函数来解决临界区问题了

例子：

```
int const n = /*进程数目*/
int bolt;

void P(int i) {
    while(true) {
        while (!testset(bolt))
            ;
        /*临界区*/
        bolt = 0;
        /*剩余区*/
    }
}

void main() {
    bolt = 0;
    /*并发进程 P(1), P(2)... P(n)*/
}
```

第一个进程进去 bolt 为0，testset 出来以后 bolt 值为1. 然后 while 判断条件为假，进程进入临界区；

后面的进程进到 testset 函数时 bolt 已为1，while 判断条件为真，然后就一直在这个 while 循环里面进行循环等待直至第一个进程出临界区后将 bolt 重新置为0，再次发生上面的情况。这种循环等待的方法就称为忙等待。

再举一个例子：比如说实现值交换的 exchange 函数

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

例子：

```
int const n = /*进程数目*/
int bolt;
void P(int i) {
    int keyi;
    while(true) {
        keyi = 1;
        while(keyi != 0)
            exchange(keyi, bolt);
        /*临界区*/
        exchange(keyi, bolt);
        /*剩余区*/
    }
}
void main() {
    bolt = 0;
    /*并发进程 P(1), P(2), ... P(n)*/
}
```

第一个进程进去时 bolt = 0，keyi = 1. keyi 不为0，发生交换使 bolt = 1，keyi = 0；此时 keyi 为0，跳出循环进入临界区。

后面的进程进去时 bolt 已经为1，keyi 为1；无论怎么交换两者最终都为1，因此一直在循环体里做无意义的交换来等待进入临界区。直至第一个进程从临界区出来将 bolt 和 keyi 的值换回来，再次发生上面的情况。这种循环等待就称为忙等待。

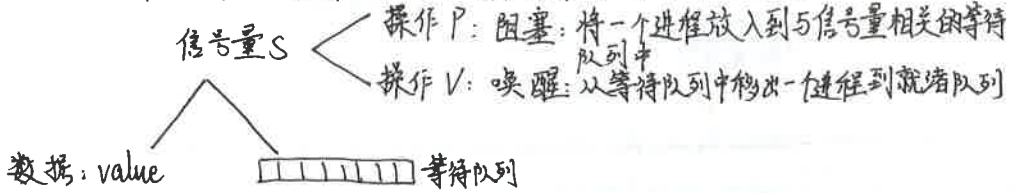
* 从上面的例子可以看出忙等待具有两大缺点：

① n个进程中 n-1 个进程都在用CPU做无用功，浪费CPU时间

② 会削弱系统可靠性，加重用户的编程负担（因为很容易编出死循环）

* 后来从交通灯那里得到启发提出了信号量这种同步工具，信号量就好比信号灯，它会告诉进程什么时候"等"，什么时候"走"

* 对于每个信号量来说定义了两种操作：P操作和V操作



* 开始先有二值信号量（即信号量只会在0、1中取值，实现较为简单）

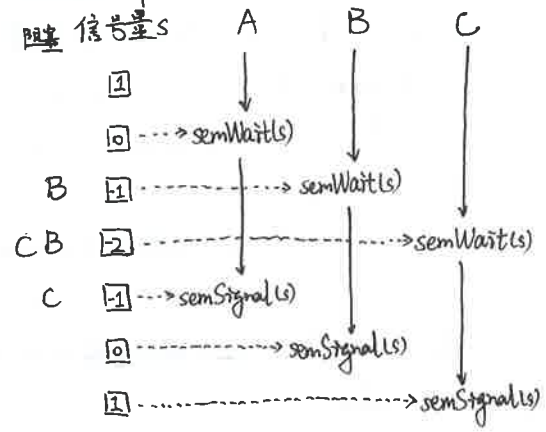
```
struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
}
void semWaitB(binary_semaphore S) {
    if (S.value == 1)
        S.value = 0;
    else {
        place this process in S.queue;
        block this process;
    }
}
void semSignalB(binary_Semaphore S){
    if (s.queue.is_empty())
        S.value = 1;
    else {
        place this process p from S.queue;
        place process p on ready list;
    }
}
```

* 一个信号量可初始化为0或1
* semWaitB操作检查信号量的值，若值为0，则执行semWaitB的进程被阻塞（进入队列）。否则将值改为0，继续执行
* semSignalB操作检查是否有任何进程在该信号上受阻，若有，受阻的进程就会被唤醒，若没有则将信号量设为1.

* 从二值信号量引申出了多值信号量

假设有三个进程A、B、C共享一个临界资源

```
const int n = /*进程数量*/
Semaphore S = 1;
void P(int i) {
    while (true) {
        semWait(S);
        /*临界区*/
        semSignal(S);
        /*剩余区*/
    }
}
void main() {
    /*并发进程 P(1), P(2), ..., P(n)*/
}
```



循环体：value = 1

A进入：value：1 → 0，0 ≮ 0，A进入临界区

B进入：value：0 → -1，-1 < 0，B进入等待队列，拿不到CPU

C进入：value：-1 → -2，-2 < 0，C进入等待队列，拿不到CPU

A出来：value：-2 → -1，-1 ≤ 0，B被唤醒，出等待队列，进临界区

B出来：value：-1 → 0，0 ≤ 0，C被唤醒，出等待队列，进临界区

C出来：value：0 → 1，1 ≰ 0，没有进程被唤醒

从这里可以看出：

信号量 S.count
- ≥0：表示可用临界资源实体数（即临界区还可进多少个进程）
- <0：绝对值表示挂起在 S.queue 队列中的进程数（即有多少个进程在排队）

* 使用信号量貌似避免了进程陷入忙等待(因为把线程放在了等待列表里，从而拿不到CPU，不会占领CPU做无用功)

* 但是信号量定义的P、V操作都利用了同一个变量 value，一个将 value ++，另一个将 value --。

```
wait (S) {
    value --;
    if (value < 0) {
        add this process to waiting queue
        block();
    }
}
```

```
Signal (S) {
    value ++;
    if (value <= 0) {
        remove a process P from waiting queue
        wakeup(P);
    }
}
```

假设一种情况，有两个进程A、B共享一个临界资源

value = 1

A进入：value：1 → 0，0 ≠ 0，A进入临界区

A出来：value：0 → 1 (此时被抢占)

B进入：value：1 → 0 (此时被抢占)

出现一种情况：从等待队列中唤醒一个进程，同时由于value ≥ 0，B也可进入临界区

因而就有两个进程同时进入临界区的风险

~~* 由此~~

还有一种情况，还存在同一时刻两个进程对同一信号进行P、V操作的可能

* 由此信号量实现成了临界区问题，因为P、V操作代码都在临界区中

而在临界区中就可能存在忙等待，只是由于它们的代码较短，相比起应用程序的忙等待会好得多

* 但是忙等待是不可避免的.

A+

Week 7 H.W.

1. A deadlock can occur only if four necessary conditions hold simultaneous in the system. Please describe and explain these four necessary conditions.

当系统中四个必要条件同时发生时才会产生死锁。请阐述这四个必要条件是什么？

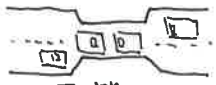
答：首先我们引入一个过桥模型：桥为资源，车为进程

然后再来理解这四个必要条件：

1. 互斥：在某一时刻某个资源只能被一个进程访问（例如：打印机）

* 这就好比一座窄桥，仅容一车通过

* 把桥拓宽（某个资源能被多个进程同时访问）是解决方法，但这不是对什么资源都有效的（比如打印机就不行）



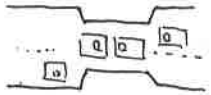
死锁

无死锁

2. 占有并等待：一个进程占有至少一个资源，并等待另一资源，而该资源为其它进程所占有

* 这就好比一座窄桥，两辆对向而行的车都在桥上，各自占用了半边桥，又在等待另外半边桥，结果两辆车都过不去.

