《操作系统原理》

实习报告

班级 191212

学号 20211003740

姓名 朱畅

院系 计算机学院

专业 计算机科学与技术

指导老师 张求明

完成日期

**目录**

**一．实验一、作业调度算法**

1.1实习题目……………………………………………………3

1.2程序功能及设计思路

1.3数据结构及算法设计

1.4程序运行情况

1.5编程中遇到的困难及解决方法、实习心得或良好建议

**二．实验二、磁盘调度算法**

2.1实习题目……………………………………………………7

2.2程序功能及设计思路

2.3数据结构及算法设计

2.4程序运行情况

2.5编程中遇到的困难及解决方法、实习心得或良好建议

**三．实验三、熟悉linux文件系统调用**

3.1实习题目……………………………………………………13

3.2程序功能及设计思路

3.3数据结构及算法设计

3.4程序运行情况

3.5编程中遇到的困难及解决方法、实习心得或良好建议

**四．实验四、进程管理**

4.1实习题目……………………………………………………22

4.2程序功能及设计思路

4.3数据结构及算法设计

4.4程序运行情况

4.5编程中遇到的困难及解决方法、实习心得或良好建议

**五．实验五、进程通信**

4.1实习题目……………………………………………………22

4.2程序功能及设计思路

4.3数据结构及算法设计

4.4程序运行情况

4.5编程中遇到的困难及解决方法、实习心得或良好建议

1. **实验一  多级队列调度算法**

**1.1实习题目**

利用多级队列调度算法，实现进程的调度，并输出进程的周转时间和全部进程的平均 周转时间。具体内容如下：

利用两个不同等级的队列，RQ1 和 RQ2，RQ1 采用轮转法，时间片为 q=7，RQ2 采用 短进程优先调度算法，其中 RQ1>RQ2，即 RQ1 的优先级高于 RQ2。测试数据如下图所示：

文本

描述已自动生成

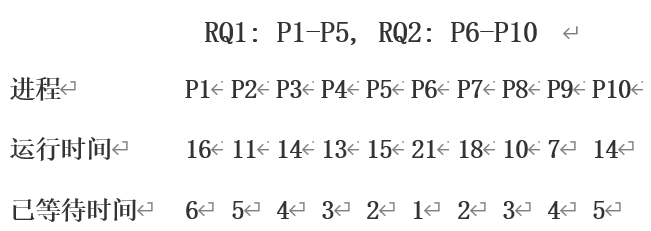


图1.1实习1测试数据

1.2程序功能及设计思路

**1.2.1功能**

对于给定的数据使用多级队列调度算法进行分析计算周转时间。其中多级队列分为RQ1和RQ2 ，RQ1采用的是时间片长度为7的时间片轮转算法，RQ2采用的是短进程优先算法。并且RQ1的优先级高于RQ2（即只有在RQ1内所有程序运行结束，RQ2才能开始运行）

**1.2.2设计思路**

短进程优先算法：涉及到排序函数，插入函数；首先开始对短进程队列按完成所需时间从小到大排序，用current=RQ2，再执行一个while循环，结束条件是RQ2队列为空，即RQ1指向的next为空，然后逐个满足当前的最小需求进程（队首进程），完成后当前队首进程复制到finish队列中，每次执行完current向后移动一个节点。

时间片轮转算法：涉及到插入函数；先用current=RQ1，取得当前的第一个进程点，然后让变量t=timepiece，针对t与current的need情况，分情况讨论，当t<need时，无法完成当前进程，当前进程的need-=t，然后将此进程插入到链表尾端。当t>=need时，可以满足进程的需求，t-=need，将当前进程current插入到finish队列中，每次执行完current向后移动一个节点。

最初始还会设计一个进程初始化函数，利用到文件读取功能。

**1.3数据结构及算法设计**

**1.3.1数据结构**

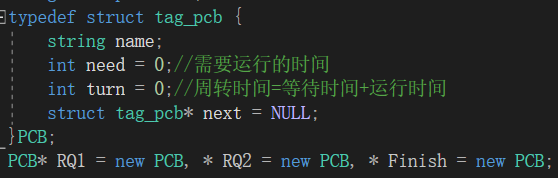


图1.2 多级队列调度算法数据结构

利用结构体设计PCB队列节点，每个节点包含四个部分，分别是名称name、需要运行时间need、周转时间turn以及指向的下一个节点next。两个队列的头结点分别是RQ1和RQ2，以及完成队列的头结点Finish。

**1.3.2算法设计**

设计了RQ2排序算法，按照RQ2进程队列的需求时间从小到大进行排序。使用排序算法之后才可使用短进程优先算法。

初始化算法，利用文件读入操作，给RQ1和RQ2两个队列赋予初始值。

**1.4程序运行情况**



图1.3 多级队列调度算法程序运行情况

如上图2所示，根据文件读取进入的数据，RQ1队列先执行，待RQ1执行完毕后，RQ2才开始执行。执行的顺序依次是P2、P3、P4、P1、P5、P9、P8、P10、P7、P6；总的周转时间是504。

