

**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

1.1 实验内容 1

1.2算法描述 1

1.3 实验结果 1

1.4 实现小结 3

1.5实验代码 3

题目：进程的同步与互斥 20

1.1 实验内容 20

1.2算法描述 21

1.3 实验结果 21

1.4 实现小结 21

1.5实验代码 21

题目：存储管理 22

1.1 实验内容 22

1.2算法描述 22

1.3 实验结果 23

1.4 实现小结 23

1.5实验代码 24

# 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

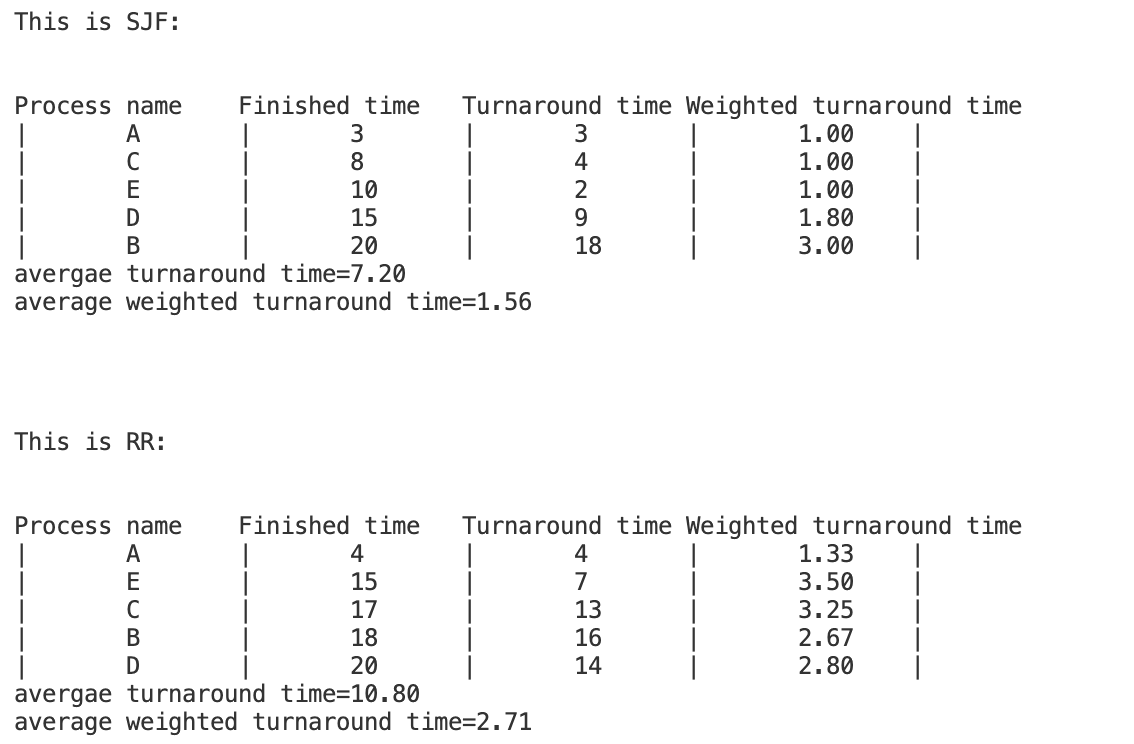
第18-19秒，执行程序D,剩余1/5

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：



## 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>
6. #define MAX\_PROCESS\_NUMBER 100
7. namespace fzy*//创建命名空间fzy，用于数据结构的实现*
8. {
9. template<class T>*//创建模板类*
10. class Less*//创建比较函数*
11. {
12. public:
13. bool operator()(const T& x, const T& y)
14. {
15. return x < y;
16. }
17. };
18. template<class T>
19. class Greater*//创建比较函数*
20. {
21. public:
22. bool operator()(const T& x, const T& y)
23. {
24. return x > y;
25. }
26. };
27. template<typename T>
28. struct queuenode*//创建队列的节点*
29. {
30. T v=T();
31. queuenode<T>\* next;
32. };
33. template<typename T>*//创建队列数据结构*
34. class queue
35. {
36. private:
37. queuenode<T>\* head;
38. queuenode<T>\* last;
39. int size\_;
40. public:
41. queue()*//队列的构造函数*
42. {
43. head=new queuenode<T>;
44. head->next=nullptr;
45. last=head;
46. size\_=0;
47. }
48. ~queue()*//队列的析构函数*
49. {
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), compare(&\_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), compare(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), compare(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 empty");*//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. template<typename T>
282. void priority\_queue<T>::heap\_up()*//上传操作，基于堆*
283. {
284. T temp;
285. int itr = size\_of\_priority\_queue-1;
286. while( itr > 0 )
287. {
288. if( (compare(pt[itr/2], pt[itr])))
289. {
290. temp = pt[itr];
291. pt[itr] = pt[itr/2];
292. pt[itr/2] = temp;
293. itr = itr/2;
294. continue;
295. }
296. break;
297. }
298. return;
299. }
300. template<typename T>
301. void priority\_queue<T>::heap\_down()*//下传操作，基于堆*
302. {
303. T temp;
304. int pitr = 0, citr;
305. while(pitr<=size\_of\_priority\_queue/2-1)
306. {
307. citr = pitr \* 2 + 1;
308. if(citr+1<size\_of\_priority\_queue&&compare(pt[citr],pt[citr+1])) ++citr;
309. if((compare(pt[pitr],pt[citr])))
310. {
311. temp = pt[citr];
312. pt[citr] = pt[pitr];
313. pt[pitr] = temp;
314. pitr = citr;*//继续将pitr指向孩子节点，进行下一次的比较*
315. continue;
316. }
317. break;*//如果处在对的位置，直接结束，不需要继续比较下去了*
318. }
319. return;
320. }
321. }
322. namespace os*//定义命名空间os，用于进程调度的书写*
323. {
324. class Process*//进程，重载运算符实现时间升序排序*
325. {
326. public:
327. char process\_name;*//进程名称*
328. int time\_arrive;*//到达时间*
329. int time\_serve;*//服务时间*
330. 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*
331. {
332. return time\_arrive>W.time\_arrive;
333. }
334. bool operator <= (const Process &W) const *//因为算法中使用到了<=号*
335. {
336. return time\_arrive>=W.time\_arrive;
337. }
338. bool operator > (const Process &W) const
339. {
340. return time\_arrive<W.time\_arrive;
341. }
342. bool operator >= (const Process &W) const
343. {
344. return time\_arrive<=W.time\_arrive;
345. }
