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, Thursday, 5/10, 1pm~4pm
    If you miss it, you make-up with Spring 2019
  – HW5 and Lab4, due next week

• Readings
  – P&H Ch2.11, Ch6
  – Synthesis Lecture: Shared-Memory Synchronization, 2013 (advanced optional)
Final Exam

- Covers lectures (L1~L26, except L20), HW, projects, assigned readings (from textbooks and papers)
- Types of questions
  - freebies: remember the materials
  - probing: understand the materials
  - applied: apply the materials in original interpretation
- **180 minutes, 180 points**
  - point values calibrated to time needed
  - closed-book, 3 8½x11-in² hand-written cribsheets
  - no electronics
  - use pencil or black/blue ink only
A simple example: producer-consumer

- Consumer waiting for result from producer in shared-memory variable **Data**
- Producer uses another shared-memory variable **Ready** to indicate readiness (**R**=0 initially)
  
  (upper-case for shared-mem **Variables**)

**producer:**

```
.....
compute into D
 R=1
.....
```

**consumer:**

```
.....
while(R!=1);
    consume D
.....
```

- Straightforward if SC; if WC, need memory fences to order operations on **R** and **D**
Data Races

• E.g., threads $T_1$ and $T_2$ increment a shared-memory variable $V$ initially 0 (assume SC)

Both threads both read and write $V$

• What happens depends on what $T_2$ does in between $T_1$’s read and write to $V$ (and vice versa)

• Correctness depends on $T_2$ not reading or writing $V$ between $T_1$’s read and write (“critical section”)

$T_1$:  
\[
\begin{align*}
t &= V \\
t &= t + 1 \\
V &= t
\end{align*}
\]

$T_2$:  
\[
\begin{align*}
t &= V \\
t &= t + 1 \\
V &= t
\end{align*}
\]
Mutual Exclusion: General Strategy

- **Goal:** allow only either T1 or T2 to execute their respective critical sections at one time
  
  *No overlapping of critical sections!*

- **Idea:** use a shared-memory variable Lock to indicate whether a thread is already in critical section and the other thread should wait

- **Conceptual Primitives:**
  - **wait-on:** to check and block if Lock is already set
  - **acquire:** to set Lock before a thread enters critical section
  - **release:** to clear Lock when a thread leaves critical section
Mutual Exclusion: 1\textsuperscript{st} Try

- Assume $L=0$ initially

\begin{align*}
\text{T1:} & \quad \text{while}(L!=0); \\
& \quad L=1; \\
& \quad t=V \\
& \quad t=\text{func}_1(t,...) \\
& \quad V=t \\
& \quad L=0;
\end{align*}

\begin{align*}
\text{T2:} & \quad \text{while}(L!=0); \\
& \quad L=1; \\
& \quad t=V \\
& \quad t=\text{func}_2(t,...) \\
& \quad V=t \\
& \quad L=0;
\end{align*}

But wait, same problem with data race on $L$
Mutual Exclusion: Dekker’s

• Using 3 shared-memory variables: **Clear1**=1, **Clear2**=1, **Turn**=1 or 2 initially (assumes SC)

```plaintext
C1=0;
while(C2==0)
  if (T==2) {
    C1=1;
    while(T==2);
    C1=0;
  }
{... Critical Section ...}
T=2;
C1=1;
```

```plaintext
C2=0;
while(C1==0)
  if (T==1) {
    C2=1;
    while(T==1);
    C2=0;
  }
{... Critical Section ...}
T=1;
C2=1;
```

• Can you decipher this? Extend to 3-way?

Need an easier, more general solution

(hint: **T** is the tie breaker)
Atomic Read-Modify-Write Instruction

• Special class of memory instructions to facilitate implementations of lock synchronizations
• Semantically atomic instruction that
  – reads a memory location
  – performs some simple calculation
  – writes something back to the same location

  *HW guarantees no intervening read/write by others*

E.g.,  

\[ \text{<swap>(addr,reg):} \]
\[ \text{temp} \leftarrow \text{MEM[addr]}; \]
\[ \text{MEM[addr]} \leftarrow \text{reg}; \]
\[ \text{reg} \leftarrow \text{temp}; \]

\[ \text{<test&set>(addr,reg):} \]
\[ \text{reg} \leftarrow \text{MEM[addr]}; \]
\[ \text{if (reg==0) } \]
\[ \text{MEM[addr]} \leftarrow 1; \]

Expensive to implement and to execute
Acquire and Release

• Could rewrite earlier examples directly using `<swap>` or `<test&set>` instead loads and stores
• Better to hide ISA-dependence behind portable `Acquire()` and `Release()` routines

**T1:**

```
Acquire(L);
\[
t = v \\
t = \text{func}_1(t, V, ...) \\
V = t
\]
Release(L);
```

**T2:**

```
Acquire(L);
\[
t = v \\
t = \text{func}_2(t, V, ...) \\
V = t
\]
Release(L);
```

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

- Using `<swap>`, \( L \) initially 0

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

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

- Using `<test&set>`, \( L \) initially 0

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

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

Many equally powerful variations of atomic RMW insts can accomplish the same
High Cost of Atomic RMW Instructions

- Literal enforcement of atomicity very early on
- In CC shared-memory multiproc/multicores
  - RMW requires a writeable M/E cache copy
  - lock cacheblock from replacement during RMW
  - expensive when lock contended by many concurrent acquires—a lot of cache misses and cacheblock transfers, just to swap “1” with “1”

- Optimization
  - check lock value using normal load on read-only S copy
  - attempt RMW only when success is probable

```
do {
    if (!L) {
        reg=1;
        <swap>(L,reg);
    }
} while (reg!=0);
```
RMW without Atomic Instructions

