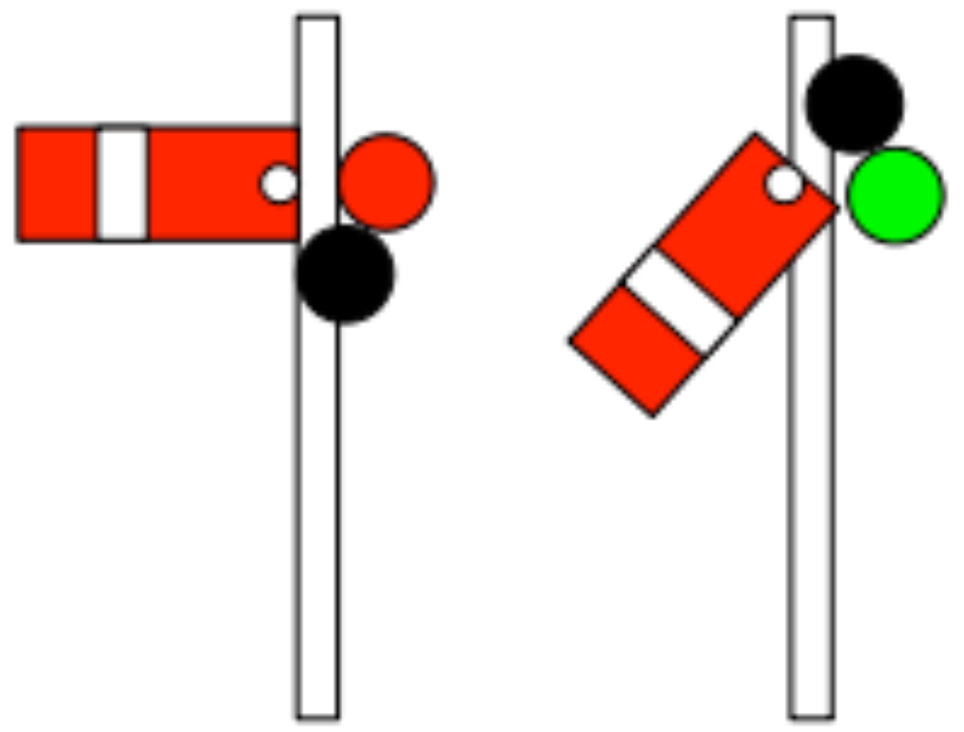
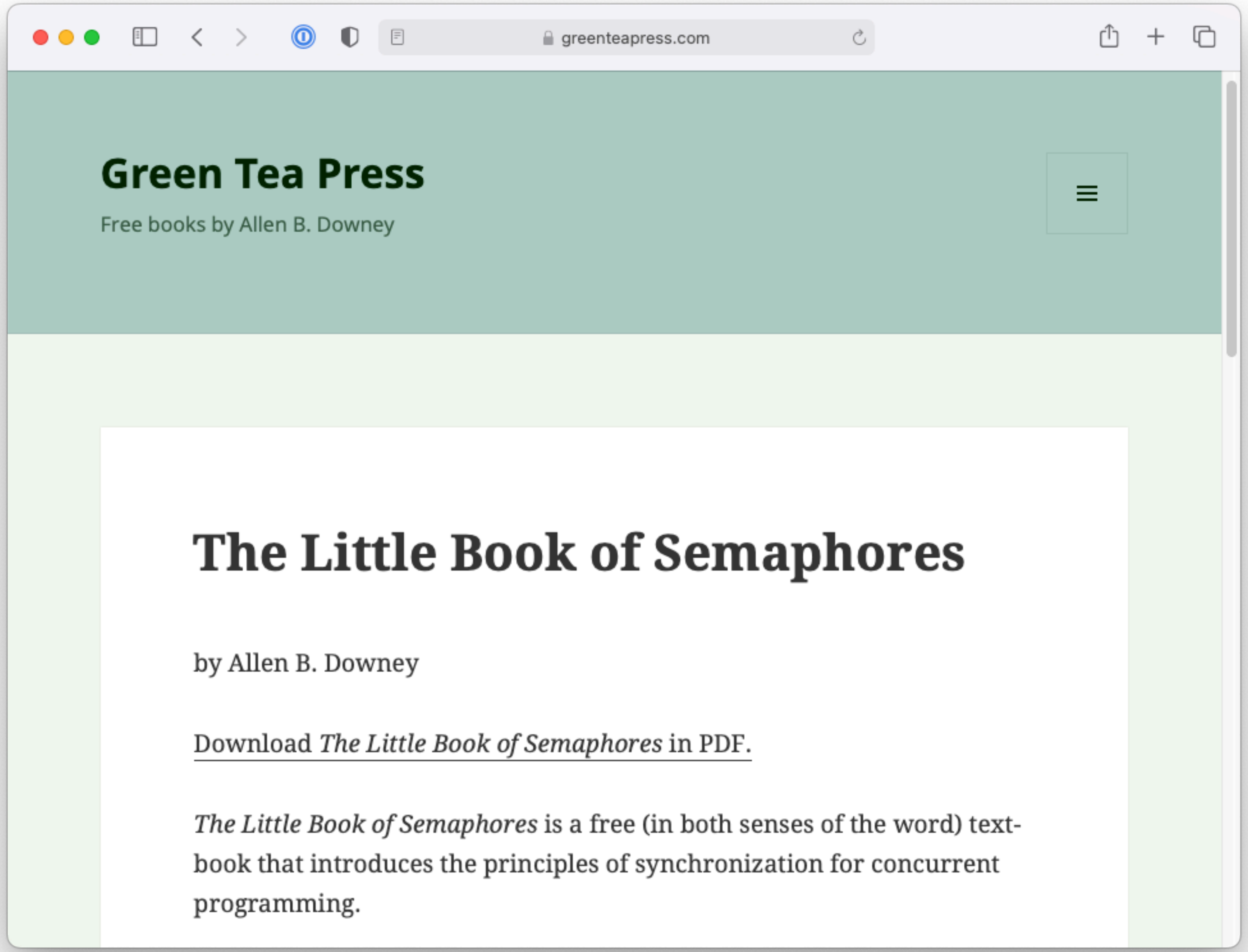
§ Semaphores



Reference: LBoS

- Little Book of Semaphores, by Allen Downey
  - Focuses on synchronization using semaphores
    - Includes classical and non-traditional problems
  - Lots of sample code in quasi-Python syntax



Semaphore rules

1. When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are allowed to perform are increment (increase by one) and decrement (decrease by one). You cannot read the current value of the semaphore.

2. When a thread decrements the semaphore, if the result is negative, the thread blocks itself and cannot continue until another thread increments the semaphore.

3. When a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked.

Initialization & Operations

```
fred = Semaphore(1)
```

Operation names?

```
fred.increment_and_wake_a_waiting_process_if_any()
fred.decrement_and_block_if_the_result_is_negative()
```

```
fred.increment()
fred.decrement()
```

```
fred.signal()
fred.wait()
```

```
fred.V()
fred.P()
```

Basic synchronization patterns

1. Rendezvous
2. Mutual exclusion (Mutex)
3. Multiplex
4. Generalized rendezvous
    - Barrier & Turnstile

1. Rendezvous

Problem: Ensure that `a1<b2`, `b1<a2`

Thread A

```
statement a1
statement a2
```

Thread B

```
statement b1
statement b2
```

rendezvous point

Hint: use the following variables

```
aArrived = Semaphore(0)
bArrived = Semaphore(0)
```

```
aArrived = Semaphore(0)
bArrived = Semaphore(0)
```

1. Rendezvous

```
aArrived = Semaphore(0)
bArrived = Semaphore(0)
```

Thread A

```
statement a1
aArrived.signal()
bArrived.wait()
statement a2
```

Thread B

```
statement b1
bArrived.signal()
aArrived.wait()
statement b2
```

2. Mutual exclusion

Problem: Ensure that critical sections do not overlap

Thread A

```
count = count + 1
```

Thread B

```
count = count + 1
```

Hint: use the following variable

```
mutex = Semaphore(1)
```

2. Mutual exclusion

```
mutex = Semaphore(1)
```

Thread A

```
mutex.wait()
    # critical section
    count = count + 1
mutex.signal()
```

Thread B

```
mutex.wait()
    # critical section
    count = count + 1
mutex.signal()
```

3. Multiplex

```
multiplex = Semaphore(N)
```

```
multiplex.wait()
    critical section
multiplex.signal()
```

Permits `N` threads through into their critical sections

4. Generalized Rendezvous

Problem: Generalize the rendezvous solution. Every thread should run the following code

```
rendezvous
critical point
```

Hint: use the following variables

```
n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

```
n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

4. Generalized Rendezvous

```
rendezvous

mutex.wait()
    count = count + 1
mutex.signal()

if count == n: barrier.signal()

barrier.wait()
barrier.signal()

critical point
```

“turnstile”

② open the floodgates

① threads pile up here

③ each thread signals another

4. Generalized Rendezvous

```
rendezvous

mutex.wait()
    count = count + 1
mutex.signal()

if count == n: turnstile.signal()

turnstile.wait()
turnstile.signal()

critical point
```

what is the value of `turnstile` when all threads reach the critical point?

4. Generalized Rendezvous

threads may be preempted here

between 1 and n threads signal here

balanced # of waits/signals

```
rendezvous

mutex.wait()
    count = count + 1
mutex.signal()

if count == n: turnstile.signal()

turnstile.wait()
turnstile.signal()

critical point
```

value of `turnstile` is in range [1,n]

can we eliminate this non-determinism?

4. Generalized Rendezvous

```
rendezvous

mutex.wait()
    count = count + 1
    if count == n: turnstile.signal()
mutex.signal()

turnstile.wait()
turnstile.signal()

critical point
```

value of `turnstile` at critical point is predictably 1
(but it is no longer a usable barrier)

4. Generalized Rendezvous

```
rendezvous

mutex.wait()
    count += 1
    if count == n: turnstile.signal()
mutex.signal()

turnstile.wait()
turnstile.signal()

critical point

mutex.wait()
    count -= 1
    if count == 0: turnstile.wait()
mutex.signal()
```

turnstile reset

A thread may “lap” the other threads, mess up `count`, and fail to block before the critical point the next time.

does this work reliably?

4. Generalized Rendezvous

Problem: Build a generalized, *reusable* rendezvous solution — i.e., where threads all rendezvous again after each time through the CS

Hint: use the following variables

```
turnstile = Semaphore(0)
turnstile2 = Semaphore(1)
mutex = Semaphore(1)
```

```
turnstile = Semaphore(0)
turnstile2 = Semaphore(1)
mutex = Semaphore(1)
```

4. Generalized Rendezvous

lock the second turnstile, unlock the first

first turnstile

lock the first turnstile, unlock the second

second turnstile

```
# rendezvous

mutex.wait()
    count += 1
    if count == n:
        turnstile2.wait()
        turnstile.signal()
mutex.signal()

turnstile.wait()
turnstile.signal()

# critical point

mutex.wait()
    count -= 1
    if count == 0:
        turnstile.wait()
        turnstile2.signal()
mutex.signal()

turnstile2.wait()
turnstile2.signal()
```

“Barrier” type

```
class Barrier:
    def __init__(self, n):
        self.n = n
        self.count = 0
        self.mutex = Semaphore(1)
        self.turnstile = Semaphore(0)
        self.turnstile2 = Semaphore(1)

    def phase1(self);
        self.mutex.wait()
            self.count += 1
            if self.count == self.n:
                self.turnstile2.wait()
                self.turnstile.signal()
        self.mutex.signal()
        self.turnstile.wait()
        self.turnstile.signal()
```

```
    def phase2(self):
        self.mutex.wait()
            self.count -= 1
            if self.count == 0:
                self.turnstile.wait()
                self.turnstile2.signal()
        self.mutex.signal()
        self.turnstile2.wait()
        self.turnstile2.signal()

    def wait(self):
        self.phase1()
        self.phase2()
```

Classical synchronization problems

1. Producer/Consumer
2. Readers/Writers
3. Dining Philosophers

1. Producer/Consumer (revisited)

Problem: producer & consumer threads repeatedly accessing a finite, non-thread-safe buffer

```
# Producer
item = produce()
buffer.put(item)
```

```
# Consumer
item = buffer.get()
consume(item)
```

Hint: use the following variables

```
mutex  = Semaphore(1)
items  = Semaphore(0)
spaces = Semaphore(buffer.capacity())
```

```
mutex  = Semaphore(1)
items  = Semaphore(0)
spaces = Semaphore(buffer.capacity())
```

1. Producer/Consumer (revisited)

```
# Producer

item = produce()

spaces.wait()

mutex.wait()
    buffer.put(item)
mutex.signal()

items.signal()
```

```
# Consumer

items.wait()

mutex.wait()
    item = buffer.get()
mutex.signal()

spaces.signal()

consume(item)
```

2. Readers/Writers

Problem: unlimited # of readers allowed to access shared resource at once, but at most one writer; no readers while writer is accessing resource

- i.e., **categorical** mutex

- can model access to the resource as a "room", where any # of readers may occupy the room, but it must be vacated for a single writer to enter

Hint: use the following variables

```
n_readers = 0
mutex     = Semaphore(1)
roomEmpty = Semaphore(1)
```

```
n_readers = 0
mutex     = Semaphore(1)
roomEmpty = Semaphore(1)
```

2. Readers/Writers

first reader blocks on `roomEmpty` and bars other readers from crossing `mutex`

```
# Writers

roomEmpty.wait()

    # critical section

roomEmpty.signal()
```

```
# Readers

mutex.wait()
    n_readers += 1
    if n_readers == 1:
        roomEmpty.wait()
mutex.signal()

    # critical section

mutex.wait()
    n_readers -= 1
    if n_readers == 0:
        roomEmpty.signal()
mutex.signal()
```

"Lightswitch" pattern

```
class Lightswitch:
    def __init__(self):
        self.counter = 0
        self.mutex = Semaphore(1)

    def lock(self, switch):
        self.mutex.wait()
            self.counter += 1
            if self.counter == 1:
                switch.wait()
        self.mutex.signal()

    def unlock(self, switch):
        self.mutex.wait()
            self.counter -= 1
            if self.counter == 0:
                switch.signal()
        self.mutex.signal()
```

- Encapsulates "first-in locks, last-out unlocks" synchronization semantic

```
roomEmpty   = Semaphore(1)
readSwitch = Lightswitch()
```

```
# Writers
roomEmpty.wait()
    # critical section
roomEmpty.signal()
```

```
# Readers
readSwitch.lock(roomEmpty)
    # critical section
readSwitch.unlock(roomEmpty)
```

