Operating Systems

Lecture 20: Locks

Nipun Batra

Oct 18, 2018

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

Atomic Instructions - Compare & Swap

```
1 void lock(lock_t *lock) {
2   while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3   ; // spin
4 }
```

Compare & Swap Evaluation

Need to add some ordering

Compare & Swap Evaluation

- Mutual exclusion: Yes

Need to add some ordering

Compare & Swap Evaluation

- Mutual exclusion: Yes
- Fairness: X **Need to add some ordering**

Compare & Swap Evaluation

- Mutual exclusion: Yes
- Fairness: X **Need to add some ordering**
- Performance: Spin Waiting is bad!

Compare & Swap Evaluation

- Mutual exclusion: Yes
- Fairness: X **Need to add some ordering**
- 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

Compare & Swap Evaluation

- Mutual exclusion: Yes
- Fairness: X **Need to add some ordering**
- 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

Compare & Swap Evaluation

- Mutual exclusion: Yes
- Fairness: X **Need to add some ordering**
- 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 - Fetch & Add

```
1 int FetchAndAdd(int *ptr) {
2   int old = *ptr;
3   *ptr = old + 1;
4   return old;
5 }
```

Atomic Instructions - Fetch & Add

```
1 int FetchAndAdd(int *ptr) {
2   int old = *ptr;
3   *ptr = old + 1;
4   return old;
5 }
```

- Atomically Increment the value and return the old value

Fairness - Ticket Lock

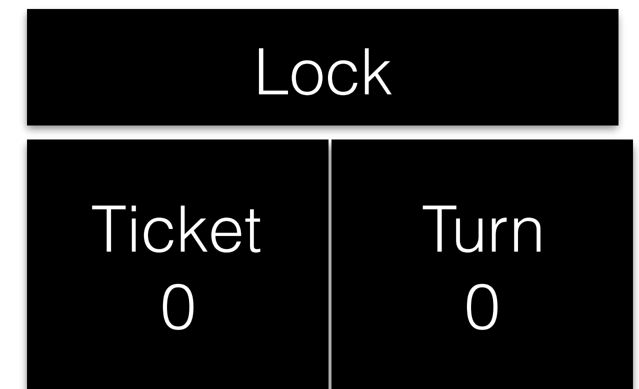
```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```

Fairness - Ticket Lock

```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```

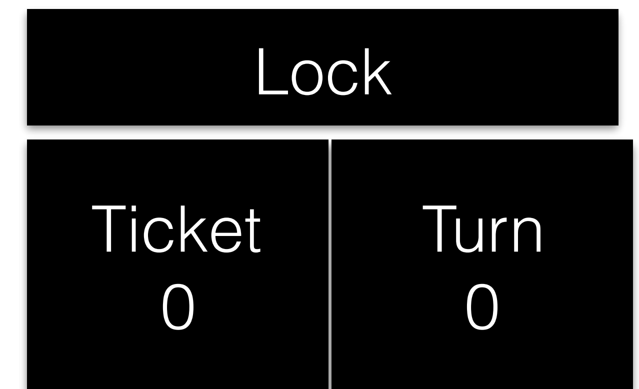
Fairness - Ticket Lock

```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```



Fairness - Ticket Lock

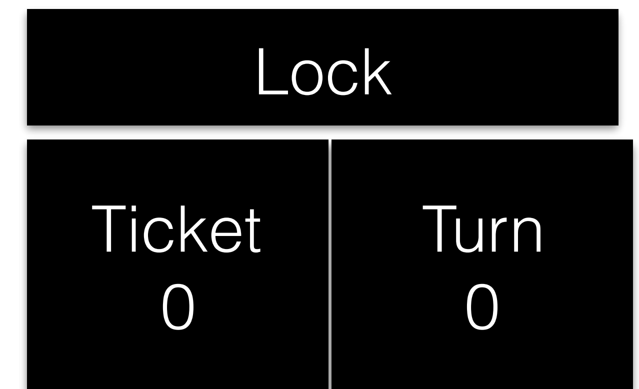
```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```



