Week 5

Multithreading

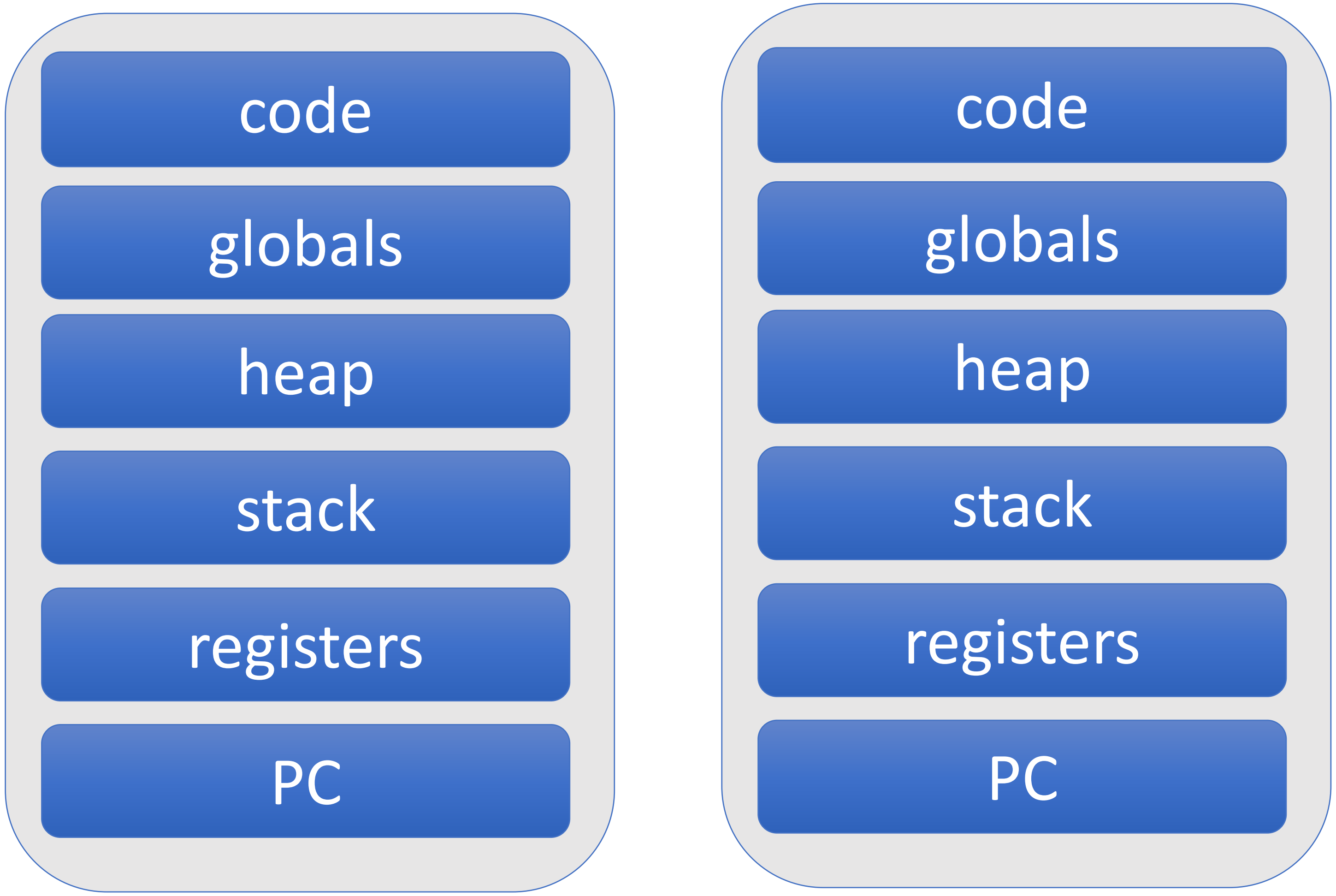
Max Kopinsky
4 February 2025

Recap Week 4
Concurrency – Option 1

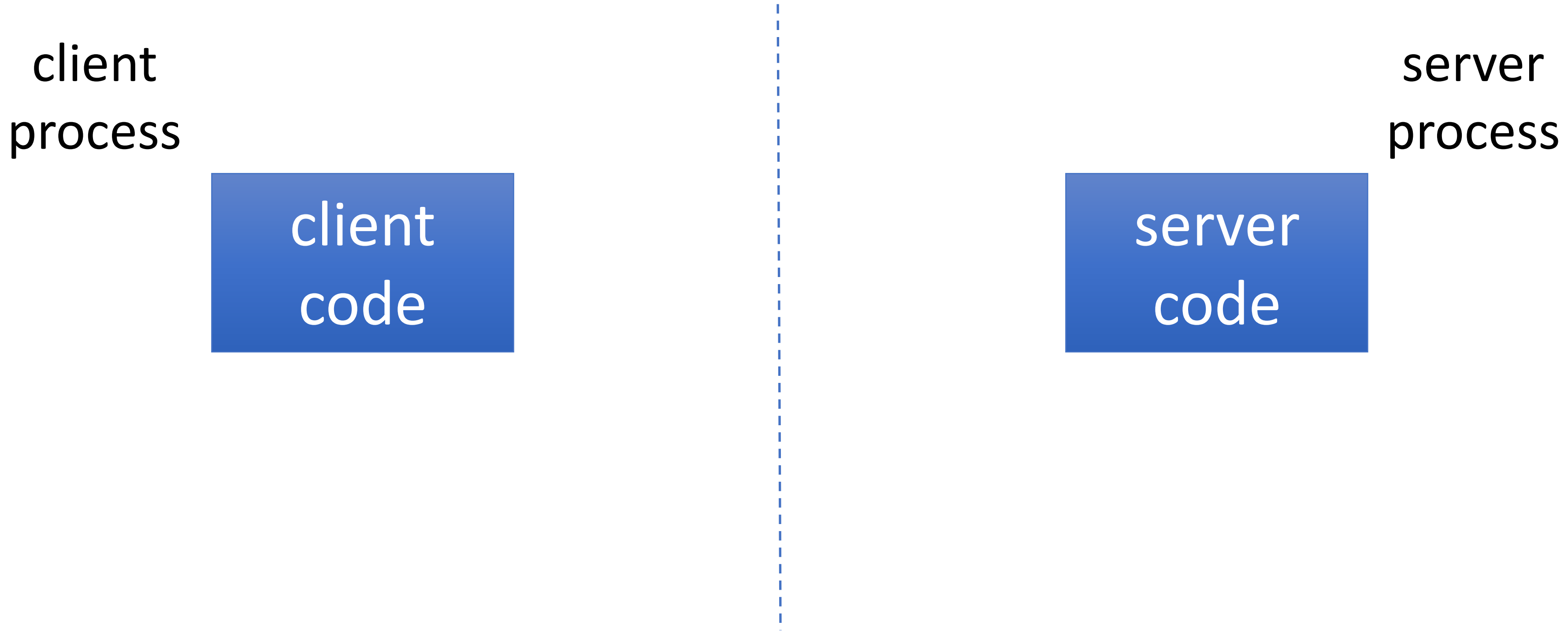
- Build apps from many communicating **processes**
- Communicate through **message passing / RPCs**
- Pros
  - If one process crashes, other processes unaffected
- Cons
  - High communication overheads

