

**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

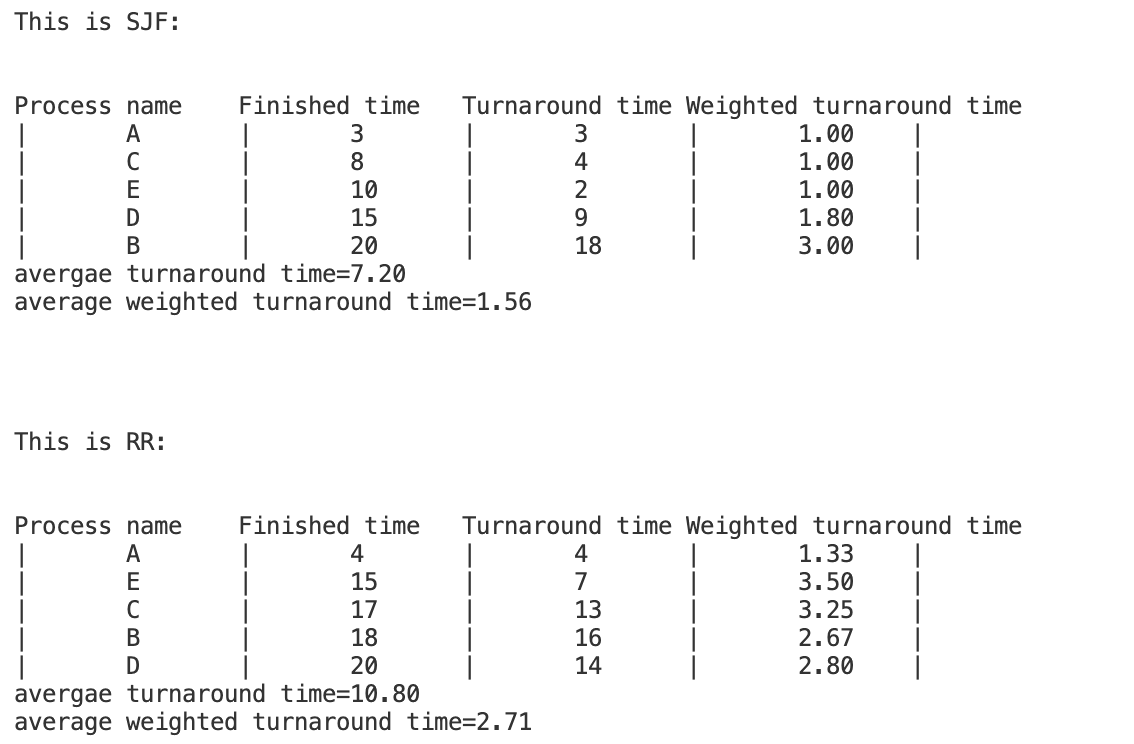
第18-19秒，执行程序D,剩余1/5

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. #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), compare(&\_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), compare(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), compare(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("queue 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. 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 time\_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\_name;
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.time\_arrive;
507. finished[finished\_index].time\_turnaround\_rights=1.0\*finished[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*
511. }
512. }
513. void display()*//显示函数*
514. {
515. printf("Process name\t");
516. printf("Finished time\t");
517. printf("Turnaround time\t");
518. printf("Weighted turnaround time\t\n");
519. for(int i=1;i<=finished\_index;++i)
520. {
521. printf("|\t%c\t|\t",finished[i].process\_name);*//输出进程名称*
522. printf("%d\t|\t",finished[i].time\_end);*//输出进程完成时间*
523. printf("%d\t|\t",finished[i].time\_turnaround);*//周转时间*
524. printf("%.2lf\t|\n",finished[i].time\_turnaround\_rights);*//带权周转时间*
525. }
526. }
527. void display\_avergae()
528. {
529. double average\_time\_turnaround=0;*//计算平均周转时间*
530. double average\_time\_turnaround\_rights=0;*//计算平均带权周转时间*
531. for(int i=1;i<=finished\_index;++i)
532. {
533. average\_time\_turnaround+=finished[i].time\_turnaround;
534. average\_time\_turnaround\_rights+=finished[i].time\_turnaround\_rights;
535. }
536. average\_time\_turnaround/=finished\_index;
537. average\_time\_turnaround\_rights/=finished\_index;
538. printf("avergae turnaround time=%.2lf\n",average\_time\_turnaround);*//输出平均周转时间*
539. printf("average weighted turnaround time=%.2lf",average\_time\_turnaround\_rights);*//输出平均带权周转时间*
540. }
541. };
542. class SJF*//抢占式短进程优先算法*
543. {
544. private:
545. void read(const char filename[])*//读取文件*
546. {
547. FILE \*fp=fopen(filename,"r+");
548. if(fp==NULL) ;*//throw std::runtime\_error("open file failed");*
549. fscanf(fp,"%d",&process\_cnt);
550. for(int i=1;i<=process\_cnt;++i)
551. {
552. char process\_name;
553. int time\_arrive,time\_serve;
554. fscanf(fp," %c%d%d",&process\_name,&time\_arrive,&time\_serve);
555. process.push(Process(process\_name,time\_arrive,time\_serve));*//创建临时类去压入优先队列中*
556. }
557. fclose(fp);*//关闭文件*
558. }
559. int process\_cnt;
560. fzy::priority\_queue<Process>process;
561. fzy::priority\_queue<Process\_remain>doing\_process;
562. public:
563. SJF()*//默认构造方法*
564. {
565. process\_cnt=0;
566. finished\_index=0;
567. char filename[]="filename.dat";
