一、目的

1）巩固和加深对操作系统基本知识的理解与掌握，能够根据操作系统的知识完成计算机领域的复杂工程问题模块的设计与实现。

2）能够合理选择技术、资源、现代工程工具和信息技术工具，具备将其运用于操作系统的分析、设计、开发及测试过程中的能力。

3）能够根据计算机领域的复杂工程问题进行有效的沟通和交流，具备具有撰写报告、设计文稿、陈述发言和清晰表达以及倾听并回应问题的能力。

二、原理

运用现代操作系统课程的相关知识，设计操作系统的内核原型，包括处理机调度、存储器管理、I/O 设备分配和文件管理。

1）处理机调度

处理机调度，也称为 CPU 调度，是操作系统中的一个核心功能，它负责决定哪个进程应该获得 CPU 时间。处理机调度的主要目标是**最大化 CPU 利用率**、提供合理的进程响应时间以及保证进程的公平性。

抢占式与非抢占式调度：抢占式调度允许操作系统在进程执行期间中断它以调度另一个更高优先级的进程，而非抢占式调度则不允许中断。

2）存储器内存分区管理

存储器内存分区管理是指将物理内存划分为固定或动态大小的分区，并进行管理以供进程使用。

固定分区：物理内存被划分为固定大小的分区，每个分区只能被一个进程占用。这种方法简单，但可能导致内存利用率低。

动态分区：物理内存被划分为动态大小的分区，可以根据进程需求动态调整大小。这可以提高内存利用率，但管理更复杂。

分区分配策略：包括首次适应（First Fit）、最佳适应（Best Fit）和最坏适应（Worst Fit）等算法，用于找到合适的分区分配给进程。

分区合并：当进程释放内存时，相邻的空闲分区可以合并，以**减少内存碎片**。

3）虚拟存储器页面管理

虚拟存储器页面管理是一种内存管理技术，它允许操作系统使用比物理内存更多的内存空间，通过**将部分内存存储在磁盘**上。

页面：虚拟内存被划分为固定大小的页面，物理内存同样被划分为页面大小相同的页框。

页面置换算法：当一个页面需要被加载到物理内存而没有足够的空间时，页面置换算法决定哪个页面被换出。常见的算法包括 FIFO、LRU、LFU 等。

页面请求：进程请求内存时，如果请求的页面不在物理内存中，会发生缺页中断，操作系统会从磁盘中加载所需的页面。

4）磁盘调度

磁盘调度是操作系统用于管理磁盘 I/O 请求的机制，目的是**减少磁盘寻道时间和旋转延迟**，提高磁盘性能。

先来先服务（FCFS）：按照磁盘请求的顺序进行服务，简单但可能导致磁头移动时间长。

最短寻道时间优先（SSTF）：选择最近的磁道进行服务，可以减少寻道时间，但可能导致饥饿（STARVATION）。

扫描（SCAN）：磁头从一个方向移动，直到最后一个请求被服务，然后改变方向。这种方法可以减少寻道时间，并且比 SSTF 更公平。

LOOK（电梯算法）：与 SCAN 类似，但是磁头在到达一端后会直接跳到另一端，而不是继续扫描。

三、方案

3.1 系统总体框架

本系统**融合了四大模块，集成成一个大系统**，并以**生产者消费者**的工作方式来运行的，主要融合形式如下:

1）在最开始时，我们需要初始化**预生产的字符或字符串**，字符串将由模拟操作系统从文件（预生产字符串.txt）中读取，该文件存放在模拟磁盘上，**模拟磁盘需要该文件同时存储其在磁盘上的位置（柱面号和磁道号）**，模拟操作系统会根据其所在位置按用户选择的**磁道调度算法（T_FCFS，T_SSTF，TSCAN）**读取文件，从而初始化字符串；

2）生产者消费者进程不运行时，放入就绪队列或阻塞队列。如果要执行某个进程，则首先将其插入运行队列中，进程的插入顺序由用户选择的**进程调度算法（FCFS、SPF、PRIORITY、HRRN）**决定，待一个执行周期结束后，运行队列已经全部插入下一个执行周期将要运行的所有进程。而生产者消费者多进程运行时，

可能会发生**不可重复读、脏读、死锁等并行问题**，需要设计 PV 原语来规避此问题，同时将可能发送死锁的进程插入阻塞队列，直到释放某些资源。

3）执行程序，事实上是将所有在运行队列上的进程插入到内存 m 中，而内存的存储形式由用户选择的**分区方式（固定分区或可变分区）**决定，一个进程可以有多个页（一个页 1024B）

4）为**进程分配内存**时，其实是检查到进程有缺页情况，并将指定的页面号插入到内存中，如果内存此时大小不足以插入这个页面号，则根据用户选择的**页面调度算法(FIFO、最近最少使用调度算法(LRU)、最近最不常用调度算法(LFU))**指定淘汰页面号，并插入新页面号。

以下是模拟操作系统的总体架构：

```
Project/
├头文件
|    ├──── DEFINITION.h          # 全局头文件
|    ├──── Process.h         # 进程模块头文件
|    ├──── Memory.h          # 内存模块头文件
|    └──── FileManage.h      # 文件模块头文件
├源文件                  # 源代码文件
├──── src/                  # 源代码文件
|    ├──── main.c            # 主程序入口
|    ├──── Init.c            # 系统初始化模块
|    ├──── Global_Init.c     # 全局参数初始化
|    ├──── Queue.c           # 队列管理模块
|    ├──── Process_Module/          # 进程管理模块
|    |    ├──── PV.c                 # 进程阻塞&释放
|    |    └──── 模拟多进程.c         # 进程管理调度
|    ├──── Memory_Module/           # 存储器管理模块
|    |    ├──── 存储器管理.c         # 存储器管理
|    |    └──── 虚拟存储器管理.h     # 进程分配内存
|    ├──── FileManage_Module/       # 文件管理模块
|    |    ├──── 磁盘调度算法.c       # 磁盘调度算法
|    |    └──── 读取文本.c           # 文件读取管理
|    ├──── Print.c           # 状态打印模块
```

```
│   └── Close.c          # 系统关闭模块
├── tests/               # 测试用例和代码
│   ├── test_main.c      # 主程序测试
│   ├── test_memory.c    # 内存管理模块测试
│   └── test_process.c   # 进程管理模块测试
└── Makefile             # 编译脚本
```

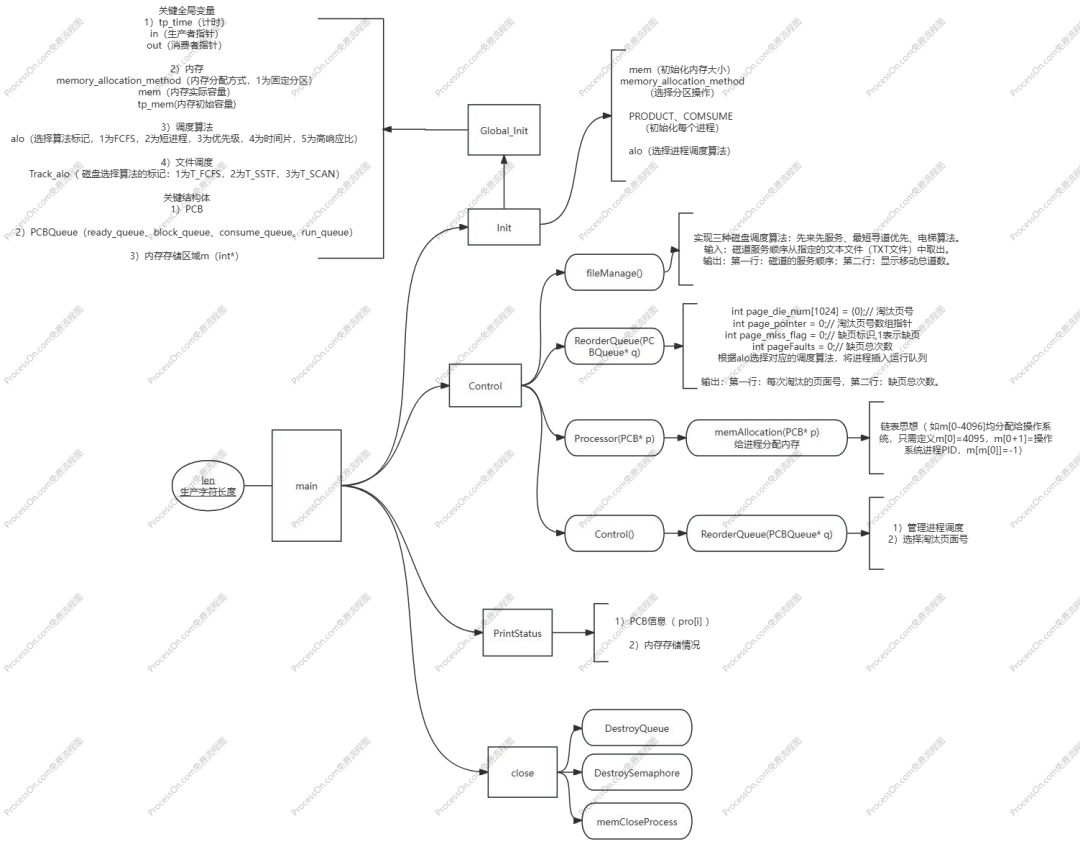


图 1 系统框图

3.2 各模块实现细节

3.2.1 进程管理模块 Process_Module

1 总体流程图

定义 PCB 结构体和 PCBQueue 队列结构体，初始化全局变量、PCB、就绪阻塞运行队列等，输入要消费的字符串，进入循环，选择调度算法（FCFS、SPF、HRRN、静态优先级），利用计时器与程序计数器以及 PV 操作开启模拟多进程生产消费，实现上述功能。

