18-447 Lecture 25: 
Synchronization

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today
  - be introduced to synchronization concepts
  - see hardware support for synchronization
- Notices
  - Final Exam: Thu, May 11, 2017, 5:30~8:30p.m, DHA302
    If you miss it, you make-up with Spring 2018
  - practice final (hardcopy)
  - HW5 and Lab4, due next week
- Readings
  - P&H Ch2.11, Ch6
  - Synthesis Lecture: Shared-Memory Synchronization, 2013 (advanced optional)
Format of Final Exam

- **Coverage**
  - lectures (L1~L26, except L20), HWs, projects, assigned readings (textbooks and papers)
- **Types of questions**
  - freebies: can you remember the materials
  - probing: did you understand the materials
  - applied: can you apply the materials in original thoughts
- **180 minutes, 180 points**
  - don’t spend 20 minutes on a 5-point question
  - skip questions you can’t do and come back later
  - closed-book, **3** 2-sided, 8½x11-in², hand-written, cribsheets
  - no electronics

*** Use pencil or black/blue ink only

---

One-Way Writer-to-Reader

- **Two threads in a producer-consumer interaction**
  - consumer-thread waiting for result from producer-thread in shared memory variable **D**ata
  - producer-thread use another shared-memory variable **R**eady to indicate readiness (R=0 initially)

(This lecture use upper-case for shared-mem **V**ariables)

```plaintext
//thread-1
......
compute into D
R=1
......
//thread-2
......
while(R!=1);
consume D
```

- **Straightforward if SC; WC needs memory fences to order operations on R and D**

Generalizable to management of data queues
Data Races

- E.g., two threads increment a shared-memory variable \( v \) initially 0 (assume SC)

```plaintext
//thread-1
//thread-2
v = v
v = v
v = v + 1
v = v + 1
```

Each thread both reads and writes \( v \)

- The intention is clear, but what happens depends on what thread-2 does in between thread-1’s read and write to \( v \) (and vice versa)

- Correctness depends on thread-2 not reading or writing \( v \) between thread-1’s read and write (the “critical section”)

Mutual Exclusion: General Strategy

- Goal: allow only either thread-1 or thread-2 to execute their respective critical sections at a time
  No overlapping interleaving of critical sections!

- Idea: use a shared-memory variable \( L \)ock to indicate whether a thread is already in critical section and the other thread should wait

- Building Blocks
  - \textbf{wait}: a way for a thread to check and wait if \( L \) is not set
  - \textbf{acquire}: a way to set \( L \) before a thread enters the critical section
  - \textbf{release}: a way to clear \( L \) when a thread leaves the critical section
Mutual Exclusion: 1st Try

- Assume $L=0$ initially

```c
//thread-1
    while ($L!=0$);
    $L=1$;
    $t=V$
    $t=func_1(t,\ldots)$
    $V=t$
    $L=0$;
```

```c
//thread-2
    while ($L!=0$);
    $L=1$;
    $t=V$
    $t=func_2(t,\ldots)$
    $V=t$
    $L=0$;
```

But wait, we are in the same trouble with data race on $L$

Mutual Exclusion: Dekker’s

- Using 3 shared-memory variables: $C1=1$, $C2=1$, $\text{Turn}=1$ or 2 initially (assumes SC)

```c
//thread-1
    $C1=0$;
    while ($C2==0$) {
        if ($T==2$) {
            $C1=1$
            while ($T==2$);
            $C1=0$
        }
    }
    \ldots critical-section \ldots
    $T=2$;
    $C1=1$;
```

```c
//thread-2
    $C2=0$;
    while ($C1==0$) {
        if ($T==1$) {
            $C2=1$
            while ($T==1$);
            $C2=0$
        }
    }
    \ldots critical-section \ldots
    $T=1$;
    $C2=1$;
```

- Can you figure this out? (hint: $T$ is the tie breaker)
- What about 3-way? N-way?
  
  There has to be an easier way to do this kind of things
Atomic Read-Modify-Write Instruction

- A special class of memory instructions to facilitate implementations of lock synchronizations
- A semantically atomic instruction that
  - reads a memory location
  - performs some simple calculation
  - writes something back to the same memory location

Note: atomic means the memory location cannot be written by anyone else in between the read and the writeback

- E.g., `<swap>(addr,reg):
  temp ← MEM[addr];
  MEM[addr] ← reg;
  reg ← temp;

  <test&set>(addr,reg):
  reg ← MEM[addr];
  if (reg==0)
    MEM[addr] ← 1;

