ICS第十二章

1. volatile保证定义的变量存放在内存中，而不总是在寄存器里。右侧为两个进程的地址空间。请在合适的位置标出变量gCount、vCount与lCount的位置。如果一个量出现多次，那么就标多次。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| long **gCount** = 0;  void \*thread(void \*vargp) {  volatile long **vCount** = \*(long \*)vargp;  static long **lCount** = 0;  gCount++; vCount++; lCount++;  printf("%ld\n", gCount+vCount+lCount);  return NULL;  }  int main() {  long var; pthread\_t tid1, tid2;  scanf("%ld", &var);  fork();  pthread\_create(&tid1,NULL,thread,&var);  pthread\_create(&tid2,NULL,thread,&var);  pthread\_join(tid1, NULL);  pthread\_join(tid2, NULL);  return 0;  } |  | vCount  vCount | 高地址 | vCount  vCount |
| ... |  | ... |
| 共享库 | 共享库 | 共享库 |
| ... |  | ... |
|  | 堆 |  |
| gCount  lCount | 数据区 | gCount  lCount |
|  | 代码区  低地址 |  |
| 父进程 |  | 子进程 |

2. 下面的程序会引发竞争。一个可能的输出结果为2 1 2 2。解释输出这一结果的原因。

|  |
| --- |
| long foo = 0, bar = 0;  void \*thread(void \*vargp) {  foo++; bar++;  printf("%ld %ld ", foo, bar); fflush(stdout);  return NULL;  }  int main() {  pthread\_t tid1, tid2;  pthread\_create(&tid1, NULL, thread, NULL);  pthread\_create(&tid2, NULL, thread, NULL);  pthread\_join(tid1, NULL);  pthread\_join(tid2, NULL);  return 0;  } |

【答】线程1将foo、bar改为1以后被线程2打断，线程2将foo改为2以后被线程1打断，线程1输出了2 1，线程2将bar改为2，并输出了2 2。

3. 信号量w,x,y,z均被初始化为1。下面的两个线程运行时可能会发生死锁。给出发生死锁的执行顺序。

|  |  |
| --- | --- |
| 线程1 | ①P(w) ②P(x) ③P(y) ④P(z) ⑤V(w) ⑥V(x) ⑦V(y) ⑧V(z) |
| 线程2 | ⅠP(x) ⅡP(z) ⅢP(y) ⅣP(w) ⅤV(x) ⅥV(y) ⅦV(w) ⅧV(z) |

【答】①→Ⅰ→Ⅱ→Ⅲ，此时线程1占用了w而在等待x，线程2占用了x而在等待w。

4. 某次考试有30名学生与1名监考老师，该教室的门很狭窄，每次只能通过一人。考试开始前，老师和学生进入考场（有的学生来得比老师早），当人来齐以后，老师开始发放试卷。拿到试卷后，学生就可以开始答卷。学生可以随时交卷，交卷后就可以离开考场。当所有的学生都上交试卷以后，老师才能离开考场。

请用信号量与PV操作，解决这个过程中的同步问题。所有空缺语句均为PV操作。

|  |
| --- |
| 全局变量：  stu\_count: int类型，表示考场中的学生数量，初值为0  信号量：  mutex\_stu\_count: 保护全局变量，初值为1  mutex\_door: 保证门每次通过一人，初值为**1**  mutex\_all\_present: 保证学生都到了，初值为**0**  mutex\_all\_handin: 保证学生都交了，初值为**0**  mutex\_test[30]: 表示学生拿到了试卷，初值均为**0** |

|  |  |
| --- | --- |
| Teacher: // 老师  **P(mutex\_door)**  从门进入考场  **V(mutex\_door)**  **P(mutex\_all\_present)** // 等待同学来齐  for (i = 1; i <= 30; i++)  **V(mutex\_test[i])** // 给i号学生发放试卷    **P(mutex\_all\_handin)** // 等待同学将试卷交齐  **P(mutex\_door)**  从门离开考场  **V(mutex\_door)** | Student(x): // x号学生  **P(mutex\_door)**  从门进入考场  **V(mutex\_door)**  P(mutex\_stu\_count);  stu\_count++;  if (stu\_count == 30)  **V(mutex\_all\_present)**  V(mutex\_stu\_count);  **P(mutex\_test[i])** // 等待拿自己的卷子  学生答卷  P(mutex\_stu\_count);  stu\_count--;  if (stu\_count == 0)  **V(mutex\_all\_handin)**  V(mutex\_stu\_count);  **P(mutex\_door)**  从门离开考场  **V(mutex\_door)** |

