

数据结构作业实验报告

**2018/2019(1)**



作业题目 优先级作业调度系统的模拟

学生姓名 林英琮

学生学号 201706062412

学生班级 数媒1702班

任课教师 王丽萍

提交日期 2019/1/1

**计算机科学与技术学院**

1. **作业内容**

**【问题描述】**Windows、Linux等操作系统都支持同时运行多个作业，但作业的执行顺序却因调度算法的不同而不同。通常，操作系统都采用优先级作业调度，即操作系统根据作业的长短来设置优先级大小，优先级高的作业先执行，优先级低的作业后执行。作业调度的详细情况如下描述：

一个作业Ji的长度为ti =（si，ei），si 为作业运行的开始时间（进入时间），ei 为作业运行的结束时间（离开时间），ti则为完成作业Ji所需要的执行时间（单位：秒）。作业调度的基本任务是从作业队列中选取一个来执行，如果没有作业则执行空操作操作。而优先级作业调度，是指每次选取优先级最高的作业来调度，优先级可以用优先数（每个作业一个优先数pi）来表示，优先数越小，优先级越高。作业Ji 进入系统时，即si 时刻，系统给该作业指定其初始优先数pi = ti，从而使越短的作业优先级越高。该优先数在作业等待调度执行的过程中会不断减小，调整公式为：pi = pi - wi，其中wi 为作业Ji的等待时间：wi = 当前时间-si。一旦作业被调度，该作业就一直执行，不能被抢占，只有当前执行的作业完成时，才产生下一轮调度。所以需要在每次调度前动态调整各作业的优先数。在每次调度的时候，如果出现相同优先级的作业，则按照先进先出(FIFO: First In First Out)的原则进行调度。

**【基本要求】**

1. 要求自己编程实现堆结构及其相关功能，从而实现优先级队列，**不允许使用标准模板类的堆函数和优先级队列**；测试时，各种情况都需要测试，并附上测试截图；
2. 要求采用类的设计思路，不允许出现类以外的函数定义，但允许友元函数。主函数中只能出现类的成员函数的调用，不允许出现对其它函数的调用。
3. 要求采用多文件方式：.h文件存储类的声明，.cpp文件存储类的实现，主函数main存储在另外一个单独的cpp文件中。如果采用类模板，则类的声明和实现都放在.h文件中。
4. 不强制要求采用类模板，也不要求采用可视化窗口；要求源程序中有相应注释；
5. 要求测试例子要比较详尽，各种极限情况也要考虑到，测试的输出信息要详细易懂，表明各个功能的执行正确，包括何时作业进入，何时调度哪个作业，何时离开，每个作业等待多长时间，优先数的动态变化情况等；

要求采用Visual C++ 6.0及以上版本进行调试；

1. **运行环境**

优先级作业调度系统（[Priority\_Scheduling\_System](https://github.com/Congb19/Priority_Scheduling_System)）

在JetBrains CLion 2018.3.2平台下开发。

Language Stantard： C++ 14

操作系统：Windows10 [版本 10.0.17134.407]