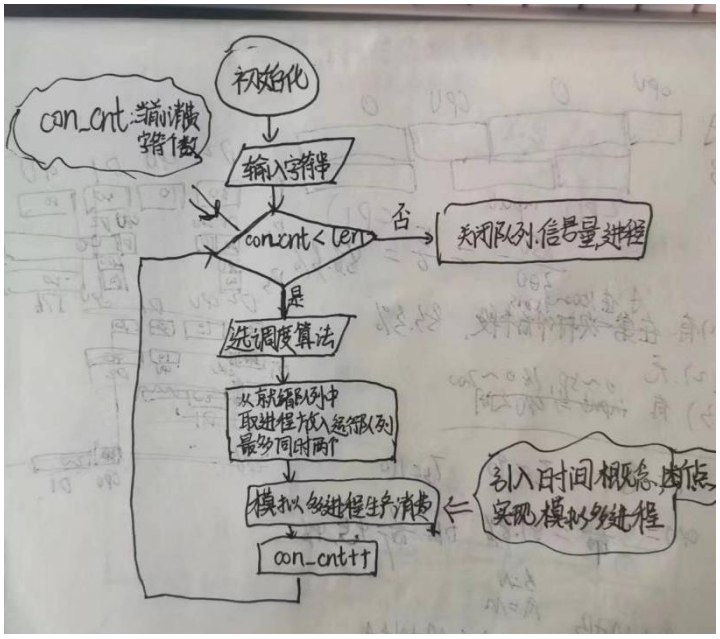


图 2 代码总体设计流程图

2 模拟多进程流程图

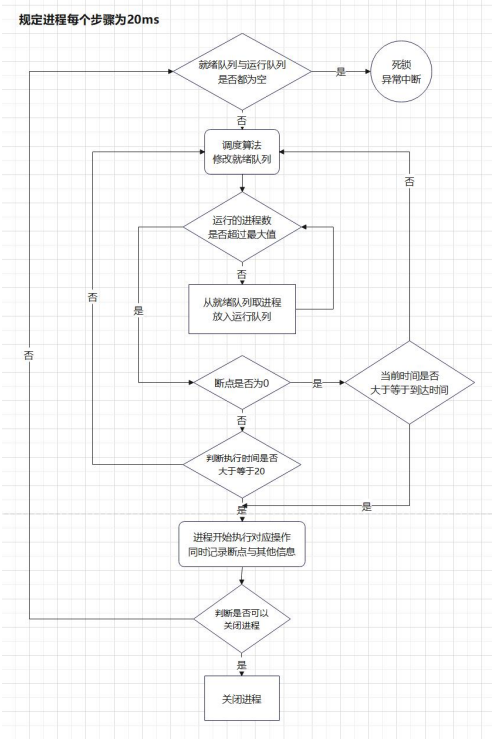


图 3 模拟多进程设计流程图

3 声明文件

```h
#pragma once

#define BUF 3           // 缓存的大小
#define MAX 30          // 最大可以输入的字符

/*------------------------------ 结构体 ------------------------------*/
typedef struct PCB {
    char name[10];      // 进程名
    int PID;            // 进程PID
    char state[10];     // 运行状态
    int arrive_time;    // 到达时间
    int priority;       // 优先级
    int start_time;     // 实际到达时间
    int running_time;   // 运行时间
    int end_time;       // 结束时间
    int copy_run_time;  // 用于时间片轮转
    int zztime;         // 周转时间
    double dqzztime;    // 带权周转时间
    char reason[10];    // 若阻塞，其原因
    int Address;        // 内存的位置
    int size;           // 进程大小

    int breakp;         // 断点保护
    int break_time;     // 断点时间

    int flag;           // 0为生产者，1为消费者
    struct PCB* next;   // 阻塞时的顺序
}PCB, Process;

typedef struct semaphore
{
    char name[10];
    int num;
    int flag;           // 0为empty，1为filled，2为rw_mux
}Semaphore;

typedef struct PCBQueue
{
    PCB* first_pro;
    PCB* last_pro;
    int size;
}PCBQueue;
```

```h
/*------------------------------ 全局变量 ------------------------------*/
extern int PRODUCT;                             // 生产者数量
extern int CONSUME;                             // 消费者数量
extern PCB pro[MAX];                            // 生产者进程 + 消费者进程
extern int PROCESS;                             // 进程最大并行数量
extern int pc;                                  // 程序计数器
extern int count;                               // 字符计数器
extern int con_cnt;                             // 消费计数器
extern int tp_time;                             // 计时
extern Semaphore* empty, * filled, * rw_mux;    // 初始化信号量

extern int len;                                 // 输入长度
extern int sp;                                  // string的指针
extern int in;                                  // 生产者指针
extern int out;                                 // 消费者指针
extern char temp;                               // 供打印的临时产品
extern char rec_p[MAX];                         // 生产记录
extern int p_f;                                 // 生产记录头指针
extern int p_l;                                 // 生产记录尾指针
extern char rec_c[MAX];                         // 消费记录
extern int c_f;                                 // 消费记录头指针
extern int c_l;                                 // 消费记录尾指针

extern PCBQueue* ready_queue;                   // 就绪队列
extern PCBQueue* block_queue;                   // 阻塞队列
extern PCBQueue* product_block_queue;           // 生产者阻塞队列
extern PCBQueue* consume_block_queue;           // 消费者阻塞队列
extern PCBQueue* run_queue;                     // 运行队列

/*------------------------------ 定义函数 ------------------------------*/
void Init();                                    // 初始化
void QueueInit(PCBQueue* q);                    // 初始化就绪队列
void SemaphoreInit();                           // 初始化信号量
void CopyProcess(PCB* pro1, PCB* pro2); // 将进程pro2的信息复制到进程pro1中
void EnQueue(PCBQueue* q, PCB* p);      // 插入队列
void Clean_Queue(PCBQueue* q);          // 清空队列
void ReorderQueue(PCBQueue* q);         // 重排队列
int BlockProCount();                    // 阻塞进程计数
PCB* DeQueue(PCBQueue* q);              // 输出队列
void P(Semaphore* mux, PCB* p);         // P操作
void V(Semaphore* mux, PCB* p);         // V操作
void Block(Semaphore* mux, PCB* p);     // 阻塞函数
void Wakeup(Semaphore* mux, PCB* p);    // 唤醒函数
void Control();                         // 处理机调度
void Processor(PCB* p);                 // 处理机执行
void PrintStatus();                     // 打印函数
void CloseProcess();                    // 关闭剩余进程
void DestroyQueue(PCBQueue* q);         // 释放队列
void Destroymaphore(Semaphore* mux);    // 释放信号量
```

图 4 模拟进程全局变量与函数

4 主要处理流程说明

参考图 2，将全局变量、PCB 队列、信号量、内存、进程等初始化后，进入循环（直到消费完所有字符后）选择调度算法将就绪队列重新排序，从就绪队列中取出 PCB 放入运行队列中，运行队列最大不能超过 PROCESS（最大并行数量），开始模拟多进程生产消费，每完成一个步骤就记录断点与时间。

在执行 PV 操作时，如果信号量的 num 值大于等于 0，则允许进程运行，反之则将进程从运行队列放入相应的阻塞队列，并修改其状态，等待其它进程释放信号量资源，该进程才能从阻塞态转换成就绪态

5 进程程并行设计说明

1）模拟多进程设计

参考图 3，首先判断是否会发生死锁，如果会，则中断。接着利用调度算法将就绪队列进行重新排序，判断运行的进程数是否超过 PROCESS，同时判断此时内存中是否有足够的空间存储下一个进程，如果都为否，则将就绪队列的头进程插入运行队列，反之则插入就绪队列队尾，之后执行运行队列中的进程（通过时间与断点模拟实现进程的并行）

