Synchronization: Going Deeper

A reader/write lock or SharedLock is a new kind of “lock” that is similar to our old definition:
- supports Acquire and Release primitives
- guarantees mutual exclusion when a writer is present

**But** a SharedLock provides better concurrency for readers when no writer is present.

```java
class SharedLock {
    AcquireRead(); // shared mode
    AcquireWrite(); // exclusive mode
    ReleaseRead();
    ReleaseWrite();
}
```

**Reader/Writer Lock Illustrated**

- Multiple readers may hold the lock concurrently in shared mode.
- Writers always hold the lock in exclusive mode and must wait for all readers or writers to exit.

**Reader/Writer Lock: First Cut**

```java
int i; /* # active readers, or -1 if writer */
Lock rwMx;
Condition rwCv;
SharedLock::
    AcquireWrite() {
        rwMx.Acquire();
        while (i != 0)
            rwCv.Wait(& rwMx);
        i = -1;
        rwMx.Release();
    }
    AcquireRead() {
        rwMx.Acquire();
        while (i < 0)
            rwCv.Wait(& rwMx);
        i += 1;
        rwMx.Release();
    }
    ReleaseWrite() {
        rwMx.Acquire();
        i = 0;
        rwCv.Broadcast();
        rwMx.Release();
    }
    ReleaseRead() {
        rwMx.Acquire();
        i -= 1;
        if (i == 0)
            rwCv.Signal();
        rwMx.Release();
    }
```

**The Little Mutex Inside SharedLock**

**Limitations of the SharedLock Implementation**

This implementation has weaknesses discussed in [Birrell89].
- **spurious lock conflicts** (on a multiprocessor): multiple waiters contend for the mutex after a signal or broadcast.
  - *Solution*: drop the mutex before signaling (if the signal primitive permits it.)
- **spurious wakeups**
  - *Solution*: add a separate condition variable for writers.
- **starvation**
  - How can we be sure that a waiting writer will ever pass its acquire if faced with a continuous stream of arriving readers?
Reader/Writer Lock: Second Try

```c
SharedLock::AcquireWrite() {
  rwMx.Acquire();
  while (i != 0)
    wCv.Wait(& rwMx);
  i = -1;
  rwMx.Release();
}

SharedLock::AcquireRead() {
  rwMx.Acquire();
  while (i < 0)
    ...rCv.Wait(& rwMx);...
  i += 1;
  rwMx.Release();
}

SharedLock::ReleaseWrite() {
  rwMx.Acquire();
  i = 0;
  if (readersWaiting)
    rCv.Broadcast();
  else
    wcv.Signal();
  rwMx.Release();
}

SharedLock::ReleaseRead() {
  rwMx.Acquire();
  i -= 1;
  if (i == 0)
    wCv.Signal();
  rwMx.Release();
}
```

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`.

Starvation

Starvation illustrates under load, a writer will be stalled forever by a stream of readers.

- **Example**: a one-lane bridge or tunnel.
  - Wait for oncoming car to exit the bridge before entering. Repeat as necessary.
- **Problem**: a “writer” may never be able to cross if faced with a continuous stream of oncoming “readers”.
- **Solution**: some reader must politely stop before entering, even though it is not forced to wait by oncoming traffic.
  - Use extra synchronization to control the lock scheduling policy. Complicates the implementation: optimize only if necessary.

Deadlock

Deadlock is closely related to starvation.

- Processes wait forever for each other to wake up and/or release resources.
- **Example**: traffic gridlock.
  - The difference between deadlock and starvation is subtle.
    - With starvation, there always exists a schedule that feeds the starving party.
      - The situation may resolve itself... if you’re lucky.
    - Once deadlock occurs, it cannot be resolved by any possible future schedule...
      - though there may exist schedules that avoid deadlock.

Dining Philosophers

- N processes share N resources
- resource requests occur in pairs
- random think times
- hungry philosopher grabs a fork
  - ...and doesn’t let go
  - ...until the other fork is free
- ...and the linguine is eaten

Four Preconditions for Deadlock

Four conditions must be present for deadlock to occur:

1. **Non-preemptabililty.** Resource ownership (e.g., by threads) is non-preemptable.
   - Resources are never taken away from the holder.
2. **Exclusion.** Some thread cannot acquire a resource that is held by another thread.
3. **Hold-and-wait.** Holder blocks awaiting another resource.
4. **Circular waiting.** Threads acquire resources out of order.
Resource Graphs

Given the four preconditions, some schedules may lead to circular waits.

