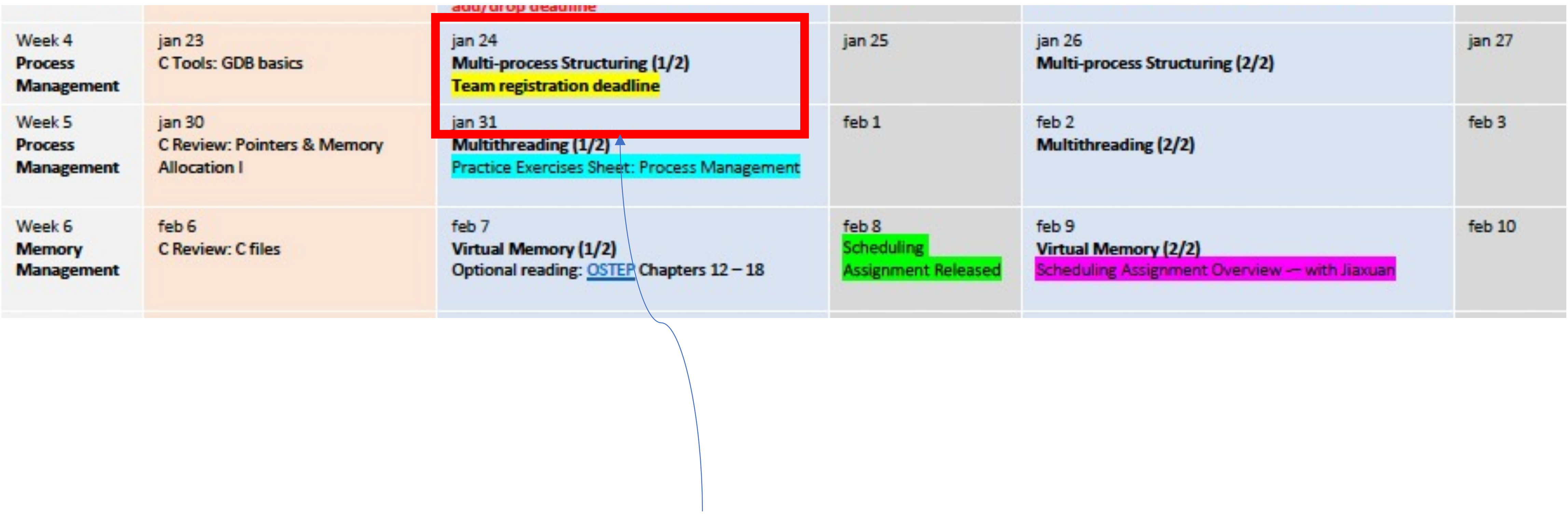
Week 4

Multiprocess Communication

Oana Balmau
January 24, 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)** Practice Exercises Sheet: Process Management	feb 1	feb 2 **Multithreading (2/2)**	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

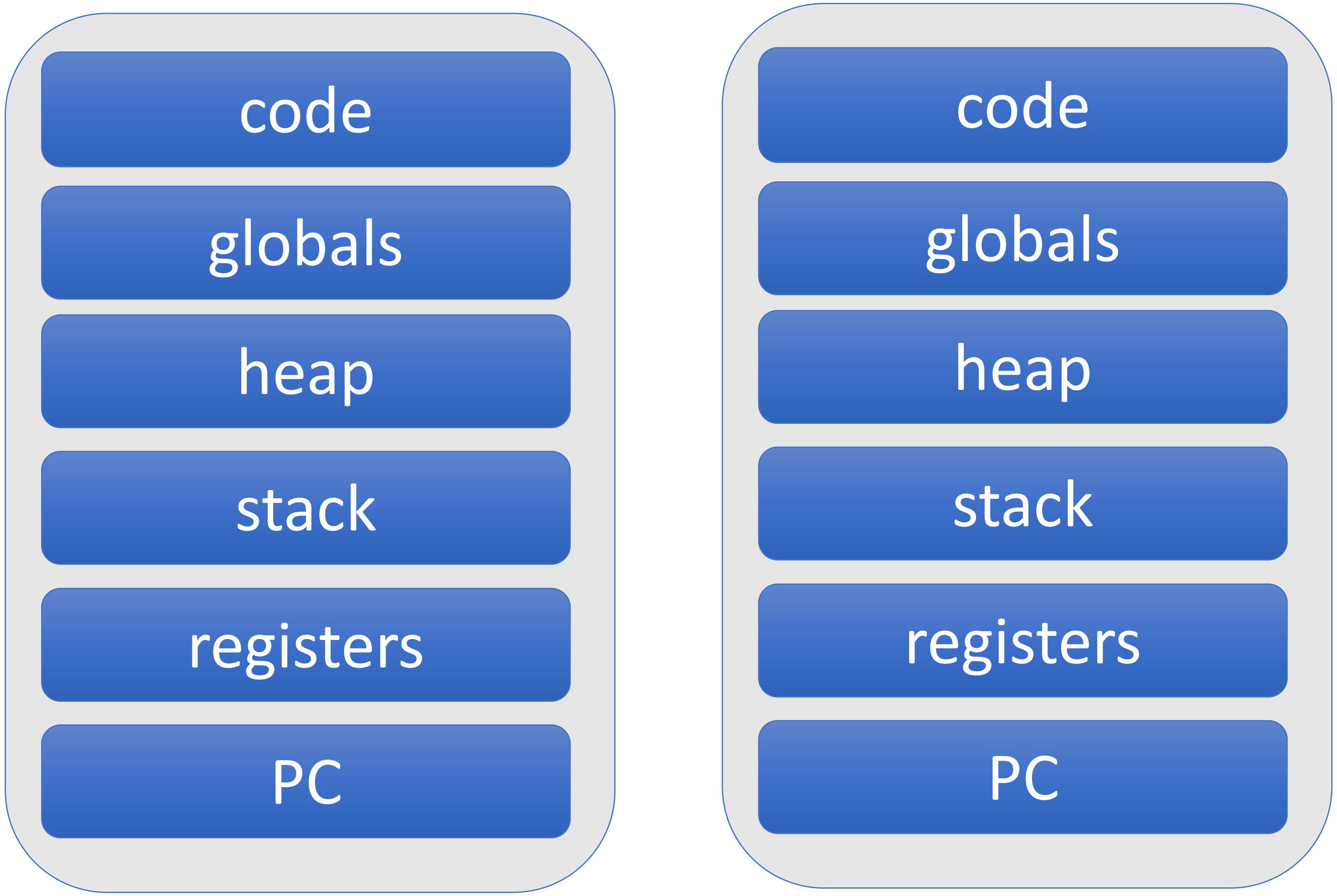
Please sign up teams of 1 as well!

Recap Week 3
Concurrency – Option 1

- Build apps from many communicating **processes**
- Communicate through message passing
  - No shared memory
- Pros
  - If one process crashes, other processes unaffected
- Cons
  - High communication overheads
  - Expensive context switching

This week's focus

Recap Week 3
Two Processes

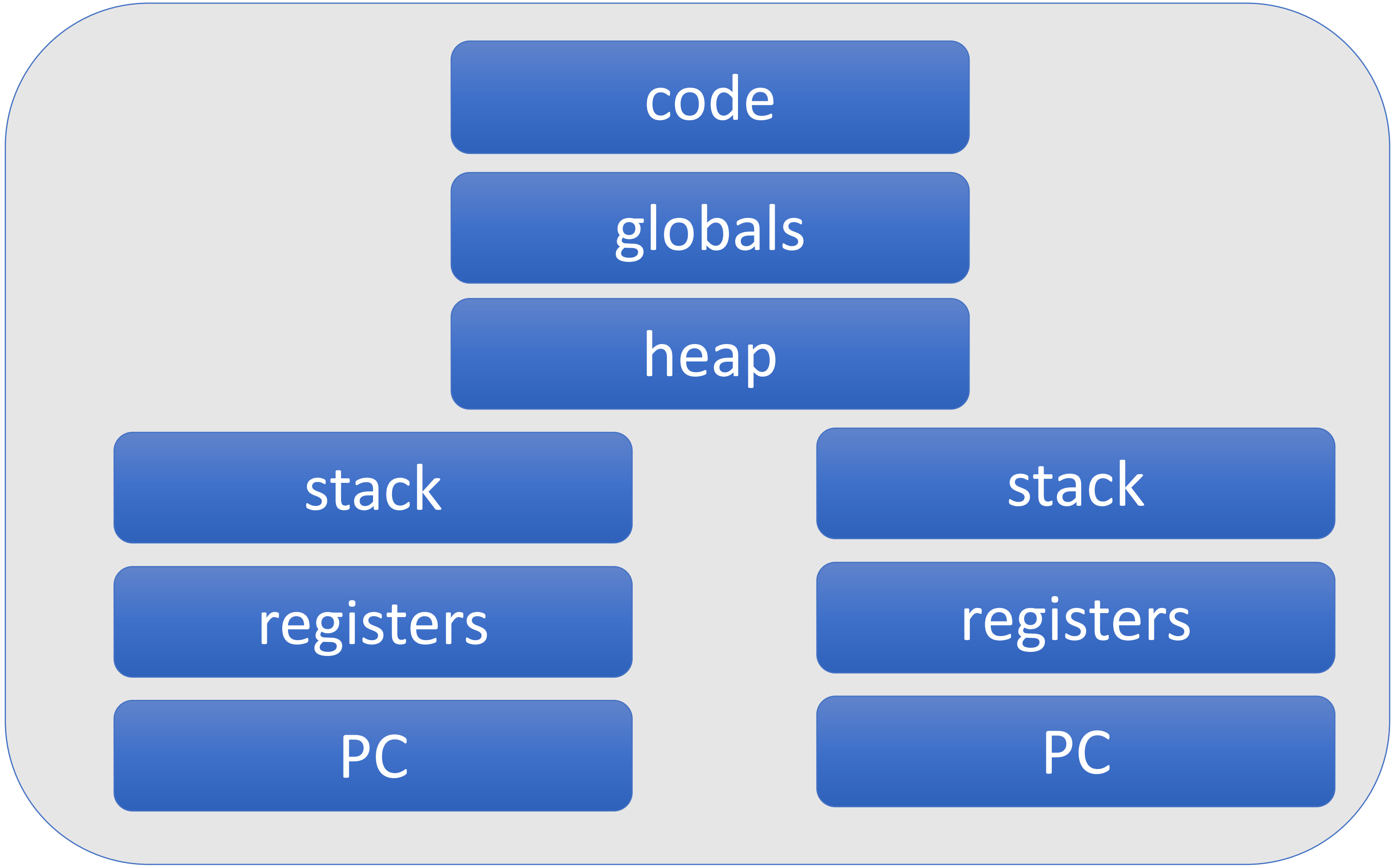


Recap Week 3
Concurrency – Option 2

- 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

Will see practical examples in Week 5

Recap Week 3
Two Threads in a Process



Key Concepts for Today

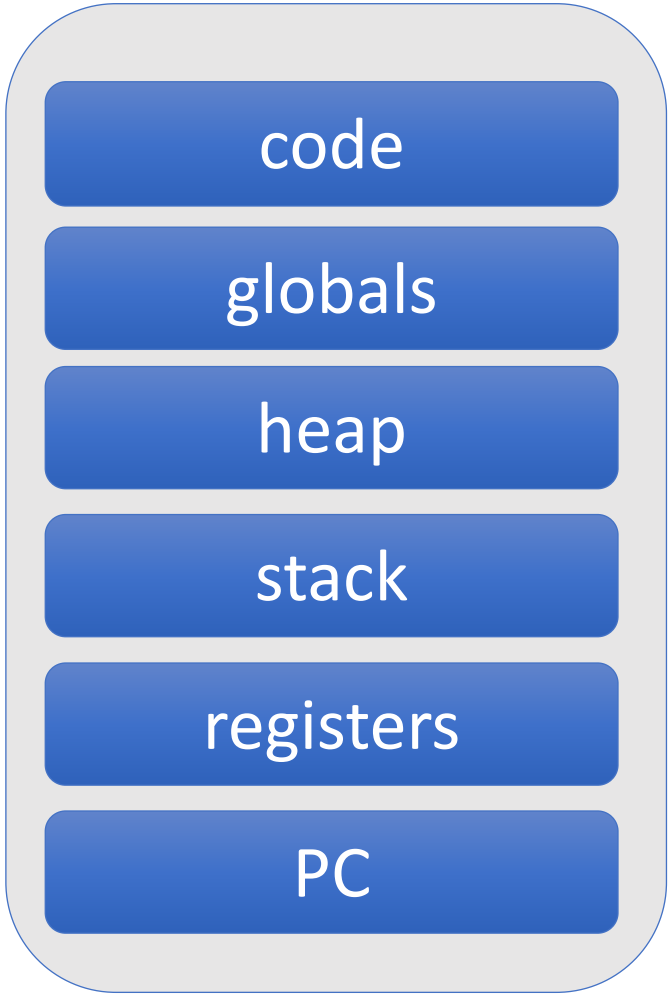
- Interprocess communication
- Message passing
- Remote procedure call (RPC)

So far

- One program

  = one process

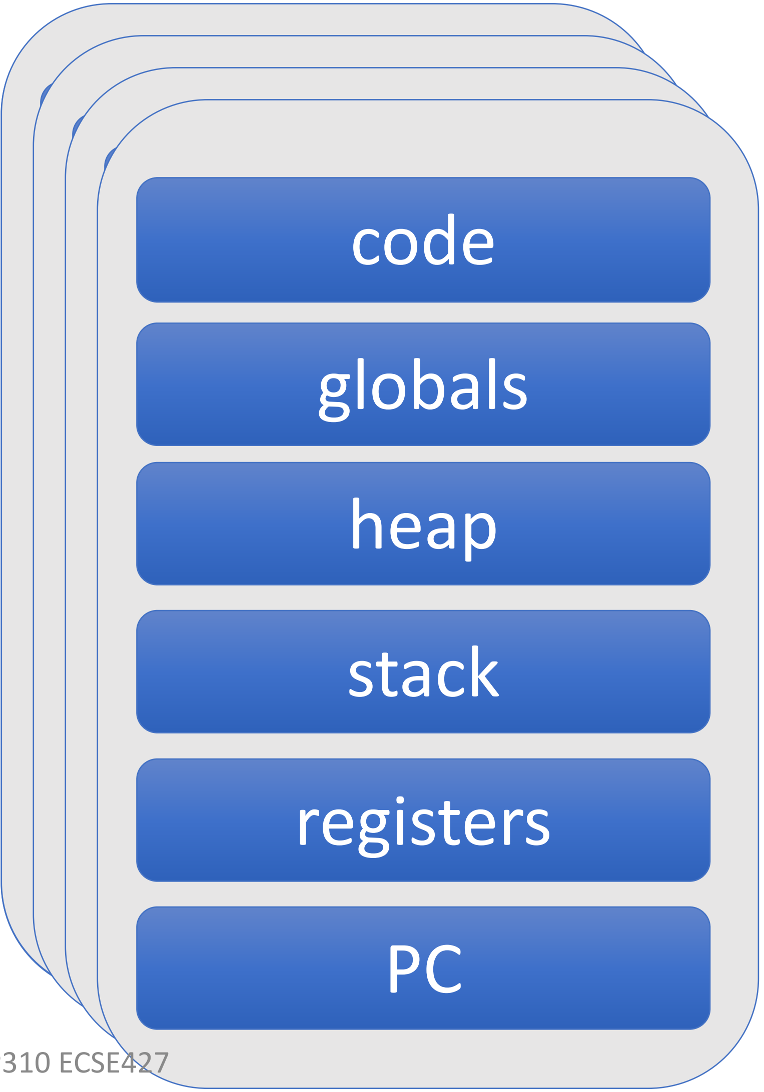
- Examples:
  - Shell
  - Compiler
  - …



This is not always the case

- One program

  = multiple processes
- Example:
  - Web server

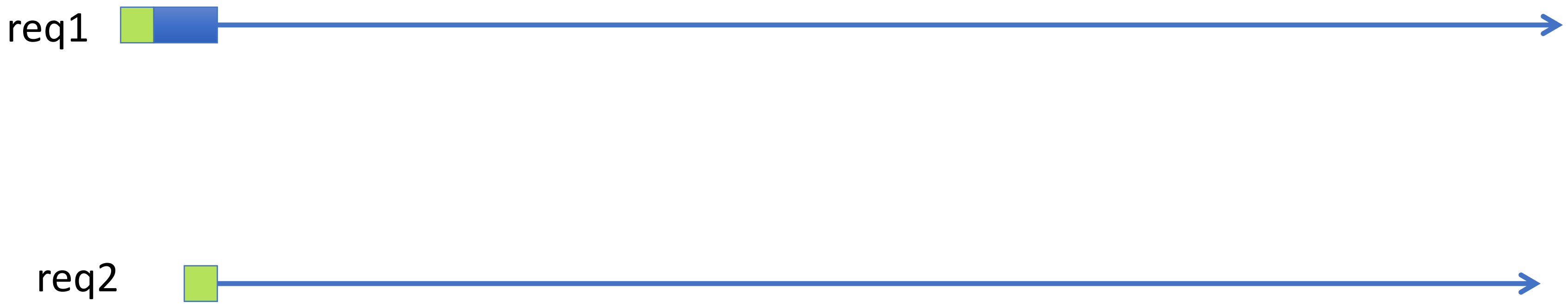


(Very Simple) Web Server

```
WebServerProcess {
    forever {
        wait for an incoming request
        read file from disk
        send file back in response
    }
}
```