Expensive to implement and to execute

Acquire and Release

- Could rewrite earlier examples directly using `<swap>` or `<test&set>` instead vanilla loads and stores
- Better to develop a portable lock Acquire( ) and Release( ) discipline to hide the details of ISA-dependent atomic RMW

```plaintext
//thread-1                        //thread-2
Acquire(L);                      Acquire(L);

critical
  t=V
  t=func_1(t,V,...)
  V=t
 Release(L);

critical
  t=V
  t=func_2(t,V,...)
  V=t
 Release(L);
```

Note: implicit in `Acquire(L)` is to wait on `L` if not free
Acquire and Release

- Using `<swap>` on shared-memory variable \( L \) initially 0

```c
void Acquire(L) {
    do {
        r=1;
        <swap>(addrL,r);
    } while(r!=0);
}
```

```c
void Release(L) {
    L=0;
}
```

- Using `<test&set>` on shared-memory variable \( L \) initially 0

```c
void Acquire(L) {
    do {
        <test&set>(addrL,r);
    } while(r!=0);
}
```

```c
void Release(L) {
    L=0;
}
```

Many equally powerful variations of atomic RMW instructions can accomplish the same

High Cost of Atomic RMW

- In uniprocessor
  - atomics needed to synchronize time-shared threads
  - quite literal enforcement of an atomic RMW sequence

- In multicore/multiprocessor (with cache coherence)
  - RMW requires a writeable Exclusive cache copy
  - locks the cacheblock from replacement during the RMW sequence
  - very expensive when many processors are simultaneously trying to acquire a lock—a lot of cache misses and cacheblock transfers (to swap “1” with “1”)

- An optimization
  - check lock value using normal loads on read-only Shared copies
  - apply RMW only when success is probable

```c
void Acquire(L) {
    do {
        while(L==1);
        r=1;
        <swap>(addrL,r);
    } while(r);
}
```
Special Load and Store for RMW

- Each core needs special reserved and address registers, and a store-status bit

\[
\text{<load-linked>}(r, addr):
\begin{align*}
  r &= \text{MEM}[addr]; \\
  \text{reserved} &\leftarrow 1; \text{address} \leftarrow \text{addr};
\end{align*}
\]

\[
\text{<store-conditional>}(addr, r):
\begin{align*}
  \text{if } (\text{reserved} \&\& \text{address}==addr) \\
  &\quad \text{M}[\text{addr}] \leftarrow r; \\
  &\quad \text{store-status} \leftarrow 1; \\
  \text{else} \\
  &\quad \text{store-status} \leftarrow 0;
\end{align*}
\]

- Hardware monitors bus requests to address; reservation is lost if another processor acquires Exclusive copy
- If S-cond attempt does succeed, L-linked and S-cond must have been uninterrupted by other stores

Acquire using L-linked and S-cond

- Basic strategy when mimicking an atomic RWM operation: you are not guaranteed atomicity but you know whether you succeeded; so keep retrying until it happens

```c
void Acquire(L) {
  do
    r=1;
    do {
      <load-linked>(t,addrL)
      while (t!=0);
      <store-conditional>(addrL, r);
    } while (store-status==0);
}
```
Transactional Memory

```
//thread-1
TxnBegin();
t = V;
t = func1(t, V,...)
V = t
TxnEnd();

//thread-2
TxnBegin();
t = V;
t = func2(t, V,...)
V = t
TxnEnd();
```

- TxnBegin and TxnEnd demarcate regions of execution that should be atomic (or appear so)
- Acquire(L)/Release(L) say do one at a time
- TxnBegin( )/TxnEnd( ) say “look like” done one at a time
- Optimistic implementations allow transactions to overlap and only fixes things if violations observable

Okay to use transactions less judiciously than locks?

Optimistic Implementation

- Allow multiple transaction executions to overlap
- Detect atomicity violations between transactions
- On violation, one of the conflicting transactions is aborted (i.e., restarting from the beginning)
  - TM writes are speculative until reaching TxnEnd
  - speculative TM writes not observable by others
- Effective when actual violations is unlikely, e.g.,
  - multiple threads sharing a complex data structure
  - cannot decide statically which part of the data structure touched by different threads’ accesses
  - conservative locking adds a cost to every access
  - TM incurs a cost only when data races actually occur

Very tempting to make everything a transaction
Granularity of Transactions

void *sumParallel
(void * _id) {
    long id=(long) _id;
    long i;
    long N=ARRAY_SIZE/P;
    TxnBegin();
    for(i=0;i<N;i++) {
        double v=A[id*N+i];
        if (v>=0) {
            SumPos+=v;
            if (v>0) {
                TxnBegin();
                SumPos+=v;
                TxnEnd();
            }
        } else {
            SumNeg+=v;
            if (v<0) {
                TxnBegin();
                SumNeg+=v;
                TxnEnd();
            }
        }
    }
    TxnEnd();
}

For P=2, compute separate sums of positive and negative elements of A in SumPos and SumNeg.

