Synchronization

ECE 469, Mar 27

Aravind Machiry

Threads

- Separate the concepts of a “thread of control” (PC, SP, registers) from the rest of the process (address space, resources, accounting, etc.)

- Modern OSes support two entities:
  - the *task* (process), which defines an address space, a resource container, accounting info
  - the *thread* (lightweight process), which defines a single sequential execution stream within a task (process)

Programming with Threads

- Flexible, but error-prone, since there no protection between threads
  - In C/C++,
    - automatic variables are private to each thread
    - global variables and dynamically allocated memory (malloc) are shared

- Need synchronization!

The need for synchronization!

- Cooperating processes may share data via
  - shared address space (code, data, heap) by using threads
  - Files
  - (Sending messages)
- What can happen if processes try to access shared data (address) concurrently?
  - Sharing bank account with sibling:

    At 3pm: If (balance > $10) withdraw $10
- How hard is the solution?

"Too much milk" Problem

Person A

1. Look in fridge: out of milk
2. Leave for Walmart
5. Arrive at Walmart
6. Buy milk
7. Arrive home

Person B

3. Look in fridge: out of milk
4. Leave for Walmart
8. Arrive at Walmart
9. Buy milk
10. Arrive home

- How to put in a locking mechanism?

Possible Solution 1

Person A

```
if ( noMilk ) {
  if (noNote) {
     leave note;
     buy milk;
     remove note;
  }
}
```

Person B

```
if ( noMilk ) {
  if (noNote) {
     leave note;
     buy milk;
     remove note;
  }
}
```

Will this work?

Person A

```
1.if ( noMilk ) {
  2.if (noNote) {
    5.leave note;
    buy milk;
    remove note;
  }
}
```

Person B

```
3.if ( noMilk ) {
  4.if (noNote) {
    6.leave note;
    buy milk;
    remove note;
  }
}
```

- Process can get context switched after checking milk and note, but before leaving note

Possible Solution 2

Person A

```
leave noteA
if (no noteB) {
  if (noMilk) {
    buy milk
  }
}
remove noteA
```

Person B

```
leave noteB
if (no noteA) {
  if (noMilk) {
    buy milk
  }
}
remove noteB
```

Will this work?

Person A

```
leave noteA
if (no noteB) {
   if (noMilk) {
      buy milk
   }
}
remove noteA
```

Person B

```
leave noteB
if (no noteA) {
   if (noMilk) {
      buy milk
   }
}
remove noteB
```

- We may not have Milk: Both process can leave note and skip buying milk

Possible Solution 3

Process A

```
leave noteA
while (noteB)
  do nothing;
if (noMilk)
  buy milk;
remove noteA
```

Process B

```
leave noteB
if (noNoteA) {
  if (noMilk) {
    buy milk
  }
}
remove noteB
```

Works, but complicated!

Process A

```
leave noteA
while (noteB)
  do nothing;
if (noMilk)
  buy milk;
remove noteA
```

Process B

```
leave noteB
if (noNoteA) {
  if (noMilk) {
    buy milk
  }
}
remove noteB
```

- A's code is different from B's
- busy waiting is a waste

How can we solve this?

- Root cause: Data Race

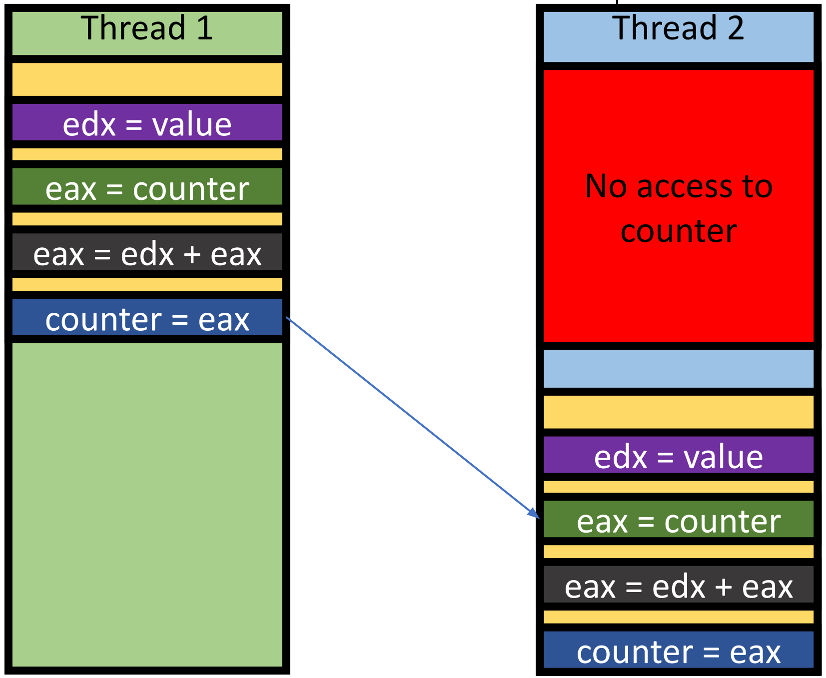
- A thread's execution result could be inconsistent if other threads intervene its execution…

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

```
mov     0x20087b(%rip),%edx          # 0x201010 <value>
mov     0x20087d(%rip),%eax          # 0x201018 <counter>
add     %edx,%eax
mov     %eax,0x200875(%rip)          # 0x201018 <counter>
```

How can we prevent data races?

- What we need?
  - Exclusive access to counter (shared variable)



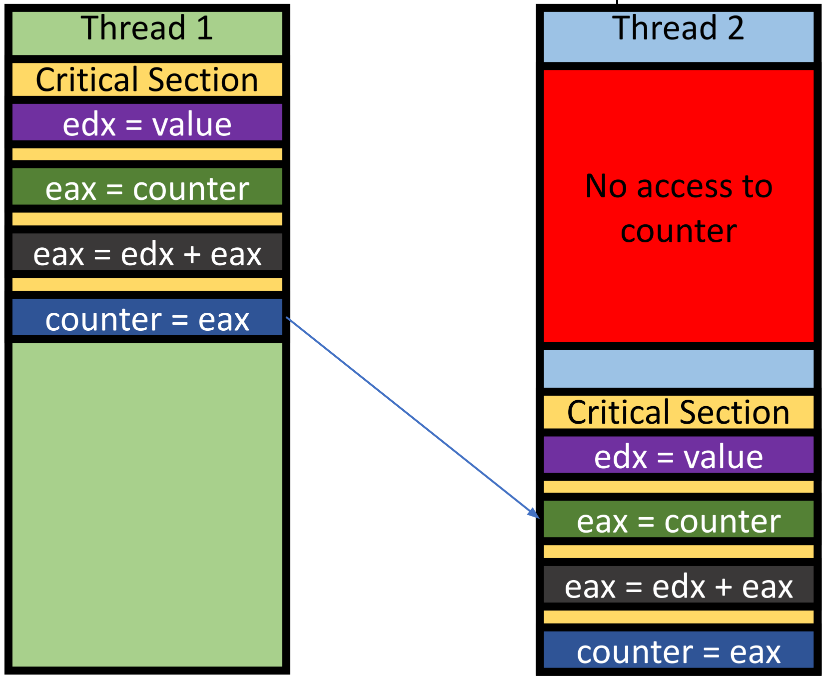
How can we prevent data races?

- *Critical section* – a section of code, or collection of operations, in which only one process shall be executing at a given time

- *Mutual exclusion (Mutex)* - mechanisms that ensure that only one person or process is doing certain things at one time (others are excluded)

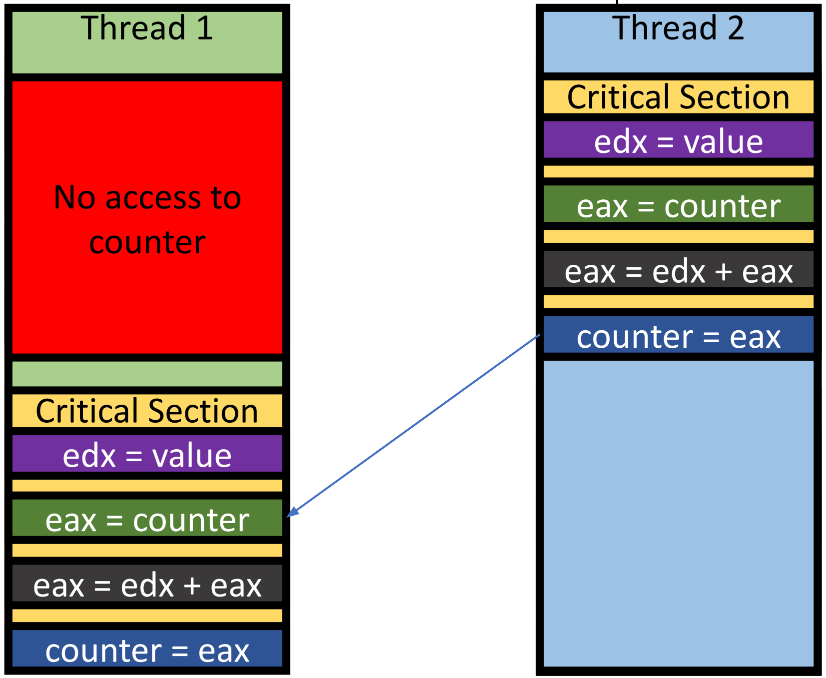
How can we prevent data races?

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



How can we prevent data races?

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



Mutex Considerations

- Mutex can synchronize multiple threads and yield consistent result
  - No read before previous thread store 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!

