**目录**

[1 设计原理 1](#_Toc60832804)

[1.1 进程调度 1](#_Toc60832805)

[1.1.1 进程调度的概念 1](#_Toc60832806)

[1.1.1 进程调度算法 1](#_Toc60832807)

[2 需求分析 1](#_Toc60832808)

[3 总体设计 2](#_Toc60832809)

[3.1 设计思路 2](#_Toc60832810)

[3.2 实现功能列表 2](#_Toc60832811)

[4 算法设计分析 2](#_Toc60832812)

[4.1 短作业优先算法](#_Toc60832813) 2

[4.2 先来先服务算法 6](#_Toc60832814)

[4.3 时间片轮转算法 7](#_Toc60832815)

[4.4 静态优先权优先算法 10](#_Toc60832816)

[4.5 最高响应比算法 12](#_Toc60832817)

[5 程序总流程图 16](#_Toc60832818)

[6 函数功能简介 16](#_Toc60832819)

[7 运行结果分析](#_Toc60832821) 19

[7.1 编译器和运行环境](#_Toc60832822) 19

[7.2 编译运行](#_Toc60832823) 19

[7.3 界面设计](#_Toc60832824) 20

[7.4 功能演示](#_Toc60832825) 20

[7.5 分析总结 2](#_Toc60832826)8

[8 参考资料](#_Toc60832827) 29

[附录：](#_Toc60832828)**[源代码](#_Toc60832828)**30

|  |
| --- |
| **通用处理机调度演示程序** 设计原理1.1 进程调度1.1.1 进程调度的概念无论是在批处理系统还是分时系统中，用户进程数一般都多于处理机数、这将导致它们互相争夺处理机。另外，系统进程也同样需要使用处理机。这就要求进程调度程序按一定的策略，动态地把处理机分配给处于就绪队列中的某一个进程，以使之执行。1.1.2 进程调度算法  1. 先来先服务调度算法（First Come First Serve，FCFS）：按照进程到达的先后顺序进行调度，「先到的进程就先被调度」，也就是说，等待时间越久的越优先得到服务。 2. 最短作业/进程优先调度算法（Shortest Job First，SJF）：每次调度时选择当前已到达且运行时间最短的进程 3. 高响应比优先算法（Highest Response Ratio Next，HRRN）：只有当前运行的进程主动放弃 CPU 时（正常/异常完成，或主动阻塞），才需要进行调度，「调度时计算所有就绪进程的响应比，为响应比最高的进程分配 CPU」。响应比 = (进程的等待时间 + 进程需要的运行时间) / 进程需要的运行时间 4. 静态优先级调度算法（Highest Priority First，HPF）就是「从就绪队列中选择最高优先级的进程进行运行」。创建进程时候，就预先规定优先级，并且整个运行过程中该进程的优先级都不会发生变化。一般来说，内核进程的优先级都是高于用户进程的。非抢占式：当就绪队列中出现优先级高的进程，则运行完当前进程后，再选择该优先级高的进程。  轮转调度算法（Round Robin，RR）也称时间片调度算法：调度程序每次把 CPU 分配给就绪队列首进程使用规定的时间间隔，称为时间片，通常为 10ms ~ 200ms，「就绪队列中的每个进程轮流地运行一个时间片，当时间片耗尽时就强迫当前运行进程让出 CPU 资源，转而排到就绪队列尾部，等待下一轮调度」。所以，一个进程一般都需要多次轮转才能完成。需求分析  1. 进程调度算法包括：时间片轮转算法、先来先服务算法、短作业优先算法、静态优先权优先调度算法、高响应比调度算法。 2. 每一个进程有一个 PCB，其内容可以根据具体情况设定。 3. 进程数、进入内存时间、要求服务时间、作业大小、优先级等均可以在界面上设定。 4. 可读取样例数据（要求存放在外部文件中）进行进程数、进入内存时间、时间片长度、作业大小、进程优先级的初始化。 5. 可以在运行中显示各进程的状态：就绪、执行（由于不要求设置互斥资源与进程间的同步关系， 故只有两种状态） 6. 有性能比较功能，可比较同一组数据在不同调度算法下的平均周转时间。  总体设计3.1 设计思路 功能实现是程序的核心，根据功能实现的数量来设计菜单中选项的多少。用户只用通过简单地键盘输入来选择算法和设置参数，就可以了解所有算法的实现过程。显示菜单，菜单上列举五个算法，选择相应的算法之后，用户决定手动输入进程的参数或者从文件中输入，调用相应的算法之后显示所有进程的完成时间和周转时间，计算并显示平均周转时间：  显示结果  用户选择文本输入或者手动输入进程的参数  显示菜单  调用相应的算法  用户选择算法  显示菜单  图3.1 1设计思路 3.2 实现功能列表  * 正常进入和退出程序：程序能正常运行，对用户的非法输入有一定的容错性，能提醒用户重新输入，避免程序意外中断。 * 用户输入：进行算法选择，输入方式选择，进程调度算法所需的初始化参数。 * 菜单显示：五种算法的列举，退出程序。 * 结果输出：每个算法都可以输出进程的到达时间，完成时间，周转时间和所有进程完成的平均周转时间 * 有性能比较功能，可以同时完成多个算法，比较每个算法下同一组数据的平均周转时间  算法设计分析4.1 短作业优先算法 短作业优先算法的思想是运行服务时间最短的进程，但不能只根据服务时间判断下一个要运行的进程。需要根据当前时间来寻找下一个需要运行的进程。另外在当前时间前、后到达的进程的开始服务时间不同。  如果是在当前时间之前就已经到达且未能运行的进程，不在乎到达时间的先后，只比较服务时间，选择一服务时间最短的进程运行，开始时间等于当前时间。  如果是在当前时间之后到达的进程，比较到达时间，选择最先到达的进程，并在同一时间最先到达的进程中选出服务时间最短的进程，开始服务时间等于进程到达时间。  未完成进程  输入进程  选择下一个运行的进程时只考虑服务时间长短  选择最先到达且服务时间最短的进程  当前时间  开始服务时间等于到达时间  开始服务时间等于当前时间  4.1.1 短作业优先算法流程图  操作系统  算法中有两重循环，第一重循环是运行的进程个数，有多少个进程就循环多少次，目的在于每个进程完成后，改变运行完成的进程的参数。  第二重循环有两个，第一个循环的作用是判断是否有在当前时间之前就到达的进程，如果没有，则找到进程中到达时间最早的进程；第二个循环的作用是找到在当前时间之前到达的进程中作业时间最短的进程，或者找到在当前时间之后，最先到达的进程中作业时间最短的进程 4.1.2 算法实现 struct Process\_struct {  int Number; //进程编号  char Name[MaxNum]; //进程名称  int ArrivalTime; //到达时间  int ServiceTime; //开始运行时间  int FinishTime; //运行结束时间  int WholeTime; //运行时间  int run\_flag; //调度标志  int order; //运行次序  double WeightWholeTime; //周转时间  double AverageWT\_FCFS, AverageWT\_SJF; //平均周转时间  double AverageWWT\_FCFS, AverageWWT\_SJF; //平均带权周转时间  }Process[MaxNum];  int SJF() { //短作业优先算法  int temp\_time = 0; //当前时间  int i = 0, j;  int number\_schedul=0, temp\_counter; //进程编号，当前已执行进程个数  float run\_time;  temp\_counter = 0;  while (temp\_counter < N)  {  int matime = INT\_MAX;//设置该临时变量的值为无穷大，用于比较寻找最先到达的进程  for (j = 0; j < N; j++)  {  //如果到达时间小于当前时间，且进程未完成  if ((Process[j].ArrivalTime <= temp\_time) && (!Process[j].run\_flag))  {//如果找到一个到达时间小于当前时间的进程，则选择该进程，不一定运行，因为有可能存在同时到达但比这个进程作业时间短的进程  run\_time = Process[j].WholeTime;  number\_schedul = j;  Process[number\_schedul].ServiceTime = temp\_time; //开始服务时间是当前时间  break;  }  else {  //如果没有在当前时刻之前到达的进程，则在就绪队列中找到达时间最早的进程  if (Process[j].ArrivalTime < matime && (!Process[j].run\_flag))  {  matime = Process[j].ArrivalTime;  run\_time = Process[j].WholeTime;  number\_schedul = j;  Process[number\_schedul].ServiceTime = Process[j].ArrivalTime; //开始服务时间是到达时间  }  }  }  for (j = 0; j < N; j++)  {  //如果到达时间小于当前时间，且进程未完成  if ((Process[j].ArrivalTime <= temp\_time) && (!Process[j].run\_flag))  //若果有比准备运行的进程运行时间更短的进程，则选择这个运行时间更短进程运行  {  if (Process[j].WholeTime < run\_time)  {  run\_time = Process[j].WholeTime;  number\_schedul = j;  Process[number\_schedul].ServiceTime = temp\_time; //开始服务时间是当前时间  }  }  else {//在同时到达的进程中找到作业时间最短的一个  if (Process[j].ArrivalTime == matime && (!Process[j].run\_flag)&& Process[j].WholeTime < run\_time)  {  matime = Process[j].ArrivalTime;  run\_time = Process[j].WholeTime;  number\_schedul = j;  Process[number\_schedul].ServiceTime = Process[j].ArrivalTime; //开始服务时间是到达时间  }  }  }  //对找到的下一个被调度的进程求相应的参数    Process[number\_schedul].FinishTime = Process[number\_schedul].ServiceTime + Process[number\_schedul].WholeTime;  Process[number\_schedul].run\_flag = 1;  temp\_time = Process[number\_schedul].FinishTime;  Process[number\_schedul].order = temp\_counter;  temp\_counter++;  }return 0;  } 4.2 先来先服务算法 先来先服务算法就是根据进程的到达时间按顺序执行进程，那么首先要做的就是找到就绪队列中未完成的且到达时间最早的进程，改变进程状态为FINISH。