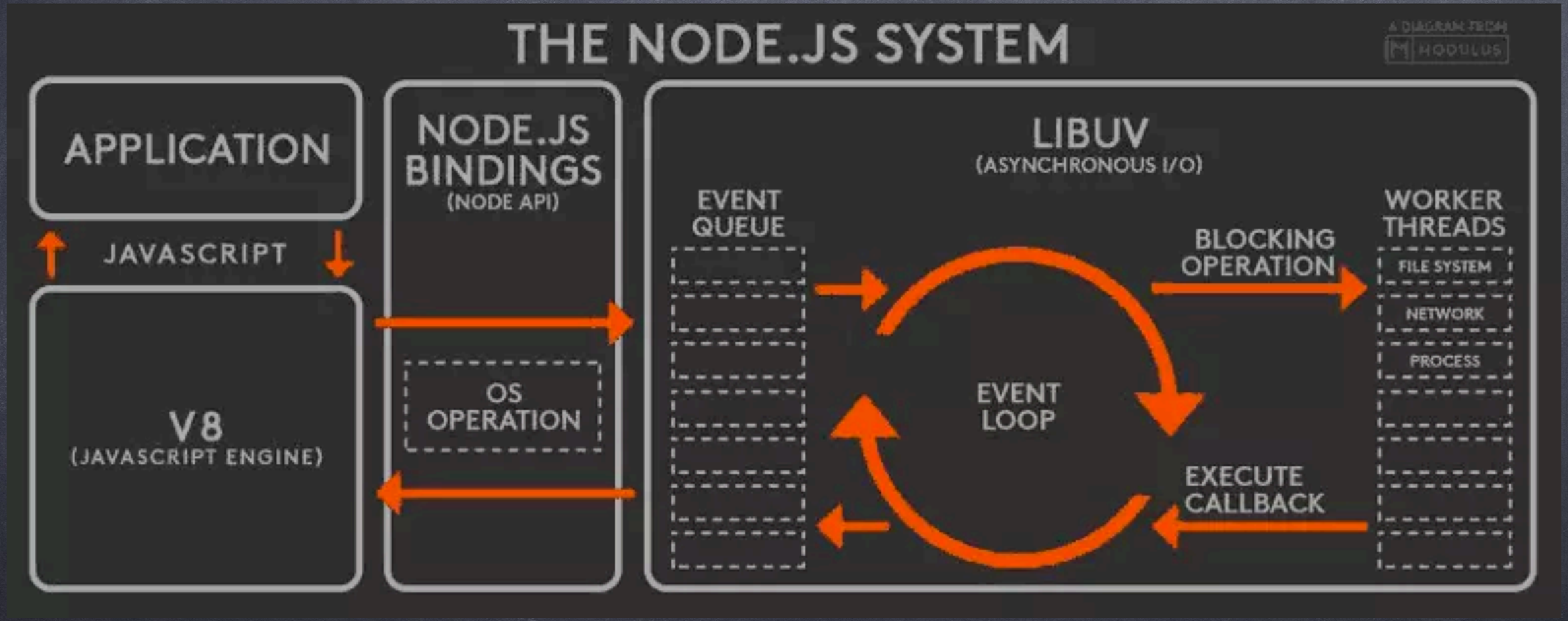
V8引擎与Linux异步机制

京程一灯
http://www.yidengxuetang.com

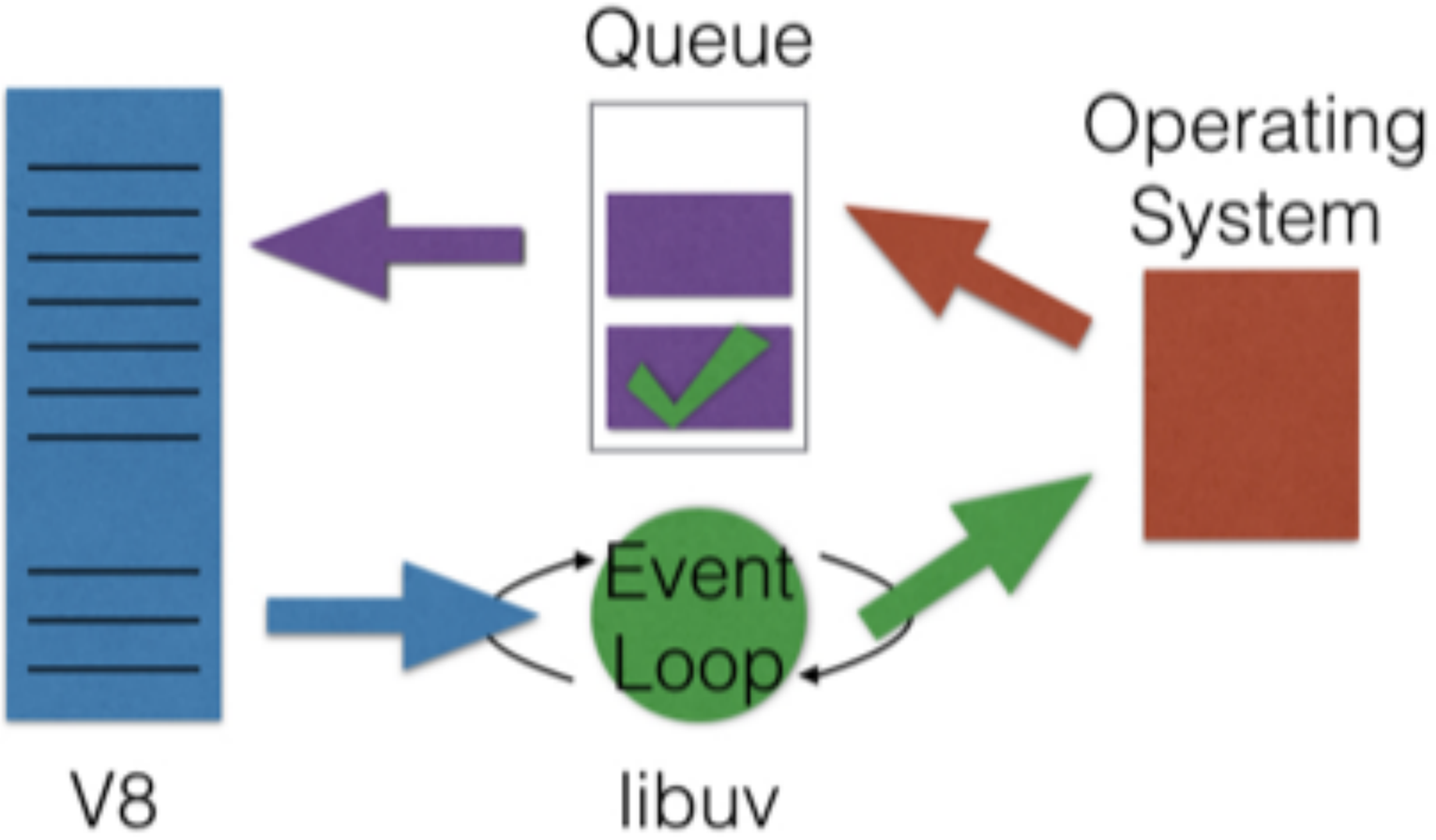
V8引擎与Linux异步机制

- Node.js与V8引擎结构
- linux底层异步通知机制
- Linux的 I/O 模型
- epoll机制
- iocp机制
- libuv库

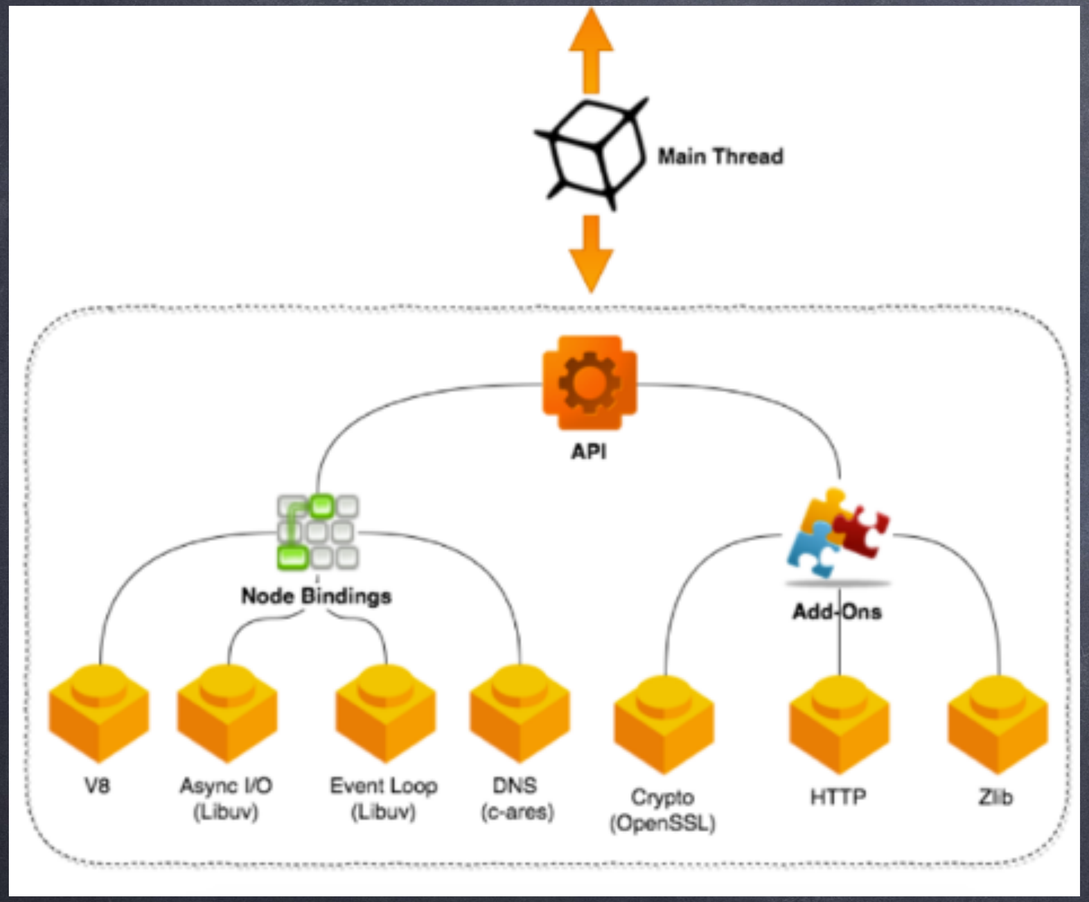
Node.js 系统体系


THE NODE.JS SYSTEM
A DIAGRAM FROM
MODULUS
APPLICATION
JAVASCRIPT
V8
(JAVASCRIPT ENGINE)
NODE.JS
BINDINGS
(NODE API)
