Distributed Operating Systems

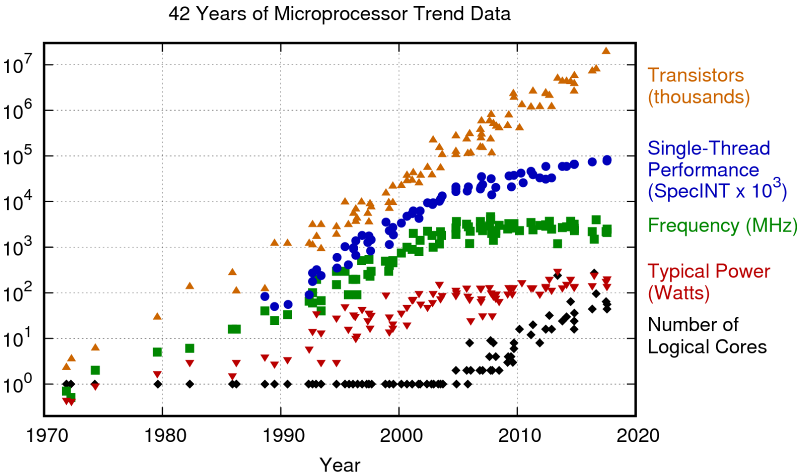
Synchronization in Parallel Systems

Till Smejkal

May 30th, 2022

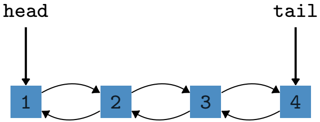
TECHNISCHE UNIVERSITÄT DRESDEN

Why do we need synchronization?

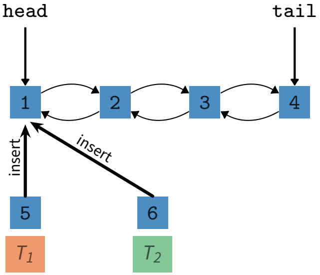


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

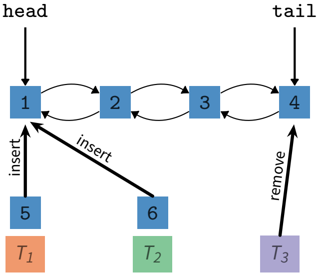
Why do we need synchronization?



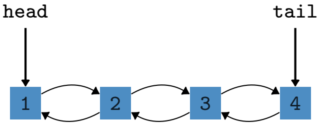
Why do we need synchronization?



Why do we need synchronization?

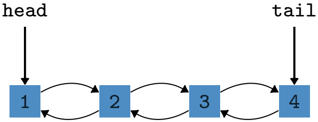


Why do we need synchronization?




```
struct ele_t *new_ele = new ele_t;
new_ele->next = head;
head->prev = new_ele;
head = new_ele;
```

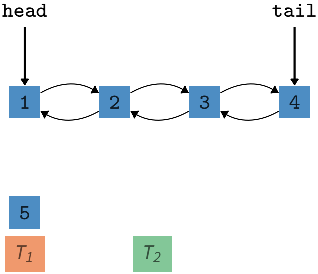
Why do we need synchronization?



T_1 T_2

```
struct ele_t *new_ele = new ele_t;
new_ele->next = head;
head->prev = new_ele;
head = new_ele;
```

Why do we need synchronization?

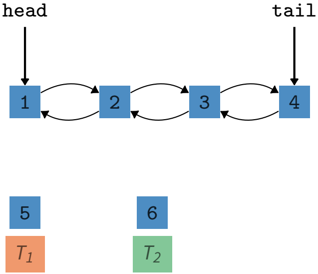


T_1 T_2

1.

```
struct ele_t *new_ele = new ele_t;
new_ele->next = head;
head->prev = new_ele;
head = new_ele;
```

Why do we need synchronization?

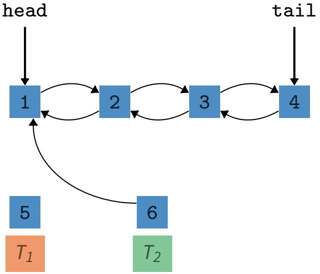


T_1 T_2

1. 2.

```
struct ele_t *new_ele = new ele_t;
new_ele->next = head;
head->prev = new_ele;
head = new_ele;
```

Why do we need synchronization?

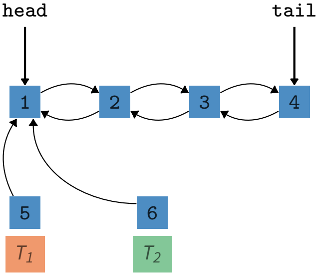


T_1 T_2

1\. 2.
3\.

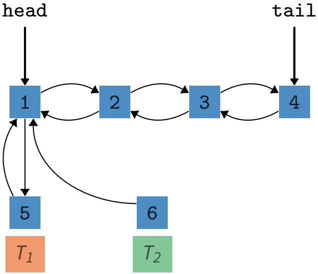
```
struct ele_t *new_ele = new ele_t;
new_ele->next = head;
head->prev = new_ele;
head = new_ele;
```

Why do we need synchronization?



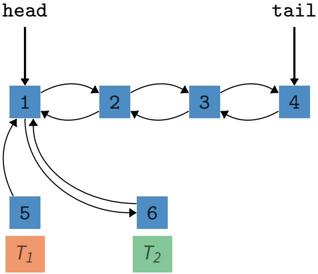
T_1	T_2	
1.	2.	`struct ele_t *new_ele = new ele_t;`
4.	3.	`new_ele->next = head;`
		`head->prev = new_ele;`
		`head = new_ele;`

Why do we need synchronization?



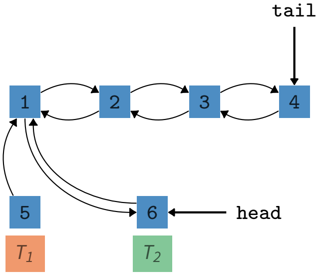
T_1	T_2	
1.	2.	`struct ele_t *new_ele = new ele_t;`
4.	3.	`new_ele->next = head;` ⬅ (T_2)
5.		`head->prev = new_ele;` ⬅ (T_1)
		`head = new_ele;`

Why do we need synchronization?



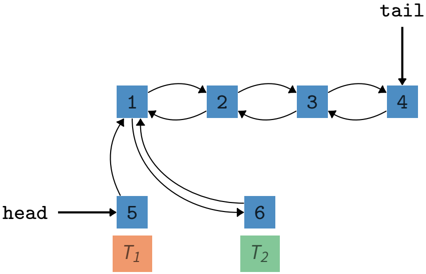
T_1	T_2	
1.	2.	`struct ele_t *new_ele = new ele_t;`
4.	3.	`new_ele->next = head;`
5.	6.	`head->prev = new_ele;`
		`head = new_ele;`

Why do we need synchronization?



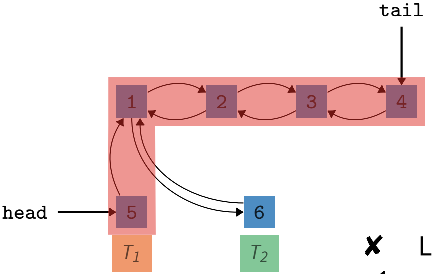
T_1	T_2	
1.	2.	`struct ele_t *new_ele = new ele_t;`
4.	3.	`new_ele->next = head;`
5.	6.	`head->prev = new_ele;` ←
	7.	`head = new_ele;` ←

Why do we need synchronization?



T_1	T_2	
1.	2.	`struct ele_t *new_ele = new ele_t;`
4.	3.	`new_ele->next = head;`
5.	6.	`head->prev = new_ele;`
8.	7.	`head = new_ele;`

Why do we need synchronization?

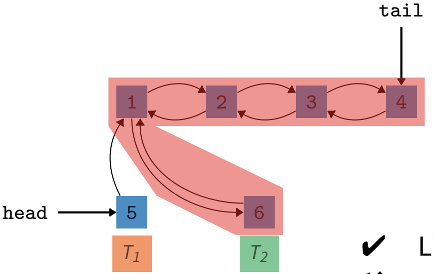


T_1	T_2	
1.	2.	`struct ele_t *new_ele = new ele_t;`
4.	3.	`new_ele->next = head;`
5.	6.	`head->prev = new_ele;`
8.	7.	`head = new_ele;`

✘ List structure correct

✔ `head` points to start of list

Why do we need synchronization?



