Week 5

Multithreading

Oana Balmau
February 2, 2023

Goals for today

- Practice multi-threading synchronization
  - Join Implementation
  - Dining Philosophers
  - Alice and Bob are back with pet dragons!
- Q&A for first module

Problem 1: Join Implementation

- Implement the equivalent of pthread_join seen in class.

Reminder: pthread_join(threadid, &status)

- Wait for thread threadid to exit
- Receive status, if any

Ordering Example: Join

```
pthread_t p1, p2;
pthread_create(&p1, NULL, mythread, "A");
pthread_create(&p2, NULL, mythread, "B");

// join waits for the threads to finish
pthread_join(p1, NULL);
pthread_join(p2, NULL);
printf("Done!\n");
return 0;
```

how to implement join?

Reminder: Condition Variables

Condition Variable ~= queue of waiting threads

B waits for a signal on CV before running

- wait(CV, …)

A sends signal to CV when time for **B** to run

- signal(CV, …)

Reminder: Condition Variables

wait(cond_t *cv, mutex_t *lock)

- assumes the lock is held when wait() is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock *immediately* before returning

signal(cond_t *cv)

- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return, doing nothing

Thinking time; How to implement join?

```
pthread_t p1, p2;
pthread_create(&p1, NULL, mythread, "A");
pthread_create(&p2, NULL, mythread, "B");

// join waits for the threads to finish
pthread_join(p1, NULL);
pthread_join(p2, NULL);
printf("Done!\n");
return 0;
```

`wait``(cond_t *cv, mutex_t *lock)`

- assumes the lock is held when wait() is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock *immediately* before returning

`signal``(cond_t *cv)`

- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return, doing nothing

Hint: Use mutex + condition variables

Join Implementation: Attempt 1

Parent:

```
void thread_join() {
  Mutex_lock(&m);          // x
  Cond_wait(&c, &m);       // y
  Mutex_unlock(&m);        // z
}
```

Child:

```
void thread_exit() {
  Mutex_lock(&m);          // a
  Cond_signal(&c);         // b
  Mutex_unlock(&m);        // c
}
```

Join Implementation: Attempt 1

Parent:

```
void thread_join() {
  Mutex_lock(&m);          // x
  Cond_wait(&c, &m);       // y
  Mutex_unlock(&m);        // z
}
```

Child:

```
void thread_exit() {
  Mutex_lock(&m);          // a
  Cond_signal(&c);         // b
  Mutex_unlock(&m);        // c
}
```

Example schedule:

Parent:	x	y				z
Child:			a	b	c	

Works!

Join Implementation: Attempt 1

Parent:

```
void thread_join() {
  Mutex_lock(&m);            // x
  Cond_wait(&c, &m);         // y
  Mutex_unlock(&m);          // z
}
```

Child:

```
void thread_exit() {
  Mutex_lock(&m);            // a
  Cond_signal(&c);           // b
  Mutex_unlock(&m);          // c
}
```

Example schedule:

Parent:	x	y				z
Child:			a	b	c	

Can you construct a schedule that doesn't work?

Join Implementation: Attempt 1 (incorrect)

Parent:

```
void thread_join() {
  Mutex_lock(&m);          // x
  Cond_wait(&c, &m);       // y
  Mutex_unlock(&m);        // z
}
```

Child:

```
void thread_exit() {
  Mutex_lock(&m);          // a
  Cond_signal(&c);         // b
  Mutex_unlock(&m);        // c
}
```

Example broken schedule:

Parent:				x	y
Child:	a	b	c		

Parent waits forever!

Idea

- **Keep state** in addition to CV’s!
- CV’s are used to signal threads when state changes
- If state is already as needed, thread doesn’t wait for a signal!

Join Implementation: Attempt 2

Parent:

```
void thread_join() {
 Mutex_lock(&m);             // w
 if (done == 0)              // x
   Cond_wait(&c, &m);        // y
 Mutex_unlock(&m);           // z
}
```

Child:

```
void thread_exit() {
  done = 1;           // a
  Cond_signal(&c);    // b
}
```

Join Implementation: Attempt 2

Parent:

```
void thread_join() {
 Mutex_lock(&m);            // w
 if (done == 0)             // x
   Cond_wait(&c, &m);       // y
 Mutex_unlock(&m);          // z
}
```

Child:

```
void thread_exit() {
  done = 1;            // a
  Cond_signal(&c);     // b
}
```

Fixes previous broken ordering:

Parent:			w	x	~~y~~	z
Child:	a	b				

Join Implementation: Attempt 2

Parent:

```
void thread_join() {
  Mutex_lock(&m);             // w
  if (done == 0)              // x
    Cond_wait(&c, &m);        // y
  Mutex_unlock(&m);           // z
}
```

Child:

```
void thread_exit() {
    done = 1;             // a
    Cond_signal(&c);      // b
}
```

Fixes previous broken ordering:

Can you construct ordering that does not work?

Parent:			w	x	~~y~~	z
Child:	a	b				

Join Implementation: Attempt 2 (incorrect)

Parent:

```
void thread_join() {
 Mutex_lock(&m);            // w
 if (done == 0)             // x
   Cond_wait(&c, &m);       // y
 Mutex_unlock(&m);          // z
}
```

Child:

```
void thread_exit() {
  done = 1;            // a
  Cond_signal(&c);     // b
}
```

… sleep forever …

Parent:	w	x			y
Child:			a	b	

Join Implementation: Correct

Parent:

```
void thread_join() {
 Mutex_lock(&m);              // w
 if (done == 0)               // x
   Cond_wait(&c, &m);         // y
 Mutex_unlock(&m);            // z
}
```

Child:

```
void thread_exit() {
  Mutex_lock(&m);             // a
  done = 1;                   // b
  Cond_signal(&c);            // c
  Mutex_unlock(&m);           // d
}
```

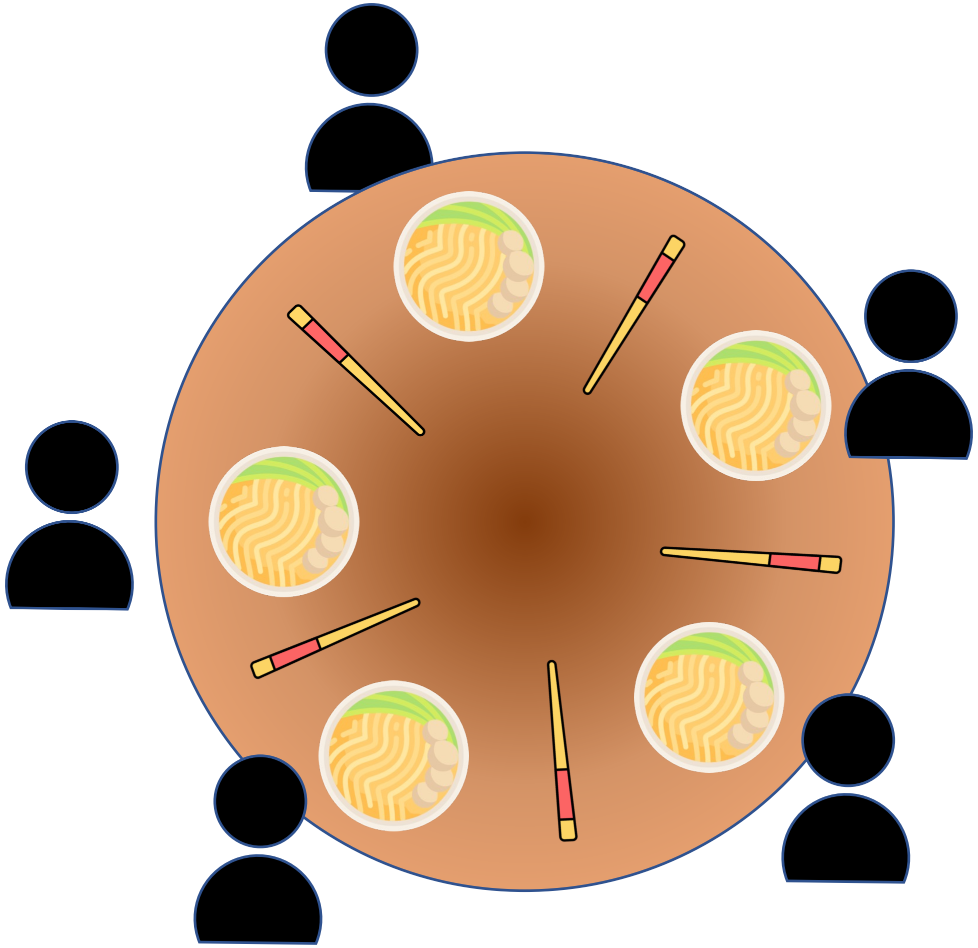
Parent: w x y z

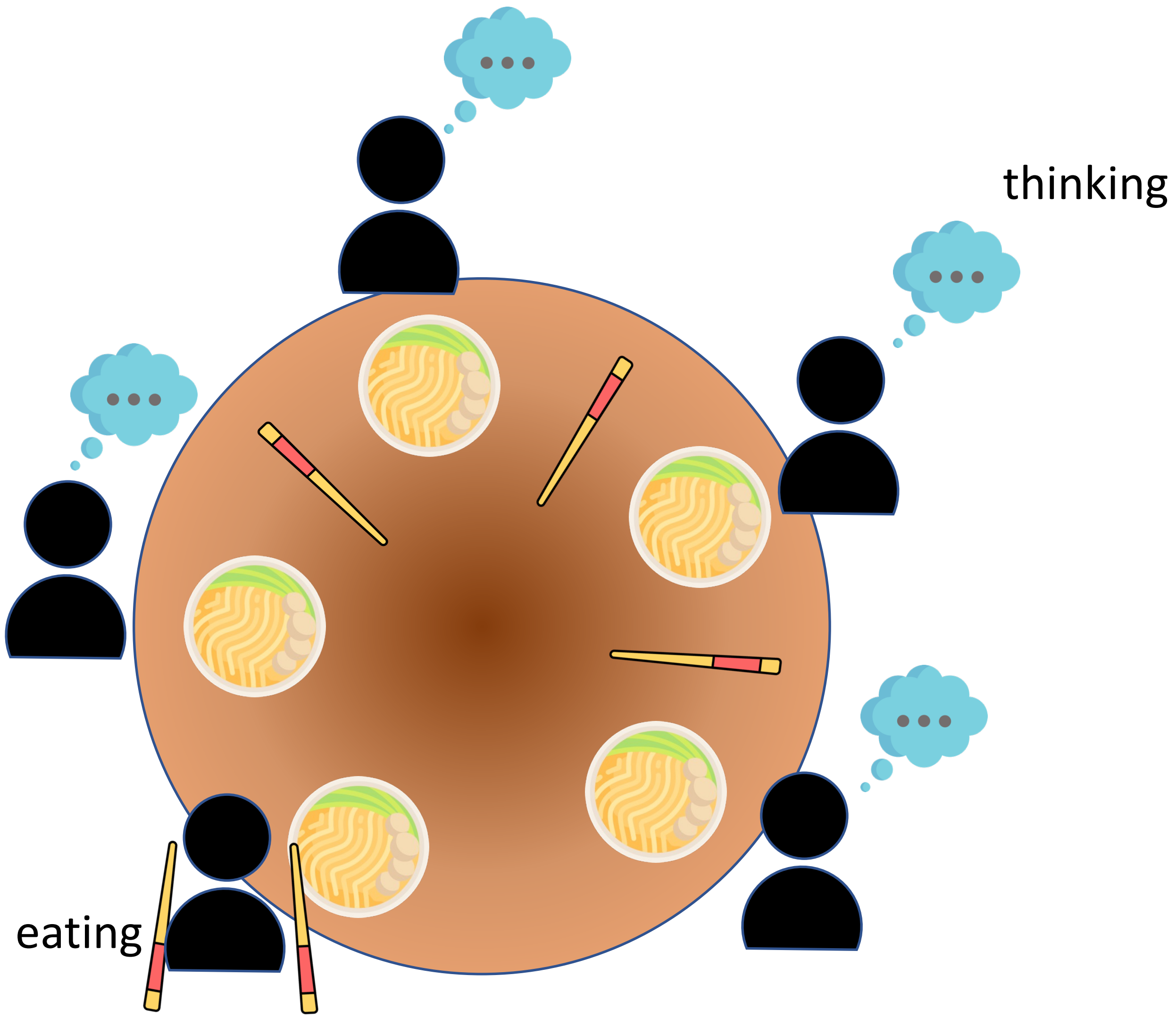
Child: a b c

Use mutex to ensure no race between interacting with state and wait/signal

Problem 2: Dining Philosophers

- Classic problem in synchronization
  - Dijkstra ’71

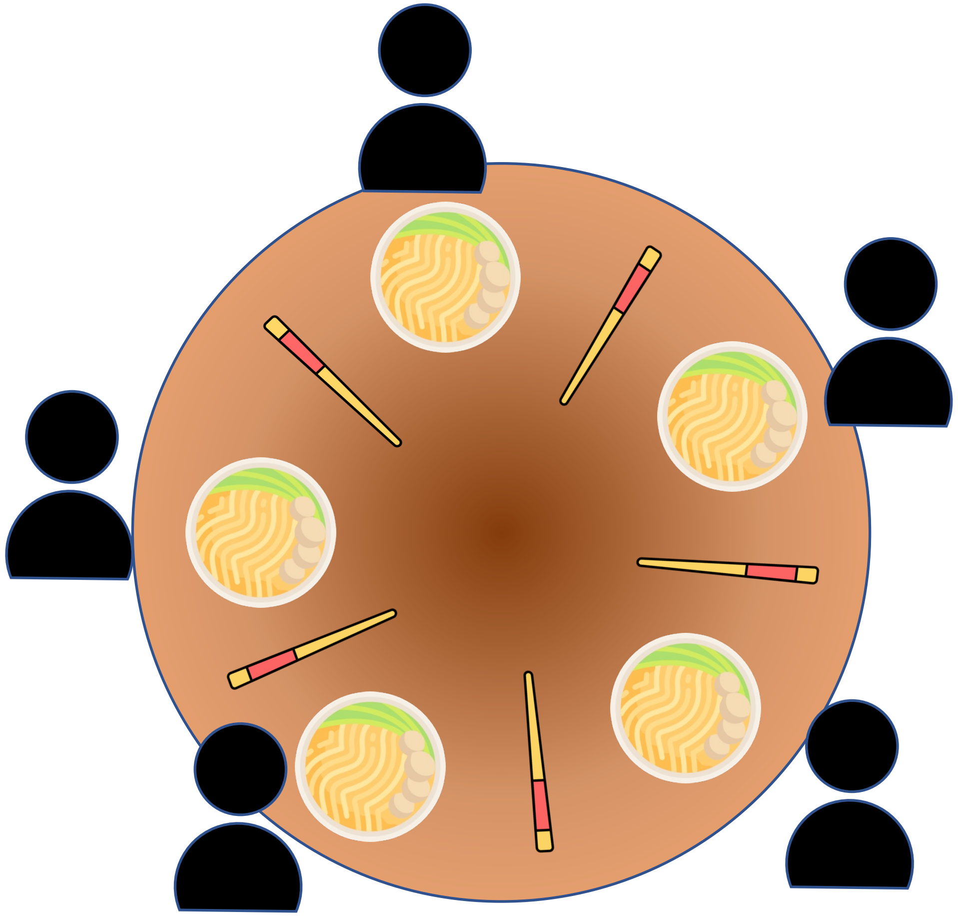




Each philosopher has 2 states:

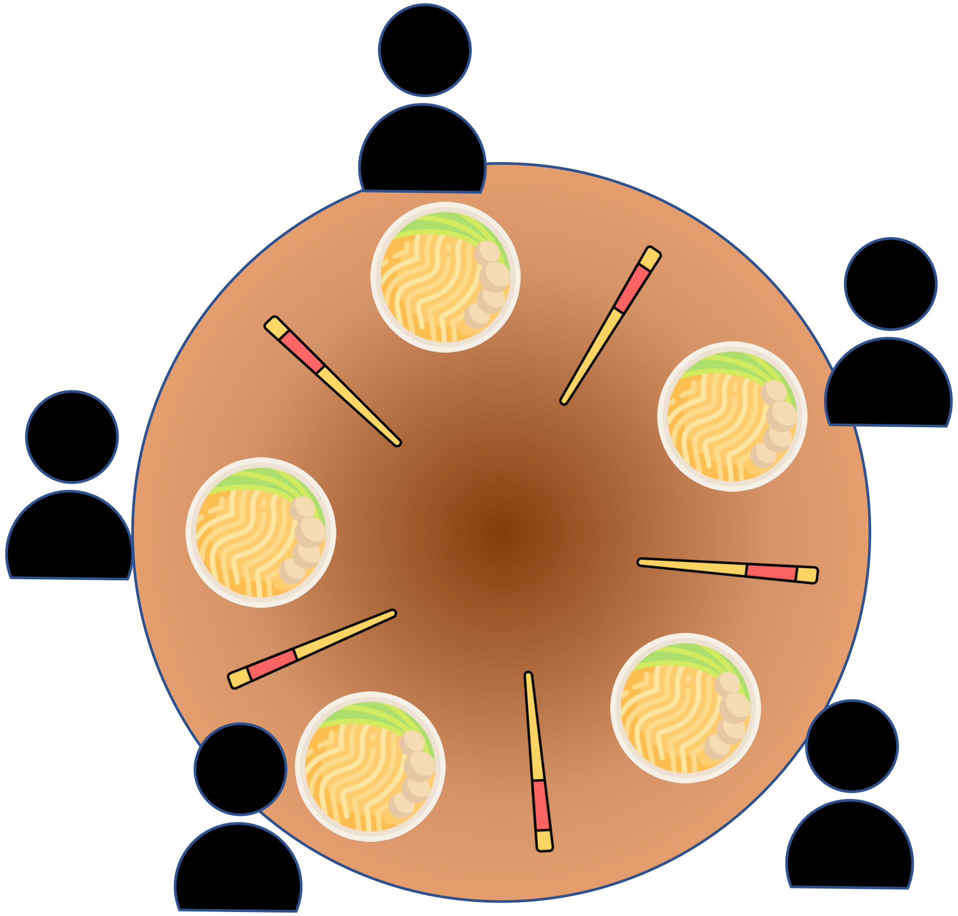
- **Eating**
  - Need 2 chopsticks to eat
- **Thinking**

When they're not eating, they're thinking (and vice-versa)



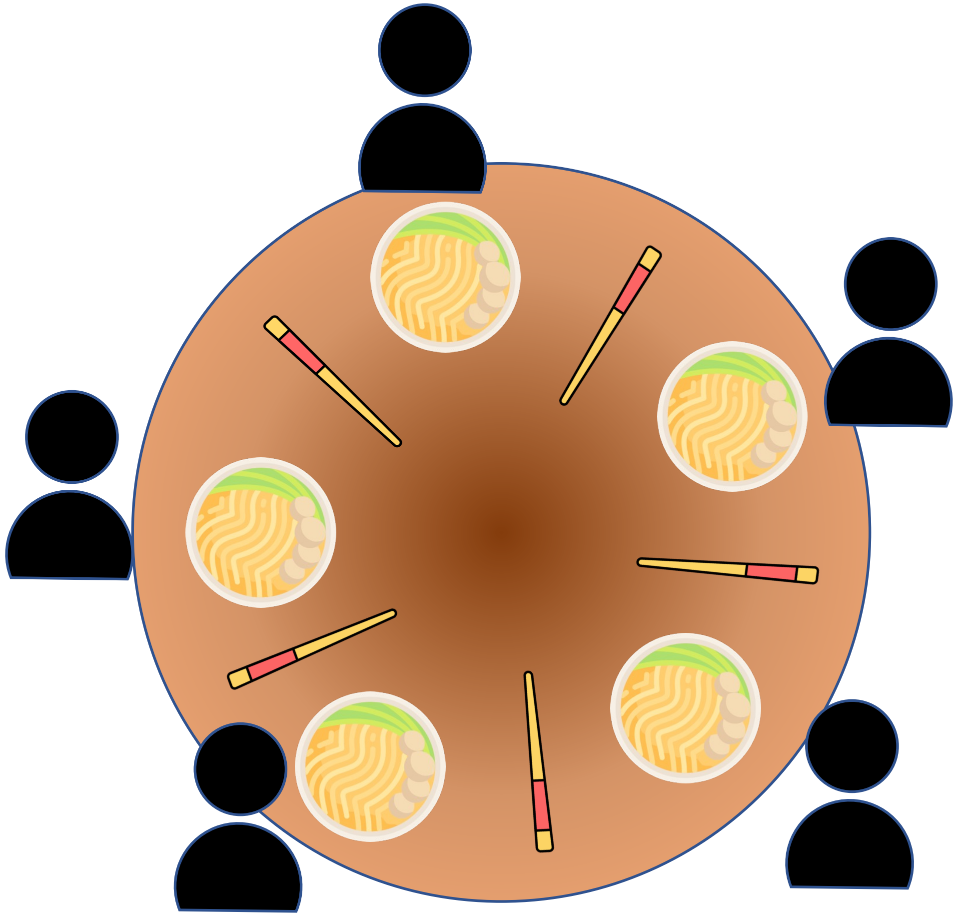
Dinner rules

- Philosophers can't speak to each other.
- Philosophers can only pick up one chopstick at a time.
- Philosophers can only pick up adjacent chopsticks.
- Infinite food supply.



Our task for today

Design a behavior such that no philosopher will starve, i.e. each can forever continue to alternate between eating and thinking.



Each philosopher has 2 states:

- **Eating**
  - Need 2 chopsticks to eat
- **Thinking**

When they're not eating, they're thinking (and vice-versa)

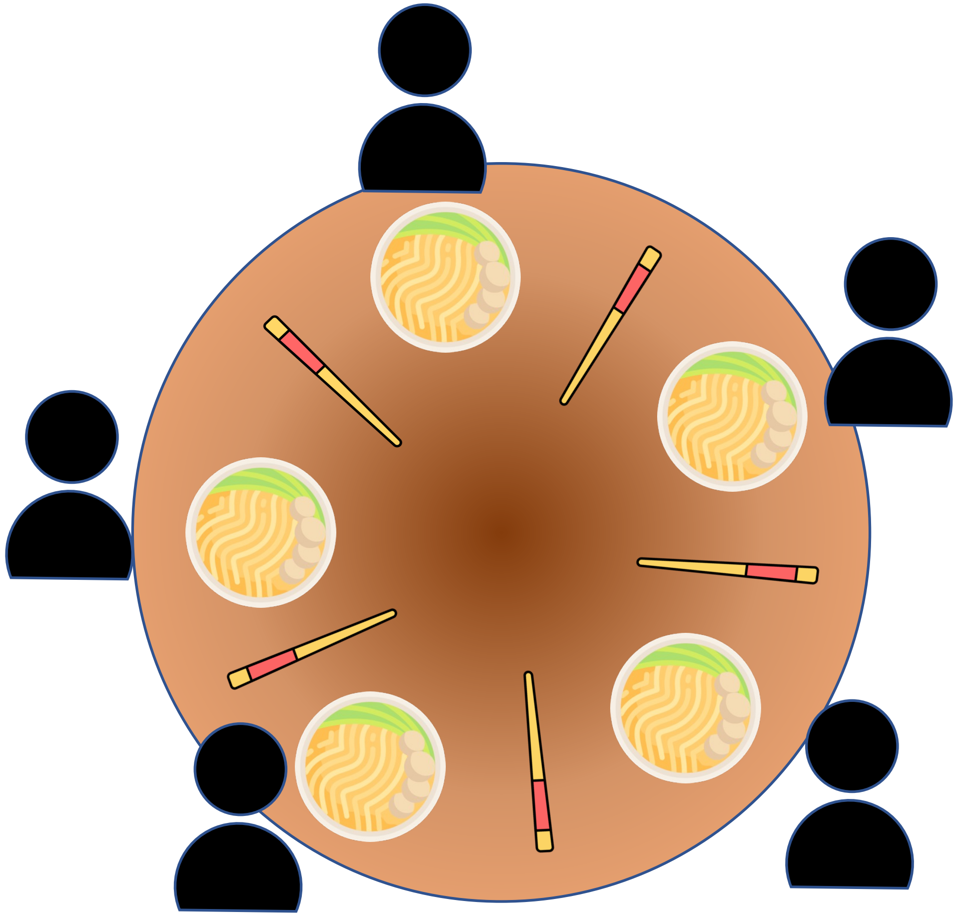
Dinner rules

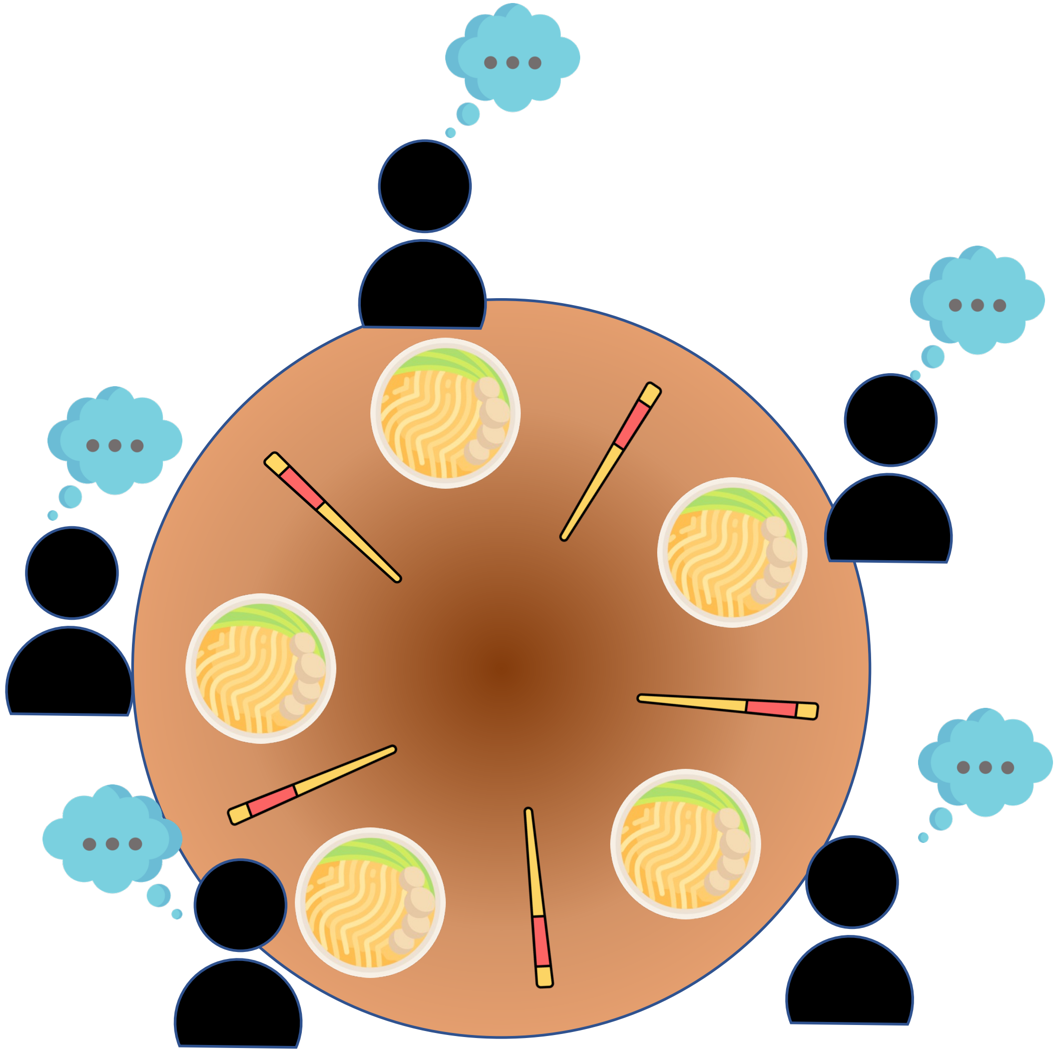
- Philosophers can't speak to each other.
- Philosophers can only pick up one chopstick at a time.
- Philosophers can only pick up adjacent chopsticks.
- Infinite food supply.

Our task

Design a behavior (i.e., an algorithm) such that no philosopher will starve, i.e. each can forever continue to alternate between eating and thinking.

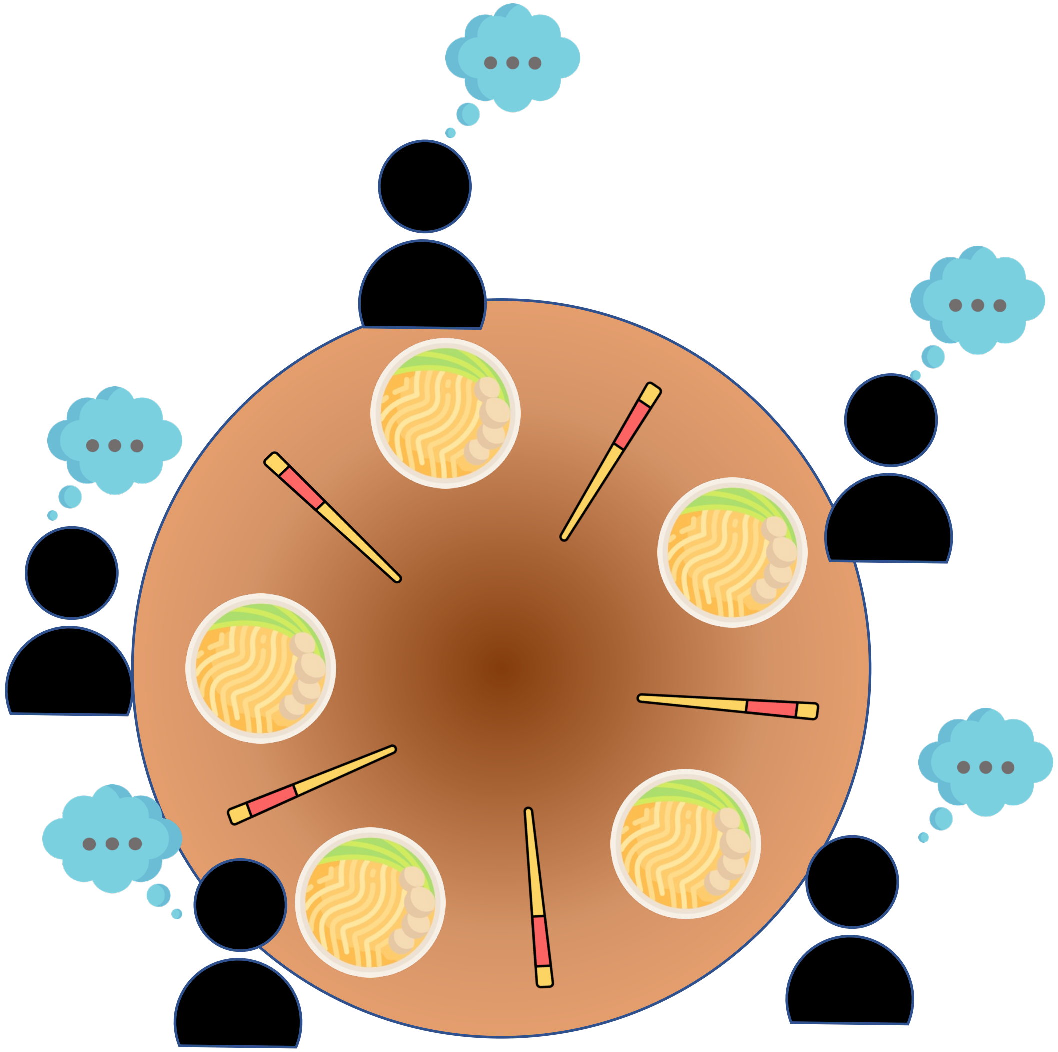
Brainstorming Slide





A simple solution (incorrect)