- Deadlock is easily seen with a resource graph or wait-for graph.

The graph has a vertex for each process and each resource.

If process $A$ holds resource $R$, add an arc from $R$ to $A$.

If process $A$ is waiting for resource $R$, add an arc from $A$ to $R$.

The system is deadlocked if the wait-for graph has at least one cycle.

Not All Schedules Lead to Collisions

The scheduler chooses a path of the executions of the threads/processes competing for resources.

Synchronization constrains the schedule to avoid illegal states.

Some paths “just happen” to dodge dangerous states as well.

What is the probability that philosophers will deadlock?

- How does the probability change as:
  - think times increase?
  - number of philosophers increases?

Dealing with Deadlock

1. Ignore it. “How big can those black boxes be anyway?”
2. Detect it and recover. Traverse the resource graph looking for cycles before blocking any customer.
   - If a cycle is found, preempt: force one party to release and restart.
3. Prevent it statically by breaking one of the preconditions.
   - Assign a fixed partial ordering to resources; acquire in order.
   - Use locks to reduce multiple resources to a single resource.
   - Acquire resources in advance of need; release all to retry.
4. Avoid it dynamically by denying some resource requests. Banker’s algorithm.
Extending the Resource Graph Model

Reasoning about deadlock in real systems is more complex than the simple resource graph model allows.
- Resources may have multiple instances (e.g., memory).
- Processes may block to await events as well as resources.
  E.g., A and B each rely on the other to wake them up for class.

Banker’s Algorithm

The Banker’s Algorithm is the classic approach to deadlock avoidance (choice 4) for resources with multiple units.
1. Assign a credit limit to each customer.
   “maximum claim” must be stated/negotiated in advance
2. Reject any request that leads to a dangerous state.
   A dangerous state is one in which a sudden request by any customer(s) for the full credit limit could lead to deadlock.
   A recursive reduction procedure recognizes dangerous states.
3. In practice, this means the system must keep resource usage well below capacity to maintain a reserve surplus.
   Rarely used in practice due to low resource utilization.

Implementing Spinlocks: First Cut

```c
class Lock {
    int held;
}

void Lock::Acquire() {
    while (held);
    held = 1;
}

void Lock::Release() {
    held = 0;
}
```

Spinlocks: What Went Wrong

```c
void Lock::Acquire() {
    while (held);
    /* test */
    held = 1;
    /* set */
}

void Lock::Release() {
    held = 0;
}
```

What Are We Afraid Of?

Potential problems with the “rough” spinlock implementation:
1. races that violate mutual exclusion
   - involuntary context switch between test and set
   - on a multiprocessor, race between test and set on two CPUs
2. wasteful spinning
   - lock holder calls deep or yield
   - interrupt handler acquires a busy lock
   - involuntary context switch for lock holder

Which are implementation issues, and which are problems with spinlocks themselves?

The Need for an Atomic “Toehold”

To implement safe mutual exclusion, we need support for some sort of “magic toehold” for synchronization.
- The lock primitives themselves have critical sections to test and/or set the lock flags.
- These primitives must somehow be made atomic.
- Uninterruptible

Two solutions:
1. hardware support: atomic instructions (test-and-set)
2. scheduler control: disable timeslicing (disable interrupts)
**Atomic Instructions: Test-and-Set**

Spinlock: Acquire()
- while(held);
- held = 1;

Wrong:
- Load 4(SP), R2 ; load "this"
- busywait:
  - Load 4(R2), R3 ; load "held" flag
  - Branch if held wasn’t zero
  - Store #1, 4(R2) ; held = 1

Right:
- Load 4(SP), R2 ; load "this"
- busywait:
  - TSL 4(R2), R3 ; test-and-set this->held
  - Branch if held wasn’t zero

Solution: TSL atomically sets the flag and leaves the old value in a register.

