Operating Systems
Lecture 19: Locks

Nipun Batra
Oct 16, 2018

Locks

Thread 1

cc7:	mov 0x20135f,%eax
ccd:	add $0x1,%eax
cd0:	mov %eax,0x20135f

Thread 2

cc7:	mov 0x20135f,%eax
ccd:	add $0x1,%eax
cd0:	mov %eax,0x20135f

Locks

Thread 1

```
cc7:  mov    0x20135f,%eax
ccd:  add    $0x1,%eax
cd0:  mov    %eax,0x20135f
```

Thread 2

```
cc7:  mov    0x20135f,%eax
ccd:  add    $0x1,%eax
cd0:  mov    %eax,0x20135f
```

- Thread 1 checks if lock is free

Locks

Thread 1

Thread 2

- Thread 1 checks if lock is free
- Lock is free, Thread 1 acquires the lock

cc7:	mov 0x20135f,%eax
ccd:	add $0x1,%eax
cd0:	mov %eax,0x20135f

cc7:	mov 0x20135f,%eax
ccd:	add $0x1,%eax
cd0:	mov %eax,0x20135f

Locks

Thread 1

Thread 2

- Thread 1 checks if lock is free
- Lock is free, Thread 1 acquires the lock
- Thread 2 checks if lock is free

cc7:	mov 0x20135f,%eax
ccd:	add $0x1,%eax
cd0:	mov %eax,0x20135f

cc7:	mov 0x20135f,%eax
ccd:	add $0x1,%eax
cd0:	mov %eax,0x20135f

Locks

Thread 1

Thread 2

- Thread 1 checks if lock is free
- Lock is free, Thread 1 acquires the lock
- Thread 2 checks if lock is free
- Is not free; does not execute till lock free

cc7:	mov 0x20135f,%eax
ccd:	add $0x1,%eax
cd0:	mov %eax,0x20135f

cc7:	mov 0x20135f,%eax
ccd:	add $0x1,%eax
cd0:	mov %eax,0x20135f

Locks

Thread 1

Thread 2

- Thread 1 checks if lock is free
- Lock is free, Thread 1 acquires the lock
- Thread 2 checks if lock is free
- Is not free; does not execute till lock free

cc7:	mov 0x20135f,%eax
ccd:	add $0x1,%eax
cd0:	mov %eax,0x20135f

cc7:	mov 0x20135f,%eax
ccd:	add $0x1,%eax
cd0:	mov %eax,0x20135f

- Thread 1 executes

Locks

Thread 1

Thread 2

- Thread 1 checks if lock is free
- Lock is free, Thread 1 acquires the lock
- Thread 2 checks if lock is free
- Is not free; does not execute till lock free

cc7:	mov 0x20135f,%eax
ccd:	add $0x1,%eax
cd0:	mov %eax,0x20135f

cc7:	mov 0x20135f,%eax
ccd:	add $0x1,%eax
cd0:	mov %eax,0x20135f

- Thread 1 executes
- Thread 1 Unlocks

Locks

Thread 1

Thread 2

- Thread 1 checks if lock is free
- Lock is free, Thread 1 acquires the lock
- Thread 2 checks if lock is free
- Is not free; does not execute till lock free

```
cc7:  mov   0x20135f,%eax
ccd:  add   $0x1,%eax
cd0:  mov   %eax,0x20135f
```

```
cc7:  mov   0x20135f,%eax
ccd:  add   $0x1,%eax
cd0:  mov   %eax,0x20135f
```

- Thread 1 executes
- Thread 1 Unlocks
- Thread 2 checks (keeps on doing so) for lock being free

Locks

Thread 1

Thread 2

- Thread 1 checks if lock is free
- Lock is free, Thread 1 acquires the lock
- Thread 2 checks if lock is free
- Is not free; does not execute till lock free

cc7:	mov 0x20135f,%eax
ccd:	add $0x1,%eax
cd0:	mov %eax,0x20135f

cc7:	mov 0x20135f,%eax
ccd:	add $0x1,%eax
cd0:	mov %eax,0x20135f

- Thread 1 executes
- Thread 1 Unlocks
- Thread 2 checks (keeps on doing so) for lock being free
- Thread 2 executes and unlocks

Goals of a Lock

Goals of a Lock

- Mutual exclusion: Only a single thread can run the critical section at a time

Goals of a Lock

- Mutual exclusion: Only a single thread can run the critical section at a time
- Fairness: Each thread should get a fair chance of running the critical section. No starvation.