硬件环境：

处理器：Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz 3.70GHz

内存：16.00GB

系统类型：64位操作系统

项目地址：<https://github.com/Congb19/Priority_Scheduling_System>

1. **设计思路**

**类的设计：**

// Heap: 最小堆实现优先队列

// Process: 作业进程类

简介：

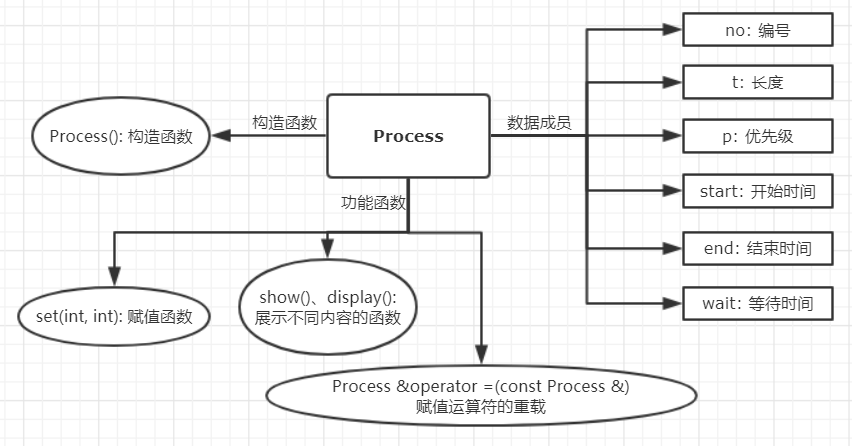


图3.1 Process类图

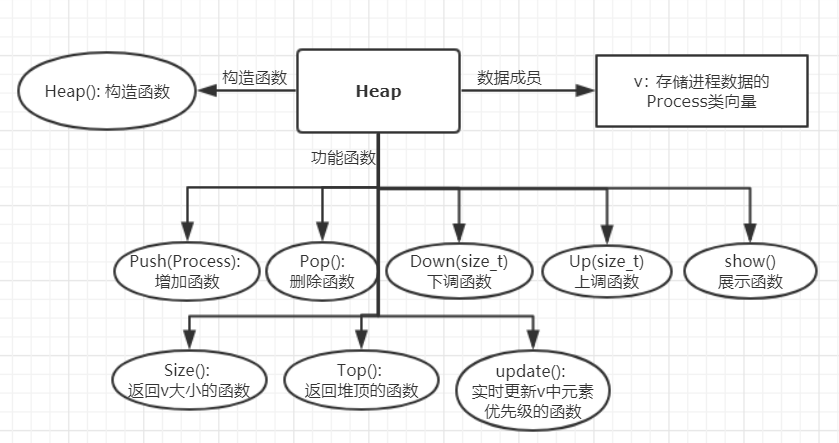


图3.2 Heap类图

**系统总体设计、主程序设计：**

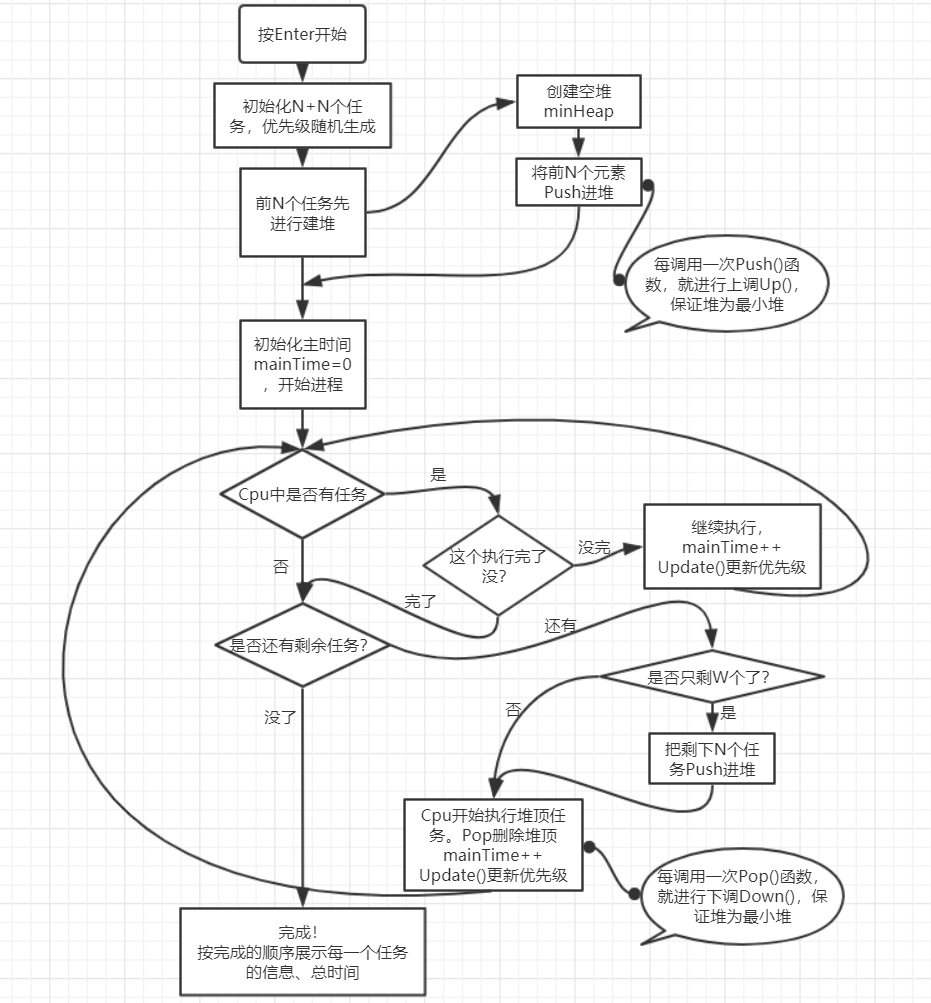


图3.3 主程序流程图

1. 首先，根据提示，为了实现优先队列，编写Process类的最小堆Heap类。

可以采用数组或向量来实现。（写的时候选择了向量）

1. 通过rand()函数来初始化一些优先级随机的任务，每组N个。
2. 用一个变量记录时间轴，再通过调用Sleep(1000); 、判断是否还有剩余任务的方式来模拟整个进程的调度。在剩余任务不足W个时，插入剩余的任务进入队列。