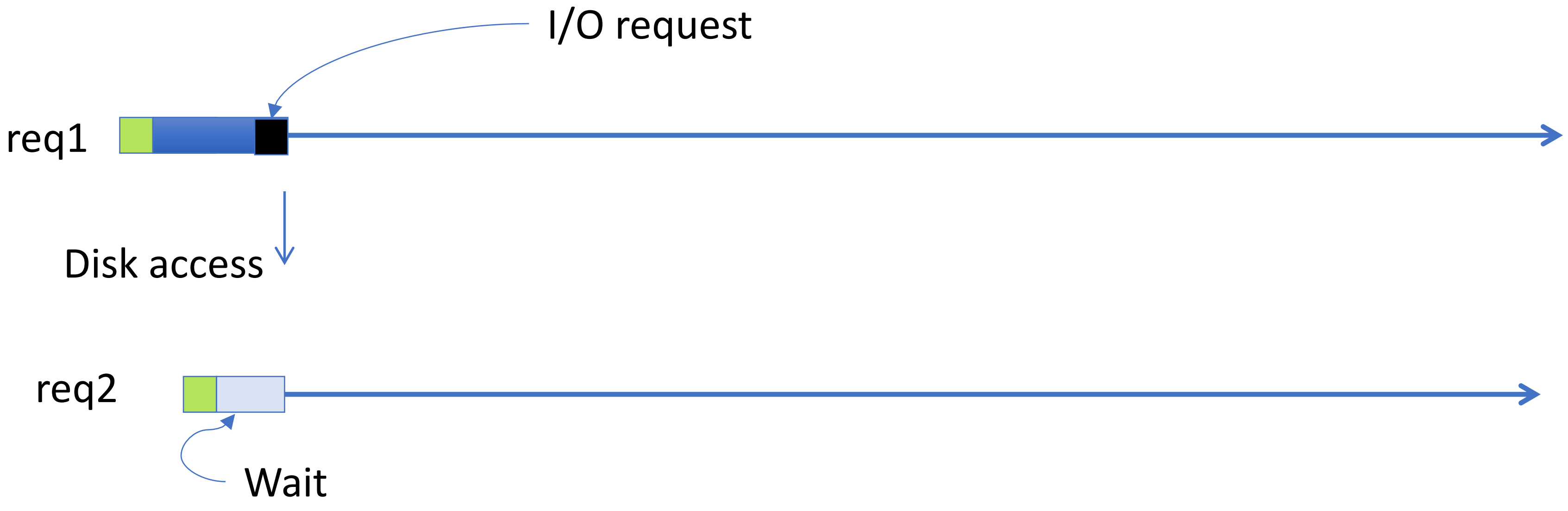
Single-Process Web Server

Example: Web server receives two requests in quick succession



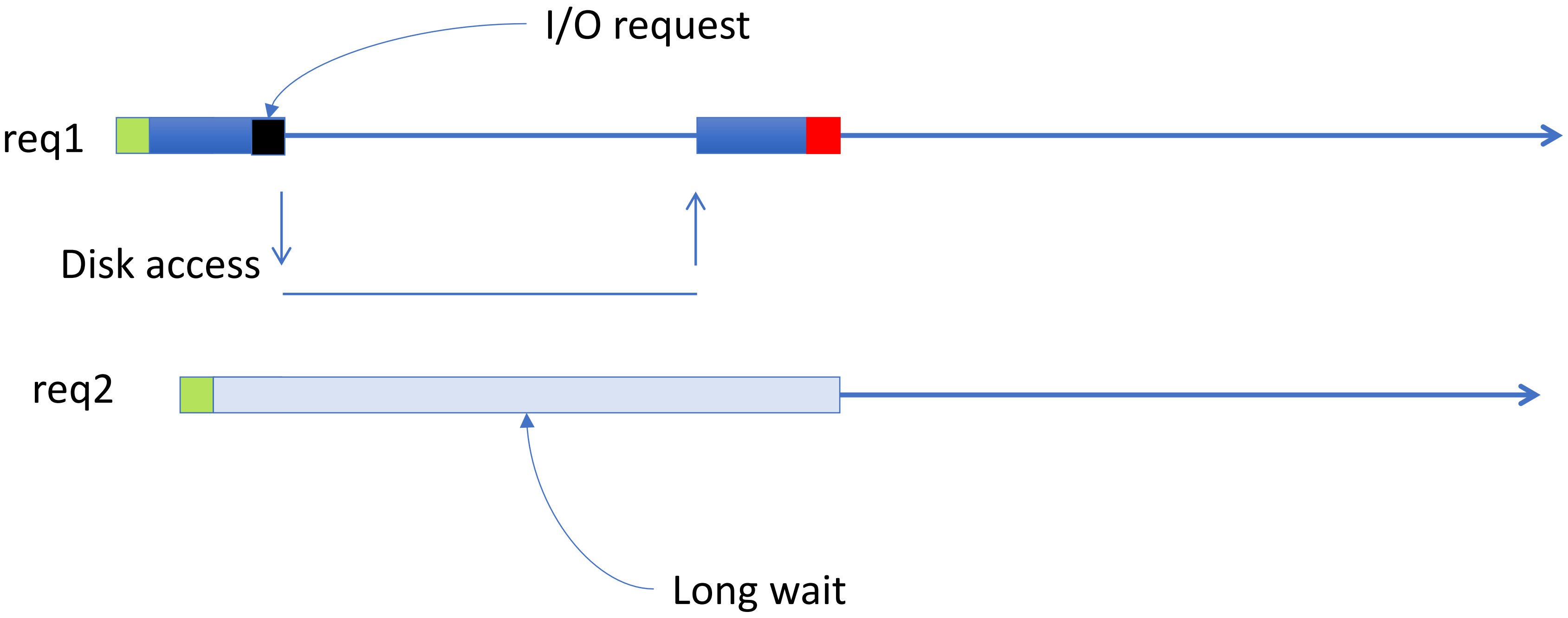
Single-Process Web Server

Example: Web server receives two requests in quick succession



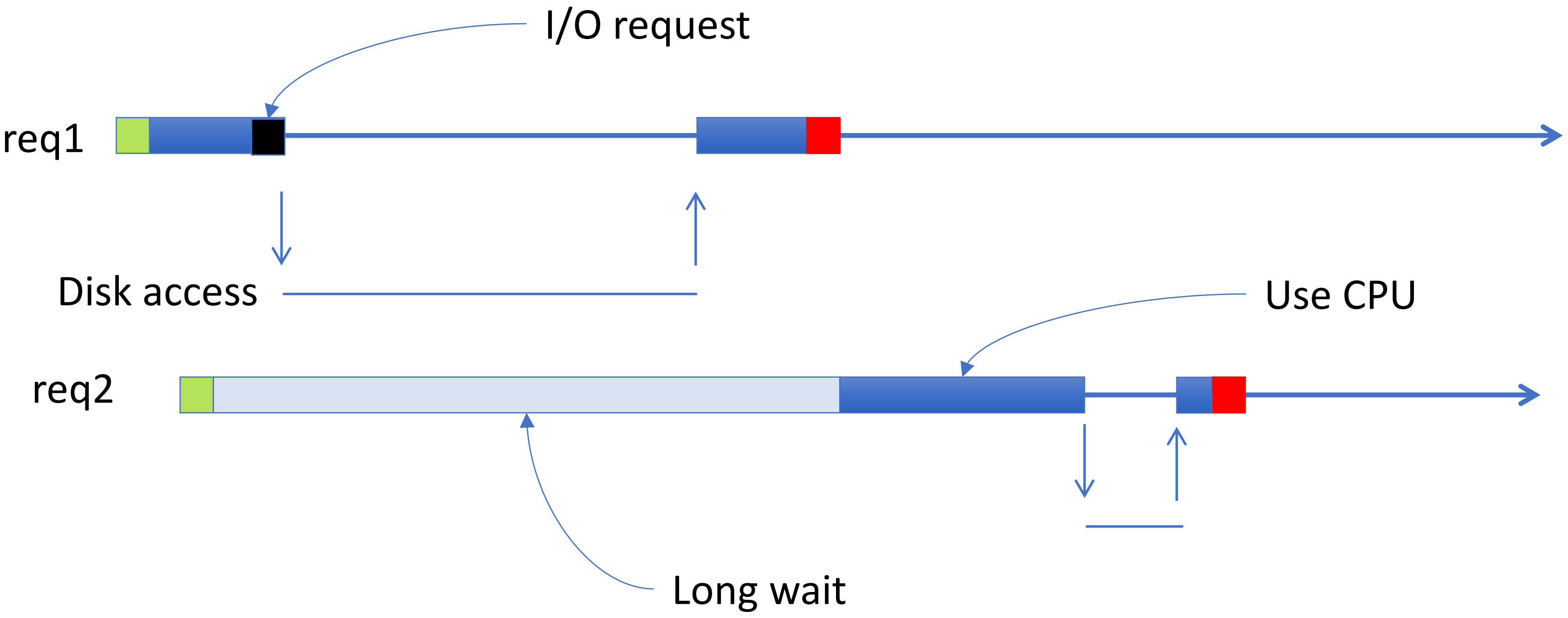
Single-Process Web Server

Example: Web server receives two requests in quick succession



Single-Process Web Server

Example: Web server receives two requests in quick succession



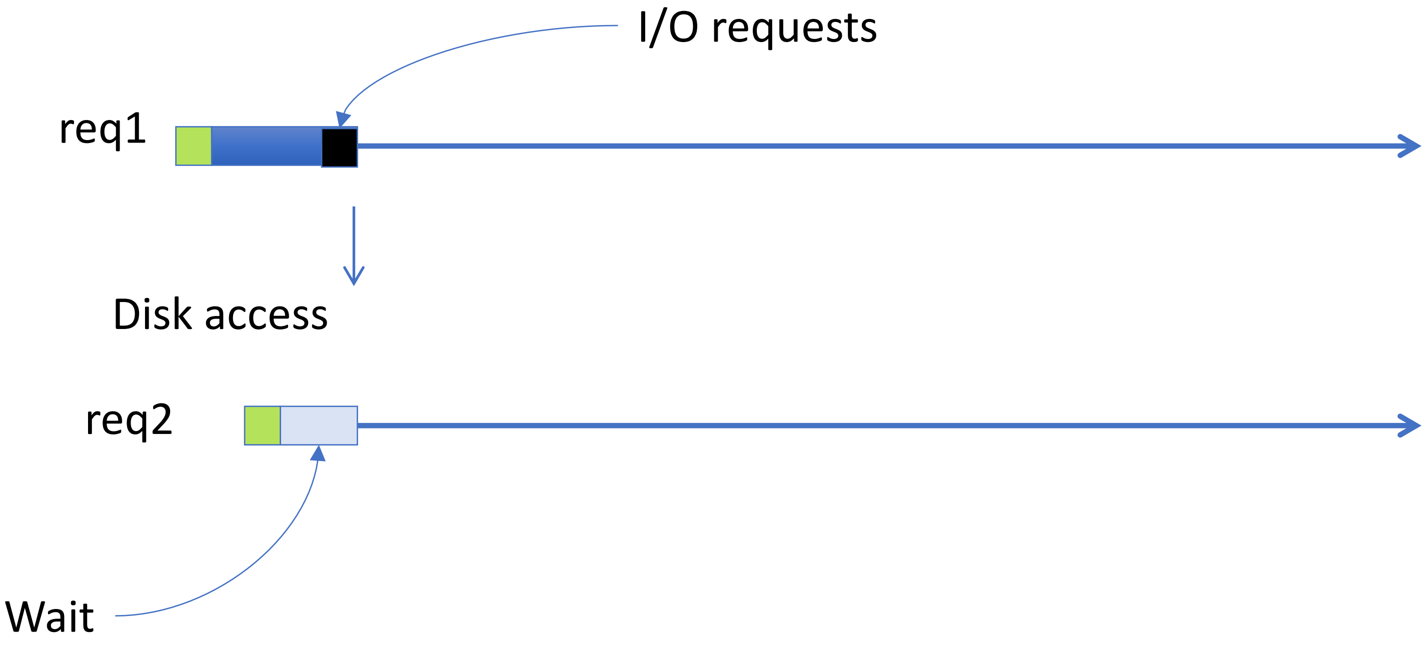
Multiprocess Web Server

```
ListenerProcess {
  forever {
    wait for incoming request
    CreateProcess( worker, request )
  }
}

WorkerProcess(request) {
  read file from disk
  send response
  exit
}
```

