Foundation and Cost of Synchronization

Shuai Mu

Based on the slides of Tiger Wang
Example

global++

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

```
Example

global++ ➔

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

But how about?

Example

global++ ➔

add $0x1, 0x20072d(%rip) // directly update global by 1
```
add is not atomic

```asm
add $0x1, 0x20072d(%rip) // directly update global by 1
```

CPU

1. Load the data to CPU’s local buffer (invisible)
2. Calculate the result
3. Store the data back to memory
Atomic instructions are special hardware instructions that perform an operation on one or more memory locations atomically.
Atomic instructions are special hardware instructions that perform an operation on one or more memory locations atomically.

**CPU**

*Lock the memory address*
1. Load the data to CPU’s local buffer (invisible)
2. Calculate the result
3. Store the data back to memory

*Unlock the memory address*
Atomic instructions are special hardware instructions that perform an operation on one or more memory locations atomically.

**CPU**

**Lock the memory address**
1. Load the data to CPU’s local buffer (invisible)
2. Calculate the result
3. Store the data back to memory

**Unlock the memory address**

Add `lock` prefix to make the instruction be atomic
**atomic instructions**

<table>
<thead>
<tr>
<th>atomic instructions</th>
<th>atomic with lock prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>add / sub</td>
</tr>
<tr>
<td>xchg</td>
<td>inc / dec</td>
</tr>
<tr>
<td>...</td>
<td>and / or / xorl</td>
</tr>
<tr>
<td></td>
<td>cmpxchg</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Reading a memory operand, performing some operation on it, write it back to the memory.
How to implement a lock?

// 1: busy, 0: free
int mutex = 0;

void lock(int *mu) {
    while(*mu == 1) {} // Busy wait
    *mu = 1;
}

void unlock(int *mu) {
    *mu = 0;
}

Busy wait
This style of locking is called “Spin Lock”
How to implement a lock?

```c
int mutex = 0;

thread-1
void lock(int *mu) {
    while(*mu == 1) {}
}

*mu = 1;
}

thread-2
void lock(int *mu) {
    while(*mu == 1) {}
    *mu = 1;
}
```
xchg instruction

xchg op1, op2
- Swap the op1 operand with the op2 operand

xchg reg, reg
xchg reg, mem
xchg mem, reg
xchg instruction

xchg op1, op2
- Swap the op1 operand with the op2 operand

xchg reg, reg
xchg reg, mem
xchg mem, reg

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

Atomically store the memory pointed by ptr with x, then return the old value stored at ptr.
Spin Lock based on xchg

// 1: busy, 0: free
int mutex = 0;

void lock(int *mu) {
    while(xchg(mu, 1) != 0) {}  
}

void unlock(int *mu) {
    xchg(mu, 0);
}

Thread-1 Thread-2
xchg(mu,1)=0
xchg(mu,1)=?
continue to busy wait
Why not always use spin locks?

- If spin 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`.

Will not be tested in final
A simple pthread_mutex impl.

typedef struct {
    int locked;
} mutex_t;

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

void mutex_lock(mutex_t *mu) {
    while(xchg(&mu->locked, 1) != 0) {
        futex_wait(&mu->locked, 1, ..);
    }
}

void mutex_unlock(mutex_t *mu) {
    xchg(&mu->locked, 0);
    futex_wake(&mu->locked, 1, ..);
}

• Actual pthread mutex and conditional variable are much more complex for better performance.
• For more information, google “futexes are tricky” by Ulrich Drepper

Not efficient as futex_wake is called even if no thread is waiting.
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: \texttt{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
- Contents: 0x2, 0x2, 0x3 ... 0x8

CPU 1
- IR
- RAX

Cache 1
- Blank

Memory
- Contents: 0x138, 0x130, 0x128, 0x120, 0x118, 0x110, 0x108, 0x100

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**
- Empty

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

**Memory bus**

---

- `addq 1, (%rax)`: Add 1 to the value in the memory location pointed to by `%rax`.
- `movq (%rax), %rax`: Move the value at the memory location pointed to by `%rax` to `%rax`.

---

- `%rax`: A register in the CPU.
Basic Idea of Cache Coherence

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

- **Cache 0**
  - Data: 0x2, 0x2, 0x3 ... 0x8

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

- **Cache 1**
  - Data: Cache miss

- **Memory**
  - Addresses:
    - 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 bus

Memory

**transfer**

```
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
- 0x100
  - 0x2
  - 0x3
  - 0x4
  - 0x5
  - 0x6
  - 0x7
  - 0x8

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: 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**
  - Contents: 0x2, 0x2, 0x3 ... 0x8

- **CPU 1**
  - IR: `movq (%rax), %rax`
  - RAX: 0x2

- **Cache 1**
  - Contents: 0x2, 0x2, 0x3 ... 0x8

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

- **Invalidation**
  - From CPU 0 to Cache 0 and CPU 1 to Cache 1
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
- 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:
- 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 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

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

Memory

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

Transfer from CPU 0 to CPU 1
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 bus

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
  cache miss
addq 1, (%rax)  CPU 1
  cache miss
```
Update global variable

