

# 中国地质大学

CHINA UNIVERSITY OF GEOSCIENCES

—— 北京 · BEIJING ——

## 课程实验报告

课程名称: 操作系统实验

班 级:	<u>10041511</u>
学 号:	<u>1004153215</u>
姓 名:	<u>陈三星</u>
指导教师:	<u>郑春梅</u>
报告日期:	<u>2018 年 6 月 1 日</u>

## 摘 要

操作系统（Operating System，简称 OS）是管理计算机硬件与软件资源的程序，是计算机系统的核心与基石，也是计算机专业重要的基础课程之一。本报告包含了 2018 年春季学期操作系统课程的相关实验内容。在正文中，报告详细记叙了具体实验项目设计与实现的过程，包括了银行家算法、处理机管理、存储器管理、缺页中断、磁盘调度、抽奖系统（多线程编程）共六个必做模块和一个选做模块——文件系统的设计。对每个模块，报告又以需求分析、概要设计、详细设计以及调试分析四个部分展开。实验中的六个必做模块都包含了计算机操作系统中至关重要的算法，而选作模块则是一系列综合算法和整体设计能力的考验。所有实验代码使用 C# 编程语言在 Visual Studio 2017 环境下完成编写。在最后一个章节中，报告总结了本学期学习操作系统的体验和得到的启发。

# 目 录

摘 要 .....	2
目 录 .....	1
正 文 .....	1
一、 银行家算法 .....	1
需求分析 .....	1
概要设计 .....	1
详细设计 .....	1
调试分析 .....	3
二、 处理机管理 .....	5
需求分析 .....	5
概要设计 .....	5
详细设计 .....	6
调试分析 .....	8
三、 存储器管理（可变式分区管理） .....	10
需求分析 .....	10
概要设计 .....	11
详细设计 .....	11
调试分析 .....	16
四、 缺页中断 .....	18
需求分析 .....	18
概要设计 .....	18
详细设计 .....	18
调试分析 .....	20
五、 磁盘调度 .....	22
需求分析 .....	22
概要设计 .....	22
详细设计 .....	23
调试分析 .....	25
六、 抽奖系统（多线程编程） .....	26
需求分析 .....	26
概要设计 .....	26
详细设计 .....	26
调试分析 .....	27
七、 文件系统设计 .....	27
需求分析 .....	27
概要设计 .....	27
详细设计 .....	28
调试分析 .....	30
八、 结论和展望 .....	33
致 谢 .....	36
参考文献 .....	36

# 正文

## 一、银行家算法

银行家算法（Banker's algorithm）是著名的计算机科学家 Edsger Dijkstra 在 1977 年发表的一个应用在操作系统资源分配时预防死锁的算法。该算法通过事先确定的最大需要资源来模拟资源分配的过程，最终得出该系统状态是否安全的结论。银行家算法顾名思义采用的是银行对待客户贷款申请时的操作思路，即当客户申请的贷款数量不超过自己当前拥有的最大值才能够满足客户的需求。

### 需求分析

本实验需要设计一个银行家算法的通用程序，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。在输入时，程序可以任意创建。

### 概要设计

银行家算法通过对可利用资源、最大需求、已分配资源、需求矩阵进行运算得到系统状态是否安全的结论。

算法的主要思想在有尚未运行的程序时，循环遍历请求资源的程序，并找到当前可以满足的程序为其分配资源，待其运行结束后返还所有被其占用资源，再继续寻找下一个程序，以此类推。如果在某一轮查找中，算法不能找到可以满足的程序，则此时系统无法继续处理任务，即可以判断为不安全。否则则为安全的系统状态。

### 详细设计

以下为程序的重点功能函数及其说明

```
private static List<int> GetTotalResources(out int resourcesNumber)
```

获取所有系统可用资源的相关信息

```
private static List<List<int>> GetClaimedResources(int resourcesNumber)
```

获取所有进程所需要的资源的相关信息

```
private static List<List<int>> GetAllocatedResources(
    int resourcesNumber, List<List<int>> claimed)
```

获取所有进程现有分配资源的相关信息（不能超过前面输入过的进程信息）

```
private static bool Judge(List<int> resources,
    List<List<int>> claimedResources,
    List<List<int>> allocatedResources)
```

真正的算法部分，综合所有信息对当前的系统环境进行判断，返回是否安全的结果

在算法部分，如下面的两个示例，程序大量使用了 C# 编程语言的 Linq 特性模拟矩阵运算，以加快运算的速度。

```
var neededResources = claimedResources.Zip(allocatedResources,
    (a, b) => a.Zip(b, (i, j) => i - j)).ToList();
```

将两个矩阵对应元素相减

```
var currentResources = resources.Zip(allocatedResources.Aggregate(
    (a, b) => a.Zip(b, (int1, int2) => int1 +
    int2).ToList()), (int1, int2) => int1 - int2).ToList();
```

将一个矩阵的所有行相加后和另一个矩阵做减运算

下图是程序的算法主体部分：

```
while (runed.Any(fg => !fg))
{
    var onceRun = false;
    for (var i = 0; i < processCount; i++)
    {
        if (!runed[i] && currentResources
            .Zip(neededResources[i], (i1, i2) => i1 - i2)
            .All(o => o >= 0))
        {
            runed[i] = true;
            Console.WriteLine(i + 1 + " is running, freed.", Color.Aqua);
            currentResources = currentResources.Zip(
                allocatedResources[i],
                (i1, i2) => i1 + i2).ToList();
            results.Add(i);

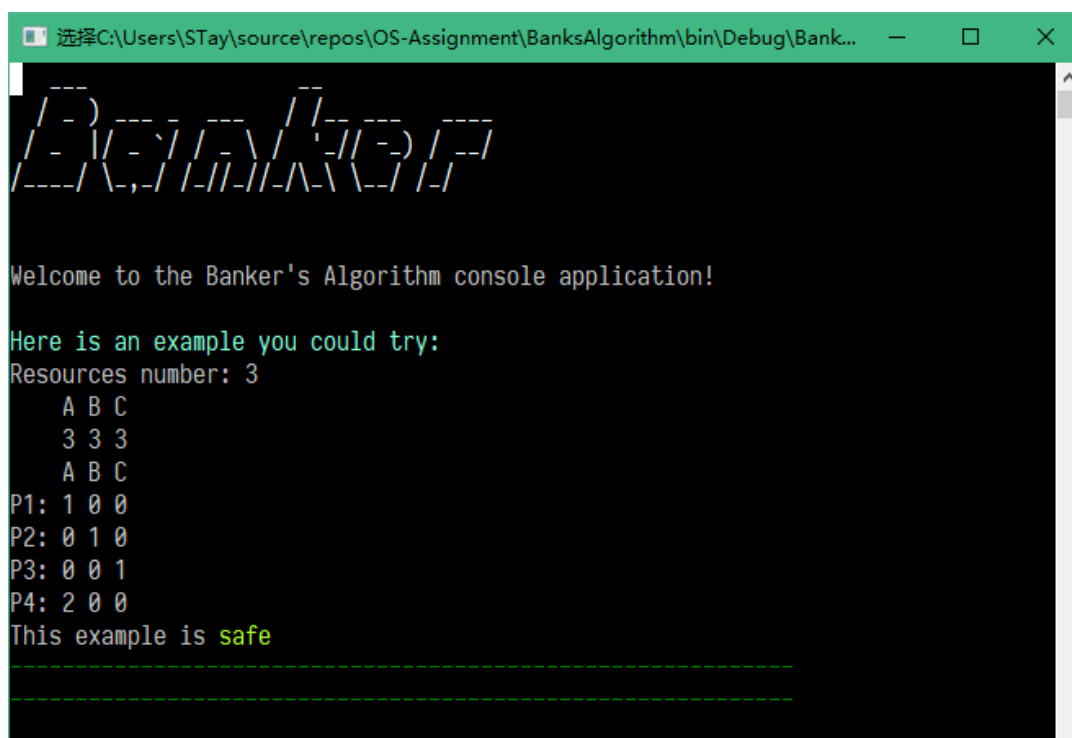
            onceRun = true;
        }
    }

    if (!onceRun)
    {
        return false;
    }
}
```

在具体实现中，算法用 `runed` 标志检测是否仍存在尚未运行的程序，循环遍历请求资源的程序，以 `currentResources` 和 `neededResources` 的减运算计算一个程序的资源要求是否能被满足。若能满足则为其分配资源，随后加入 `results` 中，待其运行结束后将 `allocatedResources` 的值加回 `currentResources` 以返还所有被改进程占用的资源，再继续寻找下一个程序，以此类推。如果在某一轮查找中，算法不能找到可以满足的程序，即 `onceRun` 标志的值为 `false`，则此时系统无法继续处理任务，即可以判断为不安全，否则即为安全的系统状态。

最后，在系统安全的情况下，算法可以通过 `results` 中保存下来的中间变量以输出安全序列。

## 调试分析



```
--
/  -)  ---  /  -)  ---  /  -)  ---  /  -)  ---
/  -  /  -  /  -  /  -  /  -  /  -  /  -  /  -
/  -  /  -  /  -  /  -  /  -  /  -  /  -  /  -
/  -  /  -  /  -  /  -  /  -  /  -  /  -  /  -

Welcome to the Banker's Algorithm console application!

Here is an example you could try:
Resources number: 3
  A B C
  3 3 3
  A B C
P1: 1 0 0
P2: 0 1 0
P3: 0 0 1
P4: 2 0 0
This example is safe
```

