**实 验 报 告**

|  |  |
| --- | --- |
| **课程名称：** | **操作系统** |
| **学生姓名：** |  |
| **学生学号：** |  |
| **学生专业：** | **软件工程** |
| **开课学期：** |  |

**XXXX单位**

**20XX年XX月**

**目 录**

[实验一XXXX1 1](#_Toc449456884)

[实验二XXXX2 2](#_Toc449456885)

[实验三XXXX3 3](#_Toc449456886)

[实验四XXXX4 4](#_Toc449456887)

# Linux 常用命令使用及C程序编写

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **地 点：** | B7 331 | 房； | **评 分：** |  |
| **实验日期与时间：** |  | | **实验教师：** |  |

一、实验目的

1． 了解 Linux 一般命令格式。

2． 掌握常用命令和选项的功能。

3． 学习使用 vi 编辑器建立、编辑、显示及加工处理文本文件。

4． 掌握 Linux 下 C 程序的编写、编译与运行方法。

二、实验内容

1. 熟练使用man命令及date等简单命令。

2. 在用户主目录下对文件进行操作：复制一个文件、显示文件内容、查找指定内容、排序、文件比较、文件删除等。

3. 对目录进行管理：创建和删除子目录、改变和显示工作目录、列出和更改文件权限、链接文件等。

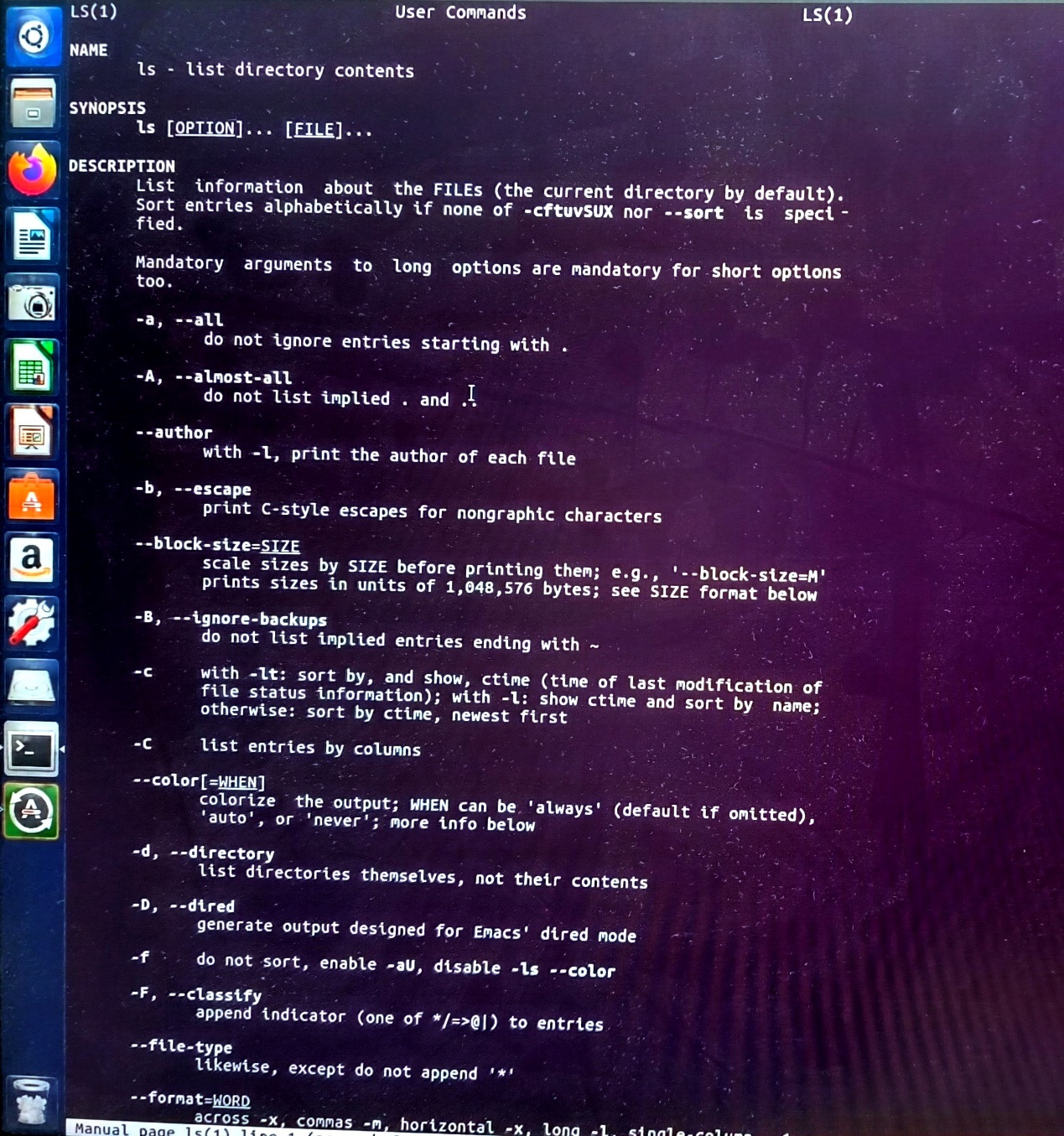
4. 使用vi 建立一个文本文件，并在该文件上移动光标位置，编写任意文件内容，然后对该文件执行删除、复原、修改、替换等操作。

5. 用vi编辑器编写一个简单的C程序，用gcc编译器编译该程序并执行。

三、实验步骤

1. 使用man显示date等命令的手册页。

2. 通过使用简单命令date, cal, who, echo, clear等，了解Linux命令格式。

如上图所示

3. 浏览文件系统

(1) 运行pwd命令，确定当前工作目录。

为/home/pc

(2) 运行ls –l命令，理解各字段含义。

1. **文件类型与权限（drwxrwxr-x）**：

**文件类型**：第一个字符 “d” 表示这是一个目录（如果是 “-” 则表示普通文件，“l” 表示符号链接，“c” 表示字符设备文件，“b” 表示块设备文件等 ）。

**权限**：从第二个字符开始，每 3 个字符一组，分别表示文件所有者（rwx）、同组用户（rwx）和其他用户（r-x）的权限。“r” 代表读权限，“w” 代表写权限，“x” 代表执行权限。 所以该目录所有者和同组用户都有读、写和执行权限，其他用户有读和执行权限，但没有写权限。

1. **链接数（3）**：表示指向该目录的硬链接数量。对于目录而言，这个数字一般大于 1 ，每创建一个子目录，父目录的链接数会增加 1，因为子目录的 “.”（表示当前目录）和父目录的 “..”（表示上级目录）都指向父目录的 inode。
2. **所有者（pc）**：拥有该目录的用户账号是 “pc”，用户对目录的权限由文件权限部分的第一组权限决定。
3. **所属组（pc）**：该目录所属的用户组是 “pc”，组内用户对目录的权限由文件权限部分的第二组权限决定。
4. **文件大小（4096）**：表示目录的大小，单位是字节。对于目录来说，这个大小主要是目录本身存储元数据（如目录项列表）占用的空间，不包含子目录和文件的大小。
5. **修改时间（4月 9 2022）**：显示该目录最后一次被修改的时间，这里是 2022 年 4 月 9 日。
6. **文件名**

