

## 计算机科学与工程学院 实验报告

实验课程名称	操作系统实验		
专业	XXXX	班级	XXXX
学号	XXXX	姓名	XXXX

### 实验项目目录

1. 实验一 进程状态转换实验
2. 实验二 生产者消费者问题模拟
3. 实验三 进程间的管道通信
4. 实验四 内存置换算法

### 实验报告正文

#### 实验题目 一、进程的状态转换及 PCB 的变化模拟实验

##### 实验简述

这是一个设计型实验。要求自行设计、编制模拟程序，通过形象化的状态显示，加深理解进程的概念、进程之间的状态转换及其所带来的 PCB 组织的变化，理解进程与其 PCB 间的一一对应关系。

设计并实现一个模拟进程状态转换及其相应 PCB 组织结构变化的程序

独立设计、编写、调试程序

程序界面应能反映出在模拟条件下，进程之间状态转换及其对应的 PCB 组织的变化。

##### 实验内容

使用了 C++ 进行开发，实现了一个五状态的进程间状态转移的模型。同时设定了内存的机制，可以设置不同进程的内存和总内存的大小。完成了界面功能，可以显示实时的不同队列中的进程的 pid、内存、优先级信息。可以实现初始化创建队列、就绪队列、运行队列、阻塞队列，Dispatch、Timeout、Event Wait、Event Occur、Create、Release、Admit 的操作。同时设定了不同操作之间的在不同情况下的连带关系。同时考虑到了非法的输入，对于不同的非法输入都有处理：比如创建两个 pid 相同的进程、创建内存大于总内存的进程等等操作。

自定义了结构体 pcb，用来存储一个进程的 pid、内存大小、priority 优先级。定义了 4 个优先队列，分别用来储存 Ready, Running, Block, New 队列。队列内部使用进程 pcb 的优先级来排序，优先级高的排在队头先出队。同时定义一个全局变量 mem 用来存储当前所有内存中的进程的内存大小。避免新建的进程内存大于总的内存，同时当内存不够时，新的 NEW 进程就会在 New 队列中，一旦内存满足 New 队列中的优先级最高的进程，该进程就会自动被调入 Ready 队列中。此外还定义了一个哈希表，用来储存已经存在的 PID 的值，如果新建的进程的 pid 已经存在，就会给

出错误提示。

主要函数的解释：

**void Dispatch\_Timeout();**

时间片到/调度，当 Ready 不为空时，会从 Ready 优先队列中取得队头的优先级最高的进程转入 Running，同时将原来 Running 中的进程放回 Ready。

若 Ready 为空，则不发生变化。

**void EventWait();**

将运行中的进程转入 Block 阻塞队列，同时若 Ready 不为空，则自动调度一个进程进入 Running 中。

**void EventOccur();**

唤醒进程，输入想要唤醒的进程的 pid，完成对该进程的唤醒，由 Block 转入 Ready 队列。同时若 Running 为空，则该进程会自动再次调度进入 Running 队列。

**void Create();**

创建一个新的进程。输入新建进程的 pid、内存大小、priority 优先级别。若 pid 已经存在或内存大小大于系统的总内存，则创建失败。否则创建成功。创建后在 New 队列中。若此时剩余内存足够，则该进程会自动从 New 队列转入 Ready 队列。

**void Release();**

释放进程，将正在 Running 的进程释放掉，并清除掉该 pid 和该进程所占用的内存。同时，自动进行调度，将 Ready 中一个进程转入 Running，同时检查 New 队列中是否有内存足够的进程使其转入 Ready 队列。

**void Init();**

初始化，可以指定内存大小、当前的各个队列中的进程的 pid、内存、priority。

**void Show();**

显示当前的所有队列中的进程信息。

包括了所有的五状态模型的状态转换的连带关系，并考虑到了各种的非法输入的处理。

实验结果（截图）

```
szw@ubuntu: ~/文档/os-experiment/experiment1/build
szw@ubuntu:~/文档/os-experiment/experiment1$ cd build/
szw@ubuntu:~/文档/os-experiment/experiment1/build$ ./experiment1
初始化:
请输入准备队列中的进程数:
6
请依次输入进程的pid、优先级、内存大小:
0 3 10
1 4 30
2 1 5
3 2 20
4 3 20
5 9 100
请输入就绪队列中的进程数:
3
请依次输入进程的pid、优先级、内存大小:
6 4 20
7 6 30
8 4 30
运行队列中是否已经存在进程: (1 | 0)
1
请输入该运行进程的pid、优先级、内存大小:
10 10 200
输入堵塞队列中的进程数:
1
请依次输入进程的pid、优先级、内存大小:
9 5 50
```

```
szw@ubuntu: ~/文档/os-experiment/experiment1/build
输入堵塞队列中的进程数:
1
请依次输入进程的pid、优先级、内存大小:
9 5 50
内存占用: 490MB / 500MB
准备队列:
pid:3  priority:2  size of memery:20MB
pid:2  priority:1  size of memery:5MB
就绪队列:
pid:5  priority:9  size of memery:100MB
pid:7  priority:6  size of memery:30MB
pid:1  priority:4  size of memery:30MB
pid:6  priority:4  size of memery:20MB
pid:8  priority:4  size of memery:30MB
pid:0  priority:3  size of memery:10MB
pid:4  priority:3  size of memery:20MB
运行队列:
pid:10 priority:10  size of memery:200MB
阻塞队列:
pid:9  priority:5  size of memery:50MB

请选择想要进行的操作:
1.Dispatch&&Timeout
2.Event Wait
3.Event Occur
4.Create
5.Release
6.结束程序
2
```

szw@ubuntu: ~/文档/os-experiment/experiment1/build

内存占用: 490MB / 500MB

准备队列:

pid:3 priority:2 size of memery:20MB  
pid:2 priority:1 size of memery:5MB

就绪队列:

pid:7 priority:6 size of memery:30MB  
pid:1 priority:4 size of memery:30MB  
pid:6 priority:4 size of memery:20MB  
pid:8 priority:4 size of memery:30MB  
pid:0 priority:3 size of memery:10MB  
pid:4 priority:3 size of memery:20MB

运行队列:

pid:5 priority:9 size of memery:100MB

阻塞队列:

pid:10 priority:10 size of memery:200MB  
pid:9 priority:5 size of memery:50MB

请选择想要进行的操作:

- 1.Dispatch&&Timeout
  - 2.Event Wait
  - 3.Event Occur
  - 4.Create
  - 5.Release
  - 6.结束程序
- 5

szw@ubuntu: ~/文档/os-experiment/experiment1/build

内存占用: 415MB / 500MB

准备队列:

就绪队列:

pid:1 priority:4 size of memery:30MB  
pid:6 priority:4 size of memery:20MB  
pid:8 priority:4 size of memery:30MB  
pid:0 priority:3 size of memery:10MB  
pid:4 priority:3 size of memery:20MB  
pid:3 priority:2 size of memery:20MB  
pid:2 priority:1 size of memery:5MB

运行队列:

pid:7 priority:6 size of memery:30MB

阻塞队列:

pid:10 priority:10 size of memery:200MB  
pid:9 priority:5 size of memery:50MB

请选择想要进行的操作:

- 1.Dispatch&&Timeout
- 2.Event Wait
- 3.Event Occur
- 4.Create
- 5.Release
- 6.结束程序

szw@ubuntu: ~/文档/os-experiment/experiment1/build

内存占用: 415MB / 500MB

准备队列:

就绪队列:

pid:1	priority:4	size of memory:30MB
pid:6	priority:4	size of memory:20MB
pid:8	priority:4	size of memory:30MB
pid:0	priority:3	size of memory:10MB
pid:4	priority:3	size of memory:20MB
pid:3	priority:2	size of memory:20MB
pid:2	priority:1	size of memory:5MB

运行队列:

pid:7	priority:6	size of memory:30MB
-------	------------	---------------------

阻塞队列:

pid:10	priority:10	size of memory:200MB
pid:9	priority:5	size of memory:50MB

请选择想要进行的操作:

- 1.Dispatch&&Timeout
  - 2.Event Wait
  - 3.Event Occur
  - 4.Create
  - 5.Release
  - 6.结束程序
- 3

szw@ubuntu: ~/文档/os-experiment/experiment1/build

请输入进程的pid、优先级、内存

6 10 500

该pid已存在!

内存占用: 415MB / 500MB

准备队列:

就绪队列:

pid:10	priority:10	size of memory:200MB
pid:1	priority:4	size of memory:30MB
pid:6	priority:4	size of memory:20MB
pid:8	priority:4	size of memory:30MB
pid:0	priority:3	size of memory:10MB
pid:4	priority:3	size of memory:20MB
pid:3	priority:2	size of memory:20MB
pid:2	priority:1	size of memory:5MB

运行队列:

pid:7	priority:6	size of memory:30MB
-------	------------	---------------------

阻塞队列:

pid:9	priority:5	size of memory:50MB
-------	------------	---------------------

请选择想要进行的操作:

- 1.Dispatch&&Timeout
- 2.Event Wait
- 3.Event Occur
- 4.Create
- 5.Release
- 6.结束程序

```
szw@ubuntu: ~/文档/os-experiment/experiment1/build
请输入进程的pid、优先级、内存
12 2 510
该进程内存超出最大内存!!!
内存占用: 415MB / 500MB
准备队列:
pid:11 priority:20 size of memery:500MB
就绪队列:
pid:10 priority:10 size of memery:200MB
pid:1 priority:4 size of memery:30MB
pid:6 priority:4 size of memery:20MB
pid:8 priority:4 size of memery:30MB
pid:0 priority:3 size of memery:10MB
pid:4 priority:3 size of memery:20MB
pid:3 priority:2 size of memery:20MB
pid:2 priority:1 size of memery:5MB
运行队列:
pid:7 priority:6 size of memery:30MB
阻塞队列:
pid:9 priority:5 size of memery:50MB

请选择想要进行的操作:
1.Dispatch&&Timeout
2.Event Wait
3.Event Occur
4.Create
5.Release
6.结束程序
```

源码:

Pcb.cpp:

```
1. #include "pcb.h"
2. using namespace std;
3.
4. //定义内存为 500MB
5. #define MEM 500
6. int now_mem=0;
7. set<pcb> Ready,Running,Block,New;
8. set<int> PID;
9.
10.
11. int main(int argc,char* argv[]){
12.     Init();
13.
14.
15.     int p;
16.     while(1) {
17.         printf("\n 请选择想要进行的操
作:\n1.Dispatch&&Timeout\n2.Event Wait\n3.Event Occur\n4.Create\n5.Release\n6.结束程序
\n");
18.         cin>>p;
```

```

19.     switch(p){
20.     case 1:system("clear");Dispatch_Timeout();break;
21.     case 2:system("clear");EventWait();break;
22.     case 3:system("clear");EventOccur();break;
23.     case 4:system("clear");Create();break;
24.     case 5:system("clear");Release();break;
25.     case 6:return 0;
26.     default:cout<<"输入错误! \n";
27.     }
28.
29. }
30. return 0;
31. }
32.
33.
34. void Init(){
35.
36.     int nums;
37.     int pid,memory,priority;
38.
39.     printf("初始化: \n 请输入准备队列中的进程数:\n");
40.     scanf("%d",&nums);
41.     printf("请依次输入进程的 pid、优先级、内存大小:\n");
42.     while(nums--){
43.         cin>>pid>>priority>>memory;
44.         if(PID.count(pid)==1) {cout<<"该 pid 已存在! "<<endl;continue;}
45.         else if(memory>MEM) {cout<<"该进程内存超出最大内存!!! "<<endl;continue;}
46.         else {PID.insert(pid);}
47.         New.insert(pcb{pid,priority,memory});
48.     }
49.     printf("请输入就绪队列中的进程数:\n");
50.     scanf("%d",&nums);
51.     printf("请依次输入进程的 pid、优先级、内存大小:\n");
52.     while(nums--){
53.         cin>>pid>>priority>>memory;
54.         if(PID.count(pid)==1) {cout<<"该 pid 已存在! "<<endl;continue;}
55.         else if(memory>MEM) {cout<<"该进程内存超出最大内存!!! "<<endl;continue;}
56.         else {PID.insert(pid);}
57.         now_mem+=memory;
58.         Ready.insert(pcb{pid,priority,memory});
59.     }
60.     printf("运行队列中是否已经存在进程: (1 | 0)\n");
61.     scanf("%d",&nums);
62.     if(nums==1){

