**校内讲义**

**计算思维与操作系统课程设计**

**实验指导书**

**（2020版）**

**编写：李玲 张海蓉 孙大洋 曲良东**

**吉林大学通信工程学院**

目 录

实验一 图论模型设计与存储 1

实验二 图的遍历输出 2

实验三 图的最小生成树设计 3

实验四 图的单源最短路径求解 4

实验五 图的任意两点间最短路径求解 5

实验六 图论模型系统联调与测试 6

实验七 AOE网中的关键路径求解与优化（选作） 7

实验八 经典互斥算法实验 8

实验九 经典同步算法实验 10

实验十 线程池任务调度实验 12

实验十一 内存管理实验 14

附录 15

# 实验一 图论模型设计与存储

**一、实验目的**

掌握图的存储思想及其存储实现。

**二、实验内容**

1、设计一个实际问题，进行图论建模。

2、分别采邻接矩阵、邻接表存储方式存储图。

**三、实验原理**

1、图的邻接矩阵存储结构。

2、图的邻接表存储结构。

**四、参考学时**

4学时。

# 实验二 图的遍历输出

**一、实验目的**

掌握图的存储思想及其存储实现。

**二、实验内容**

1、掌握图的运算基础——图的遍历——的原理及其实现。

2、训练采用递归方法进行程序设计的能力。

**三、实验原理**

1、DFS算法。

2、BFS算法。

3、递归程序设计方法。

**四、参考学时**

4学时。

# 实验三 图的最小生成树设计

**一、实验目的**

1、掌握树、图这两种数据结构的运算，理解其实际应用。

2、理解最小生成树问题在解决实际问题中的意义。

3、掌握最小生成树算法原理及其程序实现方法。

**二、实验内容**

1、分别采用Prime算法和Kruskal算法求取图的最小生成树。

2、分别输出基于上述两种算法求出的最小生成树，并比较算法的时间复杂度和空间复杂度。

**三、实验原理**

1、Prime算法。

2、Kruskal算法。

**四、参考学时**

6学时。

# 实验四 图的单源最短路径求解

**一、实验目的**

1、理解图的单源最短路径问题在解决实际问题中的意义。

2、掌握图的单源最短路径求解算法原理及其程序实现方法。

3、训练综合运用多种数据结构灵活解决实际问题的能力。

**二、实验内容**

1、编程随机选取图中的任一顶点作为源点。

2、基于Dijkstra算法求取源点到图中其他所有顶点的最短路径及最短路径长度。

3、使用堆栈方法输出从源点到每个终点的最短路径及最短路径长度。

**三、实验原理**

1、Dijkstra算法。

2、栈的存储结构与运算。

**四、参考学时**

6学时。

# 实验五 图的任意两点间最短路径求解

**一、实验目的**

1、理解图的任意两点间最短路径问题在解决实际问题中的意义。

2、掌握图的任意两点间最短路径求解算法原理及其程序实现方法。

3、训练采用递归方法进行程序设计的能力。

4、训练综合运用多种数据结构灵活解决实际问题的能力。

**二、实验内容**

1、基于Floyd算法求取图的任意两点间的最短路径及最短路径长度。

2、采用递归方法输出任意两点间的最短路径及最短路径长度。

**三、实验原理**

1、Floyd算法。

2、递归程序设计方法。

**四、参考学时**

6学时。

# 实验六 图论模型系统联调与测试

**一、实验目的**

1、掌握文本菜单或图形菜单的设计方法。

2、掌握综合应用程序的系统联调和测试方法。

**二、实验内容**

1、设计图论模型系统的菜单，鼓励设计图形菜单。

2、对完整的图论模型系统进行联调与测试。

**三、实验原理**

1、文本菜单/图形菜单的设计方法。

2、系统联调与测试的一般方法。

**四、参考学时**

4学时。

# 实验七 AOE网中的关键路径求解与优化（选作）

**一、实验目的**

1、掌握对实际问题进行计算机建模的方法。

2、以图论模型为例，训练综合运用多种数据结构、多种程序设计方法解决实际问题的能力。

3、初步训练独立从事科研的能力。

**二、实验内容**

AOE网中的关键路径求解与优化。AOE网的关键路径问题是图论的一个经典问题，鼓励自行设计或搜索实际问题并对之进行图论建模，可以采用基本方法求解关键路径，鼓励查阅文献学习和使用扩展算法或优化算法求解，以训练初步的独立科研能力。

