Foundation and the cost of Synchronization

Jinyang Li

Based on the slides of Tiger Wang
What you’ve learnt

• Challenge of Multi-threaded programming
  – Races, deadlocks
• Pthread library’s synchronization primitives:
  – mutex, conditional variable
Today

- How does pthread’s mutex work?
- What’s the cost of synchronization?
Implement a lock: a naive attempt

typedef struct {
    int busy;
} mutex_t;

void lock_init(mutex_t *mu) {
    mu->busy = 0;
}

void lock(mutex_t *mu) {
    while(mu->busy) {} // Busy wait
    mu->busy = 1;
}

void unlock(mutex_t *mu) {
    mu->busy = 0;
}

This style of locking is called “Spin Lock”
Is the naive implementation correct?

Thread 1

\[
\text{mu->busy = 1}
\]
\[
\text{while (mu->busy)}
\]
\[
//mu->busy=1
\]
\[
\text{mu->busy = 0}
\]
\[
\text{while (mu->busy)}
\]
\[
//mu->busy=0
\]
\[
\text{mu->busy = 1}
\]

Both threads grabbed lock

Thread 2

\[
\text{mu->busy = 1}
\]
\[
\text{while (mu->busy)}
\]
\[
//mu->busy=1
\]
\[
\text{mu->busy = 0}
\]
\[
\text{while (mu->busy)}
\]
\[
//mu->busy=0
\]
\[
\text{mu->busy = 1}
\]

Thread 3

\[
\text{mu->busy = 1}
\]
\[
\text{while (mu->busy)}
\]
\[
//mu->busy=1
\]
are x86 instructions atomic?

```
mov 0x20072d(%rip),%eax // load global into %eax
add $0x1,%eax           // update %eax by 1
mov %eax,0x200724(%rip) // restore global with %eax
```

Thread 1

```
mov 0x20072d(%rip),%eax
add $0x1,%eax
mov %eax,0x200724(%rip)
```

Thread 2

```
mov 0x20072d(%rip),%eax
add $0x1,%eax
mov %eax,0x200724(%rip)
```
x86 atomic instructions

- We need hardware support to implement locks
- x86 provide atomic instructions:
  - An atomic instruction performs its reads/writes on one or more memory locations atomically.

A conceptual model for how CPU executes atomic instruction:

1. Load data to CPU’s local buffer
2. Calculate the result
3. Store data back to memory

Unlock the memory address

Multiple instructions’ memory access do NOT interleave (but execute one after another)
## 2 types of atomic instructions

<table>
<thead>
<tr>
<th>atomic with lock prefix</th>
<th>atomic instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, sub</td>
<td>mov</td>
</tr>
<tr>
<td>inc, dec</td>
<td>xchchg</td>
</tr>
<tr>
<td>and, or, xor</td>
<td>...</td>
</tr>
<tr>
<td>cmpxchg</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Atomic add (using lock prefix)

How about “directly” adding to memory?

ordinary x86 add is not atomic
Atomic add (using lock prefix)

Using atomic add to increment global

 LOCK add $0x1, 0x20072d(%rip) // increment global by 1

LOCK prefix makes add atomic
**xchg instruction**

- `xchg op1, op2`
  - Swap op1 with op2

- `xchg reg, reg`
- `xchg reg, mem`
- `xchg mem, reg`

- `xchgq %rax, (%rdi)`

- `movq %rax, %r10`
- `movq (%rdi), %rax`
- `movq %r10, (%rdi)`

executes atomically
Wrap `xchg` in a C function

```c
int xchg(int *ptr, int x) {
    asm volatile("xchgl %0,%1"
    :"=r" (x)
    :"m" (ptr)
    :"memory");
    return x;
}
```

Atomically store `x` in the memory pointed by `ptr`, Return the old value stored at `ptr`.
Implement a lock using xchg

typedef struct {
    int busy;
} mutex_t;

void lock_init(mutex_t *mu) {
    mu->busy = 0;
}

void lock(mutex_t *mu) {
    while (xchg(&mu->busy, 1) != 0) {}  
}

void unlock(int *mu) {
    xchg(&mu->busy, 0);
}
Spin lock based on xchg

\[
\begin{align*}
\text{xchg}(&\mu->\text{busy},1) &= 0 \\
\text{while} \ (\text{xchg}(&\mu->\text{busy},1)! = 0) &\quad //\text{xchg}(..) = 1 \\
\text{while}(\text{xchg}(&\mu->\text{busy},1)! = 0) &\quad //\text{xchg}(..) = 1 \\
\text{xchg}(&\mu->\text{busy},0) &\quad //\text{xchg}(..) = 1 \\
\text{while} \ (\text{xchg}(&\mu->\text{busy},1)! = 0) &\quad //\text{xchg}(..) = 0 \\
\text{while}(\text{xchg}(&\mu->\text{busy},1)! = 0) &\quad //\text{xchg}(..) = 1
\end{align*}
\]
Why not always use spin locks?

