北京郵電大學

Beijing University of Posts and Telecommunications

操作系统课设

操作系统课程设计

详细设计说明书

组号：8

时间：2022/04/24

目录

1 总体设计.....1
1.1 概述.....1
1.1.1 功能描述.....1
1.1.2 运行环境.....1
1.1.3 开发环境.....1
1.2 设计思想.....2
1.2.1 软件设计构思.....2
1.2.2 关键技术与算法.....2
1.2.3 基本数据结构.....2
1.3 基本处理流程.....4
1.4 软件的体系结构设计.....6
1.4.1 软件体系结构框图.....6
1.4.2 软件主要模块及其依赖关系说明.....6
1.5 软件数据结构设计.....7
1.6 软件接口设计.....7
1.6.1 外部接口.....7
1.6.2 内部接口.....7
2 用户界面设计.....10
2.1 界面的关系图.....10
2.2 界面说明.....11
2.2.1 界面 1：预选界面.....11
2.2.2 界面二：主界面.....12
3 相关处理流程.....13
3.1 内存管理设计说明.....13
3.1.1 程序单元说明.....13
3.1.2 数据结构说明.....13
3.1.3 算法及流程.....14
3.1.4 函数说明.....14
3.2 设备管理设计说明.....15
3.2.1 程序单元说明.....15
3.2.2 数据结构说明.....15
3.2.3 算法及流程.....17
3.2.4 源程序文件说明.....18
3.2.5 函数说明.....18
3.3 UI 管理设计说明.....20
3.3.1 程序单元说明.....20
3.3.2 数据结构说明.....20
3.3.3 源程序文件说明.....23
3.3.4 函数说明.....23

3.4 进程管理设计说明.....25
3.4.1 程序单元说明.....25
3.4.2 数据结构说明.....28
3.4.3 算法及流程.....30
3.4.4 源程序文件说明.....33
3.4.5 函数说明.....34
3.5 文件管理设计说明.....34
3.5.1 数据结构说明.....34
3.5.2 算法及流程.....35
3.5.3 数据存储说明.....37
3.5.4 源程序文件说明.....37
4 总结.....38
5 附录.....38
5.1 PD-OS 的 exe 可执行文件中指令格式规定.....38
5.2 PD OS 规格参数：.....39

1 总体设计

1.1 概述

1.1.1 功能描述

我们的模拟操作系统，即 PD-OS（Paper Doll Operation System）从操作系统课设的目标出发，专注于操作系统的上层逻辑结构和操作系统主要功能的实现及相互配合。我们把操作系统所能实现的主要功能分为五块，分别是：UI、进程管理、内存管理、文件管理和设备管理。这其中，UI 采用图形化界面为用户提供使用渠道；进程管理处理内存中的进程，实现进程状态的转换、timer 和中断处理；内存管理实现对内存存储空间的管理；文件管理对文件和目录进行处理；设备管理维护 IO 设备的队列和磁盘的管理。

从用户的角度来讲，PD-OS 使用图形化的界面，为用户提供进程的创建与中断（将以运行数个可执行程序的形式实现）、文件的创建，修改和删除、系统内各个资源的监控。

1.1.2 运行环境

PD-OS 的主要运行环境为 Windows 10 操作系统，而在 Qt5 应用程序开发框架的可移植性的基础上，我们计划实现 PD-OS 在 Linux、Unix，智能手机系统 Android、iOS、WinPhone， 嵌入式系统 QNX、VxWorks 等环境上的运行。

1.1.3 开发环境

团队采用 Visual Studio 2019 平台上的 C++语言完成程序，同时，为了实现多人的协同开发，我们采用 CODING 平台：一个提供 Git/SVN 代码托管、项目协同、测试管理等在线工具的一站式软件研发管理协作平台进行代码的共享和项目的进程协作。

1.2 设计思想

1.2.1 软件设计构思

PD-OS 的主要目标是对操作系统的上层逻辑结构进行模拟。我们最初对《30 天自制操作系统》等相关书目进行了阅读，发现市面主流的教程着眼于硬件的适配，对于汇编语言和磁盘等硬件技术性质的功能着墨较多，而对于操作系统实际的逻辑功能等不够关注。而在仔细实验指导书后，我们决定我们的模拟操作系统 PD-OS 主要是为了对操作系统的上层功能进行模拟和演示，完成进程管理和调度、内存管理(存储分配与回收，进程交换)、时钟管理、中断处理、用图形界面展示多道程序并发执行的过程等操作系统的基本功能，从而加深理解操作系统的基本功能、原理和工作机制。

在设计指令格式时，我们对 MIPS 指令格式又进行了深入研究，对指令应该达到的功能进行了构思，最终决定对于 CPU 在 PD-OS 中扮演的角色进行弱化，不去过分强调各寄存器和 ALU 等设备。

因此，我们设计的构思的**既不过分着眼于底层，也不纠结于 cpu 内部运作原理，而是专注于操作系统这一中间层次。**

1.2.2 关键技术与算法

进程管理：采用 FCFS 调度算法、SJF 调度算法、多级反馈队列调度算法；
文件管理：索引分配；FCFS, SCAN, C-SCAN, LOOK, C-LOOK 等磁盘寻道算法
设备管理：FCFS 算法；
内存管理：请求分配式页式管理；best-fit 分配算法，页面替换的 LRU 算法和 FIF 算法；
以上算法的具体说明将在下面分模块介绍部分具体解释。

1.2.3 基本数据结构

①进程管理块 PCB：

```
class PCB
{
    int PID;    //进程标识符
    int processState;  //进程状态
    long IR;  //指令寄存器，保存当前正在执行指令的地址
    long pSeg;  //program segment，程序段地址
    long dSeg;  //data segment，数据段地址
    };
```

②就绪/阻塞队列：

```
class QUEUE
{
    int priority;  //队列优先级
    int PID[MAX_Process];   //进程队列
};
```

③储存设备对象的基本信息：

```
struct device{
    string deviceName;//设备名称
    double tansmitRate;//传输速率（若有）
    int isBusy;//是否被占用
    int request[REQUESTNUM];//该设备的请求队列
};
```

④储存设备请求的信息：

```
 class deviceRequest{
    int pid;//进程 ID
    string deviceName;//请求的设备名称
    string data;//数据内容（若有）
    double ioTime;//需使用该设备的时长（若有）
};
```

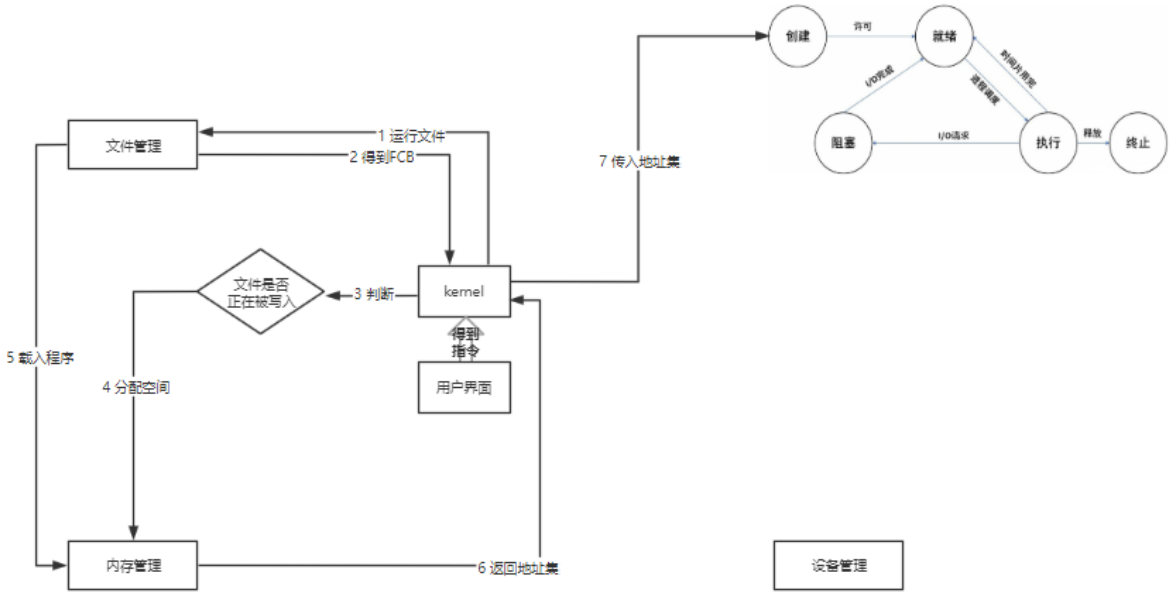
⑤进程存储位置：

```
class processStorage{
    int ID;
    int* dataSegment;
    int* codeSegment;
};
```