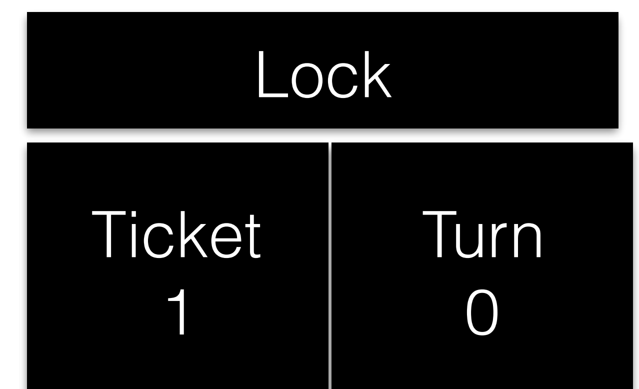
Thread T1 asks for lock

Fairness - Ticket Lock

```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```

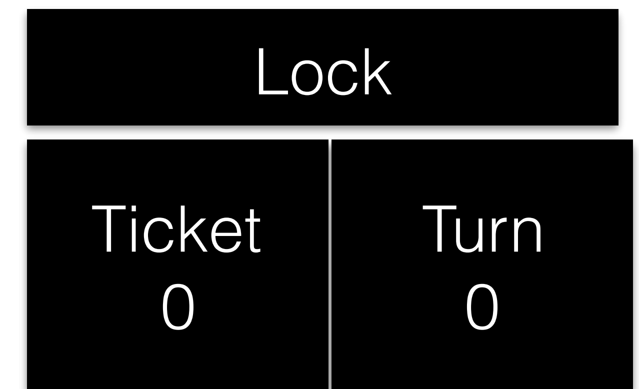


Thread T1 asks for lock

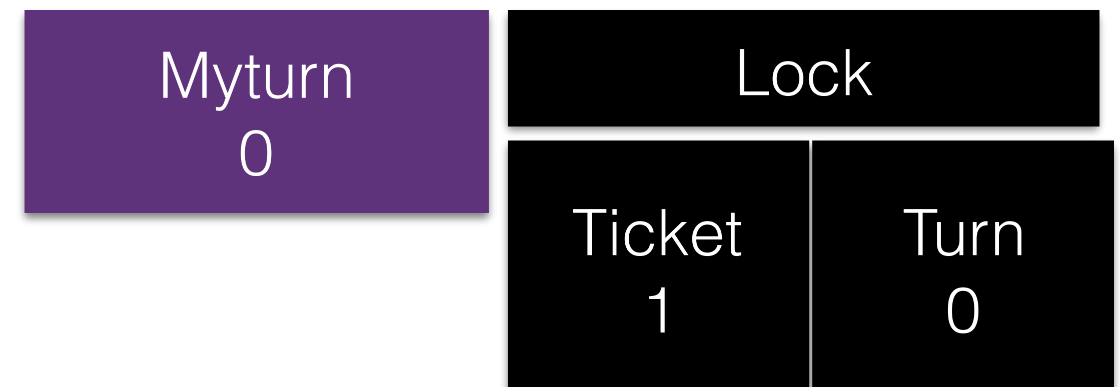


Fairness - Ticket Lock

```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```

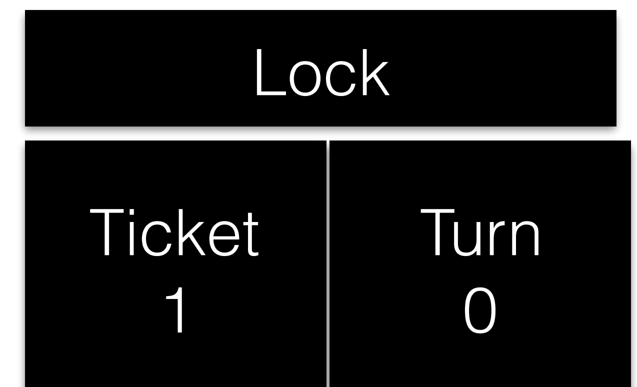


Thread T1 asks for lock



Fairness - Ticket Lock

```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```

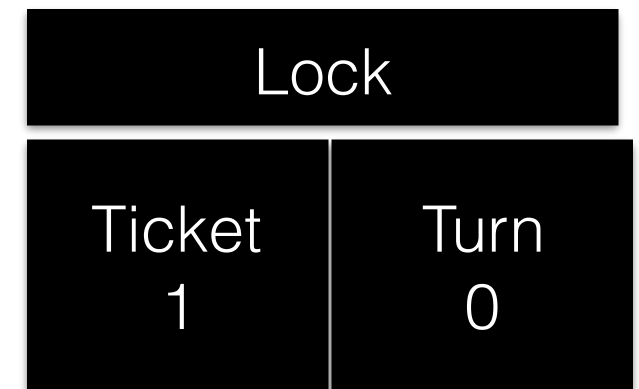


Fairness - Ticket Lock

```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```

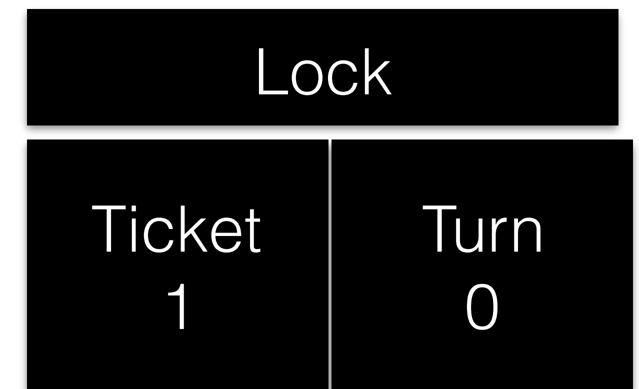
Fairness - Ticket Lock

```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```



Fairness - Ticket Lock

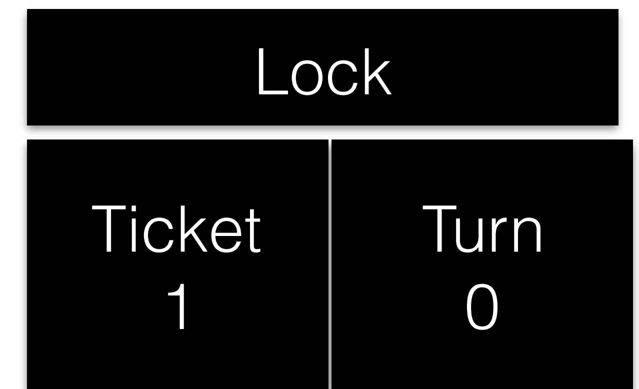
```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```