2. Readers/Writers with Lightswitch

```
roomEmpty  = Semaphore(1)
readSwitch = Lightswitch()
```

```
# Writers
roomEmpty.wait()
    # critical section
roomEmpty.signal()
```

```
# Readers
readSwitch.lock(roomEmpty)
    # critical section
readSwitch.unlock(roomEmpty)
```

- Problem: a constant stream of readers into the room may starve writers!
  - How to guarantee entry into room for a newly arrived writer?
- Hint:

```
roomEmpty  = Semaphore(1)
readSwitch = Lightswitch()
turnstile  = Semaphore(1)
```

```
roomEmpty  = Semaphore(1)
readSwitch = Lightswitch()
turnstile  = Semaphore(1)
```

2. No-starve Readers/Writers

```
roomEmpty   = Semaphore(1)
readSwitch = Lightswitch()
turnstile   = Semaphore(1)
```

writer blocks `turnstile` while waiting on `roomEmpty`, preventing readers from filing into room

```
# Writers

turnstile.wait()

roomEmpty.wait()
    turnstile.signal()
    # critical section
roomEmpty.signal()
```

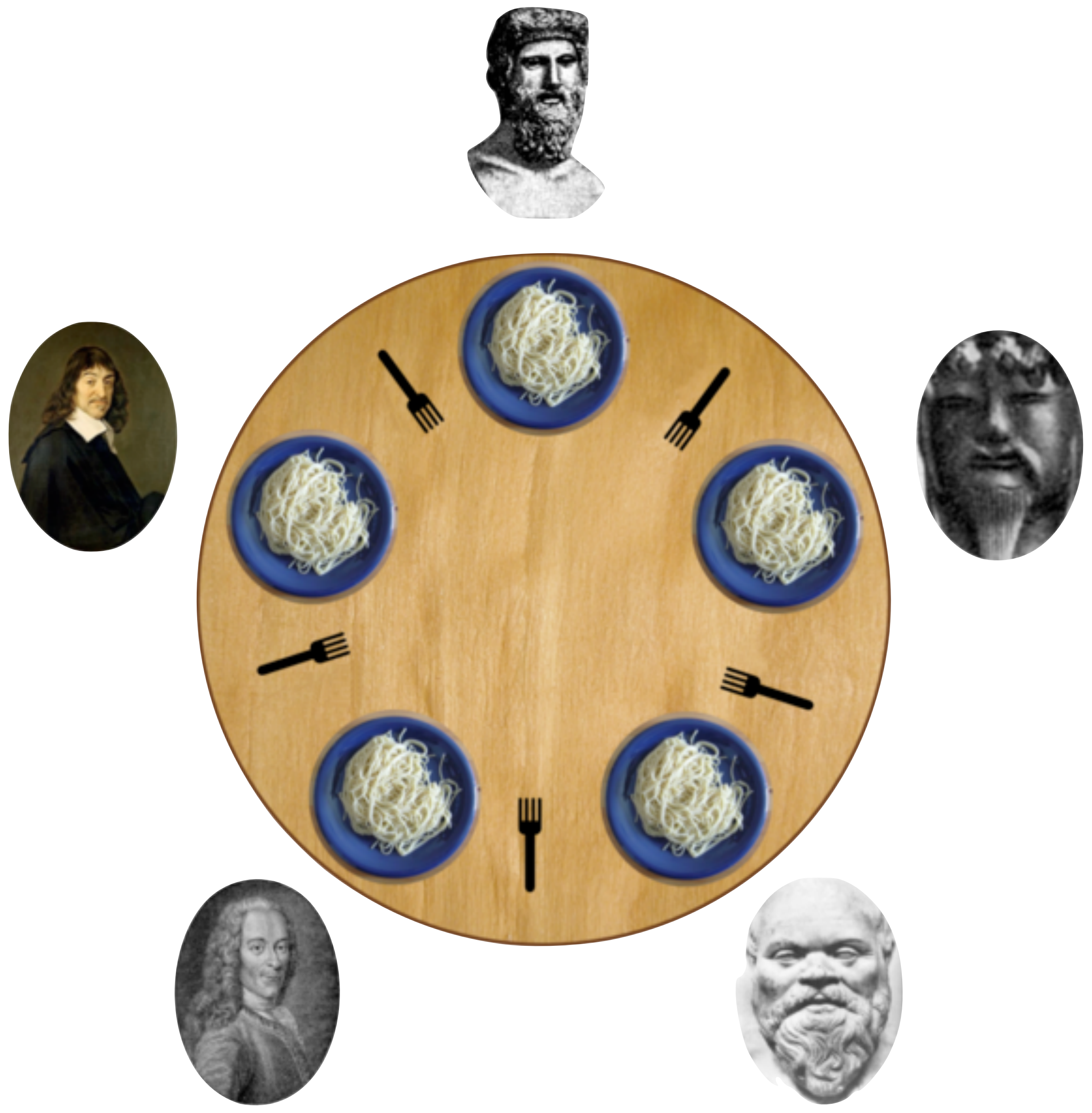
```
# Readers

turnstile.wait()
turnstile.signal()

readSwitch.lock(roomEmpty)
    # critical section
readSwitch.unlock(roomEmpty)
```

when last reader leaves the room,
writer enters and releases `turnstile`

3. Dining Philosophers

Problem: Philosophers are seated about a round table, each with a plate of spaghetti in front of, and a fork to either side of them — adjacent philosophers share a fork

- Philosophers alternate between thinking and eating
- To eat, a philosopher needs to use both forks
  - A fork can only be in use by one philosopher
- Philosophers should not be starved (of spaghetti), and cannot predict how others will behave

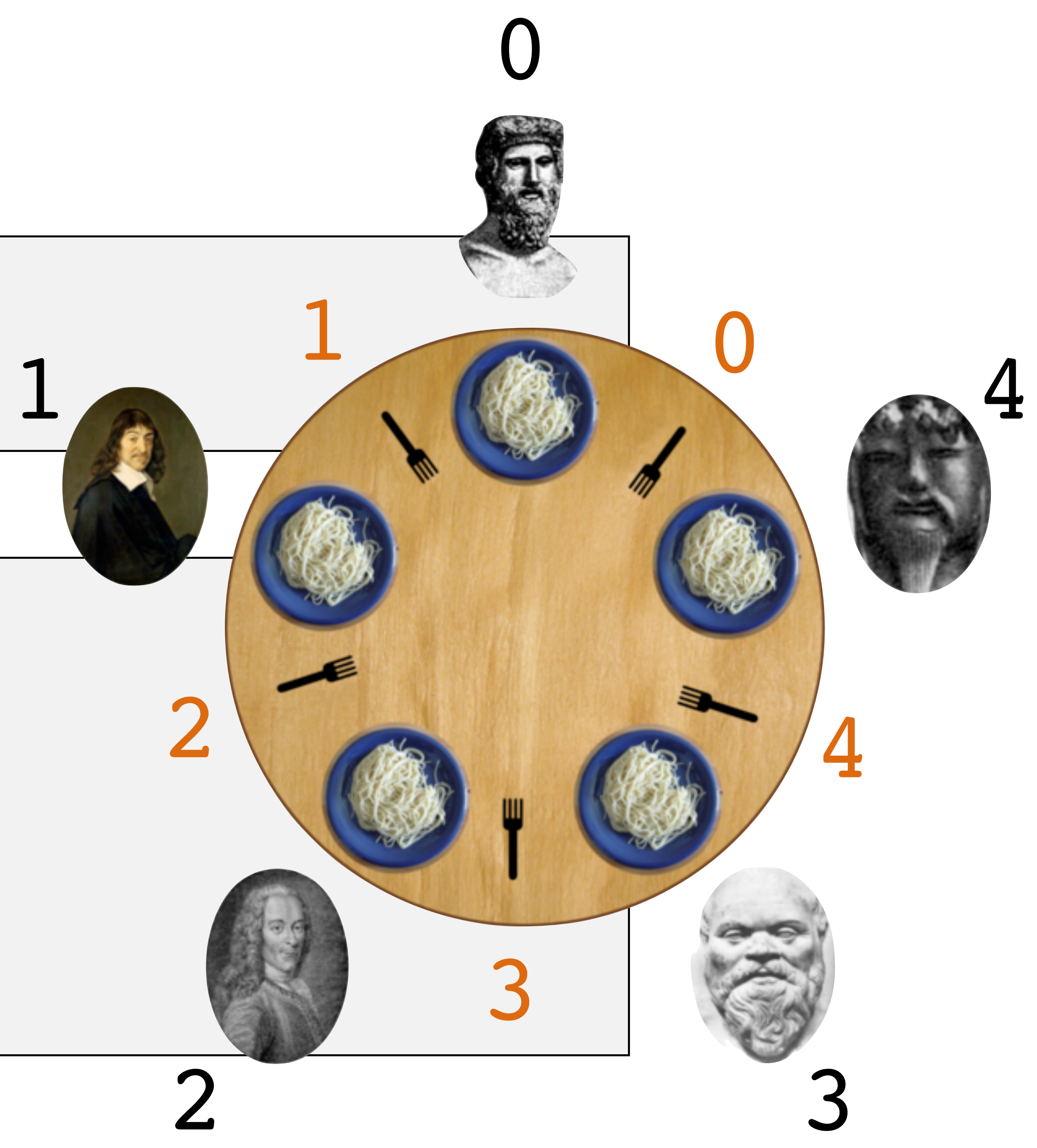


3. Dining Philosophers

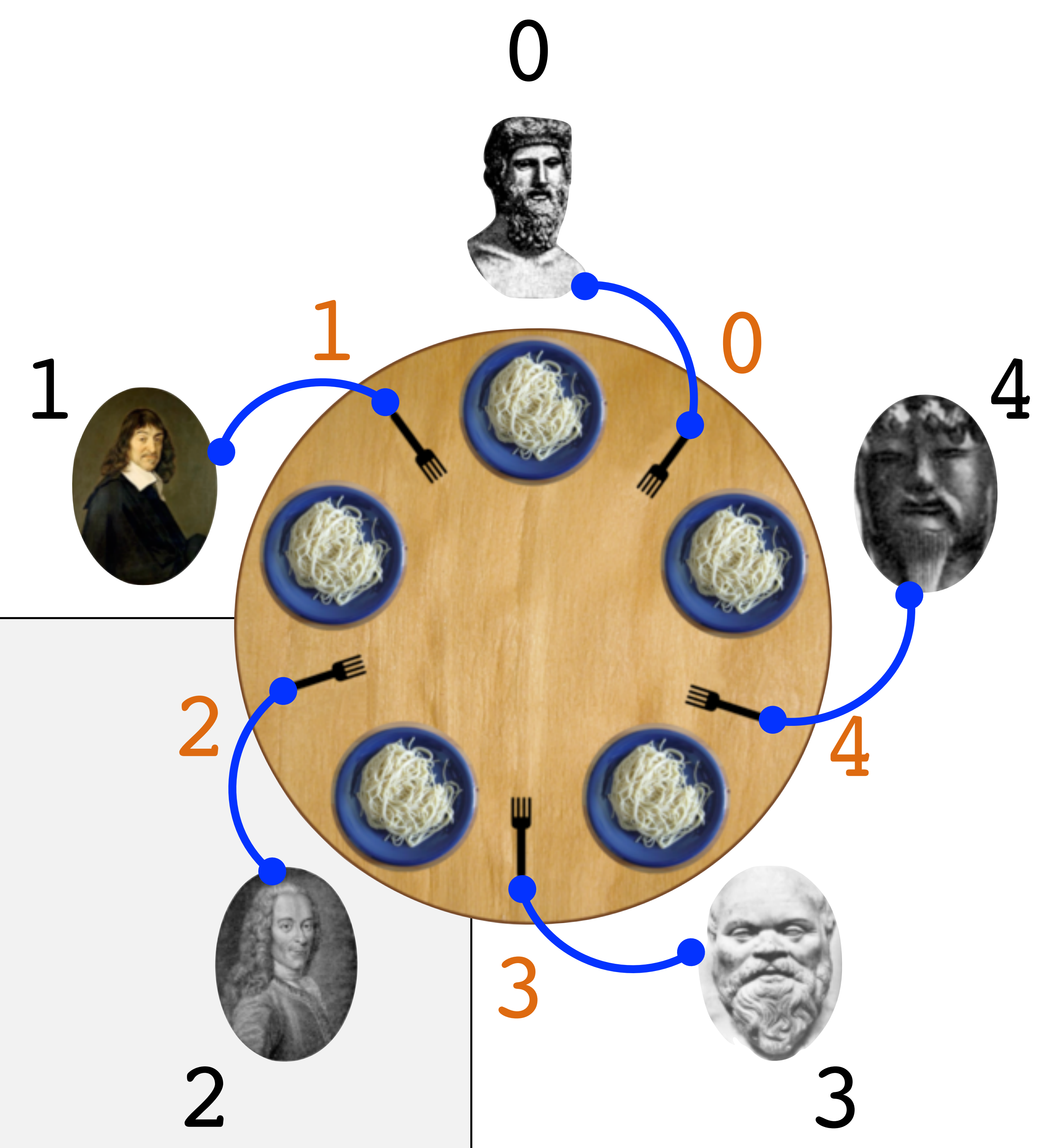
- Simple setup: model forks as semaphores

```
forks = [Semaphore(1) for i in range(5)]
```

```
# philosopher id → fork id mapping functions

def left(i): return i
def right(i): return (i + 1) % 5
```



3. DP: Naive solution




```
def get_forks(i):
    fork[left(i)].wait()
    fork[right(i)].wait()

def put_forks(i):
    fork[left(i)].signal()
    fork[right(i)].signal()
```

- Potential deadlock! All philosophers obtain left fork and starve

3. DP: Global mutex

```
def get_forks(i):
    mutex.wait()
        fork[left(i)].wait()
        fork[right(i)].wait()
    mutex.signal()

def put_forks(i):
    fork[left(i)].signal()
    fork[right(i)].signal()
```

- May prohibit a philosopher from eating when their forks are available
  - Fails to maximize concurrency