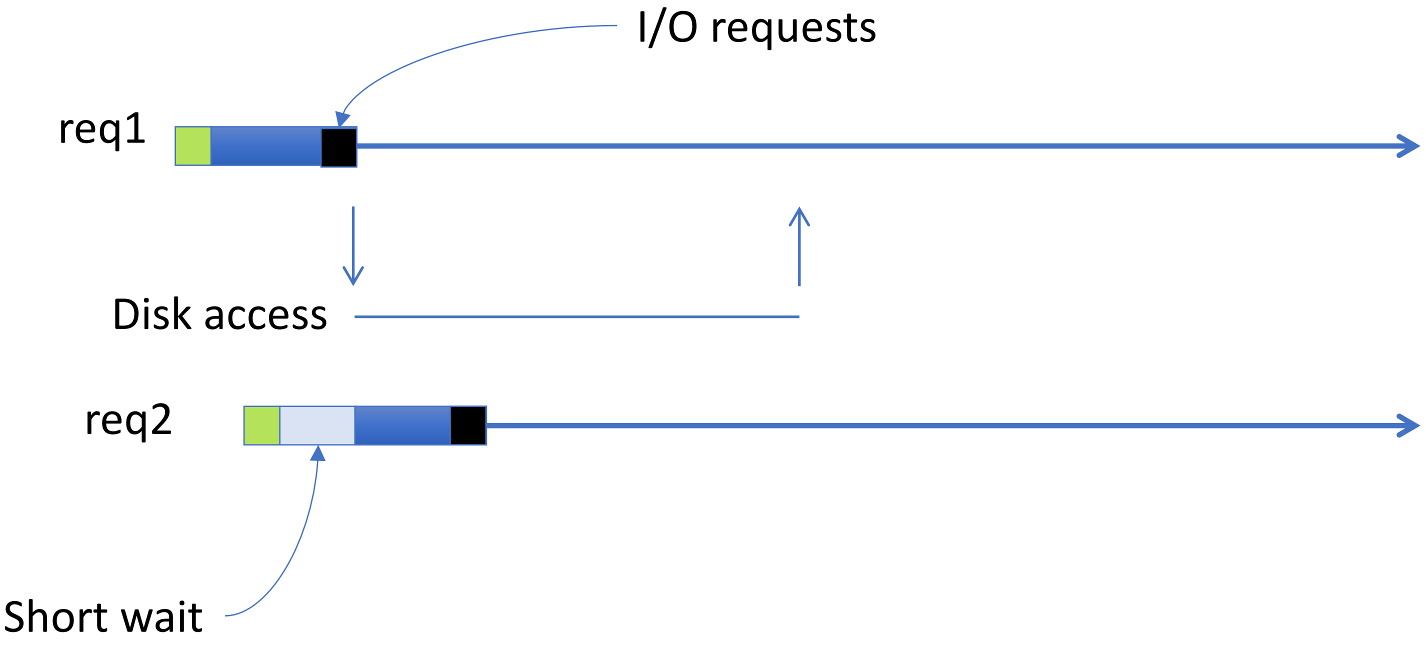
Multi vs. Single-process Web Server

Example: Web server receives two requests in quick succession



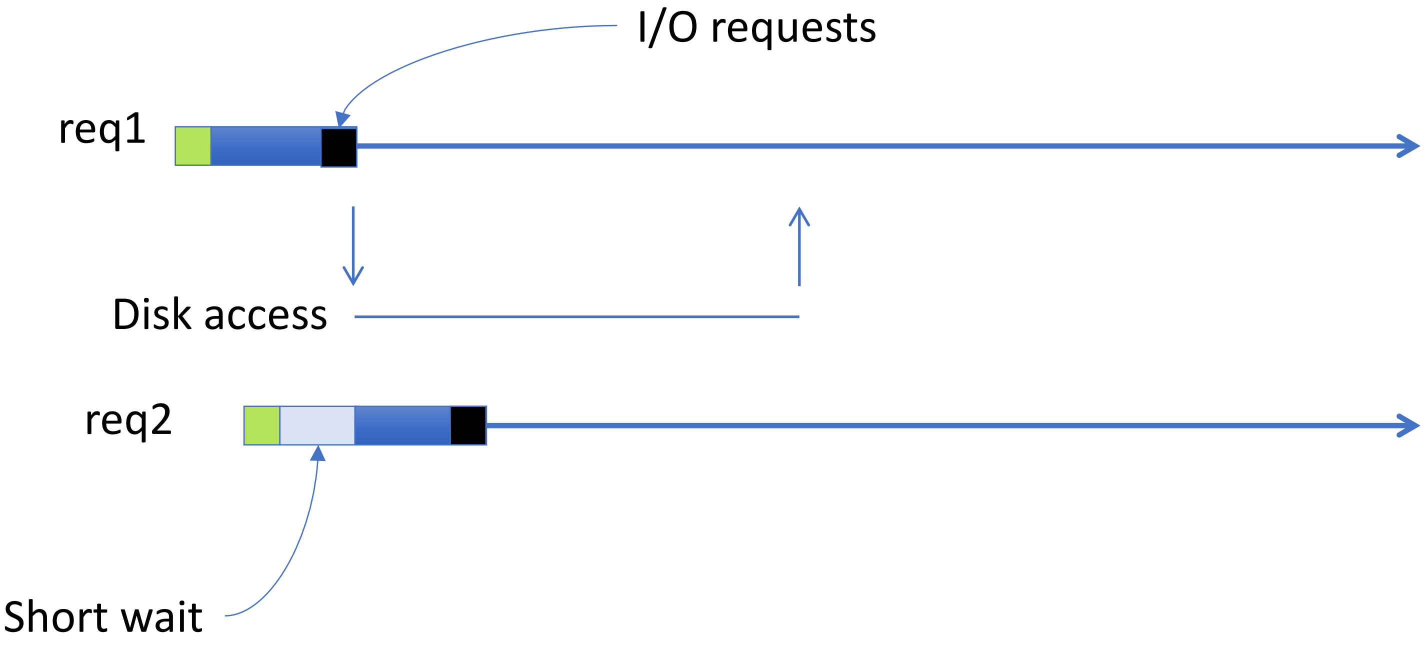
Multi vs. Single-process Web Server

Example: Web server receives two requests in quick succession



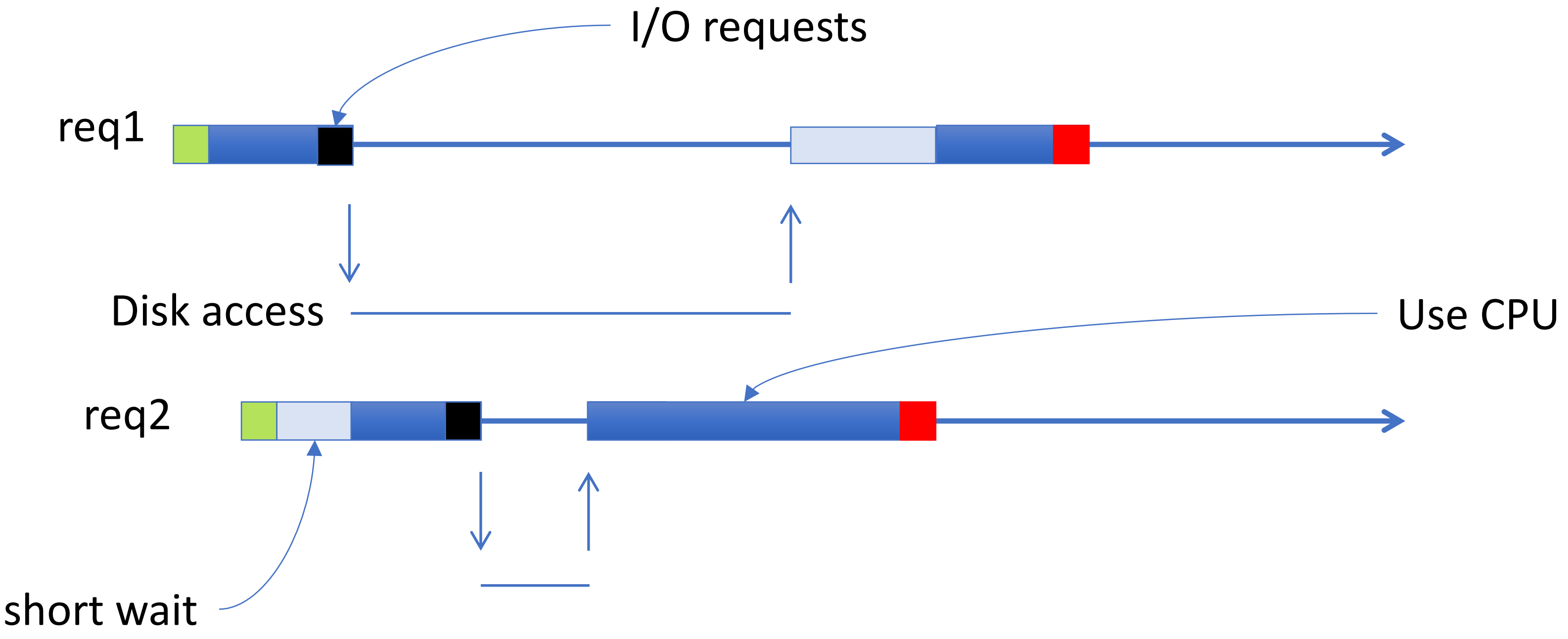
Multi vs. Single-process Web Server

Example: Web server receives two requests in quick succession

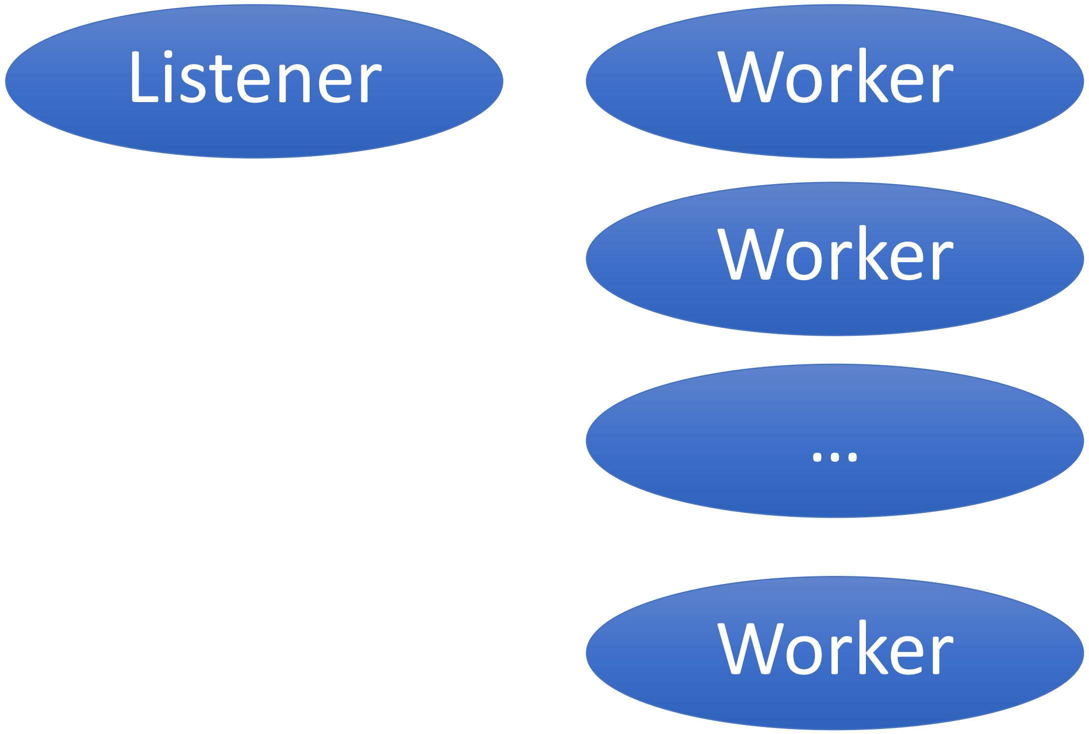


Multi vs. Single-process Web Server

Example: Web server receives two requests in quick succession



Multiprocess Web Server



Listener

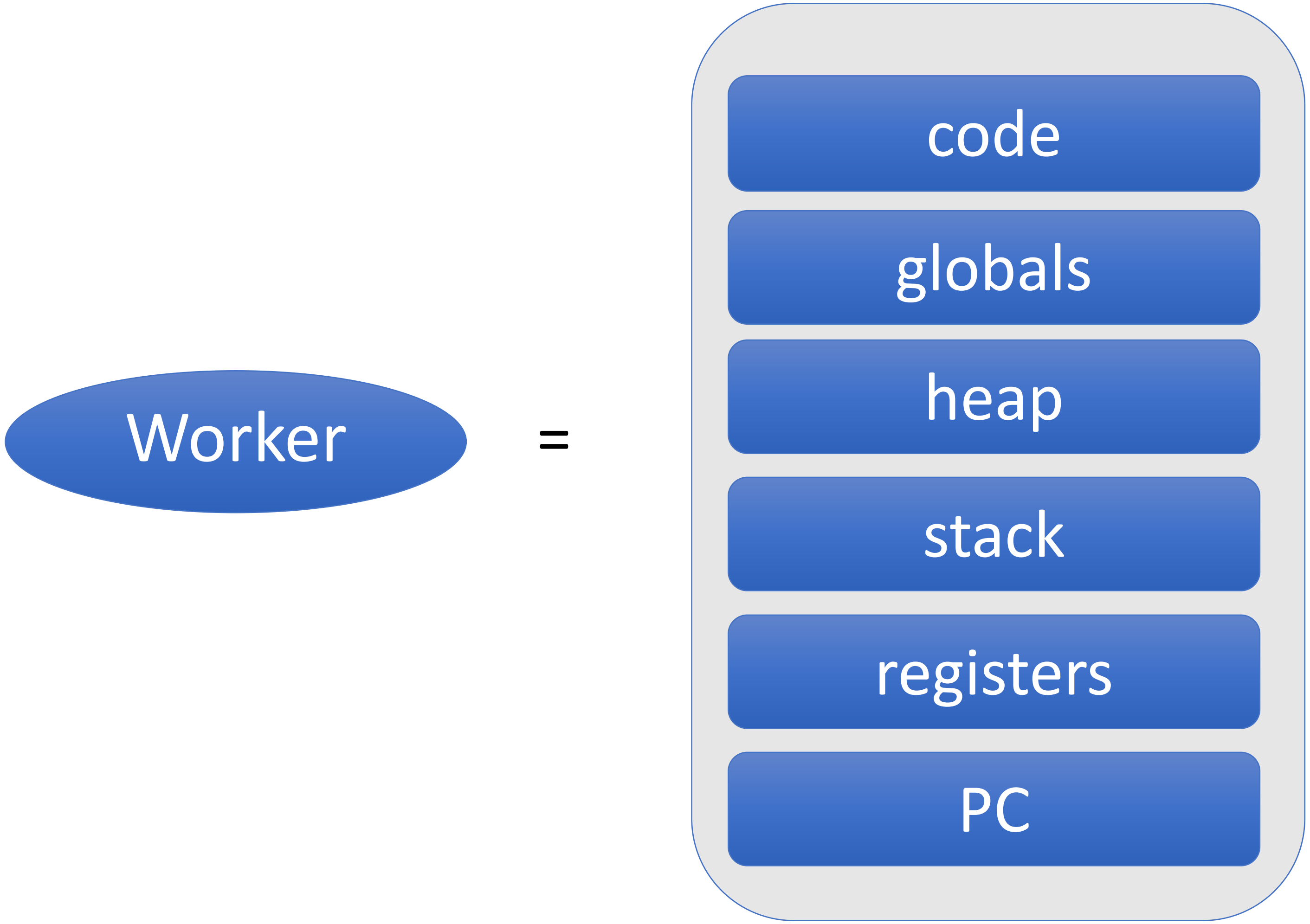
Worker

Worker

…

Worker

Each Worker is a Process



Amount of work on server per request

- Receive network packet
- Run listener process
- Create worker process
- Read file from disk
- Send network packet

Amount of work on server per request

- Receive network packet
- Run listener process
- *Create worker process is expensive*
- Read file from disk
- Send network packet

Multiprocess Web Server

```
ListenerProcess {
  forever {
    wait for incoming request
    CreateProcess( worker, request )
  }
}

WorkerProcess(request) {
  read file from disk
  send response
  exit
}
```

How can we avoid this?

Process Pool

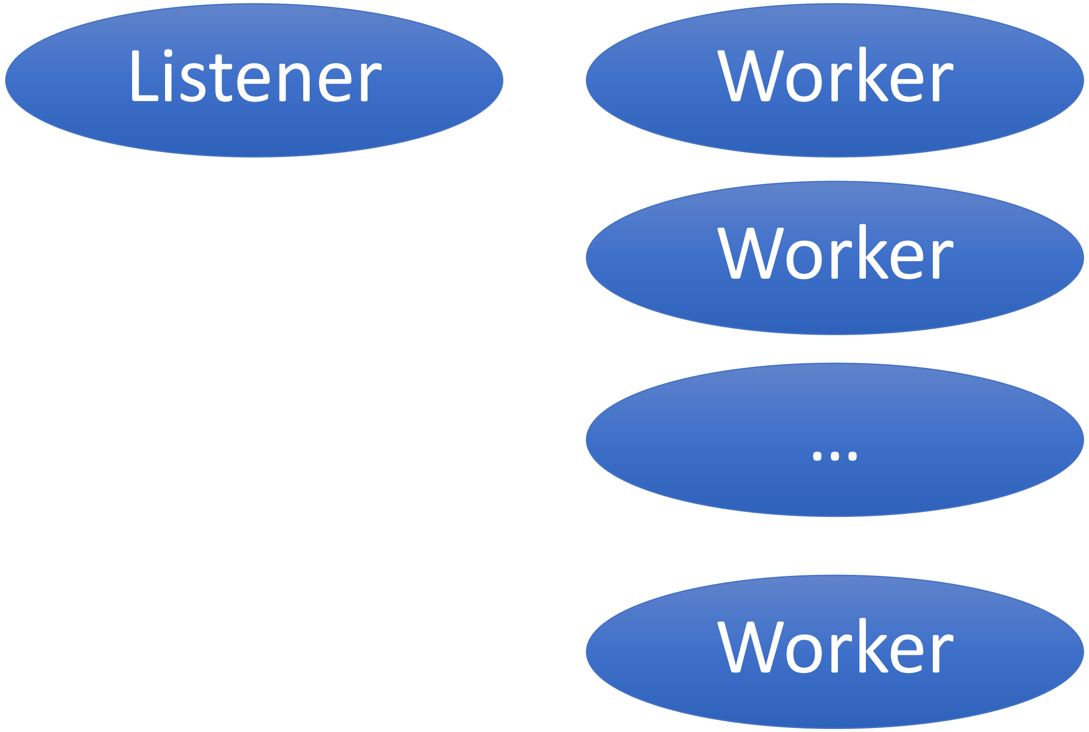
- Create worker processes during initialization
- Hand incoming request to them

Multiprocess Web Server with Process Pool

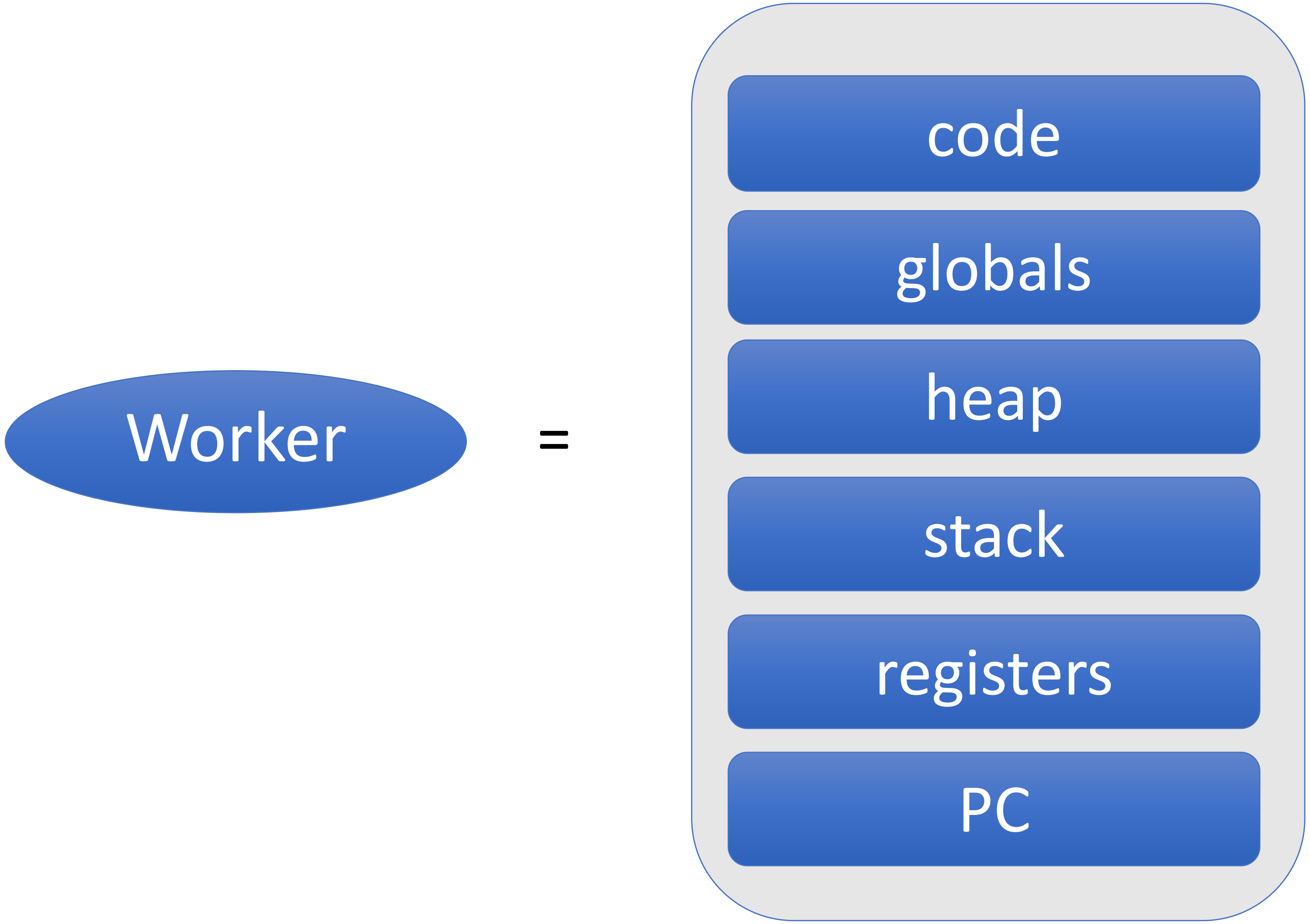
```
ListenerProcess {
  for(i=0; i<MAX_PROCESSES; i++) process[i] = CreateProcess(worker)
    forever {
      wait for incoming request
      send(request, process[?])
    }
}

WorkerProcess[?] {
  forever {
    wait for message(&request)
    read file from disk
    send response
  }
}
```

Pictures remain the same



Pictures remain the same



What changed: Amount of work on server per request

- Receive network packet
- Run listener process
- *Send message to worker process (cheaper)*
- Read file from disk
- Send network packet

Interprocess Communication (IPC)

Interprocess Communication (IPC)

- OS support to allow the processes to manage shared data
  - Through message passing
  - Through remote procedure calls (RPC)

Where do you need IPC?

Multiprocess Web Server with Process Pool

```
ListenerProcess {
  for( i=0; i<MAX_PROCESSES; i++ ) process[i] = CreateProcess(worker)
    forever {
      receive incoming request
      send( request, process[?] )
    }
}

WorkerProcess[?] {
  forever {
    wait for message( &request )
    read file from disk
    send response
  }
}
```

Need IPC here

For client-server communication

Multiprocess Web Server with Process Pool

```
ListenerProcess {
  for( i=0; i<MAX_PROCESSES; i++ ) process[i] = CreateProcess(worker)
    forever {
       receive incoming request
       send( request, process[?] )
    }
}

WorkerProcess[?] {
  forever {
    wait for message( &request )
    read file from disk
    send response
  }
}
```

Need IPC here

For communication between cooperating processes

(e.g., between listener and workers)

Balmau © 2023

Where do you need IPC?