- If lock is not available, thread busy waits (spins).
- Not efficient if critical section is long.
- Better alternative: if one thread blocks, execute another thread that can make progress
  - Need help from OS kernel to put one thread on hold and schedule another.
Futex syscall

- `futex(int *addr, FUTEX_WAIT, val, ...)`
  - atomically checks `*addr == val` and puts calling thread on OS’ wait queue for `addr` if equality holds.

- `futex(int *addr, FUTEX_WAKE, n, ...)`
  - wakes `n` threads on OS’ wait queue for `addr`.
typedef struct {
    int busy;
} mutex_t;

void mutex_init(mutex_t *mu) {
    mu->busy = 0;
}

void mutex_lock(mutex_t *mu) {
    while(xchg(&mu->busy, 1) != 0) {
        ???
    }
}

void mutex_unlock(mutex_t *mu) {
    xchg(&mu->busy, 0);
    ???
}
The cost of synchronization
Basic Idea of Cache Coherence

CPU 0
IR
RAX

Cache 0

CPU 1
IR
RAX

Cache 1

Memory bus

Memory

0x138 0x8
0x130 0x7
0x128 0x6
0x120 0x5
0x118 0x4
0x110 0x3
0x108 0x2
0x100 0x1
Basic Idea of Cache Coherence

CPU 0
IR: addq 1, (%rax)
RAX: 0x100
Cache 0: 0x1, 0x2, 0x3 ... 0x8

CPU 1
IR
RAX
Cache 1

Memory bus

Memory:
0x138: 0x8
0x130: 0x7
0x128: 0x6
0x120: 0x5
0x118: 0x4
0x110: 0x3
0x108: 0x2
0x100: 0x1
Basic Idea of Cache Coherence

CPU 0
- IR: `addq 1, (%rax)`
- RAX: 0x100

Cache 0
- 0x2, 0x2, 0x3 ... 0x8

CPU 1
- IR
- RAX

Cache 1

Memory bus

Memory
- 0x138: 0x8
- 0x130: 0x7
- 0x128: 0x6
- 0x120: 0x5
- 0x118: 0x4
- 0x110: 0x3
- 0x108: 0x2
- 0x100: 0x1
Basic Idea of Cache Coherence

CPU 0
IR: addq 1, (%rax)
RAX: 0x100

Cache 0
0x2, 0x2, 0x3 ... 0x8

CPU 1
IR: movq (%rax), %rax
RAX: 0x100

Cache 1

Memory
0x138: 0x8
0x130: 0x7
0x128: 0x6
0x120: 0x5
0x118: 0x4
0x110: 0x3
0x108: 0x2
0x100: 0x1

Memory bus
Basic Idea of Cache Coherence

- **CPU 0**
  - IR: `addq 1, (%rax)`
  - RAX: 0x100
  - Cache 0: 0x2, 0x2, 0x3 ... 0x8

- **CPU 1**
  - IR: `movq (%rax), %rax`
  - RAX: 0x100
  - Cache 1: Cache miss

- **Memory**

```
0x138 : 0x8
0x130 : 0x7
0x128 : 0x6
0x120 : 0x5
0x118 : 0x4
0x110 : 0x3
0x108 : 0x2
0x100 : 0x1
```
Basic Idea of Cache Coherence

CPU 0
IR: addq 1, (%rax)
RAX: 0x100

Cache 0
0x2, 0x2, 0x3 ... 0x8

CPU 1
IR: movq (%rax), %rax
RAX: 0x100

Cache 1
0x2, 0x2, 0x3 ... 0x8

Memory
0x138: 0x8
0x130: 0x7
0x128: 0x6
0x120: 0x5
0x118: 0x4
0x110: 0x3
0x108: 0x2
0x100: 0x1
Basic Idea of Cache Coherence

CPU 0
IR: addq 1, (%rax)
RAX: 0x100
Cache 0
0x2, 0x2, 0x3 ... 0x8

CPU 1
IR: movq (%rax), %rax
RAX: 0x2
Cache 1
0x2, 0x2, 0x3 ... 0x8

Memory bus
Memory:
0x138: 0x8
0x130: 0x7
0x128: 0x6
0x120: 0x5
0x118: 0x4
0x110: 0x3
0x108: 0x2
0x100: 0x1
Basic Idea of Cache Coherence

CPU 0
- IR: addq 1, (%rax)
- RAX: 0x100
- Cache 0: 0x2, 0x2, 0x3 ... 0x8

CPU 1
- IR: movq (%rax), %rax
- RAX: 0x2
- Cache 1: 0x2, 0x2, 0x3 ... 0x8

Memory
- 0x138: 0x8
- 0x130: 0x7
- 0x128: 0x6
- 0x120: 0x5
- 0x118: 0x4
- 0x110: 0x3
- 0x108: 0x2
- 0x100: 0x1
Basic Idea of Cache Coherence

