Demystifying Non-blocking and Asynchronous I/O

PyCon 2010, Hyatt Regency Atlanta

Peter Portante

What's the Problem?

* I am not able to meet my I/O demands using the code I have written
* I have chosen to use an elegant framework that is not helping
* My application is coded simply in a sequential, understandable manor, what should I do?

Somebody Told Me...

* to just use asynchronous I/O
* "You ought to use non-blocking I/O"
* "Why don't you just switch to using ... , that will solve your scaling problems"
    * asyncore, Twisted, Tornado, PyEv

But, but, ...

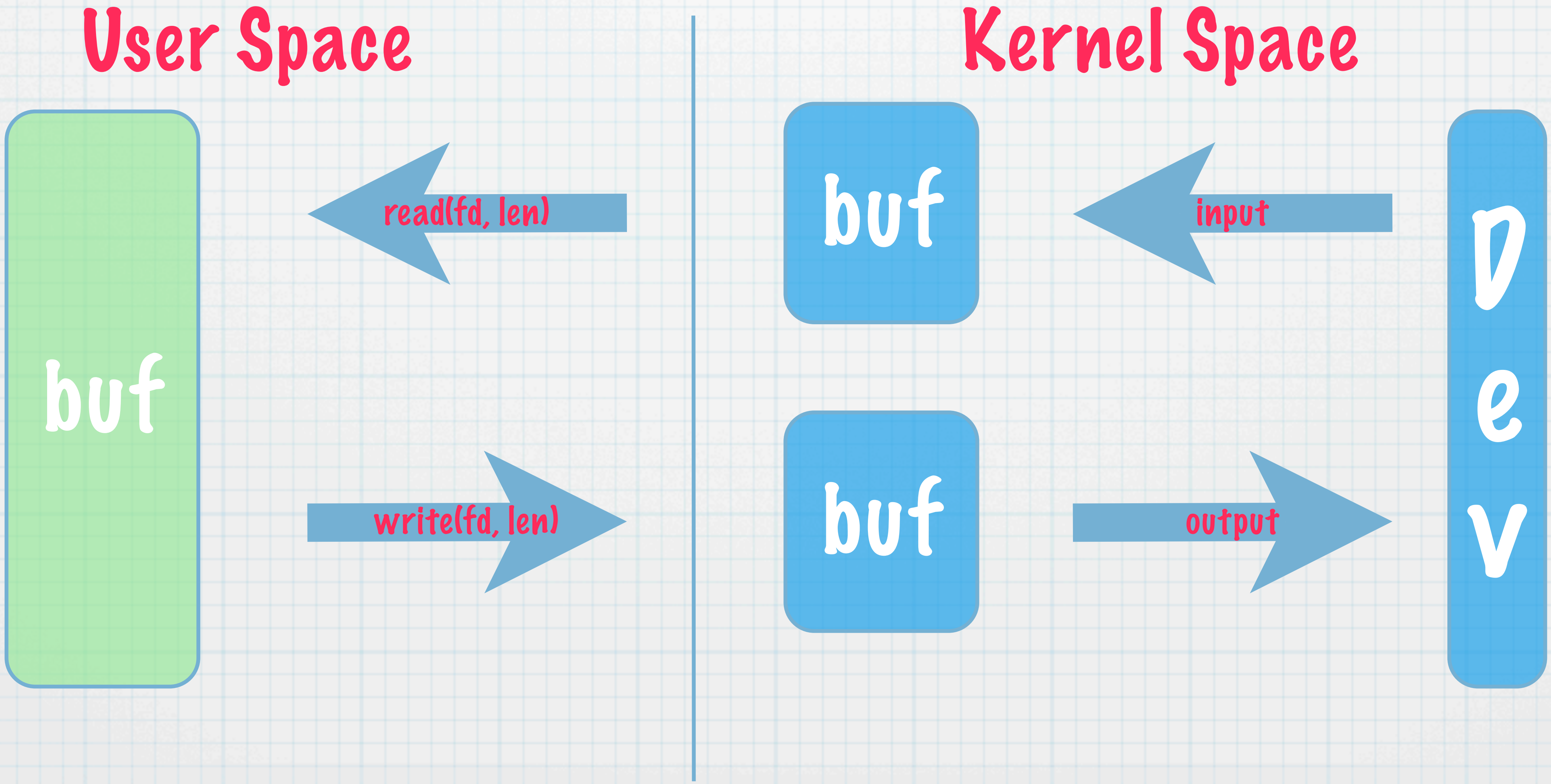
* What does non-blocking I/O mean?
* What is non-blocking compared to asynchronous I/O?
* What is involved when using that kind of I/O in my library or application?

