Inside Blocking Synchronization

Administrative

“I need more space to build Nachos.”
“Ask and ye shall receive.”

Lab 1 grades:
Now available on the course Web.

Lab 2:
Will be due THURSDAY at MIDNIGHT.
actually, 12:01 AM on Friday
(I don’t want you sleeping through MY class.)
Sign up for demos on the Web after Thursday’s class.
Implementing Mutexes: A First Cut

class Lock {
    int held;
    List sleepers;
}

void Lock::Acquire() {
    while (held) {
        sleepers.Append((void*)currentThread);
        currentThread->Sleep();
    }
    held = 1;
}

void Lock::Release() {
    held = 0;
    if (!sleepers->IsEmpty()) /* somebody’s waiting: wake up */
        scheduler->ReadyToRun((Thread*)sleepers->Remove());
}

Mutexes: What Went Wrong

Potential missed wakeup: holder could Release before thread is on sleepers list.

void Lock::Acquire() {
    while (held) {
        sleepers.Append((void*)currentThread);
        currentThread->Sleep();
    }
    held = 1;
}

void Lock::Release() {
    held = 0;
    if (!sleepers->IsEmpty()) /* somebody’s waiting: wake up */
        scheduler->ReadyToRun((Thread*)sleepers->Remove());
}
Example: Await/Awake

Consider a very simple use of sleep/wakeup to implement two new primitives:

- `currentThread->Await()`
  Block the calling thread.

- `Thread::Awake()`
  If the target thread is sleeping in a previous `Await()`, wake it up.
  Else leave it alone.

```cpp
void Thread::Await() {
    ASSERT(this == currentThread);
    ASSERT(!awaiting);
    awaiting = TRUE; /* “I’m sleeping” */
    Sleep(); /* sleep */
}

void Thread::Awake() {
    ASSERT(this != currentThread);
    if (awaiting) /* wakeup */
        scheduler->ReadyToRun(this);
    awaiting = FALSE; /* “you’re awake” */
}
```

These sleep/wakeup races are sometimes referred to as the wake-up waiter problem.
Using Sleep/Wakeup Safely

```cpp
void Thread::Await() {
    disable interrupts;
    awaiting = TRUE; /* "I'm sleeping" */
    Sleep(); /* sleep */
    enable interrupts;
}

void Thread::Awake() {
    disable interrupts;
    if (awaiting) { /* wakeup */
        scheduler->ReadyToRun(this);
        awaiting = FALSE; /* "you're awake" */
    }
    enable interrupts;
}
```

Will this work on a multiprocessor?

**Nachos Sleep requires** disabling interrupts, in part to avoid missed wakeup. How can it be OK to sleep with interrupts disabled?

Sleep and Yield in Nachos

Disable interrupts on the call to **Sleep** or **Yield**, and rely on the “other side” to re-enable on return from its own **Sleep** or **Yield**.

```cpp
Yield() {
    IntStatus old = SetLevel(IntOff);
    next = scheduler->FindNextToRun();
    if (next != NULL) {
        scheduler->ReadyToRun(this);
        scheduler->Run(next);
    }
    interrupt->SetLevel(old);
}
```

```cpp
Sleep() {
    ASSERT(getLevel = IntOff);
    this->status = BLOCKED;
    next = scheduler->FindNextToRun();
    while(next = NULL) {
        /* idle */
        next = scheduler->FindNextToRun();
    }
    scheduler->Run(next);
}
```
What to Know about Sleep/Wakeup

1. *Sleep/wakeup* primitives are the fundamental basis for all blocking synchronization.

2. Correct use of *sleep/wakeup* requires some additional low-level mechanism to avoid missed and double wakeups.
   - disabling interrupts, and/or
   - constraints on preemption, and/or
   - spin-waiting with TSL or equivalent

3. These low-level mechanisms are tricky and error-prone.

4. High-level “magic” synchronization primitives take care of the details of using *sleep/wakeup*.
   - semaphores, mutexes, condition variables

Spin-Yield: Just Say No

```c
void Thread::Await() {
    awaiting = TRUE;
    while(awaiting)
        Yield();
}

void Thread::Awake() {
    if (awaiting)
        awaiting = FALSE;
}
```
The “Magic” of Semaphores and CVs

Any use of sleep/wakeup synchronization can be replaced with semaphores or condition variables.

- Most uses of blocking synchronization have some associated state to record the blocking condition.
  e.g., list or count of waiting threads, or a table or count of free resources, or the completion status of some operation, or....
  The trouble with sleep/wakeup is that the program must update the state atomically with the sleep/wakeup.

