徽标

中度可信度描述已自动生成

**《操作系统原理》**

**课程设计报告**

学 号： 20211004354

班级序号： 111212

姓 名： 李元

指导教师： 胡茂胜

成 绩：

**中国地质大学(武汉)计算机学院**

**2023年 6月 4日**

1. **需求规格说明**

**1、实现管理工具和服务程序：**实现本地客户端，服务端，利用进程通讯技术，实现客户端指令发送，服务端响应执行并发回结果,类似命令行形式的通讯响应；

**2、实现系统状态监控：**以一定时间间隔监控电脑 CPU 状态、内存分配情况、进程状态、磁盘利用率，以及本系统管理的模拟文件系统空间利用率、作业状态等信息；

**3、实现虚拟文件系统：**在单个文件中实现一个模拟的文件系统，包括多级目录和子文件，文件系统采用分页组织，并具有 1M 大小的分页管理的内存缓存，缓存要求采用最近最久未使用（LRU）页面置换算法。模拟的文件系统要提供格式化、枚举目录和子文件、创建目录和子文件、子文件读写、子文件删除、目录删除等功能；

**4、实现作业系统：**进行作业的模拟，实现多作业运行，通过管理工具程序实现创建新作业，查看作业的运行状态和完成比率，暂停或终止作业，查看作业日志等功能。

**2．系统分析与设计**

（1）设计思想：

**1、管理工具和服务程序的设计：**

通信采用套接字通信，实现客户端服务端的双向通讯，双方都能实现一发一收，便于客户端指令的发送和结果的接收；

在服务端进行系统状态监控，构建函数**PrintSystemStatus()**，用于打印系统状态信息，包括任务信息、CPU信息、内存信息、磁盘信息以及创建的虚拟文件系统信息和作业系统信息，系统状态使用Windows系统调用函数获取，虚拟文件系统信息和作业系统信息根据创建的存储结构数据进行获取。

**2、虚拟文件系统的设计：**

用顺序表管理磁盘信息，将一个本地txt文件分成三部分，第一部分存储目录和文件路径信息，第二部分存储inode（节点的详细信息（也就是FCB）,用于储存文件的磁盘存储信息），最后的部分是储存文件内容的空间，根据相应的存储结构体对目录和文件进行操作，并写回磁盘，达到文件系统模拟的功能。

存储结构：

//磁盘

typedef struct Disk

{

int state = 0;//磁盘的状态，0为空，1为满

string content;

}Disk;

Disk disk[D\_N];

//节点的详细信息

typedef struct Inode

{

int dir\_num;

float pos\_start = -1;

float size = 0;//文件的字符长度

string content;

}Inode;

//一个文件目录项

typedef struct Dentry

{

int type = 1;//文件类型 文件 0 目录 1

int last = -1;

int num = 0;

string name = "";

struct Inode inode;

string isdir = "<Dir>";

string time = "";

}Dentry;

**3、作业系统设计：**

创建作业结构体记录作业的各项信息，并使用一系列函数改变结构体信息实现对作业执行操作，使用线程模拟作业的运行，将一个循环（每秒对作业进度加2%的循环）作为执行作业的过程进行线程封装，并分离线程，使其能够分离运行，达到多作业运行的效果。

存储结构：

struct Job {

int jobID;

int PID;

string program;

string status;

int progress;

int now\_pro;

Job(int id, int pid, const string& prog) : jobID(id), PID(pid), program(prog), status("created"), progress(0), now\_pro(0) {}

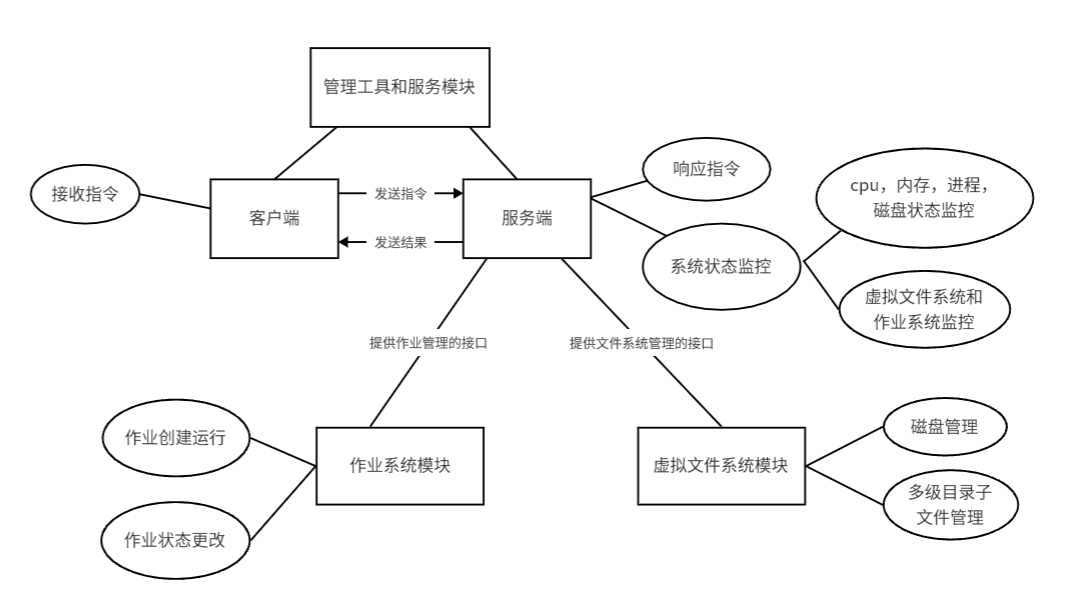
};

