Linux铁三角之I/O(一)

Yomocode.com

麦当劳喜欢您来，喜欢您再来

扫描关注

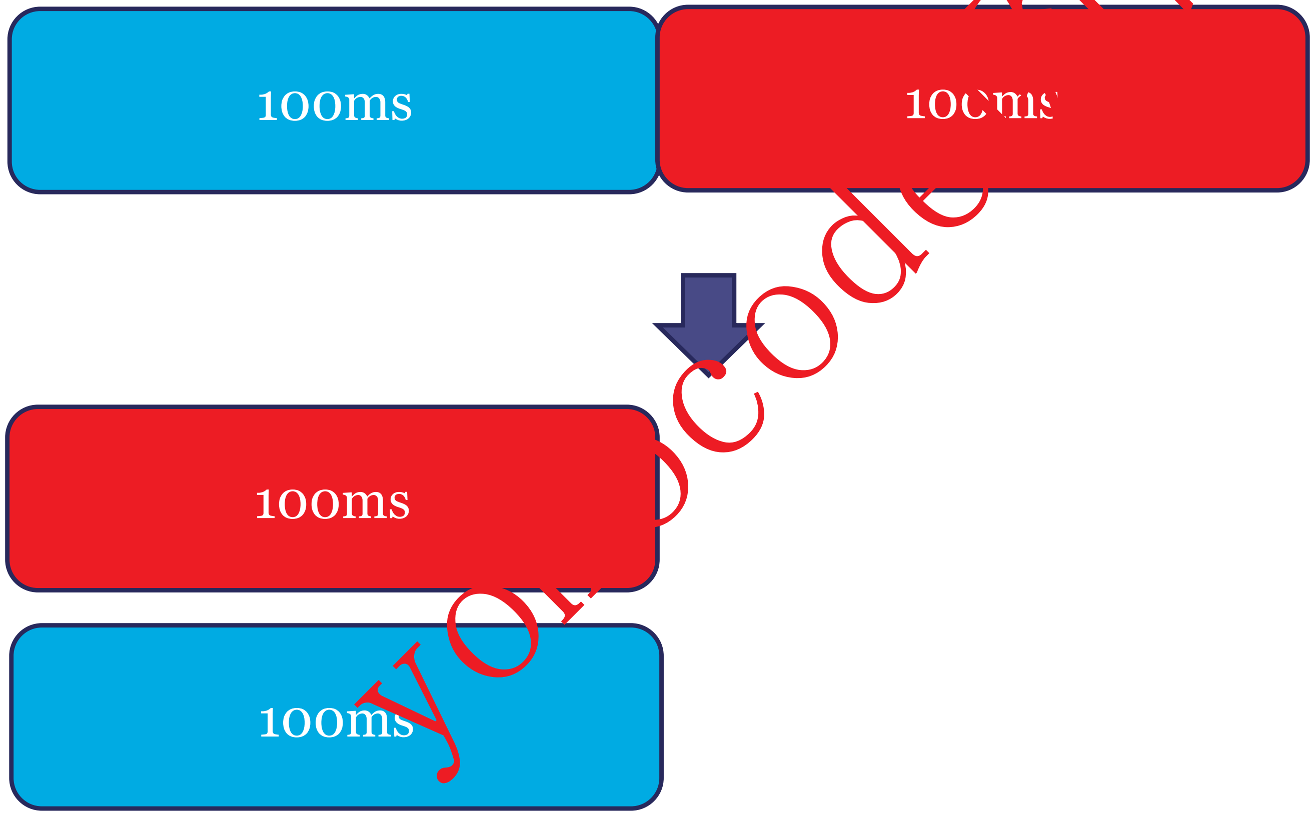
Linux阅码场

I/O与网络模型

*阻塞
*非阻塞
*多路复用
*Signal IO
*异步IO
*libevent

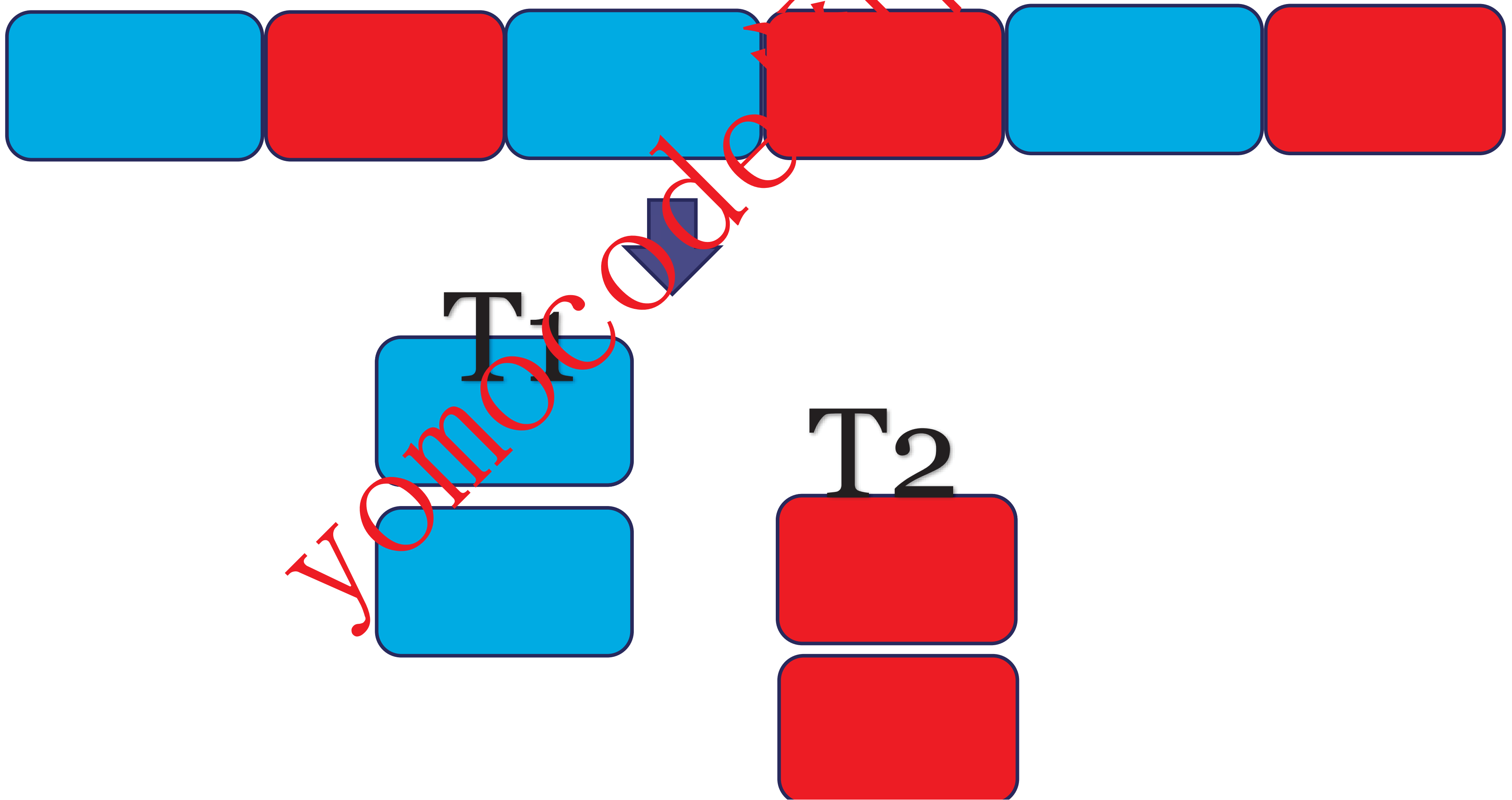
场景案例

- 初始化，而后开背景图片

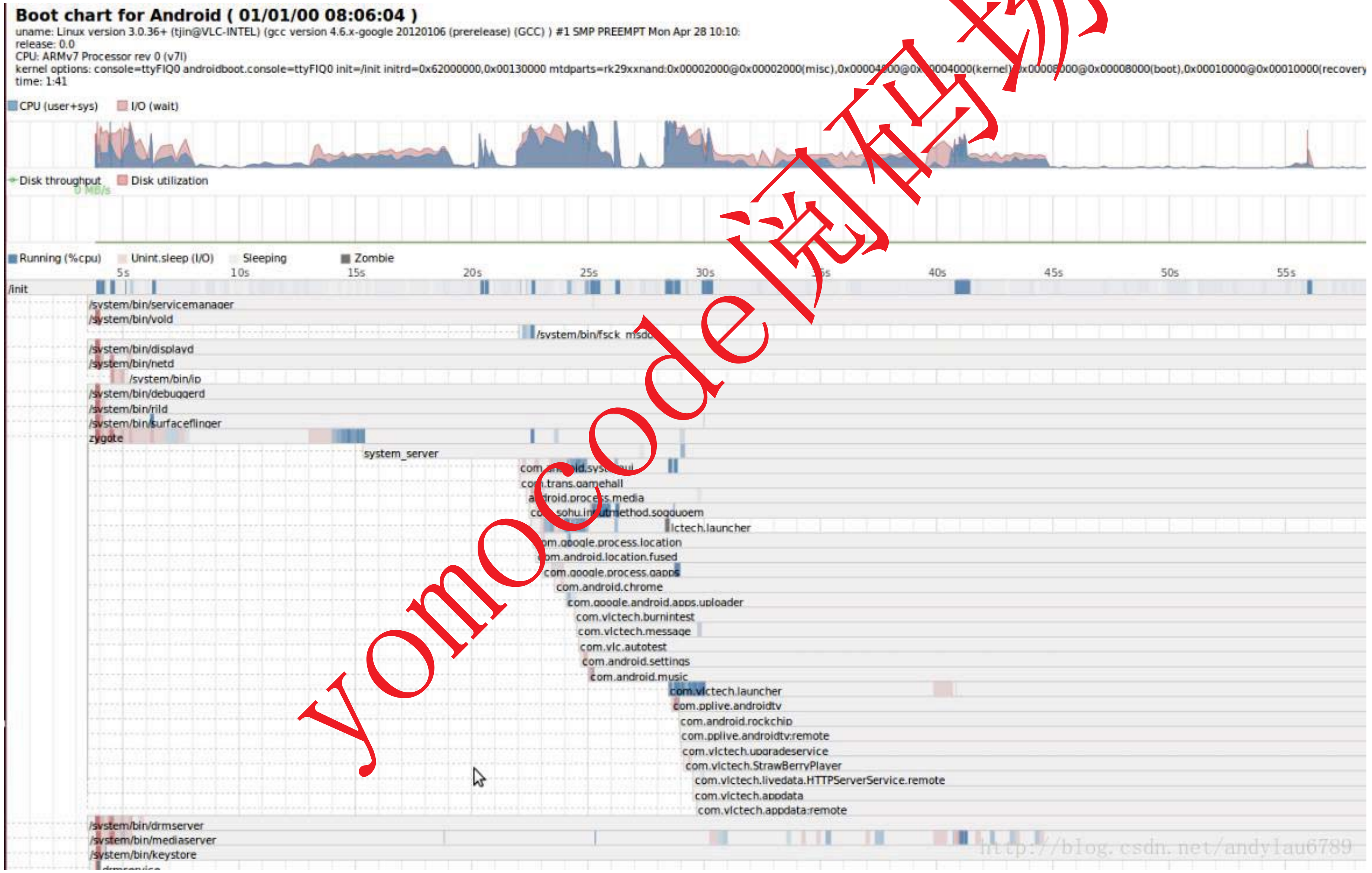


场景案例