568. read(filename);*//读取文件*
569. memset(finished,0,sizeof finished);
570. finished\_index=0;
571. }
572. void conduct()*//抢占式短进程优先*
573. {
574. int time=0;
575. Process\_remain doing;
576. while(!(process.empty()&&doing\_process.empty()))
577. {
578. Process top\_process;
579. if(!process.empty())
580. {
581. top\_process=process.top();
582. while(top\_process.time\_arrive<=time)
583. {
584. process.pop();
585. doing\_process.push(Process\_remain(top\_process));
586. top\_process=process.top();
587. if(process.empty()) break;
588. }
589. }
591. doing=doing\_process.top();
592. doing\_process.pop();
593. --doing.remain;++time;
594. if(doing.remain==0)
595. {
596. ++finished\_index;
597. finished[finished\_index].process\_name=doing.process\_name;
598. finished[finished\_index].time\_arrive=doing.time\_arrive;
599. finished[finished\_index].time\_end=time;
600. finished[finished\_index].time\_serve=doing.time\_serve;
601. finished[finished\_index].time\_turnaround=time-doing.time\_arrive;
602. finished[finished\_index].time\_turnaround\_rights=1.0\*finished[finished\_index].time\_turnaround/doing.time\_serve;
603. }
604. else doing\_process.push(doing);
605. }
606. }
607. void display()*//显示*
608. {
609. printf("Process name\t");
610. printf("Finished time\t");
611. printf("Turnaround time\t");
612. printf("Weighted turnaround time\t\n");
613. for(int i=1;i<=finished\_index;++i)
614. {
615. printf("|\t%c\t|\t",finished[i].process\_name);*//输出进程名称*
616. printf("%d\t|\t",finished[i].time\_end);*//输出进程完成时间*
617. printf("%d\t|\t",finished[i].time\_turnaround);*//周转时间*
618. printf("%.2lf\t|\n",finished[i].time\_turnaround\_rights);*//带权周转时间*
619. }
620. }
621. void display\_avergae()*//显示平均周转时间和平均带权周转时间*
622. {
623. double average\_time\_turnaround=0;
624. double average\_time\_turnaround\_rights=0;
625. for(int i=1;i<=finished\_index;++i)
626. {
627. average\_time\_turnaround+=finished[i].time\_turnaround;
628. average\_time\_turnaround\_rights+=finished[i].time\_turnaround\_rights;
629. }
630. average\_time\_turnaround/=finished\_index;
631. average\_time\_turnaround\_rights/=finished\_index;
632. printf("avergae turnaround time=%.2lf\n",average\_time\_turnaround);
633. printf("average weighted turnaround time=%.2lf",average\_time\_turnaround\_rights);
634. }
635. };
636. }
637. int main()
638. {
639. printf("\n\nThis is SJF:\n\n\n");*//使用新来进程优先的算法*
640. os::SJF \*sjf=new os::SJF();
641. sjf->conduct();
642. sjf->display();
643. sjf->display\_avergae();
644. delete(sjf);
645. printf("\n\n\n");
646. printf("\n\nThis is RR:\n\n\n");*//使用新来进程优先的算法*
647. os::RR \*rr=new os::RR();
648. rr->conduct();
649. rr->display();
650. rr->display\_avergae();
651. delete(rr);
652. printf("\n\n\n");
653. return 0;
654. }

