CPS 310
Starvation and Deadlock

Jeff Chase

Duke University

An Introduction to Programming with C# Threads

Andrew D. Birrell

This paper provides an introduction to writing concurrent programs with "threads". A threads facility allows you to write programs with multiple simultaneous points of execution, synchronizing through shared memory. The paper describes the basic thread and synchronization primitives, then for each primitive provides a tutorial on how to use

```
Thread t = new Thread(new ThreadStart(foo.A));
t.Start();
foo.B();
t.Join();
```

This paper is **recommended reading** if you want to see thread programming through the eyes of one of the great "village elders" of computer systems. He wrote it for you. We present the SharedLock example from this paper.

2003

ACM NEWS

In Memoriam: Andrew Birrell 1951-2016

By Lawrence M. Fisher
December 9, 2016
Comments

VIEW AS: SHARE:

Andrew Birrell in 2001.

Credit: Eleanor Birrell

Andrew Birrell, a computer scientist specializing in operating and distributed systems, who is credited with designing the first remote procedure call (RPC) system and the Grapevine distributed email system, passed away December 7 after a lengthy illness.

Born in Scotland, Birrell attending Trinity College, Cambridge, where he earned his doctorate in 1978 under Roger Needham for his work on the CAP Filing System and system programming in a high-level language.

Birrell came to the U.S. in 1978 to take the first of several positions at research laboratories that defined the course of his career: at the Palo Alto Research Center Inc. (Xerox PARC), where he worked through 1984. He then moved to Digital Equipment Corp. (DEC, later bought out by Compaq), where he spent 17 years in that organization's Systems Research Center in Palo Alto. In 2001, He moved on to Microsoft Research Silicon Valley in Mountain View, CA, where he remained until it closed down in 2014.

SharedLock: reader/writer lock

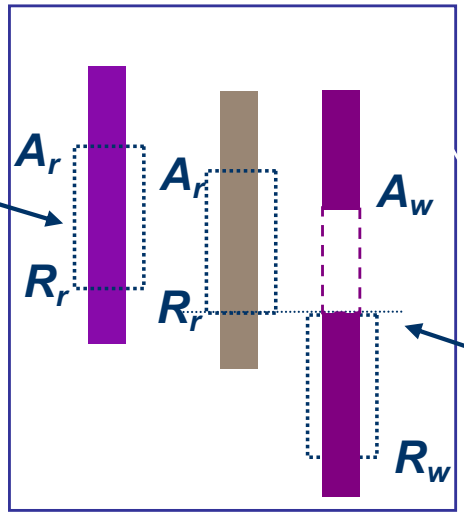
- A **reader/write lock** or **SharedLock** is a new kind of "lock" that is similar to our old definition:
  - Acquire and Release primitives
  - Mutual exclusion for writes to shared state
- But: a **SharedLock** allows more concurrency for readers when no writer is present.

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

Reader/Writer Lock illustrated

Multiple readers may hold the lock concurrently in **shared** mode.

With monitors—mutexes and conditions—as a foundation, we can build higher-level abstractions like SharedLock.



Writers always hold the lock in **exclusive** mode. They must wait for all readers or competing writer to exit.

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

Writer mode is a **mutex**: at most one holder at any given time; allows writes.

SharedLock: outline

```
int i;      /* # active readers, or -1 if writer */

void AcquireWrite() {

        while (i != 0)
                sleep….;
        i = -1;

}
void AcquireRead() {

        while (i < 0)
                sleep…;
        i += 1;

}
```

```
void ReleaseWrite() {

        i = 0;
        wakeup….;

}

void ReleaseRead() {

        i -= 1;
        if (i == 0)
                wakeup…;

}
```

SharedLock: lock it down

```
int i;       /* # active readers, or -1 if writer */
Lock rwMx;

AcquireWrite() {
        rwMx.Acquire();
        while (i != 0)
                sleep…;
        i = -1;
        rwMx.Release();
}
AcquireRead() {
        rwMx.Acquire();
        while (i < 0)
                sleep…;
        i += 1;
       rwMx.Release();
}
```

```
ReleaseWrite() {
        rwMx.Acquire();
        i = 0;
        wakeup…;
        rwMx.Release();
}

ReleaseRead() {
        rwMx.Acquire();
        i -= 1;
        if (i == 0)
                wakeup…;
        rwMx.Release();
}
```