* 等待不占有，占有不等待是解决办法（保证桥上不可能有车挡你再上桥，否则不允许上桥），但可能发生饥饿且资源利用率低

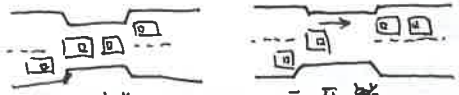


死锁

无死锁

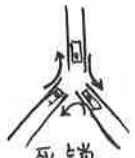
3、非抢占：资源不能被抢占，即资源只能在进程完成任务后自动释放

* 好比两辆车都在桥上，互不相让
* 把非抢占改成抢占是解决办法（好比其中是一辆车和一辆推土机，推土机能把车推回去，从而自己可过桥），但这样损失非常大（至少一个进程需要重新运行）



4、循环等待：有一组等待进程{$P_0, P_1, \dots, P_n$}，P_0等待的资源为P_1所占有，P_1等待的资源为P_2所占有…P_{n-1}等待的资源为P_n所占有，P_n等待的资源为P_0所占有.

* 好比下面一种情形



* 可制定优先级规则，按优先级进行处理，但资源利用率低.

2、Consider the following snapshot of a system
考虑下面这个系统在某一时刻的状态

	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. If 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?

使用银行家算法回答下面问题：

a. Need 矩阵的内容是怎样的？

b. 系统是否处于安全状态？

c. 如果从进程 P_1 发来一个请求 (0,4,2,0)，这个请求能否立刻被满足？

sol：a. 在银行家算法中，数据结构为

n个进程，m类资源

Available：长度为m的向量，表示每种资源现有的实例数。如果 Available[j]=k，那么资源类型 R_j（第j类资源）有k个实例

Max：n×m 矩阵，表示每个进程的最大需求。如果 Max[i][j]=k，那么进程 P_i 最多需要第j类资源的k个实例

Allocation：n×m 矩阵，表示每个进程现在所分配的各类资源的实例数。如果 Allocation[i][j]=k，那么进程 P_i 已被分配了第j类资源的k个实例

Need：n×m矩阵，表示每个进程还需要的剩余资源。如果 Need[i][j]=k，那么进程 P_i 还可能申请第j类资源的k个实例

其中 Need[i][j] = Max[i][j] - Allocation[i][j]

	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. 安全状态检测算法

① 设 Work 和 Finish 分别为长度为 m 和 n 的向量。按如下方式初始化

Work = Available

Work			
A	B	C	D
1	5	2	0

对于 i = 0, 1, ... n-1, Finish[i] = false

	Finish
P_0	false
P_1	false
P_2	false
P_3	false
P_4	false

② 查找这样的 i 使其满足

(a) Finish[i] = false

(b) $Need_i \le Work$ (需要的小于现有的)(找一个系统可满足其执行的进程)

如有: Work = Work + Allocation$_i$ (释放那个进程所占所有资源)

Finish[i] = true (已完成进程 i)

直至找不到这样的进程

∵

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

Work			
A	B	C	D
1	5	2	0

发现 $Need_0 \le Work$

$\begin{cases} 0 \le 1 \\ 0 \le 5 \\ 0 \le 2 \\ 0 \le 0 \end{cases}$

因此 Work = Work + Allocation$_i$

= Work

A	B	C	D
1	5	3	2

Finish[0] = true

∵

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

Work			
A	B	C	D
1	5	3	2

发现 $Need_2 \le Work$

$\begin{cases} 1 \le 1 \\ 0 \le 5 \\ 0 \le 3 \\ 2 \le 2 \end{cases}$

因此 Work = Work + Allocation$_i$

= Work

A	B	C	D
2	8	8	6

Finish[2] = true

∵

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

Work			
A	B	C	D
2	8	8	6

发现 $Need_1 \le Work$

$\begin{cases} 0 \le 2 \\ 7 \le 8 \\ 5 \le 8 \\ 0 \le 6 \end{cases}$

因此 Work = Work + Allocation$_i$

= Work

A	B	C	D
3	8	8	6

Finish[1] = true

∵

Need			
	A	B	C	D
P_3	0	0	2	0
P_4	0	6	4	2

Work			
A	B	C	D
3	8	8	6

发现 $Need_3 \le Work$

$\begin{cases} 0 \le 3 \\ 0 \le 8 \\ 2 \le 8 \\ 0 \le 6 \end{cases}$

因此 Work = Work + Allocation$_i$

= Work

A	B	C	D
3	14	11	8

Finish[3] = true

∵

Need			
	A	B	C	D
P_4	0	6	4	2

Work			
A	B	C	D
3	14	11	8

发现 $Need_4 \le work$

$\begin{cases} 0 \le 3 \\ 6 \le 14 \\ 4 \le 11 \\ 2 \le 8 \end{cases}$

因此 Work = Work + Allocation$_i$

= Work

A	B	C	D
3	14	12	12

Finish[4] = true

④ 此时所有 i，Finish[i] = true，因此系统处于安全状态（安全序列<P0, P2, P1, P3, P4>）

c. 进程 P_i 的资源请求算法

设 $Request_i$ 为进程 P_i 的请求向量。如果 $Request_i[j]=k$，那么进程 P_i 需要资源（第j类）的实例数为 k。当进程 P_i 作出资源请求时采取如下动作：

在这种情况下 $Request_i$ 为 (0, 4, 2, 0)

① 判断 $Request_i$ 是否小于等于 $Need_i$，如不是则说明这次需要的实例数已经超出其总共需要的最大实例数，这种情况是不被允许的，因而会产生出错条件。

$Request_i$ = (0, 4, 2, 0)　$Need_i$ = (0, 7, 5, 0)

$Request_i \leq Need_i$，往下进行

② 判断 $Request_i$ 是否小于等于 Available，如不是则说明没有足够可用资源实例进行分配，进程 P_i 必须等待。

$Request_i$ = (0, 4, 2, 0)　Available = (1, 5, 2, 0)

$Request_i \leq Available$

③ 假设系统将那么多的资源分配给了进程 i（不是真的进行了分配，只是一种假设），那么我们将按如下方式修改状态

$Available = Available - Request_i$（满足需求）

Available = (1, 5, 2, 0) − (0, 4, 2, 0) = (1, 1, 0, 0)

$Allocation_i = Allocation_i + Request_i$（分配资源）

$Allocation_i$ = (1, 0, 0, 0) + (0, 4, 2, 0) = (1, 4, 2, 0)

$Need_i = Need_i - Request_i$（进程剩余所需）

$Need_i$ = (0, 7, 5, 0) − (0, 4, 2, 0) = (0, 3, 3, 0)

现在此时刻状态为

Allocation

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

Max

	A	B	C	D
	0	0	1	2
	1	7	5	0
	2	3	5	6
	0	6	5	2
	0	6	5	6

Available = (1, 1, 0, 0)

Need

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

④ 再进行安全状态检测，若此状态为安全状态，则说明 P_1 发来的请求能立即满足；否则，则说明该请求不能被满足

1. 设 Work 和 Finish 分别为长度为 m 和 n 的向量，按如下方式初始化

Work = Available = (1, 1, 0, 0)　Finish = (false, false, false, false, false)

