信息科学技术学院、人工智能学院

课程（实习）设计

课程设计名称：操作系统实习

专　　业：计算机科学与技术

学　　号：2351610105

学 生 姓 名：方泽宇

成　　绩：

批 改 日 期：

教 师 签 名：

目　录

题目：进程调度

1.1 实验内容

- 实验目的：加深对进程调度的理解，熟悉进程调度的不同算法，比较其优劣性。
- 实验内容：假如一个系统中有 5 个进程，它们的到达时间内如表 1 所示，忽略 I/O 以及其他开销时间。若分别按抢占的短作业优先（SJF）、时间片轮转（RR，时间片=1）进行 CPU 调度，请按照上述 2 个算法，编程计算出各进程的完成时间内、周转时间、带权周转周期、平均周转周期和平均带权周转时间。

表 1 进程到达和需服务时间

进程	到达时间	服务时间
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

1.2 算法描述

- 算法 1:抢占的短作业优先(SJF)
  SJF 算法是以作业的长短来计算优先级，作业越短，其优先级越高。作业的长短是以作业所要求的运行时间来衡量的。SJF 算法可以分别用于作业调度和进程调度。在把段作业优先调度算法用于作业调度时，它将从外存的作业后备队列中选择若干个估计运行时间最短的作业，优先将他们调入内存中运行。
  SJF 可以有多种写法，一种是采取新进入的进程优先的写法，另一种是上一次运行的进程优先。我认为新进入的进程优先的写法更符合程序运行的规则，所以本次实习中我采用新进入的进程优先的办法来写。
- 算法 2:时间便流转(RR，时间片=1)
  在轮转(RR)法中，系统根据 FCFS 策略，将所有的就绪进程排列成一个就绪队列，并可设置每隔一定时间间隔(比如 30ms)完成一次中断，激活系统中的进程调度程序，完成一次调度，将 CPU 分配给队首进程，令其执行。当该新进程的时间片耗尽或运行完毕时，系统再次将 CPU 分配给心得队首进程(或者新到达的紧迫进程)。由此，可保证就绪队列中的所有进程在一个确定的时间段内，都能获得一次 CPU 执行。

1.3 实验结果

实验需要在运行程序相同目录下放置文件：filename.dat，文件内容如下：

5
A 0 3
B 2 6
C 4 4

D 6 5
E 8 2

第一种 SJF 抢占的短作业优先调度算法，新进入的进程的优先级别更高，存在如下执行日志：
第 0-1 秒，执行程序 A,剩余 2/3
第 1-2 秒，执行程序 A,剩余 1/3
第 2-3 秒，执行程序 A,剩余 0/3
第 3-4 秒，执行程序 B,剩余 5/6
第 4-5 秒，执行程序 C,剩余 3/4
第 5-6 秒，执行程序 C,剩余 2/4
第 6-7 秒，执行程序 C,剩余 1/4
第 7-8 秒，执行程序 C,剩余 0/4
第 8-9 秒，执行程序 E,剩余 1/2
第 9-10 秒，执行程序 E,剩余 0/2
第 10-11 秒，执行程序 D,剩余 4/5
第 11-12 秒，执行程序 D,剩余 3/5
第 12-13 秒，执行程序 D,剩余 2/5
第 13-14 秒，执行程序 D,剩余 1/5
第 14-15 秒，执行程序 D,剩余 0/5
第 15-16 秒，执行程序 B,剩余 4/6
第 16-17 秒，执行程序 B,剩余 3/6
第 17-18 秒，执行程序 B,剩余 2/6
第 18-19 秒，执行程序 B,剩余 1/6
第 19-20 秒，执行程序 B,剩余 0/6

对于第二种 RR 时间片轮转，采用新进入进程优先的模式，存在以下运行日志：
第 0-1 秒，执行程序 A,剩余 2/3
第 1-2 秒，执行程序 A,剩余 1/3
第 2-3 秒，执行程序 B,剩余 5/6
第 3-4 秒，执行程序 A,剩余 0/3
第 4-5 秒，执行程序 B,剩余 4/6
第 5-6 秒，执行程序 C,剩余 3/4
第 6-7 秒，执行程序 B,剩余 3/6
第 7-8 秒，执行程序 D,剩余 4/5
第 8-9 秒，执行程序 C,剩余 2/4
第 9-10 秒，执行程序 B,剩余 2/6
第 10-11 秒，执行程序 E,剩余 1/2
第 11-12 秒，执行程序 D,剩余 3/5
第 12-13 秒，执行程序 C,剩余 1/4
第 13-14 秒，执行程序 B,剩余 1/6
第 14-15 秒，执行程序 E,剩余 0/2
第 15-16 秒，执行程序 D,剩余 2/5
第 16-17 秒，执行程序 C,剩余 0/4
第 17-18 秒，执行程序 B,剩余 0/6

第 18-19 秒，执行程序 D,剩余 1/5
第 19-20 秒，执行程序 D,剩余 0/5

此程序在 macOS14.6.1 系统，采用如下编译指令：
cd "/Users/fang50253/Desktop/Files/Documents/NJFU_Training/NJFU_CS_Training/ 操作系统实习/1.进程调度/" && g++ -std=c++14 01Process_submit.cpp -o 01Process_submit && "/Users/fang50253/Desktop/Files/Documents/NJFU_Training/NJFU_CS_Training/操作系统实习/1.进程调度/"01Process_submit
运行结果的截图如下：

```
This is SJF:


Process name    Finished time   Turnaround time Weighted turnaround time
|       A       |       3       |       3       |       1.00    |
|       C       |       8       |       4       |       1.00    |
|       E       |       10      |       2       |       1.00    |
|       D       |       15      |       9       |       1.80    |
|       B       |       20      |       18      |       3.00    |
avergae turnaround time=7.20
average weighted turnaround time=1.56



This is RR:


Process name    Finished time   Turnaround time Weighted turnaround time
|       A       |       4       |       4       |       1.33    |
|       E       |       15      |       7       |       3.50    |
|       C       |       17      |       13      |       3.25    |
|       B       |       18      |       16      |       2.67    |
|       D       |       20      |       14      |       2.80    |
avergae turnaround time=10.80
average weighted turnaround time=2.71
```

