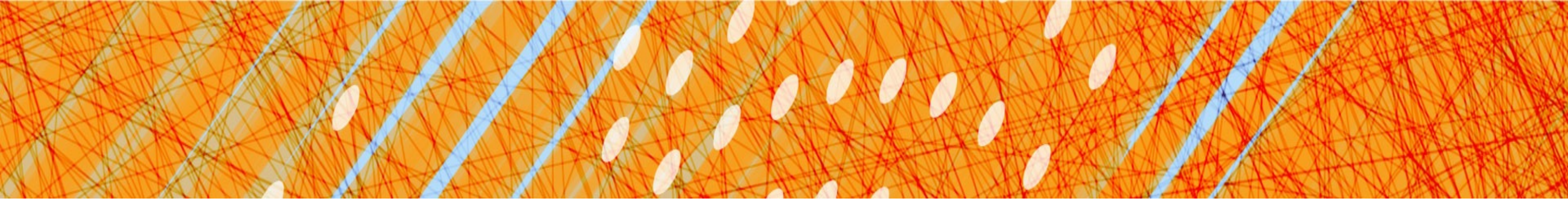


Deadlock

Chapter 32 in “Three Easy Steps”
Chapter 19 in Harmony Book

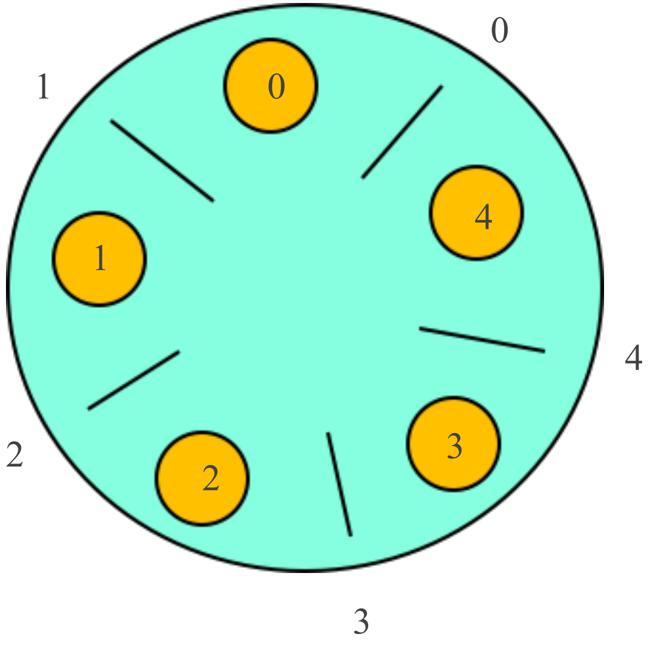
CS 4410
Operating Systems

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

The slides are the product of many rounds of teaching CS 4410
by Professors Agarwal, Alvisi, Bracy, George, Schneider, Sirer, Van Renesse.

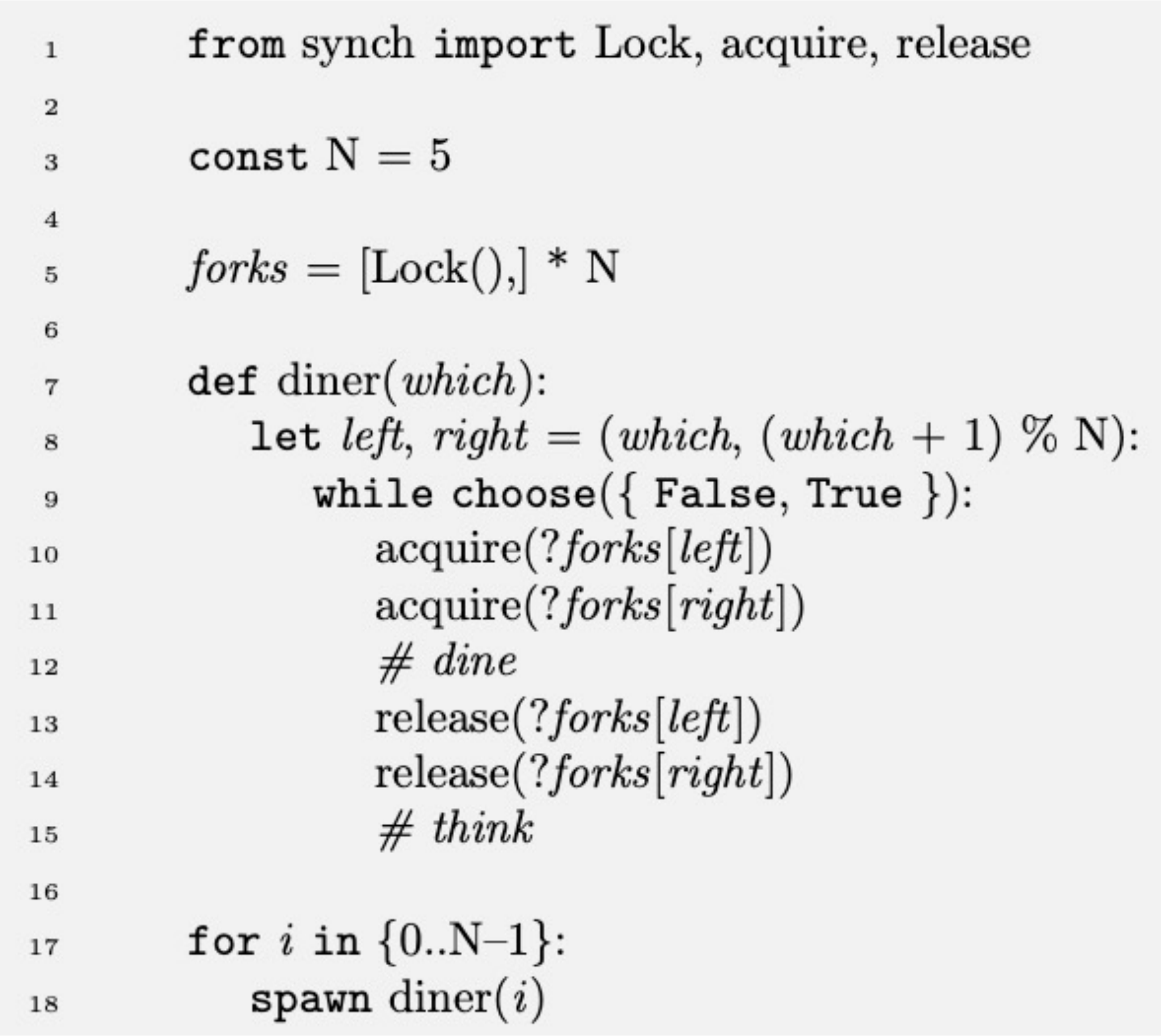
Dining Philosophers [Dijkstra 68]

```
Pi: do forever
      acquire( left(i) );
      acquire( right(i) );
      eat
      release( left(i) );
      release( right(i) );
    end
```