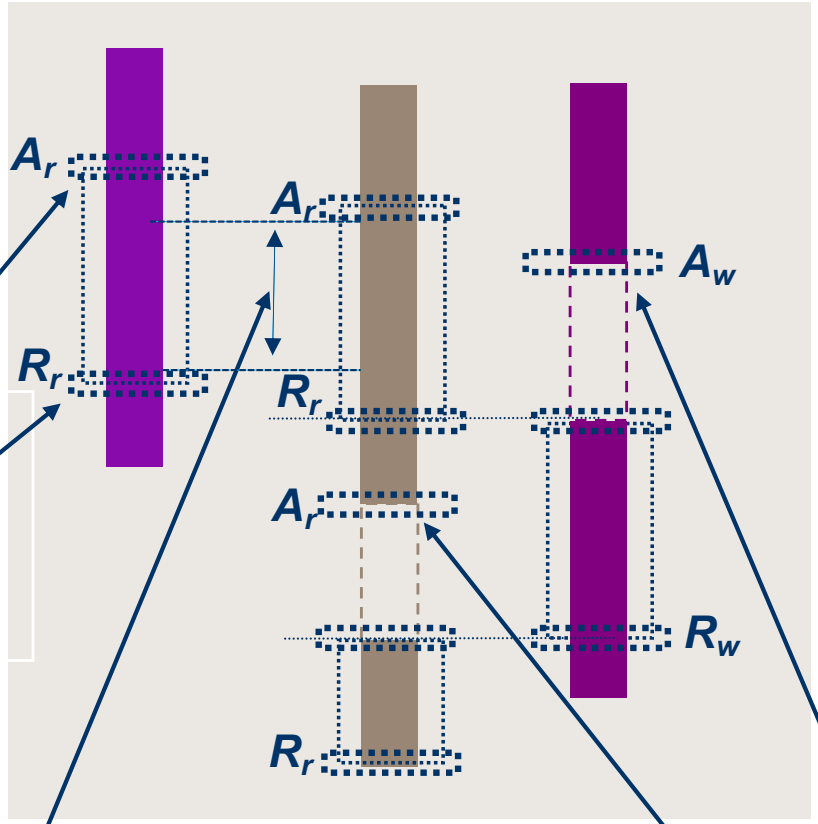
SharedLock: cleaner syntax

```
int i;       /* # active readers, or -1 if writer */

synchronized AcquireWrite() {
        while (i != 0)
                wait();
        i = -1;
}
synchronized AcquireRead() {
        while (i < 0)
                wait();
        i += 1;
}
```

```
synchronized ReleaseWrite() {
        i = 0;
        notifyAll();
}

synchronized ReleaseRead() {
        i -= 1;
        if (i == 0)
                notify();
}
```

We can use Java syntax for convenience.
That's the beauty of **pseudocode**. We use any convenient syntax.
These syntactic variants have the same meaning.

The Little Mutex Inside SharedLock



Caller thread holds mutex only while acquiring or releasing SharedLock .

SharedLock holders do not hold its mutex, permitting concurrent readers.

To block in Acquire*, use condition wait(), releasing mutex.

Limitations of the SharedLock first try

This implementation has weaknesses; see [Birrell89/03].

- Spurious lock conflicts (on a multiprocessor): multiple waiters contend for the mutex after a signal or broadcast.

  Solution: drop the mutex before signaling (if permitted).

- Spurious wakeups (thundering herd)

  ReleaseWrite awakens writers as well as readers.

  Solution: add a separate condition variable for writers.

- Starvation

  How can we be sure that a writer can *ever* acquire if faced with a continuous stream of arriving readers?

Culling the thundering herd

The first try at SharedLock illustrates thundering herd again. It is slow under contention: too many context switches.

Why?

- ReleaseWrite uses **notifyAll** to wake all waiting readers.
  - Maximizes concurrency for readers: they all get in.
- But it uses the same CV for readers and writers.
- And if a waking writer gets in first, then all other threads must wait() again: thundering herd.
- For them, the broadcast wakeup is “spurious”.

How to fix it?

- Use **two** CVs to separate readers and writers.