2. 查找这样的 i 使其满足

(a) Finish[i] = false

(b) $Need_i \leq Work$（需要的小于现有的）（找一个系统可满足其执行的进程）

如果有：$Work = Work + Allocation_i$（释放那个进程所占所有资源）

Finish[i] = true（已完成进程 i）

直至找不到这样的进程

Need

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

Work = (1, 1, 0, 0)　发现 $Need_0$ = (0, 0, 0, 0) ≤ Work = (1, 1, 0, 0)

因此 $Work = Work + Allocation_0$ = (1, 1, 0, 0) + (0, 0, 1, 2) = (1, 1, 1, 2)

Finish = (true, false, false, false, false)

Need

	A	B	C	D
P_1	0	3	3	0
P_2	1	0	0	2
P_3	0	0	2	0
P_4	0	6	4	2

Work = (1, 1, 1, 2)　发现 $Need_2$ = (1, 0, 0, 2) ≤ Work = (1, 1, 1, 2)

因此 $Work = Work + Allocation_2$ = (1, 1, 1, 2) + (1, 3, 5, 4) = (2, 4, 6, 6)

Finish = (true, false, true, false, false)

∵

	Need A	B	C	D
P_1	0	3	3	0
P_3	0	0	2	0
P_4	0	6	4	2

Work = (2, 4, 6, 6)　　发现 $Need_1$ = (0, 3, 3, 0) ≤ Work = (2, 4, 6, 6)

因此: Work = Work + $Allocation_1$ = (2, 4, 6, 6) + (1, 4, 2, 0) = (3, 8, 8, 6)

Finish = (true, true, true, false, false)

∵

	Need A	B	C	D
P_3	0	0	2	0
P_4	0	6	4	2

Work = (3, 8, 8, 6)　　发现 $Need_3$ = (0, 0, 2, 0) ≤ Work = (3, 8, 8, 6)

因此: Work = Work + $Allocation_3$ = (3, 8, 8, 6) + (0, 6, 3, 2) = (3, 14, 11, 8)

Finish = (true, true, true, true, false)

∵

	Need A	B	C	D
P_4	0	6	4	2

Work = (3, 14, 11, 8)　　发现 $Need_4$ = (0, 6, 4, 2) ≤ Work = (3, 14, 11, 8)

因此: Work = Work + $Allocation_4$ = (3, 14, 11, 8) + (0, 0, 1, 4) = (3, 14, 12, 12)

Finish = (true, true, true, true, true) ⇒ 处于安全状态.

∴ P_1 发来的请求可以立即满足

A+

3. Consider a system consisting of m resources of the same type, being shared by n processes. Resources can be requested and released by processes only one at a time. Show that the system is deadlock free if the following two conditions hold:

a. The maximum need of each process is between 1 and m resources

b. The sum of all maximum needs is less than m + n

假设一个系统有m个相同类型的资源被n个进程共享，进程每次只请求或释放一个资源，试证明只要符合下面两个条件，系统就不会发生死锁。

a. 每个进程需要资源的最大值在 1 ~ m 之间

b. 所有进程需要资源的最大值的和小于 m + n

答：由题意得对于进程 P_i 来说 Max_i 在 1 ~ m 之间，$\sum_{i=0}^{n-1} Max_i < m+n$

由于 $Need_i + Allocation_i = Max_i$

因此 $\sum_{i=0}^{n-1} Need_i + \sum_{i=0}^{n-1} Allocation_i = \sum_{i=0}^{n-1} Max_i < m+n$

假如存在死锁（反证法）

说明所有资源已经分配给每个进程了，即 $\sum_{i=0}^{n-1} Allocation_i = m$

由于 $\sum_{i=0}^{n-1} Need_i + \sum_{i=0}^{n-1} Allocation_i < m+n$

可得出 $\sum_{i=0}^{n-1} Need_i < n$，因此必有进程 P_j 不需要资源

由于 Max_j 在 1 ~ m 之间，而 $Need_j = 0$，说明 $Allocation_j > 0$

而进程 P_j 又拿到了其所掌握的所有资源，因而可执行，执行后至少释放一个资源

由于我们一次只请求一个资源，因而可满足请求，不存在死锁

No.

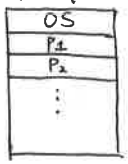
Date

Week 8 H.W.

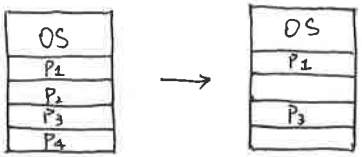
1. Please explain the difference between internal and external fragmentation.
请阐述内部碎片和外部碎片的区别.

答：首先先来看看这两者是怎么回事

* 外部碎片：
  * 怎样将进程加载到内存中，最简单的办法就是依次往下放



  * 但这会有一个问题，假设原有四个进程，占据内存空间大小依次为100KB，200KB，300KB，400KB. 此时进程P2，P4执行完退出内存了.



  * 此时来了一个占据内存空间大小为500KB的进程P5.
  * 本来P2和P4让出来的内存空间为600KB，是大于进程P5所需的500KB的内存空间的. 但P2那一块内存空间为200KB，P4那一块内存空间为400KB，均小于P5所需的500KB内存空间，因而不能完成分配(没有足够大的连续内存空间)
  * 而这两块让出来的内存空间就好像是内存上的孔洞，这也就是所谓的外部碎片
  * 如何解决这个问题：我们可以看到要是能把500KB的进程P5切成两段分开放，不就可以放进去了
  * 由此有了分页的思想：将内存分为很小的帧，将进程也分为很小的块(块和帧大小相同)，然后将每一块放入内存的帧中

这样有两个问题：帧太小 ⇒ 页表的规模很大

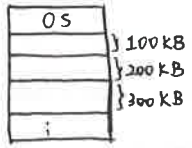
帧太大 ⇒ 最后一帧的内部碎片可能很大

  * 同时我们也可用压缩的办法将不连续的孔洞变成连续空间

No.

Date

* 内部碎片：
  * 还有一种办法，就是先将内存分好大小固定的块，然后来一个进程就根据进程占据内存空间大小进到对应的块中(打好格子再往里放)



  * 但这同样存在一个问题，如上图分好固定大小的块，这时来一个占据内存空间大小为250KB的进程P
  * 由于第一个固定块只有100KB，不足250KB，因此放不进去；第二个固定块只有200KB也不足所需的250KB，因而也放不进去；第三个固定块为300KB，足够放入250KB的进程P，因此将其放进去
  * 这时就出现了一个问题，我们分配给了进程P 300KB的内存空间，但进程P只会用到其中250KB的内存空间，而有50KB的内存空间不会被用到
  * 这种分配出去却没有用到的内存称为内部碎片
  * 如何解决这个问题：改成连续存放，不对块的大小进行固定

但这样外部碎片很大

  * 因此我们会将每个进程先分小段，这些小段连续存放在内存中，这样做可减少外部碎片(1KB的外部碎片总比100KB的外部碎片要好)

由此可以下个总结：

外部碎片：随着进程装入和移出内存，空闲内存空间被分小片段。当所有总的可用内存之和可以满足请求，但并不连续，就出现了外部碎片。外部碎片可通过压缩或分页解决.

内部碎片：将内存以固定大小的块为单元来分配，当进程所分配的内存比所要的大时，这两值之差称为内部碎片. 内部碎片可通过连续存放解决.

2. Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?