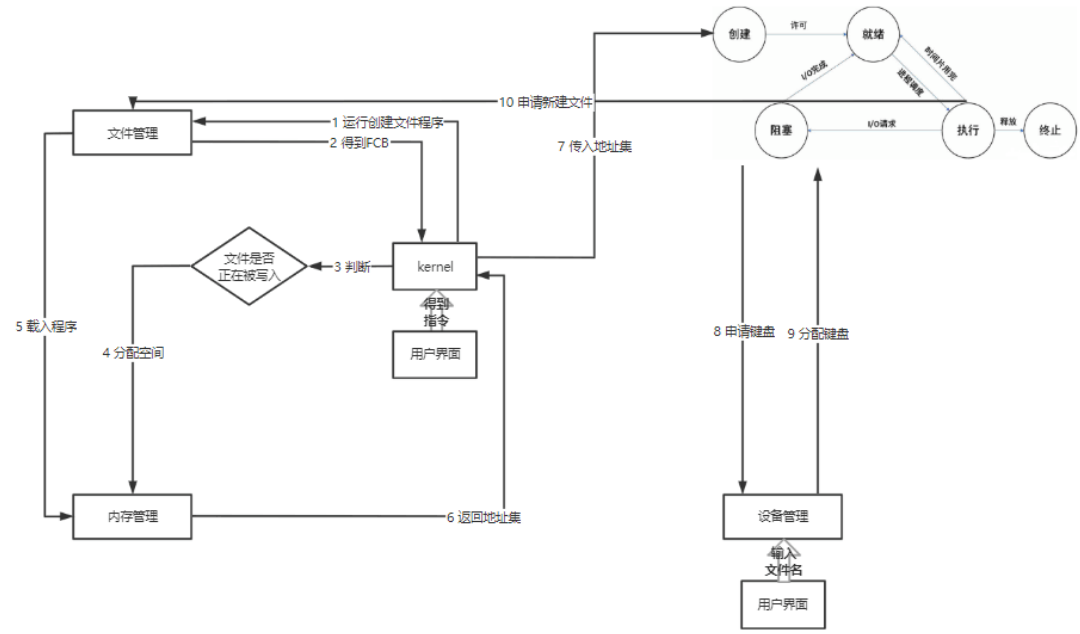
注：文件、内存和 UI 部分中数据结构请参考各模块设计说明。

1.3 基本处理流程

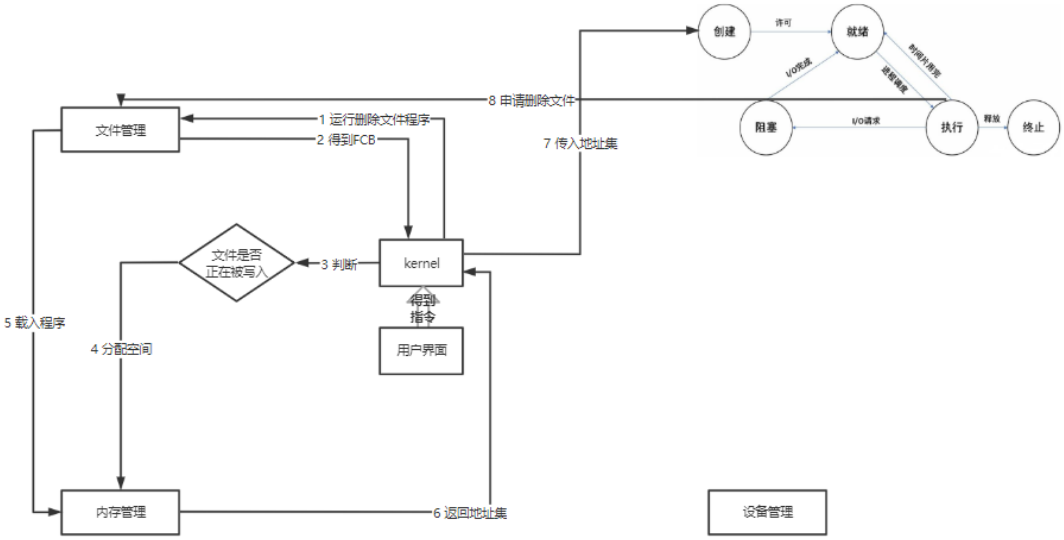
①创造进程：



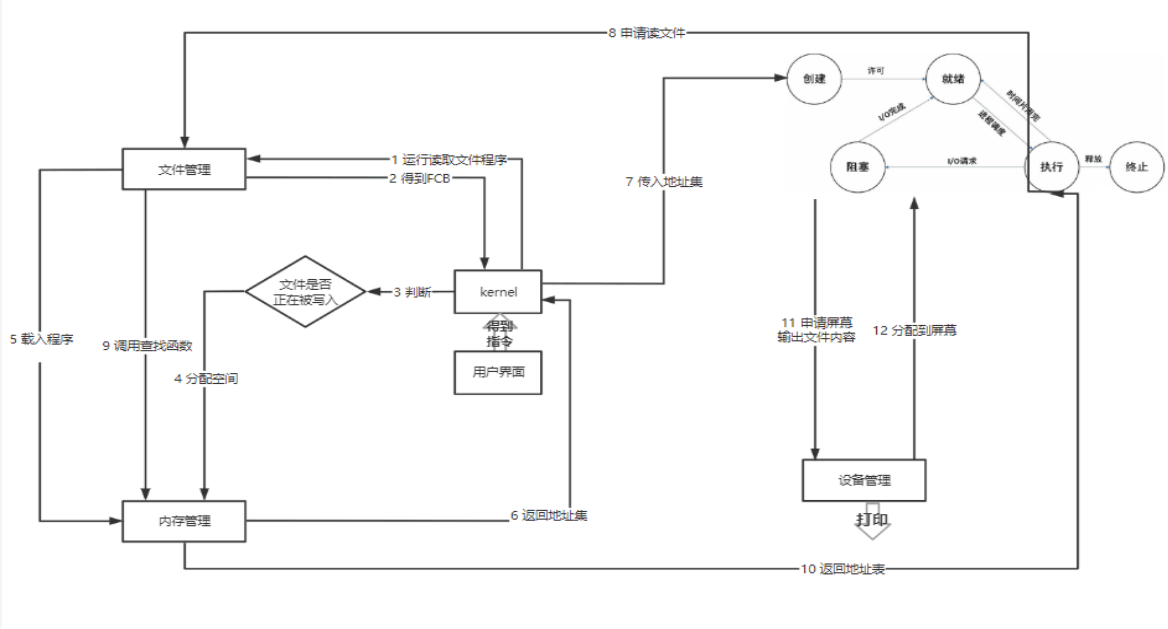
②新建文件：



③删除文件：

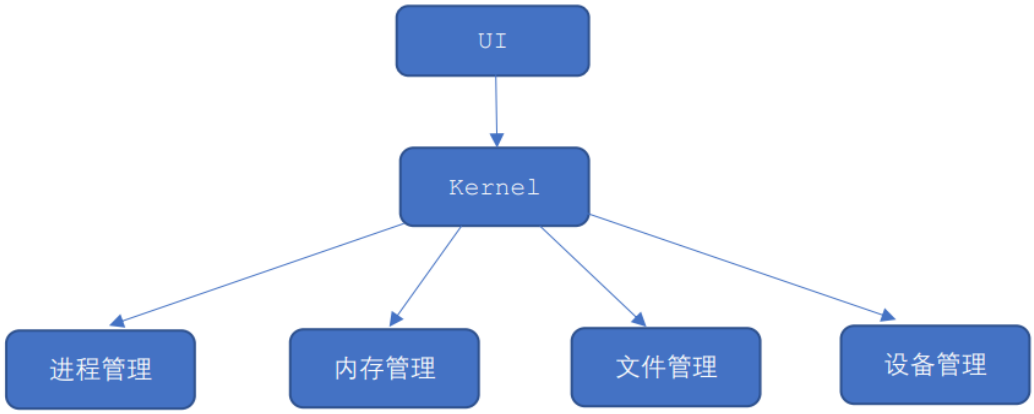

创建
许可
就绪
I/O完成
时间片用完
I/O请求
阻塞
执行
释放
终止
8 申请删除文件
文件管理
1 运行删除文件程序
2 得到FCB
7 传入地址集
文件是否
正在被写入
3 判断
kernel
得到
指令
用户界面
5 载入程序
4 分配空间
内存管理
6 返回地址集
设备管理

④读取文件：


8 申请读文件
创建
许可
就绪
I/O完成
时间片用完
I/O请求
阻塞
执行
释放
终止
文件管理
1 运行读取文件程序
2 得到FCB
7 传入地址集
文件是否
正在被写入
3 判断
kernel
得到
指令
用户界面
5 载入程序
9 调用查找函数
4 分配空间
11 申请屏幕
输出文件内容
12 分配到屏幕
设备管理
打印
内存管理
6 返回地址集
10 返回地址表

1.4 软件的体系结构设计

1.4.1 软件体系结构框图



1.4.2 软件主要模块及其依赖关系说明

从上图可以看到，PD-OS 主要分为了六个模块，分别是最外层的图形化 UI 设计，链接 UI 和 OS 后端的 kernel，进程管理，内存管理，文件管理和设备管理部分。各个模块之间互相依赖，紧密联系。

Kernel 也就是 Qt 中的信号与槽机制，进程、内存、文件、设备四个模块依赖 kernel 与用户 UI 建立联系；进程管理依赖文件存储源程序，依赖内存管理创建进程，依赖设备管理完成进程状态的转换；内存管理依赖文件和进程分配内存，依赖设备管理中的磁盘管理把文件读入内存；文件管理依赖磁盘管理存储文件，依赖进程管理创建和修改删除文件，依赖内存管理把文件调入内存；设备管理依赖进程管理分配 IO 设备，依赖文件管理整理磁盘内容。

依赖关系	进程管理	内存管理	文件管理	设备管理
UI	提供用户 UI	提供用户 UI	提供用户 UI	提供用户 UI
Kernel	链接	链接	链接	链接
进程管理		分配内存	创建修改文件	分配 IO 设备
内存管理	创建进程		把文件调入内存	
文件管理	存储进程源文件	分配内存程序段		整理磁盘内容
设备管理	完成进程状态转换	把文件读入内存	存储文件	

1.5 软件数据结构设计

详见 1.2.3 基本数据结构和各部分数据结构说明。

1.6 软件接口设计

一、操作界面（命令接口）
采用图形化界面，用户使用这个界面组织工作流程和控制程序的运行。
二、系统功能服务界面（程序接口）
即 kernel 部分，用户程序在其运行过程中，使用系统调用功能来请求操作系统的服务。

1.6.1 外部接口

键盘：输入数据保存文件。
屏幕显示：显示程序图形界面关系。
Printer：可调用的显示设备之一。
磁盘：存储文件的设备。

1.6.2 内部接口

一、内存部分：
1.best-fit:
- 分配内存
  参数：PCB，size
  功能：根据提供的参数分配内存，如果分配失败返回-1，如果分配成功返回起始地址。
- 回收内存：
  参数：PCB，size
  功能：根据提供的参数回收相应内存，无返回参数

