CS444/544
Operating Systems II

Lecture 14

Lock and Synchronization

5/20/2024

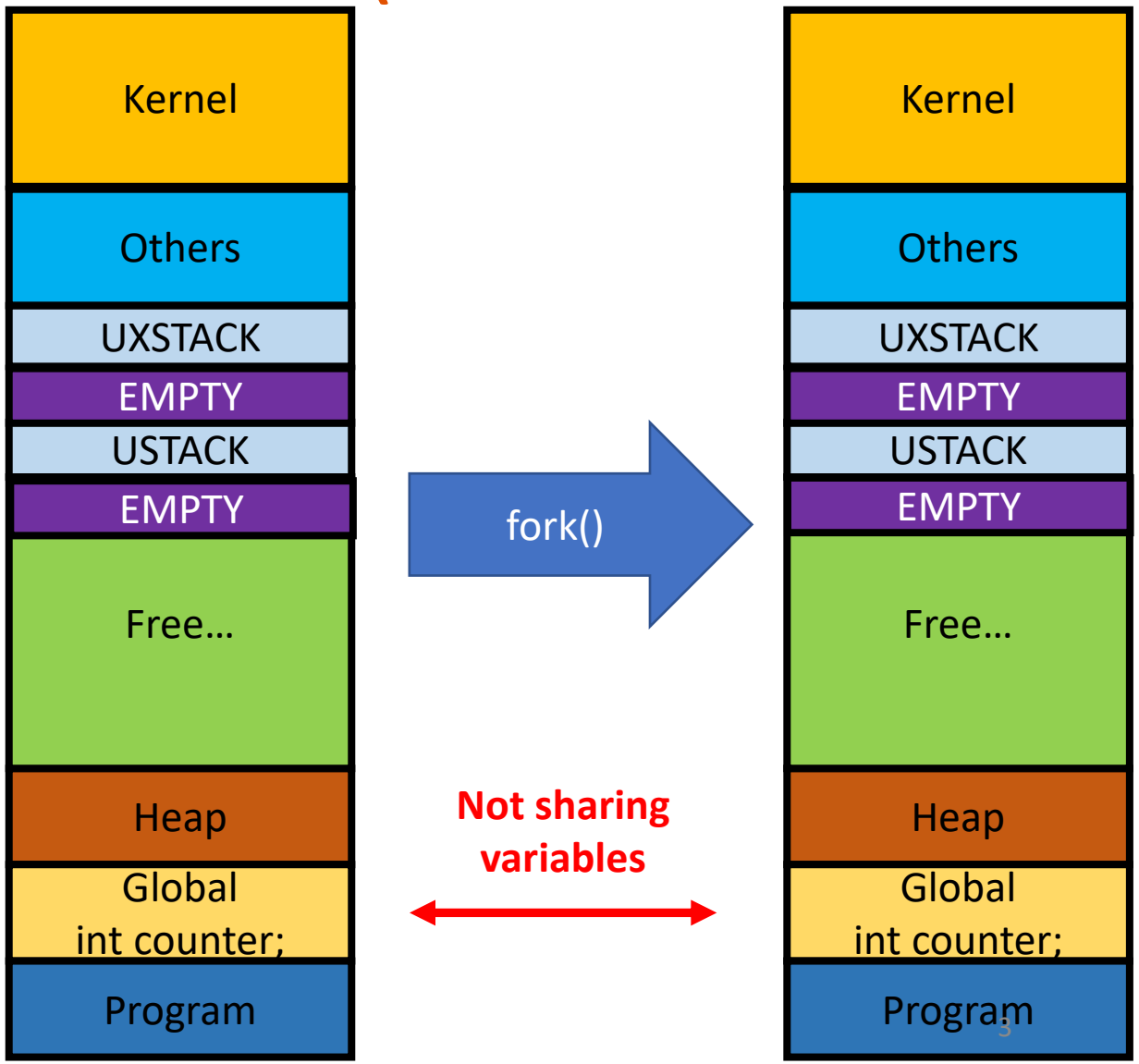
Acknowledgement: Slides drawn heavily from Yeongjin Jiang

Oregon State University

Odds and Ends

- Lab 3 due today's (5/20) midnight

Process (Environment in JOS)



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

```c
#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

pthread_create()

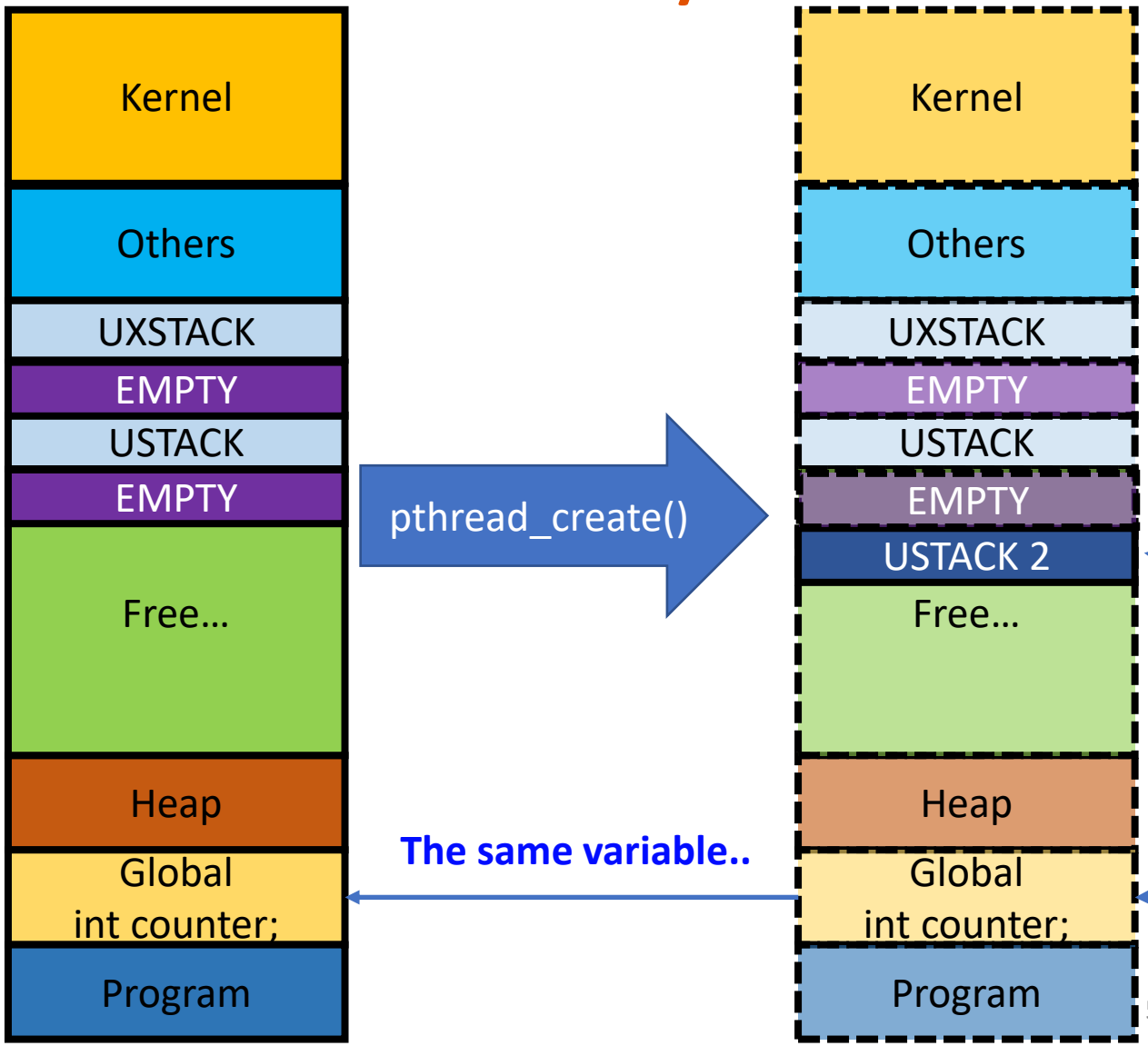

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

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);
}
```

Add a new stack!

Adding value..

The same variable..

Concurrency Issue..




```c
#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
```

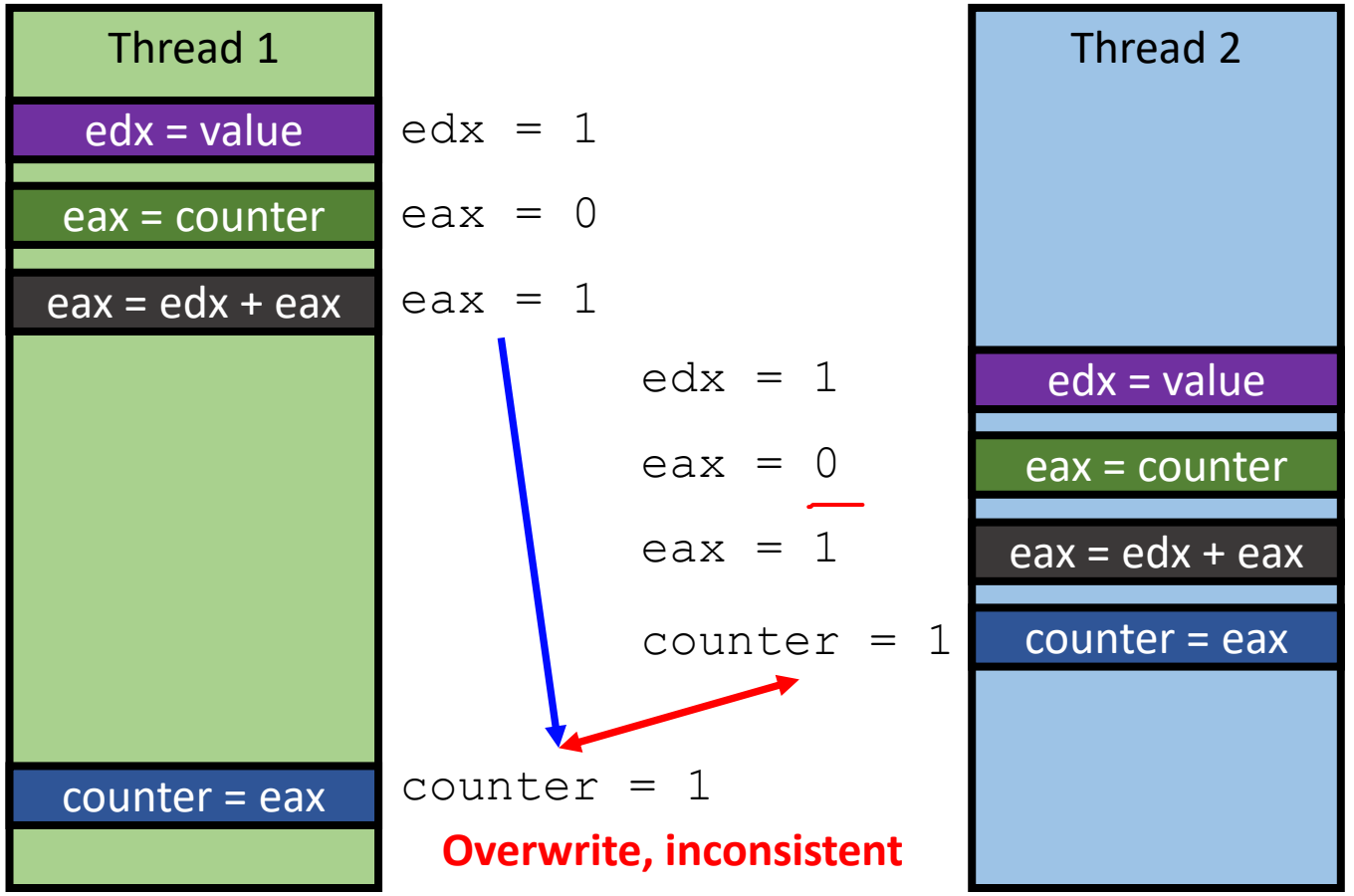
```
Child: 975822
Parent: 1081479
```

Why not 2000000?