1.4 实现小结

- 在本次实习的过程中，我原本写的答案是非抢占式的，也就是 SPF(相关代码我将会打包放在文件夹下，在本实习报告中不再展现)，后来发现这样的结果和其他同学的有些许偏差，于是我重新检查自己的思路，发现代码中存在的问题，重新编程，写了 SJF 和 RR 算法。
- 由于实习过程中不可以使用 C++的标准 STL 库，因此在本次实习中我通过 C++模板类创建了 queue、stack 和 priority_queue 三种模板类，queue 用于 RR，priority_queue 用于到时间以后将进程加入就绪态，以及 SJF 算法的实现。

1.5 实验代码

```
1.      #include<stdio.h>
2.      #include<iostream>
3.      #include<stdexcept>
4.      #include<string.h>
5.      #include<queue>
```

```
#define MAX_PROCESS_NUMBER 100
namespace fzy//创建命名空间fzy，用于数据结构的实现
{
    template<class T>//创建模板类
    class Less//创建比较函数
    {
    public:
        bool operator()(const T& x, const T& y)
        {
            return x < y;
        }
    };
    template<class T>
    class Greater//创建比较函数
    {
    public:
        bool operator()(const T& x, const T& y)
        {
            return x > y;
        }
    };
    template<typename T>
    struct queuenode//创建队列的节点
    {
        T v=T();
        queuenode<T>* next;
    };
    template<typename T>//创建队列数据结构
    class queue
    {
    private:
        queuenode<T>* head;
        queuenode<T>* last;
        int size_;
    public:
        queue()//队列的构造函数
        {
            head=new queuenode<T>;
            head->next=nullptr;
            last=head;
            size_=0;
        }
        ~queue()//队列的析构函数
        {
```

```
50.                 clear();
51.                 delete head;
52.             }
53.             void clear()//清空队列
54.             {
55.                 while(!empty()) pop();
56.             }
57.             void push(T v)//压入队列
58.             {
59.                 queuenode<T>* newnode = new queuenode<T>;
60.                 newnode->v = v;
61.                 newnode->next = nullptr;
62.                 last->next = newnode;
63.                 last = newnode;
64.                 ++size_;
65.             }
66.             void pop()//弹出队列
67.             {
68.                 if (size_>0)
69.                 {
70.                     queuenode<T>* temp=head->next;
71.                     head->next=temp->next;
72.                     if (head->next==nullptr) last=head;
73.                     delete temp;
74.                     --size_;
75.                 }
76.             }
77.             T front()//取队首
78.             {
79.                 if(size_>0) return head->next->v;
80.                 throw std::runtime_error("Queue is empty");//如果队列为空则抛出
    异常
81.             }
82.             T back()//取队尾
83.             {
84.                 if(size_>0) return last->v;
85.                 throw std::runtime_error("Queue is empty");//如果队列为空则抛出
    异常
86.             }
87.             bool empty()//队列的判空操作
88.             {
89.                 return size_==0;
90.             }
91.             int size()//返回队列的长度
```

```
92.             {
93.                 return size_;
94.             }
95.             void print()//定义为按照队列的入(出)队顺序进行打印
96.             {
97.                 queuenode<T> *i;
98.                 int j;
99.                 for(i=head->next,j=1;j<=size_;++j,i=i->next)
100.                {
101.                    std::cout<<i->v<<" ";
102.                }
103.                std::cout<<std::endl;
104.            }
105.        };
106.        template<typename T>
107.        struct stacknode//定义为栈的结点
108.        {
109.            T v;//使用模板类使其支持任意类型
110.            stacknode *next;
111.        };
112.        template<typename T>//模板类
113.        class stack
114.        {
115.            private:
116.            stacknode<T> *head;//链栈的头结点
117.            int size_;
118.            public:
119.            stack()//链栈的构造方法
120.            {
121.                head=new stacknode<T>;//创建头结点
122.                head->next = nullptr;
123.                size_=0;
124.            }
125.            ~stack()//栈的析构函数
126.            {
127.                clear();
128.            }
129.            void clear()//清空链栈
130.            {
131.                while(size()) pop();
132.            }
133.            void pop()//将栈顶元素弹出
134.            {
```

```
135.                 if(size_==0) throw std::runtime_error("stack is empty");//栈为
    空则抛出异常
136.                 stacknode<T>* tmp=head->next;//弹出操作
137.                 head->next=head->next->next;
138.                 --size_;
139.                 delete tmp;//释放内存
140.             }
141.             int size()//返回栈的长度
142.             {
143.                 return size_;
144.             }
145.             bool empty()//栈的判空函数
146.             {
147.                 return size_==0;
148.             }
149.             T top()//返回栈顶元素
150.             {
151.                 if(size_==0) throw std::runtime_error("stack is empty");//如果
    栈为空则抛出异常
152.                 return head->next->v;
153.             }
154.             void push(T v)//压入栈
155.             {
156.                 stacknode<T> *newnode=new stacknode<T>;
157.                 newnode->next=head->next;
158.                 head->next=newnode;
159.                 newnode->v=v;
160.                 ++size_;
161.             }
162.             void print()//定义为按照栈的出栈顺序进行打印
163.             {
164.                 int j=1;
165.                 stacknode<T> *i;
166.                 for(i=head->next;j<=size_;++j,i=i->next)
167.                 {
168.                     std::cout<<i->v<<" ";
169.                 }
170.                 std::cout<<std::endl;
171.             }
172.         };
173.         template<typename T>//使用模板类定义优先队列
174.         class priority_queue{
175.         public:
176.             priority_queue() //优先队列的无参数构造函数
```

```
177.             :size_of_priority_queue(0), capacity(MAX_PROCESS_NUMBER), compare(
    &_compare)
178.             {
179.                 pt = new T[capacity];
180.                 if(nullptr==pt) throw std::runtime_error("malloc failed");
181.             }
182.             priority_queue(int val) //带有默认大小的优先队列
183.                 :size_of_priority_queue(0), capacity(MAX_PROCESS_NUMBER), comp
    are(&_compare){
184.                 while( capacity < val) capacity <<=1;
185.                 pt = new T[capacity];//申请空间
186.                 if( nullptr == pt) throw std::runtime_error("malloc failed");
187.                 return;
188.             }
189.             priority_queue(bool (*cmp)(T&,T&)) //带有比较器函数的优先队列构造函数
190.                 :size_of_priority_queue(0), capacity(MAX_PROCESS_NUMBER), comp
    are(cmp)
191.                 {
192.                 pt = new T[capacity];
193.                 if( nullptr == pt )throw std::runtime_error("malloc failed");
194.                 return;
195.             }
196.             priority_queue( int val, bool (*cmp)(T&,T&) ) //带有默认大小并且有比
