**计算机科学与工程学院实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **实验课程名称** | | **操作系统** | | | **实验成绩** |  |
| **专业** | **计算机科学与技术** | | **班级** | **1603** | **指导教师签字** |  |
| **学号** | **20164448** | | **姓名** | **祝富锴** | **实验报告批改时间** |  |
| **实验项目目录**   1. 实验一 进程的状态转换及PCB的变化 2. 实验二 进程同步和通信——生产者和消费者问题模拟 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**  **实验一 进程的状态转换及PCB的变化**   1. 实验目的   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，使学生理解进程的概念、进程之间的状态转换及其所带来的PCB内容、组织的变化，理解进程与其PCB间的一一对应关系。   1. 实验原理   五状态进程模型：    三种基本状态：  1）就绪(Ready)状态：指进程已处于准备好运行的状态，及进程已经分配到需要的系统资源，只要在获得CPU就可以执行  2）执行(Running)状态：指进程获得了CPU正在执行，在单处理机系统中，最多只有一个进程处于该状态  3）阻塞(Block)状态：指正在执行的进程，在执行过程中发生了某时间（如：I/O请求、申请缓冲区失败等）  为满足进程控制块对数据及操作的完整性要求以及增强管理的灵活性，通常在引入两种状态：创建状态和终止状态  1）创建状态：  创建一个进程要经过以下几步：  - 首先进程申请一个空白PCB，并向PCB中填写用于控制和管理进程的信息  - 然后为该进程分配运行时所必须的资源  - 最后把该进程转入就绪状态并插入就绪队列  引入创建状态是为了保证进程的调度必须是在创建工作完成之后  2）终止状态：  进程的终止状态有以下两步：  - 首先，等待操作系统做善后处理  - 最后将其PCB清零，并将PCB空间返还给系统  当一个进程达到了自然结束点或是出现了无法克服的错误，或是被操作系统终结，则进入终止状态。进入终止状态的进程以后不能再执行，但在操作系统中保存状态码和一些计时统计数据供其他进程收集  1.挂起操作的引入：  1）终端用户的需要：当终端用户在运行程序期间发现有可疑问题，希望暂停程序的运行以便研究其执行情况或做一定的修改  2）父进程请求  3）符合调节的需要  4）操作系统的需要：有时希望挂起某些进程以便检查运行中的资源使用情况或进行记账  2.分别使用挂起原语Suspend和激活原语Active对进程进行挂起或激活  20171208095743596  最高优先数优先调度算法：  优先数调度算法常用于批处理系统中。在进程调度中，每次调度时，系统把处理机分配给就绪队列中优先数最高的进程。它又分为两种：非抢占式优先数算法和抢占式优先数算法。  在非抢占式优先数算法下，系统一旦把处理机分配给就绪队列中优先数最高的进程后，这个进程就会一直运行，直到完成或发生某事件使它放弃处理机，这时系统才能重新将处理机分配给就绪队列中的另一个优先数最高的进程。  在抢占式优先数算法下，系统先将处理机分配给就绪队列中优先数最高的进程度让它运行，但在运行的过程中，如果出现另一个优先数比它高的进程，它就要立即停止，并将处理机分配给新的高优先数进程。   1. 实验内容（源码、注释、基础内容、扩展点等） 2. 内容要求： 3. .设计并实现一个模拟进程状态转换及其相应PCB组织结构变化的程序 4. 独立设计、编写、调试程序 5. 程序界面应能反映出在模拟条件下，进程之间状态转换及其对应的PCB组织的变化 6. 程的状态模型（三状态、五状态、七状态或其它）可自行选择 7. 代码书写要规范，要适当地加入注释 8. 鼓励在实验中加入新的观点或想法，并加以实现 9. 认真进行预习，完成预习报告 10. 实验完成后，要认真总结，完成实验报告 11. 数据结构：   struct PCB{  char name;//进程名字  int priority;//优先数  int needtime;//需要运行时间  bool operator < (const PCB &b) const{  return priority>b.priority;  }  };   1. 程序流程图：   未命名文件   1. 源码：   #include<cstdio>  #include<algorithm>  int Ready\_len = 0;  int Blocked\_len = 0;  int CPU\_state = 0;  struct PCB  {  char name;  int priority;  int needtime;  bool operator < (const PCB &b) const  {  return priority > b.priority;  }  };  PCB Ready[100];  PCB Blocked[100];  PCB Cpu;  bool dispatch();  bool creat(int NUM) //创建一个新的进程  {  while (NUM--)  {  printf("输入进程名(一个字符)、所需时间(一个整数)、优先级(一个整数): \n");  scanf("%s%d%d", &(Ready[Ready\_len].name), &(Ready[Ready\_len].needtime), &(Ready[Ready\_len].priority));  getchar();  Ready\_len++;  }  if (CPU\_state == 0)//如果CPU空闲，则调度  