It's All Ball Bearings

* Let's first look at how I/O is performed under Linux
  * Very high level, buffered only
* Agree on some definitions
* Talk about I/O Multiplexing and Event driven programming
* Talk about what non-blocking vs. asynchronous means

Buffered I/O - 40,000'

* File descriptor has memory buffers for reading and writing

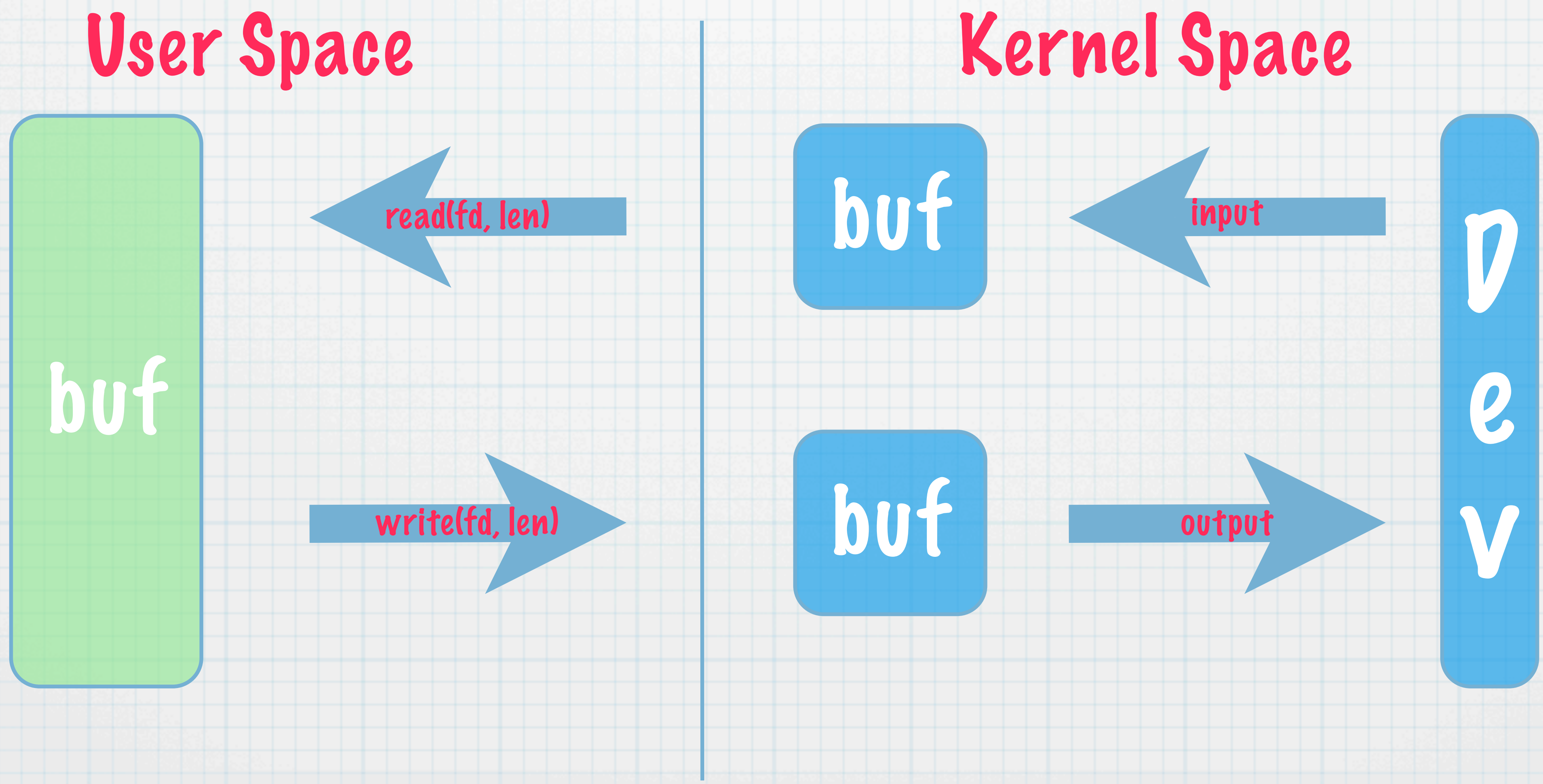


Blocking I/O
(Synchronous)