**三、实验原理**

1、图的存储。

2、图的遍历算法。

3、Floyd算法（鼓励查阅文献学习和使用扩展/优化算法）。

4、递归程序设计方法。

**四、参考学时**

10学时。

# 实验八 经典互斥算法实验

**一、实验目的**

掌握操作系统多线程开发信号灯互斥使用方法。

**二、实验内容**

1、设计并实现Linux下的多线程共享资源实验。

2、设计并实现哲学家就餐实验，了解如何利用信号量来对共享资源进行互斥访问。

**三、实验原理**

1、哲学家就餐问题描述

5个哲学家（A、B、C、D、E）共用一张圆桌，圆桌周围有五把椅子，每位哲学家坐在其中一把椅子上，桌子的中央是一碗米饭，相邻哲学家之间的桌子上放置1只筷子，一共5只筷子，如图1所示。每位哲学家穷其一生都在思考（实验中用线程休眠表示）或就餐（实验中用文字表示，A、B、C、D、E对应5位同学的姓名学号，就餐时输出该同学正在就餐）。哲学家思考时，她不与其他哲学家交流。哲学家就餐时，必须申请并获得一左一右两支筷子，就餐完毕后释放这两支筷子，然后开始思考。

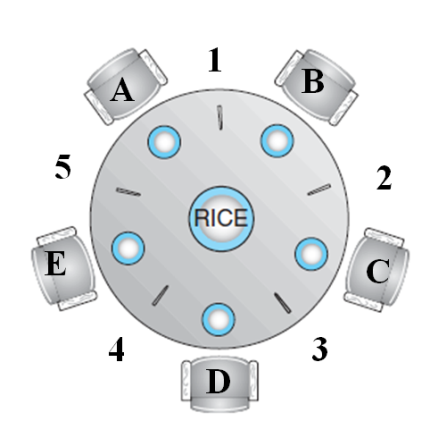


图1 哲学家就餐问题示意图

2、使用操作系统信号量实现线程互斥

一个简单的解决方案是用信号量表示每根筷子，哲学家通过对信号量执行wait()操作(信号灯接口：sem\_wait(…))来拿起一根筷子，通过对信号量执行signal()操作(信号灯接口：sem\_post(…))放下筷子。该方案可以保证没有两个相邻哲学家同时就餐，但会引起死锁。

3、哲学家就餐问题死锁的可选解决方案

（1）最多允许四位哲学家同时坐在桌边。

（2）只有左右两根筷子同时可用的时候才允许哲学家拿起左右的筷子(在临界区中拿起两只筷子)。

（3）使用不对称的方法——奇数的哲学家先拿起她的左边的筷子，然后拿起她的右边的筷子，而偶数的哲学家先拿起她的右边的筷子，然后拿起她的左边的筷子。

**四、参考学时**

8学时。

# 实验九 经典同步算法实验

**一、实验目的**

掌握操作系统同步技术使用方法。

**二、实验内容**

1、设计并实现操作系统读者写者问题读者优先算法。

2、设计并实现操作系统读者写者问题写者优先算法。

1. **实验原理**
   1. 读者写者问题描述

假设要在几个并发进程之间共享一个数据文件，其中一些进程只读取数据文件，而其他进程会更新(读取和写入)数据文件。我们通过把前者称为读者（readers），把后者称为写者（writers）来区分这两种类型的进程。如果两个读者同时访问共享的数据，不会产生任何负面影响。但是，如果写者和其他进程(读者或写者)同时访问数据文件，可能会导致混乱。因此读者写者问题需要使用操作系统同步技术进行行为约束。为了确保不会出现这些读写混乱的问题，我们要求写者在写入数据文件时对共享数据文件具有独占访问权。这种同步问题称为读者写者问题。实验中，设计3个读者，3个写者，分别对应6名同学的姓名学号，每个线程随机一段时间发起读写请求，读写操作用输出文字方式表示，且要持续一段时间。

