Recap of Week 2

Pamela Delgado

February 27, 2019

(slides Willy Zwaenepoel)

Key Concepts

- Process
- Linux primitives and process tree
- Multiprocessing and its benefits
- Process switch
- Process scheduler

Key Concepts

- Process
  - Program in execution
- Linux primitives and process tree
- Multiprocessing and its benefits
- Process switch
- Process scheduler

Key Concepts

- Process
- Linux primitives and process tree
  - fork() / exec() / wait() / exit()
  - Use in shell
- Multiprocessing and its benefits
- Process switch
- Process scheduler

Key Concepts

- Process
- Linux primitives and process tree
- Multiprocessing and its benefits
  - Switching the CPU to another process on I/O
  - Lower response time and better utilization
- Process switch
- Process scheduler

Key Concepts

- Process
- Linux primitives and process tree
- Multiprocessing and its benefits
- Process switch
  - Change of process using the CPU
  - Save and restore registers and other info
- Process scheduler

Key Concepts

- Process
- Linux primitives and process tree
- Multiprocessing and its benefits
- Process switch
- Process scheduler
  - Decides which process to run next

Scheduler Implementation

- Must be very efficient
- Runs (at least) every Δ
- If Δ = 10 msec, scheduler run takes 1 msec

  10% of your machine is gone!
- Be careful with large number of processes

Week 3 – Part 1
Application Multiprocess Structuring and Interprocess Communication

Pamela Delgado

March 6, 2019

(slides Willy Zwaenepoel)

So far

- One program
  = one process
- Examples:
  – Shell
  – Compiler
  – …

This is not always the case

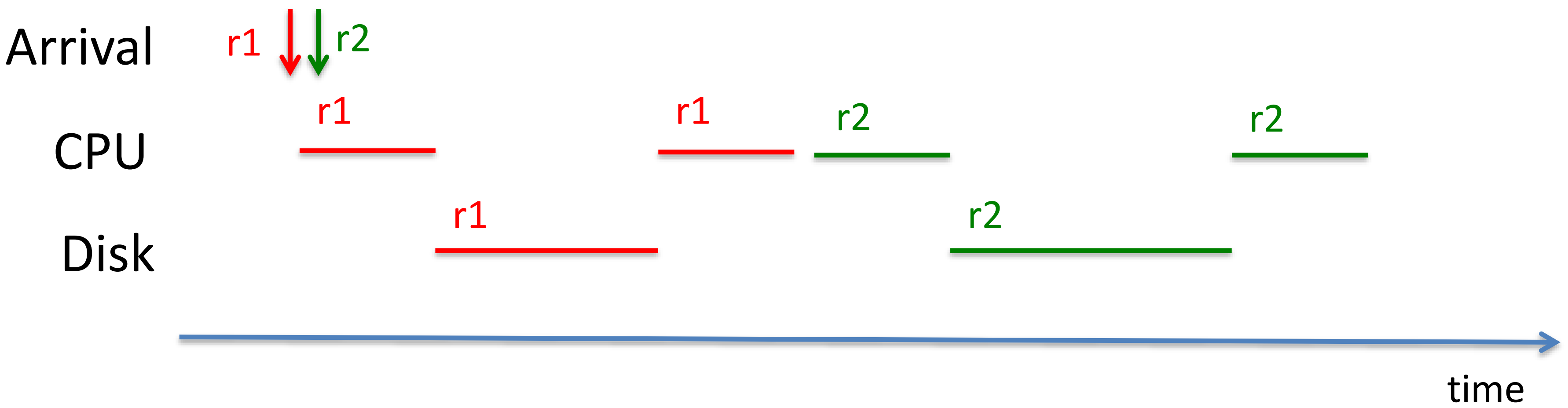
- One program
  = multiple processes
- Example:
  – Web server

(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

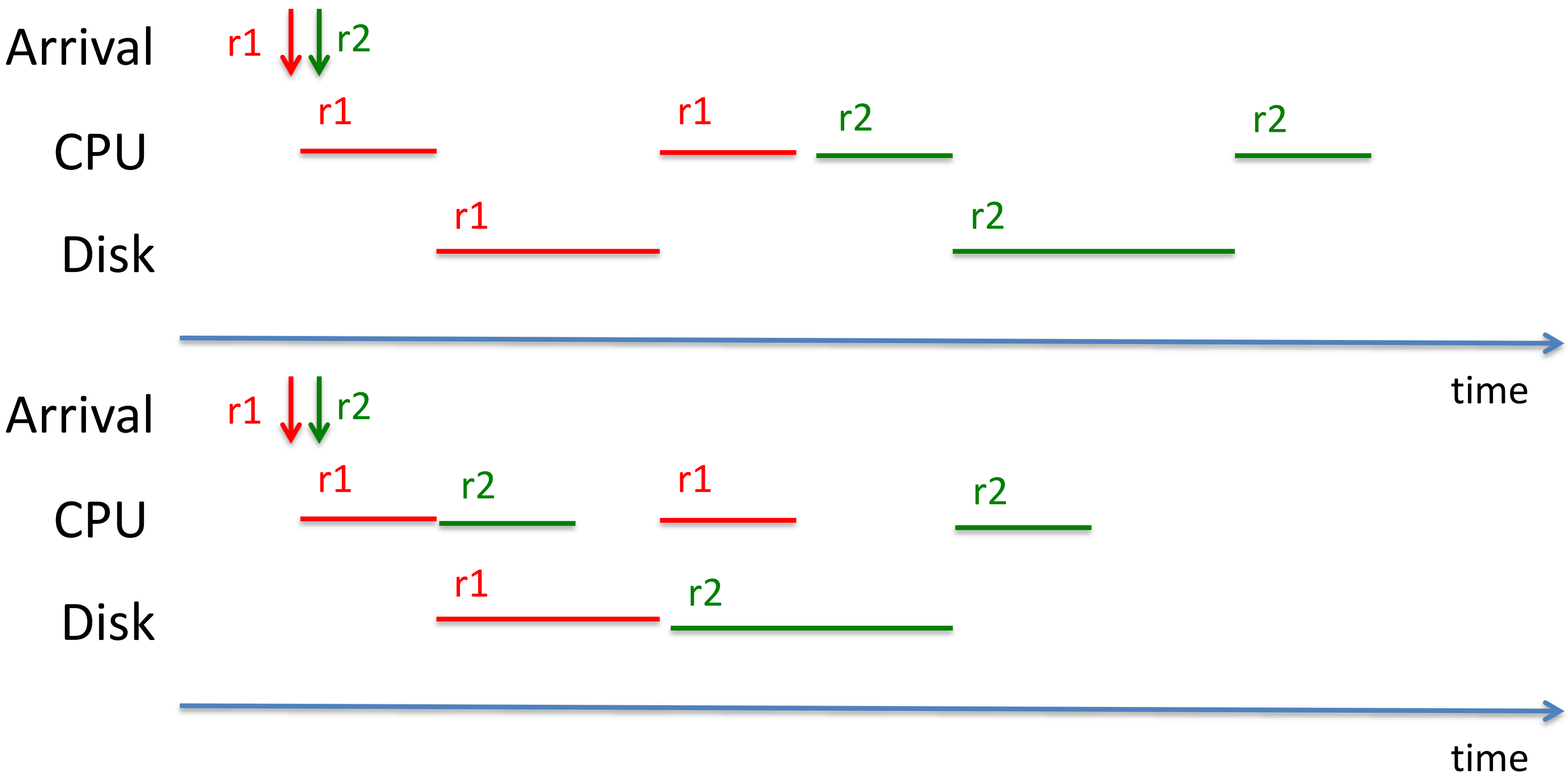


Multiprocess Web Server

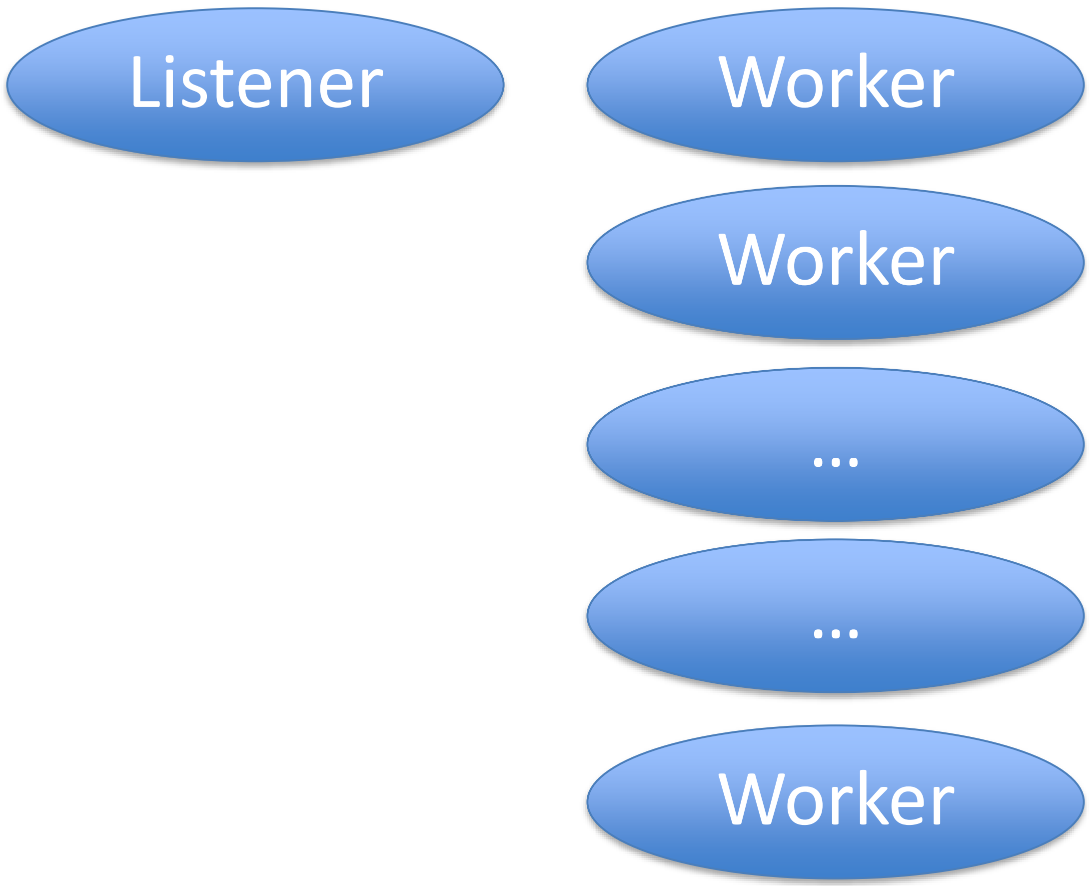
```
ListenerProcess {
    forever {
        wait for incoming request
        CreateProcess( worker, request )
        }
}

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

Multi vs. Single-process Web Server

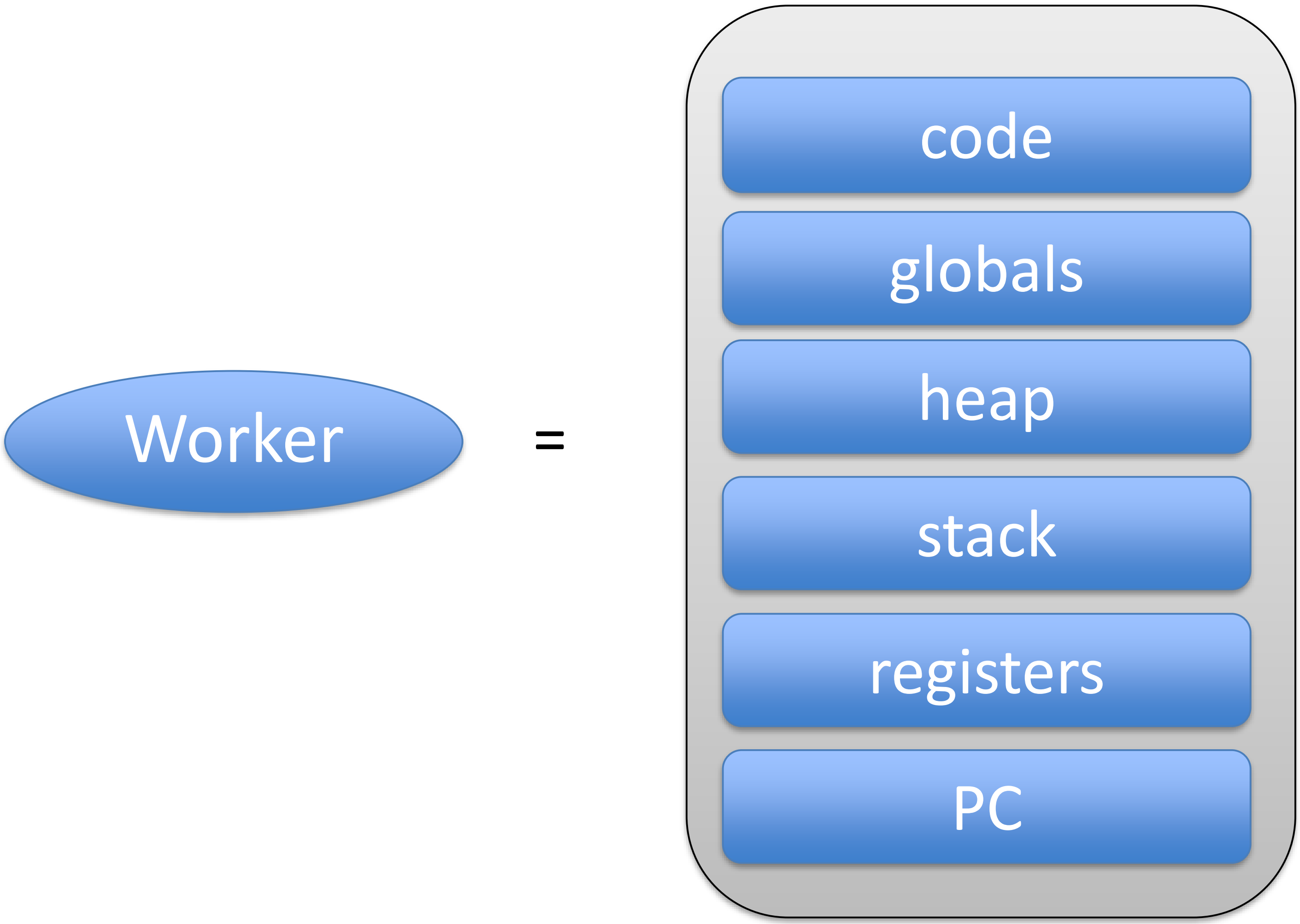
Example: Web server receives two requests in quick succession



Multiprocess Web Server



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

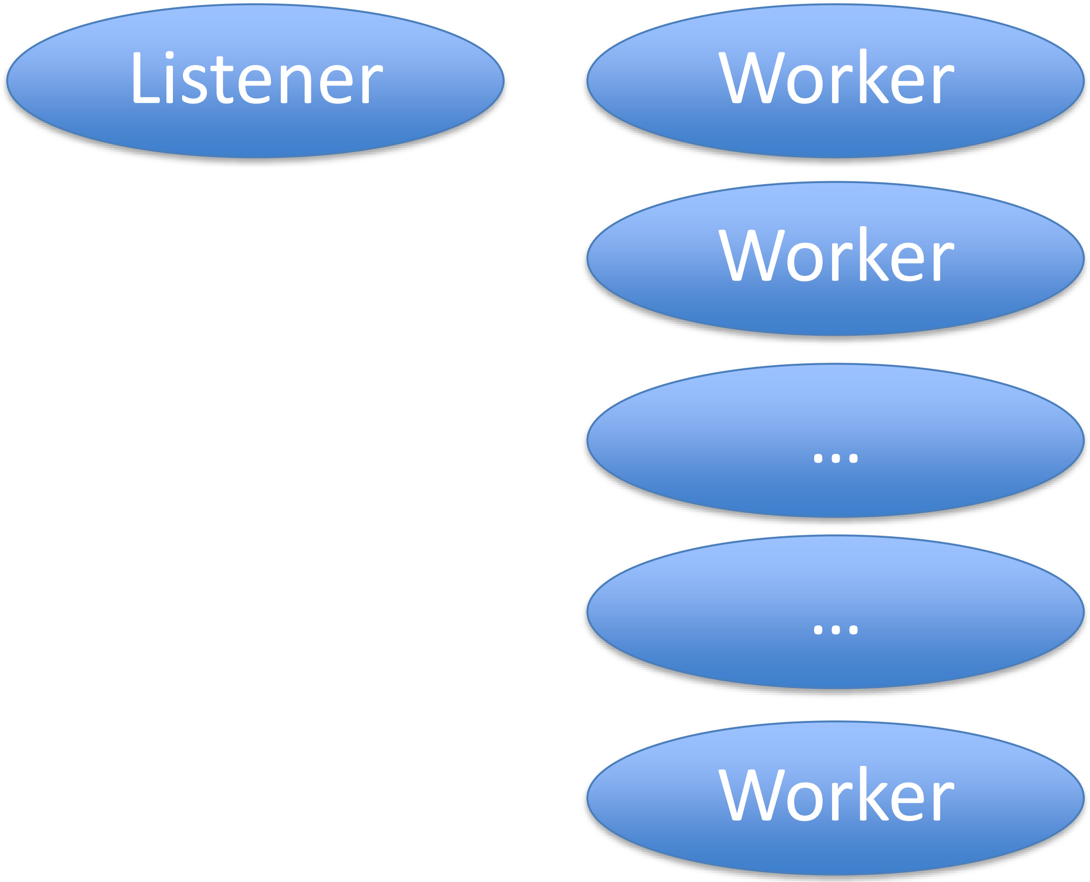
Process Pool

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

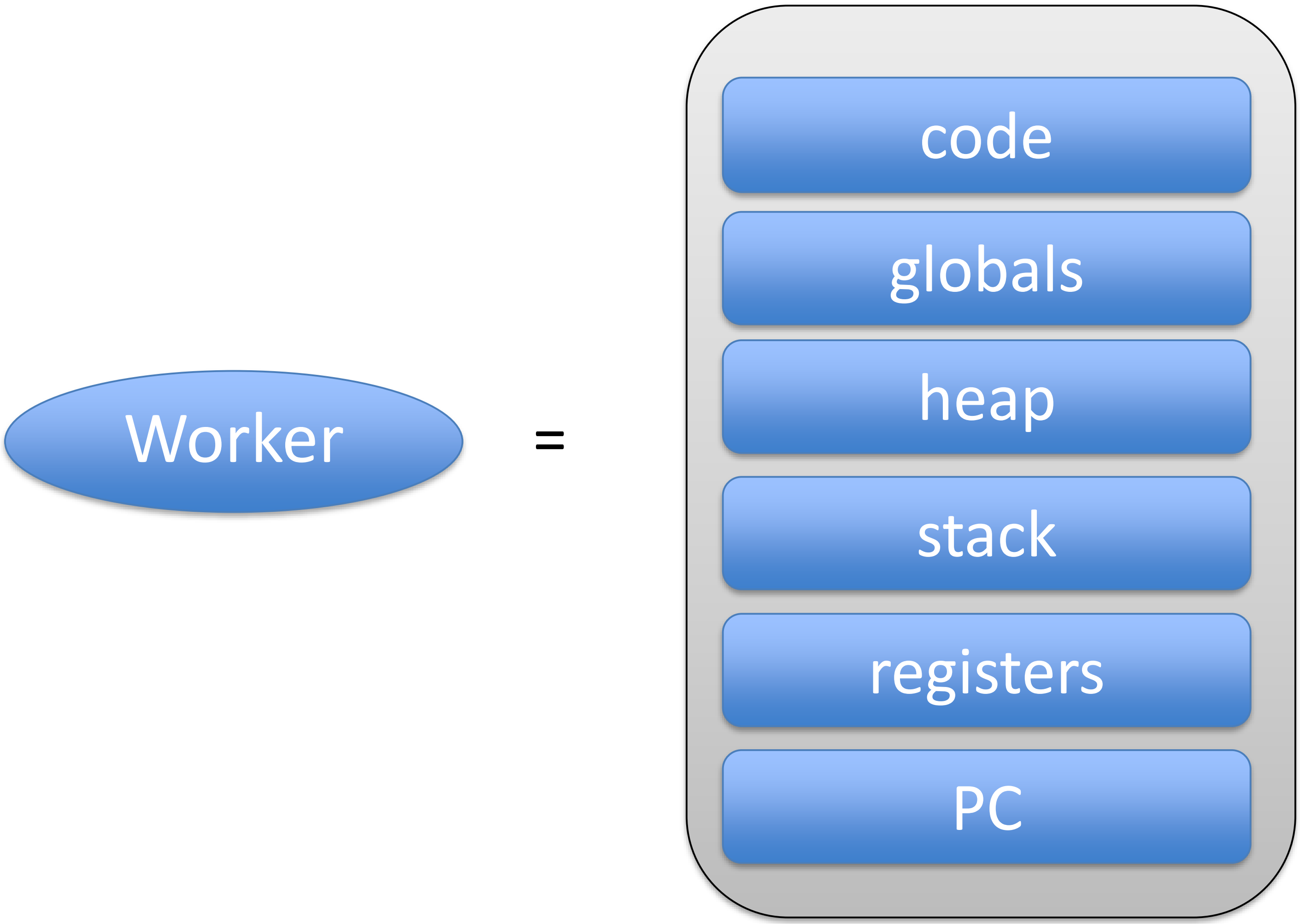
Multiprocess Web Server with Process Pool

