Two Versions of Semaphores

- Semaphores from last time (simplified):
  
  `wait(s):`
  - if (s < 0)
    - block the thread that called wait(s)
  - s = s – 1
  - otherwise
    - continue into CS

  `signal(s):`
  - if (s ≤ 0)
    - wake up one of the waiting threads
  - s = s + 1

- "Classical" version of semaphores:
  
  `wait(s):`
  - if (s ≤ 0)
    - block the thread that called wait(s)
  - s = s – 1
  - otherwise
    - continue into CS

  `signal(s):`
  - if (a thread is waiting)
    - wake up one of the waiting threads
  - s = s + 1

Do both work? What is the difference??

Implementing Semaphores

- Implementing semaphores using busy-waiting:
  
  `wait(s):`
  - while (s ≤ 0)
    - do nothing;
    - s = s – 1

  `signal(s):`
  - s = s + 1

- Evaluation:
  - Doesn’t support queue of blocked threads waiting on the semaphore
  - Waiting threads wastes time busy-waiting (doing nothing useful, wasting CPU time)
  - The code inside wait(s) and signal(s) is a critical section also, and it’s not protected

Implementing Semaphores (cont.)

- Implementing semaphores (not fully) by disabling interrupts:
  
  `wait(s):`
  - disable interrupts
  - while (s ≤ 0)
    - do nothing;
    - s = s – 1
  - enable interrupts

  `signal(s):`
  - disable interrupts
  - while (s ≤ 0)
    - do nothing;
    - s = s + 1
  - enable interrupts

- Evaluation:
  - Doesn’t support queue of blocked threads waiting on the semaphore
  - Waiting threads wastes time busy-waiting (doing nothing useful, wasting CPU time)
  - Doesn’t work on multiprocessors
  - Can interfere with timer, which might be needed by other applications
  - OK for OS to do this, but users aren’t allowed to disable interrupts! (Why not?)

Implementing Semaphores (cont.)

- Implementing semaphores (not fully) using a test&set instruction:
  
  `wait(s):`
  - while (test&set(lk)!=0)
    - do nothing;
    - s = s – 1
  - lk = 0

  `signal(s):`
  - while (test&set(lk)!=0)
    - do nothing;
    - s = s + 1
  - lk = 0

- Operation:
  - Lock “lk” has an initial value of 0
  - If “lk” is free (lk=0), test&set atomically:
    - reads 0, sets value to 1, and returns 0
    - loop test fails, meaning lock is now busy
  - If “lk” is busy (lk=1), test&set atomically:
    - reads 1, sets value to 1, and returns 1
    - loop test is true, so loop continues until someone releases the lock
Implementing Semaphores (cont.)

- Test&set is an example of an atomic read-modify-write (RMW) instruction
  - RMW instructions atomically read a value from memory, modify it, and write the new value to memory
    - Test&set — on most CPUs
    - Exchange — Intel x86 — swaps values between register and memory
    - Compare&swap — Motorola 68xxx — read value, if value matches value in register r1, exchange register r1 and value

- Evaluation:
  - ✓ Can be made to work, even on multiprocessors (although there may be some cache consistency problems)
  - ✗ Doesn’t support queue of blocked threads waiting on the semaphore
  - ✗ Waiting threads wastes time busy-waiting (doing nothing useful, wasting CPU time)

Semaphores in Nachos

- The class Semaphore is defined in threads/synch.h and synch.cc
  - The classes Lock and Condition are also defined, but their member functions are empty (implementation left as exercise)

- Interesting functions:
  - Semaphores:
    - Semaphore::Semaphore() — creates a semaphore with specified name & value
    - Semaphore::P() — semaphore wait
    - Semaphore::V() — semaphore signal
  - Locks:
    - Lock::Acquire()
    - Lock::Release()
  - Condition variables:
    - Condition::Wait()
    - Condition::Signal()

Semaphores in Nachos

```c
void Semaphore::P()
{
    IntStatus oldLevel = interrupt->
    SetLevel(IntOff);  // disable interrupts

    while (value == 0) { // sema not avail
        queue-> // so go to sleep
        Append((void *)currentThread);
        currentThread->Sleep();
    }

    value--; // semaphore available,
    // consume its value

    (void) interrupt-> // re-enable interrupts
    SetLevel(oldLevel);
}
```

```c
void Semaphore::V()
{
    Thread *thread;

    IntStatus oldLevel = interrupt->
    SetLevel(IntOff);

    thread = (Thread *)queue->Remove();
    if (thread != NULL) // make thread ready,
    // consuming the V immediately
    scheduler->ReadyToRun(thread);

    value++;

    (void) interrupt->SetLevel(oldLevel);
}
```