Goals of a Lock

- Mutual exclusion: Only a single thread can run the critical section at a time
- Fairness: Each thread should get a fair chance of running the critical section. No starvation.
- Performance: Low time overhead

Goals of a Lock

- Mutual exclusion: Only a single thread can run the critical section at a time
- Fairness: Each thread should get a fair chance of running the critical section. No starvation.
- Performance: Low time overhead
  - Performance overhead when:

Goals of a Lock

- Mutual exclusion: Only a single thread can run the critical section at a time
- Fairness: Each thread should get a fair chance of running the critical section. No starvation.
- Performance: Low time overhead
  - Performance overhead when:
    - Single thread, no contention

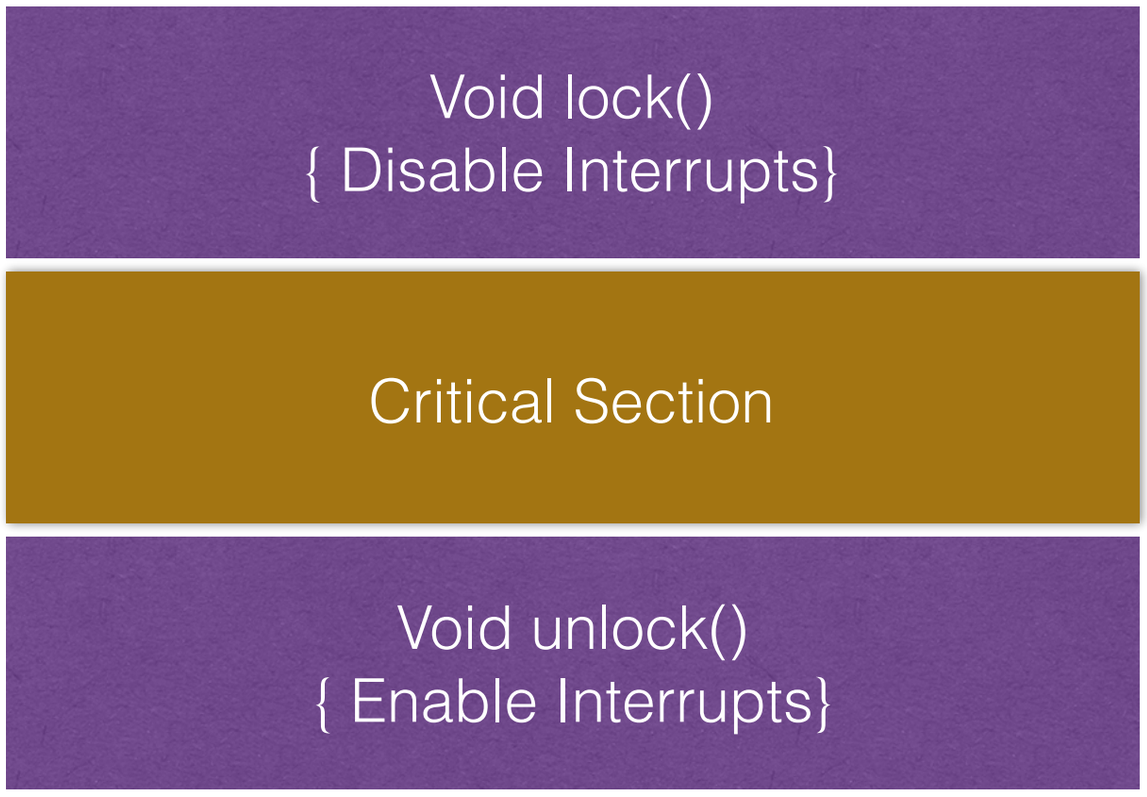
Goals of a Lock

- Mutual exclusion: Only a single thread can run the critical section at a time
- Fairness: Each thread should get a fair chance of running the critical section. No starvation.
- Performance: Low time overhead
  - Performance overhead when:
    - Single thread, no contention
    - Multiple threads, single CPU

