中国地质大学（北京）

操作系统课程设计报告

学　　院：信息工程学院

专　　业：计算机科学与技术

班　　级：10041811

学　　号：1005183121

姓　　名：周子杰

联系方式：18006163783

邮　　箱：2476331552@qq.com

指导老师：王玉柱

日　　期：2020 年 12 月 20 日

摘　要

算法实验：

实验一、银行家算法

实验二、处理机管理

轮转法

高响应比调度算法

实验三、可变式分区管理

内存回收

循环首次适应算法

最佳适应算法

实验五、分页存储管理

LRU

FIFO

改进的 Clock 算法

实训项目

生产者消费者问题

读者优先的读者-写者问题

写者优先的读者-写者问题

哲学家就餐问题

关键词：资源、安全序列、时间片、响应比、分区、页面置换、信号量等

目　录

一、操作系统理论算法实验

实验一、银行家算法

实验题目

编制银行家算法通用程序，并检测所给状态的系统安全性。假定系统的任何一种资源在任一时刻只能被一个进程使用。任何进程已经占用的资源只能由进程自己释放，而不能由其它进程抢占。进程申请的资源不能满足时，必须等待。

设计要求

1. 程序中使用的数据结构及主要符号说明
2. 资源的种类和数目可以变化的
3. 进程可以任意的顺序创建和变化

需求分析

银行家算法是在操作系统中，避免死锁的一种方法。通过银行家算法，我们能对资源进行合理的分配，从而避免了资源的冲突进而发生死锁。

数据结构

```
//全局
//系统可用（剩余）资源
int available[resourceNum] = {3, 3, 2};
//进程的最大需求
int maxRequest[processNum][resourceNum] = {{7, 5, 3}, {3, 2, 2}, {9, 0, 2}, {2, 2, 2}, {4,
3, 3}};
//进程已经占有（分配）资源
int allocation[processNum][resourceNum] = {{0, 1, 0}, {2, 0, 0}, {3, 0, 2}, {2, 1, 1}, {0, 0,
```

```
2}};
//进程还需要资源
int need[processNum][resourceNum] = {{7, 4, 3}, {1, 2, 2}, {6, 0, 0}, {0, 1, 1}, {4, 3, 1}};
//是否安全
bool Finish[processNum];
//安全序列号
int safeSeries[processNum] = {0, 0, 0, 0, 0};
//进程请求资源量
int request[resourceNum];
//资源数量计数
int num;


//局部
//表示系统可提供给进程继续运行所需的各类资源数目
int work[resourceNum] = {0};
```

算法思想

银行家算法

首先，是一些输入限制，判断输入是否是数字、分配是否合理，如请求的不能大于需要的，也不能大于系统资源数量。

然后，就是安全性分析：

第一步：预分配

当然，在预分配正式开始之前，要进行系统数组的复制、设置 `finish` 数组全为 `false`。

接下来，我们就开始正式的预分配过程了。

我们通过不断的循环去寻找能够安全分配，也就是需要的资源小于等于系统剩余资源并且这个进程尚未完成，如果找到则记录让其 `finish` 状态记为 `true`，并且加入到安全队列中去。找不到的时候就可以跳出循环了。

第二步：判断

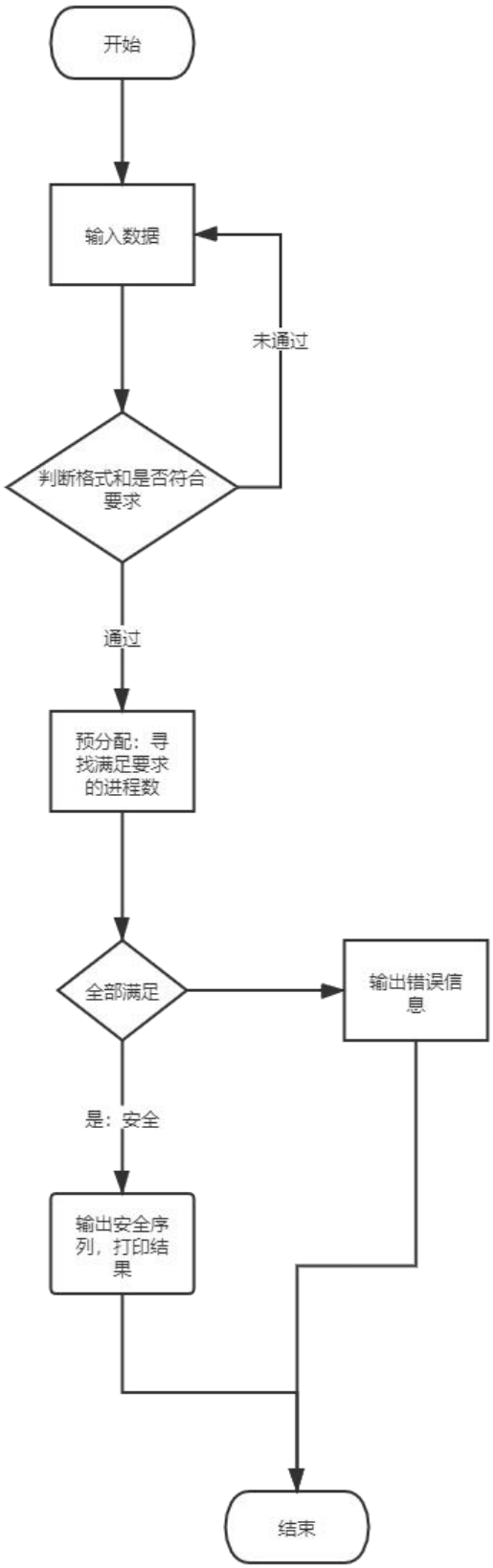
接着我们判断是不是所有的进程的 `finish` 状态都是 `true`，如果是就存在安全序列，否则则不存在。

第三步：输出

这里要判断有没有进程的需要的资源都为 0 的，如果有，那么这个进程就是已经完成的进程。相应地，我们要将输出队列减去这些进程的个数。

最后，就是判断，如果通过了安全检查，那么就可以继续分配，否则将已分配的资源换回来。

算法流程


开始
输入数据
未通过
判断格式和是否符合要求
通过
预分配：寻找满足要求的进程数
全部满足
输出错误信息
是：安全
输出安全序列，打印结果
结束

算法实现

```
#include <stdio.h>
#define resourceNum 3
#define processNum 5

//系统可用（剩余）资源
int available[resourceNum] = {3, 3, 2};
//进程的最大需求
int maxRequest[processNum][resourceNum] = {{7, 5, 3}, {3, 2, 2}, {9, 0, 2}, {2, 2, 2}, {4, 3, 3}};
//进程已经占有（分配）资源
int allocation[processNum][resourceNum] = {{0, 1, 0}, {2, 0, 0}, {3, 0, 2}, {2, 1, 1}, {0, 0, 2}};
//进程还需要资源
int need[processNum][resourceNum] = {{7, 4, 3}, {1, 2, 2}, {6, 0, 0}, {0, 1, 1}, {4, 3, 1}};
//是否安全
bool Finish[processNum];
//安全序列号
int safeSeries[processNum] = {0, 0, 0, 0, 0};
//进程请求资源量
int request[resourceNum];
//资源数量计数
int num;

//打印输出系统信息
void showInfo()
{
    printf("\n-----------------------------------------------------------------------------\n");
```

```
    printf("当前系统各类资源剩余： ");
    for (int j = 0; j < resourceNum; j++)
    {
        printf("%d ", available[j]);
    }
    printf("\n\n 当前系统资源情况： \n");
    printf(" PID\t Max\t\tAllocation\t Need\n");
    for (int i = 0; i < processNum; i++)
    {
        printf(" P%d\t", i);
        for (int j = 0; j < resourceNum; j++)
        {
            printf("%2d", maxRequest[i][j]);
        }
        printf("\t\t");
        for (int j = 0; j < resourceNum; j++)
        {
            printf("%2d", allocation[i][j]);
        }
        printf("\t\t");
        for (int j = 0; j < resourceNum; j++)
        {
            printf("%2d", need[i][j]);
        }
        printf("\n");
    }
}

//打印安全检查信息
```

```
void SafeInfo(int *work, int i)
{
    int j;
    printf(" P%d\t", i);
    for (j = 0; j < resourceNum; j++)
    {
        printf("%2d", work[j]);
    }
    printf("\t\t");
    for (j = 0; j < resourceNum; j++)
    {
        printf("%2d", allocation[i][j]);
    }
    printf("\t\t");
    for (j = 0; j < resourceNum; j++)
    {
        printf("%2d", need[i][j]);
    }
    printf("\t\t");
    for (j = 0; j < resourceNum; j++)
    {
        printf("%2d", allocation[i][j] + work[j]);
    }
    printf("\n");
}

//判断一个进程的资源是否全为零
bool isAllZero(int kang)
{
```

```
    num = 0;
    for (int i = 0; i < resourceNum; i++)
    {
        if (need[kang][i] != 0)
            return false;
    }
    return true;
}

//安全检查
bool isSafe()
{
    //int resourceNumFinish = 0;
    int safeIndex = 0;
    int allFinish = 0;              //完成的数量
    int work[resourceNum] = {0}; //表示系统可提供给进程继续运行所需的各类资源数目

    int pNum = 0;

    //预分配为了保护 available[]
    for (int i = 0; i < resourceNum; i++)
        work[i] = available[i];

    //把未完成进程置为 false
    for (int i = 0; i < processNum; i++)
    {
        bool result = isAllZero(i);
        if (result == true)
```

```
    {
        Finish[i] = true;
        allFinish++;
    }
    else
    {
        Finish[i] = false;
    }
}

//预分配开始
int r = 0; //第 r 个进程
int temp = 0;
while (allFinish != processNum)
{
    num = 0;
    for (int i = 0; i < resourceNum; i++)
    {
        //需要少于现有资源
        if (need[r][i] <= work[i] && Finish[r] == false)
            num++;
    }

    //全都满足要求，可分配
    if (num == resourceNum)
    {
        allFinish++;
        SafeInfo(work, r); //打印
        for (int i = 0; i < resourceNum; i++)
```

```
            {
                work[i] = work[i] + allocation[r][i];
            }
            safeSeries[safeIndex] = r;
            safeIndex++;
            Finish[r] = true;
        }

        r++; //下一个进程

        //找不到，跳出循环
        if (r >= processNum)
        {
            r = r % processNum;
            if (temp == allFinish)
            {
                break;
            }
            temp = allFinish;
        }
        pNum = allFinish;
    }

    //判断系统是否安全
    for (int i = 0; i < processNum; i++)
    {
        //是否有进程处于不安全状态
        if (Finish[i] == false)
        {
```

```
            printf("\n 当前系统不安全！\n\n");
            return false;
        }
    }

    //打印安全序列
    printf("\n 当前系统安全！\n\n 安全序列为：");
    //判断是否已完成，全 0 就是已完成了
    for (int i = 0; i < processNum; i++)
    {
        bool result = isAllZero(i);
        if (result == true)
        {
            pNum--;
        }
    }
    for (int i = 0; i < pNum; i++)
    {
        printf("%d ", safeSeries[i]);
    }
    return true;
}

//主函数
int main()
{
    int curProcess = 0;
    int a = -1;
    showInfo();
```

```
    printf("\n 系统安全情况分析\n");
    printf(" PID\t Work\t\tAllocation\t Need\t\tWork+Allocation\n");
    bool isStart = isSafe();
    //用户输入或者预设系统资源分配合理才能继续进行进程分配工作
    while (isStart)
    {
        //限制用户输入，以防用户输入大于进程数量的数字，以及输入其他字符（乱输是不允许的）
        do
        {
            if (curProcess >= processNum || a == 0)
            {
                printf("\n 请不要输入超出进程数量的值或者其他字符：\n");
                while (getchar() != '\n')
                {
                }; //清空缓冲区
                a = -1;
            }
            printf("\n--------------------------------------------------------------------------------\n");
            printf("\n 输入要分配的进程：");
            a = scanf("%d", &curProcess);
            printf("\n");

        } while (curProcess >= processNum || a == 0);

        //限制用户输入，此处只接受数字，以防用户输入其他字符（乱输是不允许的）
        for (int i = 0; i < resourceNum; i++)
        {
```

```
            do
            {
                if (a == 0)
                {
                    printf("\n 请不要输入除数字以外的其他字符，请重新输入：\n");
                    while (getchar() != '\n')
                    {
                    }; //清空缓冲区
                    a = -1;
                }
                printf("请输入要分配给进程 P%d 的第 %d 类资源：", curProcess, i + 1);
                a = scanf("%d", &request[i]);
            } while (a == 0);
        }

        //判断用户输入的分配是否合理，如果合理，开始进行预分配
        num = 0;
        for (int i = 0; i < resourceNum; i++)
        {
            if (request[i] <= need[curProcess][i] && request[i] <= available[i])
            {
                num++;
            }
            else
            {
                printf("\n 发生错误！可能原因如下：\n(1)您请求分配的资源可能大于该进程的某些资源的最大需要！\n(2)系统所剩的资源已经不足了！\n");
                break;
            }
```

```
        }
        //合理
        if (num == resourceNum)
        {
            num = 0;
            for (int j = 0; j < resourceNum; j++)
            {
                //分配资源
                available[j] = available[j] - request[j];
                allocation[curProcess][j] = allocation[curProcess][j] + request[j];
                need[curProcess][j] = need[curProcess][j] - request[j];
                //记录分配以后，是否该进程需要值为 0 了
                if (need[curProcess][j] == 0)
                {
                    num++;
                }
            }
            //如果分配以后出现该进程对所有资源的需求为 0 了，即刻释放该进程占用
资源（视为完成）
            if (num == resourceNum)
            {
                //释放已完成资源
                for (int i = 0; i < resourceNum; i++)
                {
                    available[i] = available[i] + allocation[curProcess][i];
                }
                printf("\n\n 本次分配进程  P%d  完成,该进程占用资源全部释放完毕！
\n", curProcess);
            }
```

```
            else
            {
                //资源分配可以不用一次性满足进程需求
                printf("\n\n 本次分配进程 P%d 未完成！\n", curProcess);
            }

            showInfo();
            printf("\n 系统安全情况分析\n");
            printf(" PID\t Work\t\tAllocation\t Need\t\tWork+Allocation\n");

            //预分配完成以后，判断该系统是否安全，若安全，则可继续进行分配，若不安全，将已经分配的资源换回来
            if (!isSafe())
            {
                for (int j = 0; j < resourceNum; j++)
                {
                    available[j] = available[j] + request[j];
                    allocation[curProcess][j] = allocation[curProcess][j] - request[j];
                    need[curProcess][j] = need[curProcess][j] + request[j];
                }
                printf("资源不足，等待中...\n\n 分配失败！\n");
            }
        }
    }
    return 0;
}
```

运行结果

```
--------------------------------------------------------------------------------
当前系统各类资源剩余：3 3 2

当前系统资源情况：
 PID      Max          Allocation       Need
 P0       7 5 3        0 1 0            7 4 3
 P1       3 2 2        2 0 0            1 2 2
 P2       9 0 2        3 0 2            6 0 0
 P3       2 2 2        2 1 1            0 1 1
 P4       4 3 3        0 0 2            4 3 1

系统安全情况分析
 PID      Work         Allocation       Need          Work+Allocation
 P1       3 3 2        2 0 0            1 2 2          5 3 2
 P3       5 3 2        2 1 1            0 1 1          7 4 3
 P4       7 4 3        0 0 2            4 3 1          7 4 5
 P0       7 4 5        0 1 0            7 4 3          7 5 5
 P2       7 5 5        3 0 2            6 0 0         10 5 7

当前系统安全！

安全序列为：1 3 4 0 2
--------------------------------------------------------------------------------

输入要分配的进程：1

请输入要分配给进程 P1 的第 1 类资源：1
```

请输入要分配给进程 P1 的第 2 类资源：2

请输入要分配给进程 P1 的第 3 类资源：2

本次分配进程 P1 完成,该进程占用资源全部释放完毕！

当前系统各类资源剩余：5 3 2

当前系统资源情况：

PID	Max	Allocation	Need
P0	7 5 3	0 1 0	7 4 3
P1	3 2 2	3 2 2	0 0 0
P2	9 0 2	3 0 2	6 0 0
P3	2 2 2	2 1 1	0 1 1
P4	4 3 3	0 0 2	4 3 1

系统安全情况分析

PID	Work	Allocation	Need	Work+Allocation
P3	5 3 2	2 1 1	0 1 1	7 4 3
P4	7 4 3	0 0 2	4 3 1	7 4 5
P0	7 4 5	0 1 0	7 4 3	7 5 5
P2	7 5 5	3 0 2	6 0 0	10 5 7

当前系统安全！

安全序列为：3 4 0 2

```
输入要分配的进程：0


请输入要分配给进程 P0 的第 1 类资源：3
请输入要分配给进程 P0 的第 2 类资源：3
请输入要分配给进程 P0 的第 3 类资源：3


发生错误！可能原因如下：
(1)您请求分配的资源可能大于该进程的某些资源的最大需要！
(2)系统所剩的资源已经不足了！


--------------------------------------------------------------------------------


输入要分配的进程：
```

