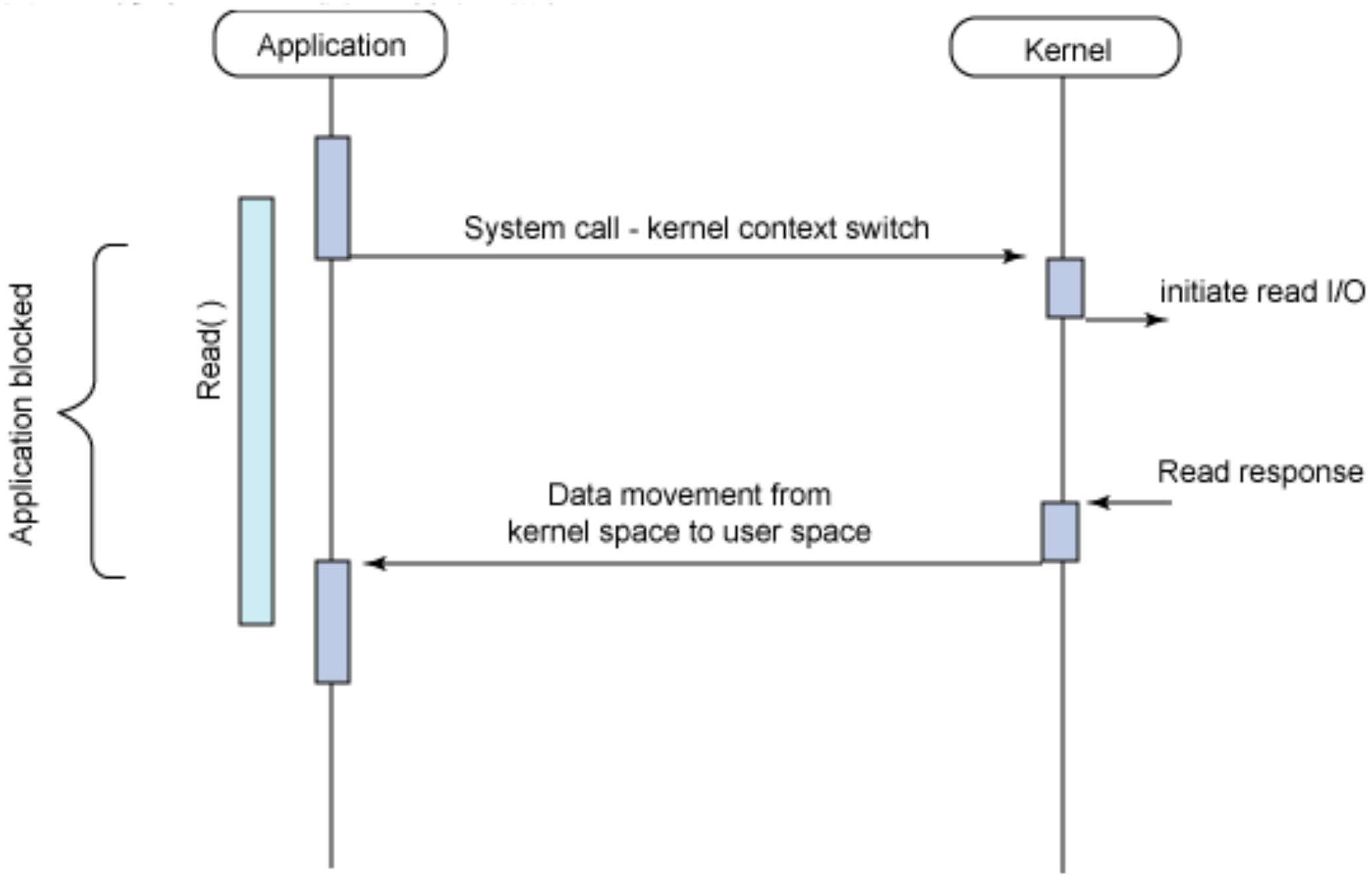