在程序的开始部分，程序给出了一个简单的示例输入方便使用者可以快速掌握使用程序的基本方法。在关键的提示部分，程序使用不同颜色的命令行字体进行高亮以提醒使用者。

```
How many resources do you want to have?
3
A B C
3 3 3
Please input processes' claimed resources,
an empty line to end the input.
    Processes (claimed resources):
        A B C
P1: 1 0 0
P2: 0 1 0
P3: 0 0 1
P4: 2 0 0
P5:
```

根据程序所提供的提示输入相关信息，在输入的过程中，程序具有健壮的输入功能，对于各种不合法的输入输出可以正确检测，并且兼容许多种输入格式，以及实现了自然的换行结束输入方式。

在输入之后，程序通过银行家算法对系统状态进行计算得到当前的安全结果，并输出一个可行的安全序列：

```
1 is running, freed.
2 is running, freed.
3 is running, freed.
4 is running, freed.
1->2->3->4
Safe
```

上图为一个安全的系统正常结束时的状态。

```
How many resources do you want to have?
3
A B C
3 3 3
Please input processes' claimed resources,
an empty line to end the input.
    Processes (claimed resources):
        A B C
P1: 4 4 4
P2:

Please input processes' allocated resources,
an empty line to skip the input.
NOTE: currently allocated resources can't exceed the claimed amount
Processes:
    A B C
P1:

Unsafe
```

上图为一个不安全的系统结束时的状态，算法输出醒目的红色 Unsafe 字样，以提示用户此系统并不能安全结束。

## 二、处理机管理

处理机调度算法是一种资源分配的算法，它通过不同的调度的策略以满足不同的系统需求。如能够获得最短等待时间的段作业优先调度算法；在分时系统中保证合理的响应时间的轮转调度算法……本实验中探索实现了轮转法和高响应比优先算法。

### 需求分析

本实验需要设计一个模拟进程的轮转法调度过程程序。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。

分别采用

- a) 轮转法进程调度算法
- b) 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。

程序要求如下：

- a) 输出系统中进程的调度次序；
- b) 计算 CPU 利用率。

### 概要设计

轮转法进程调度算法每次调度时，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，返回到就绪队列末尾重新排队，等待下一次调度。

而 HRRN 算法每次都计算作业的优先级，随着作业等待时间的变长，优先级



不断的提高，使作业能够得到更快的执行。

## 详细设计

程序首先实现了作业的数据结构 Job，Job 包含名称 Name，到达时间 ArrivalTime，服务时间 ServiceTime，等待时间 WaitTime 等成员变量：

```
4 references | Sanxing Chen, 8 days ago | 1 author, 2 changes
public class Job
{
    1 reference | Sanxing Chen, 8 days ago | 1 author, 1 change
    public Job(string name, int arrivalTime, int serviceTime)
    {
        Name = name;
        ArrivalTime = arrivalTime;
        ServiceTime = serviceTime;
    }

    2 references | Sanxing Chen, 8 days ago | 1 author, 1 change
    public string Name { get; }

    // arrival time of job
    5 references | Sanxing Chen, 8 days ago | 1 author, 1 change
    public int ArrivalTime { get; }

    // serviceTime of job
    5 references | Sanxing Chen, 8 days ago | 1 author, 1 change
    public int ServiceTime { get; }

    4 references | Sanxing Chen, 8 days ago | 1 author, 1 change
    public int WaitTime { get; set; }

    2 references | Sanxing Chen, 8 days ago | 1 author, 1 change
    public int RunAmount { get; set; }
}
```

```

var currentJob = readyList.First();
readyList.RemoveAt(0);

if ((currentJob.RunAmount += timePiece) ≥ currentJob.ServiceTime)
{
    var overuse = currentJob.RunAmount - currentJob.ServiceTime;
    var endTime = currentTime + timePiece - overuse;
    currentJob.WaitTime = endTime - currentJob.ArrivalTime;

    var turnaroundTime = currentJob.WaitTime * 1.0 / currentJob.ServiceTime;

    Console.WriteLine(
        $"{runedNum}\t{currentJob.Name}\t{endTime}\t{currentJob.WaitTime}\t{turnaroundTime}");

    totalWorkTime += currentJob.ServiceTime;
    totalWaitingTime += currentJob.WaitTime;
    totalTurnaroundTime += turnaroundTime;

    runedNum++;
}
else
{
    readyList.Add(currentJob);
}

```

对于 RR 算法，每次取 readyList 就绪队列中最靠前的作业，并为其分配一个时间片，随后统计对应的服务时间和带权轮转时间，最后计算 CPU 占用的情况。

```

while (runedNum < processNum)
{
    while(jobList.Count ≠ 0 && jobList.Last().ArrivalTime ≤ currentTime)
    {
        readyList.Add(jobList.Last());
        jobList.RemoveAt(jobList.Count - 1);
    }

    if (readyList.Count == 0)
    {
        currentTime = jobList.Last().ArrivalTime;
        continue;
    }

    foreach (var job in readyList)
    {
        job.WaitTime = currentTime - job.ArrivalTime;
    }

    readyList.Sort(hrrnCmp);
    var currentJob = readyList.Last();
    readyList.RemoveAt(readyList.Count - 1);

    var turnaroundTime = (currentJob.WaitTime + currentJob.ServiceTime) * 1.0 / currentJob.ServiceTime;

    Console.WriteLine(
        $"{runedNum}\t{currentJob.Name}\t{currentTime}\t{currentTime + currentJob.ServiceTime}\t{currentJob.WaitTime}");

    totalWorkTime += currentJob.ServiceTime;
    totalWaitingTime += currentJob.WaitTime;
    totalTurnaroundTime += turnaroundTime;

    currentTime += currentJob.ServiceTime;
    runedNum++;
}

```

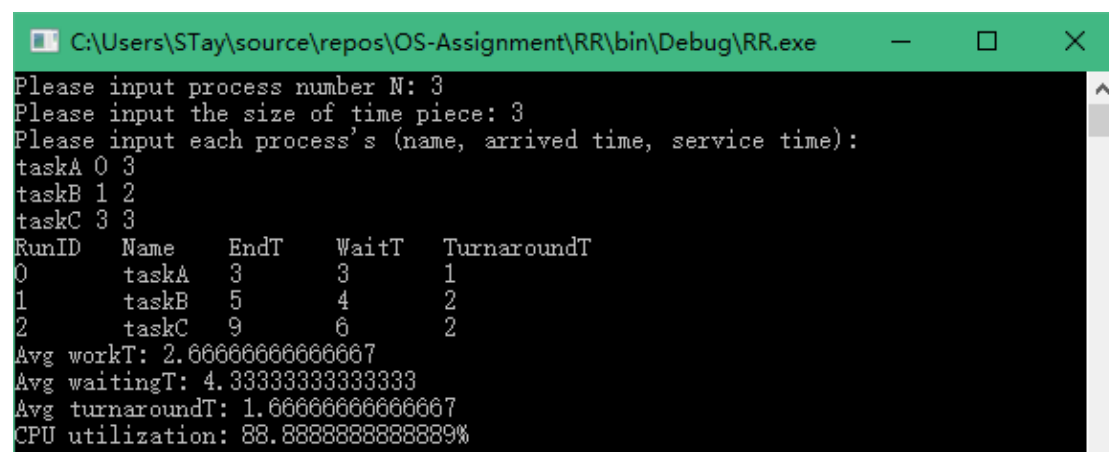
```
var hrrnCmp = Comparer<Job>.Create((b, a) =>
{
    if (a.WaitTime == b.WaitTime && a.WaitTime == 0)
    {
        return a.ServiceTime.CompareTo(b.ServiceTime);
    }
    return ((a.WaitTime + a.ServiceTime) * 1.0 / a.ServiceTime).CompareTo(
        (b.WaitTime + b.ServiceTime) * 1.0 / b.ServiceTime);
});
```

上图为算法使用的响应比计算比较函数

HRRN 算法则复杂一些，在循环的每一步需要为每个作业重新计算响应比，因此算法的复杂度也略高。

## 调试分析

下图为两组测例在 RR 算法下的运行结果：



```
C:\Users\STay\source\repos\OS-Assignment\RR\bin\Debug\RR.exe
Please input process number N: 3
Please input the size of time piece: 3
Please input each process's (name, arrived time, service time):
taskA 0 3
taskB 1 2
taskC 3 3
RunID  Name    EndT  WaitT  TurnaroundT
0      taskA    3     3      1
1      taskB    5     4      2
2      taskC    9     6      2
Avg workT: 2.666666666666667
Avg waitingT: 4.333333333333333
Avg turnaroundT: 1.666666666666667
CPU utilization: 88.8888888888889%
```

```

C:\Users\STay\source\repos\OS-Assignment\RR\bin\Debug\RR.exe
Please input process number N: 10
Please input the size of time piece: 3
Please input each process's (name, arrived time, service time):
taskA 0 3
taskB 1 2
taskC 3 3
taskD 0 3
taskE 1 2
taskF 3 3
taskG 0 3
taskH 3 3
taskI 9 3
taskJ 10 3
RunID  Name  EndT  WaitT  TurnaroundT
0      taskG  3      3      1
1      taskD  6      6      2
2      taskA  9      9      3
3      taskE  11     10     5
4      taskB  14     13     6.5
5      taskH  18     15     5
6      taskF  21     18     6
7      taskC  24     21     7
8      taskI  27     18     6
9      taskJ  30     20     6.66666666666667
Avg workT: 2.8
Avg waitingT: 13.3
Avg turnaroundT: 4.81666666666667
CPU utilization: 93.33333333333333%