Data Race Example (Race cond.)

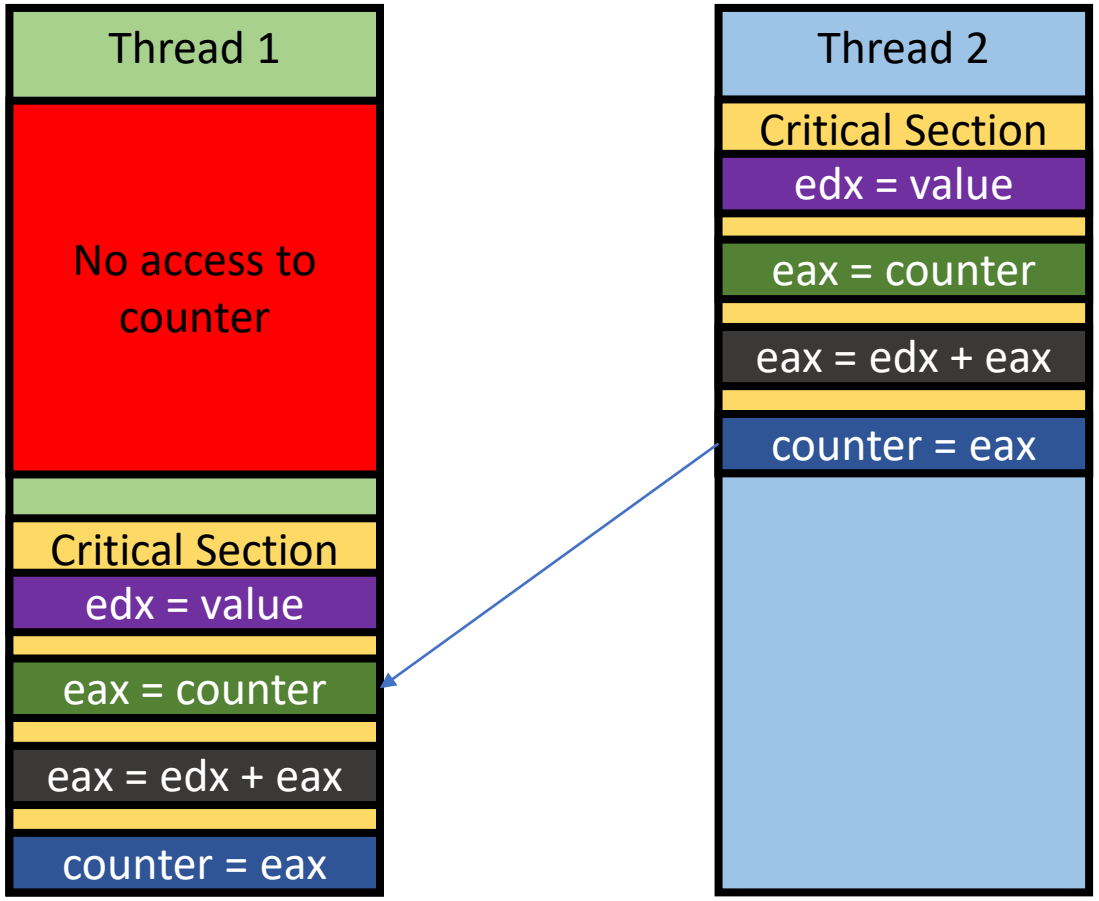
- counter += value
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`

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



How to Prevent Data Racing?

- Mutual Exclusion / Critical Section
  - Combine multiple instructions as a chunk
  - Let only one chunk execution runs
  - Block other executions



Caveat: Apply Mutex only if required

- Mutex can synchronize multiple threads and yield consistent result
  - No read before previous thread stores the shared data
- Making the entire program as critical section is meaningless
  - Running time will be the same as single-threaded execution
- Apply critical section as short as possible to maximize benefit of having concurrency
  - Non-critical sections will run concurrently!

Enabling Mutual Exclusion

- `cli`, in a single processor computer
  - Clear interrupt bit
- CPU will never get interrupt until it runs `sti`
  - Set interrupt bit
- There will be no other execution
  - Any problems?
  - Multi CPU?
  - `cli/sti` available in Ring 0

- counter += value
  - **`cli`**
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`
  - **`sti`**

Mutex (Mutual Exclusion)

- Lock
  - Prevent others enter the critical section
- Unlock
  - Release the lock, let others acquire the lock

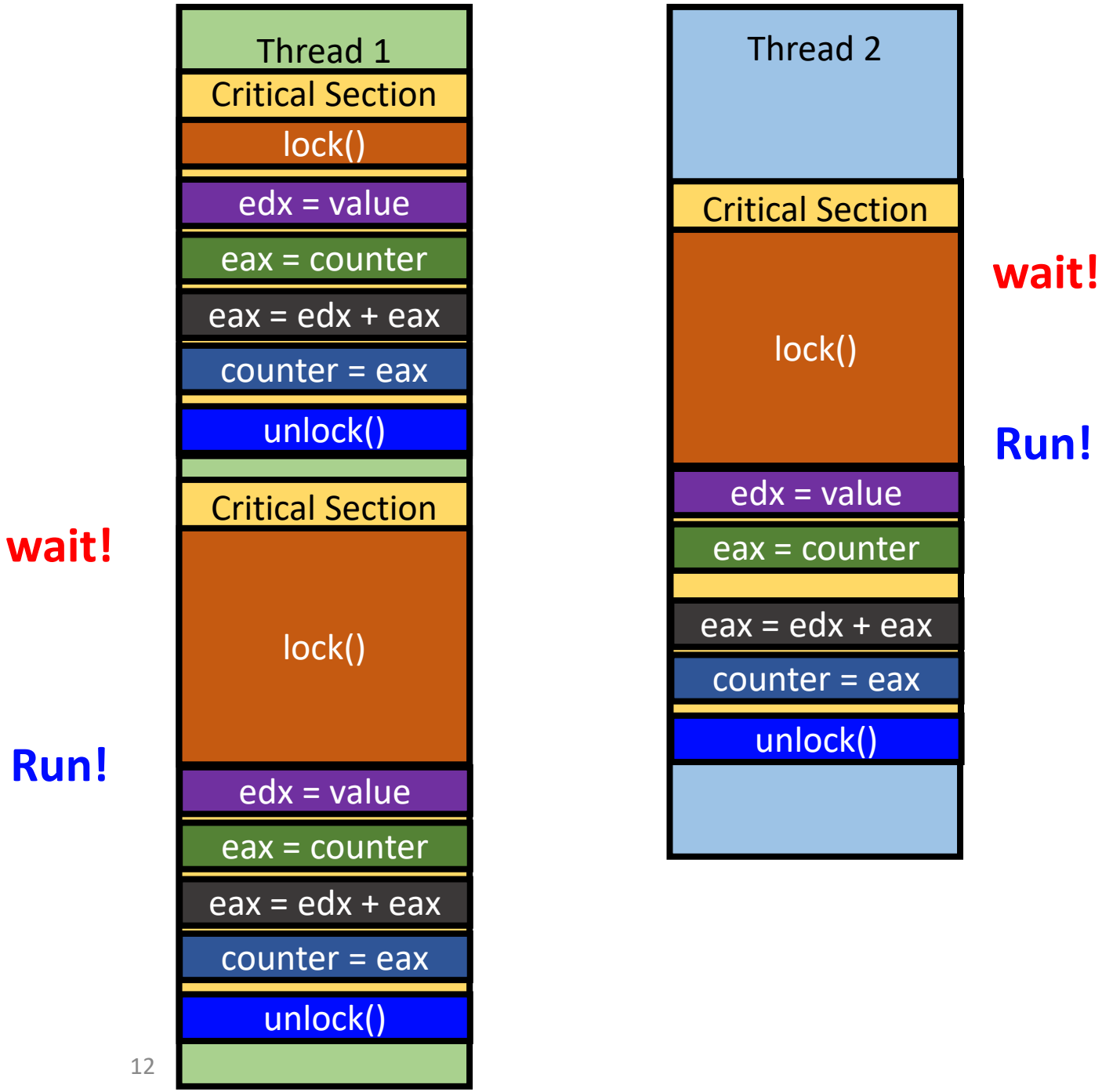
- counter += value
  - **lock()**
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`
  - **unlock()**

Mutex (Mutual Exclusion)