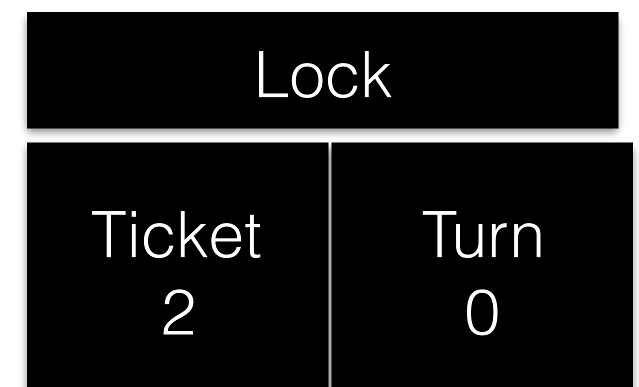
Thread T2 asks for lock

Fairness - Ticket Lock

```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```

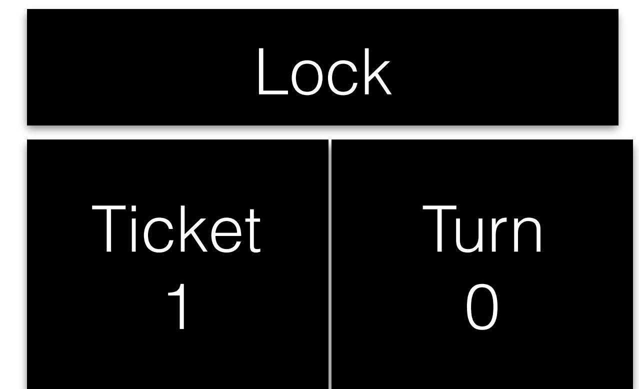


Thread T2 asks for lock

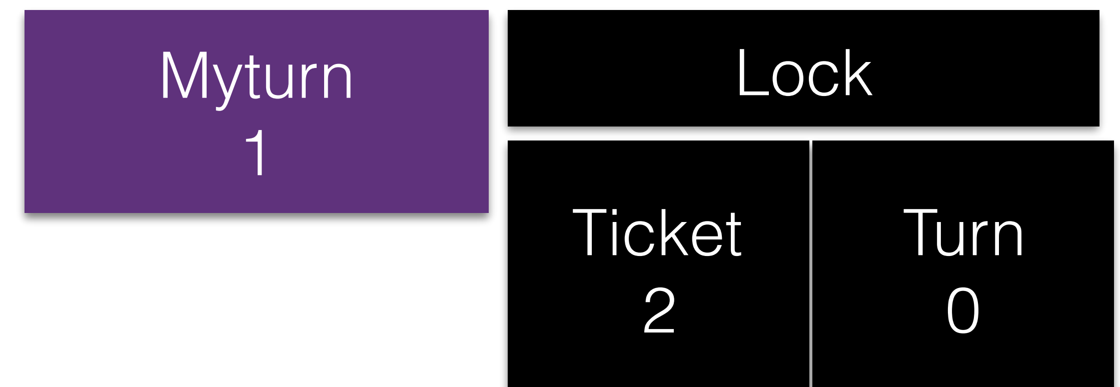


Fairness - Ticket Lock

