

**操作系统课程设计报告**

**2023年3月**

**贵州大学计算机科学与技术学院**

**《操作系统课程设计》**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 组别： 3 组长：谢航 | | | | | | | | | | | |
| 学号 |  | 姓名 | 谢航 | 分工 | | 进程管理 | | 工作量比例 | 20% | 成绩 |  |
| 学号 |  | 姓名 | 彭启槟 | 分工 | | 设备管理 | | 工作量比例 | 20% | 成绩 |  |
| 学号 |  | 姓名 | 周军 | 分工 | | 内存管理 | | 工作量比例 | 20% | 成绩 |  |
| 学号 |  | 姓名 | 杨金平 | 分工 | | 文件管理 | | 工作量比例 | 20% | 成绩 |  |
| 学号 |  | 姓名 | 李东金 | 分工 | | 页面设计，磁盘调度 | | 工作量比例 | 20% | 成绩 |  |
| 实 验 题 目 | 操作系统模拟 | | | | | | | | | | |
| 自 我 评 价 | **谢航**：  我负责的是进程调度管理部分，主要是实现的功能有：1.进程调度的三种常见方式：先来先服务，短作业优先和动态优先级时间片轮转法。 2.进程模块和内存管理模块的通信，传递需要访问的页面号 3.进程和文件管理部分的联系，打开文件需要创建进程并且生成磁盘调度所需要的磁道顺序 4.用信号量操作模拟进程同步，主要涉及到实现PV操作。  针对本次课设，让我深刻的领会到CPU作为计算机大脑的重要性，与其他板块的工作有着千丝万缕的紧密联系。在写代码前，进程调度就应该提前规划好设计思路，提前预测可能出现的问题。因为没有提前规划好设备管理部分的联系，导致我们小组后期合并难度较大，所以放弃了与该部分的联系。  同时，除了实现算法和功能，为了界面的美观，我们选择了用QT作为代码平台。课设的中途为了界面也付出了较多时间。在学习QT编程的时候，各种组件的利用和功能代码的实现，使我更加深刻的理解了模块化编程和面向对象的好处，不仅利于团队之间相互合作，互相分工，同时也利于自己代码的重复利用和功能之间的封装，保障代码的简洁和安全性。  **周军**：  我主要负责内存管理模块，主要是为进程提供申请内存的接口函数，采用固定分配局部置换策略，在创建进程之前先调用我的分配函数并传递进程所需页框数，我这边会根据内存的使用情况进行页框的分配。提供一个页面请求函数。在进程创建时预先讲程序加载到虚拟内存，进程运行的时候需要请求页面，如果内存中没有就需要在虚拟内训中查找后进行页面置换。这里我提供了LRU和FIFO页面置换算法，考虑到OPT算法是理想算法，需要后面的请求页面对列，然而实际的操作系统需要请求的页面具有随机性，所以在这里没有设计OPT算法。最后进程运行结束后释放内存，释放呢进程在虚拟内存中的程序。  在设计过程中由于我们并没有商量好实现方式所以出现了一些问题。导致用大量时间完成的工作并没有用，让我深刻体会到合作作业交流的重要性，同时，我也在这次合作作业中收获很多，不仅在技术知识上有所收获，还让我学会如何协同工作。    **彭启槟**：  我负责的是设备管理模块，该模块没有与其他成员的部分连接，故PCB各项数据均由随机函数生成。该模块模拟了4个资源，分别是蓝牙，打印机，键盘，鼠标。通过随机函数生成进程队列，以及请求各个资源的数量，得到矩阵。通过输入发送进程对资源和的请求，再通过银行家算法，安全性检测算法去计算该请求是否不会导致系统处于不安全状态。若请求予以通过，则根据该进程申请的时间占用资源，时间到后结束归还请求与原本占用的资源，完成本次请求。  在该模块的设计过程中，我学会了qt的一些基本的开发操作，对死锁的避免有了相比于一开始更进一步的了解。通过这次的课程设计，我明白了作为一名计算机专业的学生，必须时刻处于学习的状态，因为科技一直在进步，技术一直在发展，所学的某项技术可能不久后便会被时代所淘汰！  **李东金**：  我负责的是界面管理模块，我们组使用QT平台来作为基础，它跨平台，支持很多平台；接口也简单容易上手，开发的效率也很高。我们的界面总共有四个板块组成:进程管理，内存管理，文件管理（磁盘管理），设备管理。我们可以进行文件夹的创建以及文件的创建与删除，可以从界面了解磁盘的使用状态，置换算法LRU，FIFO每个进程使用内存的页框数，调度算法FCFS,SJF,DPTSR所对应进程的运行信息，还可以执行银行家等算法。  在此次课设中由于初次接触QT，对很多操作使用还不是特别熟练，因此做了很多功课，收获也很大，对于QT有了一些基本的了解和使用。通过它，我们的团队合作能力以及个人能力得到了一定的提升，让我明白了1+1>2这个道理。  **杨金平**：  我负责的是文件系统模块，该模块涉及文件的内部存储结构模块如目录结构，文件控制块 FCB 结构以及文件数据存储结构等，视图结构上最大简单化了用户的操作接口，使用 QT 控件接口就能操作文件的创建、删除、修改等操作。对于存储结构模拟上，针对不同的文件磁盘控件分配方式有不同的模拟结构，对于文件空间分配方案有：  连续分配修正方案，文件目录存储文件的基本信息以及起始文件块的块地址，对于每个文件块又有扩展标记用于文件内容扩展时往下链接下一个连续块。  显式链接分配方案在磁盘里存储一个 FAT 表，里面记录每个磁盘块的下一扩展块地址，并以指定结束值作为文件的结束标志。因此对于每个文件目录只需要存储起始磁盘块地址就能在 FAT 表中依次找到下一个文件磁盘块地址。  索引分配方案将每个索引块大小设置为磁盘块大小，里面包含文件基本信息、以及指定数目的文件磁盘块、下一索引块所在磁盘块地址。因此同理文件目录只需要存储起始索引块的磁盘块地址就能依次遍历文件磁盘块。  当然还有每个文件 FCB 的初始化及更新等其他操作，但每次分配空间前从空闲空间表中判断是否足够才分配。  虽然在设计过程中总是遇到一些 bug ,涉及到算法问题以及 QT 语法问题,但通过文件系统的模拟，对操作系统中文件的存储结构、分配方式管理等结构更加深刻，能够体会得到操作系统对文件工作的高度维护服务，也意识到操作系统的重要性。 | | | | | | | | | | |
|  | 评价指标：   * 题目内容和要求完成情况 优 □ 良 □ 中 □ 差 □ * 对算法原理的理解程度 优 □ 良 □ 中 □ 差 □ * 程序设计水平 优 □ 良 □ 中 □ 差 □ * 程序运行效果及正确性 优 □ 良 □ 中 □ 差 □ * 课程设计报告结构清晰 优 □ 良 □ 中 □ 差 □   报告中总结和分析详尽 优 □ 良 □ 中 □ 差 □ | | | | | | | | | | |
| 成绩 |  | | | | 指导教师 | |  | | | | |
| 一、 **需求分析**  1、 **课程设计目的**  加强学生对操作系统的理解，让我们能够将书本上的知识学以致用，同时也能强化我们的团队合作能力和实践操作。  2、 **编程语言与编程工具**  **编程语言：**C++  **编程工具：**QT  3、 **设计内容**  模拟一个采用多道程序设计方法的单用户操作系统， 该操作系统应包括进程管理、 存储管理、设备管理、文件管理和用户接口五个部分。 | | | | | | | | | | | |
| 二、 **概要设计**  1、 **模块划分：**  （1）、进程管理模块  （2）、内存管理模块  （3）、文件管理模块  （4）、设备管理模块  （5）、界面管理模块  2、 **程序流程图：**  **进程管理**  descript  **内存管理**    **释放内存 申请内存 页面请求**  descript  **文件管理**  descript  **设备管理**  descript    3、 **程序UML类图：**  **进程管理**  descript  descript  descript    **内存管理**  **descript**  **文件管理**  **设备管理**  descript | | | | | | | | | | | |
