**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** | | | **实验成绩** |  |
| **专业** | **计算机** | | **班级** | **计算机1507** | **指导教师签字** |  |
| **学号** | **20154414** | | **姓名** | **朱强** | **实验报告批改时间** |  |
| **实验项目目录**   1. 实验一……熟悉Linux系统 2. 实验二……三状态运行模型 3. 实验三……生产者消费者模型 4. 实验四......父子进程管道通信 5. 实验五......页面置换算法的模拟 | | | | | | |
| **实验报告正文**  **实验一 熟悉Linux**   1. 题目：熟悉Linux系统 2. 目的：   熟悉和掌握Linux系统基本命令，熟悉Linux编程环境，为以后的实验打下基础。  1、启动、退出、ls（显示目录内容）、cp（文件或目录的复制）、mv（文件、目录更名或移动）、rm（删除文件或目录）、mkdir（创建目录）、rmdir（删除空目录）、cd（改变工作目录）…  2、C语言编辑、编译   1. 内容及要求： 2. 熟练掌握Linux基本文件命令 ； 3. 掌握Linux编辑程序、对源代码进行编译、连接、运行及调试的过程 ； 4. 认真做好预习，书写预习报告 ； 5. 实验完成后要认真总结、完成实验报告。   四、内容及要求：  在Linux环境下编制、调试源程序的实际过程  **实验2 进程状态转换及其PCB的变化**  **1．目的：**  自行编制模拟程序，通过形象化的状态显示，深入理解进程的概念、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。  **2. 内容及要求：**   1. 设计并实现一个模拟进程状态转换及其相应PCB内容、组织结构变化的程序。 2. 独立编写、调试程序。进程的数目、进程的状态模型（三状态、五状态、七状态或其它）以及PCB的组织形式可自行选择。 3. 合理设计与进程PCB相对应的数据结构。PCB的内容要涵盖进程的基本信息、控制信息、资源需求及现场信息。 4. 设计出可视性较好的界面，应能反映出进程状态的变化引起的对应PCB内容、组织结构的变化。 5. 代码书写要规范，要适当地加入注释。   6） 认真进行预习，完成预习报告。  7） 实验完成后，要认真总结，完成实验报告。  **3．使用的数据结构及说明：**  在本实验中，主要用到的数据结构是PCB的结构，其中PCB的数据结构如下：enum Status  {  Running,  Ready,  Blocked,  Exit  };  struct PCB  {  int id; //进程号  int priority; //优先级  enum Status status; //进程状态  char name; //进程名称  };  struct Running  {  struct PCB pcb;  struct Running \*pcbnext;  };  struct Running \*header\_running;  struct Ready  {  struct PCB pcb;  struct Ready \*pcbnext;  };  struct Ready \*header\_ready, \*tail\_ready;  struct Blocked  {  struct PCB pcb;  struct Blocked \*pcbnext;  };  struct Blocked \*header\_blocked, \*tail\_blocked;  struct Exit  {  int id;  char name;  struct Exit \*pcbnext;  };  struct Exit \*header\_exit,\*tail\_exit;   1. **流程图**      1. **程序源代码，注释及说明文字：** 2. **#include<iostream>** 3. **#include<vector>** 4. **using namespace std;** 5. **//输出各个运行状态的表格** 6. **void print( vector<char>running,int front\_running,vector<char>ready,int front\_ready, vector<char> block,int front\_block)** 7. **{** 8. **cout<<"runing: ";** 9. **for(;front\_running!=running.size();front\_running++)** 10. **{** 11. **cout<<running[front\_running]<<" ";** 12. **}** 13. **cout<<endl<<"ready: ";** 14. **for(;front\_ready!=ready.size();front\_ready++)** 15. **{** 16. **cout<<ready[front\_ready]<<" ";** 17. **}** 18. **cout<<endl<<"block: ";** 19. **for(;front\_block!