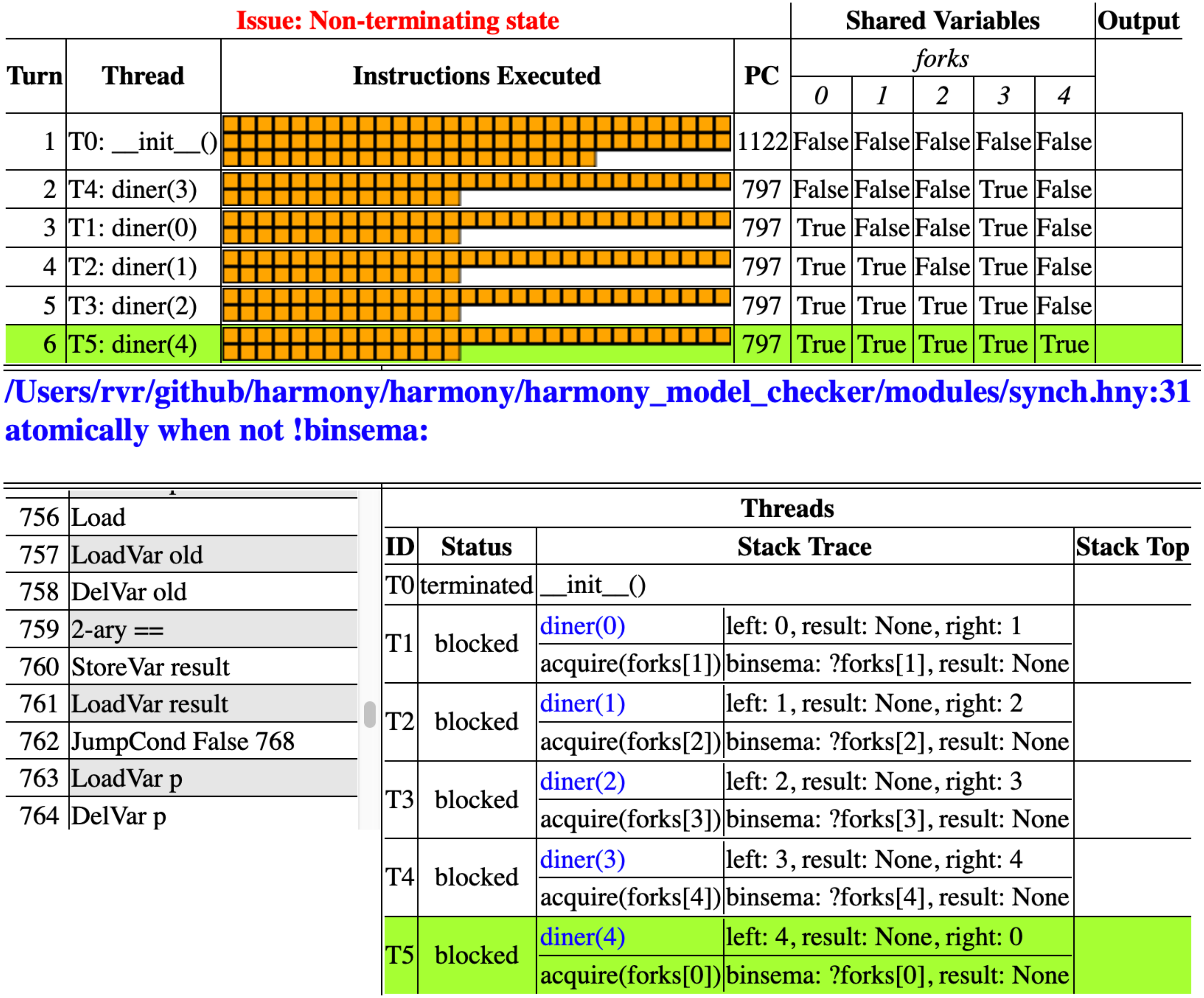
right(i): i+1 mod 5
left(i): i

Dining Philosophers in Harmony



```
from synch import Lock, acquire, release

const N = 5

forks = [Lock(),] * N

def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            acquire(?forks[left])
            acquire(?forks[right])
            # dine
            release(?forks[left])
            release(?forks[right])
            # think

for i in {0..N−1}:
    spawn diner(i)
```

Dining Philosophers in Harmony



Issue: Non-terminating state

Turn	Thread	Instructions Executed	PC	Shared Variables *forks* 0	*1*	*2*	*3*	*4*	Output
1	T0: __init__()		1122	False	False	False	False	False	
2	T4: diner(3)		797	False	False	False	True	False	
3	T1: diner(0)		797	True	False	False	True	False	
4	T2: diner(1)		797	True	True	False	True	False	
5	T3: diner(2)		797	True	True	True	True	False	
6	T5: diner(4)		797	True	True	True	True	True	

/Users/rvr/github/harmony/harmony/harmony_model_checker/modules/synch.hny:31 atomically when not !binsema:

756	Load
757	LoadVar old
758	DelVar old
759	2-ary ==
760	StoreVar result
761	LoadVar result
762	JumpCond False 768
763	LoadVar p
764	DelVar p

Threads

ID	Status	Stack Trace		Stack Top
T0	terminated	__init__()		
T1	blocked	diner(0)	left: 0, result: None, right: 1	
		acquire(forks[1])	binsema: ?forks[1], result: None	
T2	blocked	diner(1)	left: 1, result: None, right: 2	
		acquire(forks[2])	binsema: ?forks[2], result: None	
T3	blocked	diner(2)	left: 2, result: None, right: 3	
		acquire(forks[3])	binsema: ?forks[3], result: None	
T4	blocked	diner(3)	left: 3, result: None, right: 4	
		acquire(forks[4])	binsema: ?forks[4], result: None	
T5	blocked	diner(4)	left: 4, result: None, right: 0	
		acquire(forks[0])	binsema: ?forks[0], result: None	

Problematic Emergent Properties

Starvation: Process waits forever

Deadlock: A set of processes exists, where each is blocked and can become unblocked only by actions of another process in the set.

- Deadlock implies Starvation (but not *vice versa*)

- Starvation often tied to **fairness**: A process is not forever blocked awaiting a condition that (i) becomes continuously true or (ii) infinitely-often becomes true.

Testing for starvation or deadlock is difficult in practice

More Examples of Deadlock

Example (initially in1 = in2 = False):

```
in1 = True;  await not in2;  in1 = False
//
in2 := True;  await not in1;  in2 = False
```

Example (initially lk1 = lk2 = released):

```
acquire(lk1); acquire(lk2);  release(lk2); release(lk1);
//
acquire(lk2); acquire(lk1); release(lk1); release(lk2);
```

System Model

- Set of resources requiring “exclusive” access
    - Might be “k-exclusive access” if resource has capacity for k
    - Examples: buffers, packets, I/O devices, processors, …

- Protocol to access a resource causes blocking:
    - If resource is free, then access is granted; process proceeds
    - If resource is in use, then process blocks
  - Use resource
  - Release resource

When is deadlock possible?

Necessary Conditions for Deadlock

Edward Coffman 1971

1. **Mutual Exclusion**. Acquire can block invoker
2. **Hold & wait**. A process can be blocked while holding resources
3. **No preemption**. Allocated resources cannot be reclaimed. Explicit release operation needed
4. **Circular waits** are possible

Let p → q denote "p waits for q to release a resource". Then

P1 → P2 → … → Pn → P1

Deadlock is Undesirable

- Deadlock prevention: Ensure that a necessary condition cannot hold

- Deadlock avoidance: System does not allocate resources that will lead to a deadlock

- Deadlock detection: Allow system to deadlock; detect it; recover

Deadlock Prevention: Negate 1

#1: Eliminate mutual exclusion / bounded resources:

- Make resources sharable without locks
  - Harmony book Chapter 23 has examples of non-blocking data structures
- Have sufficient resources available, so acquire never delays
  - E.g., unbounded queue, or simply make sure bounded queue is “large enough”

Deadlock Prevention: Negate 2

#2: Eliminate hold and wait

Don’t hold some resources when waiting for others.