| 三、 **详细设计**  **1. 进程管理：考虑进程的创建、状态转换（就绪、执行、阻塞）、调度、撤销等，设计PCB结构，设计进程调度算法。**  **选做：使用信号量机制模拟多任务系统中的进程同步。**  进程管理部分涉及三个类，其中：PCB类为进程控制块，ProcessDialog类为创建进程时弹出的对话框， processtab为进程管理的主要逻辑代码处理类。 1.1进程的创建 随机创建三个默认的进程，以及两个进程**“获取键值”**和**“键盘输入”**用于模拟进程同步。 手动创建进程时，需要如图输入进程标识符（string类型），进程优先级和所需执行的CPU时间。  descript  每次创建进程，会生成一个PCB对象，并将加入进程控制块PCB的对象加入就绪队列中，PCB的属性如下：  descript  同时，当打开文件时也会创建file进程用于模拟进程生成磁盘调度访问磁道号的序列，效果如下：  descript 1.2 状态转换 进程创建后放入就绪队列readyQueue，变为**就绪态**；当选择不同的调度算法时，会通过不同的调度策略从就绪队列中选择一个进程调度到运行队列runningQueue，转换为**运行态**；运行态的进程需要的资源没有分配时，进程会转换为**阻塞态**；并且等待分配资源后再次转换为**就绪态**。进程执行完后，会转换为**完成状态**。  descript 1.3 使用信号量机制模拟多任务系统中的进程同步 通过上图可以观察到，先执行进程“获取键值”，该进程进入阻塞队列;只有进程“键盘输入”执行完成“获取键值”才能从阻塞队列转到就绪队列中。  模拟过程为：  信号量：  *semaphore\_full：* 代表“键盘输入”运行的生产的数据信号量，初始值为0  *semaphore\_keyboard:* 代表键盘输入的临界资源，初值为1  mutex = 1; 代表临界区互斥访问信号量  进程“键盘输入”必须完成后，进程“获取键值”才能运行，如果“获取键值”进程提前进入运行队列，P操作后更改信号量*semaphore\_full*，将会变为阻塞态放入阻塞队列中等待“键盘输入”执行完后V操作释放信号量*semaphore\_full*  进程“获取键值”执行完后会V操作释放*semaphore\_keyboard*，键盘“键盘输入”P操作申请*semaphore\_keyboard*信号量。  模拟PV操作的思路为：通过对进程PCB中“behavior”属性赋值，模拟进程的行为  核心代码如下：  **PV操作代码**   1. **bool** ProcessTab::P(**int** &semaphore){ 2. semaphore--; //qDebug()<<"P--full:"<<semaphore; 3. **if**(semaphore<0){ 4. PCB\* tempProcess = runningQueue[0]; 5. runningQueue.clear(); 6. strcpy(tempProcess->state, "阻塞"); 7. blockQueue.push\_back(tempProcess); 8. **return** **true**; 9. } 10. **return** **false**; 11. } 12. **bool** ProcessTab::V(**int** &semaphore){ 13. semaphore++; 14. //唤醒一个进程 15. **if**(semaphore<=0){ 16. semaphore++; 17. PCB\* tempProcess = blockQueue[0]; 18. blockQueue.erase(blockQueue.begin()); 19. strcpy(tempProcess->state, "就绪"); 20. readyQueue.push\_back(tempProcess); 21. **return** **true**; 22. } 23. **return** **false**; 24. }   **模拟进程同步的行为**   1. **bool** ProcessTab::Process\_Behaviour(string behaviour,PCB\* runOne){ 2. **bool** flag; 3. **if**(behaviour == "键盘输入"){ 4. flag = P(**this**->semaphore\_keyboard); 5. P(**this**->mutex); 6. **return** flag; 7. } 8. **else** **if**(behaviour == "获取键值"){ 9. flag = P(semaphore\_full); 11. **if**(flag == **true**){ 12. **return** flag; 13. } 14. P(**this**->mutex); 15. **return** flag; 16. } 17. }   因为考虑到PV操作需要结合进程调度代码，所以模拟行为中进程的V操作在进程调度部分，如下：   1. **if**(runOne->behaviour == "键盘输入"){ 2. V(mutex); 3. flag = V(semaphore\_full); 4. }**else** **if**(runOne->behaviour == "获取键值"){ 5. V(**this**->mutex); 6. V(semaphore\_keyboard); 7. }  1.4 进程调度1.4.1 FCFS先来先服务算法 进程按照到达内存的时间依次调度进入处理机执行  部分代码（有删减）如下：   1. **while**(!readyQueue.empty() || !blockQueue.empty() || !runningQueue.empty()){ 2. PCB \*runOne = readyQueue[0]; //取出就绪队列队首进程 3. readyQueue.erase(readyQueue.begin()); 4. runningQueue.push\_back(runOne); //放入运行队列中 6. **while**(runOne->needTime>0){ 7. runOne->needTime--; 8. runOne->cpuTime++; 9. } 10. //进程执行完后更新 11. runningQueue.pop\_back(); 12. }  1.4.2 SJF短时间优先算法 SJF与FCFS代码大同小异，通过对就绪队列中进程的运行时间从小到大排序然后选择时间最短的进程调入处理机运行  部分代码（有删减）如下：   1. **while**(!readyQueue.empty() || !blockQueue.empty() || !runningQueue.empty()){ 2. //就绪队列排序 3. sort(readyQueue.begin(),readyQueue.end(),ProcessTab::cmp\_needtime); 4. PCB \*runOne = readyQueue[0]; //取出就绪队列队首进程 5. readyQueue.erase(readyQueue.begin()); 6. runningQueue.push\_back(runOne); //放入运行队列中 7. **while**(runOne->needTime>0){ 8. runOne->needTime--; 9. runOne-cpuTime++; 10. } 11. //进程执行完后更新 12. runningQueue.pop\_back(); 13. }  1.4.3 DPTSR动态优先级的时间片轮转法 创建进程时固定时间片大小为2，然后分配相应的优先级（数值越大，优先级越高），模拟时运行时间越小的进程优先级越高。每个时间片后，从运行队列转入到就绪队列的进程优先级-1，同时，在就绪队列中的进程会全部相应+1，达到动态优先级的效果。  