```

```

63.     printf("请输入该运行进程的 pid、优先级、内存大小:\n");
64.     cin>>pid>>priority>>memory;
65.     if(PID.count(pid)==1) {cout<<"该 pid 已存在! "<<endl;}
66.     else if(memory>MEM) {cout<<"该进程内存超出最大内存!!! "<<endl;}
67.     else {
68.         PID.insert(pid);
69.         now_mem+=memory;
70.         Running.insert(pcb{pid,priority,memory});
71.     }
72. }
73. printf("输入堵塞队列中的进程数:\n");
74. scanf("%d",&nums);
75. printf("请依次输入进程的 pid、优先级、内存大小:\n");
76. while(nums--){
77.     cin>>pid>>priority>>memory;
78.     if(PID.count(pid)==1) {cout<<"该 pid 已存在! "<<endl;continue;}
79.     else if(memory>MEM) {cout<<"该进程内存超出最大内存!!! "<<endl;continue;}
80.     else {PID.insert(pid);}
81.     now_mem+=memory;
82.     Block.insert(pcb{pid,priority,memory});
83. }
84.
85. admit();
86. Show();
87. }
88.
89. void Show(){
90.     cout<<"内存占用: "<<now_mem<<"MB / "<<MEM<<"MB"<<endl;
91.     cout<<"准备队列: \n";
92.     for(const auto&i:New){
93.         cout<<i;
94.     }
95.
96.     cout<<"就绪队列: \n";
97.     for(const auto&i:Ready){
98.         cout<<i;
99.     }
100.
101.     cout<<"运行队列: \n";
102.     for(const auto&i:Running){
103.         cout<<i;
104.     }
105.
106.     cout<<"阻塞队列: \n";

```



```

107.     for(const auto&i:Block){
108.         cout<<i;
109.     }
110. }
111.
112.
113. void admit(){
114.     set<pcb>::iterator it=New.begin();
115.     while(!New.empty()&&(it->memory+now_mem)<=MEM){
116.         now_mem+=(it->memory);
117.         Ready.insert(*it);
118.         New.erase(it++);
119.     }
120.     if(Running.empty()&&!Ready.empty()){
121.         Running.insert(*Ready.begin());
122.         Ready.erase(Ready.begin());
123.     }
124. }
125.
126. void Dispatch_Timeout(){
127.     if(Ready.empty()){
128.         cout<<"就绪队列为空！"<<endl;
129.         Show();
130.         return;
131.     }
132.     set<pcb>::iterator it=Running.begin();
133.     Running.insert(*Ready.begin());
134.     Ready.erase(Ready.begin());
135.     Ready.insert(*it);
136.     Running.erase(it);
137.     Show();
138. }
139.
140. void EventWait(){
141.     if(Running.empty()) {
142.         cout<<"当前运行队列为空！！"<<endl;
143.         Show();
144.         return;
145.     }
146.     set<pcb>::iterator it=Running.begin();
147.     if(!Ready.empty()){
148.         Running.insert(*Ready.begin());
149.         Ready.erase(Ready.begin());
150.     }

```

```

151.     Block.insert(*it);
152.     Running.erase(it);
153.     Show();
154. }
155.
156. void EventOccur(){
157.     if(Block.empty()){
158.         cout<<"阻塞队列为空!!! "<<endl;
159.         Show();
160.         return;
161.     }
162.     int pid;
163.     Show();
164.     cout<<"请输入要唤醒的进程的 pid: "<<endl;
165.     scanf("%d",&pid);
166.     set<pcb>::iterator it=find_if(Block.begin(),Block.end(),[pid](const pcb& p)
167. {return p.pid==pid;});
168.     if(it==Block.end()) {
169.         cout<<"阻塞队列中没有该 pid 的进程! "<<endl;
170.         system("clear");
171.         Show();
172.         return;
173.     }
174.     if(Running.empty()&&Ready.empty()){
175.         Running.insert(*it);
176.         Block.erase(it);
177.     }
178.     else {
179.         Ready.insert(*it);
180.         Block.erase(it);
181.     }
182.     system("clear");
183.     Show();
184. }
185.
186. void Create(){
187.     cout<<"请输入进程的 pid、优先级、内存"<<endl;
188.     int pid,priority,memory;
189.     cin>>pid>>priority>>memory;
190.     if(PID.count(pid)==1) {cout<<"该 pid 已存在! "<<endl;Show();return;}
191.     else if(memory>MEM) {cout<<"该进程内存超出最大内存!!! "<<endl;Show();return;}
192.     PID.insert(pid);
193.     New.insert(pcb{pid,priority,memory});
194.     admit();

```

```

195.     cout<<"创建完毕！\n";
196.     Show();
197. }
198.
199. void Release(){
200.     if(Running.empty()) {
201.         cout<<"当前运行队列为空！！"<<endl;
202.         Show();
203.         return;
204.     }
205.     set<pcb>::iterator it=Running.begin();
206.     if(!Ready.empty()){
207.         Running.insert(*Ready.begin());
208.         Ready.erase(Ready.begin());
209.     }
210.     now_mem-=it->memory;
211.     PID.erase(it->pid);
212.     Running.erase(it);
213.     admit();
214.     Show();
215. }

```

#### Pcb.h:

```

1. #include<bits/stdc++.h>
2. #include<stdio.h>
3. #include<sys/types.h>
4. #include<stdlib.h>
5. #include<sys/stat.h>
6. #include<fcntl.h>
7. #include<error.h>
8. #include<wait.h>
9. #include<unistd.h>
10. using namespace std;
11.
12. typedef struct pcb{
13.     int pid=-1;
14.     int priority=-1;
15.     int memory=-1;
16.
17.     bool operator<(const pcb a)const{
18.         return (this->priority!=a.priority) ? this->priority > a.priority:
19. this->pid<a.pid;
20.     }
21.

```

```
22.     friend ostream& operator<<(ostream &out, const pcb &a){
23.         out<<"pid:"<<a.pid<<"\tpriority:"<<a.priority<<"\tsize of memory:"<<a.memory
24. <<"MB"<<endl;
25.         return out;
26.     }
27. }pcb;
28.
29.
30.
31.
32. void Init();
33. void Show();
34. /**
35.  * @brief: 将 running 中的进程放回 Ready, 并把 ready 中的优先级最高的进程放入 running
36.  */
37. void Dispatch_Timeout();
38. void EventWait();
39. void EventOccur();
40. void Create();
41. void Release();
42.
43. /**
44.  * @brief: 将 New 中转为 Ready
45.  */
46. void admit();
```

## 实验题目 二、进程同步和通信——生产者和消费者问题模拟

### 实验简述

这是一个验证型实验。通过对给出的程序进行验证、修改，进一步加深理解进程的概念，了解同步和通信的过程，掌握进程通信和同步的机制，特别是利用缓冲区进行同步和通信的过程。通过补充新功能，加强对知识的灵活运用，培养创新能力。

所给程序模拟两个进程，即生产者（producer）进程和消费者(Consumer)进程工作。

生产者每次产生一个数据，送入缓冲区中。

消费者每次从缓冲区中取走一个数据

### 实验内容

完成了 创建生产者、创建消费者、执行(ExeEnd)三个功能。

设定了缓冲区 BUFFER，设定了缓冲区的大小。

信号量有 BUFFER\_Empty, BUFFER\_Full, BUFFER\_Mutex;

同时设定了 Current\_process，创建的生产者或消费者并不会立刻完成输入或读取，而是在 Current\_process 中，直到输入 执行(ExeEnd)才会完成。

存在 Full\_queue, Empty\_queue, Mutex\_queue 三个阻塞队列。

完成了生产者消费者的同步互斥。

即完成了视频中的扩展点。

### 主要函数：

**bool P(string a);**

对信号量 a 进行 P 原语操作，a 的取值只有三种："Empty" "Full" "Mutex"。

**void V(string a);**

对信号量 a 进行 V 原语操作，a 的取值只有三种："Empty" "Full" "Mutex"。

**void consum(int n);**

创建第 n 个消费者，会调用 P(Full)和 P(Mutex),全部 P 成功后就会进入 current\_process，否则进入对应的阻塞队列。

**void product(int n);**

创建第 n 个生产者，会调用 P(Empty)和 P(Mutex),全部 P 成功后就会进入 current\_process，否则进入对应的阻塞队列。

**void exeend();**

执行当前的 current\_process，使其真正的完成写入/读取。

同时该操作会调用 V(Mutex)和 V(Empty)/V(Full),并引起一系列后续操作，通过调用后面的三个函数实现。

在 V(Mutex) 后会调用对应的 mutex\_proc()，在 V(Empty)/V(Full) 后会调用对应的 produce\_proc()/consum\_proc(),以此来完成后续的队列中的变化。

**bool mutex\_proc();**

若 Mutex\_queue 不为空，则唤醒一个 Mutex\_queue 中的进程，并把该进程置为 current\_process;