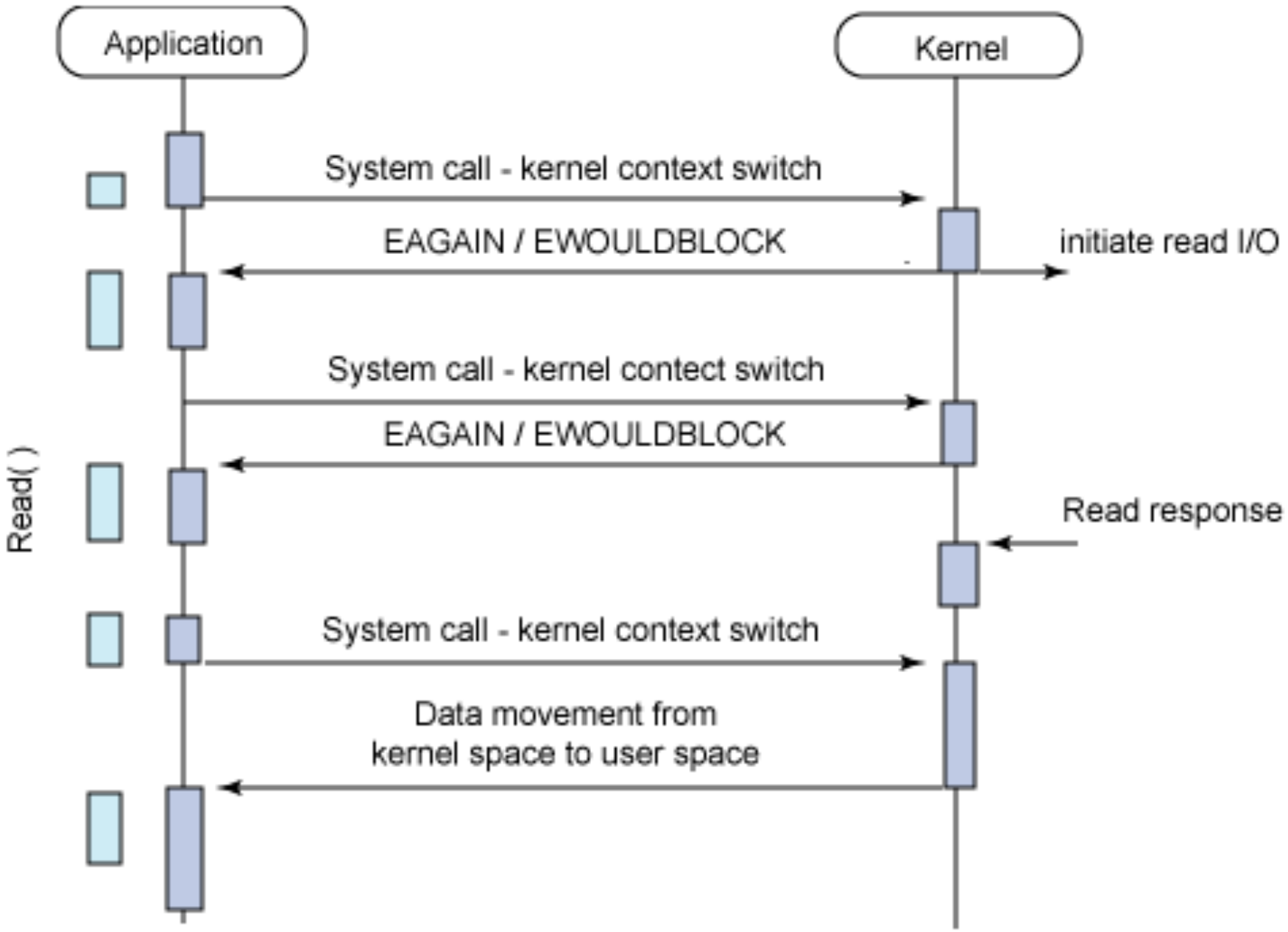