部分核心逻辑代码如下：   1. **void** ProcessTab::Dynamic\_Priority\_Time\_Slice\_Rotation(){ 2. h: **while**(!readyQueue.empty() || !blockQueue.empty() || !runningQueue.empty()){ 3. sort(readyQueue.begin(),readyQueue.end(),ProcessTab::compare); //就绪队列排序 4. PCB \*runOne = readyQueue[0]; 5. readyQueue.erase(readyQueue.begin()); 6. strcpy(runOne->state, "运行"); 7. runningQueue.push\_back(runOne); 9. //如果进程的所需执行时间<时间片，则进程执行完毕 10. **if**(runOne->needTime<=runOne->round){ 11. runOne->cpuTime += runOne->needTime; 12. runOne->needTime=0； 13. runningQueue.pop\_back(); 14. } 15. //如果进程没有执行完 16. **else**{ 17. runOne->needTime -= runOne->round; 18. runOne->cpuTime += runOne->round; 20. **for**(unsigned **int** i=0;i< readyQueue.size();i++){ 21. readyQueue[i]->prio+=1; //就绪队列中所有进程优先级+1 22. } 23. **if**(runOne->prio!=1){ 24. runOne->prio = runOne->prio-1; //运行队列中进程优先级-1 25. } 27. //将时间片轮转的进程重新放入就绪队列中 28. strcpy(runOne->state, "就绪"); 29. runningQueue.clear(); 30. readyQueue.push\_back(runOne); 31. } 32. } 33. }  1.5 进程撤销 模拟进程撤销为通过图形界面直接删除进程极其数据，包括所有队列中对应的进程控制块PCB的对象，释放进程在内存中占用的物理块，以及更新界面   1. //删除表格按钮 2. **void** ProcessTab::on\_deleteprocess\_clicked(){ 3. **int** currentRow = ui->processtable->currentRow(); 4. **for**(**int** i=0;i<processQueue.size();i++){ 5. **if**(QString::fromStdString(processQueue[i]->name) == ui->processtable->item(currentRow,0)->text()){ 6. processQueue.erase(processQueue.begin()+i); 7. } 8. } 9. **if**(!readyQueue.empty()){ 10. **for**(unsigned **int** i=0;i<readyQueue.size();i++){ 11. **if**(QString::fromStdString(readyQueue[i]->name) == ui->processtable->item(currentRow,0)->text()){ 12. readyQueue.erase(readyQueue.begin()+i); 13. } 14. } 15. }**else**{ 16. ui->processtable->removeRow(currentRow); 17. **return** ; 18. } 19. //释放内存物理块 20. w->memoryTab->freeMemery(ui->processtable->item(currentRow,0)->text()); 21. ui->processtable->removeRow(currentRow); 22. }  1.6 和其他模块的联系1.6.1 与内存模块 进程每次调入处理机运行，会每秒传递一个需要访问的页面给内存模块（便于观察）。需要访问的页面在进程创建时随机生成，运行中一次请求  descript  如上图所示，运行进程pro0，首先第一个时间片访问页面2，后面依次逐个访问 1.6.2 与文件系统和磁盘模块 打开文件时会创建相应的进程“file”，CPU通过文件所在的磁盘块所在磁道序号，访问磁盘进行磁盘调度，读取文件的信息。  **2. 内存管理：考虑内存的动态分配与回收，采取何种数据结构，采用何内存分配算法等。**  **选做：内存的离散分配（分页式、分段式或段页式管理），以及基于此种机制下的虚拟内存管理，页面置换算法。** 2.1 内存管理初始设计 采用分页式内存离散分配。虚拟内存管理策略：请求（按需）调页策略。操作系统采用固定分配局部置换策略为进进程分配页框。为进程提供内存页框分配函数，页面请求函数，内存释放函数。  内存管理辅助结构体：   1. **struct** usedMemeryBlock{ 2. **int** memeryBlockSize; 3. QString pid; 4. **int** requestPageList[50]; 5. **int** blockSize = 0; 6. **int** requestPageCount = 0; 7. **int** ranking; 8. **int** pageList[5][2]; 9. QQueue<**int**> FIFOPageQueue;//FIFO算法辅助队列 10. usedMemeryBlock \*nextBlock;   };  使用链表存储每个进程的内存分配情况。  内存初始化状态，初始为400个页框：  descript  虚拟内存初始化状态：   2.2 内存请求 在进程创建时需要先调用我的内存请求函数，根据进程的id和它需要的也框数进行内存的页框分配。  关键代码：   1. **bool** Memory::requestMemery(**int** pageFrame,QString pid){ 2. **int** count = 0; 3. **struct** usedMemeryBlock \* block = **new** **struct** usedMemeryBlock(); 4. block->memeryBlockSize = pageFrame; 5. block->pid = pid; 6. block->nextBlock = **this**->usedMemeryList; 7. block->ranking = **this**->proCount; 8. **this**->usedMemeryList = block; 9. **int** i; 10. **for**(i=0;i<50;i++){ 11. block->requestPageList[i] = -1; 12. } 13. **for**(i=0;i<pageFrame;i++){ 14. block->pageList[i][0] = -1; 15. } 16. **for**(**int** i=0;i<400;i++){ 17. **if**(**this**->isUsed[i]==0){ 18. block->pageList[count][1] = i; 19. **this**->isUsed[i] = 1; 20. count++; 21. **if**(count==pageFrame) **break**; 22. } 23. } 24. **if**(i==400) **return** **false**; 25. **this**->dye(block,0);//对已分配的内存进行染色 26. **this**->proCount++; 27. w->diskTab->createVirtualMemoryBlock(pid,block->ranking); 28. **return** **true**; 29. }   内存分配后：  descript  虚拟内存：  descript 2.3 页面请求请求 在进程运行时需要请求页面，为此提供了页面请求函数，提供了最久未使用算法(LRU）和先来先服务算法（FIFO），LRU算法用requestPageList数组进行统计请求的页面作为辅助数据结构，FIFO算法用FIFOPageQueue队列作为辅助结构  关键代码：   1. //先进先出置换算法 2. **int** OutPage = block->FIFOPageQueue.dequeue(); 3. block->FIFOPageQueue.enqueue(page); 4. //取出需要置换的位置 5. **int** index; 6. **for**(index=0;index<block->memeryBlockSize;index++){ 7. **if**(block->pageList[index][0] == OutPage){ 8. **break**; 9. } 10. } 12. //页面置换算法 最久未使用算法 13. //寻找替换的位置 14. **int** k; 15. **int** postion[block->memeryBlockSize]; 16. **int** count = 0; 17. **for**(k = 0;k<block->memeryBlockSize;k++){ 18. postion[k] = -1; 19. } 20. //对前面的页进行扫描 21. **for**(k = block->requestPageCount-1;k>=0;k--){ 22. **int** z; 23. **for**(z=0;z<block->memeryBlockSize;z++){ 24. **if**(block->requestPageList[k] == block->pageList[z][0]){ 25. **if**(postion[z]!