CS510 Advanced Topics in Concurrency

Jonathan Walpole
Reasoning About Event Ordering
Making Sense Of It All

Hardware and compilers can reorder our memory accesses, so how can we get our head around what our program will do?

Do we always have to consider the impact of all possible permutations?

How can we write portable code for CPUs with different memory consistency models?
Centralized vs Distributed Systems

What makes a system distributed?

Why does distribution affect event ordering?

What does this have to do with memory access ordering on shared memory multiprocessors?
Time in Distributed Systems

Isn’t ordering just laying events on a timeline?

How can we tell when an event occurs?
- look at a clock?
- what if we do not share the same clock?
- our clocks may drift apart, or be offset
- is there always a delay in communicating?
- does the speed of light really matter?
Sequential Processes

A process is a sequence of events
Assumed to occur in one place
Define a priori total ordering of its events
Event A \textit{happens before} B if it occurs before B in the sequence
Events include sending and receiving messages
The *happens before* Relation (\(\rightarrow\))

If \(a\) and \(b\) are in the same process and \(a\) occurs before \(b\), then \(a \rightarrow b\)

If \(a\) is the sending of a message in one process and \(b\) is the receiving of the same message in another process, then \(a \rightarrow b\)

If \(a \rightarrow b\) and \(b \rightarrow c\), then \(a \rightarrow c\)

An event does not happen before itself!

Happens before defines an irreflexive *partial* ordering on events
Concurrency

Two events a and b are concurrent if a does not happen before b and b does not happen before a.

They are concurrent if they are not ordered by the happens before relation.

This does not mean that they occur at the same physical time!
Space Time Diagrams Help
Happens Before

a -> b if you can go from a to b by following arrows in the diagram

Arrows represent causality (or potential causality)

Concurrent events have no causal relationship

Concurrent events may occur at different physical times!
Logical Clocks

A clock is just a way of assigning a number to an event

Clocks can be implemented logically to totally order events

This ordering may disagree with physical time!

But the partial order of events is converted to a total order (somewhat arbitrarily)
Clock Conditions

If $a \rightarrow b$, then $a$ gets a smaller clock number than $b$

- Does the converse hold?

Clock condition holds if:

C1: If $a$ and $b$ are in the same process, with $a$ before $b$, then $C[a] < C[b]$

C2: If $a$ is a message send, and $b$ is the receipt of the same message, then $C[a] < C[b]$
With Clock Ticks
Which Clock Ticks Are Necessary?

C1 implies there must be a tick between any two events in the same process.

C2 implies that every message must cross a tick line.

Tick lines become coordinates in the space-time diagram.
Stretched to Align Clock Ticks
Implementing a Logical Clock

Each CPU has a counter that increments for each event.

Counter values are carried by messages.

On receipt of a message, the receiver’s clock advances to the larger of the local clock value or the message’s clock value.

CPU ID can be used to break ties between events on different CPUs with the same clock value.
Partial to Total Ordering

The tie breaking completes the partial order to a total order.

The partial order respects causality.

So does the total order, but aside from that it’s somewhat arbitrary.
What is The Total Order Good For?

Now we have a shared clock! (of sorts)

We can solve problems as if we were on a centralized system

But our logical clock does not agree with physical time
Mutual Exclusion Problem

We need to agree who has the lock/resource

Requesting the resource:
- earliest request gets it first
- next request must wait for its release

Releasing the resource:
- send it to “next” requester
- everyone must agree who is next!
Correctness Conditions

A process that has been granted a resource must release it before it can be granted to another process.

Requests must be granted in the order in which they were made (fairness).

If every process granted a resource eventually releases it, every request will eventually be granted.
The Algorithm

Important assumptions:
- in order message transmission
- no lost messages
- resource is initially granted to P0
- each process has its own request queue that has the initial message in it \([P_0,T_0]\)
The Algorithm (cont.)

A process requests by sending a time-stamped request message to all processes, and puts this message in its own queue.

A process receiving a request puts it in its queue and sends a time-stamped acknowledgement.

A process releases by removing its initial request message and sending release messages to all processes.
The Algorithm (cont.)

A process receiving a release message removes the corresponding request message from its queue.

A process is granted the resource when
- its own request message is ordered before any other request message in its queue
- the process has received a message from every other process time-stamped later than this request message.
Summary

This is a kind of atomic broadcast protocol that forces a total ordering on the request and release commands.

Processes can’t take action until they hear from all other processes!

This implies a communication delay, which is a consequence of the total ordering requirement.