OS
OPERATION
LIBUV
(ASYNCHRONOUS I/O)
EVENT
QUEUE
EVENT
LOOP
BLOCKING
OPERATION
EXECUTE
CALLBACK
WORKER
THREADS
FILE SYSTEM
NETWORK
PROCESS

Node.js
Non-Blocking Event Driven I/O



Node.js 结构



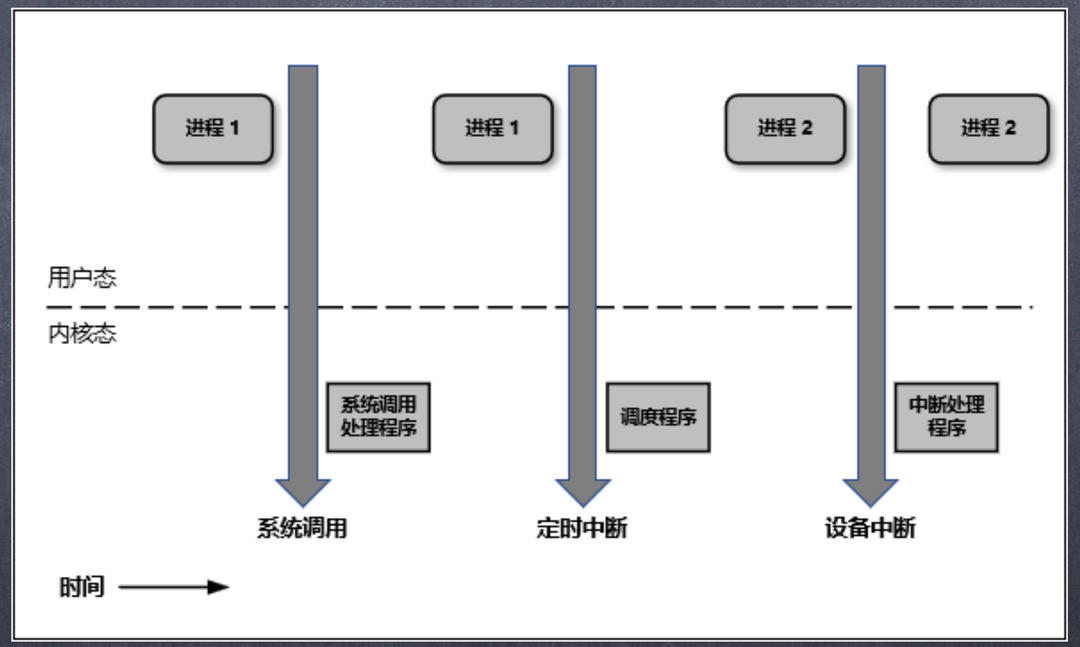
JavaScript

C/C++

Node Standard Library		
Node Bindings (socket, http, file system, etc.)		
Chrome V8 (JS engine)	**Async I/O** (libuv)	**Event Loop** (libuv)