因为用到的是链表，所以每次寻找下一个要运行的进程都需要遍历一遍链表  2-先来先服务  2-2先来先服务 4.2.1 算法实现 typedef struct PCB //定义进程控制块  {  char ID[3]; //进程号  char name[10]; //进程名  char state; //运行状态  int arrivetime; //到达时间  int starttime; //进程开始时间  int finishtime; //进程结束时间  int servicetime; //服务时间  float turnaroundtime;//周转时间  float weightedturnaroundtime;//带权周转时间  struct PCB\* next; //指向下个进程  }pcb;  void fcfs() //找先到先服务  {  int i, j;  for (i = 0; i < n; i++) //遍历链表找到服务顺序  {  p = head1;  q = p; //标记当前的进程  for (j = 0; j < n; j++)  {  if (p && p->state == 'W')  {min = p; break;}  p = p->next;  }  for (j=0;j < n;j++)  {  if(p->next) p=p->next;  if ((min->arrivetime >= p->arrivetime)&& min->state == 'W'&& p->state == 'W')  //F表示已经完成Finish状态，W-wait,R-run  {  min = p; //找到最先到达的进程  }  }  run\_fcfs(min);  }  printf("average\_turn\_round\_time=%f\n", avg\_around\_time / n);  }  以上代码片段中的min用以找到最小到达时间的进程，min指向的必须是未完成的进程。  找到下一个需要执行的进程后，根据当前时间和进程的到达时间来判断进程的开始服务时间，同样的，在当前时间时间之前到达的进程开始服务时间是当前时间，在当前时间之后到达的进程开始时间是进程的到达时间  输入进程  未完成进程  开始服务时间等于到达时间  开始服务时间等于当前时间  长短  当前时间  进程的周转周期=进程完成时间-到达时间  平均周转时间= 4.3 时间片轮转算法 需要两个链表，一个就绪队列的链表，一个完成队列的链表。  当要运行一个进程的时候，从就绪队列的首部取出进程（进程在就绪队列中删除）。  若在给定的时间片内，该进程未运行完成，改变进程的参数将进程再次加入到就绪队列；若进程在时间片内完成，把进程的状态改为F(finished)，并把进程加到完成链表中。  在这个算法中进程的开始服务时间和完成时间有可能是变化的。当进程未能在分配的时间片内完成，则返回就绪队列，当下次再运行的时候，进程的开始时间和完成时间会被刷新。  计算进程的周转时间需要得到进程真正完成的时间-进程的到达时间  IMG_256 4.3.1 时间片轮转算法流程图 时间片轮转算法 4.3.2 时间片轮转算法实现 void RoundRun() /\*时间片轮转调度算法\*/  {  int flag = 1,flag2=1;  int time = 0; //当前时间  float avf\_around\_t = 0;  GetFirst();  while (run1 != NULL)  {  Output();  while (flag)  {  if (time < run1->arrivetime)  run1->starttime = run1->arrivetime;  else  run1->starttime = time;  run1->cputime++;  run1->needtime--;  run1->count++;  if (run1->needtime == 0)  {  run1->state = 'F';  run1->finishtime = run1->starttime + run1->count;//记录每个进程的实际结束时间  InsertFinish(run1); //在分配的时间片内运行完成  flag = 0;  }  else if (run1->count == run1->round) //时间片用完  {  run1->state = 'W';  run1->finishtime = run1->starttime + run1->round;  run1->count = 0;  InsertTime(run1); //进程再次回到就绪队列  flag = 0;  }  }  if(run1->state=='F')  {  avg\_around\_time += run1->finishtime - run1->arrivetime;  }  time = run1->finishtime;  flag = 1; //用于判断一个进程是否完成  GetFirst();  }  Output();  printf("average\_turn\_round\_time=%f\n", avg\_around\_time / num);  } 4.4 静态优先权优先算法 静态优先级算法就是运行就绪队列中优先级最高的进程，先到达的进程先运行。若在运行一个进程的时候到达一个优先级更高优先级的进程，cpu不会被抢占，等运行的进程完成，才会为未完成的进程中优先级最高的进程分配cpu。  实现静态优先级的思路是在当前时间之前到达的进程只考虑优先级，选择优先级最高的进程运行；在当前时间之后到达的进程，先运行最早到达的进程中优先级最高的  未完成进程  输入进程  选择下一个运行的进程时只考虑优先级  选择最先到达且优先级最大的进程  当前时间 4.4.1 静态优先权优先算法流程图 静态优先级 4.4.2 静态优先权优先算法实现 void run() { //静态优先级算法执行。  struct PCB1\* temp; //临时节点用来存储调入内存节点  struct PCB1\* runPro; //用来接收执行的节点  int i; //循环初始条件,flag标志第一个运行的进程参数的初始化  int at=0, srt, finishtime = 0; //当前时间，开始执行时间，运行时间。  int turnOverTime; //周转时间  float avg\_around\_time = 0;  double ptot; //带权周转时间  if (head == NULL) {  printf("No processes found.\n");  }  else {  while (head != NULL) { //将所有进程都放到就绪队列中去  temp = callMemory(); //返回当前最先进入内存的进程  readyList(temp); //初始纳入就绪列表  release(temp); //释放进程  }//循环完成之后，就绪队列中的进程按照进入内存的时间由小到大排序  for (i = 0; i < ProcessAmount; i++) { //循环进程的数量次  runPro = runTheProcess(timesetout(at)); //1.返回就绪队列中当前最先进入内存且在同时间到达的进程中优先权最高的进程  //2.返回就绪队列中优先级最高的进程  if (runPro->enterMemoryTime > at) {  //初始情况下求开始执行时间，完成时间，周转时间，带权周转时间  srt = runPro->enterMemoryTime; //开始时间等于到达时间  }  else { //初始之后，计算各时间及周转  srt = at;//开始时间等于当前时间  }  finishtime = srt + runPro->serviceTime;  turnOverTime = finishtime - runPro->enterMemoryTime;  avg\_around\_time += turnOverTime;  ptot = turnOverTime \* 1.0 / runPro->serviceTime;  at = finishtime;  releaseR(runPro);  printf("\nReady process list: ");  ListAllPCB(readyHead);  }  }  printf("average\_turn\_round\_time=%f\n", avg\_around\_time / ProcessAmount);  } 4.5 最高响应比调度算法 高响应比算法的执行规律是先执行第一个提交的作业，然后其余的作业再用响应比来判断执行顺序，先要做的是找到第一个到达的进程，再算出其余进程的相应比。  由于未完成的进程的等待时间是变化的，所以未完成进程的响应比也是变化的。每执完成一个进程就需要重新计算余下未完成进程的响应比，并选择响应比最大的进程运行。 