```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```

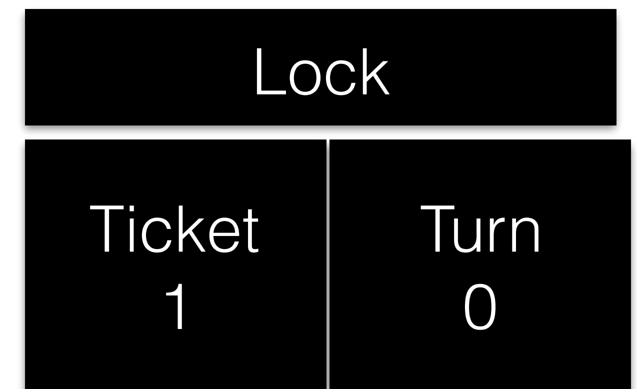


Thread T2 asks for lock

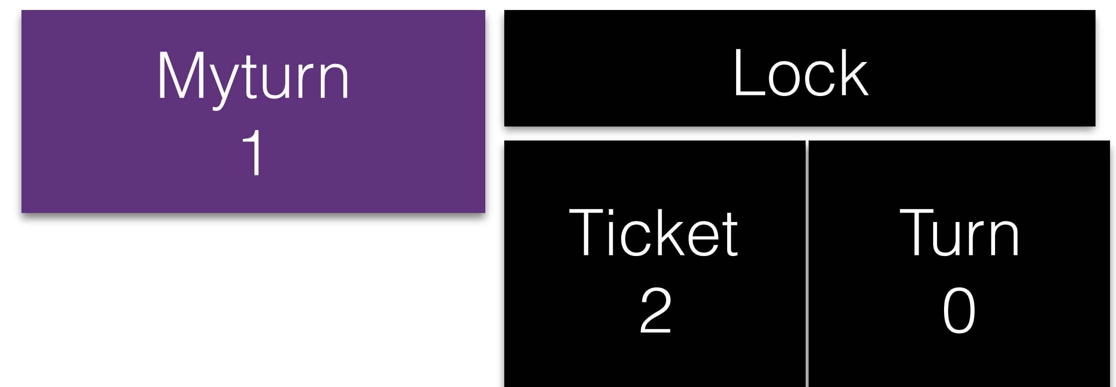


Fairness - Ticket Lock

```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```



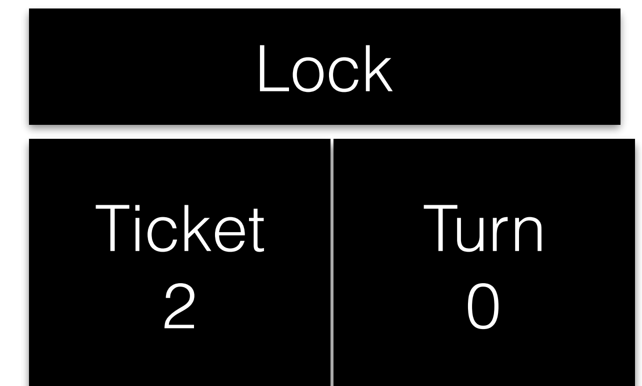
Thread T2 asks for lock



Thread T2 waits …

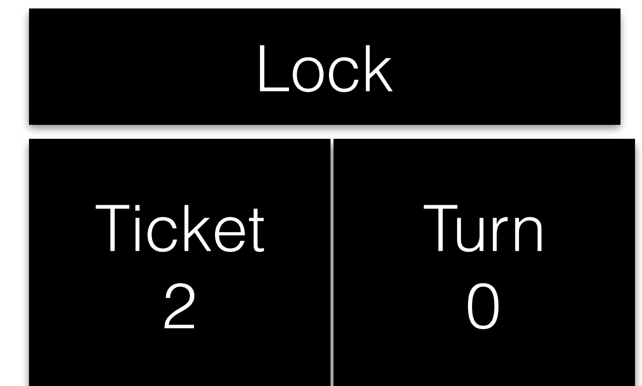
Fairness - Ticket Lock

```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```



Fairness - Ticket Lock

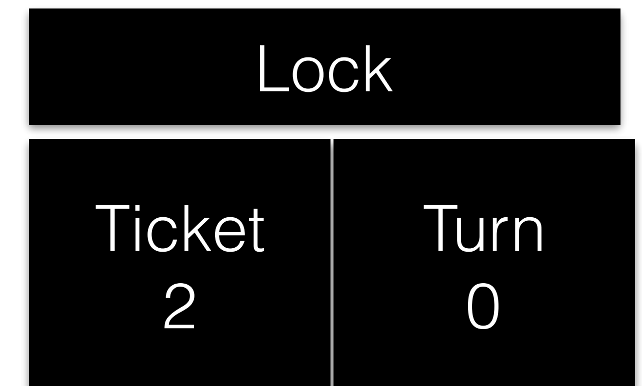
```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```



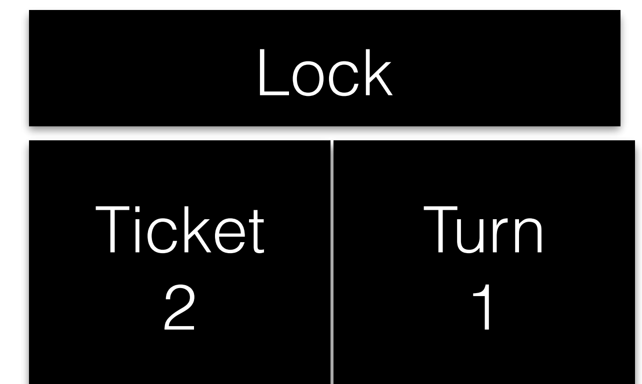
Thread T1 finishes critical section

Fairness - Ticket Lock

```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```



Thread T1 finishes critical section



Fairness - Ticket Lock

```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```

Fairness - Ticket Lock

```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```

Thread T2 can run now

Fairness - Ticket Lock

```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```

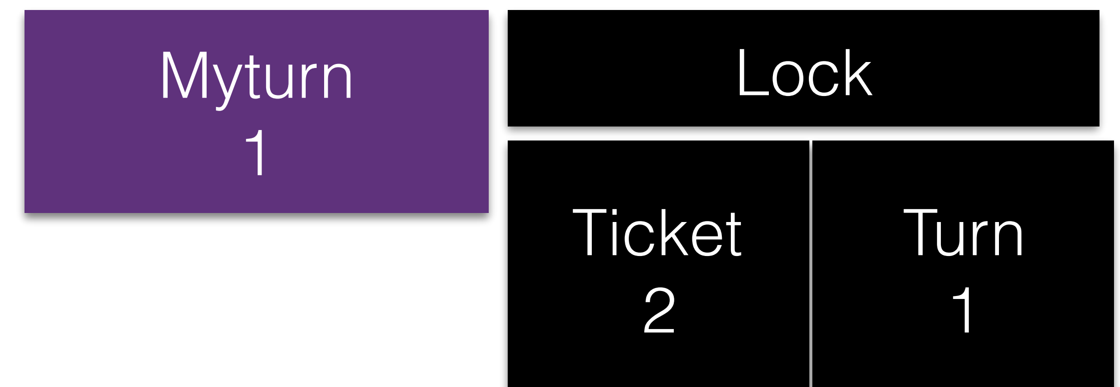
Thread T2 can run now

Lock	
Ticket 2	Turn 1

Fairness - Ticket Lock