其中：

每一个任务开始调度后，计算它现在的优先级，记录它的开始时间。

处理时间足够长后，记录它的结束时间，并重新开始这一秒的判断，以方便开始下一个任务。

1. 每一秒的最后更新正在等待的任务的优先级。
2. 全部记录完后，按处理完的顺序进行一一展示。

**系统功能设计：**

1. **样例随机**：采用rand()随机产生样例的优先级，
2. **输出内容完整**：输出内容包括了所有任务的初始情况、CPU调度时每一秒发生的事情、全部处理完后按处理完的顺序进行一一展示。
3. **阀值可变**：可以在main文件开头宏定义修改每组样例的个数（N）、自己定义剩余任务多少多少时（W）插入第二组任务。

**思路优化：**

普通的最小堆的上调下调算法不能完全实现FIFO的要求，需要改编。

首先，在下调算法中加入判断，如果p相等，则判断编号（no）的大小，小的优先。

其次，原来的思路（开始进程后计算优先级的变化）会导致剩余的任务优先级未实时更新，反被后插入的小优先级先执行。故思路修正为每个任务实时更新优先级。

具体见**（三）调试错误分析： 问题：3.** 。

1. **测试结果分析**

**（一）技术难点分析：**

1. **最小堆实现优先队列。**

最小堆为最大堆的倒置，是一种经过排序的完全二叉树，其中任一非终端节点的数据值均不大于其左子节点和右子节点的值。

由于是完全二叉树，可以考虑使用数组或者向量来存储数据。（实际在堆中选择使用了向量来存储。）

在其插入、删除操作中，通过上调和下调算法使得最小堆始终为最小堆，使得堆顶总是当前优先级最高的元素(p最小)

1. **为了满足FIFO原则，在原来的判优先级的基础上，下调函数中需要加判序号。(如果p相等，那么序号靠前的应该优先执行。)**

**具体实现过程：在下调函数中增加如下两段：**

if (v[c].p > v[c+1].p) c++; //左 > 右

else if (v[c].p == v[c+1].p) { //左 == 右

if(v[c].no > v[c+1].no) c++;

}

**以及**

else if (v[r].p == v[c].p) {

if(v[r].no > v[c].no) {

swap(v[r], v[c]);

r = c;

c = 2 \* r + 1;

}

else break;

