Week 5

Multithreading

Oana Balmau

January 31, 2023

Schedule Highlights

		add/drop deadline			
Week 4 **Process Management**	jan 23 C Tools: GDB basics	jan 24 **Multi-process Structuring (1/2)** **Team registration deadline**	jan 25	jan 26 **Multi-process Structuring (2/2)**	jan 27
Week 5 **Process Management**	jan 30 C Review: Pointers & Memory Allocation I	jan 31 **Multithreading (1/2)**	feb 1	feb 2 **Multithreading (2/2)** Practice Exercises Sheet: Process Management	feb 3
Week 6 **Memory Management**	feb 6 C Review: C files	feb 7 **Virtual Memory (1/2)** Optional reading: OSTEP Chapters 12 – 18	feb 8 Scheduling Assignment Released	feb 9 **Virtual Memory (2/2)** Scheduling Assignment Overview — with Jiaxuan	feb 10
Week 7 **Memory Management**	feb 13 **OS Shell Assignment Due** C Review: Working with pthreads I	feb 14 **Demand Paging (1/3)** Optional reading: OSTEP Chapters 19 – 22	feb 15	feb 16 **Demand Paging (2/3)**	feb 17

Practice exercises released.

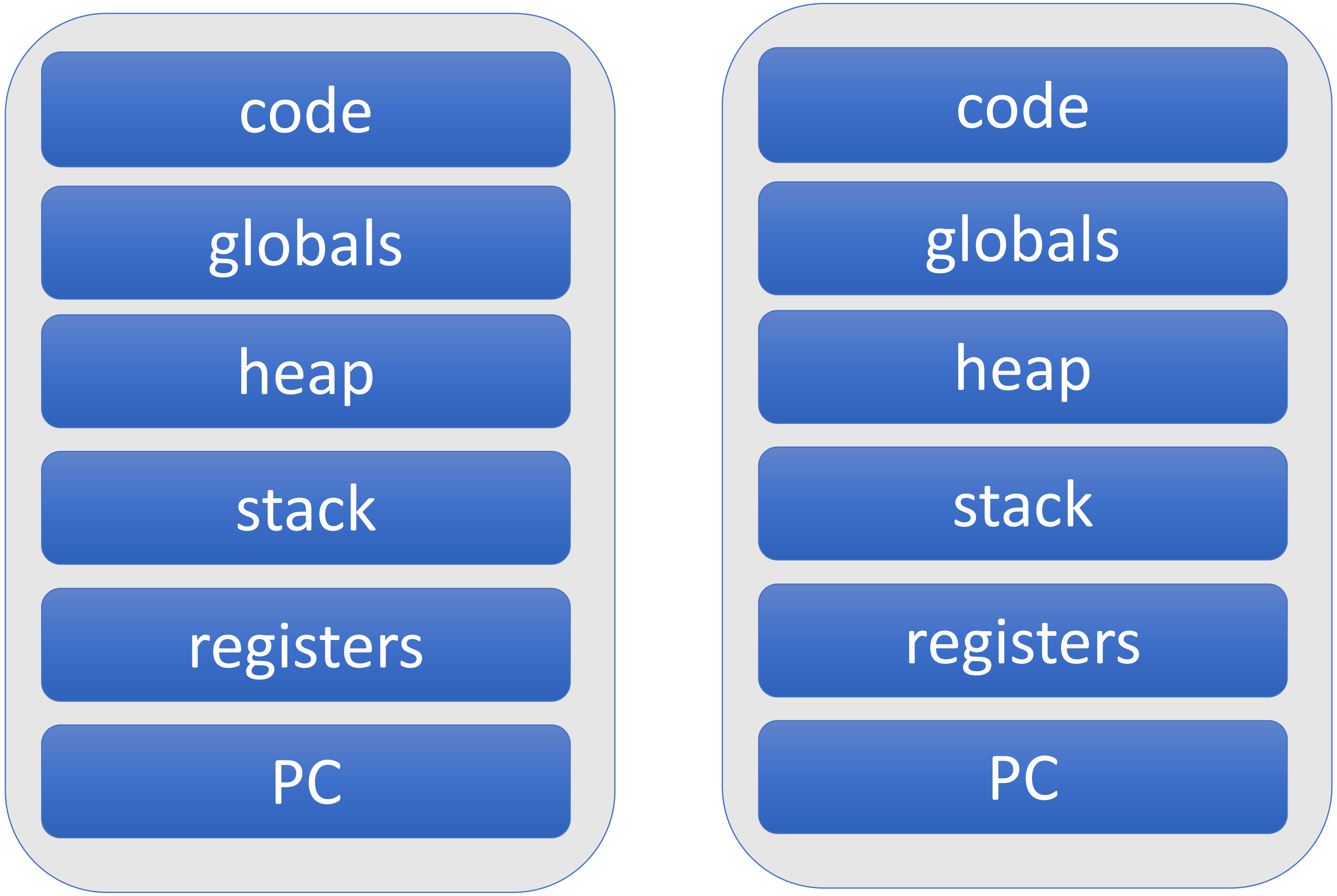
Assignment 2 released next week

Recap Week 4
Concurrency – Option 1

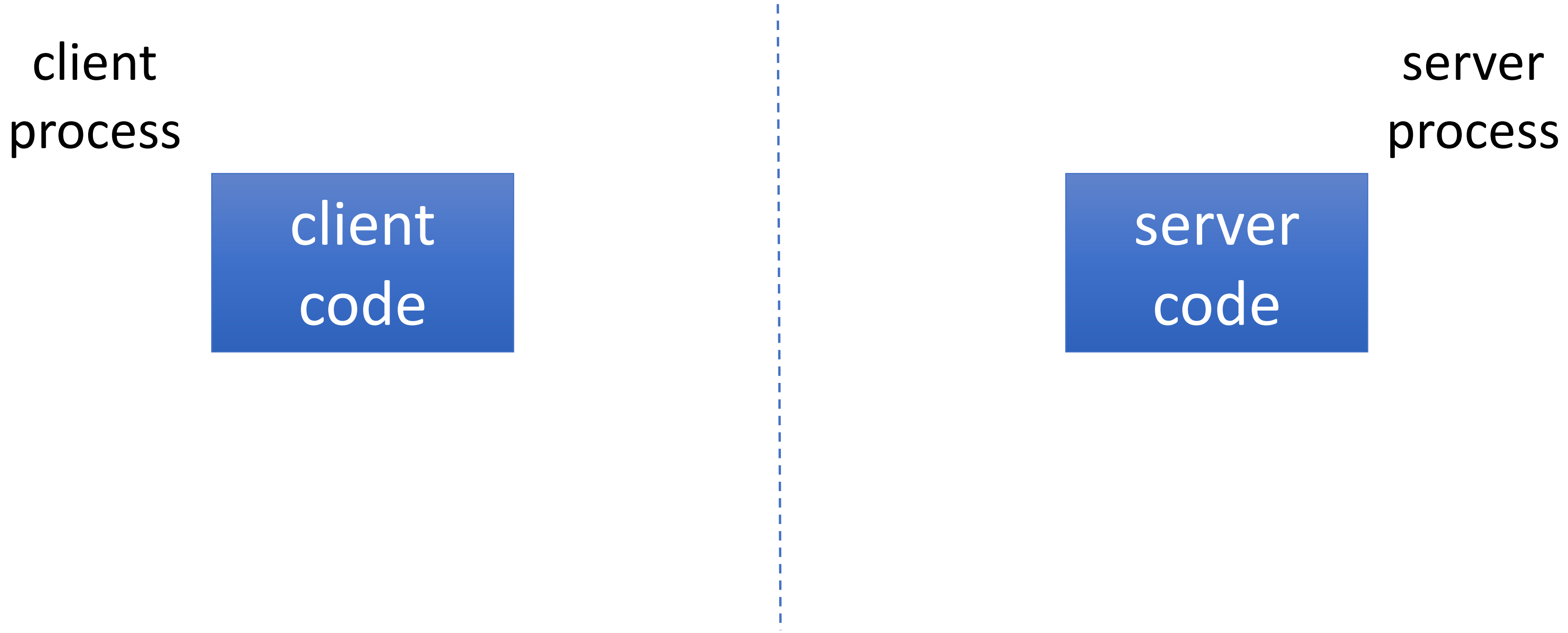
- Build apps from many communicating **processes**
- Communicate through **message passing / RPCs**
- Pros
  - If one process crashes, other processes unaffected
- Cons
  - High communication overheads

