CSCI 4730/6730 OS
(Chap #6 Synchronization Tools – Part 2)

In Kee Kim

Department of Computer Science

University of Georgia

Quiz on Tuesday

- Chapter 4
  1. What is Thread?
  2. Thread vs. Process? TCB vs. PCB?
  3. Memory Map with Threads
  4. Why context switching between threads is cheaper?
  5. Parallelism vs. Concurrency?
  6. Two Parallelisms?
  7. Amdahl's Law
  8. Kernel Thread vs. User Thread
  9. Thread Mapping Models

Quiz on Tuesday

- Chapter 5
  1. What is CPU scheduler? How it works?
  2. At least one scheduler question from uni-processor scheduling
  3. At least one scheduler question from real-time scheduling

Last Time

- Synchronization Motivation
- Race Condition
- Atomic Operation
- Critical Section
- Too Much Milk! Problem

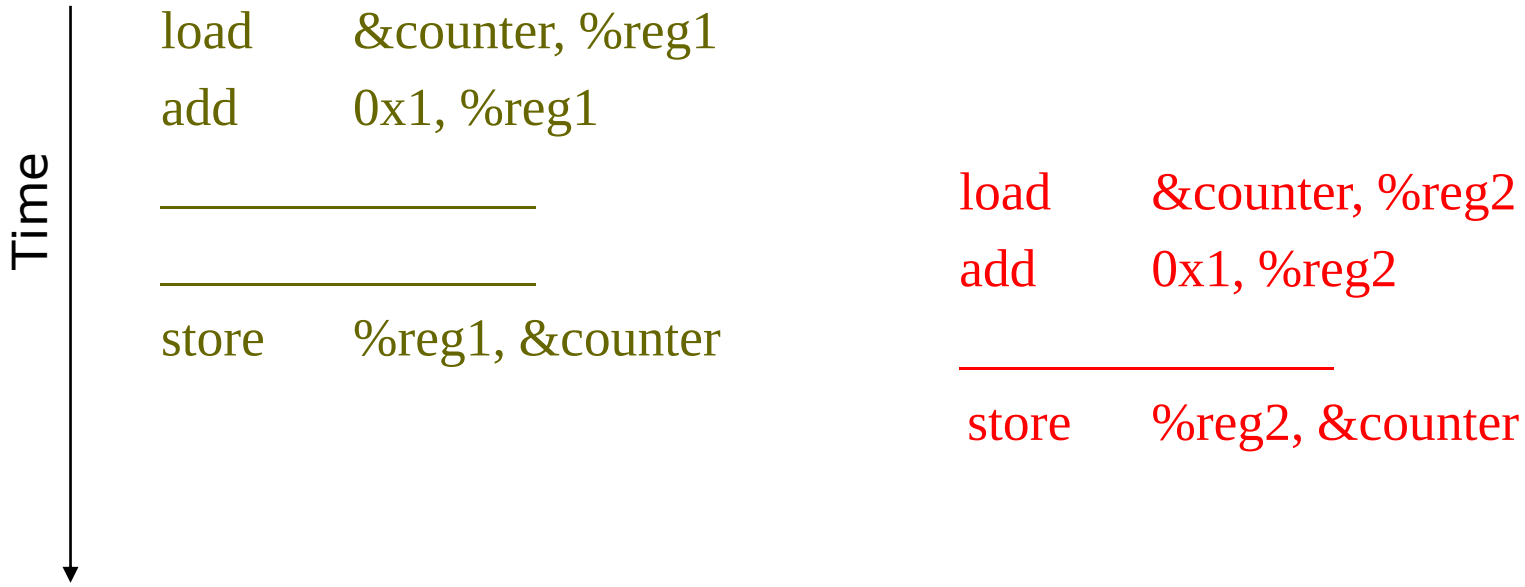
Recap: Synchronization Motivation

```
int counter = 0;
void *mythread(void *arg)
{
    int i;
    for (i=0; i<10000; i++){
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    ...
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    ...
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("counter = %d\n", counter);
    return 0;
}
```