Last week's focus

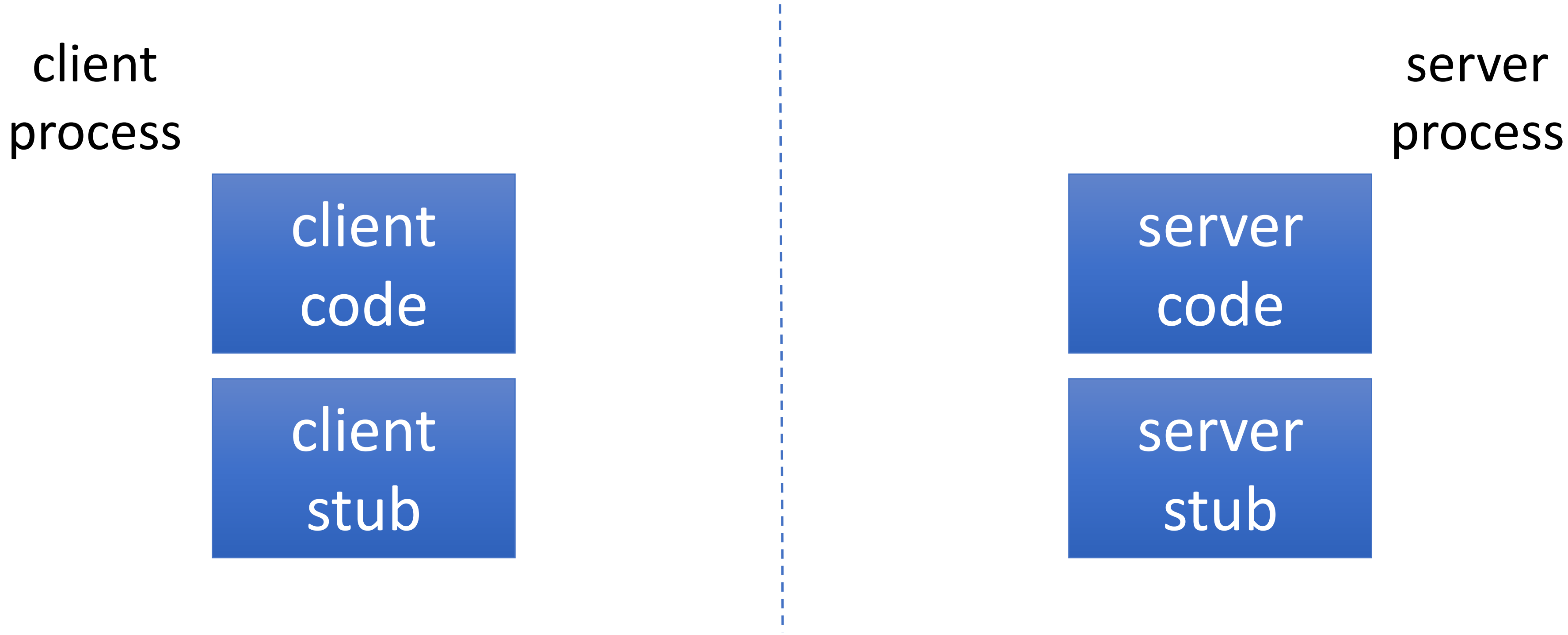
Recap Week 4: Two Processes



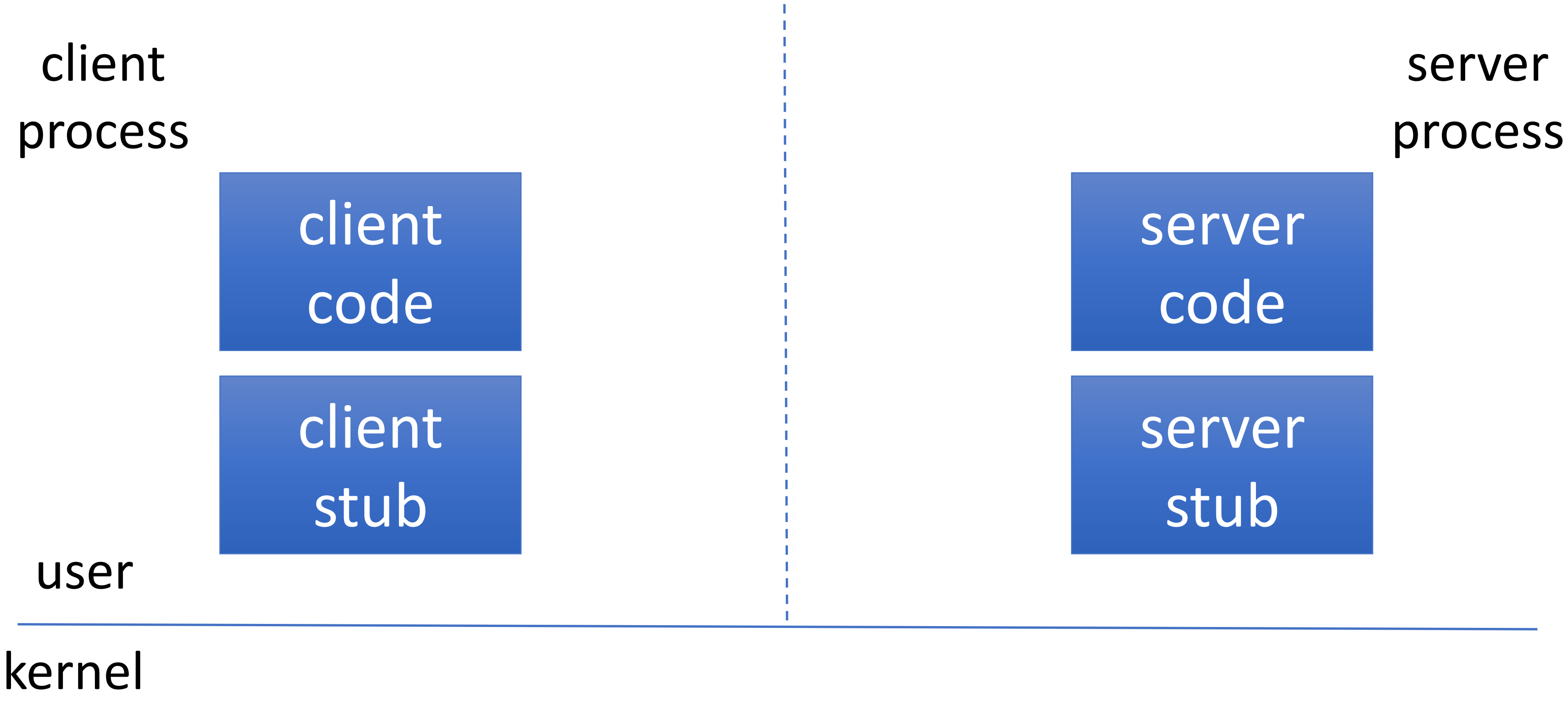
Recap Week 4 RPC Implementation



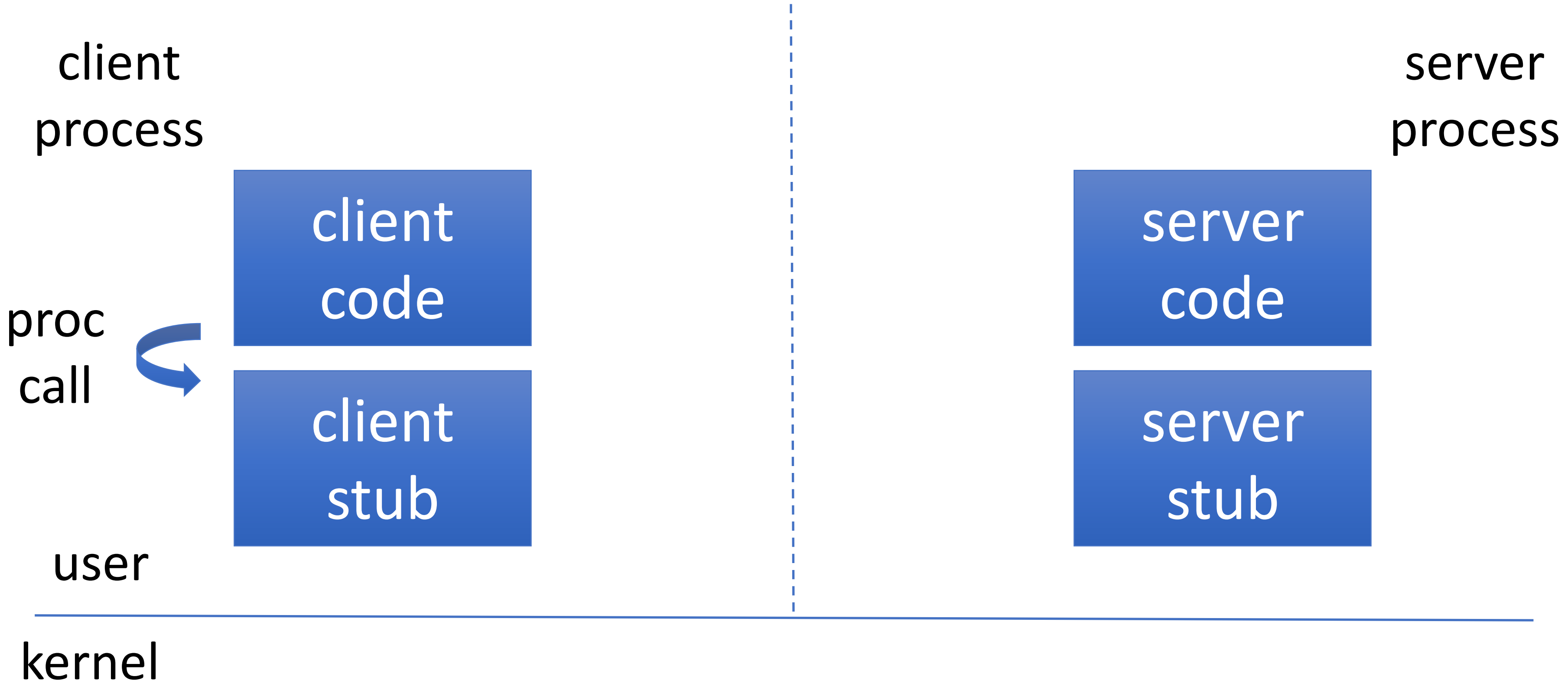
Recap Week 4 Client and Server Stubs



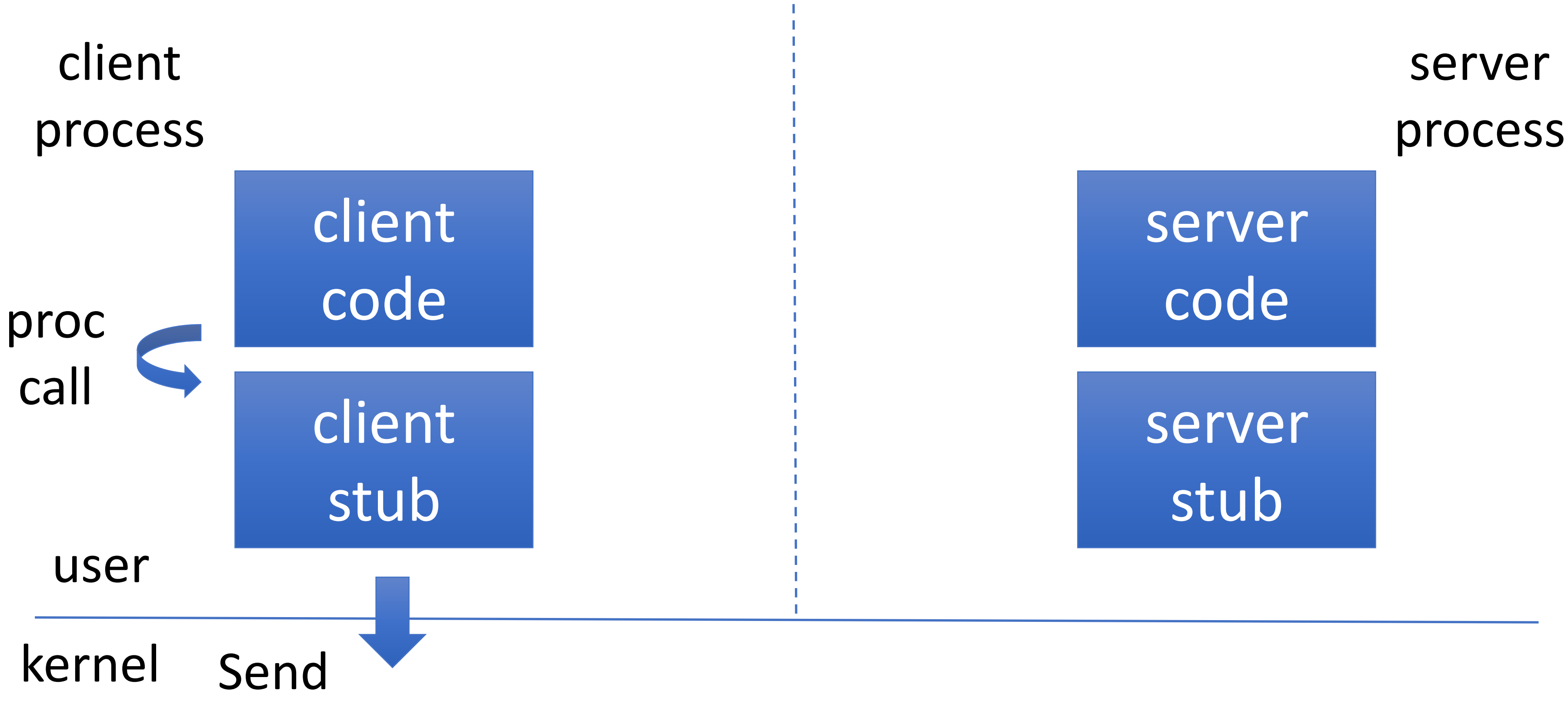
Recap Week 4 RPC Implementation: Call



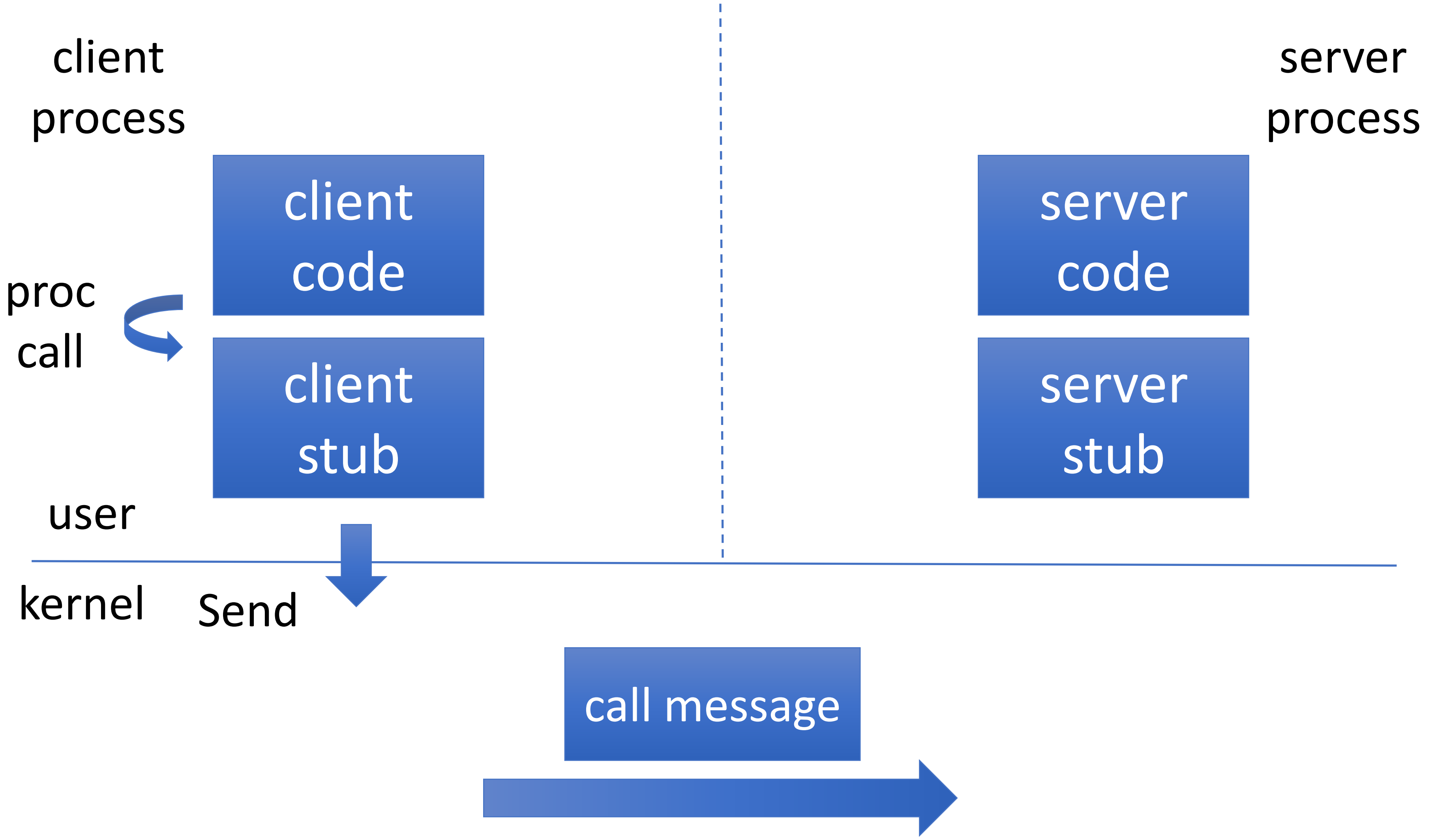
Recap Week 4 RPC Implementation: Call



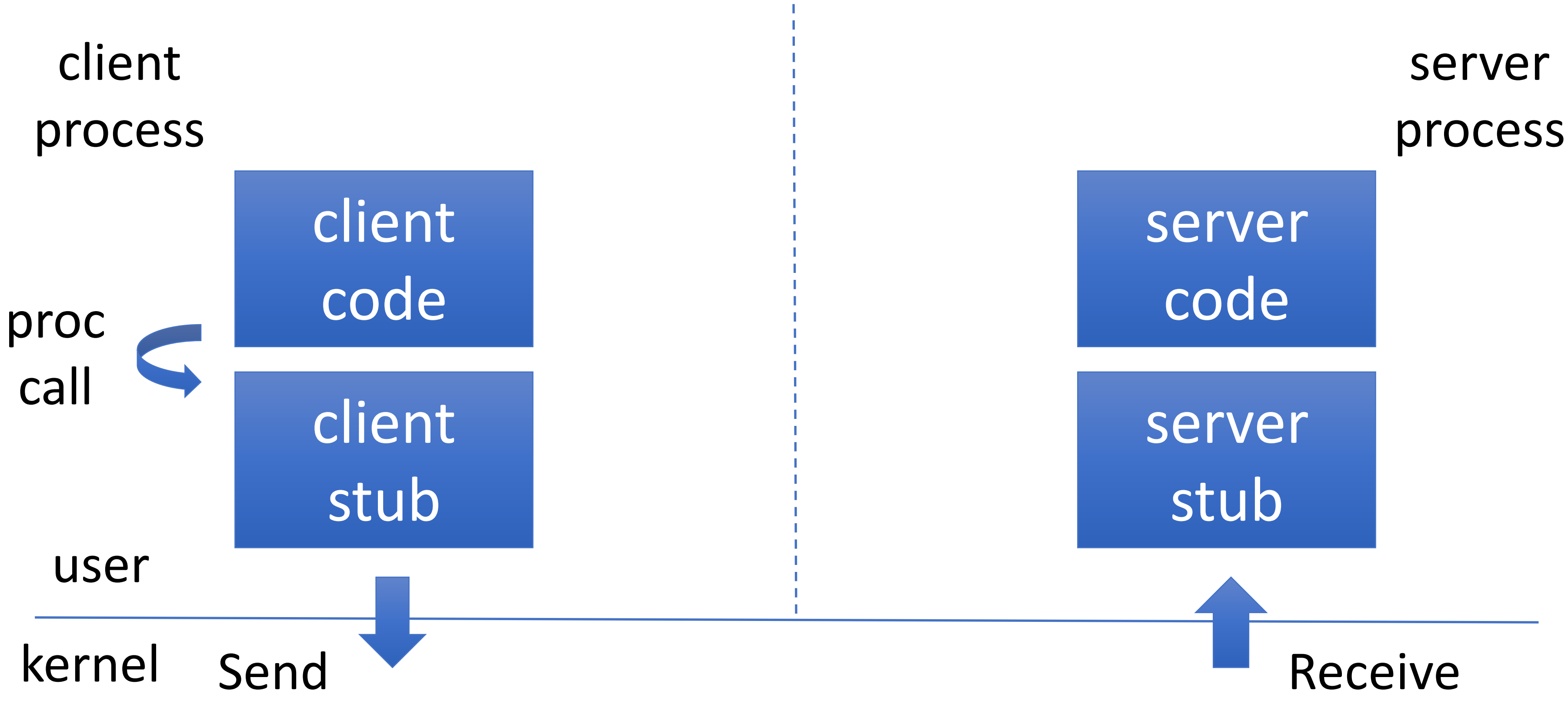
Recap Week 4 RPC Implementation: Call



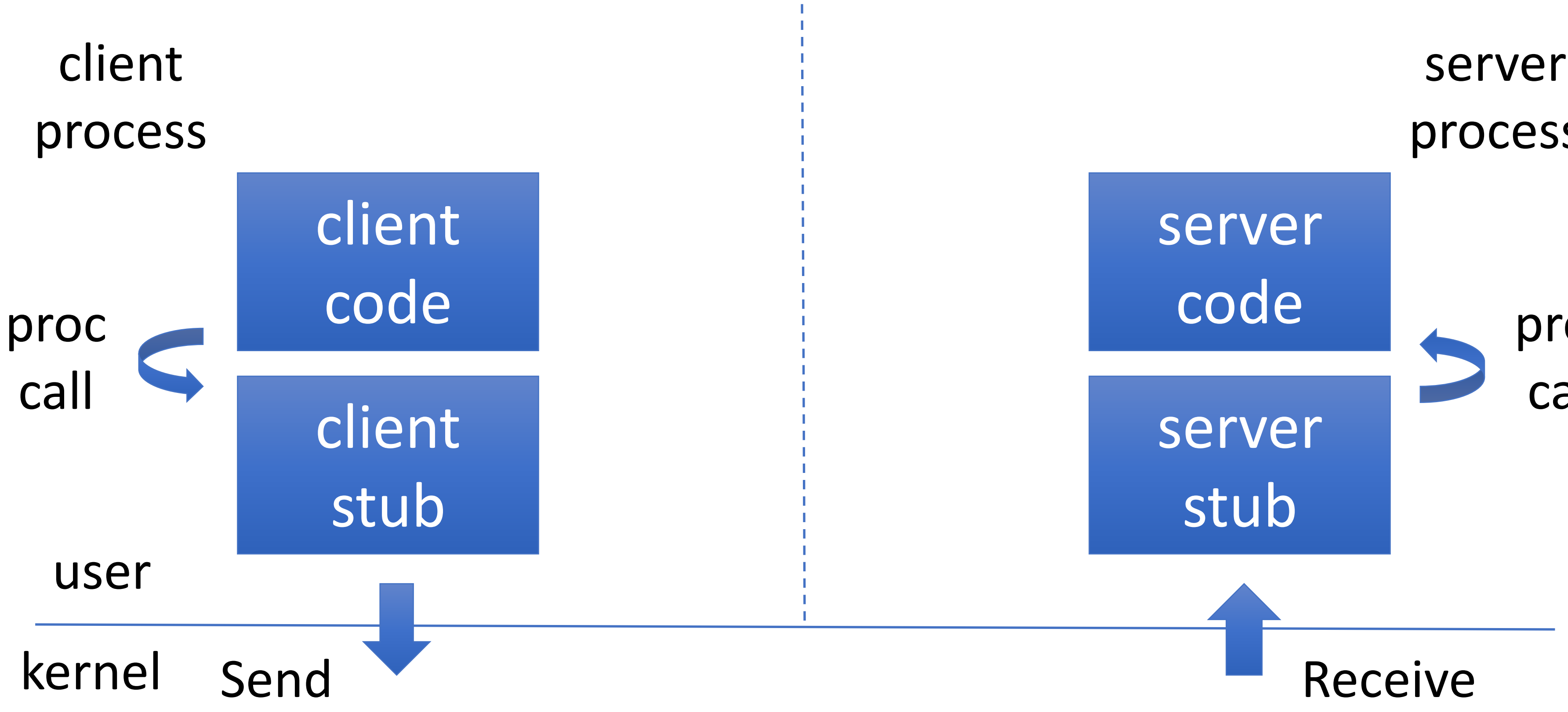
Recap Week 4 RPC Implementation: Call



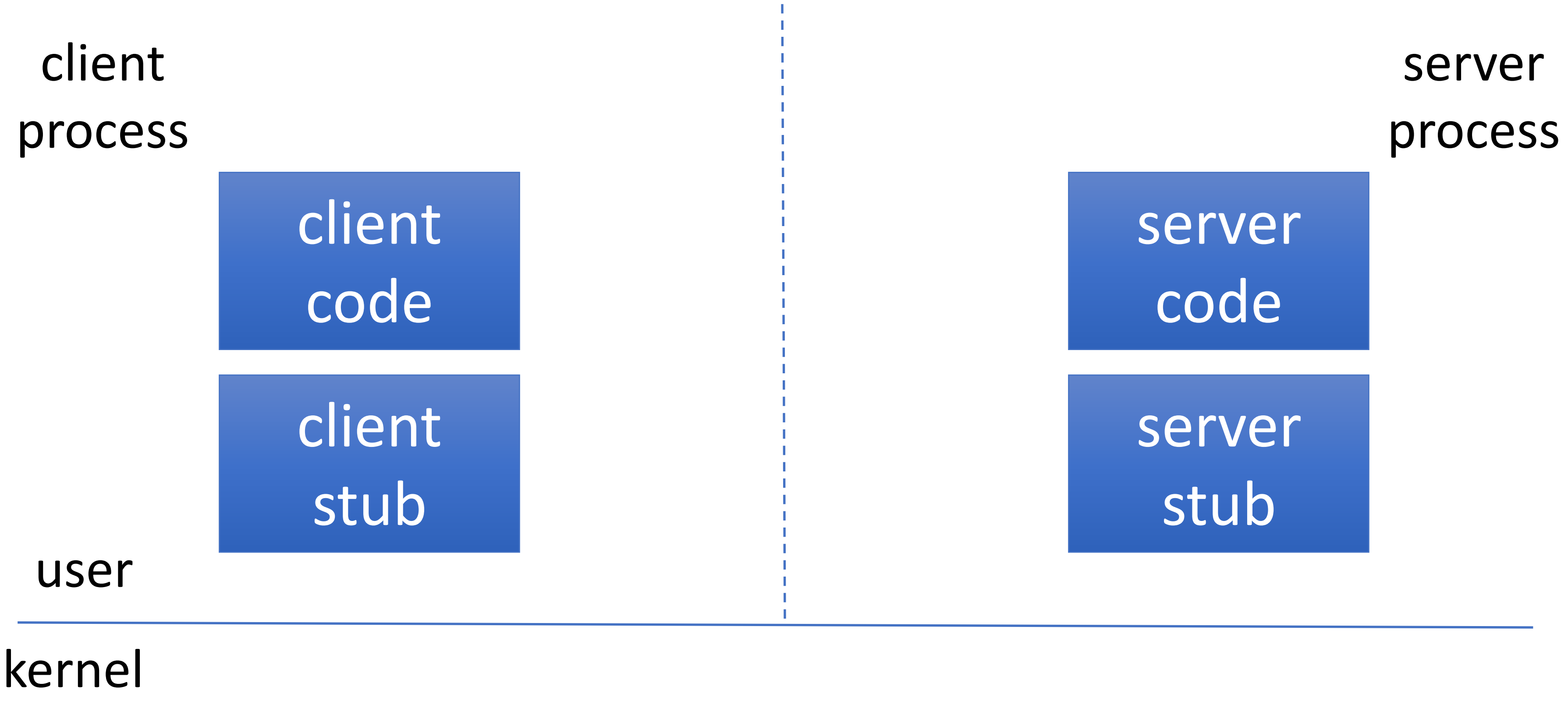
Recap Week 4 RPC Implementation: Call



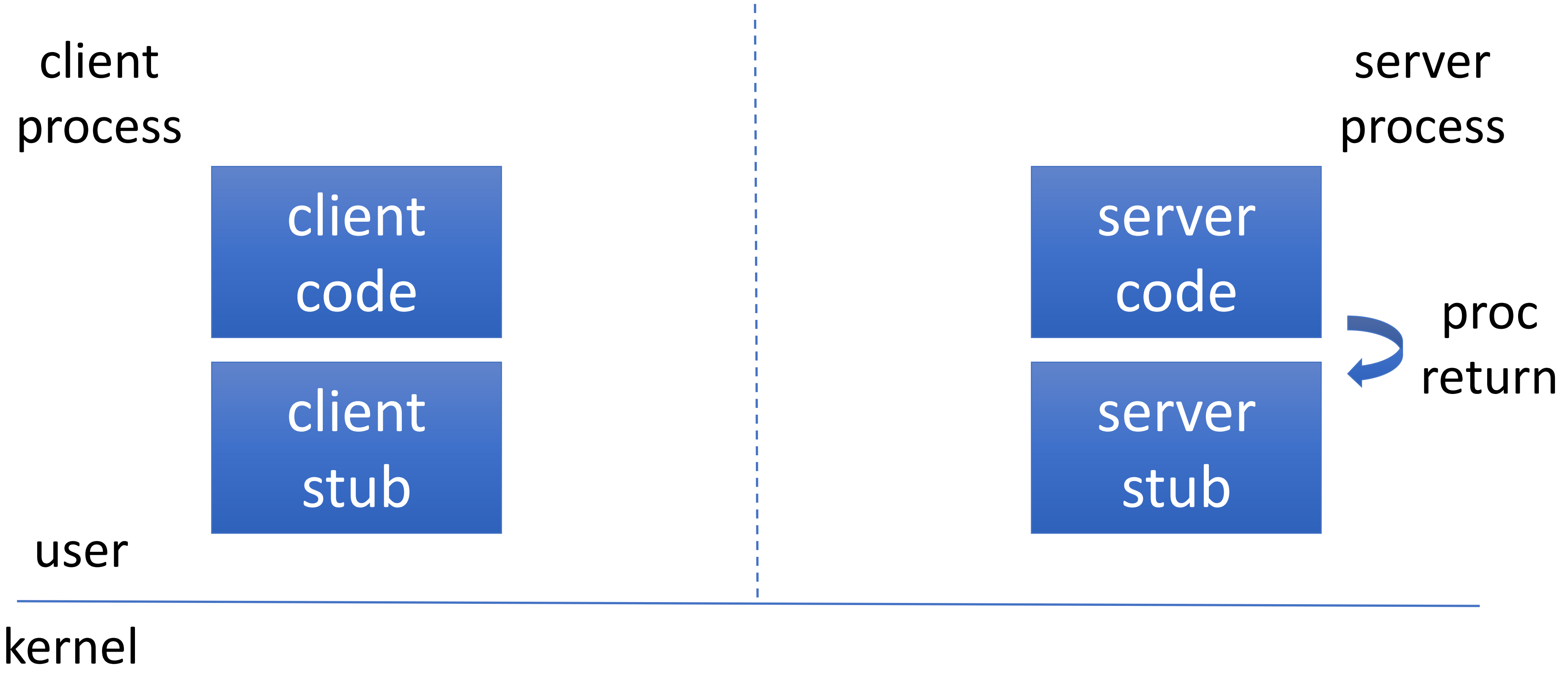
Recap Week 4 RPC Implementation: Call



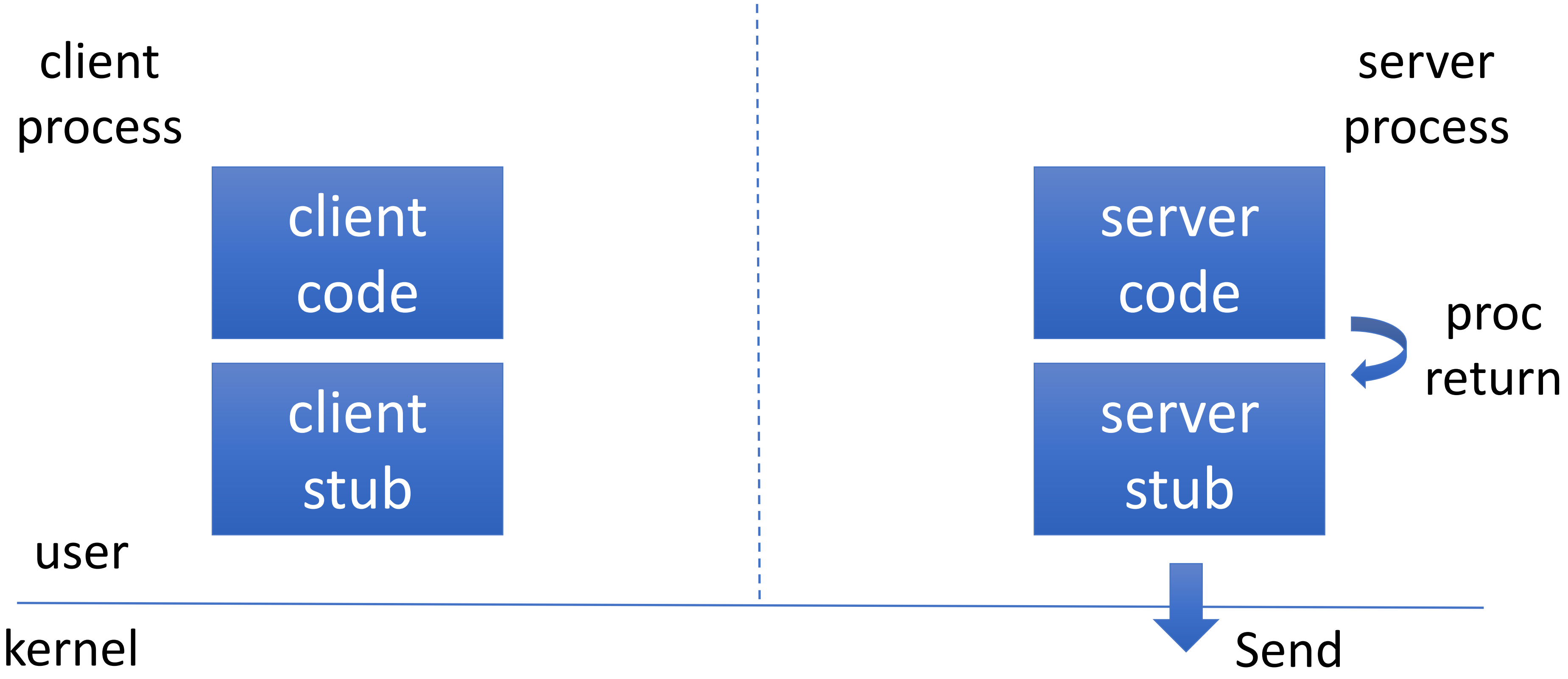
Recap Week 4 RPC Implementation: Return



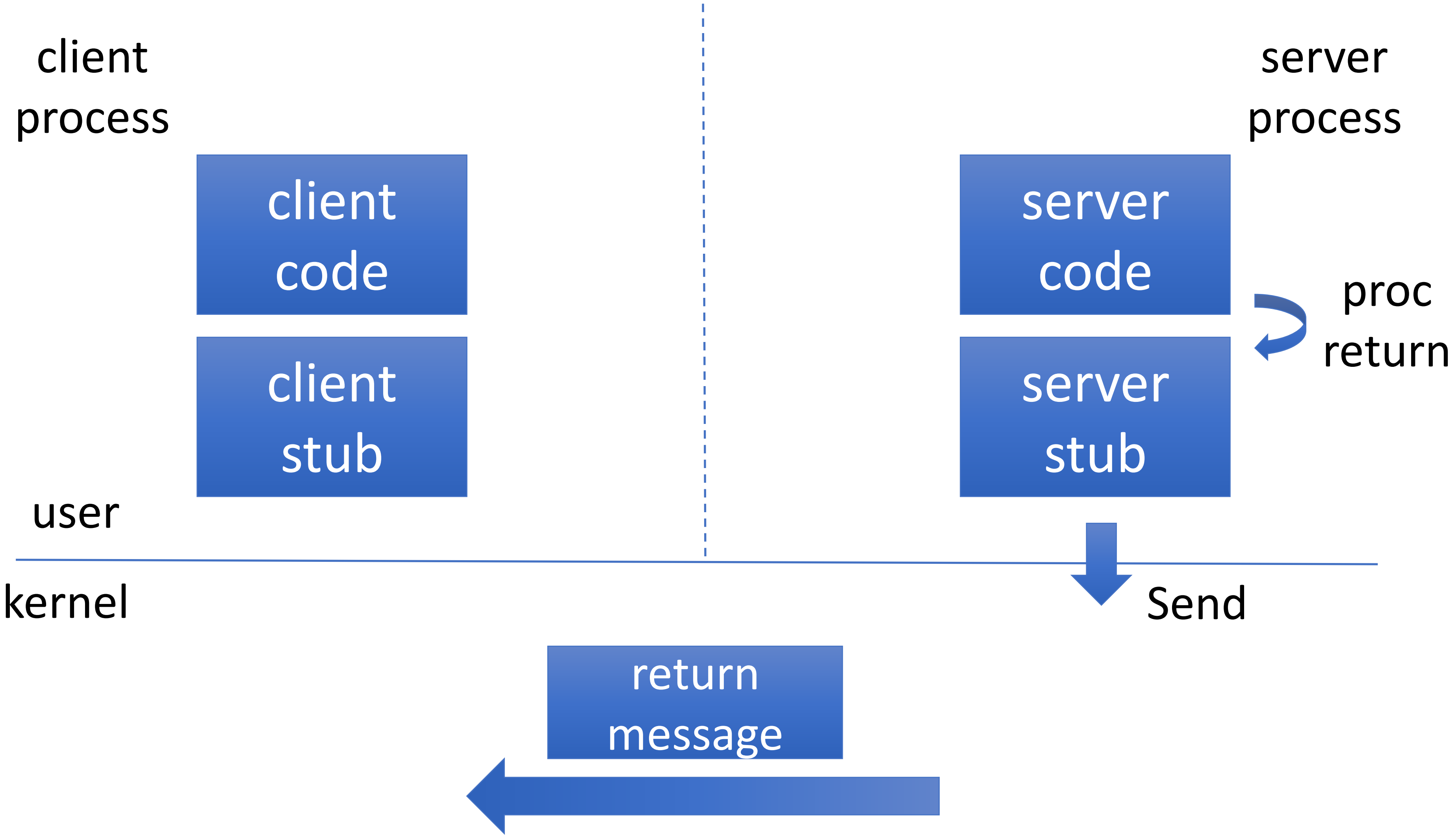
Recap Week 4 RPC Implementation: Return



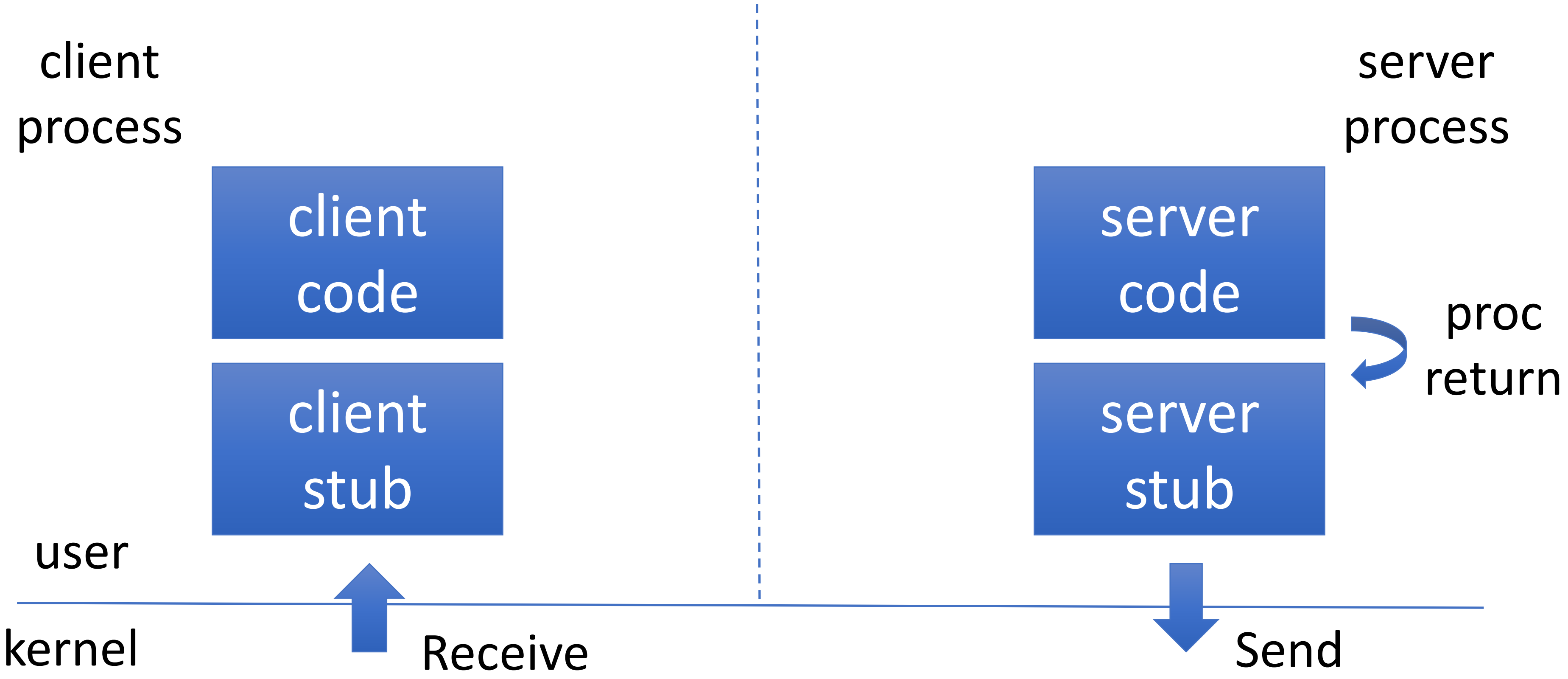
Recap Week 4 RPC Implementation: Return



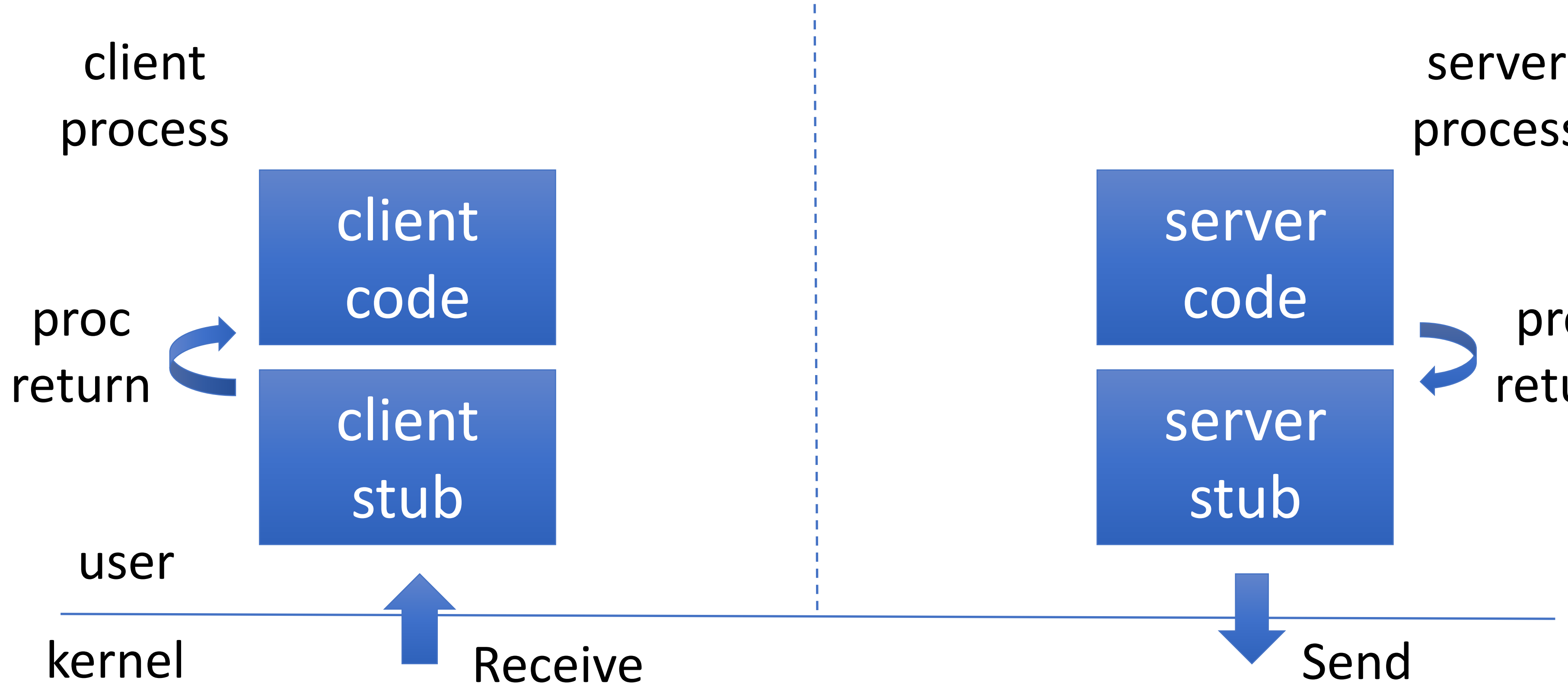
Recap Week 4 RPC Implementation: Return



Recap Week 4 RPC Implementation: Return



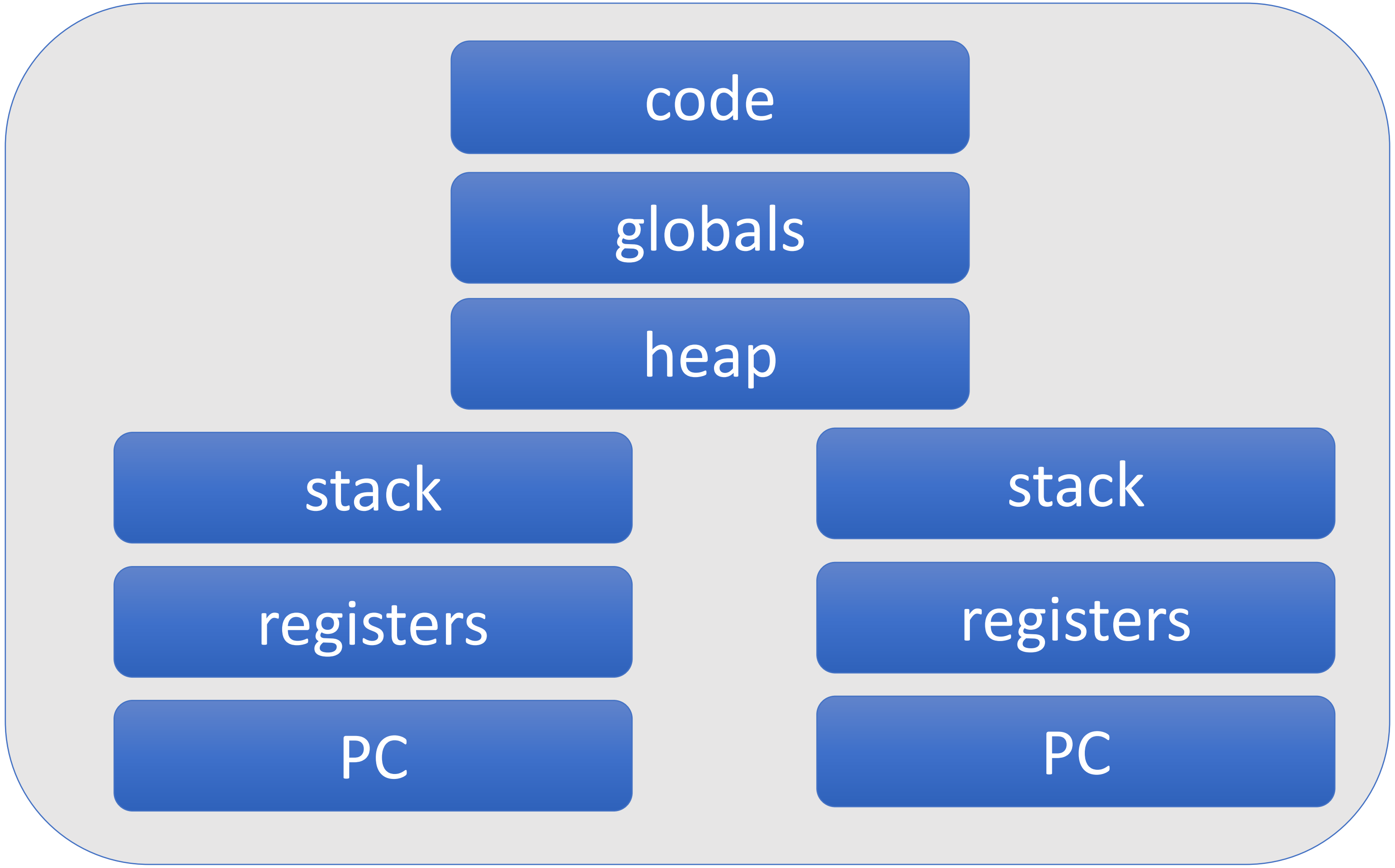
Recap Week 4 RPC Implementation: Return



Recap Week 3
Concurrency – Option 2

- New abstraction: **thread**
- Multiple threads in a process
- Threads are like processes except
  - Multiple threads in the same process share an address space
    - Communicate through shared address space
  - If one thread crashes,
    - the entire process, including other threads, crashes