T_1	T_2	
1.	2.	`struct ele_t *new_ele = new ele_t;`
4.	3.	`new_ele->next = head;`
5.	6.	`head->prev = new_ele;`
8.	7.	`head = new_ele;`

- ✔ List structure correct
- ✘ `head` points to start of list

Content

Basic Principles
Atomicity Assumption

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

Basic Principles

Atomicity Assumption

Parallel execution of A and B

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

Sequentiell execution, first A then B

Sequentiell execution, first B then A

Basic Principles

Atomicity Assumption

Parallel execution of A and B

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

Sequentiell execution, first A then B

Sequentiell execution, first B then A

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

Basic Principles

Atomicity Assumption

Parallel execution of A and B

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

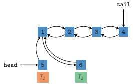
Sequentiell execution, first A then B

Sequentiell execution, first B then A

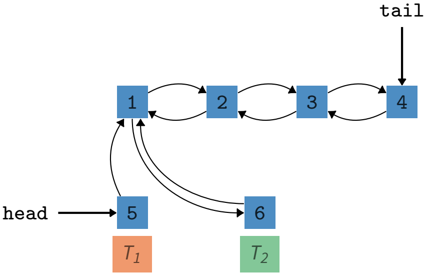
- Otherwise, the outcome of $A \parallel B$ is undefined
- Usually problematic for parallel *Read-Modify-Write* operations

Basic Principles

Mutual Exclusion

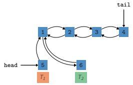


$A \parallel B$

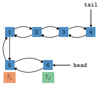



```
struct ele_t *new_ele = new ele_t;
new_ele->next = head;
head->prev = new_ele;
head = new_ele;
```

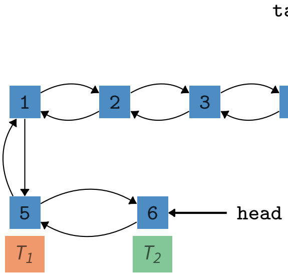
Basic Principles
Mutual Exclusion



A || B



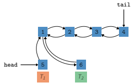
A;B



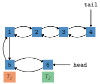

```
struct ele_t *new_ele = new ele_t;
new_ele->next = head;
head->prev = new_ele;
head = new_ele;
```

Basic Principles

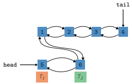
Mutual Exclusion



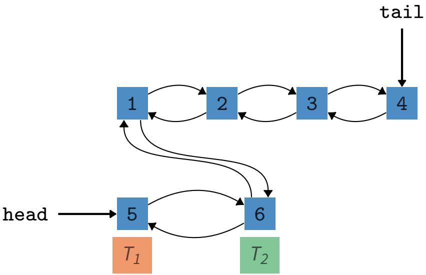
A || B



A;B



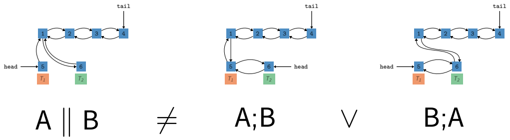
B;A




```
struct ele_t *new_ele = new ele_t;
new_ele->next = head;
head->prev = new_ele;
head = new_ele;
```

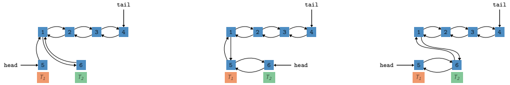
Basic Principles

Mutual Exclusion




```
struct ele_t *new_ele = new ele_t;
new_ele->next = head;
head->prev = new_ele;
head = new_ele;
```

Basic Principles
Mutual Exclusion



$$A \parallel B \neq A;B \lor B;A$$

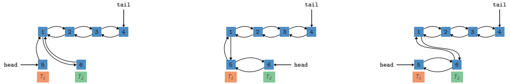
Need to ensure that only one thread at a time can execute the *Read-Modify-Write* operation.

$\Rightarrow$ **Mutual Exclusion**

```
struct ele_t *new_ele = new ele_t;
new_ele->next = head;
head->prev = new_ele;
head = new_ele;
```

Basic Principles

Mutual Exclusion



$A \parallel B \neq A;B \vee B;A$

```
struct ele_t *new_ele = new ele_t;
new_ele->next = head;
head->prev = new_ele;
head = new_ele;
```

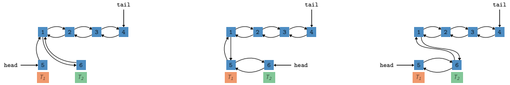
} CS (lines 2–4)

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

$\Rrightarrow$ **Mutual Exclusion**

Basic Principles

Mutual Exclusion



$A \parallel B \quad \neq \quad A;B \quad \vee \quad B;A$

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

$\Rrightarrow$ **Mutual Exclusion**

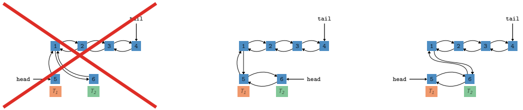
```
struct ele_t *new_ele = new ele_t;
new_ele->next = head;
head->prev = new_ele;
head = new_ele;
```

} CS

Should run with mutual exclusion!

Basic Principles

Mutual Exclusion



A || B = A;B ∨ B;A

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

⇛ **Mutual Exclusion**

```
struct ele_t *new_ele = new ele_t;
new_ele->next = head;
head->prev = new_ele;
head = new_ele;
```

} CS

Should run with mutual exclusion!

Basic Principles

Entersection & Leavesection

Simple protocol to establish mutual exclusion for critical sections.

```
struct ele_t *new_ele = new ele_t;
entersection();
new_ele->next = head;
head->prev = new_ele;
head = new_ele;
leavesection();
```

CS

```
void entersection() {
    while (!cs_free) wait();
    cs_free = false;
}
```

```
void leavesection() {
    cs_free = true;
    wake_next();
}
```

Basic Principles

Entersection & Leavesection

Simple protocol to establish mutual exclusion for critical sections.

```
struct ele_t *new_ele = new ele_t;
entersection();
new_ele->next = head;
head->prev = new_ele;
head = new_ele;
leavesection();
```

CS

```
void entersection() {
    while (!cs_free) wait();
    cs_free = false;
}
```

```
void leavesection() {
    cs_free = true;
    wake_next();
}
```

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.

```
struct ele_t *new_ele = new ele_t;
lock();
new_ele->next = head;
head->prev = new_ele;
head = new_ele;
unlock();
```

CS

```
void lock() {
    while (!cs_free) wait();
    cs_free = false;
}
```

```
void unlock() {
    cs_free = true;
    wake_next();
}
```

Rules:

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

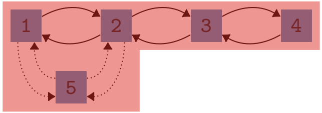
Basic Principles

Coarse Grained vs. Fine Grained

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

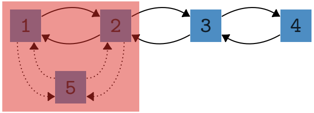
- Length of critical sections are important for scalability $\rightarrow$ Amdahl's Law

Coarse Grained



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

Fine Grained



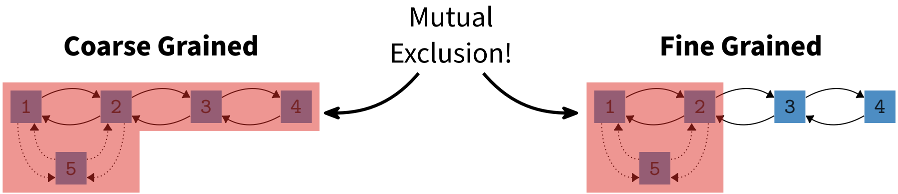
- \+ Better scalability (parallel operations possible)
- More difficult to implement
- Deadlocks may happen

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



Coarse Grained	Fine Grained
– Worse scalability (no parallel operations)	+ Better scalability (parallel operations possible)
+ Easier to implement	– More difficult to implement
	– Deadlocks may happen

Implementing Entersection & Leavesection

Peterson Algorithm

```
bool free[2] = {true, true};
int turn = 0;

void lock() {
    int other = 1 - TID;          /* TID: ID of the current thread ({0,1}) */
    free[TID] = false;
    turn = other;
    while (!free[other] && turn == other) {}
}

void unlock() {
    free[TID] = true;
}
```

- 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!)

```
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

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

Implementing Entersection & Leavesection

Naive Spinlock (Wrong!)

```
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

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

Implementing Entersection & Leavesection

Naive Spinlock (Wrong!)

```c
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

```c
/* do other stuff */
lock();
/* critical section */
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!)

```c
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

```c
/* do other stuff */
lock();
/* critical section */
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!)

```
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

```
/* do other stuff */
lock();
/* critical section */
unlock();
```

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

Implementing Entersection & Leavesection

Naive Spinlock (Wrong!)

```
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

```
/* do other stuff */
lock();
/* critical section */
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!)

```
int l = 0;

void lock() {                    l == 0
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

```
/* do other stuff */
lock();
/* critical section */
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!)

```
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

l == 0

```
/* do other stuff */
lock();
/* critical section */
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!)

l == 0

```
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

```
/* do other stuff */
lock();
/* critical section */
unlock();
```

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

Implementing Entersection & Leavesection

Naive Spinlock (Wrong!)

```
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

l == 0

```
/* do other stuff */
lock();
/* critical section */
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!)

```
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

l == 0

l == 0

```
/* do other stuff */
lock();
/* critical section */
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!)

```
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

l == 0

```
/* do other stuff */
lock();
/* critical section */
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!)

```c
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

```c
/* do other stuff */
lock();
/* critical section */
unlock();
```

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

Implementing Entersection & Leavesection

Naive Spinlock (Wrong!)

```
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

```
/* do other stuff */
lock();
/* critical section */
unlock();
```

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

Implementing Entersection & Leavesection

Naive Spinlock (Wrong!)

```
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

```
/* do other stuff */
lock();
/* critical section */
unlock();
```

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

Implementing Entersection & Leavesection

Naive Spinlock (Wrong!)

```
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

```
/* do other stuff */
lock();
/* critical section */
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!)

```
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

```
/* do other stuff */
lock();
/* critical section */
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!)

```
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

```
/* do other stuff */
lock();
/* critical section */ ⇐ ϟ
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!)