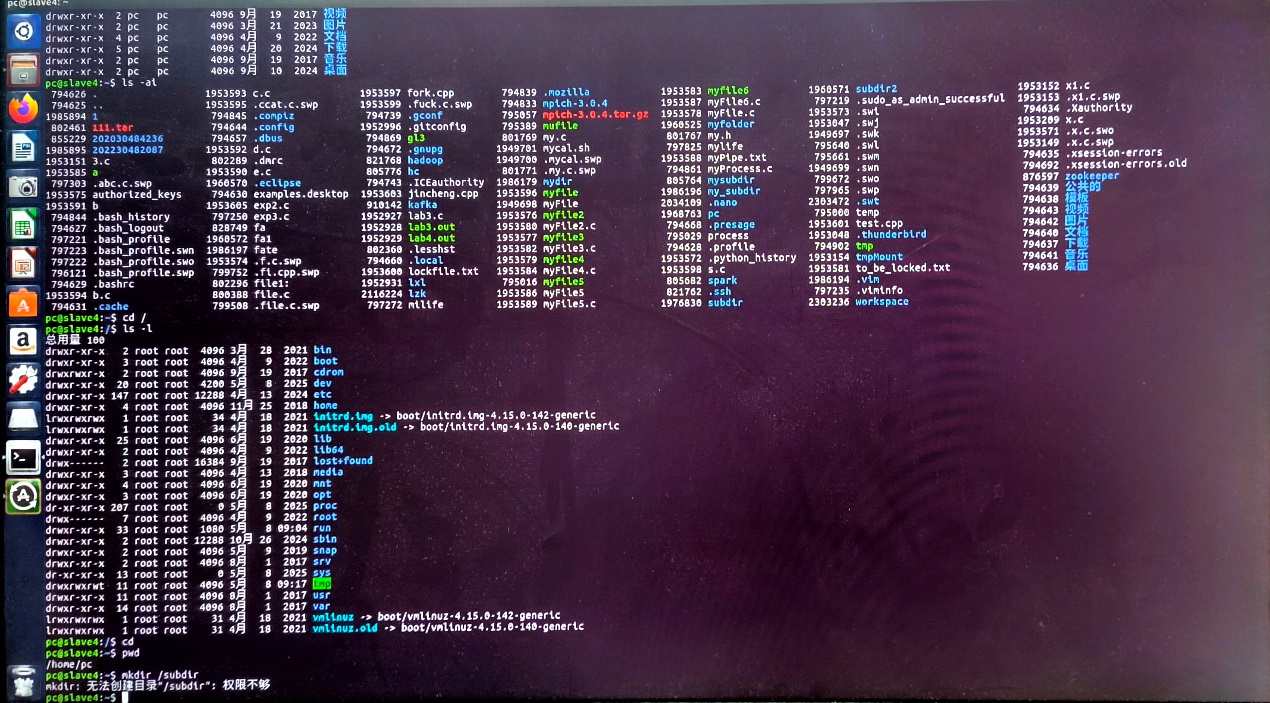


(3) 运行ls –ai命令，理解各字段含义。

(4) 使用cd命令，将工作目录改为根目录（/）。运行ls –l命令，了解各目录的作用。

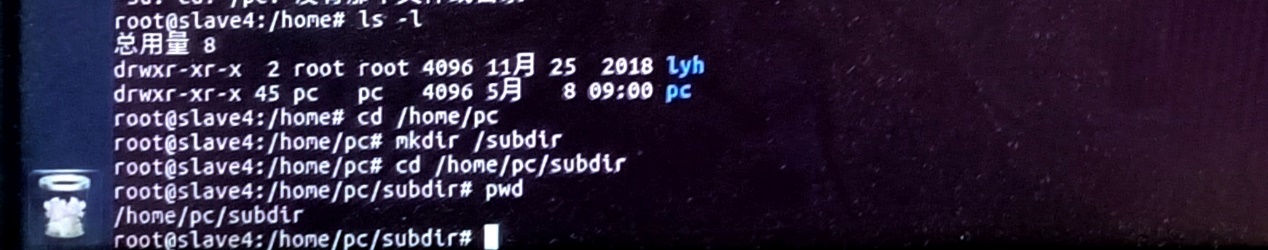
(5) 直接使用cd，回到哪里了？用pwd验证。

回到了/home/pc



(6) 用mkdir建立一个子目录subdir。

(7) 将工作目录改到subdir。



4. 文件操作

(1) 验证当前工作目录在subdir。

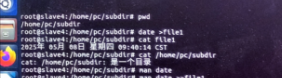
(2) 运行date > file1，然后运行cat file1，看到什么信息？

得到了文件的时间信息

(3) 运行cat subdir，会有什么结果？为什么？

发生报错，subdir显然需要是文件而不是文件夹

(4) 利用man命令显示date命令的使用说明。



(5) 运行man date >>file1，看到什么？运行cat file1，看到什么？

man date >>file1，联机帮助手册写入了file1中，附加到了刚才date信息的后面

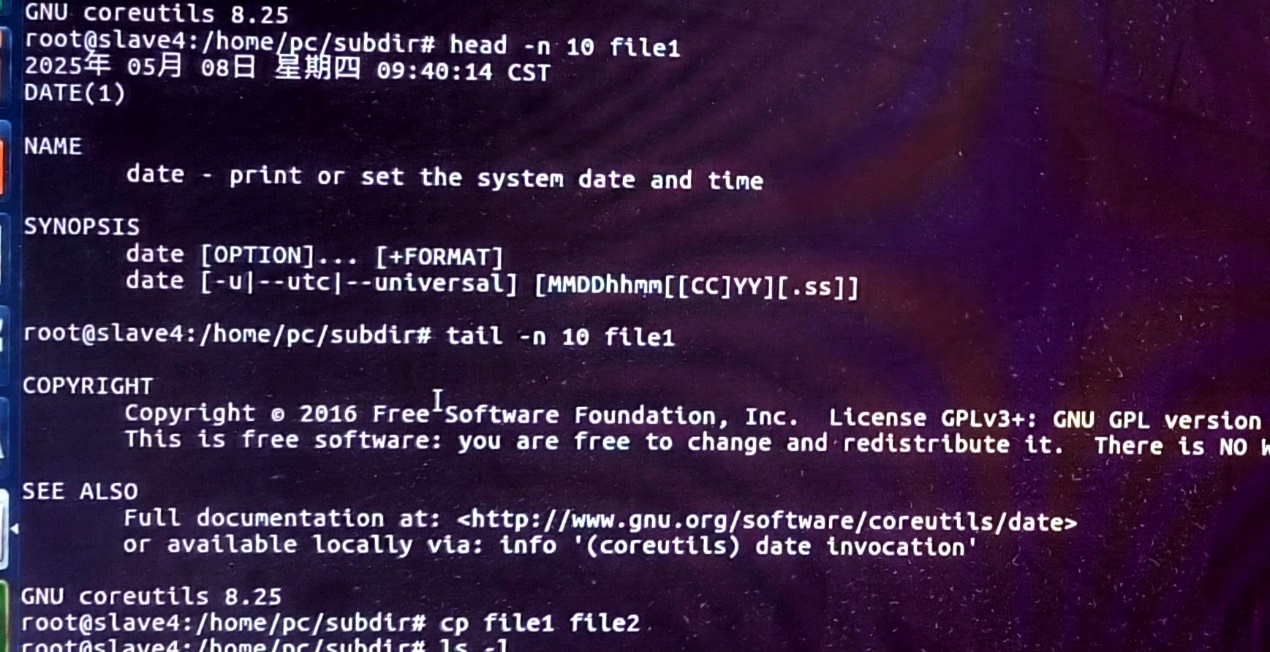
cat file1，显示文件file1中的内容

(6) 利用ls –l file1，了解链接计数是多少？运行ln file1 ../fa，再运行ls –l file1，看链接计数有无变化？用cat命令显示fa文件内容。

