**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** | **20164460** | | **姓名** | **吴岳江** | **实验报告批改时间** |  |
| **实验项目目录**   1. 实验一　进程状态：模拟进程状态转换及其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：编程实现进程的管道通信程序 4. 实验四　页面置换算法：编程实现FIFO和LRU算法 | | | | | | |
| **实验报告正文**  **实验一 进程状态：模拟进程状态转换及其PCB的变化**  一、实验目的  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。  二、实验原理  …………  三、实验内容（源码、注释、基础内容、扩展点等）   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. 鼓励在实验中加入新的观点或想法，并加以实现。 7. 认真进行预习，并完成预习报告。 8. 实验完成后，要认真总结，完成实验报告。   **源代码：**  #include<iostream>  using namespace std;  typedef struct pcb {  int id;  struct pcb \*next;  }pcb;  pcb \*ready, \*run, \*block;  pcb\* cre()//创建带头结点的ready队列  {  pcb \*head = new pcb;  head->next = NULL;  pcb \*q = head;  int n;  cout << "输入进程数:\n";  cin >> n;  for (int i = 0; i<n; i++)  {  pcb \*p = new pcb;  cin >> p->id;  p->next = NULL;  q->next = p;  q = p;  }  return head;  }  void ins(pcb \*head, pcb \*node)//插入节点  {  pcb \*p = head;  while (p->next)  {  p = p->next;  }  pcb \*n = new pcb;  n->id = node->id;  p->next = n;  n->next = NULL;  }  void del(pcb \*head)  {  pcb \*p = head->next;  if (p)  {  head->next = head->next->next;  delete p;  }  }  void dis()  {  pcb \*p;  p = ready->next;  cout << "ready:\t";  while (p)  {  cout << p->id << " ";  p = p->next;  }  cout << endl;  p = run->next;  cout << "run :\t";  while (p)  {  cout << p->id << " ";  p = p->next;  }  cout << endl;  p = block->next;  cout << "block:\t";  while (p)  {  cout << p->id << " ";  p = p->next;  }  cout << endl;  }  int main()  {  ready = cre();  run = new pcb;  run->next = NULL;  block = new pcb;  block->next = NULL;  dis();  int i;  cout << "chose:\t1:ready->run\t2:run->ready\t3:run->block\t4:block->ready\n";  while (cin >> i)  {  if (i == 1)  {  if (ready->next)  {  if (run->next)  {  ins(block, run->next);  del(run);  }  ins(run, ready->next);  del(ready);  }  else  {  if (block->next)  {  ins(block, run->next);  del(run);  ins(run, block->next);  del(block);  }  }  }  else if (i == 2)  {  if (run->next)  {  if (ready->next)  {  ins(ready, run->next);  del(run);  ins(run, ready->next);  del(ready);  }  }  }  else if (i == 3)  {  if (run->next)  {  if (ready->next)  {  ins(block, run->next);  ins(run, ready->next);  del(run);  del(ready);  }  else  {  ins(block, run->next);  ins(run, block->next);  del(block);  del(run);  }  }  }  else if (i == 4)  {  if (block->next)  {  ins(ready, block->next);  del(block);  }  }  else  break;  dis();  cout << "chose:\t1:ready->run\t2:run->ready\t3:run->block\t4:block->ready\n";  }  return 0;  }  四、实验结果（截图）  **实验二 进程同步和通信：生产者和消费者问题模拟**  一、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。  二、实验原理  …………  三、实验内容（源码、注释、基础内容、扩展点等）  **内容：**  （1）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；  （2）发现并修改程序中的原理性错误或不完善的地方；  （3）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；  （4）在程序中适当地加入注释；  （5）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；  （6）实验完成后，要认真总结，完成实验报告。  **代码：**  #include <iostream>  #include <vector>  #include <queue>  #include <string>  #include <unistd.h>  #define BUFFERSIZE 8 // 缓冲区大小  using namespace std;  vector<int> buffer(BUFFERSIZE); // 缓冲区  queue<int> pQueue; // 生产者等待队列  queue<int> cQueue; // 消费者等待队列  int data; // 0-99  int writeptr, readptr; // 写指针　读指针  int consumer, producer; // 消费者等待数量　生产者等待数量  // 初始化  void init()  {  for (int i = 0; i < BUFFERSIZE; ++i)  {  buffer[i] = 0;  }  while (!pQueue.empty())  {  pQueue.pop();  }  while (!cQueue.empty())  {  cQueue.pop();  }  data = 0;  writeptr = readptr = 0;  consumer = producer = 0;  }  // 查找空位  int findEmpty(vector<int>& v, int len)  {  for (int i = 0; i < len; ++i)  {  if (v[i] == 0)  return i;  }  return -1;  }  // 查找脏位  int findDirty(vector<int>& v, int len)  {  for (int i = 0; i < len; ++i)  {  if (v[i] > 0)  return i;  }  return -1;  }  // 缓冲区是否满  bool isFull(vector<int>& v, int len)  {  for (int i = 0; i < len; i++)  {  if (v[i] == 0)  return false;  }  return true;  }  // 缓冲区是否空  bool isEmpty(vector<int>& v, int len)  {  for (int i = 0; i < len; i++)  {  if (v[i] != 0)  return false;  }  return true;  }  // 遍历展示队列中的内容  void showQueue(queue<int> q)  {  //queue<int> temp\_q;  int len = q.size();  for (int i = 0; i < len; i++)  {  cout << q.front() << " ";  q.pop();  //temp\_q.push(x);  }  }  // 显示缓冲区内容  void show()  {  for (int i = 0; i < BUFFERSIZE; ++i)  {  if (buffer[i] > 0)  cout << " " << buffer[i] << " |";  else  cout << " |";  }  cout << "\nwriteptr = " << writeptr << "\treadptr = " << readptr << "\t";  if (producer)  {  cout << "\nPRODUCER wait : " << producer << "\n" << "pQueue: ";  showQueue(pQueue);  }  else  cout << "\nPRODUCER ready\t";  if (consumer)  cout << "\nCONSUMER wait : " << consumer;  else  cout << "\nCONSUMER ready";  cout << "\n";  }  // 生产  void produce()  {  if (findEmpty(buffer, BUFFERSIZE) == -1) // 缓冲区满,则阻塞  {  ++data;  producer++;  pQueue.push(data);  }  else if (isEmpty(buffer, BUFFERSIZE)) // 缓冲区空  {  if (cQueue.empty()) // 消费者等待队列为空,直接写入即可  {  data++;  buffer[writeptr] = data;  }  else // 有阻塞的消费者，先写入再读出  {  data++;  buffer[writeptr] = data;  show();  sleep(2);  buffer[readptr] = 0;  readptr = (readptr + 1) % BUFFERSIZE;  consumer--;  cQueue.pop();  }  writeptr = (writeptr + 1) % BUFFERSIZE;  }  else // 缓冲区不空也不满,直接写入即可  {  data++;  buffer[writeptr] = data;  writeptr = (writeptr + 1) % BUFFERSIZE;  }  }  // 消费  void consume()  {  if (findDirty(buffer, BUFFERSIZE) == -1) // 缓冲区空,则阻塞  {  consumer++;  cQueue.push(data);  }  else if (isFull(buffer, BUFFERSIZE)) // 缓冲区满  {  if (!pQueue.empty()) // 生产者等待队列不空，则先读出再写入  {  buffer[readptr] = 0;  readptr = (readptr + 1) % BUFFERSIZE;  show();  sleep(2);  buffer[writeptr] = pQueue.front();  writeptr = (writeptr + 1) % BUFFERSIZE;  pQueue.pop();  producer--;  }  else // 生产者队列为空  {  buffer[readptr] = 0;  readptr = (readptr + 1) % BUFFERSIZE;  }  }  else // 缓冲区不空也不满,直接读取即可  {  buffer[readptr] = 0;  readptr = (readptr + 1) % BUFFERSIZE;  }  }  // 启动程序  void start()  {  char c;  cin >> c;  // e = exit p = produce c = consume  while (c != 'e' && (c == 'p' || c == 'c'))  {  if (c == 'p')  {  produce();  }  else if (c == 'c')  {  consume();  }  show();  cin >> c;  }  }  int main()  {  cout << "e-exit p-produce c-consume" << endl;  init();  start();  return 0;  }  四、实验结果（截图）  **实验三 进程的管道通信：编程实现进程的管道通信程序**  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系统进程间通过管道通信的具体实现方法。  