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	计算机科学与技术	班级	计 XXXX 班
学号	XXXXXXXX	姓名	XX

实验项目目录

实验一

一．实验目的

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。

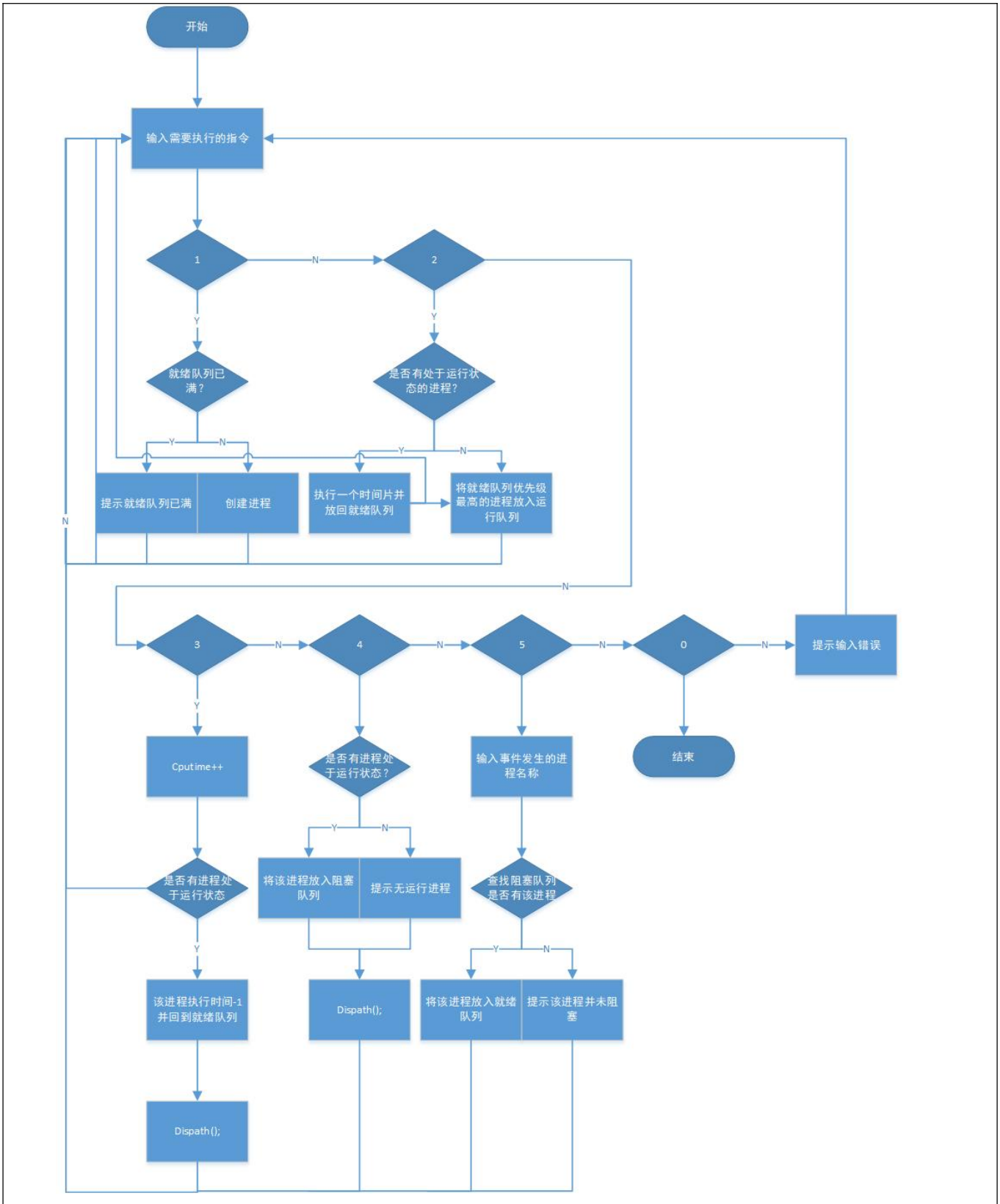
二．实验内容

1. 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。
2. 独立设计、编写、调试程序。
3. 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。
4. 进程的状态模型（三状态、五状态、七状态或其它）可自行选择。

三．实验要求

1. 代码书写要规范，要适当地加入注释。
2. 鼓励在实验中加入新的观点或想法，并加以实现。
3. 认真进行预习，完成预习报告。
4. 实验完成后，要认真总结，完成实验报告。

四．程序流程图



五．程序描述

使用 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，具备一定的运行内存限制（100M），若处理进程内存超过系统内存限制则会先被放入等待队列中。完成了界面功能，可以显示实时的不同队列中的进程的 id。

自定义了结构体 pcb，组织方式采用的是链表。结构体 pcb 用来存储一个进程的 id 和指向下一个节点的指针 next。同一状态的进程其 PCB 成一队列，多个状态对应多个不同的队列；

所以这里形成了不同的队列：新建队列、就绪队列、运行队列、阻塞队列。可以实现初始化新建队列、就绪队列、运行队列、阻塞队列以及下述七个操作。

0:exit

1:Dispatch(ReadyQueue->RunningQueue)

2:Timeout(RunningQueue->ReadyQueue)

3:Event Wait(RunningQueue->BlockedQueue)

4:Event Occurs(BlockedQueue->ReadyQueue)

5:Admit(NewQueue->ReadyQueue)

6:Release(RunningQueue->Exit)

定义了 4 个队列，分别用来储存 Ready,Running,Block,New 队列。同时考虑到了非法的输入，对于不同的非法输入都有处理。同时设定了不同操作之间的在不同情况下的连带关系，例如结束进程后会自动把新的就绪态进程派遣到运行队列中等等。

主要函数的解释：

// 创建带头结点的 ready 队列,尾插法

pcb *create()

头结点为 head，q 指向尾结点，p 表示新生成的结点。使用的是尾插法

// 插入结点

void insert(pcb *head, pcb *node)

在尾部插入新的节点

// 删除结点

void del(pcb *head)

删除第一个节点

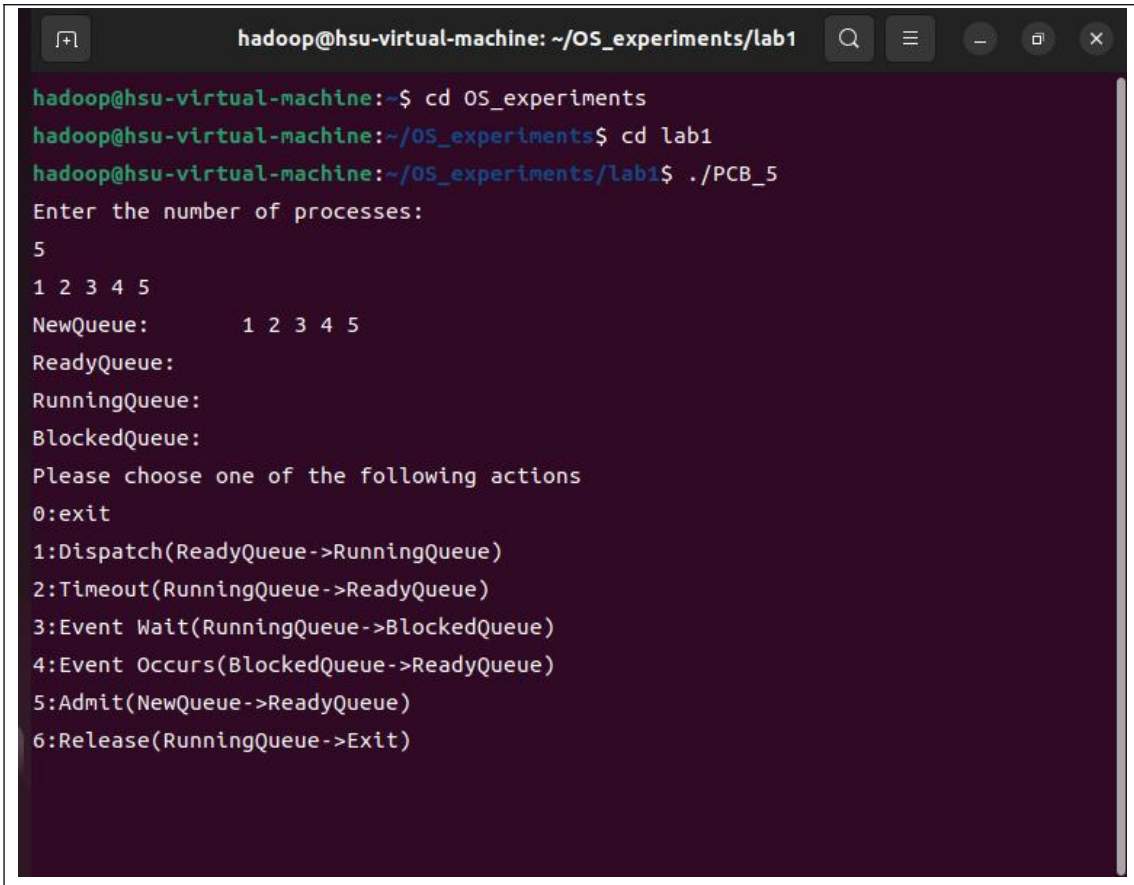
// 展示各个状态的进程

void show()

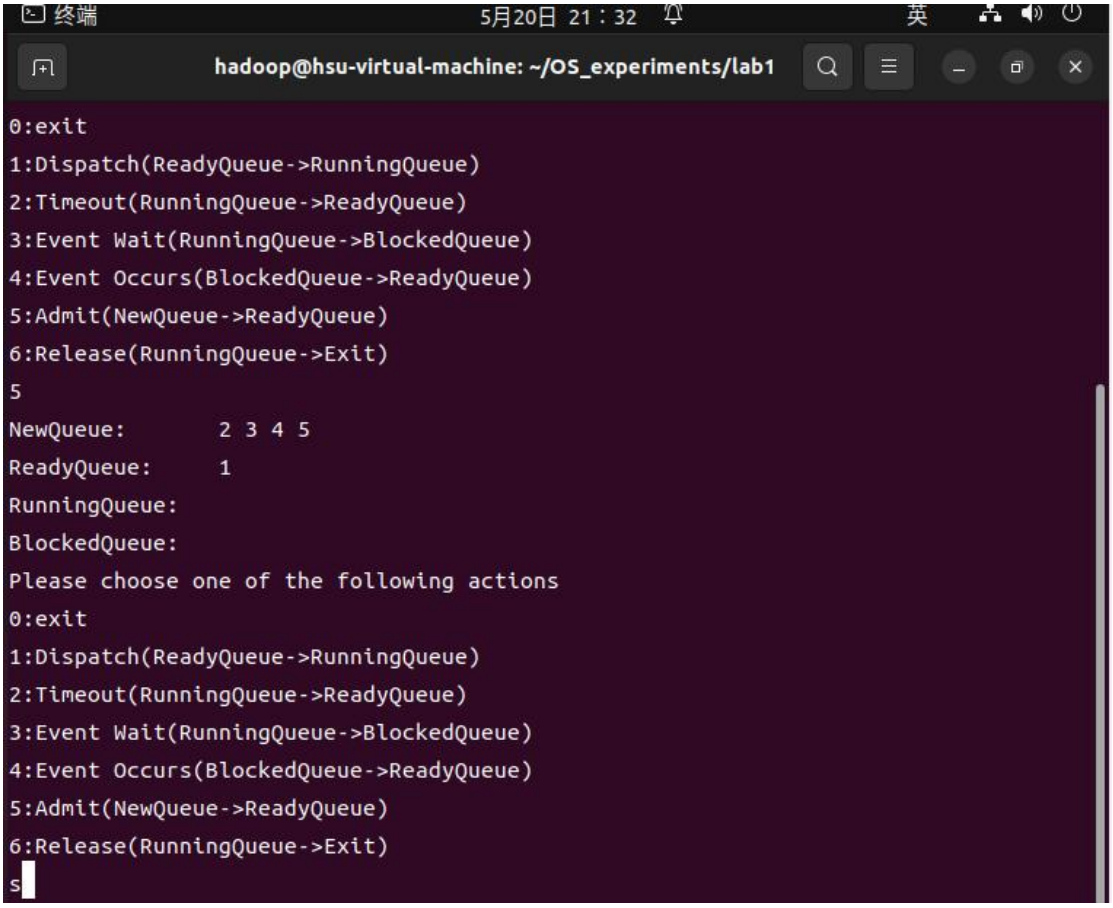
遍历 ready 和 block 队列，依次输出各个结点的 id，输出 run 结点的 id