结论展望

银行家算法的难点在于安全性检查，具体的实现过程我已经放在算法思想里头了，这里就不再赘述了。这是一个很好的算法，它能够在分配之前通过预分配的方法来找出安全序列，从而判断是否安全，这样一来就很有效地避免了死锁。

当然，它也有一些不足，比如说开销大。如何去优化这个算法，让它在实现上更加简单是一个很大的挑战。

实验二：处理机管理

实验题目

设计程序模拟进程的轮转法调度过程。假设初始状态为：有 n 个进程处于就绪状态，有 m 个进程处于阻塞状态。采用轮转法进程调度算法、高响应比优先（HRRN）进行调度(调度过程中，假设处于执行状态的进程不会阻塞)，且每过 t 个时间片系统释放资源，唤醒处于阻塞队列队首的进程。

设计要求

1. 输出系统中进程的调度次序；
2. 计算 CPU 利用率。

需求分析

对于多个就绪的进程，我们该如何分配它们的执行顺序？这是个非常重要的问题，它将直接关系到运行的效率和时间。时间片的加入让公平性得到了保障（长作业不至于发生饥饿），而优先级或者顺序执行则可根据需要来选择。

数据结构

```
//全局
struct PCB
{
    int pid;            //进程 id
    int totaltime;     //要求运行时间
    int waittime;      //等待时间
    int runtime;       //服务时间
    double priority; //响应比=(服务时间+等待时间)/服务时间，这里看作优先级理解
} pcb[100];
```

```
int n;  //就绪状态进程个数
int m;  //阻塞状态进程个数
int t;  //释放资源时间
int tt; //时间片大小

//局部
int count = 0;       //时间片个数
int wastedTime = 0; //CPU 空转时间
```

这里需要说明的是，结构体数组 `pcb` 中前 n 个表示就绪进程，后 m 个表示阻塞进程。

算法思想

高响应比调度算法：

首先，我们要知道响应比的计算公式：响应比等待时间运行时间运行时间 。

然后，我们找最高优先级（也就是最高响应比）的就绪进程先运行一个时间片。

同时，每隔 t 个时间片系统释放资源，唤醒一个阻塞的进程（也就是说，让它变成就绪进程，实现方式是 `n++; m--;`）。

这里，如果运行的进程在时间片结束前就完成了或者就绪进程全部运行完成但是还有阻塞进程未被唤醒，就会产生 CPU 的空转，需要记录这个时间。

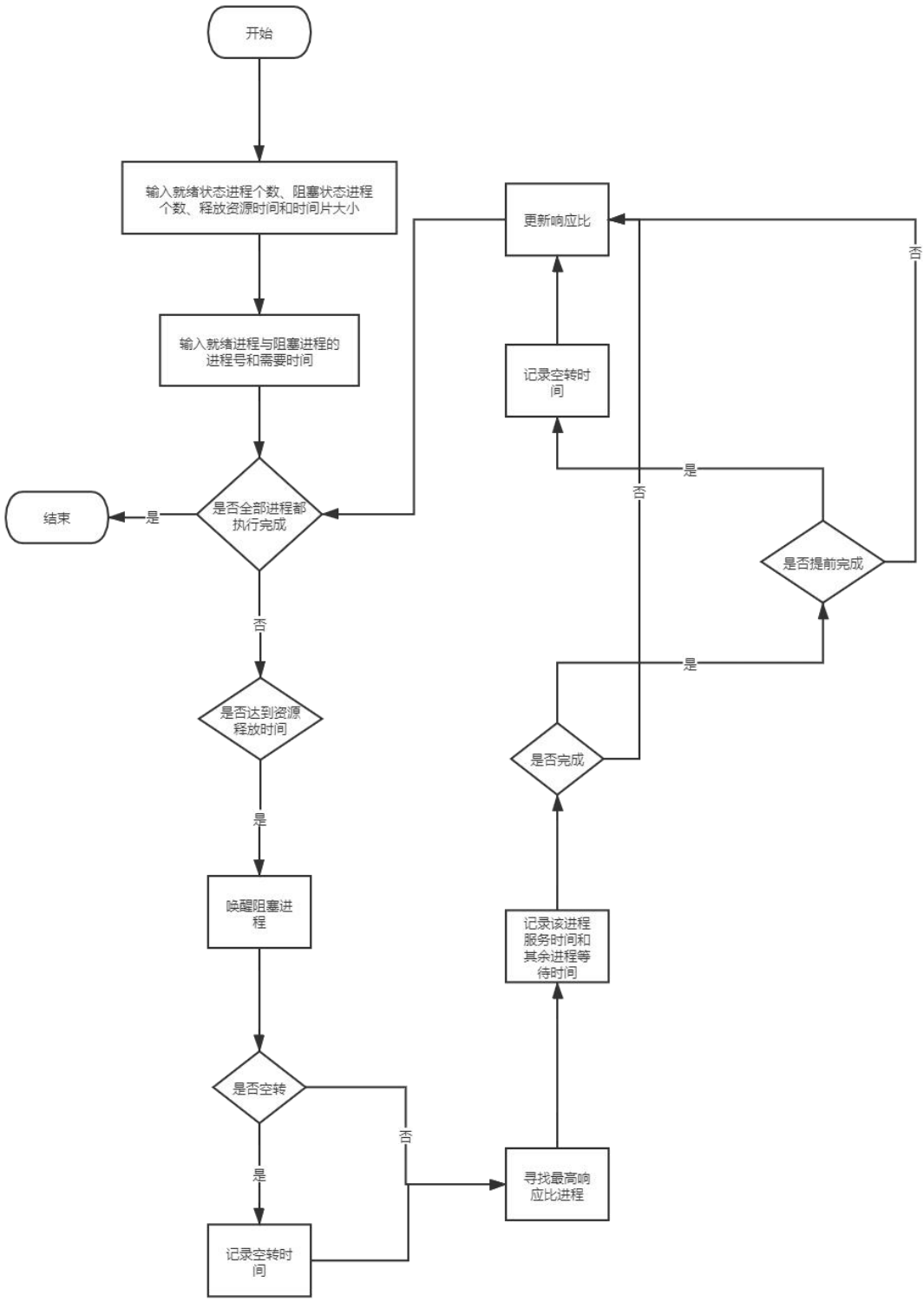
最后，我们通过 利用率总时间浪费时间总时间 这个公式即可计算出 CPU 利用率。

轮转法：

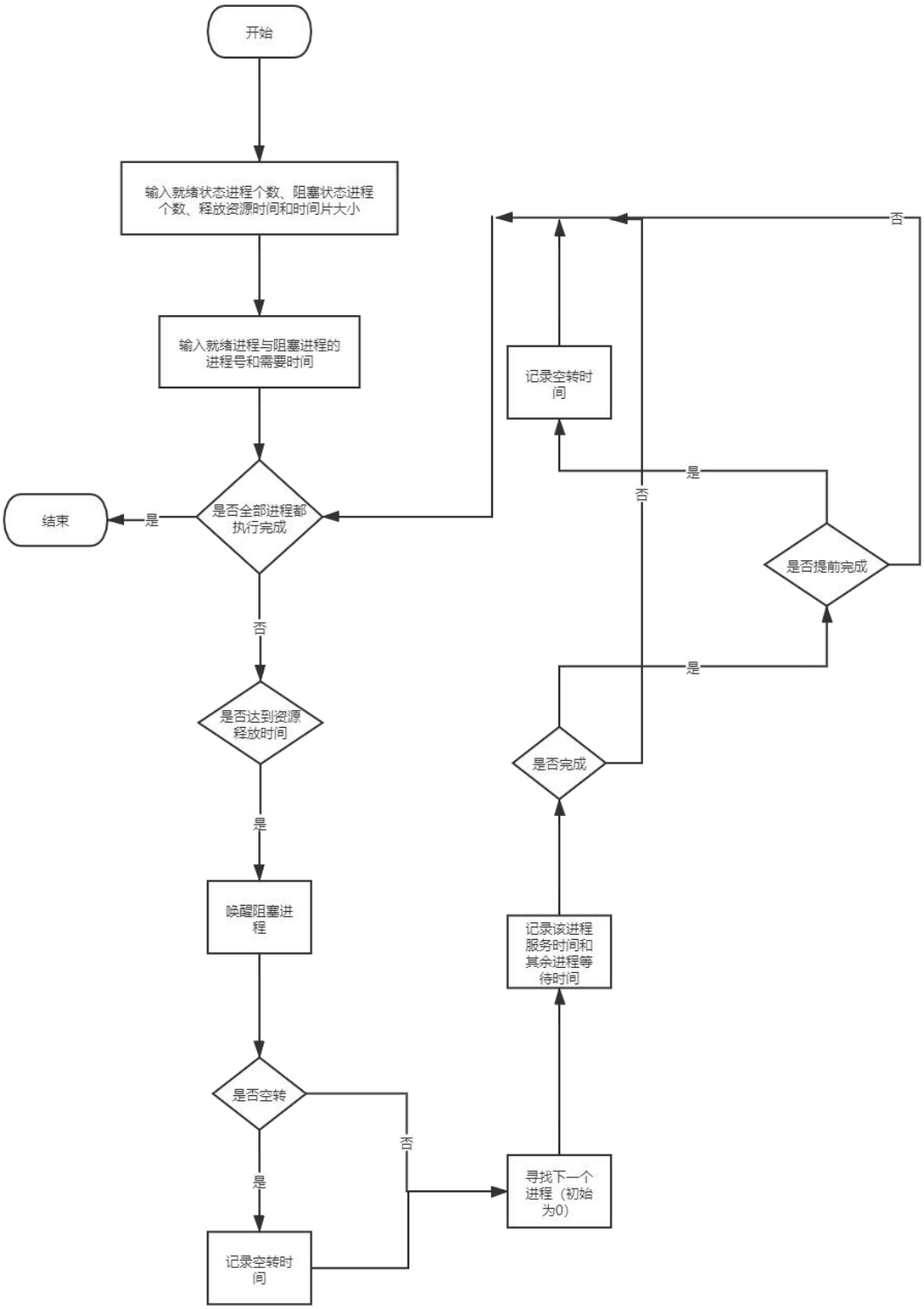
这个和高响应比类似，区别就是不用算响应比，直接按照顺序进行时间片的轮转即可。

算法流程

高响应比调度算法：



轮转法：


开始
输入就绪状态进程个数、阻塞状态进程个数、释放资源时间和时间片大小
输入就绪进程与阻塞进程的进程号和需要时间
是否全部进程都执行完成
是
结束
否
是否达到资源释放时间
是
唤醒阻塞进程
是否空转
是
记录空转时间
否
寻找下一个进程（初始为0）
记录该进程服务时间和其余进程等待时间
是否完成
否
是
记录空转时间
否
是
是否提前完成
否

算法实现

高响应比调度算法

```
/*
测试数据：
1 8 3 2
1 1
2 2
3 1
4 3
5 2
6 5
7 1
8 3
9 4
*/
#include <stdio.h>
#include <stdlib.h>

struct PCB
{
    int pid;            //进程 id
    int totaltime;    //要求运行时间
    int waittime;     //等待时间
    int runtime;      //服务时间
    double priority; //响应比=(服务时间+等待时间)/服务时间，这里看作优先级理解
} pcb[100];

int n;  //就绪状态进程个数
```

```
int m;  //阻塞状态进程个数
int t;  //释放资源时间
int tt; //时间片大小

//高响应比调度
void HRRN()
{
    for (int i = 0; i < n; ++i)
        printf("进程%d 初始状态：\t 服务时间:%d\t 需要时间:%d\t 等待时间:%d\n",
pcb[i].pid, pcb[i].runtime, pcb[i].totaltime, pcb[i].waittime);

    int count = 0;       //时间片个数
    int wastedTime = 0; //CPU 空转时间

    while (n > 0 || m > 0)
   {
        //唤醒一个阻塞进程
        if (count % t == 0 && m > 0)
        {
            printf("阻塞进程%d 被唤醒\n", pcb[n].pid);
            n++;
            m--;
        }

        //判断空转
        if (n == 0 && m != 0)
        {
            count++;
            wastedTime += tt;
```

```
            for (int i = 0; i < m; ++i)
                pcb[i].waittime += tt;
            continue;
        }

        int maxPId = 0; //优先级最高的 id

        //寻找最高优先级的 pcb
        for (int i = 1; i < n; ++i)
        {
            if (pcb[i].priority > pcb[maxPId].priority)
                maxPId = i;
        }

        //最高优先级进程运行
        count++;
        pcb[maxPId].runtime += tt;
        //其余进程等待时间+1
        for (int i = 0; i < m; ++i)
        {
            if (i != maxPId)
                pcb[i].waittime += tt;
            if (pcb[i].runtime > 0)
                pcb[i].priority = 1.0 * (pcb[i].runtime + pcb[i].waittime) / pcb[i].runtime;
        }
        printf("第%d 片时间:  进程%d 运行\t 服务时间:%d\t 需要时间:%d\t 等待时
间:%d\n", count, pcb[maxPId].pid, pcb[maxPId].runtime, pcb[maxPId].totaltime,
pcb[maxPId].waittime);
```

```
        //判断是否完成
        if (pcb[maxPId].runtime >= pcb[maxPId].totaltime)
        {
            int wt = tt * (pcb[maxPId].runtime - pcb[maxPId].totaltime);
            wastedTime += tt;
            for (int i = 0; i < m + n; ++i)
                pcb[i].waittime += tt;
            printf("第%d 片时间:  进程%d 结束\n", count, pcb[maxPId].pid);
            //释放 pcb[maxPId]
            for (int i = maxPId; i < n + m; ++i)
            {
                pcb[i].pid = pcb[i + 1].pid;
                pcb[i].priority = pcb[i + 1].priority;
                pcb[i].runtime = pcb[i + 1].runtime;
                pcb[i].totaltime = pcb[i + 1].totaltime;
                pcb[i].waittime = pcb[i + 1].waittime;
            }
            n--;
        }
    }

    printf("CPU 利用率  = (总时间-浪费时间) /  总时间  * 100% = %.2f%", (1.0 * count *
tt - 1.0 * wastedTime) / (1.0 * count * tt) * 100);
}

int main()
{
    printf("就绪状态进程个数  阻塞状态进程个数  释放资源时间 t  时间片大小 tt\n");
    scanf("%d%d%d%d", &n, &m, &t, &tt);
```

```
    for (int i = 0; i < n; ++i)
    {
        printf("就绪状态进程的进程号  需要时间 : ");
        scanf("%d %d", &pcb[i].pid, &pcb[i].totaltime);
        pcb[i].runtime = 0;
        pcb[i].waittime = 0;
        pcb[i].priority = -1.0;
    }
    for (int i = n; i < n + m; ++i)
    {
        printf("阻塞状态进程的进程号  需要时间 : ");
        scanf("%d %d", &pcb[i].pid, &pcb[i].totaltime);
        pcb[i].runtime = 0;
        pcb[i].waittime = 0;
        pcb[i].priority = -1.0;
    }

    HRRN();
    return 0;
}
```

轮转法

```
/*
测试数据：
1 8 3 2
1 1
2 2
```

```
3 1
4 3
5 2
6 5
7 1
8 3
9 4
*/
#include <stdio.h>
#include <stdlib.h>

struct PCB
{
    int pid;         //进程 id
    int totaltime; //要求运行时间
    int waittime;   //等待时间
    int runtime;    //服务时间
} pcb[100];

int n;  //就绪状态进程个数
int m;  //阻塞状态进程个数
int t;  //释放资源时间
int tt; //时间片大小

//轮转法
void RR()
{
    for (int i = 0; i < n; ++i)
        printf("进程%d 初始状态：\t 服务时间:%d\t 需要时间:%d\t 等待时间:%d\n",
```

```
pcb[i].pid, pcb[i].runtime, pcb[i].totaltime, pcb[i].waittime);

    int count = 0;        //时间片个数
    int wastedTime = 0; //CPU 空转时间
    int cn = 0;           //当前运行的进程

    while (n > 0 || m > 0)
    {
        //唤醒一个阻塞进程
        if (count % t == 0 && m > 0)
        {
            printf("阻塞进程%d 被唤醒\n", pcb[n].pid);
            n++;
            m--;
        }

        //判断空转
        if (n == 0 && m != 0)
        {
            count++;
            wastedTime += tt;
            for (int i = 0; i < m; ++i)
                pcb[i].waittime += tt;
            continue;
        }

        //模拟循环队列
        if (cn == n - 1)
        {
```

```
            cn = 0;
        }

        //当前进程运行
        count++;
        pcb[cn].runtime += tt;
        //其余进程等待时间+1
        for (int i = 0; i < n; ++i)
        {
            if (i != cn)
                pcb[i].waittime += tt;
        }
        printf("第%d 片时间:  进程%d 运行\t 服务时间:%d\t 需要时间:%d\t 等待时间:%d\n", count, pcb[cn].pid, pcb[cn].runtime, pcb[cn].totaltime, pcb[cn].waittime);

        //判断是否完成
        if (pcb[cn].runtime >= pcb[cn].totaltime)
        {
            int wt = tt * (pcb[cn].runtime - pcb[cn].totaltime);
            wastedTime += tt;
            for (int i = 0; i < m + n; ++i)
                pcb[i].waittime += tt;
            printf("第%d 片时间:  进程%d 结束\n", count, pcb[cn].pid);
            //释放 pcb[cn]
            for (int i = cn; i < n + m; ++i)
            {
                pcb[i].pid = pcb[i + 1].pid;
                pcb[i].runtime = pcb[i + 1].runtime;
                pcb[i].totaltime = pcb[i + 1].totaltime;
```

```
                pcb[i].waittime = pcb[i + 1].waittime;
            }
            n--;
        }
    }

    printf("CPU 利用率 = (总时间-浪费时间) / 总时间 * 100% = %.2f%", (1.0 * count * tt - 1.0 * wastedTime) / (1.0 * count * tt) * 100);
}

int main()
{
    printf("就绪状态进程个数 阻塞状态进程个数 释放资源时间 t 时间片大小 tt\n");
    scanf("%d%d%d%d", &n, &m, &t, &tt);

    for (int i = 0; i < n; ++i)
    {
        printf("就绪状态进程的进程号 需要时间 : ");
        scanf("%d %d", &pcb[i].pid, &pcb[i].totaltime);
        pcb[i].runtime = 0;
        pcb[i].waittime = 0;
    }
    for (int i = n; i < n + m; ++i)
    {
        printf("阻塞状态进程的进程号 需要时间 : ");
        scanf("%d %d", &pcb[i].pid, &pcb[i].totaltime);
        pcb[i].runtime = 0;
        pcb[i].waittime = 0;
    }
```

```
    RR();
    return 0;
}
```