Last week's focus

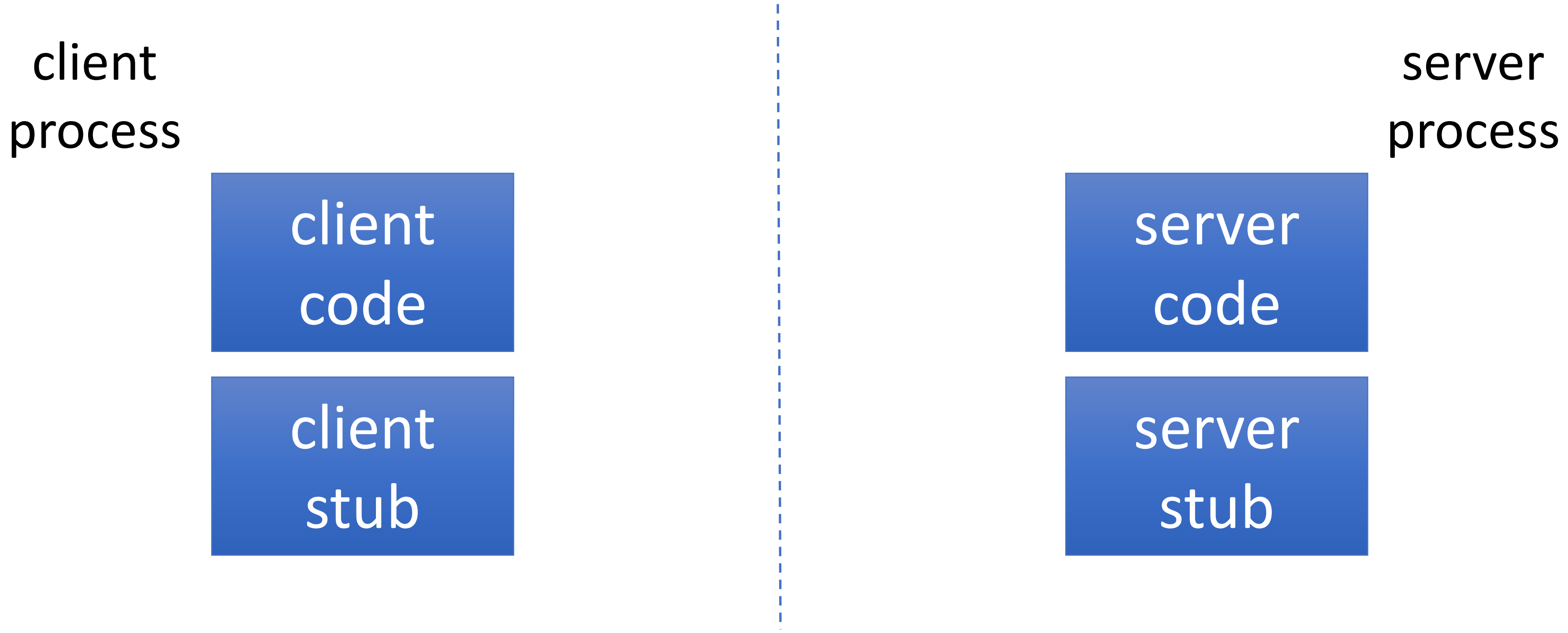
Recap Week 4: Two Processes



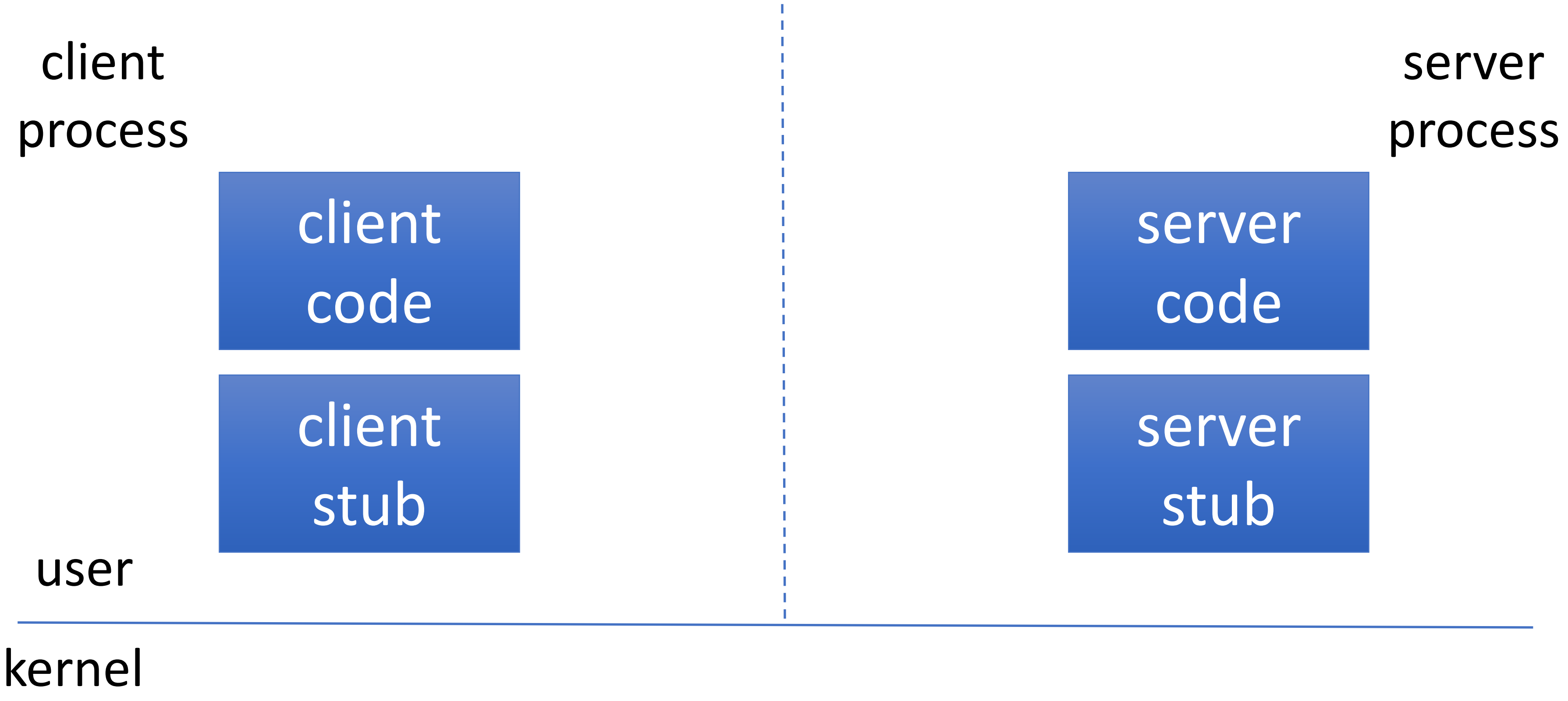
Recap Week 4 RPC Implementation



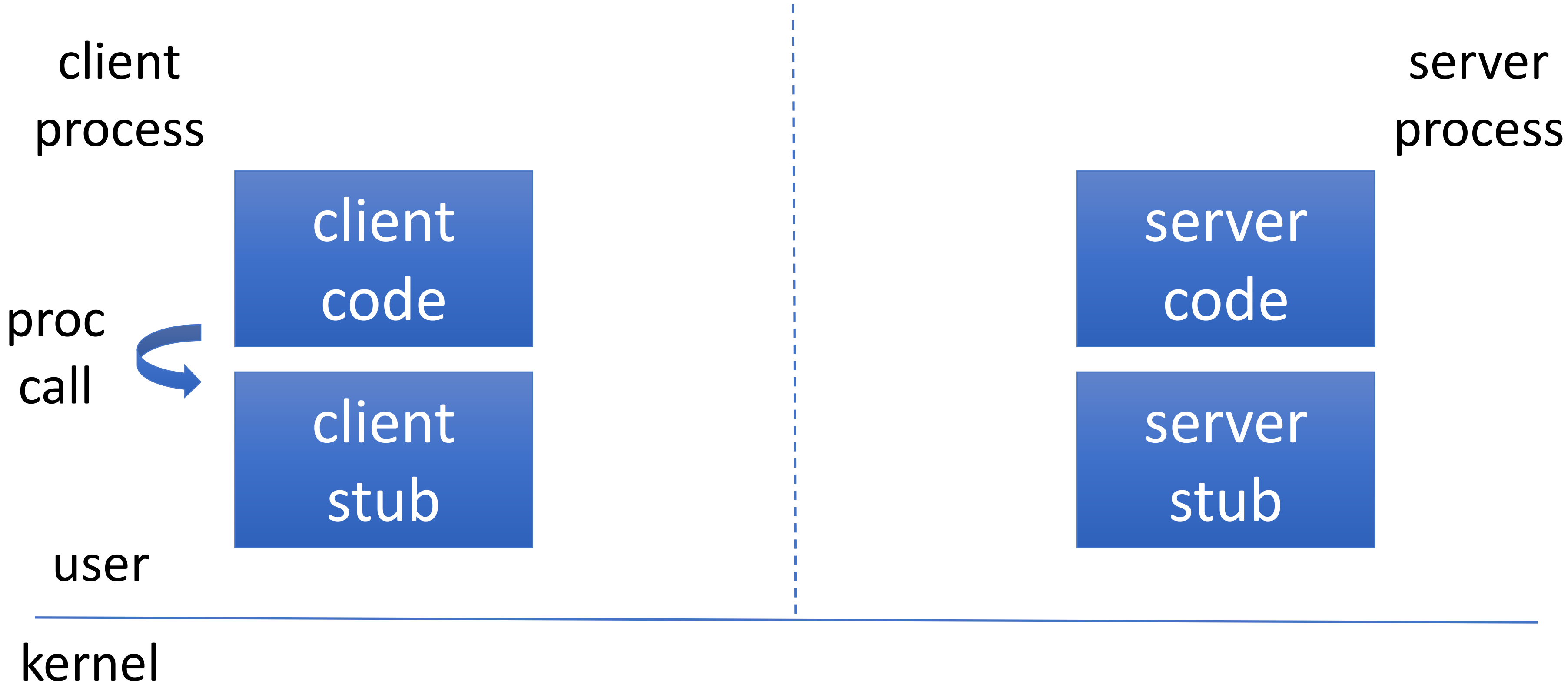
Recap Week 4 Client and Server Stubs



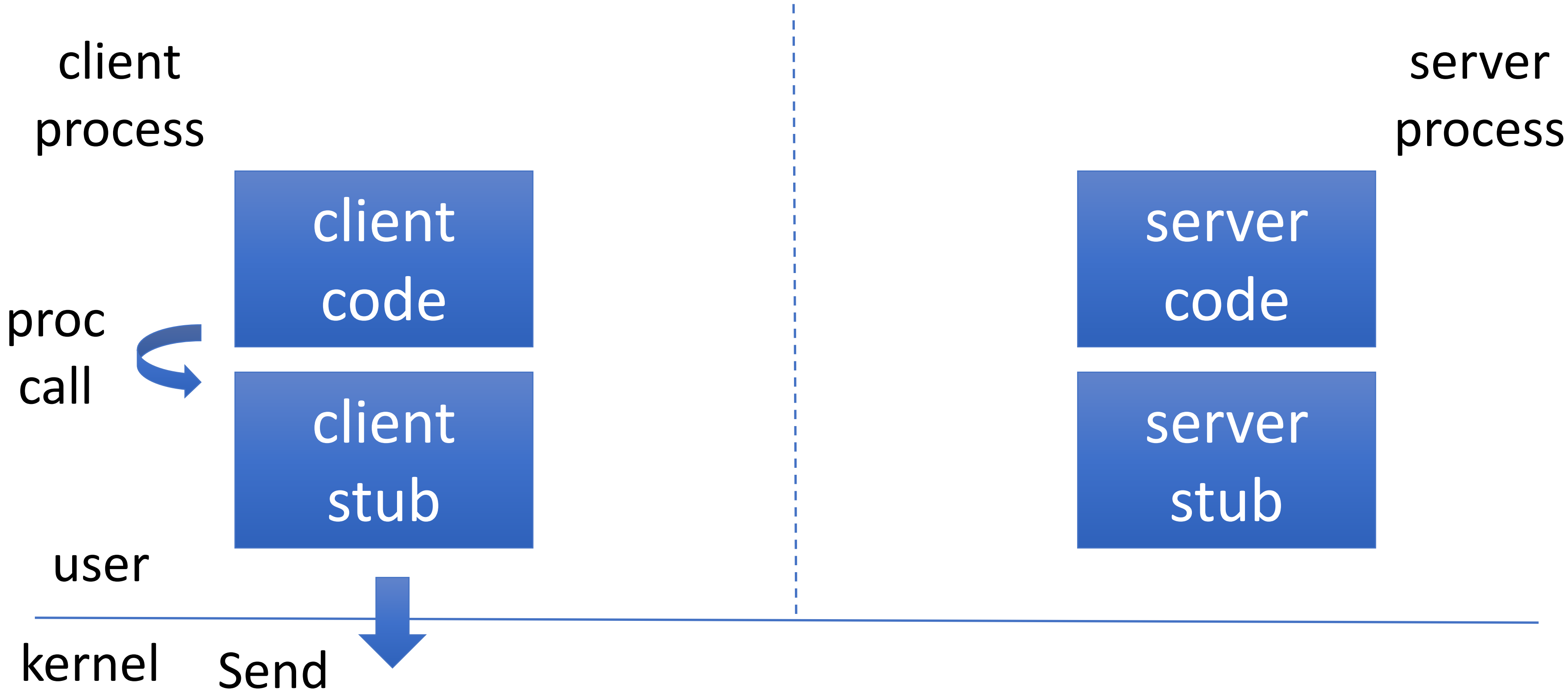
Recap Week 4 RPC Implementation: Call



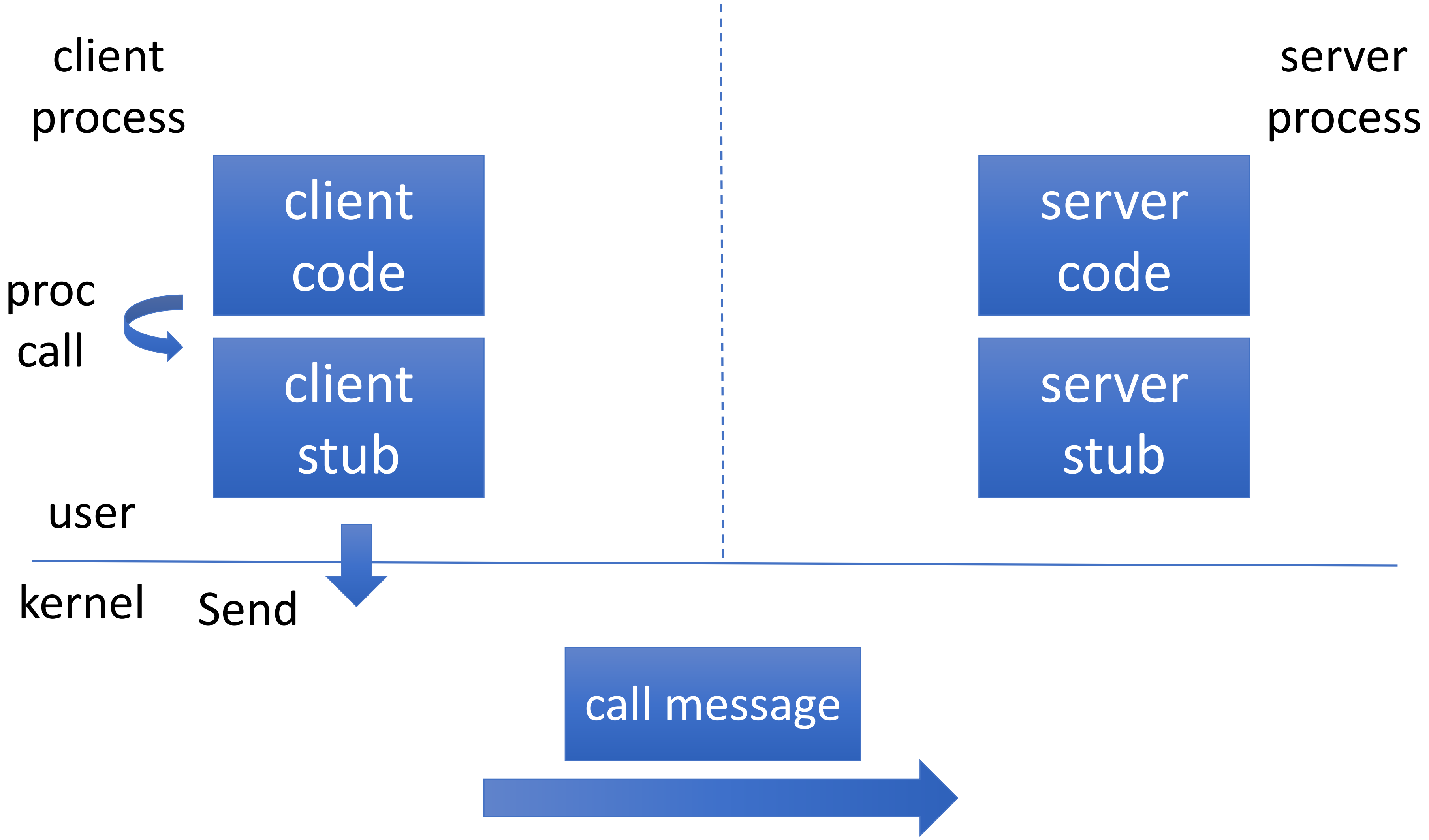
Recap Week 4 RPC Implementation: Call



