

CS6290

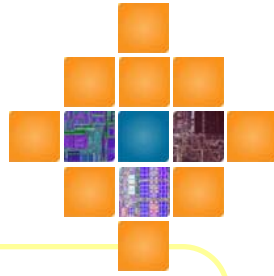
Synchronization

**Georgia
Tech**

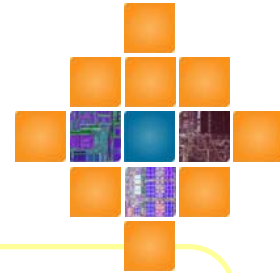


College of
Computing

Synchronization



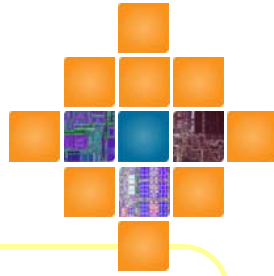
- Shared counter/sum update example
 - Use a mutex variable for mutual exclusion
 - Only one processor can own the mutex
 - Many processors may call lock(), but only one will succeed (others block)
 - The winner updates the shared sum, then calls unlock() to release the mutex
 - Now one of the others gets it, etc.
 - But how do we implement a mutex?
 - As a shared variable (1 – owned, 0 – free)



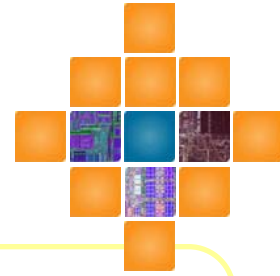
Locking

- Releasing a mutex is easy
 - Just set it to 0
- Acquiring a mutex is not so easy
 - Easy to spin waiting for it to become 0
 - But when it does, others will see it, too
 - Need a way to *atomically* see that the mutex is 0 *and* set it to 1

Atomic Read-Update Instructions



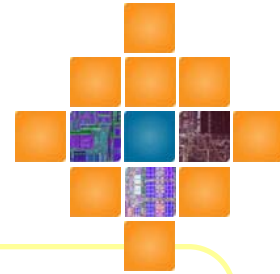
- Atomic exchange instruction
 - E.g., EXCH R1,78(R2) will swap content of register R1 and mem location at address 78+R2
 - To acquire a mutex, 1 in R1 and EXCH
 - Then look at R1 and see whether mutex acquired
 - If R1 is 1, mutex was owned by somebody else and we will need to try again later
 - If R1 is 0, mutex was free and we set it to 1, which means we have acquired the mutex
- Other atomic read-and-update instructions
 - E.g., Test-and-Set



LL & SC Instructions

- Atomic instructions OK, but specialized
 - E.g., SWAP can not atomically inc a counter
- Idea: provide a pair of linked instructions
- A load-linked (LL) instruction
 - Like a normal load, but also remembers the address in a special “link” register
- A store-conditional (SC) instruction
 - Like a normal store, but fails if its address is not the same as that in the link register
 - Returns 1 if successful, 0 on failure
- Writes by other processors snooped
 - If address matches link address, clear link register

Using LL & SC



Swap R4 w/ 0(R1)

Atomic Exchange

```
swap:  mov    R3, R4
       ll    R2, 0(R1)
       sc    R3, 0(R1)
       beqz  R3, swap
       mov   R4, R2
```

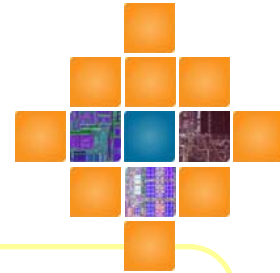
Test if 0(R1) is zero, set to one

Atomic Test&Set

```
t&s:  mov    R3, 1
       ll    R2, 0(R1)
       sc    R3, 0(R1)
       bnez  R2, t&s
       beqz  R3, t&s
```

Atomic Add to Shared Variable

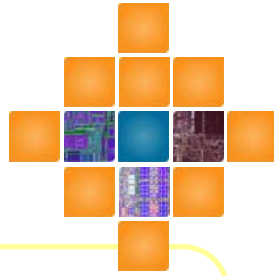
```
upd:  ll    R2, 0(R1)
       add  R3, R2, R4
       sc    R3, 0(R1)
       beqz  R3, upd
```