vector<Job> jobs; //多作业存储

unordered\_map<int, int> jobIDtoIndex; //作业ID匹配

2）设计表示：

**系统设计图：**



**部分函数的规格说明以及流程说明：**

系统状态监控：

函数名称：**PrintSystemStatus**

返回类型：void

参数：无

功能：查询并打印系统状态信息，包括任务信息、CPU信息、内存信息和文件系统信息。

函数流程：

1. 查询并打印任务信息：

创建一个大小为1024的DWORD数组processIds，用于存储进程ID。

调用EnumProcesses函数查询进程ID，并将结果存储到processIds数组中，同时获取进程数目存储到processCount变量中。

定义并初始化统计各类任务数量的变量：taskCount、runningCount、sleepingCount、stoppedCount和zombieCount。

使用循环遍历进程ID数组，对每个进程进行处理：

调用OpenProcess函数打开进程，并获取进程句柄hProcess。

如果进程句柄有效，通过GetExitCodeProcess函数获取进程的退出码exitCode，并根据退出码判断进程的状态：

如果进程仍处于活动状态（STILL\_ACTIVE），增加runningCount计数器。

否则，根据退出码的值将进程归类为停止状态（exitCode == 0）或僵尸状态（其他值），并相应地增加对应状态的计数器。

关闭进程句柄hProcess。

打印任务信息，包括任务总数、运行中的任务数、睡眠中的任务数、停止的任务数和僵尸任务数。

1. 查询并打印CPU信息：

创建一个SYSTEM\_INFO结构体变量systemInfo，用于存储系统信息。

调用GetSystemInfo函数获取系统信息，并将结果存储到systemInfo变量中。

获取CPU核心数目并存储到processorCount变量中。

定义idleTime、kernelTime和userTime变量，用于存储系统空闲时间、内核时间和用户时间。

调用GetSystemTimes函数获取系统时间，并将结果存储到相应变量中。

通过休眠1秒钟，等待一段时间。

再次调用GetSystemTimes函数获取系统时间，并计算时间差，得到CPU使用率。

打印CPU使用率信息，包括用户使用率、系统使用率和空闲使用率。

1. 查询并打印内存信息：

创建一个MEMORYSTATUSEX结构体变量memStatus，用于存储内存信息。

设置memStatus的dwLength成员为结构体大小。

调用GlobalMemoryStatusEx函数获取内存信息，并将结果存储到memStatus变量中。

打印内存使用情况，包括总内存量、可用内存量、已使用内存量和缓存量。

1. 查询并打印文件系统信息：

定义freeBytesAvailable、totalBytes和totalFreeBytes变量，用于存储文件系统信息。

设置根路径rootPath为"E:"。

调用GetDiskFreeSpaceExA函数获取文件系统信息，并将结果存储到相应变量中。

计算文件系统使用率。

打印文件系统信息，包括总空间、已使用空间、可用空间、使用率和挂载路径。

1. 打印磁盘信息：

打印磁盘数量、磁盘大小、磁盘空闲空间、磁盘使用率和挂载路径。

1. 打印作业信息，包括作业总数、运行中的作业数、成功的作业数、取消的作业数和暂停的作业数。

虚拟文件系统：

1、函数名称：**show\_pos**

返回类型：void

参数：无

功能：显示当前的路径目录。

函数流程：

1. 定义变量num并赋值为全局变量number，该变量表示当前路径目录的索引号。
2. 定义字符串变量paths并初始化为空字符串。
3. 进入循环，直到num的值为-1：

检查当前路径目录的类型：

如果是目录（type == 1）且名称不为"root"，则打印目录的时间、是否为目录（isdir）、路径符号"/"和目录名称。

如果是文件（type == 0），则打印文件的时间、大小和文件名称。

将当前目录的名称添加到paths的开头，以构建路径字符串。

更新num为当前目录的父目录的索引号（last）。

1. 打印路径字符串paths。

2、函数名称：**out\_file**

返回类型：void

参数：无

功能：将文件系统的内容输出到存储地址空间。

函数流程：

1. 将位于磁盘索引128到255之间的存储空间标记为未使用（state = 0）并清空内容（content = ""）。
2. 打开文件流OutFile，将文件系统的内容输出到文件"Test.txt"。
3. 遍历目录项数组dentry：

将目录项的相关信息转换为字符串格式，包括上级目录索引（last）、目录索引（num）、类型（type）和名称（name）。

将目录项信息存储到磁盘中的空闲位置。从磁盘索引0开始，找到第一个未使用的磁盘空间（state = 0），将目录项信息存储到该位置，然后将磁盘空间标记为已使用（state = 1）。

如果目录项表示的是文件（type == 0），还需要将文件的inode信息存储到磁盘中。将inode的相关信息转换为字符串格式，包括起始位置（pos\_start）、大小（size）和目录索引（dir\_num）。类似地，找到磁盘中的下一个未使用位置，存储inode信息，并将磁盘空间标记为已使用。

1. 遍历磁盘中的所有存储空间（索引0到D\_N-1），将每个存储空间的内容写入文件流OutFile中，每个内容占一行。

3、函数名称：**upload\_file**

返回类型：void

参数：name\_file：要上传的本地文件路径（字符串）

功能：从本地文件系统中上传文件到虚拟文件系统当前目录。

函数流程：

1. 打开名为name\_file的本地文件。
2. 如果无法打开文件，则输出错误信息提示无法打开文件。
3. 在当前目录中找到一个空闲的目录项位置，即dentry数组中的一个索引i，使得dentry[i].name为空字符串。
4. 获取当前系统时间，并将其格式化为字符串time，用于设置新文件的创建时间。
5. 设置目录项的相关属性：

dentry[i].time：设置为上述格式化的时间字符串。

dentry[i].name：设置为name\_file，即上传文件的名称。

dentry[i].last：设置为当前目录的索引number，表示该文件的上级目录为当前目录。

dentry[i].num：设置为目录项的索引i。

dentry[dentry[i].num].type：设置为0，表示该目录项为文件类型。

1. 从打开的文件中读取文件内容，并存储到字符串变量content中。
2. 设置目录项的inode属性：

dentry[dentry[i].num].inode.content：设置为文件内容字符串content。

dentry[dentry[i].num].inode.size：设置为文件内容的大小。

dentry[dentry[i].num].inode.dir\_num：设置为目录项的索引num，表示该文件所在的目录项索引。

dentry[dentry[i].num].inode.pos\_start：调用函数find\_pos(dentry[dentry[i].num].inode.size)找到文件内容在磁盘上的起始位置。

如果找到了足够的空闲磁盘空间，返回其起始位置索引；否则返回-1，表示没有足够的存储空间。

1. 如果dentry[dentry[i].num].inode.pos\_start不等于-1，表示找到了足够的存储空间：

计算需要的磁盘块数量n，其中每个磁盘块大小为disk\_size。

遍历磁盘区域，从dentry[dentry[i].num].inode.pos\_start到dentry[dentry[i].num].inode.pos\_start + n - 1，将文件内容分块写入磁盘。

作业系统模块：

1、 函数名称：**run**

返回类型：void

参数：jobID：作业的ID（整数）

功能：运行指定ID的作业。

函数流程：

1. 根据作业ID jobID 在jobIDtoIndex映射中找到对应的索引t。
2. 检查作业状态是否为"created"或"running"。
3. 如果作业状态为"created"或"running"，则创建一个新线程来模拟作业的执行过程。