Goals of a Lock

- Mutual exclusion: Only a single thread can run the critical section at a time
- Fairness: Each thread should get a fair chance of running the critical section. No starvation.
- Performance: Low time overhead
  - Performance overhead when:
    - Single thread, no contention
    - Multiple threads, single CPU
    - Multiple threads, multiple CPU

Building a Lock - Disable Interrupts



Building a Lock - Disable Interrupts

Building a Lock - Disable Interrupts

Pros

Building a Lock - Disable Interrupts

Pros

1. Simple and works!

Building a Lock - Disable Interrupts

Pros

1. Simple and works!

Cons

Building a Lock - Disable Interrupts

Pros

1. Simple and works!

Cons

1. Threads are given a lot of trust

Building a Lock - Disable Interrupts

Pros

1. Simple and works!

Cons

1. Threads are given a lot of trust
    1. Call lock() at starting of program and run infinitely

Building a Lock - Disable Interrupts

Pros

1. Simple and works!

Cons

1. Threads are given a lot of trust
    1. Call lock() at starting of program and run infinitely
2. Does not work on multiprocessors

Building a Lock - Disable Interrupts

Pros

1. Simple and works!

Cons

1. Threads are given a lot of trust
    1. Call lock() at starting of program and run infinitely
2. Does not work on multiprocessors
    1. Each processor will have own interrupts?!

Building a Lock - Disable Interrupts

Pros

1. Simple and works!

Cons

1. Threads are given a lot of trust
    1. Call lock() at starting of program and run infinitely
2. Does not work on multiprocessors
    1. Each processor will have own interrupts?!
3. Loss of interrupts

Building a Lock - Disable Interrupts

Pros

1. Simple and works!

Cons

1. Threads are given a lot of trust
    1. Call lock() at starting of program and run infinitely
2. Does not work on multiprocessors
    1. Each processor will have own interrupts?!
3. Loss of interrupts
4. Inefficient - Interrupt routines can be slow

Building a Lock - Load/Store or Flag

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section
  - Is flag set? (some other thread has critical section control)

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section
  - Is flag set? (some other thread has critical section control)
  - Yes - Spin waiting

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section
  - Is flag set? (some other thread has critical section control)
  - Yes - Spin waiting
  - No

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section
  - Is flag set? (some other thread has critical section control)
  - Yes - Spin waiting
  - No
    - set flag, execute critical section

Building a Lock - Load/Store or Flag

- Use a single flag to indicate if a thread has possession of critical section
- Thread calls lock before entering critical section
  - Is flag set? (some other thread has critical section control)
  - Yes - Spin waiting
  - No
    - set flag, execute critical section
    - After completion of critical section, unset flag

Building a Lock - Load/Store or Flag

Building a Lock - Load/Store or Flag

```
typedef struct __lock_t { int flag; } lock_t;
```

Building a Lock - Load/Store or Flag

```
typedef struct __lock_t { int flag; } lock_t;
```

```
void init(lock_t *mutex)
    { // 0 -> lock is available, 1 -> held
    mutex->flag = 0; }
```

Building a Lock - Load/Store or Flag

```
typedef struct __lock_t { int flag; } lock_t;
```

```
void init(lock_t *mutex)
    { // 0 -> lock is available, 1 -> held
    mutex->flag = 0; }
```

```
void lock(lock_t *mutex) {
    while (mutex->flag == 1);
    // spin-wait (do nothing)
    mutex->flag = 1; // now SET it!
    }
```

Building a Lock - Load/Store or Flag

```
typedef struct __lock_t { int flag; } lock_t;
```

```
void init(lock_t *mutex)
    { // 0 -> lock is available, 1 -> held
    mutex->flag = 0; }
```

```
void lock(lock_t *mutex) {
    while (mutex->flag == 1);
    // spin-wait (do nothing)
    mutex->flag = 1; // now SET it!
    }
```

```
void unlock(lock_t *mutex) { mutex->flag = 0; }
```