**bool produce\_proc();**

若 Empty\_queue 不为空则唤醒一个 Empty\_queue 中的进程。

然后完成该进程的 P(Mutex)操作，若 P(Mutex)成功，则放入 current\_process，否则放入 Mutex\_queue 阻塞队列中去。

**bool consum\_proc();**

若 Full\_queue 不为空则唤醒一个 Full\_queue 中的进程。

然后完成该进程的 P(Mutex)操作，若 P(Mutex)成功，则放入 current\_process，否则放入 Mutex\_queue 阻塞队列中去。

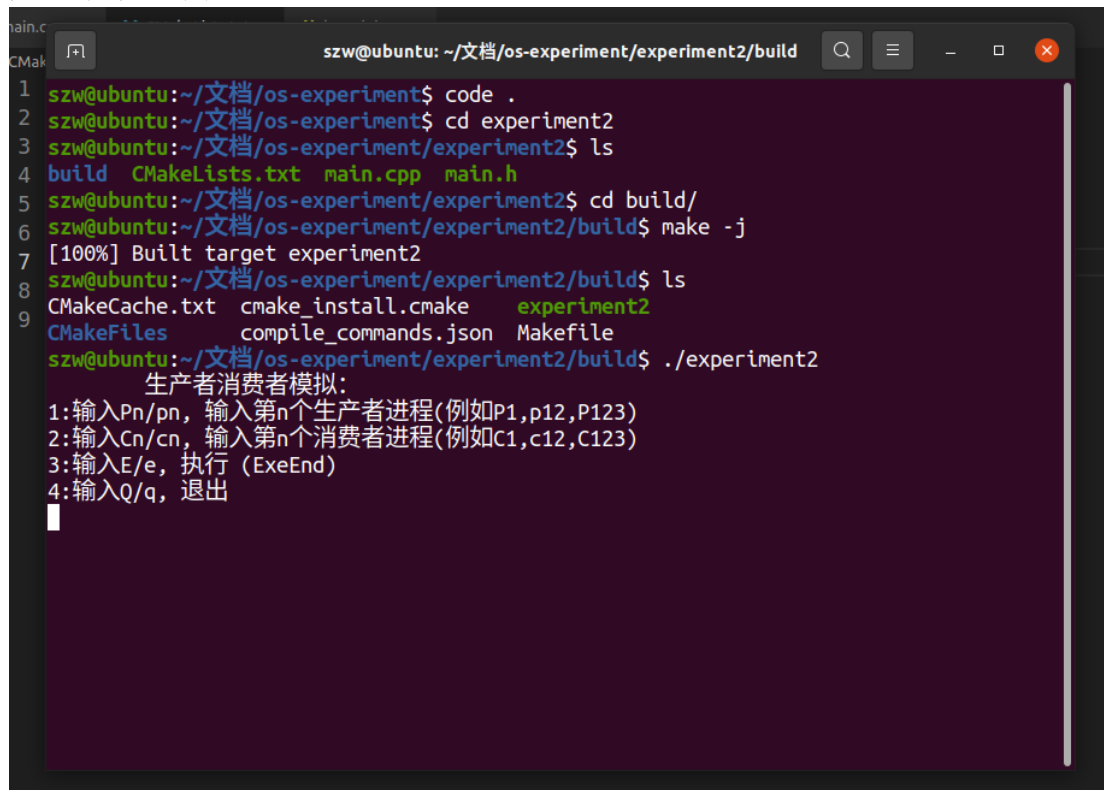
**void Show();**

**void show\_queue(queue<int> temp);**

**void show\_queue(queue<string> temp);**

显示当前缓冲区中的数据，显示当前状态的信号量 BUFFER\_Empty, BUFFER\_Full, BUFFER\_Mutex 的值，Full\_queue, Empty\_queue, Mutex\_queue 三个阻塞队列中的进程，显示当前的 current\_process。

### 实验结果（截图）



```
szw@ubuntu: ~/文档/os-experiment/experiment2/build
1 szw@ubuntu:~/文档/os-experiment$ code .
2 szw@ubuntu:~/文档/os-experiment$ cd experiment2
3 szw@ubuntu:~/文档/os-experiment/experiment2$ ls
4 build CMakeLists.txt main.cpp main.h
5 szw@ubuntu:~/文档/os-experiment/experiment2$ cd build/
6 szw@ubuntu:~/文档/os-experiment/experiment2/build$ make -j
7 [100%] Built target experiment2
8 szw@ubuntu:~/文档/os-experiment/experiment2/build$ ls
9 CMakeCache.txt cmake_install.cmake experiment2
CMakeFiles compile_commands.json Makefile
szw@ubuntu:~/文档/os-experiment/experiment2/build$ ./experiment2
生产者消费者模拟:
1:输入Pn/pn, 输入第n个生产者进程(例如p1,p12,P123)
2:输入Cn/cn, 输入第n个消费者进程(例如c1,c12,C123)
3:输入E/e, 执行 (ExeEnd)
4:输入Q/q, 退出
```

```
szw@ubuntu: ~/文档/os-experiment/experiment2/build
BUFFER:
Empty_queue:
Full_queue:
Mutex_queue: P2 P3 P4 P5 P6
Mutex: -5
Full: 0
Empty: 2
cur_process: P1
cur_data: 1 (正在读取或写入的数据)
生产者消费者模拟:
1:输入Pn/pn, 输入第n个生产者进程(例如P1,p12,P123)
2:输入Cn/cn, 输入第n个消费者进程(例如C1,c12,C123)
3:输入E/e, 执行 (ExeEnd)
4:输入Q/q, 退出
p7
```

```
szw@ubuntu: ~/文档/os-experiment/experiment2/build
BUFFER:
Empty_queue:
Full_queue: 1 2 3 4 5 6 12
Mutex_queue: P2 P3 P4 P5 P6 P7
Mutex: -6
Full: -7
Empty: 1
cur_process: P1
cur_data: 1 (正在读取或写入的数据)
生产者消费者模拟:
1:输入Pn/pn, 输入第n个生产者进程(例如P1,p12,P123)
2:输入Cn/cn, 输入第n个消费者进程(例如C1,c12,C123)
3:输入E/e, 执行 (ExeEnd)
4:输入Q/q, 退出
c7
```

```
szw@ubuntu: ~/文档/os-experiment/experiment2/build
BUFFER:
Empty_queue: 12 13 14
Full_queue: 1 2 3 4 5 6 12 7
Mutex_queue: P2 P3 P4 P5 P6 P7 P11
Mutex: -7
Full: -8
Empty: -3
cur_process: P1
cur_data: 1 (正在读取或写入的数据)
生产者消费者模拟:
1:输入Pn/pn, 输入第n个生产者进程(例如P1,p12,P123)
2:输入Cn/cn, 输入第n个消费者进程(例如C1,c12,C123)
3:输入E/e, 执行 (ExeEnd)
4:输入Q/q, 退出
e
```

```
szw@ubuntu: ~/文档/os-experiment/experiment2/build
BUFFER: 1
Empty_queue: 12 13 14
Full_queue: 2 3 4 5 6 12 7
Mutex_queue: P3 P4 P5 P6 P7 P11 C1
Mutex: -7
Full: -7
Empty: -3
cur_process: P2
cur_data: 2 (正在读取或写入的数据)
生产者消费者模拟:
1:输入Pn/pn, 输入第n个生产者进程(例如P1,p12,P123)
2:输入Cn/cn, 输入第n个消费者进程(例如C1,c12,C123)
3:输入E/e, 执行 (ExeEnd)
4:输入Q/q, 退出
```



```
szw@ubuntu: ~/文档/os-experiment/experiment2/build
1  BUFFER:      12
2  Empty_queue:
3  Full_queue:
4  Mutex_queue:
5  Mutex:       0
6  Full:        0
7  Empty:       7
8  cur_process: C999
9  cur_data:    12      (正在读取或写入的数据)
ad 生产者消费者模拟:
    1:输入Pn/pn, 输入第n个生产者进程(例如P1,p12,P123)
    2:输入Cn/cn, 输入第n个消费者进程(例如C1,c12,C123)
    3:输入E/e, 执行 (ExeEnd)
    4:输入Q/q, 退出
    e
```

```
帮助
n.cpp
MakeList: szw@ubuntu: ~/文档/os-experiment/experiment2/build
1  BUFFER:
2  Empty_queue:
3  Full_queue:
4  Mutex_queue:
5  Mutex:      1
6  Full:       0
7  Empty:      8
8  cur_process: 0
9  cur_data:   0      (正在读取或写入的数据)
ad 生产者消费者模拟:
    1:输入Pn/pn, 输入第n个生产者进程(例如P1,p12,P123)
    2:输入Cn/cn, 输入第n个消费者进程(例如C1,c12,C123)
    3:输入E/e, 执行 (ExeEnd)
    4:输入Q/q, 退出
```

源码+注释:

Main.h

```
1. #include<bits/stdc++.h>
```

```

2. #include<stdio.h>
3. #include<sys/types.h>
4. #include<stdlib.h>
5. #include<sys/stat.h>
6. #include<fcntl.h>
7. #include<error.h>
8. #include<wait.h>
9. #include<unistd.h>
10. using namespace std;
11.
12. //对信号量 a 进行 P 原语操作
13. bool P(string a);
14. //对信号量 a 进行 V 原语操作
15. void V(string a);
16. //创建第 n 个消费者，会调用 P(Full)和 P(Mutex),全部 P 成功后就会进入 current_process，否则进入对应的阻塞队列
17. void consum(int n);
18. //创建第 n 个生产者，会调用 P(Empty)和 P(Mutex),全部 P 成功后就会进入 current_process，否则进入对应的阻塞队列。
19. void product(int n);
20. void Show();
21. void show_queue(queue<int> temp);
22. void show_queue(queue<string> temp);
23. //执行当前的 current_process，使其真正的完成写入/读取。
24. void exeend();
25. //若 Mutex_queue 不为空，则唤醒一个 Mutex_queue 中的进程，并把该进程置为 current_process;
26. bool mutex_proc();
27. //若 Empty_queue 不为空则唤醒一个 Empty_queue 中的进程
28. bool produce_proc();
29. //若 Full_queue 不为空则唤醒一个 Full _queue 中的进程。
30. bool consum_proc();

```

## Main.cpp

```

1. #include"main.h"
2. using namespace std;
3.
4.
5. const int BUFF_SIZE=8;//设置缓冲区大小为 8
6. queue<int> Full_que,Empty_que;
7. queue<string> Mutex_que;
8. queue<int> BUFFER;
9. int Empty,Full,Mutex;
10.

```

```

11. //第 n 个数据
12. int i=1;
13.
14. //cur 进程和数据
15. string cur_process="0";
16. int cur_data=0;
17.
18. int main(){
19.     // 初始化 默认刚开始时缓冲区为空
20.     Mutex=1;
21.     Empty=BUFF_SIZE;
22.     Full=0;
23.
24.     while(1){
25.         cout<<"\t 生产者消费者模拟: \t"<<endl;
26.         cout<<"1:输入 Pn/pn, 输入第 n 个生产者进程(例如 P1,p12,P123)"<<endl;
27.         cout<<"2:输入 Cn/cn, 输入第 n 个消费者进程(例如 C1,c12,C123)"<<endl;
28.         cout<<"3:输入 E/e, 执行 (ExeEnd)"<<endl;
29.         cout<<"4:输入 Q/q, 退出"<<endl;
30.         string temp;
31.         cin>>temp;
32.         system("clear");
33.         switch (temp[0]){
34.             case 'P':
35.                 case 'p':product(atoi(temp.substr(1).c_str()));Show();break;
36.                 case 'C':
37.                 case 'c':consum(atoi(temp.substr(1).c_str()));Show();break;
38.                 case 'E':
39.                 case 'e':exeend();Show();break;
40.                 case 'Q':
41.                 case 'q':return 0;
42.                 default:cout<<"Wrong Input!"<<endl;break;
43.             }
44.         }
45.         return 0;
46.     }
47.
48.
49. bool P(string a){
50.     if(a=="Empty"){
51.         Empty--;
52.         return Empty>=0;
53.     }
54.     else if(a=="Full"){

