Concurrency, Races & Synchronization

Science

Computer Science

CS 450: Operating Systems

Michael Lee <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

Agenda

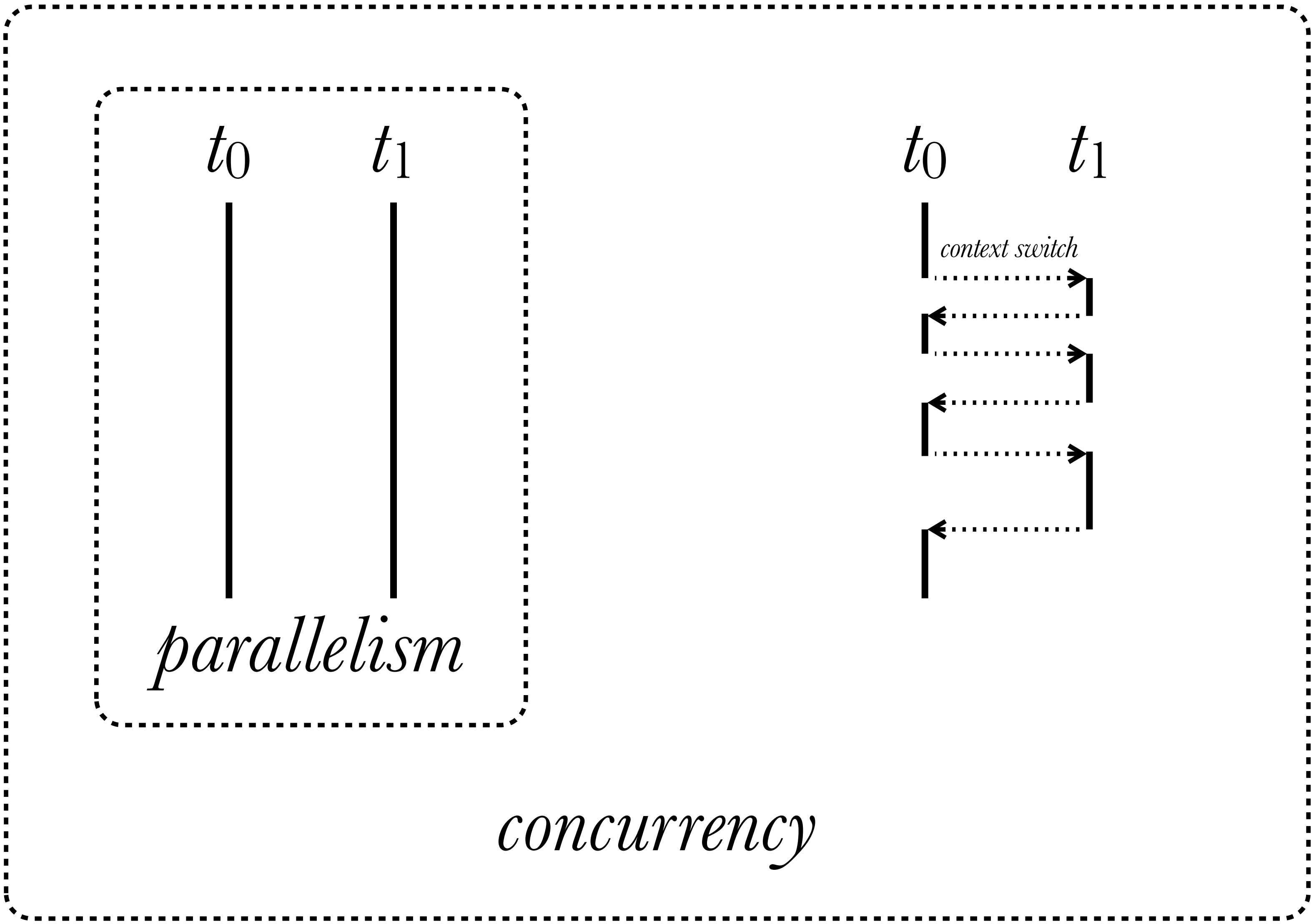
- Concurrency: what, why, how
  - Problems due to concurrency
- Locks & Locking strategies
- Concurrent programming with semaphores

§Concurrency: what, why, how

concurrency (in computing) = two or more *overlapping threads of execution*

thread [of execution] = a sequence of *instructions* and associated *state* *parallelism* (enabled by > 1 physical CPUs) is one way of realizing concurrency

… but concurrency can also be achieved via single-CPU multiplexing!


t0
t1
parallelism
t0
t1
context switch
concurrency

even on multi-CPU systems, CPU multiplexing is performed to achieve *higher levels of concurrency* (vs. hw parallelism) why concurrency?

1. multitasking
2. separate blocking activities
3. improve resource utilization
4. performance gains (most elusive!)

standard unit of concurrency: *process*

- single thread of execution "owns" virtualized CPU, memory
  - (mostly) *share-nothing* architecture

```c
int main() {
    pid_t pid;
    for (int i=0; i<5; i++) {
        if ((pid = fork()) == 0) {
            printf("Child %d says hello!\n", i);
            exit(0);
        } else {
            printf("Parent created child %d\n", pid);
        }
    }
    return 0;
}
```

```
Child 0 says hello!
Parent created child 7568
Parent created child 7569
Child 1 says hello!
Parent created child 7570
Parent created child 7571
Child 3 says hello!
Child 2 says hello!
Child 4 says hello!
Parent created child 7572
```

but single-thread model is inconvenient / non-ideal in some situations e.g., sequential operations that block on unrelated resources

```
read_from_disk1(buf1);    // block for input
read_from_disk2(buf2);    // block for input
read_from_network(buf3); // block for input
process_input(buf1, buf2, buf3);
```

would like to initiate input from separate blocking resources simultaneously

e.g., interleaved, but independent
CPU & I/O operations

```
while (1) {
    long_computation(); // CPU-intensive
    update_log_file();  // blocks on I/O
}
```

would like to start next computation while performing (blocking) log output

e.g., independent computations over large data set (software SIMD)

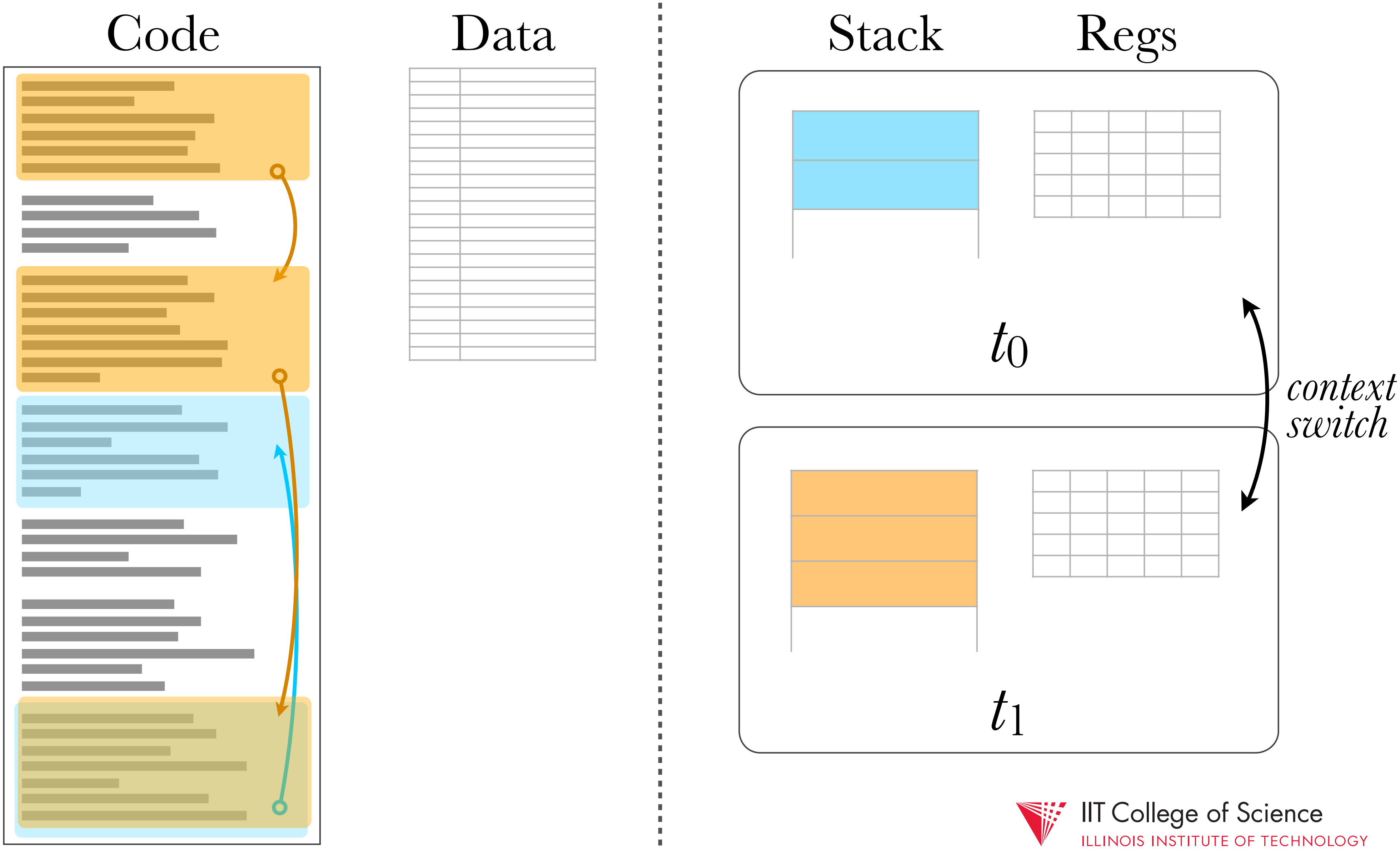
```
int A[DIM][DIM], /* src matrix A */
    B[DIM][DIM], /* src matrix B */
    C[DIM][DIM]; /* dest matrix C */

/* C = A x B */
void matrix_mult () {
    int i, j, k;
    for (i=0; i<DIM; i++) {
        for (j=0; j<DIM; j++) {
            C[i][j] = 0;
            for (k=0; k<DIM; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

each cell in result is independent — need not serialize!

within xv6 kernel there is no inherent process primitive — instead, implement concurrency via *multiple kernel stacks* (and program counters + other context)

i.e., multiple *threads of execution*,
one program

Global (shared) | Thread-local



xv6 does not support multi-threads in user processes, but most modern OSes do

- even if not supported by kernel, can emulate multi-threading at user level
  - same design: separate stacks & regs
    - user implemented context switch

multithreading libraries & APIs allow us to use threads without worrying about implementation details

POSIX Threads ("pthreads") is one API for working with threads

- both kernel-level (aka *native*) and user-level (aka *green*) implementations exist native threads provide kernel-level support for parallelism, but also increase context switch overhead (full-fledged interrupt)

```
/* thread creation */
int pthread_create ( pthread_t *tid,
                     const pthread_attr_t *attr,
                     void *(*thread_fn)(void *),
                     void *arg );

/* wait for termination; thread "reaping" */
int pthread_join ( pthread_t tid,
                   void **result_ptr );
```

```c
void *sayHello (void *num) {
    printf("Hello from thread %ld\n", (long)num);
    pthread_exit(NULL);
}

int main () {
    pthread_t tid;
    for (int i=0; i<5; i++){
        pthread_create(&tid, NULL, sayHello, (void *)i);
        printf("Created thread %ld\n", (long)tid);
    }
    pthread_exit(NULL);
    return 0;
}
```

```
Created thread 4558688256
Created thread 4559224832
Created thread 4559761408
Hello from thread 0
Created thread 4560297984
Hello from thread 1
Hello from thread 3
Created thread 4560834560
Hello from thread 4
Hello from thread 2
```

```c
int A[DIM][DIM], /* src matrix A */
    B[DIM][DIM], /* src matrix B */
    C[DIM][DIM]; /* dest matrix C */

/* C = A x B */
void matrix_mult () {
    int i, j, k;
    for (i=0; i<DIM; i++) {
        for (j=0; j<DIM; j++) {
            C[i][j] = 0;
            for (k=0; k<DIM; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Run time, with DIM=50, 500 iterations:

```
real    0m1.279s
user    0m1.260s
sys     0m0.012s
```

```c
void run_with_thread_per_cell() {
    pthread_t ptd[DIM][DIM];
    int index[DIM][DIM][2];

    for(int i = 0; i < DIM; i ++)
        for(int j = 0; j < DIM; j ++) {
            index[i][j][0] = i;
            index[i][j][1] = j;
            pthread_create(&ptd[i][j], NULL,
                           row_dot_col,
                           index[i][j]);
        }

    for(i = 0; i < DIM; i ++)
        for(j = 0; j < DIM; j ++)
            pthread_join( ptd[i][j], NULL);
}
```

```c
void row_dot_col(void *index) {
    int *pindex = (int *)index;
    int i = pindex[0];
    int j = pindex[1];

    C[i][j] = 0;
    for (int x=0; x<DIM; x++)
        C[i][j] += A[i][x]*B[x][j];
}
```

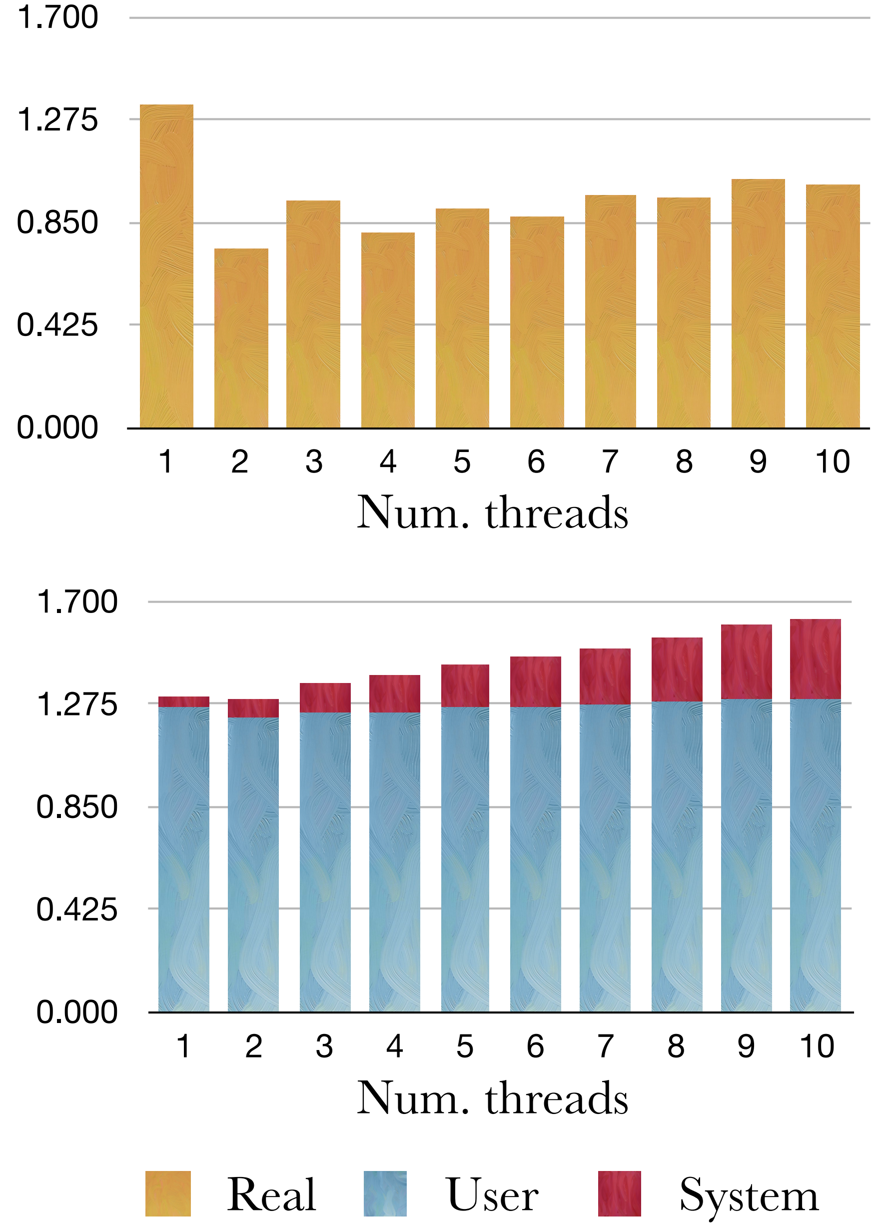
Run time, with DIM=50,
500 iterations:

```
real     4m18.013s
user     0m33.655s
sys      4m31.936s
```

```c
void run_with_n_threads(int num_threads) {
    pthread_t tid[num_threads];
    int tdata[num_threads][2];
    int n_per_thread = DIM/num_threads;

    for (int i=0; i<num_threads; i++) {
        tdata[i][0] = i*n_per_thread;
        tdata[i][1] = (i < num_threads)
                      ? ((i+1)*n_per_thread)-1
                      : DIM;
        pthread_create(&tid[i], NULL,
                       compute_rows,
                       tdata[i]);
    }
    for (int i=0; i<num_threads; i++)
        pthread_join(tid[i], NULL);
}
```

```c
void *compute_rows(void *arg) {
    int *bounds = (int *)arg;
    for (int i=bounds[0];
         i<=bounds[1];
         i++) {
        for (int j=0; j<DIM; j++) {
            C[i][j] = 0;
            for (int k=0; k<DIM; k++)
                C[i][j] += A[i][k]
                           * B[k][j];
        }
    }
}
```

Dual processor system, kernel threading, DIM=50, 500 iterations



but matrix multiplication happens to be an *embarrassingly parallelizable* computation!

- not typical of concurrent tasks!

computations on shared data are typically *interdependent* (and this isn't always obvious!)

