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
  – Lab 4, due this week
  – HW6, due Monday 5/2 noon
  – Final Exam on 5/6

• 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
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)

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

\[
\begin{align*}
T_2: & \quad t = V \\
& \quad t = t + 1 \\
& \quad V = t
\end{align*}
\]

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
  
  \[\text{No overlapping of critical sections!}\]

- **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: 1st Try

- Assume $L=0$ initially

**T1:**

```plaintext
while (L!=0);
L=1;
t=V
t=func_1(t, ...)
V=t
L=0;
```

**T2:**

```plaintext
while (L!=0);
L=1;
t=V
t=func_2(t, ...)
V=t
L=0;
```

But now have 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)

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

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
Aside: what happens in Dekker’s w/o T

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

  ```
  C1=0;
  while(C2==0) {
    C1=1;
    some delay;
    C1=0;
  }
  {... Critical Section ...}
  C1=1;
  
  C2=0;
  while(C1==0) {
    C2=1;
    some delay;
    C2=0;
  }
  {... Critical Section ...}
  C2=1;
  ```

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

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

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

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=func1(t,V,...)
V=t
Release(L);

T2:

Acquire(L);

t=V
t=func2(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 possible

```
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 \leftarrow MEM[addr];
    reserved \leftarrow 1;
    address \leftarrow addr;

<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 (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

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);
    }

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 \textbf{T1} and \textbf{T2} increment a shared-memory variable \textbf{V} initially 0 (assume SC)

\begin{itemize}
  \item Atomicity not guaranteed, but . . . .
  \item You know if you succeeded; no effect if you don’t
\end{itemize}

Just try and try again until you succeed

\begin{align*}
\textbf{T1:} \quad & \text{do} \quad \{ \\
& \quad \langle \text{ld-linked} \rangle (t, V) \\
& \quad t = t + 1 \\
& \quad \langle \text{st-cond} \rangle (V, t) \\
& \quad \text{while} \quad (\text{status} == 0) \\
\}
\end{align*}

\begin{align*}
\textbf{T2:} \quad & \text{do} \quad \{ \\
& \quad \langle \text{ld-linked} \rangle (t, V) \\
& \quad t = t + 1 \\
& \quad \langle \text{st-cond} \rangle (V, t) \\
& \quad \text{while} \quad (\text{status} == 0) \\
\}
\end{align*}
Transactional Memory

\[ T_1: \]
\[
\text{TxnBegin()};
\]
\[
t = V
\]
\[
t = \text{func}_1(t, V, \ldots)
\]
\[
V = t
\]
\[
\text{TxnEnd()};
\]

\[ T_2: \]
\[
\text{TxnBegin()};
\]
\[
t = V
\]
\[
t = \text{func}_2(t, V, \ldots)
\]
\[
V = t
\]
\[
\text{TxnEnd()};
\]

- **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 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 RdSet and 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 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/Invalidation** 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?

```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??

```
void *sumParallel
    (void *__id) {
    long id=(long) __id;
    long i;
    long N=ARRAY_SIZE/p;

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

```c

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

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

• 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

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

- `exitBar()` blocks until the barrier is complete
  \[
  \text{while } (B != \text{NUM\_THREADS});
  \]

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