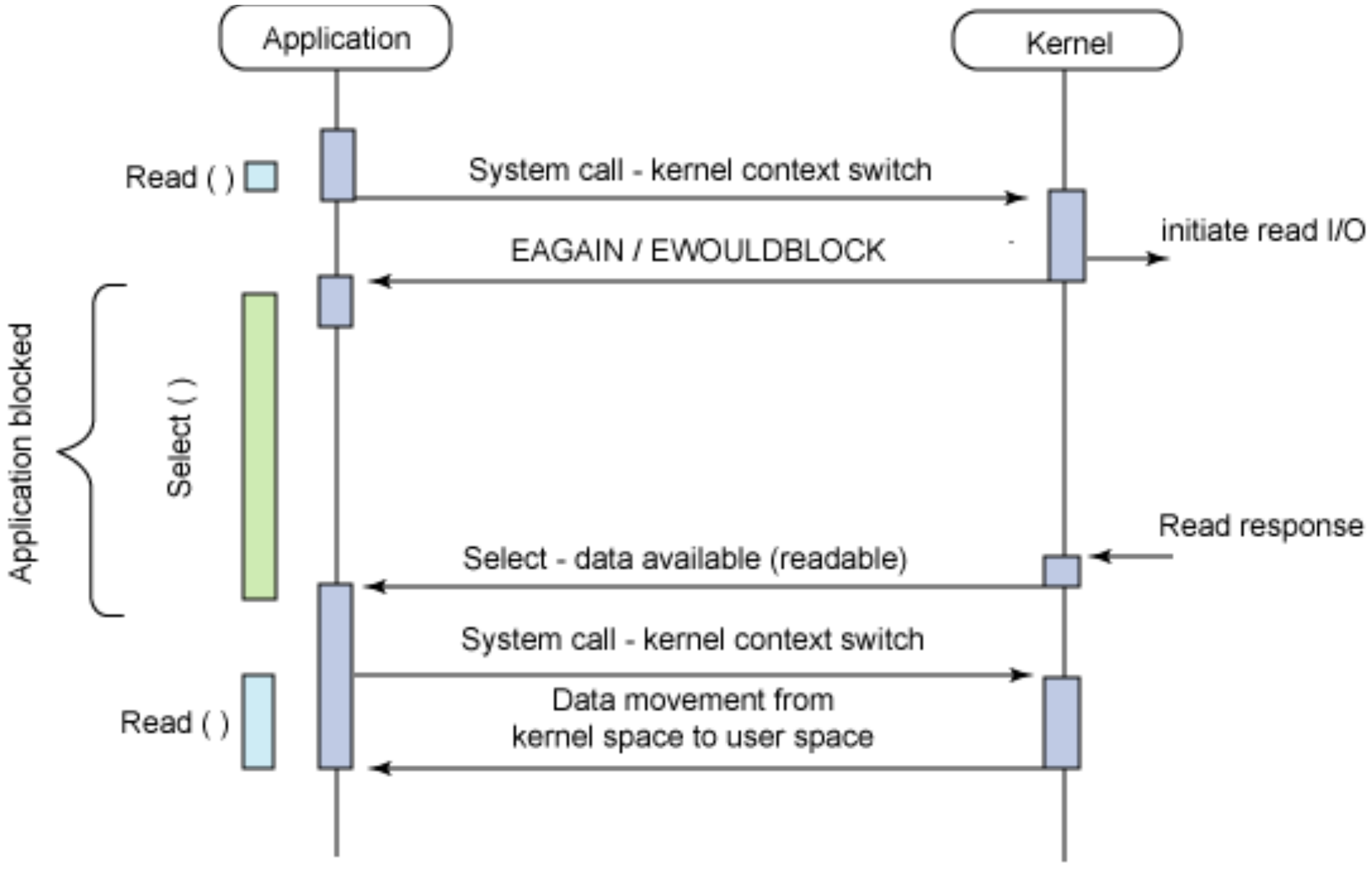
Don’t Believe me!!!