Linux用户空间与内核空间

- 为什么需要区分内核空间与用户空间
- 内核态与用户态
- 如何从用户空间进入内核空间
- 处理器在任何指定时间点上的活动概括为下列三者之一：
  - 运行于用户空间，执行用户进程。
  - 运行于内核空间，处于进程上下文，代表某个特定的进程执行。
  - 运行于内核空间，处于中断上下文，与任何进程无关，处理某个特定的中断。

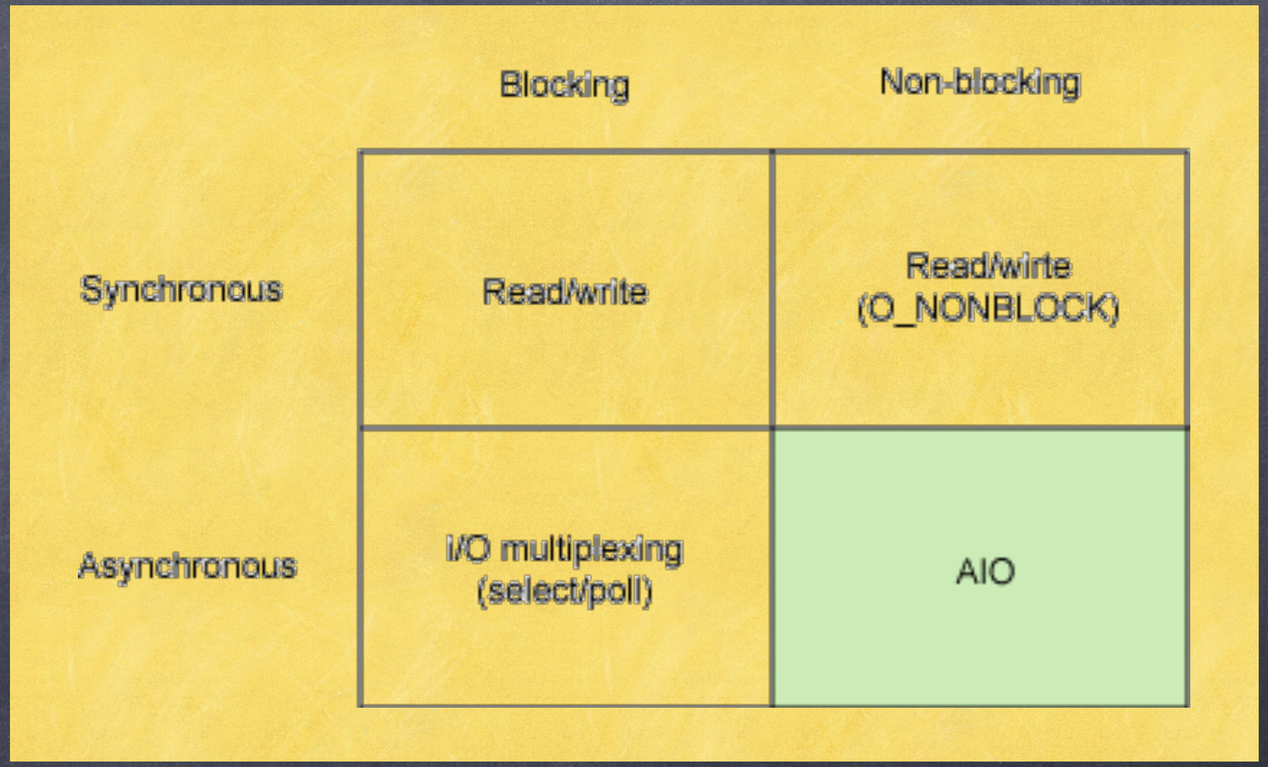


Linux 的信号机制

- 软中断信号（signal，又简称为信号）用来通知进程发生了异步事件。进程之间可以互相通过系统调用kill发送软中断信号。内核也可以因为内部事件而给进程发送信号，通知进程发生了某个事件。信号只是用来通知某进程发生了什么事件，并不给该进程传递任何数据。
- 对信号的处理方法：
  - 1、对于需要处理的信号，进程可以指定处理函数，由该函数来处 理。
  - 2、忽略某个信号，对该信号不做任何处理，就象未发生过一样。其中，有两个信号不能忽略：SIGKILL及SIGSTOP；
  - 3、对该信号的处理保留系统的默认值，这种缺省操作，对大部分的信 号的缺省操作是使得进程终止。进程通过系统调用signal来指定进程对某个信号的处理行为。
- 内核处理一个进程收到的信号的时机是在一个进程从内核态返回用户态时。所以，当一个进程在内核态下运行时，软中断信号并不立即起作用，要等到将返回用户态时才处理。进程只有处理完信号才会返回用户态，进程在用户态下不会有未处理完的信号。
- 可使用 linux 命令 kill -l 查看所有信号的定义

