

Distributed Operating Systems

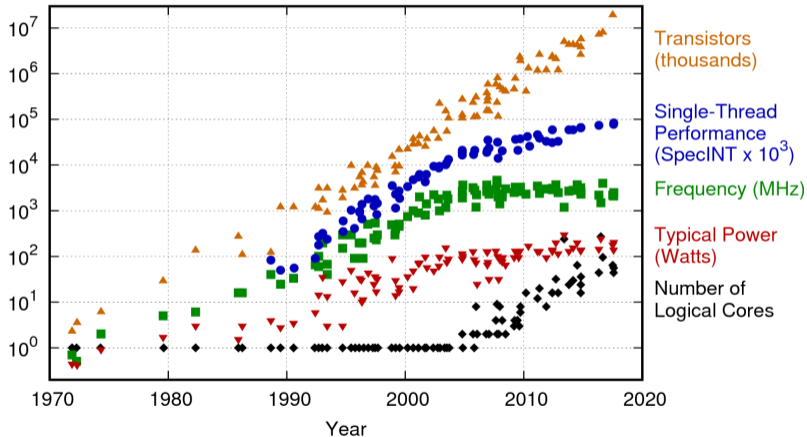
Synchronization in Parallel Systems

TILL SMEJKAL

June 3rd, 2024

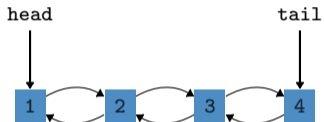
Why do we need synchronization?

42 Years of Microprocessor Trend Data

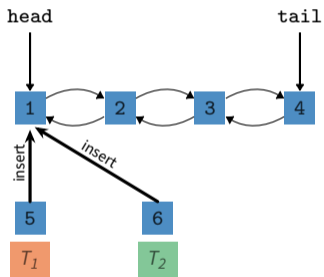


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

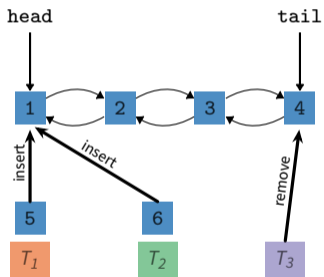
Why do we need synchronization?



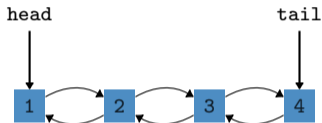
Why do we need synchronization?



Why do we need synchronization?

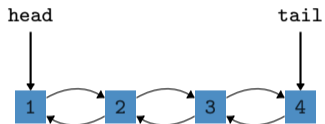


Why do we need synchronization?



```
1 struct ele_t *new_ele = new ele_t;  
2 new_ele->next = head;  
3 head->prev = new_ele;  
4 head = new_ele;
```

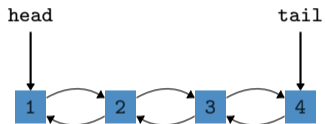
Why do we need synchronization?

 T_1 T_2

```
1 struct ele_t *new_ele = new ele_t;  
2 new_ele->next = head;  
3 head->prev = new_ele;  
4 head = new_ele;
```

 T_1 T_2

Why do we need synchronization?

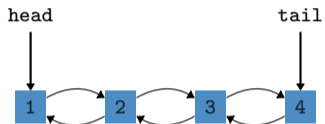

 T_1
 T_2
 $T_1 \quad T_2$

```

1. 1 struct ele_t *new_ele = new ele_t;
   2 new_ele->next = head;
   3 head->prev = new_ele;
   4 head = new_ele;

```


Why do we need synchronization?

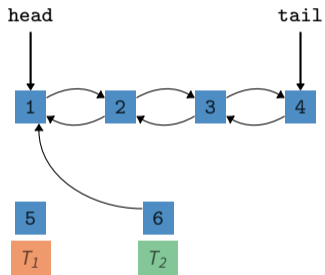

 T_1 T_2

```

1. 2. 1 struct ele_t *new_ele = new ele_t;
2 new_ele->next = head;
3 head->prev = new_ele;
4 head = new_ele;

```

Why do we need synchronization?

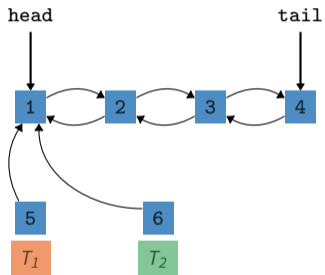

 T_1 T_2

```

1. 2. 1 struct ele_t *new_ele = new ele_t; ←
3. 2 new_ele->next = head; ←
4 head->prev = new_ele;
4 head = new_ele;

```

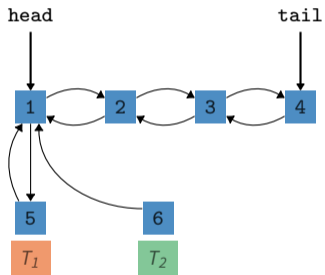
Why do we need synchronization?


 T_1 T_2

```

1. 2. 1 struct ele_t *new_ele = new ele_t;
4. 3. 2 new_ele->next = head; ←
      3 head->prev = new_ele;
      4 head = new_ele;
  
```

Why do we need synchronization?

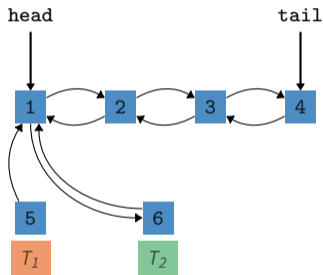

 T_1 T_2

```

1. 2. 1 struct ele_t *new_ele = new ele_t;
4. 3. 2 new_ele->next = head; ←
5.    3 head->prev = new_ele; ←
      4 head = new_ele;

```

Why do we need synchronization?

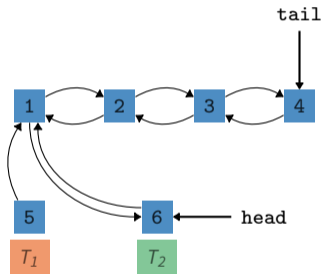

 T_1 T_2

```

1. 2. 1 struct ele_t *new_ele = new ele_t;
4. 3. 2 new_ele->next = head;
5. 6. 3 head->prev = new_ele; ←
4 head = new_ele;

```

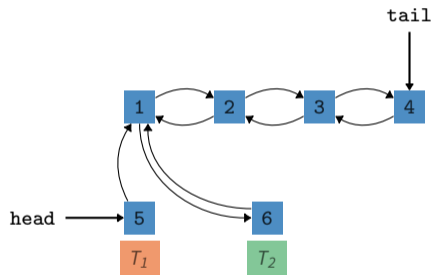
Why do we need synchronization?


 T_1 T_2

```

1. 2. 1 struct ele_t *new_ele = new ele_t;
4. 3. 2 new_ele->next = head;
5. 6. 3 head->prev = new_ele; ←
7. 4 head = new_ele; ←
  
```

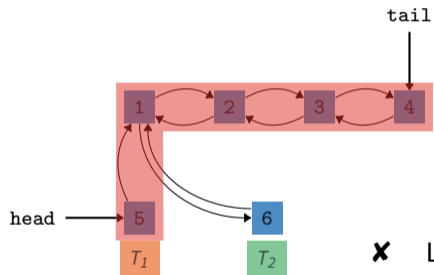
Why do we need synchronization?


 T_1 T_2

```

1. 2. 1 struct ele_t *new_ele = new ele_t;
4. 3. 2 new_ele->next = head;
5. 6. 3 head->prev = new_ele;
8. 7. 4 head = new_ele; ←
  
```

Why do we need synchronization?


 T_1 T_2

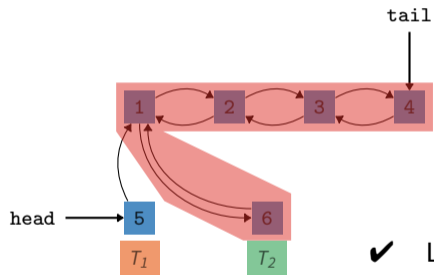
```

1. 2. 1 struct ele_t *new_ele = new ele_t;
4. 3. 2 new_ele->next = head;
5. 6. 3 head->prev = new_ele;
8. 7. 4 head = new_ele;

```

- ✗ List structure correct
- ✓ head points to start of list

Why do we need synchronization?


 T_1 T_2

```

1. 2. 1 struct ele_t *new_ele = new ele_t;
4. 3. 2 new_ele->next = head;
5. 6. 3 head->prev = new_ele;
8. 7. 4 head = new_ele;

```

- ✓ List structure correct
- ✗ head points to start of list

Content

Basic Principles

Implementing Entersection & Leavesection

Atomicity on Hardware

- Cache Lock
- Observe Cache
- Atomic Instructions

Synchronization with Locks – Part I

- Test & Set Lock
- Test & Test & Set Lock
- Ticket Lock

Synchronization without Locks

Synchronization with Locks – Part II

- MCS Lock
- Reader Writer Lock

Special Issues

- Timeouts / Aborting Locks
- Lockholder Preemption

Basic Principles

Atomicity Assumption

$$A \parallel B = A;B \vee B;A$$

Basic Principles

Atomicity Assumption

Parallel execu-
tion of A and B

$A \parallel B = A;B \vee B;A$

Sequentiell execu-
tion, first A then B

Sequentiell execu-
tion, first B then A

Basic Principles

Atomicity Assumption

Parallel execu-
tion of A and B

$A \parallel B = A;B \vee B;A$

Sequentiell execu-
tion, first A then B

Sequentiell execu-
tion, first B then A

- Otherwise, the outcome of $A \parallel B$ is undefined

Basic Principles

Atomicity Assumption

Parallel execu-
tion of A and B

$A \parallel B = A;B \vee B;A$

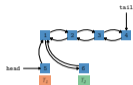
Sequentiell execu-
tion, first A then B

Sequentiell execu-
tion, first B then A

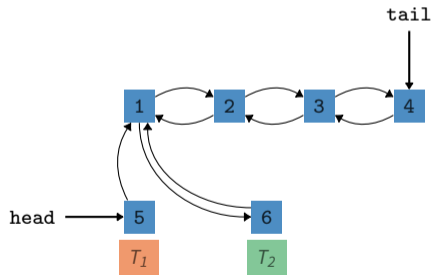
- Otherwise, the outcome of $A \parallel B$ is undefined
- Usually problematic for parallel *Read-Modify-Write* operations

Basic Principles

Mutual Exclusion



A || B



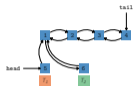
```

1 struct ele_t *new_ele = new ele_t;
2 new_ele->next = head;
3 head->prev = new_ele;
4 head = new_ele;

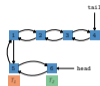
```

Basic Principles

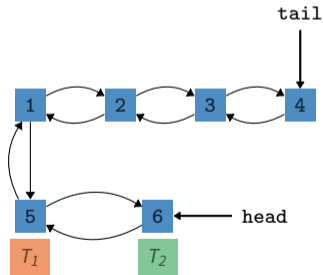
Mutual Exclusion



A || B



A;B



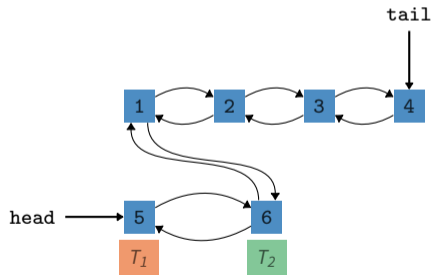
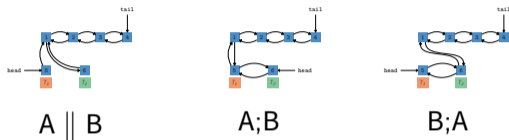
```

1 struct ele_t *new_ele = new ele_t;
2 new_ele->next = head;
3 head->prev = new_ele;
4 head = new_ele;

```


Basic Principles

Mutual Exclusion



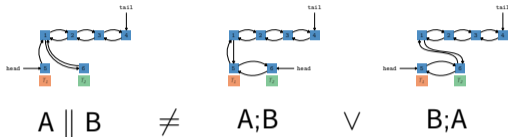
```

1 struct ele_t *new_ele = new ele_t;
2 new_ele->next = head;
3 head->prev = new_ele;
4 head = new_ele;

```

Basic Principles

Mutual Exclusion



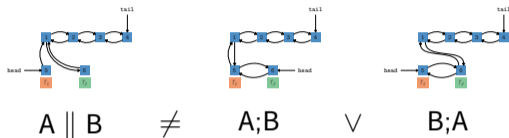
```

1 struct ele_t *new_ele = new ele_t;
2 new_ele->next = head;
3 head->prev = new_ele;
4 head = new_ele;

```

Basic Principles

Mutual Exclusion



Need to ensure that only one thread at a time can execute the *Read-Modify-Write* operation.