**1.5总结**

**1.5.1问题**

（1）排序算法不熟

对于此题，各个进程之间是靠链表连接起来的，鉴于学习内容比较狭隘，我并不熟悉链表的排序算法，根据自己的想法编写了一个排序算法，可是运行时时常出现空指针的情况，为此我很苦恼，最后在网络上寻求帮助找到了一个合适的链表排序算法。我认为链表排序的难点就是指针较多，容易混淆，然后对于链表节点的前后交换不熟。而且空指针也是时常令人头疼的一大难点。

（2）指针未置空

在编写RQ1的插入算法时，最终给插到末尾的赋值之后便未操作，在调试程序的时候发现由于未将最末尾的节点置空，我的队列成了循环链表，一直不能结束while循环，最终导致程序运行时间过长而报错。

（3）文件名不加.txt

读取文件的时候写代码需要加.txt，但在文件命名时并不需要加.txt后缀。

**1.5.2实习心得及建议**

（1）要广泛查阅网络上的学习资源。不能一味的封闭式自我孤立学习，有可能确实是将老师布置的任务实现了，但是方法不一定是最好的，可能在网络上有更加优化的版本等待我们去学习，所以我们不能闭门造车，要学会发挥网络的利处。

（2）多调试代码。随着编程经验的不断积累，我最大的感触就是设置断点调试代码是真的好用，当我的程序运行出现问题的时候，我会果断的设置断点然后逐步调试去寻找我的问题出在哪里然后一步一步去更改我的代码。

（3）链表的最末尾指针一定要置空，不然会有形成循环链表的危险。

（4）轮转法：根据程序已等待时间排序，先分配时间片执行等待时间久的程序，若执行完就添加到finish队列中，没能执行完就添加到队尾等待下次分配时间片。

（5）短进程优先调度算法：根据程序的需求时间从小到大排序，优先执行完成需求较小的程序，根据程序队列依次往后执行。

**二．实验二 银行家算法**

**2.1．实习题目**

利用银行家算法，实现系统对多个进程多个资源的分配，并通过安全状态判别算法， 防止系统进入死锁状态。该实验数据如下 m=3 种类型的资源（A,B,C,） 进程个数 n=5。Available=(2,3,3);测试数据如下图所示：



图2.1 实习二测试数据

**2.2程序功能及设计思路**

**2.2.1功能**

对于已经安排的系统资源和进程，当某个进程发出申请请求某些资源时，会调用此银行家算法方法来验证分配是否合理，分配后系统资源是否处于安全状态 ，若不是安全状态，则有可能发生死锁，导致程序进程的崩溃，若分配请求后仍为安全状态，那么则执行此次申请的请求。

**2.2.2设计思路**

操作系统安全状态和不安全状态,，等价于操作系统是否存在一个安全序列。安全序列是指一个进程序列{P1，…，Pn}是安全的，即对于每一个进程Pi(1≤i≤n），它以后尚需要的资源量不超过系统当前剩余资源量与所有进程Pj (j < i )当前占有资源量之和。

**2.3数据结构及算法设计**

**2.3.1数据结构**

图片包含 图形用户界面

描述已自动生成

图2.2 银行家算法数据结构1-3

定义两个变量m，n分别代表资源种类数量和进程个数。定义了长度为m的数组available代表当前系统空闲的资源；定义了n行m列的二维数组alloc和need，alloc的n行代表第n个进程，m列代表第m个资源，所以alloc[n][m]代表第n个进程的第m个资源已经分配了多少资源；need[n][m]代表第n个进程的第m个资源还需要分配多少个资源。

根据解决实际问题需要，也设计了

屏幕上有字

描述已自动生成

图2.3 银行家算法数据结构2-3

Bool类型的finish数组，长度为进程个数n，初始化的值均赋值为0，在循环检测安全状态时会用到此数组，最后判断整个进程队列是否都是true，是则满足安全检测，反之则否。长度为资源数量m的Int型的数组work，此数组会赋值为available空闲数组，用来替代available做运算操作。

文本

描述已自动生成

图2.4 银行家算法数据结构3-3

定义了长度为资源数量m的request矩阵m，每个元素代表当前申请的资源请求数量。

**2.3.2算法原理**

（1）银行家算法原理：

* 当某个进程Pi提出资源请求Request时，
* 1 检测Request≤Needi，若不成立则表示出错；
* 2 检测Request≤Available，若不成立则表示可用资源不够用，拒绝分配，阻塞进程Pi；
* 3 试探性分配，系统进入新状态：
  + - Alloci+=Request;
    - Needi-=Request;
    - Available-=Request;
* 4 检测新状态是否安全，若安全则实施资源分配，否则取消试探性分配，阻塞进程Pi。

