操 作 系 统

2022 年春期末考核报告

姓　　名: 武子强

学　　号: 195132

学　　院: 人工智能与数据科学学院

班　　级: 物联 NZ191

评阅记录:

成　　绩:

2022 年 7 月 13 日

一、 结合存储管理理论知识，对 Linux 操作系统的详细阐述。

1. 地址映射

Linux 的地址映射主要体现在通过内存映射将虚拟内存地址映射到物理内存地址，对实际使用虚拟内存并分配的物理内存进行管理。这之间主要就是逻辑地址、虚拟地址、线性地址和物理地址这四个地址之间的转化关系。

以 Intel x86 架构的 CPU 为例来解释这四个概念

- 逻辑地址：是由段选择符和指定段内相对地址偏移量组成的。
- 虚拟地址：是逻辑地址的指定段内相对地址偏移量。
- 线性地址：逻辑地址经过段页式的转化所得到的平坦的统一地址空间。
- 物理地址：在 x86 cpu 中，这里的物理地址是物理内存块的编号。（物理内存的大小和起始地址是固定的，所以通过存储的物理内存块编号可以得出对应内存块的起始地址）

可以看出 x86 cpu 采用的是段页式内存管理机制。所以它们之间的关系是：虚拟地址加上段标识符共同构成逻辑地址，逻辑地址通过段式内存管理转换得到线性地址，然后线性地址通过页式内存管理得到物理地址。

而对于采用页式内存管理的 Linux 内核来说，不需要段映射，于是 Linux 内核将段基址都设置为 0，通过使所有的段都重合来实现不分段。这样一来，逻辑地址也就简化为了段内的偏移量，段基地址变为了 0。那么可以说对于 x86 linux 内核来讲，题目中所说的地址映射实际上就是指的上面提到的逻辑地址映射成为物理地址的过程。

而在 Linux 储存管理的过程中，Linux 内存管理程序通过 MMU 或 TLB 将用户的逻辑地址转换为物理地址。如果发生页面交换，也要即使更改页面相关内容以确保地址映射正确。并且虚拟内存选择采取动态地址映射方式，将地址空间存储空间的对应关系在程序的执行过程中通过访问 MMU 实现。这意味着当发生映射时，可执行映射的内容并不需要调入物理内存，而是放入该进程的虚拟内存

区，当硬盘中的内容被需要或者现在需要动态申请内存时，Linux 通过缺页操作触发内存映射，供程序使用。

从映射机制来说，对于 32 位的计算机一般采用两级页表（两级地址映射机制），但为了兼容 64 位的 CPU，Linux 在页目录以及页号中间新加了一层中间目录，构成三级地址映射机制（PGD、PMD、PT）。系统中的虚拟地址也从页目录、页号和偏移量三部分组成变成了页目录、页号、中间目录和偏移量四部分。而具体的映射过程也多了一步：由虚拟地址的次位段为下标，在 PMD 中找到指向 PT 的指针。

2. 虚拟内存实现机制

对于虚拟内存的实现机制，实际上就是每个进程都维护单独的页表，这些页表存放着每个虚拟页的状态、是否映射、是否缓存等等。结合数据结构来谈就是 linux 把虚拟内存划分为区域的集合，每个区域包含连续的多个页。内核给每一个进程单独维护 task_struct，它的 mm 指针，指向 mm_struct，来描述虚拟内存的运行状态。而 mm_struct 的 pgd 指针指向进程的一级页表的基地址。mmap 指针，指向 vm_area_struct 链表（描述 area 的结构，vm_start 和 vm_end 表示 area 的位置，vm_prot 表示 area 内的页的读写权限，vm_flags 表示 area 内的页面是进程私有还是共享）vm_next 指向下一个 area 节点，最终可以访问到该进程所有的虚拟内存区。

3. 请求分页实现过程

我们知道请求分页要实现的主要就是请求调页和页面置换功能，即将所需信息从外存调入内存，将暂时不需要的信息换出到外存。所以实现过程的解析主要从页表、缺页中断、地址变换三个方面的机制入手。

对于页表来说，页表项中增加状态位、访问字段、修改位、外存地址四个字段来应对型增加的俩个主要功能。而如果发生缺页中断，系统首先去检查缺页异常程序的虚拟地址在哪个 area 内。然后如果访问的虚拟页若没有读写权限，则触发一个保护异常，终止进程。如果有权限，则选择牺牲页，刷新到磁盘（内

存够就不用），从磁盘加载缺失的内容到物理页，更新页表 。而对于地址变换，首先需要检索快表看是否存在访问页，若存在修改访问位（写指令重置修改位），利用页表信息形成物理地址。若不存在，需要查看是否调入内存，未调入则缺页中断，请求该页调入内存（内存不够要换出页面）。

4. 内存分配与回收

Linux 的内存分配问题主要是通过伙伴算法、slab 分配器和 per-CPU 页框缓存来解决的。

伙伴算法负责大块连续物理内存的分配与释放，是以页为单位进行单位管理和分配内存的。将内存按 2 的幂次划分，并搜索其内存空间找到最佳适配的内存。即在进行内存分配时，不断平分较大的空闲内存块来获得较小的空闲内存块，直到获得所需要的内存块；而在进行内存回收时，则是尽可能地合并空闲块。

将所有的空闲页框分为 MAX_ORDER+1 个块链表，每个链表中的一个块都包含着 2 的幂次方个页框，大小相同、物理地址连续的两个块被称为伙伴。具体的分配思想就是首先在满足大小要求的块链表中查找是否有空闲块，有就直接分配，反之则在更大的块中寻找。如果在最大块的链表中都没有找到空闲块的话就放弃分配并报错；逆过程就是通过把符合伙伴关系的块合并来达到释放的效果，但值得注意的是在这之前需要判断伙伴块是否为空，为空直接插入空闲链表 free_area 即可；不为空需要合并伙伴块直到伙伴块为空。每个节点处的物理内存被 Linux 分为 ZONE_DMA、ZONE_NORMAL、ZONE_HIGHMEM 三个 Zone，每个 Zone 都由自己的伙伴系统进行内存管理。

SLAB 分配器负责小块物理内存的分配，是以字节为单位来分配内存的，基于伙伴分配算法的大内存进一步细化分成小内存分配。即 SLAB 分配仍然从 Buddy 分配器中申请内存，之后再自己对申请来的内存精细化管理。并利用储存池来缓存被当作对象的小块内存以避免频繁创造和销毁对象，快速创建对象，解决初始化对象所需的时间可等于或超过为其分配空间的成本的问题。最后通过 SLAB 着色提高 CPU 硬件缓存的利用率，即通过将对象放置在 SLAB 中的不同起始偏移

处来使不同 SLAB 对象使用 CPU 硬件缓存中不同行的方案。 对象可能会在 CPU 缓存中使用不同的行确保来自同一 SLAB 缓存的对象不太可能相互刷新。

从结构上来讲，多个 SLAB 组成高速缓冲区，通过 cache_chain 查找需要分配的合适的缓存区，而 cache_chain 的元素都是对 kmem_cache 结构的引用，其中 kmem_cache 的内存又划分为多个 SLAB。每个 SLAB 根据已分配的对象数动态处于 slabs_full、slabs_partial 和 slabs_empty 三种队列之一中。SLAB 分配器优先从 slabs_partial 中分配对象。空闲对象不足则分配更多 SLAB，反之则定期回收 slabs_empty 中的对象。这样没有造成过多的空间浪费，并支持硬件缓存对齐来支持 TLB 的性能。

per-CPU 页框缓存负责单一页框的请求，per-CPU 页框缓存包含预先分配的页框，可以满足内核频繁请求和释放单一页框的需求。有时候目标管理区不一定有足够的页框去满足分配，需要从另外两个管理区中获取页框。

5. 换页策略

换页策略就是因为物理内存紧张而换出某些页面到外存，需要的时候再调入内存而诞生的。并且 Linux 使用 Swap 交换空间，把一部分磁盘虚拟成内存使用，磁盘 IO 就成为了重点之一。与此同时我们不能没有目的地胡乱换入换出，而应该换出哪些真正需要被换出的页面，避免被换出的页面下一刻又要访问的现象。提高磁盘 IO 效率的方法之一就是建立缓存，而根据局部性原理，LRU 算法是比较合理的。所以下面主要从缓存和 LRU 算法的实现两个方面介绍换入换出机制。

首先要说的是 LRU 链表。每个区域 zone 都关联一组 LRU 链表，链表记录了物理页面的状态，物理页面的回收工作就是以这些链表为依据。但又因为它直接管理着各种状态的页面，如果一直对这个链表操作毫无疑问会降低性能。所以使用 LRU 缓存优先记录 CPU 调入内存的页面并记录 page 数目等信息。当 LRU 缓存满了之后就把 LRU 缓存中的所以 page 添加到对应的 LRU 链表中，以达到批量地向 LRU 链表中快速地添加页面的目的。