2. 分页存储
- 创建进程分配内存
  参数：PCB，size
  功能：根据参数分配内存，如果有空闲页直接分配，如果没有则进行页面替换，返回页起始地址，一页系统内存，一页数据内存（供写操作用）
- 查找虚拟内存中的页是否在物理内存中：
  参数：虚拟内存块号指针（可能含有多个）
  功能：根据虚拟内存块号查找物理内存中是否有对应页，如果有则给进程返回页地

址，如果没有则重新分配页并调用读入函数，将读入后的页地址返回给进程

- 访问页表：
  参数：虚拟内存块号
  功能：根据虚拟内存块号找到物理页号并返回
- 删除特定页内容：
  参数：页地址
  功能：删除页地址原内容
- 释放内存：
  参数：PCB
  功能：释放 PCB 对应的内存

二、文件部分

- 创建文件
  参数：文件名称
  功能：在当前目录创建新的文件，成功返回 0，名称重复返回-1，名称非法返回-2
  创建文件夹的 C 语言指令：system("mkdir .\\XXX");（调用系统指令）
  设置一个特定的文件夹，为不可修改
  调用：创建文件时被进程调用

- 删除文件
  参数：文件名称
  功能：删除已有的文件，释放磁盘存储空间，成功返回 0，无权限返回-1，正在被写入返回-2
  对于目录类型的文件，需要深度优先遍历目录下的所有文件，并进行删除
  调用：删除文件时被进程调用；删除多级目录时被自身递归调用

- 读取文件
  参数：文件名称、块号（？）、虚存的真实地址
  功能：将目标文件载入到给定的内存地址中，成功返回 0，文件正在被占用返回-1，还有下一块返回-2
  文件状态修改：
  若写入标识符不为 0，则返回-1，不改变标识符的值；在文件开始时，打开文件表对应的读文件标识符+1，若返回为 0，则标识符-1；若返回为-2，则标识符保持+1 不变
  调用：读取文件时被进程调用；新建进程时被内存调用

- 写入文件
  （根据 QT 后续研究情况而定）
  情况 A：
  若：可以对输出进行随机读写
  参数：文件名称、内存地址+偏移量（表示从内存地址开始读到偏移量结束）
  功能：用新的内容替换原有文件，注意根据新文件的长度分配或释放磁盘块

情况 B：
若：无法对输出值进行随机读写
参数：文件名称、内存地址+偏移量、写入模式
功能：根据写入模式，用心的内容替换原有文件；或者在文件末尾写入，注意存储空间分配
返回值：成功返回 0，正在被占用返回-1
文件状态修改：
函数开始时检查文件状态，若读或写标识符不为 0，则返回-1；若标识符为 0，修改写文件标识符为 1，直到写入完毕，修改标识符为 0
注：因为各种原因，选用的文件锁的问题是，首先，写入必须一次完成， 进行分段写入的时候，文件可能被别的进程修改
其次，要求每一次读文件，整个文件的所有块都必须全部载入内存（仍然是因为没有 close，没有办法强行终止，只能在读到文件结尾才认为这个文件不再被读），否则不能被写入，并且如果一次读操作只读了三块中的一块就结束的话，会在文件系统中造成一些隐患。
调用：写入文件时被进程调用

- 查找文件（寻址）
  参数：文件名称
  功能：输入文件名称时，在当前目录查找，输入路径时，按绝对路径从根目录开始查找，返回文件控制块 FCB（包括文件大小、类型、权限、当前在内存的位置等信息）
  调用：被文件系统的上述接口调用；创建新进程时，被 kernel 调用，返回文件大小以判断内存空间是否足够；被内存调用确定是否已经在内存