Recap: Synchronization Motivation

```
S0: T1 executes register1 = counter          {register1 = 0}
S1: T1 executes register1 = register1 + 1    {register1 = 1}
S2: T2 executes register2 = counter          {register2 = 0}
S3: T2 executes register2 = register2 + 1    {register2 = 1}
S4: T1 executes counter = register1          {counter = 1 }
S5: T2 executes counter = register2          {counter = 1}
```



Recap: Definitions

- Race Condition
  - Output of a concurrent program depends on the order of operations between threads
- Atomic Operation
  - "load-add-store" must be performed in a single step
  - "Atomic" in this context means "all or nothing"
- Critical Section
  - Piece of code that only one thread can execute at once!

Recap: Too Much Milk!

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

Too Much Milk!!!

Recap: Too Much Milk, Try #4

Thread A

```
leave note_A
while (note_B) // X
      do nothing;
if (!milk)
      buy milk;
remove note_A
```

Thread B

```
leave note_B
if (!note_A) {    // Y
     if (!milk)
          buy milk
}
remove note_B
```

Can guarantee at X and Y that either:

(i) Safe for me to buy

(ii) Other will buy, ok to quit

Recap: Too Much Milk! Lessons

- Solution is complicated
  - "Obvious" code often has bugs
- Modern compilers/architectures reorder instructions
  - Making reasoning even more difficult
- Generalizing to many threads/processors
  - Even more complex

Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common (global) variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design a protocol to solve this

Critical Section Problem

- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

Critical-Section Problem Solution

- ***Solution to critical-section problem*** must satisfy the following three requirements
  1. **Mutual exclusion**: only one thread does a particular thing at a time
  2. **Progress :** If no process is executing in its critical section, and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
  3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Requirement #2 – Progress

- If no process is executing in its critical section, and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- When Critical Section is available, OS should select the next process to enter Critical Section ASAP
- Otherwise, users will think system is freezing or slow

Requirement #3 –Bounded Waiting

To satisfy 1) and 2), OS should have a policy (# of times to enter CS) for allowing other processes to enter CS.

A bound must exist on the number of times that other processes are allowed to enter their critical sections

after a process has made a request to enter its critical section and before that request is granted

1) Processes, who want to enter CS, have some idea ("prediction") when they will enter CS.

2) Processes, who want to enter CS, should not wait forever!

Too Much Milk, Try #5

Shared Variables

```
bool flag[2] = {false, false};
int turn;
```

Thread 0

```
flag[0] = true;
turn = 1;
while (flag[1] == true && turn == 1)
{
    // busy wait
}
if (!milk)
    buy milk;
flag[0] = false;
```

Thread 1

```
flag[1] = true;
turn = 0;
while (flag[0] == true && turn == 0)
{
    // busy wait
}
if (!milk)
    buy milk;
flag[1] = false;
```

This is called Peterson's Solution

- A classical SW-based solution for **critical section**

Again, Peterson's Solution

Shared Variables

```
bool flag[2] = {false, false};
int turn;
```

Thread 0

```
flag[0] = true;
turn = 1;
while (flag[1] == true && turn == 1)
{
    // busy wait
}
// critical section
...
// end of critical section
flag[0] = false;
```

Thread 1

```
flag[1] = true;
turn = 0;
while (flag[0] == true && turn == 0)
{
    // busy wait
}
// critical section
...
// end of critical section
flag[1] = false;
```

Assumption: changes to the variables **turn**, **flag[0]**, and **flag[1]** propagate immediately and atomically.

Is Peterson's solution correct?

- How to determine whether the solution is correct?
- The approach should satisfy
  - **Mutual exclusion**
  - **Progress**
  - **Bounded Waiting**

Is Peterson's solution correct?