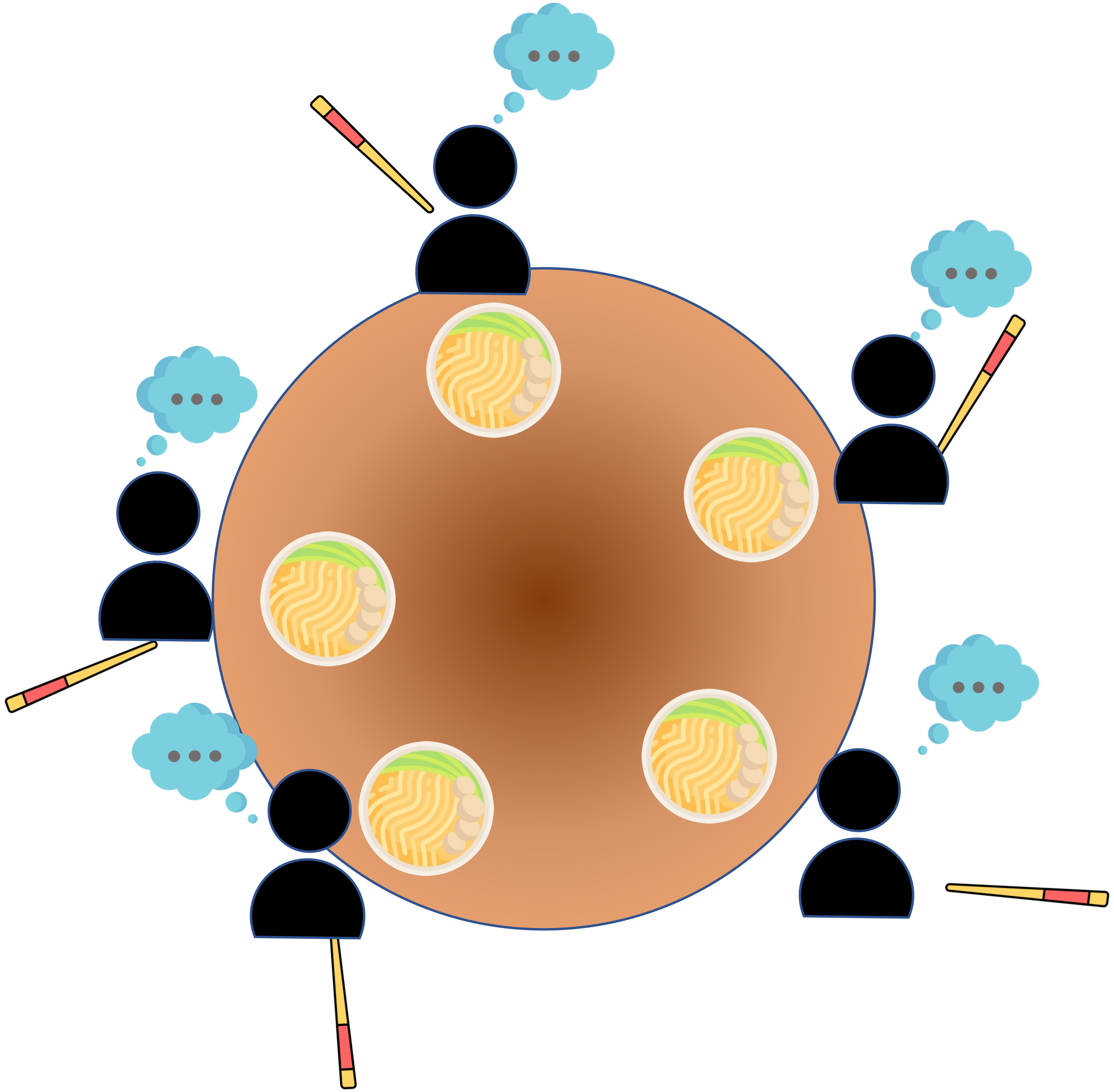
```
do forever{
 think()
 wait(chopstick_R)
 grab(chopstick_R)
 wait(chopstick_L)
 grab(chopstick_L)
 eat()
 releaseChopsticks()
}
```



A simple solution (incorrect)

```
do forever{
 think()
 wait(chopstick_R)
 grab(chopstick_R)
 wait(chopstick_L)
 grab(chopstick_L)
 eat()
 releaseChopsticks()
}
```

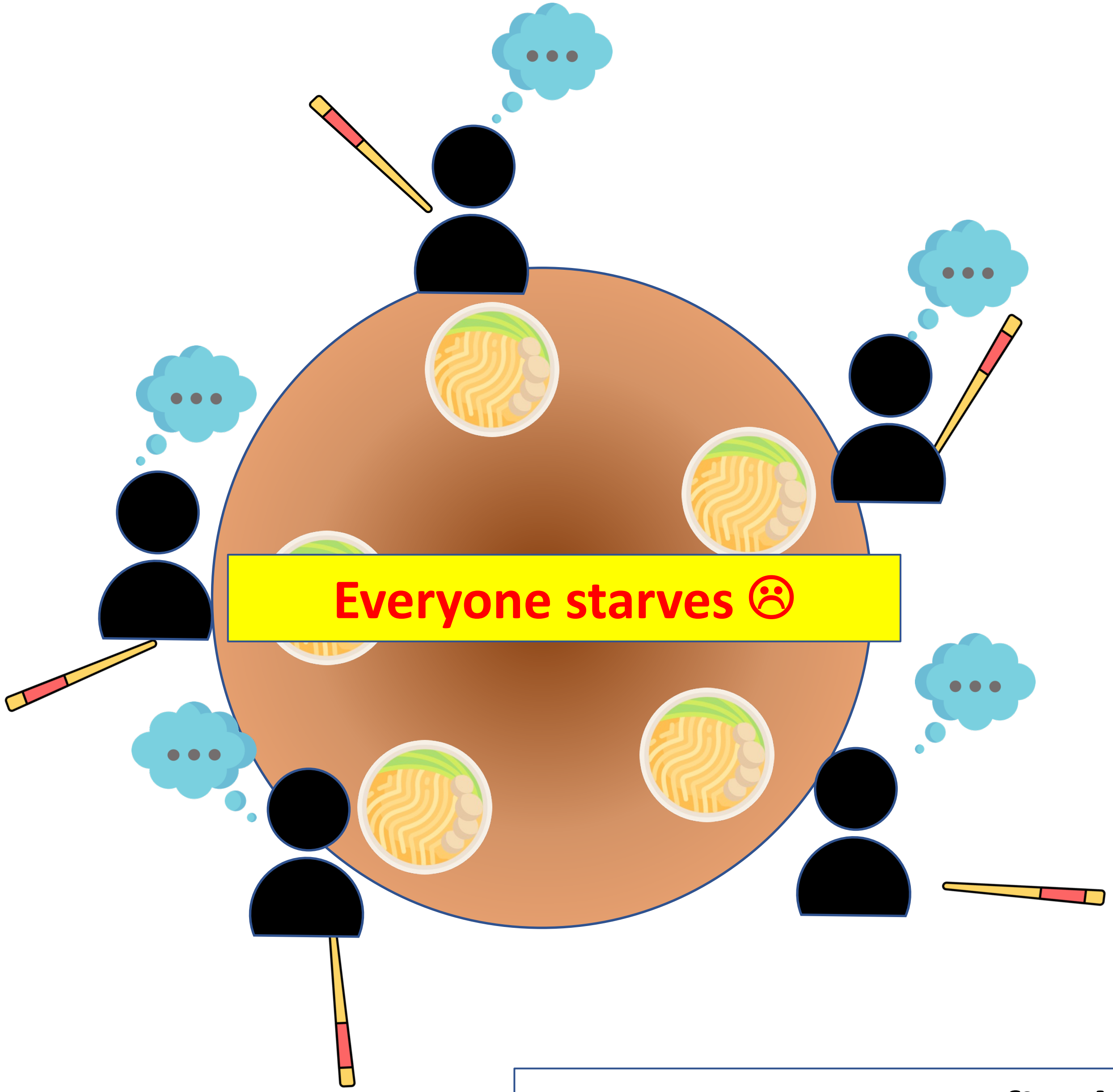
Deadlock Danger!



A simple solution (incorrect)

```
do forever{
 think()
 wait(chopstick_R)
 grab(chopstick_R)
 wait(chopstick_L)
 grab(chopstick_L)
 eat()
 releaseChopsticks()
}
```

Deadlock Danger!

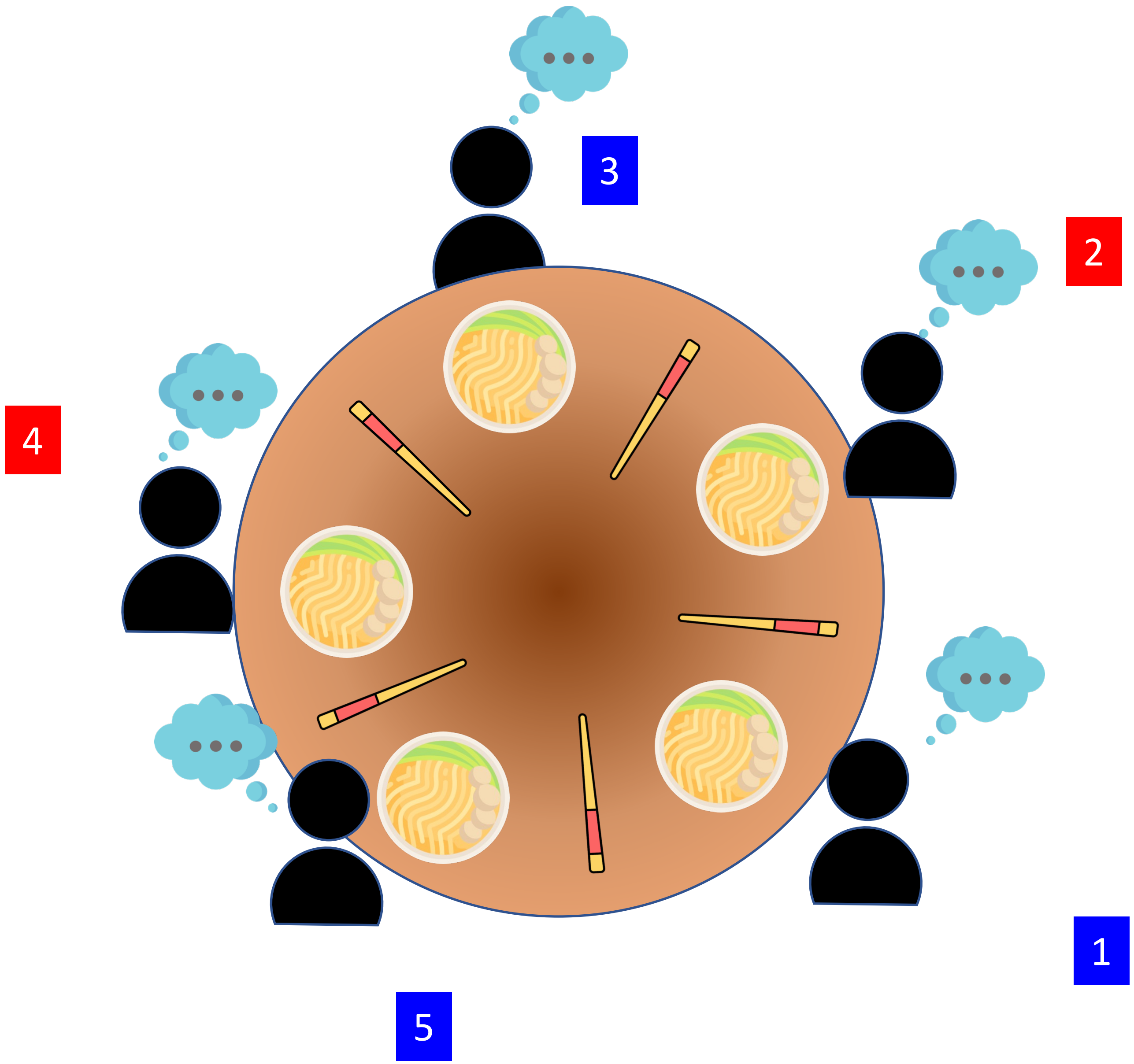


A simple solution (incorrect)

```
do forever{
  think()
  wait(chopstick_R)
  grab(chopstick_R)
  wait(chopstick_L)
  grab(chopstick_L)
  eat()
  releaseChopsticks()
}
```

Deadlock Danger!

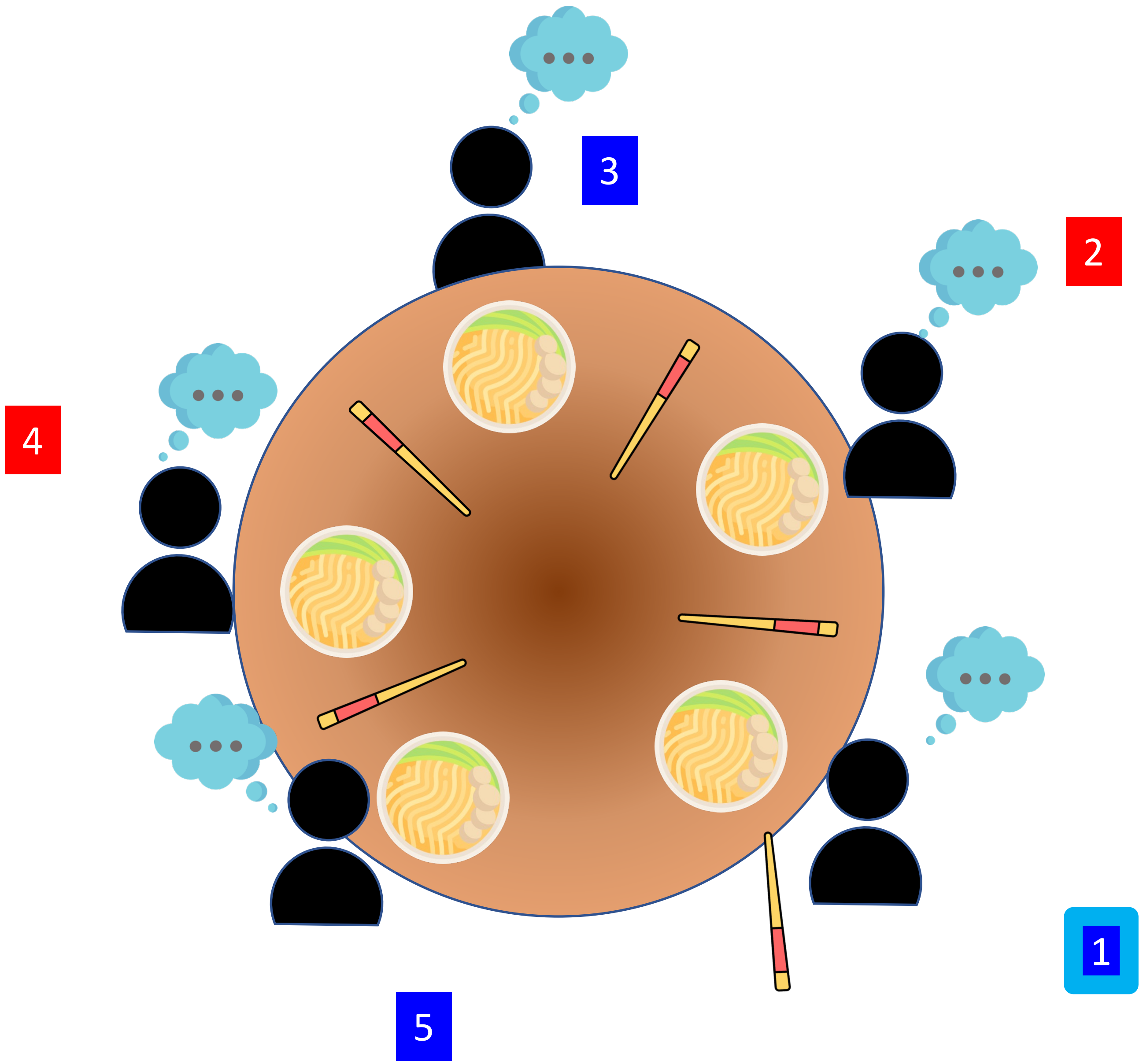
How can we fix this?