⇒ **Mutual Exclusion**

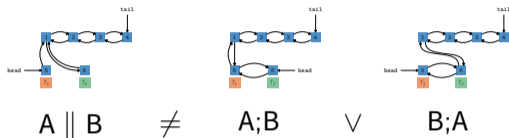
```

1 struct ele_t *new_ele = new ele_t;
2 new_ele->next = head;
3 head->prev = new_ele;
4 head = new_ele;

```

Basic Principles

Mutual Exclusion



Need to ensure that only one thread at a time can execute the *Read-Modify-Write* operation.

```

1 struct ele_t *new_ele = new ele_t;
2 new_ele->next = head;
3 head->prev = new_ele;
4 head = new_ele;

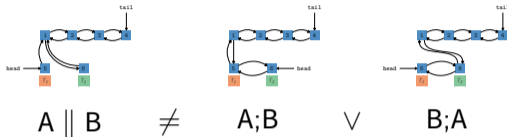
```

} CS

⇒ **Mutual Exclusion**

Basic Principles

Mutual Exclusion



Need to ensure that only one thread at a time can execute the *Read-Modify-Write* operation.

⇒ **Mutual Exclusion**

```

1 struct ele_t *new_ele = new ele_t;
2 new_ele->next = head;
3 head->prev = new_ele;
4 head = new_ele;

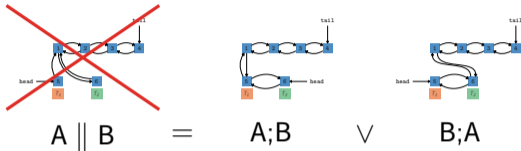
```

} CS

Should run with mutual exclusion!

Basic Principles

Mutual Exclusion



Need to ensure that only one thread at a time can execute the *Read-Modify-Write* operation.

⇒ **Mutual Exclusion**

```

1 struct ele_t *new_ele = new ele_t;
2 new_ele->next = head;
3 head->prev = new_ele;
4 head = new_ele;

```

} CS

Should run with mutual exclusion!

Basic Principles

Entersection & Leavesection

Simple protocol to establish mutual exclusion for critical sections.

```
1 struct ele_t *new_ele = new ele_t;
2 entersection();
3 new_ele->next = head;
4 head->prev = new_ele;
5 head = new_ele;
6 leavesection();
```

} CS

```
1 void entersection() {
2     while (!cs_free) wait();
3     cs_free = false;
4 }
```

```
1 void leavesection() {
2     cs_free = true;
3     wake_next();
4 }
```

Basic Principles

Entersection & Leavesection

Simple protocol to establish mutual exclusion for critical sections.

```
1 struct ele_t *new_ele = new ele_t;
2 entersection();
3 new_ele->next = head;
4 head->prev = new_ele;
5 head = new_ele;
6 leavesection();
```

} CS

```
1 void entersection() {
2     while (!cs_free) wait();
3     cs_free = false;
4 }
```

```
1 void leavesection() {
2     cs_free = true;
3     wake_next();
4 }
```

Rules:

- entersection and leavesection *must always* exist in pairs
- entersection *must always* be before the corresponding leavesection

Basic Principles

Entersection & Leavesection

Simple protocol to establish mutual exclusion for critical sections.

```
1 struct ele_t *new_ele = new ele_t;
2 lock();
3 new_ele->next = head;
4 head->prev = new_ele;
5 head = new_ele;
6 unlock();
```

} CS

```
1 void lock() {
2     while (!cs_free) wait();
3     cs_free = false;
4 }
```

```
1 void unlock() {
2     cs_free = true;
3     wake_next();
4 }
```

Rules:

- entersection and leavesection *must always* exist in pairs
- entersection *must always* be before the corresponding leavesection

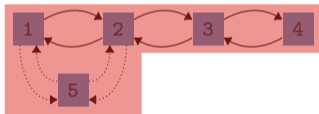
Basic Principles

Coarse Grained vs. Fine Grained

Critical Sections should be as long as necessary but also as short as possible.

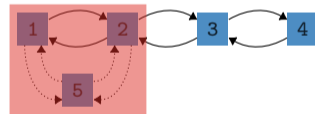
- Length of critical sections are important for scalability → Amdahl's Law

Coarse Grained



- Worse scalability (no parallel operations)
- + Easier to implement

Fine Grained



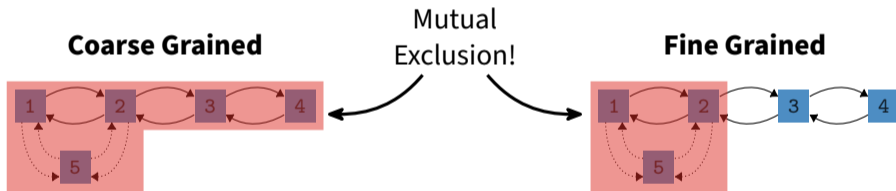
- + Better scalability (parallel operations possible)
- More difficult to implement
- Deadlocks may happen

Basic Principles

Coarse Grained vs. Fine Grained

Critical Sections should be as long as necessary but also as short as possible.

- Length of critical sections are important for scalability → Amdahl's Law



- Worse scalability (no parallel operations)
- + Easier to implement

- + Better scalability (parallel operations possible)
- More difficult to implement
- Deadlocks may happen

Implementing Entersection & Leavesection

Peterson Algorithm

```
1 bool free[2] = {true, true};
2 int turn = 0;
3
4 void lock() {
5     int other = 1 - TID;          /* TID: ID of the current thread ({0,1}) */
6     free[TID] = false;
7     turn = other;
8     while (!free[other] && turn == other) {}
9 }
10
11 void unlock() {
12     free[TID] = true;
13 }
```

- Works for two threads (more threads are possible but it gets complicated)
- Requires atomic load and stores and sequential consistency (or additional fence)

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() {
4     while (l == 1) {}
5     l = 1;
6 }
7
8 void unlock() {
9     l = 0;
10 }
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() {
4     while (l == 1) {}
5     l = 1;
6 }
7
8 void unlock() {
9     l = 0;
10 }
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() {
4     while (l == 1) {}
5     l = 1;
6 }
7
8 void unlock() {
9     l = 0;
10 }
```

```
1 /* do other stuff */ ←
2 lock();
3 /* critical section */
4 unlock();
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() {
4     while (l == 1) {}
5     l = 1;
6 }
7
8 void unlock() {
9     l = 0;
10 }
```

```
1 /* do other stuff */ ←
2 lock(); ←
3 /* critical section */
4 unlock();
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() { ←
4     while (l == 1) {}
5     l = 1;
6 }
7
8 void unlock() {
9     l = 0;
10 }
```

```
1 /* do other stuff */ ←
2 lock();
3 /* critical section */
4 unlock();
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() {
4     while (l == 1) {} ←
5     l = 1;
6 }
7
8 void unlock() {
9     l = 0;
10 }
```

```
1 /* do other stuff */ ←
2 lock();
3 /* critical section */
4 unlock();
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() {
4     while (l == 1) {}
5     l = 1;
6 }
7
8 void unlock() {
9     l = 0;
10 }
```

Diagram illustrating the lock acquisition logic:

- Line 4: `while (l == 1) {}` with an arrow pointing to the condition `l == 1`.
- Line 2: `l == 0` with an arrow pointing down to the condition `l == 1`.
- Line 4: An orange arrow points to the closing brace of the `while` loop.

```
1 /* do other stuff */ ←
2 lock();
3 /* critical section */
4 unlock();
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() {
4     while (l == 1) {}
5     l = 1;
6 }
7
8 void unlock() {
9     l = 0;
10 }
```

Diagram illustrating the lock acquisition logic:

- Line 4: `while (l == 1) {}` with an arrow pointing to the condition `l == 1`.
- Line 2: `l == 0` with an arrow pointing down to the condition `l == 1`.
- Line 4: `while (l == 1) {}` with an arrow pointing to the closing brace `}`.

```
1 /* do other stuff */
2 lock();
3 /* critical section */
4 unlock();
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```

1 int l = 0;
2
3 void lock() { ←
4     while (l == 1) {} ←
5     l = 1;
6 }
7
8 void unlock() {
9     l = 0;
10 }

```

Diagram illustrating the naive spinlock implementation. The code shows a lock variable `l` and two functions: `lock()` and `unlock()`. The `lock()` function enters a `while` loop that spins until `l` is 0. An arrow points to the condition `l == 1` in the `while` loop, and another arrow points to the `while` loop body, indicating the spinning behavior.

```

1 /* do other stuff */
2 lock();
3 /* critical section */
4 unlock();

```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() {
4     while (l == 1) {}
5     l = 1;
6 }
7
8 void unlock() {
9     l = 0;
10 }
```

Diagram illustrating a naive spinlock implementation. The code shows a lock variable `l` initialized to 0. The `lock()` function enters a `while (l == 1) {}` loop. A diagram shows the condition `l == 1` with a downward arrow pointing to the loop body, indicating that the thread is spinning (waiting) because the lock is held.

```
1 /* do other stuff */
2 lock();
3 /* critical section */
4 unlock();
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
```

```
2
```

```
3 void lock() {
```

```
4     while (l == 1) {}
```

```
5     l = 1;
```

```
6 }
```

```
7
```

```
8 void unlock() {
```

```
9     l = 0;
```

```
10 }
```

```
1 == 0
```



```
1 == 0
```

```
1 /* do other stuff */
```

```
2 lock();
```

```
3 /* critical section */
```

```
4 unlock();
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() {
4     while (l == 1) {}
5     l = 1;
6 }
7
8 void unlock() {
9     l = 0;
10 }
```

Diagram illustrating the lock acquisition logic:

- Line 2: `1 == 0` with a downward arrow pointing to the condition in line 4.
- Line 4: `while (l == 1) {}` with an orange arrow pointing to the condition.
- Line 5: `l = 1;` with a green arrow pointing to the assignment.

```
1 /* do other stuff */
2 lock();
3 /* critical section */
4 unlock();
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() {
4     while (l == 1) {}
5     l = 1; ←
6 }
7
8 void unlock() {
9     l = 0;
10 }
```

```
1 /* do other stuff */
2 lock();
3 /* critical section */
4 unlock();
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() {
4     while (l == 1) {}
5     l = 1; ←
6 } ←
7
8 void unlock() {
9     l = 0;
10 }
```

```
1 /* do other stuff */
2 lock();
3 /* critical section */
4 unlock();
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() {
4     while (l == 1) {}
5     l = 1; ←
6 }
7
8 void unlock() {
9     l = 0;
10 }
```

```
1 /* do other stuff */
2 lock();
3 /* critical section */ ←
4 unlock();
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() {
4     while (l == 1) {}
5     l = 1;
6 } ←
7
8 void unlock() {
9     l = 0;
10 }
```

```
1 /* do other stuff */
2 lock();
3 /* critical section */ ←
4 unlock();
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() {
4     while (l == 1) {}
5     l = 1;
6 }
7
8 void unlock() {
9     l = 0;
10 }
```

```
1 /* do other stuff */
2 lock();
3 /* critical section */ ←
4 unlock();
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() {
4     while (l == 1) {}
5     l = 1;
6 }
7
8 void unlock() {
9     l = 0;
10 }
```

```
1 /* do other stuff */
2 lock();
3 /* critical section */ ← ⚡
4 unlock();
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Implementing Entersection & Leavesection

Naive Spinlock (*Wrong!*)

```
1 int l = 0;
2
3 void lock() {
4     while (l == 1) {} } CS (internal)
5     l = 1;
6 }
7
8 void unlock() {
9     l = 0;
10 }
```

```
1 /* do other stuff */
2 lock();
3 /* critical section */ ← ⚡
4 unlock();
```

- Works for any number of threads
- Simple approach which can work on any hardware architecture
- Requires solving internal critical section → hardware support

Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \vee B;A$$

- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \vee B;A$$

- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

```
1 cmp [x] $0;  
2 jne retry;
```

Core 2

```
1 mov $1 [x];
```

Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \vee B;A$$

- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

```
1 load [x] %eax;  
2 cmp %eax $0;  
3 jne retry;
```

Core 2

```
1 store $1 [x];
```

Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \vee B;A$$

- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

```
1 load [x] %eax;  
2 cmp %eax $0;  
3 jne retry;
```

Core 2

```
1 store $1 [x];
```

Core 1

Memory

x:0

Core 2

Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \vee B;A$$

- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

```

1 load [x] %eax; ←
2 cmp %eax $0;
3 jne retry;

```

Core 2

```

1 store $1 [x];

```

Core 1

eax:0

Memory

x:0

Load(x)

Core 2

Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \vee B;A$$

- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

```

1 load [x] %eax; ←
2 cmp %eax $0;
3 jne retry;

```

Core 2

```

1 store $1 [x]; ←

```

Core 1

eax:0

Memory

x:1

Core 2

Store(x)

Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \vee B;A$$

- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

```

1 load [x] %eax;
2 cmp %eax $0;
3 jne retry;

```

Core 2

```

1 store $1 [x];

```

Core 1

eax:0

Memory

x:1

Core 2

Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \vee B;A$$

- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

```

1 load [x] %eax;
2 cmp %eax $0;
3 jne retry;
  
```

Comparison with 0
although $x == 1$ already

Core 2

```
1 store $1 [x];
```

Core 1

eax:0

Memory

x:1

Core 2

Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \vee B;A$$

- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

```
1 add [x] $1;
```

Core 2

```
1 mov $2 [x];
```


Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \vee B;A$$

- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