LRU 算法的实现方面来说的话，Linux 对 LRU 实现基于一对双向链表：active 链表和 inactive 链表，存在于每个内存区域。经常被访问的活跃页面在 active 链表上，而可能关联多个进程但是并不经常使用的页面则在 inactive 链表上。页面会在这两个双向链表中移动，操作系统会根据页面的活跃程度来判断应该把页面放到哪个链表上。那些最近最少使用的页面会被逐个放到 inactive 链表的尾部。进行页面回收的时候，Linux 操作系统会从 inactive 链表的尾部开始进行回收。

二、 描述某种环境下效率最高的 I/O 混合策略

题目让探究在什么场景下混合模式优于单纯的轮询或者中断，所以我会从分析轮询和中断的优缺点以及适用环境来设计相应的混合模式。

对于轮询方式来说：它的优点是程序简单，速度快、传输可靠，I / O 设备就绪，就不需要额外的访问芯片；缺点是某些设备的待机时间可能短于响应时间并且不断对 IO 端口执行死循环的方式大量的浪费了 CPU，检测效率低。

而对于中断方式来说：它的优点就是提高了处理器的利用率并提高了多道程序和 IO 设备的并行操作的效率，灵活且高效；缺点是需要更复杂的硬件/软件、随着中断次数增加 CPU 响应难度提升并且 CPU 会浪费确定是哪个单元请求中断的时间。

结合题目来说就是：I/O 操作等待的时间是关键，很短的等待时间内纯粹的轮询效率相对更高，中断在较长的等待时间内是效率相对更高的。所以混合的场景我认为可以是根据等待时间的长短在中断和轮询中互相切换。

我们可以首先在一个设定的较短的时间段 T 内进行轮询（或者指定一个合适的轮询次数 N），如果超出设定的时间段 T 或者次数 N 依然很忙，就选择中断并睡眠。这样即可以避免长时间循环检测忙碌等待，也可以减少一部分避免捕捉和分配中断造成的效率低的影响。即比较短的时间段和比较长的时间段都可以适合，在这种环境中，该策略比其他任何一种都更有效率。

三、 设计 N 个进程并发执行系统的进程调度方案

1. 数据结构设计

整个方案由三个队列构成：准备队列、三级运行队列、完成队列组成。所以需要实现的结构体有文件控制块以及队列控制块。

准备队列和完成队列都是由文件控制块组成的链表。而三级运行队列是由队列控制块组成的链表，队列结构体中含有队列中第一个进程的指针，将队列控制块与在该级队列中的进程链表相连。

2. 逻辑设计

表 1 函数作用以及调用关系

序号	名称	作用	调用
1)	void initProcess()	随机生成或初始化进程样本。	
2)	void showProcess()	显示此时所有进程信息。	
3)	void initQueues()	初始化队列。	
4)	void showQueues()	展示队列及队列内进程信息。	
5)	void insertQueue(Queue *queue, PCB *pcb)	将进程插入队列尾部。	
6)	void printAllByTime()	按时间段打印信息。	2、4
7)	void checkIfAdd()	检查是否应该添加进程进入。	
8)	void findMaxCreateTime()	预估所有进程处理时间。	
9)	void runProcess(Queue *queue, int timeSlice)	在运行队列时间片内运行进程，没有完成则插入下一级。	5、6
10)	void runQueue()	循环运行三级运行队列，直到所有进程结束。	9
11)	void freeAll()	释放所有内存。	
12)	int main()	主程序。	1-11

