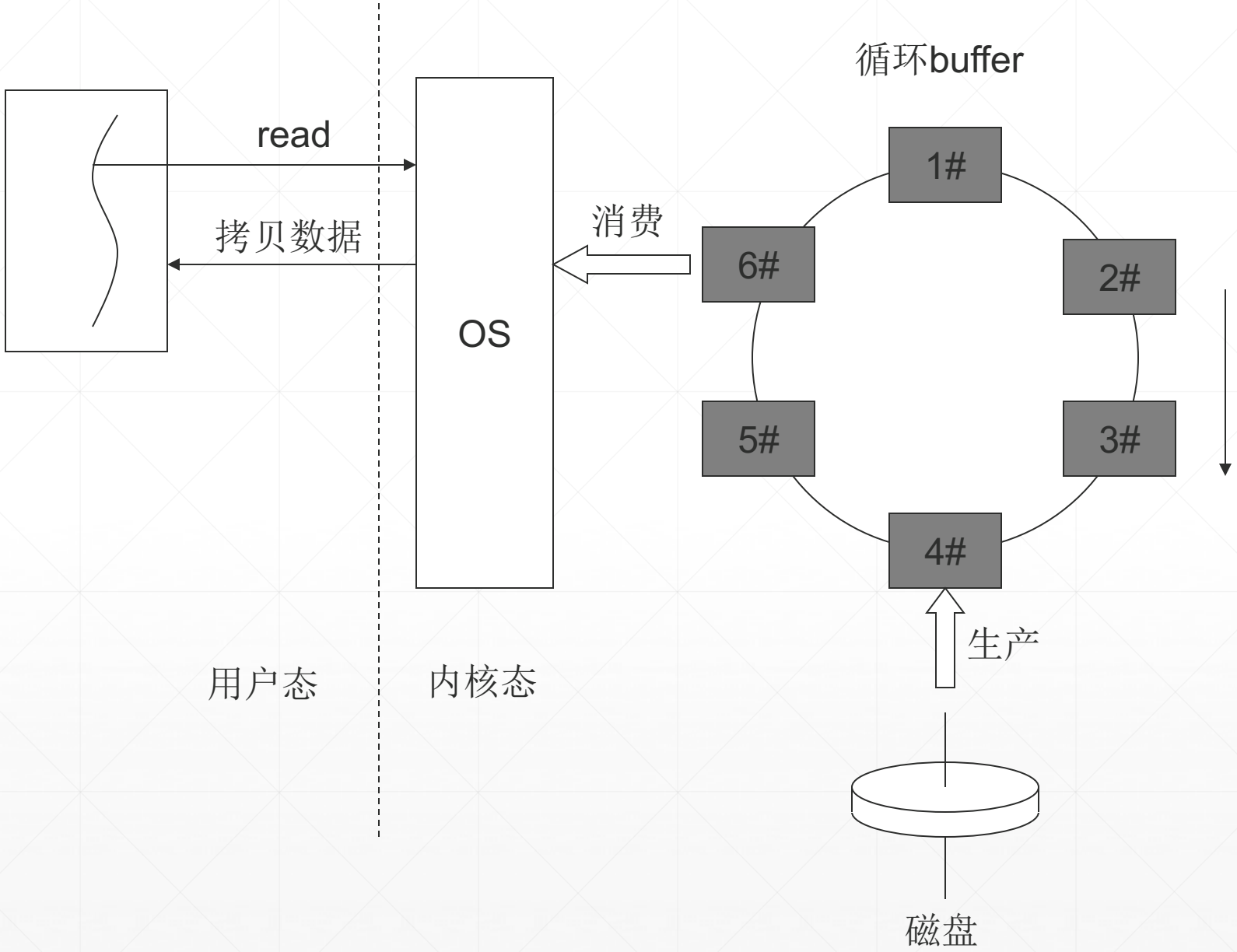
网络计算编程

EPOLL服务器设计

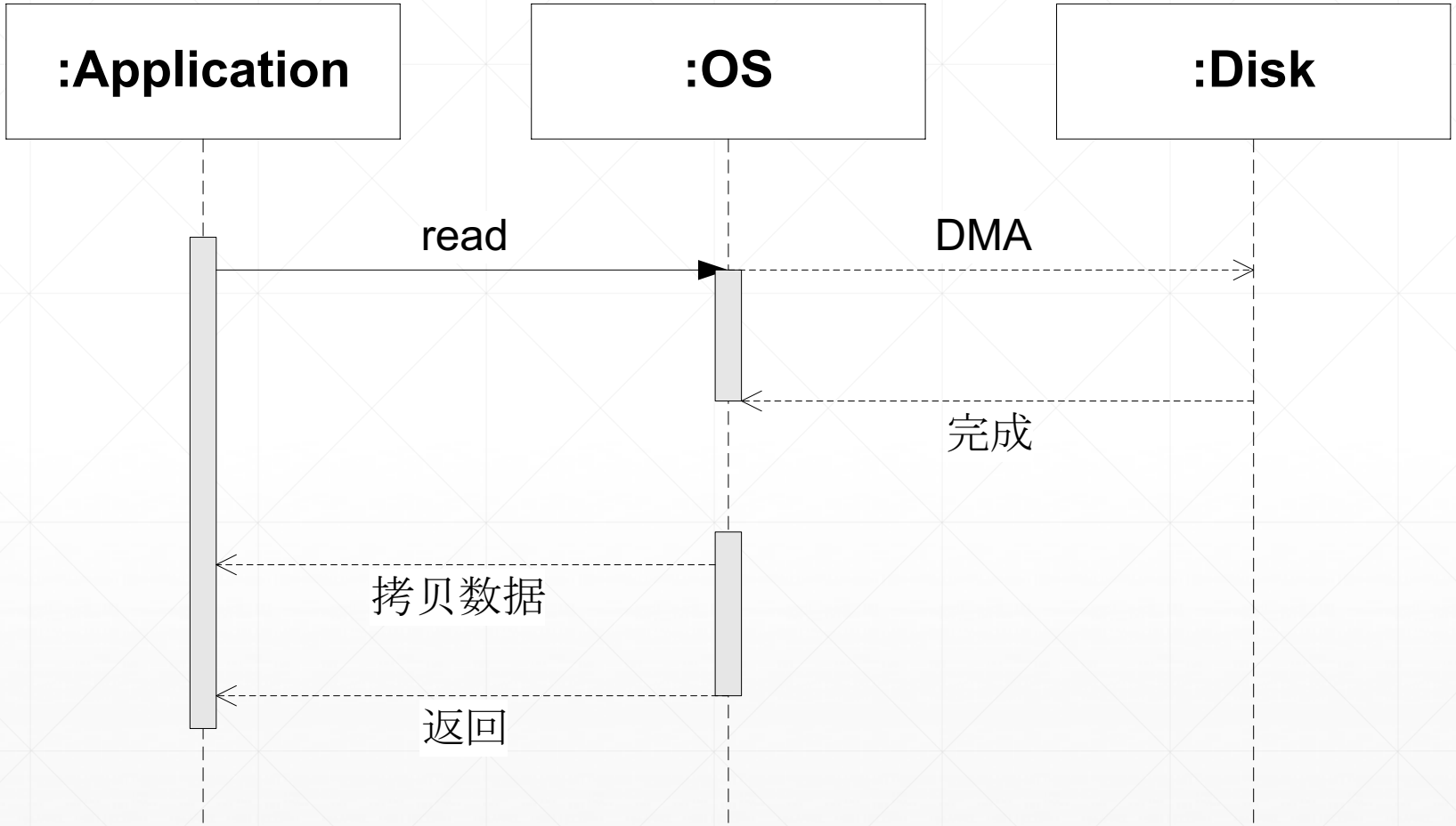
信息与软件工程学院

读写操作的IO模型

- 观察迭代式服务器代码，发现：除了bind/listen/accept这些动作外，服务器的主要动作是读写socket，标准输入/输出
- 服务器的性能与read/write这些操作密切相关，因此，有必要更透彻地了解文件的读写模型
- Unix／Linux系统将IO的读写操作抽象为文件读写，操作系统中关于IO操作的实现方式是怎样的？
  - Buffer / 双buffer / 循环缓冲 / buffer池
  - 阻塞


循环buffer
read
拷贝数据
OS
消费
1#
2#
3#
4#
5#
6#
生产
磁盘
用户态
内核态

读磁盘的顺序图



五种I/O 模式

① 阻塞 I/O (Linux下的I/O操作默认是阻塞I/O，即open和socket创建的I/O都是阻塞I/O)