=0) 26. { 27. postion[z] = 0; 28. count++; 29. } 30. } 31. } 32. **if**(count == block->memeryBlockSize-1){ 33. **break**; 34. } 35. } 36. //取出需要置换的位置 37. **int** index; 38. **for**(index=0;index<block->memeryBlockSize;index++){ 39. **if**(postion[index]==-1){ 40. **break**; 41. } 42. }  2.4 内存释放 当进程运行完时需要释放该进程的内存以及虚拟内存  关键代码：   1. //释放内存 2. **void** Memory::freeMemery(QString pid){ 3. **struct** usedMemeryBlock \* block; 4. **struct** usedMemeryBlock \* block1; 5. block = **this**->usedMemeryList; 6. w->diskTab->freeMemory(pid); 7. **if**(block->pid == pid){ 8. **this**->usedMemeryList = block->nextBlock; 9. **for**(**int** i =0;i<block->memeryBlockSize;i++){ 10. **this**->isUsed[block->pageList[i][1]] = 0; 11. } 12. **this**->dye(block,1); 13. free(block); 15. } 16. **else**{ 17. **while**(block->nextBlock!=nullptr && block->nextBlock->pid != pid){ 18. qDebug()<<block->nextBlock->pid<<"44444"; 19. block = block->nextBlock; 20. } 21. **if**(block->nextBlock==nullptr){ 22. qDebug()<<"pid找不到"<<pid; 23. } 24. **else**{ 26. **for**(**int** i =0;i<block->nextBlock->memeryBlockSize;i++){ 27. **this**->isUsed[block->nextBlock->pageList[i][1]] = 0; 28. } 29. **this**->dye(block->nextBlock,1); 30. block1 = block->nextBlock; 31. block->nextBlock = block->nextBlock->nextBlock; 32. free(block1); 33. } 34. } 35. }   3. **文件管理：考虑文件及文件目录的创建、查询、删除等操作的实现，考虑磁盘空间分配采用的数据结构及算法（连续分配、显式链接、索引分配），考虑磁盘空闲区管理采用的数据结构及算法（空闲盘块表、空闲盘块链、位示图、成组链接法）和算法。**  **选做：磁盘调度算法。** 3.1 连续文件结构 **文件目录**  descript  **文件块**  **descript** 3.2 显式链接文件结构 **文件目录**  **descript**  **FAT 表**  **descript** 3.3 **索引分配**文件结构 **文件目录**  **descript**  **索引块结构**  **descript** 3.4 文件的创建 如图所示，选中当前所在目录后，点击"创建文件"弹出需要的初始信息，包括文件名，文件磁盘空间分配方式(便于模拟不同的分配方式)，随后为文件分配目录磁盘空间和初始化文件相关的数据结构 ：  descript  每次创建文件时先创建文件控制块 FCB ,存储文件大小、位置等基本描述属性信息  **void** myFilepro::setData(**int** i,QString ty,QString n, QString p, QString s, QString t,QString po)   1. { 2. **this**->id=i; 3. **this**->type=ty; 4. **this**->name=n; 5. **this**->pos=p; 6. **this**->size=s; 7. **this**->time=t; 8. **this**->policy=po; 9. }     对于文件夹的创建也同理，效果如下：  descript    随后根据文件空间分配方式初始化文件目录信息，  连续文件目录  descript  显式链接文件目录  descript  索引分配文件目录  descript 3.5 计数法空闲空间管理 **算法实现设计思路：**  根据磁盘块状态(被分配的状态)，顺序扫描每一个块，统计其中的每个连续块，记录下该连续块的起始块地址以及连续块数量，添加一个连续块记录到空闲空间表中，每个记录含有起始块地址(块号)、连续块数两个基本数据项。  每分配前查看该表是否有记录，无记录则说明空闲内存不足，该次分配失败。一次磁盘块则重新刷新一次该表。  **关键代码：**   1. **void** Disk::flushFreeBlockTab() 2. { 3. //初始化空闲空间列表 初始状态为一个条目，紧跟数量为总块数 4. **int** conBlockNum=0;//连续空闲块数量 5. **for**(**int** i=0;i<diskBlock.size();i++){ 6. **int** start=-1;//每次连续块开头 7. **int** end=-1;//每次连续块结尾 8. **int** conNum=0;//每次连续块数 9. **if**((diskBlock[i])->fileId==-1){//找到第一个空闲块 10. start=i; 11. **int** con=0; 12. **for**(**int** j=i;j<diskBlock.size();j++){//接着查询连续块是否足够 13. **if**(j==diskBlock.size()-1){ 14. //连续块结束 15. end=j; 16. con=con+1; 17. **break**; 18. } 19. **if**((diskBlock[j]->fileId!=-1)||(j==diskBlock.size()-1)){ 20. //连续块结束 21. end=j-1; 22. **break**; 23. }**else**{ 24. con=con+1;//连续块数 +1 25. } 26. } 27. conNum=con; 29. FreeBlockTable fr; 30. fr.blockId=start; 31. fr.con\_num=conNum; 32. freeblockTab.append(fr); 34. i=end+1; 35. **if**(end+1==diskBlock.size()-1){ 36. **break**; 37. } 38. conBlockNum++; 39. } 40. } 41. }  3.6 文件分配算法 **3.5.1 连续分配**  **实现思路：**  根据连续文件分配要求，每次分配时从空闲空间表里面查找是否有足够的连续块可以分配，若有则将该记录的相应值修改，刷新空闲表，否则无法分配。  **关键代码：**   1. //文件块表内找到最后一个块记录 (下一扩展块值 -2) 2. **int** endIndex=0; 3. **while** (confBlockId!=-2) { 4. **for** (**int** i=0;i<fileBlocks.size();i++) { 5. **if**(fileBlocks[i].firstBlockId==confBlockId){ 6. endIndex=i; 7. **break**; 8. } 9. } 10. confBlockId=fileBlocks[endIndex].nextBlockId; 11. } 12. confBlockId=fileBlocks[endIndex].nextBlockId; 13. //上一个文件块结尾修改记录 14. fileBlocks[endIndex].nextBlockId=conMinPos; 15. //文件块表添加记录 16. FileBlock fb; 17. fb.firstBlockId=conMinPos; 18. fb.blockNum=needNum; 19. fb.nextBlockId=-2;// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* -2 表示文件结束 20. fileBlocks.append(fb); 21. **if**(hasConBlock==**true**){ 22. //开始分配 23. **for**(**int** i=conMinPos;i<conMinPos+needNum;i++){ 24. //设置磁盘块的文件 id 25. diskBlock[i]->fileId=id; 26. } 27. **if**(minConNum==needNum){ 28. freeblockTab.erase(freeblockTab.begin()+minConindex); 29. ftb->removeRow(minConindex); 30. }**else**{ 31. freeblockTab[minConindex].blockId=conMinPos+needNum; 32. freeblockTab[minConindex].con\_num=minConNum-needNum； 33. } 34. }**else**{ 35. //最小连续块不够 36. }   **3.5.2 显式链接分配**  **算法设计思路：**  FAT 表记录每个磁盘块记录，记录里第一项表示块地址，第二项表示对应文件的下一块(未分配时值为 -1),为 -2 时表示对应文件结束块。分配时根据文件目录找到起始块地址，随后往下遍历到最后一 块，扩展时接着该块往下分配(文件结束值 -2 变为下一块)即可。  **关键代码：**   1. //目录中找到首块号 2. **int** lafirstId; 3. //FAT 表中寻找最后一块所在行 4. **int** lastRowIndex=lafirstId; 5. **int** log=lastRowIndex; 6. **while** (lastRowIndex!=-2) { 7. lastRowIndex=Fat[lastRowIndex].value; 8. **if**(lastRowIndex!=-2){ 9. log=lastRowIndex; 10. } 11. } 13. **int** firstId=-1; 14. **int** hasgetSpace=0; 15. **int** nextId=-2; 17. **bool** isFirst=**true**;//当前寻找是否第一空闲块 18. **int** ffirst;//第一个块号 19. **while** (hasgetSpace<size) { 20. //寻找第一个空闲块 21. **for** (**int** i=0;i<Fat.size();i++){ 22. **if**(Fat[i].value==-1){ 23. nextId=Fat[i].blockId;//找到第一空闲块 i 24. **if**(isFirst){ 25. ffirst=i; 26. //将上一个文件结尾链接 27. Fat[log].value=ffirst; 28. isFirst=**false**; 29. } 30. **break**; 31. } 32. } 33. //在空闲表中移除块 nextId; 34. **this**->disrtSpaceInFreeTable(nextId); 35. //修改磁盘块文件 id 36. diskBlock[nextId]->fileId=id; 37. //把下一个空闲块加入 38. **for** (**int** i=0;i<Fat.size();i++){ 39. **if**(Fat[i].blockId==firstId){ 40. Fat[i].value=nextId; 41. **break**; 42. } 43. } 44. //文件尾部空闲块 FAT v 置 -2 45. **for** (**int** i=0;i<Fat.size();i++){ 46. **if**(Fat[i].blockId==nextId){ 47. Fat[i].value=-2; 48. **break**; 49. } 50. } 51. hasgetSpace=hasgetSpace+10; 52. firstId=nextId; 53. nextId=-2; 54. }   **3.5.3 索引分配**  **算法设计思路：**  以一个索引块大小等于一个磁盘块，每个索引块里包含文件信息，以及特定数目的磁盘块地址，最后一个表示下一个文件索引块所在块，依次链接。分配时根据文件目录找到文件起始索引块位置，然后依次遍历到最后一个索引块，将其下一个索引块指针从 -2 (结束值)变为下一索引块地址依次扩展。  **关键代码：**   1. //寻找文件目录 2. **int** indexBlockId;//索引块所在块号 3. //找到上一次分配的最后一个索引块 4. **int** log=-1; 5. **while** (indexBlockId!=-1) { 6. //索引块不结束 7. **if**(indexBlockId!=-1){ 8. log=indexBlockId; 9. } 10. indexBlockId=indexFiles[indexBlockId].nextIndexBlockId; 11. } 13. **int** s=size/(blockNum\*blockSize);//商 14. **int** y=size%(blockNum\*blockSize);//余 15. **int** needIndexBlockNum;//所需要索引块数 16. **if**(y>0){ 17. needIndexBlockNum=s+1; 18. }**else**{ 19. needIndexBlockNum=s; 20. } 21. **int** allnum=(ui->block\_tab->rowCount())\*(ui->block\_tab->columnCount()); 22. //初始化虚拟磁盘 23. **for**(**int** i=0;i<allnum;i++){ 24. IndexBlock ind; 25. ind.diskblockId=i; 26. ind.fileId=-1; 27. ind.filename=nullptr; 28. ind.blocksID=**new** **int**[blockNum]; 29. **for**(**int** j=0;j<blockNum;j++){ 30. ind.blocksID[j]=-1; 31. } 32. ind.nextIndexBlockId=-1; 33. indexFiles.append(ind); 34. } 