- Re-write code:

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

→

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

- *Assuming bar does not access shared variables and does not need the lock, are these the same?*

Deadlock Prevention: Negate 2

#2: Eliminate hold and wait

Don’t hold some resources when waiting for others.

- Re-write code:

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

→

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

- *Answer: no. The state that mutex protects may change between doSomeStuff and doOtherStuff in code on the right.*

Deadlock Prevention: Negate 2

#2: Eliminate hold and wait

Don’t hold some resources when waiting for others.

- Re-write code

- Another approach: request all resources a priori
  - Problems:
    - Processes don’t know what they need ahead of time
    - No mechanism to request all resources at the same time
    - Starvation (if waiting on many popular resources)
    - Low utilization (need resource only for a bit)

Deadlock Prevention: Negate 3

#3: Allow preemption

Requires mechanism to save / restore resource state:

multiplexing vs undo/redo

- Examples of multiplexing:
  - processor registers (contexts)
  - Regions of memory (pages)
- Examples of undo/redo
  - Database transaction processing

Deadlock Prevention: Negate 4

#4: Eliminate circular waits.

Let $R = \{R1, R2, \ldots Rn\}$ be the set of resource types.
Let $(R , <)$ be a non-symmetric relation:

- not $r < r$ [irreflexive]
- if $r < s$ and $s < t$ then $r < t$ [transitive]
- not $r < s$ and $s < r$ [non-symmetric]
- for every r and s ($r \neq s$): $r < s$ or $s < r$ [total order]

Rule: Request resources in increasing order by <
(All resources from type Ri must be requested together)

Rule: To request resources of type Ri, first release all resources from type Rj where Ri < Rj.

Why < Rules Work

Thm: Total order resource allocation avoids circular waits

Proof: By contradiction. Assume a circular wait exists

P1 → P2 → P3 → … → Pn → P1.

P1 requesting R1 held by P2.

P2 requesting R2 held by P3. (So R1 < R2 holds)

…

Conclude: R1 < R2, R2 < R3, …, Rn < R1

By transitivity: R1 < R1. A contradiction!

Havender's Scheme (OS/360)

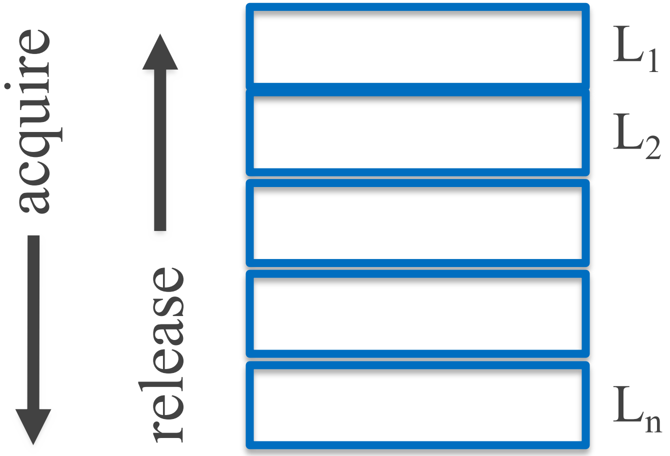
Hierarchical Resource Allocation

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 from level L_j must not acquire from L_i where $i < j$
- **Rule H3**: May not acquire from L_i unless already released from L_j where $j > i$.

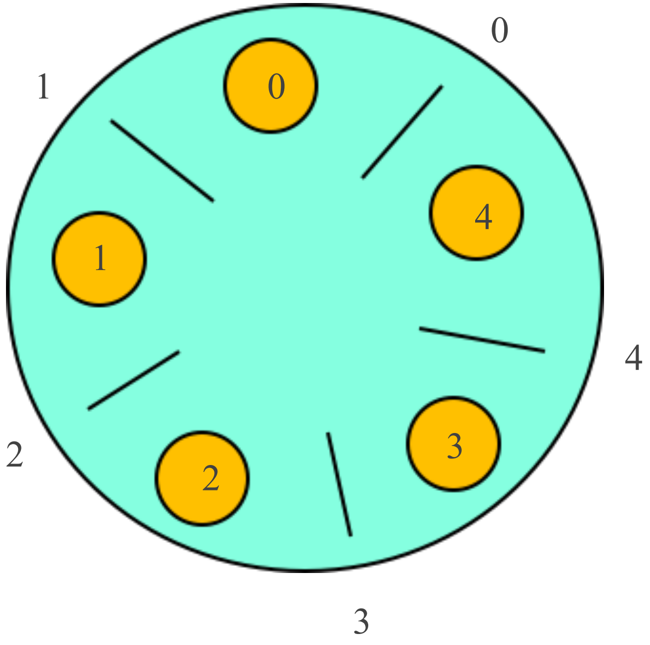
Example of allowed sequence:

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



Dining Philosophers (Again)

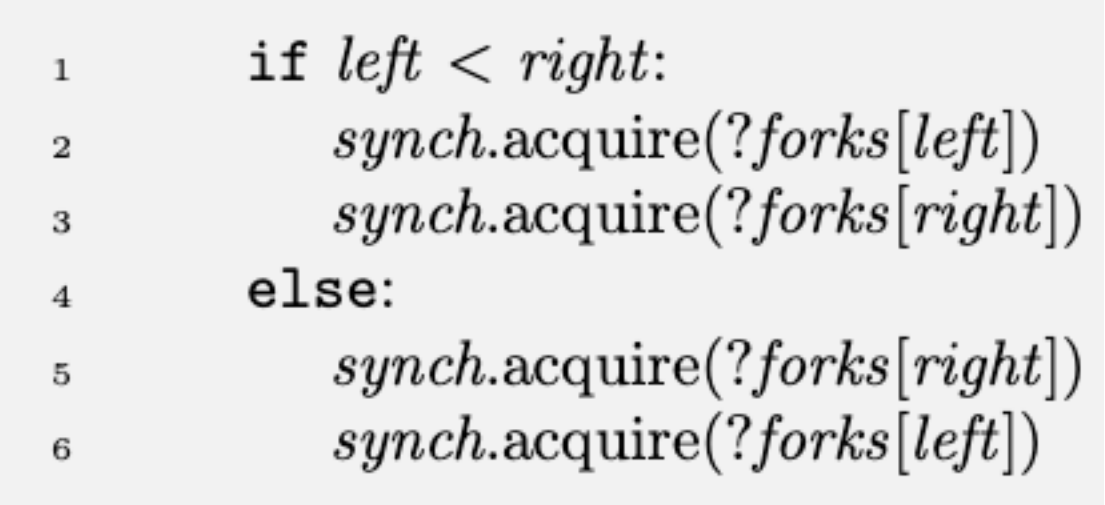
```
Pi:  do forever
       acquire( F(i) );
       acquire( G(i) );
       eat
       release( F(i) );
       release( G(i) );
     end
```