Recap Week 3 Two Threads in a Process



Key Concepts for Today

- Multithreading techniques
  - Division of work
  - Synchronization of shared data
  - Fine-grain locking
  - Privatization
  - Producer/consumer problem

In General

- Processes provide separation
  - In particular, memory separation (no shared data)
  - Suitable for coarse-grain interaction

- Threads do not provide separation
  - In particular, threads share memory (shared data)
  - Suitable for tighter integration

Concrete Example: Web Server

- Serving static content (files)
  - Probably no bugs
  - Can easily be done in a **multithreaded process**

- Serving dynamic (third-party) content
  - No guarantees about bugs
  - Keep in a different process

Shared Data

- Advantage:
  - Many threads can read/write it

- Disadvantage:
  - Many threads can read/write it
  - Can lead to *data races*

Basic Approach to Multithreading

1. Divide “work” among multiple threads
2. Share data
   - Which data is shared?
     - **Global variables and heap**
     - Not local variables, not read-only variables
   - Where is shared data accessed?
     - Put shared data access in **critical section**

Example: Single-Threaded Code

```
main(){
  int i
  int sum = 0
  int prod = 1
  for (i = 0; i < MAX; ++i) {
    c = a[i] * b[i]
    sum += c
    prod *= c
  }
}
```

Approach to Multithreading

- Divide “work” among multiple threads
- Example: give each thread equal number of iterations

Example: Divide Work

```
main() {
  int i
  int sum= 0, prod = 1
  for (i = 0; i < MAX_THREADS; i++) { Pthread_create(…) }
  for (i = 0; i < MAX_THREADS; i++) { Pthread_join(…) }
  printf(sum)
  printf(prod)
}

Threadcode() {
  int i, c
  for (i=my_min; i<my_max; i++) {
    c = a[i] * b[i]
    sum += c
    prod *=c
  }
}
```

Example: Divide Work

```
int sum, prod
main() {
  int i
  sum= 0, prod = 1
  for (i=0; i<MAX_THREADS; i++) { Pthread_create(…) }
  for (i=0; i<MAX_THREADS; i++) { Pthread_join(…) }
  printf(sum)
  printf(prod)
}

Threadcode() {
  int i, c
  for (i=my_min; i<my_max; i++) {
    c = a[i] * b[i]
    sum += c
    prod *=c
  }
}
```

local data: i, c, my_min, my_max

Shared read-only data: a, b

Shared data: sum, prod

Example: Divide Work

```
int sum, prod
main() {
  int i
  sum= 0, prod = 1
  for( i=0; i<MAX_THREADS; i++ ) { Pthread_create(...) }
  for( i=0; i<MAX_THREADS; i++ ) { Pthread_join(...) }
  printf(sum)
  printf(prod)
}

Threadcode() {
  int i, c
  for( i=my_min; i<my_max; i++ ) {
    c = a[i] * b[i]
    sum += c
    prod *=c
  }
}
```

Protect access to shared data with mutex

Shared data: sum, prod

Example: Divide Work

```
int sum, prod
main() {
  int i
  sum= 0, prod = 1
  for( i=0; i<MAX_THREADS; i++ ) { Pthread_create(...) }
  for( i=0; i<MAX_THREADS; i++ ) { Pthread_join(...) }
  printf(sum)
  printf(prod)
}

Threadcode() {
  int i, c
  for( i=my_min; i<my_max; i++ ) {
    c = a[i] * b[i]
    sum += c
    prod *=c
  }
}
```

Protection not necessary here because only the main thread accesses sum, prod

Protect access to shared data with mutex

Shared data: sum, prod

Example: Synchronization with 1 mutex

```
Threadcode() {
 int i
 for( i=my_min; i<my_max; i++ ) {
  c = a[i] * b[i]
  pthread_mutex_lock(biglock)
  sum += c
  prod *= c
  pthread_mutex_unlock(biglock)
 }
}
```

Why it will not work very well

- Single lock inhibits parallelism
- Two approaches:
  - Fine-grain locking:
    - Multiple locks on individual pieces of shared data
  - Privatization:
    - Make shared data accesses into private data accesses

Fine Grain Locking

- Define separate lock for sum and prod

Example: Finer-Grain Locking

```
Threadcode() {
  int i, c
  for (i=my_min; i<my_max; i++) {
    c = a[i] * b[i]
    pthread_mutex_lock(sumlock)
    sum += c
    pthread_mutex_unlock(sumlock)
    pthread_mutex_lock(prodlock)
    prod *= c
    pthread_mutex_unlock(prodlock)
  }
}
```

Example: Privatization

- Define for each thread
  - A local variable representing its sum
  - A local variable representing its product
- Use those for accesses in the loop
  - Become local accesses
  - No need for lock
- Only access shared data after the loop
  - Use lock there

Example: Privatization

```
Threadcode() {
  int i, c
  local_sum = 0
  local_prod = 1

  for (i=my_min; i<my_max; i++) {
    c = a[i] * b[i]
    local_sum += c
    local_prod *= c
  }

  pthread_mutex_lock(sumlock)
  sum += local_sum
  pthread_mutex_unlock(sumlock)
  pthread_mutex_lock(prodlock)
  prod *= local_prod
  pthread_mutex_unlock(prodlock)
}
```

Example: Privatization

```
Threadcode() {
  int i, c
  local_sum = 0
  local_prod = 1

  for( i=my_min; i<my_max; i++ ) {
    c = a[i] * b[i]
    local_sum += c
    local_prod *= c
  }

  pthread_mutex_lock(sumlock)
  sum += local_sum
  pthread_mutex_unlock(sumlock)
  pthread_mutex_lock(prodlock)
  prod *= local_prod
  pthread_mutex_unlock(prodlock)
}
```

Only **one access** to each lock per thread

Example: Privatization; Compare to before finer-grained lock accesses

```
Threadcode() {
 int i, c
 for (i=my_min; i<my_max; i++) {
  c = a[i] * b[i]
  pthread_mutex_lock(sumlock)
  sum += c
  pthread_mutex_unlock(sumlock)
  pthread_mutex_lock(prodlock)
  prod *= c
  pthread_mutex_unlock(prodlock)
 }
}
```

2 lock accesses
per thread, **per iteration**

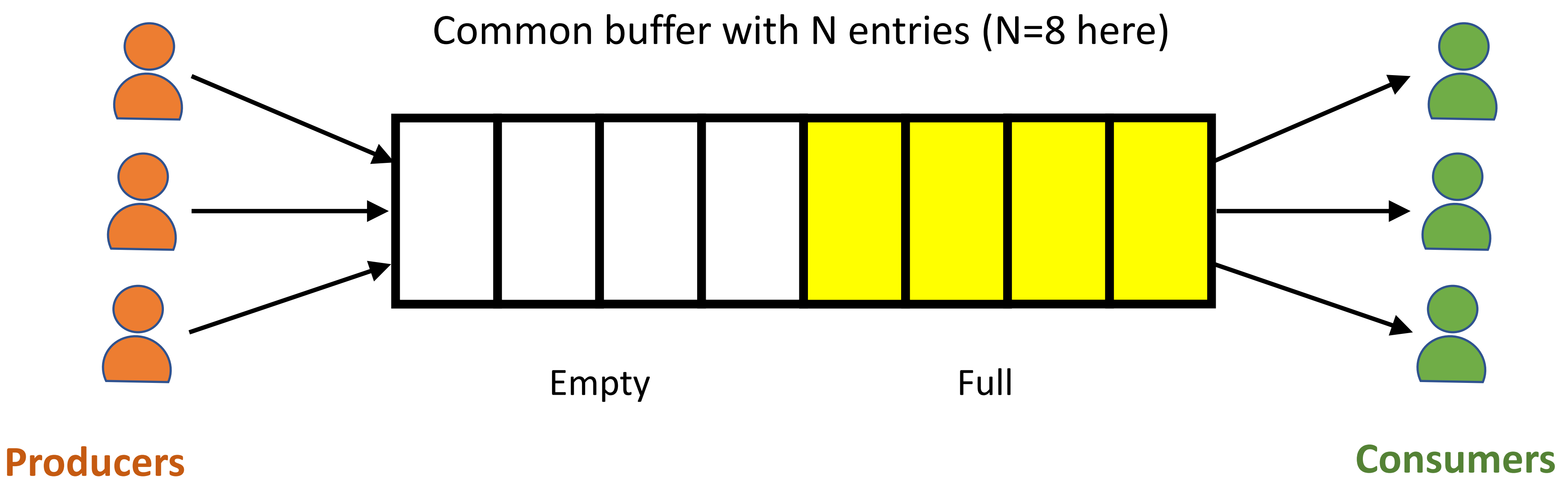
```
Threadcode() {
 int i, c
 local_sum = 0
 local_prod = 1

 for (i=my_min; i<my_max; i++) {
  c = a[i] * b[i]
  local_sum += c
  local_prod *= c
 }

 pthread_mutex_lock(sumlock)
 sum += local_sum
 pthread_mutex_unlock(sumlock)
 pthread_mutex_lock(prodlock)
 prod *= local_prod
 pthread_mutex_unlock(prodlock)
}
```

2 lock accesses
per thread, **in total**

Producer/Consumer Problem

Producer/Consumer Problem

- Arises when two or more threads communicate with each other.
  - Some threads “produce” data and other threads “consume” this data.
- Example of producer/consumer in OS: I/O queues

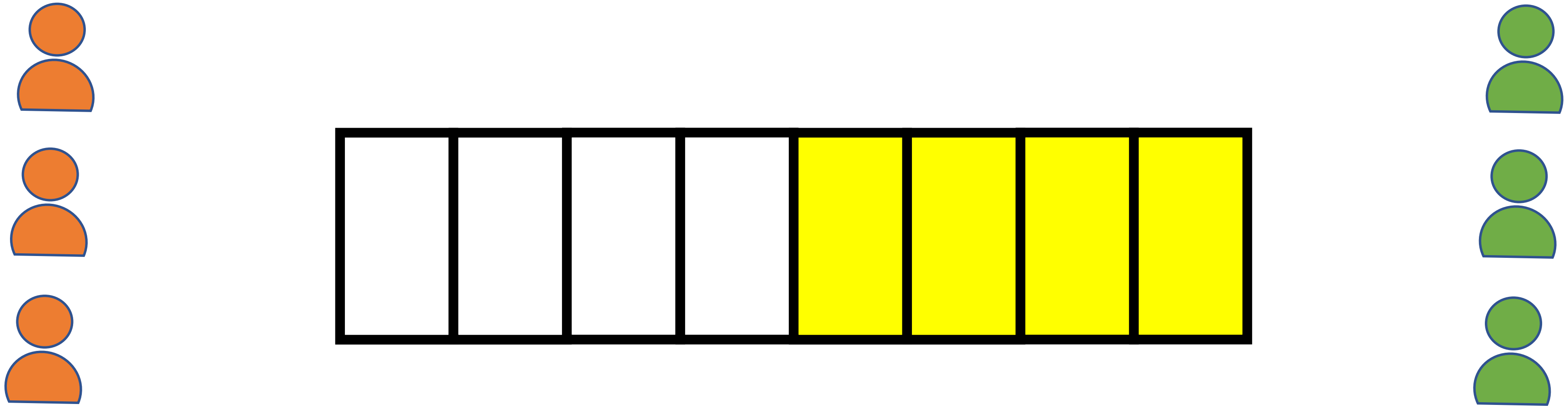


Producer/Consumer Problem

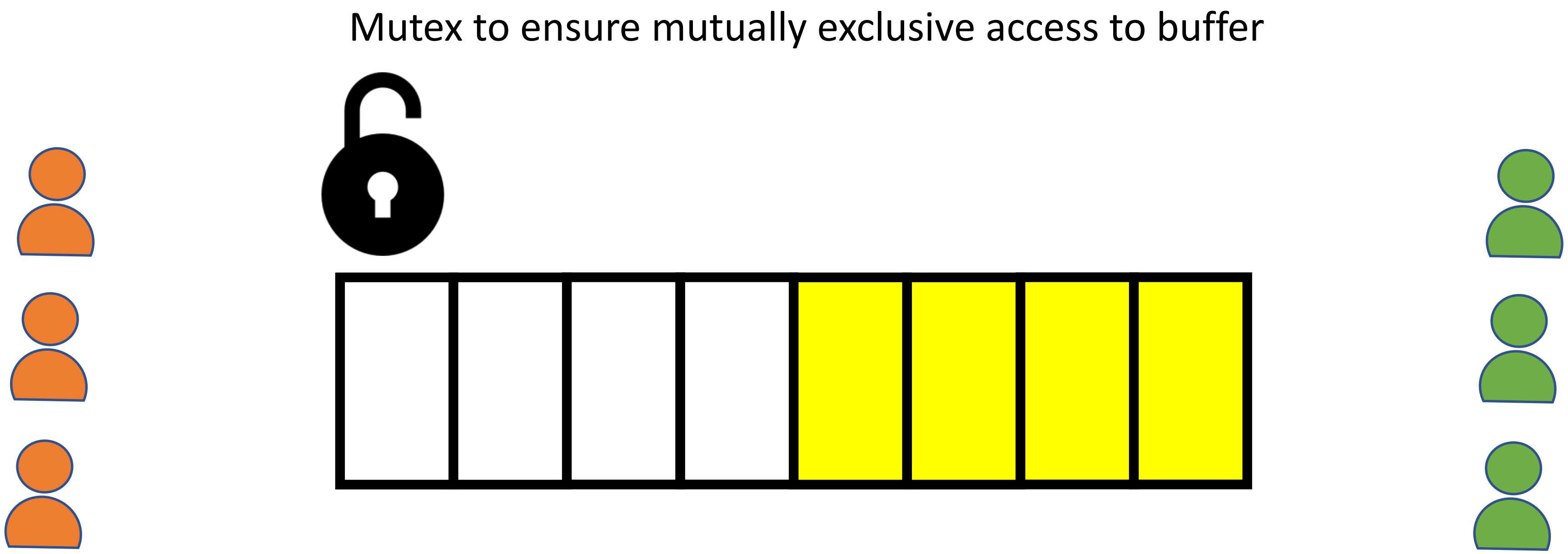
- Multiple producer-threads.
- Multiple consumer-threads.
- One shared bounded buffer with N entries.
- Requirements:
    - No production when all N entries are full.
    - No consumption when all entries are empty.
    - Access to the buffer is **mutually exclusive.**
- To avoid busy-waiting, any user that successfully accesses the buffer must produce/consume successfully.

Solve Producer-Consumer with Locks?

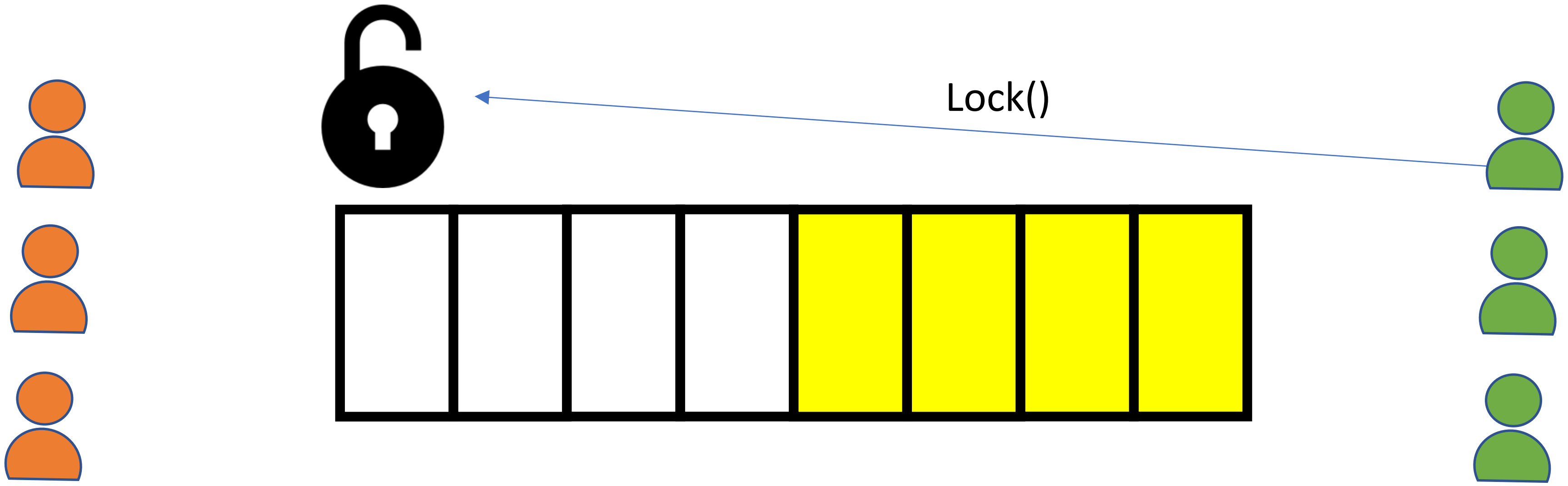
Think!



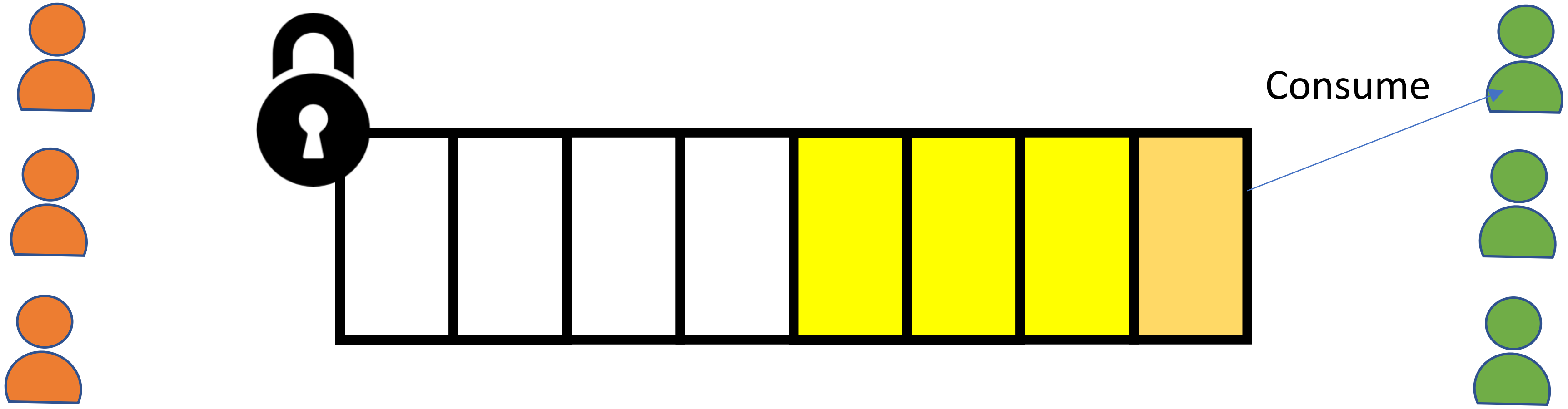
Solve Producer-Consumer with Locks? (Incorrect)



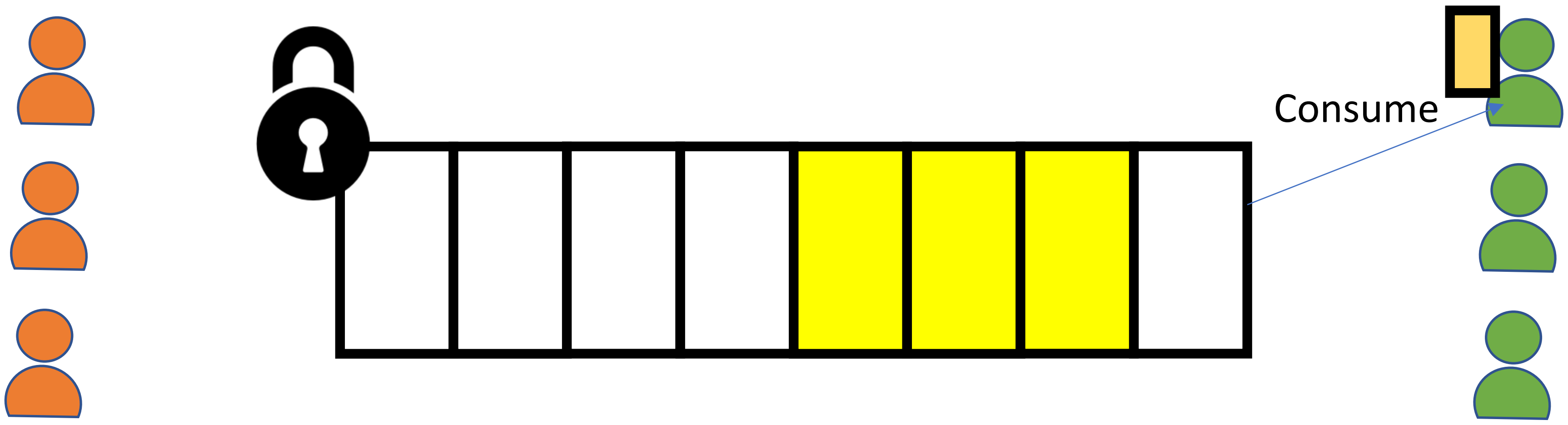
Solve Producer-Consumer with Locks? (Incorrect)



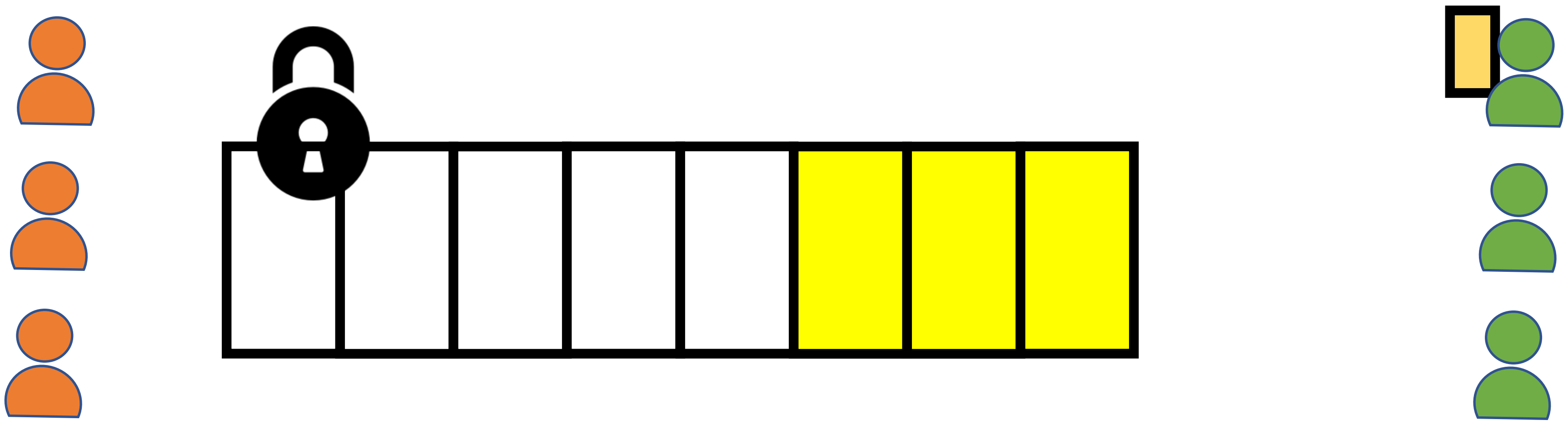
Solve Producer-Consumer with Locks? (Incorrect)



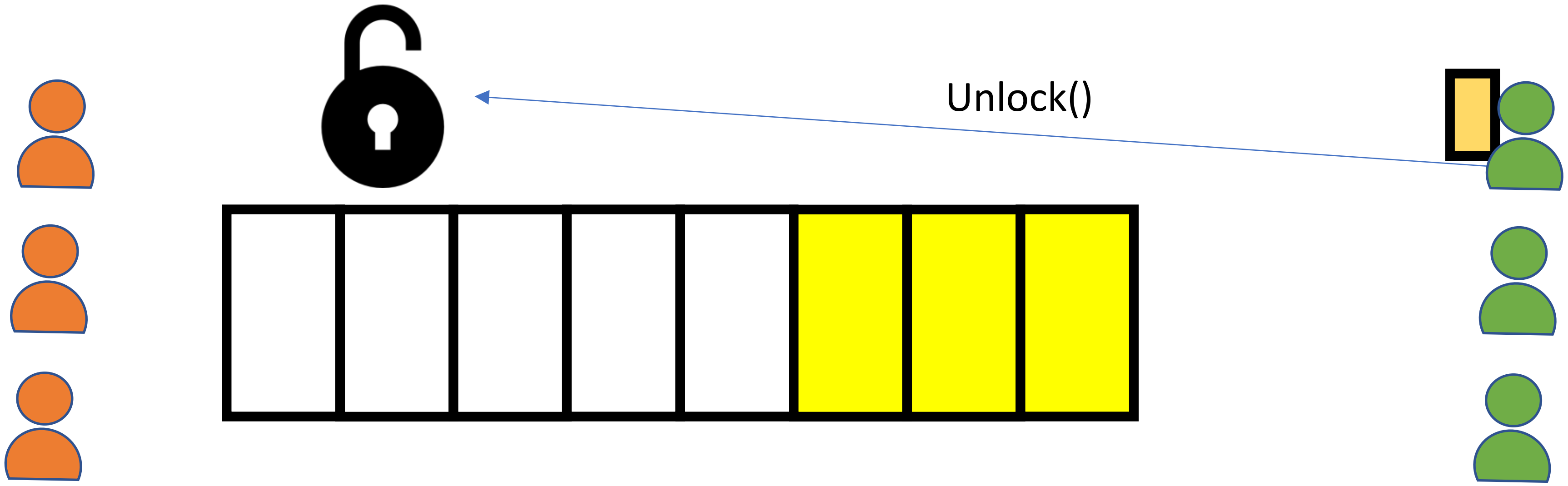
Solve Producer-Consumer with Locks? (Incorrect)



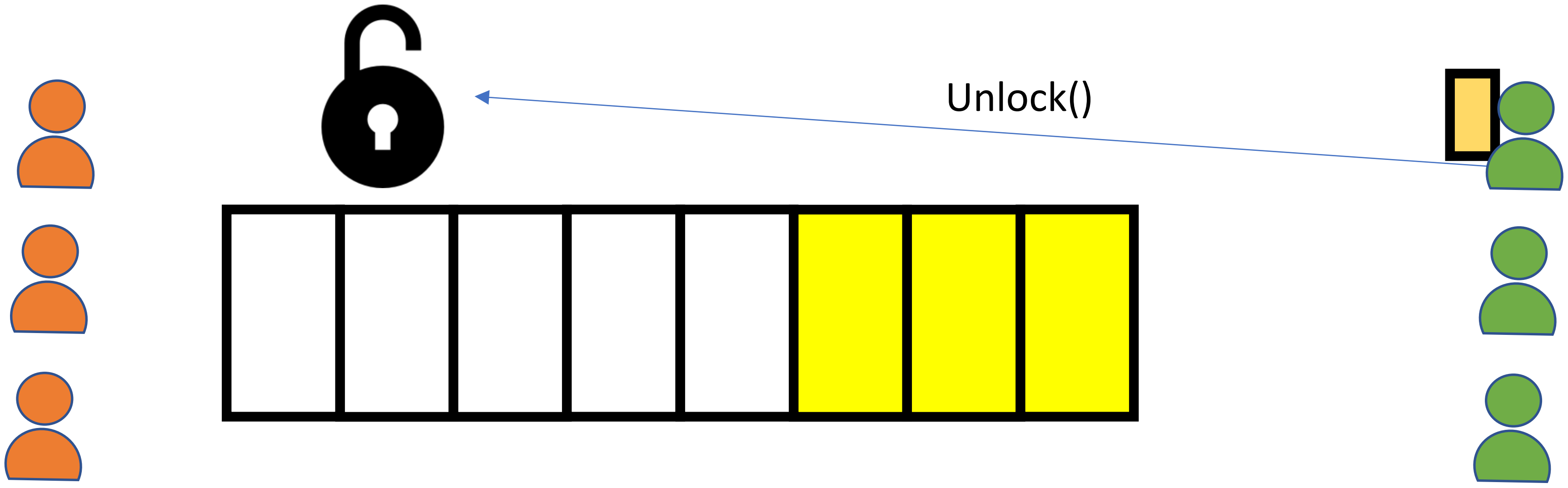
Solve Producer-Consumer with Locks? (Incorrect)



Solve Producer-Consumer with Locks? (Incorrect)



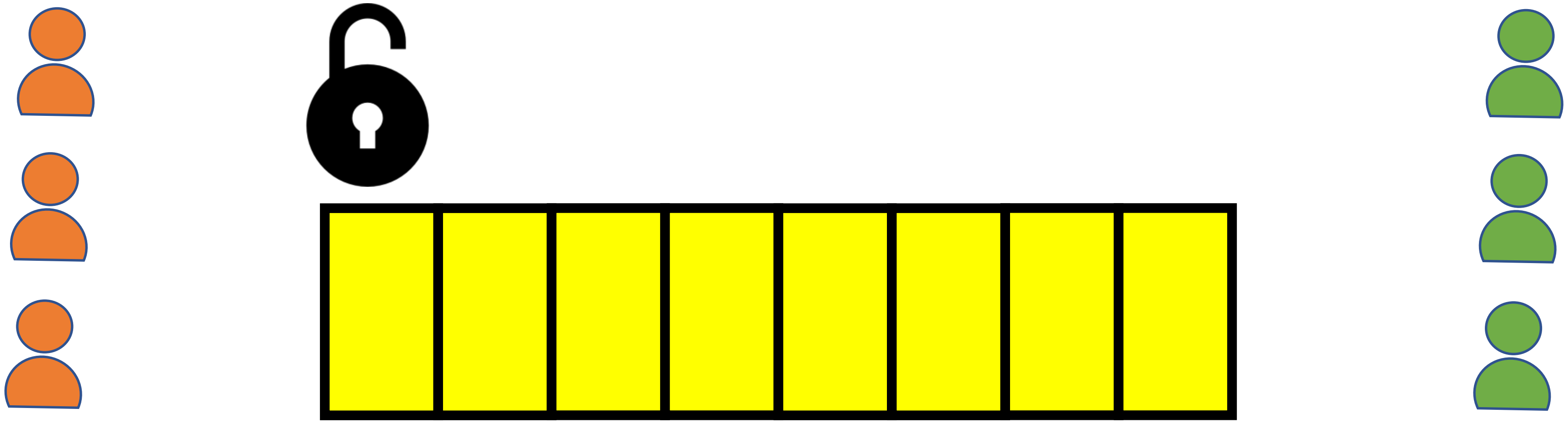
Solve Producer-Consumer with Locks? (Incorrect)



Problem? Think!