SharedLock: 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();
}
```

SharedLock: second try

```
synchronized AcquireWrite() {
        while (i != 0)
                wCv.Wait();
        i = -1;
}

synchronized AcquireRead() {
        while (i < 0) {
                readersWaiting+=1;
                rCv.Wait();
                readersWaiting-=1;
        }
        i += 1;
}
```

```
synchronized ReleaseWrite() {
        i = 0;
        if (readersWaiting)
                rCv.Broadcast();
        else
                wCv.Signal();
}
synchronized ReleaseRead() {
        i -= 1;
        if (i == 0)
                wCv.Signal();
}
```

wCv and rCv are protected by the monitor mutex: pseudocode.

Starvation

- These reader/writer locks also illustrate starvation: under load, a writer might 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…
- Solution: some reader must politely stop before entering, even though it is not forced to wait by oncoming traffic.
  - More code…
  - More complexity…


ONE LANE
BRIDGE
CR-V
HONDA
BXG-2024

Third try: writer priority

```
synchronized AcquireW() {
        while (i != 0) {
                cw++;
                wCv.Wait();
                cw--;
        }
        i -= 1;      /* -1 */
}

synchronized AcquireR() {
        while (i < 0 || cw) {
                rCv.Wait();
        }
        i += 1;
}
```

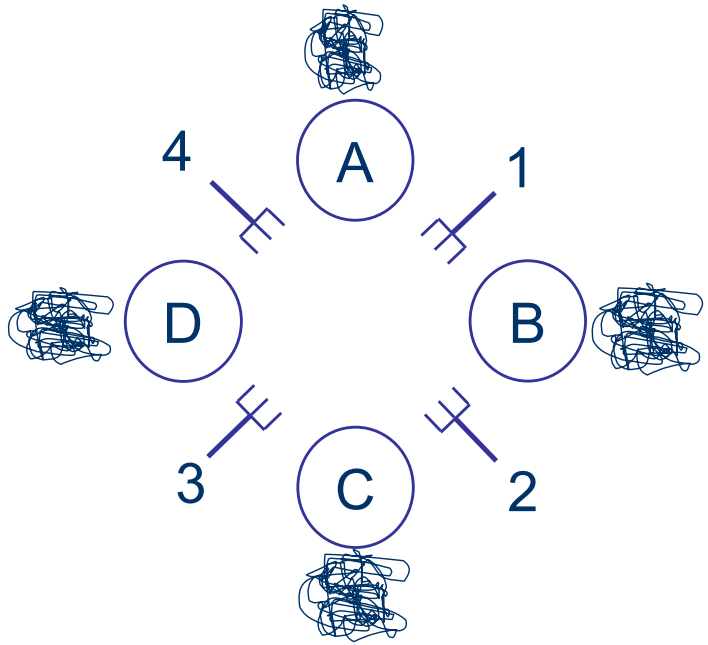
```
synchronized ReleaseW() {
        i += 1;   /* 0 */
        if (cw)
                wCv.Signal();
        else
                rCv.Broadcast();
}

synchronized ReleaseR() {
        i -= 1;
        if (i == 0 && cw)
                wCv.Signal();
}
```

Starvation in SharedLock

- The first/second try may starve the writers.
- **Why**? Because if a reader is in, then any arriving reader can just jump right in with it.
- Writers must wait for the last reader to exit.
- The third try adds logic to block any arriving reader if any writer is waiting. And the writers get priority for a signal.
- **Fixed it for you!** But now the readers can starve.
- We can fix that problem too, with more code.
- See “Highway 110-310” in list of concurrency problems.

Dining Philosophers

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

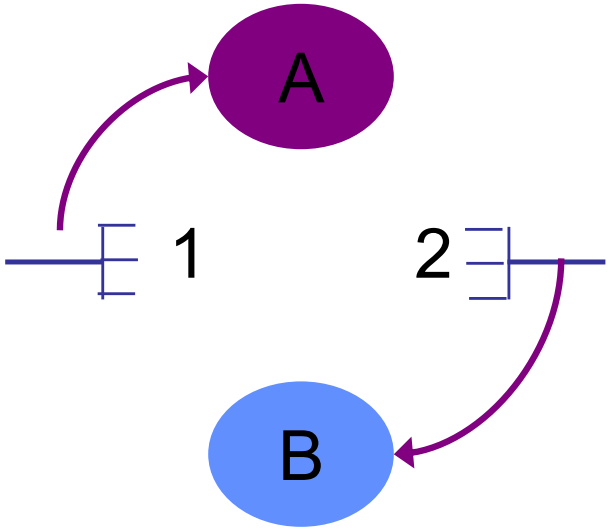