- Yes
- Why is Peterson's solution correct?
  - **Mutual exclusion**
  - **Progress**
  - **Bounded Waiting**

Shared Variables

```
bool flag[2] = {false, false};
int turn;
```

Thread 0

```
flag[0] = true;
turn = 1;
while (flag[1] == true && turn == 1)
{
    // busy wait
}
// critical section
...
// end of critical section
flag[0] = false;
```

Thread 1

```
flag[1] = true;
turn = 0;
while (flag[0] == true && turn == 0)
{
    // busy wait
}
// critical section
...
// end of critical section
flag[1] = false;
```

Example – Peterson's Solution

nike.cs.uga.edu - PuTTY

```
 1 #include <stdio.h>
 2 #include <pthread.h>
 3
 4 int flag[2];
 5 int turn;
 6 int count = 0;
 7
 8 void lock_init()
 9 {
10   flag[0] = flag[1] = 0;
11   turn = 0;
12 }
13
14 void lock(int self)
15 {
16   flag[self] = 1;
17   turn = 1-self;
18
19   while (flag[1-self] == 1 && turn == 1-self) ;
20 }
21
22 void unlock(int self)
23 {
24   flag[self] = 0;
25 }
26
27 void* func(void *s)
28 {
29   int i = 0;
30   int self = (int *)s;
31   printf("Thread Entered: %d\n", self);
32
33   for (i=0; i< 1000000; i++) {
34     lock(self);
35     count++;
36     unlock(self);
37   }
38 }
39
                                    38,1          Top
```

nike.cs.uga.edu - PuTTY

```
40 int main()
41 {
42   pthread_t p1, p2;
43   lock_init();
44
45   pthread_create(&p1, NULL, func, (void*)0);
46   pthread_create(&p2, NULL, func, (void*)1);
47
48   pthread_join(p1, NULL);
49   pthread_join(p2, NULL);
50
51   printf("Actual Count: %d\n", count);
52   return 0;
53 }
54
                                    54,0-1        Bot
```

Limitation?

- Peterson's solution may not be correct on modern computer architecture
- **(Performance)** Modern processors and compilers may reorder read and write operation when having no-dependencies

Thread 1

```
flag[0] = true;
turn = 1;
while (flag[1] == true && turn == 1)
{
    // busy wait
}
// critical section
...
```

Thread 2

```
flag[1] = true;
turn = 0;
while (flag[0] == true && turn == 0)
{
    // busy wait
}
// critical section
...
```

What are possible consequences?

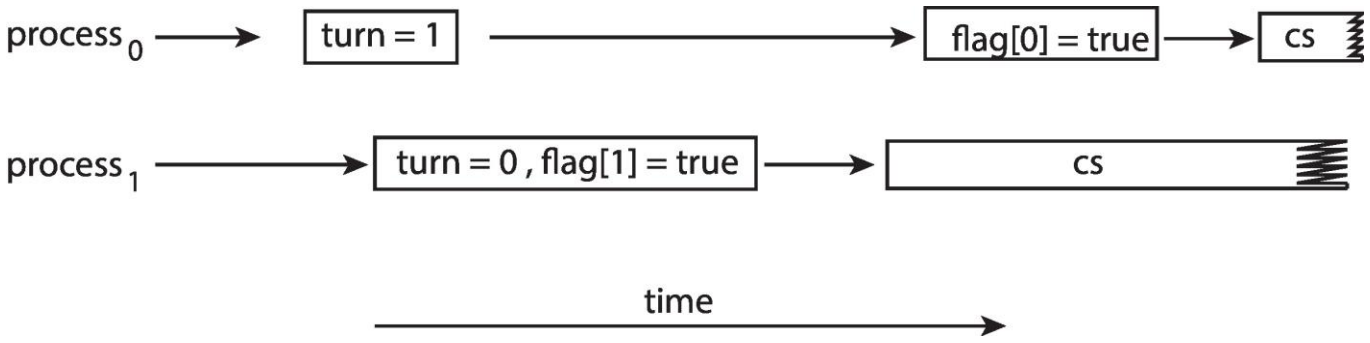
Thread 1

```
flag[0] = true;  4)
turn = 1;        1)
while (flag[1] == true && turn == 1)
{
    // busy wait
}
// critical section
...
```