=block.size();front\_block++)** 20. **{** 21. **cout<<block[front\_block]<<" ";** 22. **}** 23. **cout<<endl;** 24. **}** 25. **//初始化** 26. **void Intialize(vector<char>&running,vector<char>&ready,vector<char>&block)** 27. **{** 28. **running.push\_back('A');** 29. **for(int i=0;i<=6;i++)** 30. **ready.push\_back('B'+i);** 31. **for(int i=0;i<=6;i++)** 32. **block.push\_back('I'+i);** 33. **}** 34. **int main()** 35. **{** 36. **vector<char> running;** 37. **vector<char> ready;** 38. **vector<char> block;** 39. **Intialize( running,ready,block);** 40. **cout<<"Intialize:"<<endl;** 41. **int front\_running=0;** 42. **int front\_ready=0;** 43. **int front\_block=0;** 44. **print( running, front\_running,ready,front\_running,block,front\_block);** 45. **cout<<"please input i:"<<endl;** 46. **int i;** 47. **cin>>i;** 48. **while(i)//只要i不等于0 0跳出循环** 49. **{** 50. **if(i==2)** 51. **{** 52. **ready.push\_back( running[front\_running++]);** 53. **running.push\_back( ready[front\_ready++] );** 54. **cout<<"running->ready: "<<endl;** 55. **print( running,front\_running, ready,front\_ready,block,front\_block);** 56. **}** 57. **else if(i==3)//running->block** 58. **{** 59. **if(front\_running==running.size())** 60. **{** 61. **cout<<"running queue is empty"<<endl;** 62. **print(running,front\_running,ready,front\_ready,block,front\_block);** 63. **}** 64. **else** 65. **{** 66. **block.push\_back(running[front\_running++]);** 67. **if(front\_ready==ready.size())** 68. **{** 69. **cout<<"ready queue is empty!"<<endl;** 70. **print(running,front\_running,ready,front\_ready,block,front\_block);** 71. **}** 72. **else {** 73. **running.push\_back(ready[front\_ready++]);** 74. **cout<<"running->block"<<endl;** 75. **print( running,front\_running, ready,front\_ready,block,front\_block);** 76. **}** 77. **}** 78. **}** 79. **else if(i==4)//block->ready** 80. **{** 81. **if(front\_running==running.size() && front\_ready==running.size())** 82. **{** 83. **running.push\_back(block[front\_block++]);** 84. **print( running,front\_running, ready,front\_ready,block,front\_block);** 85. **}** 86. **else {** 87. **ready.push\_back(block[front\_block++]);** 88. **if(front\_running==running.size()) //如果此时running无进程占用** 89. **running.push\_back(ready[front\_ready++]);** 90. **cout<<"block->ready"<<endl;** 91. **print( running,front\_running, ready,front\_ready,block,front\_block);** 92. **}** 93. **}** 94. **else** 95. **cout<<"ERROR!"<<endl;** 96. **cout<<"input i:"<<endl;** 97. **cin>>i;** 98. **}** 99. **return(0);** 100. **}** 101. **运行结果:**   101112  程序使用说明  0---创建进程 1---就绪-运行  2---运行-就绪 3---运行到阻塞  4---阻塞-就绪 5---运行-终止  **实验3 进程同步和通信-生产者和消费者问题模拟**  **1. 目的：**  调试、修改、运行模拟程序，通过形象化的状态显示，使学生理解进程的概念，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，使学生能灵活运用相关知识，培养创新能力。  **2. 内容及要求：**  1) 调试、运行模拟程序。  2) 发现并修改程序中不完善的地方。  3) 修改程序，使用随机数控制创建生产者和消费者的过程。  4) 在原来程序的基础上，加入缓冲区的写互斥控制功能，模拟多个进程存取一个公共缓冲区，当有进程正在写缓冲区时，其他要访问该缓冲区的进程必须等待，当有进程正在读取缓冲区时，其他要求读取的进程可以访问，而要求写的进程应该等待。  5) 完成1)、2)、3）功能的,得基本分,完成4)功能的加2分,有其它功能改进的再加2分  **3. 程序说明：**  　　本程序是模拟两个进程，生产者（producer）和消费者(Consumer)工作。