```

```
55.     Full--;
56.     return Full>=0;
57. }
58. else if(a=="Mutex"){
59.     Mutex--;
60.     return Mutex>=0;
61. }
62. return false;
63. }
64.
65. void V(string a){
66.     if(a=="Empty"){
67.         Empty++;
68.     }
69.     else if(a=="Full"){
70.         Full++;
71.     }
72.     else if(a=="Mutex"){
73.         Mutex++;
74.     }
75.     return;
76. }
77.
78. void consum(int n){
79.     if(!P("Full")){
80.         Full_queue.push(n);
81.     }
82.     else {
83.         if(!P("Mutex")){
84.             Mutex_queue.push("C"+to_string(n));
85.         }
86.         else{
87.             cur_process="C"+to_string(n);
88.             cur_data = BUFFER.front();
89.         }
90.     }
91.
92. }
93.
94.
95. void product(int n){
96.     if(!P("Empty")){
97.         Empty_queue.push(n);
98.     }
```

```

99.     else {
100.         if(!P("Mutex")){
101.             Mutex_que.push("P"+to_string(n));
102.         }
103.         else{
104.             cur_process="P"+to_string(n);
105.             cur_data=(i++);
106.         }
107.     }
108.
109. }
110.
111. void exeend(){
112.     if(cur_process=="0"){
113.         cout<<"现在没有 current_process!"<<endl;return;
114.     }
115.     else if(cur_process[0]=='P'){
116.         BUFFER.push(cur_data);
117.         V("Mutex");
118.         bool t=false;
119.         if(!mutex_proc()) {
120.             cur_data=0;
121.             cur_process="0";
122.             t=true;
123.         }
124.         V("Full");
125.         if(!consum_proc()&&t){
126.             cur_data=0;
127.             cur_process="0";
128.         }
129.     }
130.     else if(cur_process[0]=='C'){
131.         BUFFER.pop();
132.         V("Mutex");
133.         bool t=false;
134.         if(!mutex_proc()) {
135.             cur_data=0;
136.             cur_process="0";
137.             t=true;
138.         }
139.         V("Empty");
140.         if(!produce_proc()&&t){
141.             cur_data=0;
142.             cur_process="0";

```

```

143.     }
144. }
145.     return;
146. }
147.
148. bool produce_proc(){
149.     if(Empty_que.empty()) return false;
150.     int n=Empty_que.front();Empty_que.pop();
151.
152.     if(!P("Mutex")){
153.         Mutex_que.push("P"+to_string(n));
154.         return false;
155.     }
156.     else{
157.         cur_process="P"+to_string(n);
158.         cur_data=(i++);
159.         return true;
160.     }
161. }
162.
163.
164. bool consum_proc(){
165.     if(Full_que.empty()) return false;
166.     int n=Full_que.front();Full_que.pop();
167.
168.     if(!P("Mutex")){
169.         Mutex_que.push("C"+to_string(n));
170.         return false;
171.     }
172.     else{
173.         cur_process="C"+to_string(n);
174.         cur_data = BUFFER.front();
175.         return true;
176.     }
177. }
178.
179. bool mutex_proc(){
180.     if(Mutex_que.empty()) return false;
181.     string temp=Mutex_que.front();Mutex_que.pop();
182.     cur_process=temp;
183.     cur_data = temp[0]=='P' ? (i++):BUFFER.front();
184.     return true;
185. }
186.

```

```
187. void Show(){
188.     cout<<"BUFFER:\t\t";
189.     show_queue(BUFFER);
190.     cout<<"Empty_queue:\t";
191.     show_queue(Empty_que);
192.     cout<<"Full_queue:\t";
193.     show_queue(Full_que);
194.     cout<<"Mutex_queue:\t";
195.     show_queue(Mutex_que);
196.     cout<<"Mutex:\t\t"<<Mutex<<endl;
197.     cout<<"Full:\t\t"<<Full<<endl;
198.     cout<<"Empty:\t\t"<<Empty<<endl;
199.     cout<<"cur_process:\t"<<cur_process<<endl;
200.     cout<<"cur_data:\t"<<cur_data<<"\t(正在读取或写入的数据)"<<endl;
201.
202. }
203.
204. void show_queue(queue<int> temp){
205.     while(!temp.empty()){
206.         cout<<temp.front()<<" ";
207.         temp.pop();
208.     }
209.     cout<<endl;
210.     return;
211. }
212.
213. void show_queue(queue<string> temp){
214.     while(!temp.empty()){
215.         cout<<temp.front()<<" ";
216.         temp.pop();
217.     }
218.     cout<<endl;
219.     return;
220. }
```

### 实验题目 三、进程的管道通信

#### 实验简述:

加深对进程概念的理解,明确进程和程序的区别。  
学习进程创建的过程,进一步认识进程并发执行的实质。  
分析进程争用资源的现象,学习解决进程互斥的方法。  
学习解决进程同步的方法。  
掌握 Linux 系统中进程间通过管道通信的具体实现。

#### 实验内容

##### 完成了基础点:

使用系统调用 `pipe()` 建立一条管道,系统调用 `fork()` 分别创建两个子进程,它们分别向管道写一句话,如:

Child process1 is sending a message!

Child process2 is sending a message!

父进程分别从管道读出来自两个子进程的信息,显示在屏幕上

##### 完成了扩展点一:

完成父子进程间的双向通信,父进程发送一条消息,看哪一个子进程可以接收到并显示在屏幕上,同时父进程接受所有子进程发送的信息并显示在屏幕上。

##### 完成了扩展点二:

想办法让子进程间的互斥失效,让父子进程间的同步失效,可以观察到同步互斥失效的现象。

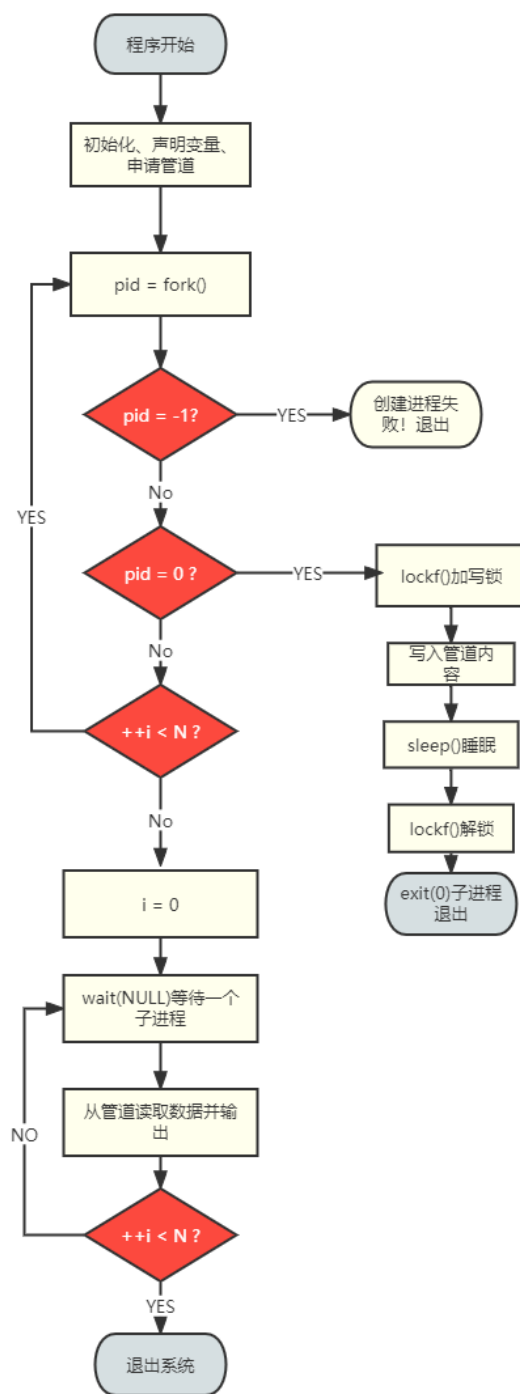
在 linux 系统的环境下使用 C++ 完成实验,使用了系统调用 `fork()`,`read()`,`write()`,`pipe()`,`lockf()`,`wait()`,`exit()`,`sleep()` 等。

##### Ppt 中的问题:

进程的互斥表现在不同的子进程的消息是分别独立完整的,而不是混乱的。同步的表现在只有所有的子进程都结束了父进程才会结束。互斥是使用了 `lockf()` 的系统调用来实现,同步是使用了 `wait(NULL)` 来实现的。

#### 基础点程序的主要流程图:





**扩展点一**即新建另外一个管道，父进程在循环  $N$  次创建完进程后，向管道发送消息后再等待接收子进程的消息。而子进程创建后先等待接收父进程的消息，接收到以后再向父进程发送消息。这里子进程再读取消息前需要先用系统调用 `close()` 关闭掉写端防止因一直接收不到消息而阻塞。

**扩展点二**需要在子进程中，写入的时候不加写锁和读锁，并且需要使用特定的系统调用来关闭掉阻塞防止互斥。而在父进程中不必使用 `wait(NULL)`，以此关闭掉同步。

**实验结果（截图）**

**基础点：**

```
CMakeLists.txt  main.cpp  1.cpp  2.cpp
main.cpp > main(int, char *[])
1  /**
2   * @author:孙
3   * @brief:基础点
4   */
5  #include<bits/stdc++.h>
6  #include<stdio.h>
7  #include<sys/types.h>
8  #include<stdlib.h>
9  #include<sys/stat.h>
10 #include<fcntl.h>
11 #include<error.h>
12 #include<wait.h>
13 #include<unistd.h>
14 using namespace std;
15
问题 输出 终端 调试控制台
=thread-group-added,id="i1"
Warning: Debuggee TargetArchitecture not detected.
=cmd-param-changed,param="pagination",value="off"
Stopped due to shared library event (no libraries
Loaded '/lib64/ld-linux-x86-64.so.2'. Symbols loaded.

szw@ubuntu: ~/文档/os-experiment/experiment3/build
./experiment3
Child process 2 is sending a message! 它的pid:3090 它的父进程的pid:3088
Child process 5 is sending a message! 它的pid:3093 它的父进程的pid:3088
Child process 1 is sending a message! 它的pid:3089 它的父进程的pid:3088
Child process 6 is sending a message! 它的pid:3094 它的父进程的pid:3088
Child process 3 is sending a message! 它的pid:3091 它的父进程的pid:3088
Child process 4 is sending a message! 它的pid:3092 它的父进程的pid:3088
szw@ubuntu:~/文档/os-experiment/experiment3/build$ ./experiment3
Child process 1 is sending a message! 它的pid:3096 它的父进程的pid:3095
Child process 4 is sending a message! 它的pid:3099 它的父进程的pid:3095
Child process 6 is sending a message! 它的pid:3101 它的父进程的pid:3095
Child process 5 is sending a message! 它的pid:3100 它的父进程的pid:3095
Child process 3 is sending a message! 它的pid:3098 它的父进程的pid:3095
Child process 2 is sending a message! 它的pid:3097 它的父进程的pid:3095
szw@ubuntu:~/文档/os-experiment/experiment3/build$ ./experiment3
Child process 1 is sending a message! 它的pid:3103 它的父进程的pid:3102
Child process 3 is sending a message! 它的pid:3105 它的父进程的pid:3102
Child process 2 is sending a message! 它的pid:3104 它的父进程的pid:3102
Child process 4 is sending a message! 它的pid:3106 它的父进程的pid:3102
Child process 5 is sending a message! 它的pid:3107 它的父进程的pid:3102
Child process 6 is sending a message! 它的pid:3108 它的父进程的pid:3102
szw@ubuntu:~/文档/os-experiment/experiment3/build$
```

