**计算机科学与工程学院 实验报告**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **实验课程名称** | | **操作系统** | | |
| **专业** | **人工智能** | | **班级** | **1901** |
| **学号** | **20191218** | | **姓名** | **孙平炜** |
| **实验项目目录**   1. 实验一 进程调度算法 2. 实验二 生产者消费者问题 3. 实验三 进程的管道通信 4. 实验四 页面置换算法 | | | | |
| **实验报告正文**  **实验题目：进程调度算法**  **实验简述：这是一个设计型实验。要求自行设计、编制模拟程序，通过形象化的状态显示，加深理解进程的概念、进程之间的状态转换及所带来的PCB组织的变化，理解进程与其PCB间的一一对应关系。**  **实验内容：**  **完成的功能点：\*实现UI交互控制，五状态下实现优先级控制调度**  **环境：python3+PyQt5**  **程序结构图：**    **分段讲解：**  由于UI界面代码大多是体力劳动，还有的是靠Qtdesniger生成，所以接下来会主要介绍控制逻辑的实现，并对UI的响应逻辑做简要说明。  首先要定义一个程序控制快类（PCB\_Class）以便后续实例化一些PCB并交给控制程序操作。    然后我们来看调度模块，首先我设置四个队列来作为容器管理五种状态下的进程。由于python不像C++的STL那样有封装好的队列供我们调用，因此我这里定义的队列实际上是队头指针，依靠后面链接PCB的next参数来模拟队列操作。    随后是一些对队列的基本操作。    接下来则是一些调度函数。    这里以running调度为例讲解一下具体实现。  # 在时间片用完时自动调用  # 在ready队列中找优先级最高的节点置于running队列中 此队列只能有一个节点  def running(self):  if self.head\_running.next != None:  if self.head\_ready.next == None: #当前进程要跑完了 又没有预备的进程  return  else: # 有预备进程在ready队列则  running\_node = self.head\_running.next  running\_node = self.drop\_node(running\_node)  running\_node.status = 'ready'  ready\_node = self.head\_ready.next  ready\_node = self.drop\_node(ready\_node)  ready\_node.status = 'running'  self.head\_running.next = ready\_node  #按优先级维护ready队列  p = self.head\_ready  while p.next != None and running\_node.PRL >= p.PRL:  p = p.next  if p.next == None:  p.next = running\_node  else:  running\_node.next = p.next  p.next = running\_node  # running队列为空  else:  if self.head\_ready.next == None:  return  # running队列为空，ready队列不为空  else:  ready\_node = self.head\_ready.next  ready\_node = self.drop\_node(ready\_node)  ready\_node.status = 'running'  self.head\_running.next = ready\_node  再结合一段处理UI请求的代码看一下调度函数是如何工作的。  def run(self):  while self.running:  head = global\_pcb.pcb.head\_new # 检查new缓冲区  p = head.next  while p != None:# 将new队列的进程自动加入ready队列 改变状态栏  if p.pid == 1:  self.mainwindow.admit1\_signal.emit(1)  elif p.pid == 2:  self.mainwindow.admit2\_signal.emit(2)  elif p.pid == 3:  self.mainwindow.admit3\_signal.emit(3)  elif p.pid == 4:  self.mainwindow.admit4\_signal.emit(4)  elif p.pid == 5:  self.mainwindow.admit5\_signal.emit(5)  p = p.next  # 检查ready队列  head = global\_pcb.pcb.head\_ready  p = head.next  if p != None: # 运行pid对应的进程  if p.pid == 1:  self.mainwindow.running1\_signal.emit(1)  elif p.pid == 2:  self.mainwindow.running2\_signal.emit(2)  elif p.pid == 3:  self.mainwindow.running3\_signal.emit(3)  elif p.pid == 4:  self.mainwindow.running4\_signal.emit(4)  elif p.pid == 5:  self.mainwindow.running5\_signal.emit(5)  else: # 处理器空闲  if global\_pcb.pcb.head\_running.next == None:  time.sleep(1)  continue  else:  p = global\_pcb.pcb.head\_running.next  # 开始执行一段时间片  global\_pcb.pcb.running()  time.sleep(1)  p.run\_progress += 100  p.run\_time += 3  **实验结果（截图）**  **初始效果**    **运行进程**    **阻塞进程**    **释放进程** | | | | |