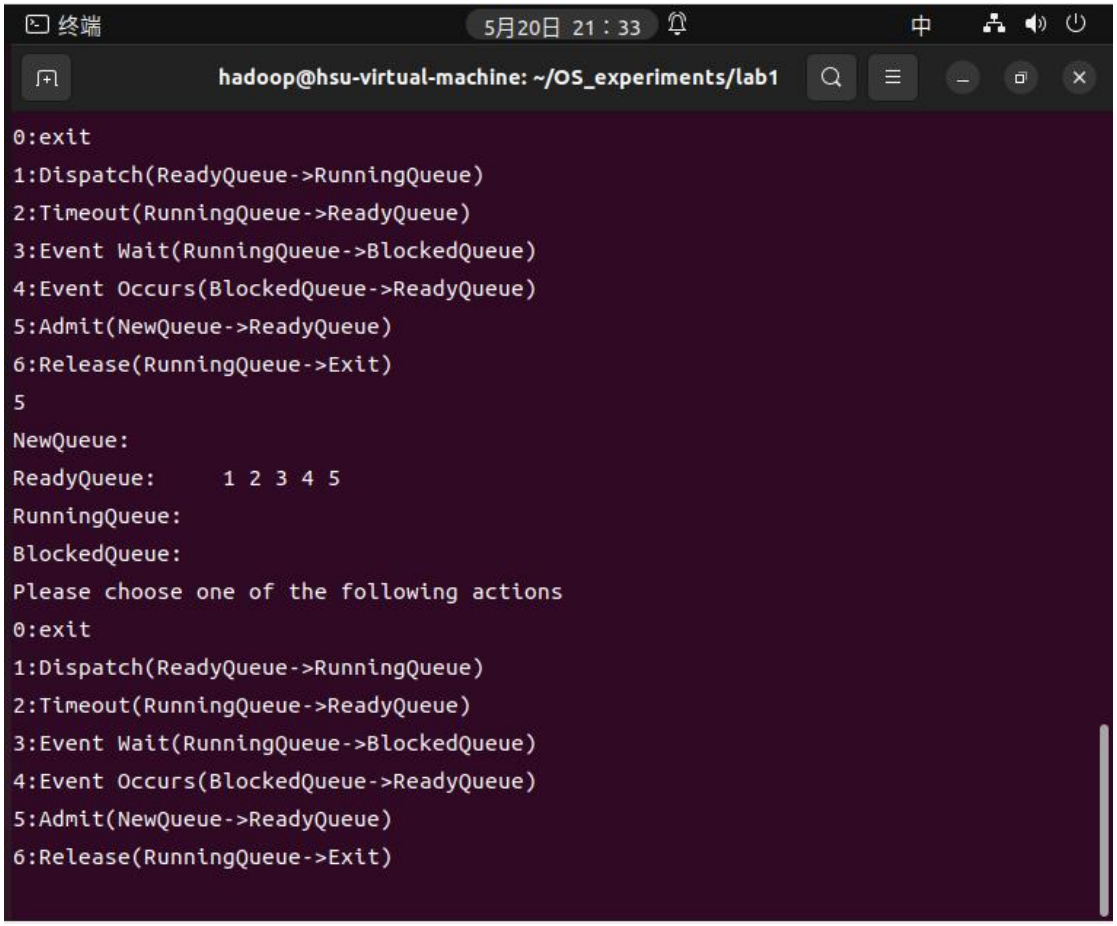
六. 实验结果（截图）

创建进程

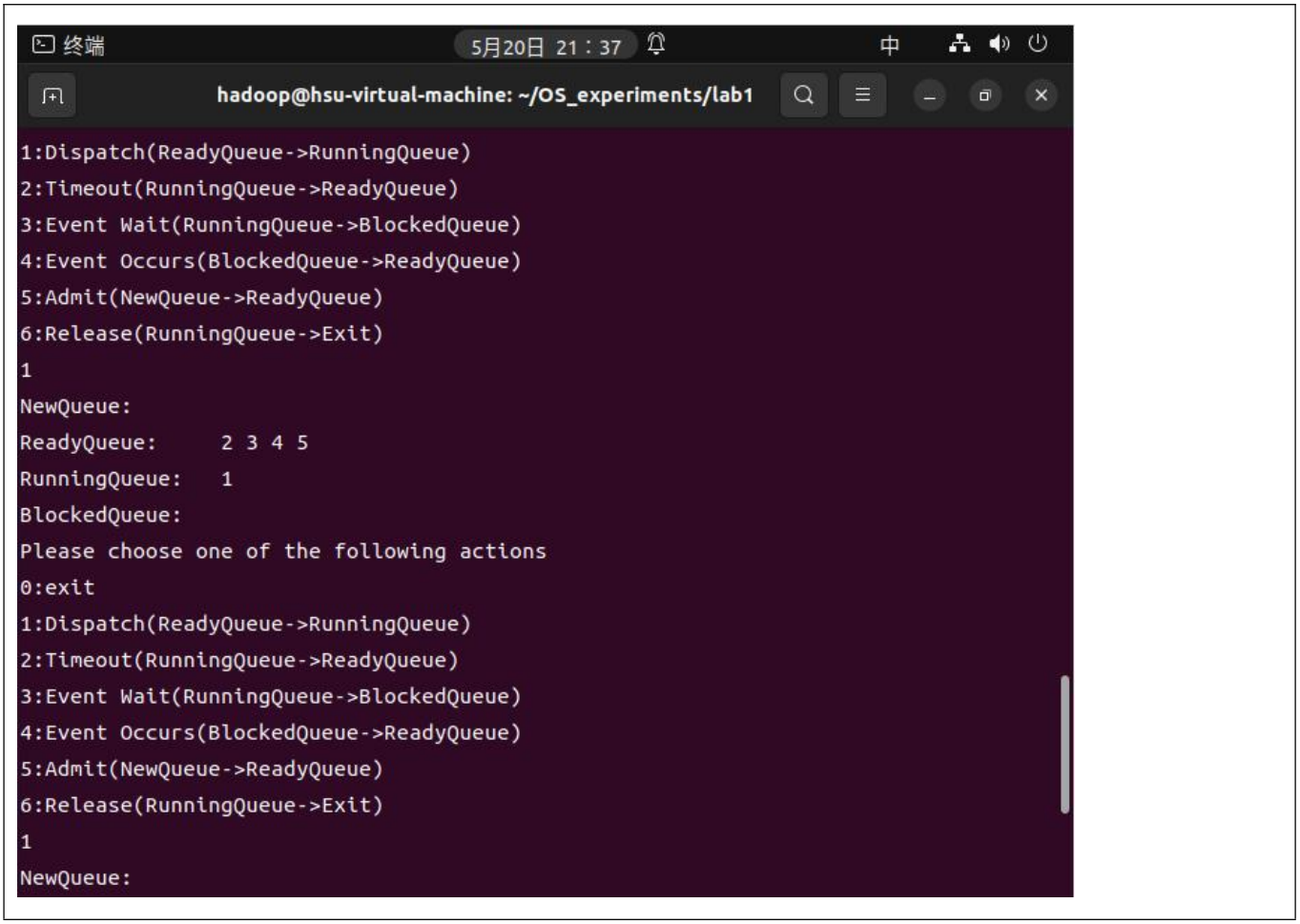


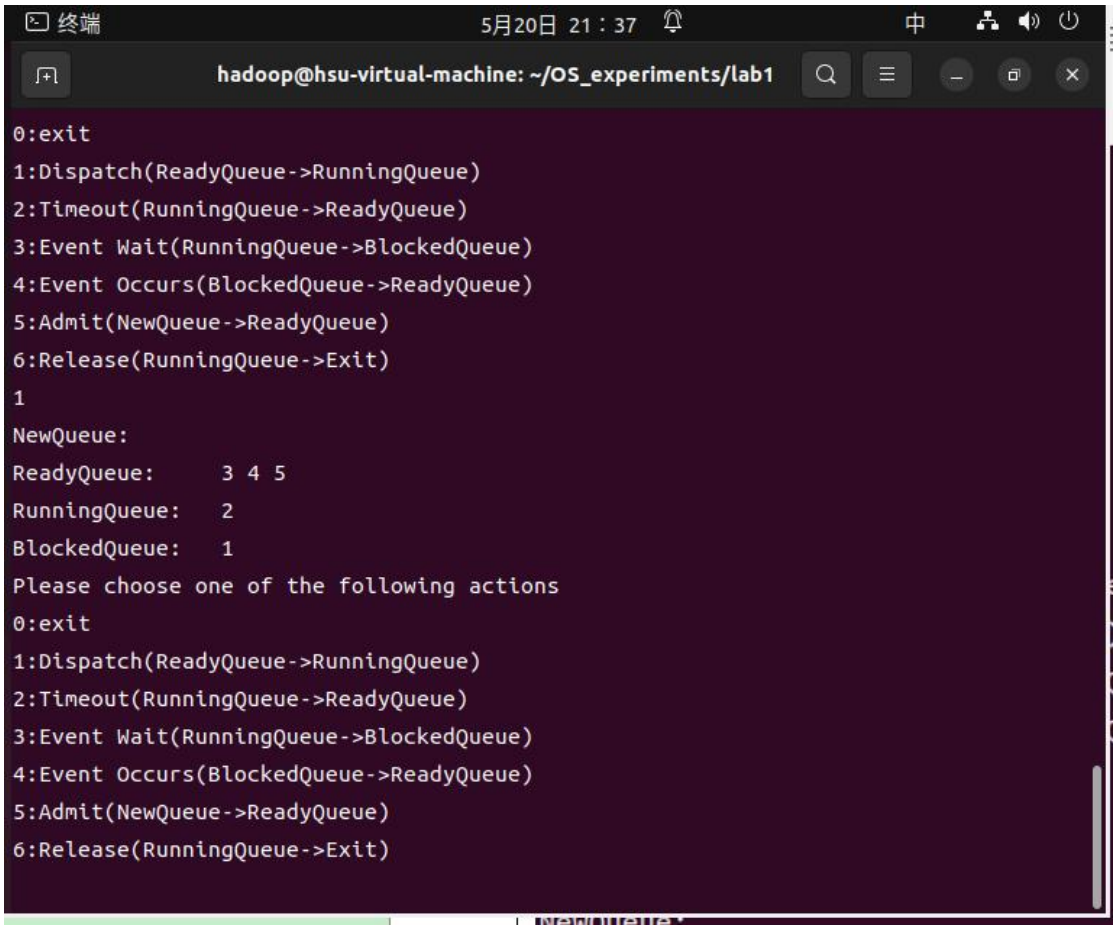
5:Admit(NewQueue->ReadyQueue)进入就绪队列。其他操作类似，都是实现进程队列的切换。



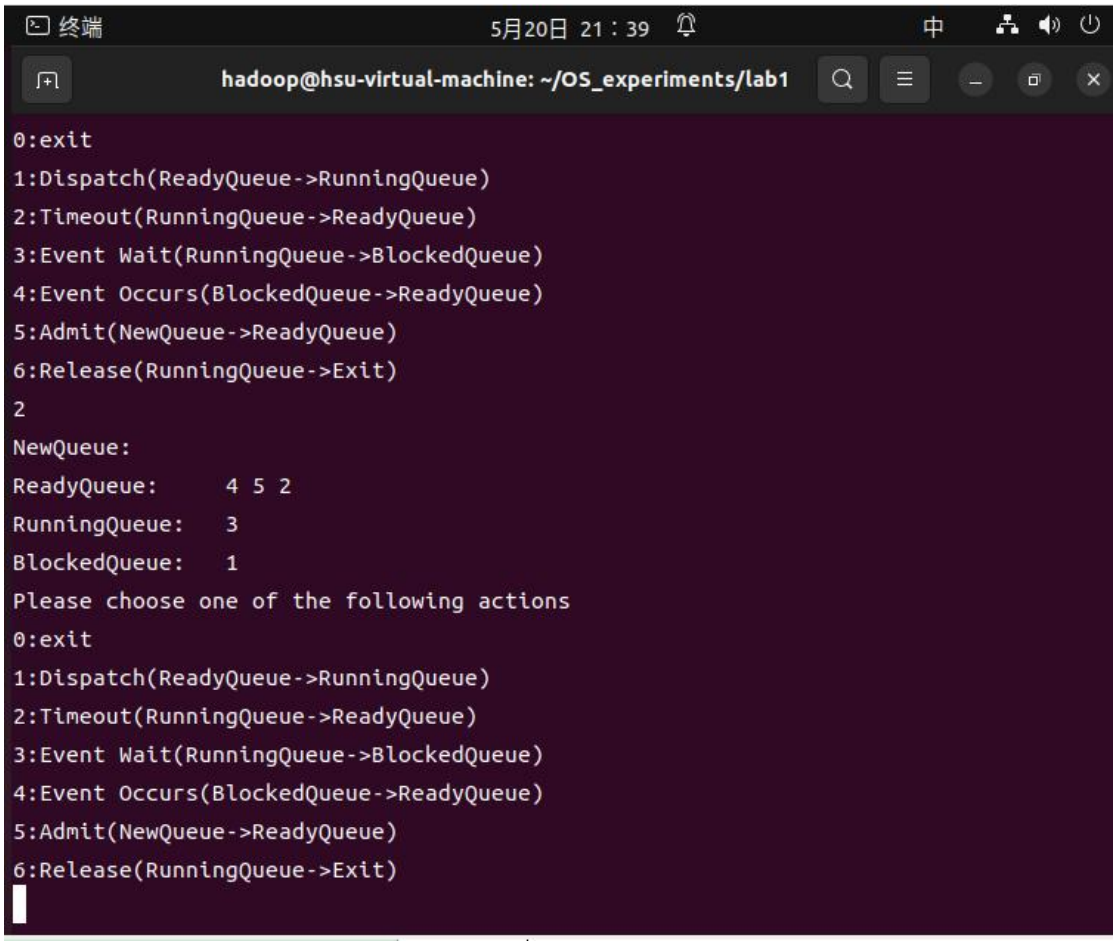


1:Dispatch(ReadyQueue->RunningQueue)就绪队列队头结点进入运行队列，如果此时运行队列有进程，则该进程被剥夺，属于非主动释放，因此自动放入阻塞队列里面。

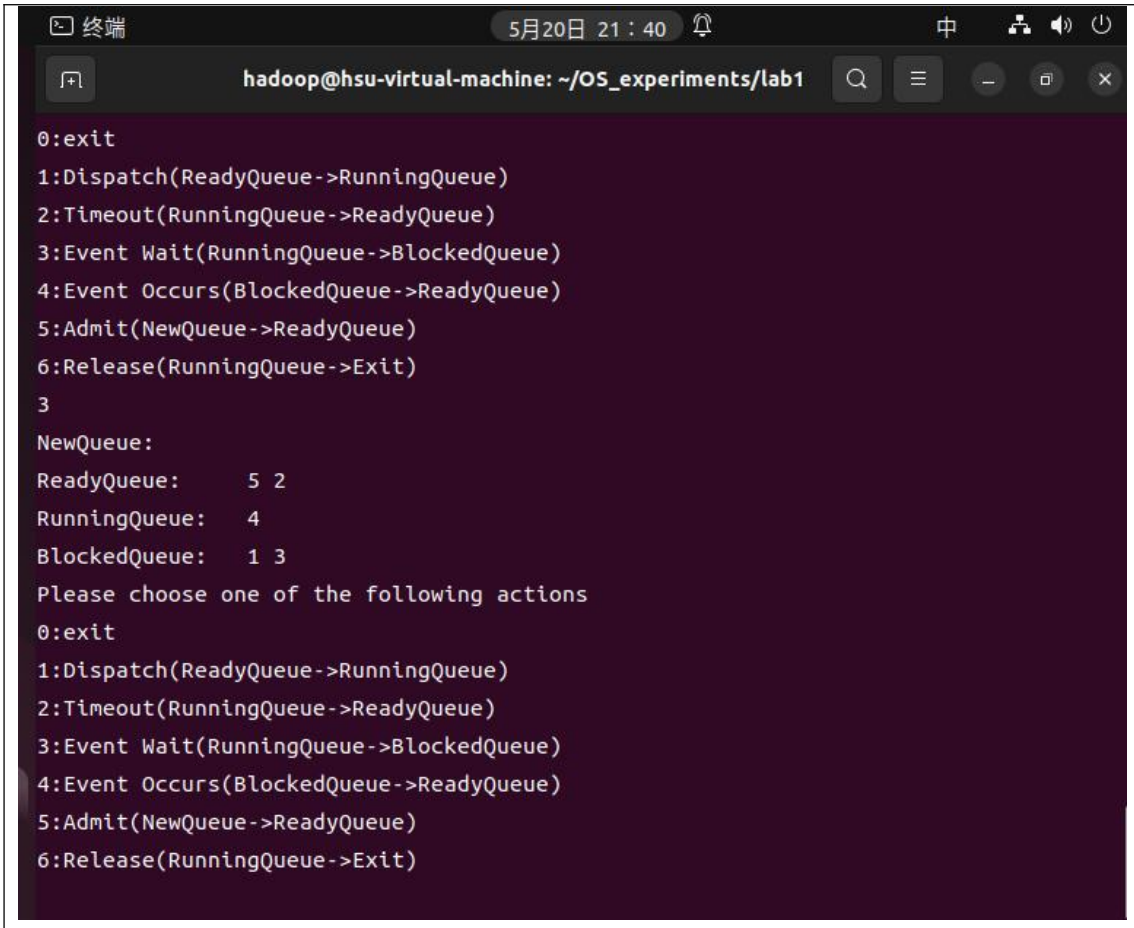

终端
5月20日 21：37
中
hadoop@hsu-virtual-machine: ~/OS_experiments/lab1
1:Dispatch(ReadyQueue->RunningQueue)
2:Timeout(RunningQueue->ReadyQueue)
3:Event Wait(RunningQueue->BlockedQueue)
4:Event Occurs(BlockedQueue->ReadyQueue)
5:Admit(NewQueue->ReadyQueue)
6:Release(RunningQueue->Exit)
1
NewQueue:
ReadyQueue: 2 3 4 5
RunningQueue: 1
BlockedQueue:
Please choose one of the following actions
0:exit
1:Dispatch(ReadyQueue->RunningQueue)
2:Timeout(RunningQueue->ReadyQueue)
3:Event Wait(RunningQueue->BlockedQueue)
4:Event Occurs(BlockedQueue->ReadyQueue)
5:Admit(NewQueue->ReadyQueue)
6:Release(RunningQueue->Exit)
1
NewQueue:



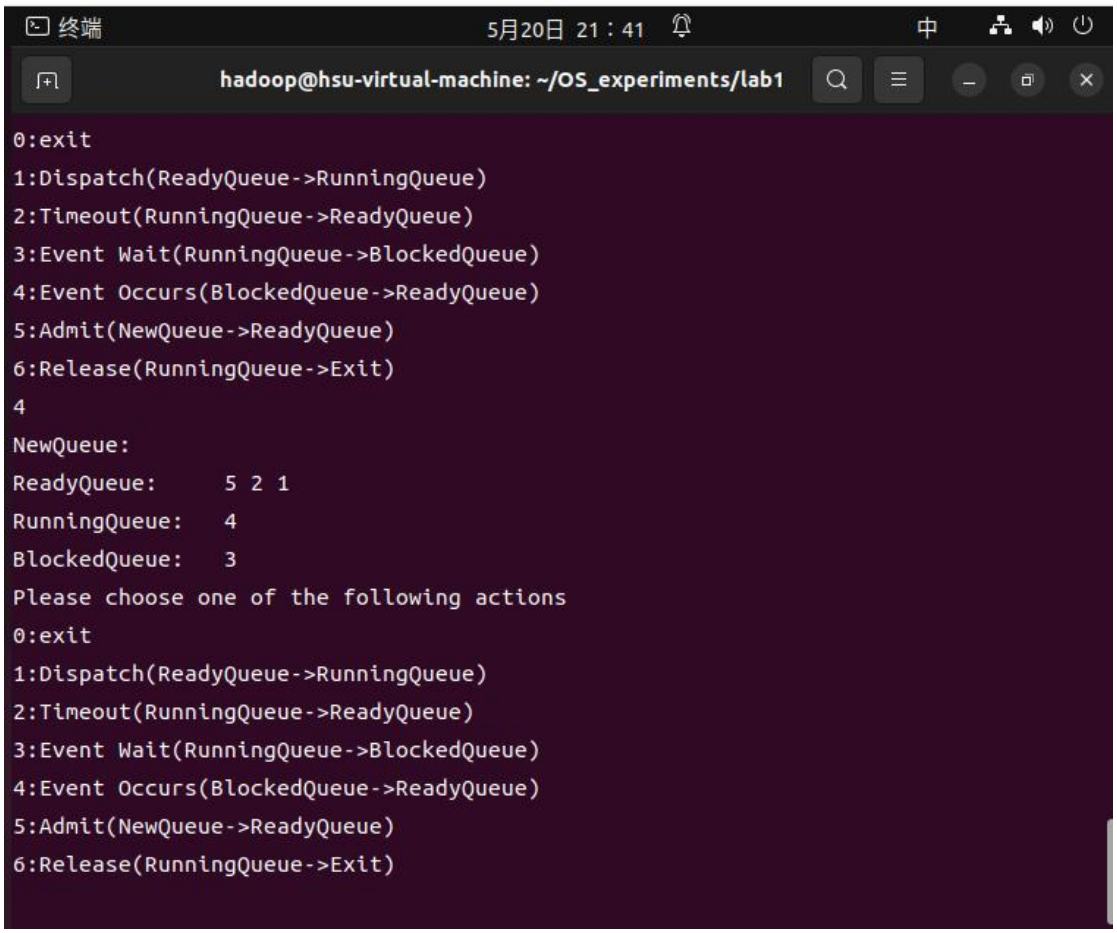
2:Timeout(RunningQueue->ReadyQueue)运行队列队头结点进入就绪队列。



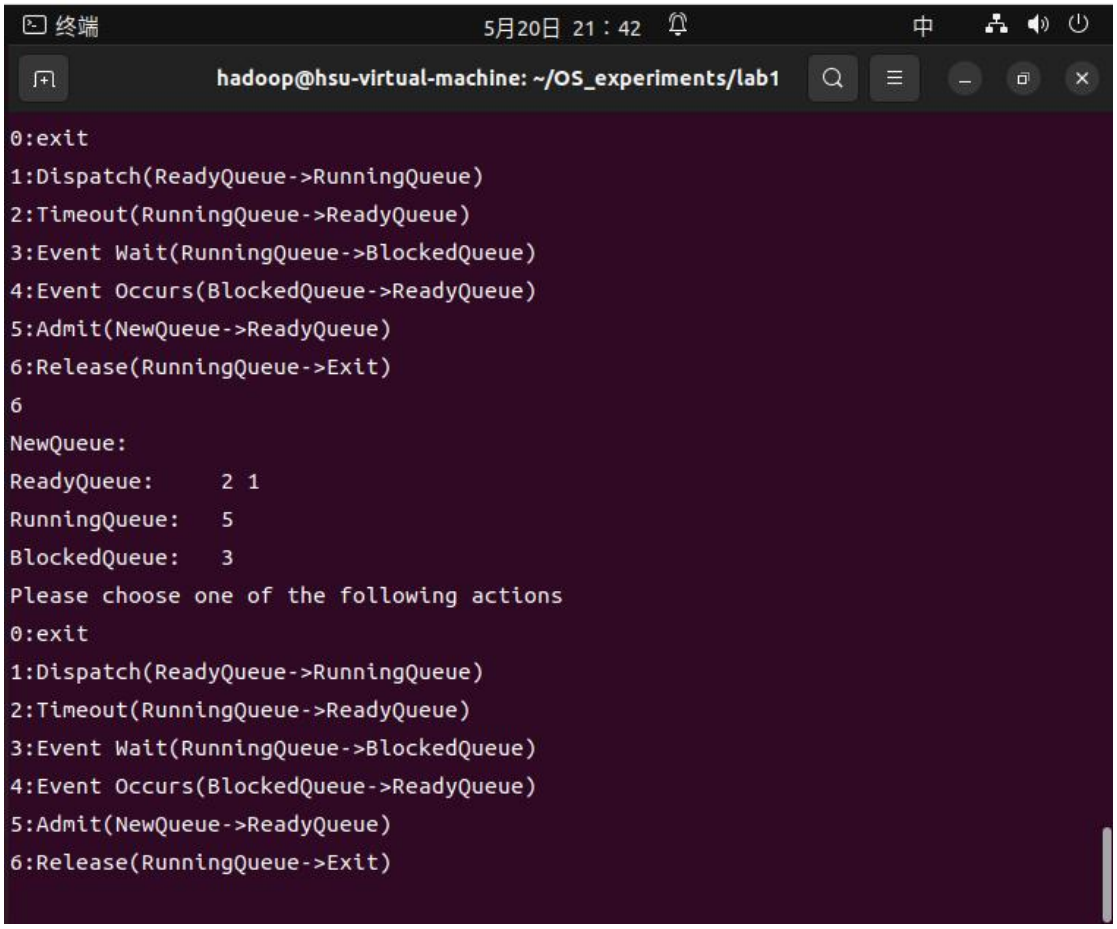
3:Event Wait(RunningQueue->BlockedQueue) 运行队列队头结点进入阻塞队列。



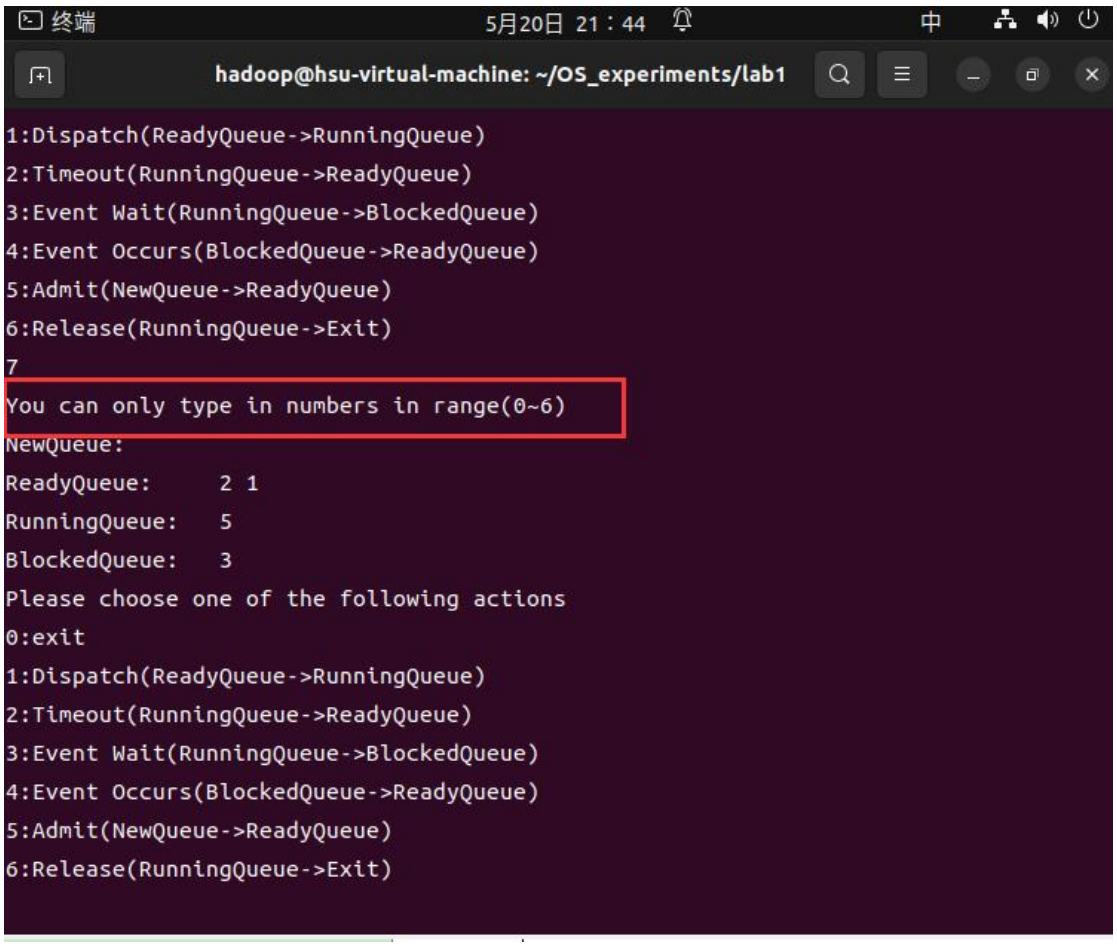
4:Event Occurs(BlockedQueue->ReadyQueue) 阻塞队列队头结点进入就绪队列。



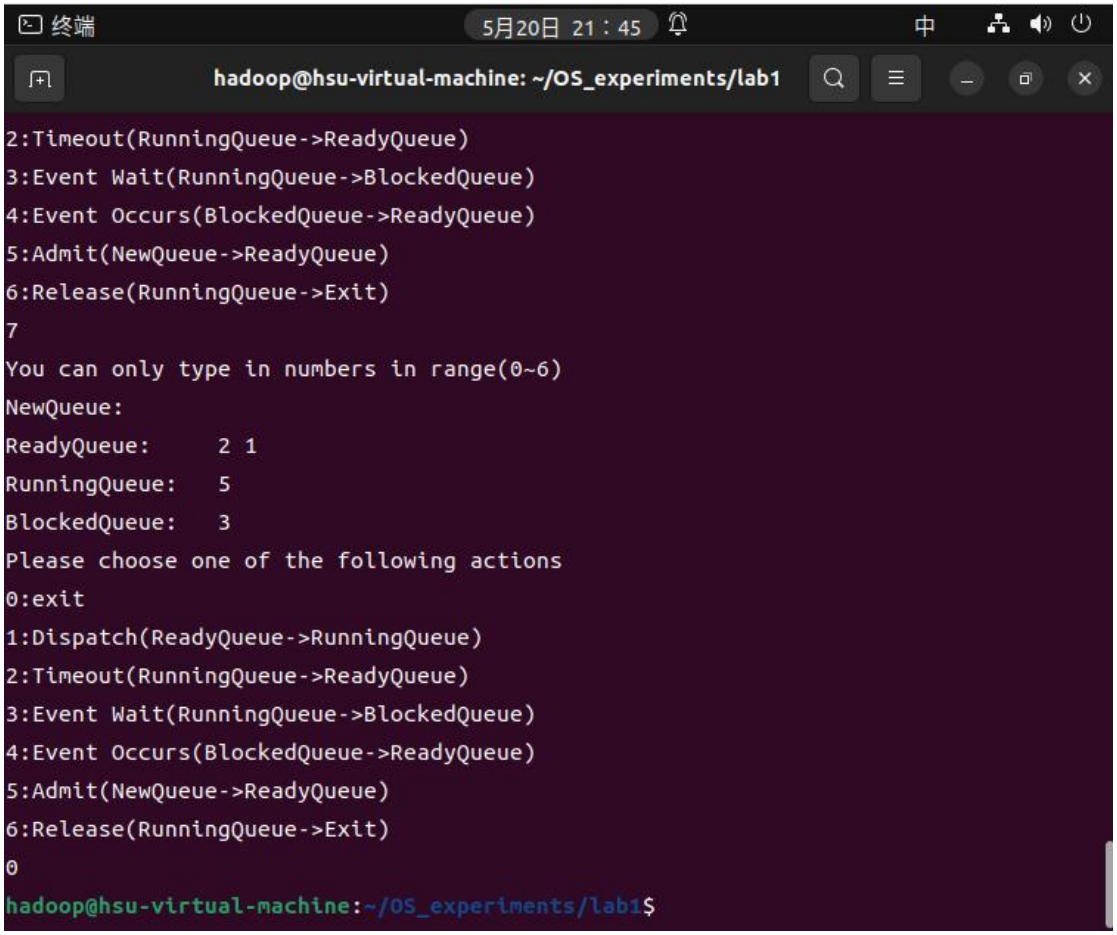
6:Release(RunningQueue->Exit) 运行队列队头结点进入结束态。此时若就绪队列不为空，则自动将就绪队列队头结点放入运行队列。


终端
5月20日 21：42
中
hadoop@hsu-virtual-machine: ~/OS_experiments/lab1
0:exit
1:Dispatch(ReadyQueue->RunningQueue)
2:Timeout(RunningQueue->ReadyQueue)
3:Event Wait(RunningQueue->BlockedQueue)
4:Event Occurs(BlockedQueue->ReadyQueue)
5:Admit(NewQueue->ReadyQueue)
6:Release(RunningQueue->Exit)
6
NewQueue:
ReadyQueue: 2 1
RunningQueue: 5
BlockedQueue: 3
Please choose one of the following actions
0:exit
1:Dispatch(ReadyQueue->RunningQueue)
2:Timeout(RunningQueue->ReadyQueue)
3:Event Wait(RunningQueue->BlockedQueue)
4:Event Occurs(BlockedQueue->ReadyQueue)
5:Admit(NewQueue->ReadyQueue)
6:Release(RunningQueue->Exit)

异常处理


终端
5月20日 21：44
中
hadoop@hsu-virtual-machine: ~/OS_experiments/lab1
1:Dispatch(ReadyQueue->RunningQueue)
2:Timeout(RunningQueue->ReadyQueue)
3:Event Wait(RunningQueue->BlockedQueue)
4:Event Occurs(BlockedQueue->ReadyQueue)
5:Admit(NewQueue->ReadyQueue)
6:Release(RunningQueue->Exit)
7
You can only type in numbers in range(0~6)
NewQueue:
ReadyQueue: 2 1
RunningQueue: 5
BlockedQueue: 3
Please choose one of the following actions
0:exit
1:Dispatch(ReadyQueue->RunningQueue)
2:Timeout(RunningQueue->ReadyQueue)
3:Event Wait(RunningQueue->BlockedQueue)
4:Event Occurs(BlockedQueue->ReadyQueue)
5:Admit(NewQueue->ReadyQueue)
6:Release(RunningQueue->Exit)

