

[537] Distributed Systems

Chapters 42
Tyler Harter
11/19/14

File-System Case Studies

Local

- **FFS**: Fast File System
- **LFS**: Log-Structured File System

Network

- **NFS**: Network File System
 - **AFS**: Andrew File System
-

File-System Case Studies

Local

- **FFS**: Fast File System
- **LFS**: Log-Structured File System

Network

- **Intro**: communication basics [today]
 - **NFS**: Network File System
 - **AFS**: Andrew File System
-

Review

Atomicity

Say we want to do several things.

Atomicity means we **don't get interrupted** when partially done (or at least that we can make it appear that way to the user).

Concurrency: we're worried about **other threads**

Persistence: we're worried about **crashes**

Atomic Update

Say we want to update a file `foo.txt`. If we crash, we want one of the following:

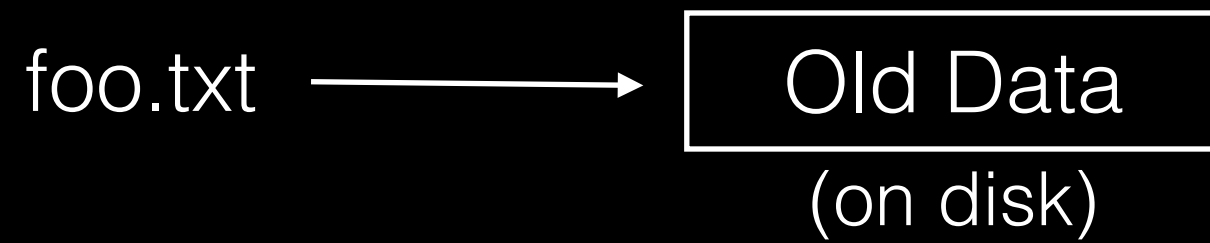
- all old data
- all new data

Strategy: write new data to `foo.tmp`, and only after that's complete, replace `foo.txt` by switching names.

Bad Protocol

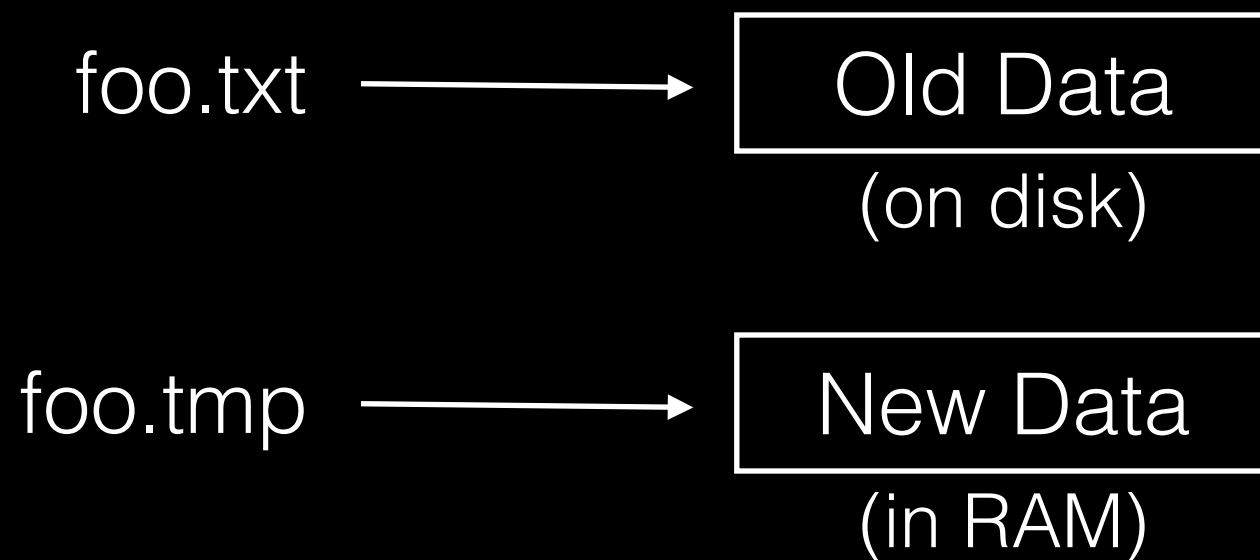
copy `foo.txt` to `foo.tmp` (with changes)
rename `foo.tmp` to `foo.txt`

Bad Protocol



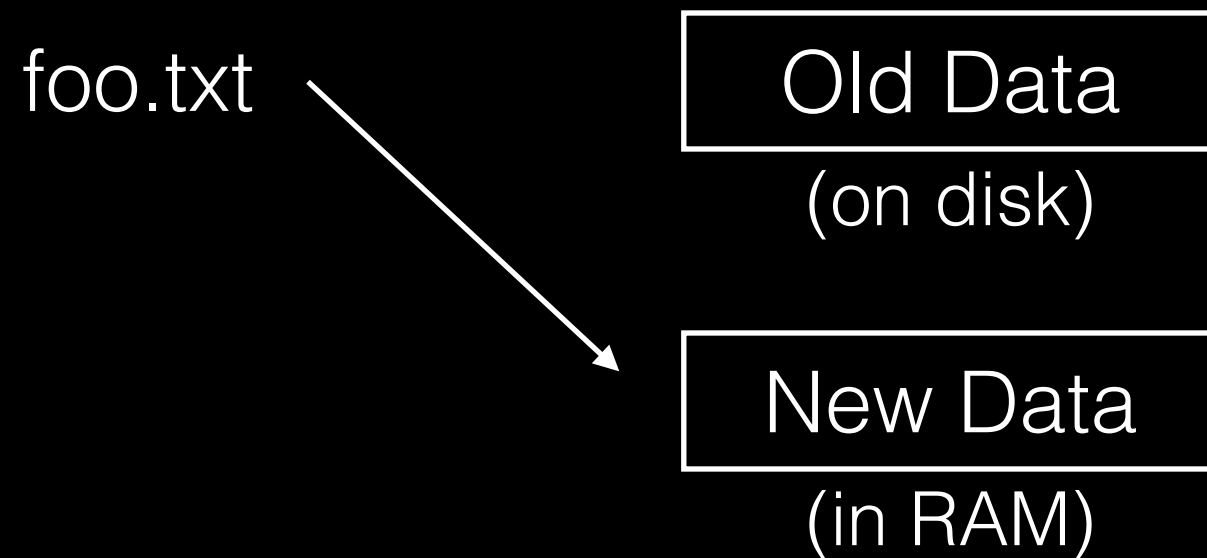
Bad Protocol

copy `foo.txt` to `foo.tmp` (with changes)



Bad Protocol

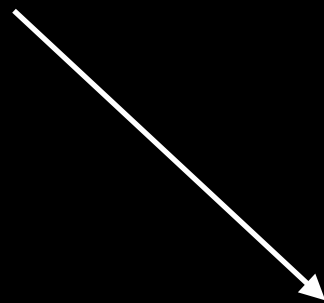
copy `foo.txt` to `foo.tmp` (with changes)
rename `foo.tmp` to `foo.txt`



Bad Protocol

copy `foo.txt` to `foo.tmp` (with changes)
rename `foo.tmp` to `foo.txt`

`foo.txt`
(on disk)



New Data
(in RAM)

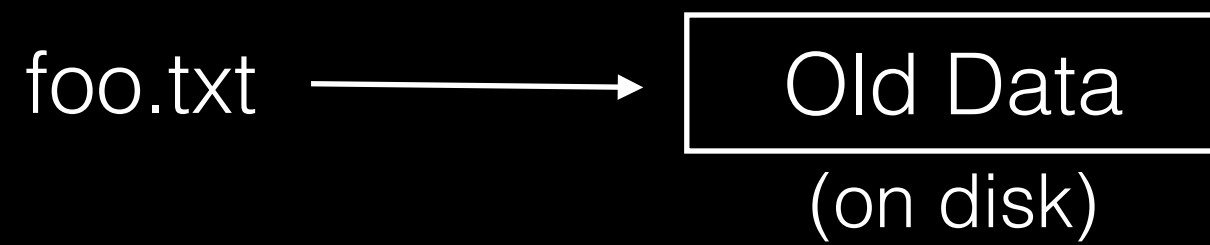
Good Protocol

copy `foo.txt` to `foo.tmp` (with changes)

`fsync foo.tmp`

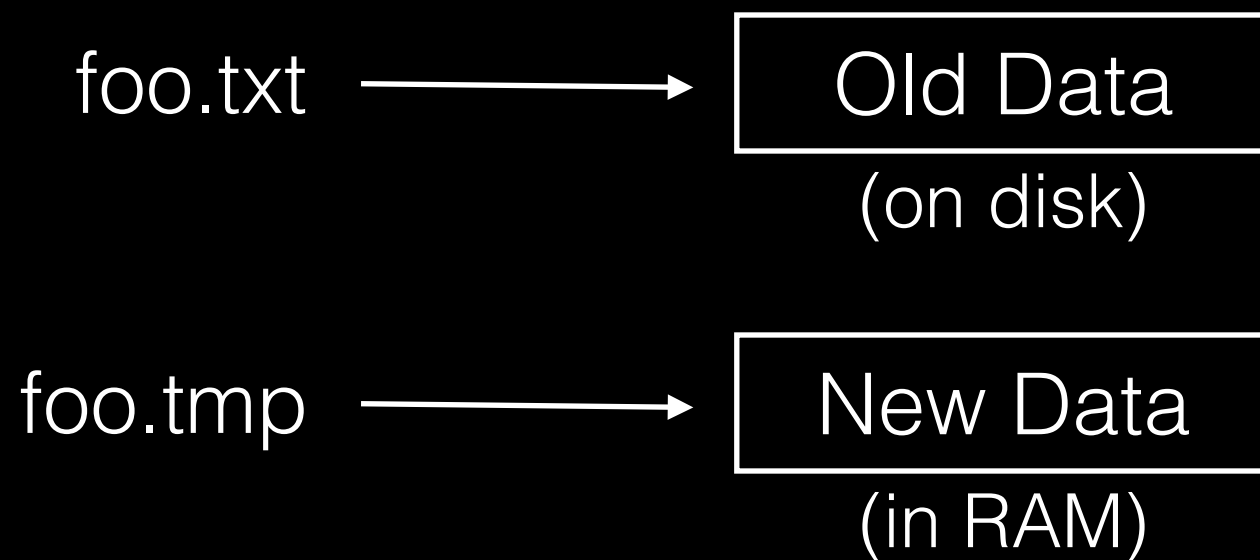
`rename foo.tmp to foo.txt`

Good Protocol



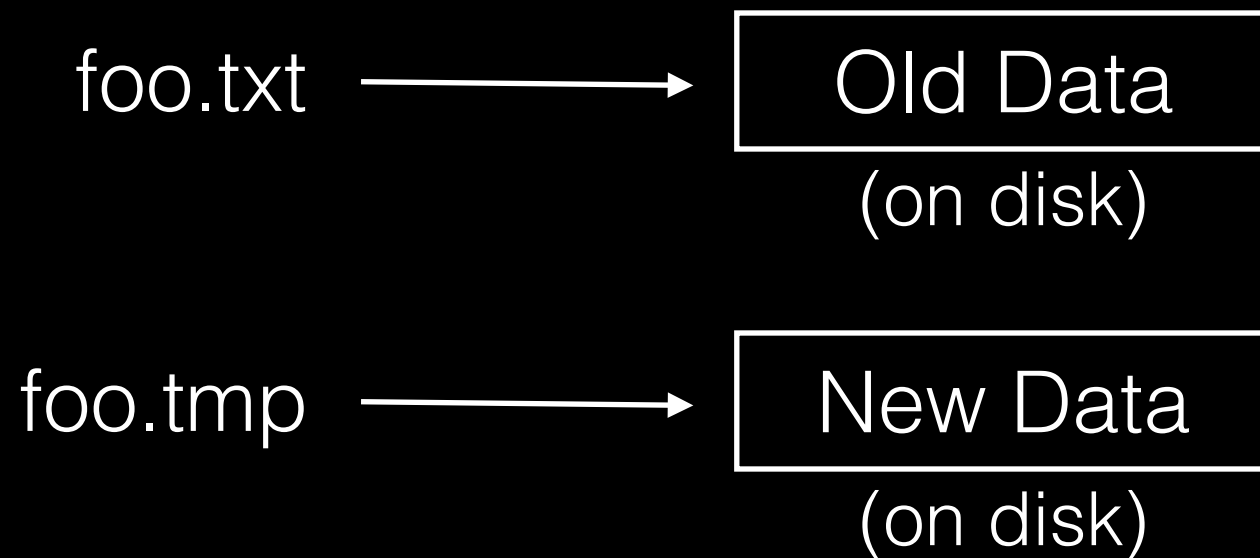
Good Protocol

copy `foo.txt` to `foo.tmp` (with changes)



Good Protocol

copy `foo.txt` to `foo.tmp` (with changes)
fsync `foo.tmp`

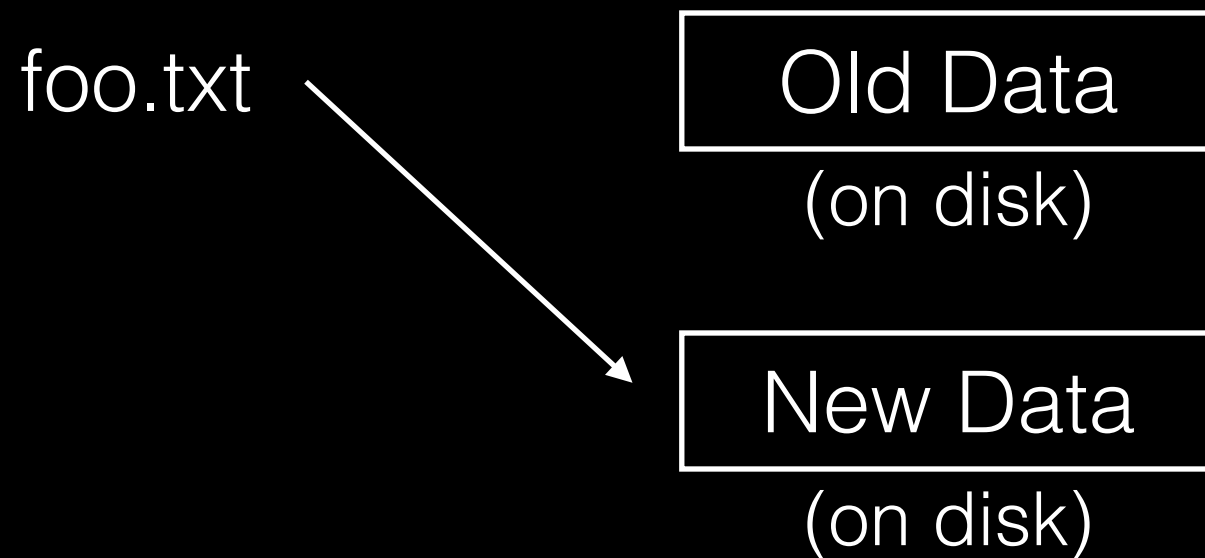


Good Protocol

copy `foo.txt` to `foo.tmp` (with changes)

`fsync foo.tmp`

rename `foo.tmp` to `foo.txt`



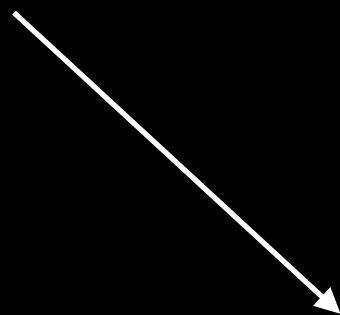
Good Protocol

copy `foo.txt` to `foo.tmp` (with changes)

`fsync foo.tmp`

rename `foo.tmp` to `foo.txt`

`foo.txt`
(on disk)



New Data
(on disk)

Local FS Comparison

FFS+Journal:

- must write data twice (writes expensive)
- can put data exactly where we like (reads cheaper)

LFS:

- all writes sequential (writes cheaper)
 - reads may be very random (reads expensive)
-

Local FS Comparison

In what ways is FFS more complex?

In what ways is LFS more complex?

Compare group descriptor to segment summary.

LFS: why don't we need to update root inode upon updating any file?

Distributed Systems

OSTEP Definition

Def: more than 1 machine

Examples:

- client/server: web server and web client
- cluster: page rank computation

Other courses:

CS 640: Networking

CS 739: Distributed Systems

Why Go Distributed?

More compute power

More storage capacity

Fault tolerance

Data sharing

New Challenges

System failure: need to worry about partial failure.

Communication failure: links unreliable

Communication

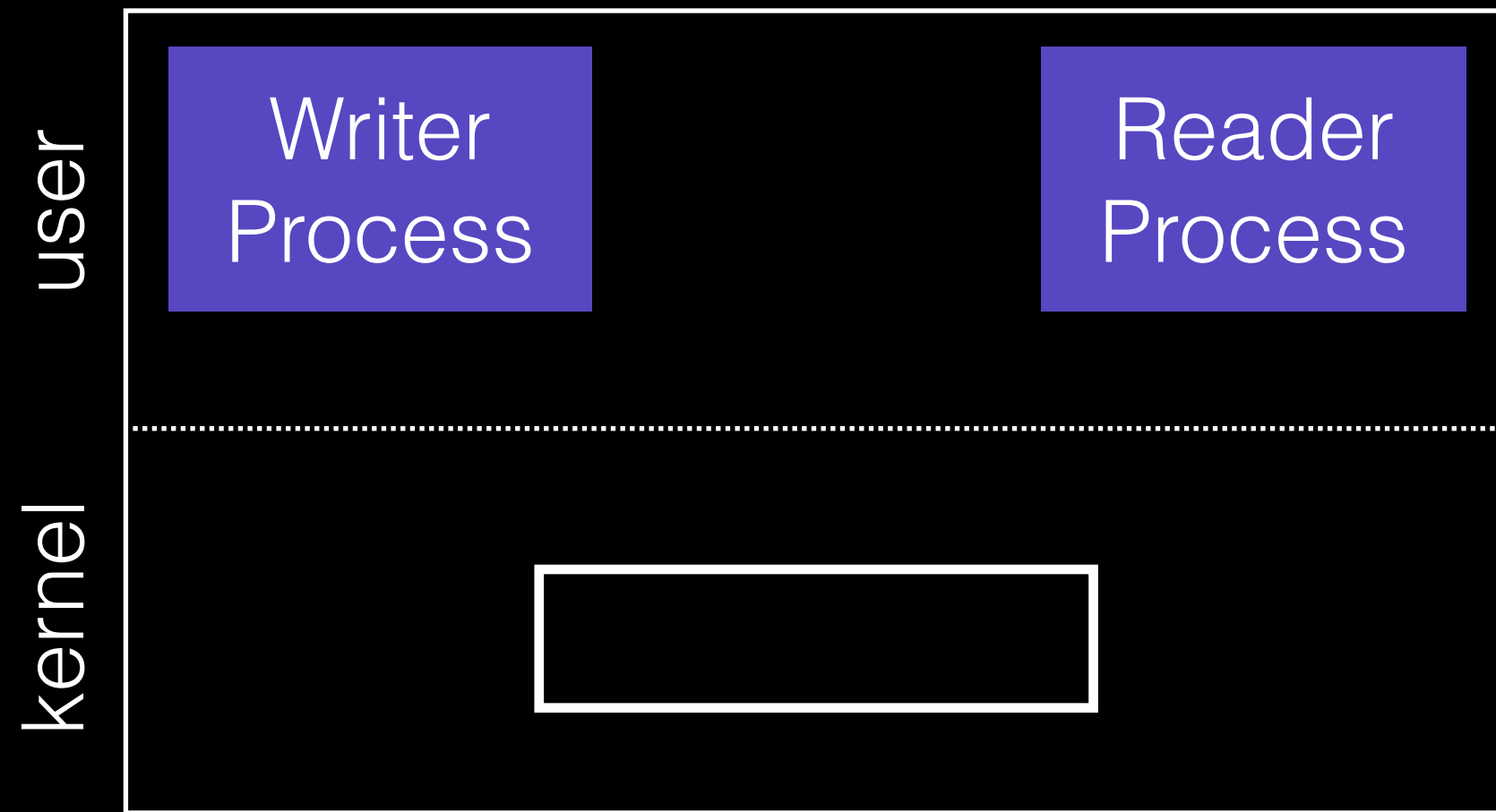
All communication is inherently unreliable.