Implementing Mutual exclusion

- Data races occur because of scheduler interleaving executing of different threads

- How to avoid this? Prevent interleaving

Preventing Interleaving

- `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?**

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

Preventing Interleaving

- `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`**

Mutual Exclusion through locks

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

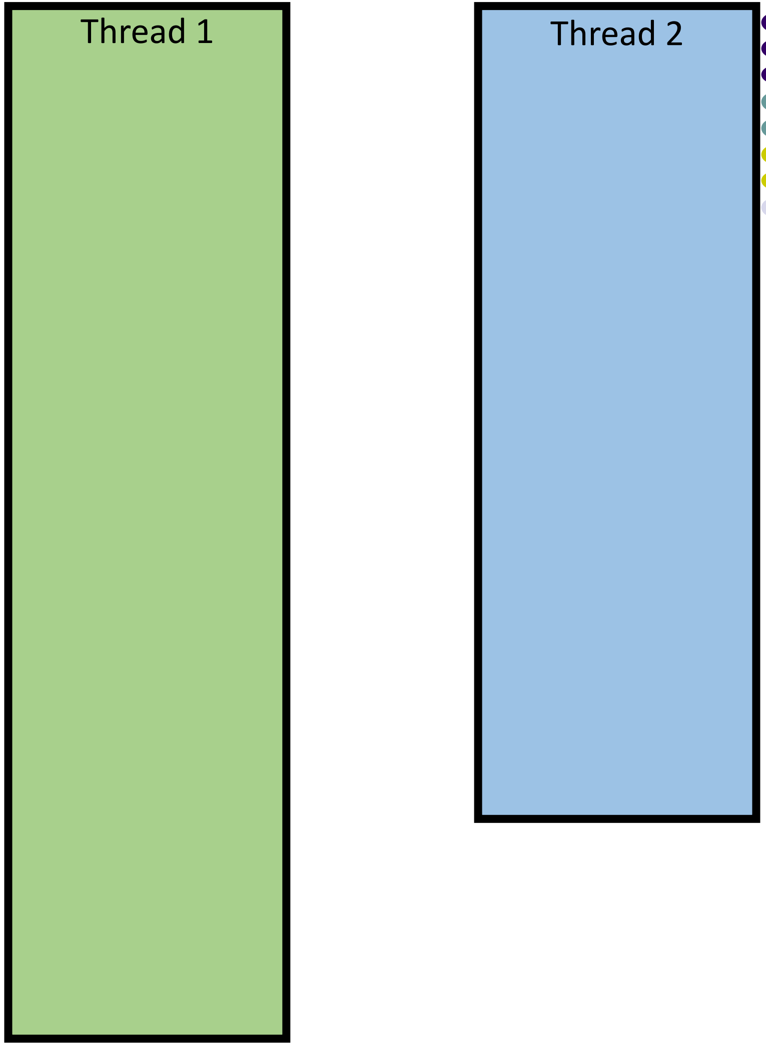
- counter += value
  - **lock()**
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`
  - **unlock()**

Mutual Exclusion through locks

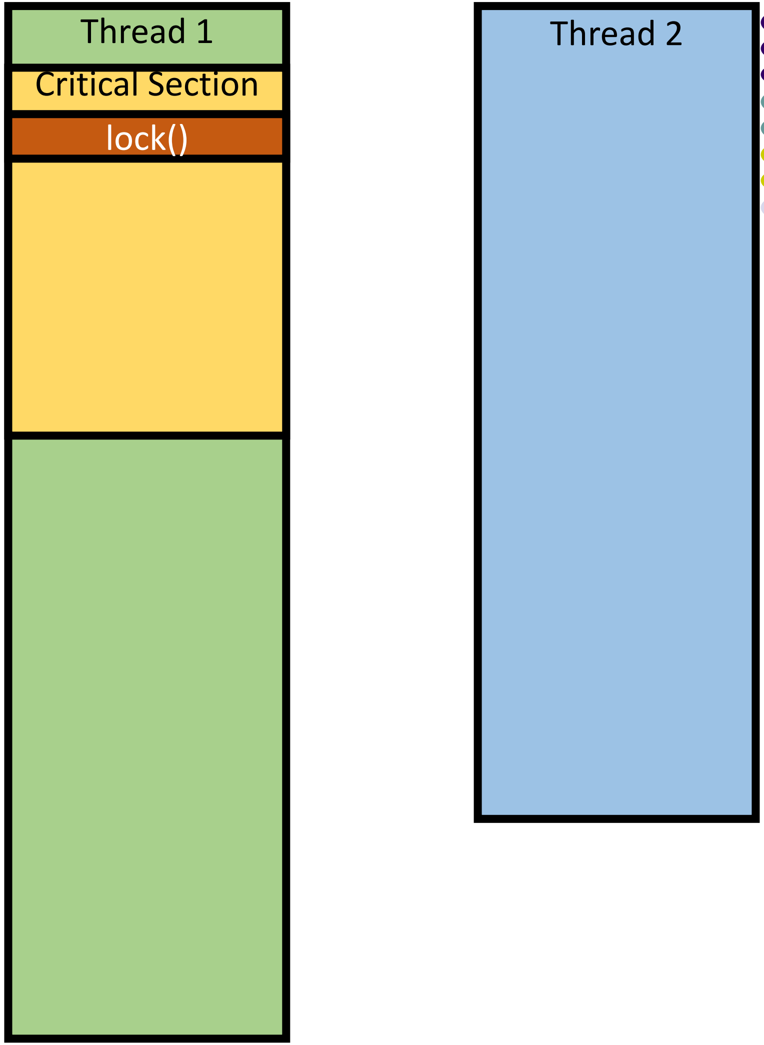
- Lock
  - Prevent others enter the critical section
- How?
  - Check if any other execution in the critical section
    - If it is, wait; busy-waiting with while()
  - If not, acquire the lock!
    - Others cannot get into the critical section
  - Run critical section
  - Unlock, let other execution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