0:exit 退出程序



七．源代码

```
// #include  <stdio.h>
// #include  <sys/types.h>
// #include  <stdlib.h>
// #include  <sys/stat.h>
// #include <fcntl.h>
// #include  <error.h>
// #include  <wait.h>
// #include  <unistd.h>

#include <stdio.h>
#include <stdlib.h>

#include <iostream>
using namespace std;

typedef int ElemType;
typedef struct pcb
{
    ElemType id;
```

```
    struct pcb *next;
} pcb;

pcb *NewQueue, *ReadyQueue, *RunningQueue, *BlockedQueue, *ExitQueue;

// 创建带头结点的ReadyQueue 队列,尾插法
pcb *create()
{
    pcb *head = new pcb;
    head->next = NULL;
    pcb *q = head;
    int n;
    cout << "Enter the number of processes:\n";
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        pcb *p = new pcb;
        cin >> p->id;
        p->next = NULL;
        q->next = p;
        q = p;
    }
    return head;
}

// 插入结点
// 在尾部插入新的节点，就是从队列尾部入队的操作
void insert(pcb *head, pcb *node)
{
    pcb *p = head;
    while (p->next)
    {
        p = p->next;
    }
    pcb *n = new pcb;
    n->id = node->id;
    p->next = n;
    n->next = NULL;
}

// 删除结点
// 删除头结点后面的第一个结点，就是从队列头部出队的操作
void del(pcb *head)
{
```

```
    pcb *p = head->next;
    if (p)
    {
        head->next = head->next->next;
        delete p;
    }
}

// 展示各个状态的进程
void show()
{
    pcb *p;

    p = NewQueue->next;
    cout << "NewQueue:\t";
    while (p)
    {
        cout << p->id << " ";
        p = p->next;
    }
    cout << endl;

    p = ReadyQueue->next;
    cout << "ReadyQueue:\t";
    while (p)
    {
        cout << p->id << " ";
        p = p->next;
    }
    cout << endl;

    p = RunningQueue->next;
    cout << "RunningQueue:\t";
    while (p)
    {
        cout << p->id << " ";
        p = p->next;
    }
    cout << endl;

    p = BlockedQueue->next;
    cout << "BlockedQueue:\t";
    while (p)
    {
```

```
        cout << p->id << " ";
        p = p->next;
    }
    cout << endl;

    // p = ExitQueue->next;
    // cout << "ExitQueue:\t";
    // while (p)
    // {
    //     cout << p->id << " ";
    //     p = p->next;
    // }
    // cout << endl;
}

int main()
{
    NewQueue = create();
    ReadyQueue = new pcb;
    ReadyQueue->next = NULL;
    RunningQueue = new pcb;
    RunningQueue->next = NULL;
    BlockedQueue = new pcb;
    BlockedQueue->next = NULL;
    ExitQueue = new pcb;
    ExitQueue->next = NULL;
    //RunningQueue 和BlockedQueue 没有对第一个结点的id 赋值，在insert（）之后，属于带头结点的队列，保持和ReadyQueue 一样的结构
    show();
    int ChangeNum;
    cout << "Please choose one of the following actions" << endl;
    cout << "0:exit" << endl;
    cout << "1:Dispatch(ReadyQueue->RunningQueue)" << endl;
    cout << "2:Timeout(RunningQueue->ReadyQueue)" << endl;
    cout << "3:Event Wait(RunningQueue->BlockedQueue)" << endl;
    cout << "4:Event Occurs(BlockedQueue->ReadyQueue)" << endl;
    cout << "5:Admit(NewQueue->ReadyQueue)" << endl;
    cout << "6:Release(RunningQueue->Exit)" << endl;

    while (cin >> ChangeNum)
    {
        if (ChangeNum == 0)
        {
            break;
```

```
        }
        else if (ChangeNum == 1)
        {
            if (ReadyQueue->next)
            {
                if (RunningQueue->next)
                {
                    insert(BlockedQueue, RunningQueue->next);
                    del(RunningQueue);
                }
                insert(RunningQueue, ReadyQueue->next);
                del(ReadyQueue);
            }
            else
            {
                if (BlockedQueue->next)
                {
                    insert(BlockedQueue, RunningQueue->next);
                    del(RunningQueue);
                    insert(RunningQueue, BlockedQueue->next);
                    del(BlockedQueue);
                }
            }
        }
        else if (ChangeNum == 2)
        {
            if (RunningQueue->next)
            {
                if (ReadyQueue->next)
                {
                    insert(ReadyQueue, RunningQueue->next);
                    del(RunningQueue);
                    insert(RunningQueue, ReadyQueue->next);
                    del(ReadyQueue);
                }
            }
        }
        else if (ChangeNum == 3)
        {
            if (RunningQueue->next)
            {
                if (ReadyQueue->next)
                {
                    insert(BlockedQueue, RunningQueue->next);
```

```
                del(RunningQueue);
                insert(RunningQueue, ReadyQueue->next);
                del(ReadyQueue);
            }
            else
            {
                insert(BlockedQueue, RunningQueue->next);
                del(RunningQueue);
                insert(RunningQueue, BlockedQueue->next);
                del(BlockedQueue);
            }
        }
    }
    else if (ChangeNum == 4)
    {
        if (BlockedQueue->next)
        {
            insert(ReadyQueue, BlockedQueue->next);
            del(BlockedQueue);
        }
    }
    else if (ChangeNum == 5)
    {
        if (NewQueue->next)
        {
            insert(ReadyQueue, NewQueue->next);
            del(NewQueue);
        }
    }
    else if (ChangeNum == 6)
    {
        if (RunningQueue->next)
        {
            insert(ExitQueue, RunningQueue->next);
            del(RunningQueue);
            // del(ExitQueue);释放处于结束状态的进程结点
            // 也可以直接释放RunnningQueue 的结点
            if (ReadyQueue->next)
            {
                insert(RunningQueue, ReadyQueue->next);
                del(ReadyQueue);
            }
        }
    }
```

```
        else
        {
            cout << "You can only type in numbers in range(0~6)" << endl;
        }
        show();
        cout << "Please choose one of the following actions" << endl;
        cout << "0:exit" << endl;
        cout << "1:Dispatch(ReadyQueue->RunningQueue)" << endl;
        cout << "2:Timeout(RunningQueue->ReadyQueue)" << endl;
        cout << "3:Event Wait(RunningQueue->BlockedQueue)" << endl;
        cout << "4:Event Occurs(BlockedQueue->ReadyQueue)" << endl;
        cout << "5:Admit(NewQueue->ReadyQueue)" << endl;
        cout << "6:Release(RunningQueue->Exit)" << endl;
    }

    return 0;
}
```

实验二

一．实验目的

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。

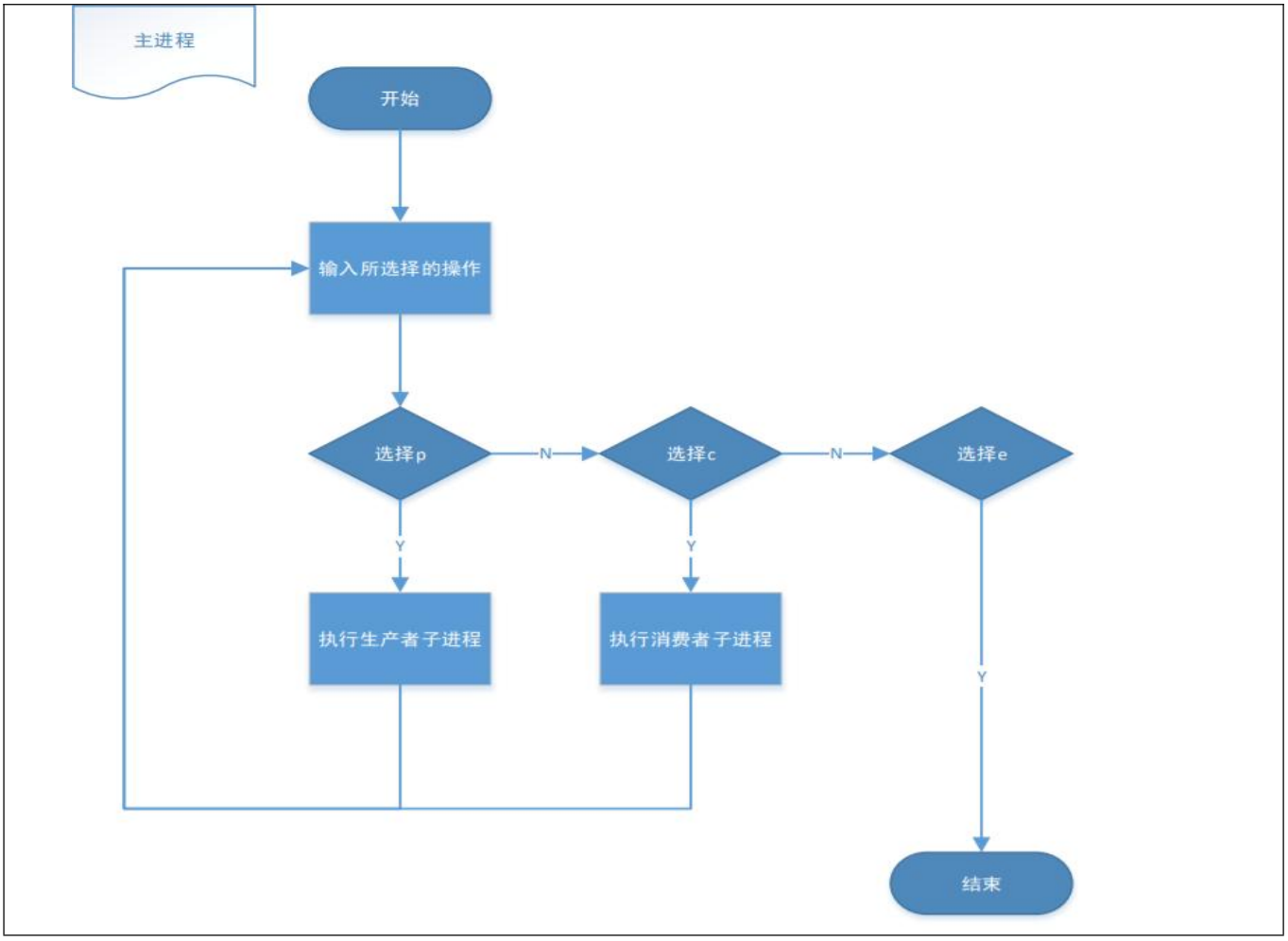
二．实验内容

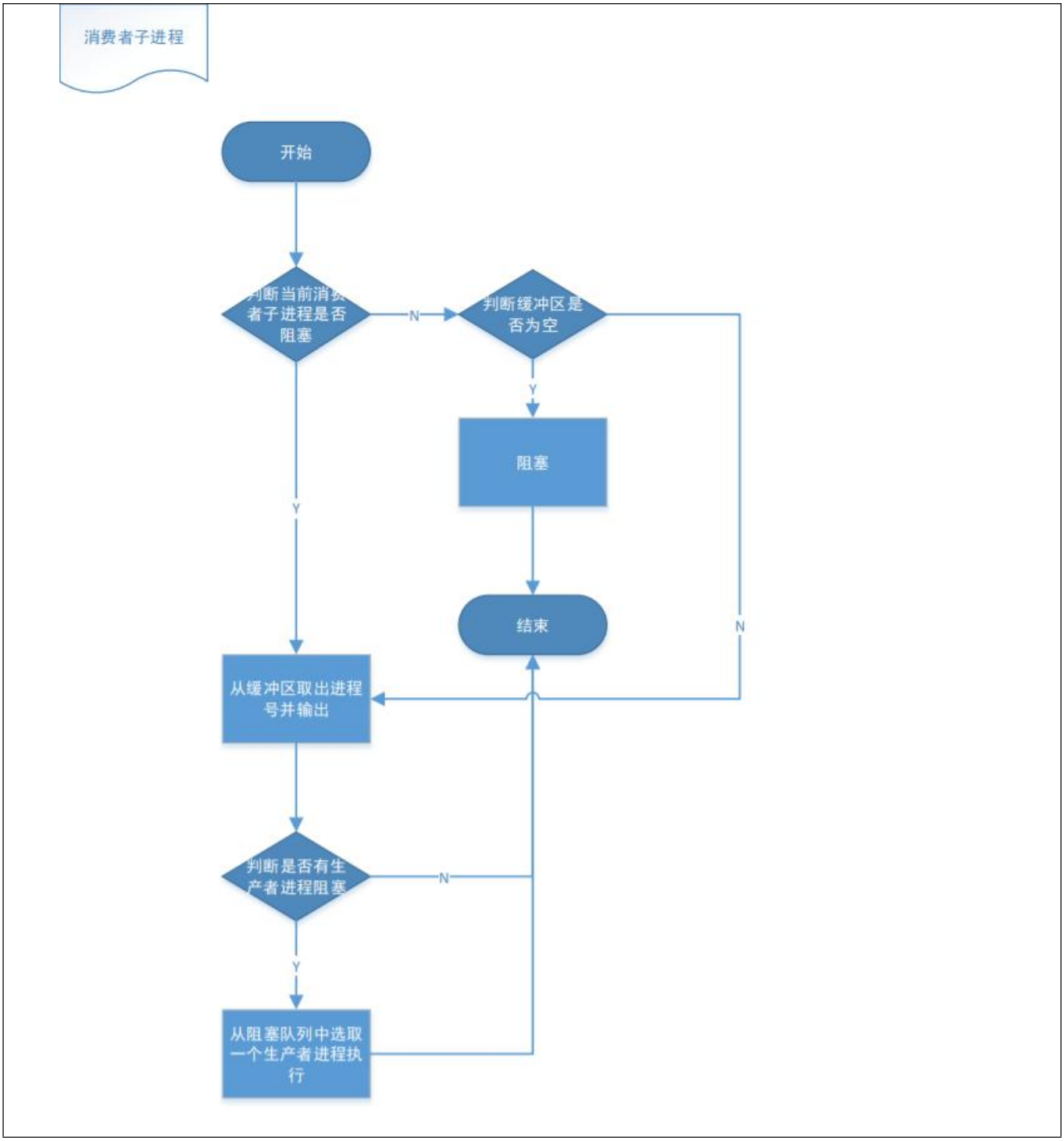
1. 调试、运行给出的程序，从操作系统原理的角度验证程序的正确性。
2. 发现并修改程序中的原理性错误或不完善的地方。
3. 鼓励在程序中增加新的功能。完成基本。