生产者每次产生一个数据，送入缓冲区中。消费者每次从缓冲区中取走一个数据。缓冲区可以容纳8个数据。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，而空时，消费者进程应该等待；当生产者向缓冲区放入了一个数据，应唤醒正在等待的消费者进程，同样，当消费者取走一个数据后，应唤醒正在等待的生产者进程。就是生产者和消费者之间的同步。  　　每次写入和读出数据时，都将读和写指针加一。当读写指针同样时，又一起退回起点。当写指针指向最后时，生产者就等待。当读指针为零时，再次要读取的消费者也应该等待。  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。两个进程的调度是通过运行者使用键盘来实现的。  **4. 程序使用的数据结构**  进程控制块：包括进程名，进程状态和执行次数。  缓冲区：一个整数数组。  缓冲区说明块：包括类型，读指针，写指针，读等待指针和写等待指针。  **5. 程序使用说明**  　　启动程序后，如果使用'p'键则运行一次生产者进程，使用'c'键则运行一次消费者进程。通过屏幕可以观察到两个进程的状态和缓冲区变化的情况。   1. **实验流程图**       **源程序**  t();  int main()  {  char in[3];  writeptr = 0;  readptr = 0;  writein = 0;  empty = PIPESIZE;  full = 0;  producer = (struct PCB \*)malloc(sizeof(struct PCB));  consumer = (struct PCB \*)malloc(sizeof(struct PCB));  headerqueue = (struct waitqueue \*)malloc(sizeof(struct waitqueue));  headerqueue->next = NULL;  tailqueue = headerqueue;  producer->status = READY;  consumer->status = WAIT;  producer->time = consumer->time = 0;  printf("Now starting the program!\n");  printf("Press 'p1' to run PRODUCER1,Press 'p1' to run PRODUCER2 \n");  printf("Press 'c1' to run CONSUMER1,Press 'c2' to run CONSUMER2 \n");  printf("Press 'e' to exit from the program.\n");  while (1)  {  strcpy(in, "N");  while (!strcmp(in, "N"))  {  printf("\n");  scanf("%s", in);  if (strcmp(in, "e") && strcmp(in, "p1") && strcmp(in, "p2") && strcmp(in, "c1") && strcmp(in, "c2"))  {  printf("error,please input again!!!\n");  strcpy(in, "N");  }  }  if ('p' == in[0])  {  runp(in);  producer->time++;  }  else if ('c' == in[0])  {  runc(in);  consumer->time++;  }  else  {  //printf("PRODUCER product %d times\n", producer->time);  //printf("CONSUMER consumer %d times\n", consumer->time);  exit(0);  }  print();  printf("\n");  }  }  void runp(char in[3])  {  if (full >= 8)  {  struct waitqueue \*search;  search = headerqueue->next;  while (search != NULL)  {  if (!strcmp(in, search->pcb.name))  {  printf("error!!!\n");  exit(1);  }  search = search->next;  }  producer->status = WAIT;  printf("PRODUCER %s process is waiting, can't be scheduled.\n", in);  struct waitqueue \*p = (struct waitqueue \*)malloc(sizeof(struct waitqueue));  strcpy(p->pcb.name, in);  p->pcb.status = WAIT;  p->pcb.time = producer->time + 1;  p->next = NULL;  tailqueue->next = p;  tailqueue = p;  full++;  }  else  {  writein = (writein + 1) % 100;  producer->status = RUN;  printf("run PRODUCER %s process . product %d ", in, writein);  buffer[writeptr] = writein;  if (empty>8)  {  struct waitqueue \*p;  p = headerqueue->next;  printf("run CONSUMER %s process. use %d", p->pcb.name, buffer[writeptr]);  if (tailqueue == p)  {  tailqueue = headerqueue;  }  headerqueue->next = p->next;  free(p);  consumer->status = WAIT;  empty--;  }  else  {  if (writeptr > readptr)  {  writeptr++;  if (writeptr >= PIPESIZE)  {  writeptr = 0;  if (readptr == 0)  producer->status = WAIT;  else producer->status = READY;  }  }  else  {  writeptr++;  if (writeptr == readptr)  producer->status = WAIT;  else producer->status = READY;  }  consumer->status = READY;  empty--;  full++;  }  }  }  void runc(char in[3])  {  if (empty >= 8)  {  struct waitqueue \*search;  search = headerqueue->next;  while (search != NULL)  {  if (!strcmp(in, search->pcb.name))  {  printf("error!!!\n");  exit(1);  }  search = search->next;  }  consumer->status = WAIT;  printf("CONSUMER %s is waiting, can't be scheduled.\n",in);  struct waitqueue \*p = (struct waitqueue \*)malloc(sizeof(struct waitqueue));  strcpy(p->pcb.name, in);  p->pcb.status = WAIT;  p->pcb.time = consumer->time + 1;  p->next = NULL;  tailqueue->next = p;  tailqueue = p;  empty++;  }  else  {  consumer->status = RUN;  readout = buffer[readptr];  printf("run CONSUMER %s process. use %d ", in, readout);  if (full>8)  {  writein = (writein + 1) % 100;  buffer[writeptr] = writein;  struct waitqueue \*p;  p = headerqueue->next;  printf("run PRODUCER %s process. product %d ", p->pcb.name, buffer[writeptr]);  if (tailqueue == p)  {  tailqueue = headerqueue;  }  headerqueue->next = p->next;  free(p);  producer->status = WAIT;  full--;  writeptr++;  readptr++;  }  else  {  if (readptr > writeptr)  {  readptr++;  if (readptr >= PIPESIZE)  {  readptr = 0;  if (writeptr == 0)  consumer->status = WAIT;  else  consumer->status = READY;  }  else consumer->status = READY;  }  else  {  readptr++;  if (readptr == writeptr)  {  consumer->status = WAIT;  writeptr = readptr = 0;  }  else consumer->status = READY;  producer->status = READY;  }  full--;  empty++;  }  }  }  void print()  {  int i = 0;  int j = 0;  int low = 0;  int high = 0;  printf("\n");  for (i; i < PIPESIZE; i++)  printf("--------");  printf("\n");  if (readptr < writeptr)  {  for (low = 0; low < readptr; low++)  printf("| |");  for (low = readptr; low < writeptr; low++)  {  printf("| %2d |", buffer[low]);  }  for (low = writeptr; low <PIPESIZE; low++)  printf("| |");  }  else if (readptr > writeptr)  {  for (low = 0; low < writeptr; low++)  printf("| %2d |", buffer[low]);  for (low = writeptr; low < readptr; low++)  printf("| |");  for (low = readptr; low < PIPESIZE; low++)  {  printf("| %2d |", buffer[low]);  }  }  else if (producer->status == WAIT)  {  for (low = 0; low < PIPESIZE; low++)  {  printf("| %2d |", buffer[low]);  }  }  else  {  for (low = 0; low < PIPESIZE; low++)  {  printf("| |");  }  }  printf("\n");  for (i = 0; i < PIPESIZE; i++)  printf("--------");  printf("\n");  if (producer->status == WAIT && NULL !