Synchronization: Going Deeper

SharedLock: Reader/Writer Lock

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.

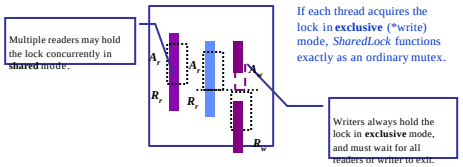
often used in database systems

easy to implement using mutexes and condition variables

a classic synchronization problem

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


Reader/Writer Lock Illustrated



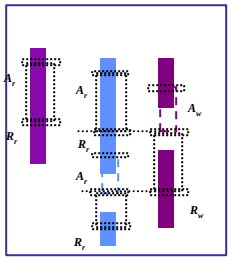
mode	read	write	max allowed
shared	yes	no	many
exclusive	yes	yes	one
not holder	no	no	many

Reader/Writer Lock: First Cut

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

```
SharedLock::ReleaseWrite() {
    rwMx.Acquire();
    i = 0;
    rwCv.Broadcast();
    rwMx.Release();
}

SharedLock::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*
  - *ReleaseWrite* awakens writers as well as readers.
  - *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

```
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

The reader/writer lock example illustrates *starvation*: 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.

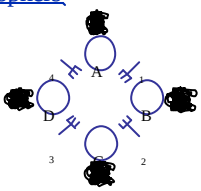
- Processes wait forever for each other to wake up and/or release resources.
- *Example:* ***traffic gridlock***.

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




```
while(true) {
    Think();
    AcquireForks();
    Eat();
    ReleaseForks();
}
```


Four Preconditions for Deadlock

Four conditions must be present for *deadlock* to occur:

1. *Non-preemptability*. 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*.

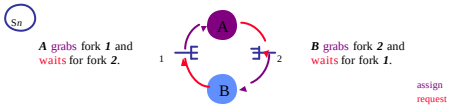
- Deadlock is easily seen with a *resource graph* or *wait-for graph.*

  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 iff the wait-for graph has at least one cycle.



A grabs fork ***1*** and waits for fork ***2***.

B grabs fork ***2*** and waits for fork ***1***.

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.

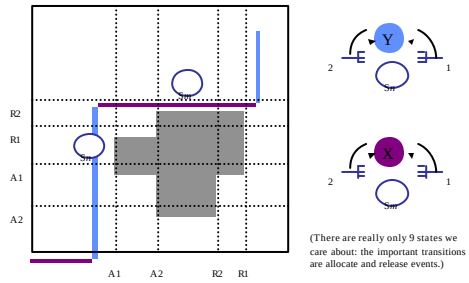
What is the probability that philosophers will deadlock?

- How does the probability change as:

  think times increase?

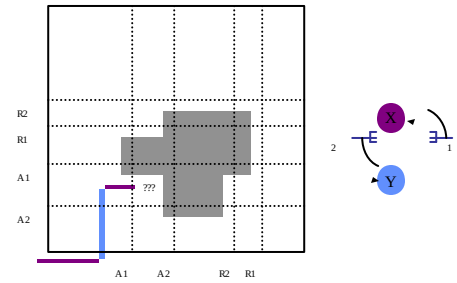
  number of philosophers increases?

RTG for Two Philosophers

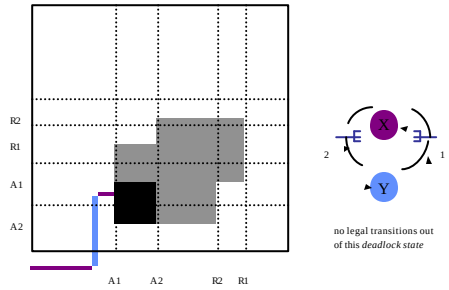


(There are really only 9 states we care about: the important transitions are allocate and release events.)

Two Philosophers Living Dangerously



The Inevitable Result



no legal transitions out of this *deadlock state*

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).
  - Cycles are necessary but not sufficient for deadlock.
  - For deadlock, each resource node with a request arc in the cycle must be fully allocated and unavailable.
- 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.
  - These "logical" producer/consumer resources can be considered to be available as long as the producer is still active.
    - Of course, the producer may not produce as expected.

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

```
class Lock {
        int held;
}

void Lock::Acquire() {
        while (held);          "busy-wait" for lock holder to release
        held = 1;
}

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


Spinlocks: What Went Wrong

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

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

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

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 **sleep** 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
  - a sequence of instructions that executes "all or nothing"
- Two solutions:
  - (1) hardware support: *atomic instructions* (**test-and-set**)
  - (2) scheduler control: *disable timeslicing* (**disable interrupts**)

Atomic Instructions: Test-and-Set

Problem: interleaved load/test/store.

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

```
Spinlock ::Acquire () {
      while(held);
      held = 1;
}

Wrong
      load   4(SP), R2          ; load "this"
busywait:
      load   4(R2), R3          ; load "held" flag
      bnz    R3, busywait       ; spin 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
      bnz    R3,busywait        ; spin if held wasn't zero
```


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

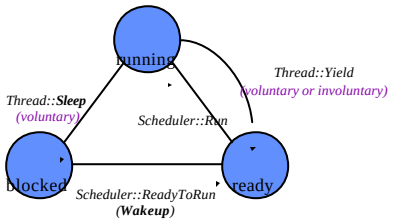
```
class Lock {
}

void Lock::Acquire() {
        disable interrupts;
}

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

Problems?