- CPU算包，网卡发包



Bootchart

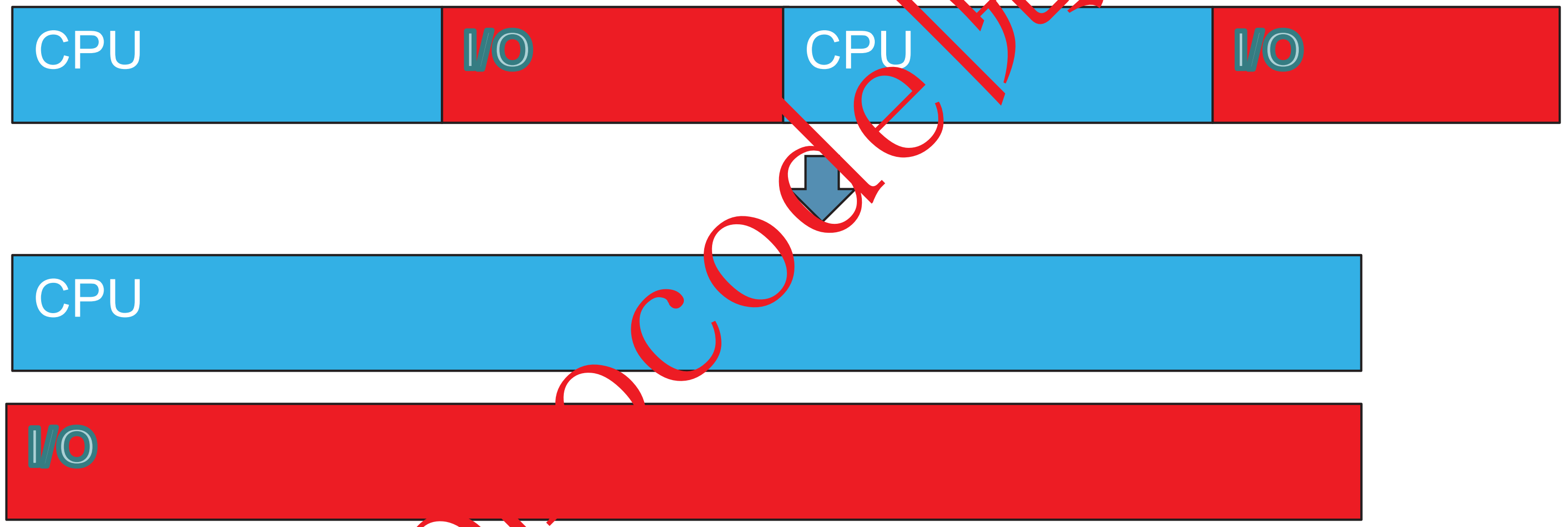


Boot chart for Android (01/01/00 08:06:04)

```
uname: Linux version 3.0.36+ (tjin@VLC-INTEL) (gcc version 4.6.x-google 20120106 (prerelease) (GCC) ) #1 SMP PREEMPT Mon Apr 28 10:10:
release: 0.0
CPU: ARMv7 Processor rev 0 (v7l)
kernel options: console=ttyFIQ0 androidboot.console=ttyFIQ0 init=/init initrd=0x62000000,0x00130000 mtdparts=rk29xxnand:0x00002000@0x00002000(misc),0x00004000@0x00004000(kernel),0x00008000@0x00008000(boot),0x00010000@0x00010000(recovery
time: 1:41
```