**On Disabling Interrupts**

Nachos has a primitive to disable interrupts, which we will use as a toehold for synchronization.

- Temporarily block notification of external events that could trigger a context switch.
  - E.g., clock interrupts (ticks) or device interrupts
- In a “real” system, this is available only to the kernel.
- Why?
- Disabling interrupts is insufficient on a multiprocessor.
  - It is thus a dumb way to implement spinlocks.
  - We will use it ONLY as a toehold to implement “proper” synchronization.
  - A blunt instrument to use as a last resort

**Implementing Locks: Another Try**

```c
class Lock {

public:
    void Acquire() {
        disable interrupts;
    }

    void Release() {
        enable interrupts;
    }
};
```

Problems?

**Implementing Mutexes: Rough Sketch**

```c
class Lock {

private:
    int held;
    List sleepers;

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

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

**Nachos Thread States and Transitions**

- **running**
- **ready**
- **sleep**
- **yield**
  - voluntary or involuntary

**Implementing Mutexes: A First Cut**

```c
class Lock {

public:
    int held;
    List sleepers;

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

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

```c
// Nachos Thread States and Transitions

// Implementing Mutexes: A First Cut
```
Mutexes: What Went Wrong

```c
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());
}
```

Potential missed wakeup: holder could Release before thread is on sleepers list.
Potential missed wakeup: holder could call to wake up before we are "fully asleep".

Race to acquire: two threads could observe held == 0 concurrently, and think they both can acquire the lock.

Potential corruption of sleepers list in a race between two Acquires or an Acquire and a Release.

Using Sleep/Wakeup Safely

```c
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);
    enable interrupts
}
```

Disabling interrupts prevents a context switch between "I'm sleeping" and "sleep".
Disabling interrupts prevents a context switch between "wakeup" and "you're awake".

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
   (Unix kernels use this instead of disabling interrupts)
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

Races: A New Definition

A program P’s Acquire events impose a partial order on memory accesses for each execution of P.
- Memory access event x1 happens before x2 if the synchronization orders x1 before x2 in that execution.
- If neither x1 nor x2 happens before the other in that execution, then x1 and x2 are concurrent.

P has a race if there exists some execution of P containing accesses x1 and x2 such that:
- Accesses x1 and x2 are conflicting.
- Accesses x1 and x2 are concurrent.

Locks and Ordering

```c
mx->Acquire();
x = x + 1;
x = x + 1;
x = x + 1;
mx->Release();
```
**Possible Interleavings?**

Possible Interleavings?

1. `mx->Acquire();`  
   `x = x + 1;`  
   `mx->Release();`

2. `mx->Acquire();`  
   `x = x + 1;`  
   `mx->Release();`

3. `mx->Acquire();`  
   `x = x + 1;`  
   `mx->Release();`

4. `mx->Acquire();`  
   `x = x + 1;`  
   `mx->Release();`

**Understand ....**

1. What if the two access pairs were to different variables `x` and `y`?
2. What if the access pairs were protected by different locks?
3. What if the accesses were all reads?
4. What if only one thread modifies the shared variable?
5. What about “variables” consisting of groups of locations?
6. What about “variables” that are fields within locations?
7. What’s a location?
8. Is every race an error?

**Locks and Ordering Revisited**

1. What ordering does happened-before define for acquires on a given mutex?
2. What ordering does happened-before define for acquires on different mutexes?
   Can a data item be safely protected by two locks?
3. When happened-before orders `x_1` before `x_2`, does every execution of `P` preserve that ordering?
4. What can we say about the happened-before relation for a single-threaded execution?

**A Look (Way) Ahead**

The happened-before relation, conflicting accesses, and synchronization events will keep coming back.
- Concurrent executions, causality, logical clocks, vector clocks are fundamental to distributed systems of all kinds.
  - Replica consistency (e.g., TACT)
  - Message-based communication and consistent delivery order
- Parallel machines often leverage these ideas to allow weakly ordered memory system behavior for better performance.
  - Cache-coherent NUMA multiprocessors
  - Distributed shared memory
- Goal: learn to think about concurrency in a principled way.