    较器的优先队列
197.                 :size_of_priority_queue(0), capacity(MAX_PROCESS_NUMBER), comp
    are(cmp){
198.                 while( capacity < val) capacity <<= 1;//申请一个大于该空间大小的
    空间
199.                 pt = new T[capacity];
200.                 if( nullptr == pt ) throw std::runtime_error("malloc failed");
201.                 return;
202.             }
203.             ~priority_queue()//优先队列的析构函数
204.             {
205.                 if( nullptr != pt){
206.                     delete[] pt;
207.                     pt = nullptr;
208.                 }
209.             }
210.             bool empty()//优先队列的判空
211.             {
212.                 return size_of_priority_queue==0;
213.             }
214.             bool push(const T& t)//压入优先队列
```

```
215.             {
216.                 T *ptt = pt;
217.                 if( size_of_priority_queue == capacity)
218.                 {
219.                     capacity *= 2;
220.                     pt = new T[capacity];
221.                     if( nullptr == pt )
222.                     {
223.                         pt = ptt;
224.                         capacity /= 2;
225.                         return false;
226.                     }
227.                     obj_cpy(pt, ptt, size_of_priority_queue);
228.                     delete[] ptt;
229.                 }
230.                 pt[size_of_priority_queue++] = t;
231.                 heap_up();//堆堆上传操作
232.                 return true;
233.             }
234.             bool pop()
235.             {
236.                 if(size_of_priority_queue==0) return 0;
237.                 if(size_of_priority_queue==1)
238.                 {
239.                     size_of_priority_queue = 0;
240.                     return 1;
241.                 }
242.                 pt[0] = pt[size_of_priority_queue-1];
243.                 size_of_priority_queue--;
244.                 heap_down();//堆堆下传操作
245.                 return 1;
246.             }
247.             T top()
248.             {
249.                 if(size_of_priority_queue<0) throw std::runtime_error("queue e
    mpty");//优先队列为空则抛出异常
250.                 return pt[0];//返回队头元素
251.             }
252.             bool is_empty_pl()const
253.             {
254.                 return 0==size_of_priority_queue;//返回队是否为空
255.             }
256.             int get_size()const
257.             {
```

```
258.                 return size_of_priority_queue;//返回队元素个数
259.             }
260.             int get_capacity()const
261.             {
262.                 return capacity;//返回队当前容量应该为2 的n 次方
263.             }
264.         private:
265.             void heap_up();//定义上传操作
266.             void heap_down();//定义下传操作
267.             void obj_cpy(T* dest, const T* sour, int n)//拷贝函数
268.             {
269.                 for(int i=0;i<n;i++) dest[i]=sour[i];
270.             }
271.             bool static _compare(T &t1, T &t2)//定义比较器
272.             {
273.                 return t1 < t2;
274.             }
275.         private:
276.             T     *pt;//数据
277.             int    size_of_priority_queue;// 元素个数
278.             int    capacity;//队容量
279.             bool (*compare)(T&,T&);//比较函数
280.
281.         };
282.         template<typename T>
283.         void priority_queue<T>::heap_up()//上传操作，基于堆
284.         {
285.             T temp;
286.             int itr = size_of_priority_queue-1;
287.             while( itr > 0 )
288.             {
289.                 if( (compare(pt[itr/2], pt[itr])))
290.                 {
291.                     temp = pt[itr];
292.                     pt[itr] = pt[itr/2];
293.                     pt[itr/2] = temp;
294.                     itr = itr/2;
295.                     continue;
296.                 }
297.                 break;
298.             }
299.             return;
300.         }
301.         template<typename T>
```

```
302.         void priority_queue<T>::heap_down()//下传操作，基于堆
303.         {
304.             T temp;
305.             int pitr = 0, citr;
306.             while(pitr<=size_of_priority_queue/2-1)
307.             {
308.                 citr = pitr * 2 + 1;
309.                 if(citr+1<size_of_priority_queue&&compare(pt[citr],pt[citr+1])
    ) ++citr;
310.                 if((compare(pt[pitr],pt[citr])))
311.                 {
312.                     temp = pt[citr];
313.                     pt[citr] = pt[pitr];
314.                     pt[pitr] = temp;
315.                     pitr = citr;//继续将pitr 指向孩子节点，进行下一次的比较
316.                     continue;
317.                 }
318.                 break;//如果处在对的位置，直接结束，不需要继续比较下去了
319.             }
320.             return;
321.         }
322.     }
323.     namespace os//定义命名空间os，用于进程调度的书写
324.     {
325.         class Process//进程，重载运算符实现时间升序排序
326.         {
327.             public:
328.             char process_name;//进程名称
329.             int time_arrive;//到达时间
330.             int time_serve;//服务时间
331.             bool operator < (const Process &W) const //最重要的一个重载，如果使用
    stl 库，那么只需要重载这一个运算符即可
332.             {
333.                 return time_arrive>W.time_arrive;
334.             }
335.             bool operator <= (const Process &W) const //因为算法中使用到了<=号
336.             {
337.                 return time_arrive>=W.time_arrive;
338.             }
339.             bool operator > (const Process &W) const
340.             {
341.                 return time_arrive<W.time_arrive;
342.             }
343.             bool operator >= (const Process &W) const
```

```
344.             {
345.                 return time_arrive<=W.time_arrive;
346.             }
347.             Process(char process_name,int time_arrive,int time_serve)//带有三个
    参数的构造函数，用于接受子类的拷贝
348.             {
349.                 this->process_name=process_name;
350.                 this->time_arrive=time_arrive;
351.                 this->time_serve=time_serve;
352.             }
353.             Process()//默认构造函数
354.             {
355.                 this->process_name=0;
356.                 this->time_arrive=0;
357.                 this->time_serve=0;
358.             }
359.         };
360.         class Process_finish:public Process//继承Process，用于SJF SPF 算法中的中
    间计算过程
361.         {
362.             public:
363.             Process_finish(Process a,int time_end,int time_turnaround,double t
    ime_turnaround_rights)//用于接收Process 的拷贝
364.             {
365.                 this->process_name=a.process_name;
366.                 this->time_arrive=a.time_arrive;
367.                 this->time_serve=a.time_serve;