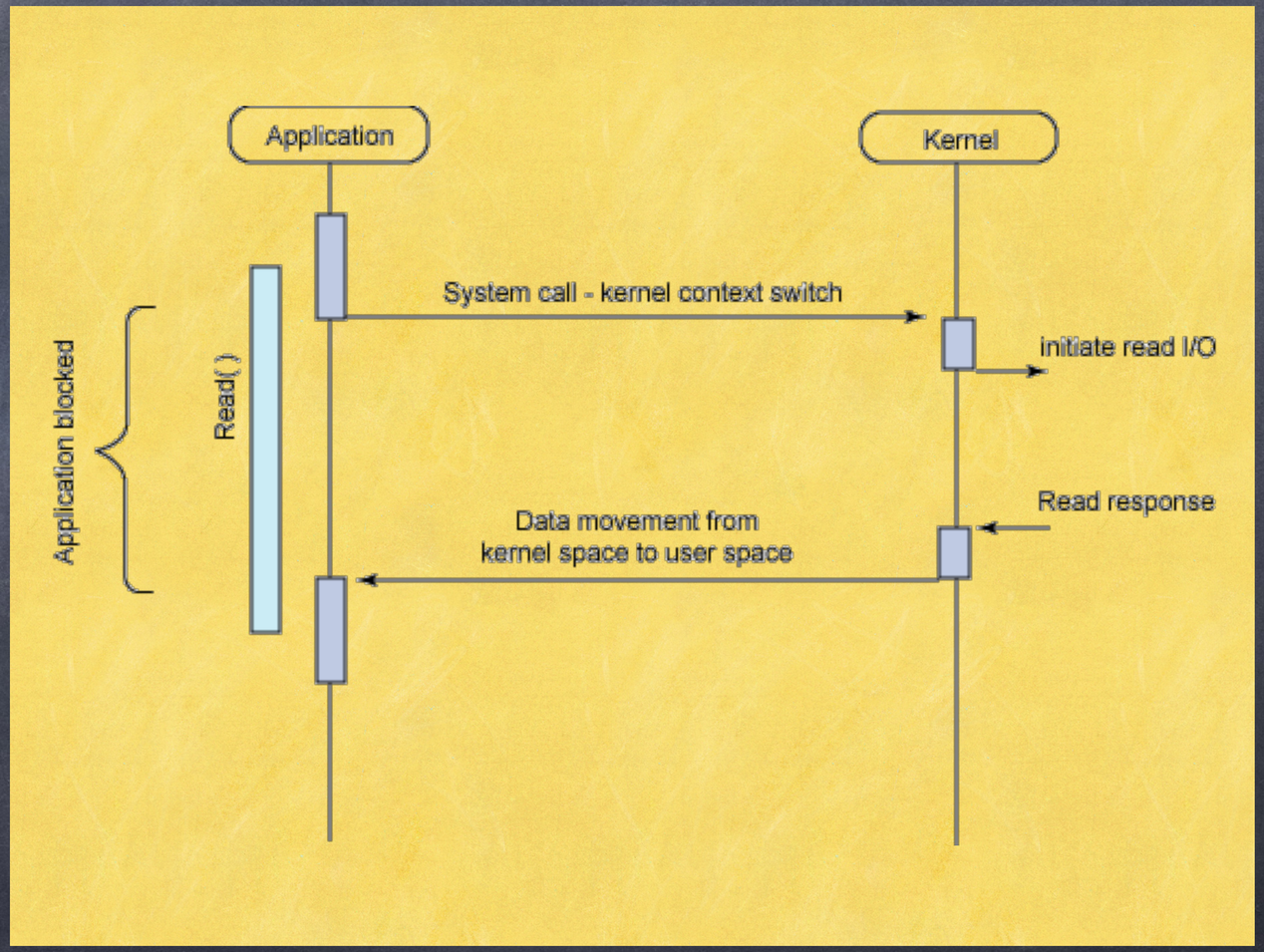
Linux中的 I/O 模型

- 同步和异步
- 阻塞和非阻塞
- 每一个I/O模型都有它适合的应用场景。



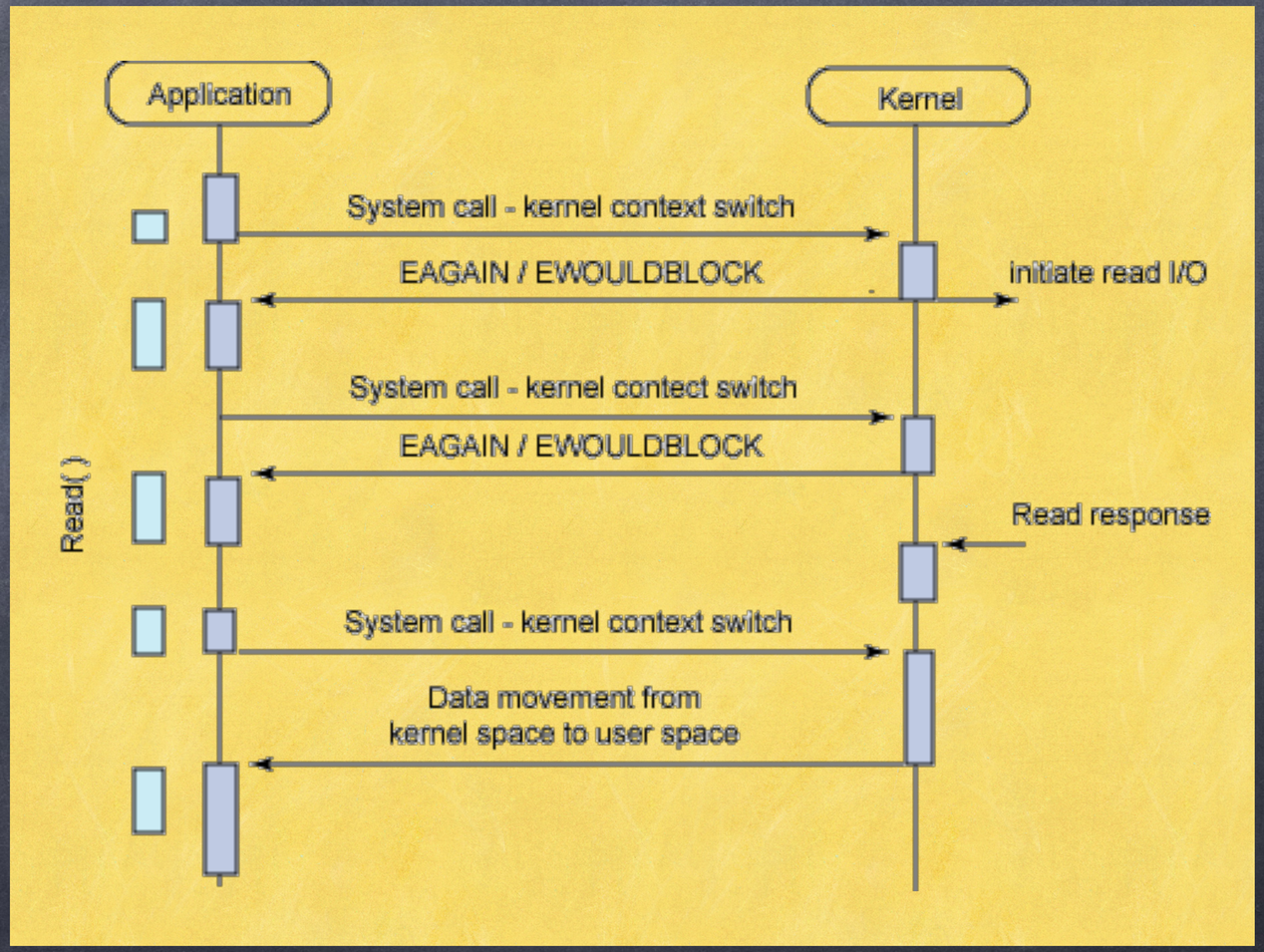
同步阻塞I/O模型

- 同步阻塞I/O模型是最常见的模型之一
- 用户空间的应用程序在执行系统调用之后会被阻塞，直到系统调用完成（数据传输完成或者出现错误）。此时应用程序只是处于简单的等待响应状态不会消耗CPU