3. DP: Thread limit

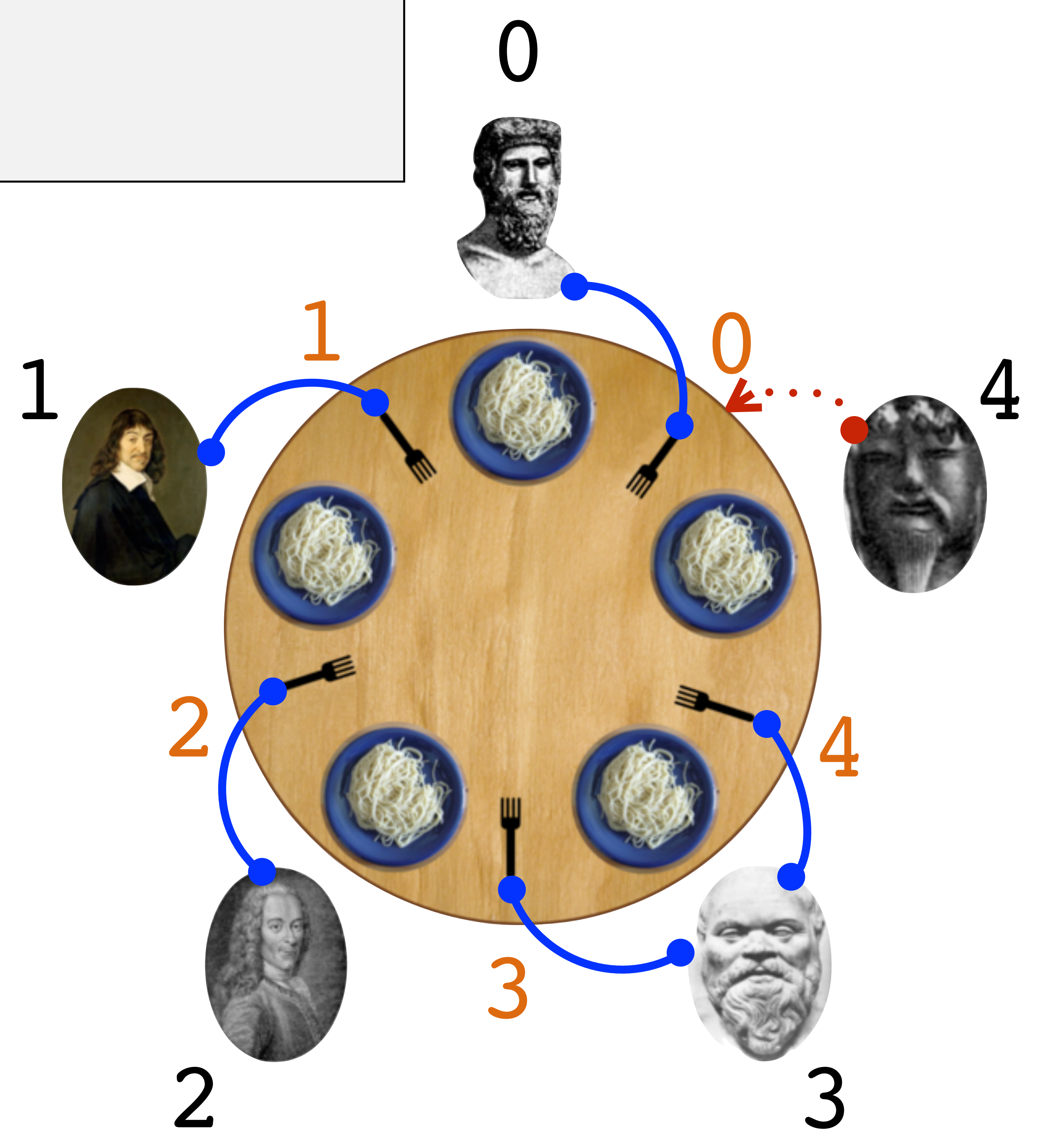
```
footman = Semaphore(4)
```

```
def get_forks(i):
    footman.wait()
    fork[left(i)].wait()
    fork[right(i)].wait()

def put_forks(i):
    fork[left(i)].signal()
    fork[right(i)].signal()
    footman.signal()
```

- How realistic is this approach?

3. DP: Resource ordering

```
def get_forks(i):
    for i in sorted([left(i), right(i)]):
        fork[i].wait()
```



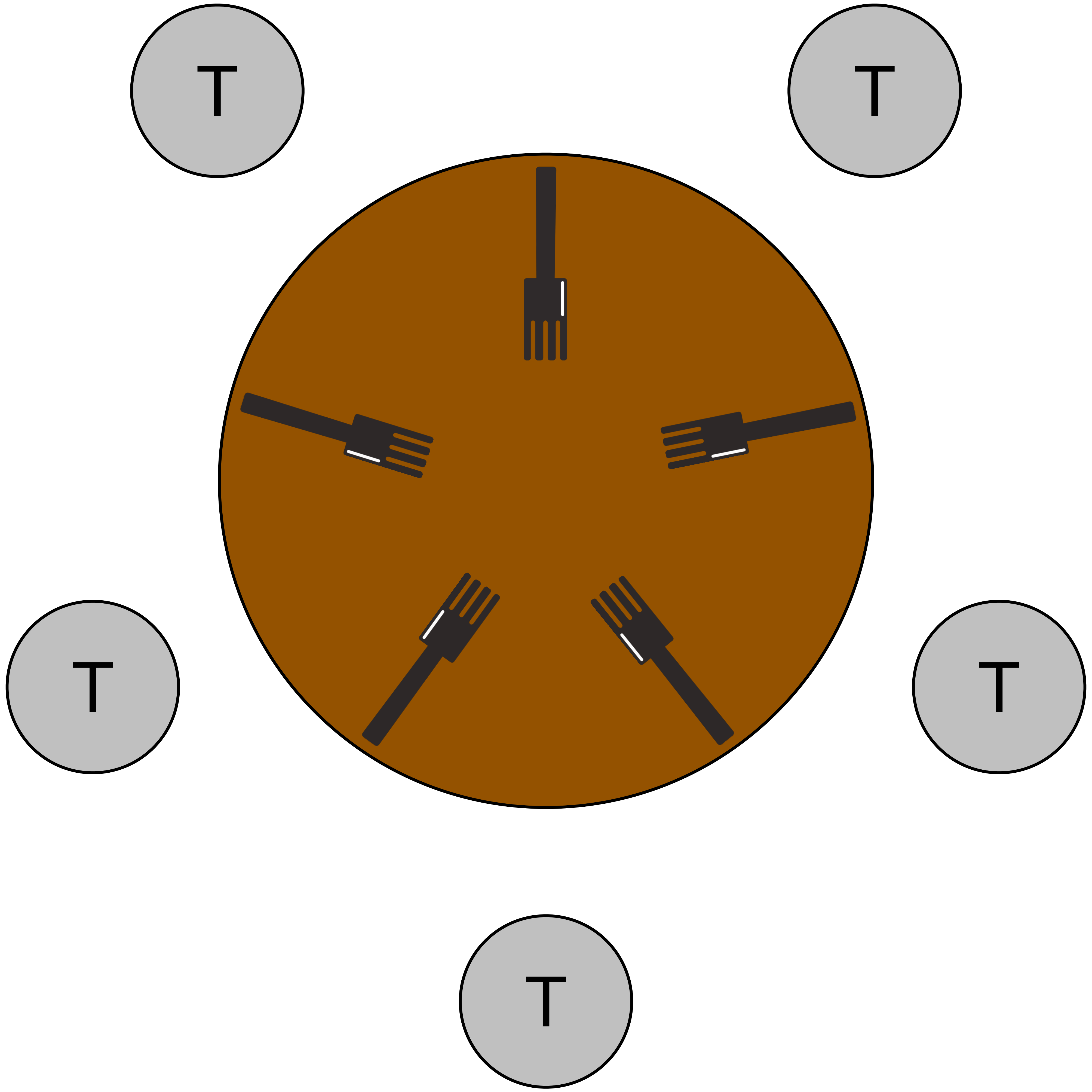
- Order all required resources and request only in increasing order
  - Prevents a cycle in the *resource allocation graph*
- How realistic is this approach?

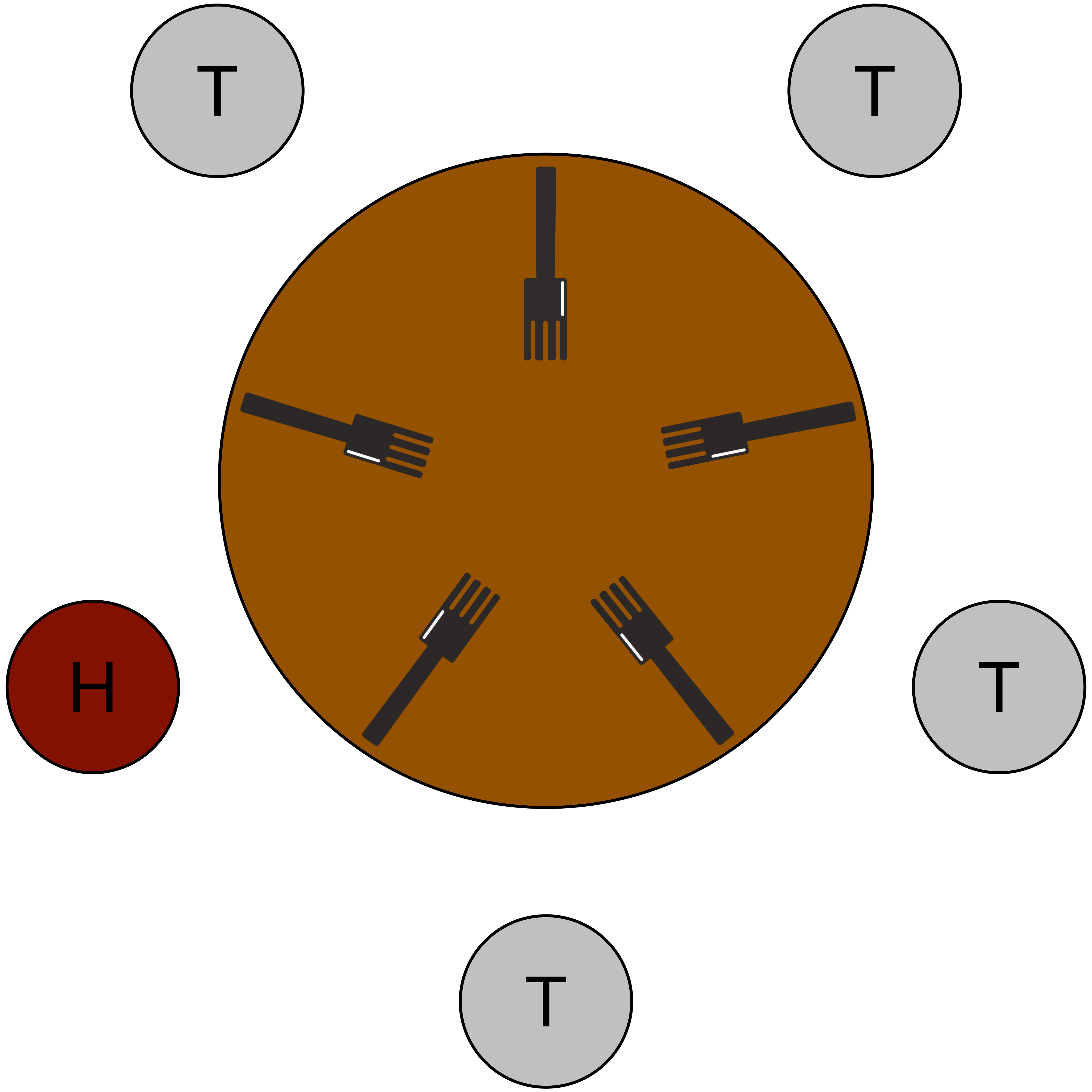
3. DP: Tanenbaum's solution

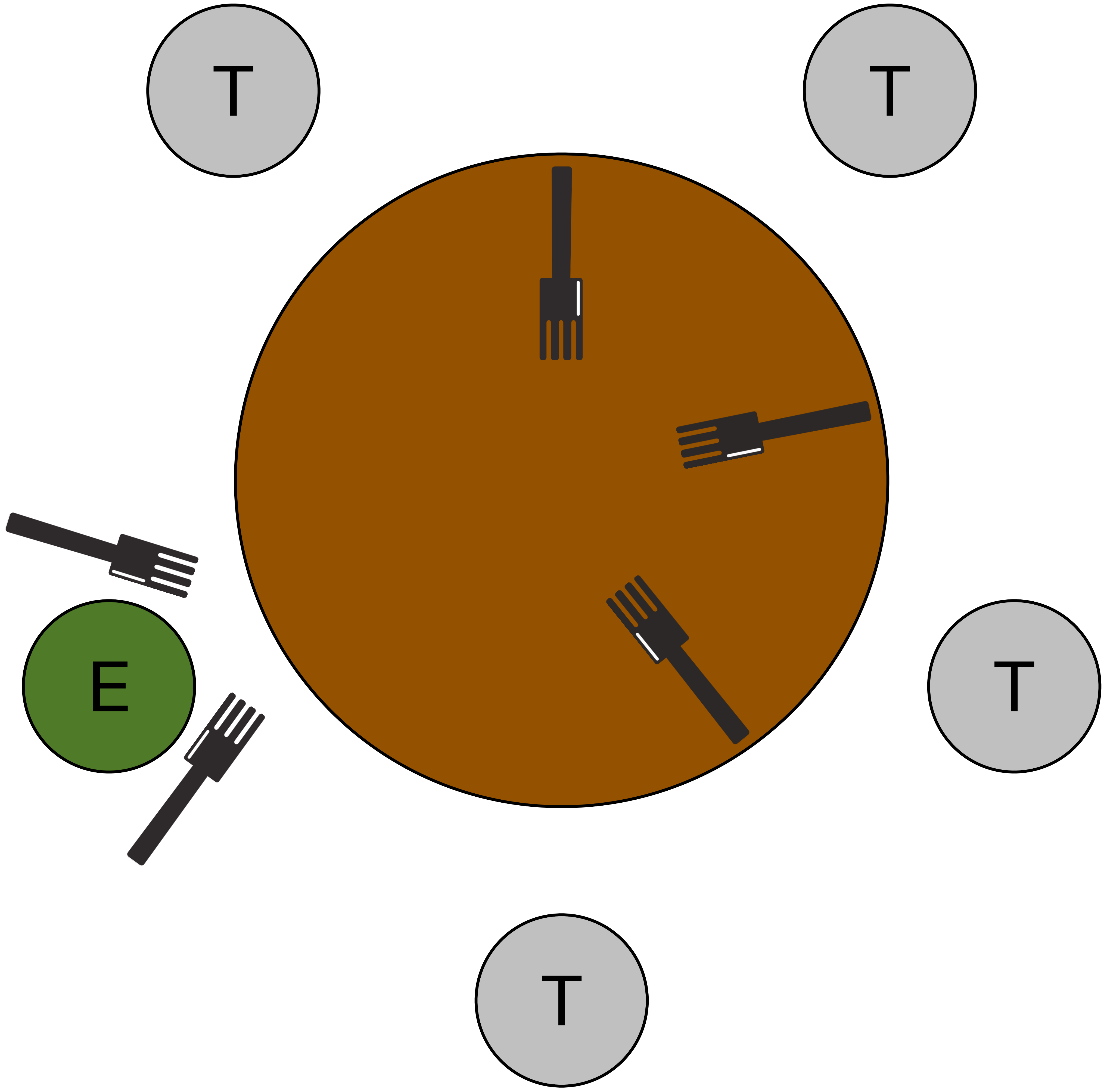
- Idea: philosophers announce their state ∈ {thinking, eating, hungry}
  - Can only eat if neighbors are both not eating
  - When done eating, check if neighbor is hungry and help them eat, if possible