```
1 load [x] %eax;  
2 add %eax $1;  
3 store %eax [x];
```

Core 2

```
1 store $2 [x];
```

Core 1

Memory

x:0

Core 2

Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \vee B;A$$

- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

```

1 load [x] %eax; ←
2 add %eax $1;
3 store %eax [x];

```

Core 2

```

1 store $2 [x];

```

Core 1

eax:0

Memory

x:0

← Load(x)

Core 2

Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \vee B;A$$

- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

```

1 load [x] %eax; ←
2 add %eax $1;
3 store %eax [x];
  
```

Core 2

```

1 store $2 [x]; ←
  
```

Core 1

eax:0

Memory

x:2

← Store(x)

Core 2

Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \vee B;A$$

- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

```

1 load [x] %eax;
2 add %eax $1;
3 store %eax [x];

```

Core 2

```

1 store $2 [x];

```

Core 1

eax:1

Memory

x:2

Core 2

Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \vee B;A$$

- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

```
1 load [x] %eax;  
2 add %eax $1;  
3 store %eax [x];
```

Core 2

```
1 store $2 [x];
```

Core 1

eax:1

Memory

x:1

Store(x)

Core 2

Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \vee B;A$$

- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

```
1 load [x] %eax;
2 add %eax $1;
3 store %eax [x];
```

Core 2

```
1 store $2 [x];
```

x == 1 although
A;B: x == 2 or B;A: x == 3

Core 1

eax: 1

Memory

x: 1 ⚡

Core 2

Atomicity on Hardware

Atomic Hardware Instructions

How to make instructions atomic?

- Bus Lock
 - Lock whole memory bus until all memory accesses of instruction are completed
 - Used in older x86 CPUs (Intel® Pentium 3 and older)
 - Uses `lock` assembler attribute

Atomicity on Hardware

Atomic Hardware Instructions

How to make instructions atomic?

- Bus Lock
 - Lock whole memory bus until all memory accesses of instruction are completed
 - Used in older x86 CPUs (Intel® Pentium 3 and older)
 - Uses `lock` assembler attribute
- Cache Lock
 - Delay cache coherency traffic until all memory accesses of instruction are completed
 - Used in newer x86 CPUs (Intel® Pentium 4 and newer)
 - Special atomic instructions (e.g. `cmpxchg` or `xadd`)

Atomicity on Hardware

Atomic Hardware Instructions

How to make instructions atomic?

- Bus Lock
 - Lock whole memory bus until all memory accesses of instruction are completed
 - Used in older x86 CPUs (Intel® Pentium 3 and older)
 - Uses `lock` assembler attribute
- Cache Lock
 - Delay cache coherency traffic until all memory accesses of instruction are completed
 - Used in newer x86 CPUs (Intel® Pentium 4 and newer)
 - Special atomic instructions (e.g. `cmpxchg` or `xadd`)
- Observe Cache
 - Install watchdog on load and check at corresponding store if a concurrent access happened and abort
 - Used on arm and Alpha CPUs
 - Uses special `ldrex` and `strex` instructions

Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

```
1 add $1 [x];
```

Core 2

```
1 mov $2 [x];
```

Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

```
1 loadx [x] %eax;  
2 add $1 %eax;  
3 store %eax [x];
```

Core 2

```
1 store $2 [x];
```

Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

```
1 loadx [x] %eax;  
2 add $1 %eax;  
3 store %eax [x];
```

Core 2

```
1 store $2 [x];
```

Core 1

x:0 → S

Memory

x:0

Core 2

x:0 → S

Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

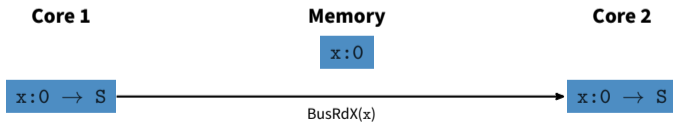
```

1 loadx [x] %eax; ←
2 add $1 %eax;
3 store %eax [x];
  
```

Core 2

```

1 store $2 [x];
  
```



Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

```

1 loadx [x] %eax; ←
2 add $1 %eax;
3 store %eax [x];

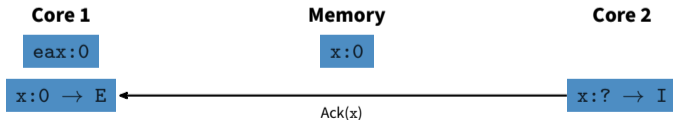
```

Core 2

```

1 store $2 [x];

```



Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

```

1 loadx [x] %eax;
2 add $1 %eax; ←
3 store %eax [x];

```

Core 2

```

1 store $2 [x];

```

Core 1

eax:1

x:0 → E

Memory

x:0

Core 2

x:? → I

Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

```
1 loadx [x] %eax;
2 add $1 %eax; ←
3 store %eax [x];
```

Core 2

```
1 store $2 [x]; ←
```

Core 1

eax:1

x:0 → E

Memory

x:0

Core 2

x:? → I

Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

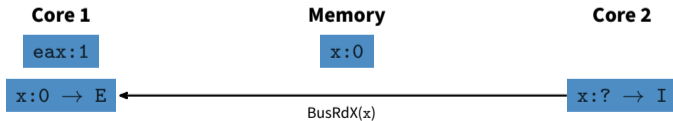
```

1 loadx [x] %eax;
2 add $1 %eax; ←
3 store %eax [x];
  
```

Core 2

```

1 store $2 [x]; ←
  
```



Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

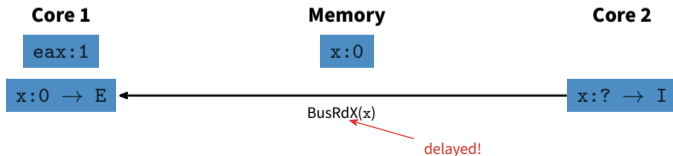
```

1 loadx [x] %eax;
2 add $1 %eax; ←
3 store %eax [x];
  
```

Core 2

```

1 store $2 [x]; ←
  
```



Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

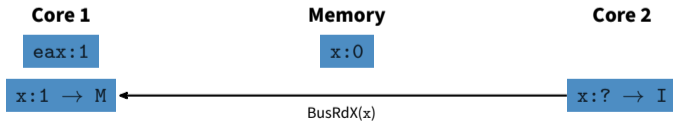
```

1 loadx [x] %eax;
2 add $1 %eax;
3 store %eax [x]; ←
  
```

Core 2

```

1 store $2 [x]; ←
  
```



Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

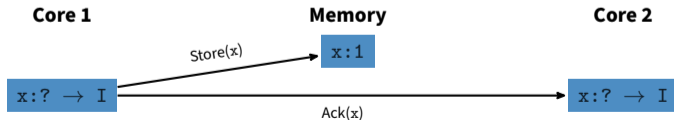
```

1 loadx [x] %eax;
2 add $1 %eax;
3 store %eax [x]; ←
  
```

Core 2

```

1 store $2 [x]; ←
  
```



Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

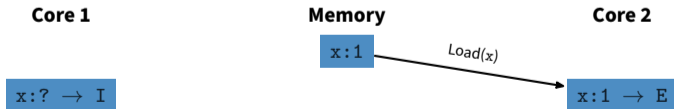
Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

```
1 loadx [x] %eax;
2 add $1 %eax;
3 store %eax [x];
```

Core 2

```
1 store $2 [x]; ←
```



Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

```
1 loadx [x] %eax;
2 add $1 %eax;
3 store %eax [x];
```

Core 2

```
1 store $2 [x]; ←
```

Core 1

x:? → I

Memory

x:1

Core 2

x:2 → M

Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

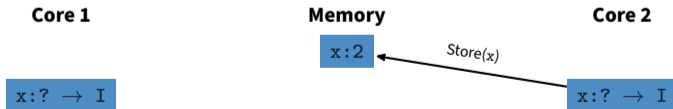
Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

```
1 loadx [x] %eax;  
2 add $1 %eax;  
3 store %eax [x];
```

Core 2

```
1 store $2 [x];
```



Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

```
1 add $1 [x];
```

Core 2

```
1 mov $2 [x];
```

Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

```
1 ldrex [x] %eax;  
2 add $1 %eax;  
3 strex %eax [x];
```

Core 2

```
1 store $2 [x];
```

Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

```
1 ldrex [x] %eax;  
2 add $1 %eax;  
3 strex %eax [x];
```

Core 2

```
1 store $2 [x];
```

Core 1

x:0 → S

Memory

x:0

Core 2

x:0 → S

Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

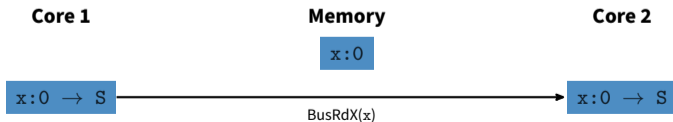
```

1 ldrex [x] %eax; ←
2 add $1 %eax;
3 strex %eax [x];
  
```

Core 2

```

1 store $2 [x];
  
```



Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

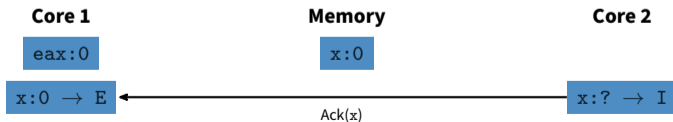
Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

```
1 ldrex [x] %eax; ←
2 add $1 %eax;
3 strex %eax [x];
```

Core 2

```
1 store $2 [x];
```



Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

```
1 ldrex [x] %eax;
2 add $1 %eax; ←
3 strex %eax [x];
```

Core 2

```
1 store $2 [x];
```

Core 1

eax:1

x:0 → E

Memory

x:0

Core 2

x:? → I

Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

```

1 ldrex [x] %eax;
2 add $1 %eax;
3 strex %eax [x]; ←
  
```

Core 2

```

1 store $2 [x];
  
```

Core 1

x:1 → M

Memory

x:0

Core 2

x:? → I

Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

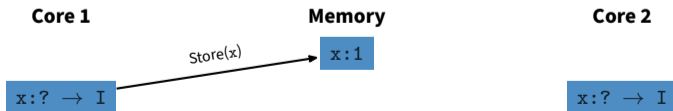
Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

```
1 ldrex [x] %eax;  
2 add $1 %eax;  
3 strex %eax [x]; ← ✓ Success
```

Core 2

```
1 store $2 [x];
```



Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

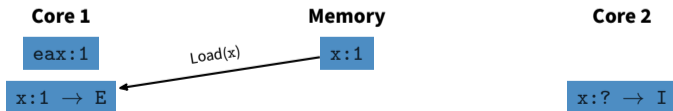
```

1 ldrex [x] %eax; ←
2 add $1 %eax;
3 strex %eax [x];
  
```

Core 2

```

1 store $2 [x];
  
```



Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

```
1 ldrex [x] %eax;  
2 add $1 %eax; ←  
3 strex %eax [x];
```

Core 1

eax:2

x:1 → E

Memory

x:1

Core 2

```
1 store $2 [x];
```

Core 2

x:? → I

Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

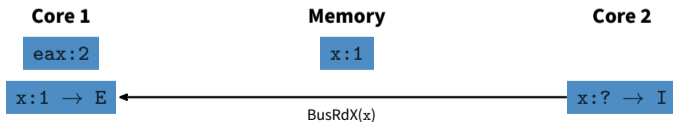
```

1 ldrex [x] %eax;
2 add $1 %eax; ←
3 strex %eax [x];
  
```

Core 2

```

1 store $2 [x]; ←
  
```



Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

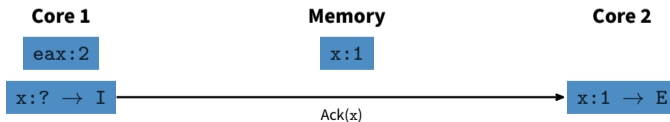
```

1 ldrex [x] %eax;
2 add $1 %eax; ←
3 strex %eax [x];
  
```

Core 2

```

1 store $2 [x]; ←
  
```



Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

```
1 ldrex [x] %eax;
2 add $1 %eax; ←
3 strex %eax [x];
```

Core 2

```
1 store $2 [x]; ←
```

Core 1

eax:2

x:? → I

Memory

x:1

Core 2

x:2 → M

Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

```

1 ldrex [x] %eax;
2 add $1 %eax; ←
3 strex %eax [x];
  
```

Core 1

eax:2

x:? → I

Core 2

```

1 store $2 [x];
  
```

Memory

x:2

Core 2

x:? → I

Store(x)

Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

```
1 ldrex [x] %eax;
2 add $1 %eax;
3 strex %eax [x]; ←
```

Core 2

```
1 store $2 [x];
```

Core 1

eax:2

x:? → I

Memory

x:2

Core 2

Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

```

1 ldrex [x] %eax;
2 add $1 %eax;
3 strex %eax [x]; ← ✖ Abort
  
```

Core 2

```
1 store $2 [x];
```

Core 1

eax:2

x:? → I

Memory

x:2

Core 2

Atomicity on Hardware

Examples of Atomic Instructions

- `swap(mem1, mem2)`

```
1 mov [mem1] %eax;  
2 mov [mem2] [mem1];  
3 mov %eax [mem2];
```

- `xadd(mem, reg)`

```
1 mov [mem] %eax;  
2 add [mem] reg;  
3 return %eax;
```

- `cas(mem, expected, desired)`