```
int l = 0;

void lock() {
    while (l == 1) {}
    l = 1;
}

void unlock() {
    l = 0;
}
```

CS (internal)

```
/* do other stuff */
lock();
/* critical section */
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

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

Core 2

```
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

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

Core 2

```
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

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

Core 2

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

Core 1 **Memory** **Core 2**

x:0

Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \lor B;A$$

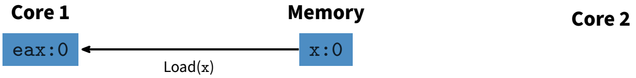
- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

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

Core 2

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



Atomicity on Hardware

Atomicity Assumption on Hardware

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

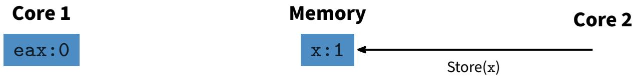
- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

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

Core 2

```
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

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

Core 2

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

Core 1	Memory	Core 2
eax:0	x:1	

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

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

Comparison with 0 although x == 1 already

Core 2

```
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

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

Core 2

```
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

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

Core 2

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

Core 1 **Memory** **Core 2**

x:0

Atomicity on Hardware

Atomicity Assumption on Hardware

$$A \parallel B = A;B \lor B;A$$

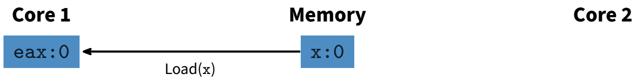
- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

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

Core 2

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



Atomicity on Hardware

Atomicity Assumption on Hardware

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

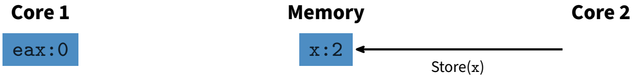
- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

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

Core 2

```
store $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

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

Core 2

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

Core 1	Memory	Core 2
eax:1	x:2	

Atomicity on Hardware

Atomicity Assumption on Hardware

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

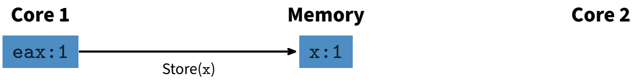
- Always guaranteed for single-core systems
- Usually not guaranteed for multi-core systems
- Especially problematic for *Read-Modify-Write* instructions

Core 1

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

Core 2

```
store $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

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

Core 2

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

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

Core 1	**Memory**	**Core 2**
`eax:1`	`x:1` ϟ	

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

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

Core 2

```
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

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

Core 2

```
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

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

Core 2

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

Core 1 | **Memory** | **Core 2**

Memory: x:0

Core 1: x:0 → S

Core 2: x:0 → S

Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

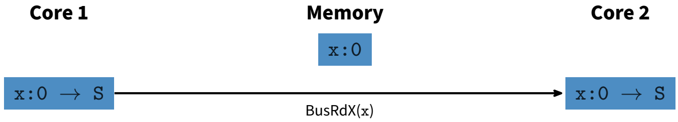
Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

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

Core 2

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



Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

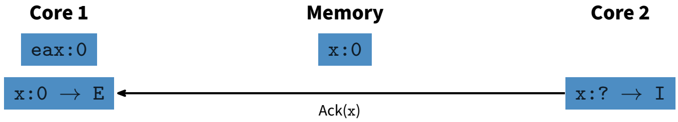
Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

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

Core 2

```
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

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

Core 2

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

Core 1	Memory	Core 2
eax:1	x:0	
x:0 → E		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

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

Core 2

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

Core 1	Memory	Core 2
eax:1	x:0	
x:0 → E		x:? → I

Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

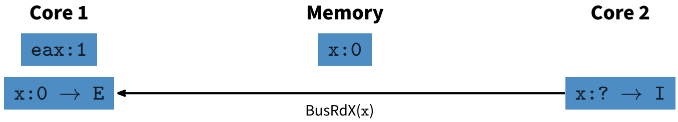
Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

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

Core 2

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



Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

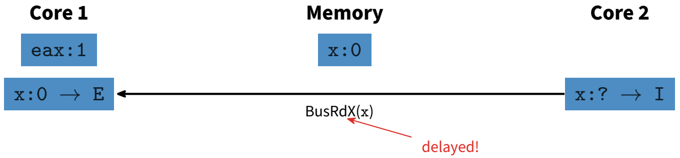
Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

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

Core 2

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



Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

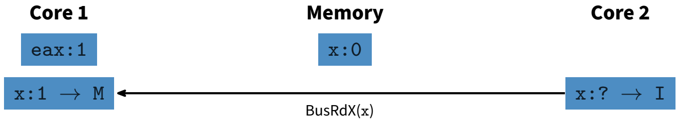
Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

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

Core 2

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



Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

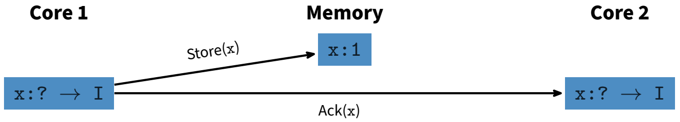
Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

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

Core 2

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



Atomicity on Hardware

Atomic Instructions with Cache Lock

General Idea

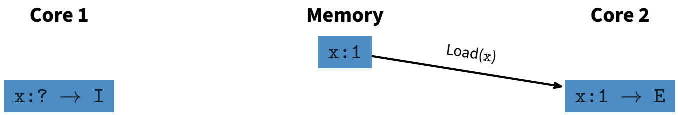
Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

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

Core 2

```
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

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

Core 2

```
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

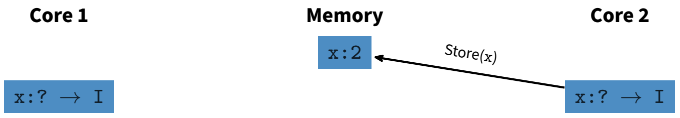
Delay all cache coherency traffic (*snoop messages*) until all memory accesses of an *atomic* Read-Modify-Write instruction are finished.

Core 1

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

Core 2

```
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

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

Core 2

```
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

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

Core 2

```
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

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

Core 2

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

Core 1	Memory	Core 2
	x:0	
x:0 → S		x:0 → S

Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

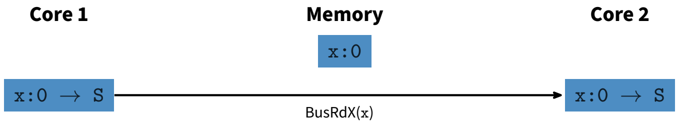
Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

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

Core 2

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



Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

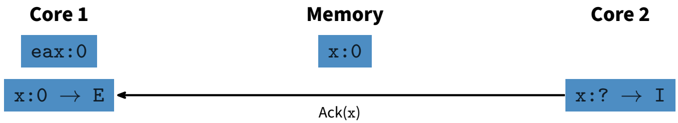
Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

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

Core 2

```
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

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

Core 2

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

Core 1	Memory	Core 2
eax:1	x:0	
x:0 → E		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

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

Core 2

```
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

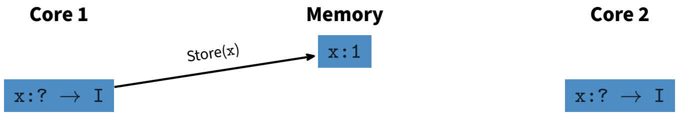
Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

```
ldrex [x] %eax;
add $1 %eax;
strex %eax [x]; ⬅ ✔ Success
```

Core 2

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



Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

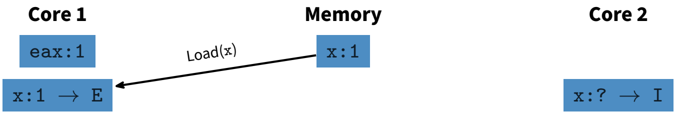
Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

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

Core 2

```
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

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

Core 2

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

Core 1	Memory	Core 2
eax:2	x:1	
x:1 → E		x:? → I

Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

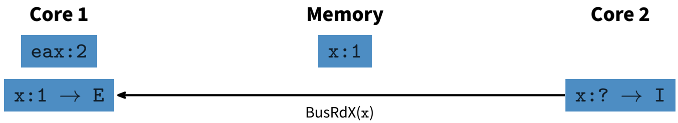
Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

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

Core 2

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



Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

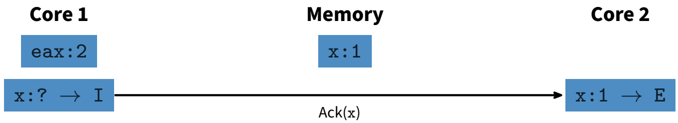
Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

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

Core 2

```
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

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

Core 2

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

Core 1	Memory	Core 2
eax:2	x:1	
x:? → I		x:2 → M

Atomicity on Hardware

Atomic Instructions with Observe Cache

General Idea

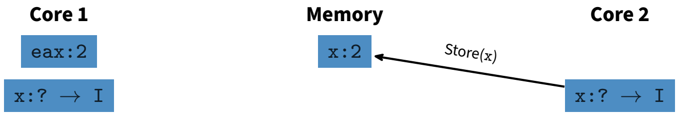
Install a watchdog when the *atomic* instruction references the memory location and check for parallel accesses before storing to the memory location again. In case of parallel accesses, abort the store and retry the whole *atomic* instruction.

Core 1

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

Core 2

```
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

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

Core 2

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

Core 1	Memory	Core 2
eax:2	x: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

```
ldrex [x] %eax;
add $1 %eax;
strex %eax [x]; ← ✘ Abort
```

Core 2

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

Core 1	Memory	Core 2
eax:2	x:2	
x:? → I		

Atomicity on Hardware

Examples of Atomic Instructions

- swap(mem1, mem2)

```
mov [mem1] %eax;
mov [mem2] [mem1];
mov %eax [mem2];
```

- xadd(mem, reg)

```
mov [mem] %eax;
add [mem] reg;
return %eax;
```

- cas(mem, expected, desired)

```
cmp [mem] [expected];
jne fail;
mov [desired] [mem];
return true;
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

```c
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

- 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

```
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Test & Set Lock

Overhead

```c
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```c
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1   **Core 2**   **Core 3**

l:? → I   l:? → I   l:? → I

Test & Set Lock

Overhead

```
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:? → I	l:? → I	l:? → I

Test & Set Lock

Overhead

```
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:? → I	l:? → I	l:? → I

Test & Set Lock

Overhead

```
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:1 → M	l:? → I	l:? → I

Test & Set Lock

Overhead