- 站在CPU的角度他是高效的。
-



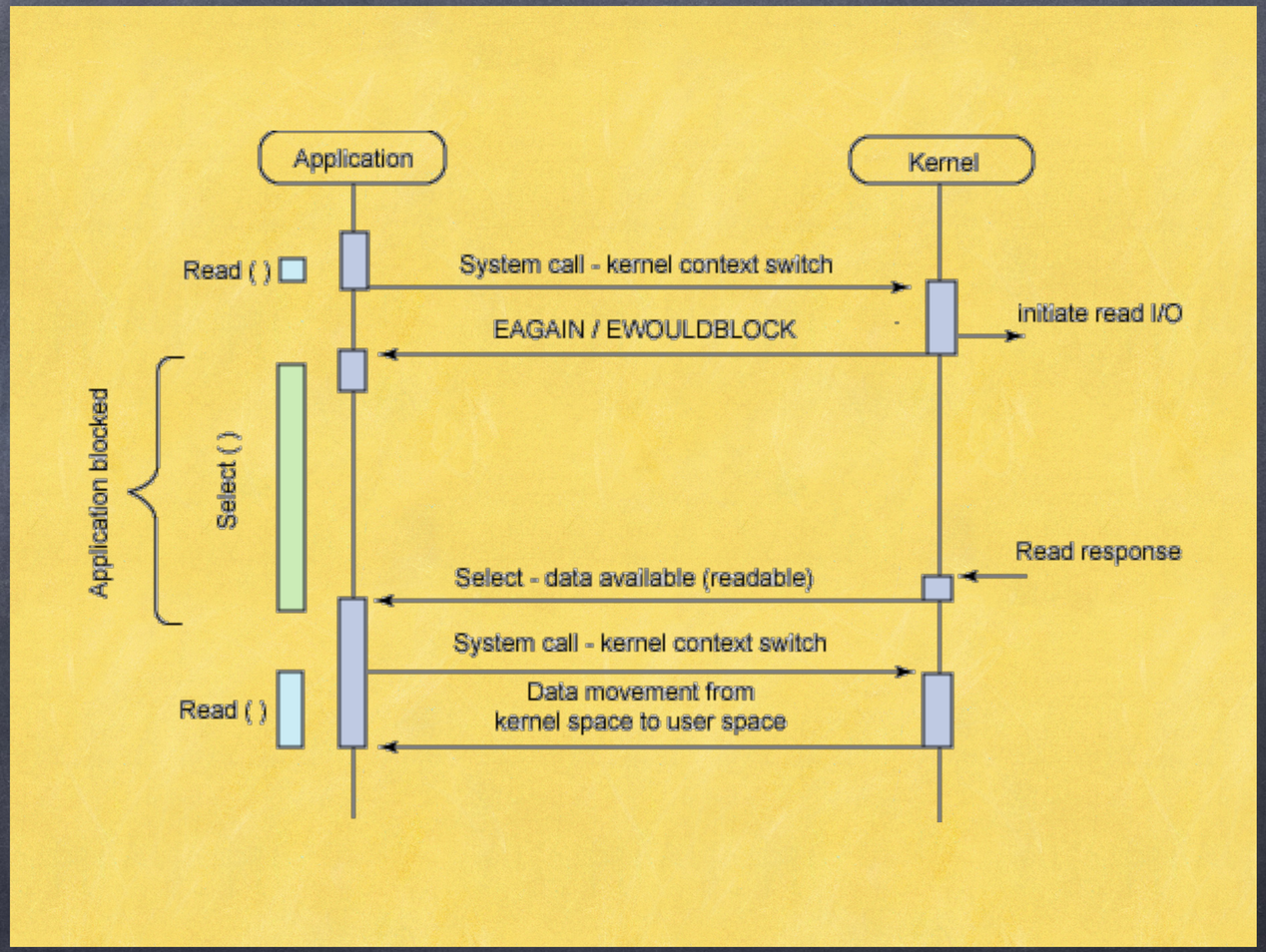
同步非阻塞I/O模型

- 同步阻塞I/O的一种变体是效率较低的同步非阻塞I/O。
- 非阻塞意味着如果I/O操作不能立即完成，则需要应用程序多次调用直到任务完成。
- 这可能非常低效，因为大多数时候应用程序必须忙等待或者尝试做其他事情直到数据可用。



