DC126732

Huang Yanzhen

CISC2005 Lab6

|  |  |
| --- | --- |
| Question 1 | |
|  | |
| Question 1.1 | Question 1.2 |
| The two threads access global variable *counter* simultaneously.  Suppose that at a certain moment *counter == 300*. Thread *tid1* recorded the value 300 and added 1 such that it’ll be 301. However, before 301 is assigned to *counter*, a **context switch** happens, *counter* is still 300. Now, *tid2* recorded, and then assigned 301 to *counter*. After the context switch, thread *tid1* will assign 301 to *counter* again.  From the overall perspective, the threads recorded (accessed) *counter* twice, so *counter* should also be added twice, yet the result is equivalent to adding *counter* for only once, causing a false operation. This false operation has a very high chance to occur. | The chance of correctness is increased when the number of iteration is reduced. However, it’s still possible that a false result is given.  This is because there isn’t a guarantee that a context switch won’t happen between the recording & assigning operations within a loop for each thread. |

|  |  |
| --- | --- |
| Question 2 | |
|  | |
| Question 2.1 | |
| *// Thread 1*  void \*thread\_1(void \*arg)  {  while (TRUE)  {  while (turn != 1)  {  *// Wait*  }  *// Enter critical region*  for (long i = 0; i < N\_ITERATIONS; i++)  counter++;  *// Leave critical region*  turn = 2;  break;  }  pthread\_exit(NULL);  }  *// Thread 2*  void \*thread\_2(void \*arg)  {  while (TRUE)  {  while (turn != 2)  {  *// Wait*  }  *// Enter critical region*  for (long i = 0; i < N\_ITERATIONS; i++)  counter++;  turn = 1;  *// Leave critical region*  break;  }  return NULL;  } | |
| Question 2.2 | Question 2.3 |
| When *turn = 1, thread2* would wait until *thread 1* exits critical region and change *turn* into 2. Likewise, when *turn = 2, thread1* would wait until *thread2* exits critical region and change *turn* into 1. Therefore, when *thread1* is in critical region, *thread2* can’t enter it, and vice versa. | **Drawbacks.**  1. The while-loop is expensive, wasting a lot of CPU time .  2. The number of threads is not predictable, and the case would become more and more complicated when the number of threads gets large. |

|  |
| --- |
| Question 3 |
| Text  Description automatically generated |
| Question 3.1 |
| *// Increment function*  void \*increment(void \*arg)  {  int thread\_id = \*(int \*)arg;  flag[thread\_id] = ENTER;  turn = !thread\_id;  while (flag[!thread\_id] && turn == !thread\_id)  {  *// Wait*  }  *// Enter critical region*  for (long I = 0; I < N\_ITERATIONS; i++)  {  counter++;  }  *// Leave critical section*  flag[thread\_id] = LEAVE;  return NULL;  } |
| Question 3.2 |
| The condition where *thread1* is allowed into the critical region is ***P1: flag2==false or turn==1***. For *thread2*, it’s ***P2: flag1==false or turn==2***.  As a thread is created, it changes the flag of itself to *true,* and changes the turn to its opponent. As a thread exists critical region, it changes its own flag to false.  The condition suggest that a thread could only enter the critical region before its opponent enters it (turn) , or after its opponent leaves it (flag). |

|  |  |
| --- | --- |
| Question 4 | |
| 1st Algorithm | 2nd Algorithm |
| Text  Description automatically generated | (Sorry, but the CPU of my computer is in ARM structure so I can’t run the *test\_and\_set* function. ) |
| Question 4.1 | |
| The first algorithm simply implement the lock using the software solution, while the second algorithm required a hardware solution. | |
| Question 4.2 | |
| The first algorithm doesn’t work. Because a context switch may occur between the *while-loop* and *lock=true* in the *acquire()* function. There’s a racing condition for the lock. | |
| Question 4.3 | |
| The second algorithm works. Because *test\_and\_set* required a hardware supports that executes atomically. The *test\_and\_set* operation record the old value of flag, set the flag to true (no matter what it was before), and return the old value to *acquire()*. | |