Solution 1

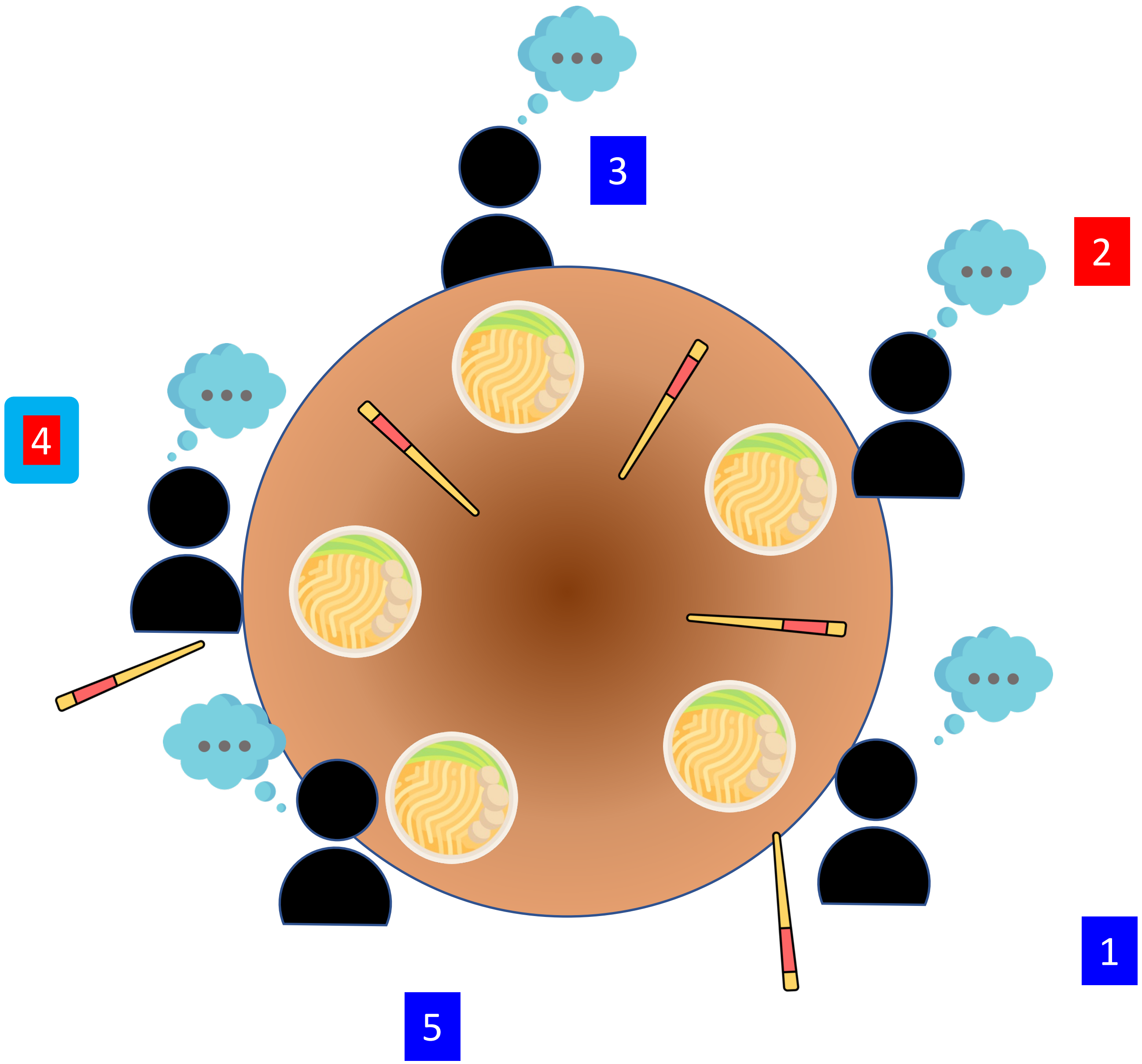
Asymmetric algorithm

- Assign odd and even IDs to philosophers.
- Odd philosophers pick up left chopstick, then right.
- Even philosophers pick up right chopstick, then left.



Solution 1
Asymmetric algorithm

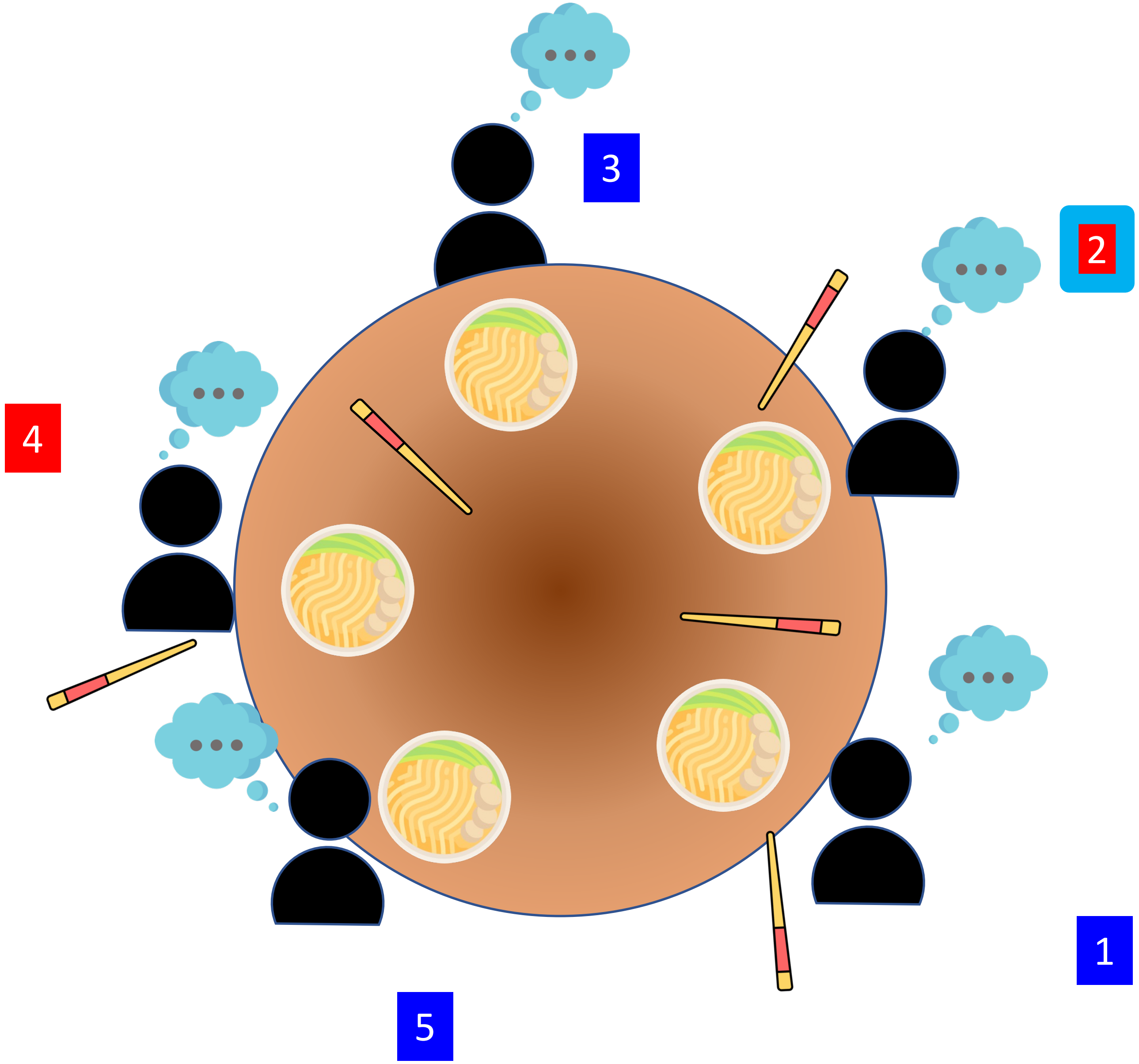
- Assign odd and even IDs to philosophers.
- Odd philosophers pick up left chopstick, then right.
- Even philosophers pick up right chopstick, then left.



Solution 1

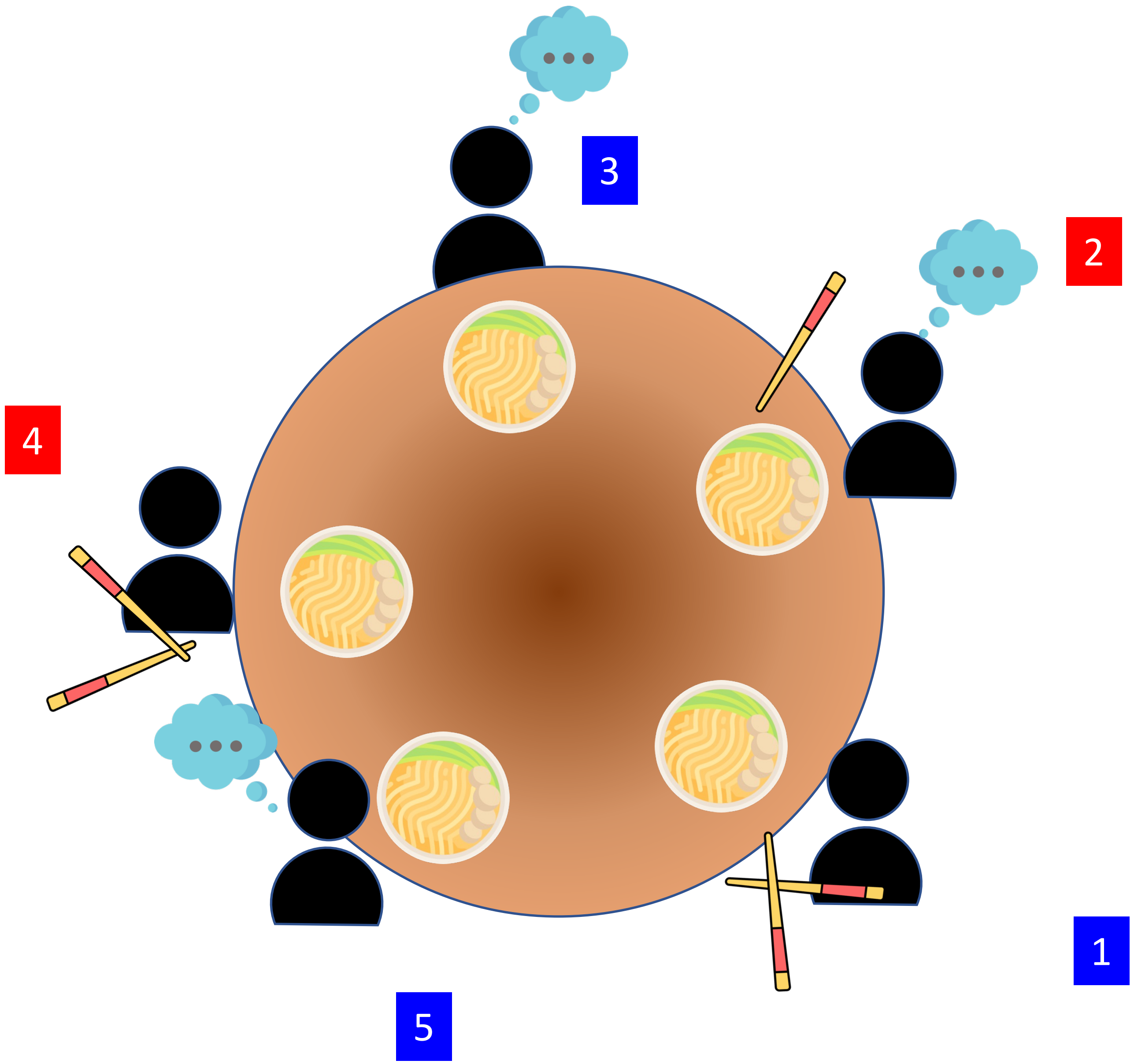
Asymmetric algorithm

- Assign odd and even IDs to philosophers.
- Odd philosophers pick up left chopstick, then right.
- Even philosophers pick up right chopstick, then left.



Solution 1
Asymmetric algorithm

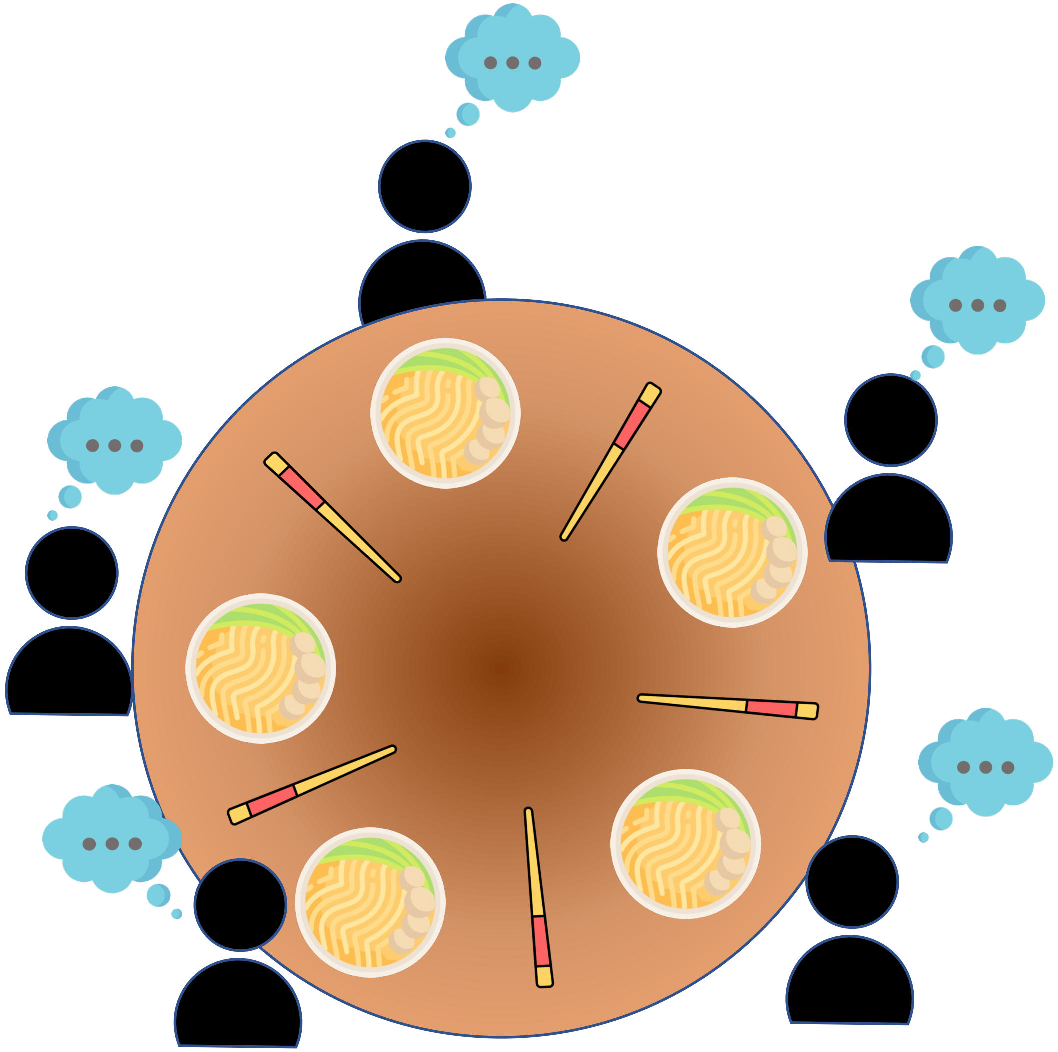
- Assign odd and even IDs to philosophers.
- Odd philosophers pick up left chopstick, then right.
- Even philosophers pick up right chopstick, then left.