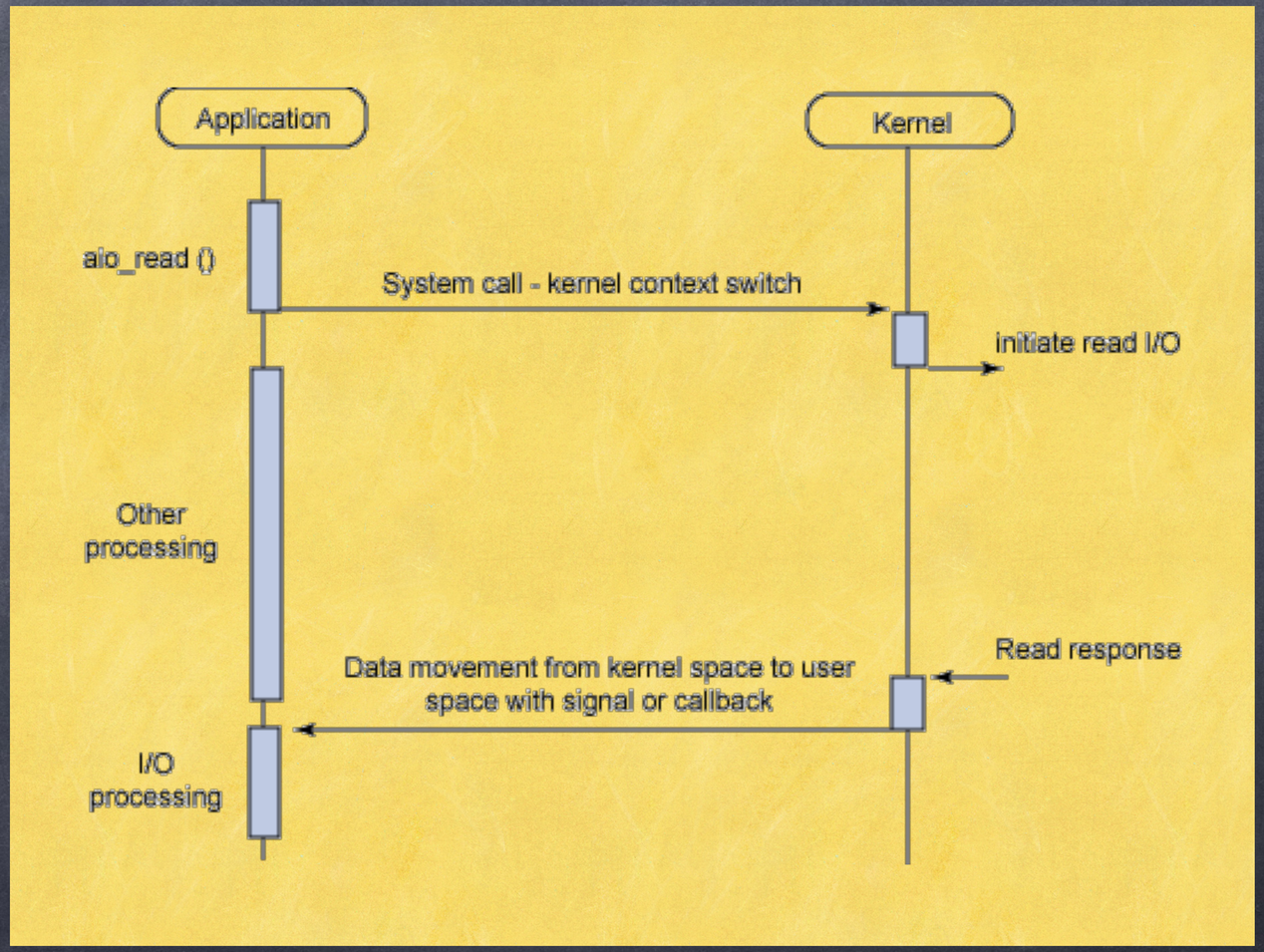
异步阻塞I/O模型

- 设备以非阻塞方式打开，然后应用程序阻塞在select系统调用中，用它来监听可用的I/O操作。
- select系统调用最大的好处是可以监听多个描述符，而且可以指定每个描述符要监听的事件：可读事件、可写事件和发生错误事件
- select系统调用的主要问题是效率不高。虽然它是一个非常方便的异步通知模型，但不建议将其用于高性能I/O中。
- 高性能场景一般使用epoll系统调用