新线程会执行以下操作：

获取作业索引：根据作业ID jobID 在jobIDtoIndex映射中找到对应的索引index。

将作业状态设置为"running"。

通过循环模拟作业执行过程：

更新作业的进度jobs[index].progress，递增步长为2，直到达到100减去当前进度jobs[index].now\_pro。

线程休眠1秒钟，模拟作业执行的时间。

如果作业进度达到100且状态仍为"running"，则将作业状态设置为"success"。

1. 分离新创建的线程，使其在后台运行。
2. 如果作业状态是"canceled"或"success"，则终止程序。

2、函数名称：**logJob**

返回类型：void

参数：

jobID：作业的ID（整数）

功能：记录作业的相关信息。

函数流程：

1. 根据作业ID jobID 在jobIDtoIndex映射中找到对应的索引index。
2. 输出作业创建的日志信息：

输出信息："job:create job "，后接作业ID jobID。

输出信息："fork:sub process ID "，后接作业的进程ID jobs[index].PID。

输出信息："exec:"，后接作业的程序名称 jobs[index].program。

1. 进行作业的日志记录循环，循环5次：

输出信息："message:sleep 1 second"，表示作业正在等待1秒钟。

检查作业的状态：

如果作业状态为"running"，则输出信息："progress ratio:"，后接作业的进度 jobs[index].progress，再后接"%"。

如果作业状态不是"running"，则输出信息："progress ratio:"，后接作业的当前进度 jobs[index].now\_pro，再后接"%"。

线程休眠1秒钟，同步等待模拟作业的执行时间。

1. 输出作业的最终状态：

输出信息："job:"，后接作业的状态 jobs[index].status。

3）详细设计表示：

内存数据结构：

文件系统：

//磁盘

typedef struct Disk

{

int state = 0;//磁盘的状态，0为空，1为满

string content;

}Disk;

Disk disk[D\_N];

//节点的详细信息

typedef struct Inode

{

int dir\_num;

float pos\_start = -1;

float size = 0;//文件的字符长度

string content;

}Inode;

//一个文件目录项

typedef struct Dentry

{

int type = 1;//文件类型 文件 0 目录 1

int last = -1;

int num = 0;

string name = "";

struct Inode inode;

string isdir = "<Dir>";

string time = "";

}Dentry;

作业系统

struct Job {

int jobID;

int PID;

string program;

string status;

int progress;

int now\_pro;

Job(int id, int pid, const string& prog) : jobID(id), PID(pid), program(prog), status("created"), progress(0), now\_pro(0) {}

};

vector<Job> jobs; //多作业存储

unordered\_map<int, int> jobIDtoIndex; //作业ID匹配

外存文件格式：

虚拟文件系统依赖文件：Test.txt；

Upload（从本地文件系统到虚拟文件系统）文件：只支持上传txt文件；

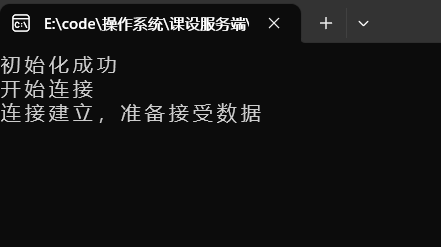
Download（从虚拟文件系统到本地文件系统）文件：只支持下载到服务端同目录下的同名txt文件；



界面设计：

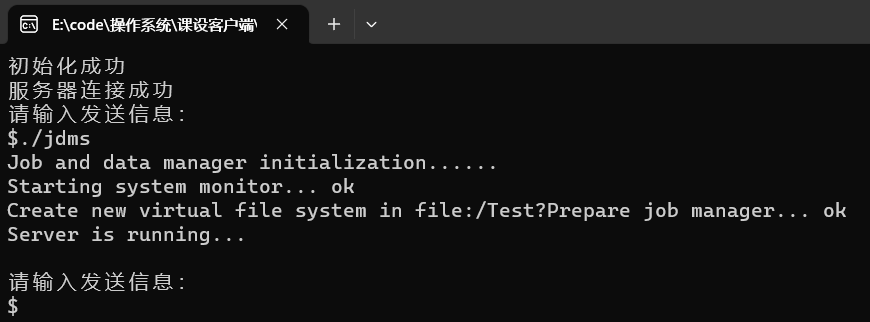
服务端界面：

展示通讯连接情况。



客户端界面：

展示通讯连接情况，正确连接后循环输出$符号等待用户输入指令，指令后会换行输出结果，并再次提示输入指令。



**3．编码**

**问题1：**使用套接字通讯，接收和发送缓冲区都使用的是char类型的固定大小数组，char类型的数组初始则全是空格符，导致指令发送到服务端后读取如果以空格作为结尾符读取指令会导致指令获取不完全；

解决方法：考虑到文件系统和作业系统还同时需要在指令基础上获取文件路径名和作业ID等数据，则将指令改成用不同分隔符连接的组合指令；

令./jdmc-vfs-mkdir\*aa，即表示在当前目录下创建名为aa的子目录，在服务端接收该指令后就以\*符作为分隔符分割指令，前半部分判断指令类型，后半部分作为参数调用相应函数进行响应。

**问题2：**在执行download指令下载虚拟文件系统中的文件时，由于参数必须是绝对路径，防止不同子目录下有相同文件名的文件，导致下载下来的文件名称保留了整个绝对路径的名称；

解决方法:改变函数逻辑，先根据参数传入的绝对路径找到文件并获取到文件内容，再对参数进行处理，编写调用函数获取到绝对路径的最后部分的文件名：

string GetFilenameFromPath(const string& path) {

size\_t lastSlash = path.find\_last\_of("/\\");

if (lastSlash != string::npos) {

return path.substr(lastSlash + 1);

}

return path;

}

再在本地创建同名文件并将获取到的文件内容写入到本地文件，进而实现下载功能。

**问题3：**每次对虚拟文件系统进行操作后，再次打开虚拟文件系统后，上次创建的文件和目录都没能保存；

解决方法：经过观察和调试后发现，虽然使用了本地文件作为依赖创建虚拟文件系统，但是在实际对文件目录等的操作是直接在创建的结构体层面进行的，并没有真正的写入本地文件，也就是没有写入磁盘，导致操作没有保存，因此创建函数out\_file（具体实现见附件），将文件系统的内容输出到存储地址空间，通过打开文件Test.txt向其中指定位置添加当前已创建的目录文件等内容，等到再次打开虚拟文件系统后就可以再次查看以前的文件系统状态。

