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, Tuesday, 5/7, 1pm~4pm
    If you miss it, you make-up with Spring 2020
  – 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 L4,L20,L26), HW, projects, assigned readings (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)

$T_1$:  
- $t = V$
- $t = t + 1$
- $V = t$

$T_2$:  
- $t = V$
- $t = t + 1$
- $V = t$

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”)

18-447-S19-L25-S5, James C. Hoe, CMU/ECE/CALEM, ©2019
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 sect
  – **release:** to clear Lock when a thread leaves critical sect
Mutual Exclusion: 1\textsuperscript{st} Try

• Assume \( L=0 \) initially

T1:

\[
\text{while}(L != 0); \\
L=1; \\
t=V \\
t=\text{func}_1(t,\ldots) \\
V=t \\
L=0;
\]

T2:

\[
\text{while}(L != 0); \\
L=1; \\
t=V \\
t=\text{func}_2(t,\ldots) \\
V=t \\
L=0;
\]

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

- Using 3 shared-memory variables: \( \text{Clear}_1 = 1, \) \( \text{Clear}_2 = 1, \) \( \text{Turn} = 1 \) or 2 initially (assumes SC)

\[
\begin{align*}
\text{C}_1 &= 0; \\
\text{while} (\text{C}_2 == 0) &
\begin{align*}
&\text{if} (\text{T} == 2) \\
&\text{C}_1 = 1; \\
&\text{while} (\text{T} == 2); \\
&\text{C}_1 = 0;
\end{align*}
\end{align*}
\]
\[
\{\ldots \text{Critical Section} \ldots \}
\]
\[
\text{T} = 2; \\
\text{C}_1 = 1;
\]

\[
\begin{align*}
\text{C}_2 &= 0; \\
\text{while} (\text{C}_1 == 0) &
\begin{align*}
&\text{if} (\text{T} == 1) \\
&\text{C}_2 = 1; \\
&\text{while} (\text{T} == 1); \\
&\text{C}_2 = 0;
\end{align*}
\end{align*}
\]
\[
\{\ldots \text{Critical Section} \ldots \}
\]
\[
\text{T} = 1; \\
\text{C}_2 = 1;
\]

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

Need an easier, more general solution
Atomic Read-Modify-Write Instruction

• Special class of memory instructions to facilitate implementations of lock synchronizations
• All effects "atomically" executed
  – reads a memory location
  – performs some simple calculation
  – writes something back to the same location

\[ \text{HW guarantees no intervening read/write by others} \]

E.g., \(\text{<swap>(addr,reg):}\)
  \[
  \begin{align*}
  \text{temp} & \leftarrow \text{MEM}[\text{addr}]; \\
  \text{MEM}[\text{addr}] & \leftarrow \text{reg}; \\
  \text{reg} & \leftarrow \text{temp};
  \end{align*}
\]

E.g., \(\text{<test&set>(addr,reg):}\)
  \[
  \begin{align*}
  \text{reg} & \leftarrow \text{MEM}[\text{addr}]; \\
  \text{if (reg==0)} \\
  \text{MEM[addr]} & \leftarrow 1;
  \end{align*}
\]

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);
\begin{align*}
  t &= V \\
  t &= \text{func}_1(t, V, \ldots) \\
  V &= t \\
\end{align*}
Release(L);
\end{equation}
```

T2:
```
Acquire(L);
\begin{align*}
  t &= V \\
  t &= \text{func}_2(t, V, \ldots) \\
  V &= t \\
\end{align*}
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

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

- Add per-thread architectural state: \textit{reserved}, \textit{address} and \textit{status}

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

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

- \textit{<ld-linked>} requests \textit{S}-copy
- HW clears \textit{reserved} if \textit{S}-copy lost due to CC (i.e., store or \textit{<st-cond>} at another thread)
- If \textit{reserved} stays valid until \textit{<st-cond>}, request \textit{M}-copy and update; can be no other intervening stores to \textit{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)

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

$T_2$:
```
    do {
        <ld-linked>$(t, V)$
        t = t + 1
        <st-cond>$(V, t)$
        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
Transactional Memory

- **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 \texttt{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 \( \text{SumPos} \) and \( \text{SumNeg} \)

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();
}
```

```c
if(psumPos) {
    Acquire(L_pos);
    SumPos+=psumPos;
    Release(L_pos);
}
if(psumNeg) {
    Acquire(L_neg);
    SumNeg+=psumNeg;
    Release(L_neg);
}
```

versus

```c
if(psumPos||psumNeg) {
    Acquire(L);
    SumPos+=psumPos;
    SumNeg+=psumNeg;
    Release(L);
}
```
Detecting Atomicity Violation

- A transaction tracks mem **RdSet** and **WrSet**
- **Txn\textsubscript{a}** appears atomic respect to **Txn\textsubscript{b}** if
  - \( \text{WrSet}(\text{Txn}\textsubscript{a}) \cap (\text{WrSet}(\text{Txn}\textsubscript{b}) \cup \text{RdSet}(\text{Txn}\textsubscript{b})) = \emptyset \)
  - \( \text{RdSet}(\text{Txn}\textsubscript{a}) \cap \text{WrSet}(\text{Txn}\textsubscript{b}) = \emptyset \)

- Lazy Detection
  - broadcast **RdSet** and **WrSet** to other txns at **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 cacheblocks 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** cacheblocks
  – on reaching **TxnEnd**, clear **RdSet/WrSet** bits

Assume **RdSet/WrSet** cacheblocks 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?
// 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==WAIT_FOR_N) B=1;
  else B=B+1;
  Release(L_B)
  while (B!=WAIT_FOR_N);
  ```

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

• Separate primitives for enter and exit
  – `enterBar()` is non-blocking and only records that a thread has reached the barrier
    ```
    Acquire(L_B)
    if (B==WAIT_FOR_N) B=1;
    else B=B+1;
    Release(L_B)
    ```
  – `exitBar()` blocks until the barrier is complete
    ```
    while (B!=WAIT_FOR_N);
    ```

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