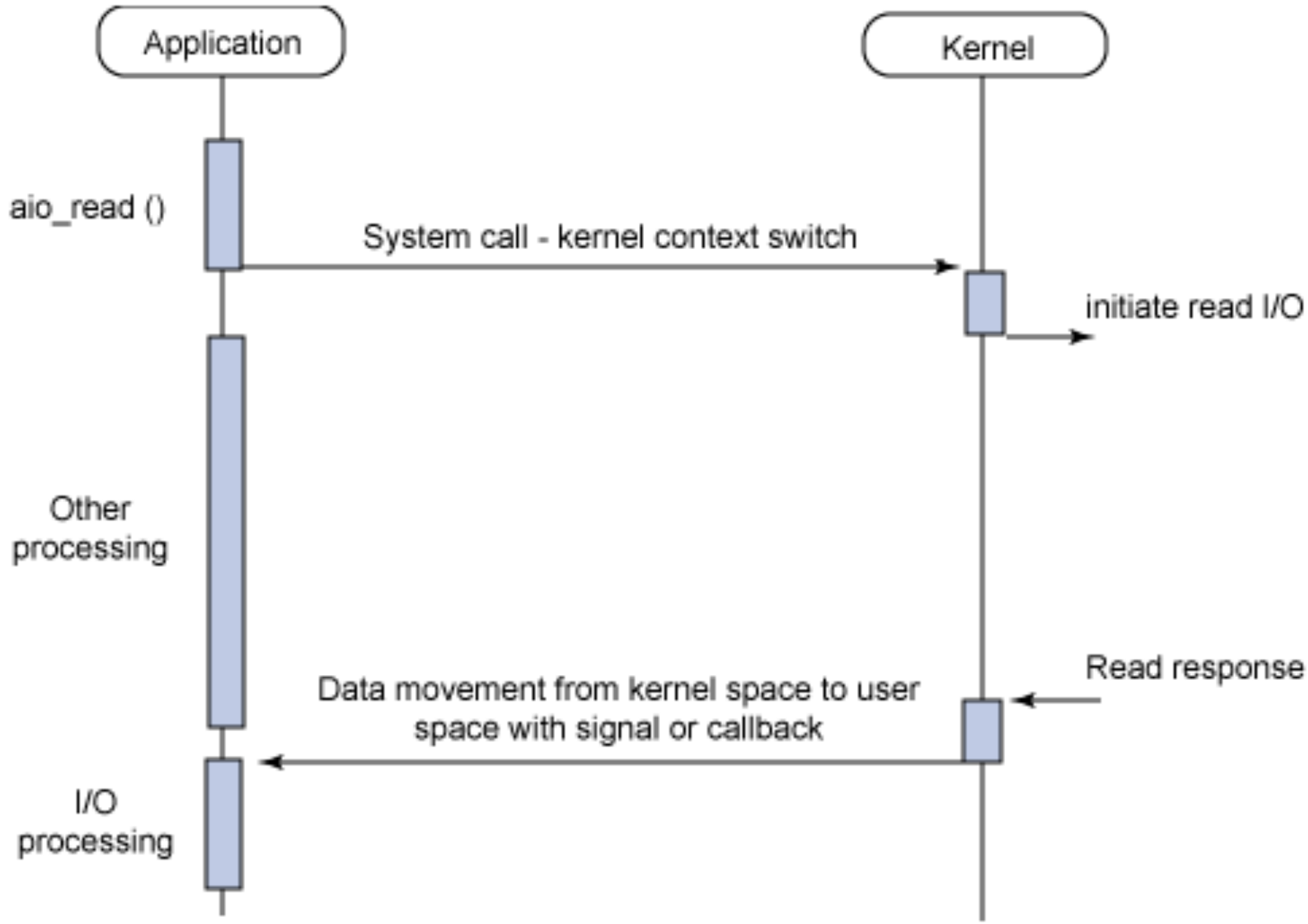
- blocking I/O
- nonblocking I/O
- I/O multiplexing
- signal driven I/O
- asynchronous I/O