检查该进程的断点，恢复进程上下文环境，根据断点执行相对应的操作，执行完操作后，判断进程是否可以关闭，如果是否，则继续执行第一步操作

```c
void Control() //处理器调度程序
{
    int num;                    // 就绪进程数目
    if (ready_queue->size == 0 && run_queue->size == 0)
    {
        // 彩色文本
        printf("\033[31m");
        printf("死锁!!!");
        printf("\033[0m");
        exit(0);
    }

    num = ready_queue->size;    // 统计就绪进程个数
    printf("\t* 就绪进程个数为:%d\n", num);
    //int num_run = run_queue->size;    // 统计就绪进程个数
    //printf("\t* 运行进程个数为:%d\n", num_run);
    //int num_block = block_queue->size;    // 统计就绪进程个数
    //printf("\t* 运行进程个数为:%d\n", num_run);

    // 每次 就绪队列重新排序
    ReorderQueue(ready_queue);

    // 插入运行队列，多进程并行最大数量为PROCESS
    PCB* p;
    for (int i = run_queue->size; i < PROCESS && num != 0; i++, num--)
    {
        p = DeQueue(ready_queue);
        strcpy(p->state, "Run");
        EnQueue(run_queue, p);
    }

    // 开启模拟多进程，执行操作
    for (PCB* p = run_queue->first_pro->next; p != NULL; p = p->next)
    {
        tp_time += 1;
        printf("当前时间: %ds\n", tp_time);

        // 检查断点，如果断点为0，则表示该进程刚开始执行，需特殊处理
        if (p->breakp == 0)
        {
            if (tp_time >= p->arrive_time)
            {
                // 彩色文本实现
                printf("\033[36m");
                Processor(p);
                printf("\033[0m");
            }
            else if(p->arrive_time - tp_time >= 20) // 到达时间过长，则特殊处理
            {
                // 如果运行队列内的每个进程的到达时间都很长，且就绪队列没有进程了，则特殊处理
                if(ready_queue->size == 0)
                {
                    p->arrive_time = tp_time;
                    // 彩色文本实现
                    printf("\033[36m");
                    Processor(p);
                    printf("\033[0m");
                }
                else
                {
                    printf("\t* 进程未到达，请等待!\n");
                    Look_DeQueue(run_queue, p);
                    strcpy(p->state, "Ready");
                    EnQueue(ready_queue, p);
                    EnQueue(ready_queue, DeQueue(ready_queue));
                }
            }
            else
            {
                printf("\t* 进程未到达，请等待!\n");
                Look_DeQueue(run_queue, p);
                strcpy(p->state, "Ready");
                EnQueue(ready_queue, p);
                EnQueue(ready_queue, DeQueue(ready_queue));
            }
        }
        else if (tp_time - p->break_time >= 2)      // 每步操作是20ms,执行不超过20ms就是没执行完毕
        {
            // 彩色文本实现
            printf("\033[36m");
            Processor(p);
            printf("\033[0m");
        }

        // 同步信息
        for (int k = 0; k < PRODUCT + CONSUME; k++)
        {
            if (pro[k].flag == p->flag && pro[k].PID == p->PID)
            {
                CopyProcess(&pro[k], p);
                break;
            }
        }

        // 判断是否已经输入完毕，是则关闭进程
        CloseProcess();
    }
}
```

图 5 模拟多进程并行处理器调度算法

2）模拟处理器指令执行

首先进行内存判断（内存可使用**固定或动态分区分配内存法**），如果内存没有足够空间存储进程，则将进程从运行态修改位就绪态；反之，则分配空间给进程，同时进程需要记录在内存的起始地址，接着执行进程的任务。（代码仅展示生产者部分）

```
void Processor(PCB* p)
{
    // 内存判断
    if (p->Address == -1)
    {
        if (memAllocation(p) != 1)
        {
            return;
        }
    }

    if (p->flag == 0) //当前进程为生产者
        switch (p->breakp)
        {
        case 0:     // produce
            p->break_time = tp_time;        // 保存时间断点

            if (count <= 0) {               // 生产结束
                Look_DeQueue(run_queue, p);
                printf("\t* 生产者%d结束生产!\n", p->PID);
                strcpy(p->state, "Stop");
                strcpy(p->reason, "Null");

                p->break_time = tp_time;
                p->end_time = p->break_time;// 保存结束时间
                p->breakp = -1;
                memClear(p);                // 内存中清除该进程
            }
            else
            {
                printf("\t* 开始运行%s %d\n", p->name, p->PID);
                p->start_time = p->break_time;// 记录开始时间
                printf("\t* %s %d生产了字符%c\n", p->name, p->PID, str[sp]);

                rec_p[p_l] = str[sp];       // 添加到生产记录
                p_l++;
                sp = (sp + 1) % len;
                count--;                    // 剩余字符个数减1

                pc++;                       // 程序计数器记录位置
                p->breakp++;                // 保存断点
            }

            break;
        case 1:     // 申请 访问临界区
            pc++;                           // 程序计数器记录位置
            p->breakp++;                    // 保存断点
            P(empty, p);

            break;
        case 2:
            // 互斥访问
            pc++;                           // 程序计数器记录位置
            p->breakp++;                    // 保存断点
            P(rw_mux, p);
            break;
        case 3:     // 访问临界区
            p->break_time = tp_time;    // 保存时间断点

            buffer[in] = rec_p[p_f];    // 放到缓冲区
            printf("\t* %c字符成功入驻空缓存!\n", buffer[in]);

            in = (in + 1) % BUF;        // 生产头指针+1
            p_f++;                      // 缓冲区尾指针+1
            pc++;                       // 程序计数器记录位置
            p->breakp++;                // 保存断点
            break;
        case 4:     // 退出临界区
            p->break_time = tp_time;    // 保存时间断点
            pc++;                       // 程序计数器记录位置
            p->breakp++;                // 保存断点

            V(rw_mux, p);
            printf("\t* 释放一个%s信号\n", rw_mux->name);
            V(filled, p);
            printf("\t* 释放一个%s信号\n", filled->name);

            break;
        case 5:     // goto01
            printf("\t* %s %d goto 0 操作\n", p->name, p->PID);
            pc = 0;                     // 程序计数器归零
            p->breakp = 0;
            Look_DeQueue(run_queue, p); // 从运行队列删除该进程
            printf("\t* 剩余字符count=%d个\n", count);

            p->break_time = tp_time;    // 保存时间断点
            p->end_time = p->break_time;// 保存结束时间
            p->zztime = p->end_time - p->start_time;        // 周转时间

            memClear(p);                // 内存中清除该进程

            if (count <= 0) {               // 生产结束
                printf("\t* 生产者%d结束生产!\n", p->PID);
                strcpy(p->state, "Stop");
                strcpy(p->reason, "Null");

                p->breakp = -1;
            }
            else
            {
                strcpy(p->state, "Ready");
                strcpy(p->reason, "Null");
                p->start_time = 0;
                EnQueue(ready_queue, p);
            }

        }
```

```
    else  //当前进程为消费者
        switch (p->breakp)
        {
        case 0:     // 判断缓冲区是否有东西
            printf("\t* 开始运行%s %d\n", p->name, p->PID);

            p->break_time = tp_time;        // 保存时间断点
            p->start_time = p->break_time;  // 记录开始时间
            pc++;                           // 程序计数器记录位置
            p->breakp++;

            P(filled, p);
            break;
        case 1:
            // 互斥访问
            pc++;                           // 程序计数器记录位置
            p->breakp++;                    // 保存断点
            P(rw_mux, p);
            break;
        case 2:     // 将字符取出缓冲区
            p->break_time = tp_time;    // 保存时间断点

            printf("\t* %s %c取字符!\n", p->name, p->PID);
            rec_c[c_l] = buffer[out];

            out = (out + 1) % BUF;      // 缓冲区头指针加一
            c_l++;                      // 输出尾指针+1
            pc++;                       // 程序计数器记录位置
            p->breakp++;                // 保存断点

            break;
        case 3:     // 释放信号量
            p->break_time = tp_time;    // 保存时间断点
            pc++;                       // 程序计数器记录位置
            p->breakp++;                // 保存断点

            V(rw_mux, p);
            printf("\t* 释放一个%s\n", rw_mux->name);
            V(empty, p);
            printf("\t* 释放一个%s\n", empty->name);

            break;
        case 4:     // 消费字符
            p->break_time = tp_time;        // 保存时间断点
            printf("\t* 消费了字符%c\n", rec_c[c_f]);          // 添加到消费记录
            c_f++;

            con_cnt++;
            if (con_cnt >= len)
            {
                strcpy(p->state, "Stop");   // 完成态
                p->breakp = -1;
                DeQueue(run_queue);

                p->break_time = tp_time;    // 保存时间断点
                p->end_time = p->break_time;// 保存结束时间
                p->zztime = p->end_time - p->start_time;        // 周转时间

                memClear(p);                // 内存中清除该进程

                return;
            }
```

```
        case 4:     // 消费字符
            p->break_time = tp_time;        // 保存时间断点
            printf("\t* 消费了字符%c\n", rec_c[c_f]);          // 添加到消费记录
            c_f++;

            con_cnt++;
            if (con_cnt >= len)
            {
                strcpy(p->state, "Stop");   // 完成态
                p->breakp = -1;
                DeQueue(run_queue);

                p->break_time = tp_time;    // 保存时间断点
                p->end_time = p->break_time;// 保存结束时间
                p->zztime = p->end_time - p->start_time;        // 周转时间

                memClear(p);                // 内存中清除该进程

                return;
            }

            pc++;                       // 程序计数器记录位置
            p->breakp++;                // 保存断点
            break;
        case 5:     // goto0
            p->break_time = tp_time;    // 保存时间断点
            printf("\t* 消费进程goto 0 操作\n");
            pc = 0;
            p->breakp = 0;              // 保存断点
            Look_DeQueue(run_queue, p);
            strcpy(p->state, "Ready");
            strcpy(p->reason, "NULL");
            p->start_time = 0;
            p->break_time = tp_time;    // 保存时间断点
            p->end_time = p->break_time;// 保存结束时间
            p->zztime = p->end_time - p->start_time;        // 周转时间

            memClear(p);                // 内存中清除该进程

            EnQueue(ready_queue, p);

            break;
        }

        p->dqzztime = (tp_time - p->start_time) / p->running_time;      // 带权周转时间
}
```