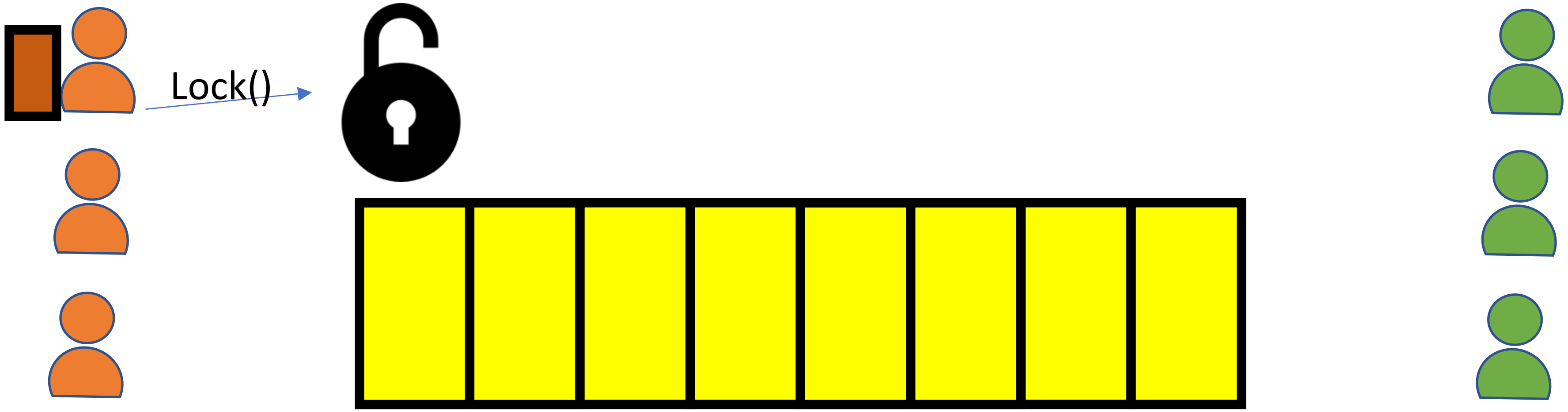
Problem: Threads can’t know the state of the buffer before acquiring the lock

- Problem 1: Buffer is full and producer want to add entries



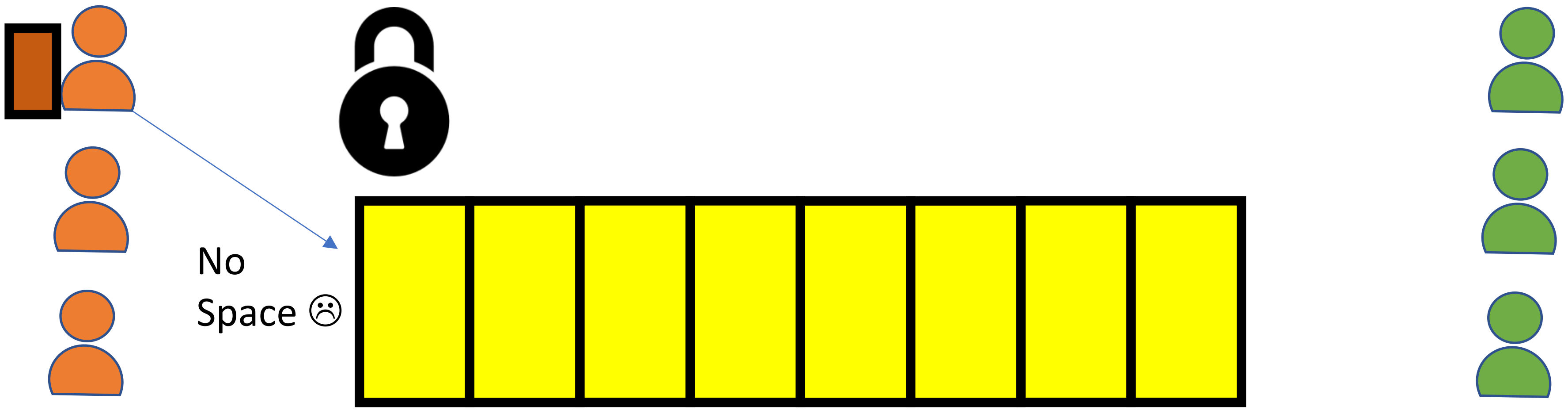
Problem: Threads can't know the state of the buffer before acquiring the lock

- Problem 1: Buffer is full and producer want to add entries



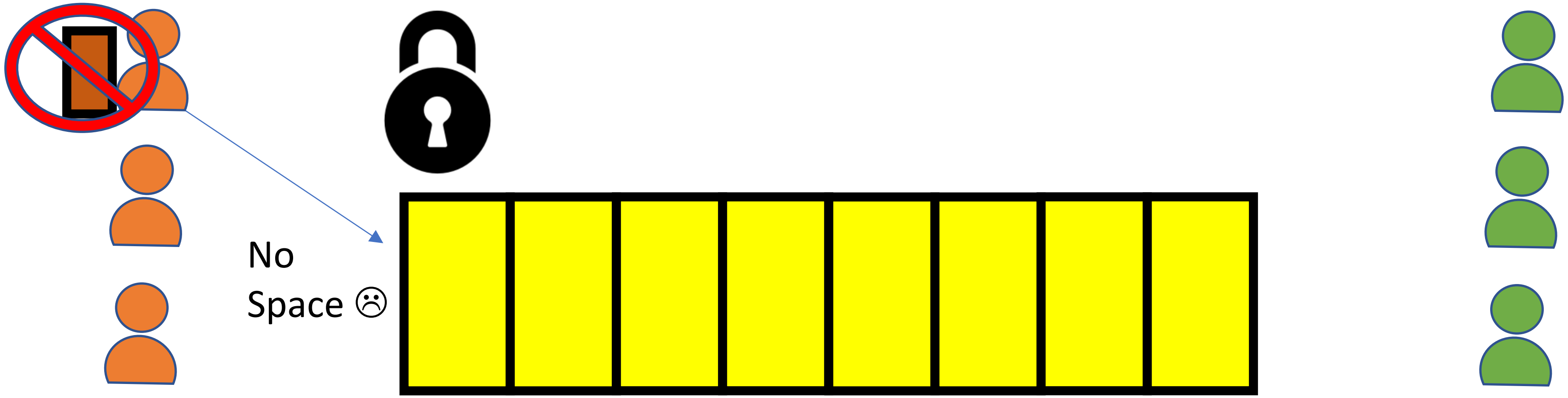
Problem: Threads can't know the state of the buffer before acquiring the lock

- Problem 1: Buffer is full and producer want to add entries



Problem: Threads can’t know the state of the buffer before acquiring the lock

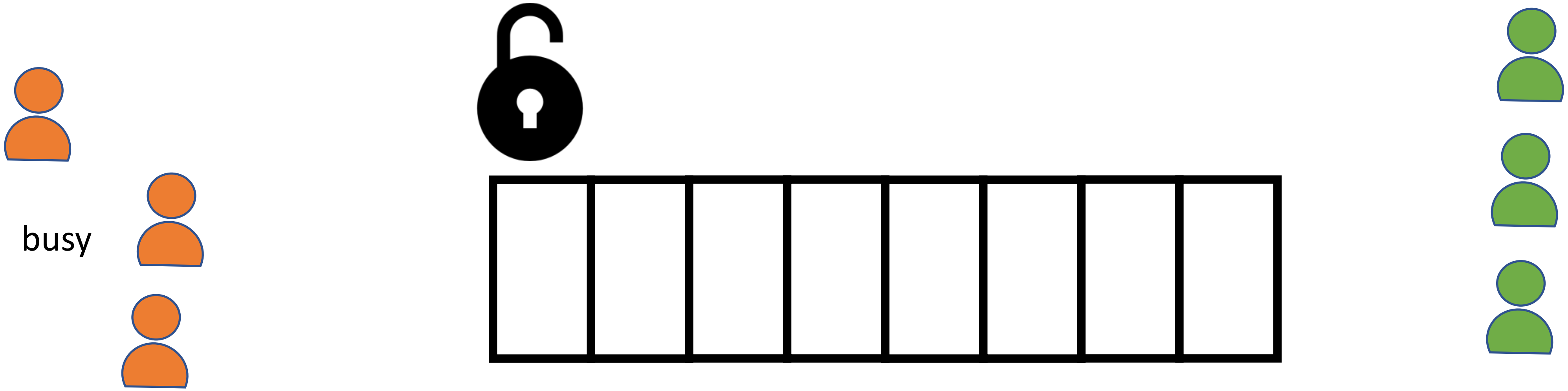
- Problem 1: Buffer is full and producer want to add entries



Packet needs to be dropped. Accessed buffer but did not produce.

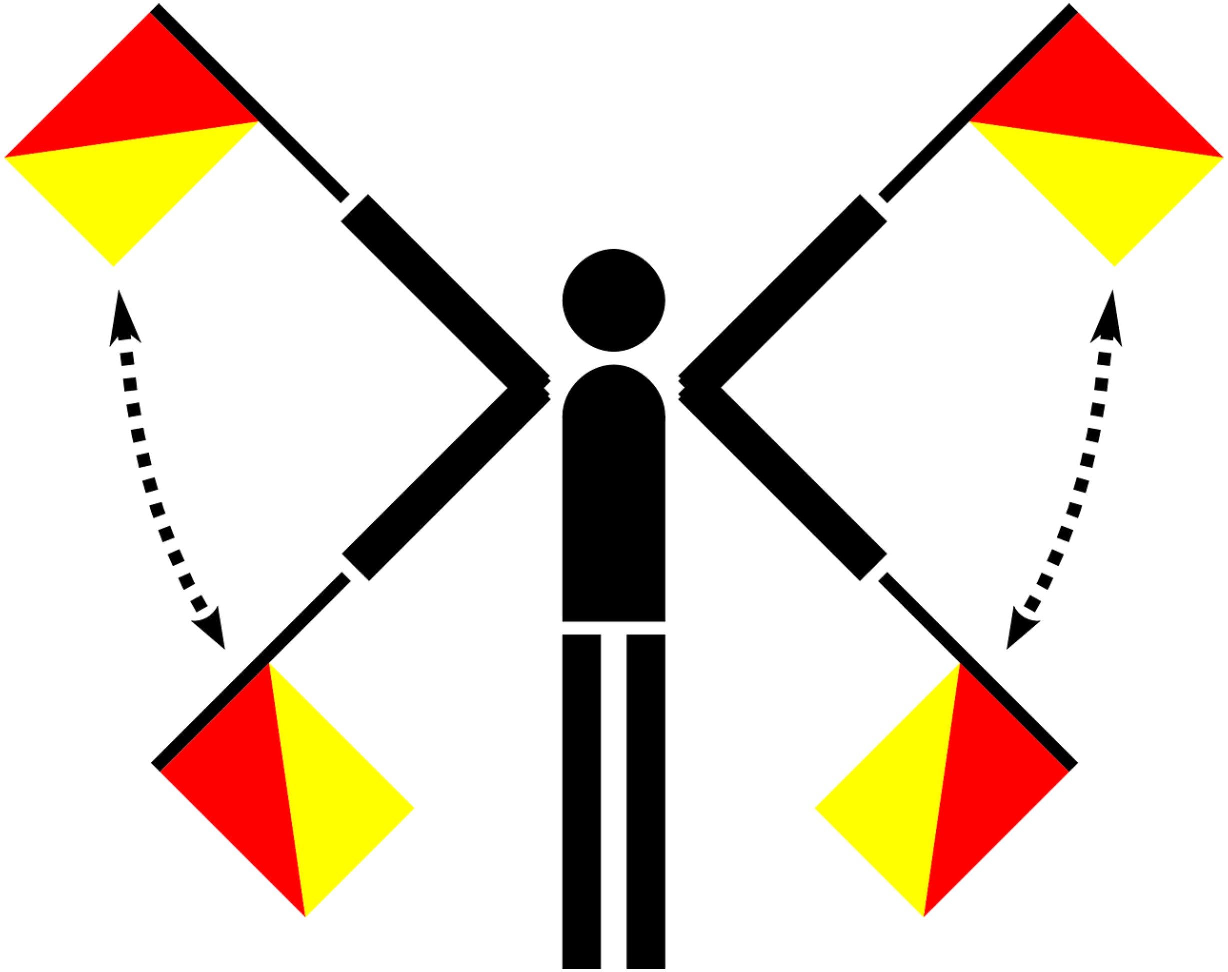
Problem: Threads can’t know the state of the buffer before acquiring the lock

- Problem 2: Similarly, buffer is empty, and consumer want to consume

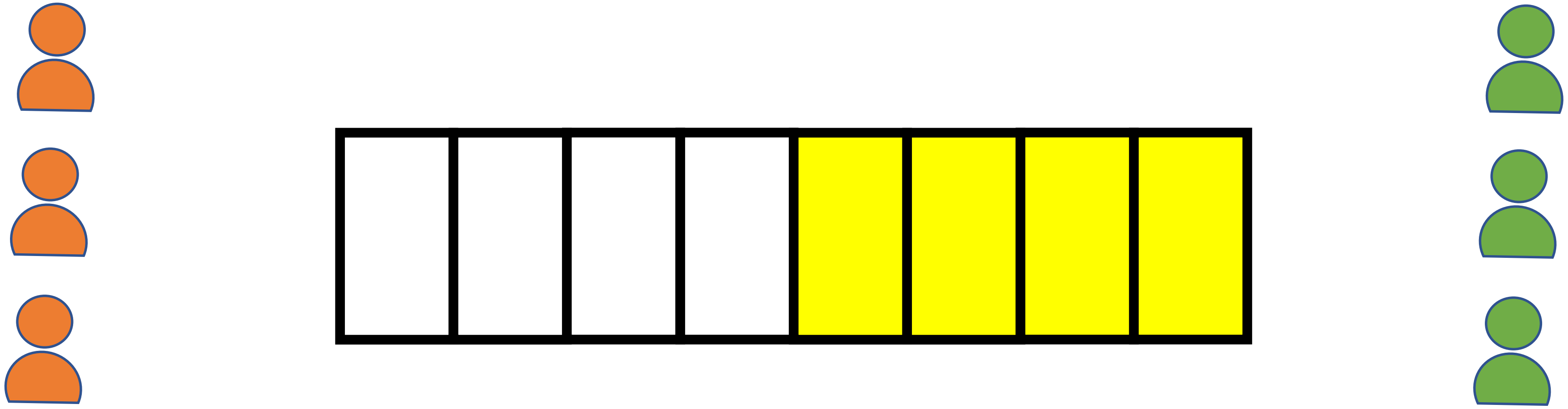


Nothing to take. Accessed buffer but did not consume.

Solution: Semaphores

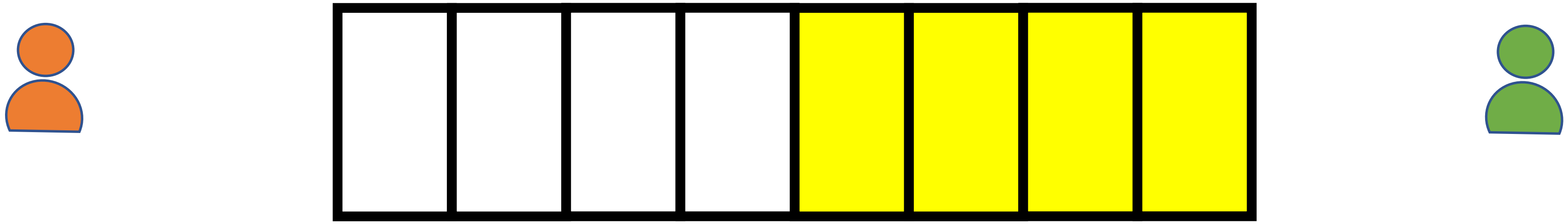


Solve Producer-Consumer with Semaphores?



Solve Producer-Consumer with Semaphores

Step 1: 1 producer, 1 consumer. Think!



Producer-Consumer: Semaphores (Step 1)

Circular Buffer, single producer thread, single consumer thread

- Shared buffer with **N** elements between producer and consumer

Requires 2 semaphores

- **emptyBuffer**: Initialize to **???**
- **fullBuffer**: Initialize to **???**

Producer-Consumer: Semaphores (Step 1)

Circular Buffer, single producer thread, single consumer thread

- Shared buffer with **N** elements between producer and consumer

Requires 2 semaphores

- **emptyBuffer**: Initialize to **N** → N empty buffers; producer can run N times first
- **fullBuffer**: Initialize to **0** → 0 full buffers; consumer can run 0 times first

Producer-Consumer: Semaphores (Step 1)

Circular Buffer, single producer thread, single consumer thread

- Shared buffer with **N** elements between producer and consumer

Requires 2 semaphores

- **emptyBuffer**: Initialize to **N** → N empty buffers; producer can run N times first
- **fullBuffer**: Initialize to **0** → 0 full buffers; consumer can run 0 times first

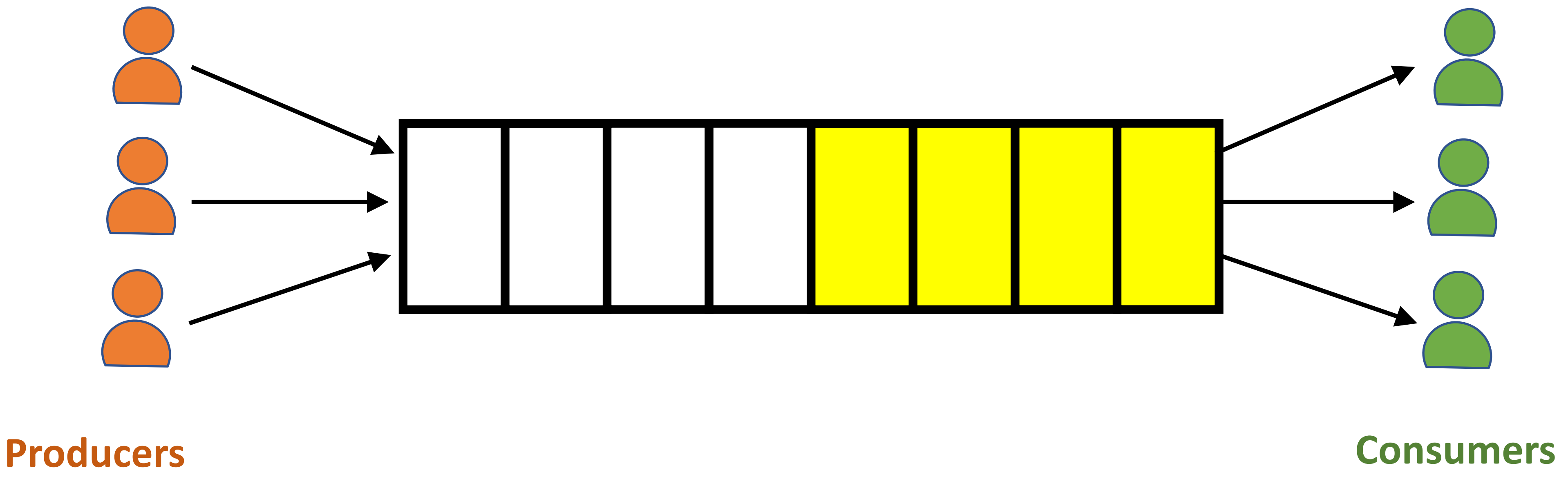
Producer

```
i = 0;
while (1) {
  wait(&emptyBuffer);
  Fill(&buffer[i]);
  i = (i+1)%N;
  post(&fullBuffer);
}
```

Consumer

```
j = 0;
while (1) {
  wait(&fullBuffer);
  Use(&buffer[j]);
  j = (j+1)%N;
  post(&emptyBuffer);
}
```

Multiple producers and multiple consumers



Producer-Consumer: Semaphore (Step 2)

Multiple producer threads, multiple consumer threads

- Shared buffer with N elements between producer and consumer

Requirements

- Each consumer must grab unique filled element
- Each producer must grab unique empty element
- **Why will previous code (shown below) not work??? Think!**

Producer

```
i = 0;
while (1) {
  wait(&emptyBuffer);
  Fill(&buffer[i]);
  i = (i+1)%N;
  post(&fullBuffer);
}
```

Consumer

```
j = 0;
while (1) {
  wait(&fullBuffer);
  Use(&buffer[j]);
  j = (j+1)%N;
  post(&emptyBuffer);
}
```

Producer-Consumer: Semaphore (Step 2)

Multiple producer threads, multiple consumer threads

- Shared buffer with N elements between producer and consumer

Requirements

- Each consumer must grab unique filled element
- Each producer must grab unique empty element
- **Why will previous code (shown below) not work???**

Producer

```
i = 0;
while (1) {
  wait(&emptyBuffer);
  Fill(&buffer[i]);
  i = (i+1)%N;
  post(&fullBuffer);
}
```

Consumer

```
j = 0;
while (1) {
  wait(&fullBuffer);
  Use(&buffer[j]);
  j = (j+1)%N;
  post(&emptyBuffer);
}
```

Are i and j private or shared? Need each producer to grab unique buffer slot

Producer-Consumer: Semaphore (Step 3)

Multiple producer threads, multiple consumer threads

- Shared buffer with N elements between producer and consumer

Requirements

- Each consumer must grab unique filled element
- Each producer must grab unique empty element
- **Why will the code below still not work??? Think!**

Producer

```
while(1){
  wait(&emptyBuffer);
  myi = findempty(&buffer);
  Fill(&buffer[myi]);
  post(&fullBuffer);
}
```

Consumer

```
while(1){
  wait(&fullBuffer);
  myj = findfull(&buffer);
  Use(&buffer[myj]);
  post(&emptyBuffer);
}
```

Producer-Consumer: Semaphore (Step 3)

Multiple producer threads, multiple consumer threads

- Shared buffer with N elements between producer and consumer

Requirements

- Each consumer must grab unique filled element
- Each producer must grab unique empty element
- **Why will the code below still not work???**

Producer

```
while(1){
  wait(&emptyBuffer);
  myi = findempty(&buffer);
  Fill(&buffer[myi]);
  post(&fullBuffer);
}
```

Consumer

```
while(1){
  wait(&fullBuffer);
  myj = findfull(&buffer);
  Use(&buffer[myj]);
  post(&emptyBuffer);
}
```

Are myi and myj private or shared? Where is mutual exclusion needed???

Producer-Consumer: Semaphore (Step 4)

Multiple producer threads, multiple consumer threads

- Consider possible locations for mutual exclusion (i.e., equivalent to semaphore initialized to 1)
- **Where is the problem with the code below???**

Producer

```
while(1){
  lock(&mutex);
  wait(&emptyBuffer);
  myi = findempty(&buffer);
  Fill(&buffer[myi]);
  post(&fullBuffer);
  unlock(&mutex);
}
```

Consumer

```
while(1){
  lock(&mutex);
  wait(&fullBuffer);
  myj = findfull(&buffer);
  Use(&buffer[myj]);
  post(&emptyBuffer);
  unlock(&mutex);
}
```

Producer-Consumer: Semaphore (Step 4)

Multiple producer threads, multiple consumer threads

- Consider possible locations for mutual exclusion (i.e., equivalent to semaphore initialized to 1)
- **Where is the problem with the code below???**

Producer

```
while(1){
  lock(&mutex);
  wait(&emptyBuffer);
  myi = findempty(&buffer);
  Fill(&buffer[myi]);
  post(&fullBuffer);
  unlock(&mutex);
}
```

Consumer

```
while(1){
  lock(&mutex);
  wait(&fullBuffer);
  myj = findfull(&buffer);
  Use(&buffer[myj]);
  post(&emptyBuffer);
  unlock(&mutex);
}
```

Problem: Deadlock at mutex (e.g., consumer runs first; won’t release mutex)

Producer-Consumer Final Solution

Multiple producer threads, multiple consumer threads

- Consider possible locations for mutual exclusion (i.e., equivalent to semaphore initialized to 1)

Producer

```
while(1){
  wait(&emptyBuffer);
  lock(&mutex);
  myi = findempty(&buffer);
  Fill(&buffer[myi]);
  unlock(&mutex);
  post(&fullBuffer);
}
```

Consumer

```
while(1){
  wait(&fullBuffer);
  lock(&mutex);
  myj = findfull(&buffer);
  Use(&buffer[myj]);
  unlock(&mutex);
  post(&emptyBuffer);
}
```

Producer-Consumer Final Solution

Multiple producer threads, multiple consumer threads

- Consider possible locations for mutual exclusion (i.e., equivalent to semaphore initialized to 1)

Producer

```
While(1){
  wait(&emptyBuffer);
  lock(&mutex);
  myi = findempty(&buffer);
  Fill(&buffer[myi]);
  unlock(&mutex);
  post(&fullBuffer);
}
```

Consumer

```
While(1){
  wait(&fullBuffer);
  lock(&mutex);
  myj = findfull(&buffer);
  Use(&buffer[myj]);
  unlock(&mutex);
  post(&emptyBuffer);
}
```

Finally, works! ☺

But limits concurrency. Only 1 thread at a time can be using or filling different buffers

Let’s practice:
Multithreaded Web Server

Let’s practice: Multithreaded Web Server

```
ListenerThread {
  forever {
    Receive(request)
    pthread_create(…)
  }
}

WorkerThread(request) {
  read file from disk
  Send(response)
  pthread_exit()
}
```

Note that clients are still in separate processes

Shared Data?

- There is none!
- Process creation serves as synchronization

Multithreaded Web Server with Thread Pool

```
ListenerThread {
  for (i=0; i<MAX_THREADS; i++) { Pthread_create(…) }
  forever {
    Receive(request)
    hand request to thread[?]
  }
}

WorkerThread[?] {
  forever {
    wait for available request
    read file from disk
    Send(reply)
  }
}
```

Shared Data?