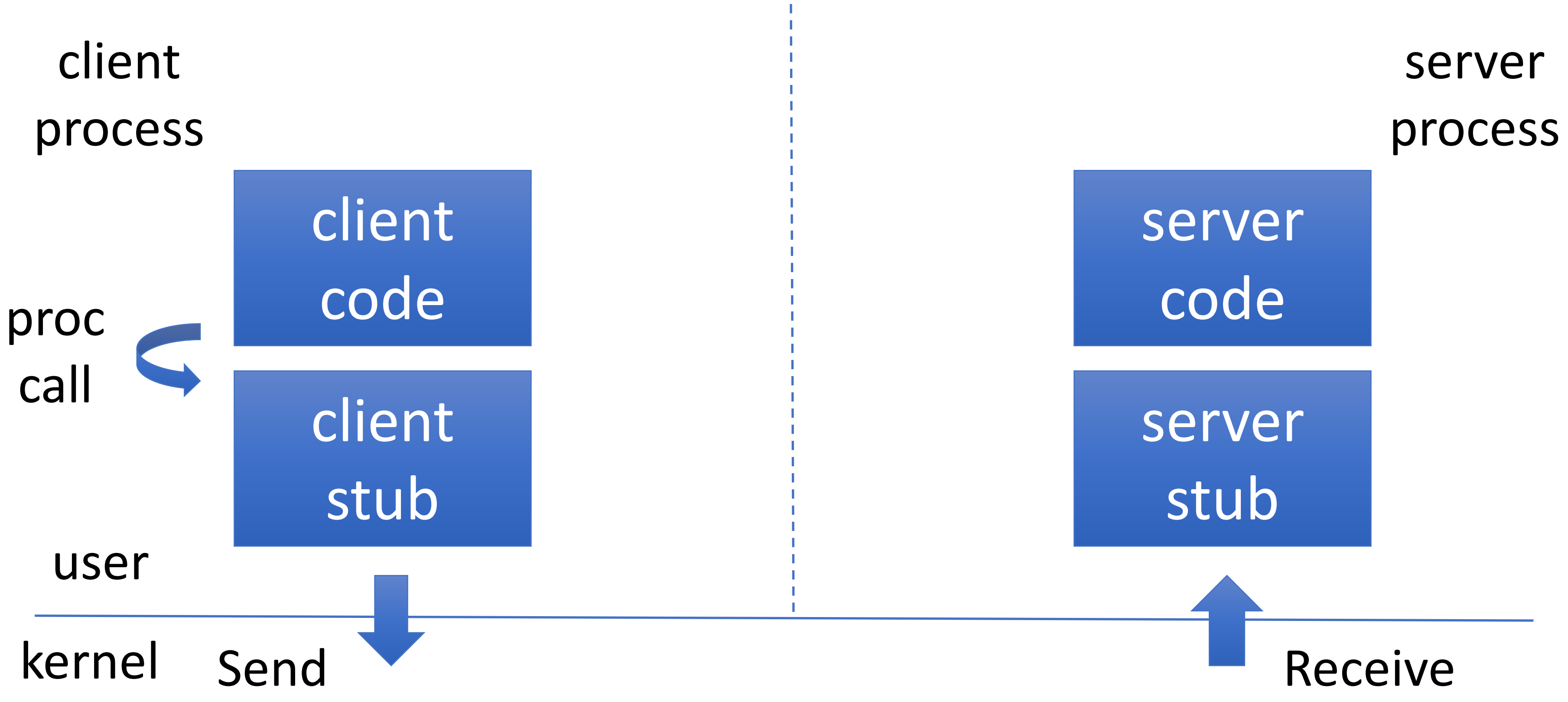
Recap Week 4 RPC Implementation: Call



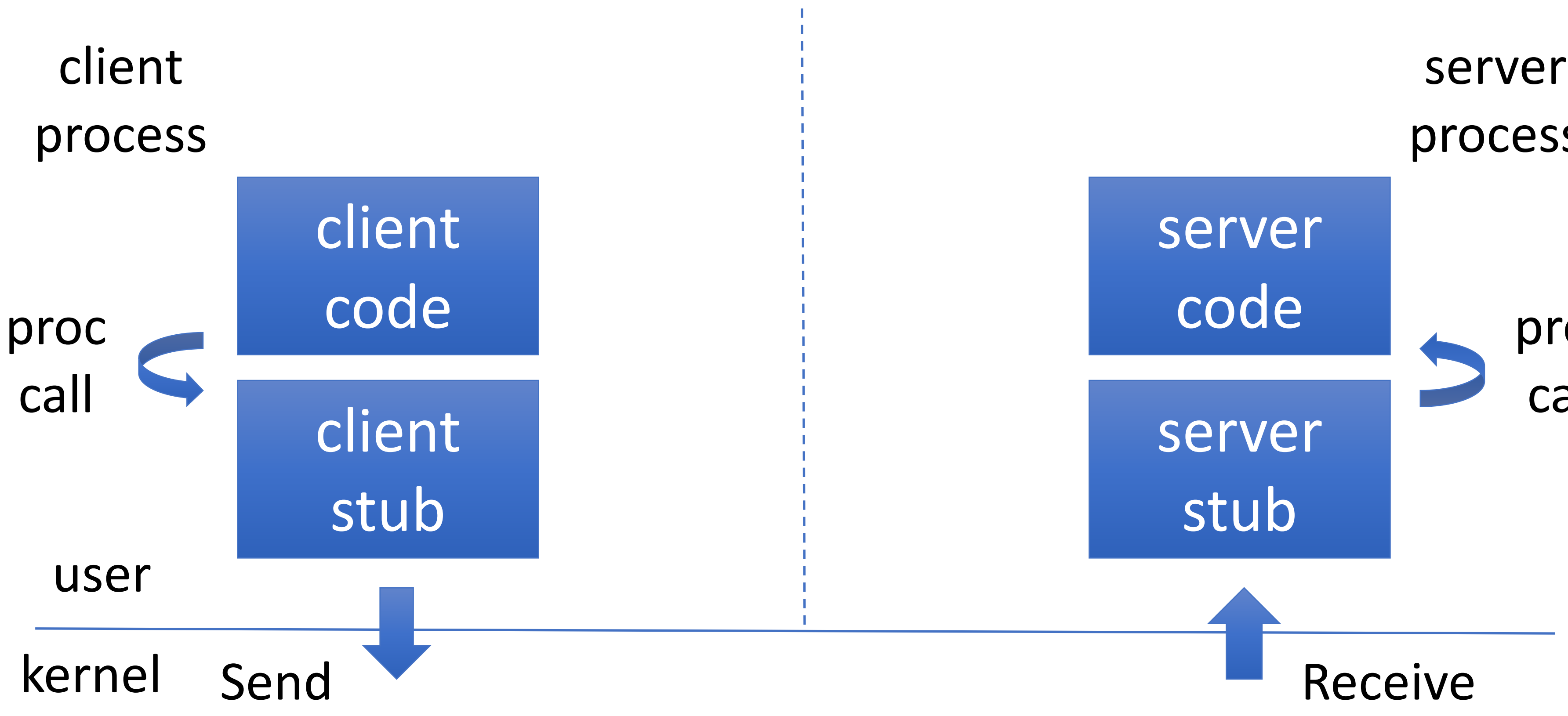
Recap Week 4 RPC Implementation: Call



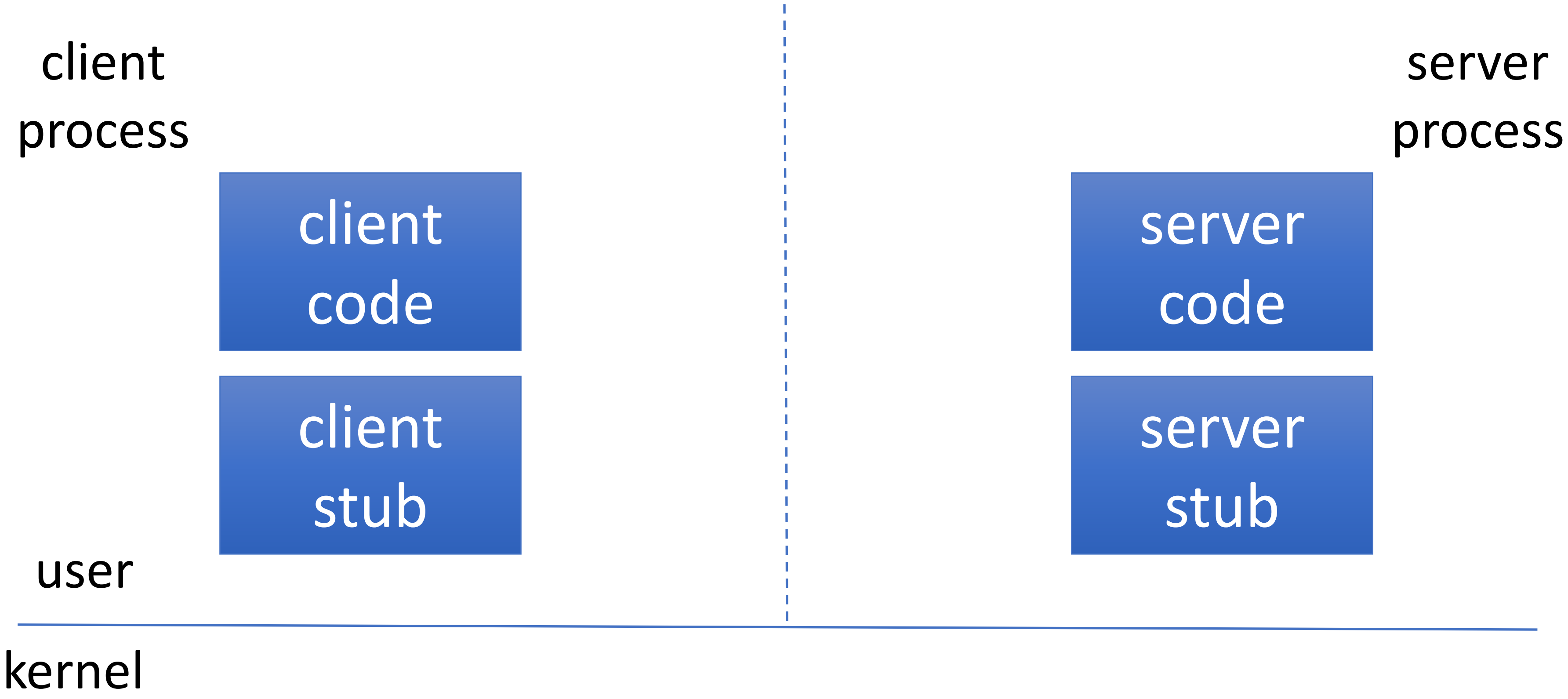
Recap Week 4 RPC Implementation: Call



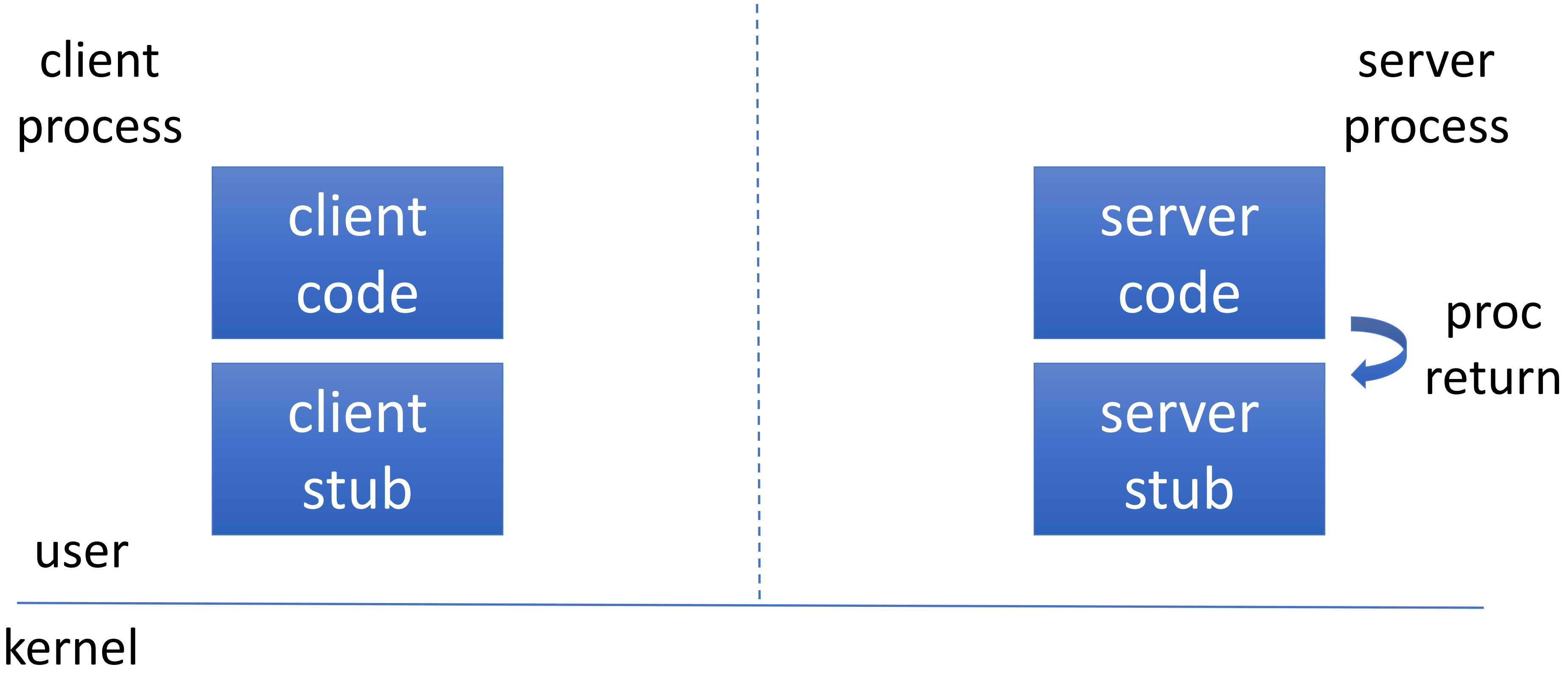
Recap Week 4 RPC Implementation: Call



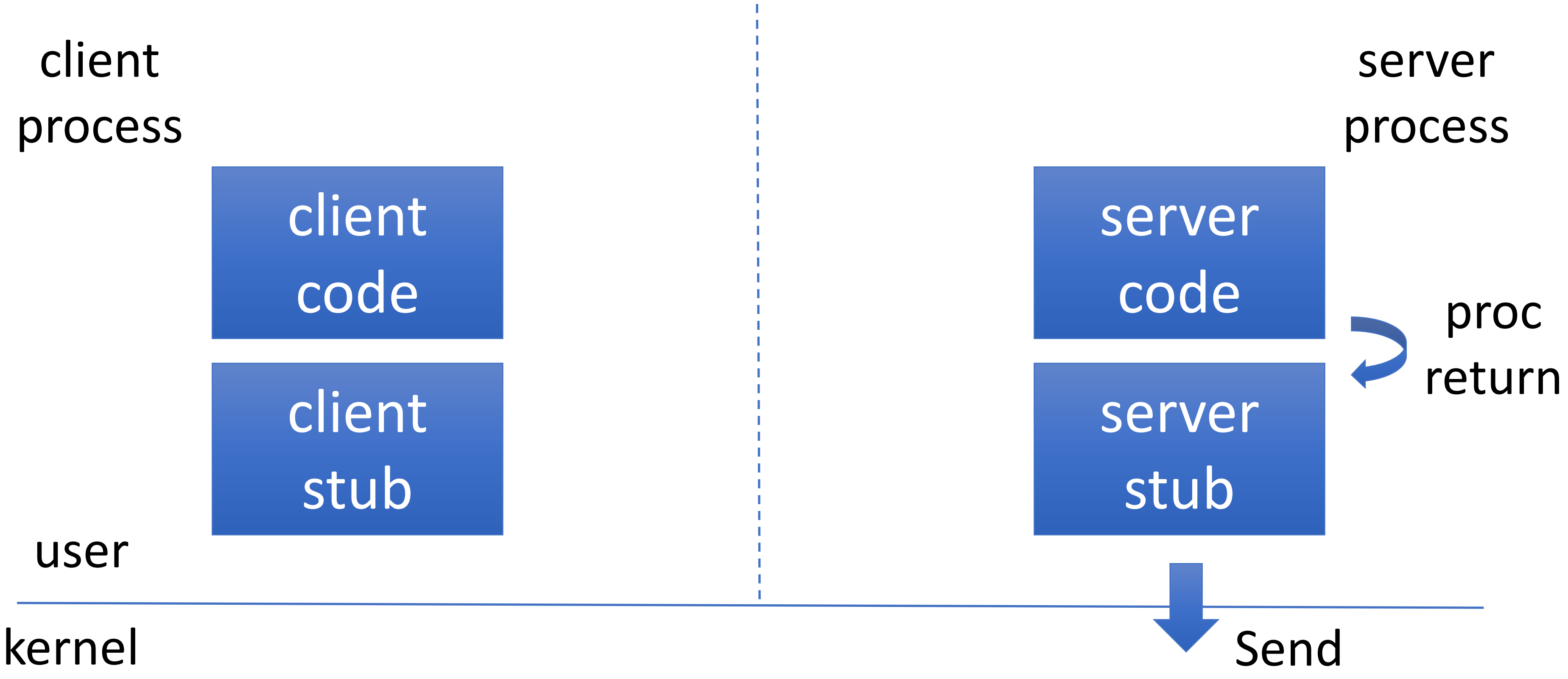
Recap Week 4 RPC Implementation: Return