```
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

Key Concepts

- Message passing
- Remote procedure call

Where do you need IPC?

Multiprocess Web Server with Process Pool

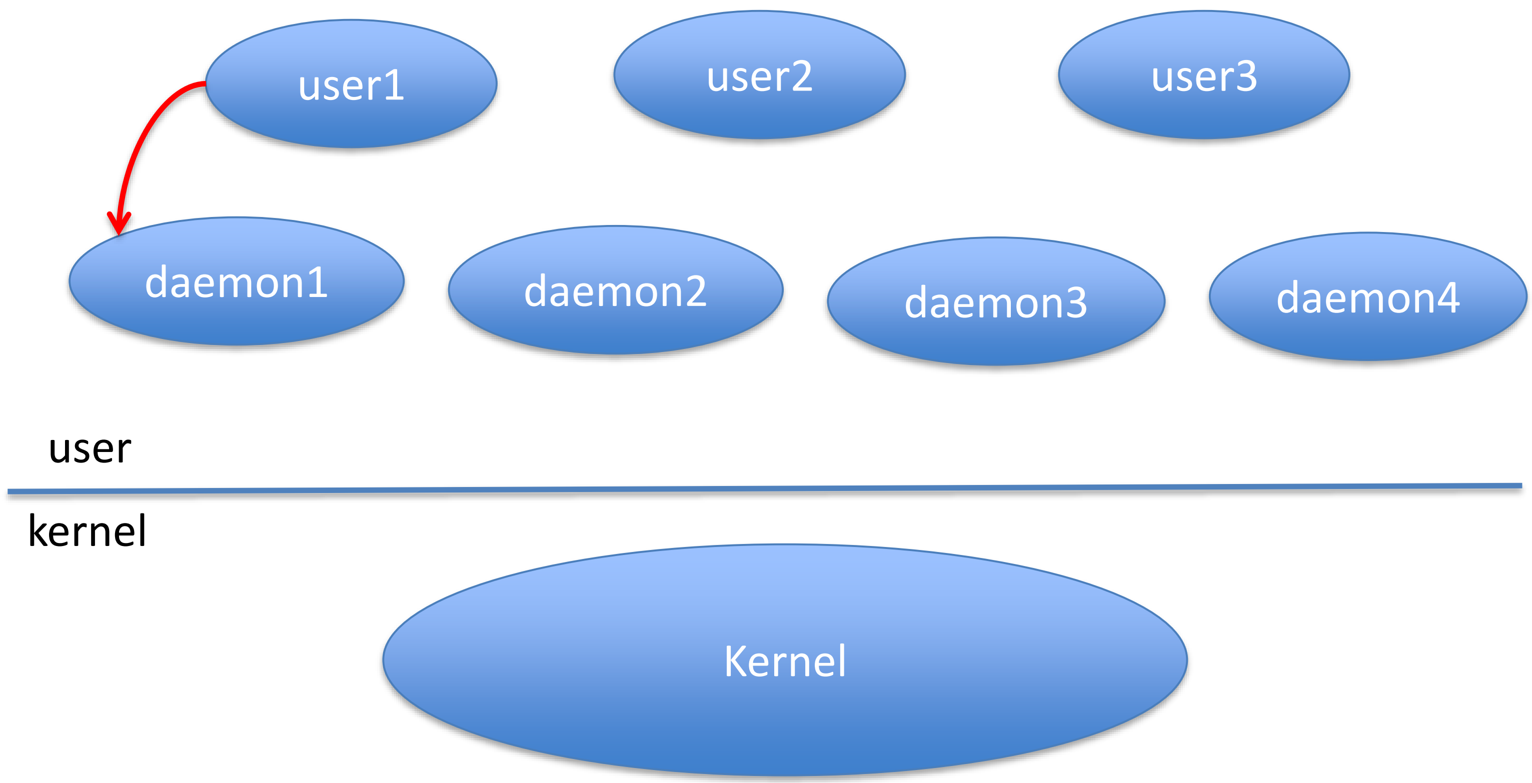
```
ListenerProcess {
    for( i=0; i<MAX_PROCESSES; i++ )
        process[i] = CreateProcess( worker )
    forever {
        receive incoming request
        send( request, process[?] )
    }
}

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

Multiprocess Web Server with Process Pool

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
    }
}
```

More Client-Server Communication: Access to System Processes



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
    }
}
```

Multiprocess Web Server with Process Pool

Communication Cooperating Processes