原本链接计数为1，运行之后的指令后变为了2，即在父目录创建了一个别名为fa的和file1相同的文件

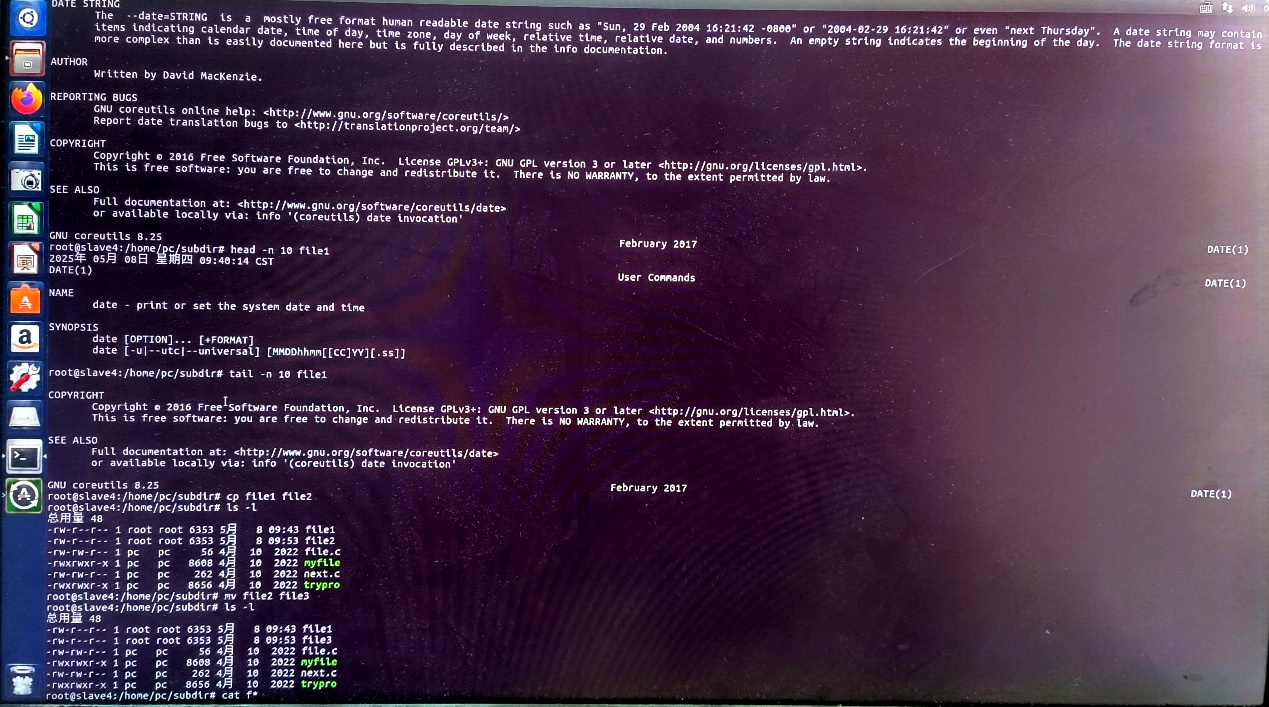


(7) 显示file1的前10行，后10行。



(8) 运行cp file1 file2，然后ls –l，看到什么？运行mv file2 file3，然后ls –l，看到什么？运行cat f\*，结果怎样？

首先发现文件发生了复制，mv后file2变为了file3，而cat f\*则打印出了文件夹中所有以f开头的文件的信息



(9) 运行rm file3，然后ls –l，结果如何？

删除了文件file3

(10)运行ls –l，理解各文件的权限是什么？

1. 文件类型标识​​

​​第一个字符​​：标识文件类型，例如：

- 普通文件（如文本、图片）

d 目录（文件夹）

l 符号链接（类似快捷方式）

其他特殊类型如 b（块设备）、c（字符设备）等

​​2. 权限分组​​

后续的 ​​9 个字符​​分为三组，每组描述不同用户的权限：

​​第一组​​（前 3 位）：文件所有者（创建者）的权限

​​第二组​​（中间 3 位）：文件所属用户组的权限

​​第三组​​（后 3 位）：其他用户的权限

每组权限的 ​​具体含义​​：

​​r​​（读取）：对文件可查看内容，对目录可列出内部文件

​​w​​（写入）：对文件可修改内容，对目录可新增/删除文件

​​x​​（执行）：对文件可运行（如脚本），对目录可进入（如 cd）

​​-​​ 表示无对应权限

​​3. 特殊权限符号​​

​​s​​：出现在执行位（x位置）时：

​​用户组权限中的 s​​：表示 setgid，目录下新建文件会继承父目录的组

​​所有者权限中的 s​​：表示 setuid，允许其他用户以所有者身份执行文件（如 /usr/bin/passwd）

​​t​​（粘滞位）：仅目录可用，用户仅能删除自己创建的文件（如 /tmp 目录）

​​4. 其他字段含义​​

​​硬链接数​​：表示文件被硬链接引用的次数（目录的硬链接数通常包含 . 和 ..）

​​所有者/所属组​​：文件属于哪个用户账户和用户组

​​文件大小​​：默认以字节显示，可用 -h 参数转为易读单位（如 KB/MB）

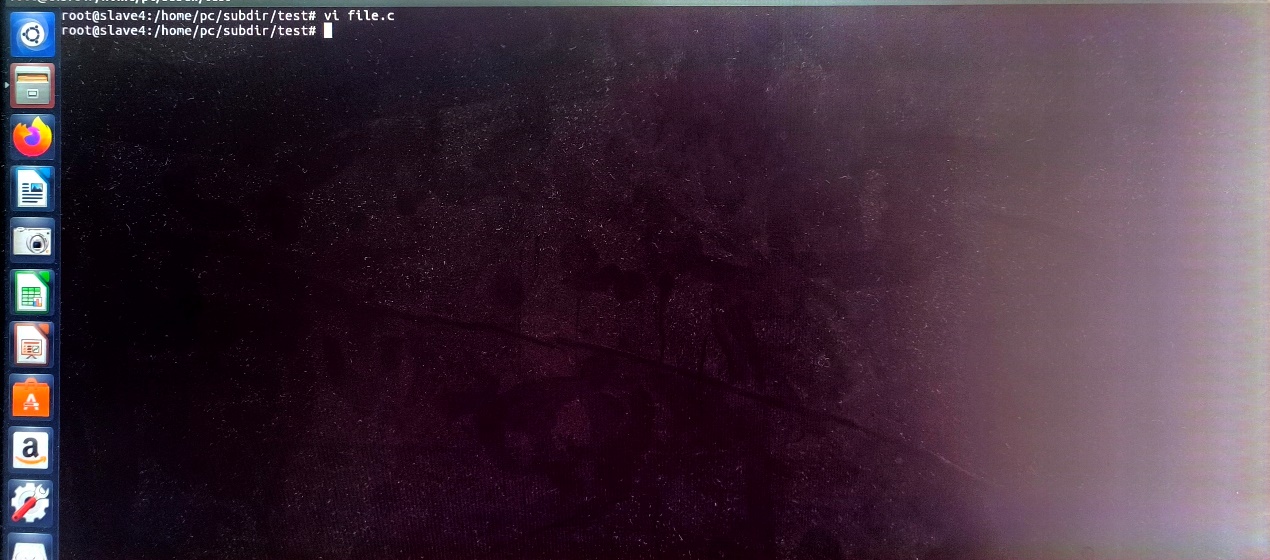
​​最后修改时间​​：文件内容最后一次被修改的时间

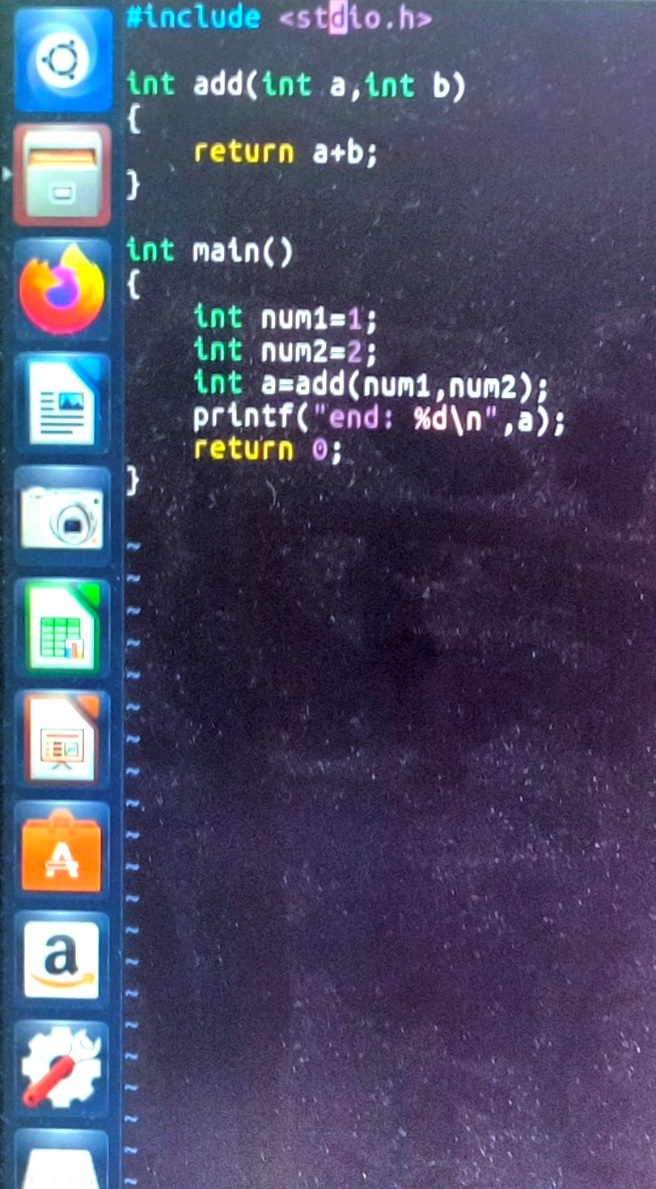
​​名称​​：文件名或目录名，符号链接会显示 -> 指向的目标路径



5. vi 编辑器

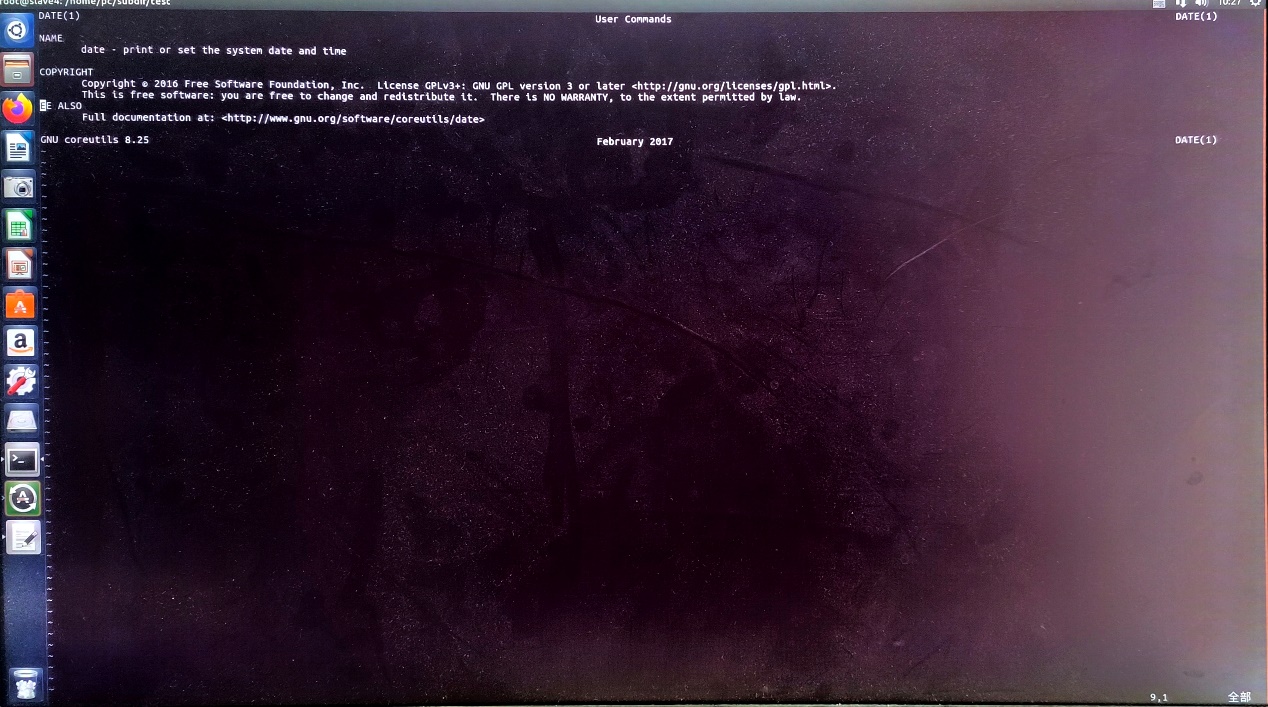
(1) 建立一个文件，如file.c。进入插入方式，输入一个C语言程序的各行内容，故意制造几处错误。最后，将该文件存盘。回到shell状态下。





(2) 运行gcc file.c –o myfile，编译该文件，会发现错误提示。理解其含义。

(3) 重新进入vi，对该文件进行修改。然后存盘，退出vi。重新编译该文件。如果编译通过了，可以用./myfile运行该程序。

(4) 运行man date > file10，然后vi file10。使用x，dd等命令删除某些文本行。使用u命令复原此前的情况。

# 进程创建及进程间通信

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **地 点：** | B7 331 楼 | 房； | **评 分：** |  |
| **实验日期与时间：** |  | | **实验教师：** |  |

一､实验目的

1. 掌握Linux 进程的创建方法,加深对进程概念的理解,明确进程和程序的区 别｡

2. 认识进程并发执行的实质｡

3. 学习控制进程同步的方法｡

4. 分析进程竞争资源的现象,学习解决进程互斥的方法｡

5. 了解管道通信的特点,掌握管道通信的使用方法｡

二､实验内容

1. 利用fork 函数创建进程｡

2. 利用wait( )来控制进程执行顺序,wait( )给我们提供了一种实现进程同步的简单方法,｡

3. 用lockf( )来给每一个进程加锁,以实现进程之间的互斥｡

4. 用pipe( )来实现进程的管道通信｡

三､实验步骤

1. 进程的创建｡编写一段C/C++程序,使用系统功能调用fork( )创建两个子进 程｡当此程序运行时,在系统中有一个父进程和两个子进程活动｡让每一个 进程在屏幕上显示一个字符;父进程显示字符“a”,子进程分别显示字符 “b”和“c”｡试观察记录屏幕上的显示结果,并分析原因｡