可以观察到子进程发送消息的顺序是随机的。

扩展点一：

```
1.cpp - experiment3 - Visual Studio Code
终端 帮助
CMakeLists.txt  main.cpp  1.cpp  2.cpp
1.cpp > main(int, char *[])
1  /**
2   * @author:孙
3   * @brief:扩展一 双向发送消息
4   */
5  #include<bits/stdc++.h>
6  #include<stdio.h>
7  #include<sys/types.h>
8  #include<stdlib.h>
9  #include<sys/stat.h>
10 #include<fcntl.h>
11 #include<error.h>
12 #include<wait.h>
13 #include<unistd.h>
14 using namespace std;
15
16 #define BUFF_SIZE 512
17 const int N = 5; // 创建的线程
18
19 int main(int argc, char* arg
20     int fd[2], fd_x[2];
21     pid_t pid;
22     if(pipe(fd) == -1 || pip
23         fprintf(stderr, "创建
24         return -1;

szw@ubuntu:~/文档/os-experiment/experiment3/build$ ./experiment3
No.3 child process get the father's message! :Father process is sending a message!
Child process 3 is sending a message! 它的pid:4113 它的父进程的pid:4110
Child process 1 is sending a message! 它的pid:4111 它的父进程的pid:4110
Child process 4 is sending a message! 它的pid:4114 它的父进程的pid:4110
Child process 2 is sending a message! 它的pid:4112 它的父进程的pid:4110
Child process 5 is sending a message! 它的pid:4115 它的父进程的pid:4110
szw@ubuntu:~/文档/os-experiment/experiment3/build$ ./experiment3
No.4 child process get the father's message! :Father process is sending a message!
Child process 4 is sending a message! 它的pid:4123 它的父进程的pid:4119
Child process 5 is sending a message! 它的pid:4124 它的父进程的pid:4119
Child process 3 is sending a message! 它的pid:4122 它的父进程的pid:4119
Child process 2 is sending a message! 它的pid:4121 它的父进程的pid:4119
Child process 1 is sending a message! 它的pid:4120 它的父进程的pid:4119
szw@ubuntu:~/文档/os-experiment/experiment3/build$ ./experiment3
No.2 child process get the father's message! :Father process is sending a message!
Child process 2 is sending a message! 它的pid:4127 它的父进程的pid:4125
Child process 5 is sending a message! 它的pid:4130 它的父进程的pid:4125
Child process 1 is sending a message! 它的pid:4126 它的父进程的pid:4125
Child process 3 is sending a message! 它的pid:4128 它的父进程的pid:4125
Child process 4 is sending a message! 它的pid:4129 它的父进程的pid:4125
szw@ubuntu:~/文档/os-experiment/experiment3/build$ ./experiment3
No.1 child process get the father's message! :Father process is sending a message!
Child process 1 is sending a message! 它的pid:4133 它的父进程的pid:4132
Child process 2 is sending a message! 它的pid:4134 它的父进程的pid:4132
Child process 3 is sending a message! 它的pid:4135 它的父进程的pid:4132
Child process 4 is sending a message! 它的pid:4136 它的父进程的pid:4132
Child process 5 is sending a message! 它的pid:4137 它的父进程的pid:4132
szw@ubuntu:~/文档/os-experiment/experiment3/build$
```

可以观察到子进程收到了父进程的消息，并且是随机一个子进程收到消息。

扩展点二：

正常有同步互斥：





```

3.  * @brief:基础点
4.  **/
5. #include<bits/stdc++.h>
6. #include<stdio.h>
7. #include<sys/types.h>
8. #include<stdlib.h>
9. #include<sys/stat.h>
10. #include<fcntl.h>
11. #include<error.h>
12. #include<wait.h>
13. #include<unistd.h>
14. using namespace std;
15.
16. #define BUFF_SIZE 512
17. const int N = 6; // 创建的线程的数量
18.
19. int main(int argc, char* argv[]){
20.     int fd[2];
21.     pid_t pid;
22.     if(pipe(fd) == -1){
23.         fprintf(stderr, "创建管道失败! " );
24.         return -1;
25.     }
26.     for(int i = 1; i <= N; i++){
27.         pid = fork();
28.         if(pid == -1){
29.             fprintf(stderr, "创建进程失败! " );
30.             return -1;
31.         }
32.         /**
33.          * pipe 的 write 和 read 的长度一定要相同, 这里全部设置为了 sizeof(char)*BUFF_SIZE
34.          * 对于 char str[BUFF_SIZE];出来的栈空间, sizeof(str)就直接为全部的长度
35.          * 而 对于 char* str = new char[BUFF_SIZE]; 出来的堆空间 需要
36.          * sizeof(char)*BUFF_SIZE.
37.          */
38.         else if(pid == 0){
39.             lockf(fd[1], 1, 0); //管道的写锁
40.             // char str[BUFF_SIZE];
41.             char* str = new char[BUFF_SIZE];
42.             sprintf(str, "Child process %d is sending a message! 它的pid:%d 它的父进程
43. 的pid:%d\n", i, getpid(), getppid());
44.             //cout << str <<endl;
45.             write(fd[1], str, sizeof(char)*BUFF_SIZE); // 写入管道
46.             // sleep(1); // 进入睡眠

```



```

45.         lockf(fd[1], 0, 0); // 释放锁
46.
47.         exit(0); // 结束子进程
48.     }
49. }
50. for(int i = 1; i <= N ; i++){
51.     char buff[BUF_SIZE];
52.     wait(NULL); // 等待一个子线程结束
53.     // cout << wait(NULL) << endl;
54.     read(fd[0], buff, sizeof(buff)); //从管道中读取
55.     cout << buff;
56. }
57.
58. return 0;
59. }

```

扩展点一代码：

```

1. /**
2.  * @author:孙
3.  * @brief:扩展一 双向发送消息
4.  */
5. #include<bits/stdc++.h>
6. #include<stdio.h>
7. #include<sys/types.h>
8. #include<stdlib.h>
9. #include<sys/stat.h>
10. #include<fcntl.h>
11. #include<error.h>
12. #include<wait.h>
13. #include<unistd.h>
14. using namespace std;
15.
16. #define  BUF_SIZE  512
17. const int N = 5; // 创建的线程的数量 可随意更改
18.
19. int main(int argc, char* argv){
20.     int fd[2], fd_x[2];
21.     pid_t pid;
22.     if(pipe(fd) == -1 || pipe(fd_x) == -1){
23.         fprintf(stderr, "创建管道失败！ ");
24.         return -1;
25.     }
26.     for(int i = 1; i <= N; i++){
27.         pid = fork();
28.         if(pid == -1){

```

```

29.         fprintf(stderr, "创建进程失败! ");
30.         return -1;
31.     }
32.     /**
33.      * pipe 的 write 和 read 的长度一定要相同, 这里全部设置为了 sizeof(char)*BUFF_SIZE
34.      * 对于 char str[BUFF_SIZE];出来的栈空间, sizeof(str)就直接为全部的长度
35.      * 而 对于 char* str = new char[BUFF_SIZE]; 出来的堆空间 需要
        sizeof(char)*BUFF_SIZE.
36.      */
37.     else if(pid == 0){
38.         // 读取父进程发送的消息
39.         char buff[BUFF_SIZE];
40.         close(fd_x[1]); // 先关闭掉写端 防止阻塞
41.         if(read(fd_x[0], buff, BUFF_SIZE) != 0) {
42.             printf("No.%d child process get the father's message! :%s\n", i, buff);
43.         }
44.
45.         // 向父进程发送消息
46.         lockf(fd[1], 1, 0); //管道的写锁
47.         // char str[BUFF_SIZE];
48.         char* str = new char[BUFF_SIZE];
49.         sprintf(str, "Child process %d is sending a message! 它的pid:%d 它的父进程
        的pid:%d\n", i, getpid(), getppid());
50.         //cout << str <<endl;
51.         write(fd[1], str, sizeof(char)*BUFF_SIZE); // 写入管道
52.         sleep(1); // 进入睡眠
53.         lockf(fd[1], 0, 0); // 释放锁
54.
55.         exit(0); // 结束子进程
56.     }
57. }
58.
59. //发送给子进程消息
60. char str[] = {"Father process is sending a message!"};
61. write(fd_x[1], str, sizeof(str));
62. close(fd_x[1]);
63.
64. for(int i = 1; i <= N ; i++){
65.     // 接受子进程的消息
66.     char buff[BUFF_SIZE];
67.     wait(NULL); // 等待一个子线程结束
68.     // cout << wait(NULL) << endl;
69.     read(fd[0], buff, sizeof(buff)); //从管道中读取
70.     cout << buff;

```

```
71.     }
72.
73.     return 0;
74. }
```

扩展点二代码:

```
1.  /**
2.   * @author:孙
3.   * @brief:取消掉父子进程的默认同步
4.   **/
5.  #include<bits/stdc++.h>
6.  #include<stdio.h>
7.  #include<sys/types.h>
8.  #include<stdlib.h>
9.  #include<sys/stat.h>
10. #include<fcntl.h>
11. #include<error.h>
12. #include<wait.h>
13. #include<unistd.h>
14. using namespace std;
15.
16. // 是否开启子进程之间的互斥 注释掉为不互斥
17. // #define Mutex
18. // 是否开启父子进程之间的同步 注释掉为不同步
19. // #define Sync
20.
21. const int N = 4; // 创建的线程的数量
22. const int NUM = 512;
23. int main(int argc, char* argv[]){
24.     int fd[2];
25.     pid_t pid;
26.     if(pipe(fd) == -1){
27.         fprintf(stderr, "创建管道失败! ");
28.         return -1;
29.     }
30.     for(int i = 1; i <= N; i++){
31.         pid = fork();
32.         if(pid == -1){
33.             fprintf(stderr, "创建进程失败! ");
34.             return -1;
35.         }
36.         else if(pid == 0){
37.             #ifdef Mutex
38.                 lockf(fd[1], 1, 0); //管道的写锁
39.             #endif
```



```
40.
41.     // 非阻塞
42.     int flags = fcntl(fd[1], F_GETFL, 0);
43.     fcntl(fd[1], F_SETFL, flags|O_NONBLOCK );
44.
45.     char p[NUM];
46.     memset(p, i+'0', sizeof(p));
47.     p[NUM-1] = '\0';
48.     // write(fd[1], p, sizeof(p));
49.     for(int j = 0; j < NUM ; j ++){
50.         write(fd[1], p+j, sizeof(char)); // 写入 NUM 个 i
51.     }
52.
53.     // sleep(1); // 进入睡眠
54.     #ifdef Mutex
55.         lockf(fd[1], 0, 0); // 释放锁 0
56.     #endif
57.     exit(0); // 结束子进程
58. }
59. }
60. // sleep(0.05);
61.
62. // 非阻塞
63. int flags = fcntl(fd[0], F_GETFL, 0);
64. fcntl(fd[0], F_SETFL, flags|O_NONBLOCK );
65.
66. for(int i = 1; i <= N ; i++){
67.     #ifdef Sync
68.         wait(NULL); // 等待一个子线程结束
69.     #endif
70.     char buff[NUM];
71.     read(fd[0], buff, sizeof(buff)); // 从管道中读取
72.     cout << buff;
73.
74. }
75. cout<<endl;
76.
77. return 0;
78. }
```