**Building a Data Race Detector**

**Race Detection Alternatives**

A locking discipline is a synchronization policy that ensures absence of data races.
- `P` follows a locking discipline iff no concurrent conflicting accesses occur in any legal execution of `P`.

**Challenge** how to build a tool that tells us whether or not any `P` follows a consistent locking discipline?
- If we had one, we could save a list of time and aggravation.
  - Option 1: static analysis of the source code?
  - Option 2: execute the program and see if it works?
  - Option 3: dynamic observation of the running program to see what happens and what could have happened?

How good an answer can we get from these approaches?
Basic Lockset Algorithm

1. **Premise:** Each shared variable is covered by exactly one lock.
2. Which one is it? Refine "candidate" lockset for each variable.
3. If \( P \) executes a set of accesses to \( v \), and no lock is common to all of them, then (1) is false.

   For each variable \( v \), \( C(v) = \{\text{all locks}\} \)
   When thread \( t \) accesses \( v \):
   \[ C(v) = C(v) \cap \text{locks\_held}(t); \]
   if \( C(v) = \emptyset \)
   then \( \text{howl}() \);

Complications to the Lockset Algorithm

- "Fast" initialization
  First access happened-before \( v \) is exposed to other threads, thus it cannot participate in a race.
- WORM data
  The only write accesses to \( v \) happened-before \( v \) is exposed to other threads, thus read-only access after that point cannot participate in a race.
- SharedLock
  Read-only accesses are not mutually conflicting, thus they may proceed concurrently as long as no writer is present.
  SharedLock guarantees this without holding a mutex.
- Heap block caching/recycling above the heap manager?

Modified Lockset Algorithm

1. **Premise:** No checks.
2. **read or write by initial thread**
3. **write**
4. **read**

   - **exclusive**
   - **shared**
   - **virgin**
   - **write**
   - **read**

   No checks.
   Update \( C(v) \), but no warnings.

   Refine \( C(v) \), and warn if \( C(v) = \emptyset \).
   If read, consider only locks held in read mode. If write, consider only locks held in write mode.

The Eraser Paper

What makes this a good "systems" paper?
What is interesting about the Experience?
What validation was required to "sell" the idea?
How does the experience help to show the limitations (and possible future extensions) of the idea?
Why is the choice of applications important?
What are the "real" contributions relative to previous work?

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 \( \text{Init} \) operation
  "souped up" increment (\( \text{Up} \) or \( \text{V} \)) and decrement (\( \text{Down} \) or \( \text{P} \))
- \( \text{P} \) does an atomic sleep if the semaphore value is zero.
  \( \text{P} \) means "probe"; it cannot decrement until the semaphore is positive.
  \( \text{V} \) does an atomic wakeup.
  \( \text{num}(\text{P}) = \text{num}(\text{V}) + 1 \)
- atomic sleep/wakeup behavior implicit in \( \text{P} \) and \( \text{V} \)

Semaphores as Mutexes

Semaphores must be initialized with a value indicating the number of free resources.
Mutexes can be a single-use resource.

\( \text{Up} \) and \( \text{Down} \) are atomic.

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

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

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

Ping-Pong with One Semaphore?

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

```
sem->Init(0);
bblue: { sem->P(); PingPong(); }
```

```
purple: { PingPong(); }
```

---

Another Example With Dual Semaphores

```c
void Blue() {
    while(!done) {
        Compute();
        purple->V();
        blue->P();
    }
}
```

```
void Purple() {
    while(!done) {
        purple->P();
        Compute();
        blue->V();
    }
}
```

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

---

Basic Barrier

```c
void IterativeCompute() {
    while(!done) {
        Compute();
        purple->V();
        blue->P();
    }
}
```

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

---

How About This? (#1)

```c
void IterativeCompute() {
    while(!done) {
        blue->P();
        Compute();
        purple->V();
    }
}
```

```
void IterativeCompute() {
    while(!done) {
        purple->P();
        Compute();
        blue->V();
    }
}
```

```
blue->Init(1);
purple->Init(1);
```
void IterativeCompute() {
    while (not done) {
        purple->P();
        Compute();
        blue->V();
    }
}

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

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

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