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：
g++ -std=c++14 01Process.cpp -o 01Process
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.      #define MAX_PROCESS_NUMBER 100
6.      namespace fzy//创建命名空间fzy，用于数据结构的实现
7.      {
8.          template<class T>//创建模板类
```

```
9.          class Less//创建比较函数
10.         {
11.         public:
12.             bool operator()(const T& x, const T& y)
13.             {
14.                 return x < y;
15.             }
16.         };
17.         template<class T>
18.         class Greater//创建比较函数
19.         {
20.         public:
21.             bool operator()(const T& x, const T& y)
22.             {
23.                 return x > y;
24.             }
25.         };
26.         template<typename T>
27.         struct queuenode//创建队列的节点
28.         {
29.             T v=T();
30.             queuenode<T>* next;
31.         };
32.         template<typename T>//创建队列数据结构
33.         class queue
34.         {
35.         private:
36.             queuenode<T>* head;
37.             queuenode<T>* last;
38.             int size_;
39.         public:
40.             queue()//队列的构造函数
41.             {
42.                 head=new queuenode<T>;
43.                 head->next=nullptr;
44.                 last=head;
45.                 size_=0;
46.             }
47.             ~queue()//队列的析构函数
48.             {
49.                 clear();
50.                 delete head;
51.             }
52.             void clear()//清空队列
```

```
53.             {
54.                 while(!empty()) pop();
55.             }
56.             void push(T v)//压入队列
57.             {
58.                 queuenode<T>* newnode = new queuenode<T>;
59.                 newnode->v = v;
60.                 newnode->next = nullptr;
61.                 last->next = newnode;
62.                 last = newnode;
63.                 ++size_;
64.             }
65.             void pop()//弹出队列
66.             {
67.                 if (size_>0)
68.                 {
69.                     queuenode<T>* temp=head->next;
70.                     head->next=temp->next;
71.                     if (head->next==nullptr) last=head;
72.                     delete temp;
73.                     --size_;
74.                 }
75.             }
76.             T front()//取队首
77.             {
78.                 if(size_>0) return head->next->v;
79.                 throw std::runtime_error("Queue is empty");//如果队列为空则抛出
    异常
80.             }
81.             T back()//取队尾
82.             {
83.                 if(size_>0) return last->v;
84.                 throw std::runtime_error("Queue is empty");//如果队列为空则抛出
    异常
85.             }
86.             bool empty()//队列的判空操作
87.             {
88.                 return size_==0;
89.             }
90.             int size()//返回队列的长度
91.             {
92.                 return size_;
93.             }
94.             void print()//定义为按照队列的入(出)队顺序进行打印
```

```
95.             {
96.                 queuenode<T> *i;
97.                 int j;
98.                 for(i=head->next,j=1;j<=size_;++j,i=i->next)
99.                 {
100.                    std::cout<<i->v<<" ";
101.                }
102.                std::cout<<std::endl;
103.            }
104.        };
105.        template<typename T>
106.        struct stacknode//定义为栈的结点
107.        {
108.            T v;//使用模板类使其支持任意类型
109.            stacknode *next;
110.        };
111.        template<typename T>//模板类
112.        class stack
113.        {
114.            private:
115.            stacknode<T> *head;//链栈的头结点
116.            int size_;
117.            public:
118.            stack()//链栈的构造方法
119.            {
120.                head=new stacknode<T>;//创建头结点
121.                head->next = nullptr;
122.                size_=0;
123.            }
124.            ~stack()//栈的析构函数
125.            {
126.                clear();
127.            }
128.            void clear()//清空链栈
129.            {
130.                while(size()) pop();
131.            }
132.            void pop()//将栈顶元素弹出
133.            {
134.                if(size_==0) ;//throw std::runtime_error("stack is empty");//
    栈为空则抛出异常
135.                stacknode<T>* tmp=head->next;//弹出操作
136.                head->next=head->next->next;
137.                --size_;
```