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

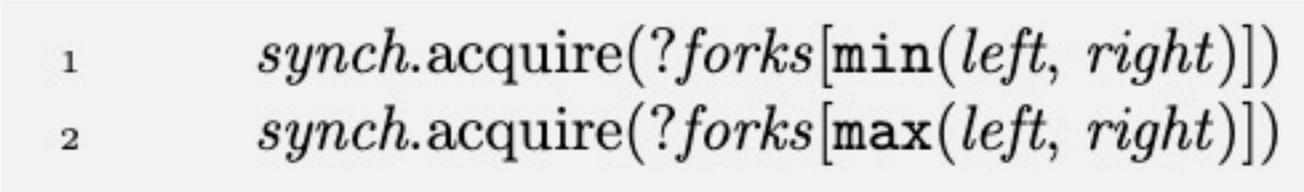
G(i): max(i, i+1 mod 5)

Ordering Resources in Harmony



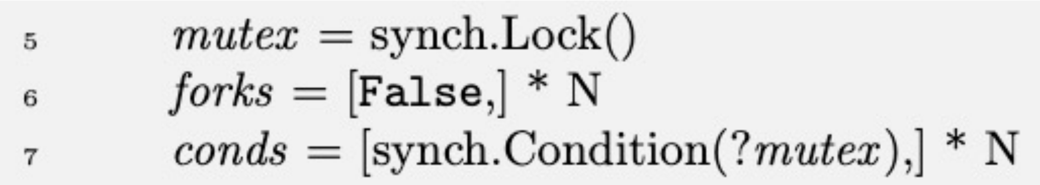
```
if left < right:
    synch.acquire(?forks[left])
    synch.acquire(?forks[right])
else:
    synch.acquire(?forks[right])
    synch.acquire(?forks[left])
```

or

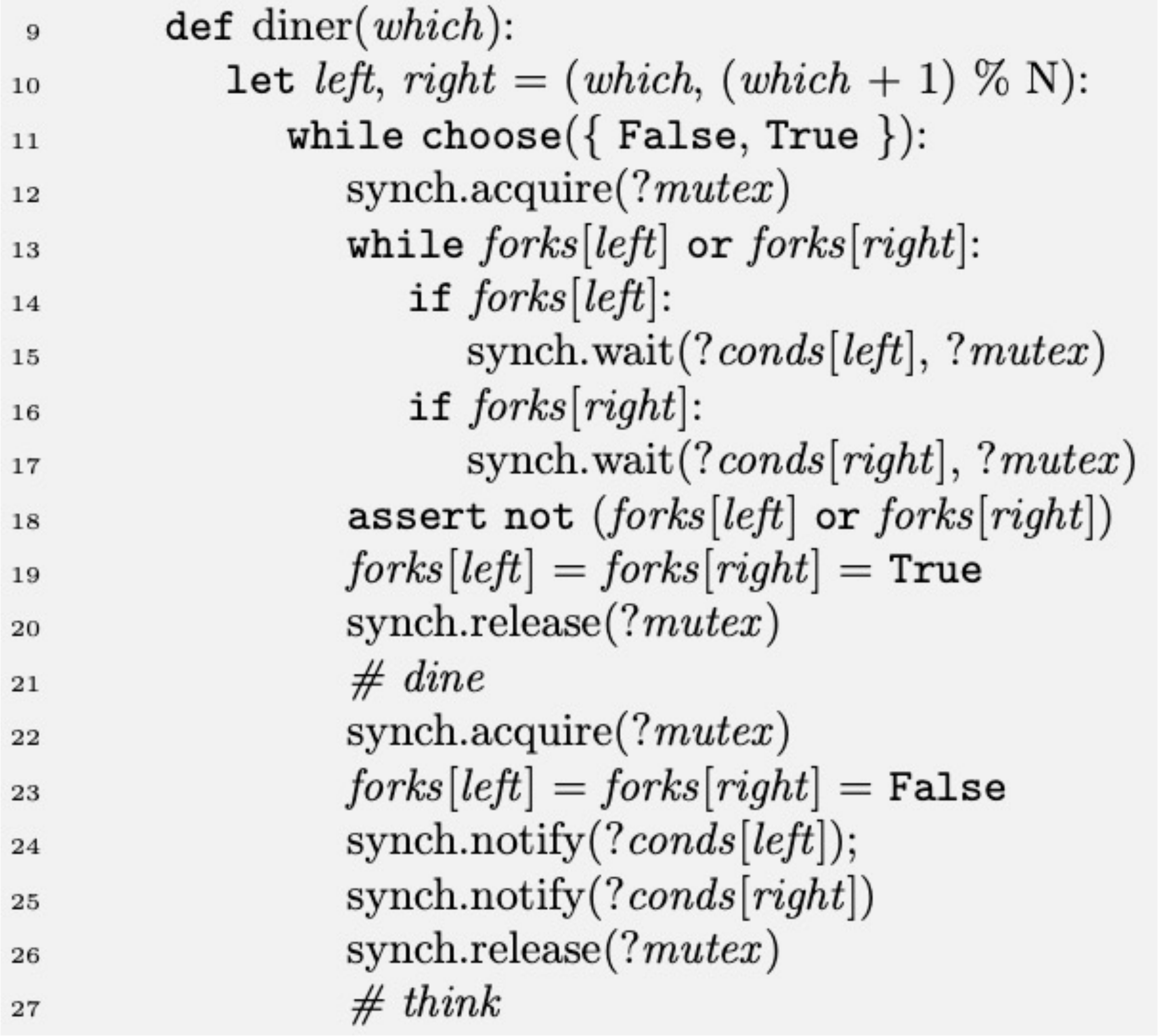


```
synch.acquire(?forks[min(left, right)])
synch.acquire(?forks[max(left, right)])
```

Simultaneous Acquisition in Harmony



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

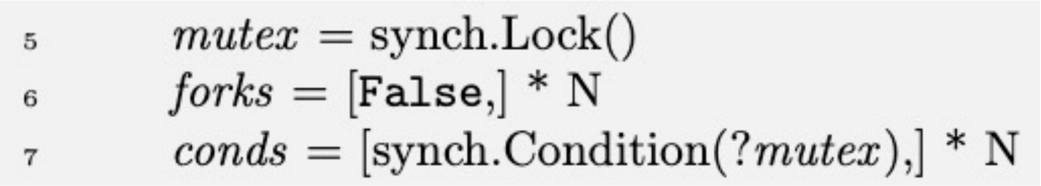


```
def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            synch.acquire(?mutex)
            while forks[left] or forks[right]:
                if forks[left]:
                    synch.wait(?conds[left], ?mutex)
                if forks[right]:
                    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
```

