**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** | **计算机1606** | **指导教师签字** |  |
| **学号** | **20164625** | | **姓名** | **戚子强** | **实验报告批改时间** |  |
| **实验项目目录**   1. 实验一 熟悉Linux系统及模拟进程状态转换及其PCB的变化 2. 实验二 进程同步和互斥 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 熟悉Linux系统及模拟进程状态转换及其PCB的变化**   1. 实验目的   1. 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。  2. 自行编制模拟程序，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   进程都是由一系列操作(动作)所组成，通过这些操作来完成其任务。因此，不同的进程，其内部操作也不相同。在操作系统中，描述一个进程除了需要程序和私有数据之外，最主要的是需要一个与动态过程相联系的数据结构，该数据结构用来描述进程的外部特性(名字、状态等)以及与其它进程的联系(通信关系)等信息，该数据结构称为进程控制块(PCB，Process Control Block)。进程控制块PCB与进程一一对应，PCB中记录了系统所需的全部信息、用于描述进程情况所需的全部信息和控制进程运行所需的全部信息。因此，系统可以通过进程的PCB来对进程进行管理。   1. 实验内容（源码、注释、基础内容、扩展点等）   基础内容：  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；独立设计、编写、调试程序；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。  **拓展点：**  **实现了五状态进程转换，相比于三状态增加了创建和释放状态，并设置了内存限制。**  源码及注释：  #include <iostream>  #include <queue>  #include <string>  using namespace std;  class Pcb  {  public:  string name;  char status;  int memory;  Pcb(string n, char s, int m)  {  name = n;  status = s;  memory = m;  }  };  queue<Pcb \*> pCreat, pReady, pRunning, pBlocked, pRelese;  int mAll, mCreat, mReady, mRunning, mSum = 0;  void output(queue<Pcb \*> toOutq)  {  queue<Pcb \*> outq = toOutq;  while (!outq.empty())  {  cout << outq.front()->name << "|";  outq.pop();  }  cout << endl;  }  void outputAll()  {  cout << "Creat:";  output(pCreat);  cout << "Ready:";  output(pReady);  cout << "Running:";  output(pRunning);  cout << "Blocked:";  output(pBlocked);  cout << "Release:";  output(pRelese);  }  bool dispatch()  {  if (pReady.empty())  {  cout << "Ready has no pcb!";  }  else  {  pRunning.push(pReady.front());  pReady.pop();  }  return true;  }  bool creatPcb()  {  string n;  int m;  cout << "Please input process (name memory):";  cin >> n >> m;  Pcb \*p = new Pcb(n, 'C', m);  if (mSum + m <= mAll)  {  pReady.push(p);  }  else  {  pCreat.push(p);  }  return true;  }  bool ioReq()  {  if (pRunning.empty())  {  cout << "Running has no process!" << endl;  }  else  {  pBlocked.push(pRunning.front());  pRunning.pop();  dispatch();  }  return true;  }  bool timeout()  {  if (pRunning.empty())  {  cout << "Running has no process!" << endl;  }  else  {  pReady.push(pRunning.front());  pRunning.pop();  dispatch();  }  return true;  }  bool ioFinish()  {  if (pBlocked.empty())  {  cout << "Blocked has no process!" << endl;  }  else  {  pReady.push(pBlocked.front());  pBlocked.pop();  }  if(pRunning.empty())  {  dispatch();  }  return true;  }  bool release()  {  if (pRunning.empty())  {  cout << "Running has no process!" << endl;  }  else  {  pRelese.push(pRunning.front());  mSum -= pRunning.front()->memory;  pRunning.pop();  dispatch();  if (!pCreat.empty() && mSum + pCreat.front()->memory <= mAll)  {  pReady.push(pCreat.front());  pCreat.pop();  }  }  return true;  }  int main()  {  /\*初始化进程\*/  int pNum;  cout << "Please