假设有五个大小分别为100 KB, 500 KB, 200 KB, 300 KB和600 KB按顺序放在内存中的内存块，我们采用首次适配、最佳适配、最差适配三种适配方法将按顺序到来的大小为212 KB, 417 KB, 112 KB和426 KB的四个进程放入内存中的过程是怎样的？同时请回答哪种适配方法的内存利用率最高？

答：首先解释一下这三种适配方法：

首次适配：将进程放到第一个可容纳它的内存块中

最佳适配：将进程放到能容纳它的最小内存块中

最差适配：将进程放到能容纳它的最大内存块中

然后再来看具体过程：

①首次适配：

初始：100 KB　500 KB　200 KB　300 KB　600 KB

此时来第一个进程，需要212 KB，由于是首次适配，可看到大于等于212 KB的第一个内存块为500 KB，因此将它放进去。原来500 KB的内存块被分出212 KB，还留下了288 KB内存块

P_1后：100 KB　288 KB　200 KB　300 KB　600 KB

此时来第二个进程，需要417 KB，由于是首次适配，可看到大于等于417 KB的第一个内存块为600 KB，因此将它放进去。原来600 KB的内存块被分出417 KB，还留下了183 KB内存块

P_2后：100 KB　288 KB　200 KB　300 KB　183 KB

此时来第三个进程，需要112 KB，由于是首次适配，可看到大于等于112 KB的第一个内存块为288 KB，因此将它放进去。原来288 KB的内存块被分出112 KB，还留下了176 KB内存块

P_3后：100 KB　176 KB　200 KB　300 KB　183 KB

此时来第四个进程，需要426 KB，但内存中却没有那么大的内存块可分配给它，因而不能完成分配

P_4不能被分配

②最佳适配

初始：100 KB　500 KB　200 KB　300 KB　600 KB

此时来第一个进程，需要212 KB，由于是最佳适配，可看到大于等于212 KB的最小内存块为300 KB，因此将它放进去。原来300 KB的内存块被分出212 KB，还留下了88 KB内存块

P_1后：100 KB　500 KB　200 KB　88 KB　600 KB

此时来第二个进程，需要417 KB，由于是最佳适配，可看到大于等于417 KB的最小内存块为500 KB，因此将它放进去。原来500 KB的内存块被分出了417 KB，还留下了83 KB内存块

P_2后：100 KB　83 KB　200 KB　88 KB　600 KB

此时来第三个进程，需要112 KB，由于是最佳适配，可看到大于等于112 KB的最小内存块为200 KB，因此将它放进去。原来200 KB的内存块被分出了112 KB，还留下了88 KB内存块

P_3后：100 KB　83 KB　88 KB　88 KB　600 KB

此时来第四个进程，需要426 KB，由于是最佳适配，可看到大于等于426 KB的最小内存块为600 KB，因此将它放进去。原来600 KB的内存块被分出了426 KB，还留下了174 KB内存块

P_4后：100 KB　83 KB　88 KB　88 KB　174 KB

③最差适配

初始：100KB　500KB　200KB　300KB　600KB

此时来第一个进程，需要212KB，由于是最差适配，可看到大于等于212KB的最大内存块为600KB，因此将它放进去。原来600KB的内存块被分出212KB，还留下了388KB内存块

P1后：100KB　500KB　200KB　300KB　212KB　388

此时来第二个进程，需要417KB，由于是最差适配，可看到大于等于417KB的最大内存块为500KB，因此将它放进去。原来500KB的内存块被分出417KB，还留下了83KB内存块

P2后：100KB　83KB　200KB　300KB　212KB

此时来第三个进程，需要112KB，由于是最差适配，可看到大于等于112KB的最大内存块为300KB，因此将它放进去。原来300KB的内存块被分出112KB，还留下了188KB内存块

P3后：100KB　83KB　200KB　188KB　212KB

此时来第四个进程，需要426KB，但内存中却没有那么大的内存块可分配给它，因而不能完成分配

→ P4不能被分配

最后再来分析内存利用率

* 只有最佳适配能完成给全部四个进程分配内存的任务，而首次适配和最差适配都只完成了给三个进程分配内存的任务，因此最佳适配的内存利用率最高
* 首次适配留下的最大内存块为300KB，而最差适配留下的最大内存块为212KB，因此可认为首次适配要比最差适配好一些

A

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

(a) If a memory reference takes 200 nanoseconds, how long does a paged memory reference take?

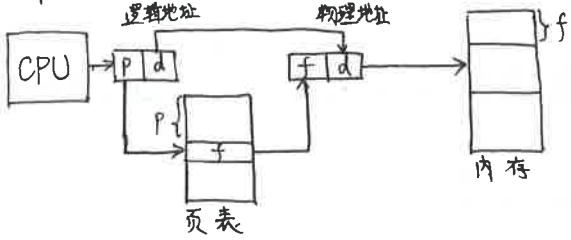
(b) If we add associative registers, and 75 percent of all page-table references are found in the associative registers, what is the effective memory reference time? (Assume that finding a page-table entry in the associative registers takes zero time, if the entry is there.)

假设一个将页表存放在内存的分页系统：

a. 如果一次内存访问需要200ns，访问一页内存要用多长时间？

b. 如果加入TLB，并且75%的页表引用发生在TLB，内存有效访问时间是多少？(假设在TLB中查找页表项占用零时间，如果页表项在其中。)

答：a. 对于没有页表的情况，内存访问图示如下：

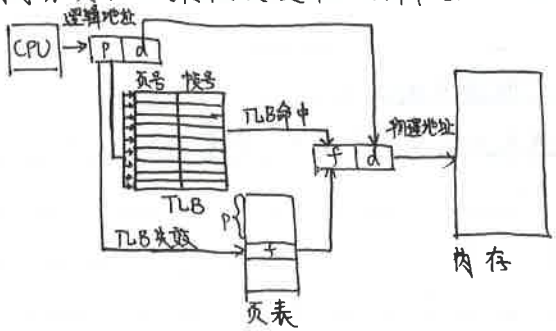


访问一次页表（页表在内存中）相当于访问一次内存，需200ns

从页表再映射到内存中特定位置，也需访问一次内存，需200ns

因此一共需要400ns

b. 对于加入TLB（转换表缓冲区）的情况，内存访问图示如下：



由于TLB中查找页表占用零时间 ⇒ 只要页表项在TLB中，只需访问一次内存 ⇒ 200ns

由于在内存中访问页表需访问一次内存 ⇒ TLB失效时，需访问两次内存 ⇒ 400ns

由于命中率为75%，则用时 = 200 × 75% + 400 × 25%

= 250 ns

Week 9 H.W.

1. 讨论在哪一种情况下，LFU(最不经常使用)页置换比LRU(最近最少使用)页置换法产生较少的页面错误，什么情况下则相反？

解：LRU是最近最少使用页置换法，即首先淘汰最长时间未被使用的页面

LFU是最近最不经常使用页置换法，即首先淘汰一定时期内被访问次数最少的页面

这两者看似相似，实际不同，

可以举一个例子，假设页面请求序列为 1, 1, 1, 1, 2, 3, 4，主存块为3

对于LRU为：

时刻	0	1	2	3	4	5	6
P	1	1	1	1	2	3	4
	1	1	1	1	2	3	3
M					1	2	2
						1	4

此时由于页面1最长时间未被使用，因而页面1被置换出来，页面4置换进去

对于LFU为：

时刻	0	1	2	3	4	5	6
P	1	1	1	1	2	3	4
	1	1	1	1	2	3	3
M					1	2	4
						1	1

此时由于页面2的访问次数最少(仅一次)而被置换出来，页面4被置换进去

要回答这个问题首先要明确两种算法的优劣

对于LRU来说处理由某一进程所主导的内存访问为势较大，比如说下面这个页面请求序列(假设主存块为3)