```
1 typedef struct __lock_t {
2   int ticket;
3   int turn;
4 } lock_t;
5
6 void lock_init(lock_t *lock) {
7   lock->ticket = 0;
8   lock->turn = 0;
9 }
10
11 void lock(lock_t *lock) {
12   int myturn = FetchAndAdd(&lock->ticket);
13   while (lock->turn != myturn)
14   ; // spin
15 }
16 void unlock(lock_t *lock) {
17   FetchAndAdd(&lock->turn);
18 }
```

Thread T2 can run now



Ticket Lock Evaluation

Ticket Lock Evaluation

- Mutual exclusion: Yes

Ticket Lock Evaluation

- Mutual exclusion: Yes
- Fairness: Yes

Ticket Lock Evaluation

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

Ticket Lock Evaluation

- Mutual exclusion: Yes
- Fairness: Yes
- Performance: Spin Waiting is bad!
  - Can we eliminate it?!

Avoid Spinning - Yield!

```
1 void init() {
2     flag = 0;
3 }
4
5 void lock() {
6     while (TestAndSet(&flag, 1) == 1)
7         yield(); // give up the CPU
8 }
9
10 void unlock() {
11     flag = 0;
12 }
```

Avoid Spinning - Yield!

```
1 void init() {
2     flag = 0;
3 }
4
5 void lock() {
6     while (TestAndSet(&flag, 1) == 1)
7         yield(); // give up the CPU
8 }
9
10 void unlock() {
11     flag = 0;
12 }
```

- Give the CPU instead of spinning

Avoid Spinning - Yield!

```
1 void init() {
2     flag = 0;
3 }
4
5 void lock() {
6     while (TestAndSet(&flag, 1) == 1)
7         yield(); // give up the CPU
8 }
9
10 void unlock() {
11     flag = 0;
12 }
```

- Give the CPU instead of spinning
- Thread goes to ready state

Avoid Spinning - Yield!

```
1 void init() {
2     flag = 0;
3 }
4
5 void lock() {
6     while (TestAndSet(&flag, 1) == 1)
7         yield(); // give up the CPU
8 }
9
10 void unlock() {
11     flag = 0;
12 }
```

- Give the CPU instead of spinning
- Thread goes to ready state
- Still inefficient - Think 1000 threads, each in round robin

Avoid Spinning - Yield!

```
1 void init() {
2     flag = 0;
3 }
4
5 void lock() {
6     while (TestAndSet(&flag, 1) == 1)
7         yield(); // give up the CPU
8 }
9
10 void unlock() {
11     flag = 0;
12 }
```

- Give the CPU instead of spinning
- Thread goes to ready state
- Still inefficient - Think 1000 threads, each in round robin
  - checks lock; yields; heavy cost of context switch!

Avoid Spinning - Yield!

```
1 void init() {
2     flag = 0;
3 }
4
5 void lock() {
6     while (TestAndSet(&flag, 1) == 1)
7         yield(); // give up the CPU
8 }
9
10 void unlock() {
11     flag = 0;
12 }
```

- Give the CPU instead of spinning
- Thread goes to ready state
- Still inefficient - Think 1000 threads, each in round robin
  - checks lock; yields; heavy cost of context switch!
  - Thread can get starved! Yes, it's left to probability!

Queues for Fairness

Queues for Fairness

- Exert control over which thread to run next

Queues for Fairness

- Exert control over which thread to run next
  - Use a Queue

Queues for Fairness

- Exert control over which thread to run next
  - Use a Queue
- Combine spin-waiting + yielding

Queues for Fairness

- Exert control over which thread to run next
  - Use a Queue
- Combine spin-waiting + yielding
- park()

Queues for Fairness

- Exert control over which thread to run next
  - Use a Queue
- Combine spin-waiting + yielding
- park()
  - Put a thread to sleep

Queues for Fairness

- Exert control over which thread to run next
  - Use a Queue
- Combine spin-waiting + yielding
- park()
  - Put a thread to sleep
- unpark(t_id)

Queues for Fairness

- Exert control over which thread to run next
  - Use a Queue
- Combine spin-waiting + yielding
- park()
  - Put a thread to sleep
- unpark(t_id)
  - Wake up t_id thread

Queues for Fairness

```
1 typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
2
3 void lock_init(lock_t *m) {
4     m->flag = 0;
5     m->guard = 0;
6     queue_init(m->q);
7 }
8
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
11         ; // acquire guard lock by spinning
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }
```

Queues for Fairness

