

**Lab report**

|  |  |
| --- | --- |
| **Course**: | Operating System Principle |
| **Semester**: | 2nd semester of the academic year **2019-2020** |
| **Major**: | Software Engineering |
| **Class**: | 2019 |
| **Student Name**: | Fu Ruoxuan |
| **Student ID:** | 222019321062060 |
| **Teacher:** | ZHAO, Hengjun (赵恒军) |

**School of Computer and Information Science**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | | Process synchronization and deadlock | | | |
| Date | | May, 2021 | Type | | √ Confirmatory  √ Design  √ Comprehensive |
| 1. **Objective & Requirements**    1. Understand the concept of process synchronization; understand the concept of deadlock    2. Learn how to use the synchronization mechanisms provided in Linux and pthread library, i.e. the mutex lock    3. Can use mutex lock to solve real synchronization problems    4. Understand the implementation of mutex lock using inline assembly of C programming language | | | | | |
| 1. **Experimental environment (**platform and software**)**   VirtualBox+Ubuntu | | | | | |
| 1. **Experimental content and design** (Main Content, Procedure, Codes and Results) 2. Task 1   Implement a function called test\_and\_set() which can test and set the pass value to it **atomically**. Hint: using the atomic instruction on intel x86-64 architecture   * **lock cmpxchg *m64, r64***   The template and requirement of test\_and\_set() is as follows:    Please complete the missing inline assembly code inside asm();   1. Task 2   Based on your implemented test\_and\_set() function in Task 1, implement your own mutex lock mechanism   * Your lock variable is defined as a long integer with initial value 0   **long int lock = 0;**   * Your unlock() and lock() function takes the pointer to lock as the argument   **void my\_lock(long int \*lock);**  **void my\_unlock(long int \*lock);**   * Please use your implemented lock() and unlock() functions to solve the producer-consumer critical section problem.  1. Please provide your procedure to perform the tasks and source codes.   Task1   1. In task1, I should refine the code in the inline assembly to achieve the effect of the test\_n\_set function. The method is to change the value of the lock and the value of the local variable by comparing the incoming lock with the local variable using the lock cmpxchg instruction 2. #include <stdio.h> 4. **long** **int** test\_n\_set(**long** **int** \*lock) 5. { 6. **long** **int** res=0; 7. asm( 8. "mov rax, %1;" 9. "mov rbx, 1;" 10. "lock cmpxchg [%0],rbx;" 11. "mov %1,rax" 12. : 13. :"r"(lock),"m"(res) 14. :"rax","rbx" 15. ); 16. **return** res; 18. //if \*lock==0, then test\_n\_set() modifies \*lock to 1 and returns 0; 19. //if \*lock==1, then test\_n\_set() returns 1; 20. } 22. **int** main() 23. { 24. **long** **int** lock = 1; 26. **long** **int** res = test\_n\_set(&lock); 28. printf("lock: %ld; res: %ld\n", lock, res); 30. **return** 0; 31. } 32. Compile and run it.       Task2   1. In task2, I should use the function perfected in the first task to implement my own mutex lock. This is done by entering the zone while loop with the judgment condition that the function return value. After executing the code in the critical zone, reinitialize the lock to unlock it 2. #include <stdio.h> 3. #ifndef \_\_USE\_GNU 4. #define \_\_USE\_GNU 5. #endif 6. #include <unistd.h> 7. #include <sched.h> 8. #include <pthread.h> 10. **int** count; 12. **long** **int** lock; //delcare mutex 14. **void** \*producer(**void** \*param); /\* threads call this function \*/ 15. **void** \*consumer(**void** \*param); /\* threads call this function \*/ 16. **long** **int** test\_n\_set(**long** **int** \*lock); 18. **int** main(**int** argc, **char** \*argv[]) 19. { 20. pthread\_t tid1, tid2; /\* the thread identifier \*/ 