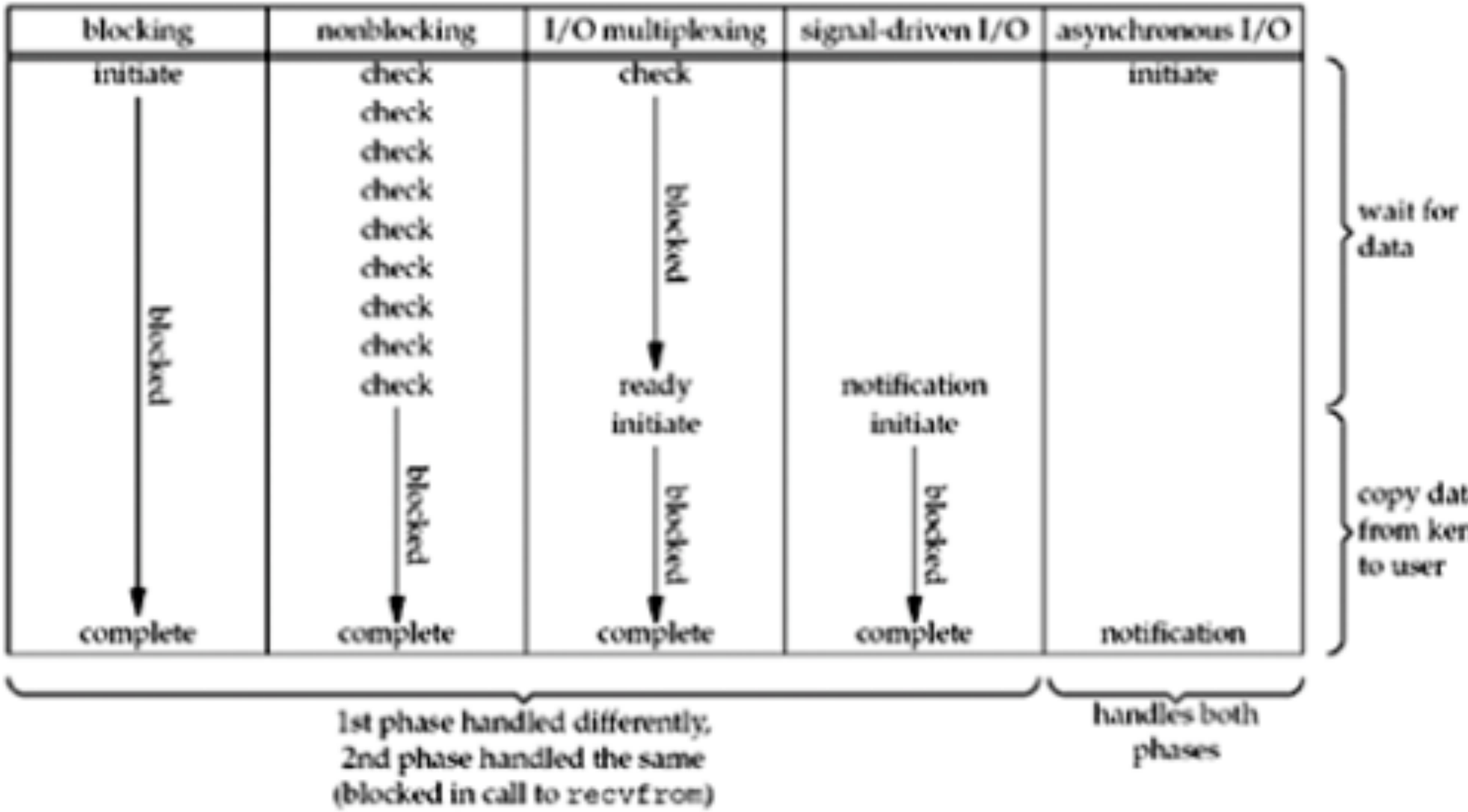
- Waiting for the data to be ready
- Copying the data from the kernel to the process