```
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
    }
}
```

Where do you need IPC?

- Between client and server
- Between cooperating processes

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

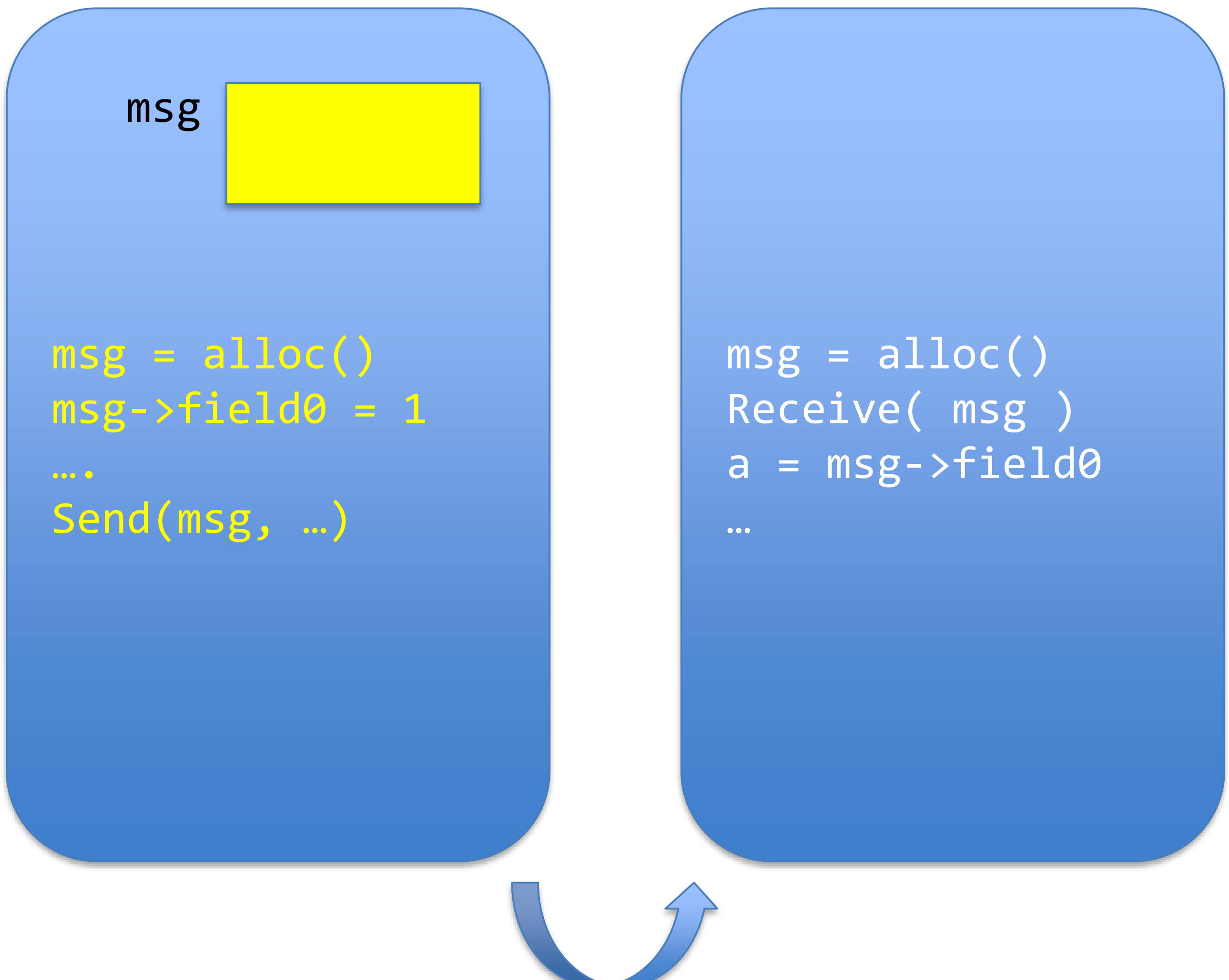
msg

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

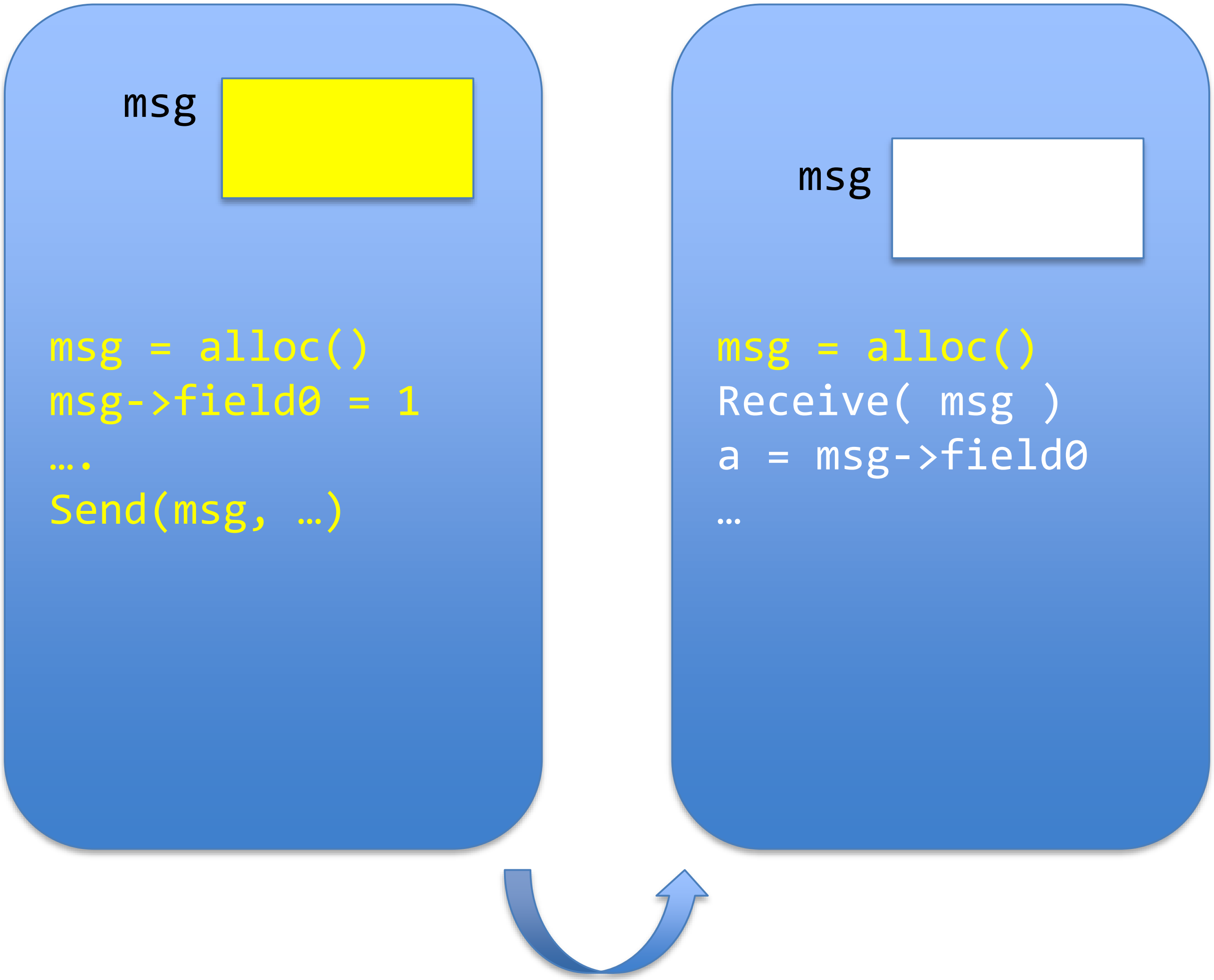
```
msg = alloc()
Receive( msg )
a = msg->field0
…
```

Message Passing Send / Receive

Message Passing Send / Receive



Message Passing Send / Receive



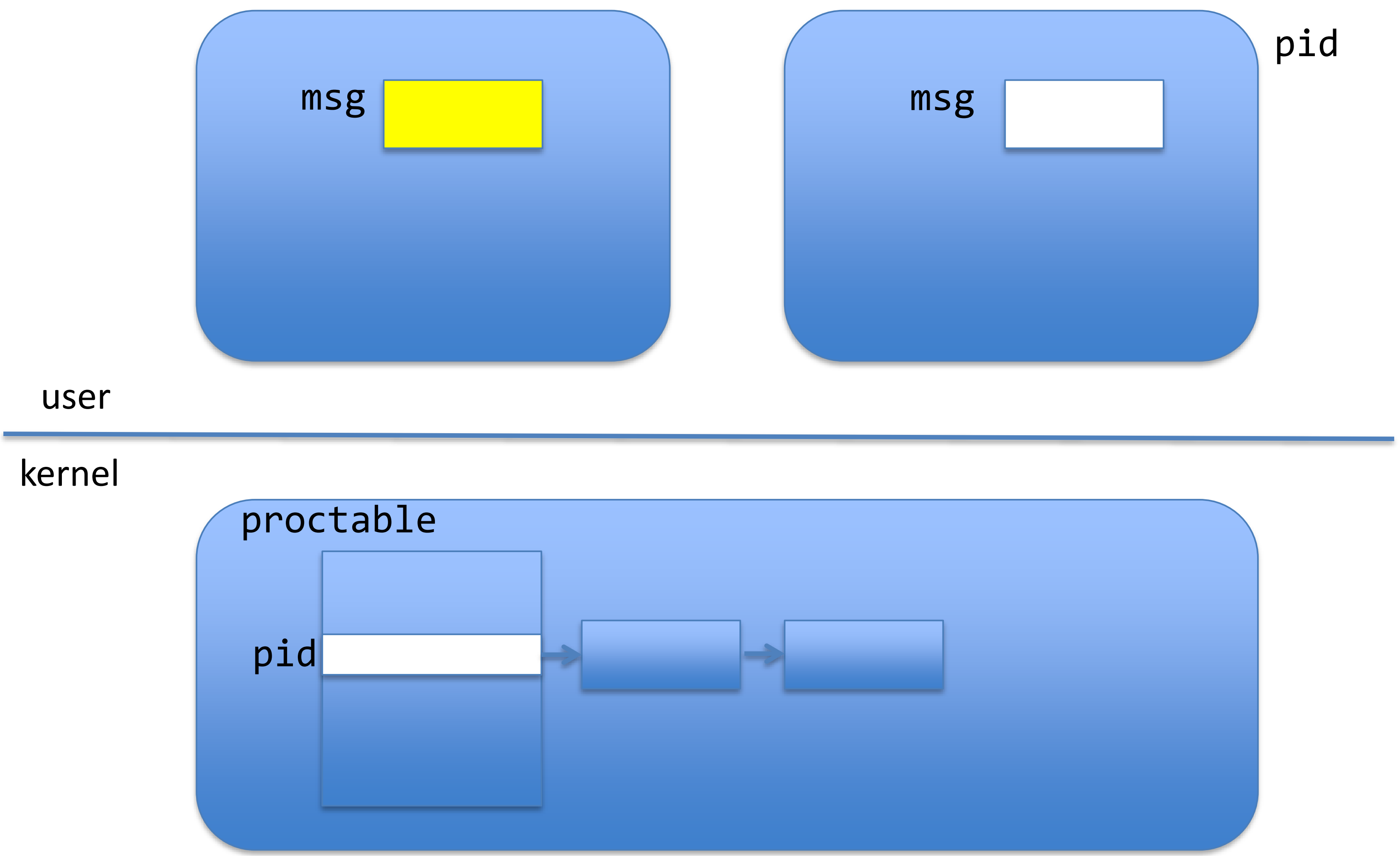
Message Passing Send / Receive

Message Passing Send / Receive

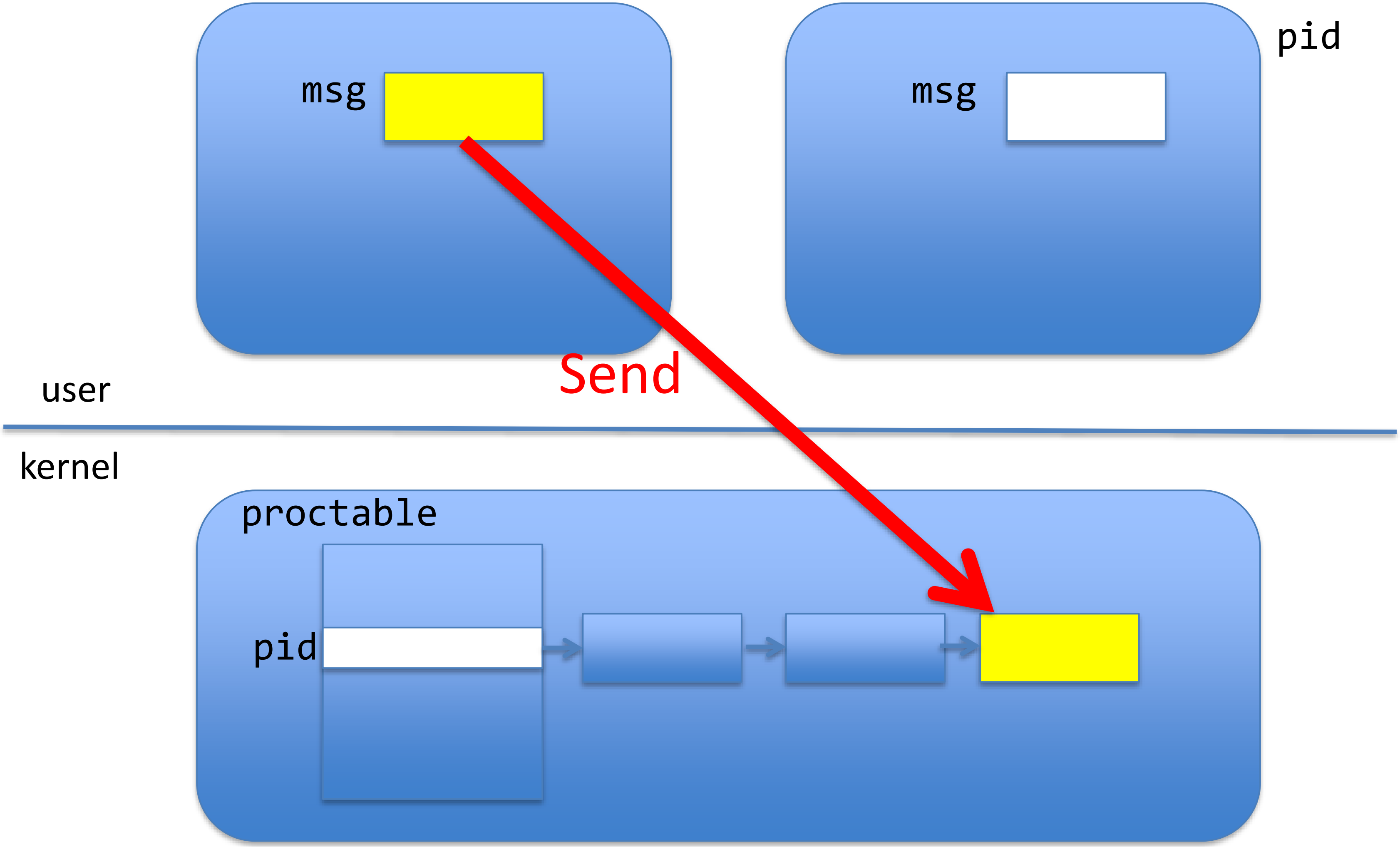
Message Passing

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

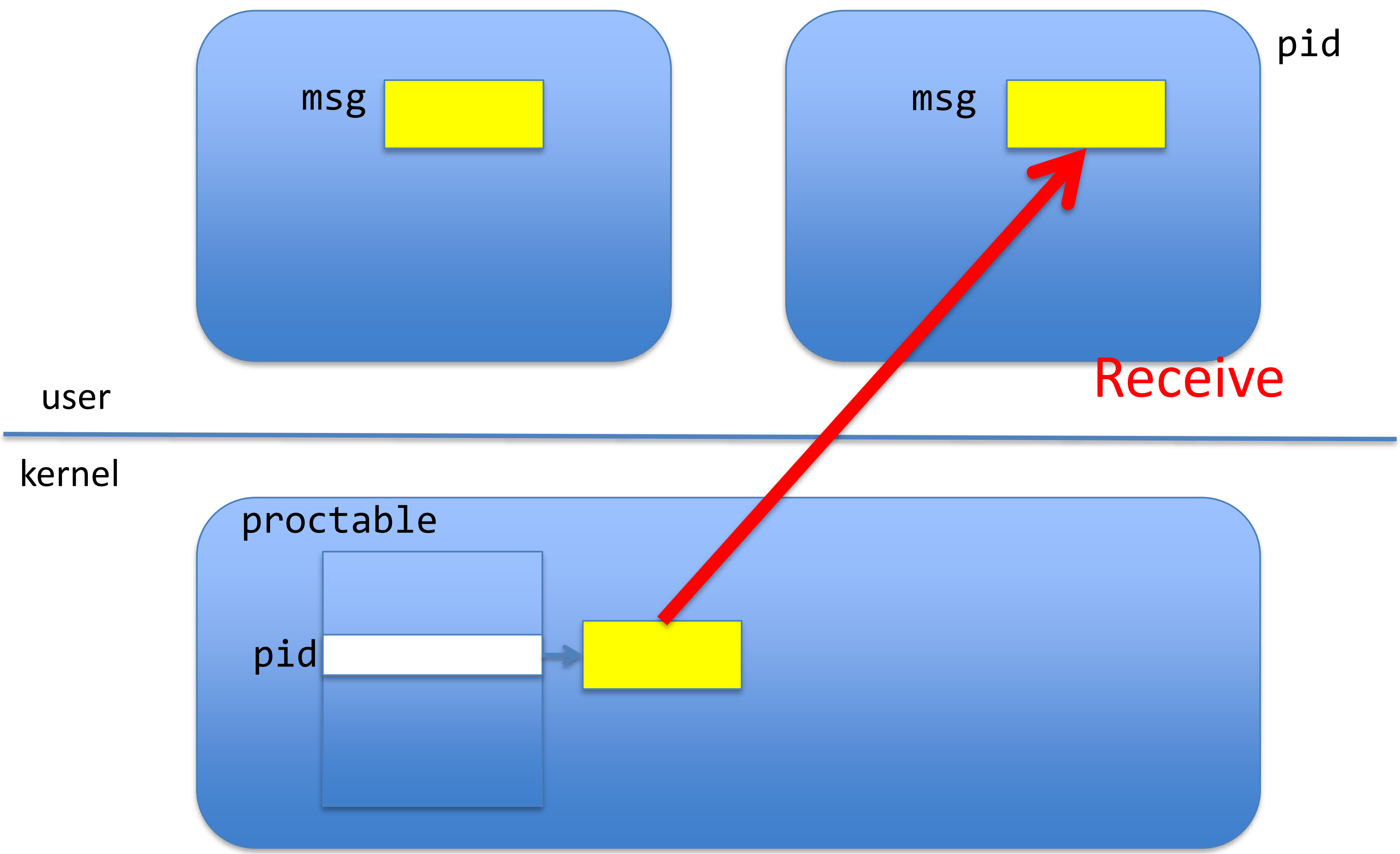
Message Passing Implementation



Message Passing Implementation



Message Passing Implementation



Message Passing Alternatives

- Symmetric / asymmetric addressing
- Blocking / non-blocking

Symmetric Addressing

- Send(msg, topid)
- Receive(msg, frompid)

- Message is (typically) a struct
- topid, frompid 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 Non-blocking Send

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

Blocking or Non-blocking Receive

- Non-blocking
  - 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 )
    }
}
```

