**作业1 操作系统概述（10分）**

**1.1（10分）通常我们说操作系统包括4大功能：进程管理、内存管理、文件管理与I/O设备管理。请回答下列问题。**

**（1）进程管理、内存管理、文件管理与I/O设备管理分别完成什么功能？在设计操作系统时，这4大功能分别要考虑哪些主要的非功能需求？**

一、进程管理

功能：

1. 进程的控制：系统使用一些具有特定功能的程序段来创建、撤销进程以及完成进程各个状态之间的转换从而达到多个进程高效率并发执行和资源共享。
2. 进程的调度：按照一定的策略，动态地把CPU分配给处于就绪队列中的某个进程，以使之执行，使得周转时间、CPU利用率、吞吐量的满足要求。
3. 解决在多进程并发执行时带来的资源共享与资源竞争问题，如进程同步问题、进程互斥问题、死锁问题。

非功能需求：

1、高性能：进行进程间的调度时，综合考量周转时间、吞吐量、CPU利用率等性能指标，选取合适的进程调度算法。

2、可靠性：有效解决并发进程的同步与互斥问题，有效预防、检测、恢复出现的死锁问题。

3、安全性：设置特权指令防止进程越权与越界。

4、进程控制块PCB集中在一起，存放在内存的固定位置构成PCB集，便于操作系统对进程进行管理。

二、内存管理

功能：

1. 内存地址变换：将逻辑地址通过某种映射关系转换为真正内存空间的物理地址。
2. 内存空间的分配与回收：由操作系统完成主存储器空间的分配和管理，并在进程执行结束后或者资源使用完后，释放其所占用的内存空间。
3. 内外存数据流动控制：控制内存与外存的程序和数据流动。

4、内存空间的扩充：将大型程序分段不一次性调入内存可以采用覆盖技术将程序分为多个段，常用的段常驻内存，不常用的段在需要时调入内存；也可以采用虚拟技术采用页的方式。

5、内存信息的共享与保护：通过设置上下界限寄存器，控制程序可访问的地址空间的界限，防止当前程序访问到不属于自己的地址空间，起到保证各道作业在各自的存储空间内运行，互不干扰的作用。

非功能需求：

1、高性能：高性能页面置换算法和虚拟内存管理的效率；增加缓冲机制引入TLB相关策略，解决频繁访存寻页的时间消耗，充分利用空间局限性，以提高性能。内存与磁盘的调度机制，设计调度算法。

2、安全性：通过设置上下界限寄存器，控制程序可访问的地址空间的界限，防止当前程序访问到不属于自己的地址空间，起到保证各道作业在各自的存储空间内运行，互不干扰的作用。