```
138.                 delete tmp;//释放内存
139.             }
140.             int size()//返回栈的长度
141.             {
142.                 return size_;
143.             }
144.             bool empty()//栈的判空函数
145.             {
146.                 return size_==0;
147.             }
148.             T top()//返回栈顶元素
149.             {
150.                 if(size_==0) ;//throw std::runtime_error("stack is empty");//
    如果栈为空则抛出异常
151.                 return head->next->v;
152.             }
153.             void push(T v)//压入栈
154.             {
155.                 stacknode<T> *newnode=new stacknode<T>;
156.                 newnode->next=head->next;
157.                 head->next=newnode;
158.                 newnode->v=v;
159.                 ++size_;
160.             }
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.         };
174.         template<typename T>//使用模板类定义优先队列
175.         class priority_queue{
176.         public:
177.             priority_queue() //优先队列的无参数构造函数
178.             :size_of_priority_queue(0), capacity(MAX_PROCESS_NUMBER), compare(
    &_compare)
179.             {
```

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

```
216.             {
217.                 T *ptt = pt;
218.                 if( size_of_priority_queue == capacity)
219.                 {
220.                     capacity *= 2;
221.                     pt = new T[capacity];
222.                     if( nullptr == pt )
223.                     {
224.                         pt = ptt;
225.                         capacity /= 2;
226.                         return false;
227.                     }
228.                     obj_cpy(pt, ptt, size_of_priority_queue);
229.                     delete[] ptt;
230.                 }
231.                 pt[size_of_priority_queue++] = t;
232.                 heap_up();//堆堆上传操作
233.                 return true;
234.             }
235.             bool pop()
236.             {
237.                 if(size_of_priority_queue==0) return 0;
238.                 if(size_of_priority_queue==1)
239.                 {
240.                     size_of_priority_queue = 0;
241.                     return 1;
242.                 }
243.                 pt[0] = pt[size_of_priority_queue-1];
244.                 size_of_priority_queue--;
245.                 heap_down();//堆堆下传操作
246.                 return 1;
247.             }
248.             T top()
249.             {
250.                 if(size_of_priority_queue<0) ;//throw std::runtime_error("queu
    e empty");//优先队列为空则抛出异常
251.                 return pt[0];//返回队头元素
252.             }
253.             bool is_empty_pl()const
254.             {
255.                 return 0==size_of_priority_queue;//返回队是否为空
256.             }
257.             int get_size()const
258.             {
```

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

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

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

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

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

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

```
                    }
                    else doing_process.push(doing);//将进程继续送回正在执行的队列

                }

            }
            void display()//显示函数
            {
                printf("Process name\t");
                printf("Finished time\t");
                printf("Turnaround time\t");
                printf("Weighted turnaround time\t\n");
                for(int i=1;i<=finished_index;++i)
                {
                    printf("|\t%c\t|\t",finished[i].process_name);//输出进程名称
                    printf("%d\t|\t",finished[i].time_end);//输出进程完成时间
                    printf("%d\t|\t",finished[i].time_turnaround);//周转时间
                    printf("%.2lf\t|\n",finished[i].time_turnaround_rights);//带权周转时间
                }
            }
            void display_avergae()
            {
                double average_time_turnaround=0;//计算平均周转时间
                double average_time_turnaround_rights=0;//计算平均带权周转时间
                for(int i=1;i<=finished_index;++i)
                {
                    average_time_turnaround+=finished[i].time_turnaround;
                    average_time_turnaround_rights+=finished[i].time_turnaround_rights;
                }
                average_time_turnaround/=finished_index;
                average_time_turnaround_rights/=finished_index;
                printf("avergae turnaround time=%.2lf\n",average_time_turnaround);//输出平均周转时间
                printf("average weighted turnaround time=%.2lf",average_time_turnaround_rights);//输出平均带权周转时间
            }
        };
        class SJF//抢占式短进程优先算法
        {
            private:
            void read(const char filename[])//读取文件
```

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

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