dispatch();  }  bool dispatch()  {  if (CPU\_state == 0)  {  if (Ready\_len != 0)  {  sort(Ready, Ready + Ready\_len);  Cpu.name = Ready[0].name;  Cpu.needtime = Ready[0].needtime;  Cpu.priority = Ready[0].priority;  if (Ready\_len != 1)//就绪队列剔除队首元素  for (int indx = 1; indx < Ready\_len; indx++)  {  Ready[indx - 1].name = Ready[indx].name;  Ready[indx - 1].needtime = Ready[indx].needtime;  Ready[indx - 1].priority = Ready[indx].priority;  }  Ready\_len--;  CPU\_state = 1;  printf("\*\*\*%c进程送往CPU执行\n", Cpu.name);  Cpu.needtime--;  Cpu.priority--;  }  else  {  printf("\*\*\*就绪队列为空，无法调度\n");  return false;  }  }  else  {  printf("\*\*\*CPU忙，无法调度\n");  }  }  bool time\_out()  {  if (CPU\_state == 1)  {  if (Cpu.needtime == 0)  printf("\*\*\*%c时间片用完，并且执行完毕，被释放\n", Cpu.name);  else  {  Ready[Ready\_len].name = Cpu.name;  Ready[Ready\_len].needtime = Cpu.needtime;  Ready[Ready\_len].priority = Cpu.priority;  Ready\_len++;  printf("\*\*\*%c时间片用完\n", Cpu.name);  }  CPU\_state = 0;  Cpu.name = 0;  Cpu.needtime = 0;  Cpu.priority = 0;  if (Ready\_len != 0)//时间片用完，如果就绪队列不为空，则调度  dispatch();  }  else  {  printf("\*\*\*没有进程在CPU中，无法超时\n");  }  }  bool event\_wait()  {  if (CPU\_state == 1)  {  Blocked[Blocked\_len].name = Cpu.name;  Blocked[Blocked\_len].needtime = Cpu.needtime;  Blocked[Blocked\_len].priority = Cpu.priority;  Blocked\_len++;  printf("\*\*\*%c被阻塞\n", Cpu.name);  CPU\_state = 0;  if (Ready\_len != 0)//进程被阻塞，如果就绪队列不为空，则调度  dispatch();  }  else  printf("\*\*\*没有进程在CPU中，无法阻塞\n");  }  bool event\_occur()  {  if (Blocked\_len != 0)  {  //sort(Blocked,Blocked+Blocked\_len);  Ready[Ready\_len].name = Blocked[0].name;  Ready[Ready\_len].needtime = Blocked[0].needtime;  Ready[Ready\_len].priority = Blocked[0].priority;  Ready\_len++;  if (Blocked\_len != 1)//阻塞队列剔除队首元素  for (int indx = 1; indx < Blocked\_len; indx++)  {  Blocked[indx - 1].name = Blocked[indx].name;  Blocked[indx - 1].needtime = Blocked[indx].needtime;  Blocked[indx - 1].priority = Blocked[indx].priority;  }  Blocked\_len--;  //printf("%d %d",Blocked\_len,Ready\_len);  printf("\*\*\*%c被唤醒\n", Ready[Ready\_len - 1].name);  if (CPU\_state == 0)//如果CPU空闲，则调度  dispatch();  //printf("%d %d",Blocked\_len,Ready\_len);  }  else  printf("\*\*\*阻塞队列为空，无法唤醒\n");  }  int main()  {  int Cputime = 1;  while (1)  {  printf("\n1:New\t\t\t2:Dispatch\n");  printf("3:Timeout\t\t4:Event wait\n");  printf("5:Event occur\t\t0:exit\n");  printf("输入1--5实现相应的功能：\n");  int select;  scanf("%d", &select);  getchar();  switch (select)  {  case 1:  int num;  printf("输入要创建的进程数：\n");  