Implementing Mutexes: Rough Sketch

```
class Lock {
        int held;
        Thread* waiting;
}

void Lock::Acquire() {
        if (held) {
                waiting = currentThread;
                currentThread->Sleep();
        }
        held = 1;
}

void Lock::Release() {
        held = 0;
        if (waiting)            /* somebody's waiting: wake up */
                scheduler->ReadyToRun(waiting);
}
```


Nachos Thread States and Transitions

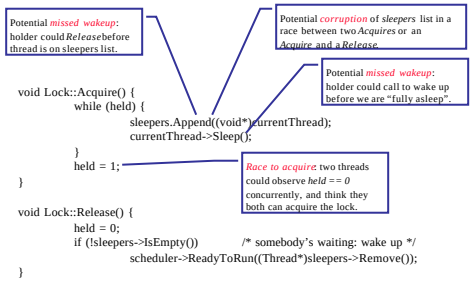


currentThread->Yield();
currentThread->Sleep();

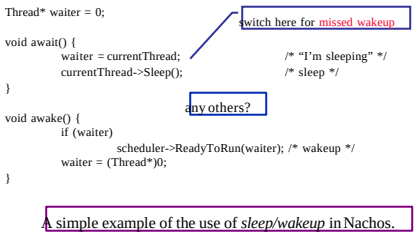
Implementing Mutexes: A First Cut

```
class Lock {
        int held;
        List sleepers;
}

void Lock::Acquire() {
        while (held) {                  Why the while loop?
                sleepers.Append((void*)currentThread);
                currentThread->Sleep();
        }
        held = 1;                       Is this safe?
}

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

Mutexes: What Went Wrong

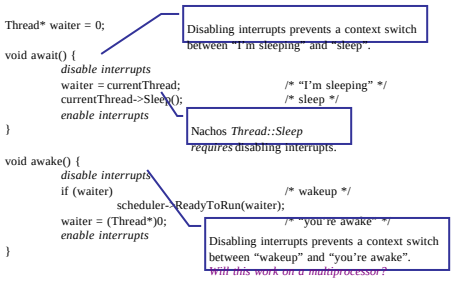


The Trouble with Sleep/Wakeup



A simple example of the use of *sleep/wakeup* in Nachos.

Using Sleep/Wakeup Safely



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 *(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

Races: A New Definition

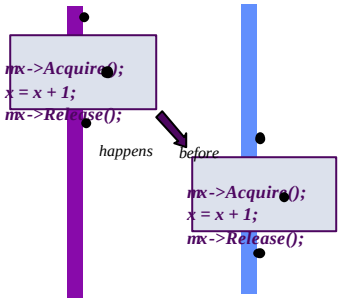
A program *P*'s *Acquire* events impose a partial order on memory accesses for each execution of *P*.

- Memory access event x_1 *happens-before* x_2 iff the synchronization orders x_1 before x_2 in that execution.
- If neither x_1 nor x_2 *happens-before* the other in that execution, then x_1 and x_2 are *concurrent*.

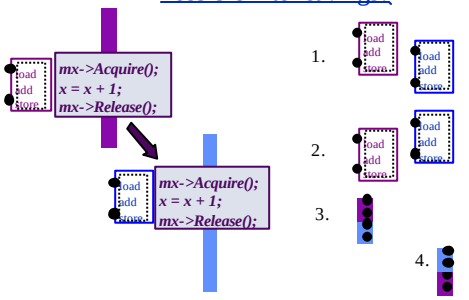
P has a *race* iff there exists some execution of *P* containing accesses x_1 and x_2 such that:

- Accesses x_1 and x_2 are *conflicting*.
- Accesses x_1 and x_2 are *concurrent*.

Locks and Ordering



Possible Interleavings?



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

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 lot 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?

Race Detection Alternatives

1. Static race detection for programs using monitors
   - Performance? Accuracy? Generality?
2. Dynamic data race detection using *happened-before*.
   - Instrument program to observe accesses.
     What other events must Eraser observe?
   - Maintain *happened-before* relation on accesses.
   - If you observe concurrent conflicting accesses, scream.
   - Performance? Accuracy? Generality?

Basic Lockset Algorithm

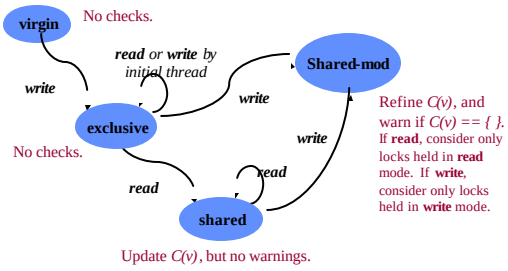
1. Premise: each shared *v* is covered by exactly one lock.
2. Which one is it? Refine "candidate" lockset for each *v*.
3. If *P* executes a set of accesses to *v*, and no lock is common to all of them, then (1) is false.

For each variable ***v*, *C(v)* = *{all locks}***
When thread ***t*** accesses ***v*:**
C(v)* = *C(v)* ∩ *locks_held(t);
if ***C(v)* == *{ }*** then ***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



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 *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

```
semapohore->Init(1);

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.

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

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

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


Ping-Pong with One Semaphore?

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

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


Ping-Pong with One Semaphore?

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

void
PingPong() {
    while(not 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

Another Example With Dual Semaphores

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

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


Basic Barrier

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

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


How About This? (#1)

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

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


How About This? (#2)

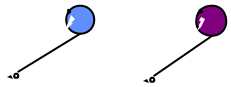
```
blue->Init(1);
purple->Init(0);

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

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

How About This? (#3)

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

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

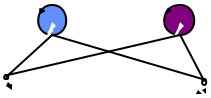
```
void CallThat() {
        purple->P();
        Compute();
        blue->V();
}
```



How About This? (#4)

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

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

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



Basic Producer/Consumer

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

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

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

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