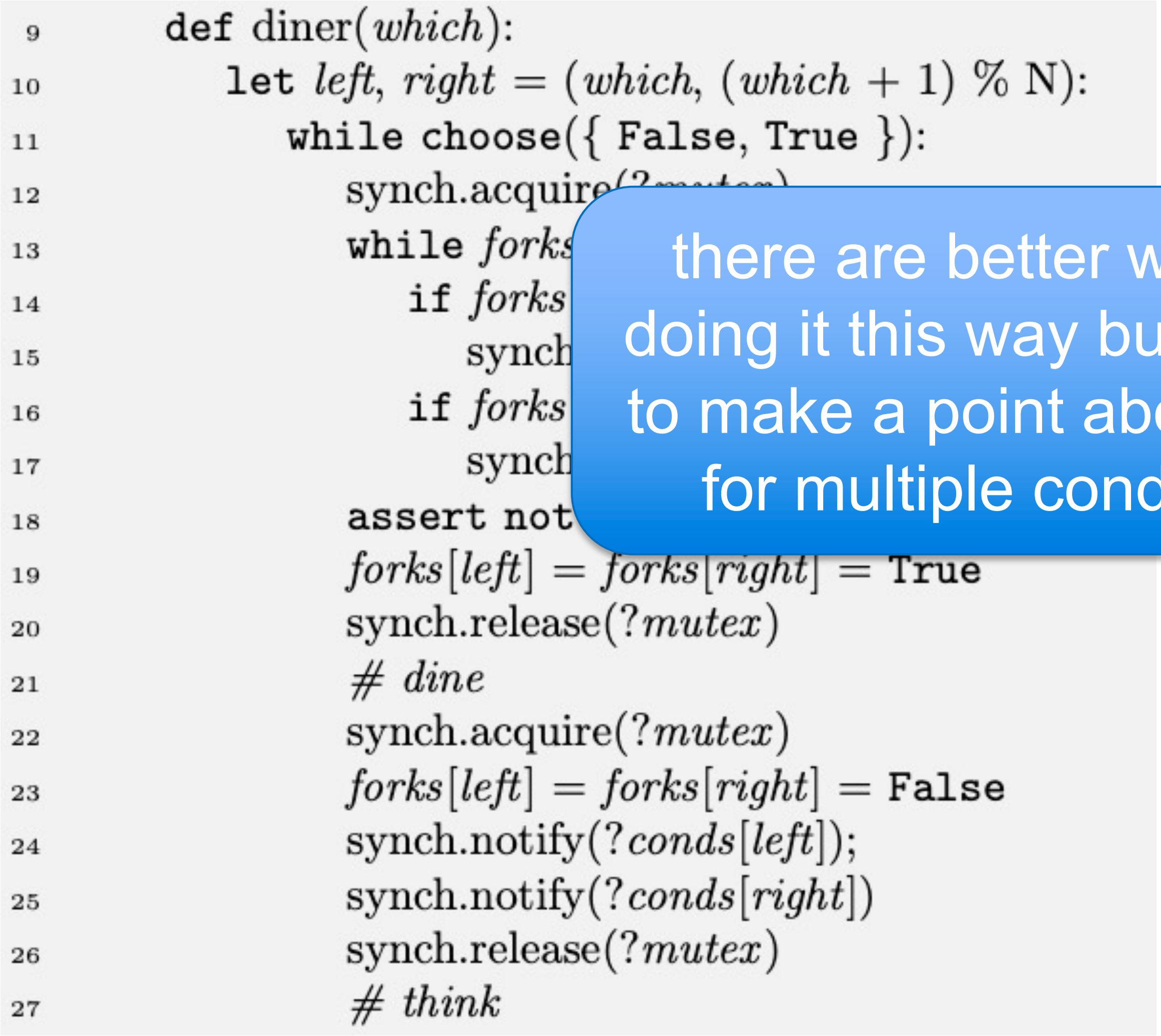
wait for both forks and then grab them both

release both forks

Simultaneous Acquisition in Harmony



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



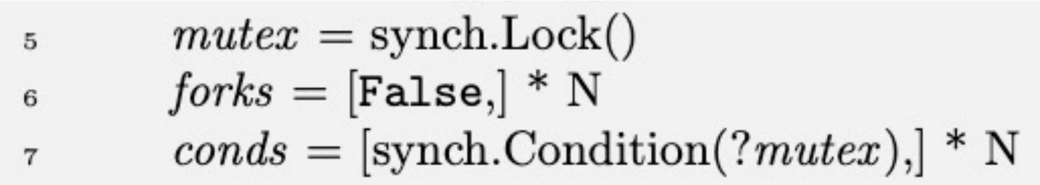
```
def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            synch.acquire(?mutex)
            while forks
                if forks
                    synch
                if forks
                    synch
            assert not
            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
```

there are better ways than doing it this way but I'm trying to make a point about waiting for multiple conditions…

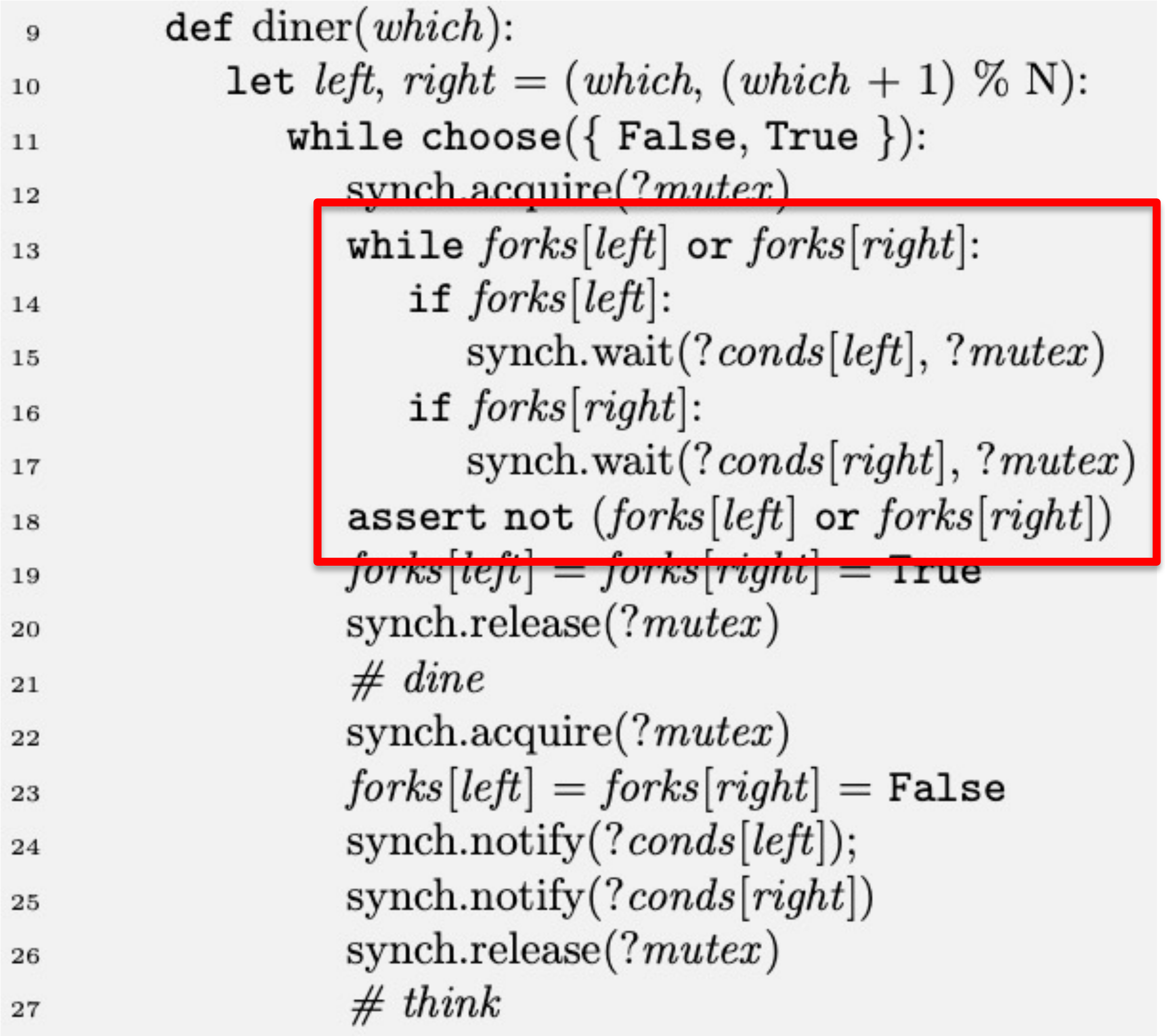
th forks and
hem both

release both forks

Simultaneous Acquisition in Harmony



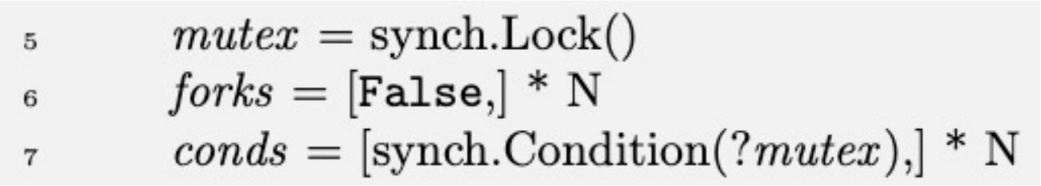
```
mutex = synch.Lock()
forks = [False,] * N
conds = [synch.Condition(?mutex),] * N
```