368.                 this->time_end=time_end;
369.                 this->time_turnaround=time_turnaround;
370.                 this->time_turnaround_rights=time_turnaround_rights;
371.             }
372.             Process_finish()//默认构造函数，生成类数组
373.             {
374.                 this->process_name=0;
375.                 this->time_arrive=0;
376.                 this->time_end=0;
377.                 this->time_serve=0;
378.                 this->time_turnaround=0;
379.                 this->time_turnaround_rights=0;
380.             }
381.             int time_end;//完成时间
382.             int time_turnaround;//周转时间
383.             double time_turnaround_rights;//带权周转时间
384.             bool operator < (const Process &W) const //重载<
```

```
385.             {
386.                 return time_serve>W.time_serve;
387.             }
388.             bool operator <= (const Process &W) const
389.             {
390.                 return time_serve>=W.time_serve;
391.             }
392.             bool operator > (const Process &W) const
393.             {
394.                 return time_serve<W.time_serve;
395.             }
396.             bool operator >= (const Process &W) const
397.             {
398.                 return time_serve<=W.time_serve;
399.             }
400.         }finished[MAX_PROCESS_NUMBER];//使用默认构造函数构造
401.         int finished_index;
402.         class Process_remain:public Process
403.         {
404.             public:
405.             int remain;
406.             Process_remain(class Process p)//构造函数
407.             {
408.                 this->process_name=p.process_name;
409.                 this->time_arrive=p.time_arrive;
410.                 this->time_serve=p.time_serve;
411.                 this->remain=p.time_serve;
412.             }
413.             Process_remain()//构造函数
414.             {
415.                 this->remain=0;
416.             }
417.             bool operator < (const Process &W) const //重载<实现以服务时间升序排
    序
418.             {
419.                 return time_serve>W.time_serve;
420.             }
421.             bool operator <= (const Process &W) const
422.             {
423.                 return time_serve>=W.time_serve;
424.             }
425.             bool operator > (const Process &W) const
426.             {
427.                 return time_serve<W.time_serve;
```

```
428.             }
429.             bool operator >= (const Process &W) const
430.             {
431.                 return time_serve<=W.time_serve;
432.             }
433.         };
434.         class RR//时间片轮转法
435.         {
436.             private:
437.             fzy::priority_queue<Process>process;//还没有进入就绪状态的进程，采用优先队列对这些进程进行排序
438.             fzy::queue<Process_remain>doing_process;//正在被执行的进程，在用一个队列去转
439.             int process_cnt;
440.             int time;
441.             public:
442.             void read(const char filename[])//读取文件
443.             {
444.                 FILE *fp=fopen(filename,"r+");
445.                 if(fp==NULL) throw std::runtime_error("open file failed");
446.                 fscanf(fp,"%d",&process_cnt);
447.                 for(int i=1;i<=process_cnt;++i)
448.                 {
449.                     char process_name;
450.                     int time_arrive,time_serve;
451.                     fscanf(fp," %c%d%d",&process_name,&time_arrive,&time_serve);
452.                     process.push(Process(process_name,time_arrive,time_serve));//使用临时的类去赋值
453.                 }
454.                 fclose(fp);//文件读取完成
455.             }
456.             RR()
457.             {
458.                 process_cnt=0;
459.                 finished_index=0;
460.                 char filename[]="filename.dat";//构造方法，读取文件
461.                 read(filename);
462.                 memset(finished,0,sizeof finished);
463.                 finished_index=0;
464.             }
465.             void conduct()//执行RR时间片轮转
466.             {
467.                 time=0;
```

```
468.                 Process_remain doing;
469.                 while(finished_index!=process_cnt)//在所有进程完成之前
470.                 {
471.                     Process top_process;
472.                     if(!process.empty())//如果进程不空就一直执行下去
473.                     {
474.                         top_process=process.top();
475.                         while(top_process.time_arrive<=time)
476.                         {
477.                             process.pop();
478.                             doing_process.push(Process_remain(top_process));
479.                             //printf("push:%c\n",top_process.process_name);
480.                             top_process=process.top();//不断读取
481.                         }
482.                     }
483.                     doing=doing_process.front();
484.                     doing_process.pop();//执行这个进程
485.                     --doing.remain;
486.                     //printf("%c",doing.process_name);
487.                     ++time;
488.                     if(!process.empty())
489.                     {
490.                         top_process=process.top();
491.                         while(top_process.time_arrive<=time)
492.                         {
493.                             process.pop();
494.                             doing_process.push(Process_remain(top_process));
495.                             top_process=process.top();
496.                             if(process.empty()) break;
497.                         }
498.                     }
499.                     if(doing.remain==0)//这个进程已经执行完毕
500.                     {
501.                         ++finished_index;
502.                         finished[finished_index].process_name=doing.process_na
    me;
503.                         finished[finished_index].time_arrive=doing.time_arrive
    ;
504.                         finished[finished_index].time_end=time;
505.                         finished[finished_index].time_serve=doing.time_serve;
506.                         finished[finished_index].time_turnaround=time-doing.ti
    me_arrive;
507.                         finished[finished_index].time_turnaround_rights=1.0*fi
    nished[finished_index].time_turnaround/doing.time_serve;
```

```
508.                     }
509.                     else doing_process.push(doing);//将进程继续送回正在执行的队列
510.
511.                 }
512.
513.             }
514.             void display()//显示函数
515.             {
516.                 printf("Process name\t");
517.                 printf("Finished time\t");
518.                 printf("Turnaround time\t");