# 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

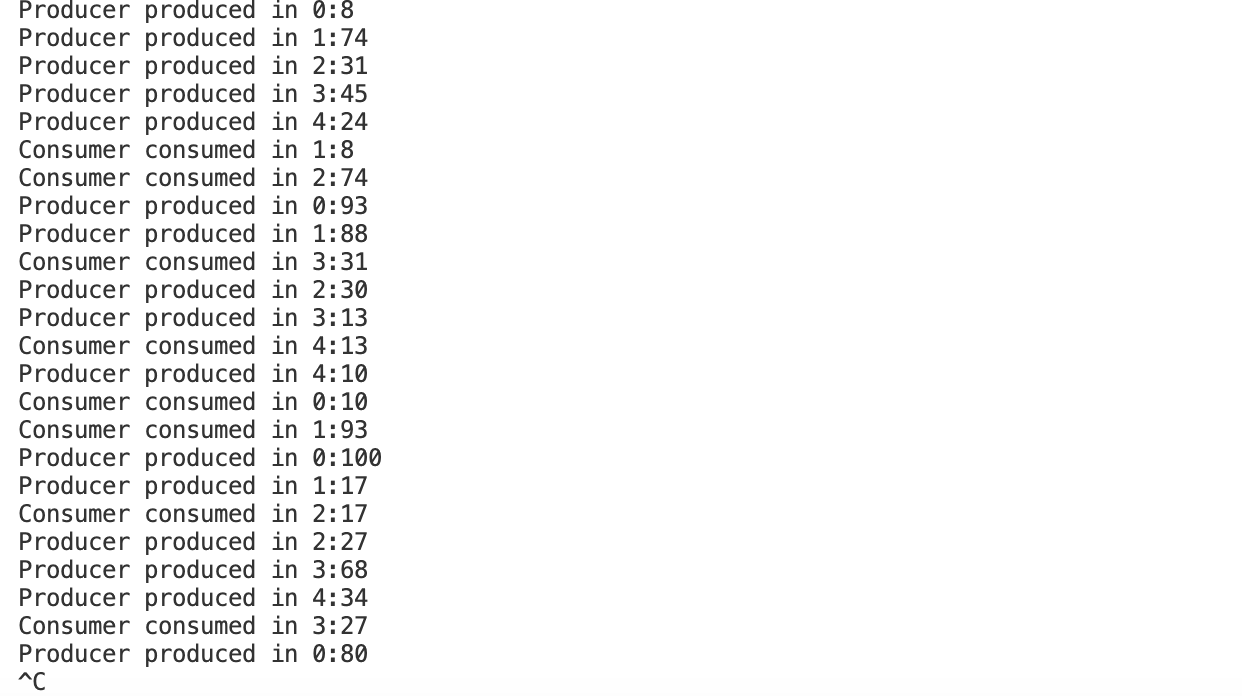
* 整型信号量：wait(S)和signal(S)是两个原子操作，因此，它们在执行时是不可中断的。亦即，当一个进程在修改某信号量时，没有其它进程可同时对该信号量进行修改。此外，在wait操作中，对S值的测试和做S=S-1操作时都不可中断。
* 记录型信号量：记录型信号量是一种不存在“忙等”现象的进程同步机制。除了需要一个用于代表资源数目的整型变量value外，再增加一个进程链表L，用于链接所有等待该资源的进程，记录型信号量得名于采用记录型的数据结构。
* 使用信号量实现线程同步：信号量机制能用于解决进程间的各种同步问题。设S为实现进程P1，P2同步的公共信号量，初始值为0。进程P2中的语句y要使用进程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：