三、文件管理

功能：

1、文件存储空间管理：对外部存储器空间进行统一管理。在用户创建文件时为其分配空闲区，在用户删除文件时回收内存。

2、文件存取：文件系统具有对用户可见的逻辑结构，用户指定路径与操作，便可完成对信息的存取和加工。

3、文件访问控制与文件保护：采用针对不同的用户规定不同的权限，防止文件被没有授权的用户随意访问。

非功能需求：

1、高性能：利用局部性原理分配文件和目录，让相关数据分配在同一个区块内，不相关的数据分配在不同区块内；高性能文件存取算法。

2、可靠性：保证崩溃时数据的一致性，可以通过文件系统检查程序或者日志进行。

3、安全性：对于不同的用户进行权限划分，防止未授权的用户访问文件或对文件进行修改。

四、I/O设备管理

功能：

1、提高CPU与I/O设备的利用率：如利用缓冲区，支持并行操作，解决CPU与I/O设备速度不匹配的问题，以提高CPU与I/O设备的使用效率。

2、控制I/O设备与CPU之间的数据交换：根据I/O设备的传输速率、传输数据单位的不同，先择合适的I/O控制方式，如中断、DMA，以提高系统效率。

3、隐藏物理设备细节，为用户提供良好的透明接口。

4、设备的调度：为进程分配要使用的I/O设备资源，如果有多个I/O设备要被使用需要阻塞等待。

5、设备分配：当设备资源出现共享竞争时，按照分配策略为申请设备的进程分配设备。分配还应包括对设备的控制器、通道等支持部件的分配，以保证分配的完整性和合理性。

非功能需求：

1、高性能：如利用缓冲区，支持并行操作，解决CPU与I/O设备速度不匹配的问题，以提高CPU与I/O设备的使用效率。

2、提供用户使用的OS接口：隐藏物理设备细节，为用户提供良好的透明接口。

3、安全性：判断用户权限，防止用户越权操作设备。

4、拓展性：为用户提供透明的、易于拓展的接口，以使用户不必了解具体设备的物理特性并且便于设备的追加和更新。

**（2）如果要支持多任务并发执行，这4大功能在实现时分别要解决哪些重要问题（针对每个功能，至少给出1个问题并加以解释）？针对相应的问题分别给出解决方案。**

一、进程管理

1、进程调度问题，当前进程何时放弃CPU资源交给其他进程。解决方法：引入外部中断，采用分时的方法为进程分配时间片，当利用完时间片中断以切换进程。

2、互斥访问问题，如何原子地访问临界资源，而不被其他进程打扰或者防止有多个进程同时访问临界资源而引发问题。解决方案：引入锁的概念通过硬件辅助实现对锁原子地测试并赋值，从而实现当一个进程占用锁，其他进程无法进入该临界区只能自旋。

3、同步问题，解决由多任务并发执行引发的异步性导致的进程无法按序执行的问题。解决方案：引入信号量机制，设置P，V操作，将信号量初值设为0先要执行的进程结束后进行V操作，后进程开始前执行P操作。

二、内存管理

1、内存安全访问问题：防止进程之间互相访问彼此的地址空间，甚至访问操作系统的地址空间，从而导致错误。解决方法：引入界限寄存器，记录上下界，从而限制其访问。

2、内存虚拟化内存空间有限的问题，内存空间相较于进程所需的空间往往较小，需要设计一些调度策略将进程所需空间从外存调入内存中。解决方案：设计相应的调度算法如LRU，FIFO等。

三、文件管理

1、保证进程访问文件时，仅能有一个进程在写，不能有多个进程在写，而且写结束前不能读。解决方法：通过引入信号量机制实现。

2、文件的保护，对于不同的进程应该有自己对应的访问文件的权限。解决方案：引入加密保护的机制。

四、I/O设备管理

1、保证进程访问I/O设备管理的互斥性。解决方案：引入互斥锁的机制。

2、I/O调度，用某种算法确定一个执行I/O操作的顺序，以处理各个I/O请求。解决方法：采用LRU等算法

3、安全性设备保护：判断用户是否有权限访问当前设备。解决方法：设置标志位表明权限。

**作业2.1 进程管理（30分）**

**2.1.1（10分）设有两个进程P1和P2，其执行代码分别如下：**

**进程P1 进程P2**

**A; C;**

**B; D;**

**E;**

**其中，A、B、C、D、E都是原子操作。请给出进程P1和P2并发执行时所有可能的执行轨迹。**

当P1先时，有：ACBDE、ACDBE、ACDEB、ABCDE

当P2先时，有：CADBE、CADEB、CABDE、CDAEB、CDABE、CDEAB

所以综上所述，有：

ABCDE ACBDE ACDBE ACDEB CABDE CADBE CADEB CDABE CDAEB CDEAB

**2.1.2（10分）假定CPU按照程序中规定的顺序执行指令（即指令不会乱序执行），以2个进程的互斥为例，分析Peterson解决方案的正确性。**

（1）首先写出Peterson解决方案的简单代码形式，当一个进程想进入临界区时，先调用enter\_region函数，判断是否能安全进入，不能的话等待；当进程从临界区退出后，需调用leave\_region函数，允许其它进程进入临界区。

#define False 0

#define True 1

#define N 2 //进程的个数

int turn; //轮到谁

int interested[N]; //兴趣数组，初始化为False

void enter\_region(int process)//process为进程号 0或者1

{

int other=1-process;//另外一个进程

interested[process]=True;//想要进入临界区

turn=process;//设置标志位

while(turn == process && interested[other] == True){};

}

void leave\_region(int process)

{

interested[process]=False;//进程离开临界区

}

（2）下面针对可能出现的三种情况进行分析

a、一个进程想要进入临界区

假设0想要进入临界区，进入临界区之前调用enter\_region函数，其中对于语句while(turn == process && interested[other] == True){};不会进入无限循环，因此能够成功返回函数的调用点，进而进入临界区。

b、一个进程在临界区中时另外一个进程想要进入临界区

假设0想要进入临界区，此时1已经在临界区中，进入临界区前调用enter\_region函数，由于1已经在临界区中，所以interested[1]=True，因此0进入无限循环，无法成功进入临界区。

c、两个进程同时想要进入临界区

当两个进程同时想要进入临界区时，会调用enter\_region函数，无论顺序如何，不可能同时达成turn!=process||interested[other]!=True的条件，因此两者中有一个会进入循环，另外一个进程进入临界区。