时刻	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P	1	1	2	3	4	1	1	3	4	1	1	6	7	8	1	1	4
	1	1	2	3	4	3	3	3	3	3	3	6	6	6	1	1	1
M			1	2	3	2	2	2	4	4	4	4	7	7	7	7	7
				1	2	1	1	1	1	1	1	1	1	8	8	8	4

错误率：8/16 = 50%

而同样序列对于LFU来说：

时刻	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P	1	1	2	3	4	1	1	3	4	1	1	6	7	8	1	1	4
	1	1	2	3	4	3	3	3	3	3	3	6	6	6	1	1	1
M			1	2	3	4	4	4	4	4	4	4	7	7	7	7	7
				1	2	1	1	1	1	1	1	1	1	8	8	8	4

错误率：4/16 = 25%

从此可以看出LFU适合那种有少数进程大量访问内存，多数进程少量访问内存的情况，因为LFU可以保证那少数的进程很少机会被置换走

但与此同时要是所有进程都存在密集访问的情况，LFU容易出现一个问题，那就是某一进程的大量访问内存会导致在它们空闲时一直占据内存的情况

比如说下面这个页面请求序列(假设主存块为3)

时刻	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
P	1	1	1	1	1	1	2	2	2	2	2	3	4	5	4	3	5	4	5	3	3
	1	1	1	1	1	1	2	2	2	2	2	3	4	5	4	3	5	4	5	3	3
M							2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
							1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

错误率：8/20 = 40%

但对于LRU来说同样序列下：

时刻	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
P	1	1	1	1	1	1	2	2	2	2	2	3	4	5	4	3	5	4	5	3	3
	1	1	1	1	1	1	2	2	2	2	2	3	4	5	4	3	5	4	5	3	3
M							2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
							1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

错误率：2/20 = 10%

而且集中访问量越大，占据内存的时间越长，这种为势越明显

2、颠簸的原因是什么？系统怎样检测颠簸？一旦系统检测到颠簸，系统怎样做来消除这个问题？

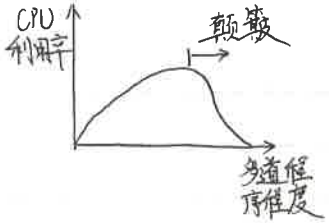
答：由于进程访问内存存在局部性的特点，即某些时段内集中访问内存，其他时段几乎不对内存发起访问。在这局部密集访问内存的时期内如果访问存储单元容量大于物理内存大小则会造成大量的页面错误。而由于页面错误率很高，系统将忙于换入换出页而导致CPU利用率很低。然而操作系统却可能误以为是加载程序太少而导致的低CPU利用率，因而会增加多道程序程度，这会使得原本不足的内存雪上加霜，出现恶性循环。

系统CPU利用率变低 → 多放进程

↑ ↓

拉低CPU利用率 ← 每个进程分配的物理内存变少，会产生大量页面错误

而CPU利用率和多道程序程度之间又满足下图关系：



检测颠簸的做法大致有：

(1) 比对多道程序程度与CPU利用率间的关系

若发现增加多道程序程度(多放进程)后CPU利用率不升反降，则说明出现颠簸

(2) 检测某一时刻访问存储单元容量是否大于物理内存大小

难实现，因为前者很难检测到，不过也是一种思路

而当出现颠簸以后解决办法在于让每个进程分配的物理内存变多，让其不再频繁换入换出页。而我们不能增加物理内存，唯有的办法是减少进程，降低多道程序程度。

A+

3、什么是写时拷贝功能，在什么情况下有利？需要什么硬件支持？

* 写时拷贝允许父进程和子进程一开始在内存中共享相同的页，只有当某个进程修改到共享页时，共享页才会发生拷贝。
* 写时拷贝会使得进程创建更高效，因为只有修改的页才发生拷贝，其他相同的页则共享。
* 需要空闲缓冲池的支持，拷贝时空闲页从空闲缓冲池中来，并采用按需填零技术清除之前内容。空闲缓冲池是一个放有空闲页的帧栈。

Week 10

1. 解释 open 和 close 操作的目的

答：open 操作：

1. 将在目录结构上搜索相应元素，并将该内容移动到内存中
2. 会根据文件名搜索目录，并将目录条目复制到打开文件表
3. 接受访问模式参数(创建、只读、读写、添加等)，并根据文件许可位进行检查。如果请求模式获得允许就打开文件
4. 返回一个指向打开文件表中一个条目的指针，以避免进一步搜索和简化系统调用接口
5. 将文件打开计数器递增

close 操作：

1. 将内存中的对应元素的内容移动到磁盘的目录结构中。
2. 删除打开文件表中对应条目
3. 将文件打开计数器递减。如果减到零，将关闭文件。

2. Consider a system that supports 5000 users. Suppose that you want to allow 4990 of these users to be able to access one file.

a. How would you specify this protection scheme in UNIX?

b. Can you suggest another protection scheme that can be used more effectively for this purpose than the scheme provided by UNIX?

考虑一个支持5000个用户的系统。假设我们想让其中4990个用户可以访问某一文件

a. 在UNIX中我们如何指定这样的保护机制

b. 我们可以在UNIX机制之外再想出一个更高效的保护机制

No.

Date

答：(1) 在UNIX环境下一个目录的列表中定义了文件或目录的权限(在第一个域中，如果开头为d说明为子目录。此外还有文件链接数、拥有者名称、组名称、文件字节数、上次修改时间和文件名称

那也就是说，我们可以将那4990个用户归到一个群组中去，然后设定该群组的访问权限为rwx（可读写执行），然后对于其它的用户来说访问权限为---（不可读不可写不可执行），就像下面的目录列表条目所示（这里假设文件为program.c，修改时间为Feb 24 2013，大小为1024，拥有者为pbg，那4990个用户组成的组叫staff）

-rwxrwx--- 1 pbg staff 1024 Feb 24 2013 program.c

当然还存在另一种情况，可以不需要将那4990个用户归到一个群组，而是将其名字一一列出，然后一一规定每个具体用户的权限。这种做法好处在于灵活性高（到时要对某个用户进行权限修改比较方便），劣势在于添加比较麻烦（因为需要一条条进行定义）

(2) UNIX方法的麻烦之处在于，赋予权限必须要由上而下进行，也就是说只能通过对有权限者进行设定，剩余的均默认为是无权限者。但我们可以看到，对这5000个用户来说绝大多数都是有权限者，极个别是无权限者。如果我们能对无权限者进行设定，告诉系统"那10个用户是没有权限的，然后剩余的用户默认均为有权限的，那我们就只需规定那10个用户的用户名就可以了。这样做当然比要规定4990个用户要来得快。 A+

3. Consider a file system where a file can be deleted and its disk space reclaimed while links to that file still exist.

a. What problem may occur if a new file is created in the same storage area or with the same absolute path name?

b. How can these problems be avoided?

No.

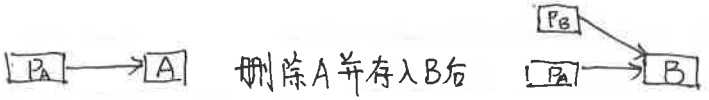
Date

考虑这样的一个文件系统，在这文件系统中文件可被删除，删除后的磁盘空间可用于其他文件，但文件指针被保留下来

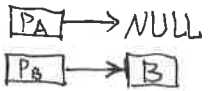
a. 当一个新文件创建在同一个存储区域或同一绝对路径下会出现什么问题

b. 如何避免这个问题

答：a. 如果有用户在不知情情况下获取到这个文件指针，那就很有可能对当前存储内容进行修改。如果改到某些重要部位（比如说引导区）可能对文件或对系统造成崩溃性影响，会产生严重后果（出现指针泄露）



b. 方法很简单，将文件指针置为NULL即可，



No.