运行结果

高响应比置换算法

```
就绪状态进程个数  阻塞状态进程个数  释放资源时间 t  时间片大小 tt
1 8 3 2
就绪状态进程的进程号  需要时间  : 1 1
阻塞状态进程的进程号  需要时间  : 2 2
阻塞状态进程的进程号  需要时间  : 3 1
阻塞状态进程的进程号  需要时间  : 4 3
阻塞状态进程的进程号  需要时间  : 5 2
阻塞状态进程的进程号  需要时间  : 6 5
阻塞状态进程的进程号  需要时间  : 7 1
阻塞状态进程的进程号  需要时间  : 8 3
阻塞状态进程的进程号  需要时间  : 9 4
进程 1 初始状态：  服务时间:0      需要时间:1      等待时间:0
阻塞进程 2 被唤醒
第 1 片时间:  进程 1 运行    服务时间:2      需要时间:1      等待时间:0
第 1 片时间:  进程 1 结束
第 2 片时间:  进程 2 运行    服务时间:2      需要时间:2      等待时间:4
第 2 片时间:  进程 2 结束
阻塞进程 3 被唤醒
第 4 片时间:  进程 3 运行    服务时间:2      需要时间:1      等待时间:10
第 4 片时间:  进程 3 结束
```

```
阻塞进程 4 被唤醒
第 7 片时间: 进程 4 运行    服务时间:2      需要时间:3      等待时间:18
第 8 片时间: 进程 4 运行    服务时间:4      需要时间:3      等待时间:18
第 8 片时间: 进程 4 结束
阻塞进程 5 被唤醒
第 10 片时间: 进程 5 运行    服务时间:2      需要时间:2      等待时间:26
第 10 片时间: 进程 5 结束
阻塞进程 6 被唤醒
第 13 片时间: 进程 6 运行    服务时间:2      需要时间:5      等待时间:34
第 14 片时间: 进程 6 运行    服务时间:4      需要时间:5      等待时间:34
第 15 片时间: 进程 6 运行    服务时间:6      需要时间:5      等待时间:34
第 15 片时间: 进程 6 结束
阻塞进程 7 被唤醒
第 16 片时间: 进程 7 运行    服务时间:2      需要时间:1      等待时间:42
第 16 片时间: 进程 7 结束
阻塞进程 8 被唤醒
第 19 片时间: 进程 8 运行    服务时间:2      需要时间:3      等待时间:48
第 20 片时间: 进程 8 运行    服务时间:4      需要时间:3      等待时间:48
第 20 片时间: 进程 8 结束
阻塞进程 9 被唤醒
第 22 片时间: 进程 9 运行    服务时间:2      需要时间:4      等待时间:34
第 23 片时间: 进程 9 运行    服务时间:4      需要时间:4      等待时间:34
第 23 片时间: 进程 9 结束
CPU 利用率  = (总时间-浪费时间) /  总时间  * 100= 21.74
```

轮转法

```
就绪状态进程个数  阻塞状态进程个数  释放资源时间 t  时间片大小 tt
1 8 3 2
就绪状态进程的进程号  需要时间  : 1 1
```

阻塞状态进程的进程号 需要时间 : 2 2

阻塞状态进程的进程号 需要时间 : 3 1

阻塞状态进程的进程号 需要时间 : 4 3

阻塞状态进程的进程号 需要时间 : 5 2

阻塞状态进程的进程号 需要时间 : 6 5

阻塞状态进程的进程号 需要时间 : 7 1

阻塞状态进程的进程号 需要时间 : 8 3

阻塞状态进程的进程号 需要时间 : 9 4

进程 1 初始状态： 服务时间:0 需要时间:1 等待时间:0

阻塞进程 2 被唤醒

第 1 片时间: 进程 1 运行 服务时间:2 需要时间:1 等待时间:0

第 1 片时间: 进程 1 结束

第 2 片时间: 进程 2 运行 服务时间:2 需要时间:2 等待时间:4

第 2 片时间: 进程 2 结束

阻塞进程 3 被唤醒

第 4 片时间: 进程 3 运行 服务时间:2 需要时间:1 等待时间:6

第 4 片时间: 进程 3 结束

阻塞进程 4 被唤醒

第 7 片时间: 进程 4 运行 服务时间:2 需要时间:3 等待时间:12

第 8 片时间: 进程 4 运行 服务时间:4 需要时间:3 等待时间:12

第 8 片时间: 进程 4 结束

阻塞进程 5 被唤醒

第 10 片时间: 进程 5 运行 服务时间:2 需要时间:2 等待时间:16

第 10 片时间: 进程 5 结束

阻塞进程 6 被唤醒

第 13 片时间: 进程 6 运行 服务时间:2 需要时间:5 等待时间:22

第 14 片时间: 进程 6 运行 服务时间:4 需要时间:5 等待时间:22

第 15 片时间: 进程 6 运行 服务时间:6 需要时间:5 等待时间:22

第 15 片时间: 进程 6 结束

```
阻塞进程 7 被唤醒
第 16 片时间: 进程 7 运行    服务时间:2      需要时间:1      等待时间:24
第 16 片时间: 进程 7 结束
阻塞进程 8 被唤醒
第 19 片时间: 进程 8 运行    服务时间:2      需要时间:3      等待时间:30
第 20 片时间: 进程 8 运行    服务时间:4      需要时间:3      等待时间:30
第 20 片时间: 进程 8 结束
阻塞进程 9 被唤醒
第 22 片时间: 进程 9 运行    服务时间:2      需要时间:4      等待时间:34
第 23 片时间: 进程 9 运行    服务时间:4      需要时间:4      等待时间:34
第 23 片时间: 进程 9 结束
CPU 利用率 = (总时间-浪费时间) / 总时间 * 100= 21.74
```

结论展望

无论是时间片轮转算法还是高响应比优先算法，都是很好的算法，它们解决了一个很大的问题，就是长作业的饥饿。但是如何选取时间片长度来让效率达到最高又是一个新的问题。

实验三：可变式分区管理

实验题目

（1）采用空闲区表，并增加已分配区表 {未分配区说明表、已分配区说明表（分区号、起始地址、长度、状态）}。分配算法采用最佳适应算法（内存空闲区按照尺寸大小从小到大的排列）和循环首次适应算法，实现内存的分配与回收。

（2）采用空闲区链法管理空闲区，结构如 P137/P128，并增加已分配区表。分配算法分别采用首次适应法（内存空闲区的地址按照从小到大的自然顺序排列）和最佳适应法（按照内存大小从小到大排列），实现内存的分配与回收。

设计要求

1. 学生自己设计一个进程申请序列以及进程完成后的释放顺序，实现主存的分配与回收。
2. 进程分配时，应该考虑以下 3 中情况：进程申请的空间小于、大于或者等于系统空闲区的大小。
3. 回收时，应该考虑 4 种情况：释放区上邻、下邻、上下都邻接和都不邻接空闲区。
4. 每次的分配与回收，都要求把记录内存使用情况的各种数据结构的变化情况以及各进程的申请、释放情况显示或打印出来。

需求分析

可变式分区管理是操作系统用来存放任务的重要手段，也能显著地提高分区的灵活性。不同的分区管理有不同的优缺点，比如最佳适应法就容易产生很多小的细碎的空间但它每次选择的分区都是最合适的。具体还是要按照需求来选择。

数据结构

```
int pos, n, Size; //查找位置,分区数量,最小分割大小


//分区
struct List
{
    int id;           //空闲分区编号
    int startAdress; //空闲分区首地址
    int room;         //空间
    int state;        //状态，0 为空，1 为未满，2 为满；
} L[2000];


//作业
struct Task
{
    int id;           //作业编号
    int room;         //作业空间
    int startAdress; //作业起始地址
} T[2000];
```

算法思想

回收内存：

先判断是否与空闲分区有邻接：

- 作业起始地址是分区的末尾地址，那就是上邻接空闲区。
- 作业末尾位置是分区的起始位置，那就是下邻接空闲区。

接下来又有四种情况：

- 上邻接空闲区：分区空间扩充作业空间即可。
- 下邻接空闲区：分区扩充空间的同时修改起始地址。
- 上下都邻接空闲区：此时两个分区合二为一，分区扩充空间的同时，分区数量减一，分区编号前移。
- 上下都不邻接空闲区：此时一个分区分为两个分区，寻找作业后面的第一个分区，自此以后的分区编号后移，分区数量加一，为新分区设置参数。

循环首次适应算法：

首先，接收用户输入的作业的大小。

然后，从当前位置 `pos` 开始找一个合适的分区存放作业（如果找到末尾则跳回第一个来模拟循环链表）。

分区分为四种情况：

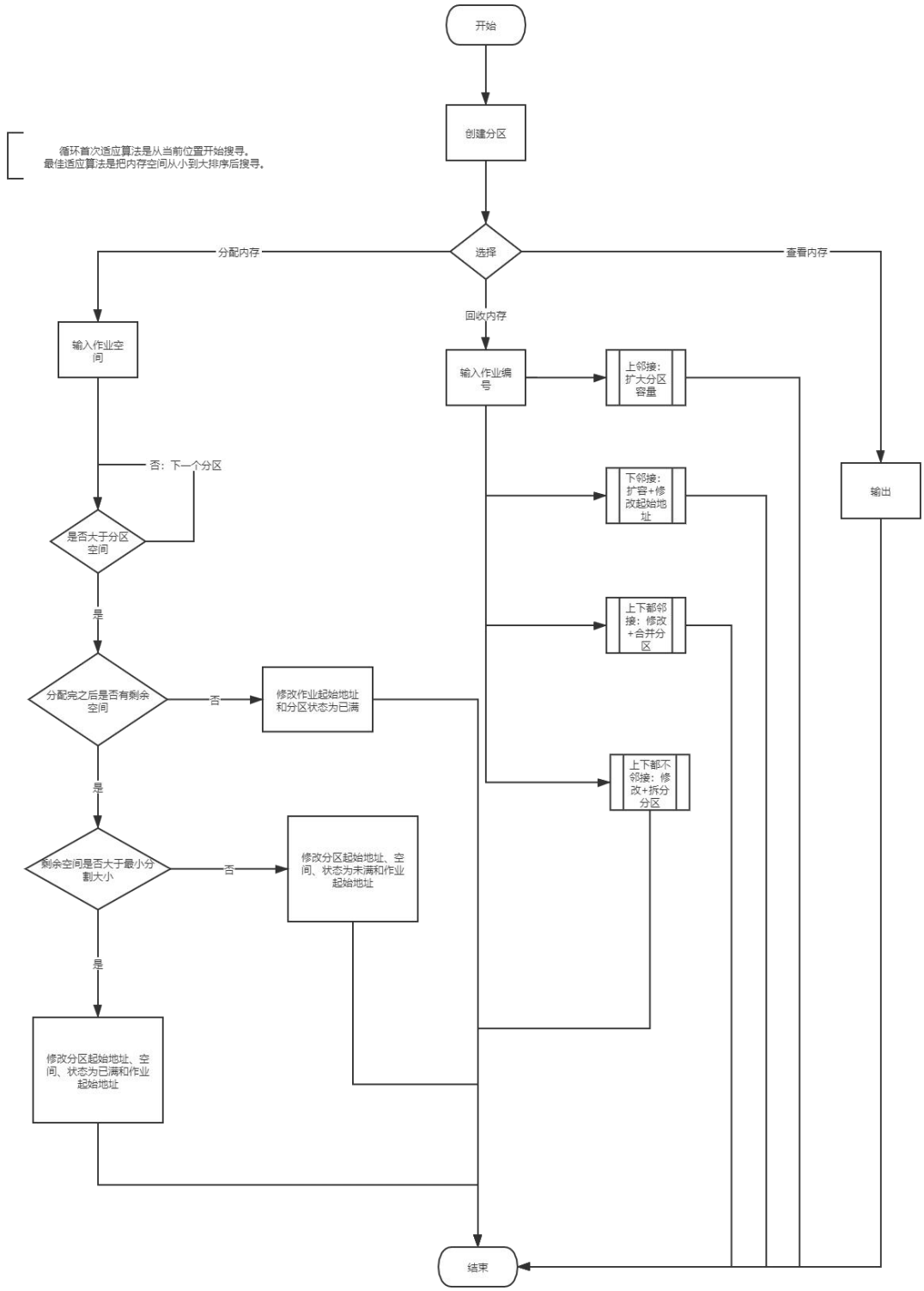
- 分配完后剩余空间大于最小分割大小：存放作业，修改分区起始地址、空间、状态和作业的起始地址。
- 分配完之后剩余空间小于最小分割大小：存放作业，修改分区起始地址、空间、状态和作业的起始地址。
- 分区刚好能放下该作业：存放作业，修改作业的起始地址和分区的状态即可。
- 作业过大：提示信息并退出循环。

最佳适应算法

整体逻辑和循环首次适应算法一样，但是加入了排序语句。

将分区空间从小到大排序，然后从最小的开始循环，这样每次找到的分区就都是最佳的分区。

算法流程


开始
创建分区
循环首次适应算法是从当前位置开始搜寻。
最佳适应算法是把内存空间从小到大排序后搜寻。
选择
分配内存
查看内存
回收内存
输入作业空间
输入作业编号
上邻接：扩大分区容量
下邻接：扩容+修改起始地址
上下都邻接：修改+合并分区
上下都不邻接：修改+拆分分区
输出
否：下一个分区
是否大于分区空间
是
分配完之后是否有剩余空间
否
修改作业起始地址和分区状态为已满
是
剩余空间是否大于最小分割大小
否
修改分区起始地址、空间、状态为未满和作业起始地址
是
修改分区起始地址、空间、状态为已满和作业起始地址
结束

