Today

- Classic Synchronization Problem: Dining Philosophers
- Synchronization Mechanisms - tradeoffs

Review

- We looked at the producer-consumer problem at length
  - What were our two solutions?
    - One with semaphores
    - One with condition variables

Review: Producer-Consumer Code

sodaLock = new Lock(); hasSoda = new CV(); hasRoom = new CV();

```
consumer () {
  sodaLock.acquire()

  while (numSodas == 0) {
    hasSoda.wait(sodaLock)
  }   CV1             Mx

  numSodas--;

  hasRoom.signal()
    CV2

  sodaLock.release()
}
```

```
producer () {
  sodaLock.acquire()

  while(numSodas==MaxSodas){
    hasRoom.wait(sodaLock)
  }   CV2             Mx

  numSodas++;

  hasSoda.signal()
    CV1

  sodaLock.release()
}
```

Requires one lock and two condition variables

Review: Producer-Consumer with Semaphores and Mutex

```
Semaphore mutex(1), fullSlots(0), emptySlots(MaxSodas)
```

```
producer () {
  // wait for empty slot
  emptySlots.P()
  // lock shared state
  mutex.P()
  put one soda in
  mutex.V()

  // signal item arrival
  fullSlots.V()
}
```

```
consumer () {
  // wait for item arrival
  fullSlots.P()
  // lock shared state
  mutex.P()
  take one soda out
  mutex.V()

  //signal empty slot
  emptySlots.V()
}
```

Does this work with multiple consumers and/or producers?

Yes!...

Analysis: Producer-Consumer with Semaphores and Mutex

```
Semaphore mutex(1), fullSlots(1), emptySlots(MaxSodas-1)
```

```
producer () {
  // wait for empty slot
  emptySlots.P()
  // lock shared state
  mutex.P()
  put one soda in
  mutex.V()

  // signal item arrival
  fullSlots.V()
}
```

```
consumer () {
  // wait for item arrival
  fullSlots.P()
  // lock shared state
  mutex.P()
  take one soda out
  mutex.V()

  //signal empty slot
  emptySlots.V()
}
```

What if 1 full slot and multiple consumers call down?
Only one will see semaphore at 1, rest see at 0.

Review: Basic Producer/Consumer

```
empty = Semaphore(1);
full = Semaphore(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**
  - sum of the values is always 1
- **It is the same as** p... producer and con... alternation

Why don't we need a lock in this solution?

Can't both be in the critical section because of the limit of only one resource.

Classical Problem: intellectually interesting, low practical utility

DINING PHILOSPHERS

Dining Philosophers Problem

- N processes share N resources
- Resource requests occur in pairs w/ random think times
- Hungry philosopher grabs *right* chopstick
  - and doesn't let go…
  - until the other chopstick is free
  - and the rice is eaten