346. 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*
347. {
348. this->process\_name=process\_name;
349. this->time\_arrive=time\_arrive;
350. this->time\_serve=time\_serve;
351. }
352. Process()*//默认构造函数*
353. {
354. this->process\_name=0;
355. this->time\_arrive=0;
356. this->time\_serve=0;
357. }
358. };
359. class Process\_finish:public Process*//继承Process，用于SJF SPF算法中的中间计算过程*
360. {
361. public:
362. Process\_finish(Process a,int time\_end,int time\_turnaround,double time\_turnaround\_rights)*//用于接收Process的拷贝*
363. {
364. this->process\_name=a.process\_name;
365. this->time\_arrive=a.time\_arrive;
366. this->time\_serve=a.time\_serve;
367. this->time\_end=time\_end;
368. this->time\_turnaround=time\_turnaround;
369. this->time\_turnaround\_rights=time\_turnaround\_rights;
370. }
371. Process\_finish()*//默认构造函数，生成类数组*
372. {
373. this->process\_name=0;
374. this->time\_arrive=0;
375. this->time\_end=0;
376. this->time\_serve=0;
377. this->time\_turnaround=0;
378. this->time\_turnaround\_rights=0;
379. }
380. int time\_end;*//完成时间*
381. int time\_turnaround;*//周转时间*
382. double time\_turnaround\_rights;*//带权周转时间*
383. bool operator < (const Process &W) const *//重载<*
384. {
385. return time\_serve>W.time\_serve;
386. }
387. bool operator <= (const Process &W) const
388. {
389. return time\_serve>=W.time\_serve;
390. }
391. bool operator > (const Process &W) const
392. {
393. return time\_serve<W.time\_serve;
394. }
395. bool operator >= (const Process &W) const
396. {
397. return time\_serve<=W.time\_serve;
398. }
399. }finished[MAX\_PROCESS\_NUMBER];*//使用默认构造函数构造*
400. int finished\_index;
401. class Process\_remain:public Process
402. {
403. public:
404. int remain;
405. Process\_remain(class Process p)*//构造函数*
406. {
407. this->process\_name=p.process\_name;
408. this->time\_arrive=p.time\_arrive;
409. this->time\_serve=p.time\_serve;
410. this->remain=p.time\_serve;
411. }
412. Process\_remain()*//构造函数*
413. {
414. this->remain=0;
415. }
416. bool operator < (const Process &W) const *//重载<实现以服务时间升序排序*
417. {
418. return time\_serve>W.time\_serve;
419. }
420. bool operator <= (const Process &W) const
421. {
422. return time\_serve>=W.time\_serve;
423. }
424. bool operator > (const Process &W) const
425. {
426. return time\_serve<W.time\_serve;
427. }
428. bool operator >= (const Process &W) const
429. {
430. return time\_serve<=W.time\_serve;
431. }
432. };
433. class RR*//时间片轮转法*
434. {
435. private:
436. 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*
437. fzy::queue<Process\_remain>doing\_process;*//正在被执行的进程，在用一个队列去转*
438. int process\_cnt;
439. int time;
440. public:
441. void read(const char filename[])*//读取文件*
442. {
443. FILE \*fp=fopen(filename,"r+");
444. if(fp==NULL) throw std::runtime\_error("open file failed");
445. fscanf(fp,"%d",&process\_cnt);
446. for(int i=1;i<=process\_cnt;++i)
447. {
448. char process\_name;
449. int time\_arrive,time\_serve;
450. fscanf(fp," %c%d%d",&process\_name,&time\_arrive,&time\_serve);
451. process.push(Process(process\_name,time\_arrive,time\_serve));*//使用临时的类去赋值*
452. }
453. fclose(fp);*//文件读取完成*
454. }
455. RR()
456. {
457. process\_cnt=0;
458. finished\_index=0;