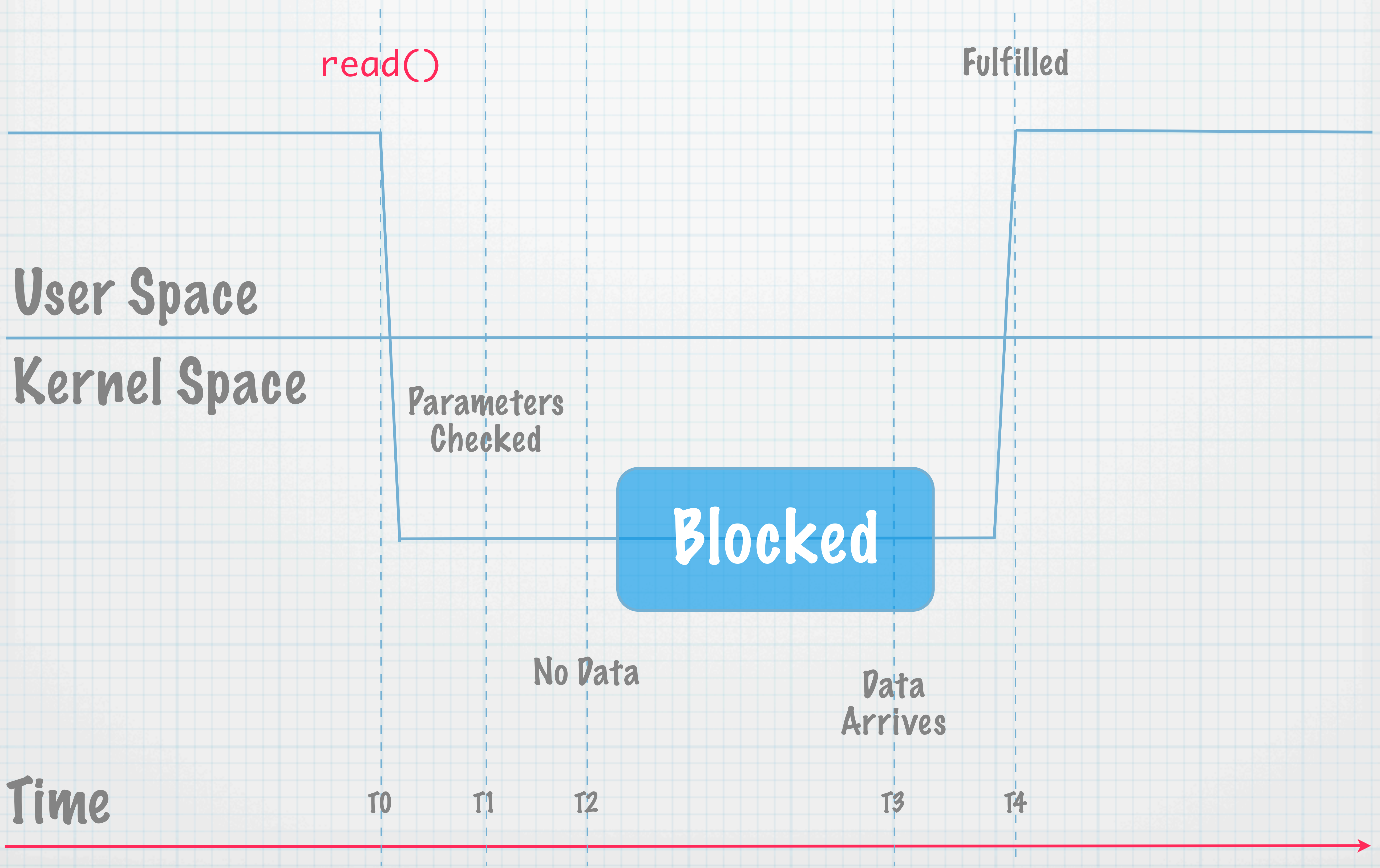
- "An I/O operation that **may** itself cause the requesting thread of execution to be blocked from further use of the processor."
- This implies that the thread of execution and the I/O operation run sequentially