代码如下：

#include <stdio.h>

#include <unistd.h>

int main() {

    pid\_t pid1, pid2;

    // 创建第一个子进程

    pid1 = fork();

    if (pid1 < 0) {

        // fork() 失败

        fprintf(stderr, "Fork failed\n");

        return 1;

    } else if (pid1 == 0) {

        // 第一个子进程

        printf("b");

    } else {

        // 父进程

        // 创建第二个子进程

        pid2 = fork();

        if (pid2 < 0) {

            // fork() 失败

            fprintf(stderr, "Fork failed\n");

            return 1;

        } else if (pid2 == 0) {

            // 第二个子进程

            printf("c");

        } else {

            // 父进程

            printf("a");

        }

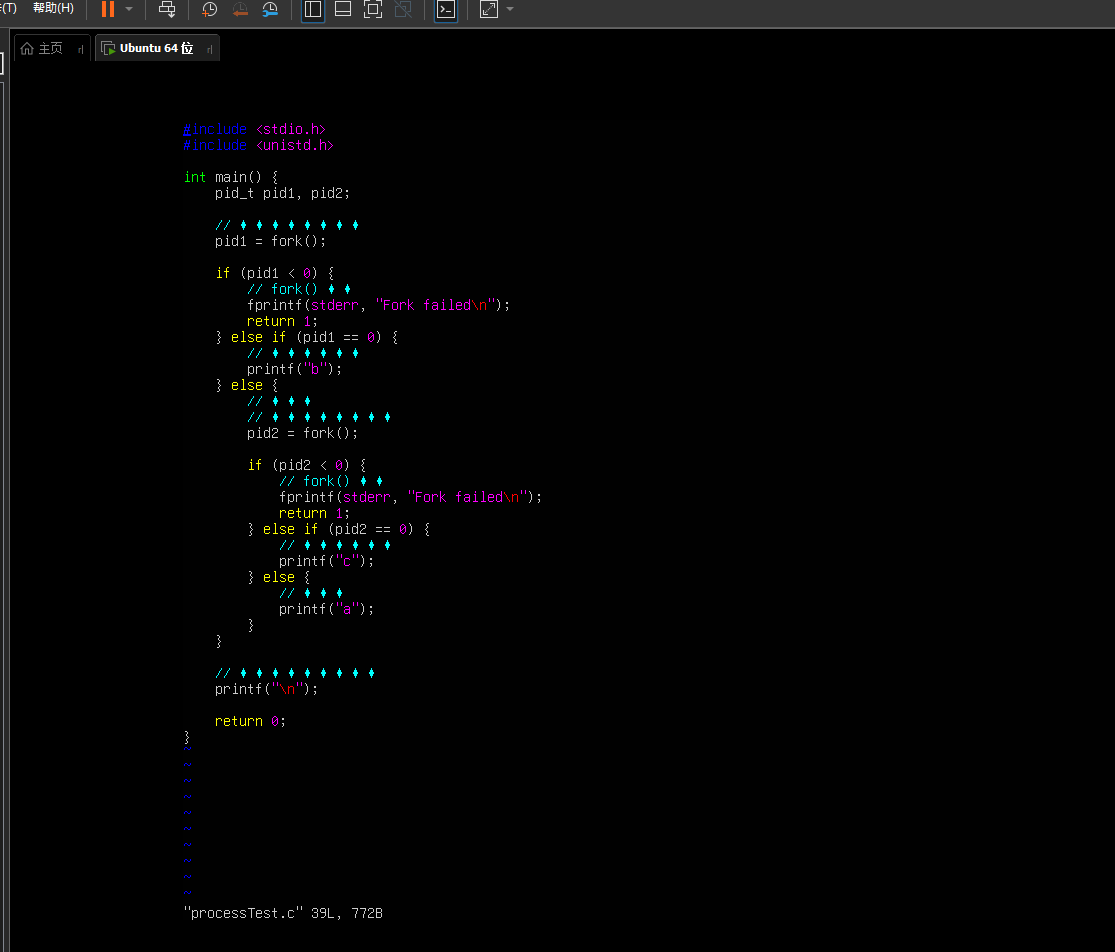
    }

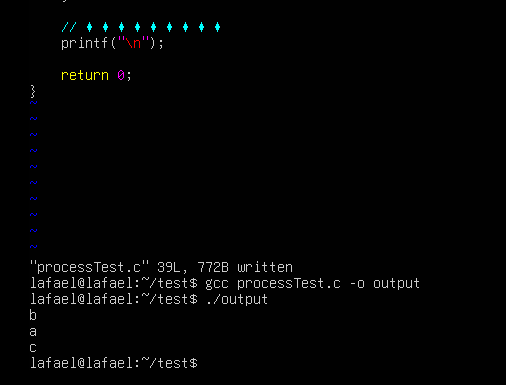
    // 换行，使输出更清晰

    printf("\n");

    return 0;

}





2. 修改1.的程序,要求子进程先输出,然后父进程再输出｡

代码如下：

#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

int main() {

    pid\_t pid1, pid2;

    int status1, status2;

    pid1 = fork();

    if (pid1 < 0) {

        fprintf(stderr, "Fork failed\n");

        return 1;

    } else if (pid1 == 0) {

        printf("b");

    } else {

        pid2 = fork();

        if (pid2 < 0) {

            fprintf(stderr, "Fork failed\n");

            return 1;

        } else if (pid2 == 0) {

            printf("c");

        } else {

            waitpid(pid1, &status1, 0);

            waitpid(pid2, &status2, 0);

            printf("a");

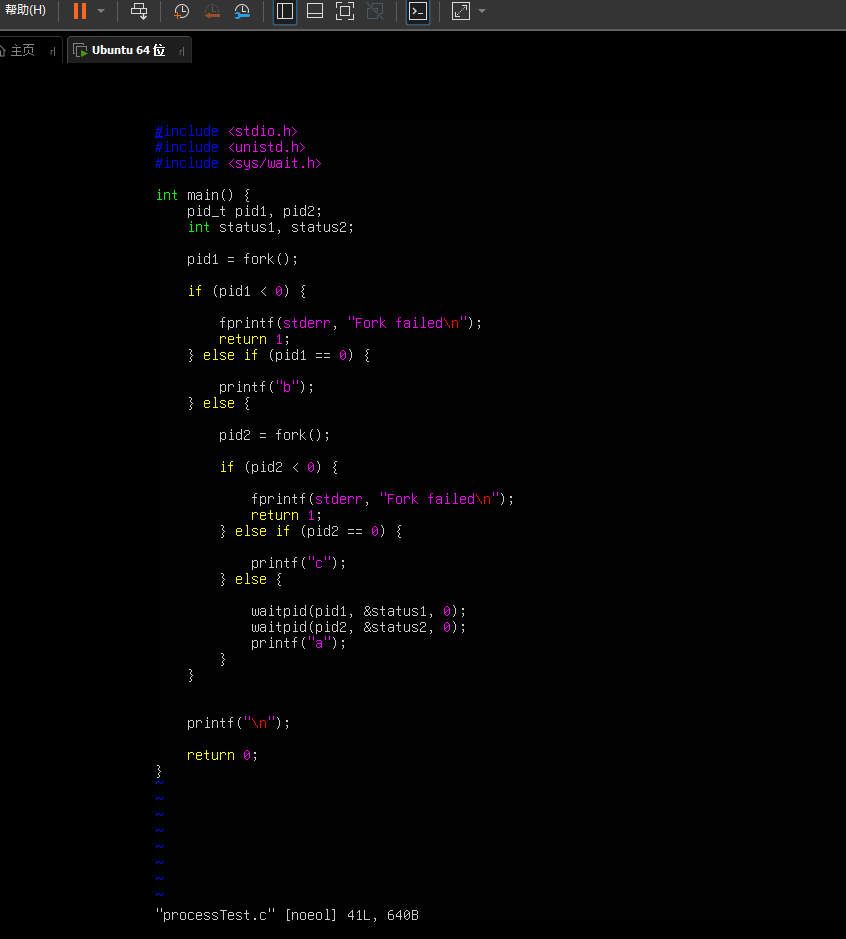
        }

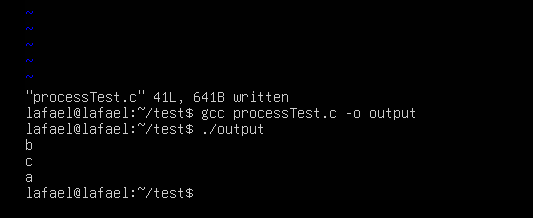
    }

    printf("\n");

    return 0;

}





3. 修改1.的程序,每一个进程循环十次,每次循环:子进程1 将“daughter i” 输出到文件to\_be\_locked.txt 文件中,子进程2 将“son i”输出到文件 to\_be\_locked.txt 文件中,父进程将“parent i”写入到文件to\_be\_locked.txt 文件中｡用lockf( )来给每一个进程加锁,以实现进程之间的互斥｡使用catto\_be\_locked.txt 查看输出结果,并分析原因｡

代码如下：

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <fcntl.h>

#define LOOP\_TIMES 10

int main() {

    pid\_t pid1, pid2;

    int fd;

    // 打开文件，如果文件不存在则创建，以写入模式打开并清空文件内容

    fd = open("to\_be\_locked.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);

    if (fd == -1) {

        perror("open");

        return 1;

    }

    // 创建第一个子进程

    pid1 = fork();

    if (pid1 < 0) {

        perror("fork");

        close(fd);

        return 1;

    } else if (pid1 == 0) {

        // 第一个子进程

        for (int i = 0; i < LOOP\_TIMES; i++) {

            // 加锁，锁定整个文件

            if (lockf(fd, F\_LOCK, 0) == -1) {

                perror("lockf");

                close(fd);

                return 1;

            }

            // 将内容写入文件

            dprintf(fd, "daughter %d\n", i);

            // 解锁

            if (lockf(fd, F\_ULOCK, 0) == -1) {

                perror("lockf");

                close(fd);

                return 1;