```

下图为两组数据在 HRRN 算法下的对应输出结果：

```

C:\Users\STay\source\repos\OS-Assignment\HRRN\bin\Debug\HRR...
Please input process number N: 3
Please input each process's (name, arrived time, service time):
3
taskA 0 3
taskB 1 2
taskC 3 3
RunID  Name  BeginT  EndT  WaitT  TurnaroundT
0      taskA  0      3      0      1
1      taskC  3      6      0      1
2      taskB  6      8      5      3.5
Avg workT: 2.66666666666667
Avg waitingT: 1.66666666666667
Avg turnaroundT: 1.83333333333333
CPU utilization: 100%

```

```
C:\Users\STay\source\repos\OS-Assignment\HRRN\bin\Debug\HRR...
Please input process number N: 10
Please input each process's (name, arrived time, service time):
3
taskA 0 3
taskB 1 2
taskC 3 3
taskD 0 3
taskE 1 2
taskF 3 3
taskG 0 3
taskH 3 3
taskI 9 3
taskJ 10 3
RunID  Name  BeginT  EndT  WaitT  TurnaroundT
0      taskA   0       3     0      1
1      taskC   3       6     0      1
2      taskF   6       9     3      2
3      taskI   9      12     0      1
4      taskJ  12      15     2     1.666666666666667
5      taskH  15      18    12      5
6      taskD  18      21    18      7
7      taskG  21      24    21      8
8      taskB  24      26    23     12.5
9      taskE  26      28    25     13.5
Avg workT: 2.8
Avg waitingT: 10.4
Avg turnaroundT: 5.266666666666667
CPU utilization: 100%
```

可以看出 HRRN 的 CPU 利用率较 RR 相比更高。

### 三、存储器管理（可变式分区管理）

动态分区分配时根据进程的实际需要，动态的为之分配内存的一种方法。

#### 需求分析

本实验要求分别使用**空闲区表**和**空闲区链表**进行存储器管理，具体描述如下：

1. 采用空闲区表，并增加已分配区表，包括未分配区说明表、已分配区说明表（分区号、起始地址、长度、状态）。分配算法分别采用：
  - a) 最佳适应算法（内存空闲区按照尺寸大小从小到大的排列）
  - b) 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 P128，并增加已分配区表。分配算法分别采用：

- a) 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）
- b) 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。

3. 要求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。

进程分配时，需要考虑以下三种情况：进程申请的空间小于、等于或者大于系统空闲区的大小。

回收时，应该考虑 4 种情况：**释放区上邻、下邻空闲区、上下都邻接空闲区、上下都不邻接空闲区**。

每次的分配与回收，程序都会将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。

## 概要设计

程序采用两种不同的数据结构分别实现动态分区分配的算法，即用数组实现的空闲分区表和用链表实现的空闲分区链。

## 详细设计

在整个程序中，我们采用一致的分区 Partition 结构。其中包含了起点 Start，大小 Size 和使用情况 Status 三个成员变量。

```

internal class Partition
{
    1 reference | STay, 7 days ago | 1 author, 1 change
    public Partition(int start, int size, bool status = false)
    {
        Start = start;
        Size = size;
        Status = status;
    }

    4 references | STay, 7 days ago | 1 author, 1 change
    public int Start { get; }
    13 references | STay, 7 days ago | 1 author, 1 change
    public int Size { get; set; }
    7 references | STay, 7 days ago | 1 author, 1 change
    public bool Status { get; set; }
}

```

程序分别用两种不同的数据结构以支持两种存储动态分区表的方式：

```

public static List<Partition> FreePartitionTable = new List<Partition>();

public static LinkedList<Partition> FreePartitionTable = new
LinkedList<Partition>();

```

内存分配时需要考虑三种情况，但申请空间小于空闲区大小和等于空闲区大小的处理方式类似：

```

2 references | STay, 12 minutes ago | 1 author, 1 change
private static void AllocateMemory(LinkedListNode<Partition> node)
{
    node.Value.Status = true;
    TaskTable.Add(NumberOfTask++, node.Value.Start);
    ShowStatusOfMemory();
}

```

内存回收时需要考虑四种情况，程序用 up 标志符来表示上临分界区的状态，用 down 标志符表示下临分界区的状态：

```
if (pos > 0 && FreePartitionTable[pos - 1].Status == false)
{
    up = true;
}

if (pos < FreePartitionTable.Count - 1 && FreePartitionTable[pos + 1].Status == false)
{
    down = true;
}

if (up && down)
{
    FreePartitionTable[pos - 1].Size +=
        FreePartitionTable[pos].Size +
        FreePartitionTable[pos + 1].Size;
    FreePartitionTable.RemoveAt(pos);
    FreePartitionTable.RemoveAt(pos);
}
else if (up)
{
    FreePartitionTable[pos - 1].Size += FreePartitionTable[pos].Size;
    FreePartitionTable.RemoveAt(pos);
}
else if (down)
{
    FreePartitionTable[pos].Size += FreePartitionTable[pos + 1].Size;
    FreePartitionTable.RemoveAt(pos + 1);
}
```

使用空闲分区链时，只是将下标访问替换为指针访问，并且在遍历的过程中使用指针进行迭代遍历：

```
if (pos.Previous != null && pos.Previous.Value.Status == false)
{
    up = true;
}

if (pos.Next != null && pos.Next.Value.Status == false)
{
    down = true;
}

if (up && down)
{
    pos.Previous.Value.Size +=
        pos.Value.Size +
        pos.Next.Value.Size;
    FreePartitionTable.Remove(pos);
    FreePartitionTable.Remove(pos);
}
else if (up)
{
    pos.Previous.Value.Size += pos.Value.Size;
    FreePartitionTable.Remove(pos);
}
else if (down)
{
    pos.Value.Size += pos.Next.Value.Size;
    FreePartitionTable.Remove(pos.Next);
}

TaskTable.Remove(index);
Console.WriteLine(index + " has been recycled.");
```



在实现了基础的分配和回收内存的功能函数后我们可以在此之上着手实现具体的算法。

下图为循环首次适应算法的实现：

```
var i = 0;
var can = false;
while (i++ < FreePartitionTable.Count)
{
    if (FreePartitionTable[nextSearchPoint].Status ==
        false && FreePartitionTable[nextSearchPoint].Size >
            newJob)
    {
        can = true;
        break;
    }
    nextSearchPoint++;
    if (nextSearchPoint == FreePartitionTable.Count)
    {
        nextSearchPoint = 0;
    }
}

if (can)
{
    AllocateMemory(nextSearchPoint);
}
else
{
    Console.WriteLine("Can not allocate memory for this task.");
}
```

NextSearchPoint 是一个用来记录当前循环搜索位置的全局变量。我们使用 i 进行额外的循环计数，保证最多循环列表一次。

下图为空闲分区链上首次适应算法的实现：

```
var pos = FreePartitionTable.First;
var can = false;
while (true)
{
    if (pos.Value.Status == false && pos.Value.Size > newJob)
    {
        can = true;
        break;
    }
    if (pos.Next == null)
    {
        break;
    }

    pos = pos.Next;
}

if (can)
{
    AllocateMemory(pos);
}
else
{
    Console.WriteLine("Can not allocate memory for this task.");
}
```

每次搜索链时我们都从头开始，链式遍历空闲分区链，找到第一次可行的空闲分区为止。

## 调试分析

```
选择C:\Users\STay\source\repos\OS-Assignment\MemoryAllocation\...
Please input the number of Free partition: 10
Index   Start   Size   Status
0        0       56     False
1       57       59     False
2      117       55     False
3      173       57     False
4      231       97     False
5      329       34     False
6      364       17     False
7      382       59     False
8      442       51     False
9      494       82     False
Please input the algorithm's name. (NF BF exit)
NF
Index   Start   Size   Status
0        0       56     False
1       57       59     False
2      117       55     False
3      173       57     False
4      231       97     False
5      329       34     False
6      364       17     False
7      382       59     False
8      442       51     False
9      494       82     False
Please choose which type of action do you what?
1. add task    2. end task
-
```

为方便使用，程序提供了自动生成初始数据的功能，在使用者输入分区数后程序会自动生成分区表，并将初始的状态展现给用户，再根据输入选择相应的算法。若使用者选择了 BF 算法，程序会展示分区按大小进行排序后的情况，如下图所示：

```

C:\Users\STay\source\repos\OS-Assignment\MemoryAllocation\bin\...
Please input the number of Free partition: 10
Index  Start  Size  Status
0      0      27   False
1      28     25   False
2      54     87   False
3      142    18   False
4      161    86   False
5      248    92   False
6      341     4   False
7      346    92   False
8      439    15   False
9      455    21   False
Please input the algorithm's name. (NF BF exit)
bf
Index  Start  Size  Status
0      341     4   False
1      439    15   False
2      142    18   False
3      455    21   False
4      28     25   False
5      0      27   False
6      161    86   False
7      54     87   False
8      248    92   False
9      346    92   False
Please choose which type of action do you what?
1. add task    2. end task

```

在此之上我们可以进行添加任务和结束任务的操作，以空闲分区表之上的最佳适应 BestFit 算法为例：

```

C:\Users\STay\source\repos\OS-Assignment\MemoryAllocation\bin\...
Please choose which type of action do you what?
1. add task    2. end task
1
Please input the size of new Job(q to exit): 80
Can not allocate memory for this task.
Please choose which type of action do you what?
1. add task    2. end task
1
Please input the size of new Job(q to exit): 70
Index  Start  Size  Status
0      476    11   False
1      43     13   False
2      102    25   False
3      0      42   False
4      186    43   False
5      57     44   False
6      128    57   True
7      408    67   False
8      230    78   True
9      309    98   True
Please choose which type of action do you what?
1. add task    2. end task
2
Index  Start
0      128
1      309
2      230
Please input the ID of task: 0
0 has been recycled.
Please choose which type of action do you what?