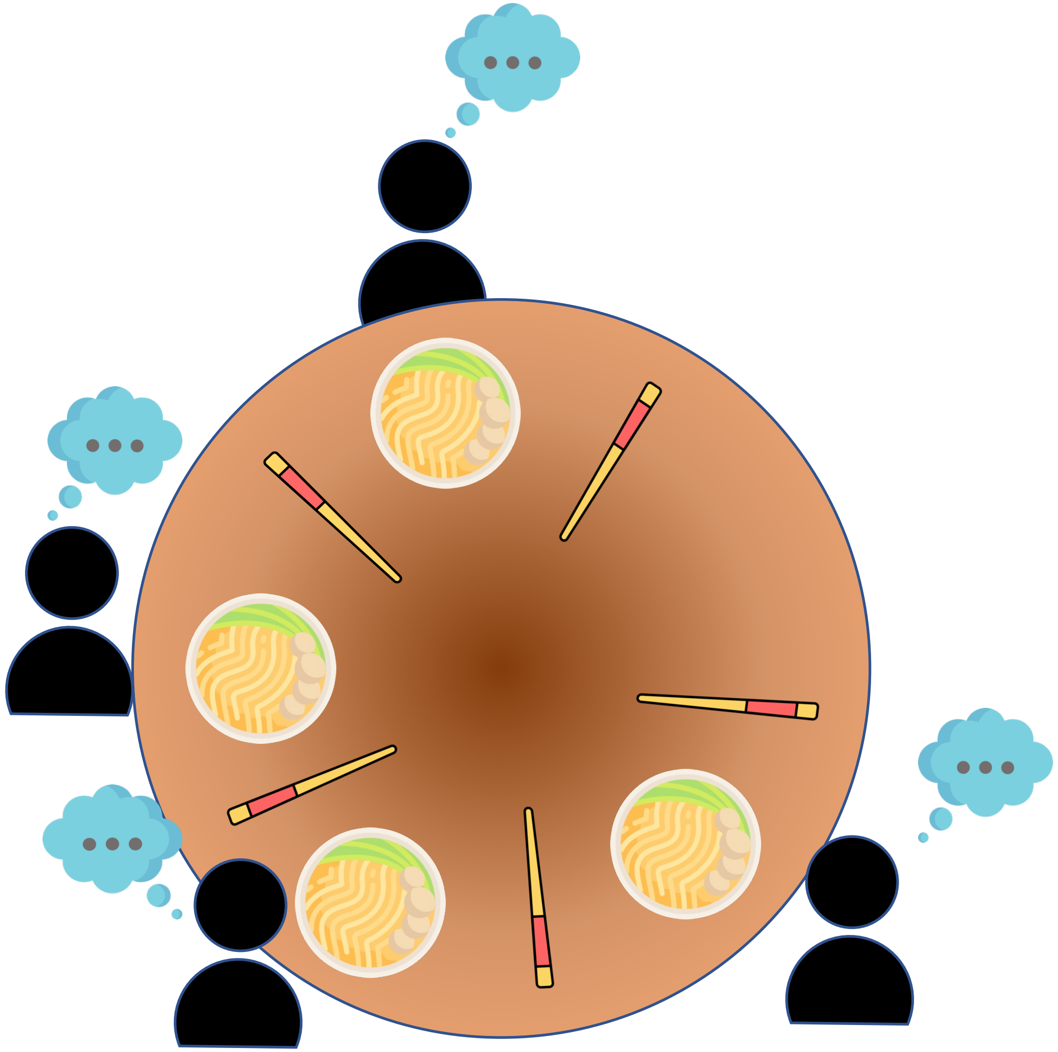
Solution 1

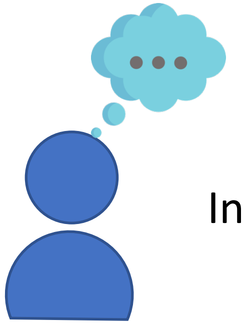
Asymmetric algorithm

- Assign odd and even IDs to philosophers.
- Odd philosophers pick up left chopstick, then right.
- Even philosophers pick up right chopstick, then left.



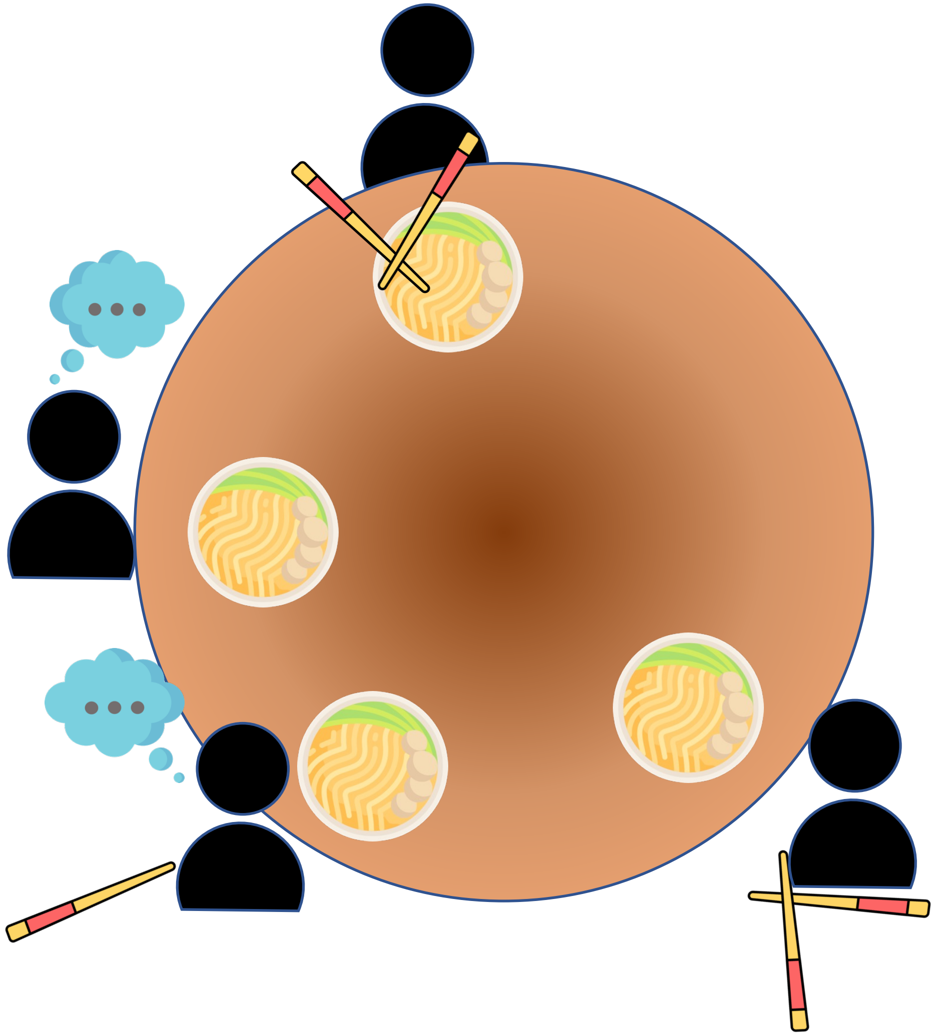
Any other approaches?