input num of processes:";  cin >> pNum;  cout << "Please input memory:";  cin >> mAll;  int a = 2;  while (pNum--)  {  string n;  char s;  int m;  cout << "Please input every process (name status memory):";  cin >> n >> s >> m;  Pcb \*p = new Pcb(n, s, m);  mSum += m;  //cout<<p->name;  switch (s)  {  case 'R':  {  pReady.push(p);  break;  }  case 'B':  {  pBlocked.push(p);  break;  }  case 'U':  {  pRunning.push(p);  break;  }  default:  {  cout << "ERROR";  break;  }  }  }  outputAll();  //cout << pReady.front()->name << endl;  int order;  while (cin >> order)  {  switch (order)  {  //Creat a Process  case 1:  {  creatPcb();  break;  }  //IO Request  case 2:  {  ioReq();  break;  }  //IO Finish  case 3:  {  ioFinish();  break;  }  // Timeout  case 4:  {  timeout();  break;  }  //Release  case 5:  {  release();  break;  }  default:  break;  }  outputAll();  }  return 0;  }  四、实验结果（截图）    **实验二 进程同步和互斥**   1. 实验目的   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。   1. 实验原理   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； 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。   1. 实验内容（源码、注释、基础内容、扩展点等）   基础内容：  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； 发现并修改程序中的原理性错误或不完善的地方；鼓励在程序中增加新的功能。完成基本功能的， 得基本分；添加新功能的加分；在程序中适当地加入注释；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；实验完成后，要认真总结，完成实验报告。  源码及注释：  #include <cstdio>  #include <cstdlib>  #include <deque>  #define PIPESIZE 8 //缓冲区大小为PIPESIZE  #define RUN 0 /\* status of process \*/  #define WAIT 1 /\* status of process \*/ //阻塞态  #define READY 2 /\* status of process \*/  #define FINISH 3 /\* status of process \*/  using namespace std;  struct pcb  { //pcb  char \*name;  int status;  int time; /\* times of execution \*/  };  struct pipetype  { //记录缓冲区状况的表  int num\_data; //缓冲区数据数目  int producer\_ptr; //写指针  int consumer\_ptr; //读指针  deque<int> prod\_wait; //生产者等待队列（保存的是生产的数据）  int consu\_waite\_size; //消费者等待队列，只记录消费者等待数目  };  pipetype pipetb;  int pipe[PIPESIZE]; //缓冲区  void runp(int output, pcb &p, int \*pipe, pipetype &pipetb);  void runc(pcb &p, int \*pipe, pipetype &pipetb);  void print(pcb &p, int \*pipe, pipetype &pipetb);  int main()  {  int output = 0; //生产者生产的数据  for (int i = 0; i < PIPESIZE; i++)  pipe[i] = -1;  char ch = 'n';  printf("Now starting the program!\n");  printf("Press 'p' to run PRODUCER, press 'c' to run CONSUMER.\n");  printf("Press 'e' to exit from the program.\n");  for (int i = 0; i < 1000; i++)  {  while (ch == 'n')  {  scanf("%c", &ch);  if (ch != 'p' && ch != 'c' && ch != 'e')  {  printf("Please enter again\n");  ch = 'n';  }  }  switch (ch)  {  case 'p':  {  pcb process = {"PRODUCER", READY, 0}; //产生一个生产者进程并去执行 最后调用print函数  output = (output + 1) % 100;  runp(output, process, pipe, pipetb);  print(process, pipe, pipetb);  break;  }  case 'c':  {  pcb process = {"CONSUMER", READY, 0}; //产生一个消费者进程并去执行 最后调用print函数  runc(process, pipe, pipetb);  print(process, pipe, pipetb);  break;  }  case 'e':  exit(0);  }  ch = 'n';  }  }  void runp(int output, pcb &p, int \*pipe, pipetype &pipetb)  { /\* run producer \*/  p.status = RUN;  printf("run PRODUCER. product %d.