459. char filename[]="filename.dat";*//构造方法，读取文件*
460. read(filename);
461. memset(finished,0,sizeof finished);
462. finished\_index=0;
463. }
464. void conduct()*//执行RR时间片轮转*
465. {
466. time=0;
467. Process\_remain doing;
468. while(finished\_index!=process\_cnt)*//在所有进程完成之前*
469. {
470. Process top\_process;
471. if(!process.empty())*//如果进程不空就一直执行下去*
472. {
473. top\_process=process.top();
474. while(top\_process.time\_arrive<=time)
475. {
476. process.pop();
477. doing\_process.push(Process\_remain(top\_process));
478. *//printf("push:%c\n",top\_process.process\_name);*
479. top\_process=process.top();*//不断读取*
480. }
481. }
482. doing=doing\_process.front();
483. doing\_process.pop();*//执行这个进程*
484. --doing.remain;
485. *//printf("%c",doing.process\_name);*
486. ++time;
487. if(!process.empty())
488. {
489. top\_process=process.top();
490. while(top\_process.time\_arrive<=time)
491. {
492. process.pop();
493. doing\_process.push(Process\_remain(top\_process));
494. top\_process=process.top();
495. if(process.empty()) break;
496. }
497. }
498. if(doing.remain==0)*//这个进程已经执行完毕*
499. {
500. ++finished\_index;
501. finished[finished\_index].process\_name=doing.process\_name;
502. finished[finished\_index].time\_arrive=doing.time\_arrive;
503. finished[finished\_index].time\_end=time;
504. finished[finished\_index].time\_serve=doing.time\_serve;
505. finished[finished\_index].time\_turnaround=time-doing.time\_arrive;
506. finished[finished\_index].time\_turnaround\_rights=1.0\*finished[finished\_index].time\_turnaround/doing.time\_serve;
507. }
508. else doing\_process.push(doing);*//将进程继续送回正在执行的队列*
510. }
511. }
512. void display()*//显示函数*
513. {
514. printf("Process name\t");
515. printf("Finished time\t");
516. printf("Turnaround time\t");
517. printf("Weighted turnaround time\t\n");
518. for(int i=1;i<=finished\_index;++i)
519. {
520. printf("|\t%c\t|\t",finished[i].process\_name);*//输出进程名称*
521. printf("%d\t|\t",finished[i].time\_end);*//输出进程完成时间*
522. printf("%d\t|\t",finished[i].time\_turnaround);*//周转时间*
523. printf("%.2lf\t|\n",finished[i].time\_turnaround\_rights);*//带权周转时间*
524. }
525. }
526. void display\_avergae()
527. {
528. double average\_time\_turnaround=0;*//计算平均周转时间*
529. double average\_time\_turnaround\_rights=0;*//计算平均带权周转时间*
530. for(int i=1;i<=finished\_index;++i)
531. {
532. average\_time\_turnaround+=finished[i].time\_turnaround;
533. average\_time\_turnaround\_rights+=finished[i].time\_turnaround\_rights;
534. }
535. average\_time\_turnaround/=finished\_index;
536. average\_time\_turnaround\_rights/=finished\_index;
537. printf("avergae turnaround time=%.2lf\n",average\_time\_turnaround);*//输出平均周转时间*
538. printf("average weighted turnaround time=%.2lf",average\_time\_turnaround\_rights);*//输出平均带权周转时间*
539. }
540. };
541. class SJF*//抢占式短进程优先算法*
542. {
543. private:
544. void read(const char filename[])*//读取文件*
545. {
546. FILE \*fp=fopen(filename,"r+");
547. if(fp==NULL) throw std::runtime\_error("open file failed");
548. fscanf(fp,"%d",&process\_cnt);
549. for(int i=1;i<=process\_cnt;++i)
550. {
551. char process\_name;
552. int time\_arrive,time\_serve;