scanf("%d", &num);  getchar();  creat(num);  break;  case 2:  dispatch();  break;  case 3:  time\_out();  break;  case 4:  event\_wait();  break;  case 5:  event\_occur();  break;  case 0:  exit(0);  break;  }  printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Cputime:%3d\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n", Cputime);  printf("状态\t\t进程名\t\t需要时间\t\t优先级\n");  if (CPU\_state) //显示CPU中的进程  {  printf("Running:\t%c\t\t", Cpu.name);  printf("%d\t\t\t", Cpu.needtime);  printf("%d\n", Cpu.priority);  }  if (Ready\_len) //显示Ready队列中的进程  {  for (int a = 0; a < Ready\_len; a++)  {  printf("Ready%d:\t\t", a);  printf("%c\t\t", Ready[a].name);  printf("%d\t\t\t", Ready[a].needtime);  printf("%d\n", Ready[a].priority);  }  }  if (Blocked\_len) //显示Blocked队列中的程序  {  for (int b = 0; b < Blocked\_len; b++)  {  printf("Blocked%d:\t", b);  printf("%c\t\t", Blocked[b].name);  printf("%d\t\t\t", Blocked[b].needtime);  printf("%d\n", Blocked[b].priority);  }  }  printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  Cputime++;  }  }  四、实验结果（截图）  Screenshot from 2019-06-16 08-34-15Screenshot from 2019-06-16 08-34-18Screenshot from 2019-06-16 08-34-20Screenshot from 2019-06-16 08-34-22Screenshot from 2019-06-16 08-34-26Screenshot from 2019-06-16 08-34-29Screenshot from 2019-06-16 08-34-32Screenshot from 2019-06-16 08-34-40Screenshot from 2019-06-16 08-33-50Screenshot from 2019-06-16 08-33-59Screenshot from 2019-06-16 08-34-02Screenshot from 2019-06-16 08-34-06Screenshot from 2019-06-16 08-34-12  **实验二 进程同步和通信——生产者和消费者问题模拟**   1. 实验目的   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。   1. 实验原理   一、进程/线程间同步机制。  临界区、互斥区、事件、信号量四种方式  临界区（Critical Section）、互斥量（Mutex）、信号量（Semaphore）、事件（Event）的区别：  1、临界区：通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。在任意时刻只允许一个线程对共享资源进行访问，如果有多个线程试图访问公共资源，那么在有一个线程进入后，其他试图访问公共资源的线程将被挂起，并一直等到进入临界区的线程离开，临界区在被释放后，其他线程才可以抢占。  2、互斥量：采用互斥对象机制。 只有拥有互斥对象的线程才有访问公共资源的权限，因为互斥对象只有一个，所以能保证公共资源不会同时被多个线程访问。互斥不仅能实现同一应用程序的公共资源安全共享，还能实现不同应用程序的公共资源安全共享 .互斥量比临界区复杂。因为使用互斥不仅仅能够在同一应用程序不同线程中实现资源的安全共享，而且可以在不同应用程序的线程之间实现对资源的安全共享。  3、信号量：它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目 .信号量对象对线程的同步方式与前面几种方法不同，信号允许多个线程同时使用共享资源，这与操作系统中的PV操作相同。它指出了同时访问共享资源的线程最大数目。它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。  PV操作及信号量的概念都是由荷兰科学家E.W.Dijkstra提出的。信号量S是一个整数，S大于等于零时代表可供并发进程使用的资源实体数，但S小于零时则表示正在等待使用共享资源的进程数。  　　 P操作申请资源：  　　（1）S减1；  　　（2）若S减1后仍大于等于零，则进程继续执行；  　　（3）若S减1后小于零，则该进程被阻塞后进入与该信号相对应的队列中，然后转入进程调度。  　　V操作 释放资源：  　　（1）S加1；  　　（2）若相加结果大于零，则进程继续执行；  　　（3）若相加结果小于等于零，则从该信号的等待队列中唤醒一个等待进程，然后再返回原进程继续执行或转入进程调度。  