二、实验原理  …………  三、实验内容（源码、注释、基础内容、扩展点等）  **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。  **代码：**  #include <stdio.h>  #include <sys/types.h>  #include <unistd.h>  #include <stdlib.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <wait.h>  #include <error.h>  # define F\_ULOCK 0 //解锁  # define F\_LOCK 1 //互斥锁定区域  int main()  {  int num, fd[2], result = -1;  pid\_t pid, reid, gtid;  char outpipe[100], father\_str[100];  //创建管道, fd[0]读管道，fd[1]写管道  result = pipe(fd);  if (result == -1)  {  printf("fail to create pipe \n");  return -1;  }  // 创建子进程  printf("input the number of process: ");  scanf("%d", &num);  printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  while (num > 0)  {  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  sprintf(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");  num--;  }  return 0;  }  四、实验结果（截图）  **实验四 页面置换算法：编程实现FIFO和LRU算法**  一、实验目的  （1）进一步理解父子进程之间的关系  （2）理解内存页面调度的机理  （3）掌握页面置换算法的实现方法  （4）通过实验比较不同调度算法的优劣  （5）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。  二、实验原理  …………  三、实验内容（源码、注释、基础内容、扩展点等）  **内容：**   * 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：   * 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。   + 缺页率 = disaffect/total\_instruction   + 命中率 = 1- disaffect/total\_instruction   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。  **代码：**  #include <stdio.h>  #include <sys/types.h>  #include <stdlib.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <error.h>  #include <wait.h>  #include <time.h>  #include <unistd.h>  #define true 1  #define false 0  #define frame\_num 4  #define frame\_num\_FIFO 6  #define total\_instruction 20  typedef enum { USED, UNUSED }frame\_flag;/\* flag of frame \*/  typedef struct oneframe {  int page\_no;  frame\_flag flag;  }one\_frame;  int main()  {  /\*\*\*\*\*\*\*\*\*\*\*initialize variable\*\*\*\*\*\*\*\*\*\*\*/  int i, j, pid, gtid, reid;  int access\_index = 0; // the index of Access\_Series  int success\_flag = false; // value is true if page exists  float diseffect = 0.0; // pages absence counts  float absence\_rate = 0.0; // absence\_rate = diseffect / total\_instruction  float success\_rate = 0.0; // success\_rate = 1 - absence\_rate  int Access\_Series[total\_instruction]; // random number sequence  one\_frame M\_Frame[frame\_num]; // memory access sequence  one\_frame M\_Frame\_FIFO[frame\_num\_FIFO]; // memory access sequence FIFO  int Belady[total\_instruction] = { 2, 7, 2, 3, 6, 3, 10, 7, 4, 5, 2, 6, 5, 10, 7, 6, 8, 5, 8, 10 }; // Belady  int Equal[total\_instruction] = { 5, 1, 10, 2, 5, 1, 2, 2, 1, 10, 10, 6, 7, 2, 8, 5, 4, 8, 8, 8 };  /\*\*\*\*\*\*\*\*\*\*\*initialize memory access series\*\*\*\*\*\*\*\*\*\*\*/  printf("Do you want to see Belady? (1-yes 2-no)\n");  do {  printf("Your choice: ");  scanf("%d", &i);  } while (i != 1 && i != 2);  if (i == 1)  {  for (int k = 0; k < total\_instruction; k++)  {  Access\_Series[k] = Belady[k];  printf("%d ", Access\_Series[k]);  }  }  else if (i == 2)  {  