```
1 typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
2
3 void lock_init(lock_t *m) {
4     m->flag = 0;
5     m->guard = 0;
6     queue_init(m->q);
7 }
8
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
11        ; // acquire guard lock by spinning
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }
```

Two variables instead of one

Queues for Fairness

```
1 typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
2
3 void lock_init(lock_t *m) {
4     m->flag = 0;
5     m->guard = 0;
6     queue_init(m->q);
7 }
8
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
11        ; // acquire guard lock by spinning
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }
```

Spin waiting for guard lock

Queues for Fairness

```
1 typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
2
3 void lock_init(lock_t *m) {
4     m->flag = 0;
5     m->guard = 0;
6     queue_init(m->q);
7 }
8
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
11         ; // acquire guard lock by spinning
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }
```

Setting the main lock

Queues for Fairness

```
1 typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
2
3 void lock_init(lock_t *m) {
4     m->flag = 0;
5     m->guard = 0;
6     queue_init(m->q);
7 }
8
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
11         ; // acquire guard lock by spinning
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }
```

Add to Queue if can not set main lock

Queues for Fairness

```
1 typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
2
3 void lock_init(lock_t *m) {
4     m->flag = 0;
5     m->guard = 0;
6     queue_init(m->q);
7 }
8
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
11         ; // acquire guard lock by spinning
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }
```

Pop Quiz
How much time is spent in spin-waiting?

Queues for Fairness

```
1 typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
2
3 void lock_init(lock_t *m) {
4     m->flag = 0;
5     m->guard = 0;
6     queue_init(m->q);
7 }
8
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
11         ; // acquire guard lock by spinning
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }
```

Pop Quiz
How much time is spent in spin-waiting?

Not much!

Queues for Fairness

```
22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
24         ; // acquire guard lock by spinning
25     if (queue_empty(m->q))
26         m->flag = 0; // let go of lock; no one wants it
27     else
28         unpark(queue_remove(m->q)); // hold lock (for next thread!)
29     m->guard = 0;
30 }
```

Queues for Fairness

```
22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
24         ; // acquire guard lock by spinning
25     if (queue_empty(m->q))
26         m->flag = 0; // let go of lock; no one wants it
27     else
28         unpark(queue_remove(m->q)); // hold lock (for next thread!)
29     m->guard = 0;
30 }
```

Spin waiting for guard lock

Queues for Fairness

```
22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
24         ; // acquire guard lock by spinning
25     if (queue_empty(m->q))
26         m->flag = 0; // let go of lock; no one wants it
27     else
28         unpark(queue_remove(m->q)); // hold lock (for next thread!)
29     m->guard = 0;
30 }
```

Only two condition possible

Queue based lock - Worked Out Example

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m-
>guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is
acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m-
>guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m-
>q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m-
>guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is
acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m-
>guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m-
>q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m-
>guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is
acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m-
>guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m-
>q));
29     m->guard = 0;
30 }
```

Thread T1 wants to enter critical section & acquires guard lock

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

Thread T1 sets the flag — it now holds the lock!

T1

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

Thread T2 is brought into context and tries to execute critical section

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

Thread T2 spin waits ...

T1

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

Thread T1 is context switched back again and unsets the guard flag

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

Thread T3 is context switched in and wants to execute critical section

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

Test & Set on guard immediately returns since guard was 0

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

But, flag is still set by T1

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3 added to Queue

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2 is context switched in and wants to run critical section

T1

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

T2 spin waits … (since guard is set by T3!)

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3 unsets guard and parks …

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2 is context switched back in and acquires guard lock (finally :))

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2 added to queue since flag held by T1
Queue looks: [T2, T3]

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

(Lines 16–18 are bold in T2.)

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T1 executes critical section