- Add per-thread architectural state: `reserved`, `address` and `status`

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

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

- \text{<ld-linked>} requests S-copy

- HW clears `reserved` if S-copy lost due to CC (i.e., store or \text{<st-cond>} at another thread)

- If `reserved` stays valid until \text{<st-cond>}, request M-copy and update; can be no other intervening stores to \text{addr} in between!!
Resolving Data Race without Lock

- E.g., two threads $T_1$ and $T_2$ increment a shared-memory variable $V$ initially 0 (assume SC)

$T_1$:  
\[
\text{do } \{ \text{<ld-linked>}(t,V) \text{ } t=t+1 \text{<st-cond>}(V,t) \text{ while(status==0)} \}
\]

$T_2$:  
\[
\text{do } \{ \text{<ld-linked>}(t,V) \text{ } t=t+1 \text{<st-cond>}(V,t) \text{ while(status==0)} \}
\]

- Atomicity not guaranteed, but . . . .
- You know if you succeeded; no effect if you don’t
  Just try and try again until you succeed
Transaction Memory

**T1:**

```plaintext
TxnBegin();
t=V
t=func1(t, V, ...)
V=t
TxnEnd();
```

**T2:**

```plaintext
TxnBegin();
t=V
t=func2(t, V, ...)
V=t
TxnEnd();
```

- **Acquire(L)/Release(L)** say do one at a time
- **TxnBegin()/TxnEnd()** say “look like” done one at a time

Implementation can allow transactions to overlap and only fixes things if violations observable
Optimistic Implementation

• Allow multiple transaction executions to overlap
• Detect atomicity violations between transactions
• On violation, one of the conflicting transactions is aborted (i.e., restarted from the beginning)
  – TM writes are speculative until reaching `TxnEnd`
  – speculative TM writes not observable by others
• Effective when actual violation 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 occur
Why not transaction’ize everything?

```c
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;
        else
            SumNeg+=v;
    }
    TxnEnd();
}
```

Compute separate sums of positive and negative elements of A in SumPos and SumNeg

Better??
Overhead vs Likelihood of Succeeding

```c
void *sumParallel
    (void * _id) {
    long id=(long) _id;
    long i;
    long N=ARRAY_SIZE/P;
    double psumPos=0;
    double psumNeg=0;

    for(i=0;i<N;i++) {
        double v=A[id*N+i];
        if (v>=0)
            psumPos+=v;
        else
            psumNeg+=v;
    }
    TxnBegin();
    if(psumPos) SumPos+=psumPos;
    if(psumNeg) SumNeg+=psumNeg;
    TxnEnd();
}
```

versus

```c
if(psumPos) {
    Acquire(Lpos);
    SumPos+=psumPos;
    Release(Lpos);
}
if(psumNeg) {
    Acquire(Lneg);
    SumNeg+=psumNeg;
    Release(Lneg);
}
if(psumPos||psumNeg) {
    Acquire(L);
    SumPos+=psumPos;
    SumNeg+=psumNeg;
    Release(L);
}
```

local non-shared
Detecting Atomicity Violation

- A transaction tracks mem \( \text{RdSet} \) and \( \text{WrSet} \)
- \( \text{Txn}_a \) appears atomic respect to \( \text{Txn}_b \) if
  - \( \text{WrSet}(\text{Txn}_a) \cap (\text{WrSet}(\text{Txn}_b) \cup \text{RdSet}(\text{Txn}_b)) = \emptyset \)
  - \( \text{RdSet}(\text{Txn}_a) \cap \text{WrSet}(\text{Txn}_b) = \emptyset \)
- Lazy Detection
  - broadcast \( \text{RdSet} \) and \( \text{WrSet} \) to other txns at \( \text{TxnEnd} \)
  - waste time on txns that failed early on
- Eager Detection
  - check violations on-the-fly by monitoring other txns’ reads and writes
  - require frequent communications
Oversimplified HW-based TM using CC

• Add **RdSet** and **WrSet** status bits to identify cachelines accessed since **TxnBegin**

• Speculative TM writes
  – issue **BusRdOwn/Invalidate** if starting in **I** or **S**
  – issue **BusWr** (old value) on first write to **M** block
  – on abort, silently invalidate **WrSet** cachelines
  – on reaching **TxnEnd**, clear **RdSet/WrSet** bits

  Assume **RdSet/WrSet** cachelines are never displaced

• Eager Detection
  – snoop for **BusRd**, **BusRdOwn**, and **Invalidation**
  – **M → S**, **M → I** or **S → I** downgrades to **RdSet/WrSet** indicative of atomicity violation

Which transaction to abort?
Barrier Synchronization

```c
// at the end of L20's sumParallel()
remain=p;
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

• Can build from shared memory on small systems
  e.g.,

  Acquire($L_B$)
  if ($B==\text{WAIT\_FOR\_N}$) $B=1$;
  else $B=B+1$;
  Release($L_B$)
  while ($B!=\text{WAIT\_FOR\_N}$);

• Barrier on large systems are expensive, often supported/assisted by dedicated HW
Nonblocking Barriers

• Separate primitives for enter and exit
  – \texttt{enterBar()} is non-blocking and only records that a thread has reached the barrier

\begin{verbatim}
Acquire(L_B)
if (B==WAIT_FOR_N) B=1;
else B=B+1;
Release(L_B)
\end{verbatim}

  – \texttt{exitBar()} blocks until the barrier is complete

\begin{verbatim}
while (B!NEWAIT_FOR_N);
\end{verbatim}

• A thread
  – calls \texttt{enterBar()} then go on to independent work
  – calls \texttt{exitBar()} only when no more work that doesn’t depend on the barrier