```
while(true) {
    think();
    getChopsticks();
    eat();
    putChopsticks();
}
```

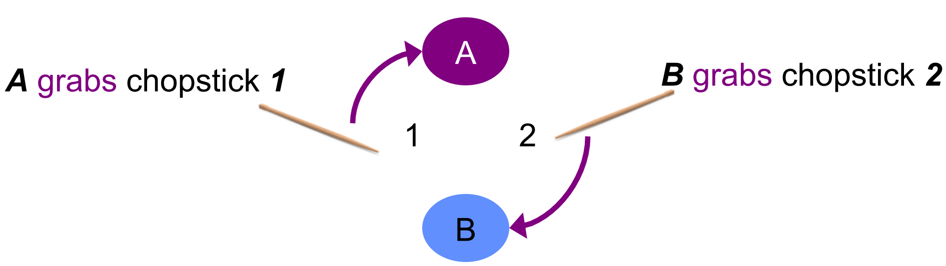
What is shared?
What are the ordering constraints?

What happens in the case of 5 philosophers?
What if fewer or more philosophers?
What are your goals for a solution?

Observations?

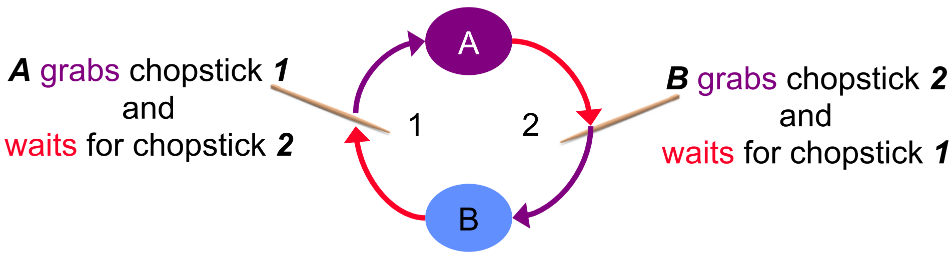
Resource Graph or Wait-for Graph

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



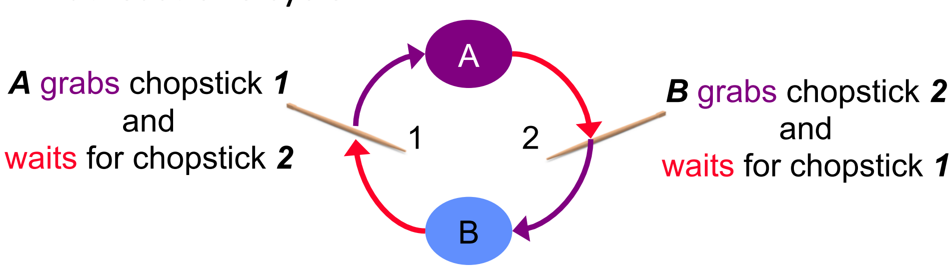
Resource Graph or Wait-for Graph

- 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



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.



How does this help us think about dining philosophers?

Possible Solutions to Dining Philosophers

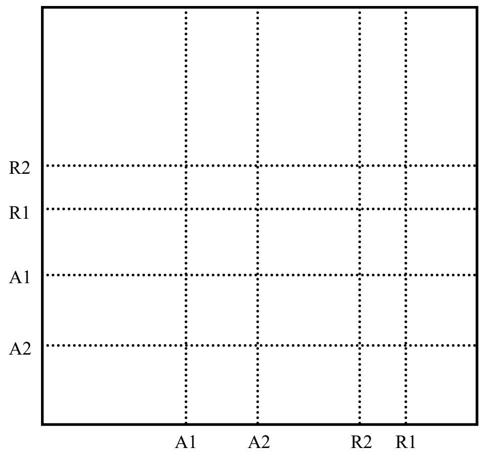
- Asymmetric solution
  - Some pick up left chopstick first, some pick up right
    - How does that play out?
- Don't pick up either chopstick until *both* are free
  - How would you implement this?
- Allow a philosopher to take a chopstick from another philosopher who isn't yet eating
- Not ideal
  - Reduce the number of philosophers or increase the number of resources

Still issues with starvation--
Need guarantee of locks being acquired in order

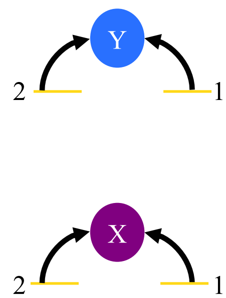
Deadlock vs. starvation

- 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

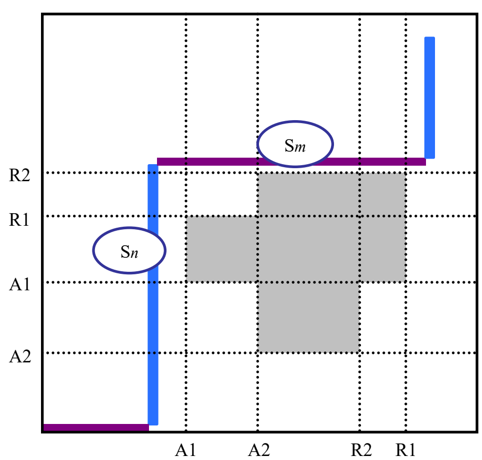


Philosophers X and Y

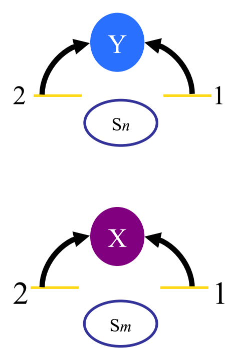


Synchronization: acquiring and releasing locks for each chopstick (1 and 2)

RTG for Two Philosophers

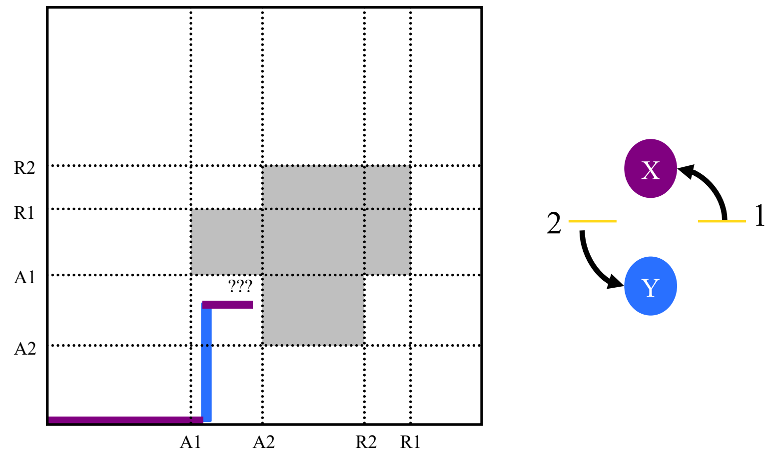


Philosophers X and Y

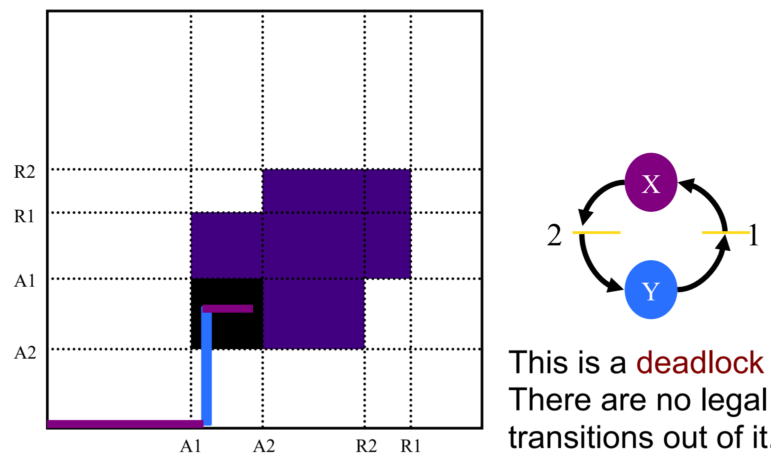


There are really only 9 states we care about: the key transitions are **acquire** and **release** events.

Two Philosophers Living Dangerously



The Inevitable Result



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

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
- How likely is deadlock to occur as:
  - think times increase?
  - number of philosophers and number of resources (value of N) increases?

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".

Guidelines for Lock Granularity

- Keep critical sections short. Push "non-critical" statements outside to reduce contention.
- Limit lock overhead. Keep to a minimum the number of times mutexes are acquired and released.
  - Note tradeoff between contention and lock overhead.
- 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!"

Looking Ahead

- Synchronization Assignment – Due Monday
  - Part 1: Discussion/pseudocode
  - Part 2: implementation in Java