Blocking I/O
(cont'd)

* read(fd, len) - blocks < len data in kernel buffer
* write(fd, len) - blocks < len empty space in kernel buffer



Blocking I/O Timeline



Blocking I/O Example

```
import os
msg = ""
while True:
    if msg == "exit":
        os.write(1, "Goodbye\n")
        break
    elif msg:
        os.write(1, "Hello [%s]\n" % msg)
        msg = ""
    os.write(1, ":")
    while True:
        val = os.read(0, 4)
        if val[-1] == '\n':
            msg += val[:-1]
            break
        msg += val
```

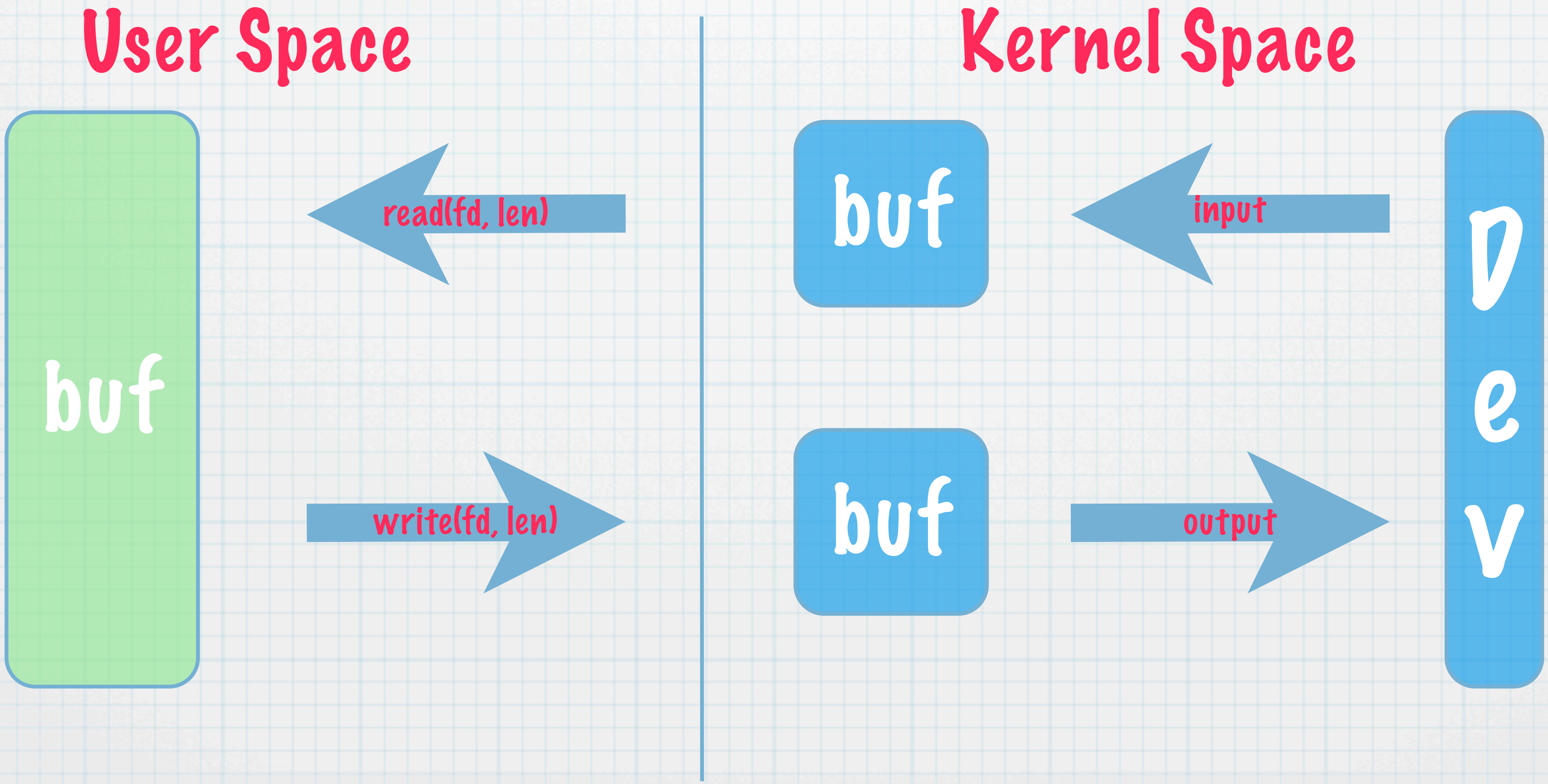
Non-blocking I/O

(Still **Synchronous!**)