图 6 模拟进程执行函数

5 运行结果

```
*****************************
*开始初始化生产者和消费者进程...
*请依次输入进程的:到达时间       运行时间         优先级
*****************
*Producer 1       2
3
3
*****************

*****************
*Producer 2       12 20 3
*****************

*****************
*Consume 3        2 3 32
*****************

*****************
*Consume 4        2 12 32
*****************

*****************
*Consume 5        23 23 21
*****************

按FCFS调度算法重新排序!

-----------------------------------
按下任意键继续...
```

图 7 进程初始化

```
---------模拟指令流程--------
        * 就绪进程个数为:2
按FCFS调度算法重新排序!
当前时间: 11s
        * 释放一个rw_mux信号
        * Consume 3进程被唤醒了!
        * 释放一个filled信号
当前时间: 12s
进程2需要多个分区存储!
        * 开始运行Producer 2
        * Producer 2生产了字符h
---------生产者消费者模拟-------
* 模拟过程的字符串为:   chenPi20221070213
* 已生产:h
* 空缓存:c
* 已消费:
-------进程控制块的信息--------
进程名          状态    开始时间        运行时间        高响应比        优先级  等待原因        断点
生产者1         Run           3              3        2.000              3  Null            5
生产者2         Run          12             20        0.000              3  Null            1
消费者3         Ready         2              3        0.000             32  Null            1
消费者4         Block         4             12        0.000             32  filled          1
消费者5         Ready        23             23        0.000             21  Null            0
--------------------------------
```

图 8 FCFS-进程管理信息

```
---------模拟指令流程--------
        * 就绪进程个数为:5
按SPF调度算法重新排序!
当前时间: 31s
进程1需要多个分区存储!
        * 开始运行Producer 1
        * Producer 1生产了字符P
当前时间: 32s
进程2需要多个分区存储!
        * rw_mux信号申请成功!
---------生产者消费者模拟-------
* 模拟过程的字符串为:   chenPi20221070213
* 已生产:nP
* 空缓存:he
* 已消费:c
-------进程控制块的信息--------
进程名          状态    开始时间        运行时间        高响应比        优先级  等待原因        断点
生产者1         Run          31              3        0.000              4  Null            1
生产者2         Run          21              6        1.000              7  Null            3
消费者3         Ready         0              9        3.000              8  NULL            0
消费者4         Ready        13             12        0.000             11  Null            0
消费者5         Ready         9             18        0.000             20  Null            0
--------------------------------
```

图 9 SPF-进程管理信息

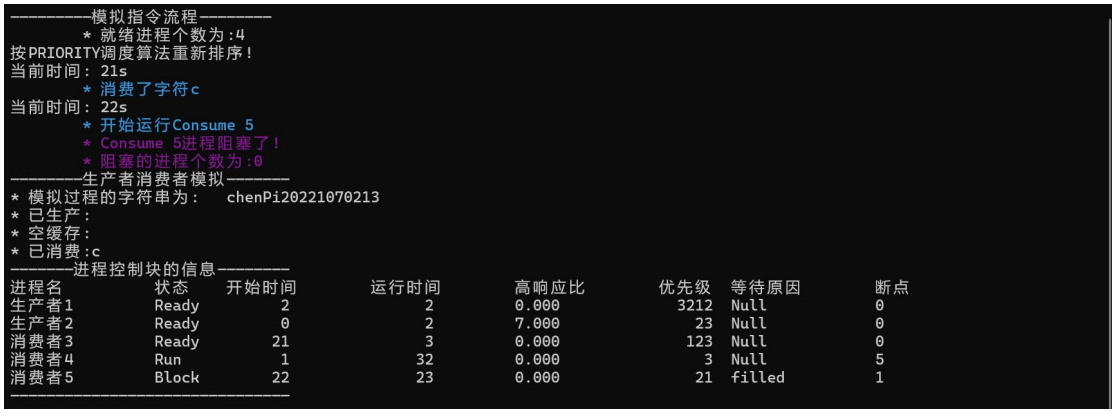


图 10 PRIORITY-进程管理信息

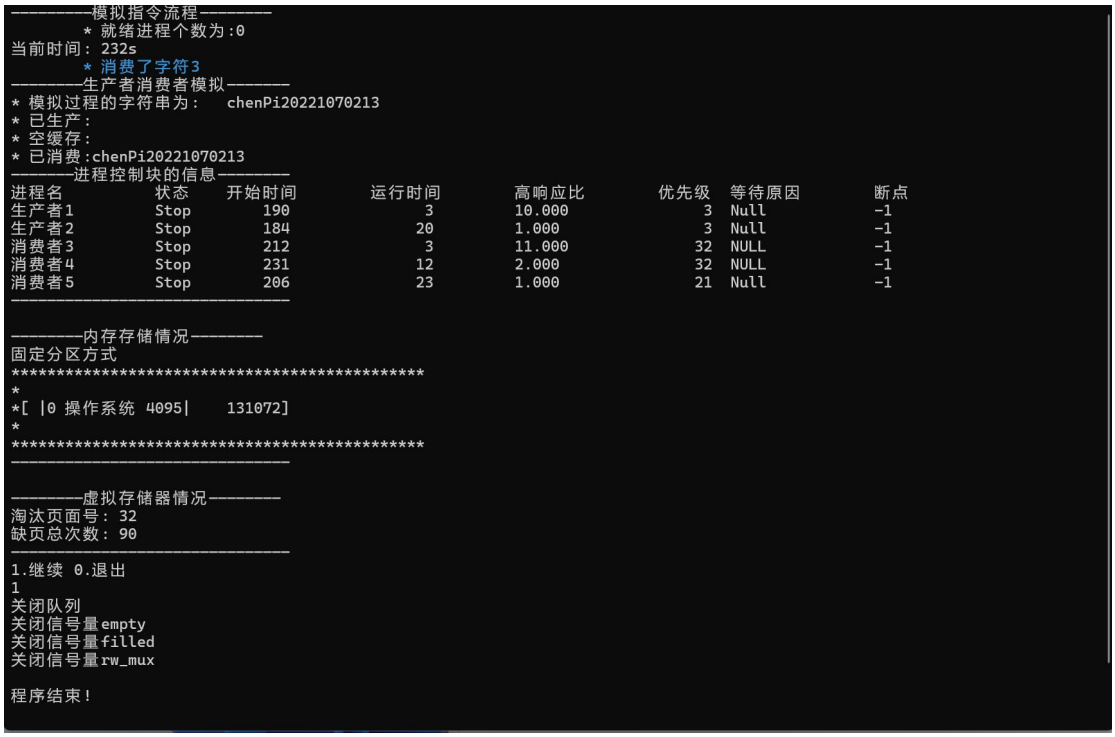


图 11 运行结束信息

3.2.2 存储器管理模块 Memory_Module

1 声明文件

```
#define MMAX 128       // 默认内存大小，K
#define MPAGE 1024         // 最大页面号

/*------------------------------ 全局变量 ------------------------------*/
extern int mem;                                 // 内存
extern int tp_mem;                              // 临时内存
extern int* m;                                  // 内存数组
extern int memory_allocation_method;// 内存分配方式，1 为固定分区方式，2 为可变分区方式
```

```
extern int alo;                                  // 选择进程调度算法的标记：1 为 FCFS，2 为短进程，3 为优先级，4 为时间片，5 为高响应比
extern int page_die_num[MPAGE];                  // 淘汰页号
extern int page_pointer;                 // 淘汰页号数组指针
extern int page_miss_flag;               // 缺页标识，1 表示缺页
extern int pageFaults;                   // 缺页总次数

/*------------------------------ 定义函数 ------------------------------*/
void setMem();                                   // 初始化内存
int memAllocation(PCB* p);               // 内存是否有足够空间
void memClear(PCB* p);                   // 内存中清除指定进程
void memCloseProcess();                          // 内存清空

void FCFS(PCB** arr, int n);             // 先进先出调度算法
void SPF(PCB** arr, int n);                      // 非抢占式短进程优先调度算法
void PRIORITY(PCB** arr, int n);         // 静态优先级调度算法
void HRRN(PCB** arr, int n);             // 高响应比优先调度算法
void Alogorithm();                               // 算法
```