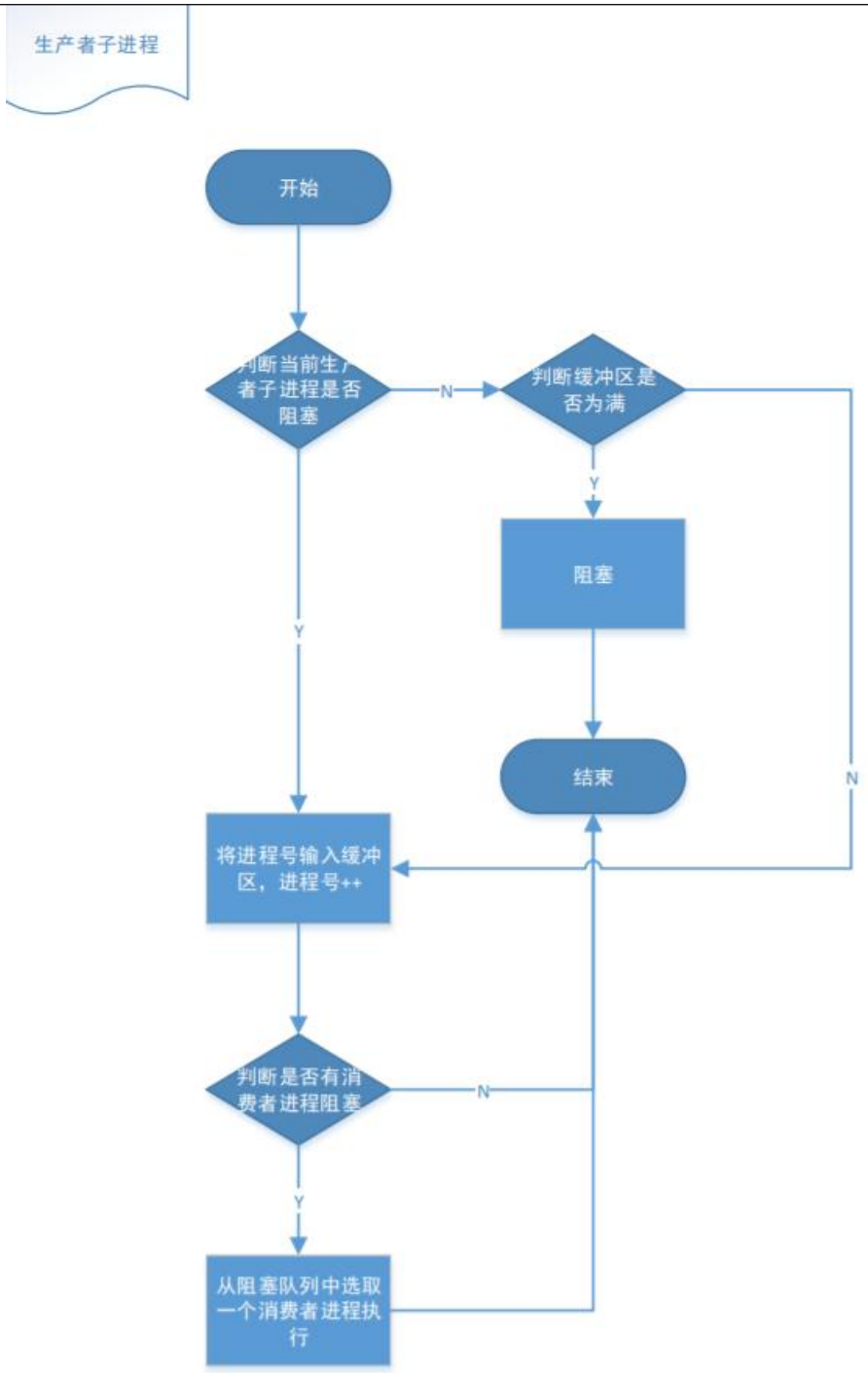
三．实验要求

1. 在程序中适当地加入注释。
2. 认真进行预习，阅读原程序，发现其中的原理性错误，完成预习报告。
3. 实验完成后，要认真总结，完成实验报告。

四．程序流程图


主进程
开始
输入所选择的操作
选择p
N
选择c
N
选择e
Y
Y
Y
执行生产者子进程
执行消费者子进程
结束


消费者子进程
开始
判断当前消费者子进程是否阻塞
N
判断缓冲区是否为空
Y
阻塞
Y
结束
N
从缓冲区取出进程号并输出
判断是否有生产者进程阻塞
N
Y
从阻塞队列中选取一个生产者进程执行



五．程序描述

利用写指针和读指针实现了生产者和消费者对缓冲区按顺序进行生产和消费。同时设定了缓冲区BUFFER，设定了缓冲区的大小。

当缓冲区已满时，再次创建的生产进程会进入阻塞队列中，而下次消费者消费了一个产品后，会自动将阻塞队列中的生产进程执行并将产品放入缓冲区。同理消费者也有消费者阻塞队列，也能在产品生产后自动执行消费进程。

主要函数的解释：

// 查找空位

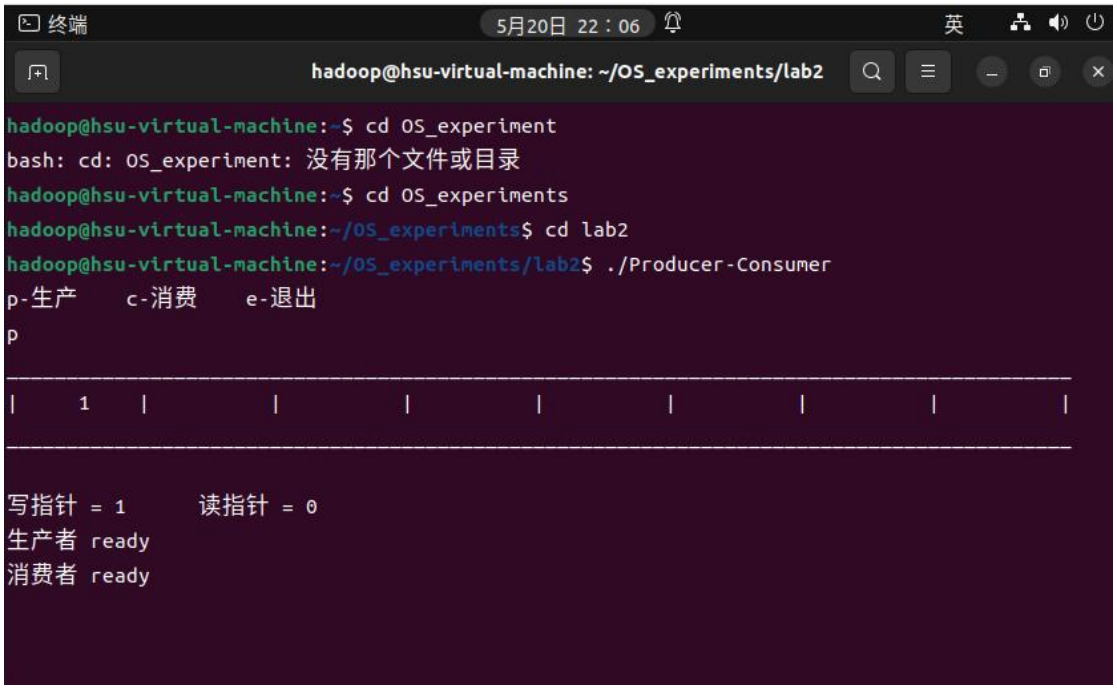
```
int findEmpty(vector<int> &v, int len)
```

查找哪个位置是空的，返回 0。

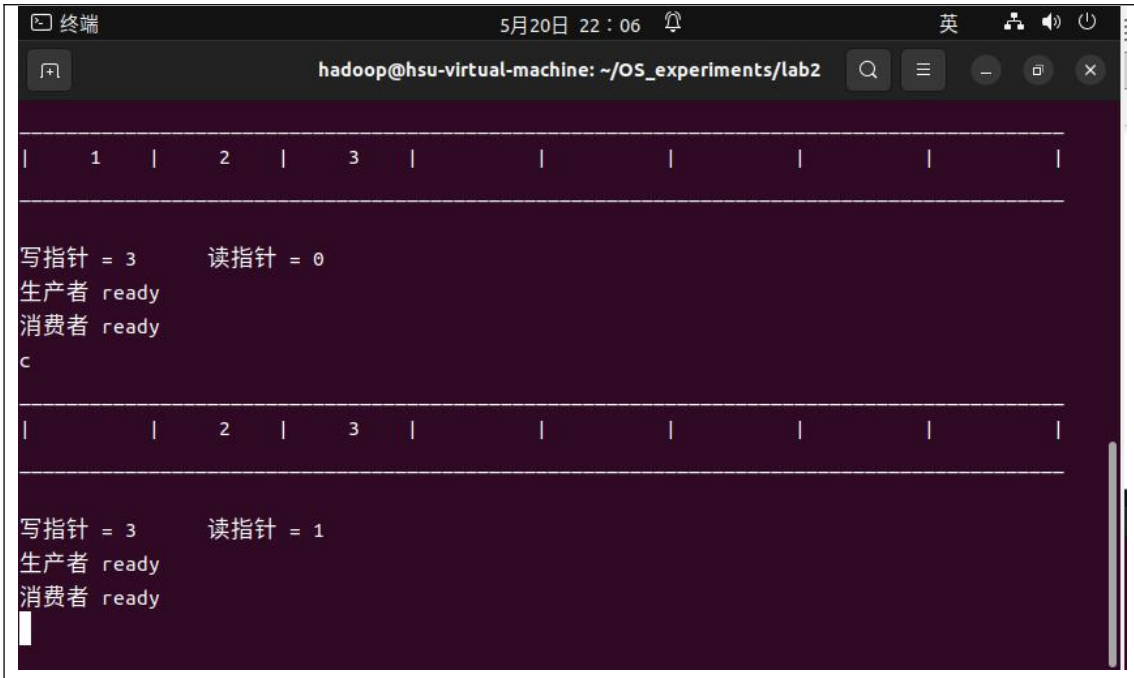
```
    // 查找脏位
    int findDirty(vector<int> &v, int len)

    // 缓冲区是否满，如果有大于零的位置，（查找到被写入正常数据的位置呗 i）就返回 i 否则都是 0，返回-1
    bool isFull(vector<int> &v, int len)

    // 缓冲区是否空
    bool isEmpty(vector<int> &v, int len)

    // 遍历展示队列中的内容
    void showQueue(queue<int> q)

    // 显示缓冲区内容
    void show()

    // 生产
    void produce()

    // 消费
    void consume()
```