=headerqueue->next)  {  struct waitqueue \*p = headerqueue->next;  while (NULL != p)  {  printf(" PRODUCER %s process are waitting\n", p->pcb.name);  p = p->next;  }  }  else  printf("PRODUCER ready \n");  if (consumer->status == WAIT &&headerqueue->next)  {  struct waitqueue \*p = headerqueue->next;  while (NULL != p)  {  printf(" CONSUMER %s process are waitting\n", p->pcb.name);  p = p->next;  }  }  else  printf("CONSUMER ready\n ");  }  实验结果：  1234  **实验4 进程的管道通信**  **1. 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） 进一步认识并发执行的实质。  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） 了解Linux系统中进程通信的基本原理。  　　进程是操作系统中最重要的概念，贯穿始终，也是学习现代操作系统的关键。通过本次实验，要求理解进程的实质和进程管理的机制。在Linux系统下实现进程从创建到终止的全过程，从中体会进程的创建过程、父进程和子进程之间的关系、进程状态的变化、进程之间的互斥、同步机制、进程调度的原理和以管道为代表的进程间的通信方式的实现。  **2. 内容及要求：**  　　这是一个设计型实验，要求自行编制程序。  　　使用系统调用pipe()建立一条管道，两个子进程分别向管道写一句话：  　　Child process1 is sending a message!  　　Child process2 is sending a message!  　　父进程从管道读出来自两个子进程的信息，显示在屏幕上。  　　要求：  1） 父进程先接收子进程1发来的消息，然后再接收子进程2发来的消息。  2） 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。  3） 实现父子进程的同步，当子进程把数据写入管道后，便去睡眠等待；当父进程试图从一空管道中读取数据时，也应等待，直到子进程将数据写入管道后，才将其唤醒。  **3．相关的系统调用**  1） 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。  2） 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。  3） 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。  4） 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。  返回值：0 调用成功；-1 调用失败。  5） sleep() 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。  6） 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。  7） write(fd[1],String,Length) 将字符串String的内容写入管道的写入口。  8） read(fd[0],String,Length) 从管道的读入口读出信息放入字符串String中。  **4.程序流程**  父进程：  1） 创建管道；  2） 创建子进程1；  3） 创建子进程2；  4） 等待从管道中读出子进程1写入的数据，并显示在屏幕上；  5） 等待从管道中读出子进程2写入的数据，并显示在屏幕上；  6） 退出。  子进程：  1） 将管道的写入口加锁；  2） 将信息“Child process n is sending message!”输入到变量OutPipe中，n=1，2；  3） 将OutPipe中信息写入管道；  4） 睡眠等待；  5） 将管道的写入口解锁；  6） 退出。  **5．预习报告要求：**  　　1）题目，目的，要求  　　2）初步的程序流程图  　　3）初步的程序源代码、文档注释及必要的文字说明  　　4）预期的程序运行结果    **6．实验报告要求：**  1）题目，目的，内容，要求  2）程序流程图  3）程序源代码、文档注释及文字说明  4） 运行结果及其说明  5） 回答以下问题：  ① 指出父进程与两个子进程并发执行的顺序，并说明原因。  ② 若不对管道加以互斥控制，会有什么后果？  ③ 说明你是如何实现父子进程之间的同步的。   1. **流程图**      1. **源程序**   #include <unistd.h>  #include <signal.h>  #include <stdio.h>  #include <stdlib.h>  #include <sys/types.h>  #include <wait.h>  int pid1, pid2;  int main()  {  int fd[2];  char outpipe[100], inpipe[100];  pipe(fd);  while ((pid1 = fork()) == -1);  if (pid1 == 0)  {  lockf(fd[1], 1, 0);  sprintf(outpipe, "child 1 process is sending message!");  /\*把串放入数组outpipe中 \*/  write(fd[1], outpipe, 50);  sleep(5);  lockf(fd[1], 0, 0);  exit(0);  }  else  {  while ((pid2 = fork()) == -1);  if (pid2 == 0)  {  lockf(fd[1], 1, 0);  sprintf(outpipe, "child 2 process is sending message!");  write(fd[1], outpipe, 50);  sleep(5);  lockf(fd[1], 0, 0);  exit(0);  }  else  {  wait(0); /\*同步 \*/  read(fd[0], inpipe, 50);  printf("%s\n", inpipe);  wait(0);  read(fd[0], inpipe, 50);  printf("%s\n", inpipe);  exit(0);  }  }  }   1. **程序运行结果**     **实验4页面置换算法的实现：**  **进一步理解父子进程之间的关系**  **理解内存页面调度的机理**  **掌握页面置换算法的实现方法**  **通过实验比较不同调度算法的优劣**  **培养综合运用所学知识的能力**  **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，比较各种置换算法的效率及优缺点，从而了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。**  **这是一个综合型实验，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，自行编制程序**  **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。要求：**  **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。**  **sleep() 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**  **rand() 返回一个随机整数。需要包含文件<stdlib.h>**  **父进程：**  **随机产生内存访问页面序列，存于数组Acess\_Series[total\_instruction]中 ；**  **数据结构M\_Frame的初始化；**  **分别创建两个子进程；**  **等待子进程执行结束，退出。**  **子进程：**  **1.读页面访问序列Acess\_Series[]，若序列中已无下一个元素，转5）；否则取出序列中的下一个元素作为下次要访问的页面；**  **2.如果待访问的页面在内存中（即在M\_Frame[]中找到），则不发生缺页，命中率加1，转1)，注意LRU算法中要调整该页在数组中的位置；**  **3.否则就要将这页调入内存，通过修改相应的数据结构M\_Frame[]反映出来。首先看M\_Frame[]中有无空闲页面，如果有，将待访问页面的页号以及被占用的信息写入数组中适当位置，如要统计缺页情况，缺页次数diseffect加1，返回1）；**  **4.如果M\_Frame[]中的所有页面均被占满，则淘汰M\_Frame[0]，装入待访问页，重新调整各页面在数组中的位置。如要统计缺页情况，缺页次数diseffect加1，返回1）；**  **5.所有页面均已访问完成，统计命中率或缺页率；**  **源码：**  #include <stdio.h>  #include <unistd.h>  #include <sys/types.h>  #include <string.h>  #include <wait.h>  #include <stdlib.h>  #include <time.h>  int loc\_1; //栈底指针  #define frame\_num 3 //主存大小  #define Total\_instruction 6 //总的额访问页数  int loc\_2;//FIFO未满时候的队头指针  //命中和未命中  int is\_hit;  #define HIT 1  #define MISS 0  //页面是否在内存的标志  #define IN '1'  #define OUT '0'  //存放页面序列的数组  int Access\_Series[Total\_instruction]; //(1)  //页表  struct one\_frame  {  int page\_no; //对应的页号  char flag; //是否在内存的标志  };  struct one\_frame M\_Frame[frame\_num]; //(2)  int disaffect; //缺页的数量  //初始化函数 每个进程开始均调用  void init()  {  loc\_1 = 0;  is\_hit = MISS; //开始未命中  disaffect = 0;  for (int index = 0; index < frame\_num; index++)  M\_Frame[index].flag = OUT; //页表初始化 均不在页表中  for(int i=0;i<frame\_num;i++)  M\_Frame[i].page\_no=-1;//初始化-1  }  //输出访问序列  void print\_Access\_Series()  {  printf("Access Series:\n");  for (int index = 0; index < Total\_instruction; index++)  printf("%d ", Access\_Series[index]);  printf("\n");  }  //输出在内存的页面  void print\_M\_Frame()  {  for (int i = 0; i < frame\_num; i++)  {  if (M\_Frame[i].page\_no!