srand((int)time(0));  printf("\n\* access series numbers:");  for (int k = 0; k < total\_instruction; k++)  {  Access\_Series[k] = 1 + (int)(10.0 \* rand() / (RAND\_MAX + 1.0)); //[1,10]  printf("%d ", Access\_Series[k]);  }  }  printf("\n\n");  /\*\*\*\*\*\*\*\*\*\*\*initialize data structure M\_Frame\*\*\*\*\*\*\*\*\*\*\*/  for (i = 0; i < frame\_num; i++)  {  M\_Frame[i].page\_no = 0;  M\_Frame[i].flag = UNUSED;  }  for (i = 0; i < frame\_num\_FIFO; i++)  {  M\_Frame\_FIFO[i].page\_no = 0;  M\_Frame\_FIFO[i].flag = UNUSED;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*subprocess\_FIFO\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  pid = fork();  if (pid == -1) // create process error  {  printf("\* creat subprocess\_FIFO failed!\n");  exit(0);  }  else if (pid == 0) // subprocess  {  printf("\* subprocess\_FIFO %d\n", gtid = getpid());  while (access\_index < total\_instruction)  {  for (i = 0; i < frame\_num; i++) // check if page exists in memory  {  if (M\_Frame[i].flag == USED)  {  if (M\_Frame[i].page\_no == Access\_Series[access\_index])  {  success\_flag = true; // page exists in memory  break;  }  }  else // memory isn't full  break;  }  if (!success\_flag) // page absent/page not exists in memory  {  diseffect += 1.0;  for (i = 0; i < frame\_num; i++) // check if frames are full  {  if (M\_Frame[i].flag == UNUSED) // not full  {  M\_Frame[i].flag = USED;  M\_Frame[i].page\_no = Access\_Series[access\_index];  break;  }  }  if (i == frame\_num) // full  {  for (i = 0; i < frame\_num - 1; i++)  M\_Frame[i].page\_no = M\_Frame[i + 1].page\_no; // 栈中元素依次前移  M\_Frame[frame\_num - 1].page\_no = Access\_Series[access\_index];// 新元素在栈顶  }  }  // for(i = frame\_num - 1; i >= 0; i--) // print the state of frames  // {  // if(M\_Frame[i].flag == UNUSED)  // printf("\* M\_Frame[%d]: \n",i);  // else  // printf("\* M\_Frame[%d]: %d\n", i, M\_Frame[i].page\_no);  // }  // printf("--------------\n");  success\_flag = false;  access\_index++;  }  absence\_rate = diseffect / total\_instruction; // 缺页率计算  success\_rate = 1 - absence\_rate;  printf("\* subprocess\_FIFO %d successful [%d]:absence\_rate=%f, success\_rate=%f\n\n",  gtid, frame\_num, absence\_rate, success\_rate);  access\_index = 0;  success\_flag = false;  diseffect = 0.0;  exit(0);  }  reid = wait(NULL);  if (reid == -1)  printf("\* call subprocess\_FIFO failed!\n");  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*subprocess\_FIFO\_Belady\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  pid = fork();  if (pid == -1) // create process error  {  printf("\* creat subprocess\_FIFO failed!\n");  exit(0);  }  else if (pid == 0) // subprocess  {  printf("\* subprocess\_FIFO %d\n", gtid = getpid());  while (access\_index < total\_instruction)  {  for (i = 0; i < frame\_num\_FIFO; i++) // check if page exists in memory or not  {  if (M\_Frame\_FIFO[i].flag == USED)  {  if (M\_Frame\_FIFO[i].page\_no == Access\_Series[access\_index])  {  success\_flag = true; // page exists in memory  break;  }  }  else  break;  }  if (!success\_flag) // page absent/page not exists in memory  {  diseffect += 1.0;  for (i = 