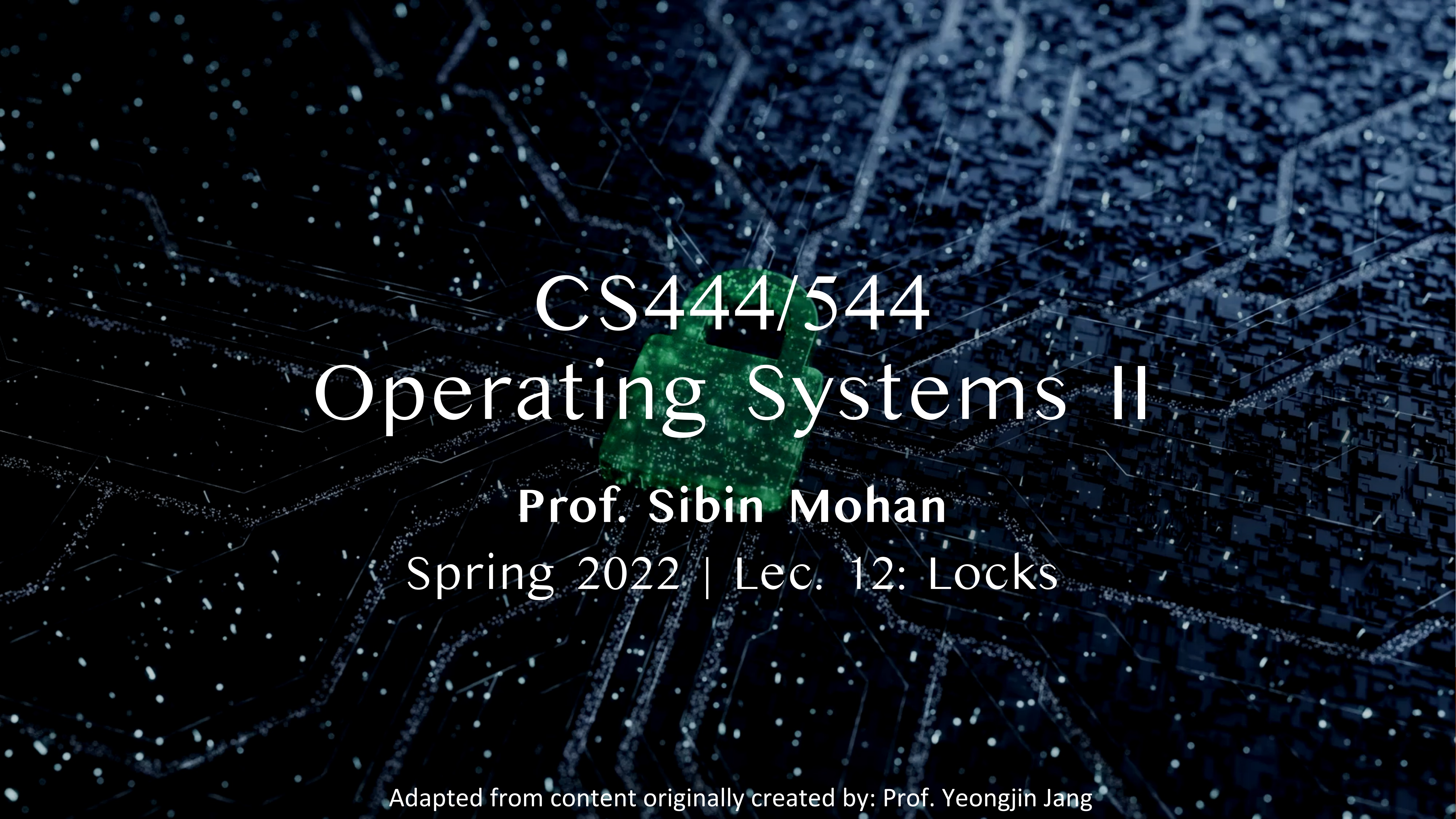


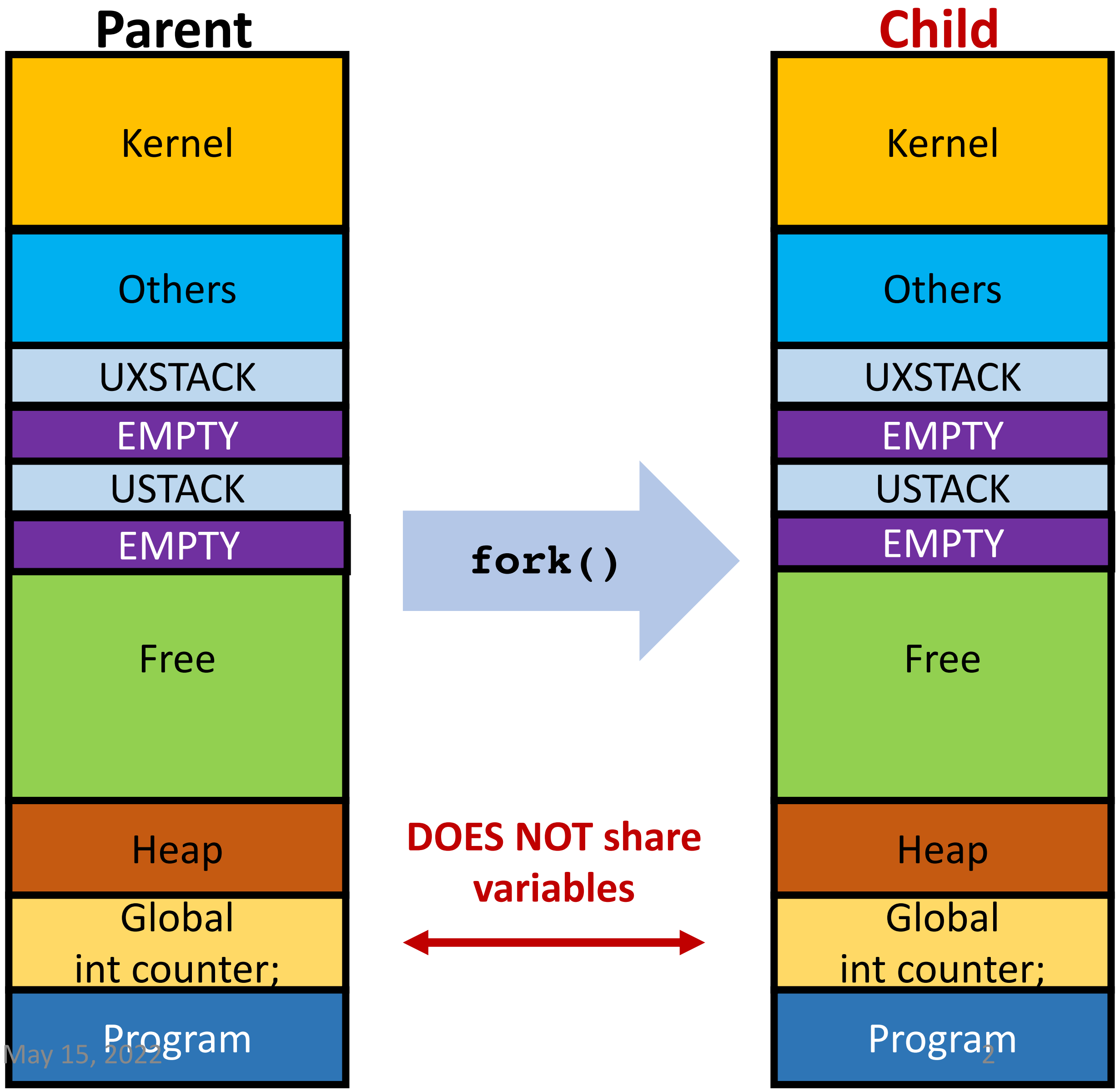
CS444/544 Operating Systems II

Prof. Sibin Mohan

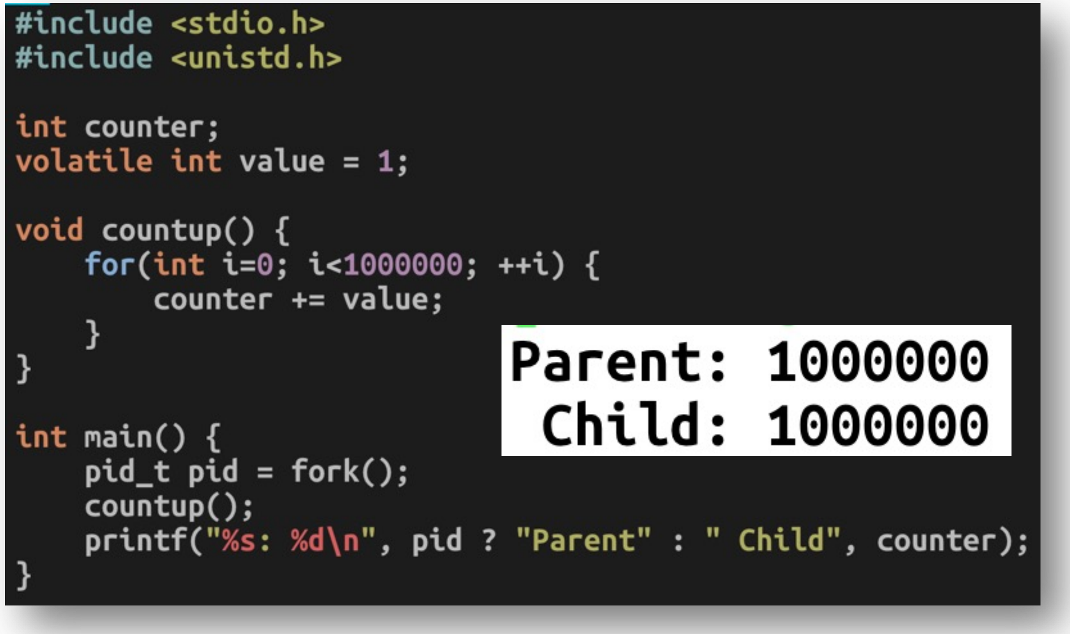
Spring 2022 | Lec. 12: Locks

Adapted from content originally created by: Prof. Yeongjin Jang

Process (Environment in JOS)



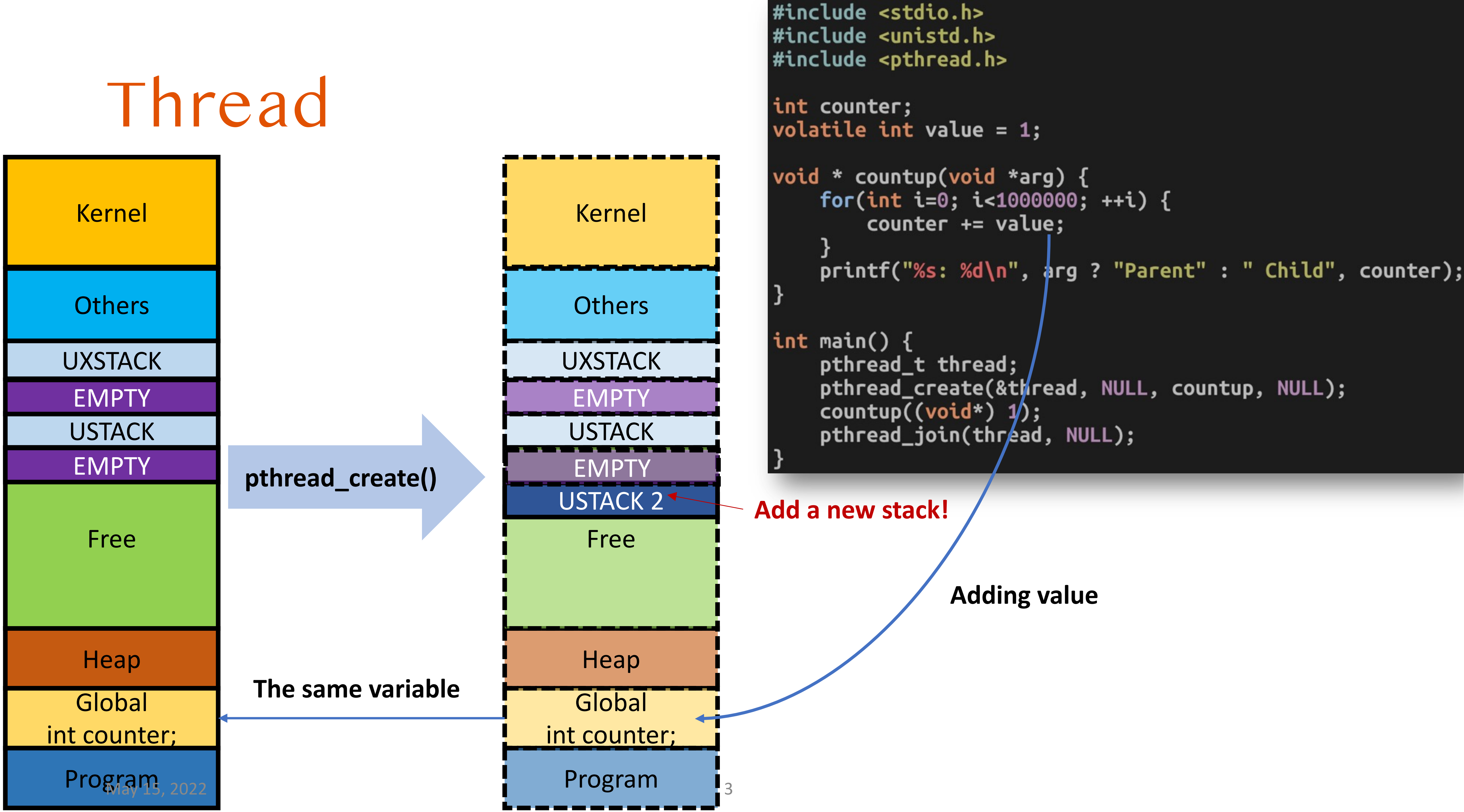
Fork() creates new process by copying memory space
Process creates a new PRIVATE memory space



```
#include <stdio.h>
#include <unistd.h>

int counter;
volatile int value = 1;

void countup() {
    for(int i=0; i<1000000; ++i) {
        counter += value;
    }
}

int main() {
    pid_t pid = fork();
    countup();
    printf("%s: %d\n", pid ? "Parent" : " Child", counter);
}
```

```
Parent: 1000000
 Child: 1000000
```

Thread



Kernel

Others

UXSTACK

EMPTY

USTACK

EMPTY

Free

Heap

Global
int counter;

Program

pthread_create()

Kernel

Others

UXSTACK

EMPTY

USTACK

EMPTY

USTACK 2

Free

Heap

Global
int counter;

Program

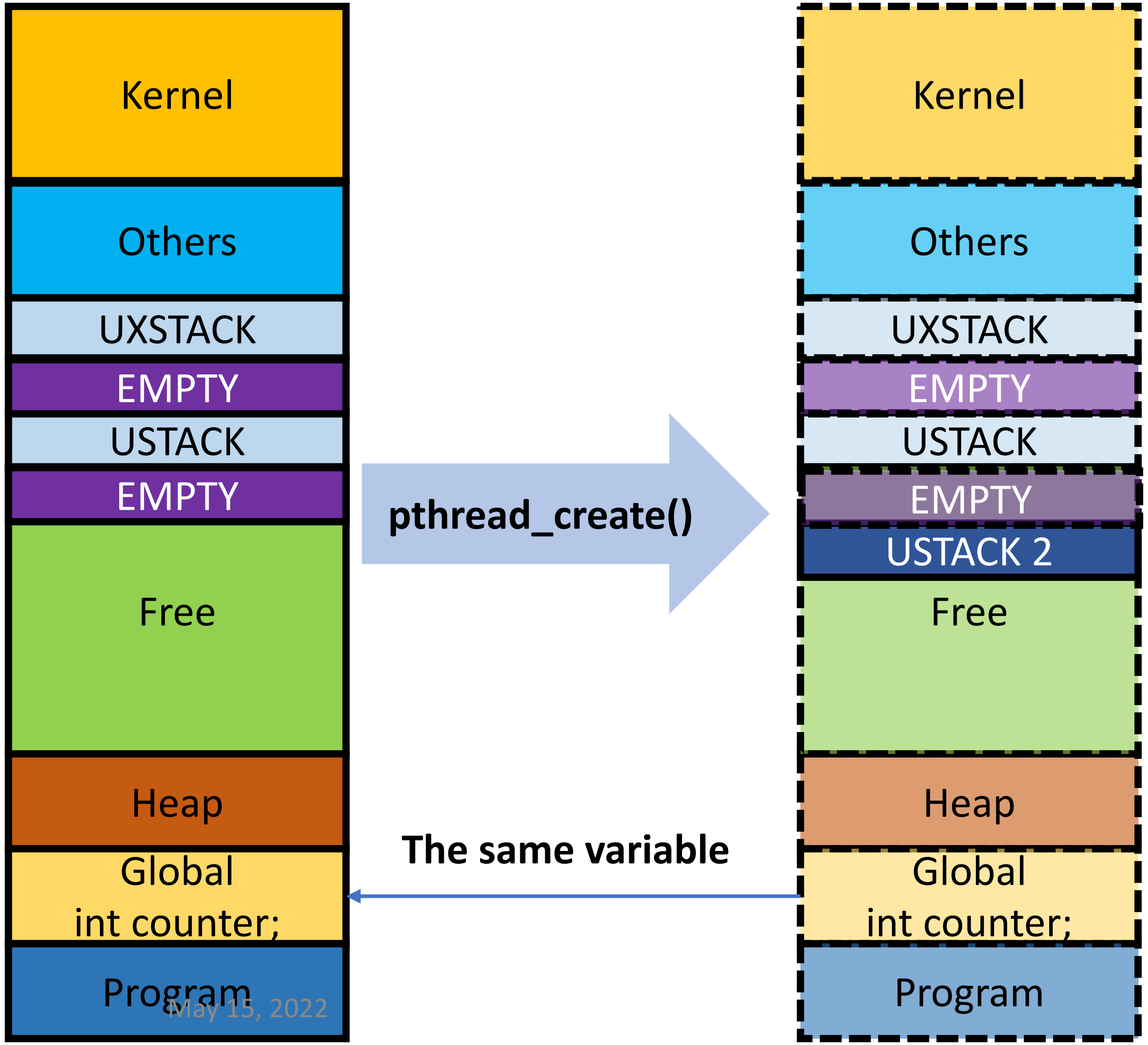
Add a new stack!