Systemd readahead

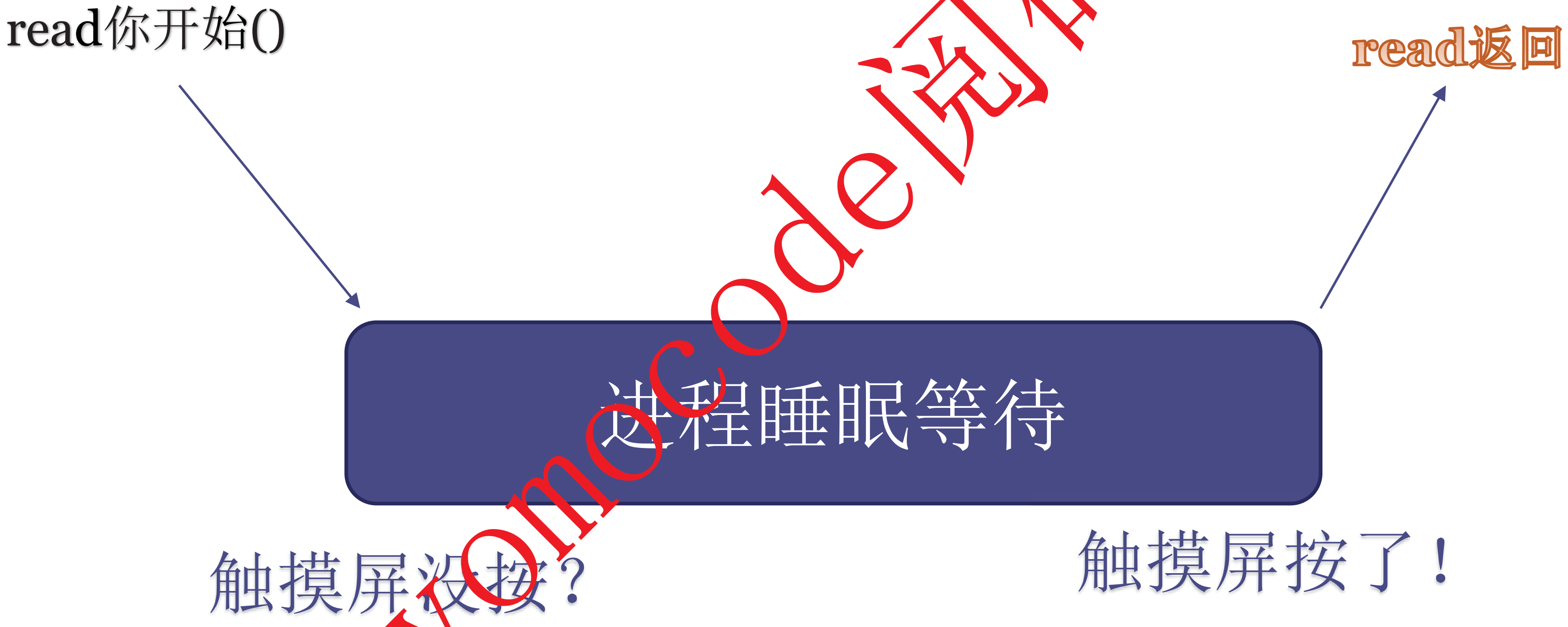
- systemd-readahead-collect.service 搜集系统启动过程中的文件访问信息; systemd-readahead-replay.service在后续启动过程中完成回放。