（2）安全状态判别算法：

* 1初始化：
* Work=Available且全部进程标记为未完成;
* 2 查找满足如下条件的进程Pi：
  + - * 该进程未完成且Needi≤Work
* 如果没有这样的进程存在，则转4；
* 3 将资源分配给所选进程Pi，该进程最终能运行完，标记这个进程为完成进程，并将它占有的全部资源归还给向量Work（Work+=Alloci），转2循环查找；
* 4 若所有进程标记为完成则系统为安全状态，否则不安全。

文本

描述已自动生成

图2.5 安全状态判别算法原理

首先进入一次for循环，循环次数为进程数量n，最终看for循环之后各个进程的finish状态是否被置为true，若出现非全true则代表着当前的请求不满足安全状态，不能进行分配。将错误提示为mistake置为0，然后寻找当前i对应的finish【i】值为false的进程，针对此进程进行资源分配考虑，利用一个大小为m的for循环对当前进程的need资源与当前空闲资源work进行比较，当且仅当此进程的所有资源需求need都小于work时才能进行到下一步，不然将mistake置为1,并且跳出当前循环，回到最外层的for循环寻找下一个false进程；对于满足mistake等于0的进程，则进入到资源分配环节，现将finish【i】置为true，证明这个进程可以完成分配，然后将mistake重新置为0，将当前的进程角标（i+1）添加到vector容器中作为分配队列的顺序存储起来。然后对work数组逐个加上当前进程的alloc值，最终将i赋值为-1，下一次循环时会从0开始运行，重新寻找整个finish数组中的false成员，直到所有指令都完成后。

**2.4程序运行情况**

文本

描述已自动生成

图2.6 银行家算法程序运行情况

如上图7所示，当进程2请求资源为0、3、4时，程序运行结果为请求资源过大，请求失败，原因是c资源目前空闲为3个，而此命令申请了4个，超出了当前的空闲资源量，所以会导致资源分配失败；当进程4请求资源为1、0、1时，最终返回的安全序列结果为4 2 3 5 1，由此可得是可以满足进程四提出的资源申请的，完成资源大小检测，即available是否大于request，need是否大于request，都大于的情况下通过安全检查，系统尝试将资源分配给目前的申请，然后再进行安全判断，考虑此情况下是否会导致死锁和不安全状态，如果导致不安全（safecheck函数最终的finish数组值不全部为true），那么不执行此次资源分配的请求。第3和4次指令原理与上述相似，在此不再赘述。

**2.5总结**

**2.5.1问题**

（1）函数参数掌握程度不够

对于传入函数safecheck的vector成员vec（用来记录资源分配的顺序），最初是按照值传递进入到函数中，最终显示输出vec成员值时发现vec为空，这时我才意识到不应该用值传递而应该使用引用传递，值传递不会改变实参的值，传入时只是将实参的值拷贝一份进入函数中运行，所以不会有元素插入到真正的vec容器中，改为引用传递之后问题果真解决了。

（2）重新循环时i应该置为-1

对于一个第三个参数为i++的循环，想要此循环重新从0开始时，应该将i的值置为-1，下次进入循环时i会自动进行i++最终的值变为0；而不是置为0，这样进入循环的值会变为1。

**2.5.2实习心得及建议**

（1）编程实现算法并没有想象中的那么复杂，只要将核心思想和伪代码弄清楚，按照这些思想一步一步写算法，遵循严格完整的逻辑，就不会出现太多的错误，遇到问题和bug要及时调试来解决问题。

（2）银行家算法是预防动态资源分配死锁问题的一大妙招，通过这次编程实现银行家算法，我更深入了解到了这个算法的内在思想和底层逻辑，要学会利用矩阵进行运算。用矩阵形式化表示一些数据也会使程序更加的简洁健壮。

**三．实验三 动态分区式存贮区管理**

**3.1．实习题目**

实现一个动态分区式存贮区管理程序，要求支持不同的放置策略，如首次适应策略、 最佳适应策略、最坏适应策略。

测试实例数据如下： 主存大小 Maxsize = 512, J1 申请 162，J2 申请 64，J3 申请 120，J4 申请 86，J1 完 成，J3 完成，J5 申请 72，J6 申请 100，J2 完成，J7 申请 36，J8 申请 60，J4 完成，J9 申 请 110，J10 申请 42。 所有大小单位为节（1 节=rd 的大小）。作业申请 n 节,实际分配的分区大小应为 n+1 节，其中一节作为分区描述器，其他 n 节提供给作业。

**3.2程序功能及设计思路**

**3.2.1功能**

对于用户输入的一系列待进入主存的进程，输入参数有进程的名称和进程的大小，程序完成对用户输入进程的放置，并返回进程的存放地址，对于用户还显示输出当前空闲主存的序列，方便用户进行程序调动操作。用户输入不同的数字，系统将分别按照最佳适应算法、优先适应算法、最坏适应算法给进程分配空闲区的空间。

**3.2.2设计思路**

利用链表对空闲分区进行连接操作，根据不同的适应算法，链表连接的方式也有所不同，比如对于最佳适应算法，空闲分区按照空闲分区的大小从小到大排序，确保每个进程分到的空闲区位置是最合适的，即浪费的程度最小；对于优先适应算法，空闲分区按照其首地址升序排列，优先满足使用低地址位置的空闲分区；对于最坏适应算法，空闲分区按照其大小降序排列，保证每次都是使用的大空闲区的空间，可以确保大作业正常执行。