```

在添加任务后，我们可以发现大小最合适的空闲分区被占用了。在选择结束任务时，程序会给出当前正在运行的程序的列表和对应的内存起始地址。

## 四、缺页中断

缺页中断就是要访问的页不在主存，需要操作系统将其调入主存后再进行访问。进程运行过程中，如果发生缺页中断，而此时内存中有没有空闲的物理块是，为了能够把所缺的页面装入内存，系统必须从内存中选择一页调出到磁盘的对换区。但此时应该把哪一个页面换出，则需要根据一定的页面置换算法来确定。

### 需求分析

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。

要求：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。

### 概要设计

LRU 的原理是置换最近一段时间以来最长时间未访问过的页面；FIFO 的原理是置换最先调入内存的页面，即置换在内存中驻留时间最久的页面；改进的 CLOCK 算法则为每个页帧设置了两个标志位，一个为修改位（M），一个为访问位（A），并通过对应的逻辑进行页面替换。

### 详细设计

LRU 算法使用字典 mem 统计一个内存中页的近期命中情况

```
var mem = new Dictionary<int, int>();
foreach (var pageRef in pageRefs)
{
    var hit = false;
    var leastRecentUsedKey = -1;
    foreach (var i in mem.ToList())
    {
        if (i.Key == pageRef)
        {
            hit = true;
            hitCount++;
            mem[i.Key] = 0;
        }
        else
        {
            mem[i.Key]++;
            if (leastRecentUsedKey == -1 || mem[i.Key] > mem[leastRecentUsedKey])
            {
                leastRecentUsedKey = i.Key;
            }
        }
    }

    if (!hit)
    {
        if (mem.Count >= memoryNum)
        {
            mem.Remove(leastRecentUsedKey);
        }

        mem.Add(pageRef, 0);
    }

    ShowPageResult(hit, pageRef, mem.Keys.ToList());
}
```

FIFO 最为简单，直接使用队列 Queue 进行模拟，对每个到达的请求直接处理即可

```
var mem = new Queue<int>();
foreach (var pageRef in pageRefs)
{
    var hit = false;
    foreach (var i in mem)
    {
        if (i == pageRef)
        {
            hit = true;
            hitCount++;
        }
    }

    if (!hit)
    {
        if (mem.Count >= memoryNum)
        {
            mem.Dequeue();
        }

        mem.Enqueue(pageRef);
    }

    ShowPageResult(hit, pageRef, mem.ToList());
}
```

改进的 CLOCK 算法代码较长不再粘贴，程序可能进行两种搜索，最多四次。从指针所指示的当前位置开始，扫描循环队列，寻找 (A = 0, M = 0) 的页面，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 A。如果第一步失败，则开始第二轮扫描，寻找 (A = 0, M = 1) 的页面，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，则将指针返回到开始位置，并将所有访问位复 0。程序返回第一步，循环执行。

## 调试分析

为避免繁琐的输入过程，程序预置了教科书上的对应的测试序列，当输入空测例时即可直接使用。在输入测例之后，程序可以选择三种算法之一进行运算。

```

C:\Users\STay\source\repos\OS-Assignment\LRU\bin\Debug\PageR...
Please input memory number N: 3
Please input request array(enter to use example):
Example: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Please input the algorithm's name. (FIFO LRU CLOCK exit)
FIFO
7[M]7
0[M]7 0
1[M]7 0 1
2[M]0 1 2
0[H]
3[M]1 2 3
0[M]2 3 0
4[M]3 0 4
2[M]0 4 2
3[M]4 2 3
0[M]2 3 0
3[H]
2[H]
1[M]3 0 1
2[M]0 1 2
0[H]
1[H]
7[M]1 2 7
0[M]2 7 0
1[M]7 0 1
Hit 5 times

```

上图为 FIFO 算法的运算结果

```

LRU
7[M]7
0[M]7 0
1[M]7 0 1
2[M]2 0 1
0[H]
3[M]2 0 3
0[H]
4[M]4 0 3
2[M]4 0 2
3[M]4 3 2
0[M]0 3 2
3[H]
2[H]
1[M]1 3 2
2[H]
0[M]1 0 2
1[H]
7[M]1 0 7
0[H]
1[H]
Hit 8 times

```

上图为 LRU 算法的运算结果



```
Clock
7[M]7
0[M]7 0
1[M]7 0 1
2[M]7 0 2
0[H]
3[M]3 0 2
0[H]
4[M]4 0 2
2[H]
3[M]4 3 2
0[M]4 0 2
3[M]4 0 3
2[M]2 0 3
1[M]1 0 3
2[M]1 2 3
0[M]1 2 0
1[H]
7[M]1 2 7
0[M]0 2 7
1[M]0 1 7
Hit 4 times
```

上图为 CLOCK 算法的运算结果（算法支持大小写输入）

程序为每次页面请求输出结果 H（Hit）或 M（miss），若 miss 则还会输出对应的替换结果。在每个算法的最后，程序会输出一共命中的次数。可以发现在这组测例中，LRU 算法达到了最优的效果。

## 五、磁盘调度

### 需求分析

本实验是模拟操作系统的磁盘寻道方式，运用磁盘访问顺序的不同来设计磁盘的调度算法 FCFS，SSTF，SCAN，CSCAN 和 NStepSCAN 算法。

程序需要设定开始磁道号寻道范围，依据起始扫描磁道号和最大磁道号数，随机产生要进行寻道的磁道号序列。

选择磁盘调度算法，显示该算法的磁道访问顺序，计算出移动的磁道总数和平均寻道总数。

按算法的寻道效率进行排序，并对各算法的性能进行分析比较。

### 概要设计

首先是 FIFS 算法，也就是先来先服务算法。假设当前磁道在某一位置，依次处理服务队列里的每一个磁道，处理起来比较简单。

SSTF，最短寻道时间算法的本质是利用贪心算法来实现，假设当前磁道在某一位置，接下来处理的是距离当前磁道最近的磁道号，处理完成之后再处理离这个磁道号最近的磁道号，直到所有的磁道号都服务完了程序结束。

SCAN 算法，电梯调度算法。先按照一个方向(比如从外向内扫描)，扫描的过程中依次访问要求服务的序列。

CSCAN 算法的思想是，访问完最里面一个要求服务的序列之后，从最外层的序号开始往里走。也就是始终保持一个方向，循环扫描。

NStepSCAN 算法则将磁盘请求队列分成若干个长度为 N 的子序列，对每个序列按 SCAN 算法处理。

## 详细设计

```
//first Come First Serve
private static void Fcfs(List<int> diskRequestList, int currentDisk)
{
    var moveQueue = new List<int>();
    var moveDistance = 0;
    var current = currentDisk;
    var cnt = 0;
    foreach (var item in diskRequestList)
    {
        moveDistance += Math.Abs(current - item);
        current = item;
        moveQueue.Add(cnt++);
    }
    ShowResult(moveQueue, moveDistance);
}
```

FCFS 算法最好实现，简单的顺序处理请求即可。

SCAN 算法的代码较为冗长，在此不粘贴全部代码。程序一个 forward 标记符记录磁头移动的方向。以下是正向移动时的代码实现，反向类似：

```
if (forward)
{
    var temp = currentDisk;
    var index = -1;

    for (int i = 0; i < diskRequestList.Count; i++)
    {
        if (!vis[i] && diskRequestList[i] >= temp)
        {
            temp = diskRequestList[i];
            index = i;
        }
    }
    if (index == -1)
    {

```

```

        forward = false;
    }
    else
    {
        moveDistance += Math.Abs(currentDisk - diskRequestList[index]);
        currentDisk = diskRequestList[index];
        vis[index] = true;
        moveQueue.Add(index);
    }
}

```

在 SCAN 之上只需要对序列进行分割后再调用 FCFS 函数就可以实现 NstepScan 算法：

```

//n Step Scan
private static void NStepScan(List<int> diskRequestList, int currentDisk)
{
    Console.WriteLine("Please input the number of step: ");
    int.TryParse(Console.ReadLine(), out var step);

    var moveQueue = new List<int>();
    var moveDistance = 0;
    var count = 0;
    while (count < diskRequestList.Count)
    {
        var range = Math.Min(step, diskRequestList.Count - count);
        if (range == 0)
        {
            break;
        }

        var stepRange = diskRequestList.GetRange(count, range);
        var res = Scan(stepRange, currentDisk, true);

        currentDisk = res.Item3;
        moveDistance += res.Item3;
        moveQueue.AddRange(res.Item1.Select(o => o + count));

        count += range;
    }

    ShowResult(moveQueue, moveDistance);
}