Need to worry about:

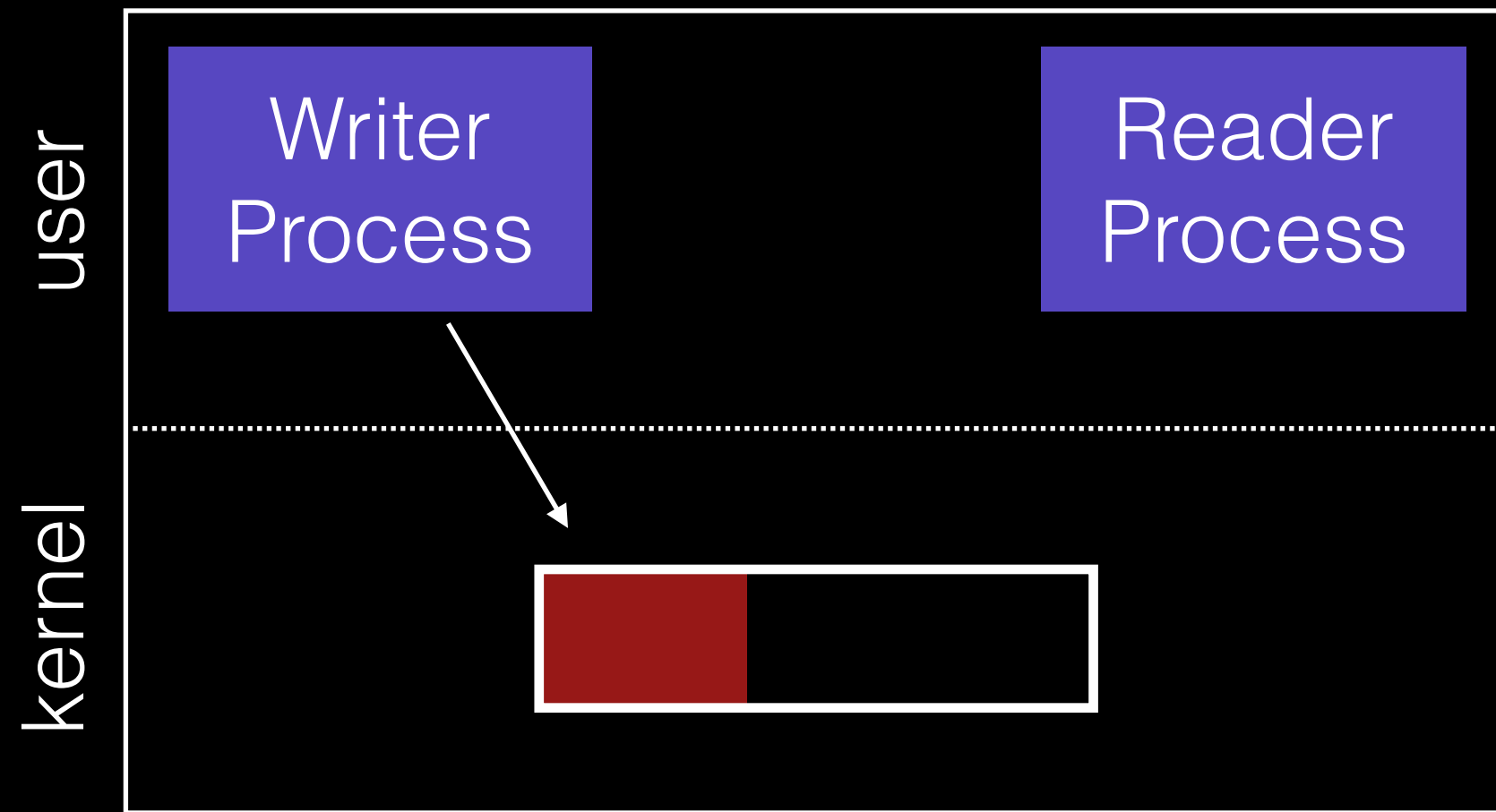
- bit errors
- packet loss
- node/link failure

Why are network sockets
less reliable than pipes?