**问题4：**由于虚拟文件系统和作业系统与服务端客户端是分开进行编码的，函数的响应输出也直接在对应模块输出，整合的时候把虚拟文件系统和作业系统的响应放在服务端后，需要将所有函数的输出换成使用的通讯方法进行逐一发送到客户端。

解决方法：为了减少工作量，不逐个改变系统的函数，对服务端的信息发送逻辑进行了更改，首先创将 cout 的输出捕捉到一个内存中的流对象 buffer 中，而不是直接输出到控制台：

stringstream buffer;

streambuf\* oldCoutBuffer = cout.rdbuf(buffer.rdbuf());

然后判断指令类型，进行对应的函数调用响应，最后将获取到的cout信息转变成string类型再发送给客户端：

// 从缓冲区中获取输出信息

string output = buffer.str();

send(sockCli, output.c\_str(), sizeof(sendbuf), 0);

这样操作后就可以直接获取到响应的cout信息作为指令结果发送给客户端，实现了响应功能，同时不用挨个改变函数的输出。

**问题5：**在作业系统中，由于使用线程模拟作业，在暂停和恢复作业时只改变作业状态不能使线程暂停，进度仍然在增加，如果直接终止进程再重新创建作业又无法记录已经执行的进度，导致又从0开始执行；

解决方法：在作业的存储结构体中增加一个变量now\_pro保存作业当前的作业进度初值为0，在执行暂停作业时，就将当前进度保存到now\_pro变量中，然后在作业运行函数run中，首先判断作业状态是running和created的作业才能运行，其他状态终止线程，同时在作业模拟中初始进度设置为now\_pro，循环更改为：

for (int i = 0; i <= 100 - jobs[index].now\_pro; i += 2) {

jobs[index].progress = i + jobs[index].now\_pro;

this\_thread::sleep\_for(chrono::seconds(1));

}

这样在暂停时可以将进度更改为now\_pro的值，状态改为suspend，设定该作业暂停，进度不再增加，重启时将作业状态改为running再调用run函数重新运行，就可以在暂停时的进度基础上再次增加进度，实现重启功能，cancel终止作业则直接终止线程即可。

**问题6：**在作业系统实现作业清除即清除已完成和已终止作业时，使用迭代器遍历访问vector<Job> jobs数组时，发生vector数组越界的问题；

解决方法：在迭代器循环处加断点调试后发现原循环（for (auto it = jobs.begin(); it != jobs.end();it++ )）会导致多读取一个数据，导致读取到空值，迭代器出现空指针错误，正确的方式是：

for (auto it = jobs.begin(); it != jobs.end(); ) {

if (it->status == "success" || it->status == "canceled") {

it = jobs.erase(it);

}

else {

++it;

}

}

在判断中移动迭代器位置，可以有效防止空指针的错误。

**4．结果分析**

**使用说明：**

1、先运行服务端，使服务端初始化套接字；

2、再运行客户端，使客户端连接到服务端套接字，并提示用户输入指令；

3、用户在客户端输入指令；

**指令类型以及格式说明：**

\*号后加文件路径或目录文件名或作业编号；

+号后加文件内容；

1. 启动服务进程

./jdms

1. 关闭服务进程

./jdmc-server-shutdown

1. 查看系统状态

./jdmc-server-status

1. 格式化虚拟文件系统

./jdmc-vfs-format

1. 在虚拟文件系统中创建目录

./jdmc-vfs-mkdir\*test

1. 在虚拟文件系统中创建文件

./jdmc-vfs-mkfile\*test.txt+abcdef

1. 打开虚拟文件系统中的文件

./jdmc-vfs-openfile\*/root/test.txt

1. 保存当前虚拟文件系统状态

./jdmc-vfs-save

(9)复制本地文件到虚拟文件系统中（数据上载）

./jdmc-vfs-upload\*upload.txt

(10)枚举虚拟文件系统中的目录和子文件

./jdmc-vfs-ls

(11)跳转到指定目录下

./jdmc-vfs-cd\*/root/test

(12)复制虚拟文件系统中的子文件到本地文件目录中(数据下载)

./jdmc-vfs-download\*/root/download.txt

(13)删除虚拟文件系统中的目录和文件(递归删除)

./jdmc-vfs-rmdir\*/root

(14)创建新作业并启动运行

./jdmc-job-create

(15)查看作业的运行状态和完成比率

./jdmc-job-status

(16)暂停作业

./jdmc-job-suspend

(17)终止作业

./jdmc-job-cancel

(18)继续执行暂停的作业

./jdmc-job-resume

(19)查看作业日志

./jdmc-job-log

(20)清理完成和取消的作业记录及日志

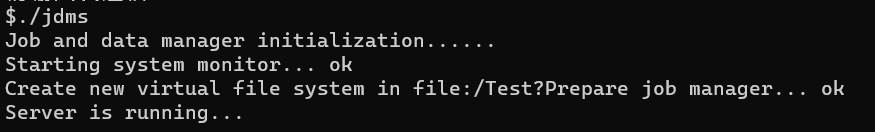
./jdmc-job-clear

**系统运行结果：**

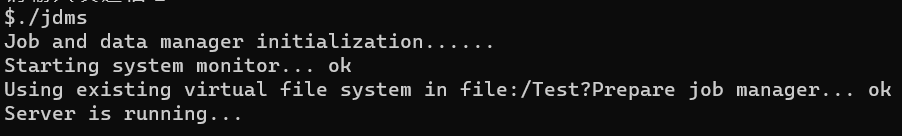
(1) 启动服务进程

./jdms

(首次运行)



(再次运行)