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

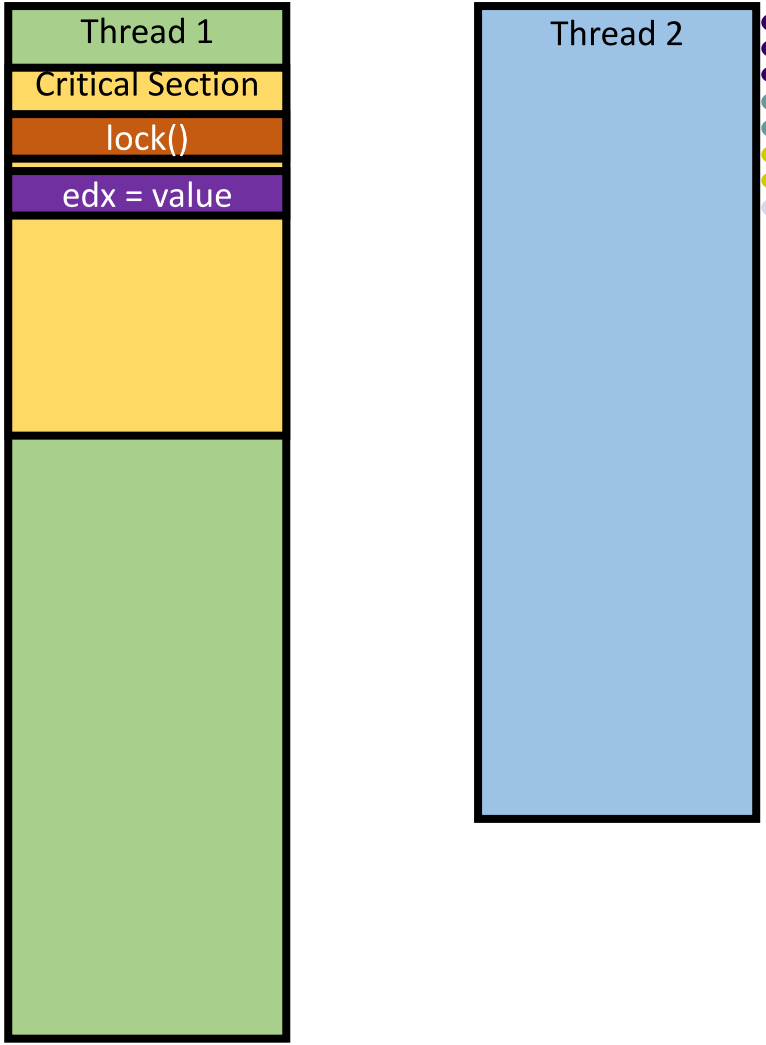
Mutex Example



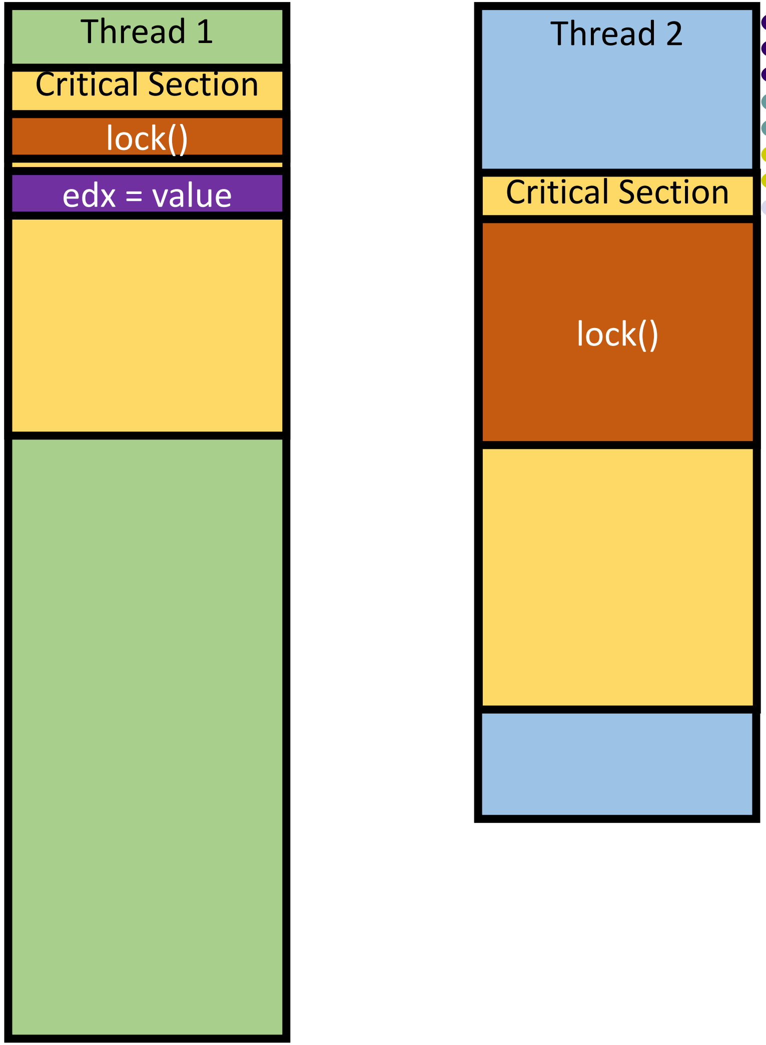
Mutex Example



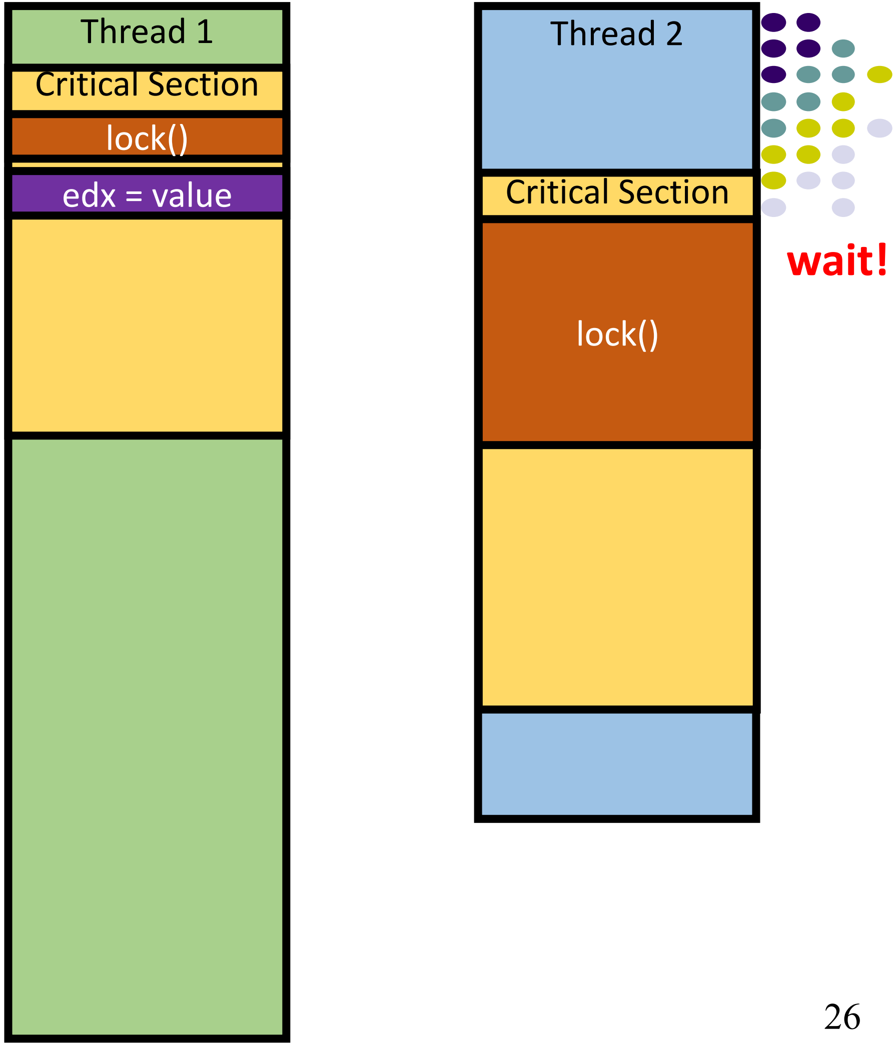
Mutex Example



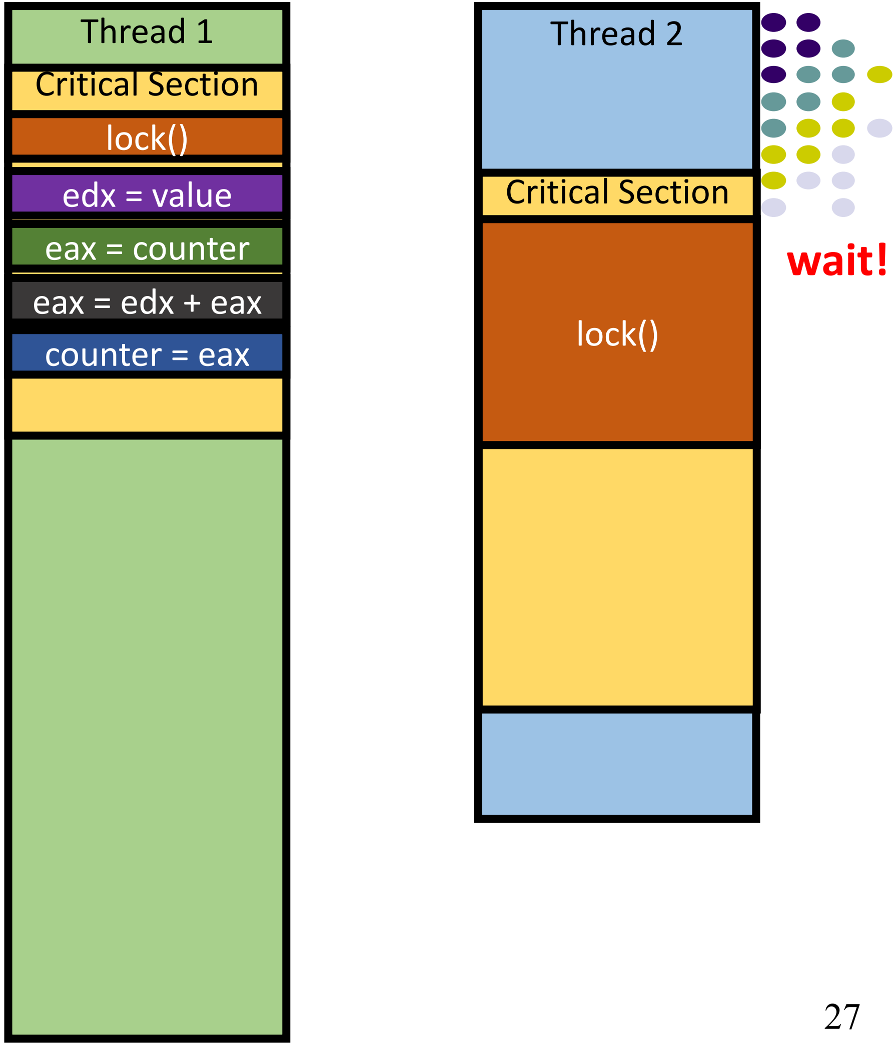
Mutex Example



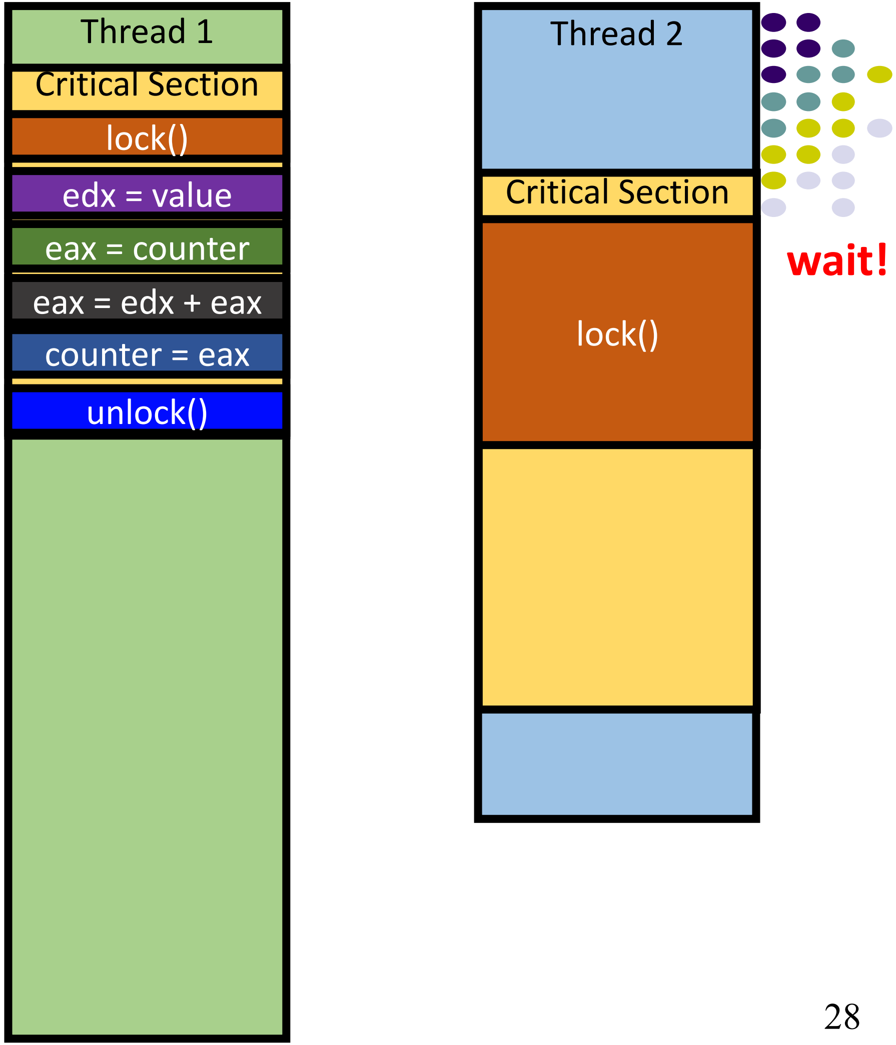
Mutex Example