- 修改目录
  UI 每一次改变目录，kernel 告知文件系统，修改当前位置指针

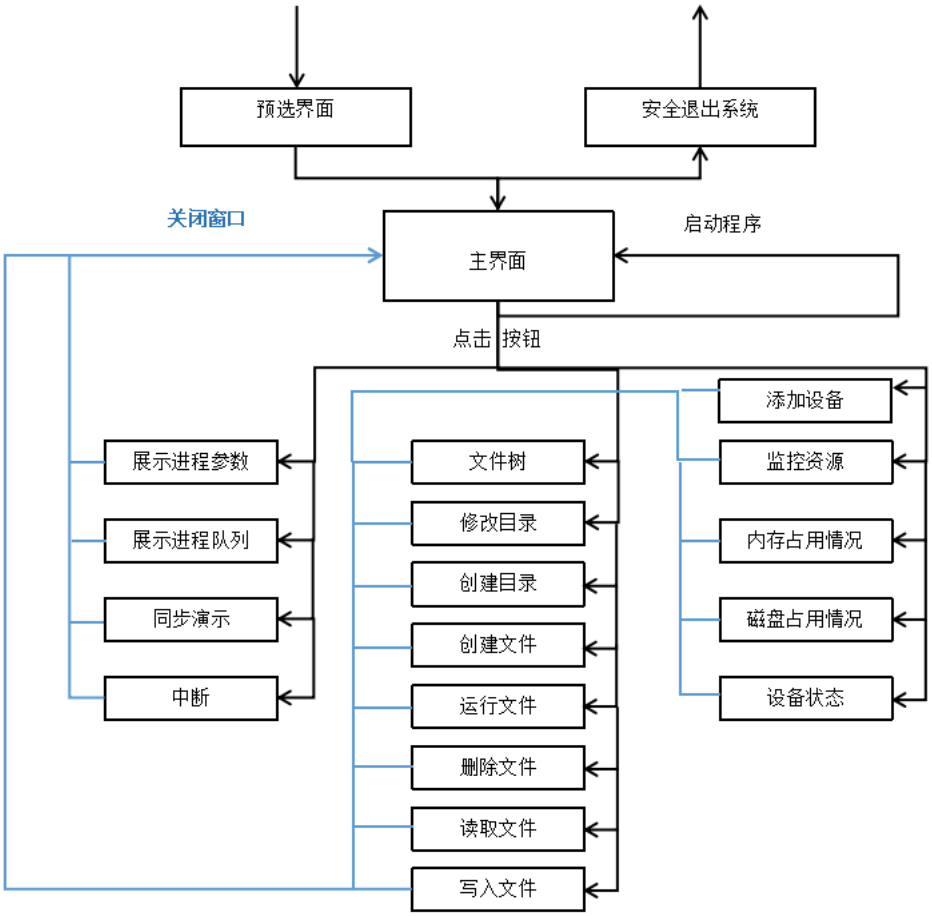
- 打印目录
  功能：打印当前路径到 UI 界面
  调用：被 UI 调用

- 展示当前目录下所有条目
  功能：遍历并打印当前目录下所有文件名称

- 查找当前目录下符合某种条件的所有文件
  返回值：符合某种条件的文件列表
  调用：在 UI 界面选择操作时被调用

2 用户界面设计

2.1 界面的关系图

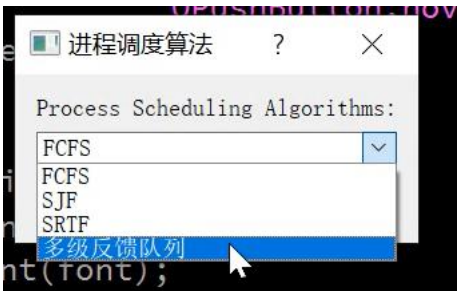


2.2 界面说明

2.2.1 界面 1：预选界面

算法具有多个选项可供用户选择，但选择过后在本次退出模拟操作系统之前不可更改
界面模拟：

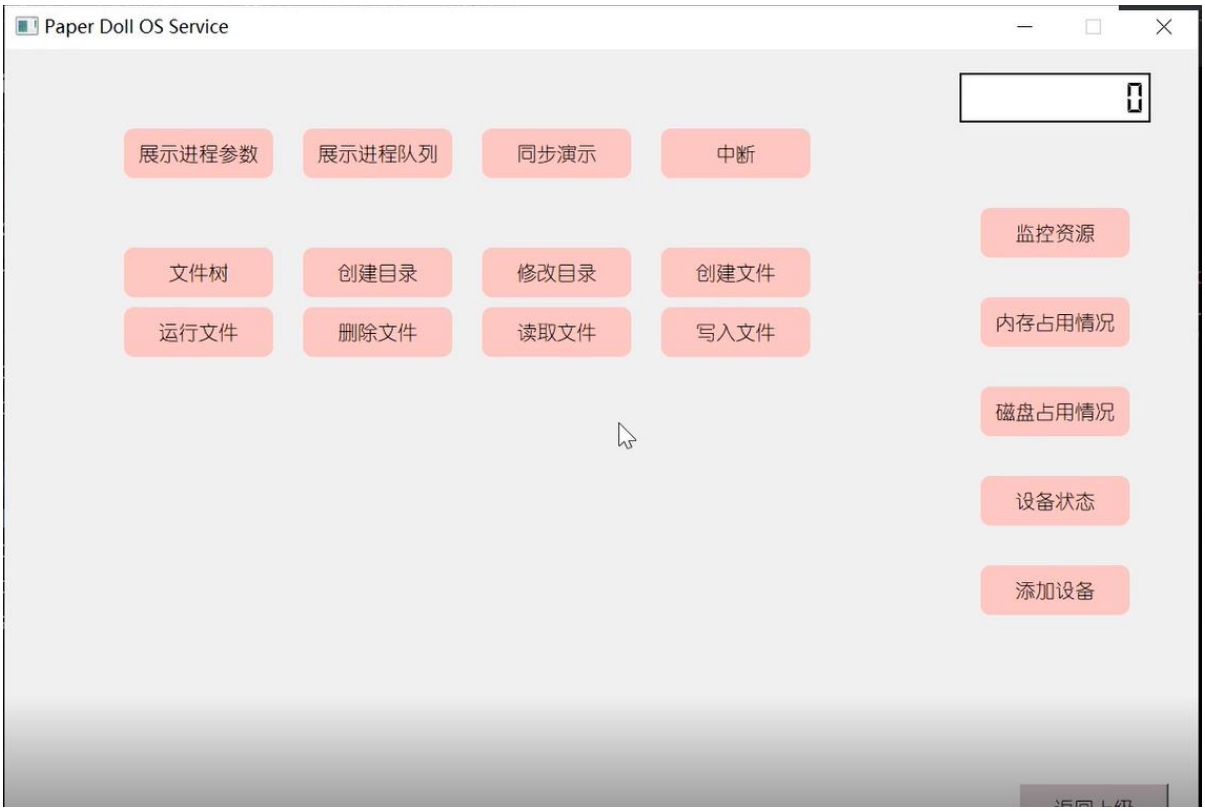
界面演示：



2.2.2 界面二：主界面

最开始进程还没创建，所以灰色按钮不可点：
界面模拟：

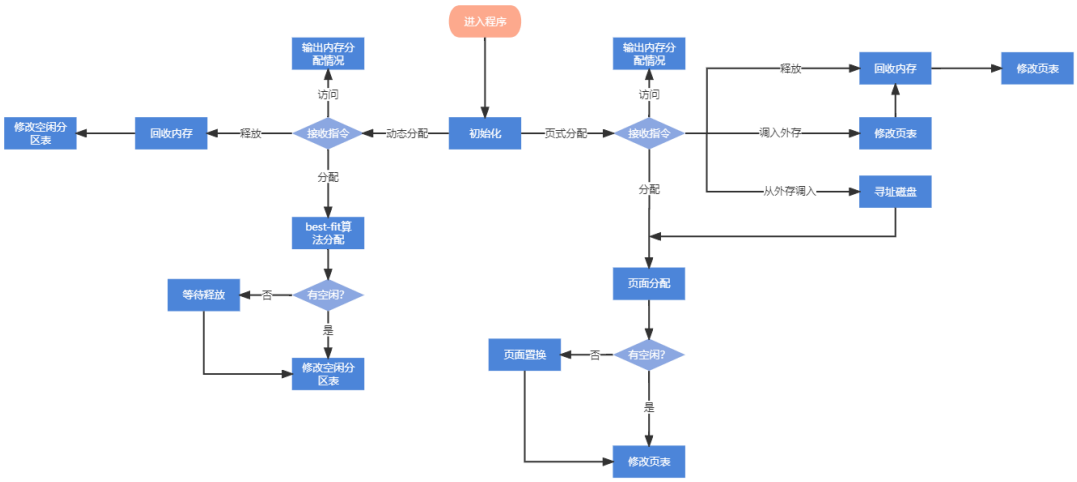
界面演示：



3　相关处理流程

3.1　内存管理设计说明

3.1.1　程序单元说明



3.1.2　数据结构说明

①全局变量：

```
const char mode = 'p'; //分页 mode=p  连续 mode=c
const int pageSize = 1024; //页面大小
const int virtualNumber = 8; //虚拟页数
const int physicalNumber = 3; //物理页数
const int replaceStrategy = 0; //替换策略，0 代表 FIFO，1 代表 LRU
long allocated = 0; //已分配内存
int virtualMemory[virtualNumber][3]; //虚拟内存分配情况
int physicalMemory[physicalNumber]; //物理内存分配情况
list<int> schedule_queue; //调度队列
int physicalSize=0; //物理内存已使用情况
int page_fault=0; //缺页次数
string Virtual[virtualNumber]; //虚拟内存里存储内容
string Physical[physicalNumber]; //物理内存里存储内容、
```

②局部变量：

pageTable 类：

```
typedef std::map<int, int> TABLE; //map<virtual page, frame number>
TABLE table;
int max_adress = 0;
```

3.1.3 算法及流程

1.页式分配

采用页式分配时可以指定页面数量及页面大小，并记录每一页的分配情况（占用空间/占用进程 ID/分配 ID），页式分配算法在进行分配与回收时都是以页面为单位；

2.best-fit 分配算法

使用数组结构分别记录内存的分配情况（基址/分配大小/占用进程 ID/分配 ID）及空闲区的情况（基址/空闲区域大小），在每一次分配时，遍历空闲记录表，挑选符合条件的空闲区进行分配，在每一次回收时，需要考虑是否有与释放区相邻的空闲区，若有，则需要将该释放区合并到相邻的空闲区中，并修改该区的大小和首址，否则，将释放区加入空闲区的记录表中。

1）页面替换的 LRU 算法：

依据进程页表判断当前所访问的虚拟页是否已经被调入到物理内存中：

- 若已在物理内存中，则将该页放在调度队列尾部（表示最近访问）；
- 若未在物理内存中，则判断物理内存是否空闲，若空闲，则将访问页面调入物理内存中，并将该页放在调度队列尾部；若不空闲，则将调度队列头部的页面替换出去，将该页面所在页表的有效位置为-1（无效），并将该页面从调度队列中删除，后将访问页面调入物理内存，并将访问页面放在调度队列尾部。

2）页面替换的 FIFO 算法：

依据进程页表判断当前所访问的虚拟页是否已经被调入到物理内存中：

- 如已在物理内存中，则不做任何操作；
- 若未在物理内存中，则判断物理内存是否空闲，若空闲，则将访问页面调入物理内存中，并将该页放在调度队列尾部；若不空闲，则将调度队列头部的页面替换出去，将该页面所在页表的有效位置为-1（无效），并将该页面从调度队列中删除，后将访问页面调入物理内存，并将访问页面放在调度队列尾部。

3.1.4 函数说明

①class pageTable 类：

```
void insert_vertual(int pageNumber); //插入虚拟页（pageNumber, -1），-1 表示没有待分配物理页
void delete_page(int pageNumber); //删除虚拟页
int transform(int pid, int address, int pageSize); //查找待访问页面在虚拟页的第几页，返回虚拟页号
void modify(int vNum, int pNum, int valid); //将虚拟页调入、调出物理内存
int find_physical(int id); //返回 tableid
```

②memoryManager 类：

```
void Initial(); //初始化，确定分配策略，调度算法等
void alloc(int pid, int size); //选择分配策略分配内存
void free(int pid); //根据分配策略释放内存
void Access(int pid, int address); //访问内存
```

```
void pageAlloc(int pid, int size); //分页存储分配内存
void conAlloc(int pid, int size); //连续存储分配内存
int pageFree(int pid); //分页存储释放内存
int conFree(int pid); //连续存储
void pageAccess(int pid, int address); //分页存储访问内存
void conAccess(int pid, int address); //连续存储访问内存
void showPage(int pid, int address); //分页存储查看物理内存
void showCon(int pid, int address); //连续存储查看物理内存
void watch_virtual_memory(); //监视虚拟内存
void watch_physical_memory(); //监视物理内存
void LRU(int vnum, pageTable ptable); //访问内存时使用 LRU 策略
void FIFO(int vnum, pageTable ptable); //访问内存时使用 FIFO 策略
```

③tool 类：

```
int find_physical(int elem); //根据元素查找内存
```

3.2 设备管理设计说明

3.2.1 程序单元说明

该程序单元完成对设备的管理功能。
具体完成：
①支持添加新设备；
②将请求队列中的设备按“先来先服务”的原则分配给申请该设备的进程；
③若申请的设备是键盘，则把键盘输入暂时储存在设备缓存中，并将其与进程的 pid 一起传给内存；
④展示所有设备的状态。

3.2.2 数据结构说明

①常量

```
//最大同时请求个数
#define REQUESTMAXNUM 12

//最大设备数
#define DEVICEMAXNUM 12
```

②全局变量

```
//当前设备数量
int DEVICENUM=0;
```

```
//用数组存储各设备当前请求数量
int REQUESTNUM[DEVICEMAXNUM]={0};

//全局变量储存所有设备的信息
Device devices[DEVICEMAXNUM];

//储存键盘的输入，之后传给内存
string deviceBuffer;
```

③结构体

```
//储存设备请求的信息的结构体
typedef struct deviceRequest{
    //进程 ID
    int pid; //-1 表示无，初始化的时候赋值为-1
    //数据内容（若有）
    //string data;
    //需使用该设备的时长
    double ioTime;
}Request;
```

④类

```
class Device{
    public:
        //设备名称
        string deviceName;
        //传输速率
        double tansmitRate;//默认为 0，即为没有
        //是否被占用
        int isBusy;//1 为忙碌，0 为空闲
        //该设备的请求队列
        Request request[REQUESTMAXNUM];

        //函数
        void initDevice(string name,double transRate);
        void initIO();
        void createDevice();
        void showDevices();
        void requestDevice(int id,string device,double time);
        int FCFS(string device);
        void deviceWork();
};
```

3.2.3 算法及流程

主要算法：

FCFS 算法：将申请同一设备的进程存入一个队列中，即请求队列，每次将设备分配给该队列内储存的第一个进程，当该进程使用完设备或屏幕已经打印出相关信息后，则可以将设备分配给下一个请求队列中的进程。

```
int FCFS(string device){

   for(int i=0;i<DEVICENUM;i++){
         //对所有设备进行循环
            if(device==devices[i].deviceName && REQUESTNUM[i]!=0){
                  //若该设备申请队列不为空
                  if(devices[i].isBusy==1){
                        //若该设备处于忙碌状态
                        //则输出该设备正忙
           return -1;
                        //并返回-1 表示没有进程的申请设备被完成
                  }
                  else{
                        //若该设备不处于忙碌状态
                        //则先将其状态改为忙碌


                        //将该设备分配给申请的进程，时间为申请的 iotime 时长

                        //若申请的设备是键盘
                        //则获取键盘输入
                        //并将其与 pid 一起传给内存


                        //使用完设备后回收设备
           //设备的状态改为不忙碌
                        //记录刚刚完成的任务是哪个进程的，pid

                        //循环更新该设备的请求队列
           //即将每一条请求都往前挪动一个位置
                        //该设备的请求数量减一

                        break;
                        //退出循环
                  }
```

```
            }
            else if(device==devices[i].deviceName && REQUESTNUM[i]==0){
        //若申请的设备的请求队列为空

                return -1;
        //则返回-1 表示没有进程的申请设备被完成
            }
            if(i==DEVICENUM-1){
        //若循环了一圈没有找到该设备
        //则返回-1 表示没有进程的申请设备被完成
                return -1;
            }
    }

    //接上面的 break
    //将刚刚储存在临时变量 tmp 中的完成使用设备的进程 ID 返回
        return tmp;
}
```