```
628.                 {
629.                     average_time_turnaround+=finished[i].time_turnaround;
630.                     average_time_turnaround_rights+=finished[i].time_turnaroun
    d_rights;
631.                 }
632.                 average_time_turnaround/=finished_index;
633.                 average_time_turnaround_rights/=finished_index;
634.                 printf("avergae turnaround time=%.2lf\n",average_time_turnarou
    nd);
635.                 printf("average weighted turnaround time=%.2lf",average_time_t
    urnaround_rights);
636.             }
637.         };
638.     }
639.     int main()
640.     {
641.         printf("\n\nThis is SJF:\n\n\n");//使用新来进程优先的算法
642.         os::SJF *sjf=new os::SJF();
643.         sjf->conduct();
644.         sjf->display();
645.         sjf->display_avergae();
646.         delete(sjf);
647.         printf("\n\n\n");
648.         printf("\n\nThis is RR:\n\n\n");//使用新来进程优先的算法
649.         os::RR *rr=new os::RR();
650.         rr->conduct();
651.         rr->display();
652.         rr->display_avergae();
653.         delete(rr);
654.         printf("\n\n\n");
655.         return 0;
656.     }
```

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

- 整型信号量：wait(S)和 signal(S)是两个原子操作，因此，它们在执行时是不可中断的。亦即，当一个进程在修改某信号量时，没有其它进程可同时对该信号量进行修改。此外，在 wait 操作中，对 S 值的测试和做 S=S-1 操作时都不可中断。
- 记录型信号量：记录型信号量是一种不存在“忙等”现象的进程同步机制。除了需要一个用于代表资源数目的整型变量 value 外，再增加一个进程链表 L，用于链接所有等待该资源的进程，记录型信号量得名于采用记录型的数据结构。
- 使用信号量实现线程同步：信号量机制能用于解决进程间的各种同步问题。设 S 为实现进程 P1，P2 同步的公共信号量，初始值为 0。进程 P2 中的语句 y 要使用进程 P1 中的语句 x 的运行结果，所以只有当语句 x 执行完成之后，语句 y 才可以执行。
- 利用信号量实现进程互斥：信号量机制能很方便地解决进程互斥问题。设 S 为实现进程 P1，P2 互斥的信号量，由于每次只允许一个进程进入临界区，所以 S 的初始值应为 1(即可用资源数为 1)。只需要把临界区置于 P(S)和 V(S)之间，即可实现两个进程对临界资源的互斥访问。

1.3 实验结果

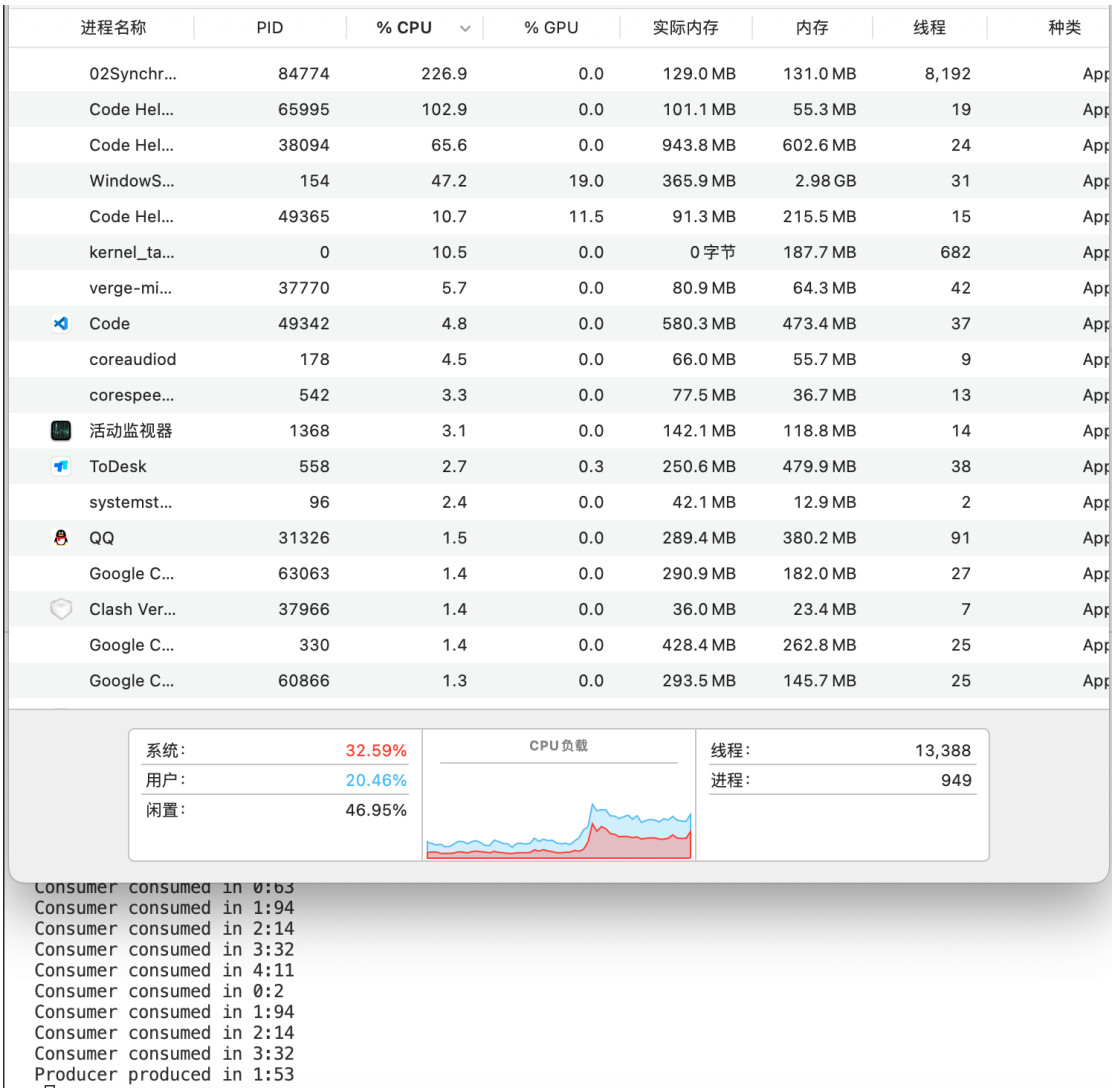
此程序在 macOS14.6.1 系统，采用如下编译指令：
g++ -std=c++14 02Synchronization_Mutex.cpp -o 02Synchronization_Mutex
运行结果的截图如下：