```
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:1 → M	l:? → I	l:? → I

Test & Set Lock

Overhead

```c
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```c
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:1 → M	l:? → I	l:? → I

Test & Set Lock

Overhead

```c
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```c
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:1 → M	l:? → I	l:? → I

Test & Set Lock

Overhead

```
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

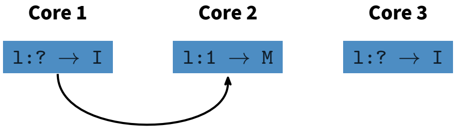
Core 1	**Core 2**	**Core 3**
l:1 → M	l:? → I	l:? → I

Test & Set Lock

Overhead

```
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```



Test & Set Lock

Overhead

```
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:? → I	l:1 → M	l:? → I

Test & Set Lock

Overhead

```c
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```c
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:? → I	l:1 → M	l:? → I

Test & Set Lock

Overhead

```
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1

l:? → I

Core 2

l:1 → M

Core 3

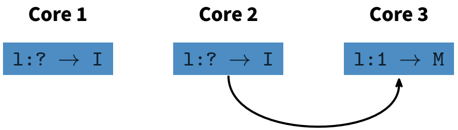
l:? → I

Test & Set Lock

Overhead

```c
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```c
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```



Test & Set Lock

Overhead

```
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:? → I	l:? → I	l:1 → M

Test & Set Lock

Overhead

```
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

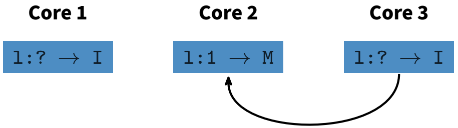
Core 1	Core 2	Core 3
l:? → I	l:? → I	l:1 → M

Test & Set Lock

Overhead

```
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```



Test & Set Lock

Overhead

```c
struct ts_lock_t {
    volatile int lock;
};

void lock(ts_lock_t *l) {
    do {
        int tmp = 1;
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(ts_lock_t *l) {
    l->lock = 0;
}
```

```c
struct ts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:? → I	l:1 → M	l:? → I

Test & Test & Set Lock

```
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

- 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

```c
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

```c
struct tts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Test & Test & Set Lock

Overhead

```
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

```
struct tts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:? → I	l:? → I	l:? → I

Test & Test & Set Lock
Overhead

```
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

```
struct tts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:? → I	l:? → I	l:? → I

Test & Test & Set Lock

Overhead

```
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

```
struct tts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:? → I	l:? → I	l:? → I

Test & Test & Set Lock

Overhead

```
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

```
struct tts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:0 → E	l:? → I	l:? → I

Test & Test & Set Lock

Overhead

```
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

```
struct tts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:1 → M	l:? → I	l:? → I

Test & Test & Set Lock

Overhead

```c
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

```c
struct tts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:1 → M	l:? → I	l:? → I

Test & Test & Set Lock

Overhead

```
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

```
struct tts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:1 → M	l:? → I	l:? → I

Test & Test & Set Lock

Overhead

```c
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

```c
struct tts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:1 → M	l:? → I	l:? → I

Test & Test & Set Lock

Overhead

```
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

```
struct tts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

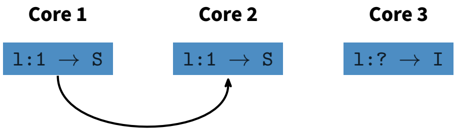
Core 1	Core 2	Core 3
l:1 → M	l:? → I	l:? → I

Test & Test & Set Lock

Overhead

```
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

```
struct tts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```



Test & Test & Set Lock

Overhead

```
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

```
struct tts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:1 → S	l:1 → S	l:? → I

Test & Test & Set Lock

Overhead

```c
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

```c
struct tts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

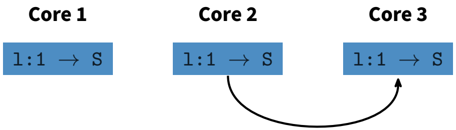
Core 1	Core 2	Core 3
l:1 → S	l:1 → S	l:? → I

Test & Test & Set Lock

Overhead

```c
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

```c
struct tts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```



Test & Test & Set Lock

Overhead

```
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

```
struct tts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:1 → S	l:1 → S	l:1 → S

Test & Test & Set Lock

Overhead

```
struct tts_lock_t {
    volatile int lock;
};

void lock(tts_lock_t *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}

void unlock(tts_lock_t *l) {
    l->lock = 0;
}
```

```
struct tts_lock_t l;

void thread_fn(void) {
    /* Other stuff */
    lock(&l);
    /* CS */
    unlock(&l);
}
```

Core 1	Core 2	Core 3
l:1 → S	l:1 → S	l:1 → S

Test & Test & Set Lock

Fairness

```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```

T_1 T_2 T_3

Test & Test & Set Lock

Fairness

```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```

Test & Test & Set Lock

Fairness

```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```

T_1 T_2 T_3

Test & Test & Set Lock

Fairness

```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```

T_1 T_2 T_3

Test & Test & Set Lock

Fairness

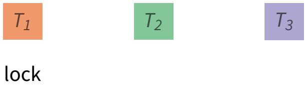
```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```

T_1 T_2 T_3

Test & Test & Set Lock

Fairness

```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Test & Test & Set Lock

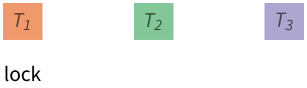
Fairness

```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```

Test & Test & Set Lock

Fairness

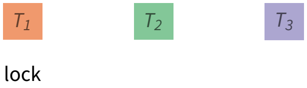
```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Test & Test & Set Lock

Fairness

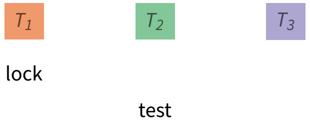
```c
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Test & Test & Set Lock

Fairness

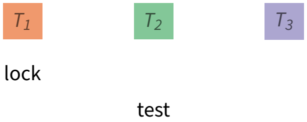
```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Test & Test & Set Lock

Fairness

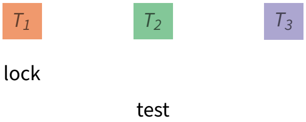
```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Test & Test & Set Lock

Fairness

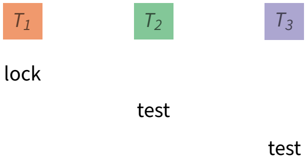
```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Test & Test & Set Lock

Fairness

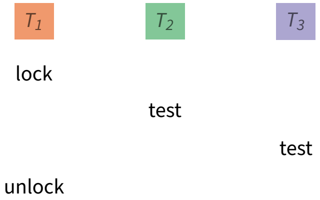
```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Test & Test & Set Lock

Fairness

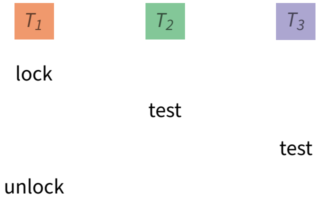
```c
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Test & Test & Set Lock

Fairness

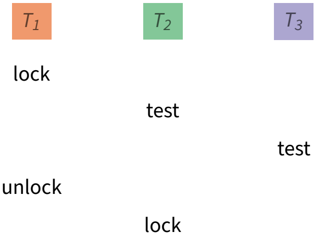
```c
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Test & Test & Set Lock

Fairness

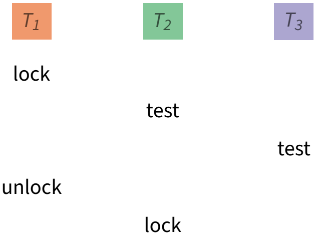
```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Test & Test & Set Lock

Fairness

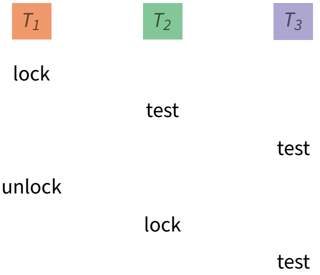
```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Test & Test & Set Lock

Fairness

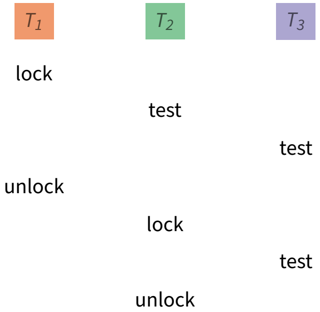
```c
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Test & Test & Set Lock

Fairness

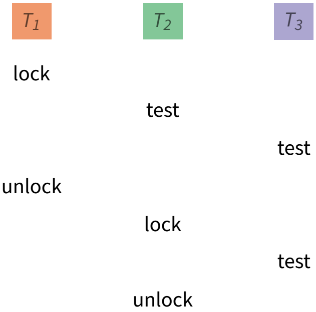
```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Test & Test & Set Lock

Fairness

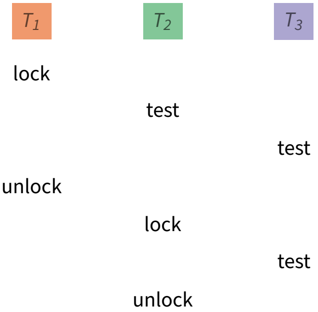
```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Test & Test & Set Lock

Fairness

```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Test & Test & Set Lock

Fairness

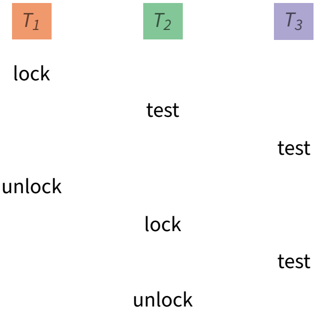
```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```

T_1	T_2	T_3
lock		
	test	
		test
unlock		
	lock	
		test
	unlock	

Test & Test & Set Lock

Fairness