4、事 件： 通过通知操作的方式来保持线程的同步，还可以方便实现对多个线程的优先级比较的操作 .  生产者和消费者的分析：  （1）关系分析：生产者和消费者对缓冲区 互斥访问是互斥关系，同时生产者和消费者又是一个相互协作的关系（只有生产者生产之后，消费者才能消费）。  （2）只有生产者和消费者两个进程，所以需要解决的是互斥和同步PV操作的位置。  （3）信号量设置：信号量mutex作为互斥信号量，用于控制互斥访问缓冲区，初值设置为1；信号量full用于记录当前缓冲区“满”，初值设置为0;信号量empty用于记录当前缓冲区“空”缓冲数，初值设置为n。   1. 实验内容（源码、注释、基础内容、扩展点等） 2. 内容要求： 3. 调试、运行给出的程序，从操作系统原理的角度验证程序的正确性 4. 发现并修改程序中的原理性错误或不完善的地方 5. 鼓励在程序中增加新的功能。完成基本 6. 在程序中适当地加入注释 7. 认真进行预习，阅读原程序，发现其中的原理性错误，完成预习报告 8. 实验完成后，要认真总结，完成实验报告 9. 所给程序模拟两个进程，即生产者（producer）进程和消费者(Consumer)进程工作 10. 生产者每次产生一个数据，送入缓冲区中 11. 消费者每次从缓冲区中取走一个数据 12. 缓冲区可以容纳8个数据 13. 因为缓冲区是有限的，因此当其满了时生产者进程应该等待；当消费者取走一个数据后，应唤醒正在等待的生产者进程 14. 当缓冲区空时，消费者进程应该等待；当生 产者向缓冲区放入了一个数据时，应唤醒正在等待的消费者进程 15. 每次写入和读出数据时，都将读和写指针加一。当指针到达缓冲区尾，重新将指针退回起点 16. 为简单起见，每次产生的数据为0-99的整数，从0开始，顺序递增 17. 两个进程的调度是通过运行者使用键盘来 实现的 18. 数据结构：   const int bufferSize = 8;  vector<int> buffer(bufferSize);  queue<int> waitQueue;  int id;  int writeptr, readptr;  int consumer, producer;   1. 程序流程图：   未命名文件 (5)   1. 源码：   #include<iostream>  #include<vector>  #include<queue>  using namespace std;  const int bufferSize = 8;  vector<int> buffer(bufferSize);  queue<int> waitQueue;  int id;  int writeptr, readptr;  int consumer, producer;  void init()  {  for (int i = 0; i < bufferSize; ++i)  {  buffer[i] = 0;  }  while (!waitQueue.empty())  {  waitQueue.pop();  }  id = 0;  writeptr = readptr = 0;  consumer = producer = 0;//the num of waiting  }  int findEmpty(vector<int> &v, int len)  {  for (int i = 0; i < len; ++i)  {  if (v[i] == 0)  return i;  }  return -1;  }  int findDirty(vector<int> &v, int len)  {  for (int i = 0; i < len; ++i)  {  if (v[i] > 0)  return i;  }  return -1;  }  void show()  {  for (int i = 0; i < bufferSize; ++i)  {  if (buffer[i] > 0)  cout << " " << buffer[i] << " |";  else  cout << " |";  }  cout << "\nwriteptr = \t" << writeptr << "\treadptr = \t" << readptr << "\t";  if (producer)  cout << "PRODUCER wait : " << producer << "\t";  else  cout << "PRODUCER ready\t\t";  if (consumer)  cout << "CONSUMER wait : " << consumer;  else  cout << "CONSUMER ready";  cout << "\n";  }  void produce()  {  if (findEmpty(buffer, bufferSize) == -1) // buffer full  {  ++id;  producer++;  waitQueue.push(id);  }  else  {  if (waitQueue.empty())  {  id++;  buffer[writeptr] = id;  }  else  {  producer--;  buffer[writeptr] = waitQueue.front();  waitQueue.pop();  }  writeptr = (writeptr + 1) % bufferSize;  }  }  void consume()  {  if (findDirty(buffer, bufferSize) == -1) // buffer empty  {  if (waitQueue.empty())  {  consumer++;  }  else  {  