• 第一次生产：

• Producer produced in 0:8，产品8被生产并放入位置0。

• Producer produced in 1:74，产品74被生产并放入位置1。

• Producer produced in 2:31，产品31被生产并放入位置2。

• Producer produced in 3:45，产品45被生产并放入位置3。

• Producer produced in 4:24，产品24被生产并放入位置4。

• 第一次消费：

• Consumer consumed in 1:8，消费者从位置1消费了产品8。

• Consumer consumed in 2:74，消费者从位置2消费了产品74。

• 接下来生产和消费交替进行：

• Producer produced in 0:93，生产者在线程0的位置放入93的产品。

• Producer produced in 1:88，生产者在线程1的位置放入88的产品。

• Consumer consumed in 3:31，消费者从位置3消费了31。

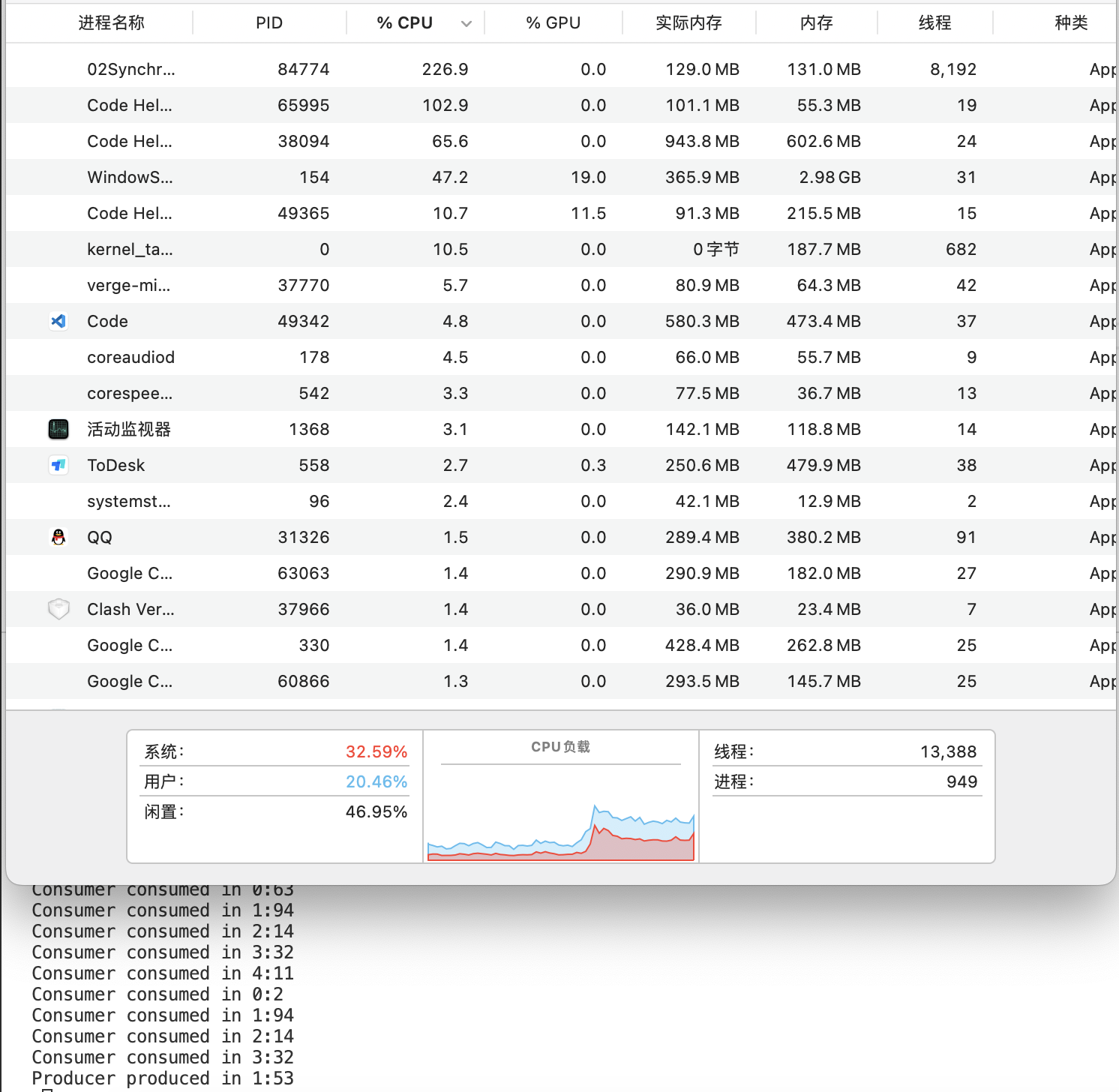
• Producer produced in 2:30，生产者在线程2的位置放入30的产品。

• Producer produced in 3:13，生产者在线程3的位置放入13的产品。

• 继续交替，直到缓冲区达到满状态或线程停止运行。

## 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

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. if(in==out) 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. printf("Consumer consumed in %d:%d\n",out,item);
39. sem\_post(&mutex);
40. if((in+1)%MAX\_BUFFER==out) 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. pthread\_create(&prod,NULL,producer,NULL);*//创建生产者进程*
53. sleep(2);
54. pthread\_create(&cons,NULL,consumer,NULL);*//创建消费者进程*
55. pthread\_join(prod, NULL);
56. pthread\_join(cons, NULL);
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

