Necessary conditions for deadlock

Deadlock only if they all hold

① Bounded resources

Acquire can block invoker

② No preemption

the resource is mine, MINE! (until I release it)

③ Wait while holding

holds one resource while waiting for another

④ Circular waiting

P_i waits for P_{i+1} and holds a resource requested by P_{i-1}

sufficient if one instance of each resource

Deadlock Prevention: Negate ①

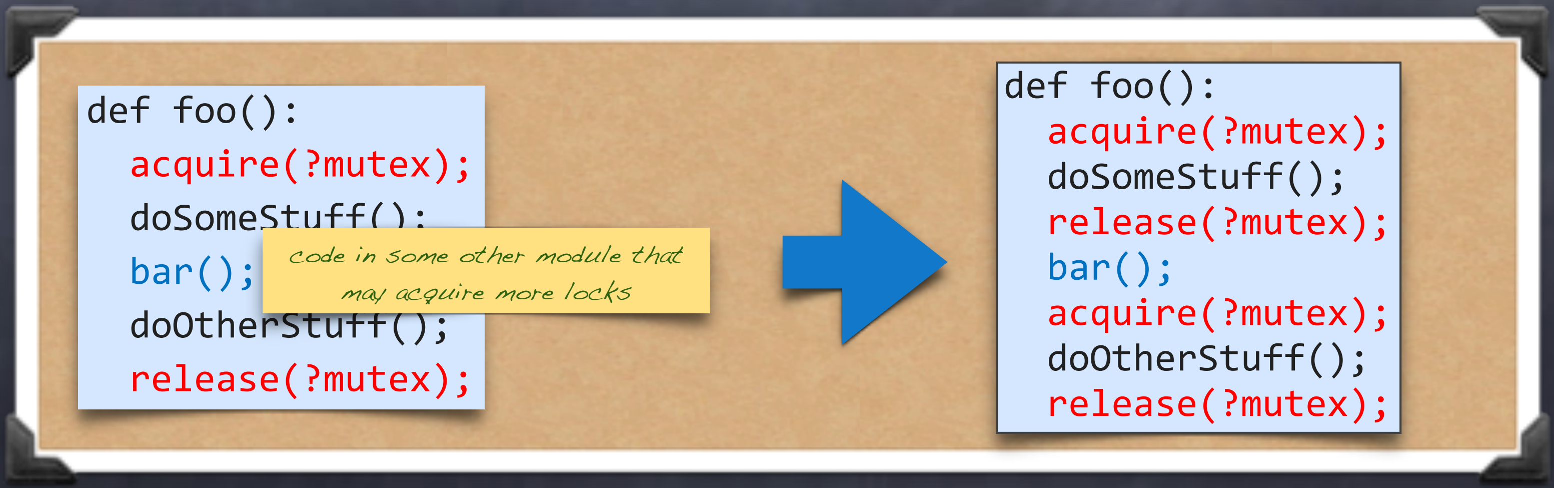
- Eliminate "Acquire can block invoker/bounded resources"
  - Make resources sharable without locks
    - Wait-free synchronization
    - The Harmony book (Chapter 23) has examples of non-blocking data structures
  - Have sufficient resources available, so acquire never delays (duh!)
    - E.g., use an unbounded queue, or make sure that queue is "large enough"

Deadlock Prevention: Negate ②

- Allow preemption
  - Requires mechanisms to save/restore resource state
    - multiplexing (registers, memory, etc). VS.
    - undo/redo (database transaction processing)
  - Allow OS to preempt resources of waiting processes
  - Allow OS to preempt resources of requesting processes

Deadlock Prevention: Negate ③

- Eliminate Hold & Wait
  - Don't hold resource while waiting for others
    - Rewrite code



```
def foo():
  acquire(?mutex);
  doSomeStuff();
  bar();
  doOtherStuff();
  release(?mutex);
```