=-1)  printf("%d ", M\_Frame[i].page\_no);  }  printf("\n");  }  //输出缺页次数和缺页率  void print\_disaffect(int dis)  {  float disaffect\_rate=(float)dis/Total\_instruction;  printf("disaffect=%d\ndisafffect\_rate=%f",disaffect,disaffect\_rate);  }  //LRU命中函数 页表M\_Frame0优先级最高 [0]-[index-1]一次向后移动一位  void hit(int index)  {  printf("运行命中函数\n");  int temp; //用于交换  temp = M\_Frame[index].page\_no;  for (int i = index; i >0; i--)  {  M\_Frame[i].page\_no = M\_Frame[i-1].page\_no;  }  M\_Frame[0].page\_no = temp;  printf("\n");  }  //LRU 未命中函数 调入  void miss\_call\_in(int page)  {  if (loc\_1 < frame\_num) //此时的内存未满  {  //全体下移动  for (int i=loc\_1; i>0; i--)  {  M\_Frame[i].flag=IN;  M\_Frame[i].page\_no=M\_Frame[i-1].page\_no;  }  M\_Frame[0].flag = IN;  M\_Frame[0].page\_no = page;//移动完赋值  loc\_1++;  }  else  {  //满了就全体后移 栈顶赋值  for (int i = frame\_num-1; i>0; i--)  M\_Frame[i].page\_no= M\_Frame[i-1].page\_no;  M\_Frame[0].page\_no = page;  }  }  //FIFO  void FIFO(int page\_n)  {  printf("运行FIFO函数\n");  // 全体依次后移  for(int i=frame\_num-1 ;i>0 ;i--)  {  M\_Frame[i].page\_no=M\_Frame[i-1].page\_no;  }  M\_Frame[0].page\_no=page\_n;  M\_Frame[0].flag=IN;  //改变是否在内存的状态位  for(int i=0;i<loc\_2;i++)  {  M\_Frame[i].flag=IN;//在内存的依次赋值为在内存  }  if(loc\_2<=(frame\_num-1))  loc\_2++;  }  int main()  {  srand(time(NULL));  for (int i = 0; i < Total\_instruction; i++) //随机产生访问页面序列  Access\_Series[i] = rand() % Total\_instruction;  pid\_t id1 = fork();  if (id1 == 0) //子进程1 LRU int Access\_Series[Total\_instruction]  {  printf("\nrun LRU:\n");  init();  print\_Access\_Series();  for (int cur = 0; cur < Total\_instruction; cur++)  {  is\_hit=MISS;  int index;  for ( index = 0; index < frame\_num; index++) //遍历内存 M\_Frame  {  if (M\_Frame[index].flag == IN && M\_Frame[index].page\_no == Access\_Series[cur]) //在内存 且 页号命中  {  // printf(" M\_Frame[index].page\_no == Access\_Series[cur]%d %d ",M\_Frame[index].page\_no,Access\_Series[cur]);  is\_hit = HIT;  break;  }  }  if (is\_hit == HIT)  {  //printf("hit\_index=%d " ,index);  hit(index);  printf("hit ");  print\_M\_Frame();  }  else //未命中  {  miss\_call\_in(Access\_Series[cur]);  printf("miss ");  print\_M\_Frame();  disaffect++;  }  }  print\_disaffect(disaffect);  }  else  {  pid\_t id2 = fork(); //子进程2  if (id2 == 0)//FIFO  {  printf("run FIFO:\n");  init();  print\_Access\_Series();  for (int cur = 0; cur < Total\_instruction; cur++)  {  is\_hit=MISS;  int index;  for(int index=0;index<frame\_num;index++)//遍历页表  {  if (M\_Frame[index].page\_no!=-1&&M\_Frame[index].page\_no == Access\_Series[cur]) //在内存 且 页号命中  {  is\_hit = HIT;  break;  }  }  if(is\_hit==MISS)//miss  {    FIFO(Access\_Series[cur]);//进队列  disaffect++;  }  print\_M\_Frame();  }  print\_disaffect(disaffect);  }  else  {  wait(0);  wait(0);  printf("\nrun is over\n");  }  }  return (0);  } | | | | | | |