5. 竞争

以下几段代码创建两个对等线程，并希望第一个线程输出0，第二个输出1；但有些代码会因为变量myid的竞争问题导致错误，请你判断哪些代码会在myid上存在竞争。如果不存在竞争，请你判断这段代码是否一定先输出0再输出1？

1. 不会，因为两个线程myid对应heap中不同位置的变量

|  |  |
| --- | --- |
| void \*foo(void \*vargp) {  int myid;  myid = \*(int \*)vargp;  free(vargp);  printf("Thread %d\n", myid);  } | int main() {  pthread\_t tid[2];  int i, \*ptr;  for (i = 0; i < 2; ++i) {  ptr = malloc(sizeof(int));  \*ptr = i;  pthread\_create(&tid[i], 0, foo, ptr);  }  pthread\_join(tid[0], 0);  pthread\_join(tid[1], 0);  } |

1. 存在竞争，两个myid都是对main函数堆栈中i的引用

|  |  |
| --- | --- |
| void \*foo(void \*vargp) {  int myid;  myid = \*(int \*)vargp;  printf("Thread %d\n", myid);  } | int main() {  pthread\_t tid[2];  int i;  for (i = 0; i < 2; ++i) {  pthread\_create(&tid[i], 0, foo, &i);  }  pthread\_join(tid[0], 0);  pthread\_join(tid[1], 0);  } |

1. 不存在竞争，因为创建线程传递的是值而非指针

|  |  |
| --- | --- |
| void \*foo(void \*vargp) {  int myid;  myid = (int)vargp;  printf("Thread %d\n", myid);  } | int main() {  pthread\_t tid[2];  int i;  for (i = 0; i < 2; ++i) {  pthread\_create(&tid[i], 0, foo, i);  }  pthread\_join(tid[0], 0);  pthread\_join(tid[1], 0);  } |

1. 存在竞争，两个对等线程和主线程都会访问i，即使在对等线程中加了互斥锁进行保护，但是主线程仍然可以修改vargp对应的内存（也就是i）的值

|  |  |
| --- | --- |
| sem\_t s;  void \*foo(void \*vargp) {  int myid;  P(&s);  myid = \*(int \*)vargp;  V(&s);  printf("Thread %d\n", myid);  } | int main() {  pthread\_t tid[2];  int i;  sem\_init(&s, 0, 1);  for (i = 0; i < 2; ++i) {  pthread\_create(&tid[i], 0, foo, &i);  }  pthread\_join(tid[0], 0);  pthread\_join(tid[1], 0);  } |

1. 不存在竞争，因为这段代码实现了同步，三个线程对i的访问实现了完全互斥，同时这段代码也保证先输出0再输出1

|  |  |
| --- | --- |
| sem\_t s;  void \*foo(void \*vargp) {  int myid;  myid = \*(int \*)vargp;  V(&s);  printf("Thread %d\n", myid);  } | int main() {  pthread\_t tid[2];  int i;  sem\_init(&s, 0, 0);  for (i = 0; i < 2; ++i) {  pthread\_create(&tid[i], 0, foo, &i);  P(&s);  }  pthread\_join(tid[0], 0);  pthread\_join(tid[1], 0);  } |

6. 读者写者问题

一组并发的线程想要访问一个共享对象，有无数的读者和写者想要访问共享对象，读者可以和其它读者同时访问，而写者必须独占对象。以下是第一类读者写者问题的代码。

|  |  |
| --- | --- |
| void reader() {  P(&mutex);  readcnt++;  if (readcnt == 1)  P(&w); /\* line a \*/  V(&mutex);  /\* reading... line b \*/  P(&mutex);  readcnt--;  if (readcnt == 0)  V(&w);  V(&mutex);  } | void writer() {  P(&w); /\* line c \*/  /\* writing... line d \*/  V(&w);  } |

(1) 假设在时刻0~4分别有五个读、写者到来；它们的顺序为R1, R2, W1, R3, W2；已知读操作需要等待3个周期，写操作需要等待5个周期；假设忽略其他语句的执行时间、线程的切换/调度的时间开销，因此在任意时刻，每个读者、写者只能处在上面标注好的abcd四处语句，请你分析这五个读者/写者线程终止的顺序？