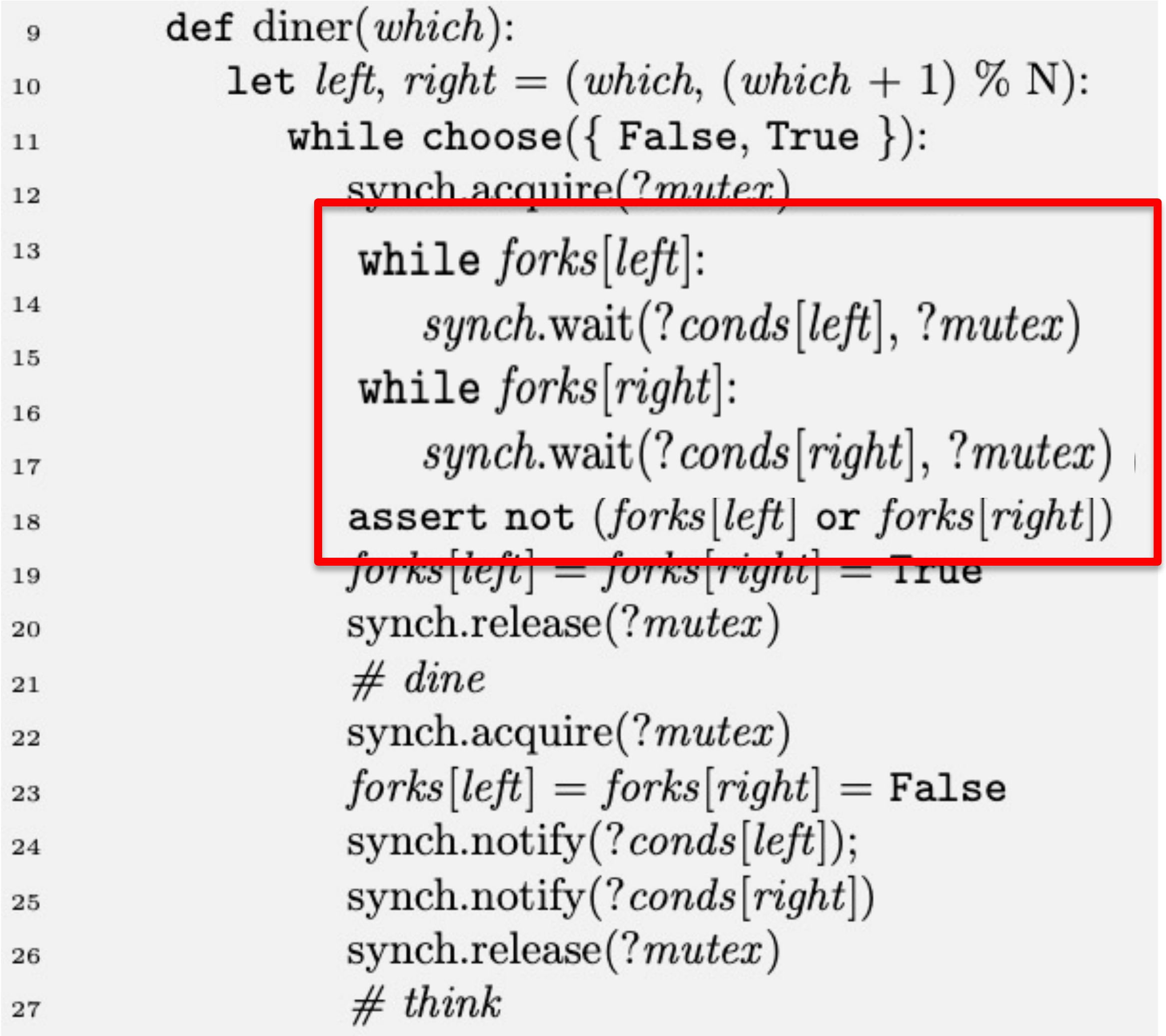
```
def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            synch.acquire(?mutex)
            while forks[left] or forks[right]:
                if forks[left]:
                    synch.wait(?conds[left], ?mutex)
                if forks[right]:
                    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
```

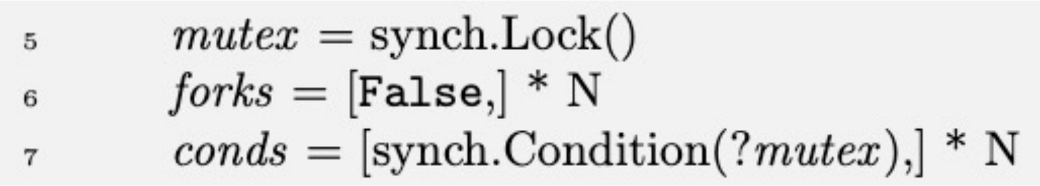


```
def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            synch.acquire(?mutex)
            while forks[left]:
                synch.wait(?conds[left], ?mutex)
            while forks[right]:
                synch.wait(?conds[right], ?mutex)
            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
```