(2) 关闭服务进程

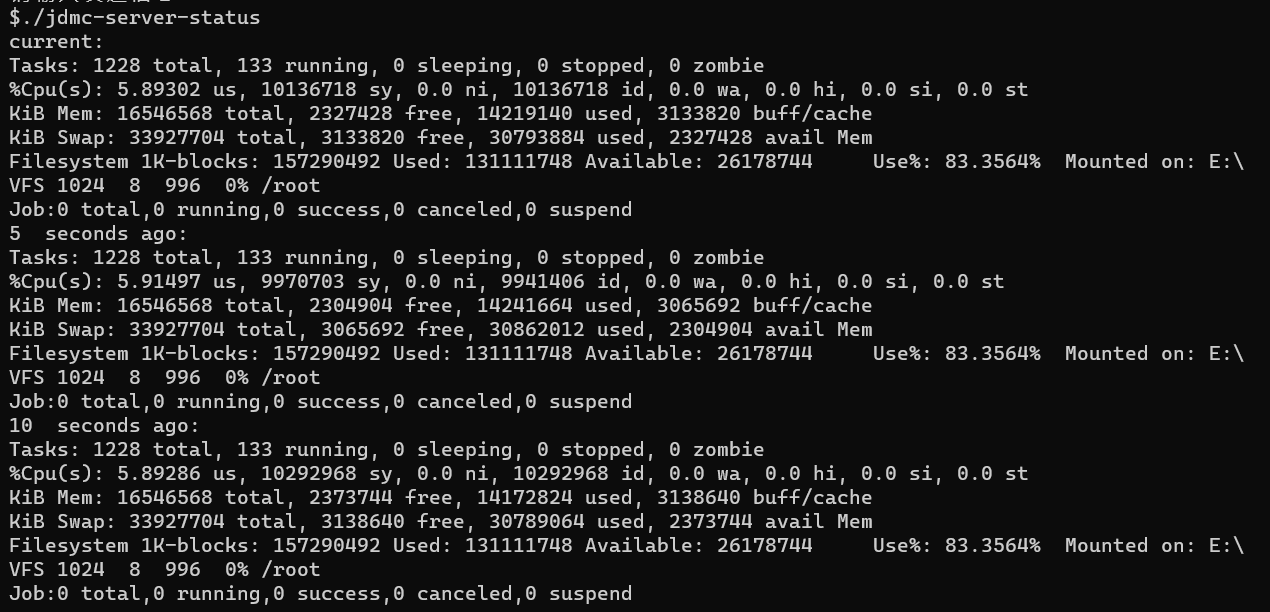
./jdmc-server-shutdown

文本

描述已自动生成

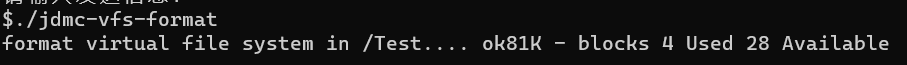
(3) 查看系统状态

./jdmc-server-status



(4) 格式化虚拟文件系统

./jdmc-vfs-format



(5) 在虚拟文件系统中创建目录

./jdmc-vfs-mkdir\*test



(6) 在虚拟文件系统中创建文件

./jdmc-vfs-mkfile\*test.txt+abcdef

文本

描述已自动生成

(7) 打开虚拟文件系统中的文件

./jdmc-vfs-openfile\*/root/test.txt

文本

描述已自动生成

(8) 保存当前虚拟文件系统状态

./jdmc-vfs-save

文本, 表格

描述已自动生成

(9)复制本地文件到虚拟文件系统中（数据上载）

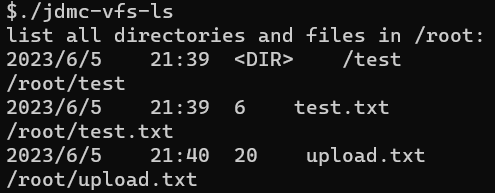
./jdmc-vfs-upload\*upload.txt

文本

描述已自动生成

(10)枚举虚拟文件系统中的目录和子文件

./jdmc-vfs-ls



(11)跳转到指定目录下

./jdmc-vfs-cd\*/root/test

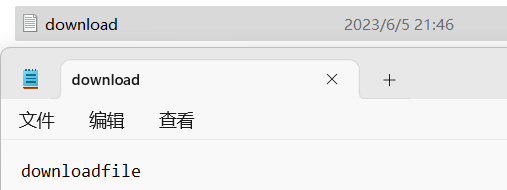


(12)复制虚拟文件系统中的子文件到本地文件目录中(数据下载)

./jdmc-vfs-download\*/root/download.txt

文本

描述已自动生成



(13)删除虚拟文件系统中的目录和文件(递归删除)

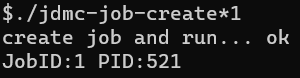
./jdmc-vfs-rmdir\*/root/test

文本

描述已自动生成

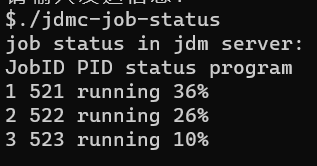
(14)创建新作业并启动运行

./jdmc-job-create\*1



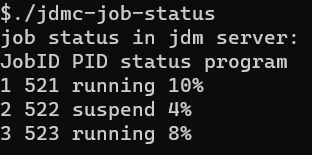
(15)查看作业的运行状态和完成比率

./jdmc-job-status



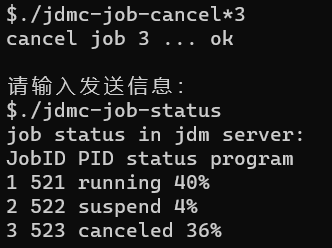
(16)暂停作业

./jdmc-job-suspend\*2



(17)终止作业

./jdmc-job-cancel



(18)继续执行暂停的作业

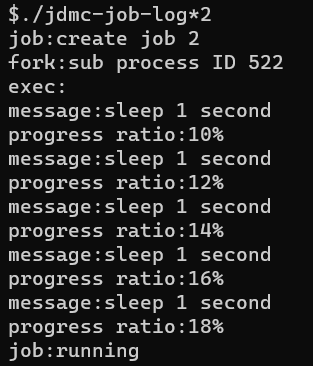
./jdmc-job-resume\*2

文本

描述已自动生成

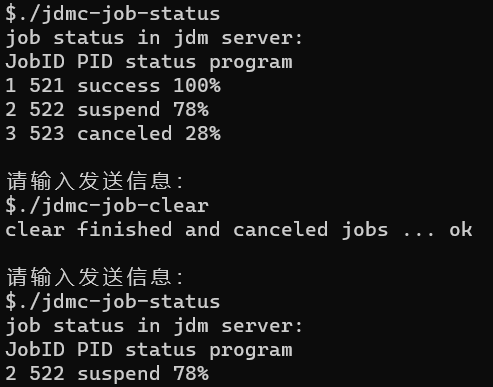
(19)查看作业日志

./jdmc-job-log\*2

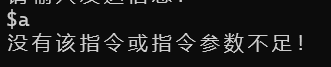


(20)清理完成和取消的作业记录及日志

./jdmc-job-clear



(21)错误指令：



**系统分析：**能够实现题目给出的所有指令和需求，所有指令都能正常执行响应，在给出指令的基础上增加了三条指令：openfile：打开文件，save：保存当前文件系统到磁盘，cd：跳转到指定目录，能够对文件系统的多级目录进行操作，提供查看文件内容的方法，并提供了错误指令的响应，以及保存状态以供下次读取的方法，基本实现了题目需求，指令响应也比较迅速，能够及时传输指令获取结果，并给出成功或失败的信息，但是文件系统使用的是顺序存储，没有使用分页存储，没有使用lru页面置换算法，作业系统没有做到与服务端分离，只能依赖于服务端运行，线程模拟也存在很多潜在问题，还有很大的改进空间。