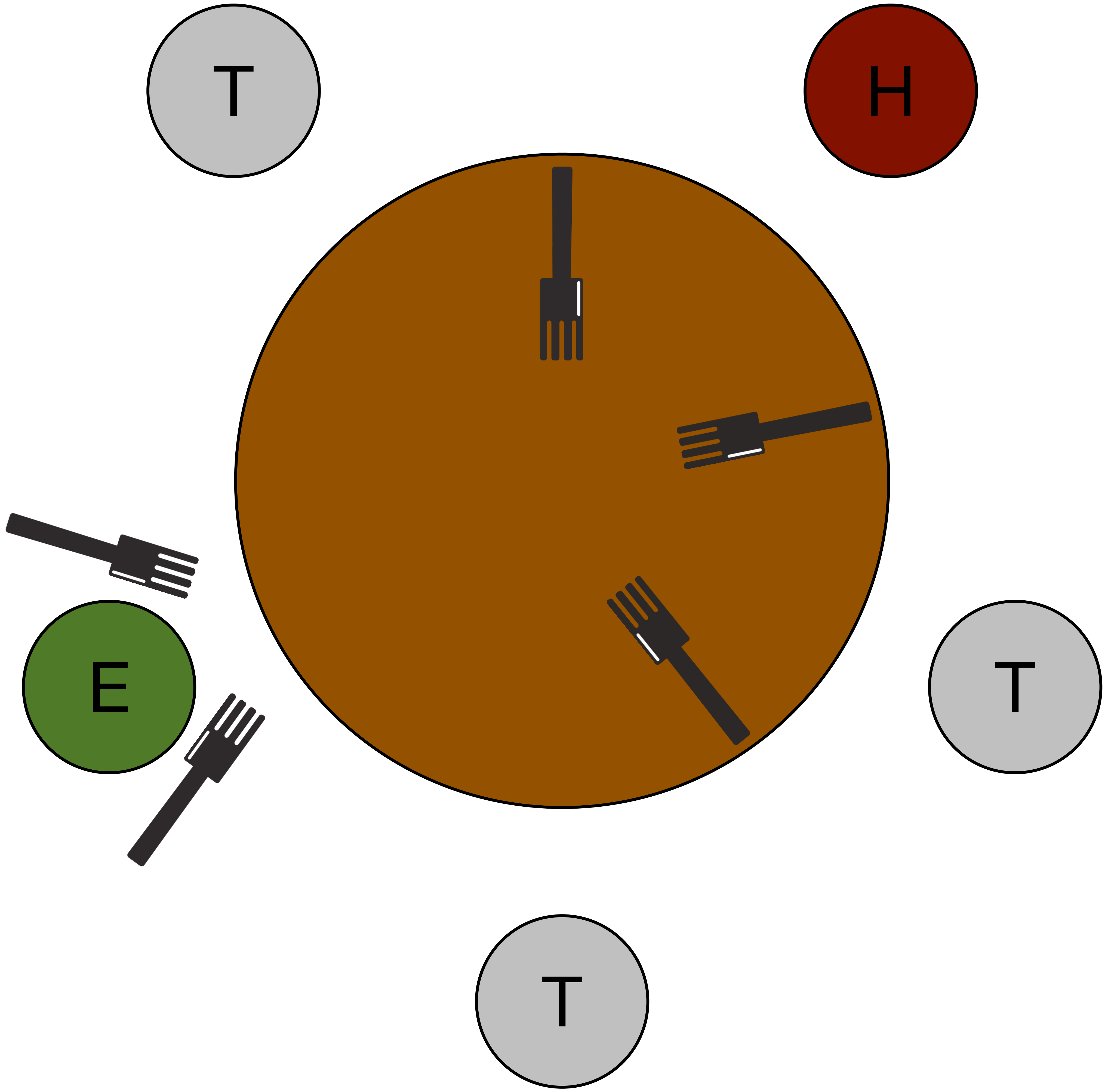
```
state = ['thinking'] * 5
sem = [Semaphore(0) for i in range(5)]
mutex = Semaphore(1)
```

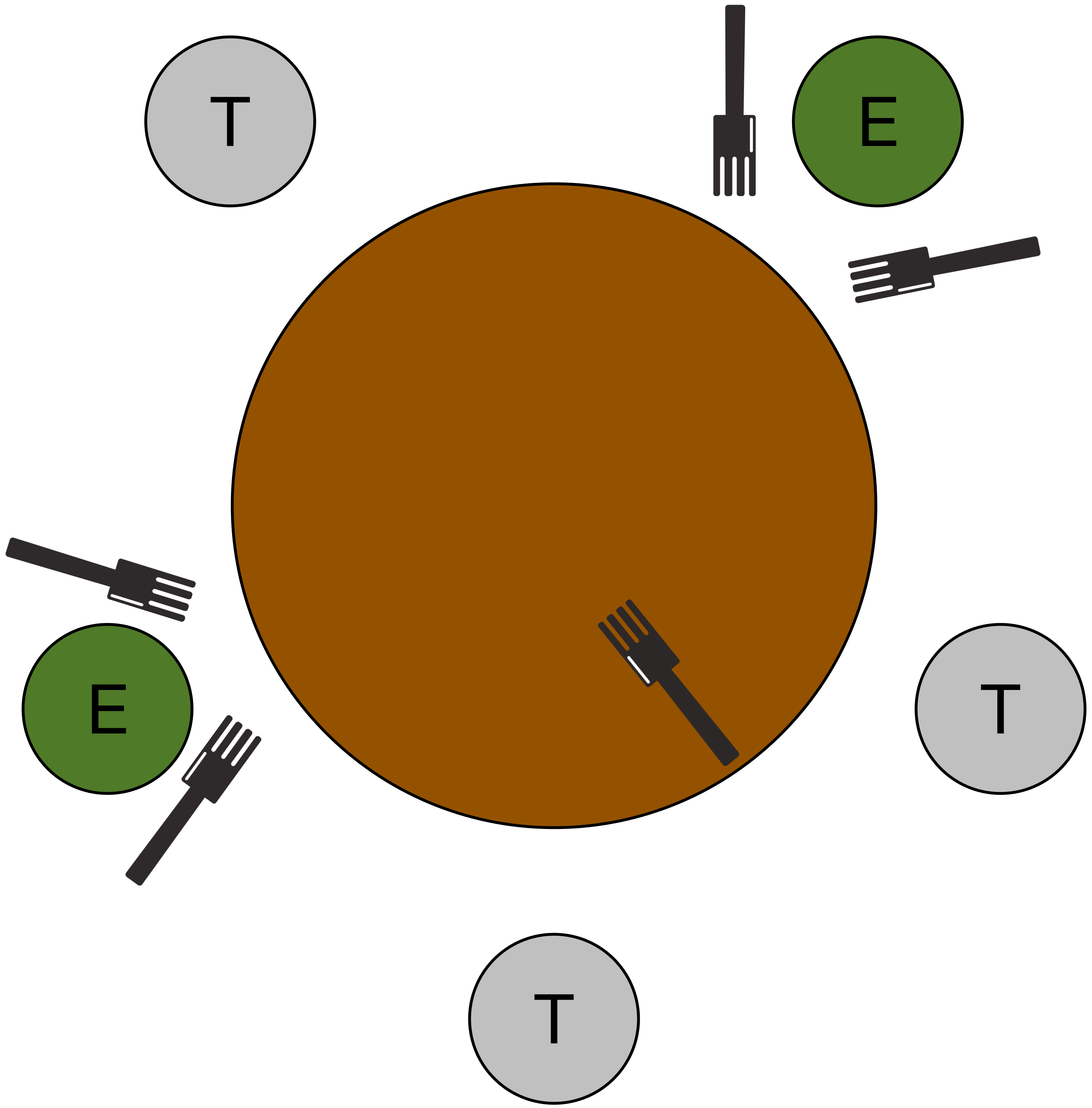
```
def get_fork(i):
    mutex.wait()
        state[i] = 'hungry'
        test(i)
    mutex.signal()
    sem[i].wait()

def put_fork(i):
    mutex.wait()
        state[i] = 'thinking'
        test(right(i))
        test(left(i))
    mutex.signal()

def test(i):
    if state[i] == 'hungry' \
       and state[left(i)] != 'eating' \
       and state[right(i)] != 'eating':
        state[i] = 'eating'
        sem[i].signal()
```