以上这三种情况都能够满足：不允许多个进程同时访问临界区。从而说明了Peterson解决方案的正确性。

**2.1.3（10分）若实现一个OS的进程管理功能，支持多进程并发执行，需要建立哪些关于进程的数据结构？其中需要记录哪些基本信息？这些信息分别起什么作用？**

创建一个进程管理块的数据结构，用以保存记录有关进程描述信息、控制信息、资源管理信息，以保证进程间上下文切换调度时，进程间的恢复与切换。

另外还需要一个数据结构可以将每个进程的进程管理块组织在一起，构成PCB表。

记录的信息：

（1）进程描述信息

a、进程标识号PID。作用：唯一识别一个进程。

b、用户标识号UID，作用：唯一标识一个用户，便于资源的共享和保护。

c、家族关系。作用：描述进程之间的关系。

d、对应程序的地址。作用：方便进程调用相应的程序。

（2）进程控制信息

a、进程当前状态、进程优先级。作用：便于选取进程占用CPU。

b、各种计时信息。作用：给出进程占有和利用资源的有关情况。

c、通信信息。作用：说明该进程在执行过程中与其他进程之间的信息交换情况。

（3）进程资源管理信息

a、CPU现场保护信息。作用：便于进程回到被中止处继续执行。

b、进程间同步和通信机制，如信号量、消息队列指针。作用：保存进程间同步和通信机制，维护进程间的同步关系，保护临界资源的占有。

c、打开文件的信息，如文件描述符表。作用：记录当前该进程下各文件的状态以便操作系统再次执行该进程时对文件的恢复

作业2.2 进程同步（60分）

**2.2.1（16分）共享缓冲区问题。三个进程P1、P2、P3互斥使用一个包含N(N>0)个单元的缓冲区。P1每次用produce()生成一个正整数并用put()送入缓冲区某一空单元中；P2每次用getodd()从该缓冲区中取出一个奇数并用countodd()统计奇数个数；P3每次用geteven()从该缓冲区中取出一个偶数并用 counteven()统计偶数个数。试用信号量的P、V操作设计该问题的同步算法，给出所用共享变量（如果需要）和信号量及其初始值，并说明各自的含义。**

（1）简单的代码实现如下：

semaphore mutex=1;//进程互斥访问临界区

semaphore odd=0;//奇数的个数

semaphore even=0;//偶数的个数

semaphore empty=N;//缓冲区待填单元个数

void P(sem)//P操作

{

sem--；

if (sem < 0) {

Add this thread t to q；

block（t）；

｝

}

void V(sem)//V操作

{

sem++；

if (sem < =0) {

Remove a thread t from q；

wakeup（t）；

}

void P1()

{

while(1){

P(empty);

int temp=produce();//生成正整数

P(mutex);//进入临界区

put();//送入正整数

V(mutex);//离开临界区

//对odd和even进行处理

if(temp%2==0)

{

V(even);

}

else

{

V(odd);

}

}

}

void P2()

{

while(1){

P(odd);

P(mutex);

getodd();

countodd();

V(mutex);

V(empty);

}

}

void P3()

{

while(1){

P(even);

P(mutex);

geteven();

counteven();

V(mutex);

V(empty);

}

}

（2）相关说明如下：

a、设计四个信号量mutex=1用于实现对缓冲区的互斥访问，odd=0用于表示缓冲区odd的数量，even=0用于表示缓冲区even的数量,empty=N用于指明空闲缓冲区数量。

b、对于P1进程，首先等待缓冲区不空，如果不空则可以生产数据，然后尝试获取进入缓冲区的锁变量，进入临界区后，将产生数据放入缓冲区。退出临界区后根据产生数据的奇偶，唤醒对应的奇偶进程。

c、对于P2奇进程首先等待odd不为空，进入缓冲区，获取一个奇数，然后计算奇数的数目，最后离开临界区，使空闲空间+1。

d、对于P3偶进程首先等待even不为空，进入缓冲区，获取一个偶数，然后计算偶数的数目，最后离开临界区，使空闲空间+1。

**2.2.2（14分）吸烟者问题。在一个房间内有三个吸烟者和一个香烟供应者。为了制造并抽掉香烟，每个吸烟者需要三样物品：烟草、纸和火柴，供应者有丰富物品提供。在三个吸烟者中，第一个有自己的烟草，第二个有自己的纸，第三个有自己的火柴。供应者随机地将两样不同的物品放在桌子上，允许一个吸烟者制造香烟并进行不利于**健康**的吸烟。当吸烟者完成吸烟后唤醒供应者，供应者再随机地把两样不同的物品放在桌子上，唤醒一个吸烟者。试用信号量的P、V操作设计该问题的同步算法，给出所用共享变量（如果需要）和信号量及其初始值，并说明各自的含义。**