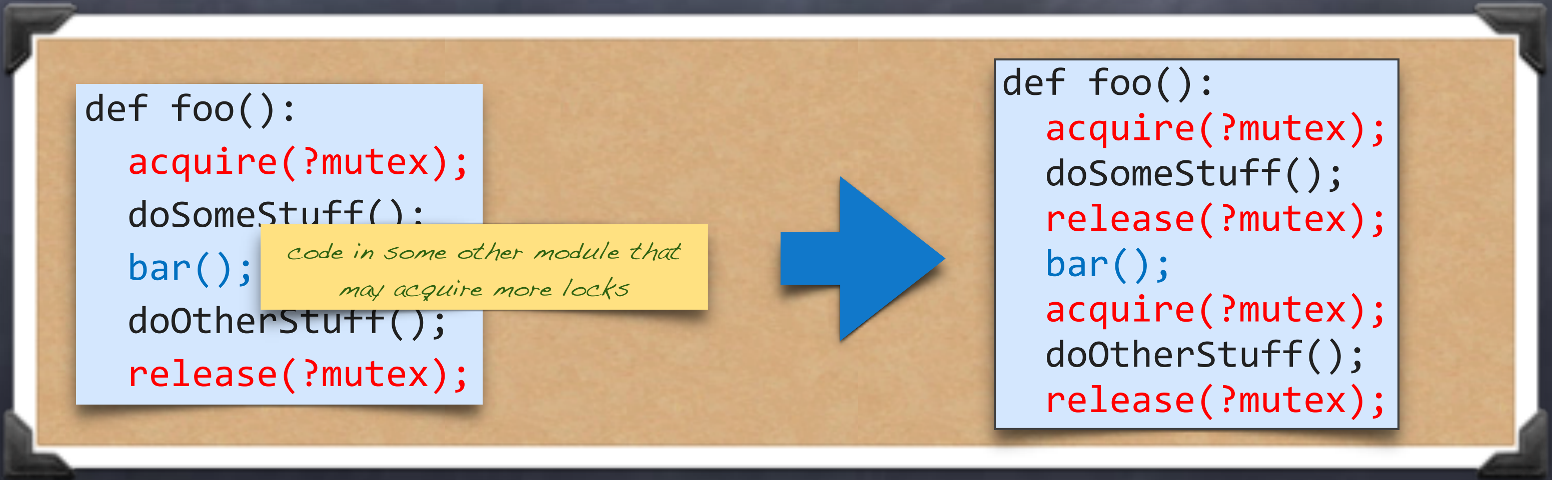
code in some other module that may acquire more locks

```
def foo():
  acquire(?mutex);
  doSomeStuff();
  release(?mutex);
  bar();
  acquire(?mutex);
  doOtherStuff();
  release(?mutex);
```

Q: If bar() does not access shared variables and does not need a lock, are these the same?

Deadlock Prevention:
Negate ③

- Eliminate Hold & Wait
  - Don't hold resource while waiting for others
    - Rewrite code



```
def foo():
  acquire(?mutex);
  doSomeStuff();
  bar();
  doOtherStuff();
  release(?mutex);
```

code in some other module that may acquire more locks

```
def foo():
  acquire(?mutex);
  doSomeStuff();
  release(?mutex);
  bar();
  acquire(?mutex);
  doOtherStuff();
  release(?mutex);
```

A: No! In the code on the right, the state that the mutex protects can change between doSomeStuff and doOtherStuff

Deadlock Prevention: Negate ③

- Eliminate Hold & Wait
  - Don't hold resource while waiting for others
    - Rewrite code
    - Request all resources before execution begins... but
      - Processes don't know what they need
      - No mechanism to request all resources at the same time
      - Starvation (if waiting on popular resources)
      - Low utilization (if resources needed only briefly)
    - Release all resources before asking new ones
      - Still has the last two problems...

Deadlock Prevention: Negate ④

- Eliminate circular waiting
  - Single lock for the entire system?
  - Impose a total order on the sequence in which different types of resources can be acquired
    - Each resource type is assigned to a level
    - Makes cycles impossible, since a cycle needs to go from low to high level resources, and then back to low
    - Can be relaxed to a strict partial order* if all resources "of the same level" are acquired together

*a binary relation < that is:

1. irreflexive: not $a < a$
2. asymmetric: if $a < b$, then not $b < a$
3. transitive: if $a < b$ and $b < c$, then $a < c$

Havender's Scheme (OS/360)