- Between client and server
- Between cooperating processes

How to do IPC?

- Message passing
- Remote procedure calls (RPC)

Message Passing Primitives

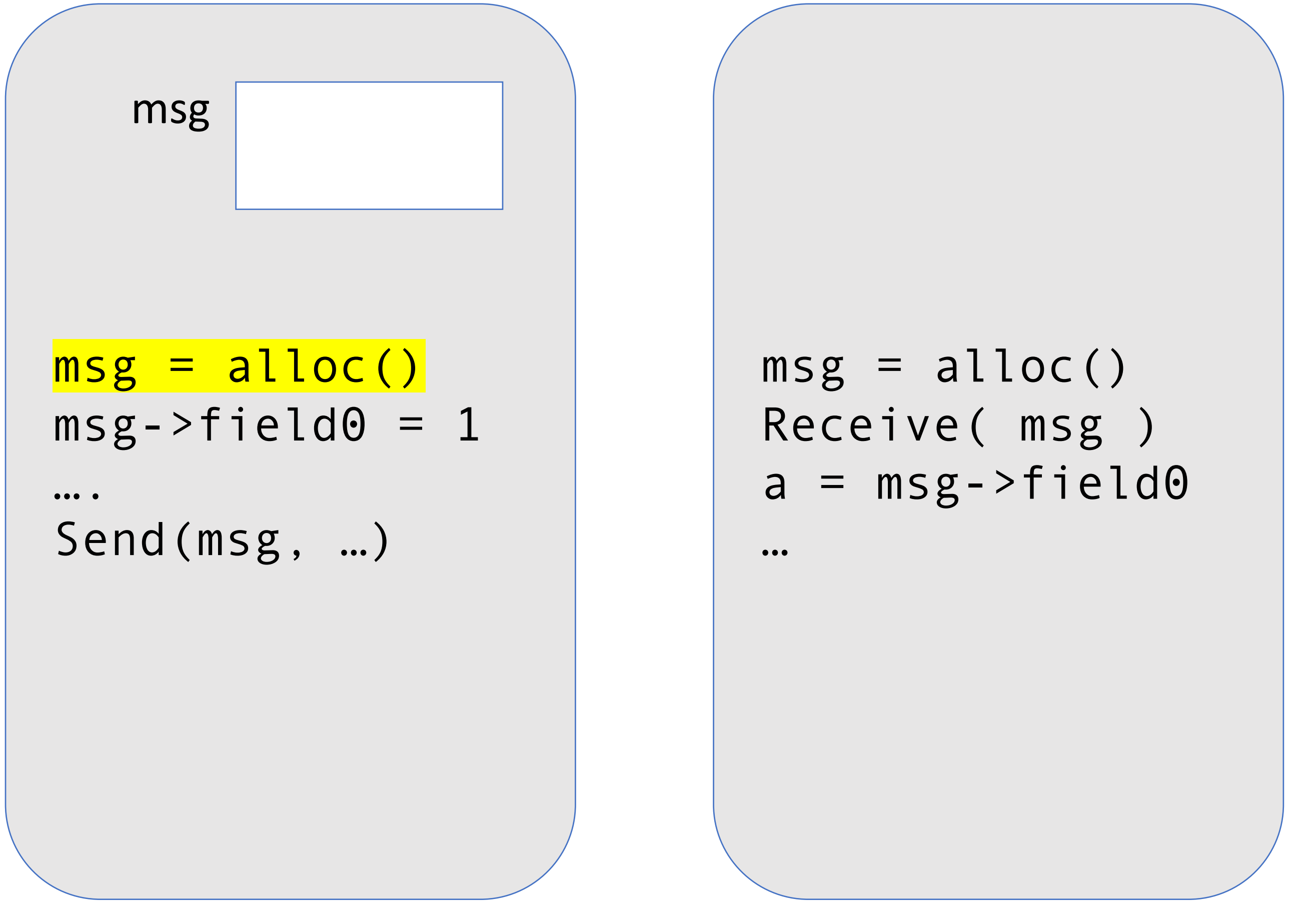
- Send message
- Receive message

Message Passing Send / Receive

```
msg = alloc()
msg->field0 = 1
….
Send(msg, …)
```

```
msg = alloc()
Receive( msg )
a = msg->field0
…
```

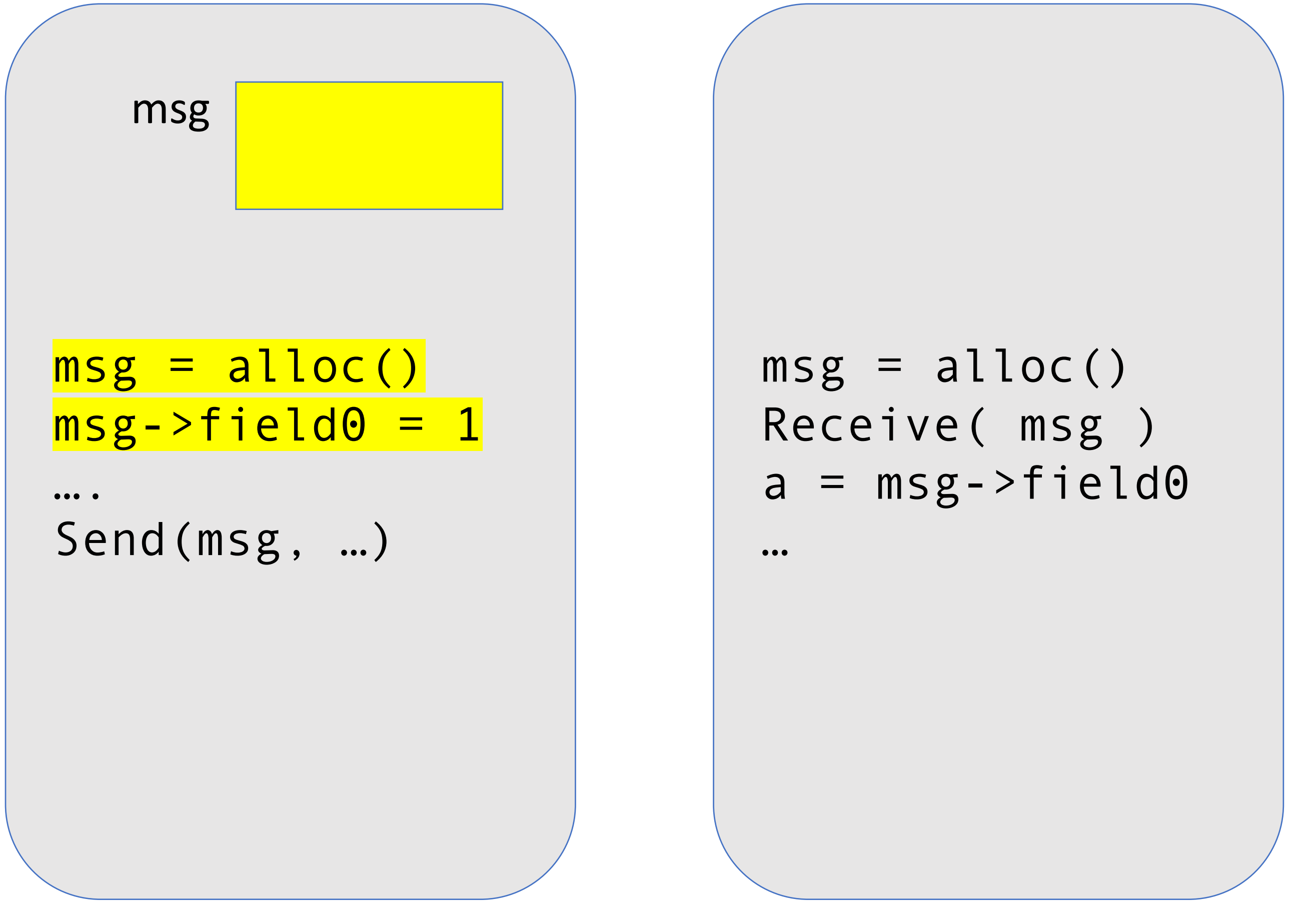
Message Passing Send / Receive



```
msg = alloc()
msg->field0 = 1
….
Send(msg, …)
```

```
msg = alloc()
Receive( msg )
a = msg->field0
…
```

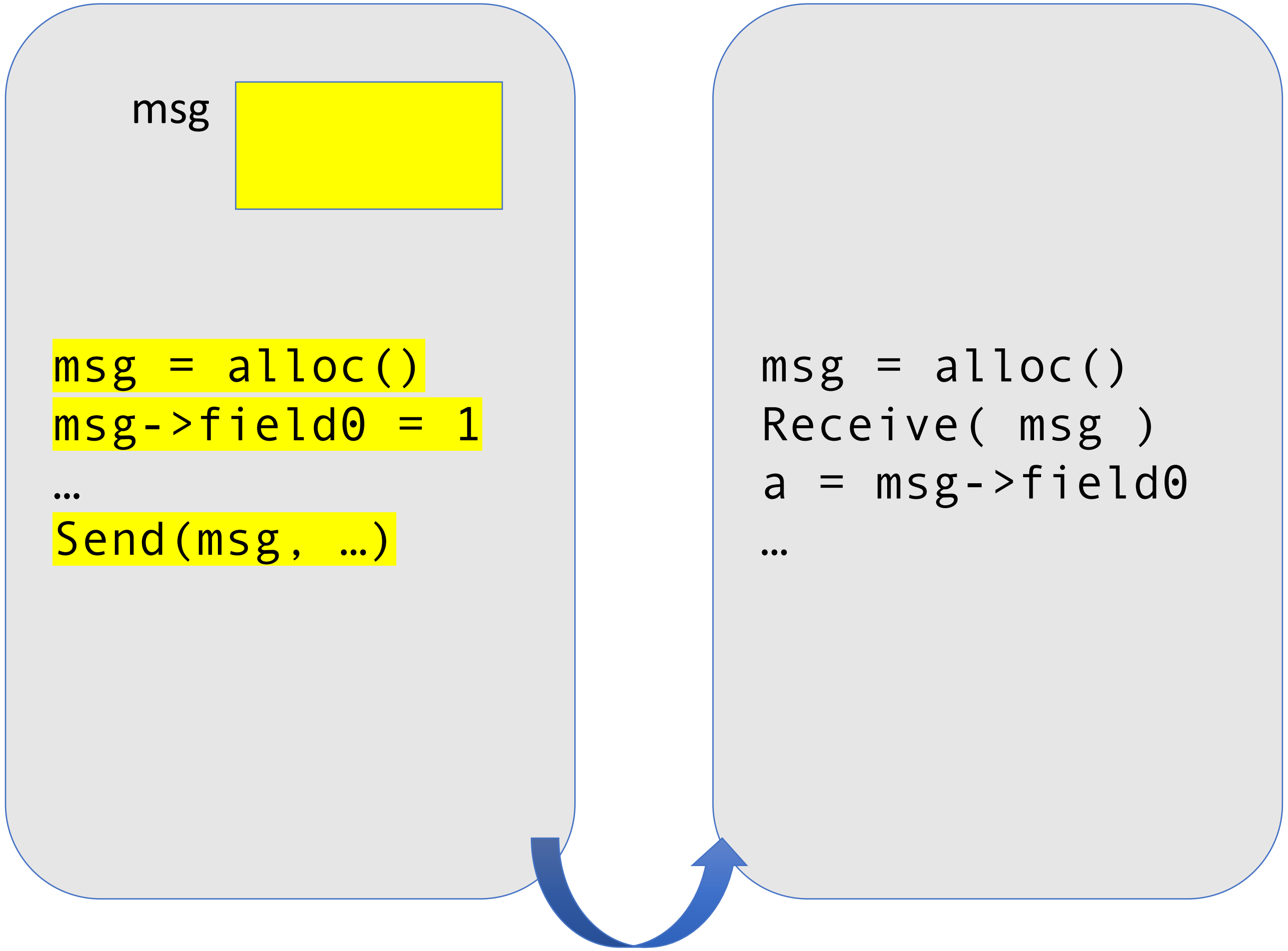
Message Passing Send / Receive



```
msg = alloc()
msg->field0 = 1
….
Send(msg, …)
```

```
msg = alloc()
Receive( msg )
a = msg->field0
…
```

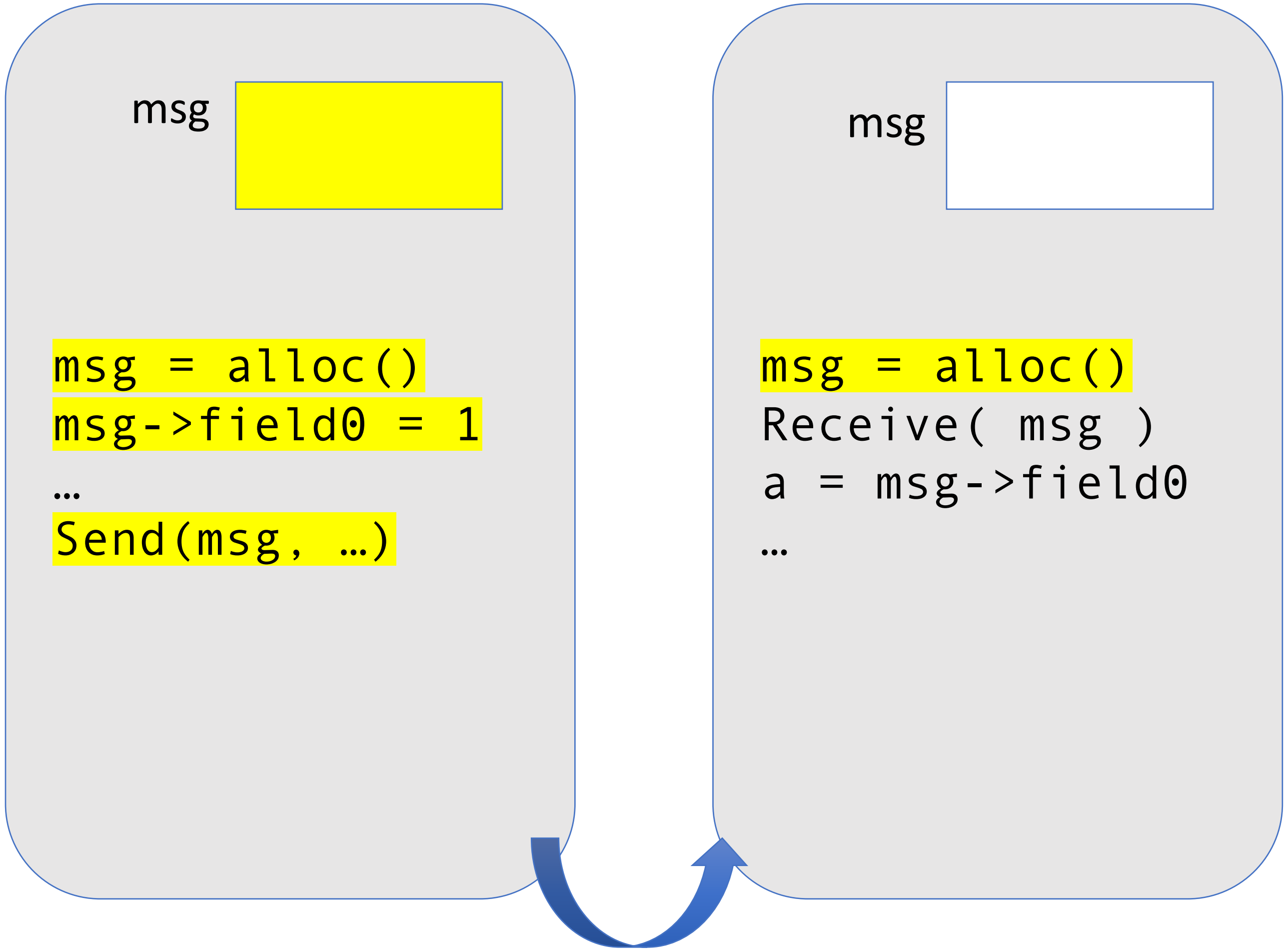
Message Passing Send / Receive



```
msg = alloc()
msg->field0 = 1
…
Send(msg, …)
```

```
msg = alloc()
Receive( msg )
a = msg->field0
…
```