- We need to create shared data
- Going to be some kind of queue
- Put lock/unlock around it

Multithreaded Web Server with Thread Pool (incorrect)

```
ListenerThread {
  for( i=0; i<MAX_THREADS; i++ ) thread[i] = pthread_create(…)
  forever {
    Receive(request)
    pthread_mutex_lock(queuelock)
    put request in queue
    pthread_mutex_unlock(queuelock)
  }
}

WorkerThread {
  forever {
    pthread_mutex_lock(queuelock)
    take request out of queue
    pthread_mutex_unlock(queuelock)
    read file from disk
    Send(reply)
  }
}
```

It will not work

- You need to tell worker(s) there is something for them to do (i.e., in the queue)
- Producer/consumer problem

Multithreaded Web Server with Thread Pool (incorrect)

```
ListenerThread {
  for( i=0; i<MAX_THREADS; i++ ) thread[i] = pthread_create(...)
  forever {
    Receive(request)
    pthread_mutex_lock(queuelock)
    put request in queue
    pthread_cond_signal(notempty)
    pthread_mutex_unlock(queuelock)
  }
}

WorkerThread {
  forever {
    pthread_mutex_lock(queuelock)
    pthread_cond_wait(notempty, queuelock)
    take request out of queue
    pthread_mutex_unlock(queuelock)
    read file from disk
    Send(reply)
  }
}
```

Incorrect

- All worker threads busy (none waiting)
- Listener does a signal
- No thread waiting: signal is no-op
- Worker thread finishes what it was doing
  - Will do a wait
  - Although request is waiting in queue

In General

- Signals have no memory
- Forgotten if no thread waiting
- So need an extra variable to remember them

Multithreaded Web Server with Thread Pool

```
ListenerThread {
  for( i=0; i<MAX_THREADS; i++ ) thread[i] = pthread_create(…)
  forever {
    Receive( request )
    pthread_mutex_lock( queuelock )
    put request in queue
    avail++
    pthread_cond_signal( notempty )
    pthread_mutex_unlock( queuelock )
  }
}
WorkerThread {
  forever {
    pthread_mutex_lock( queuelock )
    if( avail <= 0 ) pthread_cond_wait( notempty, queuelock )
    take request out of queue
    avail--
    pthread_mutex_unlock( queuelock )
    read file from disk
    Send( reply )
  }
}
```

Note

- Should now be clear why mutex must be held
- Avail is a shared data item
- Without mutex could have data race

Imagine Solution Without Locks

```
ListenerThread {
  for( i=0; i<MAX_THREADS; i++ ) thread[i] = pthread_create(…)
  forever {
    Receive( request )
    ~~pthread_mutex_lock( queuelock )~~
    put request in queue
    avail++
    pthread_cond_signal( notempty )
    ~~pthread_mutex_unlock( queuelock )~~
  }
}
WorkerThread {
  forever {
    ~~pthread_mutex_lock( queuelock )~~
    if( avail <= 0 ) pthread_cond_wait( notempty, queuelock )
    take request out of queue
    avail--
    ~~pthread_mutex_unlock( queuelock )~~
    read file from disk
    Send( reply )
  }
}
```

Imagine Solution Without Locks

```
ListenerThread {
 for( i=0; i<MAX_THREADS; i++ ) thread[i] = pthread_create(…)
 forever {
   Receive( request )
   pthread_mutex_lock( queuelock )
   put request in queue
   avail++
   pthread_cond_signal( notempty )
   pthread_mutex_unlock( queuelock )
 }
}
WorkerThread {
 forever {
   pthread_mutex_lock( queuelock )
   if( avail <= 0 ) pthread_cond_wait( notempty, queuelock )
   take request out of queue
   avail--
   pthread_mutex_unlock( queuelock )
   read file from disk
   Send( reply )
 }
}
```

Worker checks avail and finds it to be 0

Worker interrupted and listener runs

Listener sets avail to 1 and signals

No thread is waiting, so signal is no-op

Listener interrupted and worker runs

Worker does a wait

Incorrect: worker waits with request in queue

Example incorrect execution: One Worker Thread

- Worker checks avail and finds it to be 0
- Worker interrupted and listener runs
- Listener sets avail to 1 and signals
- No thread is waiting, so signal is no-op
- Listener interrupted and worker runs
- Worker does a wait
- Incorrect: worker waits with request in queue

Back to Solution with Locks (still incorrect)

```
ListenerThread {
 for( i=0; i<MAX_THREADS; i++ ) thread[i] = pthread_create(…)
 forever {
  Receive( request )
  pthread_mutex_lock( queuelock )
  put request in queue
  avail++
  pthread_cond_signal( notempty )
  pthread_mutex_unlock( queuelock )
 }
}
WorkerThread {
 forever {
  pthread_mutex_lock( queuelock )
  if( avail <= 0 ) pthread_cond_wait( notempty, queuelock )
  take request out of queue
  avail--
  pthread_mutex_unlock( queuelock )
  read file from disk
  Send( reply )
 }
}
```

Back to Solution with Locks (still incorrect)

```
ListenerThread {
  for( i=0; i<MAX_THREADS; i++ ) thread[i] = pthread_create(…)
  forever {
    Receive( request )
    pthread_mutex_lock( queuelock )
    put request in queue
    avail++
    pthread_cond_signal( notempty )
    pthread_mutex_unlock( queuelock )
  }
}
WorkerThread {
  forever {
    pthread_mutex_lock( queuelock )
    if( avail <= 0 ) pthread_cond_wait( notempty, queuelock )
    take request out of queue
    avail--
    pthread_mutex_unlock( queuelock )
    read file from disk
    Send( reply )
  }
}
```

Can this execution happen?

Queue is empty, worker thread W1 waits

Listener thread L puts request in queue

- Sets avail to 1
- Signals
- W1 is unblocked but does not run

Worker thread W2 runs and takes something out of queue

- Sets avail to 0

Now W1 runs

- Sets avail to -1??

Can this execution happen?

- Queue is empty, worker thread W1 waits
- Listener thread L puts request in queue
  - Sets avail to 1
  - Signals
  - W1 is unblocked
- Worker thread W2 runs and takes something out of queue
  - Sets avail to 0
- Now W1 runs
  - Sets avail to -1??

Answer: Yes
Remember pthreads Condition Variables

- Pthread_cond_wait(cond, mutex)
  - Wait for a signal on cond
  - Release mutex
  - Block
  - When woken, pthread_mutex_lock(mutex), then return
- Pthread_cond_signal(cond)
  - Can’t interact with mutex – mutex is not given as arg!
  - Therefore, **cannot** give the mutex to woken thread
- Pthread_cond_broadcast(cond)

So Then Why Is It Possible?

- Great question! It’s subtle.

- **Only one thread can ever hold the mutex at a time.**
- This is a very important concept that will probably appear on a test.
- Let’s see how pthreads actually maintains that invariant.

So Then Why Is It Possible?

- pthread_cond_wait(&cond, &mutex)
  - Release mutex + block for signal on cond, *atomically*
  - When woken up **and run**, reacquire mutex **as if by pthread_mutex_lock**
    - Might cause thread to block again, but *not* on cond. Just on mutex.
  - pthread_cond_wait only returns once the mutex is reacquired.

- Next time scheduled may also be a **long** time after the signal.

So Then Why Is It Possible?

- pthread_cond_signal(&cond)
  - If any thread is waiting on cond, wakeup one at random.
    - This is the **ONLY** effect of signal!
  - No knowledge of mutex associated with cond.
  - Owner of the mutex is **not modified at all.**
- Let's look at the question again…

The Question, Precisely

```
ListenerThread {
  for (i=0; i<MAX_THREADS; i++) thread[i] = pthread_create(…)
  forever {
    Receive(request)
    pthread_mutex_lock(queuelock)              // A
    put request in queue
    avail++                                    // B
    pthread_cond_signal(notempty)              // C
    pthread_mutex_unlock(queuelock)            // D
  }
}
WorkerThread {
  forever {
    pthread_mutex_lock(queuelock)              // U
    if (avail <= 0)                            // V
      pthread_cond_wait(notempty, queuelock)   // W,X
    take request out of queue
    avail--                                    // Y
    pthread_mutex_unlock(queuelock)            // Z
    read file from disk
    Send( reply )
  }
}
```

Note: W is release mutex + block, X is reacquire mutex
Both are part of wait()!

So what about this order?

L:				A	B	C	D							
W1:	U	V	W									X	Y	Z
W2:								U	V	Y	Z			

The Question, Precisely

```
ListenerThread {
  for (i=0; i<MAX_THREADS; i++) thread[i] = pthread_create(…)
  forever {
    Receive(request)
    pthread_mutex_lock(queuelock)               // A
    put request in queue
    avail++                                     // B
    pthread_cond_signal(notempty)               // C
    pthread_mutex_unlock(queuelock)             // D
  }
}
WorkerThread {
  forever {
    pthread_mutex_lock(queuelock)               // U
    if (avail <= 0)                             // V
      pthread_cond_wait(notempty, queuelock)    // W,X
    take request out of queue
    avail--                                     // Y
    pthread_mutex_unlock(queuelock)             // Z
    read file from disk
    Send( reply )
  }
}
```

Note: W is release mutex + block, X is reacquire mutex

So what about this order?

Wakeup W1

avail = -1 ???

L:				A	B	C	D							
W1:	U	V	W									X	Y	Z
W2:								U	V	Y	Z			

W1 does not reacquire queuelock until here

Oh no!

```
ListenerThread {
 for (i=0; i<MAX_THREADS; i++) thread[i] = pthread_create(…)
 forever {
  Receive(request)
  pthread_mutex_lock(queuelock)                  // A
  put request in queue
  avail++                                        // B
  pthread_cond_signal(notempty)                  // C
  pthread_mutex_unlock(queuelock)                // D
 }
}
WorkerThread {
 forever {
  pthread_mutex_lock(queuelock)                  // U
  if (avail <= 0)                                // V
   pthread_cond_wait(notempty, queuelock)        // W,X
  take request out of queue
  avail--                                        // Y
  pthread_mutex_unlock(queuelock)                // Z
  read file from disk
  Send( reply )
 }
}
```

Note: W is release mutex + block, X is reacquire mutex

So what about this order?

L:				A	B	C	D						
W1:	U	V	W								X	Y	Z
W2:								U	V	Y	Z		

Only one thread ever holds the mutex at once ✓

This **is** possible, and it’s a problem!

We Can Now Rest Easy

```
ListenerThread {
  for (i=0; i<MAX_THREADS; i++) thread[i] = pthread_create(...)
  forever {
    Receive(request)
    pthread_mutex_lock(queuelock)              // A
    put request in queue
    avail++                                    // B
    pthread_cond_signal(notempty)              // C
    pthread_mutex_unlock(queuelock)            // D
  }
}
WorkerThread {
  forever {
    pthread_mutex_lock(queuelock)              // U
    while (avail <= 0)                         // V
      pthread_cond_wait(notempty, queuelock)   // W,X
    take request out of queue
    avail--                                    // Y
    pthread_mutex_unlock(queuelock)            // Z
    read file from disk
    Send( reply )
  }
}
```

Note: W is release mutex + block, X is reacquire mutex

So what about this order?

L:				A	B	C	D						
W1:	U	V	W								X	V	W
W2:								U	V	Y	Z		

Fix: if → while

This is another reason we should **always** check condition in a while loop.

Final Solution with Locks (correct)

```
ListenerThread {
  for( i=0; i<MAX_THREADS; i++ ) thread[i] = pthread_create(…)
  forever {
    Receive( request )
    pthread_mutex_lock( queuelock )
    put request in queue
    avail++
    pthread_cond_signal( notempty )
    pthread_mutex_unlock( queuelock )
  }
}
WorkerThread {
  forever {
    pthread_mutex_lock( queuelock )
    while( avail <= 0 ) pthread_cond_wait( notempty, queuelock )
    take request out of queue
    avail--
    pthread_mutex_unlock( queuelock )
    read file from disk
    Send( reply )
  }
}
```

We've just re-implemented a semaphore!

Summary

- Multithreading techniques
  - Division of work
  - Synchronization of shared data
  - Fine-grain locking
  - Privatization
  - Producer/consumer problem