```

而 SSTF 算法实习则相对简单，只需要循环遍历序列找到其中最靠近磁头的值即可。

```

//shortest Seek Time First
private static void Sstf(List<int> diskRequestList, int currentDisk)
{
    var moveQueue = new List<int>();
    var moveDistance = 0;
    var vis = Enumerable.Repeat(false, diskRequestList.Count).ToList();

    while (moveQueue.Count != diskRequestList.Count)
    {
        var temp = int.MaxValue;
        var index = -1;

```

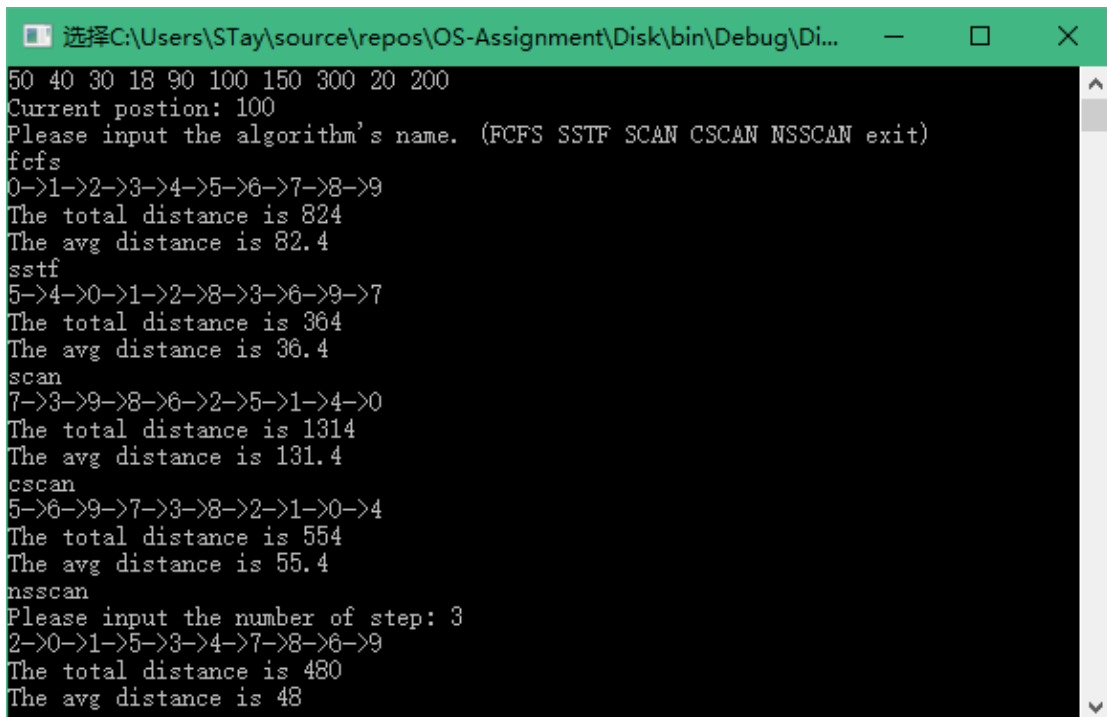
```
for (var i = 0; i < diskRequestList.Count; i++)
{
    if (!vis[i] && Math.Abs(diskRequestList[i] - currentDisk) <= temp)
    {
        temp = Math.Abs(diskRequestList[i] - currentDisk);
        index = i;
    }
}

moveDistance += temp;
currentDisk = diskRequestList[index];
vis[index] = true;
moveQueue.Add(index);
}

ShowResult(moveQueue, moveDistance);
}
```

## 调试分析

本程序内置了一组测例方便测试：



```
选择C:\Users\STay\source\repos\OS-Assignment\Disk\bin\Debug\Di...
50 40 30 18 90 100 150 300 20 200
Current position: 100
Please input the algorithm's name. (FCFS SSTF SCAN CSCAN NSSCAN exit)
fcfs
0->1->2->3->4->5->6->7->8->9
The total distance is 824
The avg distance is 82.4
sstf
5->4->0->1->2->8->3->6->9->7
The total distance is 364
The avg distance is 36.4
scan
7->3->9->8->6->2->5->1->4->0
The total distance is 1314
The avg distance is 131.4
cscan
5->6->9->7->3->8->2->1->0->4
The total distance is 554
The avg distance is 55.4
nsscan
Please input the number of step: 3
2->0->1->5->3->4->7->8->6->9
The total distance is 480
The avg distance is 48
```

五种算法对应的结果

分析这组测例，我们可以发现，SSTF 算法在这之中的表现最好，其平均寻道数仅仅为 36.4。而 Step 数为 3 的 NSSCAN 算法次之，CSCAN 算法也能达到 55.4 的平均寻道数。

## 六、抽奖系统（多线程编程）

### 需求分析

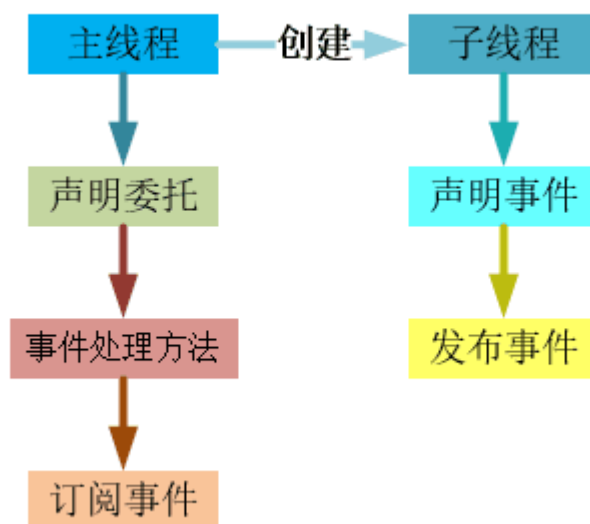
本实验要求实现一个多线程编程实现的抽奖系统。

### 概要设计

抽奖系统分为两个线程，一个后台的线程控制数字的变化，一个前台的线程则负责监听前端的输入并向后端线程发出指令。

### 详细设计

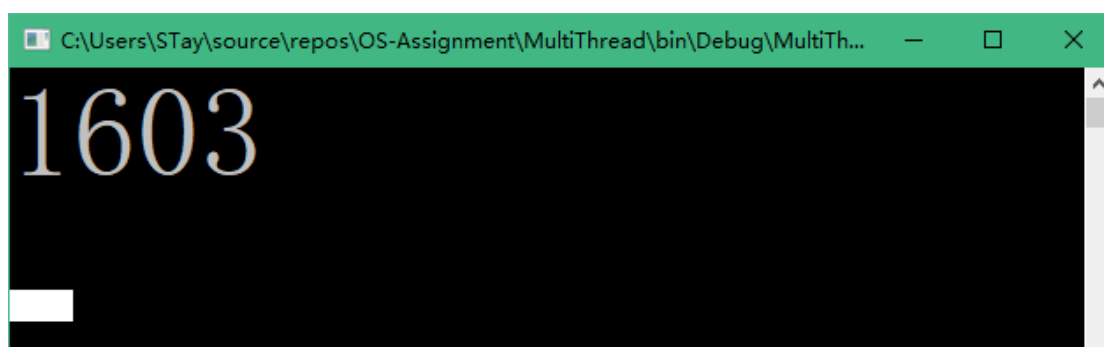
在程序的具体实现中，程序基于 C# 的事件监听机制和多线程接口实现。



主线程创建子线程，子线程声明、发布事件，主进程订阅事件。

```
var drawThread =  
    new Thread(Draw) {IsBackground = true};  
drawThread.Start();  
  
var frontThread =  
    new Thread(Pause) {IsBackground = true};  
frontThread.Start();
```

## 调试分析



由于专注于功能性的实现，本实验的 UI 较为简单。后续计划通过 .NET 平台实现一个美观的桌面应用。

在多线程程序的调试中，靠猜测判断线程的状态时不科学的。因此实验在 Visual Studio 上进行了长时间的调试，中断，查看堆栈等过程。

## 七、文件系统设计

### 需求分析

本实验要求设计并模拟实现一个简单的二级文件系统。文件系统需要实现下列几条命令 login, dir, create, delete, open, close, read, write。

实验目的：

通过一个简单多用户文件系统的设计，加深理解文件系统的内部功能及内部实现。

实验要求：

在 Linux 或者 Windows 系统上进行实验。

要求程序编写规范，运行结果正确，并写出实验报告。

### 概要设计

列目录时要列出文件名、物理地址、保护码和文件长度；源文件可以进行读写保护。

## 详细设计

程序的主要逻辑是在名为 `FileNode` 的对象中实现，所有的操作都是面向对象，对该对象本身或孩子进行操作。

12 references | STay, 5 days ago | 2 authors, 4 changes

`internal class FileNode`

{

3 references | STay, 5 days ago | 2 authors, 3 changes

`public FileNode(FileNode father, string name, PermissionType permission)`...