553. fscanf(fp," %c%d%d",&process\_name,&time\_arrive,&time\_serve);
554. process.push(Process(process\_name,time\_arrive,time\_serve));*//创建临时类去压入优先队列中*
555. }
556. fclose(fp);*//关闭文件*
557. }
558. int process\_cnt;
559. fzy::priority\_queue<Process>process;
560. fzy::priority\_queue<Process\_remain>doing\_process;
561. public:
562. SJF()*//默认构造方法*
563. {
564. process\_cnt=0;
565. finished\_index=0;
566. char filename[]="filename.dat";
567. read(filename);*//读取文件*
568. memset(finished,0,sizeof finished);
569. finished\_index=0;
570. }
571. void conduct()*//抢占式短进程优先*
572. {
573. int time=0;
574. Process\_remain doing;
575. while(!(process.empty()&&doing\_process.empty()))
576. {
577. Process top\_process;
578. if(!process.empty())
579. {
580. top\_process=process.top();
581. while(top\_process.time\_arrive<=time)
582. {
583. process.pop();
584. doing\_process.push(Process\_remain(top\_process));
585. top\_process=process.top();
586. if(process.empty()) break;
587. }
588. }
590. doing=doing\_process.top();
591. doing\_process.pop();
592. --doing.remain;++time;
593. if(doing.remain==0)
594. {
595. ++finished\_index;
596. finished[finished\_index].process\_name=doing.process\_name;
597. finished[finished\_index].time\_arrive=doing.time\_arrive;
598. finished[finished\_index].time\_end=time;
599. finished[finished\_index].time\_serve=doing.time\_serve;
600. finished[finished\_index].time\_turnaround=time-doing.time\_arrive;
601. finished[finished\_index].time\_turnaround\_rights=1.0\*finished[finished\_index].time\_turnaround/doing.time\_serve;
602. }
603. else doing\_process.push(doing);
604. }
605. }
606. void display()*//显示*
607. {
608. printf("Process name\t");
609. printf("Finished time\t");
610. printf("Turnaround time\t");
611. printf("Weighted turnaround time\t\n");
612. for(int i=1;i<=finished\_index;++i)
613. {
614. printf("|\t%c\t|\t",finished[i].process\_name);*//输出进程名称*
615. printf("%d\t|\t",finished[i].time\_end);*//输出进程完成时间*
616. printf("%d\t|\t",finished[i].time\_turnaround);*//周转时间*
617. printf("%.2lf\t|\n",finished[i].time\_turnaround\_rights);*//带权周转时间*
618. }
619. }
620. void display\_avergae()*//显示平均周转时间和平均带权周转时间*
621. {
622. double average\_time\_turnaround=0;
623. double average\_time\_turnaround\_rights=0;
624. for(int i=1;i<=finished\_index;++i)
625. {
626. average\_time\_turnaround+=finished[i].time\_turnaround;
627. average\_time\_turnaround\_rights+=finished[i].time\_turnaround\_rights;
628. }
629. average\_time\_turnaround/=finished\_index;
630. average\_time\_turnaround\_rights/=finished\_index;
631. printf("avergae turnaround time=%.2lf\n",average\_time\_turnaround);
632. printf("average weighted turnaround time=%.2lf",average\_time\_turnaround\_rights);
633. }
634. };
635. }
636. int main()
637. {
638. printf("\n\nThis is SJF:\n\n\n");*//使用新来进程优先的算法*
639. os::SJF \*sjf=new os::SJF();
640. sjf->conduct();
641. sjf->display();
642. sjf->display\_avergae();
643. delete(sjf);
644. printf("\n\n\n");
645. printf("\n\nThis is RR:\n\n\n");*//使用新来进程优先的算法*
646. os::RR \*rr=new os::RR();
647. rr->conduct();
648. rr->display();
649. rr->display\_avergae();
650. delete(rr);
651. printf("\n\n\n");
652. return 0;
653. }