```
1 cmp [mem] [expected];  
2 jne fail;  
3 mov [desired] [mem];  
4 return true;  
5 fail: return false;
```

Synchronization with Locks

Properties

Main Properties

- Mutual Exclusion
 - Required by every *correct* implementation of the Entersection & Leavesection protocol
- Overhead
 - Acquiring a lock should be a cheap operation
 - If the lock is currently free, acquiring the lock should be especially cheap
- Fairness
 - Every thread should be able to acquire the lock eventually

Synchronization with Locks

Properties

Advanced Properties

- Concurrent access to critical section
 - Allow multiple threads to acquire the lock simultaneously
- Abort pending lock operations
 - Abort acquiring a currently taken lock after a timeout
 - Kill threads currently acquiring a lock
- Lock holder preemption
 - Prevent the threads currently holding the lock from making progress
- Priority inversion
 - Prevent higher priority threads from making progress because of a lower priority thread holding a shared lock
- Spinning vs. Blocking

Test & Set Lock

```
1 struct ts_lock_t {
2     volatile int lock;
3 };
4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1;
7         swap(&(l->lock), &(tmp));
8     } while (tmp == 1);
9 }
11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }
```

- Very easy to implement
- Only requires one atomic instruction

But

- High cache-coherency bus traffic when lock is taken
- No fairness between threads

Test & Set Lock

Overhead

```
1 struct ts_lock_t {
2     volatile int lock;
3 };
4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1;
7         swap(&(l->lock), &(tmp));
8     } while (tmp == 1);
9 }
11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }
```

```
1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */
5     lock(&l);
6     /* CS */
7     unlock(&l);
8 }
```

Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1;
7         swap(&(l->lock), &(tmp));
8     } while (tmp == 1);
9 }

11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l);
6     /* CS */
7     unlock(&l);
8 }

```

Core 1

l:? → I

Core 2

l:? → I

Core 3

l:? → I

Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1;
7         swap(&(l->lock), &(tmp));
8     } while (tmp == 1);
9 }

11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l); ←
6     /* CS */
7     unlock(&l);
8 }

```

Core 1

l:? → I

Core 2

l:? → I

Core 3

l:? → I

Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1; ←
7         swap(&(l->lock), &(tmp));
8     } while (tmp == 1);
9 }

11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l);
6     /* CS */
7     unlock(&l);
8 }

```

Core 1

l:? → I

Core 2

l:? → I

Core 3

l:? → I

Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1;
7         swap(&(l->lock), &(tmp)); ←
8     } while (tmp == 1);
9 }

11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l);
6     /* CS */
7     unlock(&l);
8 }

```

Core 1

l:1 → M

Core 2

l:? → I

Core 3

l:? → I

Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1;
7         swap(&(l->lock), &(tmp));
8     } while (tmp == 1); ←
9 }

11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l);
6     /* CS */
7     unlock(&l);
8 }

```

Core 1

l:1 → M

Core 2

l:? → I

Core 3

l:? → I

Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1;
7         swap(&(l->lock), &(tmp));
8     } while (tmp == 1);
9 }

11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```

Core 1

l:1 → M

Core 2

l:? → I

Core 3

l:? → I

Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1;
7         swap(&(l->lock), &(tmp));
8     } while (tmp == 1);
9 }

11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l); ←
6     /* CS */ ←
7     unlock(&l);
8 }

```

Core 1

l:1 → M

Core 2

l:? → I

Core 3

l:? → I

Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1; ←
7         swap(&(l->lock), &(tmp));
8     } while (tmp == 1);
9 }

11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```

Core 1

l:1 → M

Core 2

l:? → I

Core 3

l:? → I

Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1;
7         swap(&(l->lock), &(tmp)); ←
8     } while (tmp == 1);
9 }

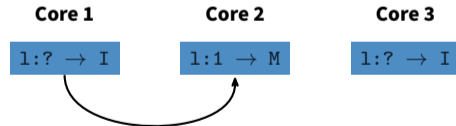
11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```



Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1;
7         swap(&(l->lock), &(tmp));
8     } while (tmp == 1); ←
9 }

11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```

Core 1

l:? → I

Core 2

l:1 → M

Core 3

l:? → I

Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1;
7         swap(&(l->lock), &(tmp));
8     } while (tmp == 1); ←
9 }

11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */
5     lock(&l); ←
6     /* CS */ ←
7     unlock(&l);
8 }

```

Core 1

l:? → I

Core 2

l:1 → M

Core 3

l:? → I

Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1; ←
7         swap(&(l->lock), &(tmp));
8     } while (tmp == 1); ←
9 }

11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```

Core 1

l:? → I

Core 2

l:1 → M

Core 3

l:? → I

Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1;
7         swap(&(l->lock), &(tmp)); ←
8     } while (tmp == 1); ←
9 }

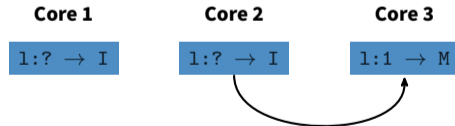
11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```



Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1;
7         swap(&(l->lock), &(tmp));
8     } while (tmp == 1); ←
9 }

11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```

Core 1

l:? → I

Core 2

l:? → I

Core 3

l:1 → M

Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1; ←
7         swap(&(l->lock), &(tmp));
8     } while (tmp == 1); ←
9 }

11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```

Core 1

l:? → I

Core 2

l:? → I

Core 3

l:1 → M

Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1;
7         swap(&(l->lock), &(tmp)); ←
8     } while (tmp == 1); ←
9 }

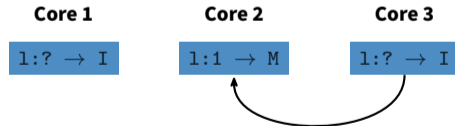
11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```



Test & Set Lock

Overhead

```

1 struct ts_lock_t {
2     volatile int lock;
3 };

4 void lock(ts_lock_t *l) {
5     do {
6         int tmp = 1;
7         swap(&(l->lock), &(tmp));
8     } while (tmp == 1); ←
9 }

11 void unlock(ts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct ts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```

Core 1

l:? → I

Core 2

l:1 → M

Core 3

l:? → I

Test & Test & Set Lock

```
1 struct tts_lock_t {
2     volatile int lock;
3 };
4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1;
7         do {} while (l->lock == 1);
8         swap(&(l->lock), &(tmp));
9     } while (tmp == 1);
10 }
11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }
```

- As simple as Test & Set Lock but with less cache traffic
- Most widespread lock implementation

But

- No fairness between threads

Test & Test & Set Lock

Overhead

```
1 struct tts_lock_t {
2     volatile int lock;
3 };
4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1;
7         do {} while (l->lock == 1);
8         swap(&(l->lock), &(tmp));
9     } while (tmp == 1);
10 }
11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }
```

```
1 struct tts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */
5     lock(&l);
6     /* CS */
7     unlock(&l);
8 }
```

Test & Test & Set Lock

Overhead

```

1 struct tts_lock_t {
2     volatile int lock;
3 };

4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1;
7         do {} while (l->lock == 1);
8         swap(&(l->lock), &(tmp));
9     } while (tmp == 1);
10 }

11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct tts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l);
6     /* CS */
7     unlock(&l);
8 }

```

Core 1

l:? → I

Core 2

l:? → I

Core 3

l:? → I

Test & Test & Set Lock

Overhead

```

1 struct tts_lock_t {
2     volatile int lock;
3 };

4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1;
7         do {} while (l->lock == 1);
8         swap(&(l->lock), &(tmp));
9     } while (tmp == 1);
10 }

11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct tts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l); ←
6     /* CS */
7     unlock(&l);
8 }

```

Core 1

l:? → I

Core 2

l:? → I

Core 3

l:? → I

Test & Test & Set Lock

Overhead

```

1 struct tts_lock_t {
2     volatile int lock;
3 };

4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1; ←
7         do {} while (l->lock == 1);
8         swap(&(l->lock), &(tmp));
9     } while (tmp == 1);
10 }

11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct tts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l);
6     /* CS */
7     unlock(&l);
8 }

```

Core 1

l:? → I

Core 2

l:? → I

Core 3

l:? → I

Test & Test & Set Lock

Overhead

```

1 struct tts_lock_t {
2     volatile int lock;
3 };

4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1;
7         do {} while (l->lock == 1); ←
8         swap(&(l->lock), &(tmp));
9     } while (tmp == 1);
10 }

11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct tts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l);
6     /* CS */
7     unlock(&l);
8 }

```

Core 1

l:0 → E

Core 2

l:? → I

Core 3

l:? → I

Test & Test & Set Lock

Overhead

```

1 struct tts_lock_t {
2     volatile int lock;
3 };

4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1;
7         do {} while (l->lock == 1);
8         swap(&(l->lock), &(tmp)); ←
9     } while (tmp == 1);
10 }

11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct tts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l);
6     /* CS */
7     unlock(&l);
8 }

```

Core 1

l:1 → M

Core 2

l:? → I

Core 3

l:? → I

Test & Test & Set Lock

Overhead

```

1 struct tts_lock_t {
2     volatile int lock;
3 };

4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1;
7         do {} while (l->lock == 1);
8         swap(&(l->lock), &(tmp));
9     } while (tmp == 1); ←
10 }

11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct tts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l);
6     /* CS */
7     unlock(&l);
8 }

```

Core 1

l:1 → M

Core 2

l:? → I

Core 3

l:? → I

Test & Test & Set Lock

Overhead

```

1 struct tts_lock_t {
2     volatile int lock;
3 };

4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1;
7         do {} while (l->lock == 1);
8         swap(&(l->lock), &(tmp));
9     } while (tmp == 1);
10 }

11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct tts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```

Core 1

l:1 → M

Core 2

l:? → I

Core 3

l:? → I

Test & Test & Set Lock

Overhead

```

1 struct tts_lock_t {
2     volatile int lock;
3 };

4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1;
7         do {} while (l->lock == 1);
8         swap(&(l->lock), &(tmp));
9     } while (tmp == 1);
10 }

11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct tts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l); ←
6     /* CS */ ←
7     unlock(&l);
8 }

```

Core 1

l:1 → M

Core 2

l:? → I

Core 3

l:? → I

Test & Test & Set Lock

Overhead

```

1 struct tts_lock_t {
2     volatile int lock;
3 };

4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1; ←
7         do {} while (l->lock == 1);
8         swap(&(l->lock), &(tmp));
9     } while (tmp == 1);
10 }

11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct tts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```

Core 1

l:1 → M

Core 2

l:? → I

Core 3

l:? → I

Test & Test & Set Lock

Overhead

```

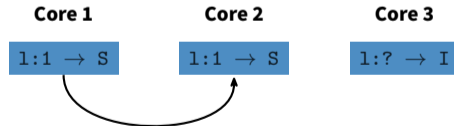
1 struct tts_lock_t {
2     volatile int lock;
3 };
4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1;
7         do {} while (l->lock == 1); ←
8         swap(&(l->lock), &(tmp));
9     } while (tmp == 1);
10 }
11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct tts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */ ←
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```



Test & Test & Set Lock

Overhead

```

1 struct tts_lock_t {
2     volatile int lock;
3 };

4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1;
7         do {} while (l->lock == 1); ←
8         swap(&(l->lock), &(tmp));
9     } while (tmp == 1);
10 }

11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct tts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */
5     lock(&l); ←
6     /* CS */ ←
7     unlock(&l);
8 }

```

Core 1

l:1 → S

Core 2

l:1 → S

Core 3

l:? → I

Test & Test & Set Lock

Overhead

```

1 struct tts_lock_t {
2     volatile int lock;
3 };
4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1; ←
7         do {} while (l->lock == 1); ←
8         swap(&(l->lock), &(tmp));
9     } while (tmp == 1);
10 }
11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct tts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```

Core 1

l:1 → S

Core 2

l:1 → S

Core 3

l:? → I

Test & Test & Set Lock

Overhead

```

1 struct tts_lock_t {
2     volatile int lock;
3 };

4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1;
7         do {} while (l->lock == 1); ←
8         swap(&(l->lock), &(tmp));
9     } while (tmp == 1);
10 }

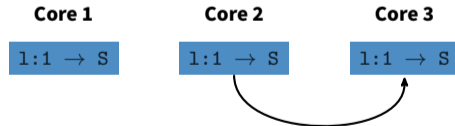
11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct tts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```



Test & Test & Set Lock

Overhead

```

1 struct tts_lock_t {
2     volatile int lock;
3 };

4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1;
7         do {} while (l->lock == 1); ←
8         swap(&(l->lock), &(tmp));
9     } while (tmp == 1);
10 }

11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct tts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```

Core 1

1:1 → S

Core 2

1:1 → S



Core 3

1:1 → S

Test & Test & Set Lock

Overhead

```

1 struct tts_lock_t {
2     volatile int lock;
3 };

4 void lock(tts_lock_t *l) {
5     do {
6         int tmp = 1;
7         do {} while (l->lock == 1); ←
8         swap(&(l->lock), &(tmp));
9     } while (tmp == 1);
10 }