35. //修改文件大小 36. //新建文件目录 37. //找到第一个空闲块存储起始索引块 38. **for**(**int** i=0;i<diskBlock.size();i++){ 39. **if**((diskBlock[i])->fileId==-1){ 40. //将上一次最后一个索引块的末尾链接 41. indexFiles[log].nextIndexBlockId=i; 42. //为起始索引块分配磁盘 43. diskBlock[i]->fileId=id; 44. //修改空闲表\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 45. **this**->disrtSpaceInFreeTable(i); 46. **break**; 47. } 48. } 49. **int** hasgetSpace=0;//已分配内存 50. //创建起始索引块 51. (indexFiles[ifd.indexBlockId]).fileId=id; 52. (indexFiles[ifd.indexBlockId]).filename=name; 53. **if**(needIndexBlockNum<=1){ 54. //小文件：只需要一个索引块 55. **int** ss=size/blockSize; 56. **int** yy=size%blockSize; 57. **int** needCNum;//需要磁盘块数 58. **if**(yy>0){ 59. needCNum=ss+1; 60. }**else**{ 61. needCNum=ss; 62. } 63. **for**(**int** j=0;j<needCNum;j++){ 64. //对起始索引块中寻找每个磁盘块 65. **bool** cipanHasget=**false**;//改索引块中改磁盘是否分配完成 66. **for**(**int** i=0;i<diskBlock.size();i++){ 67. **if**(hasgetSpace>=size){ 68. cipanHasget=**true**; 69. **break**; 70. } 71. **else** **if**(diskBlock[i]->fileId==-1){ 72. hasgetSpace=hasgetSpace+blockSize; 73. indexFiles[ifd.indexBlockId].blocksID[j]=i; 74. //修改索引块 i 的属性 75. indexFiles[i].fileId=id; 76. indexFiles[i].filename=name; 77. //修改磁盘块文件 id 及颜色 78. diskBlock[i]->fileId=id; 79. cipanHasget=**true**; 80. //修改空闲表 81. **this**->disrtSpaceInFreeTable(i); 82. **break**; 83. } 84. } 85. } 86. }**else**{//大文件，需要多个索引块 87. //分配 起始索引块 中每个磁盘块 88. **for**(**int** j=0;j<blockNum;j++){ 89. //对起始索引块中寻找每个磁盘块 90. **for**(**int** i=0;i<diskBlock.size();i++){ 91. **if**(hasgetSpace>=size){ 92. **break**; 93. } 94. **else** **if**(diskBlock[i]->fileId==-1){ 95. hasgetSpace=hasgetSpace+blockSize; 96. indexFiles[ifd.indexBlockId].blocksID[j]=i; 97. //修改索引块 i 的属性 98. indexFiles[i].fileId=id; 99. indexFiles[i].filename=name; 100. //修改磁盘块文件 id 及颜色 101. diskBlock[i]->fileId=id; 102. //修改空闲表 103. **this**->disrtSpaceInFreeTable(i); 104. **break**; 105. } 106. } 108. } 109. **int** lastPos=ifd.indexBlockId;//所在块号 110. **int** end=-2;//终止块号 111. // 继续寻找下一个索引块 112. **while** (hasgetSpace<size) { 113. //索引块数量不够，继续分配 114. //寻找第一个空闲块存储作为下一个索引块 115. **int** nextIndBlockId;//下一个索引块所在 id 116. **for**(**int** i=0;i<diskBlock.size();i++){ 117. **if**((diskBlock[i])->fileId==-1){ 118. **int** rr=i/(ui->block\_tab->columnCount()); 119. **int** cc=i%(ui->block\_tab->columnCount()); 120. nextIndBlockId=i; 121. //为索引块分配磁盘 122. diskBlock[i]->fileId=id; 123. //修改颜色 124. **this**->disrtSpaceInFreeTable(i); 125. **break**; 126. } 127. } 128. //修改索引块首尾指针 129. (indexFiles[lastPos]).nextIndexBlockId=nextIndBlockId; 130. lastPos=nextIndBlockId; 131. //修改下一个索引块属性 132. (indexFiles[lastPos]).fileId=id; 133. (indexFiles[lastPos]).filename=name; 134. //对下一个索引块中寻找每个磁盘块 135. **int** sss=(size-hasgetSpace)/blockSize; 136. **int** yyy=(size-hasgetSpace)%blockSize; 137. **int** needCNumm;//需要磁盘块数 138. **if**(yyy>0){ 139. needCNumm=sss+1; 140. }**else**{ 141. needCNumm=sss; 142. } 143. **if**(needCNumm>=blockNum){ 144. //一个索引块中最多只能分配 blockNum 个磁盘块 145. //寻找 blockNum 个磁盘块 146. **for**(**int** k=0;k<blockNum;k++){ 147. **for**(**int** i=0;i<diskBlock.size();i++){ 148. **if**(hasgetSpace>=size){ 149. **break**; 150. } 151. **if**(diskBlock[i]->fileId==-1){ 152. hasgetSpace=hasgetSpace+blockSize; 153. indexFiles[nextIndBlockId].blocksID[k]=i; 154. //修改索引块 i 的属性 155. indexFiles[i].fileId=id; 156. indexFiles[i].filename=name; 157. //修改磁盘块文件 id 及颜色 158. diskBlock[i]->fileId=id; 159. //修改空闲表 160. **this**->disrtSpaceInFreeTable(i); 161. **break**; 162. } 163. } 164. } 165. }**else**{ 166. // needCNumm<blockNum 只需要分配 needCNum 个磁盘 167. **for**(**int** k=0;k<needCNumm;k++){ 168. **for**(**int** i=0;i<diskBlock.size();i++){ 169. **if**(hasgetSpace>=size){ 170. **break**; 171. } 172. **if**(diskBlock[i]->fileId==-1){ 173. hasgetSpace=hasgetSpace+blockSize; 174. indexFiles[nextIndBlockId].blocksID[k]=i; 175. //修改索引块 i 的属性 176. indexFiles[i].fileId=id; 177. indexFiles[i].filename=name; 178. //修改磁盘块文件 id 及颜色 179. diskBlock[i]->fileId=id; 180. //修改空闲表 181. **this**->disrtSpaceInFreeTable(i); 182. **break**; 183. } 184. } 185. } 186. } 187. } 188. }  3.6 **与进程模块的联系** 打开文件时用创建文件时的遍历方法将磁盘块所在的磁道号加入 cyIDorder 中，然后传给进程模块，进程模块创建一个新进程用于调度磁盘文件块用于磁盘调度。  