Thread 2

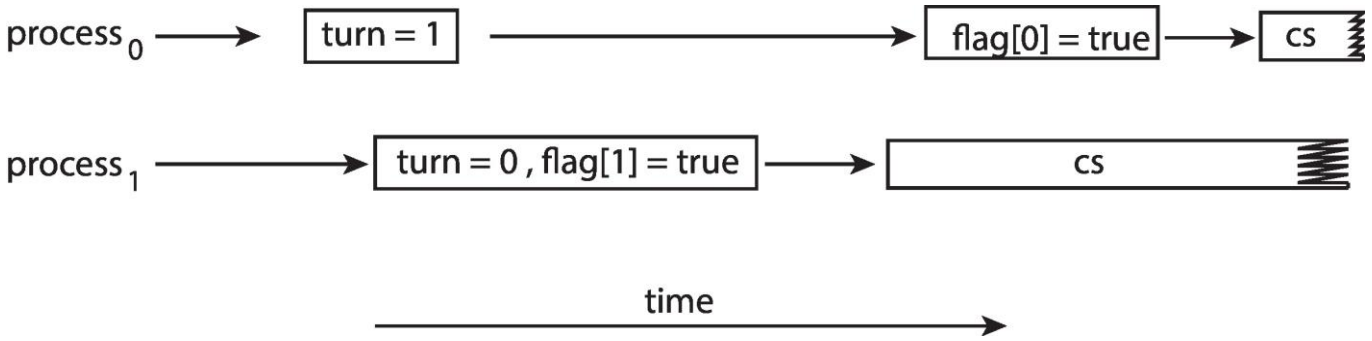
```
flag[1] = true;  3)
turn = 0;        2)
while (flag[0] == true && turn == 0)
{
    // busy wait
}
// critical section
...
```

What happens if the execution sequence is like 1) → 2) → 3) → 4)



Limitation?

- The effects of instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier**.
- Can you find another limitation?

Memory Barrier

- To ensure that Peterson's solution will work correctly on modern computer architecture we must use Memory Barrier.
- Two Memory models
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

Memory Barrier Instructions

- When a memory barrier instruction is performed, the system ensures that ***all loads and stores are completed before any subsequent load or store operations are performed***.
- Therefore, even if instructions were reordered, the ***memory barrier ensures that the store operations are completed in memory and visible to other processors*** before future load or store operations are performed.

Memory Barrier Example

nike.cs.uga.edu - PuTTY

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  int flag[2];
5  int turn;
6  int count = 0;
7
8  void lock_init()
9  {
10     flag[0] = flag[1] = 0;
11     turn = 0;
12 }
13
14 void lock(int self)
15 {
16     flag[self] = 1;
17     turn = 1-self;
18
19     asm volatile ("mfence");
20
21     while (flag[1-self] == 1 && turn == 1-self) ;
22 }
23
24 void unlock(int self)
25 {
26     flag[self] = 0;
27 }
28
29 void* func(void *s)
30 {
31     int i = 0;
32     int self = (int *)s;
33     printf("Thread Entered: %d\n", self);
34
35     for (i=0; i< 1000000; i++)
36         {
37             lock(self);
38       count++;
39             unlock(self);
40         }
41 }
                                                    1,1           Top
```

nike.cs.uga.edu - PuTTY