            }

        }

        close(fd);

        return 0;

    } else {

        // 创建第二个子进程

        pid2 = fork();

        if (pid2 < 0) {

            perror("fork");

            close(fd);

            return 1;

        } else if (pid2 == 0) {

            // 第二个子进程

            for (int i = 0; i < LOOP\_TIMES; i++) {

                // 加锁

                if (lockf(fd, F\_LOCK, 0) == -1) {

                    perror("lockf");

                    close(fd);

                    return 1;

                }

                // 将内容写入文件

                dprintf(fd, "son %d\n", i);

                // 解锁

                if (lockf(fd, F\_ULOCK, 0) == -1) {

                    perror("lockf");

                    close(fd);

                    return 1;

                }

            }

            close(fd);

            return 0;

        } else {

            // 父进程

            for (int i = 0; i < LOOP\_TIMES; i++) {

                // 加锁

                if (lockf(fd, F\_LOCK, 0) == -1) {

                    perror("lockf");

                    close(fd);

                    return 1;

                }

                // 将内容写入文件

                dprintf(fd, "parent %d\n", i);

                // 解锁

                if (lockf(fd, F\_ULOCK, 0) == -1) {

                    perror("lockf");

                    close(fd);

                    return 1;

                }

            }

            // 等待子进程结束

            waitpid(pid1, NULL, 0);

            waitpid(pid2, NULL, 0);

            close(fd);

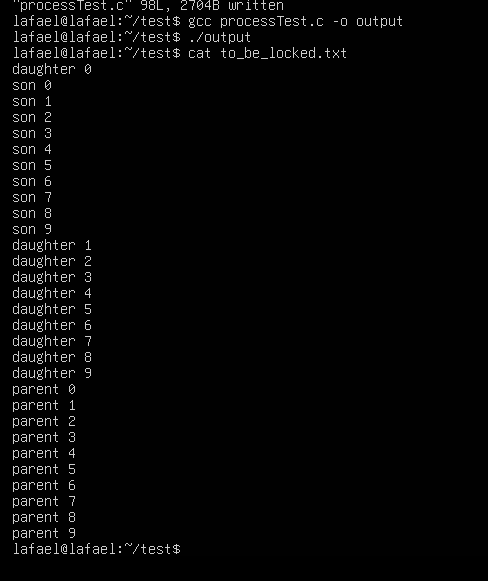
            return 0;

        }

    }

}





4. 编写程序:实现进程的管道通信,用系统调用pipe( )建立一管道,两个子进 程P1 和P2 分别向管道各写一句话:

Child 1 is sending a message!

Child 2 is sending a message!

父进程从管道中读出两个来自子进程的信息并显示(要求先接收P1,后P2)｡

代码如下：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <string.h>

#define BUFFER\_SIZE 100

int main() {

    int pipefd[2];

    pid\_t pid1, pid2;

    char buffer[BUFFER\_SIZE];

    // 创建管道

    if (pipe(pipefd) == -1) {

        perror("pipe");

        return EXIT\_FAILURE;

    }

    // 创建第一个子进程

    pid1 = fork();

    if (pid1 == -1) {

        perror("fork");

        return EXIT\_FAILURE;

    }

    if (pid1 == 0) {

        // 子进程 1

        close(pipefd[0]); // 关闭读端

        const char \*message1 = "Message from child process 1";

        write(pipefd[1], message1, strlen(message1));

        close(pipefd[1]); // 关闭写端

        exit(EXIT\_SUCCESS);

    }

    // 创建第二个子进程

    pid2 = fork();

    if (pid2 == -1) {

        perror("fork");

        return EXIT\_FAILURE;

    }

    if (pid2 == 0) {

        // 子进程 2

        close(pipefd[0]); // 关闭读端

        const char \*message2 = "Message from child process 2";

        write(pipefd[1], message2, strlen(message2));

        close(pipefd[1]); // 关闭写端

        exit(EXIT\_SUCCESS);

    }

    // 父进程

    close(pipefd[1]); // 关闭写端

    // 等待子进程写入数据

    waitpid(pid1, NULL, 0);

    waitpid(pid2, NULL, 0);

    // 从管道读取并输出子进程 1 的消息

    ssize\_t bytesRead = read(pipefd[0], buffer, BUFFER\_SIZE - 1);

    if (bytesRead > 0) {

        buffer[bytesRead] = '\0';

        printf("Received from child process 1: %s\n", buffer);

    }

    // 从管道读取并输出子进程 2 的消息

    bytesRead = read(pipefd[0], buffer, BUFFER\_SIZE - 1);

    if (bytesRead > 0) {

        buffer[bytesRead] = '\0';

        printf("Received from child process 2: %s\n", buffer);

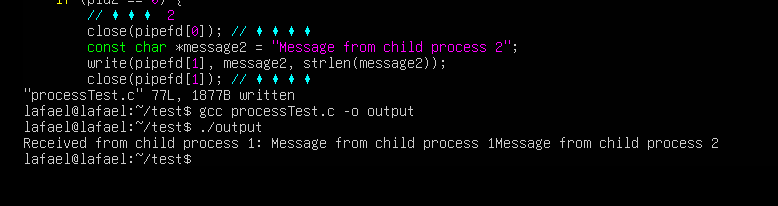
    }

    close(pipefd[0]); // 关闭读端

    return EXIT\_SUCCESS;

}





# 进程调度算法模拟实验

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **地 点：** | B7 楼 | 333 房； | **评 分：** |  |
| **实验日期与时间：** |  | | **实验教师：** |  |

**一、实验目的**

通过模拟单处理器进程调度算法，加深对进程调度的概念理解。

**二、实验内容**

模拟实现一个按下列算法进行进程调度的程序： 1. 时间片轮转法 (RR) 2. 短作业优先法 (SJF)

**三、实验要求**

1. 假设系统有 3-5 个进程，每个进程用一个进程控制块（PCB）来表示。进程控制块的格式可参考图 1。

图 1 进程控制块结构

图中的参数意义如下：进程名：即进程标识。

链接指针：按照进程到达系统的时间将处于就绪状态的进程连接成一个就绪队列。指针指出下一个到达的进程控制块地址。最后一个进程的链接指针为 NULL。

到达时间：进程创建时的系统时间或由用户指定，调度时，总是选择到达时间最早的进程。

估计运行时间：可由设计者任意指定一个时间值。

进程状态：为简单起见，这里假定进程有两种状态：就绪态和完成态。就绪状态用“R”表示，完成状态用“C”表示。假定进程一创建就处于就绪状态，运行结束时，就被置成完成状态。

1. 测试数据可以随机输入或从文件中读入。
   * 应包含显示或打印语句，轮转调度算法需显示或打印每次选中进程的名称及运行 一轮后队列的变化情况；短作业优先算法需显示或打印每次选中进程的名称及进程最终完成的序列。

最终能够计算每一个进程的周转时间或等待时间

实验代码如下：

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// 定义进程状态枚举

typedef enum { READY, COMPLETED } State;

// 进程控制块（PCB）结构体

typedef struct PCB {

    char name[20];        // 进程名

    int arriveTime;       // 到达时间

    int burstTime;        // 估计运行时间

    int remainingTime;    // 剩余运行时间（用于 RR 算法）

    int startTime;        // 开始时间

    int finishTime;       // 完成时间

    int turnaroundTime;   // 周转时间

    int weightedTT;       // 带权周转时间

    State state;          // 进程状态

    struct PCB\* next;     // 指向下一个PCB的指针

} PCB;

// 就绪队列结构体

typedef struct {

    PCB\* head;

    PCB\* tail;

} ReadyQueue;

// 初始化就绪队列

void initQueue(ReadyQueue\* queue) {

    queue->head = NULL;

    queue->tail = NULL;

}

// 判断队列是否为空

int isEmpty(ReadyQueue\* queue) {

    return queue->head == NULL;

}

// 入队操作

void enqueue(ReadyQueue\* queue, PCB\* pcb) {

    pcb->next = NULL;

    if (isEmpty(queue)) {

        queue->head = pcb;

        queue->tail = pcb;

    }

    else {

        queue->tail->next = pcb;

        queue->tail = pcb;

    }

}

// 出队操作

PCB\* dequeue(ReadyQueue\* queue) {

    if (isEmpty(queue)) return NULL;

    PCB\* temp = queue->head;

    queue->head = queue->head->next;

    if (queue->head == NULL) queue->tail = NULL;

    return temp;

}

// 创建新进程

PCB\* createProcess(const char\* name, int arriveTime, int burstTime) {

    PCB\* newProcess = (PCB\*)malloc(sizeof(PCB));

    strcpy\_s(newProcess->name, name);

    newProcess->arriveTime = arriveTime;

    newProcess->burstTime = burstTime;

    newProcess->remainingTime = burstTime;

    newProcess->startTime = -1;

    newProcess->finishTime = 0;

    newProcess->turnaroundTime = 0;

    newProcess->weightedTT = 0;

    newProcess->state = READY;

    newProcess->next = NULL;

    return newProcess;

}

int compareByArrival(const void\* a, const void\* b) {

    PCB\* pcbA = \*(PCB\*\*)a;

    PCB\* pcbB = \*(PCB\*\*)b;

    return pcbA->arriveTime - pcbB->arriveTime;

}

// 时间片轮转法（RR）

void roundRobin(PCB\* processes[], int n, int timeQuantum) {

    qsort(processes, n, sizeof(PCB\*), compareByArrival);

    ReadyQueue readyQueue;

    initQueue(&readyQueue);

    int currentTime = 0;

    int completed = 0;

    int nextProcess = 0;

    printf("\n===== 时间片轮转法（RR）调度过程 =====\n");

    printf("时间片大小: %d\n\n", timeQuantum);

    printf("时间\t执行进程\t剩余时间\n");

    while (completed < n) {

        // 将到达的进程加入就绪队列

        while (nextProcess < n && processes[nextProcess]->arriveTime <= currentTime) {

            enqueue(&readyQueue, processes[nextProcess]);

            nextProcess++;