0; i < frame\_num\_FIFO; i++) // check if frames are full or not  {  if (M\_Frame\_FIFO[i].flag == UNUSED) // not full  {  M\_Frame\_FIFO[i].flag = USED;  M\_Frame\_FIFO[i].page\_no = Access\_Series[access\_index];  break;  }  }  if (i == frame\_num\_FIFO) // full  {  for (i = 0; i < frame\_num\_FIFO - 1; i++)  M\_Frame\_FIFO[i].page\_no = M\_Frame\_FIFO[i + 1].page\_no;  M\_Frame\_FIFO[frame\_num\_FIFO - 1].page\_no = Access\_Series[access\_index];  }  }  // for(i = frame\_num\_FIFO - 1; i >= 0; i--) // print the state of frames  // {  // if(M\_Frame\_FIFO[i].flag == UNUSED)  // printf("\* M\_Frame[%d]: \n",i);  // else  // printf("\* M\_Frame[%d]: %d\n", i, M\_Frame\_FIFO[i].page\_no);  // }  // printf("--------------\n");  success\_flag = false;  access\_index++;  }  absence\_rate = diseffect / total\_instruction;  success\_rate = 1 - absence\_rate;  printf("\* subprocess\_FIFO %d successful [%d]: absence\_rate=%f, success\_rate=%f\n\n",  gtid, frame\_num\_FIFO, absence\_rate, success\_rate);  access\_index = 0;  success\_flag = false;  diseffect = 0.0;  exit(0);  }  reid = wait(NULL);  if (reid == -1)  printf("\* call subprocess\_FIFO failed!\n");  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*subprocess\_LRU\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  pid = fork();  if (pid == -1) // create process error  {  printf("\* creat subprocess\_LRU failed!\n");  exit(0);  }  else if (pid == 0) // subprocess  {  printf("\* subprocess\_LRU %d\n", gtid = getpid());  while (access\_index < total\_instruction)  {  for (i = 0; i < frame\_num; i++) // check if page exists in memory  {  if (M\_Frame[i].flag == USED)  {  if (M\_Frame[i].page\_no == Access\_Series[access\_index])  {  success\_flag = true; // page exists in memory  for (j = i; j < frame\_num - 1; j++) // 将命中的页重新放到栈顶  {  if (M\_Frame[j + 1].flag == UNUSED)  break;  else  {  M\_Frame[j].page\_no = M\_Frame[j + 1].page\_no;  M\_Frame[j + 1].page\_no = Access\_Series[access\_index];  }  }  break;  }  }  else  break;  }  if (!success\_flag) // 未命中  {  diseffect += 1.0;  for (i = 0; i < frame\_num; i++) // check if frames are full  {  if (M\_Frame[i].flag == UNUSED) // not full  {  M\_Frame[i].flag = USED;  M\_Frame[i].page\_no = Access\_Series[access\_index];  break;  }  }  if (i == frame\_num) // full  {  for (i = 0; i < frame\_num - 1; i++) // 栈底元素出栈，栈顶压入新元素  M\_Frame[i].page\_no = M\_Frame[i + 1].page\_no;  M\_Frame[frame\_num - 1].page\_no = Access\_Series[access\_index];  }  }  for (i = frame\_num - 1; i >= 0; i--) // print the state of frames  {  if (M\_Frame[i].flag == UNUSED)  printf("\* M\_Frame[%d]: \n", i);  else  printf("\* M\_Frame[%d]: %d\n", i, M\_Frame[i].page\_no);  }  printf("--------------\n");  success\_flag = false;  access\_index++;  }  absence\_rate = diseffect / total\_instruction;  success\_rate = 1 - absence\_rate;  printf("\* subprocess\_LRU %d successful [%d]: absence\_rate=%f, success\_rate=%f\n",  gtid, frame\_num, absence\_rate, success\_rate);  exit(0);  }  reid = wait(NULL);  if (reid == -1)  printf("\* call subprocess\_LRU failed!\n");  return 0;  }  四、实验结果（截图） | | | | | | |