Non-blocking 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
    }
}
```

Returning to (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 Interface

- Interface
  - List of remotely callable procedures
  - With their arguments and return values
- Example: file system interface
  - Open(string filename) returns int fd
  - …

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

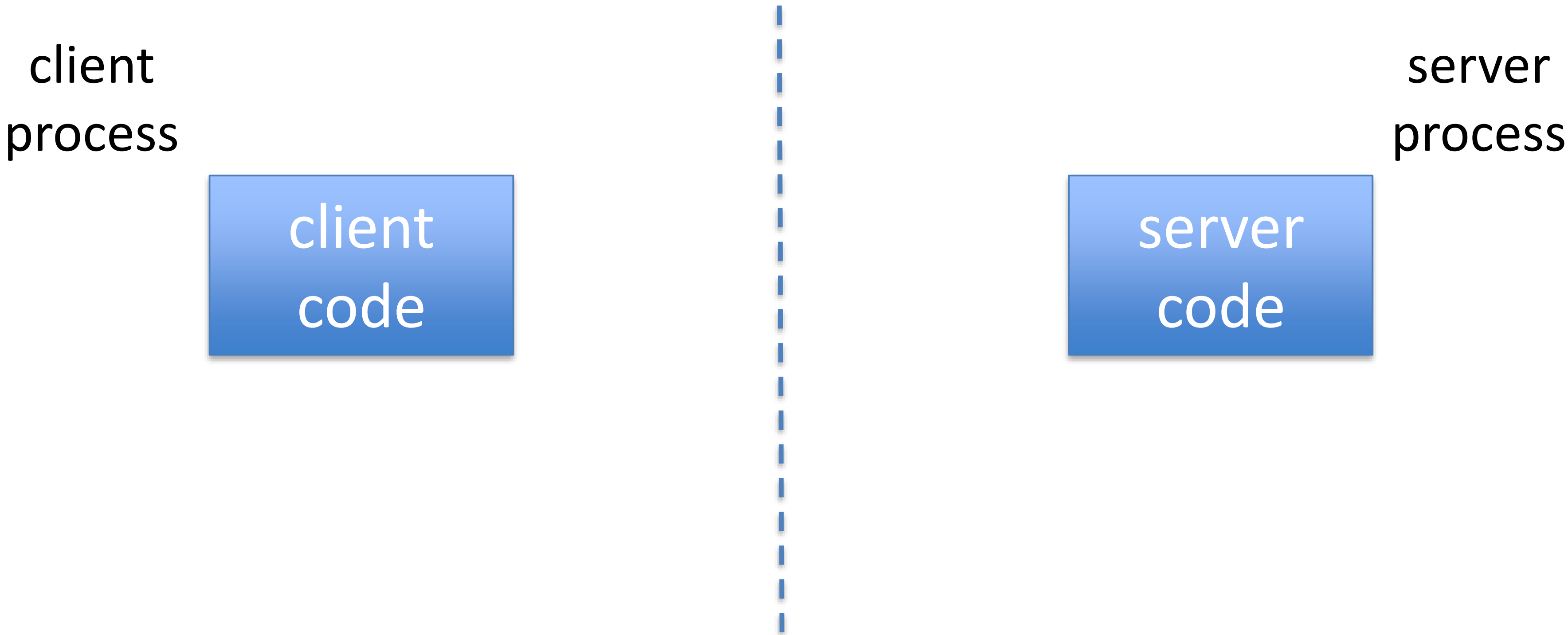
Client Stub

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

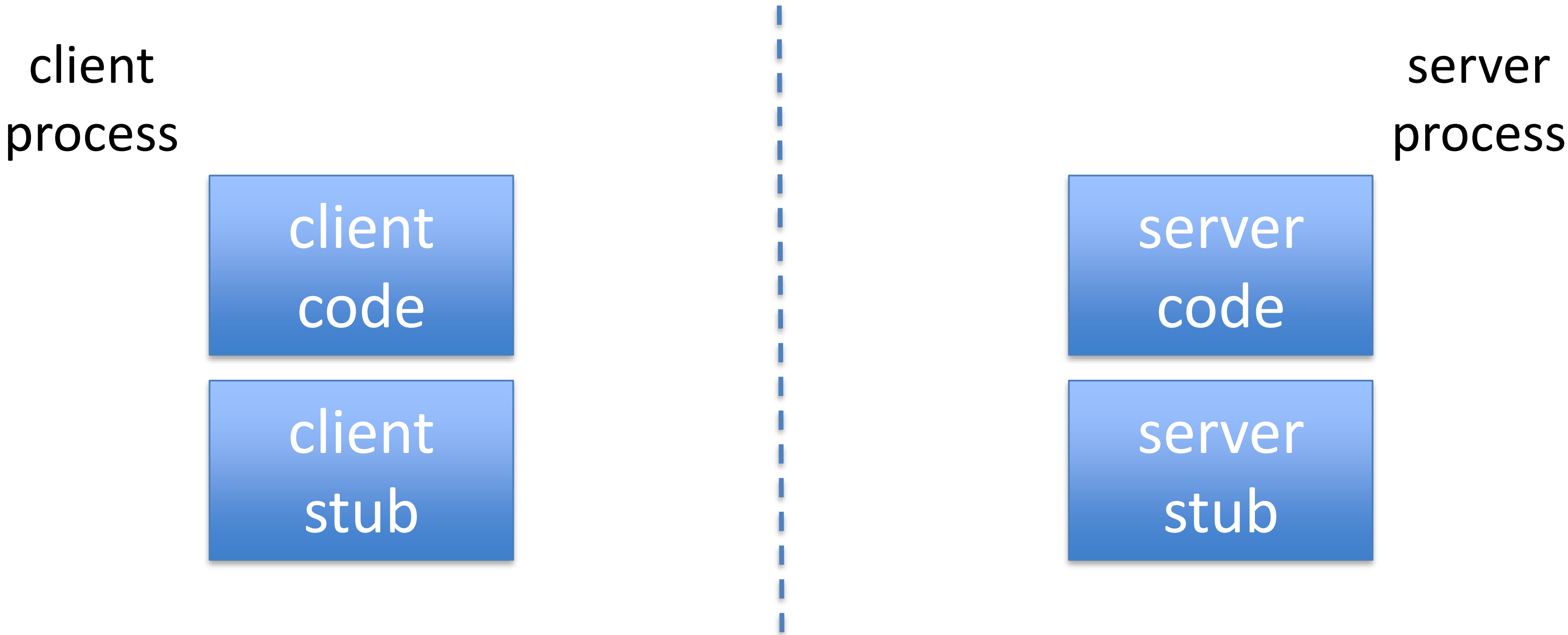
Server Stub

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

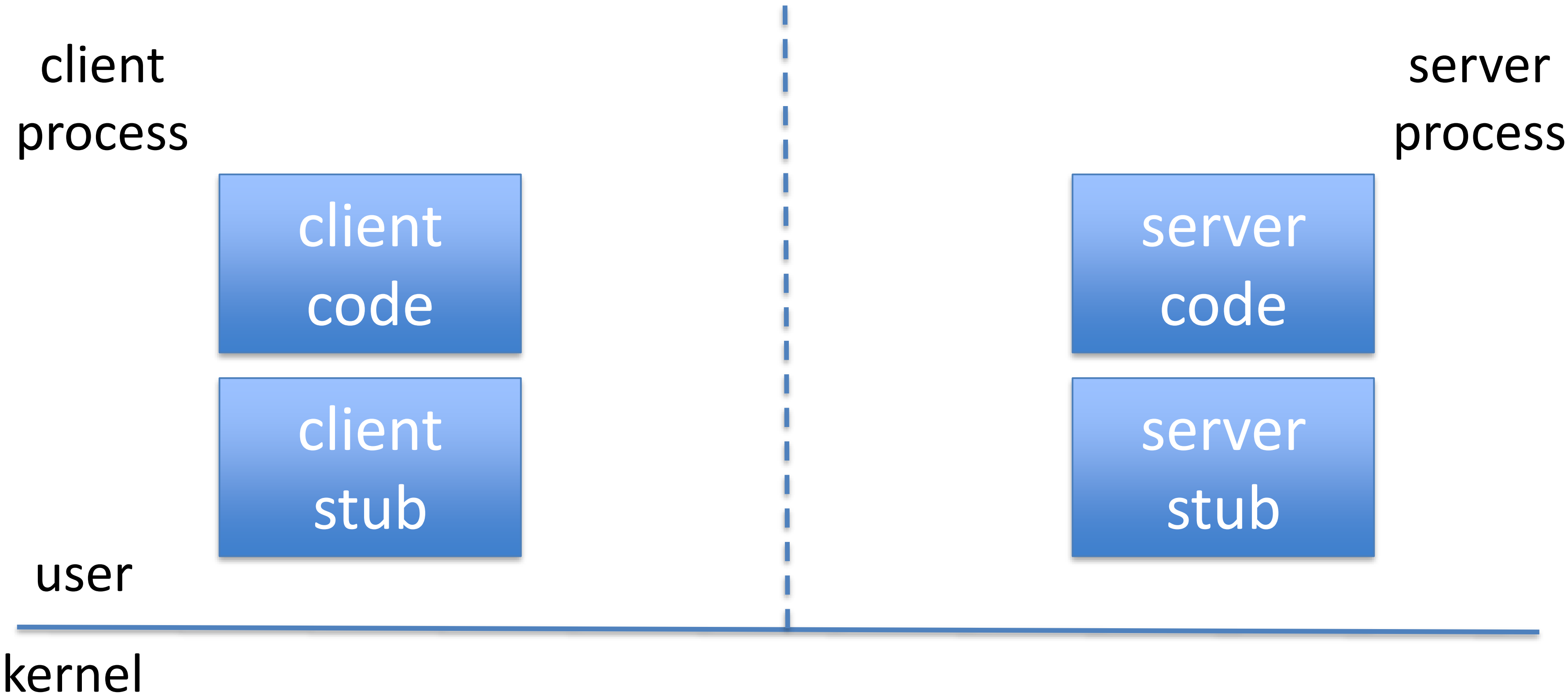
RPC Implementation



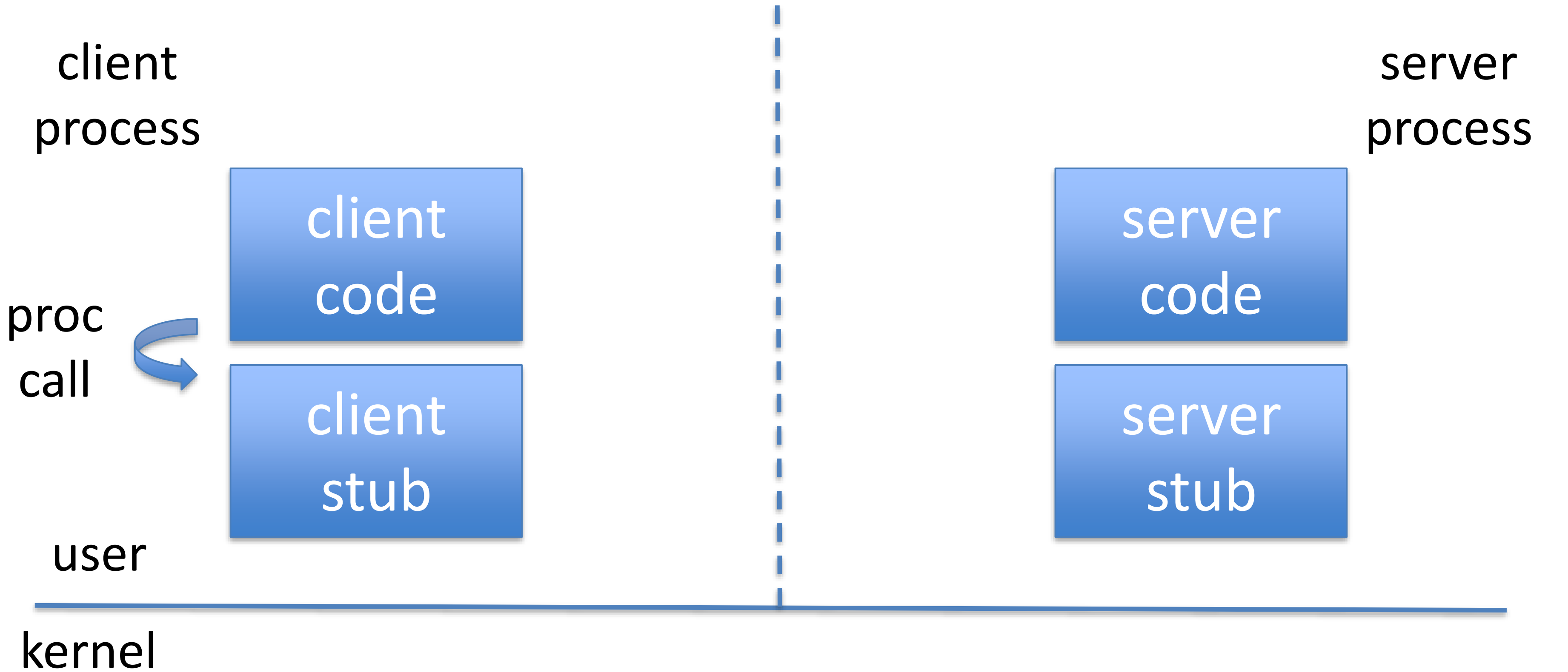
Client and Server Stubs



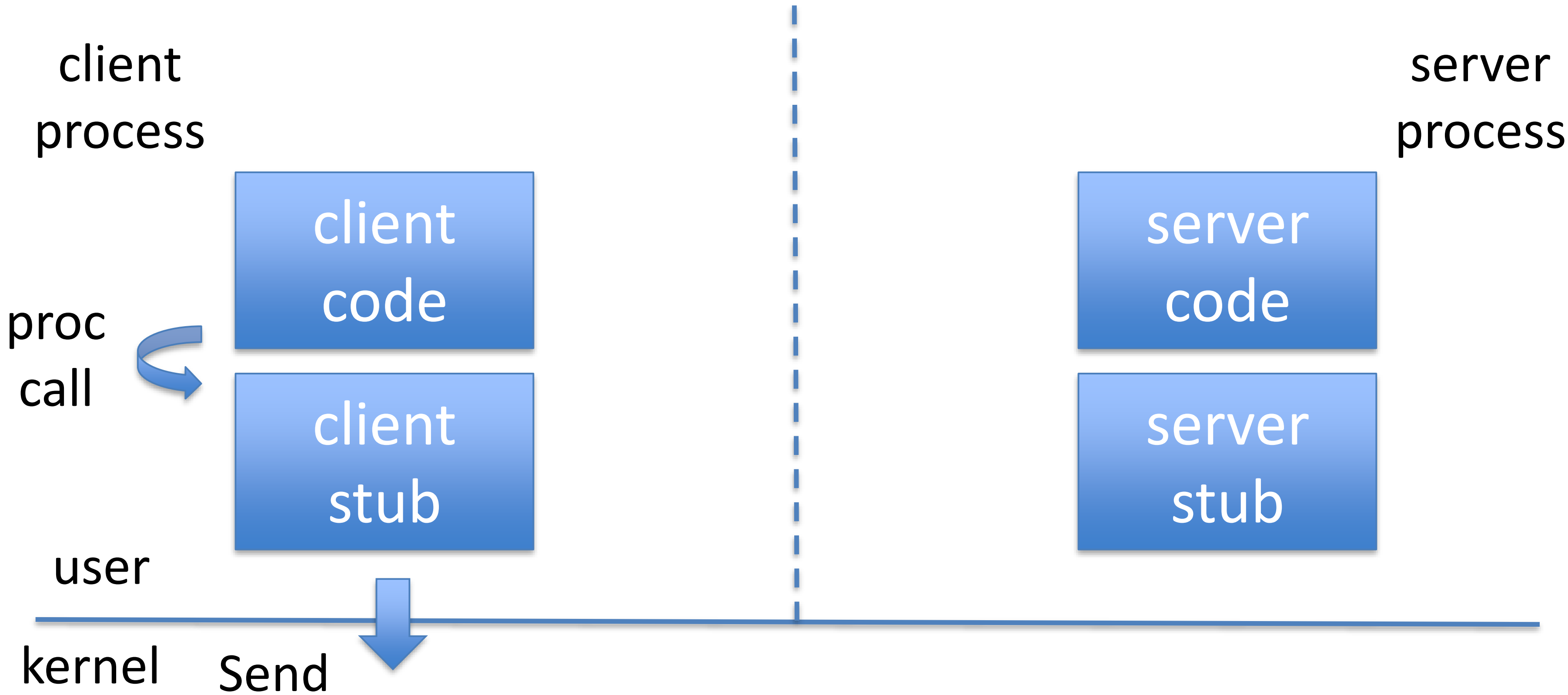