**改进设想：**

1、整个系统虽然实现了题目要求的基础功能，但是文件系统和作业系统的实现都进行了简化，文件系统应当进一步使用分页存储的方式管理，并通过存储管理算法对缓存进行管理，作业系统可以与服务端进行分离，使其在管理工具创建作业后能够自行运行并实时传输状态信息到服务端；

2、文件系统还可以进一步实现复制，重命名，移动等功能，使功能更加完整；

3、作业系统的日志文件不仅要能输出，还可以生成本地真正的日志文件，创建本地文本文件，在作业执行过程中不断更新日志文件，能够在输出的文件中查看完整的作业运行日志；

4、可以进行ui界面设计，虚拟文件系统可以类似于Windows上的文件系统，在右键时进行相应的文件操作。

**5．小结**

**经验体会：**

在本次课程设计中，综合运用了进程控制、进程通信、文件系统调用等相关知识，完成了一个作业与存储管理系统。该系统由服务程序和管理工具程序两部分组成。

服务程序是一个常驻内存并持续运行的程序，主要提供三类服务：系统状态监控、存储管理和作业管理。管理工具程序通过套接字与服务程序进行通信，接收命令并返回结果。这样，用户可以通过管理工具程序与服务程序进行交互，实现对系统的监控和管理。

系统状态监控部分，设计了服务程序能够定期监控系统的各项状态，如CPU状态、内存分配情况、进程状态、磁盘利用率、模拟文件系统空间利用率、作业状态等。通过定期采集这些信息，并将其以一定的格式展示给用户，用户可以及时了解系统的运行情况。

存储管理部分，我在单个文件中实现了一个模拟的文件系统。这个文件系统包括多级目录和子文件，采用顺序组织方式。我设计了相应的文件系统调用接口，实现了格式化、枚举目录和子文件、创建目录和子文件、子文件读写、子文件删除、目录删除等功能。通过这些功能，用户可以对模拟文件系统进行管理和操作，实现文件的存储和管理。

作业管理部分，我使用了线程模拟作业的运行。当用户通过管理工具程序创建新作业时，服务程序会为该作业创建一个线程。在该线程中，我模拟了作业的执行过程，并通过定期更新作业的进度来模拟作业的完成情况。通过线程的并发执行，多个作业可以同时运行，提高了系统的效率和并发处理能力。用户可以通过管理工具程序查看作业的运行状态和完成比率，暂停或终止作业，以及查看作业日志等功能。

综合运用多个知识领域：这个课程设计项目要求综合运用进程控制、进程通信、文件系统调用等多个知识领域。通过将这些知识融会贯通，我深刻体会到了这些概念和技术之间的关联和互动。这种综合运用的过程让我更好地理解了操作系统的整体架构和运行机制。

理论与实践相结合：在课堂上学到的理论知识通过实际的项目实现得到了验证。通过实践，我更深入地理解了理论知识的应用场景和实际运用方法。同时，我也发现了一些理论与实践之间的差距和挑战，这促使我深入思考和探索更好的解决方案。

深入理解进程通信和线程：项目中使用了进程通信和线程来实现不同模块之间的协作和并发处理。这让我更深入地理解了进程通信和线程的概念、特性和应用场景。通过实际编码和调试，我体会到了进程通信和线程同步的实现机制。

设计思维和系统思维：在整个项目的设计和实现过程中，需要充分发挥设计思维和系统思维。设计思维使我能够合理规划和组织代码结构，考虑系统的可扩展性和可维护性。而系统思维则让我能够从整体的角度来思考和分析问题，理解系统的各个模块之间的相互作用和影响。

通过完成这个作业与存储管理系统的设计，我不仅巩固了进程控制、进程通信和文件系统调用等相关知识，还深入理解了线程的概念和运用。同时，我也通过实践锻炼了系统设计和编程实现的能力。这次课程设计为我提供了宝贵的实践经验，使我更加熟悉和了解了操作系统的相关概念和技术，并提升了我在操作系统领域的综合能力。

**6．附录**

**服务端套接字初始化：**

SOCKET sockSer; //创建服务器套接字

SOCKET sockCli;

//SOCKADDR\_IN是系统封装的一个结构体，具体包含了成员变量：sin\_family、sin\_addr、sin\_zero

SOCKADDR\_IN addrSer;

SOCKADDR\_IN addrCli;

initialization();

//domain表示要使用的协议AF\_INET指ipv4,type指套接字类型 stream指tcp

//协议“protocol”一般设置为“0”，就是说在已经确定套接字使用的协议簇和类型时，这个参数的值就为0。

sockSer = socket(AF\_INET, SOCK\_STREAM, 0);

//初始化地址包

//sin\_addr主要用来保存IP地址信息

//htonl 函数可用于将主机字节顺序中的 IPv4 地址转换为网络字节顺序中的 IPv4 地址。

addrSer.sin\_addr.s\_addr = htonl(INADDR\_ANY);

//sin\_family主要用来定义是哪种地址族

addrSer.sin\_family = AF\_INET;

//sin\_port主要用来保存端口号

addrSer.sin\_port = htons(8000);

// 在建立套接字文件描述符成功后，需要对套接字进行地址和端口的绑定，才能进行数据的接收和发送操作。

if (bind(sockSer, (SOCKADDR\*)&addrSer, sizeof(SOCKADDR)) == SOCKET\_ERROR)

{

//调用 WSACleanup 时，将取消此过程中任何线程发出的挂起阻止或异步 Windows 套接字调用，

//而无需发布任何通知消息，也不发出任何事件对象的信号。

WSACleanup();

}

else {

//cout << "绑定套接字成功" << endl;

}

//设置监听

cout << "开始连接" << endl;

//等待连接最大 0x7fffffff 即32位最大值，成功返回0，失败失败返回SOCKET\_ERROR

if (listen(sockSer, SOMAXCONN) != 0) {

cout << "监听失败！";

WSACleanup();

}

//接受连接:

//accept将客户端的信息绑定到一个socket上，

//也就是给客户端创建一个socket，通过返回值返回给我们客户端的socket

int addrlen = sizeof(SOCKADDR);

sockCli = accept(sockSer, (SOCKADDR\*)&addrCli, &addrlen);

if (sockCli == SOCKET\_ERROR) {

cout << "接受连接失败！" << endl;