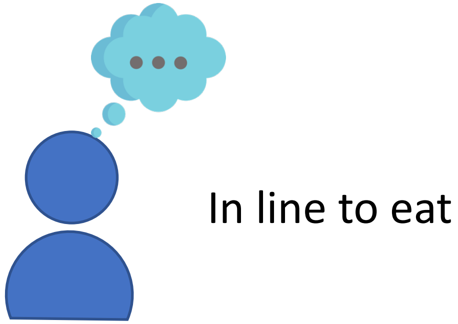




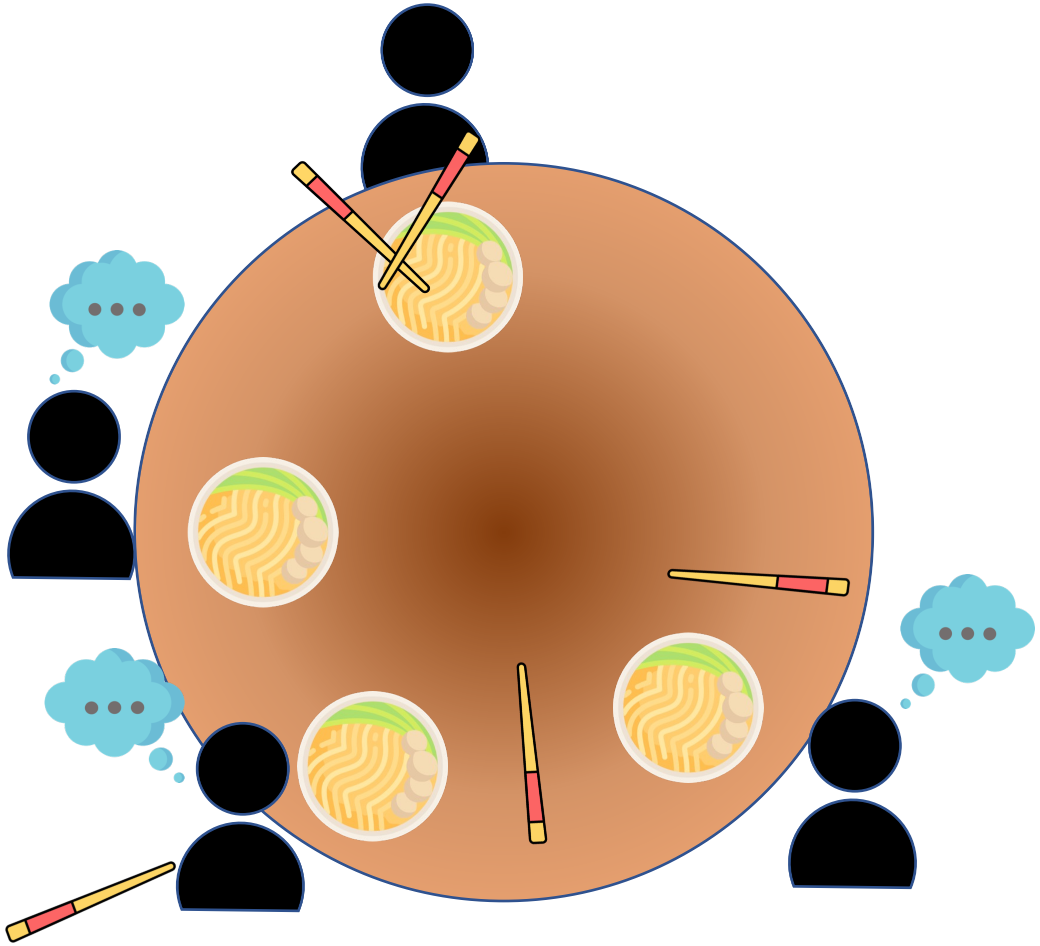
In line to eat

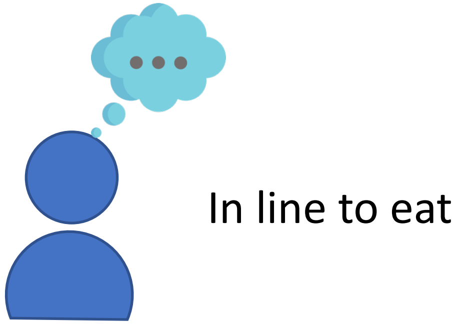
Only allow 4 philosophers
At the table



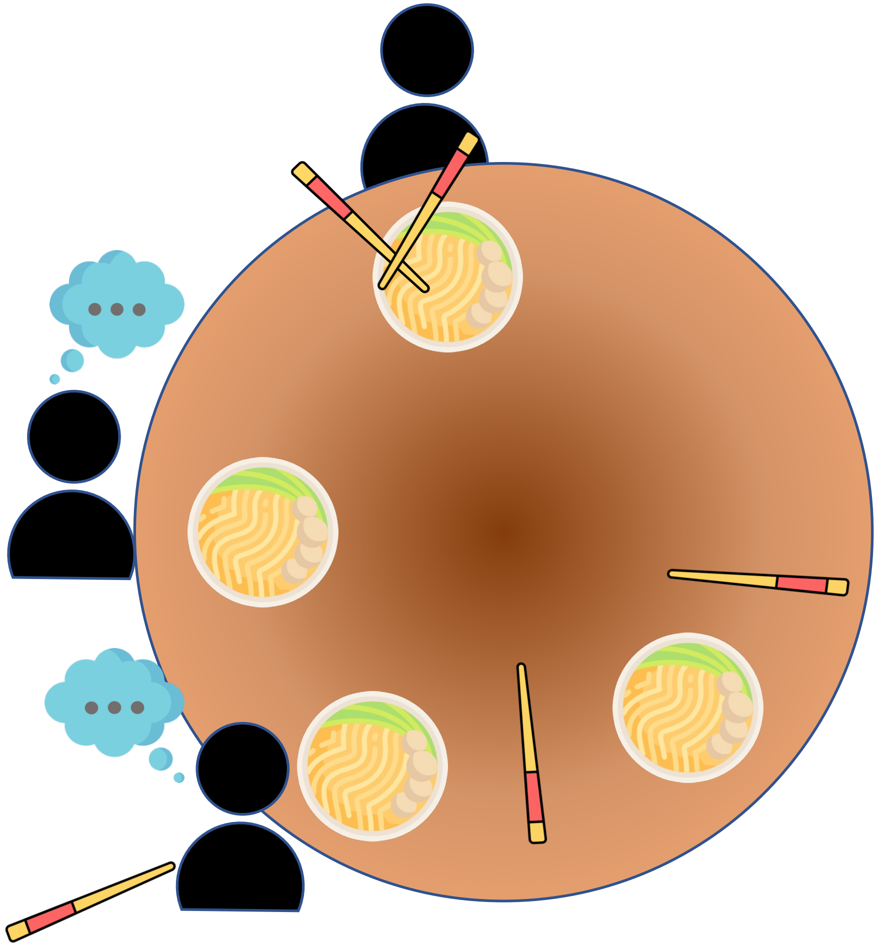


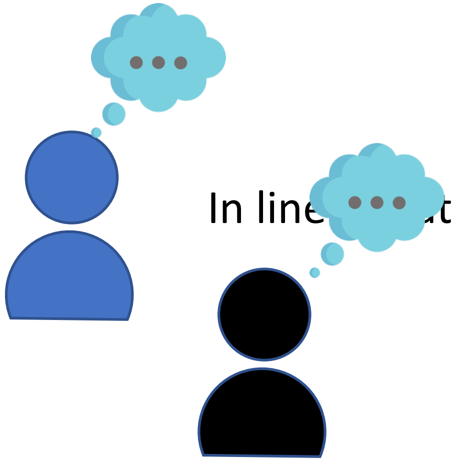
Only allow 4 philosophers
At the table



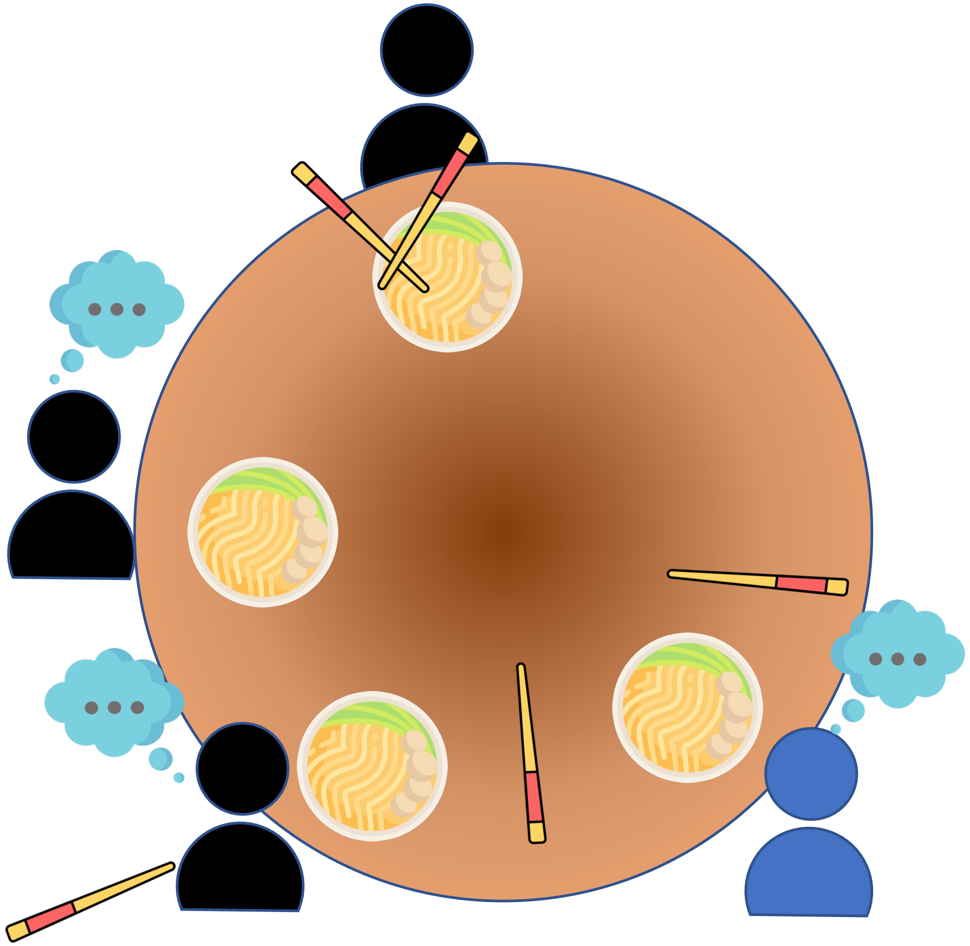


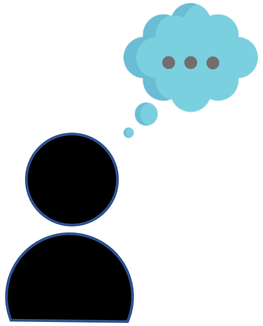
Only allow 4 philosophers
At the table





Only allow 4 philosophers
At the table





In line to eat

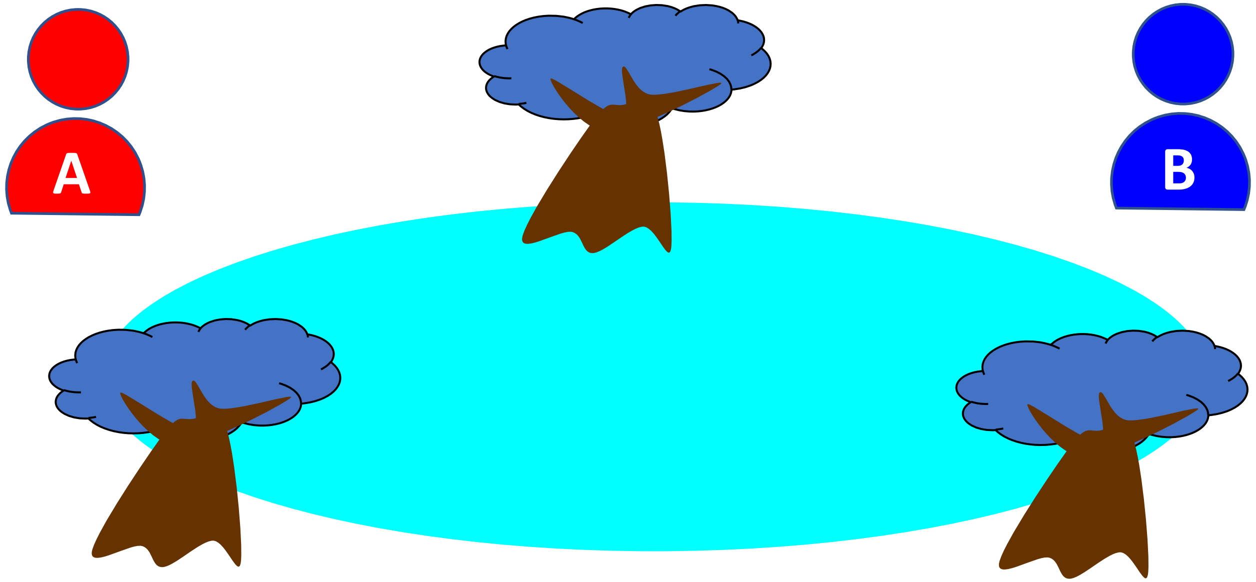
Only allow 4 philosophers
At the table

Dining philosophers – more solutions

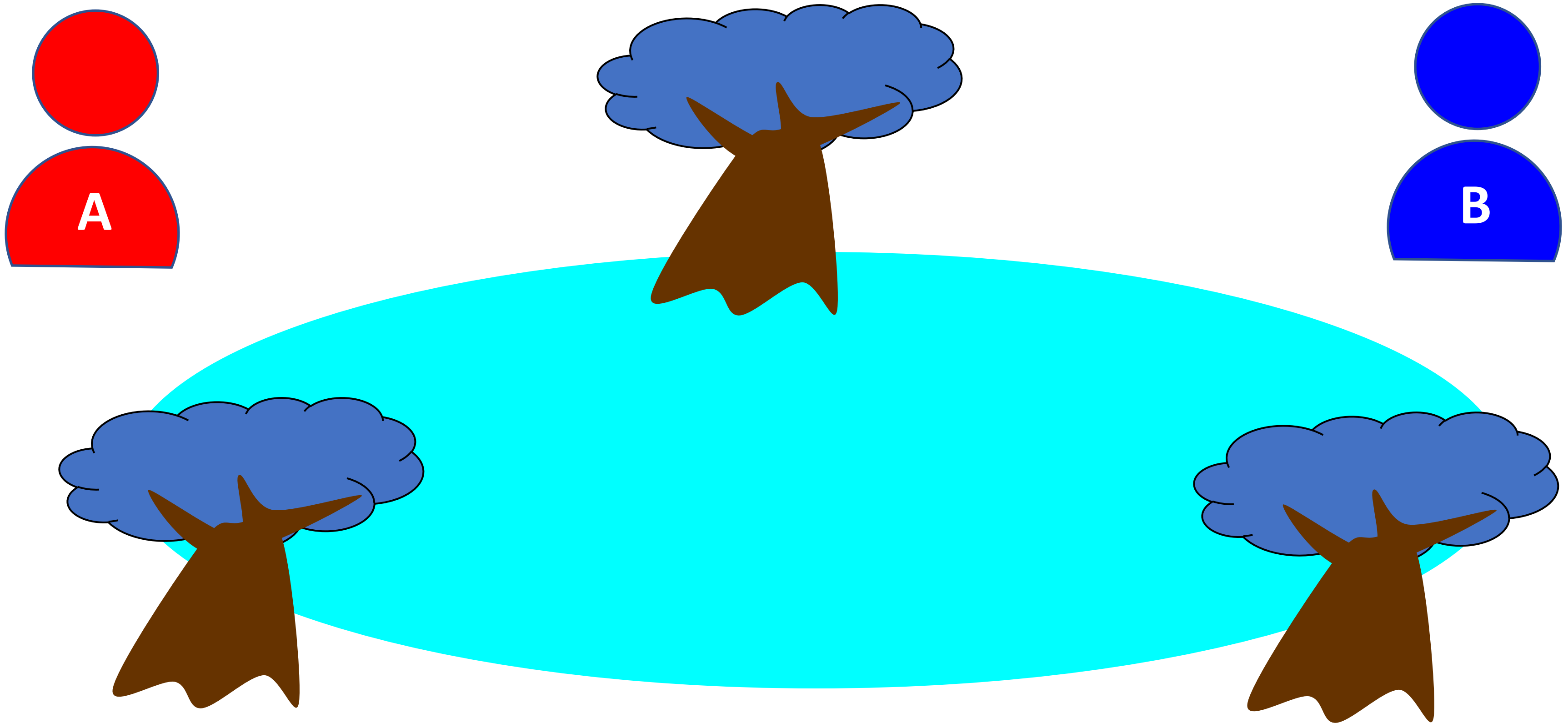
- Other solutions are possible as well:
  - Different asymmetry: make one philosopher grab the left fork first and then the right fork; all others grab the right fork first and then the left fork.
  - Use an arbiter who determines the order in which the philosophers can eat. Arbiter allows philosophers to pick up 2 chopsticks at once.
  - Use backoffs and randomness to break deadlock.

Problem 3:
Alice and Bob Share a Pond and Pet Dragons

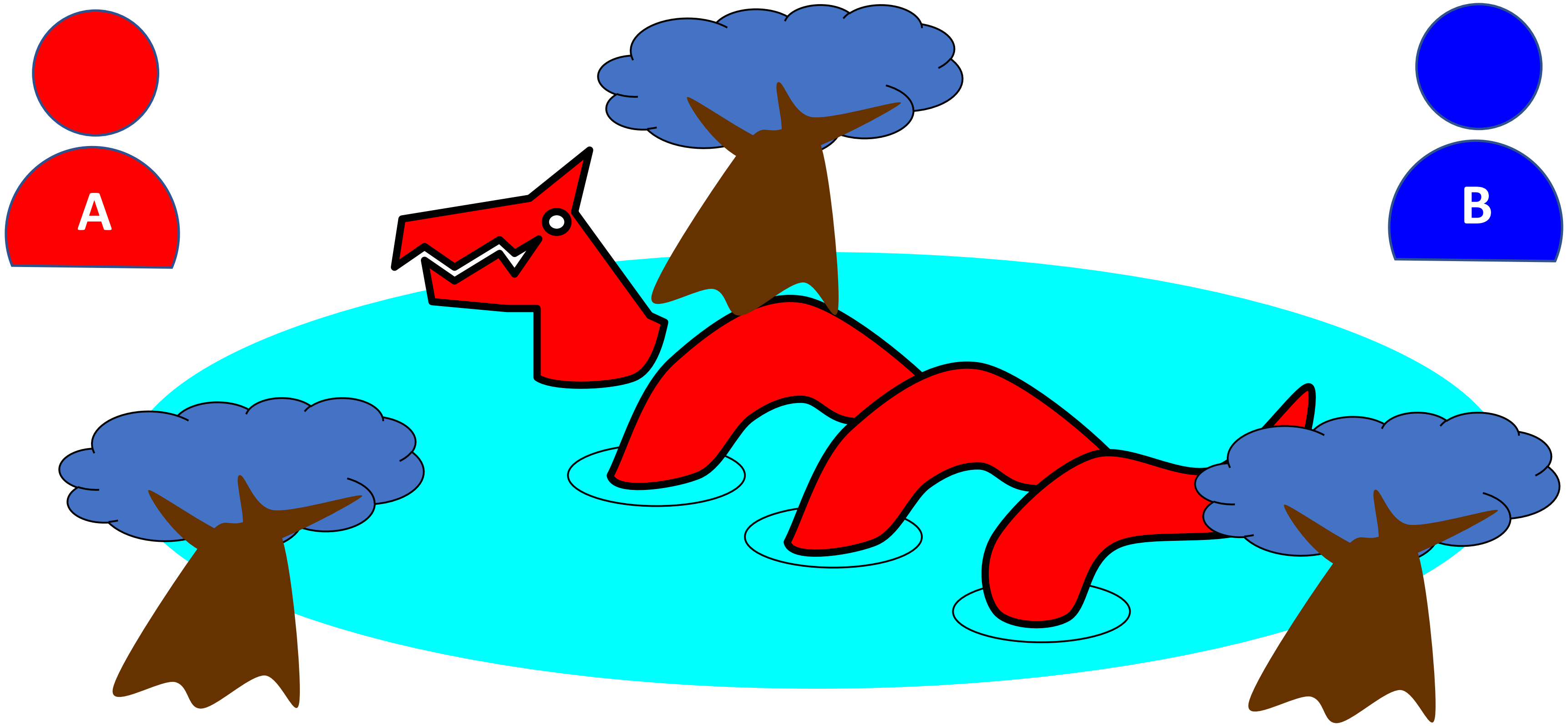
- The following story was told by a famous multiprocessing pioneer
- Leslie Lamport.



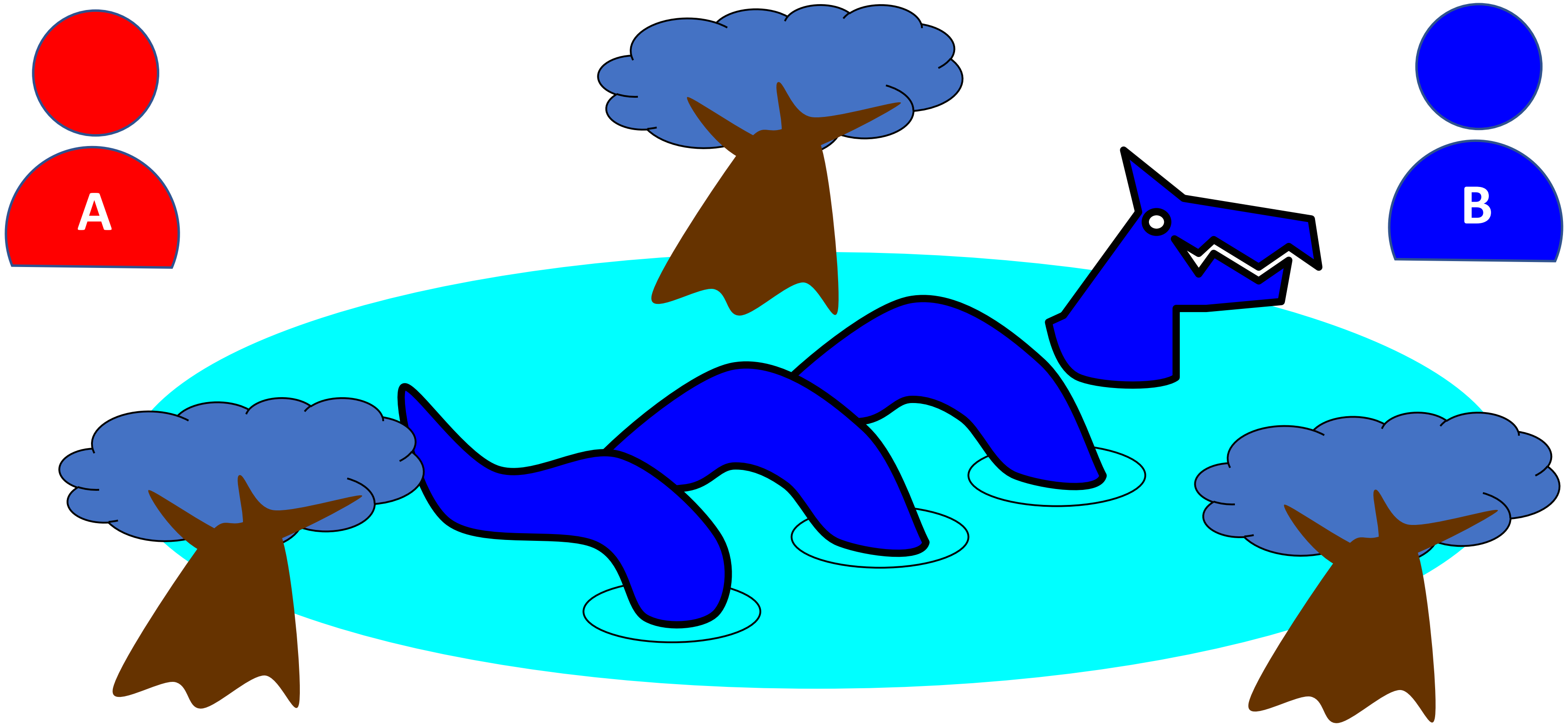
Mutual Exclusion,
or “Alice & Bob share a pond”



Alice has a pet



Bob has a pet



The Problem

Formalizing the Problem

- Mutual exclusion
  - Both pets never in pond simultaneously
- No deadlock
  - If one wants in, it gets in
  - If both want in, one gets in

Formalizing the Problem

- Mutual exclusion
  - Both pets never in pond simultaneously
  - Safety property!
- No deadlock
  - If one wants in, it gets in
  - If both want in, one gets in
  - Liveness property!