六. 实验结果（截图）

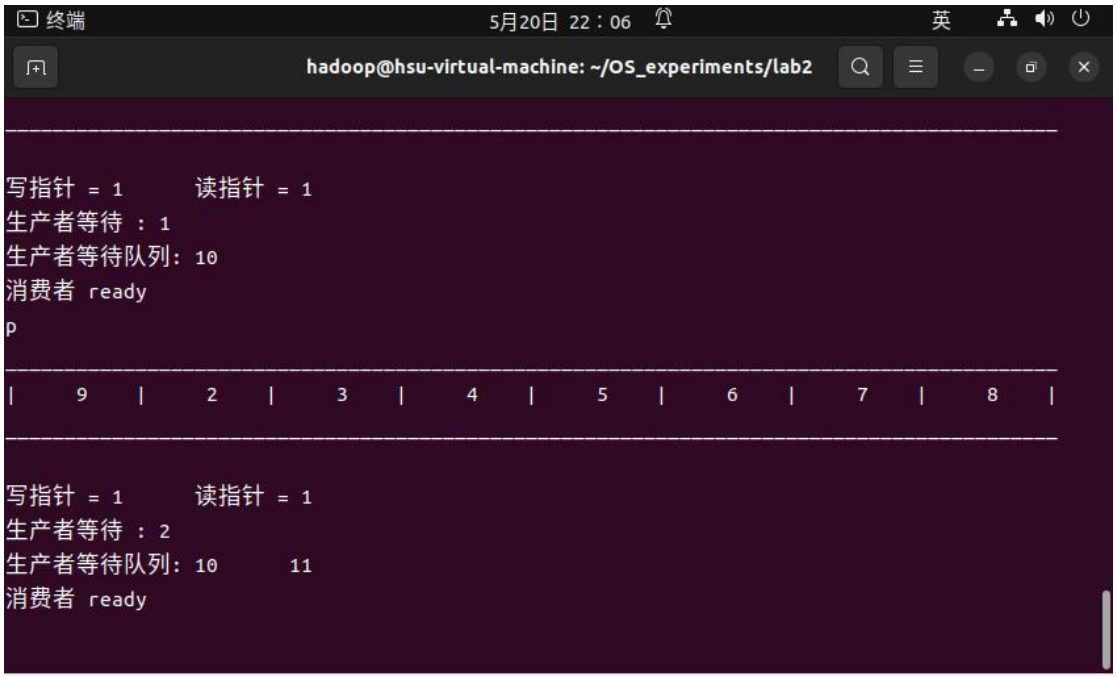
按 P 创建生产进程，将产出产品放入缓冲区。



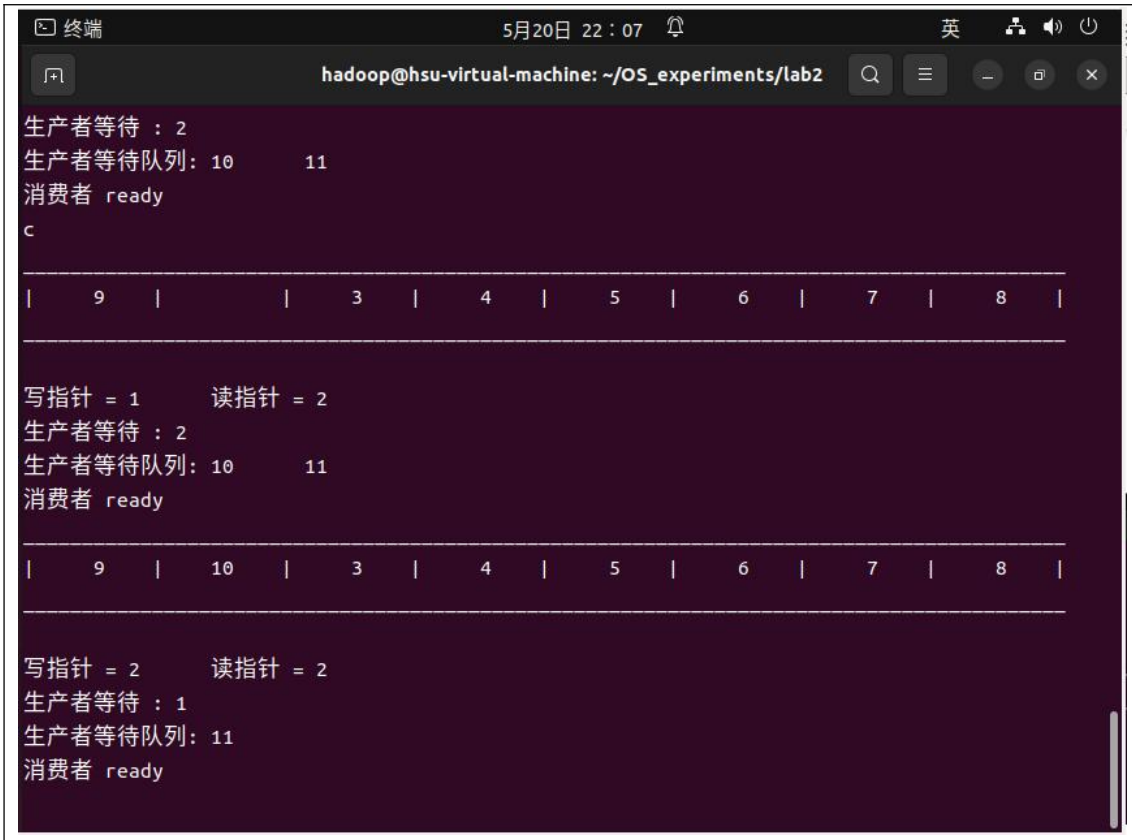
按 C 创建消费进程，对缓存区产品进行消费。



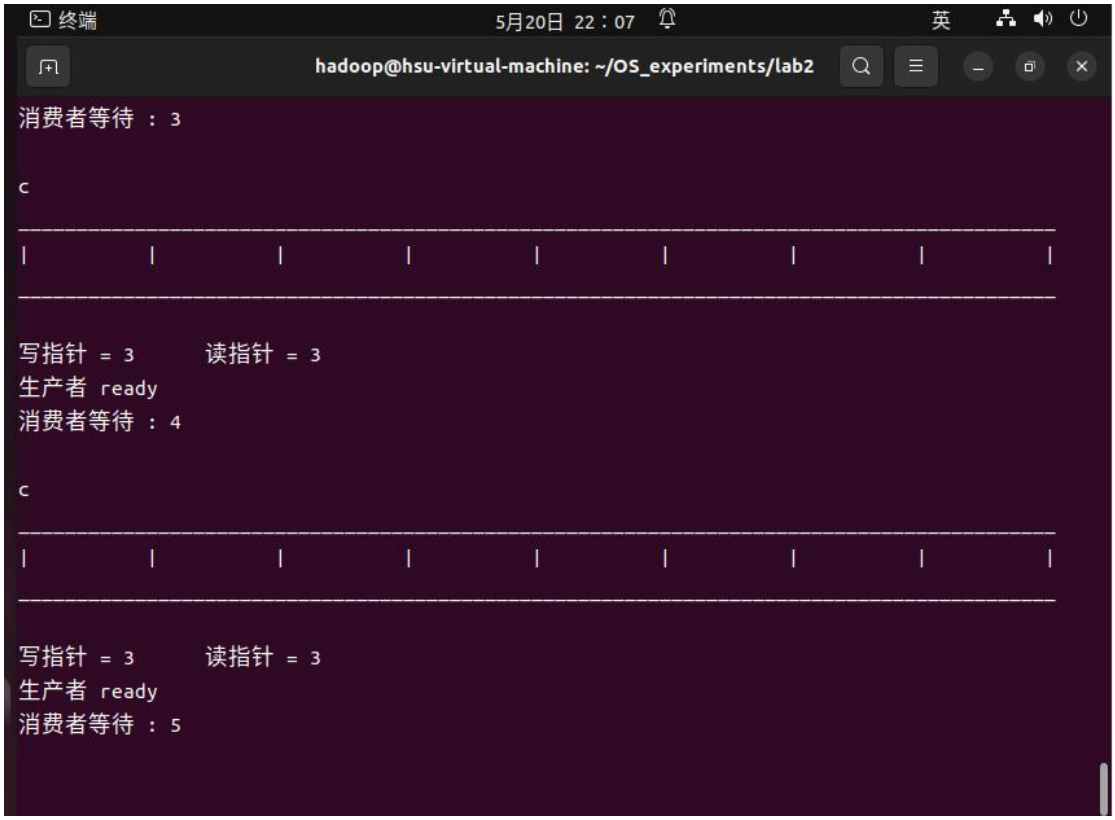
当缓冲区满时，新创建的生产者进程会被加入生产者阻塞队列。



当消费者进程执行后，缓冲区重新获得空间，会自动执行阻塞队列中的生产者进程。



消费者进程同样会被阻塞，并放入阻塞队列。



七. 源代码

```
// #include  <stdio.h>
// #include  <sys/types.h>
// #include  <stdlib.h>
// #include  <sys/stat.h>
// #include <fcntl.h>
// #include  <error.h>
// #include  <wait.h>
// #include  <unistd.h>

#include <iostream>
#include <vector>
#include <queue>
#include <string>
#include <unistd.h>

#define BUFFER_SIZE 8 // 缓冲区大小
using namespace std;

vector<int> buffer(BUFFER_SIZE); // 缓冲区
queue<int> pQueue;               // 生产者等待队列
queue<int> cQueue;               // 消费者等待队列
int number;                      // 0-99
```

```cpp
int writeptr, readptr;          // 写指针  读指针
int consumer, producer;         // 消费者等待数量  生产者等待数量

// 初始化
void init()
{
    for (int i = 0; i < BUFFER_SIZE; ++i)
    {
        buffer[i] = 0;
    }
    while (!pQueue.empty())
    {
        pQueue.pop();
    }
    while (!cQueue.empty())
    {
        cQueue.pop();
    }
    number = 0;
    writeptr = readptr = 0;
    consumer = producer = 0;
}

// 查找空位
int findEmpty(vector<int> &v, int len)
{
    for (int i = 0; i < len; ++i)
    {
        if (v[i] == 0)
        {
            return i;
        }
    }
    return -1;
}

// 查找脏位
int findDirty(vector<int> &v, int len)
{
    for (int i = 0; i < len; ++i)
    {
        if (v[i] > 0)
        {
            return i;
```

```
        }
    }
    return -1;
}

// 缓冲区是否满
bool isFull(vector<int> &v, int len)
{
    for (int i = 0; i < len; i++)
    {
        if (v[i] == 0)
        {
            return false;
        }
    }
    return true;
}

// 缓冲区是否空
bool isEmpty(vector<int> &v, int len)
{
    for (int i = 0; i < len; i++)
    {
        if (v[i] != 0)
        {
            return false;
        }
    }
    return true;
}

// 遍历展示队列中的内容
void showQueue(queue<int> q)
{
    int len = q.size();
    for (int i = 0; i < len; i++)
    {
        cout << q.front() << "  ";
        q.pop();
    }
}

// 显示缓冲区内容
void show()
```

```
{
    cout << "_____________________________________";
    cout << "_____________________________________________" << endl;
    for (int i = 0; i < BUFFER_SIZE; ++i)
    {
        if (buffer[i] > 0)
        {
            printf("|   %3d    ", buffer[i]);
            // cout << "|\t" << buffer[i] << "    ";
        }
        else
        {
            printf("|         ");

            // cout << "|\t";
            // cout << "    ";
        }
    }
    cout << "|\n";
    cout << "_____________________________________";
    cout << "_____________________________________________" << endl;

    cout << "\n 写指针 = " << writeptr << "\t 读指针 = " << readptr << "\t";

    // 打印生产者
    if (producer)
    {
        cout << "\n 生产者等待 : " << producer << "\n";
        cout << "生产者等待队列: ";
        showQueue(pQueue);
    }
    else
    {
        cout << "\n 生产者 ready\t";
    }

    // 打印消费者
    if (consumer)
    {
        cout << "\n 消费者等待 : " << consumer << "\n";
        // 不用显示 cQueue 里面的数值，事实上，里面的 number 数值不增加
        //  cout << "cQueue: ";
        //  showQueue(cQueue);
    }
```

```
    else
    {
        cout << "\n 消费者 ready";
    }
    cout << endl;
}

// 生产
void produce()
{
    if (findEmpty(buffer, BUFFER_SIZE) == -1)
    {
        // 缓冲区满,阻塞
        number = number + 1;
        producer++;
        pQueue.push(number);
    }
    else if (isEmpty(buffer, BUFFER_SIZE))
    {
        // 缓冲区空
        if (cQueue.empty())
        {
            // 消费者等待队列为空,直接写入即可
            number = number + 1;
            buffer[writeptr] = number;
        }
        else
        {
            // 有阻塞的消费者，先写入再读出
            number = number + 1;
            buffer[writeptr] = number;
            show(); // 不仅在 produce 函数运行全部完成后 show，在中间就 show 一次
            sleep(2);
            buffer[readptr] = 0;
            readptr = (readptr + 1) % BUFFER_SIZE;
            consumer--;
            cQueue.pop();
        }
        writeptr = (writeptr + 1) % BUFFER_SIZE;
    }
    else
    {
        // 缓冲区不空也不满,直接写入即可
        number = number + 1;
```

```
        buffer[writeptr] = number;
        writeptr = (writeptr + 1) % BUFFER_SIZE;
    }
}

// 消费
void consume()
{
    if (findDirty(buffer, BUFFER_SIZE) == -1) // buffer 里面全都是 0
    {
        // 缓冲区空,则阻塞
        // ++number;但事实上不能让 number++，不然 number 的数值就变了,
        consumer++;
        cQueue.push(number);
    }
    else if (isFull(buffer, BUFFER_SIZE))
    {
        // 缓冲区满
        if (!pQueue.empty())
        {
            // 生产者等待队列不空，则先读出再写入
            buffer[readptr] = 0;
            readptr = (readptr + 1) % BUFFER_SIZE;
            show(); // 不仅在 consume 函数运行全部完成后 show，在中间就 show 一次
            sleep(2);
            buffer[writeptr] = pQueue.front();
            writeptr = (writeptr + 1) % BUFFER_SIZE;
            pQueue.pop();
            producer--;
        }
        else
        {
            // 生产者队列为空
            buffer[readptr] = 0;
            readptr = (readptr + 1) % BUFFER_SIZE;
        }
    }
    else
    {
        // 缓冲区不空也不满,直接读取即可
        buffer[readptr] = 0;
        readptr = (readptr + 1) % BUFFER_SIZE;
    }
}
```

```
int main()
{
    cout << "p-生产    c-消费    e-退出" << endl;

    init();
    char c;
    cin >> c;
    while (c)
    {
        if (c == 'e')
        {
            break;
        }
        else if (c == 'p')
        {
            produce();
        }
        else if (c == 'c')
        {
            consume();
        }
        else
        {
            cout << "Please enter p-生产    c-消费    e-退出" << endl;
        }
        show();
        cin >> c;
    }
    return 0;
}
```

实验三

一．实验目的

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

二．实验内容

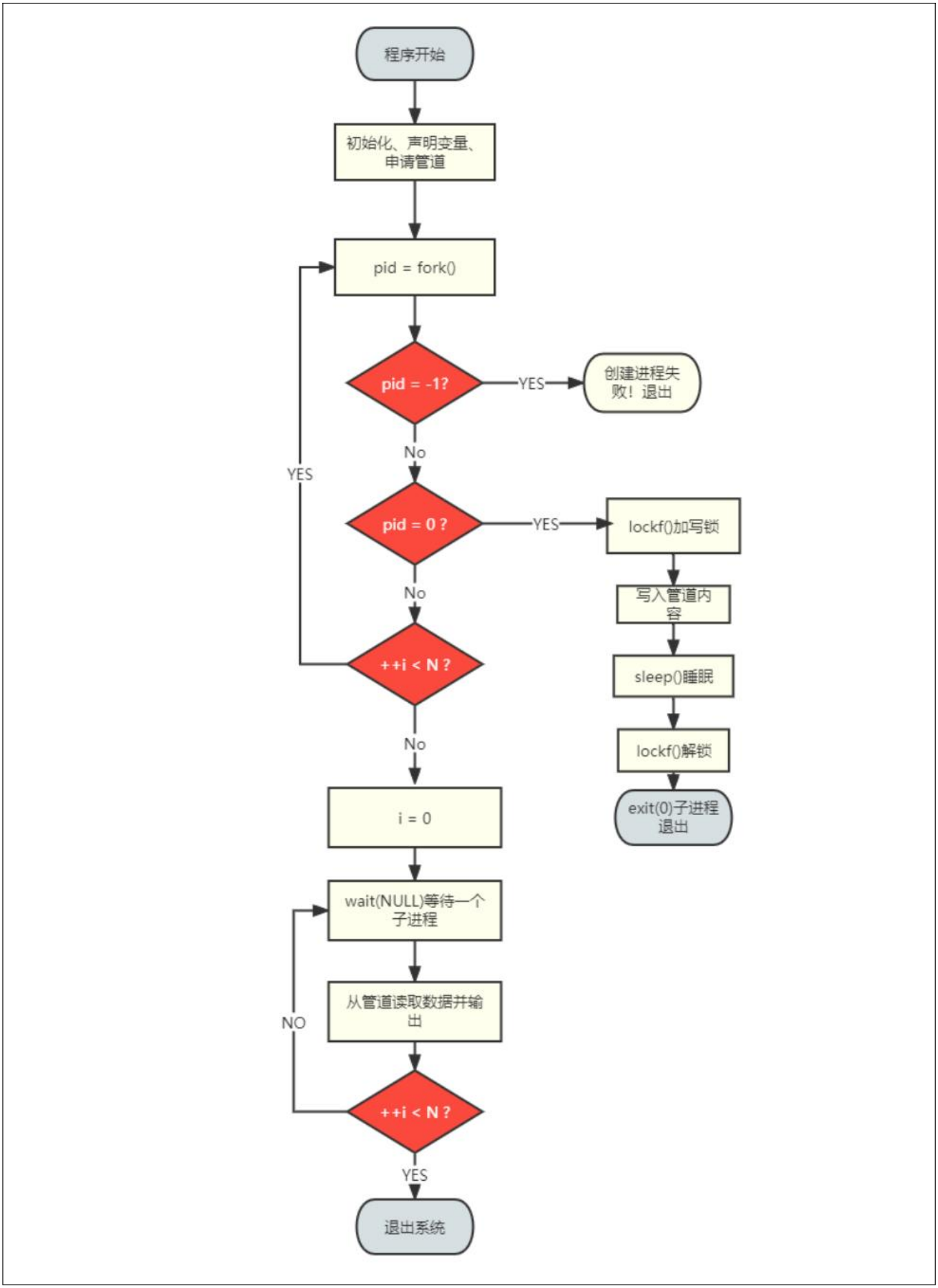
1. 使用系统调用 pipe()建立一条管道，系统调用 fork()分别创建两个子进程，它们分别向管道写一句话，如：
   Child process1 is sending a message!
   Child process2 is sending a message!

2.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。

三．实验要求

1.这是一个设计型实验，要求自行独立编制程序
2.两个子进程要并发执行。
3.实现管道的互斥使用。当一个子进程正在对管道进行写操作时，另一个欲写入管道的子进程必须等待。 使用系统调用 lockf(fd[1],1,0)实现对管道的加锁操作，用 lockf(fd[1],0,0)解除对管道的锁定。
4.实现父子进程的同步，当父进程试图从一空管道中读取数据时，便进入等待状态，直到子进程将数据写入管道返回后，才将其唤醒。

四．程序流程图


程序开始
初始化、声明变量、申请管道
pid = fork()
pid = -1?
YES
创建进程失败！退出
No
YES
pid = 0 ?
YES
lockf()加写锁
No
写入管道内容
++i < N ?
sleep()睡眠
No
lockf()解锁
exit(0)子进程退出
i = 0
wait(NULL)等待一个子进程
从管道读取数据并输出
NO
++i < N ?
YES
退出系统

五．程序描述

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。随机创建了三个子进程，并且按照相关语句进行输出以检查进程执行顺序。过程中通过加锁解锁和 wait 函数对进程进行控制。（事实上是 n 个紫荆城

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

六．实验结果（截图）

```
hadoop@hsu-virtual-machine:~/OS_experiments/lab3$ ./Pipeline
Please input the number of processes: 3

------------------------分割线---------------------
pid: 5175
pid: 0
now I'm writing in subprocess 5175
send message successfully and exit subprocess 5175

father process reads from subprocess 5175:
child process 5175 is sending message!


------------------------分割线---------------------
pid: 5176
pid: 0
now I'm writing in subprocess 5176
send message successfully and exit subprocess 5176

father process reads from subprocess 5176:
child process 5176 is sending message!


------------------------分割线---------------------
pid: 5177
pid: 0
now I'm writing in subprocess 5177
send message successfully and exit subprocess 5177

father process reads from subprocess 5177:
child process 5177 is sending message!


------------------------分割线---------------------
```

七. 源代码

```
#include <stdio.h>
#include <sys/types.h>
```

```c
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <wait.h>
#include <error.h>

#define F_ULOCK 0 // 解锁
#define F_LOCK 1  // 互斥锁定区域

int main()
{
    int num, fd[2], result = -1;
    pid_t pid, reid, gtid;
    char outpipe[100], father_str[100];

    // 创建管道，fd[0]读管道，fd[1]写管道
    result = pipe(fd);
    if (result == -1) // 调用失败
    {
        printf("fail to create pipe \n");
        return -1;
    }

    // 创建子进程
    printf("Please input the number of processes: ");
    scanf("%d", &num);
    printf("\n--------------------------分割线----------------------\n");

    while (num > 0)
    {
        pid = fork();
        printf("pid: %d\n", pid);

        if (pid == -1)
        {
            // 创建失败
            printf("creat subprocess failed!\n");
            exit(0);
        }
        else if (pid == 0)
        {
            // 创建子进程，从子进程返回ID
            printf("now I'm writing in subprocess %d\n", gtid = getpid());
```

```
            lockf(fd[1], F_LOCK, 0); // 加锁
            sprintf(outpipe, "child process %d is sending message!\n", gtid);
            write(fd[1], outpipe, sizeof(outpipe));
            sleep(1);
            lockf(fd[1], F_ULOCK, 0); // 解锁
            printf("send message successfully and exit subprocess %d\n", gtid);
            exit(0);
        }
        else if (pid > 0)
        {
            // 创建子进程，从父进程返回子进程的ID
            reid = wait(NULL); // 等待一个子线程结束？？？
            if (reid == -1)
            {
                printf("father process calls subprocess failed!\n");
            }
            else
            {
                read(fd[0], father_str, sizeof(father_str));
                printf("\nfather process reads from subprocess %d:\n", reid);
                printf("%s\n", father_str);
            }
        }
        printf("\n-------------------------分割线----------------------\n");

        num--;
    }
    return 0;
}
```