— may impose a *cap* on parallelizeability

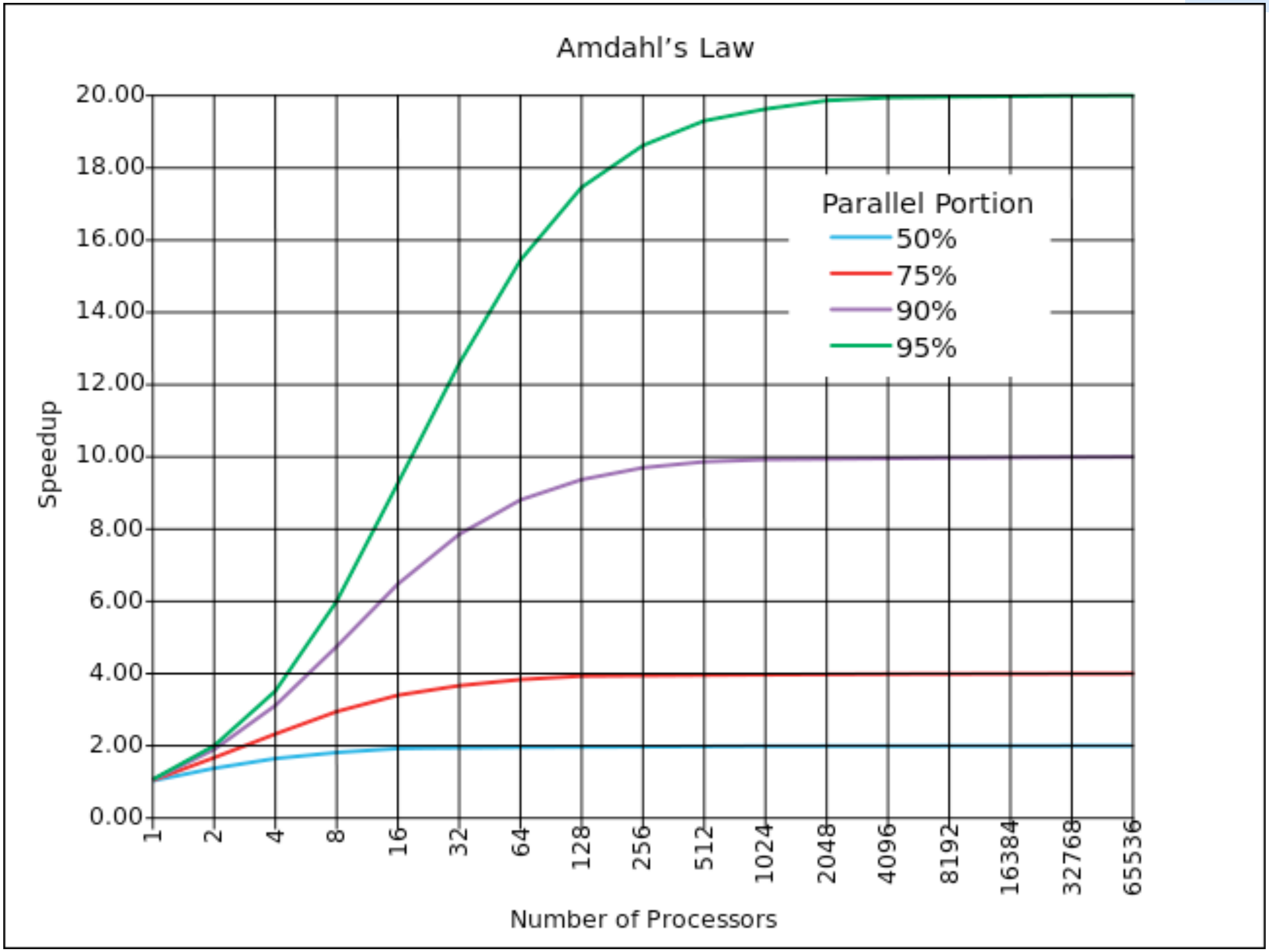
Amdhal's law predicts max speedup given two parameters:

- P : fraction of program that's parallelized
- N : # of execution cores

$$\text{max speedup } S = \frac{1}{\frac{P}{N} + (1 - P)}$$

† $P \rightarrow 1;\ S \rightarrow N$

‡ $N \rightarrow \infty;\ S \rightarrow 1/(1 - P)$



source: http://en.wikipedia.org/wiki/File:AmdahlsLaw.svg note: Amdahl's law is based on a *fixed problem size* (with *fixed parallelized portion*)

— but we can argue that as we have more computing power we simply tend to throw *larger / more granular problem sets* at it e.g.,

graphics processing: keep turning up resolution/detail

weather modeling: increase model parameters/accuracy

chess/weiqi AI: deeper search tree

Gustafson & Barsis posit that

- we tend to scale problem size to complete in the *same amount of time*, regardless of the number of cores

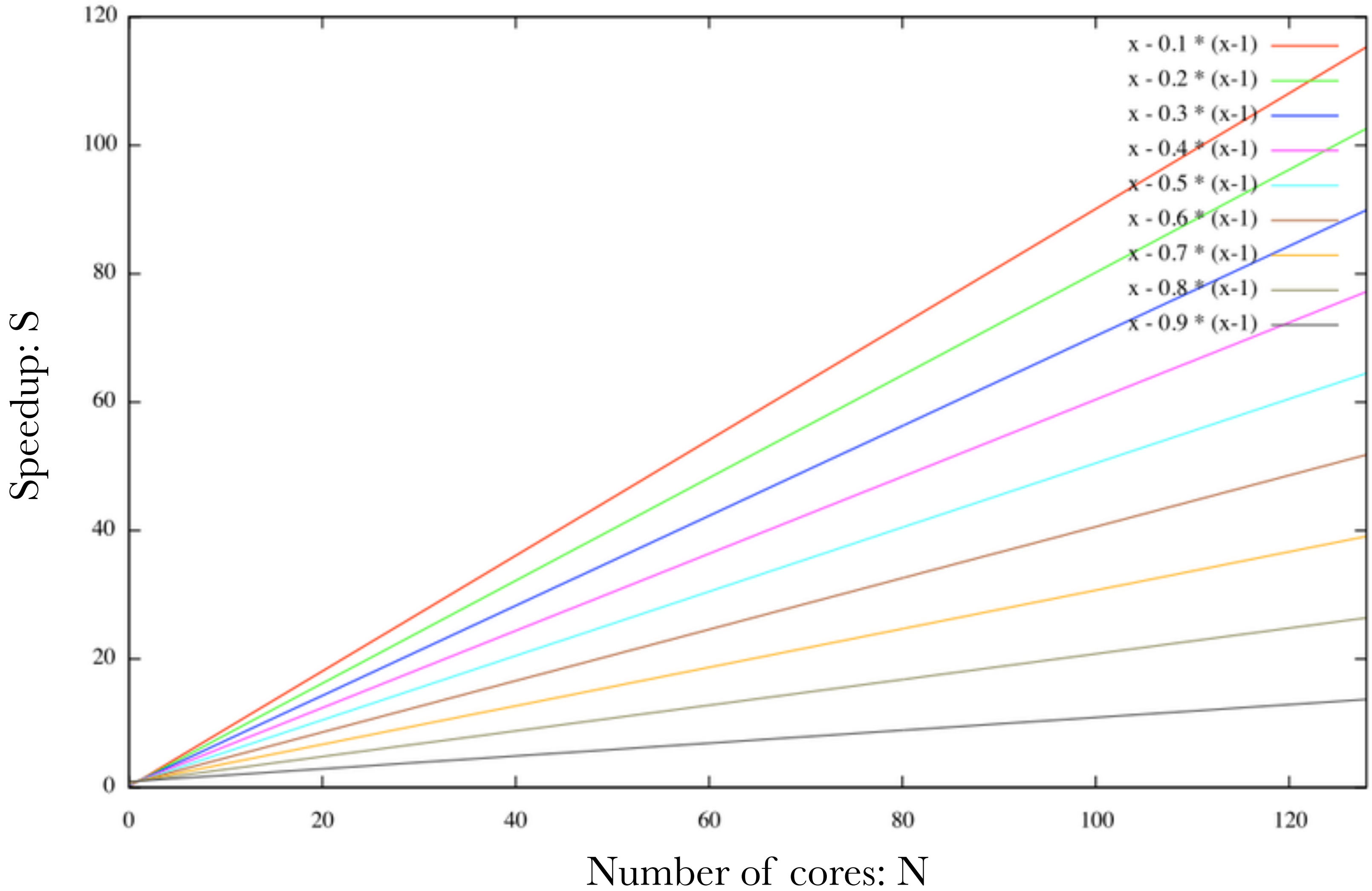
- parallelizeable amount of work scales linearly with number of cores

Gustafson's Law computes speedup based on:

- N cores
- *non*-parallelized fraction, P

$$\text{speedup } S = N - P \cdot (N - 1)$$

- note that speedup is linear with respect to number of cores!


x - 0.1 * (x-1)
x - 0.2 * (x-1)
x - 0.3 * (x-1)
x - 0.4 * (x-1)
x - 0.5 * (x-1)
x - 0.6 * (x-1)
x - 0.7 * (x-1)
x - 0.8 * (x-1)
x - 0.9 * (x-1)
Speedup: S
Number of cores: N
0
20
40
60
80
100
120

Amdahl's vs. Gustafson's:

- latter has rosier implications for big data analysis / data science
  - but not all datasets naturally expand / increase in resolution
- both stress the import of maximizing the parallelizeable fraction

some of the primary challenges of concurrent programming are to:

1. identify thread interdependencies
2. identify (1)'s potential ramifications
3. ensure correctness

e.g., final change in count? (expected = 2)

Thread A

```
a1  count = count + 1
```

Thread B

```
b1  count = count + 1
```

interdependency: shared var `count`

factoring in machine-level granularity:

Thread A

```
a1  lw   (count), %r0
a2  add $1, %r0
a3  sw  %r0, (count)
```

Thread B

```
b1  lw   (count), %r0
b2  add $1, %r0
b3  sw  %r0, (count)
```

answer: either +1 or +2!

race condition(s) exists when results are dependent on the *order of execution* of concurrent tasks

shared resource(**s**) are the problem

or, more specifically, *concurrent mutability* of those shared resources code that accesses shared resource(s) = *critical section*

synchronization:

time-sensitive coordination of critical sections so as to *avoid race conditions* e.g., specific *ordering* of different threads, or *mutually exclusive* access to variables important: try to separate application logic from synchronization details

- another instance of policy vs. mechanism
- this can be hard to get right!

most common technique for implementing synchronization is via software "locks"

- explicitly required & released by consumers of shared resources

§ Locks & Locking Strategies

basic idea:

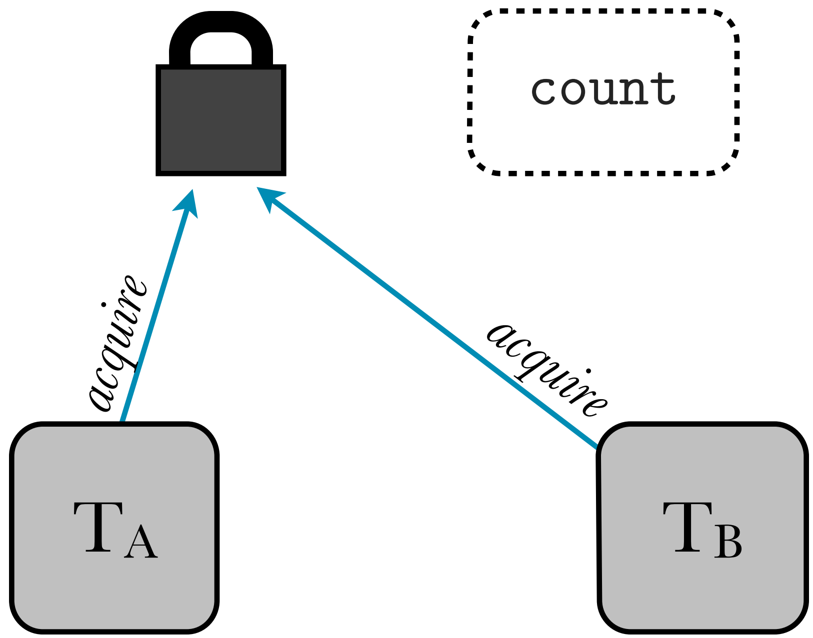
- create a shared software construct that has well defined concurrency semantics
  - aka. a "thread-safe" object
- Use this object as a guard for another, un-thread-safe shared resource

Thread A

```
a1  count = count + 1
```

Thread B

```
b1  count = count + 1
```

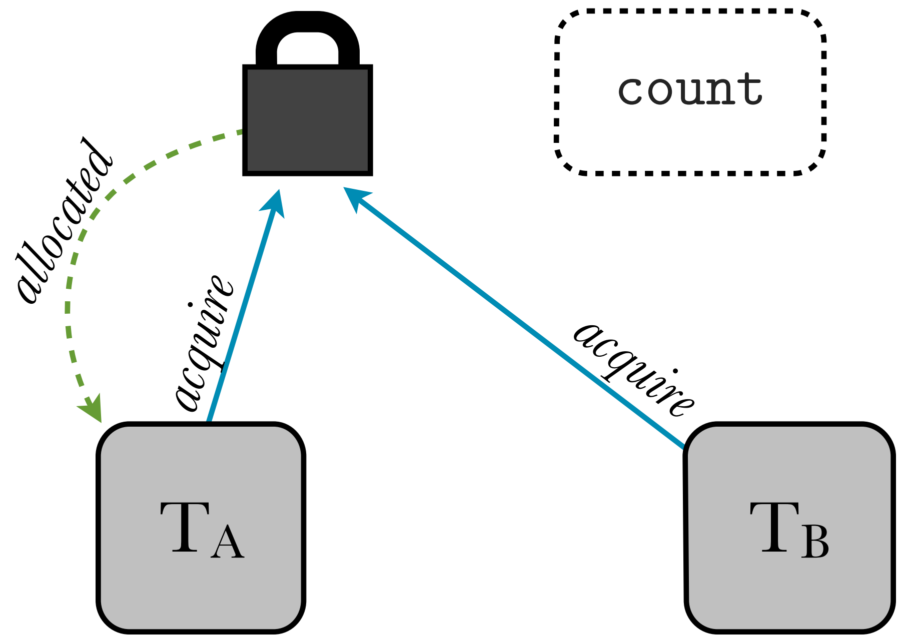


Thread A

```
a1  count = count + 1
```

Thread B

```
b1  count = count + 1
```

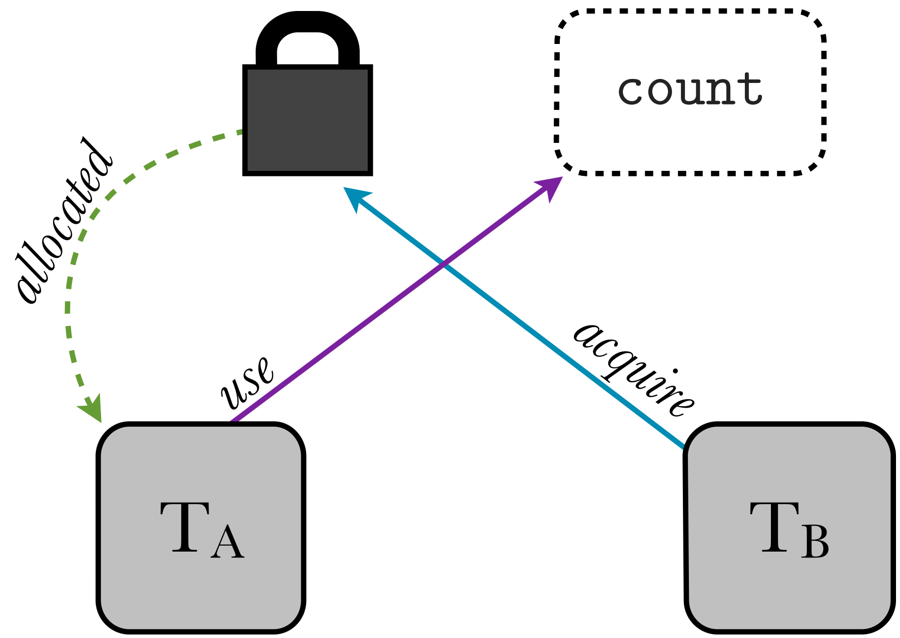


Thread A

```
a1  count = count + 1
```

Thread B

```
b1  count = count + 1
```

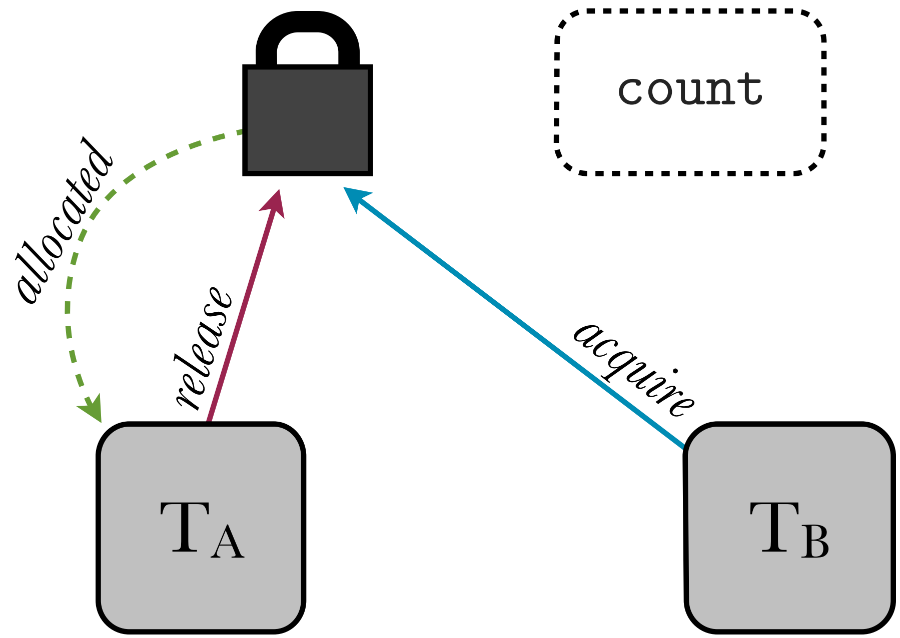


Thread A

```
a1  count = count + 1
```

Thread B

```
b1  count = count + 1
```

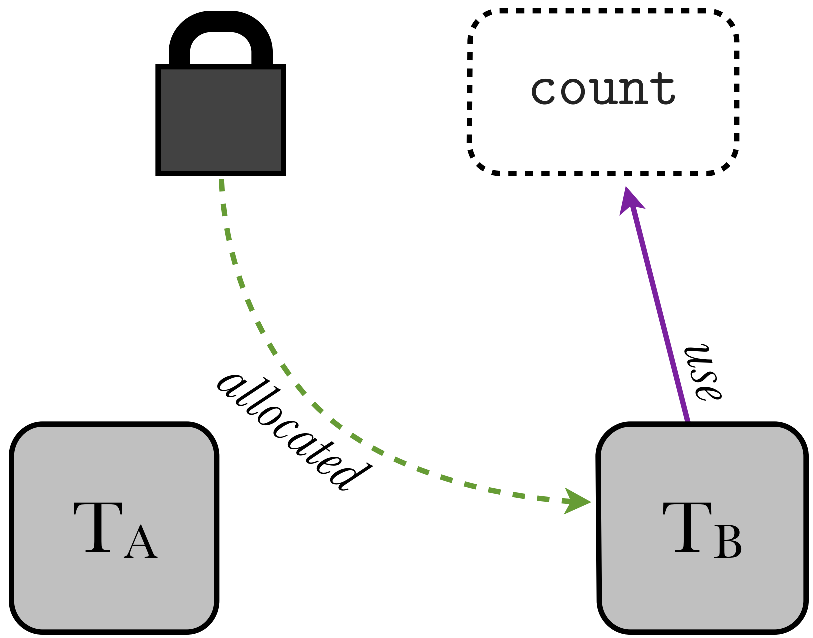


Thread A

```
a1  count = count + 1
```

Thread B

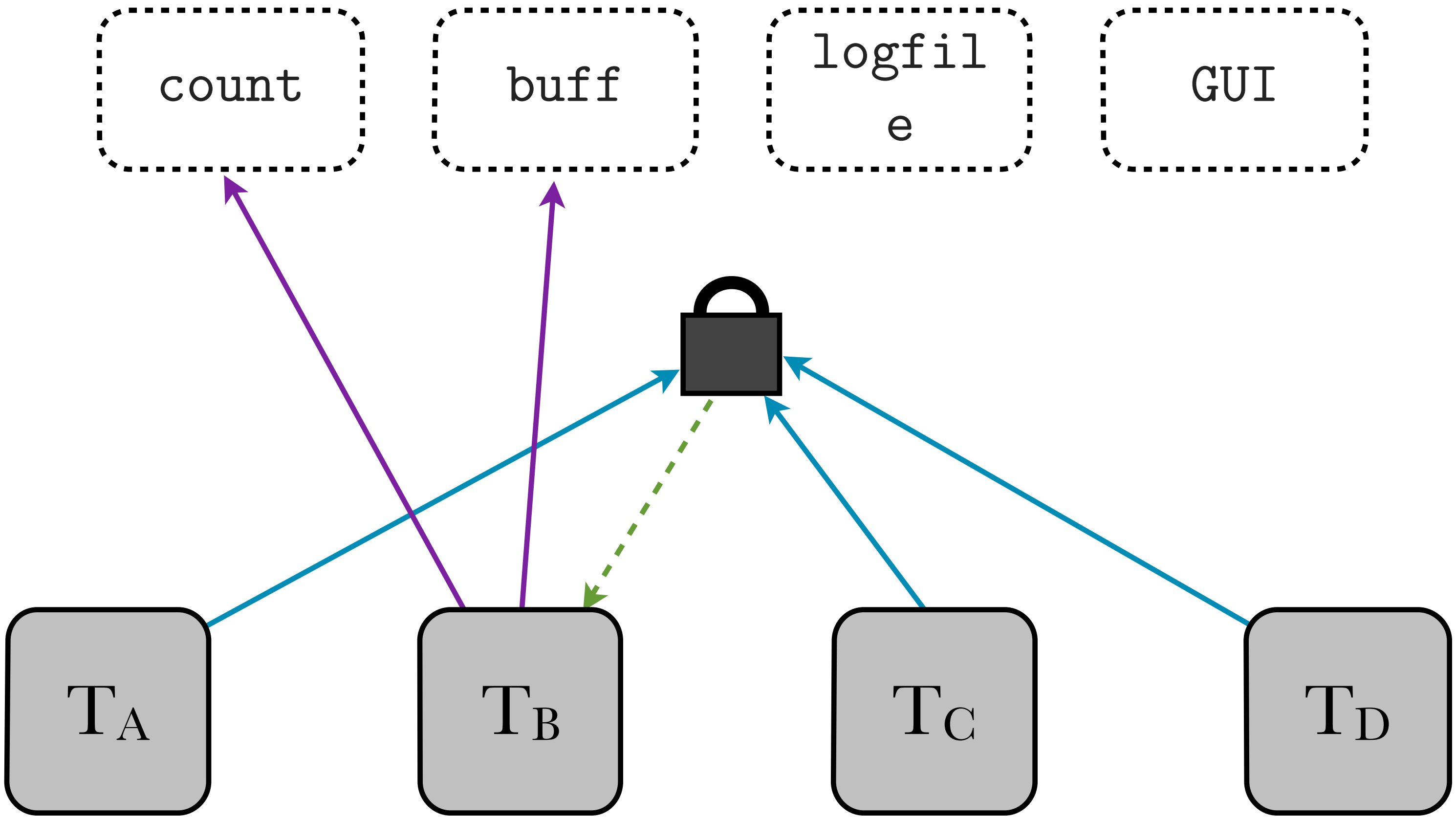
```
b1  count = count + 1
```



locking can be:

- **global** (*coarse-grained*)
- **per-resource** (*fine-grained*)

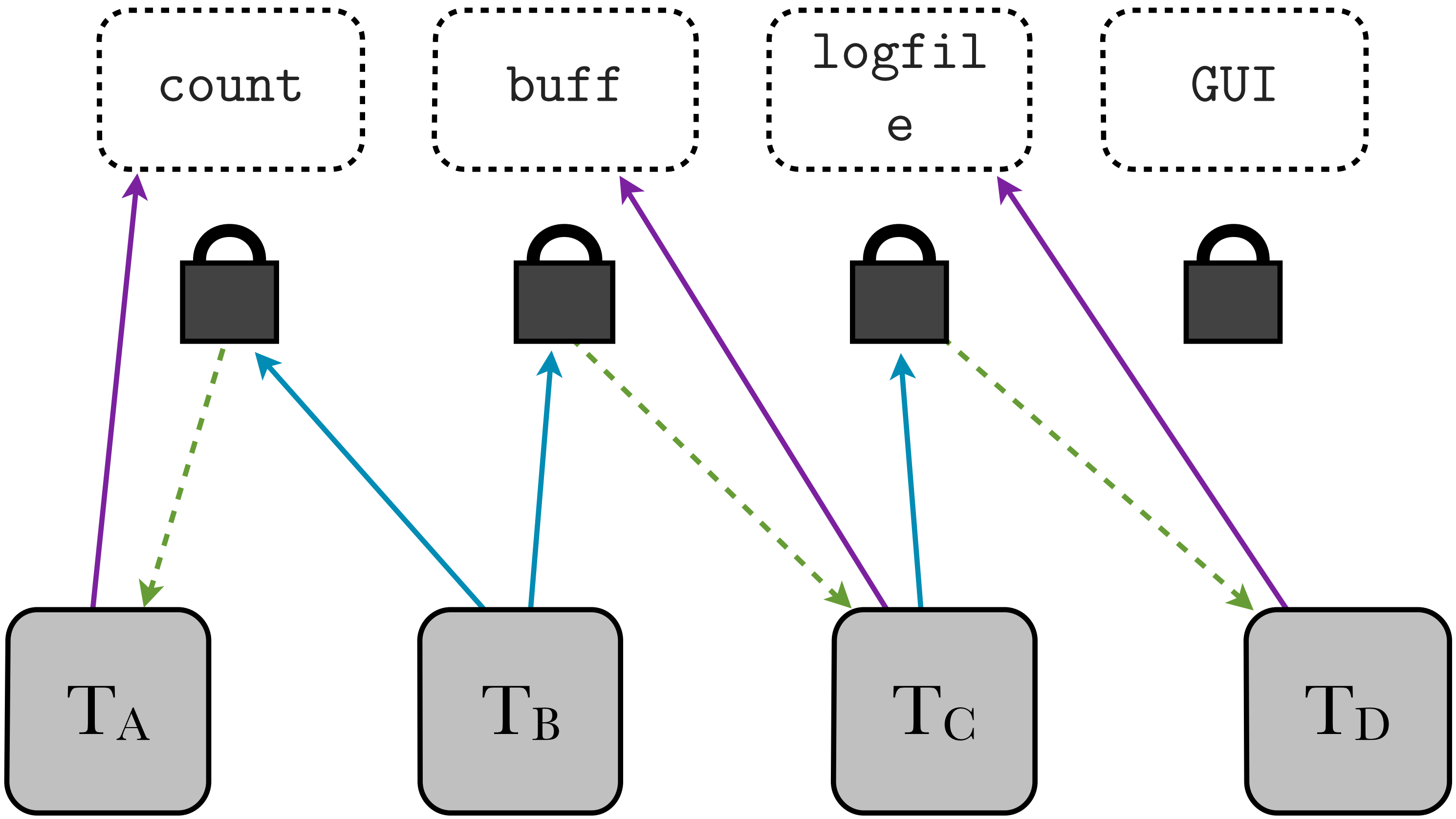
coarse-grained locking policy



coarse-grained locking:

- is (typically) easier to reason about
- results in a lot of *lock contention*
- could result in *poor resource utilization* — may be impractical for this reason

fine-grained locking policy



fine-grained locking:

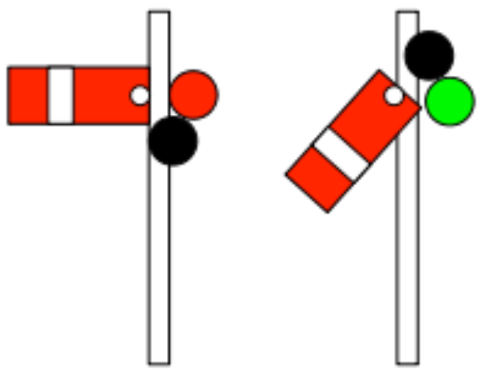
- may reduce (individual) lock contention
- may improve resource utilization
- can result in a lot of locking overhead
- can be much harder to verify correctness!

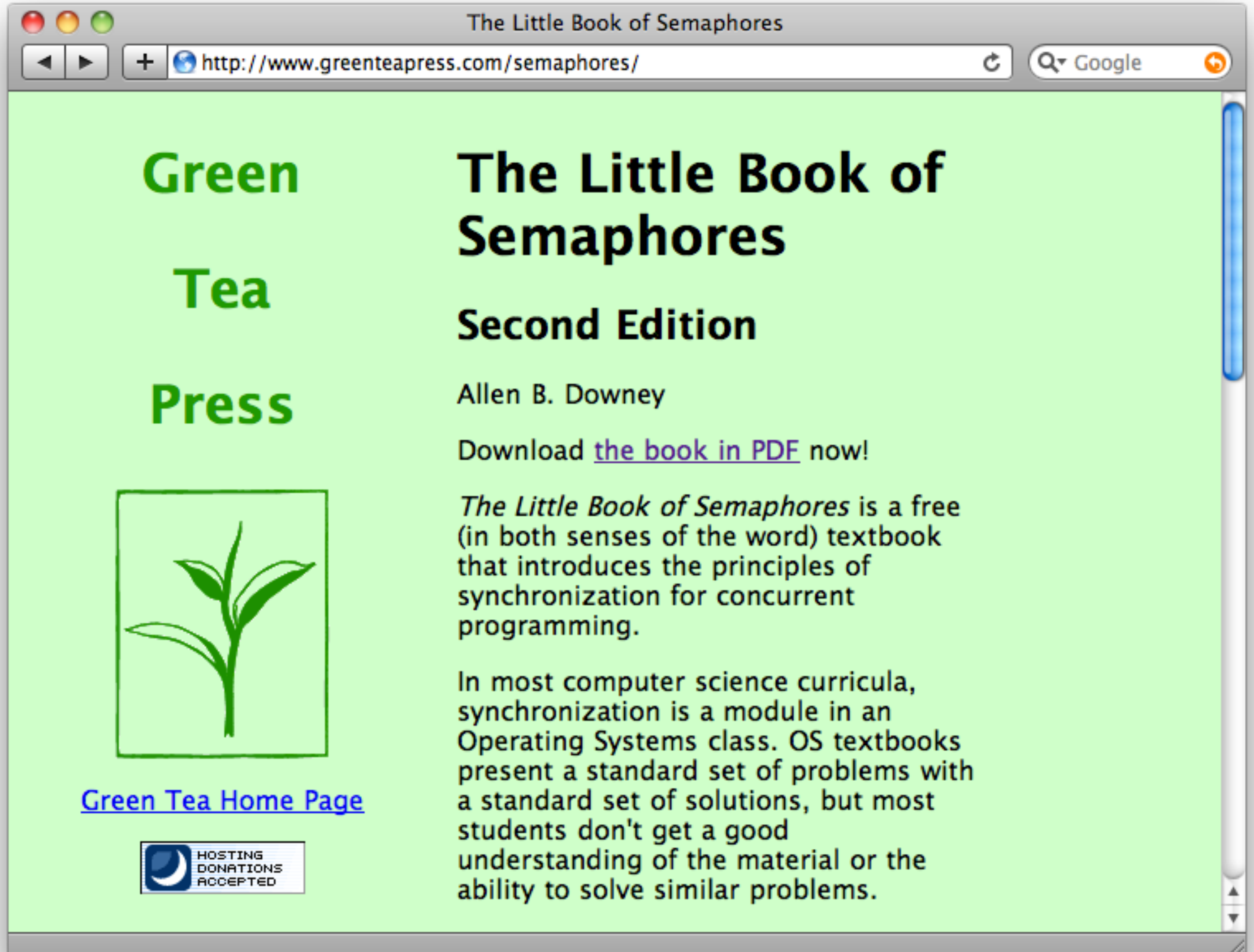
so far, have only considered *mutual exclusion*

what about instances where we require a *specific order* of execution?

- often very difficult to achieve with simple-minded locks

§Abstraction: **Semaphore**





Little Book of Semaphores

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 syntax:

```
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()
```

How to use semaphores for synchronization?

1. Identify essential usage "patterns"
2. Solve "classic" synchronization problems

Essential synchronization criteria:

1. avoid *starvation*

2. guarantee *bounded waiting*

3. no assumptions on *relative speed* (of threads)

4. allow for *maximum concurrency*

§Using Semaphores for Synchronization

Basic patterns:

I. Rendezvous
II. Mutual exclusion (Mutex)
III. Multiplex
IV. Generalized rendezvous / Barrier & Turnstile

I. Rendezvous

Thread A

```
statement a1
statement a2
```

Thread B

```
statement b1
statement b2
```

Guarantee: $a1 < b2, b1 < a2$

```
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
```

Note: Swapping 2 & 3 → Deadlock!

Thread A

```
1  statement a1
2  bArrived.wait()
3  aArrived.signal()
4  statement a2
```

Thread B

```
1  statement b1
2  aArrived.wait()
3  bArrived.signal()
4  statement b2
```

II. Mutual exclusion

Thread A

```
count = count + 1
```

Thread B

```
count = count + 1
```

```
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()
```

III. multiplex = Semaphore(N)

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

IV. Generalized Rendezvous / Barrier

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

Listing 3.2: Barrier code

```
rendezvous
critical point
```

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

```
rendezvous

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

if count == n: barrier.signal()

barrier.wait()
barrier.signal()

critical point
```

```
1   rendezvous
2
3   mutex.wait()
4       count = count + 1
5   mutex.signal()
6
7   if count == n: turnstile.signal()
8
9   turnstile.wait()
10  turnstile.signal()
11
12  critical point
```

state of turnstile after all threads make it to 12?

```
rendezvous

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

turnstile.wait()
turnstile.signal()

critical point
```

fix for non-determinism (but still off by one) next: would like a **reusable** barrier

need to **re-lock** turnstile

```
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()
```

(doesn't work!)

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

```
# rendezvous

mutex.wait()
    count += 1
    if count == n:
        turnstile2.wait()     # lock the second
        turnstile.signal()    # unlock the first
mutex.signal()

turnstile.wait()              # first turnstile
turnstile.signal()

# critical point

mutex.wait()
    count -= 1
    if count == 0:
        turnstile.wait()       # lock the first
        turnstile2.signal()    # unlock the second
mutex.signal()

turnstile2.wait()              # second turnstile
turnstile2.signal()
```

```
# rendezvous

mutex.wait()
    count += 1
    if count == n:
        turnstile.signal(n)    # unlock the first
mutex.signal()

turnstile.wait()               # first turnstile

# critical point

mutex.wait()
    count -= 1
    if count == 0:
        turnstile2.signal(n)   # unlock the second
mutex.signal()

turnstile2.wait()              # second turnstile
```

next: classic synchronization problems

I. Producer / Consumer

Assume that producers perform the following operations over and over:

Listing 4.1: Basic producer code

```
event = waitForEvent()
buffer.add(event)
```

Also, assume that consumers perform the following operations:

Listing 4.2: Basic consumer code

```
event = buffer.get()
event.process()
```

important: `buffer` is *finite* and *non-thread-safe*!

- finite, non-thread-safe buffer
- 1 semaphore per item/space

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

Listing 4.11: Finite buffer consumer solution

```
items.wait()
mutex.wait()
    event = buffer.get()
mutex.signal()
spaces.signal()

event.process()
```

Listing 4.12: Finite buffer producer solution

```
event = waitForEvent()

spaces.wait()
mutex.wait()
    buffer.add(event)
mutex.signal()
items.signal()
```

II. Readers/Writers

categorical mutex

Listing 4.13: Readers-writers initialization

```
int readers = 0
mutex = Semaphore(1)
roomEmpty = Semaphore(1)
```

Listing 4.14: Writers solution

```
roomEmpty.wait()
    critical section for writers
roomEmpty.signal()
```

Listing 4.15: Readers solution

```
mutex.wait()
    readers += 1
    if readers == 1:
        roomEmpty.wait()   # first in locks
mutex.signal()

# critical section for readers

mutex.wait()
    readers -= 1
    if readers == 0:
        roomEmpty.signal() # last out unlocks
mutex.signal()
```

→ "lightswitch" pattern

Listing 4.16: Lightswitch definition

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

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

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

Listing 4.17: Readers-writers initialization

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

`readLightswitch` is a shared `Lightswitch` object whose counter is initially zero.

Listing 4.18: Readers-writers solution (reader)

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

recall criteria:

1. no starvation
2. bounded waiting

… but ***writer can starve***!

need a mechanism for the writer to prevent new readers from getting "around" it (and into the room)

i.e., "single-file" entry

Listing 4.19: No-starve readers-writers initialization

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

Listing 4.20: No-starve writer solution

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

roomEmpty.signal()
```

Listing 4.21: No-starve reader solution

```
turnstile.wait()
turnstile.signal()

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

exercise for the reader: *writer priority*?

bounded waiting?

- simple if we assume that threads blocking on a semaphore are queued (FIFO)
- i.e., thread blocking longest is woken next
- but semaphore semantics *don't require this*

→ *FIFO queue* pattern

goal: use semaphores to build a thread-safe FIFO wait queue

given: non-thread-safe queue

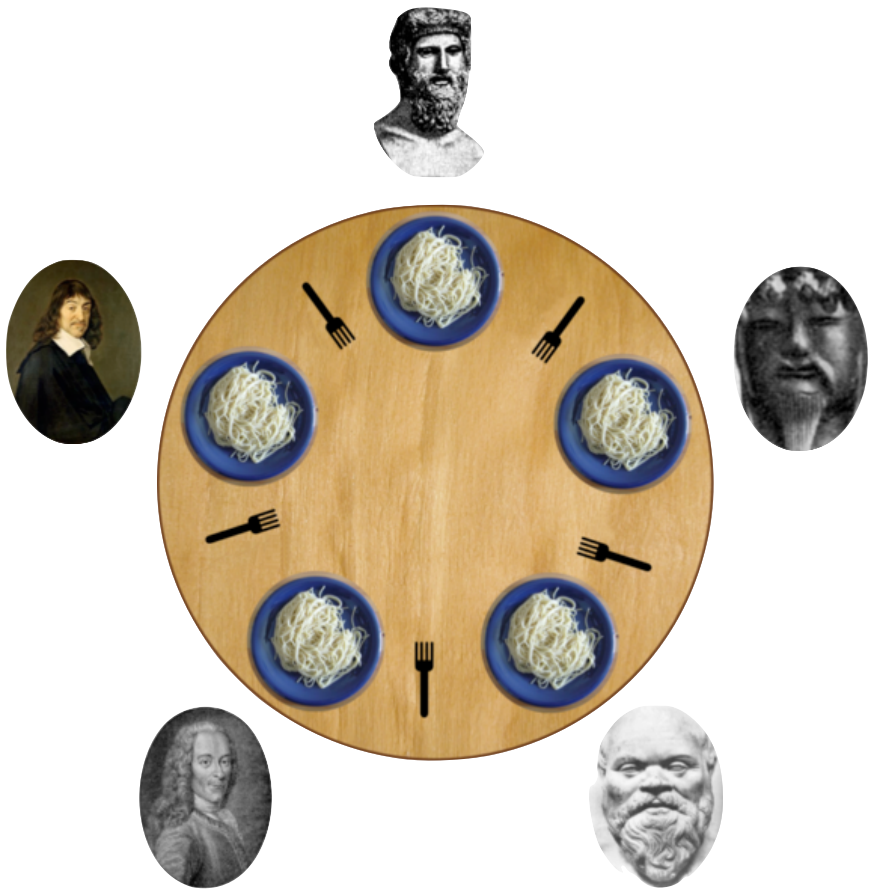
approach:

- protect queue with shared mutex
- each thread enqueues its own *thread-local* semaphores and blocks on it
- to signal, dequeue & unblock a semaphore

```python
class FifoSem:
    def __init__(self, val):
        self.val   = val            # FifoSem's semaphore value
        self.mutex = Semaphore(1) # possibly non-FIFO semaphore
        self.queue = deque()      # non-thread-safe queue

    def wait(self):
        barrier = Semaphore(0)    # thread-local semaphore
        block = False
        self.mutex.wait()         # modify val & queue in mutex
            self.val -= 1
            if self.val < 0:
                self.queue.append(barrier)
                block = True
        self.mutex.signal()
        if block:
            barrier.wait()        # block outside mutex!

    def signal(self):
        self.mutex.wait()         # modify val & queue in mutex
            self.val += 1
            if self.queue:
                barrier = self.queue.popleft() # FIFO!
                barrier.signal()
        self.mutex.signal()
```

henceforth, we will assume that all semaphores have *built-in* FIFO semantics

III. "Dining Philosophers" problem



typical setup: protect shared resources with semaphores

Listing 4.30: Variables for dining philosophers

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

Listing 4.29: Which fork?