        }

        if (isEmpty(&readyQueue)) {

            currentTime++;

            continue;

        }

        // 取出队首进程执行一个时间片

        PCB\* currentProcess = dequeue(&readyQueue);

        // 记录首次执行时间

        if (currentProcess->startTime == -1) {

            currentProcess->startTime = currentTime;

        }

        // 执行一个时间片

        printf("%d\t%s\t\t%d\n", currentTime, currentProcess->name, currentProcess->remainingTime);

        int executionTime = (currentProcess->remainingTime < timeQuantum) ?

            currentProcess->remainingTime : timeQuantum;

        currentProcess->remainingTime -= executionTime;

        currentTime += executionTime;

        // 将到达的进程加入就绪队列

        while (nextProcess < n && processes[nextProcess]->arriveTime <= currentTime) {

            enqueue(&readyQueue, processes[nextProcess]);

            nextProcess++;

        }

        // 检查进程是否完成

        if (currentProcess->remainingTime == 0) {

            currentProcess->finishTime = currentTime;

            currentProcess->turnaroundTime = currentProcess->finishTime - currentProcess->arriveTime;

            currentProcess->weightedTT = currentProcess->turnaroundTime / currentProcess->burstTime;

            currentProcess->state = COMPLETED;

            completed++;

            printf("%d\t%s\t\t完成\n", currentTime, currentProcess->name);

        }

        else {

            enqueue(&readyQueue, currentProcess);

        }

    }

    // 计算平均周转时间和平均带权周转时间

    float avgTurnaround = 0, avgWeightedTT = 0;

    printf("\n===== RR 调度结果统计 =====\n");

    printf("进程名\t到达时间\t运行时间\t完成时间\t周转时间\t带权周转时间\n");

    for (int i = 0; i < n; i++) {

        printf("%s\t%d\t\t%d\t\t%d\t\t%d\t\t%.2f\n",

            processes[i]->name,

            processes[i]->arriveTime,

            processes[i]->burstTime,

            processes[i]->finishTime,

            processes[i]->turnaroundTime,

            (float)processes[i]->turnaroundTime / processes[i]->burstTime);

        avgTurnaround += processes[i]->turnaroundTime;

        avgWeightedTT += (float)processes[i]->turnaroundTime / processes[i]->burstTime;

    }

    printf("\n平均周转时间: %.2f\n", avgTurnaround / n);

    printf("平均带权周转时间: %.2f\n", avgWeightedTT / n);

}

// 比较函数：按到达时间排序

// 比较函数：按剩余运行时间排序（用于 SJF）

int compareByRemainingTime(const void\* a, const void\* b) {

    PCB\* pcbA = \*(PCB\*\*)a;

    PCB\* pcbB = \*(PCB\*\*)b;

    return pcbA->remainingTime - pcbB->remainingTime;

}

// 短作业优先法（SJF）

void shortestJobFirst(PCB\* processes[], int n) {

    // 按到达时间排序

    qsort(processes, n, sizeof(PCB\*), compareByArrival);

    int currentTime = 0;

    int completed = 0;

    int nextProcess = 0;

    printf("\n===== 短作业优先法（SJF）调度过程 =====\n\n");

    printf("时间\t执行进程\t剩余时间\n");

    while (completed < n) {

        // 找出当前已到达且剩余时间最短的进程

        PCB\* shortestJob = NULL;

        int shortestIndex = -1;

        for (int i = 0; i < n; i++) {

            if (processes[i]->state == READY &&

                processes[i]->arriveTime <= currentTime &&

                (shortestJob == NULL || processes[i]->remainingTime < shortestJob->remainingTime)) {

                shortestJob = processes[i];

                shortestIndex = i;

            }

        }

        if (shortestJob == NULL) {

            currentTime++;

            continue;

        }

        // 记录首次执行时间

        if (shortestJob->startTime == -1) {

            shortestJob->startTime = currentTime;

        }

        // 执行该进程直到完成

        printf("%d\t%s\t\t%d\n", currentTime, shortestJob->name, shortestJob->remainingTime);

        currentTime += shortestJob->remainingTime;

        shortestJob->remainingTime = 0;

        shortestJob->finishTime = currentTime;

        shortestJob->turnaroundTime = shortestJob->finishTime - shortestJob->arriveTime;

        shortestJob->weightedTT = shortestJob->turnaroundTime / shortestJob->burstTime;

        shortestJob->state = COMPLETED;

        completed++;

        printf("%d\t%s\t\t完成\n", currentTime, shortestJob->name);

    }

    // 计算平均周转时间和平均带权周转时间

    float avgTurnaround = 0, avgWeightedTT = 0;

    printf("\n===== SJF 调度结果统计 =====\n");

    printf("进程名\t到达时间\t运行时间\t完成时间\t周转时间\t带权周转时间\n");

    for (int i = 0; i < n; i++) {

        printf("%s\t%d\t\t%d\t\t%d\t\t%d\t\t%.2f\n",

            processes[i]->name,

            processes[i]->arriveTime,

            processes[i]->burstTime,

            processes[i]->finishTime,

            processes[i]->turnaroundTime,

            (float)processes[i]->turnaroundTime / processes[i]->burstTime);

        avgTurnaround += processes[i]->turnaroundTime;

        avgWeightedTT += (float)processes[i]->turnaroundTime / processes[i]->burstTime;

    }

    printf("\n平均周转时间: %.2f\n", avgTurnaround / n);

    printf("平均带权周转时间: %.2f\n", avgWeightedTT / n);

}

// 打印初始进程信息

void printInitialProcessInfo(PCB\* processes[], int n) {

    printf("\n===== 初始进程信息 =====\n");

    printf("进程名\t到达时间\t运行时间\n");

    for (int i = 0; i < n; i++) {

        printf("%s\t%d\t\t%d\n",

            processes[i]->name,

            processes[i]->arriveTime,

            processes[i]->burstTime);

    }

}

int main() {

    // 测试数据

    int n = 5;

    PCB\* processes[5];

    processes[0] = createProcess("P1", 5, 4);

    processes[1] = createProcess("P2", 4, 6);

    processes[2] = createProcess("P3", 3, 3);

    processes[3] = createProcess("P4", 0, 6);

    processes[4] = createProcess("P5", 0, 2);

    printInitialProcessInfo(processes, n);

    // 复制进程数组用于两种算法测试

    PCB\* rrProcesses[5];

    PCB\* sjfProcesses[5];

    for (int i = 0; i < n; i++) {

        rrProcesses[i] = createProcess(processes[i]->name,

            processes[i]->arriveTime,

            processes[i]->burstTime);

        sjfProcesses[i] = createProcess(processes[i]->name,

            processes[i]->arriveTime,

            processes[i]->burstTime);

    }

    // 执行 RR 算法

    roundRobin(rrProcesses, n, 2);

    // 重置进程状态

    for (int i = 0; i < n; i++) {

        rrProcesses[i]->remainingTime = rrProcesses[i]->burstTime;

        rrProcesses[i]->startTime = -1;

        rrProcesses[i]->finishTime = 0;

        rrProcesses[i]->turnaroundTime = 0;

        rrProcesses[i]->weightedTT = 0;

        rrProcesses[i]->state = READY;

    }

    // 执行 SJF 算法

    shortestJobFirst(sjfProcesses, n);

    // 释放内存

    for (int i = 0; i < n; i++) {

        free(processes[i]);

        free(rrProcesses[i]);

        free(sjfProcesses[i]);

    }

    return 0;

}

有结果：

===== 初始进程信息 =====

进程名  到达时间        运行时间

P1      5               4

P2      4               6

P3      3               3

P4      0               6

P5      0               2

===== 时间片轮转法（RR）调度过程 =====

时间片大小: 2

时间    执行进程        剩余时间

0       P4              6

2       P5              2

4       P5              完成

4       P4              4

6       P3              3

8       P2              6

10      P1              4

12      P4              2

14      P4              完成

14      P3              1

15      P3              完成

15      P2              4

17      P1              2

19      P1              完成

19      P2              2

21      P2              完成

===== RR 调度结果统计 =====

进程名  到达时间        运行时间        完成时间        周转时间        带权周转时间

P4      0               6               14              14              2.33

P5      0               2               4               4               2.00

P3      3               3               15              12              4.00

P2      4               6               21              17              2.83

P1      5               4               19              14              3.50

平均周转时间: 12.20

平均带权周转时间: 2.93

===== 短作业优先法（SJF）调度过程 =====

时间    执行进程        剩余时间

0       P5              2

2       P5              完成

2       P4              6

8       P4              完成

8       P3              3

11      P3              完成

11      P1              4

15      P1              完成

15      P2              6

21      P2              完成

===== SJF 调度结果统计 =====

进程名  到达时间        运行时间        完成时间        周转时间        带权周转时间

P4      0               6               8               8               1.33

P5      0               2               2               2               1.00

P3      3               3               11              8               2.67

P2      4               6               21              17              2.83

P1      5               4               15              10              2.50

平均周转时间: 9.00

平均带权周

# 存储管理

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **地 点：** | B7 楼 | 331 房； | **评 分：** |  |
| **实验日期与时间：** |  | | **实验教师：** |  |

1. 实验目的

页式管理是一种常用的虚拟存储管理技术，本实验的目的是通过模拟页式存储管理中的页面替换算法，了解虚拟存储技术的特点，认识操作系统对内存的管理。

1. 实验内容

设计一个虚拟存储区和内存工作区，并使用下述页面替换算法计算页面访问的命中率。其中，命中率＝页面命中次数／页面访问序列长度，或者命中率＝１－页面失效次数／页面访问序列长度。

1. 先进先出算法（FIFO）

2. 最近最少使用算法（LRU）

1. 实验要求

本实验要求编程模拟一个拥有若干个虚页的进程在给定的若干个实页中运行、并在缺页中断发生时分别使用 FIFO 和 LRU 算法进行页面置换的情形。其中虚页的个数可以事先给定（例如 10 个），对这些虚页进行访问的页面访问序列（其长度可以事先给定，例如 20 次虚页访问）可以由程序随机产生，也可以事先保存在文件中。要求程序运行时屏幕能显示出置换过程中的状态信息并输出访问结束时的页面命中率。程序应允许通过为该进程分配不同的实页数，来比较两种置换算法的稳定性。