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

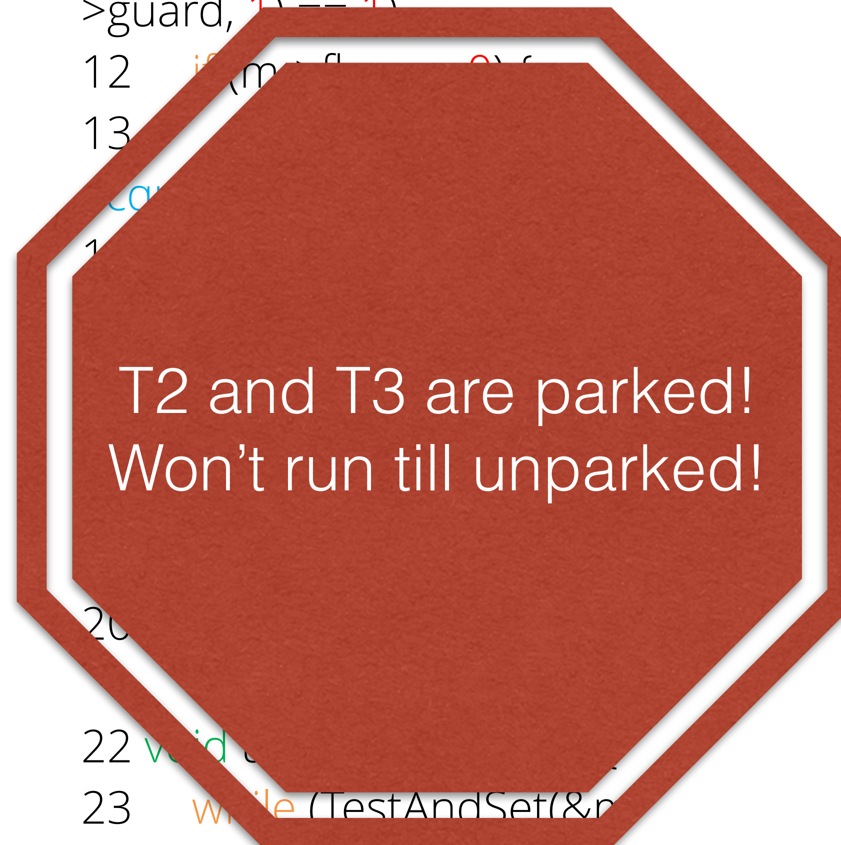
T2 is context switched back in

T1

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m-
>guard,
12
13

20

22
23
>guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```



T3

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

T1 completes critical section and proceeds to unlock; acquires guard lock

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

Unparks head of queue (T3); Now T3 can be scheduled …

T1

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

(Line 28 is in bold.)

T2

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

T1 unsets guard …

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3 now wants to enter critical section …

T1

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

Wakeup-Waiting Race Condition

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

Wakeup-Waiting Race Condition
T1 acquires guard and flag, and then unsets guard

T1

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

Wakeup-Waiting Race Condition
T2 acquires guard; tries to add itself to queue and unsets guard

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

(Lines 16 and 17 are shown in bold.)

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

Wakeup-Waiting Race Condition
T1 is context switched back in, runs critical section and unparks T2

T1

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

(Line 28 in T1 is highlighted in bold.)

T2

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10    while (TestAndSet(&m->guard, 1) == 1)
12    if (m->flag == 0) {
13        m->flag = 1; // lock is acquired
14        m->guard = 0;
15    } else {
16        queue_add(m->q, gettid());
17        m->guard = 0;
18        park();
19    }
20 }

22 void unlock(lock_t *m) {
23    while (TestAndSet(&m->guard, 1) == 1)
25    if (queue_empty(m->q))
26        m->flag = 0;
      else
28        unpark(queue_remove(m->q));
29    m->guard = 0;
30 }
```

Wakeup-Waiting Race Condition
T2 is unparked; and context switched back in and now **parks**

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m-
>guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is
acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m-
>guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m-
>q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m-
>guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is
acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m-
>guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m-
>q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m-
>guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is
acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m-
>guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m-
>q));
29     m->guard = 0;
30 }
```

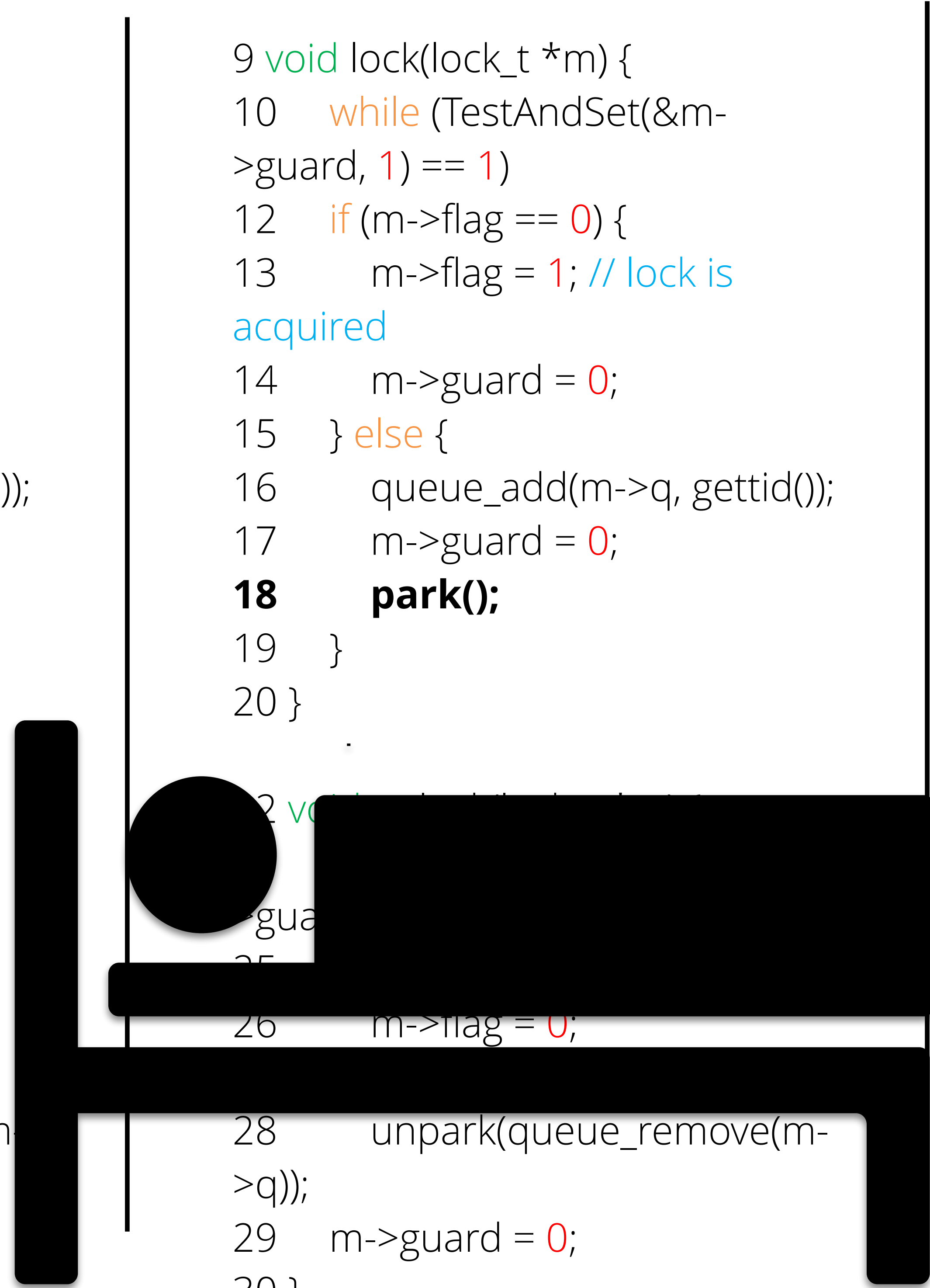
Wakeup-Waiting Race Condition
T2 can now potentially sleep forever …