响应比=作业周转时间/作业处理时间=（作业处理时间+作业等待时间）/作业处理时间=1+（作业等待时间/作业处理时间） 等待时间=最后一个的提交时间-该作业到达的时间 作业执行规则，响应比高的先执行 周转时间=完成时间-提交时间  先将所有的进程按到达时间排序，找到最先到达的进程并运行，改变进程的参数（设置开始服务时间，完成时间，周转时间，进程状态）。  计算余下进程的响应比，选择响应比最大的进程并运行，改变进程参数。重复这一个步骤，直到所有进程都完成 最高响应比4.5.1 最高响应比调度算法流程图4.5.2 最高响应比调度算法实现 void calculate(struct zgxyb\* p)  {  for (int n = 1; n < N; n++) //计算各个进程的相应比  {  if (p[n].stats == 'W')  {  p[n].zztime = timecounter5 - p[n].arrivetime; //进程的等待时间=当前时间-到达时间  p[n].dqzztime = 1 + p[n].zztime / p[n].servicetime;  }  }  }  void ZGXYB(struct zgxyb\* p, int N)  {  int m, i, n, follow, k;  int schedul=0;  float max=0;  struct zgxyb temp;  sort(p,N); //按到达时间对进程进行排序  for(i=0;i<N;i++) //运行N个进程  {  if (timecounter5 < p[schedul].arrivetime)// schedul初始值=0，进程按时间顺序到达排序后执行第一进程  p[schedul].starttime = p[schedul].arrivetime;  else  p[schedul].starttime = timecounter5;  p[schedul].finishtime = p[schedul].starttime + p[schedul].servicetime;  p[schedul].around\_time = p[schedul].finishtime - p[schedul].arrivetime;  p[schedul].stats = 'F';  timecounter5 = p[schedul].finishtime;  Print(p);  calculate(p); //计算各进程相应比  for (m = 1; m < N; m++) //选择一个未完成进程  {  if (p[m].stats == 'W')  {  max = p[m].dqzztime;  schedul = m;  break;  }  }  for (int k = 1; k < N; k++) //选出响应比最高的进程  {  if (max <= p[k].dqzztime && p[k].stats == 'W')  {  max = p[k].dqzztime;  schedul = k;  }  }  }  } 4.6 实现从文件输入进程的数据 以只读的方式打开一个文件  printf("Enter file path:");  scanf("%s", filePath);  fp = fopen(filePath, "r");  获取文件行数，我们的文件为以下内容，第一行是无关的数据，我们需要的是从第二行开始的进程数据    图4.6 输入进程数据的文件内容  while (i < times - 1) { //循环次数=文本行数  if (i == 0) { //这是用来接收文本格式第一行的汉字。  fgets(temp, sizeof(temp), fp);//把读取的第一行的内容放在temp数组中，指针已指向下一行  }  else { //以下用来接收进程数据  fscanf(fp, "%s", name);  fscanf(fp, "%f", &emt);  fscanf(fp, "%f", &st);  fscanf(fp, "%d", &pri);  fscanf(fp, "%f", &slice);  }i++;  }  通过上面的循环来获取每一行的数据，每获取一行数据就将数据添加到进程的相应存储结构中  int timeOfData(FILE\* f) { //求取从文本中获取数据的进程数量  char temp[20];  int i = 0;  int flag = 0;  if (f == NULL) {  printf("File not found.\n");  exit(0);  }  else {  while (!feof(f))  {  flag = fgetc(f);  if (flag == '\n')  i++;  }  }  rewind(f); //将文件指针从新指向文件的开头  return i;  }  C 库函数 **int fgetc(FILE \*stream)** 从指定的流 stream 获取下一个字符（一个无符号字符），并把位置标识符往前移动。  rewind () 功 能: 将文件内部的位置 指针 重新指向一个流（ 数据流 /文件）的开头. 注意：不是 文件指针 而是文件内部的位置指针，随着对文件的读写文件的位置指针（指向当前读写字节）向后移动 程序总流程图 总流程  图 5 总流程图 函数功能简介            运行结果分析7.1 编译器和运行环境  * 编译环境：vs2019 * 运行环境：Linux Ubuntu 18.0.4 * 远程登录工具：putty  7.2 编译运行 用putty与Ubuntu建立SSH连接，实现用putty登陆虚拟机。在windows的cmd中用scp命令将vs2019下编译的文件复制的Linux机器下的/usr/local/prcess.c文件中:  scp C:\Users\Lenovo\source\repos\通用处理机调度\通用处理机调度\processdd.c jqq@192.168.56.13:\usr\local\process.c  Jqq是linux虚拟机的主机名，192.168.56.13是虚拟机ip    process.c文件必须有写权限，外部机器才可以操作这个文件。为了方便，我们修改process.c文件的权限为777：chmod 777 process.c  在 Linux环境下，输入进入项目所在目录/usr/local/，执行命令sudo gcc process.c -o process.exe，自动完成对项目的编译。再查看该目录就可以看到process.exe可执行文件：    图7.2 1编译过程  在/usr/local目录下输入./process.exe既可以运行 7.3 界面设计   图 7.3 1 主界面 7.4 功能演示  1. 文件输入同时运行两个算法：     图7.4 1文件输入及显示  文件内容如下     1. 手动输入：     图 7.4 .2 输出内容显示   1. 五种算法的计算过程演示及验证：           图 7.4 .3 SJF      ……    图 7.4 .4 FCFS          图7.4.5 RR   0时刻(P1):只有P1到达，P1上处理机。7时刻(P2、P3，P4):P1运行完成主动放弃处理机，其余进程都已到达，P3优先级最高，P3上处理机。8时刻(P2、P4):P3完成，P2P4优先级相同，由于P2先到达，因此P2优先上处理机 平均周转时间=（7+4+10+11）/4=32/4=8    图 7.4.6 静态优先级优先    图 7.4.7 最高响应比    先执行的是第一个提交作业，然后其余的作业再用响应比来判断执行顺序  先执行p1 ：    设响应比为R  此时 R(p2)=1+((12.0-10.2)/1.0)=2.8  R(p3)=1+((12.0-10.4)/0.5)=4.2  R(p4)=1+((12.0-10.5)/0.3)=6  所以 执行p4：    设响应比为R  此时 R(p2)=1+((12.3-10.2)/1.0)=3.1  R(p3)=1+((12.3-10.4)/0.5)=4.8  所以 再执行p3：    因此最后执行p2：    此算法作业的平均周转时间为：(2.0+3.6+2.4+1.8)/4=2.45  **手动计算和程序计算结果是一致的**，可知程序编写是正确的。 7.5 分析总结  * 问题解决  1. 编码方式：在VS 2019中运行项目成功，中文显示正常，而拷贝到Ubuntu下运行时，中文全是乱码，经查阅才得知Ubuntu默认编码都是UTF-8，与VS默认不同，所以在外部编写好代码后需要转换为UTF-8才可以。 2. 编译环境：在我的Ubuntu下没有集成的编译器，只能用文本编辑器，用起来特别麻烦。所以我在windows下的VS2019中编辑代码，通过scp命令把源文件拷贝到linux虚拟机中编译运行。  * 优点  1. 可以同时调用多个算法，对于同用一组数据就可以比较直观的比较调用每个算法的平均周转时间 2. 算法比较完善，对于第一个需要运行的程序可以准确的判别出它的开始服务时间  * 缺点  1. 人性化设计不足：同时运行多个算法时，每个算法下数据都需要重新输入 2. 因为控制台下局限性，界面设计比较简陋。  参考资料 [1]郁红英、王磊、武磊、李春强．《计算机操作系统》（第3版），清华大学出版社， 2018． |