WSACleanup();

return 0;

}

cout << "连接建立，准备接受数据" << endl;

**系统状态监控：**  
void PrintSystemStatus() {

// Query and print task information

DWORD processIds[1024];

DWORD processCount;

if (EnumProcesses(processIds, sizeof(processIds), &processCount)) {

DWORD taskCount = processCount;

DWORD runningCount = 0;

DWORD sleepingCount = 0;

DWORD stoppedCount = 0;

DWORD zombieCount = 0;

for (DWORD i = 0; i < taskCount; i++) {

HANDLE hProcess = OpenProcess(PROCESS\_QUERY\_INFORMATION | PROCESS\_VM\_READ, FALSE, processIds[i]);

if (hProcess) {

DWORD exitCode;

if (GetExitCodeProcess(hProcess, &exitCode) && exitCode == STILL\_ACTIVE) {

runningCount++;

}

else {

if (exitCode == 0) {

stoppedCount++;

}

else {

zombieCount++;

}

}

CloseHandle(hProcess);

}

}

cout << "Tasks: " << taskCount << " total, " << runningCount << " running, "

<< sleepingCount << " sleeping, " << stoppedCount << " stopped, " << zombieCount << " zombie" << endl;

}

// Query and print CPU information

SYSTEM\_INFO systemInfo;

GetSystemInfo(&systemInfo);

DWORD processorCount = systemInfo.dwNumberOfProcessors;

FILETIME idleTime, kernelTime, userTime;

if (GetSystemTimes(&idleTime, &kernelTime, &userTime)) {

ULONGLONG idleTimePrev = (reinterpret\_cast<ULONGLONG\*>(&idleTime))[0];

ULONGLONG kernelTimePrev = (reinterpret\_cast<ULONGLONG\*>(&kernelTime))[0];

ULONGLONG userTimePrev = (reinterpret\_cast<ULONGLONG\*>(&userTime))[0];

Sleep(1000); // Wait for 1 second

if (GetSystemTimes(&idleTime, &kernelTime, &userTime)) {

ULONGLONG idleTimeNow = (reinterpret\_cast<ULONGLONG\*>(&idleTime))[0];

ULONGLONG kernelTimeNow = (reinterpret\_cast<ULONGLONG\*>(&kernelTime))[0];

ULONGLONG userTimeNow = (reinterpret\_cast<ULONGLONG\*>(&userTime))[0];

ULONGLONG idleTimeDiff = idleTimeNow - idleTimePrev;

ULONGLONG kernelTimeDiff = kernelTimeNow - kernelTimePrev;

ULONGLONG userTimeDiff = userTimeNow - userTimePrev;

float cpuUsage = 100.0f \* (kernelTimeDiff + userTimeDiff) / (kernelTimeDiff + userTimeDiff + idleTimeDiff \* processorCount);

cout << "%Cpu(s): " << cpuUsage << " us, " << kernelTimeDiff / processorCount << " sy, 0.0 ni, "

<< idleTimeDiff / processorCount << " id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st" << endl;

}

}

// Query and print memory information

MEMORYSTATUSEX memStatus;

memStatus.dwLength = sizeof(memStatus);

GlobalMemoryStatusEx(&memStatus);

cout << "KiB Mem: " << memStatus.ullTotalPhys / 1024 << " total, "

<< memStatus.ullAvailPhys / 1024 << " free, " << (memStatus.ullTotalPhys - memStatus.ullAvailPhys) / 1024

<< " used, " << memStatus.ullAvailPageFile / 1024 << " buff/cache" << endl;

cout << "KiB Swap: " << memStatus.ullTotalPageFile / 1024 << " total, "

<< memStatus.ullAvailPageFile / 1024 << " free, " << (memStatus.ullTotalPageFile - memStatus.ullAvailPageFile) / 1024

<< " used, " << memStatus.ullAvailPhys / 1024 << " avail Mem" << endl;

ULARGE\_INTEGER freeBytesAvailable, totalBytes, totalFreeBytes;

char rootPath[MAX\_PATH] = "E:\\";

if (GetDiskFreeSpaceExA(rootPath, &freeBytesAvailable, &totalBytes, &totalFreeBytes))

{

double usedPercent = (1 - (double)totalFreeBytes.QuadPart / totalBytes.QuadPart) \* 100;

cout << "Filesystem 1K-blocks: " << totalBytes.QuadPart / 1024 << "\t"

<< "Used: " << (totalBytes.QuadPart - totalFreeBytes.QuadPart) / 1024 << "\t"

<< "Available: " << totalFreeBytes.QuadPart / 1024 << "\t"

<< "Use%: " << usedPercent << "%\t"

<< "Mounted on: " << rootPath << endl;

}

else

{

cout << "Failed to get file system information." << endl;

}

cout << "VFS " <<D\_N << " " << disk\_size << " "

<< 1024 - sizeof(disk->content) << " " << (sizeof(disk->content) / D\_N) \* 100 << "%" << " /root" << endl;

int run\_num = 0;

int suc\_num = 0;

int can\_num = 0;

int sus\_num = 0;

int i = 0;

for (auto it = jobManager.jobs.begin(); it != jobManager.jobs.end(); )

{

if (jobManager.jobs[i].status == "running")

{

run\_num++;

}

if (jobManager.jobs[i].status == "success")

{

suc\_num++;

}

if (jobManager.jobs[i].status == "canceled")

{

can\_num++;

}

if (jobManager.jobs[i].status == "suspend")

{

sus\_num++;

}

++it;

++i;

}

cout << "Job:" << sizeof(jobManager.jobs) << " total, " <<

run\_num << " running, " << suc\_num << " success, " << can\_num << " canceled, " << sus\_num << " suspend" << endl;

}

**虚拟文件系统：**

//显示当前的路径目录

void show\_pos()

{

int num = number;

string paths;

while (num != -1)

{

if (dentry[num].type == 1 && dentry[num].name != "root")

{

cout << dentry[num].time + " " + dentry[num].isdir + " " +

"/" + dentry[num].name << endl;

}

if (dentry[num].type == 0)

{

cout << dentry[num].time + " " << dentry[num].inode.size << " " <<

dentry[num].name << endl;

}

paths = "/" + dentry[num].name + paths;

num = dentry[num].last;

}

paths = paths + " ";

// cout << "路径为：" ;

cout << paths << endl;

}

//创建一个目录

int make\_dir(string name\_file)