The same variable

Adding value

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

int counter;
volatile int value = 1;

void * countup(void *arg) {
    for(int i=0; i<1000000; ++i) {
        counter += value;
    }
    printf("%s: %d\n", arg ? "Parent" : " Child", counter);
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, countup, NULL);
    countup((void*) 1);
    pthread_join(thread, NULL);
}
```

Concurrency Issues



Kernel

Others

UXSTACK

EMPTY

USTACK

EMPTY

Free

Heap

Global
int counter;

Program

pthread_create()

Kernel

Others

UXSTACK

EMPTY

USTACK

EMPTY

USTACK 2

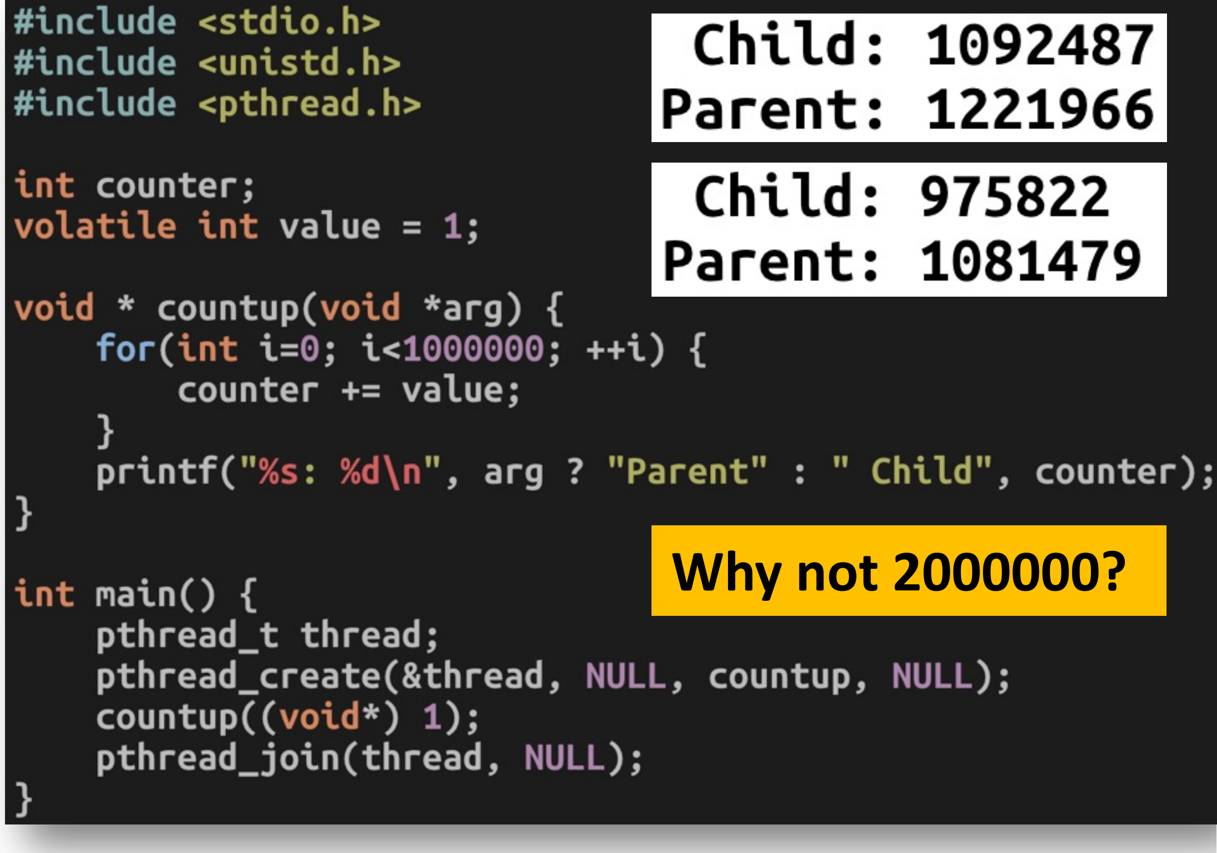
Free

Heap

Global
int counter;

Program

The same variable

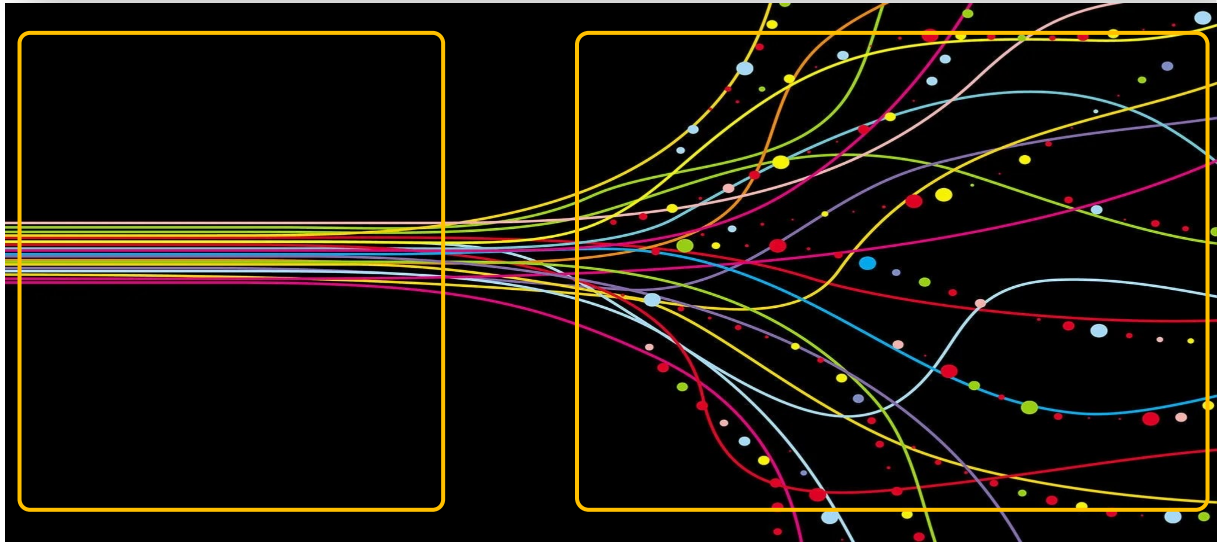


```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

int counter;
volatile int value = 1;

void * countup(void *arg) {
    for(int i=0; i<1000000; ++i) {
        counter += value;
    }
    printf("%s: %d\n", arg ? "Parent" : " Child", counter);
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, countup, NULL);
    countup((void*) 1);
    pthread_join(thread, NULL);
}
```

```
 Child:  1092487
Parent:  1221966
```

```
 Child:  975822
Parent:  1081479
```

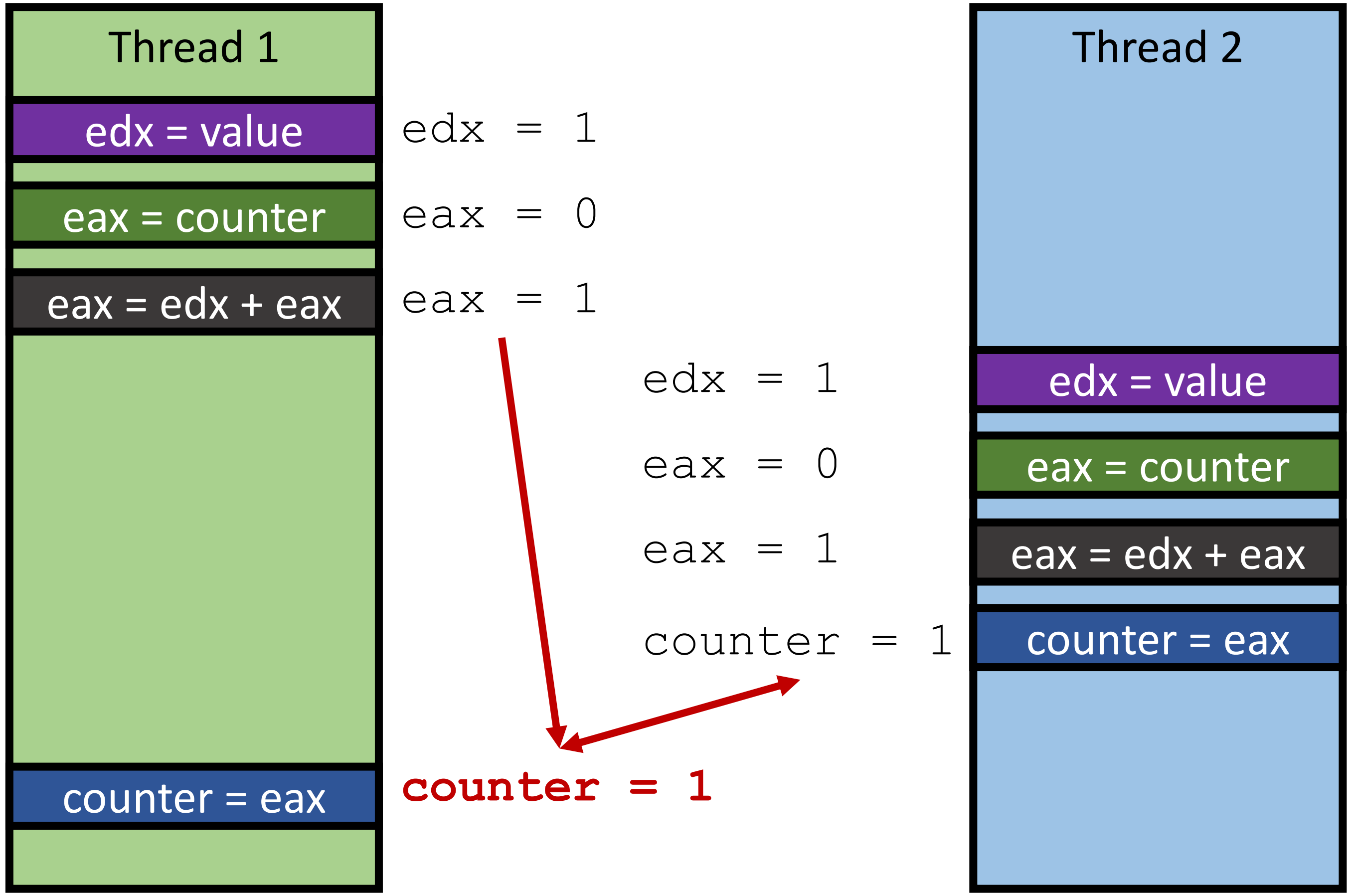
Why not 2000000?



Data Race Example (Race cond.)

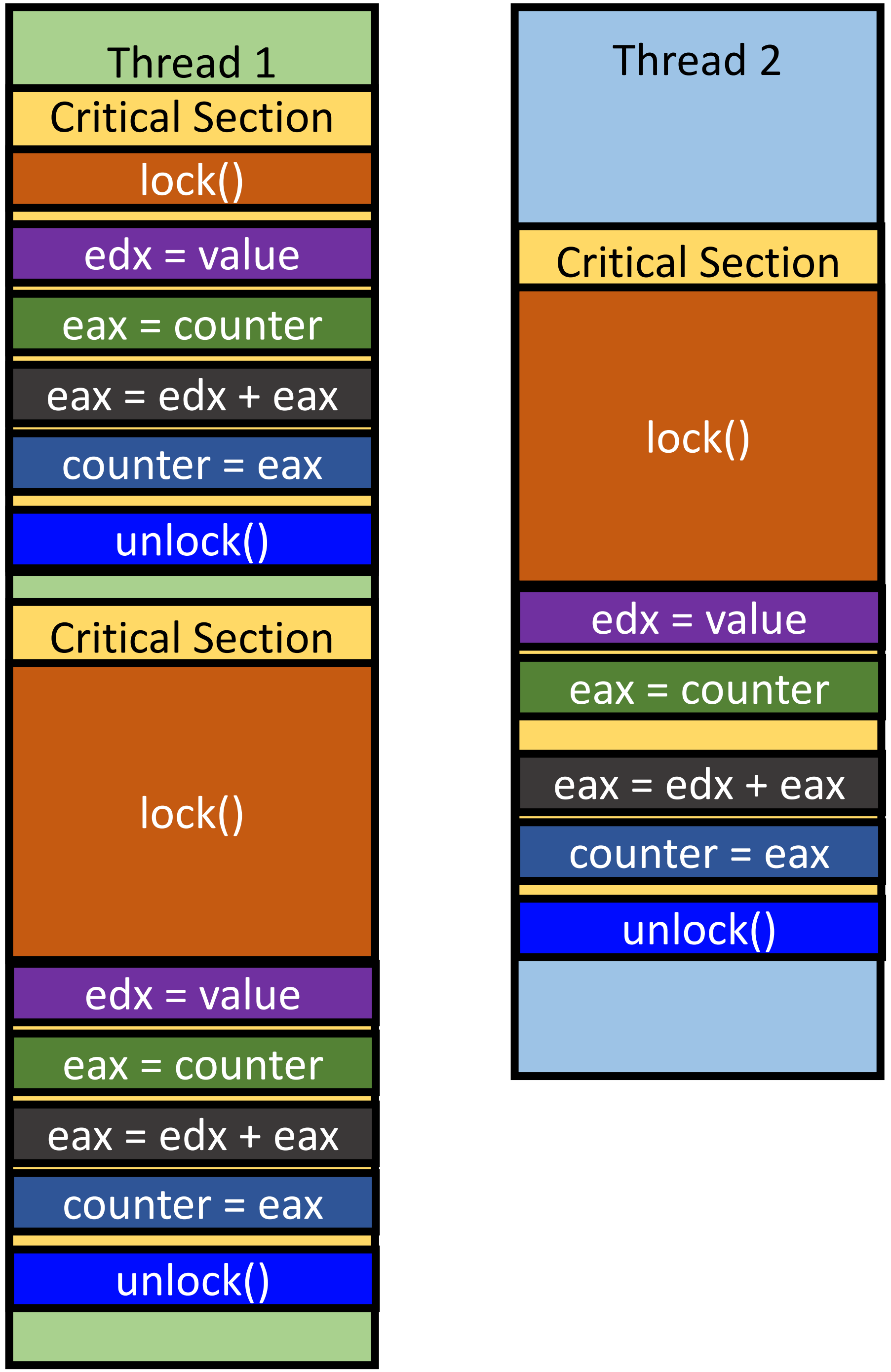
- counter += value
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`

- Assume at start,
  - counter = 0
  - value = 1



Overwrite, inconsistent!

Mutex Example



Thread 1

Critical Section

lock()

edx = value

eax = counter

eax = edx + eax

counter = eax

unlock()

Critical Section

lock()

edx = value

eax = counter

eax = edx + eax

counter = eax

unlock()

Thread 2

Critical Section

lock()

edx = value

eax = counter

eax = edx + eax

counter = eax

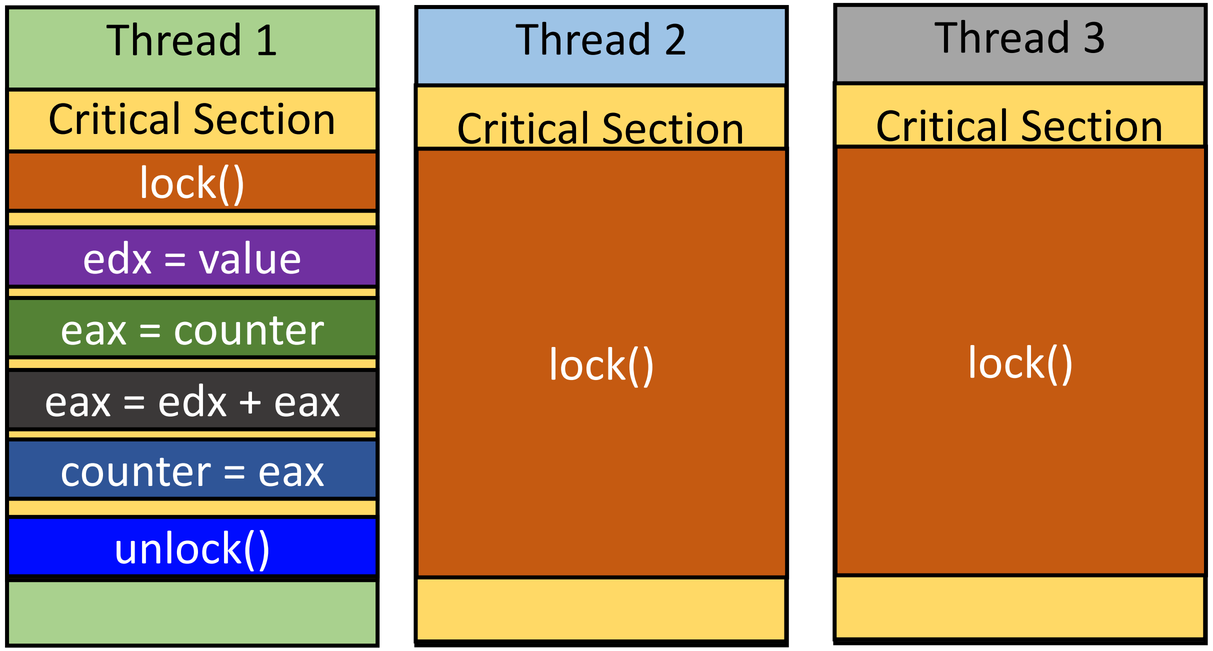
unlock()

How Can We Create Lock/Unlock for Mutex? -- Spinlock

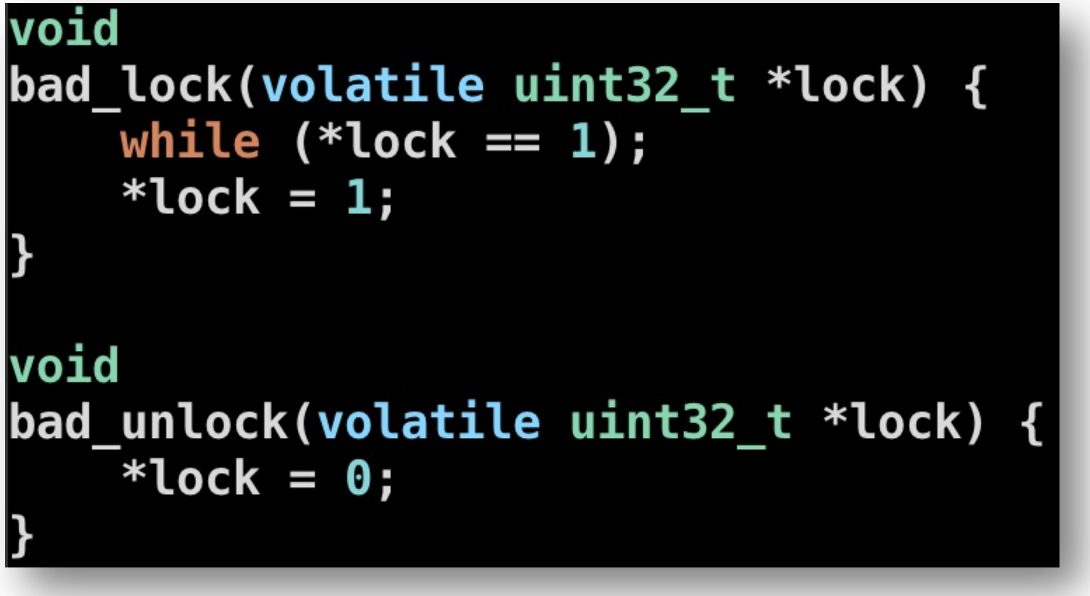
- Only one can run in critical section

- Others must wait!
  - Until nobody runs in critical section

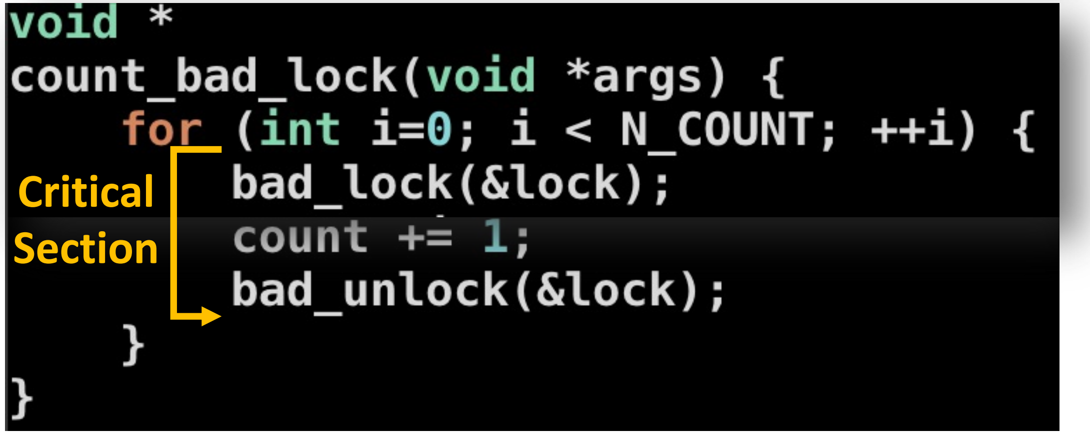
- How can we create such
  - Lock() / Unlock() ?



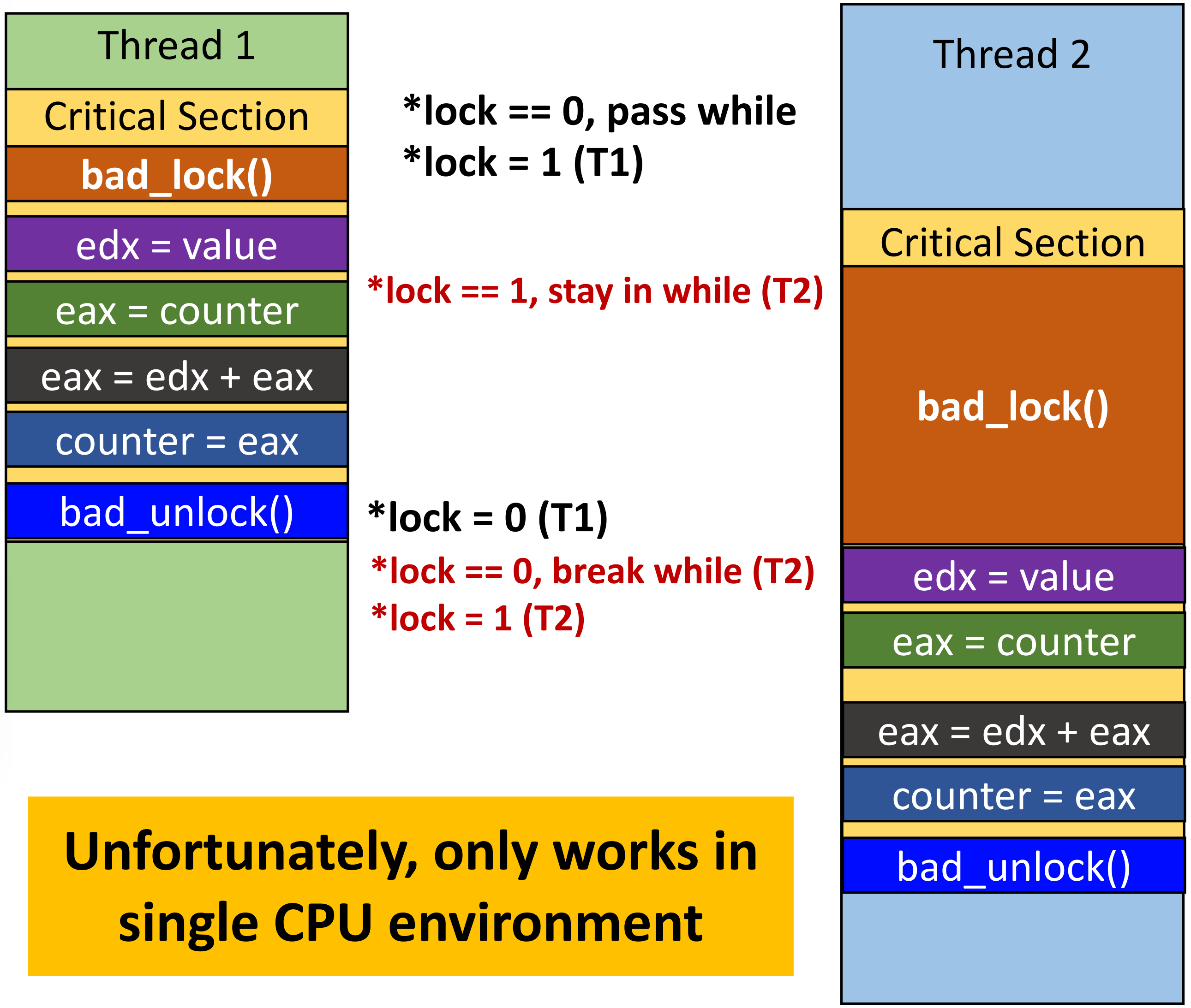
How Can We Implement Locks?