把CPU和IO的交替等，变为CPU和IO的同时工作，充分利用系统资源。

阻 塞

进程阻塞等I/O ready



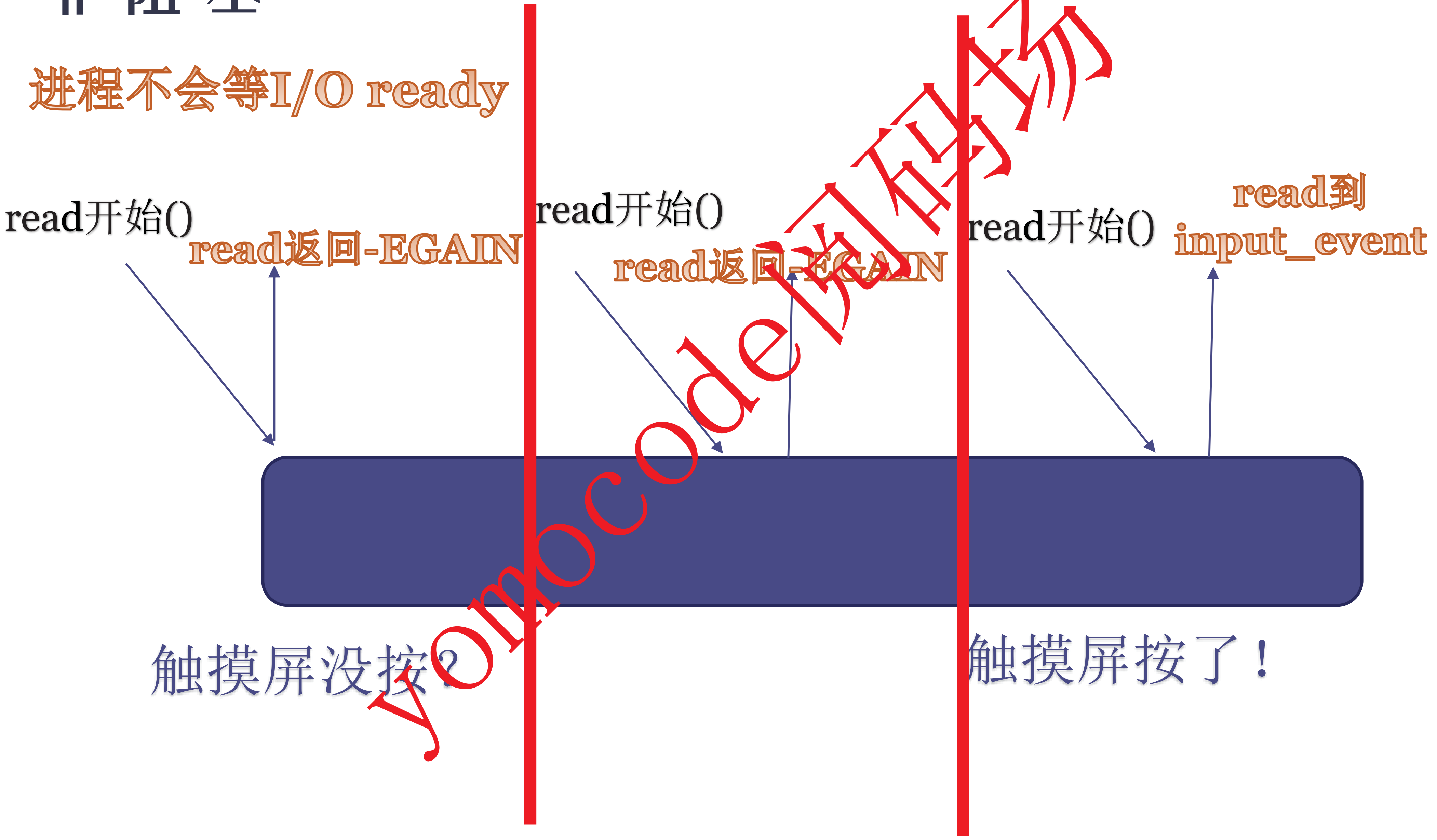
EINTR

被信号打断的系统调用Interrupted system call

```
      act.sa_handler = sig_handler;
       act.sa_flags = 0;
       //  act.sa_flags |= SA_RESTART;
       sigemptyset(&act.sa_mask);
       if (-1 == sigaction(SIGUSR1, &act, &oldact)) {
…
       do {
              ret = read(STDIN_FILENO, buf, 10);
              if ((ret == -1) && (errno == EINTR))
                     printf("retry after eintr\n");
        } while((ret == -1) && (errno == EINTR));
```