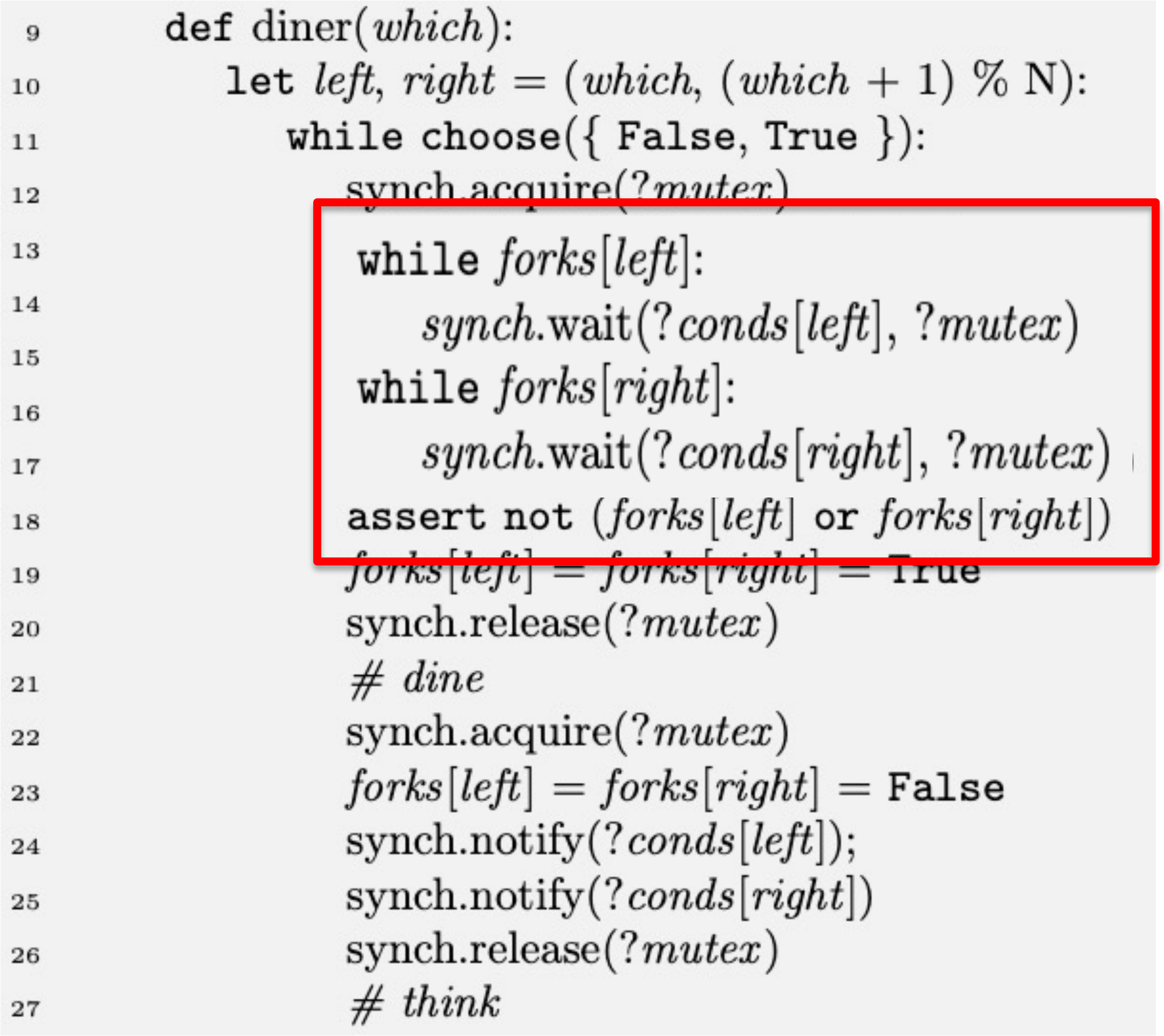
Wait for left fork, then wait for right fork.

Wouldn't this be just as good?

Simultaneous Acquisition in Harmony



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



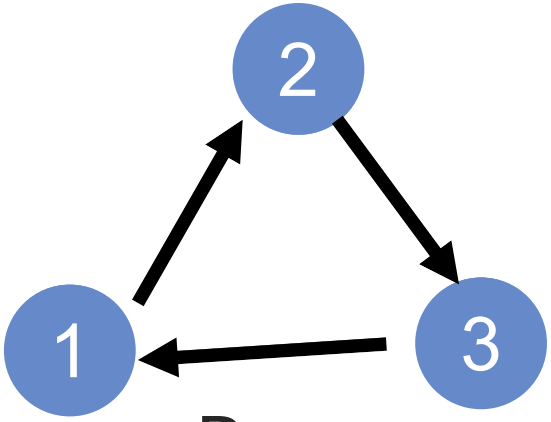
```
def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            synch.acquire(?mutex)
            while forks[left]:
                synch.wait(?conds[left], ?mutex)
            while forks[right]:
                synch.wait(?conds[right], ?mutex)
            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
```

Wait for left fork, then wait for right fork.
Wouldn't this be just as good?

NO!

(run through harmony if you don't believe me)

Deadlock Detection



Create a Wait-For Graph

- 1 Node per Process
- 1 Outgoing Edge per Waiting Process, P (from P to the process it's waiting for)

Note: graph holds for a single instant in time

Cycle in graph indicates deadlock

Testing for cycles (= deadlock)

Reduction Algorithm:
Find a node with no outgoing edges

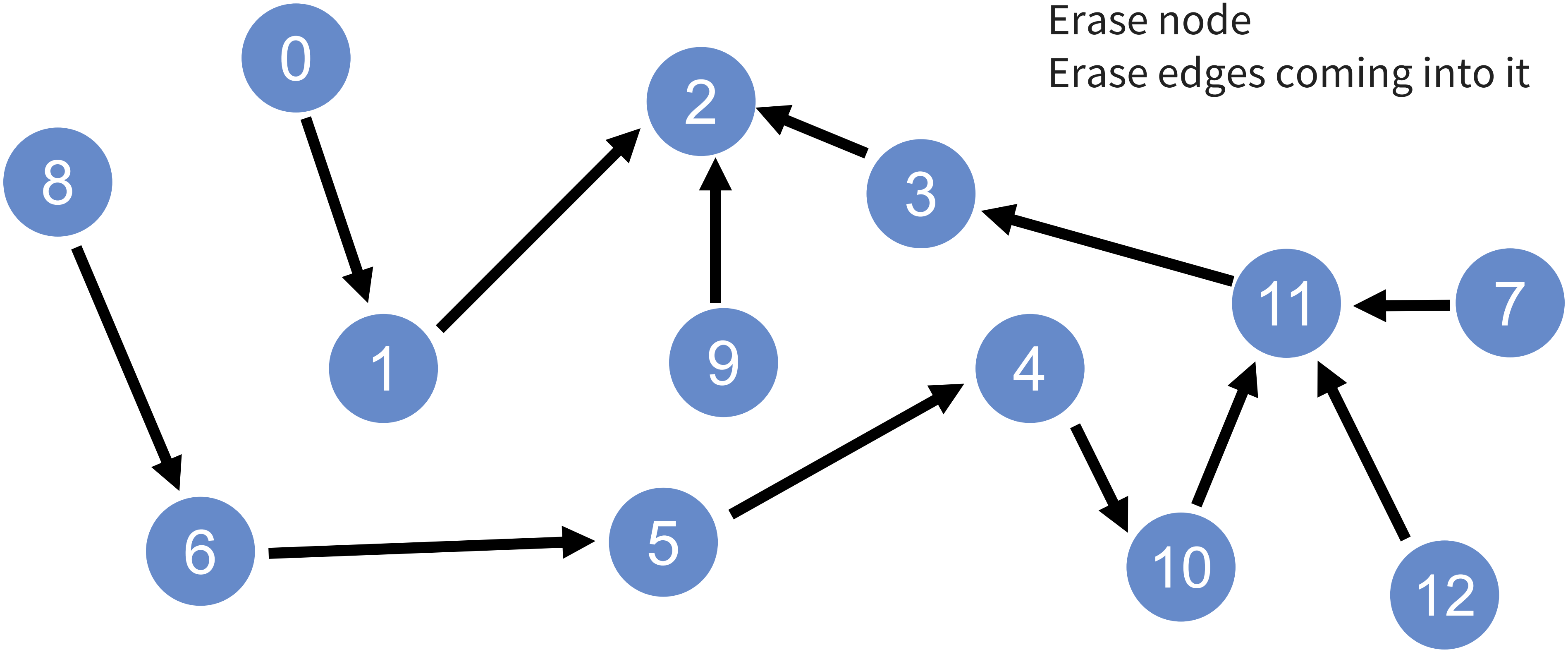
- Erase node
- Erase any edges coming into it
- Repeat until no such node

Intuition: Deleted node is for process that is not waiting. It will eventually finish and release its resources, so any process waiting for those resources will longer be waiting.