```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Test & Test & Set Lock

Fairness

```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```

T_1	T_2	T_3
lock		
	test	
		test
unlock		
	lock	
		test
	unlock	
lock		

Test & Test & Set Lock

Fairness

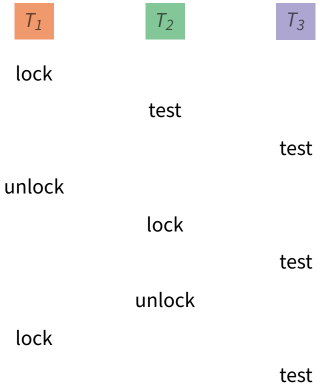
```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```

T_1	T_2	T_3
lock		
	test	
		test
unlock		
	lock	
		test
	unlock	
lock		

Test & Test & Set Lock

Fairness

```
struct tts_lock l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

void lock(tts_lock *l) {
    do {
        int tmp = 1;
        do {} while (l->lock == 1);
        swap(&(l->lock), &(tmp));
    } while (tmp == 1);
}
```



Ticket Lock

```
struct ticket_lock_t {
    int next;
    volatile int current;
};

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

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

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

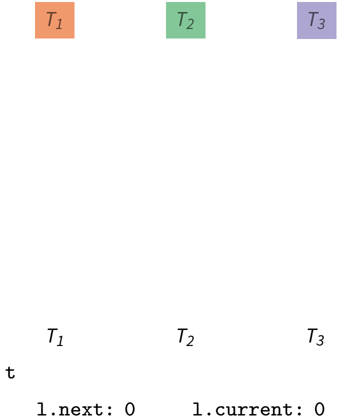
But

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

Ticket Lock

Fairness

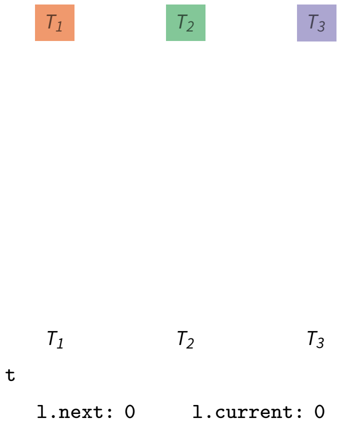
```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



Ticket Lock

Fairness

```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



Ticket Lock

Fairness

```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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

T_1 T_2 T_3

lock

T_1 T_2 T_3

t

`l.next: 0` `l.current: 0`

Ticket Lock

Fairness

```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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

T_1	T_2	T_3
lock		

	T_1	T_2	T_3
`t`	0		

`l.next: 1`   `l.current: 0`

Ticket Lock

Fairness

```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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

T_1 T_2 T_3

lock

	T_1	T_2	T_3
t	0		

l.next: 1 l.current: 0

Ticket Lock

Fairness

```c
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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

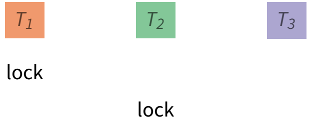
	T_1	T_2	T_3
t	0		

`l.next: 1` `l.current: 0`

Ticket Lock

Fairness

```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



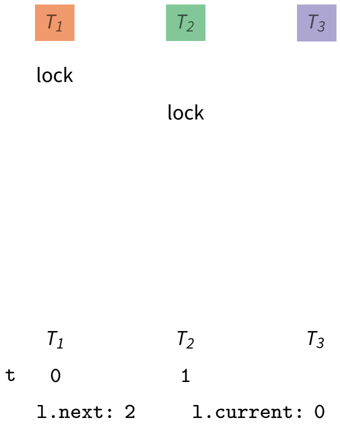
	T_1	T_2	T_3
t	0		

`l.next: 1`   `l.current: 0`

Ticket Lock

Fairness

```c
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



Ticket Lock

Fairness

```c
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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

T_1 T_2 T_3

lock

lock

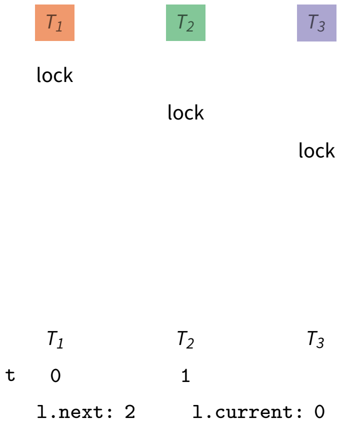
	T_1	T_2	T_3
t	0	1	

`l.next: 2` `l.current: 0`

Ticket Lock

Fairness

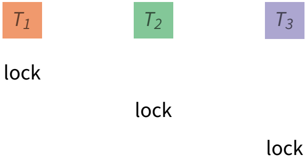
```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



Ticket Lock

Fairness

```c
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



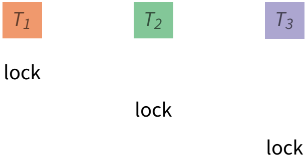
	T_1	T_2	T_3
t	0	1	2

l.next: 3   l.current: 0

Ticket Lock

Fairness

```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



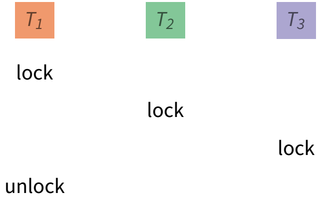
	T_1	T_2	T_3
t	0	1	2

`l.next: 3` `l.current: 0`

Ticket Lock

Fairness

```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



	T_1	T_2	T_3
t	0	1	2

`l.next: 3` `l.current: 0`

Ticket Lock

Fairness

```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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

T_1	T_2	T_3
lock		
	lock	
		lock
unlock		

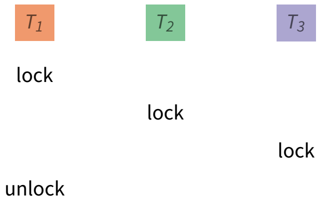
	T_1	T_2	T_3
`t`		1	2

`l.next: 3` `l.current: 1`

Ticket Lock

Fairness

```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



	T_1	T_2	T_3
t		1	2

`l.next: 3` `l.current: 1`

Ticket Lock

Fairness

```c
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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

T_1	T_2	T_3
lock		
	lock	
		lock
unlock		
lock		

	T_1	T_2	T_3
t		1	2

l.next: 3 l.current: 1

Ticket Lock

Fairness

```c
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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

T_1	T_2	T_3
lock		
	lock	
		lock
unlock		
lock		

	T_1	T_2	T_3
t	3	1	2

`l.next: 4` `l.current: 1`

Ticket Lock

Fairness

```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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

T_1 T_2 T_3

T_1: lock

T_2: lock

T_3: lock

T_1: unlock

T_1: lock

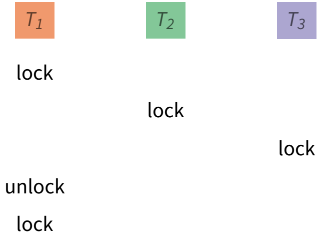
	T_1	T_2	T_3
t	3	1	2

`l.next: 4` `l.current: 1`

Ticket Lock

Fairness

```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



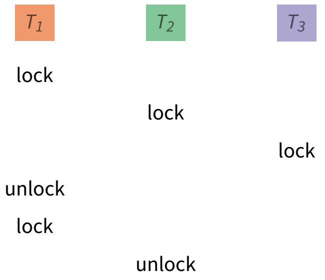
	T_1	T_2	T_3
t	3	1	2

`l.next: 4` `l.current: 1`

Ticket Lock

Fairness

```c
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



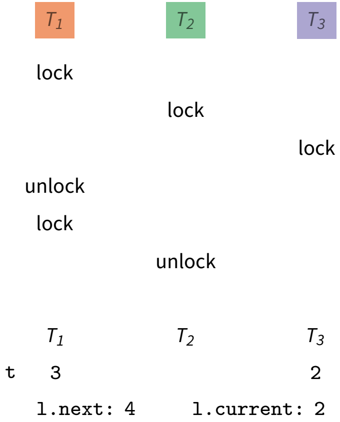
	T_1	T_2	T_3
t	3	1	2

`l.next: 4`   `l.current: 1`

Ticket Lock

Fairness

```c
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



Ticket Lock

Fairness

```c
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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

T_1	T_2	T_3
lock		
	lock	
		lock
unlock		
lock		
	unlock	

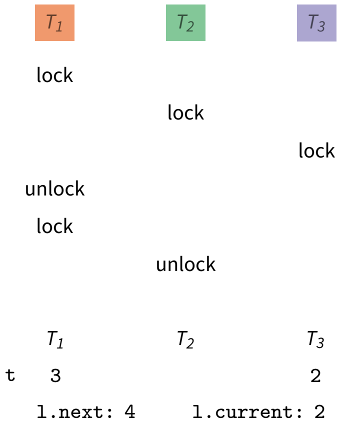
	T_1	T_2	T_3
t	3		2

l.next: 4 l.current: 2

Ticket Lock

Fairness

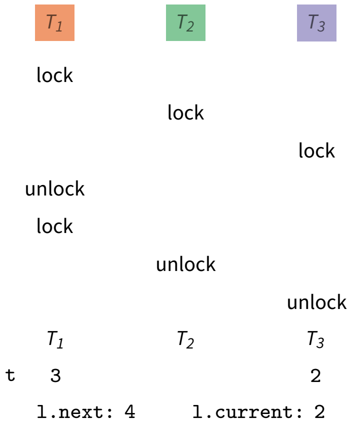
```c
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



Ticket Lock

Fairness

```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



Ticket Lock

Fairness

```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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