（1）简单的代码实现如下：

semaphore smoker[3]={0,0,0};//吸烟者是否具有吸烟条件

semaphore finish=0;//用户吸烟者与提供者的同步信号量

int i, j; //表示供应者放的两种资源，取值0、1、2分别代表烟草、纸和火柴

void P(semaphore);

void V(semaphore);

void supplier() //供应者

{

while (1){//随机生成两种提供的物品

        i = j = 0;

        while (i == j) {

            i = rand() % 3;

            j = rand() % 3;

        }

   P(finish);

   put\_items(i，j); //提供物品

if ((i==0 && j==1) || (i==1 && j==0)) //表示供应者提供的是烟草和纸

          V(smoker[2])  ; //第三个吸烟者具备了吸烟的条件

        else if ((i==0 && j==2) || (i==2 && j==0))

          V(smoker[1])  ;

        else  V(smoker[0]);

}

}

void smoker(int i) //吸烟者

{

    while (1) {

      P(smoker[i])  ;

        make\_and\_smoke(); //吸烟

      V(finish) ;

    }

}

（2）相关说明如下：

定义了四个信号量smoker[0]~smoker[2]、finish初始值置为0，共有四个进程分别为一个生产者和三个吸烟者。生产者随机提供材料并选择对应的吸烟者，所以执行对应的V(smoker[i])操作，当将准备好的东西放在桌子上后执行P(finish)等待某吸烟者吸完烟。对于某吸烟者进程来说，首先消耗生产者提供的材料，即执行P(smoker[i])操作，然后执行V(finish)操作，从而让生产者可以继续生产。

**2.2.3（12分）写出Reader-Writer问题的算法，避免由于不断有Reader出现而使得Writer无限期等待。**

int count=0;//读者数量

semaphore mutex\_count=1;//读者计数的互斥访问信号量

semaphore rw\_mutex=1;//访问文件互斥信号量

semaphore w=1;//写优先

void P(semaphore);

void V(semaphore);

void WRITE();//写操作

void READ();//读操作

void writer(){//写进程

    while(1){

        P(w);

        P(rw\_mutex);

        WRITE();

        V(rw\_mutex);

        V(w);

    }

}

void reader(){

    while(1){

        P(w);

        P(mutex\_count);

        if(count==0){

            P(rw\_mutex);

        }

        count++;

        V(mutex\_count);

        V(w);

        READ();

        P(mutex\_count);

        count--;

        if(count==0){

            V(rw\_mutex);

        }

        V(mutex\_count);

    }

}

**2.2.4（12分）面包师问题。面包师有很多面包和蛋糕，由N个销售人员销售。每个顾客进店后先取一个号，并且等着叫号。当一个销售人员空闲下来，就叫下一个号。试用信号量的P、V操作设计该问题的同步算法，给出所用共享变量（如果需要）和信号量及其初始值，并说明各自的含义。**

1. 简单的代码实现如下：

semaphore seller=N;//销售人员的数量

semaphore customer\_number\_mutex=1;//customer\_number的互斥信号量

semaphore seller\_number\_mutex=1;//seller\_number的互斥信号量

int seller\_number=0;//叫号

int customer\_number=0;//取号

semaphore customer=0;//顾客的数量

void P(semaphore);

void V(semaphore);

void Customer()

{

//进店

P(customer\_number\_mutex);

//取号

customer\_number++;

V(customer\_number\_mutex);

V(customer);

P(seller);

//买面包离开

}

void Seller()

{

while(1){

P(customer);

P(seller\_number\_mutex);

//叫号

seller\_number++;

V(seller\_number\_mutex);

V(seller);