T1

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T2

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }
```



```
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

T3

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();
19     }
20 }

22 void unlock(lock_t *m) {
23     while (TestAndSet(&m->guard, 1) == 1)
25     if (queue_empty(m->q))
26         m->flag = 0;
       else
28         unpark(queue_remove(m->q));
29     m->guard = 0;
30 }
```

Avoiding Race Condition

```
9 void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
11         ; // acquire guard lock by spinning
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17           setpark()
18         m->guard = 0;
19         park();
20     }
21 }
```

Later Study (After covering more ground for labs..)

Later Study (After covering more ground for labs..)

- Some important topics …

Later Study (After covering more ground for labs..)

- Some important topics …
  - Two phase locks

Later Study (After covering more ground for labs..)

- Some important topics …
  - Two phase locks
  - Futex

Later Study (After covering more ground for labs..)

- Some important topics …
  - Two phase locks
  - Futex
  - Priority Inversion

Lock-based concurrent data structures

Lock-based concurrent data structures

- Goal: Add locks to a data structure to make it thread safe

Lock-based concurrent data structures

- Goal: Add locks to a data structure to make it thread safe
  - Correctness

Lock-based concurrent data structures

- Goal: Add locks to a data structure to make it thread safe
  - Correctness
  - Performance

Non-threaded counter

```
1  typedef struct __counter_t {
2   int value;
3   } counter_t;
4
5   void init(counter_t *c) {
6   c->value = 0;
7   }
8
9   void increment(counter_t *c) {
10   c->value++;
11   }
12
13   void decrement(counter_t *c) {
14   c->value--;
15   }
16
17   int get(counter_t *c) {
18   return c->value;
19   }
```

Concurrent counter

```
1  typedef struct __counter_t {
2  int value;
3  pthread_lock_t lock;
4  } counter_t;
5
6  void init(counter_t *c) {
7  c->value = 0;
8  Pthread_mutex_init(&c->lock, NULL);
9  }
10
11  void increment(counter_t *c) {
12  Pthread_mutex_lock(&c->lock);
13  c->value++;
14  Pthread_mutex_unlock(&c->lock);
15  }
16
```

Concurrent counter

```
1  typedef struct __counter_t {
2  int value;
3  pthread_lock_t lock;
4  } counter_t;
5
6  void init(counter_t *c) {
7  c->value = 0;
8  Pthread_mutex_init(&c->lock, NULL);
9  }
10
11  void increment(counter_t *c) {
12  Pthread_mutex_lock(&c->lock);
13  c->value++;
14  Pthread_mutex_unlock(&c->lock);
15  }
16
```

Initialization

Concurrent counter

```
1   typedef struct __counter_t {
2   int value;
3   pthread_lock_t lock;
4   } counter_t;
5
6   void init(counter_t *c) {
7   c->value = 0;
8   Pthread_mutex_init(&c->lock, NULL);
9   }
10
11  void increment(counter_t *c) {
12  Pthread_mutex_lock(&c->lock);
13  c->value++;
14  Pthread_mutex_unlock(&c->lock);
15  }
16
```

Lock, Modify, Unlock

Concurrent counter

```
17        void decrement(counter_t *c) {
18                Pthread_mutex_lock(&c->lock);
19                c->value--;
20                Pthread_mutex_unlock(&c->lock);
21        }
22
23        int get(counter_t *c) {
24                Pthread_mutex_lock(&c->lock);
25                int rc = c->value;
26                Pthread_mutex_unlock(&c->lock);
27                return rc;
28        }
```

Lock, Modify, Unlock

Lock, Modify, Unlock

Condition Variables