2 存储器管理设计思想

1）模拟分配内存思想

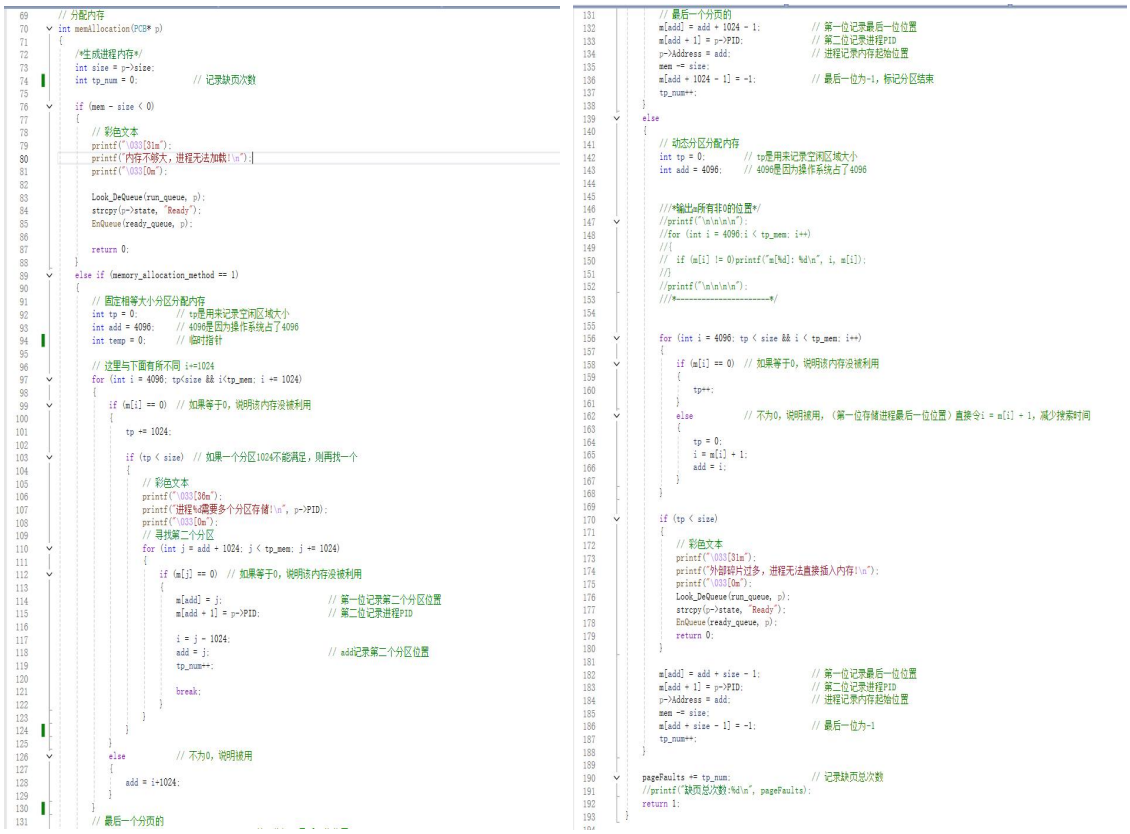
内存用 int*类型定义，旨在可拓展内存空间和利用链表思想，减少操作流程，提高存储效率。

链表思想：在内存中只存储进程的在内存的每个分页的首地址以及最后一个分页的尾地址，尾地址的内存存储-1 标识这是进程的最后一个分页；进程存储在内存的首地址。（ 如每个分页大小为 1024，m[0-4095]均分配给操作系统，只需定义 m[0]=1024，m[0+1]=操作系统进程 PID，m[1024]=2048，m[2048]=3072，m[3072]=4095，m[4095]=-1）

2）模拟虚拟存储器管理思想

设计虚拟存储器管理模块，实现三种页面调度算法：FIFO、最近最少使用调度算法（LRU）、最近最不常用调度算法（LFU）。

根据 page_scheduling_alo 选择对应的页面调度算法，将进程插入运行队列，欲输出的缺页数关注进程分配内存函数，因为为它分配多少页，等于它缺多少页；而淘汰页面号 关注 memClear 函数，设计全局变量 pageFaults 记录缺页总次数



```
// 分配内存
int memAllocation(PCB* p)
{
    /*生成进程内存*/
    int size = p->size;
    int tp_num = 0;             // 记录缺页次数

    if (mem - size < 0)
    {
        // 彩色文本
        printf("\033[31m");
        printf("内存不够大，进程无法加载!\n");
        printf("\033[0m");

        Look_DeQueue(run_queue, p);
        strcpy(p->state, "Ready");
        EnQueue(ready_queue, p);

        return 0;
    }
    else if (memory_allocation_method == 1)
    {
        // 固定相等大小分区分配内存
        int tp = 0;             // tp是用来记录空闲区域大小
        int add = 4096;     // 4096是因为操作系统占了4096
        int temp = 0;       // 临时指针

        // 这里与下面有所不同 i+=1024
        for (int i = 4096; tp<size && i<tp_mem; i += 1024)
        {
            if (m[i] == 0)  // 如果等于0，说明该内存没被利用
            {
                tp += 1024;

                if (tp < size)  // 如果一个分区1024不能满足，则再找一个
                {
                    // 彩色文本
                    printf("\033[36m");
                    printf("进程%d需要多个分区存储!\n", p->PID);
                    printf("\033[0m");
                    // 寻找第二个分区
                    for (int j = add + 1024; j < tp_mem; j += 1024)
                    {
                        if (m[j] == 0)  // 如果等于0，说明该内存没被利用
                        {
                            m[add] = j;                 // 第一位记录第二个分区位置
                            m[add + 1] = p->PID;        // 第二位记录进程PID

                            i = j - 1024;
                            add = j;                    // add记录第二个分区位置
                            tp_num++;

                            break;
                        }
                    }
                }
            }
            else            // 不为0，说明被用
            {
                add = i+1024;
            }
        }
        // 最后一个分页的
        m[add] = add + 1024 - 1;            // 第一位记录最后一位位置
        m[add + 1] = p->PID;                // 第二位记录进程PID
        p->Address = add;                   // 进程记录内存起始位置
        mem -= size;
        m[add + 1024 - 1] = -1;             // 最后一位为-1，标记分区结束
        tp_num++;
    }
    else
    {
        // 动态分区分配内存
        int tp = 0;             // tp是用来记录空闲区域大小
        int add = 4096;     // 4096是因为操作系统占了4096


        ///*输出m所有非0的位置*/
        //printf("\n\n\n\n");
        //for (int i = 4096;i < tp_mem; i++)
        //{
        //  if (m[i] != 0)printf("m[%d]: %d\n", i, m[i]);
        //}
        //printf("\n\n\n\n");
        ///*----------------------*/


        for (int i = 4096; tp < size && i < tp_mem; i++)
        {
            if (m[i] == 0)  // 如果等于0，说明该内存没被利用
            {
                tp++;
            }
            else            // 不为0，说明被用，（第一位存储进程最后一位位置）直接令i = m[i] + 1，减少搜索时间
            {
                tp = 0;
                i = m[i] + 1;
                add = i;
            }
        }

        if (tp < size)
        {
            // 彩色文本
            printf("\033[31m");
            printf("外部碎片过多，进程无法直接插入内存!\n");
            printf("\033[0m");
            Look_DeQueue(run_queue, p);
            strcpy(p->state, "Ready");
            EnQueue(ready_queue, p);
            return 0;
        }

        m[add] = add + size - 1;            // 第一位记录最后一位位置
        m[add + 1] = p->PID;                // 第二位记录进程PID
        p->Address = add;                   // 进程记录内存起始位置
        mem -= size;
        m[add + size - 1] = -1;             // 最后一位为-1
        tp_num++;
    }

    pageFaults += tp_num;                   // 记录缺页总次数
    //printf("缺页总次数:%d\n", pageFaults);
    return 1;
}
```