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

solution requirements:

1. each fork held by one phil at a time
2. no deadlock
3. no philosopher may starve
4. max concurrency should be possible

Naive solution:

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

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

possible deadlock!

Solution 2: global mutex

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

no starvation & max concurrency?

- may prohibit a philosopher from eating when his forks are available

Solution 3: limit # diners

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

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

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

no starvation & max concurrency?

Solution 4: leftie(s) vs. rightie(s)

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

vs. (at least one of each)

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

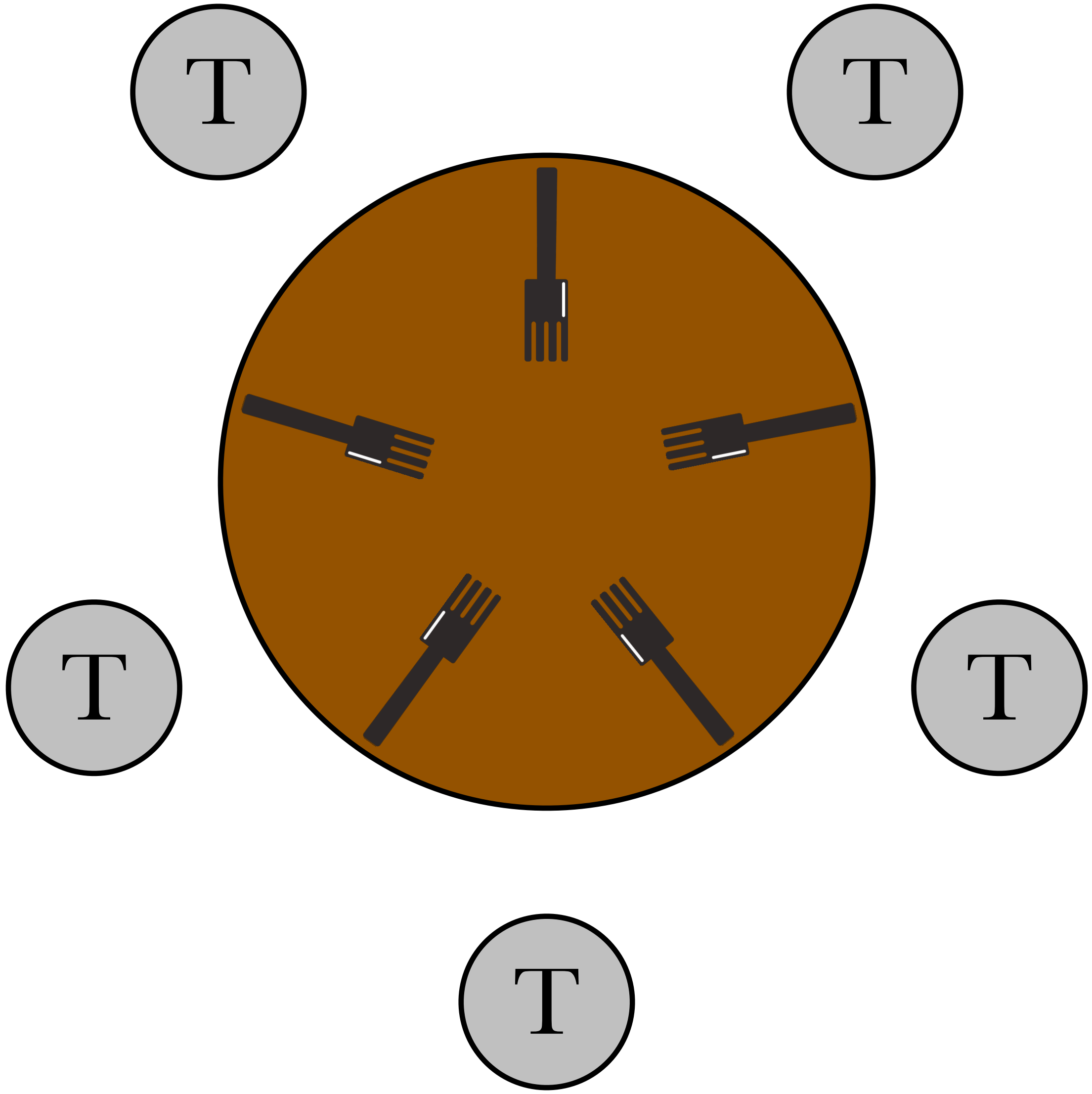
no starvation & max concurrency?

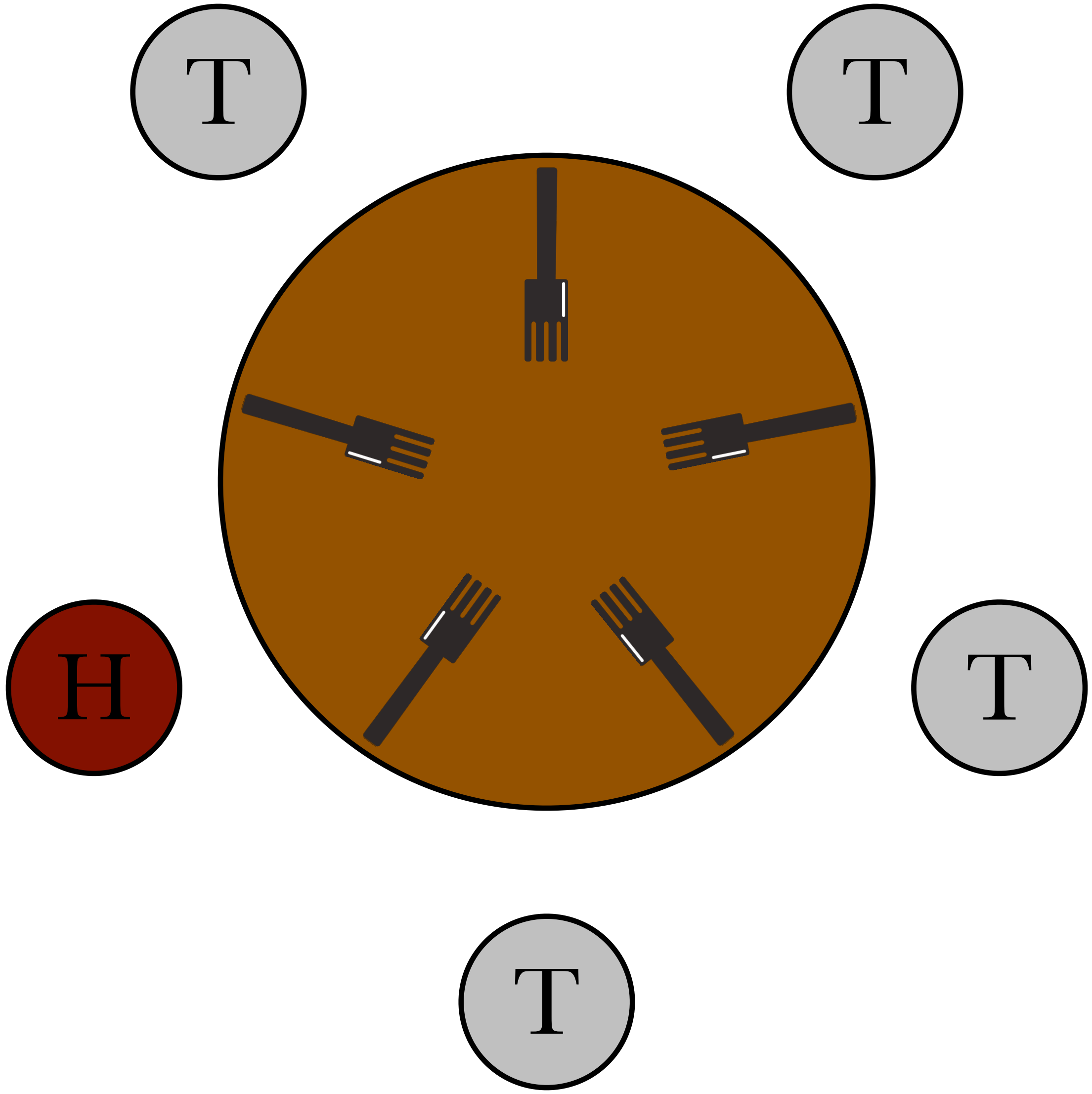
Solution 4: Tanenbaum's solution

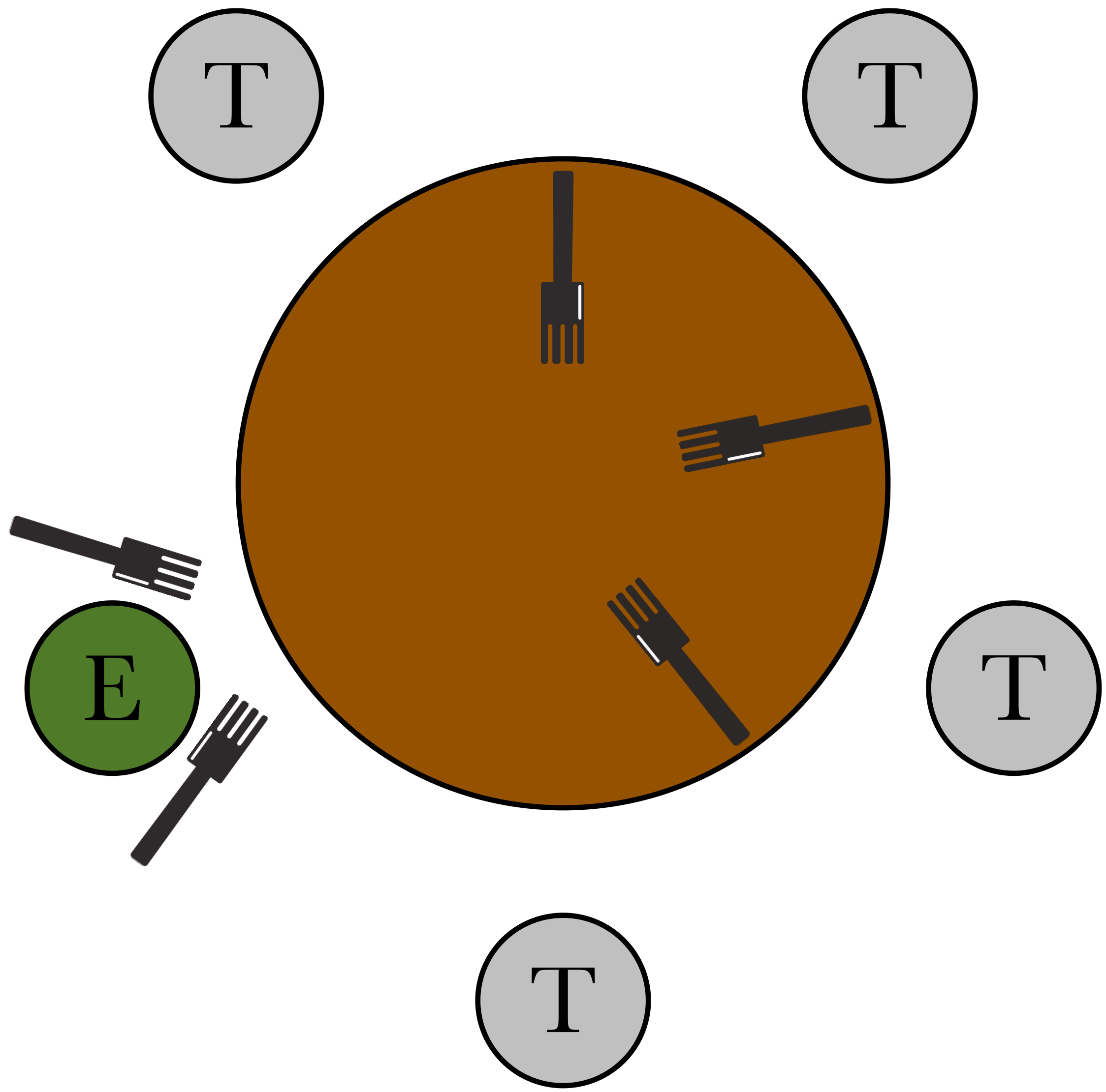
```
state = ['thinking'] * 5
sem = [Semaphore(0) for i in range(5)]
mutex = Semaphore(1)
```

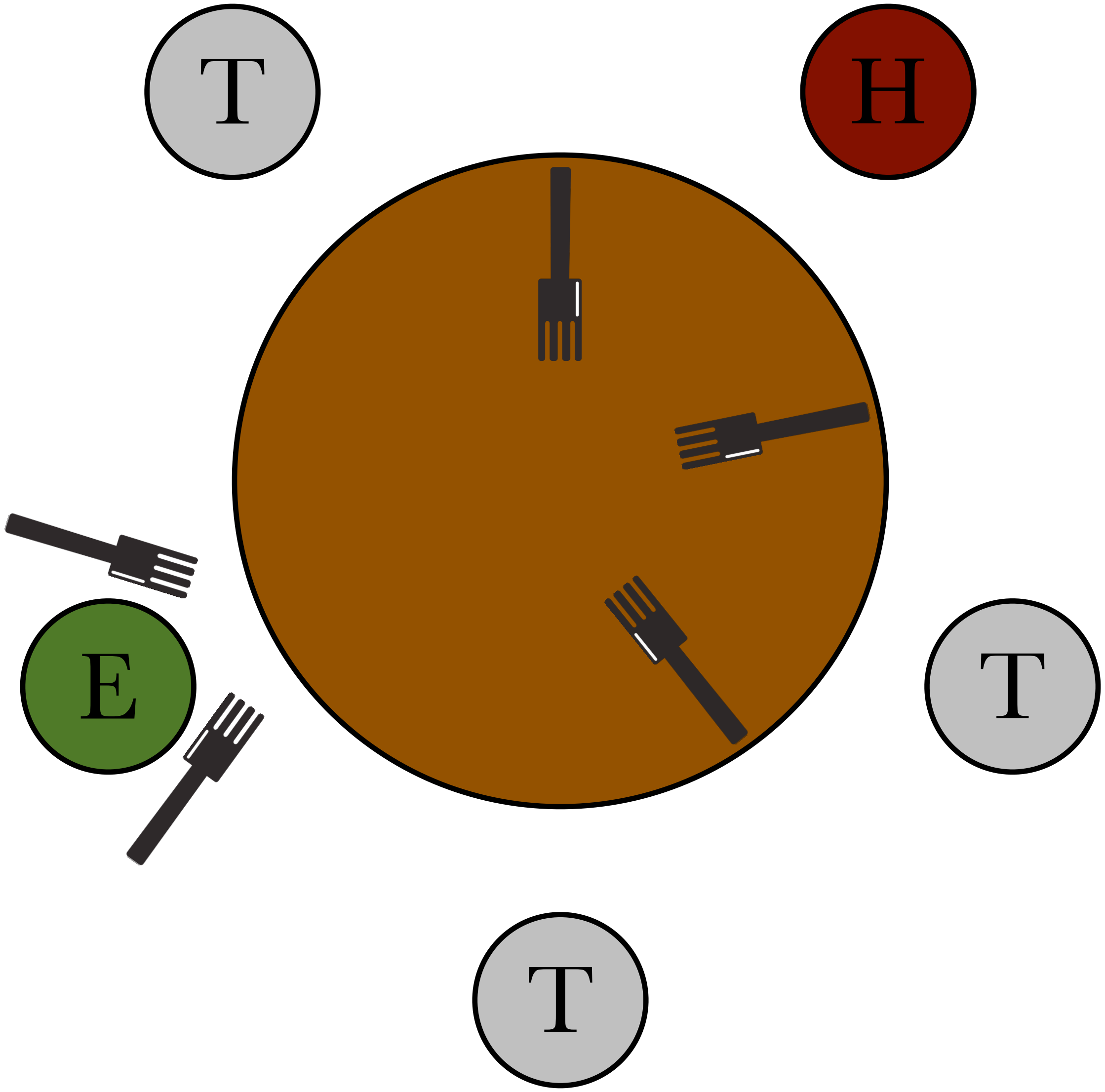
```
def get_fork(i):
    mutex.wait()
        state[i] = 'hungry'
        test(i)                 # check neighbors' states
    mutex.signal()
    sem[i].wait()               # wait on my own semaphore

def put_fork(i):
    mutex.wait()
        state[i] = 'thinking'
        test(right(i))          # signal neighbors if they can eat
        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()         # this signals me OR a neighbor
```

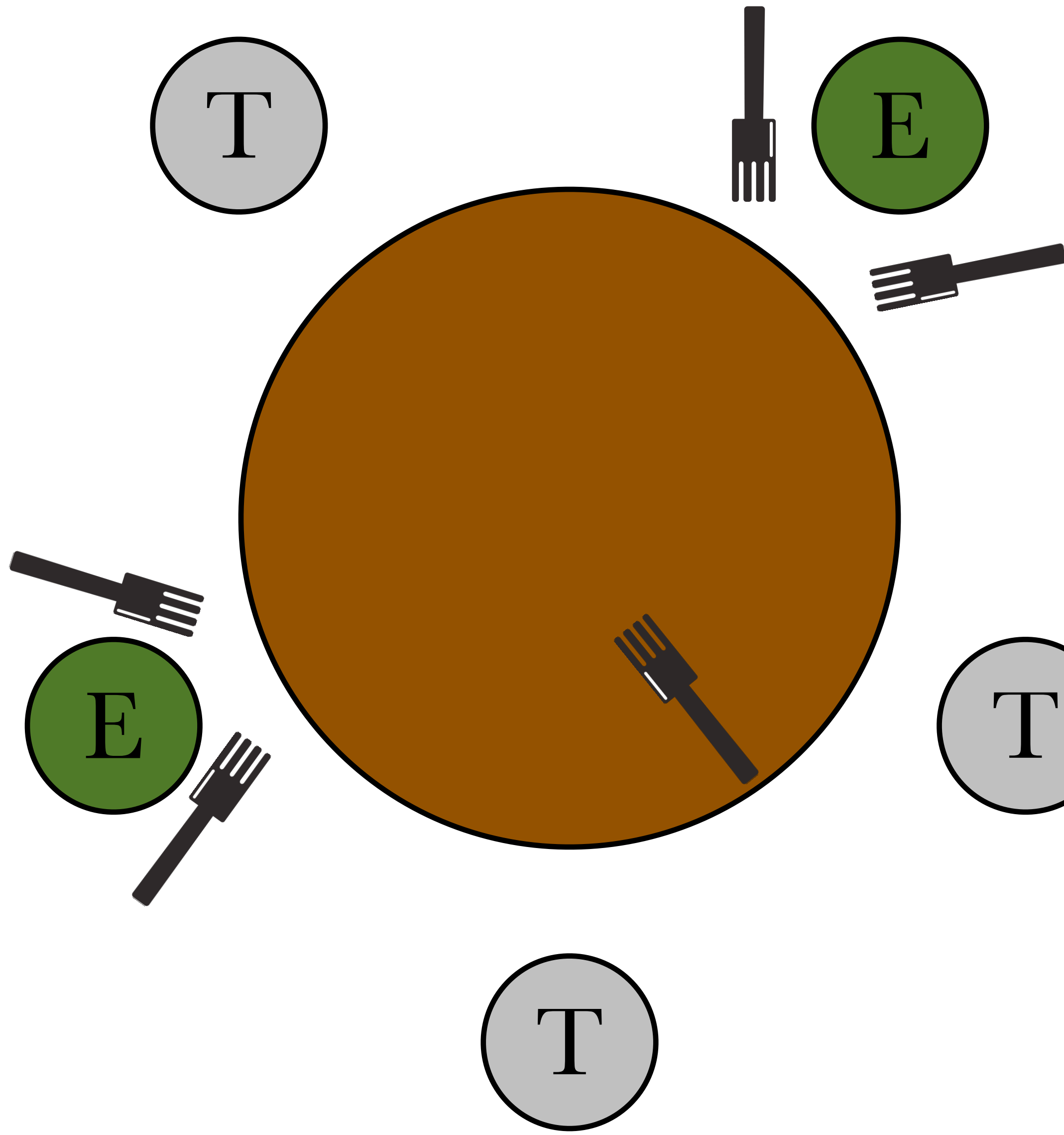
no starvation & max concurrency?