```
Producer produced in 0:8
Consumer consumed in 1:8
Producer produced in 1:10
Producer produced in 2:66
Producer produced in 3:43
Producer produced in 4:4
Consumer consumed in 3:66
Producer produced in 0:41
Producer produced in 1:4
Consumer consumed in 4:43
Producer produced in 2:58
Producer produced in 3:34
Consumer consumed in 0:4
Consumer consumed in 1:41
Consumer consumed in 2:4
Consumer consumed in 3:58
Producer produced in 4:98
Producer produced in 0:68
Producer produced in 1:34
Consumer consumed in 4:34
```

生产者先生产了 8 个产品在 0 号位置，消费者消费了。生产者再在 1 2 3 4 号位置分别生产了 10 66 43 4 个产品，然后消费者消费了 3 号位置中的所有产品，以此类推……

1.4 实现小结

- 在本次实习的过程中，我学习到了通过 c 语言中的 pthread.h 库创建进程的操作，学习到了通过信号量 semaphore 库实现进程同步的操作。
- 实习代码一开始我没有加上 sleep(1);等待语句，导致程序一开始创建了很多进程，在发现代码运行出现问题之后，我才加上了运行次数的限制和运行时间的等待。



1.5 实验代码

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>
#define MAX_BUFFER 5
int buffer[MAX_BUFFER];//缓冲
int in=0,out=0;
sem_t s1;//控制缓冲区未满
sem_t s2;//控制缓冲区非空
```

```
11.     sem_t mutex;//控制对临界区的访问
12.     void *producer(void *arg)//生产者进程
13.     {
14.         int item;
15.         while(1)
16.         {
17.             item=rand()%100+1;
18.             printf("Producer produced in %d:%d\n",in,item);
19.             sem_wait(&s1);
20.             sem_wait(&mutex);
21.             buffer[in]=item;
22.             in=(in+1)%MAX_BUFFER;
23.             sem_post(&mutex);
24.             sem_post(&s2);
25.             sleep(rand()%2);
26.         }
27.         return NULL;
28.     }
29.     void *consumer(void *arg)//消费者进程
30.     {
31.         int item;
32.         while(1)
33.         {
34.             sem_wait(&s2);
35.             sem_wait(&mutex);
36.             item=buffer[out];
37.             out=(out+1)%MAX_BUFFER;
38.             if(item!=0) printf("Consumer consumed in %d:%d\n",out,it
    em);
39.             sem_post(&mutex);
40.             sem_post(&s1);
41.             sleep(rand()%2);
42.         }
43.         return NULL;
44.     }
45.     int main()
46.     {
47.         pthread_t prod,cons;
48.         sem_init(&s1,0,MAX_BUFFER);//初始化缓冲区未满信号量
49.         sem_init(&s2,0,0);//初始化缓冲区非空信号量
50.         sem_init(&mutex,0,1);//初始化临界区访问信号量
51.         for(int i=1;i<=3;++i)//只运行3次
52.         {
53.             pthread_create(&prod,NULL,producer,NULL);//创建生产者进程
```

```
54.             pthread_create(&cons,NULL,consumer,NULL);//创建消费者进程
55.             sleep(1);//防止运行太快
56.         }
57.         sem_destroy(&s1);//删除信号量
58.         sem_destroy(&s2);
59.         sem_destroy(&mutex);
60.         return 0;
61.     }
```