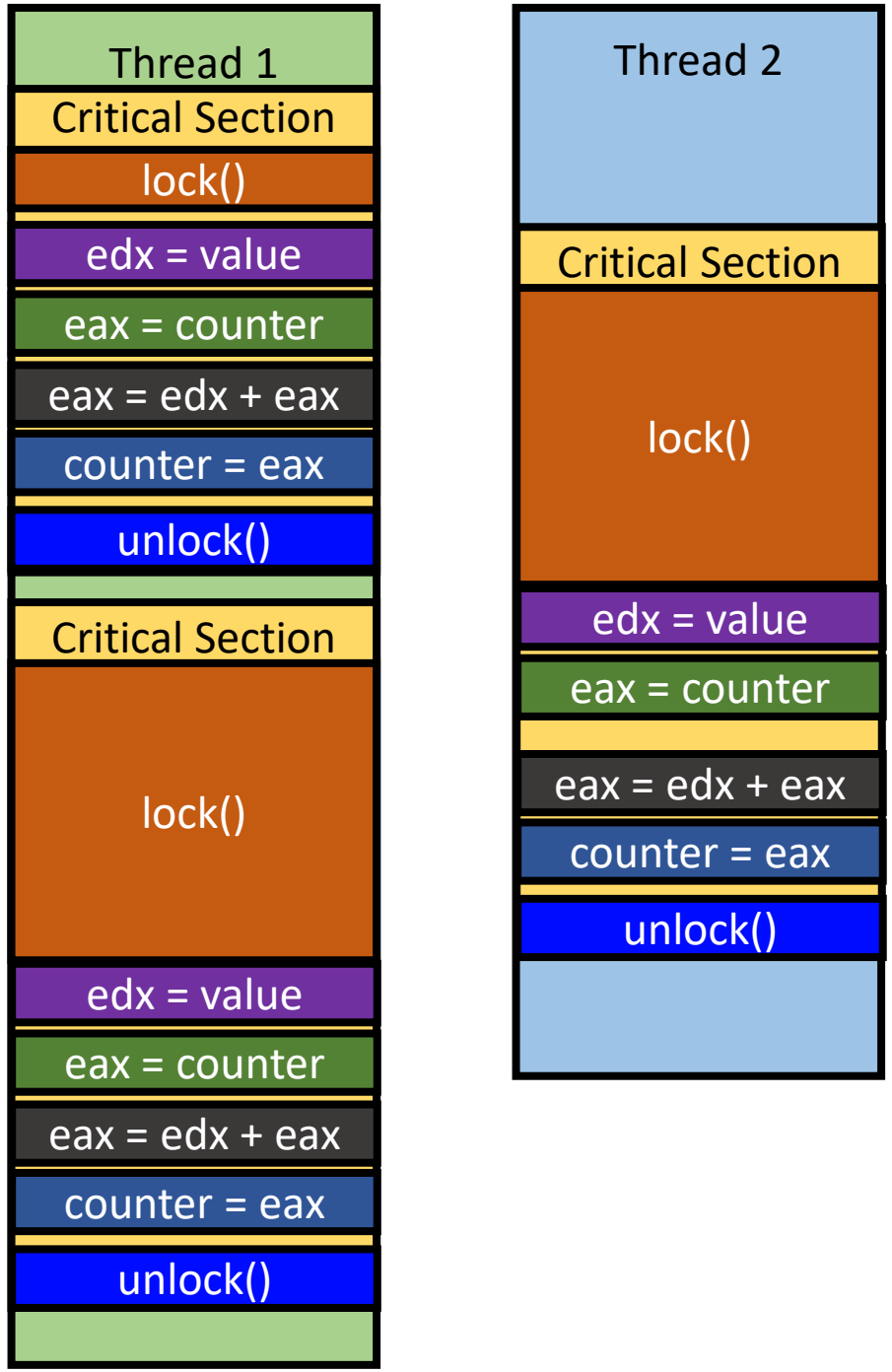
- Lock
  - Prevent others enter the critical section
- How?
  - Check if any other execution in the critical section
    - If it is, wait; busy-waiting with while()
  - If not, acquire the lock!
    - Others cannot get into the critical section
  - Run critical section
  - Unlock, let other execution know that I am out!

- counter += value
  - **`lock()`**
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`
  - **`unlock()`**

Mutex Example

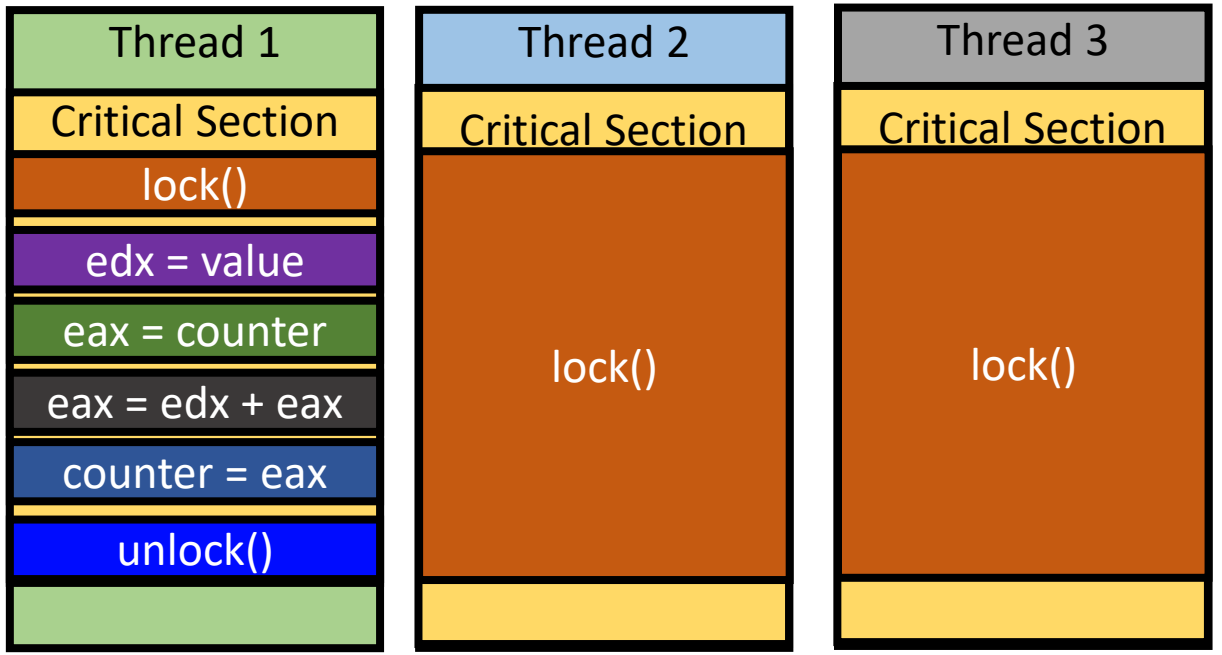


Mutex Example



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

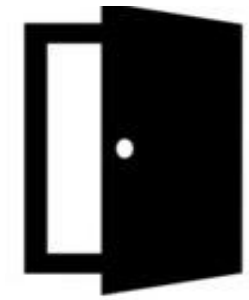
- Only one can run in critical section
- Others must wait!
  - Until nobody runs in critical section
- How can we create such
  - Lock() / Unlock() ?



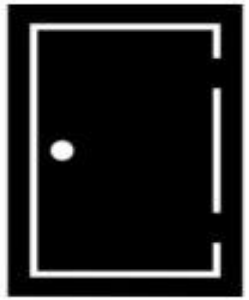
How Can We Create Lock/Unlock for Mutex? -- Spinlock

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

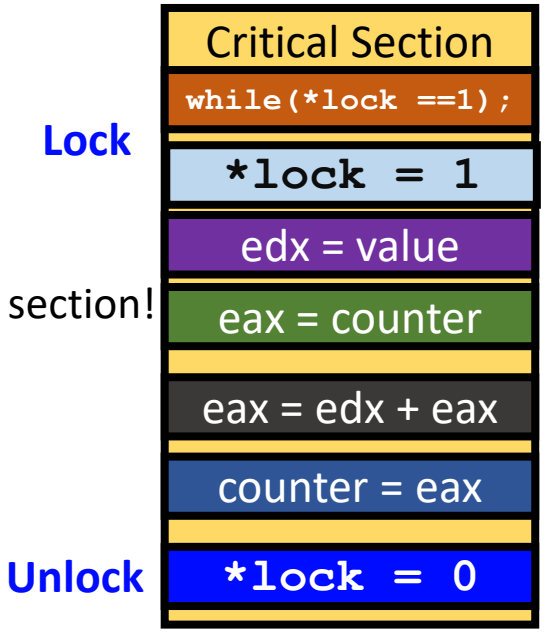
- Spinlock
  - Run a loop to check if critical section is empty
  - Set a lock variable, e.g., `lock`
  - Lock semantic
    - Nobody runs critical section if *`lock` == 0, so one can run the section
      - At the start of the section, set *`lock` = 1
    - Somebody runs in critical section if *`lock` == 1, so one must wait
  - lock(`lock`)
    - Wait until lock becomes 0, e.g., `while(*lock == 1);`
      - Then, if *lock == 0, break the loop, meaning nobody is running in the critical section!
    - set *`lock` = 1
  - unlock(lock)
    - Set *`lock` = 0



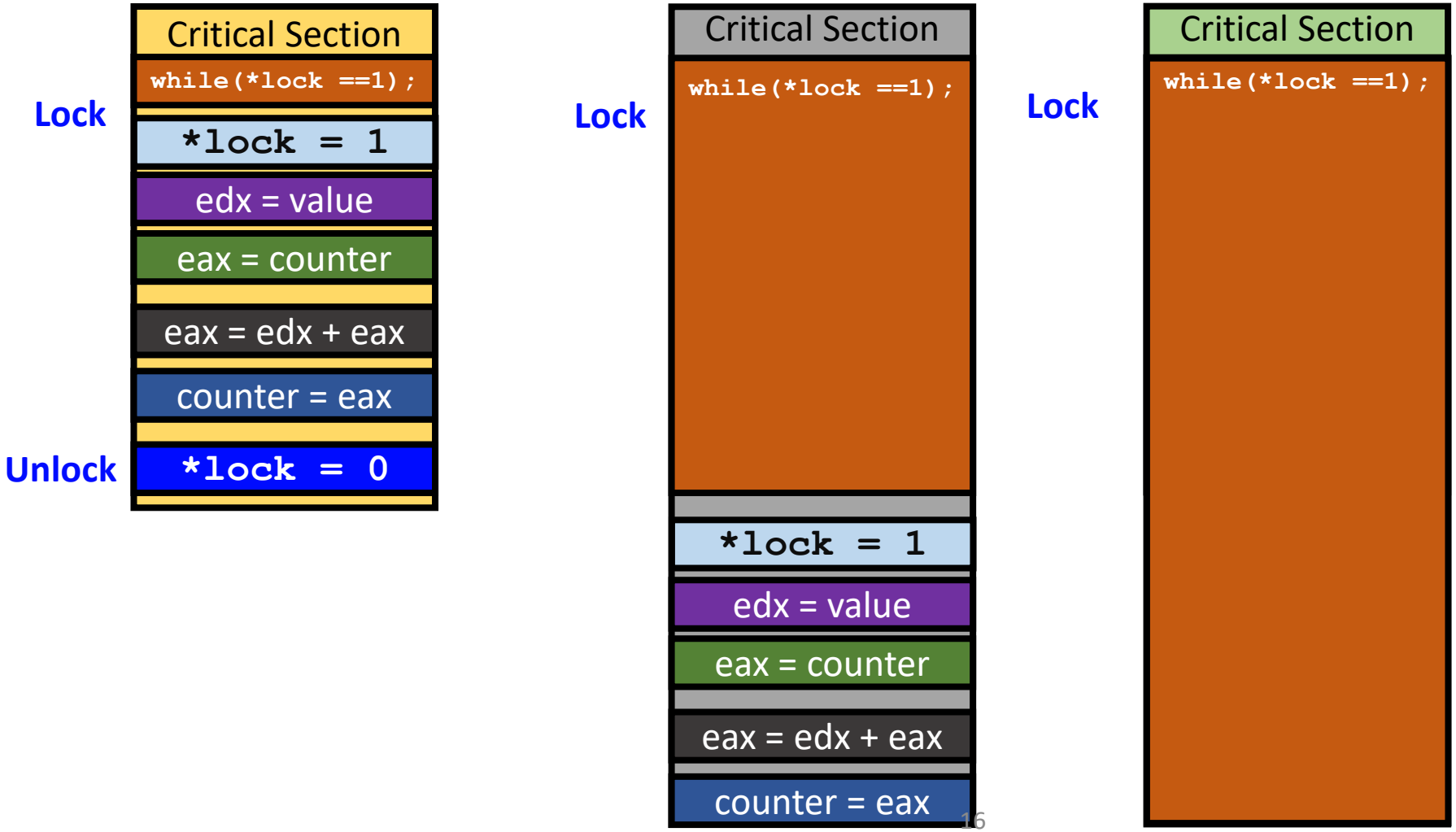
***lock == 0**



***lock == 1**



Spinlock


Critical Section
while(*lock ==1);
Lock
*lock = 1
edx = value
eax = counter
eax = edx + eax
counter = eax
Unlock
*lock = 0
Critical Section
while(*lock ==1);
Lock
*lock = 1
edx = value
eax = counter
eax = edx + eax
counter = eax
Critical Section
while(*lock ==1);
Lock