算法实现

```
/*
测试数据：
循环首次适应算法：
5
50 85
32 155
70 275
60 532
10 980
3
1
1
25
3
2
1
3
0

最佳适应算法：
5
50 85
32 155
70 275
60 532
10 980
```

```
3
2
1
25
3
2
1
3
0
*/
#include <iostream>
#include <cmath>
#include <stdio.h>
#include <algorithm>
#include <string.h>
using namespace std;

int pos, n, Size; //查找位置,分区数量,最小分割大小

//分区
struct List
{
    int id;            //空闲分区编号
    int startAdress; //空闲分区首地址
    int room;          //空间
    int state;         //状态，0 为空，1 为未满，2 为满；
} L[2000];

//作业
```

```
struct Task
{
    int id;            //作业编号
    int room;          //作业空间
    int startAdress; //作业起始地址
} T[2000];

bool cmp(List a, List b)
{
    return a.room < b.room;
}

//输出分区
void print()
{
    int i;
    printf("|-----------------------------------------------------------|\n");
    printf("|          分区号      分区始址      分区大小      分区状态          |\n");
    printf("|-----------------------------------------------------------|\n");
    for (i = 1; i <= n; i++)
   {
        printf("|          %3d          %3d          %3d          %3d              |\n",
                L[i].id, L[i].startAdress, L[i].room, L[i].state);
        printf("|-----------------------------------------------------------|\n");
   }
}

//回收内存
void recycle()
```

```
{
    printf("请输入要释放占用空间任务的序号：\n");
    int l;                  //要回收的任务编号
    int f1 = 0, f2 = 0; //f1 判断是否上邻接空闲区，f2 判断是否下邻接空闲区
    int p1, p2;             //p1 保存上邻接的分区序号，p2 保存下邻接的分区序号
    cin >> l;               //输入序号
    for (int i = 1; i <= n; i++)
    {
        //第 i 块分区空闲区下端，上邻接空闲区
        //起始地址与空闲区下端邻接
        if (T[l].startAdress == L[i].startAdress + L[i].room)
        {
            f1 = 1;
            p1 = i;
        }
        //第 i 块分区空闲区上端，下邻接空闲区
        //末尾地址与空闲区上端邻接
        if (T[l].startAdress + T[l].room == L[i].startAdress)
        {
            f2 = 1;
            p2 = i;
        }
    }

    //情况 1：上邻接空闲区
    if (f1 == 1 && f2 == 0)
    {
        L[p1].room = L[p1].room + T[l].room;
    }
```

```
    //情况 2：下邻接空闲区
    else if (f1 == 0 && f2 == 1)
    {
        L[p2].room = L[p2].room + T[l].room;
        L[p2].startAdress = T[l].startAdress;
    }

    //情况 3：上下都邻接空闲区，两个分区合二为一
    else if (f1 == 1 && f2 == 1)
    {
        L[p1].room = L[p1].room + T[l].room + L[p2].room;
        //其余分区编号前移
        for (int j = p1 + 1; j <= n; j++)
            L[j].id--;
        n--; //分区数-1
    }

    //情况 4：上下都不邻接空闲区，一个分区分为两个分区
    else if (f1 == 0 && f2 == 0)
    {
        int temp;
        //找到排在它后面的第一个分区
        for (int j = 1; j <= n; j++)
       {
            if (L[j].startAdress > T[l].startAdress + T[l].room) //实际上大于
T[l].startAdress 就可以
            {
                temp = j;
```

```
                break;
            }
        }
        //后面的分区编号后移
        for (int j = temp; j <= n; j++)
            L[j].id++;
        n++; //分区数+1
        L[temp].id = temp;
        L[temp].room = T[l].room;
        L[temp].startAdress = T[l].startAdress;
        L[temp].state = 0;
    }
    cout << "内存回收完毕！" << endl;
}

//循环首次适应算法
//从当前位置不断扫描
void NF()
{
    cout << "选择采用'循环首次适应算法'进行内存分配\n"
         << endl;
    int tmp;
    pos = 1; //开始从第一个分区查找
    while (1)
    {
        cout << "      ****************************************" << endl;
        cout << "      1: 分配内存              2: 回收内存        " << endl;
        cout << endl;
        cout << "      3: 查看空闲分区表        0: 退出        " << endl;
```

```
        cout << "      ****************************************" << endl;
        cout << "请输入您的操作 ： ";
        cin >> tmp;
        int k = 0;

        //分配内存
        if (tmp == 1)
        {
            k++;
            printf("请输入第%d 个作业占用空间大小：\n", k);
            cin >> T[k].room; //作业空间
            T[k].id = k;
            int num = 0;
            for (int i = pos;; i++)
            {
                num++;
                if (num > n)
                {
                    printf("作业请求内存空间过大，空闲分区表不能满足要求，内存分配失败!\n");
                    break;
                }

                //模拟循环列表
                if (i > n)
                {
                    i = 1;
                    continue;
                }
```

```
            //第 i 个分区容得下该作业，且有剩余空间，则放入
            if ((L[i].state == 0 || L[i].state == 1) && L[i].room >= T[i].room)
            {
                //第一种情况：剩余空间大于最小分割大小
                if (L[i].room - T[k].room > Size)
                {
                    L[i].startAdress = L[i].startAdress + T[k].room;
                    L[i].room = L[i].room - T[k].room;
                    L[i].state = 1;
                    T[k].startAdress = L[i].startAdress - T[k].room;
                    printf("内存分配成功!\n作业申请空间为%d\n起始地址为%d\n",
T[k].room, T[k].startAdress);
                    break;
                }
                //第二种情况：剩余空间小于最小分割大小
                else
                {
                    L[i].startAdress = L[i].startAdress + Size;
                    L[i].room = L[i].room - Size;
                    L[i].state = 2;
                    T[k].startAdress = L[i].startAdress - Size;
                    printf("内存分配成功!\n作业申请空间为%d\n起始地址为%d\n",
L[i].room, T[k].startAdress);
                    break;
                }
            }
            //分区正好能放下该作业
            else if (L[i].state = 0 && L[i].room - T[k].room == Size)
```

```
                {
                    L[i].state = 2;
                    T[k].startAdress = L[i].startAdress;
                    printf("内存分配成功！ \n 作业申请空间为%d\n 起始地址为%d\n",
T[k].room, T[k].startAdress);
                    break;
                }
            }
        }

        //回收内存
        else if (tmp == 2)
        {
            recycle();
        }

        //查看内存
        else if (tmp == 3)
            print();

        //退出
        else if (tmp == 0)
            return;
        else
        {
            printf("输入有误，请重新输入！ \n");
            continue;
        }
    }
```

```
}

//最佳适应算法
//空闲区按从小到大递增顺序
void BF()
{
    cout << "选择采用'最佳适应算法'进行内存分配\n"
         << endl;
    sort(L + 1, L + n + 1, cmp);
    int tmp;
    while (1)
    {
        cout << "      ****************************************" << endl;
        cout << "      1: 分配内存              2: 回收内存        " << endl;
        cout << endl;
        cout << "      3: 查看空闲分区表        0: 退出          " << endl;
        cout << "      ****************************************" << endl;
        cout << "请输入您的操作 ： ";
        cin >> tmp;
        int k = 0; //作业序号
        if (tmp == 1)
        {
            k++;
            printf("请输入第%d 个作业占用空间大小：\n", k);
            cin >> T[k].room;
            T[k].id = k;
            int i;
            for (i = 1; i <= n; i++)
            {
```

```
            //模拟循环列表
            if (i > n)
            {
                i = 1;
                continue;
            }

            //第 i 个分区容得下该作业，且有剩余空间，则放入
            if ((L[i].state == 0 || L[i].state == 1) && L[i].room >= T[i].room)
            {
                //第一种情况：剩余空间大于最小分割大小
                if (L[i].room - T[k].room > Size)
                {
                    L[i].startAdress = L[i].startAdress + T[k].room;
                    L[i].room = L[i].room - T[k].room;
                    L[i].state = 1;
                    T[k].startAdress = L[i].startAdress - T[k].room;
                    printf("内存分配成功!\n作业申请空间为%d\n起始地址为%d\n",
T[k].room, T[k].startAdress);
                    break;
                }
                //第二种情况：剩余空间小于最小分割大小
                else
                {
                    L[i].startAdress = L[i].startAdress + Size;
                    L[i].room = L[i].room - Size;
                    L[i].state = 2;
                    T[k].startAdress = L[i].startAdress - Size;
                    printf("内存分配成功!\n作业申请空间为%d\n起始地址为%d\n",
```

```
L[i].room, T[k].startAdress);
                        break;
                    }
                }
                //分区正好能放下该作业
                else if (L[i].state = 0 && L[i].room - T[k].room == Size)
                {
                    L[i].state = 2;
                    T[k].startAdress = L[i].startAdress;
                    printf("内存分配成功！\n 作业申请空间为%d\n 起始地址为%d\n",
T[k].room, T[k].startAdress);
                    break;
                }
            }
            if (i > n)
            {
                printf("作业请求内存空间过大，空闲分区表不能满足要求，内存分配失
败!\n");
                break;
            }
        }

        //回收内存
        else if (tmp == 2)
        {
            recycle();
            sort(L + 1, L + n + 1, cmp); //排序
        }
```

```
        //查看内存
        else if (tmp == 3)
            print();

        //退出
        else if (tmp == 0)
            return;
        else
        {
            printf("输入有误，请重新输入！\n");
            continue;
        }
    }
}

int main()
{
loop1:
    pos = 1;
    printf("请输入空闲分区表分区数量：\n");
    cin >> n;
    printf("请输入每个空闲分区的分区大小，分区始址\n");
    for (int i = 1; i <= n; i++)
    {
        printf("请输入第%d 个分区的信息：\n", i);
        cin >> L[i].room >> L[i].startAdress;
        L[i].id = i;
        L[i].state = 0;
    }
```

```
    printf("输入完毕，当前空闲分区表状态为：\n");
    print();
    printf("请输入不再切割的剩余空间的大小：\n");
    cin >> Size;
loop2:
    printf("选择内存分配的算法：1.循环首次适应算法 2.最佳适应算法 3.重新编辑空闲分区表 4.退出 \n");
    int tmp;
    cin >> tmp;
    if (tmp == 1)
        NF();
    else if (tmp == 2)
        BF();
    else if (tmp == 3)
    {
        goto loop1;
    }
    else if (tmp == 4)
        return 0;
    else
    {
        printf("输入有误，请重新输入!\n");
        goto loop2;
    }
}
```

运行结果

循环首次适应算法：

```
请输入空闲分区表分区数量：
5
请输入每个空闲分区的分区大小，分区始址
请输入第 1 个分区的信息：
50 85
请输入第 2 个分区的信息：
32 155
请输入第 3 个分区的信息：
70 275
请输入第 4 个分区的信息：
60 532
请输入第 5 个分区的信息：
10 980
输入完毕，当前空闲分区表状态为：
|-------------------------------------------------------|
|        分区号      分区始址      分区大小      分区状态        |
|-------------------------------------------------------|
|          1         85          50          0           |
|-------------------------------------------------------|
|          2         155         32          0           |
|-------------------------------------------------------|
|          3         275         70          0           |
|-------------------------------------------------------|
|          4         532         60          0           |
|-------------------------------------------------------|
|          5         980         10          0           |
|-------------------------------------------------------|
请输入不再切割的剩余空间的大小：
3
```

选择内存分配的算法：1.循环首次适应算法 2.最佳适应算法 3.重新编辑空闲分区表 4.退出

1

选择采用'循环首次适应算法'进行内存分配

```
    ******************************************
    1: 分配内存              2: 回收内存

    3: 查看空闲分区表        0: 退出
    ******************************************
```

请输入您的操作 ： 1

请输入第 1 个作业占用空间大小：

25

内存分配成功!

作业申请空间为 25

起始地址为 85

```
    ******************************************
    1: 分配内存              2: 回收内存

    3: 查看空闲分区表        0: 退出
    ******************************************
```

请输入您的操作 ： 3

```
|-------------------------------------------------------------|
|          分区号      分区始址      分区大小      分区状态          |
|-------------------------------------------------------------|
|           1          110           25           1            |
|-------------------------------------------------------------|
|           2          155           32           0            |
|-------------------------------------------------------------|
```

```
|          3          275          70          0          |
|-------------------------------------------------------|
|          4          532          60          0          |
|-------------------------------------------------------|
|          5          980          10          0          |
|-------------------------------------------------------|
     ****************************************
     1: 分配内存              2: 回收内存

     3: 查看空闲分区表        0: 退出
     ****************************************
请输入您的操作 ： 2
请输入要释放占用空间任务的序号：
1
内存回收完毕！
     ****************************************
     1: 分配内存              2: 回收内存

     3: 查看空闲分区表        0: 退出
     ****************************************
请输入您的操作 ： 3
|-------------------------------------------------------|
|          分区号      分区始址      分区大小      分区状态          |
|-------------------------------------------------------|
|          1          85          50          1          |
|-------------------------------------------------------|
|          2          155          32          0          |
|-------------------------------------------------------|
|          3          275          70          0          |
```

```
|-------------------------------------------------------------|
|          4          532          60          0             |
|-------------------------------------------------------------|
|          5          980          10          0             |
|-------------------------------------------------------------|
     ******************************************
     1: 分配内存              2: 回收内存


     3: 查看空闲分区表        0: 退出
     ******************************************
请输入您的操作 ： 0
```

最佳适应算法：

```
请输入空闲分区表分区数量：
5
请输入每个空闲分区的分区大小，分区始址
请输入第 1 个分区的信息：
50 85
请输入第 2 个分区的信息：
32 155
请输入第 3 个分区的信息：
70 275
请输入第 4 个分区的信息：
60 532
请输入第 5 个分区的信息：
10 980
输入完毕，当前空闲分区表状态为：
|-------------------------------------------------------------|
|          分区号       分区始址       分区大小       分区状态          |
```

```
|-----------------------------------------------------------|
|          1          85          50          0            |
|-----------------------------------------------------------|
|          2          155         32          0            |
|-----------------------------------------------------------|
|          3          275         70          0            |
|-----------------------------------------------------------|
|          4          532         60          0            |
|-----------------------------------------------------------|
|          5          980         10          0            |
|-----------------------------------------------------------|
请输入不再切割的剩余空间的大小：
3
选择内存分配的算法：1.循环首次适应算法  2.最佳适应算法  3.重新编辑空闲分区表  4.退出
2
选择采用'最佳适应算法'进行内存分配


     ******************************************
     1: 分配内存              2: 回收内存


     3: 查看空闲分区表        0: 退出
     ******************************************
请输入您的操作 ： 1
请输入第 1 个作业占用空间大小：
25
内存分配成功!
作业申请空间为 25
起始地址为 155
```

```
    ****************************************
    1: 分配内存              2: 回收内存

    3: 查看空闲分区表        0: 退出
    ****************************************
请输入您的操作 ： 3
|-----------------------------------------------------------|
|         分区号      分区始址      分区大小      分区状态          |
|-----------------------------------------------------------|
|          5          980           10            0           |
|-----------------------------------------------------------|
|          2          180           7             1           |
|-----------------------------------------------------------|
|          1          85            50            0           |
|-----------------------------------------------------------|
|          4          532           60            0           |
|-----------------------------------------------------------|
|          3          275           70            0           |
|-----------------------------------------------------------|
    ****************************************
    1: 分配内存              2: 回收内存

    3: 查看空闲分区表        0: 退出
    ****************************************
请输入您的操作 ： 2
请输入要释放占用空间任务的序号：
1
内存回收完毕！
    ****************************************
```

```
    1: 分配内存              2: 回收内存


    3: 查看空闲分区表        0: 退出
    ****************************************
请输入您的操作 ： 3
|-----------------------------------------------------------|
|         分区号      分区始址      分区大小      分区状态          |
|-----------------------------------------------------------|
|          5         980          10           0            |
|-----------------------------------------------------------|
|          2         155          32           1            |
|-----------------------------------------------------------|
|          1         85           50           0            |
|-----------------------------------------------------------|
|          4         532          60           0            |
|-----------------------------------------------------------|
|          3         275          70           0            |
|-----------------------------------------------------------|
    ****************************************
    1: 分配内存              2: 回收内存


    3: 查看空闲分区表        0: 退出
    ****************************************
请输入您的操作 ： 0
```

结论展望

可变式分区管理带来的最大好处就是更加高效地利用了分区，但也带来了不小的问题--容易产生细小的不可利用的内存碎片。怎么去解决这个问题，又是留给未来的一道难题。

实验五：分页存储管理

实验题目

模拟分页式存储管理中硬件的地址转换和产生缺页中断，然后分别用 LRU、FIFO、改进型的 CLOCK 算法实现分页管理的缺页中断。

实验要求

1. 显示每个页面在内存中的绝对地址，页表信息、列出缺页情况等。

需求分析

操作系统中往往有多个页面，但是物理块却是有大小的，不可能存下所有页面。所以，分页存储管理的存在就很有必要了，通过不同的页面置换策略来让这种替换页面的开销达到最低。

数据结构

```
int n;          //页面引用号个数
int m;          //物理块数目
int page[N];    //页号
int block[N]; //物理块，内存


//改进的 clock
bool changed[N]; //修改位
bool visited[N]; //访问位
```

算法思想

LRU（最近最久未使用算法）：

遍历各个页面，分三种情况

- 页面在内存中：其余的后移，把它移到内存中的第一个位置。
- 页面不在内存中且内存未满：其他的后移，然后把它插入第一个位置。
- 页面不在内存中且内存已满：其他的后移，把它放入第一个位置。