This is very much like sequential consistency, where everyone agrees on the ordering!
Linux Kernel Memory Model

Need to write concurrent code that is portable across architectures with different memory consistency models

None of the hardware models are sequentially consistent

But they are also not entirely random!
Ordering Relations in LKMM

Many different sources of ordering:
- program order (not necessarily preserved!)
- coherence order (per variable)
- reads from (write to read)
- from reads (read missed write)
- RCU (discussed later ...)
Cycles

Ordering rules out cycles!

Can’t have $X<Y$, $Y<Z$, and $Z<X$, because that implies $X<X$

If a memory model requires accesses to be ordered, and if a certain outcome can only occur due to a cycle, then the memory model disallows the outcome!
Types of Event in LKMM

Read events:
- `READ_ONCE()`, `smp_load_acquire()`, `rcu_dereference()`

Write events:
- `WRITE_ONCE()`, `smp_store_release()`, `atomic_set()`

Fence events:
- `smp_rmb()`, `rcu_read_lock()`, ...

Allows no ordinary memory accesses!
- `READ_ONCE` and `WRITE_ONCE` disallow compiler optimizations!
Program Order in LKMM

Program order (po)
- X po-before Y, is written as $X \rightarrow_{po} Y$
- if X occurs before Y in the instruction stream

Sub-relation po-loc if both accesses are to the same location

Program order is not necessarily preserved!
Preserved program order (ppo) derived from dependencies, fences etc.
Dependency Relations

Data dependency (data)
- value read affects value written

Address dependency (addr)
- value read affects location accessed

Control dependency (ctrl)
- value read determines execution of other event
Data Dependency

```c
int x, y;

r1 = READ_ONCE(x);
WRITE_ONCE(y, r1 + 5);
```
Address Dependency

```c
int a[20];
int i;

r1 = READ_ONCE(i);
r2 = READ_ONCE(a[r1]);
```
Control Dependency

```c
int x, y;

r1 = READ_ONCE(x);
if (r1)
    WRITE_ONCE(y, 1984);
```
Relation to Program Order

R ->data X implies R ->po X
R ->addr X implies R ->po X
R ->ctrl X implies R ->po X
Reads From (rf) Relation

Links a write to a read when the value read is the value that was written

Has internal and external sub-relations

W ->rf R means R reads from W
W ->rfi R means R reads from W on the same CPU
W ->rfe R means R reads from W on different CPU

Assumes no load tearing (from multiple stores)
Reads From (rf) Relation

```
CPU 0
CPU 1
CPU 2
CPU 3
```

```
WRITE_ONCE(x, 1);
```

```
r1 = READ_ONCE(x);
```

```
x = 0
```

```
x = 1
```

```
rf
```

```
Time
```

```
``
Coherence Order Relation

For each memory location, the stores occur in a total order that all CPUs agree upon.

This order must be consistent with po for access to that location.

\[ W \rightarrow_{co} W' \]

if W comes before W’ in the coherence order.
Coherence Order (co) Relation
Coherency Rules

Write-write coherence:
- If $W \to \text{po-loc } W'$ then $W \to \text{co } W'$

Write-read coherence:
- If $W \to \text{po-loc } R$ then $R$ must read from $W$ or from some other store which comes after $W$ in the coherence order
Coherency Rules (cont.)

Read-write coherence:
- If R -> po-loc W then the store which R reads from must come before W in the coherence order

Read-read coherence:
- If R -> po-loc R’ then either they read from the same store or else the store read by R comes before the store read by R’ in the coherence order

This is essentially sequential consistency *per variable*
Stores to different locations are never ordered by co
From Reads (fr) Relation

R ->fr W if R reads from a store earlier in co than W
(R ->fr W) := (there exists W' with W' ->rf R and W' ->co W)

```c
int x = 0;
P0()
{
    int r1;
    r1 = READ_ONCE(x);
    WRITE_ONCE(x, 2);
}
```
From Reads (fr) Relation