5 references | Sanxing Chen, 9 days ago | 1 author, 1 change

`public string Name { get; set; }`

3 references | Sanxing Chen, 9 days ago | 1 author, 1 change

`public string Content { get; set; }`

16 references | Sanxing Chen, 9 days ago | 1 author, 1 change

`public PermissionType Permission { get; set; }`

26 references | Sanxing Chen, 9 days ago | 1 author, 1 change

`public Dictionary<string, FileNode> ChildrenFiles { get; set; }`

7 references | STay, 5 days ago | 2 authors, 2 changes

`public FileNode Father { get; set; }`

2 references | Sanxing Chen, 6 days ago | 2 authors, 2 changes

`public bool AddChildrenFile(string name)`...

2 references | Sanxing Chen, 6 days ago | 1 author, 1 change

`public bool AddChildrenDir(string name)`...

2 references | Sanxing Chen, 6 days ago | 1 author, 1 change

`public bool FindChildDirectory(string name, out FileNode fn)`...

1 reference | Sanxing Chen, 6 days ago | 1 author, 1 change

`public void RemoveChild(string name)`...

1 reference | Sanxing Chen, 6 days ago | 1 author, 1 change

`public void MoveChild(string name, string newName)`...

1 reference | Sanxing Chen, 6 days ago | 1 author, 1 change

`public void ChangePermission(string name, string op, string mod)`...

2 references | Sanxing Chen, 6 days ago | 1 author, 1 change

`public void ShowContent()`...

1 reference | Sanxing Chen, 6 days ago | 1 author, 1 change

`public void ShowChildContent(string name)`...

3 references | STay, 5 days ago | 2 authors, 2 changes

`public string PathInfo()`...

1 reference | Sanxing Chen, 6 days ago | 2 authors, 2 changes

`public void ShowBaseInfo()`...

0 references | Sanxing Chen, 6 days ago | 1 author, 1 change

`public void ShowPathInfo()`...

1 reference | STay, 5 days ago | 2 authors, 3 changes

`public void ShowChildrenFiles()`...

}

FileNode 概览

而对于文件的权限控制，则是通过一个枚举型变量表示。

```
[Flags]
internal enum PermissionType
{
    Read = 4,
    Write = 2,
    Excute = 1
}
```

在创建一个新的文件节点时有两个选项，新建一个目录或者新建一个文件。对新建的目录，我们在初始化时增加两个符号链接“.”和“..”分别指向自身路径和父路径（对 root 需要特殊处理）。

```
public FileNode(FileNode father, string name, PermissionType permission, bool isDir)
{
    Father = father;
    Name = name;
    Permission = permission;
    ChildrenFiles = new Dictionary<string, FileNode>();
    Content = "";

    if (isDir)
    {
        ChildrenFiles.Add("..", father);
        ChildrenFiles.Add(".", this);
    }
}
```

对于一个 session 程序提供以下全局变量作为入口，root 是系统的根目录，\_current 是当前用户所处的路径，而\_currentFcb 是当前 open 的文件的文件控制块的模拟。

```
private static readonly FileNode root =
    new FileNode(null, "",
        PermissionType.Excute | PermissionType.Read |
        PermissionType.Write);

private static FileNode _current = root;

private static bool isLogin = false;

private static string username;

private static FileNode _currentFcb = null;
```

处理主体程序设计之外，为了程序的健壮性，程序还进行了许多异常处理：

如对输入的字符串去除开头和结尾的空格：

```
var input = Console.ReadLine()?.Trim().Split(' ').Select(o =>
o.Trim()).ToArray();
```



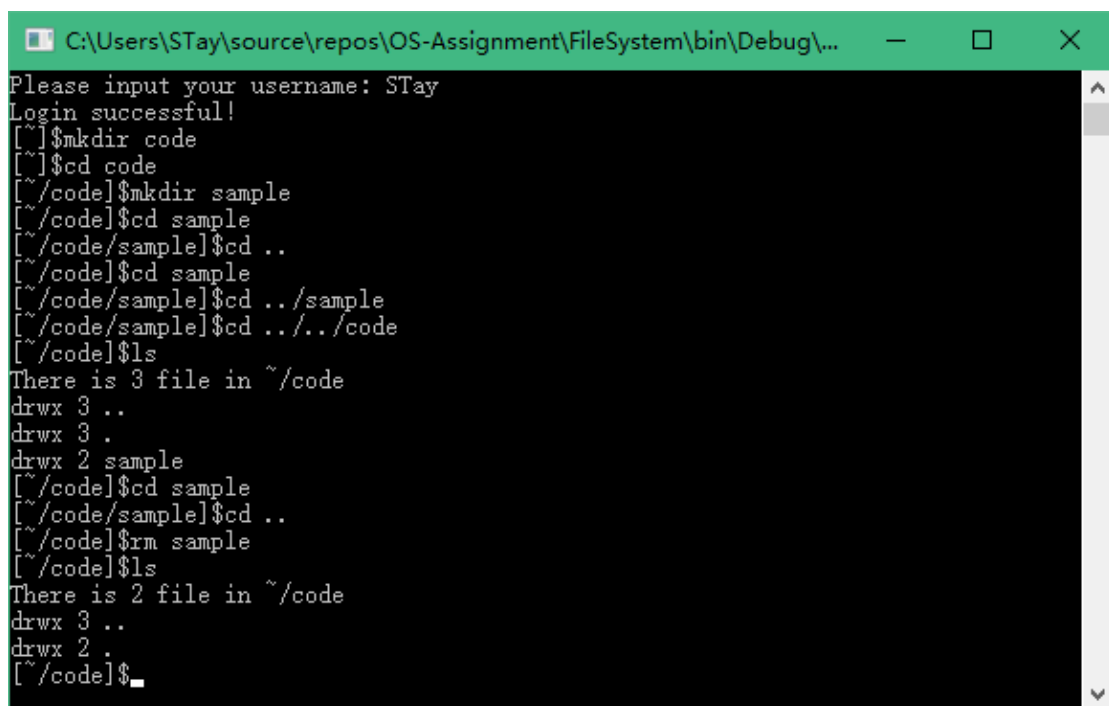
如在输入不支持的函数时的提示：

```
Console.WriteLine(  
    $"The term \"{cmd}\" is not recognized as the name of a function");
```

再如对异常的捕捉：

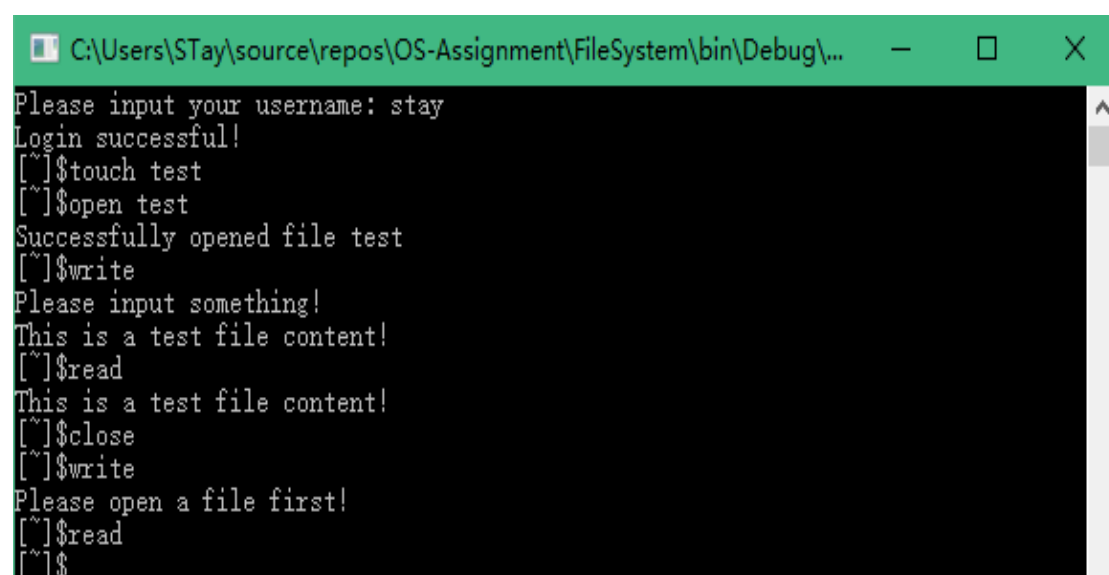
```
catch (Exception e)  
{  
    if (e is NotImplementedException)  
    {  
        Console.WriteLine(  
            $"The function \"{cmd}\" hasn't been implemented. 0.o");  
    }  
}
```