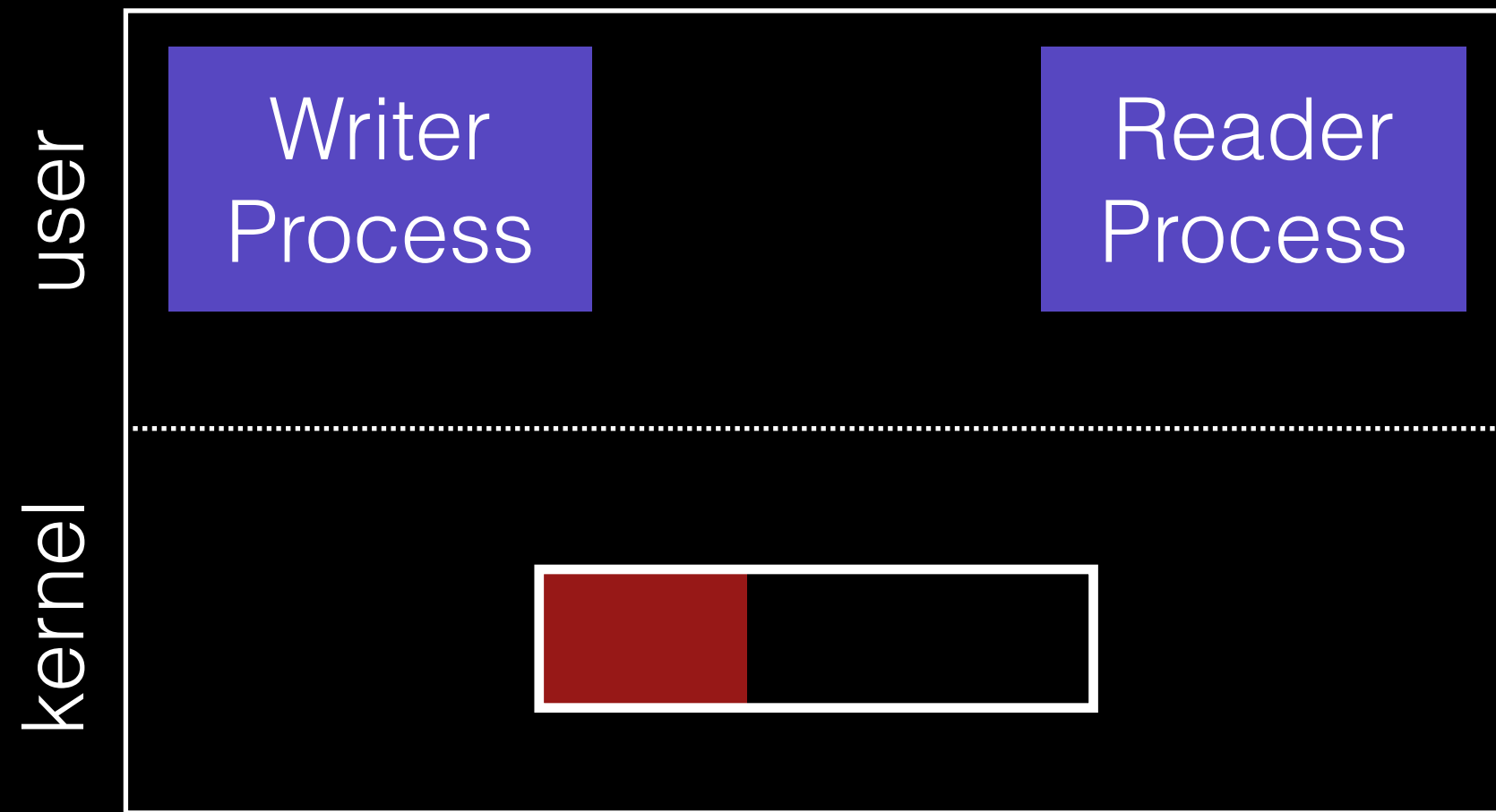
Pipe



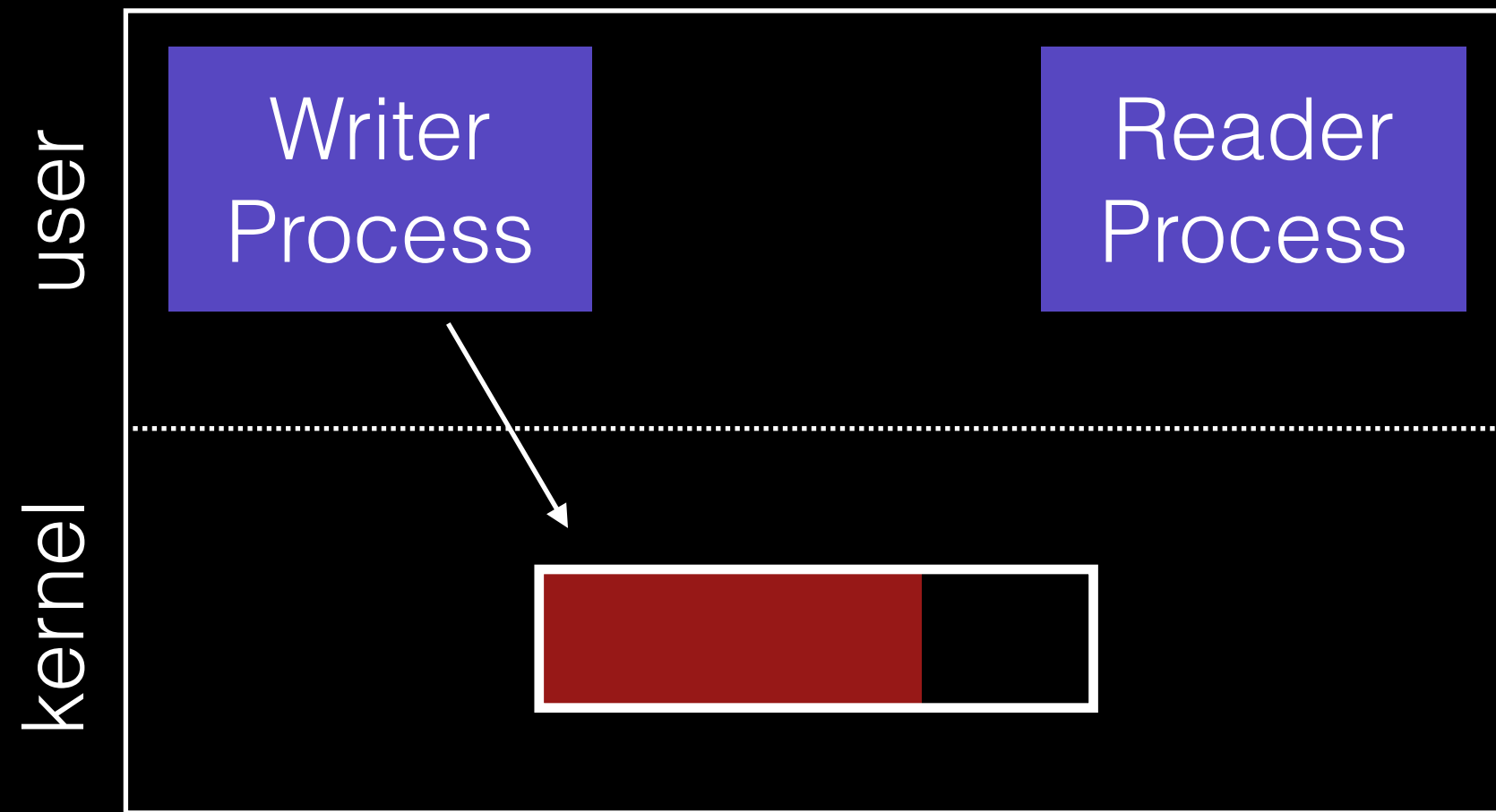
Pipe



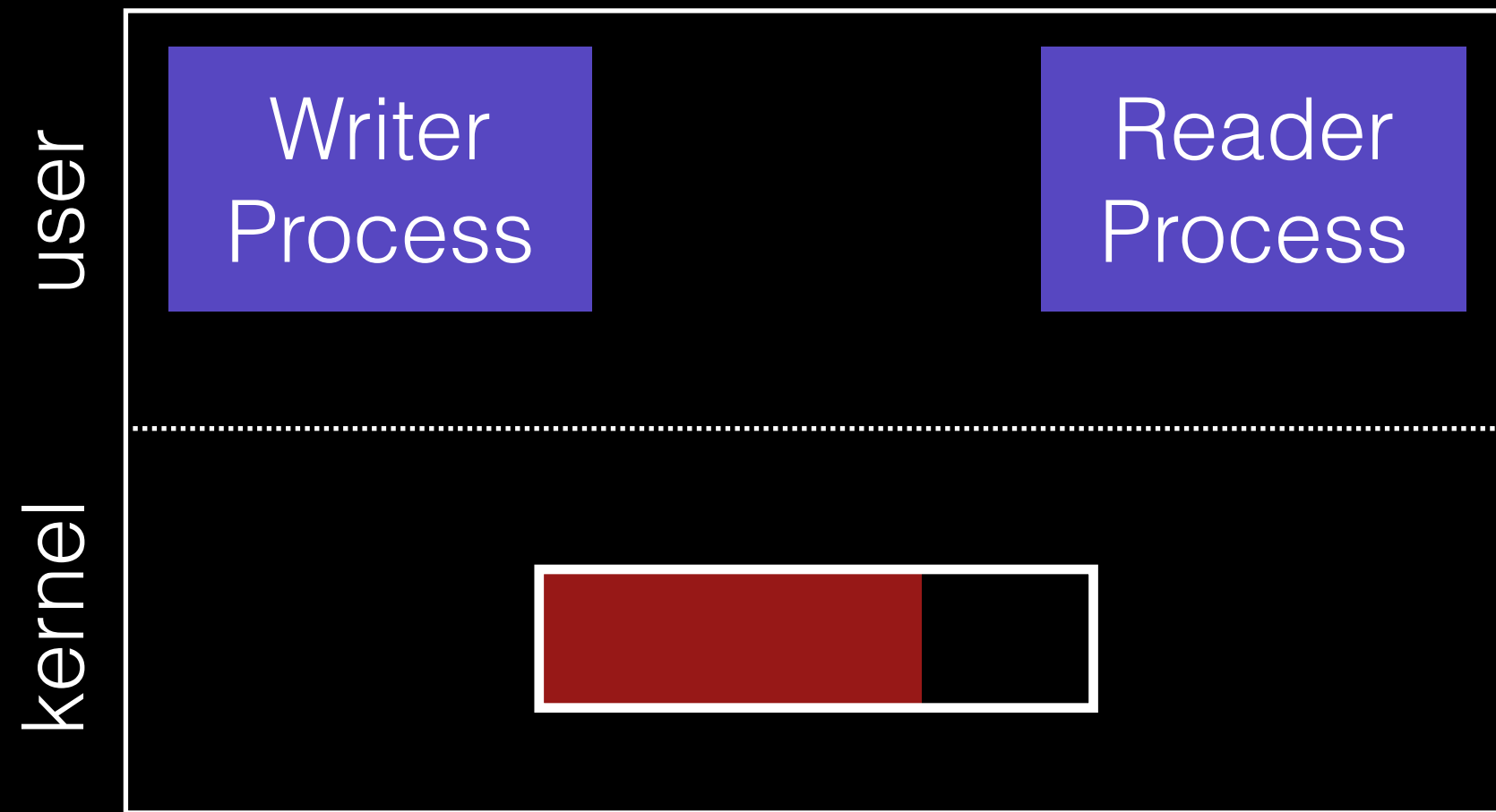
Pipe



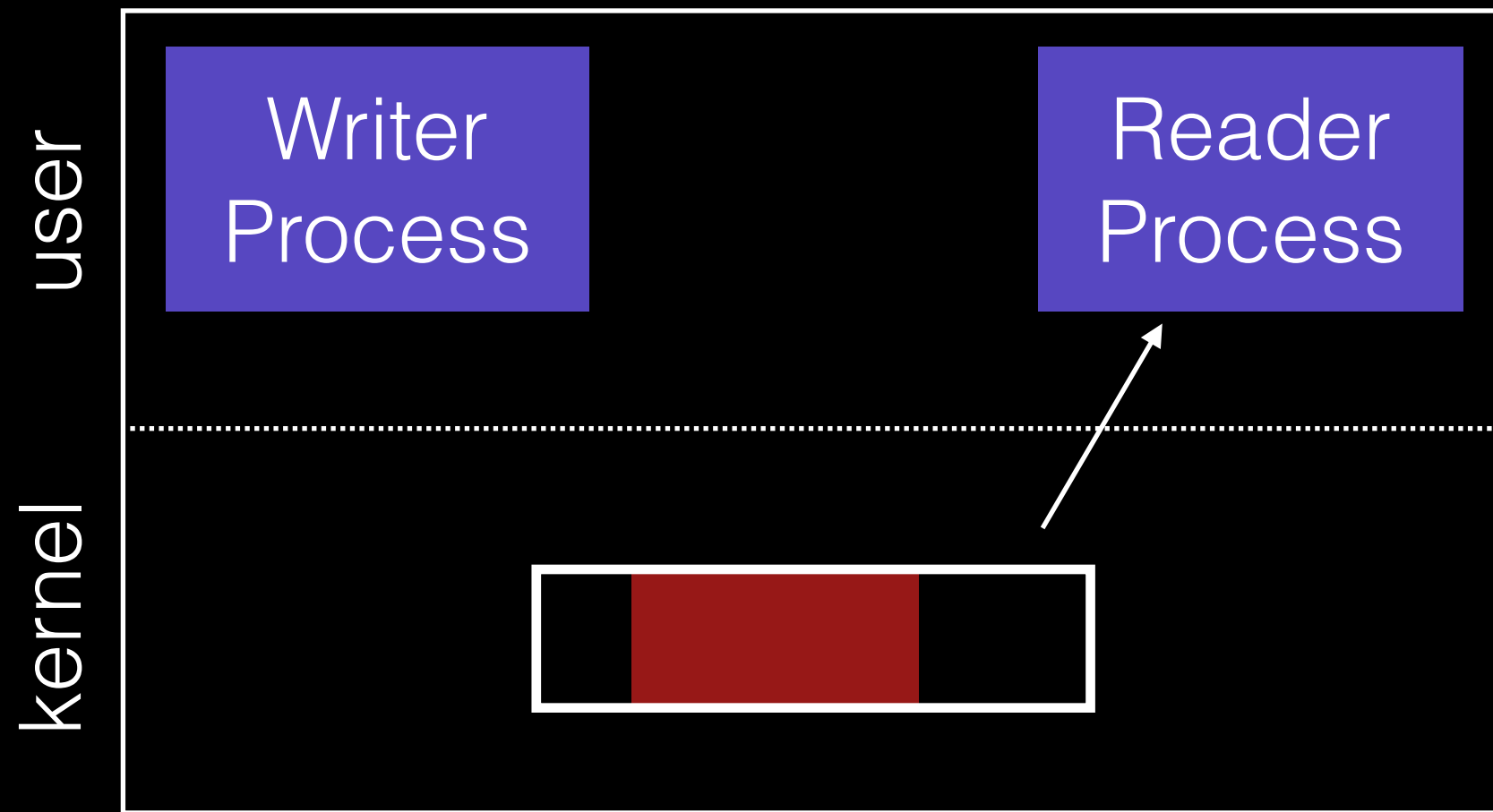
Pipe



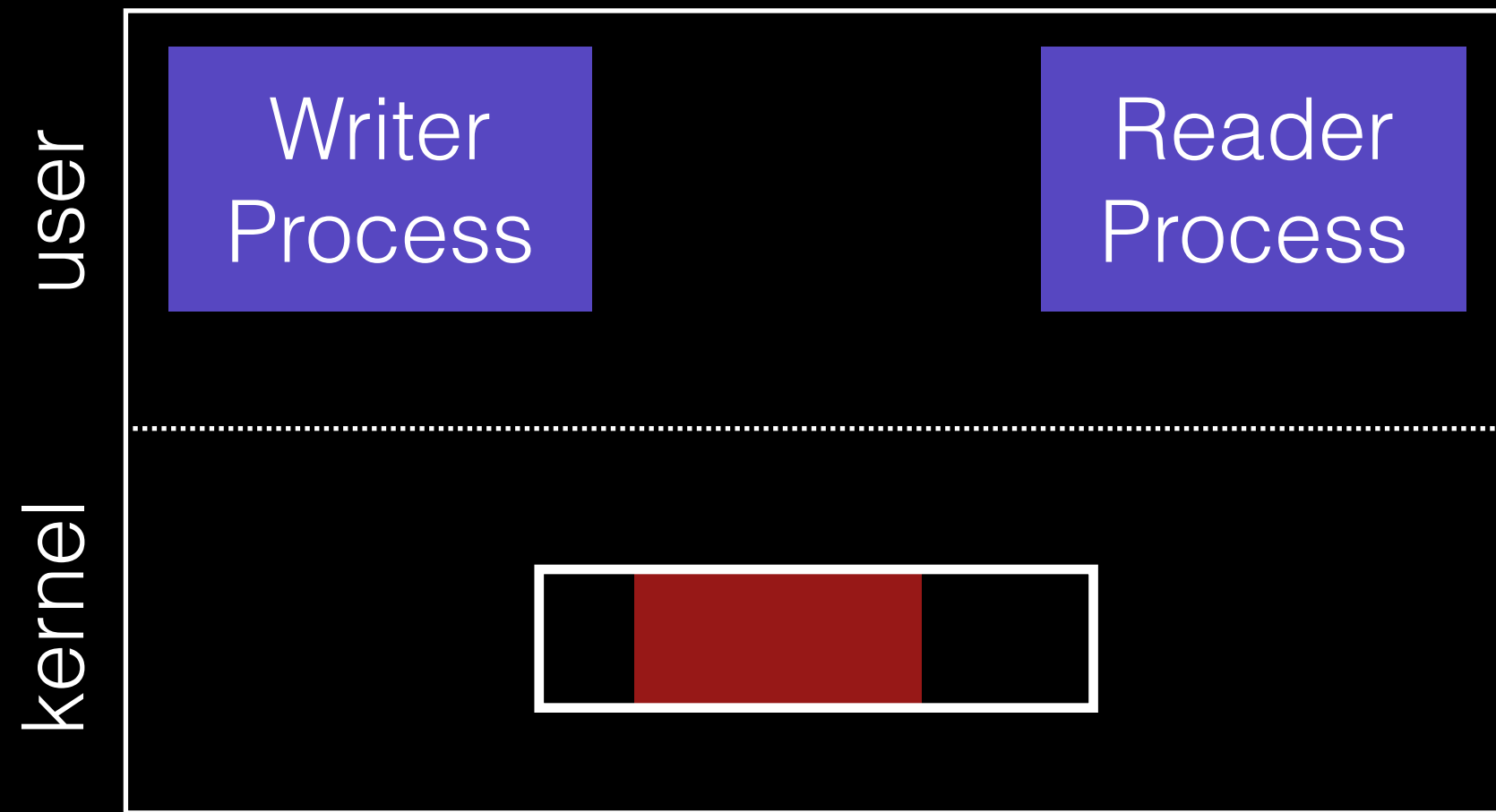
Pipe



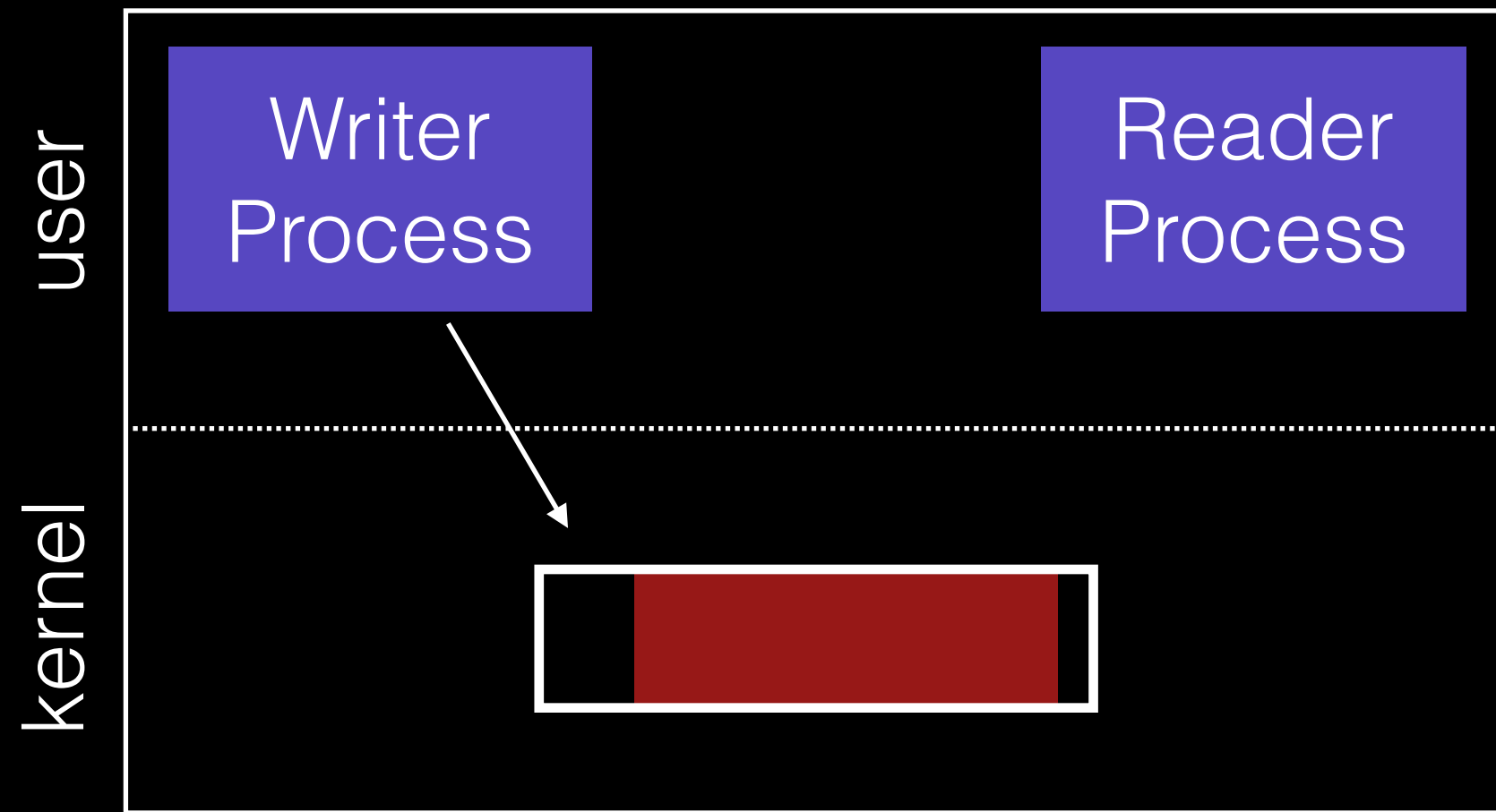
Pipe



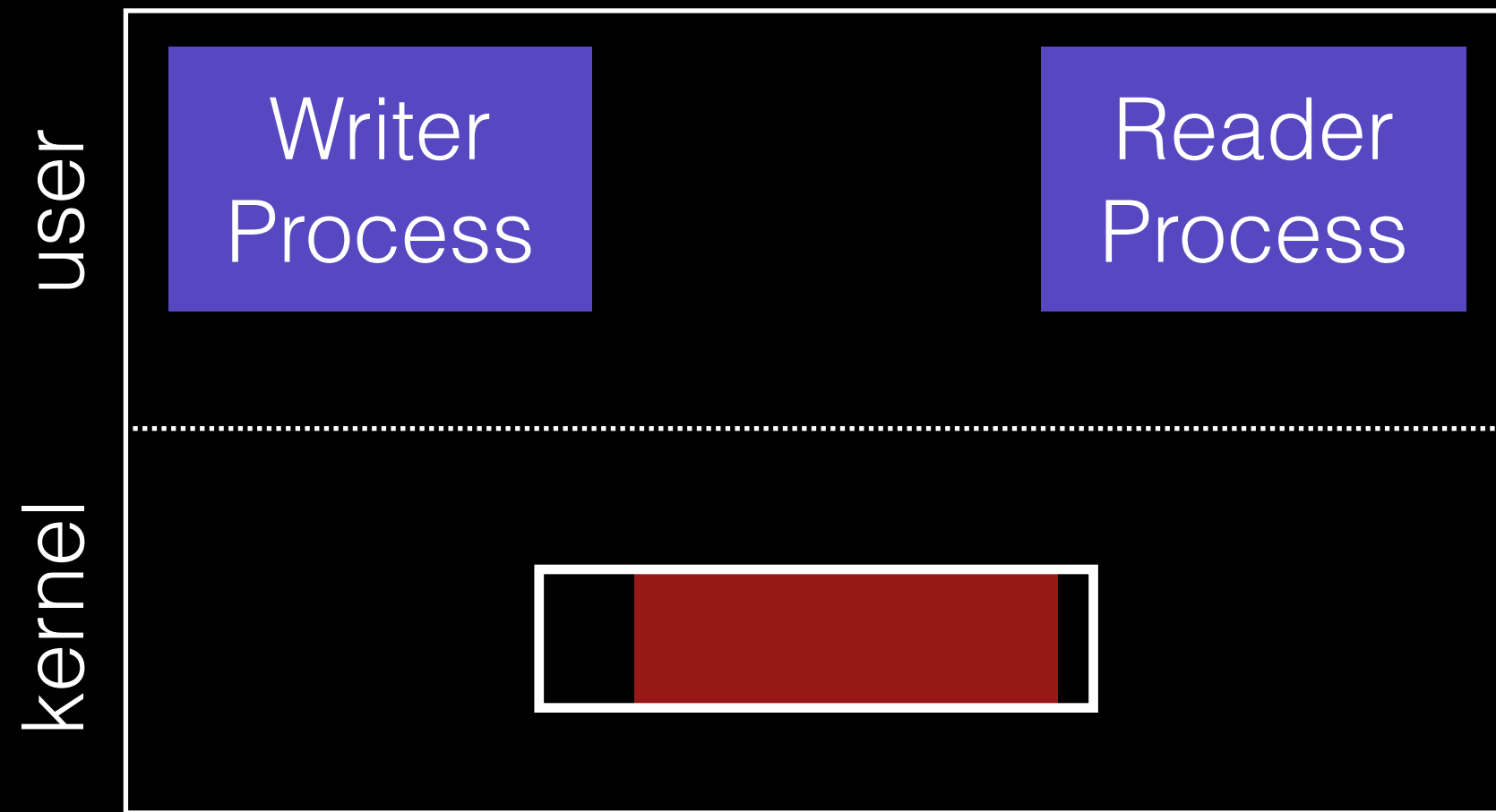
Pipe



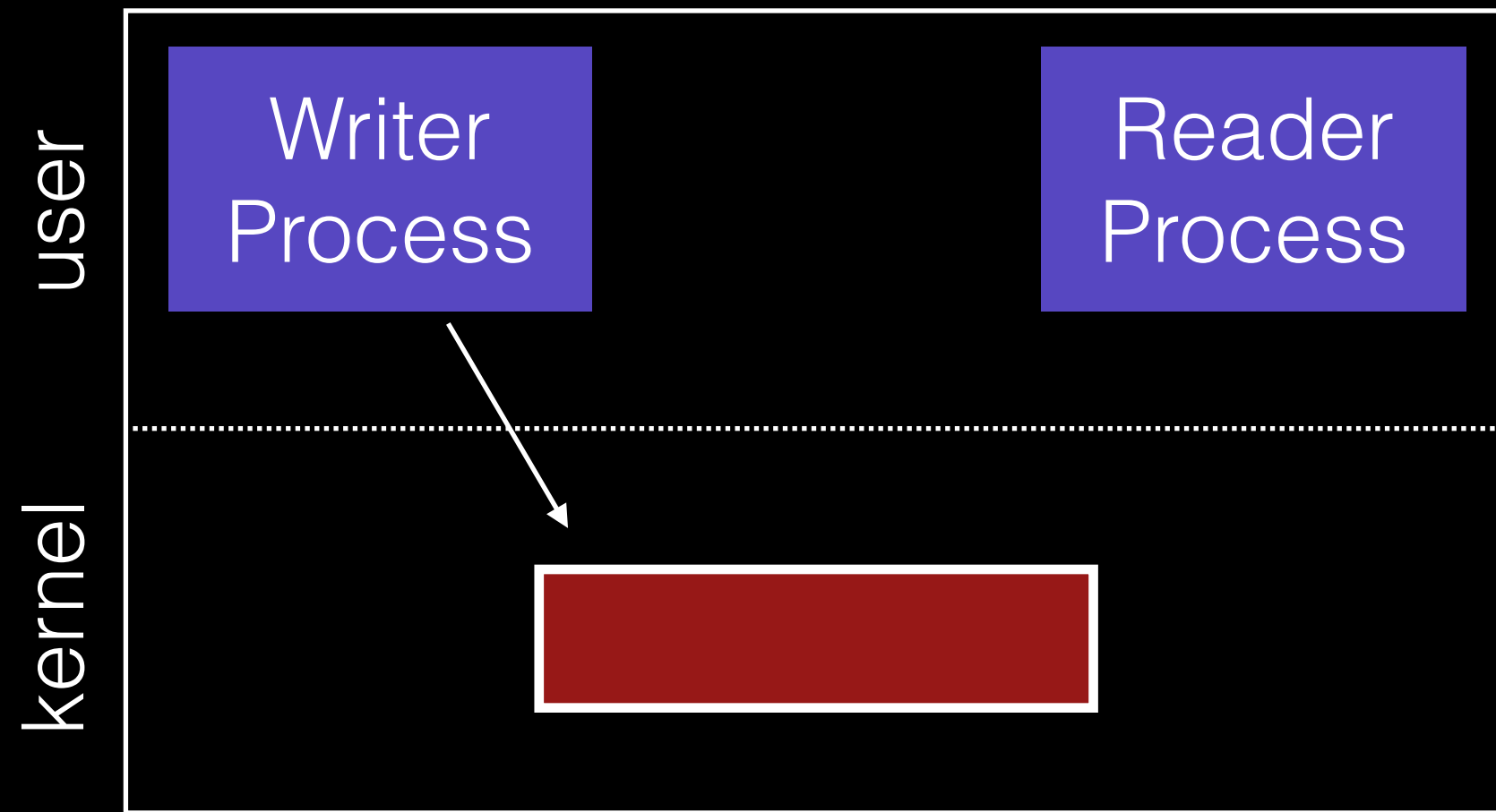
Pipe



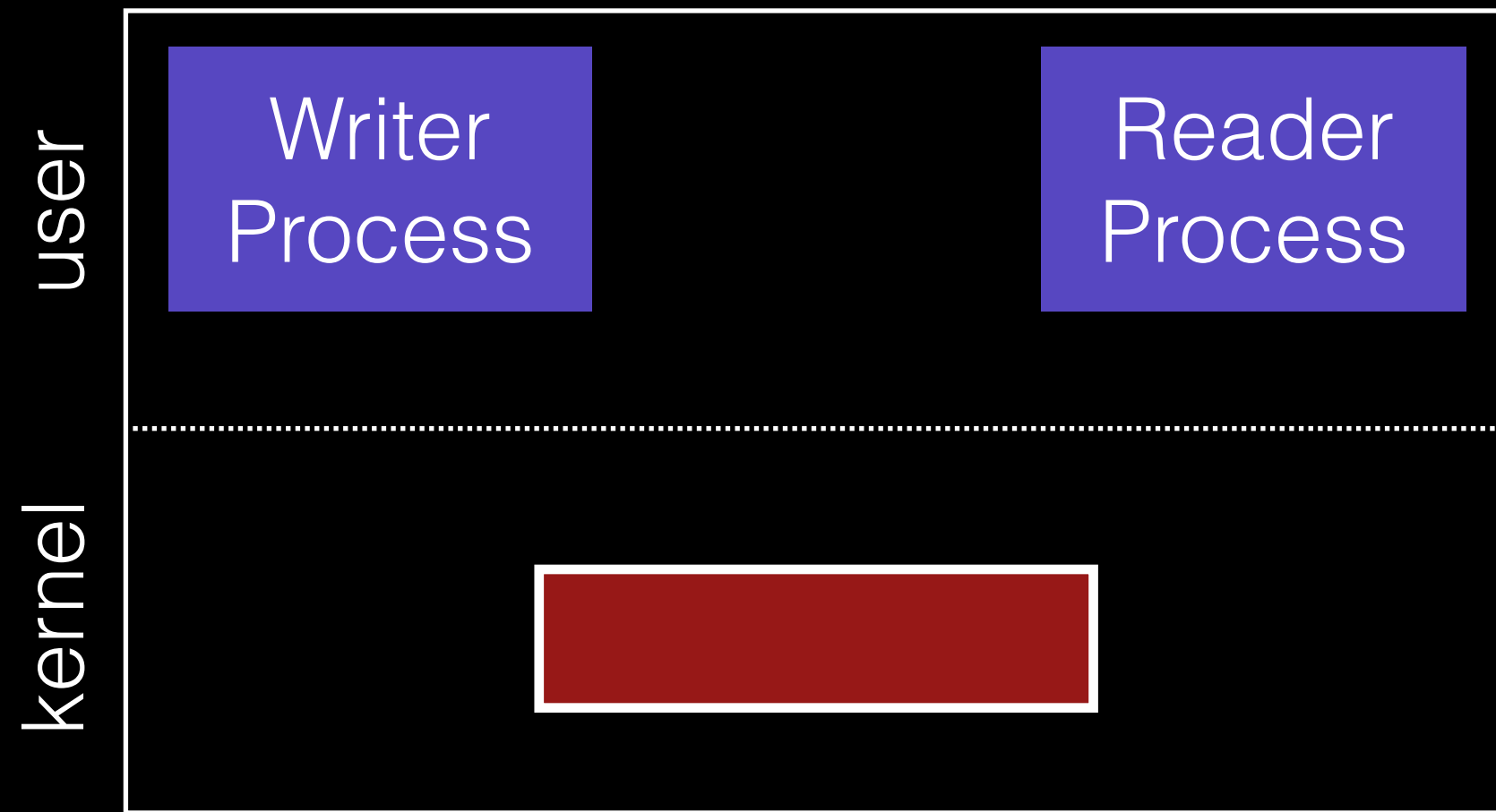
Pipe



Pipe

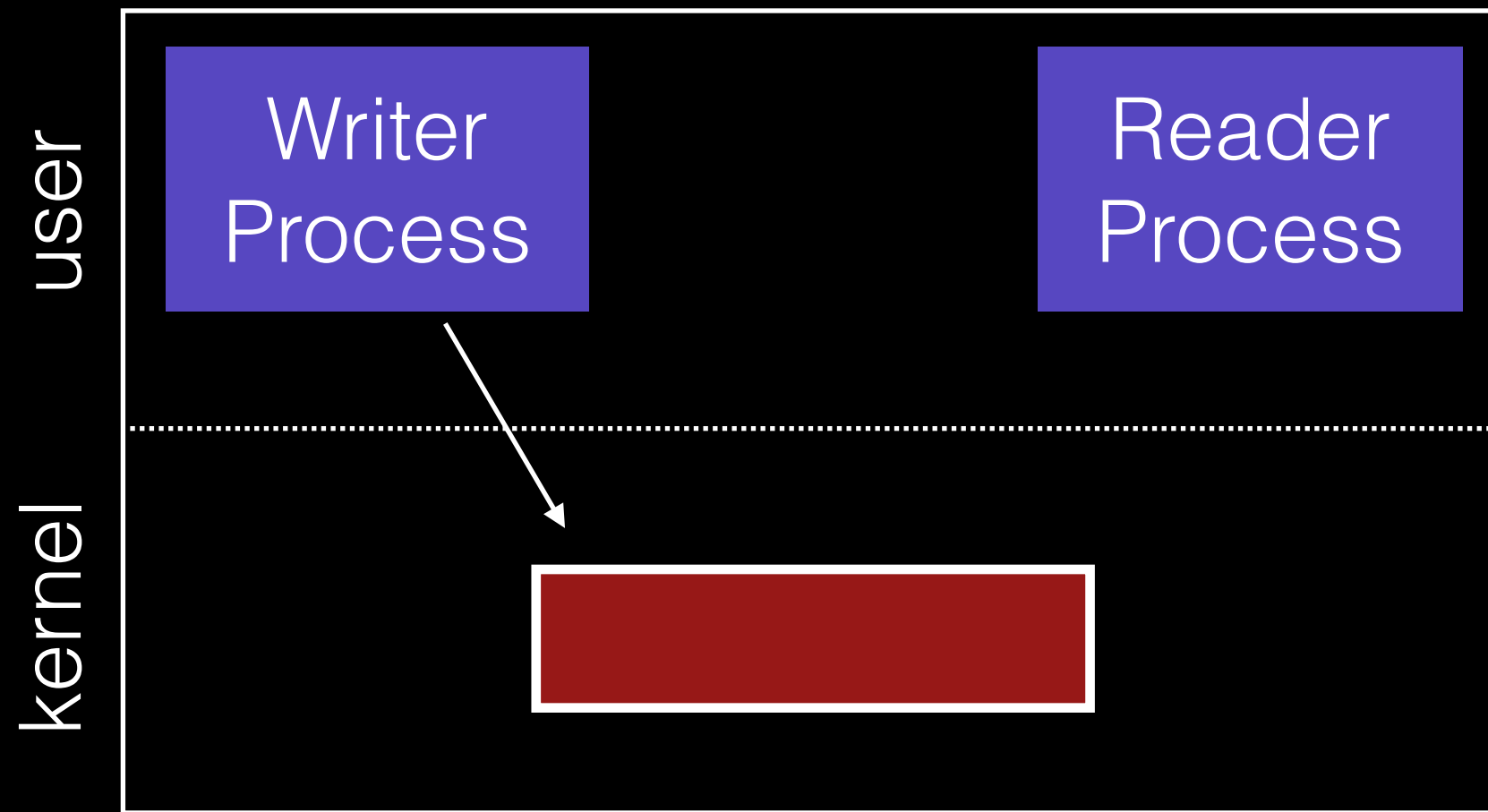


Pipe



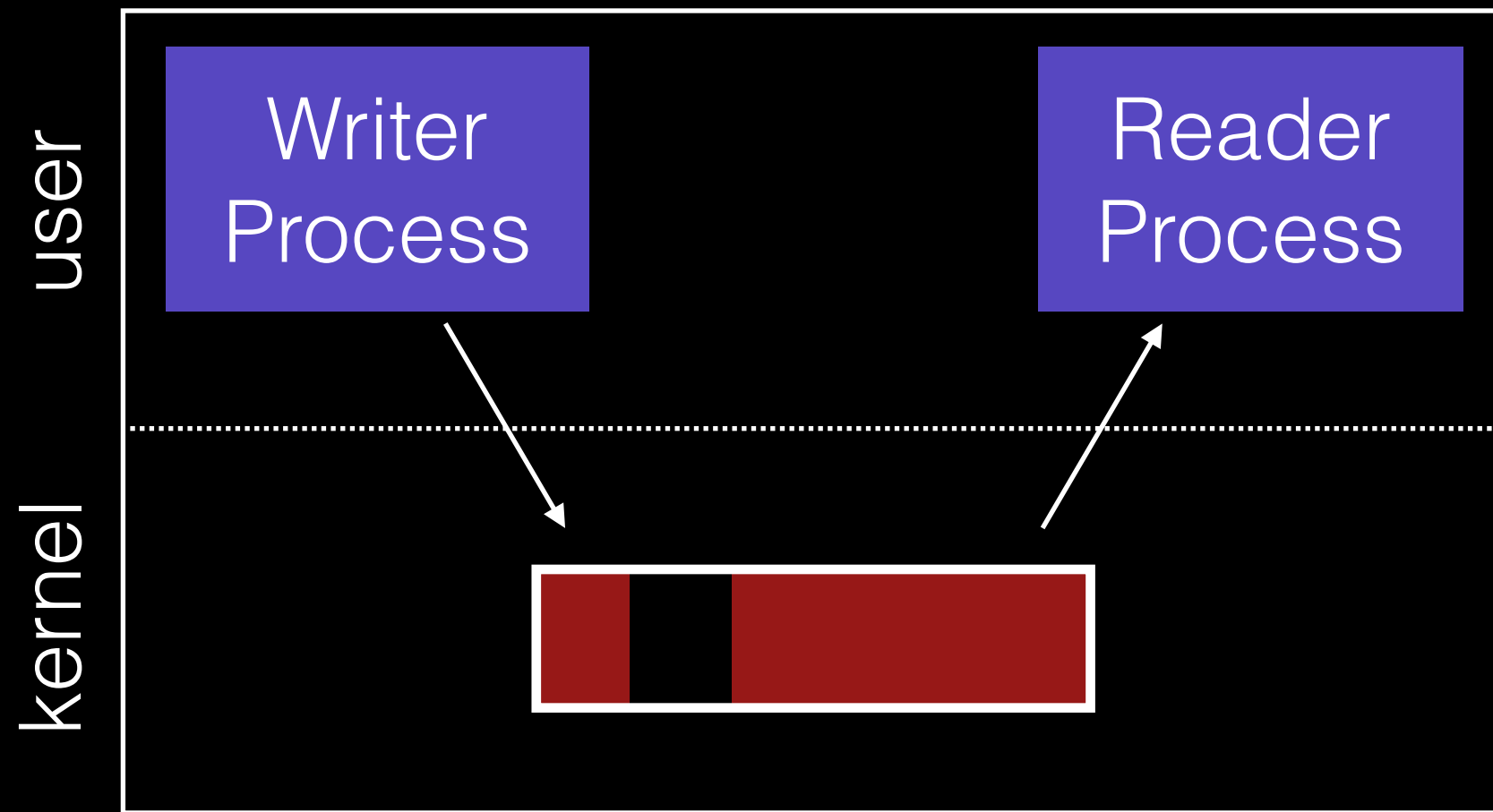
Pipe

write waits for space



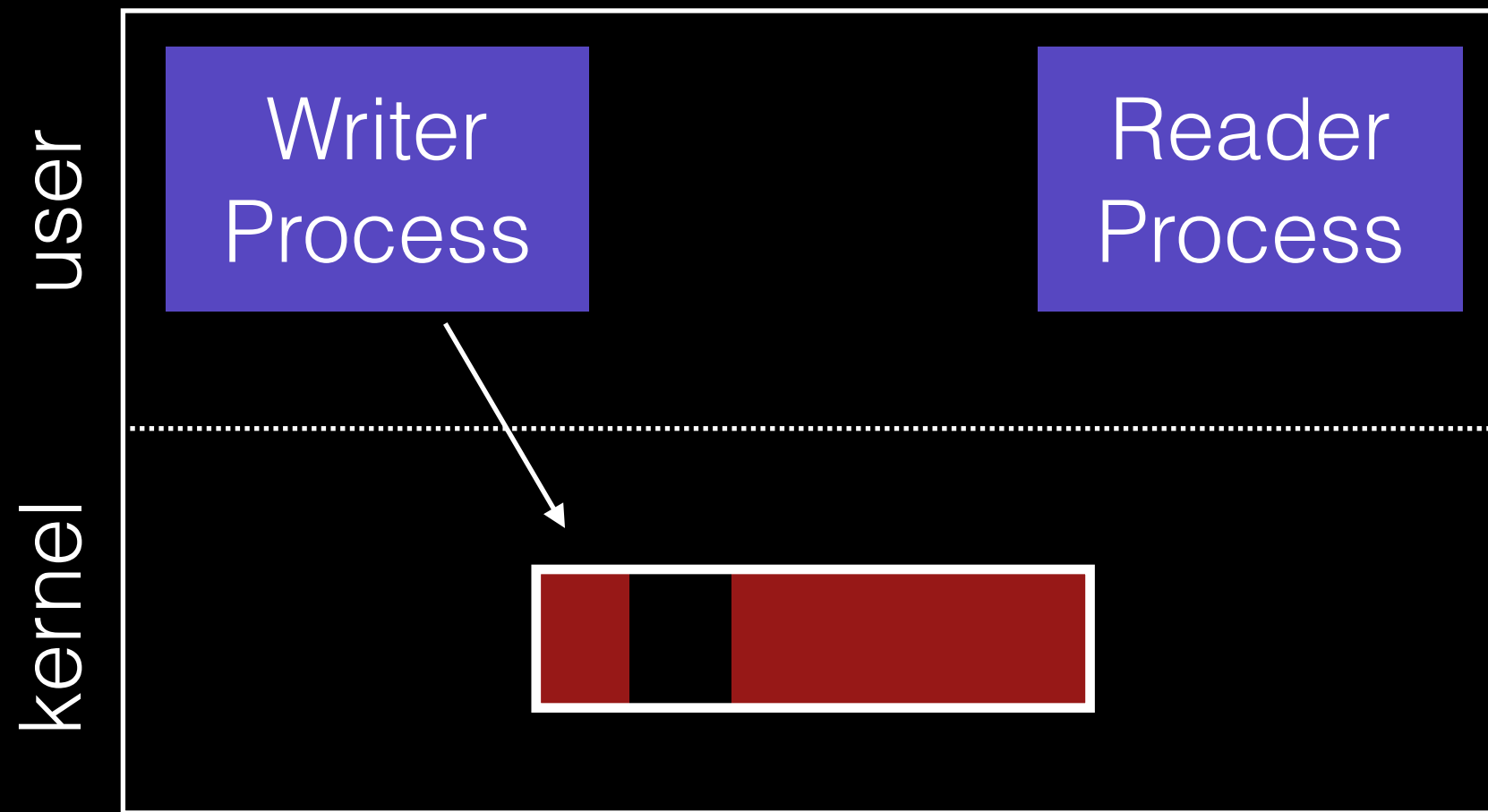
Pipe

write waits for space



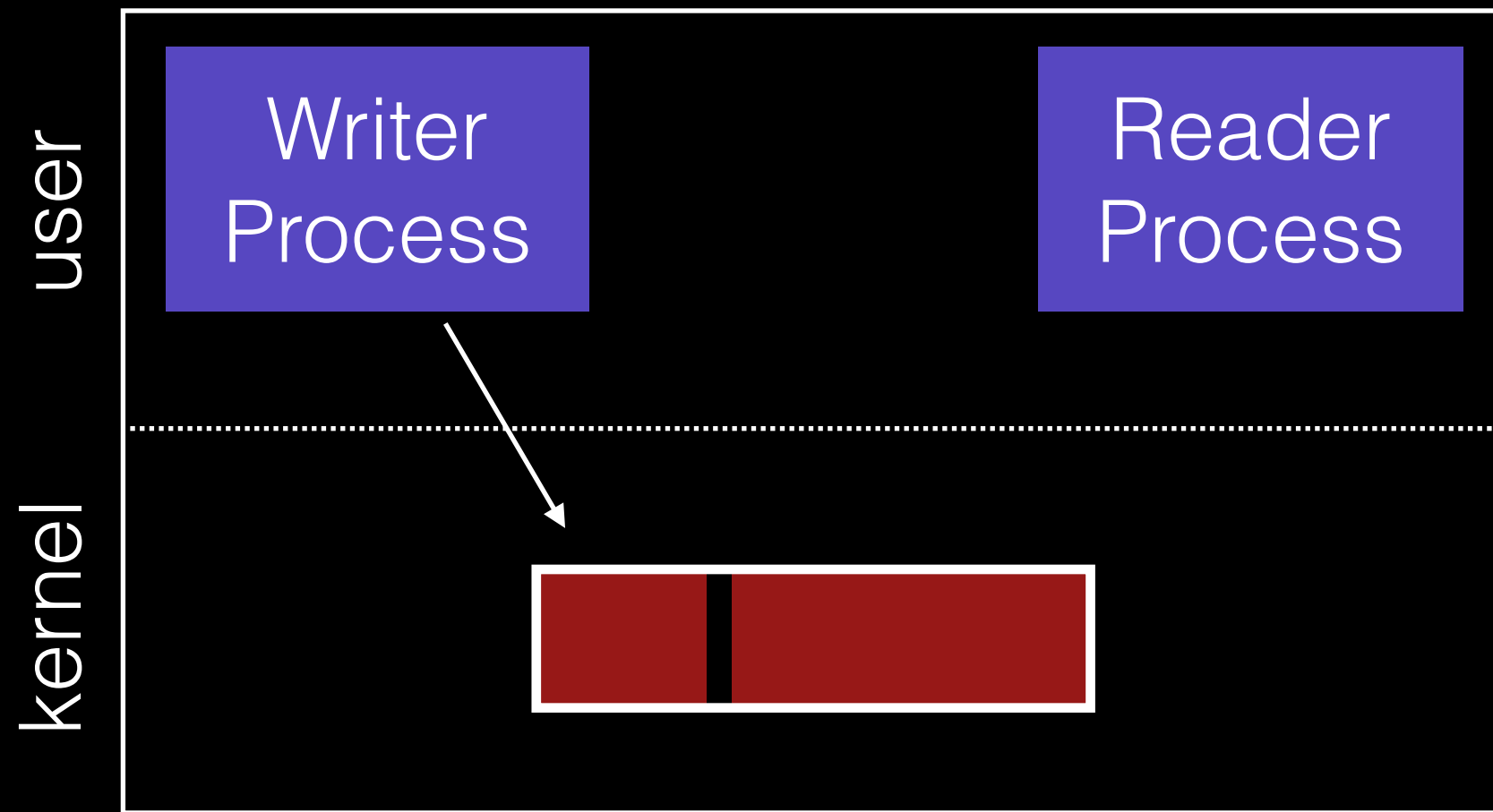
Pipe

write waits for space

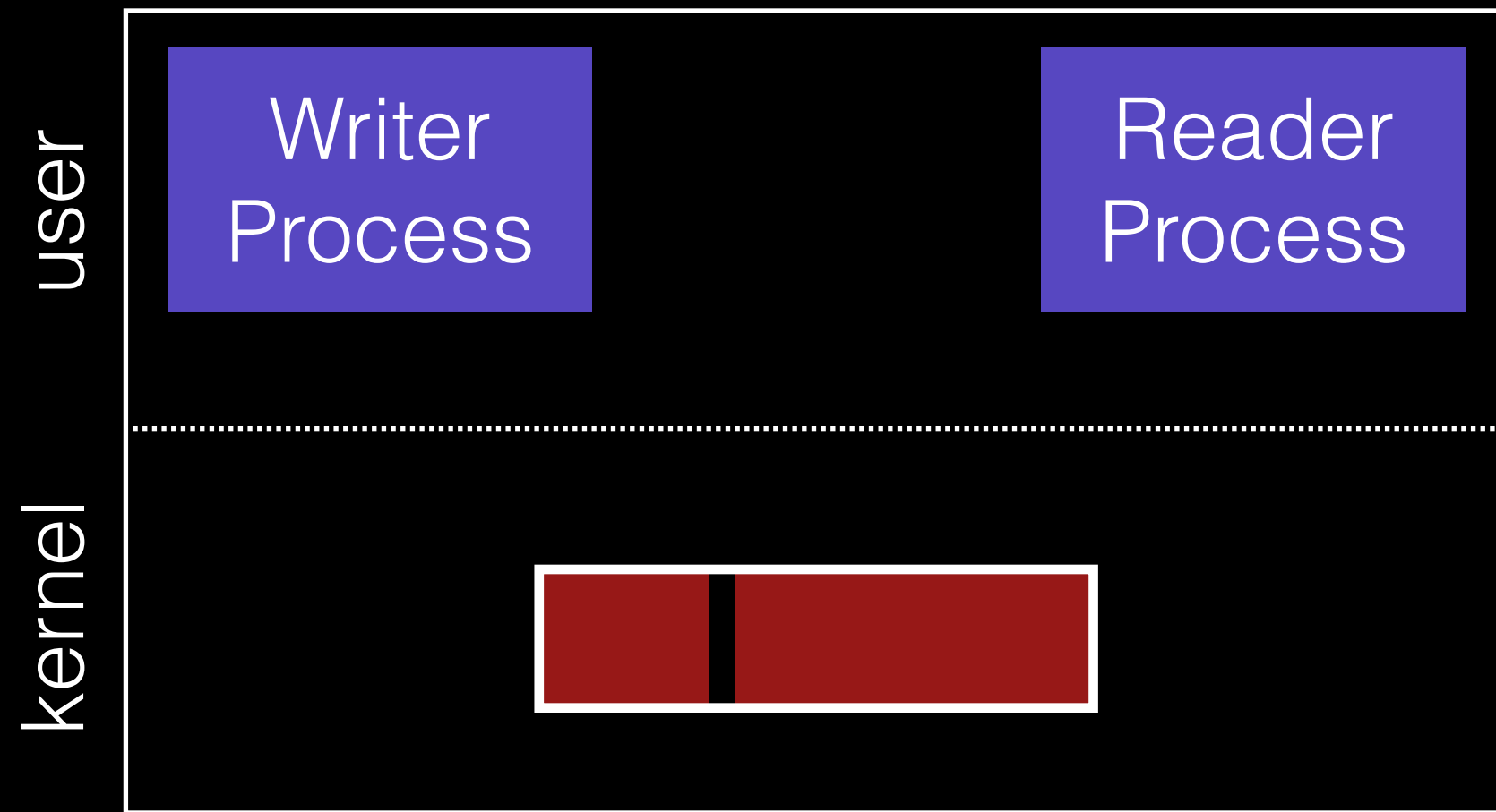


Pipe

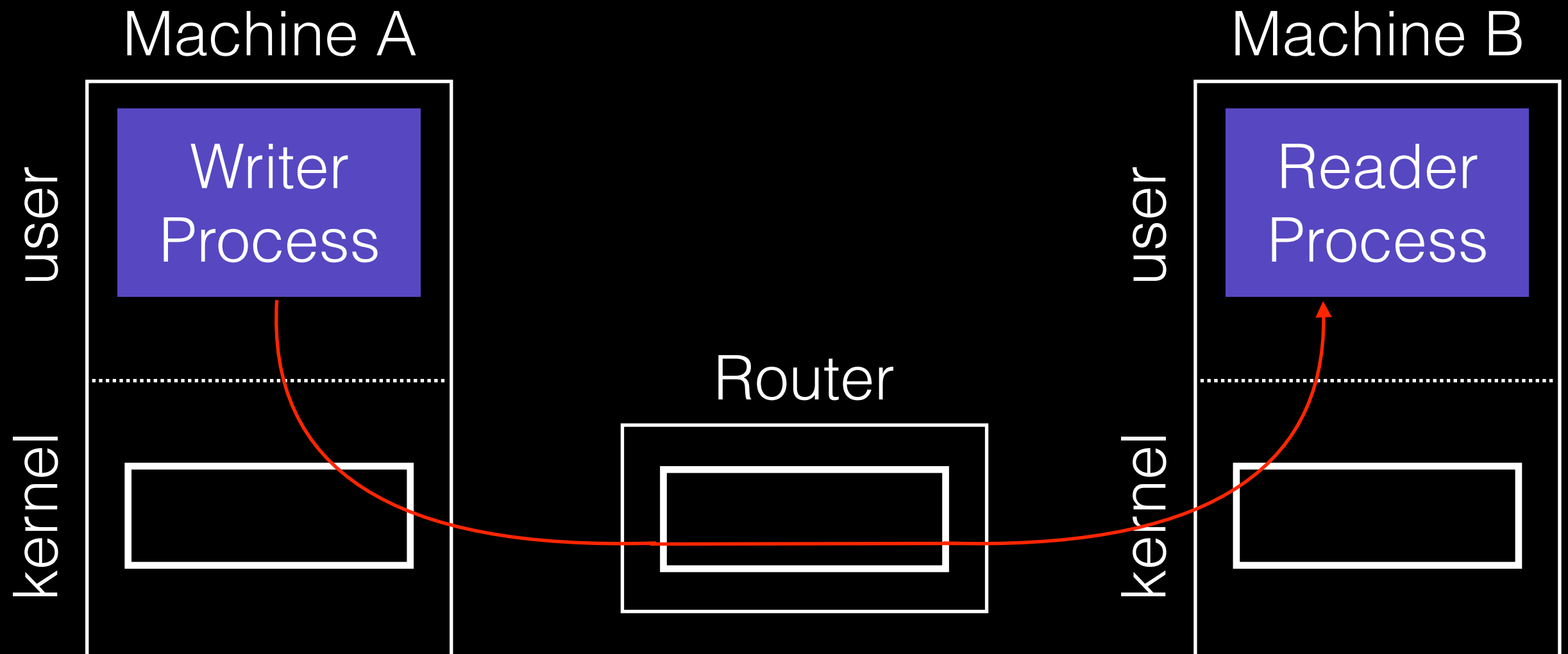
write waits for space