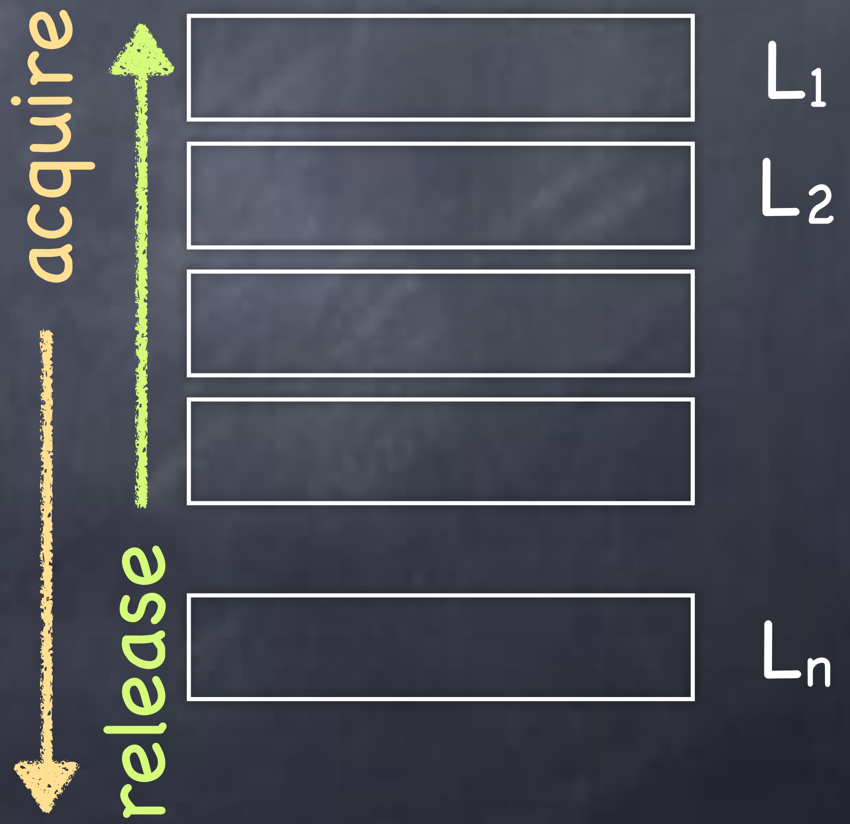
Hierarchical Resource Allocation

Every resource is associated with a level.

Rule H1: All resources from a given level must be acquired using a single request.

Rule H2: After acquiring (and holding) from level L_j, must not acquire from L_i where $i<j$.

Rule H3: May not release from L_i unless already released from L_j where $j>i$.



Example of allowed sequence:

1. acquire(W@L1, X@L1)
2. acquire(Y@L3)
3. release(Y@L3)
4. acquire(Z@L2)

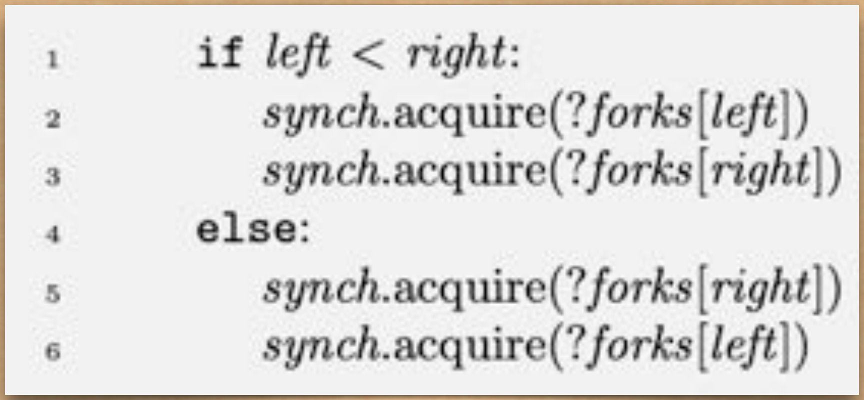
Dining Philosophers (Again)

```
P_i: do forever
       acquire( F(i) );
       acquire( G(i) );
       eat;
       release( F(i) );
       release( G(i) );
     end
```

F(i): min(i, (i+1) mod 5)

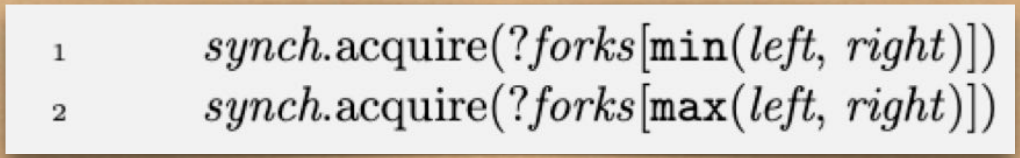
G(i): max(i, (i+1) mod 5)