}

}

1. 相关说明如下：
2. 定义了四个信号量seller、customer、seller\_number\_mutex、customer\_number\_mutex和变量customer、seller。变量customer用于记录顾客取号数，变量seller用于记录叫号数，信号量seller\_number\_mutex、customer\_number\_mutex分别作为seller、customer的锁变量信号量customer表示顾客人数，信号量seller表示销售员人数初始为n。
3. 对于顾客进程进店后，先取号互斥地执行customer\_number++，然后执行V(customer)使顾客数+1，然后等待seller空闲P(seller)，空闲后执行买面包离店。

c、对于销售员进程，首先等待顾客P(customer)，顾客来后互斥地执行seller\_number叫号，然后执行卖面包，卖完后V(seller)空闲销售员+1

作业3 进程调度（30分）

**3.1（12分）设有5个进程A、B、C、D、E（没有I/O），就绪时刻和运行时间如下：**

**进程 就绪时刻 运行时间**

**A 0 3**

**B 2 6**

**C 4 4**

**D 6 5**

**E 8 2**

**分别采用下列调度算法，给出调度顺序，计算平均周转时间和平均带权周转时间（忽略进程切换开销）。**

**1）先来先服务**

**2）时间片轮转（时间片大小=1）**

**3）时间片轮转（时间片大小=4）**

**4）短进程优先**

**说明：若进程P1执行时间片到的时刻与进程P2的就绪时刻相等，则认为进程P2先处于就绪状态。**

1. 先来先服务

调度顺序：ABCDE

平均周转时间：(3+7+9+12+12)/5=8.60

平均带权周转时间：(3/3+7/6+9/4+12/5+12/2)/5=2.56

二、时间片轮转（时间片大小为1）

调度顺序：AABABCBDCBEDCBEDCBDD

平均周转时间：(4+16+13+14+7)/5=10.80

平均带权周转时间：(4/3+16/6+13/4+14/5+7/2)/5=2.71

三、时间片轮转（时间片大小为4）

调度顺序：ABCDBED

平均周转时间：(3+15+7+14+11)/5=10.00

平均带权周转时间：(3/3+15/6+7/4+14/5+11/2)/5=2.71

四、短进程优先

调度顺序：ABECD

平均周转时间：(3+7+11+14+3)/5= 7.60

平均带权周转时间：(3/3+7/6+11/4+14/5+3/2)/5=1.84

**3.2（12分）你认为应该用哪些指标来评价进程调度算法？请给出你认为最重要的4个指标，并采用你给出的这些指标，对先来先服务、时间片轮转、短进程优先、静态优先级、动态优先级、多级反馈队列等调度算法进行评价。**

四个指标：

a、周转时间（平均周转时间或平均带权周转时间）

b、CPU利用率

c、吞吐量

d、响应时间

评价先来先服务：

1. 周转时间：周转时间波动较大，取决于队列中的进程。

b、CPU利用率：当CPU 密集型进程得到 CPU，并使用它。在这段时间内，所有其他进程会处理完它们的 I/O，并转移到就绪队列来等待 CPU。当这些进程在就绪队列中等待时，I/O 设备空闲。最终，CPU 密集型进程完成 CPU 执行并且移到 I/O 设备。所有 I/O 密集型进程，由于只有很短的 CPU 执行，故很快执行完并移回到 I/O 队列。这时，CPU 空闲。之后，CPU 密集型进程会移回到就绪队列并分配到 CPU。再次，所有 I/O 进程会在就绪队列中等待 CPU 密集型进程的完成。所有其他进程都等待一个大进程释放 CPU，会导致 CPU 和设备的使用率降低。

