Sleep/Wakeup and Condition Variables

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.
Await/Awake with Sleep/Wakeup (first cut)

```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” */
}
```

The Trouble with Sleep/Wakeup

```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

Thread* waiter = 0;
void await() {
    disable interrupts
    waiter = currentThread; /* “I’m sleeping” */
currentThread->Sleep(); /* sleep */
    enable interrupts
}
void awake() {
    disable interrupts
    if (waiter) /* wakeup */
        scheduler->ReadyToRun(waiter);
        waiter = (Thread*)0; /* “you’re awake” */
    enable interrupts
}

Disabling interrupts prevents a context switch between “I’m sleeping” and “sleep”.

Nachos Thread::Sleep requires disabling interrupts (???).

Disabling interrupts prevents a context switch between “wakeup” and “you’re awake”.

Will this work on a multiprocessor?

What to Know about Sleep/Wakeup

1. *Sleep/wakeup* primitives are the fundamental basis for all blocking synchronization.
2. All 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 *(Unix kernels use this instead of disabling interrupts)*
   - spin-waiting, e.g., with TSL *(on a multiprocessor)*
3. These low-level mechanisms are tricky and error-prone.
4. High-level synchronization primitives take care of the details of using *sleep/wakeup*, hiding them from the caller.
   - semaphores, mutexes, condition variables
Digression: Sleep and Yield in Nachos

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

Sleep() {
  ASSERT(getLevel = IntOff);
  this->status = BLOCKED;
  next = scheduler->FindNextToRun();
  while(next = NULL) {
    /* idle */
    next = scheduler->FindNextToRun();
  }
  scheduler->Run(next);
}

Context switch itself is a critical section, which we enter only via Sleep or Yield.

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.

A New Synchronization Problem: Ping-Pong

void PingPong() {
  while(not done) {
    if (blue)
      switch to purple;
    if (purple)
      switch to blue;
  }
}

How to do this correctly using sleep/wakeup?

How to do it without using sleep/wakeup?
Ping-Pong with Sleep/Wakeup?

```c
void PingPong () {
    while(not done) {
        blue->Sleep();
        purple->Wakeup();
    }
}
```

Ping-Pong with Mutexes?

```c
void PingPong() {
    while(not done) {
        Mx->Acquire();
        Mx->Release();
    }
}
```
Mutexes Don’t Work for Ping-Pong

Condition Variables

*Condition variables* allow explicit event notification.

- much like a souped-up *sleep/wakeup*
- associated with a mutex to avoid *sleep/wakeup* races

\[
\begin{align*}
\text{Condition::Wait(Lock*)} & : \text{Called with lock held: sleep, atomically releasing lock.} \\
& \quad \text{Atomically reacquire lock before returning.} \\
\text{Condition::Signal(Lock*)} & : \text{Wake up one waiter, if any.} \\
\text{Condition::Broadcast(Lock*)} & : \text{Wake up all waiters, if any.}
\end{align*}
\]
Ping-Pong Using Condition Variables

```
void PingPong() {
    mx->Acquire();
    while(not done) {
        cv->Signal();
        cv->Wait();
    }
    mx->Release();
}
```

See how the associated mutex avoids sleep/wakeup races?

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

Using Condition Variables

```c
Condition *cv;
Lock* cvMx;
int waiter = 0;

void await() {
    cvMx->Lock();
    waiter = waiter + 1; /* “I’m sleeping” */
    cv->Wait(cvMx); /* sleep */
    cvMx->Unlock();
}

void wakeup() {
    cvMx->Lock();
    if (waiter) 
        cv->Signal(cvMx);
    waiter = waiter - 1;
    CvMx->Unlock();
}
```

Must hold lock when calling `Wait`.

`Wait` atomically releases lock and sleeps until next `Signal`.

`Wait` atomically reacquires lock before returning.

Association with mutex allows threads to safely manage state related to the sleep/wakeup coordination (e.g., `waiters` count).
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.

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

A thread may wait in the monitor, allowing another thread to enter.

A thread in the monitor may signal a waiting thread, causing it to return from its wait and reenter the monitor.

Hoare Semantics

Suppose purple signals blue in the previous example.

*Hoare semantics:* the signaled thread immediately takes over the monitor, and the signaler is suspended.

The signaler does not continue in the monitor until the signaled thread exits or waits again.

Hoare semantics allow the signaled thread to assume that the state has not changed since the signal that woke it up.
Mesa Semantics

Suppose again that purple signals blue in the original example.

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

There is no **suspended** state: the signaler continues until it exits the monitor or waits.

The signaled thread contends with other ready threads to (re)enter the monitor and return from *wait*.

Mesa semantics are easier to understand and implement...

BUT: the signaled thread must examine the monitor state again after the *wait*, as the state may have changed since the *signal*.

*Loop before you leap!*

---

From Monitors to Mx/Cv Pairs

Mutexes and condition variables (as in Nachos) are based on monitors, but they 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.
- But 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!*
**Mutual Exclusion in Java**

Mutexes and condition variables are built in to every Java object.

- no explicit classes for mutexes and condition variables

Every object is/has a "monitor".

- At most one thread may "own" any given object’s monitor.
- A thread becomes the owner of an object’s monitor by executing a method declared as `synchronized`
  
  some methods may choose not to enforce mutual exclusion (unsynchronized)
  
  by executing the body of a `synchronized` statement
  
  supports finer-grained locking than “pure monitors” allow
  
  exactly identical to the Modula-2 "LOCK(m) DO" construct in Birrell

---

**Wait/Notify in Java**

Every Java object may be treated as a condition variable for threads using its monitor.

```java
public class PingPong (extends Object) {
    public synchronized void PingPong() {
        while(true) {
            notify();
            wait();
        }
    }
}
```

A thread must own an object’s monitor to call `wait/notify`, else the method raises an `IllegalMonitorStateException`.

```java
public class Object {
    void notify(); /* signal */
    void notifyAll(); /* broadcast */
    void wait();
    void wait(long timeout);
}
```

```java
public class PingPong (extends Object) {
    public synchronized void PingPong() {
        while(true) {
            notify();
            wait();
        }
    }
}
```

Wait(*) waits until the timeout elapses or another thread notifies, then it waits some more until it can re-obtain ownership of the monitor: *Mesa semantics.*

*Loop before you leap!*
Spin-Yield: Just Say No

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

void
Thread::Awake() {
    if (awaiting)
        awaiting = FALSE;
}
```