Synchronization

Synchronization

- How to synchronize processes?
  - Need to protect access to shared data to avoid problems like race conditions
  - Typical example: Updating a shared account balance. Problem below?

Processor 1

```
lw      $t0,balance
lw      $t1,amount
add     $t0,$t0,t1
sw      $t0,balance
```

Processor 2

```
lw      $t2,balance
lw      $t3,amount
sub     $t2,$t2,t3
sw      $t2,balance
```

Critical Sections

- In general:
  - *n* processes all competing to use some shared data
  - Each process has a **critical section** in which shared data is accessed
- Goal
  - Ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section

Processor

```
Entry_section()            // Wait until exclusive access
Critical Section
Remainder_section()        // Release exclusivity
```

Criteria for Critical Sections

- Mutual exclusion
  - Two processors can never be in the critical section at the same time
- Progress
  - If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next.
- Bounded waiting
  - There exists a bound or limit on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

First Attempt - Alternation

Processor 0

```
shared int turn = 0;

while (turn != 0);
Critical Section
turn = 1;
Remainder_section()
```

Processor 1

```
shared int turn = 0;

while (turn != 1);
Critical Section
turn = 0;
Remainder_section()
```

Satisfies mutual exclusion
Does not satisfy progress
Might satisfy bounded wait if processors eventually set the turn variable

Second Attempt – Warning Flags

Processor 0

```
shared int flag[2] = {0,0};

flag[0] = TRUE;
while (flag[1]);
Critical Section
flag[0] = FALSE;
Remainder_section()
```

Processor 1

```
shared int flag[2] = {0,0};

flag[1] = TRUE;
while (flag[0]);
Critical Section
flag[1] = FALSE;
Remainder_section()
```

Satisfies mutual exclusion and progress
P0 only enters critical section if flag[0] && !flag[1]
P1 only enters critical section if flag[1] && !flag[0]

Satisfy bounded wait? Deadlock possible?

Software Solution – Peterson's Algorithm

Processor 0

```
shared int flag[2] = {0,0};
shared int turn;

flag[0] = TRUE;
turn = 1;
while (turn == 1 && flag[1]);
Critical Section
flag[0] = FALSE;
Remainder_section()
```

Processor 1

```
shared int flag[2] = {0,0};
shared int turn;

flag[1] = TRUE;
turn = 0;
while (turn == 0 && flag[0]);
Critical Section
flag[1] = FALSE;
Remainder_section()
```

Combines alternation and warning flags

Polite – we want to enter, but if someone else wants to enter, go ahead

Satisfies mutual exclusion, progress, and bounded wait

Slow (busy wait) and unwieldy to extend to many processors

Hardware Synchronization

- For large scale MPs, synchronization can be a bottleneck
  - Contention adds additional delays, latency is potentially great
- Hardware primitives needed
  - all solutions based on "atomically inspect and update a memory location"
- Higher level synchronization solutions can be build on top of the basic hardware primitives
  - Locks, barriers
  - Generally employed by the system software not the programmer directly

Uninterruptible Instructions to Fetch and Write to Memory

- Atomic exchange: interchange a value in a register for a value in memory
  - Used to build a simple lock
  - To lock, set register to 1 and exchange
    - 0 => synchronization variable is free
    - 1 => synchronization variable is locked by some other processor and unavailable
  - Must be atomic or synchronization may fail
- Test-and-set: tests a value and sets it if the value passes the test
  - Ex. Test for 0 and if so set to 1
- Fetch-and-increment: it returns the value of a memory location and atomically increments it
  - 0 => synchronization variable is free
- All implementations require coherence

A spin lock using the **exchange** primitive

- Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock:

```
          LI    R2,#1       ;load immediate
lock:     EXCH  R2,0(R1)    ;atomic exchange
          BNEZ  R2,lock     ;already locked?
```

- Multiprocessors with cache coherency
  - Want to spin on cache copy to avoid full memory latency
  - Likely to get cache hits for such variables
  - **Problem**: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic
  - **Solution**: start by simply repeatedly reading the variable; when it changes, then try exchange (“test and test & set”):

```
try:      LI    R2,#1       ;load immediate
lock:     LW    R3,0(R1)    ;load var
          BNEZ  R3,lock     ;not free=>spin
          EXCH  R2,0(R1)    ;atomic exchange
          BNEZ  R2,try      ;already locked?
```

Load Linked / Store Conditional

- Challenging to implement a single atomic memory operation
  - Has to do a memory read and a write
  - Complicates coherence since the hardware cannot allow any other operations between the read and the write
- Alternative
  - Have a pair of instructions, a load and a store
  - The second instruction returns a value from which it can deduced if the pair executed as if it were atomic
  - The load is a special load called **load linked** or **load locked** and the special store is called **store conditional**