Goals for Next Part

- 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 ~= unordered 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 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
  while (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
  while (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
  while (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
  while (done == 0) {          // x
    Cond_wait(&c, &m);         // y
  }
  Mutex_unlock(&m);            // z
}
```

Child:

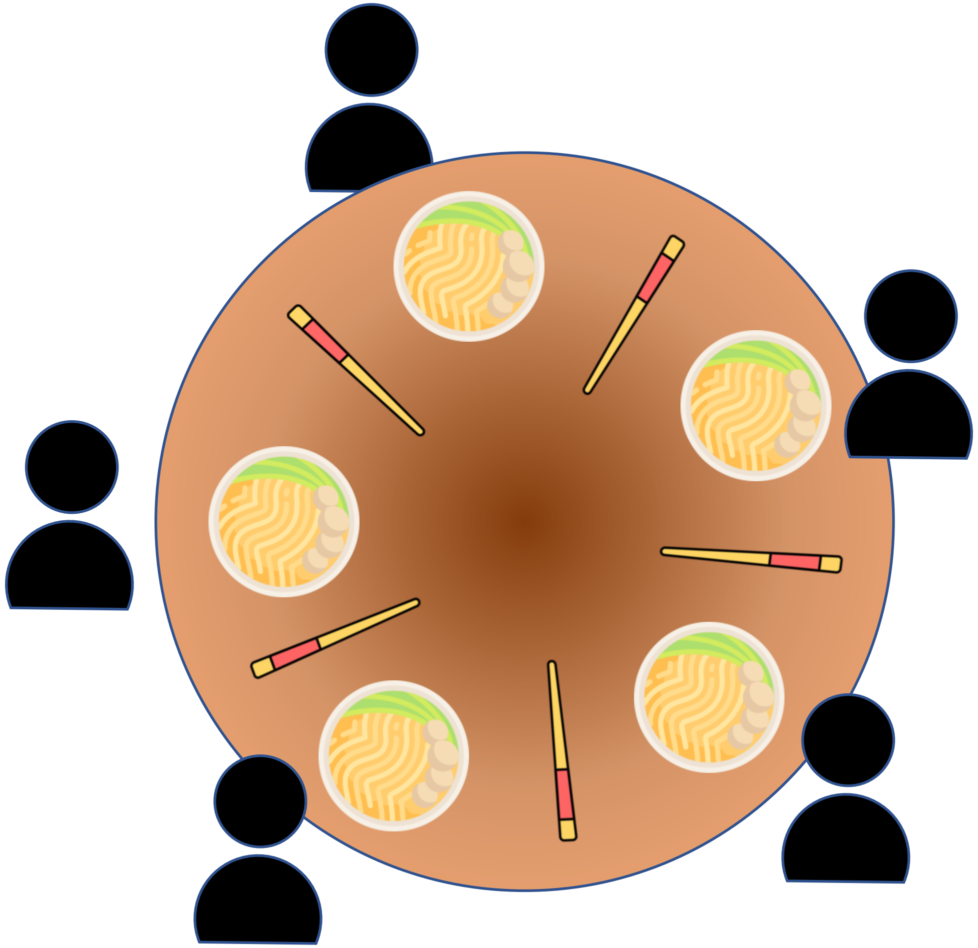
```
void thread_exit() {
  Mutex_lock(&m);              // a
  done = 1;                    // b
  Cond_signal(&c);             // c
  Mutex_unlock(&m);            // d
}
```

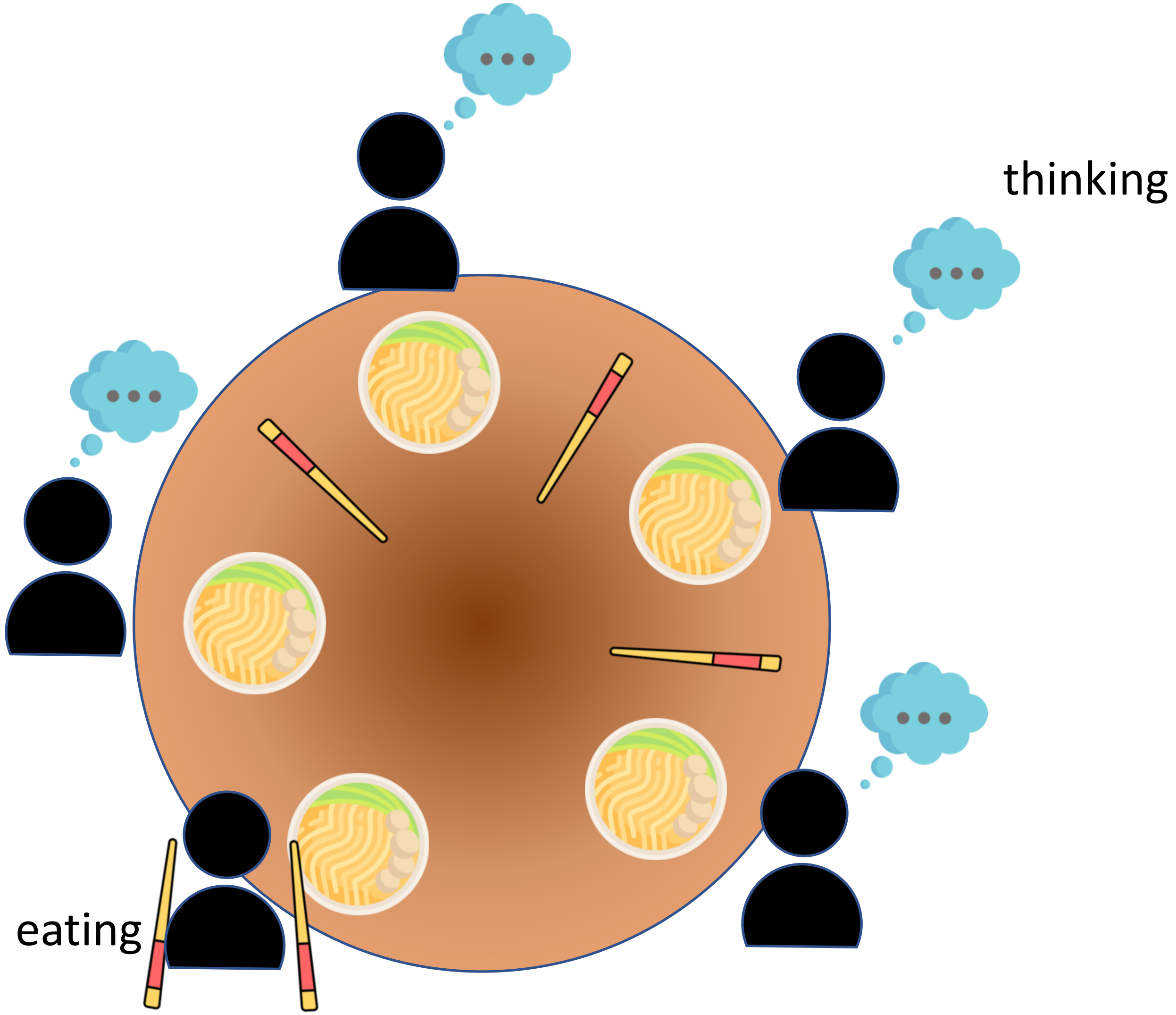
Parent:	w	x	y				z
Child:				a	b	c	

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

Problem 2: Dining Philosophers

- Classic problem in synchronization
  - Dijkstra ’71

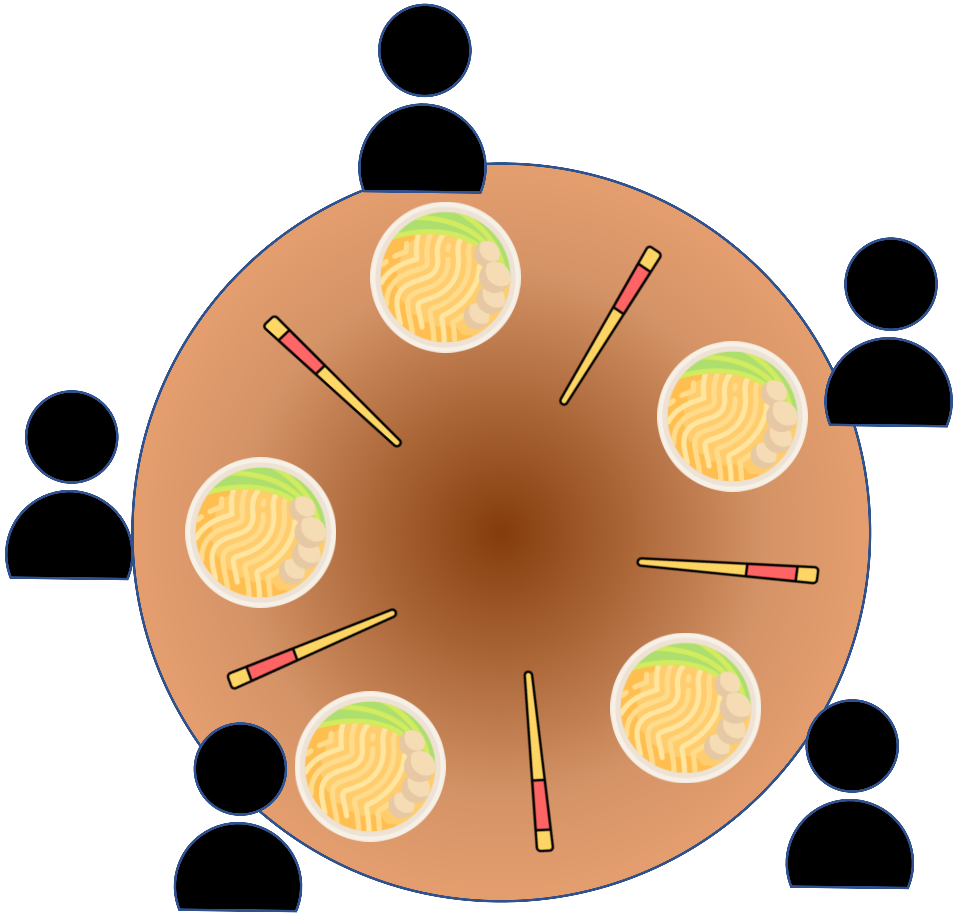




Each philosopher has 2 states:

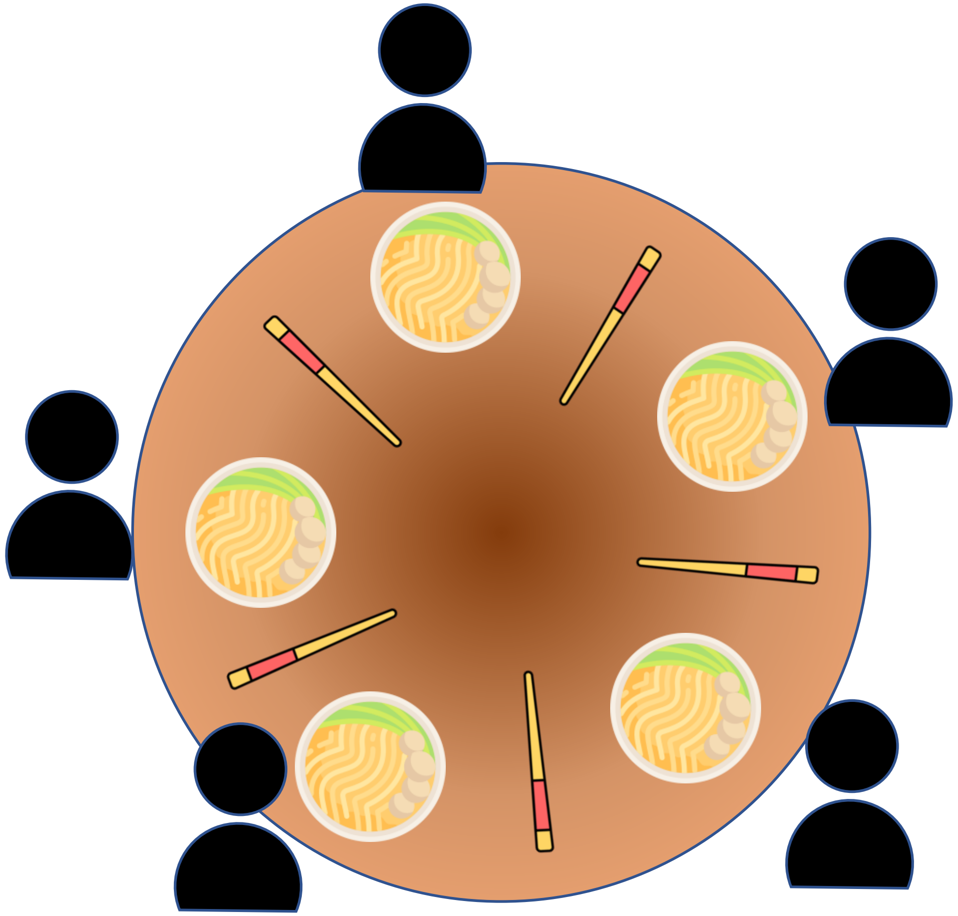
- **Eating**
  - Need 2 chopsticks to eat
- **Thinking**

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



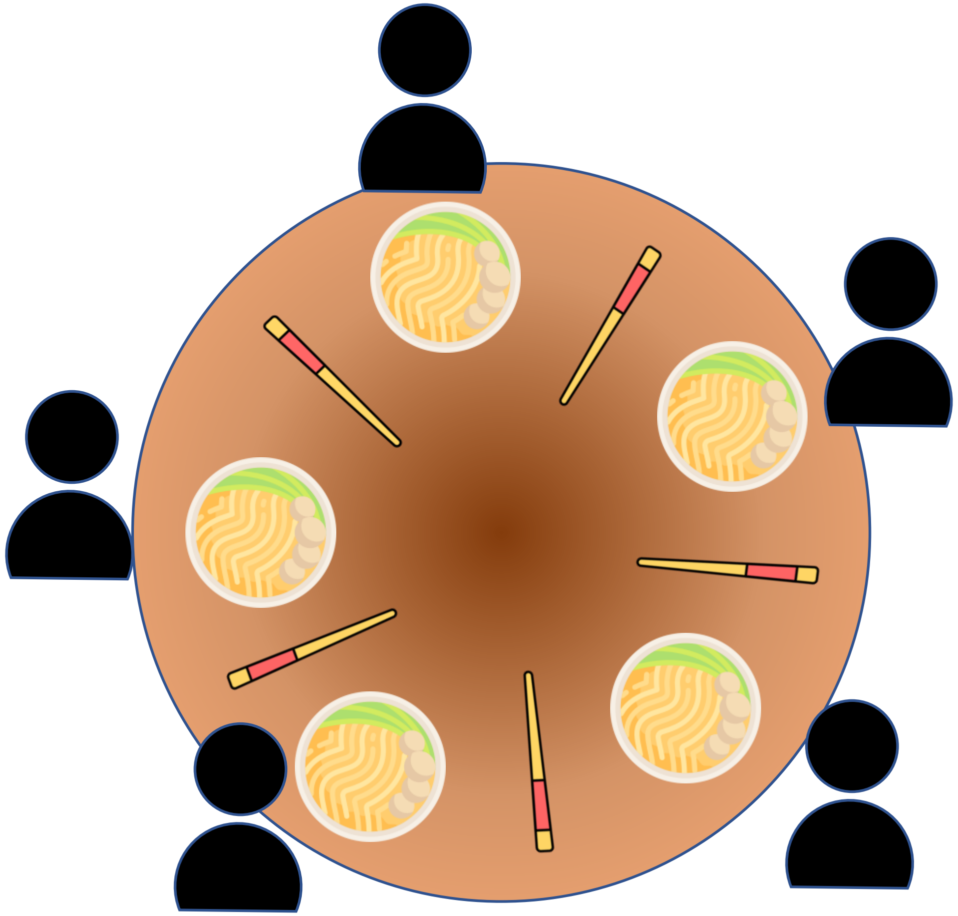
Diner rules

- 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)

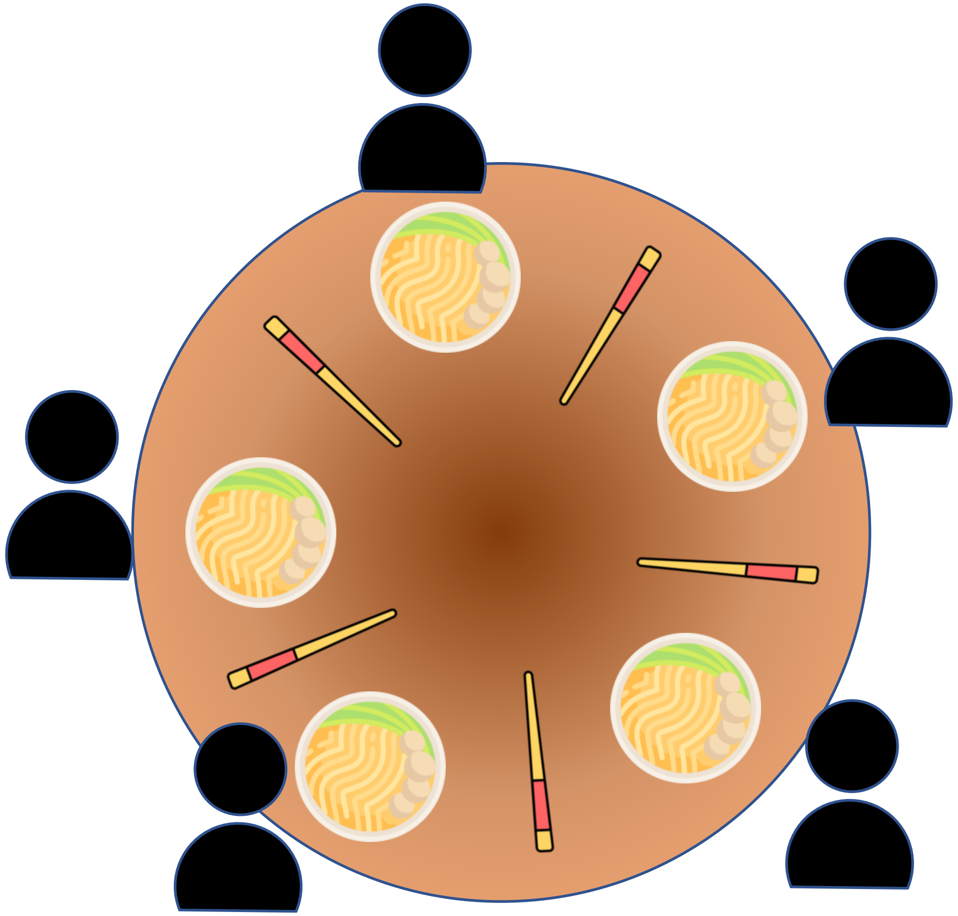
Dinner rules

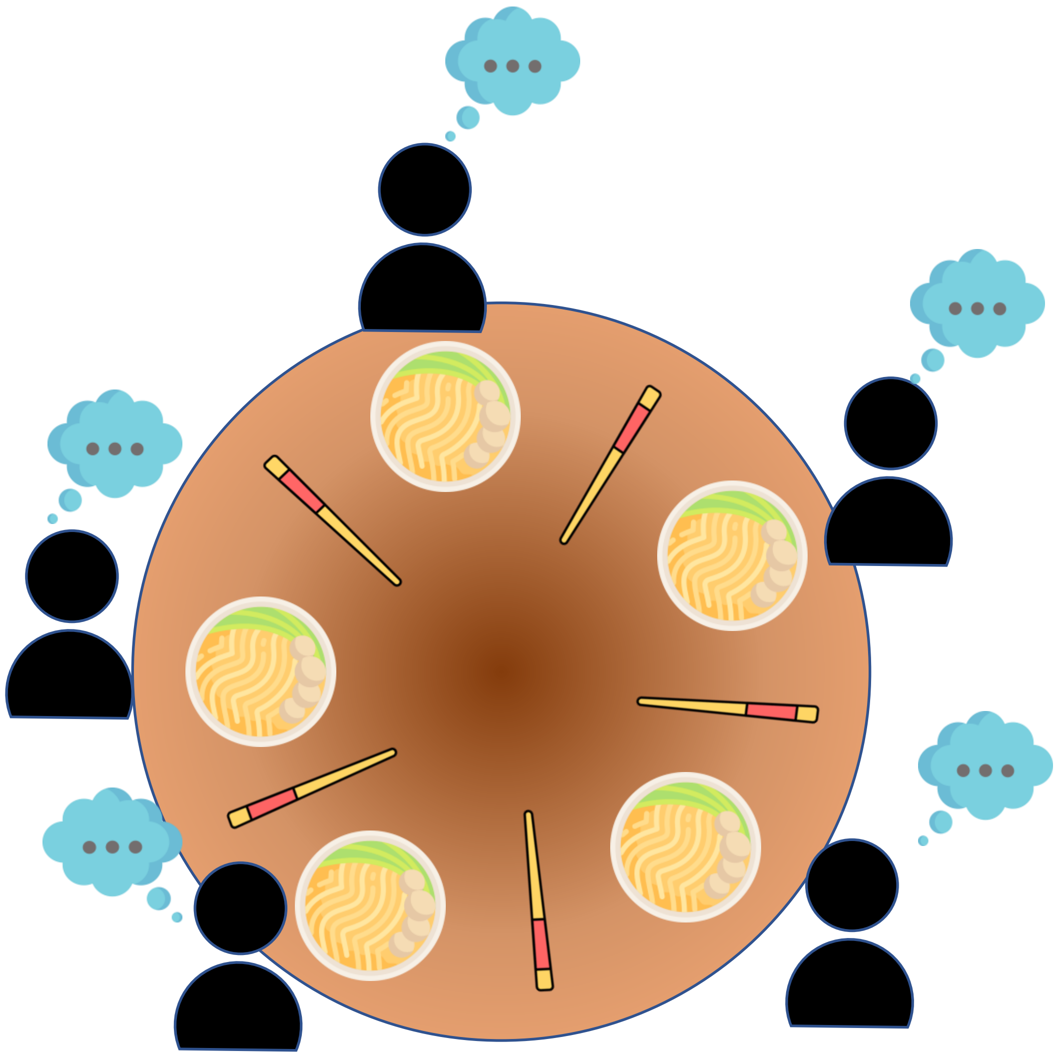
- Philosophers can't speak to each other.
- Philosophers can only pick up one chopstick at a time.
- Philosophers can only pick up adjacent chopsticks.
- Infinite food supply.

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.

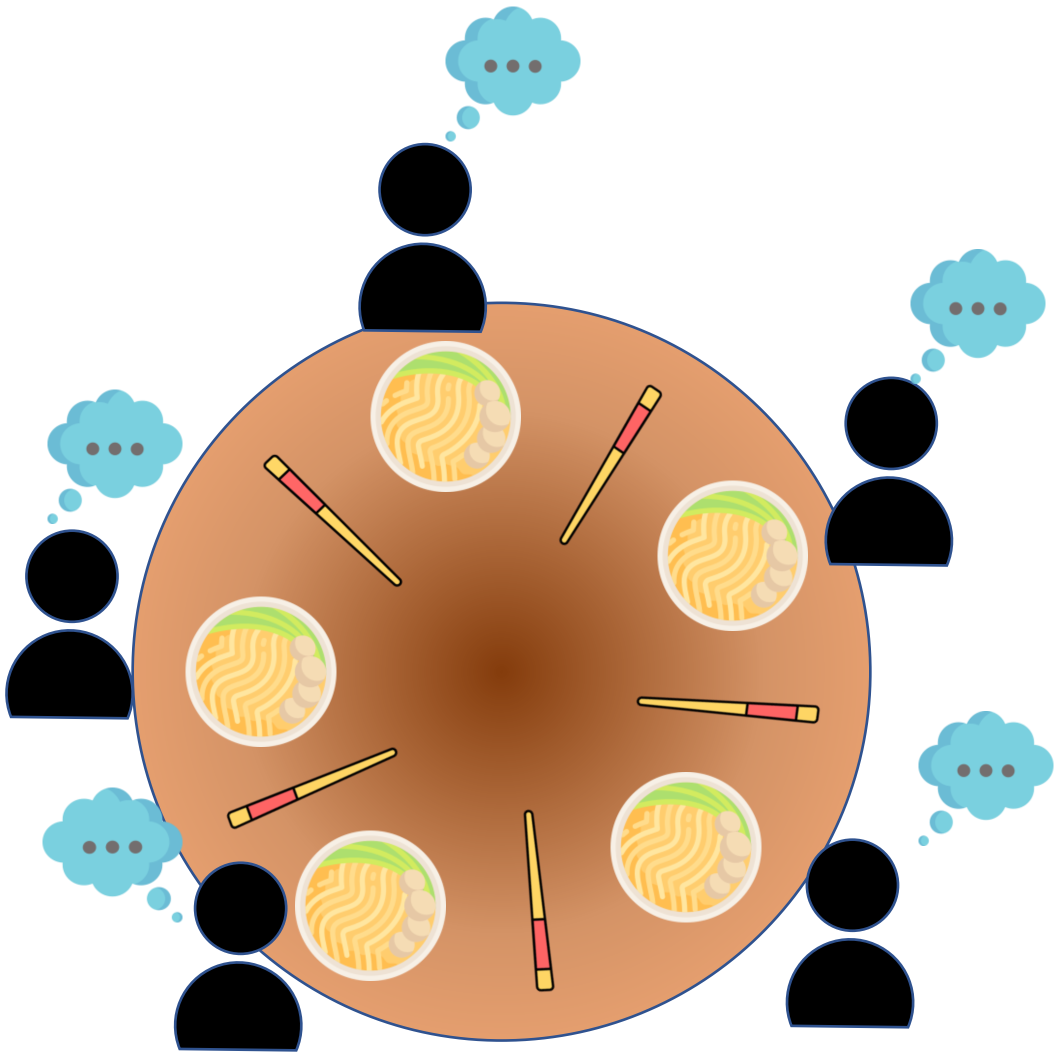
Brainstorming Slide





A simple solution (incorrect)

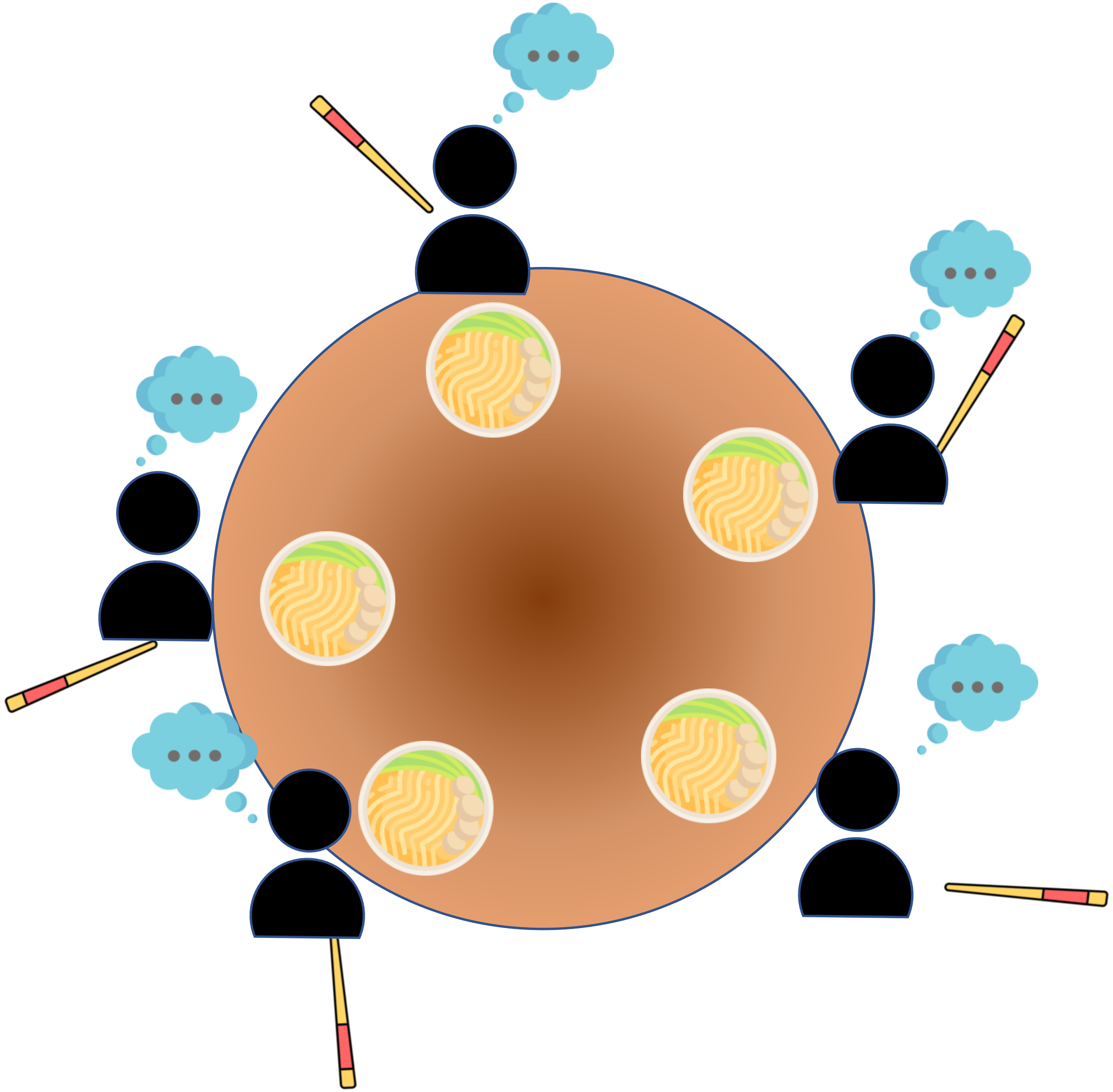
```
do forever{
 think()
 grab(chopstick_R)
 grab(chopstick_L)
 eat()
 releaseChopsticks()
}
```



A simple solution (incorrect)

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

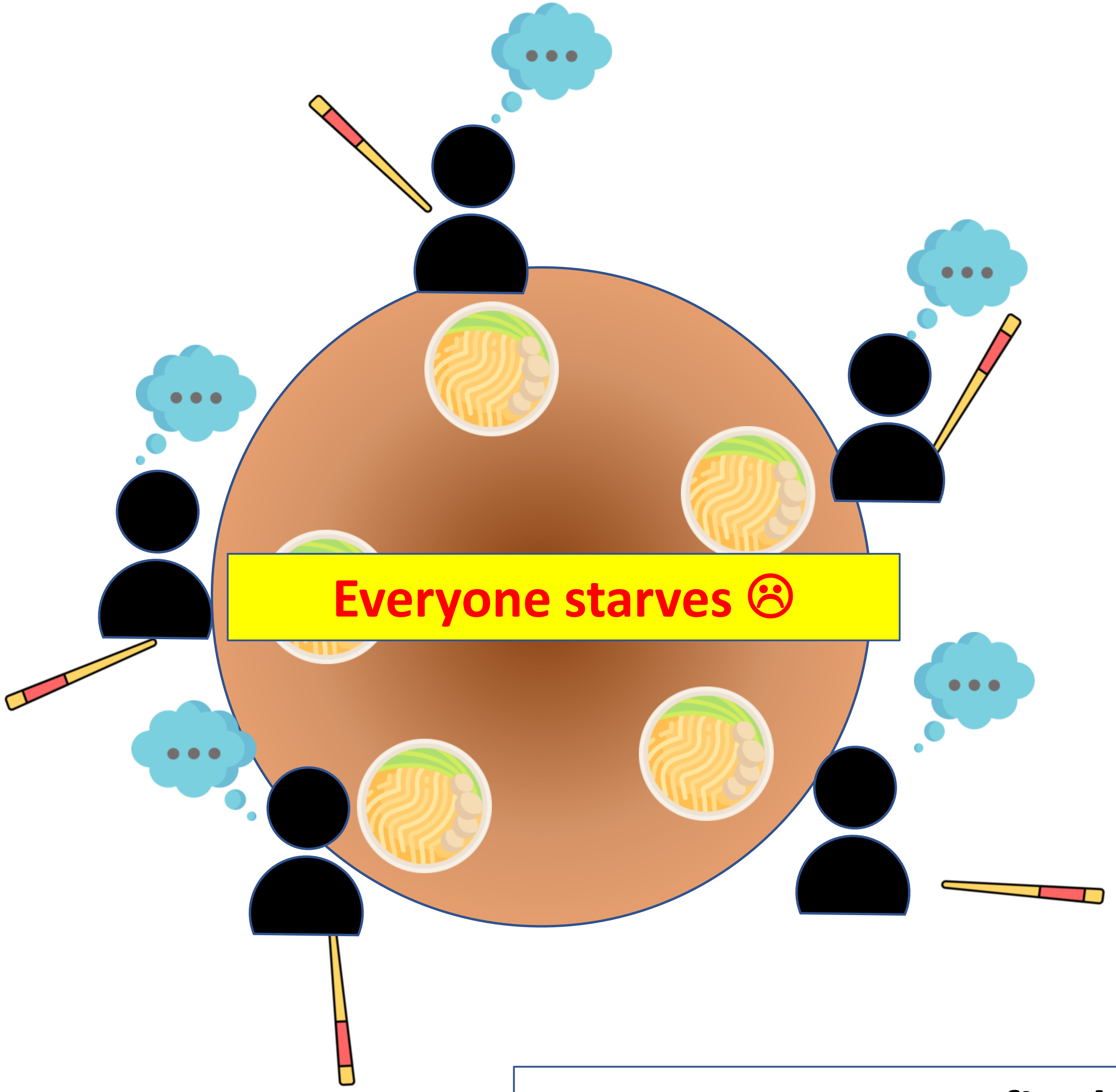
Deadlock Danger!



A simple solution (incorrect)

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

Deadlock Danger!

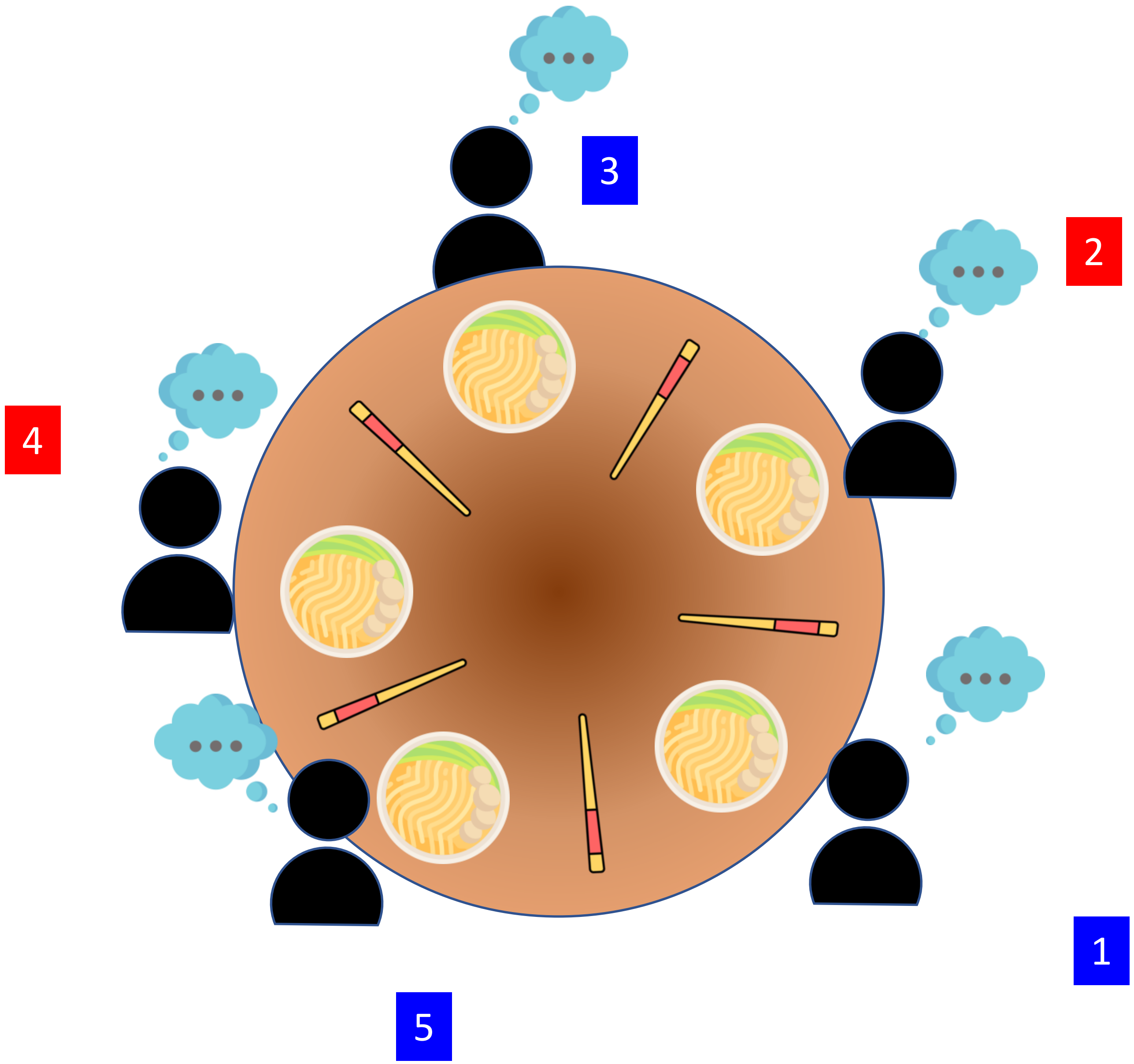


A simple solution (incorrect)

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

Deadlock Danger!

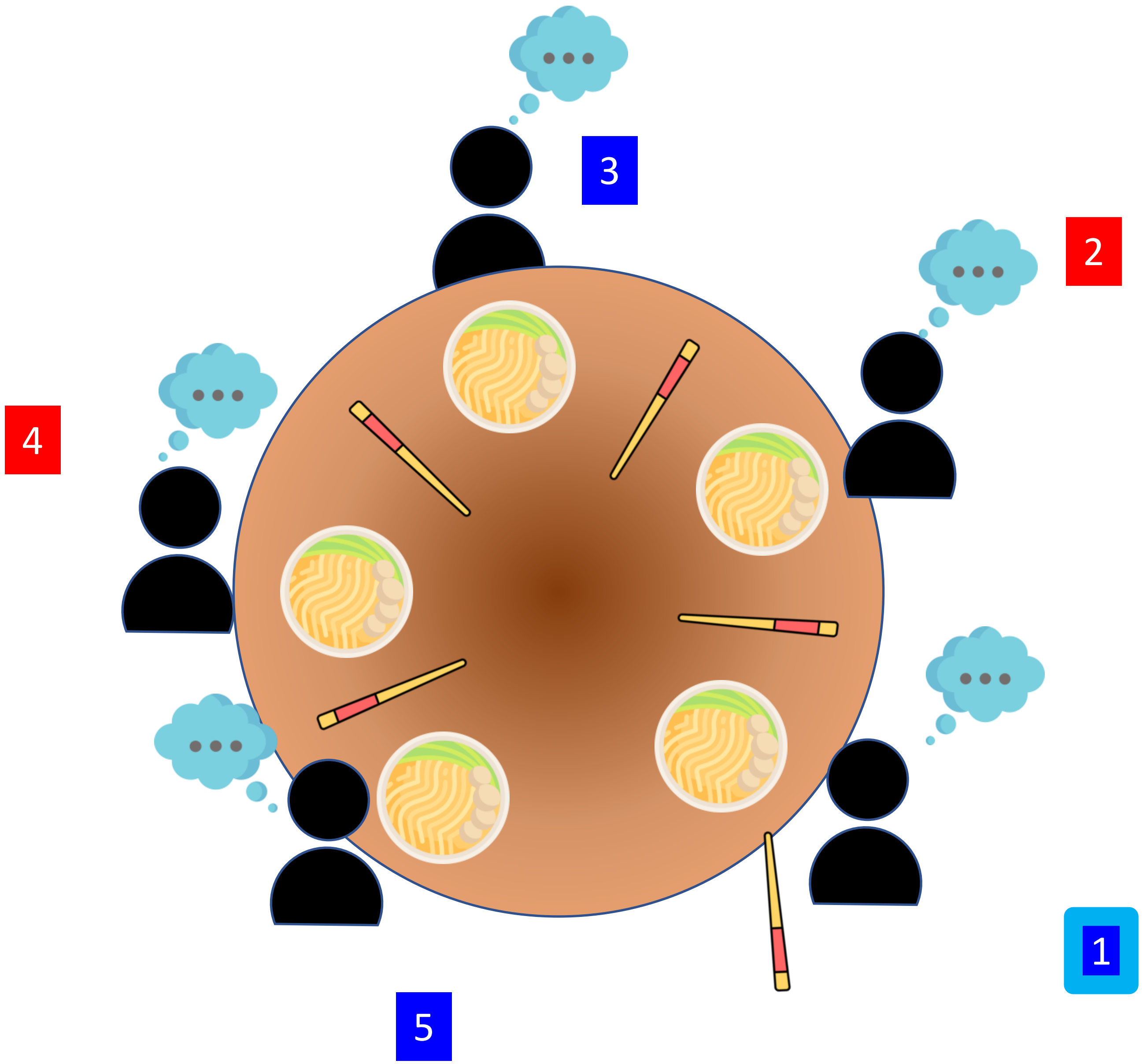
How can we fix this?



Solution 1

Asymmetric algorithm

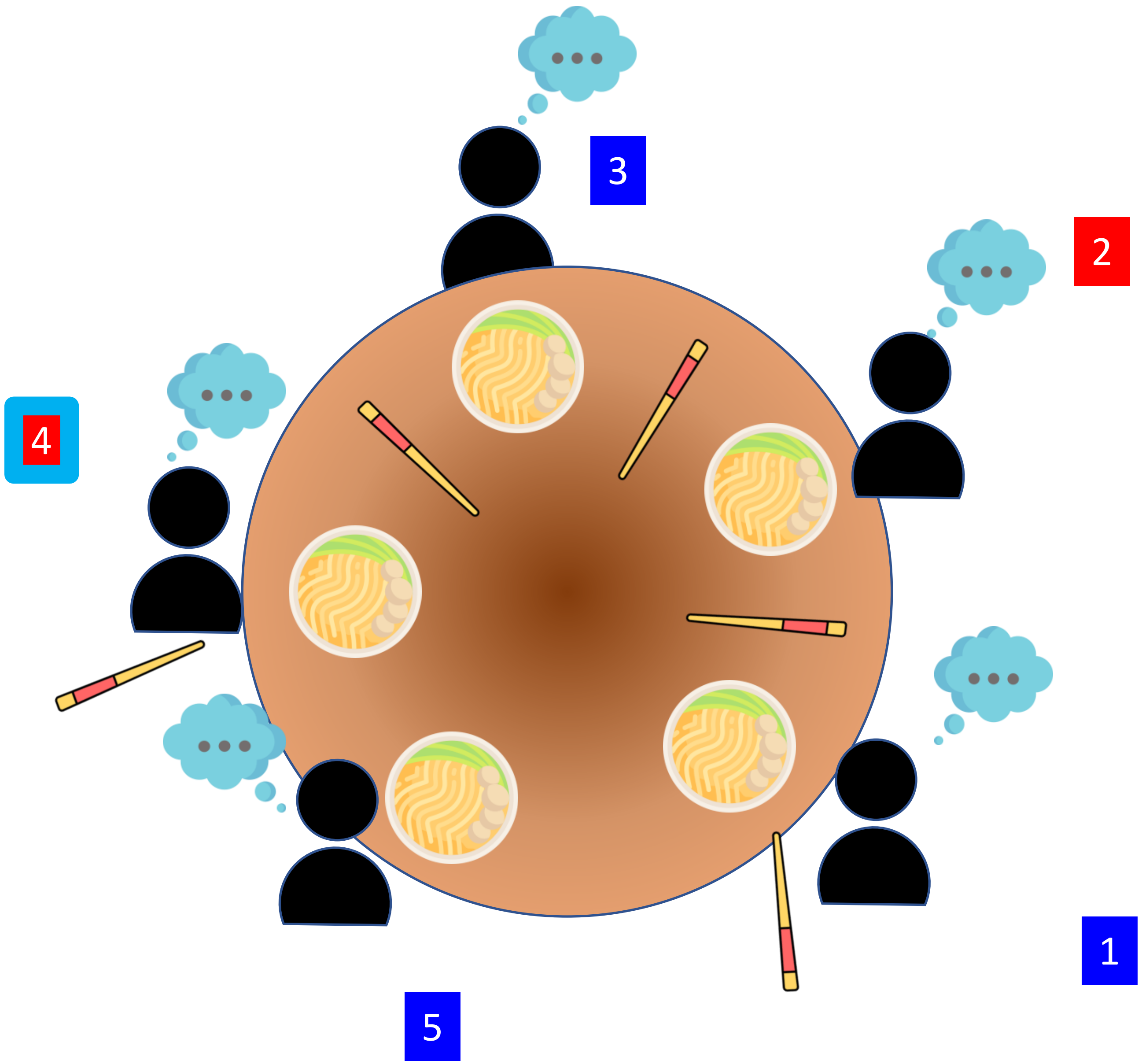
- Assign odd and even IDs to philosophers.
- Odd philosophers pick up left chopstick, then right.
- Even philosophers pick up right chopstick, then left.



Solution 1

Asymmetric algorithm

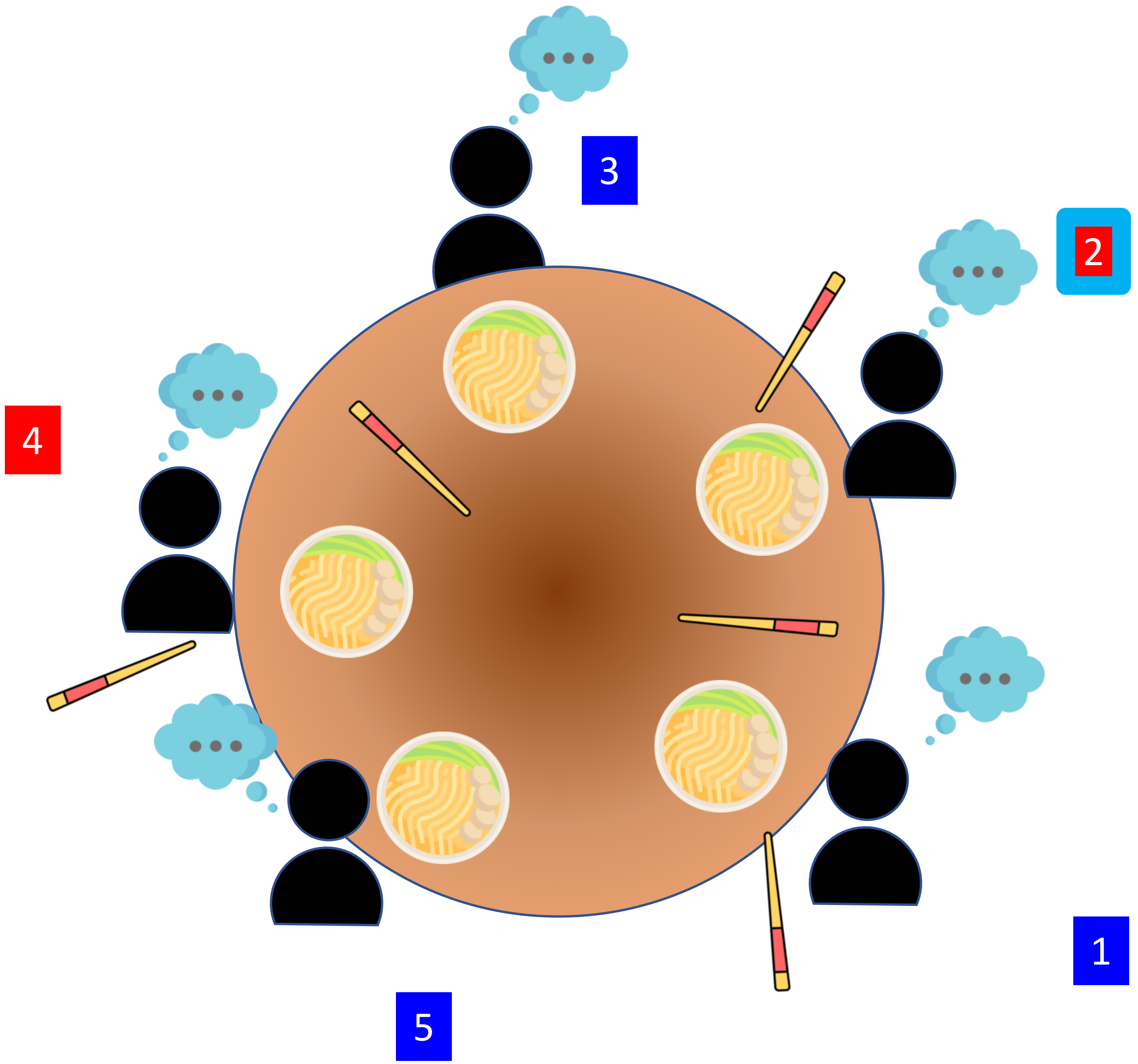
- Assign odd and even IDs to philosophers.
- Odd philosophers pick up left chopstick, then right.
- Even philosophers pick up right chopstick, then left.



Solution 1

Asymmetric algorithm

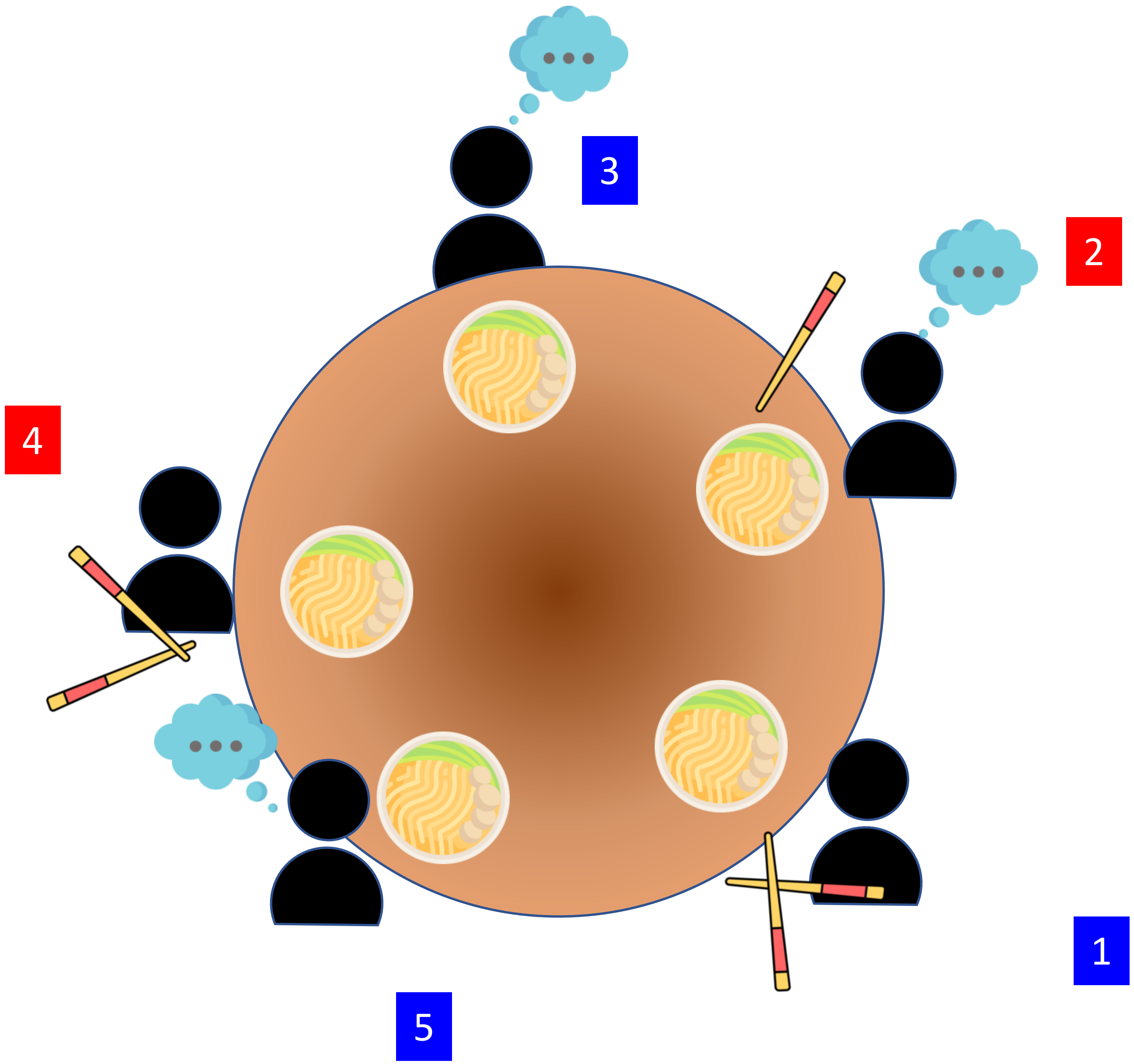
- Assign odd and even IDs to philosophers.
- Odd philosophers pick up left chopstick, then right.
- Even philosophers pick up right chopstick, then left.



Solution 1

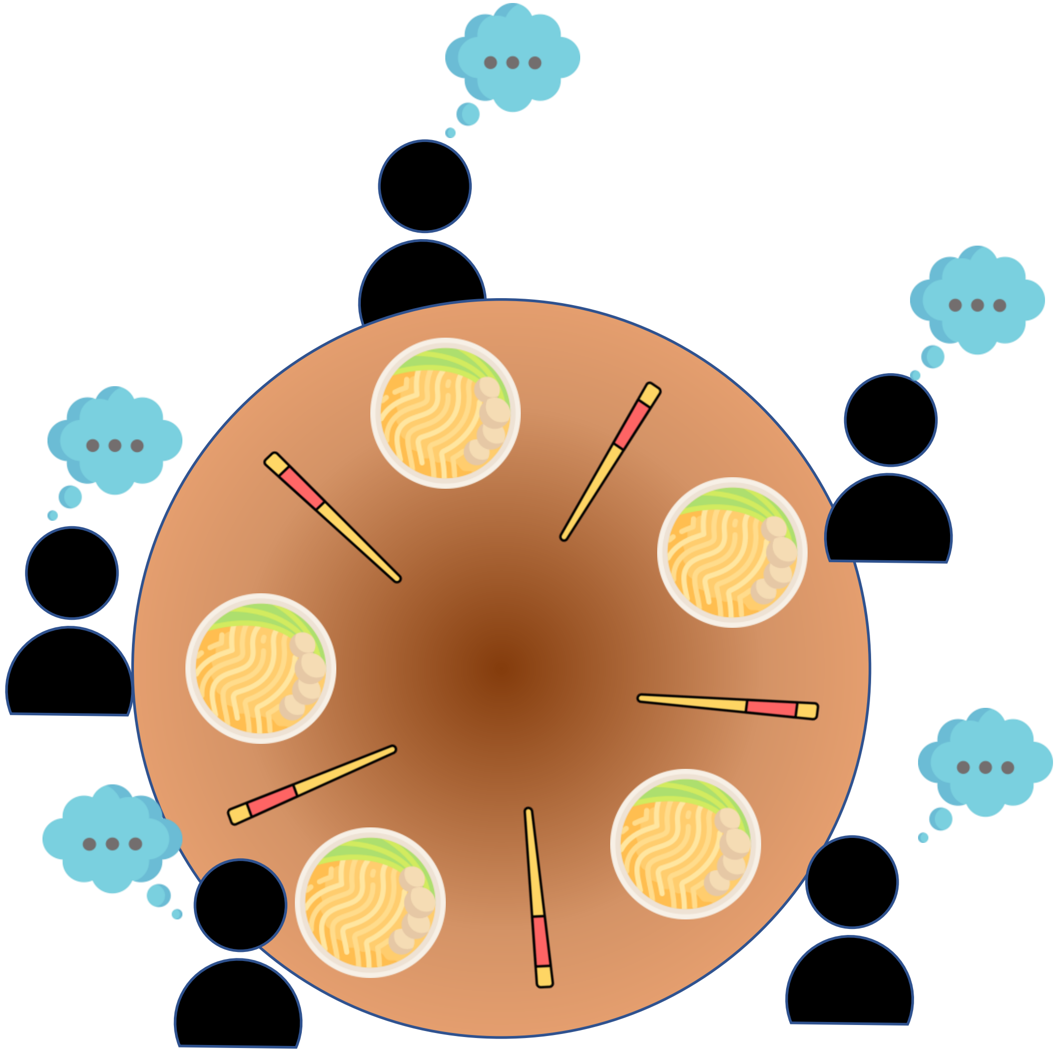
Asymmetric algorithm

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

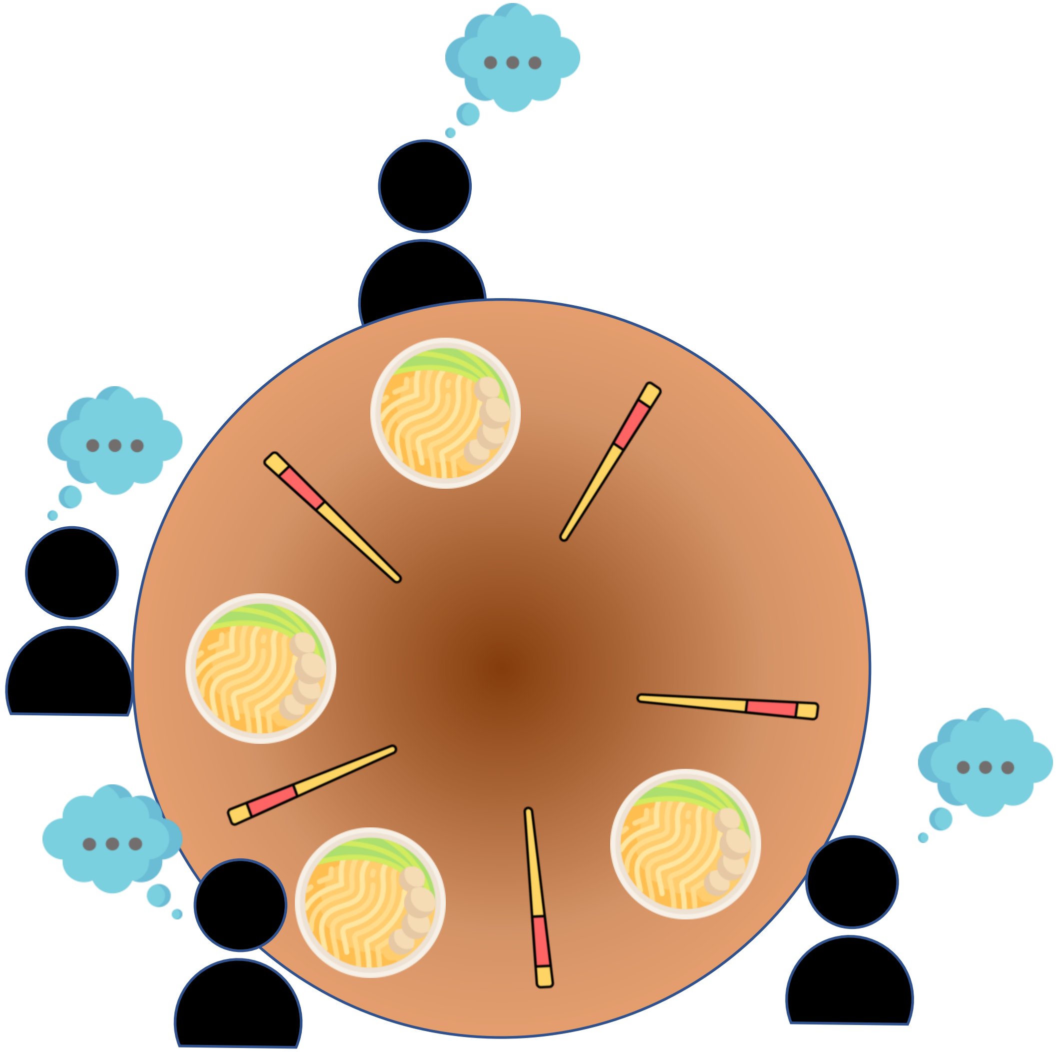