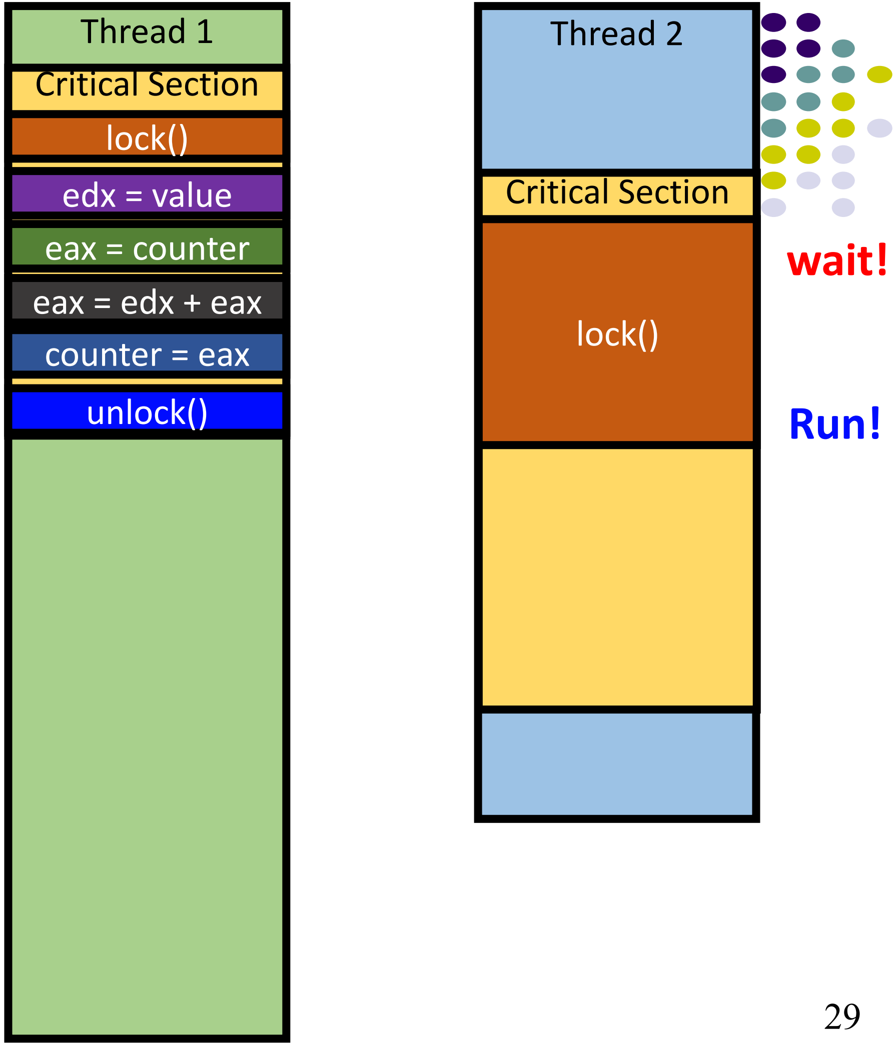
Mutex Example



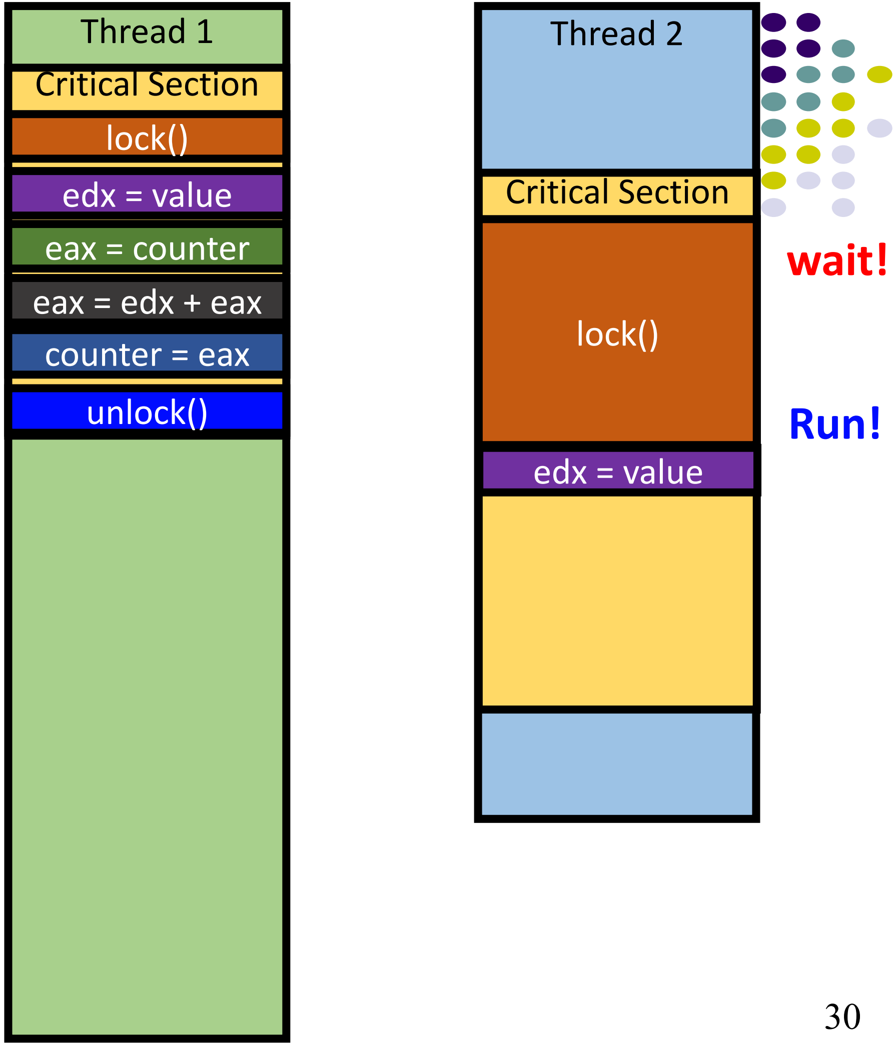
Mutex Example



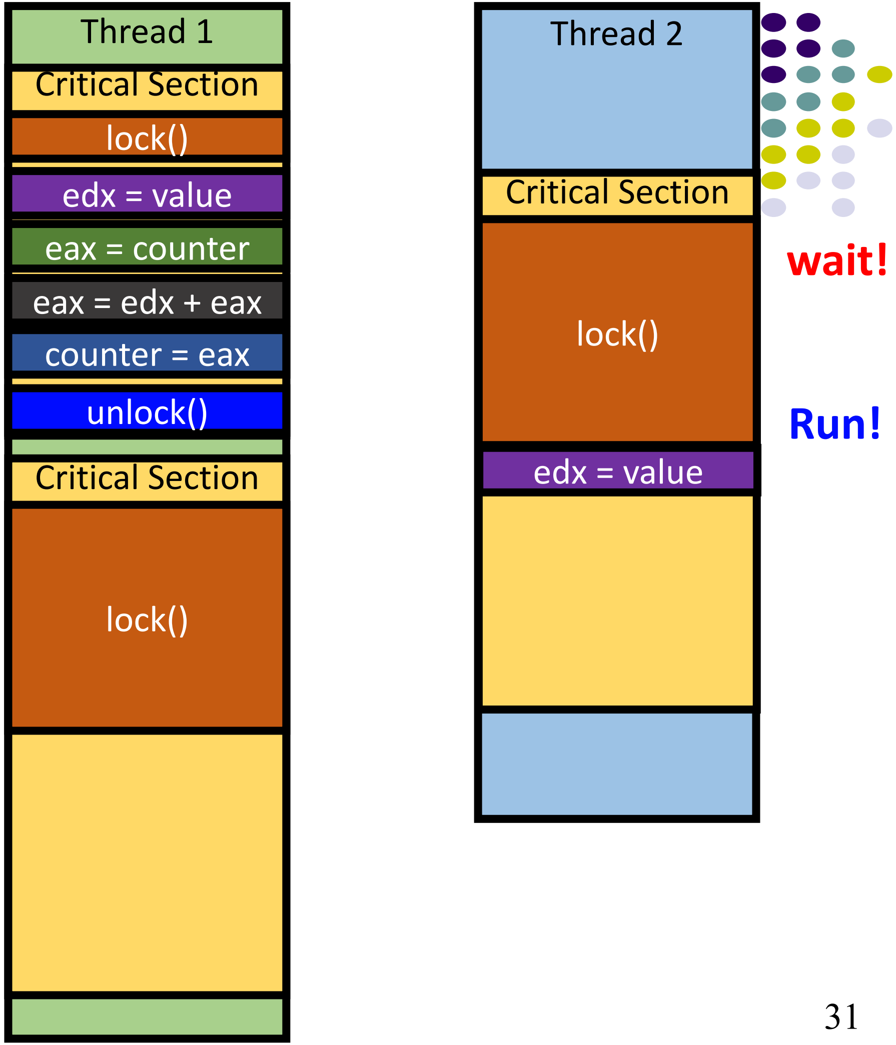
Mutex Example



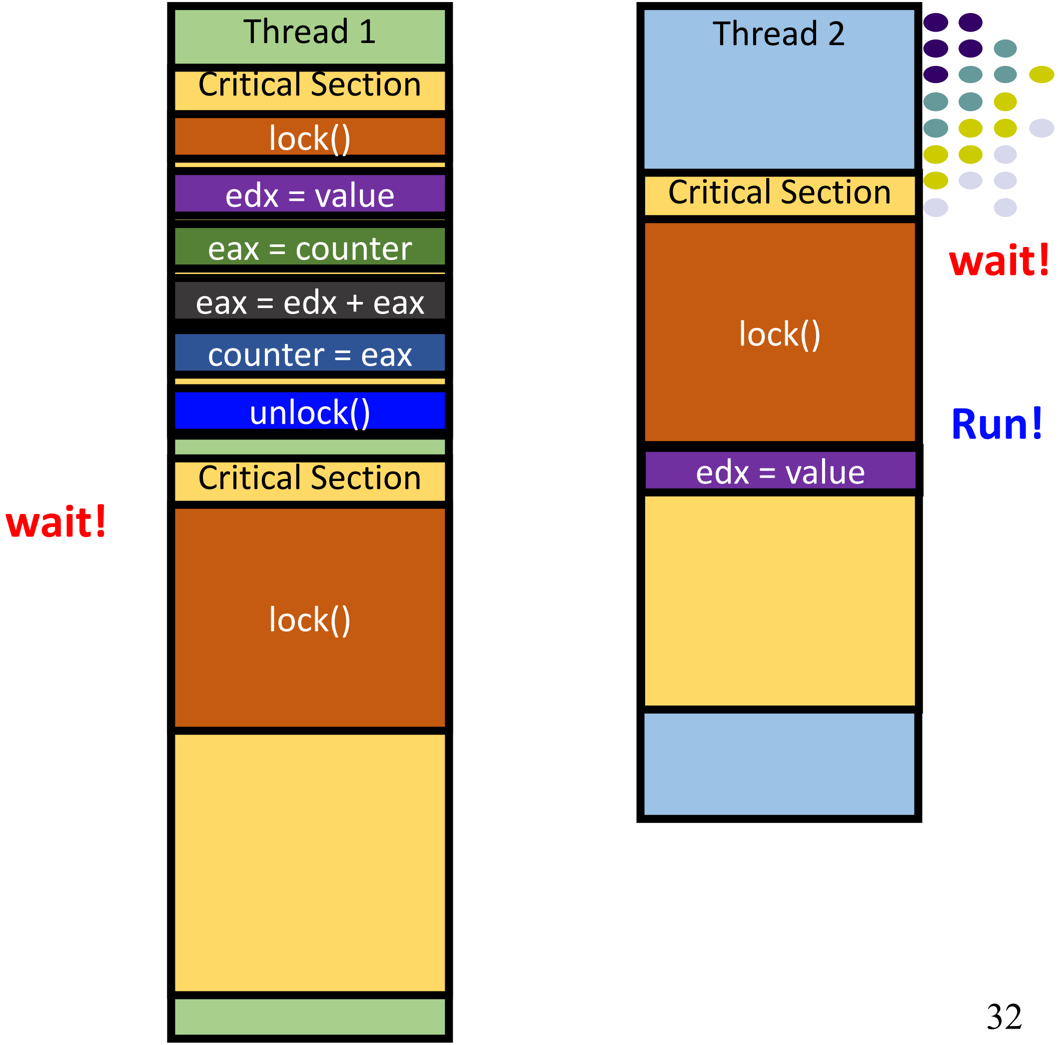
Mutex Example



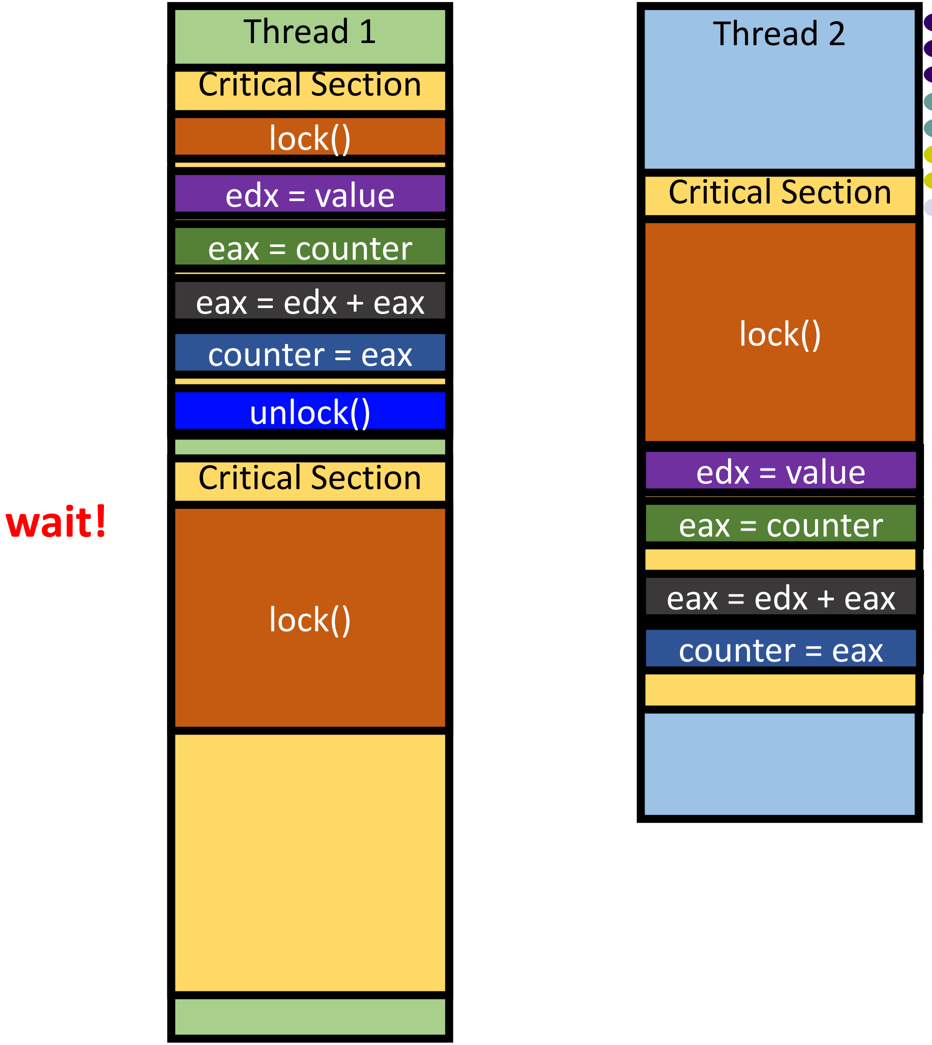
Mutex Example



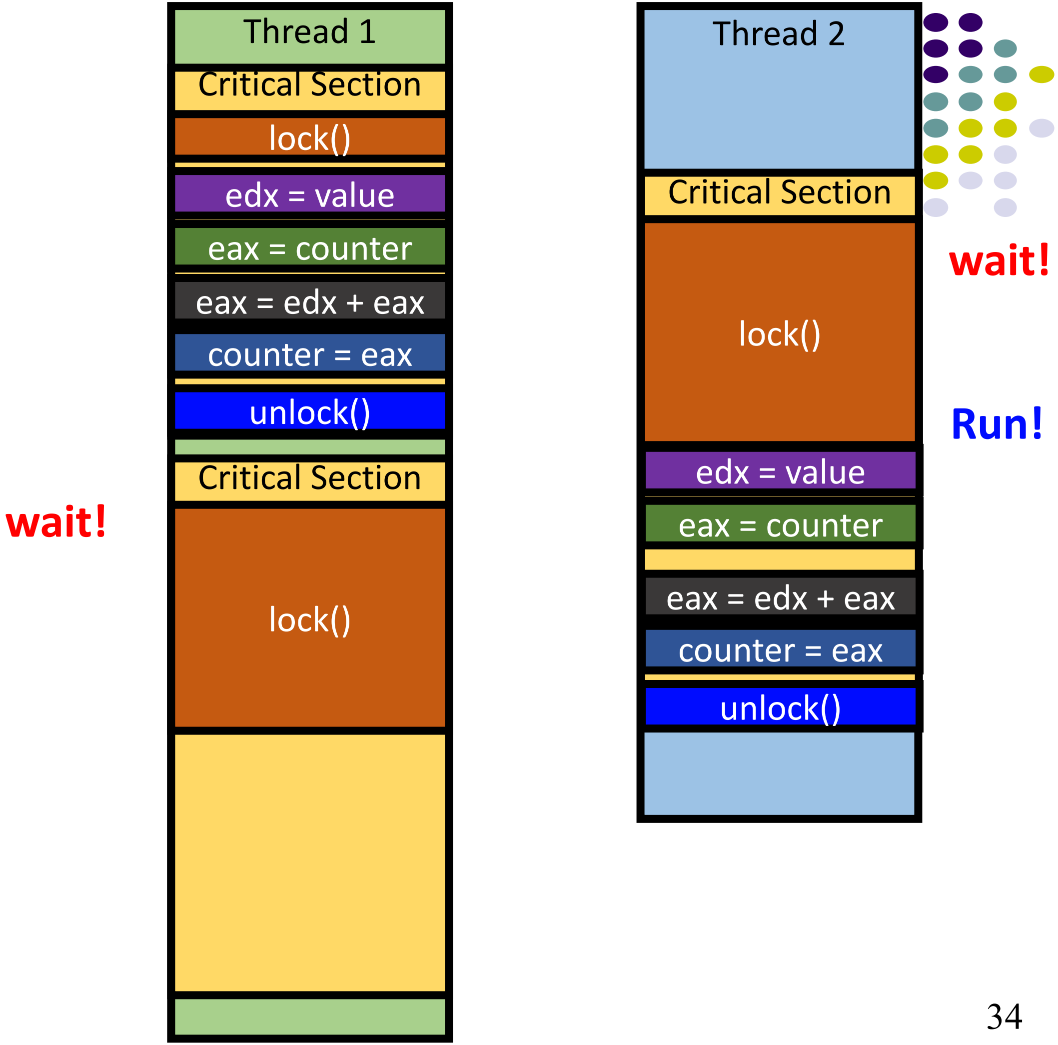