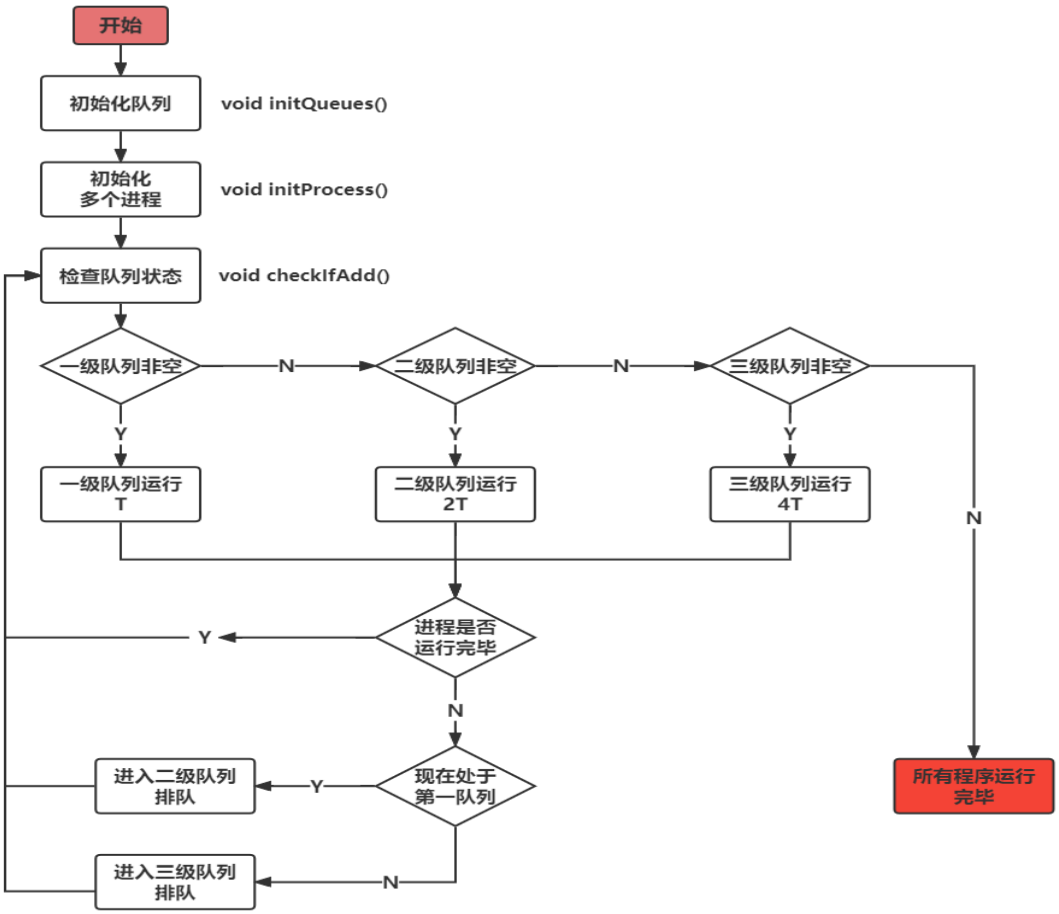

图 1 进程调度流程图

3. 完整代码（C 语言实现）

```c
#include <stdio.h>
#include<stdlib.h>
#include<math.h>
#define T 1

//进程的状态：初始化赋值 I(initialize)，就绪状态 W，运行状态 R，完成状态 F
typedef struct node
{
    int name;
    int create_time;
    int need_time;
    int used_time;
    int remain_time;
    char status;
    struct node *next;
} PCB;
```

```
//队列结构体
typedef struct Queues
{
    int name;
    int timeSlice;
    PCB *data; //队列中的第一个进程指针
    struct Queues *next;
} Queue;

int time = 0;
int tempTimeSlice = 0;
int minWaitTime = 0;
Queue *head = NULL;
Queue *tail = NULL;
Queue *finishQF = NULL;
PCB *processHead = NULL;
PCB *processTail = NULL;

/*
 * 进程的初始化并随机录入进程的具体信息，进程链表作为准备队列
*/

//初始化进程链表（准备队列）并录入信息
void initProcess() {
    processHead = (PCB *)malloc(sizeof(PCB));
    processHead->next = NULL;
    processTail = processHead;

//    //随机录入信息
//    for (int i = 0; i < 5; ++i) {
//        PCB *newPCB = (PCB *)malloc(sizeof(PCB));
//        newPCB->name = i+1;
//        newPCB->create_time = rand() % 12;
//        newPCB->need_time = rand() % 10 + 1;
//        newPCB->used_time = 0;
//        newPCB->remain_time = newPCB->need_time;
//        newPCB->status = 'I';
//        newPCB->next = NULL;
//        processTail->next = newPCB;
//        processTail = newPCB;
//    }

//建立数据集，并将信息录入
    PCB *newPCB = (PCB *)malloc(sizeof(PCB));
```

```
    newPCB->name = 1;
    newPCB->create_time = 1;
    newPCB->need_time = 9;
    newPCB->used_time = 0;
    newPCB->remain_time = newPCB->need_time;
    newPCB->status = 'I';
    newPCB->next = NULL;
    processTail->next = newPCB;
    processTail = newPCB;

    newPCB = (PCB *)malloc(sizeof(PCB));
    newPCB->name = 2;
    newPCB->create_time = 3;
    newPCB->need_time = 5;
    newPCB->used_time = 0;
    newPCB->remain_time = newPCB->need_time;
    newPCB->status = 'I';
    newPCB->next = NULL;
    processTail->next = newPCB;
    processTail = newPCB;

    newPCB = (PCB *)malloc(sizeof(PCB));
    newPCB->name = 3;
    newPCB->create_time = 5;
    newPCB->need_time = 3;
    newPCB->used_time = 0;
    newPCB->remain_time = newPCB->need_time;
    newPCB->status = 'I';
    newPCB->next = NULL;
    processTail->next = newPCB;
    processTail = newPCB;

    newPCB = (PCB *)malloc(sizeof(PCB));
    newPCB->name = 4;
    newPCB->create_time = 6;
    newPCB->need_time = 1;
    newPCB->used_time = 0;
    newPCB->remain_time = newPCB->need_time;
    newPCB->status = 'I';
    newPCB->next = NULL;
    processTail->next = newPCB;
    processTail = newPCB;

    newPCB = (PCB *)malloc(sizeof(PCB));
```

```
    newPCB->name = 5;
    newPCB->create_time = 10;
    newPCB->need_time = 3;
    newPCB->used_time = 0;
    newPCB->remain_time = newPCB->need_time;
    newPCB->status = 'I';
    newPCB->next = NULL;
    processTail->next = newPCB;
    processTail = newPCB;

}

//显示进程信息
void showProcess() {
    if (time == 0) {
        printf("-----------------------所有进程情况--------------------------\n");
    } else {
        printf("-------------------在运行队列中的进程情况----------------------
\n");
    }
    printf("进程名 创建时间 需要运行时间 已用 CPU 时间 还需时间 进程状态\n");
    //如果三级队列为空，单纯显示进程链表信息
    if (head == NULL) {
        PCB *p = processHead;
        while(p->next!=NULL) {
            p=p->next;
            printf(" P%d\t %d\t  %d\t     %d\t      %d\t    %c\t\n",p->name,p->crea
te_time,p->need_time,p->used_time,p->remain_time,p->status);
        }
        printf("-------------------------------------------------------------
\n\n");
    } else {
        //如果三级队列不为空，显示三级队列信息
        Queue *q = head;
        while (q != NULL) {
            PCB *p = q->data;
            while (p != NULL) {
                printf(" P%d\t %d\t  %d\t     %d\t      %d\t    %c\t\n",p->name,p->
create_time,p->need_time,p->used_time,p->remain_time,p->status);
                p = p->next;
            }
            q = q->next;
        }
```

```
        printf("------------------------------------------------------------
\n\n");
    }
}

/*
 * 初始化三级队列，每一级可以存放的进程数量为 4。
 * 初始化时间片为 2，一级队列的时间片为 1，二级队列的时间片为 2，三级队列的时间片为 4
*/

//初始化队列：三级运行队列和完成队列
void initQueues() {
    //初始化三级运行队列
    for(int i = 0; i < 3; i++) {
        Queue *newQueue = (Queue *)malloc(sizeof(Queue));
        newQueue->timeSlice = pow(2, i) * T;
        newQueue->name = i+1;
        newQueue->data = NULL;
        newQueue->next = NULL;
        if(head == NULL) {
            head = newQueue;
        } else {
            Queue *p = head;
            while(p->next != NULL) {
                p = p->next;
            }
            p->next = newQueue;
        }
        //tail
        if(i == 0) {
            tail = newQueue;
        } else {
            tail = tail->next;
        }
    }

    //初始化完成队列
    Queue *newQueue = (Queue *)malloc(sizeof(Queue));
    newQueue->timeSlice = 0;
    newQueue->name = 4;
    newQueue->data = NULL;
    newQueue->next = NULL;
    finishQF = newQueue;
}
```

```
//显示队列
void showQueues() {
    printf("----------------------三级运行队列情况-------------------------\n");
    //显示三级运行队列
    Queue *p = head;
    printf("队列名\t 时间片\t   队列内进程\n");
    while(p != NULL) {
        printf(" Q%d\t %d\t", p->name, p->timeSlice);
        PCB *q = p->data;
        while(q != NULL) {
            printf("P%d ", q->name);
            q = q->next;
        }
        printf("\n");
        p = p->next;
    }
    printf("-------------------------------------------------------------\n\n");

    //显示完成队列
    Queue *q = finishQF;
    printf("----------------------已完成进程情况-------------------------\n");
    printf("进程名 创建时间 需要运行时间 已用 CPU 时间 还需时间 进程状态\n");
    while (q != NULL) {
        PCB *temp = q->data;
        while (temp != NULL) {
            printf(" P%d\t %d\t  %d\t     %d\t      %d\t    %c\t\n", temp->name, te
mp->create_time, temp->need_time, temp->used_time, temp->remain_time, temp->status)
;
            temp = temp->next;
        }
        q = q->next;
    }
    printf("-------------------------------------------------------------\n\n\n");
}


//将进程插入队列尾部
void insertQueue(Queue *queue, PCB *pcb) {
    if(queue->data == NULL) {
        pcb->next = NULL;
        queue->data = pcb;
    } else {
        PCB *p = queue->data;
```

```
        while(p->next != NULL) {
            p = p->next;
        }
        pcb->next = NULL;
        p->next = pcb;
    }
}

//打印每个时刻进程块和队列的情况
void printAllByTime() {
    printf("\n");
    printf("时间块：%d - %d\n", time-1, time);
    printf("进程块\n");
    showProcess();
    showQueues();
}

/*
 * 进程插入队列
*/

//检查进程链表节点的 create_time，当 time 等于 create_time 时，将该节点插入队列尾部
void checkIfAdd() {
    PCB *p = processHead;
    while(p != NULL) {
        if(p->create_time <= time) {
            //将该节点从进程链表中删除,头结点后移
            PCB *q = processHead;
            if(q == p) {
                processHead = p->next;
            } else {
                while(q->next != p) {
                    q = q->next;
                }
                q->next = p->next;
            }
            insertQueue(head, p);
            p->status = 'W';
        }
        p = p->next;
    }
}

//找出到达时间最大的片段
```

```
void findMaxCreateTime() {
    PCB *p = processHead;
    int maxCreateTime = 0;
    int maxNeedTime = 0;
    while(p != NULL) {
        if(p->create_time >= maxCreateTime) {
            maxCreateTime = p->create_time;
            maxNeedTime = p->need_time;
        }
        p = p->next;
    }
    minWaitTime = (maxCreateTime + maxNeedTime)*5;
}



//运行进程：在该队列时间片内运行进程，如果进程运行完成，则将其从队列中删除，运行下一个；如果进程没有完成，就添加到下一级队列里
void runProcess(Queue *queue, int timeSlice) {
    //按时间片循环运行进程
    if(tempTimeSlice == 0) {
        tempTimeSlice = timeSlice;
    }
    while(queue->data != NULL&&tempTimeSlice > 0) {
        PCB *p = queue->data;
        while(p != NULL) {
            time++;
            p->used_time++;
            p->remain_time--;
            p->status = 'R';
            tempTimeSlice--;

            //进程运行完成,但时间片有剩余，则将其从队列中删除，在剩余时间片运行下一个。
            if(p->remain_time == 0) {
                printAllByTime();
                p->status = 'F';
                //将进程从队列中删除
                if(p->next != NULL) {
                    queue->data = p->next;
                    //添加到完成队列
                    insertQueue(finishQF, p);
                    break;
                } else {
                    queue->data = NULL;
```

```
                    //添加到完成队列
                    insertQueue(finishQF, p);
                    break;
                }

            }
            //如果进程没有完成，时间片已经用完，则将其从队列中删除,添加到下一级队列里，如果是最后一个队列，则重新添加到队列尾部
            if(p->remain_time > 0 && tempTimeSlice == 0) {
                printAllByTime();
                if(p->next != NULL) {
                    queue->data = p->next;
                } else {
                    queue->data = NULL;
                }
                if(queue->next != NULL) {
                    p->status = 'W';
                    insertQueue(queue->next, p);
                } else {
                    p->status = 'W';
                    insertQueue(tail, p);
                }
                break;
            }

            //进程没有运行完成,时间片未运行完，则继续运行该进程。
            if(p->remain_time > 0) {
                printAllByTime();
                continue;
            }
        }
    }
}

/*
 * 正式循环运行三级队列，直到所有程序运行结束
*/

//运行队列:
void runQueue() {
    Queue *q1 = head;
    Queue *q2 = head->next;
    Queue *q3 = head->next->next;
    //循环运行三级队列，直到所有程序运行结束
```

```
    while(1) {
        checkIfAdd();
//        printf("time:%d minWaitTime:%d\n",time,minWaitTime);
        if(q1->data != NULL && time >= q1->data->create_time) {
            runProcess(q1, q1->timeSlice);
        } else if(q2->data != NULL && time >= q2->data->create_time) {
            runProcess(q2, q2->timeSlice);
        } else if(q3->data != NULL && time >= q3->data->create_time) {
            runProcess(q3, q3->timeSlice);
        } else {
            if(q1->data == NULL && q2->data == NULL && q3->data == NULL && time ==
minWaitTime) {
                printf("三级队列中的所有进程都运行完成啦！\n");
                break;
            } else {
                time++;
            }
        }
    }
}

//释放所有内存
void freeAll() {
    //释放队列进程
    PCB *p1 = processHead;
    while(p1 != NULL) {
        PCB *q1 = p1;
        p1 = p1->next;
        free(q1);
    }
    //释放三级队列
    Queue *p2 = head;
    while(p2 != NULL) {
        Queue *q2 = p2;
        p2 = p2->next;
        free(q2);
    }
    //释放完成队列
    Queue *p3 = finishQF;
    free(p3);
}

int main() {
    system("chcp 65001");
```

```
    initProcess();
    showProcess();

    initQueues();
    showQueues();
    processHead = processHead->next;
    findMaxCreateTime();

    runQueue();
    freeAll();
    return 0;
}
```