## 调试分析



灵活且健壮的目录操作

程序提供了健壮的目录跳转操作，在每个目录中可以看到两个符号链接，分别指向本身和上级目录。在跳转过程中支持包括符号链接在内的级联跳转。

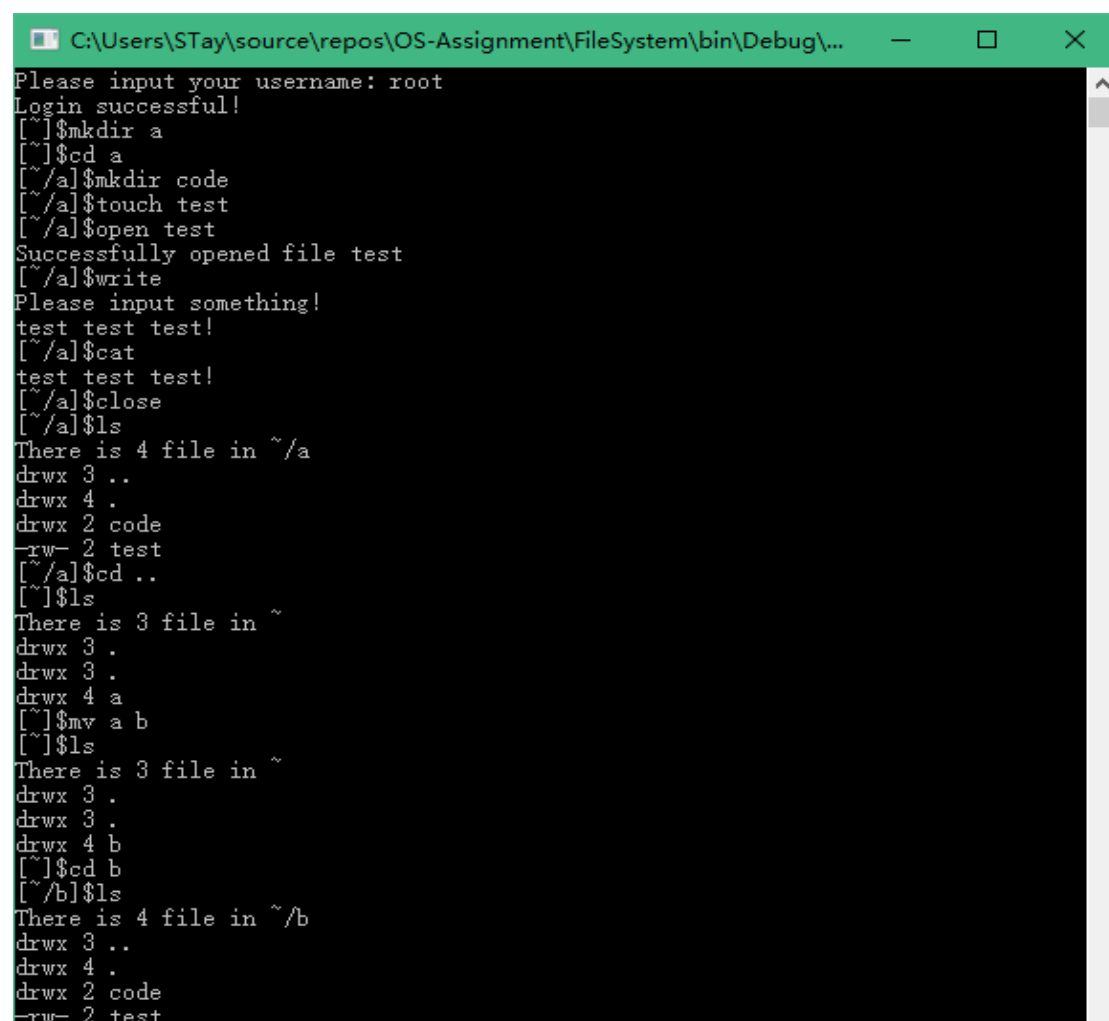
A screenshot of a terminal window with a green title bar. The title bar text is "C:\Users\STay\source\repos\OS-Assignment\FileSystem\bin\Debug\...". The terminal content shows a sequence of commands and their outputs: "Please input your username: stay", "Login successful!", "[~]\$touch test", "[~]\$open test", "Successfully opened file test", "[~]\$write", "Please input something!", "This is a test file content!", "[~]\$read", "This is a test file content!", "[~]\$close", "[~]\$write", "Please open a file first!", "[~]\$read", and "[~]\$".

```
C:\Users\STay\source\repos\OS-Assignment\FileSystem\bin\Debug\...  
Please input your username: stay  
Login successful!  
[~]$touch test  
[~]$open test  
Successfully opened file test  
[~]$write  
Please input something!  
This is a test file content!  
[~]$read  
This is a test file content!  
[~]$close  
[~]$write  
Please open a file first!  
[~]$read  
[~]$
```

读写文件操作示意图

在使用 touch 指令创建文件之后，可以使用 open 指令打开该文件，创建该文件的文件控制块，之后程序将通过该文件控制块直接访问该文件。此时使用 write 可以通过文件控制块直接向该文件中写入内容。

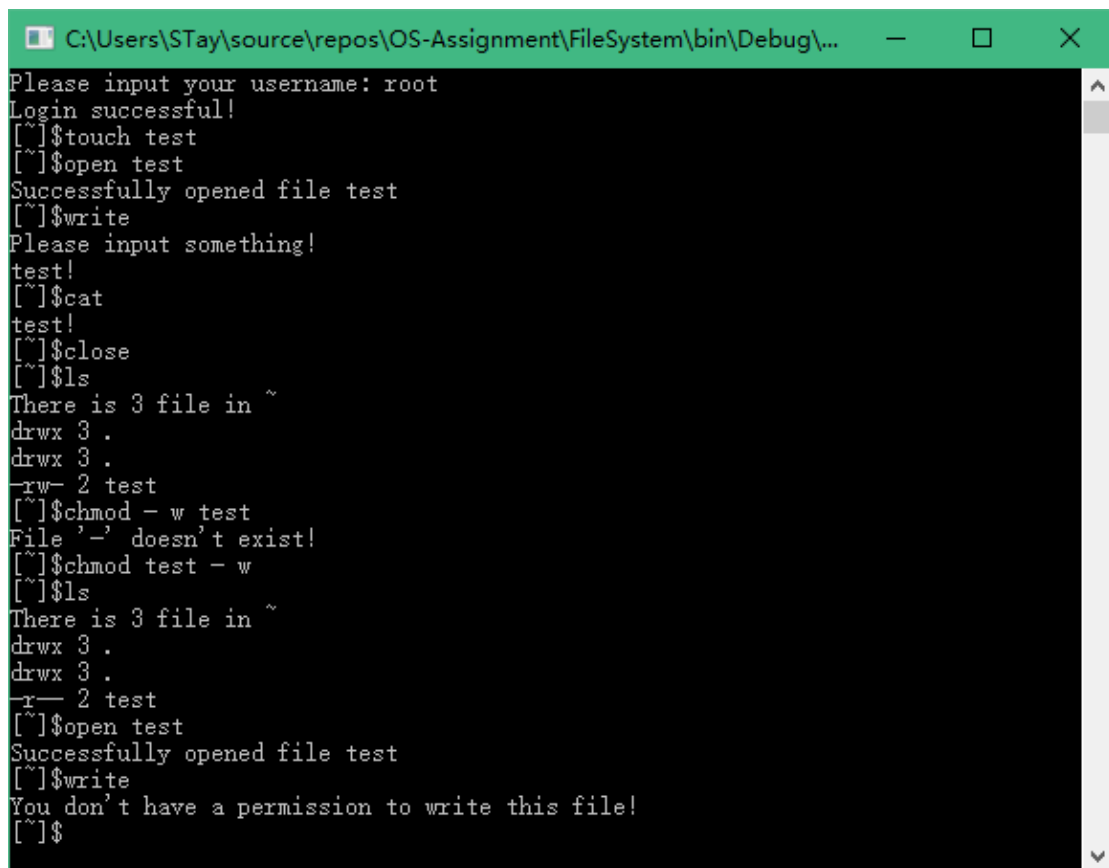
而使用 close 指令关闭文件之后，再次使用 write 尝试写文件，则会遭到系统的拒绝。系统提示用户必须打开一个文件后再进行写操作。

A terminal window with a green title bar showing the path 'C:\Users\STay\source\repos\OS-Assignment\FileSystem\bin\Debug\...'. The terminal text shows a sequence of commands and their outputs: login as root, create directory 'a', enter 'a', create 'code' subdirectory, create 'test' file, open 'test', write 'test test test!', cat 'test', close 'test', list files in 'a' (showing 4 files: '..', '.', 'code', 'test'), return to root, list files in root (showing 3 files: '.', '..', 'a'), move 'a' to 'b', list files in root (showing 3 files: '.', '..', 'b'), enter 'b', list files in 'b' (showing 4 files: '..', '.', 'code', 'test').

```
C:\Users\STay\source\repos\OS-Assignment\FileSystem\bin\Debug\...
Please input your username: root
Login successful!
[~]$mkdir a
[~]$cd a
[~/a]$mkdir code
[~/a]$touch test
[~/a]$open test
Successfully opened file test
[~/a]$write
Please input something!
test test test!
[~/a]$cat
test test test!
[~/a]$close
[~/a]$ls
There is 4 file in ~/a
drwx 3 ..
drwx 4 .
drwx 2 code
-rw- 2 test
[~/a]$cd ..
[~]$ls
There is 3 file in ~
drwx 3 .
drwx 3 .
drwx 4 a
[~]$mv a b
[~]$ls
There is 3 file in ~
drwx 3 .
drwx 3 .
drwx 4 b
[~]$cd b
[~/b]$ls
There is 4 file in ~/b
drwx 3 ..
drwx 4 .
drwx 2 code
-rw- 2 test
```

移动目录示意图

Mv 指令支持移动文件和目录，在上图所示的过程中，我们在 a 目录下创建了一个子目录和一个文件，随后，我们将 a 移动到 b，可以看到 b 中仍然保存有 a 中的文件。