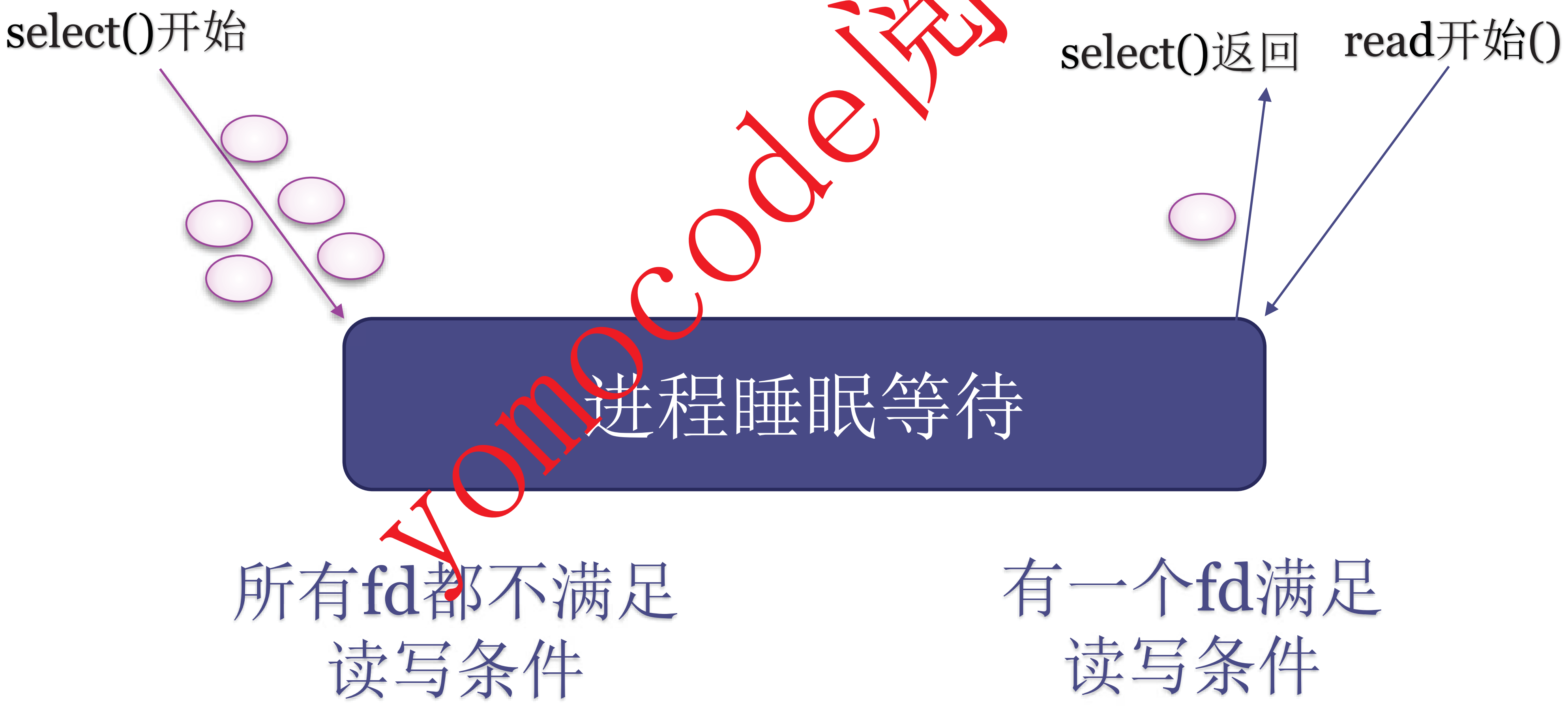
非 阻 塞

进程不会等I/O ready



多路复用

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```



epoll

Epoll的事件注册

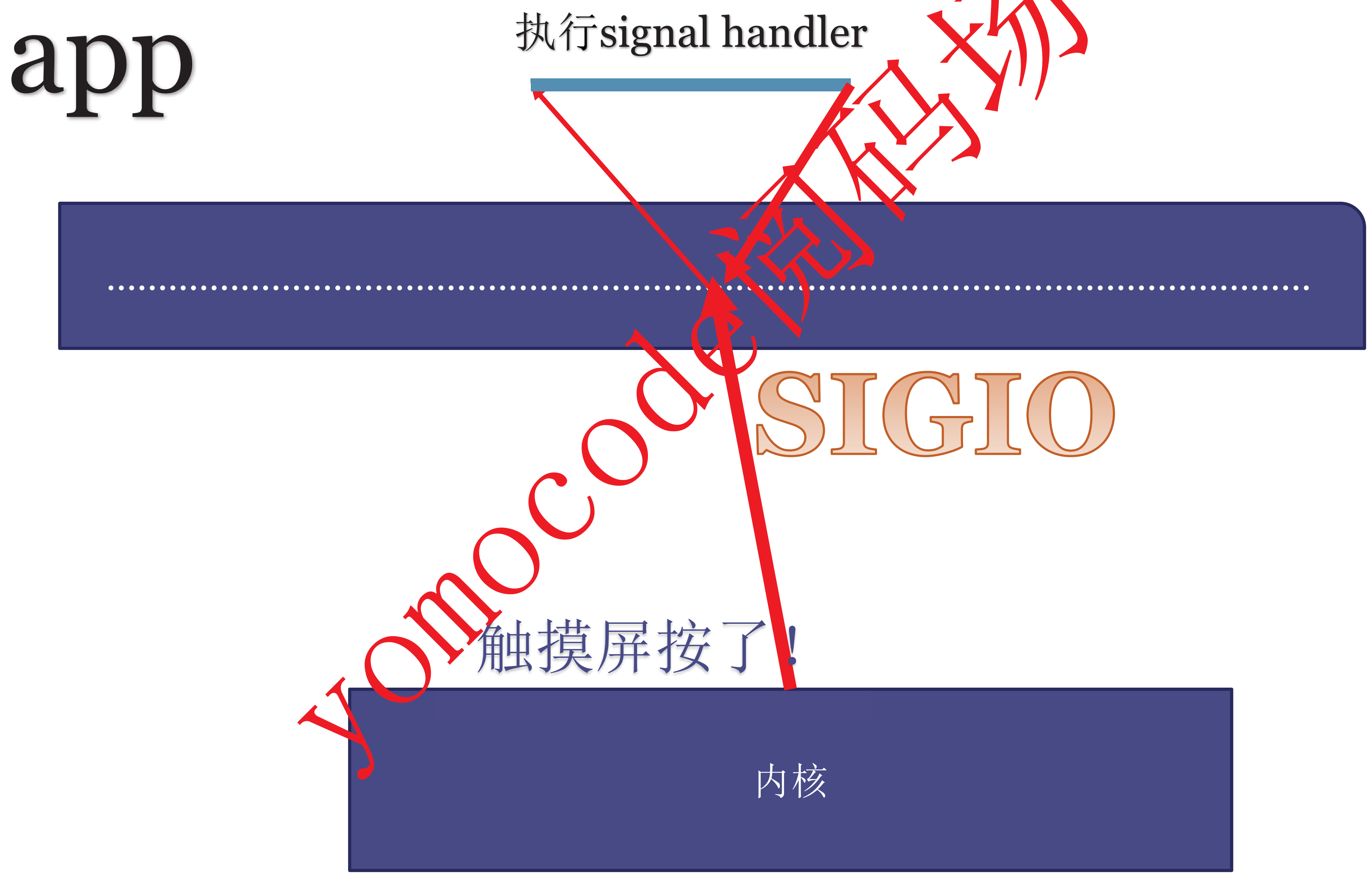
int epoll_ctl(int epfd, intop, int fd, struct epoll_event*event);