Ordering Resources in Harmony



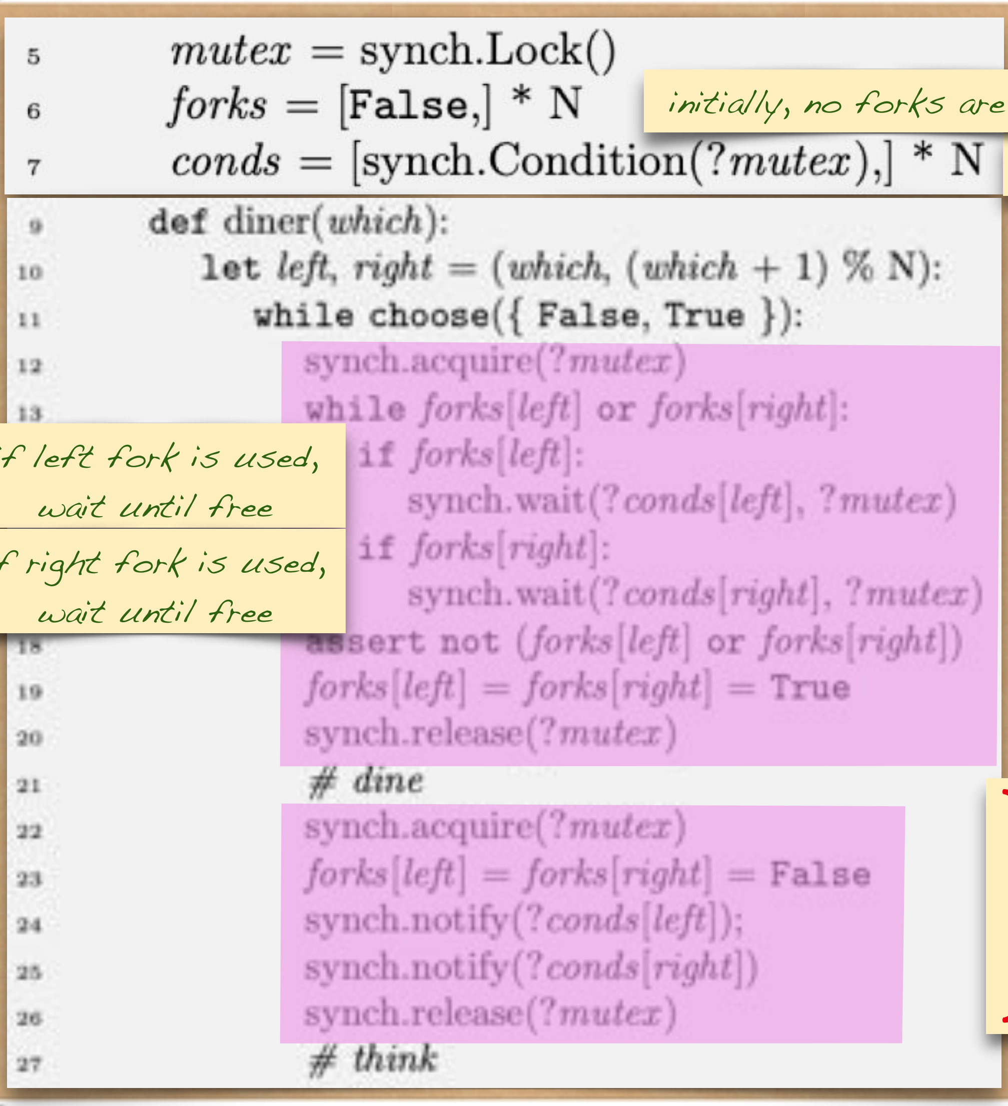
```
if left < right:
    synch.acquire(?forks[left])
    synch.acquire(?forks[right])
else:
    synch.acquire(?forks[right])
    synch.acquire(?forks[left])
```

or



```
synch.acquire(?forks[min(left, right)])
synch.acquire(?forks[max(left, right)])
```

Simultaneous Acquisition in Harmony



```
mutex = synch.Lock()
forks = [False,] * N
conds = [synch.Condition(?mutex),] * N

def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            synch.acquire(?mutex)
            while forks[left] or forks[right]:
                if forks[left]:
                    synch.wait(?conds[left], ?mutex)
                if forks[right]:
                    synch.wait(?conds[right], ?mutex)
            assert not (forks[left] or forks[right])
            forks[left] = forks[right] = True
            synch.release(?mutex)
            # dine
            synch.acquire(?mutex)
            forks[left] = forks[right] = False
            synch.notify(?conds[left]);
            synch.notify(?conds[right])
            synch.release(?mutex)
            # think
```