4. 输入输出结果

输出结果是每个时间段的到达进程情况、三级运行队列情况、已完成进程情况。进程信息如上代码所示可以随机生成或使用实例运行。这里展示的是其中一个例子。需要 22 个时间块。如图 3 所示，蓝色代表在一级运行队列，紫色二级，橙色三级。高亮黄色外框意味着在该时间段结束进程将运行完成。

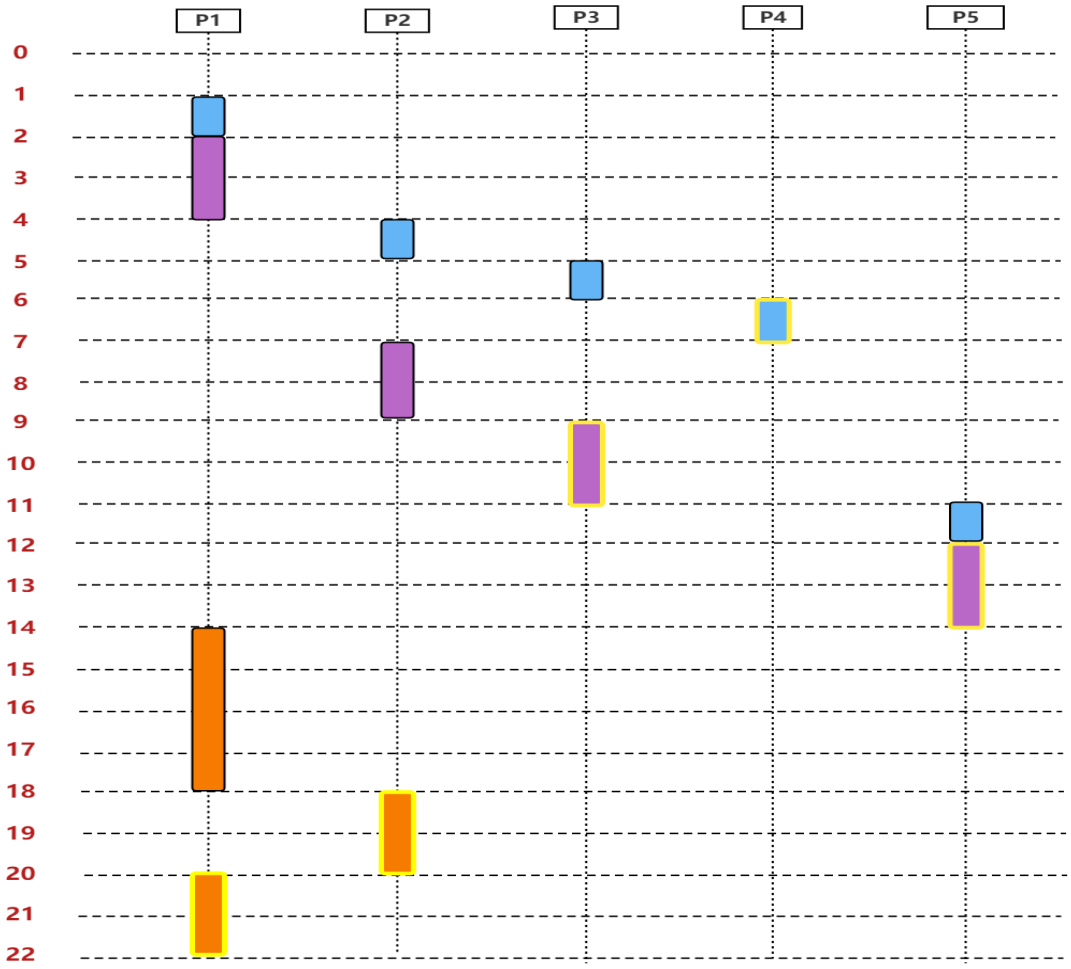

图 2 实例进程调度的时序图

控制台所有内容很长，所以只展示了其中部分截图。如图 4 所示，是初始化后示例数据的具体信息。位于所有输出结果的最上面。

图 4 实例进程初始化情况

然后队列开始运行。如图 5 所示，P1 在时间块 1-2 到达并开始在一级队列运行。

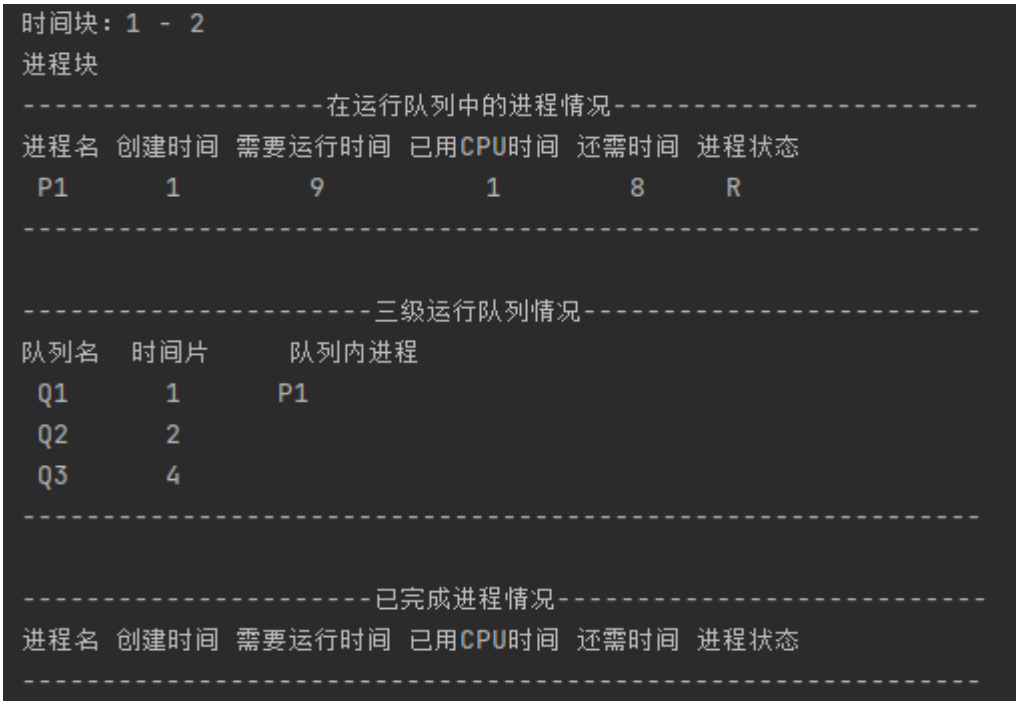

图 5 时间块 1-2 时进程及队列情况

如图 6 所示，在时间块 7-8 中，P4 刚刚运行结束，P2 正在二级队列运行

图 6 时间块 7-8 时进程及队列情况

如图 7 所示，最后一个时间块，P1 在三级队列运行，其它进程已经运行完毕。T = 22 时，所有进程运行完毕。

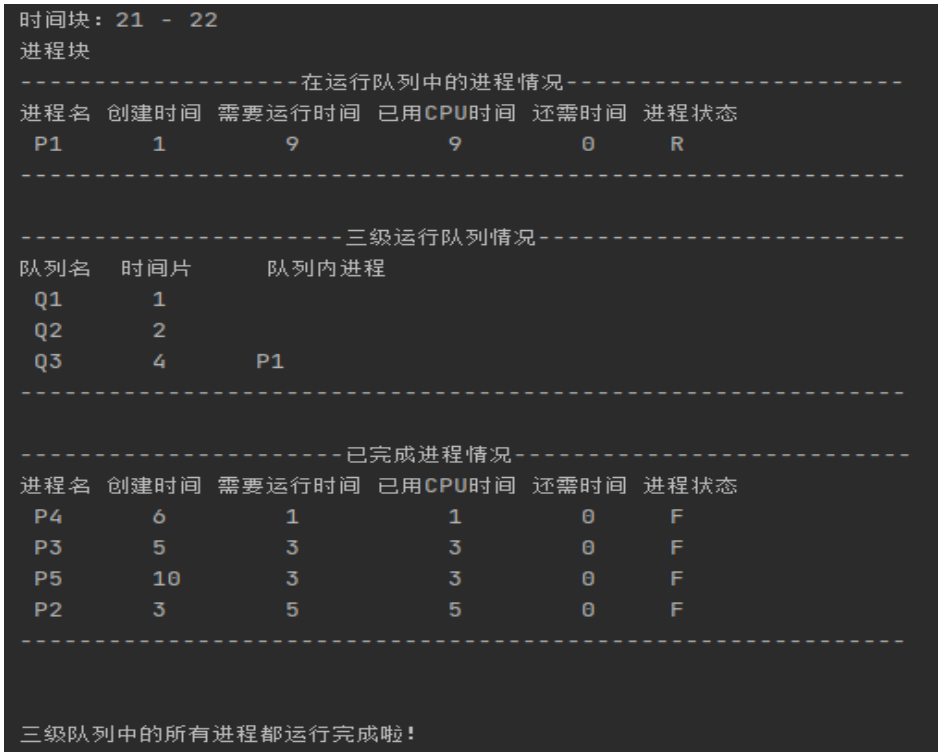

图 7 时间块 21-22 时进程及队列情况

四、 设计简单文件系统

1. 数据结构设计

结构体主要是磁盘块（虚拟磁盘空间模拟磁盘）、用户目录（用户文件）、主目录、运行目录。因为不涉及具体读写操作，所以默认所有文件只是根据大小在磁盘上分配空间，并不存储数据。磁盘块使用链表链接。用户目录、主目录和运行目录都是使用的数组。

2. 逻辑设计

表 2 函数作用以及调用关系

序号	名称	作用	调用
1)	void initDisk()	初始化虚拟磁盘。	
2)	void allocateDiskBlock(int currentUserID, int fileNum)	分配磁盘块。	
3)	void freeDiskBlock(int currentUserID, int fileNum)	释放磁盘块。	
4)	void initData()	初始化示例原始数据。	2
5)	void initAFD()	初始化打开目录信息。	
6)	void showDiskBlock()	显示各个磁盘块的占用情况。	
7)	void showUserFile(int userNum)	展示该用户名下所有文件。	
8)	void showAllFileInfo(int currentUserID)	展示所有用户所有文件信息。	
9)	void showOpenFileInfo()	展示打开文件表列。	
10)	void showAllCommand()	展示所有命令的提示语句。	
11)	int login()	登录账户。	
12)	void createFile(int currentUserID)	新建文件并分配内存。	2
13)	void deleteFile(int currentUserID)	删除文件并释放内存。	3

序号	名称	作用	调用
14)	void openFile(int currentUserID)	打开文件。	
15)	void closeFile(int currentUserID)	关闭文件。	
16)	void readFile()	进行读操作	
17)	void writeFile()	进行写操作	
18)	int main()	主函数	1-17