| **实验题目：生产者消费者问题**  **实验简述：这是一个验证型实验。通过对给出的程序进行验证、修改，进一步加深理解进程的概念，了解同步和通信的过程，掌握进程通信和同步的机制，特别是利用缓冲区进行同步和通信的过程。通过补充新功能，加强对知识的灵活运用，培养创新能力。**  **实验内容：**  **完成的功能点：模拟生产者消费者问题，展示相关队列和\*信号量情况**  **环境：C++**  **程序说明：**  vector<int> buffer(BUFFER\_SIZE);    // 缓冲区  queue<int> pQueue,cQueue;           // 等待队列  int production\_id;  int pptr, cptr;             // 写指针　读指针  int consumer, producer;             // 消费者等待数量　生产者等待数量  void init() {//将相关数据容器都置为空或0}  int findEmpty(vector<int>& v, int len)  {// 查找空位}  int findDirty(vector<int>& v, int len)  {// 查找占用位}  bool isFull(vector<int>& v, int len)  {// 缓冲区是否满}  bool isEmpty(vector<int>& v, int len)  {// 缓冲区是否空}  void showQueue(queue<int> q)  {// 展示队列}  void show()  {// 展示数据内容}  void produce() {      if(findEmpty(buffer, BUFFER\_SIZE) == -1) // 缓冲区满,阻塞      {          ++production\_id;          producer++;          pQueue.push(production\_id);      }      else if(isEmpty(buffer, BUFFER\_SIZE)) // 缓冲区空      {          if(cQueue.empty()) // 消费者等待队列为空,直接写入          {              production\_id++;              buffer[pptr] = production\_id;          }          else // 有阻塞的消费者，先写入再读出          {                production\_id++;              buffer[pptr] = production\_id;              show();              sleep(2);              buffer[cptr] = 0;              cptr = (cptr + 1) % BUFFER\_SIZE;              consumer--;              cQueue.pop();          }          pptr = (pptr + 1) % BUFFER\_SIZE;      }      else // 缓冲区不空也不满,直接写入即可      {          production\_id++;          buffer[pptr] = production\_id;          pptr = (pptr + 1) % BUFFER\_SIZE;      }  }  void consume() {      if(findDirty(buffer, BUFFER\_SIZE) == -1) // 缓冲区空,则阻塞      {          consumer++;          cQueue.push(production\_id+1);      }      else if(isFull(buffer, BUFFER\_SIZE)) // 缓冲区满      {          if(!pQueue.empty()) // 生产者等待队列不空，则先读出再写入          {              buffer[cptr] = 0;              cptr = (cptr + 1) % BUFFER\_SIZE;              show();              sleep(2);              buffer[pptr] = pQueue.front();              pptr = (pptr + 1) % BUFFER\_SIZE;              pQueue.pop();              producer--;          }          else // 生产者队列为空          {                buffer[cptr] = 0;              cptr = (cptr + 1) % BUFFER\_SIZE;          }      }      else // 缓冲区不空也不满,直接读取即可      {          buffer[cptr] = 0;          cptr = (cptr + 1) % BUFFER\_SIZE;      }  }  void start()  {//处理IO调用函数进行模拟}  int main() {      cout << "e-exit    p-produce    c-consume" << endl;      init();      start();      return 0;  }  **实验结果（截图）**  生产者依次装入直到缓冲区满开始等待  消费者依次取走直到缓冲区空开始等待    生产者先补足消费者等待队列再装入 | | | | |
| **实验题目：进程的管道通信**  **实验简述：加深对进程概念的理解，明确进**  **程和程序的区别。**  **学习进程创建的过程，进一步认**  **识进程并发执行的实质。**  **分析进程争用资源的现象，学习**  **解决进程互斥的方法。**  **学习解决进程同步的方法。**  **掌握Linux系统中进程间通过管道通信的具体实现。**  **实验内容：**  **完成的功能点：从父进程创建子进程并实现管道通信**  **程序说明：**  pid = fork();          printf("pid: %d\n", pid);          if(pid == -1)       //创建失败          {              printf("creat subprocess failed!\n");              exit(0);          }          else if(pid == 0)   // 创建子进程，从子进程返回ID          {              printf("now I'm writing in subprocess %d\n",gtid = getpid());              lockf(fd[1], F\_LOCK, 0);    // 加锁            printf(outpipe, "child process %d is sending message!\n", gtid);              write(fd[1], outpipe, sizeof(outpipe));              sleep(1);              lockf(fd[1], F\_ULOCK, 0);   // 解锁          printf("send message successfully and exit subprocess %d\n", gtid);              exit(0);          }          else if(pid > 0)        // 创建子进程，从父进程返回子进程的ID          {              reid = wait(NULL);           if(reid == -1) printf("father process calls subprocess failed!