\t", output);  if (pipetb.num\_data == PIPESIZE)  { //相当于P(empty);  pipetb.prod\_wait.push\_back(output);  printf("The buffer is full. The PRODUCE is BLOCKED. Now there is %d producers blocked.\n", pipetb.prod\_wait.size());  }  else  { //如果P通过，把数据放到缓冲区里  pipe[pipetb.producer\_ptr] = output;  pipetb.num\_data++;  //pipetb->producer\_ptr++;  pipetb.producer\_ptr = (pipetb.producer\_ptr + 1) % PIPESIZE;  p.status = FINISH;  printf("The PRODUCE is END.\n");  p.time++;  }  if (pipetb.consu\_waite\_size)  { //相当于V(full);唤醒消费者进程  pipetb.consu\_waite\_size--;  printf("A CONSUMER is WAKEUP and RUNNING. Consuming: %d. Now there is %d consumers blocked.\n", output, pipetb.consu\_waite\_size);  pipe[pipetb.consumer\_ptr] = -1;  pipetb.consumer\_ptr = (pipetb.consumer\_ptr + 1) % PIPESIZE;  pipetb.num\_data--;  }  }  void runc(pcb &p, int \*pipe, pipetype &pipetb)  { /\* run consumer \*/  int c;  p.status = RUN;  printf("run CONSUMER.\t");  if (pipetb.num\_data == 0)  { //相当于P(full);  pipetb.consu\_waite\_size++;  printf("The buffer is empty. The CONSUMER is BLOCKED. Now there is %d consumcers blocked.\n", pipetb.consu\_waite\_size);  }  else  { //如果P通过，从缓冲区拿一个数据，顺序为从前向后循环  c = pipe[pipetb.consumer\_ptr];  pipe[pipetb.consumer\_ptr] = -1;  //pipetb->consumer\_ptr++;  pipetb.consumer\_ptr = (pipetb.consumer\_ptr + 1) % PIPESIZE;  pipetb.num\_data--;  p.status = FINISH;  printf("Use %d. The CONSUMER is END.\n", c);  p.time++;  }  if (pipetb.prod\_wait.size())  { //相当于V(empty);唤醒生产者进程  pipe[pipetb.producer\_ptr] = pipetb.prod\_wait.front();  pipetb.num\_data++;  pipetb.prod\_wait.pop\_front();  printf("A PRODUCER is WAKEUP and RUNNING. Now there is %d producers blocked.\n", pipetb.prod\_wait.size());  pipetb.producer\_ptr = (pipetb.producer\_ptr + 1) % PIPESIZE;  }  }  void print(pcb &p, int \*pipe, pipetype &pipetb)  { //打印显示缓冲区情况、读写指针、数据数目  int i;  printf(" ");  for (i = 0; i < PIPESIZE; i++)  printf("------ ");  printf("\n |");  for (i = 0; i < PIPESIZE; i++)  if (pipe[i] != -1)  printf(" %2d |", pipe[i]);  else  printf(" |");  printf("\n ");  for (i = 0; i < PIPESIZE; i++)  printf("------ ");  printf("\nproducer\_ptr = %d, consumer\_ptr = %d, ", pipetb.producer\_ptr, pipetb.consumer\_ptr);  printf("Number of data : %d\n", pipetb.num\_data);  printf("\n");  }   1. 实验结果（截图）     **实验三 进程的管道通信**  **一、**实验目的  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系统进程间通过管道通信的具 体实现方法。  二、实验原理  处理系统调用：OS中有一张系统调用入口 表，表中每个表目对应一条系统调用命令， 包含该系统调用自带参数的数目、系统调用 命令处理程序的入口地址等。OS内核便是 根据所输入的系统调用号在该表中查找到相 应的系统调用，进而转入它的入口地址去执 行系统调用程序。Linux的系统调用机制：通过中断机制实现。  三、实验内容（源码、注释、基础内容、扩展点等）  基础内容： 使用系统调用pipe()建立一条管道线，两 个子进程分别向管道写一句话（写的内 容自己定，但要有该进程的一些信息） 父进程从管道中读出来自两个子进程的 消息，显示在屏幕上。 要求：父进程首先接收子进程p1发来的 消息，然后再接收子进程p2发来的消息。  **拓展点：**  **尝试父进程先于子进程结束，即对管道先读后写，读失败抛出错误。**  源码及注释：  //正常次序读写  #include <unistd.h>  #include <sys/types.h>  #include <errno.