T_1 T_2 T_3

lock

lock

lock

unlock

lock

unlock

unlock

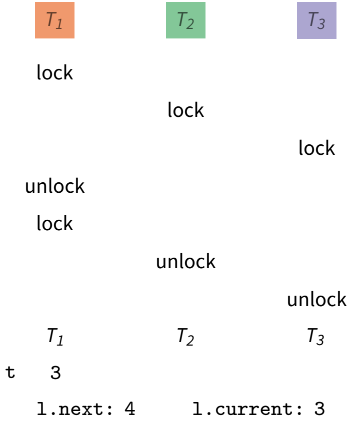
	T_1	T_2	T_3
t	3		

`l.next: 4` `l.current: 3`

Ticket Lock

Fairness

```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



Ticket Lock

Fairness

```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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

T_1	T_2	T_3
lock		
	lock	
		lock
unlock		
lock		
	unlock	
		unlock

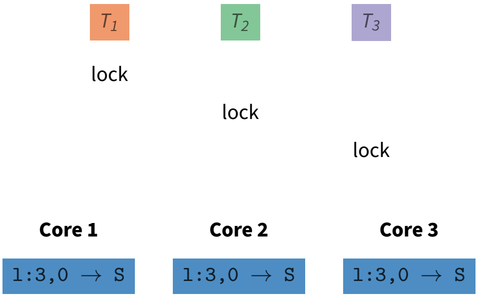
	T_1	T_2	T_3
t	3		

`l.next: 4` `l.current: 3`

Ticket Lock

Overhead

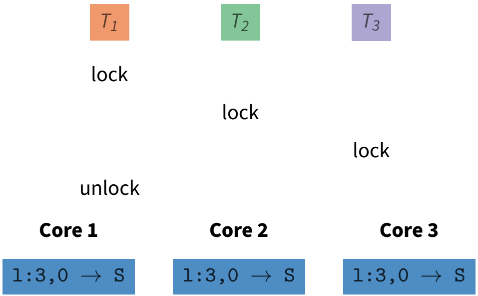
```c
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



Ticket Lock

Overhead

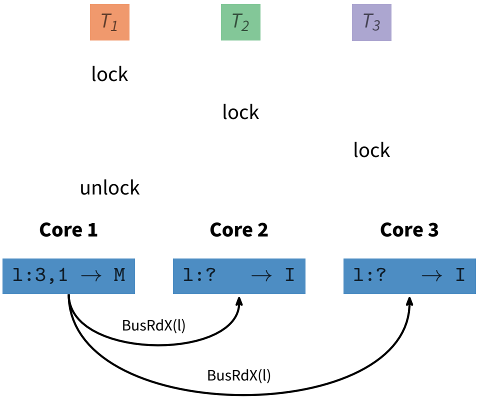
```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



Ticket Lock

Overhead

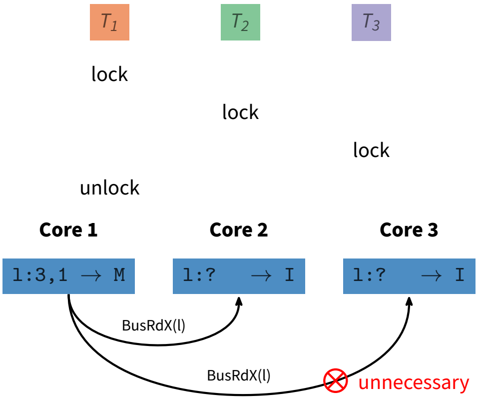
```c
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



Ticket Lock

Overhead

```
struct ticket_lock_t l;

void thread_fn(void) {
    while (true) {
        /* Other stuff */
        lock(&l);
        /* CS */
        unlock(&l);
    }
}

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

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



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

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

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

```
void insert(ele_t *new_ele, ele_t *prev) {
    do {
        new_ele->next = prev->next;
        new_ele->prev = prev;
    } while (!dcas(&(prev->next), &(prev->next->prev),
                   new_ele->next, new_ele->prev,
                   new_ele, new_ele));
}
```

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)

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

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

But:

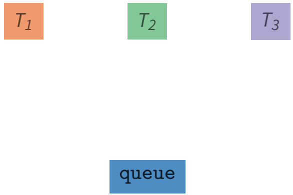
- Difficult to implement correctly

MCS Lock

Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};

struct mcs_lock_t {
    mcs_node_t *queue;
};

void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}

void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```

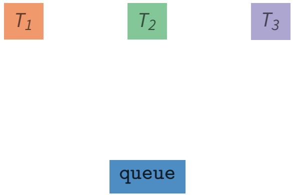


MCS Lock

Fairness & Overhead

```c
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
struct mcs_lock_t {
    mcs_node_t *queue;
};
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```c
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```

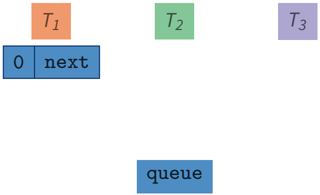


MCS Lock

Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
struct mcs_lock_t {
    mcs_node_t *queue;
};
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

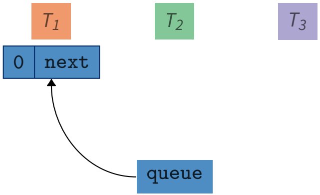
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

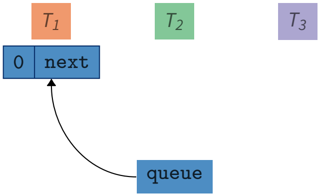
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```

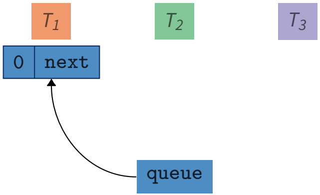


MCS Lock

Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};

struct mcs_lock_t {
    mcs_node_t *queue;
};

void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}

void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

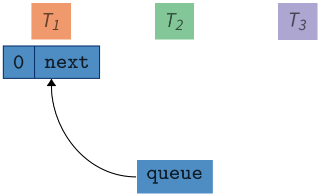
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

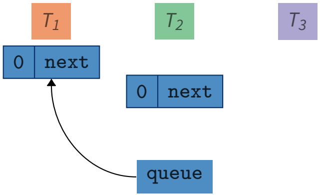
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```

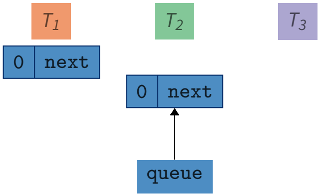


MCS Lock

Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
struct mcs_lock_t {
    mcs_node_t *queue;
};
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

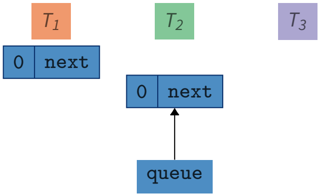
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

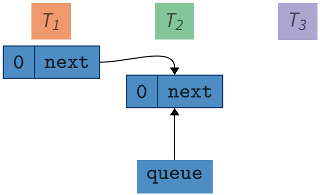
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

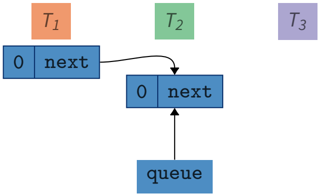
Fairness & Overhead

```c
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```c
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```c
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```c
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

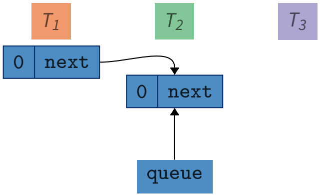
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

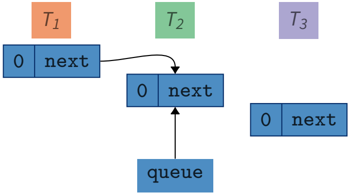
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

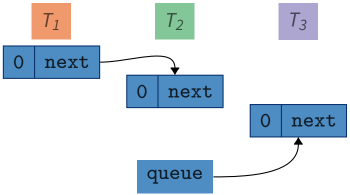
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

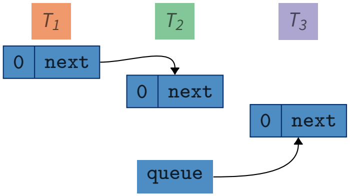
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

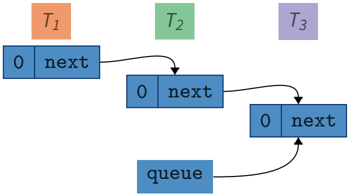
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

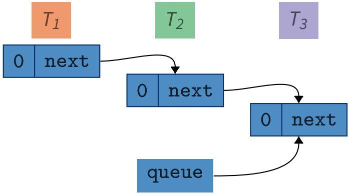
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

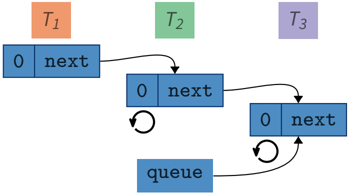
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