Mutex Example



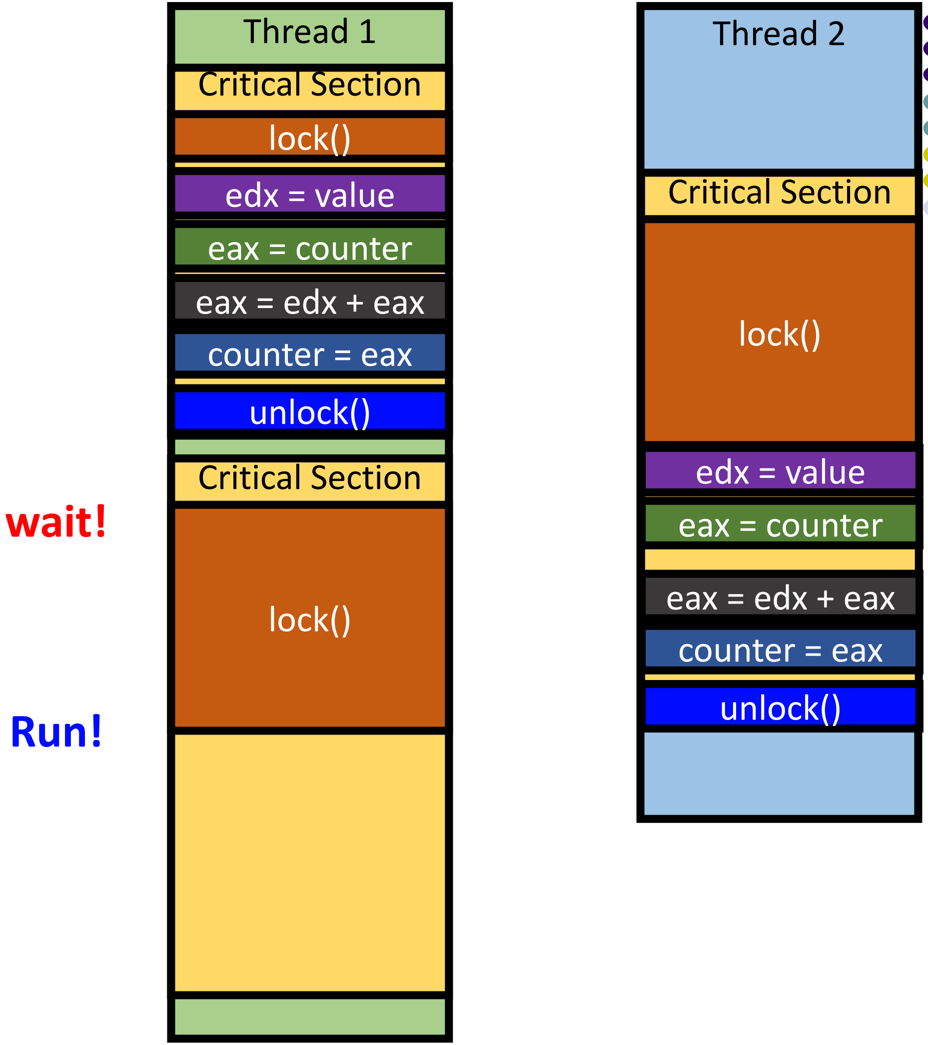
Mutex Example



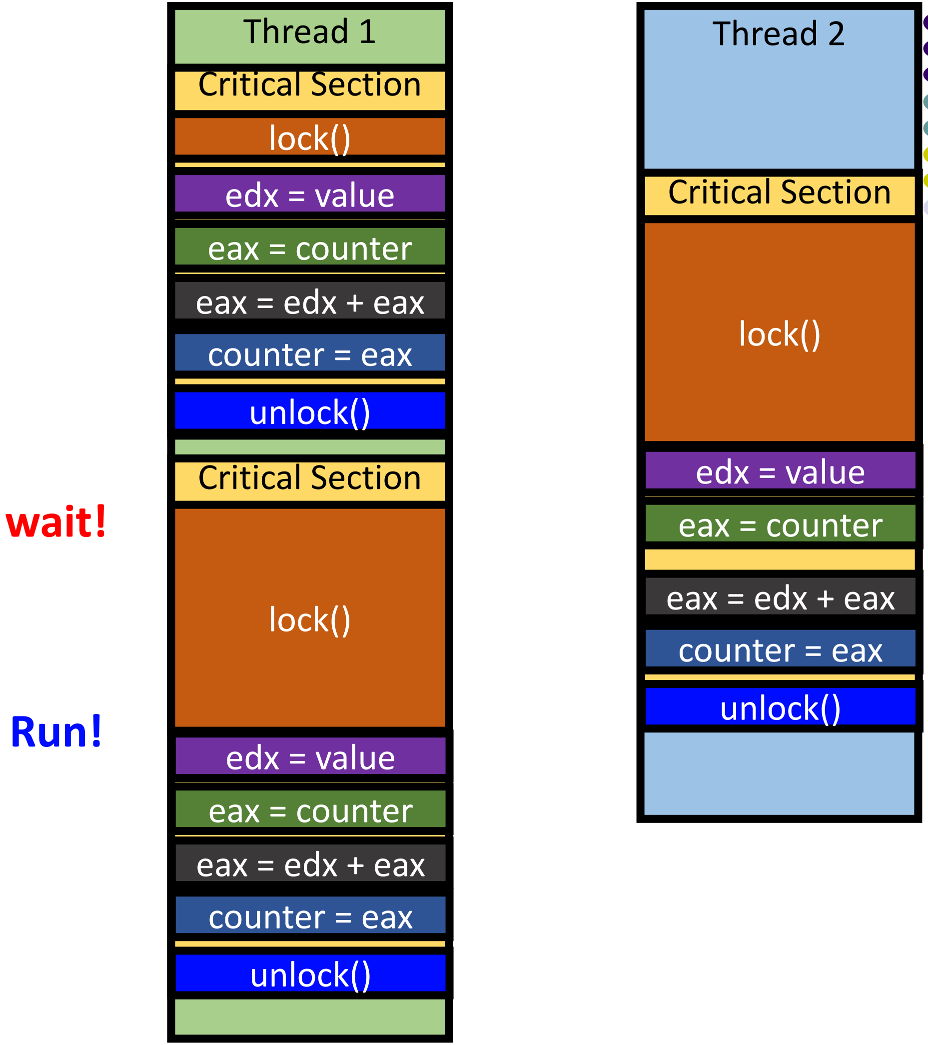
Mutex Example



Mutex Example

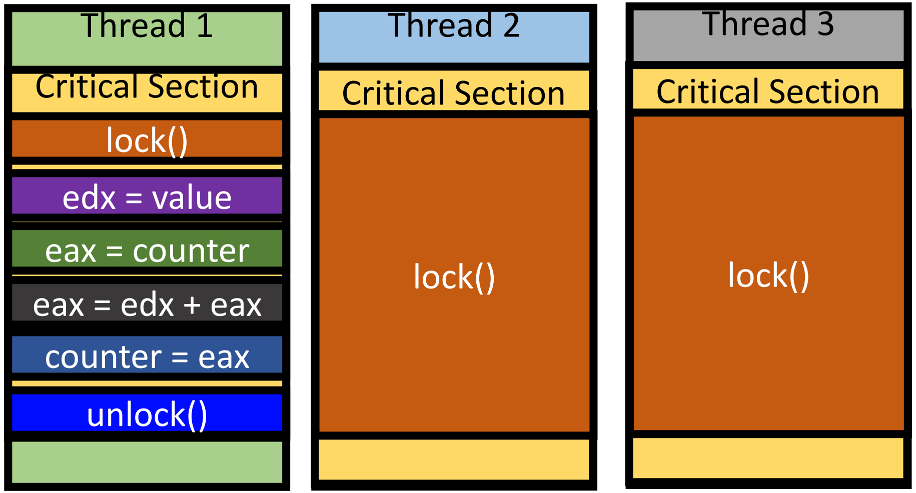


Mutex Example



Implementing lock

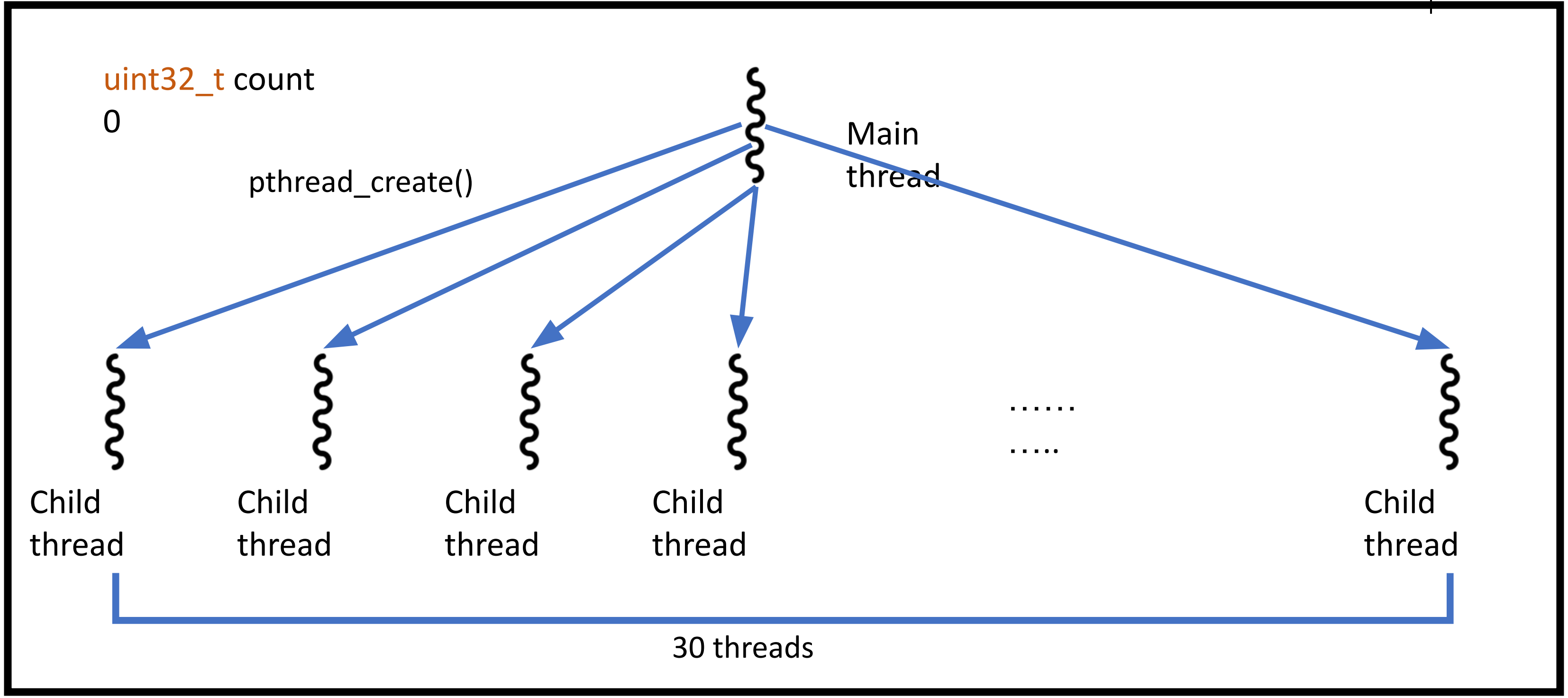
- Only one can run in critical section
- Others must wait!