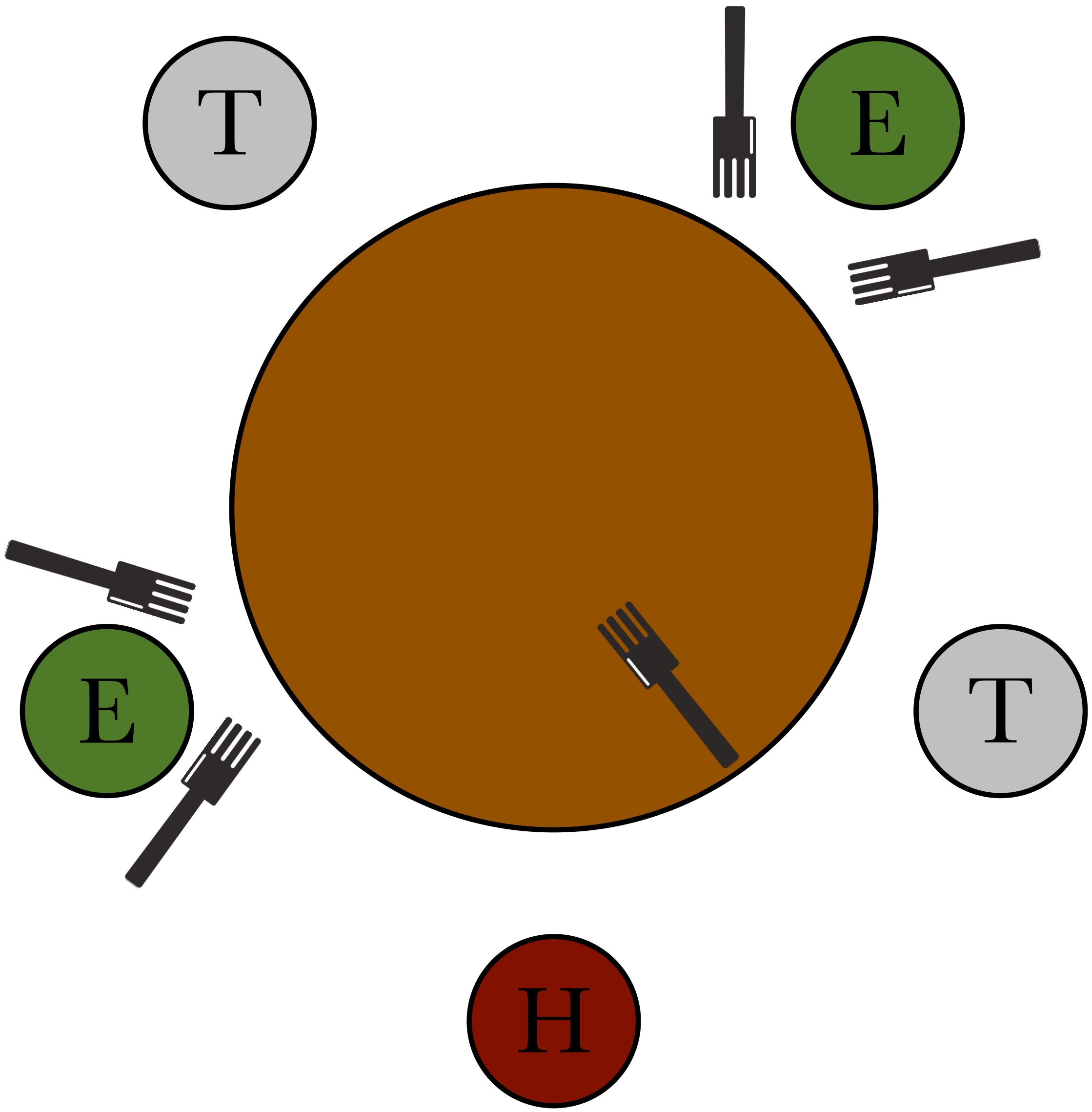

T
T
T
T
T

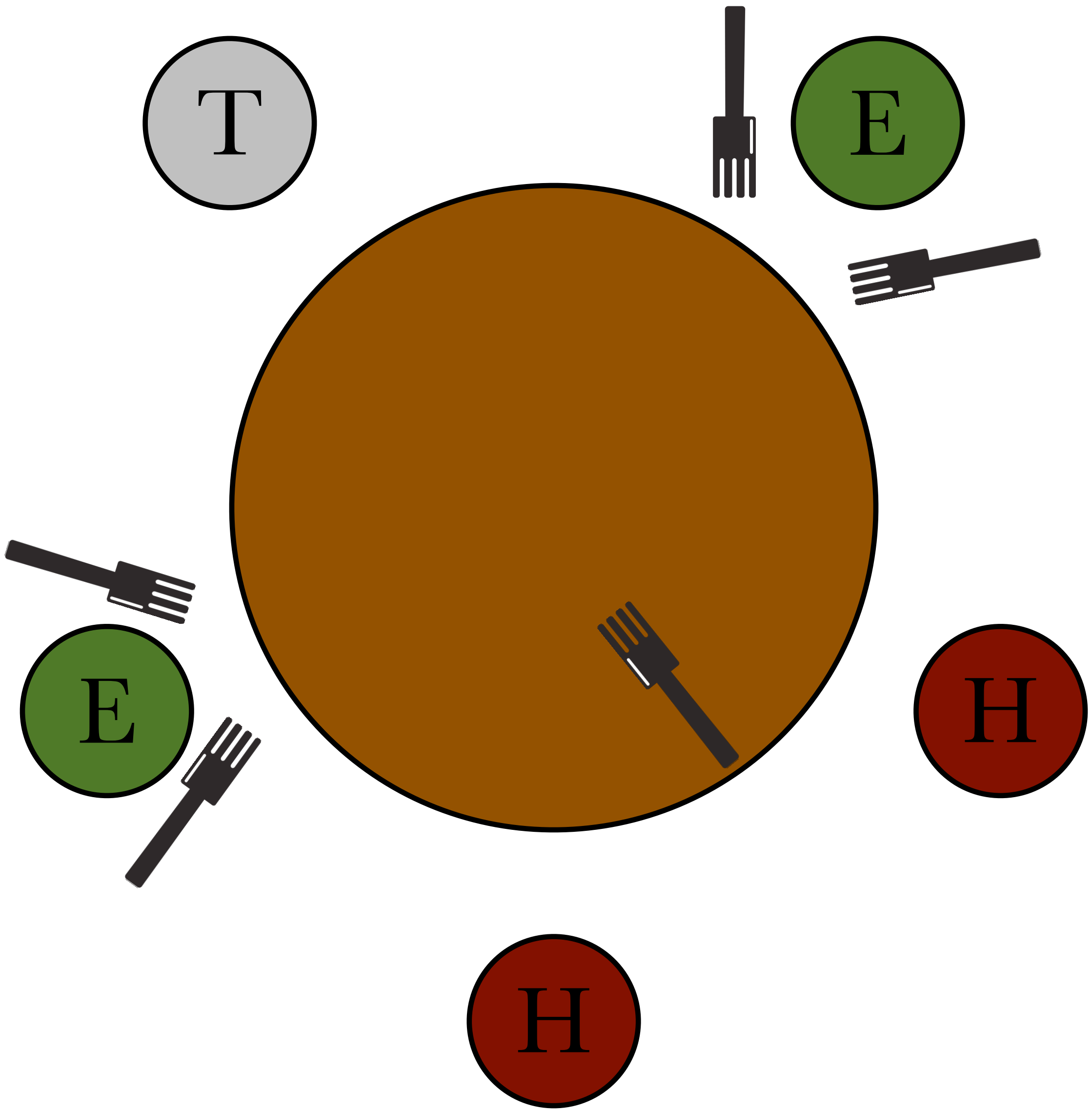

T
T
H
T
T
T

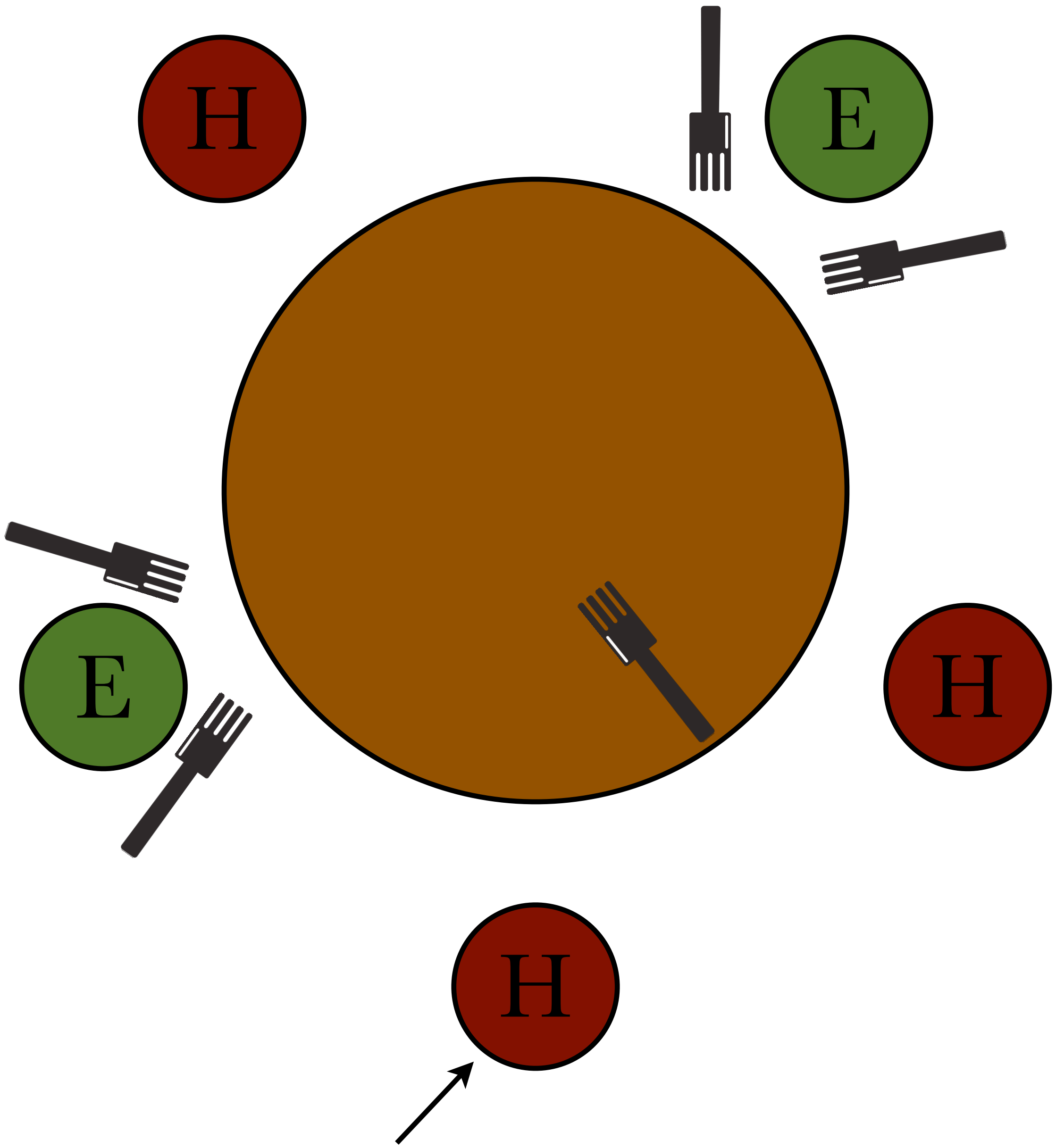

T
T
E
T
T


T
H
E
T
T

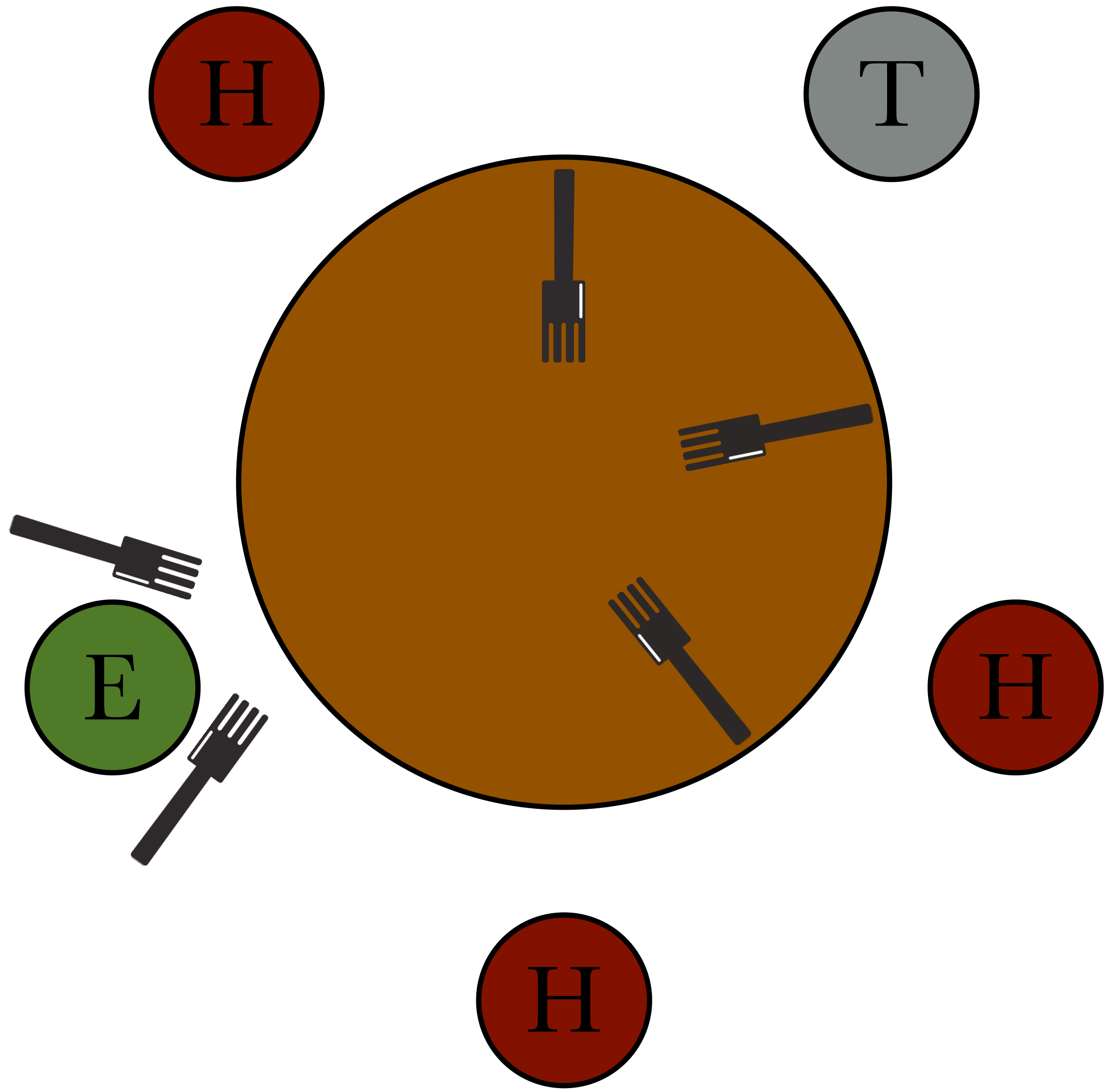

T
E
E