```
while(true) {
        Think();
        AcquireForks();
        Eat();
        ReleaseForks();
}
```

Resource graph or wait-for graph

- A vertex for each process and each resource
- If process *A* holds resource *R*, add an arc from *R* to *A*.

A grabs fork ***1***

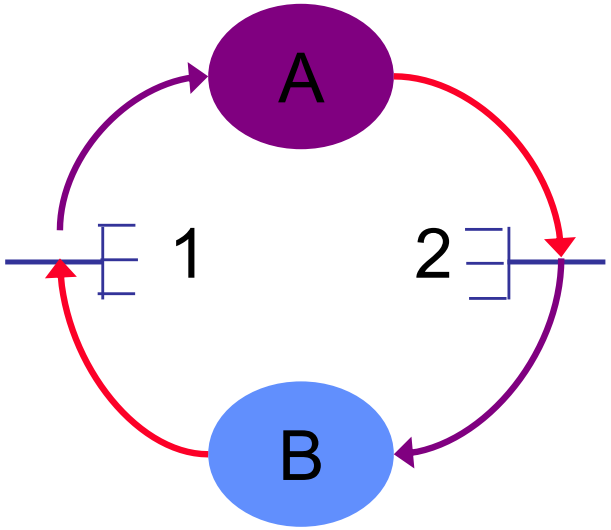


B grabs fork ***2***

Resource graph or wait-for graph

- 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 *R*, add an arc from *A* to *R*.

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



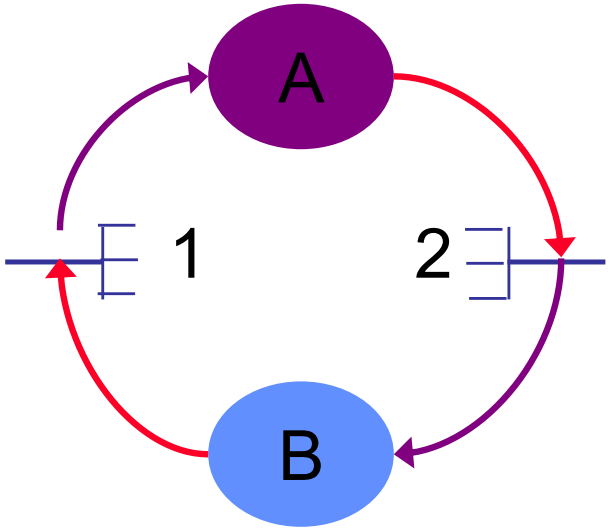
B grabs fork ***2***
and
waits for fork ***1***.

Resource graph or wait-for graph