Spinlock Examples

- wget https://classes.engr.oregonstate.edu/eecs/spring2024/cs444-001/lock-example-master.zip
- unzip lock-example-master.zip
- Run 30 threads, each count upto 10000

- Build code
  - $ make

```
os2 ~/cs444/s21/lock-example-master 146% make
gcc -o lock lock.c -std=c99 -g -Wno-implicit-function-declaration -O2 -lpthread
```

Lock Example

- List of example
  - $ ./lock no # using no lock at all
  - $ ./lock bad # using a bad lock implementation
  - $ ./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

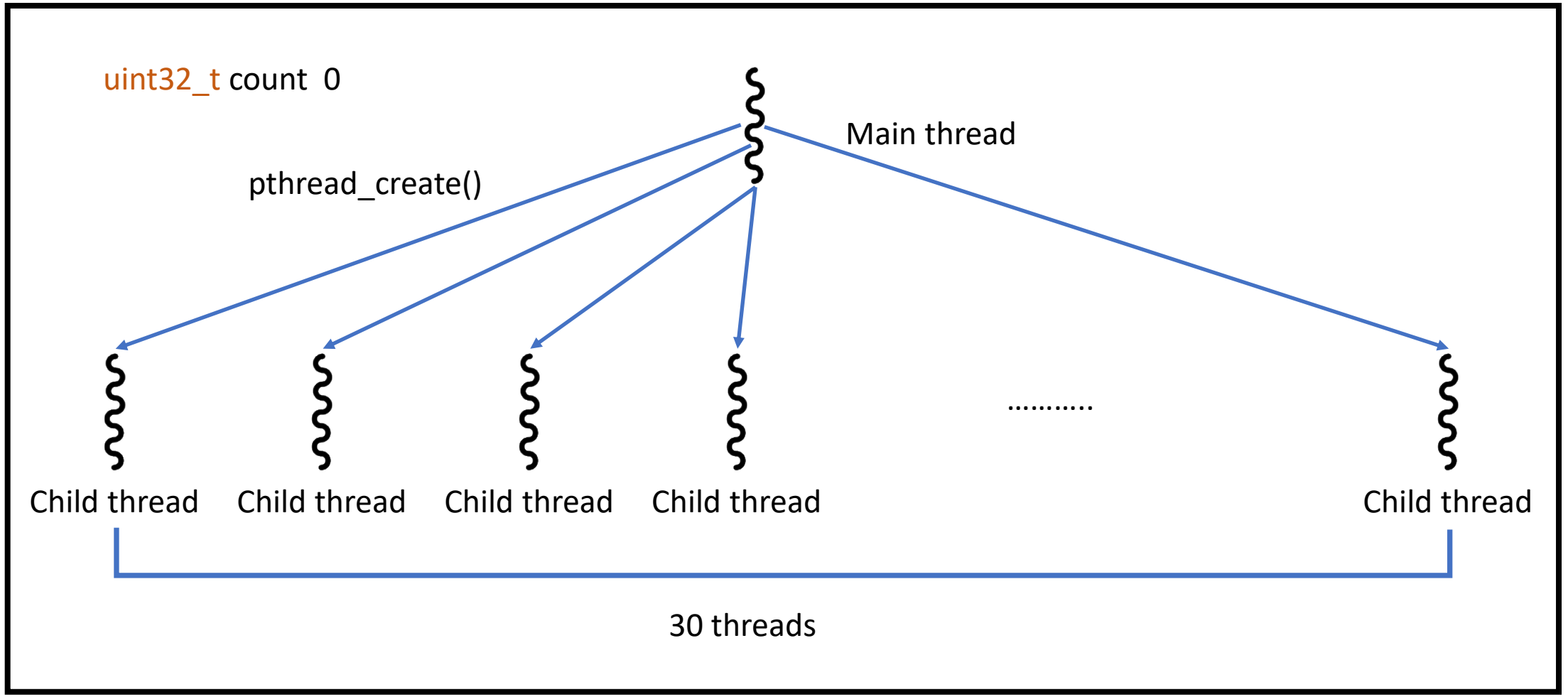
Spinlock Examples

- 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

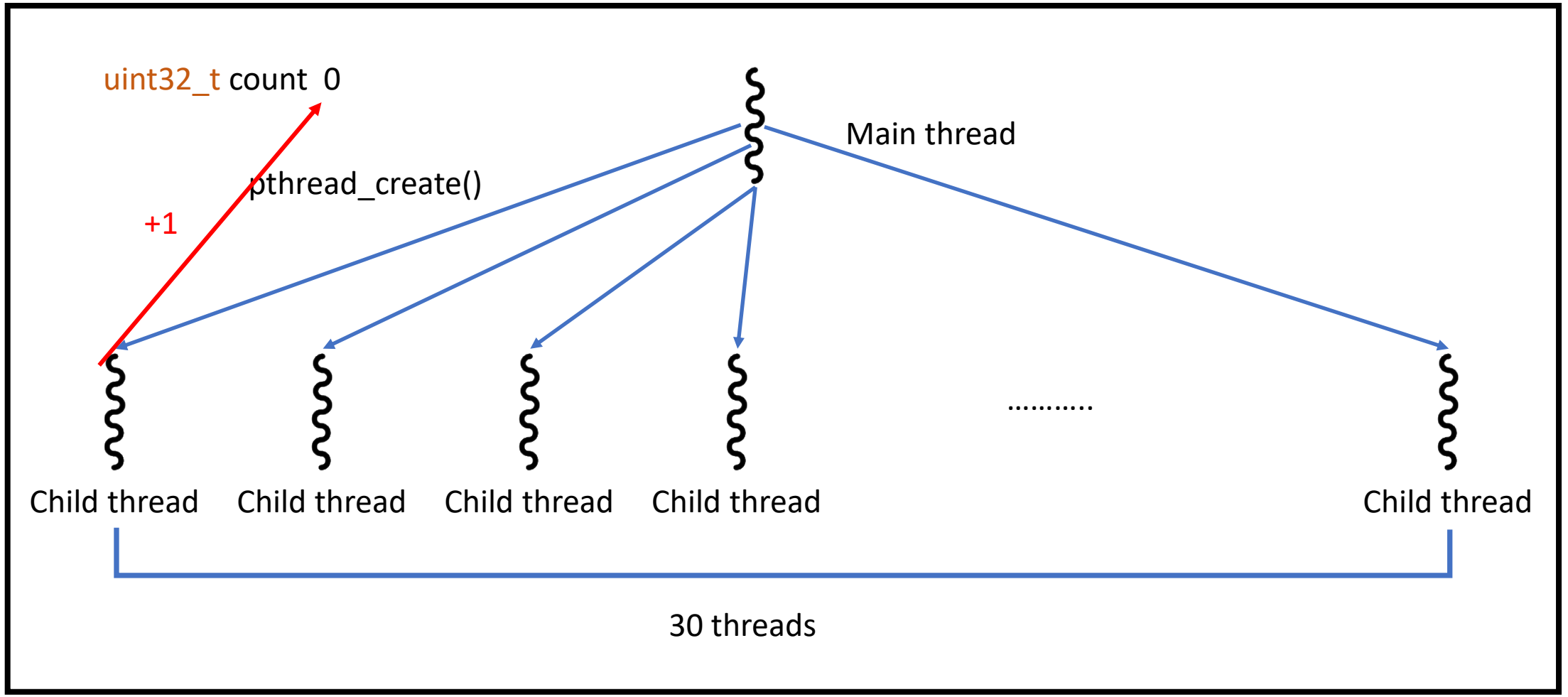
```
os2 ~/cs444/s21/lock-example-master 147% ./lock xchg
Counting 10000 with 30 threads using XCHG_LOCK...
Count: 300000, elapsed Time:    993.120 ms
```

```
os2 ~/cs444/s21/lock-example-master 148% ./perf-lock.sh xchg
Counting 10000 with 30 threads using XCHG_LOCK...
Count: 300000, elapsed Time:    877.739 ms

 Performance counter stats for './lock xchg':

       15,605,097      L1-dcache-load-misses:u

      0.881950454 seconds time elapsed

     20.486671000 seconds user
      0.090785000 seconds sys
```

How lock-example runs?

