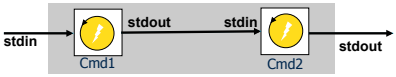
Today

- Concurrency Problems
  - Dining Philosophers
- Deadlock

Review: Pipes

```
read(pipe, buf, N) {
   pipeMx.lock();
   for (int i = 0; i<N; i++) {
      while (no bytes in pipe)
         dataCv.wait();
      move one byte from pipe
      into buf[i];
      spaceCV.signal();
   }
   pipeMx.unlock();
}
```

- Read N bytes from the pipe into the user buffer named by **buf**.
- Think of this code as deep inside the **read** system call implementation on a pipe.
- The **write** implementation is similar.



Dining Philosophers Problem

- 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 rice is eaten

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

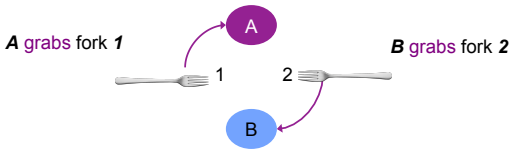
What is shared?
What are the ordering constraints?

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

Our Observations

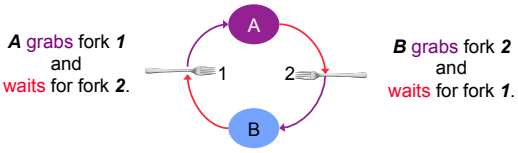
Resource Graph or Wait-for Graph

- 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

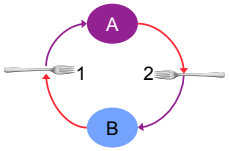
- A vertex for each process and each resource
- If process A holds resource R, add an arc from R to A.
- If process A is waiting for R, add an arc from A to R.



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.

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

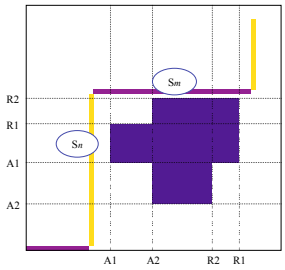


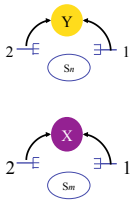
B grabs fork ***2*** and waits for fork ***1***.

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.

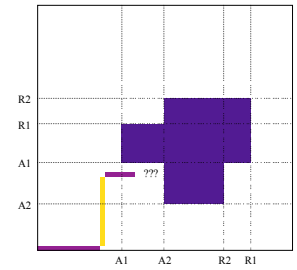
RTG for Two Philosophers

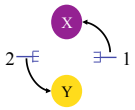




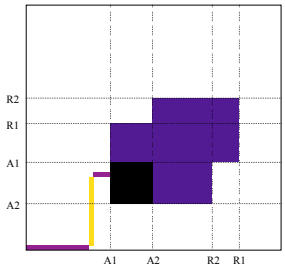
There are really only 9 states we care about: the key transitions are **acquire** and **release** events.

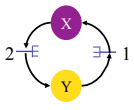
Two Philosophers Living Dangerously





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 likely is deadlock to occur:
    - 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 "noncritical" 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

- Project 3