\n");          else              {                  read(fd[0],father\_str,sizeof(father\_str));                  printf("\nfather process reads from subprocess %d:\n",reid);                  printf("%s\n",father\_str);              }          }          printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  **实验结果（截图）** | | | | |
| **实验题目：页面置换算法**  **实验简述：页面置换算法是虚拟存储管理实现的关键，通过本次试验理解内存页面调度的机制，在模拟实现FIFO、LRU等经典页面置换算法的基础上，理解虚拟存储实现的过程。**  **将不同的置换算法放在不同的子进程中加以模拟，培养综合运用所学知识的能力。**  **实验内容：**  **完成的功能点：实现基于FIFO和LRU算法的页面置换模拟，\*体现belady现象**  **程序说明：**  **FIFO算法**  pid = fork();      if(pid == -1)      {//创建失败}      else if(pid == 0)      {          while(access\_index < sequence\_len) // 依次处理每条访问请求          {              for(i = 0; i < frame\_num\_FIFO; i++)              {//查找当前请求页是否在内存}              if(内存中没有当前页)              {                  diseffect += 1.0;                  for(i = 0; i < frame\_num\_FIFO; i++)                  {  //检查进程的内存分区是否用完  //如果没用完就把请求页装入队尾  }                  if(//用完了)                  {                      for(i = 0; i < frame\_num\_FIFO - 1; i++)        M\_Frame\_FIFO内元素依次前移；      末位载入当前请求页                  }              }                for(i = frame\_num\_FIFO - 1; i >= 0; i--)              {//打印信息}              printf("--------------\n");              success\_flag = false;              access\_index++;          }          //计算缺页率  //重置全局变量          exit(0);      }  //回收资源      reid = wait(NULL);      if(reid == -1)          printf("\* call subprocess\_FIFO failed!\n");  **LRU算法：利用栈实现**  pid = fork();     if(pid == -1)      {//创建失败}      else if(pid == 0)   // subprocess      {          while(//依次处理访存请求)          {              for(i = 0; i < frame\_num; i++) //查找当前页是否在内存              {                  if (M\_Frame[i].flag == USED)                  {                      if (M\_Frame[i].page\_no == Access\_Series[access\_index])                      {                          success\_flag = true; // 找到了                          for(j = i; j < frame\_num - 1; j++)                          { // 将命中的页重新放到栈顶}                          break;                      }                  }                  else                      break;              }              if(!success\_flag)   // 未命中              {                  diseffect += 1.0;                 for(i = 0; i < frame\_num\_FIFO; i++)                  {  //检查进程的内存分区是否用完  //如果没用完就把请求页装入队尾  }                  if(//用完了)                  {                      for(i = 0; i < frame\_num\_FIFO - 1; i++)        M\_Frame\_FIFO内元素依次前移；      末位载入当前请求页                  }              for(i = frame\_num - 1; i >= 0; i--) // print the state of frames             {//打印信息}              printf("--------------\n");              success\_flag = false;              access\_index++;          }          //计算缺页率  //重置全局变量          exit(0);      }  //回收资源      reid = wait(NULL);      if(reid == -1)          printf("\* call subprocess\_LRU failed!\n");  **实验结果（截图）** | | | | |