11 void unlock(tts_lock_t *l) {
12     l->lock = 0;
13 }

```

```

1 struct tts_lock_t l;
2
3 void thread_fn(void) {
4     /* Other stuff */
5     lock(&l);
6     /* CS */ ←
7     unlock(&l);
8 }

```

Core 1

1:1 → S

Core 2

1:1 → S

Core 3

1:1 → S



Test & Test & Set Lock

Fairness

```
1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }

```



```
12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1);
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }
```

 T_1  T_2  T_3

Test & Test & Set Lock

Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←←←
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1);
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```


 T₁

 T₂

 T₃

Test & Test & Set Lock

Fairness

```
1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l); ←
7         /* CS */
8         unlock(&l);
9     }
10 }

```



```
12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1);
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }
```

 T₁ T₂ T₃

Test & Test & Set Lock

Fairness

```
1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }

```



```
12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1; ←
15         do {} while (l->lock == 1);
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }
```

 T_1  T_2  T_3

Test & Test & Set Lock

Fairness

```
1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }

```



```
12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }
```

 T₁ T₂ T₃

Test & Test & Set Lock

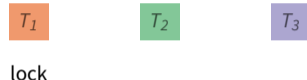
Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1);
16         swap(&(l->lock), &(tmp)); ←
17     } while (tmp == 1);
18 }

```



Test & Test & Set Lock

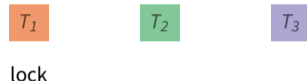
Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1);
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```



Test & Test & Set Lock

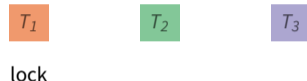
Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l); ←
7         /* CS */ ←
8         unlock(&l);
9     }
10 }

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1);
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```



Test & Test & Set Lock

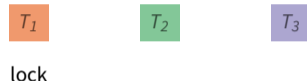
Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1; ←
15         do {} while (l->lock == 1);
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```



Test & Test & Set Lock

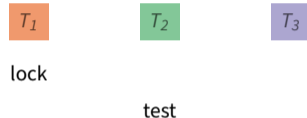
Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```



Test & Test & Set Lock

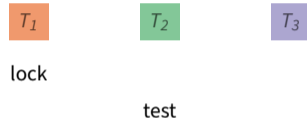
Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l); ←
7         /* CS */ ←
8         unlock(&l);
9     }
10 }

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```



Test & Test & Set Lock

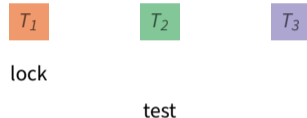
Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1; ←
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```



Test & Test & Set Lock

Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```


 T₁

lock


 T₂

test


 T₃

test

Test & Test & Set Lock

Fairness

```

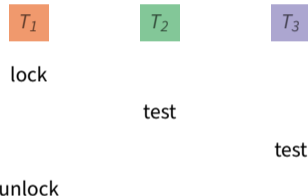
1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */
8         unlock(&l); ←
9     }
10 }

```

```

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```



Test & Test & Set Lock

Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```


 T₁

lock

unlock


 T₂

test


 T₃

test

Test & Test & Set Lock

Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }

```

```

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp)); ←
17     } while (tmp == 1);
18 }

```


 T₁

lock

unlock


 T₂

test

lock


 T₃

test

Test & Test & Set Lock

Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```


 T₁

lock

unlock


 T₂

test

lock


 T₃

test

Test & Test & Set Lock

Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }

```

```

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```


 T₁

lock

unlock


 T₂

test

lock


 T₃

test

test

Test & Test & Set Lock

Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */
8         unlock(&l); ←
9     }
10 }
12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```


 T₁

lock

unlock


 T₂

test

lock

unlock


 T₃

test

test

Test & Test & Set Lock

Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }

```

```

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```


 T₁

lock

unlock


 T₂

test

lock

unlock


 T₃

test

test

Test & Test & Set Lock

Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l); ←
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```


 T₁

lock

unlock


 T₂

test

lock

unlock


 T₃

test

test

Test & Test & Set Lock

Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1; ←
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```


 T₁

lock

unlock


 T₂

test

lock

unlock


 T₃

test

test

Test & Test & Set Lock

Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```


 T₁

lock

unlock


 T₂

test

lock

unlock


 T₃

test

test

Test & Test & Set Lock

Fairness

```

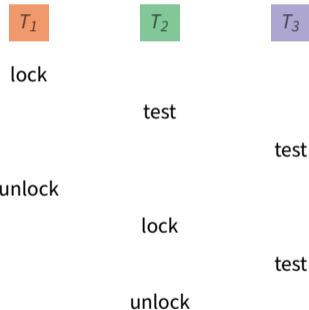
1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }

```

```

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp)); ←
17     } while (tmp == 1);
18 }

```



Test & Test & Set Lock

Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }

```

```

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```


 T₁

lock

unlock

lock


 T₂

test

lock

unlock


 T₃

test

test

Test & Test & Set Lock

Fairness

```

1 struct tts_lock l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }

```

```

12 void lock(tts_lock *l) {
13     do {
14         int tmp = 1;
15         do {} while (l->lock == 1); ←
16         swap(&(l->lock), &(tmp));
17     } while (tmp == 1);
18 }

```


 T₁

lock

unlock

lock


 T₂

test

lock

unlock


 T₃

test

test

test

Ticket Lock

```
1 struct ticket_lock_t {
2     int next;
3     volatile int current;
4 };

5 void lock(ticket_lock_t *l) {
6     int t = xadd(&(l->next), 1);
7     do {} while (l->current != t);
8 }

10 void unlock(ticket_lock_t *l) {
11     l->current++;
12 }
```

- As simple and cheap as Test & Test & Set Lock
- Ensures fairness between threads

But

- High bus traffic on `unlock`
- Aborting `lock` is difficult

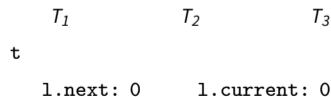
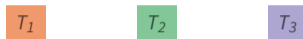
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t);
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



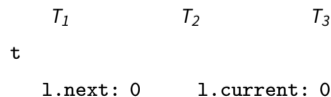
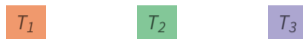
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ← ← ←
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t);
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



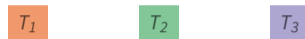
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l); ←
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t);
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



lock

 T_1 T_2 T_3

t

l.next: 0 l.current: 0

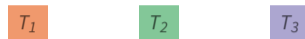
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1); ←
14     do {} while (l->current != t);
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



lock

	T_1	T_2	T_3
t	0		
	l.next: 1	l.current: 0	

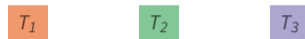
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



lock

	T_1	T_2	T_3
t	0		
	l.next: 1	l.current: 0	

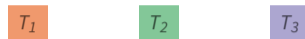
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t);
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



lock

	T_1	T_2	T_3
t	0		
	l.next: 1	l.current: 0	

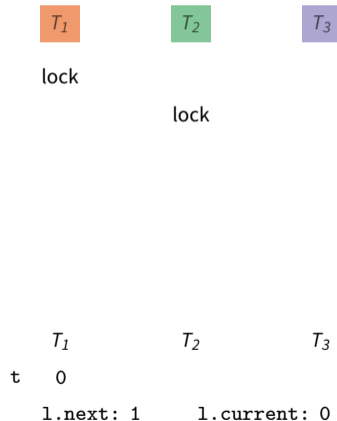
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l); ←
7         /* CS */ ←
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t);
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



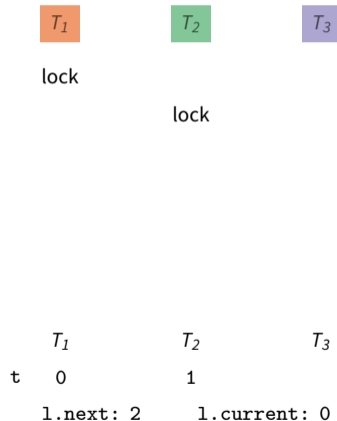
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1); ←
14     do {} while (l->current != t);
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



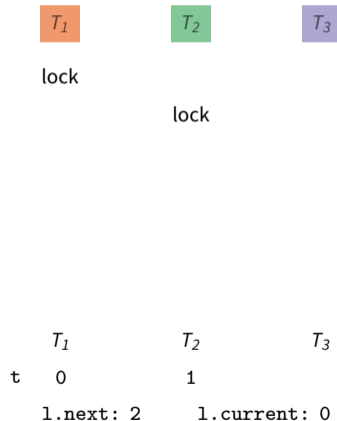
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



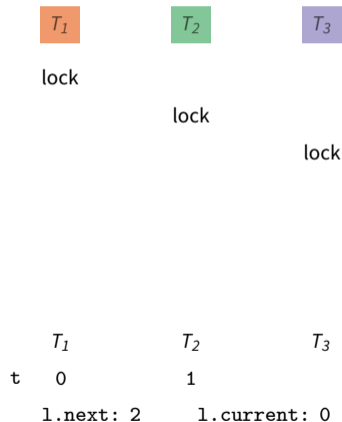
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l); ←
7         /* CS */ ←
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



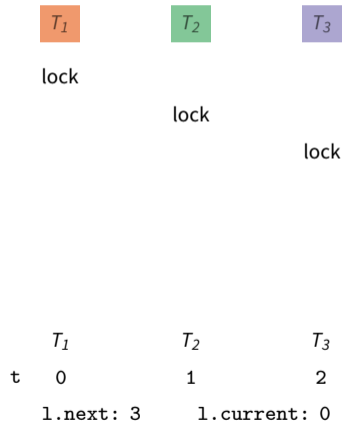
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1); ←
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



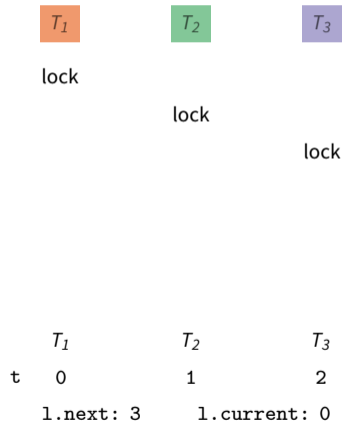
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



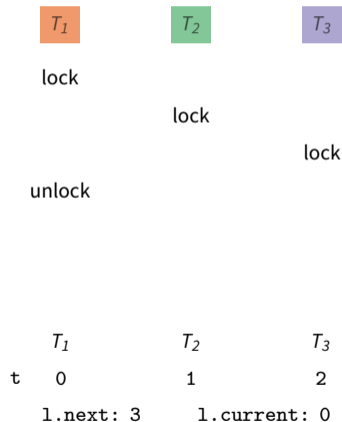
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */
8         unlock(&l); ←
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



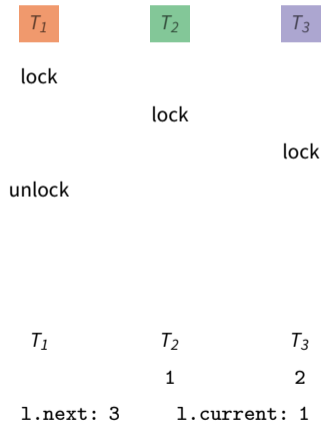
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++; ←
19 }

```



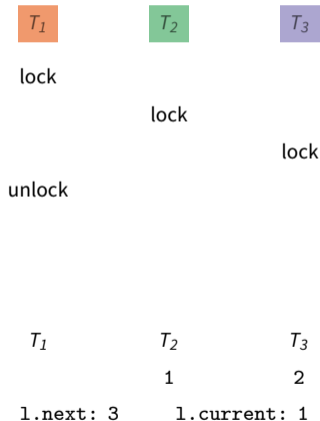
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



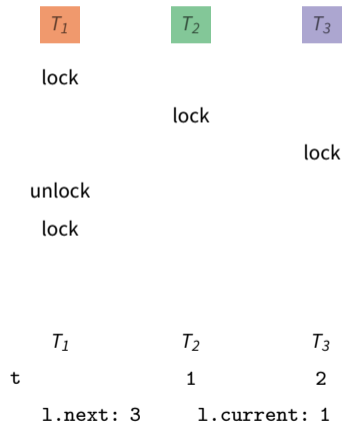
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l); ←
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



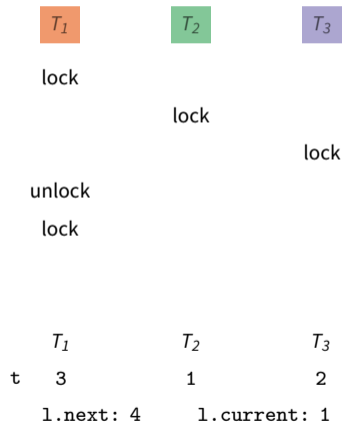
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1); ←
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



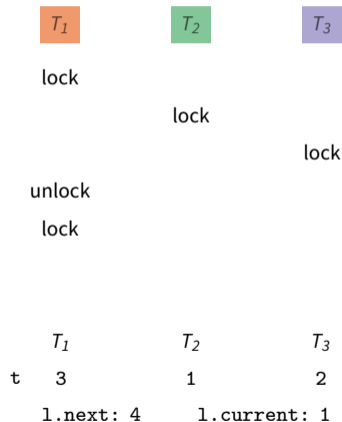
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t);
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



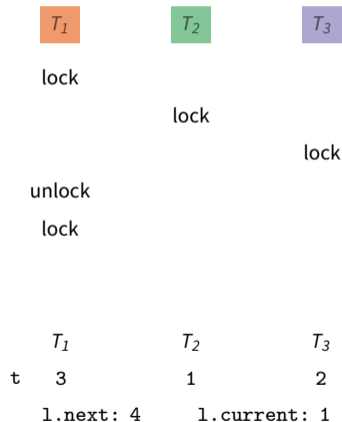
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



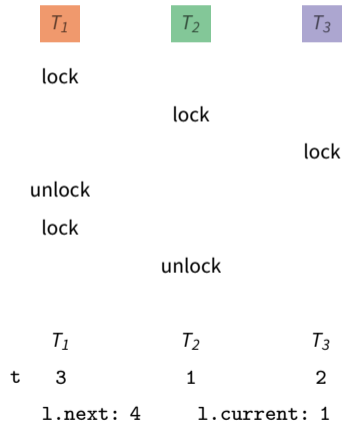
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */
8         unlock(&l); ←
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



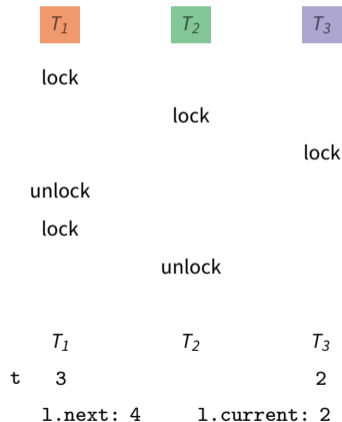
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++; ←
19 }