- Semaphores integrate the state into atomic P/V primitives.
  ....but the only state that is supported is a simple counter.

- Condition variables (CVs) allow the program to define the condition/state, and protect it with an integrated mutex.

The Roots of Condition Variables: Monitors

A monitor is a module (a collection of procedures) in which execution is serialized.

CVs are easier to understand if we think about them in terms of the original monitor formulation.

Java supports a variant of monitors called synchronized methods.

At most thread may be active in the monitor at a time.

A thread may \textit{wait} in the monitor, allowing another thread to enter.

A thread in the monitor may \textit{signal} a waiting thread, causing it to return from its \textit{wait} and reenter the monitor.
Hoare Semantics

Suppose purple signals blue in the previous example.

*Hoare semantics*: the signalled thread immediately takes over the monitor, and the signaller is suspended.

![Diagram showing Hoare semantics](image1)

The signaller cannot continue in the monitor until the signalled thread exits or waits again.

Mesa Semantics

Suppose again that purple signals blue in the original example.

*Mesa semantics*: the signalled thread transitions back to the *ready* state.

![Diagram showing Mesa semantics](image2)

There is no *suspended* state; the signaller continues until it exits the monitor or waits.

BUT: the signalled thread must examine the monitor state again after the *wait*, as the state may have changed since the *signal*. **Loop before you leap!**
Nachos mutexes and condition variables are based on monitors, but are more flexible.

- A monitor is “just like” a module whose state includes a mutex and a condition variable.
- It’s “just as if” the module’s methods Acquire the mutex on entry and Release the mutex before returning.
- With mutexes, the critical regions within the methods can be defined at a finer grain, to allow more concurrency.
- With condition variables, the module methods may wait and signal on multiple independent conditions.
- Nachos (and Topaz and Java) use Mesa semantics for their condition variables: loop before you leap!

```
void Down() {
    mutex->Acquire();
    ASSERT(count >= 0);
    while(count == 0) {
        condition->Wait(mutex);
    }
    count = count - 1;
    mutex->Release();
}

void Up() {
    mutex->Acquire();
    count = count + 1;
    condition->Signal(mutex);
    mutex->Release();
}
```

This constitutes a proof that mutexes and condition variables are just as powerful as semaphores.
Semaphores vs. Condition Variables

1. *Up* differs from *Signal* in that:
   - *Signal* has no effect if no thread is waiting on the condition.
     Condition variables are not variables! They have no value!
   - *Up* has the same effect whether or not a thread is waiting.
     Semaphores retain a memory of all calls to *Up*.

2. *Down* differs from *Wait* in that:
   - *Down* checks the condition and blocks only if necessary.
     no need to recheck the condition after returning from *Down*
     wait condition is defined internally, but is limited to a counter
   - *Wait* is explicit: it does not check the condition, ever.
     condition is defined externally and protected by integrated mutex

Example: A Counting Queue

```cpp
void CountingQueue::Await() {
    mx->Acquire();
    count++;
    cv->Wait();
    mx->Release();
}

void CountingQueue::Awake() {
    mx->Acquire();
    if (count > 0) {
        count--;
        cv->Signal();
    }
    mx->Release();
}
```

**What do we assume about Wait/Signal?**

**Will this work using Birrell’s CV semantics?**

Birrell claims that (given Mesa semantics) it is acceptable for a *signal* to wake up more than one waiter.
("wake up at least one")
Example: A Better Counting Queue

```cpp
void CountingQueue::Await()
{
    mx->Acquire();
    count++;
    cv->Wait();
    count--;
    mx->Release();
}
```

This version will work even if condition variables have Birrell’s “wake up at least one” semantics.

```cpp
void CountingQueue::Awake()
{
    mx->Acquire();
    if (count > 0)
        cv->Signal();
    mx->Release();
}
```

How would we do this with a semaphore?

Counting Queue with Semaphores

```cpp
mx->Init(1);
cv->Init(0);

void CountingQueue::Await()
{
    mx->P();
    count++;
    cv->P();
    count--;
    mx->V();
}
```

Replace our mutex with a binary semaphore.

```cpp
void CountingQueue::Awake()
{
    mx->P();
    if (count > 0)
        cv->V();
    mx->V();
}
```

Replace our condition variable with a semaphore whose value is always zero, so P always sleeps and V always wakes up.

But the resulting code is harder to understand.
void PingPong() {
    mx->Acquire();
    while(not done) {
        cv->Signal();
        cv->Wait();
    }
    mx->Release();
}

Will this work using Birrell’s CV semantics?

How would you do this using semaphores?

Will your Lab #2 condition variables execute this example correctly?