CPU 0
- IR: addq 1, (%rax)
- RAX: 0x100
- Cache 0: 0x2, 0x2, 0x3 ... 0x8

CPU 1
- IR: movq (%rax), %rax
- RAX: 0x2
- Cache 1: 0x2, 0x2, 0x3 ... 0x8

Invalidate

Memory bus

Memory:
- 0x138: 0x8
- 0x130: 0x7
- 0x128: 0x6
- 0x120: 0x5
- 0x118: 0x4
- 0x110: 0x3
- 0x108: 0x2
- 0x100: 0x1
Basic Idea of Cache Coherence

CPU 0
- IR: `addq 1, (%rax)`
- RAX: 0x100
- Cache 0: 0x3, 0x2, 0x3 ... 0x8

CPU 1
- IR: `movq (%rax), %rax`
- RAX: 0x2
- Cache 1: 0x2, 0x2, 0x3 ... 0x8

Memory
- 0x138: 0x8
- 0x130: 0x7
- 0x128: 0x6
- 0x120: 0x5
- 0x118: 0x4
- 0x110: 0x3
- 0x108: 0x2
- 0x100: 0x1
Basic Idea of Cache Coherence

CPU 0
IR: addq 1, (%rax)
RAX: 0x100
Cache 0
0x3, 0x2, 0x3 ... 0x8

CPU 1
IR: movq (%rax), %rax
RAX: 0x2
Cache 1
0x2, 0x2, 0x3 ... 0x8

Memory bus
Memory
0x138: 0x8
0x130: 0x7
0x128: 0x6
0x120: 0x5
0x118: 0x4
0x110: 0x3
0x108: 0x2
0x100: 0x1
Basic Idea of Cache Coherence

CPU 0
IR: addq 1, (%rax)
RAX: 0x100
Cache 0:
0x3, 0x2, 0x3 ... 0x8

CPU 1
IR: lea 0x100, %rax
RAX: 0x2
Cache 1:
0x2, 0x2, 0x3 ... 0x8

Memory bus

Memory:
0x138: 0x8
0x130: 0x7
0x128: 0x6
0x120: 0x5
0x118: 0x4
0x110: 0x3
0x108: 0x2
0x100: 0x1
Basic Idea of Cache Coherence

CPU 0
IR: addq 1, (%rax)
RAX: 0x100
Cache 0: 0x3, 0x2, 0x3 ... 0x8

CPU 1
IR: lea 0x100, %rax
RAX: 0x100
Cache 1: 0x2, 0x2, 0x3 ... 0x8

Memory bus

Memory:
0x138: 0x8
0x130: 0x7
0x128: 0x6
0x120: 0x5
0x118: 0x4
0x110: 0x3
0x108: 0x2
0x100: 0x1
Basic Idea of Cache Coherence

CPU 0
IR: addq 1, (%rax)
RAX: 0x100

Cache 0
0x3, 0x2, 0x3 ... 0x8

CPU 1
IR: movq (%rax), %rax
RAX: 0x100

Cache 1
0x2, 0x2, 0x3 ... 0x8
Cache miss

Memory
0x138: 0x8
0x130: 0x7
0x128: 0x6
0x120: 0x5
0x118: 0x4
0x110: 0x3
0x108: 0x2
0x100: 0x1
Basic Idea of Cache Coherence

CPU 0
- IR: addq 1, (%rax)
- RAX: 0x100
- Cache 0
  - Memory:
    - 0x138: 0x8
    - 0x130: 0x7
    - 0x128: 0x6
    - 0x120: 0x5
    - 0x118: 0x4
    - 0x110: 0x3
    - 0x108: 0x2
    - 0x100: 0x1

CPU 1
- IR: movq (%rax), %rax
- RAX: 0x100
- Cache 1
  - Memory:
    - 0x138: 0x8
    - 0x130: 0x7
    - 0x128: 0x6
    - 0x120: 0x5
    - 0x118: 0x4
    - 0x110: 0x3
    - 0x108: 0x2
    - 0x100: 0x1