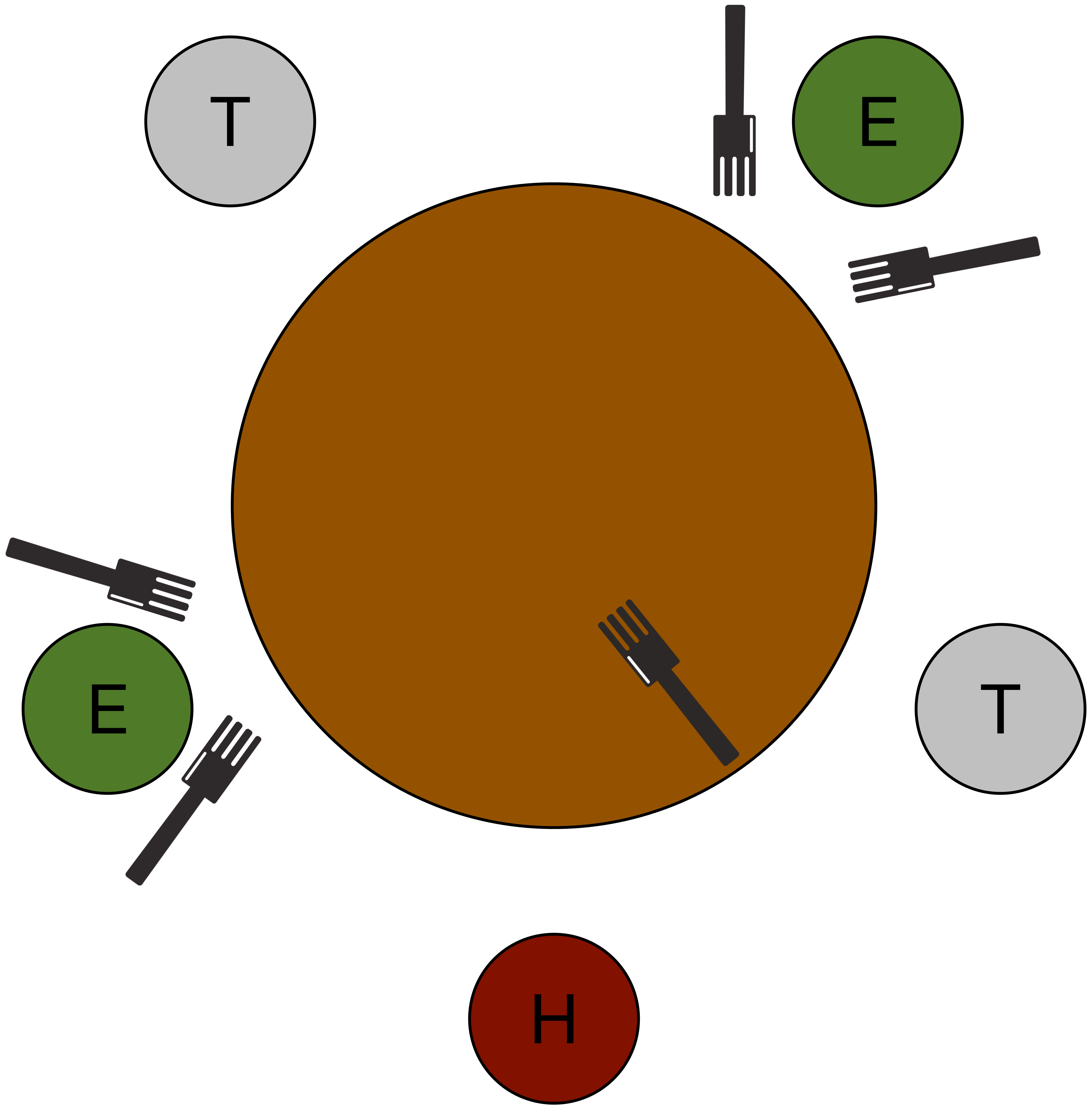

T
T
T
T
T

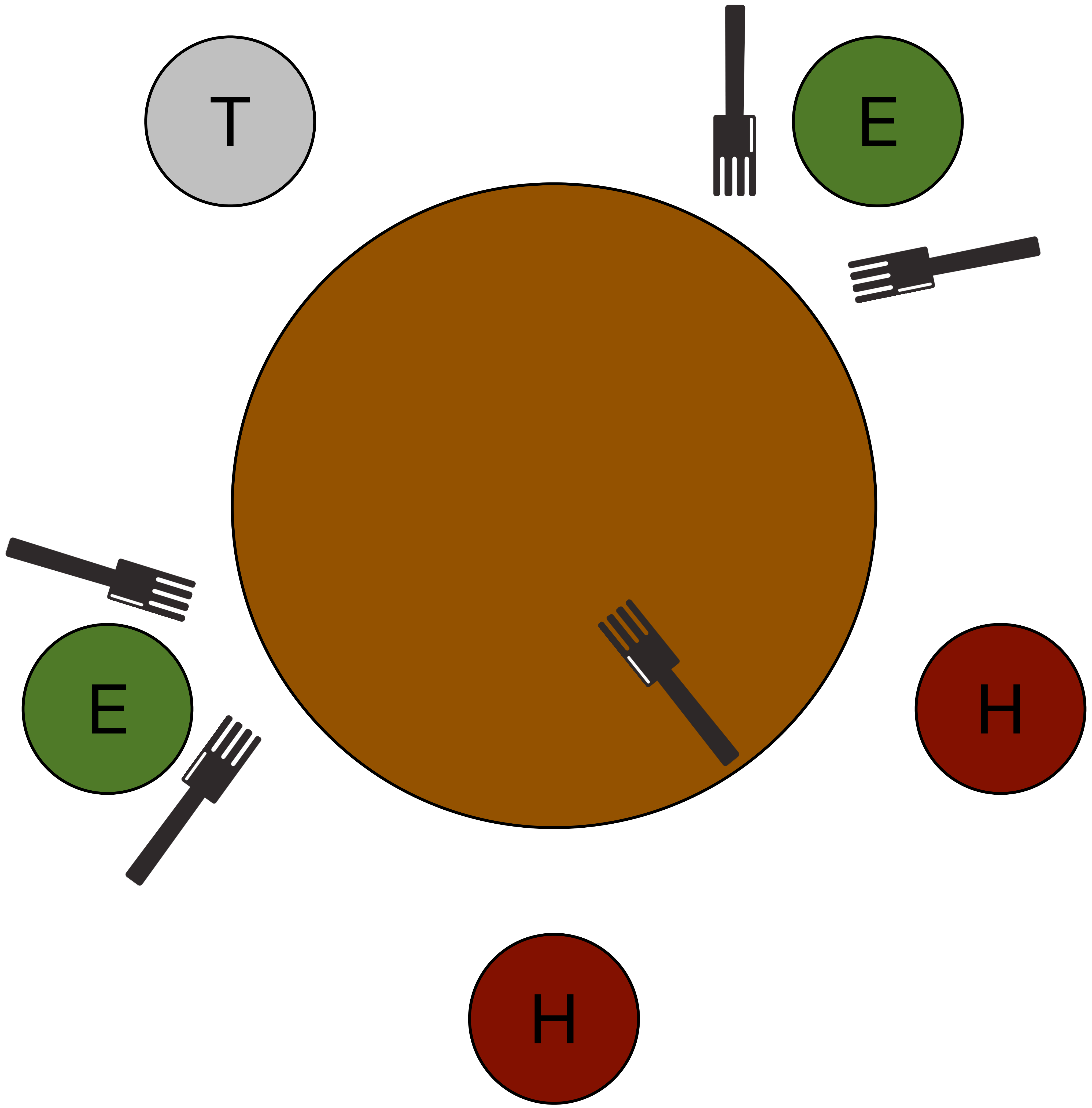

T
T
H
T
T

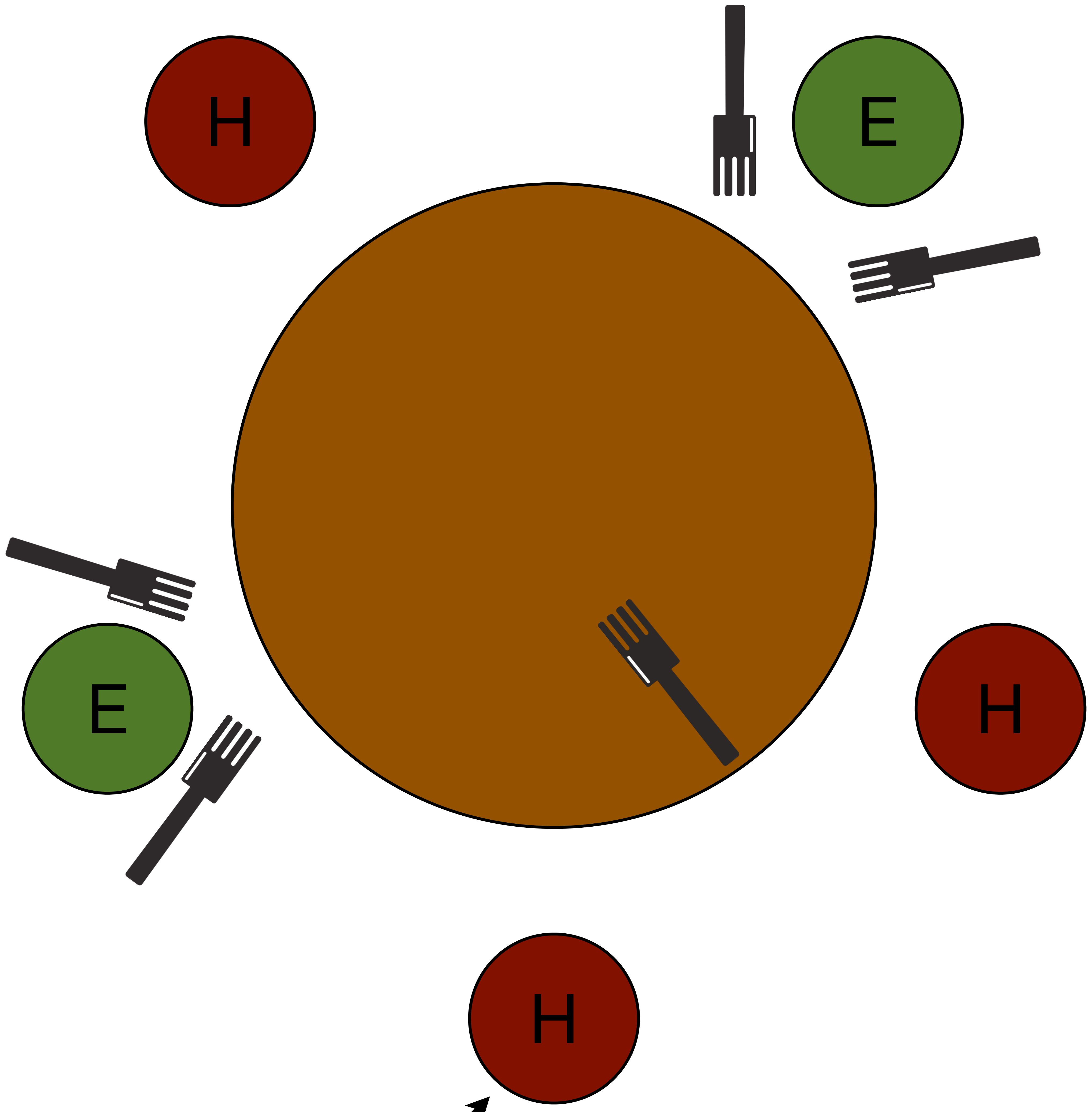

T
T
E
T
T


T
H
E
T
T

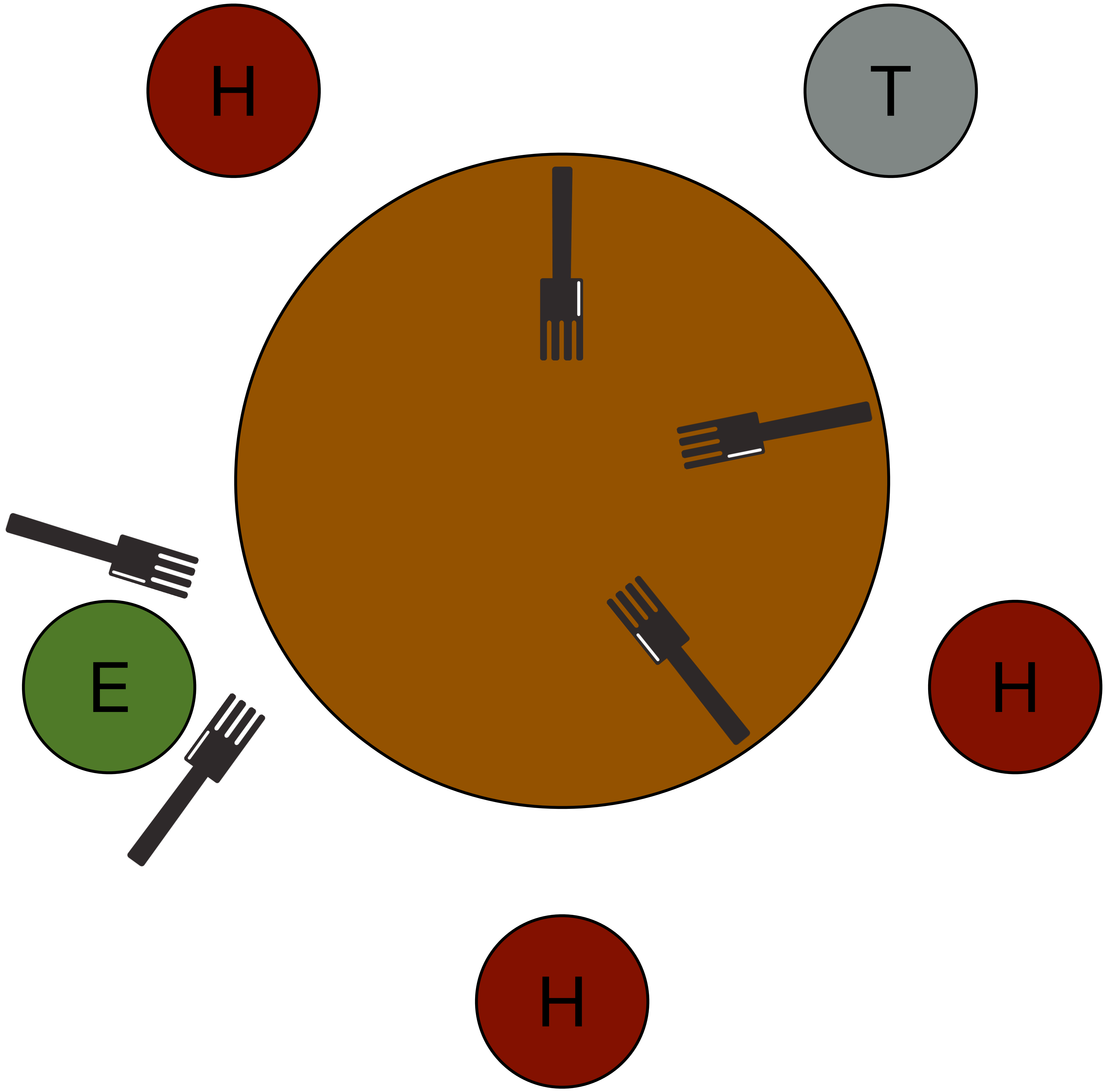

T
E
E
T