```
42
43 int main()
44 {
45     // Initialized the lock then fork 2 threads
46     pthread_t p1, p2;
47     lock_init();
48
49     // Create two threads (both run func)
50     pthread_create(&p1, NULL, func, (void*)0);
51     pthread_create(&p2, NULL, func, (void*)1);
52
53     // Wait for the threads to end.
54     pthread_join(p1, NULL);
55     pthread_join(p2, NULL);
56
57     printf("Actual Count: %d\n", count);
58
59     return 0;
60 }
                                                    60,1          Bot
```

Hardware Instructions

- Note – These approach may not be accessible to application programmers
- HW Instructions
  - **Test-and-Set** instruction
  - **Compare-and-Swap** instruction

Test-and-Set

- Special HW instructions
- Shared lock variable. The value can be either **T** or **F**
- Before entering into the critical section, process needs to check the lock value
  - False: Can enter the critical section
  - True: Busy waiting (Someone else is in CS)

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

1. Executed ***atomically***
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to **`true`**

While you have two cores, Two cannot run `test_and_set()` in parallel on two different cores.

Test-and-Set

Initial value of lock is False

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Proc #1

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Proc #2

Test-and-Set

- Discussion questions
  - Does “test-and-set” solve the critical-section problem?
    - Guarantees Mutual Exclusion and Progress
    - How about “bounded waiting” – hint: what if you have three procs?
  - Starvation

Compare-and-Swap

- Generalized version of test-and-set

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

```
    while (true) {
       while (compare_and_swap(&lock, 0, 1) != 0)
          ; /* do nothing */

          /* critical section */

       lock = 0;

          /* remainder section */
    }
```

Software Tools for Synchronization

- Mutex Locks
- Semaphores

- Sync tools for application programmers

Mutex Locks

- Mutex; **Mut**ual **Ex**clusion
  - To protect critical section and prevent race condition
- Simplest is **`mutex`** lock
  - Boolean variable indicating if lock is *available* or not
- Protect a critical section by
  - First **`acquire()`** a lock
  - Then **`release()`** the lock

Mutex Locks

- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;
}

release() {
   available = true;
}
```

```
while (true) {
        acquire lock

            critical section

        release lock

remainder section
}
```

Too Much Milk, Try #4

Thread A

```
leave note_A
while (note_B) // X
      do nothing;
if (!milk)
      buy milk;
remove note_A
```

Thread B

```
leave note_B
if (!note_A) {    // Y
     if (!milk)
          buy milk
}
remove note_B
```

Too Much Milk, Try #6

- Locks allow concurrent code to be much simpler:

Thread A

```
lock.acquire();
if (!milk)
  buy milk
lock.release();
```

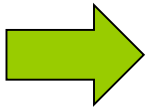
Thread B

```
lock.acquire();
if (!milk)
  buy milk
lock.release();
```

Mutex Lock Example

ik2sb@DESKTOP-J70810U: ~

```
 1 #include <stdio.h>
 2 #include <pthread.h>
 3
 4 int counter = 0;
 5
 6 void *mythread(void *arg)
 7 {
 8     int i;
 9
10     for (i = 0; i < 10000; i++) {
11         counter++;
12     }
13
14     return NULL;
15 }
16
17 int main(int argc, char *argv[])
18 {
19     pthread_t p1, p2;
20     printf("main: begin (counter = %d)\n", counter);
21     pthread_create(&p1, NULL, mythread, "A");
22     pthread_create(&p2, NULL, mythread, "B");
23
24     // join waits for the threads to finish
25     pthread_join(p1, NULL);
26     pthread_join(p2, NULL);
27     printf("main: done (counter = %d)\n", counter);
28     return 0;
29 }
30
31
~
~
~
~
~
~
~
~
~
~
"sync.c" 31L, 552C                    29,1          All
```



ik2sb@DESKTOP-J70810U: ~

```
 1 #include <stdio.h>
 2 #include <pthread.h>
 3
 4 int counter = 0;
 5 pthread_mutex_t lock;
 6
 7 void *mythread(void *arg)
 8 {
 9     int i;
10
11     for (i = 0; i < 10000; i++) {
12       pthread_mutex_lock(&lock);
13       counter++;
14       pthread_mutex_unlock(&lock);
15     }
16
17     return NULL;
18 }
19
20 int main(int argc, char *argv[])
21 {
22     pthread_t p1, p2;
23
24     if (pthread_mutex_init(&lock, NULL) != 0)
25     {
26       printf("mutex init failed\n");
27       return 1;
28     }
29
30     printf("main: begin (counter = %d)\n", counter);
31     pthread_create(&p1, NULL, mythread, "A");
32     pthread_create(&p2, NULL, mythread, "B");
33
34     // join waits for the threads to finish
35     pthread_join(p1, NULL);
36     pthread_join(p2, NULL);
37     printf("main: done (counter = %d)\n", counter);
38     return 0;
39 }
40
                                      1,1           Top
```