R1R2R3W1W2 or R1R2R3W2W1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 时刻 | R1 | R2 | W1 | R3 | W2 |
| 0 | b |  |  |  |  |
| 1 | b | b |  |  |  |
| 2 | b | b | c |  |  |
| 3 | - | b | c | b |  |
| 4 | - | - | c | b | c |
| 5 | - | - | c | b | c |
| 6 | - | - | c/d | - | d/c |

根据上表分析可知两个写者将等待最后一个读者退出执行V(&w)语句才得以继续执行；由于V操作会随机唤醒一个睡眠在P操作的线程，所以W1和W2的执行顺序不可知。

(2) 基于(1)的发现，这段代码容易导致饥饿，于是一位同学规定：当有写者在等待时，后来的读者不能进行读操作，写出了第二类读者写者问题的代码如下（所有信号量初始化为1）：

|  |  |
| --- | --- |
| void reader() {  P(&r); /\* a \*/  P(&mutex);  readcnt++;  if (readcnt == 1)  P(&w); /\* b \*/  V(&mutex);  V(&r);  /\* reading... c \*/  P(&mutex);  readcnt--;  if (readcnt == 0)  V(&w);  V(&mutex);  } | void writer() {  P(&mutex);  writecnt++;  if (writecnt == 1)  P(&r); /\* d \*/  V(&mutex);  P(&w); /\* e \*/  /\* writing... f \*/  V(&w);  P(&mutex);  writecnt--;  if (writecnt == 0)  V(&r);  V(&mutex);  } |

这段代码会导致死锁，请你列举一种可能导致死锁的线程控制流，并提出一种改进的方案。

初始时刻某个读者和写者同时到来，读者执行P(&r)，写者执行P(&mutex)，然后两个线程都无法继续执行，后来的线程也会阻塞在第一个P操作。

解决方案：1.使用不同的信号量实现对readcnt和writecnt的互斥保护；2.将读者线程V(&r)移到P(&r)的后面

(3) 在修改了(2)中的问题后，请你基于第二类读者写者问题的代码再回答(1)中的题目。

R1R2W1W2R3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 时刻 | R1 | R2 | W1 | R3 | W2 |
| 0 | c |  |  |  |  |
| 1 | c | c |  |  |  |
| 2 | c | c | e |  |  |
| 3 | - | c | e | a |  |
| 4 | - | - | f | a | e |
| 5 | - | - | f | a | e |
| 6 | - | - | f | a | e |
| 7 | - | - | f | a | e |
| 8 | - | - | f | a | e |
| 9 | - | - | - | a | f |

7. 线程安全函数

吴用功同学找了一个找素数的函数next\_prime，ta在实现这个函数的线程安全版本ts\_next\_prime的时候出现了问题，请你帮助ta。

|  |
| --- |
| struct big\_number \*next\_prime(struct big\_number current\_prime) {  static struct big\_number next;  next = current\_prime;  addOne(next);  while(!isNotPrime(next))  addOne(next);  return &next;  }  struct big\_number \*ts\_next\_prime(struct big\_number current\_prime) {  return next\_prime(current\_prime);  } |

1. 现在的ts\_next\_prime为什么线程不安全？

返回了一个指向静态变量的指针

1. 下面的代码是否线程安全？

|  |
| --- |
| struct big\_number \*ts\_next\_prime(struct big\_number current\_prime) {  struct big\_number \*value\_ptr;  P(&mutex); /\* mutex is initialized to 1\*/  value\_ptr = next\_prime(current\_prime);  V(&mutex);  return value\_ptr;  } |

并不安全，在V语句后，其他线程也可以调用next\_prime，进而导致value\_ptr静指向的静态变量next被修改。

1. 请使用lock&copy技术实现线程安全的ts\_next\_prime

|  |
| --- |
| sem\_t mutex;  struct big\_number \*ts\_next\_prime(struct big\_number current\_prime) {  struct big\_number \*value\_ptr;  struct big\_number \*ret\_ptr = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;  P(&mutex); /\* mutex is initiallized to 1\*/  value\_ptr = next\_prime(current\_prime);  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;  V(&mutex);  return value\_ptr;  } |

(struct big\_number \*)malloc(sizeof(struct big\_number))

memcpy(ret\_ptr, value\_ptr, sizeof(struct big\_number))