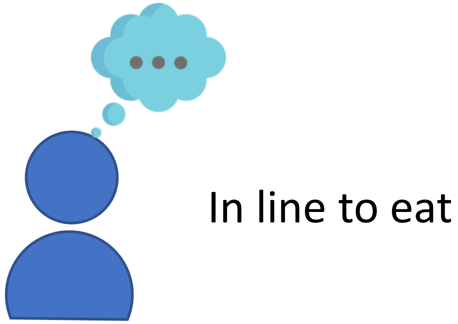
Solution 1
Asymmetric algorithm

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

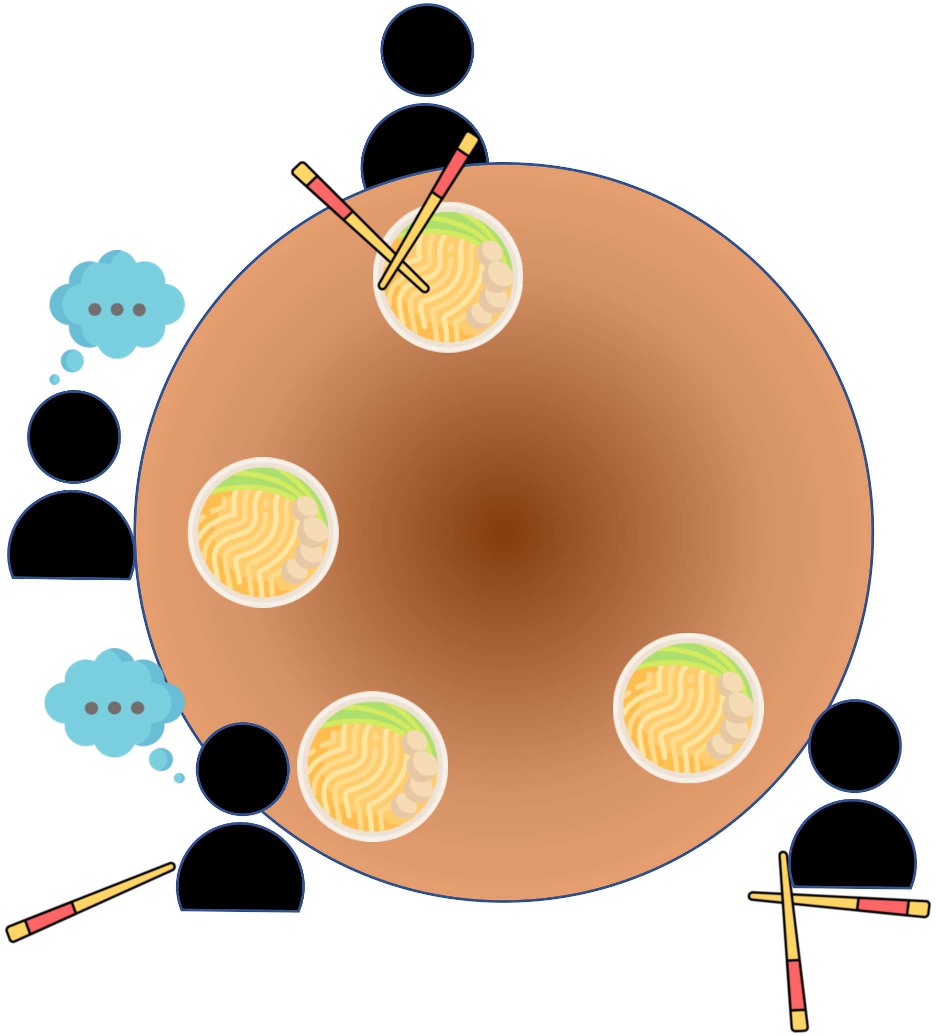


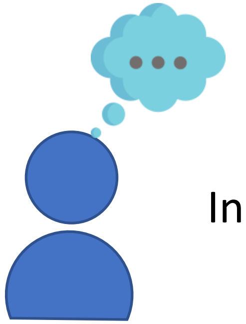
Any other approaches?



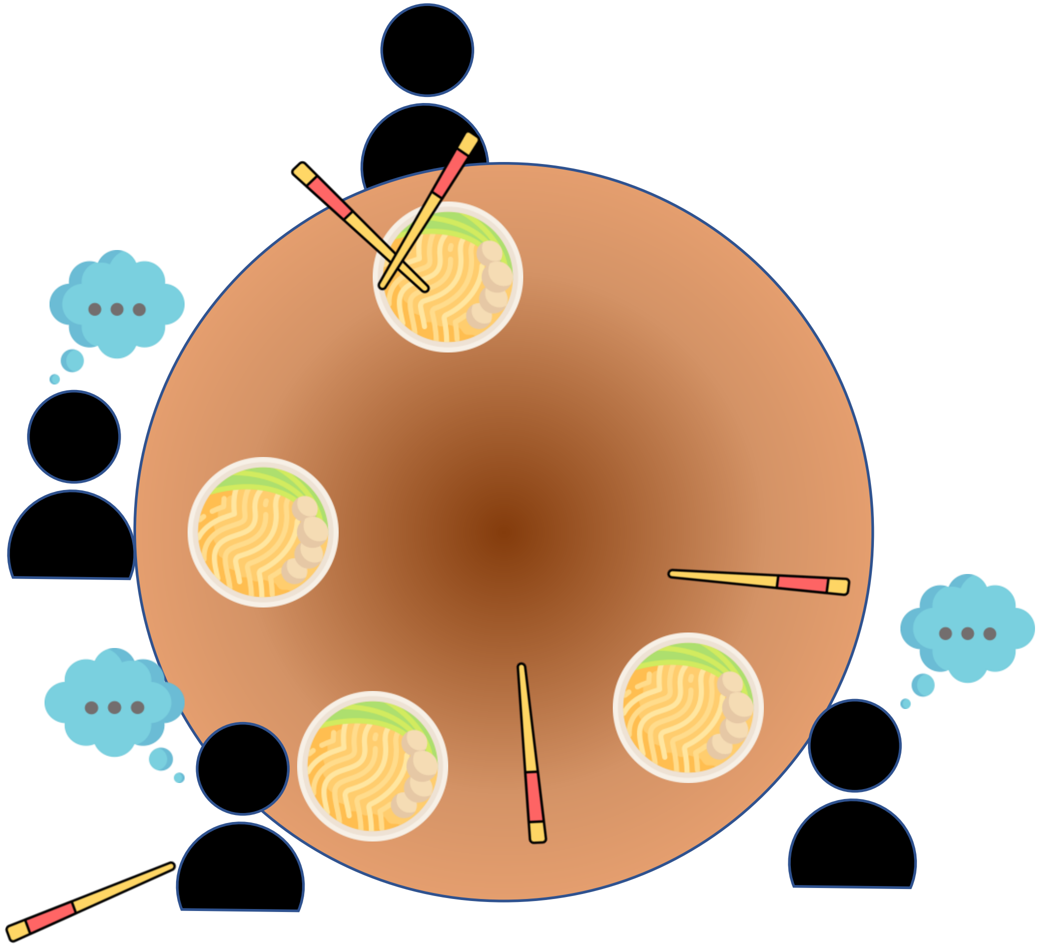


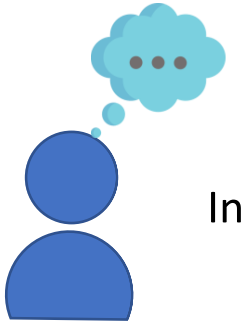
Only allow 4 philosophers
At the table





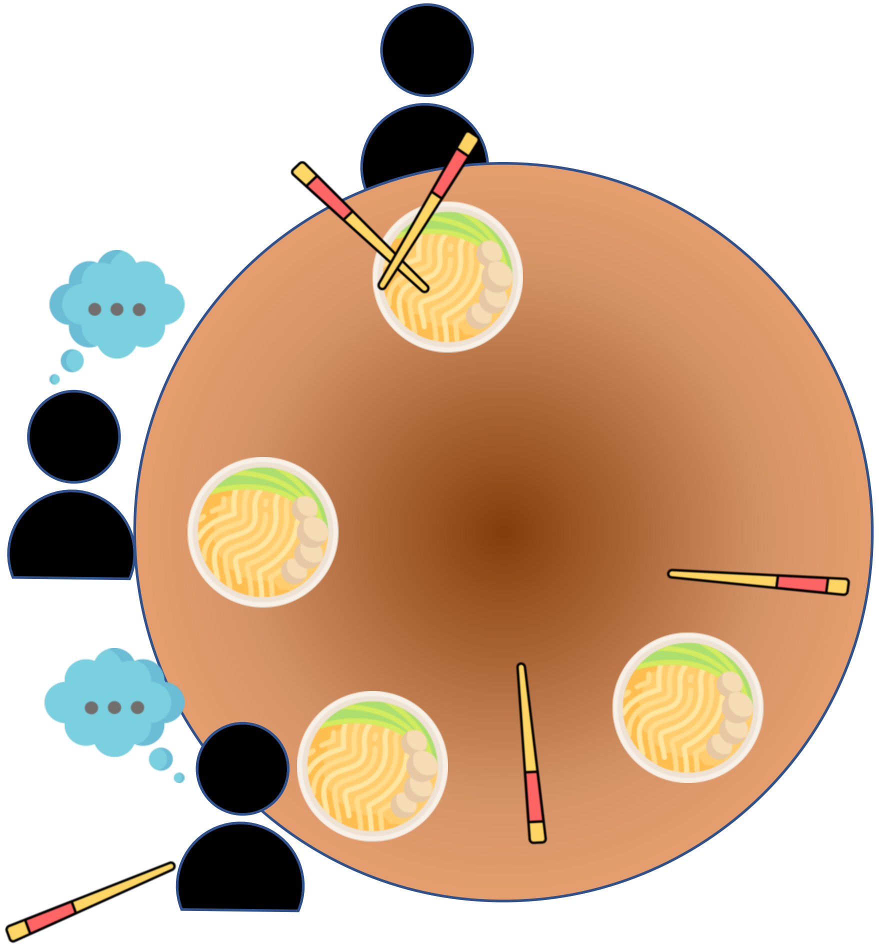
Only allow 4 philosophers
At the table

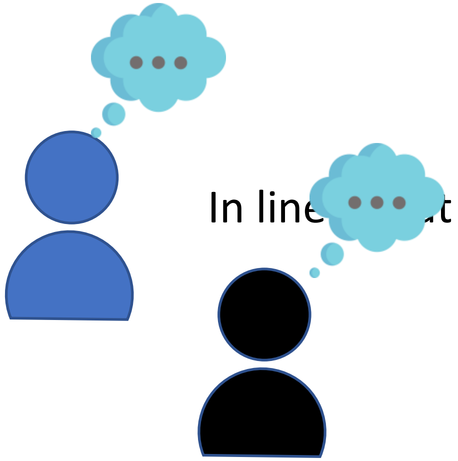




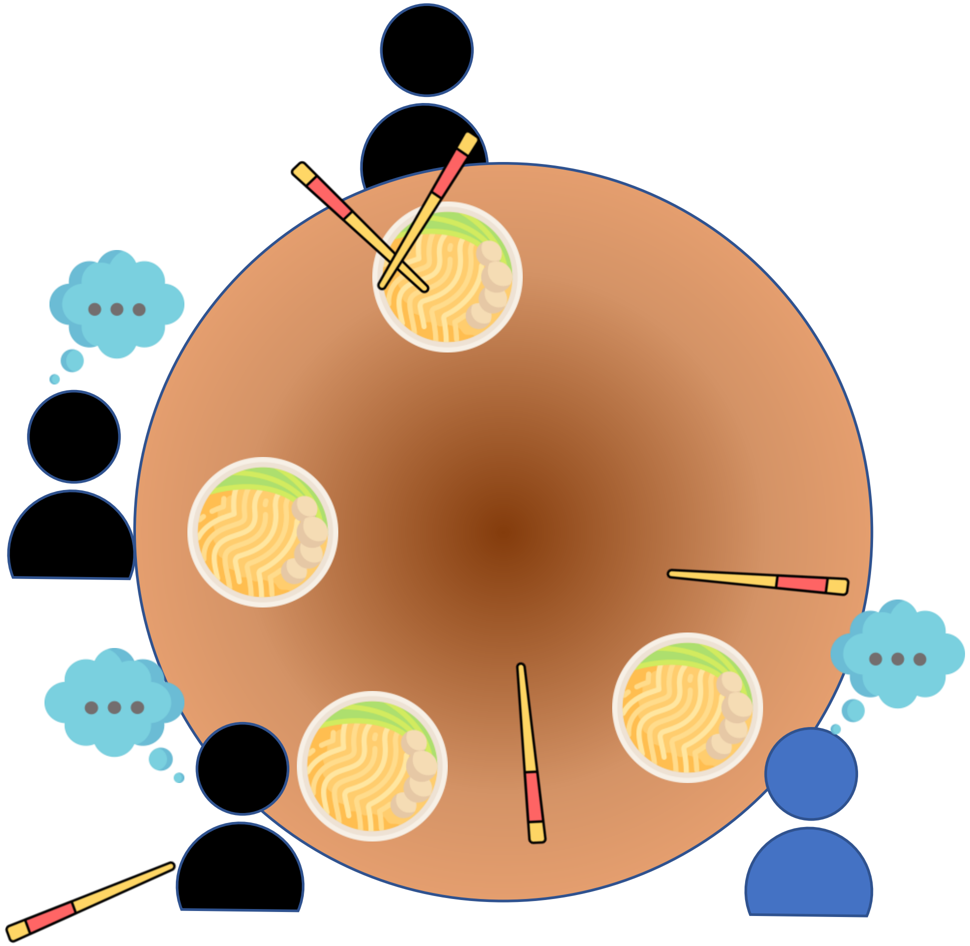
In line to eat

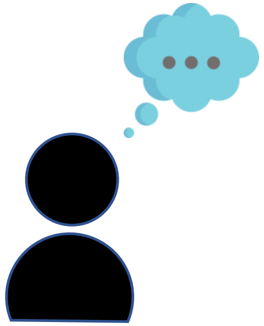
Only allow 4 philosophers
At the table





Only allow 4 philosophers
At the table





In line to eat

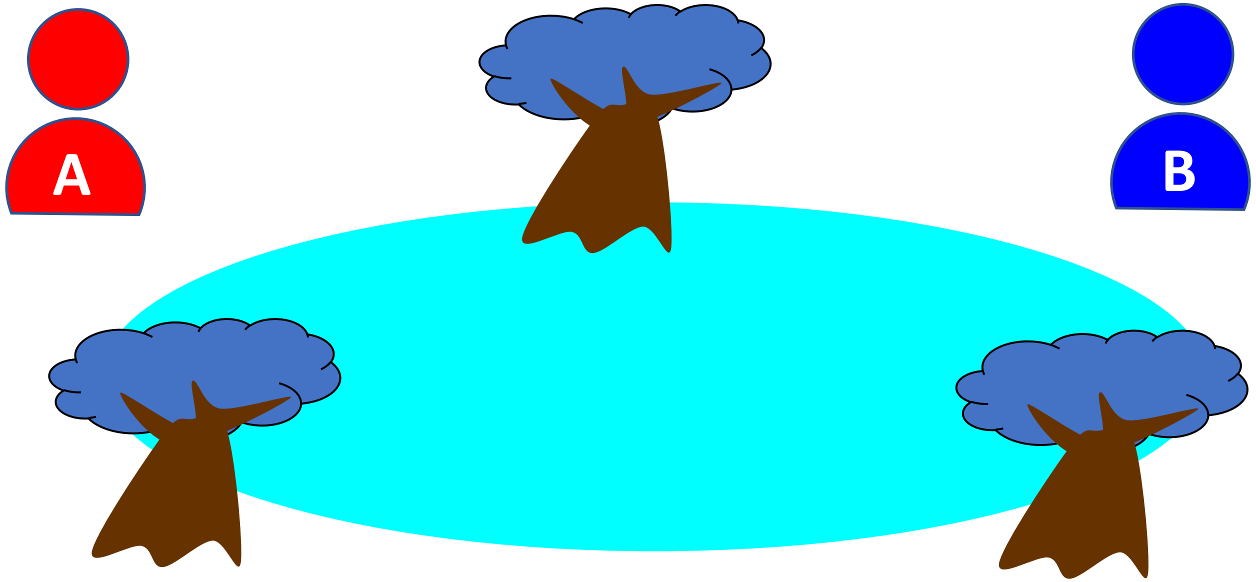
Only allow 4 philosophers
At the table

Dining philosophers – more solutions

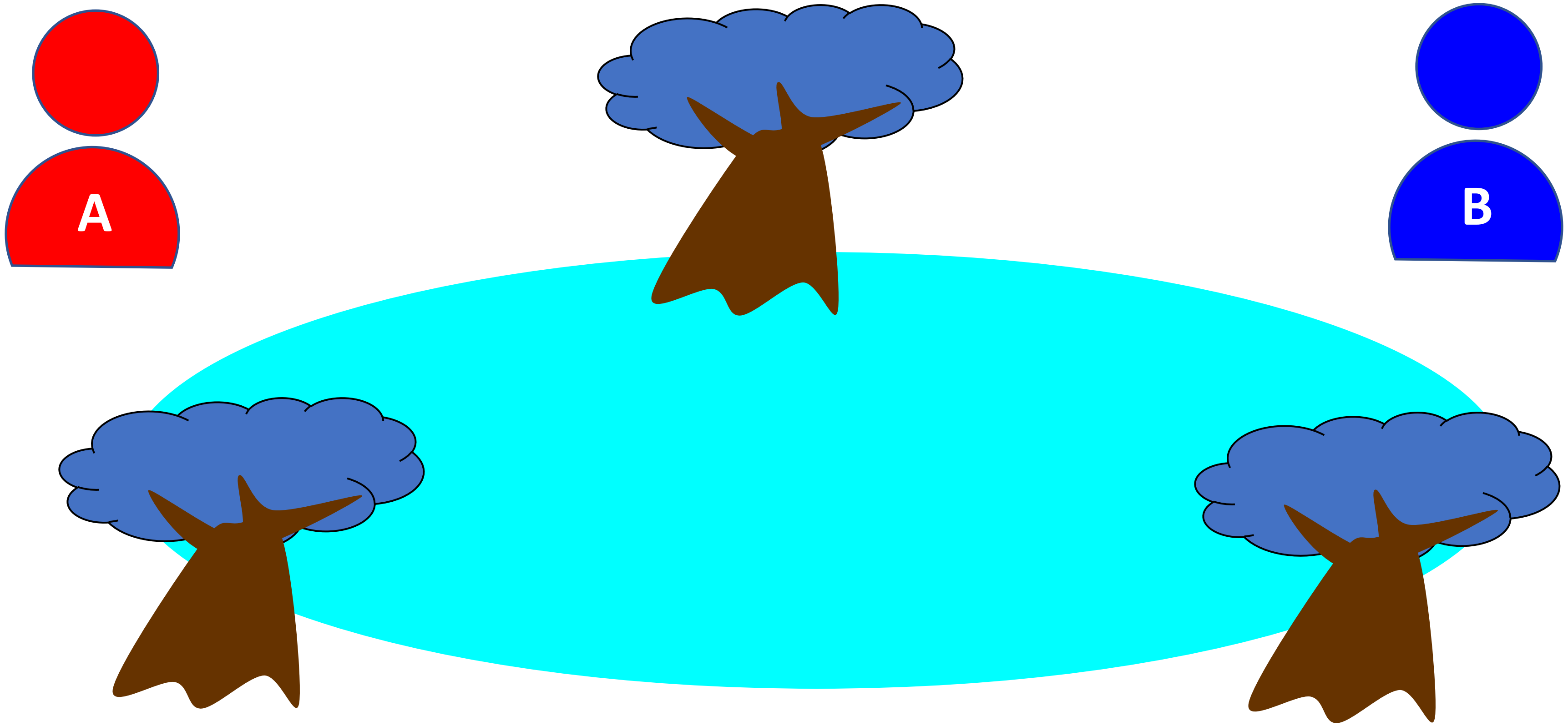
- Other solutions are possible as well:
  - Different asymmetry: make one philosopher grab the left fork first and then the right fork; all others grab the right fork first and then the left fork.
  - Use an arbiter who determines the order in which the philosophers can eat. Arbiter allows philosophers to pick up 2 chopsticks at once.
    - Such an “arbiter” could be a Monitor as mentioned two weeks ago (ideal)
    - Or could just be a single global lock that must be held to pick up any chopsticks
  - Use backoffs and randomness to break deadlock.

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

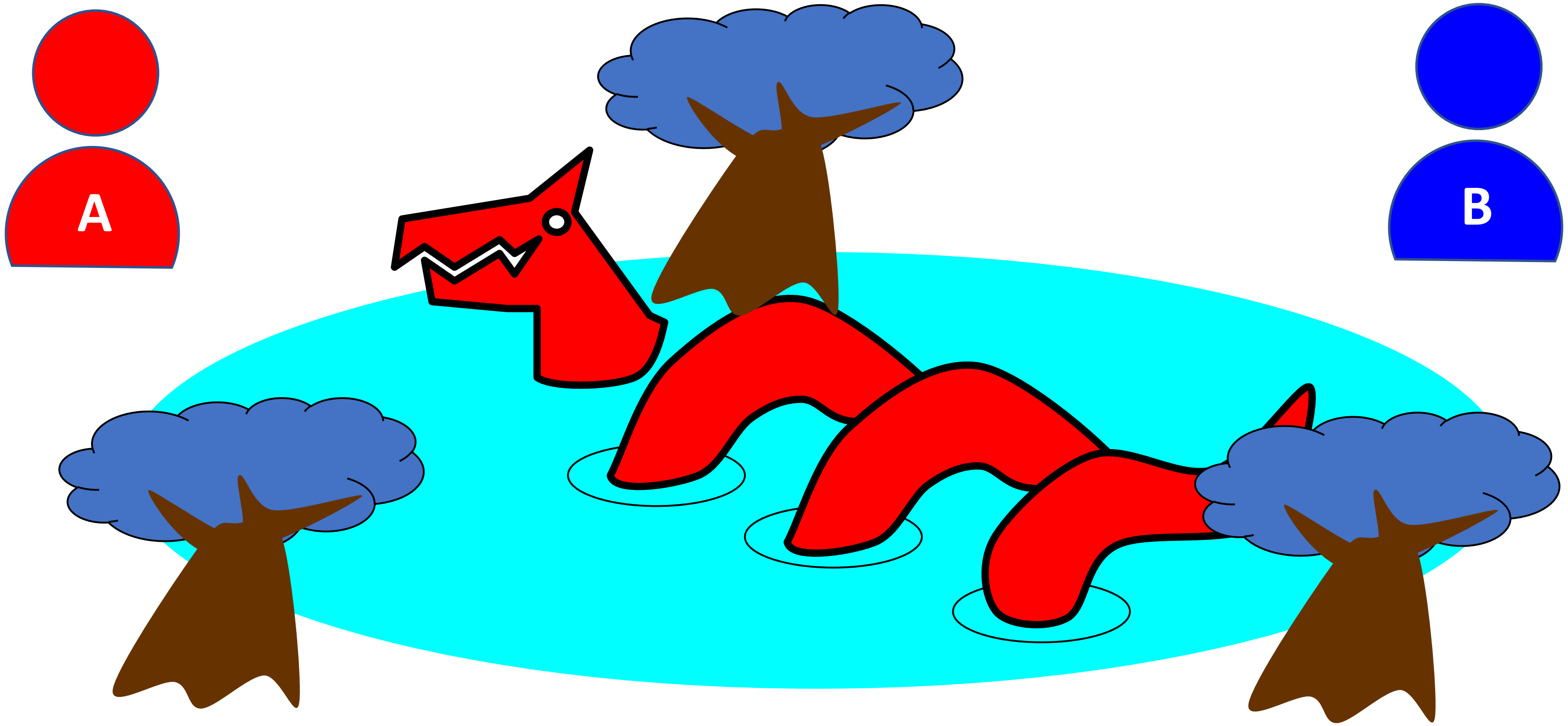
- The following story was told by a famous multiprocessing pioneer
- Leslie Lamport. (who also authored LaTeX!)



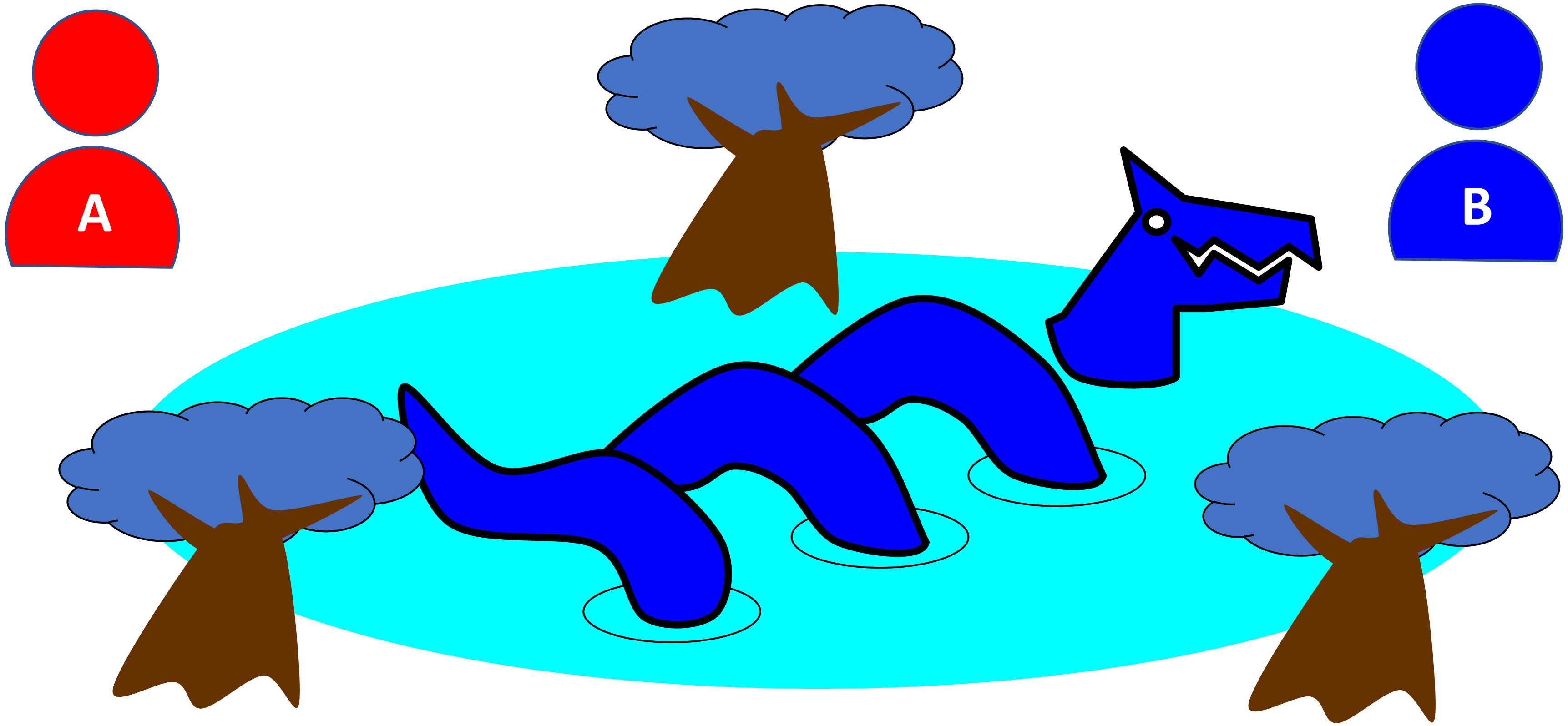
Mutual Exclusion, or "Alice & Bob share a pond"



Alice has a pet



Bob has a pet



The Problem

Formalizing the Problem

- Mutual exclusion
  - Both pets never in pond simultaneously
- No starvation
  - If one wants in, eventually it gets in
  - (“No starvation” implies “no deadlock”!)

Formalizing the Problem

- Mutual exclusion
  - Both pets never in pond simultaneously
  - Safety property!
- No deadlock
  - If one wants in, eventually it gets in
  - (“No starvation” implies “no deadlock”!)
  - Liveness property!

Simple Protocol

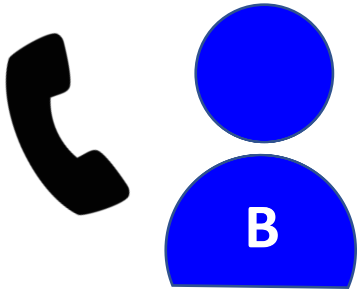
- Idea
  - Just look at the pond
- Gotcha
  - Not atomic
  - “Trees obscure the view”

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

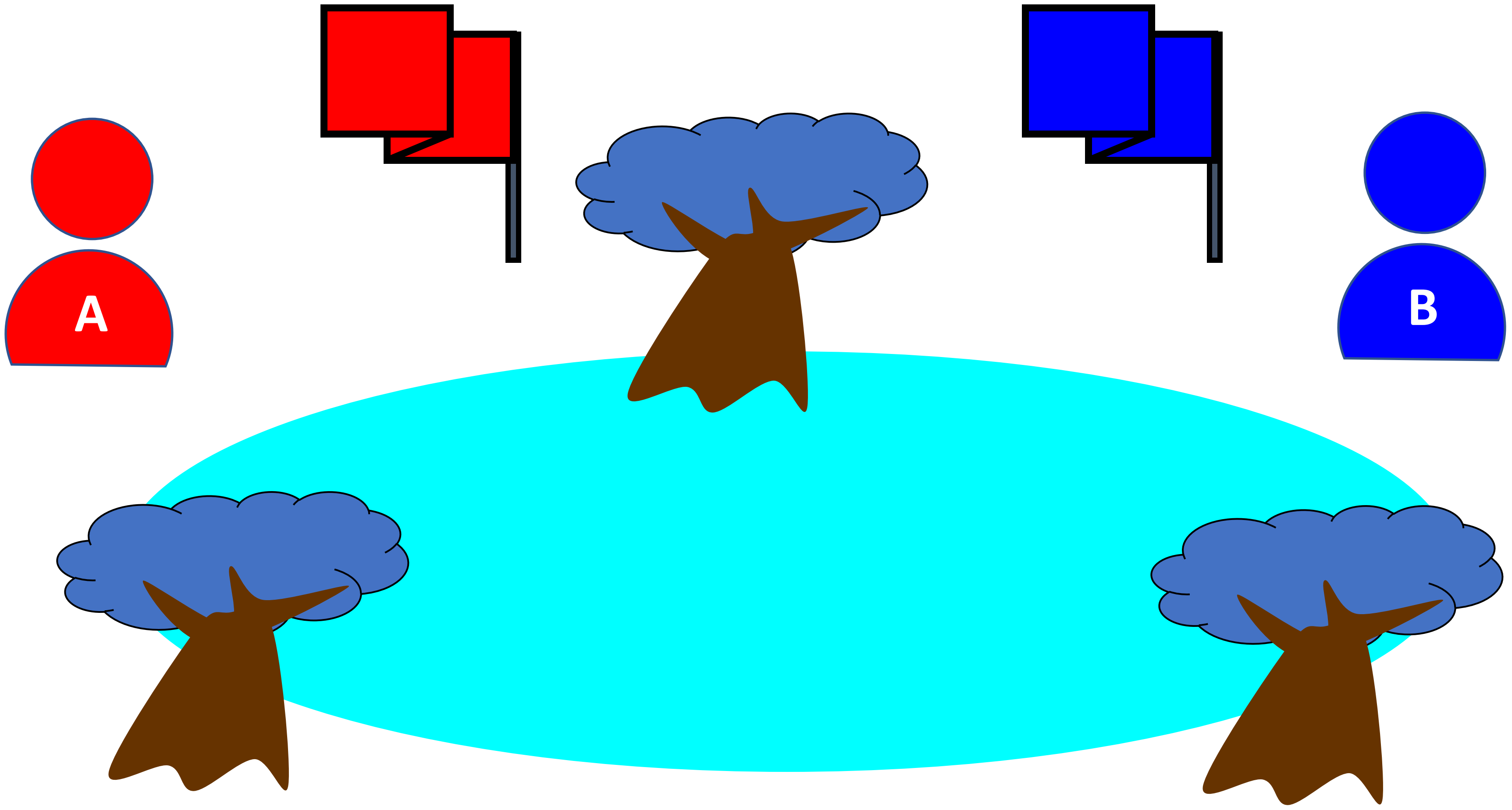


Interpretation

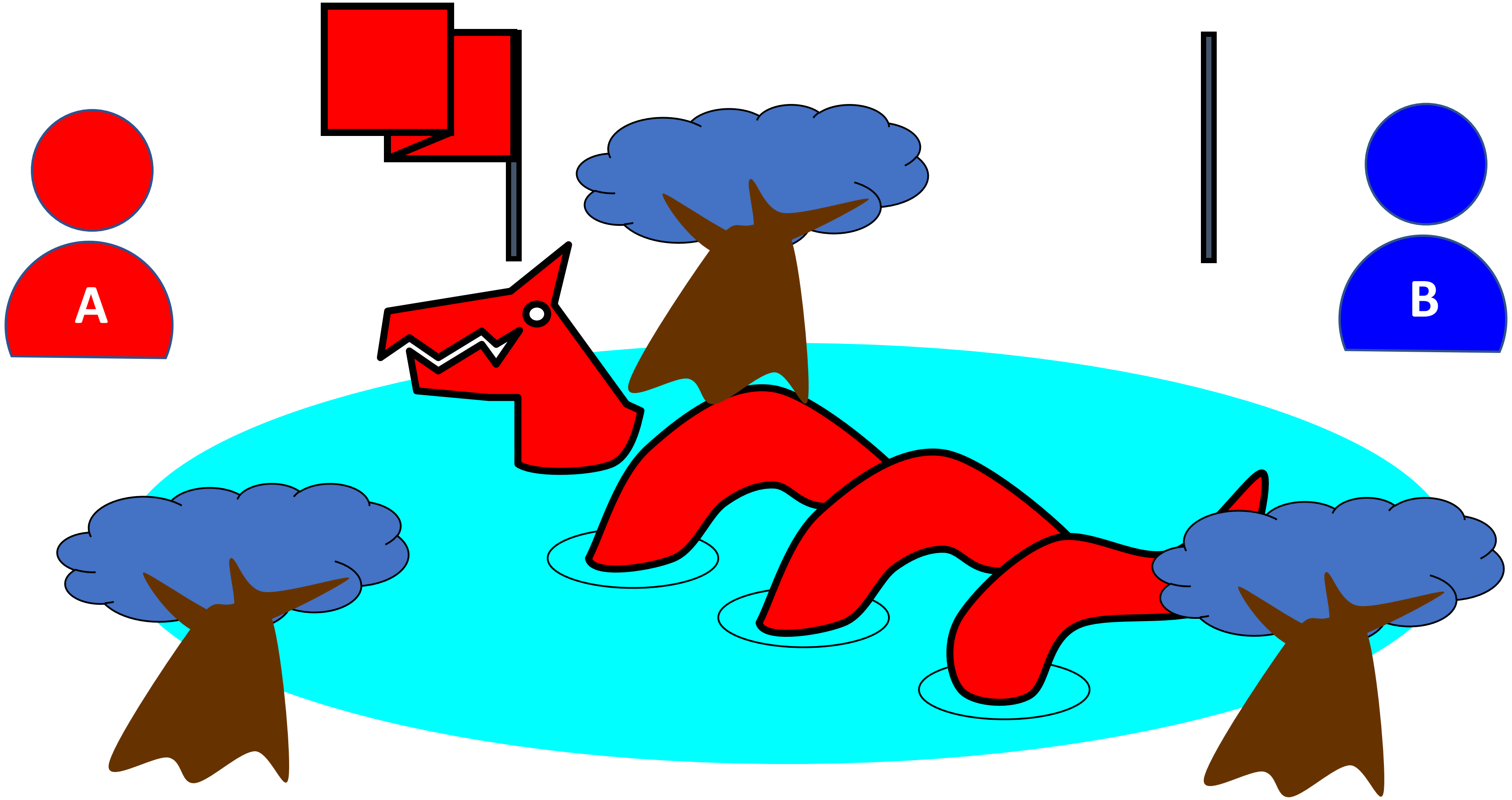
- Message-passing with shared memory doesn't work
- Recipient might not be
  - Listening
  - There at all

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

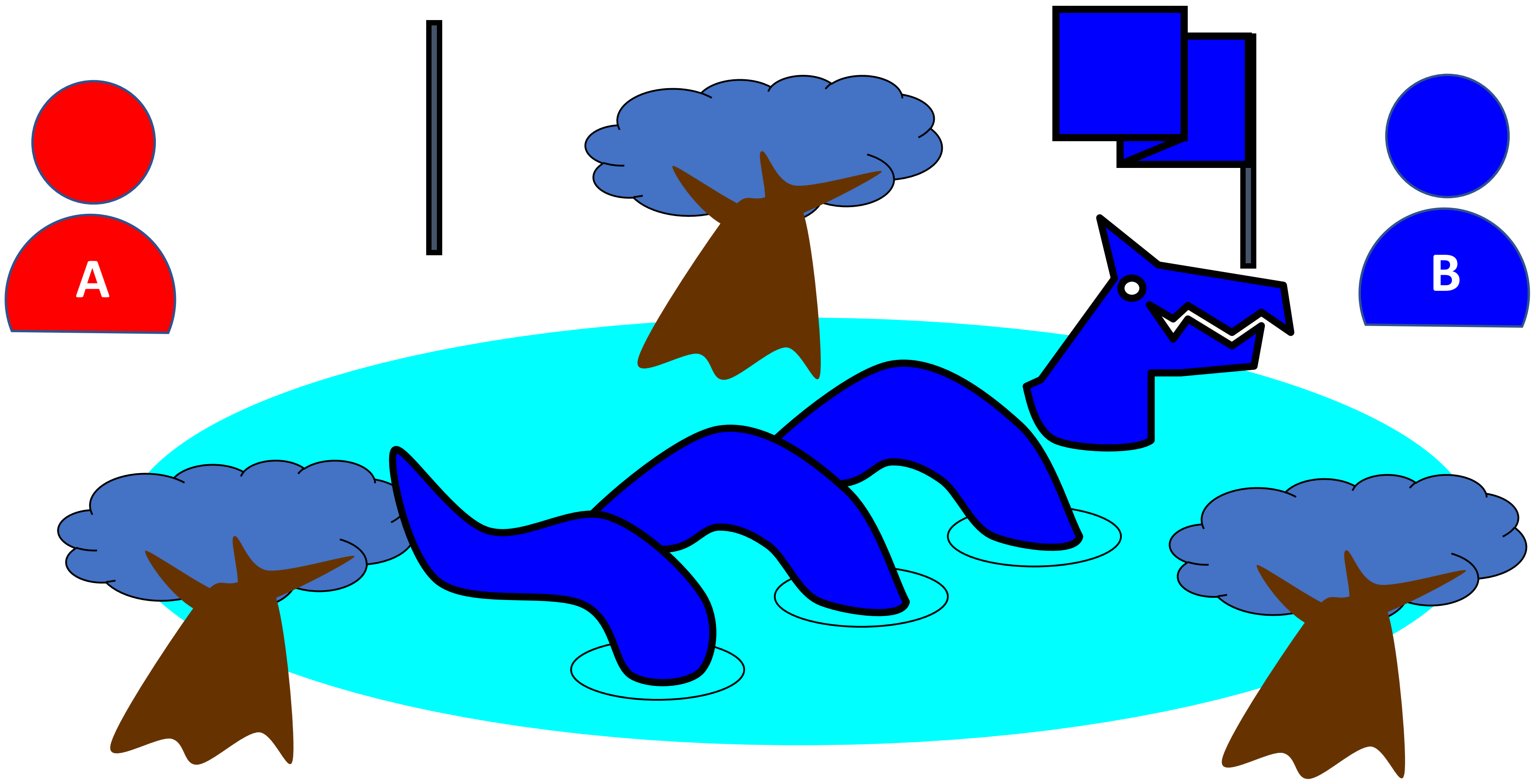
Flag Protocol



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 (2nd 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 a pet is in the pond, it means that a **thread is 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)

- Same essential problem as "Alice and Bob share a Fridge."
  - Obviously solved by a mutex… but how to make one?
  - Can Alice and Bob solve it differently from last time?
    - Want: they are both **using the same protocol**?

Peterson’s Algorithm