Building a Lock - Load/Store or Flag

Thread 1	Thread 2

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	
Lock held by some other thread	

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	
Lock held by some other thread	
while(flag ==1) // Busy spinning	

Building a Lock - Load/Store or Flag

Thread 1	Thread 2

Call Lock()

Lock held by some other thread

while(flag ==1) // Busy spinning

Other thread unlocks —> flag = 0

Building a Lock - Load/Store or Flag

Thread 1	Thread 2

Call Lock()

Lock held by some other thread

while(flag ==1) // Busy spinning

Other thread unlocks —> flag = 0

Context Switch

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	
Lock held by some other thread	
while(flag ==1) // Busy spinning	
Other thread unlocks —> flag = 0	
Context Switch	
	Call Lock()

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	
Lock held by some other thread	
while(flag ==1) // Busy spinning	
Other thread unlocks —> flag = 0	
Context Switch	
	Call Lock()
	while(flag ==1)

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	
Lock held by some other thread	
while(flag ==1) // Busy spinning	
Other thread unlocks —> flag = 0	
Context Switch	
	Call Lock()
	while(flag ==1)
	flag = 1

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	
Lock held by some other thread	
while(flag ==1) // Busy spinning	
Other thread unlocks —> flag = 0	
Context Switch	
	Call Lock()
	while(flag ==1)
	flag = 1
Context Switch	

Building a Lock - Load/Store or Flag

Thread 1	Thread 2
Call Lock()	
Lock held by some other thread	
while(flag ==1) // Busy spinning	
Other thread unlocks —> flag = 0	
Context Switch	
	Call Lock()
	while(flag ==1)
	flag = 1
Context Switch	
flag = 1	

Goals of a Lock

Goals of a Lock

- Mutual exclusion: X

Goals of a Lock

- Mutual exclusion: X
- Fairness: X

Goals of a Lock

- Mutual exclusion: X
- Fairness: X
- Performance: Spin Waiting is bad!

Goals of a Lock

- Mutual exclusion: X
- Fairness: X
- Performance: Spin Waiting is bad!

Goals of a Lock

- Mutual exclusion: X
- Fairness: X
- Performance: Spin Waiting is bad!

Goals of a Lock

- Mutual exclusion: X
- Fairness: X
- Performance: Spin Waiting is bad!

Need Hardware Support!

Atomic Instructions - Test & Set

```
1 int TestAndSet(int *ptr, int new) {
2   int old = *ptr;  // fetch old value at ptr
3   *ptr = new;  // store ‘new’ into ptr
4   return old;  // return the old value
5 }
```

Atomic Instructions - Test & Set

```
1 int TestAndSet(int *ptr, int new) {
2   int old = *ptr;  // fetch old value at ptr
3   *ptr = new;  // store ‘new’ into ptr
4   return old;  // return the old value
5 }
```

- Return old value pointed by ptr

Atomic Instructions - Test & Set

```
1 int TestAndSet(int *ptr, int new) {
2   int old = *ptr;  // fetch old value at ptr
3   *ptr = new;  // store ‘new’ into ptr
4   return old;  // return the old value
5 }
```

- Return old value pointed by ptr
- Simultaneously update to new

Atomic Instructions - Test & Set

```
1 int TestAndSet(int *ptr, int new) {
2   int old = *ptr;  // fetch old value at ptr
3   *ptr = new;  // store ‘new’ into ptr
4   return old;  // return the old value
5 }
```

- Return old value pointed by ptr
- Simultaneously update to new
- Performed Atomically and by Hardware!