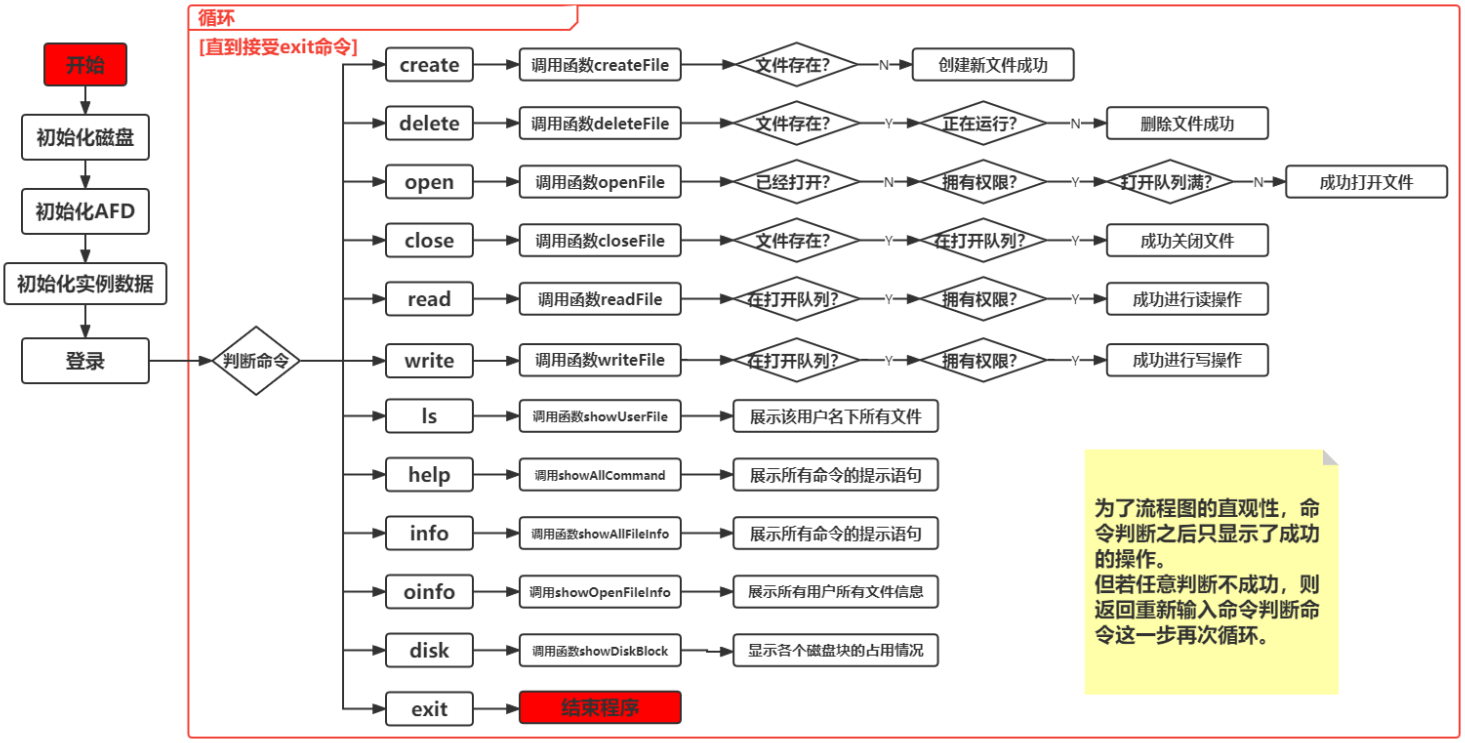

图 8 简单文件系统流程图

3. 完整代码（C 语言实现）

```c
#include <stdio.h>
#include<stdlib.h>
#include <string.h>
#define MaxDiskSize 512 //磁盘大小 512KB

//磁盘块结构体
typedef struct DiskBlock
{
    char userName[10];
    char fileName[10];
    int blockSize;
    int startPos;
    int endPos;
    struct DiskBlock *next;
}diskNode;

//用户目录（某用户名下存放的文件）
typedef struct ufd
{
    char fileName[10];
    int flag;
    int fileProtectCode[3]; //r\w\e
    char fileCreateDate[20];
    int fileLength;
    int fileStartPos;
    int fileEndPos;
}UF[10];

//主目录(用户名和密码)
struct mfd
{
    char userName[10];
    char password[20];
    UF uDir;
}UFD[10];

//运行文件目录
struct afd
{
    char openFileName[10];
    int flag;
    char openFileProtectCode[3];
```

```
    int rwPoint;
}AFD[5];

diskNode *diskHead; //磁盘头指针

//在内存中建立一个虚拟磁盘空间作为文件存储器（模拟磁盘）分配磁盘块，返回分配的磁盘块的起始地址
//因为不涉及具体读写操作，所以默认所有文件只是根据大小在磁盘上分配空间，并不存储数据
void initDisk() {
    if (diskHead == NULL) {
        diskHead = (diskNode *) malloc(sizeof(diskNode));
    }
    //初始化磁盘块
    for(int i = 0; i < 10; i++) {
        diskHead->userName[i] = '\0';
        diskHead->fileName[i] = '\0';
    }

    diskHead->blockSize = MaxDiskSize;
    diskHead->startPos = 0;
    diskHead->endPos = 0;
    diskHead->next = NULL;
}

//分配磁盘块
void allocateDiskBlock(int currentUserID, int fileNum) {
    //如果需求的磁盘块大小大于磁盘大小，则报错
    if (UFD[currentUserID].uDir[fileNum].fileLength > MaxDiskSize) {
        printf("文件大小超出磁盘大小，请重新分配磁盘块\n");
    }
    //如果是第一个文件，则直接分配磁盘块
    if (currentUserID == 0 && fileNum == 0) {
        diskHead->startPos = 0;
        diskHead->endPos = UFD[currentUserID].uDir[fileNum].fileLength;
        diskHead->blockSize = diskHead->blockSize - UFD[currentUserID].uDir[fileNum].fileLength;
        for(int i = 0; i < 10; i++) {
            diskHead->userName[i] = '\0';
            diskHead->fileName[i] = '\0';
        }
        strcpy(diskHead->userName, UFD[currentUserID].userName);
        strcpy(diskHead->fileName, UFD[currentUserID].uDir[fileNum].fileName);
        diskHead->next = NULL;
        UFD[currentUserID].uDir[fileNum].fileStartPos = diskHead->startPos;
```

```
        UFD[currentUserID].uDir[fileNum].fileEndPos = diskHead->startPos + diskHead
->endPos;

    } else {
        //如果不是第一个文件，则需要在磁盘上分配空间，并且将文件信息写入磁盘块
        //如果需求的磁盘块大小小于剩余磁盘大小，则分配磁盘块
        diskNode *temp = diskHead;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        if (UFD[currentUserID].uDir[fileNum].fileLength <= temp->blockSize) {
            temp->next = (diskNode *) malloc(sizeof(diskNode));
            temp->next->startPos = temp->endPos;
            temp->next->endPos = temp->endPos + UFD[currentUserID].uDir[fileNum].fi
leLength;
            temp->next->blockSize = temp->blockSize - UFD[currentUserID].uDir[fileN
um].fileLength;
            temp = temp->next;
            for(int i = 0; i < 10; i++) {
                temp->userName[i] = '\0';
                temp->fileName[i] = '\0';
            }
            strcpy(temp->userName, UFD[currentUserID].userName);
            strcpy(temp->fileName, UFD[currentUserID].uDir[fileNum].fileName);
            temp->next = NULL;
            UFD[currentUserID].uDir[fileNum].fileStartPos = temp->startPos;
            UFD[currentUserID].uDir[fileNum].fileEndPos = temp->endPos;
        } else {
            printf("磁盘空间不足，请重新分配磁盘块\n");
        }
    }
}

//释放块，将磁盘块释放到磁盘上
void freeDiskBlock(int currentUserID, int fileNum) {
    diskNode *temp = diskHead;
    //寻找文件
    while (strcmp(temp->userName, UFD[currentUserID].userName) != 0 || strcmp(temp-
>fileName, UFD[currentUserID].uDir[fileNum].fileName) != 0) {
        temp = temp->next;
    }
    if (temp->startPos == 0) {
        diskHead = diskHead->next;
        diskNode *temp2 = diskHead;
```

```
        int size = temp->endPos - temp->startPos;
        //空间增加
        while (temp2->next != NULL) {
            temp2->blockSize = temp2->blockSize + size;
            temp2->startPos = temp2->startPos - size;
            temp2->endPos = temp2->endPos - size;
            temp2 = temp2->next;
        }
        temp->next = NULL;
        free(temp);
    } //如果是最后一个
    else if (temp->next == NULL) {
        diskNode *temp2 = diskHead;
        int size = temp->endPos - temp->startPos;
        while (temp2->next != temp) {
            temp2->blockSize = temp2->blockSize + size;
            temp2 = temp2->next;
        }
        temp2->next = NULL;
        free(temp->next);
        temp->next = NULL;
    } else {
        diskNode *temp2 = temp->next;
        diskNode *temp3 = temp->next;
        diskNode *temp4 = diskHead;
        //寻找前一个磁盘块
        while (temp4->next != temp) {
            temp4 = temp4->next;
        }
        int size = temp->endPos - temp->startPos;
        //计算节点数量
        int count = 0;
        while (temp3 != NULL) {
            count++;
            temp3 = temp3->next;
        }
        //空间增加
        while (count > 0) {
            temp2->blockSize = temp2->blockSize + size;
            temp2->startPos = temp2->startPos - size;
            temp2->endPos = temp2->endPos - size;
            temp2 = temp2->next;
            count--;
        }
```

```
        //链接节点
        temp4->next = temp->next;
        temp->next = NULL;
        free(temp);
    }
}


//初始化主目录和用户目录的展示例子：俩个账户 root 和 alex，每个账户下面都有 2 个文件。
void initData()
{
    //root 账户
    strcpy(UFD[0].userName, "1");
    strcpy(UFD[0].password, "1");

    //a 和 b2 个 txt 文件
    strcpy(UFD[0].uDir[0].fileName, "a.txt");
    UFD[0].uDir[0].flag = 1;
    for(int i=0; i < 3; i++)
    {
        UFD[1].uDir[1].fileProtectCode[i] = 1;
    }
    strcpy(UFD[0].uDir[0].fileCreateDate, "2022/07/01");
    UFD[0].uDir[0].fileLength = 10;
    //将文件 a.txt 分配到磁盘上
    allocateDiskBlock(0, 0);

    strcpy(UFD[0].uDir[1].fileName, "b.txt");
    UFD[0].uDir[1].flag = 1;
    for(int i=0; i < 3; i++)
    {
        UFD[1].uDir[1].fileProtectCode[i] = 1;
    }
    strcpy(UFD[0].uDir[1].fileCreateDate, "2022/07/01");
    UFD[0].uDir[1].fileLength = 20;
    //将文件 b.txt 分配到磁盘上
    allocateDiskBlock(0, 1);


    //alex 账户
    strcpy(UFD[1].userName, "alex");
    strcpy(UFD[1].password, "111");
    //c 和 d2 个 txt 文件
    strcpy(UFD[1].uDir[0].fileName, "c.txt");
```

```
    UFD[1].uDir[0].flag = 1;
    for(int i=0; i < 3; i++)
    {
        UFD[1].uDir[1].fileProtectCode[i] = 0;
    }
    strcpy(UFD[1].uDir[0].fileCreateDate, "2022/07/02");
    UFD[1].uDir[0].fileLength = 30;
    //将文件 c.txt 分配到磁盘上
    allocateDiskBlock(1, 0);

    strcpy(UFD[1].uDir[1].fileName, "d.txt");
    UFD[1].uDir[1].flag = 1;
    for(int i=0; i < 3; i++)
    {
        //随机生成 0-1 的三个字符串，作为文件的保护码
        UFD[1].uDir[1].fileProtectCode[i] = 0;
    }
    strcpy(UFD[1].uDir[1].fileCreateDate, "2022/07/02");
    UFD[1].uDir[1].fileLength = 40;
    //将文件 d.txt 分配到磁盘上
