The Synchronization Toolbox

Mutual Exclusion

Race conditions can be avoided by ensuring mutual exclusion in critical sections.

- Critical sections are code sequences that are vulnerable to races.
  Every race (possible incorrect interleaving) involves two or more threads executing related critical sections concurrently.
- To avoid races, we must serialize related critical sections.
  Never allow more than one thread in a critical section at a time.

Mutual Exclusion

Race conditions can be avoided by ensuring mutual exclusion in critical sections.

- Critical sections are code sequences that are vulnerable to races.
  Every race (possible incorrect interleaving) involves two or more threads executing related critical sections concurrently.
- To avoid races, we must serialize related critical sections.
  Never allow more than one thread in a critical section at a time.

Locks

Locks can be used to ensure mutual exclusion in conflicting critical sections.

- A lock is an object, a data item in memory.
- Methods: `Lock::Acquire` and `Lock::Release`.
- Threads pair calls to `Acquire` and `Release`.
- `Acquire` before entering a critical section.
- `Release` after leaving a critical section.
- Between `Acquire` and `Release`, the lock is `held`.
- `Acquire` does not return until any previous holder releases.
- Waiting locks can spin (a `spinlock`) or block (a `mutex`).

Example: Per-Thread Counts and Total

- Increment a counter by a specified value, and keep a running sum.
- `TouchCount` is called repeatedly by each of `N` threads.
- `tid` is an integer thread identifier for the current thread.
- `value` is just some arbitrary number.

```c
void TouchCount(int tid, int value) {
    counters[tid] += value; /* critical section code is atomic... */
    total += value; /* ...as long as the lock is held */
    lock->Release();
}
```

Using Locks: An Example

```c
int counters[N];
int total;
Lock *lock;

void TouchCount(int tid, int value) {
    lock->Acquire();
    counters[tid] += value; /* critical section code is atomic... */
    total += value; /* ...as long as the lock is held */
    lock->Release();
}
```

Reading Between the Lines of C

```c
load counters, R1; load counters base
load [tid], R2, R1; load tid index
mul R2, #32, R2; index = index * sizeof(int)
load [tid], R1, R2; compute index to array
load [R1], R3; load value
load [R1], R2, R3; counters[tid] += value
store [R1], R2; store back to counters[tid]
load R2, R1; load total
load R2, R3, R2; total += value
store R2, R1; store total
```

Counters shared by all threads?/!
int counters[N];
int total;
Lock *lock;

Increment a counter by a specified value, and keep a running sum.
This is called repeatedly by each of `N` threads.
`tid` is an integer thread identifier for the current thread.
`value` is just some arbitrary number.

```c
void TouchCount(int tid, int value) {
    counters[tid] += value; /* critical section code is atomic... */
    total += value; /* ...as long as the lock is held */
    lock->Release();
}
```
**Portrait of a Lock in Motion**

**Condition Variables**

Condition variables allow explicit event notification.

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

- Condition::Wait(Lock*)
  Called with lock held: sleep, atomically releasing lock.
  Atomically reacquire lock before returning.

- Condition::Signal(Lock*)
  Wake up one waiter, if any.

- Condition::Broadcast(Lock*)
  Wake up all waiters, if any.

**A New Synchronization Problem: Ping-Pong**

```c
void PingPong() {
    while(!done) {
        if (blue)
            switch to purple;
        if (purple)
            switch to blue;
    }
}
```

**Ping-Pong with Sleep/Wakeup?**

```c
void PingPong() {
    while(!done) {
        blue->Sleep();
        purple->Wakeup();
    }
}
```

```c
void PingPong() {
    while(!done) {
        blue->Wakeup();
        purple->Sleep();
    }
}
```

**Ping-Pong with Mutexes?**

```c
void PingPong() {
    while(!done) {
        Mx->Acquire();
        Mx->Release();
    }
}
```

**Mutexes Don’t Work for Ping-Pong**
Ping-Pong Using Condition Variables

```c
void PingPong() {
    mx->Acquire();
    while(!done) {
        cv->Signal();
        cv->Wait();
    }
    mx->Release();
}
```

See how the associated mutex avoids sleep/wakeup races?