}

**（二）某一次的运行过程截图：**

**（样例为随机生成）**

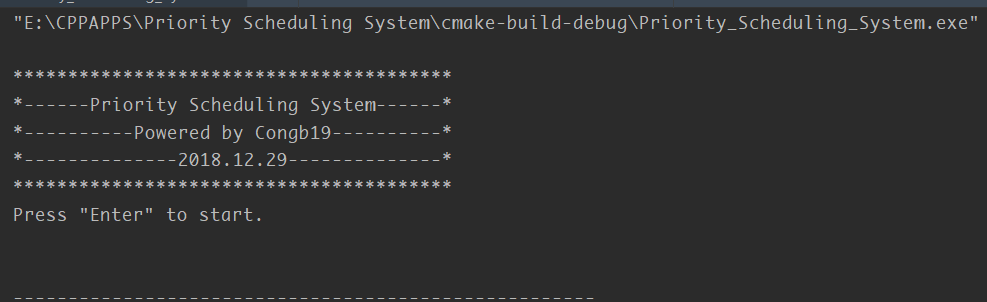


图4.1 开始界面

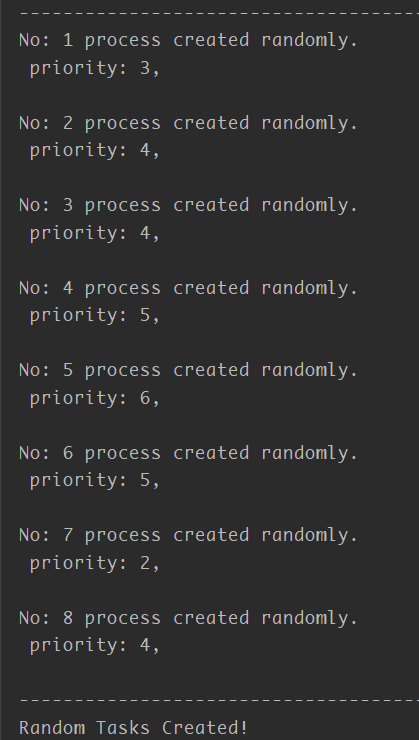


图4.2 随机初始化

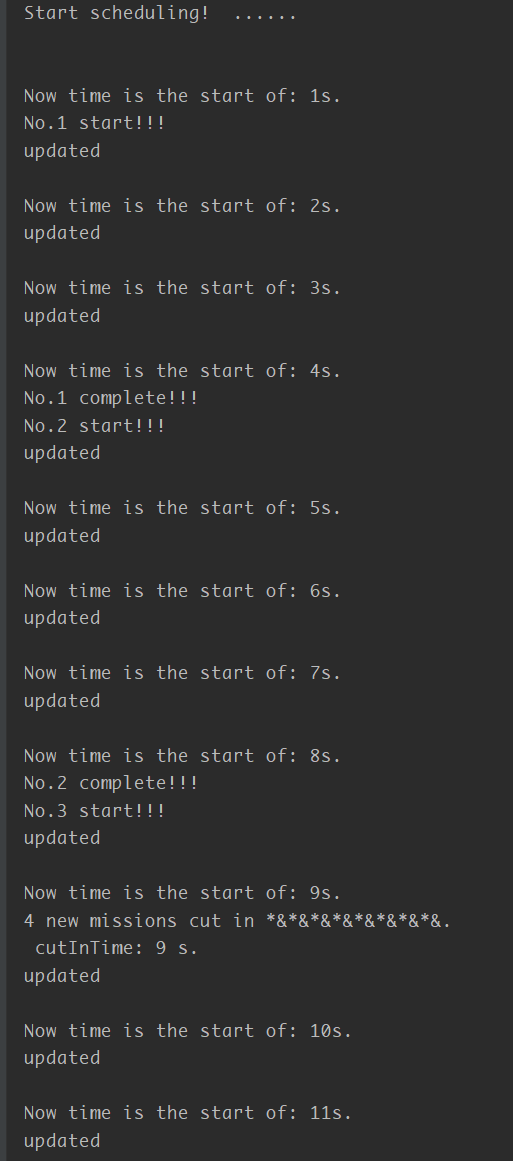
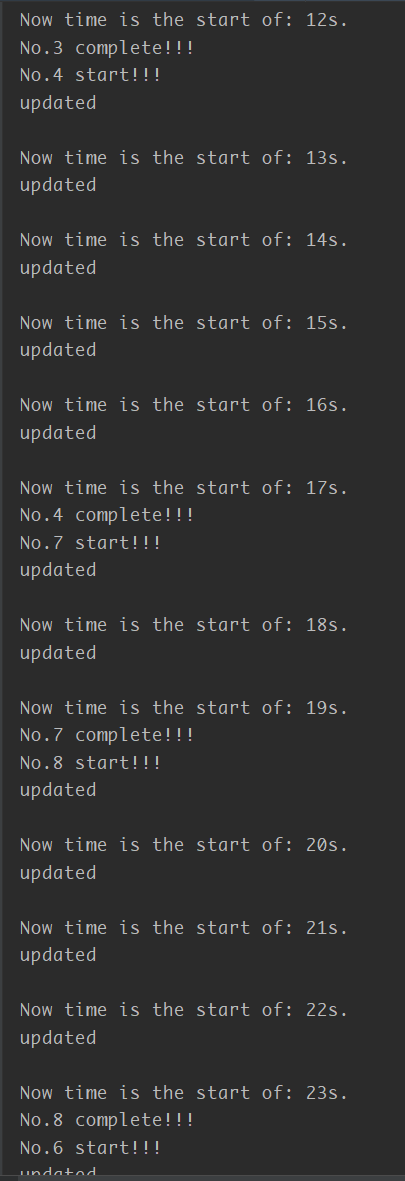
 