    allocateDiskBlock(1, 1);
}

//初始化打开文件目录
void initAFD()
{
    for(int i=0;i<5;i++)
    {
        strcpy(AFD[i].openFileName, "XXX");
        AFD[i].flag=0;
        for(int k=0;k<3;k++)
        {
            AFD[k].openFileProtectCode[k]=0;
        }
        AFD[i].rwPoint=0;
    }
}

//显示磁盘块链表信息
void showDiskBlock()
{
    //计算链表长度
    int length = 1;
    diskNode *temp = diskHead;
```

```
    while (temp->next != NULL) {
        length++;
        temp = temp->next;
    }
    diskNode *temp1 = diskHead;
    int index = 1;
    printf("磁盘块链表信息:\n");
    printf("------------------------------------------------------------------------
---\n");
    printf("序号       用户名       文件名       文件大小       文件开始位置       文件结束位
置\n");
    while (index <= length) {
        printf(" %d\t %s\t    %s\t  %d\t       %d\t       %d\t\n", index, temp1->us
erName, temp1->fileName,temp1->endPos - temp1->startPos, temp1->startPos, temp1->en
dPos);
        temp1 = temp1->next;
        index++;
    }
    printf("------------------------------------------------------------------------
---\n\n");
}

//展示用户拥有的所有文件
void showUserFile(int userNum)
{
    printf("------------------------------------------------------------------------
---\n\n");
    printf("用户%s 的文件列表:\n", UFD[userNum].userName);
    for(int i=0;i<10;i++)
    {
        if(UFD[userNum].uDir[i].flag == 1)
        {
            printf("%s\n", UFD[userNum].uDir[i].fileName);
        }
    }
    printf("------------------------------------------------------------------------
---\n\n");
}

//显示所有文件的详细信息
void showAllFileInfo(int currentUserID)
{
    printf("该用户所有文件的详细信息:\n");
```

```
    printf("------------------------------------------------------------------------
---\n");
    printf("文件名       创建时间       文件大小  读权限    写权限    执行权限 文件起始位置 文
件结束位置\n");
    for(int i=0;i<10;i++)
    {
        if(UFD[currentUserID].uDir[i].flag == 1)
        {
            printf("%s\t %s\t %d\t %d\t %d\t %d\t   %d\t     %d\t\n", UFD[currentUs
erID].uDir[i].fileName, UFD[currentUserID].uDir[i].fileCreateDate, UFD[currentUserI
D].uDir[i].fileLength, UFD[currentUserID].uDir[i].fileProtectCode[0], UFD[currentUs
erID].uDir[i].fileProtectCode[1], UFD[currentUserID].uDir[i].fileProtectCode[2], UF
D[currentUserID].uDir[i].fileStartPos, UFD[currentUserID].uDir[i].fileEndPos);
        }
    }
    printf("------------------------------------------------------------------------
---\n\n");
}

//显示已经打开的文件的详细信息
void showOpenFileInfo()
{
    printf("该用户所有已经打开的文件的详细信息:\n");
    printf("------------------------------------------------------------------------
---\n");
    printf("文件名    读权限     写权限    执行权限 \n");
    for(int i=0;i<5;i++)
    {
        if(AFD[i].flag == 1)
        {
            printf("%s\t %d\t %d\t %d\t \n",AFD[i].openFileName, AFD[i].openFilePro
tectCode[0], AFD[i].openFileProtectCode[1], AFD[i].openFileProtectCode[2]);
        }
    }
    printf("------------------------------------------------------------------------
---\n\n");
}

//显示所有指令
void showAllCommand()
{
    printf("---------------------------------可用指令如下----------------------------
-----\n");
    printf("1. 新建文件 -- create\n");
```

```
    printf("2. 删除文件 -- delete\n");
    printf("3. 打开文件 -- open\n");
    printf("4. 关闭文件 -- close\n");
    printf("5. 文件读写 -- read/write\n");
    printf("6. 文件信息 -- info\n");
    printf("7. 打开文件信息 -- oinfo\n");
    printf("8. 文件列表 -- ls\n");
    printf("9. 磁盘信息 -- disk\n");
    printf("10. 退出系统 -- exit\n");
    printf("------------------------------------------------------------------------
---\n\n");
}

//登录到系统
int login()
{   while(1)
    {
        int i;
        char userName[10];
        char password[20];
        printf("-----------------------------简单文件系统登录页面--------------------
---------\n");
        printf("请输入用户名：");
        scanf("%s",userName);
        printf("请输入密码：");
        scanf("%s",password);
        for(i=0;i<10;i++)
        {
            if(strcmp(userName,UFD[i].userName)==0 && strcmp(password,UFD[i].passwo
rd)==0)
            {
                printf("登录成功！\n");
                printf("------------------------------------------------------------
---------------\n\n");
                return i;
            }
        }
        printf("用户名或密码错误，登录失败！\n");
        exit(0);
    }
}

//创建文件
void createFile(int currentUserID)
```

```
{
    int i;
    char fileName[10];
    char fileCreateDate[10];
    int fileLength = 0;

    //初始化 char
    for(i=0;i<10;i++)
    {
        fileName[i] = '\0';
        fileCreateDate[i] = '\0';
    }

    LOOP:while(1)
    {
        printf("请输入文件名: ");
        scanf("%s", fileName);
        for (i = 0; i < 10; i++) {
            if (strcmp(fileName, UFD[currentUserID].uDir[i].fileName) == 0) {
                printf("文件已存在，请重新输入文件名！\n");
                goto LOOP;
            }
        }
        for (i = 0; i < 10; i++) {
            if (strcmp(UFD[currentUserID].uDir[i].fileName, "") == 0) {
                strcpy(UFD[currentUserID].uDir[i].fileName, fileName);
                UFD[currentUserID].uDir[i].flag = 1;
                for(int k=0; k < 3; k++)
                {
                    //默认权限为 1
                    UFD[currentUserID].uDir[i].fileProtectCode[k] = 1;
                }

                UFD[currentUserID].uDir[i].fileLength = fileLength;
                printf("请输入文件创建时间（例如：2022/07/02）: ");
                scanf("%s", fileCreateDate);
                strcpy(UFD[currentUserID].uDir[i].fileCreateDate, fileCreateDate);

                printf("请输入文件大小（KB）:\n");
                scanf_s("%d", &fileLength);
                UFD[currentUserID].uDir[i].fileLength = fileLength;
                //分配到磁盘上
                allocateDiskBlock(currentUserID, i);
                printf("文件创建成功！\n");
```

```
                break;
            }
        }
        break;
    }
}

//删除文件
void deleteFile(int currentUserID)
{
    char fileName[10];
    //初始化 char
    for(int i=0;i<10;i++)
    {
        fileName[i] = '\0';
    }
    LOOP:while(1)
    {
        printf("请输入文件名: ");
        scanf("%s", fileName);
        //如果文件正在运行则不能删除
        for(int i=0;i<5;i++)
        {
            if(strcmp(fileName, AFD[i].openFileName)==0)
            {
                printf("文件正在运行，不能删除！\n");
                goto LOOP;
            }
        }
        for (int i = 0; i < 10; i++) {
            if (strcmp(fileName, UFD[currentUserID].uDir[i].fileName) == 0) {
                freeDiskBlock(currentUserID,i);
                //将该用户名下的文件从后面移到前面
                for (int j = i; j < 9; j++) {
                    for(int k = 0; k < 10; k++)
                    {
                        UFD[currentUserID].uDir[j].fileName[k] = UFD[currentUserID]
.uDir[j+1].fileName[k];
                    }
                    UFD[currentUserID].uDir[j].flag = UFD[currentUserID].uDir[j + 1
].flag;