2、第一类读者写者问题

第一类读者写者问题是读者写者问题最简单的一个变种，它要求除非写者已获得使用共享对象的权限，否则不会让读者一直等待。换句话说，读者不应该仅仅因为写者在等待而等待其他读者。它是读优先的。

第一类读者写者问题的新读者进程的同步方式：

1）如无读者、写者，新读者可以读；

2）如有写者等，但有其它读者正在读，新读者也可以读；

3）如有写者写，则新读者等待。

第一类读者写者问题的新写者进程的同步方式：

1）如无读者，新写者可以写；

2）如有读者，新写者等待；

3）如有其它写者，新写者等待。

3、第二类读者写者问题

第二类读者写者问题要求，一旦一个写者准备就绪，这个写者就必须尽快执行它的写操作。换句话说，如果写者正在等待访问对象，则没有新的读者可以开始阅读。

第二类读者写者问题的新读者进程的同步方式：

1）如无读者且无写者，新读者可以读；

2）如有读者但无写者等待，新读者可以读；

3）如有读者但有写着等待，新读者等待；

4）如有写者在写，则新读者等待。

第二类读者写者问题的新写者进程的同步方式：

1）如无读者且无写者，新写者可以写；

2）如有读者在读，新写者等待；

3）如有写者写，新写者等待。

**四、参考学时**

8学时。

# 实验十 线程池任务调度实验

**一、实验目的**

掌握操作系统线程池管理技术。

**二、实验内容**

1、设计并实现Linux下的线程池管理框架。

2、使用时间片轮转算法或面包店算法实现线程池管理的线程调度机制。

**三、实验原理**

假设服务器的硬件资源“充裕”，那么提高服务器性能的一个很直接的方法就是空间换时间，即“浪费”服务器的硬件资源，以换取其运行效率。提升服务器性能的一个重要方法就是采用“池”的思路，即对一组资源在服务器启动之初就被完全创建好并初始化，这称为静态资源分配。当服务器进入正式运行阶段，即开始处理客户端请求时，如果它需要相关资源就可以直接从池中获取，无需动态分配。很显然，直接从池中取得所需要资源比动态分配资源的速度快得多，因为分配系统资源的系统调用都是很耗时的。当服务器处理完一个客户端连接后，可以把相关资源放回池中，无须执行系统调用释放资源。从最终效果来看，资源分配和回收的系统调用只发生在服务器的启动和结束，这种“池”的方式避免了中间的任务处理过程对内核的频繁访问，提高了服务器的性能。具体过程如下：

1、主程序创建拥有10个线程的线程池，线程池中的线程在创建之后全部挂起等待调度。

提示：采用链表来保存线程池的信息，如线程编号（自行定义），线程优先级（自行定义），线程运行时间（自行定义）等。

2、主程序创建线程池之后启动调度线程，调度线程采用时间片轮转算法或面包店算法对线程进行调度（算法原理等参考附录），每次最多调用3个线程运行，注意由于算法不同，调用的线程每次调度应当不同。