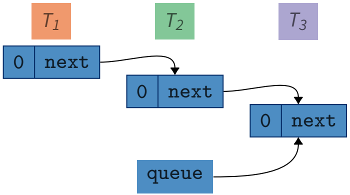
Fairness & Overhead

```c
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```c
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```c
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```c
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

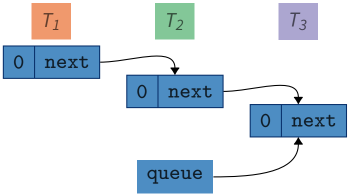
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

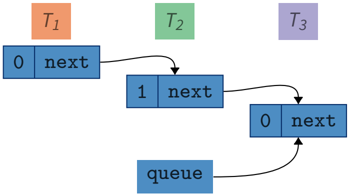
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

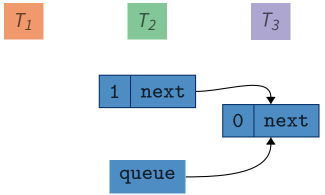
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

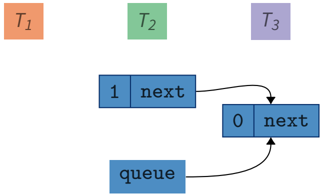
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

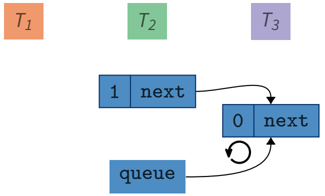
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

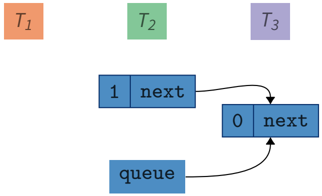
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```

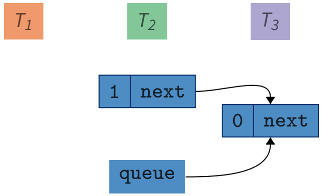


MCS Lock

Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
struct mcs_lock_t {
    mcs_node_t *queue;
};
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```

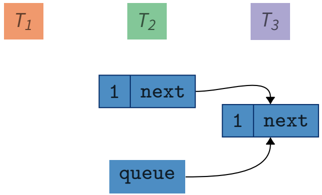


MCS Lock

Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
struct mcs_lock_t {
    mcs_node_t *queue;
};
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

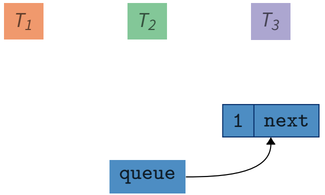
Fairness & Overhead

```c
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```c
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```c
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```c
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

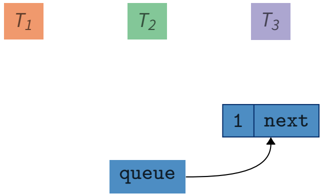
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

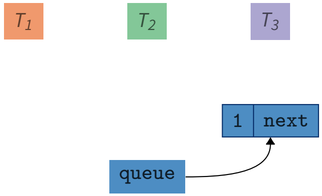
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

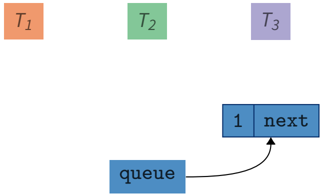
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

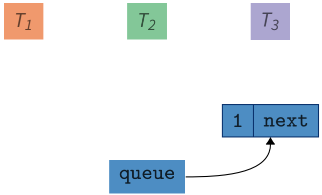
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

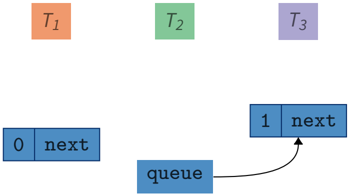
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

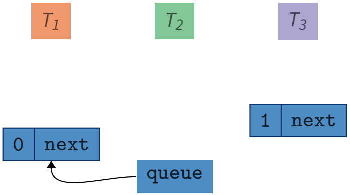
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

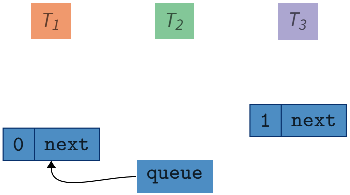
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

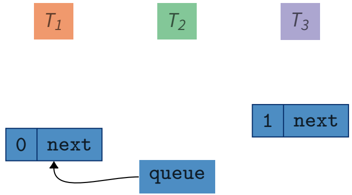
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

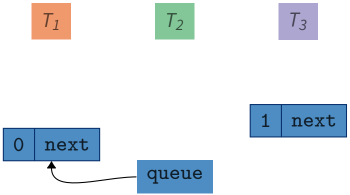
Fairness & Overhead

```c
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```c
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```c
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```c
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

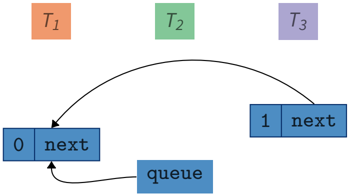
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

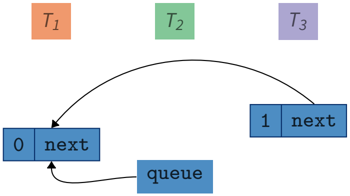
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

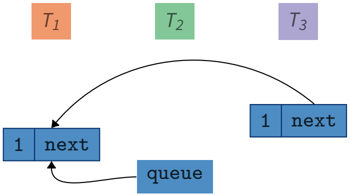
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

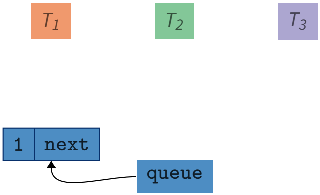
Fairness & Overhead

```c
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```c
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```c
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```c
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

Fairness & Overhead

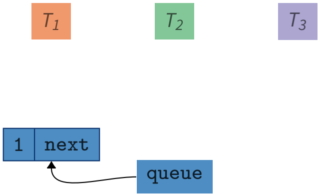
```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
```

```
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```

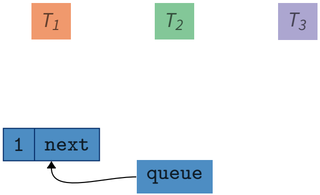


MCS Lock

Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
struct mcs_lock_t {
    mcs_node_t *queue;
};
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

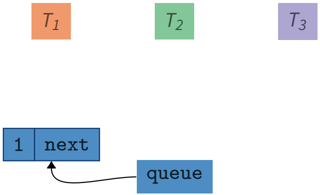
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```



MCS Lock

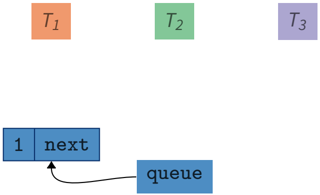
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};
```

```
struct mcs_lock_t {
    mcs_node_t *queue;
};
```

```
void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}
void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```

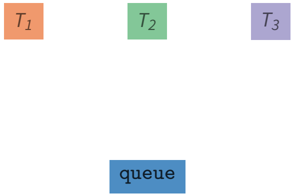


MCS Lock

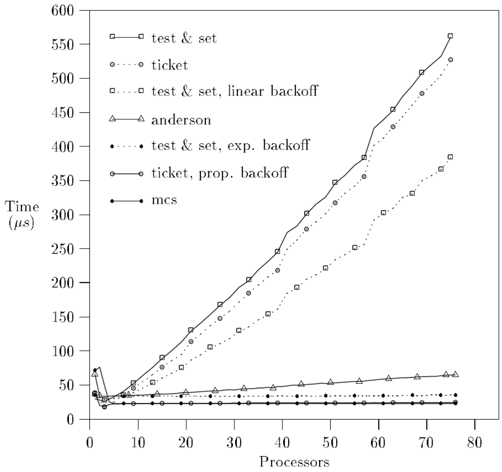
Fairness & Overhead

```
struct mcs_node_t {
    mcs_node_t *next;
    bool free;
};

struct mcs_lock_t {
    mcs_node_t *queue;
};

void lock(mcs_lock_t *l, mcs_node_t *cur) {
    cur->next = NULL; cur->free = false;
    auto prev = fetch_and_store(&(l->queue), cur);
    if (prev) {
        prev->next = cur;
        do {} while (!cur->free);
    }
}

void unlock(mcs_lock_t *l, mcs_node_t *cur) {
    if (!cur->next) {
        if (cas(&(l->queue), cur, NULL)) return;
        do {} while (!cur->next);
    }
    cur->next->free = true;
}
```

```
struct mcs_lock_t l;

void thread_fn(void) {
    struct mcs_node_t n;
    while (true) {
        /* Other stuff */
        lock(&l, &n);
        /* CS */
        unlock(&l, &n);
    }
}
```

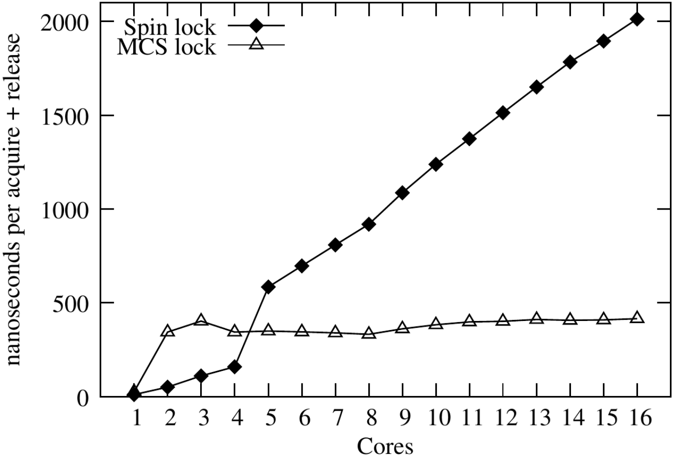


MCS Lock
Performance



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)

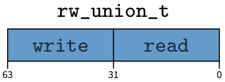
Different levels of fairness can be implemented:

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

Reader Writer Lock

Fair Ticket Reader Writer Lock

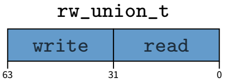
```
struct rw_lock_t {
    rw_union_t current;
    rw_union_t next;
};
void lock_read(rw_lock_t *l) {
    auto t = xadd(&(l->next), 1);
    do {} while (l->current.write != t.write);
}
void lock_write(rw_lock_t *l) {
    auto t = xadd(&(l->next.write), 1);
    do {} while (l->current != t);
}
void unlock_read(rw_lock_t *l) {
    xadd(&(l->current.read), 1);
}
void unlock_write(rw_lock_t *l) {
    l->current.write++;
}
```



Reader Writer Lock

Fair Ticket Reader Writer Lock