| 附录：源代码 **Process.c**  #define MaxNum 100  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include<limits.h>  //1.短作业优先算法  struct Process\_struct {  int Number; //进程编号  char Name[MaxNum]; //进程名称  int ArrivalTime; //到达时间  int ServiceTime; //开始运行时间  int FinishTime; //运行结束时间  int WholeTime; //运行时间  int run\_flag; //调度标志  int order; //运行次序  double WeightWholeTime; //周转时间  double AverageWT\_FCFS, AverageWT\_SJF; //平均周转时间  double AverageWWT\_FCFS, AverageWWT\_SJF; //平均带权周转时间  }Process[MaxNum];  //找到当前到达的运行时间最短的进程  int N; //实际进程个数  int SJF() { //短作业优先算法  int temp\_time = 0; //当期那时间  int i = 0, j;  int number\_schedul=0, temp\_counter; //进程编号，当前已执行进程个数  float run\_time= INT\_MAX;  temp\_counter = 0;  while (temp\_counter < N)  {  int matime = INT\_MAX;//设置该临时变量的值为无穷大，用于比较寻找最先到达的进程  for (j = 0; j < N; j++)  {  //如果到达时间小于当前时间，且进程未完成  if ((Process[j].ArrivalTime <= temp\_time) && (!Process[j].run\_flag))  {//如果找到一个到达时间小于当前时间的进程，则选择该进程，不一定运行，因为有可能存在同时到达但比这个进程作业时间短的进程  run\_time = Process[j].WholeTime;  number\_schedul = j;  Process[number\_schedul].ServiceTime = temp\_time; //开始服务时间是当前时间  break; //只需要找一次就行  }  else {  //如果没有在当前时刻之前到达的进程，则在就绪队列中找到达时间最早的进程  if (Process[j].ArrivalTime < matime && (!Process[j].run\_flag))  {  matime = Process[j].ArrivalTime;  run\_time = Process[j].WholeTime;  number\_schedul = j;  Process[number\_schedul].ServiceTime = Process[j].ArrivalTime; //开始服务时间是到达时间  }  }  }  for (j = 0; j < N; j++)  {  //如果到达时间小于当前时间，且进程未完成  if ((Process[j].ArrivalTime <= temp\_time) && (!Process[j].run\_flag))  //若果有比准备运行的进程运行时间更短的进程，则选择这个运行时间更短进程运行  {  if (Process[j].WholeTime < run\_time)  {  run\_time = Process[j].WholeTime;  number\_schedul = j;  Process[number\_schedul].ServiceTime = temp\_time; //开始服务时间是当前时间  }  }  else {//在同时到达的进程中找到作业时间最短的一个  if (Process[j].ArrivalTime == matime && (!Process[j].run\_flag)&& Process[j].WholeTime < run\_time)  {  matime = Process[j].ArrivalTime;  run\_time = Process[j].WholeTime;  number\_schedul = j;  Process[number\_schedul].ServiceTime = Process[j].ArrivalTime; //开始服务时间是到达时间  }  }  }    //对找到的下一个被调度的进程求相应的参数    Process[number\_schedul].FinishTime = Process[number\_schedul].ServiceTime + Process[number\_schedul].WholeTime;  Process[number\_schedul].run\_flag = 1;  temp\_time = Process[number\_schedul].FinishTime;  Process[number\_schedul].order = temp\_counter;  temp\_counter++;  }return 0;  }  int Pinput() //进程参数输入  {  int i;  printf("Enter the number of processes:");  scanf("%d", &N);  for (i = 0; i < N; i++)  {  printf("----Enter the informations of process %d----\n", i + 1);  printf("Enter the process name:");  scanf("%s", Process[i].Name);  printf("Enter the arrival time:");  scanf("%d", &Process[i].ArrivalTime);  printf("Enter the service time:");  scanf("%d", &Process[i].WholeTime);  Process[i].ServiceTime = 0;  Process[i].FinishTime = 0;  Process[i].WeightWholeTime = 0;  Process[i].order = 0;  Process[i].run\_flag = 0;  }return 0;  }  int Poutput() //调度结果输出  {  int i;  float turn\_round\_time = 0, f1, w = 0;  printf(" Name arrive\_t run\_t start\_t finish\_t execute\_seq around\_t weight\_around\_t\n");  for (i = 0; i < N; i++)  {  Process[i].WeightWholeTime = Process[i].FinishTime - Process[i].ArrivalTime;  f1 = Process[i].WeightWholeTime / Process[i].WholeTime;  turn\_round\_time += Process[i].FinishTime-Process[i].ArrivalTime;  w += f1;  printf("time%2d:", Process[i].ServiceTime);  printf("%-5s%-9d%-6d%-8d%-9d%-12d%-9.2f%-.2f\n", Process[i].Name, Process[i].ArrivalTime, Process[i].WholeTime, Process[i].ServiceTime, Process[i].FinishTime, Process[i].order, Process[i].WeightWholeTime, f1);  }  printf("average\_turn\_round\_time=%.3f\n", turn\_round\_time / N);  printf("weight\_average\_turn\_round\_time=%.3f\n", w / N);  return 0;  }  //2.先来先服务算法  typedef struct PCB //定义进程控制块  {  char name[10]; //进程名  char state; //运行状态  int arrivetime; //到达时间  int starttime; //进程开始时间  int finishtime; //进程结束时间  int servicetime; //服务时间  float turnaroundtime;//周转时间  float weightedturnaroundtime;//带权周转时间  struct PCB\* next; //指向下个进程  }pcb;  int time=0; //计时器  int n; //进程个数  float avg\_around\_time = 0; //平均周转时间  pcb\* head1 = NULL, \* p, \* q,\*min; //进程链表指针  void run\_fcfs(pcb\* p1) //运行未完成的进程  {  time =( p1->arrivetime > time ? p1->arrivetime : time);  p1->starttime = time;  printf("\nNow the time is %d,Start running job %s\n", time, p1->name);  time += p1->servicetime; //计时器  p1->state = 'F';  p1->finishtime = time;  p1->turnaroundtime = p1->finishtime - p1->arrivetime;  p1->weightedturnaroundtime = p1->turnaroundtime / p1->servicetime;  avg\_around\_time += p1->turnaroundtime;  printf("Name Arrive\_t start\_t served\_t finish\_t around\_t weight\_around\_t\n");  printf("%-5s %-9d%-8d%-9d%-9d%-9.1f%.2f\n",  p1->name,p1->arrivetime, p1->starttime, p1->servicetime, p1->finishtime,  p1->turnaroundtime, p1->weightedturnaroundtime);  }  void fcfs() //找先到先服务  {  int i, j;  for (i = 0; i < n; i++) //遍历链表找到服务顺序  {  p = head1;  q = p; //标记当前的进程  for (j = 0; j < n; j++)  {  if (p && p->state == 'W')  {min = p; break;}  p = p->next;  }  for (j=0;j < n;j++)  {  