**3.3数据结构及算法设计**

**3.3.1数据结构**

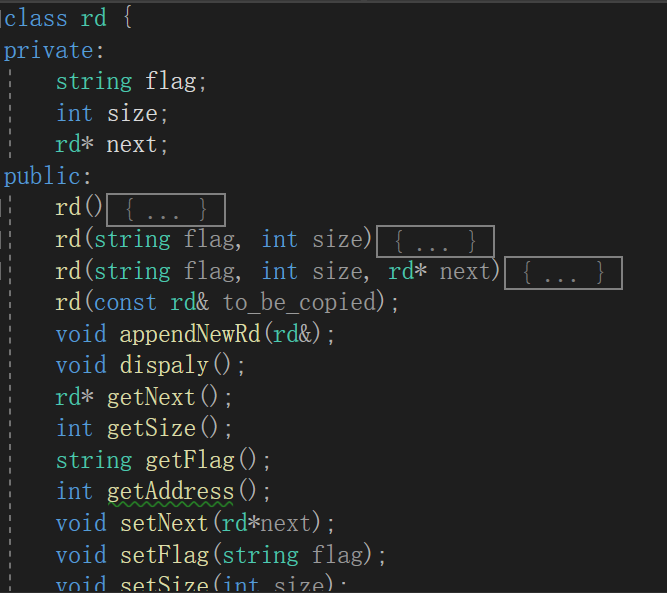


图3.1 rd分区描述器类

根据题目要求设置了rd分区描述器类，来描述各个空闲分区的信息，类中的数据结构有空闲分区的标志flag，0为空闲，1为被占用；空闲分区的大小size；指向下一个空闲分区的指针next。

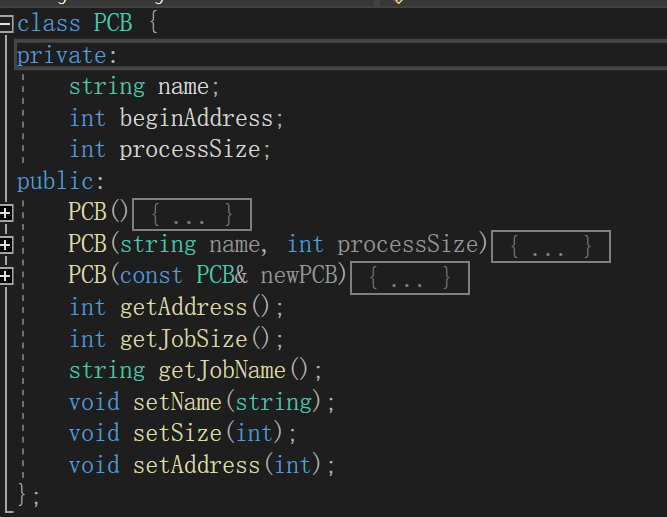


图3.2 PCB进程管理块类

如上图所示，还设置了PCB进程管理块，来管理不同作业对应的不同需求。类中的数据结构有描述进程信息的进程名称name，进程的起始位置beginAddress，进程的大小processSize。



图3.3 rdAndPCBoperator类

仅仅有rd分区描述器类和PCB进程控制块类还不够，这些只是会被利用到的数据结构，还需要有一个类来操作这些数据结构，于是便设置了rdAndPCBoperator类。类中的数据结构有指针free，为rd类型，是当前空闲分区的管理者，空闲分区通过free指针组织起来，类似的有工作分区occupied。还有部分在函数中会使用到的数据，比如要求用户输入model，程序根据输入model的不同来进行不同的空闲分区空间分配操作。

**3.3.2算法原理**

（1）释放分区原理

对于最佳适应算法：即将要释放的分区先把标签flag置为0，然后对于空闲分区链表的连接，按照空闲分区的大小从小到大依次插入。依次遍历当前空闲分区的size，直到遍历到下一个空闲分区节点对应的size小于当前分区的size，最终将分区安插在这个节点之后。优先适应算法和最坏适应算法和最佳适应算法的描述十分类似，只是空闲分区链表的组织连接规则不同，优先适应算法，空闲分区按照其首地址升序排列，优先满足使用低地址位置的空闲分区；对于最坏适应算法，空闲分区按照其大小降序排列，保证每次都是使用的大空闲区的空间，可以确保大作业正常执行。

（2）申请分区原理



图3.4 Request申请分区算法

申请空间时，不需要关心空闲分区的连接顺序，其连接顺序在空闲分区释放连接在空闲分区链表里时已经被Release释放算法确定了，所以当前空闲分区里的分区都是按顺序排列的，只需要根据给的free空闲分区首节点依次遍历整个分区节点，找到第一个空闲分区size大于当前进程所需要size的空闲分区，在进行分配即可。