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。

此程序在 macOS14.6.1 系统，采用如下编译指令：

g++ -std=c++14 03Storage.cpp -o 03Storage

运行结果的截图如下：

```
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>
#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
```

```
23.             {
24.                 head = new queuenode<T>;
25.                 last = head;
26.                 size_ = 0;
27.             }
28.             queue(const queue<T>& other) //提供带参的构造函数
29.             {
30.                 head = new queuenode<T>;
31.                 last = head;
32.                 size_ = 0;
33.                 queuenode<T>* current = other.head->next;
34.                 while (current != nullptr)
35.                 {
36.                     push(current->v);
37.                     current = current->next;
38.                 }
39.             }
40.             queue<T>& operator=(const queue<T>& other)//重载队列拷贝
    函数，深拷贝
41.             {
42.                 if (this != &other)
43.                 {
44.                     clear();
45.                     queuenode<T>* current = other.head->next;
46.                     while (current != nullptr)
47.                     {
48.                         push(current->v);
49.                         current = current->next;
50.                     }
51.                 }
52.                 return *this;
53.             }
54.             ~queue() //析构函数
55.             {
56.                 clear();
57.                 delete head;
58.             }
59.             void clear() //清空队列
60.             {
61.                 while (!empty()) pop();
62.             }
63.             void push(T v) //将元素压入队列
64.             {
```

```
65.                 queuenode<T>* newnode = new queuenode<T>;//创建一个
    新的结点
66.                 newnode->v = v;
67.                 last->next = newnode;
68.                 last = newnode;
69.                 ++size_;
70.             }
71.             void pop() //弹出
72.             {
73.                 if (!empty())
74.                 {
75.                     queuenode<T>* temp = head->next;
76.                     head->next = temp->next;
77.                     if (head->next == nullptr) last = head;
78.                     delete temp;
79.                     --size_;
80.                 }
81.                 else
82.                 ;//throw std::runtime_error("queue_empty");
83.             }
84.             T front() const //返回队列队首
85.             {
86.                 if (!empty()) return head->next->v;
87.                 //throw std::runtime_error("Queue is empty");
88.             }
89.             T back() const //返回队列队尾
90.             {
91.                 if (!empty()) return last->v;
92.                 //throw std::runtime_error("Queue is empty");
93.             }
94.             bool empty() const //队列判空
95.             {
96.                 return size_ == 0;
97.             }
98.             int size() const //返回队列的长度
99.             {
100.                return size_;
101.            }
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.                {
```

```
                    std::cout << current->v << " ";
                    current = current->next;
                }
                std::cout << std::endl;
            }
            */
        };
    }
    struct p_str
    {
        int pagenum;     // 页号
        int count;       // 页面访问次数
        int clock_visit; // 改进 Clock 的访问标记
        int clock_revise;// 改进 Clock 的修改标记
        int LRU_time;    // LRU 的未使用时间
        p_str(int pagenum = -1, int count = -1, int clock_visit = -1
, int clock_revise = -1, int LRU_time = -1)
            : pagenum(pagenum), count(count), clock_visit(clock_visi
t), clock_revise(clock_revise), LRU_time(LRU_time) {}//提供构造方法
    };
    class storage
    {
    private:
        int n;                          // 内存大小
        double effective_times;         // 命中次数
        fzy::queue<p_str> fifo;         // FIFO 队列
        fzy::queue<p_str> clock;        // 改进 Clock 队列
        fzy::queue<p_str> lru_queue;  // LRU 模拟队列（用于访问）
        void init()
        {
            effective_times = 0;
            while (!fifo.empty()) fifo.pop();//初始化每一个数据结构
            while (!clock.empty()) clock.pop();//初始化每一个数据结构
            while (!lru_queue.empty()) lru_queue.pop();//初始化每一个
数据结构
        }
        void print(const char way[],int Page_num,int RAM,double succ
ess)
        {
            printf("This is %s:\nPage_num=%d,RAM=%dPages,success_rat
e=%.1lf %%\n\n\n",way,Page_num,RAM,success/4);
            //输出结果
        }
    public:
```

```
147.        storage(int n) : n(n) {}//创建默认构造函数
148.        void FIFO()
149.        {
150.            init();//初始化
151.            for (int i = 0; i < MAXSIZE; ++i) {
152.                int t = 0;
153.                int page=rand()%400/10;// 检查 FIFO 队列是否命中
154.                fzy::queue<p_str>temp=fifo;
155.                while (!temp.empty())
156.                {