流程说明：
当有新设备时可以添加新设备；
对各个设备按 FCFS 调度算法依次处理对设备的请求进程，修改分配给进程的设备的占用状态，待使用结束后回收、修改状态、再分配；
当用户用键盘输入后，UI 将输入框中内容传给设备管理中的缓存，设备管理再将 pid 与缓存中的内容传给内存管理；
当向用户展示设备状态时，打印出所有设备的名称以及是否正在被使用，若是，则还要输出正在使用该设备的进程号。

3.2.4 源程序文件说明

源程序文件为 device.cpp。
完成的功能：对设备进行管理：添加新设备，将设备按 FCFS 原则分配给需要的进程，将键盘的输入缓存并传给内存，展示设备状态。

3.2.5 函数说明

① void Device::initIO();
功能：调用多个 initDevice()对多个设备进行初始化；
最初的设备为键盘 keyboard、屏幕 screen、鼠标 mouse 三个；
参数：无；
返回：无；

② void Device::initDevice(string name,double transRate);
功能：对某个设备进行具体初始化，给该设备的各参数赋初始值；
设备名称、传输速率赋值传入的参数，isBusy 初始化为否 0，申请设备的 pid 设为-1，申请设备时长设为 0；
参数：string 类型的设备名称，double 类型的设备传输速率；
返回：无；

③ void Device::createDevice();
功能：添加一个设备；
先判断是否已达设备上限，若否，则输入要添加的设备的名称和传输速率（默认为0），判断该设备是否已存在，若是一个新设备，则 isBusy 初始化为否 0，申请设备的 pid 设为-1，申请设备时长设为 0；
参数：无；
返回：输出是否添加成功；

④ void Device::showDevices();
功能：展示所有设备状态列表；
打印出所有设备的名称以及是否正在被使用，若是，则还要输出正在使用该设备的进程号；
参数：无；
返回：输出设备状态列表；

⑤ void Device::requestDevice(int id,string device,double time);
功能：申请使用设备，将申请设备的进程放入相应的请求队列中；
先判断申请的设备是否存在，若存在，则再判断申请队列是否已满，若否，则在该设备的请求队列中加入 pid 和 iotime；
参数：int 类型的进程 ID，string 类型的进程申请的设备名称，double 类型的进程申请要使用设备的时长；
返回：输出是否申请成功；

⑥ void Device::deviceWork();
功能：循环调用 FCFS 函数，使设备在使用结束后再分配；
先计算出所有设备的总申请数的和，在所有设备的总申请数大于 0 的情况下，对各个设备循环调用 FCFS；
参数：无；
返回：输出每一次循环使用完设备的进程 ID；

⑦ int Device::FCFS(string device);

功能：按先来先服务原则处理请求队列；

将空闲的设备分配给请求队列中的第一个进程，并在使用结束后，对设备进行回收，修改 isBusy 为否 0，并修改该设备的请求队列；

若是键盘设备，则还要将输入框中的内容暂时储存在设备缓存中，并将其与进程 ID 一起传给内存；

参数：设备名称；

返回：若该设备请求队列不为空，则返回已经被完成的进程 ID，即原本请求队列中的第一个进程 ID，若该设备请求队列为空，则返回-1；

3.3 UI 管理设计说明

3.3.1 程序单元说明

采用图形化界面，用户使用这个界面组织工作流程和控制程序的运行。

3.3.2 数据结构说明

```
//显示刚进入的开始界面，用户可以进入或退出操作系统
#include <QWidget>
#include <QLabel>
#include <QInputDialog>
#include "button.h"
#include "mainscene.h"

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);//初始化窗口
    ~Widget();

    void on_Item_clicked();

public:
    QLabel *title;//标题（欢迎来到操作系统）

    Button *enter;//进入按钮
```

```
    Button *exit;//退出按钮

    MainScene *scene;//主场景的入口

    QString ProcessSchAlgitem;//进程调度算法

    QString MemoryStorageitem;//内存存储方式

    QString HeadSeekAlgitem;//磁头寻道算法

};
#endif // WIDGET_H

#ifndef MAINSCENE_H
#define MAINSCENE_H

//此界面为主界面，显示地图和所有的服务按钮
#define BUTTONNUM 18

struct SystemTime
{
    int hour;
    int min;
    int second;
};

class MainScene : public QWidget
{
    Q_OBJECT
public://函数
    explicit MainScene(QWidget *parent = nullptr);//设置主界面初始布局
    void startTimer();//系统开始计时
    QString getCurrentSystemTime();
    void setPageLayout();//设置窗口和布局
    void init();//初始化信息
    void recordStart(ServiceType type);//日志记录起始状态
    void recordEnd();//记录终止状态

public://变量

    /*服务栏*/
```

```
//    Button *mybutton[BUTTONNUM];
    Button *proInfo;//进程参数
    Button *proQueue;//进程队列
    Button *synShow;//同步演示
    Button *suspend;//中断

    Button *fileTree;//文件树
    Button *createContent;//创建目录
    Button *modifyContent;//修改目录
    Button *createFile;//创建文件
    Button *runFile;//运行文件
    Button *deleteFile;//删除文件
    Button *readFile;//读取文件
    Button *writeFile;//写入文件

    Button *monResource;//监控资源
    Button *memoryUsage;//内存占用情况
    Button *diskUsage;//磁盘占用情况
    Button *deviceInfo;//设备状态
    Button *addDevice;//添加设备

    Button *backBtns;//返回按钮

    QPixmap *menu_pix;//菜单栏画布
    InputWidget *iw;//显示输入窗口
    /*系统计时器*/
    QLCDNumber *tm_widget;//显示时间当前时间
    SystemTime time;//系统时间

//private:
//    FileLoader *loader;
//    Log record;

//public slots:
//    void updatePage();//刷新页面
//    void enableButton(Button *btn);//按钮启用
//    void disableButton(Button *btn);//按钮禁用
//protected:

signals:
    void backToWidget();
};
```

```
#endif // MAINSCENE_H
```

3.3.3 源程序文件说明

Widget.cpp 窗口显示刚进入的开始界面，用户可以进入或退出系统
Mainscene.cpp 此界面为主界面，显示所有的服务

3.3.4 函数说明

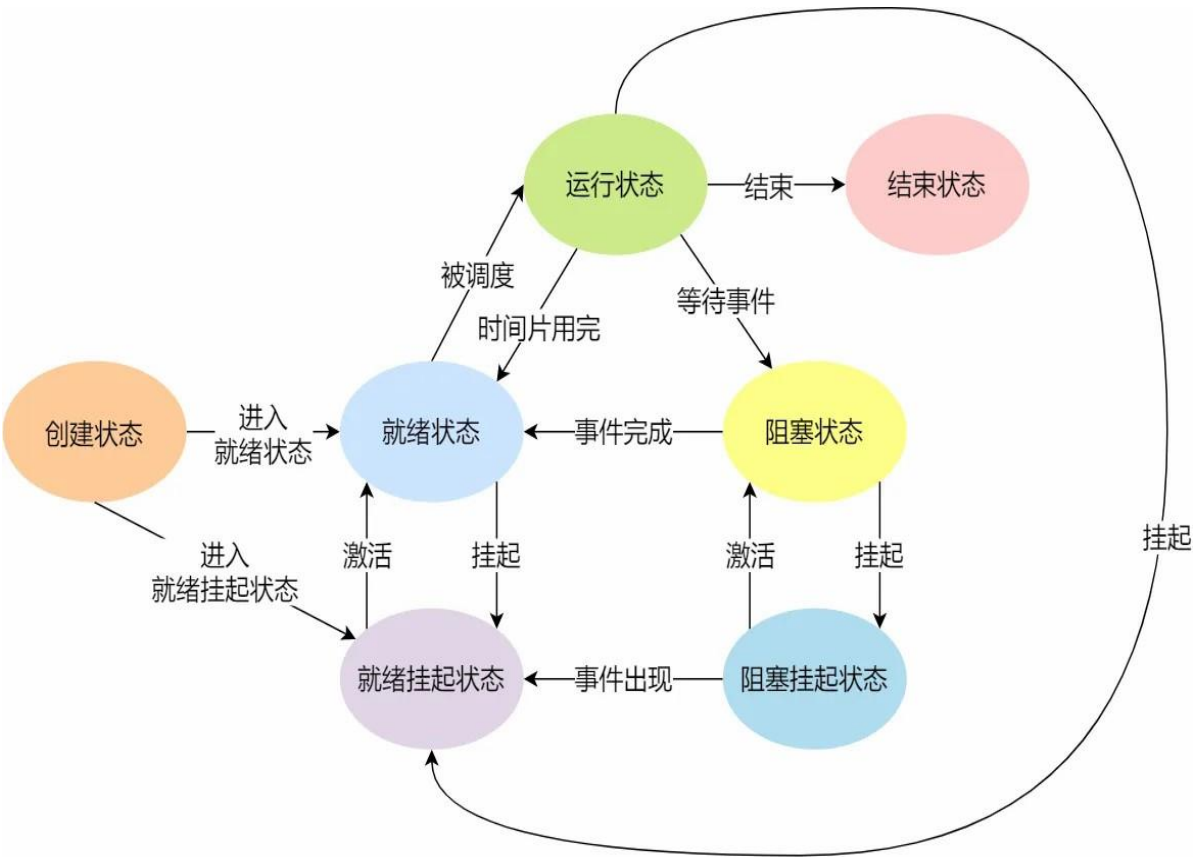
```
#include "mainscene.h"
#include "button.h"
#include <QPixmap>
#include <QFile>
#include <QTextStream>
#include <QTimer>
#include <QMessageBox>
#include <QtDebug>


MainScene::MainScene(QWidget *parent) : QWidget(parent)
{
    /*第一步，设置窗口整体布局*/
    //设置窗口和布局
    this->setPageLayout();



    /*第二步，加载文件，初始化等*/
//    loader = new FileLoader;
//    loader->writeEnterLog("进入操作系统");

    //初始化信息
//    this->initUser();



    /*第三步，将每个按钮连接到对应的函数*/
    //设置 connect

    //1.连接到函数
   this->disableButton(this->proInfo);//先设置禁用
   connect(this->proInfo,&QPushButton::clicked,this,&MainScene::showProInfo);
```

```
    //对其他按钮的操作

    //2.关闭窗口，打开开始界面
    connect(this->backBtns,&QPushButton::clicked,this,[=](){
       loader->writeExitLog("退出系统");
         emit this->backToWidget();//

    });


    //设置计时
//    this->startTimer();
}

void MainScene::setPageLayout()
{
    //设置窗口
    this->setFixedSize(1200,800);
    this->setWindowTitle("Paper Doll OS Service");

    //定义 button
    this->proInfo = new Button(this,"展示进程参数");
    this->proQueue = new Button(this,"展示进程队列");
    this->synShow = new Button(this,"同步演示");
    this->suspend = new Button(this,"中断");
    //设置按钮的位置
    int x = this->width()*0.1;
    int y = this->height()*0.1;
    int spaceX = proInfo->width()*1.2;
    int spaceY = proInfo->height()*1.2;
    proInfo->move(x, y);
    proQueue->move(x + spaceX, y);
    synShow->move(x + 2*spaceX, y);
    suspend->move(x + 3*spaceX, y);
    //设置按钮的位置
//    this->choice = new QPushButton[Num];
//    for(int i = 0; i < Num; i ++)
//    {
//        int w_space = 1.2*space;
//        choice[i].setParent(this);
//        choice[i].setText(QString("xxx").arg(i + 1));
//        choice[i].resize(90,50);
//        choice[i].move(this->width()*0.88 - w_space*i,this->height()*0.85);
```

```
//    }


    //设置定时器的位置
    tm_widget = new QLCDNumber(this);

    //设置位置
    tm_widget->resize(this->width()*0.16,50);
    tm_widget->move(this->width()*0.80,this->height()*0.03);

    //设置样式和模式
    tm_widget->setMode(QLCDNumber::Dec);
    tm_widget->setDigitCount(10);
    tm_widget->setSegmentStyle(QLCDNumber::Flat);
    tm_widget->setStyleSheet("border: 2px solid black; color: black; background: whit
e;");
```