**3.4程序运行情况**

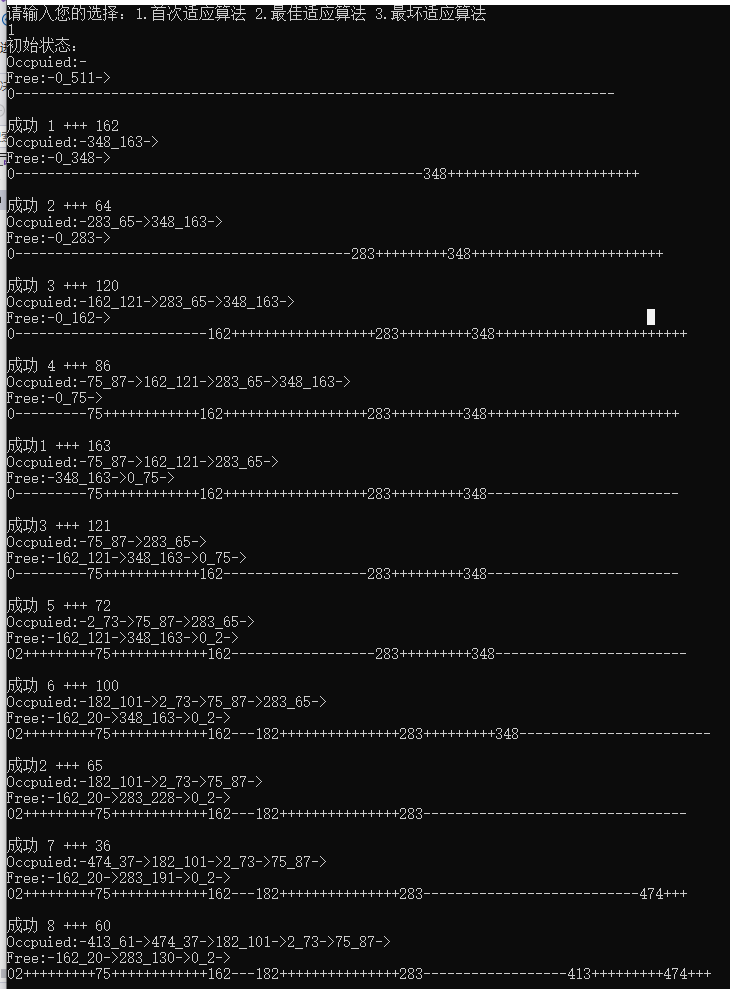


图3.5 首次适应算法结果图1/2

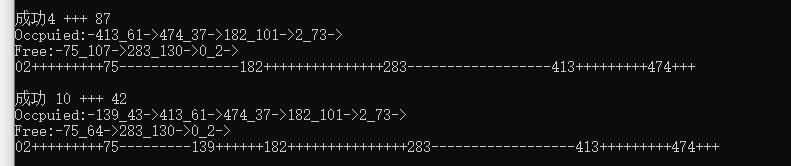


图3.6 首次适应算法结果图2/2

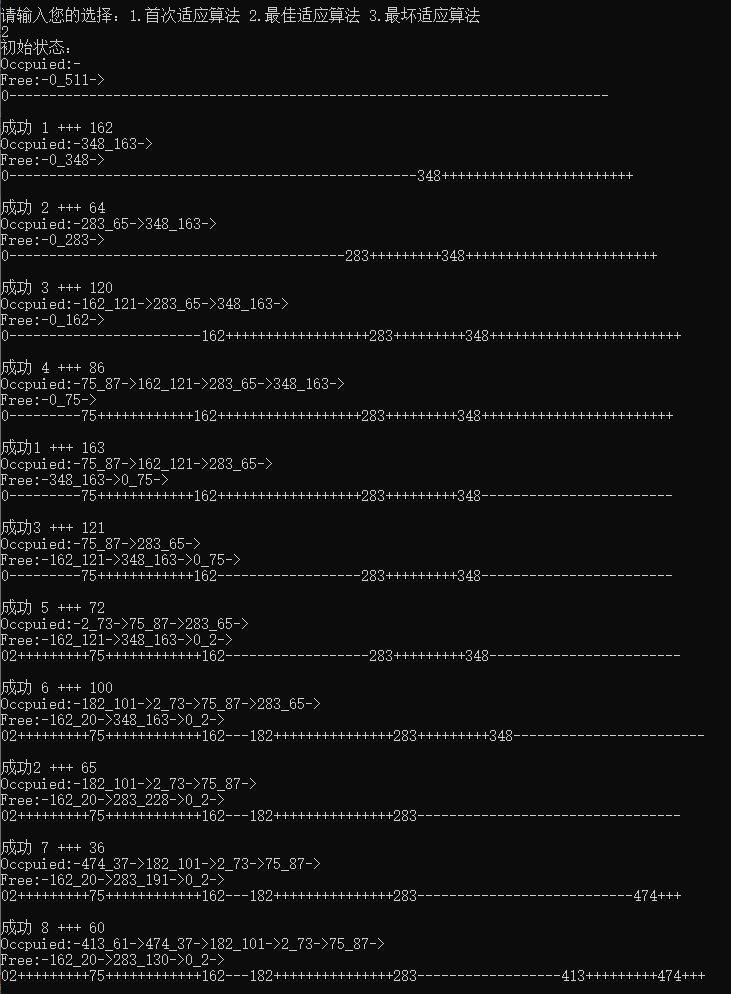


图3.7 最佳适应算法结果图1/2

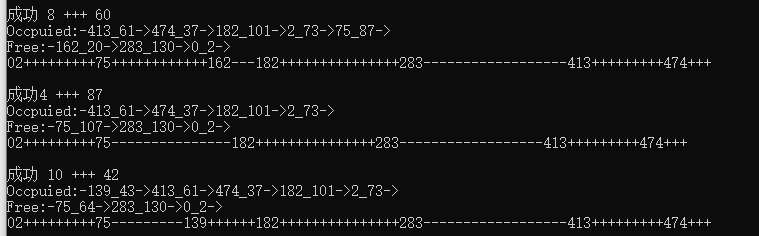


图3.8 最佳适应算法结果图2/2