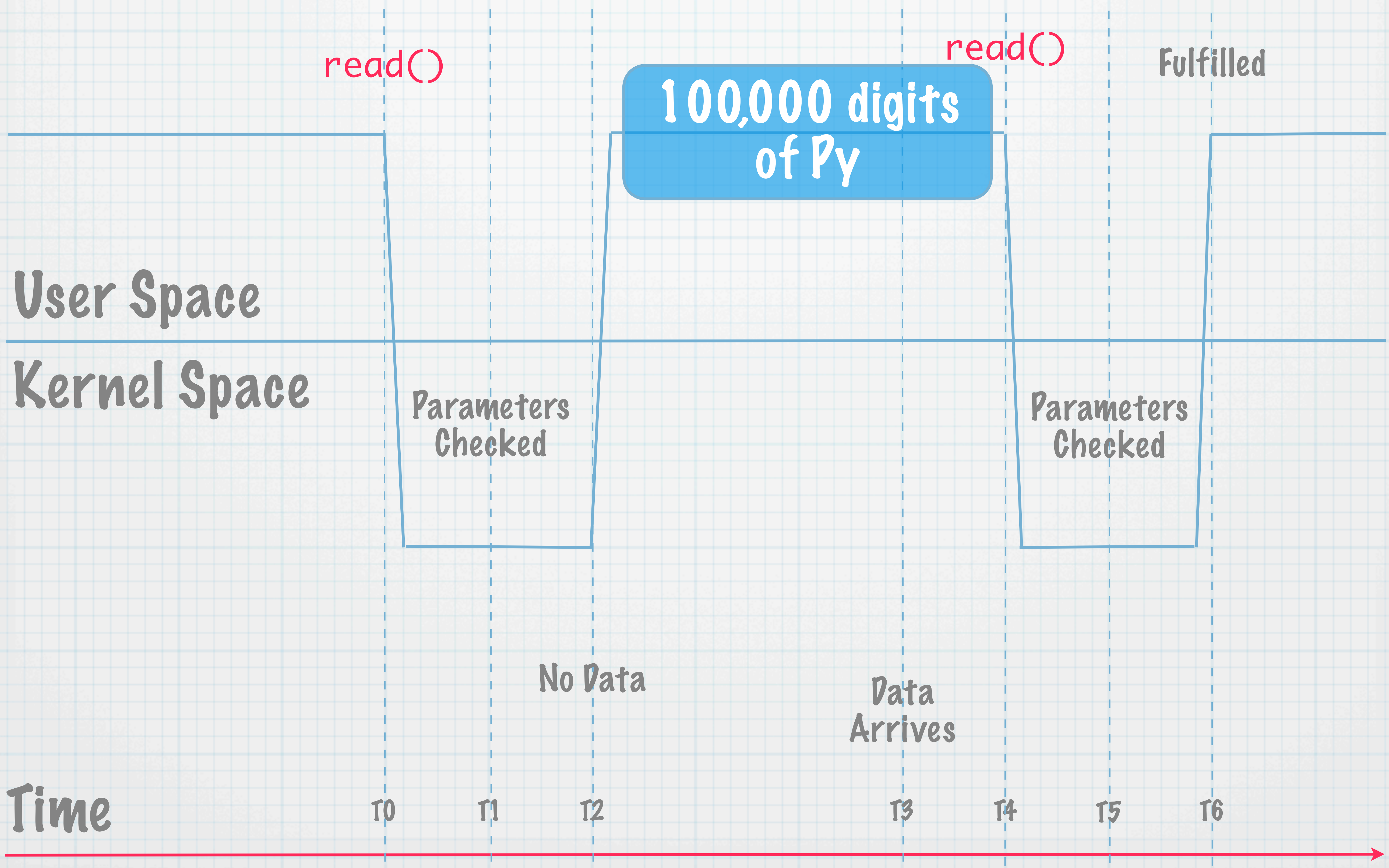
* An I/O operation **that is only initiated if** it does not of itself cause the thread of execution requesting the I/O to be blocked from further use of the processor
* Implies that the thread of execution and the I/O operation still run sequentially
* Implies that the thread of execution will be notified when an I/O operation is not initiated, or partially initiated