A screenshot of a terminal window with a green title bar. The title bar text is "C:\Users\STay\source\repos\OS-Assignment\FileSystem\bin\Debug\...". The terminal shows a sequence of commands and their outputs. It starts with a login prompt for 'root', followed by creating a file 'test', opening it, writing to it, and then listing files. The initial permissions for 'test' are 'rw-'. Then, the 'w' permission is removed using 'chmod -w', resulting in 'r-x'. The user attempts to write again but receives a 'permission denied' error. The terminal text is as follows:

```
Please input your username: root
Login successful!
[~]$touch test
[~]$open test
Successfully opened file test
[~]$write
Please input something!
test!
[~]$cat
test!
[~]$close
[~]$ls
There is 3 file in ~
drwx 3 .
drwx 3 .
-rw- 2 test
[~]$chmod -w test
File '-' doesn't exist!
[~]$chmod test -w
[~]$ls
There is 3 file in ~
drwx 3 .
drwx 3 .
-r- 2 test
[~]$open test
Successfully opened file test
[~]$write
You don't have a permission to write this file!
[~]$
```

文件权限示意图

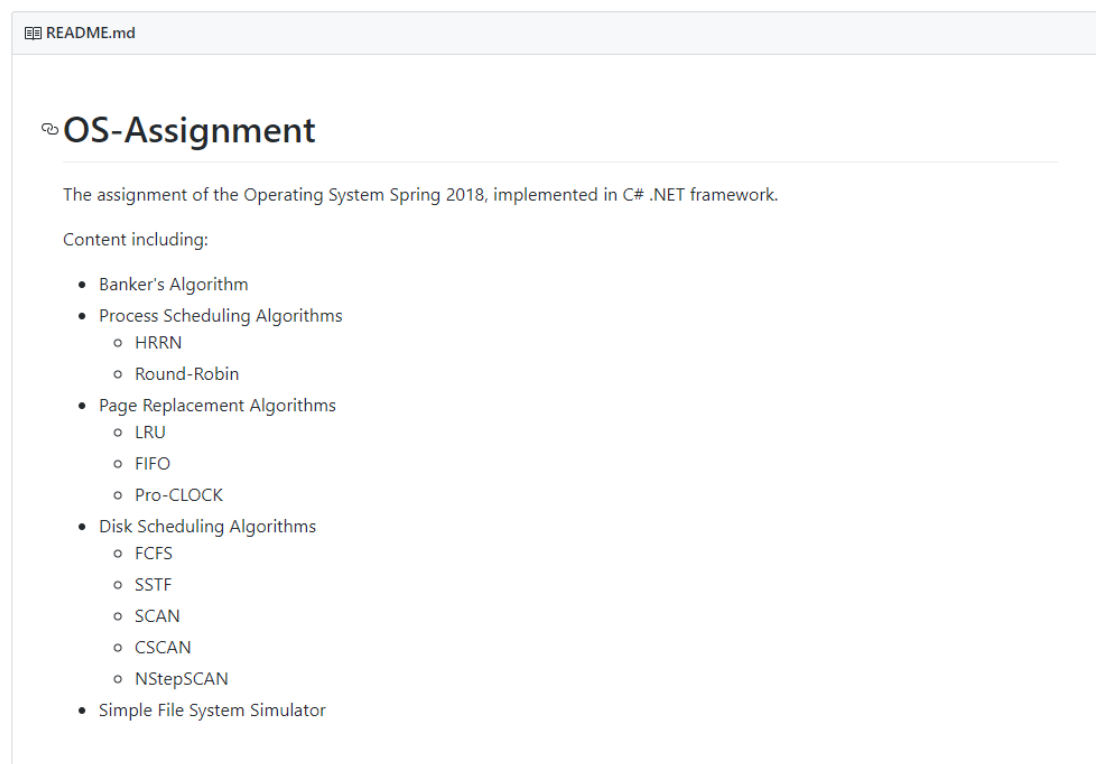
程序支持多种形式改变文件的权限，如+增加权限，-去除权限，=覆盖设置权限。在上图所示过程中，一个文件在创建之后其权限为可读可写，即“rw-”但当我们将一个文件的写权限去除后，一个文件为“r-x”，文件可以被打开但已经不能够向其中写入信息。值得一提的是，从截图中可以看到，在实验时修改权限的语句第一次输入错误，但程序仍然能健壮的应对错误的输入，提示用户其所输入的希望更改权限的文件不存在。

## 八、结论和展望

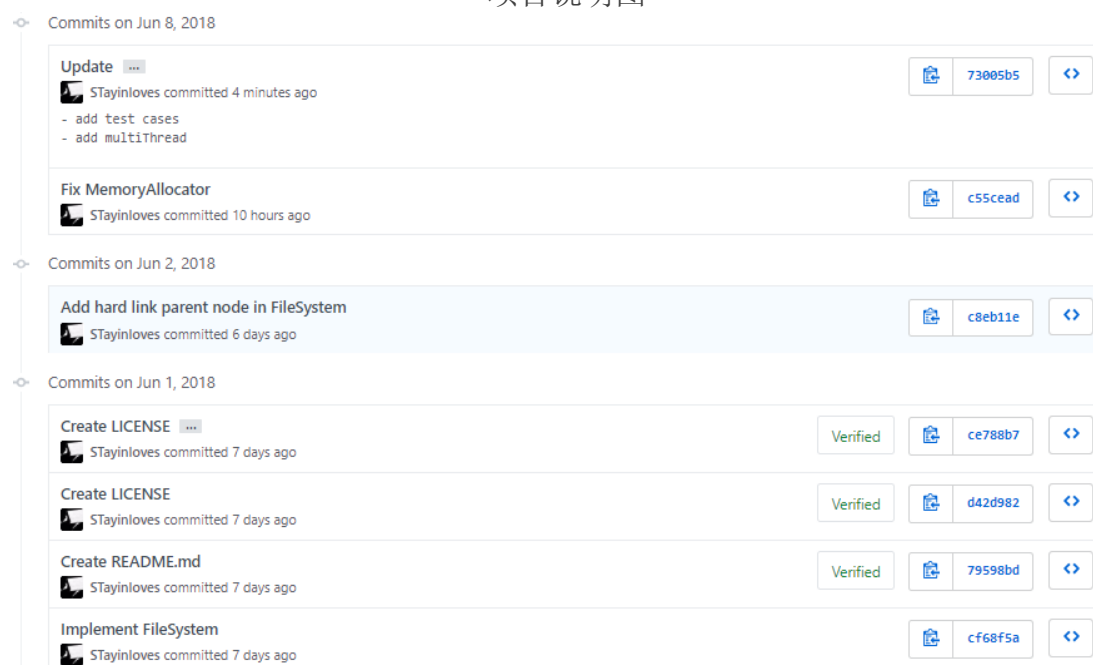
本实验报告完成了共计六个必做实验，一个选作实验的设计。其中六个必做实验设计的算法都是计算机操作系统中需要考虑到的关键算法。通过自行实现这些算法，我对这些算法的本质有了很深的理解。

在程序的具体设计和实现过程中，我着重考虑了程序的健壮性，输入输出的灵活性，以及以完全面向对象的方式使用 C#语言进行开发……以最开始的银行家实验为例，该算法不能说是一个复杂的算法，但由于处理输入输出，该算

法的文件长度最终达到 331 行。并且在实验的代码编写过程中，我全程使用 git 进行版本控制，记录每次代码的更新和改动。并在实验结束提交后将其在 GitHub 上开源



项目说明图



实验编写记录（部分）

以上所述都是对我个人代码和算法能力的一种考验。在此之前，我从未在

短时间进行过这样体量的开发。这个过程无疑也锻炼了我编写、分析和调试代码的能力。

我从大一开始接触 Linux、Unix 操作系统，在熟悉了其中的一些操作后，我也接触到了一些内核级别的调度问题。从那开始我就对操作系统的原理产生了浓厚的兴趣，但由于知识积淀的不足，我一直没有办法深入理解这一领域的知识。而本学期的课程正给予了我这个机会。

操作系统是计算机科学的专业基础课程，其重要性无需多言。在本学期的学习过程中，我一直对这门课程的学习十分重视，不局限于课本上的内容，我还购买了一本国外的经典计算机操作系统教材并结合网络上的公开课进行参照学习。

操作系统是一门资源调度与管理的科学，在资源分配的过程中，必然是有得有失的，我从中体会到一些资源分配的“哲学”。资源分配问题永远没有最好的解答，我们需要在每个具体情况进行分析与取舍，而算法则是在某种粒度上抽取出其中的模式，并对主要的模式进行分析，以谋求全局上的统一和相对最优。

在课程的结束之际，我深知在操作系统的学习上，我们仅仅踏出了第一步。而对我来说，这一步只是开始，不会是结束。

## 致 谢

感谢郑春梅老师在实验过程中对我的指导，没有您的帮助，我无法完成本实验报告。

## 参考文献

- [1] 《计算机操作系统（第三版）》汤小丹、梁红兵、哲凤屏、汤子瀛 编著
- [2] Dijkstra, Edsger W. The mathematics behind the Banker's Algorithm (EWD-623)  
Springer-Verlag, 1982. ISBN 0-387-90652-5
- [3] "Operating System Concepts" by Silberschatz, Galvin, and Gagne