T

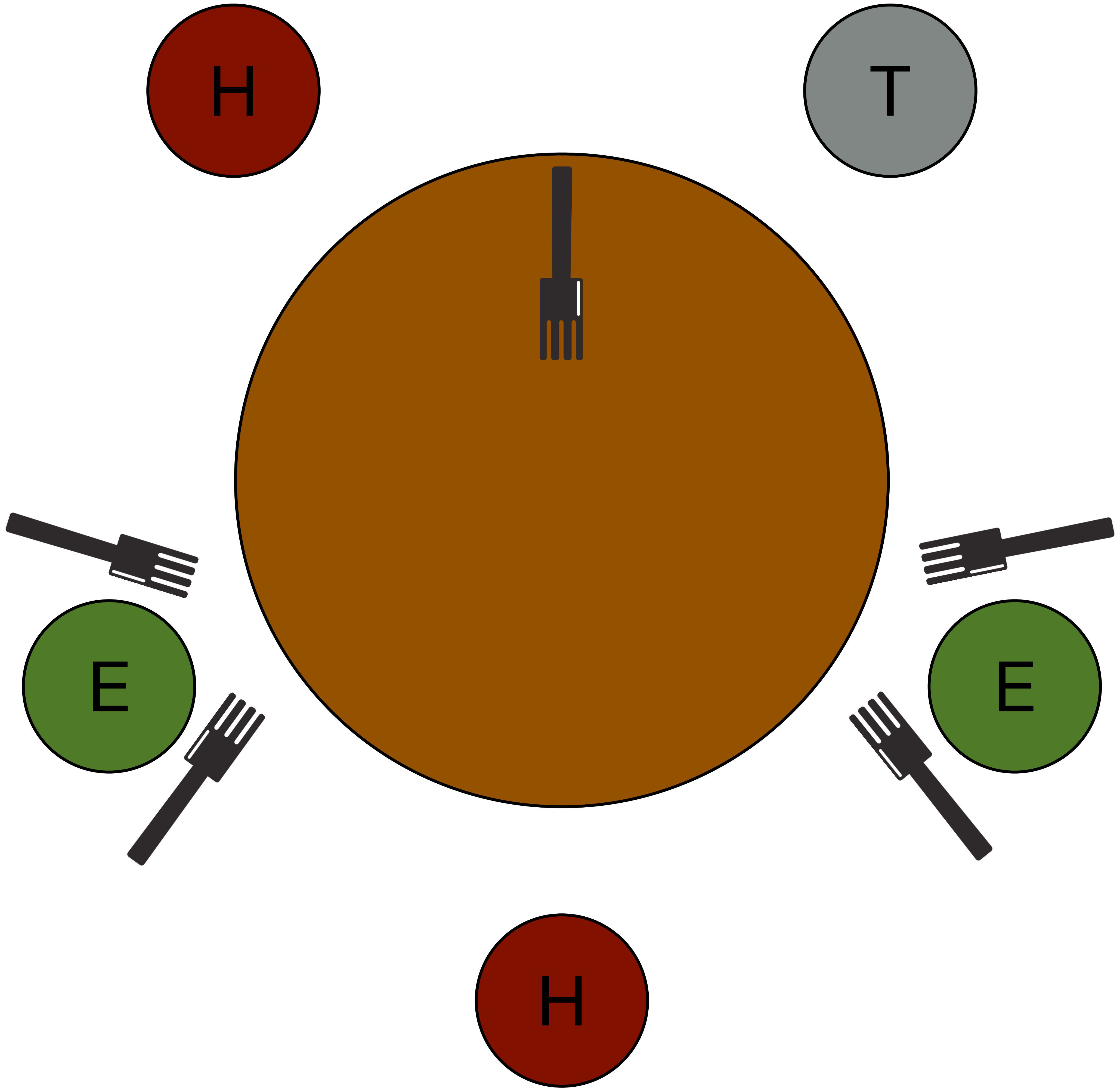

T
E
E
T
H

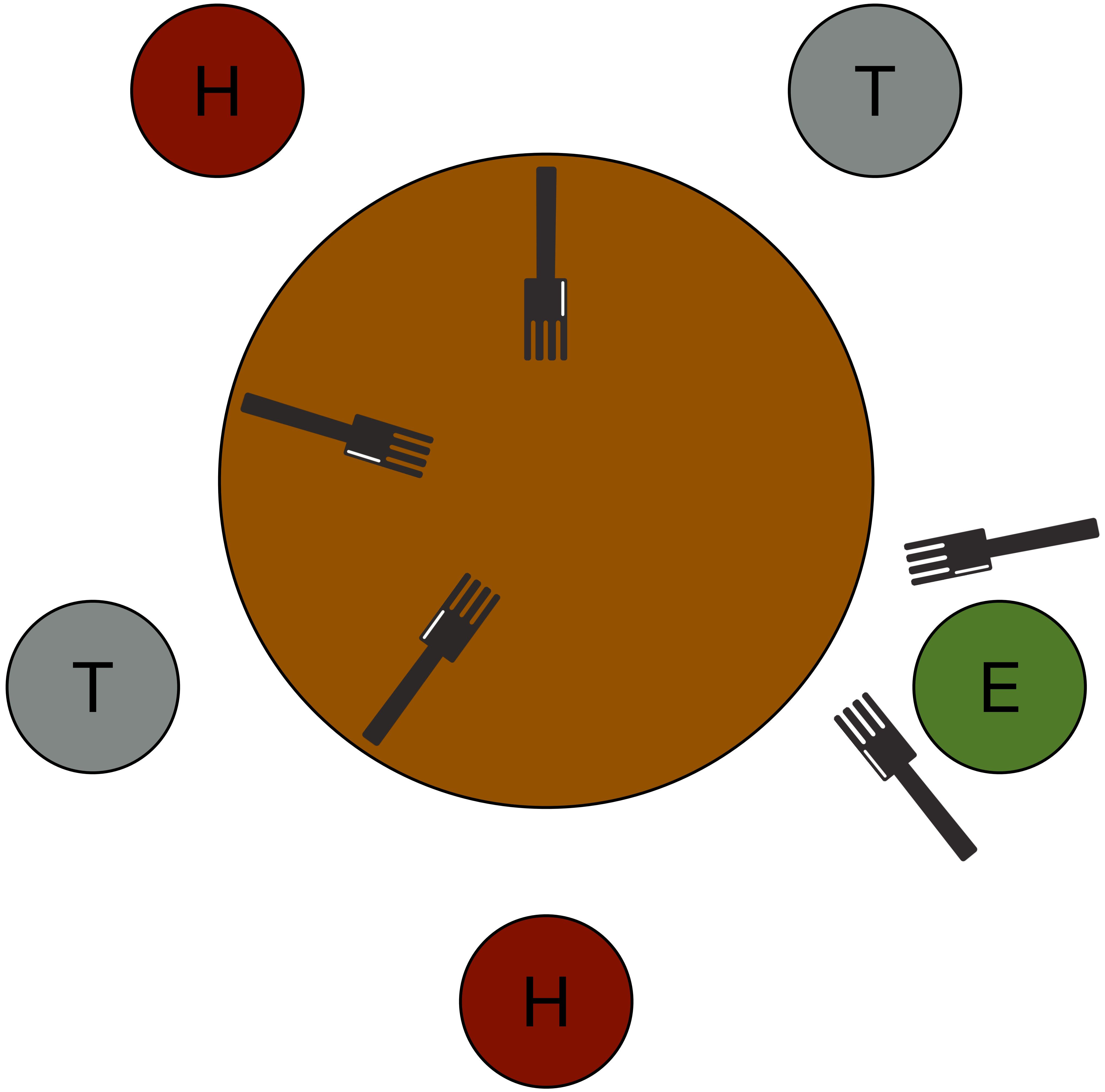

T
E
E
H
H

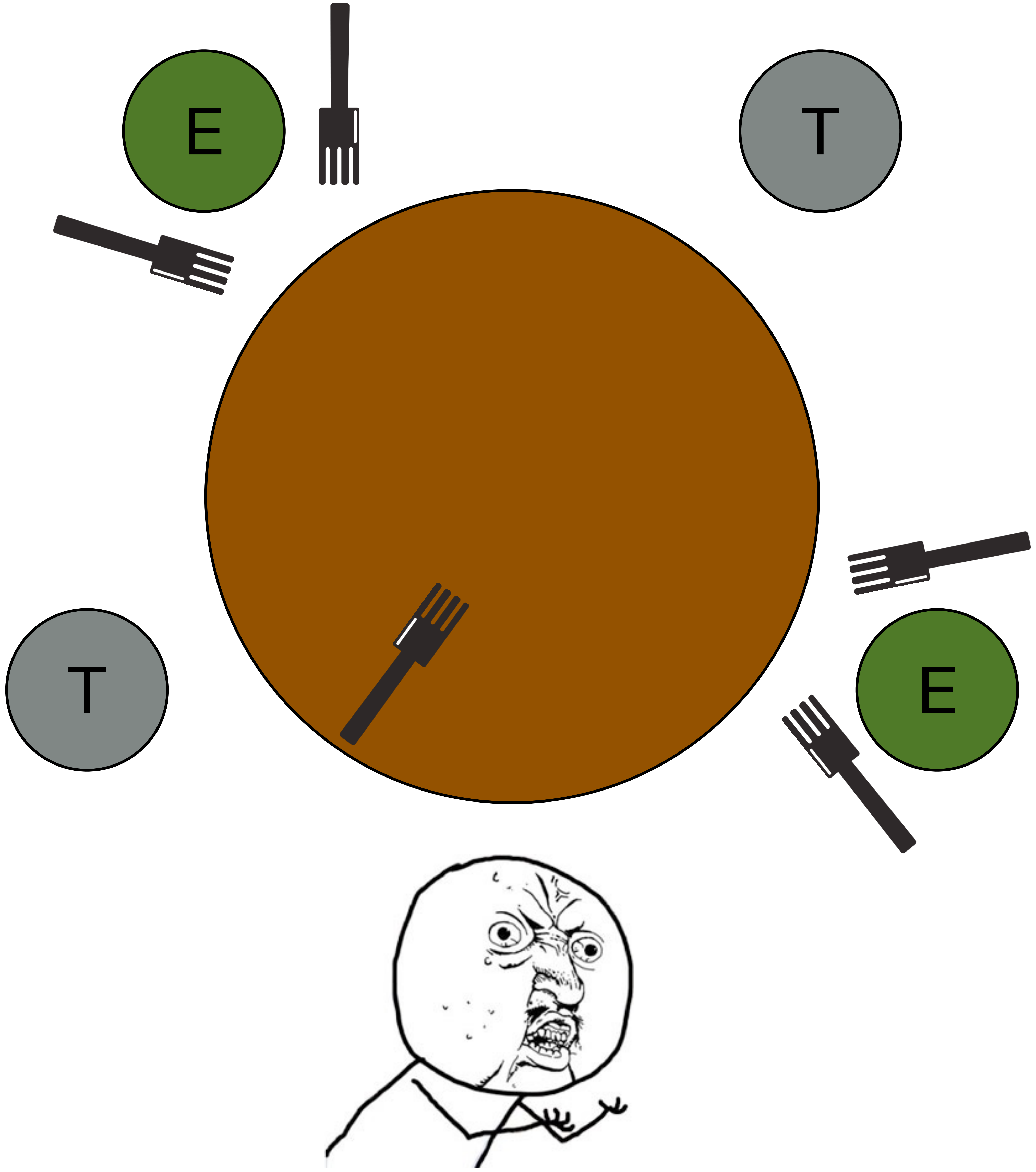


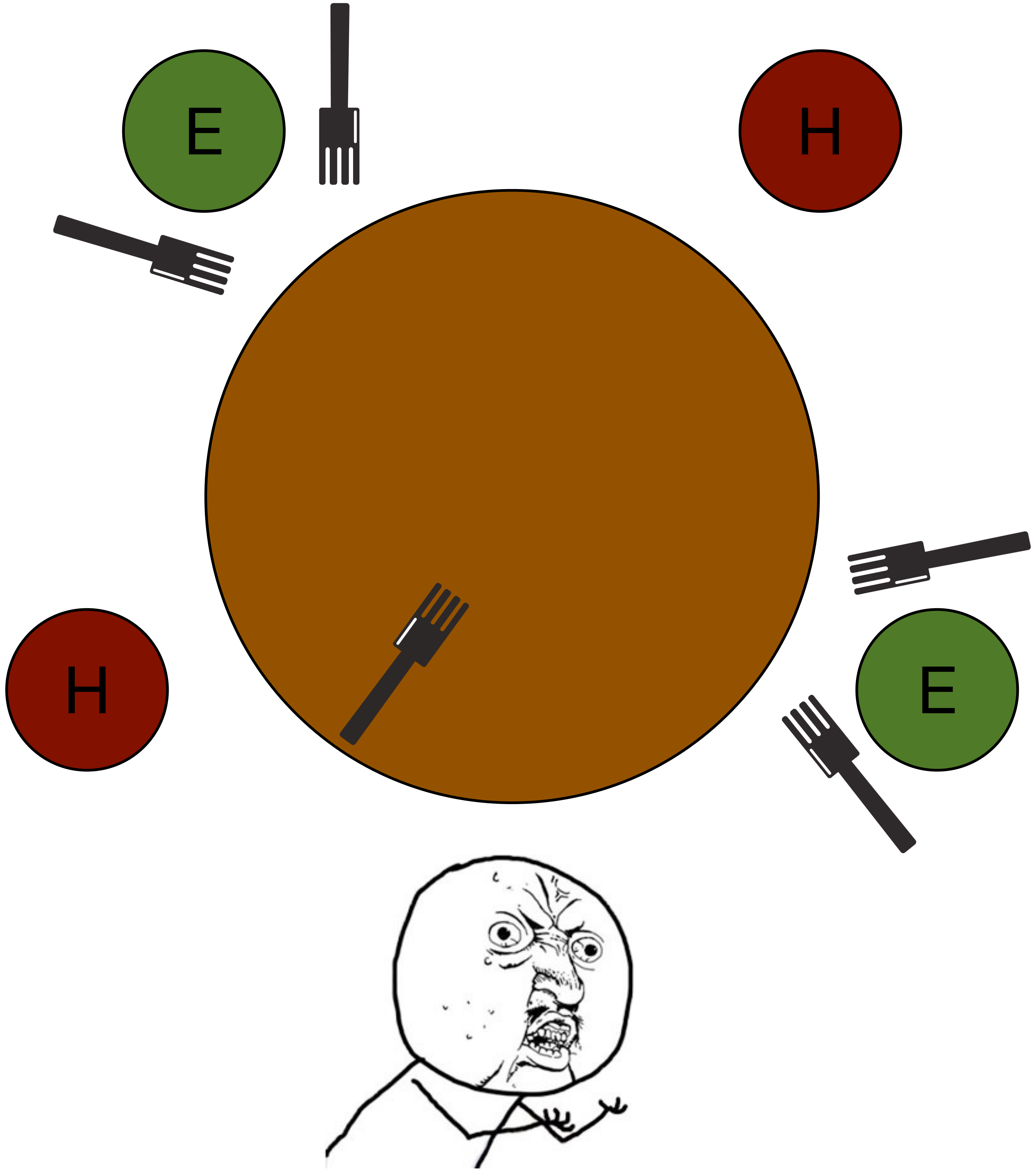
(let’s mess with this guy)