3.4 进程管理设计说明

3.4.1 程序单元说明



进程模块负责对进程进行管理，主要完成的功能有：

一、 进程状态切换

在进程的生命周期中共包含七个状态：

- 创建：进程正在被创建。
- 执行：指令正在被执行。
- 阻塞：进程等待某个事件的发生（如 I/O 完成或收到信号）
- 就绪：进程等待分配处理器。
- 终止：进程完成执行。
- 就绪挂起：进程不参与 CPU 调度，但进程仍在内存中。
- 阻塞挂起：进程不参与 CPU 调度，且进程处于外存中。

当新建一个进程后，进程就由创建态转换到就绪态，等待分配处理器。通过进程调度 算法将选中的进程由就绪态转变成执行状态，进程开始执行。进程处于执行状态时，若发生中断或时间片用尽则回到就绪状态；若产生 I/O 操作或等待事件，则从运行态转换到阻塞态，等待 I/O 操作或事件的完成，进而转换到就绪态。在程序运行的过程中，如果接收到退出指令，或者程序运行完毕，则转入终止状态，释放进程占用的资源。特别的，当系统内存不足时，为了缓和内存紧张状况，选取部分进程将其挂起；当然用户也可主动将进程挂起。

二、 进程调度算法

对于进程的调度常分为长期调度程序和短期调度程序。前者从磁盘的作业池中选择进程调入内存以准备执行，后者从准备执行的进程中选择进程并为之分配 CPU。查阅资料后，我们了解到 UNIX 和 Windows 的分时操作系统均没有长期调度程序，而只依赖短期调度程序维护系统进程，所以我们的 PD-OS 同样采取**只有短期调度**的方式。

a) FCFS 算法

先请求 CPU 的进程先分配到 CPU。

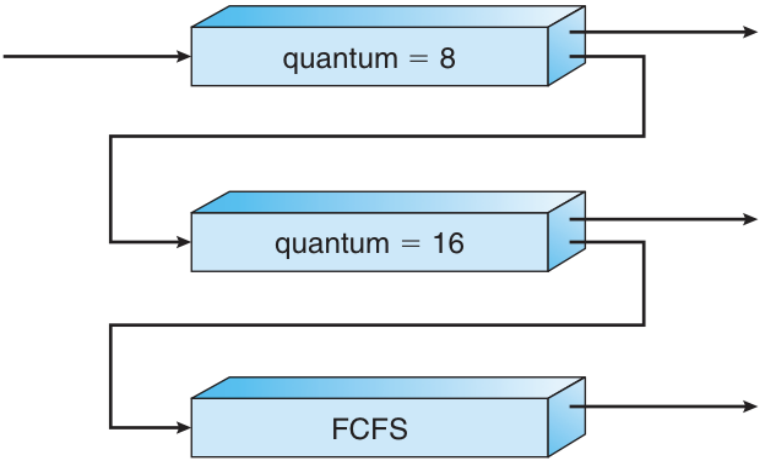
该算法利用 FIFO 队列来实现：当一个进程进入到就绪队列，其 PCB 链接到队列的尾部。当 CPU 空闲时，CPU 分配给位于队列头的进程，接着该运行进程从就绪队列中删除。

b) SJF 算法——支持抢占、非抢占两种模式

具有最短 CPU 区间的进程先分配到 CPU，若两进程具有相同的 CPU 区间长度，则按照 FCFS 原则来调度。

在非抢占模式下，只有当 CPU 空闲时才会使用调度算法从就绪队列中选择进程来运行；而在抢占模式下，CPU 空闲、有新的进程加入到就绪队列两种情况都会启用调度算法。

c) 多级反馈队列调度



我们将进程的优先级分为三等：0、1、2（0 为优先度最高），按照优先度分为 3 个队列。其中，队列 0 和队列 1 采用 RR 轮转算法，队列 0 每个进程拥有 8 个时间片，队列 1 则是 16 个（具体时间片数量和比例将根据今后 PD-OS 的实际运行情况进行调整）；而队列 2 采用 FCFS 算法。调度程序将首先执行队列 0 中的所有进程，只有 0 为空时，才能执行 1，类似的有队列 1 和队列 2。到达队列 1 的进程会抢占队列 2 的进程，类似有队列 0 和队列 1。队列 0 中的进程如果不能在规定的时间片中完成，就会被移到队列 1 的尾部；而如果队列 0 为空，队列 1 的头部进程将得到时间片，不能完成则会被抢占，并放到队列 2 中；当队列 0 和 1 均为空时，队列 2 按照 FCFS 执行进程。

三、 进程同步的实现

在我们设计的操作系统中，多个进程的临界区问题发生在对同一文件的读写操作时。为了解决这一问题，对每一个文件都设置一个状态量 is_writing，当 is_writing == 1 时表示有进程正在对该文件进行“写”的操作，此时不允许其他进程再次访问该文件。当“写”的操作结束后，将该状态量更改为 0，以此避免并发访问过程中可能的冲突。

四、 中断机制

每个进程管理块 PCB 都包含一个指令寄存器，记录该进程正在执行的指令。

在我们的操作系统中，发生中断的情况有以下几种：

- 时间片中断——时间片用尽但进程并未结束，指令寄存器的值修改为下一条指令，将该进程移入就绪队列重新等待调度，等到再次被调度时从指令寄存器显示的指令处开始执行。
- I/O 中断——修正当前进程指令寄存器的值，将该进程由运行态移入阻塞态，等待 I/O 请求完成后将其移入就绪队列等到再次被调度时从指令寄存器显示的指令处开始执行。
- 缺页中断——当前进程需要访问磁盘中的文件，修正当前进程指令寄存器的值，将该进程由运行态移入阻塞态，等待文件系统的返回信息，此后过程同上。
- 用户手动设置中断——该功能主要用于中断机制的展示，我们在 UI 界面上

设置一个中断按钮，用户可在任意时间点击此按钮中断当前正在进行的进程，系统开始执行事先设置好的一个中断处理程序，此后该进程被唤醒继续执行。我们将向用户动态展示这个过程。

对于每一种中断情形都设置对应的中断处理程序，并制成中断向量表。

3.4.2 数据结构说明

- **常量**

```
#define TimeSlice 0;  //时间片中断
#define IO 1;         //IO中断
#define PageMiss 2;   //缺页中断
#define User 3;       //用户中断
```

- **全局变量**

```
double timeslice0 = 8;   //队列0的时间片
double timeslice1 = 16;  //队列 1 的时间片
int isMLFQ = 0;  //是否选择MLFQ算法，是=1，否=0
int isFCFS = 0;  //是否选择SJF算法，是=1，否=0
int preemptive = 0;  //是否抢占，是=1，否=0，仅 SJF 模式下使用
```