图4.3 4.4 时间轴

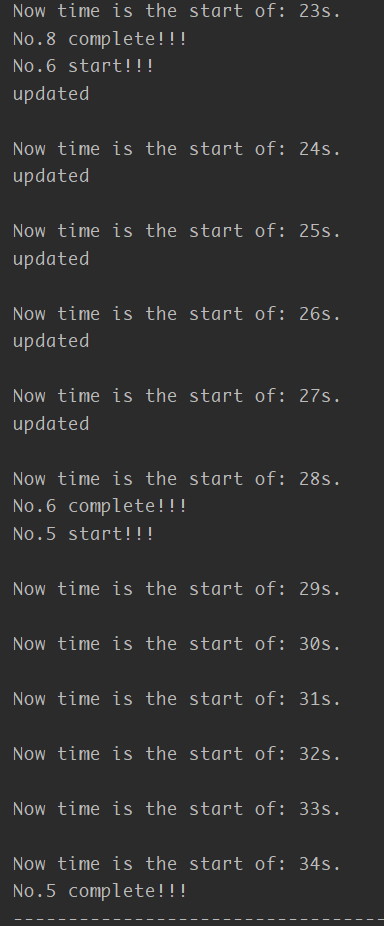


图4.5 时间轴

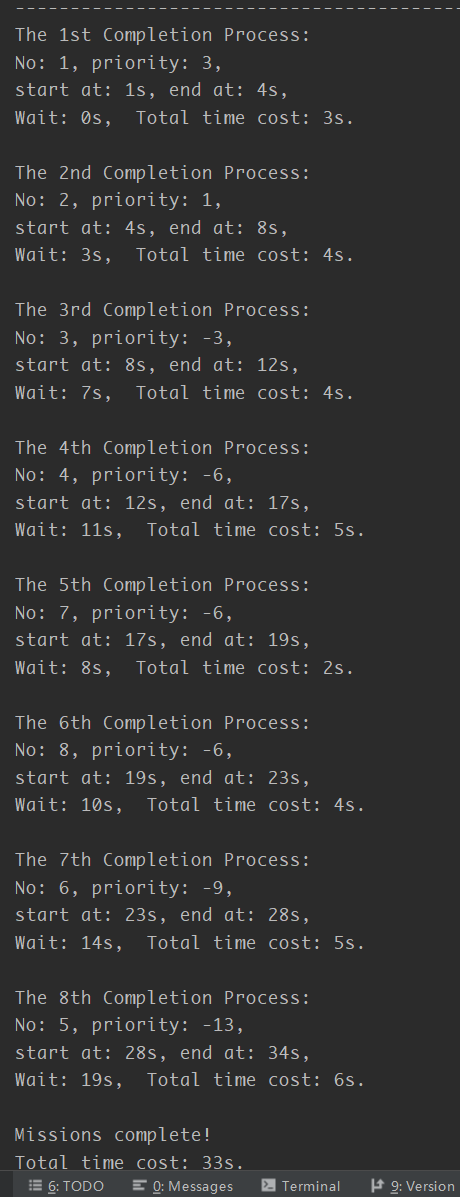


图4.6 运行结果展示

**（三）调试错误分析：**

**问题1.**

写堆的构造函数的时候：

for (size\_t i = 0; i < size; i++){

//v[i] = array[i]; //这样写不行。

v.push\_back(array[i]);

}

可能导致向量v的越界。

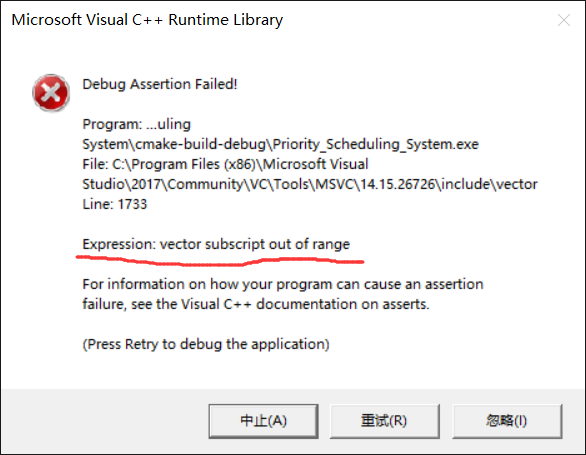


图4.7 bug1的报错

**问题2.**

完成最小堆的算法后，测试的过程中发现将随机生成的测试数组放入堆以后，堆仍旧无序。（改变了顺序但并未从小到大）。

**解决历程：**

1. 首先发现堆中未重载swap函数，可能导致上调下调算法出错。立刻跟进。
2. 排序以后发现，堆中所有元素都变成了同一个。随后发现Process类未重载=运算符，可能导致swap函数不正常执行。立刻跟进=的重载。
3. 还是一个奇怪的无序。

…………

**推倒原来的堆构建方式重来。**

由于Push函数自带了上调函数，具有排序功能，故**不再在堆构造函数中排序。直接采用Push函数进行堆的构建。**

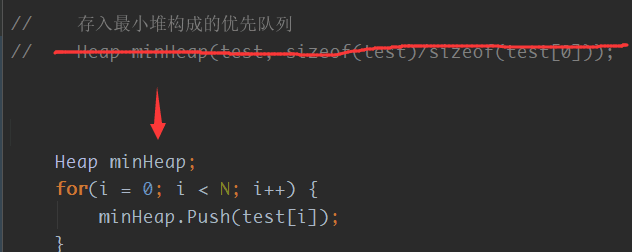


图4.8 bug2的修改(3)