T
T

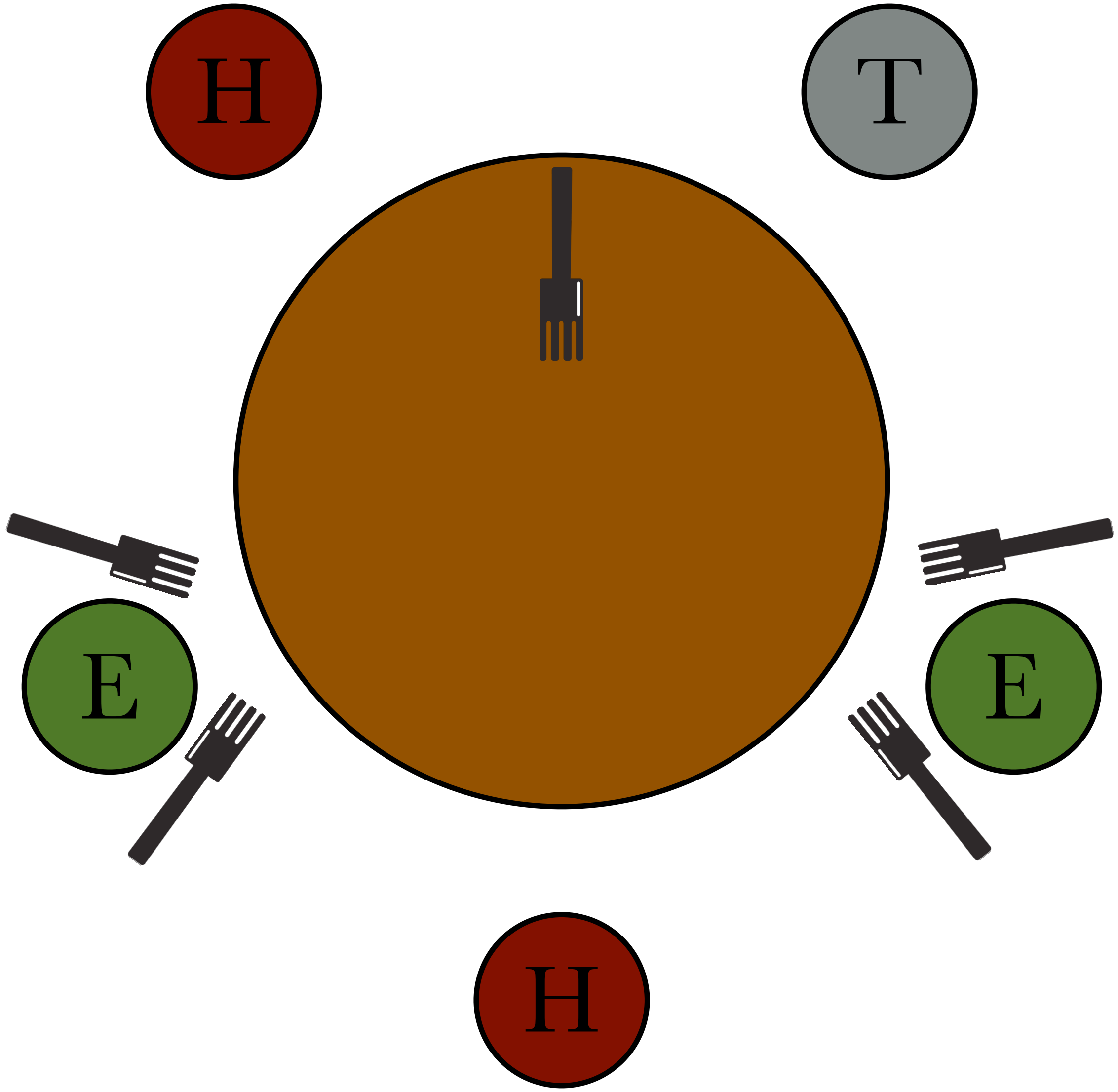

T
E
E
T
H

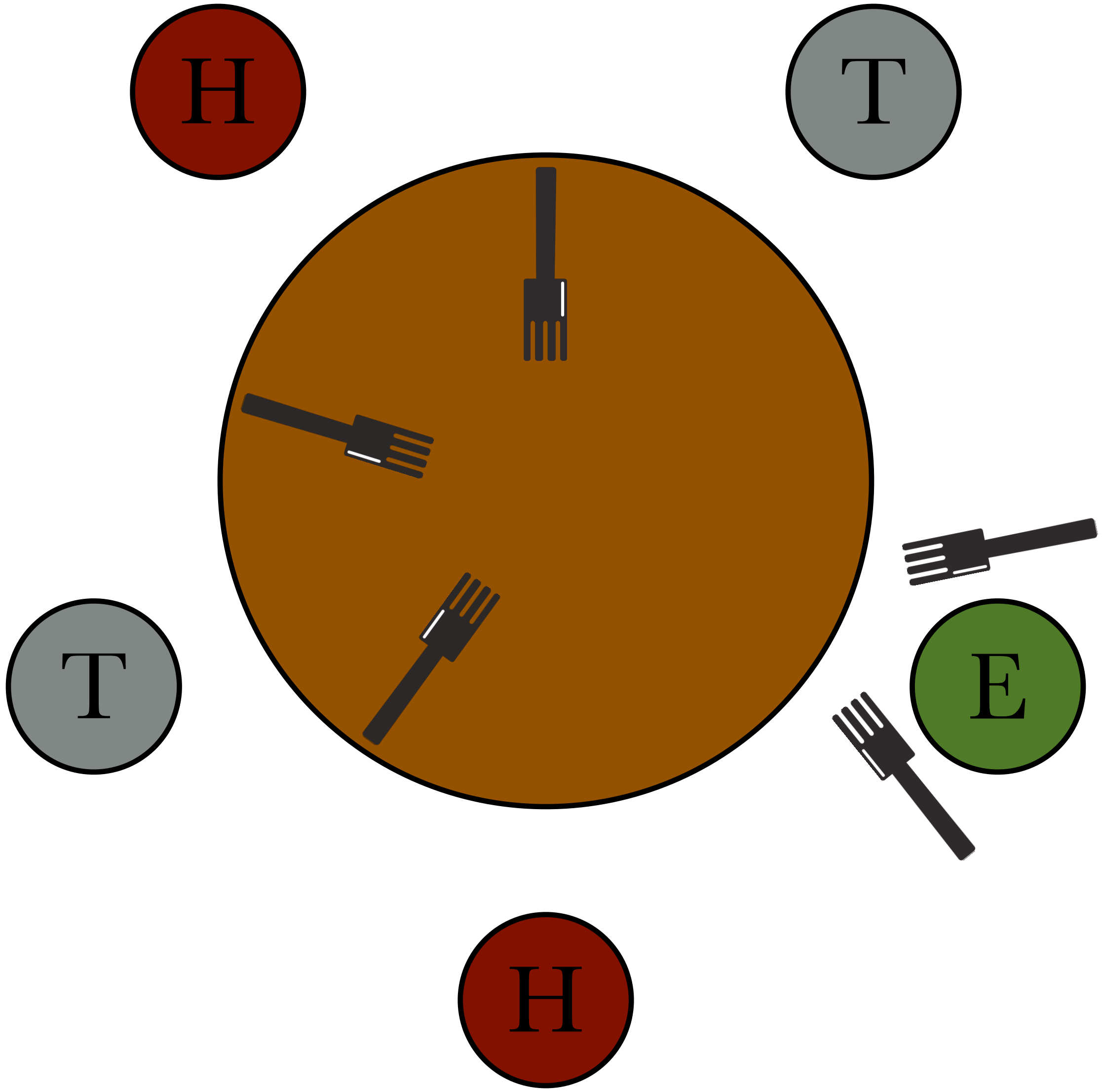

T
E
E
H
H

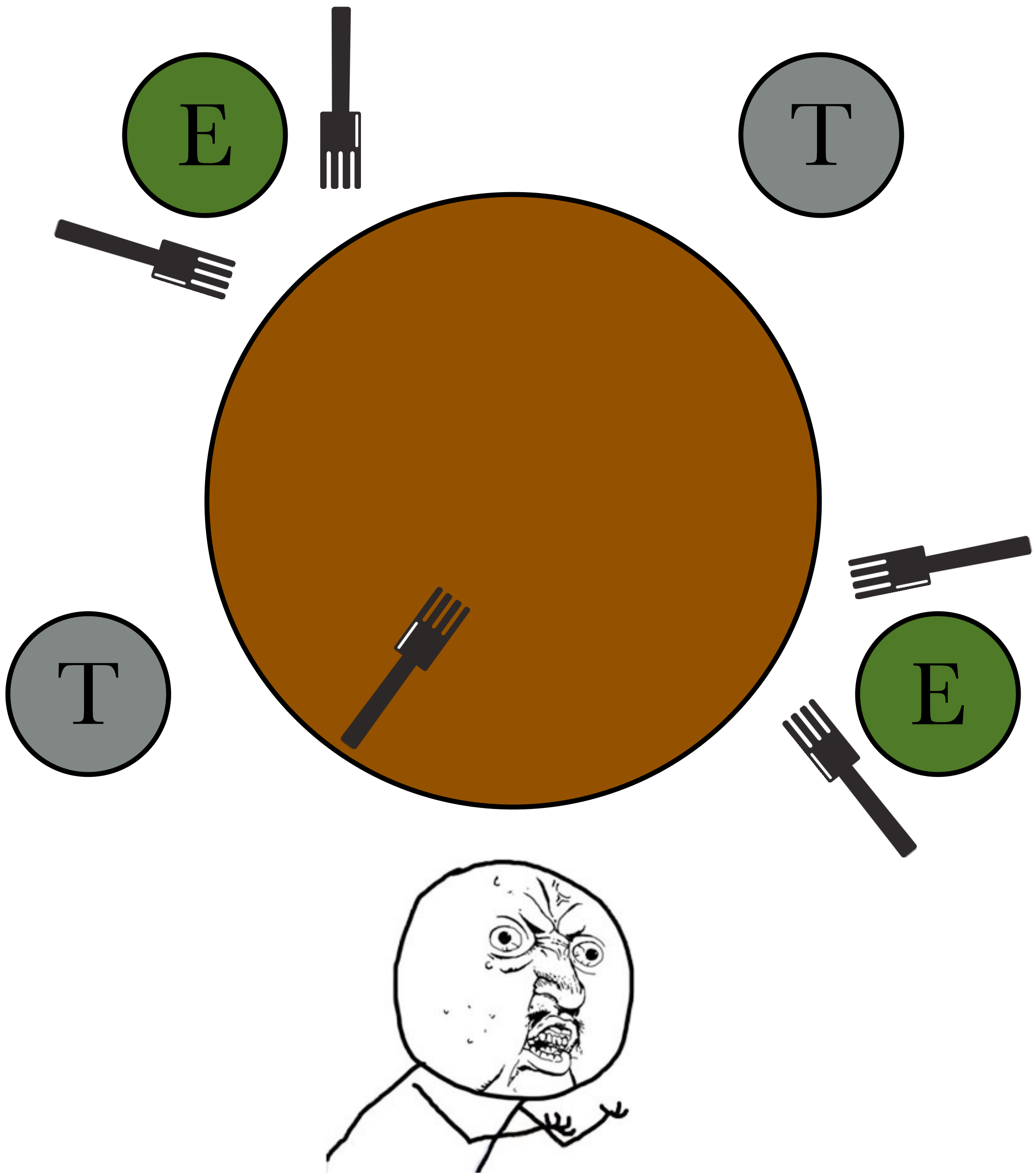


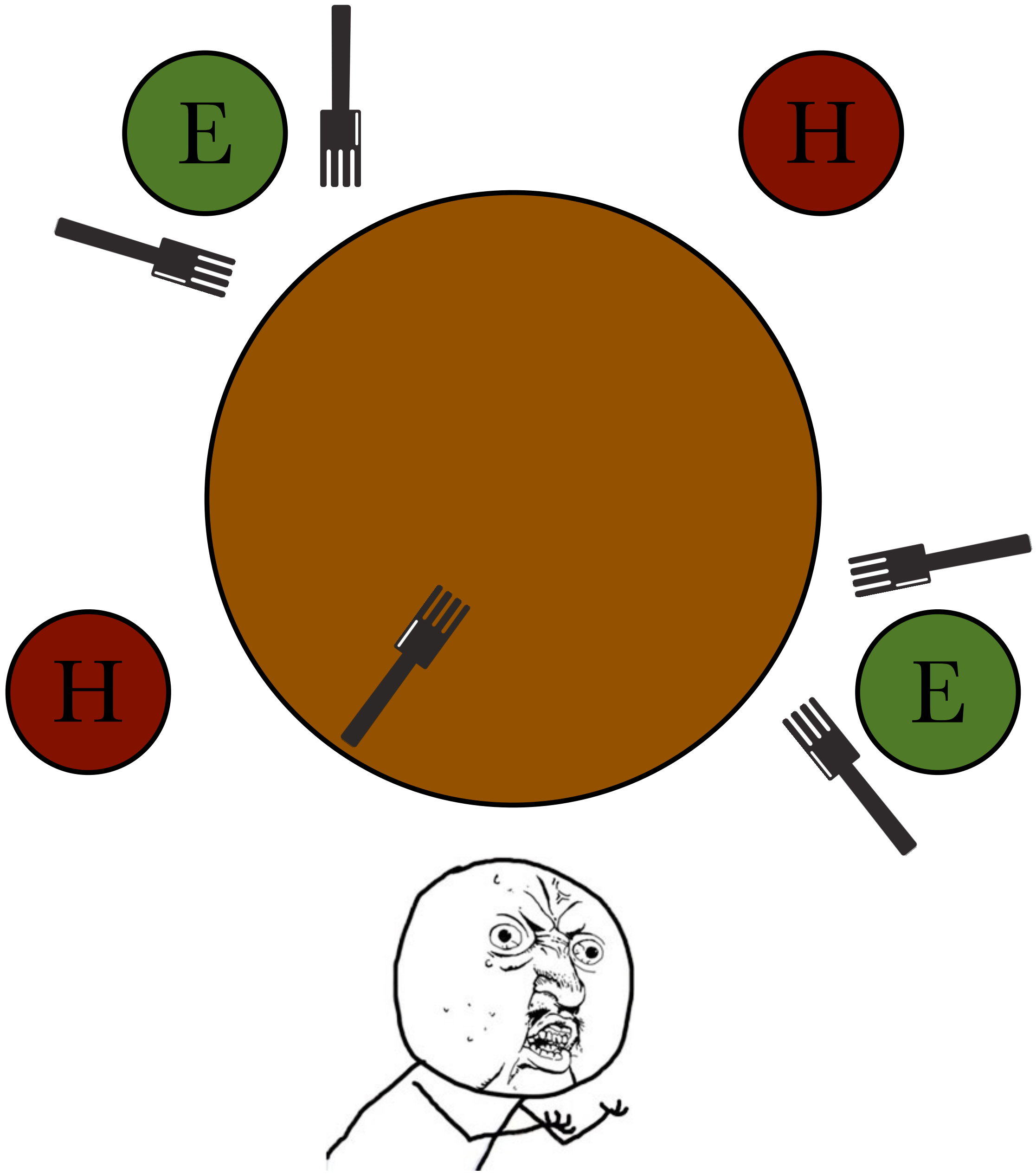
(let's mess with this guy)