21. pthread\_attr\_t attr; /\* set of thread attributes \*/ 22. pthread\_attr\_init(&attr); /\* get the default attributes \*/ 24. lock = 0; //initialization of mutex lock; 25. //NULL means using the default attribute 27. **while**(1) 28. { 29. count = 5; //init 31. pthread\_create(&tid1, &attr, producer, NULL); /\* create the thread \*/ 32. pthread\_create(&tid2, &attr, consumer, NULL); /\* create the thread \*/ 34. pthread\_join(tid1, NULL); 35. pthread\_join(tid2, NULL); 37. printf("count is: %d\n", count); 39. **if**(count == 4 || count == 6) 40. { 41. printf("press enter to continue\n"); 42. getchar(); 43. } 45. }//end-of-while 46. } 48. **long** **int** test\_n\_set(**long** **int** \*lock) 49. { 50. **long** **int** res=0; 51. asm( 52. "mov rax, %1;" 53. "mov rbx, 1;" 54. "lock cmpxchg [%0],rbx;" 55. "mov %1,rax" 56. : 57. :"r"(lock),"m"(res) 58. :"rax","rbx" 59. ); 60. **return** res; 62. //if \*lock==0, then test\_n\_set() modifies \*lock to 1 and returns 0; 63. //if \*lock==1, then test\_n\_set() returns 1; 64. } 66. **void** \*producer(**void** \*param) 67. { 68. cpu\_set\_t cpuSet; 69. CPU\_ZERO(&cpuSet);     //clear 70. CPU\_SET(0, &cpuSet);   //set core 0 71. sched\_setaffinity(0, **sizeof**(cpuSet), &cpuSet); 73. **while**(test\_n\_set(&lock)); //lock 74. count++; //produce one element 75. lock = 0; //unlock 77. pthread\_exit(0); 78. } 80. **void** \*consumer(**void** \*param) 81. { 82. cpu\_set\_t cpuSet; 83. CPU\_ZERO(&cpuSet);     //clear 84. CPU\_SET(1, &cpuSet);   //set core 1 85. sched\_setaffinity(0, **sizeof**(cpuSet), &cpuSet); 87. **while** (count == 0); 89. **while**(test\_n\_set(&lock)); //lock 90. count--; //consume one element 91. lock = 0; //unlock 93. pthread\_exit(0); 94. } 95. Compile and run it. And it can be seen that the value of count is right. | | | | | |
| 1. **Result analysis and discussion**（Analysis of experimental results and summing up the harvest and the existing problems）   **Analysis of experimental results**  The first task's lock has the correct value update and return value; the second task's mutually exclusive lock works. The experiment works well overall  **Harvest**  Modern linux systems make extensive use of concurrent programming to share resources, but if the wrong access pattern is created, it is likely to generate data update exceptions. This situation is called "Race Condition". This experiment introduces the use of mutually exclusive locks to resolve value update exceptions.  The rationale for the value update exception is that assembly code compiled from a high-level programming language is often more lines than a high-level programming language. Due to the concurrency of processes, the order of instruction execution of two processes affects the update of values.  We will call the region that causes the value update error the critical region. The entry zone is introduced before the critical zone for locking and the exit zone for unlocking. This mutual exclusion locking is used to achieve that only one process can access a value update to avoid value update errors.  You can use the mutex locks that come with the c library, or you can implement them yourself.The same problem exists when you implement your own mutex lock as when you update the value, that is, the order of execution of the assembly code will cause the locking to fail. The way to solve this problem is to use inline assembly.  Inline assembly can be used to compare incoming locks and temporary variables with lock cmpxchg to achieve the effect of the test\_n\_set function.  Existing problems  None | | | | | |
| Comments & Evaluation | Content & Design (A-E) | | |  | |
| Procedure & Codes (A-E) | | |  | |
| Results (A-E) | | |  | |
| Analysis & Discussion (A-E) | | |  | |
| Score (A-E):  Feedback comments: | | | | |