RPC Implementation: Call



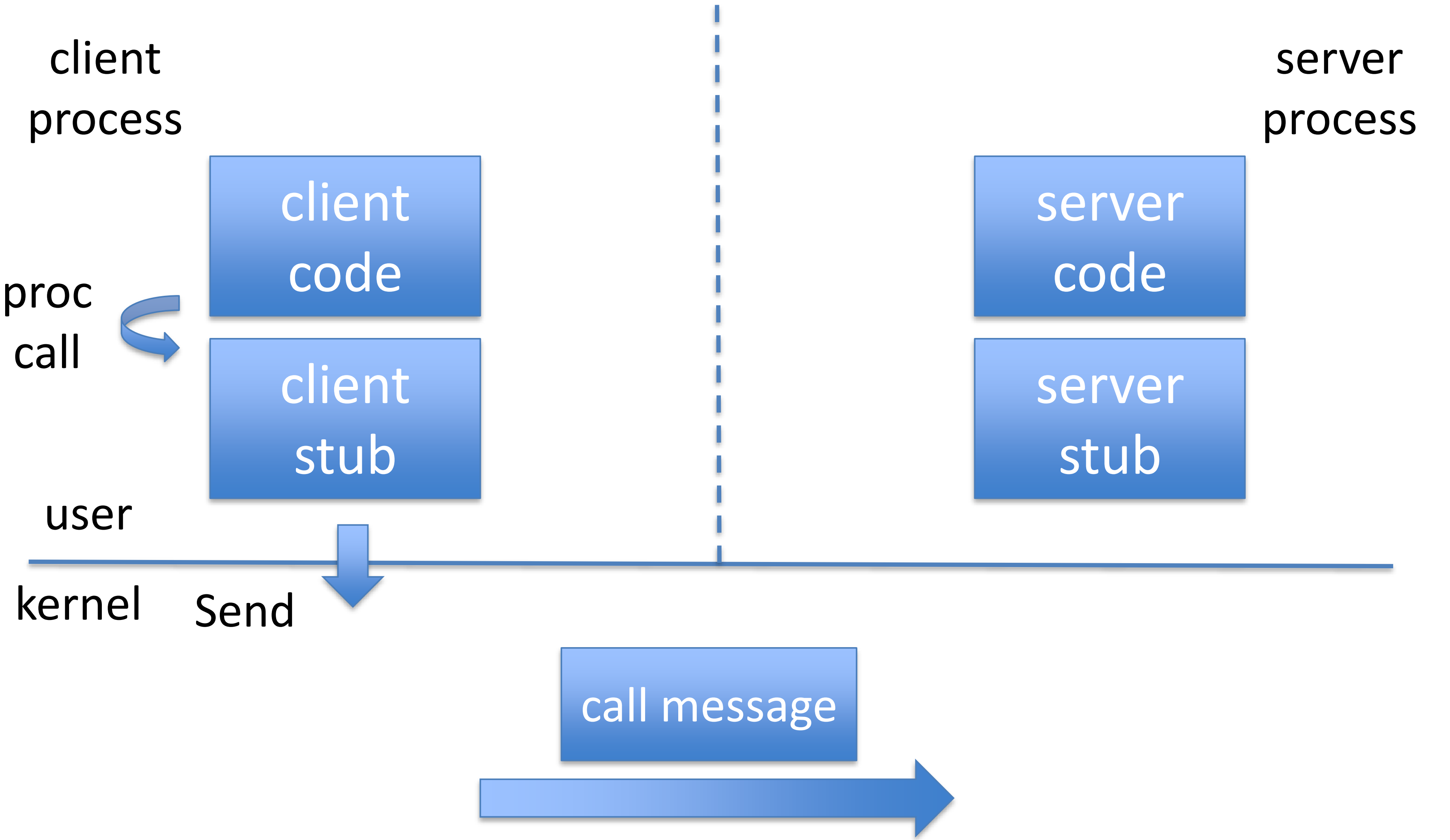
RPC Implementation: Call



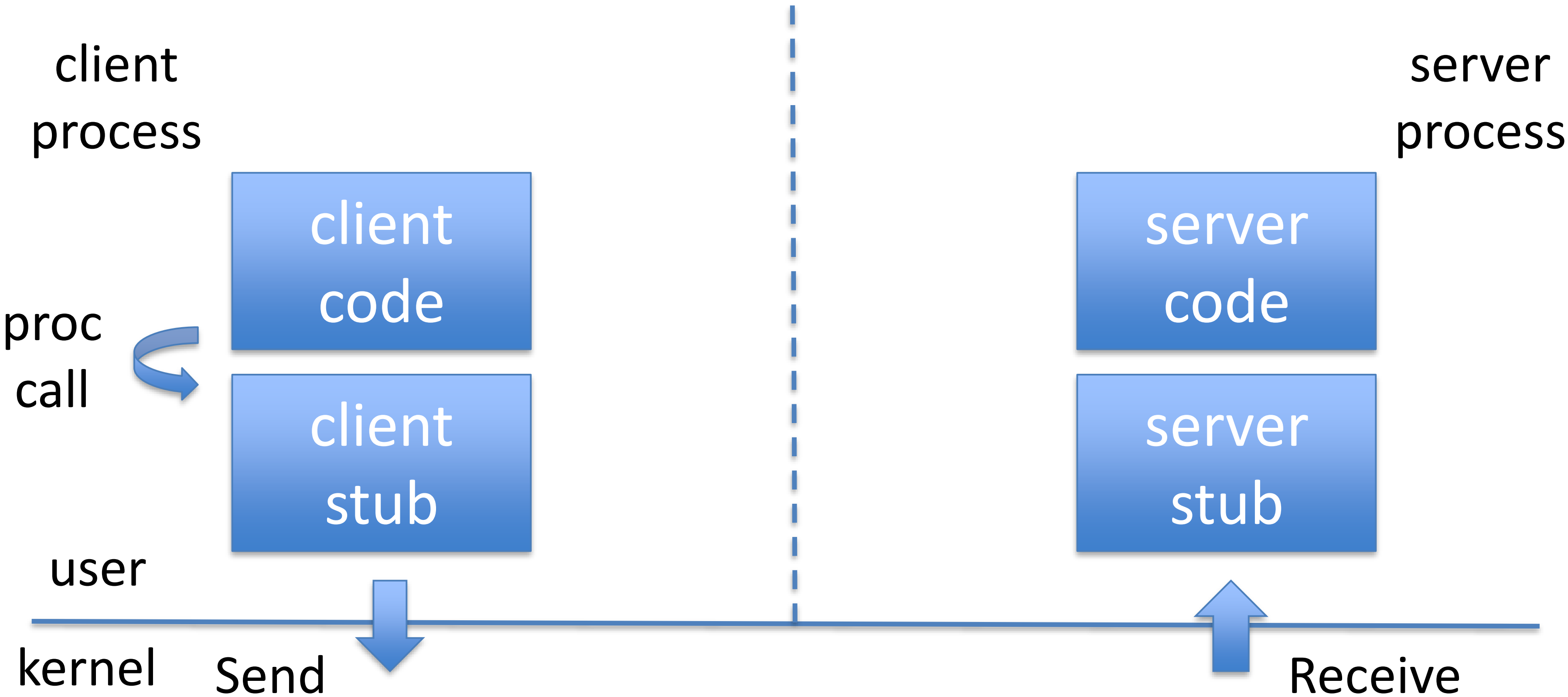
RPC Implementation: Call



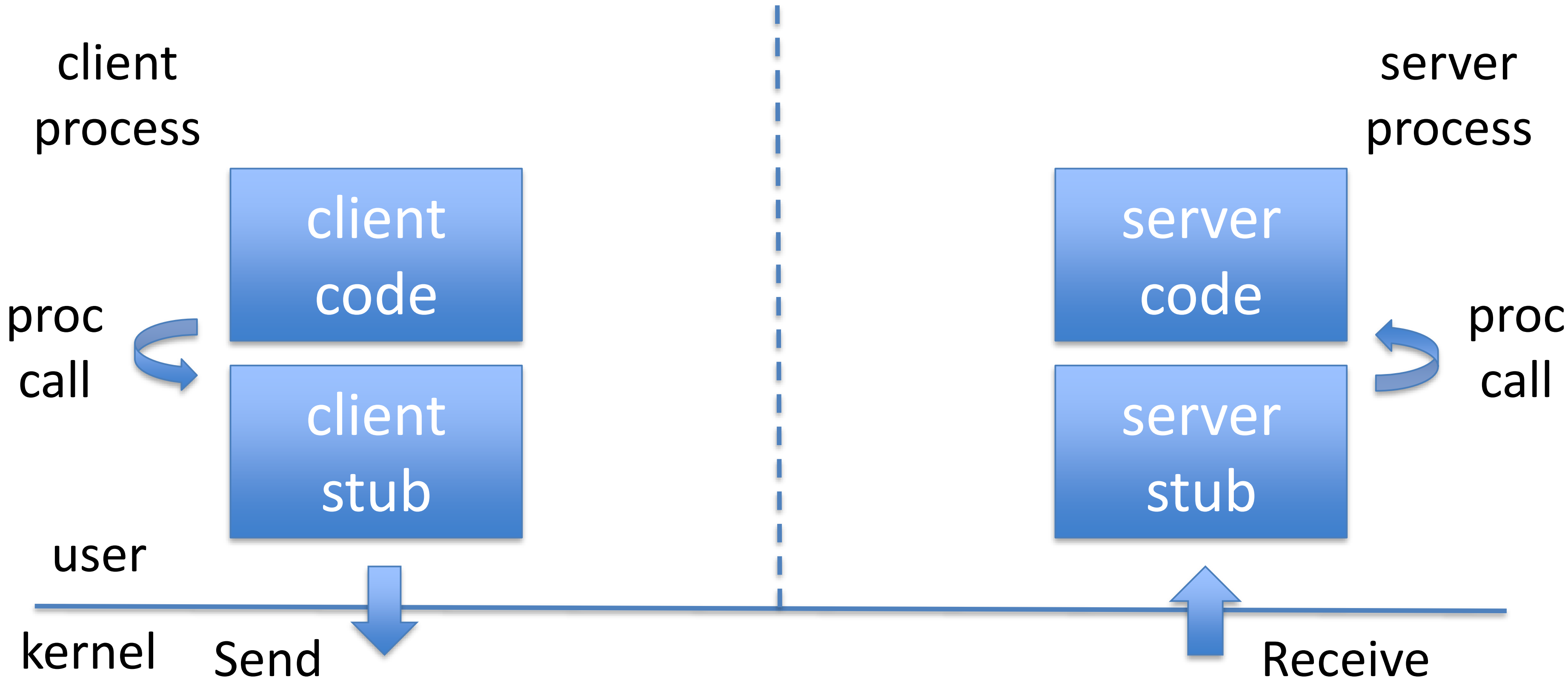
RPC Implementation: Call



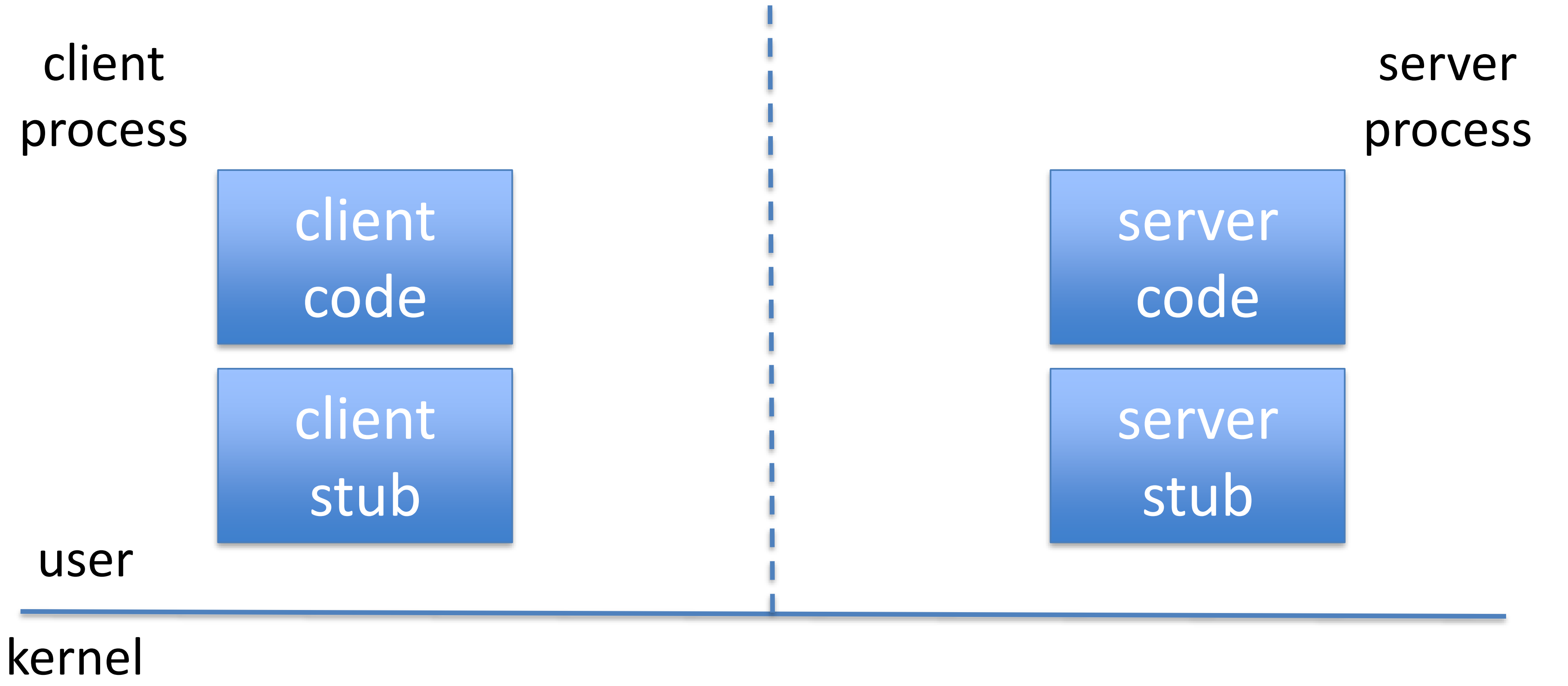
RPC Implementation: Call



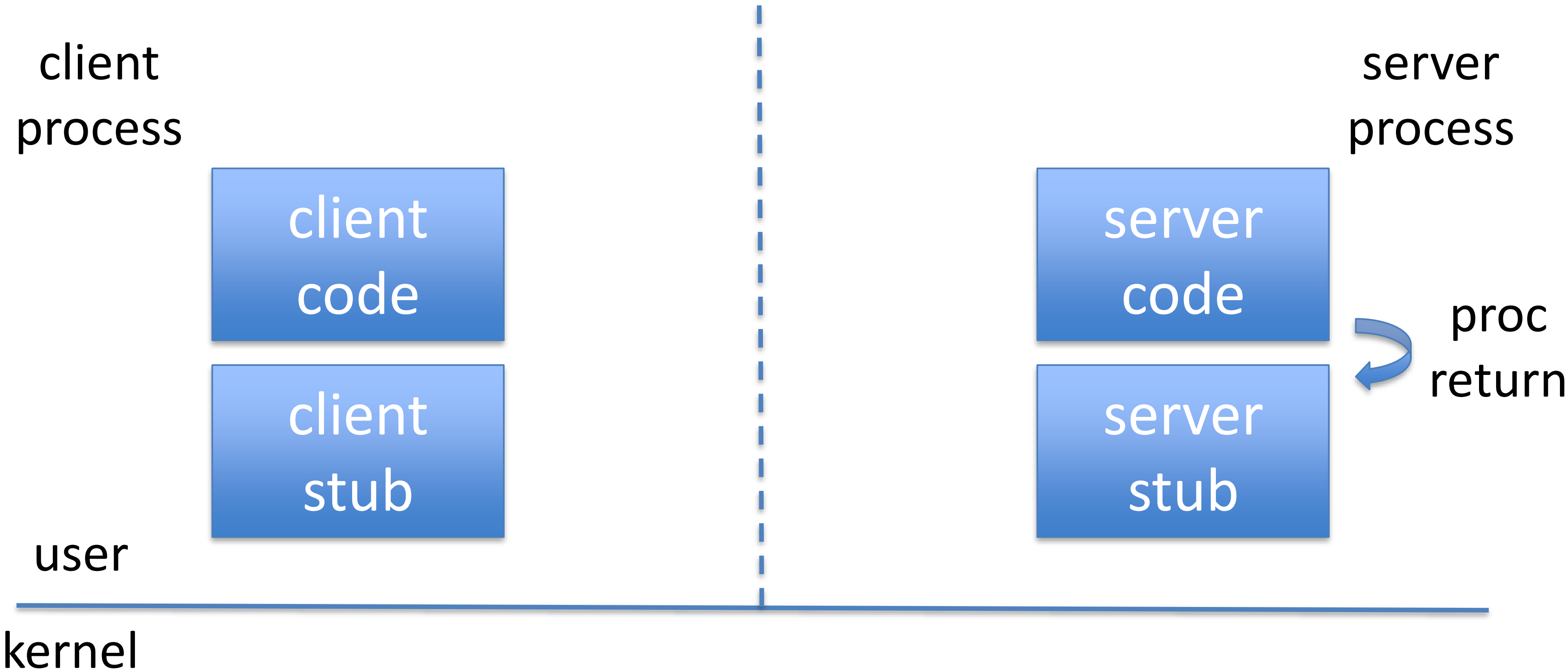
RPC Implementation: Call



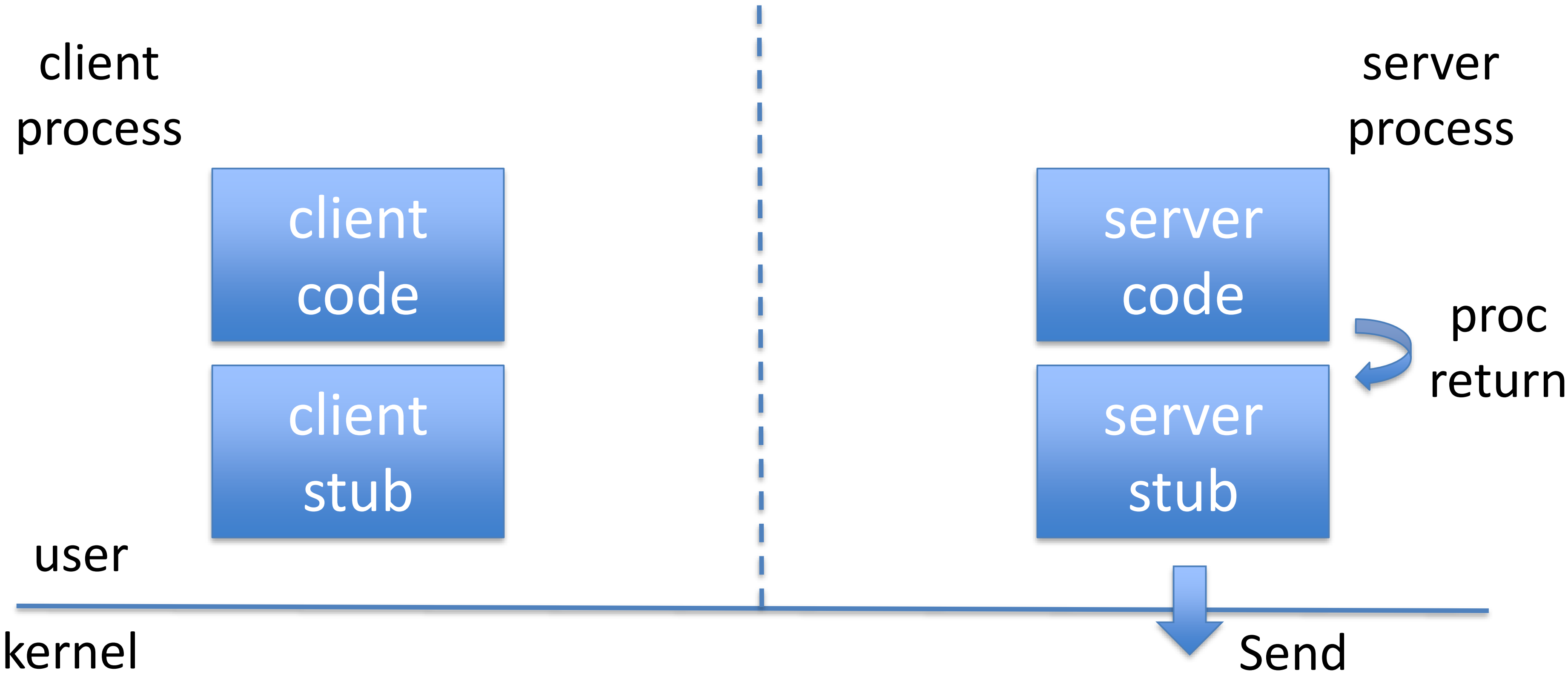
RPC Implementation: Return



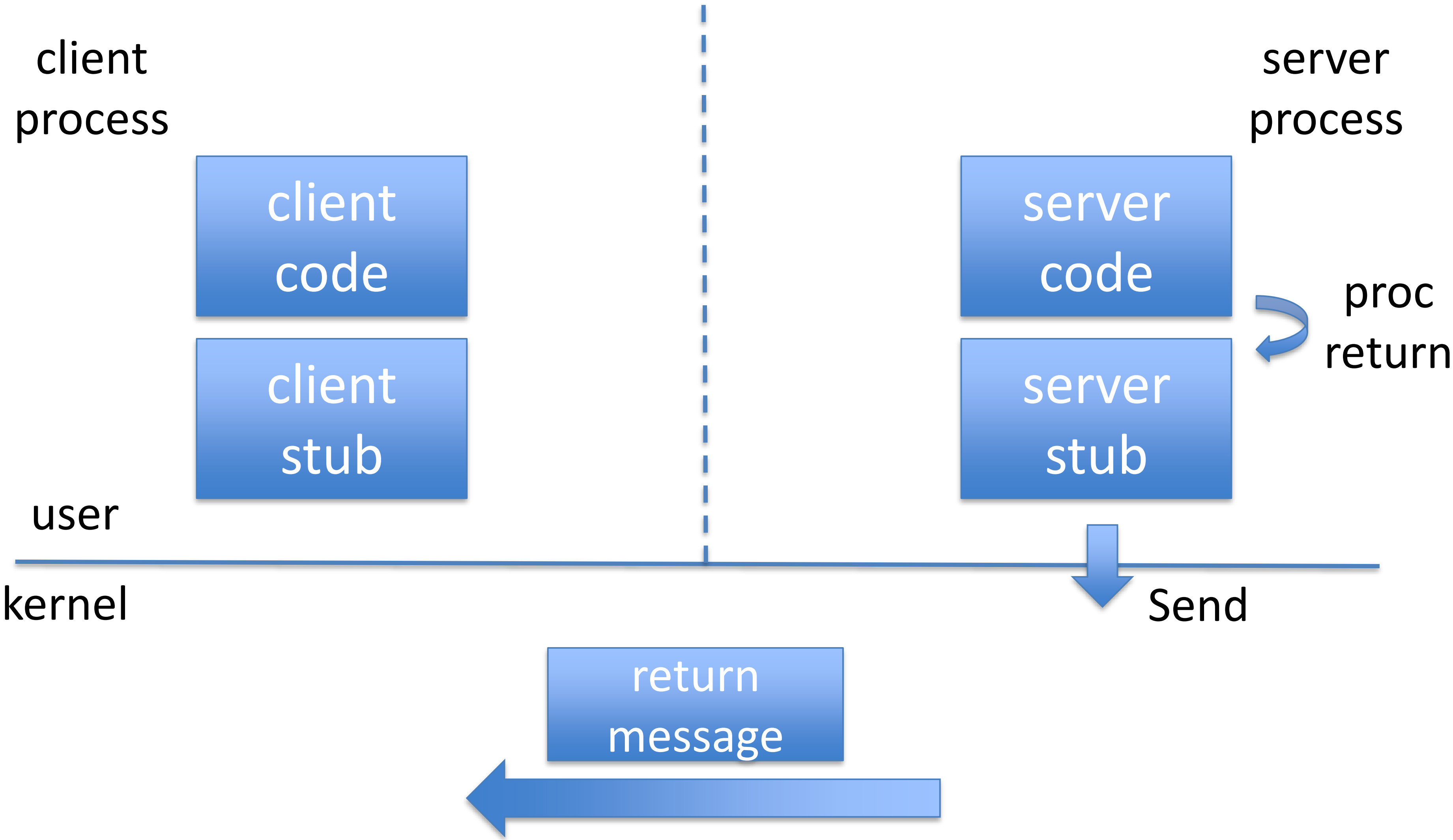
RPC Implementation: Return



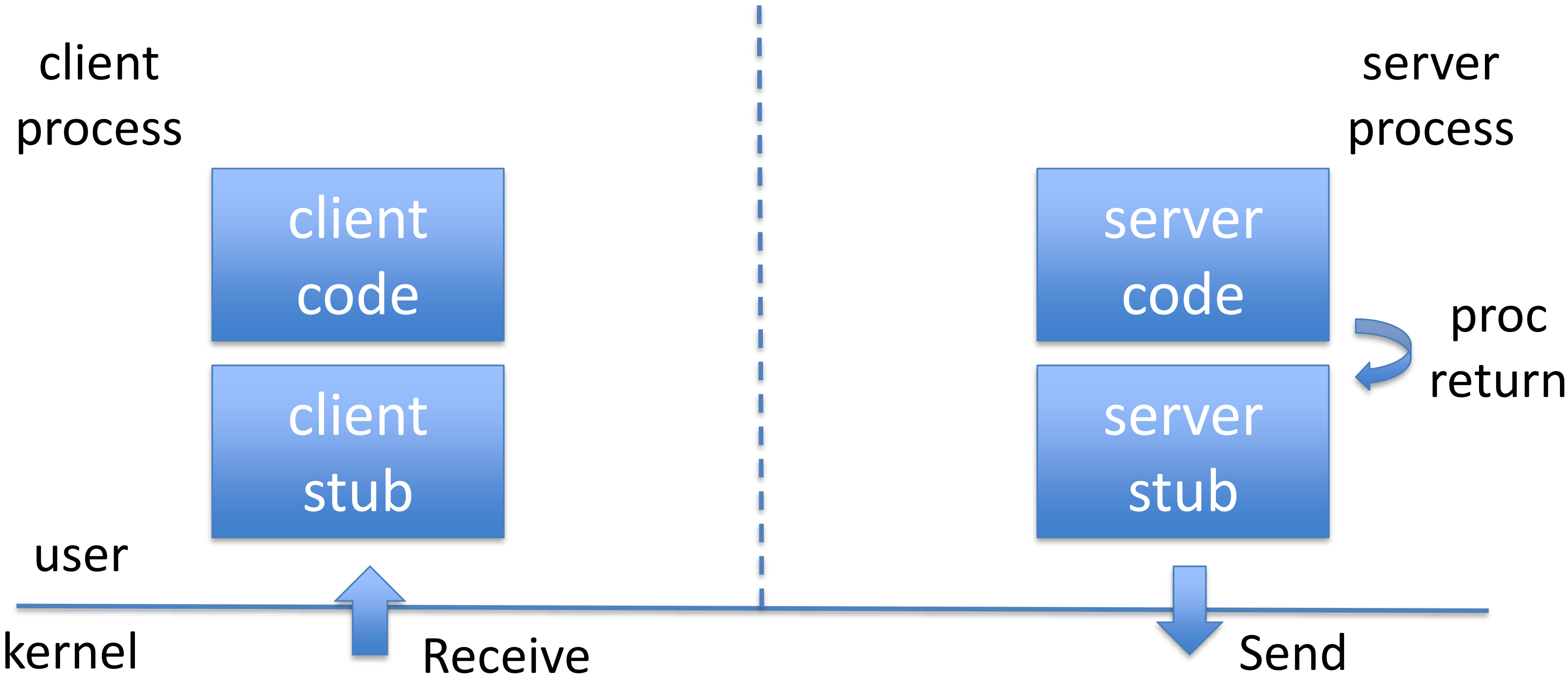
RPC Implementation: Return



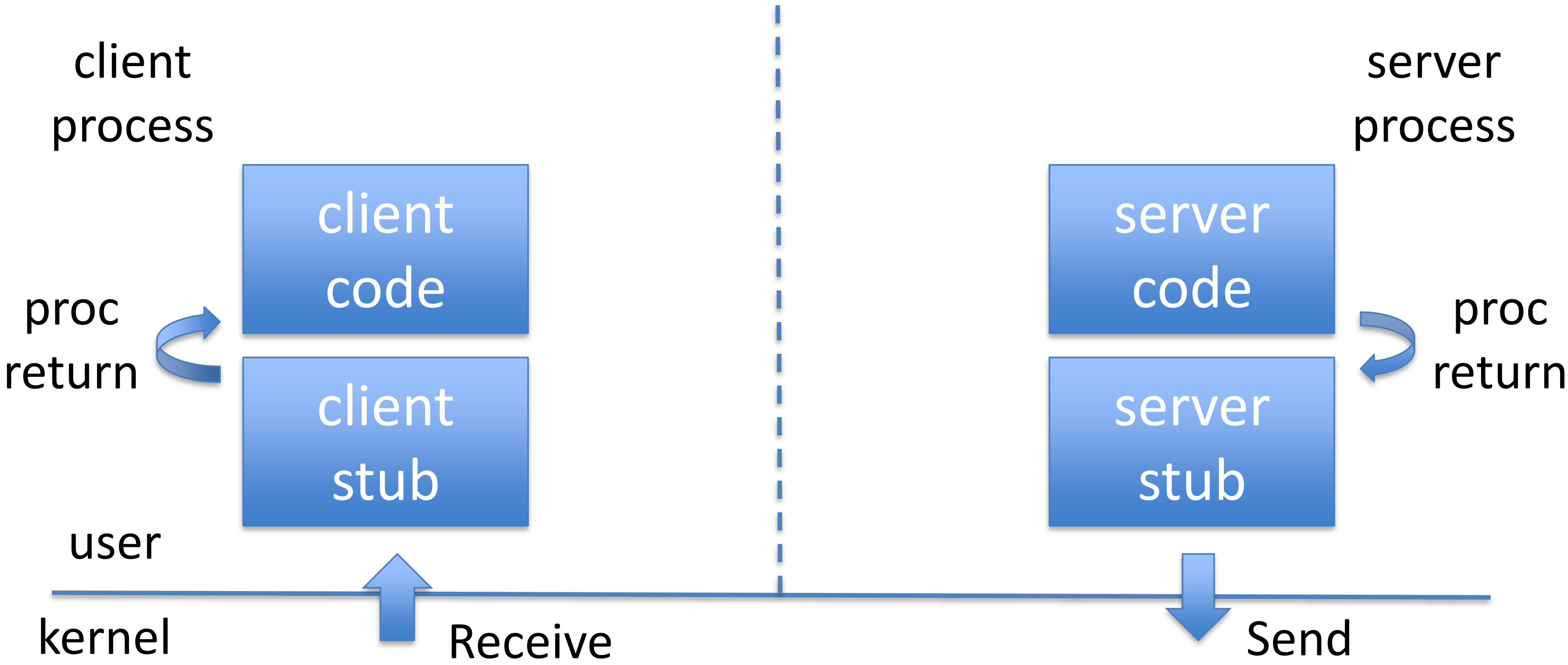
RPC Implementation: Return



RPC Implementation: Return



RPC Implementation: Return



An Example

- Timeserver
- Supports GetTime() and SetTime()

Interface