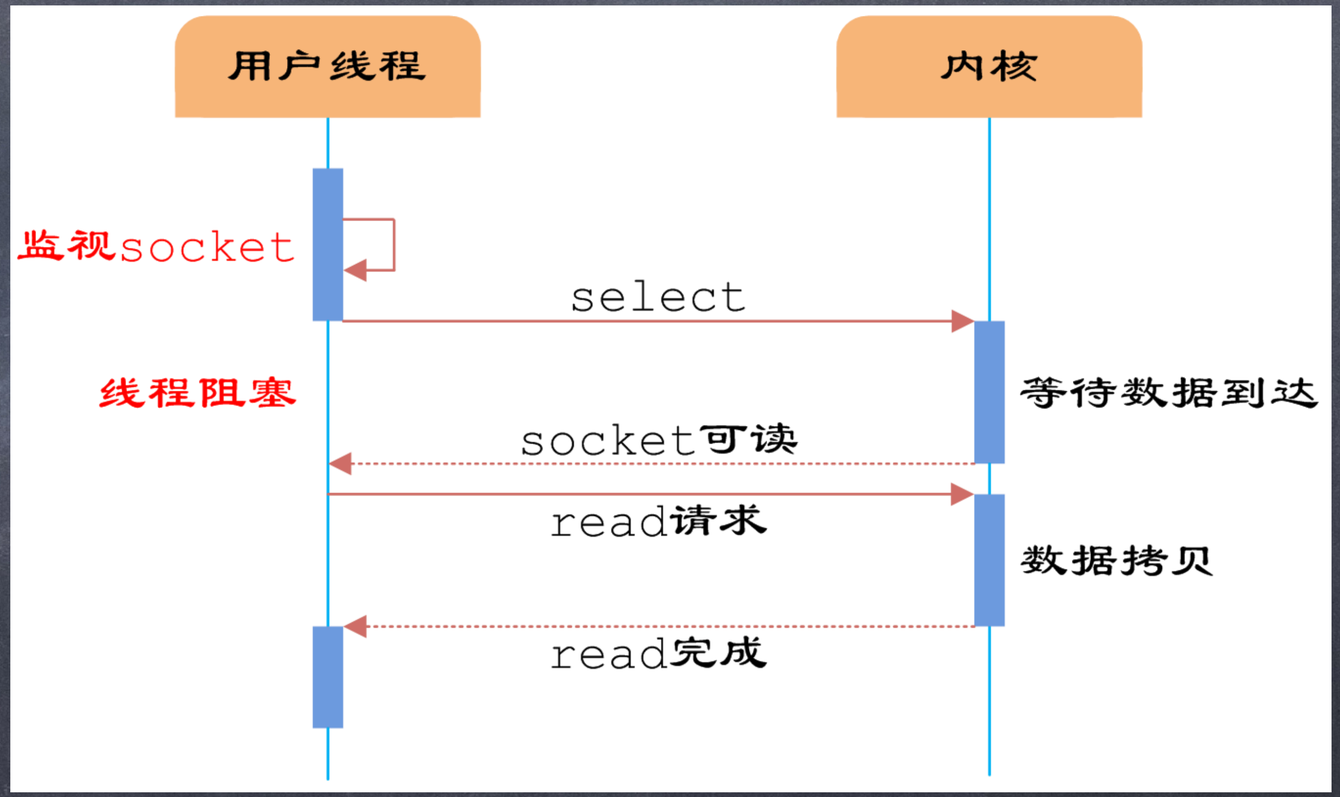
异步非阻塞I/O模型

- 异步非阻塞I/O模型是可以并行处理I/O的模型
- 异步非阻塞I/O模型的读请求会立即返回，表明读操作成功启动。然后应用程序就可以在读操作完成之前做其他的事情。当读操作完成时，内核可以通过信号或者基于线程的回调函数来通知应用程序读取数据。
- 在单个进程可以并行执行多个I/O请求是因为CPU的处理速度要远大于I/O的处理速度。 当一个或多个I/O请求在等待处理时，CPU可以处理其他任务或者处理其他已完成的I/O请求。



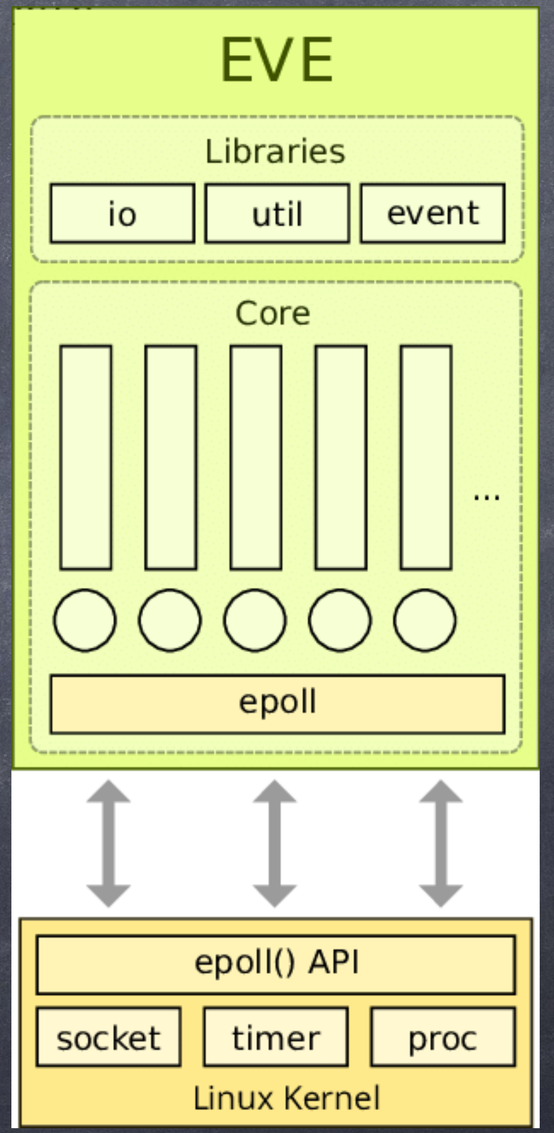
select

- select模型的关键是使用一种有序的方式，对多个套接字进行统一管理与调度
- select模型的缺点：
  - 单个进程能够监视的文件描述符的数量存在最大限制，通常是1024，当然可以更改数量，但由于select采用轮询的方式扫描文件描述符，文件描述符数量越多，性能越差；(在linux内核头文件中，有这样的定义：#define __FD_SETSIZE 1024)
  - 内核/用户空间内存拷贝问题，select需要复制大量的句柄数据结构，产生巨大的开销；
  - select返回的是含有整个句柄的数组，应用程序需要遍历整个数组才能发现哪些句柄发生了事件；
  - select的触发方式是水平触发，应用程序如果没有完成对一个已经就绪的文件描述符进行IO操作，那么之后每次select调用还是会将这些文件描述符通知进程。