图 12 存储器分配内存及缺页计算

```
void memClear(PCB* p)
{
    page_miss_flag = 1;
    //printf("淘汰的页面号为：%d\n", p->Address / 1024);
    int temp_page_pointer;          // 临时页面号数组指针

    int temp = m[p->Address];
    while (m[temp] != -1)
    {
        temp_page_pointer = 1;
        // 记录淘汰页面号
        page_die_num[page_pointer + temp_page_pointer] = m[temp] / 1024;
        temp_page_pointer++;

        int temp_temp = m[temp];
        m[temp+1] = 0;
        m[temp] = 0;
        temp = temp_temp;
    }
    temp_page_pointer = 0;
    page_die_num[page_pointer + temp_page_pointer] = m[p->Address] / 1024;
    m[p->Address] = 0;
    m[p->Address + 1] = 0;
    m[temp] = 0;
    p->Address = -1;
    //mem += sizeof(*p);
    mem += p->size;
}
```

图 13 存储器淘汰页面

3 运行结果图

```
--------------------------------------
正在初始化内存...

请输入内存的大小(K):
128
*****************************
*请选择内存管理方式:
*       1: 固定分区
*       2: 可变分区
*****************************
1
*****************************
*请选择以下的一种进程调度算法:
*       1: 先来先服务调度
*       2: 非抢占式短进程优先调度
*       3: 静态优先级调度
*       4: 高响应比调度
*****************************
1

--------------------------------------
按下任意键继续...
```

图 14 内存初始化

```
--------内存存储情况--------
固定分区方式
*********************************************
*
*[ |0 操作系统 4095| |4096 进程1 5120| |5120 进程1 6143| |6144 进程2 7168| |7168 进程2 8191|    131072]
*
*********************************************
--------------------------------
```

```
---------模拟指令流程---------
        * 就绪进程个数为:2
按FCFS调度算法重新排序!
当前时间: 31s
        * 释放一个rw_mux
        * Producer 2进程被唤醒了!
        * 释放一个empty
当前时间: 32s
        * Producer 1进程阻塞了!
        * 阻塞的进程个数为:0
---------生产者消费者模拟-------
* 模拟过程的字符串为:   chenPi20221070213
* 已生产:nP
* 空缓存:he
* 已消费:c
--------进程控制块的信息---------
进程名          状态    开始时间        运行时间        高响应比        优先级  等待原因        断点
生产者1         Block       30              3       0.000               3  empty         2
生产者2         Ready       24             20       0.000               3  Null          2
消费者3         Run          2              3       9.000              32  Null          4
消费者4         Ready        4             12       0.000              32  Null          1
消费者5         Ready       23             23       0.000              21  Null          0
--------------------------------

---------内存存储情况---------
固定分区方式
*********************************************
*
*[ |0 操作系统 4095| |4096 进程1 5120| |5120 进程1 8192| |6144 进程2 7168| |7168 进程2 9216| |8192 进程3 9215| |9216 进
程1 10240|      131072]
*
*********************************************
--------------------------------

--------虚拟存储器情况--------
淘汰页面号: 10
缺页总次数: 13
--------------------------------
1.继续 0.退出
1
```

```
--------模拟指令流程--------
        * 就绪进程个数为:0
当前时间: 232s
        * 消费了字符3
--------生产者消费者模拟--------
* 模拟过程的字符串为:   chenPi20221070213
* 已生产:
* 空缓存:
* 已消费:chenPi20221070213
-------进程控制块的信息--------
进程名          状态    开始时间        运行时间        高响应比        优先级  等待原因        断点
生产者1         Stop        190             3       10.000              3  Null          -1
生产者2         Stop        184            20        1.000              3  Null          -1
消费者3         Stop        212             3       11.000             32  NULL          -1
消费者4         Stop        231            12        2.000             32  NULL          -1
消费者5         Stop        206            23        1.000             21  Null          -1
--------------------------------

--------内存存储情况--------
固定分区方式
**********************************************
*
*[ |0 操作系统 4095|    131072]
*
**********************************************
--------------------------------

--------虚拟存储器情况--------
淘汰页面号: 32
缺页总次数: 90
--------------------------------
1.继续 0.退出
1
关闭队列
关闭信号量empty
关闭信号量filled
关闭信号量rw_mux

程序结束!
```

图 15 固定分区方式

```
--------内存存储情况--------
可变分区方式
**********************************************
*
*[ |0 操作系统 4095| |4096 进程2 6719|  131072]
*
**********************************************
--------------------------------

--------虚拟存储器情况--------
淘汰页面号: 9
缺页总次数: 61
--------------------------------
```

```
--------模拟指令流程--------
        * 就绪进程个数为:2
按PRIORITY调度算法重新排序!
当前时间: 220s
        * Producer 2 goto 0 操作
        * 剩余字符count=1个
当前时间: 221s
        * rw_mux信号申请成功!
--------生产者消费者模拟--------
* 模拟过程的字符串为:   chenPi20221070213
* 已生产:1
* 空缓存:2
* 已消费:chenPi20221070
-------进程控制块的信息--------
进程名          状态    开始时间        运行时间        高响应比        优先级  等待原因        断点
生产者1         Ready       191             2       0.000            3212  Null          2
生产者2         Ready         0             2     110.000              23  Null          0
消费者3         Block       219             3       0.000             123  filled        1
消费者4         Block       217            32       0.000               3  filled        1
消费者5         Run         211            23       0.000              21  Null          2
--------------------------------

--------内存存储情况--------
可变分区方式
**********************************************
*
*[ |0 操作系统 4095| |4096 进程5 5824|  131072]
*
**********************************************
--------------------------------

--------虚拟存储器情况--------
淘汰页面号: 9
缺页总次数: 71
--------------------------------
1.继续 0.退出
```

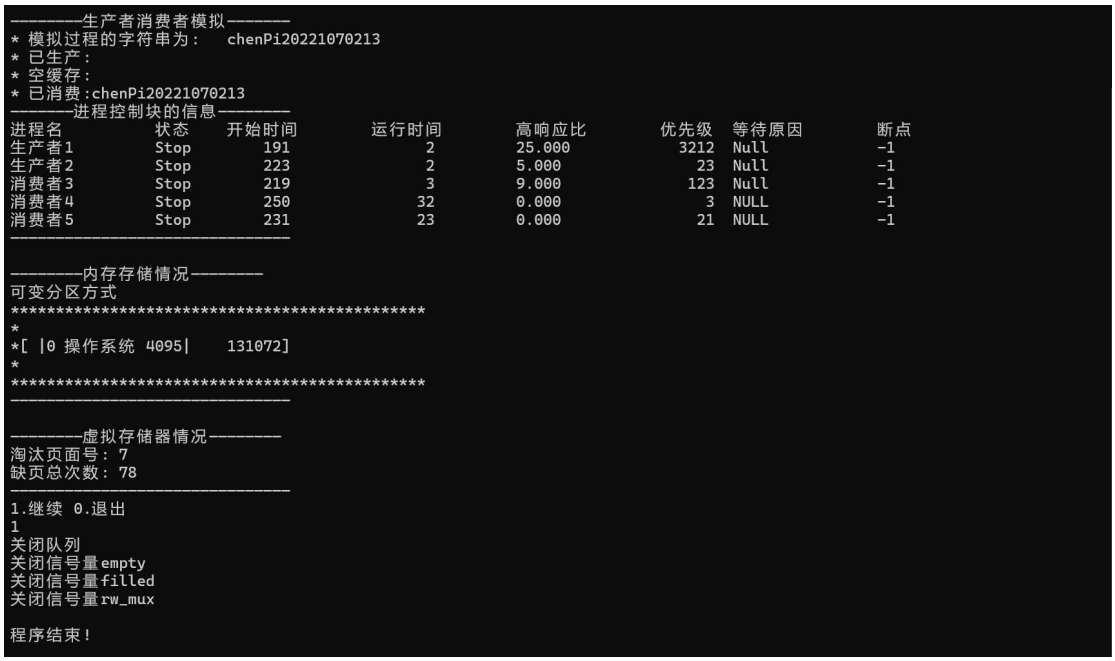


图 16 可变分区方式

图 1 虚拟存储器缺页情况

3.2.3 文件管理模块 FileManage_Module

1 声明文件

```
/*------------------------------ 全局变量 ------------------------------*/
extern int Track_alo;                    // 磁盘选择算法的标记：1 为 T_FCFS，2 为 T_SSTF，3 为 T_SCAN
extern int serviceOrder[MAX_TRACKS];     // 存储磁盘调度算法中的服务顺序
extern int totalTracks;                      // 记录当前 serviceOrder 数组中已经存储的磁道数量
/*------------------------------ 定义函数 ------------------------------*/
void fileManage();                           // 磁盘调度
const char* readFile();                      // 读取文件
void T_FCFS(int requests[], int numRequests);
void T_SSTF(int requests[], int numRequests, int currentTrack);
void T_SCAN(int requests[], int numRequests, int currentTrack, int direction);
```