- **类**

```
class PCB
{
public:
    int pid;   //进程标识符
    int state;  //进程状态
    string IR;  //指令寄存器，保存当前指令
    long PC;  //程序计数器，存放下一条指令的地址
    time_t timeArrive;   //进程开始使用CPU时间
    time_t timeLast;  //进程需要持续时间
    int priority;  //进程优先级
    int ppid;  //父进程id
    vector<int> cpid;  //子进程id
    string bufferIO;   //存储IO交互的内容
    long p_seg;  //program segment，程序段地址
    long d_seg;  //data segment，数据段地址
};

class QUEUE
{
```

```
private:
    int priority;  //队列优先级
    vector<int> queue;   //进程队列，存储进程pid
public:
    int getQueueNum(); //返回队列中的进程数量
    int getPID(int pos); //根据进程处在队列中位置(从0始)返回pid
    int shortestJob();   //遍历队列，找到最小的timeLast，返回pid
};


//多级反馈队列
//三个队列分别为队列0、队列1和队列2，对应的进程优先级分别为0、1、2（0最高）
//队列0和队列1：RR
//队列2：FCFS
class MLFQUEUE
{
private:
    QUEUE queue0;
    QUEUE queue1;
public:
    QUEUE queue2;
    int getQueueNum(int num);  //返回队列中的进程数量，参数为队列编号(取值0/1/2)
    int getPID(int queueNum, int pos); //根据进程所在队列号和处在队列中位置(从0始)返回pid
};


class PROCESS
{
    map<int, PCB> process; //用pid唯一对应PCB
    int runningPid;    //正在运行的进程
    QUEUE QueueReady;  //就绪队列——CPU调度方式为FCFS 或 SJF
    MLFQUEUE MLFQueue;
    QUEUE QueueBlock;  //阻塞队列
    QUEUE suspend;     //挂起队列

    void creatProcess(int pid, double size);  //创建进程
    void killProcess(int pid);   // 终止进程
    void runProcess(int pid);  //运行进程
    void FCFS(QUEUE QueueReady);   //first come first served
    void SJF();    //shorest job first
    void preeCPU();  //SJF==1 && preemptive==1 && 有进程进入就绪态时使用，判断是否抢占
    void MLFQ();   //multilevel feedback queue
```

```
    //进程状态迁移
        //注：如果CPU调度算法选择了MLFQ，则根据进程优先级找到进程所在三级就绪队列
            //注：运次 ——> 就绪时，先降低进程优先级(0——>1，1——>2，2不变)
        //注：如果SJF且preemptive，则每次有进程进入就绪态时，都要判断是否抢占
    void NewToReady(int pid);      //进程创建成功，进入就绪态
    void NewToSuspendReady(int pid);   //进程创建成功，但内存不足，被挂起
    void ReadyToRunning(int pid); //进程被调度，进入运行态
    void ReadyToSuspendReady(int pid); //内存不足，暂时把处于就绪态的进程挂起
    void RunningToReady(int pid); //运行态 ——> 就绪态
    void RunningToBlock(int pid); //运行态 ——> 阻塞态
    void RunningToSuspendReady(int pid); //运行态 ——> 就绪挂起态
    void BlockToSuspendBlock(int pid); //内存不足，暂时把处于阻塞态的进程挂起
    void BlockToReady(int pid);   //阻塞态 ——> 就绪态
    void BlockSuspendToReady(int pid); //阻塞挂起 ——> 就绪挂起
    void BlockSuspendToBlock(int pid); //阻塞挂起状态被激活 ——> 阻塞态
    void ReadySuspendToReady(int pid); //就绪挂起状态被激活 ——> 就绪态

};
```

3.4.3 算法及流程

主要算法：

I　//运行进程

```
void PROCESS::runProcess(int pid)
{
    //从程序计数器PC中读取程序要执行的指令所在地址
    //跟内存交互，得到当前指令IR
    //解析指令IR
    //      switch(IR)
    //      {
    //            case 运算：
    //            case Fork:
    //                新建子进程，PCB同父进程，内存独立。
    //                修正ppid和cpid
    //                子进程运行，父进程进入阻塞态（这里相当于wait()）
    //                    若是RR调度算法，则父进程进入就绪态而非阻塞态
    //            case 新建文件：
    //            case 读文件：
    //            case 写文件:
    //            case 删除文件：
```

```
//            case 访问内存:
//            case 申请键盘:
//                 IO中断
//            case 申请显示器:
//                 IO中断
//            case printer:
//                 IO中断
//      }
//
//
}
```

II //first come first served

```
void PROCESS::FCFS(QUEUE QueueReady)
{
    int num = QueueReady.getQueueNum();
    if (num == 0)
    {
        cout << "当前系统无进程可运行！" << endl;
    }
    else  //取队列头的进程运行
    {
        int pid = QueueReady.getPID(0);
        ReadyToRunning(pid);
    }
}
```

III //shorest job first

```
void PROCESS::SJF()
{
    int num = QueueReady.getQueueNum();
    if (num == 0)
    {
        cout << "当前系统无进程可运行！" << endl;
    }
    else
    {
        int pid = QueueReady.shortestJob();
        ReadyToRunning(pid);
    }
}
```

IV //SJF==1 && preemptive==1 && 有进程进入就绪态时使用，判断是否抢占

```
void PROCESS::preeCPU()
{
    int pidNew = QueueReady.shortestJob(); //找到就绪队列中时间最短的进程
    //获取其所需运行时间
    map<int, PCB>::iterator itNew = process.find(pidNew);
    time_t timeNew;
    if (itNew != process.end())
        timeNew = itNew->second.timeLast;
    else
    {
        cout << "系统发生错误！找不到pid=" << pidNew << "的PCB！" << endl;
        exit(0);
    }
    //获得runningPid的时间
    map<int, PCB>::iterator itRunning = process.find(runningPid);
    time_t timeRunning;
    if (itRunning != process.end())
        timeRunning = itRunning->second.timeLast - itRunning->second.timeArrive;
    else
    {
        cout << "系统发生错误！找不到pid=" << runningPid << "的PCB！" << endl;
        exit(0);
    }
    if (timeNew < timeRunning)  //抢占CPU
    {
        RunningToReady(runningPid);
        ReadyToRunning(pidNew);
    }
}
```

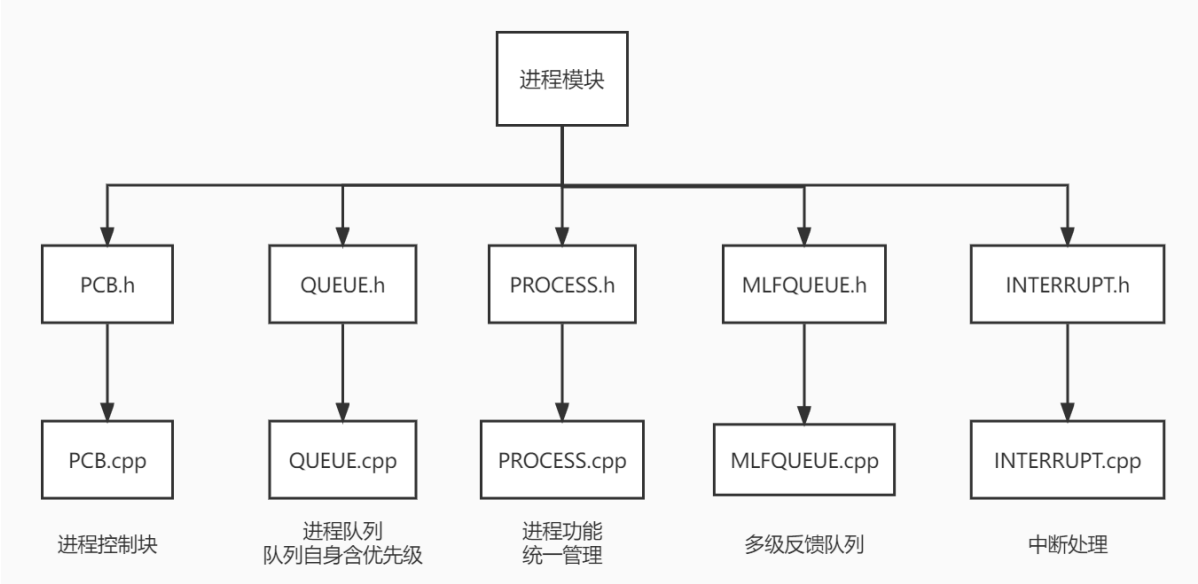
V //multilevel feedback queue

```
void PROCESS::MLFQ()
{
    int num0 = MLFQueue.getQueueNum(0);   //队列0中的进程数量
    if (num0 > 0)  //队列0采用RR算法，因此取队列头的进程运行
    {
        int pid = MLFQueue.getPID(0, 0);  //被调度的进程pid
        ReadyToRunning(pid);
    }
```

```
        else
        {
            int num1 = MLFQueue.getQueueNum(1);    //队列1中的进程数量
            if (num1 > 0)  //队列1采用RR算法，因此取队列头的进程运行
            {
                int pid = MLFQueue.getPID(1, 0);  //被调度的进程pid
                ReadyToRunning(pid);
            }
            else  //队列2采用FCFS调度算法
            {
                FCFS(MLFQueue.queue2);
            }
        }
    }
```

3.4.4 源程序文件说明

进程模块又可以细分为以上五个模块，各模块的功能如下图所示。



3.4.5 函数说明

模块	函数	功能
QUEUE.h	int getQueueNum();	//返回队列中的进程数量
	int getPID(int pos);	//根据进程处在队列中位置(从0始)返回pid
	int shortestJob();	//遍历队列，找到最小的timeLast，返回pid
MLFQUEUE.h	int getQueueNum(int num);	//返回队列中的进程数量，参数为队列编号(取值0/1/2)
	int getPID(int queueNum, int pos);	//根据进程所在队列号和处在队列中位置(从0始)返回pid
INTERRUPT.h	void interruptTimeSlice(int pid);	//处理时间片中断
	void interruptIO(int pid);	//处理IO中断
	void interruptPageMiss(int pid);	//处理缺页中断
	void interruptUser(int pid);	//处理用户中断
PROCESS.h	void creatProcess(int pid, double size);	//创建进程
	void killProcess(int pid);	//终止进程
	void runProcess(int pid);	//运行进程
	void FCFS(QUEUE QueueReady);	//first come first served
	void SJF();	//shorest job first
	void preeCPU();	//SJF==1 && preemptive==1 && 有进程进入就绪态时使用，判断是否抢占
	void MLFQ();	//multilevel feedback queue
	void NewToReady(int pid);	//进程创建成功，进入就绪态
	void NewToSuspendReady(int pid);	//进程创建成功，但内存不足，被挂起
	void ReadyToRunning(int pid);	//进程被调度，进入运行态
	void ReadyToSuspendReady(int pid);	//内存不足，暂时把处于就绪态的进程挂起
	void RunningToReady(int pid);	//运行态 ——> 就绪态
	void RunningToBlock(int pid);	//运行态 ——> 阻塞态
	void RunningToSuspendReady(int pid);	//运行态 ——> 就绪挂起态
	void BlockToSuspendBlock(int pid);	//内存不足，暂时把处于阻塞态的进程挂起
	void BlockToReady(int pid);	//阻塞态 ——> 就绪态
	void BlockSuspendToReady(int pid);	//阻塞挂起 ——> 就绪挂起
	void BlockSuspendToBlock(int pid);	//阻塞挂起状态被激活 ——> 阻塞态
	void ReadySuspendToReady(int pid);	//就绪挂起状态被激活 ——> 就绪态