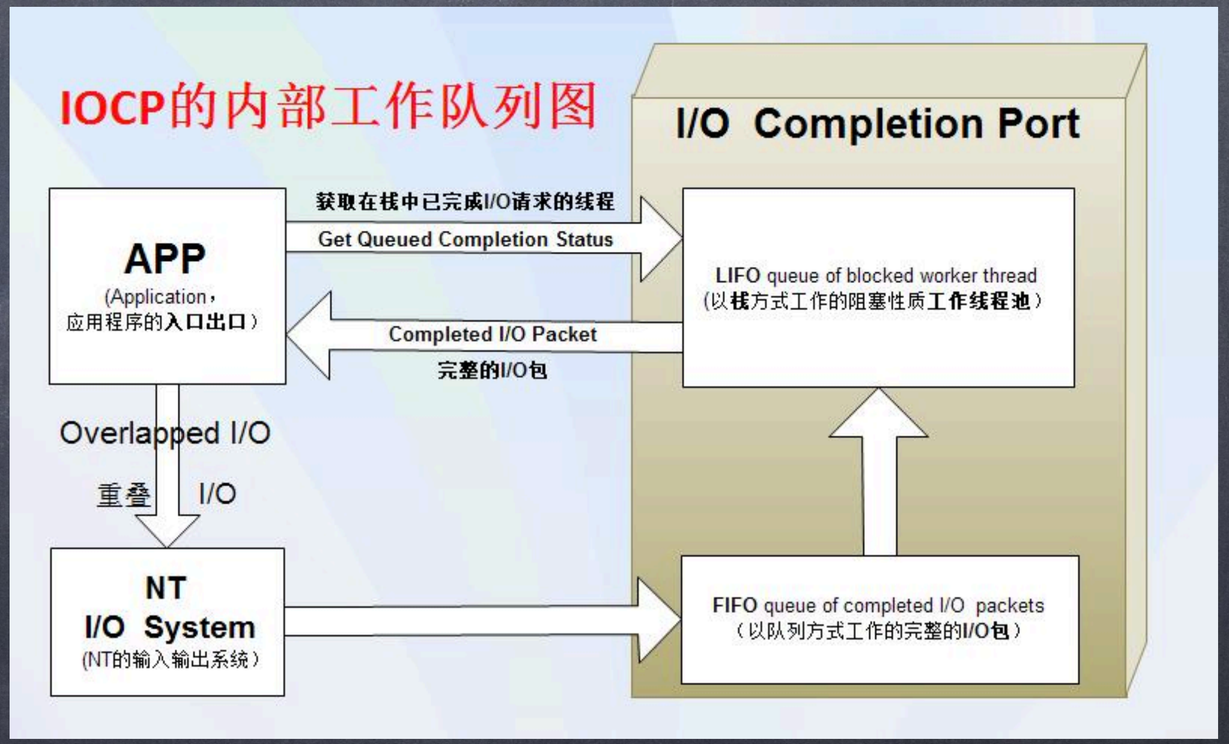
epoll

- epoll模型的优点：
  - 支持一个进程打开大数目的socket描述符
  - IO效率不随FD数目增加而线性下降
  - 使用mmap加速内核与用户空间的消息传递
- epoll 的两种工作模式：
  - LT(level triggered，水平触发模式)是缺省的工作方式，并且同时支持 block 和 non-block socket。在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。比如内核通知你其中一个fd可以读数据了，你赶紧去读。你还是懒懒散散，不去读这个数据，下一次循环的时候内核发现你还没读刚才的数据，就又通知你赶紧把刚才的数据读了。这种机制可以比较好的保证每个数据用户都处理掉了。
  - ET(edge-triggered，边缘触发模式)是高速工作方式，只支持no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过epoll告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知，等到下次有新的数据进来的时候才会再次出发就绪事件。简而言之，就是内核通知过的事情不会再说第二遍，数据错过没读，你自己负责。这种机制确实速度提高了，但是风险相伴而行。



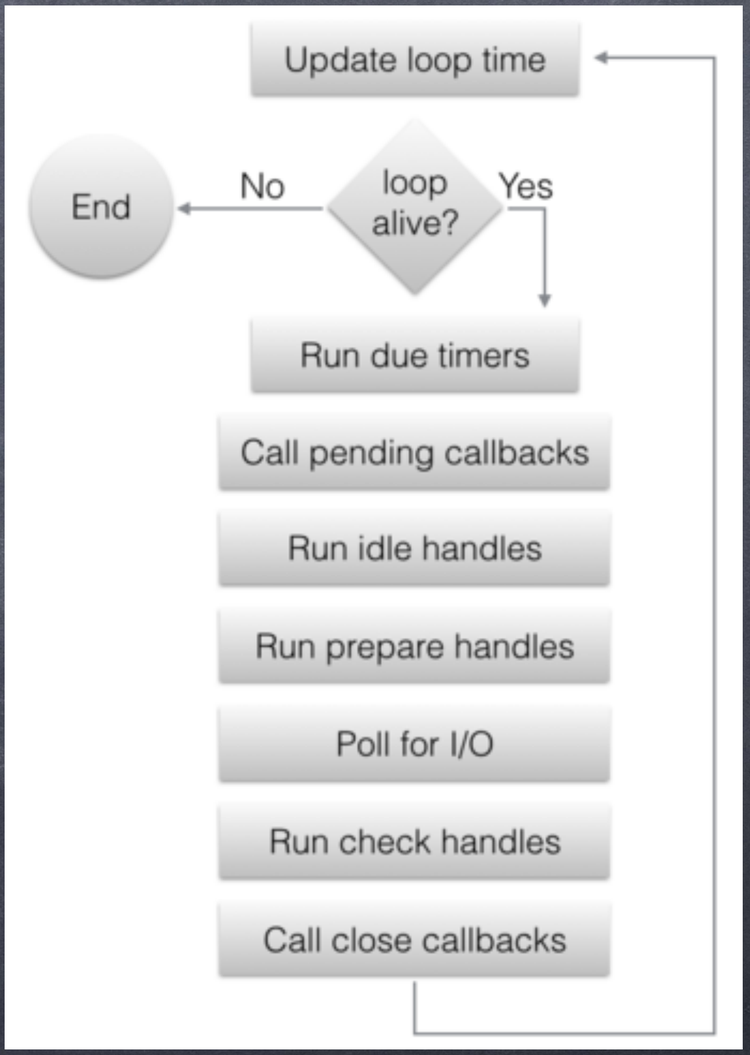
IOCP模型

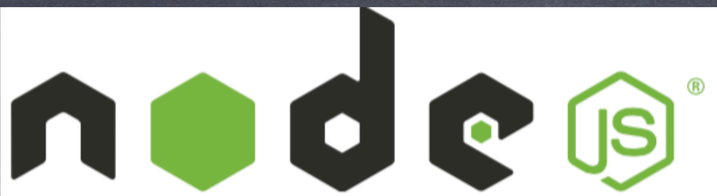
- 1：创建一个完成端口。
- 2：创建一个线程A。
- 3：A线程循环调用GetQueuedCompletionStatus()函数来得到IO操作结果，这个函数是个阻塞函数。
- 4：主线程循环里调用accept等待客户端连接上来。
- 5：主线程里accept返回新连接建立以后，把这个新的套接字句柄用CreateIoCompletionPort关联到完成端口，然后发出一个异步的 WSASend或者WSARecv调用，因为是异步函数，WSASend/WSARecv会马上返回，实际的发送或者接收数据的操作由WINDOWS系统去做。
- 6：主线程继续下一次循环，阻塞在accept这里等待客户端连接。
- 7：WINDOWS系统完成WSASend或者WSArecv的操作，把结果发到完成端口。
- 8：A线程里的GetQueuedCompletionStatus()马上返回，并从完成端口取得刚完成的
- 9：在A线程里对这些数据进行处理(如果处理过程很耗时，需要新开线程处理)，然后接着发



LibUV

- libuv 是用 C 写的，因此，它具有很高的可移植性，非常适用嵌入到像 JavaScript 和 Python 这样的高级语言中。
- libuv使用异步，事件驱动的编程方式，核心是提供 i/o 的事件循环和异步回调。
- libuv的API包含有时间，非阻塞的网络，异步文件操作，子进程等等。
- 当程序在等待i/o完成的时候，我们希望cpu不要被这个等待中的程序阻塞，libuv提供的编程方式使我们开发异步程序变得简单。
-





1. node.js calls fs.readSync()

fs.readSync

2. readSync calls uv_fs_read, which calls uv_fs_work

uv_fs_read

uv_fs_work

3. uv_fs_work calls uv__fs_buf_iter

6. if errno == EINTR, uv_fs_work calls uv__fs_buf_iter again

5. uv__fs_read finishes, returning to uv_fs_work which NULLifies req->bufs

uv__fs_buf_iter

4, 7. uv_fs_buf_iter calls uv__fs_read

uv__fs_read

8. the second read crashes since req->bufs is NULL