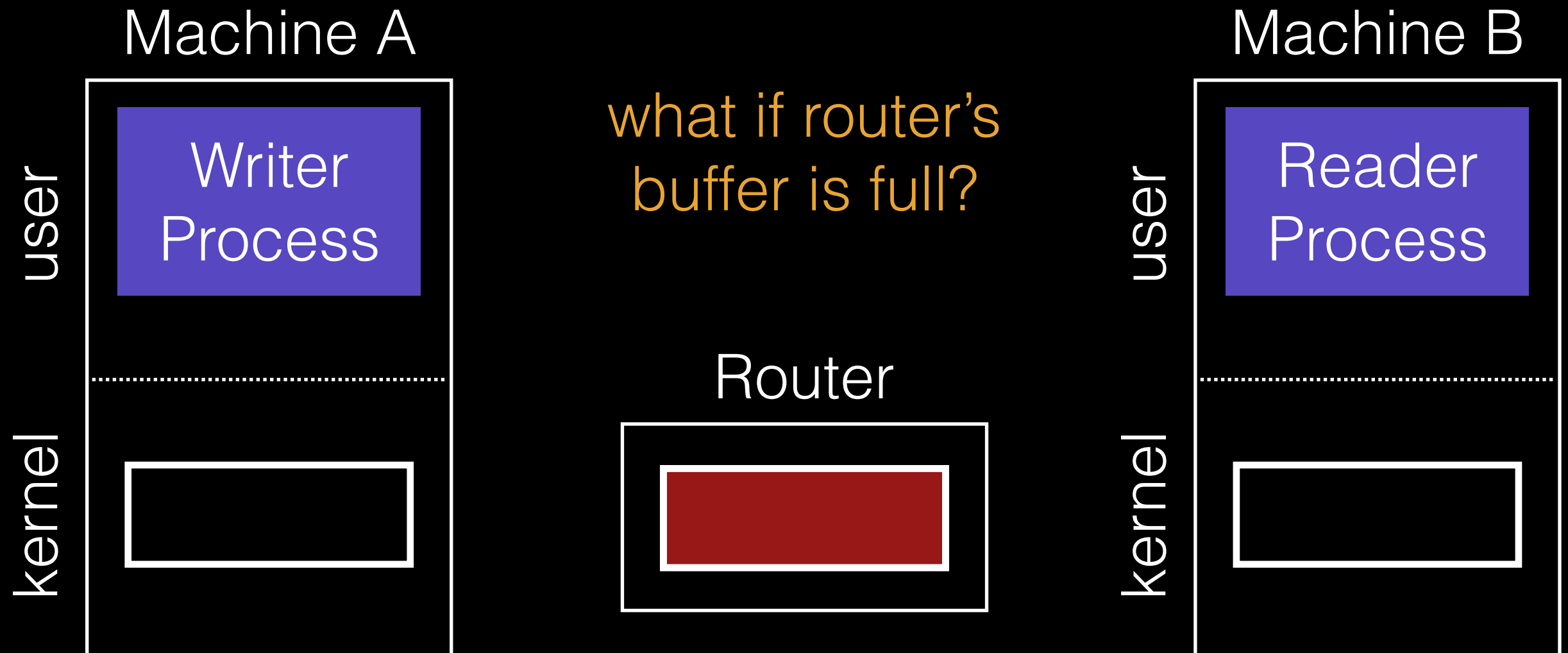
Pipe



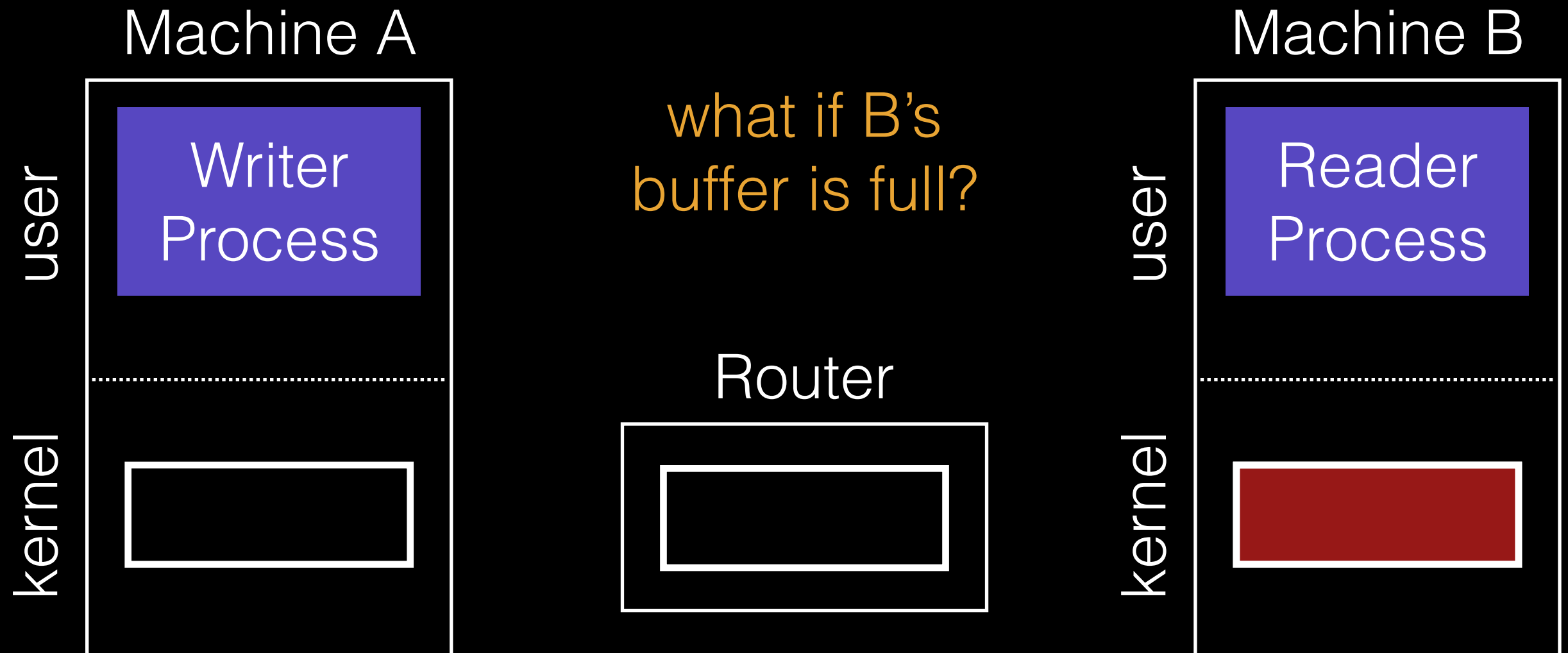
Network Socket



Network Socket

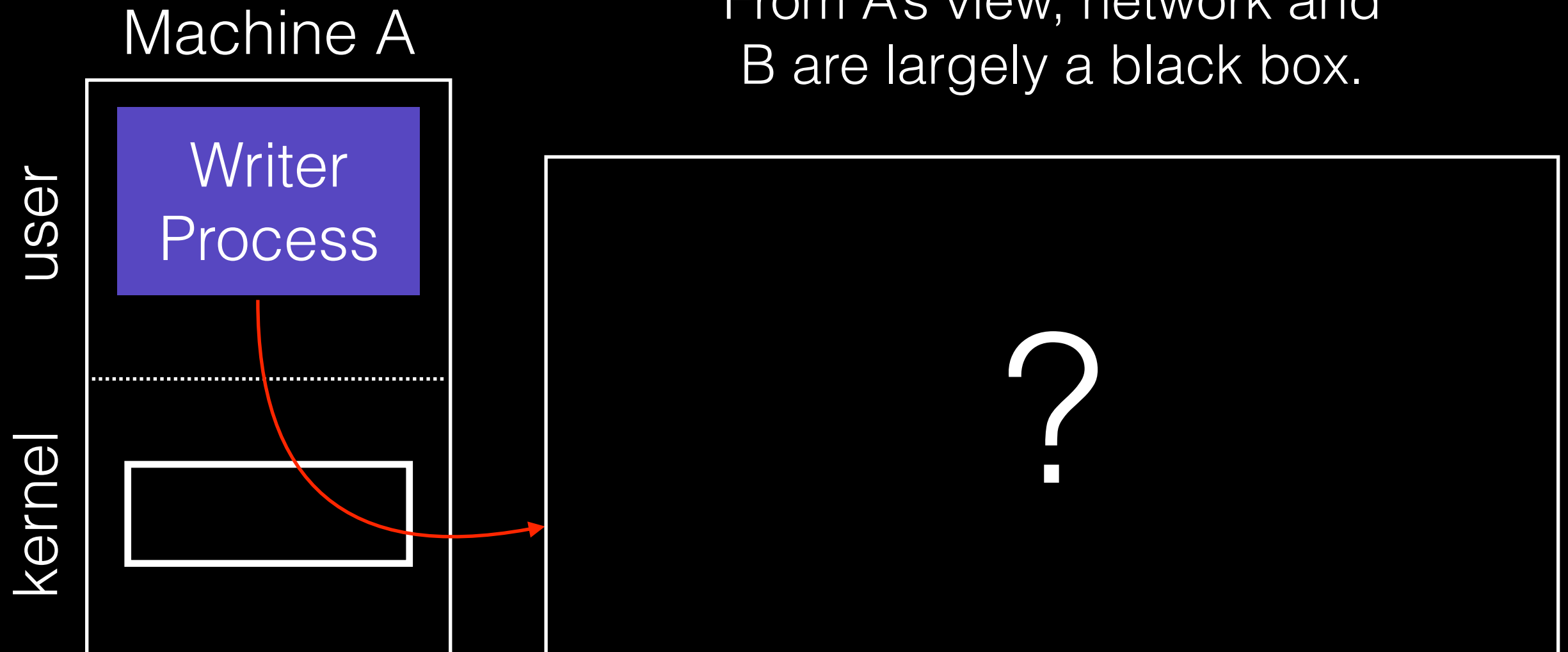


Network Socket



Network Socket

From A's view, network and B are largely a black box.



Overview

Raw messages

Reliable messages

OS abstractions

- virtual memory
- global file system

Programming-languages abstractions

- remote procedure call
-

Raw Messages: UDP

API:

- reads and writes over **socket file descriptors**
- messages sent from/to **ports** to target a process on machine

Provide minimal reliability features:

- messages may be **lost**
 - messages may be **reordered**
 - messages may be **duplicated**
 - **only protection**: checksums
-

Raw Messages: UDP

Advantages

- lightweight
- some applications make better reliability decisions themselves (e.g., video conferencing programs)