519.                 printf("Weighted turnaround time\t\n");
520.                 for(int i=1;i<=finished_index;++i)
521.                 {
522.                     printf("|\t%c\t|\t",finished[i].process_name);//输出进程名
    称
523.                     printf("%d\t|\t",finished[i].time_end);//输出进程完成时间
524.                     printf("%d\t|\t",finished[i].time_turnaround);//周转时间
525.                     printf("%.2lf\t|\n",finished[i].time_turnaround_rights);//
    带权周转时间
526.                 }
527.             }
528.             void display_avergae()
529.             {
530.                 double average_time_turnaround=0;//计算平均周转时间
531.                 double average_time_turnaround_rights=0;//计算平均带权周转时间
532.                 for(int i=1;i<=finished_index;++i)
533.                 {
534.                     average_time_turnaround+=finished[i].time_turnaround;
535.                     average_time_turnaround_rights+=finished[i].time_turnaroun
    d_rights;
536.                 }
537.                 average_time_turnaround/=finished_index;
538.                 average_time_turnaround_rights/=finished_index;
539.                 printf("avergae turnaround time=%.2lf\n",average_time_turnarou
    nd);//输出平均周转时间
540.                 printf("average weighted turnaround time=%.2lf",average_time_t
    urnaround_rights);//输出平均带权周转时间
541.             }
542.         };
543.         class SJF//抢占式短进程优先算法
544.         {
545.             private:
546.             void read(const char filename[])//读取文件
```

```
547.             {
548.                 FILE *fp=fopen(filename,"r+");
549.                 if(fp==NULL) throw std::runtime_error("open file failed");
550.                 fscanf(fp,"%d",&process_cnt);
551.                 for(int i=1;i<=process_cnt;++i)
552.                 {
553.                     char process_name;
554.                     int time_arrive,time_serve;
555.                     fscanf(fp," %c%d%d",&process_name,&time_arrive,&time_serve
    );
556.                     process.push(Process(process_name,time_arrive,time_serve))
    ;//创建临时类去压入优先队列中
557.                 }
558.                 fclose(fp);//关闭文件
559.             }
560.             int process_cnt;
561.             fzy::priority_queue<Process>process;
562.             fzy::priority_queue<Process_remain>doing_process;
563.             public:
564.             SJF()//默认构造方法
565.             {
566.                 process_cnt=0;
567.                 finished_index=0;
568.                 char filename[]="filename.dat";
569.                 read(filename);//读取文件
570.                 memset(finished,0,sizeof finished);
571.                 finished_index=0;
572.             }
573.             void conduct()//抢占式短进程优先
574.             {
575.                 int time=0;
576.                 Process_remain doing;
577.                 while(!(process.empty()&&doing_process.empty()))
578.                 {
579.                     Process top_process;
580.                     if(!process.empty())
581.                     {
582.                         top_process=process.top();
583.                         while(top_process.time_arrive<=time)
584.                         {
585.                             process.pop();
586.                             doing_process.push(Process_remain(top_process));
587.                             top_process=process.top();
588.                             if(process.empty()) break;
```

```
589.                         }
590.                     }
591.
592.                     doing=doing_process.top();
593.                     doing_process.pop();
594.                     --doing.remain;++time;
595.                     if(doing.remain==0)
596.                     {
597.                         ++finished_index;
598.                         finished[finished_index].process_name=doing.process_na
    me;
599.                         finished[finished_index].time_arrive=doing.time_arrive
    ;
600.                         finished[finished_index].time_end=time;
601.                         finished[finished_index].time_serve=doing.time_serve;
602.                         finished[finished_index].time_turnaround=time-doing.ti
    me_arrive;
603.                         finished[finished_index].time_turnaround_rights=1.0*fi
    nished[finished_index].time_turnaround/doing.time_serve;
604.                     }
605.                     else doing_process.push(doing);
606.                 }
607.             }
608.             void display()//显示
609.             {
610.                 printf("Process name\t");
611.                 printf("Finished time\t");
612.                 printf("Turnaround time\t");
613.                 printf("Weighted turnaround time\t\n");
614.                 for(int i=1;i<=finished_index;++i)
615.                 {
616.                     printf("|\t%c\t|\t",finished[i].process_name);//输出进程名
    称
617.                     printf("%d\t|\t",finished[i].time_end);//输出进程完成时间
618.                     printf("%d\t|\t",finished[i].time_turnaround);//周转时间
619.                     printf("%.2lf\t|\n",finished[i].time_turnaround_rights);//
    带权周转时间
620.                 }
621.             }
622.             void display_avergae()//显示平均周转时间和平均带权周转时间
623.             {
624.                 double average_time_turnaround=0;
625.                 double average_time_turnaround_rights=0;
626.                 for(int i=1;i<=finished_index;++i)
```

```
627.                 {
628.                     average_time_turnaround+=finished[i].time_turnaround;
629.                     average_time_turnaround_rights+=finished[i].time_turnaroun
    d_rights;
630.                 }
631.                 average_time_turnaround/=finished_index;
632.                 average_time_turnaround_rights/=finished_index;
633.                 printf("avergae turnaround time=%.2lf\n",average_time_turnarou
    nd);
634.                 printf("average weighted turnaround time=%.2lf",average_time_t
    urnaround_rights);
635.             }
636.         };
637.     }
638.     int main()
639.     {
640.         printf("\n\nThis is SJF:\n\n\n");//使用新来进程优先的算法
641.         os::SJF *sjf=new os::SJF();
642.         sjf->conduct();
643.         sjf->display();
644.         sjf->display_avergae();
645.         delete(sjf);
646.         printf("\n\n\n");
647.         printf("\n\nThis is RR:\n\n\n");//使用新来进程优先的算法
648.         os::RR *rr=new os::RR();
649.         rr->conduct();
650.         rr->display();
651.         rr->display_avergae();
652.         delete(rr);
653.         printf("\n\n\n");
654.         return 0;
655.     }
```

