Deadlock

CS 241

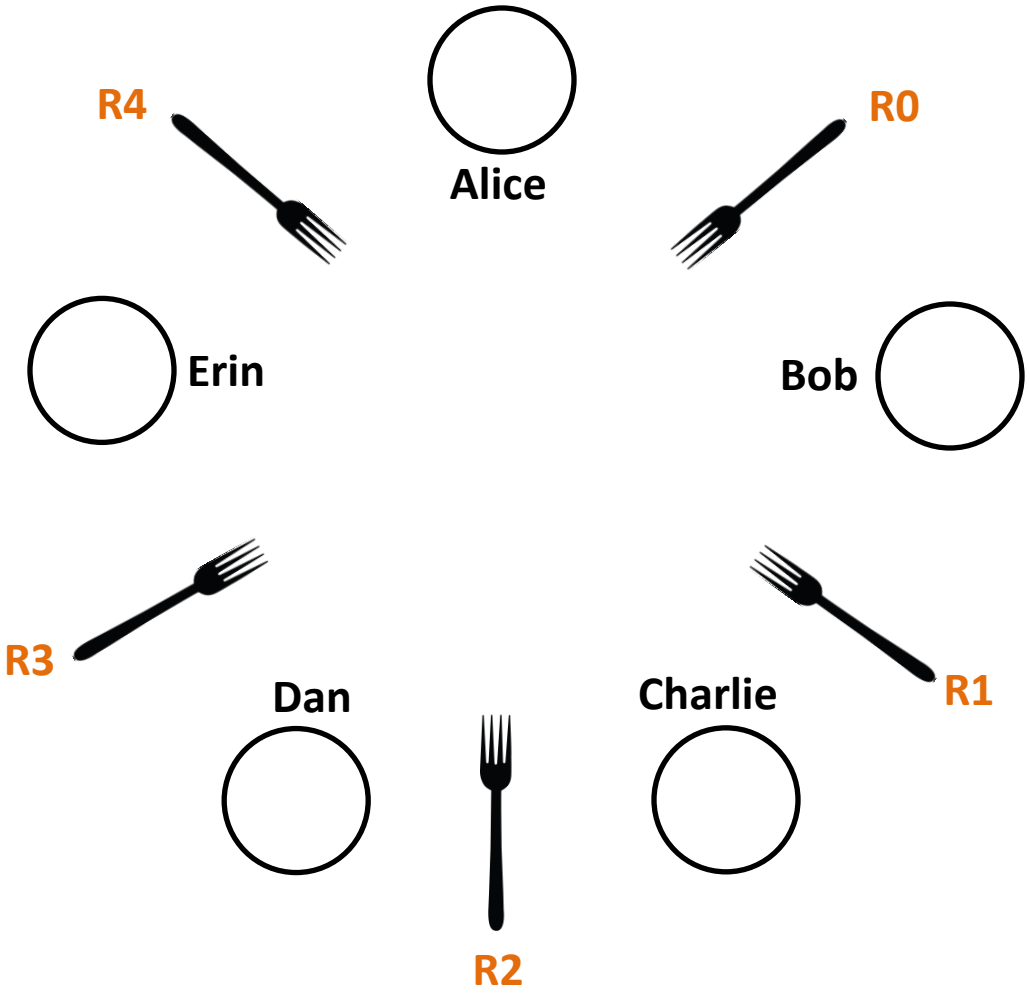
Oct. 23, 2013

Review: Synchronization

- Three synchronization primitives:

- Four classical problems:

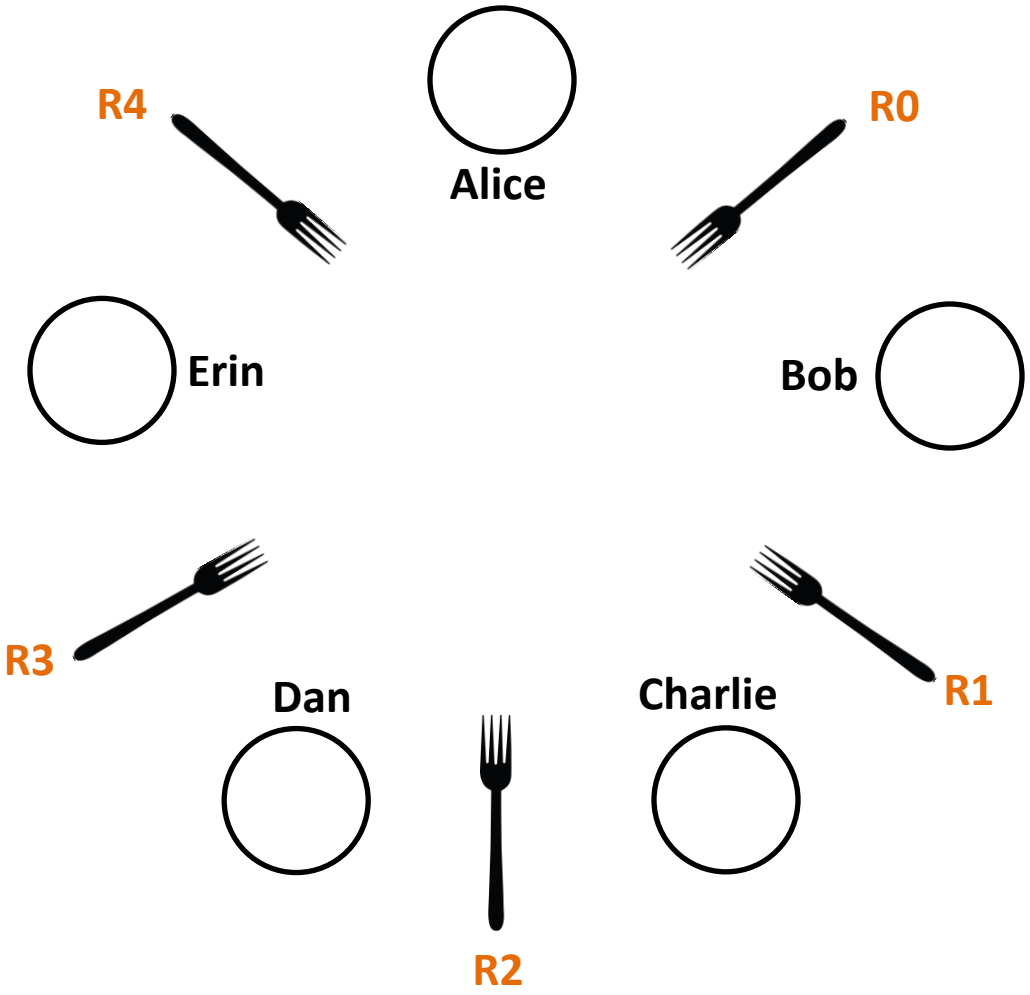
Dinning Philosophers Problem



Dinning Philosophers Code

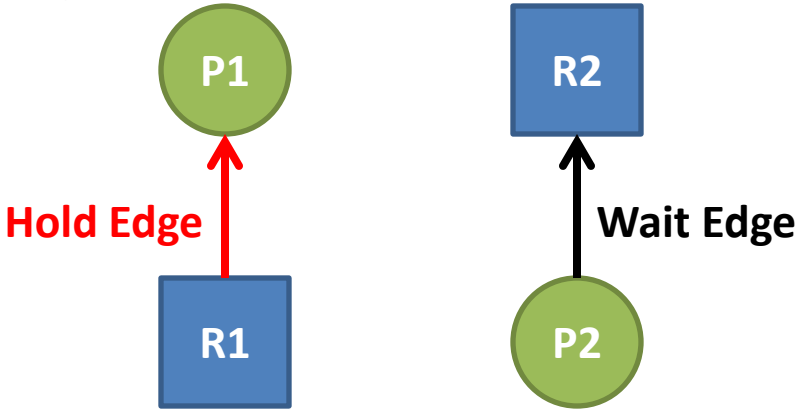
```
void philosopher(int id) {
  /* id is in the range of [0, MAX-1] */
  while (1) {
    think();

    pthread_mutex_lock( R[id] );
    pthread_mutex_lock( R[(id-1) % MAX] );
    eat();
    pthread_mutex_unlock( R[(id-1) % MAX] );
    pthread_mutex_unlock( R[id] );
  }
}
```