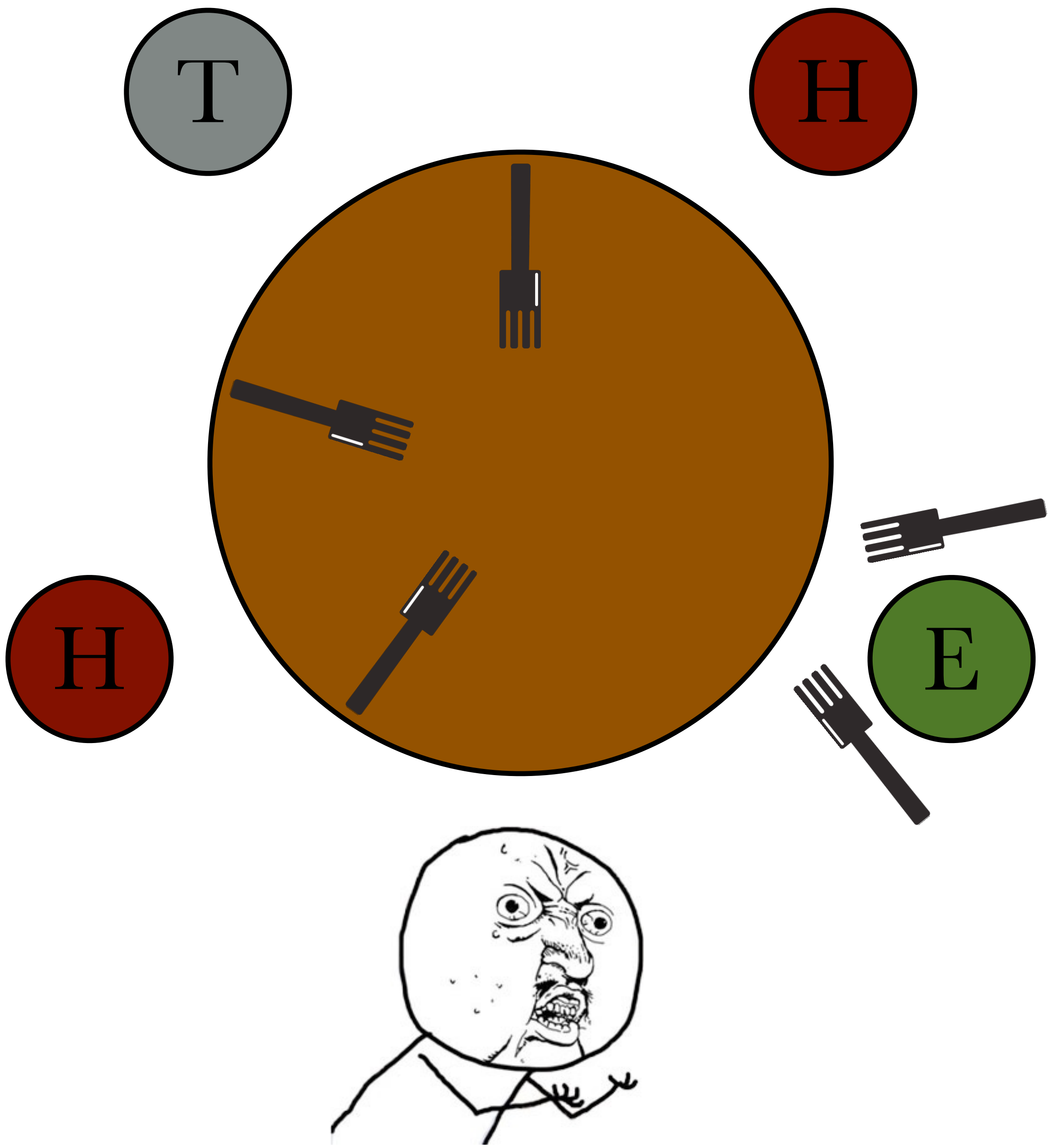

H
T
E
H
H

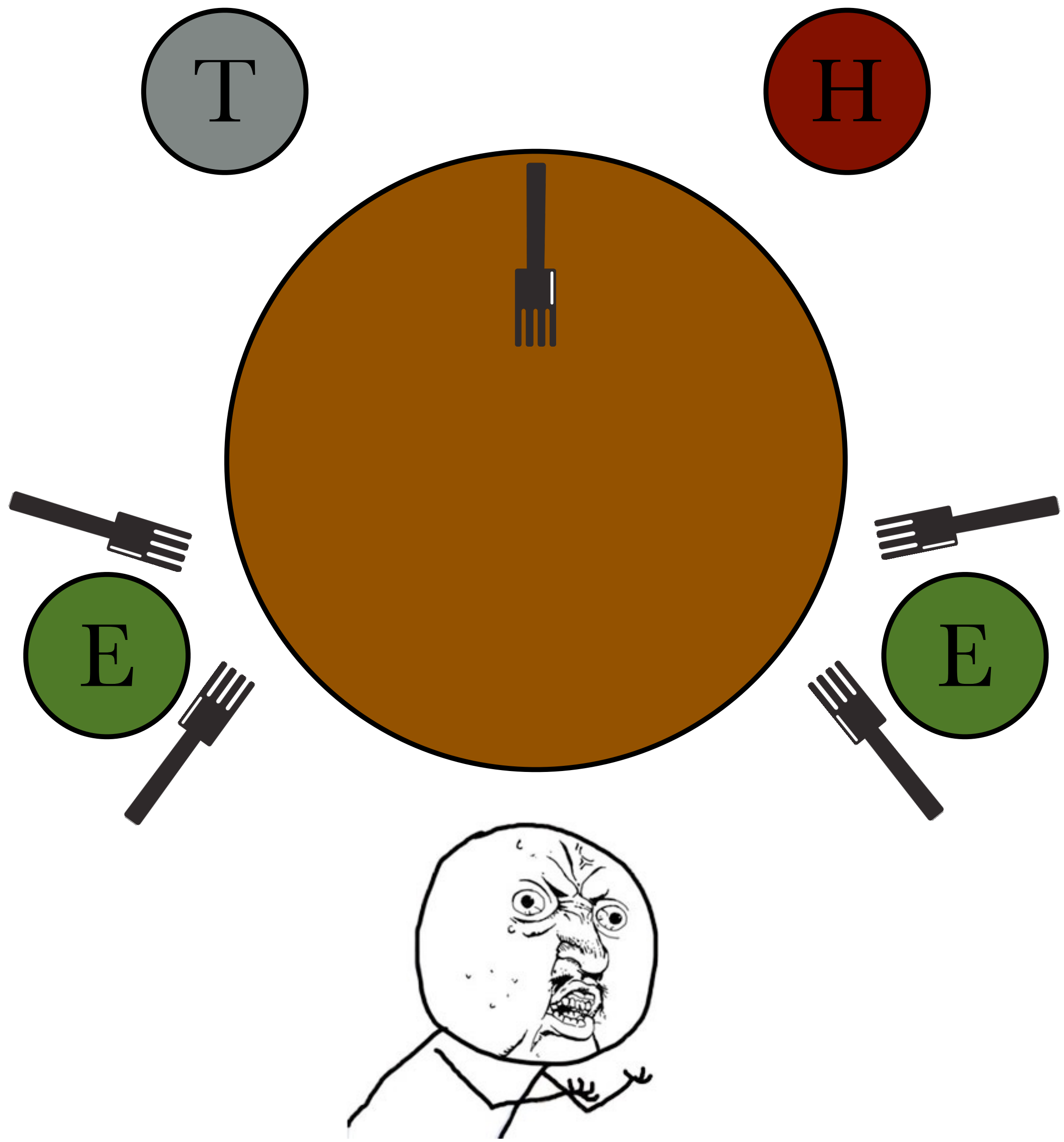

H
T
E
E
H

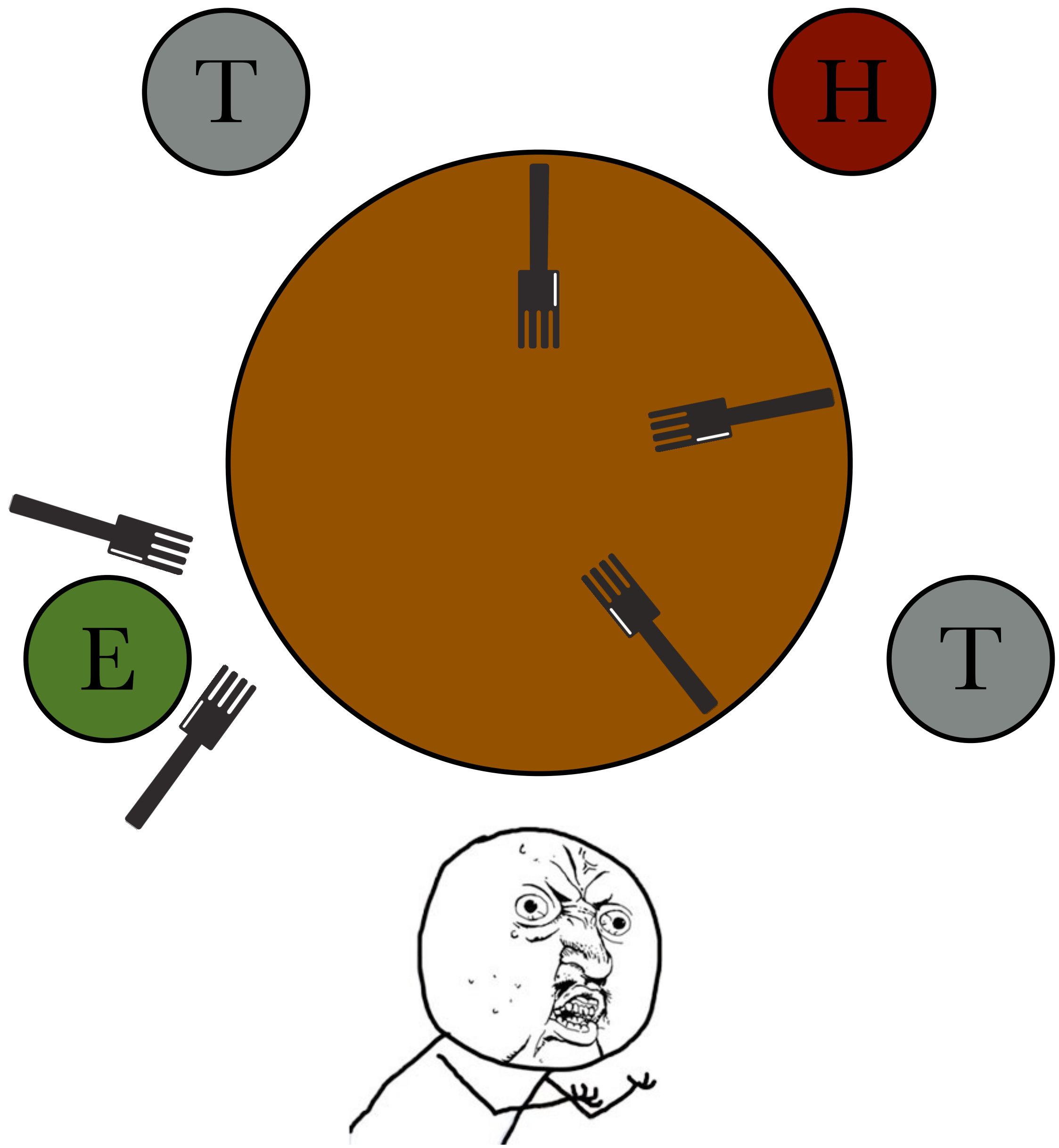

H
T
T
E
H

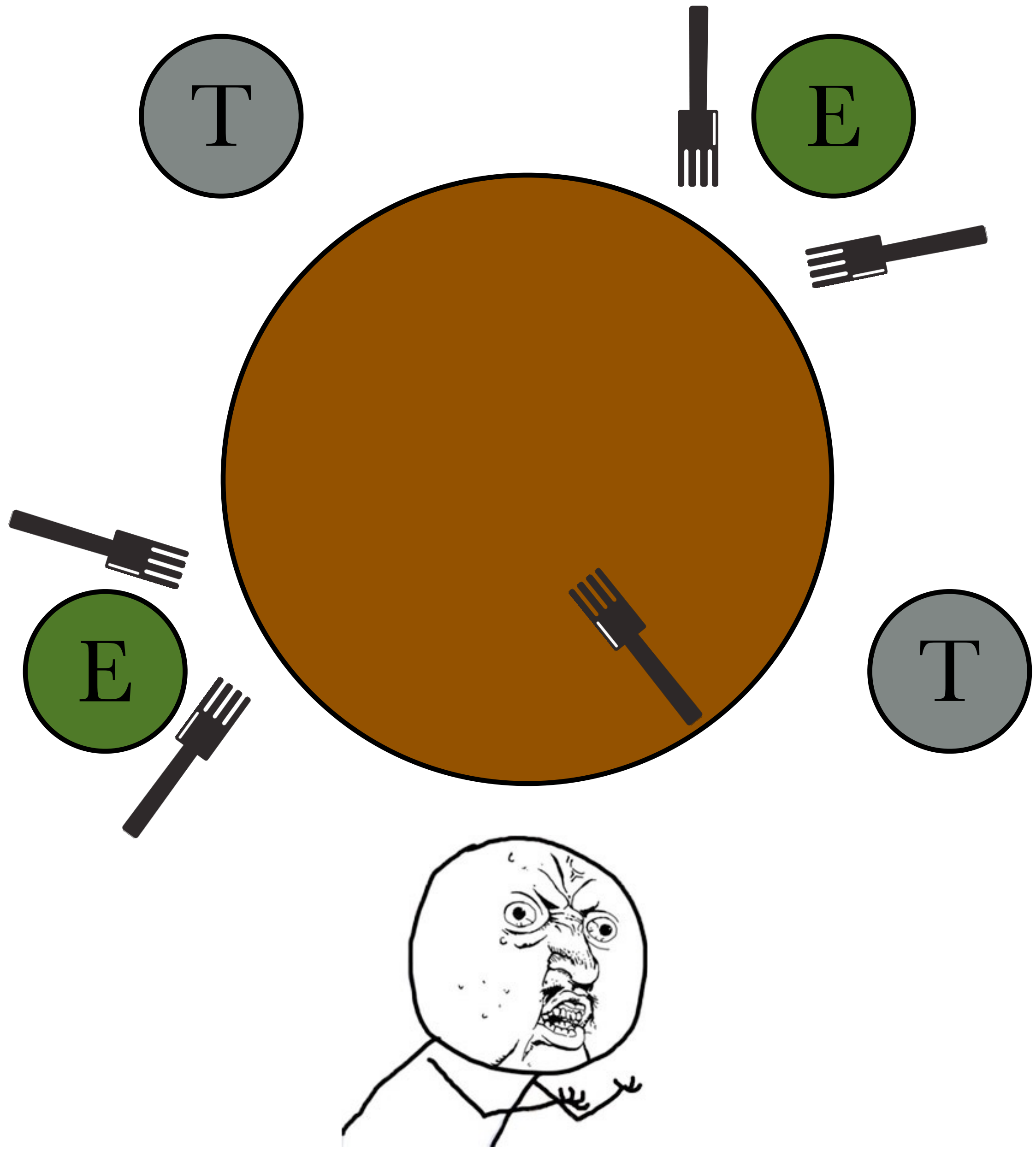

E
T
T
E

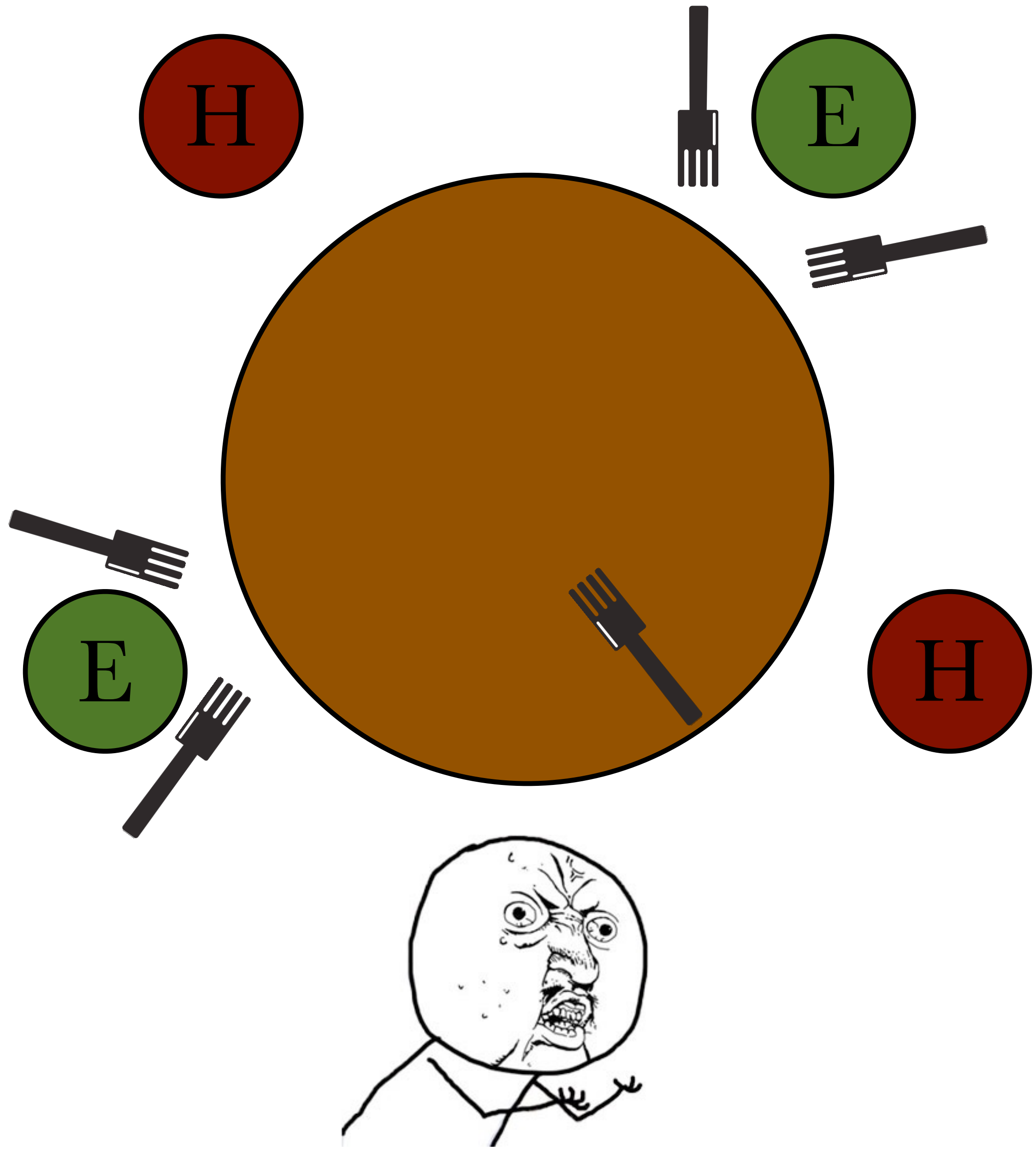

E
H
H
E

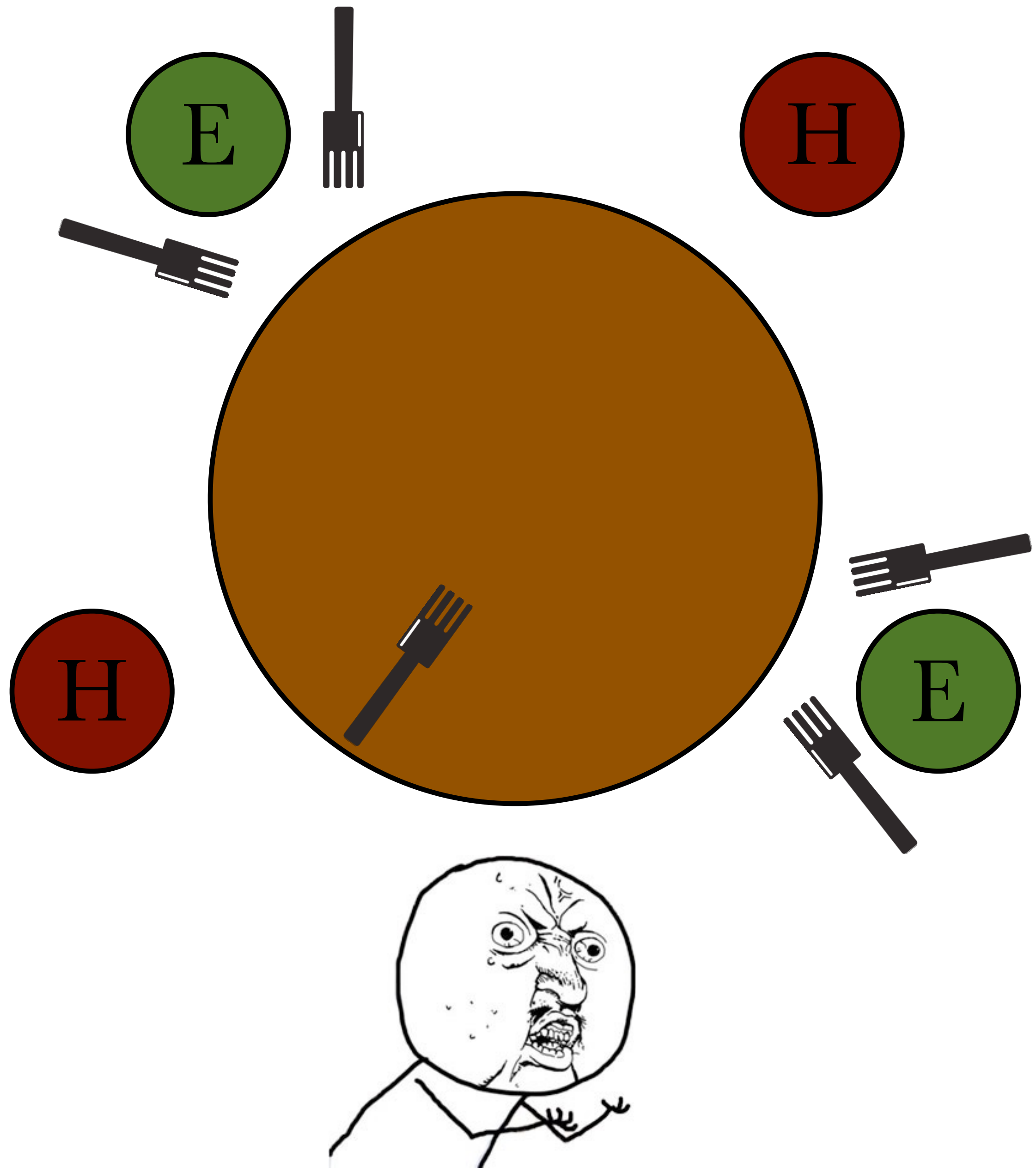

T
H
H
E

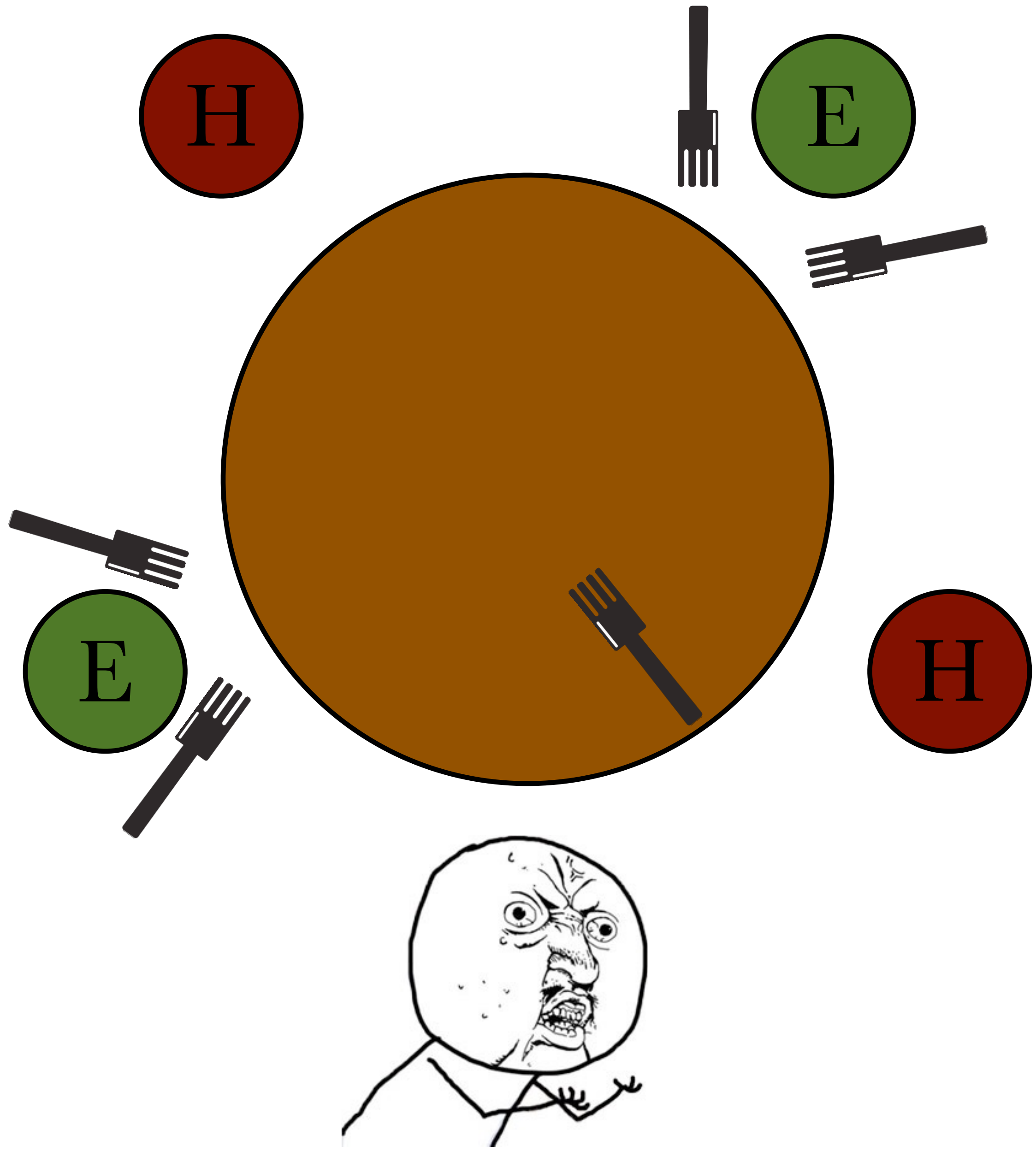

T
H
E
E

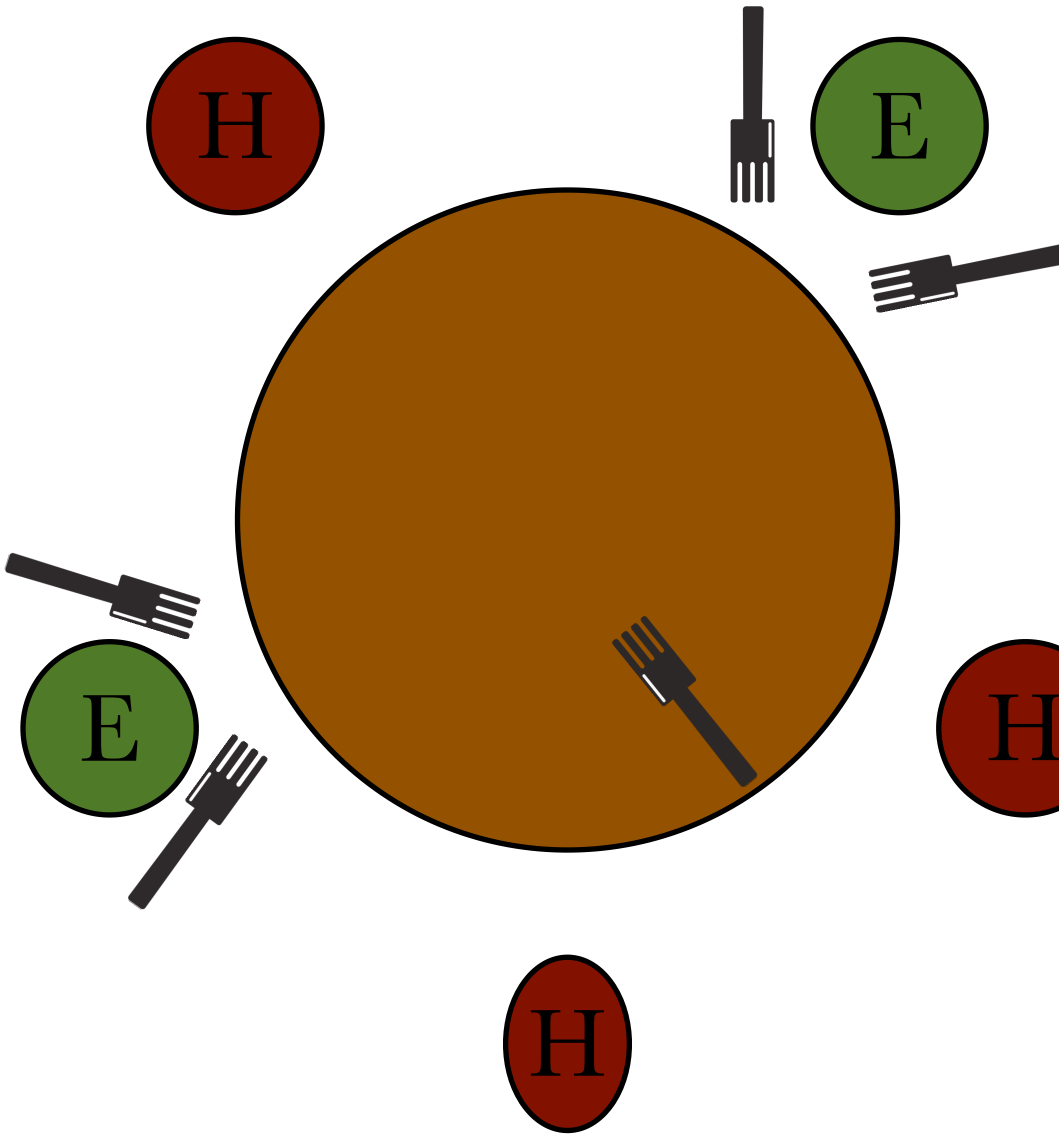

T
H
E
T


T
E
E
T


H
E
E
H


E
H
H
E


H
E
E
H



(starves)

moral: synchronization problems are *insidious*!

IV. Dining Savages

"Mugulu, how often have I told you not to play with your food?"

A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary. When a savage wants to eat, he helps himself from the pot, unless it is empty. If the pot is empty, the savage wakes up the cook and then waits until the cook has refilled the pot.

Listing 5.1: Unsynchronized savage code

```
while True:
    getServingFromPot()
    eat()
```

And one cook thread runs this code:

Listing 5.2: Unsynchronized cook code

```
while True:
    putServingsInPot(M)
```

Listing 5.1: Unsynchronized savage code

```
while True:
    getServingFromPot()
    eat()
```

And one cook thread runs this code:

Listing 5.2: Unsynchronized cook code

```
while True:
    putServingsInPot(M)
```

rules:

- savages cannot invoke `getServingFromPot` if the pot is empty
- the cook can invoke `putServingsInPot` only if the pot is empty

hint:

```
servings = 0
mutex    = Semaphore(1)
emptyPot = Semaphore(0)
fullPot  = Semaphore(0)
```

Listing 5.1: Unsynchronized savage code

```
while True:
    getServingFromPot()
    eat()
```

And one cook thread runs this code:

Listing 5.2: Unsynchronized cook code

```
while True:
    putServingsInPot(M)
```

Listing 5.4: Dining Savages solution (cook)

```
while True:
    emptyPot.wait()
    putServingsInPot(M)
    fullPot.signal()
```

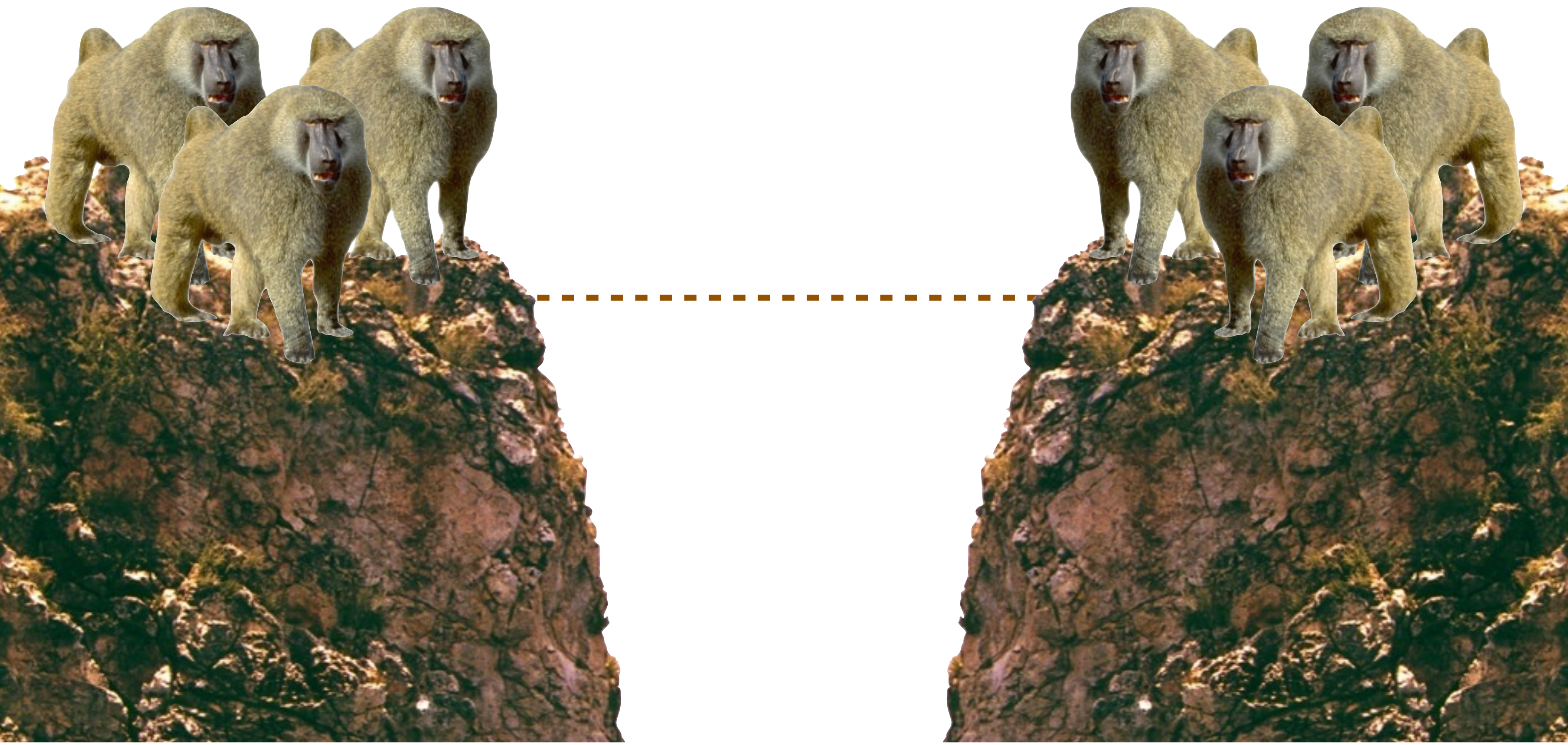
Listing 5.5: Dining Savages solution (savage)

```
while True:
    mutex.wait()
        if servings == 0:
            emptyPot.signal()
            fullPot.wait()
            servings = M
        servings -= 1
        getServingFromPot()
    mutex.signal()

    eat()
```

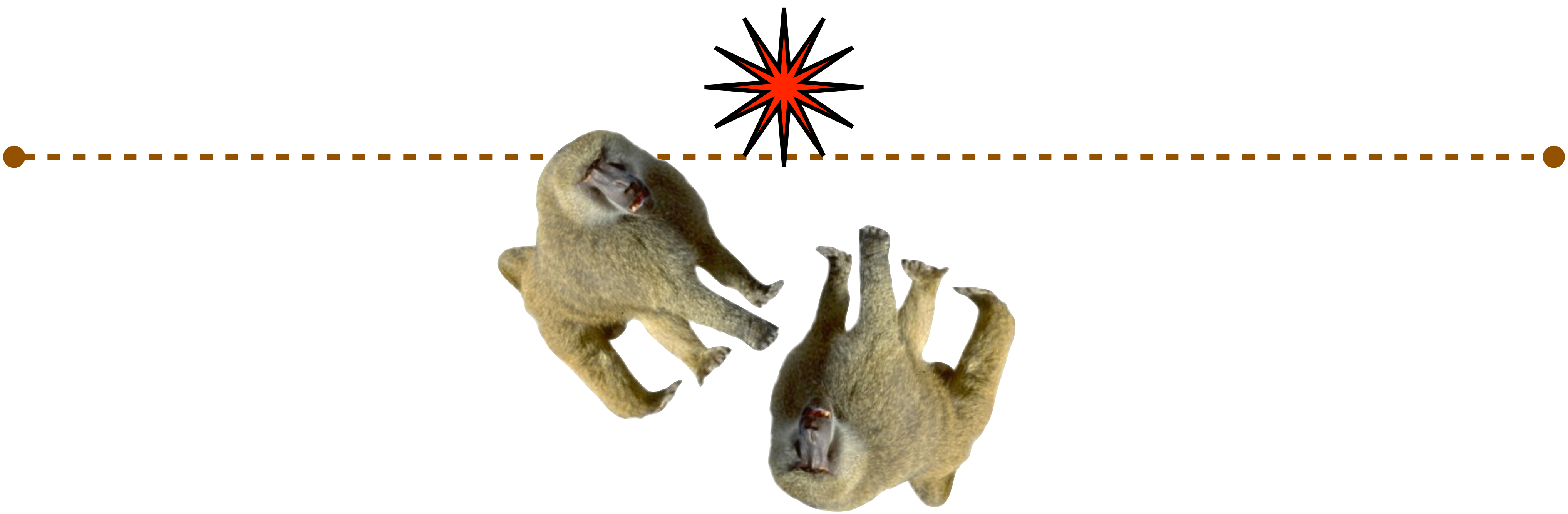
shared servings counter → *scoreboard* pattern

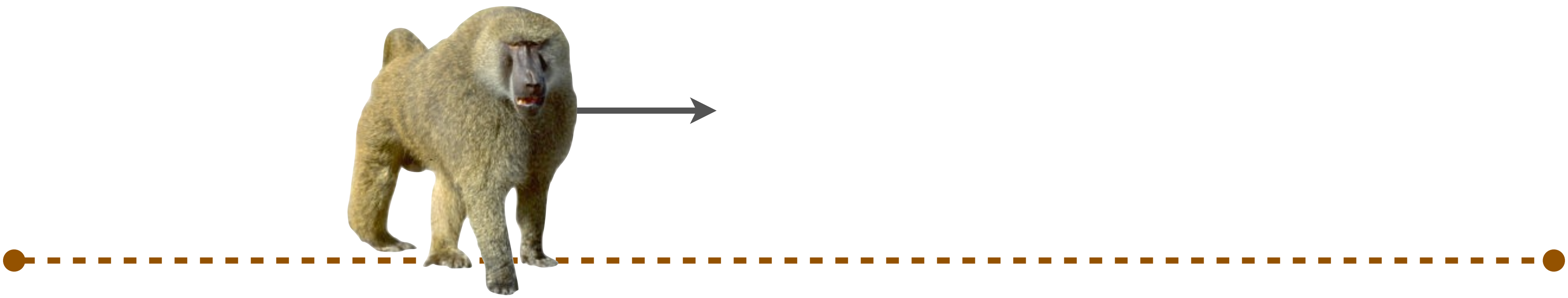