题目：进程的同步与互斥

1.1 实验内容

- 实验目的：分析进程争用资源的现象，学习解决进程互斥的方法。
- 设计内容：

用程序实现生产者—消费者问题。具体问题描述：一个仓库可以存放 K 件物品。生产者每生产一件产品，将产品放入仓库，仓库满了就停止生产。消费者每次从仓库中去一件物品，然后进行消费，仓库空时就停止消费。

数据结构：

Producer - 生产者进程，Consumer - 消费者进程

buffer: array [0..k-1] of integer;

in, out: 0..k-1; in 记录第一个空缓冲区，out 记录第一个不空的缓冲区

s1,s2,mutex: semaphore; s1 控制缓冲区不满,s2 控制缓冲区不空,mutex 保护临界区；

初始化 s1=k,s2=0,mutex=1

原语描述：

producer（生产者进程）：

```
 item_Type item;
{
    while (true)
    {
       produce(&item);
       p(s1);
       p(mutex);
       buffer[in]:=item;
       in:=(in+1) mod k;
       v(mutex);
       v(s2);
    }
}
```

consumer（消费者进程）：

```
 item_Type item;
{
    while (true)
    {
       p(s2);
       p(mutex);
         item:=buffer[out];
         out:=(out+1) mod k;
         v(mutex);
         v(s1);
    }
```

}

1.2 算法描述

- 对算法的原理进行系统描述。

1.3 实验结果

- 展示运行部分，把结果统计为表格展示；
- 需要对结果有一定的解释说明。

1.4 实现小结

- 不少于 150 字，主要写关于在实现过程中遇到的问题，以及如何解决的。

1.5 实验代码

- 实现代码，需要提供清晰的注释；
- 采用 CSDN 网站代码的显示方式，可以通过网址实现：https://highlightcode.com；
- 代码格式：小五，1.0 行间距。

题目：存储管理

1.1 实验内容

- 实现目的：通过请求页面式存储管理中页面置换算法设计，了解存储技术的特点，掌握请求页式存储管理的页面置换算法。
- 存储管理：

用程序实现生产者——消费者问题，将指令序列转换为用户虚存中的请求调用页面流。

具体要求：

页面大小为 1K

用户内存容量为 4 页到 40 页

用户外存的容量为 40k

在用户外存中，按每 K 存放 10 条指令，400 条指令在外存中的存放方式为：

0-9 条指令为第 0 页

0-19 条指令为第 1 页

.....

90-399 条指令为第 39 页

按以上方式，用户指令可组成 40 页，通过随机数产生一个指令序列，共 400 个指令（0-399）。模拟请求页式存储管理中页面置换算法，执行一条指令，首先在外存中查找所对应的页面和页面号，然后将此页面调入内存中，模拟并计算下列三种算法在不同内存容量下的命中率(页面有效次数/页面流的个数):

1. 最久未使用算法(LRU)
2. 改进的 Clock 置换算法

提示

- 随机指令的产生 ：rand() 或 srand()
- 用户内存中页面控制结构采用链表

```
struct p_str{
int pagenum; /* 页号 */
int count; /* 访问页面的次数 */
struct p_str next; /* 下一指针 */
}p_str;
```

1.2 算法描述

- 最近最久未使用置换算法(LRU)：LRU 算法的基本思想是：当内存空间不足，需要换出一个页面时，选择最近最久没有被访问的页面。即，替换掉在最近一段时间内最少被使用的页面。它假设如果某个页面最近被访问过，那么它在未来一段时间内也可能会被再次访问，因此优先保留最近访问过的页面。这种算法可以改进 FIFO 性能较差的问题
- 改进的 Clock 置换算法：将一个页面换出时，如果该页已经被修改过，便需要将该页重新写回磁盘上；但是如果该页未被修改过，则不必将其拷贝回磁盘。换而言之，对于修改过的页面，在换出时所付出的代价比未修改过的页面大。在改进型 Clock

置换算法中，除了需要考虑页面大使用情况外，还需要考虑增加一个因素——置换代价。这样在页面换出时候，既要是未使用过的页面，也要是未被修改过的页面。把同事满足这两个条件的页面作为首要淘汰的页面，由访问位 A 和修改未 M 可以组成以下四种情况：

