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 Friday, 5/7 noon
  – HW5: due Friday, 5/7 10pm
  – Midterm 3: Tuesday, 5/11, 5:30~6:25pm

• Readings
  – P&H Ch2.11, Ch6
  – Synthesis Lecture: Shared-Memory Synchronization, 2013 (advanced optional)
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 **T1** and **T2** increment a shared-memory variable **V** initially 0 (assume SC)

<table>
<thead>
<tr>
<th>T1:</th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{t=}V &amp; \text{t=}V</td>
<td></td>
</tr>
<tr>
<td>\text{t=}t+1 &amp; \text{t=}t+1</td>
<td></td>
</tr>
</tbody>
</table>

Both threads both read and write **V**

- What happens depends on what **T2** does in between **T1**’s read and write to **V** (and vice versa)
- Correctness depends on **T2** not reading or writing **V** between **T1**’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 $\text{Lock}$ 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 $\text{Lock}$ is already set
  - **acquire:** to set $\text{Lock}$ before a thread enters critical section
  - **release:** to clear $\text{Lock}$ when a thread leaves critical section
Mutual Exclusion: 1st Try

- Assume $L=0$ initially

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

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

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)

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
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

*HW guarantees no intervening read/write by others*

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

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

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

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

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

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

- \( <\text{ld-linked}> \) requests S-copy (if not cached S /M)
- HW clears reserved if cached 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 (if not already M) and update; can be no other intervening stores to \text{addr} in between!!
Acquire() by ld-linked and st-cond

void Acquire(L) {
    do
        reg=1;
        do {
            <ld-linked>(reg,L)
            while (reg!=0);
            <st-cond>(L,reg);
        } while (status==0);
    } while (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 **T1** and **T2** increment a shared-memory variable **V** initially 0 (assume SC)

```plaintext
T1:
do {
   <ld-linked>(t,V)
   t=t+1
   <st-cond>(V,t)
}while(status==0)
```

```plaintext
T2:
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

©2021 James C. Hoe, CMU/ECE/CALCM, 18-447-S21-L25-S14
Transactional Memory

\[ T1: \]
\[
\begin{align*}
\text{TxnBegin}() &; \\
t = \text{V} &; \\
t = \text{func}_1(t, V, \ldots) &; \\
V = t &; \\
\text{TxnEnd}() &;
\end{align*}
\]

\[ T2: \]
\[
\begin{align*}
\text{TxnBegin}() &; \\
t = \text{V} &; \\
t = \text{func}_2(t, V, \ldots) &; \\
V = t &; \\
\text{TxnEnd}() &;
\end{align*}
\]

- **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 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?

Compute separate sums of positive and negative elements of $A$ in `SumPos` and `SumNeg`
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(Lpos);
    SumPos+=psumPos;
    Release(Lpos);
}
if(psumNeg) {
    Acquire(Lneg);
    SumNeg+=psumNeg;
    Release(Lneg);
}
```

versus

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

local non-shared
Detecting Atomicity Violation

• A transaction tracks memory $\text{RdSet}$ 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 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?
Barrier Synchronization

```c
// 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)

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

• 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