```
Alice() {
  raise flag
  point sign at Bob
  while (Bob’s flag & sign pointed at Bob) { }
  release pet dragon
  lower flag when pet returns
}
```

```
Bob() {
  raise flag
  point sign at Alice
  while (Alice’s flag & sign pointed at Alice) { }
  release pet dragon
  lower flag when pet returns
}
```

Peterson’s Algorithm

Alice() {
  raise flag
  point sign at Bob
  while (Bob’s flag & sign pointed at Bob) { }
  release pet dragon
  lower flag when pet returns
}

Bob() {
  raise flag
  point sign at Alice
  while (Alice’s flag & sign pointed at Alice) { }
  release pet dragon
  lower flag when pet returns
}

- Flag means “I want to use the pond”
- But *both* Alice and Bob try deferring to the other first
  - Sign is an extra shared variable

Correctness (mutual exclusion)

- Proof sketch (don’t memorize this, but concepts may be helpful)

- Assume both dragons in pond at the same time
  - Alice => !(Bob’s flag & sign pointed at Bob) => Bob’s flag down or sign -> Alice
    - But Bob’s flag down => Bob’s pet cannot be in the pond right now
    - Contradiction!
  - Therefore, sign -> Alice.
  - But Bob => !(Alice’s flag & sign pointed at Alice) => …
  - Likewise, either Alice’s pet isn’t in the pond, or sign -> Bob.
  - In either case, contradiction!

Correctness (no deadlocks)

- Only possible point of deadlock is while loop
- Conditions:
  - (Bob’s flag & sign pointed at Alice) and
  - (Alice’s flag & sign pointed at Bob)
- Can both conditions be true at once?
- **No**. Even with preemption, the sign can only face one way!

- So, if Alice and Bob are both at the while loop, one can advance

Correctness (no starvation - weak)

- Don’t worry about weak vs strong liveness properties

- Bob can’t just keep using the pond without giving Alice a turn
  - (Or vice versa)
- After giving up the pond, if Bob wants it again, he must:
  - Raise his flag
  - **Point the sign at Alice again**
- Guarantees Alice will get a turn between Bob’s, if she wants one.

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).