Rules for Using (Mutex) Locks

1. Lock is initially free
2. Always acquire before accessing shared data structure
3. Always release after finishing with shared data
4. **Never access shared data without lock**

```
int global_var = 0;

/* Thread 1 */                  /* Thread 2 */

int r = global_var;             int r = global_var;
r = r + 1;                      r = r + 1;
global_var = r;                 global_var = r;
```

Locks

```
/* Thread 1 */          /* Thread 2 */

lock(l);                lock(l);
int r = global_var;     int r = global_var;
unlock(l);              unlock(l);

r = r + 1;              r = r + 1;

lock(l);                lock(l);
global_var = r;         global_var = r;
unlock(l);              unlock(l);
```

How about this?

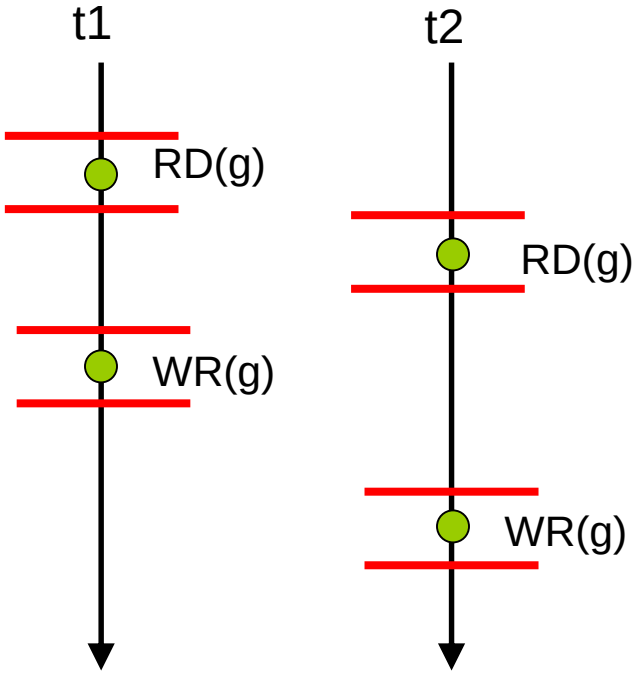
Locks

```
/* Thread 1 */          /* Thread 2 */

lock(l);                lock(l);
int r = global_var;     int r = global_var;
unlock(l);              unlock(l);

r = r + 1;              r = r + 1;

lock(l);                lock(l);
global_var = r;         global_var = r;
unlock(l);              unlock(l);
```




```
/* Thread 1 */          /* Thread 2 */

lock(l);                lock(l);
int r = global_var;     int r = global_var;
r = r + 1;              r = r + 1;
global_var = r;         global_var = r;
unlock(l);              unlock(l);
```

Locks

```
/* Thread 1 */

(1) if(ptr != NULL)
(4)   ptr->value = 10;
```

```
/* Thread 2 */

(2) if(ptr != NULL)
(3)   free(ptr);
```

Is this OK?

Locks

```
/* Thread 1 */

global++;
/* other tasks */

global2++;
```

```
/* Thread 2 */

global++;
/* other tasks */

global2++;
```

```
/* Thread 1 */

lock(g1);
global++;
unlock(g1);
/* other tasks */

lock(g2);
global2++;
unlock(g2);
```

```
/* Thread 2 */

lock(g1);
global++;
unlock(g1);
/* other tasks */

lock(g2);
global2++;
unlock(g2);
```