Disadvantages

- more difficult to write application correctly

Overview

Raw messages

Reliable messages

OS abstractions

- virtual memory
- global file system

Programming-languages abstractions

- remote procedure call
-

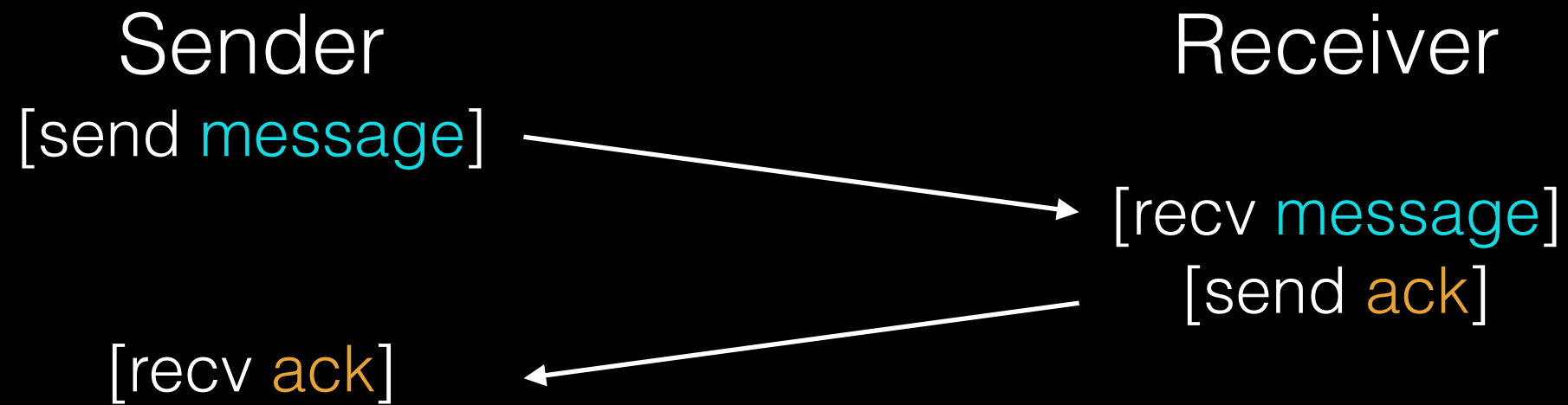
Strategy

Using software, build reliable, **logical connections** over unreliable connections.

Strategies:

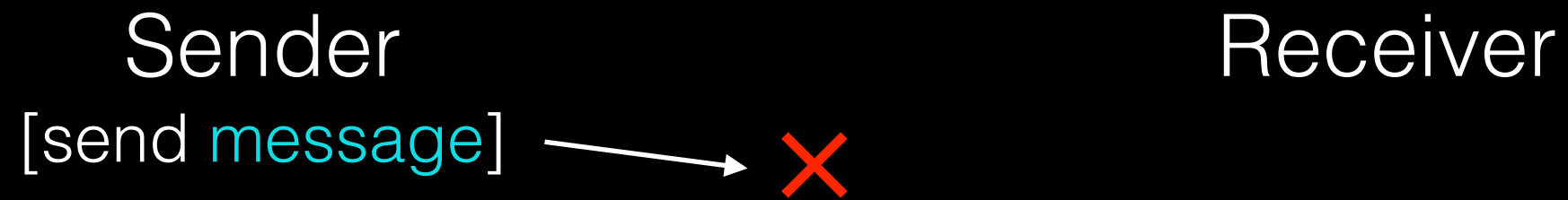
- acknowledgment

ACK



Sender knows message was received.

ACK



Sender misses ACK... What to do?

Strategy

Using software, build reliable, **logical connections** over unreliable connections.

Strategies:

- acknowledgment

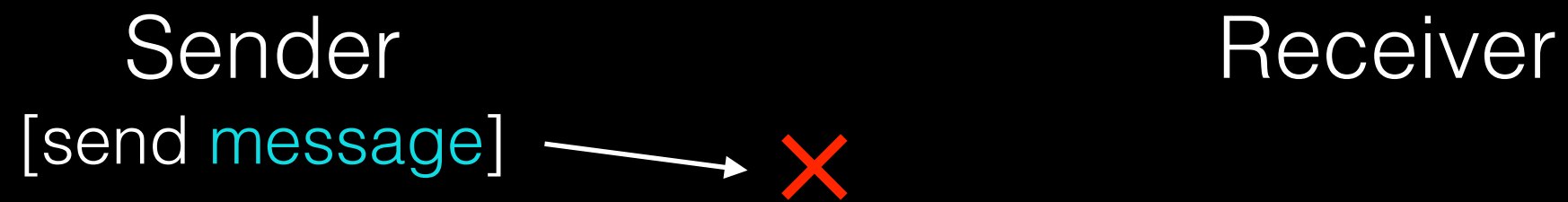
Strategy

Using software, build reliable, **logical connections** over unreliable connections.