```
struct rw_lock_t {
    rw_union_t current;
    rw_union_t next;
};

void lock_read(rw_lock_t *l) {
    auto t = xadd(&(l->next), 1);
    do {} while (l->current.write != t.write);
}
void lock_write(rw_lock_t *l) {
    auto t = xadd(&(l->next.write), 1);
    do {} while (l->current != t);
}

void unlock_read(rw_lock_t *l) {
    xadd(&(l->current.read), 1);
}
void unlock_write(rw_lock_t *l) {
    l->current.write++;
}
```



T_1 T_2 T_3

T_1 T_2 T_3

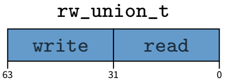
t

l.next: 0|0 l.current: 0|0

Reader Writer Lock

Fair Ticket Reader Writer Lock

```
struct rw_lock_t {
    rw_union_t current;
    rw_union_t next;
};

void lock_read(rw_lock_t *l) {
    auto t = xadd(&(l->next), 1);
    do {} while (l->current.write != t.write);
}
void lock_write(rw_lock_t *l) {
    auto t = xadd(&(l->next.write), 1);
    do {} while (l->current != t);
}

void unlock_read(rw_lock_t *l) {
    xadd(&(l->current.read), 1);
}
void unlock_write(rw_lock_t *l) {
    l->current.write++;
}
```



T_1	T_2	T_3
/* CS */		
read		

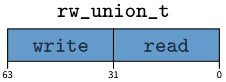
	T_1	T_2	T_3
t	0|0		

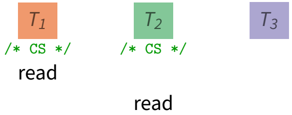
l.next: 0|1 l.current: 0|0

Reader Writer Lock

Fair Ticket Reader Writer Lock

```
struct rw_lock_t {
    rw_union_t current;
    rw_union_t next;
};
void lock_read(rw_lock_t *l) {
    auto t = xadd(&(l->next), 1);
    do {} while (l->current.write != t.write);
}
void lock_write(rw_lock_t *l) {
    auto t = xadd(&(l->next.write), 1);
    do {} while (l->current != t);
}
void unlock_read(rw_lock_t *l) {
    xadd(&(l->current.read), 1);
}
void unlock_write(rw_lock_t *l) {
    l->current.write++;
}
```





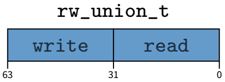
	T_1	T_2	T_3
t	0|0	0|1	

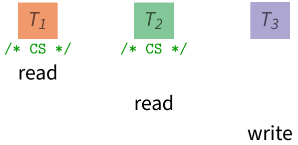
l.next: 0|2 l.current: 0|0

Reader Writer Lock

Fair Ticket Reader Writer Lock

```
struct rw_lock_t {
    rw_union_t current;
    rw_union_t next;
};

void lock_read(rw_lock_t *l) {
    auto t = xadd(&(l->next), 1);
    do {} while (l->current.write != t.write);
}
void lock_write(rw_lock_t *l) {
    auto t = xadd(&(l->next.write), 1);
    do {} while (l->current != t);
}

void unlock_read(rw_lock_t *l) {
    xadd(&(l->current.read), 1);
}
void unlock_write(rw_lock_t *l) {
    l->current.write++;
}
```





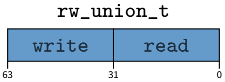
	T_1	T_2	T_3
t	0|0	0|1	0|2

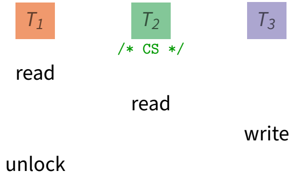
`l.next: 1|2` `l.current: 0|0`

Reader Writer Lock

Fair Ticket Reader Writer Lock

```c
struct rw_lock_t {
    rw_union_t current;
    rw_union_t next;
};

void lock_read(rw_lock_t *l) {
    auto t = xadd(&(l->next), 1);
    do {} while (l->current.write != t.write);
}
void lock_write(rw_lock_t *l) {
    auto t = xadd(&(l->next.write), 1);
    do {} while (l->current != t);
}

void unlock_read(rw_lock_t *l) {
    xadd(&(l->current.read), 1);
}
void unlock_write(rw_lock_t *l) {
    l->current.write++;
}
```





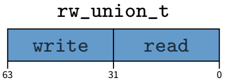
	T_1	T_2	T_3
t		0|1	0|2

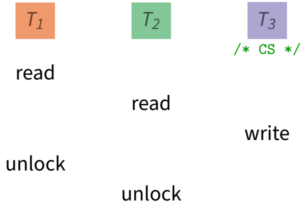
l.next: 1|2   l.current: 0|1

Reader Writer Lock

Fair Ticket Reader Writer Lock

```
struct rw_lock_t {
    rw_union_t current;
    rw_union_t next;
};
void lock_read(rw_lock_t *l) {
    auto t = xadd(&(l->next), 1);
    do {} while (l->current.write != t.write);
}
void lock_write(rw_lock_t *l) {
    auto t = xadd(&(l->next.write), 1);
    do {} while (l->current != t);
}
void unlock_read(rw_lock_t *l) {
    xadd(&(l->current.read), 1);
}
void unlock_write(rw_lock_t *l) {
    l->current.write++;
}
```





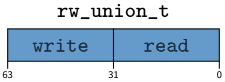
	T_1	T_2	T_3
t			0|2

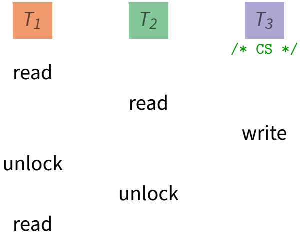
`l.next: 1|2` `l.current: 0|2`

Reader Writer Lock

Fair Ticket Reader Writer Lock

```
struct rw_lock_t {
    rw_union_t current;
    rw_union_t next;
};

void lock_read(rw_lock_t *l) {
    auto t = xadd(&(l->next), 1);
    do {} while (l->current.write != t.write);
}
void lock_write(rw_lock_t *l) {
    auto t = xadd(&(l->next.write), 1);
    do {} while (l->current != t);
}

void unlock_read(rw_lock_t *l) {
    xadd(&(l->current.read), 1);
}
void unlock_write(rw_lock_t *l) {
    l->current.write++;
}
```





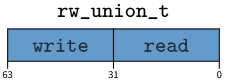
	T_1	T_2	T_3
t	1|2		0|2

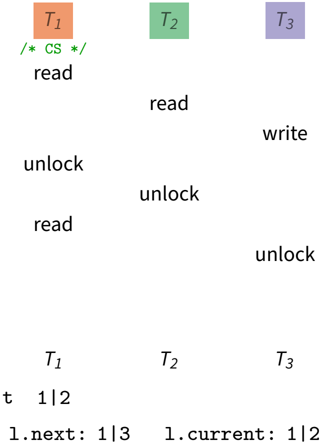
l.next: 1|3 l.current: 0|2

Reader Writer Lock

Fair Ticket Reader Writer Lock

```
struct rw_lock_t {
    rw_union_t current;
    rw_union_t next;
};

void lock_read(rw_lock_t *l) {
    auto t = xadd(&(l->next), 1);
    do {} while (l->current.write != t.write);
}
void lock_write(rw_lock_t *l) {
    auto t = xadd(&(l->next.write), 1);
    do {} while (l->current != t);
}

void unlock_read(rw_lock_t *l) {
    xadd(&(l->current.read), 1);
}
void unlock_write(rw_lock_t *l) {
    l->current.write++;
}
```

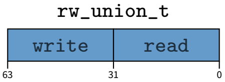


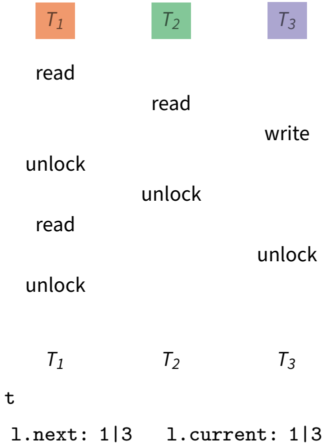


Reader Writer Lock

Fair Ticket Reader Writer Lock

```
struct rw_lock_t {
    rw_union_t current;
    rw_union_t next;
};
void lock_read(rw_lock_t *l) {
    auto t = xadd(&(l->next), 1);
    do {} while (l->current.write != t.write);
}
void lock_write(rw_lock_t *l) {
    auto t = xadd(&(l->next.write), 1);
    do {} while (l->current != t);
}
void unlock_read(rw_lock_t *l) {
    xadd(&(l->current.read), 1);
}
void unlock_write(rw_lock_t *l) {
    l->current.write++;
}
```





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

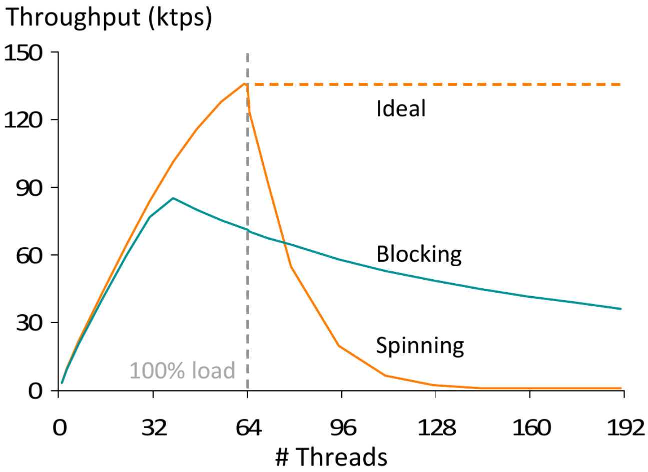
Timeouts & Aborting Locks

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`)