3.5 文件管理设计说明

3.5.1 数据结构说明

①模块内的全局变量

```
static int MAX_INDEX;        //iNode 最多有多少直接索引
int POS_POINTER;        //当前位置指针
文件属性（文件控制块）
class FCB
{
string name;            //文件名，在每个目录下唯一
char type;          //目录(.f)、程序文件(.e)、用户文件(.t)
pointer;            //从根目录开始的绝对路径，指向所在的目录
int size;           //文件长度，每次写操作后都需要更新
```

int auth;　　//权限，无限制（auth=0），不可删除（auth=1），只读（auth=2）
vector<int> index;　//索引，第 i 个元素表示文件第 i 块地址，限制最多 MAX_INDEX 块
int singleIndirect;　//一级间接索引，对大文件可用
成员函数见 4.1.2
}

②打开文件表

class OftEntry
{
FCB file;　　//文件控制块副本，包含文件信息
int readFlag;　//读文件标识符，计数信号量，标识正在读的进程数量
int writeFlag;　　//写文件标识符，二元信号量，一次只允许一个进程写入
成员函数包括更改两个信号量的函数，以及返回 FCB 中的一些信息
}
class OpenFileTable
{
vector<OftEntry> OFT;　//打开文件表的主体，包括所有打开文件的 FCB 副本、读写情况
成员函数：新增/删除条目
}

③目录

逻辑上：树形结构，根节点为根目录，每个节点的孩子标识目录下的文件，叶子结点标识可执行文件或文本文件
存储上：哈希表
class directoryTree
{
unordered_map<string, void *> directoryTree
每个目录下的内容用哈希表存储，key 是文件名，内容是一个无类型指针：
若文件为目录，则指针类型为 unordered_map*，指向另一个哈希表；
若文件为.t 或.e，则指针类型为 FCB*，指向文件控制块
成员函数见 4.1.2
}

④空闲空间表

vector<int> freeSpaceList
用位向量 bit vector 实现，位数等于磁盘块数，每位标识一个磁盘块
每位只能为 0 或 1，0 表示块不可用，1 表示块为空

3.5.2　算法及流程

①文件系统接口

- 创建文件

函数名：createFile (string fileName)
参数：文件名称
调用：创建文件时被进程调用
流程：在当前目录创建新的文件。从参数中获取文件名称，检验名称合法性，名称重复返回-1，名称非法返回-2。创建新 FCB，写入文件名、类型、位置，根据位置设置权限，初始化长度、索引为零，成功返回值为 0。

- 删除文件
  函数名：deleteFile (string fileName)
  参数：文件名称
  调用：删除文件时被进程调用；删除多级目录时被自身递归调用
  流程：删除已有文件。从参数中获取文件名称，检验文件权限，无权限返回-1；检验读写标志，若正在被读写则返回-2；若为.e 或.t 文件，则调用 findFile()读取对应的 FCB，按照索引顺序删除并释放数据块，最后释放索引块；若为.f 目录文件，则顺序读取文件，对目录下的每一项递归调用 deleteFile()自身，直至完成整个目录的删除，成功返回值为 0。

- 读取文件
  函数名：readFile (string fileName, int blockNum, int targetAddress)
  参数：文件名称、块号、虚存的真实地址
  调用：读取文件时被进程调用；新建进程时被内存调用
  流程：调用 findFile()找到对应文件的 FCB，若写入标识符不为 0，则返回-1，不改变标识符的值；否则，打开文件表对应的读文件标识符+1，读取目标文件的对应数据块，若已经是最后一块则返回 0，读文件标识符-1；若还有后续数据块，则返回-2，不改变读文件标识符。

- 写入文件
  函数名：writeFile(string fileName, int memoryAdress, int mode)
  参数：文件名称、内存地址+偏移量、写入模式
  调用：写入文件时被进程调用
  流程：调用 findFile()获取目标文件 FCB，若读或写标识符不为 0，则返回-1；否则，修改写标识符为 1。从内存地址处获取写入内容。若写入模式为“覆写”，则先按索引顺序删除文件块的内容并释放除第一块外的所有数据块，在第一块内写入；若写入模式为“在文件尾写入”，则获取最后一个文件块的内容，并在内容后连接新内容。写入时，用文件长度除以块大小，根据所需数量进行数据块的申请和分配。写入结束后，将写入标识符修改为 0，返回值为 0。

- 查找文件
  函数名：findFile (string fileName)
  参数：文件名称
  调用：被文件系统的上述接口调用；创建新进程时，被 kernel 调用，返回文件大小以判断内存空间是否足够；被内存调用确定是否已经在内存
  流程：根据参数中的文件名，通过哈希函数找到对应的 FCB，并返回。

- 修改目录
  函数名：changeRoute(string route)
  参数：新的路径
  调用：UI 每一次改变目录，kernel 告知文件系统
  流程：根据输入修改当前位置指针 POS_POINTER。

- 打印目录
  函数名：printRoute()
  调用：被 UI 调用
  流程：根据 POS_POINTER 打印当前路径到 UI 界面。

- 展示当前目录下所有条目
  函数名：printDir()
  调用：被 UI 或 kernel 调用
  流程：根据 POS_POINTER，遍历当前目录，并且递归打印目录及子目录下所有文件名称。

- 查找当前目录下符合某种条件的所有文件
  函数名：printFile(char fileType)
  调用：被 UI 调用
  流程：选择当前目录下类型为 fileType 的所有文件，以 List 形式返回。

②逻辑存储结构

- 选用索引分配，为简化操作，将索引块并入 FCB。
- 每个索引块的最后一条为一级间接块。考虑到本 OS 规模，应该不需要更多层次的间接块。
- 与真实操作系统不同的是，真实 OS 需要内存先读入索引再根据索引读文件，但因为我们限制了文件操作的复杂度，所以文件系统会遍历索引段将文件块全部载入内存。

3.5.3 数据存储说明

一方面，磁盘具有非易失的特点，要求每一次重启操作系统，磁盘上的数据不应丢失或损坏；另一方面，频繁的文件操作效率低、性能差。因此我们决定在每一次关闭操作系统时，将文件树和所有文件以文件的形式写入程序所在文件夹；重新启动时，作为初始化，将文件树和存储的文件载入内存，后续的磁盘读写操作在程序内存中实现。

需要存储的文件：目录结构、对应每个文件的 txt 文档。

3.5.4 源程序文件说明

file.cpp, file.h: 实现文件类和 3.1.3 中的部分函数

fcb.cpp, fcb.h: 实现文件控制块 FCB
oft.cpp, oft.h: 实现文件打开表 OpenFileTable 和 OptEntry
dir.cpp, dir.h: 实现树形目录结构 directoryTree 和 3.1.3 中只涉及目录/路径的函数
fsl.cpp, fsl.h: 实现空闲空间表 freeSpaceList

4 总结

在详细设计的这两周里，我们组织了一次项目会议，对 PD-OS 各个模块之间的配合做出了进一步细化的讨论和设计，也对之前疏忽错落的地方进行了更正和改进，如：①配合进程 swap 的具体操作，把进程的五状态模型更改为七状态模型；②为了实现内存管理的缺页功能在指令集中加入访问内存指定逻辑地址的指令；③我们考虑到计算机以进程的形式完成自己各个功能，于是决定以进程的方式完成文件相关操作；④对 Qt 和 C++的交互进行了深入研究，对信号和槽的机制进行了了解……项目会议后，成员们各自对自己负责的部分进行了细化和完善。

目前，我们的详细设计还有一定缺陷：各模块之间的接口还未实际测试，创建进程时内存分配后将必要的进程页调入内存的选择还未确定……在接下来的编码测试阶段，我们会继续完善代码设计，统一代码风格，完成单元测试和集成测试。

5 附录

5.1 PD-OS 的 exe 可执行文件中指令格式规定

名称	格式	实例	意义
运算	Computing+时间长度	Computing 10	需求 cpu10 秒进行运算
Fork	Fork	Fork	创建子进程
新建文件（待定）	NewFile+文件名称	NewFile xx.txt	新建 xx.txt 文件
读文件	ReadFile+文件名称	ReadFile xx.txt	读 xx.txt 文件
写文件	WriteFile+文件名称	WriteFile xx.txt	在 xx.txt 文件后添加内容
删除文件	DelFile+文件名称	DelFile xx.txt	删除 xx.txt
访问内存	Access + 地址	Access 150	访问逻辑地址 150 的内存单元
申请键盘	Keyboard	Keyboard	申请 keyboard 资源
申请显示器	Monitor	Monitor	申请显示器资源
申请 printer	Printer	Printer	申请 printer 资源

用户输入：delete xx.txt → kernel 解析指令 → kernel 把文件名称传给文件管理 → 文件管理判断文件是否存在 →存在则 kernel 调度 deleteFile.exe 程序 → 程序进入进程管理开始运行 → 程序中指令逐条执行 → 删除文件指令把要删除文件的名称传给文件管理

→ 文件系统返回删除成功 → 进程完毕

5.2 PD OS 规格参数：

CPU 个数：1
内存大小：512MB
外存大小：2GB
内存块=页面大小=磁盘块=1KB