- arriving threads check value of scoreboard to determine system state
- note: scoreboard may consist of more than one variable

V. Baboon Crossing

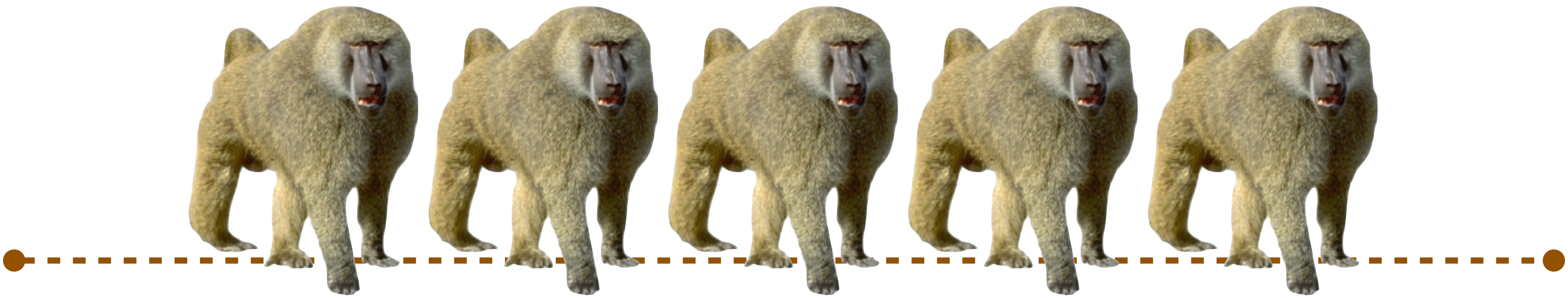



west
east

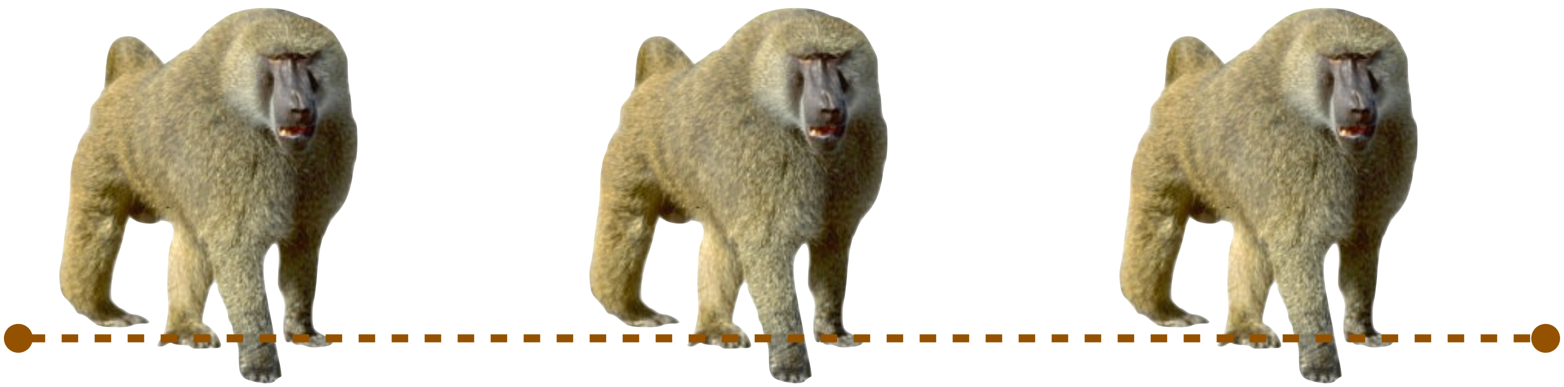




gurantee rope mutex



max of 5 at a time



no starvation

solution consists of east&west baboon threads:

1. categorical mutex
2. max of 5 on rope
3. no starvation

unsynchronized baboon code (identical for both sides)

```
while True:
    climbOnRope()
    crossChasm()
```

hint:

```
multiplex  = Semaphore(5)
turnstile  = Semaphore(1)
rope       = Semaphore(1)
e_switch   = Lightswitch()
w_switch   = Lightswitch()
```

Reminder: Lightswitch ADT

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

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

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

```
multiplex  = Semaphore(5)
turnstile  = Semaphore(1)
rope       = Semaphore(1)
e_switch   = Lightswitch()
w_switch   = Lightswitch()
```

```
while True:
    # west side
    turnstile.wait()
    w_switch.lock(rope)
    turnstile.signal()

    multiplex.wait()
    climbOnRope()
    crossChasm()
    multiplex.signal()

    w_switch.unlock(rope)
```

```
while True:
    # east side
    turnstile.wait()
    e_switch.lock(rope)
    turnstile.signal()

    multiplex.wait()
    climbOnRope()
    crossChasm()
    multiplex.signal()

    e_switch.unlock(rope)
```

```
multiplex   = Semaphore(5)
turnstile   = Semaphore(1)
rope        = Semaphore(1)
mutex_east = Semaphore(1)
mutex_west = Semaphore(1)
east_count = west_count = 0
```

```
# west side
turnstile.wait()
mutex_west.wait()
  west_count++
  if west_count == 1:
    rope.wait()
mutex_west.signal()
turnstile.signal()

multiplex.wait()
  # cross the chasm
multiplex.signal()

mutex_west.wait()
  west_count--
  if west_count == 0:
    rope.signal()
mutex_west.signal()
```

```
# east side
turnstile.wait()
mutex_east.wait()
  east_count++
  if east_count == 1:
    rope.wait()
mutex_east.signal()
turnstile.signal()

multiplex.wait()
  # cross the chasm
multiplex.signal()

mutex_east.wait()
  east_count--
  if east_count == 0:
    rope.signal()
mutex_east.signal()
```

… many, many more contrived problems await you in the little book of semaphores!