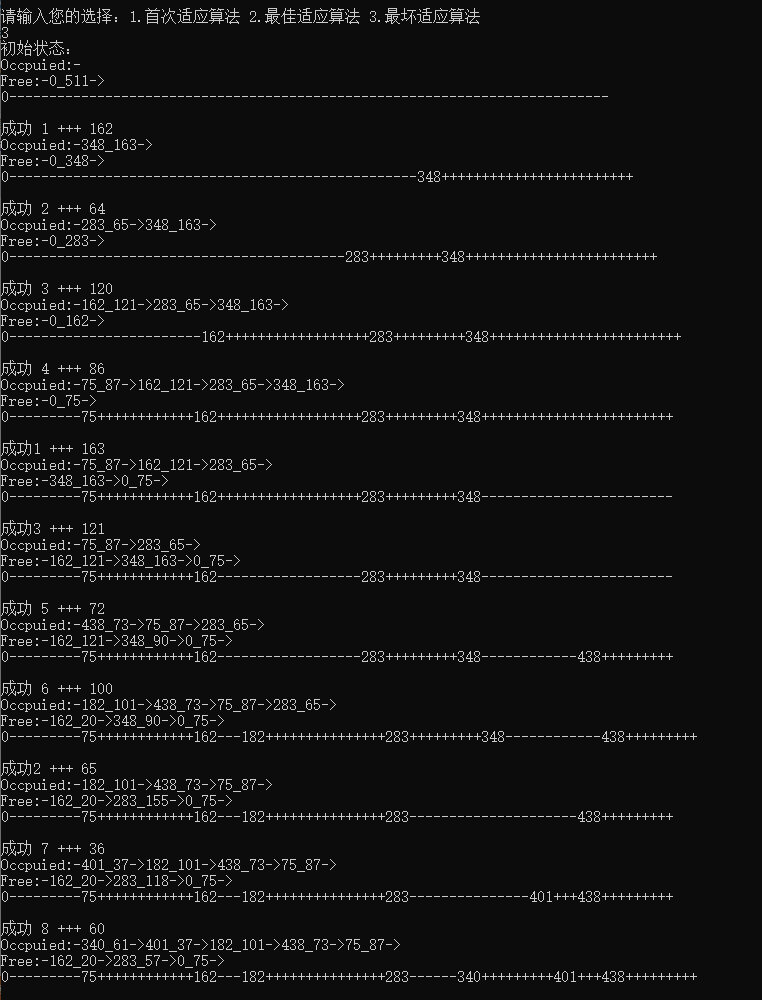


图3.9 最坏适应算法结果图1/2

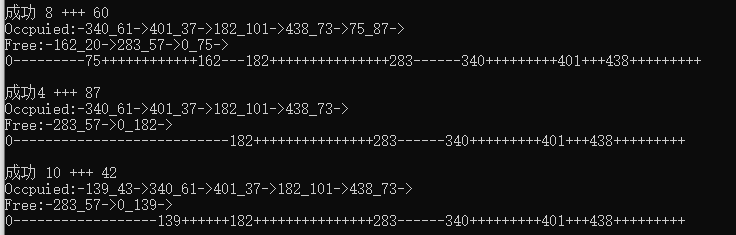


图3.10 最坏适应算法结果图2/2

**2.5总结**

**2.5.1问题**

（1）细节问题

比如拷贝构造函数编写仍不熟练，编写append函数调试的时候发现append后面的节点flag值没有传入，只穿入了size值，后面发现应该把函数传入的参数string改为引用传递而不是值传递，值传递不会改变参数的值。还有许多细节错误，比如解决了系统报错问题。