Erase whole graph ↔ graph has no cycles
Graph remains ↔ deadlock

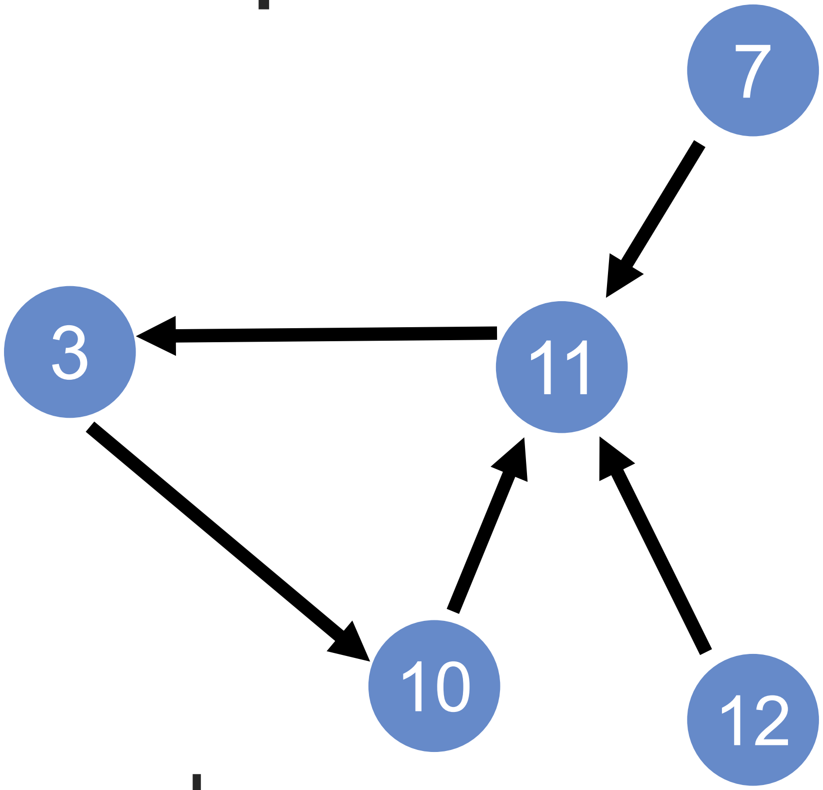
Graph Reduction: Example 1

Find node w/o outgoing edges
- Erase node
- Erase edges coming into it



Graph can be fully reduced, hence there was no deadlock at the time the graph was drawn. (Obviously, things could change later!)

Graph Reduction: Example 2



No node with no outgoing edges…

Irreducible graph, contains a cycle

(only some processes are in the cycle)

➛ deadlock

Question:

Does choice of node for reduction matter?

Answer: No.
Explanation: an unchosen candidate at one step remains a candidate for later steps. Eventually—regardless of order—every node will be reduced (if there’s no deadlock).

Question:

Suppose no deadlock detected at time T.
Can we infer about a later time T+x?

Answer: Nothing.
Explanation: The very next step could be to run some process that will request a resource…
    … establishing a cyclic wait
    … and causing deadlock

Implementing Deadlock Detection

- Track resource allocation (who has what)
- Track pending requests (who’s waiting for what)

Maintain a wait-for graph.

When to run graph reduction?

- Whenever a request is blocked?
- Periodically?
- Once CPU utilization drops below a threshold?

Deadlock Recovery

Blue screen & reboot?

Kill one/all deadlocked processes

- Pick a victim
- Terminate
- Repeat if needed

Preempt resource/processes till deadlock broken

- Pick a victim (# resources held, execution time)
- Rollback (partial or total, not always possible)

Deadlock Avoidance

How do cars do it?

- Try not to block an intersection
- Don’t drive into the intersection if you see that you might get stuck there

Why does this work?

- Prevents a wait-for relationship
- Cars won’t take up a resource if they see they won’t be able to acquire the next one…

Deadlock Avoidance

state: allocation to each process

safe state: a state from which some execution is possible that does not cause deadlock

- Requires knowing max allocation for each process
- Check that
  - Exists sequence P1 P2 … Pn of processes where:

    Forall i where $1 \le i \le n$:

    Pi can be satisfied by Avail + resources held by P1 … Pi-1.

Assumes no synchronization between processes, except for resource requests

Safe State Example

Suppose: 12 tape drives and 3 processes: p0, p1, and p2

	max need	current usage	could still ask for
p0	10	5	5
p1	4	2	2
p2	9	2	7

3 drives remain

Is this a safe state (i.e, is there a sequence of granting requests that will work without deadlock)?

Safe State Example

Suppose: 12 tape drives and 3 processes: p0, p1, and p2

	max need	current usage	could still ask for
p0	10	5	5
p1	4	2	2
p2	9	2	7

3 drives remain

Current state is *safe* because a safe sequence exists: [p1, p0, p2]

- p1 can complete with remaining resources
- p0 can complete with remaining+p1
- p2 can complete with remaining+p1+p0

What if p2 requests 1 drive? Grant or not?

Safe State Example

Suppose: 12 tape drives and 3 processes: p0, p1, and p2

	max need	current usage	could still ask for
p0	10	5	5
p1	4	2	2
p2	9	3	6

2 drives remain

Is this state safe? (Is there a sequence of requests that works?)

Safe State Example

Suppose: 12 tape drives and 3 processes: p0, p1, and p2

	max need	current usage	could still ask for
p0	10	5	5
~~p1~~	~~0~~	~~0~~	~~0~~
p2	9	3	6

4 drives remain

Is this state safe? (Is there a sequence of requests that works?)

Safe State Example

Suppose: 12 tape drives and 3 processes: p0, p1, and p2

	max need	current usage	could still ask for
p0	10	5	5
~~p1~~	~~0~~	~~0~~	~~0~~
p2	9	3	6

4 drives remain

Is this state safe? (Is there a sequence of requests that works?)

(potentially) STUCK…

(non-terminating state)

Safe State Example

Suppose: 12 tape drives and 3 processes: p0, p1, and p2

	max need	current usage	could still ask for
p0	10	5	5
p1	4	2	2
p2	9	2	7

`3 drives remain`

Current state is *safe* because a safe sequence exists: [p1, p0, p2]

- p1 can complete with remaining resources
- p0 can complete with remaining+p1
- p2 can complete with remaining+p1+p0

What if p2 requests 1 drive? Grant or not?

Safe State Example

Suppose: 12 tape drives and 3 processes: p0, p1, and p2

	max need	current usage	could still ask for
p0	10	5	5
p1	4	2	2
p2	9	2	7

`3 drives remain`

Current state is *safe* because a safe sequence exists: [p1, p0, p2]

- p1 can complete with remaining resources
- p0 can complete with remaining+p1
- p2 can complete with remaining+p1+p0

What if p2 requests 1 drive? Grant or not? NO (block or deny)

Banker's Algorithm

Dijkstra 1977

- from 10,000 feet:
  - Process declares its worst-case needs, asks for what it "really" needs, a little at a time
  - Algorithm decides when to grant requests
    - Build a graph assuming request granted
    - Reducible? yes: grant request, no: wait

Problems:

- Fixed number of processes
- Need worst-case needs ahead of time
- Expensive

→ not used much practice