1:(A=0,M=0)表示该页最近既未被访问，又未被修改，是最佳淘汰页

2:(A=0,M=1)表示该页最近未被访问，但已经被修改，并不是很好的淘汰页

3:(A=1,M=0)表示最近已被访问，但未被修改，有可能再次被访问

4:(A=1,M=1)表示最近已被访问且被修改，可能再被访问

1.3 实验结果

- 当页面大小增大时，不同内存容量下的命中率有上升的趋势(但不一定绝对的单调递增)。
- LRU 和改进型 Clock 置换算法的命中率相差不大，但在内存空间较大时，改进型 Clock 置换算法的效率回略高于 LRU。

```
fang50253@MacBook_Pro 3.存储管理 % cd "/Users/fang50253/Desktop/Files/Documents/NJFU_Training/NJFU_CS_Training/操
g++ -std=c++14 03提交带注释.cpp -o 03提交带注释 && "/Users/fang50253/Desktop/Files/Documents/NJFU_Training/NJFU_C
存储管理/"03提交带注释
10

This is LRU:
Page_num=400,RAM=10Pages,success_rate=22.5 %


This is Improved_Clock:
Page_num=400,RAM=10Pages,success_rate=20.5 %


fang50253@MacBook_Pro 3.存储管理 % cd "/Users/fang50253/Desktop/Files/Documents/NJFU_Training/NJFU_CS_Training/操
g++ -std=c++14 03提交带注释.cpp -o 03提交带注释 && "/Users/fang50253/Desktop/Files/Documents/NJFU_Training/NJFU_C
存储管理/"03提交带注释
20

This is LRU:
Page_num=400,RAM=20Pages,success_rate=44.5 %


This is Improved_Clock:
Page_num=400,RAM=20Pages,success_rate=46.8 %


fang50253@MacBook_Pro 3.存储管理 % cd "/Users/fang50253/Desktop/Files/Documents/NJFU_Training/NJFU_CS_Training/操
g++ -std=c++14 03提交带注释.cpp -o 03提交带注释 && "/Users/fang50253/Desktop/Files/Documents/NJFU_Training/NJFU_C
存储管理/"03提交带注释
30

This is LRU:
Page_num=400,RAM=30Pages,success_rate=69.8 %


This is Improved_Clock:
Page_num=400,RAM=30Pages,success_rate=71.8 %


fang50253@MacBook_Pro 3.存储管理 % cd "/Users/fang50253/Desktop/Files/Documents/NJFU_Training/NJFU_CS_Training/操
g++ -std=c++14 03提交带注释.cpp -o 03提交带注释 && "/Users/fang50253/Desktop/Files/Documents/NJFU_Training/NJFU_C
存储管理/"03提交带注释
40

This is LRU:
Page_num=400,RAM=40Pages,success_rate=90.0 %


This is Improved_Clock:
Page_num=400,RAM=40Pages,success_rate=90.0 %
```

行 29

1.4 实现小结

- 在本次实习过程中，我直接使用了之前已经写好的 queue 类创建实例，但是在拷贝 queue 的对象的时候发生了错误，经过检查发现是因为在拷贝函数中我使用了浅拷贝，导致只拷贝了原来对象的指针，而原对象执行析构函数该指针成为了野指针，

导致了内存的越界访问。通过本次实习我了解了深拷贝和浅拷贝的区别。

特点	浅拷贝	深拷贝
拷贝内容	只拷贝对象的基本数据，指针成员指向同一地址	拷贝对象及其指针成员指向的数据，确保独立
内存共享	新对象和旧对象共享指针指向的内存	新对象和旧对象拥有独立的内存空间
性能	相对较快，因为只拷贝地址和简单的值	较慢，因为涉及到分配新内存和递归拷贝
使用场景	对象数据相对独立或不涉及指针数据时使用	对象间数据必须完全独立时使用

- LRU 算法通过追踪页面的使用顺序，将最久未使用的页面淘汰出去，这种方法简单且直观，但在实现时需要频繁更新每个页面的访问时间，可能导致效率低下。改进型 Clock 算法则在 LRU 的基础上进行优化，通过模拟时钟指针来追踪页面的使用状态，每次替换时只需扫描一次页面，减少了频繁更新的开销，提升了性能。
- LRU 虽然直观，但效率不高，而改进型 Clock 算法则通过优化时钟指针的方式提高了性能，展现了算法设计中不断迭代和改进的重要性。

1.5 实验代码

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>
#include<iostream>
#define MAXSIZE 400
namespace fzy//创建命名空间fzy，用于写一些数据结构
{
    template<typename T>
    struct queuenode //创建队列结点
    {
        T v = T();//提供队列结点的默认构造方法
        queuenode<T>* next = nullptr;//指向下一个位置，并提供默认构造方法
    };
    template <typename T>//创建队列模板类
    class queue
    {
    private:
        queuenode<T>* head;
        queuenode<T>* last;
        int size_;//队列的大小
    public:
        queue() //队列模板的默认构造方法
        {
            head = new queuenode<T>;
            last = head;
            size_ = 0;
        }
```

```
29.         queue(const queue<T>& other) //提供带参的构造函数
30.         {
31.             head = new queuenode<T>;
32.             last = head;
33.             size_ = 0;
34.             queuenode<T>* current = other.head->next;
35.             while (current != nullptr)
36.             {
37.                 push(current->v);
38.                 current = current->next;
39.             }
40.         }
41.         queue<T>& operator=(const queue<T>& other)//重载队列拷贝函数，深拷贝
42.         {
43.             if (this != &other)
44.             {
45.                 clear();
46.                 queuenode<T>* current = other.head->next;
47.                 while (current != nullptr)
48.                 {
49.                     push(current->v);
50.                     current = current->next;
51.                 }
52.             }
53.             return *this;
54.         }
55.         ~queue() //析构函数
56.         {
57.             clear();
58.             delete head;
59.         }
60.         void clear() //清空队列
61.         {
62.             while (!empty()) pop();
63.         }
64.         void push(T v) //将元素压入队列
65.         {
66.             queuenode<T>* newnode = new queuenode<T>;//创建一个新的结点
67.             newnode->v = v;
68.             last->next = newnode;
69.             last = newnode;
70.             ++size_;
71.         }
72.         void pop() //弹出
```