  - Until nobody runs in critical section
- How can we create such
  - Lock() / Unlock() ?



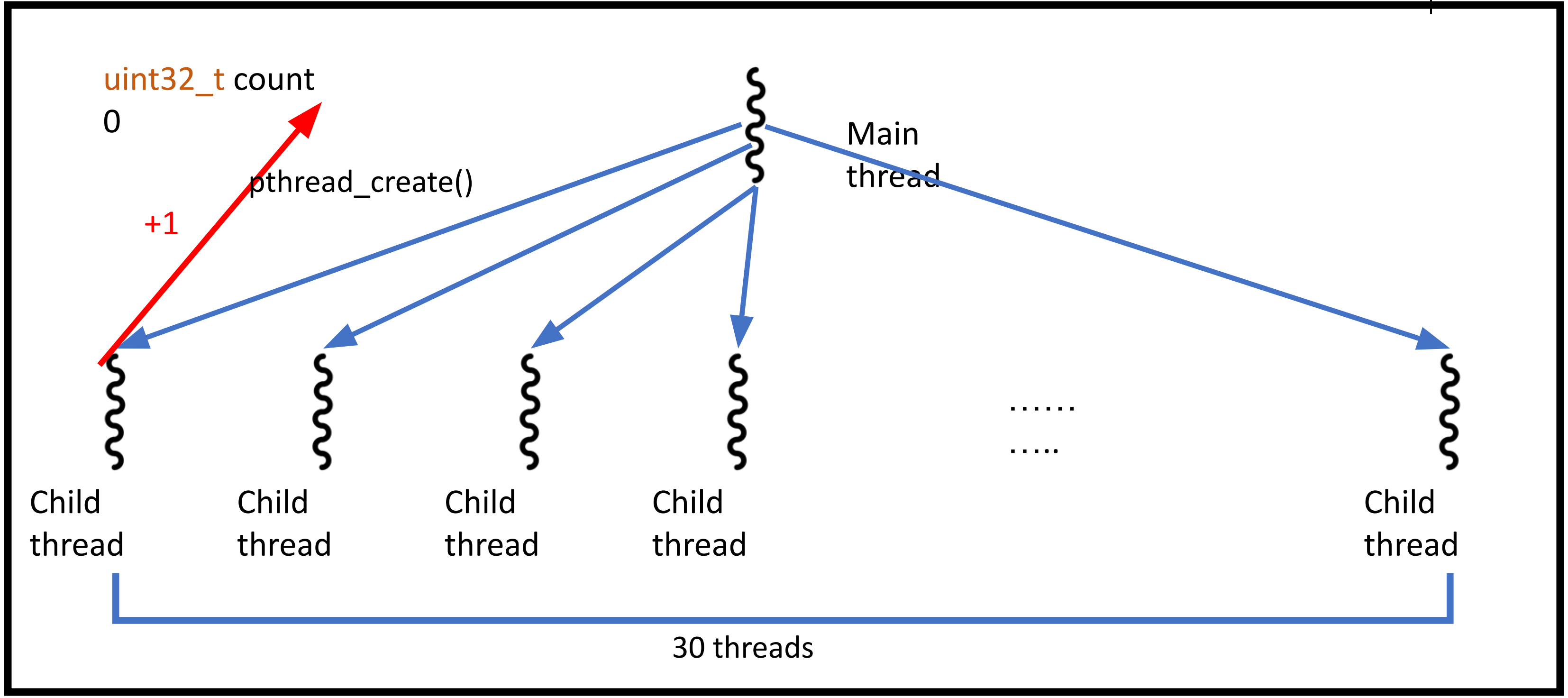
lock example

- https://github.com/purs3lab/ee469_examples/tree/master/lock_example
- Run 30 threads, each count upto 50
- Build code
  - $ make
- Run code
  - $ ./lock xchg   # shows the result of using xchg lock