initially, no forks are held

one condition per fork

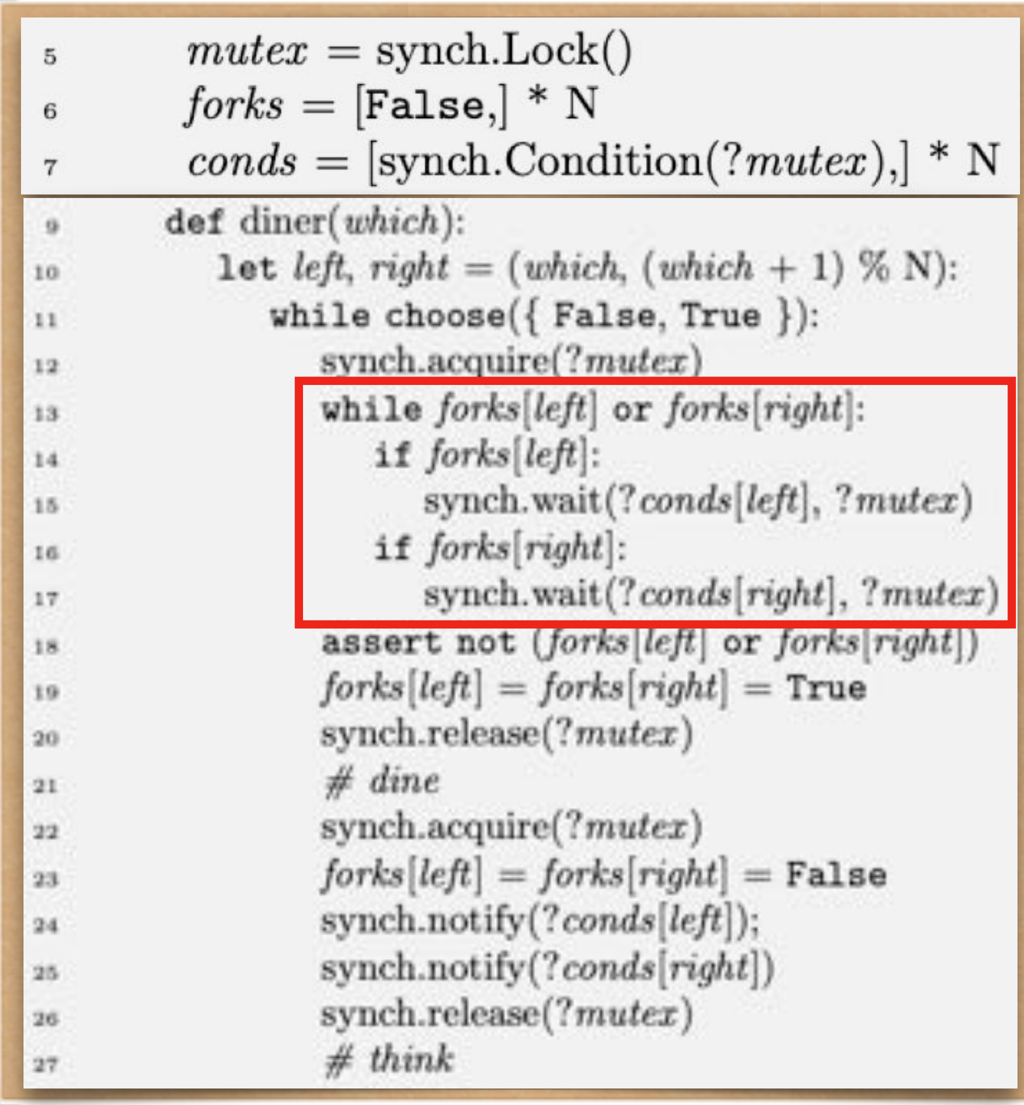
if left fork is used, wait until free

if right fork is used, wait until free

Wait for both forks and then grab them both!