3、运行的线程输出线程编号+学号，并输出采用的调度算法名称，线程运行结束后应当归还到线程池，并重新挂起等待再次调用。

4、调度算法完成至少10次调度，即至少运行完成30个线程并输出相关内容。

注意两种算法选择1个实现即可。

**四、参考学时**

8学时。

# 实验十一 内存管理实验

**一、实验目的**

理解操作系统内存管理原理及技术。

**二、实验内容**

1、使用线程池技术设计并实现线程池共享父进程内存资源的内存管理机制。

2、在1的基础上实现共享内存资源的连续内存管理。

**三、实验原理**

1、仿照操作系统内存管理的形式，使用链表生成空闲内存空间，共8块，每块内存块大小10字节。

2、线程池调度2个线程运行，分别申请内存区块保存学号（8位学号）和姓名的拼音形式（全名，字符可能超过10个，自行思考如何安排）。

3、采用FIFO和LIFO算法将存储的学号和姓名输出，同时输出线程编号和被调度的次数。

4、每次调度输出结束后释放内存归还空闲链表中。

5、连续进行10次调度，如果内存不够，线程需要挂起等待。

注意：分别按照首次适应算法FF，最佳适应算法BF和最差适应算法WF进行内存分配。

**四、参考学时**

6学时。

# 附录

面包店算法：

Lamport面包店算法可以有效地用于多个相互竞争的控制线程，该算法中线程之间的通信只能在共享内存中进行。

这个思想来自于面包店，医院等，需要排队取号的场所。顾客进入面包店前，首先抓取一个号码，然后按号码从小到大的次序依次进入面包店购买面包。

前提：

面包店按由小到大的次序发放号码

两个或两个以上的顾客有可能得到相同号码

当多个顾客抓到相同号码，则按顾客名字的字典次序排序

发号器按由小到大的次序发放号码。进程进入临界区前先抓取一个号码，然后按号码从小到大的次序依次进入临界区。若多个进程抓到相同的号码则按进程编号依次进入。

伪码例程：

process(i) {

while (true) {

// 当前进程i正在取号

choosing[i] = true;

// number为上一个已发放的排队号加1

number[i] = 1 + max(number[1], number[2], ..., number[n-1]);

// 当前进程i取号完毕

choosing[i] = false;

// 迭代所有进程

for (j = 0; j < n; j++)

{

// 若当前迭代到的进程j正在取号，则等待其取号完毕

while(choosing[j]);

// 同时满足，当前进程才能通过

while (number[j] != 0 && (number[j], j) < (number[i], i));

}

// 临界区代码

// 当前进程注销排队号

// 一旦线程在临界区执行完毕，需要把自己的排队签到号码置为0，表示处于非临界区

number[i] = 0;

// 其它代码

}

}

注意：

1) 进程需要排队等待的三种情况：

情况1：存在没有取得排队号的进程

情况2：当前迭代到的进程没有取得排队号

情况3：当前迭代到的进程的排队号小于当前进程的排队号，或当前迭代到的进程PID小于当前进程PID

2) 只有当前进程注销了排队号，在排队的其它进程才能进入临界区，满足进程互斥和有限等待

3) 符号说明：(a, b) < (c, d) 表示 (a < c) or ((a == c) and (b < d))

4) 使用choosing数组是必须的，假设不使用choosing数组，就可能会出现这种情况：设进程i的优先级高于进程j(即 i < j)，两个进程获得了相同的number，进程i在写number[i]之前，被优先级低的进程j抢先获得了CPU时间片，这时进程j读取到的number[i]为0，因此进程j进入了临界区。随后进程i又获得CPU时间片，它读取到的number[i]与number[j]相等，且i < j，因此进程i也进入了临界区。这样，两个进程同时在临界区内访问，可能会导致数据腐烂(data corruption)。算法使用了choosing数组变量，使得修改number数组的元素值变得"原子化"，解决了上述问题。

时间片轮转算法：

用于分时系统中的进程调度。每次调度时，总是选择就绪队列的队首进程，让其在CPU上运行一个系统预先设置好的时间片。一个时间片内没有完成运行的进程，返回到绪队列末尾重新排队，等待下一次调度。

给每个进程固定的执行时间，根据进程到达的先后顺序让进程在单位时间片内执行，执行完成后便调度下一个进程执行，时间片轮转调度不考虑进程等待时间和执行时间，属于抢占式调度。优点是兼顾长短作业；缺点是平均等待时间较长，上下文切换较费时。适用于分时系统。