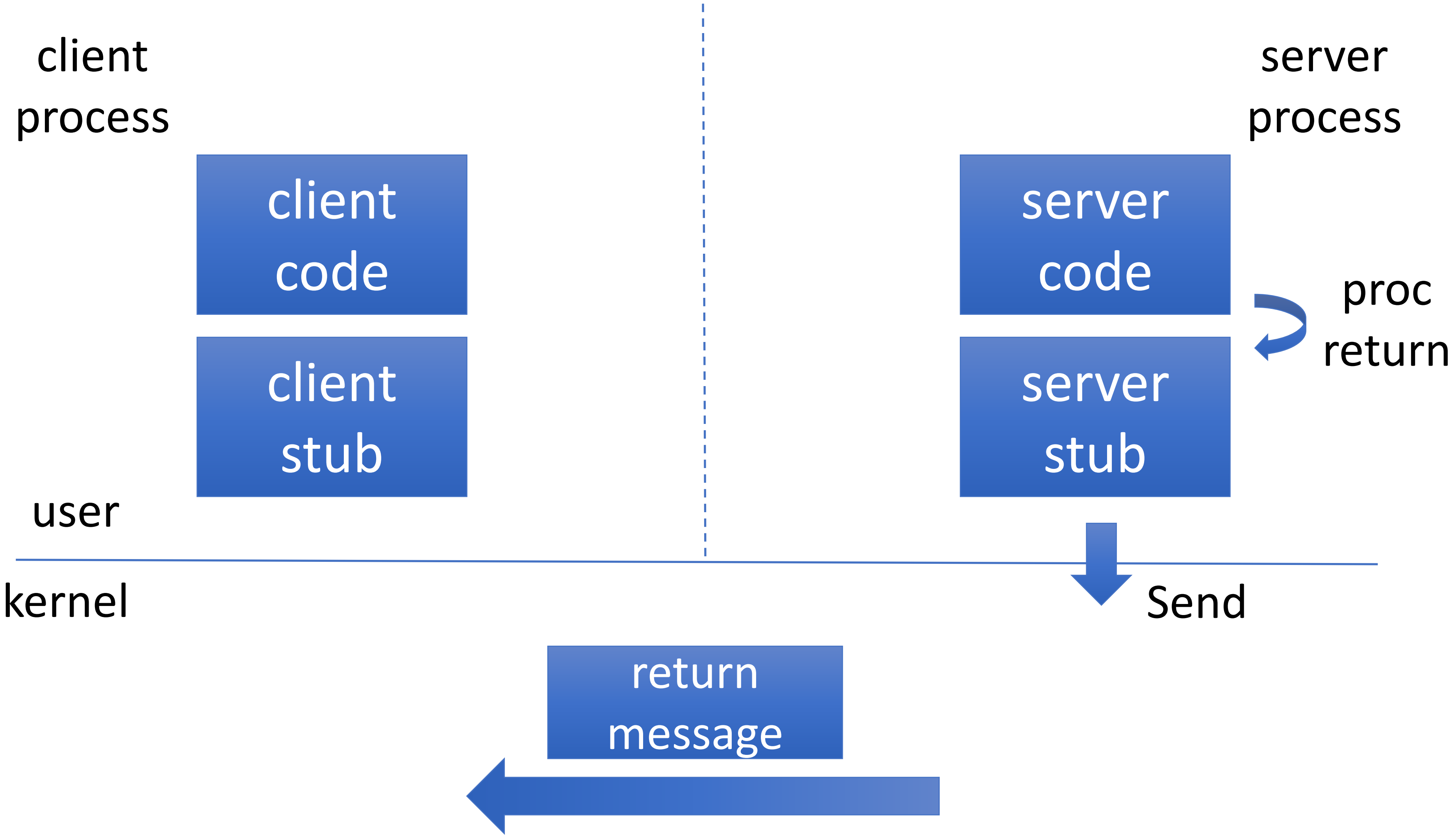
Recap Week 4 RPC Implementation: Return



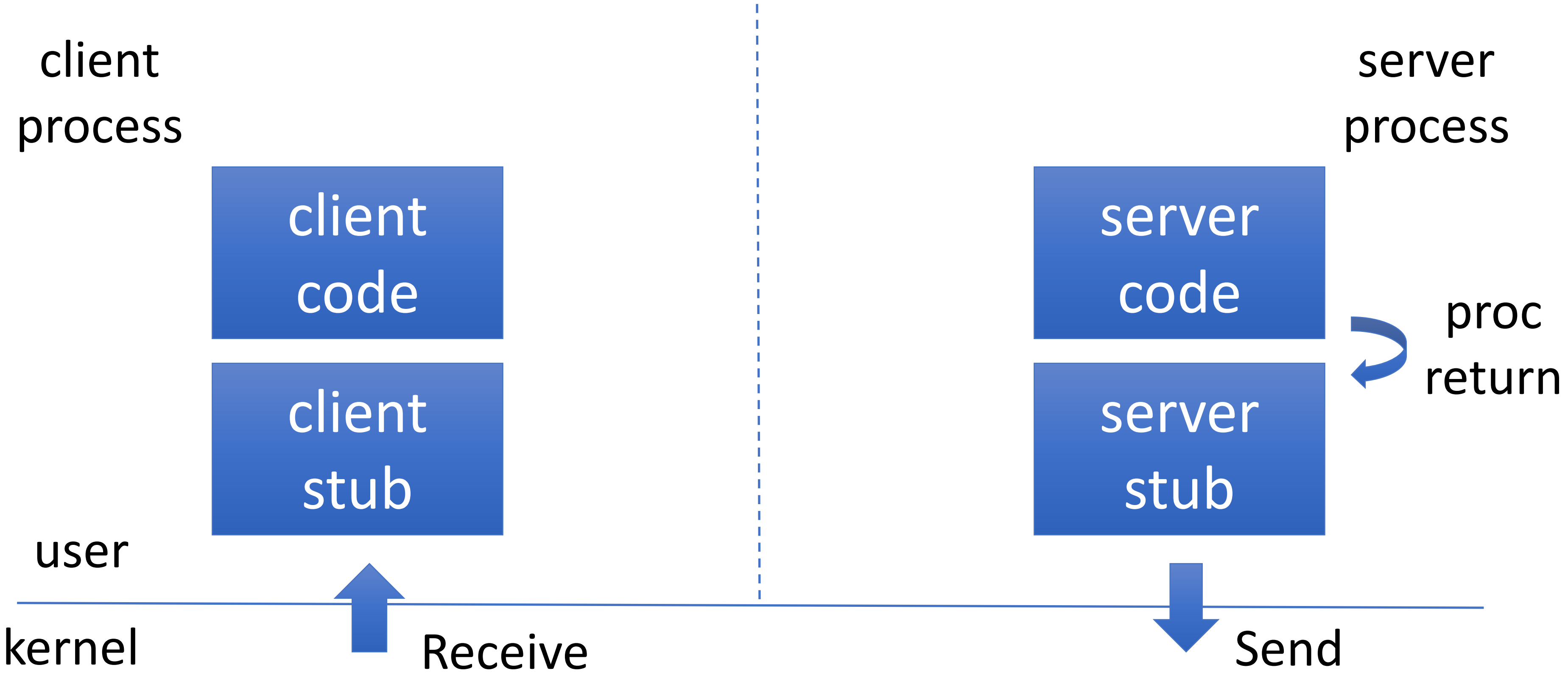
Recap Week 4 RPC Implementation: Return



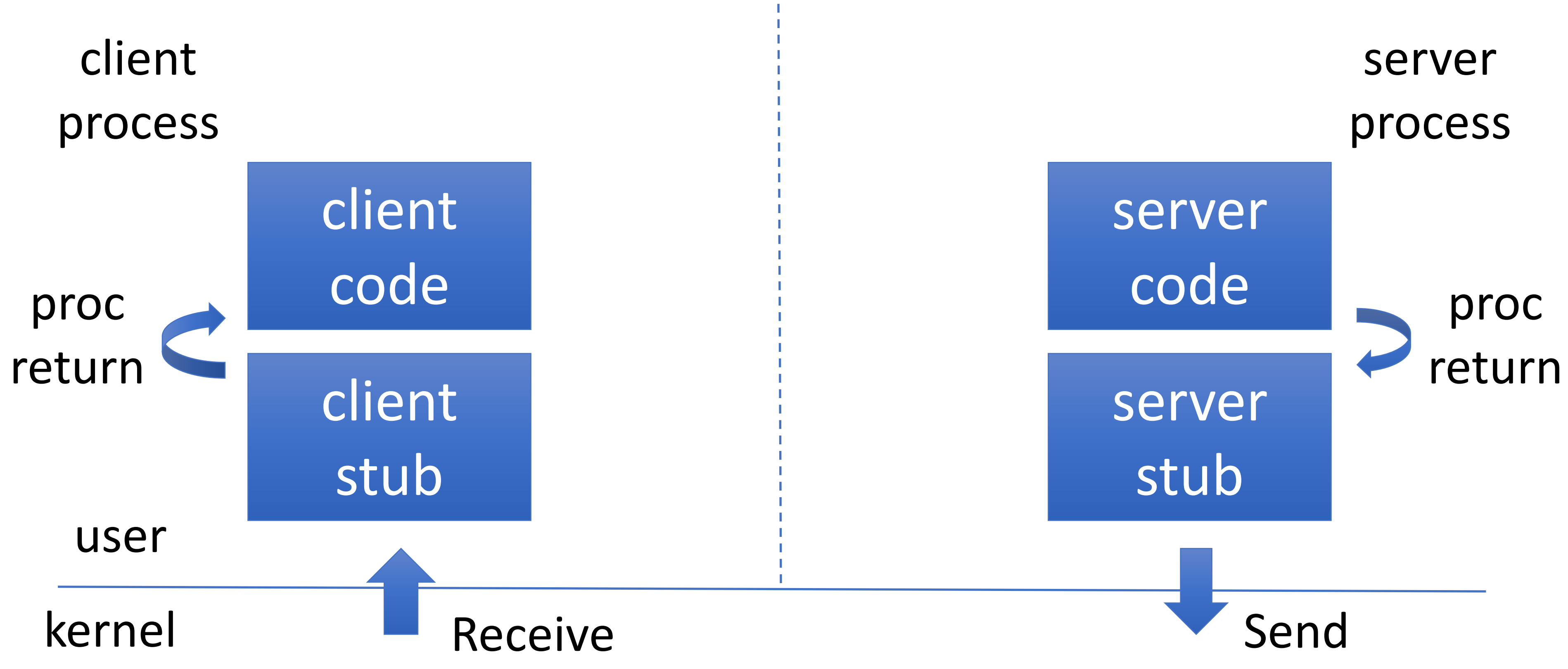
Recap Week 4 RPC Implementation: Return



Recap Week 4 RPC Implementation: Return



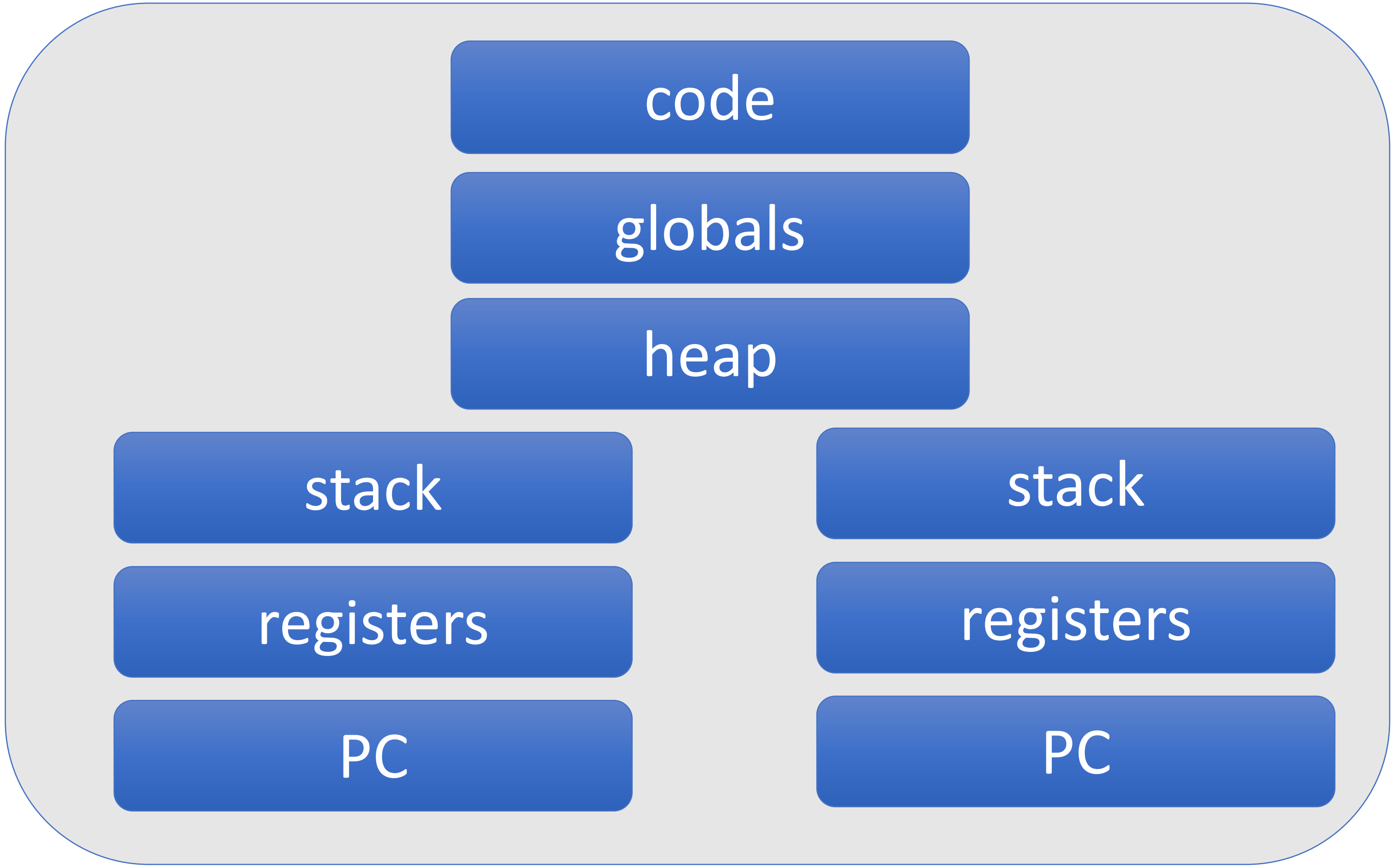
Recap Week 4 RPC Implementation: Return



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

```
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
  }
}
```

local data: i, c, my_min, my_max

Shared read-only data: a, b

Shared data: sum, prod

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
  }
}
```

Protect access to shared data with mutex

Shared data: sum, prod

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)
}
```