# 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。

## 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

1. #include<stdio.h>
2. #include<stdlib.h>
3. #include<pthread.h>
4. #include<semaphore.h>
5. #include<unistd.h>
6. #define MAX\_BUFFER 5
7. int buffer[MAX\_BUFFER];*//缓冲*
8. int in=0,out=0;
9. sem\_t s1;*//控制缓冲区未满*
10. 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,item);
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

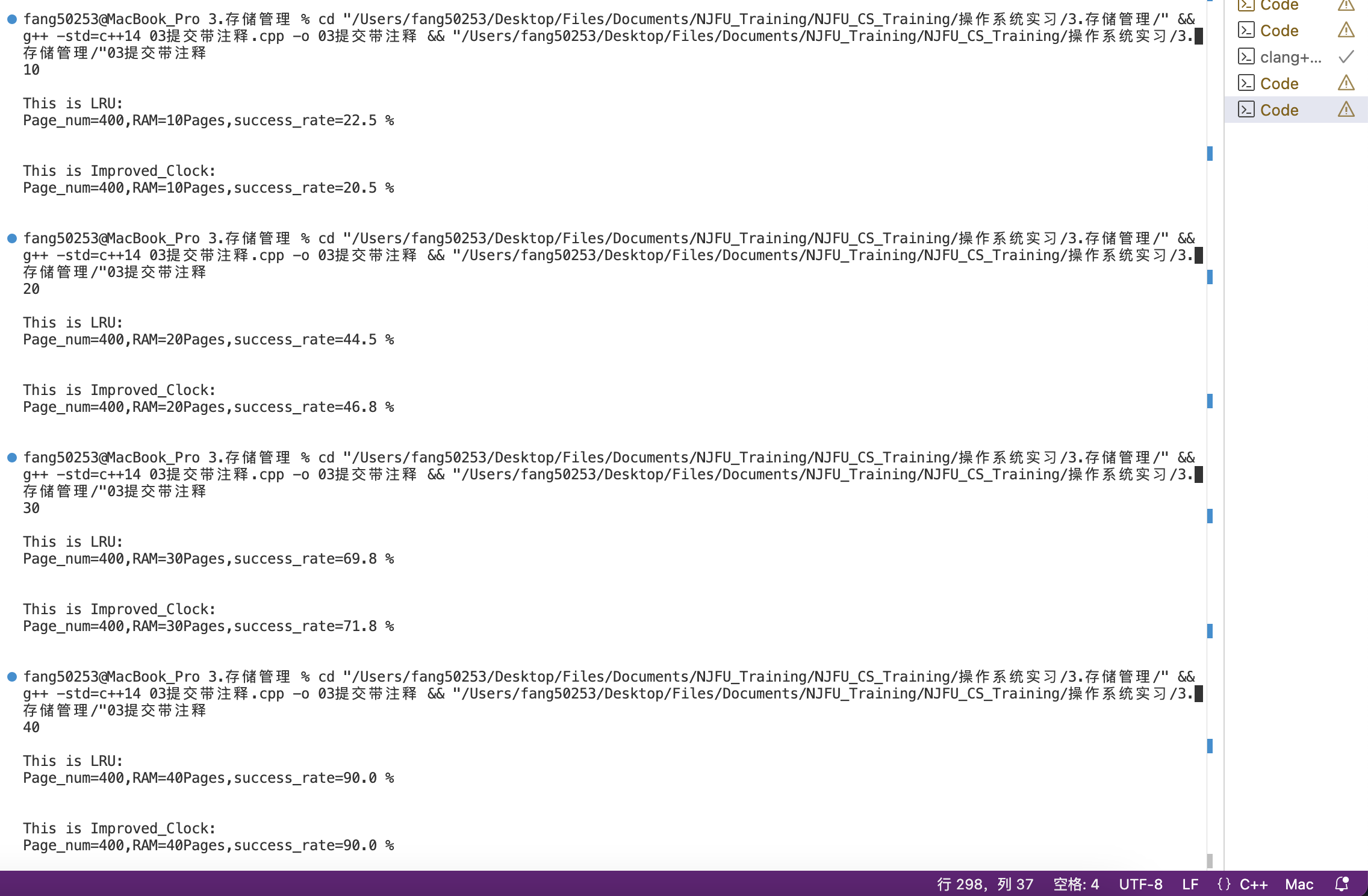
## 1.3 实验结果

* 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：



## 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。



* 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

1. #include<stdio.h>
2. #include<stdlib.h>
3. #include<math.h>
4. #include<time.h>
5. #include<iostream>
6. #define MAXSIZE 400
7. namespace fzy*//创建命名空间fzy，用于写一些数据结构*
8. {
9. template<typename T>
10. struct queuenode *//创建队列结点*
11. {
12. T v = T();*//提供队列结点的默认构造方法*
13. queuenode<T>\* next = nullptr;*//指向下一个位置，并提供默认构造方法*
14. };
15. template <typename T>*//创建队列模板类*
16. class queue
17. {
18. private:
19. queuenode<T>\* head;
20. queuenode<T>\* last;
21. int size\_;*//队列的大小*
22. public:
23. queue() *//队列模板的默认构造方法*
24. {
25. head = new queuenode<T>;
26. last = head;
27. size\_ = 0;
28. }
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\_revise = -1, int LRU\_time = -1)
123. : pagenum(pagenum), count(count), clock\_visit(clock\_visit), clock\_revise(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.push(current);
198. }