Non-blocking I/O
(cont'd)

* An attribute of the FD which changes its behavior
* When enabled, read(fd, len)/write(fd, len) returns EWOULDBLOCK if it cannot read/write any data, otherwise the count of bytes



Non-blocking I/O Timeline



Non-blocking I/O 1st Example

```
import os, fcntl
ofl = fcntl.fcntl(0, fcntl.F_GETFL)
fcntl.fcntl(0, fcntl.F_SETFL, ofl | os.O_NONBLOCK)
msg = ""
while True:
    if msg == "exit":
        os.write(1, "Goodbye\n"); break
    elif msg:
        os.write(1, "Hello [%s]\n" % msg); msg = ""
    os.write(1, ":")
    import time; time.sleep(1)
    while True:
        val = os.read(0, 4)
        if val[-1] == '\n':
            msg += val[:-1]
            break
        msg += val
```

Non-blocking I/O 2nd Example

```
import os, fcntl
ofl = fcntl.fcntl(0, fcntl.F_GETFL)
fcntl.fcntl(0, fcntl.F_SETFL, ofl | os.O_NONBLOCK)
try:
    msg = ""
    while True:
        if msg == "exit":
            os.write(1, "Goodbye\n")
            break
        elif msg:
            os.write(1, "Hello [%s]\n" % msg)
            msg = ""
        os.write(1, ":")
        while True:
            val = nread(fd=0, length=4)
            if val[-1] == '\n':
                msg += val[:-1]
                break
            msg += val
finally:
    fcntl.fcntl(0, fcntl.F_SETFL, ofl)
```

2nd Example (cont'd)

```
def nread(fd=None, length=None):
    import time, errno
    val = None
    while val is None:
        try:
            val = os.read(fd, length)
        except OSError, e:
            if e.errno != errno.EWOULDBLOCK:
                raise
        if val is None:
            time.sleep(1)
    return val
```

Wait ... that is Ugly!

* Yes, a non-blocking FD is not the whole story
* Let's talk
  * I/O Multiplexing
  * Event Driven I/O Models

I/O Multiplexing

* The kernel offers `poll()`
* You ask for which FDs are ready for I/O
* Returns a list flagged w/ read/write
* If none ready, can ask to:
  * wait indefinitely
  * wait for a period of time
  * return immediately

Event Driven I/O Models

* The readiness of an FD for I/O is often referred to as an **event**
* Libraries and frameworks supporting event driven I/O typically allow you to register a callback for a particular **event** on an FD

I/O Multiplexor Example

```
class _IoManager(object):
    def __init__(self):
        self.fd_flags = {}
        self.fd_ctx = {}
        self.poll = select.poll()
    def manage(self):
        ...
    def register(self, fd=None, op=None, ctx=None):
        ...
    def unregister(self, fd=None, op=None):
        ...
iomanager = _IoManager()
```

I/O Multiplexor Example

* Context object can be anything that has a `ready` method accepting two parameters
    * A file descriptor
    * Flag for what the FD is ready for

I/O Multiplexor Example

```
def register(self, fd=None, op=None, ctx=None):
    if fd is not None:
        ofl = fcntl.fcntl(fd, fcntl.F_GETFL)
        self.fd_flags[fd] = ofl
        fcntl.fcntl(fd, fcntl.F_SETFL,
                    ofl | os.O_NONBLOCK)
        self.fd_callbacks[fd] =
                    {op:{'fd':fd,'ctx':ctx}}
        if op == 'read':
            pollop = select.POLLIN
        else:
            pollop = select.POLLOUT
        self.poll.register(fd, pollop)
```