## 实验题目 四、页面置换算法

### 实验简述

进一步加深理解父子进程之间的关系及其并发执行。理解内存页面调度的机理。掌握页面置换算法及其实现方法。培养综合运用所学知识的能力。

页面置换算法是虚拟存储管理实现的关键，通过本次试验理解内存页面调度的机制，在模拟实现 FIFO、LRU 等经典页面置换算法的基础上，理解虚拟存储实现的过程。

将不同的置换算法放在不同的子进程中加以模拟，培养综合运用所学知识的能力。

### 实验内容

程序涉及一个父进程和两个子进程。父进程使用 `rand()` 函数随机产生若干随机数，经过处理后，存于一数组 `Acess_Series[]` 中，作为内存页面访问的序列。两个子进程根据这个访问序列，分别采用 FIFO 和 LRU 两种不同的页面置换算法对内存页面进行调度。同时统计不同的页面调度算法的缺页率。

在 linux 系统的环境下使用 C++ 并用 g++ 进行编译完成实验，使用了系统调用 `fork()`, `wait()`, `exit()`, `sleep()`, `rand()` 等。

可以随意修改程序序列的长度、内存大小：

```
const int MAX_FRAME = 3; // 一个进程的最多的实际分配的物理页面数
const int K = 6; // 程序逻辑页面的面数 1~K
const int N = 16; // 随机访问序列的长度
```

### Ppt 中的问题：

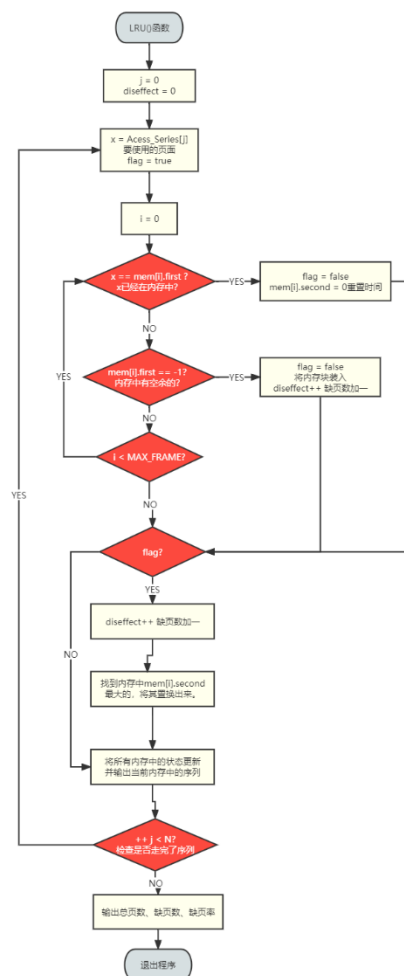
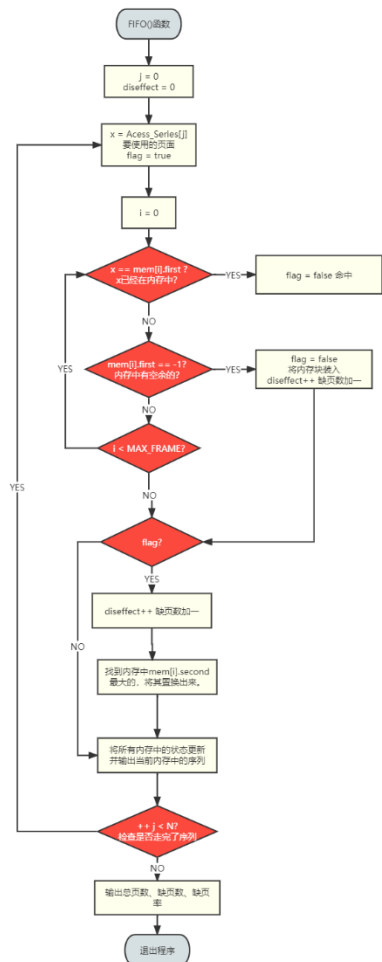
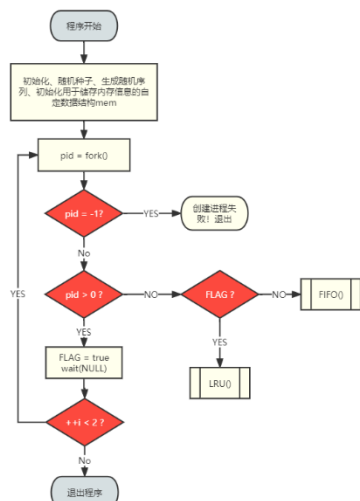
父进程的空间与子进程的空间是一模一样但是又相互独立的，子进程在被创建的时候会复制父进程的内存空间，获得一模一样的变量和状态。但是接下来父子进程分别独立运行，变量互不干扰。

虚拟寄存器就是在使用到某个页面的时候再将该页面置入内存，原理是程序并不需要在同一时刻使用所有的页面。以此解决了程序所需要的内存远大于物理内存的问题。

我使用的 FIFO 算法中 belady 现象的序列为 {3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4}，驻留集分别为 3、4 时可以观察到 belady 现象。

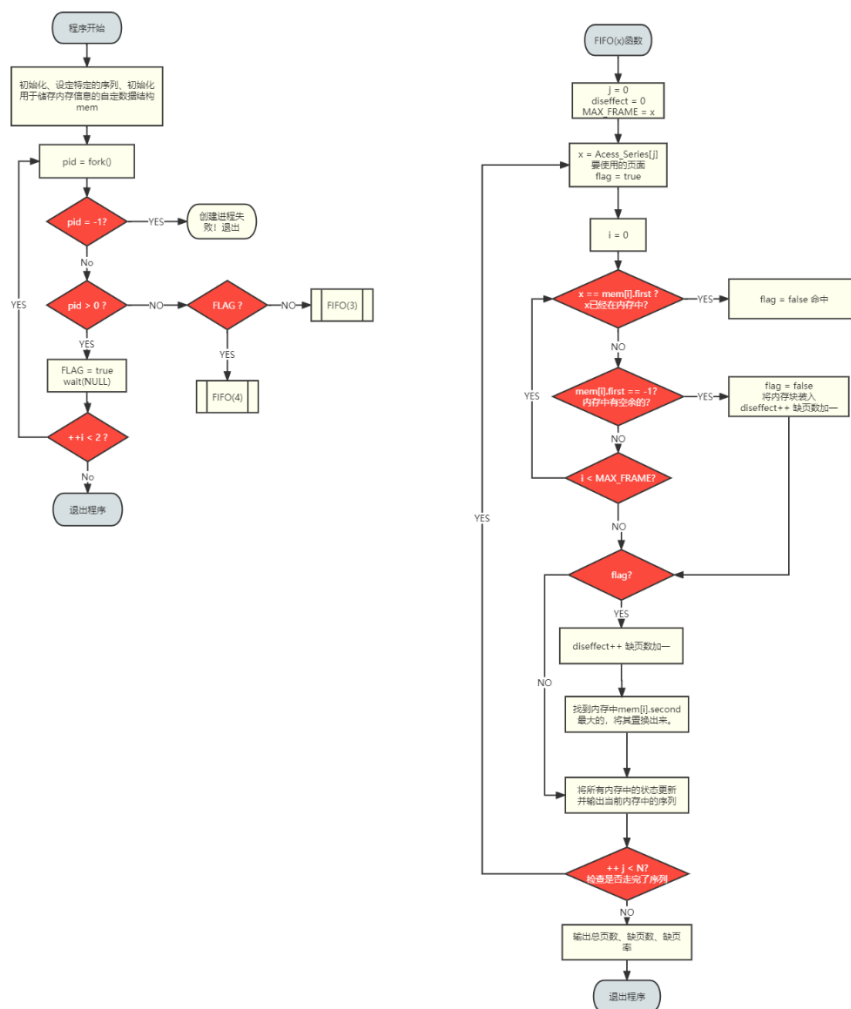
**完成了基础点：**程序用父进程创建两个子进程，父进程产生随机序列并完成初始化，两个子进程分别实现 FIFO 算法和 LRU 算法，并分别输出每个状态的内存中页帧状态，最终输出缺页率。

基础点的程序大体流程图如下：



完成了扩展点一：可以使用父进程创建了两个子进程,两个子进程分别使用不同大小的驻留集,都是用 FIFO 的置换算法,最终可以观察到驻留集更大的子进程反而缺页率更高。

扩展点一的流程图:



完成了扩展点二：父进程创建了四个子进程,四个子进程分别使用 FIFO、LRU、CLOCK、改进型 CLOCK 算法,分别输出每个状态的内存中页帧状态,并最终输出缺页率。

实验流程与基础点类似,但是需要多创建两个子进程,分别来执行 CLOCK 算法和改进型 CLOCK 算法。CLOCK 算法使用数组来模拟队列,每次对下标取余。而改进型 CLOCK 算法使用 `pair<int,int>` 来分别储存修改位和访问位。并且再改进型 CLOCK 算法中,每次随机对内存中的页面进行修改并将他们的修改位置为 1。

实验结果（截图）

基础点:

```

root@localhost:~/os4# g++ main.cpp -o main
root@localhost:~/os4# ./main
页面序列: 4 6 6 4 5 5 6 5 3 3 5 6 6 1 6 6
FIFO! 访问页面      当前内存
缺页! 4              4      -1      -1
缺页! 6              4      6      -1
        6              4      6      -1
        4              4      6      -1
缺页! 5              4      6      5
        5              4      6      5
        6              4      6      5
        5              4      6      5
缺页! 3              3      6      5
        3              3      6      5
        5              3      6      5
        6              3      6      5
        6              3      6      5
缺页! 1              3      1      5
缺页! 6              3      1      6
        6              3      1      6
缺页共6个, 总页面16个, 缺页率: 0.375
LRU! 访问页面      当前内存
缺页! 4              4      -1      -1
缺页! 6              4      6      -1
        6              4      6      -1
        4              4      6      -1
缺页! 5              4      6      5
        5              4      6      5
        6              4      6      5
        5              4      6      5
缺页! 3              3      6      5
        3              3      6      5
        5              3      6      5
        6              3      6      5
        6              3      6      5
缺页! 1              1      6      5
        6              1      6      5
        6              1      6      5
缺页共5个, 总页面16个, 缺页率: 0.3125
root@localhost:~/os4# █

```

扩展点一:

```

root@localhost:~/os4# g++ 1.cpp -o 1
root@localhost:~/os4# ./1
页面序列: 3 2 1 0 3 2 4 3 2 1 0 4
有3块物理块!
FIFO!: 访问页面      当前内存
缺页!  3          3      -1      -1
缺页!  2          3      2       -1
缺页!  1          3      2       1
缺页!  0          0      2       1
缺页!  3          0      3       1
缺页!  2          0      3       2
缺页!  4          4      3       2
      3          4      3       2
      2          4      3       2
缺页!  1          4      1       2
缺页!  0          4      1       0
      4          4      1       0
缺页共9个, 总页面12个, 缺页率: 0.75
有4块物理块!
FIFO!: 访问页面      当前内存
缺页!  3          3      -1      -1      -1
缺页!  2          3      2      -1      -1
缺页!  1          3      2      1      -1
缺页!  0          3      2      1      0
      3          3      2      1      0
      2          3      2      1      0
缺页!  4          4      2      1      0
缺页!  3          4      3      1      0
缺页!  2          4      3      2      0
缺页!  1          4      3      2      1
缺页!  0          0      3      2      1
缺页!  4          0      4      2      1
缺页共10个, 总页面12个, 缺页率: 0.833333
root@localhost:~/os4#