if(p->next) p=p->next;  if ((min->arrivetime >= p->arrivetime)&& min->state == 'W'&& p->state == 'W')  //F表示已经完成Finish状态，W-wait,R-run  {  min = p; //找到最先到达的进程  }  }  run\_fcfs(min);  }  printf("average\_turn\_round\_time=%f\n", avg\_around\_time / n);  }  void getInfo() //获得进程信息并创建进程  {  int num;  printf("\nNumber of jobs:");  scanf("%d", &n);  for (num = 0; num < n; num++)  {  if ((p = (pcb\*)malloc(sizeof(pcb))) == NULL)  {  perror("malloc");  exit(1);  }  printf("----Enter the informations of process %d----\n", num + 1);  printf("Enter the process name:");  scanf("%s", &p->name);  printf("Enter the arrival time:");  scanf("%d", &p->arrivetime);  printf("Enter the service time:");  scanf("%d", &p->servicetime);  if (head1 == NULL) { head1 = p; q = p; time = p->arrivetime; }  if (p->arrivetime < time) time = p->arrivetime;  q->next = p;  p->starttime = 0;  p->finishtime = 0;  p->turnaroundtime = 0;  p->weightedturnaroundtime = 0;  p->next = NULL;  p->state = 'W';  q = p;  }  }  //3.时间片轮转算法  typedef struct node  {  char name[20];  int round;  int cputime; //进程总共使用cpu的时间，即进程的服务时间  int starttime; //进程开始运行时间  int arrivetime; //进程到达内存的时间  int finishtime; //进程完成时间  int needtime; //进程的运行时间  char state; //进程状态  int count;  struct node\* next;  }PCB;  PCB\* ready = NULL, \* run1 = NULL, \* finish = NULL;  int num, i;  void GetFirst();  void Output();  void InsertTime(PCB\* in);  void InsertFinish(PCB\* in);  void TimeCreate();  void RoundRun();  void GetFirst() /\*取得第一个就绪队列节点\*/  {  run1 = ready;  if (ready != NULL)  {  run1->state = 'R'; //R--run  ready = ready->next;  run1->next = NULL;  }  }  void Output() /\*输出队列信息\*/  {  PCB\* p;  p = ready; //就绪队列信息  printf("Time Name cpu\_t Need\_Time status Counter\n");  while (p != NULL)  {  printf("%2d-%-2d %-5s%-6d%-10d%-7c%d\n",p->starttime,p->finishtime, p->name, p->cputime, p->needtime, p->state, p->count);  p = p->next;  }  p = finish; //完成队列信息  while (p != NULL)  {  printf("%2d-%-2d %-5s%-6d%-10d%-7c%d\n", p->starttime, p->finishtime, p->name, p->cputime, p->needtime, p->state, p->count);  p = p->next;  }  p = run1; //运行队列信息  while (p != NULL)  {  printf("%2d-%-2d %-5s%-6d%-10d%-7c%d\n", p->starttime, p->finishtime, p->name, p->cputime, p->needtime, p->state, p->count);  p = p->next;  }  }  void InsertTime(PCB\* in) /\*将进程插入到就绪队列尾部\*/  {  PCB\* fst;  fst = ready;  if (ready == NULL)  {  in->next = ready;  ready = in;  }  else  {  while (fst->next != NULL)  {  fst = fst->next;  }  in->next = fst->next;  fst->next = in;  }  }  void InsertFinish(PCB\* in) /\*将进程插入到完成队列尾部\*/  {  PCB\* fst;  fst = finish;  if (finish == NULL)  {  in->next = finish;  finish = in;  }  else  {  while (fst->next != NULL)  {  fst = fst->next;  }  in->next = fst->next;  fst->next = in;  }  }  void TimeCreate() /\*时间片输入函数\*/  {  PCB\* tmp;  int i;  for (i = 0; i < num; i++)  {  printf("----Enter the informations of process %d----\n", i+ 1);  if ((tmp = (PCB\*)malloc(sizeof(PCB))) == NULL)  {  perror("malloc");  exit(1);  }  printf("Enter the process name:");  scanf("%s", tmp->name);  printf("Enter the arrival time:");  scanf("%d", &(tmp->arrivetime));  printf("Enter the service time:");  scanf("%d", &(tmp->needtime));  printf("Enter the time slice:");  scanf("%d", &(tmp->round));  getchar();  tmp->cputime = 0;  tmp->state = 'W';  // tmp->prio=0;  tmp->count = 0;  InsertTime(tmp);  }  }  void RoundRun() /\*时间片轮转调度算法\*/  {  int flag = 1,flag2=1;  int time = 0; //当前时间  float avf\_around\_t = 0;  GetFirst();  while (run1 != NULL)  {  Output();  while (flag)  {  if (time < run1->arrivetime)  run1->starttime = run1->arrivetime;  else  run1->starttime = time;  run1->cputime++;  run1->needtime--;  run1->count++;  if (run1->needtime == 0)  {  run1->state = 'F';  run1->finishtime = run1->starttime + run1->count;//记录每个进程的实际结束时间  InsertFinish(run1); //在分配的时间片内运行完成  flag = 0;  }  else if (run1->count == run1->round) //时间片用完  {  run1->state = 'W';  run1->finishtime = run1->starttime + run1->round;  run1->count = 0;  InsertTime(run1); //进程再次回到就绪队列  flag = 0;  }  }  if(run1->state=='F')  {  avg\_around\_time += run1->finishtime - run1->arrivetime;  }  time = run1->finishtime;  flag = 1; //用于判断一个进程是否完成  GetFirst();  }  Output();  printf("average\_turn\_round\_time=%f\n", avg\_around\_time / num);  }  //4.