

Two Versions of Semaphores

- Semaphores from last time (simplified):

<u>wait (s):</u>	<u>signal (s):</u>
$s = s - 1$	$s = s + 1$
if ($s < 0$)	if ($s \leq 0$)
block the thread that called wait(s)	wake up one of the waiting threads
otherwise	
continue into CS	

- "Classical" version of semaphores:

<u>wait (s):</u>	<u>signal (s):</u>
if ($s \leq 0$)	if (a thread is waiting)
block the thread that called wait(s)	wake up one of the waiting threads
$s = s - 1$	$s = s + 1$
continue into CS	

- Do both work? What is the difference??

1

Fall 2002, Lecture 12

Implementing Semaphores

- Implementing semaphores using *busy-waiting*:

<u>wait (s):</u>	<u>signal (s):</u>
while ($s \leq 0$)	$s = s + 1$
do nothing;	
$s = s - 1$	

- Evaluation:

- ✗ 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
- ✗ Doesn't support a queue of multiple blocked threads waiting on the semaphore (why is this bad?)

2

Fall 2002, Lecture 12

Implementing Semaphores (cont.)

- Implementing semaphores (not fully) by *disabling interrupts*:

<u>wait (s):</u>	<u>signal (s):</u>
disable interrupts	disable interrupts
while ($s \leq 0$)	$s = s + 1$
do nothing;	
$s = s - 1$	
enable interrupts	enable interrupts

- Evaluation:

- ✓ Protects code inside wait(s) and signal(s)
- ✗ Waiting threads wastes time *busy-waiting*
- ✗ Doesn't support queue of blocked threads waiting on the semaphore
- ✗ Users can't disable interrupts
- ✗ Can interfere with timer, which might be needed by other applications
- ✗ Doesn't work on multiprocessors

3

Fall 2002, Lecture 12

Implementing Semaphores (cont.)

- Implementing semaphores (not fully) using a *test&set instruction*:

<u>wait (s):</u>	<u>signal (s):</u>
while (test&set(lk)!=0)	while (test&set(lk)!=0)
do nothing;	do nothing;
while ($s \leq 0$)	$s = s + 1$
do nothing;	
$s = s - 1$	
lk = 0	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

4

Fall 2002, Lecture 12

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)
 - ✗ Waiting threads wastes time *busy-waiting*
 - ✗ Doesn't support queue of blocked threads waiting on the semaphore

5

Fall 2002, Lecture 12

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()

6

Fall 2002, Lecture 12

Semaphores in Nachos

```
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);
}
```

7

Fall 2002, Lecture 12

Semaphores in Nachos (cont.)

```
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);
}
```

8

Fall 2002, Lecture 12