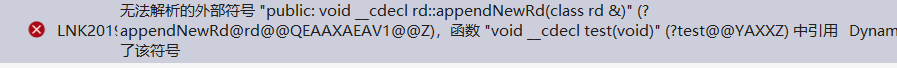


图3.11 系统报错

出现了这个报错指令，通过网上查阅资料之后了解到，报错的原因是 报这个错误一般是因为使用了一些未定义或者没找到定义的函数或者变量等等。可能原因有很多，比如头文件中声明的函数没有定义；编译环境被设置成了x64，最终改成x86就可以了。

（2）其他问题

一开始以为空闲队列需要人为排序，但是实际实现时才发现空闲区间队列是从零一直递增的，每次的节点连接方式不同，只需要对每次链接进行方法上的区分即可。

申请n节作业位置，实际上分配n+1节位置，一节作为分区描述器，其他n节提供作业

本来是打算写三个查找位置的函数，后面编写完第一个最佳适合放置算法后发现这三个查找位置的函数都是沿着free找第一个空闲区大小大于jobSize的位置，原理都一样，所以就合并成一个了。

请求作业区时，当空闲区与请求大小相等，一开始想法是将当前free链表的节点的值delete掉，后面发现Occupied队列也会使用到这个节点，所以不用直接delete，可以更改flag为“1”并且添加到Occupied链表中去。这样就达到了更改分区的目的。

调试程序时出现class rd重定义的现象，经过查阅资料知道是文件rd.h重复编译了，导致程序报错，解决方法是在rd.h文件头加上#pragma once ，保证此个文件只会被编译一次

**2.5.2实习心得及建议**

该实习实现了动态分区式存贮区管理，加强了我对操作系统主存资源分配和管理的理解，也深刻的体会到了首次适应法、最佳适应法、最坏适应法三者的原理和区别。此次实习的内容较前两次更加复杂，可能主要就是需要去对模拟内存和空闲区队列进行合理的调整。

还加强了我对链表连接，处理的理解程度，能够更好地利用链表这一数据结构来处理问题。也加强了我对类的使用熟练程度。

**四．实验四 请求分页系统中的置换算法**

**4.1．实习题目**

实现一种模拟请求分页系统的请求分页程序。 具体内容如下：

1. 通过如下方法产生一指令序列，共 320 条指令。

①在[1，32k-2]的指令地址之间随机选取一起点 M,访问 M；

②顺序访问 M+1；

③在[0，M-1]中随机选取 M1，访问 M1；

④顺序访问 M1+1；

⑤在[M1+2，32k-2]中随机选取 M2，访问 M2；

⑥顺序访问 M2+1；

⑦重复 ①—⑥，直到执行 320 次指令。

**指令序列变换成页地址流设：**

①页面大小为1K；

②用户虚存容量为 32K。

**计算并输出下述各种算法在不同内存页块(页块个数范围:8-32)下的命中率**。

①先进先出（FIFO）页面置换算法

②最近最久未使用（LRU）页面置换算法

③最佳（Optimal）页面置换算法

**4.2程序功能及设计思路**

**4.2.1功能**

直接返回三种不同页面置换调度算法的命中率，也会返回每个算法的缺页次数，让结果更加的明显，三个算法之间的区别更加直观。

**4.2.2设计思路**

**首先设置实现生成随机数的函数，对于函数而言，按照题目的要求生成320个随机数存放在int类型的指针中，让人为给这些生成的数据作除以1024运算，将所生成的值存入新的int型指针中，这些值代表着指令的页块号。然后利用三个函数分别处理指针中的值，利用不同的页面淘汰算法得到不同的结果。**

**4.3数据结构及算法设计**

**4.3.1数据结构**



图4.1 随机数数据结构

利用srand(time(0))设置随机数种子，利用getRand函数获取各个区间对应的随机数，将随机数整除1024得到的结果存放到instructArr指针中，最终将指针的值返回，这样长度为320的指令数组就初始化完成了。

