CS241 #15   
Condition Variables. Implement a Mutex Lock.  
The Critical Section Problem

1. How do I block a thread (= send it to 'sleep')?

2. How do I wake up threads that are blocked on a condition variable?

Example: Fix the following methods using a condition variable and mutex lock to ensure the cake integer is never negative.

|  |
| --- |
| 1. pthread\_mutex\_t m = PTHREAD=MUTEX\_INITIALIZER; 2. pthread\_cond\_t cv = PTHREAD\_COND\_INITIALIZER; 3. int cake = 0; 4. void decrement() { // Will block if zero 5. while(cake == 0) { 6. sleep(1) 7. } 8. cake --; 9. } 10. void increment() { 11. cake ++; 12. } |

3. How does pthread\_cond\_wait *really* work?

4. Challenge. A fixed size stack:

|  |
| --- |
| 1. pthread\_mutex\_t m = PTHREAD=MUTEX\_INITIALIZER; 2. pthread\_cond\_t cv = PTHREAD\_COND\_INITIALIZER; 3. double array[10]; 4. int n = 0; 5. // blocks while full (n ==10) 6. void push(double v) { 7. } 8. // blocks while empty (n == 0) 9. double pop() { 10. } 11. // Test with 2+ threads that add values... 12. void\* generator(void\*){ 13. for(int i =0; i < 10000; i++) 14. push( i); 15. return; 16. } 17. // And one thread that remove values 18. void \* consumer(void\*result) { 19. double sum = 0, i=0; 20. while( (i=pop() != -1) sum += i; 21. printf("%.0f", sum); 22. } |

**How can you *implement* a reliable mutex lock?**

**5. Let's try writing a simple implementation...**

1. pthread\_mutex\_init(int \* m) { \*m= 0; }
2. pthread\_mutex\_lock(int\* m) {
3. while(\*m ==1) {
4. pthread\_yield(); /\*sleeps for a short time \*/
5. }
6. \*m = 1;
7. }
8. pthread\_mutex\_unlock(int\* m) { ? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_}

Problems?

**6. CPU support: Use an atomic CPU instruction.**Suppose a special ‘*Atomic\_Exchange’* instruction 'exch' exists that swaps the values at two addresses as an *uninterruptable* operation

1. pthread\_mutex\_init(int\* m) { \*m= 0;}
2. pthread\_mutex\_lock(int\* m) {   
    for(int q = 1; q ; ) { *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_* }  
    }
3. pthread\_mutex\_unlock(int\* m) { *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_* }

**7. The Critical Section Problem**

while(running) {

1. Wait to enter the critical section if another thread is in the CS.

2. Critical Section Code here. Only one thread in here at a time!

3. Leave critical section. Allow another waiting thread to enter.

4. // do other stuff most of the time

}

~~ Welcome to the **Critical Section Problem** game show! ~~

Today's prizes: mutual exclusion and progress

Candidate #1. Use a single, boolean "flag"

boolean flag

|  |  |
| --- | --- |
| *Thread A*  wait while the flag is up  raise the flag!  *Critical Section* code here  lower the flag!  ... | *Thread B*  wait while the flag is up  raise the flag!  *Critical Section* code here  lower the flag! ... |

// Then each thread does other work but will repeat this again sometime in the future

Problems?

Candidate #2. Give each thread its own a flag.

boolean flagA, flagB

|  |  |
| --- | --- |
| wait while B's flag is up  raise A flag  *Critical Section* code here  lower A flag | wait while A's flag is up  raise B flag  *Critical Section* code here  lower B flag |

Problems?

Candidate #3. Change the sequence order

|  |  |
| --- | --- |
| raise A flag  wait until B flag is down   *Critical Section* code here lower A flag | raise B flag  wait until A flag is down  *Critical Section* code here lower B flag |

Problems?

Candidate #4. Try a single turn-based shared variable.

turn=1

|  |  |
| --- | --- |
| while( turn == 2) { }  *Critical Section* code here  turn = *2* | while( turn == 1) { }  *Critical Section* code here turn = 1 |

Problems?