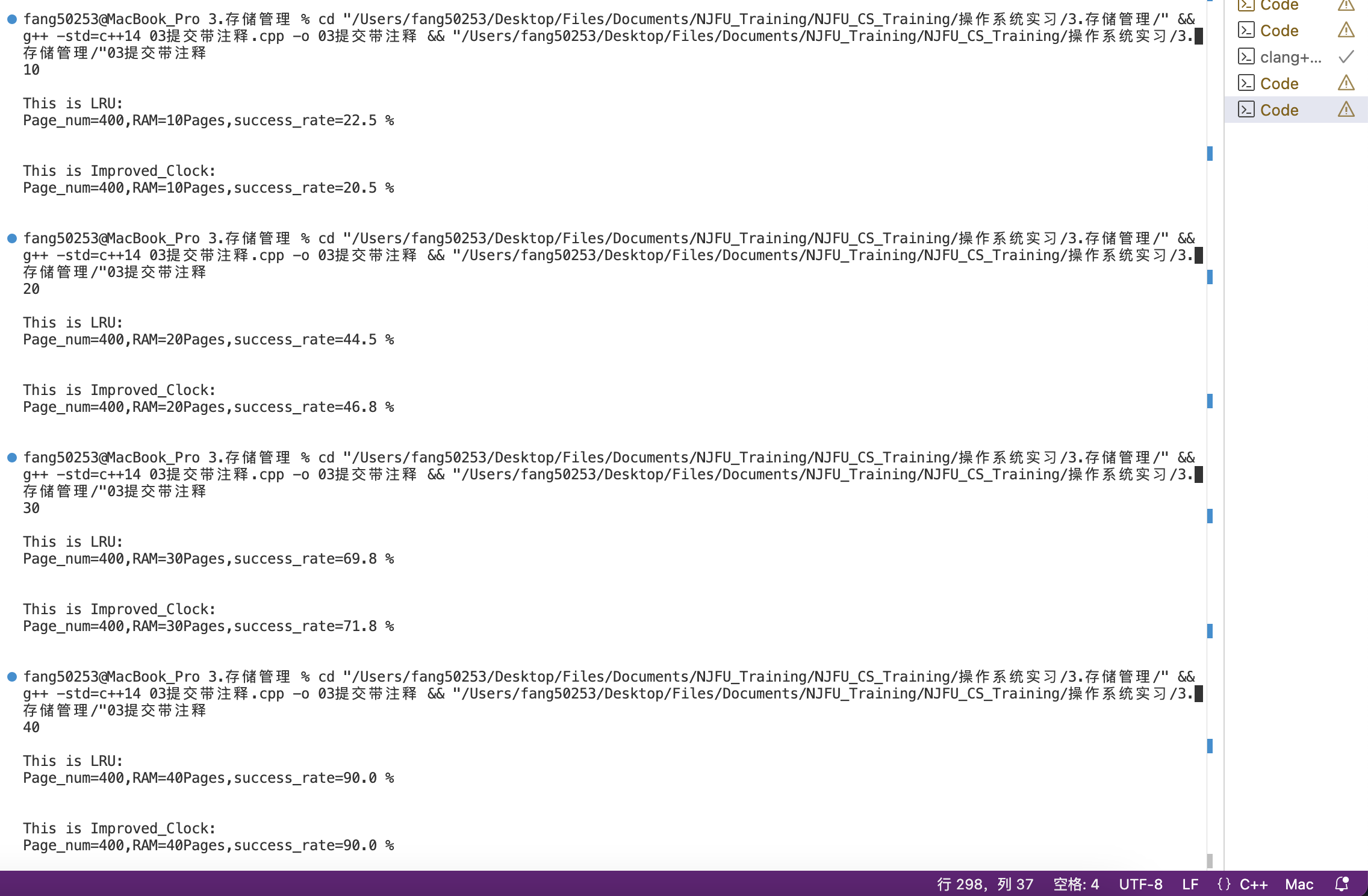
## 1.3 实验结果

* 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. #define MAXSIZE 400
6. namespace fzy*//创建命名空间fzy，用于写一些数据结构*
7. {
8. template<typename T>
9. struct queuenode *//创建队列结点*
10. {
11. T v = T();*//提供队列结点的默认构造方法*
12. queuenode<T>\* next = nullptr;*//指向下一个位置，并提供默认构造方法*
13. };
14. template <typename T>*//创建队列模板类*
15. class queue
16. {
17. private:
18. queuenode<T>\* head;
19. queuenode<T>\* last;
20. int size\_;*//队列的大小*
21. public:
22. 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. {
108. std::cout << current->v << " ";
109. current = current->next;
110. }
111. std::cout << std::endl;
112. }
113. \*/
114. };
115. }
116. struct p\_str
117. {
118. int pagenum;     *// 页号*
119. int count;       *// 页面访问次数*
120. int clock\_visit; *// 改进 Clock 的访问标记*
121. int clock\_revise;*// 改进 Clock 的修改标记*
122. int LRU\_time;    *// LRU 的未使用时间*
123. p\_str(int pagenum = -1, int count = -1, int clock\_visit = -1, int clock\_revise = -1, int LRU\_time = -1)
124. : pagenum(pagenum), count(count), clock\_visit(clock\_visit), clock\_revise(clock\_revise), LRU\_time(LRU\_time) {}*//提供构造方法*
125. };
126. class storage
127. {
128. private:
129. int n;                        *// 内存大小*
130. double effective\_times;       *// 命中次数*
131. fzy::queue<p\_str> fifo;       *// FIFO 队列*
132. fzy::queue<p\_str> clock;      *// 改进 Clock 队列*
133. fzy::queue<p\_str> lru\_queue;  *// LRU 模拟队列（用于访问）*
134. void init()
135. {
136. effective\_times = 0;
137. while (!fifo.empty()) fifo.pop();*//初始化每一个数据结构*
138. while (!clock.empty()) clock.pop();*//初始化每一个数据结构*
139. while (!lru\_queue.empty()) lru\_queue.pop();*//初始化每一个数据结构*
140. }
141. void print(const char way[],int Page\_num,int RAM,double success)
142. {