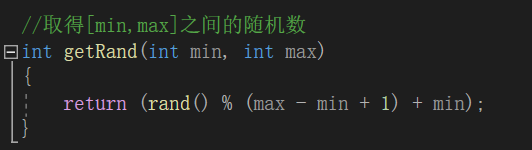


图4.2 生成随机数



图4.3 包含deque头文件

包含deque头文件，将页面的判断是否缺页，页面的淘汰算法都系统的整理到deque容器中执行，利用了deque容器可以头尾操作也可以在中间操作的优点。

**2.3.2算法原理**

（1）FIFO算法原理

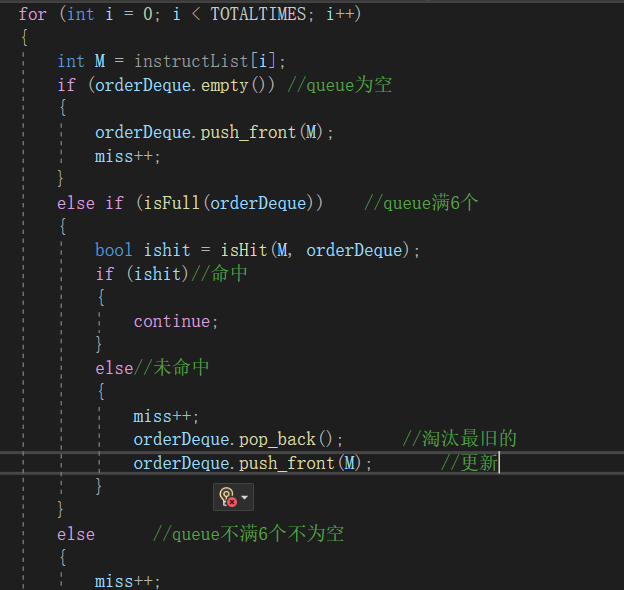


图4.4 FIFO算法

对于FIFO算法，利用一个deque容器，执行320次for循环，每次循环都会取当前的指令链表的值，首先判断deque容器是否为空，为空指令就直接插入主存中，缺页次数miss对应加一；然后对于不为空的情况，判断容器是否已满，满了就判断是否命中，命中了就直接切换到下一条指令，未命中就将容器的最后指令移出，插入当前的指令；对于未满的情况，先判断在不在容器中，在就命中了，不在就直接插入。一直循环320次。

（2）LRU算法原理



图4.5 LRU算法

算法实现上大致流程与FIFO算法类似，这里就不再赘述，只是在淘汰页面的选择上有所变化，具体变化就是在命中页面之后不是对页面什么也不操作，而是命中的页面放到容器首端，所以每次淘汰容器的末尾即可。

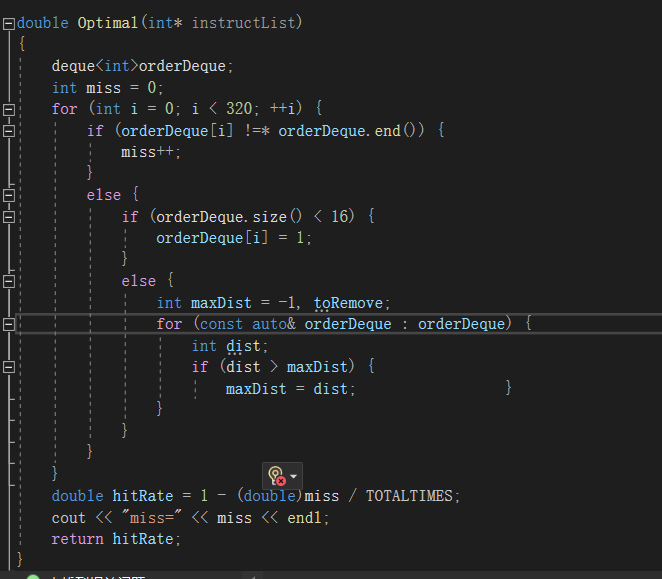


图4.6 最佳页面置换算法

最佳页面置换算法的核心思想就是淘汰的时候考虑以后所有的指令，将当前容器中未来会用到最少次数的指令淘汰出局，这样达到了最有效的置换，也会使命中率提高许多，所以该算法的高效性是三个算法中最强的，但是实际工作中因为不可能一开始就掌握所有指令的执行情况和执行顺序，所以该算法很难以实现。

**3.4程序运行情况**



图4,7实验四运行结果图

此程序是当页面块大小为16k时，即页面块可以存储16个页面时的运行结果，很明显最佳页面置换算法的命中率是最高的，先进先出算法和最近最久未使用算法相对而言差别不是很大，特别是当页面的大小不断增大的时候，二者之间的差距会越来越小，有时候甚至FIFO算法的命中率会大于LRU算法，但这都是正常的现象。

**4.5总结**

**4.5.1问题**

（1）细节问题

使用deque容器时由于编程的疏忽，将迭代器erase之后仍在调用该容器，所以导致多次崩溃。

**4.5.2实习心得及建议**

通过这次实验，我掌握了有效的生成随机数的方法，也学会如何熟练使用deque容器了，在老师的提示下，我了解到了这些算法的实现原理：

1. **FIFO算法**

该算法总是淘汰最先进入内存的页面，即选择在内存中驻留时间最久的页面予以淘汰。该算法实现简单，只需把一个进程已调入内存的页面，按先后次序链接成一个队列，并设置一个指针，称为替换指针，使它总是指向最老的页面。

1. **LRU算法**

最近最久未使用（LRU）页面置换算法，是根据页面调入内存后的使用情况进行决策的。由于无法预测各页面将来的使用情况，只能利用“最近的过去”作为“最近的将来”的近似， 因此，LRU 置换算法是选择最近最久未使用的页面予以淘汰。该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t，当需淘汰一个页面时，选择现有页面中其 t 值最大的，即最近最久未使用的页面予以淘汰。

1. **optimal算法**

该算法选择的被淘汰页面，将是以后永远不使用的，或许是在最长（未来）时间内不再被访问的页面。采用该算法，通常可保证获得最低的缺页率。但由于人们目前还无法预知一个进程在内存的若干个页面中，哪一个页面是未来最长时间内不再被访问的，因而该算法是无法实现的，但可以利用该算法去评价其他算法。