静态优先权优先算法  int SequenceNumber = 1; //进程编号  int ProcessAmount; //进程数量  int StartCount = 0; //调入内存进程计数  struct PCB1 { //进程控制块  int No; //进程号  char name[16]; //进程名  int enterMemoryTime;//进入内存时间  int serviceTime; //服务时间  int priority; //优先级  struct PCB1\* next;  };  struct PCB1\* head, \* thisP, \* newP; //动态接收进程及详细  struct PCB1\* readyHead; //动态存储就绪队列  //函数声明  int timeOfData(FILE\*);  void ListAllPCB(struct PCB1\* h);  void release(struct PCB1\*);  void releaseR(struct PCB1\*);  void setNewPCB(char\*, int, int, int);  struct PCB1\* runTheProcess(int i);  void newPCB() { //建立PCB  newP = (struct PCB1\*)malloc(sizeof(struct PCB1));  if (head == NULL) { //判断头节点是否为空  head = newP; //为空，头节点指向新开辟的内存  }  else {  thisP = head;  while (thisP->next != NULL) {  thisP = thisP->next;  }  thisP->next = newP; //遍历单链表，找到最后一个元素  }  thisP = newP;  //插入新节点  thisP->No = SequenceNumber;  SequenceNumber++;  //printf("ID %d\n", thisP->No);  printf("----Enter the informations of process %d----\n", thisP->No);  printf("Enter the process name:");  scanf("%s", thisP->name);  printf("Enter priority:");  scanf("%d", &thisP->priority);  printf("Enter arrive time:");  scanf("%d", &thisP->enterMemoryTime);  printf("Enter service time:");  scanf("%d", &thisP->serviceTime);  thisP->next = NULL;  }  void buildProcess() { //创建进程  int i = 0;  printf("Enter the number of processes:");  scanf("%d", &ProcessAmount);  while (i < ProcessAmount) {  newPCB();  i++;  }  }  void readyList(struct PCB1\* pcb) { //单链表的形式创建就绪队列  newP = (struct PCB1\*)malloc(sizeof(struct PCB1));  if (readyHead == NULL) {  readyHead = newP;  }  else {  thisP = readyHead;  while (thisP->next != NULL) {  thisP = thisP->next;  }  thisP->next = newP;  }  thisP = newP;  strcpy(thisP->name, pcb->name);  thisP->No = pcb->No;  thisP->priority = pcb->priority;  thisP->enterMemoryTime = pcb->enterMemoryTime;  thisP->serviceTime = pcb->serviceTime;  thisP->next = NULL;  }  struct PCB1\* callMemory() { //调入内存，返回调入内存的链表节点  int at; //到达时间  struct PCB1\* markP;  if (head == NULL) {  printf("Program not found.\n");  }  else {  markP = thisP = head; //标记指向头节点  at = thisP->enterMemoryTime; //到达时间为头节点到达时间  while (thisP->next != NULL) { //当下一节点不为空  if (at > thisP->next->enterMemoryTime) { //判断当前时间是否大于下一节点时间  markP = thisP->next; //是，标记此节点  at = markP->enterMemoryTime; //到达时间更改为标记时间  }  thisP = thisP->next; //向后遍历  }  }  return markP;  }  int timesetout(int at) //判断就绪队列中是否有在当前时间之前到达的进程  {  struct PCB1\* markThis;  if (readyHead == NULL) {  printf("There are no runnable processes.\n");  }  else {  markThis = thisP = readyHead;  time = thisP->enterMemoryTime;//当前到达时间  while (thisP->next != NULL) {  if (thisP->enterMemoryTime < at)  return 1; //表示有在当前时间之前到达的进程  thisP = thisP->next;  }  return 0;  }  }  void run() { //静态优先级算法执行。  struct PCB1\* temp; //临时节点用来存储调入内存节点  struct PCB1\* runPro; //用来接收执行的节点  int i; //循环初始条件  int at=0, srt, finishtime = 0; //当前时间，开始执行时间，运行时间。  int turnOverTime; //周转时间  float avg\_around\_time = 0;  double ptot; //带权周转时间  if (head == NULL) {  printf("No processes found.\n");  }  else {  while (head != NULL) { //将所有进程都放到就绪队列中去  temp = callMemory(); //返回当前最先进入内存的进程  readyList(temp); //初始纳入就绪列表  release(temp); //释放进程  }//循环完成之后，就绪队列中的进程按照进入内存的时间由小到大排序  for (i = 0; i < ProcessAmount; i++) { //循环进程的数量次  runPro = runTheProcess(timesetout(at)); //1.返回就绪队列中当前最先进入内存且在同时间到达的进程中优先权最高的进程  //2.返回就绪队列中优先级最高的进程  if (runPro->enterMemoryTime > at) {  //初始情况下求开始执行时间，完成时间，周转时间，带权周转时间  srt = runPro->enterMemoryTime; //开始时间等于到达时间  }  else { //初始之后，计算各时间及周转  srt = at;//开始时间等于当前时间  }  finishtime = srt + runPro->serviceTime;  turnOverTime = finishtime - runPro->enterMemoryTime;  avg\_around\_time += turnOverTime;  ptot = turnOverTime \* 1.0 / runPro->serviceTime;  at = finishtime;  printf("Currently executing process:");  printf("\n");  printf("Name ID priority arrive\_t serve\_t start\_t finish\_t around\_t weight\_around\_t\n");  if (runPro != NULL) {  printf("%-5s%-5d%-9d%-9d%-8d", runPro->name, runPro->No, runPro->priority, runPro->enterMemoryTime, runPro->serviceTime);  printf("%-8d%-9d%-9d%-8.2lf", srt, finishtime, turnOverTime, ptot);  }  else  printf("There are no processes.\n");  releaseR(runPro);  printf("\nReady process list: ");  ListAllPCB(readyHead);  }  }  printf("average\_turn\_round\_time=%f\n", avg\_around\_time / ProcessAmount);  }  struct PCB1\* runTheProcess(int i) { //执行的进程，返回要执行进程的节点  int time, pri;  struct PCB1\* markThis; //标记要返回的节点  if (readyHead == NULL) {  printf("There are no runnable processes.