1. 堆仍旧无序。

但是好消息是发现每一次总是固定位置的几个元素不在该在的位置上。

1. 对每一个Down和Up进行打断点检查后，发现Down下调算法中，有一个>写成了<。

果然照着最大堆仿写最小堆容易出事。

问题解决，最小堆正常创建。

后来发现，swap函数不用重构，使用自带的也可以。

**问题3.**

测试进程时发现，没有实现FIFO原则。

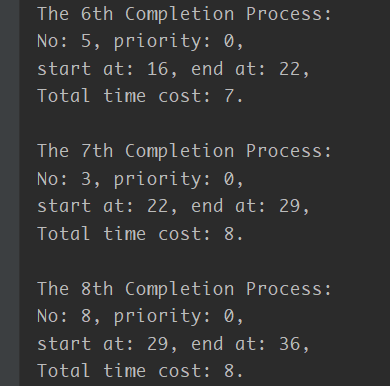


图4.9 bug3 未实现FIFO

（先来的3号任务却在5号任务后执行了，导致3等了很久。）。

**解决历程：**

1. 发现普通的上调和下调算法并不能实现FIFO原则。
2. 于是在下调算法中加入判断，如果p相等，则判断编号（no）的大小，小的优先。

……………………

1. 改完后发现，发现了新的问题：

原来的思路（开始进程后计算优先级的变化）会导致剩余的任务优先级未更新，反被后插入的小优先级先执行。

**更换思路推倒重来。**

思路转变为每个任务**实时更新优先级**。

1. 新编写update函数，每一秒执行一次，实现优先级的实时调度。

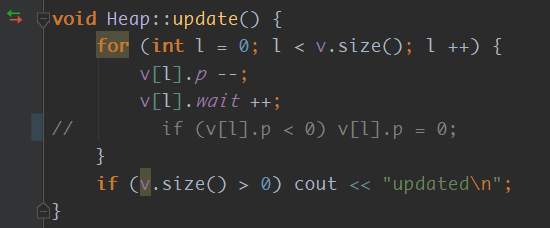


图4.10 bug3的修改

//问题解决。

**问题**4.

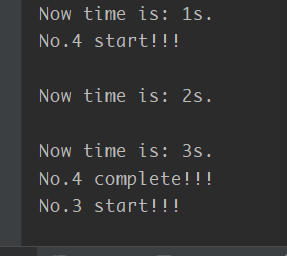
随后进程停滞。

图4.11 bug4 进程停滞

这个情况出现的十分偶然。有20%左右几率出现。

**解决历程：**

1. 经过特殊样例的测试，发现如果所有进程的p值相等，那么在第二个任务开始就会停滞。
2. 猜测: **在某处陷入死循环**。
3. 历尽千辛万苦，使用万能的断点输出cout << "log1\n"; 细致检查，发现：

在解决问题3的过程中，第(2)步的判断有一处没有跳出。屏幕不断地输出log1。

1. 如下加了一行break。

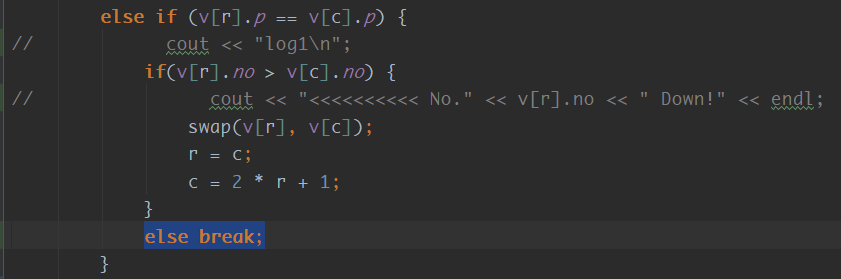


图4.12 bug4的修改

问题解决。

***验收时的部分提问：***

***Q:*** 你是如何实现优先队列的？

***A:*** 通过对每一个任务的优先级编写最小堆，通过上调和下调算法使得这个最小堆在插入和删除时，一直保持根节点优先级最靠前。但是，这样虽然实现了优先队列，但并不能满足FIFO原则，因为优先级一样时，没有对任务进入系统的先后顺序进行判断。

***Q:*** 那你是如何解决FIFO原则问题的？

***A:*** 通过在下调算法中增加了一个判断：优先级一样时，序号小的优先。伴随着堆里任务优先级的实时更新，堆的增删才能在系统中顺利进行。

1. **程序代码**

程序中保留了所有的debug前中后的代码，被优化的部分已被注释，故代码略冗长。

**项目同时分享于：**

<https://github.com/Congb19/Priority_Scheduling_System>

* **Process.h**

|  |
| --- |
| //  // Created by Congb on 2018/12/29.  //  #ifndef PRIORITY\_SCHEDULING\_SYSTEM\_PROCESS\_H  #define PRIORITY\_SCHEDULING\_SYSTEM\_PROCESS\_H  #include <iostream>  using namespace std;  class Process {  public:  Process() {};  Process(int, int);  void set(int, int);  void show();  void display();  int no;//编号  int t;//长度  int p;//优先级  int start;//开始时间  int end;//结束时间  int wait;//等待时间  Process &operator =(const Process &);  private:  };  #endif //PRIORITY\_SCHEDULING\_SYSTEM\_PROCESS\_H |

* **Process.cpp**

|  |
| --- |
| //  // Created by Congb on 2018/12/29.  //  #include "Process.h"  #include <iostream>  using namespace std;  //换方案了。于是这一个构造函数没有用到。  Process::Process(int a, int b):p(b),start(0),end(0),t(b),no(a),wait(0){  // a 编号  // b 优先级  }  void Process::set(int a, int b) {  t=b; p=b; no=a; start=0; end=0; wait=0;  }  void Process::show() {  cout << "No: " << no <<", ";  cout << "priority: "<< p << "," << endl;  cout << "start at: " << start << "s, ";  cout << "end at: " << end << "s, " << endl;  cout << "Wait: " << wait << "s, ";  cout << "Total time cost: " << t << "s." << endl;  cout << endl;  }  void Process::display() {  cout << "No: " << no <<" process created randomly.\n ";  cout << "priority: "<< p << "," << endl;  cout << endl;  }  Process& Process::operator=(const Process &pp) {  if (this!=&pp) {  this->no = pp.no;  this->p = pp.p;  this->t = pp.t;  this->start = pp.start;  this->end = pp.end;  this->wait = pp.wait;  }  return \*this;  // Process a(pp.no, pp.p);  // return a;  } |