buffer[writeptr] = waitQueue.front();  waitQueue.pop();  writeptr = (writeptr + 1) % bufferSize;  }  }  else  {  buffer[readptr] = 0;  readptr = (readptr + 1) % bufferSize;  }  }  void work()  {  char c;  cin >> c;  while (c != 'q' && (c == 'p' || c == 'c'))  // q = quit p = produce one c = consume one  {  if (c == 'p')  {  produce();  }  else if (c == 'c')  {  consume();  }  show();  cin >> c;  }  }  int main(int argc, char const \*argv[])  {  init();  work();  return 0;  }   1. 实验结果（截图）   **Screenshot from 2019-06-16 08-27-22Screenshot from 2019-06-16 08-27-26**  **实验三 进程的管道通信**  一、实验目的  1） 加深对进程概念的理解，明确进程和程序的区别。  2） 学习进程创建的过程，进一步认识进程并发执行的实质。  3）分析进程争用资源的现象，学习解决进程互斥的方法。  4）学习解决进程同步的方法。  5）掌握Linux系统中进程间通过管道通信的具体实现。  二、实验原理  管道是一种最基本的进程间通信机制。管道由pipe函数来创建：调用pipe函数，会在内核中开辟出一块缓冲区用来进行进程间通信，这块缓冲区称为管道，它有一个读端和一个写端。  pipe函数接受一个参数，是包含两个整数的数组，如果调用成功，会通过pipefd[2]传出给用户程序两个文件描述符，需要注意pipefd [0]指向管道的读端, pipefd [1]指向管道的写端，那么此时这个管道对于用户程序就是一个文件，可以通过read(pipefd [0]);或者write(pipefd [1])进行操作。pipe函数调用成功返回0，否则返回-1.  》父进程创建管道，得到两个文件描述符指向管道的两端  IMG_256  》利用fork函数创建出子进程，则子进程也得到两个文件描述符指向同一管道  IMG_257  》父进程关闭读端（pipe[0]）,子进程关闭写端pipe[1]，则此时父进程可以往管道中进行写操作，子进程可以从管道中读，从而实现了通过管道的进程间通信。  IMG_258  三、实验内容（源码、注释、基础内容、扩展点等）  1. 内容要求：  1）使用系统调用pipe()建立一条管道，系统调用fork()分别创建两个子进程，它们分别向管道写一句话  2）父进程分别从管道读出来自两个子进程的信息，显示在屏幕上  3）这是一个设计型实验，要求自行、独立编制程序  4）两个子进程要并发执行  5）实现管道的互斥使用。当一个子进程正在对管道进行写操作时，另一个欲写入管道的子进程必须等待。 使用系统调用lockf(fd[1],1,0)实现对管道的加锁操作，用lockf(fd[1],0,0)解除对管道的锁定  6）实现父子进程的同步，当父进程试图从一空管道中读取数据时，便进入等待状态，直到子进程将数据写入管道返回后，才将其唤醒  7）fork() 用于创一个子进程。  格式：int fork();  返回值：在子进程中返回0；在父进程中返回所创建的子进程的ID值；当返回-1时，创建失败。  8）wait() 常用来控制父进程与子进程的同步。  在父进程中调用wait()，则父进程被阻塞，进入等待队列，等待子进程结束。当子进程结束时，父进程从wait()返回继续执行原来的程序。  返回值：大于0时，为子进程的ID值；等于-1时，调用失败。  9）exit() 是进程结束时最常调用的。  格式：void exit( int status); 其中，status为进程结束状态  10）pipe() 用于创建一个管道  格式：pipe(int fd);  其中fd是一个由两个数组元素fd[0]和fd[1]组成的整型  数组，fd[0]是管道的读端口，用于从管道读出数据,fd[1]是管道的写端口，用于向管道写入数据  11）sleep() 使调用进程睡眠若干时间，之后唤醒。  格式：sleep(int t); 其中t为睡眠时间  12） lockf() 用于对互斥资源加锁和解锁。在本实验中该调用的格式为：  lockf(fd[1],1,0)；/\* 表示对管道的写入端口加锁。  lockf(fd[1],0,0)；/\* 表示对管道的写入端口解锁。   1. write(fd[1],String,Length) 将字符串String的内容写入 管道的写入口 2. read(fd[0],String,Length) 从管道的读入口读出信息放入字符串String中 3. 数据结构：   int pid1, pid2, pid3;  int fd[2];  char outpipe[60], inpipe[60];   1. 实验流程图：   未命名文件 (1)未命名文件 (3)  4. 源码：  #include <stdio.h>  #include <sys/types.h>  #include <stdlib.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <error.h>  #include <wait.h>  #include <unistd.h>  int main()  {  int pid1, pid2, pid3;  int fd[2];  char outpipe[60], inpipe[60];  pipe(fd);//′′?