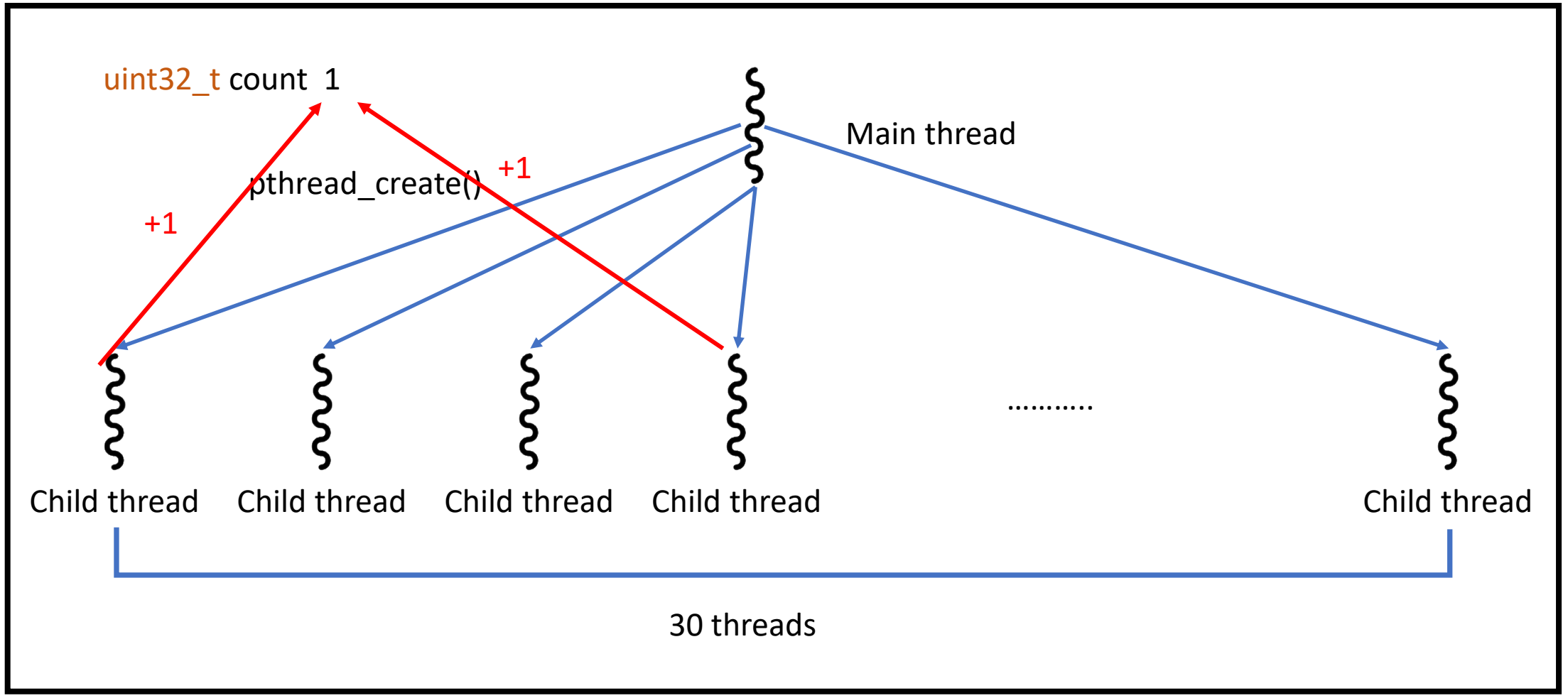


How lock-example runs?



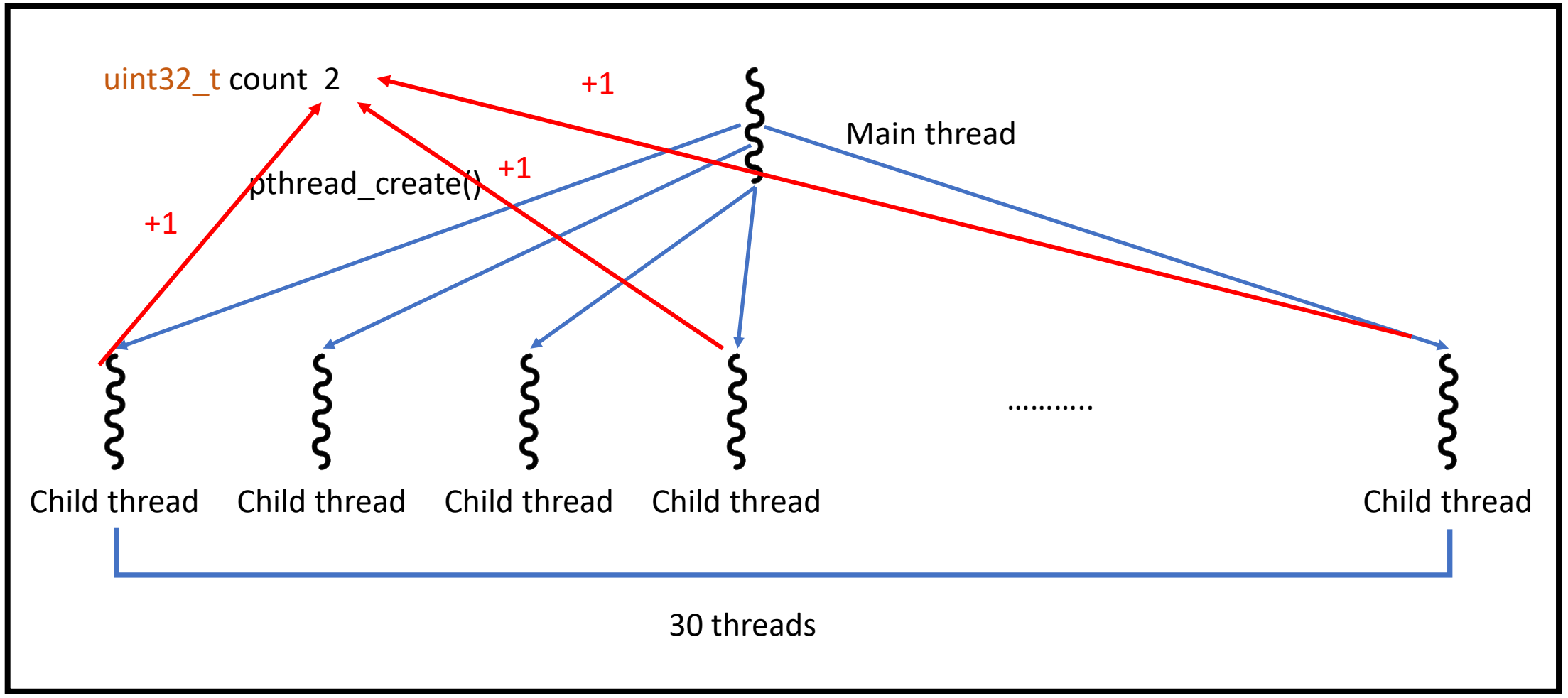
Each thread will increase count by 1 for 10,000 times

How lock-example runs?



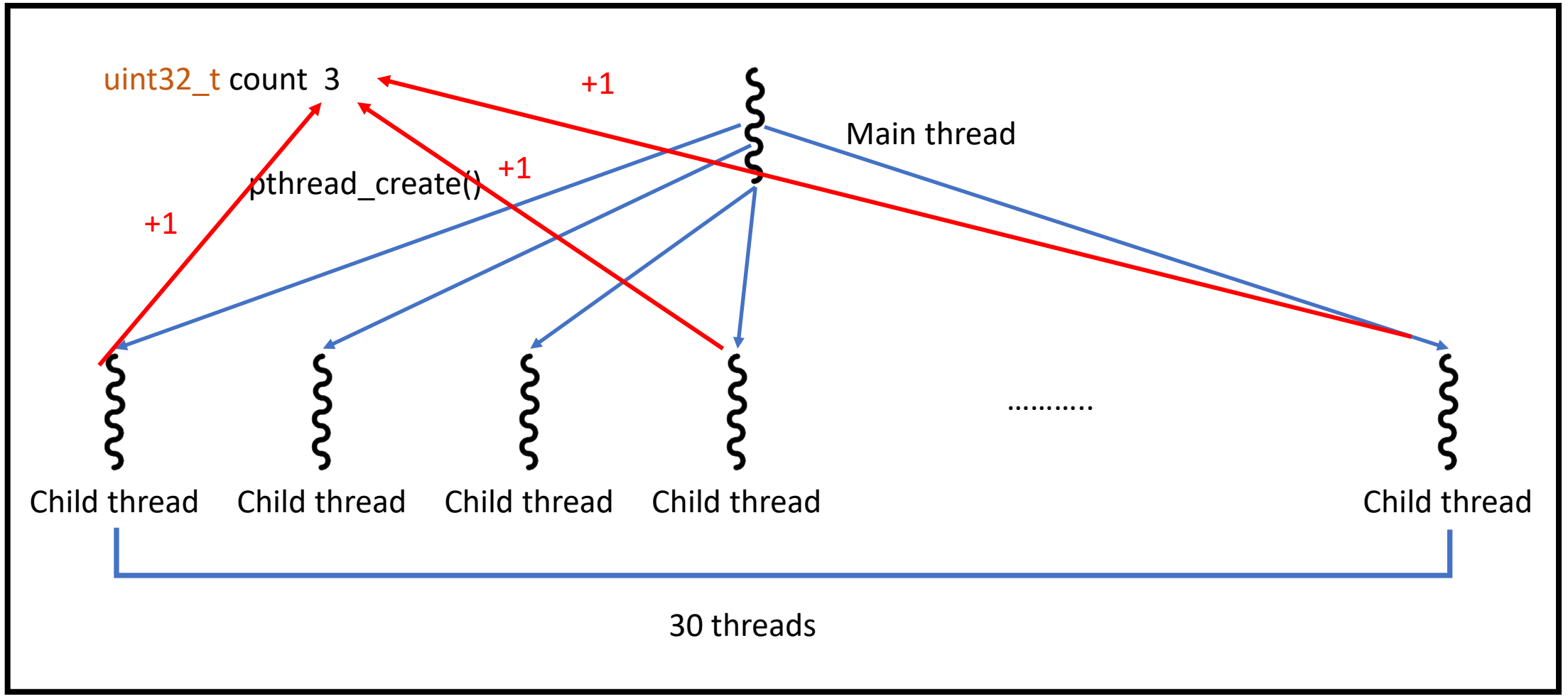
Each thread will increase count by 1 for 10,000 times

How lock-example runs?



Each thread will increase count by 1 for 10,000 times

How lock-example runs?



Each thread will increase count by 1 for 10,000 times

lock.c

```
os2 ~/cs444/s21/lock-example-master 153% ls -l
total 264
-rwxrwx---. 1 songyip upg56220 27360 May 20 11:18 lock
-rw-rw----. 1 songyip upg56220  5617 May 21  2020 lock.c
-rw-rw----. 1 songyip upg56220   187 May 21  2020 Makefile
-rwxr-xr-x. 1 songyip upg56220    55 May 21  2020 perf-lock.sh
```

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

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

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

```
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 and Wait with join()

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

Race condition may happen

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

Each thread Count 10,000

1st Candidate: bad_lock

- What will happen if we implement lock
  - As bad_lock / bad_unlock?

- bad_lock
  - Wait until lock becomes 0 (loops if 1)
  - And then, set lock as 1
    - Because it was 0, we can set it as 1
  - Others must wait!

- bad_unlock
  - Just set *lock as 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;
}
```

Can pass this if lock=0

Sets lock=1 to block others

Sets lock=0 to release

1st Candidate: bad_lock Result

- Inconsistent!

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

WHY?

Race Condition in bad_lock

- There is a room for race condition!

LOAD

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

STORE

Race condition may happen

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

thread 1

Load value 0 from lock
Compare that to 1
Break the loop
Store 1 to lock

thread 2

Load value 0 from lock
Compare that to 1
Break the loop
Store 1 to lock

Both threads may enter the critical section!

How Can We Avoid Race Condition on Loading/Updating a Value?