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

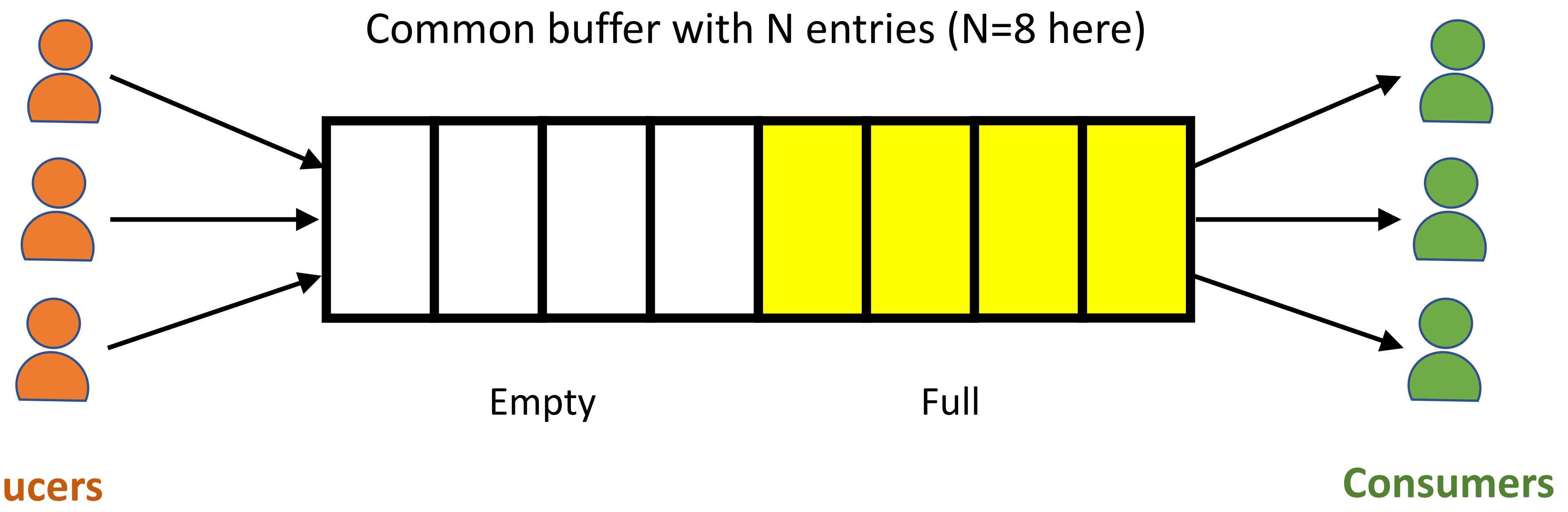
```
Threadcode() {
  int i, c
  local_sum = 0
  local_prod = 1

  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

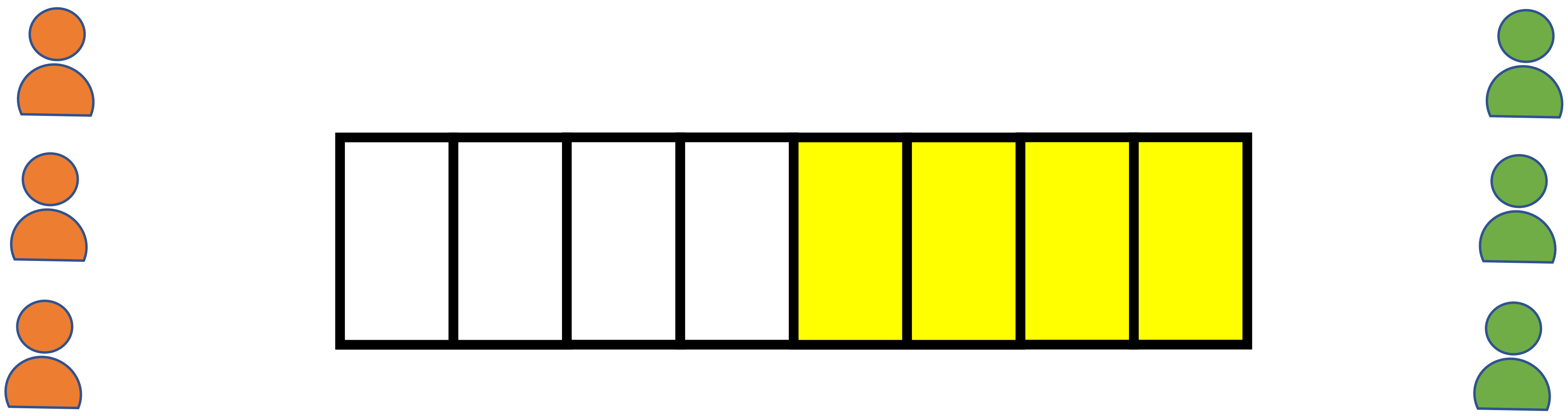


Producer/Consumer Problem

- 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 no entry is full.
  - Access to the buffer is **mutually exclusive.**

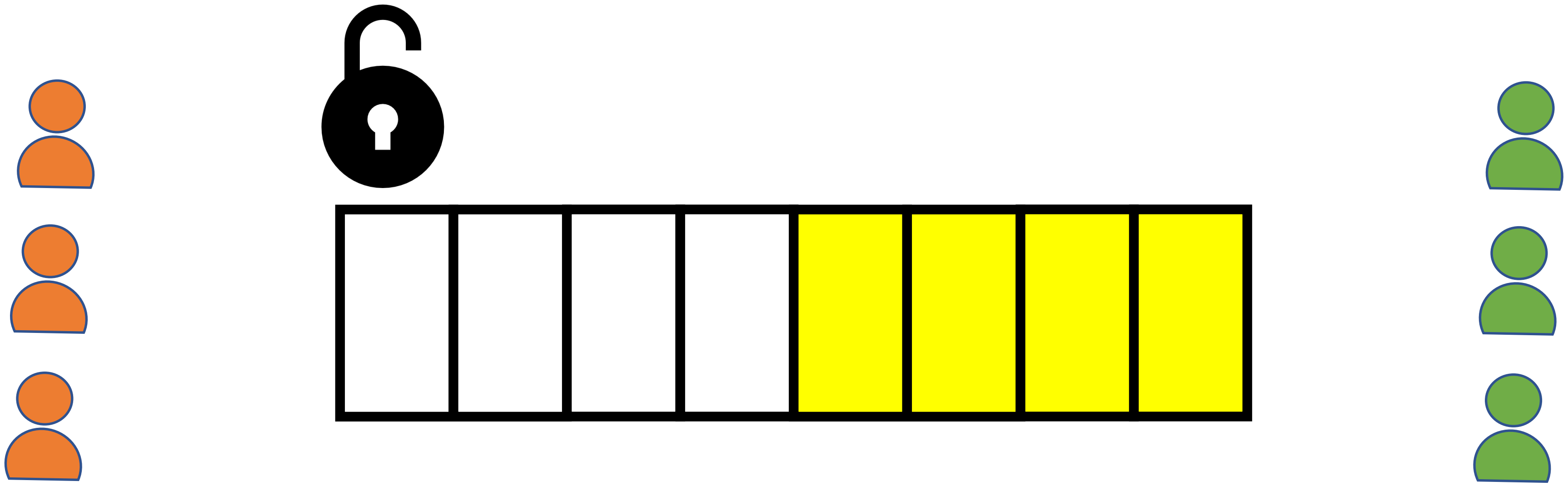
Solve Producer-Consumer with Locks?

Think!

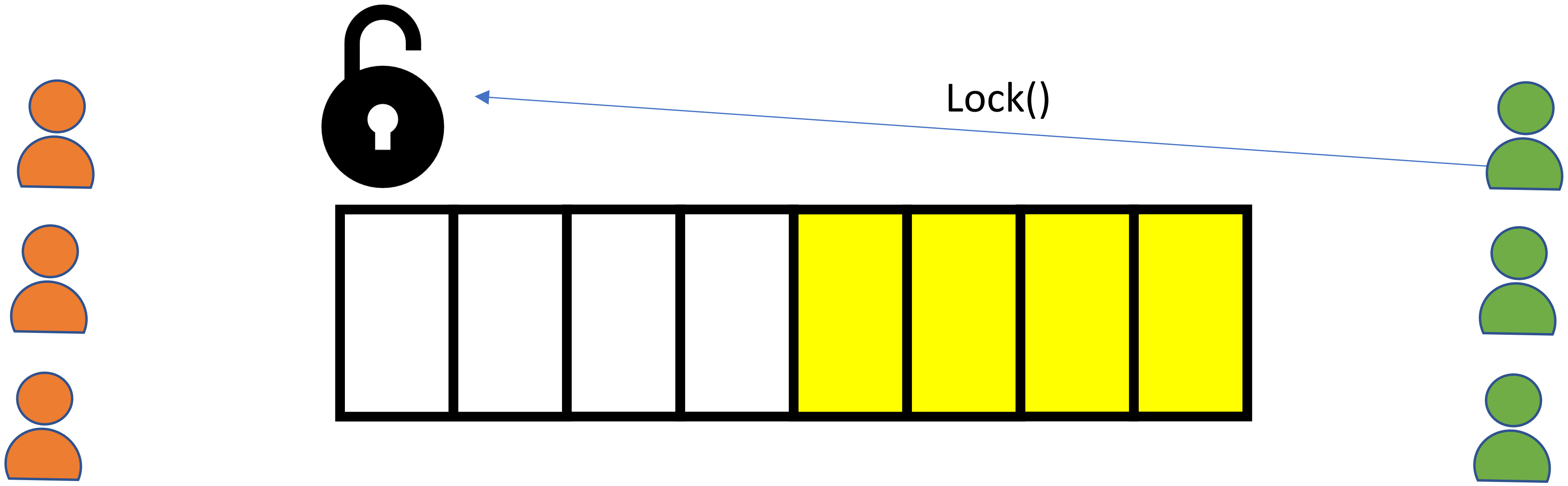


Solve Producer-Consumer with Locks? (Incorrect)

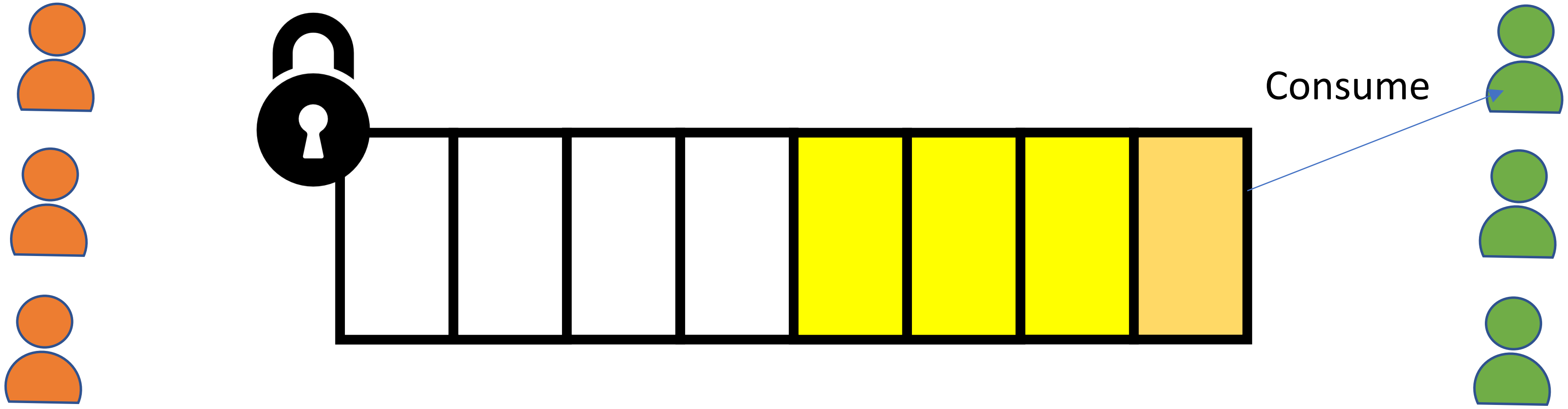
Mutex to ensure mutually exclusive access to buffer



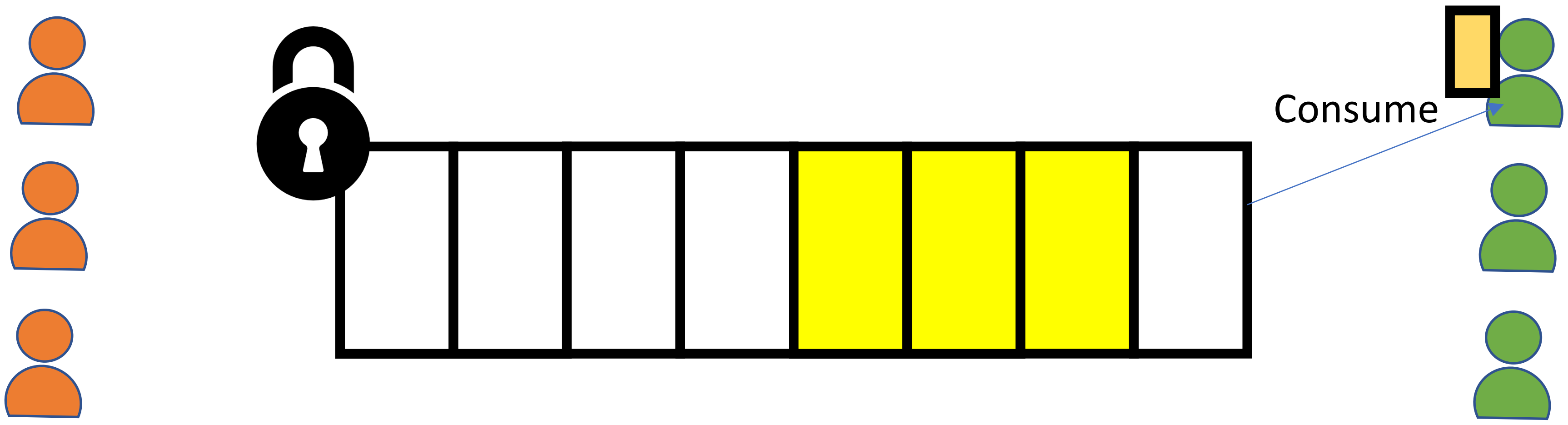
Solve Producer-Consumer with Locks? (Incorrect)



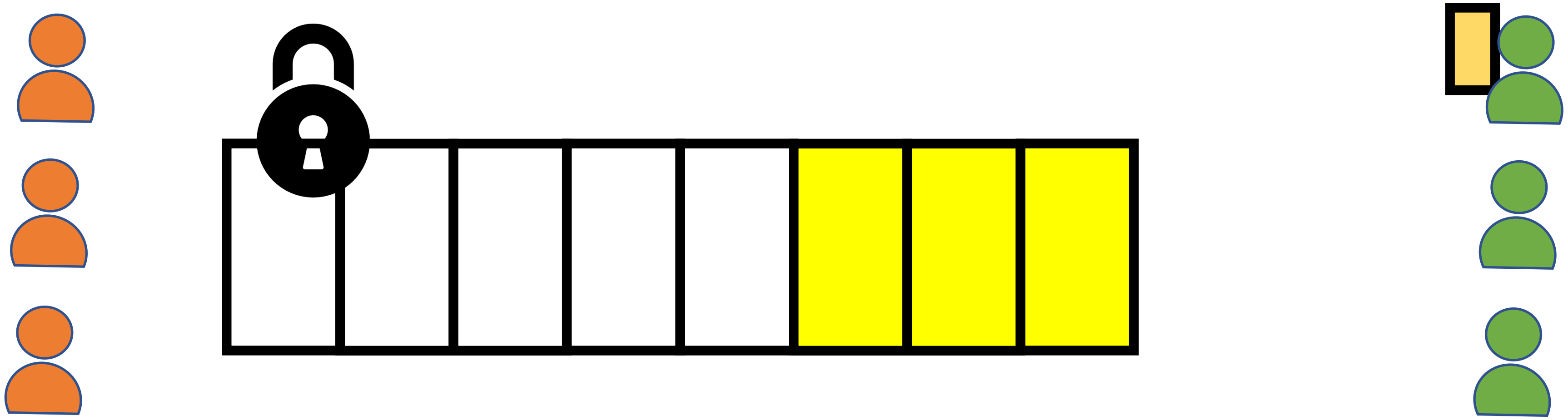
Solve Producer-Consumer with Locks? (Incorrect)



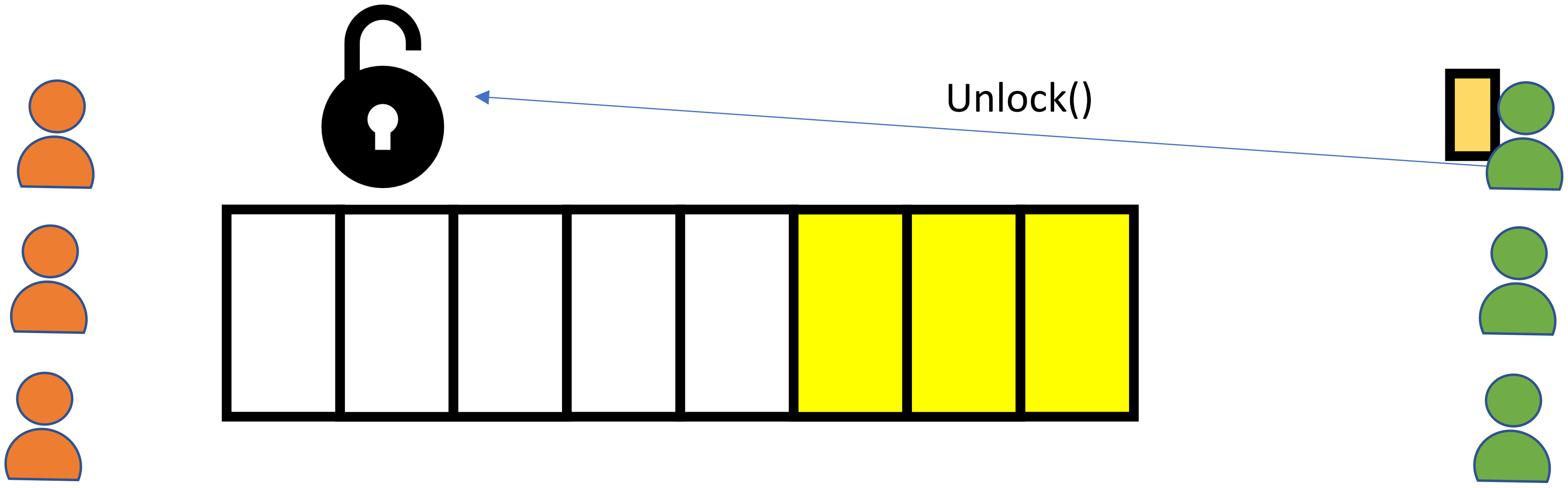
Solve Producer-Consumer with Locks? (Incorrect)



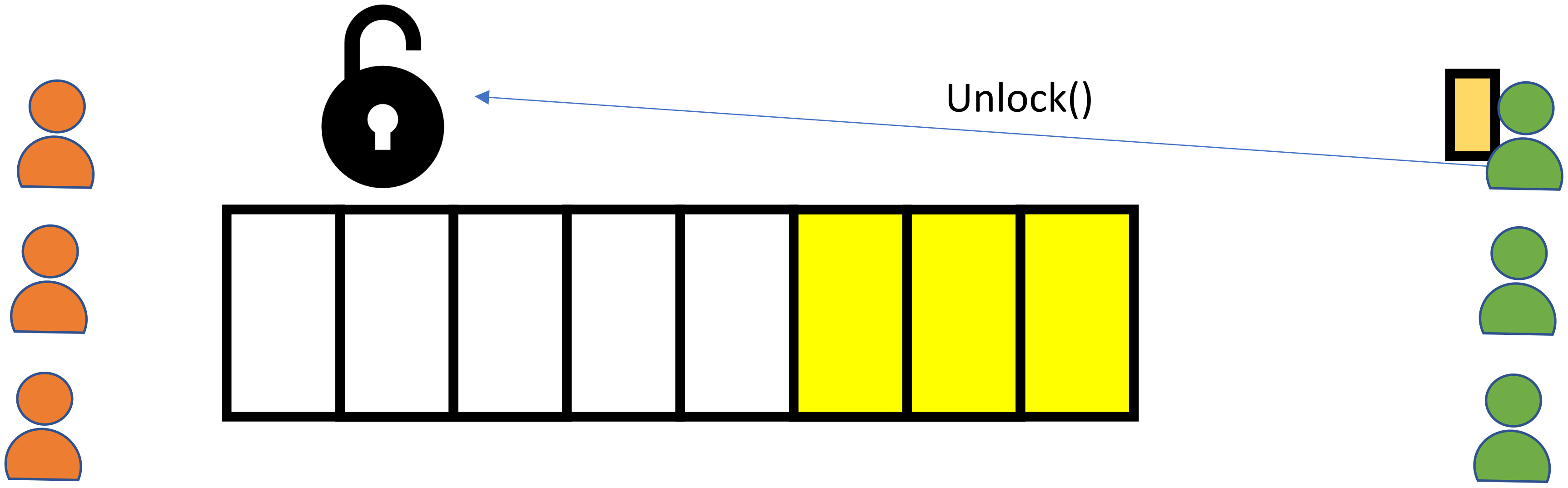
Solve Producer-Consumer with Locks? (Incorrect)



Solve Producer-Consumer with Locks? (Incorrect)



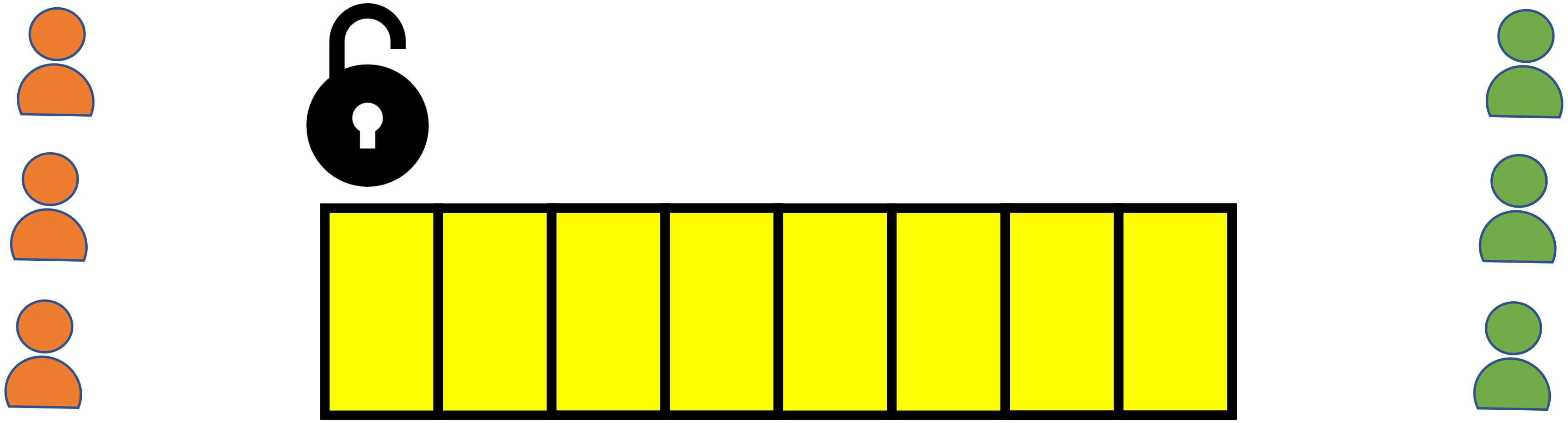
Solve Producer-Consumer with Locks? (Incorrect)



Problem? Think!

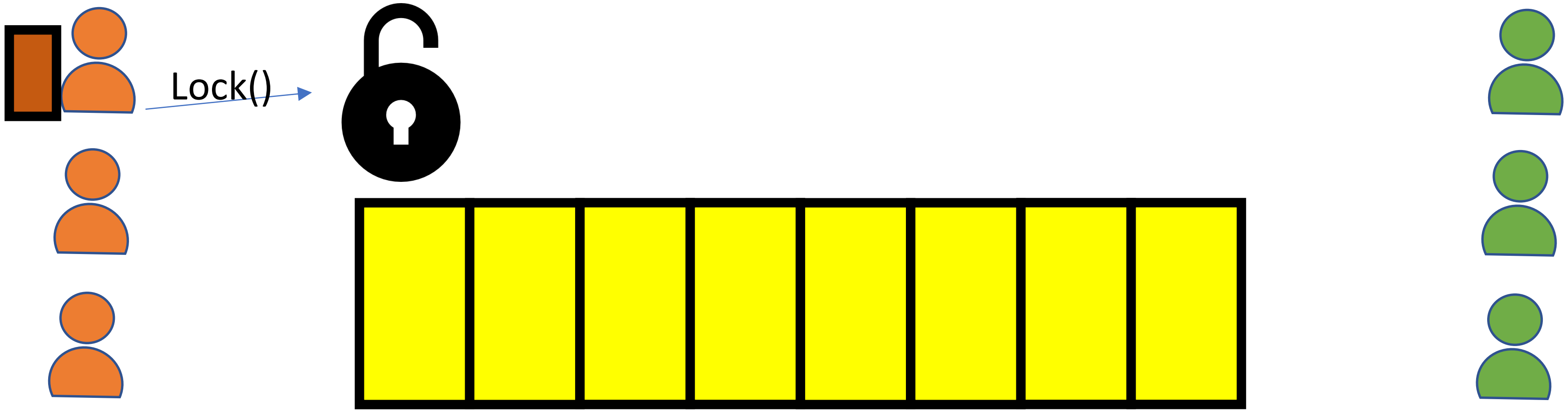
Problem: Threads can’t know the state of the buffer before acquiring the lock

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



