**重 庆 大 学**

**学 生 实 验 报 告**

**实验课程名称 操作系统原理**

**开课实验室 DS1501**

**学 院 软件学院 年级2020 专业班 软件工程1 班**

**学 生 姓 名 李易燔 学 号 20205644**

**开 课 时 间 2021 至 2022 学年第 二 学期**

|  |  |
| --- | --- |
| **总 成 绩** |  |
| **教师签名** | **刘寄** |

**软件学院制**

**《操作系统原理》实验报告**

**开课实验室： 2022年 4月 16日**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 学院 | | 软件学院 | 年级、专业、班 | | 2020级软件工程1班 | 姓名 | | 李易燔 | | 成绩 |  |
| 课程  名称 | | 操作系统原理 | | 实验项目  名 称 | 线程的同步 | | 指导教师 | | 刘寄 | | |
| 教师评语 | 教师签名：刘寄  年 月 日 | | | | | | | | | | |
| **1．实验目的：**   * 掌握线程的同步   **2．实验内容：**  根据实验步骤编写函数实现功能。  **3．实验步骤：**   * Step1：实现信号量的四个函数 * Step2：增加信号量系统调用 * Step3：创建生产者/消费者线程 * Step4：创建控制线程   **4．实验报告：**  信号量结构的实现以及基本的链表功能函数 //实验4 struct semaphore//信号量结构体 {     int sem\_Id;//信号量ID     int sem\_val;//信号量初值     struct wait\_queue\* list;//等待队列     struct semaphore\* next;//下一个信号量 }; extern struct semaphore\* g\_sem\_head = NULL;//信号量链表的表头 extern struct semaphore\* g\_sem\_select;//当前信号量 extern int g\_sem\_id;//信号量ID int sys\_sem\_create(int value); int sys\_sem\_destroy(int semid); int sys\_sem\_wait(int semid); int sys\_sem\_signal(int semid);  汇编接口 WRAPPER(sem\_create) WRAPPER(sem\_destroy) WRAPPER(sem\_wait) WRAPPER(sem\_signal)  #define SYSCALL\_sem\_create 13 #define SYSCALL\_sem\_destroy 14 #define SYSCALL\_sem\_wait 15 #define SYSCALL\_sem\_signal 16  /\*\*  \* vim: filetype=c:fenc=utf-8:ts=4:et:sw=4:sts=4  \*/ #include <stddef.h> #include "kernel.h"   struct semaphore\* g\_sem\_head=NULL;//信号量链表的表头 struct semaphore\* g\_sem\_select=NULL;//当前信号量 int g\_sem\_id=0;//信号量ID  //添加新的信号量 void sem\_append(struct semaphore\* sem) {     if (g\_sem\_head == NULL) {         g\_sem\_head = sem;         sem->next = NULL;         return;     }     else {         struct semaphore\* select = g\_sem\_head;         while (select->next != NULL) {             select = select->next;         }         select -> next = sem;         sem->next = NULL;         return;     } } //寻找信号量 struct semaphore\* get\_sem(int semid) {     struct semaphore\* select = g\_sem\_head;     while (select != NULL) {         if (select->sem\_Id == semid) {             return select;         }         else {             select = select->next;         }     }     return NULL; }  int sys\_sem\_create(int value) {     struct semaphore\* sem;     sem = (struct semaphore\*)kmalloc(sizeof(struct semaphore));     if (sem == NULL)return -1;     else {         sem->sem\_Id = g\_sem\_id++;         sem->sem\_val = value;         sem->list = NULL;         sem\_append(sem);         return sem->sem\_Id;     } }  int sys\_sem\_destroy(int semid) {     if (g\_sem\_head == NULL) {         return -1;     }     else {         struct semaphore\* select = get\_sem(semid);         struct semaphore\* prev = get\_sem(semid - 1);         if (select == NULL) {             return -1;         }         else if(prev==NULL) {             g\_sem\_head = select->next;             kfree(select);             return 0;         }         else {             prev->next = select->next;             kfree(select);             return 0;         }     } }  int sys\_sem\_wait(int semid)//P操作 {     struct semaphore\* select = get\_sem(semid);     if (select == NULL) {         return -1;     }     else {         select->sem\_val--;         if (select->sem\_val < 0) {             uint32\_t flags;             save\_flags\_cli(flags);             sleep\_on(&(select->list));//加入等待队列             restore\_flags(flags);         }         return 0;     } }  int sys\_sem\_signal(int semid)//V操作 {     struct semaphore\* select = get\_sem(semid);     if (select == NULL) {         return -1;     }     else {         select->sem\_val++;         if (select->sem\_val <= 0) {             uint32\_t flags;             save\_flags\_cli(flags);             wake\_up(&(select->list), 1);//唤醒一个线程             restore\_flags(flags);         }         return 0;     } }       //实验4     case SYSCALL\_sem\_create: {         int val = \*(int\*)(ctx->esp + 4);         ctx->eax = sys\_sem\_create(val);          break;     }     case SYSCALL\_sem\_destroy: {         int semid= \*(int\*)(ctx->esp + 4);         ctx->eax =sys\_sem\_destroy(semid);          break;     }     case SYSCALL\_sem\_wait: {         int semid = \*(int\*)(ctx->esp + 4);         ctx->eax = sys\_sem\_wait(semid);         break;     }     case SYSCALL\_sem\_signal: {         int semid = \*(int\*)(ctx->esp + 4);         ctx->eax = sys\_sem\_signal(semid);         break;     }  Main函数  /\*  \* vim: filetype=c:fenc=utf-8:ts=4:et:sw=4:sts=4  \*/  #include <inttypes.h>  #include <stddef.h>  #include <math.h>  #include <stdio.h>  #include <sys/mman.h>  #include <syscall.h>  #include <netinet/in.h>  #include <stdlib.h>  #include "graphics.h"  #include<time.h>  extern void\* tlsf\_create\_with\_pool(void\* mem, size\_t bytes);  extern void\* g\_heap;  /\*\*  \* GCC insists on \_\_main  \* http://gcc.gnu.org/onlinedocs/gccint/Collect2.html  \*/  void \_\_main()  {  size\_t heap\_size = 32 \* 1024 \* 1024;  void\* heap\_base = mmap(NULL, heap\_size, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANON, -1, 0);  g\_heap = tlsf\_create\_with\_pool(heap\_base, heap\_size);  }  //线程优先级  int tid\_foo1, tid\_foo2, tid\_foo3;  int tskbuff1 = 0;  int tskbuff2 = 0;  int fullbuffer;//满槽  int emptybuffer;//空槽  int mutex[6];  int max\_buf = 6;//分成6片区域  int countLen = 200;  int sort\_buf[6][200];//缓冲区  //睡眠函数  void mySleep()  {  struct timespec tim, tim2;  tim.tv\_sec = 0;  tim.tv\_nsec = 1000000;  nanosleep(&tim, &tim2);  }  void myLongSleep()  {  struct timespec tim, tim2;  tim.tv\_sec = 1;  tim.tv\_nsec = 0;  nanosleep(&tim, &tim2);  }  void tsk\_foo1(void\* pv) //生产者  {  srand(time(NULL));  int i;  while (1)  {  sem\_wait(emptybuffer);  sem\_wait(mutex[tskbuff1 % 6]);//上锁  for (i = 0; i < countLen; i++)  {  sort\_buf[tskbuff1 % 6][i] = rand() % 100;  line(tskbuff1 % 6 \* 100, i \* 2 + 20, tskbuff1 % 6 \* 100 + sort\_buf[tskbuff1 % 6][i], i \* 2 + 20, RGB(0, 255, 255));  }  // mySleep();  myLongSleep();  sem\_signal(mutex[tskbuff1 % 6]);//开锁  sem\_signal(fullbuffer);//将满槽加一  tskbuff1++;  }  task\_exit(0);  }  void tsk\_foo2(void\* pv) //消费者  {  int i, j, a[200], temp;  while (1)  {  sem\_wait(fullbuffer);  sem\_wait(mutex[tskbuff2 % 6]);//上锁  for (i = 0; i < countLen; i++)  {  a[i] = sort\_buf[tskbuff2 % 6][i];  }//拷贝缓存区数据  for (i = 0; i < countLen; i++)  {  for (j = 0; j < countLen - i - 1; j++)  {  if (a[j] > a[j + 1])  {  //覆盖排序前的两条线段  line((tskbuff2 % 6) \* 100, j \* 2 + 20, (tskbuff2 % 6) \* 100 + a[j], j \* 2 + 20, RGB(0, 0, 0));  line((tskbuff2 % 6) \* 100, (j + 1) \* 2 + 20, (tskbuff2 % 6) \* 100 + a[j + 1], (j + 1) \* 2 + 20, RGB(0, 0, 0));  temp = a[j];  a[j] = a[j + 1];  a[j + 1] = temp;  line((tskbuff2 % 6) \* 100, j \* 2 + 20, (tskbuff2 % 6) \* 100 + a[j], j \* 2 + 20, 0x836FFF);  line((tskbuff2 % 6) \* 100, (j + 1) \* 2 + 20, (tskbuff2 % 6) \* 100 + a[j + 1], (j + 1) \* 2 + 20, 0x836FFF);  }  }  mySleep();  }  mySleep();  //myLongSleep();  //清除线段  for (j = 0; j < countLen; j++)  line((tskbuff2 % 6) \* 100, j \* 2 + 20, (tskbuff2 % 6) \* 100 + a[j], j \* 2 + 20, RGB(0, 0, 0));  sem\_signal(mutex[tskbuff2 % 6]);  sem\_signal(emptybuffer);  tskbuff2++;  }  task\_exit(0);  }  //优先级展示条  void show\_priority(int tid, int k) {  int length = 20 \* getpriority(tid);  int m = 0;  switch (k)  {  case 1:  {  for (m = 0; m < 20; m++)  line(0, 450 + m, 800, 450 + m, RGB(0, 0, 0));  for (m = 0; m < 20; m++)  line(0, 450 + m, length, 450 + m, 0x7D26CD);  }break;  case 2:  {  for (m = 0; m < 20; m++)  line(0, 500 + m, 800, 500 + m, RGB(0, 0, 0));  for (m = 0; m < 20; m++)  line(0, 500 + m, length, 500 + m, 0x6B8E23);  }break;  default:  break;  }  }  //控制线程  void mytask\_control(void\* pv) {  show\_priority(tid\_foo1, 1);  show\_priority(tid\_foo2, 2);  int mykeypress;  while (1) {  mykeypress = getchar();  switch (mykeypress)  {  case 0x4800://(up)  {  setpriority(tid\_foo1, getpriority(tid\_foo1) + 1);  show\_priority(tid\_foo1, 1);  }  break;  case 0x5000://(down)  {  setpriority(tid\_foo1, getpriority(tid\_foo1) - 1);  show\_priority(tid\_foo1, 1);  }  break;  //0x4d00(right)/0x4b00(left)  case 0x4d00:  {  setpriority(tid\_foo2, getpriority(tid\_foo2) + 1);  show\_priority(tid\_foo2, 2);  }  break;  case 0x4b00:  {  setpriority(tid\_foo2, getpriority(tid\_foo2) - 1);  show\_priority(tid\_foo2, 2);  }  break;  default:  break;  }  }  }  void main(void\* pv)  {  fullbuffer = sem\_create(0);  emptybuffer = sem\_create(max\_buf);  unsigned char\* stack\_foo1, \* stack\_foo2, \* stack\_foo3;  unsigned int stack\_size = 1024 \* 1024;  stack\_foo1 = (unsigned char\*)malloc(stack\_size);  stack\_foo2 = (unsigned char\*)malloc(stack\_size);  stack\_foo3 = (unsigned char\*)malloc(stack\_size);  //图形模式  int i;  init\_graphic(0x143);  for (i = 0; i < 6; i++)  {  line(1 + i \* 100, 0, 1 + i \* 100, 420, RGB(233, 233, 233));  mutex[i] = sem\_create(1);  }  tid\_foo1 = task\_create(stack\_foo1 + stack\_size, &tsk\_foo1, (void\*)0);  tid\_foo2 = task\_create(stack\_foo2 + stack\_size, &tsk\_foo2, (void\*)0);  tid\_foo3 = task\_create(stack\_foo3 + stack\_size, &mytask\_control, (void\*)0);  setpriority(tid\_foo3, -20);  task\_wait(tid\_foo1, NULL);  task\_wait(tid\_foo2, NULL);  task\_wait(tid\_foo3, NULL);  free(stack\_foo1);  free(stack\_foo2);  free(stack\_foo3);  sem\_destroy(fullbuffer);  for (i = 0; i < 6; i++) {  sem\_destroy(mutex[i]);  }  while (1);  task\_exit(0);  }  效果图： | | | | | | | | | | | |