Atomic Instructions - Test & Set

```
1 int TestAndSet(int *ptr, int new) {
2   int old = *ptr;  // fetch old value at ptr
3   *ptr = new;  // store ‘new’ into ptr
4   return old;  // return the old value
5 }
```

- Return old value pointed by ptr
- Simultaneously update to new
- **Performed Atomically and by Hardware!**
  - The above is just a software depiction

Atomic Instructions - Test & Set

```
1 typedef struct __lock_t {
2   int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6   // 0 indicates that lock is available,
7   // 1 that it is held
8   lock->flag = 0;
9 }
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12   ;  // spin-wait
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
18 }
```

Atomic Instructions - Test & Set

```
1 typedef struct __lock_t {
2   int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6   // 0 indicates that lock is available,
7   // 1 that it is held
8   lock->flag = 0;
9 }
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12   ;  // spin-wait
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
18 }
```

Define lock structure

Atomic Instructions - Test & Set

```
1 typedef struct __lock_t {
2   int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6   // 0 indicates that lock is available,
7   // 1 that it is held
8   lock->flag = 0;
9 }
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12   ;  // spin-wait
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
18 }
```

Init by setting flag to 0

Atomic Instructions - Test & Set

```
1 typedef struct __lock_t {
2   int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6   // 0 indicates that lock is available,
7   // 1 that it is held
8   lock->flag = 0;
9 }
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12   ;  // spin-wait
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
18 }
```

Atomic Instructions - Test & Set

```
1 typedef struct __lock_t {
2   int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6   // 0 indicates that lock is available,
7   // 1 that it is held
8   lock->flag = 0;
9 }
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12   ;  // spin-wait
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
18 }
```

Case 1: Lock not held by any thread

Atomic Instructions - Test & Set

```
1 typedef struct __lock_t {
2   int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6   // 0 indicates that lock is available,
7   // 1 that it is held
8   lock->flag = 0;
9 }
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12   ;  // spin-wait
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
18 }
```

Case 1: Lock not held by any thread

Atomic Instructions - Test & Set

```
1 typedef struct __lock_t {
2   int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6   // 0 indicates that lock is available,
7   // 1 that it is held
8   lock->flag = 0;
9 }
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12   ;  // spin-wait
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
18 }
```

Case 1: Lock not held by any thread

- old value of flag = 0

Atomic Instructions - Test & Set

```
1 typedef struct __lock_t {
2   int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6   // 0 indicates that lock is available,
7   // 1 that it is held
8   lock->flag = 0;
9 }
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12   ;  // spin-wait
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
18 }
```

Case 1: Lock not held by any thread

- old value of flag = 0
- Set flag to 1 and return 0 from test and set —> Current thread acquires lock

Atomic Instructions - Test & Set

```
1 typedef struct __lock_t {
2   int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6   // 0 indicates that lock is available,
7   // 1 that it is held
8   lock->flag = 0;
9 }
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12   ;  // spin-wait
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
18 }
```

Case 1: Lock not held by any thread

- old value of flag = 0
- Set flag to 1 and return 0 from test and set —> Current thread acquires lock
- No spin waiting for current thread

Atomic Instructions - Test & Set

```
1 typedef struct __lock_t {
2   int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6   // 0 indicates that lock is available,
7   // 1 that it is held
8   lock->flag = 0;
9 }
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12   ;  // spin-wait
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
18 }
```

Atomic Instructions - Test & Set

```
1 typedef struct __lock_t {
2   int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6   // 0 indicates that lock is available,
7   // 1 that it is held
8   lock->flag = 0;
9 }
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12   ;  // spin-wait
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
18 }
```

Case 2: Lock held by some other thread

Atomic Instructions - Test & Set

```
1 typedef struct __lock_t {
2   int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6   // 0 indicates that lock is available,
7   // 1 that it is held
8   lock->flag = 0;
9 }
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12   ;  // spin-wait
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
18 }
```

Case 2: Lock held by some other thread

Atomic Instructions - Test & Set

```
1 typedef struct __lock_t {
2   int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6   // 0 indicates that lock is available,
7   // 1 that it is held
8   lock->flag = 0;
9 }
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12   ;  // spin-wait
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
18 }
```

Case 2: Lock held by some other thread

- old value of flag = 1

Atomic Instructions - Test & Set

```
1 typedef struct __lock_t {
2   int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6   // 0 indicates that lock is available,
7   // 1 that it is held
8   lock->flag = 0;
9 }
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12   ;  // spin-wait
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
18 }
```

Case 2: Lock held by some other thread

- old value of flag = 1
- Set flag to 1 and return 1 from test and set

Atomic Instructions - Test & Set

```
1 typedef struct __lock_t {
2   int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6   // 0 indicates that lock is available,
7   // 1 that it is held
8   lock->flag = 0;
9 }
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12   ;  // spin-wait
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
18 }
```