八．思考题

① 指出父进程与三个子进程并发执行的顺序，并说明原因。

答：①父进程　：通过 wait()阻塞父进程
②子进程 1　：操作系统调度算法调度
③父进程　：子进程 1 结束，父进程从 wait()返回继续执行原来程序
④父进程　：通过 wait()阻塞父进程
⑤子进程 3　：操作系统调度算法调度
⑥父进程　：子进程 3 结束，父进程从 wait()返回继续执行原来程序
⑦父进程　：通过 wait()阻塞父进程
⑧子进程 2　：操作系统调度算法调度
⑨父进程　：子进程 2 结束，父进程从 wait()返回继续执行原来程序

② 若不对管道加以互斥控制，会有什么后果？

答：会导致父进程从管道中读出的信息不一定为对应子进程输入的信息

③ 说明你是如何实现父子进程之间的同步的。

通过 wait()函数实现父子进程之间的同步。在父进程中调用 wait()，则父进程被阻塞，进入等待队列，等待子进程结束。当子进程结束时，父进程从 wait()返回继续执行原来的程序。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

实验四

一、实验目的：

进一步理解父子进程之间的关系。

1） 理解内存页面调度的机理。

2） 掌握页面置换算法的实现方法。

3） 通过实验比较不同调度算法的优劣。

4） 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，比较各种置换算法的效率及优缺点，从而了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。

二、实验内容及要求

这是一个综合型实验，要求在掌握父子进程并发执行机制和内存页面置换算法的基础上，能综合运用这两方面的知识，自行编制程序。

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。要求：

1） 每个子进程应能反映出页面置换的过程，并统计页面置换算法的命中或缺页情况。

设缺页的次数为 diseffect。总的页面访问次数为 total_instruction。

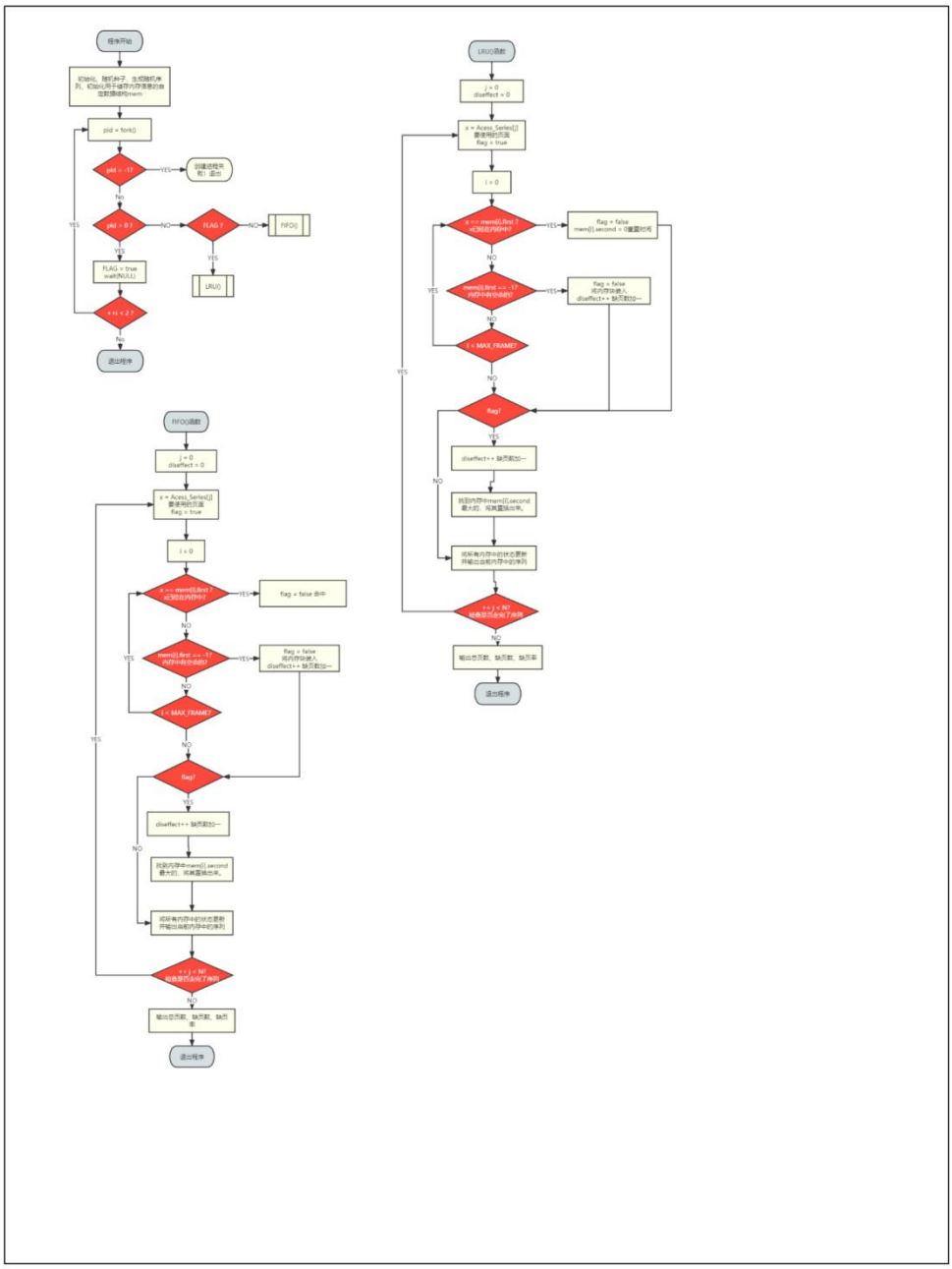
缺页率 = disaffect/total_instruction

命中率 = 1- disaffect/total_instruction

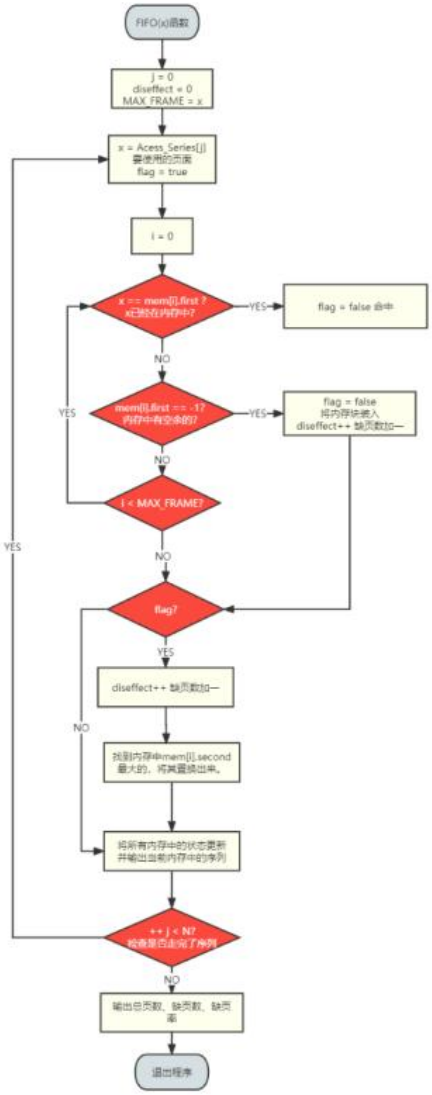
2)将为进程分配的内存页面数 mframe 作为程序的参数,通过多次运行程序,说明 FIFO 算法存在的 Belady 现象。

三、程序流程图

基础点的程序大体流程图如下：


程序开始
pid = fork()
pid == -1?
YES
创建进程失败！退出
No
pid > 0 ?
NO
FLAG ?
NO
FIFO()
YES
FLAG = true
wait(NULL)
YES
LRU()
i = i + 1 ?
No
退出程序
FIFO()函数
j = 0
diseffect = 0
x = Acess_Series[j]
i = 0
x == mem[i].first ?
YES
flag = false 命中
NO
mem[i].first == -1 ?
YES
flag = false
将内存块装入
diseffect++ 缺页数加一
NO
i < MAX_FRAME?
NO
flag?
YES
diseffect++ 缺页数加一
NO
找到内存中mem[i].second
最大的，将其置换出来。
将所有内存中的状态更新
并输出当前内存中的序列
++ j < N?
检查是否全部了序列
NO
输出总页数、缺页数、缺页率
退出程序
LRU()函数
j = 0
diseffect = 0
x = Acess_Series[j]
i = 0
x == mem[i].first ?
YES
flag = false
mem[i].second = 0重置时间
NO
mem[i].first == -1 ?
YES
flag = false
将内存块装入
diseffect++ 缺页数加一
NO
i < MAX_FRAME?
NO
flag?
YES
diseffect++ 缺页数加一
NO
找到内存中mem[i].second
最大的，将其置换出来。
将所有内存中的状态更新
并输出当前内存中的序列
++ j < N?
检查是否全部了序列
NO
输出总页数、缺页数、缺页率
退出程序

扩展点一的流程图：



四、程序描述

对生产的随机页面数据，开展了 FIFO 与 LRU 的页面置换算法，并对置换效果和命中率等信息进行展示。并对 FIFO 中的 Belady 现象进行了检验。

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。

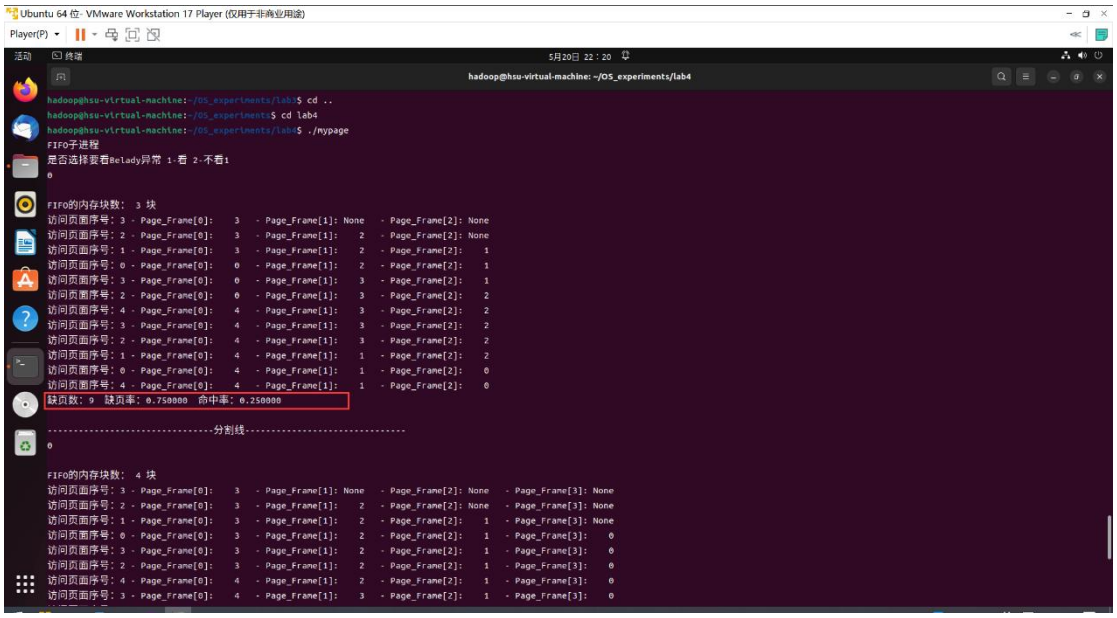
完成了扩展点一： 可以使用父进程创建了两个子进程,两个子进程分别使用不同大小的驻留集，都是用 FIFO 的置换算法，最终可以观察到驻留集更大的子进程反而缺页率更高。

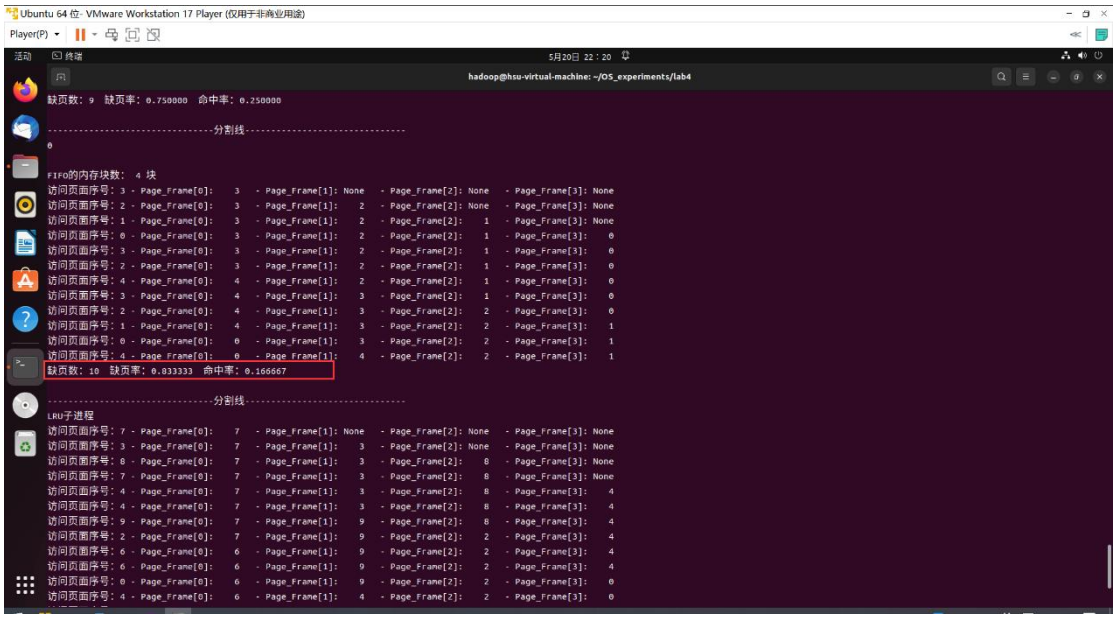
程序涉及一个父进程和两个子进程。父进程使用 rand()函数随机产生若干随机数，经过处理后，存于一数组 Acess_Series[]中，作为内存页面访问的序列。两个子进程根据这个访问序列，分别采用 FIFO 和 LRU 两种不同的页面置换算法对内存页面进行调度。同时统计不同的页面调度算法的缺页率。在 linux 系统的环境下使用 C++ 并用 g++ 进行编译完成实验，使用了系统调用 fork(),wait(),exit(),sleep(),rand()等。可以随意修改程序序列的长度。