Application
Kernel
Application blocked
Read( )
System call - kernel context switch
initiate read I/O
Read response
Data movement from
kernel space to user space


Application
Kernel
System call - kernel context switch
EAGAIN / EWOULDBLOCK
initiate read I/O
System call - kernel contect switch
EAGAIN / EWOULDBLOCK
Read( )
Read response
System call - kernel context switch
Data movement from
kernel space to user space


Application
Kernel
Read ( )
System call - kernel context switch
initiate read I/O
EAGAIN / EWOULDBLOCK
Application blocked
Select ( )
Read response
Select - data available (readable)
System call - kernel context switch
Read ( )
Data movement from
kernel space to user space


Application
Kernel
aio_read ()
System call - kernel context switch
initiate read I/O
Other
processing
Read response
Data movement from kernel space to user
space with signal or callback
I/O
processing


blocking
nonblocking
I/O multiplexing
signal-driven I/O
asynchronous I/O
initiate
check
check
check
check
check
check
check
check
check
check
blocked
ready
initiate
blocked
notification
initiate
blocked
initiate
blocked
complete
complete
complete
complete
notification
wait for data
copy data from kernel to user
1st phase handled differently,
2nd phase handled the same
(blocked in call to recvfrom)
handles both phases

- A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes.
- An asynchronous I/O operation does not cause the requesting process to be blocked.