w->processTab->Create\_Process\_For\_File(cyIDorder); 3.7 磁盘调度 当打开文件时，生成“file”文件，调用磁盘调度算法。  磁盘部分设计：每个方格表示一个磁盘块，一行模拟代表一个磁道。磁道号从0-7，分别对应磁盘表格的1-8行。    模拟磁盘调度采用的SCAN算法，先将需要访问的磁道号排序，然后模拟每次磁头的位置都在磁道号1，定位磁头处于什么位置：   1. 若磁头处于访问磁道号顺序左边，则向右扫描 2. 若磁头处于访问磁道号顺序右边，则向左扫描 3. 若磁头处于访问磁道号顺序中间，则先向右扫描，然后回到磁头初始位置向左扫描   部分代码如下   1. //找到起始位置位于访问序列的相对位置 2. **if**(start\_index < seq[0]){ 3. **for** (**int** i=0;i<seq.size();i++) { 4. file->track[i] = seq[i]; 5. } 6. total\_mov\_distance = seq[num-1]-start\_index; 7. }**else** **if**(start\_index > seq[num-1]){ 8. **for** (**int** i=seq.size()-1;i>= 0;i--) { 9. file->track[i] = seq[i]; 10. } 11. total\_mov\_distance = start\_index-seq[0]; 12. }**else**{ 13. //找到磁头在哪个具体位置 14. **for** (**int** i=0;i<seq.size();i++) { 15. **int** index = 0; 16. **if**(seq[i]<start\_index && seq[i+1]>=start\_index){ 17. //先向右扫描 18. **for** (**int** j=i+1;j<seq.size();j++) { 19. file->track[index++] = seq[j]; 20. } 21. //然后向左扫描 22. **for** (**int** j=0;j<i+1;j++) { 23. file->track[index++] = seq[j]; 24. } 25. } 26. } 27. total\_mov\_distance = seq[num-1]-seq[0]+seq[num-1]-start\_index; 28. }   4. **设备管理：考虑设备的分配和回收，采取何种数据结构，采用何种分配算法，使用银行家算法避免死锁。** 4.1 进程队列的生成 点击生成按钮，产生一个进程个数为24的进程队列。  进程的生成是用随机函数（默认为24个进程），进程的资源请求个数随机生成，allocation向量大于max，则随机数重新赋值。  进程队列生成后，通过ui文件将其展示出来。  descript  并且通过计算得出available向量（默认初始资源有200个）.  descript  矩阵展示：  descript 4.2发送请求 通过输入，得到当前请求的进程以及request队列  descript  若请求资源大于need向量，则拒绝请求。  descript  descript  请求合理则执行安全性算法检测  descript 4.3 安全性算法 执行安全性算法，通过保存available副本，循环扫描进程队列，若找到可执行的进程，将Finishied[i]置1，更新副本，且保存上一次副本，每次循环之前对比新旧副本，若相同，且Finished数组还有进程为false，则安全性算法不通过，否则通过。（进程的各个向量都是通过定位到该进程行的某一列得到。）   1. //安全性算法 2. **bool** Dequip::security(**int** pId){ 3. **int** oldData[4]; 4. **for** (**int** i = 0; i < 4; i++){ 5. oldData[i] = ui->tableWidget->item(0, i + 13)->text().toInt(); 6. } 7. **while**(**true**){ 8. **bool** isSame = **true**; 9. **for** (**int** i = 0; i < 24; i++){ 10. **bool** isOk = **true**; 11. **if** (i == pId)**continue**; 12. **if** (Finished[i] == **true**) **continue**; //如果当前进程已分配，则继续循环 13. isOk = **false**; 14. } 15. } 16. **if** (isOk == **true**){ 17. newFinished[i] = **true**; } 18. } 19. **for** (**int** i = 0; i < 24; i++){ 20. **if** (Finished[i] != newFinished[i]){ 21. isSame = **false**; 22. **for** (**int** j = 0; j < 24; j++){ 23. Finished[j] = newFinished[j]; 24. } 25. } 26. } 27. **if** (isSame == **true**){ 28. **break**; 29. } 30. } 31. **for** (**int** i = 0; i < 24; i++){ 32. **if** (i == pId) **continue**; 33. **if** (Finished[i] == **false**){ 34. **return** **false**; 35. } 36. } 37. **return** **true**; 38. }   descript 4.4资源分配 改变矩阵中的a向量和available向量表示资源的分配，并开始计时。   1. **for** (**int** i = 0; i < 4; i++){ 2. myTimerId = **this**->startTimer(time); 3. pe[peCount].timeId = myTimerId; 4. pe[peCount].pId = pId; 5. peCount++; 6. QMessageBox::information(**this**,"提示","请求占用资源成功，该进程占用资源"+QString::number(time)+"毫秒");  4.5资源回收 计时器到时间后，触发回收事件，该进程向量置零，availab向量增加。   