{

/\*cout << "请输入要在当前目录下要创建的文件名：" << endl;

string name\_file;

cin >> name\_file;\*/

int i = 0;

while (dentry[i].name != "")

{

i += 1;

}

SYSTEMTIME t;

GetLocalTime(&t);

string time = to\_string(t.wYear) + "/" + to\_string(t.wMonth) + "/"

+ to\_string(t.wDay) + " " + to\_string(t.wHour) + ":" + to\_string(t.wMinute);

size\_t delimiterPos = name\_file.find(".txt");

dentry[i].name = name\_file;

dentry[i].last = number;

dentry[i].time = time;

if (delimiterPos != string::npos)

{

dentry[i].isdir = "";

}

else

{

dentry[i].isdir = "<DIR>";;

}

dentry[i].num = i;

return dentry[i].num;

}

//将磁盘内的数据读到内存中

void read\_file()

{

ifstream ins("Test.txt");

if (!ins.is\_open())

{

cout << "打开文件失败！" << endl;

dentry[0].name = "root";

dentry[0].last = -1;

dentry[0].num = 0;

dentry[0].inode;

}

else

{

for (int j = 0; j < D\_N; j++)

{

if (j >= 256)//读取磁盘文件内容数据

{

string con;

getline(ins, con);

//cout << "con is: " << con << endl;

if (con != "")

{

disk[j].content = con;

//cout << disk[j].content << endl;

disk[j].state = 1;

}

//cout << "disk is : " << con << " " << j << endl;

}

if ((j >= 128) && (j < 256))//读取inode的详细数据

{

string con;

getline(ins, con);

//cout << "con is : " << con << " " << j << endl;

if (con.size() == 0)

{

}

else

{

Inode inodes;

istringstream pos(get\_first(con));

pos >> inodes.pos\_start;

istringstream size(get\_first(con));

size >> inodes.size;

istringstream dir\_num(get\_first(con));

dir\_num >> inodes.dir\_num;

//将inode与目录连在一起

for (int q = 0; q < N; q++)

{

if (dentry[q].num == inodes.dir\_num)

{

dentry[q].inode = inodes;

}

}

}

}

if (j < 128)//读取磁盘目录内容数据

{

string context;

getline(ins, context);

if ((context == " ") || (context == ""))

{

//cout << "全部内容为空" << endl;

}

else

{

int nums = 0;

istringstream last(get\_first(context));

istringstream num(get\_first(context));

num >> nums;

istringstream type(get\_first(context));

last >> dentry[nums].last;

//cout << dentry[nums].last << " ";

dentry[nums].num = nums;

//cout << dentry[nums].num << " ";

dentry[nums].name = get\_first(context);

//cout << dentry[nums].name << " ";

type >> dentry[nums].type;

//cout << dentry[nums].type << " ";

//cout<< j << endl;

}

dentry[0].name = "root";

}

}

}

ins.close();

}

void upload\_file(string name\_file)

{

ifstream file(name\_file);

if (!file.is\_open()) {

cout << "无法打开文件：" << name\_file << std::endl;

}

//在当前目录创建同名文件

int i = 0;

while (dentry[i].name != "")

{

i += 1;

}

SYSTEMTIME t;

GetLocalTime(&t);

string time = to\_string(t.wYear) + "/" + to\_string(t.wMonth) + "/"

+ to\_string(t.wDay) + " " + to\_string(t.wHour) + ":" + to\_string(t.wMinute);

size\_t delimiterPos = name\_file.find(".txt");

dentry[i].time = time;

dentry[i].name = name\_file;

dentry[i].last = number;

dentry[i].num = i;

dentry[dentry[i].num].type = 0;

string content((istreambuf\_iterator<char>(file)), istreambuf\_iterator<char>());

dentry[dentry[i].num].inode.content = content;

dentry[dentry[i].num].inode.size = content.size();

dentry[dentry[i].num].inode.dir\_num = dentry[dentry[i].num].num;

dentry[dentry[i].num].inode.pos\_start = find\_pos(dentry[dentry[i].num].inode.size);

if (dentry[dentry[i].num].inode.pos\_start != -1)

{

int size = dentry[dentry[i].num].inode.size;

int n = size / disk\_size;

if ((size % disk\_size) != 0)

{

n = n + 1;

}

for (int i2 = dentry[dentry[i].num].inode.pos\_start; i2 < dentry[dentry[i].num].inode.pos\_start + n; i2++)

{

disk\_write(content, i2);

}

}

else

{

//没有储存空间的情况

}

// 关闭文件

file.close();

}

**作业系统：**

void createJob(int jobID, const string& program) {

int PID = 520 + jobID;

Job job(jobID, PID, program);

jobs.push\_back(job);

jobIDtoIndex[jobID] = jobs.size() - 1;

cout << "create job and run... ok" << endl;

cout << "JobID:" << jobID << " PID:" << PID << endl;

}

void run(int jobID)

{

int t = jobIDtoIndex[jobID];

if (jobs[t].status == "created" || jobs[t].status == "running")

{

// 模拟作业执行过程的线程

thread jobThread([this, jobID]() {

// 模拟作业执行过程

int index = jobIDtoIndex[jobID];

jobs[index].status = "running";

for (int i = 0; i <= 100 - jobs[index].now\_pro; i += 2) {

jobs[index].progress = i + jobs[index].now\_pro;

this\_thread::sleep\_for(chrono::seconds(1));

}

if (jobs[index].progress == 100 && jobs[index].status == "running")

{

jobs[index].status = "success";

}

});

jobThread.detach(); // 分离线程，使其在后台运行

}

else

{

terminate();

}

}

void displayJobStatus() {

cout << "job status in jdm server:" << endl;

cout << "JobID PID status program" << endl;

for (const auto& job : jobs) {

if (job.status == "running"|| job.status == "success")

{

cout << job.jobID << " " << job.PID << " " << job.status << " " << job.progress << "% " << job.program << endl;

}

else

{

cout << job.jobID << " " << job.PID << " " << job.status << " " << job.now\_pro << "% " << job.program << endl;

}

}

}

void suspendJob(int jobID) {

int index = jobIDtoIndex[jobID];

jobs[index].status = "suspend";

jobs[index].now\_pro = jobs[index].progress;

jobs[index].progress = 0;

cout << "suspend job " << jobID << " ... ok" << endl;

}

void clearJobs() {

for (auto it = jobs.begin(); it != jobs.end(); ) {

if (it->status == "success" || it->status == "canceled") {

it = jobs.erase(it);

}

else {

++it;

}

}

cout << "clear finished and canceled jobs ... ok" << endl;

}