                    for(int k=0; k < 3; k++)
                    {
```

```
                        UFD[currentUserID].uDir[j].fileProtectCode[k] = UFD[current
UserID].uDir[j + 1].fileProtectCode[k];
                    }
                    UFD[currentUserID].uDir[j].fileLength = UFD[currentUserID].uDir
[j + 1].fileLength;
                    for(int k=0; k < 20; k++)
                    {
                        UFD[currentUserID].uDir[j].fileCreateDate[k] = UFD[currentU
serID].uDir[j + 1].fileCreateDate[k];
                    }
                    //起始位置不变，结束位置变化
                    UFD[currentUserID].uDir[j].fileEndPos = UFD[currentUserID].uDir
[j].fileStartPos + UFD[currentUserID].uDir[j].fileLength;
                }
                printf("文件删除成功！\n");
                break;
            } else if(i == 9) {
                printf("文件不存在，删除失败！\n");
            }
        }
        break;
    }
}

//打开文件
void openFile(int currentUserID)
{
    int i;
    char fileName[10];
    LOOP:while(1)
    {
        printf("请输入文件名：");
        scanf("%s", fileName);
        //文件是否已经打开
        for(i=0;i<5;i++)
        {
            if(strcmp(fileName, AFD[i].openFileName)==0)
            {
                printf("文件已经打开！\n");
                goto LOOP;
            }
        }
        //检查文件是否存在
        for (i = 0; i < 10; i++) {
```

```
            //检查权限是否允许读取文件
            if (strcmp(fileName, UFD[currentUserID].uDir[i].fileName) == 0 && UFD[c
urrentUserID].uDir[i].fileProtectCode[2] == 0) {
                printf("没有权限执行该文件！\n");
                break;
            } else if (strcmp(fileName, UFD[currentUserID].uDir[i].fileName) == 0 &
& UFD[currentUserID].uDir[i].fileProtectCode[2] == 1) {
                //将文件添加到运行文件列表中
                for (int j = 0; j < 5; j++) {
                    if (AFD[j].flag == 0) {
                        for (int k = 0; k < 10; k++) {
                            AFD[j].openFileName[k] = UFD[currentUserID].uDir[i].fil
eName[k];
                        }
                        AFD[j].flag = 1;
                        for(int k=0; k < 3; k++)
                        {
                            AFD[j].openFileProtectCode[k] = UFD[currentUserID].uDir
[i].fileProtectCode[k];
                        }
                        break;
                    } else if (j == 4) {
                        printf("运行文件列表已满，请关闭一个文件后再打开新文件！\n");
                        break;
                    }
                }
                printf("文件打开成功！\n");
                break;
            } else if(i == 9)
            {
                printf("文件不存在！\n");
                break;
            }
        }
        break;
    }
}

//关闭文件
void closeFile(int currentUserID)
{
    int i;
    char fileName[10];
    while(1)
```

```
    {
        printf("请输入文件名: ");
        scanf("%s", fileName);
        //将文件从运行文件列表中删除
        for (int j = 0; j < 5; j++) {
            if (strcmp(AFD[j].openFileName, fileName) == 0) {
                strcpy(AFD[j].openFileName, "XXX");
                AFD[j].flag = 0;
                for(int k=0;k<3;k++)
                {
                    AFD[j].openFileProtectCode[k] = 0;
                }
                AFD[j].rwPoint = 0;
                printf("文件关闭成功! \n");
                break;
            } else if (j == 4) {
                printf("文件不在运行文件列表中，请检查! \n");
                break;
            }
        }
        break;
    }
}

//读文件
void readFile()
{
    int i;
    char fileName[10];
    while(1)
    {
        printf("请输入文件名: ");
        scanf("%s", fileName);
        //从运行文件列表中查找文件
        for (i = 0; i < 5; i++) {
            if (strcmp(fileName, AFD[i].openFileName) == 0) {
                //检查权限是否允许读取文件
                if (AFD[i].openFileProtectCode[2] == 0) {
                    printf("没有权限执行该文件! \n");
                    break;
                } else if (AFD[i].openFileProtectCode[2] == 1) {
                    AFD[i].rwPoint = 1;
                    printf("文件的读写标志位已置%d! \n", AFD[i].rwPoint);
                    printf("文件读出成功! \n");
```

```
                    AFD[i].rwPoint = 0;
                    printf("文件的读写标志位已置%d！\n", AFD[i].rwPoint);
                    break;
                }
            } else if (i == 4) {
                printf("文件不在运行文件列表中，请检查！\n");
                break;
            }
        }
        break;
    }
}

//写文件
void writeFile()
{
    int i;
    char fileName[10];
    while(1)
    {
        printf("请输入文件名：");
        scanf("%s", fileName);
        //从运行文件列表中查找文件
        for (i = 0; i < 5; i++) {
            if (strcmp(fileName, AFD[i].openFileName) == 0) {
                //检查权限是否允许读取文件
                if (AFD[i].openFileProtectCode[1] == 0) {
                    printf("没有权限执行该文件！\n");
                    break;
                } else if (AFD[i].openFileProtectCode[1] == 1) {
                    AFD[i].rwPoint = 1;
                    printf("文件的读写标志位已置%d！\n", AFD[i].rwPoint);
                    printf("文件写入成功！\n");
                    AFD[i].rwPoint = 0;
                    printf("文件的读写标志位已置%d！\n", AFD[i].rwPoint);
                    break;
                }
            } else if (i == 4) {
                printf("文件不在运行文件列表中，请检查！\n");
                break;
            }
        }
        break;
    }
```

```
}
//主流程循环
int main() {
    system("chcp 65001");
    initDisk();
    initAFD();
    initData();
    int currentUserID = login();
    printf("-----------------------------%s 用户的文件系统-------------------------
---\n", UFD[currentUserID].userName);
    printf("友情提示：输入 help 可以查看可用指令哦!\n");
    while (1) {
        char command[10];
        printf("%s@simpleFileSys:~$", UFD[currentUserID].userName);
        scanf(" %s", command);
        if (strcmp(command, "create") == 0) {
            createFile(currentUserID);
        } else if (strcmp(command, "delete") == 0) {
            deleteFile(currentUserID);
        } else if (strcmp(command, "open") == 0) {
            openFile(currentUserID);
        } else if (strcmp(command, "close") == 0) {
            closeFile(currentUserID);
        } else if (strcmp(command, "read") == 0) {
            readFile();
        } else if (strcmp(command, "write") == 0) {
            writeFile();
        } else if(strcmp(command, "ls") == 0) {
            showUserFile(currentUserID);
        } else if(strcmp(command,"help") == 0) {
            showAllCommand();
        } else if(strcmp(command, "info") == 0) {
            showAllFileInfo(currentUserID);
        } else if(strcmp(command, "oinfo") == 0) {
            showOpenFileInfo(currentUserID);
        } else if(strcmp(command, "disk") == 0) {
            showDiskBlock();
        } else if (strcmp(command, "exit") == 0) {
            break;
        } else {
            printf("\n 命令错误,输入 help 可以查看命令提示！ \n\n");
        }
    }
}
```