Better??
Detecting Atomicity Violation

- A transaction tracks its memory ReadSet and WriteSet
- $\text{Txn}_a$ appears atomic respect to $\text{Txn}_b$ if
  - $\text{WriteSet}(\text{Txn}_a) \cap (\text{WriteSet}(\text{Txn}_b) \cup \text{ReadSet}(\text{Txn}_b)) = \emptyset$
  - $\text{ReadSet}(\text{Txn}_a) \cap \text{WriteSet}(\text{Txn}_b) = \emptyset$

- Lazy Detection
  - broadcast ReadSet and WriteSet to all other transactions only at $\text{TxnEnd}$
  - waste time on transactions that fail early on

- Eager Detection
  - check for violations on the fly by monitoring other transactions’ reads and writes
  - require more frequent communications

An Oversimplified Sketch of HW-based TM using Cache Coherence

- Add ReadSet and WriteSet status bits to identify those cachelines read or written since $\text{TxnBegin}$
- Speculative TM writes
  - issue $\text{BusRdOwn/Invalidate}$ if starting in $I$ or $S$
  - issue $\text{BusWr(old value)}$ on first write to $M$ cache block
  - on abort, silently invalidate WriteSet cachelines
  - on reaching $\text{TxnEnd}$, clear ReadSet/WriteSet bits

  For simplicity, assuming ReadSet/WriteSet cachelines not displaced due to conflict or capacity

- Eager Detection
  - snoop for $\text{BusRd}$, $\text{BusRdOwn}$, and $\text{Invalidation}$
  - downgrades $M\rightarrow S$, $M\rightarrow I$, $S\rightarrow I$ to the ReadSet/WriteSet cachelines are indicative of atomicity violation

Which transaction to abort is a separate decision
Barrier Synchronization

```c
at the end of L22's SumParallel()
remain=100;
do {
    pthread_barrier_wait(&barrier);
    half=(remain+1)/2;
    if (id<(remain/2))
        psum[id]=psum[id]+psum[id+half];
    remain=half;
} while (remain>1);
```

(Blocking) Barriers

- Ensure a group of threads have all reached an agreed upon point
  - threads that arrive early have to wait
  - all are released when the last thread enters
- Barrier for large systems are expensive, often supported by dedicated HW mechanisms
- Easy to build in small shared-memory systems, e.g., assume shared barrier variable \( B=WAIT\_FOR\_N \) initially

```c
[[ if (B==WAIT\_FOR\_N)
    B=1;
  else
    B=B+1; ]]_\text{atomic}_;
while (B!=WAIT\_FOR\_N);
```
Nonblocking Barriers

- Split into separate primitives for entering and exiting
  - `enterBarrier()` is non-blocking and only registers the fact a thread has reached the barrier point
    
    ```
    [[ if ... B=B+1; ]] //atomically
    ```
  - `exitBarrier()` blocks until the barrier is complete
    
    ```
    while (B!=WAIT_FOR_N);
    ```
- If a thread is ready to enter barrier but has other independent work, call `enterBarrier()` and move on
- Call `exitBarrier()` only when the thread runs out of work that doesn’t depend on the barrier

What if a thread makes a mistake and enters twice?

Now for something a little different
The firing squad problem

- A chain of FSMs synchronized by a common clock
  - FSMs can exchange 1 fixed-sized message with its direct left and right neighbors per cycle
  - the chain is of unknown length $N$ and can be arbitrarily long (for simplicity sake assume $N$ is a 2-power)
  - FSMs know whether they are on the ends and if so which end

- Design the FSMs so they assert fire! on the same cycle a finite delay after start is asserted.

Some attempts

- An example of a basic behavior
  - the interior FSMs on each cycle passes the message-from-left to right and message-from-right to left
  - the right-end FSM reflects message-from-left back to left after 1 cycle
  - So what good is this? The left-end FSM can send a token message to its right and it will be returned after $2N$ cycles

- Try to figure out how long the chain is? Don’t bother
  - FSMs must work for all length $N$, and $N$ could be arbitrarily large
  - cannot count or represent an arbitrarily large $N$ in a finite amount of state

Also forget about assigning ranked ordered IDs to FSMs
More Attempts

- What about finding the mid-point? (still assuming N is a 2-power)
  - right-end FSM of the left-half and the left-end FSM of the right half need to figure out who themselves are
- If they become aware at the same time
  - a problem for N is reduced to 2 concurrent problems for N/2
  - and recurse; eventually N=1. *This we can solve!*
- How about having a fast-token and a slow-token?
  - fast-token message passed same as before
  - slow-token message held and send 2 cycles later

Fire!

- Below edges labeled with arrival time of fast token
- Below edges labeled with arrival time of slow token
- Node N/2 receives slow and fast tokens simultaneously