```
long GetTime()
boolean SetTime( long time )
```

Server Code

```
GetTime() {
    return( ReadHardwareClock() )
}
SetTime( time ) {
    WriteHardwareClock( time )
    return( 1 )
}
```

Client Code

```
main() {
    time = GetTime()
    SetTime( time + 100 )
}
```

Message Format

- We already saw:
  - Call message contains arguments
- Must also include which procedure is called

Message Format

Call Message

procno
arg0

Return Message

retval0

Client Stub

```
GetTime(){
    msg->procno = 1
    Send( msg )
    Receive( msg )
    return( msg->retval0 )
}
SetTime( long time ){
    msg->procno = 2
    msg->arg0 = time
    Send( msg )
    Receive( msg )
    return( msg->retval0 )
}
```

Server Stub

```
while( true ) do {
    Receive( msg )
    switch msg->procno {
        case 1: {
            time = GetTime()
            msg->retval0 = time
            Send( msg )
        }
        case 2: {
            ret = SetTime( msg->arg0 )
            msg->retval0 = ret
           Send( msg )
        }
    }
}
```

client code

```
main() {
   time = GetTime()
   SetTime( time + 100 )
}
```

client stub

```
GetTime() {
   msg->procno = 1
   Send( msg )
   Receive( msg )
   return( msg->retval0 )
}
SetTime( long time ) {
   msg->procno = 2
   msg->arg0 = time
   Send( msg )
   Receive( msg )
   return( msg->retval0 )
}
```

server code

```
GetTime() {
   return( ReadHardwareClock() )
}
SetTime( time ) {
   WriteHardwareClock( time )
   return( 1 )
}
```

server stub

```
while( true ) do {
   Receive( msg )
   switch msg->procno {
      case 1: { time = GetTime()
                msg->retval0 = time
                Send( msg ) }
      case 2: { ret = SetTime( msg->arg0 )
                msg->retval0 = ret
                Send( msg ) }
   }
}
```

client code

```
main() {
    time = GetTime()
    SetTime( time + 100 )
}
```

client stub

```
GetTime() {
    msg->procno = 1
    Send( msg )
    Receive( msg )
    return( msg->retval0 )
}
SetTime( long time ) {
    msg->procno = 2
    msg->arg0 = time
    Send( msg )
    Receive( msg )
    return( msg->retval0 )
}
```

server code

```
GetTime() {
    return( ReadHardwareClock() )
}
SetTime( time ) {
    WriteHardwareClock( time )
    return( 1 )
}
```

server stub

```
while( true ) do {
    Receive( msg )
    switch msg->procno {
        case 1: { time = GetTime()
                  msg->retval0 = time
                  Send( msg ) }
        case 2: { ret = SetTime( msg->arg0 )
                  msg->retval0 = ret
                  Send( msg ) }
    }
}
```

client code

```
main() {
   time = GetTime()
   SetTime( time + 100 )
}
```

client stub

```
GetTime() {
   msg->procno = 1
   Send( msg )
   Receive( msg )
   return( msg->retval0 )
}
SetTime( long time ) {
   msg->procno = 2
   msg->arg0 = time
   Send( msg )
   Receive( msg )
   return( msg->retval0 )
}
```

server code

```
GetTime() {
   return( ReadHardwareClock() )
}
SetTime( time ) {
   WriteHardwareClock( time )
   return( 1 )
}
```

server stub

```
while( true ) do {
   Receive( msg )
   switch msg->procno {
      case 1: { time = GetTime()
                msg->retval0 = time
                Send( msg ) }
      case 2: { ret = SetTime( msg->arg0 )
                msg->retval0 = ret
                Send( msg ) }
   }
}
```

client code

```
main() {
    time = GetTime()
    SetTime( time + 100 )
}
```

client stub

```
GetTime() {
    msg->procno = 1
    Send( msg )
    Receive( msg )
    return( msg->retval0 )
}
SetTime( long time ) {
    msg->procno = 2
    msg->arg0 = time
    Send( msg )
    Receive( msg )
    return( msg->retval0 )
}
```

server code

```
GetTime() {
    return( ReadHardwareClock() )
}
SetTime( time ) {
    WriteHardwareClock( time )
    return( 1 )
}
```

server stub

```
while( true ) do {
    Receive( msg )
    switch msg->procno {
        case 1: { time = GetTime()
                  msg->retval0 = time
                  Send( msg ) }
        case 2: { ret = SetTime( msg->arg0 )
                  msg->retval0 = ret
                  Send( msg ) }
    }
}
```

client code

```
main() {
   time = GetTime()
   SetTime( time + 100 )
}
```

client stub

```
GetTime() {
   msg->procno = 1
   Send( msg )
   Receive( msg )
   return( msg->retval0 )
}
SetTime( long time ) {
   msg->procno = 2
   msg->arg0 = time
   Send( msg )
   Receive( msg )
   return( msg->retval0 )
}
```

server code

```
GetTime() {
   return( ReadHardwareClock() )
}
SetTime( time ) {
   WriteHardwareClock( time )
   return( 1 )
}
```

server stub

```
while( true ) do {
   Receive( msg )
   switch msg->procno {
      case 1: { time = GetTime()
                msg->retval0 = time
                Send( msg ) }
      case 2: { ret = SetTime( msg->arg0 )
                msg->retval0 = ret
                Send( msg ) }
   }
}
```

client code

```
main() {
    time = GetTime()
    SetTime( time + 100 )
}
```

client stub

```
GetTime() {
    msg->procno = 1
    Send( msg )
    Receive( msg )
    return( msg->retval0 )
}
SetTime( long time ) {
    msg->procno = 2
    msg->arg0 = time
    Send( msg )
    Receive( msg )
    return( msg->retval0 )
}
```

server code

```
GetTime() {
    return( ReadHardwareClock() )
}
SetTime( time ) {
    WriteHardwareClock( time )
    return( 1 )
}
```

server stub

```
while( true ) do {
    Receive( msg )
    switch msg->procno {
        case 1: { time = GetTime()
                  msg->retval0 = time
                  Send( msg ) }
        case 2: { ret = SetTime( msg->arg0 )
                  msg->retval0 = ret
                  Send( msg ) }
    }
}
```

client code

```
main() {
   time = GetTime()
   SetTime( time + 100 )
}
```

client stub

```
GetTime() {
   msg->procno = 1
   Send( msg )
   Receive( msg )
   return( msg->retval0 )
}
SetTime( long time ) {
   msg->procno = 2
   msg->arg0 = time
   Send( msg )
   Receive( msg )
   return( msg->retval0 )
}
```

server code

```
GetTime() {
   return( ReadHardwareClock() )
}
SetTime( time ) {
   WriteHardwareClock( time )
   return( 1 )
}
```

server stub

```
while( true ) do {
   Receive( msg )
   switch msg->procno {
      case 1: { time = GetTime()
                  msg->retval0 = time
                  Send( msg ) }
      case 2: { ret = SetTime( msg->arg0 )
                  msg->retval0 = ret
                  Send( msg ) }
   }
}
```

client code

```
main() {
   time = GetTime()
   SetTime( time + 100 )
}
```

client stub

```
GetTime() {
   msg->procno = 1
   Send( msg )
   Receive( msg )
   return( msg->retval0 )
}
SetTime( long time ) {
   msg->procno = 2
   msg->arg0 = time
   Send( msg )
   Receive( msg )
   return( msg->retval0 )
}
```

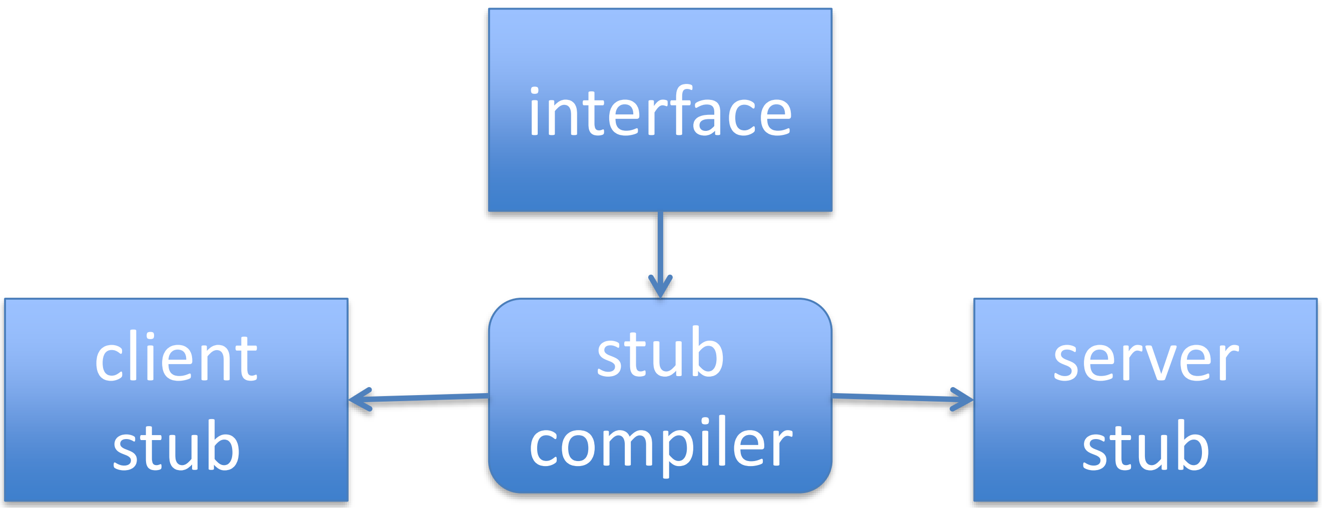
server code

```
GetTime() {
   return( ReadHardwareClock() )
}
SetTime( time ) {
   WriteHardwareClock( time )
   return( 1 )
}
```

server stub

```
while( true ) do {
   Receive( msg )
   switch msg->procno {
      case 1: { time = GetTime()
                msg->retval0 = time
                Send( msg ) }
      case 2: { ret = SetTime( msg->arg0 )
                msg->retval0 = ret
                Send( msg ) }
   }
}
```

Note: Stubs Generated Automatically



Week 3 – Part 2
Application Multithreading and Synchronization

Pamela Delgado

March 6, 2019

(slides Willy Zwaenepoel)

Key Concepts

- Multithreading vs. multiprocessing
- Synchronization
- Pthreads examples

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
    }
}
```