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define MAX\_DATA\_LEN 256  #define SLEEP\_TIME 0  int main()  {  pid\_t pid1, pid2, pid3;  int pipe\_fd[2];  int len, realen, pipeid;  char buf[MAX\_DATA\_LEN], outpipe[MAX\_DATA\_LEN];  pipeid = pipe(pipe\_fd);  if (pipeid < 0)  {  printf("pipe error\n");  exit(1);  }  if ((pid1 = fork()) == -1)  {  printf("pid1 error\n");  exit(1);  }  if (pid1 == 0)  {  sleep(SLEEP\_TIME);  close(pipe\_fd[0]);  sprintf(outpipe, "Child1\n");  len = strlen(outpipe);  realen = write(pipe\_fd[1], outpipe, len);  exit(0);  }  if ((pid2 = fork()) == -1)  {  printf("pid2 error\n");  exit(1);  }  if (pid2 == 0)  {  sleep(SLEEP\_TIME);  close(pipe\_fd[0]);  sprintf(outpipe, "Child2\n");  len = strlen(outpipe);  realen = write(pipe\_fd[1], outpipe, len);  exit(0);  }  if ((pid3 = fork()) == -1)  {  printf("pid3 error\n");  exit(1);  }  if (pid3 == 0)  {  sleep(SLEEP\_TIME);  close(pipe\_fd[0]);  sprintf(outpipe, "Child3\n");  len = strlen(outpipe);  realen = write(pipe\_fd[1], outpipe, len);  exit(0);  }  else  {  pid1 = waitpid(pid1, NULL, 0);  pid2 = waitpid(pid2, NULL, 0);  pid3 = waitpid(pid3, NULL, 0);  read(pipe\_fd[0], buf, sizeof(buf) - 3);  printf("%s", buf);  }  return 0;  }  //父进程先于子进程结束，有名管道实现  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <fcntl.h>  #include <sys/types.h>  #include <sys/stat.h>  #define FIFO\_NAME "/tmp/myfifo"  #define BUF\_SIZE 1024  #define MAX\_DATA\_LEN 256  #define SLEEP\_TIME 5  int main()  {  int wfd;  char buf[BUF\_SIZE] = "Child1";  umask(0);  if (mkfifo(FIFO\_NAME, S\_IFIFO | 0666))  {  printf("Can't create FIFO.\n");  exit(1);  }  while ((wfd = open(FIFO\_NAME, O\_WRONLY | O\_NONBLOCK)) == -1)  {  perror("fopen error!");  sleep(3);  }  pid\_t pid1, pid2, pid3;  int pipe\_fd[2];  int len, realen, pipeid;  char buff[MAX\_DATA\_LEN], outpipe[MAX\_DATA\_LEN];  pipeid = pipe(pipe\_fd);  pid1 = fork();  if (pid1 == 0)  {  sleep(3);  printf("Write: ");  write(wfd, buf, strlen(buf) + 1);  close(wfd);  exit(0);  }  else  {  int rfd;  char buf[BUF\_SIZE];  umask(0);  if ((rfd = open(FIFO\_NAME, O\_RDONLY | O\_NONBLOCK)) == -1)  {  perror("fopen error.\n");  exit(1);  }  read(rfd, buf, BUF\_SIZE);  printf("Read content:%s\n", buf);  sleep(2);  close(rfd);  exit(0);  }  }  四、实验结果（截图）  正常次序：    父进程先于子进程结束：    **实验四**  **一、**实验目的  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。  二、实验原理  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。  三、实验内容（源码、注释、基础内容、扩展点等）  基础内容：  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。  **拓展点：**  **自定义随机序列大小及进程种类，使用父子进程协作实现。**  **源码及注释：**  #include <iostream>  #include <unistd.h>  #include <sys/types.h>  #include <errno.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include<sys/wait.h>  #define MAXN 1023  using namespace std;  int AcessSeries[MAXN]= {1,2,3,4,1,2,5,1,2,3,4,5};  int reg[MAXN],frame[MAXN],exist[MAXN];;  int regSize,hitNum,ifout;  int acessSize, pagesSize;  void randomG(int acc, int pag)  {  for(int i=acc;i<MAXN;AcessSeries[i]=-1, i++);  cout<<"AcessSeries:"<<endl;  for(int i=12;i<acc;i++)  {  int randnum = rand()%pag + 1;  AcessSeries[i] = randnum;  }  for(int i=0;i<acc;cout<<AcessSeries[i]<<" ",i++);  cout<<endl;  }  void display()  {  for(int i=0;i<regSize && frame[i]!