Dinning Philosophers Problem



Formalizing Circular Wait

- **Resource Allocation Graph**:
  - Circle: Process
  - Square: Resources



  - Edges:
    - Process ➔ Resource: A process requests a resource
    - Resource ➔ Process: A resource is held by a process

- **Wait-For Graph**: Reduce the graph to only processes, preserving edges when possible

Dinning Philosophers Code

```
  Alice: lock(r0);
    Bob: lock(r1);
Charlie: lock(r2);
    Dan: lock(r3);
   Erin: lock(r4);
  Alice: lock(r4);
    Bob: lock(r0);
Charlie: lock(r1);
    Dan: lock(r2);
   Erin: lock(r3);
  Alice: eat();
    Bob: eat();
       ...
```

Producer-Consumer Solution

```
P1: lock(r2);
    lock(r1);
    do_work();
    unlock(r1);
    unlock(r2);

P2: lock(r2);
    lock(r1);
    do_work();
    unlock(r1);
    unlock(r2);

P3: lock(r1);
    lock(r3);
    do_work();
    unlock(r3);
    unlock(r1);

P4: lock(r3);
    lock(r2);
    do_work();
    unlock(r2);
    unlock(r3);
```

Circular Wait

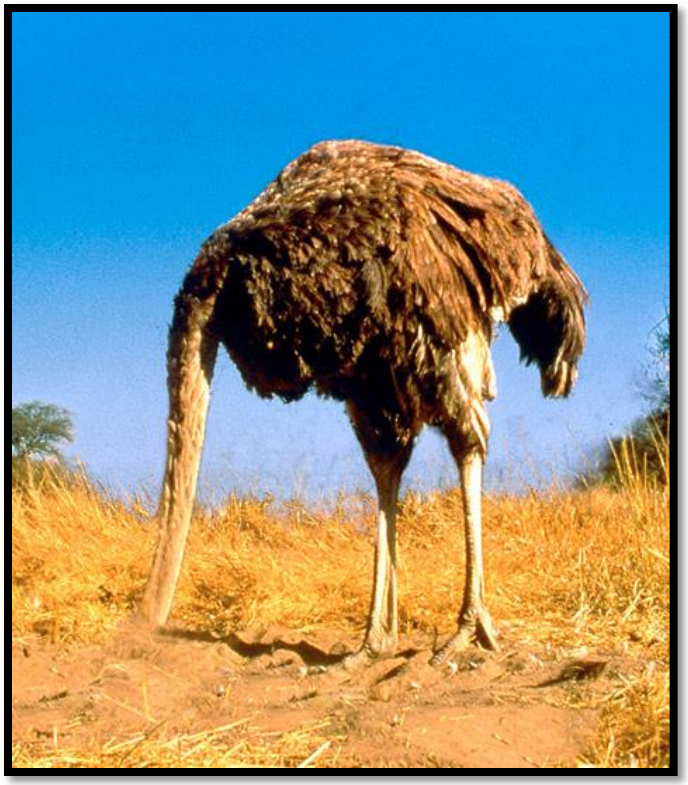
```
P1: lock(r2);
    P3: lock(r1);
     P4: lock(r3);
 P2: lock(r2);
 P2: lock(r1);
P1: lock(r1);
     P4: lock(r2);
    P3: lock(r3);
*: do_work();
*: unlock(*);
```

Circular Wait

```
 P2: lock(r2);
P1: lock(r2);
   P4: lock(r3);
 P2: lock(r1);
P1: lock(r1);
  P3: lock(r1);
   P4: lock(r2);
  P3: lock(r3);
*: do_work();
*: unlock(*);
```

Solving Deadlock

- Approach #1: “The Ostrich Approach”
  - Solving deadlock is hard.
    …so we won’t worry about it!



Solving Deadlock

- Three approaches to *actually* solve deadlock:
  -
  -
  -