在处理 IO 的时候，阻塞和非阻塞都是同步 IO。

只有使用了特殊的 API 才是异步 IO。

同步	IO multiplexing (select/poll/epoll)	
	阻塞	非阻塞

异步	Linux	Windows	.NET
	AIO	IOCP	BeginInvoke/EndInvoke

I/O completion ports provide an efficient threading model for processing multiple asynchronous I/O requests on a multiprocessor system. When a process creates an I/O completion port, the system creates an associated queue object for requests whose sole purpose is to service these requests. Processes that handle many concurrent asynchronous I/O requests can do so more quickly and efficiently by using I/O completion ports in conjunction with **a pre-allocated thread pool** than by creating threads at the time they receive an I/O request.

http://www.slideshare.net/sm9kr/iocp-vs-epoll-perfor

阻塞和非阻塞，描述的是一种状态，同步与非同步描述的是行为方式

单进程

多进程

多线程

http://www.zhihu.com/question/20114168

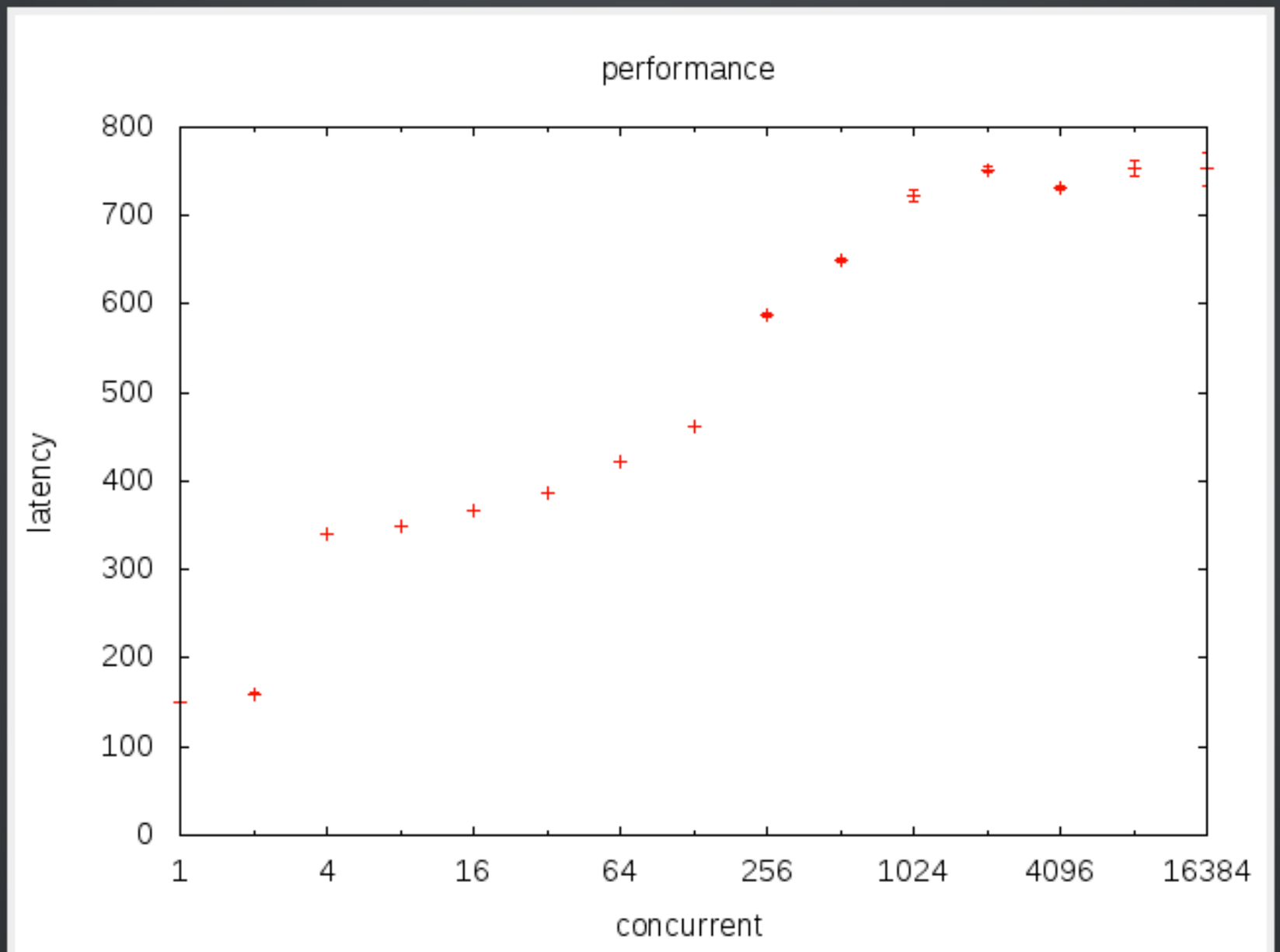
多线程的问题

- 内存，看到资料说，一个线程栈会消耗8M内存(linux默认值，ulimit可以看到)，512个线程栈就会消耗4G内存，而10K个线程就是80G。（http://unix.stackexchange.com/questions/145557/how-does-stack-allocation-work-in-linux）

- 陷入内核，线程模型主要通过陷入切换上下文，因此陷入开销大。（都要陷入内核）

开销

yield每次耗费的时间随活跃线程数变化曲线



http://www.ibm.com/developerworks/cn/linux/l-cn-scheduler/

- 就绪事件通知(Reactor)

- 异步 I/O(Proactor)

- select
- poll

epoll

网卡设备对应一个中断号, 当网卡收到网络端的消息的时候会向CPU发起中断请求, 然后CPU处理该请求. 通过驱动程序 进而操作系统得到通知, 系统然后通知epoll, epoll通知用户代码.