```c
WRITE_ONCE(x, 1);  
r1 = READ_ONCE(x) == 0;
```
Relationship to Time

Only rf implies a global temporal constraint! co and fr can contradict physical time
Operational Model

A distributed system of CPUs and a memory subsystem

Memory subsystem enforces coherence order

CPU sends stores to all other CPUs and memory subsystem

CPU uses po-latest store to satisfy a load, or it is satisfied from memory using co-latest store for that CPU
Operational Model

Fences force CPUs to execute various instructions in program order (depending on type of fence)

They also affect the way the memory subsystem propagates stores

- `smp_wmb()` forces all po-earlier stores on this CPU to propagate before any po-later stores
Axioms of LKML

Sequential consistency per variable
- system must obey the coherence rules

Atomicity
- rmw operations must be atomic (constraints co)

Happens-before
- all cases where po must be preserved (ppo)
- fences between appropriate operations
- data, addr, cntrl dependencies
- operations on the same location (po-loc)
- acquire-release, rfe, ...

Propagates-before (pb)

RCU .... discussed later
Example 1

\[
\text{WRITE\_ONCE(*y,1)} \quad \text{r1 = READ\_ONCE(*x)}
\]

\[
\text{po} \quad \text{smp\_mb()} \quad \text{po}
\]

\[
\text{rf} \quad \text{WRITE\_ONCE(*x,1)} \quad \text{fr}
\]

\[
\text{r2 = READ\_ONCE(*y)}
\]
Example 1 - detail

```
INIT(*x,0)  INIT(*y,0)
co          rf

WRITE_ONCE(*y,1)  r1 = READ_ONCE(*x)
po

WRITE_ONCE(*x,1)  r2 = READ_ONCE(*y)
rf

```
Example 2

\[ \text{WRITE}_{\text{ONCE}}(*y,1) \]

\[ \text{WRITE}_{\text{ONCE}}(*x,1) \]

\[ \text{r}_1 = \text{READ}_{\text{ONCE}}(*x) \]

\[ \text{r}_2 = \text{READ}_{\text{ONCE}}(*y) \]
Example 2 - detail

INIT(*x,0) -> WRITE_ONCE(*y,1) -> r1 = READ_ONCE(*x)

INIT(*y,0) -> WRITE_ONCE(*x,1) -> r2 = READ_ONCE(*y)

co -> rf

c0 -> fr

c0 -> fence

rf -> fence
Automated Checking Tools

Herd tool for automated checking
Herd programs are called litmus tests
Cat language for defining memory models
  - specifies which cycles are prohibited
Example

P0(int *a, int *b)
{
    int r1;
    r1 = READ_ONCE(*a);
    WRITE_ONCE(*b, 1);
}

P1(int *b, int *c)
{
    int r1;
    r1 = READ_ONCE(*b);
    WRITE_ONCE(*c, 1);
}

P2(int *a, int *c)
{
    int r1;
    r1 = READ_ONCE(*c);
    WRITE_ONCE(*a, 1);
}

exists
(0:r1=1 \ 1:r1=1 \ 2:r1=1)
Results

Test C-LB+o-o+o-o+o-o Allowed
States 8
0:r1=0; 1:r1=0; 2:r1=0;
0:r1=0; 1:r1=0; 2:r1=1;
0:r1=0; 1:r1=1; 2:r1=0;
0:r1=0; 1:r1=1; 2:r1=1;
0:r1=1; 1:r1=0; 2:r1=0;
0:r1=1; 1:r1=0; 2:r1=1;
0:r1=1; 1:r1=1; 2:r1=0;
0:r1=1; 1:r1=1; 2:r1=1;
Ok
Witnesses
Positive: 1 Negative: 7
Condition exists (0:r1=1 \ 1:r1=1 \ 2:r1=1)
Observation C-LB+o-o+o-o+o-o Sometimes 1 7
Hash=63d9717e69db4ea05bb2e6840d1d22d4
Adding Dependencies

```c
{
    a=x0;
c=y0;
l:r2=b;
}
P0(int **a)
{
    int *r1;
    r1 = READ_ONCE(*a);
    WRITE_ONCE(*r1, 1);
}
P1(int *b, int **c)
{
    int r1;
    r1 = READ_ONCE(*b);
    if (r1)
        WRITE_ONCE(*c, r2);
}
P2(int **a, int **c)
{
    int *r1;
    r1 = READ_ONCE(*c);
    WRITE_ONCE(*a, r1);
}
exists
(0:r1=b /\ 1:r1=1 /\ 2:r1=b)
```
Results

Test C-LB+ldref-o+o-ctrl-o+o-dep-o Allowed
States 2
0:r1=x0; 1:r1=0; 2:r1=y0;
0:r1=y0; 1:r1=0; 2:r1=y0;
No
Witnesses
Positive: 0 Negative: 2
Condition exists \((0:r1=b \lor 1:r1=1 \lor 2:r1=b)\)
Observation C-LB+ldref-o+o-ctrl-o+o-dep-o Never 0 2
Hash=d8909b22b7196d3b91cd78e700c68cc6

But, removing any of the dependencies would allow a cycle again!