c、吞吐量：一般，适用于非交互式系统。

d、响应时间：未必长也未必短，方差大，实际响应时间取决于等待队列中的进程。

评价时间片轮转法：

a、周转时间：适中，取决于时间片的大小，在极端情况下，如果时间片很大，那么就等同于先来先服务方法。

b、CPU利用率：CPU利用率较高，但不是为进程服务，而是由于频繁的进程上下文切换导致的CPU资源消耗。

c、吞吐量：较高，适用于多任务环境。

d、响应时间：通常短，因为进程能够周期性地获得CPU。

评价短进程优先：

a、周转时间：短进程友好，等待时间短；长进程不友好，等待时间长，极端情况下可能会导致长进程一直得不到CPU。

b、CPU利用率：倾向于选择执行时间较短的进程，从而更快地完成任务，减少了CPU的空闲时间。

c、吞吐量：高，适用于短进程密集型的环境。

d、响应时间：可能较长，因为长进程可能要等待长时间，直到队列前的短进程执行完毕。

评价静态优先级：

a、周转时间：周转时间与进程的优先级分配情况有关。

b、CPU利用率：取决于进程的静态优先级分配策略和各进程的相对优先级设置。

c、吞吐量：吞吐量与进程的优先级分配情况有关。

d、响应时间：高优先级进程响应时间短，低优先级进程响应时间长。

评价动态优先级：

a、周转时间：取决于动态调整，可高可低。

b、CPU利用率：取决于进程的优先级分配策略和各进程的相对优先级设置。

c、吞吐量：可以是高或低，取决于动态调整。

d、响应时间：可以是高或低，取决于动态调整。

评价多级反馈队列：

a、周转时间：性能通常较好，较高优先级的队列通常会分配给短期任务，使它们能够快速完成。

b、CPU利用率：可以实现相对高的CPU利用率，因为它能够灵活地为不同类型的任务分配CPU时间片。

c、吞吐量：通常能够实现良好的吞吐量，特别是在任务混合多样的情况下。

d、响应时间：可以提供相对低的响应时间，因为高优先级队列中的任务能够快速响应。

**3.3（6分）设计实现一个模拟多级反馈队列的多任务调度程序。给出实现方法、完整源代码和测试验证。**

实现方法

1. 创建队列：创建多个队列，每个队列对应一个不同的优先级。通常会有三个队列，但实际情况可以有更多。

2. 初始化任务：初始化任务，包括任务的名字、优先级、和估计的执行时间。

3. 任务入队：将任务添加到初始的队列中，通常是根据任务的初始优先级。

4. 调度：从队列中选择任务来执行，通常选择最高优先级的任务执行。任务执行一段时间后，如果没有完成，将其放回到队列的尾部，以便后续调度。

5. 降低优先级：如果任务在当前队列执行了一段时间仍未完成，其优先级可能会降低，将其移到较低优先级的队列中等待执行。

6. 任务完成检查：检查任务是否已经执行完成，如果完成则将其从队列中移除。

7. 循环调度：重复执行上述步骤，直到所有任务都完成。

#include <stdio.h>

#include <stdlib.h>

// 定义任务结构

typedef struct Task {

char name;

int priority;

int burst\_time;

} Task;

// 定义任务队列

typedef struct Queue {

Task\* tasks;

int front;

int rear;

int size;

} Queue;

// 初始化队列

Queue\* initQueue(int size) {

Queue\* queue = (Queue\*)malloc(sizeof(Queue));

queue->tasks = (Task\*)malloc(size \* sizeof(Task));

queue->front = 0;

queue->rear = -1;

queue->size = size;

return queue;

}

// 入队

void enqueue(Queue\* queue, Task task) {

if (queue->rear == queue->size - 1) {

printf("Queue is full\n");

} else {

queue->tasks[++queue->rear] = task;

}

}

// 出队

Task dequeue(Queue\* queue) {

if (queue->front > queue->rear) {

printf("Queue is empty\n");

Task null\_task = {'\0', -1, -1};

return null\_task;

}

return queue->tasks[queue->front++];

}

// 调度函数

void schedule(Queue\* queues[], int num\_queues, int time\_slice) {

int current\_time = 0;

int current\_queue = 0;

int quantum = time\_slice;

while (1) {

if (queues[current\_queue]->front <= queues[current\_queue]->rear) {

Task current\_task = dequeue(queues[current\_queue]);

printf("Executing task %c from queue %d for time slice %d\n", current\_task.name, current\_queue, quantum);

current\_time += quantum;

current\_task.burst\_time -= quantum;

if (current\_task.burst\_time > 0) {

if (current\_queue < num\_queues - 1) {

current\_queue++;

}

enqueue(queues[current\_queue], current\_task);

} else {

printf("Task %c completed\n", current\_task.name);

}

}

if (queues[current\_queue]->front > queues[current\_queue]->rear) {

if (current\_queue < num\_queues - 1) {

current\_queue++;

}

}

if (current\_queue == 0 && queues[0]->front > queues[0]->rear)

break;

}

}

int main() {

int num\_queues = 3;

Queue\* queues[num\_queues];

for (int i = 0; i < num\_queues; i++) {

queues[i] = initQueue(10);

}

Task tasks[] = {

{'A', 0, 5},

{'B', 1, 10},

{'C', 2, 3},

{'D', 0, 7},

{'E', 1, 2},

};

for (int i = 0; i < sizeof(tasks) / sizeof(Task); i++) {

enqueue(queues[0], tasks[i]);

}

int time\_slice = 2;

schedule(queues, num\_queues, time\_slice);

return 0;

}