

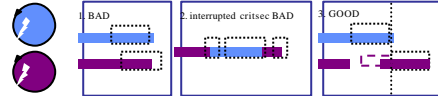
The Synchronization Toolbox

DUKE
OpenJDK Architecture

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.

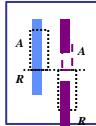


DUKE
OpenJDK Architecture

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/Release*, the lock is *held*.
- *Acquire* does not return until any previous holder releases.
- Waiting locks can spin (a *spinlock*) or block (a *mutex*).



DUKE
OpenJDK Architecture

Example: Per-Thread Counts and Total

```

/* shared by all threads */
int counters[N];
int total;

/*
 * 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.
 */
void
TouchCount(int tid, int value)
{
    counters[tid] += value;
    total += value;
}
    
```



DUKE
OpenJDK Architecture

Using Locks: An Example

```

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

/*
 * Increment a counter by a specified value, and keep a running sum.
 */
void
TouchCount(int tid, int value)
{
    lock->Acquire();
    counters[tid] += value;
    total += value;
    lock->Release();
}
    
```



DUKE
OpenJDK Architecture

Reading Between the Lines of C

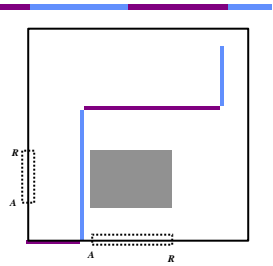
```

/*
 * counters[tid] += value;
 * total += value;
 */
load    counters, R1      ; load counters base
load    8(SP), R2        ; load tid index
shl     R2, #2, R2       ; index = index * sizeof(int)
add     R1, R2, R1       ; compute index to array
load    4(SP), R3        ; load value
load    (R1), R2         ; load counters[tid]
add     R2, R3, R2       ; counters[tid] += value
store   R2, (R1)         ; store back to counters[tid]
load    total, R2        ; load total
add     R2, R3, R2       ; total += value
store   R2, total        ; store total
    
```



DUKE
OpenJDK Architecture

Portrait of a Lock in Motion



DUKE
Design & Architecture

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.
```

DUKE
Design & Architecture

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*?

DUKE
Design & Architecture

Ping-Pong with Sleep/Wakeup?

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

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

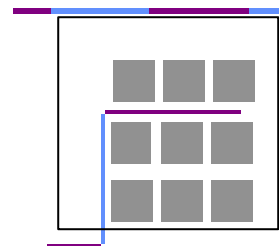
DUKE
Design & Architecture

Ping-Pong with Mutexes?

```
void
PingPong() {
    while(not done) {
        Mx->Acquire();
        Mx->Release();
    }
}
```

DUKE
Design & Architecture

Mutexes Don't Work for Ping-Pong



DUKE
Design & Architecture

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?

Bounded Resource with a Condition Variable

```
Mutex* mx;
Condition *cv;
```

```
int AllocateEntry() {
    int i;
    mx->Acquire();
    while(!FindFreeItem(&i))
        cv.Wait(mx);
    slot[i] = 1;
    mx->Release();
    return(i);
}
```

Loop before you leap!

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

Why is this Acquire needed?

Semaphores using Condition Variables

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

(Loop before you leap!)

```
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.

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*.
 - $num(P) \leq num(V) + init$

A Bounded Resource with a Counting Semaphore

```
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?

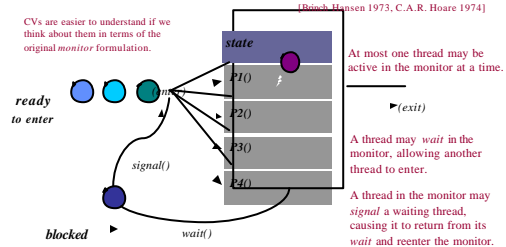
Note: the current value of the semaphore is the number of resource instances free to allocate.

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

The Roots of Condition Variables: Monitors

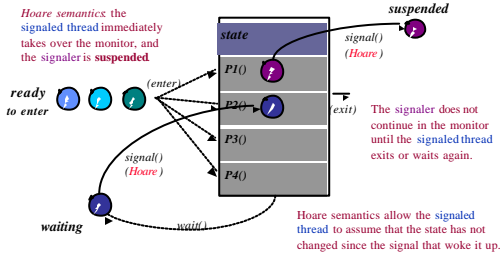
A *monitor* is a module (a collection of procedures) in which execution is serialized.

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



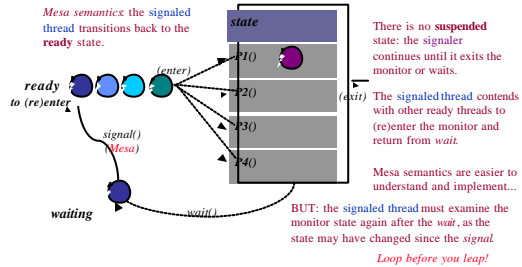
Hoare Semantics

Suppose purple signals blue in the previous example.



Mesa Semantics

Suppose again that purple signals blue in the original example.



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.

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

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*.

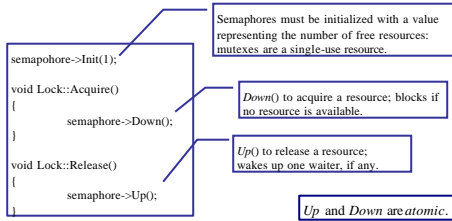
Wait(*) waits until the timeout elapses or another thread notifies, then it waits some more until it can re-obtain ownership of the monitor. *Mesa semantics*

Loop before you leap!

What to Know about Sleep/Wakeup

- Sleep/wakeup* primitives are the fundamental basis for *all* blocking synchronization.
- 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 (on a multiprocessor)
- These low-level mechanisms are tricky and error-prone.
- High-level synchronization primitives take care of the details of using *sleep/wakeup*, hiding them from the caller.
 - semaphores, mutexes, condition variables

Semaphores as Mutexes



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

DUKE
JVM & Architecture

Spin-Yield: Just Say No

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

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

DUKE
JVM & Architecture

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 *PV* 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.

DUKE
JVM & Architecture

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 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

DUKE
JVM & Architecture

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!"

DUKE
JVM & Architecture

Tricks of the Trade #1

```
int initialized = 0;
Lock initMx;

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) /* have to check again while holding the lock */
            Init();
        initMx.Unlock(); /* slow, safe path */
    }
    DoThis(); DoThat();
}
```

DUKE
JVM & Architecture

Things Your Mother Warned You About #1

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

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

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

#define WIRED 0x1
#define DIRTY 0x2
#define FREE 0x4

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

DUKE
Design & Architecture

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 ?

DUKE
Design & Architecture

Guidelines for Condition Variables

1. Understand/document the condition(s) associated with each CV.
 - What are the waiters waiting for?
 - When can a waiter expect a *signal*?
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*.

DUKE
Design & Architecture

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 wakeup, double wakeup, interleavings, critical sections, the adversarial scheduler, multiprocessors, thread interactions, ping-pong.
- Understand the variants of the abstractions: Mesa vs. Hoare 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?

DUKE
Design & Architecture