```
void
bad_lock(volatile uint32_t *lock) {
    while (*lock == 1);
    *lock = 1;
}

void
bad_unlock(volatile uint32_t *lock) {
    *lock = 0;
}
```



```
void *
count_bad_lock(void *args) {
    for (int i=0; i < N_COUNT; ++i) {
        bad_lock(&lock);
        count += 1;
        bad_unlock(&lock);
    }
}
```



Thread 1

Critical Section

bad_lock()

edx = value

eax = counter

eax = edx + eax

counter = eax

bad_unlock()

*lock == 0, pass while
*lock = 1 (T1)

*lock == 1, stay in while (T2)

*lock = 0 (T1)

*lock == 0, break while (T2)
*lock = 1 (T2)

Thread 2

Critical Section

bad_lock()

edx = value

eax = counter

eax = edx + eax

counter = eax

bad_unlock()

Unfortunately, only works in single CPU environment

May 15, 2022

How Can We Create Lock/Unlock for Mutex? | **Spinlock**

- Run in a **loop** to **check if critical section is empty**
- Set a lock variable, e.g., **`lock`**
  - 1 → locked
  - 0 → open
- **locking(`lock`)**
  - Wait until lock value becomes 0

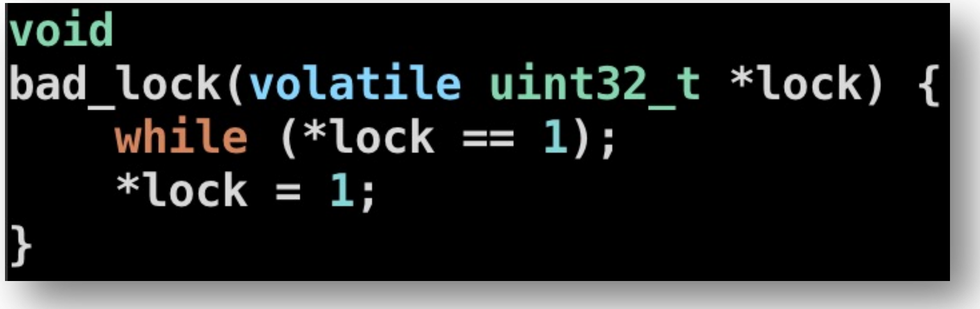
```
while(*lock == 1);
```

Then, nobody runs in the critical section!

  - set *`lock` = 1
- **unlocking(`lock`)**
  - set *`lock` = 0

*lock == 0

*lock == 1

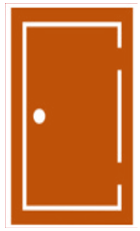


```
void
bad_lock(volatile uint32_t *lock) {
    while (*lock == 1);
    *lock = 1;
}
```

How Can We Create Lock/Unlock for Mutex? | **Spinlock**

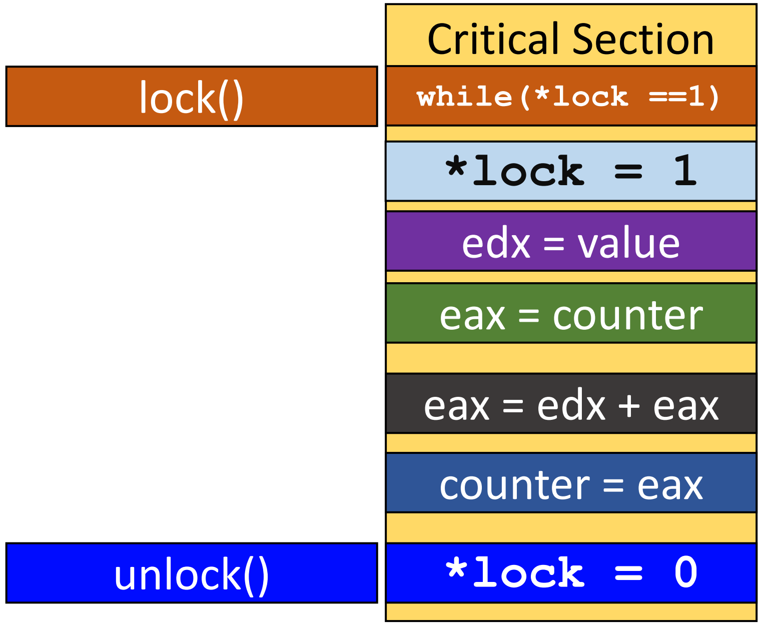
- Run in a **loop** to **check if critical section is empty**
- Set a lock variable, e.g., **`lock`**
  - 1 → locked
  - 0 → open
- **locking(`lock`)**
  - Wait until lock value becomes 0

    `while(*lock == 1);`
  - set *`lock` = 1
- **unlocking(`lock`)**
  - set *`lock` = 0



*lock == 1

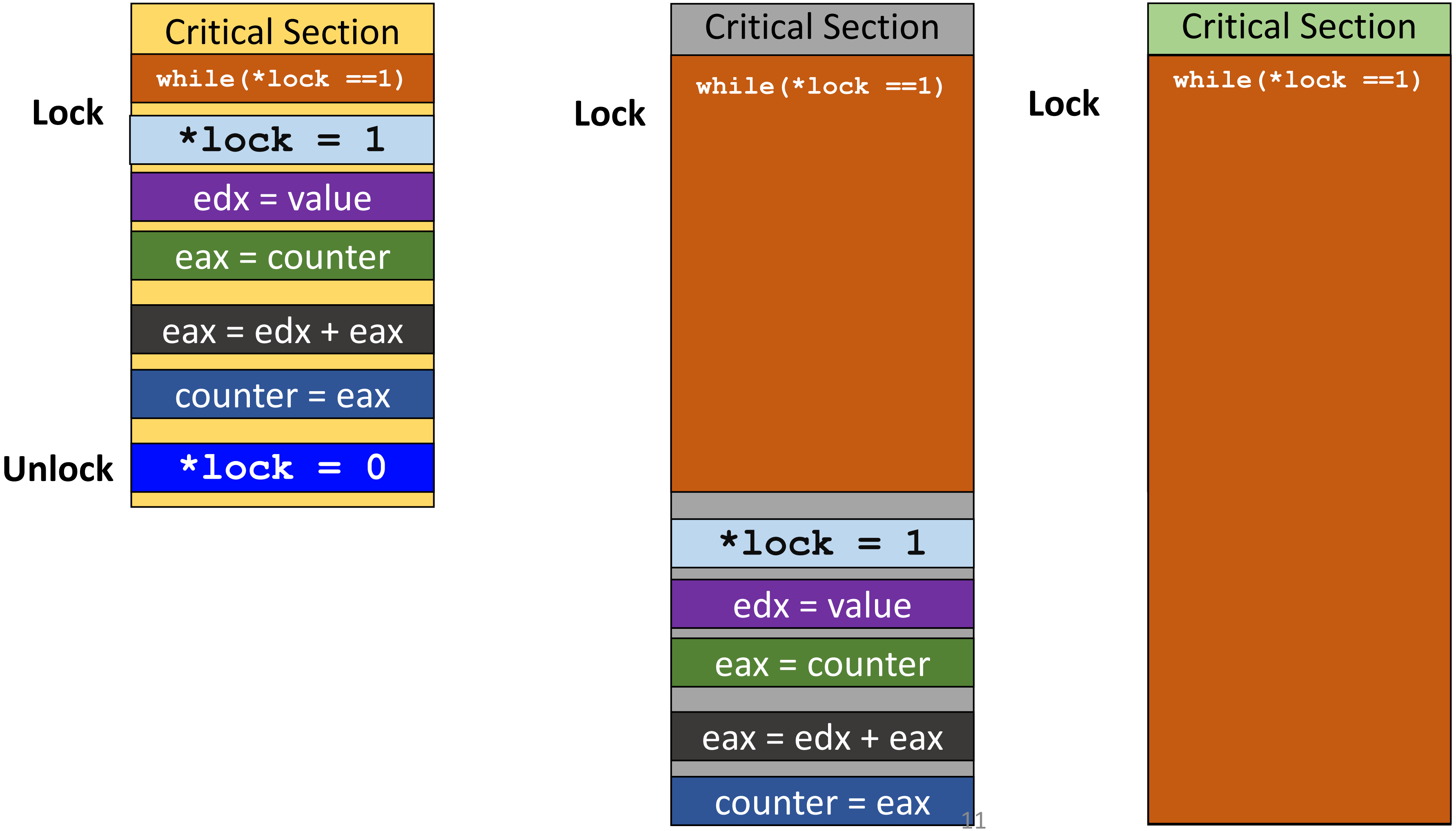
*lock == 0



Critical Section

lock()	`while(*lock ==1)`
	`*lock = 1`
	edx = value
	eax = counter
	eax = edx + eax
	counter = eax
unlock()	`*lock = 0`

Spinlock



Lock

Unlock

Critical Section

while(*lock ==1)

*lock = 1

edx = value

eax = counter

eax = edx + eax

counter = eax

*lock = 0

Lock

Critical Section

while(*lock ==1)

*lock = 1

edx = value

eax = counter

eax = edx + eax

counter = eax

Lock

Critical Section

while(*lock ==1)

Spinlock Candidates

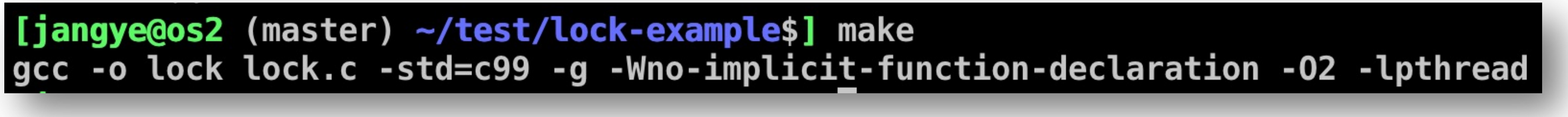
- no lock
- bad lock
- xchg lock
- cmpxchg lock
- tts lock
- backoff cmpxchg
- pthread_mutex

Spinlock Implementations

- https://gitlab.unexploitable.systems/root/lock-example

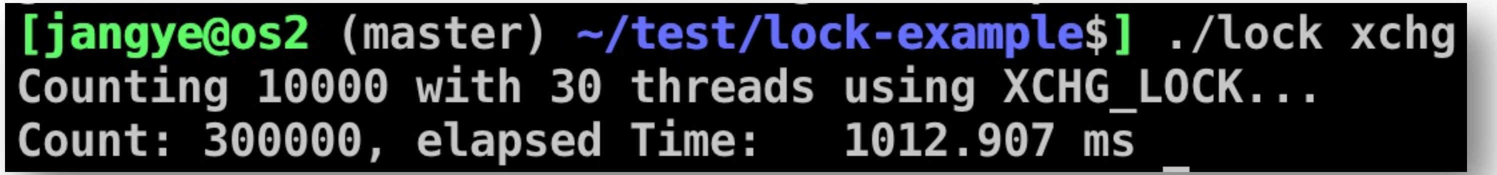
```
git clone git@gitlab.unexploitable.systems:root/lock-example
```

- Run **30** threads, each counts up to **10000** → **total 300,000 counts**
- Build code
  - $ make



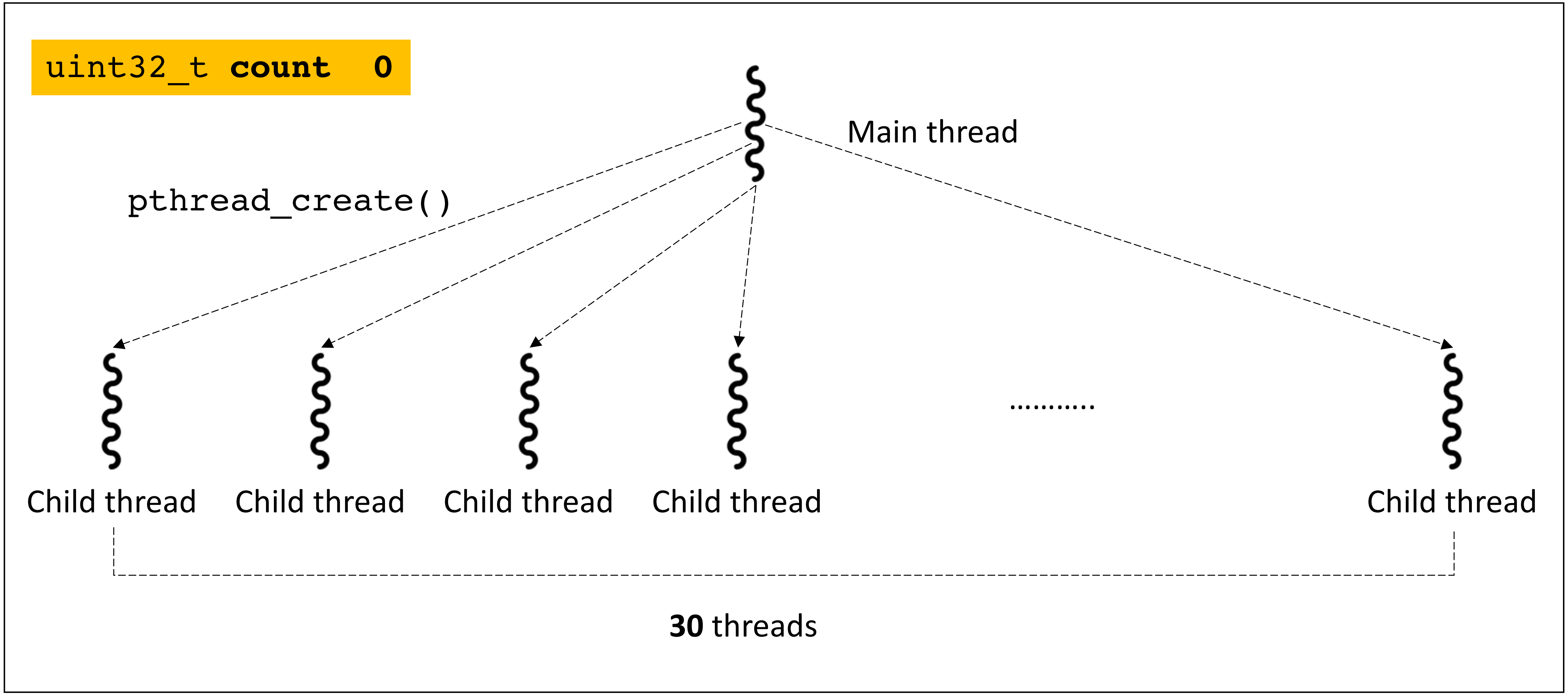
```
[jangye@os2 (master) ~/test/lock-example$] make
gcc -o lock lock.c -std=c99 -g -Wno-implicit-function-declaration -O2 -lpthread
```

- Run code
  - $ ./lock xchg # shows the result of using xchg lock
  - $ ./perf-lock.sh xchg # shows the result of using xchg lock, with cache-miss

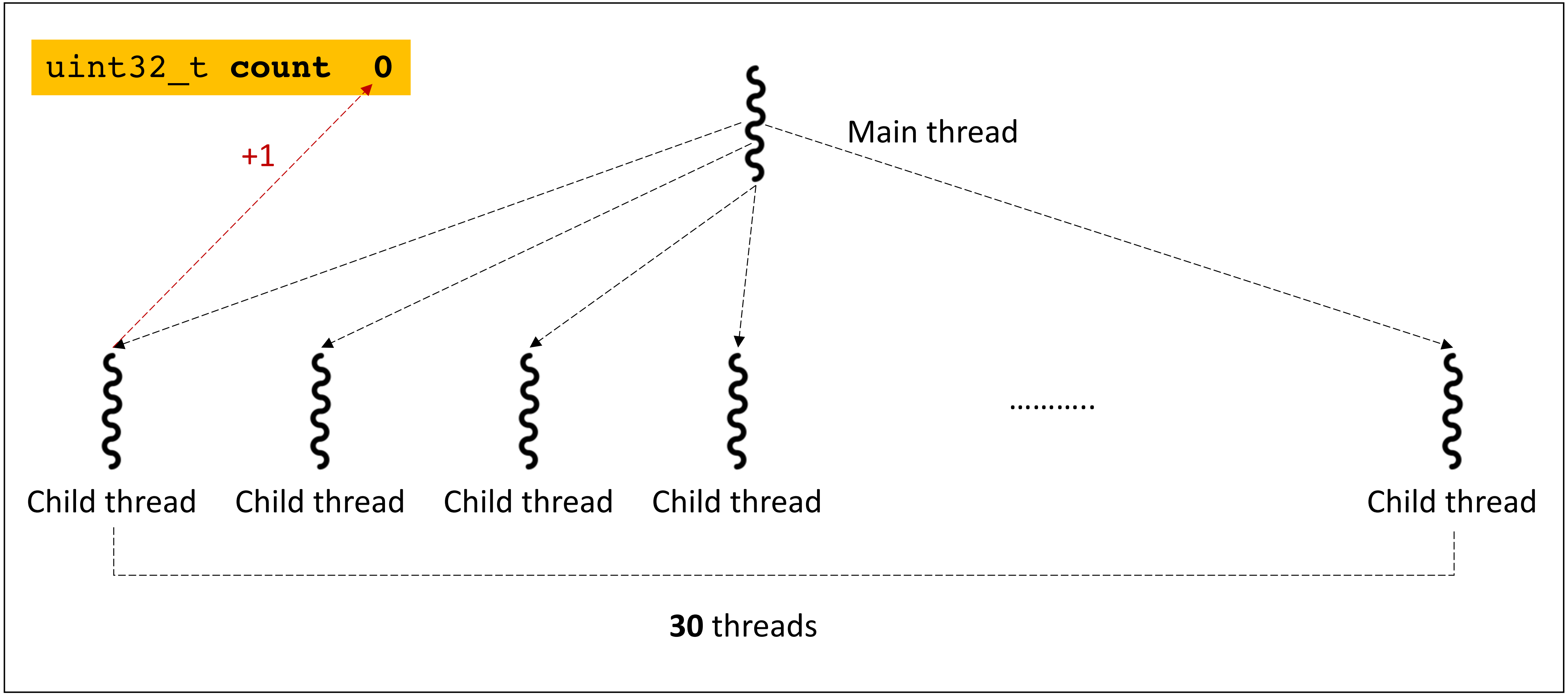