Release both forks

Simultaneous Acquisition in Harmony

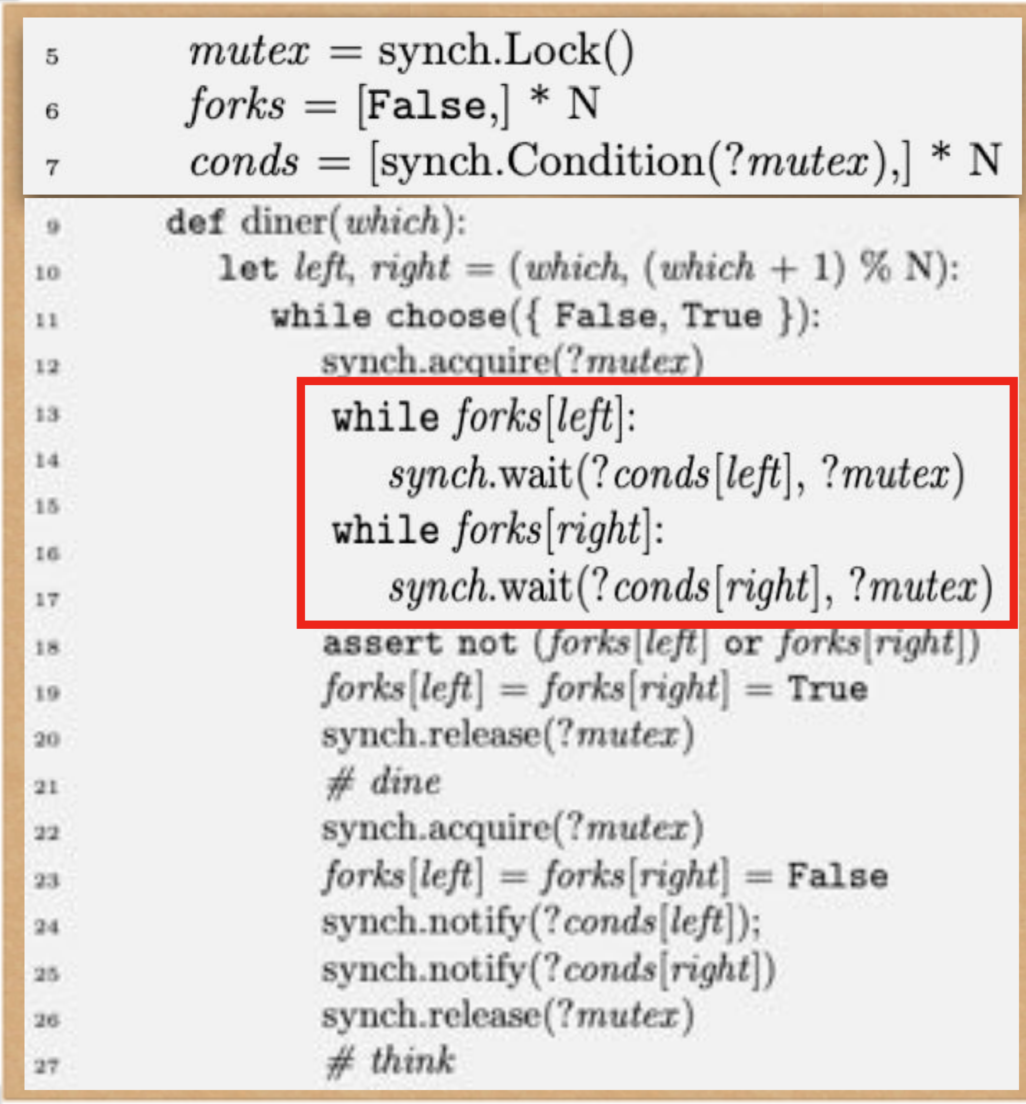


```
mutex = synch.Lock()
forks = [False,] * N
conds = [synch.Condition(?mutex),] * N

def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            synch.acquire(?mutex)
            while forks[left] or forks[right]:
                if forks[left]:
                    synch.wait(?conds[left], ?mutex)
                if forks[right]:
                    synch.wait(?conds[right], ?mutex)
            assert not (forks[left] or forks[right])
            forks[left] = forks[right] = True
            synch.release(?mutex)
            # dine
            synch.acquire(?mutex)
            forks[left] = forks[right] = False
            synch.notify(?conds[left]);
            synch.notify(?conds[right])
            synch.release(?mutex)
            # think
```