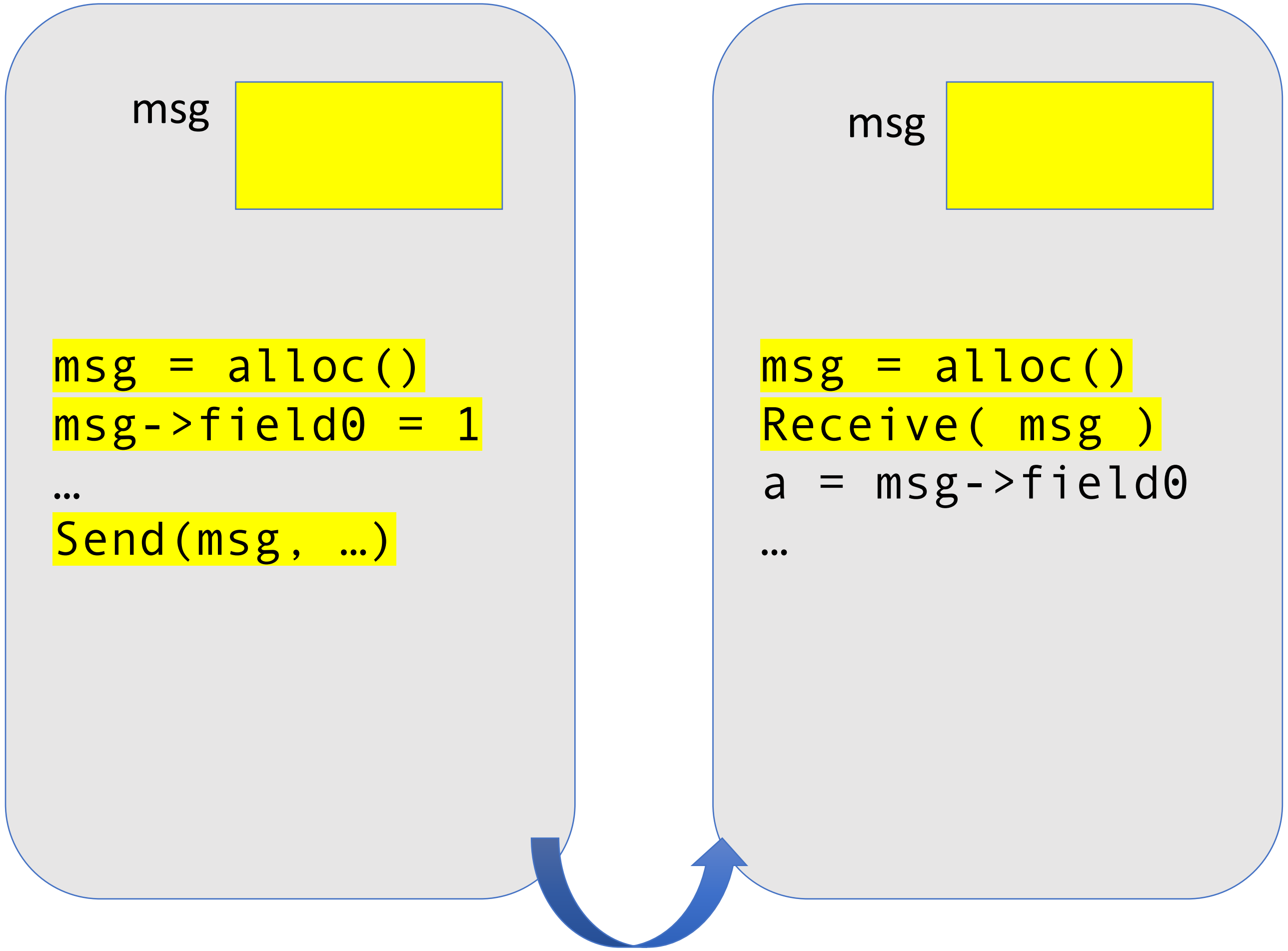
Message Passing Send / Receive



```
msg = alloc()
msg->field0 = 1
…
Send(msg, …)
```

```
msg = alloc()
Receive( msg )
a = msg->field0
…
```

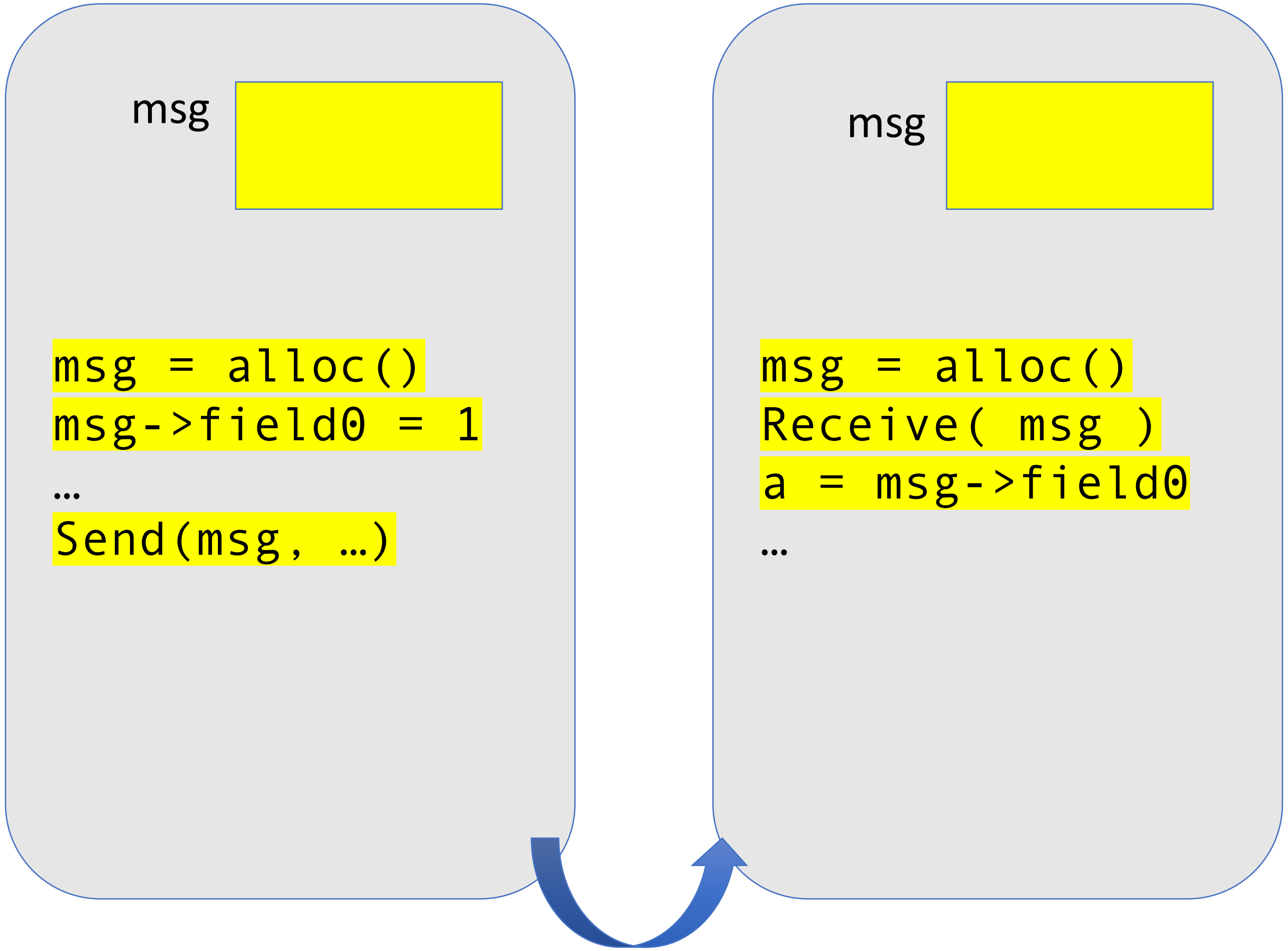
Message Passing Send / Receive



```
msg = alloc()
msg->field0 = 1
…
Send(msg, …)
```

```
msg = alloc()
Receive( msg )
a = msg->field0
…
```

Message Passing Send / Receive



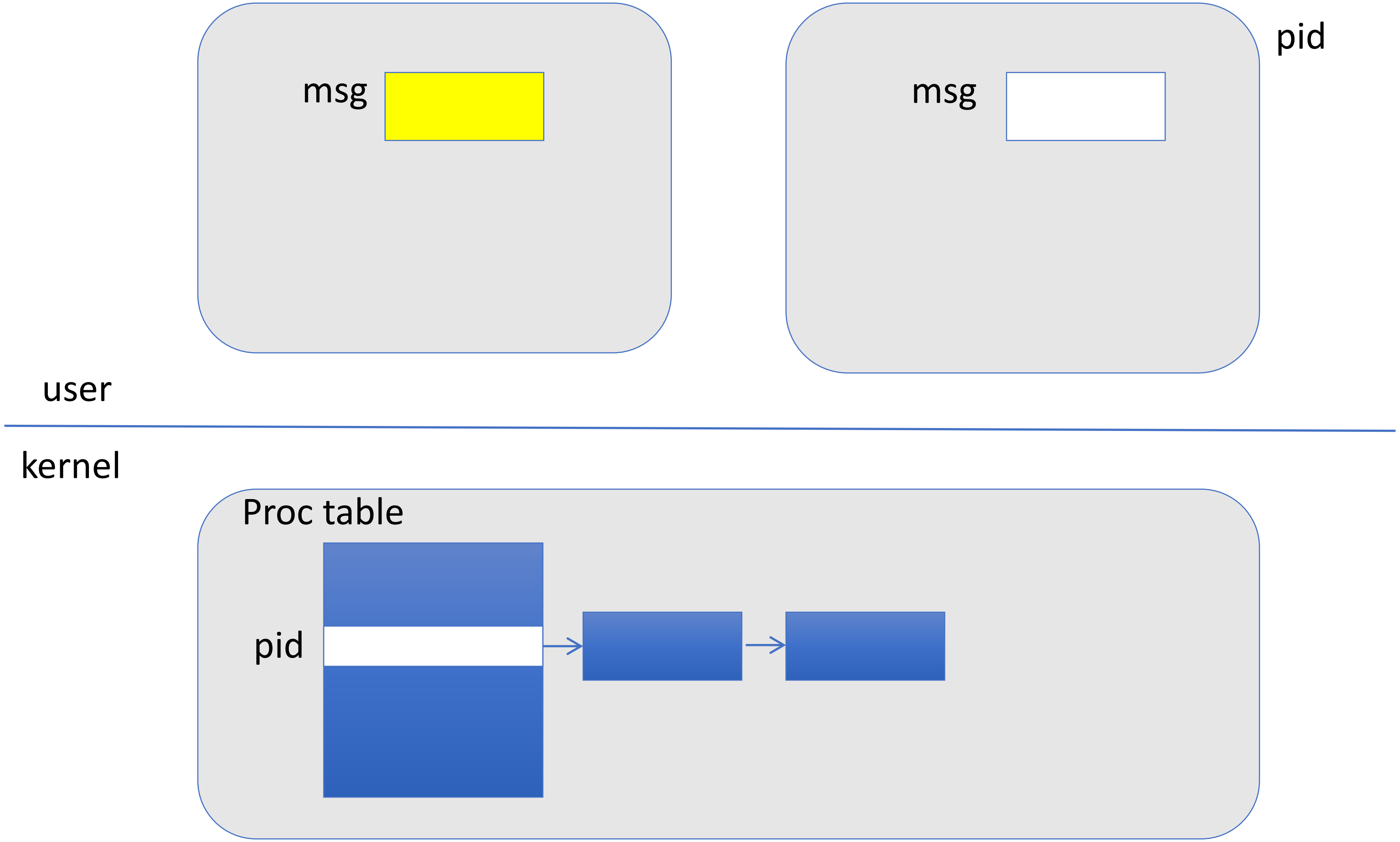
```
msg = alloc()
msg->field0 = 1
…
Send(msg, …)
```

```
msg = alloc()
Receive( msg )
a = msg->field0
…
```

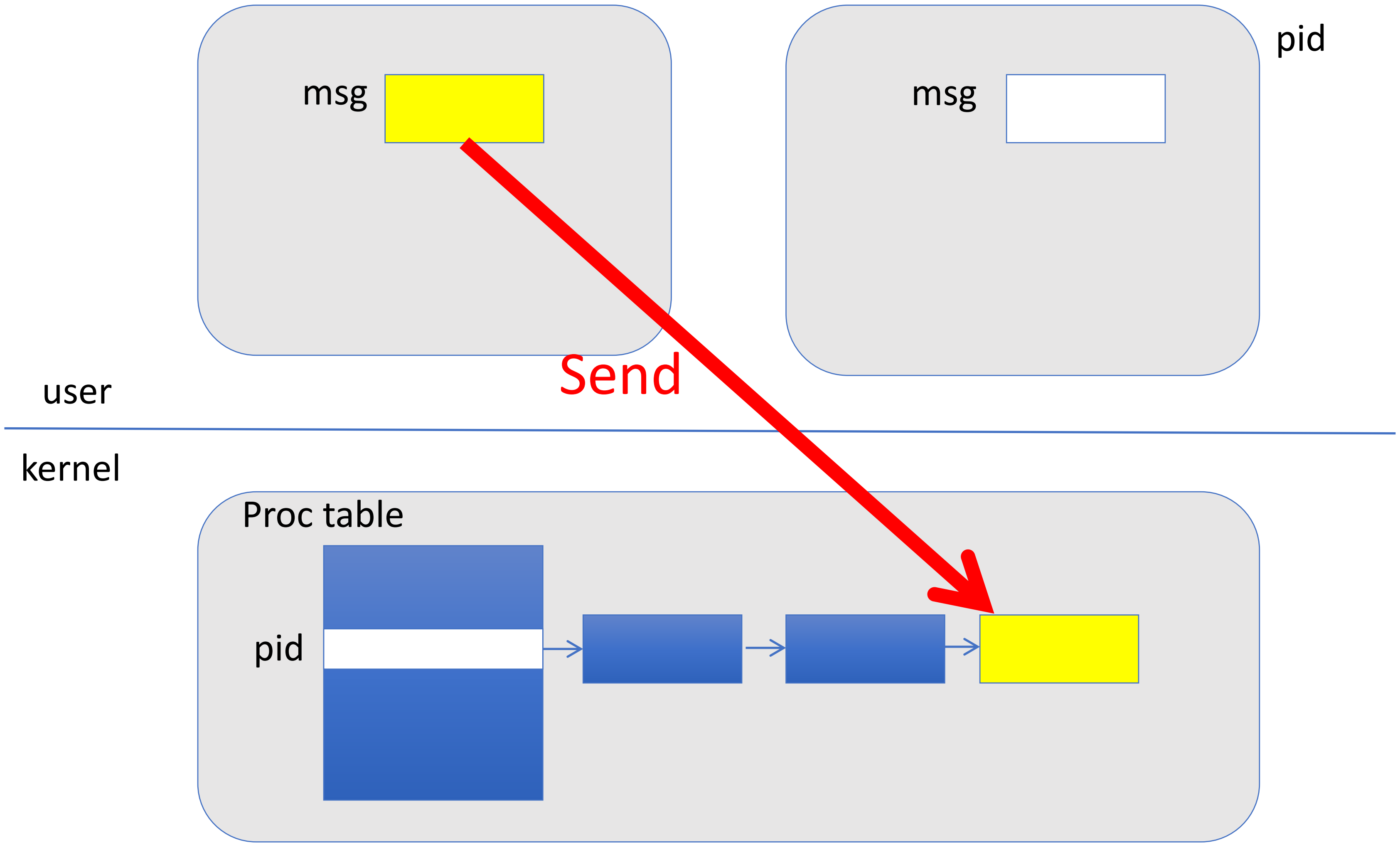
Message Passing

- By value communication
- **Never by reference**
- Receiver cannot affect message in sender

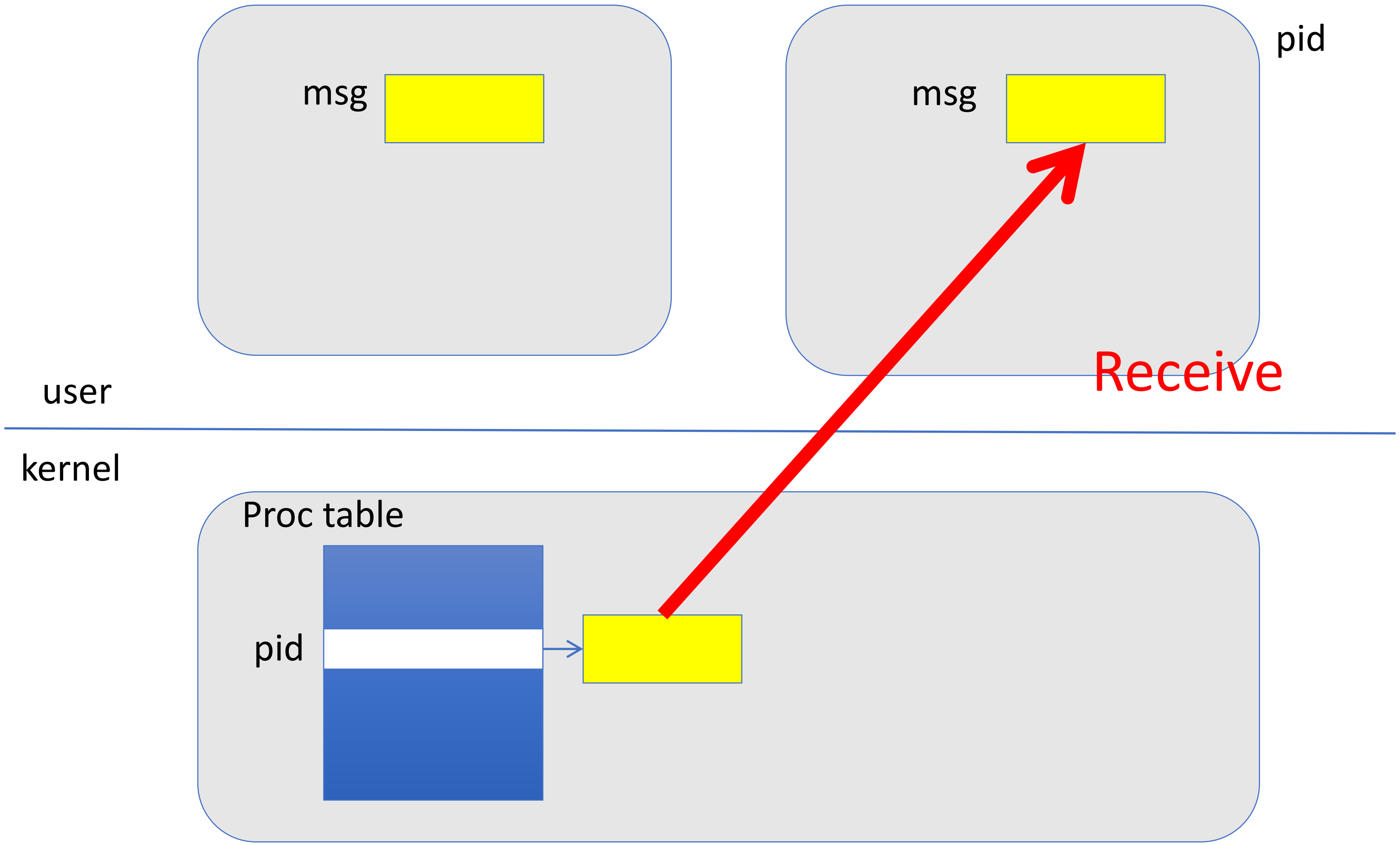
Message Passing Implementation



Message Passing Implementation



Message Passing Implementation



Message Passing Alternatives

- Symmetric / asymmetric addressing
- Blocking / nonblocking

Symmetric Addressing

- **`Send`**`(msg, to_pid)`
- **`Receive`**`(msg, from_pid)`

- Message is (typically) a struct
- to_pid, from_pid are process identifiers

- Symmetric addressing seldom used

Asymmetric Addressing

- **`Send`**`(msg, pid)`
  - Send msg to process pid

- `pid = `**`Receive`**`(msg)`
  - Receive msg from *any* process
  - Return the pid of sending process