Memory bus
- transfer
Basic Idea of Cache Coherence

CPU 0
- IR: addq 1, (%rax)
- RAX: 0x100
- Cache 0: 0x3, 0x2, 0x3 ... 0x8

CPU 1
- IR: movq (%rax), %rax
- RAX: 0x3
- Cache 1: 0x3, 0x2, 0x3 ... 0x8

Memory
- 0x138: 0x8
- 0x130: 0x7
- 0x128: 0x6
- 0x120: 0x5
- 0x118: 0x4
- 0x110: 0x3
- 0x108: 0x2
- 0x100: 0x1
Update global variable

addq 1, (%rax)  CPU 0  
global: 0  \rightarrow  1  addq 1, (%rax)  CPU 1

cache miss

cache miss

0  load cacheline from memory

0  load cacheline from memory

0  load data into cpu buffer

0  load data into cpu buffer

1  calculation

1  calculation

invalidate cpu1’s cacheline

cache invalidated

1  write data from cpu buffer to cache

1  write data from cpu buffer to cache

be invalidated

invalidate cpu0’s cacheline

load cacheline from CPU 0

load cacheline from CPU 0

write data from cpu buffer to cache

write data from cpu buffer to cache
Update global variable with lock

CPU 0
lock; addq 1, (%rax)
lock cacheline/ address /memory bus
cache miss
load cacheline from memory
load into CPU buffer and calculate
write data from cpu buffer to cache
Instruction retired

global: 2

CPU 1
lock; addq 1, (%rax)
lock cacheline/ address /memory bus
cache miss
load cacheline from CPU0
load into CPU buffer and calculate
write data from cpu buffer to cache
Instruction retired

invalidate cpu0’s cacheline

cache invalidated
### Synchronization Cost

<table>
<thead>
<tr>
<th></th>
<th>No Lock</th>
<th>Atomic Instruction</th>
<th>Spin Lock</th>
<th>Pthread Mutex</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single thread</strong></td>
<td>5.5</td>
<td>19.3</td>
<td>24</td>
<td>50.4</td>
</tr>
<tr>
<td><strong>Two threads / Same variable</strong></td>
<td>3.0</td>
<td>32.9</td>
<td>124</td>
<td>166.8</td>
</tr>
<tr>
<td><strong>Two threads / Same cacheline</strong></td>
<td>3.1</td>
<td>30</td>
<td>63</td>
<td>124</td>
</tr>
<tr>
<td><strong>Two threads / Different cachelines</strong></td>
<td>2.9</td>
<td>10</td>
<td>13</td>
<td>25.8</td>
</tr>
</tbody>
</table>

- Synchronizing per add makes multi-threading slower than single thread.
- Synchronization magnifies the cost of cache coherence.
A brief note about lab 5

• How to implement a read-write lock?

typedef struct {
    ...
} rwl;

void rwl_init(rwl *l);
int rwl_nwaiters(rwl *l);
int rwl_rlock(rwl *l, struct timespec *expire);
int rwl_runlock(rwl *l);
int rwl_wlock(rwl *l, struct timespec *expire);
int rwl_wunlock(rwl *l);
Implementing a read-write lock

• Need to track mode of the grabbed lock?
  – read (“shared”) vs. write (“exclusive”)

• Need to track how many readers have the lock?
  – Multiple readers can grab lock in “read” mode

• Need to track waiting threads (waiters)?
  – If there are waiters, we should wake them up upon lock release
  – Shall we track waiting writers and readers separately?
    • Lab requires you to prioritize writer, i.e. a waiting writer should get the lock over waiting readers
A brief note about lab 5

- How to implement a read-write lock? An example.

```c
typedef struct {
    int n_waiting_readers;
    int n_waiting_writers;
    int n_writers;
    int n_readers;
    pthread_mutex_t mu;
    pthread_mutex_t cond;  // all waiting threads block on this cond
} rwl;
```

What’s the state of the lock if it’s locked on “exclusive” mode?
What’s the state of the lock if it’s locked on “shared” mode?
An example `rwl_wlock`: it prioritizes readers instead of writers

```c
int rwl_wlock(rwl *l, struct timespec *expire) {
    pthread_mutex_lock(&l->mu);
    l->n_waiting_writers++;
    //if lock has been locked, block
    //if lock has waiting readers, also block
    while (l->n_waiting_readers > 0
           || l->n_writers > 0
           || l->n_readers > 0) {
        pthread_cond_wait(&l->cond, &l->mu);
    }
    //update lock state
    l->n_waiting_writers--;
    l->n_writers++;
    pthread_mutex_unlock(&l->mu);
}
```
An example rwl_wlock: it prioritizes readers instead of writers

```c
int rwl_wlock(rwl *l, struct timespec *expire) {
  pthread_mutex_lock(&l->mu);
  l->n_waiting_writers++;
  while (l->n_waiting_readers > 0
       || l->n_writers > 0
       || l->n_readers > 0) {
    if (cond_timedwait(&l->cond,&l->mu, expire)==ETIMEDOUT){
      l->n_waiting_writers--;
      pthread_mutex_unlock(&l->m);
      return ETIMEDOUT;
    }
  }
  //update lock state
  l->n_waiting_writers--;
  l->n_writers++;
  pthread_mutex_unlock(&l->mu);
}
```