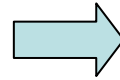
Implementing Locks

- A simple swap (or test-and-set) works
 - But causes a lot of invalidations
 - Every write sends an invalidation
 - Most writes redundant (swap 1 with 1)
- More efficient: test-and-swap
 - Read, do swap only if 0
 - Read of 0 does not guarantee success (not atomic)
 - But if 1 we have little chance of success
 - Write only when good chance we will succeed

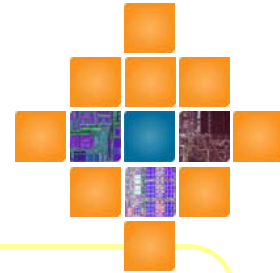
Example: Test and Test and Set



```
try:  mov    R3, #1
      ll     R2, 0(R1)
      sc     R3, 0(R1)
      bnez  R2, try
      beqz  R3, try
```



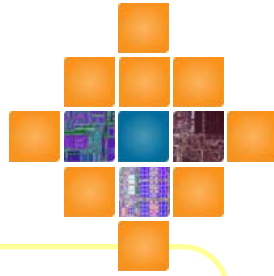
```
try:  mov    R3, #1
      ll     R2, 0(R1)
      bnez  R2, try
      sc     R3, 0(R1)
      beqz  R3, try
```



Barrier Synchronization

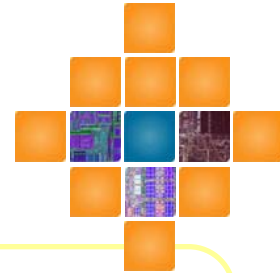
- All must arrive before any can leave
 - Used between different parallel sections
- Uses two shared variables
 - A counter that counts how many have arrived
 - A flag that is set when the last processor arrives

Simple Barrier Synchronization



```
lock(counterlock);
    if(count==0) release=0; /* First resets release */
    count++;                /* Count arrivals */
unlock(counterlock);
if(count==total){         /* All arrived */
    count=0;               /* Reset counter */
    release = 1;          /* Release processes */
}else {                    /* Wait for more to come */
    spin(release==1);     /* Wait for release to be 1 */
}
```

- Problem: not really reusable
 - Two processes: fast and slow
 - Slow arrives first, reads release, sees 0
 - Fast arrives, sets release to 1, goes on to execute other code, comes to barrier again, resets release, starts spinning
 - Slow now reads release again, sees 0 again
 - Now both processors are stuck and will never leave

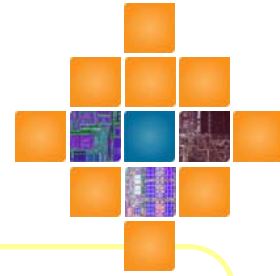


Correct Barrier Synchronization

```
localSense=!localSense;    /* Toggle local sense */
lock(counterlock);
    count++;                /* Count arrivals */
    if(count==total){      /* All arrived */
        count=0;          /* Reset counter */
        release=localSense; /* Release processes */
    }
unlock(counterlock);
spin(release==localSense); /* Wait to be released */
```

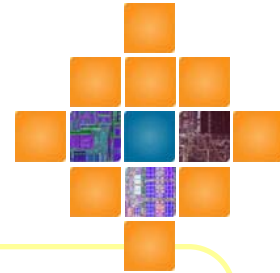
- Release in first barrier acts as reset for second
 - When fast comes back it does not change release, it just waits for it to become 0
 - Slow eventually sees release is 1, stops spinning, does work, comes back, sets release to 0, and both go forward.

init: localSense = 0, release = 0



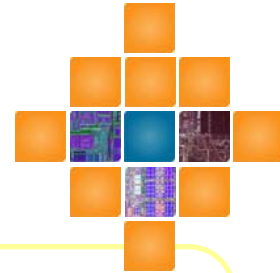
Large-Scale Systems: Barriers

- Barrier with many processors
 - Have to update counter one by one – takes a long time
 - Solution: use a combining tree of barriers
 - Example: using a binary tree
 - Pair up processors, each pair has its own barrier
 - E.g. at level 1 processors 0 and 1 synchronize on one barrier, processors 2 and 3 on another, etc.
 - At next level, pair up pairs
 - Processors 0 and 2 increment a count a level 2, processors 1 and 3 just wait for it to be released
 - At level 3, 0 and 4 increment counter, while 1, 2, 3, 5, 6, and 7 just spin until this level 3 barrier is released
 - At the highest level all processes will spin and a few “representatives” will be counted.
 - Works well because each level fast and few levels
 - Only 2 increments per level, $\log_2(\text{numProc})$ levels
 - For large numProc, $2 * \log_2(\text{numProc})$ still reasonably small



