Synchronization
+
Cache Coherence

Chris Rossbach + Calvin Lin
CS380p
Today

• Reminder: Homework & Reading
• Foundations
  • Synchronization Implementation
  • Cache coherence
Review: Schedules/Interleavings

Model of concurrent execution

• Interleave statements from each thread into a single thread
• If any interleaving yields incorrect results, synchronization is needed

Thread 1

```plaintext
tmp1 = X;
tmp1 = tmp1 + 1;
X = tmp1;
```

Thread 2

```plaintext
tmp2 = X;
tmp2 = tmp2 + 1;
X = tmp2;
```
Review: Schedules/Interleavings

Model of concurrent execution

• Interleave statements from each thread into a single thread

• If any interleaving yields incorrect results, synchronization is needed

If X==0 initially, X == 1 at the end. WRONG result!
Locks implement Mutual Exclusion

Mutual exclusion ensures only safe interleavings

- *But it limits concurrency, and hence scalability/performance*
Implementing Locks

```c
int lock_value = 0;
int* lock = &lock_value;
```
Implementing Locks

```c
int lock_value = 0;
int* lock = &lock_value;

lock::acquire() {
    while (*lock == 1) //spin
        *lock = 1;
}
```
Implementing Locks

```cpp
int lock_value = 0;
int* lock = &lock_value;

lock::acquire() {
    while (*lock == 1) //spin
        *lock = 1;
}

lock::release() {
    *lock = 0;
}
```
Implementing Locks

```c
int lock_value = 0;
int* lock = &lock_value;

lock::acquire() {
    while (*lock == 1)  //spin
        *lock = 1;
}

lock::release() {
    *lock = 0;
}
```

What are the problem(s) with this?

- A. CPU usage
- B. Memory usage
- C. lock::acquire() latency
- D. Memory bus usage
- E. Does not work
Multiprocessor Cache Coherence

\[ F = ma \]
Multiprocessor Cache Coherence

Physics | Concurrency

\[ F = ma \sim coherence \]
Multiprocessor Cache Coherence
Multiprocessor Cache Coherence

- P1: read X
Multiprocessor Cache Coherence

- P1: read X
Multiprocessor Cache Coherence

- P1: read X
- P2: read X
Multiprocessor Cache Coherence

- P1: read X
- P2: read X
Multiprocessor Cache Coherence

- P1: read X
- P2: read X
- P2: X++
Multiprocessor Cache Coherence

- P1: read X
- P2: read X
- P2: X++
Multiprocessor Cache Coherence

- P1: read X
- P2: read X
- P2: X++
- P3: read X
Multiprocessor Cache Coherence

- P1: read X
- P2: read X
- P2: X++
- P3: read X
Multiprocessor Cache Coherence
Multiprocessor Cache Coherence
Multiprocessor Cache Coherence

Each cache line has a state (M, E, S, I)
Multiprocessor Cache Coherence

Each cache line has a state (M, E, S, I)
- Processors “snoop” bus to maintain states
Multiprocessor Cache Coherence

Each cache line has a state (M, E, S, I)
- Processors “snoop” bus to maintain states
- Initially \( \rightarrow 'I' \rightarrow \text{Invalid} \)
Multiprocessor Cache Coherence

Each cache line has a state (M, E, S, I)
- Processors “snoop” bus to maintain states
- Initially → ‘I’ → Invalid
- Read one → ‘E’ → exclusive
Each cache line has a state (M, E, S, I)
• Processors “snoop” bus to maintain states
• Initially → ‘I’ → Invalid
• Read one → ‘E’ → exclusive
• Reads → ‘S’ → multiple copies possible
Each cache line has a state (M, E, S, I)
• Processors “snoop” bus to maintain states
• Initially \( \rightarrow 'I' \rightarrow \text{Invalid} \)
• Read one \( \rightarrow 'E' \rightarrow \text{exclusive} \)
• Reads \( \rightarrow 'S' \rightarrow \text{multiple copies possible} \)
• Write \( \rightarrow 'M' \rightarrow \text{single copy} \rightarrow \text{lots of cache coherence traffic} \)
Each cache line has a state (M, E, S, I)
- Processors “snoop” bus to maintain states
- Initially $\rightarrow$ ‘I’ $\rightarrow$ Invalid
- Read one $\rightarrow$ ‘E’ $\rightarrow$ exclusive
- Reads $\rightarrow$ ‘S’ $\rightarrow$ multiple copies possible
- Write $\rightarrow$ ‘M’ $\rightarrow$ single copy $\rightarrow$ lots of cache coherence traffic
Cache Coherence: single-thread

lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
Cache Coherence: single-thread

P1

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try: load lock, R0
  test R0
  bnz try
  store lock, 1
}
Cache Coherence: single-thread