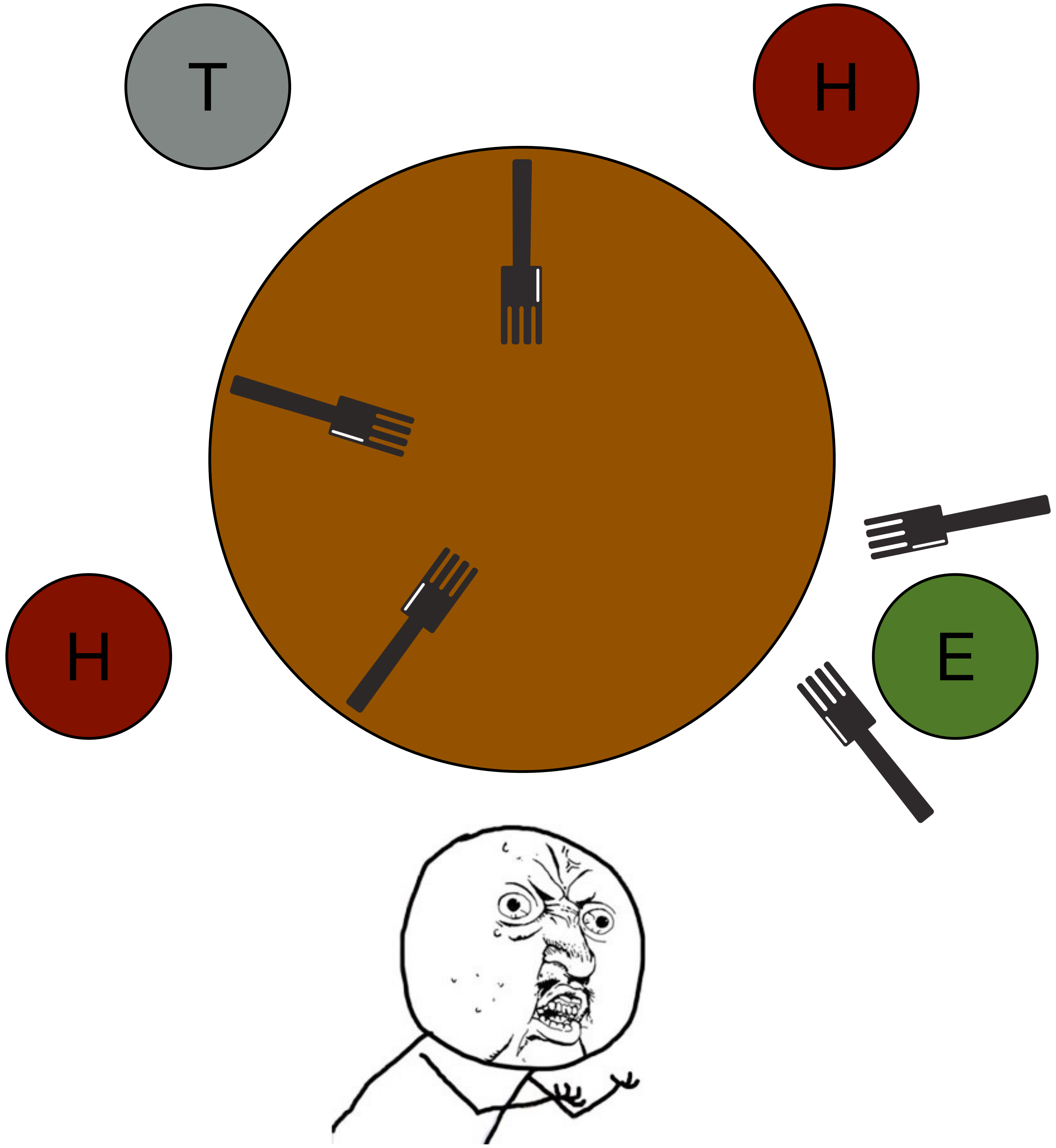

H
T
E
H
H

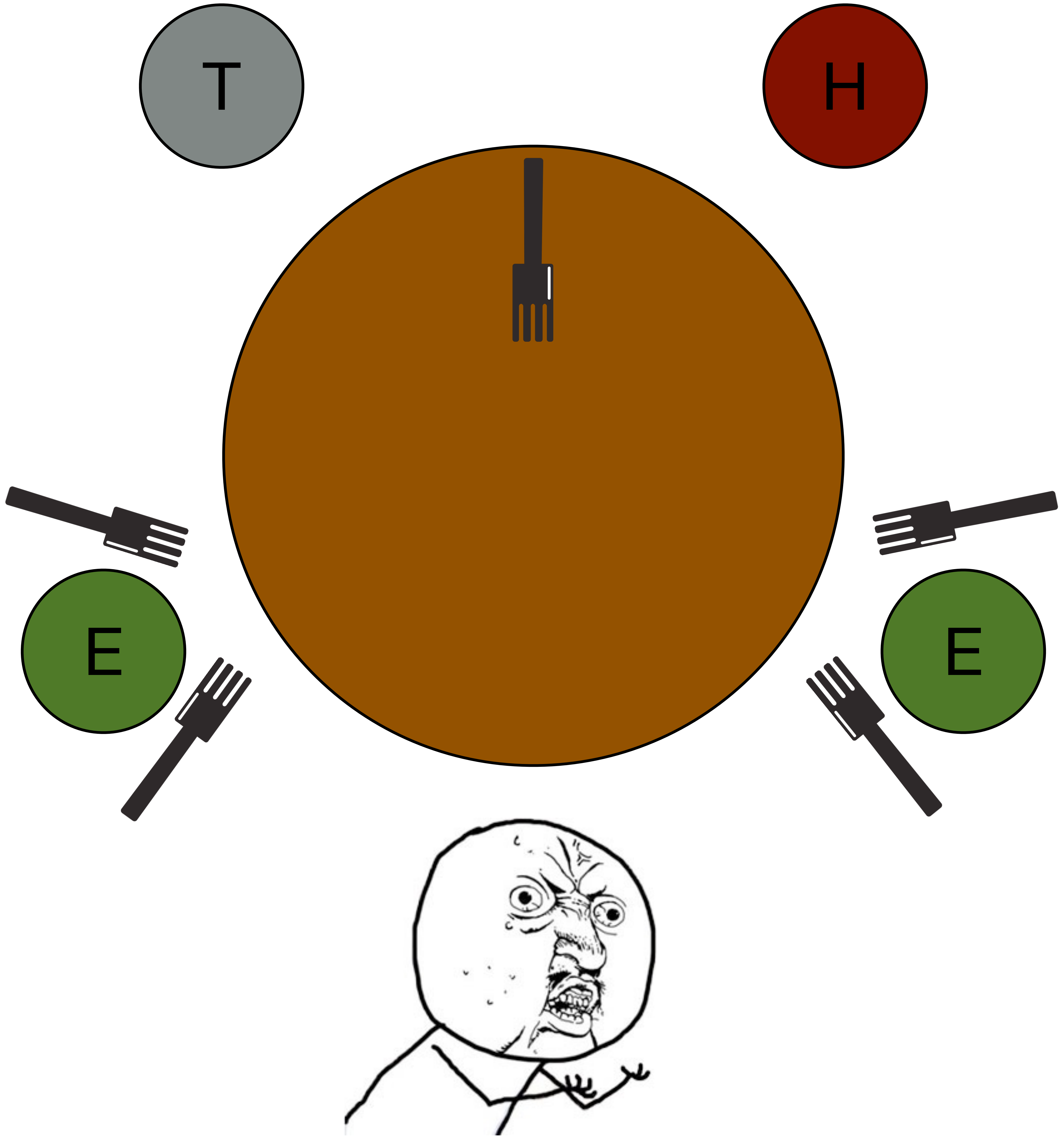

H
T
E
E
H

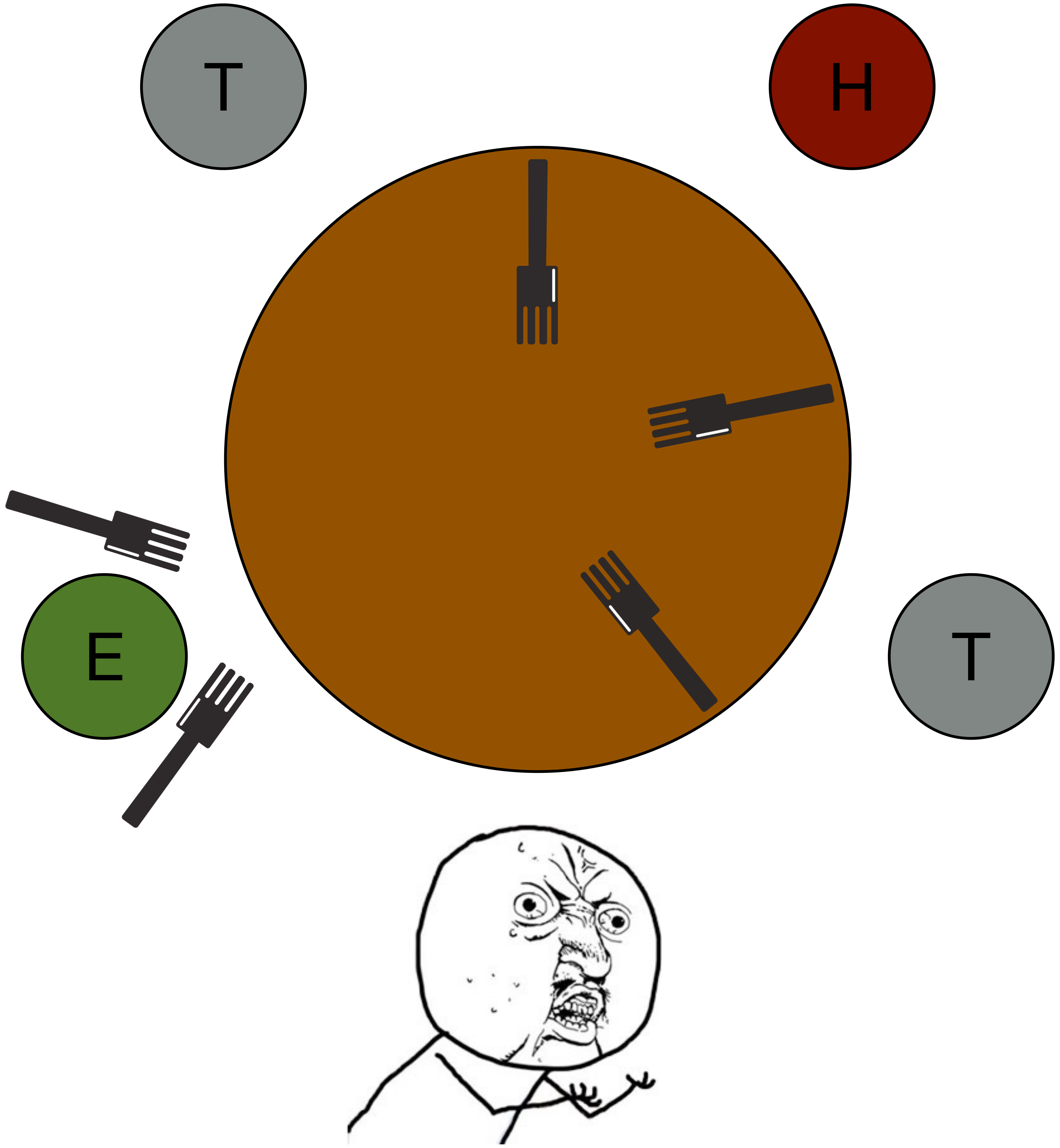

H
T
T
E
H

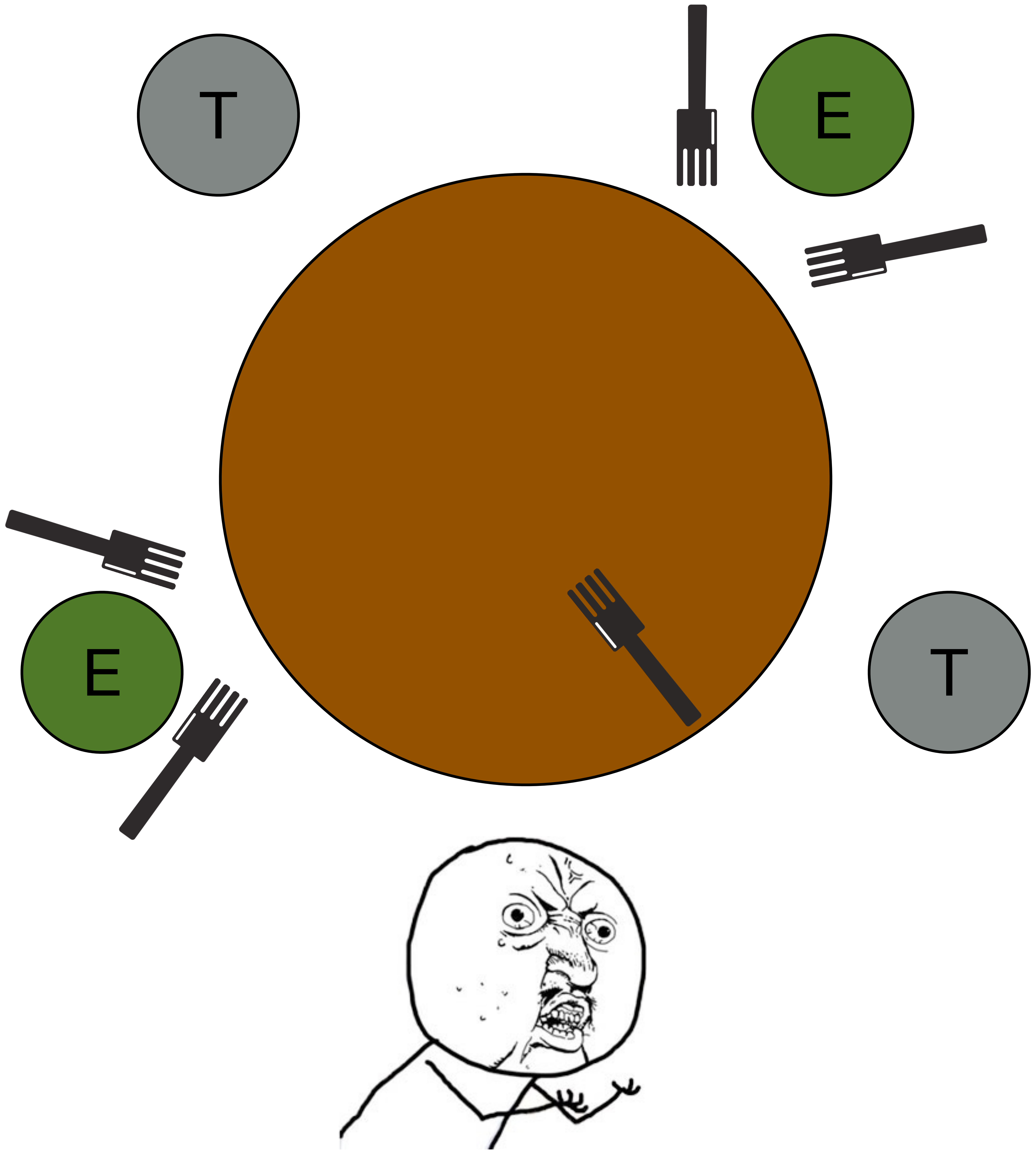

E
T
T
E

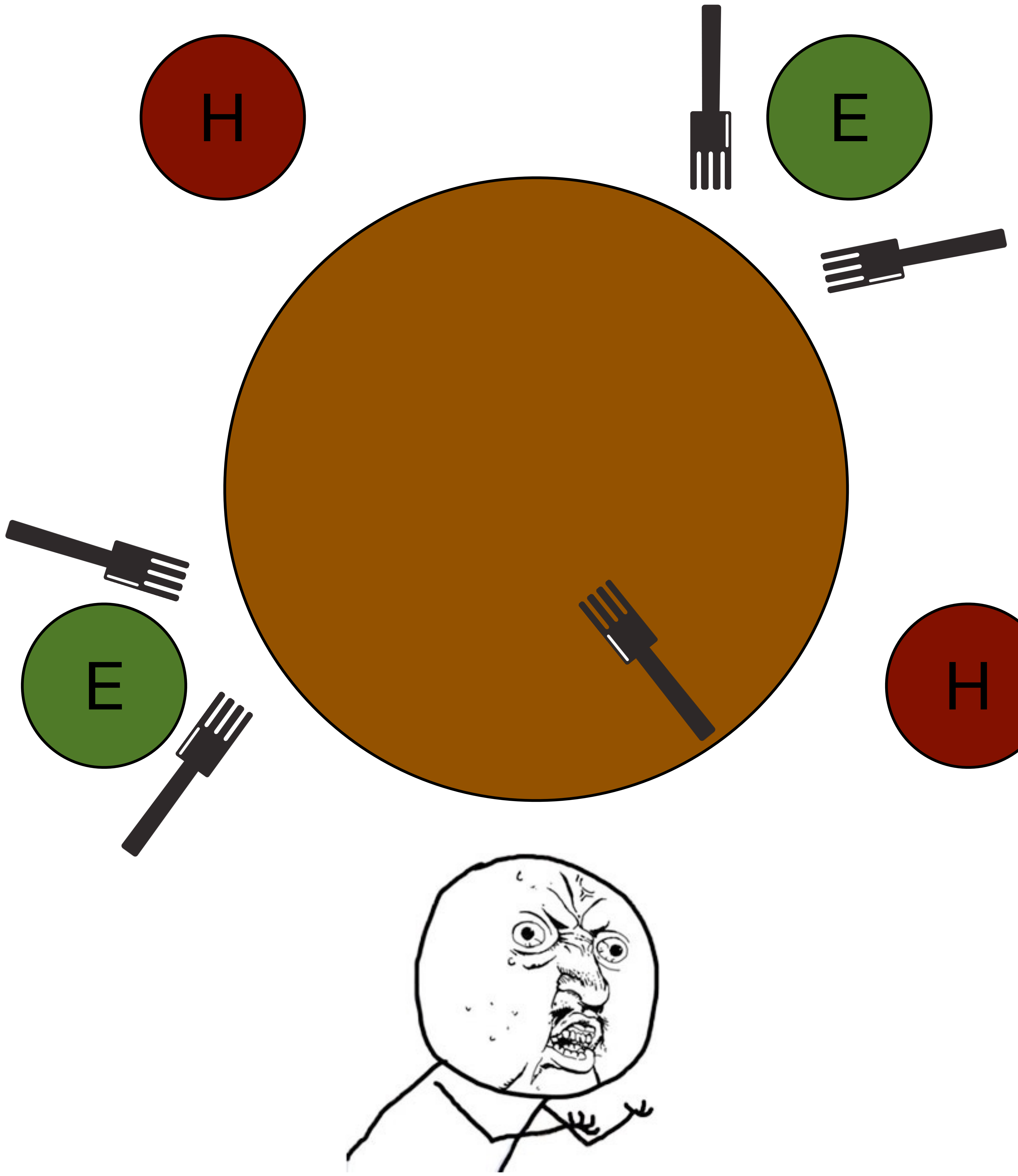

E
H
H
E

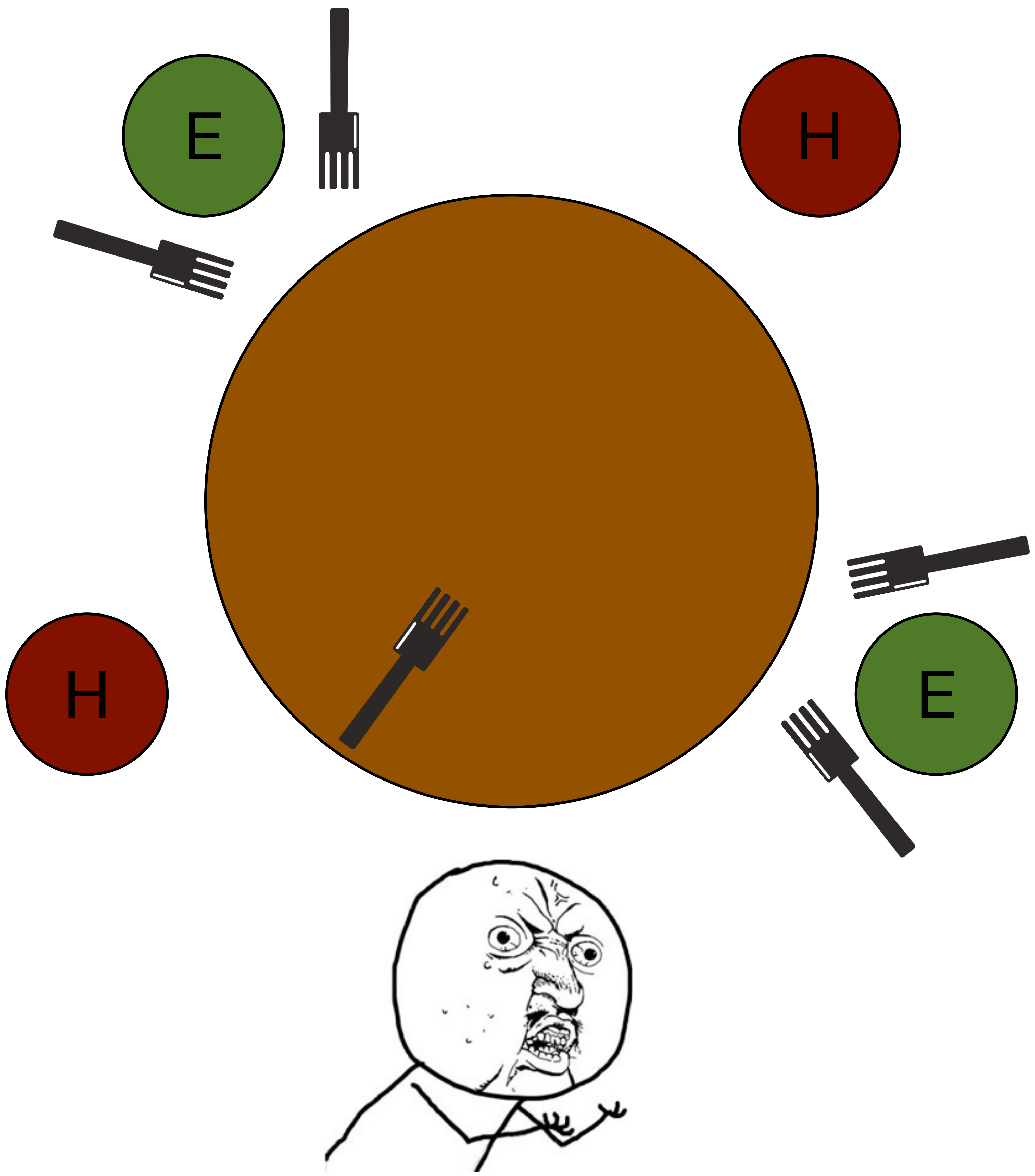

T
H
H
E

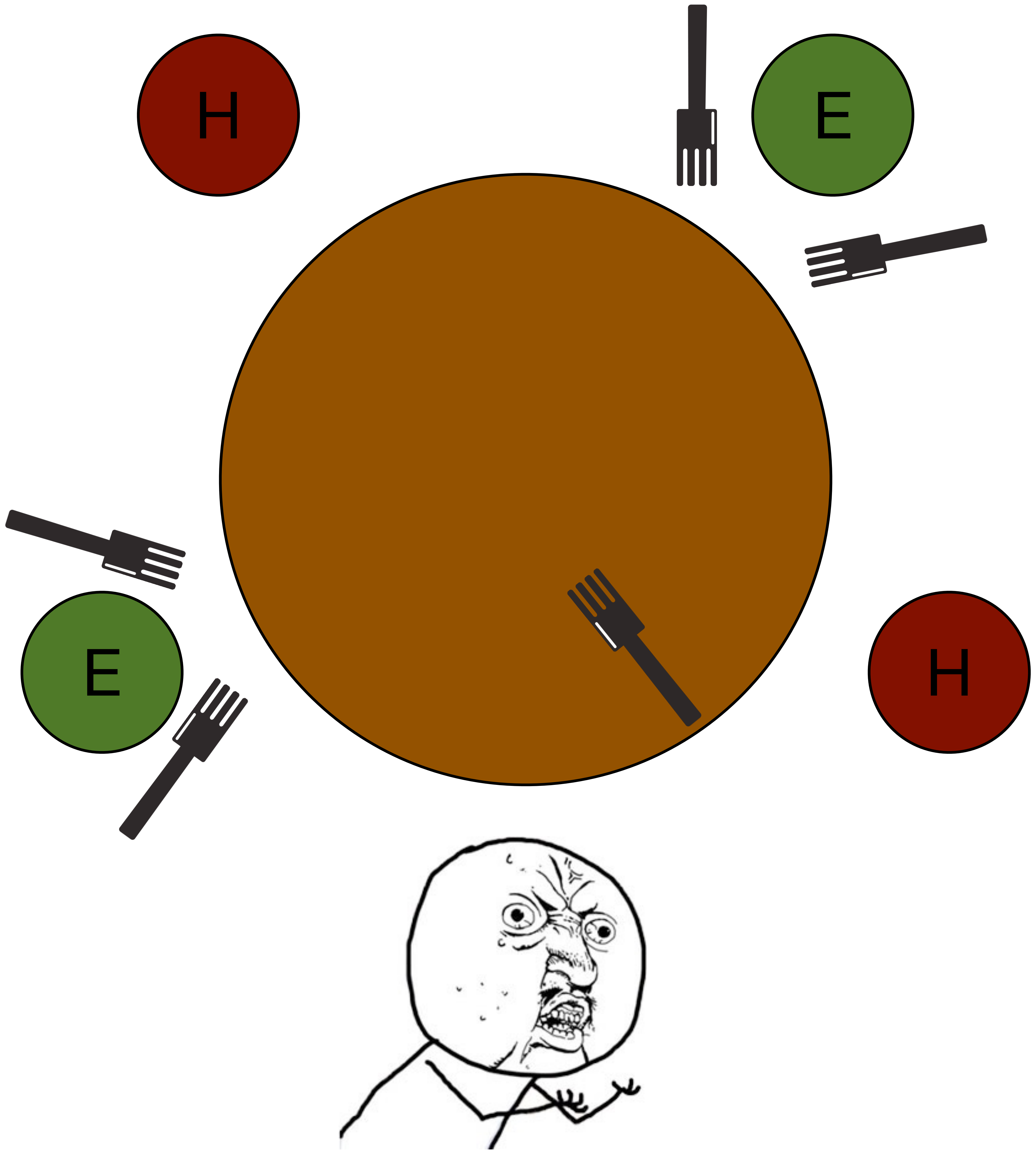

T
H
E
E

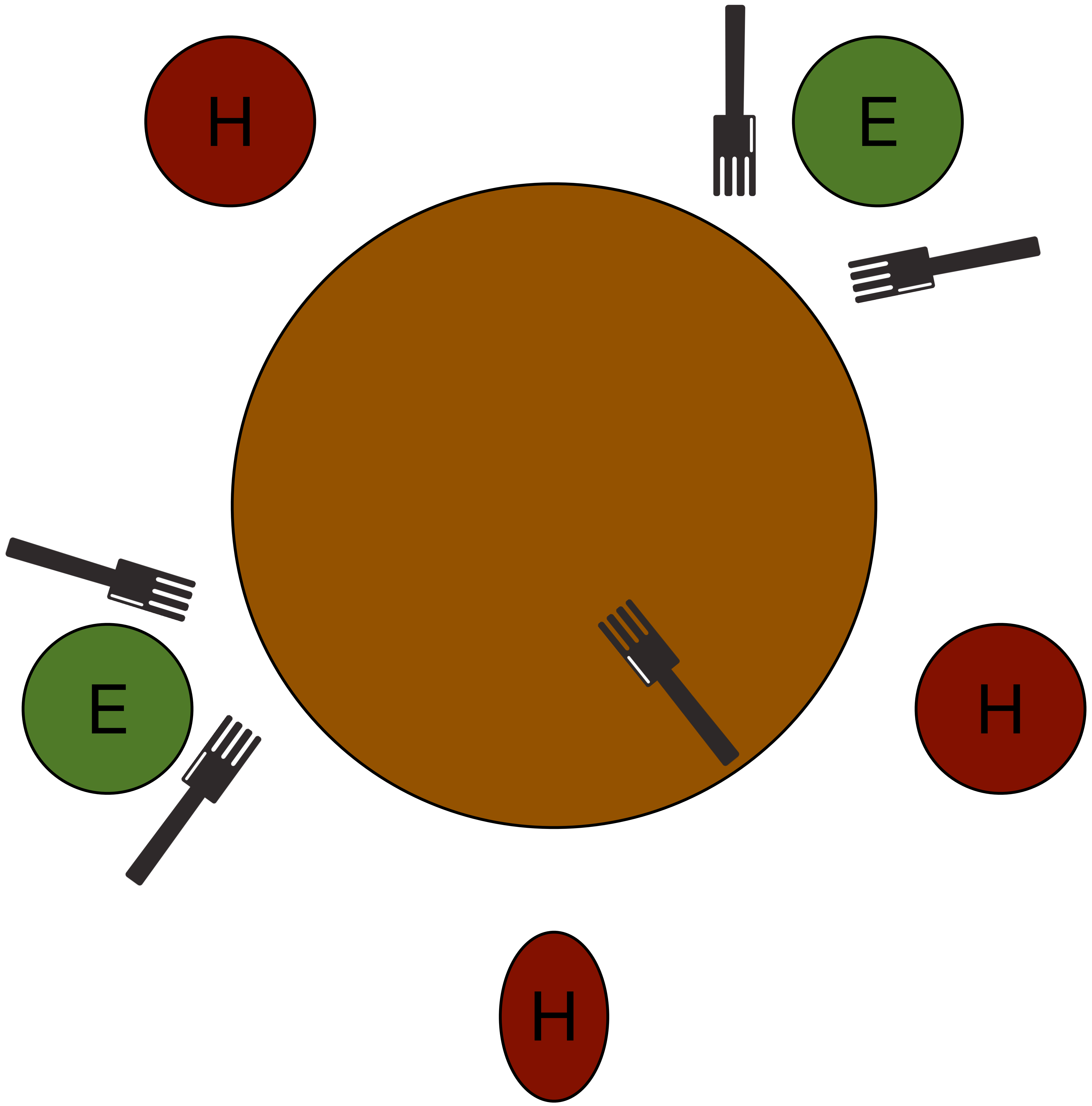

T
H
E
T


T
E
E
T


H
E
E
H


E
H
H
E


H
E
E
H



(starves)

§ Summary

Concurrency is desirable

- Can help improve CPU and I/O utilization
  - By blocking only part of a task/process instead of the whole thing
- May leverage parallelism for increase in performance
  - Limited by parallel portion of workload (Amdhal's/Gustafson's)
- May also help logically partition a task into discrete subtasks

Concurrency relies on the OS & HW

- The kernel is the original concurrent program
- Without kernel-level threads, we cannot translate user-level concurrency into performance gains
- Hardware support is needed to build robust and efficient mechanisms for concurrent programming
    - E.g., atomic instructions, interrupt mechanisms

Concurrency is hard!

- Concurrent tasks overlap non-deterministically, and when they access shared data, we may end up with *race conditions*
- Synchronizing concurrent tasks to eliminate race conditions while maximizing efficiency, eliminating starvation, etc., is hard!
  - Requires thinking in multiple dimensions and accounting for nearly infinite scenarios
  - When not done carefully, may entangle application and synchronization logic, and make code difficult to maintain