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

LOAD

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

Race condition may happen

STORE

- If we run multiple instructions for
  - Loading a value
  - Storing a value
- Then we must face race condition…

Atomic `Test-and-Set`

- We need a way to test
  - if lock == 0
- And we would like to set
  - lock = 1
- And do this atomically

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

Not like these four instructions…

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

xchg: Atomic Value Exchange in x86

- `xchg [memory], %reg`
  - Exchange the content in `[memory]` with the value in `%reg` atomically
- E.g.,
  - `mov $1, %eax`
  - `xchg $lock, %eax` **Swap lock and eax atomically**
- This will set `%eax` as the value in `lock`
  - `%eax` will be `0` if `lock==0`, will be `1 if lock==1`
- At the same time, this will set `lock = 1` (the value was in `%eax`)
- CPU applies 'lock' at hardware level (cache/memory) to do this
  - Hardware guarantees no data race when running `xchg`

xchg: Atomic Value Exchange in x86

- E.g.,
  - `mov $1, %eax`
  - `xchg $lock, %eax` **Swap lock and eax atomically**
- This will set `%eax` to the value in `lock`
  - `%eax` will be `0` if `lock==0`, will be `1 if lock==1`
- How can we determine if a thread acquired the lock?
  - if `eax == 0`
    - This means the `lock` was 0, and after running `xchg`, `lock` will be 1 (`eax` was 1)
    - We acquired the lock!!! (`lock` was 0 and now the `lock` is 1)
  - if `eax == 1`
    - This means the `lock` was 1, and after running `xchg`, `lock` will be 1
    - We did not acquire the lock (it was 1)
    - `lock == 1` means some other thread acquired this…

2nd Candidate: xchg_lock

- xchg_lock
  - Use atomic ‘xchg’ instruction to
  - Load and store values atomically
  - Set value to 1, and compare ret
    - If 0, then you can acquire the lock
    - If 1, lock as 1, you must wait
- xchg_unlock
  - Use atomic ‘xchg’
  - Set value to 0
    - Do not need to check
    - You are the only thread that runs in the
    - 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));
}

void
xchg_unlock(volatile uint32_t *lock) {
    xchg(lock, 0);
}
```

2nd Candidate: xchg_lock

- xchg_lock()/xchg_unlock()

```
>>> disass xchg_lock
Dump of assembler code for function xchg_lock:
   0x0000000000400b80 <+0>:     mov    $0x1,%edx
   0x0000000000400b85 <+5>:     nopl   (%rax)
   0x0000000000400b88 <+8>:     mov    %edx,%eax
   0x0000000000400b8a <+10>:    xchg   %eax,(%rdi)
   0x0000000000400b8c <+12>:    test   %eax,%eax
   0x0000000000400b8e <+14>:    jne    0x400b88 <xchg_lock+8>
   0x0000000000400b90 <+16>:    repz retq
```

1. Put 1 to edx

2. exchange

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

void
xchg_unlock(volatile uint32_t *lock) {
    xchg(lock, 0);
}
```

if eax == 1, lock is held by others, loop to +8

if eax == 0, lock acquired, return!

```
>>> disass xchg_unlock
Dump of assembler code for function xchg_unlock:
   0x0000000000400ba0 <+0>:     xor    %eax,%eax
   0x0000000000400ba2 <+2>:     xchg   %eax,(%rdi)
   0x0000000000400ba4 <+4>:     retq
```

2nd Candidate: xchg_lock Result

- Consistent!

```
os2 ~/cs444/s21/lock-example-master 158% ./lock xchg
Counting 10000 with 30 threads using XCHG_LOCK...
Count: 300000, elapsed Time:    906.339 ms
```

- (Run this code yourself!)

xchg Works well. Any Problem?

- Atomic xchg instruction load/store data at the same time
  - There is no aperture for race condition

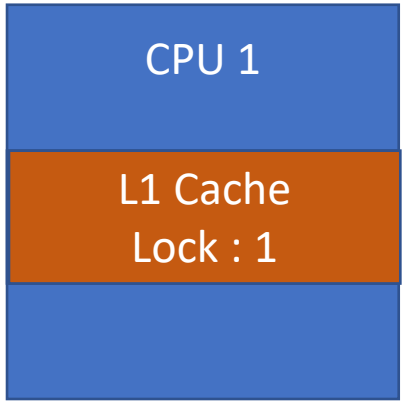
- But it could cause cache contention
  - Many threads updates the same ‘lock’ variable
  - CPUs cache data (thus cache ‘lock’), and we have multiple CPUs
  - Update invalidates cache…

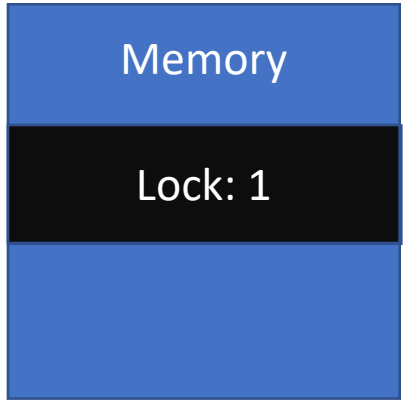
Cache Coherence

- xchg will always update the value
  - If lock == 0
    - lock = 1 **Swap with eax == 1, update lock to 1**
    - eax = 0
  - If lock == 1
    - lock = 1 **Swap with eax == 1, update lock to 1**
    - eax = 1
- We use while() to check the value in lock
  - Will be cached into L1 cache of the CPU
- After updating a value in cache
  - We need to invalidate the cache in other CPUs…

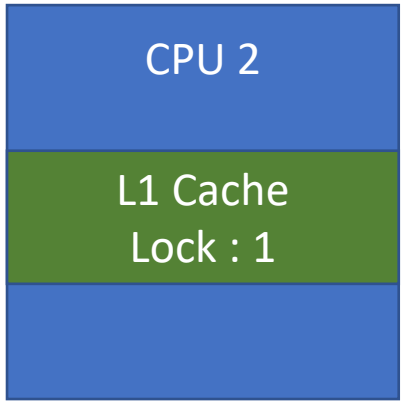
Cache Coherence and Write

Acquire Lock
xchg(lock, 1)



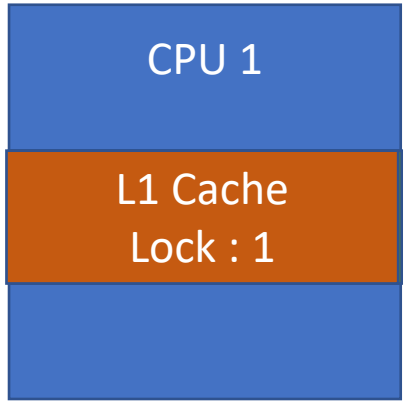


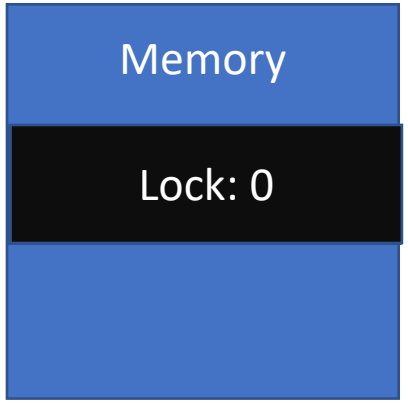
Spinlock
Waiting..
xchg(lock, 1)



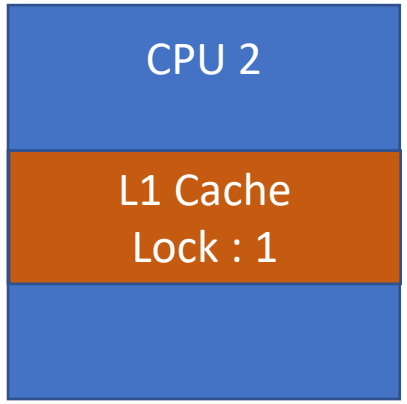
Cache Coherence and Write

Release Lock
xchg(lock, 0)





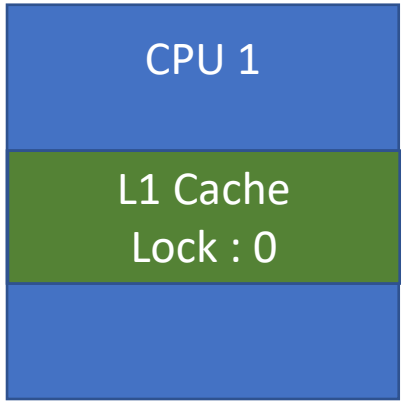
Spinlock
Waiting..
xchg(lock, 1)

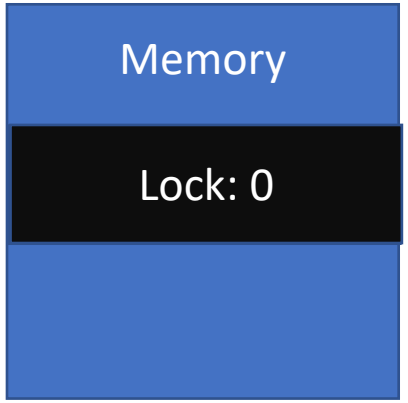


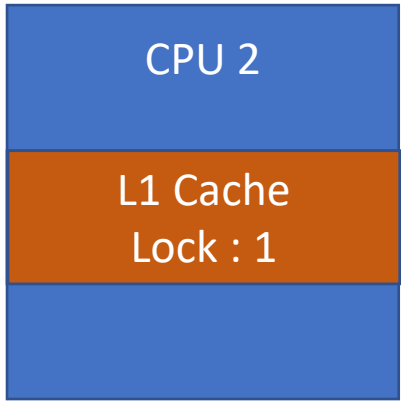
Invalid!

We need to flush the cache block if we update lock..

Cache Coherence and Write

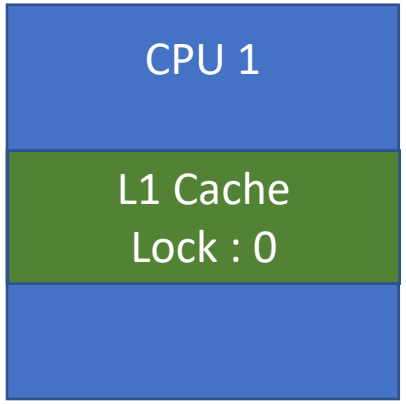


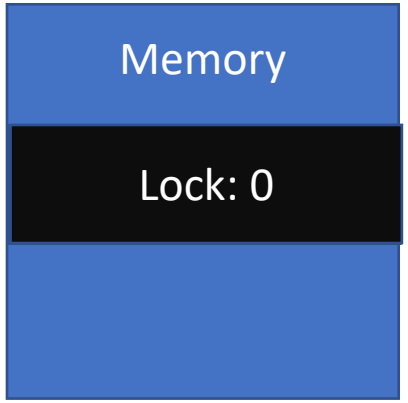




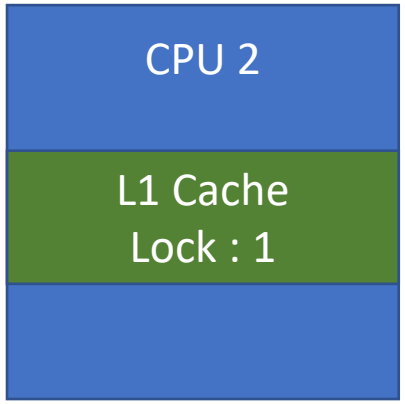
We need to flush the cache block if we update lock..