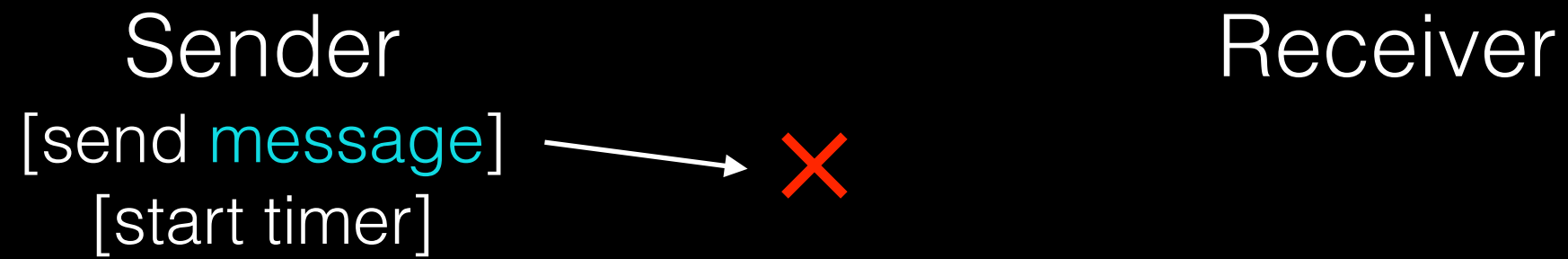
Strategies:

- acknowledgment
- timeout

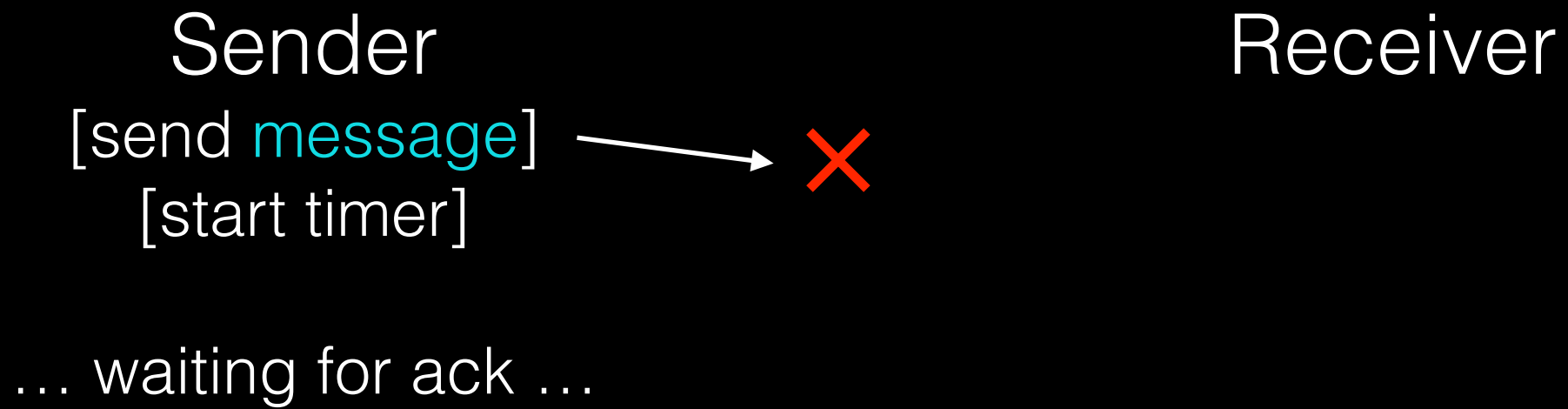
Timeout



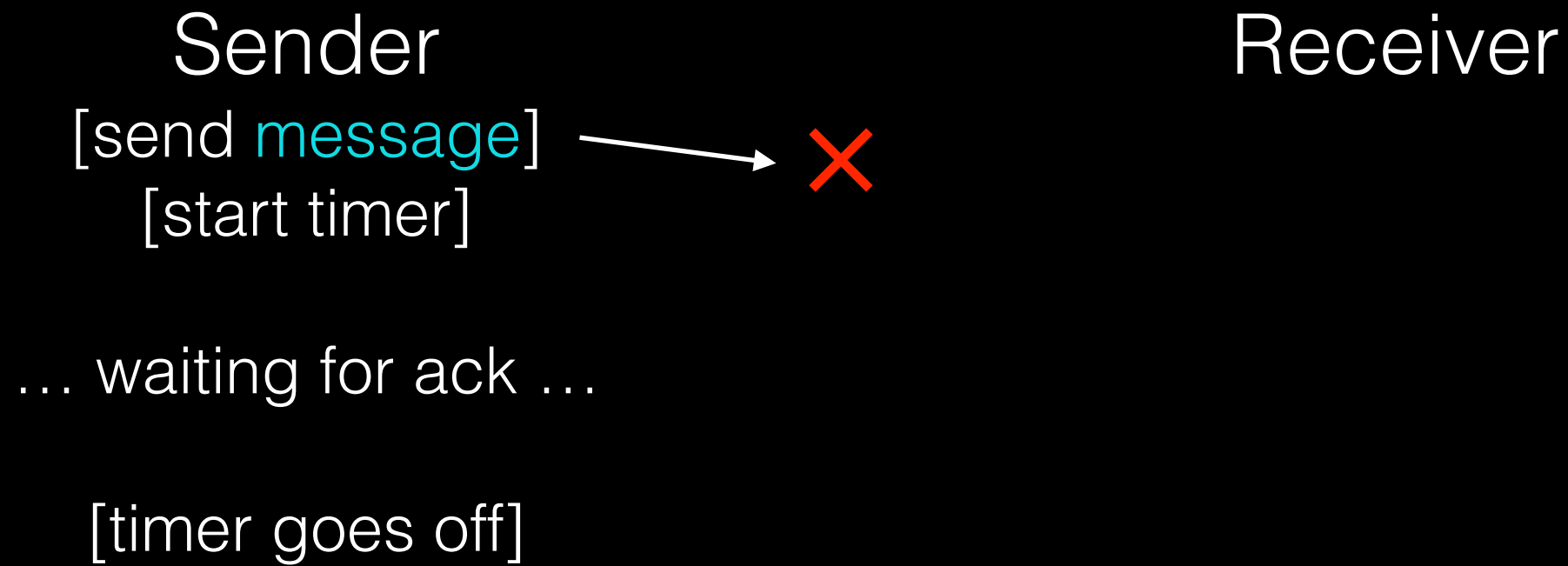
Timeout



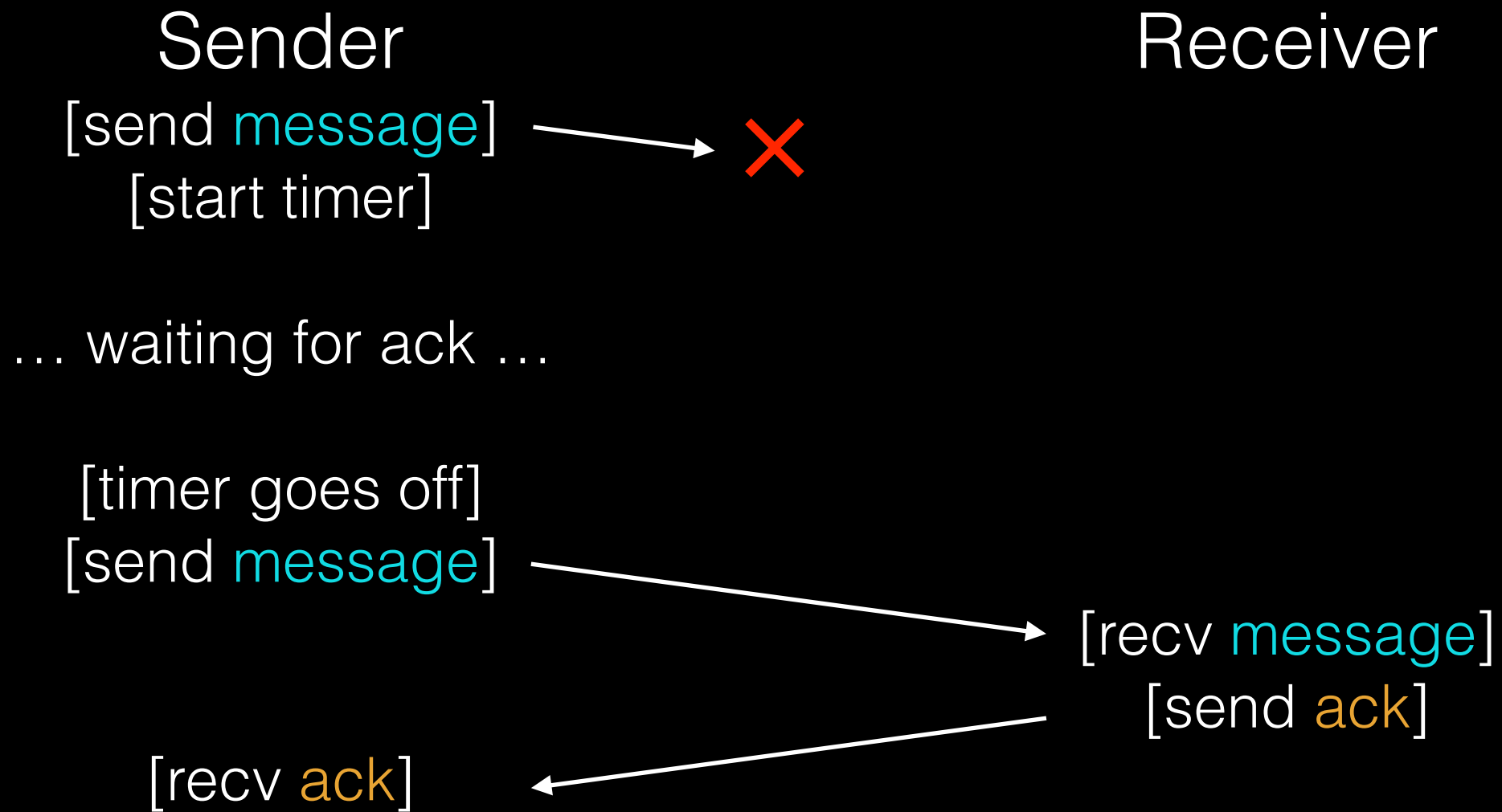
Timeout



Timeout



Timeout



Timeout: Issue 1

How long to wait?

Timeout: Issue 1

How long to wait?

Too long: system feels unresponsive

Too short: messages needlessly re-sent

Messages may have been dropped due to overloaded server. Aggressive clients worsen this.

Timeout: Issue 1

How long to wait?

One strategy: **be adaptive**.

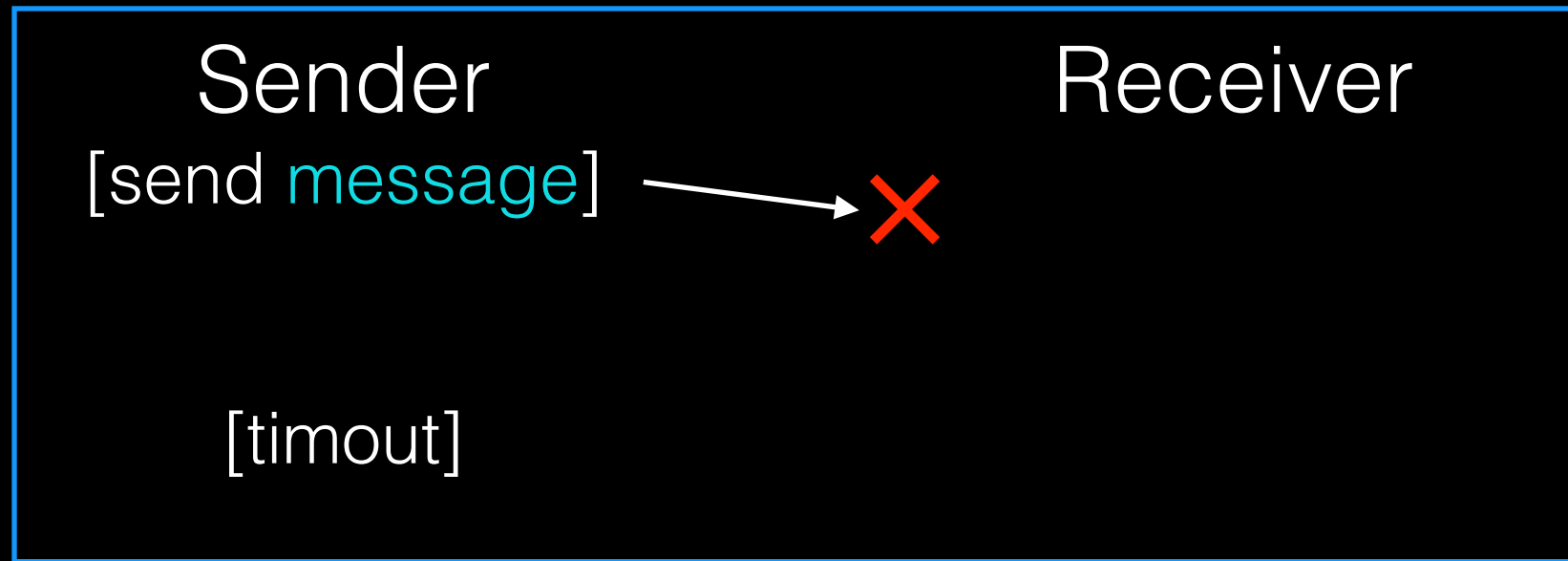
Adjust time based on how long acks usually take.

For each missing ack, wait longer between retries.

Timeout: Issue 2

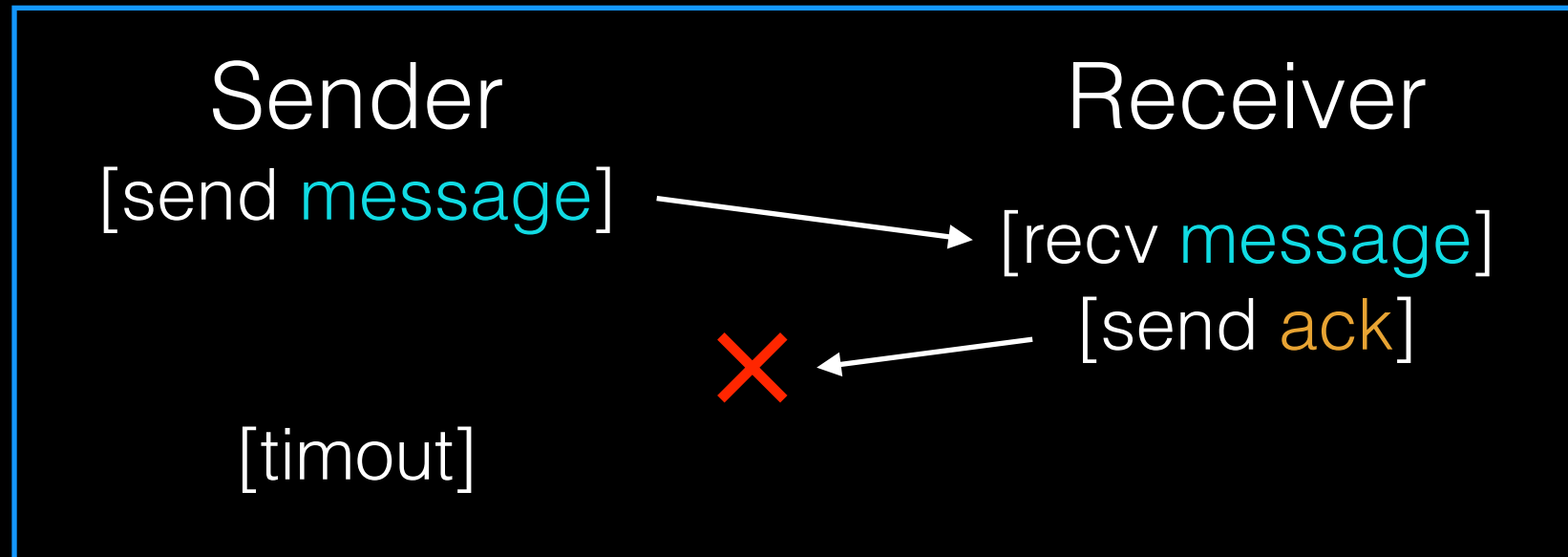
What does a lost ack really mean?

Case 1



How can sender tell between these two cases?

Case 2



Timeout: Issue 2

What does a lost ack really mean?

ACK: message received **exactly** once

No ACK: message received **at most** once

Timeout: Issue 2

What does a lost ack really mean?

ACK: message received **exactly** once

No ACK: message received **at most** once

What if message is command to increment counter?

Proposed Solution

Sender could send an **AckAck** so receiver knows whether to retry sending an **Ack**.

Sound good?

Aside: Two Generals' Problem



Aside: Two Generals' Problem



Suppose a generals agree after N messages.
Did the arrival of the N 'th message change anybody's decision?

Aside: Two Generals' Problem



Suppose a generals agree after N messages.

Did the arrival of the N 'th message change anybody's decision?

- if yes: then what if the N 'th message had been lost?
- if no: then why bother sending N messages?

Timeout: Issue 2

What does a lost ack really mean?

ACK: message received **exactly** once

No ACK: message received **at most** once

What if message is command to increment counter?

Strategy

Using software, build reliable, **logical connections** over unreliable connections.