实验代码：

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctime>

#define MAX\_VIRTUAL\_PAGES 10  // 最大虚页数

#define MAX\_ACCESS\_TIMES 20   // 最大访问次数

#define MAX\_FRAMES 6          // 最大实页数

// 生成随机页面访问序列（0~virtual\_pages-1）

void generate\_access\_sequence(int\* sequence, int length, int virtual\_pages) {

    for (int i = 0; i < length; i++) {

        sequence[i] = rand() % virtual\_pages;

    }

}

// 打印页面序列

void print\_sequence(int\* sequence, int length) {

    printf("页面访问序列: ");

    for (int i = 0; i < length; i++) {

        printf("%d ", sequence[i]);

    }

    printf("\n");

}

// FIFO 页面置换算法

float fifo\_algorithm(int\* sequence, int length, int frames) {

    int page\_table[MAX\_FRAMES];

    int front = 0, rear = 0, count = 0;

    int page\_faults = 0;

    int hit = 0;

    memset(page\_table, -1, sizeof(page\_table));

    printf("\nFIFO 算法（实页数=%d）:\n", frames);

    for (int i = 0; i < length; i++) {

        int current\_page = sequence[i];

        int found = 0;

        // 检查当前页是否在页表中

        for (int j = 0; j < count; j++) {

            int idx = (front + j) % frames;

            if (page\_table[idx] == current\_page) {

                found = 1;

                hit++;

                break;

            }

        }

        if (!found) {

            page\_faults++;

            if (count < frames) {

                // 队列未满，直接插入队尾

                page\_table[rear] = current\_page;

                rear = (rear + 1) % frames;

                count++;

            }

            else {

                // 队列已满，弹出队头，插入队尾

                page\_table[front] = current\_page;

                front = (front + 1) % frames;

                rear = (rear + 1) % frames;

            }

        }

        // 输出当前页表状态

        printf("访问页%d: ", current\_page);

        for (int j = 0; j < count; j++) {

            int idx = (front + j) % frames;

            printf("%d ", page\_table[idx]);

        }

        for (int j = count; j < frames; j++) {

            printf("- ");

        }

        printf("%s\n", found ? "命中" : "缺页");

    }

    return 1.0 - (float)page\_faults / length;

}

// LRU 页面置换算法

float lru\_algorithm(int\* sequence, int length, int frames) {

    int page\_table[MAX\_FRAMES];

    int last\_used[MAX\_FRAMES];

    int page\_faults = 0;

    int hit = 0;

    memset(page\_table, -1, sizeof(page\_table));

    memset(last\_used, 0, sizeof(last\_used));

    printf("\nLRU 算法（实页数=%d）:\n", frames);

    for (int i = 0; i < length; i++) {

        int current\_page = sequence[i];

        int found = 0;

        // 检查当前页是否在页表中

        for (int j = 0; j < frames; j++) {

            if (page\_table[j] == current\_page) {

                found = 1;

                hit++;

                last\_used[j] = i; // 更新最后使用时间

                break;

            }

        }

        if (!found) {

            page\_faults++;

            // 查找空位

            int empty = -1;

            for (int j = 0; j < frames; j++) {

                if (page\_table[j] == -1) {

                    empty = j;

                    break;

                }

            }

            if (empty != -1) {

                page\_table[empty] = current\_page;

                last\_used[empty] = i;

            }

            else {

                // 找到最久未使用的页面

                int lru = 0;

                for (int j = 1; j < frames; j++) {

                    if (last\_used[j] < last\_used[lru]) {

                        lru = j;

                    }

                }

                page\_table[lru] = current\_page;

                last\_used[lru] = i;

            }

        }

        // 输出当前页表状态

        printf("访问页%d: ", current\_page);

        for (int j = 0; j < frames; j++) {

            if (page\_table[j] != -1) {

                printf("%d ", page\_table[j]);

            }

            else {

                printf("- ");

            }

        }

        printf("%s\n", found ? "命中" : "缺页");

    }

    return 1.0 - (float)page\_faults / length;

}

int main() {

    int virtual\_pages = 10;  // 虚页数

    int access\_length = 20;  // 访问序列长度

    int frames;             // 实页数

    int access\_sequence[MAX\_ACCESS\_TIMES];

    // 初始化随机数种子

    srand(time(NULL));

    // 生成随机页面访问序列

    generate\_access\_sequence(access\_sequence, access\_length, virtual\_pages);

    print\_sequence(access\_sequence, access\_length);

    //int access\_sequence[MAX\_ACCESS\_TIMES] = { 1,2,4,5,9,9,9,4,0,4,8,3,0,0,8,9,9,4,8,0 };

    // 测试不同实页数下的算法性能

    for (frames = 3; frames <= MAX\_FRAMES; frames++) {

        float fifo\_hit = fifo\_algorithm(access\_sequence, access\_length, frames);

        float lru\_hit = lru\_algorithm(access\_sequence, access\_length, frames);

        printf("\n实页数=%d 时，FIFO 命中率=%.2f%%，LRU 命中率=%.2f%%\n",

            frames, fifo\_hit \* 100, lru\_hit \* 100);

    }

    return 0;

}

实例结果：

页面访问序列: 2 5 1 4 1 9 6 3 6 3 2 3 8 7 7 8 6 3 2 9

FIFO 算法（实页数=5）:

访问页2: 2 - - - - 缺页

访问页5: 2 5 - - - 缺页

访问页1: 2 5 1 - - 缺页

访问页4: 2 5 1 4 - 缺页

访问页1: 2 5 1 4 - 命中

访问页9: 2 5 1 4 9 缺页

访问页6: 5 1 4 9 6 缺页

访问页3: 1 4 9 6 3 缺页

访问页6: 1 4 9 6 3 命中

访问页3: 1 4 9 6 3 命中

访问页2: 4 9 6 3 2 缺页

访问页3: 4 9 6 3 2 命中

访问页8: 9 6 3 2 8 缺页

访问页7: 6 3 2 8 7 缺页

访问页7: 6 3 2 8 7 命中

访问页8: 6 3 2 8 7 命中

访问页6: 6 3 2 8 7 命中

访问页3: 6 3 2 8 7 命中

访问页2: 6 3 2 8 7 命中

访问页9: 3 2 8 7 9 缺页

LRU 算法（实页数=5）:

访问页2: 2 - - - - 缺页

访问页5: 2 5 - - - 缺页

访问页1: 2 5 1 - - 缺页

访问页4: 2 5 1 4 - 缺页

访问页1: 2 5 1 4 - 命中

访问页9: 2 5 1 4 9 缺页

访问页6: 6 5 1 4 9 缺页

访问页3: 6 3 1 4 9 缺页

访问页6: 6 3 1 4 9 命中

访问页3: 6 3 1 4 9 命中

访问页2: 6 3 1 2 9 缺页

访问页3: 6 3 1 2 9 命中

访问页8: 6 3 8 2 9 缺页

访问页7: 6 3 8 2 7 缺页

访问页7: 6 3 8 2 7 命中

访问页8: 6 3 8 2 7 命中

访问页6: 6 3 8 2 7 命中

访问页3: 6 3 8 2 7 命中

访问页2: 6 3 8 2 7 命中

访问页9: 6 3 8 2 9 缺页

实页数=5 时，FIFO 命中率=45.00%，LRU 命中率=45.00%

# 简单文件系统模拟

一、实验目的

1. 理解文件存储空间的管理、文件的物理结构和目录结构以及文件操作的实现。

2. 加深对文件系统内部功能和实现过程的理解。

二、实验内容

模拟实现一个简单的、类 Unix 文件系统。

三、实验要求

1. 设计一个简单的二级文件系统，要求可以实现下列几条命令：

（1）login 用户登录

（2）ls 列文件目录

（3）create 创建文件

（4）delete 删除文件

（5）open 打开文件

（6）close 关闭文件

（7）read 读文件

（8）write 写文件

2. 假设这个文件系统工作在一个 1GB 的磁盘上，文件分配方式参照课内所讲的Unix 文件系统，文件块大小为 1KB，空闲块使用链表管理。

代码如下：

#define MAX\_BLOCKS 1024\*1024       // 1GB磁盘，1KB块大小

#define MAX\_FILES 1000             // 最大文件数

#define MAX\_OPEN\_FILES 50          // 最大打开文件数

#define MAX\_USERS 10               // 最大用户数

#define MAX\_FILENAME 32            // 文件名最大长度

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

// 文件类型

typedef enum { REGULAR, DIRECTORY } FileType;

// 文件控制块

typedef struct {

    char filename[MAX\_FILENAME];

    FileType type;

    int size;                      // 文件大小（字节）

    int block\_count;              // 已分配的块数量

    int blocks[128];              // 直接指向数据块编号

    int owner\_uid;

} FCB;

// 用户信息

typedef struct {

    char username[20];

    int valid;

} User;

// 打开文件表项

typedef struct {

    FCB\* fcb;

    int pos;                       // 当前读写位置

    int in\_use;

} OpenFile;

// 超级块

typedef struct {

    int free\_blocks\_count;

    int free\_blocks\_head;

    int next\_free\_block;

} SuperBlock;

// 全局变量

User users[MAX\_USERS];            // 用户列表

FCB fcbs[MAX\_FILES];             // 文件控制块表

OpenFile open\_files[MAX\_OPEN\_FILES]; // 打开文件表

SuperBlock super\_block;          // 超级块

char disk[MAX\_BLOCKS][1024];     // 模拟磁盘

int current\_uid = -1;            // 当前登录用户ID

// 初始化空闲块链表

void init\_disk() {

    for (int i = 0; i < MAX\_BLOCKS; i++) {

        \*(int\*)disk[i] = i + 1;   // 每个块前4字节保存下一个空闲块编号

    }

    super\_block.free\_blocks\_head = 0;

    super\_block.next\_free\_block = 0;

    super\_block.free\_blocks\_count = MAX\_BLOCKS;