FIFO（先进先出算法）：

和 LRU 类似，唯一的不同点在于当页面在内存中的时候不需要把它放入第一个位置。

改进的 Clock：

1 类(访问位=0, 修改位=0)：该页最近既未被访问，又未被修改，是最佳淘汰页。

2 类(访问位=0, 修改位=1)：该页最近未被访问，但已被修改，并不是很好的淘汰页。

3 类(访问位=1, 修改位=0)：最近已被访问，但未被修改，该页有可能再被访问。

4 类(访问位=1, 修改位=1)：最近已被访问且被修改，该页可能再被访问。

首先要有几个函数，

第一个是用随机数来判断页面是否被修改。

第二个就是检测页号是否在内存中并把访问位置 1，并且根据上一个函数把修改位置 1 或 0。

第三个就是寻找 1 类和 2 类页面，并返回他们的位置。也是与朴素的 clock 算法不同的地方。

这里分三步：

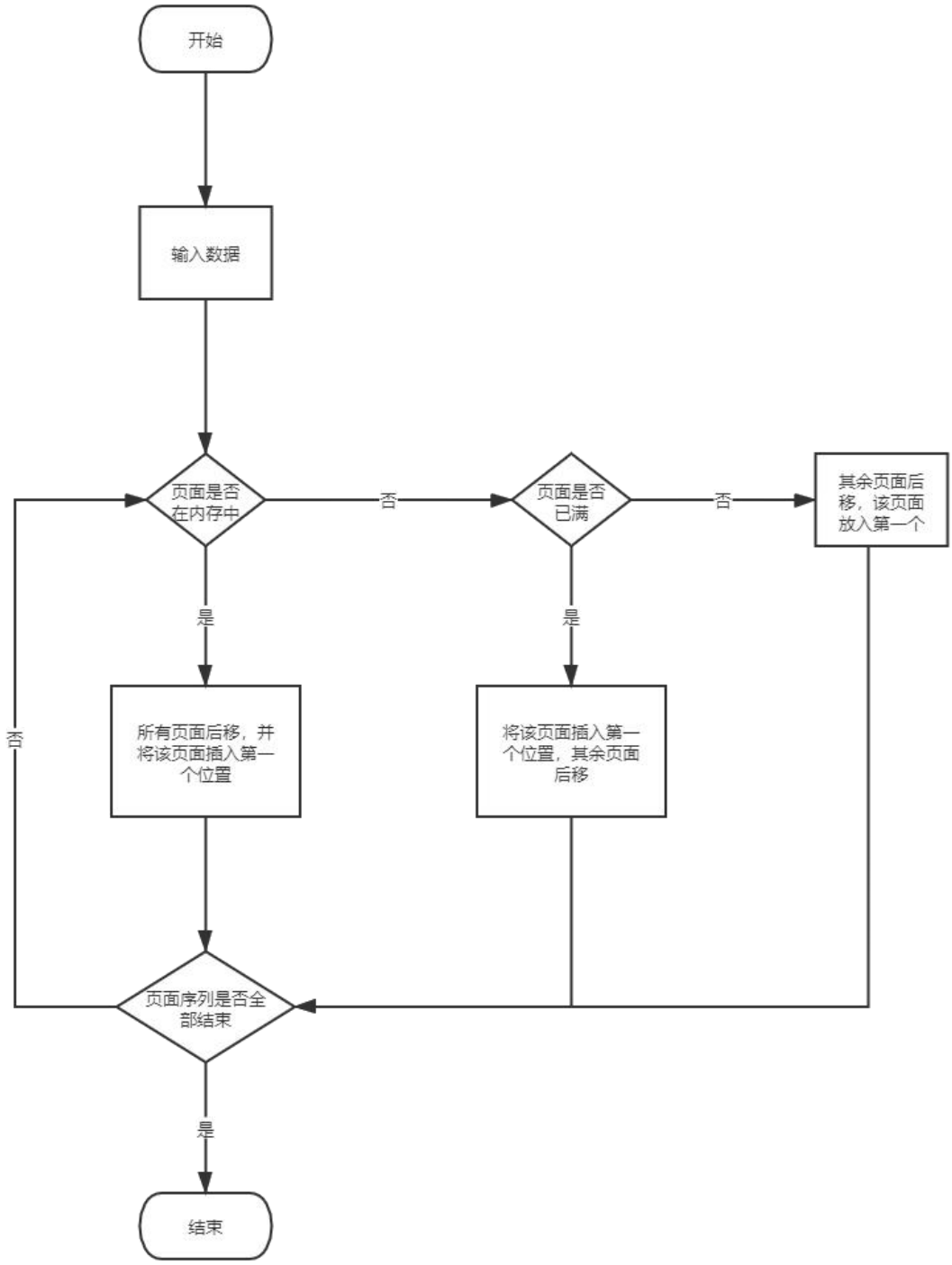
- 选择最佳淘汰页面，从指针当前位置开始,扫描循环队列，寻找第一类页面，将所遇到的第一个页面作为所选中的淘汰页。在第一次扫描期间不改变访问位。
- 如果第一步失败，即查找一周后未遇到第一类页面，则开始第二轮扫描，寻找第二类页面，将所遇到的第一个这类页面作为淘汰页。在第二轮扫描期间，将所有扫描过的页面的访问位都置 0。
- 如果第二步也失败，亦即未找到第二类页面，则将指针返回到开始的位置，并将所有的访问位复 0。然后重复第一步，如果仍失败，必要时再重复第二步，此时就一定能找到被淘汰的页。

接着们就可以开始我们的改进的 clock 算法了，这里同样分三类：

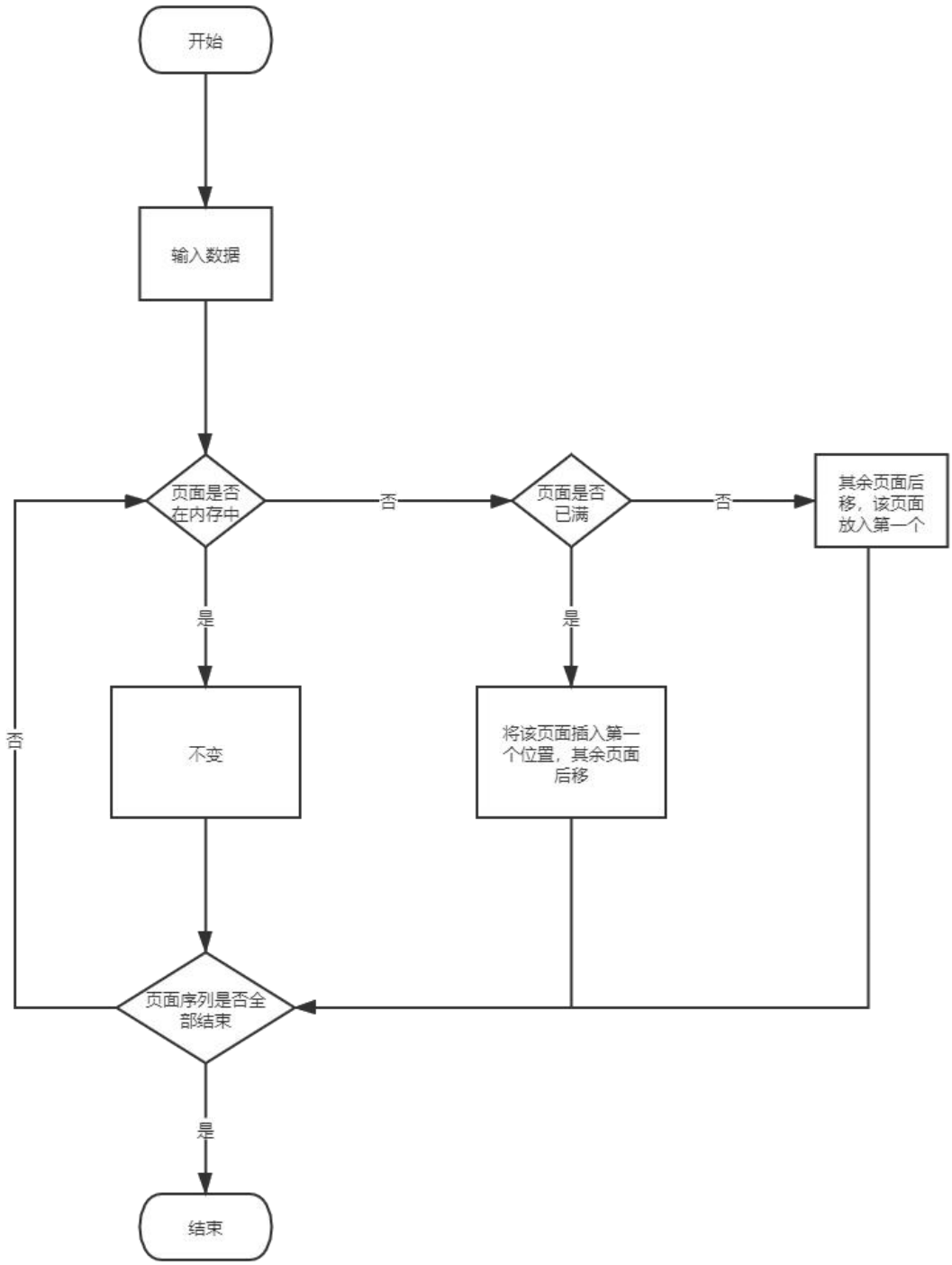
- 页面在内存中：不变
- 页面不在内存中且内存未满：用第三个函数找最佳替换页并替换。
- 页面不在内存中且内存已满：直接插入内存中去。

算法流程

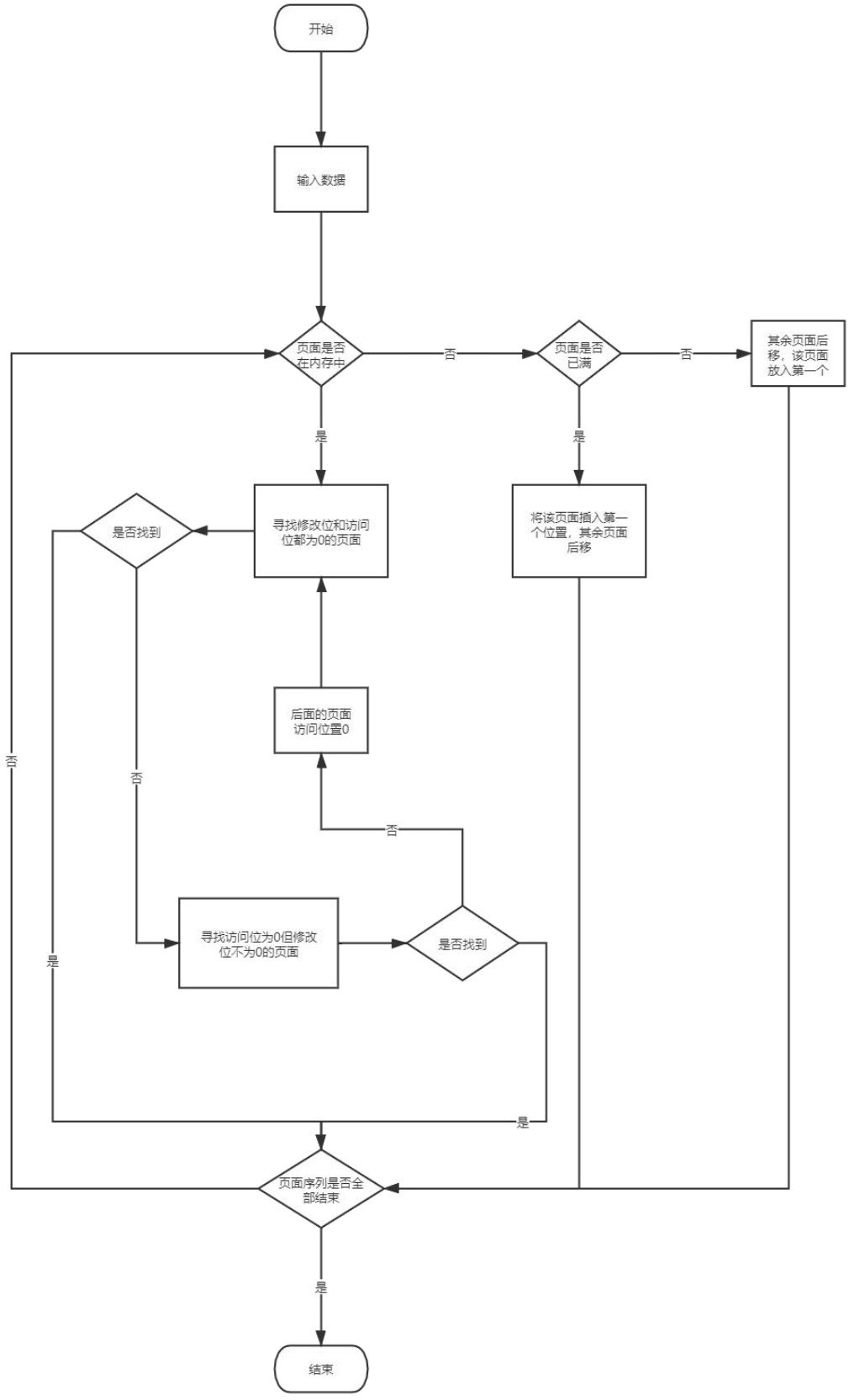
LRU：


开始
输入数据
页面是否在内存中
否
页面是否已满
否
其余页面后移，该页面放入第一个
是
是
所有页面后移，并将该页面插入第一个位置
将该页面插入第一个位置，其余页面后移
否
页面序列是否全部结束
是
结束

FIFO：


开始
输入数据
页面是否在内存中
否
页面是否已满
否
其余页面后移，该页面放入第一个
是
是
否
不变
将该页面插入第一个位置，其余页面后移
页面序列是否全部结束
是
结束

改进的 Clock：


开始
输入数据
页面是否在内存中
否
页面是否已满
否
其余页面后移，该页面放入第一个
是
是
寻找修改位和访问位都为0的页面
是否找到
将该页面插入第一个位置，其余页面后移
后面的页面访问位置0
否
否
寻找访问位为0但修改位不为0的页面
是否找到
否
是
是
页面序列是否全部结束
是
结束

这里我发现我 LRU 和 FIFO 多判断了，里头几个操作其实可以合并。不过对结果没有影响，所以就不改了。

算法实现

```
/*
测试数据:
20 3
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

12 3
2 3 2 1 5 2 4 5 3 2 5 2
*/
#include <iostream>
#include <set>
#include <algorithm>
#include <cstring>
#include <cmath>
#define N 200
using namespace std;

int n;          //页面引用号个数
int m;          //物理块数目
int page[N];    //页号
int block[N]; //物理块，内存

//最近最久未使用算法
void lru()
{
    //初始化
```

```
int page_lack = 0;
memset(block, -1, sizeof(block));

//循环
for (int i = 1; i <= n; ++i)
{
    printf("当前需访问的页面为%d\n", page[i]);
    int j;
    int max_dist = 0, p;

    for (j = 1; j <= m; ++j)
    {
        //页面在内存中,放到第一个，其余后移
        if (block[j] == page[i])
        {
            printf("%d 已在内存中，提升到第一个\n", page[i]);
            int tmp = block[j];
            for (int tp = j; tp >= 2; --tp)
                block[tp] = block[tp - 1];
            block[1] = tmp;
            break;
        }
        //不在内存，插入第一个，其他后移
        else if (block[j] == -1)
        {
            for (int tp = j; tp >= 2; --tp)
                block[tp] = block[tp - 1];
            block[1] = page[i];
            printf(" %d 不在内存，放入\n", page[i]);
```

```
                page_lack++;
                break;
            }
        }

        //页面不在内存中-页面替换，其他右移，插入第一个
        if (j > m)
        {
            printf("%d 不在内存，放入，页面%d 被替换\n", page[i], block[m]);
            for (int tp = m; tp >= 2; --tp)
                block[tp] = block[tp - 1];
            block[1] = page[i];
            page_lack++;
        }

        //输出内存中页面加载情况
        for (int k = 1; k <= m; ++k)
            cout << block[k] << "\t";
        cout << endl
             << endl;
    }

    //输出缺页次数
    cout << endl
         << "缺页次数：" << page_lack << endl;
}

//先进先出页面置换算法
void fifo()
```

```
{
    //初始化
    int page_lack = 0;
    memset(block, -1, sizeof(block));

    //循环
    for (int i = 1; i <= n; ++i)
    {
        printf("当前需访问的页面为%d\n", page[i]);
        int j;
        int max_dist = 0, p;

        for (j = 1; j <= m; ++j)
        {
            //页面在内存中
            if (block[j] == page[i])
            {
                printf("%d 已在内存中，不变\n", page[i]);
                break;
            }
            //不在内存，插入第一个，其他后移
            else if (block[j] == -1)
            {
                for (int tp = j; tp >= 2; --tp)
                    block[tp] = block[tp - 1];
                block[1] = page[i];
                printf("%d 不在内存，放入\n",  page[i]);
                page_lack++;
                break;
```

```
            }
        }

        //页面不再内存中-页面替换，其他右移，插入第一个
        if (j > m)
        {
            printf("%d 不在内存，放入，页面%d 被替换\n", page[i], block[m]);
            for (int tp = m; tp >= 2; --tp)
                block[tp] = block[tp - 1];
            block[1] = page[i];
            page_lack++;
        }

        //输出内存中页面加载情况
        for (int k = 1; k <= m; ++k)
            cout << block[k] << "\t";
        cout << endl
             << endl;
    }

    //输出缺页次数
    cout << endl
         << "缺页次数：" << page_lack << endl;
}

 bool changed[N]; //修改位
 bool visited[N]; //访问位

//判断页面是否已经被修改
```

```
bool change()
{
    if (rand() % 2)
    {
        printf("该页面被修改!\n");
        return true;
    }
    else
        return false;
}

//检测页号是否在内存中并把访问位置 1，修改位根据情况置 1 或 0
bool inblock(int num)
{
    for (int i = 1; i <= m; i++)
    {
        if (page[num] == block[i])
        {
            if (change())
            {
                changed[i] = 1;
                visited[i] = 1;
            }
            else
            {
                visited[i] = 1;
                changed[i] = 0;
            }
            return true;
```

```
        }
    }
    return false;
}

int Search()
{
    int j;
    for (j = 1; j <= m; j++)
    {
        //1 类：最佳淘汰页
        if (visited[j] == 0 && changed[j] == 0)
            return j;
    }
    for (j = 1; j <= m; j++)
    {
        //2 类
        if (visited[j] == 0 && changed[j] == 1)
            return j;
        visited[j] == 0;
    }
    for (j = 1; j <= m; j++)
    {
        visited[j] = 0;
    }
    return Search();
}

void improved_Clock()
```

```
{
    memset(block, -1, sizeof(block));
    memset(visited, 0, sizeof(visited));
    memset(changed, 0, sizeof(changed));
    int j;
    int page_lack = 0;
    for (int i = 1; i <= n; i++)
    {
        printf("当前需访问的页面为%d\n", page[i]);
        if (inblock(i)) ///页面已存在于内存中
        {
            printf("%d 已在内存中，不变\n",page[i]);
        }

        //页面未存在于内存中并且内存已满
        else if (block[m] != -1)
        {
            j = Search();
            block[j] = page[i];
            visited[j] = 1;
            page_lack++;
        }

        //内存未满，并且页面未存在于内存中
        else
        {
            for (int k = 1; k <= m; k++)
            {
                if(block[k] == -1)
```

```
                {
                    block[k] = page[i];
                    page_lack++;
                    break;
                }
            }
        }
        for (int k = 1; k <= m; ++k)
            cout << block[k] << "\t";
        cout << endl
             << endl;
    }

    //输出缺页次数
    cout << endl
         << "缺页次数：" << page_lack << endl;
}

int main()
{
    cin >> n >> m;
    for (int i = 1; i <= n; ++i)
    {
        cin >> page[i];
    }

    cout << "------------------------------------------------------------" << endl;
    cout << "LRU 算法:\n";
    lru();
```

```
    cout << "------------------------------------------------------------" << endl;
    cout << "FIFO 算法:\n";
    fifo();
    cout << "------------------------------------------------------------" << endl;
    cout << "改进的 CLOCK 算法:\n";
    improved_Clock();
    cout << "------------------------------------------------------------" << endl;
    return 0;
}
```