¨ò???1üμà  while ((pid1 = fork()) == -1);  printf("pid1=%d\n", pid1);  if (pid1 == 0)  {  printf("The Child process 1 is sending message!\n");  lockf(fd[1], 1, 0);//?￥3a  sprintf(outpipe, "This is the child 1 process's message!\n");  write(fd[1], outpipe, 60);  sleep(1);//×??ò×èè?1??,è?3??ú?á?′DD??ò?????3ì￡????ó2￠·￠?è  lockf(fd[1], 0, 0);  exit(0);  }  else  {  while ((pid2 = fork()) == -1);  printf("pid2=%d\n", pid2);  if (pid2 == 0)  {  printf("The Child process 2 is sending message!\n");  lockf(fd[1], 1, 0);  sprintf(outpipe, "This is the child 2 process's message!\n");  write(fd[1], outpipe, 60);  sleep(1);  lockf(fd[1], 0, 0);  exit(0);  }  else  {  while ((pid3 = fork()) == -1);  printf("pid3=%d\n", pid3);  if (pid3 == 0)  {  printf("The Child process 3 is sending message!\n");  lockf(fd[1], 1, 0);  sprintf(outpipe, "This is the child 3 process's message!\n");  write(fd[1], outpipe, 60);  sleep(1);  lockf(fd[1], 0, 0);  exit(0);  }  else  {  wait(0);//í?2?  read(fd[0], inpipe, 60);  printf("\n%s", inpipe);  wait(0);  read(fd[0], inpipe, 60);  printf("%s\n", inpipe);  wait(0);  read(fd[0], inpipe, 60);  printf("%s\n", inpipe);  exit(0);  }  }  }  return 0;  }  四、实验结果（截图）  Screenshot from 2019-06-16 08-38-19  **实验四 页面置换算法**  一、实验目的  1）进一步加深理解父子进程之间的关系及其并发执行。  2）理解内存页面调度的机理。  3）掌握页面置换算法及其实现方法。  4）培养综合运用所学知识的能力。  5）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。  6）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。  二、实验原理  先进先出FIFO页面置换算法：  当需要淘汰一个页面时，总是选择驻留主存时间最长的页面进行淘汰，即先进入主存的页面先淘汰。优先淘汰最早进入内存的页面，亦即在内存中驻留时间最久的页面。该算法实现简单，只需把调入内存的页面根据先后次序链接成队列，设置一个指针总指向最早的页面。但该算法与进程实际运行时的规律不适应，因为在进程中，有的页面经常被访问。 假定系统为某进程分配了三个物理块，并考虑有以下页面号引用串：7, 0, 1, 2, 0, 3, 0,4,2,3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1。釆用FIFO算法进行页面置换，进程访问页面2时，把最早进入内存的页面7换出。然后访问页面3时，再把2, 0, 1中最先进入内存的页换出  最近最久未使用LRU页面置换算法：  利用局部性原理，根据一个作业在执行过程中过去的页面访问历史来推测未来的行为。它认为过去一段时间里不曾被访问过的页面，在最近的将来可能也不会再被访问。所以，这种算法的实质是：当需要淘汰一个页面时，总是选择在最近一段时间内最久不用的页面予以淘汰。 实现过程：对上面的实例釆用LRU算法进行页面置换，进程第一次对页面2访问时，将最近最久未被访问的页面7置换出去。然后访问页面3时，将最近最久未使用的页面1换出。  三、实验内容（源码、注释、基础内容、扩展点等）   1. 内容要求： 2. 程序涉及一个父进程和两个子进程。父进程使用rand()函数随机产生若干随机数，经过处理后，存于一数组Acess\_Series[]中，作为内存页面访问的序列。两个子进程根据这个访问序列，分别采用FIFO和LRU两种不同的页面置换算法对内存页面进行调度 3. 这是一个综合性实验，要求在掌握父子进程并发执行机制和内存页面置换算法的基础上，能综合运用这两方面的知识，自行、独立编制程序 4. 每个子进程应能反映出页面置换的过程，并统计页面置换算法的命中或缺页情况。 设缺页的次数为diseffect。总的页面访问次数为total\_instruction。则：缺页率 = disaffect/total\_instruction 命中率 = 1- disaffect/total\_instruction 5. 将为进程分配的内存页面数mframe作为程序的参数，通过多次运行程序，说明FIFO算法存在的Belady现象 6. 数据结构： 7. 