Cache Coherence and Write



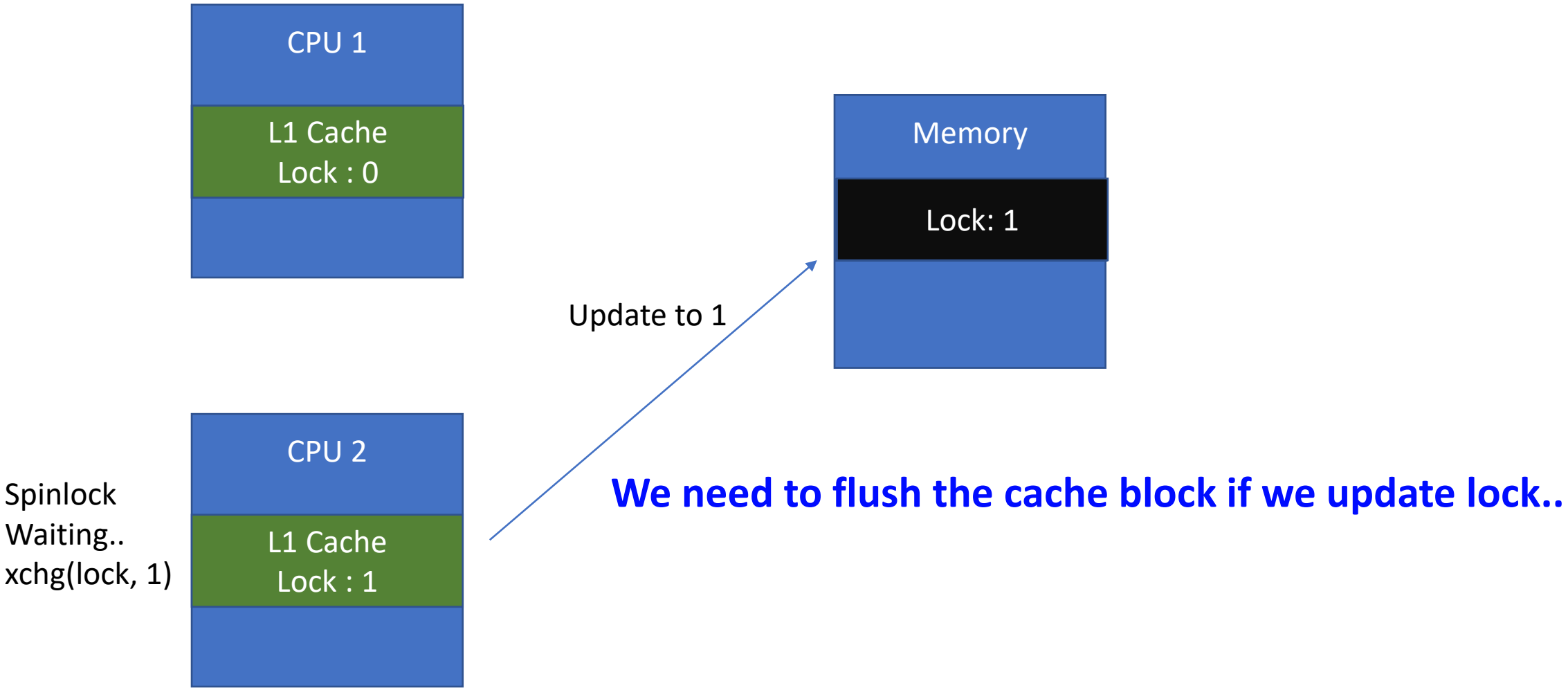


Spinlock
Waiting..
xchg(lock, 1)



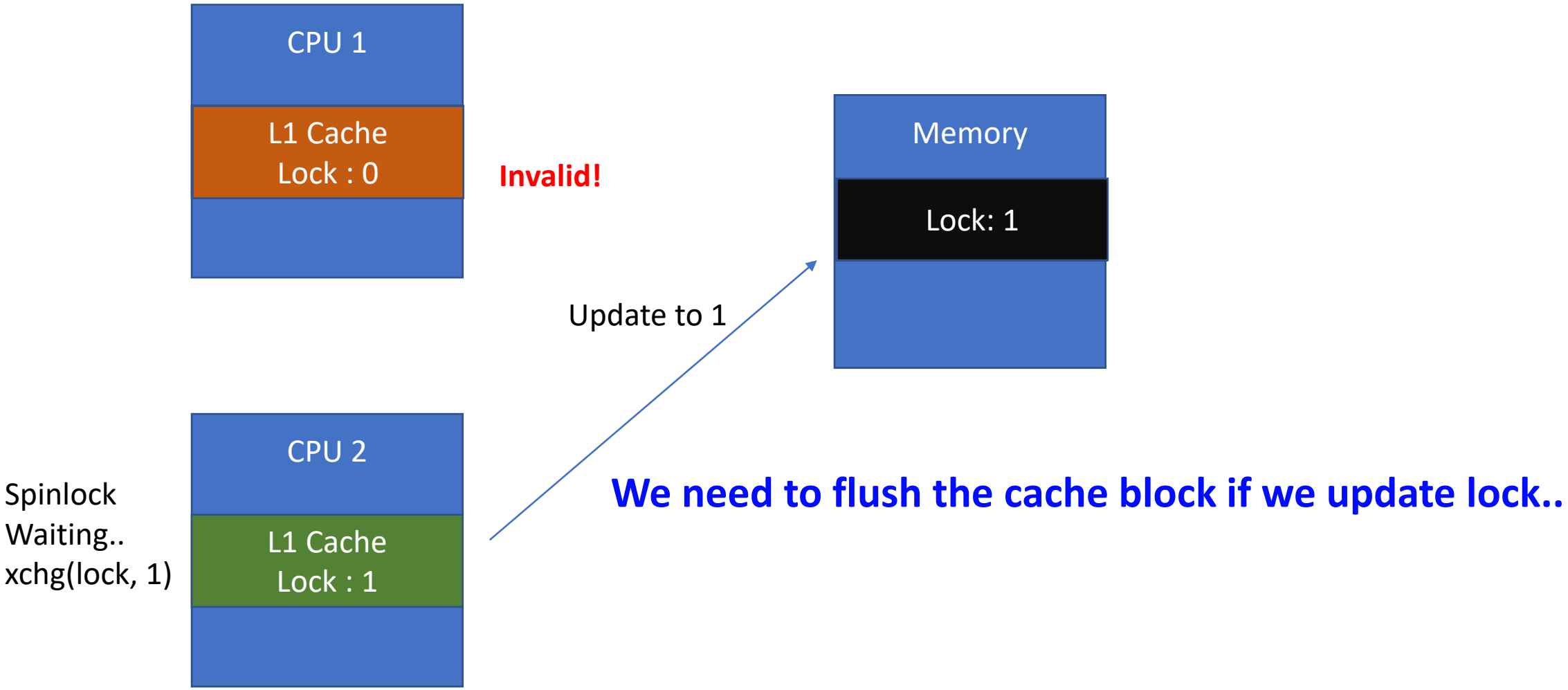
We need to flush the cache block if we update lock..

Cache Coherence and Write

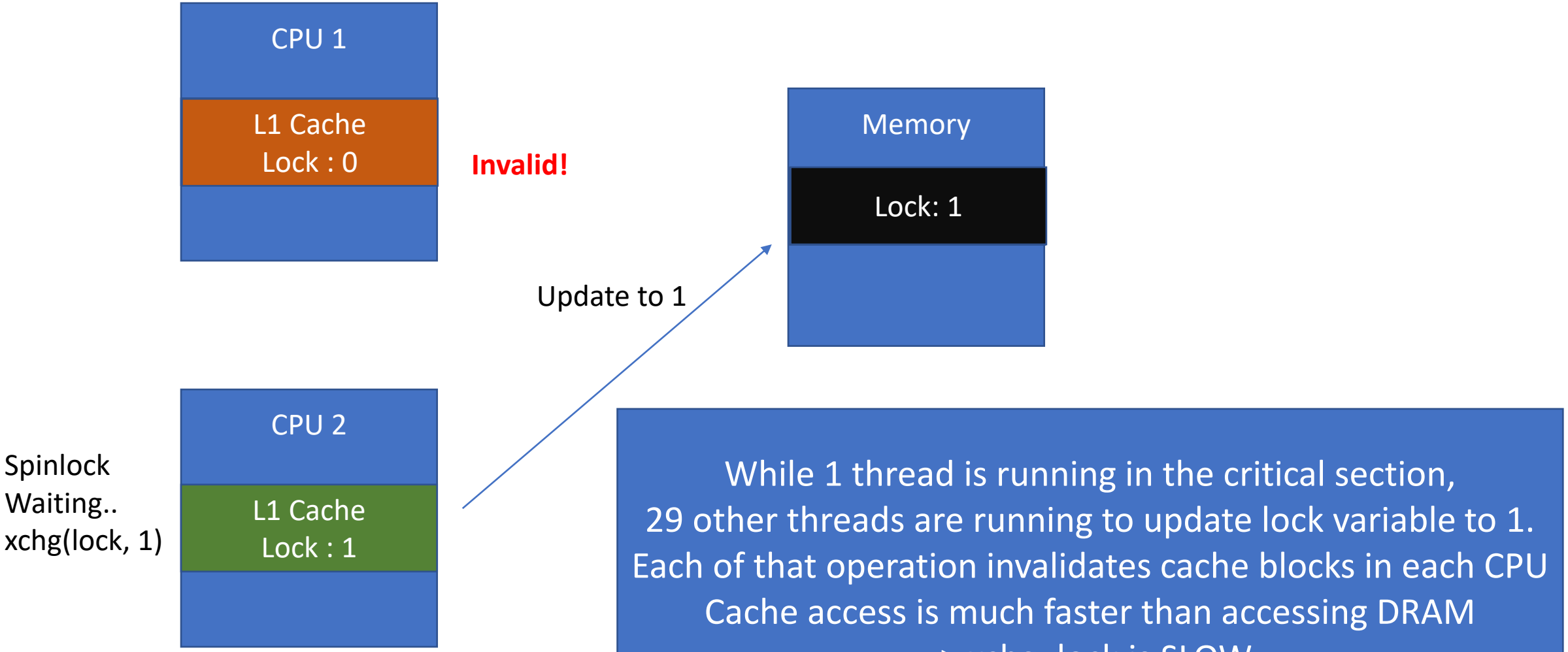


We need to flush the cache block if we update lock..