```



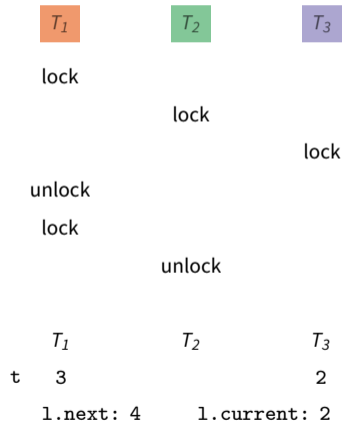
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



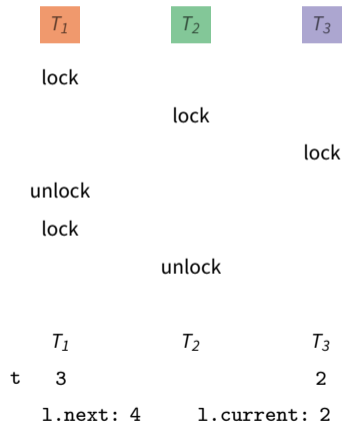
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



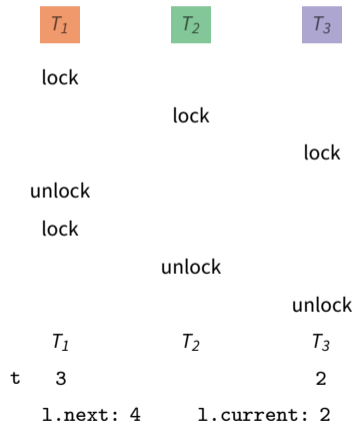
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */
8         unlock(&l); ←
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



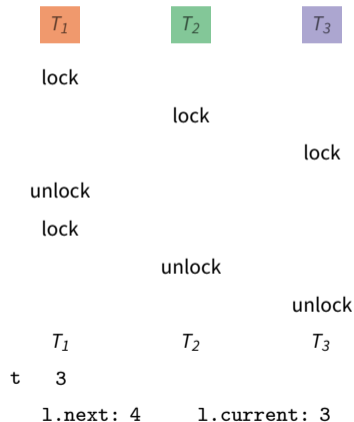
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++; ←
19 }

```



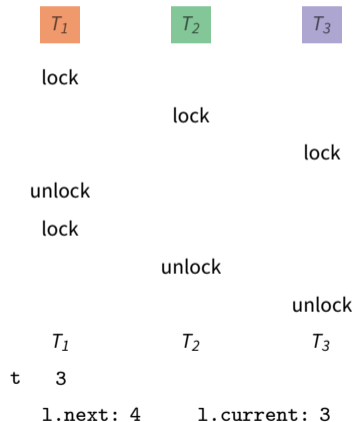
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t);
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++; ←
19 }

```



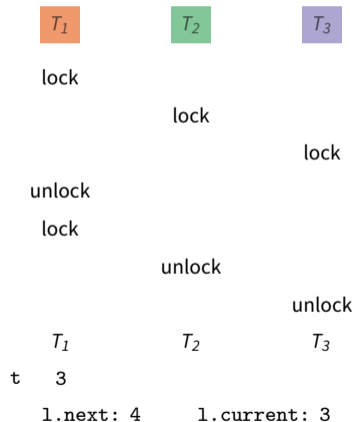
Ticket Lock

Fairness

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */ ←
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t);
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



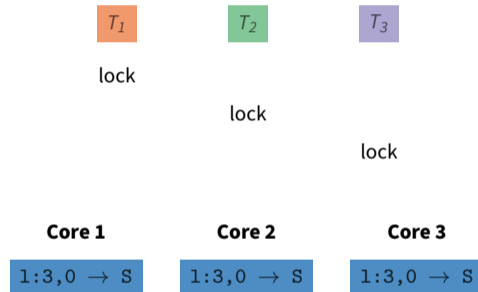
Ticket Lock

Overhead

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */ ←
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



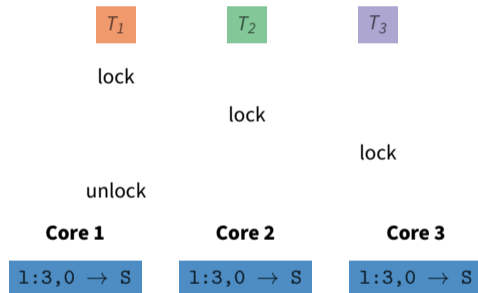
Ticket Lock

Overhead

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */
8         unlock(&l); ←
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



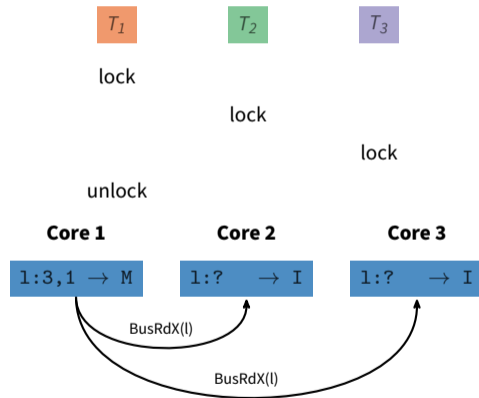
Ticket Lock

Overhead

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++; ←
19 }

```



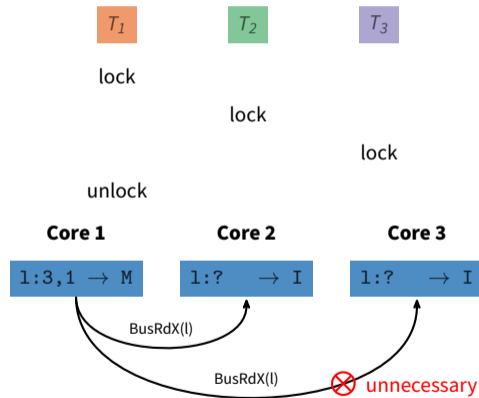
Ticket Lock

Overhead

```

1 struct ticket_lock_t l;
2
3 void thread_fn(void) {
4     while (true) {
5         /* Other stuff */
6         lock(&l);
7         /* CS */
8         unlock(&l);
9     }
10 }
12 void lock(ticket_lock_t *l) {
13     int t = xadd(&(l->next), 1);
14     do {} while (l->current != t); ←
15 }
17 void unlock(ticket_lock_t *l) {
18     l->current++;
19 }

```



Lock-Free Data Structures

Many data structures can be implemented without the usage of locks but instead directly with atomic hardware instructions.

- Single-Linked List

```
1 void insert(ele_t *new_ele, ele_t *prev) {  
2     do {  
3         new_ele->next = prev->next;  
4     } while (!cas(&(prev->next), new_ele->next, new_ele));  
5 }
```

Lock-Free Data Structures

Many data structures can be implemented without the usage of locks but instead directly with atomic hardware instructions.

- Single-Linked List
- Double-Linked List

```
1 void insert(ele_t *new_ele, ele_t *prev) {
2     do {
3         new_ele->next = prev->next;
4         new_ele->prev = prev;
5     } while (!dcas(&(prev->next), &(prev->next->prev),
6                 new_ele->next, new_ele->prev,
7                 new_ele, new_ele));
8 }
```

Lock-Free Data Structures

Many data structures can be implemented without the usage of locks but instead directly with atomic hardware instructions.

- Single-Linked List
- Double-Linked List
- Binary Tree

Lock-Free Data Structures

Many data structures can be implemented without the usage of locks but instead directly with atomic hardware instructions.

- Single-Linked List
- Double-Linked List
- Binary Tree
- Red-Black Tree

Mellor-Crummey and Scott (MCS Lock)

```
1 struct mcs_node_t {          6 struct mcs_lock_t {
2     mcs_node_t *next;      7     mcs_node_t *queue;
3     bool free;            8 };
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }
```

- Fair between threads
- Only local spinning
- No unnecessary cache trashing
- Easy to abort `lock` operation

But:

- Difficult to implement correctly

MCS Lock

Fairness & Overhead

```

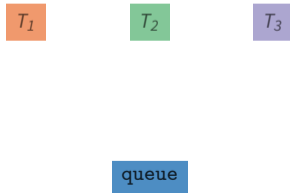
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
5
6 void lock(mcs_lock_t *l, mcs_node_t *cur) {
7     cur->next = NULL; cur->free = false;
8     auto prev = fetch_and_store(&(l->queue), cur);
9     if (prev) {
10        prev->next = cur;
11        do {} while (!cur->free);
12    }
13 }
14
15 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
16     if (!cur->next) {
17         if (cas(&(l->queue), cur, NULL)) return;
18         do {} while (!cur->next);
19     }
20     cur->next->free = true;
21 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
5
6 void lock(mcs_lock_t *l, mcs_node_t *cur) {
7     cur->next = NULL; cur->free = false;
8     auto prev = fetch_and_store(&(l->queue), cur);
9     if (prev) {
10        prev->next = cur;
11        do {} while (!cur->free);
12    }
13 }
14
15 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
16     if (!cur->next) {
17         if (cas(&(l->queue), cur, NULL)) return;
18         do {} while (!cur->next);
19     }
20     cur->next->free = true;
21 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n); ←
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```