I/O Multiplexor Example

```
def unregister(self, fd=None, op=None):
    if fd is None:
        return
    del self.fd_callbacks[fd][op]
    if self.fd_callbacks[fd]:
        return
    del self.fd_callbacks[fd]
    self.poll.unregister(fd)
    if fd in self.fd_flags:
        ofl = self.fd_flags[fd]
        del self.fd_flags[fd]
        fcntl.fcntl(fd, fcntl.F_SETFL, ofl)
```

I/O Multiplexor Example

```
def manage(self):
    try:
        while self.fd_callbacks:
            cbs = []; fds = self.poll.poll()
            for fd, eventmask in fds:
                if eventmask & select.POLLIN:
                    cb = self.fd_callbacks[fd]['read']
                    cbs.append(('read', cb))
                if eventmask & select.POLLOUT:
                    cb = self.fd_callbacks[fd]['write']
                    cbs.append(('write', cb))
            for op, cb in cbs:
                cb['ctx'].ready(cb['fd'], op)
    finally:
        for fd, ofl in self.fd_flags.items():
            fcntl.fcntl(fd, fcntl.F_SETFL, ofl)
```

I/O Multiplexor Example

```
from nonblockio import iomanager; import os

class MyFD(object):
    def __init__(self, fd):
        self._fd = fd
        self._readBuf = ""; self._writeBuf = ""
    def ready(self, fd, op):
        if op == 'read':
            self._readBuf = os.read(fd, 20)
        elif op == 'write':
            cnt = os.write(fd, self._writeBuf, 20)
            self._writeBuf = self._writeBuf[cnt:]

iomanager.register(0, 'read', MyFD(0))
iomanager.register(1, 'write', MyFD(1))
iomanager.manage()
```

Here's the Rub

* I/O multiplexing still means it is synchronous I/O
* Once the kernel's buffers fill up, not much is going to happen until a `read()` or a `write()` system call is made

So What is Asynchronous I/O then?

* The **cause** of an event is asynchronous to the application
* The **handling** of an event is performed synchronously
* That means the act of reading and writing data from/to the kernel still occurs synchronously

How 'bout them Apples?

* So if your thread of execution:
  * is involuntarily context switched
  * page faults
  * blocks on a mutex or semaphore
  * goes compute bound
* All I/O stops being issued until control is restored to the I/O polling event loop

So Why is it “Better”

* The primary reason is memory usage
  * Blocking I/O requires one thread of execution for each FD
    * That has a “large” execution stack
    * Kernel has a number of data structures need to manage threads of execution
  * Context switching threads of execution means lots of memory references
  * Contrast that to an object describing an FD
*

I/O Multiplexor Context

```
class MyFD(object):

    def __init__(self, fd):
        self._fd = fd
        self._readBuf = ""
        self._writeBuf = ""

    def ready(self, fd, op):
        if op == 'read':
            self._readBuf = os.read(fd, 20)
        elif op == 'write':
            cnt = os.write(fd, self._writeBuf, 20)
            self._writeBuf = self._writeBuf[cnt:]
```

So Why Else is it "Better"?

* You can drive lots of I/O
  * Without involving threads
    * Avoids the effects of the GIL
  * Without using multiple processes
    * Don't have to manage shared memory

Non-blocking I/O Services

* C implementations w/ Python wrappers
  * libev (http://software.schmorp.de/pkg/libev.html)
    * pyev (http://code.google.com/p/pyev/)
  * libevent (http://www.monkey.org/~provos/libevent/)
    * pyevent (not updated since 2007) (http://code.google.com/p/pyevent/)

Non-blocking I/O Frameworks/Libraries

* **Tornado** (http://www.tornadoweb.org/)
* **Twisted** (http://twistedmatrix.com/trac/)
* **asyncore** (http://docs.python.org/library/asyncore.html)

Lemme Sum Up

* Non-blocking I/O involves an I/O multiplexor to create an event driven mechanism
* I/O readiness events
  * occur asynchronously
  * handled synchronously
* Benefits are increased scalability
* Costs are complexity and the use of an event driven model