```

可以看到驻留集由 **3** 增加为 **4**, 缺页率反而增加了。

扩展点二:

```

root@localhost:~/os4# g++ 2.cpp -o 2
root@localhost:~/os4# ./2
页面序列: 5 2 2 4 2 1 3 2 2 2 4 2
FIFO!:
访问页面      当前内存
缺页!  5      5      -1      -1
缺页!  2      5      2      -1
      2      5      2      -1
缺页!  4      5      2      4
      2      5      2      4
缺页!  1      1      2      4
缺页!  3      1      3      4
缺页!  2      1      3      2
      2      1      3      2
      2      1      3      2
缺页!  4      4      3      2
      2      4      3      2
缺页共7个, 总页面12个, 缺页率: 0.583333
LRU!:
访问页面      当前内存
缺页!  5      5      -1      -1
缺页!  2      5      2      -1
      2      5      2      -1
缺页!  4      5      2      4
      2      5      2      4
缺页!  1      1      2      4
缺页!  3      1      2      3
      2      1      2      3
      2      1      2      3
      2      1      2      3
缺页!  4      4      2      3
      2      4      2      3
缺页共6个, 总页面12个, 缺页率: 0.5
CLOCK!:
访问页面      当前内存
缺页!  5      5      -1      -1
缺页!  2      5      2      -1

```

```

CLOCK!:
访问页面      当前内存
缺页!  5      5      -1      -1
缺页!  2      5      2      -1
      2      5      2      -1
缺页!  4      5      2      4
      2      5      2      4
缺页!  1      1      2      4
缺页!  3      1      3      4
缺页!  2      1      3      2
      2      1      3      2
      2      1      3      2
      2      1      3      2
缺页!  4      4      3      2
      2      4      3      2
缺页共7个, 总页面12个, 缺页率: 0.583333
改进型CLOCK!:
访问页面      当前内存
缺页!  5      5      -1      -1
缺页!  2      5      2      -1
      2      5      2      -1
缺页!  4      5      2      4
      2      5      2      4
缺页!  1      5      2      1
缺页!  3      3      2      1
      2      3      2      1
      2      3      2      1
      2      3      2      1
      2      3      2      1
缺页!  4      3      2      4
      2      3      2      4
缺页共6个, 总页面12个, 缺页率: 0.5
root@localhost:~/os4# █

```

可以看到，同时创建了 4 个子进程分别完成了 FIFO,LRU,CLOCK,改进型 CLOCK 算法。

源码+注释:

基础点:

```
1.  /**
2.   * @author:孙
3.   * @brief:基础点 两个子进程分别完成 FIFO 和 LRU
4.   **/
5. #include<bits/stdc++.h>
6. #include<stdio.h>
7. #include<sys/types.h>
8. #include<stdlib.h>
9. #include<sys/stat.h>
10. #include<fcntl.h>
11. #include<error.h>
12. #include<wait.h>
13. #include<unistd.h>
14. using namespace std;
15.
16.
17. const int MAX_FRAME = 3; //一个进程的最多的实际分配的物理页面数
18. const int K = 6; //程序逻辑页面的面数 1~K
19. const int N = 20; //随机访问序列的长度
20.
21.
22.
23. int Acess_Series[N]; // 随机访问序列
24. typedef pair<int, int> PII;
25. PII mem[MAX_FRAME]; // 内存中的页面编号 和对应算法记录的值
26.
27. void FIFO(){
28.     cout<<"FIFO!:\t"<<"访问页面\t"<<"\t 当前内存"<<endl;
29.     int diseffect = 0; //缺页数
30.     for(int j = 0; j < N; j++){
31.         int x = Acess_Series[j]; //要使用的页面
32.         bool flag = true;
33.         for(int i = 0; i < MAX_FRAME; i++){
34.             if(x == mem[i].first ){
35.                 flag = false;
36.                 cout<<"\t";
37.                 break; //命中
```



```

38.         }
39.         else if(mem[i].first == -1){
40.             mem[i].first = x;
41.             mem[i].second = 0;
42.             flag = false;
43.             diseffect++;
44.             cout <<"缺页!\t";
45.             break; //有空余 也是缺页
46.         }
47.     }
48.     if(flag){
49.         diseffect++;
50.         int t = 0, ma=0;
51.         // 找出时间最久的
52.         for(int i = 0; i < MAX_FRAME; i++){
53.             if(mem[i].second > ma) t = i, ma = mem[i].second;
54.         }
55.         //置换掉
56.         cout <<"缺页!\t";
57.         mem[t].first = x;
58.         mem[t].second = 0;
59.     }
60.     // 进入的时间加一
61.     for(int i = 0; i < MAX_FRAME; i++){
62.         if(mem[i].first != -1) mem[i].second ++;
63.     }
64.     //输出状态
65.     cout << x <<"\t\t";
66.     for(int i = 0; i < MAX_FRAME; i++){
67.         cout << mem[i].first << "\t";
68.     }
69.     puts("");
70. }
71. cout <<"缺页共" << diseffect <<"个，总页面" << N <<"个，缺页率：
    " << (double) diseffect/N<<endl;
72. return ;
73. }
74.
75. void LRU(){
76.     cout<<"LRU!:\t"<<"访问页面\t"<<"\t 当前内存"<<endl;
77.     int diseffect = 0;
78.     for(int j = 0; j < N; j++){
79.         int x = Acess_Series[j]; //要使用的页面
80.         bool flag = true;

```

```

81.         for(int i = 0; i < MAX_FRAME; i++){
82.             if(x == mem[i].first ){
83.                 flag = false;
84.                 mem[i].second = 0; // 重置访问时间 与 FIFO 的唯一区别在于此罢了
85.                 cout<<"\t";
86.                 break; //命中
87.             }
88.             else if(mem[i].first == -1){
89.                 mem[i].first = x;
90.                 mem[i].second = 0;
91.                 flag = false;
92.                 diseffect++;
93.                 cout <<"缺页!\t";
94.                 break; //有空余 也是缺页
95.             }
96.         }
97.         if(flag){
98.             diseffect++;
99.             int t = 0, ma=0;
100.            // 找出时间最久的
101.            for(int i = 0; i < MAX_FRAME; i++){
102.                if(mem[i].second > ma) t = i, ma = mem[i].second;
103.            }
104.            //置换掉
105.            cout <<"缺页!\t";
106.            mem[t].first = x;
107.            mem[t].second = 0;
108.        }
109.        // 进入的时间加一
110.        for(int i = 0; i < MAX_FRAME; i++){
111.            if(mem[i].first != -1) mem[i].second ++;
112.        }
113.        //输出状态
114.        cout << x <<"\t\t";
115.        for(int i = 0; i < MAX_FRAME; i++){
116.            cout << mem[i].first << "\t";
117.        }
118.        puts("");
119.    }
120.    cout <<"缺页共" << diseffect <<"个，总页面" << N <<"个，缺页率：
    " << (double) diseffect/N<<endl;
121.    return ;
122. }
123.

```

```
124.
125.  int main(){
126.      srand((unsigned int)(time(NULL)));
127.      // 随机生成 20 个 [1,K] 之间的数 放入 Acess_Series 访问序列数组。
128.      for(int i = 0; i < N; i++){
129.          Acess_Series[i] = rand() % K + 1;
130.      }
131.      cout << "页面序列: ";
132.      for(int i = 0; i < N; i++){
133.          cout << Acess_Series[i] << " ";
134.      }
135.      puts("");
136.
137.      //初始化 first 为编号 second 为记录的值。
138.      for(int i = 0; i < MAX_FRAME; i++){
139.          mem[i].first = -1;
140.          mem[i].second = 0;
141.      }
142.
143.      pid_t pid;
144.      bool FLAG = false;
145.      for(int i = 1; i <= 2; i++){
146.          pid = fork();
147.          if(pid == -1){
148.              fprintf(stderr, "创建进程失败! ");
149.              return -1;
150.          }
151.          else if(pid > 0){
152.              FLAG = true;
153.              wait(NULL);
154.          }
155.          else{
156.              if(FLAG) {
157.                  LRU();
158.              }
159.              else {
160.                  FIFO();
161.              }
162.              exit(0);
163.          }
164.      }
165.      return 0;
166.  }
```

### 扩展点一：

```
1.  /**
2.   * @author:孙
3.   * @brief:扩展点一 观察 FIFO 的 Belady 现象 两个线程分别使用不同大小的物理块数
4.   */
5. #include<bits/stdc++.h>
6. #include<stdio.h>
7. #include<sys/types.h>
8. #include<stdlib.h>
9. #include<sys/stat.h>
10. #include<fcntl.h>
11. #include<error.h>
12. #include<wait.h>
13. #include<unistd.h>
14. using namespace std;
15.
16. typedef pair<int, int> PII;
17. const int N = 12;
18. //用于观察 belady 现象的页面序列
19. int Acess_Series[] = {3,2,1,0,3,2,4,3,2,1,0,4};
20.
21. void FIFO(int MAX_FRAME);
22.
23. int main(){
24.     cout << "页面序列: ";
25.     for(int i = 0; i < N; i++){
26.         cout << Acess_Series[i] <<" ";
27.     }
28.     puts("");
29.
30.     pid_t pid;
31.     bool FLAG = false;
32.     for(int i = 1; i <= 2; i++){
33.         pid = fork();
34.         if(pid == -1){
35.             fprintf(stderr, "创建进程失败! ");
36.             return -1;
37.         }
38.         else if(pid > 0){
39.             FLAG = true;
40.             wait(NULL);
41.         }
42.         else{
43.             if(FLAG) {
```

```

44.         FIFO(4);
45.     }
46.     else {
47.         FIFO(3);
48.     }
49.     exit(0);
50. }
51. }
52. return 0;
53. }
54.
55. /**
56.  * @brief: 有 k 个内存页面的 FIFO 算法
57.  *
58.  */
59. void FIFO(int MAX_FRAME){
60.     cout <<"有"<<MAX_FRAME<<"块物理块！"<<endl;
61.     PII mem[MAX_FRAME]; // 内存中的页面编号 和对应算法记录的值
62.     //初始化 first 为编号 second 为记录的值。
63.     for(int i = 0; i < MAX_FRAME; i++){
64.         mem[i].first = -1;
65.         mem[i].second = 0;
66.     }
67.     cout<<"FIFO!:\t"<<"访问页面\t"<<"\t 当前内存"<<endl;
68.     int diseffect = 0;//缺页数
69.     for(int i = 0; i < N; i++){
70.         int x = Acess_Series[i]; //要使用的页面
71.         bool flag = true;
72.         for(int i = 0; i < MAX_FRAME; i++){
73.             if(x == mem[i].first ){
74.                 flag = false;
75.                 cout<<"\t";
76.                 break; //命中
77.             }
78.             else if(mem[i].first == -1){
79.                 mem[i].first = x;
80.                 mem[i].second = 0;
81.                 flag = false;
82.                 diseffect++;
83.                 cout <<"缺页!\t";
84.                 break; //有空余 也是缺页
85.             }
86.         }
87.         if(flag){

```

```

88.         diseffect++;
89.         int t = 0, ma=0;
90.         // 找出时间最久的
91.         for(int i = 0; i < MAX_FRAME; i++){
92.             if(mem[i].second > ma) t = i, ma = mem[i].second;
93.         }
94.         //置换掉
95.         cout <<"缺页!\t";
96.         mem[t].first = x;
97.         mem[t].second = 0;
98.     }
99.     // 进入的时间加一
100.    for(int i = 0; i < MAX_FRAME; i++){
101.        if(mem[i].first != -1) mem[i].second ++;
102.    }
103.    //输出状态
104.    cout << x <<"\t\t";
105.    for(int i = 0; i < MAX_FRAME; i++){
106.        cout << mem[i].first << "\t";
107.    }
108.    puts("");
109. }
110. cout <<"缺页共" << diseffect <<"个，总页面" << N <<"个，缺页率：
    " << (double) diseffect/N<<endl;
111.     return ;
112. }