Wait for both forks to be available

Simultaneous Acquisition in Harmony



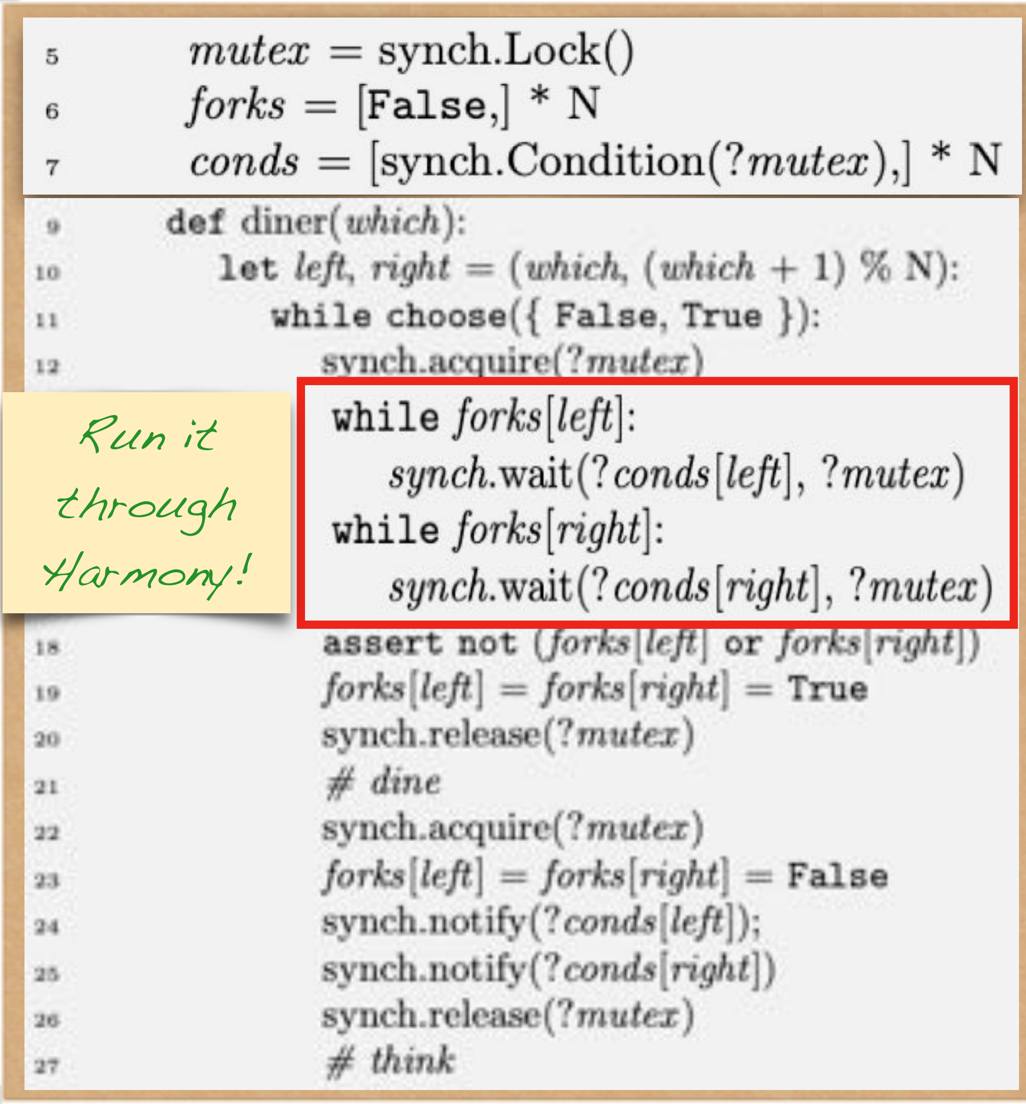
```
mutex = synch.Lock()
forks = [False,] * N
conds = [synch.Condition(?mutex),] * N

def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            synch.acquire(?mutex)
            while forks[left]:
                synch.wait(?conds[left], ?mutex)
            while forks[right]:
                synch.wait(?conds[right], ?mutex)
            assert not (forks[left] or forks[right])
            forks[left] = forks[right] = True
            synch.release(?mutex)
            # dine
            synch.acquire(?mutex)
            forks[left] = forks[right] = False
            synch.notify(?conds[left]);
            synch.notify(?conds[right])
            synch.release(?mutex)
            # think
```

Wait for
left fork
then
wait for
right fork

Wouldn't
this be just
as good?

Simultaneous Acquisition in Harmony



```
mutex = synch.Lock()
forks = [False,] * N
conds = [synch.Condition(?mutex),] * N

def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            synch.acquire(?mutex)
            while forks[left]:
                synch.wait(?conds[left], ?mutex)
            while forks[right]:
                synch.wait(?conds[right], ?mutex)
            assert not (forks[left] or forks[right])
            forks[left] = forks[right] = True
            synch.release(?mutex)
            # dine
            synch.acquire(?mutex)
            forks[left] = forks[right] = False
            synch.notify(?conds[left]);
            synch.notify(?conds[right])
            synch.release(?mutex)
            # think
```

Run it
through
Harmony!

Wait for
left fork
then
wait for
right fork

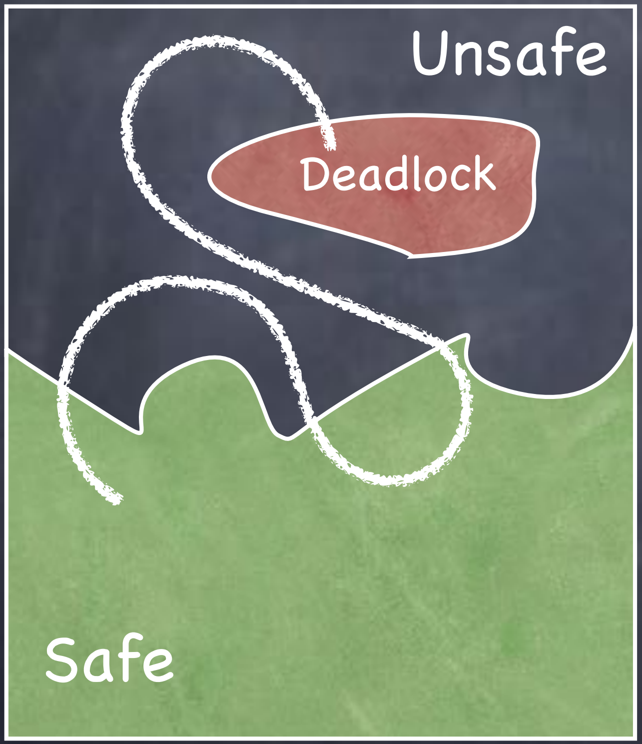
NO!