\n");  }  else {  markThis = thisP = readyHead;  time = thisP->enterMemoryTime;//当前到达时间  pri = thisP->priority; //当前进程的优先级  while (thisP->next != NULL) {  if (i == 0)  {  if (time == thisP->next->enterMemoryTime) {//判断之后节点的到达时间是否与当前一致  if (pri < thisP->next->priority) {//一致，则判断优先级决定  pri = thisP->next->priority;//更改优先级判断条件为下一优先级  markThis = thisP->next; //标记当前节点的下一节点  }  else {  break;  }  }  }  else {  if (pri < thisP->next->priority) {//判断优先级决定  pri = thisP->next->priority;//更改优先级判断条件为下一优先级  markThis = thisP->next; //标记当前节点的下一节点  }  else {  break;  }  }  thisP = thisP->next;  }  }  return markThis;  }  void release(struct PCB1\* pcb) { //删除原单链表的进程节点  struct PCB1\* markP, \* f;  markP = head;  if (pcb != NULL) {  if (pcb == head) {  f = head;  head = head->next;  free(f);  }  else {  thisP = head->next;  while (thisP != NULL) {  if (pcb == thisP) {  f = thisP;  markP->next = thisP->next;  free(f);  break;  }  else {  markP = thisP;  }  thisP = thisP->next;  }  }  }  }  void releaseR(struct PCB1\* pcb) { //删除就绪列表中进程节点  struct PCB1\* markP, \* f;  markP = readyHead;  if (pcb != NULL) {  if (pcb == readyHead) {  f = readyHead;  readyHead = readyHead->next;  free(f);  }  else {  thisP = readyHead->next;  while (thisP != NULL) {  if (pcb == thisP) {  f = thisP;  markP->next = thisP->next;  free(f);  break;  }  else {  markP = thisP;  }  thisP = thisP->next;  }  }  }  }  void ListAllPCB(struct PCB1\* h) { //打印所有进程  if (h == NULL) {  printf("\nThere are no processes.\n\n");  }  else {  printf("\n");  printf("Name ID priority arrie\_t served\_t\n");  thisP = h;  while (thisP != NULL) {  printf("%-5s%-5d%-9d%-8d%-8d\n", thisP->name, thisP->No, thisP->priority, thisP->enterMemoryTime, thisP->serviceTime);  thisP = thisP->next;  }  printf("\n");  }  }  //5.高响应比调度算法  int N;  float timecounter5 = 0; //计时器  struct zgxyb\* c;  struct zgxyb {  char name[10];  float arrivetime;  float servicetime;  float starttime;  float finishtime;  float around\_time; //周转时间  float zztime;//等待时间  float dqzztime; //进程的响应比  char stats; //进程状态  };  struct zgxyb a[100];  void input(struct zgxyb\* p, int N)  {  int i;  for (i = 0; i <= N - 1; i++)  {  printf("----Enter the informations of process %d----\n",i + 1);  //printf("Name, arrive\_t need\_t:\n");  //scanf("%s%f%f", &p[i].name, &p[i].arrivetime, &p[i].servicetime);  printf("Enter the process name:");  scanf("%s", &p[i].name);  printf("Enter the arrival time:");  scanf("%f", &p[i].arrivetime);  printf("Enter the service time:");  scanf("%f", &p[i].servicetime);  p[i].stats = 'W';  }  }  void Print(struct zgxyb\* p)  {  int k;  printf("\nrun order:");  printf("%s", p[0].name);  for (k = 1; k < N; k++)  {  printf("-->%s", p[k].name);  }  printf("\nCurrent executing\_process information:\n");  printf("Name arrive\_t served\_t start\_t finish\_t around\_t RRatio\n");  for (k = 0; k <= N - 1; k++)  {  printf(" %-5s%-9.2f%-9.2f%-8.2f%-9.2f%-9.2f%-.2f\n", p[k].name, p[k].arrivetime, p[k].servicetime, p[k].starttime, p[k].finishtime, p[k].around\_time, p[k].dqzztime);  }  }  //按到达时间排序  void sort(struct zgxyb\* p,int N)  {  int i, j;  for (i = 0; i <= N - 1; i++)  for (j = 0; j <= i; j++)  if (p[i].arrivetime < p[j].arrivetime)  {  struct zgxyb temp;  temp = p[i];  p[i] = p[j];  p[j] = temp;  }  }  void calculate(struct zgxyb\* p)  {  for (int n = 1; n < N; n++) //计算各个进程的相应比  {  if (p[n].stats == 'W')  {  p[n].zztime = timecounter5 - p[n].arrivetime; //进程的等待时间=当前时间-到达时间  p[n].dqzztime = 1 + p[n].zztime / p[n].servicetime;  }  }  }  void ZGXYB(struct zgxyb\* p, int N)  {  int m, i, n, follow, k;  int schedul=0; //当前时间  float max=0,avg\_around\_time=0;  struct zgxyb temp;  sort(p,N); //按到达时间对进程进行排序  for(i=0;i<N;i++) //运行N个进程  {  if (timecounter5 < p[schedul].arrivetime)  p[schedul].starttime = p[schedul].arrivetime;  else  p[schedul].starttime = timecounter5;  p[schedul].finishtime = p[schedul].starttime + p[schedul].servicetime;  p[schedul].around\_time = p[schedul].finishtime - p[schedul].arrivetime;  avg\_around\_time+= p[schedul].around\_time;  p[schedul].stats = 'F';  timecounter5 = p[schedul].finishtime;    calculate(p); //计算各进程相应比  for (m = 1; m < N; m++) //选择一个未完成进程  {  if (p[m].stats == 'W')  {  max = p[m].dqzztime;  schedul = m;  break;  }  }  for (int k = 1; k < N; k++) //选出响应比最高的进程  {  if (max <= p[k].dqzztime && p[k].stats == 'W')  {  max = p[k].dqzztime;  schedul = k;  }  }  }  Print(p);  printf("average\_turn\_round\_time=%f\n", avg\_around\_time / N);  }  //从文件读取数据  /\*数据格式如下  进程名 到达时间 服务时间 优先级 时间片  p1 0 7 1 5  p2 2 4 2 3  p3 5 4 2 3  \*/  //1.短作业优先  void SJFsetfin(char\* name, int entime, int sertime,int i) //进程参数输入  {  strcpy(Process[i].Name,name);  Process[i].ArrivalTime=entime;  Process[i].WholeTime=sertime;  Process[i].ServiceTime = 0;  Process[i].FinishTime = 0;  Process[i].WeightWholeTime = 0;  Process[i].order = 0;  Process[i].run\_flag = 0;  }  //2.