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
  – HW5, due Friday 4/26 midnight
  – Lab 4, due this week
  – Final Exam, May 3 Fri, 1:00am-3:30pm

• Readings
  – P&H Ch2.11, Ch6
  – Synthesis Lecture: Shared-Memory Synchronization, 2013 (advanced optional)
This is not 18-447
What is 18-447

• **Lab 1~3**: knowledge and skill
  – anyone with a wrench can take apart a car
  – Google Lens can tell you what each part is
  – trained person can put back a working car

• **Lab 4**: analyze and optimize
  – what design decisions make for a car that is fast vs good mileage?
  – how to decide how fast or efficient to make it?

• **Think, Ask, Invent**: what is the “right” future for personal transport?
Computer Architecture is Engineering

• An applied discipline of finding and optimizing solutions under the joint constraints of demand, technology, economics, and ethics
• Thus, instances of what we practice evolve continuously
• Need to learn the principles that govern how to develop solutions to meet constraints
• Don’t memorize instances; understand why it is that way
What exponential really looks like

You just have to stand far back enough to see it
Returning to normally scheduled programming
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:**

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

**consumer:**

```plaintext
......
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”)

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

But now have same problem with data race on $L$

### $T_1$:

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

### $T_2$:

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

wait

acquire

release
Mutual Exclusion: Dekker’s

• Using 3 shared-memory variables: $C_1=1$, $C_2=1$, $T\text{urn}=1$ or 2 initially (assumes SC)

\[
\begin{align*}
C_1 &= 0; \\
\text{while}(C_2==0) &\quad \text{if } (T==2) \{ \\
&\quad \quad C_1=1; \\
&\quad \quad \text{while}(T==2); \\
&\quad \quad C_1=0; \\
&\quad \}; \\
&\quad \{ \ldots \text{Critical Section} \ldots \} \\
T &= 2; \\
C_1 &= 1; \\
\end{align*}
\]

\[
\begin{align*}
C_2 &= 0; \\
\text{while}(C_1==0) &\quad \text{if } (T==1) \{ \\
&\quad \quad C_2=1; \\
&\quad \quad \text{while}(T==1); \\
&\quad \quad C_2=0; \\
&\quad \}; \\
&\quad \{ \ldots \text{Critical Section} \ldots \} \\
T &= 1; \\
C_2 &= 1; \\
\end{align*}
\]

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

Need an easier, more general solution
Aside: what happens in Dekker’s w/o T

- Using shared-memory variables: \( \text{Clear1}=1, \text{Clear2}=1 \) initially (assumes SC)

\[
\begin{align*}
\text{\texttt{C1}} &= 0; \\
\text{while}(\text{\texttt{C2}}==0) \{ \\
    &\quad \text{\texttt{C1}} = 1; \\
    &\quad \text{some delay;} \\
    &\quad \text{\texttt{C1}} = 0; \\
\} \\
&\{\ldots \text{Critical Section} \ldots \} \\
\text{\texttt{C1}} &= 1;
\end{align*}
\]

\[
\begin{align*}
\text{\texttt{C2}} &= 0; \\
\text{while}(\text{\texttt{C1}}==0) \{ \\
    &\quad \text{\texttt{C2}} = 1; \\
    &\quad \text{some delay;} \\
    &\quad \text{\texttt{C2}} = 0; \\
\} \\
&\{\ldots \text{Critical Section} \ldots \} \\
\text{\texttt{C2}} &= 1;
\end{align*}
\]

- Above is safe—if one side in C.S., the other isn’t
- Either or both loop forever if pathological timing

Liveloop possible
Aside: Dumb it down more

- Using shared-memory variables: $C_{lear1}=1$, $C_{lear2}=1$ initially (assumes SC)

```c
C1=0;
while(C2==0) {
    some delay;
}

{ . . . Critical Section . . . }

C1=1;
```

```c
C2=0;
while(C1==0) {
    some delay;
}

{ . . . Critical Section . . . }

C2=1;
```

- Above is still safe—if one side in C.S., the other isn’t
- Both loop forever if tried at same time

Deadlock possible
Atomic Read-Modify-Write Instruction

• Special class of memory instructions to facilitate implementations of lock synchronizations

• Effects executed “atomically” (i.e. not interleaved by other reads and writes)
  – 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.,

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

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

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

T2:
Acquire(L);
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>`, \( L \) 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 $M$ cache copy
  - lock $M$ 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 possible

```c
do {
    reg=1;
    if (!L) {
        <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 ← 1;
    address ← addr;

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

• \textit{<ld-linked>} requests \textit{S}-copy (if not alrdy \textit{S} or \textit{M})

• HW clears \textit{reserved} if cached 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 (if not already \textit{M}) and update; can be no other
  intervening stores to \textit{address} in between!!
**Acquire()** by ld-linked and st-cond

```c
void Acquire(L) {
  do {
    reg_w=1;
    do {
      <ld-linked>(reg_r,L)
      while (reg_r!=0);
      <st-cond>(L,reg_w);
    } while (status==0);
  } while (status==0);
}
```

If `L` is modified in between by another thread, `<st-cond>` will fail and you know to try again.
Resolving Data Race without Lock

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

<table>
<thead>
<tr>
<th>Context Switch</th>
<th>Okay?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 ):</td>
<td>do {</td>
</tr>
<tr>
<td></td>
<td>&lt;ld-linked&gt;(t, V)</td>
</tr>
<tr>
<td></td>
<td>t=t+1</td>
</tr>
<tr>
<td></td>
<td>&lt;st-cond&gt;(V, t)</td>
</tr>
<tr>
<td></td>
<td>} while(status==0)</td>
</tr>
</tbody>
</table>

| \( T_2 \):     | 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.
// at the end of L20 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., for a simple 1-time-use barrier \((B=0\text{ initially})\)

\[
\text{Acquire}(L_B) \\
B = B + 1; \\
\text{Release}(L_B) \\
\text{while } (B \neq \text{NUM\_THREADS});
\]

• 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)
  B = B + 1;
  Release(L_B)
  ```

  – `exitBar()` blocks until the barrier is complete
  
  ```
  while (B != NUM_THREADS);
  ```

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

For more, go read “Synthesis Lecture: Transactional Memory,” 2nd Ed., 2010
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 Execution Strategy

- 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 large structure/array
  - cannot decide statically which part of structure/array touched by different threads
  - conservative locking adds a cost to every access
  - TM incurs a cost only when data races occur
Detecting Atomicity Violation

• A transaction tracks memory \( \text{RdSet} \) and \( \text{WrSet} \)

• \( \text{Txn}_a \) appears atomic with 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 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?
Why not transaction’ize everything?

Compute separate sums of positive and negative elements of $A$ in $\text{SumPos}$ and $\text{SumNeg}$

Better??
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();
}

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

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