}

// 分配一个空闲块

int alloc\_block() {

    if (super\_block.free\_blocks\_count == 0) return -1;

    int block\_num = super\_block.next\_free\_block;

    super\_block.next\_free\_block = \*(int\*)disk[block\_num];

    super\_block.free\_blocks\_count--;

    return block\_num;

}

// 释放一个数据块

void free\_block(int block\_num) {

    \*(int\*)disk[block\_num] = super\_block.next\_free\_block;

    super\_block.next\_free\_block = block\_num;

    super\_block.free\_blocks\_count++;

}

// 创建文件

int create\_file(char\* filename) {

    // 查找空闲FCB

    int fcb\_index = -1;

    for (int i = 0; i < MAX\_FILES; i++) {

        if (!fcbs[i].filename[0]) {

            fcb\_index = i;

            break;

        }

    }

    if (fcb\_index == -1) return -1; // 无可用FCB

    strncpy(fcbs[fcb\_index].filename, filename, MAX\_FILENAME);

    fcbs[fcb\_index].type = REGULAR;

    fcbs[fcb\_index].size = 0;

    fcbs[fcb\_index].block\_count = 0;

    fcbs[fcb\_index].owner\_uid = current\_uid;

    return fcb\_index;

}

// 删除文件

int delete\_file(char\* filename) {

    for (int i = 0; i < MAX\_FILES; i++) {

        if (strcmp(fcbs[i].filename, filename) == 0 &&

            fcbs[i].owner\_uid == current\_uid) {

            // 释放所有数据块

            for (int j = 0; j < fcbs[i].block\_count; j++) {

                if (fcbs[i].blocks[j] != -1) {

                    free\_block(fcbs[i].blocks[j]);

                }

            }

            // 清除FCB

            memset(&fcbs[i], 0, sizeof(FCB));

            return 0;

        }

    }

    return -1;

}

// 打开文件

int open\_file(char\* filename) {

    for (int i = 0; i < MAX\_FILES; i++) {

        if (strcmp(fcbs[i].filename, filename) == 0 &&

            fcbs[i].owner\_uid == current\_uid) {

            // 在打开文件表中查找空位

            for (int j = 0; j < MAX\_OPEN\_FILES; j++) {

                if (!open\_files[j].in\_use) {

                    open\_files[j].fcb = &fcbs[i];

                    open\_files[j].pos = 0;

                    open\_files[j].in\_use = 1;

                    return j;

                }

            }

            return -1; // 打开文件数已达上限

        }

    }

    return -1; // 文件不存在

}

// 关闭文件

int close\_file(int fd) {

    if (fd < 0 || fd >= MAX\_OPEN\_FILES || !open\_files[fd].in\_use)

        return -1;

    open\_files[fd].in\_use = 0;

    open\_files[fd].fcb = NULL;

    open\_files[fd].pos = 0;

    return 0;

}

// 读取文件

int read\_file(int fd, char\* buf, int size) {

    if (fd < 0 || fd >= MAX\_OPEN\_FILES || !open\_files[fd].in\_use)

        return -1;

    OpenFile\* file = &open\_files[fd];

    FCB\* fcb = file->fcb;

    int bytes\_left = size;

    int total\_read = 0;

    while (bytes\_left > 0 && file->pos < fcb->size) {

        int block\_num = file->pos / 1024; // 计算块号

        int offset = file->pos % 1024;    // 块内偏移

        int bytes\_to\_read = (bytes\_left < 1024 - offset) ?

            bytes\_left : 1024 - offset;

        memcpy(buf + total\_read,

            disk[fcb->blocks[block\_num]] + offset,

            bytes\_to\_read);

        file->pos += bytes\_to\_read;

        total\_read += bytes\_to\_read;

        bytes\_left -= bytes\_to\_read;

    }

    return total\_read;

}

// 写入文件

int write\_file(int fd, char\* buf, int size) {

    if (fd < 0 || fd >= MAX\_OPEN\_FILES || !open\_files[fd].in\_use)

        return -1;

    OpenFile\* file = &open\_files[fd];

    FCB\* fcb = file->fcb;

    int bytes\_left = size;

    int total\_written = 0;

    while (bytes\_left > 0) {

        int block\_num = file->pos / 1024; // 计算块号

        int offset = file->pos % 1024;    // 块内偏移

        if (block\_num >= fcb->block\_count) {

            // 需要分配新块

            if (fcb->block\_count >= 128) return -1; // 达到最大块数

            int new\_block = alloc\_block();

            if (new\_block == -1) return -1; // 无可用空间

            fcb->blocks[block\_num] = new\_block;

            fcb->block\_count++;

        }

        int bytes\_to\_write = (bytes\_left < 1024 - offset) ?

            bytes\_left : 1024 - offset;

        memcpy(disk[fcb->blocks[block\_num]] + offset,

            buf + total\_written,

            bytes\_to\_write);

        file->pos += bytes\_to\_write;

        total\_written += bytes\_to\_write;

        bytes\_left -= bytes\_to\_write;

        if (file->pos > fcb->size)

            fcb->size = file->pos;

    }

    return total\_written;

}

// 用户登录

int login(char\* username) {

    for (int i = 0; i < MAX\_USERS; i++) {

        if (!users[i].valid) {

            strncpy(users[i].username, username, 20);

            users[i].valid = 1;

            current\_uid = i;

            return i;

        }

    }

    // 如果用户已存在

    for (int i = 0; i < MAX\_USERS; i++) {

        if (users[i].valid && strcmp(users[i].username, username) == 0) {

            current\_uid = i;

            return i;

        }

    }

    return -1;

}

// 列出当前目录文件

void list\_directory() {

    printf("Files owned by user %d:\n", current\_uid);

    for (int i = 0; i < MAX\_FILES; i++) {

        if (fcbs[i].filename[0] && fcbs[i].owner\_uid == current\_uid) {

            printf("%s\t%d bytes\n", fcbs[i].filename, fcbs[i].size);

        }

    }

}

int main() {

    // 初始化文件系统

    init\_disk();

    // 初始化用户表和FCB表

    memset(users, 0, sizeof(users));

    memset(fcbs, 0, sizeof(fcbs));

    memset(open\_files, 0, sizeof(open\_files));

    char command[100];

    printf("Simple File System Simulator\n");

    while (1) {

        printf("\> ");

        fgets(command, 100, stdin);

        command[strcspn(command, "\n")] = '\0'; // 去除换行符

        char\* context = nullptr;

        char\* token = strtok\_s(command, " ", &context);

        if (!token) continue;

        if (strcmp(token, "login") == 0) {

            token = strtok(NULL, " ");

            if (token) {

                int uid = login(token);

                if (uid != -1)

                    printf("Login successful as user %d\n", uid);

                else

                    printf("Login failed\n");

            }

        }

        else if (strcmp(token, "create") == 0) {

            token = strtok(NULL, " ");

            if (token && current\_uid != -1) {

                int index = create\_file(token);

                if (index != -1)

                    printf("File created at index %d\n", index);

                else

                    printf("Failed to create file\n");

            }

        }

        else if (strcmp(token, "delete") == 0) {

            token = strtok(NULL, " ");

            if (token && current\_uid != -1) {

                if (delete\_file(token) == 0)

                    printf("File deleted successfully\n");

                else

                    printf("Failed to delete file\n");

            }

        }

        else if (strcmp(token, "open") == 0) {

            token = strtok(NULL, " ");

            if (token && current\_uid != -1) {

                int fd = open\_file(token);

                if (fd != -1)

                    printf("File opened with descriptor %d\n", fd);

                else

                    printf("Failed to open file\n");

            }

        }

        else if (strcmp(token, "close") == 0) {

            token = strtok(NULL, " ");

            if (token && current\_uid != -1) {

                int fd = atoi(token);

                if (close\_file(fd) == 0)

                    printf("File closed successfully\n");

                else

                    printf("Failed to close file\n");

            }

        }

        else if (strcmp(token, "read") == 0) {

            token = strtok(NULL, " ");

            if (token && current\_uid != -1) {

                int fd = atoi(token);

                token = strtok(NULL, " ");

                if (token) {

                    int size = atoi(token);

                    char\* buffer = (char\*)malloc(size + 1);

                    int bytes\_read = read\_file(fd, buffer, size);

                    if (bytes\_read > 0) {

                        buffer[bytes\_read] = '\0';

                        printf("Read %d bytes: %s\n", bytes\_read, buffer);

                    }

                    else {

                        printf("Failed to read file\n");

                    }

                    free(buffer);

                }

            }

        }

        else if (strcmp(token, "write") == 0) {

            token = strtok(NULL, " ");

            if (token && current\_uid != -1) {

                int fd = atoi(token);

                token = strtok(NULL, "");

                if (token) {

                    int bytes\_written = write\_file(fd, token, strlen(token));

                    if (bytes\_written > 0)

                        printf("Wrote %d bytes to file\n", bytes\_written);

                    else

                        printf("Failed to write to file\n");

                }

            }

        }

        else if (strcmp(token, "ls") == 0) {

            if (current\_uid != -1) {

                list\_directory();

            }

        }

        else if (strcmp(token, "exit") == 0) {

            break;

        }

        else {

            printf("Unknown command\n");

        }

    }

    return 0;

}

**实验结果：**

Simple File System Simulator

> login lafael

Login successful as user 0

> create file1.txt

File created at index 0

> create file2.txt

File created at index 1

> ls

Files owned by user 0:

file1.txt       0 bytes

file2.txt       0 bytes

>  open file1.txt

File opened with descriptor 0

> write 0 Hello

Wrote 5 bytes to file

> close 0

File closed successfully

>  open file1.txt

File opened with descriptor 0

> read 0 100

Read 100 bytes: Hello

> close 0

File closed successfully

> ls

Files owned by user 0:

file1.txt       5 bytes

file2.txt       0 bytes

>