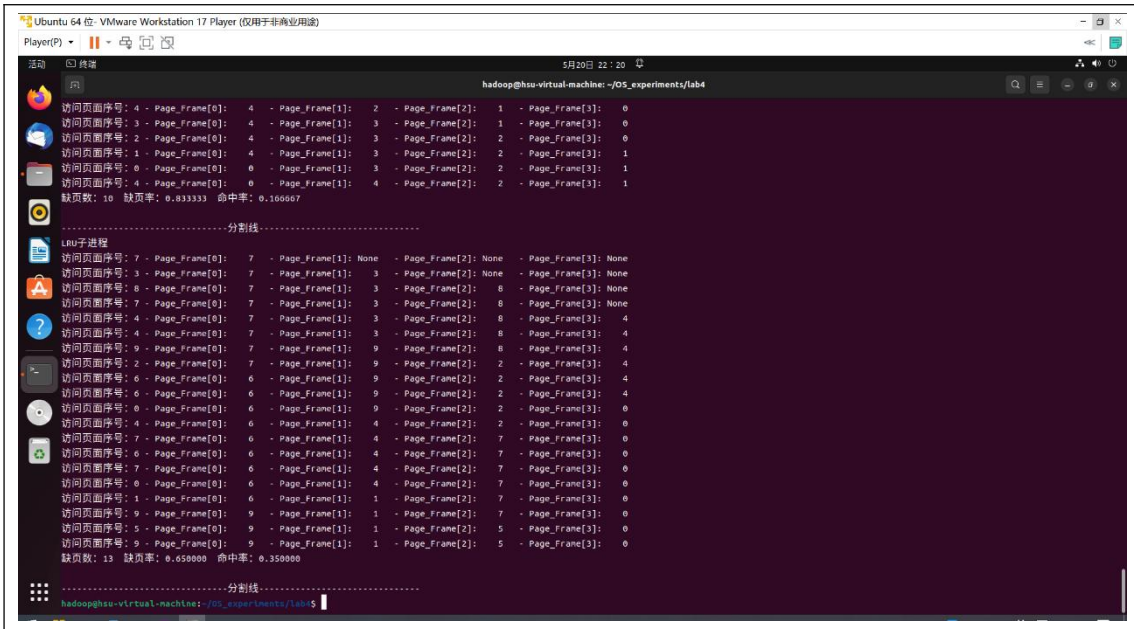
五、实验结果（截图）

对于随机生产的页面序号分别进行 FIFO、LRU 进行页面置换，对 FIFO 进行再一次执行以检验是否有 Belady 现象发生。

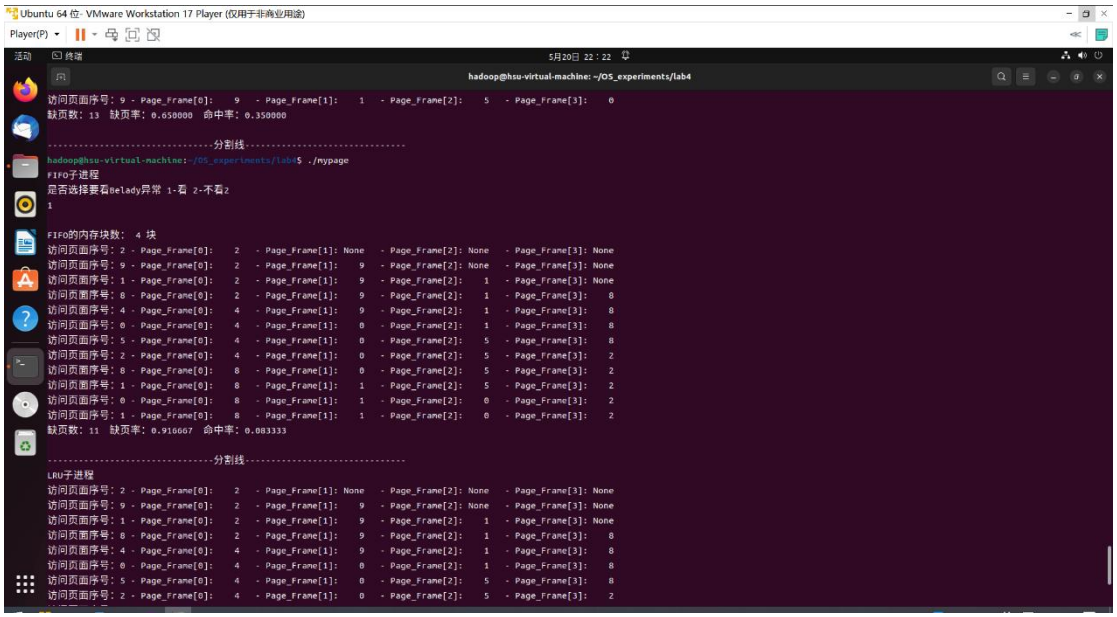
若选择要看 Belady 异常

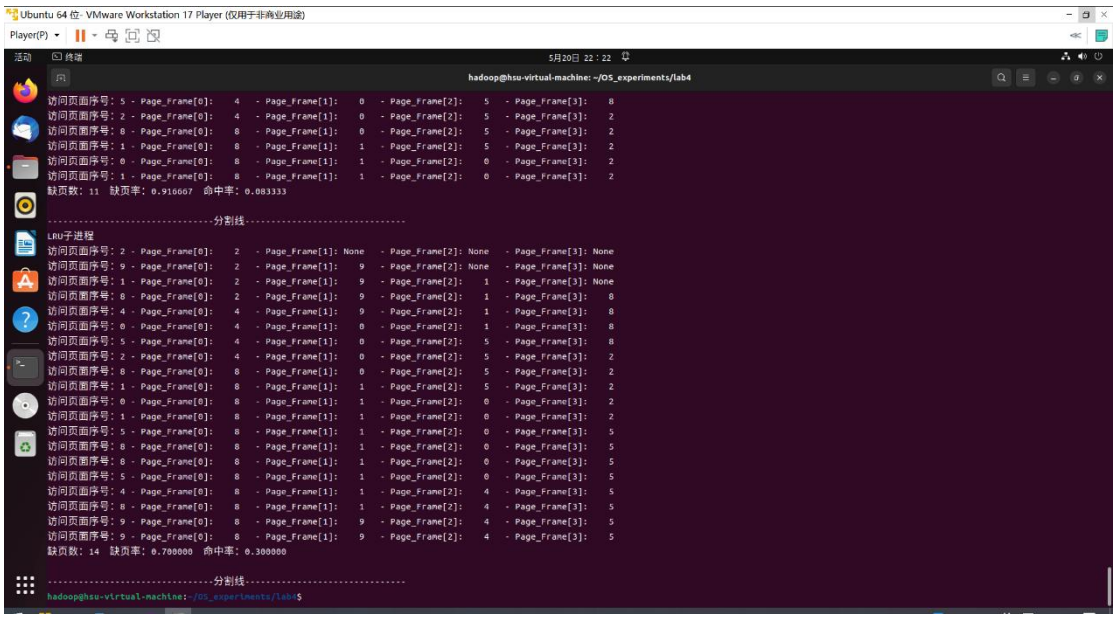






若选择不要看 Belady 异常





六、源代码

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <error.h>
#include <wait.h>
#include <unistd.h>
#include <sys/select.h>
#include <semaphore.h>
#include <deque>
#include <vector>
#include <algorithm>
#include <time.h>

class Page_Frame
{
public:
    int pageframe_size;                     // 内存块数
    int occupied_size;                      // 被使用的内存块数，如果没满，直接加入就好了
    std::vector<int> pageframe_array;       // 内存块存放的访问页面的序号组成的数组
    std::vector<int> pageframe_array_time; // LRU 里面的判断最久的依据

    // 初始化
    Page_Frame(int m = 0, int n = 0)
    {
        pageframe_size = m;
```

```
        occupied_size = 0;
    }
    // 输出信息的函数
    void print_info()
    {
        for (int i = 0; i < pageframe_size; i++)
        {
            if (i < pageframe_array.size()) // 内存块还没有全部被页面序号站满
            {
                printf("- Page_Frame[%d]: %4d\t", i, pageframe_array[i]);
            }
            else
            {
                printf("- Page_Frame[%d]: None\t", i); // N 代表内存块为空
            }
        }
        printf("\n");
    }

    // pageframe_array_time 是当下时刻的内存块存放的访问页面的序号组成的数组
    void add_pageframe_array_time()
    {
        // iter 就是指针
        for (auto iter = pageframe_array_time.begin(); iter !=
pageframe_array_time.end(); iter++)
        {
            (*iter)++;
            /*
            (*iter)++ 表示先解引用 iter 获取到指向的元素，再对该元素进行自增操作，最后返回
该元素自增前的值。
            这种写法会修改数组元素的值，但不会修改迭代器 iter 的指向。
            *iter++ 表示先解引用 iter 获取到指向的元素，再将迭代器向后移动一个位置。
            该写法会修改迭代器 iter 的指向，但不会修改数组元素的值。
            */
        }
    }
};
// 实现FIFO 算法
void FIFO(int memory_blocks_number, int choice = 0)
{
    printf("%d\n",choice);
    printf("\nFIFO 的内存块数： %d 块\n", memory_blocks_number);
    // 这里没有随机化数组元素，之后再加
    /*
```

```
    total_instructions 为总的页面访问次数
    diseffect 为缺页数
    pageid 为想访问的页面的序号
    */
    int total_instructions = 12;
    srand((unsigned)time(NULL));

    int Acess_Series[total_instructions] = {3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4};
    if (choice == 1)
    {
        for (int i = 0; i < total_instructions; i++)
        {
            Acess_Series[i] = rand() % 10;
        }
    }
    // srand((unsigned)time(NULL));

    // 3
    Page_Frame page_Frame(memory_blocks_number);
    std::deque<int> fifo_queue; // 双段队列
    int diseffect = 0;

    for (int i = 0; i < total_instructions; ++i)
    {
        int pageid = Acess_Series[i];
        printf("访问页面序号: %d ", pageid);
        /*
        对 Acess_Series 数组里面的数字，判断以下三种情况
        1 内存块中存在此页面,什么都不用做
        2 内存块中无此页面，且内存块未满，直接加入
        3 内存块中无此页面，且内存块已满，取出队列首个元素（FIFO 算法）
        */
        // find 返回的是指针，指向找序列里面第一个出现的地方
        // 存在此页面
        if (find(page_Frame.pageframe_array.begin(),
page_Frame.pageframe_array.end(), pageid) != page_Frame.pageframe_array.end())
        {
            page_Frame.print_info();
        }
        else
        {
            // 无此页面，缺页数先加一
            diseffect++;
            fifo_queue.push_back(pageid);
```

```
            // 同时内存快未满
            if (page_Frame.occupied_size < page_Frame.pageframe_size)
            // if (page_Frame.pageframe_array.size() < page_Frame.pageframe_size)
            {
                // 直接加入
                page_Frame.occupied_size++;
                page_Frame.pageframe_array.push_back(pageid);
                page_Frame.print_info();
            }
            else
            {
                // 同时内存块已满
                int be_replaced_pageid = fifo_queue.front();
                fifo_queue.pop_front();
                auto be_replaced_pageid_iter =
find(page_Frame.pageframe_array.begin(), page_Frame.pageframe_array.end(),
be_replaced_pageid);
                *be_replaced_pageid_iter = pageid;
                page_Frame.print_info();
            }
        }
    }
    printf("缺页数：%d  缺页率：%f  命中率：%f\n", diseffect, (float)diseffect /
total_instructions, (1 - (float)diseffect / total_instructions));
    printf("\n---------------------------------分割线
--------------------------------\n");
}

void LRU()
{
    int total_instructions = 20;

    int Acess_Series[total_instructions] = {1, 8, 1, 7, 8, 2, 7, 2, 1, 8, 3, 8, 2,
1, 3, 1, 7, 1, 3, 7};
    for (int i = 0; i < total_instructions; i++)
    {
        Acess_Series[i] = rand() % 10;
    }

    Page_Frame page_Frame(4); // 是一个数组
    int diseffect = 0;        // 缺页数
    for (int i = 0; i < total_instructions; i++)
    {
        int pageid = Acess_Series[i];
```

```
        printf("访问页面序号：%d ", pageid);
        /*
       对Acess_Series数组里面的数字，判断以下三种情况
       1 内存块中存在此页面,则把其次数置为0即可
       2 内存块中无此页面，且内存块未满，直接加入
       3 内存块中无此页面，且内存块已满，找到最近最久未使用的页面
       */
        std::vector<int>::iterator iter;
        if ((iter = find(page_Frame.pageframe_array.begin(),
page_Frame.pageframe_array.end(), pageid)) != page_Frame.pageframe_array.end())
        {
            page_Frame.add_pageframe_array_time();
            int position = std::distance(page_Frame.pageframe_array.begin(),
iter);
            page_Frame.pageframe_array_time.at(position) = 0; // 新出现一次，就置成
0,越大代表越久没有访问，0代表刚刚访问
            page_Frame.print_info();
        }
        else
        {
            // 内存中无此页面,缺页数加一
            diseffect++;
            // 且内存块未满，直接加入
            if (page_Frame.occupied_size < page_Frame.pageframe_size)
            {
                page_Frame.add_pageframe_array_time();
                page_Frame.occupied_size++;
                page_Frame.pageframe_array.push_back(pageid);
                page_Frame.pageframe_array_time.push_back(0); // 0代表刚刚访问
                page_Frame.print_info();
            }
            else
            {
                // 且内存块已满，找到最近最久未使用的页面,也就之找到times次数最大的数组下标
                page_Frame.add_pageframe_array_time();
                auto max_iter =
std::max_element(page_Frame.pageframe_array_time.begin(),
page_Frame.pageframe_array_time.end());
                int max_position =
std::distance(page_Frame.pageframe_array_time.begin(), max_iter);
                page_Frame.pageframe_array.at(max_position) = pageid;
                page_Frame.pageframe_array_time.at(max_position) = 0;
                page_Frame.print_info();
            }
```

```
        }
    }
    printf("缺页数：%d  缺页率：%f  命中率：%f\n", diseffect, (float)diseffect /
total_instructions, (1 - (float)diseffect / total_instructions));
    printf("\n--------------------------------分割线
--------------------------------\n");
}
int main()
{

    // srand((unsigned)time(NULL));

    // FIFO(3);
    // FIFO(4);
    // printf("_______________________________");
    // LRU();

    sem_t *mutex;
    // create mutex semaphore
    if ((mutex = sem_open("/1mutex_semaphore", O_CREAT | O_EXCL, 0666, 1)) ==
SEM_FAILED)
    {
        perror("mutex_semphore error");
        exit(1);
    }

    pid_t pid;
    for (int childprocess = 1; childprocess <= 2; childprocess++)
    {
        while ((pid = fork()) == -1)
            ;
        if (pid == 0)
        {
            // printf("%d %d %d\n", childprocess, getpid(), getppid());
            if (childprocess == 1) // 子进程1
            {
                sem_wait(mutex);
                printf("FIFO 子进程\n");
                int n;
                printf("是否选择要看 Belady 异常 1-看 2-不看");
                scanf("%d", &n);
                if (n == 1)
                {
                    FIFO(3);
```

```
                FIFO(4);
            }
            else
            {
                srand((unsigned)time(NULL));
                FIFO(4, 1);
            }

            sem_post(mutex);
        }

        else // 子进程2
        {
            sem_wait(mutex);
            printf("LRU 子进程\n");
            srand((unsigned)time(NULL));
            LRU();
            sem_post(mutex);
        }
        exit(0);
    }
    else
        wait(NULL);
}
// delete semaphore
if (sem_unlink("/1mutex_semaphore") == -1)
{
    perror("semaphore unlink error");
    exit(1);
}
return 0;
}
```

七、思考题

① 父进程、子进程之间的并发执行的过程

父进程与子进程之间的并发执行宏观并行，微观串行。从宏观来说，父进程创建子进程 1，子进程 1 和父进程同时执行，直到父进程创建子进程 2 遇到 wait()函数挂机为止，当子进程 1 结束父进程和子进程 2 并发执行到再次遇见 wait()函数是挂起等待子进程 2 结束，到子进程 2 结束返回父进程父进程继续执行至结束。

从微观来说，父进程先执行至创建子进程 1，接下来父进程挂起执行子进程 1 知道子进程 1 结束回到父进程；父进程继续执行到创建子进程 2 再次挂起，执行子进程 2，直到子进程 2 结束回到父进程继续执行至结束。

② 通过完成实验，根据你的体会，阐述虚拟存储器的原理。

虚拟存储器实际上是用来解决作业大而内存小的问题，他通过页面置换算法来提供远大于内存地址空间的地址范围，针对不同的程序将不同的数据页面读取到虚拟存储器中用来实现。

③ 写出 FIFO 算法中出现 Belady 现象的内存页面访问序列。

3 个内存页面数：

int Acess_Series[total_instructions] = {3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4};

缺页数：9

缺页率：0.75

命中率：0.25

4 个内存页面数：

int Acess_Series[total_instructions] = {3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4};

缺页数：10

缺页率：0.833333

命中率：0.166667