```
[jangye@os2 (master) ~/test/lock-example$] ./lock xchg
Counting 10000 with 30 threads using XCHG_LOCK...
Count: 300000, elapsed Time:   1012.907 ms
```

How lock-example works

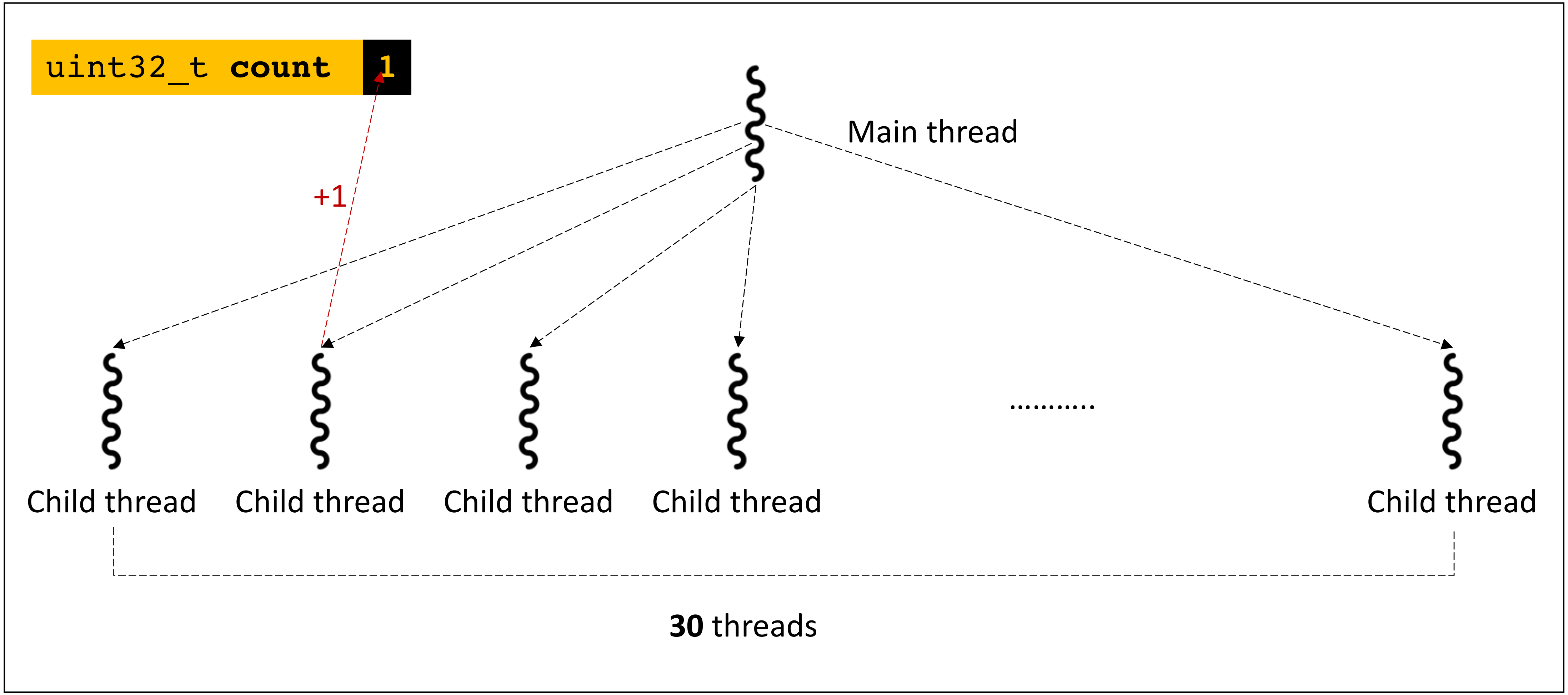


How lock-example works



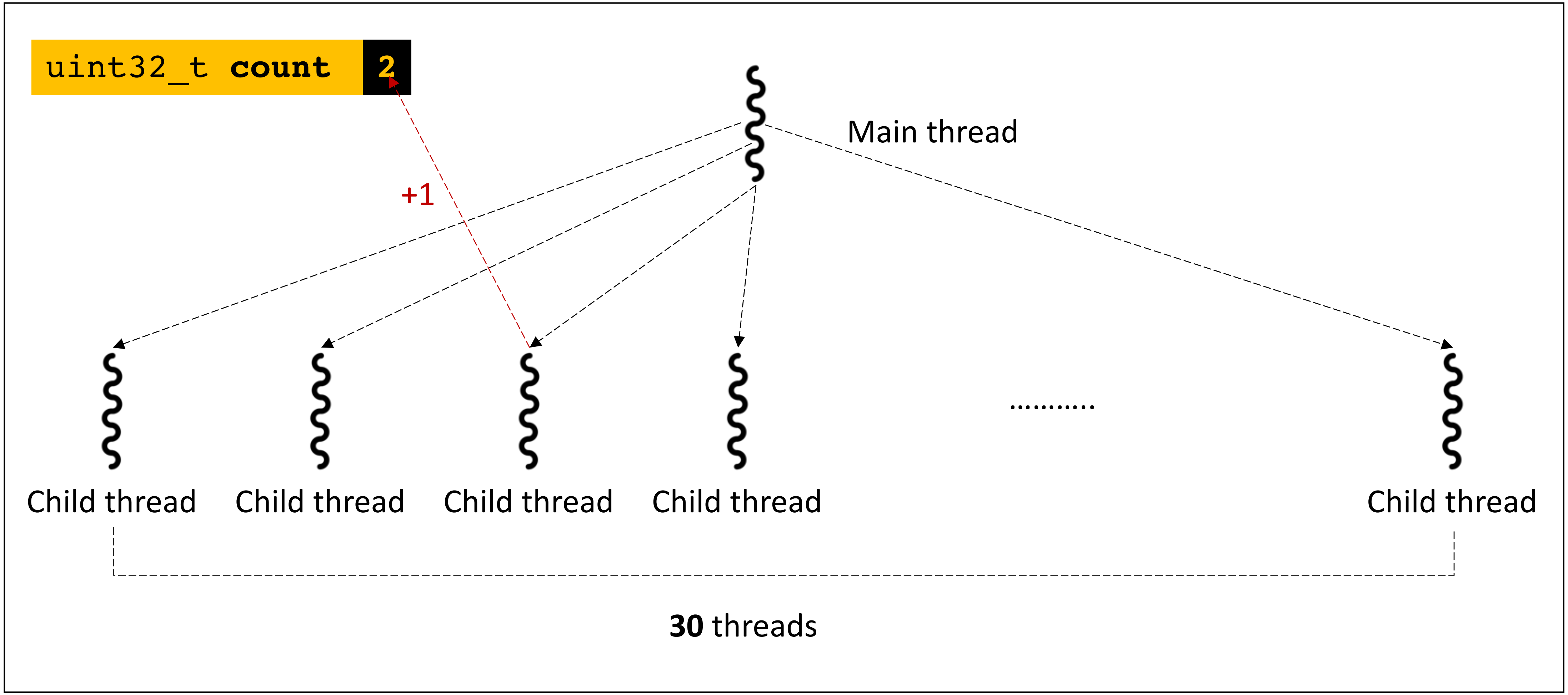
Each thread increases count by 1 for a total of 10000 times

How lock-example works



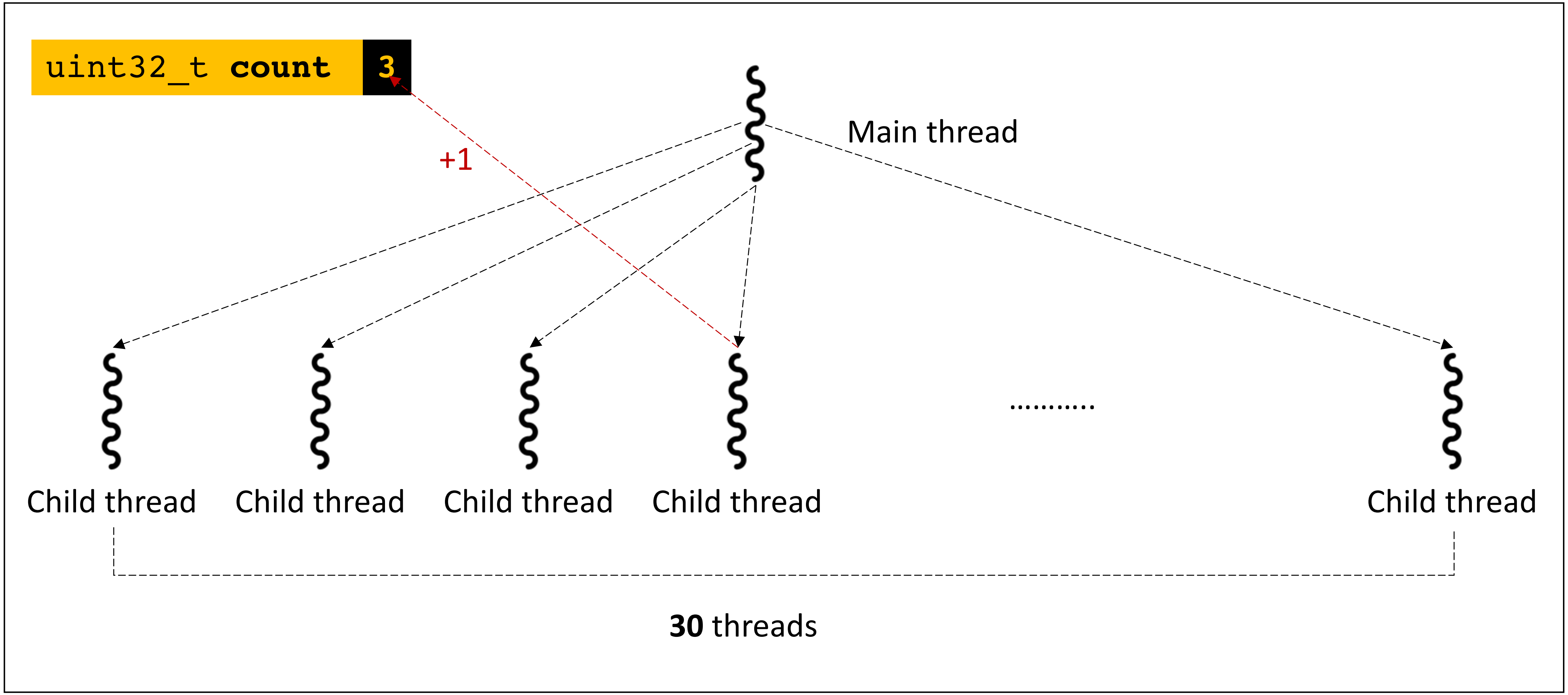
Each thread increases count by 1 for a total of 10000 times

How lock-example works



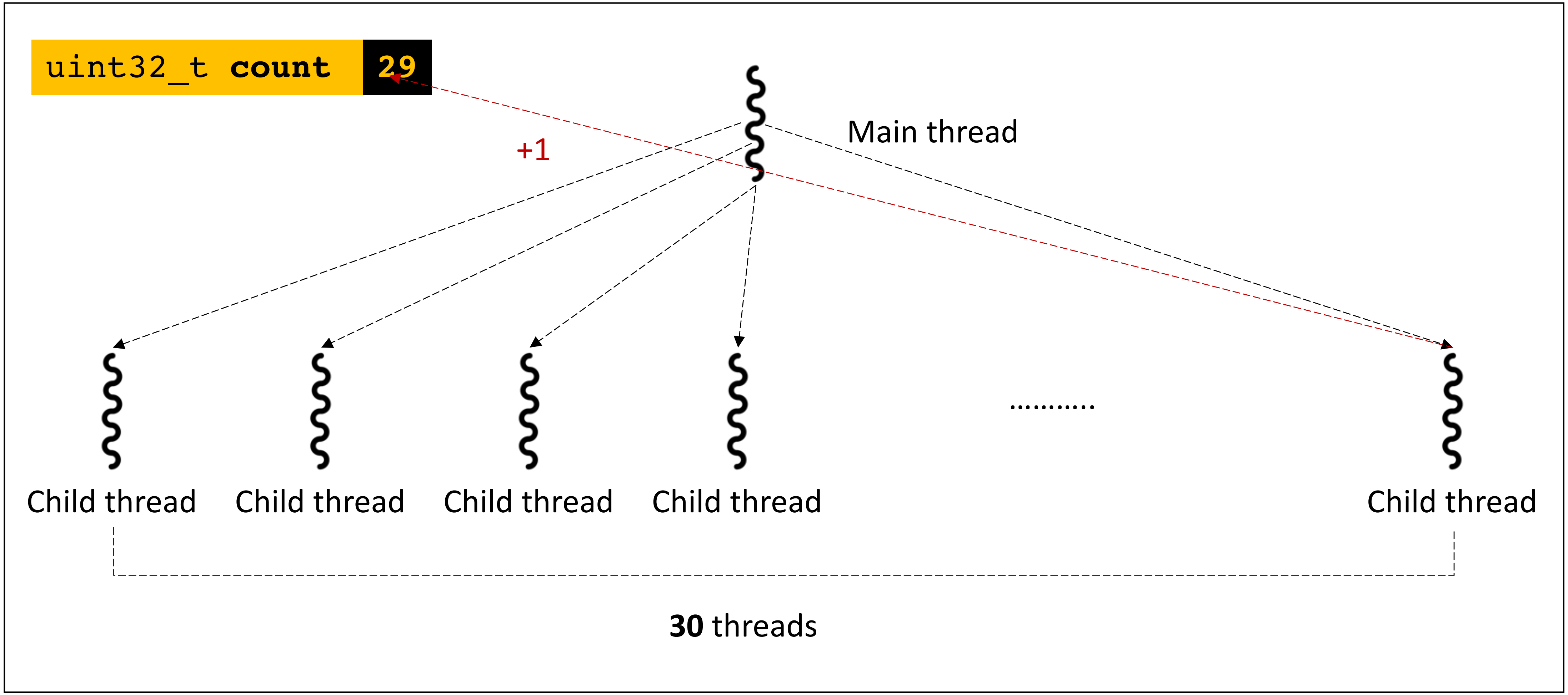
Each thread increases count by 1 for a total of 10000 times

How lock-example works



Each thread increases count by 1 for a total of 10000 times

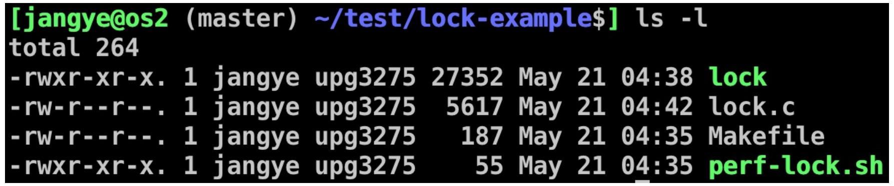
How lock-example works



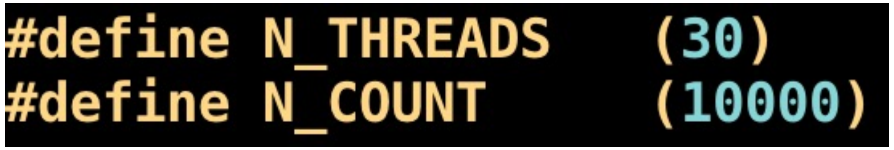
Each thread increases count by 1 for a total of 10000 times

lock.c Implementation

- Multi-threaded Program
  - **30 threads**
  - Each counts **10,000**
- Correct result = 300,000

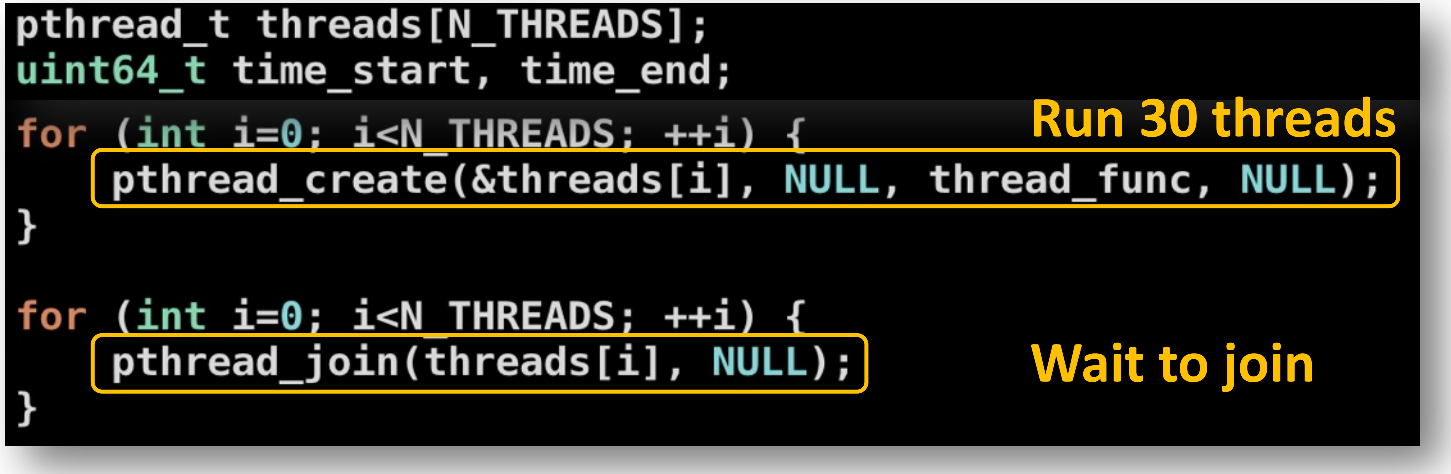