Semaphores using Condition Variables

```c
void Down() {
    mutex->Acquire();
    ASSERT(count >= 0);
    while(count == 0) {   // (loop before use count)
        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 at least as powerful as semaphores.

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.

But semaphores do not allow a thread to read this value directly. Why not?

Bounded Resource with a Condition Variable

```c
int AllocEntry() {
    int i;
    mutex->Acquire();
    while(FindFreeItem () )
        cv.Wait(mutex);
    slot[i] = 1;
    mutex->Release(),
    return(i);
}
```

```c
void ReleaseEntry(int i) {
    mutex->Acquire();
    slot[i] = 0;
    cv->Signal();
    mutex->Release();
}
```

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 means “probe”: it cannot decrement until the semaphore is positive.

V does an atomic wakeup.

num(P) <= num(V) + init

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?

The Roots of Condition Variables: Monitors

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

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 enter the monitor.

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

- 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.
- Mesa semantics are easier to understand and implement.

Suppose again that purple signals blue in the original example.

- Mesa semantics: the signaled thread transitions back to the ready state.
- Mesa semantics: the module methods Acquire and Release the mutex on entry and return it back.

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/Release the mutex on entry and return it back.
- 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 into every Java object.
  - A mutex is 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.

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
   - spin-waiting (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

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.

- Wait/notify: Wait 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!
Semaphores as Mutexes

Mutexes are often called **binary semaphores**. However, “real” mutexes have additional constraints on their use.

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

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

```
semaphore ->Init(1);
```

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.

Spin-Yield: Just Say No

```
void Thread::Await()
{
    awaiting = TRUE;
    while(awaiting)
    Yield();
}
void Thread::Awake()
{
    if (awaiting)
    awaiting = FALSE;
}
```

Spin and Down are atomic:

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 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* is explicit: it does not check the condition, ever.
   - Condition is defined externally and protected by an integrated mutex.

Guidelines for Choosing Lock Granularity

1. **Keep critical sections short.** Push “noncritical” statements outside of critical sections to reduce contention.
2. **Limit lock overhead.** Keep to a minimum the number of times mutexes are acquired and released.
   - Note tradeoff between contention and lock overhead.
3. **Use as few mutexes as possible, but no fewer.** Choose lock scope carefully: if the operations on two different data structures can be separated, it may be more efficient to synchronize those structures with separate locks.
   - Add new locks only as needed to reduce contention. “Correctness first, performance second!”

Tricks of the Trade #1

```
void Init() {
    InitThis(); InitThat();
    initialized = 1;
}
```

```
void DoSomething()
{
    if (!initialized) { /* fast unsynchronized read of a WORM datum */
        initMx.Lock(); /* gives us a “hint” that we’re in a race to write */
        if (!initialized)
            initMx.Unlock(); /* slow, safe path */
    }
    DoThis(); DoThat();
}
```
Things Your Mother Warned You About #1

```c
struct buffer {
    unsigned int flags;
    struct OtherStuff etc;
};

void MarkWired(buffer *b) {
    wiredLock.Acquire();
    b->flags |= WIRED;
    wiredList.Append(b);
    wiredLock.Release();
}

Lock dirtyLock;
List dirtyList;
Lock wiredLock;
List wiredList;

void MarkDirty(buffer* b) {
    dirtyLock.Acquire();
    b->flags |= DIRTY;
    dirtyList.Append(b);
    dirtyLock.Release();
}
```

More Locking Guidelines

1. Write code whose correctness is obvious.
2. Strive for symmetry.
   - Show the Acquire/Release pairs.
   - Factor locking out of interfaces.
   - Acquire and Release at the same layer in your “layer cake” of abstractions and functions.
3. Hide locks behind interfaces.
4. Avoid nested locks.
   - If you must have them, try to impose a strict order.
5. Sleep high; lock low.
   - Design choice: where in the layer cake should you put your locks?

Guidelines for Condition Variables

1. Understand/document the condition(s) associated with each CV.
   - What are the waiters waiting for?
2. Always check the condition to detect spurious wakeups after returning from a wait: “loop before you leap”!
   - Another thread may beat you to the mutex.
   - The signaler may be careless.
   - A single condition variable may have multiple conditions.
3. Don’t forget: signals on condition variables do not stack!
   - A signal will be lost if nobody is waiting: always check the wait condition before calling wait.

Stuff to Know

- Know how to use mutexes, CVs, and semaphores. It is a craft. Learn to think like Birrell: write concurrent code that is clean and obviously correct, and balances performance with simplicity.
- Understand why these abstractions are needed: sleep/wakeup races, missed wakeups, double wakeups, interleavings, critical sections, the adversarial scheduler, multiprocessors, thread interactions, ping-pong.
- Understand the variants of the abstractions: Hoare vs. Mesa semantics, monitors vs. mutexes, binary semaphores vs. counting semaphores, spinlocks vs. blocking locks.
- Understand the contexts in which these primitives are needed, and how those contexts are different: processes or threads in the kernel, interrupts, threads in a user program, servers, architectural assumptions.
- Where should we define/implement synchronization abstractions? Kernel? Library? Language/compiler?
- Reflect on scheduling issues associated with synchronization abstractions: how much should a good program constrain the scheduler? How much should it assume about the scheduling semantics of the primitives?