先来先服务  void FCFSsetfin(char\* name, int entime, int sertime)  {  p = (pcb\*)malloc(sizeof(pcb));  strcpy(p->name, name);  p->arrivetime = entime;  p->servicetime = sertime;  if (head1 == NULL) { head1 = p; q = p; time = p->arrivetime; }  if (p->arrivetime < time) time = p->arrivetime;  q->next = p;  p->starttime = 0;  p->finishtime = 0;  p->turnaroundtime = 0;  p->weightedturnaroundtime = 0;  p->next = NULL;  p->state = 'W';  q = p;  }  //3.时间片轮询  PCB\* tmp;  void RRsetfin(char\* name, int entime, int sertime,int slice)  {  if ((tmp = (PCB\*)malloc(sizeof(PCB))) == NULL)  {  perror("malloc");  exit(1);  }  strcpy(tmp->name, name);  tmp->arrivetime = entime;  tmp->needtime = sertime;  tmp->round = slice;  tmp->cputime = 0;  tmp->state = 'W';  tmp->count = 0;  InsertTime(tmp);  }  //4.最高响应比  void ZGXYBsetin(struct zgxyb\* p,char\* name, float entime, float sertime,int i)  {  strcpy(p[i].name, name);  p[i].arrivetime = entime;  p[i].servicetime = sertime;  p[i].stats = 'W';  }  //5.静态优先级  void setNewPCB(char\* name, int pri, int entime, int sertime) {//赋值建立PCB，用于从文本中获取数据  newP = (struct PCB1\*)malloc(sizeof(struct PCB1));  if (head == NULL) {  head = newP;  }  else {  thisP = head;  while (thisP->next != NULL) {  thisP = thisP->next;  }  thisP->next = newP;  }  thisP = newP;  thisP->No = SequenceNumber;  SequenceNumber++;  strcpy(thisP->name, name);  thisP->priority = pri;  thisP->enterMemoryTime = entime;  thisP->serviceTime = sertime;  thisP->next = NULL;  }  void readFromFile(int option) {  FILE\* fp;  int times;  int i = 0,pri;  char filePath[20], temp[1024];  char name[16];  float emt, st, slice;  printf("Enter file path:");  scanf("%s", filePath);  fp = fopen(filePath, "r");  times = timeOfData(fp);  ProcessAmount =N=n=num=times - 2; //进程个数  while (i < times - 1) { //循环次数=文本行数  if (i == 0) { //这是用来接收文本格式第一行的汉字。  fgets(temp, sizeof(temp), fp); //把读取的第一行的内容放在temp数组中  }  else { //以下用来接收进程数据  fscanf(fp, "%s", name);  fscanf(fp, "%f", &emt);  fscanf(fp, "%f", &st);  fscanf(fp, "%d", &pri);  fscanf(fp, "%f", &slice);  switch (option)  {  case 1:  SJFsetfin(name, emt, st, i-1); //添加一个进程到就绪队列中  break;  case 2:  RRsetfin(name, emt, st, slice);//添加一个进程  break;  case 3:  FCFSsetfin(name, emt, st);//添加一个进程  break;  case 4:  ZGXYBsetin(a, name, emt, st, i-1);//添加一个进程  break;  case 5:  setNewPCB(name, pri, emt, st);//添加一个进程  break;  default:  break;  }  }  i++;  }  fclose(fp);  }  int timeOfData(FILE\* f) { //求取从文本中获取数据的进程数量  char temp[20];  int i = 0;  int flag = 0;  if (f == NULL) {  printf("File not found.\n");  exit(0);  }  else {  while (!feof(f))  {  flag = fgetc(f);  if (flag == '\n')  i++;  }  }  rewind(f);  return i;  }  //主函数  void choose(char option, int select)  {  switch (option)  {  case '0':  printf("End of operation.\n");  return;  case '1':  printf("\n -----Short job first scheduling algorithm------\n\n");  if (select != 'F')  Pinput();  else  readFromFile(1);  SJF(); sh:  Poutput();  return;  case '2':  printf("\n --------------Round-Robin algorithm------------\n\n");  if (select != 'F')  {  printf("Enter the number of processes to create :");  scanf("%d", &num);  TimeCreate();  }  else  readFromFile(2);  RoundRun();  return;  case '3':  printf("\n ------First come first serve algorithm-------\n");  if (select != 'F')  getInfo();  else  readFromFile(3);  p = head1;  fcfs();  return;  case '4':  printf("\n -----Highest Response Ratio Next algorithm----\n\n");  if (select != 'F')  {  printf("Enter the number of processes:");  scanf("%d", &N);  input(a, N);  }  else  readFromFile(4);  ZGXYB(a, N);  return;  case '5':  printf("\n -----Static priorit algorithm process scheduling---\n\n");  if (select != 'F')  {  buildProcess();  }  else  readFromFile(5);  printf("Process list: ");  ListAllPCB(head);  run();  return;  default:  break;  }  }  void main()  {  char select,option[5];  printf(" --------------Welcome to use this system------------\n");  printf(" ----------------------------------------------------\n");  printf(" -- 1 SJF,Shortest Job First --\n");  printf(" -- 2 RR,Round-Robin --\n");  printf(" -- 3 FCFS,first come first served --\n");  printf(" -- 4 HRRN,Highest Response Ratio Next --\n");  printf(" -- 5 static priority algorithm --\n");  printf(" -- 0 exit --\n");  printf(" ----------------------------------------------------\n");  printf("Enter 'F'or'f' to input information from file.\n\  If you want to enter it manually enter any key to continue:");  scanf("%c", &select);  printf("\nPlease enter a num to select algorithm:");  scanf("%s", &option);  int len=strlen(option);  for (int i = 0; i < len; i++)  {  choose(option[i], select);  }  } |