EPOLL_CTL_ADD： 注册新的fd到epfd中；
EPOLL_CTL_MOD： 修改已经注册的fd的监听事件；
EPOLL_CTL_DEL： 从epfd中删除一个fd；

等待事件触发

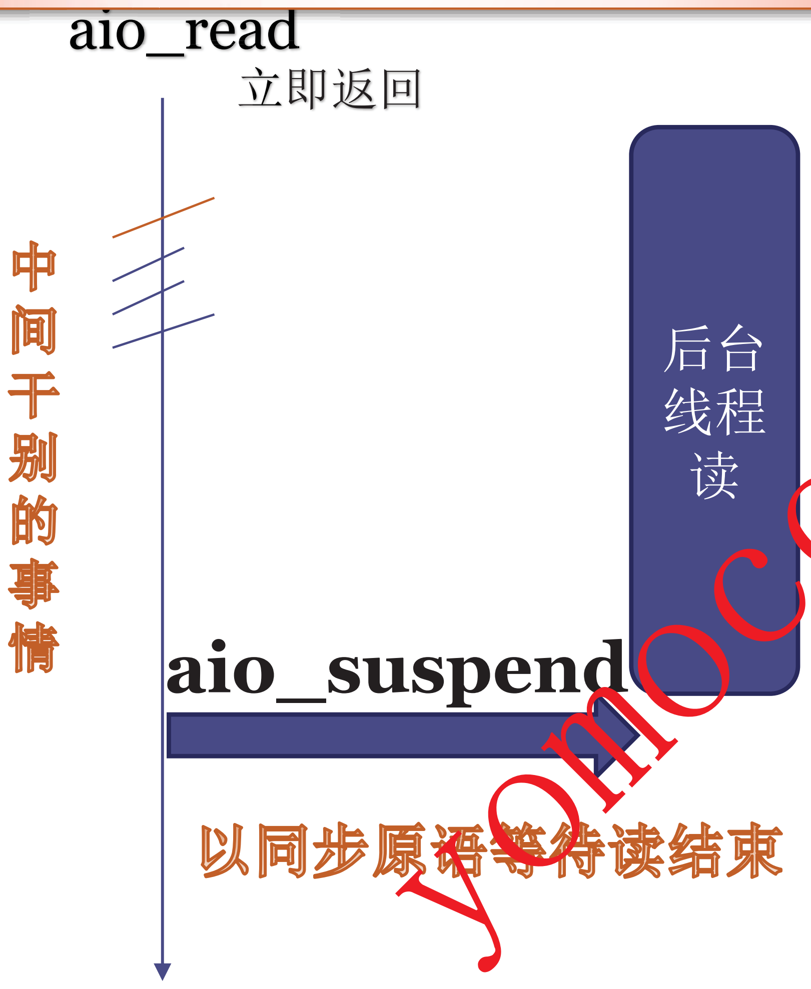
int epoll_wait(int epfd, struct epoll_event * events, intmaxevents, int timeout);

SIGIO



异步I/O

Glibc-AIO



Kernel-AIO w/O_DIRECT

准备上下文
int io_setup(unsigned nr_events, aio_context_t *ctx_idp);
发布io请求
int io_submit(aio_context_t ctx_id, long nr, struct iocb **iocbpp);
等待Completions
int io_getevents(aio_context_t ctx_id, long min_nr, long nr, struct io_event *events, struct timespec *timeout);
销毁上下文
int io_destroy(aio_context_t ctx_id);

Libevent

libevent是一个Reactor:

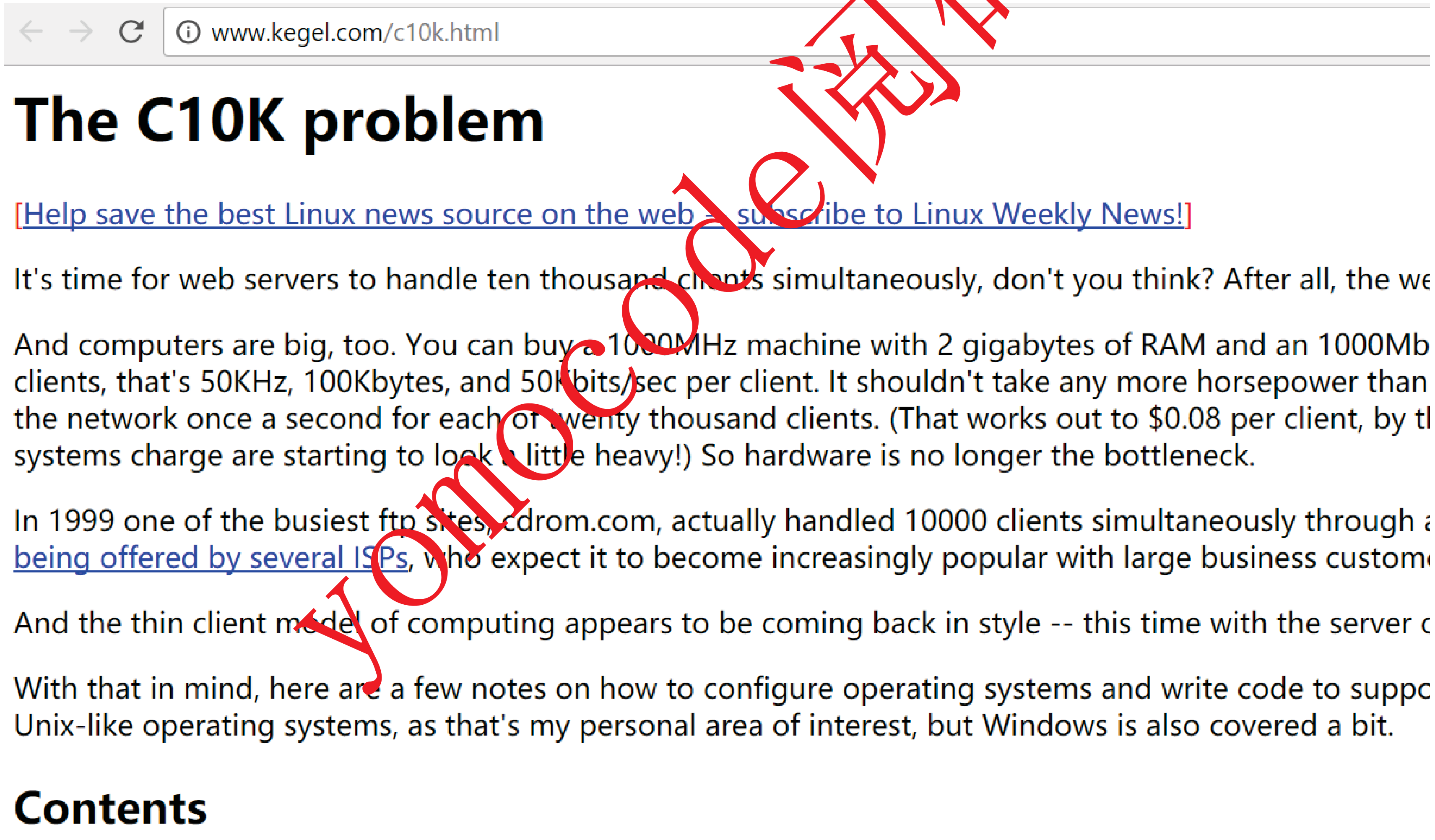
一种事件驱动机制，注册回调函数，如果事件发生，被回调。

```
static void cmd_event(int fd, short events, void *arg)
{
        char msg[1024];

        int ret = read(fd, msg, sizeof(msg));
        if (ret <= 0) {
                perror("read fail ");
                exit(1);
        }
        msg[ret]='\0';
        printf("%s", msg);
}

int main(int argc, char **argv)
{
        struct event ev_cmd;
        event_init();
        //监听终端输入事件
        event_set(&ev_cmd, STDIN_FILENO,
                   EV_READ | EV_PERSIST, cmd_event, NULL);

        event_add(&ev_cmd, NULL);
        event_dispatch();

        return 0;
}
```

有人输入，触发回调

C10K问题

大多数开发人员都能很容易地从功能上实现，
但一旦放到大并发场景下……


www.kegel.com/c10k.html

The C10K problem

[Help save the best Linux news source on the web -- subscribe to Linux Weekly News!]

It's time for web servers to handle ten thousand clients simultaneously, don't you think? After all, the we

And computers are big, too. You can buy a 1000MHz machine with 2 gigabytes of RAM and an 1000Mb clients, that's 50KHz, 100Kbytes, and 50Kbits/sec per client. It shouldn't take any more horsepower than the network once a second for each of twenty thousand clients. (That works out to $0.08 per client, by tl systems charge are starting to look a little heavy!) So hardware is no longer the bottleneck.

In 1999 one of the busiest ftp sites, cdrom.com, actually handled 10000 clients simultaneously through a being offered by several ISPs, who expect it to become increasingly popular with large business custom

And the thin client model of computing appears to be coming back in style -- this time with the server c

With that in mind, here are a few notes on how to configure operating systems and write code to suppc Unix-like operating systems, as that's my personal area of interest, but Windows is also covered a bit.

Contents

模 型 对 比

模型	特点
一个连接，一个进程/线程	进程/线程会占用大量的系统资源，切换开销大； 可扩展性差
一个进程/线程，处理多个连接 select	fd上限+重复初始化+逐个排查所有fd状态，O(n)的效率不断地去查fd
一个进程/线程，处理多个连接 epoll	epoll_wait()返回的时候只给应用提供发生了状态变化的fd 典型用户：nginx, node.js
Libevent: 跨平台，封装底层平台的调用，提供统一的 API (Windows-IOCP, Solaris- /dev/poll, FreeBSD-kqueue, Linux - epoll)	当一个fd的特定事件（如可读，可写或出错）发生了，libevent就会自动执行用户指定的callback，来处理事件。

谢 谢！