2 模拟磁盘调度设计思想

目的是初始化预生产的字符或字符串，字符串将由**模拟操作系统**从文件（预生产字符串.txt）中读取。该文件存放在**模拟磁盘（二维数组）**上，**模拟磁盘需要该文件同时存储其在磁盘上的位置（柱面号和磁道号）**，模拟操作系统会根据其所在位置获得**初设磁道服务顺序**，接着按用户选择的磁道调度算法（T_FCFS，T_SSTF，TSCAN）**修改磁道服务顺序**读取文件，从而初始化字符串；

设计全局变量 Track_alo 记录选择的算法，serviceOrder[MAX_TRACKS]存储磁盘调度算法中的服务顺序，totalTracks 记录当前 serviceOrder 数组中已经存储的磁道数量

```
void fileManage()
{
    int requests[MAX_TRACKS];
    int numRequests;

    readRequests("..\\data\\磁道服务顺序.txt", requests, &numRequests);

    printf("\n*****************************\n*请选择磁道调度算法:\n");
    printf("*\t1: 先来先服务(T_FCF)\n");
    printf("*\t2: 最短寻道优先(T_SSTF)\n");
    printf("*\t3: 电梯算法(T_SCAN)\n");
    printf("*****************************\n");

    scanf("%d", &Track_alo);
    getchar();
    switch (Track_alo)
    {
    case 1:
        printf("\n*****************************\n*按 FCFS 磁道调度算法重新排序!\n");
        T_FCFS(requests, numRequests);
       printf("*****************************\n");
        break;
    case 2:
        printf("\n*****************************\n*按 SSTF 次磁道调度算法重新排序!\n");
        T_SSTF(requests, numRequests, 0);
       printf("*****************************\n");
        break;
    case 3:
        printf("\n*****************************\n*按 SCAN 磁道调度算法重新排序!\n");
       int tp_direction = 0;
       printf("****************\n*选择移动方向: \n*0: 低轨道\n*1: 高轨道\n");
       printf("****************\n");
       scanf("%d", &tp_direction);
       getchar();
```

```
            T_SCAN(requests, numRequests, 0, tp_direction);
            printf("*****************************\n");
            break;
        default:
            printf("\n*****************************\n*按 FCFS 磁道调度算法重新排序!\n");
            T_FCFS(requests, numRequests);
            printf("*****************************\n");
            break;
        }

    return 0;
}
```

3 运行结果

```
*生产者消费者模拟
---------------------------------
正在从磁盘文件中读取预生产的字符...

*****************************
*请选择磁道调度算法:
*       1: 先来先服务(T_FCF)
*       2: 最短寻道优先(T_SSTF)
*       3: 电梯算法(T_SCAN)
*****************************
1

*****************************
*按FCFS磁道调度算法重新排序!
*FCFS Service Order: 10 23 11 15 4 16 9 5 2 19
*Total Movement: 93
*****************************

预生产字符: chenPi20221070213

---------------------------------
按下任意键继续...
```

图 13 磁道调度 FCFS

```
*生产者消费者模拟
-----------------------------------
正在从磁盘文件中读取预生产的字符...

*****************************
*请选择磁道调度算法:
*       1: 先来先服务(T_FCF)
*       2: 最短寻道优先(T_SSTF)
*       3: 电梯算法(T_SCAN)
*****************************
2

*****************************
*按SSTF次磁道调度算法重新排序!
*SSTF Service Order: 2 4 5 9 10 11 15 16 19 23
*Total Movement: 23
*****************************

预生产字符: chenPi20221070213

-----------------------------------
按下任意键继续...
```

图 14 磁道调度 SSTF

```
*生产者消费者模拟
-----------------------------------
正在从磁盘文件中读取预生产的字符...

*****************************
*请选择磁道调度算法:
*       1: 先来先服务(T_FCF)
*       2: 最短寻道优先(T_SSTF)
*       3: 电梯算法(T_SCAN)
*****************************
3

*****************************
*按SCAN磁道调度算法重新排序!
*****************
*选择移动方向:
*0: 低轨道
*1: 高轨道
*****************
1
*SCAN Service Order: 2 4 5 9 10 11 15 16 19 23
*Total Movement: 23
*****************************

预生产字符: chenPi20221070213

-----------------------------------
按下任意键继续...
```

图 15 磁道调度 SCAN

四、测试用例及测试结果

4.1 设计测试用例

图 16 设计 4 个测试用例

```
---------模拟指令流程---------
        * 就绪进程个数为:0
当前时间: 252s
        * 消费进程goto 0 操作
当前时间: 253s
        * 消费了字符3
--------生产者消费者模拟--------
* 模拟过程的字符串为:   chenPi20221070213
* 已生产:
* 空缓存:
* 已消费:chenPi20221070213
-------进程控制块的信息--------
进程名          状态    开始时间        运行时间        高响应比        优先级  等待原因        断点
生产者1         Stop        223             3       6.000               4  Null            -1
生产者2         Stop        208             6       4.000               7  Null            -1
消费者3         Stop          0             9       4.000               8  NULL            -1
消费者4         Stop        229            12       1.000              11  Null            -1
消费者5         Stop        242            18       2.000              20  NULL            -1
--------------------------------

--------内存存储情况--------
固定分区方式
**********************************************
*
*[ |0 操作系统 4095|    131072]
*
**********************************************
--------------------------------

--------虚拟存储器情况--------
淘汰页面号: 38
缺页总次数: 117
--------------------------------
1.继续 0.退出
1
关闭队列
关闭信号量empty
关闭信号量filled
关闭信号量rw_mux

程序结束!
```