- More common and useful form of addressing

Blocking or Nonblocking Send

- Nonblocking:
  - Send returns immediately after message is sent
- Blocking
  - Sender blocks until message is delivered
- Nonblocking is the more common form

Blocking or Nonblocking Receive

- Nonblocking
  - Receive returns immediately
  - Regardless of message present or not
- Blocking
  - Receive blocks until message is present
- Blocking is the more common form

(Slightly Rewritten) Example: Multiprocess Web Server with Process Pool

```
ListenerProcess {
  for(i=0; i<MAX_PROCESSES; i++) process[i] = CreateProcess(worker)
    forever {
       client_pid = receive(msg)
       msg’ = slightly modify msg to include client_pid
       send(msg’, worker_process[i])
    }
}

WorkerProcess[i] {
  forever {
    receive(msg)
    read file from disk
    send(resp, client_pid)
  }
}
```

Asymmetric Addressing: Send

```
ListenerProcess {
  for(i=0; i<MAX_PROCESSES; i++) process[i] = CreateProcess(worker)
    forever {
      client_pid = receive(msg)
      msg’ = slightly modify msg to include client_pid
      send(msg’, worker_process[i])
    }
}

WorkerProcess[i] {
  forever {
    receive(msg)
    read file from disk
    send(resp, client_pid)
  }
}
```

Asymmetric Addressing: Receive

```
ListenerProcess {
  for(i=0; i<MAX_PROCESSES; i++) process[i] = CreateProcess(worker)
    forever {
      client_pid = receive(msg)  /* receive msg from any client */
      msg' = slightly modify msg to include client_pid
      send(msg', worker_process[i])
    }
}

WorkerProcess[i] {
  forever {
    receive(msg)  /* receive msg' from listener; could be symmetric */
    read file from disk
    send(resp, client_pid)
  }
}
```

Blocking Receive

```
ListenerProcess {
  for(i=0; i<MAX_PROCESSES; i++) process[i] = CreateProcess(worker)
    forever {
      client_pid = receive(msg) /* nothing else to do*/
      msg’ = slightly modify msg to include client_pid
      send(msg’, worker_process[i])
    }
}

WorkerProcess[i] {
  forever {
    receive(msg) /* nothing else to do*/
    read file from disk
    send(resp, client_pid)
  }
}
```

Nonblocking Send

```
ListenerProcess {
  for(i=0; i<MAX_PROCESSES; i++) process[i] = CreateProcess(worker)
    forever {
      client_pid = receive(msg)
      msg' = slightly modify msg to include client_pid
      send(msg', worker_process[i]) /* must not block */
    }
}

WorkerProcess[i] {
  forever {
    receive(msg)
    read file from disk
    send(resp, client_pid) /* must not block */
  }
}
```

Client-Server Communication

(Server Side) Client-Server Communication

```
ListenerProcess {
 for(i=0; i<MAX_PROCESSES; i++) process[i] = CreateProcess(worker)
  forever {
   receive incoming request
   send( request, process[?] )
  }
}

WorkerProcess[?] {
 forever {
  wait for message( &request )
  read file from disk
  send response
 }
}
```

(Client-Side) Client-Server Communication

```
send(msg to server)

receive(reply msg from server)
```

A Very Common Pattern

- Client:
  - Send /* send request to server */
  - Blocking receive /* wait for reply */
- Server
  - Blocking receive /* wait for request */
  - Send /* send reply */

This looks like …

• Client:	calling site
• Send	call procedure
• Blocking receive	return
• Server	callee site
• Blocking receive	invoke procedure
• Send	return

Remote Procedure Call (RPC)

- Client:
  - Send
  - Blocking receive
- Server
  - Blocking receive
  - Send

calling site

call procedure

return

callee site

invoke procedure

return

RPC: when client wants to call a function that belongs to server code

RPC Interface

- Interface
  - List of remotely callable procedures
  - With their arguments and return values

- Example: file system interface
  - `Open(string filename)`
  - returns `int fd`
    - fd = file descriptor; will see later in course
  - …

RPC Client Code

- Import file system interface

- `fd = open("/a/b/c")`
- `nbytes = read(fd, buffer, size)`

RPC Server Code

- Export file system interface

- `int open(stringname) { … }`
- `int read(fd, buffer, nbytes) { … }`
- …

Problem

- Want a procedure call interface
- Have only message passing between processes
- How to bridge the gap?

Solution: Stub Library

- Client stub and server stub
- Client stub linked with client process
- Server stub linked with server process

Two Message Types

- Call message
  - From client to server
  - Contains arguments
- Return message
  - From server to client
  - Contains return values

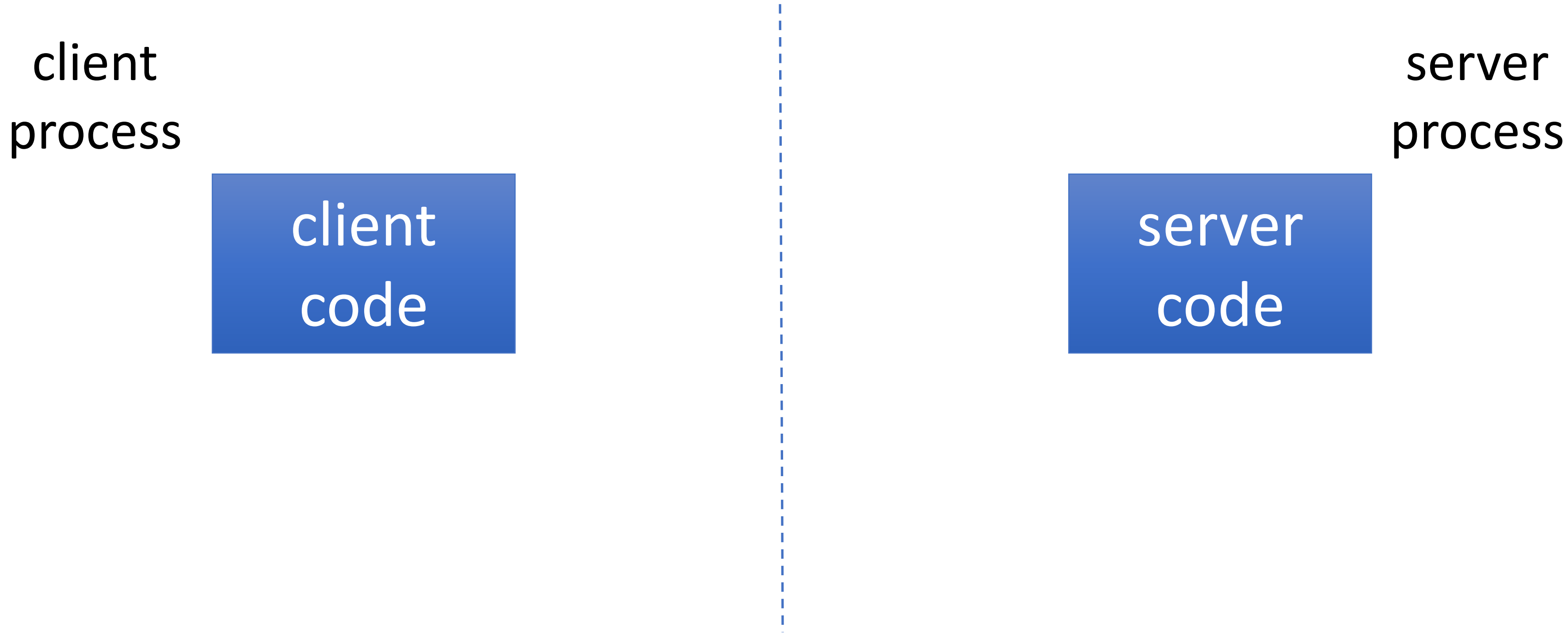
Client Stub

- Sends arguments in call message
- Receives return values in return message

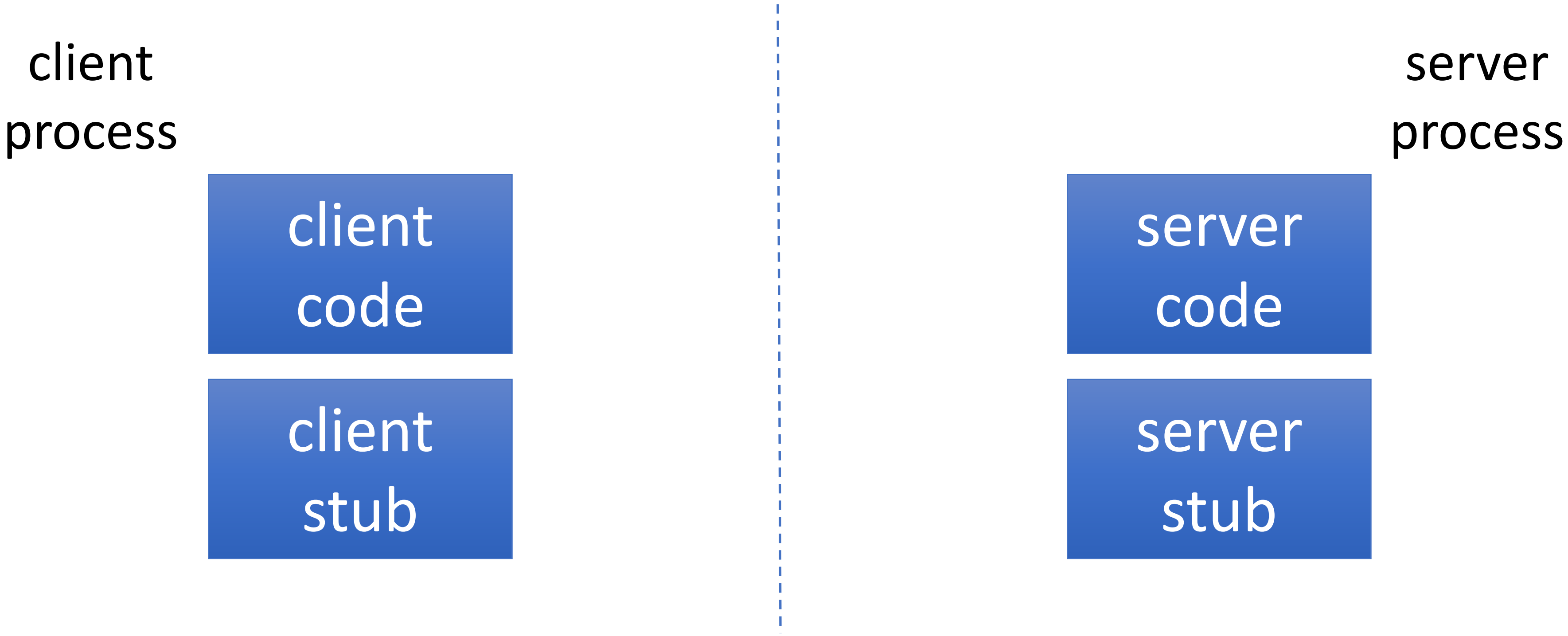
Server Stub

- Receives arguments in call message
- Invokes procedure
- Sends return values in return message