存放页面访问序列的数组：   int Acess\_Series[total\_instruction]；  int total\_instruction; //进程总的页面数  （2）用一个结构数组M\_Frame[]记录为进程分配的内存页面的使用情况：  struct one\_frame{ //记录为进程分配的内存页面情况;  int page\_No; //记录页面号  **int time; //记录页面进入内存时间**  **int used\_time; //记录页面最近使用时间**  };  one\_frame M\_Frame[frame\_num];  int frame\_num; //驻留集大小  FIFO依据进入内存时间来置换页面、LRU依据最近使用时间来置换页面   1. 程序流程图：   未命名文件 (4)   1. 源码：   #include<stdio.h>  #include<sys/types.h>  #include<stdlib.h>  #include<sys/stat.h>  #include<fcntl.h>  #include<error.h>  #include<wait.h>  #include<unistd.h>  #include<iostream>  using namespace std;  int total\_instruction; //进程总的页面数  int frame\_num; //驻留集大小  struct one\_frame //记录为进程分配的内存页面情况;  {  int page\_No; //记录页面号  int time; //记录页面进入内存时间  int used\_time; //记录页面最近使用时间  };  int main()  {  int time1 = 0, time2 = 0; //系统时间  int page1, page2; //每次读入一个逻辑页面  int p1 = 0, p2 = 0; //内存页框指针，最大为 frame\_num  int diseffect1 = 0, diseffect2 = 0; //缺页次数  float f;//缺页率  printf("Please input the number of Page Frames:\n");  scanf("%d", &frame\_num);  one\_frame M\_Frame[frame\_num];  printf("Please input the number of Pages:\n");  scanf("%d", &total\_instruction);  int Access\_series[total\_instruction];  for(int i = 0; i < frame\_num; i++)  M\_Frame[i].page\_No = -1;  int a = 0;  printf("是否随机生成访问页序号？（1为是）\n");  scanf("%d", &a);  if(a != 1)  {  printf("Please input %d 个数字\n", total\_instruction);  for(int i = 0; i < total\_instruction; i++)  scanf("%d", &Access\_series[i]);  }  else //产生随机数,即页面分配情况  {  for(int i = 0; i < total\_instruction; i++)  Access\_series[i] = rand() % total\_instruction + 1;  }  int pc1, pc2; //两个子进程，p1执行FIFO算法，p2执行LRU算法  while((pc1 = fork()) == -1);  if(pc1 == 0)  {  for(int i = 0; i < total\_instruction; i++)  {  int flag1 = 0; //页面是否已存在于内存中的标志，1表示存在  page1 = Access\_series[i]; //读入一个逻辑页面  for(int j = 0; j < p1; j++) //查找该页面是否已存在于内存中  {  if(page1 == M\_Frame[j].page\_No)  {  for(int i = 0; i < total\_instruction; i++)  printf("%d ", Access\_series[i]);  printf("\n");  printf("%d have found!\n", page1);  M\_Frame[j].used\_time = time1;  for(int k = 0; k < p1; k++)  printf("%d ", M\_Frame[k].page\_No);  printf("\n\n");  flag1 = 1;  break;  }  }  if(flag1 == 0) //若不存在，发生缺页，缺页次数加1  {  diseffect1++;  if(p1 < frame\_num) //若内存页面没有分配满  {  M\_Frame[p1].page\_No = page1; //分配一个空的内存页面  M\_Frame[p1].time = time1; //新进入的页面要注意记录进入时间  M\_Frame[p1].used\_time = time1;  p1++;  for(int j = 0; j < total\_instruction; j++)  printf("%d ", Access\_series[j]);  printf("\n");  printf("%d diseffect!