Still a Performance Problem

- Disk access is expensive

Multiprocess Web Server with Cache

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

WorkerProcess[?] {
    forever {
        wait for message( request )
        if( requested file is not in cache ) {
            read file from disk
            put file in cache
        }
        send response
    }
}
```

Now there is a different problem (1)

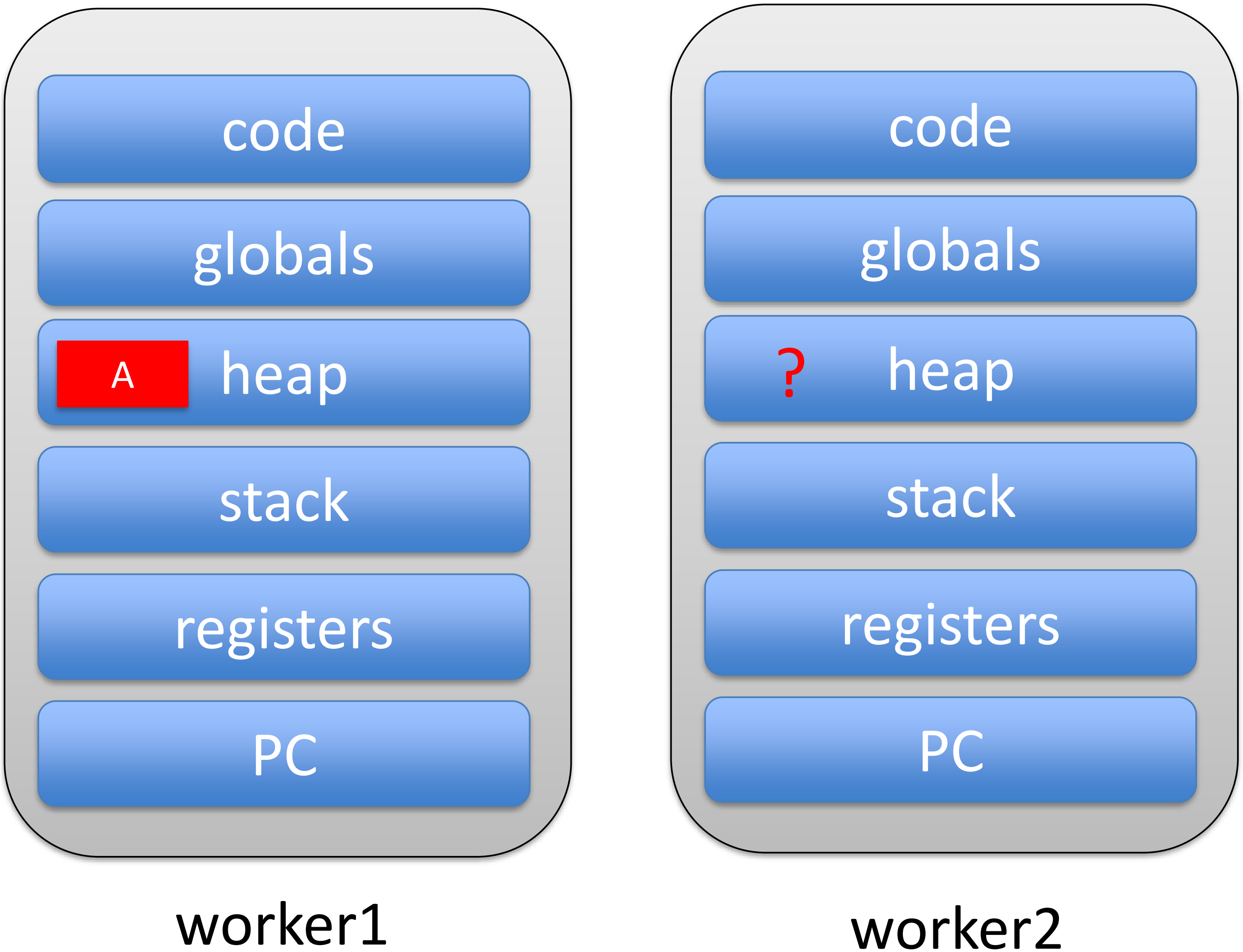
- Incoming request for file A
- Listener sends request to worker1
- Worker1 reads file A from disk
- Worker1 puts file A in its memory

Now there is a different problem (2)

Now there is a different problem (3)

- Incoming request for file A
- Listener sends request to worker1
- Worker1 reads file A from disk
- Worker1 puts file A in its memory
- Another incoming request for file A
- Listener sends request to worker2

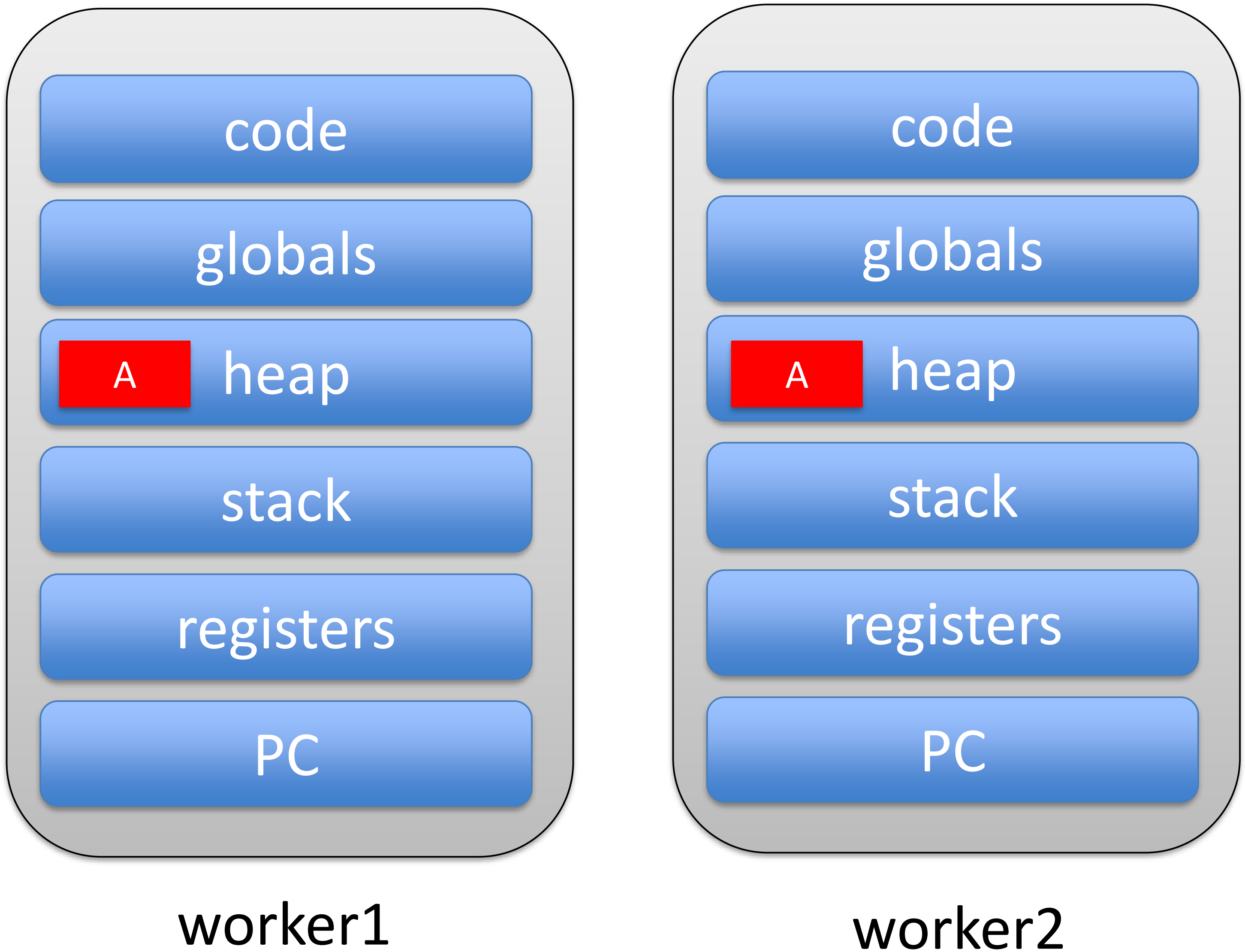
Now there is a different problem (4)



Now there is a different problem (3)

- Incoming request for file A
- Listener sends request to worker1
- Worker1 reads file A from disk
- Worker1 puts file A in its memory
- Another incoming request for file A
- Listener sends request to worker2
- Worker2 reads file A from disk
- Worker2 puts file A in its memory

Now there is a different problem (4)



What is the Problem?

- Worker1 and Worker2 do not share memory
- Effectiveness of cache is much reduced

What is the Solution?

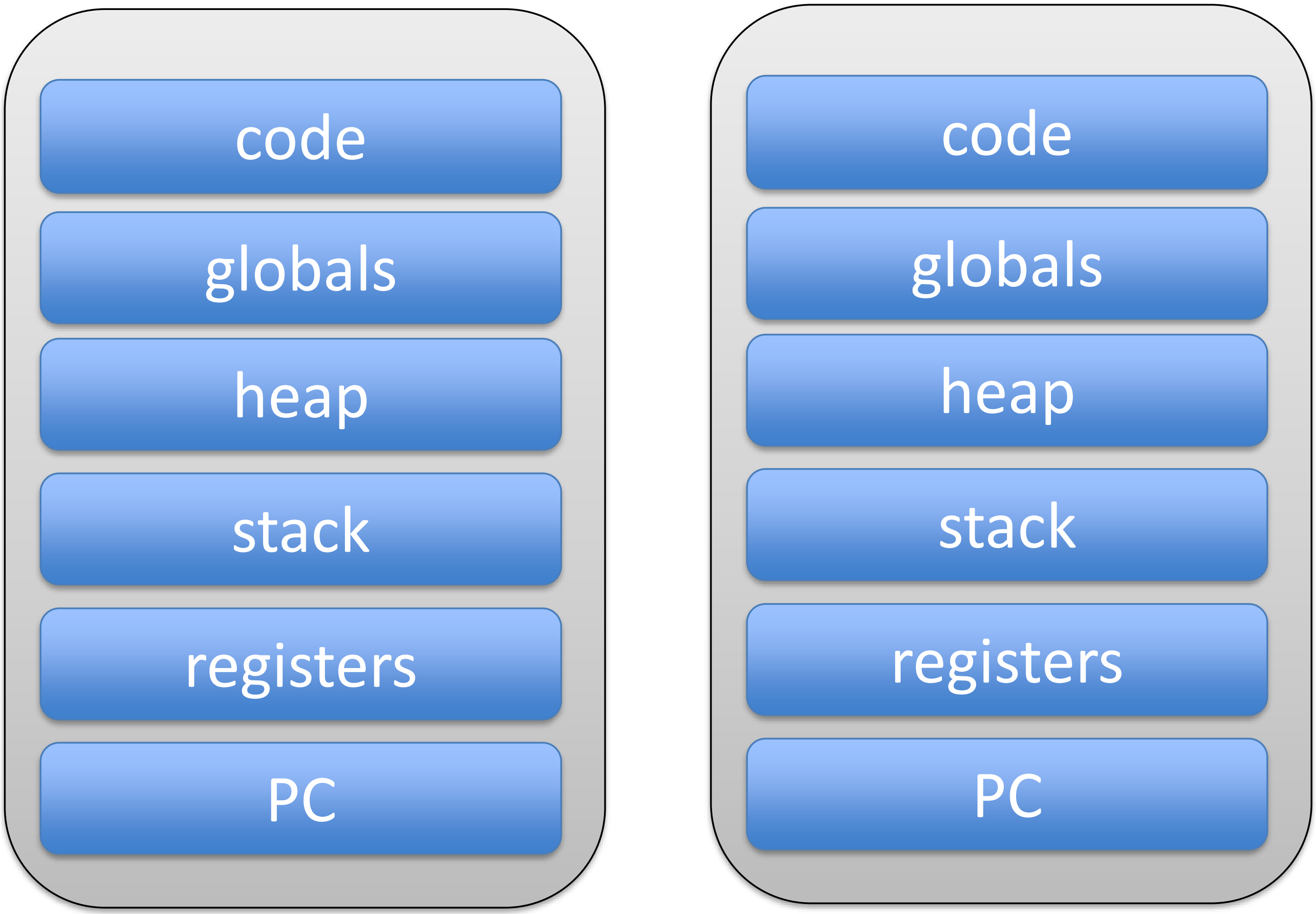
- Make Worker1 and Worker2 share memory
- This is multithreading

Multithreading

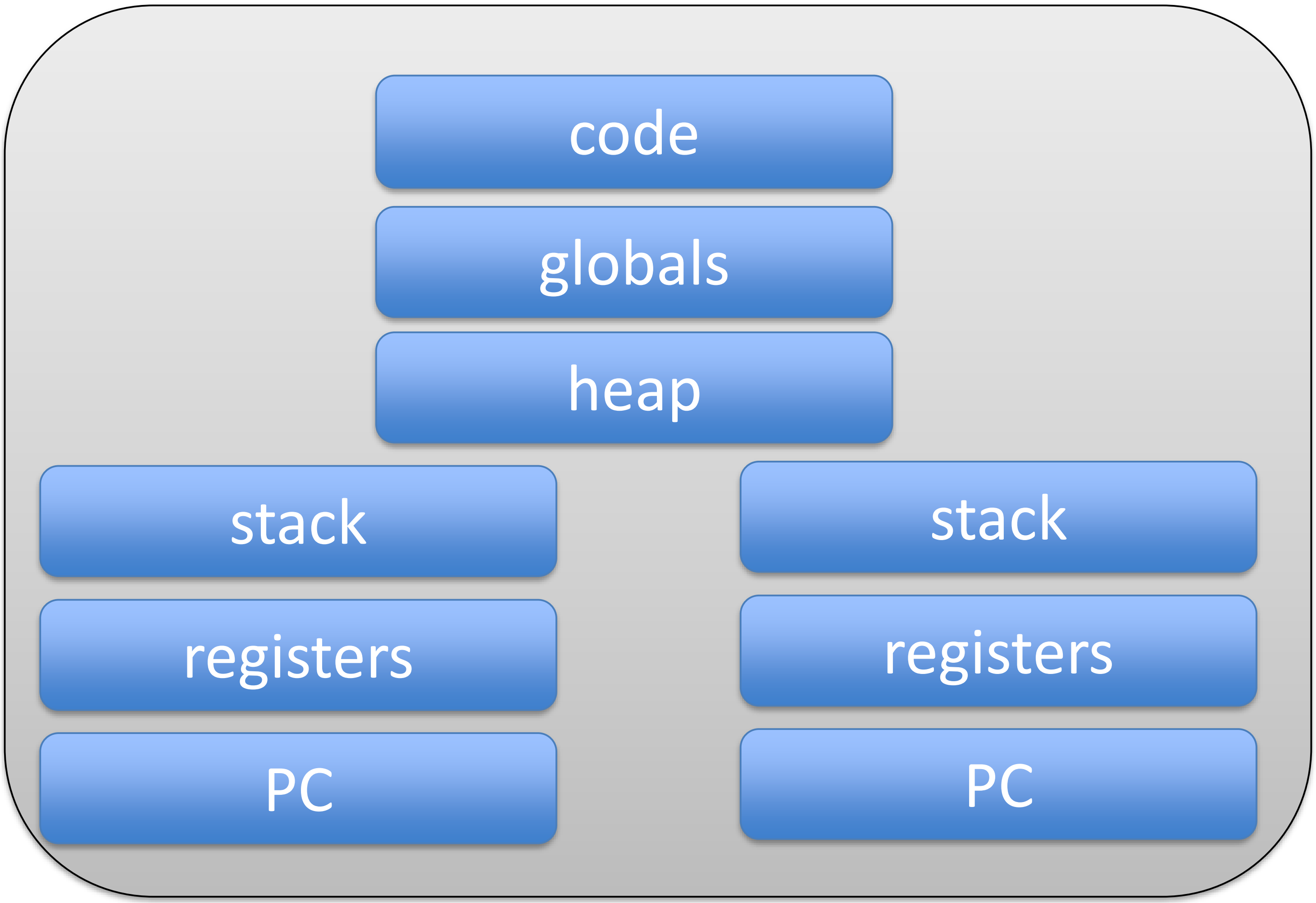
- A thread is just like a process
- But it does NOT have its own heap and globals

- Has its own PC, registers, stack
- Shares heap and globals with other threads in the same process

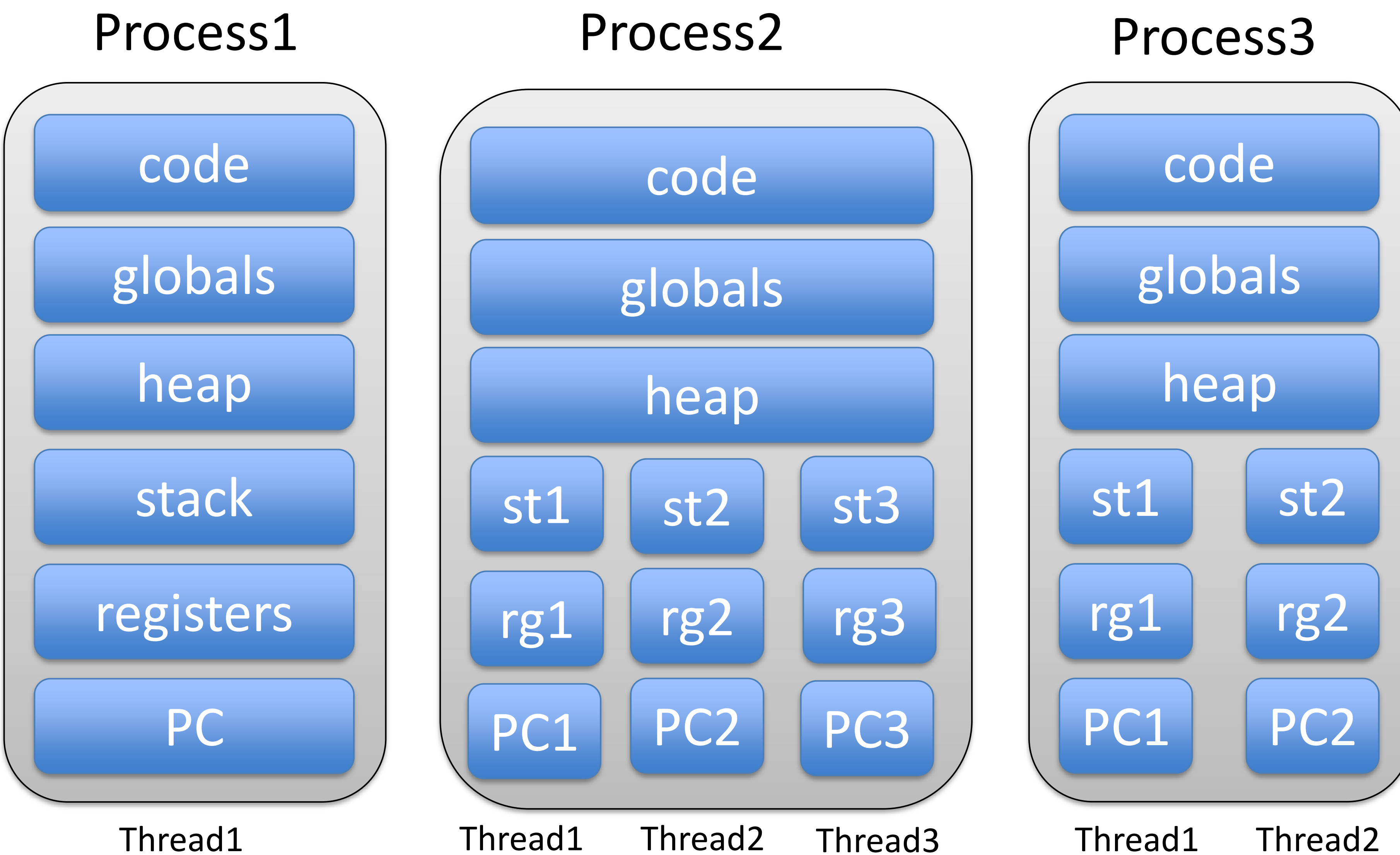
Two Processes



Two Threads in a Process



More Complex Example



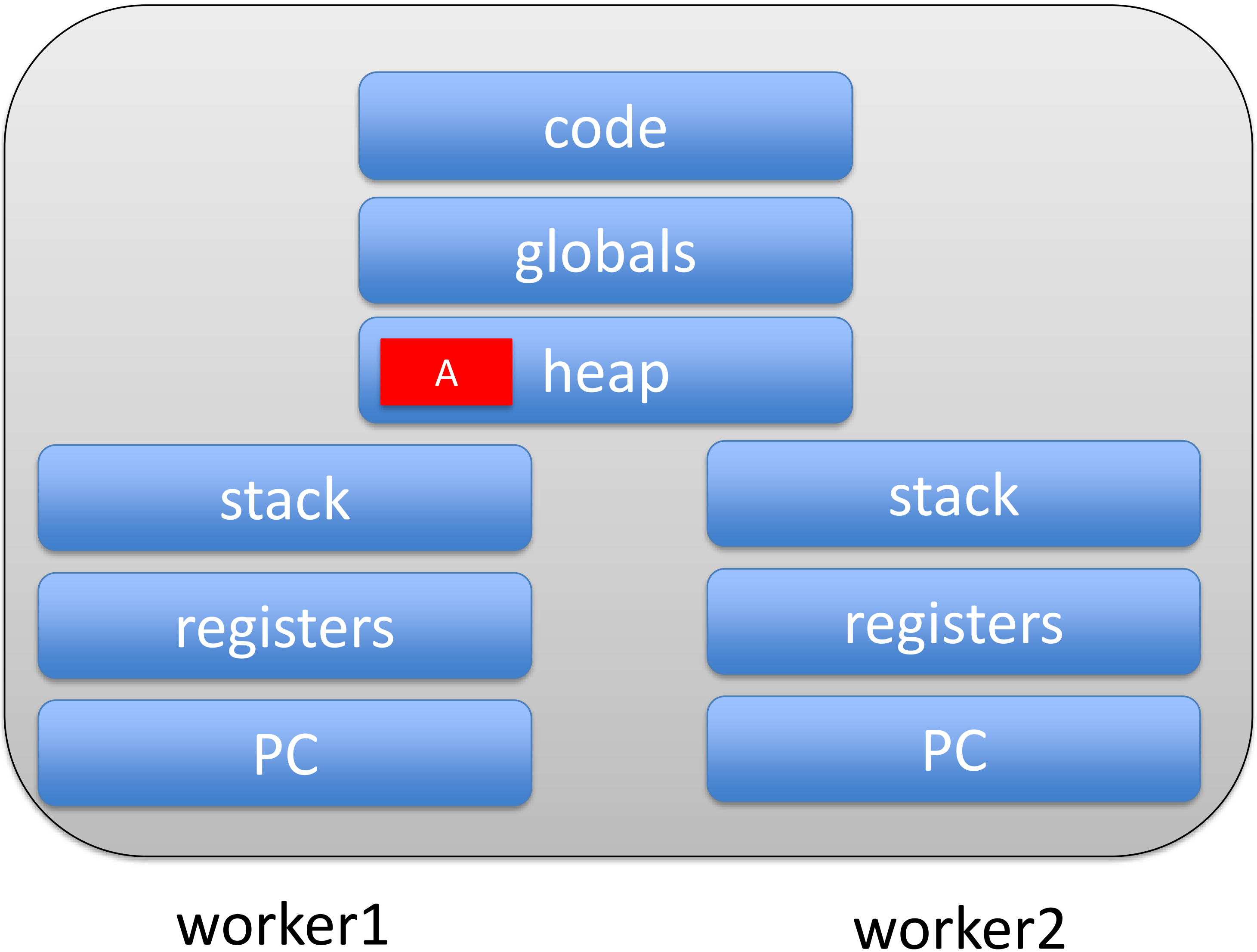
Multithreaded Web Server with Cache