=-1;i++)  {  cout<<frame[i]<<"|";  }  }  int hit(int come)  {  for(int i=0;i<regSize && reg[i]!=-1;i++)  {  if(reg[i]==come)  return i;  }  return -1;  }  int getRealLen(int order)  {  int len=0;  if(order==1)for(int i=0;reg[i]!=-1;i++,len=i);  if(order==2)for(int i=0;frame[i]!=-1;i++,len=i);  if(order==3)for(int i=0;exist[i]!=-1;i++,len=i);  return len;  }  int getBig()  {  for(int i=0;i<MAXN;exist[i]=-1,i++);  for(int i=0;i<MAXN;i++)  {  int flag = 0;  for(int j=0;exist[j]!=-1;j++)  {  if(AcessSeries[i]==exist[j]){flag=1;break;}  }  if(flag==0){exist[getRealLen(3)]=AcessSeries[i];}  }  return getRealLen(3);  }  void universal(int style)  {  int hitflag;  for(int i=0;AcessSeries[i]!=-1;i++)  {  hitflag = hit(AcessSeries[i]);  if(hitflag>=0)  {  hitNum ++;  //FIFO  if(style == 1){}  //LRU  else if(style == 2)  {  int temp = AcessSeries[i];  for(int j=hitflag;j>0;reg[j]=reg[j-1],j--);  reg[0] = temp;  }  }  else  {  //adjust reg  int target=-1;  for(int j=0;j<regSize&&reg[j]!=-1;target=reg[j],j++);  for(int j=regSize-1;j>0;reg[j]=reg[j-1],j--);  reg[0]=AcessSeries[i];  //adjust frame  int len = getRealLen(2);  if(len<regSize){frame[len]=AcessSeries[i];}  else  {  for(int j=0;j<regSize;j++)  {  if(frame[j]==target)  {  frame[j]=AcessSeries[i];  break;  }  }  }  }  if(ifout==1)  {  cout<<AcessSeries[i]<<":";  display();  if(hitflag>=0)cout<<"->Hit :)";  cout<<endl;  }  }  if(ifout==1) cout<<hitNum<<"/"<<acessSize<<" were hit."<<endl;  }  int fifo()  {  hitNum = 0;  for(int i=0;i<MAXN;reg[i]=frame[i]=-1,i++);  if(ifout==1)cout<<"FIFO:"<<endl;  universal(1);  return hitNum;  }  int lru()  {  hitNum = 0;  for(int i=0;i<MAXN;reg[i]=frame[i]=-1,i++);  if(ifout==1)cout<<"LRU:"<<endl;  universal(2);  return hitNum;  }  void child()  {  pid\_t pid1, pid2;  cout<<endl<<"Input Size of Reg:";  pid1 = fork();  pid2 = fork();  if(pid1>0){fifo();}  if(pid2>0){lru();}  else  {  wait(0);  }  }  int main()  {  cout<<"Input Size of Acess\_Series:";  cin>>acessSize;  cout<<"Input Types of pages:";  cin>>pagesSize;  randomG(acessSize,pagesSize);  int hitThis = 0, ifbelady = 0, big = getBig();  cout<<"Hit times of FIFO(Size:1-"<<big<<"): ";  for(int i=1;i<=big;i++)  {  regSize=i;  int temp = fifo();  if(temp<hitThis)ifbelady=1;  cout<<temp<<"|";  hitThis = temp;  }  cout<<" of "<<acessSize;  if(ifbelady==1){cout<<endl<<"FIFO Belady!";}  hitThis = 0, ifbelady = 0;  cout<<endl<<"Hit times of LRU(Size:1-"<<big<<"): ";  for(int i=1;i<=big;i++)  {  regSize=i;  int temp = lru();  if(temp<hitThis)ifbelady=1;  cout<<temp<<"|";  hitThis = temp;  }  cout<<" of "<<acessSize;  if(ifbelady==1){cout<<endl<<"LRU Belady!";}  ifout = 1;  child();  return 0;  }   1. 实验结果（截图） | | | | | | |