运行结果

```
12 3
2 3 2 1 5 2 4 5 3 2 5 2
------------------------------------------------------------
LRU 算法:
当前需访问的页面为 2
 2 不在内存，放入
2        -1      -1

当前需访问的页面为 3
 3 不在内存，放入
3        2       -1

当前需访问的页面为 2
2 已在内存中，提升到第一个
2        3       -1

当前需访问的页面为 1
```

1 不在内存，放入

1　　2　　3

当前需访问的页面为 5

5 不在内存，放入，页面 3 被替换

5　　1　　2

当前需访问的页面为 2

2 已在内存中，提升到第一个

2　　5　　1

当前需访问的页面为 4

4 不在内存，放入，页面 1 被替换

4　　2　　5

当前需访问的页面为 5

5 已在内存中，提升到第一个

5　　4　　2

当前需访问的页面为 3

3 不在内存，放入，页面 2 被替换

3　　5　　4

当前需访问的页面为 2

2 不在内存，放入，页面 4 被替换

2　　3　　5

当前需访问的页面为 5

5 已在内存中，提升到第一个

```
5        2        3

当前需访问的页面为 2
2 已在内存中，提升到第一个
2        5        3


缺页次数：7
------------------------------------------------------------
FIFO 算法:
当前需访问的页面为 2
2 不在内存，放入
2        -1       -1

当前需访问的页面为 3
3 不在内存，放入
3        2        -1

当前需访问的页面为 2
2 已在内存中，不变
3        2        -1

当前需访问的页面为 1
1 不在内存，放入
1        3        2

当前需访问的页面为 5
5 不在内存，放入，页面 2 被替换
5        1        3
```

当前需访问的页面为 2

2 不在内存，放入，页面 3 被替换

2　　5　　1

当前需访问的页面为 4

4 不在内存，放入，页面 1 被替换

4　　2　　5

当前需访问的页面为 5

5 已在内存中，不变

4　　2　　5

当前需访问的页面为 3

3 不在内存，放入，页面 5 被替换

3　　4　　2

当前需访问的页面为 2

2 已在内存中，不变

3　　4　　2

当前需访问的页面为 5

5 不在内存，放入，页面 2 被替换

5　　3　　4

当前需访问的页面为 2

2 不在内存，放入，页面 4 被替换

2　　5　　3

缺页次数：9

改进的 CLOCK 算法:

当前需访问的页面为 2

2 -1 -1

当前需访问的页面为 3

2 3 -1

当前需访问的页面为 2

该页面被修改!

2 已在内存中，不变

2 3 -1

当前需访问的页面为 1

2 3 1

当前需访问的页面为 5

2 5 1

当前需访问的页面为 2

该页面被修改!

2 已在内存中，不变

2 5 1

当前需访问的页面为 4

2 5 4

```
当前需访问的页面为 5
5 已在内存中，不变
2       5       4

当前需访问的页面为 3
2       3       4

当前需访问的页面为 2
2 已在内存中，不变
2       3       4

当前需访问的页面为 5
2       3       5

当前需访问的页面为 2
该页面被修改!
2 已在内存中，不变
2       3       5



缺页次数：7
-----------------------------------------------------------------
```

结论展望

分页存储管理解决了如何让有限的内存物理块去覆盖最多的页面的方法--页面置换，但如何去减少置换带来的开销仍然是个问题。

二、实训项目：Linux 系统、Windows 系统

实验二：进程同步模拟

实验内容

实验二：利用 Windows 下的 VC++或者 Linux 下的 C 或 C++编程模拟解决各种进程同步问题

1) 生产者-消费者问题；读者优先的读者-写者问题；写者优先的读者-写者问题；哲学家就餐问题；

2) 撰写实验报告，阐述实验目的、目标、开发/运行/测试环境、实验步骤、技术难点及解决方案、关键数据结构和算法流程、编译运行测试过程及结果截图、结论与体会等；

3) 要求提交实验报告、源程序及可执行程序；

4) 提交材料，可以是动画或视频演示。

实验目的/目标

- 了解生产者-消费者问题、读者优先的读者-写者问题、写者优先的读者-写者问题和哲学家就餐问题的描述；
- 了解这四个问题的解决方式；
- 通过代码模拟这四个问题的进程同步问题的解决。

开发/运行/测试环境

平台：VS code 1.51.1

语言：C/C++

环境：Windows 10 202H

实验步骤

首先，查阅资料

去了解各个问题具体是什么，现阶段有哪些解决方案，以及一起其他与之相关的资料。

其次，分析措施

通过这些方案，我们去分析如何去用代码实现这些方案。

接着，代码实现

参考他人的代码、加上自己的理解来完成对这些问题的具体的模拟和解决的实现。

最后，结果分析

看一下结果是否符合预期，如果不符，是出现了什么问题。

关键数据结构

```
//生产者-消费者问题
int in = 0, out = 0; //代表执行生产和消费的变量
int flg_pro = 0, flg_con = 0; //线程结束的标志
int mutex = 1; //互斥信号量，实现线程对缓冲池的互斥访问；
int empty = N, full = 0; //资源信号量，分别表示缓冲池中空缓冲池和满缓冲池的数量(注意初值，从生产者的角度)

//读者优先的读者-写者问题
sem_t wrt, mutex; //互斥信号量
int readCount; //正在读的读者数量

//写者优先的读者-写者问题
```

```
sem_t RWMutex, mutex1, mutex2, mutex3, wrt; //互斥信号量
int writeCount, readCount; //等待写的写者数量，正在读的读者数量


//哲学家就餐问题
pthread_mutex_t chopstick[6]; //筷子作为 mutex
```

问题解析

生产者-消费者问题：

生产者-消费者问题是经典的线程同步问题，主要有三点：

- 不能互斥访问共享资源(不能同时访问共享数据)；
- 当公共容器满时，生产者不能继续生产(生产者应阻塞并唤醒消费者消费)；
- 当公共容器为空时，消费者不能继续消费(消费者应阻塞并唤醒生产者生产)。

其中，第一点是其同步关系表现，第二点是其互斥关系的表现。

读者优先的读者-写者问题：

即使写者发出了请求写的信号，但是只要还有读者在读取内容，就还允许其他读者继续读取内容，直到所有读者结束读取，才真正开始写。

- 有读者在读后面来的读者可以直接进入临界区，而已经在等待的写者继续等待直到没有任何一个读者时。
- 读者之间不互斥，写者之间互斥，只能一个写，可以多个读，
- 读者写者之间互斥，有写者写则不能有读者读
- 如果在读访问非常频繁的场合，有可能造成写进程一直无法访问文件的局面

写者优先的读者-写者问题：

如果有写者申请写文件，在申请之前已经开始读取文件的可以继续读取，但是如果再有读者申请读取文件，则不能够读取，只有在所有的写者写完之后才可以读取。

- 写者线程的优先级高于读者线程。
- 当有写者到来时应该阻塞读者线程的队列。
- 当有一个写者正在写时或在阻塞队列时应当阻塞读者进程的读操作，直到所有写者进程完成写操作时放开读者进程。
- 当没有写者进程时读者进程应该能够同时读取文件。

哲学家就餐问题：

5 个哲学家 5 只筷子，他们只做两件事---吃饭和思考。

- 吃饭的时候，必须要两只筷子
- 思考的时候不需要筷子

每个哲学家先拿左手边，再拿右手边，如果右边筷子被拿走放下左手的筷子。

算法思想

生产者-消费者问题：

我们定义了一个互斥信号量 `mutex` 来实现线程对缓冲池的互斥访问，也就是说生产者和消费者不能互斥地去访问资源。

判断如果缓冲池中没有资源，则消费者无法消费，如果缓冲池中资源已满，则生产者无法生产。

其中有几个注意点：

1）在每一个线程中，用来实现互斥的 `wait(&mutex);`和 `signal(&mutex);`必须成对出现。而对于资源信号量的 `wait` 和 `signal` 操作，分别成对出现在不同的线程中。

2）先执行对资源信号量的 wait 操作，在执行对互斥信号量的 wait 操作，不然会产生死锁。

读者优先的读者-写者问题：

用一个 int 型 read_count 变量来记录当前的读者数目，用于确定是否需要释放正在等待的写者线程，当 read_count=0 时，表明所有的读者读完，需要释放写者等待队列中的一个写者。

每一个读者开始读文件时，必须修改 read_count 变量。因此我增加了一个互斥对象 mutex 来实现对全局变量 read_count 修改时的互斥。

为了实现写-写互斥，又增加一个临界区对象 write。当写者发出写请求时，必须申请临界区对象的所有权。通过这种方法，也可以实现读-写互斥，当 read_count=1 时(即第一个读者到来时)，读者线程也必须申请临界区对象的所有权。

当读者拥有临界区的所有权时，写者阻塞在临界区对象 write 上。当写者拥有临界区的所有权时，第一个读者判断完“read_count==1”后阻塞在 write 上，其余的读者由于等待对 read_count 的判断，阻塞在 mutex 上。

写者优先的读者-写者问题：

我们将上面读者优先的读者写者问题里头的 read_count 改为 write_count，用来记录当前正在等待的写者数目。同样的，当 write_count=0 时，才可以释放等待的读者线程队列。

为了对全局变量 write_count 实现互斥，我增加一个互斥对象 mutex2。

为了实现写者优先，添加一个临界区对象 read，当有写者在写文件或等待时，读者必须阻塞在 read 上。

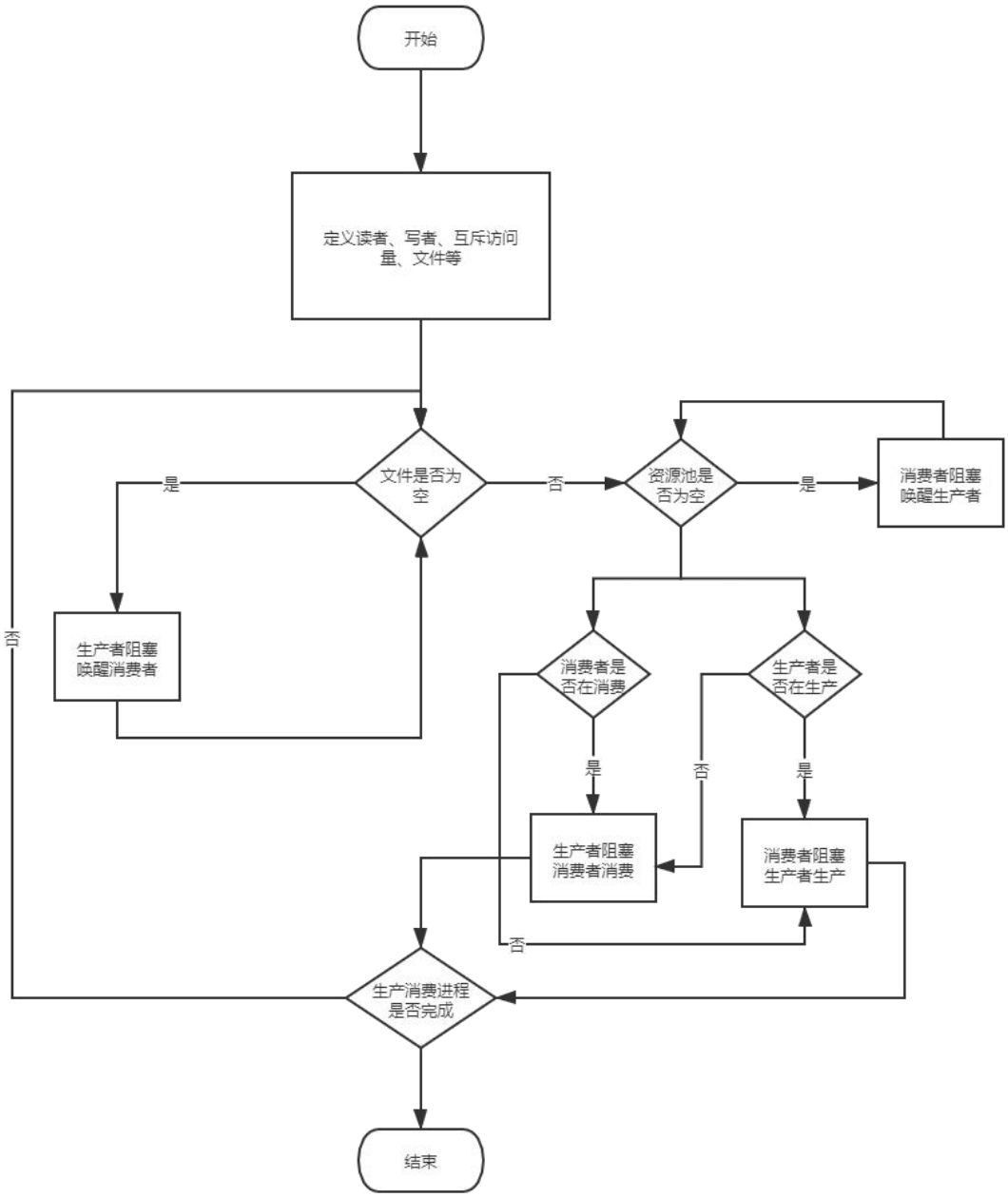
同样，有读者读时，写者必须等待。于是，必须有一个互斥对象 RW_mutex 来实现这个互斥。有写者在写时，写者必须等待。读者线程要对全局变量 read_count 实现操作上的互斥，必须有一个互斥对象命名为 mutex1。

哲学家就餐问题：

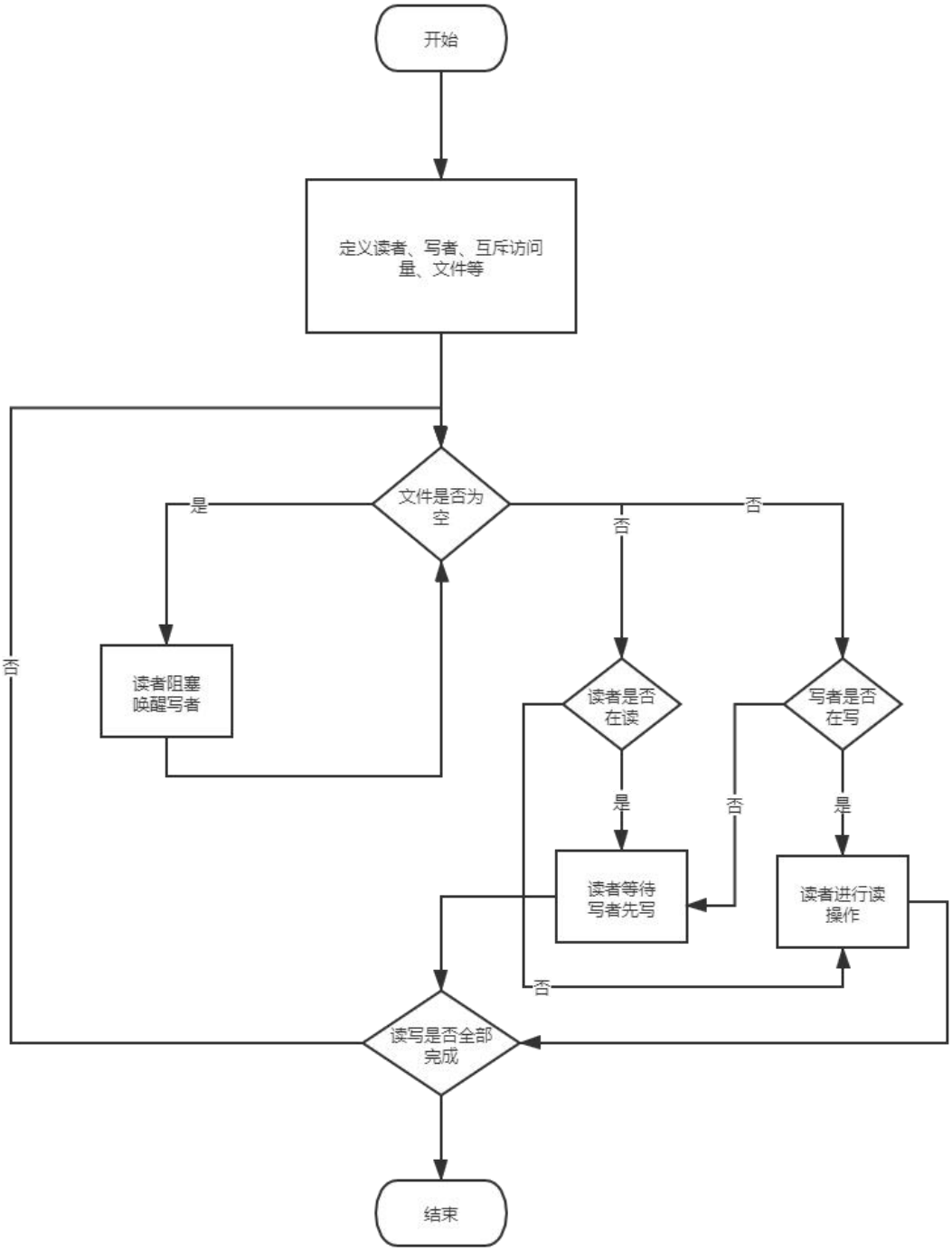
我们把筷子作为 mutex，哲学家每次先拿左手边的筷子，如果没有，则等待。如果有则继续拿右边的筷子，如果右边的筷子被拿走则放下左边的筷子并等待。