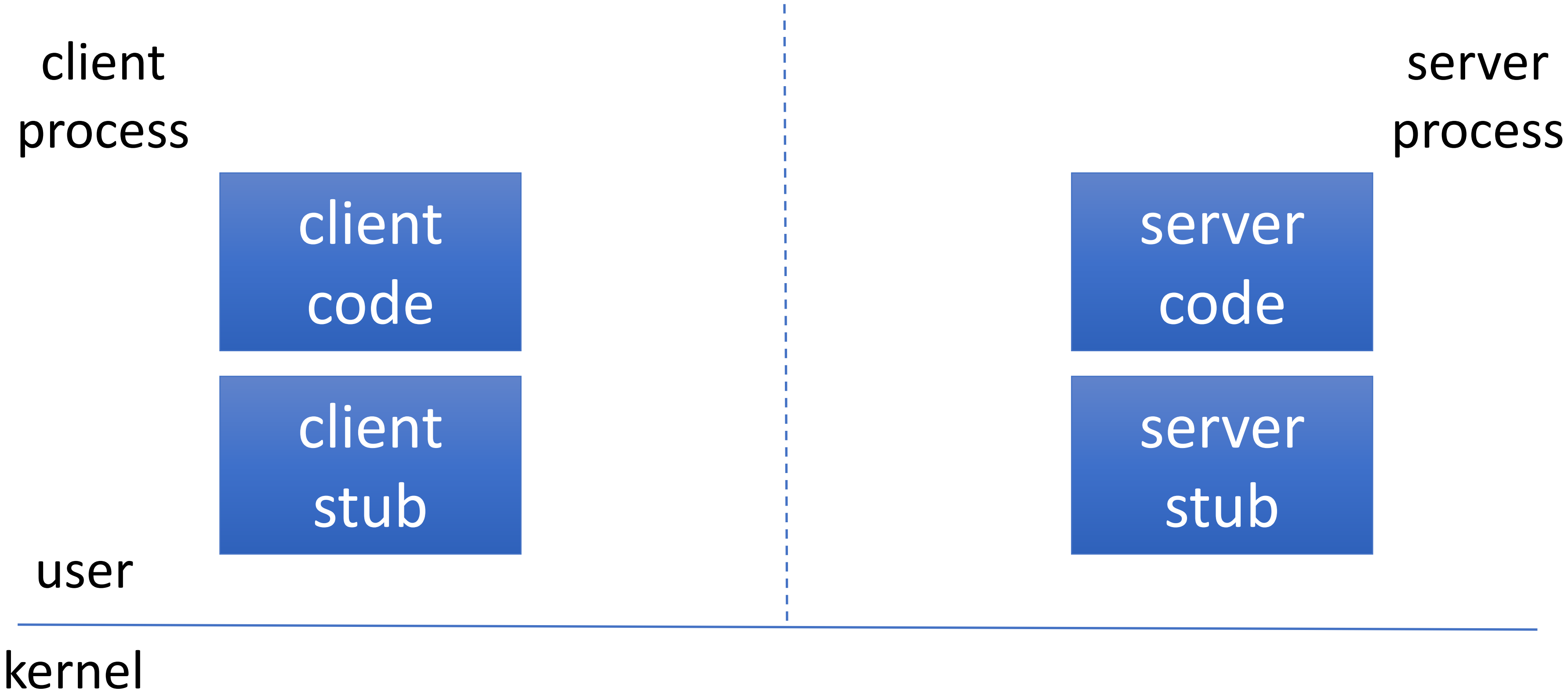
RPC Implementation



Client and Server Stubs



RPC Implementation: Call



client process

client code

client stub

user

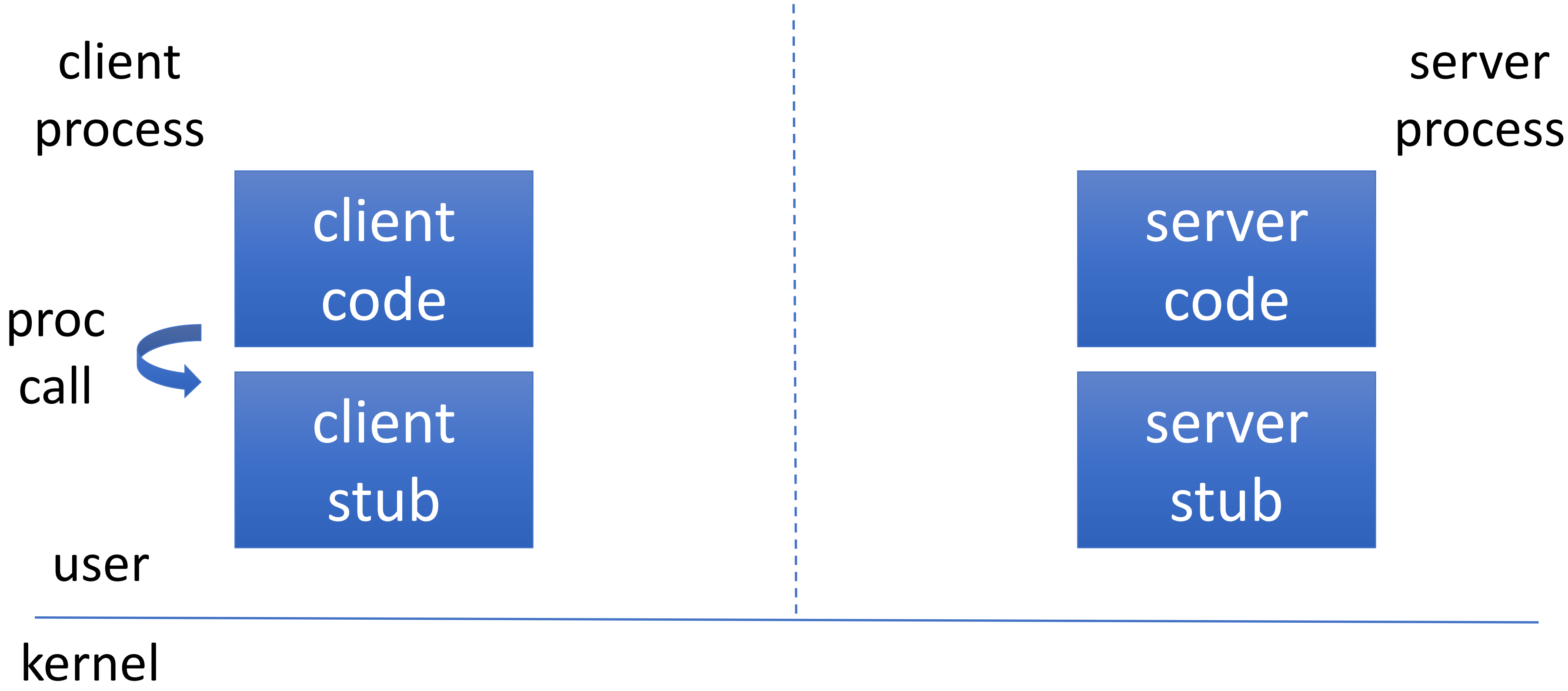
kernel

server process

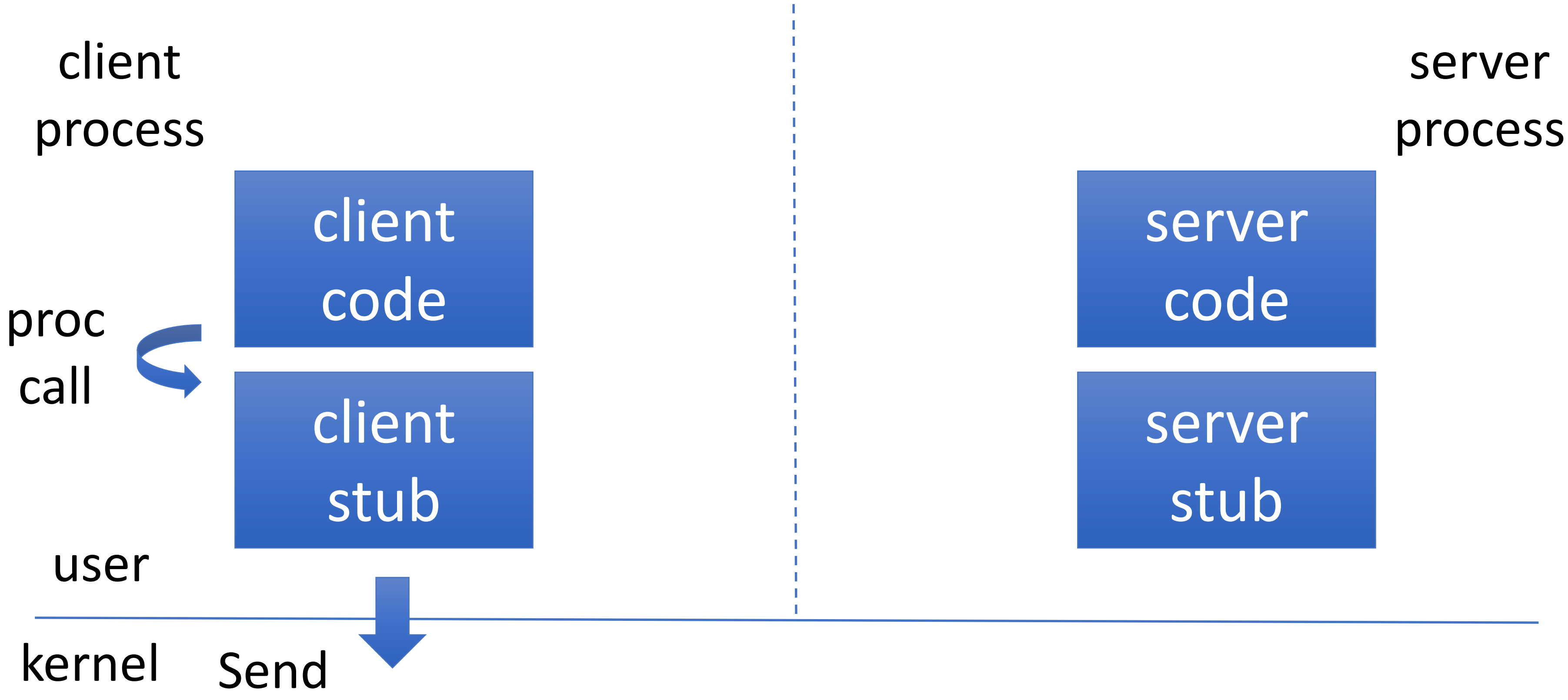
server code

server stub

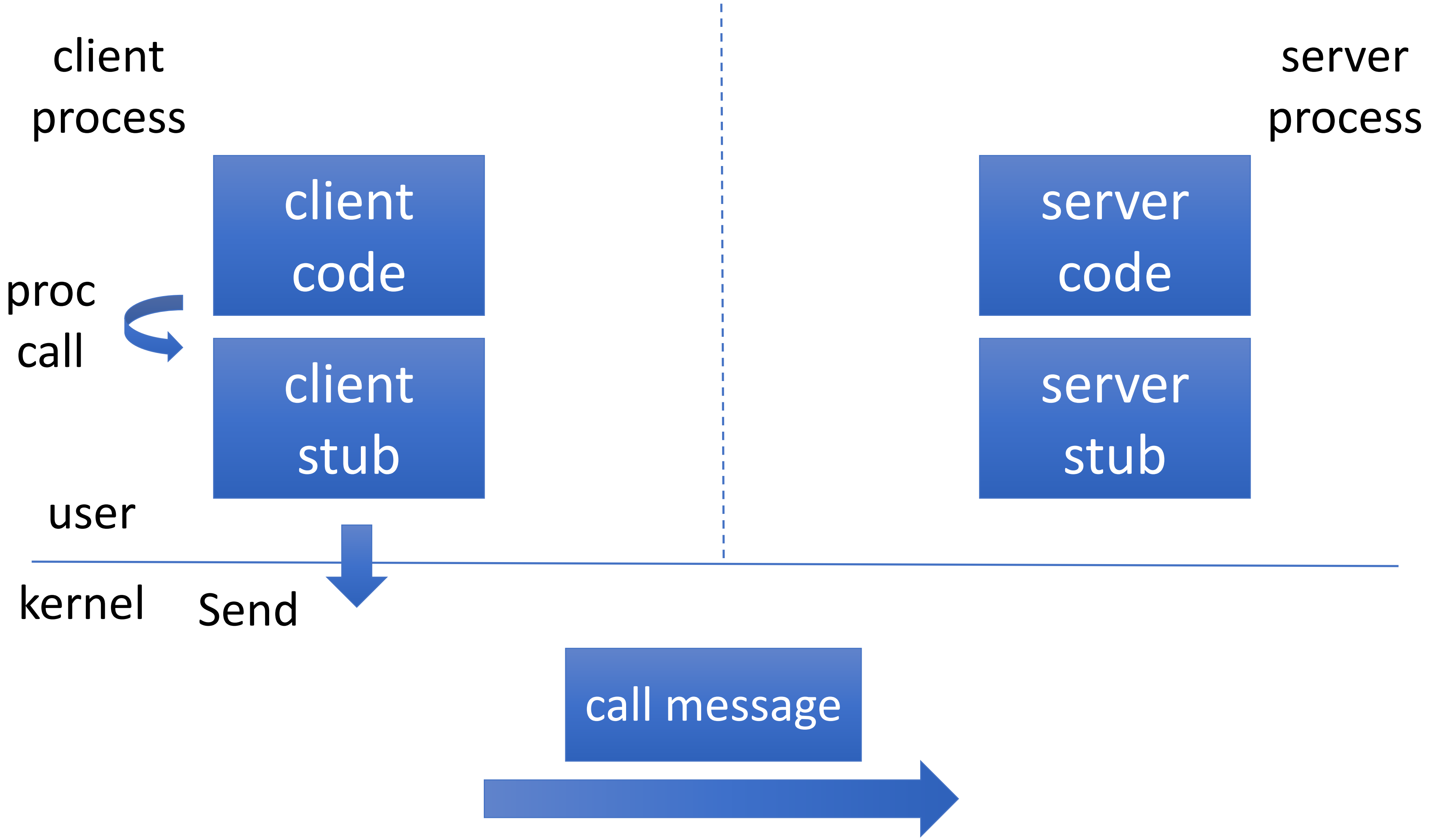
RPC Implementation: Call



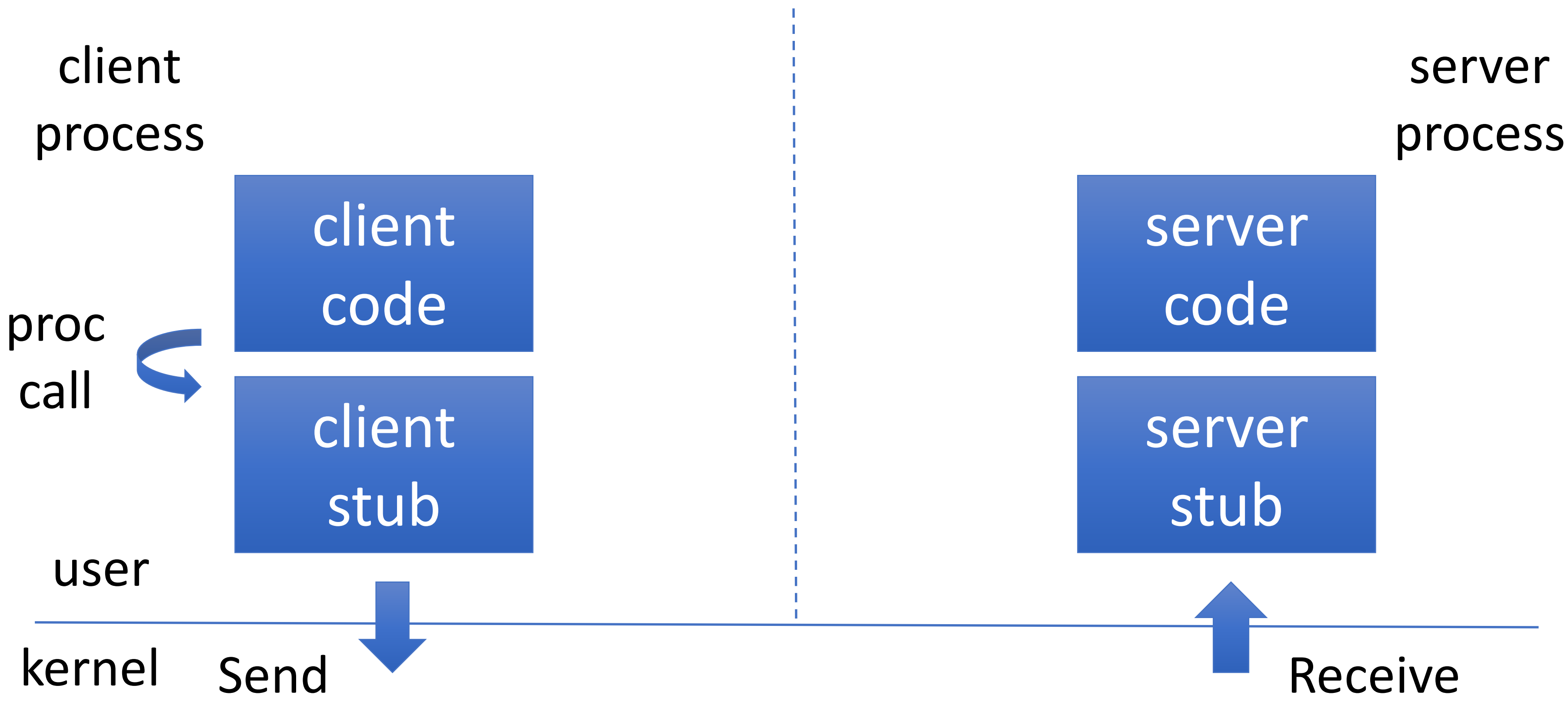
RPC Implementation: Call



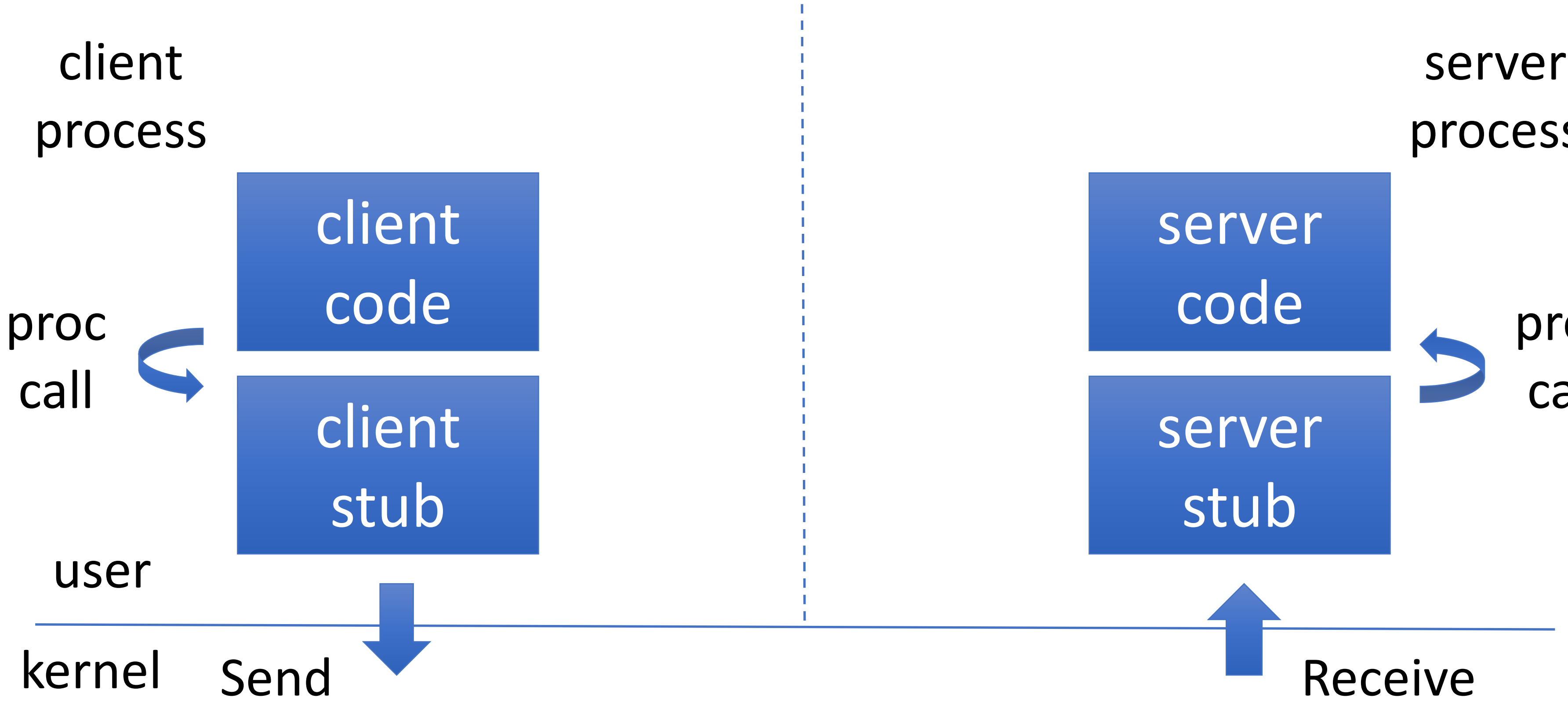
RPC Implementation: Call



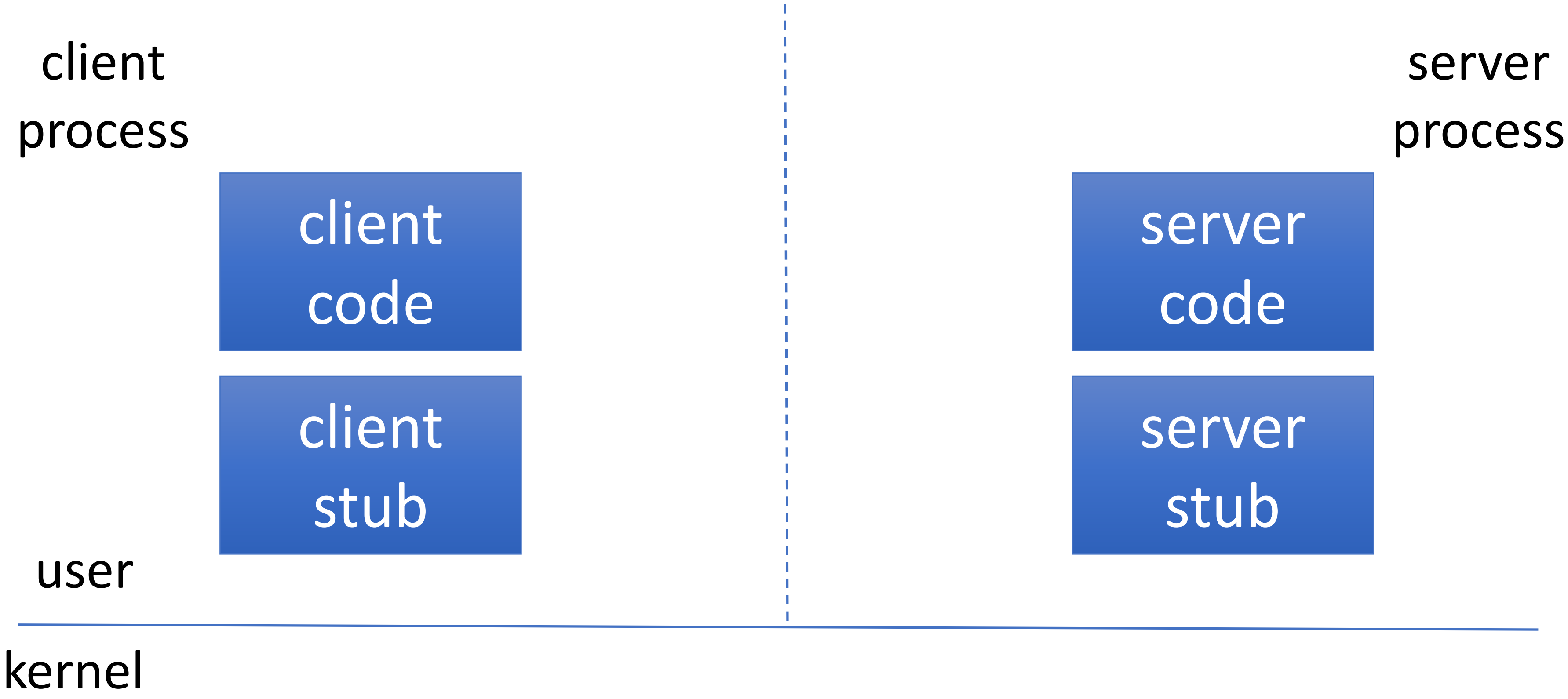
RPC Implementation: Call



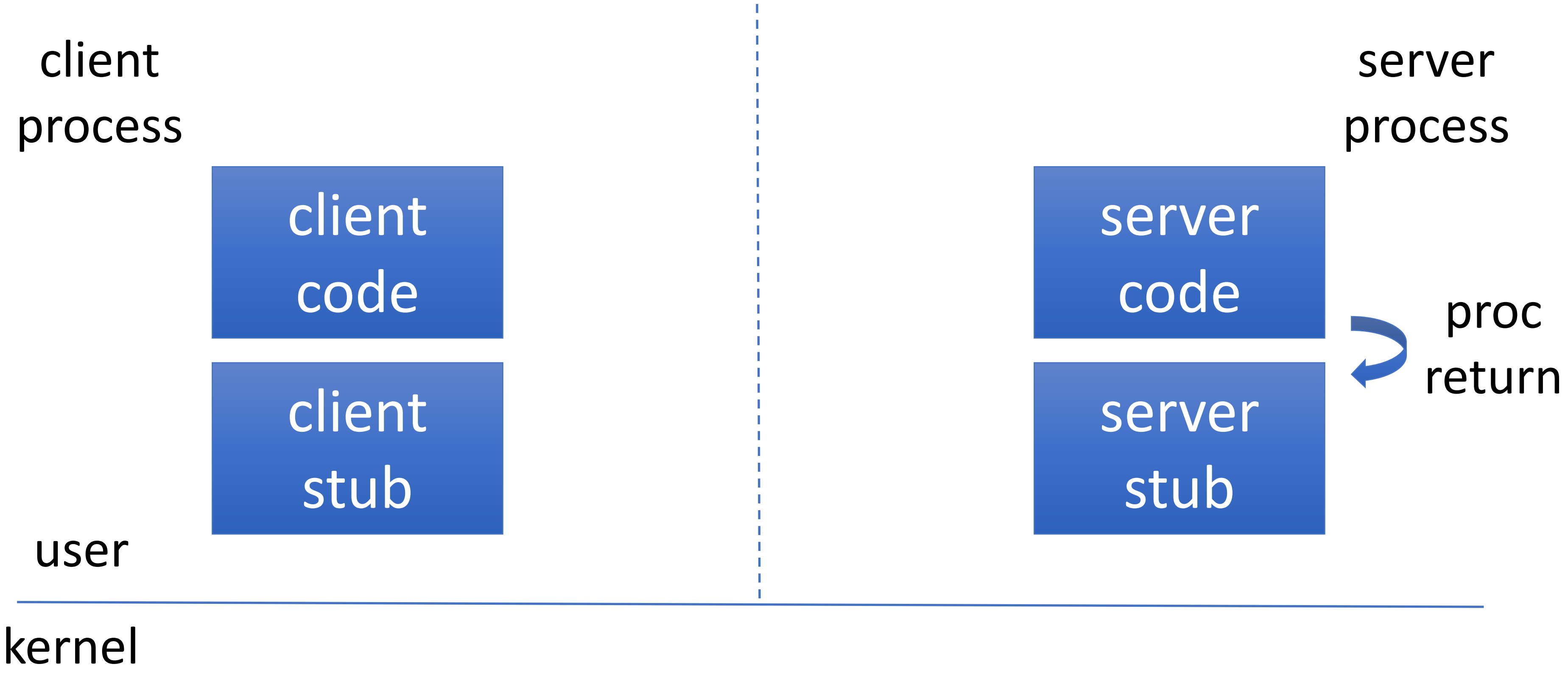
RPC Implementation: Call



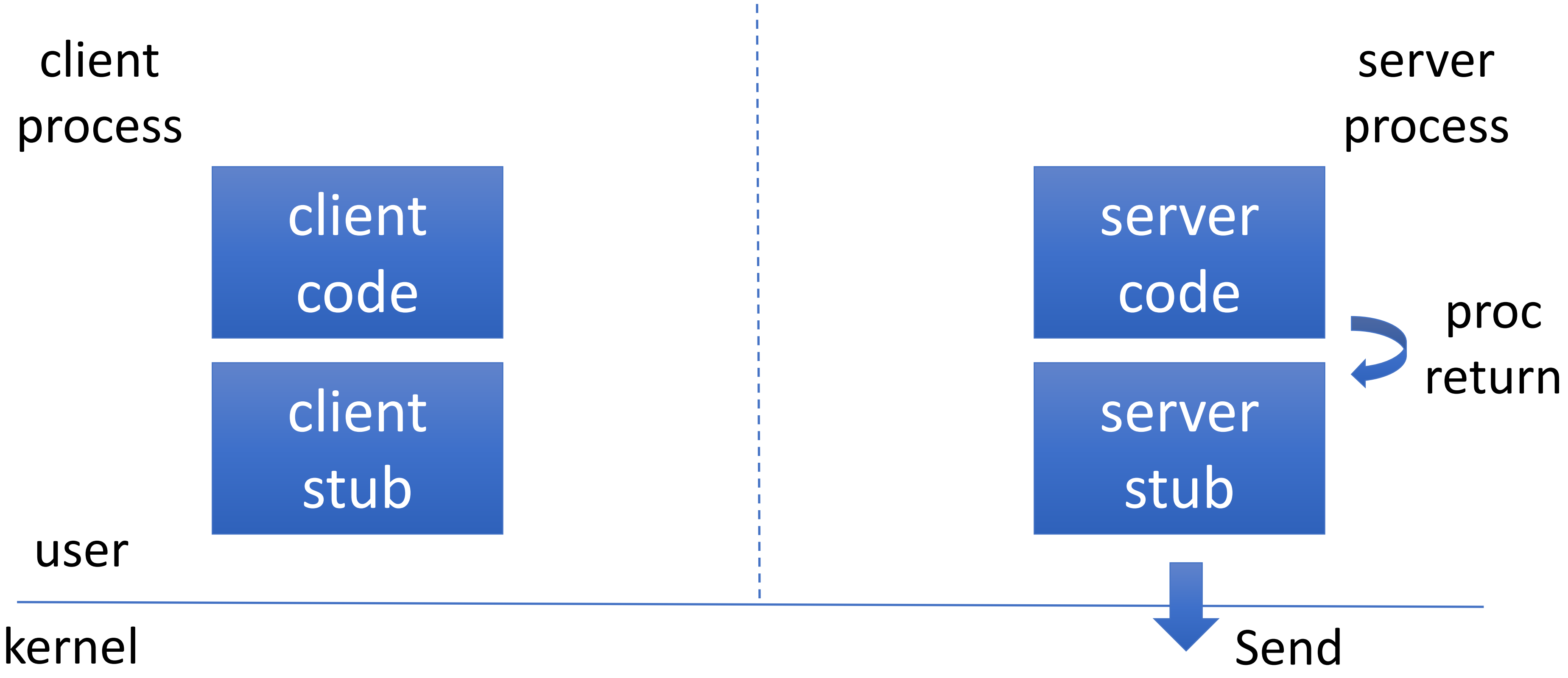