```
[jangye@os2 (master) ~/test/lock-example$] ls -l
total 264
-rwxr-xr-x. 1 jangye upg3275 27352 May 21 04:38 lock
-rw-r--r--. 1 jangye upg3275  5617 May 21 04:42 lock.c
-rw-r--r--. 1 jangye upg3275   187 May 21 04:35 Makefile
-rwxr-xr-x. 1 jangye upg3275    55 May 21 04:35 perf-lock.sh
```



```
#define N_THREADS    (30)
#define N_COUNT      (10000)
```

lock.c Implementation [contd.]

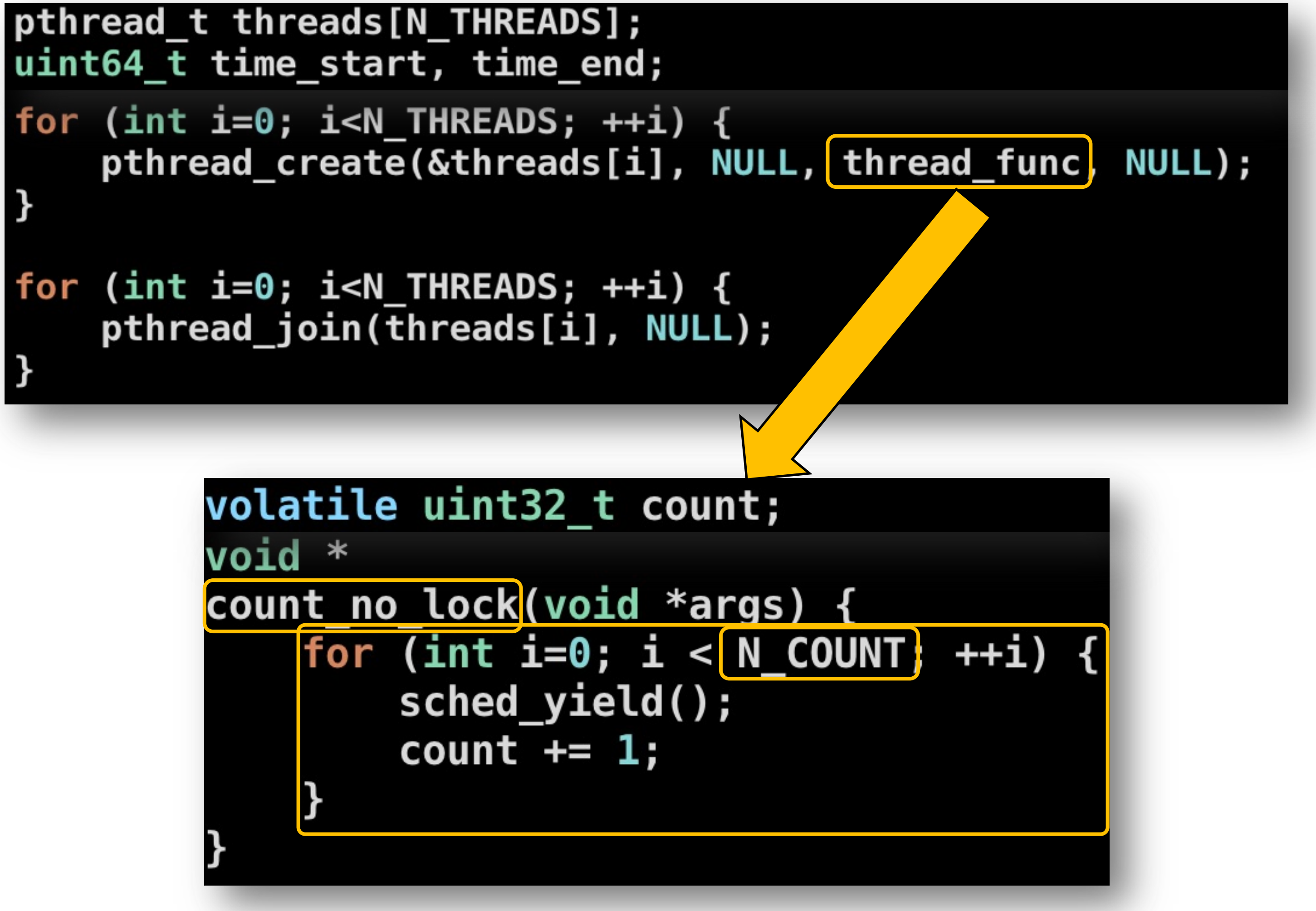


```
pthread_t threads[N_THREADS];
uint64_t time_start, time_end;

for (int i=0; i<N_THREADS; ++i) {
    pthread_create(&threads[i], NULL, thread_func, NULL);
}

for (int i=0; i<N_THREADS; ++i) {
    pthread_join(threads[i], NULL);
}
```

Run 30 threads

Wait to join

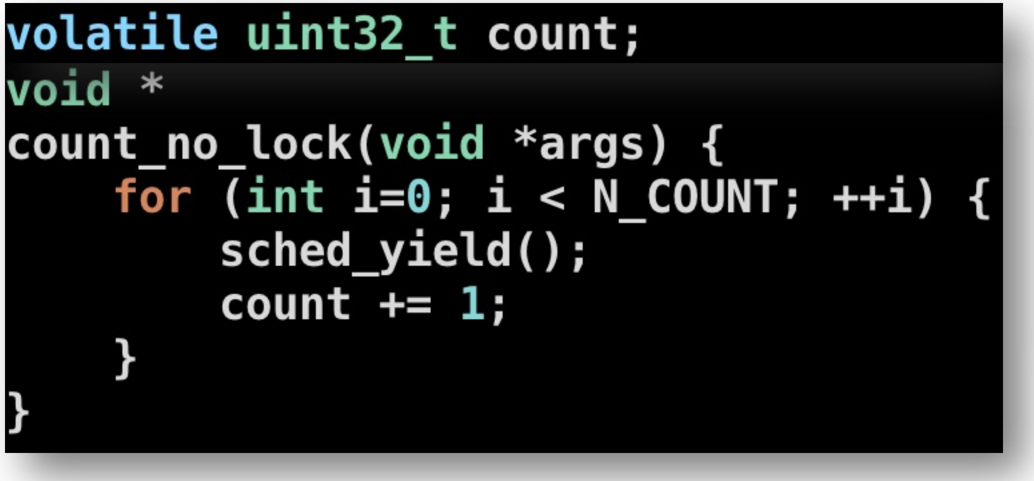
lock.c Implementation [contd.]



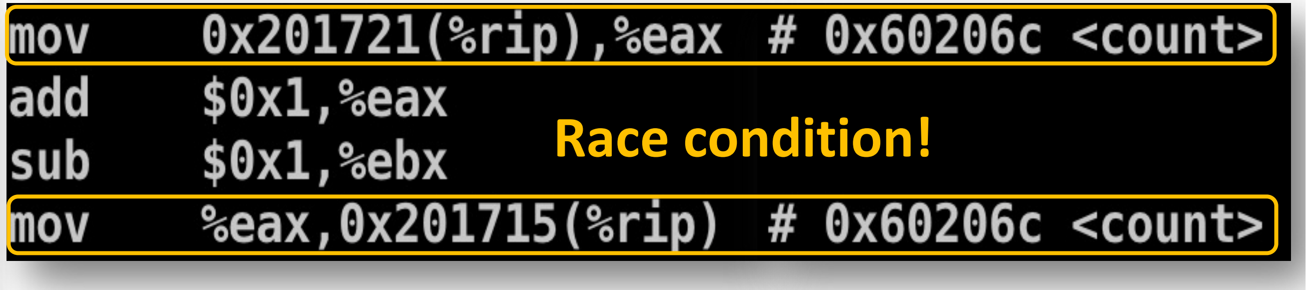
```
pthread_t threads[N_THREADS];
uint64_t time_start, time_end;
for (int i=0; i<N_THREADS; ++i) {
    pthread_create(&threads[i], NULL, thread_func, NULL);
}

for (int i=0; i<N_THREADS; ++i) {
    pthread_join(threads[i], NULL);
}
```

```
volatile uint32_t count;
void *
count_no_lock(void *args) {
    for (int i=0; i < N_COUNT; ++i) {
        sched_yield();
        count += 1;
    }
}
```

lock.c Implementation [contd.]



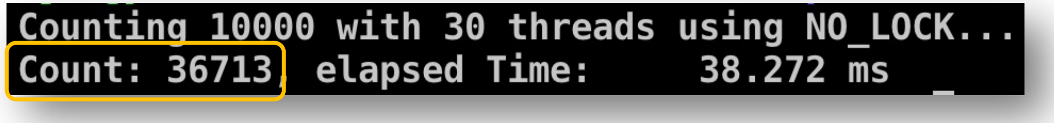
```c
volatile uint32_t count;
void *
count_no_lock(void *args) {
    for (int i=0; i < N_COUNT; ++i) {
        sched_yield();
        count += 1;
    }
}
```



```
mov     0x201721(%rip),%eax  # 0x60206c <count>
add     $0x1,%eax
sub     $0x1,%ebx
mov     %eax,0x201715(%rip)  # 0x60206c <count>
```

Race condition!

Results:



```
Counting 10000 with 30 threads using NO_LOCK...
Count: 36713, elapsed Time:     38.272 ms
```

Lock Example

- **Thread functions**

```
$ ./lock no            # using no lock at all
$ ./lock bad           # using a bad lock implementation
```

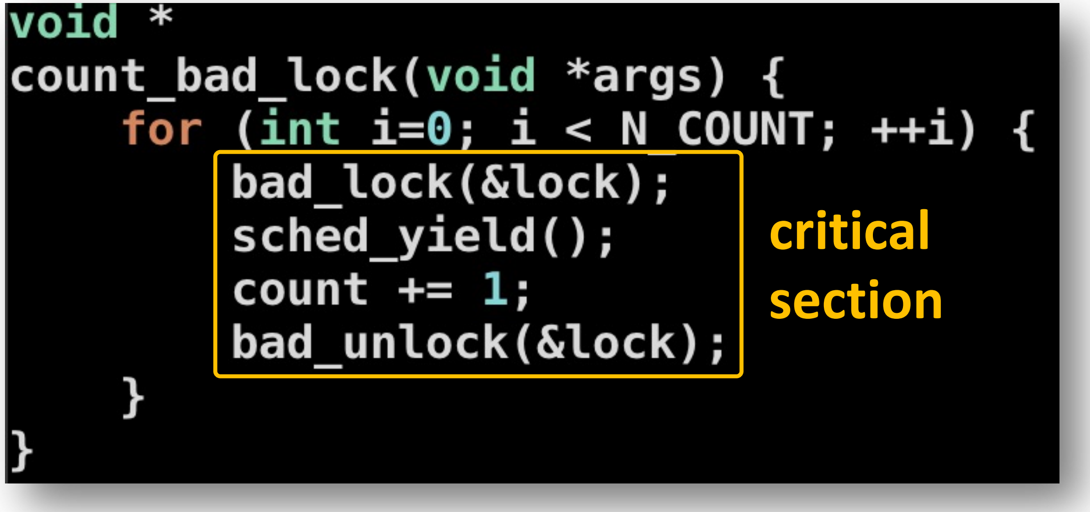
inconsistent

```
$ ./lock xchg          # using xchg lock
$ ./lock cmpxchg       # using lock cmpxchg
$ ./lock tts           # using soft test-and-test & set with xchg
$ ./lock backoff       # using exponential backoff cmpxchg
$ ./lock mutex         # using pthread mutex
```

consistent

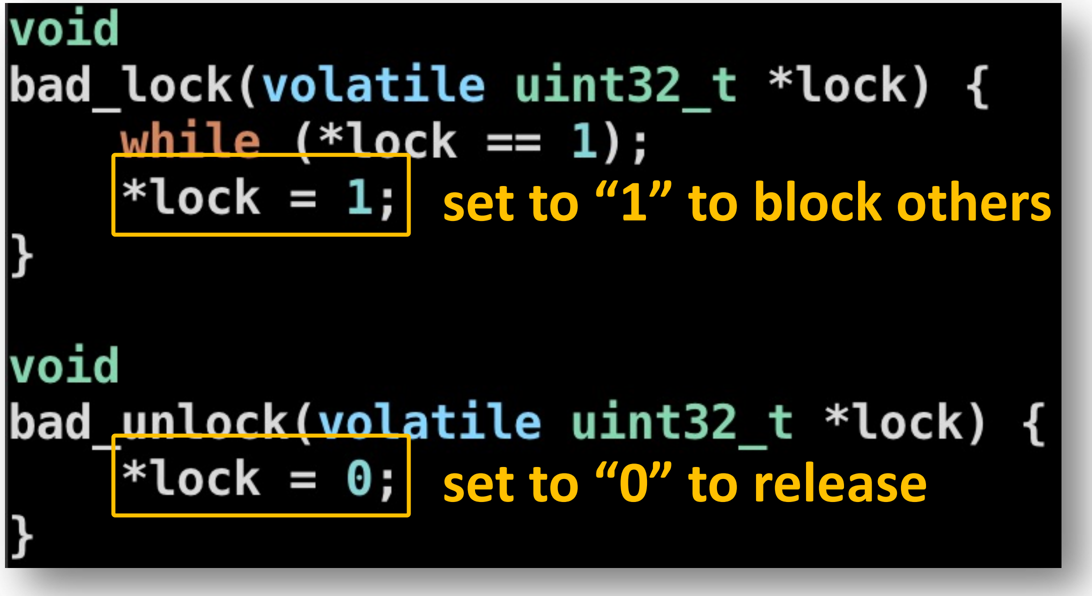
1st Candidate: bad_lock

- **bad_lock**
  - Wait until **lock** becomes 0 (loops if 1)
  - set **lock** → 1
  - **Others must wait**!
- **bad_unlock**
  - Just set ***lock** → 0