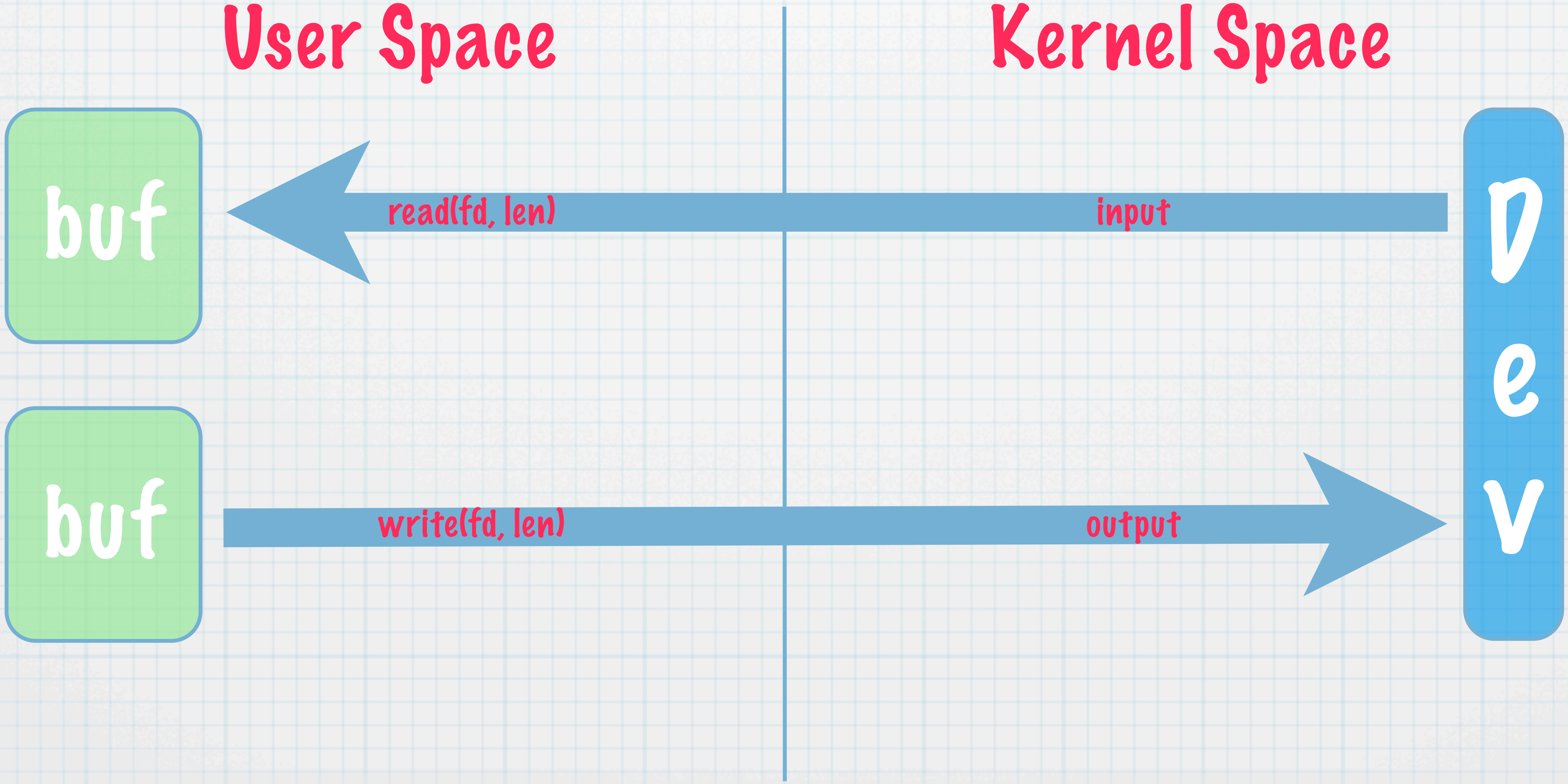
Questions?

Books/Web

* "Programming with POSIX Threads" by David R. Butenhof
* "Unix Network Programming : Networking APIs: Sockets and XTI" by W. Richard Stevens
* "Advanced Programming in the UNIX Environment" by W. Richard Stevens w/ Stephen A. Rago
* "The Design and Implementation of the FreeBSD Operating System" by Marshall Kirk McKusick and George V. Neville-Neil
* Dan Kegel's "The C10K problem", http://www.kegel.com/c10k.html

Direct I/O - 40,000'

* Data is written directly into user's buffer for reads, taken directly from user's buffer for writes



Asynchronous I/O Timeline