Date

Week 11

1. Consider a file currently consisting of 100 blocks. Assume that the file control block (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous allocation cases, assume that there is no room to grow in the beginning, but there is room to grow in the end. Assume that the block information to be added is stored in memory.
   a. The block is added at the beginning
   b. The block is added in the middle
   c. The block is added at the end
   d. The block is removed from the beginning
   e. The block is removed from the middle
   f. The block is removed from the end

现有一个100块数据块的文件。假设这个文件的文件控制块(与用于索引分配的索引块)都在内存中。请计算连续分配、链接分配和(一级)索引分配机制中对一个块做如下所示的操作需要多少次的I/O操作。同时注意在连续分配机制中,头部已不可再增长空间,只可能对尾部增长空间。假设需要被加入的块来自内存中。

   a. 块被添加至头部
   b. 块被添加至中段(指被添加至原第50块与原第51块之间)
   c. 块被添加至尾部
   d. 块被从头部移除
   e. 块被从中段移除(指移除原第51块)
   f. 块被从尾部移除

No.

Date

①

答:首先先来看连续分配的情况

连续分配是把连续空闲的块分配给文件。这种分配方法的优点在于简单,既可以顺序读取也可以随机读取而且由于块是连续分配的因此顺序读取的速度快缺点在于会产生外部碎片,同时在文件扩展上存可能存在问题(预分配问题)。

连续分配的图示如下

目录

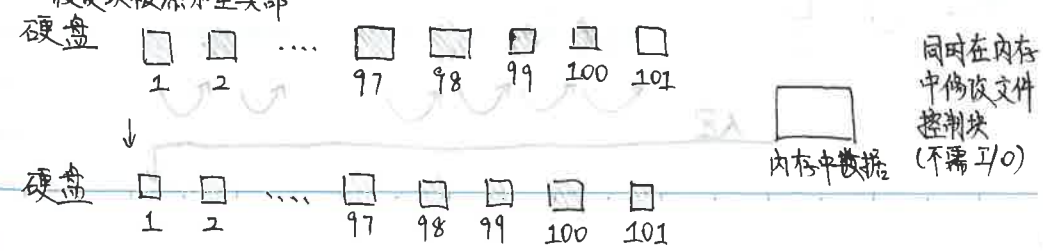
文件	起始	长度
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

硬盘

当在连续分配下需要添加块时,由于块头部不可以增长,只可对尾部增长空间的前提下,只能将块一项项往后移直至我们需要插入的位置出现空间才可完成添加工作因此在添加时满足下列公式: I/O数 = 需要移动的块数 × 2 + 1

(解释一下:因为每移动一块就需要将那一块的数据先从硬盘移出至内存再从内存写回到硬盘对应块,因此是一次读一次写的两次I/O操作,因此是乘上2;而最后加上的那个1是指我们将需要添加的块(也就是对应内存的数据)要写回到硬盘的对应块中,因此需要一次写操作)

图示:假设块被添加至头部



因此对于连续分配来说：

当块被添加至头部时：需要移动全部100块的数据，因此I/O数根据公式为：

$I/O = 100 \times 2 + 1 = 201$

当块被添加至中部时：需要移动全部50块的数据，因此I/O数根据公式为：

$I/O = 50 \times 2 + 1 = 101$

当块被添加至尾部时：不需移动任何数据块，因此I/O数根据公式为：

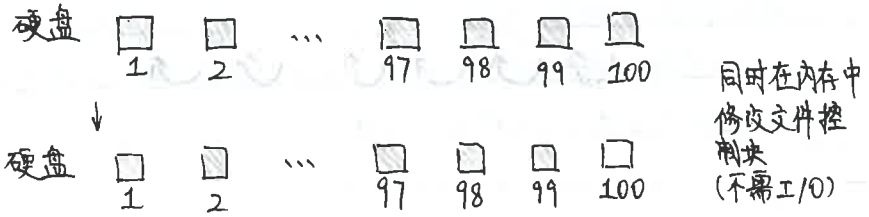
$I/O = 0 \times 2 + 1 = 1$

当在连续分配下需要完成移除工作时，由于我们不能随意更改文件的起始块序号（就好比在内存出现外部碎片时看着在实际情况下不能直接通过压缩的方法避免外部碎片一样，随意更改可能会使一些读取该文件的程序在读取上出现问题；而且这种做法可能会加剧外部碎片的问题（在头部又多出一块外部碎片），因此我们需要将后面块的数据一块一块依次往前挪（因为连续分配需保证块的连续性）方可实现块的移除

因此在移除时满足下列公式：I/O数 = 需要移动的块数 × 2

（解释一下：这里面要乘上2也是因为每移动一块就需要先将那一块的数据从硬盘写回内存，然后再从内存写回硬盘对应块，因此是一次读一次写的两次I/O操作）

图示：假设块被从头部移除



因此对于连续分配来说：

当块被从头部移除时：需要移动后面99块的数据，因此I/O数根据公式为：

$I/O = 99 \times 2 = 198$

当块被从中部移除时：需要移动后面49块的数据，因此I/O数根据公式为：

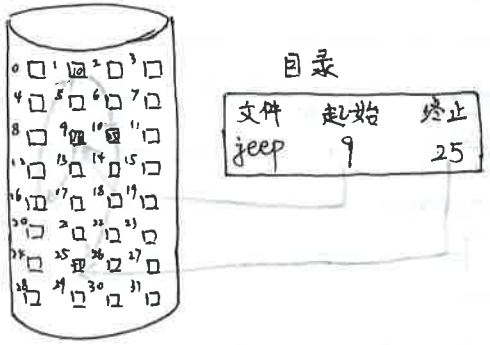
$I/O = 49 \times 2 = 98$

当块被从尾部移除时：不需移动任何数据块，因此I/O数根据公式为：

$I/O = 0 \times 2 = 0$

②其次我们来看链接分配的情况

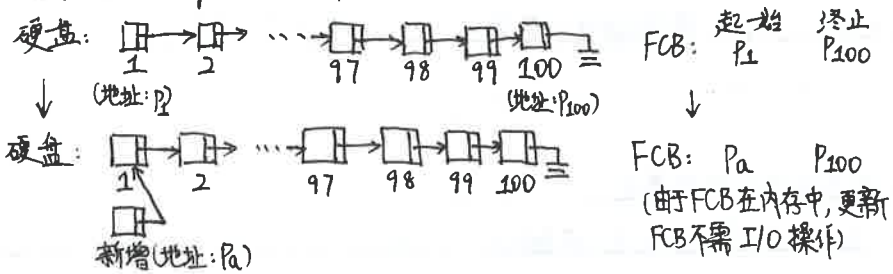
在链接分配中在创建文件时，每一块都包含下一块的指针。因此每个文件其实是一个个磁盘块的链表，而这些磁盘块可以零散地放置在磁盘各处。这种分配方法的优点在于可以将每一块零散地放置到磁盘各处，因此不存在外部碎片问题，提高了磁盘空间利用率。而且由于是链表操作，因而文件中数据块的增删扩充比较高效。但与此同时链表中的链接指针也会占用一定空间，同时也会带来可靠性问题（只要有任意一块出现故障，后面的块均无法读取，也就相当于后面的块全部故障）。同时链接分配由于用的是单向链表，因而只支持顺序读取，不支持随机读取。存取速度也比较慢（磁头要动来动去）。链接分配的图示如下：