算法流程

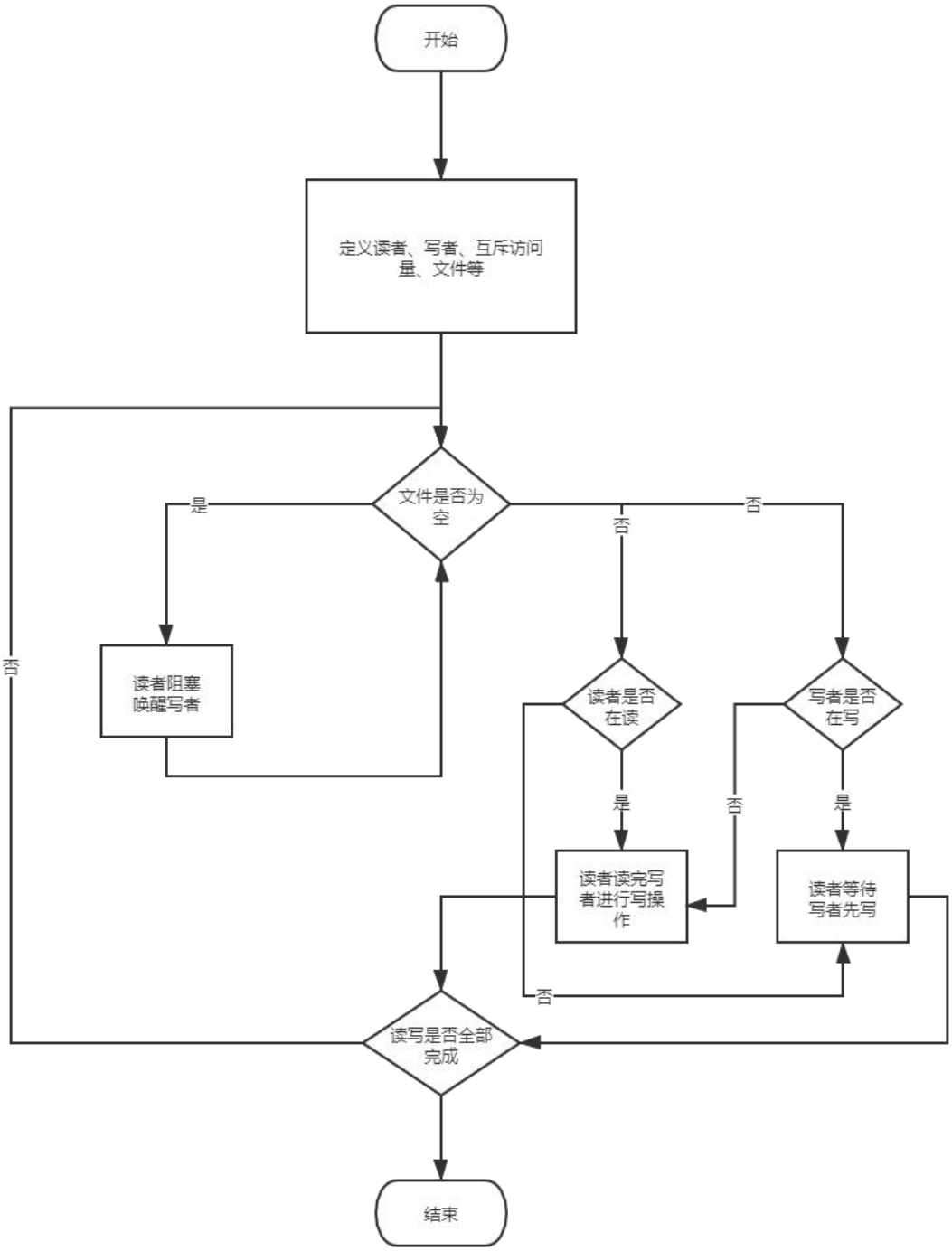
生产者-消费者问题：



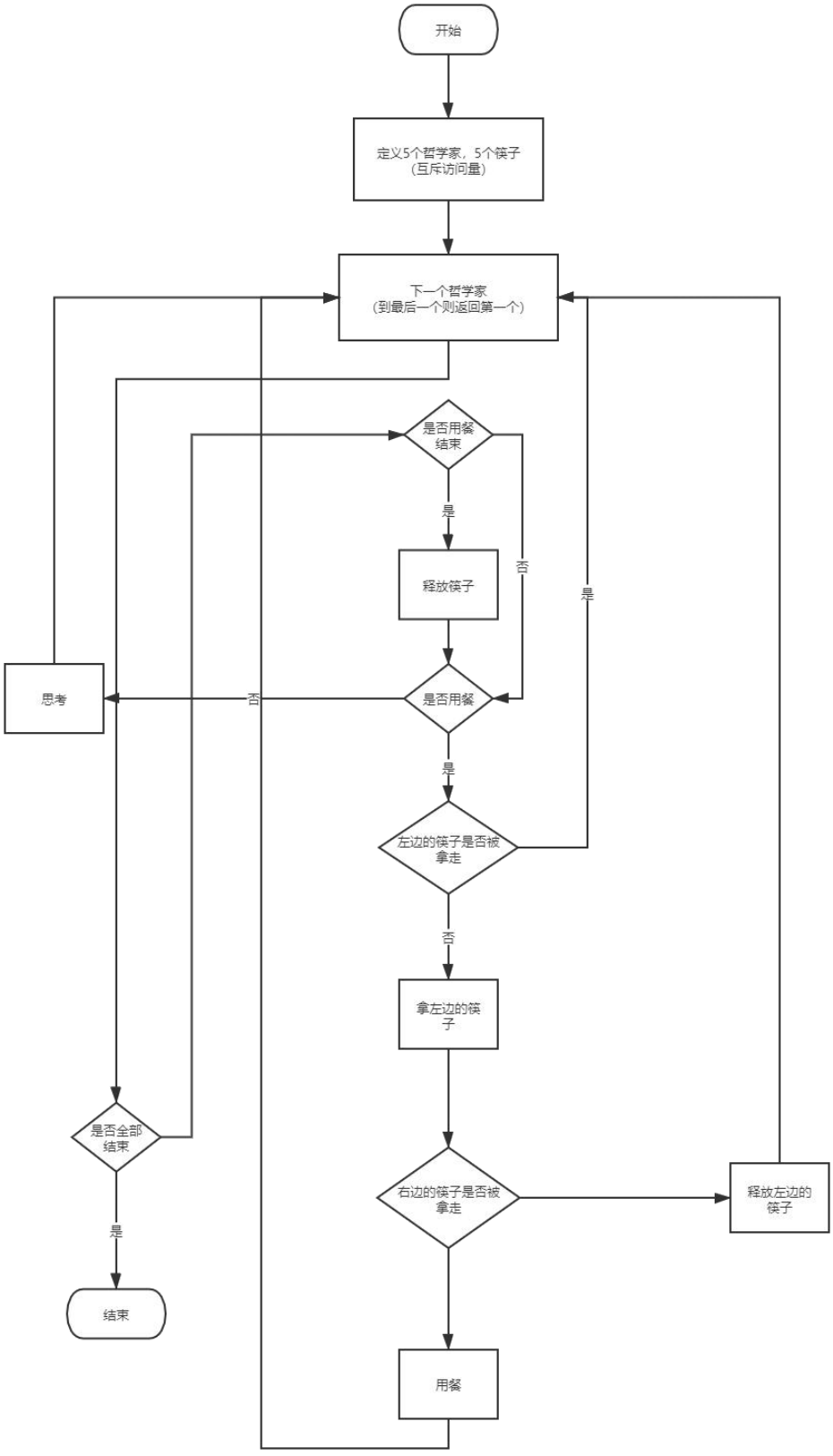
读者优先的读者-写者问题：



写者优先的读者-写者问题：



哲学家就餐问题：


开始
定义5个哲学家，5个筷子
（互斥访问量）
下一个哲学家
（到最后一个则返回第一个）
是否用餐
结束
是
释放筷子
否
是
思考
否
是否用餐
是
左边的筷子是否被
拿走
否
拿左边的筷
子
是否全部
结束
右边的筷子是否被
拿走
释放左边的
筷子
是
结束
用餐

算法实现

生产者-消费者问题：

```
#include <stdio.h>
#include <process.h>
#define N 10

//代表执行生产和消费的变量
int in = 0, out = 0;

//线程结束的标志
int flg_pro = 0, flg_con = 0;

//mutex：互斥信号量，实现线程对缓冲池的互斥访问；
//empty 和 full：资源信号量，分别表示缓冲池中空缓冲池和满缓冲池的数量(注意初值，从生产者的角度)
int mutex = 1, empty = N, full = 0;

//打印测试
void print(char c)
{
    printf("%c    一共生产了%d 个资源，消费了%d 个资源，现在有%d 个资源\n", c, in, out, full);
}

//请求某个资源
void wait(int *x)
{
    while ((*x) <= 0)
```

```
        ;
    (*x)--;
}

//释放某个资源
void signal(int *x)
{
    (*x)++;
}

//生产者
void produce(void *a)
{
    while (1)
    {
        printf("开始阻塞生产者\n");
        wait(&empty); //申请一个缓冲区，看有无其他线程访问
        wait(&mutex);
        printf("结束阻塞生产者\n");

        in++;

        signal(&mutex);
        signal(&full); //full 加一，唤醒消费者，告诉消费者可以消费

        printf("结束生产。。。\n");
        print('p');

        Sleep(200);
```

```
        if (flg_pro == 1)
        {
            _endthread();
        }
    }
}

//消费者
void consumer(void *a)
{
    while (1)
    {
        printf("开始阻塞消费者\n");
        wait(&full);
        wait(&mutex);
        printf("结束阻塞消费者\n");

        out++;

        signal(&mutex);
        signal(&empty);
        printf("结束消费。。。\n");
        print('c');

        Sleep(200);
        if (flg_con == 1)
        {
            _endthread();
        }
```

```
    }
}

//主函数
int main()
{
    _beginthread(consumer, 0, NULL);
    _beginthread(produce, 0, NULL);
    //总的执行时间为 1 分钟
    Sleep(10000);
    flg_pro = flg_con = 1;
    system("pause");
    return 0;
}
```

读者优先的读者-写者问题：

```
/*
读者优先
测试数据：
1 R 3 5
2 W 4 5
3 R 5 2
4 R 6 5
5 W 7 3
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```c
#include <sys/types.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>
#include <unistd.h>

//semaphores
sem_t wrt, mutex;
int readCount;

struct data
{
    int id;
    int opTime;
    int lastTime;
};

//读者
void *Reader(void *param)
{
    int id = ((struct data *)param)->id;
    int lastTime = ((struct data *)param)->lastTime;
    int opTime = ((struct data *)param)->opTime;

    sleep(opTime);
    printf("线程 %d: 等待读\n", id);
    sem_wait(&mutex);
    readCount++;
    if (readCount == 1)
```

```
        sem_wait(&wrt);
    sem_post(&mutex);

    printf("线程 %d: 开始读\n", id);
    /* reading is performed */
    sleep(lastTime);
    printf("线程 %d: 结束读\n", id);

    sem_wait(&mutex);
    readCount--;
    if (readCount == 0)
        sem_post(&wrt);
    sem_post(&mutex);
    pthread_exit(0);
}

//写者
void *Writer(void *param)
{
    int id = ((struct data *)param)->id;
    int lastTime = ((struct data *)param)->lastTime;
    int opTime = ((struct data *)param)->opTime;

    sleep(opTime);
    printf("线程 %d: 等待写\n", id);
    sem_wait(&wrt);

    printf("线程 %d: 开始写\n", id);
    /* writing is performed */
```

```
    sleep(lastTime);
    printf("线程 %d: 结束写\n", id);

    sem_post(&wrt);
    pthread_exit(0);
}

int main()
{
    //pthread
    pthread_t tid; // the thread identifier

    pthread_attr_t attr; //set of thread attributes

    /* get the default attributes */
    pthread_attr_init(&attr);

    //initial the semaphores
    sem_init(&mutex, 0, 1);
    sem_init(&wrt, 0, 1);
    readCount = 0;

    int id = 0;
    while (scanf("%d", &id) != EOF)
    {

        char role;      //producer or consumer
        int opTime;     //operating time
        int lastTime; //run time
```

```
        scanf("%c%d%d", &role, &opTime, &lastTime);
        struct data *d = (struct data *)malloc(sizeof(struct data));

        d->id = id;
        d->opTime = opTime;
        d->lastTime = lastTime;

        if (role == 'R')
        {
            printf("创建 %d 线程: 读者\n", id);
            pthread_create(&tid, &attr, Reader, d);
        }
        else if (role == 'W')
        {
            printf("创建 %d 线程: 写者\n", id);
            pthread_create(&tid, &attr, Writer, d);
        }
    }

    //信号量销毁
    sem_destroy(&mutex);
    sem_destroy(&wrt);

    return 0;
}
```

写者优先的读者-写者问题：

```
/*
写者优先
测试数据：
1 R 3 5
2 W 4 5
3 R 5 2
4 R 6 5
5 W 7 3
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/types.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>
#include <unistd.h>

//semaphores
sem_t RWMutex, mutex1, mutex2, mutex3, wrt;
int writeCount, readCount;

struct data
{
    int id;
    int opTime;
    int lastTime;
```

```
};

//读者
void *Reader(void *param)
{
    int id = ((struct data *)param)->id;
    int lastTime = ((struct data *)param)->lastTime;
    int opTime = ((struct data *)param)->opTime;

    sleep(opTime);
    printf("线程 %d: 等待读\n", id);

    sem_wait(&mutex3);
    sem_wait(&RWMutex);
    sem_wait(&mutex2);
    readCount++;
    if (readCount == 1)
        sem_wait(&wrt);
    sem_post(&mutex2);
    sem_post(&RWMutex);
    sem_post(&mutex3);

    printf("线程 %d: 开始读\n", id);
    /* reading is performed */
    sleep(lastTime);
    printf("线程 %d: 结束读\n", id);

    sem_wait(&mutex2);
    readCount--;
```

```
    if (readCount == 0)
        sem_post(&wrt);
    sem_post(&mutex2);

    pthread_exit(0);
}

//写者
void *Writer(void *param)
{
    int id = ((struct data *)param)->id;
    int lastTime = ((struct data *)param)->lastTime;
    int opTime = ((struct data *)param)->opTime;

    sleep(opTime);
    printf("线程 %d: 等待写\n", id);

    sem_wait(&mutex1);
    writeCount++;
    if (writeCount == 1)
    {
        sem_wait(&RWMutex);
    }
    sem_post(&mutex1);

    sem_wait(&wrt);
    printf("线程 %d: 开始写\n", id);
    /* writing is performed */
    sleep(lastTime);
```

```
    printf("线程 %d: 结束写\n", id);
    sem_post(&wrt);

    sem_wait(&mutex1);
    writeCount--;
    if (writeCount == 0)
    {
        sem_post(&RWMutex);
    }
    sem_post(&mutex1);

    pthread_exit(0);
}

int main()
{
    //pthread
    pthread_t tid; // the thread identifier

    pthread_attr_t attr; //set of thread attributes

    /* get the default attributes */
    pthread_attr_init(&attr);

    //initial the semaphores
    sem_init(&mutex1, 0, 1);
    sem_init(&mutex2, 0, 1);
    sem_init(&mutex3, 0, 1);
    sem_init(&wrt, 0, 1);
```

```
sem_init(&RWMutex, 0, 1);

readCount = writeCount = 0;

int id = 0;
while (scanf("%d", &id) != EOF)
{

    char role;     //producer or consumer
    int opTime;    //operating time
    int lastTime; //run time

    scanf("%c%d%d", &role, &opTime, &lastTime);
    struct data *d = (struct data *)malloc(sizeof(struct data));

    d->id = id;
    d->opTime = opTime;
    d->lastTime = lastTime;

    if (role == 'R')
    {
        printf("创建 %d 线程: 读者\n", id);
        pthread_create(&tid, &attr, Reader, d);
    }
    else if (role == 'W')
    {
        printf("创建 %d 线程: 写者\n", id);
        pthread_create(&tid, &attr, Writer, d);
    }
```

```
    }

    sem_destroy(&mutex1);
    sem_destroy(&mutex2);
    sem_destroy(&mutex3);
    sem_destroy(&RWMutex);
    sem_destroy(&wrt);

    return 0;
}
```