199. new\_queue.push(p\_str(page,1,-1,-1,0));
200. lru\_queue = new\_queue;
201. }
202. }
203. }
204. print("LRU",MAXSIZE,n,effective\_times);*//输出结果*
205. }
206. void Improved\_Clock()
207. {
208. init();
209. for (int i = 0; i < MAXSIZE; ++i)
210. {
211. int t = 0;
212. int page = rand() % 400 / 10;
213. int m = rand()>>1;
214. *// 检查 Clock 队列是否命中*
215. fzy::queue<p\_str>temp=clock;
216. while (!temp.empty())
217. {
218. if (temp.front().pagenum == page)
219. {
220. t = 1;
221. ++effective\_times;
222. p\_str cur = temp.front(); *// 获取队首元素*
223. cur.clock\_visit = 1;      *// 修改访问标记*
224. temp.pop();               *// 弹出旧的队首*
225. temp.push(cur);           *// 将修改后的节点重新加入队列*
226. break;
227. }
228. temp.pop();
229. }
230. *// 未命中则置换*
231. if(!t)
232. {
233. if ((int)clock.size() < n) clock.push(p\_str(page, 1, 1, m));*//队列没有满*
234. else *//队列满了，则只能置换*
235. {
236. *// 查找替换目标*
237. while (true)
238. {
239. p\_str cur = clock.front();
240. clock.pop();
241. if (cur.clock\_visit == 0)
242. {
243. clock.push(p\_str(page, 1, 1, m));
244. break;
245. }
246. else
247. {
248. cur.clock\_visit = 0;
249. clock.push(cur); *// 重置访问标记，放回队列末尾*
250. }
251. }
252. }
253. }
254. }
255. print("Improved\_Clock",MAXSIZE,n,effective\_times);*//输出*
256. }
257. };
258. int main()
259. {
260. int n;
261. srand((unsigned)time(NULL));
262. scanf("%d", &n);
263. printf("\n\n\n");
264. storage storage(n);
265. storage.LRU();*//最近最久未使用*
266. storage.Improved\_Clock();*//改进的时钟*
267. printf("\n\n\n");
268. return 0;
269. }