Monitors and Semaphores

Annotated Condition Variable Example

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

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

void awake() {
    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 lock/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 “magic” module (a collection of procedures and state) with serialized execution and integrated wait/signal primitives. [Brinch Hansen 1973, C.A.R. Hoare 1974]

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 a given monitor at any 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, and a waiting blue is selected to wake up.

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 (Nachos, Topaz, Java).

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 the monitors concept, but they are more flexible.

- A monitor is “just like” a module whose state includes a mutex and a condition variable.

  The difference is syntactic; the basic semantics (and implementation) are the same for mutex/CV and monitors.

- It’s “just 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.
**Mutual Exclusion in Java**

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

- no explicit classes for mutuxes 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 or block
  
  `synchronized` construct specifies which object to acquire
  
  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 {
    public synchronized void notify(); /* signal */
    public synchronized void notifyAll(); /* broadcast */
    public void wait();
    public void wait(long timeout);
}
```

Loop before you leap!
**Semaphores**

Semaphores handle all of your synchronization needs with one elegant but confusing abstraction.

- controls allocation of a resource with multiple instances
- a non-negative integer with special operations and properties
  - initialize to arbitrary value with `Init` operation
  - “souped up” increment (`Up` or `V`) and decrement (`Down` or `P`)
- atomic sleep/wakeup behavior implicit in `P` and `V`
  - `P` does an atomic **sleep**, if the semaphore value is zero.
    `P` means “probe”; it cannot decrement until the semaphore is positive.
  - `V` does an atomic **wakeup**.
    `num(P) <= num(V) + init`

---

**Semaphores as Mutexes**

Semaphores must be initialized with a value representing the number of free resources: mutexes are a single-use resource.

```c
semaphore->Init(1);
void Lock::Acquire()
{
    semaphore->Down();
}
void Lock::Release()
{
    semaphore->Up();
}
```

*Down*() to acquire a resource; blocks if no resource is available.

*Up*() to release a resource; wakes up one waiter, if any.

*Up and Down are atomic.*

Mutexes are often called **binary semaphores**.
However, “real” mutexes have additional constraints on their use.
### Ping-Pong with Semaphores

```c
blue->Init(0);
purple->Init(1);

void PingPong() {
    while(not done) {
        blue->P();
        Compute();
        purple->V();
    }
}
```

### Ping-Pong with One Semaphore?

```c
sem->Init(0);
blue:    { sem->P(); PingPong(); }
purple: { PingPong(); }

void PingPong() {
    while(not done) {
        Compute();
        sem->V();
        sem->P();
    }
}
```
Ping-Pong with One Semaphore?

```c
void PingPong() {
    while(!done) {
        Compute();
        sem->V();
        sem->P();
    }
}
```

Nachos semaphores have Mesa-like semantics:
- They do not guarantee that a waiting thread wakes up “in time” to consume the count added by a V().
- Semaphores are not “fair”
- No count is “reserved” for a waking thread
- Uses “passive” vs. “active” implementation

```
blue: { sem->P(); PingPong(); }
purple: { PingPong(); }
```

Another Example With Dual Semaphores

```c
blue->Init(0);
purple->Init(0);

void Blue() {
    while(not done) {
        Compute();
        purple->V();
        blue->P();
    }
}

void Purple() {
    while(not done) {
        Compute();
        blue->V();
        purple->P();
    }
}
```
**Basic Barrier**

```
void IterativeCompute() {
    while(not done) {
        Compute();
        purple->V();
        blue->P();
    }
}
```

**How About This? (#1)**

```
void IterativeCompute() {
    while(not done) {
        Compute();
        blue->V();
        purple->P();
    }
}
```
How About This? (♯2)

\[
\begin{align*}
blue & \rightarrow \text{Init}(1); \\
purple & \rightarrow \text{Init}(0); \\
\text{void} & \text{IterativeCompute?()} \{ \\
& \text{while(not done) } \{ \\
& \quad blue \rightarrow \text{P}(); \\
& \quad \text{Compute}(); \\
& \quad purple \rightarrow \text{V}(); \\
& \} \\
& \} \\
\end{align*}
\]

How About This? (♯3)

\[
\begin{align*}
blue & \rightarrow \text{Init}(1); \\
purple & \rightarrow \text{Init}(0); \\
\text{void} & \text{CallThis()} \{ \\
& \quad blue \rightarrow \text{P}(); \\
& \quad \text{Compute}(); \\
& \quad purple \rightarrow \text{V}(); \\
& \} \\
\} \\
\text{void} & \text{CallThat()} \{ \\
& \quad purple \rightarrow \text{P}(); \\
& \quad \text{Compute}(); \\
& \quad blue \rightarrow \text{V}(); \\
& \} \\
\} \\
\end{align*}
\]
void CallThis() {
    blue->P();
    Compute();
    purple->V();
}

void CallThat() {
    purple->P();
    Compute();
    blue->V();
}

blue->Init(1);
purple->Init(0);

void Produce(int m) {
    empty->P();
    buf = m;
    full->V();
}

void Consume() {
    int m;
    full->P();
    m = buf;
    empty->V();
    return(m);
}

empty->Init(1);
full->Init(0);
int buf;

This use of a semaphore pair is called a split binary semaphore: the sum of the values is always one.
### A Bounded Resource

```c
int AllocateEntry() {
    int i;
    while (!FindFreeItem(&i))
        block and wait for a free slot
    slot[i] = 1; /* grab free slot */
    return(i);
}

void ReleaseEntry(int i) {
    slot[i] = 0;
    wakeup waiter, if any
}

boolean FindFreeItem(int* index) {
    for (i = 0; i < TableSize; i++)
        if (slot[i] == 0) return it;
    return (FALSE);
}
```

### A Bounded Resource with a Counting Semaphore

```c
semaphore->Init(N);

int AllocateEntry() {
    int i;
    semaphore->Down();
    ASSERT(FindFreeItem(&i));
    slot[i] = 1;
    return(i);
}

void ReleaseEntry(int i) {
    slot[i] = 0;
    semaphore->Up();
}
```

A semaphore for an N-way resource is called a **counting semaphore**.

A caller that gets past a **Down** is guaranteed that a resource instance is reserved for it.

**Problems?**

*Note:* the current value of the semaphore is the number of resource instances free to allocate.

But semaphores do not allow a thread to read this value directly. Why not?
Spin-Yield: Just Say No

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

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