4. 输入输出结果

刚刚登录之前初始化结束之后已经有 2 个账户 root 和 alex，每个账户下面都有 2 个文件分别是 a.txt（10KB）、b.txt（20KB）和 c.txt（30KB）、d.txt（40KB）。

这里举的例子是这样的:登录 root 账户，创建文件 e.txt，运行 e.txt，打开 e.txt,读、写 e.txt，关闭 e.txt,删除 b.txt。截图按此顺序叙述。

```
----------------------------简单文件系统登录页面----------------------------
请输入用户名: root
root
请输入密码: 1
1
登录成功!
--------------------------------------------------------------------------------


------------------------------root用户的文件系统------------------------------
友情提示: 输入help可以查看可用指令哦!
root@simpleFileSys:~$help
help
---------------------------------可用指令如下---------------------------------
1. 新建文件 -- create
2. 删除文件 -- delete
3. 打开文件 -- open
4. 关闭文件 -- close
5. 文件读写 -- read/write
6. 文件信息 -- info
7. 打开文件信息 -- oinfo
8. 文件列表 -- ls
9. 磁盘信息 -- disk
10. 退出系统 -- exit
--------------------------------------------------------------------------------

root@simpleFileSys:~$
```

图 9 登录

```
root@simpleFileSys:~$ls
ls
--------------------------------------------------------------------------------

用户root的文件列表:
a.txt
b.txt
--------------------------------------------------------------------------------

root@simpleFileSys:~$info
info
该用户所有文件的详细信息:
--------------------------------------------------------------------------------
文件名      创建时间      文件大小  读权限   写权限   执行权限  文件起始位置  文件结束位置
a.txt    2022/07/01      10      0       0       0         0         10
b.txt    2022/07/01      20      0       0       0         10        30
--------------------------------------------------------------------------------

root@simpleFileSys:~$disk
disk
磁盘块链表信息:
--------------------------------------------------------------------------------
序号      用户名      文件名      文件大小      文件开始位置      文件结束位置
 1       root       a.txt         10             0               10
 2       root       b.txt         20             10              30
 3       alex       c.txt         30             30              60
 4       alex       d.txt         40             60              100
--------------------------------------------------------------------------------
```

图 10 登陆后查看用户信息、文件信息和磁盘信息

```
root@simpleFileSys:~$create
create
请输入文件名:e.txt
 e.txt
请输入文件创建时间（例如：2022/07/02）:2022/07/04
 2022/07/04
请输入文件大小（KB）:
25
25
文件创建成功!
```

图 11 创建文件

```
用户root的文件列表:
a.txt
b.txt
e.txt
--------------------------------------------------------------------------------

root@simpleFileSys:~$info
info
该用户所有文件的详细信息:
--------------------------------------------------------------------------------
文件名      创建时间      文件大小  读权限  写权限  执行权限 文件起始位置 文件结束位置
a.txt    2022/07/01      10      0       0       0          0          10
b.txt    2022/07/01      20      0       0       0          10         30
e.txt    2022/07/04      25      1       1       1          100        125
--------------------------------------------------------------------------------

root@simpleFileSys:~$disk
disk
磁盘块链表信息:
--------------------------------------------------------------------------------
序号      用户名      文件名      文件大小      文件开始位置      文件结束位置
 1       root       a.txt         10             0               10
 2       root       b.txt         20             10              30
 3       alex       c.txt         30             30              60
 4       alex       d.txt         40             60              100
 5       root       e.txt         25             100             125
--------------------------------------------------------------------------------
```

图 12 创建文件后查看用户信息、文件信息和磁盘信息

```
root@simpleFileSys:~$open e.txt
open e.txt
请输入文件名：文件打开成功!
root@simpleFileSys:~$oinfo
oinfo
该用户所有已经打开的文件的详细信息:
-------------------------------------------------------------------------
文件名    读权限     写权限    执行权限
e.txt     1        1        1
-------------------------------------------------------------------------

root@simpleFileSys:~$read
read
请输入文件名：e.txt
e.txt
文件的读写标志位已置1!
文件读出成功!
文件的读写标志位已置0!
root@simpleFileSys:~$write
write
请输入文件名：e.txt
e.txt
文件的读写标志位已置1!
文件写入成功!
文件的读写标志位已置0!
```

图 13 打开 e.txt 并查看运行目录然后进行读写操作

```
root@simpleFileSys:~$close
close
请输入文件名：e.txt
e.txt
文件关闭成功!
root@simpleFileSys:~$oinfo
oinfo
该用户所有已经打开的文件的详细信息:
-------------------------------------------------------------------------
文件名    读权限     写权限    执行权限
-------------------------------------------------------------------------
```

图 14 关闭 e.txt 并查看运行目录

```
root@simpleFileSys:~$delete
delete
请输入文件名: b.txt
b.txt
文件删除成功!
root@simpleFileSys:~$ls
ls
--------------------------------------------------------------------------------

用户root的文件列表:
a.txt
e.txt
--------------------------------------------------------------------------------

root@simpleFileSys:~$info
info
该用户所有文件的详细信息:
--------------------------------------------------------------------------------
文件名      创建时间      文件大小  读权限  写权限  执行权限 文件起始位置 文件结束位置
a.txt    2022/07/01      10      0       0       0          0          10
e.txt    2022/07/04      25      1       1       1          10         35
--------------------------------------------------------------------------------

root@simpleFileSys:~$disk
disk
磁盘块链表信息:
--------------------------------------------------------------------------------
序号      用户名      文件名      文件大小      文件开始位置      文件结束位置
 1       root       a.txt         10            0               10
 2       alex       c.txt         30            10              40
 3       alex       d.txt         40            40              80
 4       root       e.txt         25            80              105
--------------------------------------------------------------------------------
```

图 15 删除 b.txt 并查看用户信息、文件信息和磁盘信息

```
root@simpleFileSys:~$exit
exit

进程已结束,退出代码0
```

图 16 退出程序