Large-Scale Systems: Locks

- Contention even with test-and-test-and-set
 - Every write goes to many, many spinning procs
 - Making everybody test less often reduces contention for high-contention locks but hurts for low-contention locks
 - Solution: exponential back-off
 - If we have waited for a long time, lock is probably high-contention
 - Every time we check and fail, double the time between checks
 - Fast low-contention locks (checks frequent at first)
 - Scalable high-contention locks (checks infrequent in long waits)
 - Special hardware support



Memory Consistency

- Coherence is about order of accesses to the same address
- Consistency is about order of accesses to different addrs

Program order

<u>Proc 1</u>	<u>Proc 2</u>
ST 1->D	LD F
ST 1->F	LD D

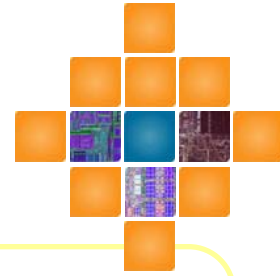


Execution order

<u>Proc 1</u>	<u>Proc 2</u>
ST D	LD D
ST F	LD F

Proc 2 can reorder loads

- Possible outcomes on Proc 2 in program order
 - (F,D) can be (0,0), (0,1), (1,1)
- Execution order can also give (1,0)
 - Exposes instruction reordering to programmer
 - We need something that makes sense intuitively



Memory Consistency

- But first: why would we want to write code like that?
 - If we never actually need consistency, we don't care if it isn't there
- Example: flag synchronization
 - Producer produces data and sets flag
 - Consumer waits for flag to be 1, then reads data

Source Code

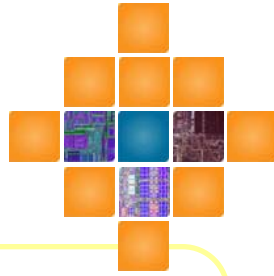
```
Proc 1      Proc 2  
D=Val1;     while(F==0);  
F=1;        Val2=D;
```

Assembler Code

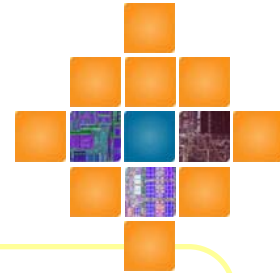
```
Proc 1      Proc 2  
ST D        wait: LD F  
ST F        BEQZ F,wait  
            LD D
```

- Now we can have the following situation
 - Proc 2 has cache miss on F, predicts the branch not taken, reads D
 - Proc 1 writes D, writes F
 - Proc 2 gets F, checks, sees 1, verifies branch is correctly predicted

Sequential Consistency

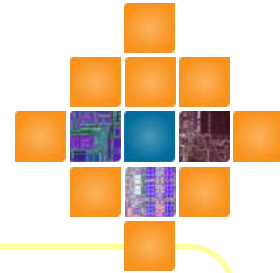


- The result of any execution should be the same as if the accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved
 - The same interleaved order seen by everybody
- Simple implementation
 - A processor issues next access only when its previous access is complete
 - Sloooow!
- A better implementation
 - Processor issues accesses as it sees fit, but detects and fixes potential violations of sequential consistency



Relaxed Consistency Models

- Two kinds of memory accesses
 - Data accesses and synchronization accesses
- Synchronization accesses determine order
 - Data accesses ordered by synchronization
 - Any two of accesses to the same variable in two different processes, such that at least one of the accesses is a write, are always ordered by synchronization operations
- Good performance even in simple implementation
 - Sequential consistency for sync accesses
 - Data accesses can be freely reordered (except around sync accesses)



Relaxed Consistency Models

- Data Races
 - When data accesses to same var in different procs are not ordered by sync
 - Most programs are data-race-free (so relaxed consistency models work well)
- There are many relaxed models
 - Weak Consistency, Processor Consistency, Release Consistency, Lazy Release Consistency
 - All work just fine for data-race-free programs
 - But when there are data races, more relaxed models \Rightarrow weirder program behavior