因为链接分配实际上是链表操作，因此很受插入位置的影响。现在分成三类情况进行分析：

<1>块被添加至头部

由于目录中已经标好了该文件起始位置的块号，因此我们只需将新增块写入硬盘即可，图示如下：



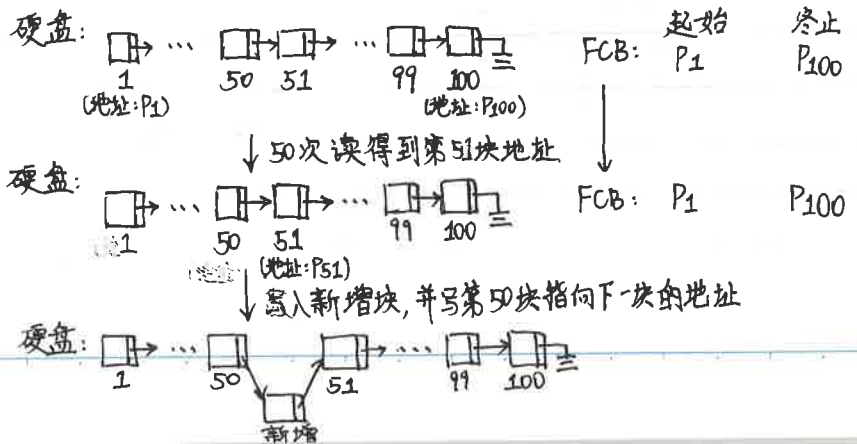
因此我们只需要将新增块写入硬盘这一次I/O操作

<2>块被添加至中部

由于是一个链表，因此我们需要从起始指针开始不断往下索引，直到索引至第50块。这里面一共是50次的I/O操作（50次读）

然后需要修改第50块的指针为新增块（1次写新增块，1次写第50块，共2次写）

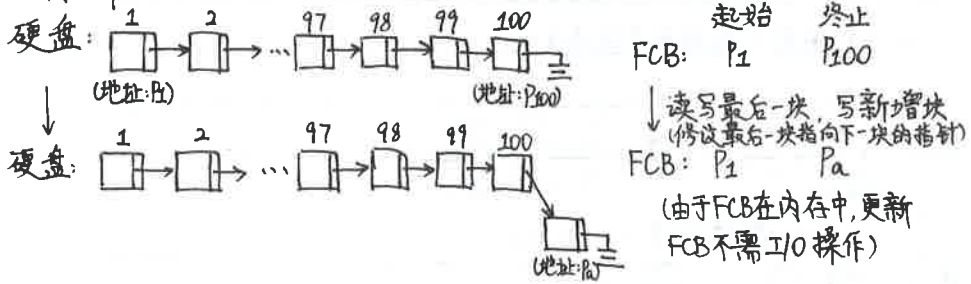
图示如下（一共52次I/O操作）



<3>块被添加至尾部

由于目录中已经标好了该文件终止位置的块号，因此我们只需修改最后一块的指针为新增块（1次读写最后一块，1次写新增块，共3次I/O操作）

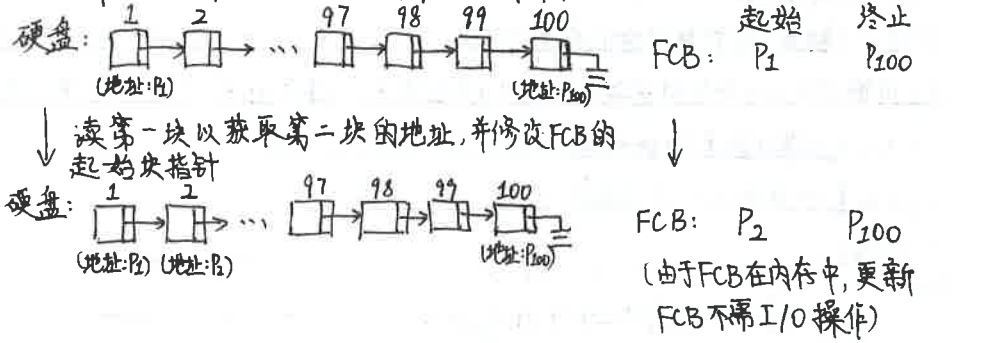
图示如下：



而对于移除块来说也很受移除块位置的影响。现在也分成三类情况进行分析：

<1>块被从头部移除

由于要更新起始指针，因此必定要拿到新的起始指针，而这个起始指针必须通过读起始块获取，因此需要1次的I/O操作，如下图所示



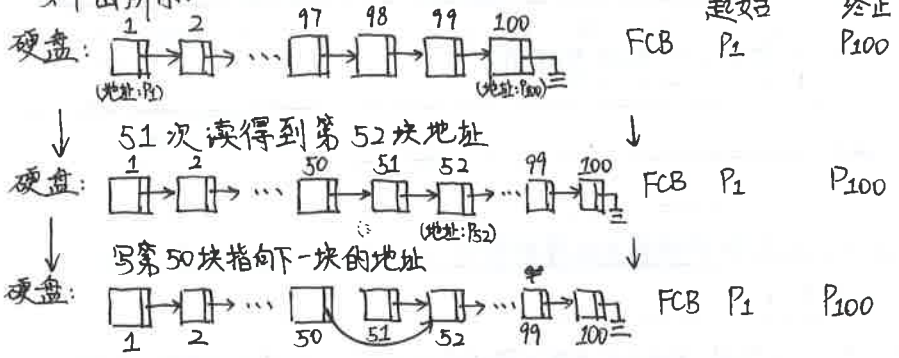
<2>块被从中部移除

由于是一个链表，因此需要我们从起始指针不断往下索引，直到索引到第50块，这里面一共是50次的I/O操作（50次读）

由于需要得到新指针（即得到第52块的地址），因此必预读第51块

同时我们需要将新指针写回第50块

如下图所示：

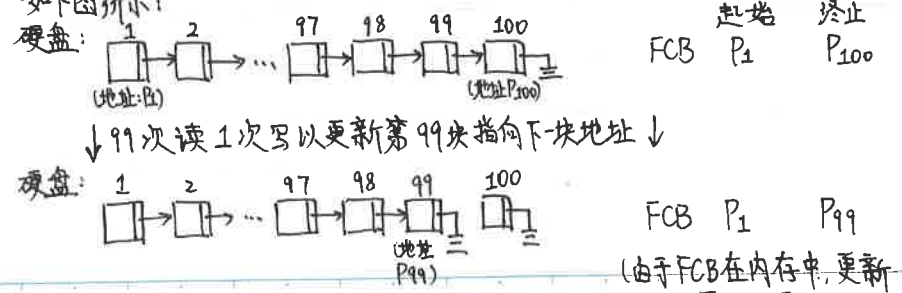


<3>块被从尾部移除

由于是一个链表，在移除时我们需要修改第99块的指向下一块的指针。而要修改第99块通过直接访问终止块指针是实现不了的（因为链表是一个单向链表，而终止块是第99块的下一块）。因此索引到第99块一共是99次的I/O操作（99次读）

由于要写第99块指向下一块的指针，因而还有一次写操作

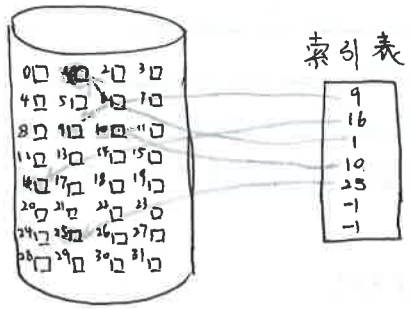
如下图所示：



③最后我们来看索引分配的情况

索引分配是将所有指针放在一起成为索引块。这样做既能顺序存取，又能随机存取，同时也满足了文件动态增长，插入删除的要求（因为只要操作索引块就行），也能充分利用外存空间（因为没有外部碎片）。问题在于索引表（块）本身就有开销（下图有样例）

由于本题中索引表在内存中，因此添加块只需将块写入，并将块号放到索引表中对应位置即可；而删除更简单，直接删掉索引表中对应块号即可。



因此对于索引分配来说添加1块均只需做1次I/O（将块写入），删除均不需I/O操作

④答案

	连续分配	链接分配	索引分配
a.	201	1	1
b.	101	52	1
c.	1	3	1
d.	198	1	0
e.	98	52	0
f.	0	100	0

2. Consider a file system that uses a modified 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 sizes, and these sizes are predetermined.