157.                    if (temp.front().pagenum == page) //在页表中
158.                    {
159.                        ++t;
160.                        ++effective_times;//有效
161.                        break;
162.                    }
163.                    temp.pop();
164.                }
165.                // 未命中则进行置换
166.                if (!t)
167.                {
168.                    if ((int)fifo.size() < n) fifo.push(p_str(page,
    1));//队列没有满
169.                    else
170.                    {
171.                        fifo.pop();             // 移除队首页面
172.                        fifo.push(p_str(page, 1)); // 加入新页面
173.                    }
174.                }
175.            }
176.            print("FIFO",MAXSIZE,n,effective_times);//输出
177.        }
178.        void LRU()
179.        {
180.            init();
181.            for (int i = 0; i < MAXSIZE; ++i)
182.            {
183.                int t = 0;
184.                int page = rand() % 400 / 10;
185.                // 检查 LRU 队列是否命中
186.                fzy::queue<p_str> temp = lru_queue;
187.                fzy::queue<p_str> new_queue; // 用于构建更新后的队列
188.                while (!temp.empty())
189.                {
```

```
                    p_str current = temp.front();
                    temp.pop();
                    if (current.pagenum == page)
                    {
                        ++t;
                        ++effective_times;
                        current.LRU_time = 0; // 重置未访问时间
                    }
                    else ++current.LRU_time; // 更新未访问时间
                    new_queue.push(current);
                }
                lru_queue = new_queue;
                // 未命中则置换
                if (!t)
                {
                    if ((int)lru_queue.size() < n) //队列没有满
                    lru_queue.push(p_str(page, 1, -1, -1, 0));
                    else //队列满了，找到最久没有使用的页面
                    {
                        // 找到最久未使用页面
                        fzy::queue<p_str> temp = lru_queue;
                        p_str max_page;
                        int max_time = -1;
                        while (!temp.empty())
                        {
                            p_str current = temp.front();
                            temp.pop();
                            if (current.LRU_time > max_time)
                            {
                                max_time = current.LRU_time;
                                max_page = current;
                            }
                        }
                        fzy::queue<p_str> new_queue;
                        temp = lru_queue;
                        while (!temp.empty())
                        {
                            p_str current = temp.front();
                            temp.pop();
                            if (current.pagenum != max_page.pagenum)
    new_queue.push(current);
                        }
                        new_queue.push(p_str(page,1,-1,-1,0));
                        lru_queue = new_queue;
```

```
233.                     }
234.                 }
235.             }
236.             print("LRU",MAXSIZE,n,effective_times);//输出结果
237.         }
238.         void Improved_Clock()
239.         {
240.             init();
241.             for (int i = 0; i < MAXSIZE; ++i)
242.             {
243.                 int t = 0;
244.                 int page = rand() % 400 / 10;
245.                 int m = rand()>>1;
246.                 // 检查 Clock 队列是否命中
247.                 fzy::queue<p_str>temp=clock;
248.                 while (!temp.empty())
249.                 {
250.                     if (temp.front().pagenum == page)
251.                     {
252.                         t = 1;
253.                         ++effective_times;
254.                         p_str cur = temp.front(); // 获取队首元素
255.                         cur.clock_visit = 1;      // 修改访问标记
256.                         temp.pop();               // 弹出旧的队首
257.                         temp.push(cur);           // 将修改后的节点重
    新加入队列
258.                         break;
259.                     }
260.                     temp.pop();
261.                 }
262.                 // 未命中则置换
263.                 if(!t)
264.                 {
265.                     if ((int)clock.size() < n) clock.push(p_str(page
    , 1, 1, m));//队列没有满
266.                     else //队列满了，则只能置换
267.                     {
268.                         // 查找替换目标
269.                         while (true)
270.                         {
271.                             p_str cur = clock.front();
272.                             clock.pop();
273.                             if (cur.clock_visit == 0)
274.                             {
```

```
                                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");
        storage storage(n);
        storage.FIFO();//先进先出
        storage.LRU();//最近最久未使用
        storage.Improved_Clock();//改进的时钟
        return 0;
    }
```