lock example

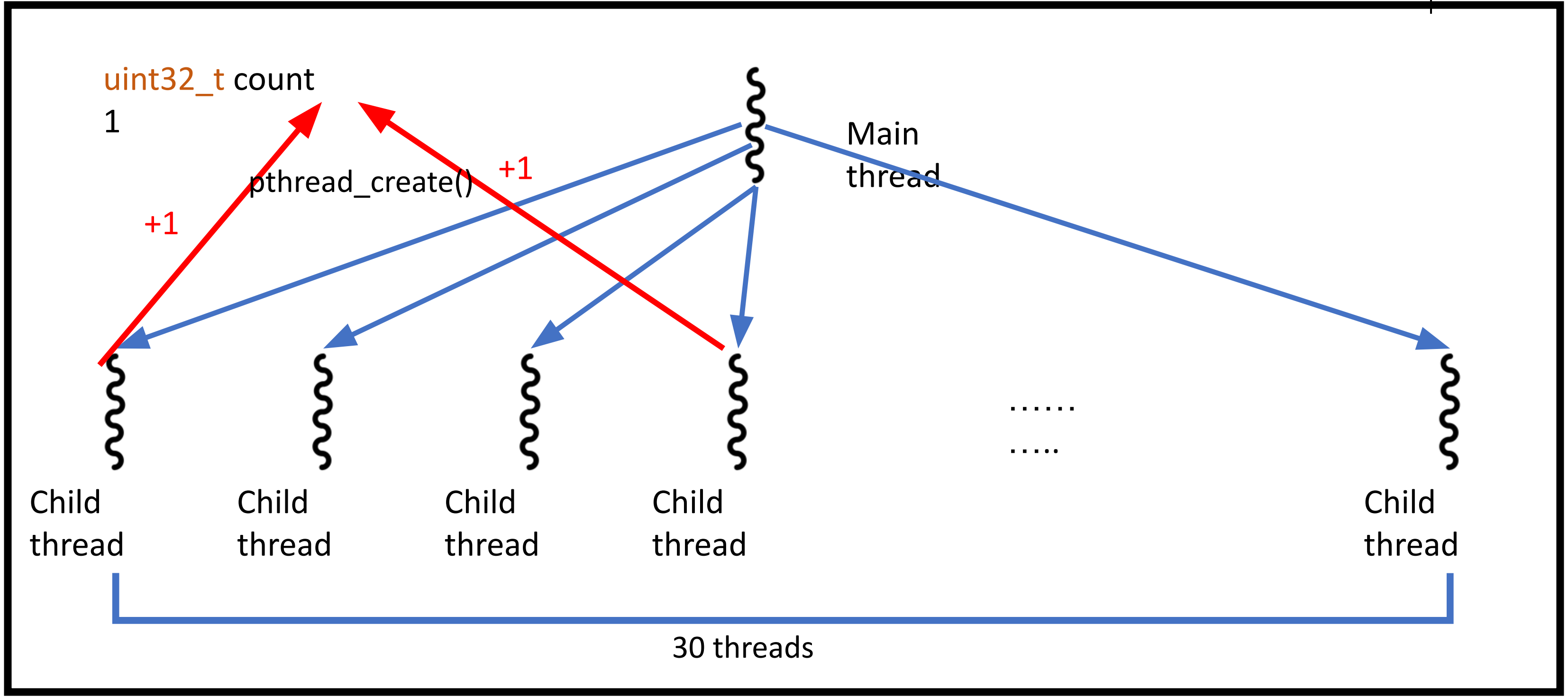


lock example



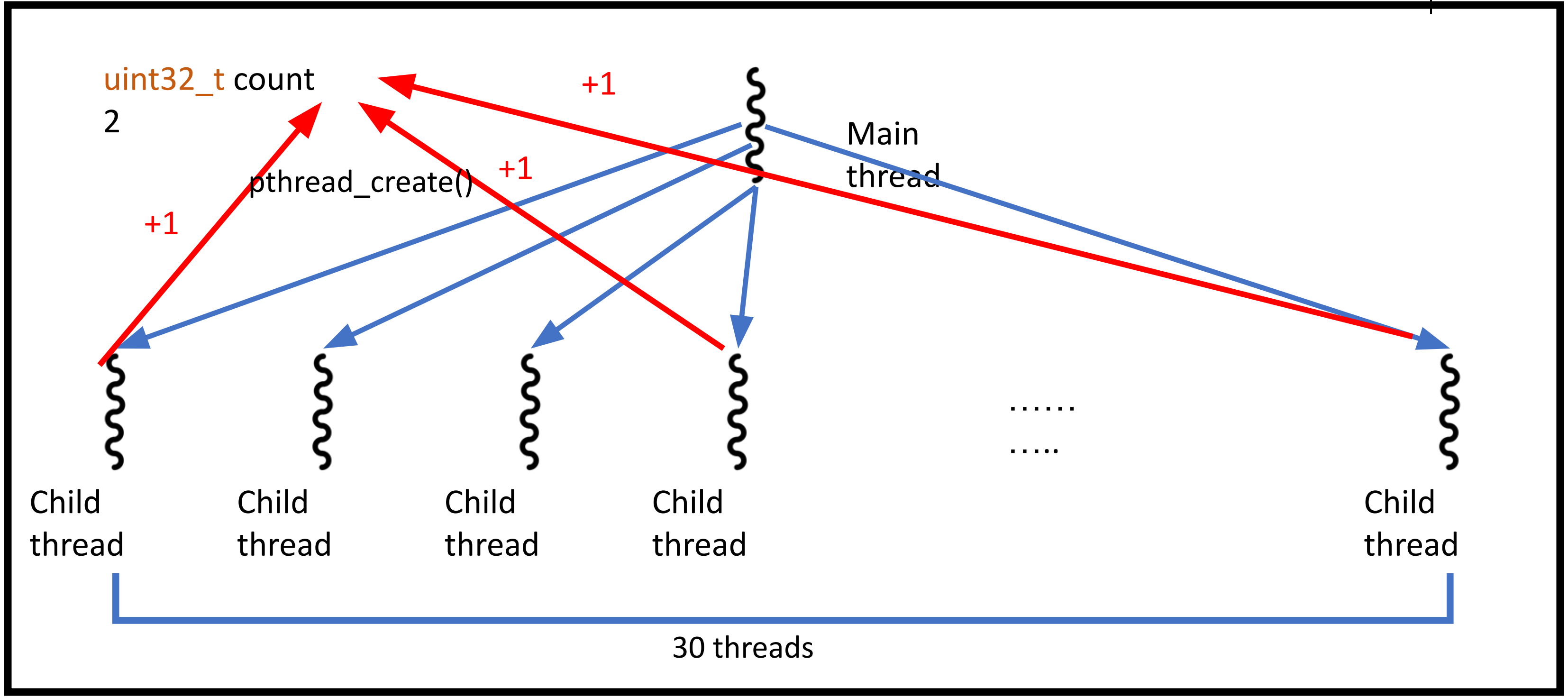
Each thread will increase count by 1 for 50 times

lock example



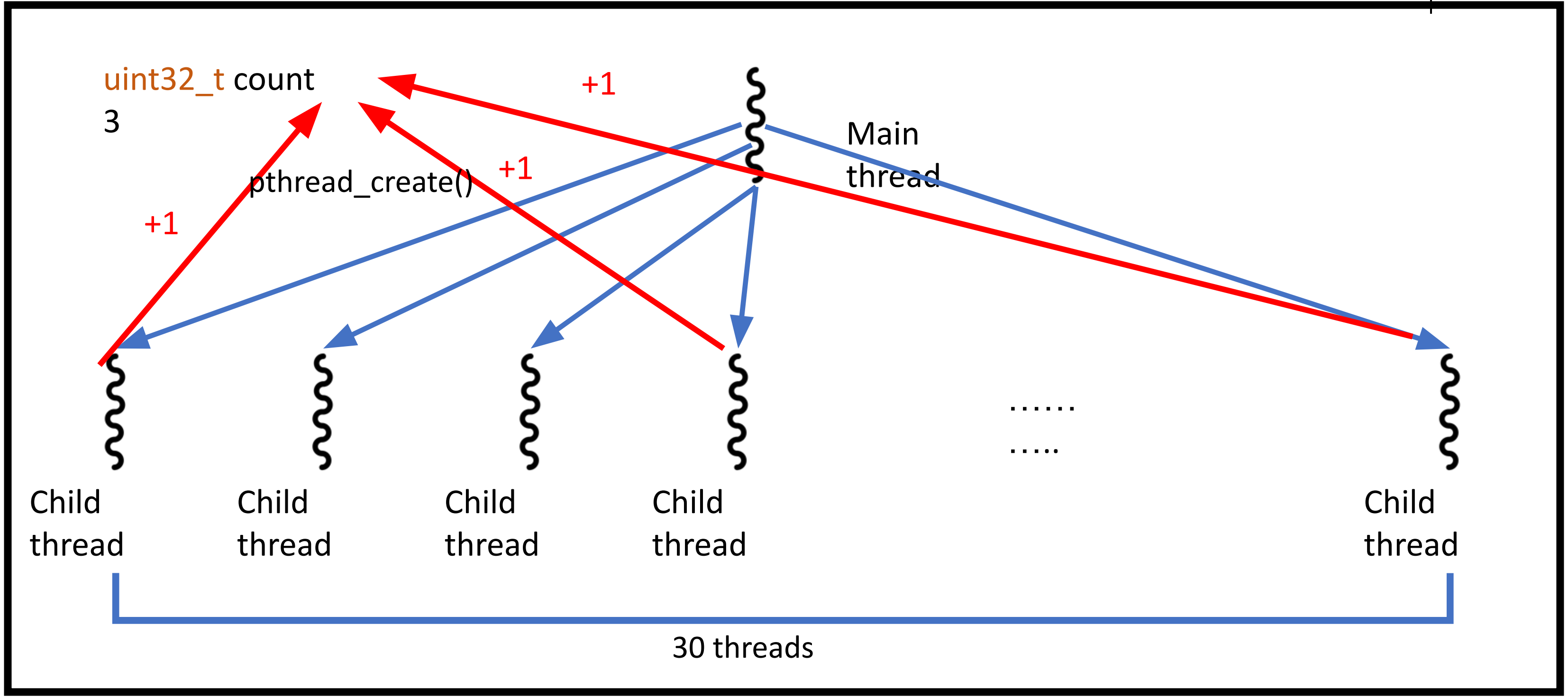
Each thread will increase count by 1 for 50 times