143. printf("This is %s:\nPage\_num=%d,RAM=%dPages,success\_rate=%.1lf %%\n\n\n",way,Page\_num,RAM,success/4);
144. *//输出结果*
145. }
146. 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. {
190. p\_str current = temp.front();
191. temp.pop();
192. if (current.pagenum == page)
193. {
194. ++t;
195. ++effective\_times;
196. current.LRU\_time = 0; *// 重置未访问时间*
197. }
198. else ++current.LRU\_time; *// 更新未访问时间*
199. new\_queue.push(current);
200. }
201. lru\_queue = new\_queue;
202. *// 未命中则置换*
203. if (!t)
204. {
205. if ((int)lru\_queue.size() < n) *//队列没有满*
206. lru\_queue.push(p\_str(page, 1, -1, -1, 0));
207. else *//队列满了，找到最久没有使用的页面*
208. {
209. *// 找到最久未使用页面*
210. fzy::queue<p\_str> temp = lru\_queue;
211. p\_str max\_page;
212. int max\_time = -1;
213. while (!temp.empty())
214. {
215. p\_str current = temp.front();
216. temp.pop();
217. if (current.LRU\_time > max\_time)
218. {
219. max\_time = current.LRU\_time;
220. max\_page = current;
221. }
222. }
223. fzy::queue<p\_str> new\_queue;
224. temp = lru\_queue;
225. while (!temp.empty())
226. {
227. p\_str current = temp.front();
228. temp.pop();
229. if (current.pagenum != max\_page.pagenum) new\_queue.push(current);
230. }
231. new\_queue.push(p\_str(page,1,-1,-1,0));
232. 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. {
275. clock.push(p\_str(page, 1, 1, m));
276. break;
277. }
278. else
279. {
280. cur.clock\_visit = 0;
281. clock.push(cur); *// 重置访问标记，放回队列末尾*
282. }
283. }
284. }
285. }
286. }
287. print("Improved\_Clock",MAXSIZE,n,effective\_times);*//输出*
288. }
289. };
290. int main()
291. {
292. int n;
293. srand((unsigned)time(NULL));
294. scanf("%d", &n);
295. printf("\n");
296. storage storage(n);
297. storage.FIFO();*//先进先出*
298. storage.LRU();*//最近最久未使用*
299. storage.Improved\_Clock();*//改进的时钟*
300. return 0;
301. }