在内核的最底层是中断 类似系统回调的机制 不是轮询,

协程

COBOL编译器

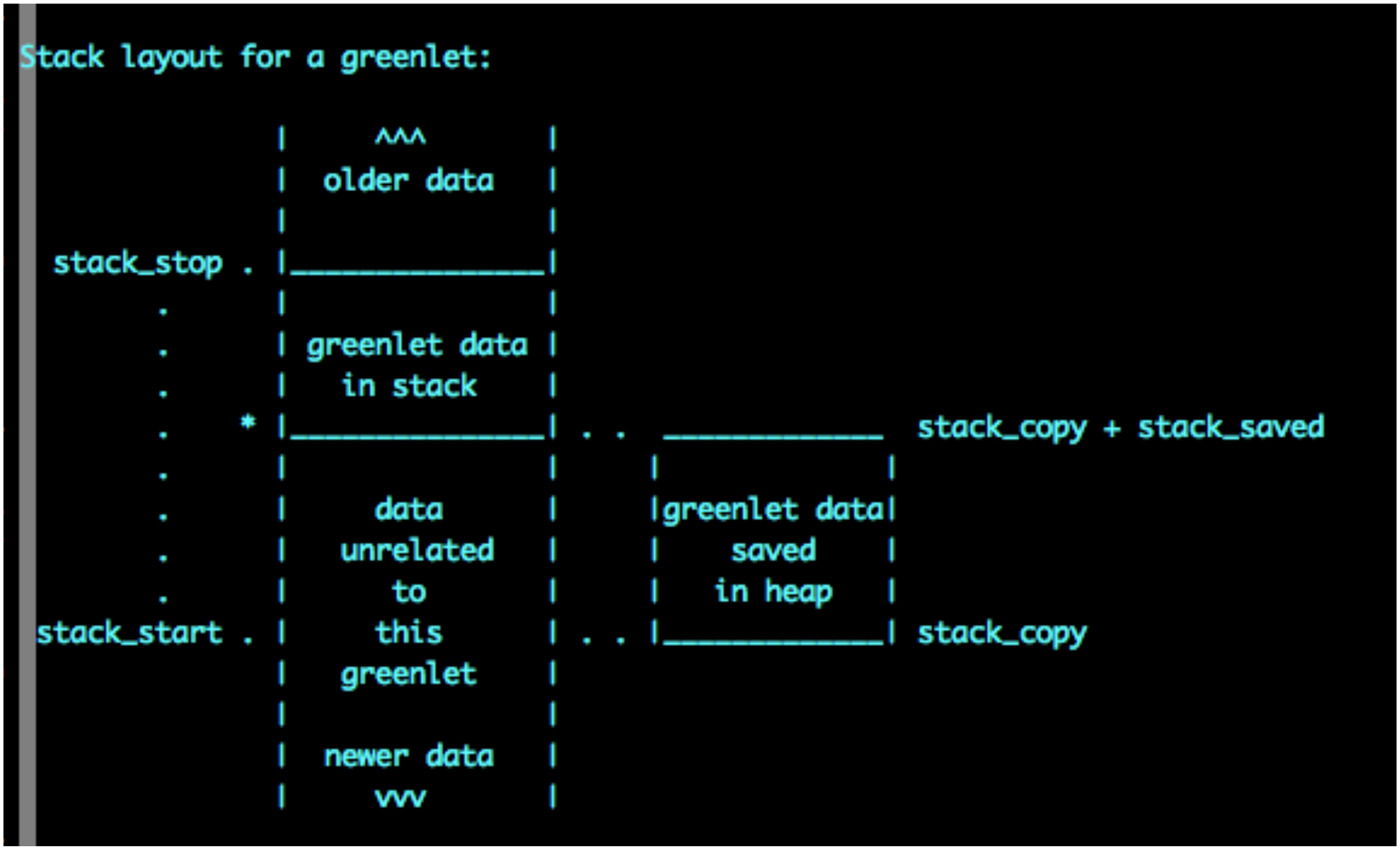
自顶向下

协同到抢占

`StackContext` allows applications to maintain threadlocal-like state
that follows execution as it moves to other execution contexts.

The motivating examples are to eliminate the need for explicit
``async_callback`` wrappers (as in `tornado.web.RequestHandler`), and to
allow some additional context to be kept for logging.

This is slightly magic, but it's an extension of the idea that an
exception handler is a kind of stack-local state and when that stack
is suspended and resumed in a new context that state needs to be
preserved. `StackContext` shifts the burden of restoring that state
from each call site (e.g. wrapping each `.AsyncHTTPClient` callback
in ``async_callback``) to the mechanisms that transfer control from
one context to another (e.g. `.AsyncHTTPClient` itself, `.IOLoop`,
thread pools, etc).


Stack layout for a greenlet:
^^^
older data
stack_stop
greenlet data
in stack
stack_copy + stack_saved
data
unrelated
to
this
greenlet
greenlet data
saved
in heap
stack_start
stack_copy
newer data
vvv

greenlet是通过stack_stop,stack_start来保存其stack的栈底和栈顶的,如果出现将要执行的greenlet的stack_stop和目前栈中的greenlet重叠的情况,就要把这些重叠的greenlet的栈中数据临时保存到heap中.保存的位置通过stack_copy和stack_saved来记录,以便恢复的时候从heap中拷贝回栈中stack_stop和stack_start的位置.不然就会出现其栈数据会被破坏的情况.所以应用程序创建的这些greenlet就是通过不断的拷贝数据到heap中或者从heap中拷贝到栈中来实现并发的

- IO model
- 高并发服务的发展流程和原因

参考资料

- http://shell909090.org/blog/2014/11/%E4%B8%8A%E4%B8%8B%E6%96%87%E5%88%87%E6%8D%A2%E6%8A%80%E6%9C%AF/
- http://www.zhihu.com/question/21896633/answer/19665381
- http://www.zhihu.com/question/19732473/answer/20851256
- https://code.google.com/p/libhjw/wiki/notes_on_greenlet
- http://boolan.com/lecture/1000001045
- http://www.ibm.com/developerworks/cn/linux/l-cn-scheduler/
- https://gitcafe.com/shell909090/context
- https://www.ibm.com/developerworks/cn/linux/l-async/
- http://blog.youxu.info/2014/12/04/coroutine/
- http://msdn.microsoft.com/en-us/library/windows/desktop/aa365198(v=vs.85).aspx