lock example



Each thread will increase count by 1 for 50 times

lock example



Each thread will increase count by 1 for 50 times

lock example

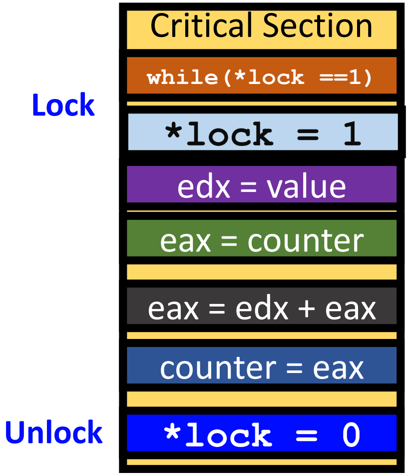
- Running
  - $ ./lock no # using no lock at all
  - $ ./lock bad # using a bad lock implementation
  - $ ./lock xchg # using xchg lock
  - $ ./lock cmpxchg # using lock cmpxchg
  - $ ./lock tts # using soft test-and-test & set with xchg
  - $ ./lock backoff # using exponential backoff cmpxchg
  - $ ./lock mutex # using pthread mutex

Manual Spinlock (bad_lock)

```
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 l becomes 0, e.g., `while(*lock == 1);`
      - Then, nobody runs in the critical section!
    - set *`lock` = 1
  - unlock(lock)
    - Set *`lock` = 0



Manual Spinlock (bad_lock)

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

- 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);
    }
}
```

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

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

Manual Spinlock (bad_lock)

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

- 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! **Can pass this if lock=0**

- 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);
    }
}
```

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

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

Manual Spinlock (bad_lock)

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

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

- 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! **Can pass this if lock=0**

Sets lock=1 to block others

- bad_unlock
  - Just set *lock as 0

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

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

Manual Spinlock (bad_lock)

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

- 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! **Can pass this if lock=0**

Sets lock=1 to block others

- 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;
}
```

Manual Spinlock (bad_lock)

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

```
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

- 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! **Can pass this if lock=0**

Sets lock=1 to block others

- bad_unlock
  - Just set *lock as 0

Sets lock=0 to release

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

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

Why does bad_lock doesn’t work?

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

Why does bad_lock doesn’t work?

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

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

Why does bad_lock doesn't work?

LOAD

STORE

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

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

Why does bad_lock doesn't work?

LOAD

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

STORE

Another thread might get scheduled here.

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

Why does bad_lock doesn't work?

- There is a room for race condition!

LOAD

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

STORE

Another thread might get scheduled here.

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

Why does bad_lock doesn't work?

- 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;
}
```

Why does bad_lock doesn't work?

- 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;
}
```

Recall: Why does this work for humans?

- Human can perform *test* (look for other person & milk) and *set* (leave note) at the same time.

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

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

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

Not like these four instructions…

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`
- 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

- `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` as 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 acquired the lock (it was 1)
    - `lock == 1` means some other thread acquired this…

Lock using xchg

- 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);
    }
}
```

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

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

Lock using xchg

- 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);
}
```

Does xchg_lock works?

- Yes!!

```
Running 30 threads each counting to 50 using xchg lock
Result:1500, Time taken: 2358.591000 ms
```

- Any issues?

Issues with xchg_lock

- xchg will always update the value
  - If lock == 0
    - lock = 1
    - eax = 0
  - If lock == 1
    - lock = 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…

Issues with xchg_lock

- xchg will always update the value
  - If lock == 0
    - lock = 1
    - eax = 0

    Swap with eax == 1, update lock to 1
  - If lock == 1
    - lock = 1
    - eax = 1

    Swap with eax == 1, update lock to 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…

No need to write when lock == 1

- Let’s not do that
  - xchg can’t do that

No need to write when lock == 1

- Let's not do that
  - xchg can't do that

- 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**)

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

```
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;
        }
    }
}
```

Lock using test and 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

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

```
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

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

Test and Set in x86

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

Test

Test-and-set

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

Lock using cmpxchg_lock

- Cmpxchg_lock
  - Use cmpxchg to set lock = 1
    - Do not update if lock == 1
    - Only write 1 to lock if lock == 0
- Xchg_unlock
  - Use xchg_unlock to set lock = 0
  - Because we have 1 writer and
  - This always succeeds…

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

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

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

Lock using cmpxchg_lock

- Cmpxchg_lock
  - Use cmpxchg to set lock = 1
    - Do not update if lock == 1
    - Only write 1 to lock if lock == 0
- Xchg_unlock
  - Use xchg_unlock to set lock = 0
  - Because we have 1 writer and
  - This always succeeds…

Critical Section

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

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

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

Reading fine print : x86 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 value regardless the result of comparision

tts_xchg_lock v/s cmpxchg_lock

- tts_xchg_lock is faster then cmpxchg_lock

Not everything in hardware is fast!

Observation 2: AddressSanitizer, despite being a software-only approach, performs on par with ICC-MPX and better than GCC-MPX. This unexpected result testifies that the HW-assisted performance improvements of MPX are offset by its complicated design. At the same time,

Using hardware features smartly

- 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

Summary

- Mutex is implemented with Spinlock
  - Waits until lock == 0 with a while loop (that's why it's called spin)
- Naïve code implementation
  - Load/Store must be atomic
- xchg is a "test and set" atom
  - Consistent, however, many c
- Lock cmpxchg is a "test and
  - But Intel implemented this a
- We can implement test-and-
  - Faster!
- We can also implement expo
  - Much faster! **Faster Than p**

```
./lock no
Running 30 threads each counting to 50 using no lock
Result:1400, Time taken: 3.913000 ms
./lock bad
Running 30 threads each counting to 50 using bad lock
Result:1465, Time taken: 2.256000 ms
./lock xchg
Running 30 threads each counting to 50 using xchg lock
Result:1500, Time taken: 853.585000 ms
./lock cmpxchg
Running 30 threads each counting to 50 using cmpxchg lock
Result:1500, Time taken: 12997.561000 ms
./lock tts
Running 30 threads each counting to 50 using tts lock
Result:1500, Time taken: 1.779000 ms
./lock backoff
Running 30 threads each counting to 50 using backoff lock
Result:1500, Time taken: 0.939000 ms
./lock mutex
Running 30 threads each counting to 50 using mutex lock
Result:1500, Time taken: 5.313000 ms
```

Other synchronization primitives

- We may want to have more than one thread/process to execute at same time

Producer

```
while (1) {

    produce an item;

    lock();
    insert(item to pool);
    unlock();
}
```

Consumer

```
While (1) {

    lock();
    remove(item from pool);
    unlock();

    consume the item;
}
```

How many producers/consumers can run at a given time?

Producer

```
while (1) {

    produce an item;

    lock();
    insert(item to pool);
    unlock();
}
```

Consumer

```
While (1) {

    lock();
    remove(item from pool);
    unlock();

    consume the item;
}
```

What we want!

- To be more efficient we want to be able to allow **more than one producer/consumer, i.e., equal to the number of items that can be inserted into/removed from the pool**

Producer

```
while (1) {

    produce an item;

    lock();
    insert(item to pool);
    unlock();
}
```

Consumer

```
While (1) {

    lock();
    remove(item from pool);
    unlock();

  consume the item;
}
```

Semaphore

A semaphore is like an **integer**, with three differences:

When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are **allowed to perform** are **increment** (increase by one) and **decrement** (decrease by one). *You cannot read the current value of the semaphore.*

When a thread **decrements** the semaphore, if the **result is negative**, the **thread blocks itself** and cannot continue until another thread increments the semaphore.

When a thread **increments** the semaphore, if there are **other threads waiting, one of the waiting threads gets unblocked**.

Semaphore operations

```
wait(S) {
  while (S<=0);
   S--;
}
```

```
signal(S) {
        S++;
}
```

Producers/consumers using Semaphores

Producer

```
while (1) {

    produce an item;

    lock();
    insert(item to pool);
    unlock();
}
```

Consumer

```
While (1) {

    lock();
    remove(item from pool);
    unlock();

    consume the item;
}
```

Init: FULL = 0; **EMPTY = N**;

Producers/consumers using Semaphores

Producer

```
while (1) {

    produce an item;

    lock();
    insert(item to pool);
    unlock();
    signal(FULL);
}
```

Consumer

```
While (1) {

    lock();
    remove(item from pool);
    unlock();

    consume the item;
}
```

Init: FULL = 0; **EMPTY = N**;

Producers/consumers using Semaphores

Producer

```
while (1) {

    produce an item;
    wait(EMPTY);
    lock();
    insert(item to pool);
    unlock();
    signal(FULL);
}
```

Consumer

```
While (1) {

    lock();
    remove(item from pool);
    unlock();

    consume the item;
}
```

Init: FULL = 0; **EMPTY = N**;

Producers/consumers using Semaphores

Producer

```
while (1) {

    produce an item;
    wait(EMPTY);
    lock();
    insert(item to pool);
    unlock();
    signal(FULL);
}
```

Consumer

```
While (1) {

    wait(FULL);
    lock();
    remove(item from pool);
    unlock();

    consume the item;
}
```

Init: FULL = 0; **EMPTY = N**;

Producers/consumers using Semaphores

Producer

```
while (1) {

    produce an item;
    wait(EMPTY);
    lock();
    insert(item to pool);
    unlock();
    signal(FULL);
}
```

Consumer

```
While (1) {

    wait(FULL);
    lock();
    remove(item from pool);
    unlock();
    signal(EMPTY);
    consume the item;
}
```

Init: FULL = 0; **EMPTY = N**;

Is Semaphore good for producers/consumers?

Need to know the size of buffer!

How to accommodate dynamic pool size?