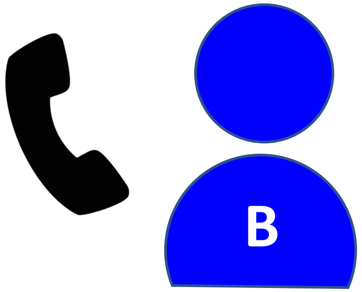
Simple Protocol

- Idea
  - Just look at the pond
- Gotcha
  - Not atomic
  - “Trees obscure the view”

Interpretation

- Threads can't "see" what other threads are doing
- Explicit communication required for coordination

Cell-phone protocol



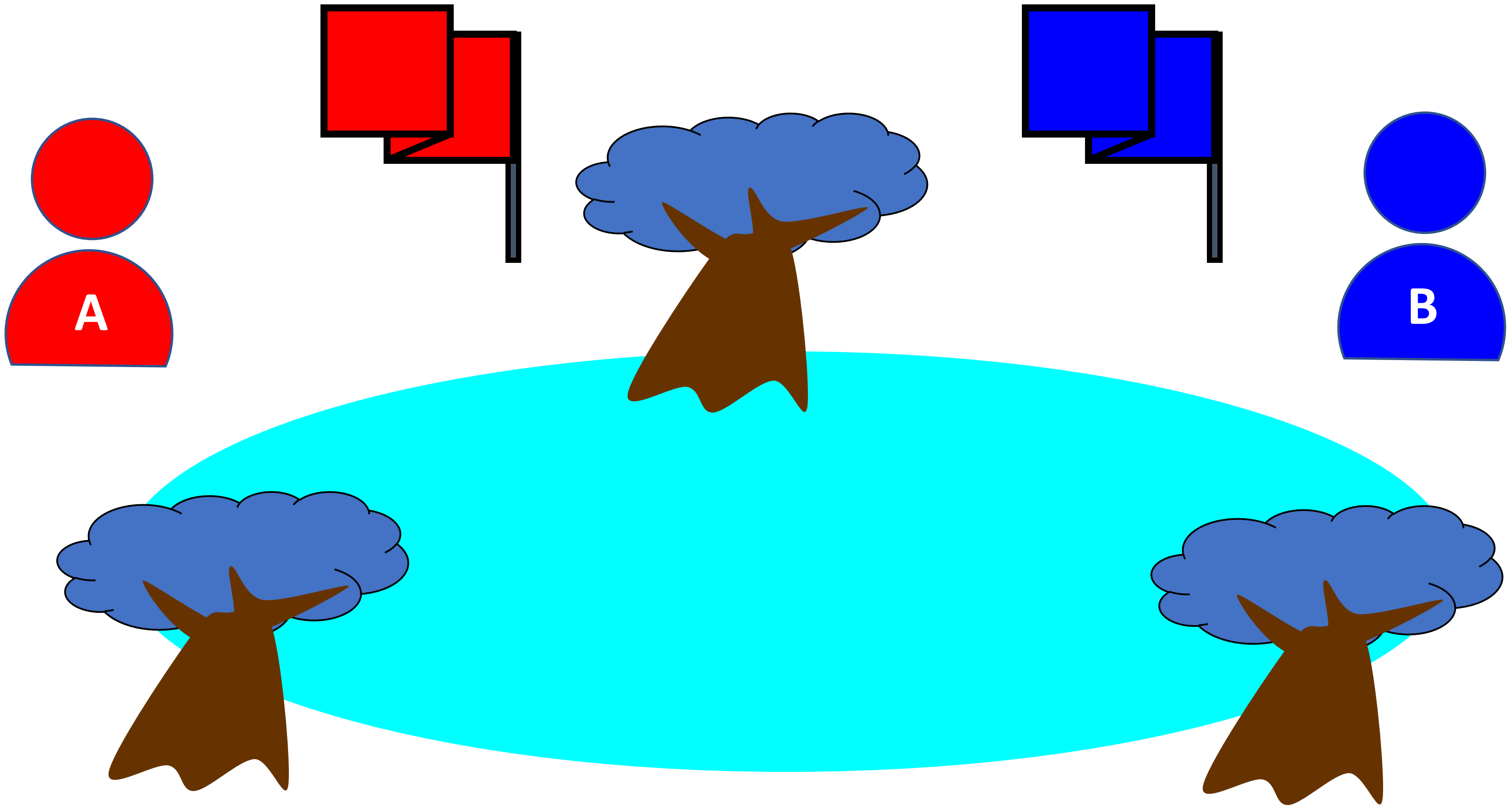
- Idea
  - Bob calls Alice (or vice-versa)
- Gotcha
  - Bob takes shower
  - Alice recharging battery
  - Bob out shopping for pet food

Interpretation

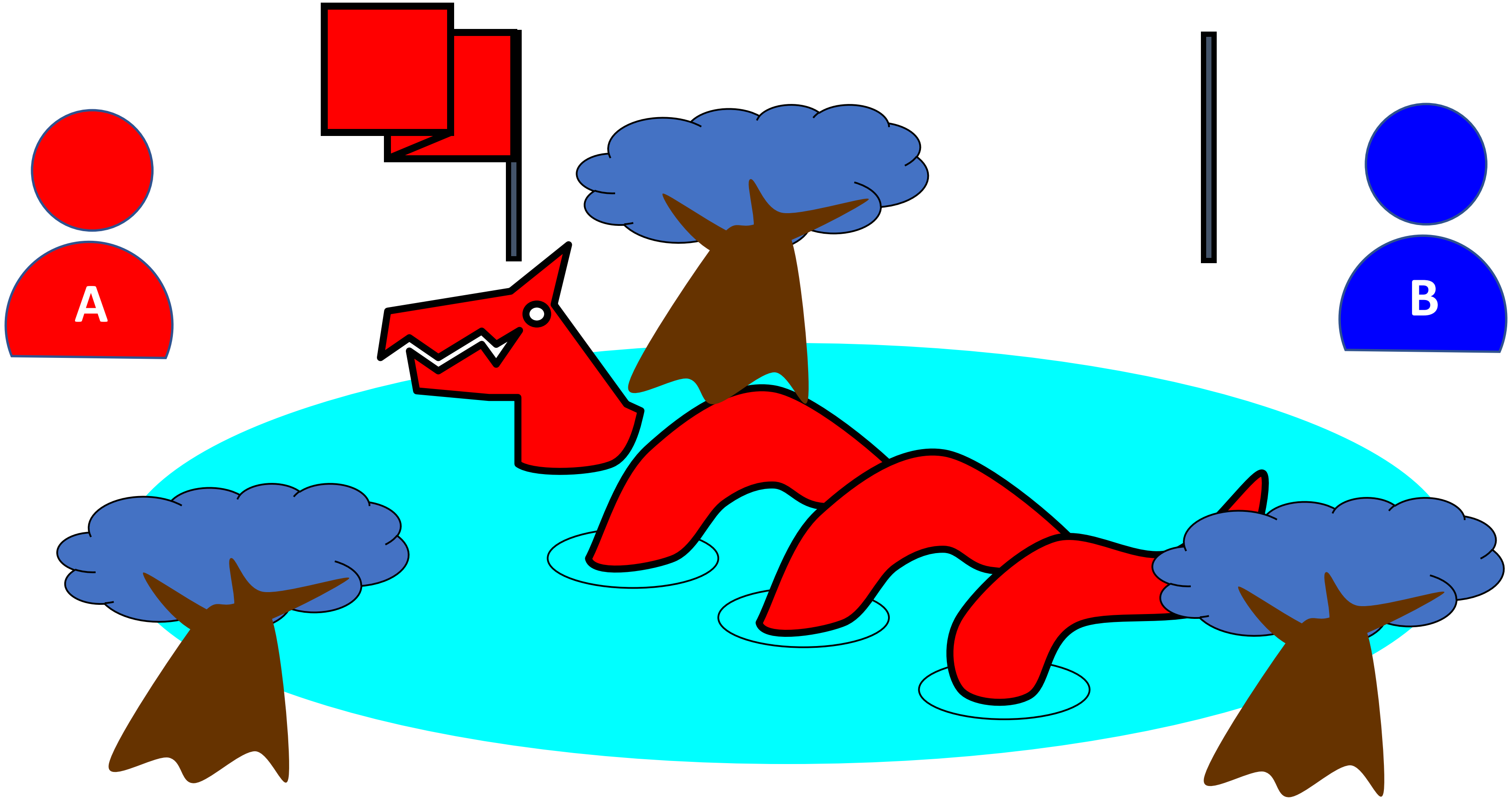
- Message-passing with shared memory doesn't work
- Recipient might not be
  - Listening
  - There at all

- Communication must be
  - Persistent (like writing)
  - Not transient (like speaking)

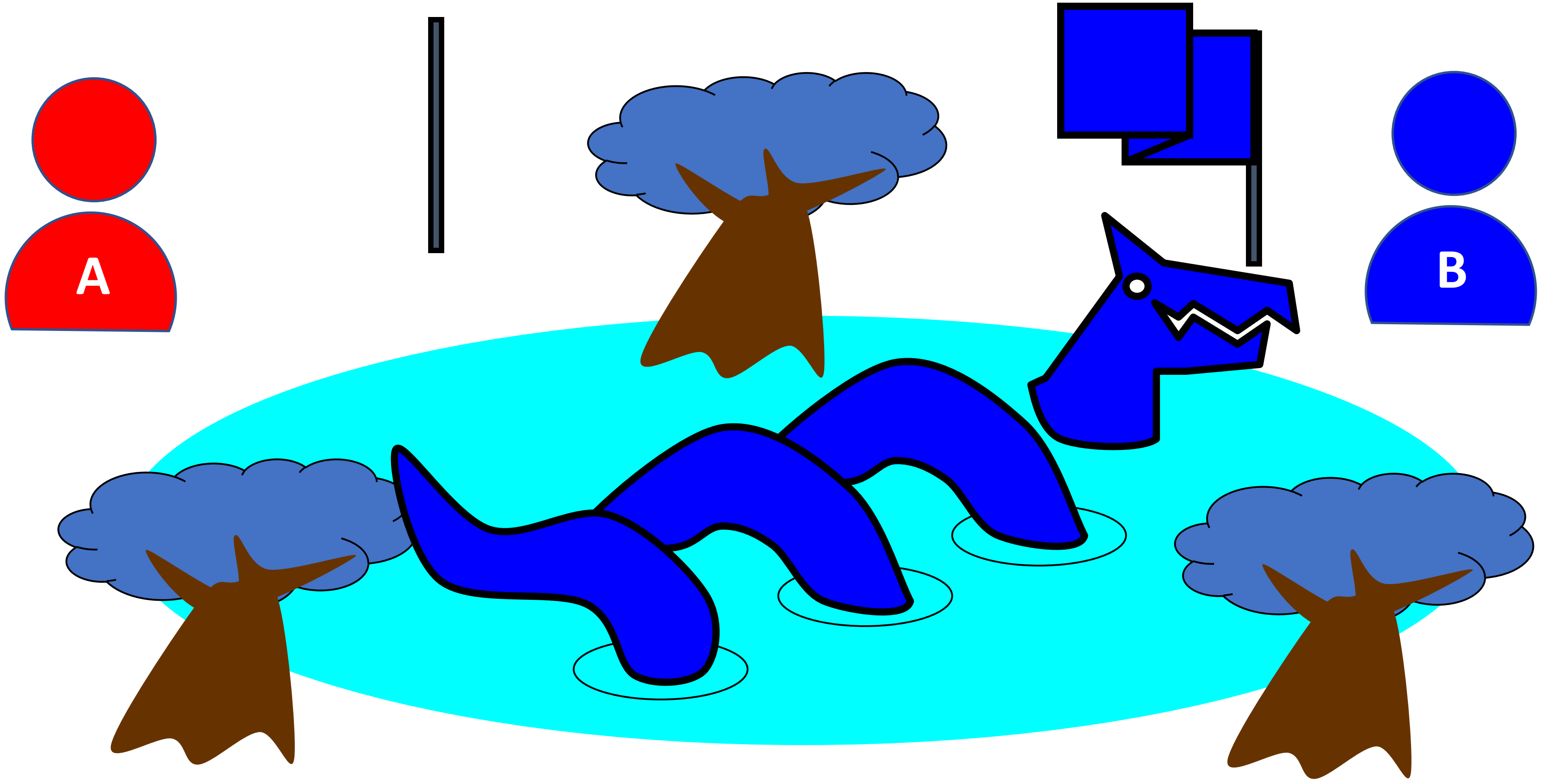
Flag Protocol



Alice’s Protocol (sort of)



Bob's Protocol (sort of)



Alice's Protocol

- Raise flag
- Wait until Bob's flag is down
- Unleash pet
- Lower flag when pet returns

Bob's protocol

- Raise flag
- Wait until Alice's flag is down
- Unleash pet
- Lower flag when pet returns

deadlock danger!

Bob's protocol (2^{nd} try)

- Raise flag
- While Alice's flag is up…
  - Lower flag
  - Wait for Alice's flag to go down
  - Raise flag
- Unleash pet
- Lower flag when pet returns

Bob’s protocol (2^{nd} try)

- Raise flag
- While Alice’s flag is up…
  - Lower flag
  - Wait for Alice’s flag to go down
  - Raise flag
- Unleash pet
- Lower flag when pet returns

Bob defers to Alice

The Flag Principle

- Raise Flag
- Look at other’s flag
- Flag principle:
  - If each raises and looks, then
  - Last to look must see both flags up

Alice-Bob-Pond: What does it mean? (1/3)

- What do all these story elements mean in OS?
- Alice & Bob = threads in the same process
- Pond = Shared memory region that needs to be accessed in a mutually exclusive way
- Pets = functions in the threads’ code that need to access the shared memory region
  - When pets are in the pond, it means that **threads are in the critical section**

Alice-Bob-Pond: What does it mean? (2/3)

- Trees = a metaphor to signify that threads cannot observe the state of a shared memory region with confidence. Why?
  - Because threads can be interrupted by the OS at any time,
  - for an indefinite amount of time.

Thread A

```
1 mem_state = read(shared_mem)
2 switch(mem_state)
3 case(x) { do X}
4 case(y) { do Y}
  …
```

Alice-Bob-Pond: What does it mean? (2/3)

- Trees = a metaphor to signify that threads cannot observe the state of a shared memory region with confidence. Why?
  - Because threads can be interrupted by the OS at any time,
  - for an indefinite amount of time.