Load Linked / Store Conditional

- If the contents of the memory location specified by the load linked are changed before the store conditional occurs, then the store conditional fails
- If the processor does a context switch between the two instructions then the store conditional fails
- Store conditional returns 1 if successful and 0 otherwise

Examples with LL/SC

Atomic swap with LL & SC on memory location specified by R1

```
try:    MOV    R3, R4       ; Mov Exchange value
        LL     R2, 0(R1)    ; Load Linked
        SC     R3, 0(R1)    ; Store Conditional
        BEQZ   R3, try      ; Branch store fails
        MOV    R4, R2       ; Put load value in R4
```

If SC fails then R3 = 0 and we branch and try again (e.g. a processor intervened and modified the value in 0(R1)).

Atomic fetch and increment:

```
try:    LL     R2, 0(R1)    ; Load Linked
        DADDUI R3, R2, #1   ; Increment
        SC     R3, 0(R1)    ; Store Conditional
        BEQZ   R3, try      ; Branch store fails
```

Implementing LL/SC

- Link Register
  - Stores the address specified in the LL instruction
- Interrupt or Cache block matching the link register is invalidated
  - Clear the link register
  - Same idea as snooping, listening for a write on the address of the link register
- Some care must be taken; deadlock conditions possible depending upon what is placed between LL and SC (should only be register-register instructions)

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();
if(count==0) release=FALSE;  /* First resets release */
count++;                     /* Count arrivals */
unlock();
if(count==total)             /* All arrived */
{
  count=0;                   /* Reset counter */
  release = TRUE;            /* Release processes */
}
else                         /* Wait for more to come */
{
  while (!release);          /* Wait for release  */
}
```

- Problem: deadlock possible if reused
  - Two processes: fast and slow
  - Slow arrives first, reads release, sees FALSE
  - Fast arrives, sets release to TRUE, goes on to execute other code, comes to barrier again, resets release to FALSE, starts spinning on wait for release
  - Slow now reads release again, sees FALSE again
  - Now both processors are stuck and will never leave

Correct Barrier Synchronization

initially localSense = FALSE, release = FALSE

```
localSense=!localSense;     /* Toggle local sense */
lock();
  count++;                  /* Count arrivals */
  if(count==total){         /* All arrived */
    count=0;                /* Reset counter */
    release=localSense;     /* Release processes */
  }
unlock();
while(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 FALSE
  - Slow eventually sees release is TRUE, stops waiting, does work, comes back, sets release to FALSE, and both go forward.

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, $\log_2$(numProc) levels
      - For large numProc, 2*$\log_2$(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

- When does a processors see another processor's written value?
  - Do different processors see writes at the same time?

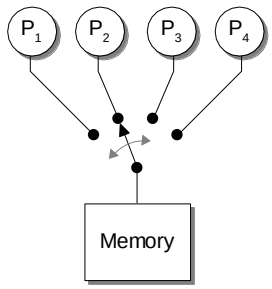
P1:	A = 0;	P2:	B = 0;
	…		…
	A = 1;		B = 1;
L1:	if (B == 0) …	L2:	if (A == 0) …

  - If writes are immediately seen by other processors it will be impossible for both if statements to evaluate as true

  - Suppose write invalidate is delayed then it is possible P1 or P2 have not seen the invalidations before attempting to read the values
    - Should this be allowed?

Sequential Consistency

Lamport (1979): A multiprocessor is **sequentially consistent** if the result of any execution is the same as if the (memory) operations of all processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program



All processors see all loads/stores happening in the same order

Sequential Consistency

- Simple implementation
  - A processor issues next access only when its previous access is complete
  - Under sequential consistency, we cannot place the write in a write buffer and continue
  - Slow
- Better performance
  - Latency hiding techniques
  - Less restrictive memory consistency models

Relaxed Consistency Models

- Synchronized programs
  - A program is synchronized if all access to shared data are ordered by synchronization operations
  - 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
- If variables may be updated without synchronization
  - Data race and outcome is unpredictable
  - Goal is data-race-free
- Good performance even in simple implementation
  - Seems acceptable to programmers
  - Accepted that most programs are synchronized and behaves as if the hardware implemented sequential consistency

Relaxed Consistency Models

- There are many other relaxed models
  - Processor Consistency, Partial Store Order, Release Consistency, Lazy Release Consistency
  - All work just fine for data-race-free programs
  - But when there are data races,
    more relaxed models $\Rightarrow$ weird program behavior