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
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:  
    ......  
    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)

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

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

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”)

18-447-S22-L25-S8, James C. Hoe, CMU/ECE/CALCM, ©2022
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 **L** is already set
  – **acquire:** to set **L** before a thread enters critical sect
  – **release:** to clear **L** when a thread leaves critical sect
Mutual Exclusion: 1\textsuperscript{st} Try

- Assume $L=0$ initially

\begin{itemize}
  \item \textbf{T1:}
  \begin{align*}
  \text{while}(L\neq 0); \\
  L=1; \\
  t=V \\
  t=func_1(t,\ldots) \\
  V=t \\
  L=0;
  \end{align*}

  \item \textbf{T2:}
  \begin{align*}
  \text{while}(L\neq 0); \\
  L=1; \\
  t=V \\
  t=func_2(t,\ldots) \\
  V=t \\
  L=0;
  \end{align*}
\end{itemize}

But now have same problem with data race on $L$
Mutual Exclusion: Dekker’s

- Using 3 shared-memory variables: \( \text{Clear1} = 1, \) \( \text{Clear2} = 1, \) \( \text{Turn} = 1 \) or 2 initially (assumes SC)

\[
\begin{align*}
\text{C1} &= 0; \\
\text{while}(\text{C2}==0) & \quad \text{if (T==2)} \quad \{ \\
\quad & \quad \text{C1} = 1; \\
\quad & \quad \text{while}(\text{T}==2); \\
\quad & \quad \text{C1} = 0; \\
\} & \quad \{ \ldots \text{Critical Section} \ldots \} \\
\text{T} &= 2; \\
\text{C1} &= 1;
\end{align*}
\]

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

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

Need an easier, more general solution

(hint: \( T \) is the tie breaker)
Aside: what happens in Dekker’s w/o $T$

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

```c
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
Aside: Dumb it down more

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

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

{ ... Critical Section ... }
C1=1;
```

```
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
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.,

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

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

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 possible

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

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

\[
\text{<ld-linked> (reg, addr)}: \\
\quad \text{reg} \leftarrow \text{MEM}[\text{addr}]; \\
\quad \text{reserved} \leftarrow 1; \\
\quad \text{address} \leftarrow \text{addr};
\]

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

- \text{<ld-linked>} requests S-copy (if not alrdy S or M)
- HW clears reserved if cached copy lost due to CC (i.e., store or <st-cond> at another thread)
- If reserved stays valid until <st-cond>, request M-copy (if not already M) and update; can be no other intervening stores to 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);
}
```

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)

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

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

• 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 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 $\text{TxnEnd}$
  – 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/Invalid** 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?

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();
}

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

local non-shared

```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)

  ```
  Acquire(L_B)
  B = B + 1;
  Release(L_B)
  while ($B != \text{NUM\_THREADS}$);
  ```

• 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