Cache Coherence and Write



Cache Coherence and Write



While 1 thread is running in the critical section,
29 other threads are running to update lock variable to 1.
Each of that operation invalidates cache blocks in each CPU
Cache access is much faster than accessing DRAM
-> xchg_lock is SLOW

Use `perf` to measure # of L1 Cache Miss

4 / 30 cores ≈ 130ms

930ms / 130ms ≈ 7

- `./perf-lock.sh xchg`

```
os2 ~/cs444/s21/lock-example-master 161% taskset -c 1 ./perf-lock.sh xchg
Counting 10000 with 30 threads using XCHG_LOCK...
Count: 300000, elapsed Time:   3928.363 ms

 Performance counter stats for './lock xchg':

          120,481      L1-dcache-load-misses:u

       3.930350597 seconds time elapsed

       3.896006000 seconds user
       0.023944000 seconds sys
```

≈ 45 with 1 core

Running on Single CPU, so no cache coherence invalidate
120,484 L1 cache miss

~150x more cache misses!

Running on 30 CPUs, MANY cache coherence invalidate
16,512,510 L1 cache miss

```
os2 ~/cs444/s21/lock-example-master 159% ./perf-lock.sh xchg
Counting 10000 with 30 threads using XCHG_LOCK...
Count: 300000, elapsed Time:    930.011 ms

 Performance counter stats for './lock xchg':

       16,512,510      L1-dcache-load-misses:u

      0.934165522 seconds time elapsed

     22.914399000 seconds user
      0.098768000 seconds sys
```

Test-and-Set (`xchg`)

- Pros
  - Synchronizes threads well!
- Cons
  - SLOW
  - Lots of cache miss

```
os2 ~/cs444/s21/lock-example-master 159% ./perf-lock.sh xchg
Counting 10000 with 30 threads using XCHG_LOCK...
Count: 300000, elapsed Time:    930.011 ms

 Performance counter stats for './lock xchg':

       16,512,510      L1-dcache-load-misses:u

      0.934165522 seconds time elapsed

     22.914399000 seconds user
      0.098768000 seconds sys
```

Updating Lock if Lock == 1 is **Not Required**

- Updating the same value causes unnecessary cache invalidation
- Avoid this, but how?

- New method: `Test and test-and-set`
  - Check the value first (if lock == 0) ← `TEST`
  - If it is,
    - Do `test-and-set`
  - Otherwise (if lock == 1),
    - Do nothing
    - DO NOT UPDATE lock if lock == 1 (**No cache invalidate**)

Test and Test-and-set in x86: `lock cmpxchg`

- `cmpxchg [update-value], [memory]`
  - Compare the value in `[memory]` with `%eax` — **Test**
  - If matched, exchange value in `[memory]` with `[update-value]` — **Test-and-set**
  - Otherwise, do not perform exchange

- xchg(lock, 1)
  - Lock = 1
  - Returns old value of the lock
- cmpxchg(lock, 0, 1)
  - Arguments: Lock, test value, update value
  - Returns old value of lock

CAVEAT

- `xchg` is an atomic operation in x86

- `cmpxchg` is not an atomic operation in x86
  - Must be used with lock prefix to guarantee atomicity

  - `lock cmpxchg`

3rd Candidate: cmpxchg_lock

- Cmpxchg_lock
  - Use cmpxchg to set lock = 1
    - Do not update if lock == 1
    - Only write 1 to lock if lock == 0

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

Critical Section

- Xchg_unlock
  - Use xchg_unlock to set lock = 0
  - Because we have 1 writer and
  - This always succeeds…

```
void
cmpxchg_lock(volatile uint32_t *lock) {
    while(cmpxchg(lock, 0, 1));
}

void
xchg_unlock(volatile uint32_t *lock) {
    xchg(lock, 0);
}
```

3rd Candidate: cmpxchg_lock Result

- Consistent!

```
os2 ~/cs444/s21/lock-example-master 165% ./perf-lock.sh cmpxchg
Counting 10000 with 30 threads using CMPXCHG_LOCK...
Count: 300000, elapsed Time:   1024.987 ms

 Performance counter stats for './lock cmpxchg':

       18,153,123      L1-dcache-load-misses:u

      1.028728892 seconds time elapsed

     26.794265000 seconds user
      0.080822000 seconds sys
```

But showing much more cache misses than xchg.. Why????

it does not update if lock ==1…

Intel CPU is TOO COMPLEX

This *[cmpxchg]* instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processors bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

Cmpxchg designed to be Test and Test & Set instruction
However, Intel CPU gets too complex, so they decided to always update the value regardless the result of comparison

LAME! Let's Implement Software Test and Test & Set

4th Candidate: Test and Test & Set

- tts_xchg_lock
- Algorithm
  - Wait until lock becomes 0
  - After lock == 0
    - xchg (lock, 1)
    - This only updates lock = 1 if lock was 0

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

Critical Section

- Why xchg, why not *lock = 1 directly
  - while and xchg are not atomic
  - Load/Store must happen at
    - The same time!

```
void
tts_xchg_lock(volatile uint32_t *lock) {
    while (1) {
        while(*lock == 1);
        if (xchg(lock, 1) == 0) {
            break;
        }
    }
}
```

4th Candidate TTS Result

- Consistent!

```
os2 ~/cs444/s21/lock-example-master 166% ./perf-lock.sh tts
Counting 10000 with 30 threads using TTS_LOCK...
Count: 300000, elapsed Time:    473.709 ms

 Performance counter stats for './lock tts':

       13,661,665      L1-dcache-load-misses:u

      0.477827903 seconds time elapsed

     13.089950000 seconds user
      0.106730000 seconds sys
```

- A little less cache misses but
- Faster (~500ms vs. 900 ~ 1200 ms)

Still Slow and Many Cache Misses..

- Can we do better? Why we still have too many misses?
  - A thread acquires the lock (update 0 -> 1)
    - Invalidate caches in 29 other cores
  - A thread releases the lock (update 1 -> 0)
    - Invalidate caches in 29 other cores
  - 29 other cores are all reading the variable lock
    - Immediately after invalidate, it loads data to cache
    - Then invalidated again by either lock/release…
    - This happens in every 3~4 cycles…

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

```
void
tts_xchg_lock(volatile uint32_t *lock) {
    while (1) {
        while(*lock == 1);
        if (xchg(lock, 1) == 0) {
            break;
        }
    }
}
```

5th Candidate: Backoff Lock

- Too many contention on reading lock while only 1 can run critical sec.
  - All other 29 cores running while `(*lock == 1);`
  - This is the slow down factor

- Idea: can we slow down that check?
  - Let's set a wait time if CPU checked the lock value as 1
- Something like, exponential backoff
  - After checking lock == 1,
    - Wait 1 cycle
  - After checking lock == 1 again,
    - Wait 2 cycles
    - Wait 4 cycles
    - Wait 8 cycles
    - …

```
void
tts_xchg_lock(volatile uint32_t *lock) {
    while (1) {
        while(*lock == 1);
        if (xchg(lock, 1) == 0) {
            break;
        }
    }
}
```

5th Candidate: Backoff Lock

- backoff_cmpxchg_lock(lock)
- Try cmpxchg
  - If succeeded, acquire the lock.
  - If failed
    - Wait 1 cycle (pause) for 1st trial
    - Wait 2 cycles for 2nd trial
    - Wait 4 cycles for 3rd trial
    - …
    - Wait 65536 cycles for 17th trial..
    - Wait 65536 cycles for 18th trial..

```
void
backoff_cmpxchg_lock(volatile uint32_t *lock) {
    uint32_t backoff = 1;
    while(cmpxchg(lock, 0, 1)) {
        for (int i=0; i<backoff; ++i) {
            __asm volatile("pause");
        }
        if (backoff < 0x10000) {
            backoff <<= 1;
        }
    }
}
```

- https://en.wikipedia.org/wiki/Exponential_backoff

5th Candidate: Backoff Result

- **Consistent!**

```
os2 ~/cs444/s21/lock-example-master 168% ./perf-lock.sh backoff
Counting 10000 with 30 threads using BACKOFF_LOCK...
Count: 300000, elapsed Time:    210.387 ms

 Performance counter stats for './lock backoff':

          196,227      L1-dcache-load-misses:u

       0.214007977 seconds time elapsed

       4.405105000 seconds user
       0.112746000 seconds sys
```

- **Much lower cache miss**
- **Faster! (~200ms!)**

Even Faster Than pthread_mutex

```
os2 ~/cs444/s21/lock-example-master 168% ./perf-lock.sh backoff
Counting 10000 with 30 threads using BACKOFF_LOCK...
Count: 300000, elapsed Time:    210.387 ms

 Performance counter stats for './lock backoff':

          196,227      L1-dcache-load-misses:u

       0.214007977 seconds time elapsed

       4.405105000 seconds user
       0.112746000 seconds sys
```

```
os2 ~/cs444/s21/lock-example-master 170% ./perf-lock.sh mutex
Counting 10000 with 30 threads using MUTEX_LOCK...
Count: 300000, elapsed Time:    473.064 ms

 Performance counter stats for './lock mutex':

        1,656,537      L1-dcache-load-misses:u

       0.477209142 seconds time elapsed

       0.519430000 seconds user
      12.487676000 seconds sys
```

Summary

- Mutex is implemented with Spinlock
  - Waits until lock == 0 with a while loop (that's why it's called spin)
- Naïve code implementation never works
  - Load/Store must be atomic
- xchg is a "test and set" atomic instruction
  - Consistent, however, many cache misses, slow! (950ms)
- Lock cmpxchg is a "test and test&set" atomic instruction
  - But Intel implemented this as xchg… slow! (1150ms)
- We can implement test-and-test-and-set (tts) with while + xchg
  - Faster! (500ms)
- We can also implement exponential backoff to reduce contention
  - Much faster! (200ms)

```
os2 ~/cs444/s21/lock-example-master 172% ./lock
Counting 10000 with 30 threads using NO_LOCK...
Count: 37484, elapsed Time:     37.261 ms
Counting 10000 with 30 threads using BAD_LOCK...
Count: 45567, elapsed Time:     43.420 ms
Counting 10000 with 30 threads using XCHG_LOCK...
Count: 300000, elapsed Time:    908.793 ms
Counting 10000 with 30 threads using CMPXCHG_LOCK...
Count: 300000, elapsed Time:    956.066 ms
Counting 10000 with 30 threads using TTS_LOCK...
Count: 300000, elapsed Time:    465.198 ms
Counting 10000 with 30 threads using BACKOFF_LOCK...
Count: 300000, elapsed Time:    142.791 ms
Counting 10000 with 30 threads using MUTEX_LOCK...
Count: 300000, elapsed Time:    428.405 ms
```