```
73.         {
74.             if (!empty())
75.             {
76.                 queuenode<T>* temp = head->next;
77.                 head->next = temp->next;
78.                 if (head->next == nullptr) last = head;
79.                 delete temp;
80.                 --size_;
81.             }
82.             else
83.             throw std::runtime_error("queue_empty");
84.         }
85.         T front() const //返回队列队首
86.         {
87.             if (!empty()) return head->next->v;
88.             throw std::runtime_error("Queue is empty");
89.         }
90.         T back() const //返回队列队尾
91.         {
92.             if (!empty()) return last->v;
93.             throw std::runtime_error("Queue is empty");
94.         }
95.         bool empty() const //队列判空
96.         {
97.             return size_ == 0;
98.         }
99.         int size() const //返回队列的长度
100.         {
101.             return size_;
102.         }
103.         void print() const //输出队列，用于调试
104.         {
105.             queuenode<T>* current = head->next;
106.             while (current != nullptr)
107.             {
108.                 std::cout << current->v << " ";
109.                 current = current->next;
110.             }
111.             std::cout << std::endl;
112.         }
113.     };
114. }
115. struct p_str
116. {
```

```
117.      int pagenum;     // 页号
118.      int count;       // 页面访问次数
119.      int clock_visit; // 改进 Clock 的访问标记
120.      int clock_revise;// 改进 Clock 的修改标记
121.      int LRU_time;    // LRU 的未使用时间
122.      p_str(int pagenum = -1, int count = -1, int clock_visit = -1, int clock_r
    evise = -1, int LRU_time = -1)
123.          : pagenum(pagenum), count(count), clock_visit(clock_visit), clock_rev
    ise(clock_revise), LRU_time(LRU_time) {}//提供构造方法
124.  };
125.  class storage
126.  {
127.  private:
128.      int n;                          // 内存大小
129.      double effective_times;         // 命中次数
130.      fzy::queue<p_str> clock;        // 改进 Clock 队列
131.      fzy::queue<p_str> lru_queue;  // LRU 模拟队列（用于访问）
132.      void init()
133.      {
134.          effective_times = 0;
135.          while (!fifo.empty()) fifo.pop();//初始化每一个数据结构
136.          while (!clock.empty()) clock.pop();//初始化每一个数据结构
137.          while (!lru_queue.empty()) lru_queue.pop();//初始化每一个数据结构
138.      }
139.      void print(const char way[],int Page_num,int RAM,double success)
140.      {
141.          printf("This is %s:\nPage_num=%d,RAM=%dPages,success_rate=%.1lf %%\n\
    n\n",way,Page_num,RAM,success/4);
142.          //输出结果
143.      }
144.  public:
145.      storage(int n) : n(n) {}//创建默认构造函数
146.      void LRU()
147.      {
148.          init();
149.          for (int i = 0; i < MAXSIZE; ++i)
150.          {
151.              int t = 0;
152.              int page = rand() % 400 / 10;
153.              // 检查 LRU 队列是否命中
154.              fzy::queue<p_str> temp = lru_queue;
155.              fzy::queue<p_str> new_queue; // 用于构建更新后的队列
156.              while (!temp.empty())
157.              {
```

```
158.                 p_str current = temp.front();
159.                 temp.pop();
160.                 if (current.pagenum == page)
161.                 {
162.                     ++t;
163.                     ++effective_times;
164.                     current.LRU_time = 0; // 重置未访问时间
165.                 }
166.                 else ++current.LRU_time; // 更新未访问时间
167.                 new_queue.push(current);
168.             }
169.             lru_queue = new_queue;
170.             // 未命中则置换
171.             if (!t)
172.             {
173.                 if ((int)lru_queue.size() < n) //队列没有满
174.                 lru_queue.push(p_str(page, 1, -1, -1, 0));
175.                 else //队列满了，找到最久没有使用的页面
176.                 {
177.                     // 找到最久未使用页面
178.                     fzy::queue<p_str> temp = lru_queue;
179.                     p_str max_page;
180.                     int max_time = -1;
181.                     while (!temp.empty())
182.                     {
183.                         p_str current = temp.front();
184.                         temp.pop();
185.                         if (current.LRU_time > max_time)
186.                         {
187.                             max_time = current.LRU_time;
188.                             max_page = current;
189.                         }
190.                     }
191.                     fzy::queue<p_str> new_queue;
192.                     temp = lru_queue;
193.                     while (!temp.empty())
194.                     {
195.                         p_str current = temp.front();
196.                         temp.pop();
197.                         if (current.pagenum != max_page.pagenum) new_queue.pu
    sh(current);
198.                     }
199.                     new_queue.push(p_str(page,1,-1,-1,0));
200.                     lru_queue = new_queue;
```

```
                }
            }
        }
        print("LRU",MAXSIZE,n,effective_times);//输出结果
    }
    void Improved_Clock()
    {
        init();
        for (int i = 0; i < MAXSIZE; ++i)
        {
            int t = 0;
            int page = rand() % 400 / 10;
            int m = rand()>>1;
            // 检查 Clock 队列是否命中
            fzy::queue<p_str>temp=clock;
            while (!temp.empty())
            {
                if (temp.front().pagenum == page)
                {
                    t = 1;
                    ++effective_times;
                    p_str cur = temp.front(); // 获取队首元素
                    cur.clock_visit = 1;      // 修改访问标记
                    temp.pop();               // 弹出旧的队首
                    temp.push(cur);           // 将修改后的节点重新加入队列
                    break;
                }
                temp.pop();
            }
            // 未命中则置换
            if(!t)
            {
                if ((int)clock.size() < n) clock.push(p_str(page, 1, 1, m));//
  //队列没有满
                else //队列满了，则只能置换
                {
                    // 查找替换目标
                    while (true)
                    {
                        p_str cur = clock.front();
                        clock.pop();
                        if (cur.clock_visit == 0)
                        {
                            clock.push(p_str(page, 1, 1, m));
```

```
                                break;
                            }
                            else
                            {
                                cur.clock_visit = 0;
                                clock.push(cur); // 重置访问标记，放回队列末尾
                            }
                        }
                    }
                }
            }
            print("Improved_Clock",MAXSIZE,n,effective_times);//输出
        }
};
int main()
{
    int n;
    srand((unsigned)time(NULL));
    scanf("%d", &n);
    printf("\n\n\n");
    storage storage(n);
    storage.LRU();//最近最久未使用
    storage.Improved_Clock();//改进的时钟
    printf("\n\n\n");
    return 0;
}
```