```

## 扩展点二：

```

1.  /**
2.   * @author:孙
3.   * @brief:扩展点2 四个子进程分别完成 FIFO,LRU,CLOCK,CLOCK 改进型
4.   */
5. #include<bits/stdc++.h>
6. #include<stdio.h>
7. #include<sys/types.h>
8. #include<stdlib.h>
9. #include<sys/stat.h>
10. #include<fcntl.h>
11. #include<error.h>
12. #include<wait.h>
13. #include<unistd.h>
14. using namespace std;
15.
16. const int MAX_FRAME = 4; //一个进程的最多的实际分配的物理页面数

```

```

17. const int K = 6; //程序逻辑页面的面数 1~K
18. const int N = 16; //随机访问序列的长度
19.
20. int Acess_Series[N]; // 随机访问序列
21. typedef pair<int, int> PII;
22. PII mem[MAX_FRAME]; // 内存中的页面编号 和对应算法记录的值
23.
24. void FIFO(){
25.     cout<<"FIFO!\n"<<"访问页面\t"<<"\t 当前内存"<<endl;
26.     int diseffect = 0; //缺页数
27.     for(int j = 0; j < N; j++){
28.         int x = Acess_Series[j]; //要使用的页面
29.         bool flag = true;
30.         for(int i = 0; i < MAX_FRAME; i++){
31.             if(x == mem[i].first ){
32.                 flag = false;
33.                 cout<<"\t";
34.                 break; //命中
35.             }
36.             else if(mem[i].first == -1){
37.                 mem[i].first = x;
38.                 mem[i].second = 0;
39.                 flag = false;
40.                 diseffect++;
41.                 cout <<"缺页!\t";
42.                 break; //有空余 也是缺页
43.             }
44.         }
45.         if(flag){
46.             diseffect++;
47.             int t = 0, ma=0;
48.             // 找出时间最久的
49.             for(int i = 0; i < MAX_FRAME; i++){
50.                 if(mem[i].second > ma) t = i, ma = mem[i].second;
51.             }
52.             //置换掉
53.             cout <<"缺页!\t";
54.             mem[t].first = x;
55.             mem[t].second = 0;
56.         }
57.         // 进入的时间加一
58.         for(int i = 0; i < MAX_FRAME; i++){
59.             if(mem[i].first != -1) mem[i].second ++;
60.         }

```

```

61.         //输出状态
62.         cout << x << "\t\t";
63.         for(int i = 0; i < MAX_FRAME; i++){
64.             cout << mem[i].first << "\t";
65.         }
66.         puts("");
67.     }
68.     cout << "缺页共" << diseffect << "个, 总页面" << N << "个, 缺页率:
    " << (double) diseffect/N << endl;
69.     return ;
70. }
71.
72. void LRU(){
73.     cout << "LRU!\n" << "访问页面\t" << "\t 当前内存" << endl;
74.     int diseffect = 0;
75.     for(int j = 0; j < N; j++){
76.         int x = Acess_Series[j]; //要使用的页面
77.         bool flag = true;
78.         for(int i = 0; i < MAX_FRAME; i++){
79.             if(x == mem[i].first ){
80.                 flag = false;
81.                 mem[i].second = 0; // 重置访问时间 与 FIFO 的唯一区别在于此罢了
82.                 cout << "\t";
83.                 break; //命中
84.             }
85.             else if(mem[i].first == -1){
86.                 mem[i].first = x;
87.                 mem[i].second = 0;
88.                 flag = false;
89.                 diseffect++;
90.                 cout << "缺页!\t";
91.                 break; //有空余 也是缺页
92.             }
93.         }
94.         if(flag){
95.             diseffect++;
96.             int t = 0, ma=0;
97.             // 找出时间最久的
98.             for(int i = 0; i < MAX_FRAME; i++){
99.                 if(mem[i].second > ma) t = i, ma = mem[i].second;
100.            }
101.            //置换掉
102.            cout << "缺页!\t";
103.            mem[t].first = x;

```



```

104.         mem[t].second = 0;
105.     }
106.     // 进入的时间加一
107.     for(int i = 0; i < MAX_FRAME; i++){
108.         if(mem[i].first != -1) mem[i].second ++;
109.     }
110.     //输出状态
111.     cout << x << "\t\t";
112.     for(int i = 0; i < MAX_FRAME; i++){
113.         cout << mem[i].first << "\t";
114.     }
115.     puts("");
116. }
117.     cout << "缺页共" << diseffect << "个, 总页面" << N << "个, 缺页率:
    " << (double) diseffect/N<<endl;
118.     return ;
119. }
120.
121. void CLOCK(){
122.     cout<<"CLOCK!:\n"<<"访问页面\t"<<"\t 当前内存"<<endl;
123.     int diseffect = 0; //缺页数
124.     int idx = 0; //替换指针
125.     for(int j = 0; j < N; j++){
126.         int x = Acess_Series[j]; //要使用的页面
127.         bool flag = true;
128.         for(int i = 0; i < MAX_FRAME; i++){
129.             if(x == mem[i].first ){
130.                 flag = false;
131.                 mem[i].second = 1; //命中 访问位为1
132.                 cout<<"\t";
133.                 break; //命中
134.             }
135.             else if(mem[i].first == -1){
136.                 mem[i].first = x;
137.                 mem[i].second = 1;
138.                 flag = false;
139.                 diseffect++;
140.                 cout << "缺页!\t";
141.                 break; //有空余 也是缺页
142.             }
143.         }
144.         if(flag){
145.             diseffect++;
146.             while(mem[idx].second != 0){

```

```

147.             mem[idx].second = 0;
148.             idx = (idx + 1) % MAX_FRAME;
149.         }
150.         mem[idx].first = x;
151.         mem[idx].second = 1;
152.         idx = (idx + 1) % MAX_FRAME;
153.
154.         //置换掉
155.         cout << "缺页!\t";
156.     }
157.     //输出状态
158.     cout << x << "\t\t";
159.     for(int i = 0; i < MAX_FRAME; i++){
160.         cout << mem[i].first << "\t";
161.     }
162.     puts("");
163. }
164.     cout << "缺页共" << diseffect << "个, 总页面" << N << "个, 缺页率:
    " << (double) diseffect/N<<endl;
165.     return ;
166. }
167.
168. void CLOCK_PLUS(){
169.     cout<<"改进型 CLOCK!:\n"<<"访问页面\t"<<"\t 当前内存"<<endl;
170.     pair<int,PII> Mem[MAX_FRAME];// mem.first 为序号 mem.second.first 为访问位
    A mem.second.second 为修改位 M
171.     memset(Mem, -1, sizeof Mem);
172.     int diseffect = 0;//缺页数
173.     int idx = 0; //替换指针
174.     for(int j = 0; j < N; j++){
175.         //随机进行修改 设置修改位
176.         if(rand() % 2 == 1 ){
177.             Mem[rand() % MAX_FRAME].second.second = 1;
178.         }
179.
180.
181.         int x = Acess_Series[j]; //要使用的页面
182.         bool flag = true;
183.         for(int i = 0; i < MAX_FRAME; i++){
184.             if(x == Mem[i].first ){
185.                 flag = false;
186.                 Mem[i].second.first = 1;//命中 访问位为 1
187.                 cout<<"\t";
188.                 break; //命中

```

```

189.         }
190.         else if(Mem[i].first == -1){
191.             Mem[i].first = x;
192.             Mem[i].second.first = 1;
193.             Mem[i].second.second = 0;
194.             flag = false;
195.             diseffect++;
196.             cout <<"缺页!\t";
197.             break; //有空余 也是缺页
198.         }
199.     }
200.     if(flag){
201.         diseffect++;
202.         //第一遍循环
203.         for(int i = 0; i < MAX_FRAME; i++){
204.             if(Mem[idx].second.first == 0 && Mem[idx].second.second == 0)
205.                 goto label;
206.             idx = (idx + 1) % MAX_FRAME;
207.         }
208.         //第二遍循环
209.         for(int i = 0; i < MAX_FRAME; i++){
210.             if(Mem[idx].second.first == 0 && Mem[idx].second.second == 1)
211.                 goto label;
212.             Mem[idx].second.first = 0; //将访问位置为0
213.             idx = (idx + 1) % MAX_FRAME;
214.         }
215.         //第三遍循环
216.         for(int i = 0; i < MAX_FRAME; i++){
217.             if(Mem[idx].second.first == 0 && Mem[idx].second.second == 0)
218.                 goto label;
219.             idx = (idx + 1) % MAX_FRAME;
220.         }
221.         //第四遍循环
222.         for(int i = 0; i < MAX_FRAME; i++){
223.             if(Mem[idx].second.first == 0 && Mem[idx].second.second == 1)
224.                 goto label;
225.             Mem[idx].second.first = 0; //将访问位置为0
226.             idx = (idx + 1) % MAX_FRAME;
227.         }
228.         label:
229.         Mem[idx].first = x;
230.         Mem[idx].second.first = 1;
231.         idx = (idx + 1) % MAX_FRAME;

```

```

229.
230.         cout <<"缺页!\t";
231.     }
232.     //输出状态
233.     cout << x <<"\t\t";
234.     for(int i = 0; i < MAX_FRAME; i++){
235.         cout << Mem[i].first << "\t";
236.     }
237.     puts("");
238. }
239.     cout <<"缺页共" << diseffect <<"个, 总页面" << N <<"个, 缺页率:
    " << (double) diseffect/N<<endl;
240.     return ;
241. }
242.
243. int main(){
244.     srand((unsigned int)(time(NULL)));
245.     // 随机生成 20 个 [1,K] 之间的数 放入 ACESS_Series 访问序列数组。
246.     for(int i = 0; i < N; i++){
247.         ACESS_Series[i] = rand() % K + 1;
248.     }
249.     cout << "页面序列: ";
250.     for(int i = 0; i < N; i++){
251.         cout << ACESS_Series[i] <<" ";
252.     }
253.     puts("");
254.     //初始化 first 为编号 second 为记录的值。
255.     for(int i = 0; i < MAX_FRAME; i++){
256.         mem[i].first = -1;
257.         mem[i].second = 0;
258.     }
259.
260.     pid_t pid;
261.     bool FLAG[4];
262.     for(int i = 0; i < 4; i++){
263.         pid = fork();
264.         if(pid == -1){
265.             fprintf(stderr, "创建进程失败! ");
266.             return -1;
267.         }
268.         else if(pid > 0){
269.             FLAG[i] = true;
270.             wait(NULL);
271.         }

```

```
272.         else{
273.             if(!FLAG[0]) {
274.                 FIFO();
275.             }
276.             else if(!FLAG[1]) {
277.                 LRU();
278.             }
279.             else if(!FLAG[2]) {
280.                 CLOCK();
281.             }
282.             else {
283.                 CLOCK_PLUS();
284.             }
285.             exit(0);
286.         }
287.     }
288.     return 0;
289. }
290.
```

注意，此位置分页内容要确保到下页，避免隐藏到表格下导致内容丢失。