RPC Implementation: Return



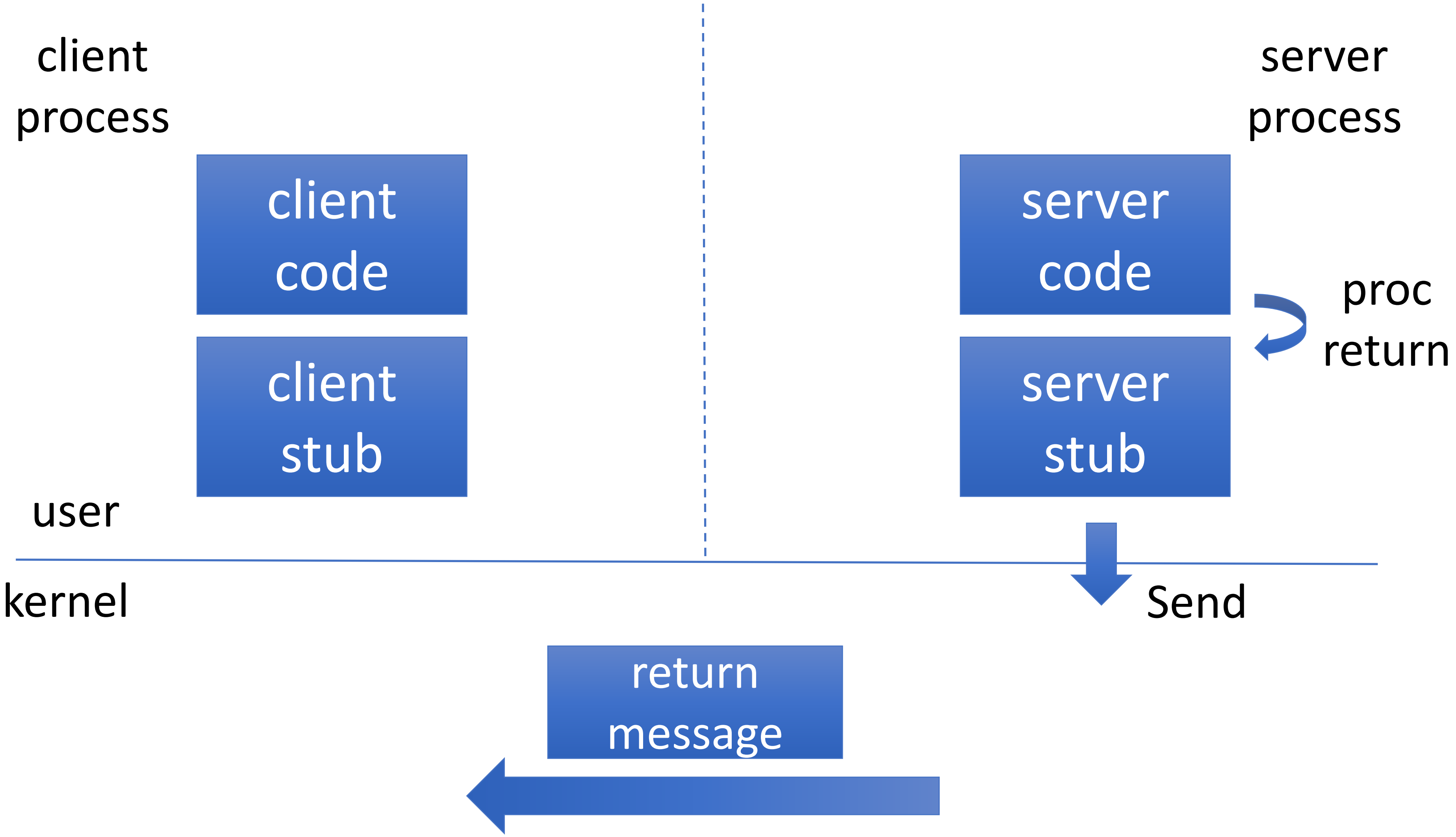
RPC Implementation: Return



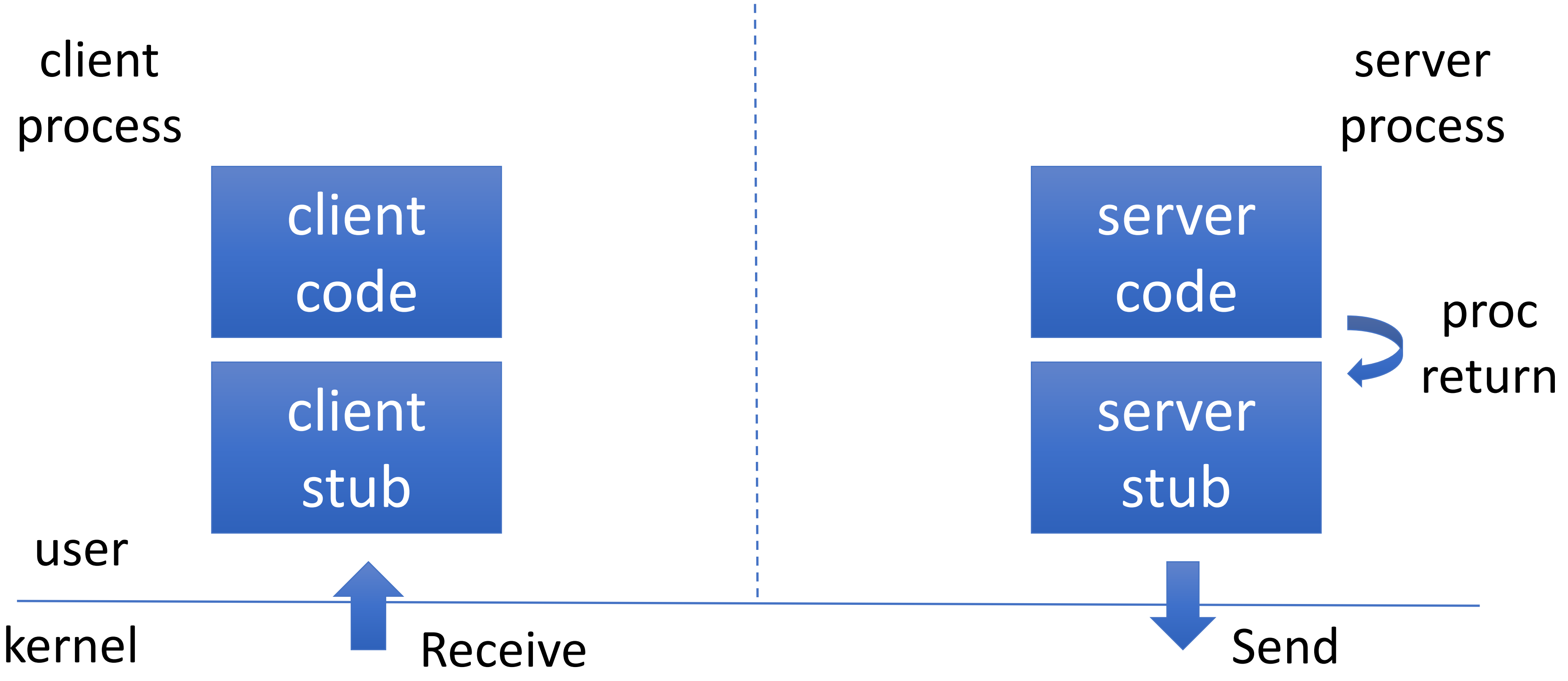
RPC Implementation: Return



RPC Implementation: Return



RPC Implementation: Return



client process

client code

client stub

user

kernel

Receive

server process

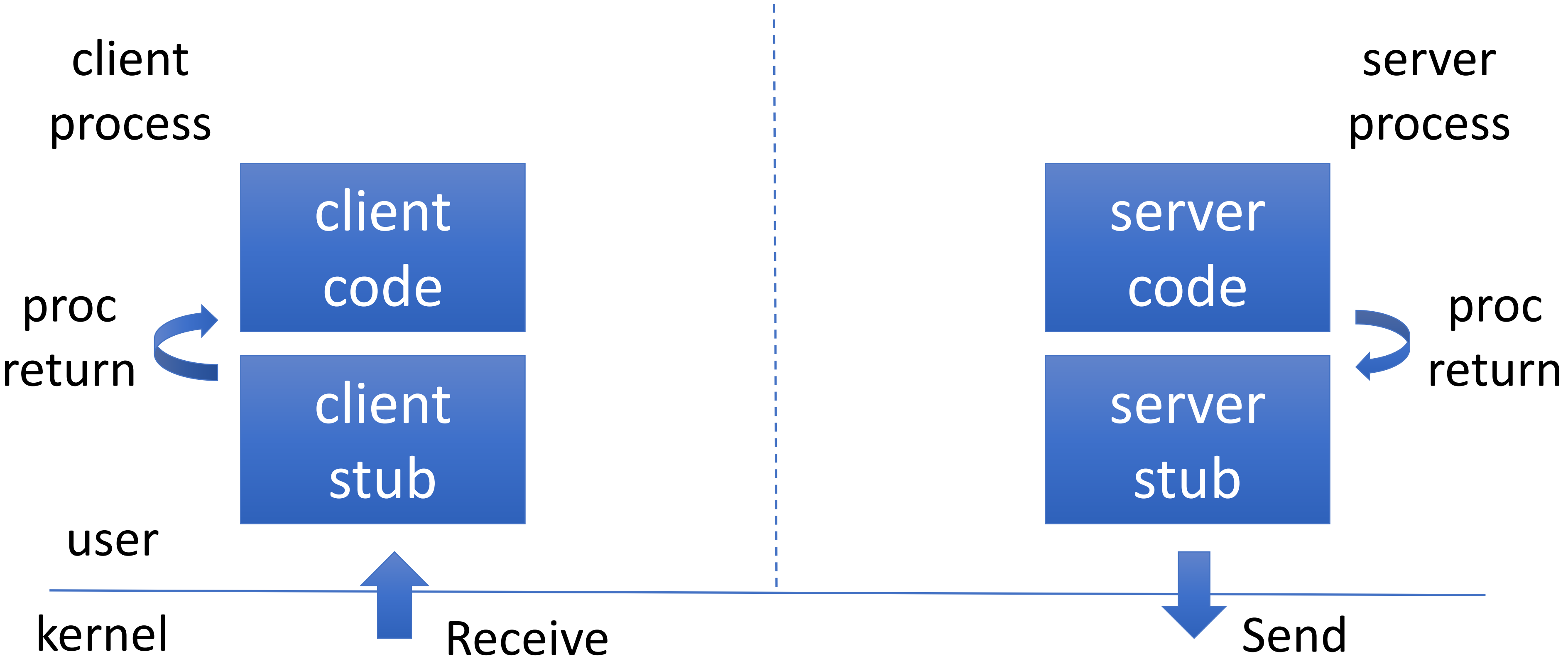
server code

server stub

proc return

Send

RPC Implementation: Return



Further Optional Reading

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

Chapters 25 – 32 (inclusive) https://pages.cs.wisc.edu/~remzi/OSTEP/

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), and Prof. Maurice Herlihy (Brown University)