参考例程：

typedef struct{

char name;//进程名字

int ArrivalTime;//到达时间

int ServiceTime;//服务时间

int FinishedTime;//完成时间

int WholeTime;//周转时间

double WeightWholeTime;//带权周转时间

}RR;

static queue<RR>RRqueue; //用来模拟进程执行RR调度算法的队列

static double SumWT=0,SumWWT=0,AverageWT =0,AverageWWT=0;//平均周转时间、平均带权周转时间

static int q; //时间片数

static int n; //进程个数

static RR RRarray[100]; //进程结构体

static RR temp; //进程结构

void Enter();//输入时间片、到达时间、服务时间等

void RRfunction();//执行RR调度算法

void display();//显示各时间执行情况，以及各个时间值

//执行RR调度算法

void RRfunction(){

char Moment[100]; //存储每个时间片p对应的进程名称

RRqueue.push(RRarray[0]); //第一个进程进队列

int MomentPoint = 0;

int CurrentTime=0;

int tempTime; //控制CurrentTime的累加时间，当前进程的服务时间小于时间片q的时候，起到重要作用

int i=1,j,k,p; //指向还未处理的进程的下标

int finalNumber = 0; //执行RR算法后，进程的个数

int Time[50];

bool canAdd=false;//用于解决CurrentTime在第一个进程完成计算时多加了一次服务时间的问题

//判断第一个进程的服务时间是否大于时间片，如果大于CurrentTime=q，如果小于CurrentTime=服务时间

if (RRarray[0].ServiceTime>=q)

CurrentTime = q;

else

CurrentTime = RRarray[0].ServiceTime;

while(!RRqueue.empty()){

for (j=i;j<n;j++){ //使得满足进程的到达时间小于当前时间的进程都进入队列

if (RRarray[j].name!=NULL && CurrentTime >= RRarray[j].ArrivalTime){

RRqueue.push(RRarray[j]);

i++;

}

}

if (RRqueue.front().ServiceTime<q)

tempTime = RRqueue.front().ServiceTime;

else

tempTime = q;

RRqueue.front().ServiceTime -= q; //进程每执行一次，就将其服务时间 -q

//将队首进程的名称放入数组中

Moment[MomentPoint] = RRqueue.front().name;

MomentPoint++;

Time[finalNumber] = tempTime;

finalNumber++;

if (RRqueue.front().ServiceTime <= 0) //把执行完的进程退出队列

RRqueue.pop(); //如果进程的服务时间小于等于，即该进程已经服务完了，将其退栈

else{

//将队首移到队尾

RRqueue.push(RRqueue.front());

RRqueue.pop();

}

if(canAdd){

CurrentTime += tempTime;

}

canAdd=true;

}

//进程输出处理 每个时间段对应的执行的进程

cout<<"各进程的执行时刻信息："<<endl;

Time[finalNumber]=0;

int time = Time[0];

int count = 0;

int countTime=0;

for (int i=0;i<finalNumber;i++)//计算完成时间

{

count = 0;

while(RRarray[count].name!=Moment[i] && count<n)

{

count++;

}

RRarray[count].FinishedTime = time;

if (i<finalNumber - 1)

{

time += Time[i+1];

}

}

for(int i=0;i<finalNumber;i++){//输出

for(int j=0;j<Time[i];j++){

cout<<setw(3)<<countTime<<" --"<<setw(3)<<countTime+1<<setw(6)<<"时刻 "<<"进程"<<Moment[i]<<"执行";

for(int k=0;k<n;k++){

if(RRarray[k].ArrivalTime==countTime){

cout<<" 进程"<<RRarray[k].name<<"到达";

}

else if(RRarray[k].FinishedTime==countTime+1&&countTime!=0){

cout<<" 进程"<<RRarray[k].name<<"完成";

}

}

countTime+=1;

cout<<endl;

}

}

cout<<endl;

//周转时间、带权周转时间、平均周转时间、带权平均周转时间的计算

for (i=0;i<n;i++){

RRarray[i].WholeTime = RRarray[i].FinishedTime - RRarray[i].ArrivalTime;

RRarray[i].WeightWholeTime = (double)RRarray[i].WholeTime/RRarray[i].ServiceTime;

}

for (i=0;i<n;i++){

SumWT += RRarray[i].WholeTime;

SumWWT += RRarray[i].WeightWholeTime;

}

AverageWT = SumWT/n;

AverageWWT = SumWWT/n;

}