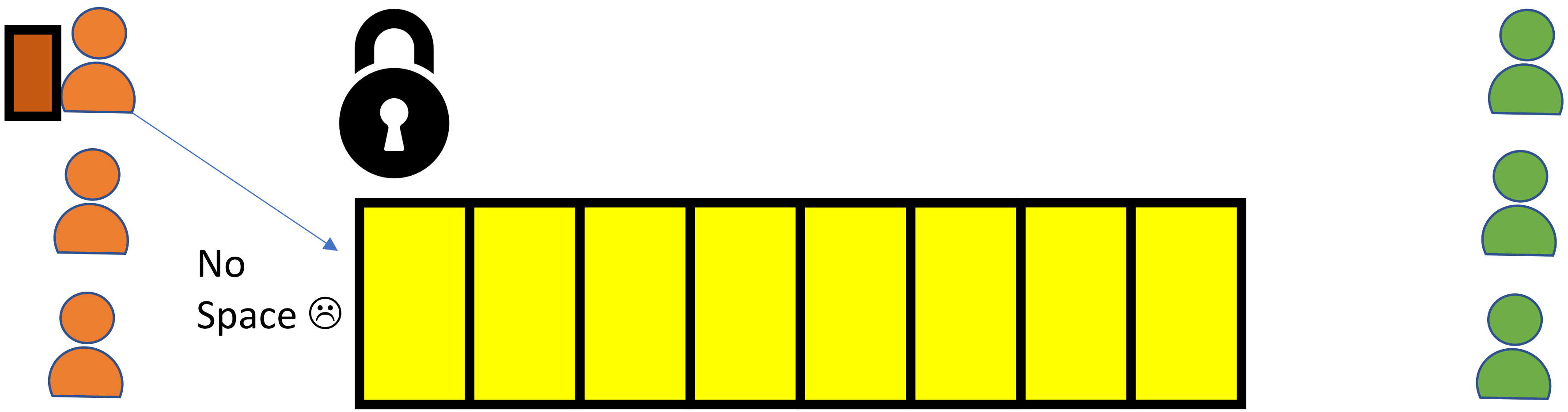
Problem: Threads can't know the state of the buffer before acquiring the lock

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



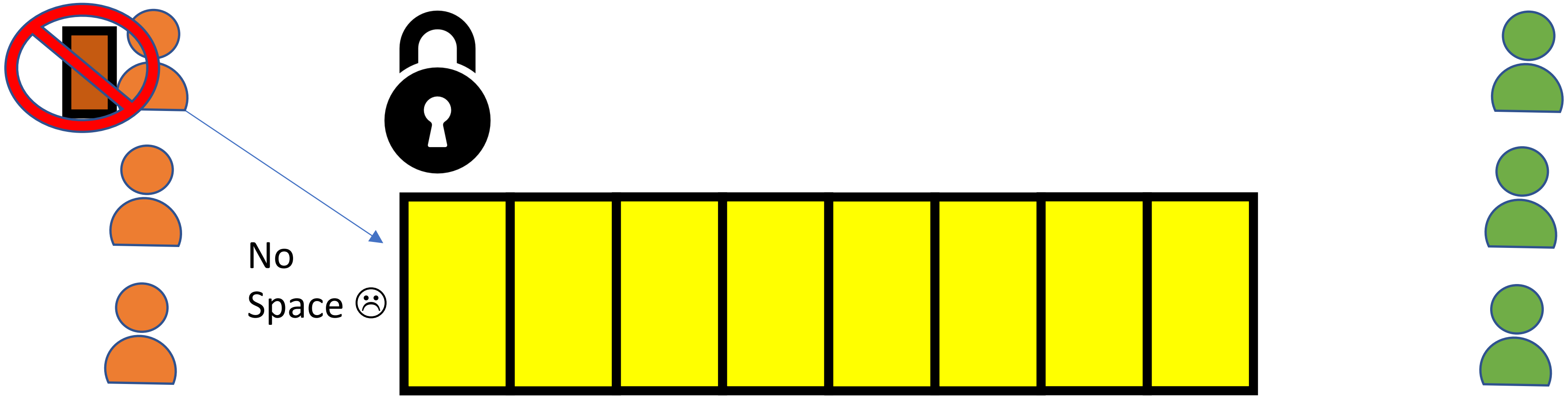
Problem: Threads can't know the state of the buffer before acquiring the lock

- 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

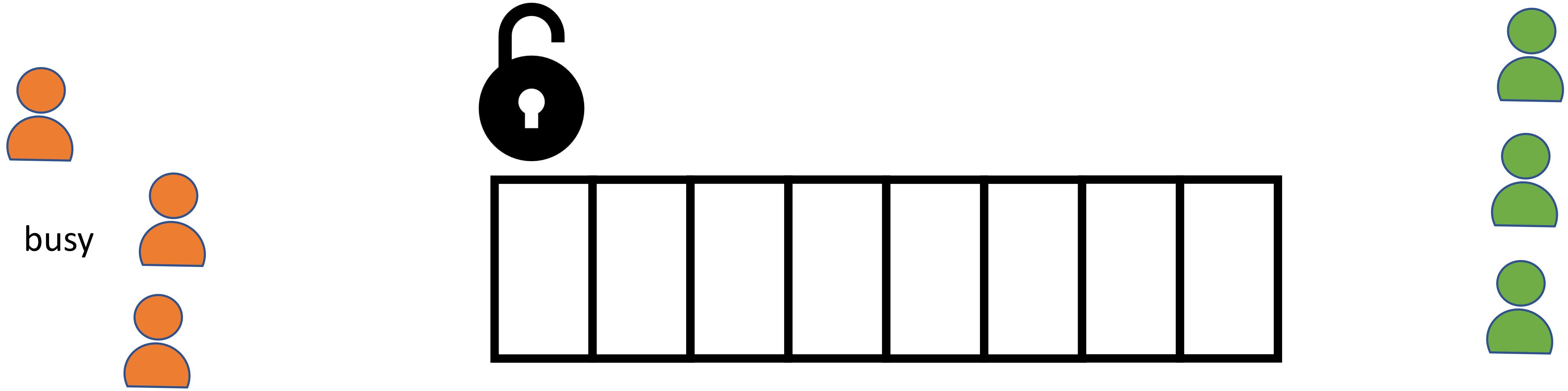
- Problem 1: Buffer is full and producer want to add entries



Packet needs to be dropped and problem requirement is violated
→ “No production when all N entries are full.”

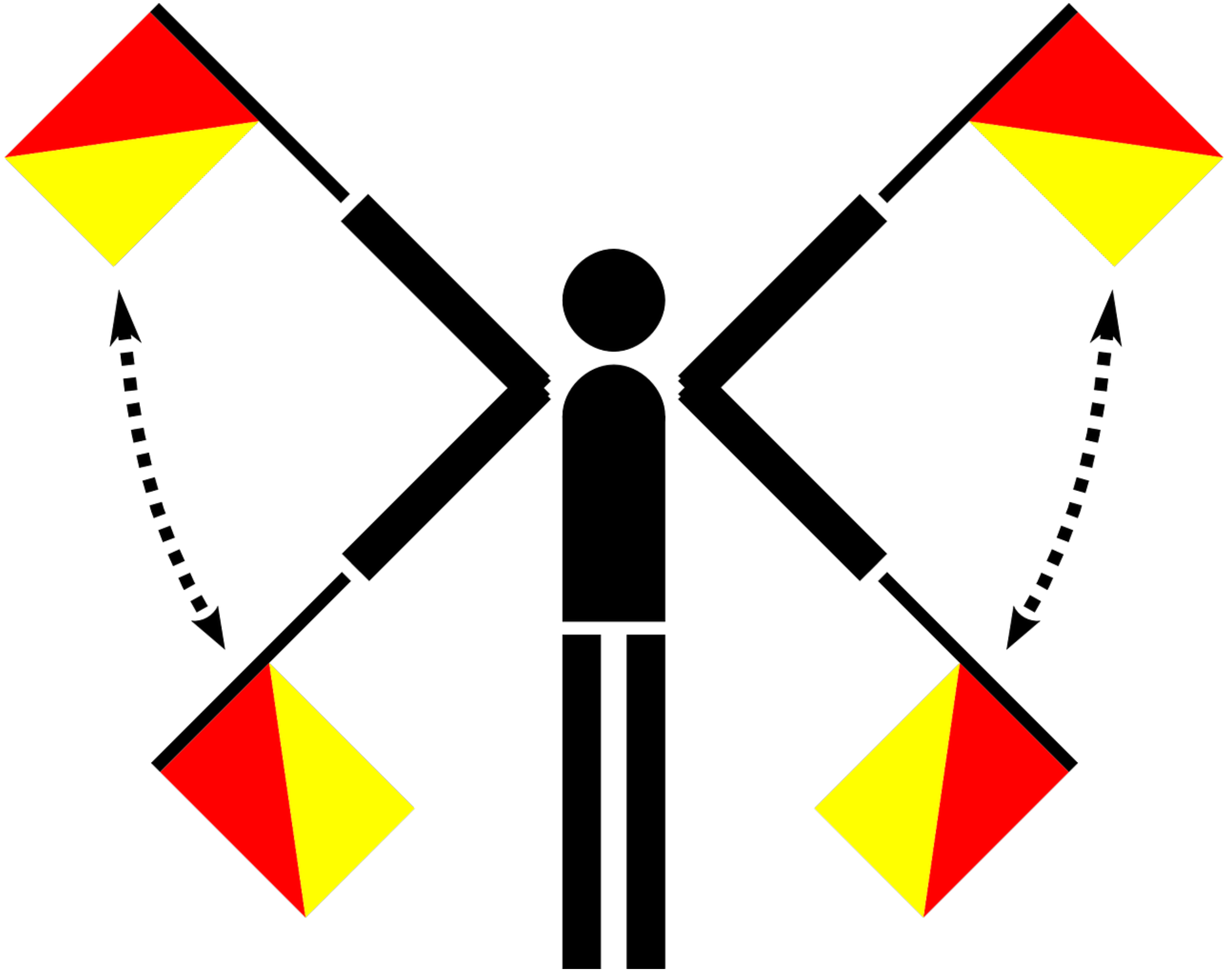
Problem: Threads can't know the state of the buffer before acquiring the lock

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

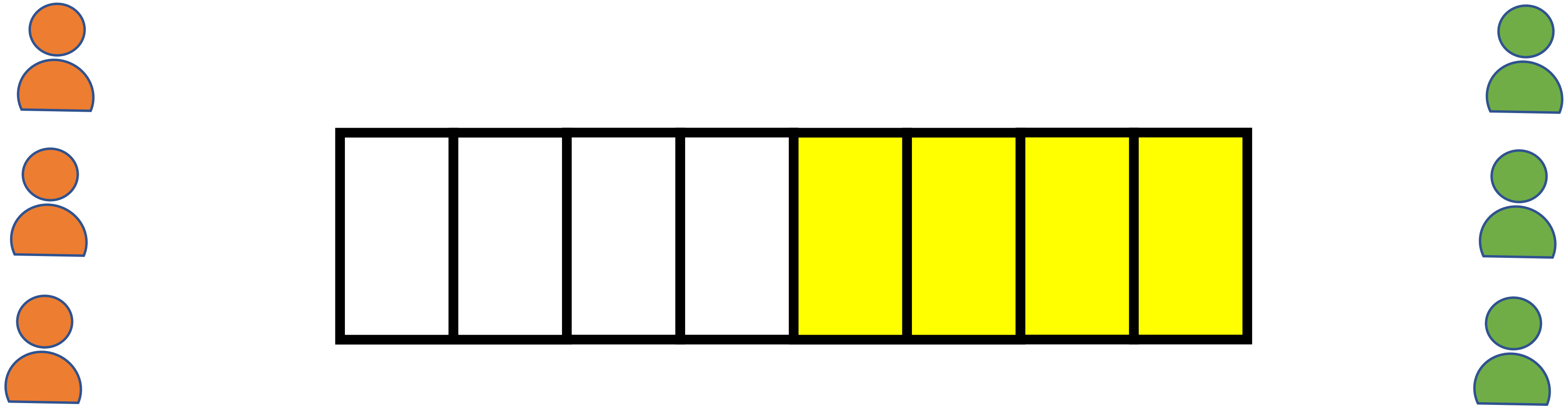


Problem requirement is *not* violated, but inefficient
Consumers waste CPU cycles locking the buffer just to see that it's empty

Solution: Semaphores

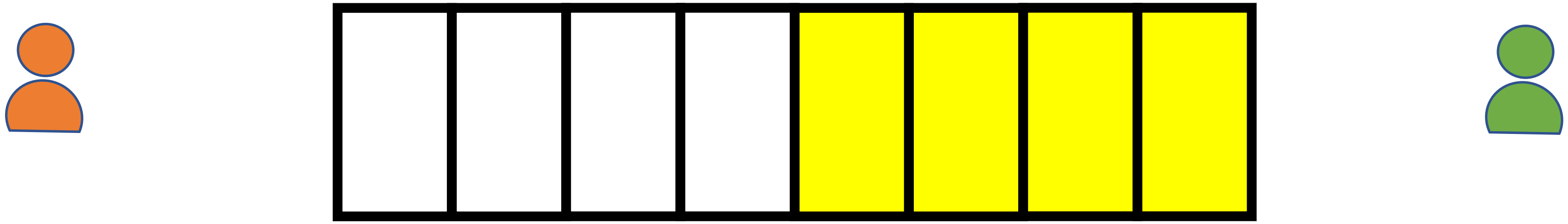


Solve Producer-Consumer with Semaphores?



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

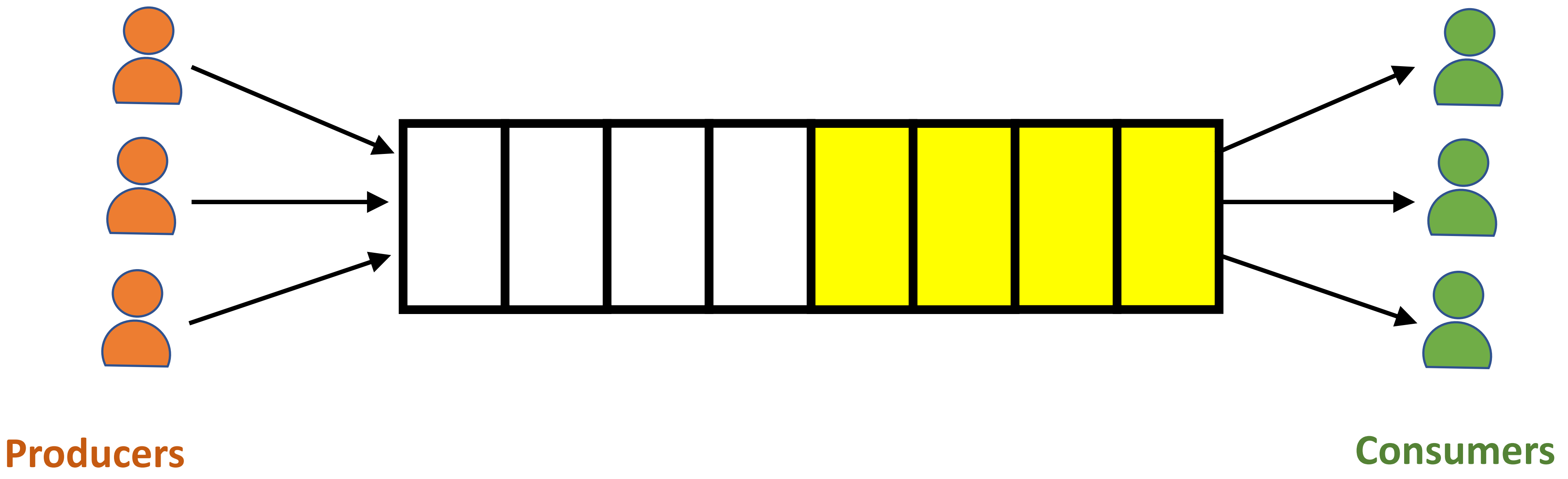
Producer

```
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

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, queuelock)
    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, queuelock)
    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, queuelock)
    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, queuelock)
    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 (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, queuelock)
    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

```
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, queuelock)
    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

Worker thread W2 runs and takes something out of queue

Sets avail to 0

Now W1 runs

It must check the value of avail again

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
  - **It must check the value of avail again**

Answer: No
Remember pthreads Condition Variables

- Pthread_cond_wait(cond, mutex)
  - Wait for a signal on cond
  - Release mutex
- Pthread_cond_signal(cond, mutex)
- Pthread_cond_broadcast(cond, mutex)

- Must hold mutex when calling any of these!

Answer: No
Remember pthreads Condition Variables

- Pthread_cond_wait(cond, mutex)
- Pthread_cond_signal(cond, mutex)
  - Signal one thread waiting on cond
  - Signaled thread re-acquires mutex
    - immediately
  - If no thread waiting, no-op
- Pthread_cond_broadcast(cond, mutex)

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, queuelock)
    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 )
  }
}
```

Summary

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

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).