- 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 *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***.

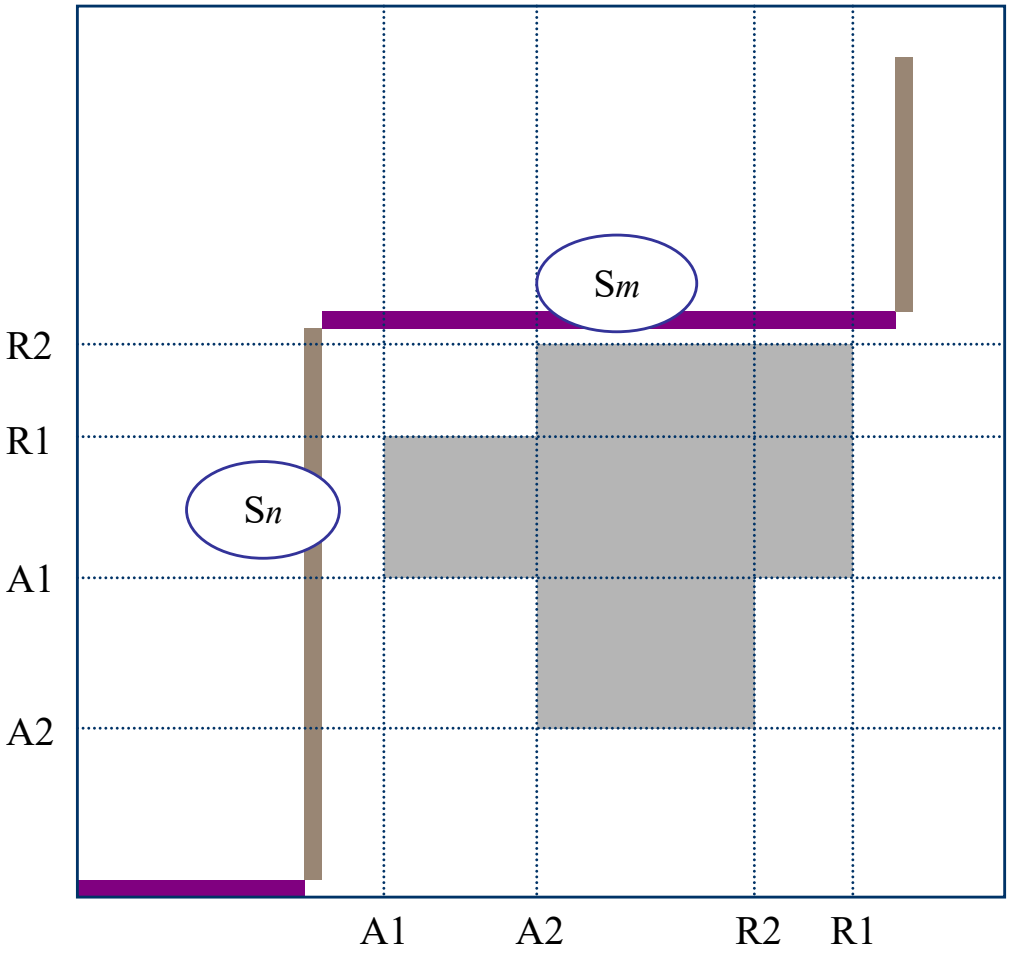


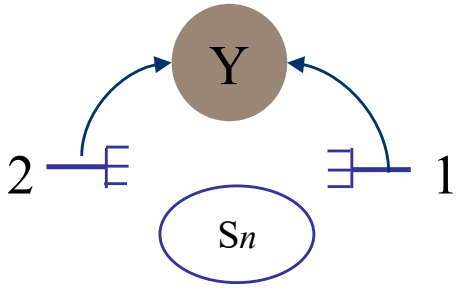
B grabs fork ***2*** and waits for fork ***1***.

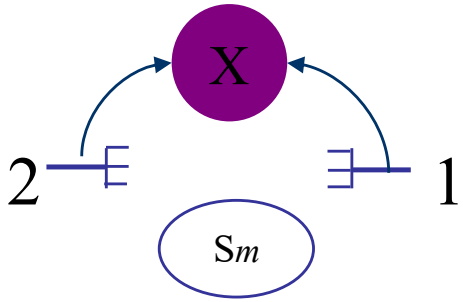
Deadlock vs. starvation

- A deadlock is a situation in which a set of threads are all waiting for another thread to move.
- But none of the threads can move because they are all waiting for another thread to do it.
- Deadlocked threads sleep “forever”: the software “freezes”. It stops executing, stops taking input, stops generating output. There is no way out.