```
void *
count_bad_lock(void *args) {
    for (int i=0; i < N_COUNT; ++i) {
        bad_lock(&lock);
        sched_yield();
        count += 1;
        bad_unlock(&lock);
    }
}
```

critical section



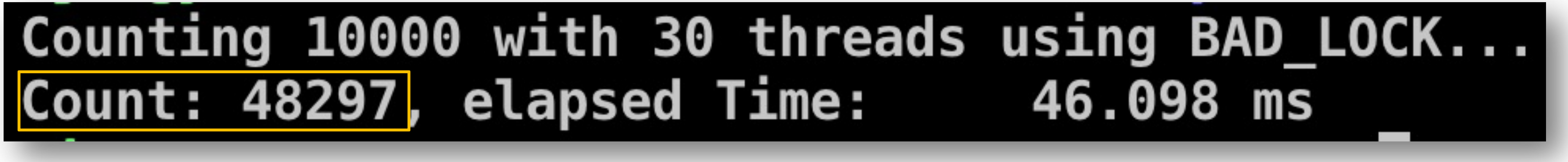
```
void
bad_lock(volatile uint32_t *lock) {
    while (*lock == 1);
    *lock = 1;
}

void
bad_unlock(volatile uint32_t *lock) {
    *lock = 0;
}
```

set to “1” to block others

set to “0” to release

1^{st} Candidate: bad_lock Result

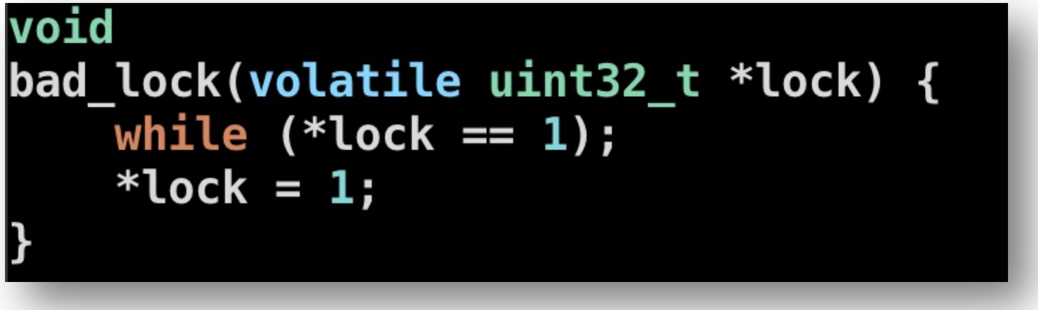
- **Inconsistent!**



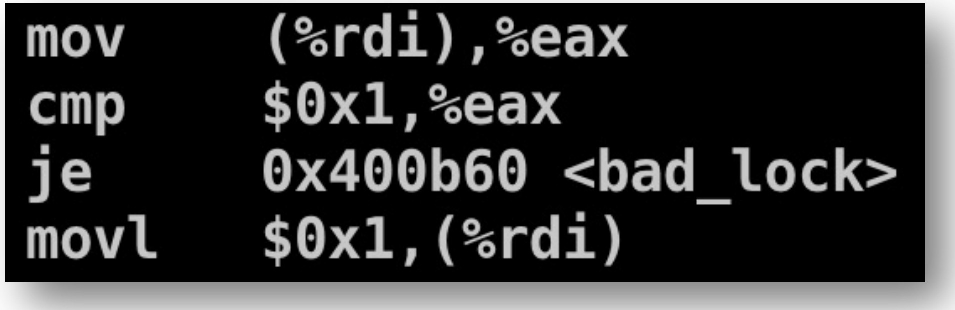
```
Counting 10000 with 30 threads using BAD_LOCK...
Count: 48297, elapsed Time:      46.098 ms
```

WHY?

Inconsistency in bad_lock



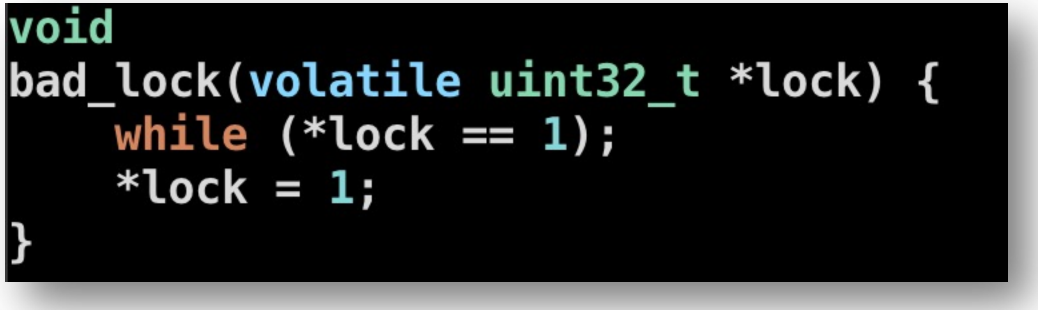
```
void
bad_lock(volatile uint32_t *lock) {
    while (*lock == 1);
    *lock = 1;
}
```



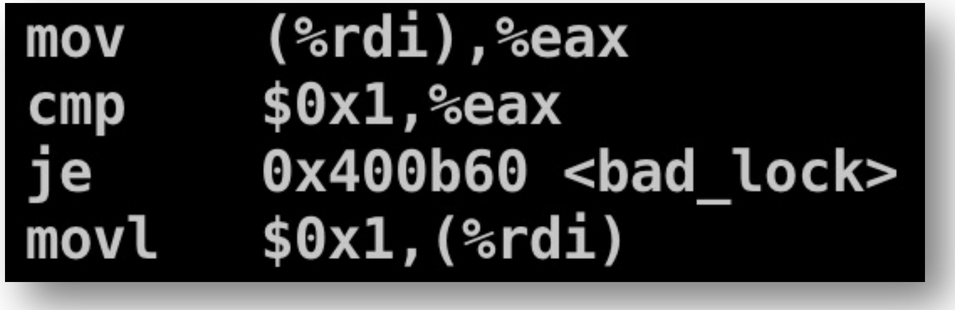
```
mov     (%rdi),%eax
cmp     $0x1,%eax
je      0x400b60 <bad_lock>
movl    $0x1,(%rdi)
```

Is there an issue here?

Inconsistency in bad_lock



```
void
bad_lock(volatile uint32_t *lock) {
    while (*lock == 1);
    *lock = 1;
}
```



```
mov     (%rdi),%eax
cmp     $0x1,%eax
je      0x400b60 <bad_lock>
movl    $0x1,(%rdi)
```

expected behavior

thread 1

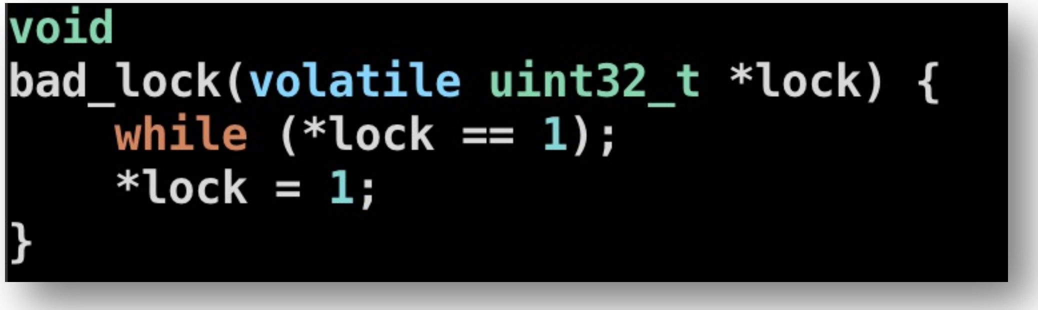
```
mov (%rdi), %eax
cmp $0x1, %eax
je       0x400b60, <bad_lock>
movl $0x1, (%rdi)
```

thread 2

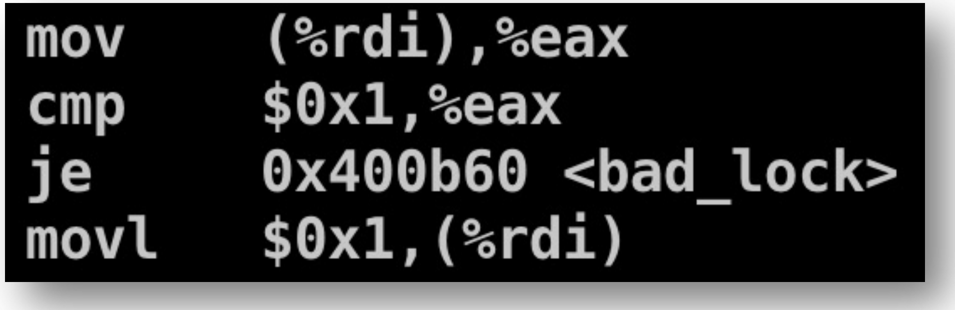
waiting on spin lock

```
mov (%rdi), %eax
cmp $0x1, %eax
je       0x400b60, <bad_lock>
movl $0x1, (%rdi)
```

Inconsistency in bad_lock



```c
void
bad_lock(volatile uint32_t *lock) {
    while (*lock == 1);
    *lock = 1;
}
```



```
mov     (%rdi),%eax
cmp     $0x1,%eax
je      0x400b60 <bad_lock>
movl    $0x1,(%rdi)
```

thread 1

thread 2

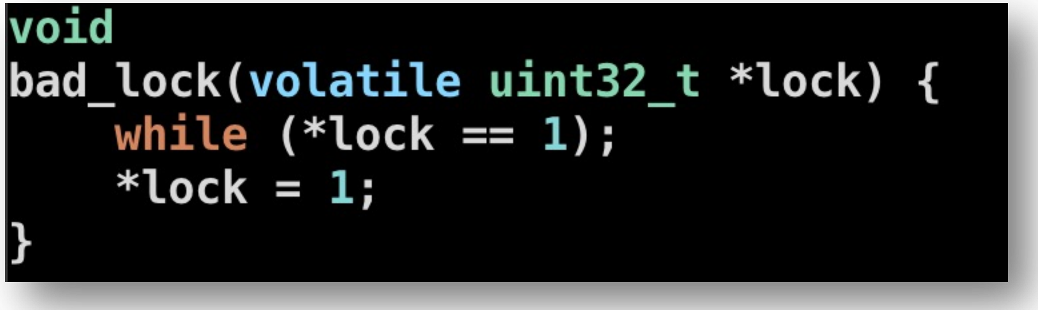
actual behavior!

```
mov (%rdi), %eax
mov (%rdi), %eax
cmp $0x1, %eax
cmp $0x1, %eax
je       0x400b60, <bad_lock>
je       0x400b60, <bad_lock>
movl $0x1, (%rdi)
movl $0x1, (%rdi)
```

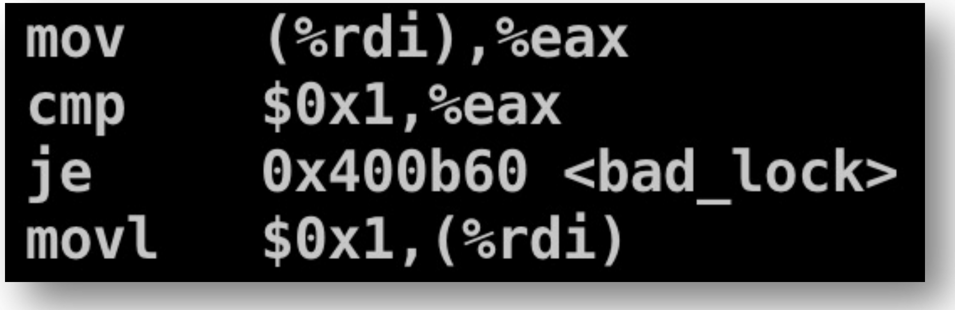
both get locks!
both enter critical section!
Inconsistent!

race condition still exists

Inconsistency in bad_lock



```
void
bad_lock(volatile uint32_t *lock) {
    while (*lock == 1);
    *lock = 1;
}
```



```
mov     (%rdi),%eax
cmp     $0x1,%eax
je      0x400b60 <bad_lock>
movl    $0x1,(%rdi)
```

thread 1

thread 2

actual behavior!

```
mov (%rdi), %eax
cmp $0x1, %eax
je      0x400b60, <bad_lock>
mov (%rdi), %eax
movl $0x1, (%rdi)
cmp $0x1, %eax
je      0x400b60, <bad_lock>
movl $0x1, (%rdi)
```

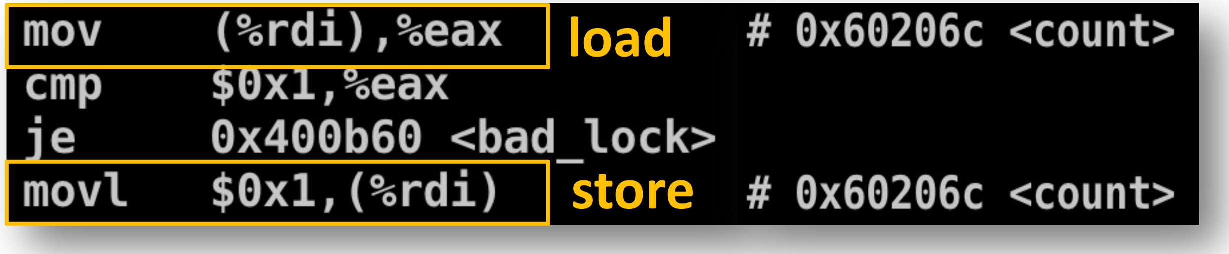
both get locks!
both enter critical section!
Inconsistent!

race condition still exists

How to avoid race conditions?

Reason for Race Conditions?

- **Separate load and store** instructions



```
mov     (%rdi),%eax            # 0x60206c <count>
cmp     $0x1,%eax
je      0x400b60 <bad_lock>
movl    $0x1,(%rdi)            # 0x60206c <count>
```

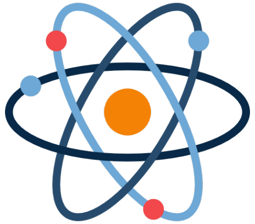
- `while (*lock == 1); *lock = 0;` was a **bad implementation**

- Need a method to **remove gap between load and store!**

Atomic Test-and-Set

- `if (*lock == 0); *lock = 1;` **Must be a SINGLE INSTRUCTION!**

- The “`test`” and “set” must be **atomic**!



- **Hardware support** is required
  - `xchg` in x86 does exactly this
  - An atomic test-and-set operation

xchg: Atomic Value Exchange in x86

- **Exchange** content in `[memory]` with the value in `%reg` **atomically**

  `xchg [memory], %reg`

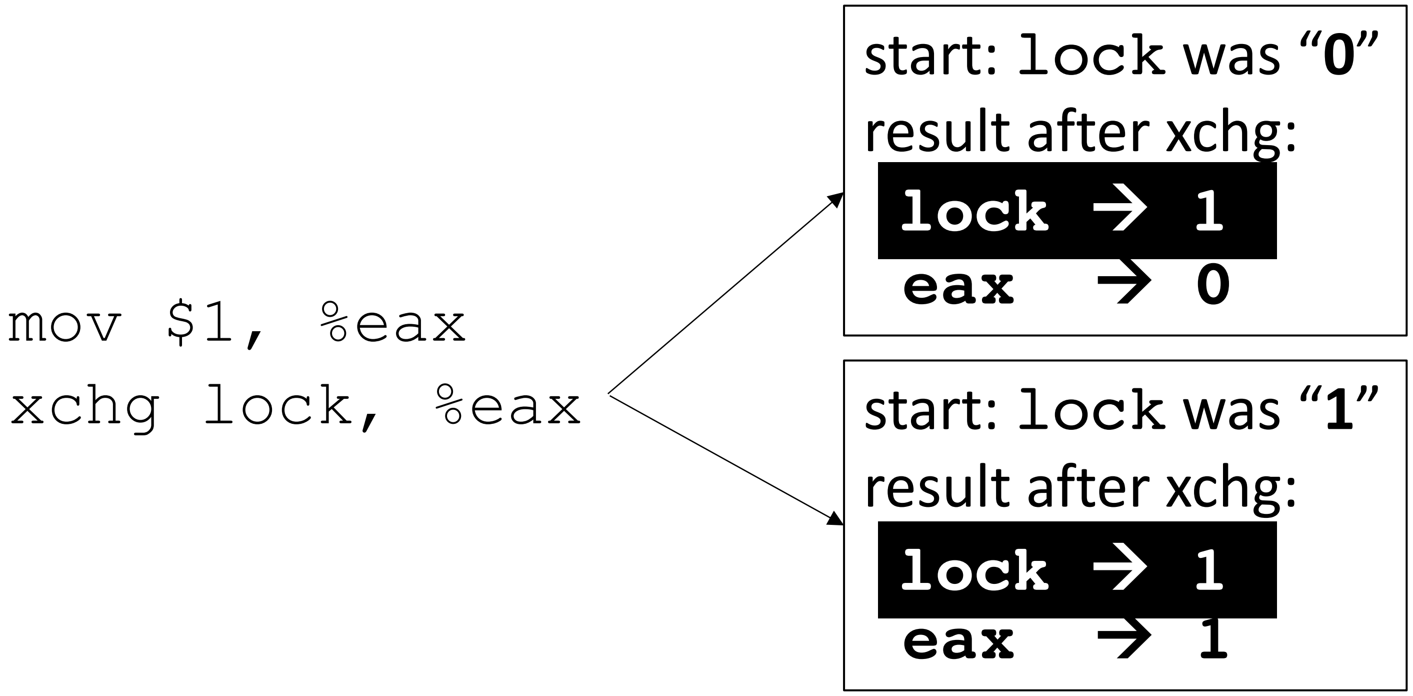
- How do we use it?

- Consider the following example:

`mov $1, %eax`	load the value "**1**" into the `eax` register
`xchg lock, %eax`	**exchange** that with the value in "`lock`"

How does **xchg** work?

- `xchg` **always sets** `lock` to "**1**"
- Returns **previous value of lock** into (`eax`) register



```
mov $1, %eax
xchg lock, %eax
```

start: `lock` was "**0**"

result after xchg:

```
lock  →  1
eax   →  0
```

start: `lock` was "**1**"

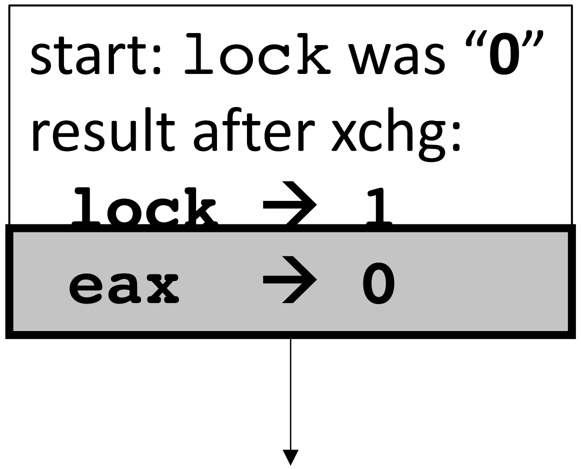
result after xchg:

```
lock  →  1
eax   →  1
```

only **one** thread will see **`lock == 0`**

hardware locking memory bus

Details



start: `lock` was “**0**”

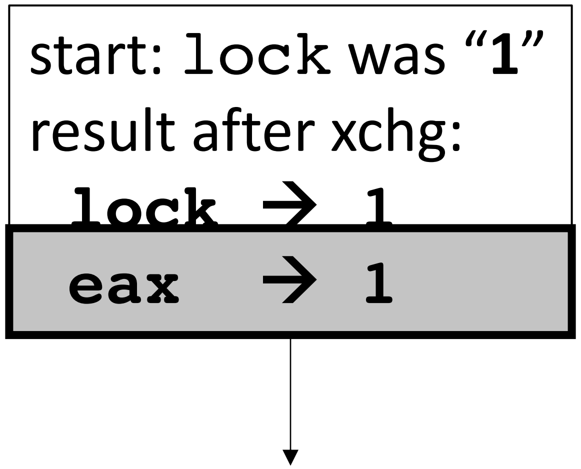
result after xchg:

`lock → 1`

`eax → 0`

`lock` was ‘**0**’

We **acquired** the lock!



start: `lock` was “**1**”

result after xchg:

`lock → 1`

`eax → 1`

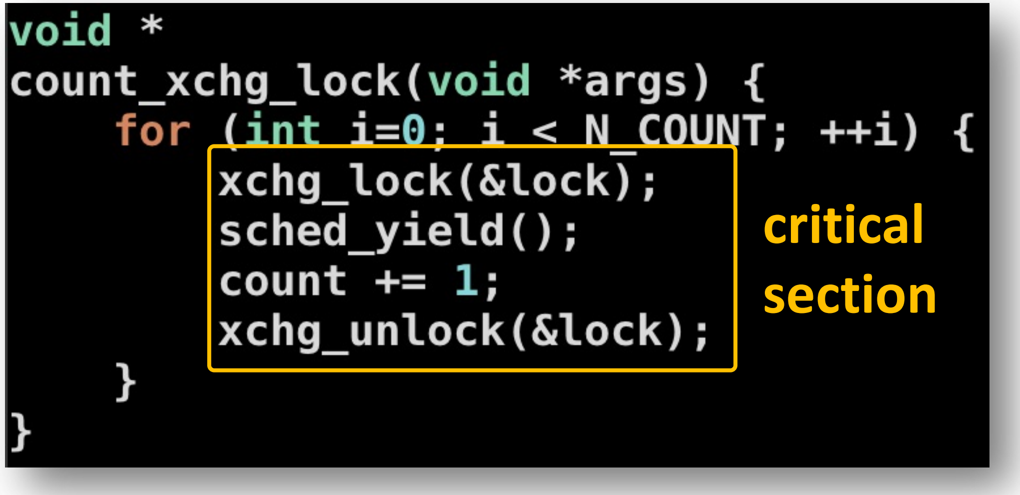
`lock` was ‘**1**’

We **did not acquire** the lock!

2nd Candidate: **`xchg_lock`** [using ‘xchg’]

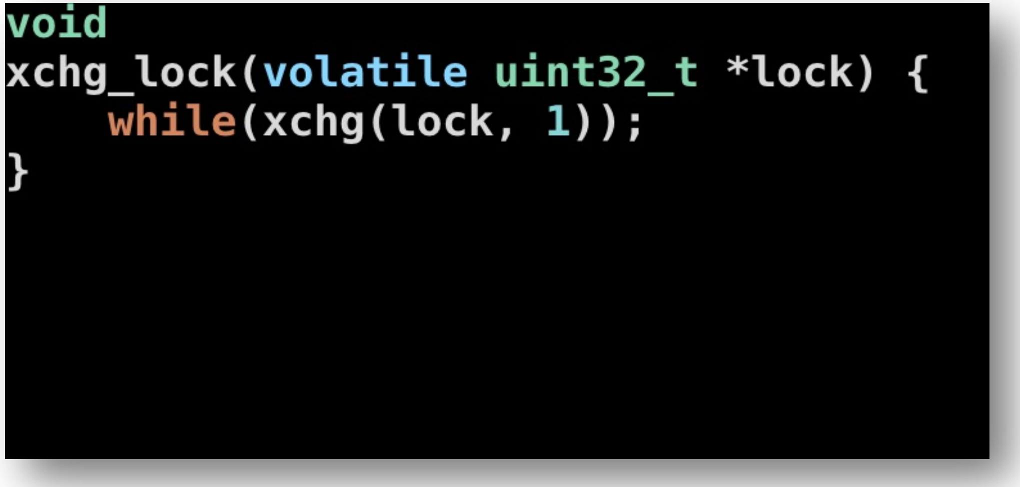
- **`xchg_lock`**
- Use atomic ‘`xchg`’ instruction
- Load and store values atomically
- Set value to `1’, and **compare return value**
  - If 0, then you can acquire the lock
  - If 1, `lock` is 1, you must wait

- **`xchg_unlock`**
- Use atomic ‘`xchg`’ instructon
- Set value to `**0**’
- **No need to check!**
  - You are the only thread in critical section!



```
void *
count_xchg_lock(void *args) {
    for (int i=0; i < N_COUNT; ++i) {
        xchg_lock(&lock);
        sched_yield();
        count += 1;
        xchg_unlock(&lock);
    }
}
```

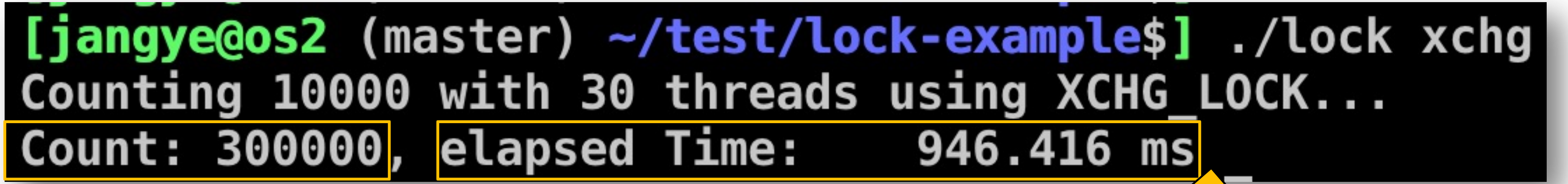
critical section



```
void
xchg_lock(volatile uint32_t *lock) {
    while(xchg(lock, 1));
}
```

2nd Candidate: **xchg_lock** Result

- **Consistent!**