* **Heap.h**

|  |
| --- |
| //  // Created by Congb on 2018/12/29.  //  #ifndef PRIORITY\_SCHEDULING\_SYSTEM\_HEAP\_H  #define PRIORITY\_SCHEDULING\_SYSTEM\_HEAP\_H  #include <vector>  #include "Process.h"  #include <iostream>  using namespace std;  class Heap {  public:  Heap();  Heap(const Process\* array, size\_t size);  void Push(Process a);  void Pop();  //一些函数  bool Empty() const {  return v.empty();  }  size\_t Size() const {  return v.size();  }  Process Top() {  return v[0];  }  void show() {  for (int i = 0; i < v.size(); ++i) {  v[i].show();  }  }  void update();  private:  vector<Process> v;  void Down(size\_t r);  void Up(size\_t loc);  };  #endif //PRIORITY\_SCHEDULING\_SYSTEM\_HEAP\_H |

* **Heap.cpp**

|  |
| --- |
| //  // Created by Congb on 2018/12/29.  //  #include <vector>  #include "Heap.h"  #include <iostream>  using namespace std;  //void swap(Process &t1, Process &t2) {  // Process temp = t1;  // t2 = t1;  // t1 = temp;  //}  Heap::Heap() {}  //这个构造函数是原本的堆构造方案。现在用不上它了。安息吧。  Heap::Heap(const Process \*array, size\_t size) {  for (size\_t i = 0; i < size; i++){  //v[i] = array[i]; //这样写不行。  v.push\_back(array[i]);  cout << "\*\*\*\*\*\*\*\*\*\* No." << array[i].no << " cut in!" << endl;  }  v.resize(size);  cout << "v.size= " << v.size() << endl;  // CreateHeap  if (v.size() <= 1) return;  for (int t = (v.size() - 1 - 1) / 2; t >= 0; t--) { // t=执行下调的次数  Down(t);  }  }  void Heap::Push(Process a) {  v.push\_back(a);  // cout << "\*\*\*\*\*\*\*\*\*\* No." << a.no << " cut in!" << endl;  if (v.size() < 2) return; // 不需要上调  Up(v.size() - 1); // 秘技！无限上调  }  void Heap::Pop() {  if (v.empty()) return;  size\_t last = v.size() - 1;  if (v.size() > 1) swap(v[0], v[last]);  // cout << "No." << v[last].no << " poped from heap.\n";  v.pop\_back();  if (v.size() > 1) Down(0); // 秘技！无限下调  // cout << "now top is No." << v[0].no << " \n";  }  void Heap::Down(size\_t r) {  size\_t c = r \* 2 + 1;  while (c < v.size()) {  if (c + 1 < v.size()) { //有两个孩子  if (v[c].p > v[c+1].p) c++; //左 > 右  else if (v[c].p == v[c+1].p) { //左 == 右  if(v[c].no > v[c+1].no) c++;  }  }  if(v[r].p > v[c].p) {  // cout << "<<<<<<<<<< No." << v[r].no << " Down!" << endl;  swap(v[r], v[c]);  r = c;  c = 2 \* r + 1;  }  else if (v[r].p == v[c].p) {  // cout << "log1\n";  if(v[r].no > v[c].no) {  // cout << "<<<<<<<<<< No." << v[r].no << " Down!" << endl;  swap(v[r], v[c]);  r = c;  c = 2 \* r + 1;  }  else break;  }  else {  break;  }  }  }  void Heap::Up(size\_t loc) {  size\_t r = (loc - 1) / 2;  while (loc != 0) {  if (v[loc].p < v[r].p) {  // cout << ">>>>>>>>>> No." << v[loc].no << " Up!" << endl;  swap(v[loc], v[r]);  loc = r;  r = (loc - 1) / 2;  }  // else if (v[loc].p == v[r].p) {  // if (v[loc].no < v[r].no) {  // cout << ">>>>>>>>>> No." << v[loc].no << " Up!" << endl;  // swap(v[loc], v[r]);  // loc = r;  // r = (loc - 1) / 2;  // }  // }  else return;  }  }  void Heap::update() {  for (int l = 0; l < v.size(); l ++) {  v[l].p --;  v[l].wait ++;  // if (v[l].p < 0) v[l].p = 0;  }  if (v.size() > 0) cout << "updated\n";  } |