- Starvation (also called livelock) is different: some schedule exists that can exit the livelock state, and the scheduler may select it, even if the probability is low.

RTG for Two Philosophers

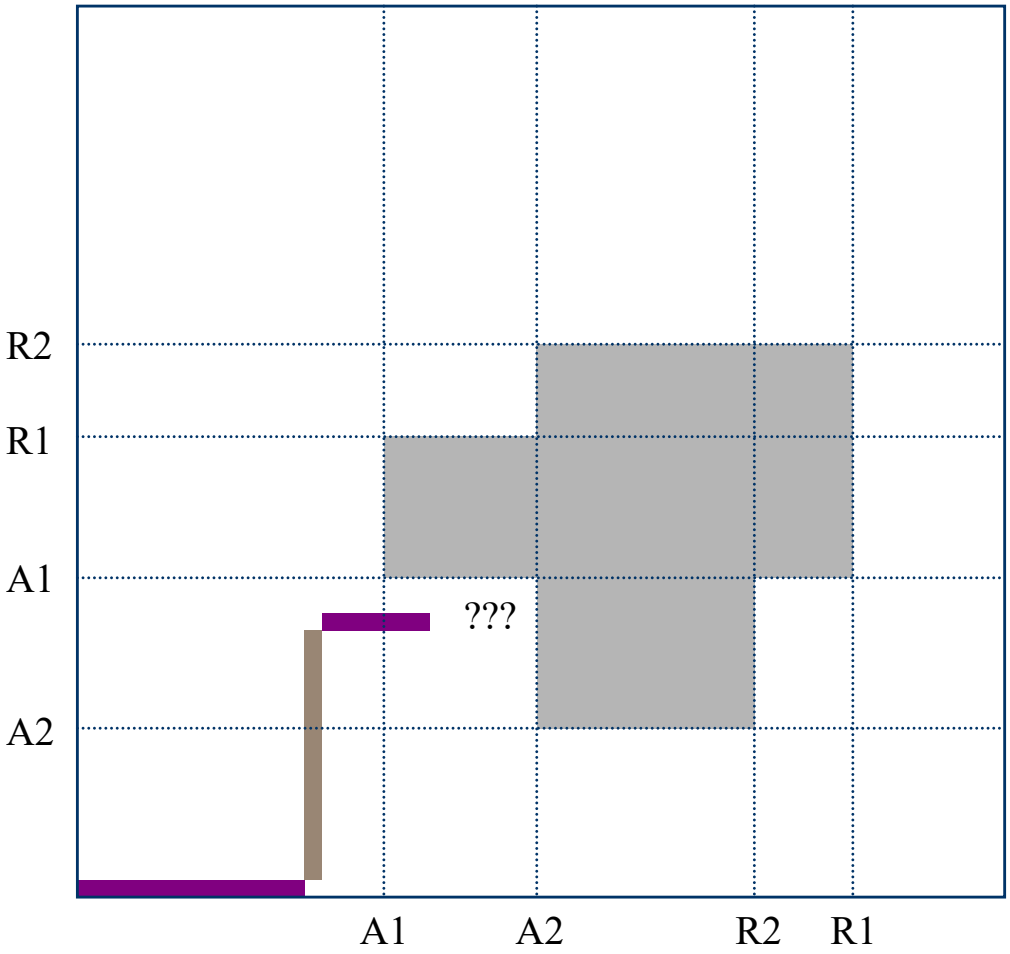


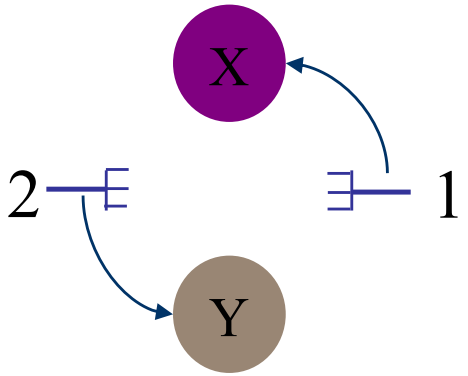




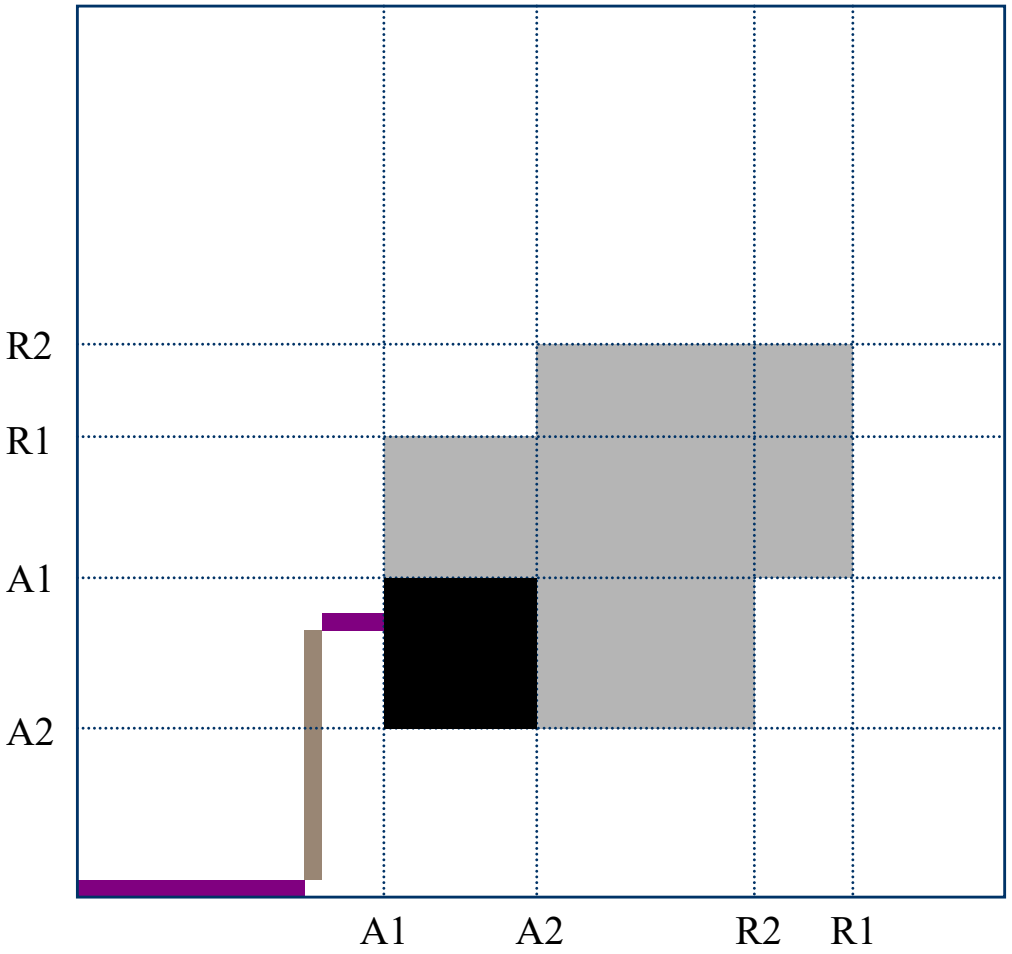
(There are really only 9 states we care about: the key transitions are **acquire** and **release** events.)

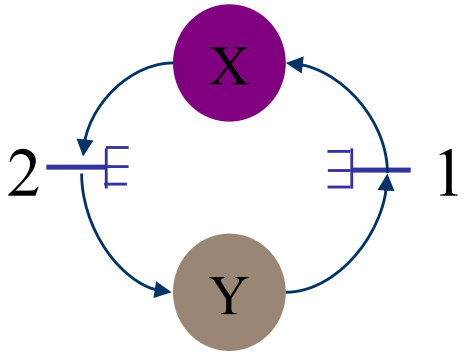
Two Philosophers Living Dangerously





The Inevitable Result





This is a deadlock state: There are no legal transitions out of it.

Four conditions for deadlock

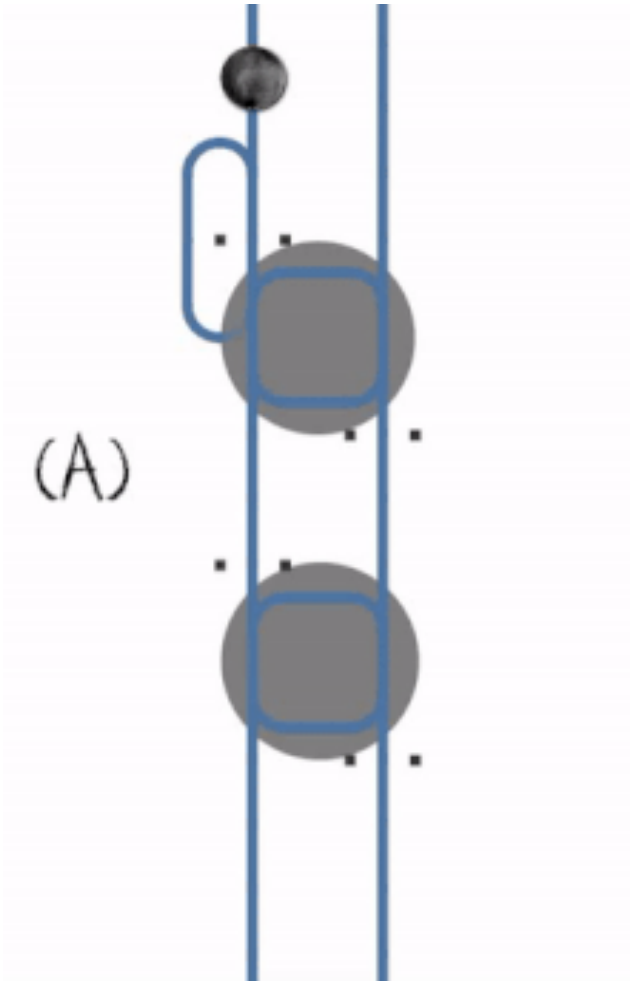
Four conditions must be present for deadlock to occur:

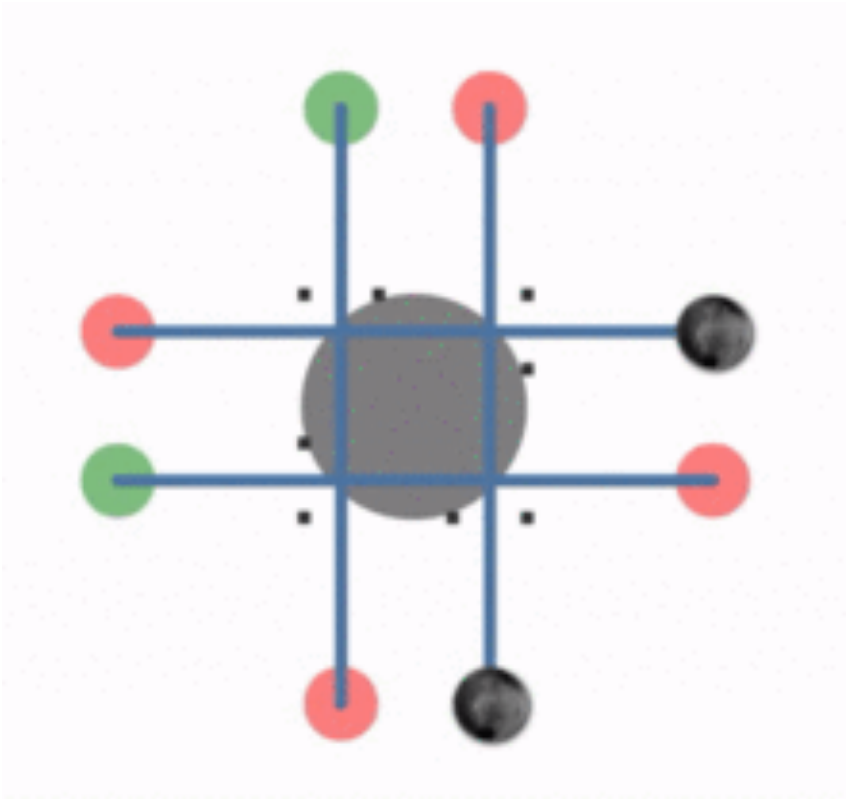
1. Non-preemption of ownership. Resources are never taken away from the holder.
2. Exclusion. A resource has at most one holder.
3. Hold-and-wait. Holder blocks to wait for another resource to become available.
4. Circular waiting. Threads acquire resources in different orders.

Not all schedules lead to collisions

- The scheduler+machine choose a schedule, i.e., a trajectory or path through the graph.
  - Synchronization constrains the schedule to avoid illegal states.
  - Some paths “just happen” to dodge dangerous states as well.
- What is the probability of deadlock?
  - How does the probability change as:
    - think times increase?
    - number of philosophers increases?

Wikipedia animations





Dealing with deadlock

1. Ignore it. Do you feel lucky?
2. Detect and recover. Check for cycles and break them by restarting activities (e.g., killing threads).
3. Prevent it. Break any precondition.
   - Keep it simple. Avoid blocking with any lock held.
   - Acquire nested locks in some predetermined order.
   - Acquire resources in advance of need; release all to retry.
   - Avoid “surprise blocking” at lower layers of your program.
4. Avoid it.
   - Deadlock can occur by allocating variable-size resource chunks from bounded pools: google “Banker’s algorithm”.