\n", page1);  for(int j = 0; j < p1; j++)  printf("%d ", M\_Frame[j].page\_No);  printf("\n\n");  }  else //内存中没有，内存又满了，只能使用FIFO算法淘汰内存中最先进入的页面  {  int min = 999, pos = 0; //找到内存中最先进入的页面，其占用的内存页面号为 pos  for(int j = 0; j < p1; j++)  {  if(M\_Frame[j].time < min)  {  min = M\_Frame[j].time;  pos = j;  }  }  int temp1 = M\_Frame[pos].page\_No;  M\_Frame[pos].page\_No = page1; //将所读的页面调入  M\_Frame[pos].time = time1; //新进入的页面要注意记录进入时间  M\_Frame[pos].used\_time = time1;  for(int j = 0; j < total\_instruction; j++)  printf("%d ", Access\_series[j]);  printf("\n");  printf("%d diseffect! lose %d\n", page1, temp1);  for(int j = 0; j < p1; j++)  printf("%d ", M\_Frame[j].page\_No);  printf("\n\n");  }  }  time1++; //系统时间增加  //cout<<"i"<<" "<<i;  }  f = diseffect1 / (total\_instruction \* 1.0); //计算缺页率  printf("diseffect rate of FIFO %f\n\n\n", f);  exit(0);  }  wait(0);  for(int i = 0; i < frame\_num; i++)  M\_Frame[i].page\_No = -1;  while((pc2 = fork()) == -1);  if(pc2 == 0)  {  for(int i = 0; i < total\_instruction; i++)  {  int flag2 = 0;  page2 = Access\_series[i];  for(int j = 0; j < p2; j++) //首先在内存中查找  {  if(page2 == M\_Frame[j].page\_No)  {  printf("%d have found!\n", page2);  M\_Frame[j].used\_time = time2; //用到的页面要注意记录最后一次使用时间  for(int k = 0; k < p2; k++)  printf("%d ", M\_Frame[k].page\_No);  printf("\n\n");  flag2 = 1;  break;  }  }  if(flag2 == 0) //没有找到，发生缺页  {  diseffect2++;  if(p2 < frame\_num) //内存页面还没有分配完  {  M\_Frame[p2].page\_No = page2;  M\_Frame[p2].time = time2;  M\_Frame[p2].used\_time = time2; //用到的页面要注意记录最后一次使用时间  p2++;  for(int j = 0; j < total\_instruction; j++)  printf("%d ", Access\_series[j]);  printf("\n");  printf("%d diseffect! \n", page2);  for(int j = 0; j < p2; j++)  printf("%d ", M\_Frame[j].page\_No);  printf("\n\n");  }  else //内存中没有，内存又满了，只能使用LRU算法淘汰内存中最近最久未使用的页面  {  int min = 999, pos = 0;  for(int j = 0; j < p2; j++)  {  if(M\_Frame[j].used\_time < min)  {  min = M\_Frame[j].used\_time;  pos = j;  }  }  int temp2 = M\_Frame[pos].page\_No;  M\_Frame[pos].page\_No = page2;  M\_Frame[pos].time = time2;  M\_Frame[pos].used\_time = time2; //用到的页面要注意记录最后一次使用时间  for(int j = 0; j < total\_instruction; j++)  printf("%d ", Access\_series[j]);  printf("\n");  printf("%d diseffect! lose %d\n", page2, temp2);  for(int j = 0; j < p2; j++)  printf("%d ", M\_Frame[j].page\_No);  printf("\n\n");  }  }  time2++;  }  f = diseffect2 / (total\_instruction \* 1.0);  printf("diseffect rate of LRU %f\n", f);  exit(0);  }  return 0;  }  四、实验结果（截图）  Screenshot from 2019-06-16 09-04-24Screenshot from 2019-06-16 09-05-50Screenshot from 2019-06-16 09-05-55Screenshot from 2019-06-16 09-04-15 | | | | | | |