Avoiding Deadlock: The Banker's Algorithm

E.W. Dijkstra & N. Habermann

- Sum of maximum resources needs can exceed the total available resources
  - if there exists a schedule of loan fulfillments such that
    - all clients receive their maximal loan
    - build their house
    - pay back all the loan
- More efficient than acquiring atomically all resources

Living dangerously: Safe, Unsafe, Deadlocked



A system's trajectory
through its state space

- Safe: For any possible set of resource requests, there exists one safe schedule of processing requests that succeeds in granting all pending and future requests
  - no deadlock as long as system can enforce that safe schedule!
- Unsafe: There exists a set of (pending and future) resource requests that leads to a deadlock, independent of the schedule in which requests are processed
  - unlucky set of requests can force deadlock
- Deadlocked: The system has at least one deadlock

Proactive Responses to Deadlock: Avoidance

The Banker's Algorithm

E.W. Dijkstra & N. Habermann

- Processes declare worst-case needs (big assumption!), but then ask for what they "really" need, a little at a time
  - Sum of maximum resource needs can exceed total available resources
- Algorithm decides whether to grant a request
  - Build a graph assuming request granted
  - Check whether state is safe (i.e., whether RAG is reducible)
    - A state is safe if there exists some permutation of [P_1, P_2,...,P_n] such that, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources currently held by all P_j, for P_j preceding P_i in the permutation

Available = 3

Process	Max	Holds	Needs
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7

Safe?

✓ Available resources can satisfy P_1's needs
✓ Once P_1 finishes, 5 available resources
✓ Now, available resources can satisfy P_0's needs
✓ Once P_0 finishes, 10 available resources
✓ Now, available resources can satisfy P_3's needs

Yes! Schedule: [P_1, P_0, P_3]

Proactive Responses to Deadlock: Avoidance

The Banker's Algorithm

E.W. Dijkstra & N. Habermann

- Processes declare worst-case needs (big assumption!), but then ask for what they "really" need, a little at a time
  - Sum of maximum resource needs can exceed total available resources
- Algorithm decides whether to grant a request
  - Build a graph assuming request granted
  - Check whether state is safe (i.e., whether RAG is reducible)
    - A state is safe if there exists some permutation of [P_1, P_2,…,P_n] such that, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources currently held by all P_j, for P_j preceding P_i in the permutation

Available = 3

Process	Max	Holds	Needs
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7

Suppose P_2 asks for 2 resources

If granted, is the resulting state Safe?

Proactive Responses to Deadlock: Avoidance

The Banker's Algorithm

E.W. Dijkstra & N. Habermann

- Processes declare worst-case needs (big assumption!), but then ask for what they "really" need, a little at a time
  - Sum of maximum resource needs can exceed total available resources
- Algorithm decides whether to grant a request
  - Build a graph assuming request granted
  - Check whether state is safe (i.e., whether RAG is reducible)
    - A state is safe if there exists some permutation of [P_1, P_2,…,P_n] such that, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources currently held by all P_j, for P_j preceding P_i in the permutation

Available = 3

Process	Max	Holds	Needs
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7

Safe?

Available = 1

Process	Max	Holds	Needs
P_0	10	5	5
P_1	4	2	2
P_2	9	4	5

  - If so, request is granted; otherwise, requester must wait

The Banker's books

- Assume n processes, m resources
- **Max_{ij}** = max amount of units of resource R_j needed by P_i
  - **MaxClaim_i:** Vector of size m such that $\text{MaxClaim}_i[j] = \text{Max}_{ij}$
- **Holds_{ij}** = current allocation of R_j held by P_i
  - **HasNow_i** = Vector of size m such that $\text{HasNow}_i[j] = \text{Holds}_{ij}$
- **Available** = Vector of size m such that Available[j] = units of R_j available
- A request by P_k is safe if, assuming the request is granted, there is a permutation of $P_1, P_2, \ldots, P_n$ such that, for all P_i in the permutation

$$\text{Needs}_i = \text{MaxClaim}_i - \text{HasNow}_i \leq \text{Avail} + \sum_{j=1}^{i-1} \text{HasNow}_j$$