P1

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try: load lock, R0
  test R0
  bnz try
  store lock, 1
}
Cache Coherence: single-thread

```c
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:  load lock, R0
    test R0
  bnz try
  store lock, 1
}
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
Cache Coherence: single-thread

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
Cache Coherence Action Zone

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try: load lock, R0
  test R0
  bnz try
  store lock, 1
}
Cache Coherence Action Zone

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try: load lock, R0
  test R0
  bnz try
  store lock, 1
}

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try: load lock, R0
  test R0
  bnz try
  store lock, 1
}
Cache Coherence Action Zone

P1

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}

P2

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:  load lock, R0
  test R0
  bnz try
  store lock, 1
}
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:  load lock, R0
       test R0
       bnz try
       store lock, 1
}
Cache Coherence Action Zone

P1

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}

P2

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
Cache Coherence Action Zone

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
Cache Coherence Action Zone

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:  load lock, R0
  test R0
  bnz try
  store lock, 1
}

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:  load lock, R0
  test R0
  bnz try
  store lock, 1
}
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try: load lock, R0
  test R0
  bnz try
  store lock, 1
}
Cache Coherence Action Zone

// (straw-person lock impl)  
// Initially, lock == 0 (unheld)  
lock() {  
  try: load lock, R0  
  test R0  
  bnz try  
  store lock, 1  
}

// (straw-person lock impl)  
// Initially, lock == 0 (unheld)  
lock() {  
  try: load lock, R0  
  test R0  
  bnz try  
  store lock, 1  
}
Cache Coherence Action Zone II

P1

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try: load lock, R0
  test R0
  bnz try
  store lock, 1
}

P2

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try: load lock, R0
  test R0
  bnz try
  store lock, 1
}
P1

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}

P2

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}

P3

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
Cache Coherence Action Zone II

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try: load lock, R0
  test R0
  bnz try
  store lock, 1
}

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try: load lock, R0
  test R0
  bnz try
  store lock, 1
}
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
Cache Coherence Action Zone II

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:  load lock, R0
  test R0
  bnz try
  store lock, 1
}

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:  load lock, R0
  test R0
  bnz try
  store lock, 1
}
Cache Coherence Action Zone II

lock():
try: load lock, R0
    test R0
    bnz try
    store lock, 1

// (straw-person lock impl)
// Initially, lock == 0 (unheld)

lock() {
try: load lock, R0
test R0
bnz try
store lock, 1
}
Cache Coherence Action Zone II

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
    test R0
    bnz try
    store lock, 1
}
Cache Coherence Action Zone II

P1

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try: load lock, R0
  test R0
  bnz try
  store lock, 1
}

P2

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try: load lock, R0
  test R0
  bnz try
  store lock, 1
}
Cache Coherence Action Zone II

P1

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}

P2

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}

P3

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
Cache Coherence Action Zone II

lock() {
  try: load lock, R0
  test R0
  bnz try
  store lock, 1
}

Initial lock status:
- P1: lock: 0 (unheld)
- P2: lock: M
- P3: lock: 1

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
Cache Coherence Action Zone II

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
Cache Coherence Action Zone II

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try: load lock, R0
  test R0
  bnz try
  store lock, 1
}
Read-Modify-Write (RMW)

- Implementing locks requires read-modify-write operations
- Required effect is:
  - An atomic and isolated action
    1. read memory location AND
    2. write a new value to the location
  - RMW is very tricky in multi-processors
  - Cache coherence alone doesn’t solve it

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
        test R0
        bnz try
        store lock, 1
}
Essence of HW-supported RMW

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:
    load lock, R0
    test R0
    bnz try
  store lock, 1
}

Make this into a single (atomic hardware instruction)
# HW Support for Read-Modify-Write (RMW)

<table>
<thead>
<tr>
<th>Test &amp; Set</th>
<th>CAS</th>
<th>Exchange, locked increment/decrement,</th>
<th>LLSC: load-linked store-conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most architectures</td>
<td>Many architectures</td>
<td>x86</td>
<td>PPC, Alpha, MIPS</td>
</tr>
</tbody>
</table>

```c
bool cas(addr, old, new) {
    atomic {
        if(*addr == old) {
            *addr = new;
            return true;
        }
        return false;
    }
}
```

```c
int TST(addr) {
    atomic {
        ret = *addr;
        if(!*addr)
            *addr = 1;
        return ret;
    }
    return false;
}
```

```c
int XCHG(addr, val) {
    atomic {
        ret = *addr;
        *addr = val;
        return ret;
    }
}
```

```c
bool LLSC(addr, val) {
    atomic {
        if(*addr == ret) {
            *addr = val;
            return true;
        }
        return false;
    }
}
```
# HW Support for Read-Modify-Write (RMW)

<table>
<thead>
<tr>
<th>Test &amp; Set</th>
<th>CAS</th>
<th>Exchange, locked increment/decrement,</th>
<th>LLSC: load-linked store-conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most architectures</td>
<td>Many architectures</td>
<td>x86</td>
<td>PPC, Alpha, MIPS</td>
</tr>
</tbody>
</table>