哲学家就餐问题：

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <pthread.h>
#include <errno.h>
#include <math.h>
//筷子作为 mutex
pthread_mutex_t chopstick[6];
void *eat_think(void *arg)
{
    char phi = *(char *)arg;
    int left, right; //左右筷子的编号
    switch (phi)
    {
    case 'A':
        left = 5;
```

```
        right = 1;
        break;
    case 'B':
        left = 1;
        right = 2;
        break;
    case 'C':
        left = 2;
        right = 3;
        break;
    case 'D':
        left = 3;
        right = 4;
        break;
    case 'E':
        left = 4;
        right = 5;
        break;
    }

    int i;
    for (i = 0; i < 3; ++i)
    {
        usleep(3);                              //思考
        pthread_mutex_lock(&chopstick[left]); //拿起左手的筷子
        printf("Philosopher %c fetches chopstick %d\n", phi, left);
        if (pthread_mutex_trylock(&chopstick[right]) == EBUSY)
        {                                       //拿起右手的筷子
            pthread_mutex_unlock(&chopstick[left]); //如果右边筷子被拿走放下左手的
```

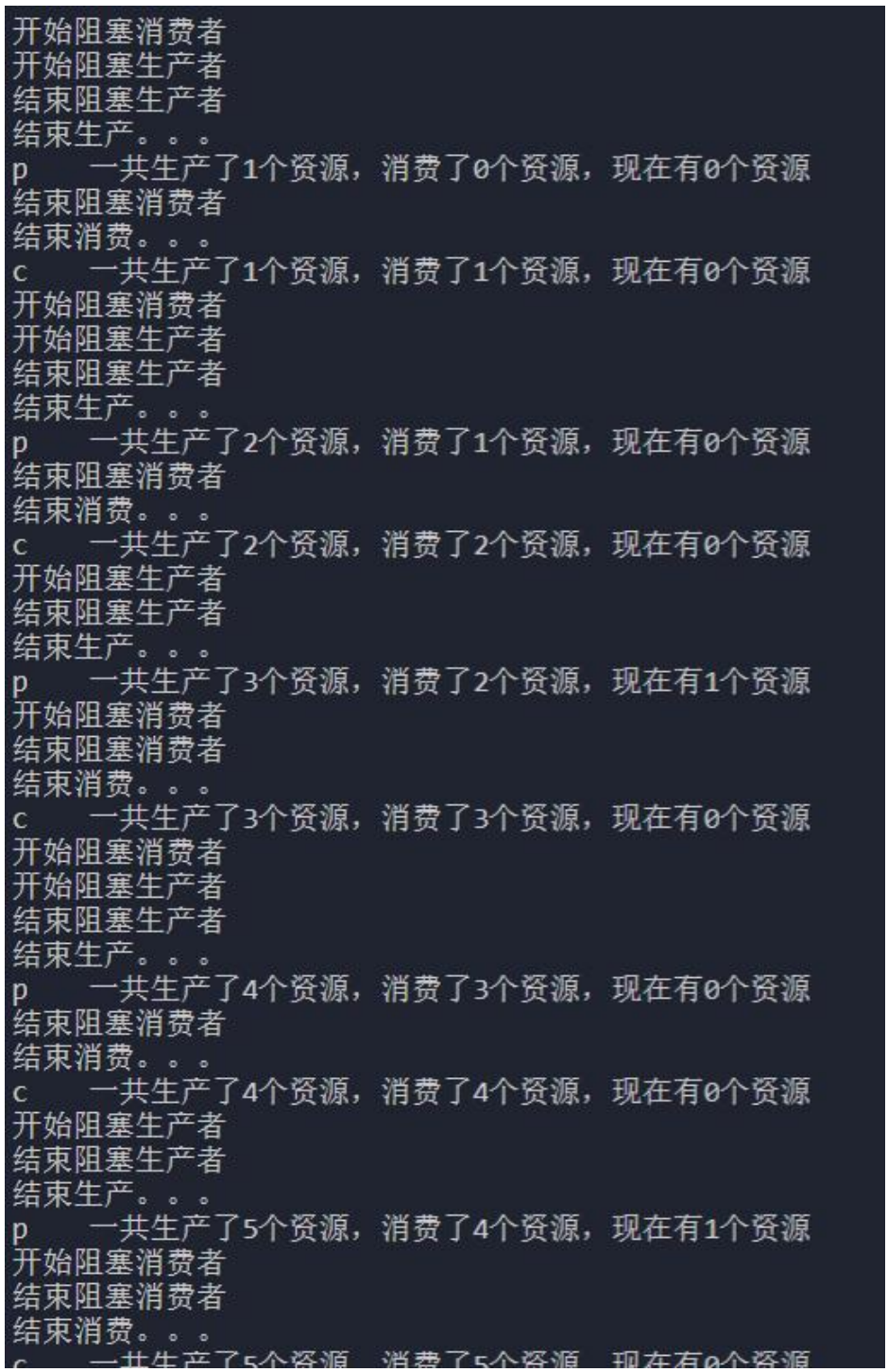
```
筷子
            continue;
        }

        // pthread_mutex_lock(&chopstick[right]); //拿起右手的筷子，如果想观察死锁，把上一句 if 注释掉，再把这一句的注释去掉
        printf("Philosopher %c fetches chopstick %d\n", phi, right);
        printf("Philosopher %c is eating.\n", phi);
        usleep(3);                                  //吃饭
        pthread_mutex_unlock(&chopstick[left]); //放下左手的筷子
        printf("Philosopher %c release chopstick %d\n", phi, left);
        pthread_mutex_unlock(&chopstick[right]); //放下左手的筷子
        printf("Philosopher %c release chopstick %d\n", phi, right);
    }
}
int main()
{
    pthread_t A, B, C, D, E; //5 个哲学家

    int i;
    for (i = 0; i < 5; i++)
        pthread_mutex_init(&chopstick[i], NULL);
    pthread_create(&A, NULL, eat_think, "A");
    pthread_create(&B, NULL, eat_think, "B");
    pthread_create(&C, NULL, eat_think, "C");
    pthread_create(&D, NULL, eat_think, "D");
    pthread_create(&E, NULL, eat_think, "E");

    pthread_join(A, NULL);
```

```
    pthread_join(B, NULL);
    pthread_join(C, NULL);
    pthread_join(D, NULL);
    pthread_join(E, NULL);
    return 0;
}
```

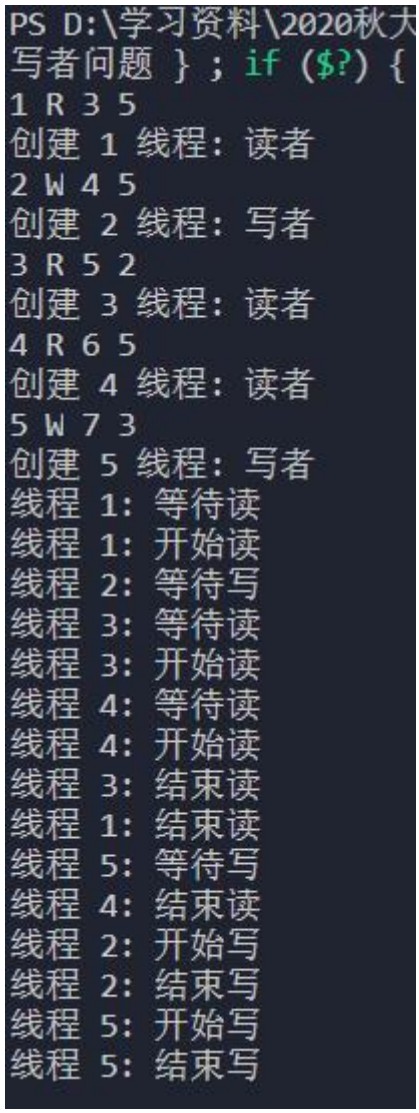
运行结果

由于输出太长了，所以这里只展示部分，以截图的形式

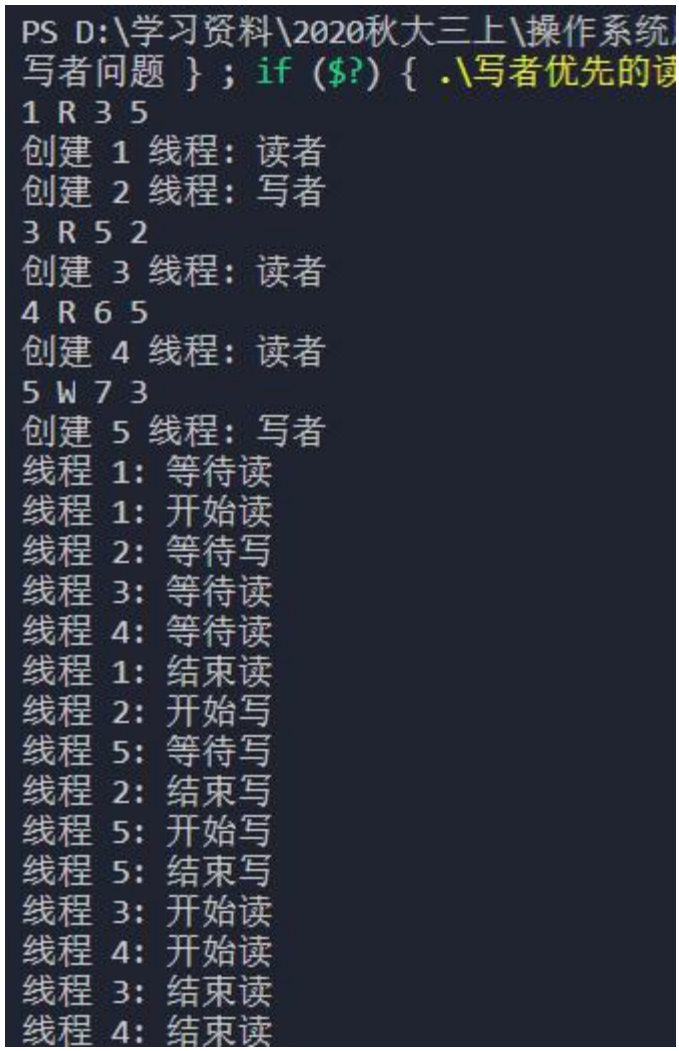
生产者-消费者问题：



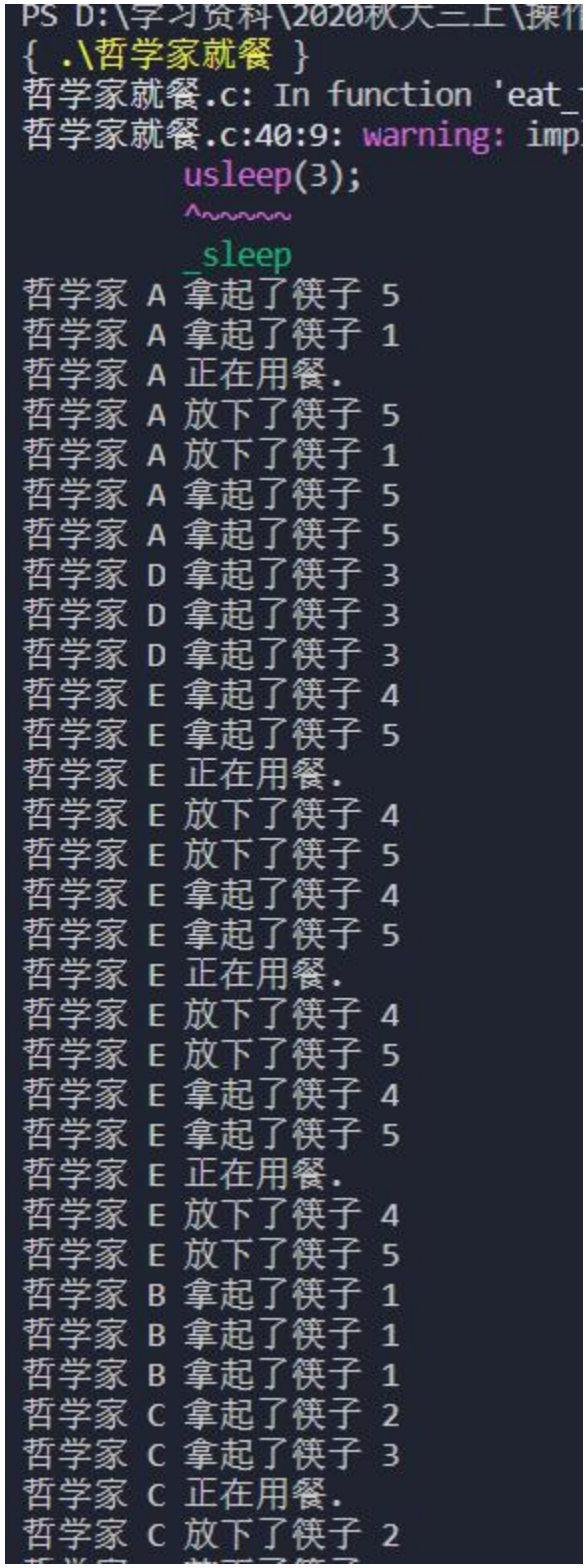
读者优先的读者-写者问题：



写者优先的读者-写者问题：



哲学家就餐问题：


PS D:\学习资料\2020秋大三上\操作
{ .\哲学家就餐 }
哲学家就餐.c: In function 'eat_
哲学家就餐.c:40:9: warning: imp
usleep(3);
^~~~~~
_sleep
哲学家 A 拿起了筷子 5
哲学家 A 拿起了筷子 1
哲学家 A 正在用餐.
哲学家 A 放下了筷子 5
哲学家 A 放下了筷子 1
哲学家 A 拿起了筷子 5
哲学家 A 拿起了筷子 5
哲学家 D 拿起了筷子 3
哲学家 D 拿起了筷子 3
哲学家 D 拿起了筷子 3
哲学家 E 拿起了筷子 4
哲学家 E 拿起了筷子 5
哲学家 E 正在用餐.
哲学家 E 放下了筷子 4
哲学家 E 放下了筷子 5
哲学家 E 拿起了筷子 4
哲学家 E 拿起了筷子 5
哲学家 E 正在用餐.
哲学家 E 放下了筷子 4
哲学家 E 放下了筷子 5
哲学家 E 拿起了筷子 4
哲学家 E 拿起了筷子 5
哲学家 E 正在用餐.
哲学家 E 放下了筷子 4
哲学家 E 放下了筷子 5
哲学家 B 拿起了筷子 1
哲学家 B 拿起了筷子 1
哲学家 B 拿起了筷子 1
哲学家 C 拿起了筷子 2
哲学家 C 拿起了筷子 3
哲学家 C 正在用餐.
哲学家 C 放下了筷子 2

结论体会

这个实训还是有一定难度的，它比较注重对这些问题的理解以及运用。但是在实际进行过程中，我也出现了很多问题，比如说，很多头文件在 vscode 中无法使用，这就导致了很多参考代码的参考性对我来说就不是很高，但好在思想都是一样的。这些进程同步问题都是操作系统中非常常见的问题，其解决方式也可以说是非常的有意思，看似非常浅显但是又富含智慧。

结论和展望

由于在每一个实验内容中我都有各自的结论展望，这里对于实验的细节部分就不再展开了，主要就谈一下关于这次实验的心得体会。

可以说，这次的实验其实还是挺仓促的，因为是穿插在各种期末考试之中进行的实验，所以多少会对实验有些影响。但是就结果而言，我感觉还是不错的，在过程中我也收获了不少。

首先就是加深了对操作系统的理解，对各种问题的理解，以及亲自实践了其解决方案。这些方案大多都不是很难，重要的是其思想。实现的过程也是很有趣的，能让我们体会到人类思想的智慧。

其次，就是提高了我的代码能力。在实现的过程中，除却思想，其实还是有很多地方我遇到了不少问题的，找 bug 的过程有时会很漫长，但正是通过了这些过程，让我的代码能力得到了提升。

最后，就是满满的满足感。能在考试周的时候完成一份这样子的实验我觉得本身也是对自己自身能力的一种提升和肯定。在未来我也会更加努力，去挑战更多的可能。