* **main.cpp**

|  |
| --- |
| //  // Created by Congb on 2018/12/29.  // Heap: 最小堆实现优先队列  // Process: 作业进程类  //  #include <iostream>  #include <cstdlib>  #include <cstdio>  #include <vector>  #include <windows.h>  #include <time.h>  #include "Process.h"  #include "Heap.h"  #define N 4 //每组任务个数  #define W 2 //剩余W的时候插入新任务  //W < N.  using namespace std;  int main() {  cout << endl;  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;  cout << "\*------Priority Scheduling System------\*" << endl;  cout << "\*----------Powered by Congb19----------\*" << endl;  cout << "\*--------------2018.12.29--------------\*" << endl;  cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;  cout << "Press \"Enter\" to start." << endl;  srand(time(NULL));  int mainTime, cutInTime = 0, remainTasks = N + N, comTasks = 0; //时间轴、剩余任务数量  Process test[N + N], disp[N + N]; //将要被执行的进程组  int pt, i, j; //存放随机优先数  bool cpu = false; //是否有任务在执行  int cpuTime = 0; //当前任务已进行的时间  Process inCpu; //在cpu中执行的任务  int cutIn = 0; //是否已经插入剩余任务  getchar();  cout << "-----------------------------------------------------" << endl;  // 随机初始化一堆任务  for(i = 0; i < N; i ++) {  pt = 2 + rand() % 5;  test[i].set(i+1, pt);  test[i].display();  }  // 再随机初始化一堆 待插入的任务  for(i = N; i < N + N; i ++) {  pt = 2 + rand() % 5;  test[i].set(i + 1, pt);  test[i].display();  }  cout << "-----------------------------------------------------" << endl;  // 换方案了。安息吧  // Heap minHeap(test, sizeof(test)/sizeof(test[0]));  // 把初始任务队列存入最小堆构成的优先队列  Heap minHeap;  for(i = 0; i < N; i ++) {  minHeap.Push(test[i]);  }  // 。测试用。  // 直接查看向量  // minHeap.show();  //  // 按顺序查看堆  // for(i = 0; i < N; i++) {  // minHeap.Top().show();  // minHeap.Pop();  // }  cout << "Random Tasks Created!" << endl;  // 开始进程  int k = 0;  cout << "Start scheduling! ......\n" << endl;  for (mainTime = 1; remainTasks > 0; mainTime ++) {  cout << "\nNow time is the start of: " << mainTime << "s." << endl;  // 实时更新优先级。  // 这一个for已经被优化了。  // for (int l = 0; l < minHeap.Size(); ++l) {  // test[l].p --;  // if (test[l].p < 0) test[l].p = 0;  // if (l == N - 1 && cutIn == 0) break;  // }  // 在当前队列中剩余不到W个任务且没有插入过的话，插入剩余的任务。(W < N)  if (remainTasks == N + W && cutIn == 0) {  cout << N << " new missions cut in \*&\*&\*&\*&\*&\*&\*&\*&. \n";  for(i = N; i < N + N; i++) {  minHeap.Push(test[i]);  // cout << "No: " << i + 1 << " cut in. \n";  }  cutIn ++;  cutInTime = mainTime;  cout << " cutInTime: " << cutInTime << " s. \n";  }  Start:  if (!cpu) { //如果当前没有任务在跑  cpu = true;  inCpu = minHeap.Top();  cout << "No." << inCpu.no << " start!!! \n";  //// 旧的方案（进来以后更新优先级）。已被证明存在bug，已优化。  // if (inCpu.no < N + 1) {  // inCpu.wait = (mainTime - 1);  //// inCpu.p -= inCpu.wait;  // }  // else {  // inCpu.wait = (mainTime - cutInTime + 1);  //// inCpu.p -= inCpu.wait;  // }  //// if(inCpu.p < 0) inCpu.p = 0;  disp[k] = inCpu;  disp[k].start = mainTime;  minHeap.Pop();  } else { //如果当前有任务在跑  cpuTime++;  if (cpuTime >= inCpu.t) { //当前的运算结束  cout << "No." << inCpu.no <<" complete!!! \n";  cpu = false;  remainTasks --;  comTasks ++;  disp[k].end = mainTime;  k ++;  cpuTime = 0;  if (remainTasks > 0) goto Start; //如果还有任务，需要再判断一遍这一秒。  }  }  minHeap.update();  Sleep(1000);  }  // 最后展示所有任务的运行情况  cout << "-----------------------------------------------------" << endl;  for (j = 0; j < N + N; j ++) {  if (j == 0) cout << "The 1st" << " Completion Process: " << endl;  else if (j == 1) cout << "The 2nd" << " Completion Process: " << endl;  else if (j == 2) cout << "The 3rd" << " Completion Process: " << endl;  else cout << "The " << j + 1 << "th Completion Process: " << endl;  disp[j].show();  }  cout << "Missions complete! \nTotal time cost: " << mainTime - 2 << "s." << endl;  return 0;  } |