图 17 测试结果 T_FCFS-固定-PRI

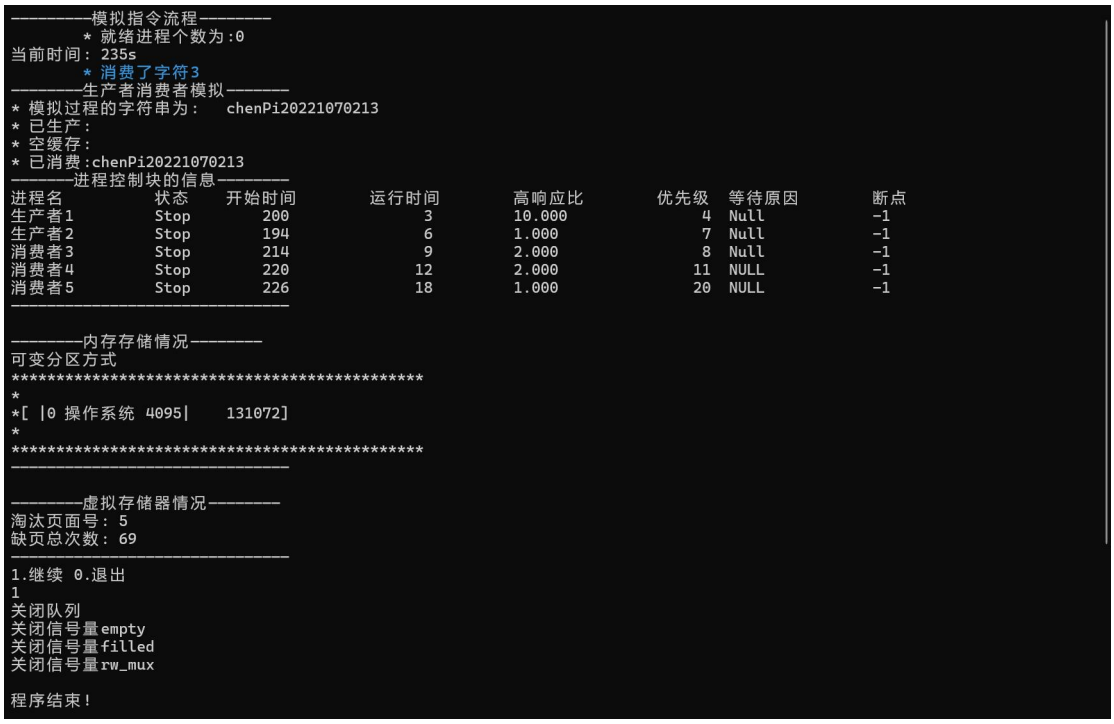


图 18 测试结果 T_SCAN0-可变-FCFS

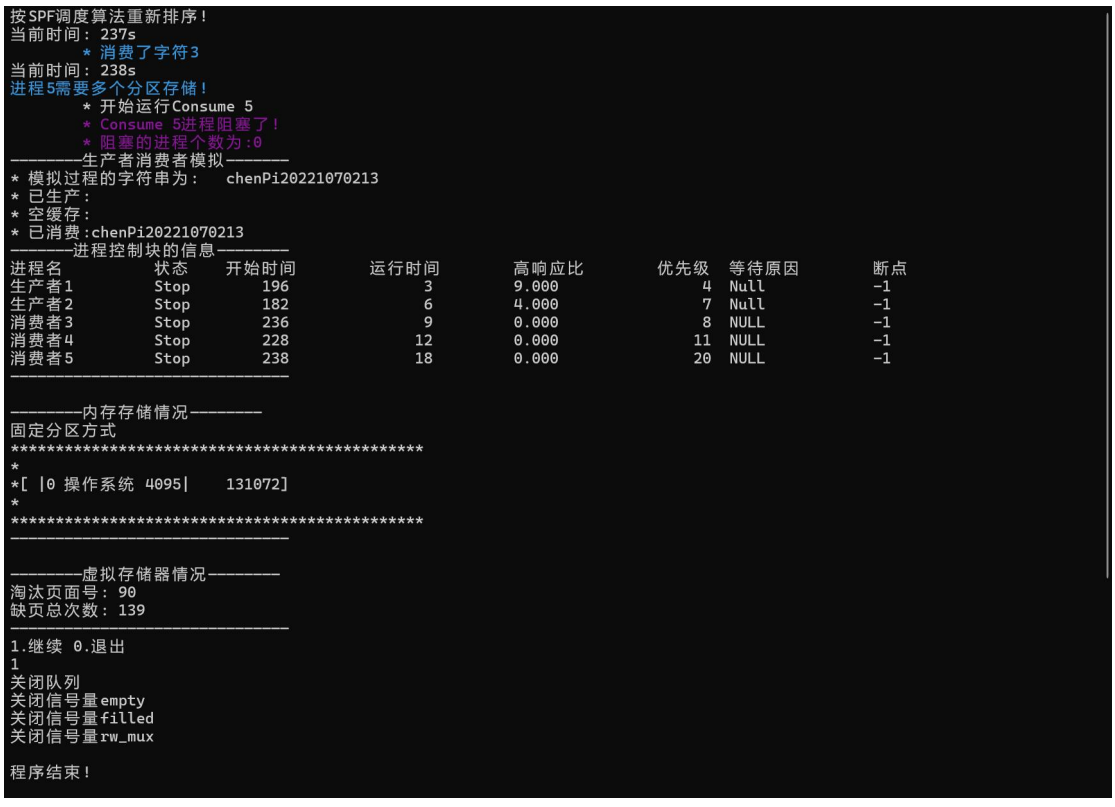


图 19 测试结果 T_SCAN1-固定-SPF

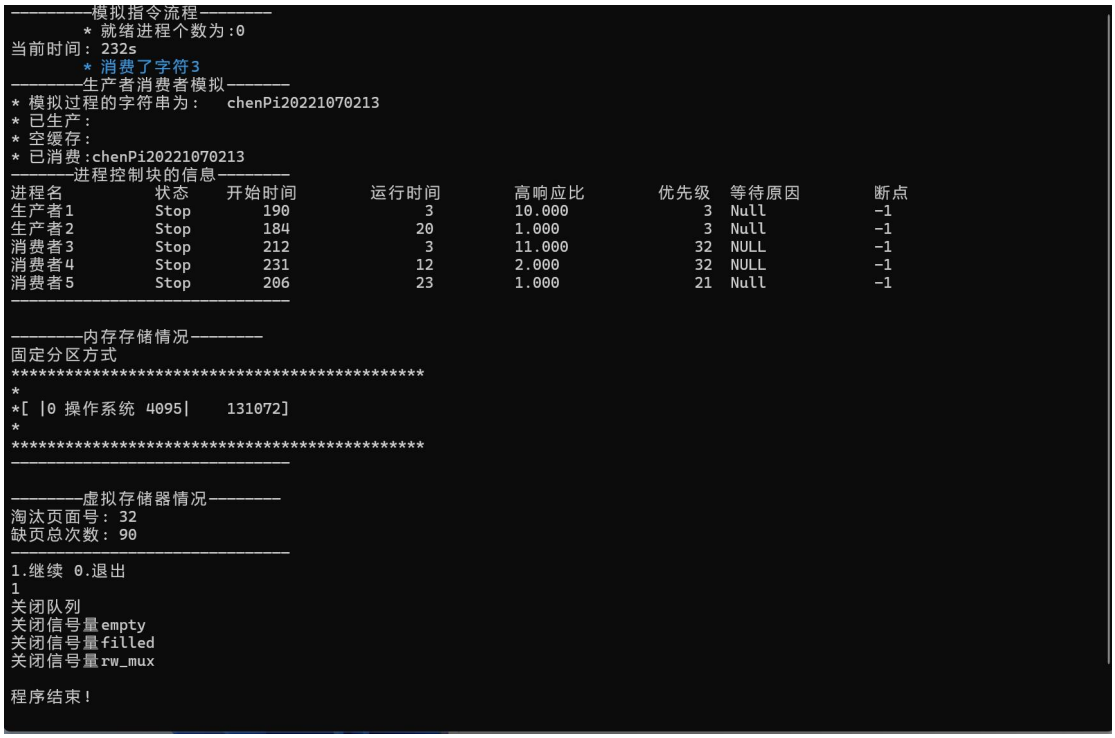


图 20 测试结果 T_SSTF-可变-高响应比

五、总结与心得

本次实验设计与实现了操作系统中的四个核心模块，包括进程管理、存储器管理、虚拟存储器管理和文件管理模块。通过完成实验，全面加深了对操作系统关键功能模块的理解，提升了编程实践和问题解决能力。以下是对各模块的总结：

1 进程管理模块

该模块模拟了操作系统的进程调度与控制功能，支持多个进程并行运行。通过设计进程控制块（PCB）实现了对进程状态、优先级和资源信息的记录与管理。实验成功实现了多种调度算法（如 FCFS、SJF、优先级调度等），并通过运行状态的动态输出观察到进程的切换与调度过程。实验帮助理解了调度算法的核心逻辑以及进程同步和资源竞争的基本原理，同时学会了如何通过编程实现进程调度系统。

2 存储器管理模块

该模块实现了模拟内存管理中的两种分区分配方式：固定分区和可变分区。在固定分区模式下，分区表显示了内存分配情况；在可变分区模式下，通过首次适应和最佳适应算法实现了动态分区分配，同时使用分配表和未分配表展示内存

使用情况。实验通过菜单化的程序实现了内存分配和释放功能，并验证了不同分配策略的优缺点。通过实验，理解了内存分配算法在解决内部碎片与外部碎片问题中的作用，掌握了动态内存管理的基本思想。

3 虚拟存储器管理模块

该模块模拟了虚拟存储器的页面置换过程，设计并实现了 FIFO、LRU 和 LFU 三种页面调度算法。通过读取页面访问序列的输入文件，模拟了内存页的动态替换过程，输出了页面淘汰顺序和缺页次数。实验展现了不同调度算法在处理页面置换问题时的优缺点，尤其是 FIFO 算法可能出现“Belady 异常”，而 LRU 和 LFU 算法较好地利用了访问局部性。通过实验进一步理解了虚拟存储器的工作机制和置换算法的实现细节。

4 文件管理模块

该模块实现了三种磁盘调度算法：先来先服务（FCFS）、最短寻道优先（SSTF）和电梯算法（SCAN）。实验通过模拟磁盘请求序列的读取和处理，成功输出了磁头的服务顺序及移动总道数。通过比较不同算法的性能，发现 SSTF 能够减少磁头移动距离，而 SCAN 适合实际应用场景。实验加深了对磁盘调度算法的理解，掌握了如何通过算法优化提高磁盘 I/O 效率。

本次实验从操作系统原理出发，通过程序实现进一步巩固了对进程调度、内存管理、虚拟存储器和磁盘调度等核心概念的理解。通过模块化设计与实现，不仅掌握了多种调度算法的编程技巧，还提高了代码组织和问题调试能力。

参考文献

[1]王道论坛. 计算机操作系统考研复习指导[M]. 北京: 电子工业出版社, 2021.

[2]周东华. 操作系统原理[M]. 北京: 清华大学出版社, 2015.

[3]尹宝才. 现代操作系统教程[M]. 北京: 高等教育出版社, 2018.