Thread A

```
1 mem_state = read(shared_mem)
2 switch(mem_state)
3 case(x) { do X}
4 case(y) { do Y}
  …
```

Scheduler can interrupt A between lines 1 and 2
Thread B can run and change shared_mem
A’s mem_state variable is stale

Alice-Bob-Pond: What does it mean? (3/3)

- Flags = Bits that threads use to coordinate access to the shared memory region.
  - In a system with N threads, generally we define an array of size N.
  - Each thread can write in one entry (i.e., write only their flag)
  - Can read all the other entries (i.e., look at all the others' flags)

Proof of Mutual Exclusion (Sketch)

- Proof by contradiction
- Assume both pets in pond somehow
  - Derive a contradiction
  - By reasoning backwards
- Consider the last time Alice and Bob each looked before letting the pets in.
- Without loss of generality, assume Alice was the last to look…

Proof of Mutual Exclusion

Thread A

```
Raise flag
Wait until B's flag is down (looking)
Release pet
Lower flag when pet returns
```

Thread B

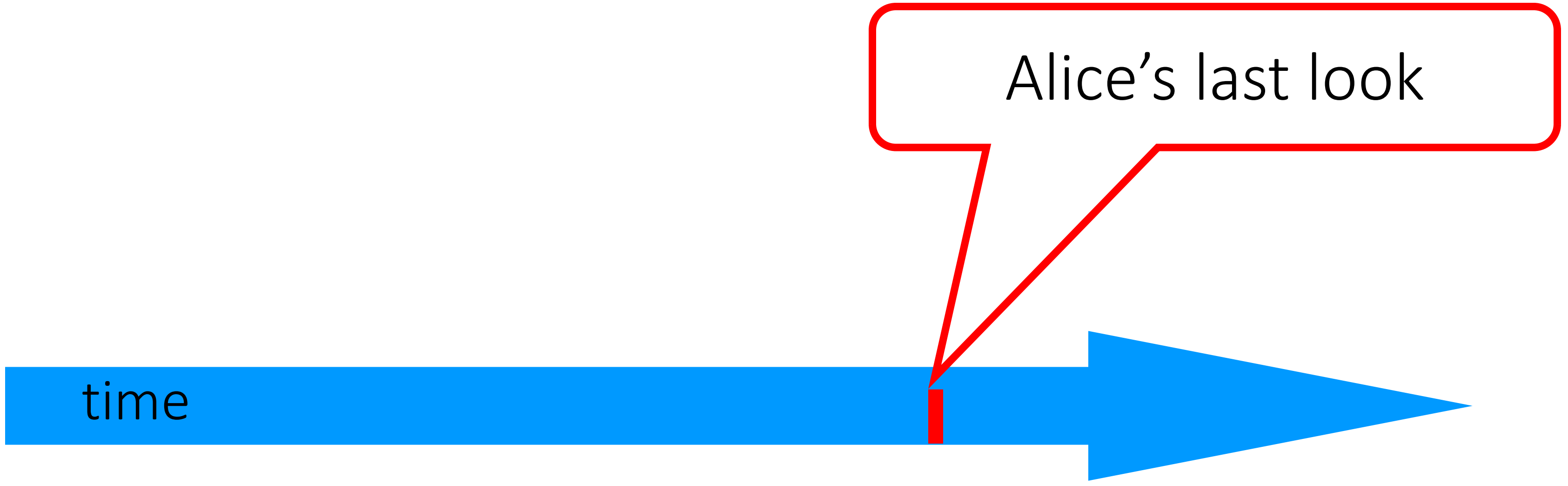
```
Raise flag
While (A's flag is up)
 Lower B flag
 Wait for A flag to go down (looking)
 Raise flag
Release pet
Lower flag when pet returns
```

Protocol: first raise flag, then look for both A and B

Proof of Mutual Exclusion

Thread A
```
Raise flag
Wait until B’s flag is down (looking)
Release pet
Lower flag when pet returns
```

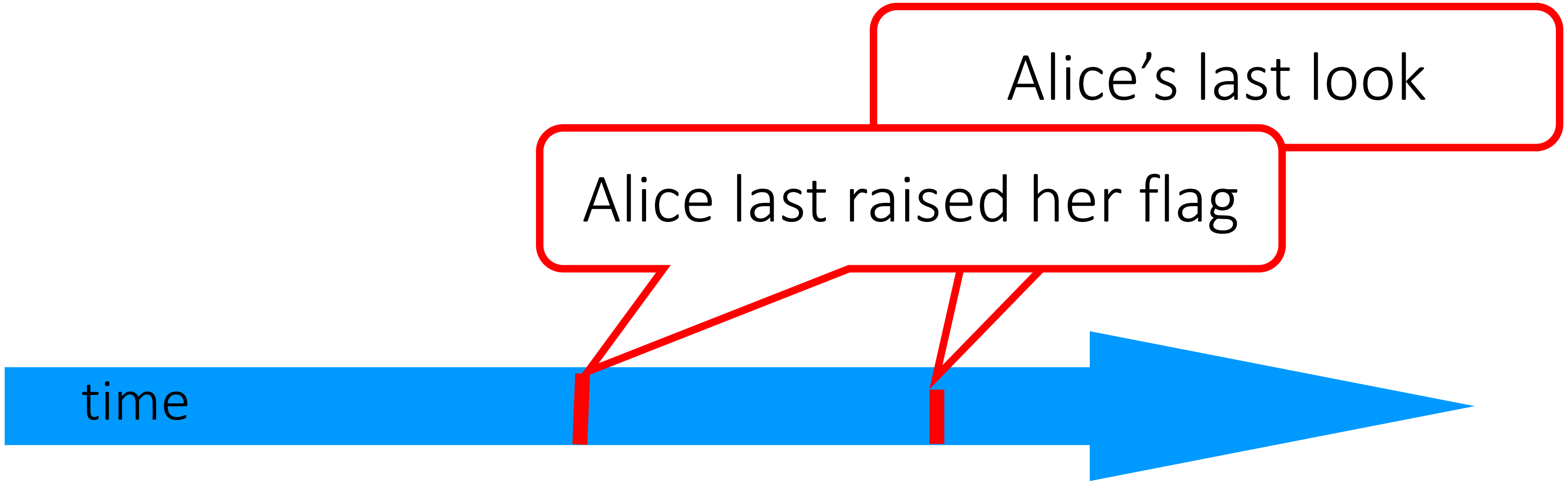
Thread B
```
Raise flag
While (A’s flag is up)
 Lower B flag
 Wait for A flag to go down (looking)
 Raise flag
Release pet
Lower flag when pet returns
```



Proof of Mutual Exclusion

```
Thread A
Raise flag
Wait until B's flag is down (looking)
Release pet
Lower flag when pet returns
```

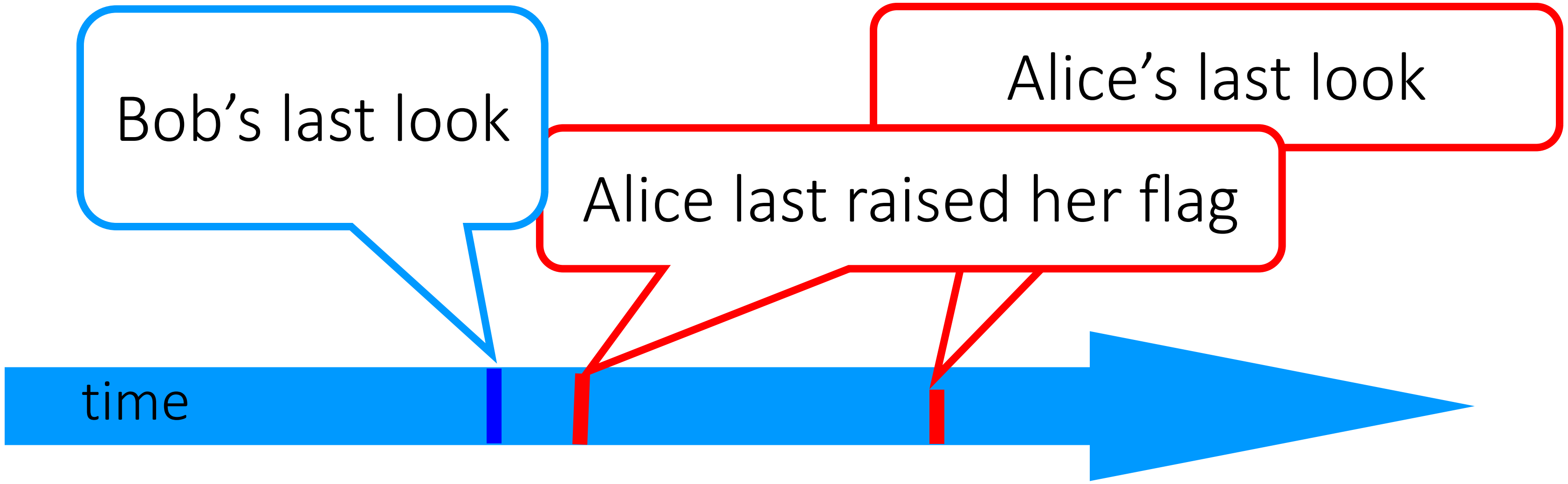
```
Thread B
Raise flag
While (A's flag is up)
 Lower B flag
 Wait for A flag to go down (looking)
 Raise flag
Release pet
Lower flag when pet returns
```



Proof of Mutual Exclusion

```
Thread A
Raise flag
Wait until B's flag is down (looking)
Release pet
Lower flag when pet returns
```

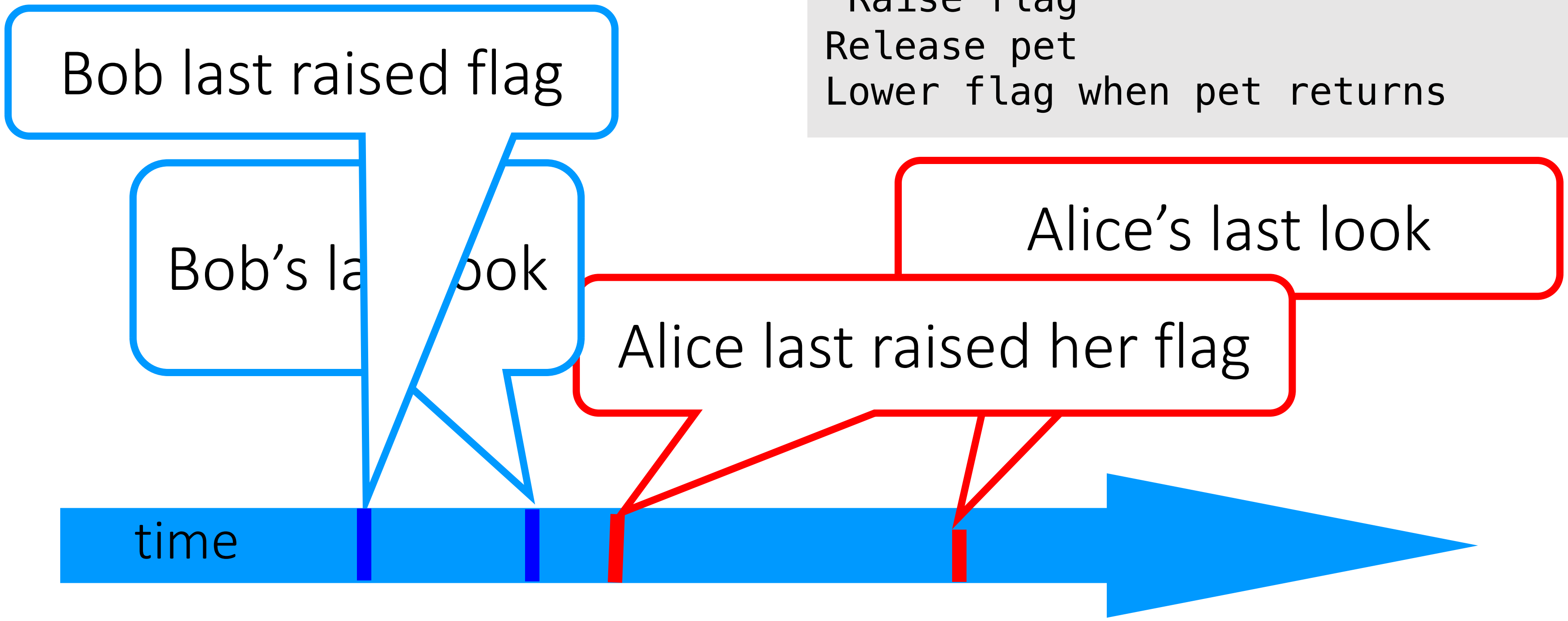
```
Thread B
Raise flag
While (A's flag is up)
 Lower B flag
 Wait for A flag to go down (looking)
 Raise flag
Release pet
Lower flag when pet returns
```



Proof of Mutual Exclusion

Thread A
```
Raise flag
Wait until B’s flag is down (looking)
Release pet
Lower flag when pet returns
```

Thread B
```
Raise flag
While (A’s flag is up)
 Lower B flag
 Wait for A flag to go down (looking)
 Raise flag
Release pet
Lower flag when pet returns
```



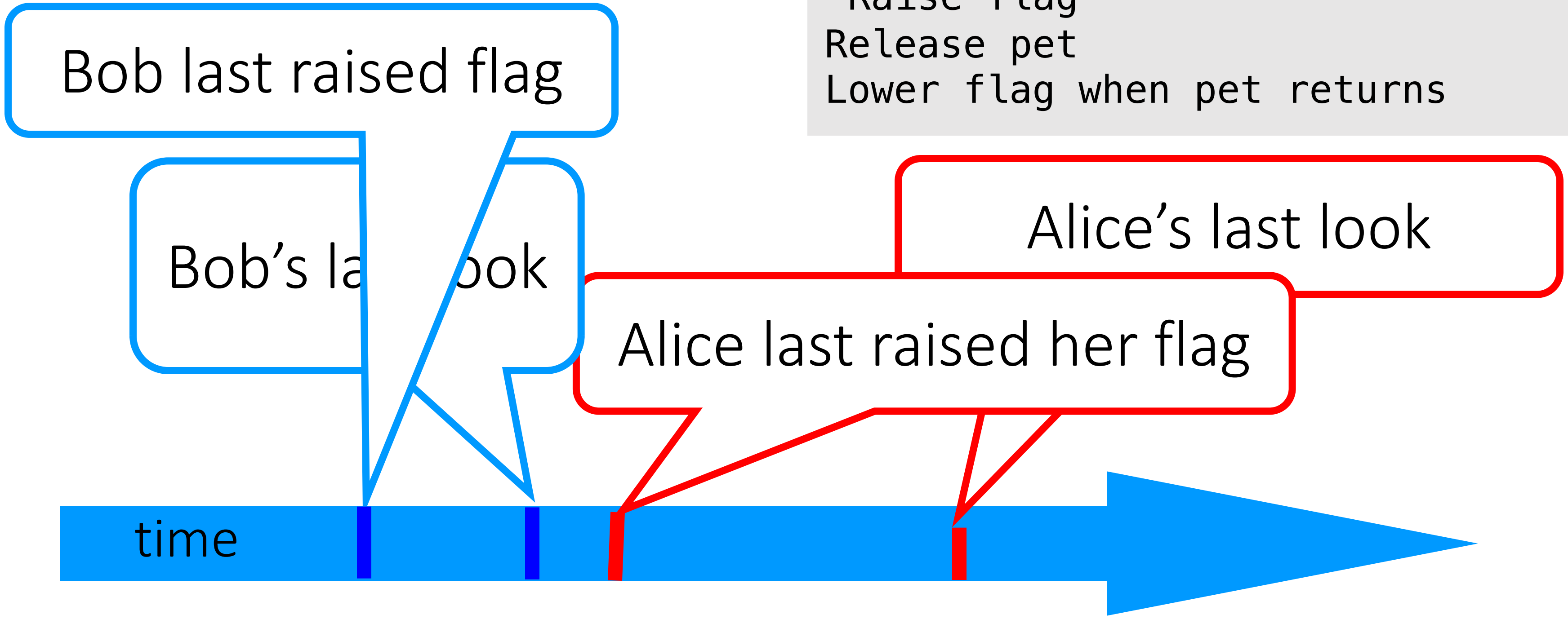
Proof of Mutual Exclusion

Thread A

```
Raise flag
Wait until B’s flag is down (looking)
Release pet
Lower flag when pet returns
```

Thread B

```
Raise flag
While (A’s flag is up)
 Lower B flag
 Wait for A flag to go down (looking)
 Raise flag
Release pet
Lower flag when pet returns
```



Alice must have seen Bob’s Flag. A Contradiction

Alice and Bob Mutual Exclusion Proof cont'd

Assume by way contradiction that we do not have mutual exclusion. **We are assuming that both pets are in the pond.**

Therefore, both A and B had a last "looking" action before they let their pet entered the pond.

Consider the one who finished this looking action first.

- Someone must be first (i.e., in a single-CPU, instructions are executed sequentially)

→ **When B (or A) looked, B (or A) saw that the other one's flag was down.**

Alice and Bob Mutual Exclusion Proof cont'd

Without loss of generality, assume it was B who finished the looking action first (i.e., A looked last).

- so B had (A's flag == down) as true, otherwise B couldn't have entered the critical section (i.e., send pet in the pond).
- So it follows that A's flag was up ***after*** B finished the looking action.

- Therefore, **A's looking was completely after** the end of B's raising of their flag (remember, threads first raise flag, then look)
- So A must have seen B's flag up and could not have entered the critical section,
  → Contradiction with the initial assumption that both threads are in the critical section at the same time.

Proof of No Deadlock

- Claim: If only one pet wants in, it gets in
- Deadlock requires both continually trying to get in
- If Bob sees Alice’s flag, he backs off, gives her priority

(Alice’s lexicographic privilege)

Further Optional Reading

Operating Systems: Three Easy Pieces by R. & A. Arpaci-Dusseau

Chapters 25 – 31 (inclusive) https://pages.cs.wisc.edu/~remzi/OSTEP/

For a very helpful alternative explanation on the producer/consumer problem, with C code tutorial, check out this link from the CodeVault YouTube channel.

You are also encouraged to check the other CodeVault tutorials on multi-processing and multi-threading in C/Unix (link).

Credits:

Some slides adapted from the OS courses of Profs. Remzi and Andrea Arpaci-Dusseau (University of Wisconsin-Madison), Prof. Willy Zwaenepoel (University of Sydney).