1. **void** Dequip::timerEvent(QTimerEvent \*event){ 2. **for** (**int** i = 0; i < 500; i++){ 3. **if** (event->timerId() == pe[i].timeId){ 4. **for** (**int** j = 0; j < 4; j++){ 5. ui->tableWidget->setItem(0, j+13, **new** QTableWidgetItem(QString::number(ui->tableWidget->item(0, j+13)->text().toInt()+ui->tableWidget->item(pe[i].pId, j+5)->text().toInt()))); 6. ui->tableWidget->setItem(pe[i].pId, j+5, **new** QTableWidgetItem("0")); 7. ui->tableWidget->setItem(pe[i].pId, j+9, **new** QTableWidgetItem("0")); 8. } 9. QMessageBox::information(**this**, "提示", "进程占用资源结束，该进程占有的所有资源已归还"); 10. **this**->killTimer(pe[i].timeId); 11. } 12. } 14. }  4.6银行家算法代码和安全性检测代码 （QT界面代码有删减）   1. void Dequip::compareSlot(int requestData[], int pId, int time){ 2. for (int i = 0; i< 5; i++){ 3. rrequest[i] = requestData[i]; 4. } 5. if (requestData[0] <= ui->tableWidget->item(pId,9)->text().toInt() && requestData[1] <= ui->tableWidget->item(pId,10)->text().toInt() && requestData[2] <= ui->tableWidget->item(pId,11)->text().toInt() && requestData[3] <= ui->tableWidget->item(pId,12)->text().toInt()){ 6. if (requestData[0] <= ui->tableWidget->item(0,13)->text().toInt() && requestData[1] <= ui->tableWidget->item(0,14)->text().toInt() && requestData[2] <= ui->tableWidget->item(0,15)->text().toInt() && requestData[3] <= ui->tableWidget->item(0,16)->text().toInt()){ 7. *//执行安全性算法* 8. for (int i = 0; i < 4; i ++){ 9. ui->tableWidget->setItem(0, i+13, new QTableWidgetItem(QString::number(ui->tableWidget->item(0, i+13)->text().toInt()-requestData[i]))); 10. } 11. Finished[pId] = true; 12. *//更新剩余资源数* 13. myTimerId = this->startTimer(time); 14. pe[peCount].timeId = myTimerId; 15. pe[peCount].pId = pId; 16. peCount++; 17. } 18. else{ 19. return; 20. } 21. } 22. else{ 23. emit sendWaitToWidget(pId); 24. } 25. } 26. else{ 27. QMessageBox::warning(this,"警告","请求资源数大于所需资源数，拒绝请求！"); 28. } 29. } 30. //安全性算法 31. **bool** Dequip::security(**int** pId){ 32. **int** oldData[4]; 33. **while**(**true**){ 34. **bool** isSame = **true**; 35. **for** (**int** i = 0; i < 24; i++){ 36. **bool** isOk = **true**; 37. **if** (i == pId)**continue**; 38. **if** (Finished[i] == **true**) **continue**; //如果当前进程已分配，则继续循环 39. **for** (**int** j = 0; j < 4; j++){ 40. **if** (ui->tableWidget->item(i, j + 9)->text().toInt() > ui->tableWidget->item(0, j + 13)->text().toInt()){ 41. isOk = **false**; 42. } 43. } 44. **if** (isOk == **true**){ 45. newFinished[i] = **true**; 46. } 47. } 48. } 49. **for** (**int** i = 0; i < 24; i++){ 50. **if** (Finished[i] != newFinished[i]){ 51. isSame = **false**; 52. **for** (**int** j = 0; j < 24; j++){ 53. Finished[j] = newFinished[j]; 54. } 55. } 56. } 57. **if** (isSame == **true**){ 58. **break**; 59. } 60. } 61. **for** (**int** i = 0; i < 24; i++){ 62. **if** (i == pId) **continue**; 63. **if** (Finished[i] == **false**){ 64. **return** **false**; 65. } 66. } 67. **return** **true**; 68. } | | | | | | | | | | | |
| 四、 **测试与运行结果** 1.进程模块： 创建进程时，所有进程就绪  descript  开始调度，由下图可见，进程“获取键值”在“键盘输入”未运行完，只执行了一部分时调入运行队列后阻塞。同时“键盘输入”在运行自己的第三个时间片（一个时间片2s，图上显示第6秒）时，请求调入第19号页面。 且运行完的进程所占用的内存块释放。  descript  磁盘调度时显示磁盘调度信息  访问磁道号2，磁盘模块第三行正在被访问，变为红色   2.内存模块 内存初始化状态  descript  虚拟内存初始化状态：  descript  运行时：  descript  descript 3.文件模块 文件初始创建时  descript  descript  **扩展各个文件后**  descript  操作删除文件  descript  操作后  descript 设备管理模块 **4.1 进程队列的生成**  点击生成按钮，产生一个进程个数为24的进程队列。  descript  点击每个进程可以查看该进程详情  descript  **4.2 进程资源矩阵展示**  点击开始，展示资源的分配以及进程请求的详细情况  descript  **4.3 发送请求**  点击发送请求按钮，开始发出请求。  descript  若请求资源大于need向量，则拒绝请求。  descript  descript  请求合理则执行安全性算法检测  descript  **4.4 安全性算法**  执行安全性算法，  descript  **4.5 资源分配**  通过则分配资源，进程的need减少，allocati增加，available减少。  例如：  descript  请求后（request = (1,1,1,1)）  descript)  descript  **4.6资源回收**  资源占有时间到后，则回收，该进程allocation和need置零，available=available+allocationi  descript  descript | | | | | | | | | | | |