 T_1

 T_2

 T_3

 queue

MCS Lock

Fairness & Overhead

```

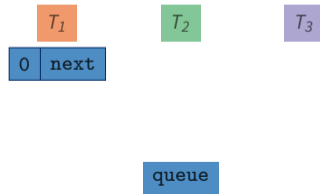
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
5
6 void lock(mcs_lock_t *l, mcs_node_t *cur) {
7     cur->next = NULL; cur->free = false; ←
8     auto prev = fetch_and_store(&(l->queue), cur);
9     if (prev) {
10        prev->next = cur;
11        do {} while (!cur->free);
12    }
13 }
14
15 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
16     if (!cur->next) {
17         if (cas(&(l->queue), cur, NULL)) return;
18         do {} while (!cur->next);
19     }
20     cur->next->free = true;
21 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

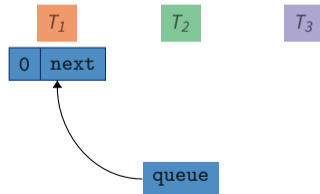
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur); ←
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

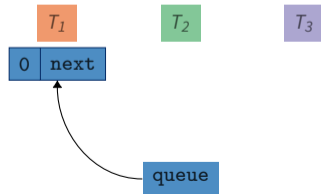
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {←
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

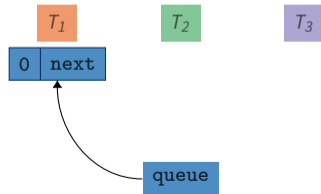
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

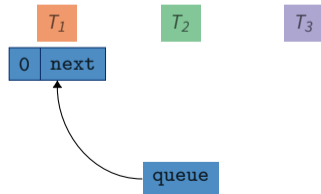
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n); ←
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

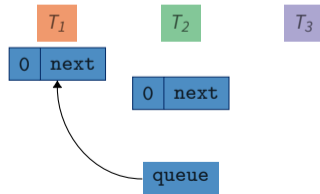
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false; ←
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

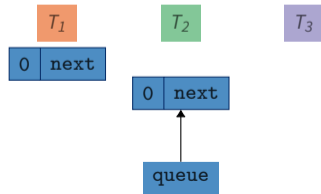
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur); ←
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

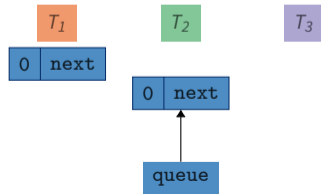
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {←
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

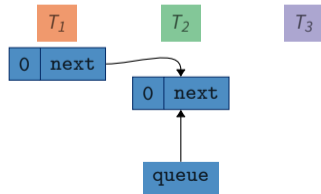
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur; ←
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

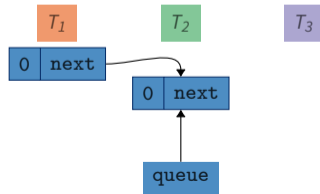
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

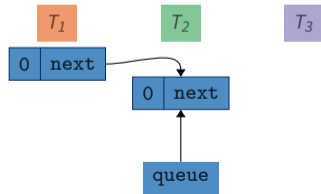
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */
7         lock(&l, &n); ←
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

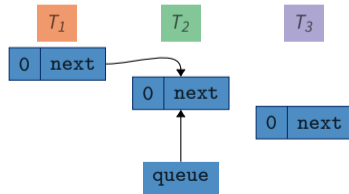
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false; ←
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */
7         lock(&l, &n);
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

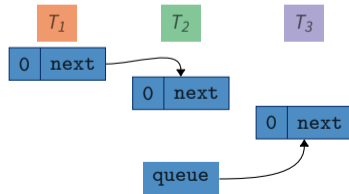
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur); ←
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */
7         lock(&l, &n);
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

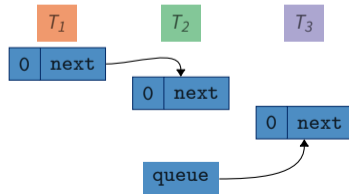
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {←
14         prev->next = cur;
15         do {} while (!cur->free);←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */
7         lock(&l, &n);
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

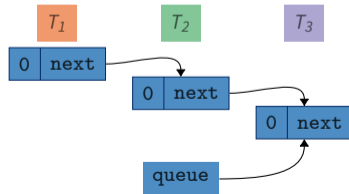
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur; ←
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */
7         lock(&l, &n);
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

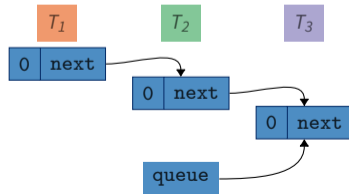
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */
7         lock(&l, &n);
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

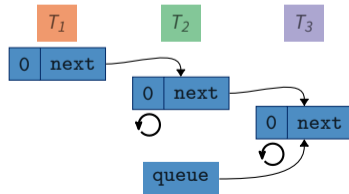
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */
7         lock(&l, &n);
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

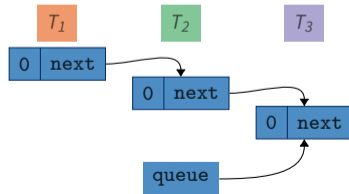
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n); ←
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

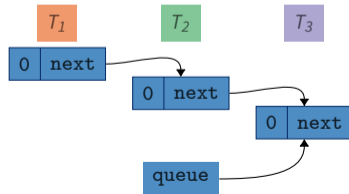
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) { ←
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

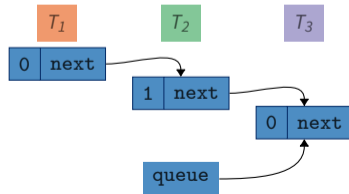
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true; ←
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

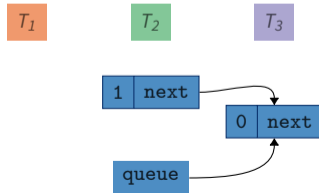
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

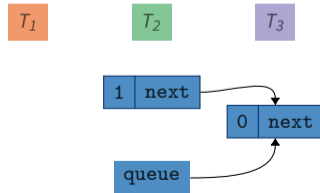
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

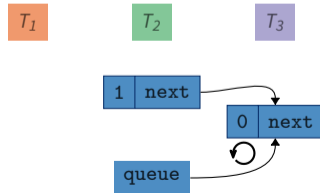
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

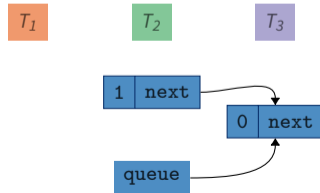
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n); ←
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

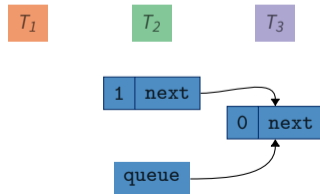
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) { ←
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

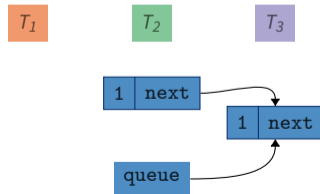
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true; ←
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

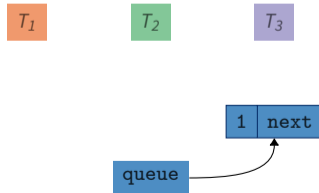
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
5
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
18
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

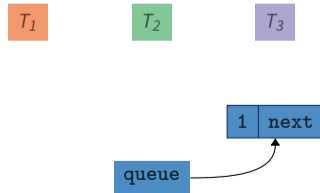
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

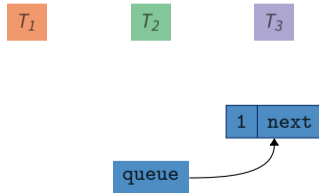
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n); ←
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

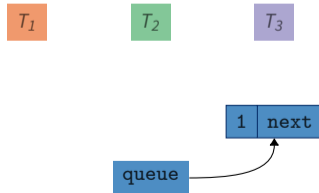
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) { ←
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

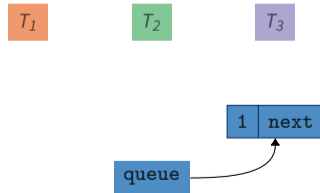
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) { ←
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n); ←
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

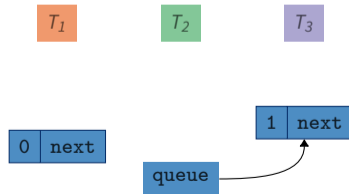
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false; ←
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) { ←
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

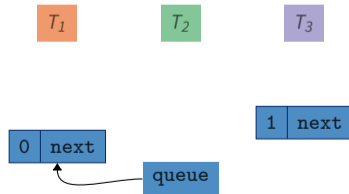
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur); ←
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) { ←
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

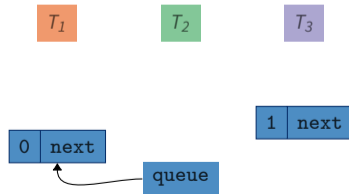
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {←
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {←
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

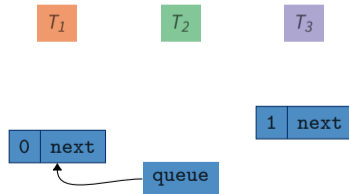
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {←
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;←
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

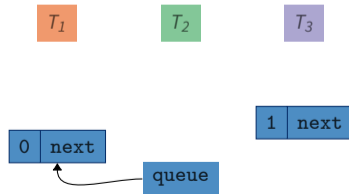
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {←
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);←
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

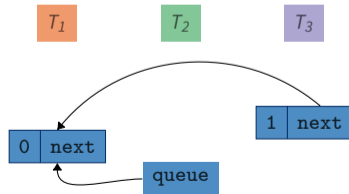
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur; ←
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next); ←
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

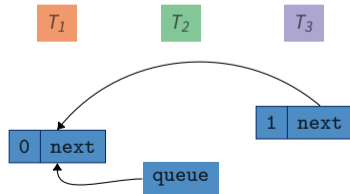
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next); ←
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

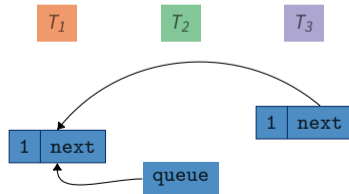
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true; ←
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

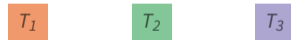
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free); ←
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

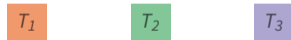
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */ ←
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

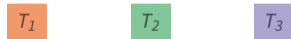
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n); ←
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

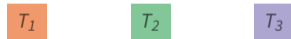
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) { ←
21         if (cas(&(l->queue), cur, NULL)) return;
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

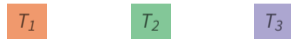
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
10 void lock(mcs_lock_t *l, mcs_node_t *cur) {
11     cur->next = NULL; cur->free = false;
12     auto prev = fetch_and_store(&(l->queue), cur);
13     if (prev) {
14         prev->next = cur;
15         do {} while (!cur->free);
16     }
17 }
19 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
20     if (!cur->next) {
21         if (cas(&(l->queue), cur, NULL)) return; ←
22         do {} while (!cur->next);
23     }
24     cur->next->free = true;
25 }

```

```

1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */ ←
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



MCS Lock

Fairness & Overhead

```

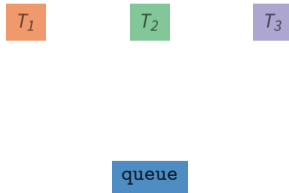
1 struct mcs_node_t {
2     mcs_node_t *next;
3     bool free;
4 };
5
6 void lock(mcs_lock_t *l, mcs_node_t *cur) {
7     cur->next = NULL; cur->free = false;
8     auto prev = fetch_and_store(&(l->queue), cur);
9     if (prev) {
10        prev->next = cur;
11        do {} while (!cur->free);
12    }
13 }
14
15 void unlock(mcs_lock_t *l, mcs_node_t *cur) {
16     if (!cur->next) {
17         if (cas(&(l->queue), cur, NULL)) return;
18         do {} while (!cur->next);
19     }
20     cur->next->free = true;
21 }

```

```

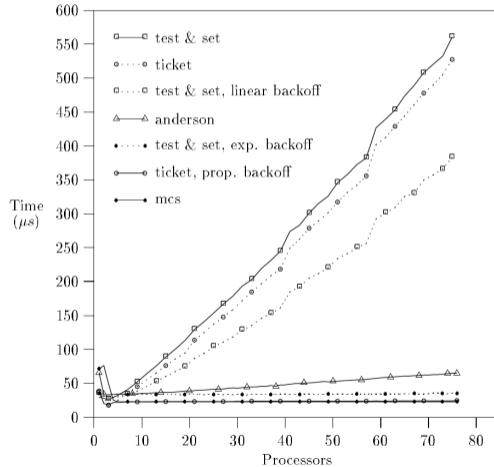
1 struct mcs_lock_t l;
2
3 void thread_fn(void) {
4     struct mcs_node_t n;
5     while (true) {
6         /* Other stuff */
7         lock(&l, &n);
8         /* CS */
9         unlock(&l, &n);
10    }
11 }

```



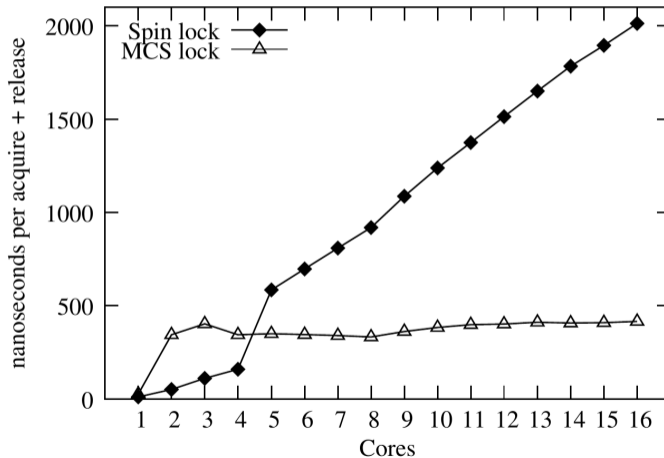
MCS Lock

Performance



MCS Lock

Performance



Reader Writer Lock

Differentiate between two types of lock holders:

- Readers
 - Do not modify the *lock-protected* object
 - Multiple readers can use the object at the same time
- Writers
 - Modify the *lock-protected* object
 - Requires exclusive access to the object (no other readers *or* writers)

Different levels of fairness can be implemented:

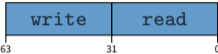
- Readers and writes get access granted in the order they appear → fair
- Later readers can overtake earlier writers → unfair for writers
- Later writers can overtake earlier readers → unfair for readers

Reader Writer Lock

Fair Ticket Reader Writer Lock

```
1 struct rw_lock_t {
2     rw_union_t current;
3     rw_union_t next;
4 };
5
6 void lock_read(rw_lock_t *l) {
7     auto t = xadd(&(l->next), 1);
8     do {} while (l->current.write != t.write);
9 }
10 void lock_write(rw_lock_t *l) {
11     auto t = xadd(&(l->next.write), 1);
12     do {} while (l->current != t);
13 }
14
15 void unlock_read(rw_lock_t *l) {
16     xadd(&(l->current.read), 1);
17 }
18 void unlock_write(rw_lock_t *l) {
19     l->current.write++;
20 }
```

rw_union_t



The diagram shows a horizontal bar representing the `rw_union_t` structure. It is divided into two sections: a left section labeled 'write' and a right section labeled 'read'. Below the bar, bit positions are indicated: '63' is at the left end, '31' is at the boundary between 'write' and 'read', and '0' is at the right end.

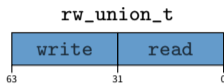
Reader Writer Lock

Fair Ticket Reader Writer Lock

