Administrivia

• Please say your name if you answer a question today
• If we don’t have a photo of you yet, stay after class
• If you didn’t get test email, let us know
Program A

int flag1 = 0, flag2 = 0;

void p1 () {
    flag1 = 1;
    if (!flag2) { /* critical section */ }
}

void p2 () {
    flag2 = 1;
    if (!flag1) { /* critical section */ }
}

• Can both critical sections run?
Program B

```c
int data = 0, ready = 0;

void p1 () {
    data = 2000;
    ready = 1;
}

void p2 () {
    while (!ready)
        ;
    use (data);
}

- Can use be called with value 0?
```
Program C

```c
int a = 0, b = 0;

void p1 () { a = 1; }

void p2 () {
    if (a == 1)
        b = 1;
}

void p3 () {
    if (b == 1)
        use (a);
}

• Can use be called with value 0?
```
Correct answers

- Program A: I don’t know
- Program B: I don’t know
- Program C: I don’t know
- Why?
  - It depends on your hardware
  - If it provides *sequential consistency*, then answers all No
  - But not all hardware provides sequential consistency

- [BTW, examples and subsequent analysis from excellent Tech Report by Adve & Gharachorloo]
Sequential Consistency

- **Sequential consistency**: The result of execution is as if all operations were executed in some sequential order, and the operations of each processor occurred in the order specified by the program. [Lamport]

- Boils down to two requirements:
  1. Maintaining *program order* on individual processors
  2. Ensuring *write atomicity*

- Why doesn’t all hardware support sequential consistency?
S.C. thwarts hardware optimizations

- **Write buffers**
  - E.g., read flag\(n\) before flag\((2 - n)\) written through in Program A

- **Overlapping write operations can be reordered**
  - Concurrent writes to different memory modules
  - Coalescing writes to same cache line

- **Non-blocking reads**
  - E.g., speculatively prefetch data in Program B

- **Cache coherence**
  - Write completion only after invalidation/update (Program B)
  - Can’t have overlapping updates (Program C)
S.C. thwarts compiler optimizations

- Code motion
- Caching value in register
  - E.g., ready flag in Program B
- Common subexpression elimination
- Loop blocking
- Software pipelining
Possible optimizations

- "Prefetch" writes
  - Invalidate memory in other CPU’s caches while waiting for previous reads to complete

- Speculatively execute reads (optimistically)
  - If program order violated, roll back state
Relaxed Consistency Models

• Relax program order
  - Relax Write to Read order
    E.g., Re-order read wrt. writes from same proc, breaks A
  - Relax Write to Read and Write to Write order
    E.g., Read own writes before other people
  - Relax Read to Read and Read to Write order

• Relax write atomicity
  - Read others’ writes early

• Relax both
  - Read own writes early (in conjunction with other relaxation)
Weak ordering

- Define two classes of memory operation
  - data
  - synchronization
- System can reorder any operations between sync references
- Easy to implement:
  - Processor keeps counter of outstanding operations
How to classify memory accesses?

- Find variables that *race* under S.C.:
  - Two operations access variable
  - At least one is a write
  - No intervening references (in S.C.)

- **E.g., in Prog B,** *ready* races, *not* data
Release consistency

• 4 types of memory operation:
  - ordinary, nsync, acquire, release

• Preserve the following orderings [RCsc]:
  - acquire $\rightarrow$ all
  - all $\rightarrow$ release
  - \{release, nsync\} $\rightarrow$ \{acquire, nsync\}

• Perfect for data protected by mutexes
Implementing synchronization

- Applications use high-level abstractions
  - Monitors, semaphores, mutexes, condition variables, …
  - These interact with thread/process scheduler
  - E.g., if P1 blocked on a mutex, schedule P2

- But how do these abstractions synchronize?
  - E.g., how to implement a mutex?
  - E.g., how can scheduler protect the run queue?

- Need lower-level synchronization primitives
  - Implementations typically use spinlocks
  - No scheduler…just keep spinning until you get lock
Uniprocessor synchronization

• Can “cheat” to make life easier

• At user-level, can use one kernel thread
  - Context switch on timer interrupts (setitimer)
  - In critical section: Set “do not interrupt” bit
  - If timer interrupt arrives, set “interrupted” bit
  - Manipulate protected low-level data structure
  - Clear DNI bit
  - If interrupted bit set, yield

• In kernel, can do what old UNIX kernels did
UNIX Synchronization 1

- Interface designed before multiprocessors common
- Top half kernel procedures can mask interrupts

```c
int x = splhigh ();
/* ... */
splx (x);
```

- `splhigh` disables all interrupts, but also `splnet`, `splbio`, `splsoftnet`, ...
- **Masking interrupts in hardware can be expensive**
  - Optimistic implementation – set mask flag on `splhigh`, check interrupted flag on `splx`
UNIX Synchronization 2

- Need to relinquish CPU when waiting for events
  - Disk read, network packet arrival, pipe write, signal, etc.
- int tsleep(void *ident, int priority, ...);
  - Switches to another process
  - ident is arbitrary pointer—e.g., buffer address
  - priority is priority at which to run when woken up
  - PCATCH, if ORed into priority, means wake up on signal
  - Returns 0 if awakened, or ERESTART/EINTR on signal
- int wakeup(void *ident);
  - Awakens all processes sleeping on ident
  - Restores SPL to value when they went to sleep
    (so fine to sleep at splhigh)
For MP, need hardware support

- Need atomic read-write or read-modify-write:
  - **Example:** `int test_and_set (int *lockp);`
    - Sets `*lockp = 1` and returns old value

- **Now can implement spinlocks:**
  
  #define lock(lockp) while (test_and_set (lockp))
  #define unlock(lockp) *lockp = 0

- **When more threads than processors, don’t just spin**
  - Wastes CPU when other runnable work exists
    - Especially if thread holding lock doesn’t have a CPU

- **But gratuitous context switch has cost**
  - Good plan: spin for a bit, then yield
Synchronization on i386

- **xchg instruction**, exchanges reg with mem

  _test_and_set:
  
  ```
  movl 8(%esp), %edx
  movl $1, %eax
  xchg %eax, (%edx)
  ret
  ```

- **CPU locks memory system around read and write**
  - I.e., `xchg` always acts like it has lock prefix
  - Prevents other uses of the bus (e.g., DMA)

- **Operates at memory bus speed, not CPU speed**
  - Much slower than cached read/buffered write
Synchronization on alpha

- `ldl_l` – load locked
- `stl_c` – store conditional

```assembly
_test_and_set:
  ldq_l   v0, 0(a0)
bne       v0, 1f
addq     zero, 1, v0
stq_c    v0, 0(a0)
beq      v0, _test_and_set
mb
  addq     zero, zero, v0
1:
  ret      zero, (ra), 1
```