```
[jangye@os2 (master) ~/test/lock-example$] ./lock xchg
Counting 10000 with 30 threads using XCHG_LOCK...
Count: 300000, elapsed Time:     946.416 ms
```

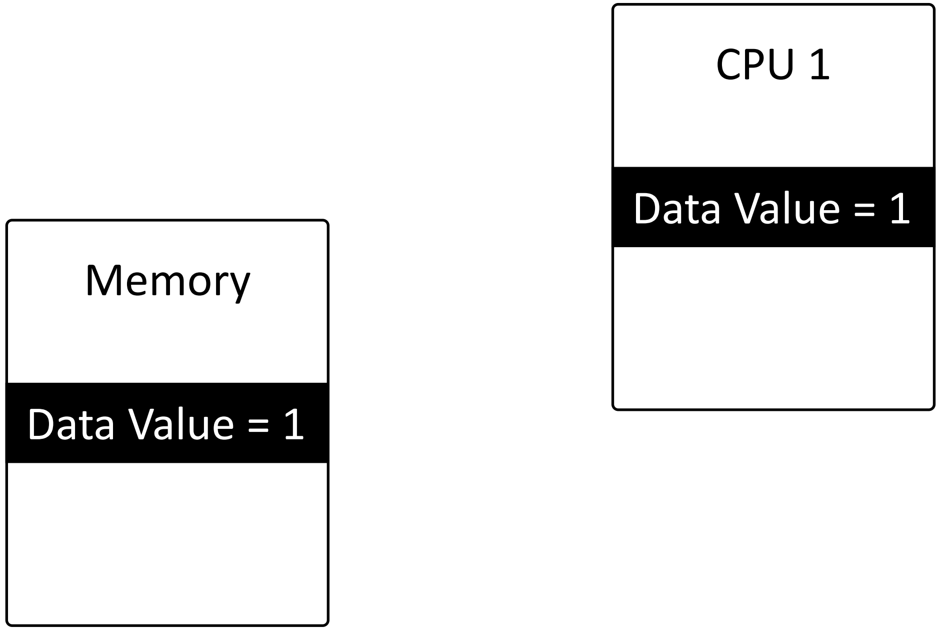
High overheads!

cache coherency!

https://gitlab.unexploitable.systems/root/lock-example

[You can run this by cloning the repo!]

Detour | Cache Coherency



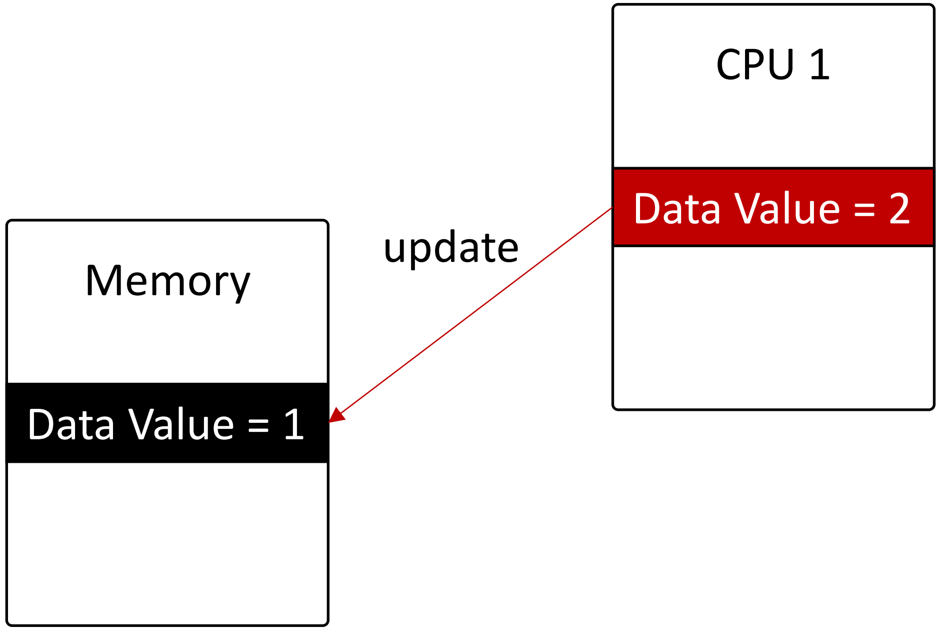
Memory

Data Value = 1

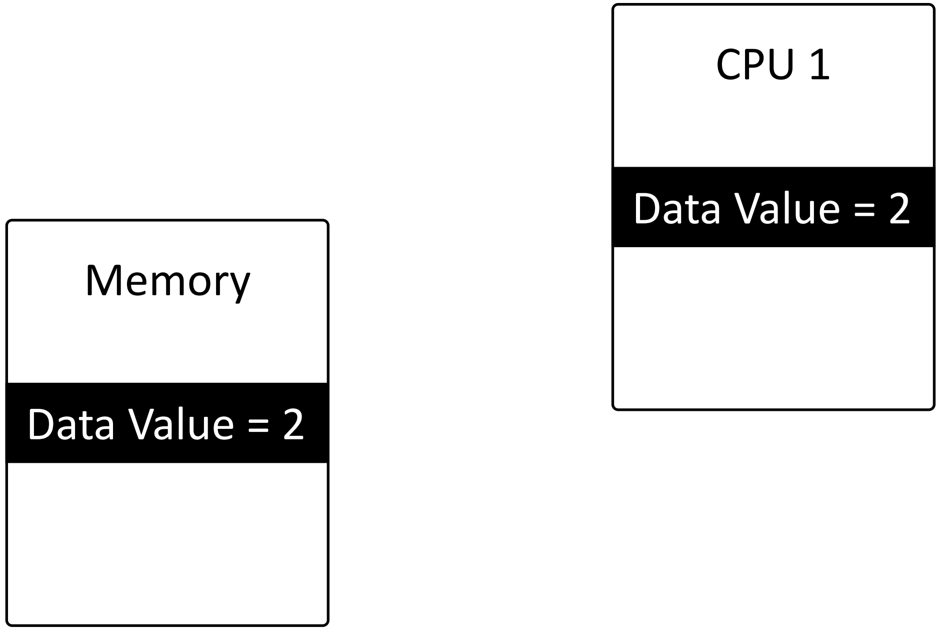
CPU 1

Data Value = 1

Detour | Cache Coherency



Detour | Cache Coherency



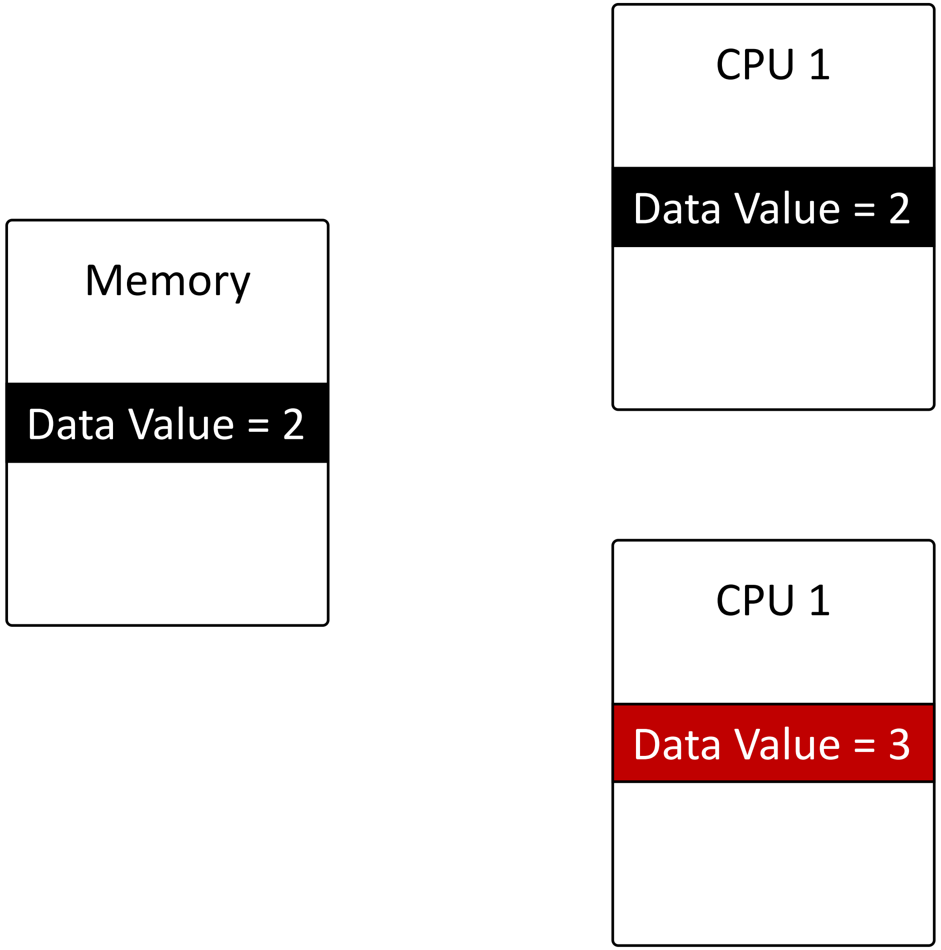
CPU 1

Data Value = 2

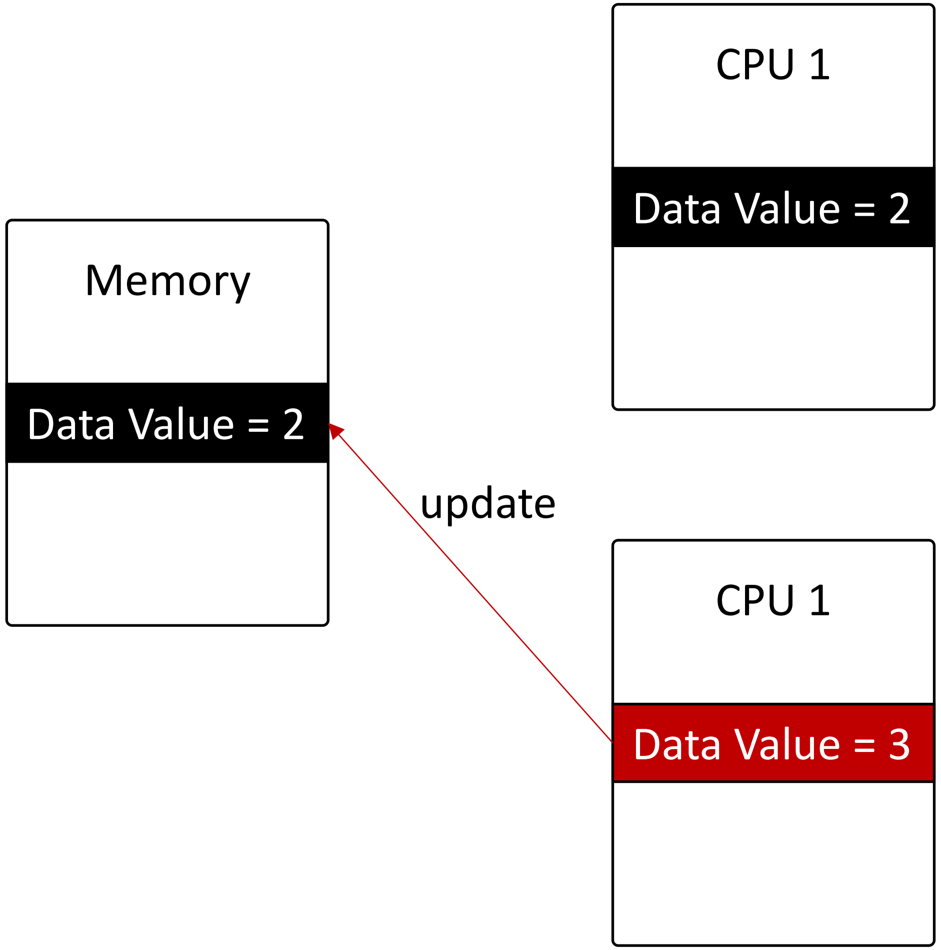
Memory

Data Value = 2

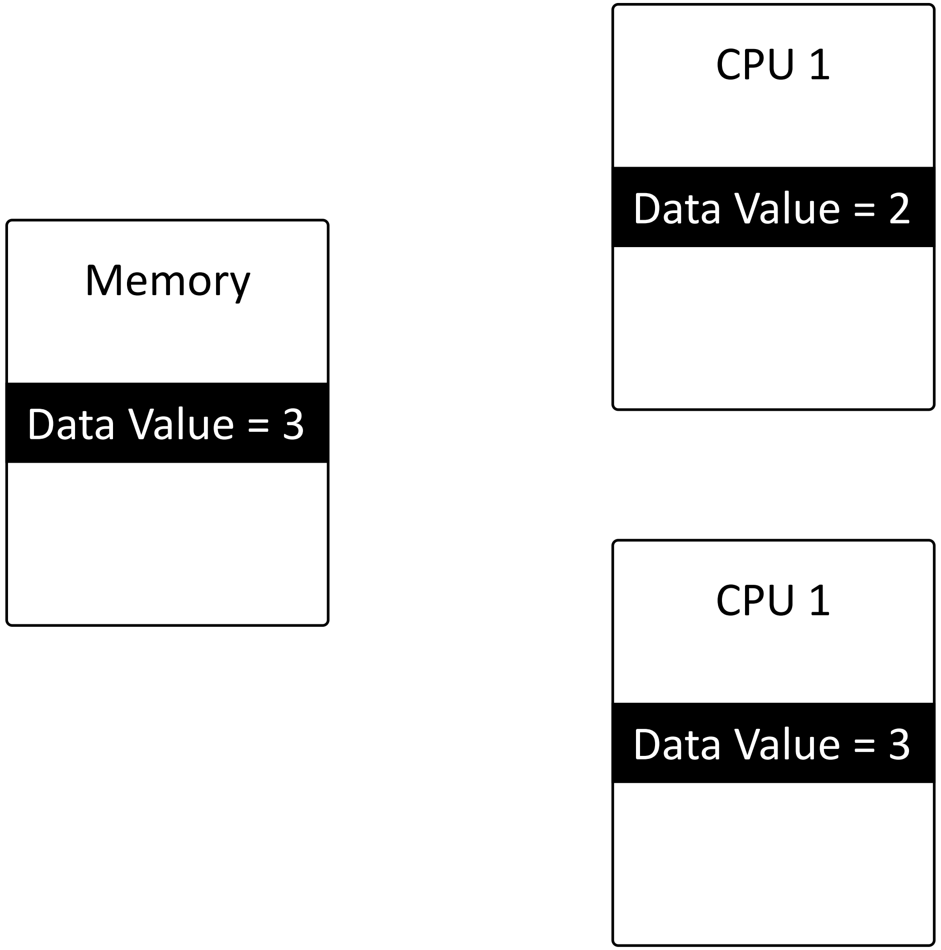
Detour | Cache Coherency



Detour | Cache Coherency



Detour | Cache Coherency



CPU 1

Data Value = 2

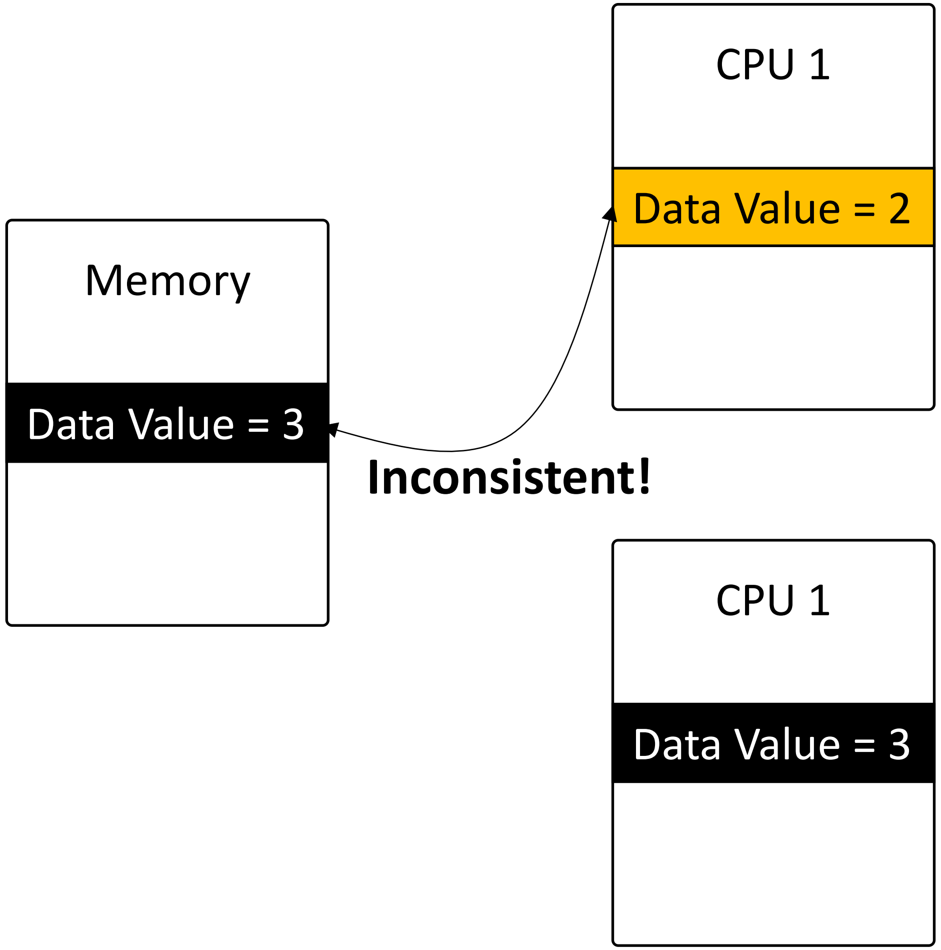
Memory

Data Value = 3

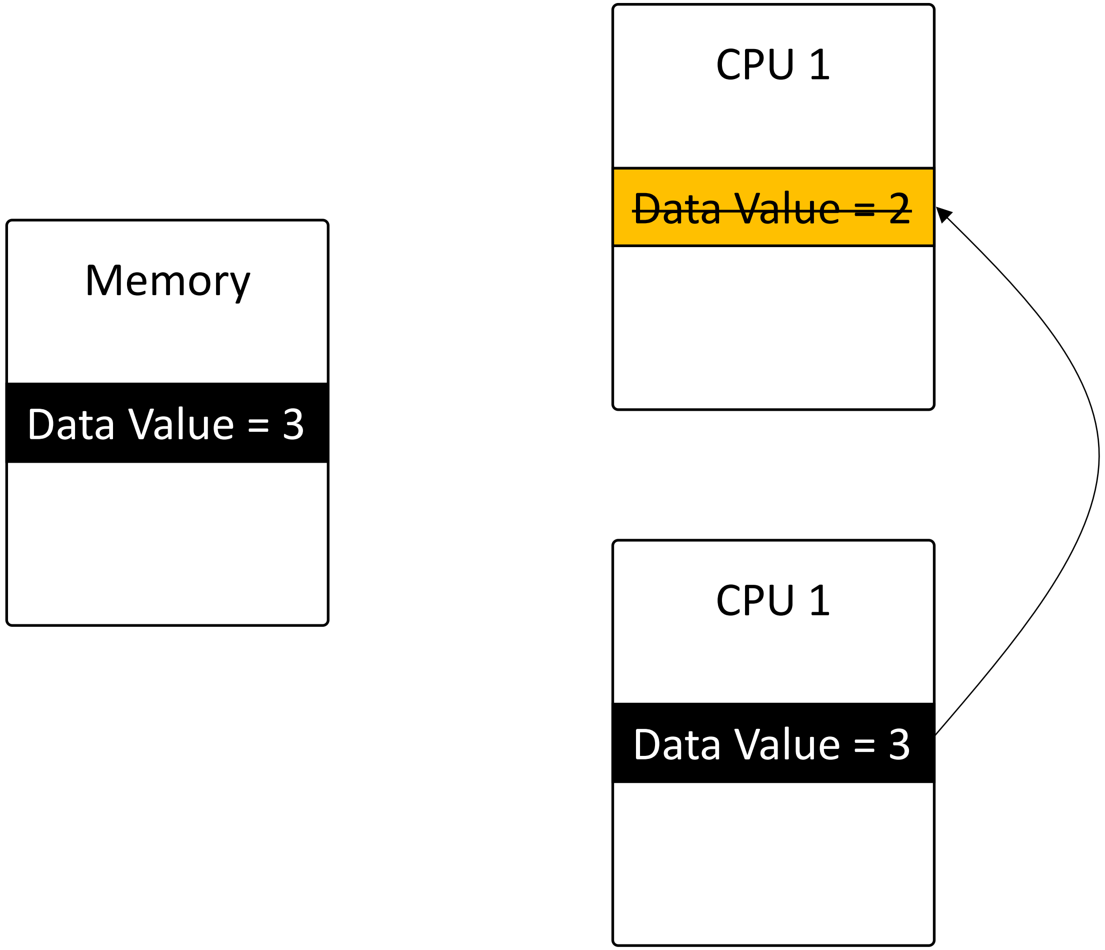
CPU 1

Data Value = 3

Detour | Cache Coherency

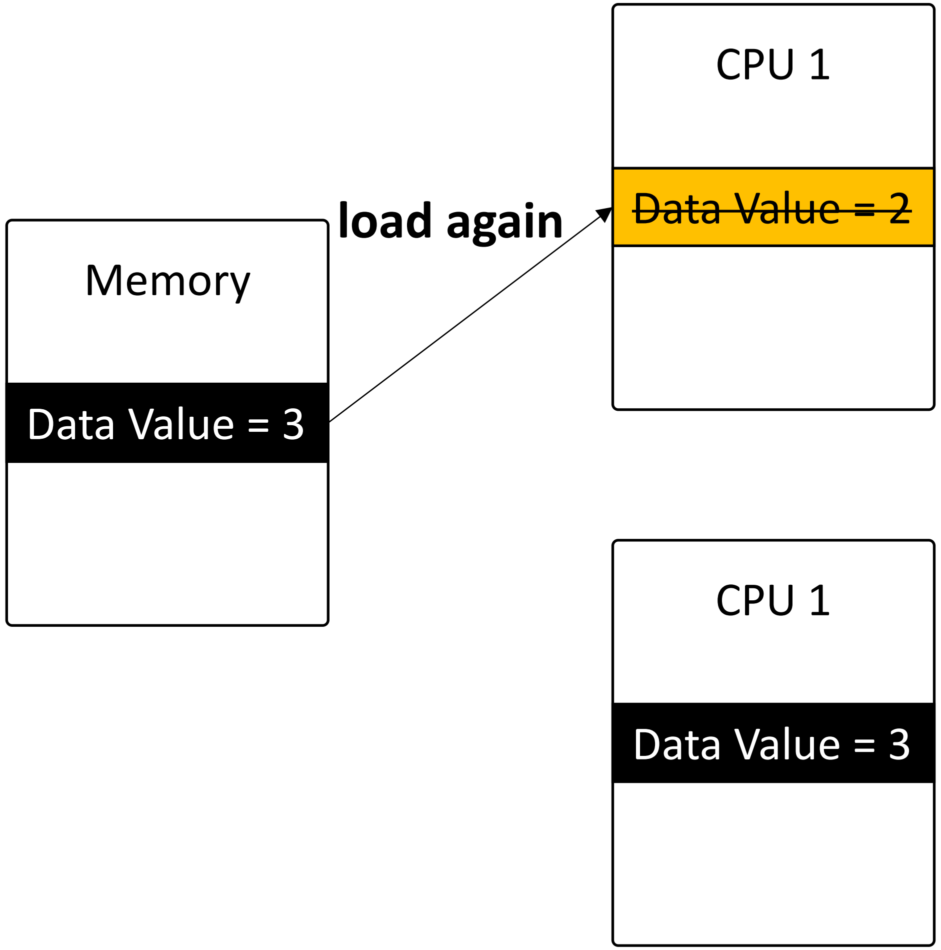


Detour | Cache Coherency

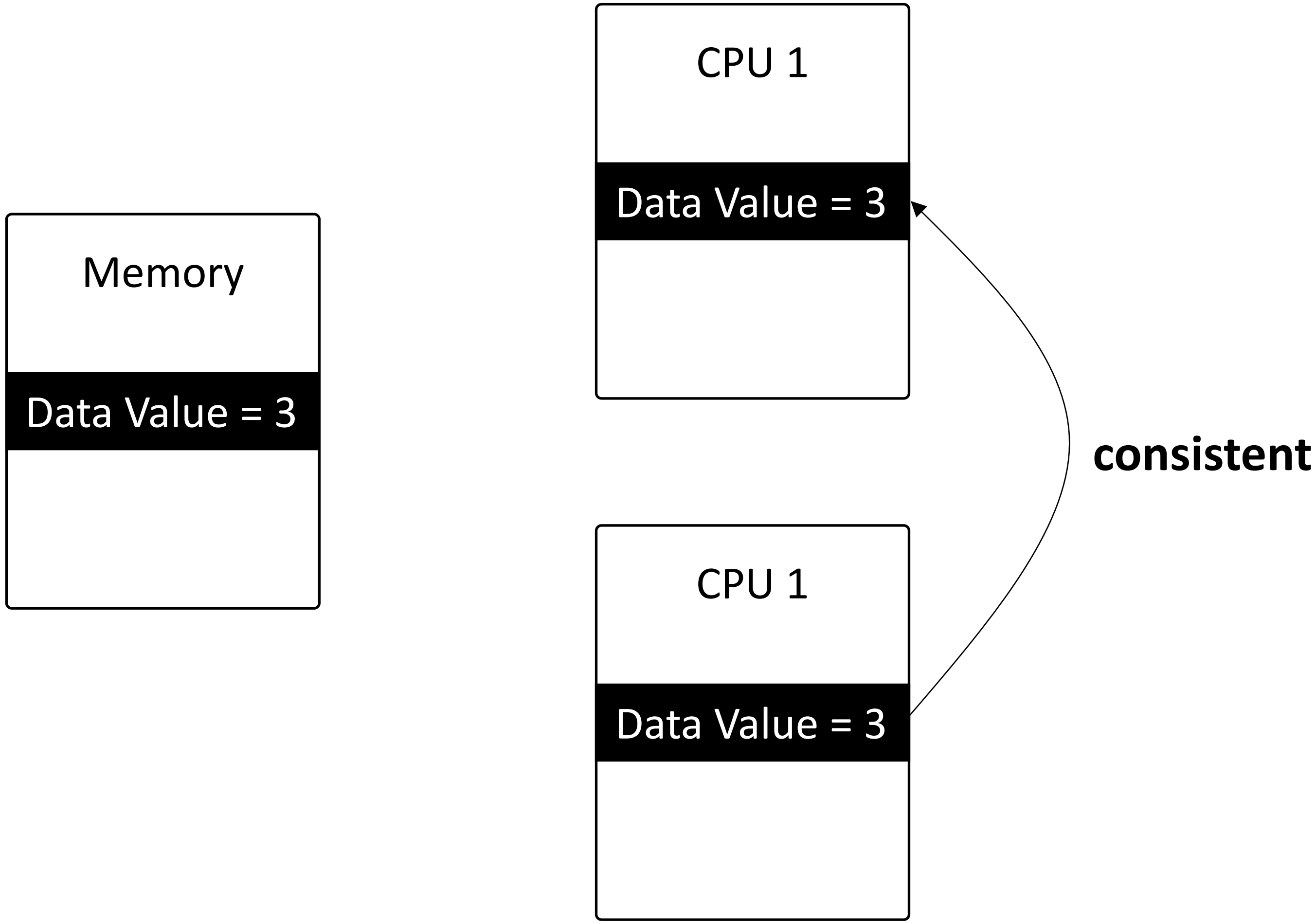


Invalidate!
Flush CPU1 L1 cache

Detour | Cache Coherency



Detour | Cache Coherency



Memory

Data Value = 3

CPU 1

Data Value = 3

CPU 1

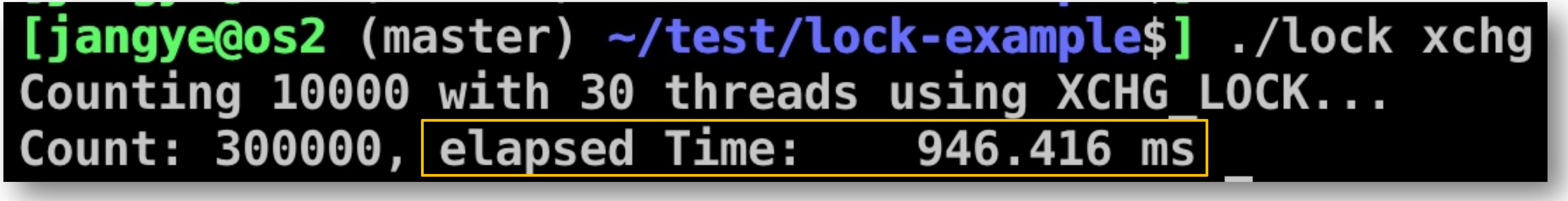
Data Value = 3

consistent

Back to xchg

- Atomic xchg instruction loads/stores data at the same time
  - There is no gap for race condition

- But it could **cause cache contention!**
  - Many threads update the same '**`lock`**' variable
  - Multiple CPUs cache '**`lock`**' variable
  - **Update to `lock` invalidates cache!**



```
[jangye@os2 (master) ~/test/lock-example$] ./lock xchg
Counting 10000 with 30 threads using XCHG_LOCK...
Count: 300000, elapsed Time:     946.416 ms
```