```

1 struct rw_lock_t {
2     rw_union_t current;
3     rw_union_t next;
4 };
5
6 void lock_read(rw_lock_t *l) {
7     auto t = xadd(&(l->next), 1);
8     do {} while (l->current.write != t.write);
9 }
10 void lock_write(rw_lock_t *l) {
11     auto t = xadd(&(l->next.write), 1);
12     do {} while (l->current != t);
13 }
14
15 void unlock_read(rw_lock_t *l) {
16     xadd(&(l->current.read), 1);
17 }
18 void unlock_write(rw_lock_t *l) {
19     l->current.write++;
20 }

```



 T_1

 T_2

 T_3

T_1 T_2 T_3
 t
 l.next: 0|0 l.current: 0|0

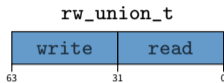
Reader Writer Lock

Fair Ticket Reader Writer Lock

```

1 struct rw_lock_t {
2     rw_union_t current;
3     rw_union_t next;
4 };
5
6 void lock_read(rw_lock_t *l) {
7     auto t = xadd(&(l->next), 1);
8     do {} while (l->current.write != t.write);
9 }
10 void lock_write(rw_lock_t *l) {
11     auto t = xadd(&(l->next.write), 1);
12     do {} while (l->current != t);
13 }
14
15 void unlock_read(rw_lock_t *l) {
16     xadd(&(l->current.read), 1);
17 }
18 void unlock_write(rw_lock_t *l) {
19     l->current.write++;
20 }

```



T_1
/* CS */
read

T_2

T_3

	T_1	T_2	T_3
t	0 0		
l.next:	0 1		
l.current:	0 0		

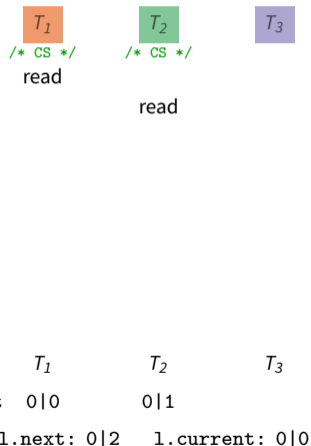
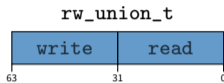
Reader Writer Lock

Fair Ticket Reader Writer Lock

```

1 struct rw_lock_t {
2     rw_union_t current;
3     rw_union_t next;
4 };
5
6 void lock_read(rw_lock_t *l) {
7     auto t = xadd(&(l->next), 1);
8     do {} while (l->current.write != t.write);
9 }
10 void lock_write(rw_lock_t *l) {
11     auto t = xadd(&(l->next.write), 1);
12     do {} while (l->current != t);
13 }
14
15 void unlock_read(rw_lock_t *l) {
16     xadd(&(l->current.read), 1);
17 }
18 void unlock_write(rw_lock_t *l) {
19     l->current.write++;
20 }

```



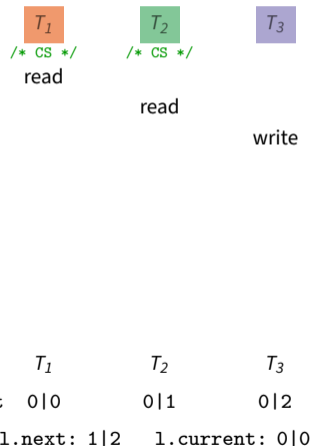
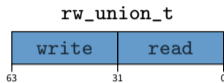
Reader Writer Lock

Fair Ticket Reader Writer Lock

```

1 struct rw_lock_t {
2     rw_union_t current;
3     rw_union_t next;
4 };
5
6 void lock_read(rw_lock_t *l) {
7     auto t = xadd(&(l->next), 1);
8     do {} while (l->current.write != t.write);
9 }
10 void lock_write(rw_lock_t *l) {
11     auto t = xadd(&(l->next.write), 1);
12     do {} while (l->current != t);
13 }
14
15 void unlock_read(rw_lock_t *l) {
16     xadd(&(l->current.read), 1);
17 }
18 void unlock_write(rw_lock_t *l) {
19     l->current.write++;
20 }

```



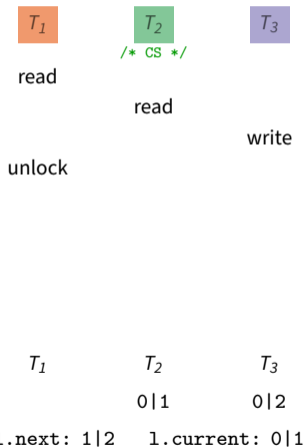
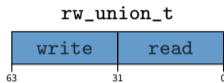
Reader Writer Lock

Fair Ticket Reader Writer Lock

```

1 struct rw_lock_t {
2     rw_union_t current;
3     rw_union_t next;
4 };
5
6 void lock_read(rw_lock_t *l) {
7     auto t = xadd(&(l->next), 1);
8     do {} while (l->current.write != t.write);
9 }
10 void lock_write(rw_lock_t *l) {
11     auto t = xadd(&(l->next.write), 1);
12     do {} while (l->current != t);
13 }
14
15 void unlock_read(rw_lock_t *l) {
16     xadd(&(l->current.read), 1);
17 }
18 void unlock_write(rw_lock_t *l) {
19     l->current.write++;
20 }

```



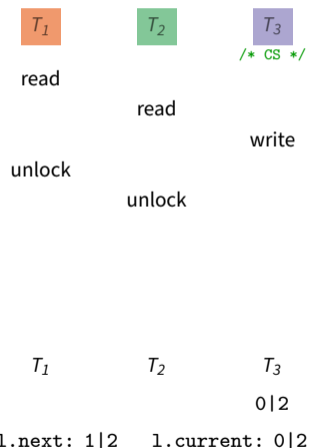
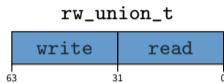
Reader Writer Lock

Fair Ticket Reader Writer Lock

```

1 struct rw_lock_t {
2     rw_union_t current;
3     rw_union_t next;
4 };
5
6 void lock_read(rw_lock_t *l) {
7     auto t = xadd(&(l->next), 1);
8     do {} while (l->current.write != t.write);
9 }
10 void lock_write(rw_lock_t *l) {
11     auto t = xadd(&(l->next.write), 1);
12     do {} while (l->current != t);
13 }
14
15 void unlock_read(rw_lock_t *l) {
16     xadd(&(l->current.read), 1);
17 }
18 void unlock_write(rw_lock_t *l) {
19     l->current.write++;
20 }

```



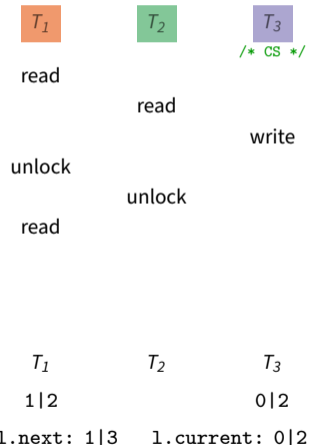
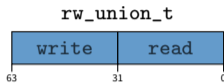
Reader Writer Lock

Fair Ticket Reader Writer Lock

```

1 struct rw_lock_t {
2     rw_union_t current;
3     rw_union_t next;
4 };
5
6 void lock_read(rw_lock_t *l) {
7     auto t = xadd(&(l->next), 1);
8     do {} while (l->current.write != t.write);
9 }
10 void lock_write(rw_lock_t *l) {
11     auto t = xadd(&(l->next.write), 1);
12     do {} while (l->current != t);
13 }
14
15 void unlock_read(rw_lock_t *l) {
16     xadd(&(l->current.read), 1);
17 }
18 void unlock_write(rw_lock_t *l) {
19     l->current.write++;
20 }

```

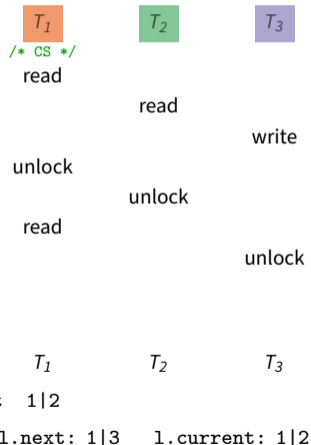
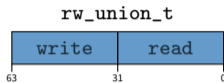


Reader Writer Lock

Fair Ticket Reader Writer Lock

```

1 struct rw_lock_t {
2     rw_union_t current;
3     rw_union_t next;
4 };
5
6 void lock_read(rw_lock_t *l) {
7     auto t = xadd(&(l->next), 1);
8     do {} while (l->current.write != t.write);
9 }
10 void lock_write(rw_lock_t *l) {
11     auto t = xadd(&(l->next.write), 1);
12     do {} while (l->current != t);
13 }
14
15 void unlock_read(rw_lock_t *l) {
16     xadd(&(l->current.read), 1);
17 }
18 void unlock_write(rw_lock_t *l) {
19     l->current.write++;
20 }
  
```



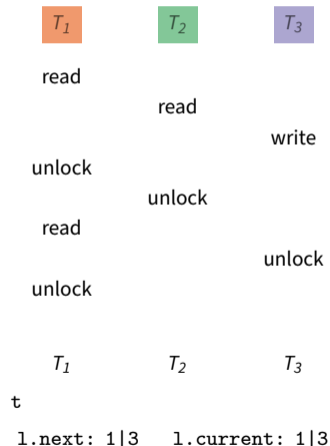
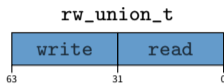
Reader Writer Lock

Fair Ticket Reader Writer Lock

```

1 struct rw_lock_t {
2     rw_union_t current;
3     rw_union_t next;
4 };
5
6 void lock_read(rw_lock_t *l) {
7     auto t = xadd(&(l->next), 1);
8     do {} while (l->current.write != t.write);
9 }
10 void lock_write(rw_lock_t *l) {
11     auto t = xadd(&(l->next.write), 1);
12     do {} while (l->current != t);
13 }
14
15 void unlock_read(rw_lock_t *l) {
16     xadd(&(l->current.read), 1);
17 }
18 void unlock_write(rw_lock_t *l) {
19     l->current.write++;
20 }

```



Timeouts & Aborting Locks

In some circumstances, acquiring a lock should be aborted after a specific timeout.
Thus raising the question, how can this be done?

Timeouts & Aborting Locks

In some circumstances, acquiring a lock should be aborted after a specific timeout.
Thus raising the question, how can this be done?

- Test & Set Lock or Test & Test & Set Lock
 - `lock` operation can simply be aborted (just `return`)

Timeouts & Aborting Locks

In some circumstances, acquiring a lock should be aborted after a specific timeout.
Thus raising the question, how can this be done?

- Test & Set Lock or Test & Test & Set Lock
 - `lock` operation can simply be aborted (just `return`)
- Ticket Lock
 - Update global next-in-queue (`l->next`) and thread local ticket (`t`) variables
 - Very difficult to not make any mistakes

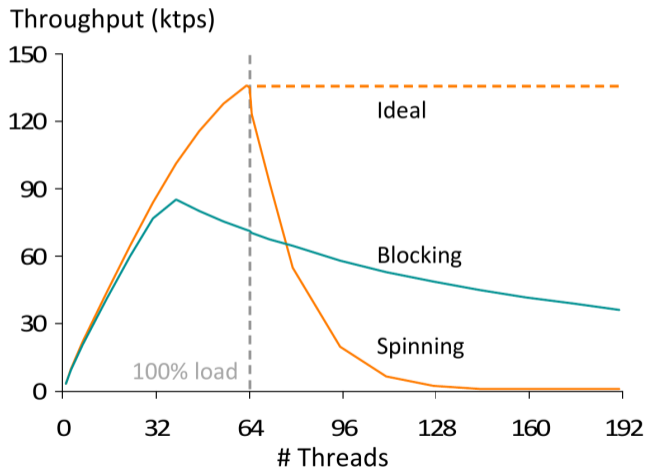
Timeouts & Aborting Locks

In some circumstances, acquiring a lock should be aborted after a specific timeout. Thus raising the question, how can this be done?

- Test & Set Lock or Test & Test & Set Lock
 - `lock` operation can simply be aborted (just `return`)
- Ticket Lock
 - Update global next-in-queue (`l->next`) and thread local ticket (`t`) variables
 - Very difficult to not make any mistakes
- MCS Lock
 - `lock` operation can be aborted by dequeuing the thread from the internal queue

Lockholder Preemption

Spinning vs. Blocking



Lockholder Preemption

Spinning vs. Blocking

Wait time of a thread is increased by the time the current lock holder cannot execute.

- Thread in CS gets preempted by the scheduler due to ready (*but spinning*) threads
- Especially problematic for Ticket Locks and MCS Locks

Blocking

- Actively prevent the waiting thread from executing
- Reduces the system load and thereby the chance for lock holder preemption
- Requires OS support and adds additional overhead to the `lock` operation

Disabling Interrupts

- Prevents the scheduler from preempting the currently running thread
- Only allowed in the kernel because of its great power (`cli + sti` and `pushf + popf`)