DC126732

Huang Yanzhen

CISC2005 Lab7

|  |  |
| --- | --- |
| Question 1 | |
| Code | Screenshot |
| . . .  *// Enter critical region*  for (long i = 0; i < N\_ITERATIONS; i++)  {  *// Block on a semaphore count*  sem\_wait(&s); *// P(S);*  *// Shared resource*  counter++;  *// Increment a semaphore*  sem\_post(&s); *// V(S)*  }  . . .  *// Leave critical region*  *// Initialize the semaphore*  sem\_init(&s, 0, 1); | Not implementing Semaphore:  文本  描述已自动生成  Implementing Semaphore: |
| Answers | |
| **1.2 How the semaphore *s* changes as the program runs.**  Theoretically, 5 threads are generated, each performs the action of incrementing *counter* by 100000 (with 100000 iterations). So, the theoretical value of *counter* at last should be 500000. We take it as a reference *(There might be some errors probably caused by the ARM architecture of the CPU of my computer, and the online compiler doesn’t work with the code in any condition. Hence, I made a reference by NOT implementing the semaphore, allowing the existence of a race condition.)*  The semaphore is initialized with value 1, meaning it’s not locked. When *s* is unlocked, assume that thread T1 reaches the semaphore and locks it by *sem\_wait()*, who decrements the value of *s* by 1, resulting *s->val* to be 0.  Assumed that another thread T2 reaches *s* and attempts to decrease its value by *sem\_wait()*. However, at this time, T1 didn’t finish the execution in the critical region, hence *s->val* is still 0. Here, *sem\_wait()* will block T2’s execution until *s* is released. Totally having 5 threads, other thread may perform this action as well. Consequently, before T1 finishes its execution in the critical region, 4 thread would be in the waiting queue of the semaphore *s*.  When T1 finishes its execution, the semaphore *s* is released, where *sem\_post()* increases its value by 1, resulting *s->val=1*, indicating that *s* is in an unlock state. Hence, randomly one of the 4 waiting threads would be wakened and acquire the semaphore *s*, decreasing its value to 0. This repetition continues until all 5 thread finishes their work.  The semaphore would ultimately be 1, which is unlocked. It would be destroyed lastly. | |

|  |  |
| --- | --- |
| Question 2 | |
| Code | Screenshot |
| *. . .*  *// Simulate the process of service*  void \*get\_service(void \*arg)  {  int id = \*((int \*)arg);  *// Block on a semaphore count*  sem\_wait(&s);  *// Enter critical region*  printf("Customer [%d] is getting service.\n", id);  sleep(2);  printf("Customer [%d] has left.\n", id);  *// Leave critical region*  *// Increment a semaphore*  sem\_post(&s);  return 0;  }  . . .  *// Initialize the semaphore*  sem\_init(&s, 0, 2);  . . . | From GDB online compiler:  Text  Description automatically generated |
| Answers | |
| **2.2 How the semaphore *s* changes as the program runs.**  The semaphore *s* was initialized to be 5 (predefined by *NUM\_CUSTOMERS*). There are 5 threads: T1, T2, T3, T4 and T5, sequentially created and acquiring semaphore *s* by decreasing it by 1 using *sem\_wait()*. As *s* was initially 2, only two threads would be serviced at a time, decrementing *s->val* to 0, locking the semaphore, where the rest of the three waits. After the two threads ended its execution in the critical region, *s->val* would be increased to 2, where 3 threads would be attempting to acquire *s*, decrease it to 0 gain, leaving 1 process waiting. Finally, after the execution of two threads, *s* is increased to 2 again. Here, the last thread would acquire it, decreasing it to 1. After its execution, it is increased to 2 again. | |

|  |
| --- |
| Question 3 |
| Screenshot |
|  |
| Answers |
| Denote the two threads as T1 and T2.  Simultaneously, T1 acquired *A* and T2 acquired *B*. Then, T1 waits for T2 to release *B,* and T2 waits for T1 to release *A*. As a result, T1 and T2 are waiting for each other to stop, causing a deadlock. |

|  |
| --- |
| Question 4 |
| Screenshot |
| Graphical user interface, text  Description automatically generated |