```
ListenerThread {
    for ( i=0; i<MAXTHREADS; i++ )
        thread[i] = CreateThread()
    forever {
        wait for incoming request
        send( request, thread[?] )
    }
}

WorkerThread[?] {
    forever {
        wait for message( request )
        if( requested file is not in cache ) {
            read file from disk
            put file in cache
        }
    send response
}
```

Problem Solved (1)

- Incoming request for file A
- Listener sends request to worker1
- Worker1 reads file A from disk
- Worker1 puts file A in memory

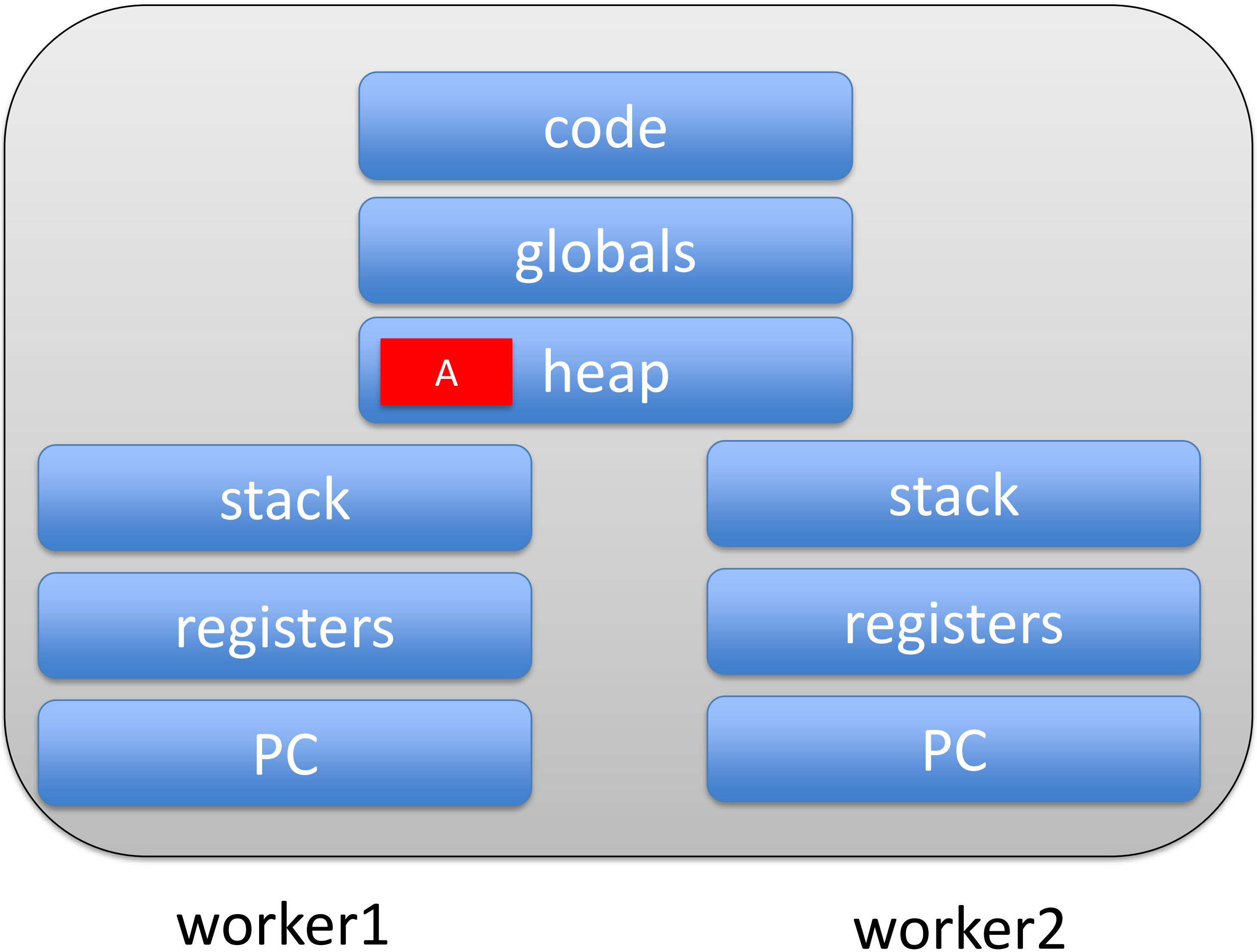
Problem Solved (2)



Problem Solved (3)

- Incoming request for file A
- Listener sends request to worker1
- Worker1 reads file A from disk
- Worker1 puts file A in memory
- Another incoming request for file A
- Listener sends request to worker2
- Worker2 finds file A in cache
- Responds with file from cache

Problem Solved (4)



In General

- Processes provide separation
  - In particular, memory separation (no shared data)
  - Suitable for coarse-grain interaction
- Threads do not
  - In particular, share memory (shared data)
  - Suitable for tighter integration

Shared Data

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

Data Race

- Unexpected/unwanted access to shared data

Data Race Example

- Unexpected/unwanted access to shared data

```
Thread 1:
    i = my_value; … ; array[i] = …

Thread 2:
    i = other_value

Interleaving:
    i = my_value; … ; i = other_value; … ; array[i] =
    …
```

Data Race

- Unexpected/unwanted access to shared data
- Result of *interleaving* of thread executions
- Program must be correct for all interleavings

Application Multithreading

- You studied this already in Concurrency
- Repeating basic principles
- Show structured approach
- Use Pthreads

- Note:
  - Showing essentials
  - Not necessarily working Pthreads code

Basic Approach to Multithreading

- Divide “work” among multiple threads
- Which data is shared?
  - Globals and heap
  - Not locals
  - Not read-only
- Where is shared data accessed?
- Put shared data access in critical section
  - Only one process at a time can access it

Why this (mostly) works

- Trouble with multithreaded execution:
  - Data races
  - Data changed by another thread
- Critical section:
  - No other thread can change data
- So you are (mostly) ok

Data Race Example

- Unexpected/unwanted access to shared data

```
Thread 1:
    critical section { i = my_value; … ; array[i] = …
    }

Thread 2:
    critical section { i = other_value }

Interleaving:
    i = my_value; … ; i = other_value; … ; array[i] =
    …
```

Pthreads: Thread Creation and Destruction

- Pthread_create(&threadid, threadcode, arg)
- Pthread_exit(status)
- Pthread_join(threadid, &status)

Pthreads: Thread Creation and Destruction

- Pthread_create(&threadid, threadcode, arg)
  - Create thread
  - Return threadid
  - Run threadcode
  - With argument arg
- Pthread_exit(status)
- Pthread_join(threadid, &status)

Pthreads: Thread Creation and Destruction

- Pthread_create(&threadid, threadcode, arg)
- Pthread_exit(status)
  - Terminate thread
  - Optionally return status
- Pthread_join(threadid, &status)

Pthreads: Thread Creation and Destruction

- Pthread_create(&threadid, threadcode, arg)
- Pthread_exit(status)
- Pthread_join(threadid, &status)
  - Wait for thread threadid to exit
  - Receive status, if any

Example: Fork-Join Parallelism

- Main thread
  - Creates number of worker threads
  - Waits for them to finish
- Worker threads
  - Do work more or less independently
  - Exit

Simple Pthreads Example

```
#include <pthreads.h>
#define NUM_THREADS     5

int main(void) {
   pthread_t threads[NUM_THREADS];
   int thread_args[NUM_THREADS];
   int rc, i;

   /* create all threads */
   for (i=0; i<NUM_THREADS; ++i) {
      thread_args[i] = i;
      pthread_create(&threads[i], ThreadCode, (void *) &thread_args[i]);
   }

   /* wait for all threads to complete */
   for (i=0; i<NUM_THREADS; ++i) {
      pthread_join(threads[i], NULL);
   }
   exit(0);
}
```

Simple Pthreads Example

```
#include <pthreads.h>
#define NUM_THREADS     5

int main(void) {
   pthread_t threads[NUM_THREADS];
   int thread_args[NUM_THREADS];
   int rc, i;

   /* create all threads */
   for (i=0; i<NUM_THREADS; ++i) {
      thread_args[i] = i;
      pthread_create(&threads[i], ThreadCode, (void *) &thread_args[i]);
   }

   /* wait for all threads to complete */
   for (i=0; i<NUM_THREADS; ++i) {
      pthread_join(threads[i], NULL);
   }
   exit(0);
}
```

Simple Pthreads Example

```
#include <pthreads.h>
#define NUM_THREADS     5

int main(void) {
   pthread_t threads[NUM_THREADS];
   int thread_args[NUM_THREADS];
   int rc, i;

   /* create all threads */
   for (i=0; i<NUM_THREADS; ++i) {
      thread_args[i] = i;
      pthread_create(&threads[i], ThreadCode, (void *) &thread_args[i]);
   }

   /* wait for all threads to complete */
   for (i=0; i<NUM_THREADS; ++i) {
      pthread_join(threads[i], NULL);
   }
   exit(0);
}
```

Simple Pthreads Example

```
void *ThreadCode(void *argument) {
   int tid;
   tid = *((int *) argument);
   printf("Hello World! It's me, thread %d!\n", tid);
   /* optionally: insert more useful stuff here */
   return NULL;
}
```

Pthreads: Locks

- Pthread_mutex_lock(mutex)
- Pthread_mutex_unlock(mutex)

Pthreads: Locks

- Pthread_mutex_lock(mutex)
  - If mutex is held, block
  - If mutex is not held
    - Acquire mutex
    - Proceed
- Pthread_mutex_unlock(mutex)

Pthreads: Locks

- Pthread_mutex_lock(mutex)
- Pthread_mutex_unlock(mutex)
  - Release mutex

Example: Single-Threaded Code

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

Basic Approach to Multithreading

- Divide “work” among multiple threads
- Which data is shared?
  - Globals and heap
  - Not locals
  - Not read-only
- Where is shared data accessed?
- Define one mutex
- Put lock/unlock around each shared access

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: Shared Data

- Shared data
  - sum
  - prod
- Shared read-only data
  - a[], b[]
- Local data
  - i (loop index), c
- mutex on access to sum and prod

Example: Synchronization

```
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

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)
    }
}
```

Caveat

- When using fine-grain locking
- Or, when using multiple locks

- Be careful with deadlocks

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)
}
```

Another Example: Multithreaded Web Server

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

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

Shared Data?

- There is none!

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 a queue
- Put lock/unlock around it

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
        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 (at all)

- Not fork-join parallelism
- You need to tell worker(s) there is something for them to do (i.e., in the queue)
- Sometimes called task parallelism

Pthreads: Condition Variables

- Pthread_cond_wait(cond, mutex)
- Pthread_cond_signal(cond, mutex)
- Pthread_cond_broadcast(cond, mutex)

Pthreads: Condition Variables

- Pthread_cond_wait(cond, mutex)
- Pthread_cond_signal(cond, mutex)
- Pthread_cond_broadcast(cond, mutex)

- Must hold mutex when calling any of these!

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)

Pthreads: Condition Variables

- Pthread_cond_wait(cond, mutex)
- Pthread_cond_signal(cond, mutex)
  - Signal one thread waiting on cond
  - Signaled thread re-acquires mutex
    - At some later time, not necessarily immediately
- Pthread_cond_broadcast(cond, mutex)

Pthreads: Condition Variables

- Pthread_cond_wait(cond, mutex)
- Pthread_cond_signal(cond, mutex)
- Pthread_cond_broadcast(cond, mutex)
  - Signal all threads waiting on cond

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
        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 )
    }
}
```

Not correct

- Signals have no memory
- Signal when no one is waiting is lost

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

Still not quite correct

- Q is empty, thread W1 waits
- Thread L puts something in Q
  - Sets avail to 1
  - Signals
  - W1 is unblocked
- Thread W2 runs and takes something out of Q
  - Sets avail to 0
- Now W1 runs
  - It must check the value of avail

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 )
        while( avail <= 0 ) Pthread_cond_wait( notempty, queuelock )
        take request out of queue
        avail--
        Pthread_mutex_unlock( queuelock )
        read file from disk
        Send( reply )
    }
}
```

Summary

- Why shared data and multithreading?
- Application multithreading
  - Division of work
  - Synchronization of shared data
  - Fine-grain locking
  - Privatization