Case 2: Lock held by some other thread

- old value of flag = 1
- Set flag to 1 and return 1 from test and set
- Spin waiting for current thread since it goes in while loop

Atomic Instructions - Test & Set

```
1 typedef struct __lock_t {
2   int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6   // 0 indicates that lock is available,
7   // 1 that it is held
8   lock->flag = 0;
9 }
10 void lock(lock_t *lock) {
11   while (TestAndSet(&lock->flag, 1) == 1)
12   ;  // spin-wait
13 }
14
15 void unlock(lock_t *lock) {
16   lock->flag = 0;
18 }
```

Once out of critical section, unset flag

Test & Set Evaluation

Test & Set Evaluation

- Mutual exclusion: Yes

Test & Set Evaluation

- Mutual exclusion: Yes
- Fairness: X

Test & Set Evaluation

- Mutual exclusion: Yes
- Fairness: X
- Performance: Spin Waiting is bad!

Test & Set Evaluation

- Mutual exclusion: Yes
- Fairness: X
- Performance: Spin Waiting is bad!
  - Single core: Each thread spins away its allotted time slot, eating away the time for the thread holding the critical section

Test & Set Evaluation

- Mutual exclusion: Yes
- Fairness: X
- Performance: Spin Waiting is bad!
  - Single core: Each thread spins away its allotted time slot, eating away the time for the thread holding the critical section
  - Multi core: If num threads ~ num cores

Test & Set Evaluation

- Mutual exclusion: Yes
- Fairness: X
- Performance: Spin Waiting is bad!
  - Single core: Each thread spins away its allotted time slot, eating away the time for the thread holding the critical section
  - Multi core: If num threads ~ num cores
    - Each thread waiting to acquire lock can spin on its core, not eating up the time needed (quick) for the critical section to execute on other

Atomic Instructions - Compare & Swap

```
1 int CompareAndSwap(int *ptr, int expected, int new) {
2   int actual = *ptr;
3   if (actual == expected)
4   *ptr = new;
5   return actual;
6 }
```

Atomic Instructions - Compare & Swap

```
1 int CompareAndSwap(int *ptr, int expected, int new) {
2   int actual = *ptr;
3   if (actual == expected)
4   *ptr = new;
5   return actual;
6 }
```

- Test whether value at address (ptr) is equal to expected

Atomic Instructions - Compare & Swap

```
1 int CompareAndSwap(int *ptr, int expected, int new) {
2   int actual = *ptr;
3   if (actual == expected)
4   *ptr = new;
5   return actual;
6 }
```

- Test whether value at address (ptr) is equal to expected
  - Yes

Atomic Instructions - Compare & Swap

```
1 int CompareAndSwap(int *ptr, int expected, int new) {
2   int actual = *ptr;
3   if (actual == expected)
4   *ptr = new;
5   return actual;
6 }
```

- Test whether value at address (ptr) is equal to expected
  - Yes
    - Set new value at address

Atomic Instructions - Compare & Swap

```
1 int CompareAndSwap(int *ptr, int expected, int new) {
2   int actual = *ptr;
3   if (actual == expected)
4   *ptr = new;
5   return actual;
6 }
```

- Test whether value at address (ptr) is equal to expected
  - Yes
    - Set new value at address
    - Return old value at address

Atomic Instructions - Compare & Swap

```
1 int CompareAndSwap(int *ptr, int expected, int new) {
2   int actual = *ptr;
3   if (actual == expected)
4   *ptr = new;
5   return actual;
6 }
```

- Test whether value at address (ptr) is equal to expected
  - Yes
    - Set new value at address
    - Return old value at address
  - No

Atomic Instructions - Compare & Swap

```
1 int CompareAndSwap(int *ptr, int expected, int new) {
2   int actual = *ptr;
3   if (actual == expected)
4   *ptr = new;
5   return actual;
6 }
```

- Test whether value at address (ptr) is equal to expected
  - Yes
    - Set new value at address
    - Return old value at address
  - No
    - Return old value at address