Strategies:

- acknowledgment
- timeout

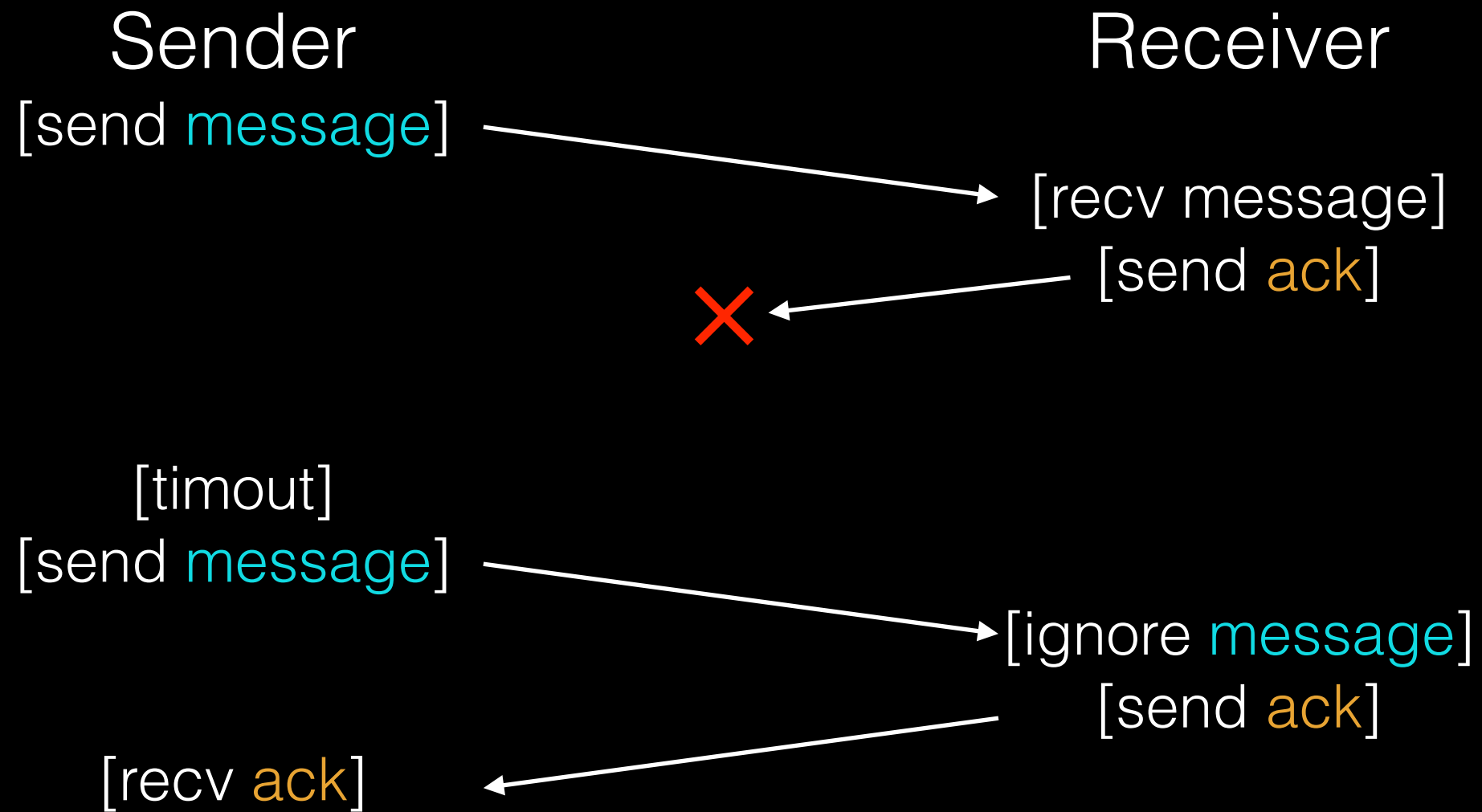
Strategy

Using software, build reliable, **logical connections** over unreliable connections.

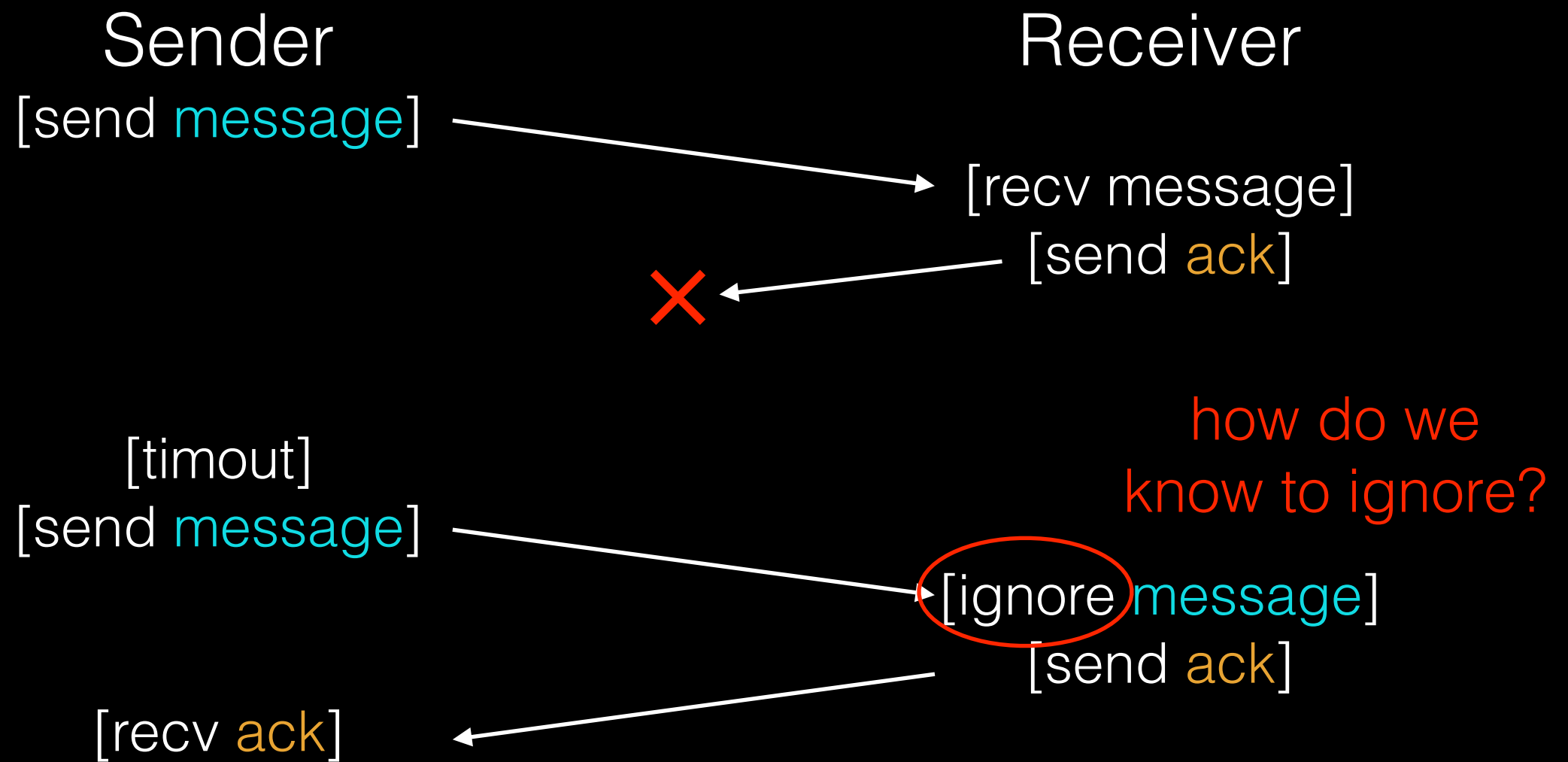
Strategies:

- acknowledgment
- timeout
- remember sent messages

Receiver Remembers Messages



Receiver Remembers Messages



Solutions

Solution 1: remember *every* message ever sent.

Solutions

Solution 1: remember *every* message ever sent.

Solution 2: sequence numbers

- give each message a *seq number*
- receiver knows all messages before an *N* have been seen
- receiver remembers messages sent after *N*

TCP

Most popular protocol based on seq nums.

Also buffers messages so they arrive **in order**.

Timeouts are **adaptive**.

Overview

Raw messages

Reliable messages

OS abstractions

- virtual memory
- global file system

Programming-languages abstractions

- remote procedure call
-

Virtual Memory

Inspiration: threads share memory

Idea: processes on different machines share mem

Virtual Memory

Inspiration: threads share memory

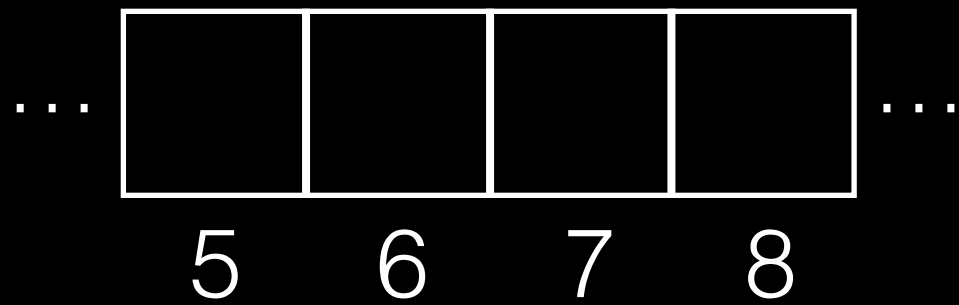
Idea: processes on different machines share mem

Strategy:

- a bit like swapping we saw before
 - instead of swap to disk, swap to other machine
 - sometimes multiple copies may be in memory on different machines
-

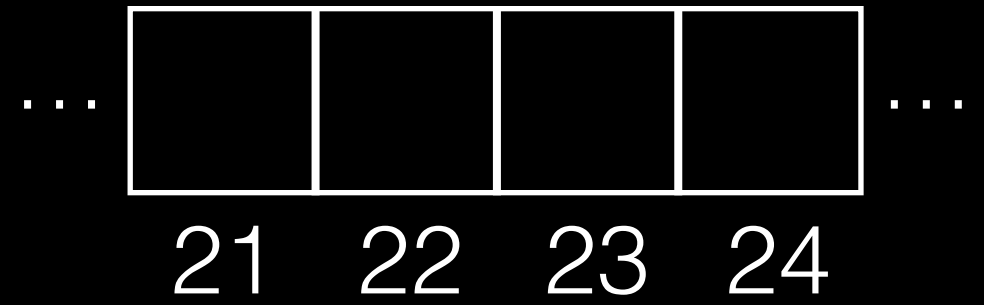
Process on Machine A

PFN	valid	present
-	0	-
-	0	-
-	0	-
-	0	-



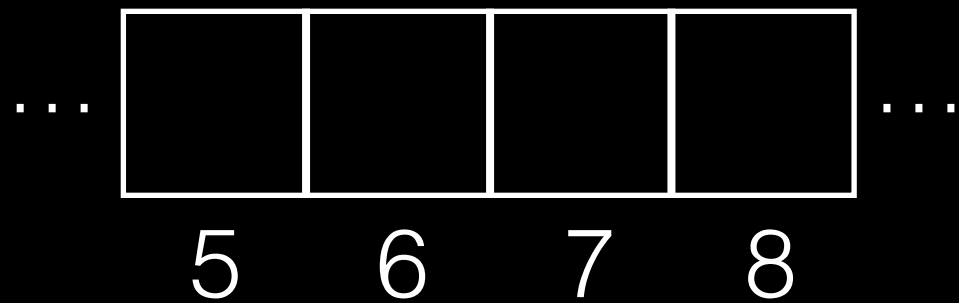
Process on Machine B

PFN	valid	present
-	0	-
-	0	-
-	0	-
-	0	-



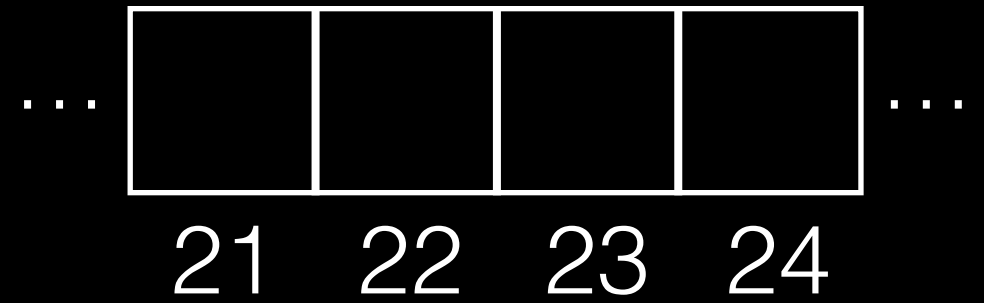
Process on Machine A

PFN	valid	present
-	0	-
-	1	0
-	1	0
-	1	0



Process on Machine B

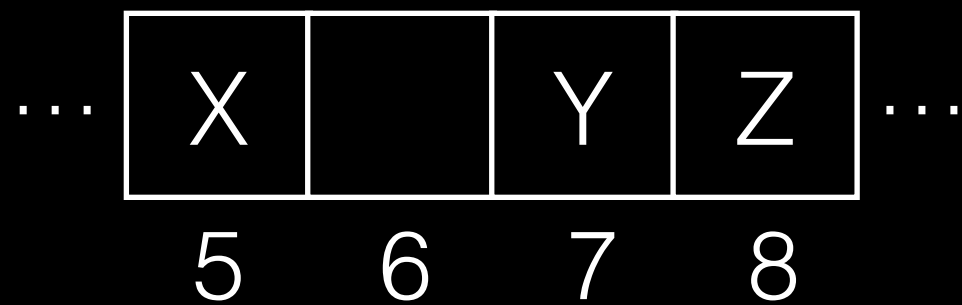
PFN	valid	present
-	0	-
-	1	0
-	1	0
-	1	0



map 3-page region into both memories.

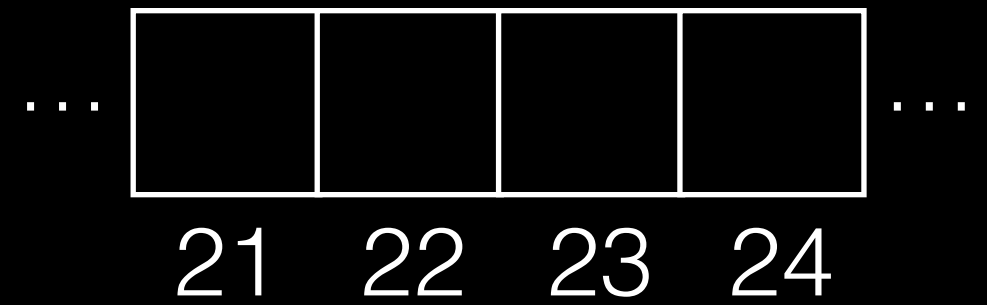
Process on Machine A

PFN	valid	present
-	0	-
5	1	1
7	1	1
8	1	1



Process on Machine B

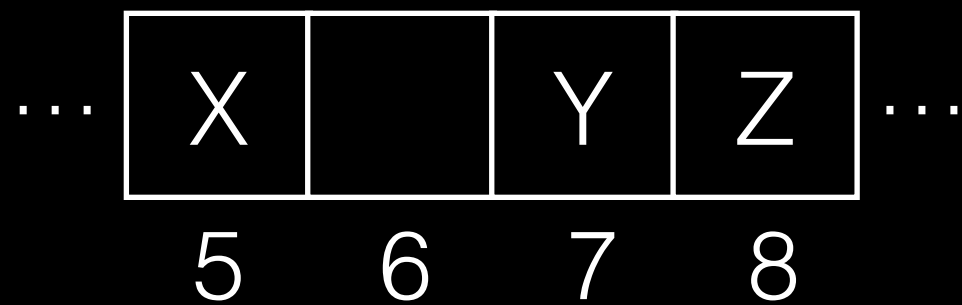
PFN	valid	present
-	0	-
-	1	0
-	1	0
-	1	0



A writes X,Y,Z

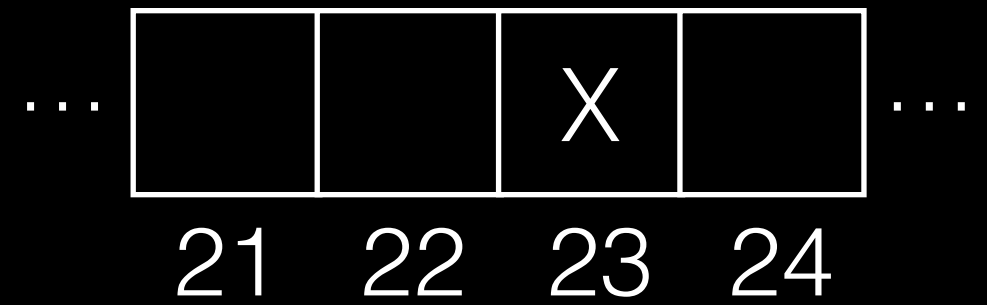
Process on Machine A

PFN	valid	present
-	0	-
5	1	1
7	1	1
8	1	1



Process on Machine B

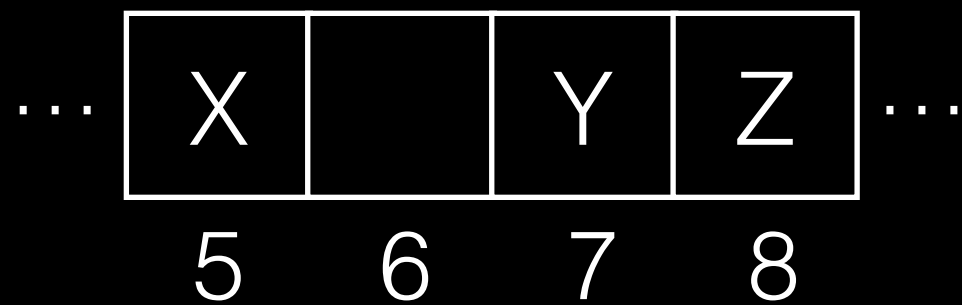
PFN	valid	present
-	0	-
23	1	1
-	1	0
-	1	0



B reads 1st page

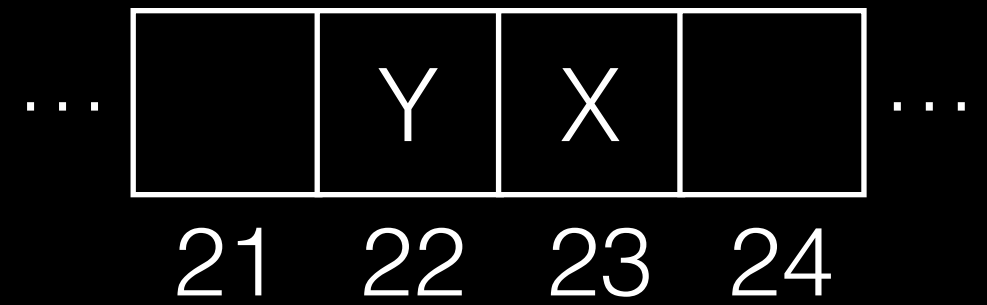
Process on Machine A

PFN	valid	present
-	0	-
5	1	1
7	1	1
8	1	1



Process on Machine B

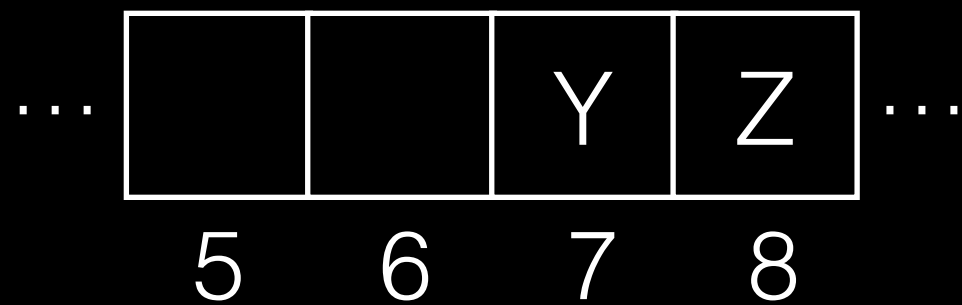
PFN	valid	present
-	0	-
23	1	1
22	1	1
-	1	0



B reads 2st page

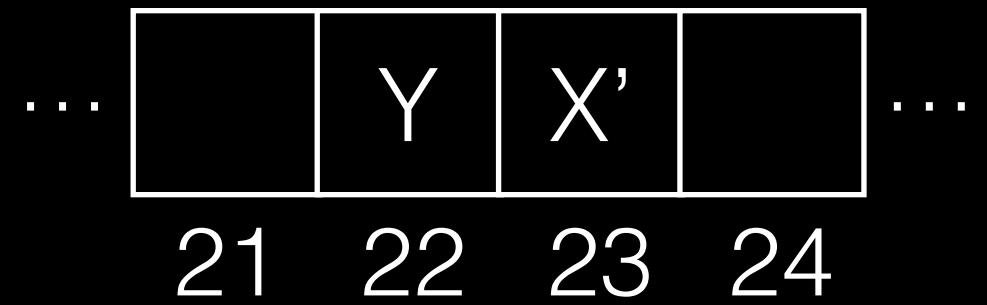
Process on Machine A

PFN	valid	present
-	0	-
-	1	0
7	1	1
8	1	1



Process on Machine B

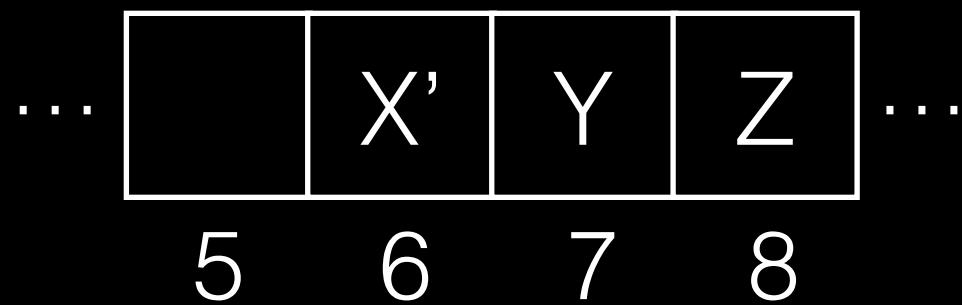
PFN	valid	present
-	0	-
23	1	1
22	1	1
-	1	0



B writes X' to 1st page

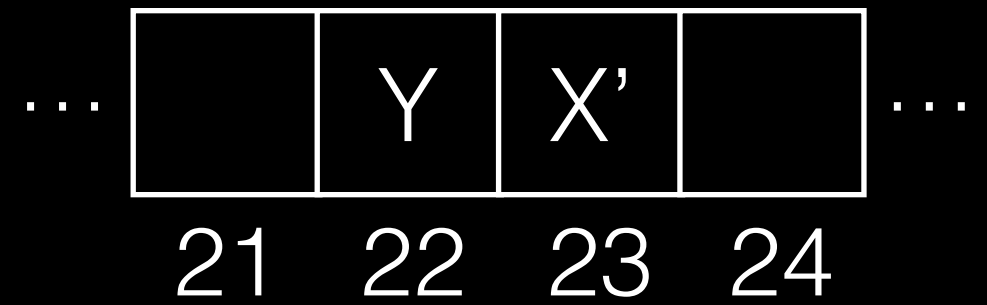
Process on Machine A

PFN	valid	present
-	0	-
6	1	1
7	1	1
8	1	1



Process on Machine B

PFN	valid	present
-	0	-
23	1	1
22	1	1
-	1	0



A reads 1st page

Virtual Memory Problems

What if a machine crashes?