```c
int TST(addr) {
    atomic {
        ret = *addr;
        if(!*addr) {
            *addr = 1;
            return ret;
        }
        return false;
    }
}

bool cas(addr, old, new) {
    atomic {
        if(*addr == old) {
            *addr = new;
            return true;
        }
        return false;
    }
}

int XCHG(addr, val) {
    atomic {
        ret = *addr;
        *addr = val;
        return ret;
    }
}

bool LLSC(addr, val) {
    atomic {
        if(*addr == ret) {
            *addr = val;
            return true;
        }
        return false;
    }
}
```

```c
void CAS_lock(lock) {
    while(CAS(&lock, 0, 1) != true);
}
```
HW Support for Read-Modify-Write (RMW)

<table>
<thead>
<tr>
<th>Test &amp; Set</th>
<th>CAS</th>
<th>Exchange, locked increment/decrement,</th>
<th>LLSC: load-linked store-conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most architectures</td>
<td>Many architectures</td>
<td>x86</td>
<td>PPC, Alpha, MIPS</td>
</tr>
</tbody>
</table>

```c
int TST(addr) {
    atomic {
        ret = *addr;
        if(!*addr)
            *addr = 1;
        return ret;
    }
}
```

```c
bool cas(addr, old, new) {
    atomic {
        if(*addr == old) {
            *addr = new;
            return true;
        }
        return false;
    }
}
```

```c
int XCHG(addr, val) {
    atomic {
        ret = *addr;
        *addr = val;
        return ret;
    }
}
```

```c
bool LLSC(addr, val) {
    ret = *addr;
    atomic {
        if(*addr == ret) {
            *addr = val;
            return true;
        }
        return false;
    }
}
```
HW Support for RMW: LL-SC

**LLSC: load-linked store-conditional**

PPC, Alpha, MIPS

```c
bool LLSC(addr, val) {
    ret = *addr;
    atomic {
        if(*addr == ret) {
            *addr = val;
            return true;
        }
    }
    return false;
}
```

- load-linked is a load that is “linked” to a subsequent store-conditional
- Store-conditional only succeeds if value from linked-load is unchanged
LLSC: load-linked store-conditional

- load-linked is a load that is “linked” to a subsequent store-conditional
- Store-conditional only succeeds if value from linked-load is unchanged

```c
bool LLSC(addr, val) {
    ret = *addr;
    atomic {
        if(*addr == ret) {
            *addr = val;
            return true;
        }
        return false;
    }
    return false;
}

void LLSC_lock(lock) {
    while(1) {
        old = load-linked(lock);
        if(old == 0 && store-cond(lock, 1))
            return;
    }
}
```
LLSC Lock Action Zone

P1
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}

lock:
lock: 0

P2
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        if(sc(lock, 1))
          return;
  }
}

lock:
lock:
LLSC Lock Action Zone

P1
lock(lock) {
while(1) {
old = ll(lock);
if(old == 0)
if(sc(lock, 1))
return;
}
}

P2
lock(lock) {
while(1) {
old = ll(lock);
if(old == 0)
if(sc(lock, 1))
if(sc(lock, 1))
return;
}
}
LLSC Lock Action Zone

P1

lock: S[ ] 0

lock: 0

P2

lock: I

P1

lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                return;
    }
}

P2

lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                return;
    }
}
LLSC Lock Action Zone

P1

lock: S[L] 0

lock: 0

P2

lock: 1

lock: 0

lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                return;
    }
}

lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                if(sc(lock, 1))
                    return;
    }
}
LLSC Lock Action Zone

**P1**

```c
lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                return;
    }
}
```

**P2**

```c
lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                if(sc(lock, 1))
                    return;
    }
}
```
LLSC Lock Action Zone II

P1

lock: 0

lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}

P2

lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        if(sc(lock, 1))
          return;
  }
}

Lock Implementation & Cache Coherence
LLSC Lock Action Zone II

```
lock: 0

lock: S[0]

lock: 0

P1
lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                return;
    }
}

P2
lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                if(sc(lock, 1))
                    return;
    }
}
```
LLSC Lock Action Zone II

P1
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}

P2
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        if(sc(lock, 1))
          return;
  }
}
LLSC Lock Action Zone II

P1
lock: S[L] 0
lock: 0

P2
lock: S[L] 0

P1
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}

P2
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
LLSC Lock Action Zone II

P1
lock: S[L] 0

lock: 0

P2
lock: S[L] 0

lock: 0

P1
lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                return;
    }
}

P2
lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                return;
    }
}
LLSC Lock Action Zone II

P1

lock: M 1

lock: 0

lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                return;
    }
}

P2

lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                if(sc(lock, 1))
                    return;
    }
}
LLSC Lock Action Zone II

P1

lock: M 1

lock: 0

P2

lock: 1

lock: 

lock:

Store conditional fails

P1

lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                return;
    }
}

P2

lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                return;
    }
}