`addq 1, (%rax)`   **CPU 0**

  cache miss

  load cacheline from memory

`addq 1, (%rax)`   **CPU 1**

  cache miss

  load cacheline from memory
Update global variable

```
addq 1, (%rax)  CPU 0
  cache miss
  ↓
  load cacheline from memory
  ↓
  load data into cpu buffer

addq 1, (%rax)  CPU 1
  cache miss
  ↓
  load cacheline from memory
  ↓
  load data into cpu buffer
```
Update global variable

addq 1, (%rax)  **CPU 0**

- cache miss
- load cacheline from memory
- load data into cpu buffer
- calculation

addq 1, (%rax)  **CPU 1**

- cache miss
- load cacheline from memory
- load data into cpu buffer
- calculation
Update global variable

```
addq 1, (%rax)  CPU 0
  cache miss
  ↓
  load cacheline from memory
  ↓
  load data into cpu buffer
  ↓
  calculation
  ↓
  invalidate cpu1’s cacheline

addq 1, (%rax)  CPU 1
  cache miss
  ↓
  load cacheline from memory
  ↓
  load data into cpu buffer
  ↓
  calculation
```

be invalidated
Update global variable

CPU 0

- cache miss
- load cacheline from memory
- load data into cpu buffer
- calculation
- invalidate cpu1’s cacheline
- restore data from cpu buffer to cache

CPU 1

- cache miss
- load cacheline from memory
- load data into cpu buffer
- calculation
- invalidate cpu1’s cacheline
- be invalidated
- cache miss
Update global variable

CPU 0

addq 1, (%rax)

cache miss

load cacheline from memory

load data into cpu buffer

calculation

invalidate cpu1’s cacheline

restore data from cpu buffer to cache

CPU 1

addq 1, (%rax)

cache miss

load cacheline from memory

load data into cpu buffer

calculation

be invalidated

load cacheline from CPU 0
Update global variable

CPU 0
- addq 1, (%rax)
- cache miss
  - load cacheline from memory
  - load data into cpu buffer
  - calculation
  - invalidate cpu1’s cacheline
  - restore data from cpu buffer to cache

CPU 1
- addq 1, (%rax)
- cache miss
  - load cacheline from memory
  - load data into cpu buffer
  - calculation
  - be invalidated
  - cache miss
    - load cacheline from CPU 0
    - invalidate cpu0’s cacheline
Update global variable

```plaintext
addq 1, (%rax)  CPU 0
   cache miss
      ↓
load cacheline from memory
      ↓
load data into cpu buffer
      ↓
calculation
      ↓
invalidate cpu1’s cacheline
      ↓
restore data from cpu buffer to cache
```

```plaintext
addq 1, (%rax)  CPU 1
   cache miss
      ↓
load cacheline from memory
      ↓
load data into cpu buffer
      ↓
calculation
      ↓
be invalidated
```

```plaintext
load cacheline from CPU 0
      ↓
invalidate cpu0’s cacheline
      ↓
restore data from cpu buffer to cache
```
Update global variable

\[
\begin{align*}
\text{CPU 0} & \quad \text{global: } 0 \rightarrow 1 \\
\text{addq 1, (%rax)} & \\
\text{cache miss} & \\
0 & \quad \text{load cacheline from memory} \\
0 & \quad \text{load data into cpu buffer} \\
1 & \quad \text{calculation} \\
& \quad \text{invalidate cpu1's cacheline} \\
1 & \quad \text{restore data from cpu buffer to cache} \\
\text{CPU 1} & \\
\text{addq 1, (%rax)} & \\
\text{cache miss} & \\
0 & \quad \text{load cacheline from memory} \\
0 & \quad \text{load data into cpu buffer} \\
1 & \quad \text{calculation} \\
& \quad \text{be invalidated} \\
1 & \quad \text{restore data from cpu buffer to cache} \\
\end{align*}
\]
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 calculation
restore data from cpu buffer to cache
Instruction retired

CPU 1
lock; addq 1, (%rax)
lock cacheline/ address /memory bus
**Update global variable with lock**

CPU 0

1. lock; addq 1, (%rax)
2. lock cacheline/ address /memory bus
3. cache miss
4. load cacheline from memory
5. load into CPU buffer and calculation
6. restore data from cpu buffer to cache
7. Instruction retired

CPU 1

1. lock; addq 1, (%rax)
2. lock cacheline/ address /memory bus
3. cache miss
4. load cacheline from CPU0
5. load into CPU buffer and calculation
6. invalidate cpu0’s cacheline
7. restore data from cpu buffer to cache
8. Instruction retired
9. be 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>Single thread</td>
<td>5.5</td>
<td>19.3</td>
<td>24</td>
<td>50.4</td>
</tr>
<tr>
<td>Two threads / Same variable</td>
<td>3.0</td>
<td>32.9</td>
<td>124</td>
<td>166.8</td>
</tr>
<tr>
<td>Two threads / Same cacheline</td>
<td>3.1</td>
<td>30</td>
<td>63</td>
<td>124</td>
</tr>
<tr>
<td>Two threads / Different cachelines</td>
<td>2.9</td>
<td>10</td>
<td>13</td>
<td>25.8</td>
</tr>
</tbody>
</table>

Cycles / Operation

Synchronization magnify the cost of cache coherence