- mapping disappears in other machines
- how to handle?

Performance?

- when to prefetch?
- loads/stores expected to be fast

DSM (distributed shared memory) not used today.

Global File System

Advantages

- file access is already expected to be slow
- use common API
- no need to modify applications (sorta true, flocks over NFS don't work)

Disadvantages

- doesn't always make sense, e.g., for video app

Overview

Raw messages

Reliable messages

OS abstractions

- virtual memory
- global file system

Programming-languages abstractions

- remote procedure call
-

RPC

Remote **P**rocedure **C**all.

What could be easier than calling a function?

Strategy: create **wrappers** so calling a function on another machine feels just like calling a local function.

This abstraction is very common in industry.

RPC

Machine A

```
int main(...) {  
  
}
```

Machine B

```
int foo(char *msg) {  
    ...  
}
```

RPC

Machine A

```
int main(...) {  
    int x = foo();  
}
```

Machine B

```
int foo(char *msg) {  
    ...  
}
```



Want main() on A to call foo() on B.

RPC

Machine A

```
int main(...) {  
    int x = foo();  
}
```

Machine B

```
int foo(char *msg) {  
    ...  
}
```

Want main() on A to call foo() on B.

RPC

Machine A

```
int main(...) {  
    int x = foo();  
}  
  
int foo(char *msg) {  
    send msg to B  
    recv msg from B  
}
```

Machine B

```
int foo(char *msg) {  
    ...  
}
```

Want main() on A to call foo() on B.

RPC

Machine A

```
int main(...) {  
    int x = foo();  
}  
  
int foo(char *msg) {  
    send msg to B  
    recv msg from B  
}
```

Machine B

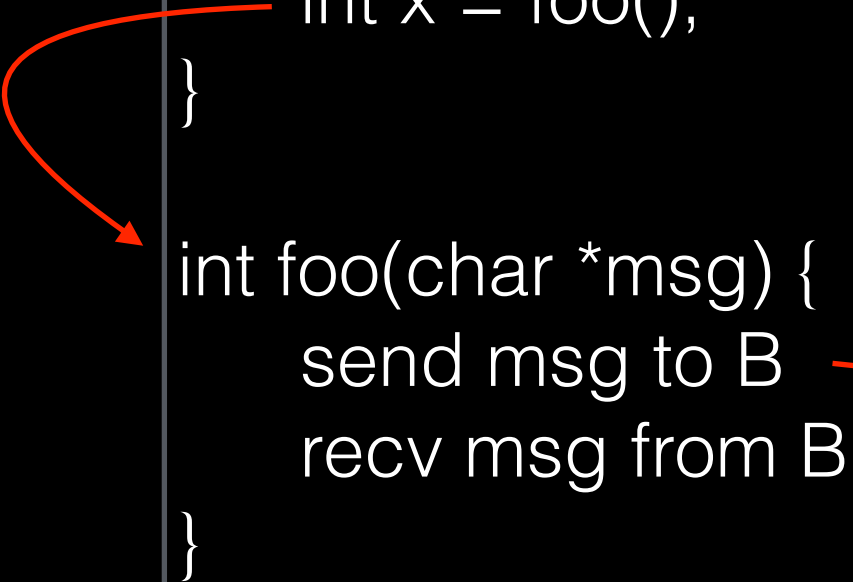
```
int foo(char *msg) {  
    ...  
}  
  
void foo_listener() {  
    while(1) {  
        recv, call foo  
    }  
}
```

Want main() on A to call foo() on B.

RPC

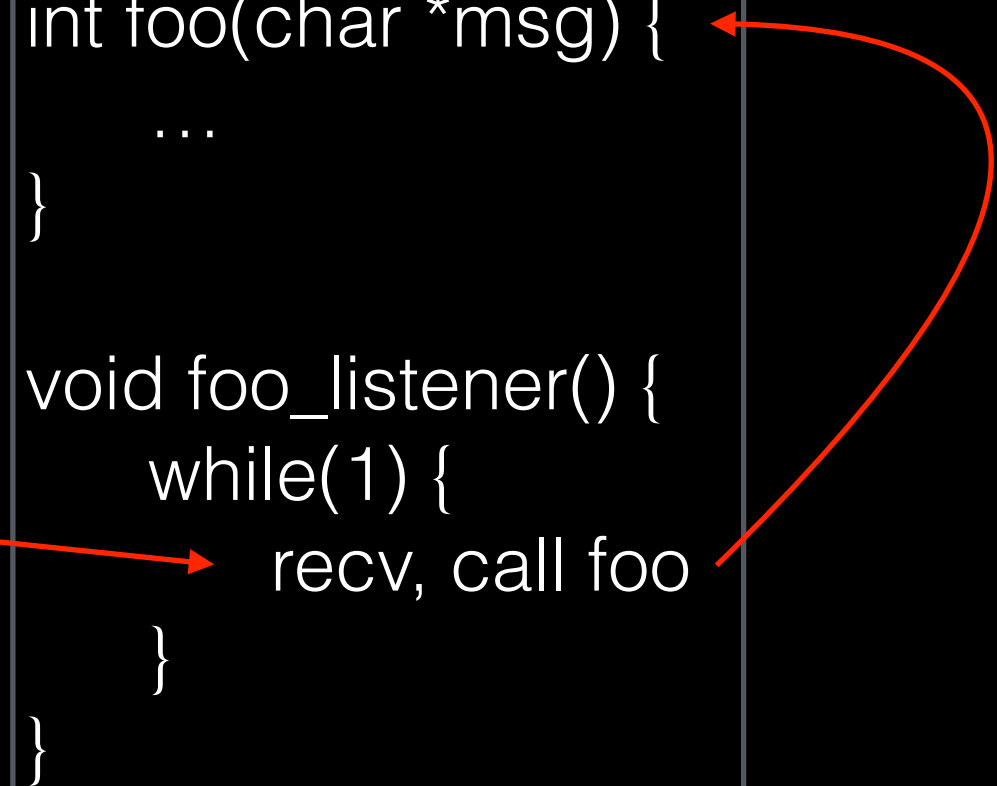
Machine A

```
int main(...) {  
    int x = foo();  
}  
  
int foo(char *msg) {  
    send msg to B  
    recv msg from B  
}
```



Machine B

```
int foo(char *msg) {  
    ...  
}  
  
void foo_listener() {  
    while(1) {  
        recv, call foo  
    }  
}
```



Actual calls.

RPC

Machine A

```
int main(...) {  
    int x = foo();  
}  
  
int foo(char *msg) {  
    send msg to B  
    recv msg from B  
}
```

Machine B

```
int foo(char *msg) {  
    ...  
}  
  
void foo_listener() {  
    while(1) {  
        recv, call foo  
    }  
}
```

What it feels like for programmer.

RPC

Machine A

```
int main(...) {  
    int x = foo();  
}
```

```
int foo(char *msg) {  
    send msg to B  
    recv msg from B  
}
```

client
wrapper

Machine B

```
int foo(char *msg) {  
    ...  
}
```

```
void foo_listener() {  
    while(1) {  
        recv, call foo  
    }  
}
```

server
wrapper

Wrappers.

RPC Tools

RPC packages help with this with two components.

(1) Stub generation

- create wrappers automatically

(2) Runtime library

- thread pool
- socket listeners call functions on server

RPC Tools

RPC packages help with this with two components.

(1) Stub generation

- create wrappers automatically

(2) Runtime library

- thread pool
- socket listeners call functions on server

Stub Generation

Many tools will automatically generate wrappers:

- rpcgen
- thrift
- protobufs

Programmer fills in generated stubs.

Wrapper Generation

Wrappers must do conversions:

- client arguments to **message**
- **message** to server arguments
- server return to **message**
- **message** to client return

Need uniform endianness (wrappers do this).

Conversion is called marshaling/unmarshaling, or serializing/deserializing.

Wrapper Generation: Pointers

Why are pointers problematic?

Wrapper Generation: Pointers

Why are pointers problematic?

The addr passed from the client will not be valid on the server.

Solutions?

Wrapper Generation: Pointers

Why are pointers problematic?

The addr passed from the client will not be valid on the server.

Solutions?

- smart RPC package: follow pointers
- distribute **generic data structs** with RPC package

RPC Tools

RPC packages help with this with two components.

(1) Stub generation

- create wrappers automatically

(2) Runtime library

- thread pool
- socket listeners call functions on server

RPC Tools

RPC packages help with this with two components.

(1) Stub generation

- create wrappers automatically

(2) Runtime library

- thread pool
- socket listeners call functions on server

Runtime Library

Design decisions:

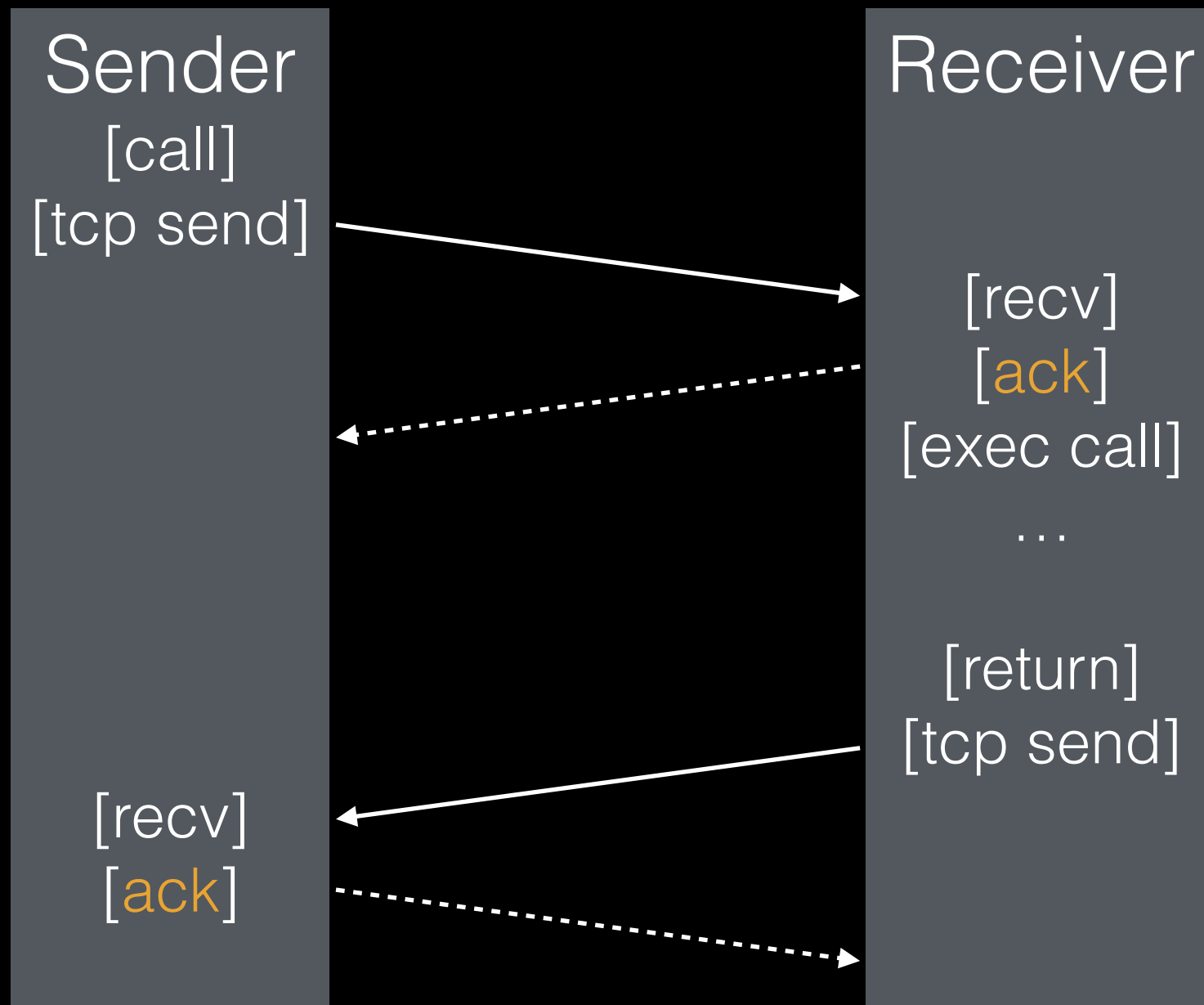
How to serve calls?

- usually with a **thread pool**

What **underlying protocol** to use?

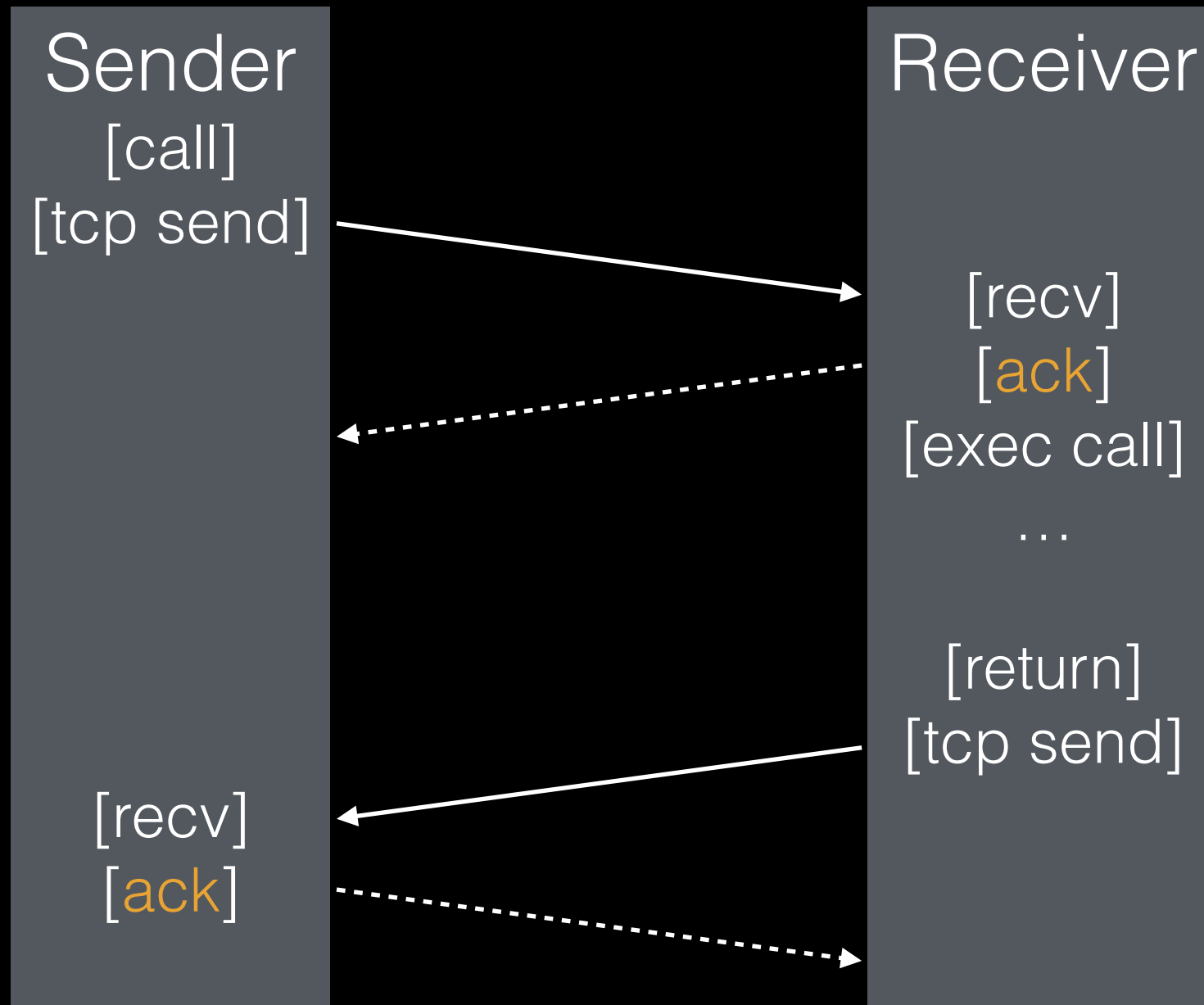
- usually UDP

RPC over TCP



RPC over TCP

Why wasteful?



RPC over UDP

Strategy: use function return as implicit ACK.

Piggybacking technique.

What if function takes a long time?

- then send a separate ACK

Conclusion

Many communication abstraction possible:

Raw messages (UDP)

Reliable messages (TCP)

Virtual memory (OS)

Global file system (OS)

Function calls (RPC)

Announcements

Thursday **discussion**

- review midterm 2.

Office hours

- today at 1pm, in office