假设一个文件系统采用修改过的、支持扩展的连续分配算法。一个文件是一组扩展，每个扩展对应一个连续的块集合。这种系统中的关键问题是扩展大小的可变级别。下述机制的优点和缺点分别是什么。

a. 所有扩展都大小一样，这个大小是预定义的

b. 扩展可以是任意大小，并且动态分配

c. 扩展可以从预定义的一些大小中选取

答：由于传统的连续分配中文件不能扩展，因而出现了支持扩展的连续分配来解决这个问题。当连续分配的空间不够时，被扩展的连续空间会被添加至原来的分配中，使得一个文件可由一个或多个扩展组成，如下图所示：

a. 类似于内存分页，优点是没有外部碎片，缺点是可能存在内部碎片。如果预定义扩展块太大可能内部碎片很大，太小就会出现大量指针，硬盘空间利用率低。

b. 类似于连续分配，优点是没有内部碎片，缺点是会出现孔洞(外部碎片)

c. 上述二者的结合，优点也是没有外部碎片，缺点也是可能存在内部碎片。只是此时很好地平衡了内部碎片和硬盘空间利用率的问题

A+

Week 12

Suppose that a disk drive has 5000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending request, in FIFO order, is:

86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

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?

1. FCFS 2. SSTF 3. SCAN 4. LOOK 5. C-SCAN 6. C-LOOK

假设一个磁盘驱动器有5000个柱面，从0～4999。驱动器正在为柱面143的一个请求提供服务，且前面的一个服务请求是在柱面125。按FIFO顺序，即将到来的服务队列是

86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

从现在磁头位置开始，按照下面的磁盘调度法，要满足队列中的服务要求磁头总的移动距离是多少？

1. FCFS 2. SSTF 3. SCAN 4. LOOK 5. C-SCAN 6. C-LOOK

答：首先是FCFS，FCFS将会根据每个要求的到来顺序判断服务顺序，越先到达的请求越先服务。这样实现的好处在于简单，坏处在于可能寻道时间很长。

对于这题来说服务顺序是(按请求到来先后进行判定)

143 → 86 → 1470 → 913 → 1774 → 948 → 1509 → 1022 → 1750 → 130

总共寻道距离为 57 + 1384 + 557 + 861 + 826 + 561 + 487 + 728 + 1620

= 7081

No.

Date

其次是SSTF，SSTF将会选取距离目前磁头所在柱面的最近的请求先进行服务，这样做好处在于时间短但坏处在于可能由于磁头所在柱面附近连续请求，而导致较远位置的请求饥饿。

现磁头在柱面143，距此柱面位置最近的柱面为柱面130，先服务
现磁头在柱面130，距此柱面位置最近的柱面为柱面86，先服务
现磁头在柱面86，距此柱面位置最近的柱面为柱面913，先服务
现磁头在柱面913，距此柱面位置最近的柱面为柱面948，先服务
现磁头在柱面948，距此柱面位置最近的柱面为柱面1022，先服务
现磁头在柱面1022，距此柱面位置最近的柱面为柱面1470，先服务
现磁头在柱面1470，距此柱面位置最近的柱面为柱面1509，先服务
现磁头在柱面1509，距此柱面位置最近的柱面为柱面1750，先服务
现磁头在柱面1750，距此柱面位置最近的柱面为柱面1774，先服务

因此服务顺序是：143 → 130 → 86 → 913 → 948 → 1022 → 1470 → 1509
→ 1750 → 1774

总共寻道距离为：13 + 44 + 827 + 35 + 74 + 448 + 39 + 241 + 24
= 1745

再次是SCAN，SCAN称为电梯算法，会像电梯一样来回移动遇到请求即进行服务（磁头从初始柱面进行单方向移动，直至移到顶点然后再换方向移回来），这样做优点在于可能时间较短，缺点在于可能出现序号上差别不大，但寻道时间很长的情况（例如一个磁盘驱动器有5000个柱面，一开始磁头在柱面2，并在往3的方向移动，此时有一个对柱面1的服务请求，那就要等到磁头从柱面2移到柱面4999，再从柱面4999反方向移回柱面1，总共寻道距离高达9995，而只不过此柱面2差了一个柱面而已）

由于驱动器现在正在为柱面143提供服务，而前一个服务请求在柱面125。因此对于本题来说，服务顺序是：143 → 913 → 948 → 1022 → 1470 → 1509 → 1750 → 1774 → 4999 → 130 → 86

总共寻道距离为：770 + 35 + 74 + 448 + 39 + 241 + 24 + 3225 + 4869 + 44
= 9769

第四是LOOK算法，它从是SCAN算法的一种改进。可以看到SCAN算法的其中一个劣势在于它需要移到顶点再移回。但问题是如果最末请求很小（例如柱面2），移到柱面4999再移回来势必浪费大量时间。一种解决办法是移到磁头2以后发现移动方向上再没有请求就马上移回来，而不需要再往前移到顶点。

由于驱动器正在为柱面143提供服务，而前一个服务请求在柱面125，最后一个服务请求在柱面1774，因此对于本题来说服务顺序是：143 → 913 → 948 → 1022 → 1470 → 1509 → 1750 → 1774 → 130 → 86

总共寻道距离为：770 + 35 + 74 + 448 + 39 + 241 + 24 + 1644 + 44
= 3319

第五是C-SCAN算法，它从另一方面改进了SCAN算法。SCAN算法会出现序号上差别不大，但寻道时间很长的情况。针对这个问题，我们不需要将磁头反方向从柱面4999移回，而是先统一把它拉回到柱面0，再往下移动。就好像一台打印机打完一行的字以后从下一行的开头开始打印。这样使得原本磁头移动反方向的请求能更快地被服务。

由于驱动器正在为柱面143提供服务，而前一个服务请求在柱面125。因此对于本题来说，服务顺序是：143 → 913 → 948 → 1022 → 1470 → 1509 → 1750 → 1774 → 4999 → 0 → 86 → 130

总共寻道距离为：770 + 35 + 74 + 448 + 39 → 241 + 24 + 3225 + 4999 + 86 + 44
= ~~9899~~ 9985

第六是C-LOOK算法，它为LOOK算法和C-SCAN算法的结合版，它既可以保证在移动方向上没有请求就马上移回，而且保证移回时不用从0开始，而是从最小请求开始。

由于磁盘驱动器正在为柱面143提供服务，而前一个服务请求在柱面125，最后一个服务请求在柱面1774，最前一个服务请求在柱面86，因此对于本题来说服务顺序是：

143 → 913 → 948 → 1022 → 1470 → 1509 → 1750 → 1774 → 86 → 130

总共寻道距离为：770 + 35 + 74 + 448 + 39 + 241 + 24 + 1688 + 44

= 3363

A+