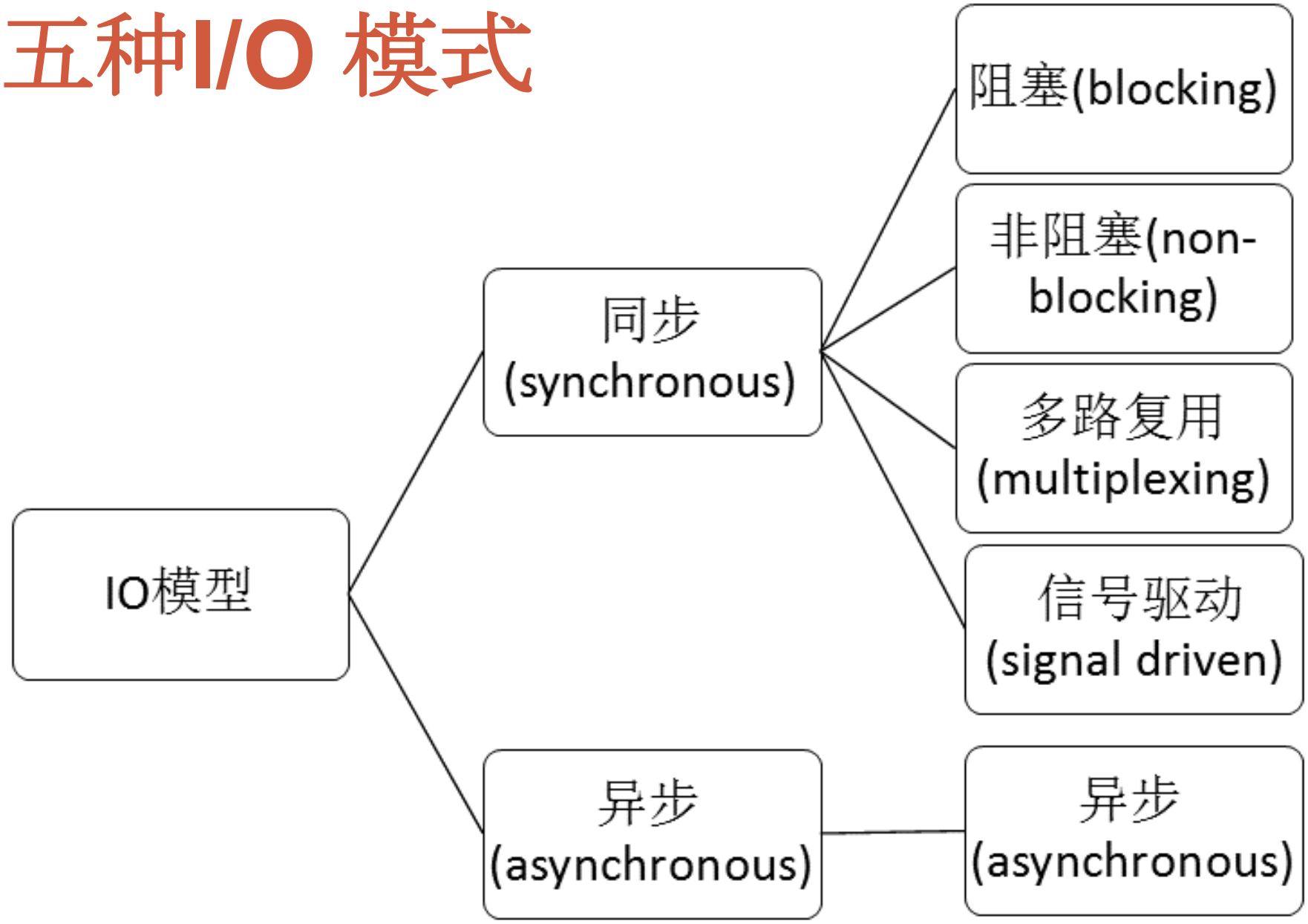
② 非阻塞 I/O (可以通过fcntl或者open时使用O_NONBLOCK参数，将fd设置为非阻塞的I/O)

③ I/O 多路复用 (I/O多路复用，通常需要非阻塞I/O配合使用)—select，poll，epoll

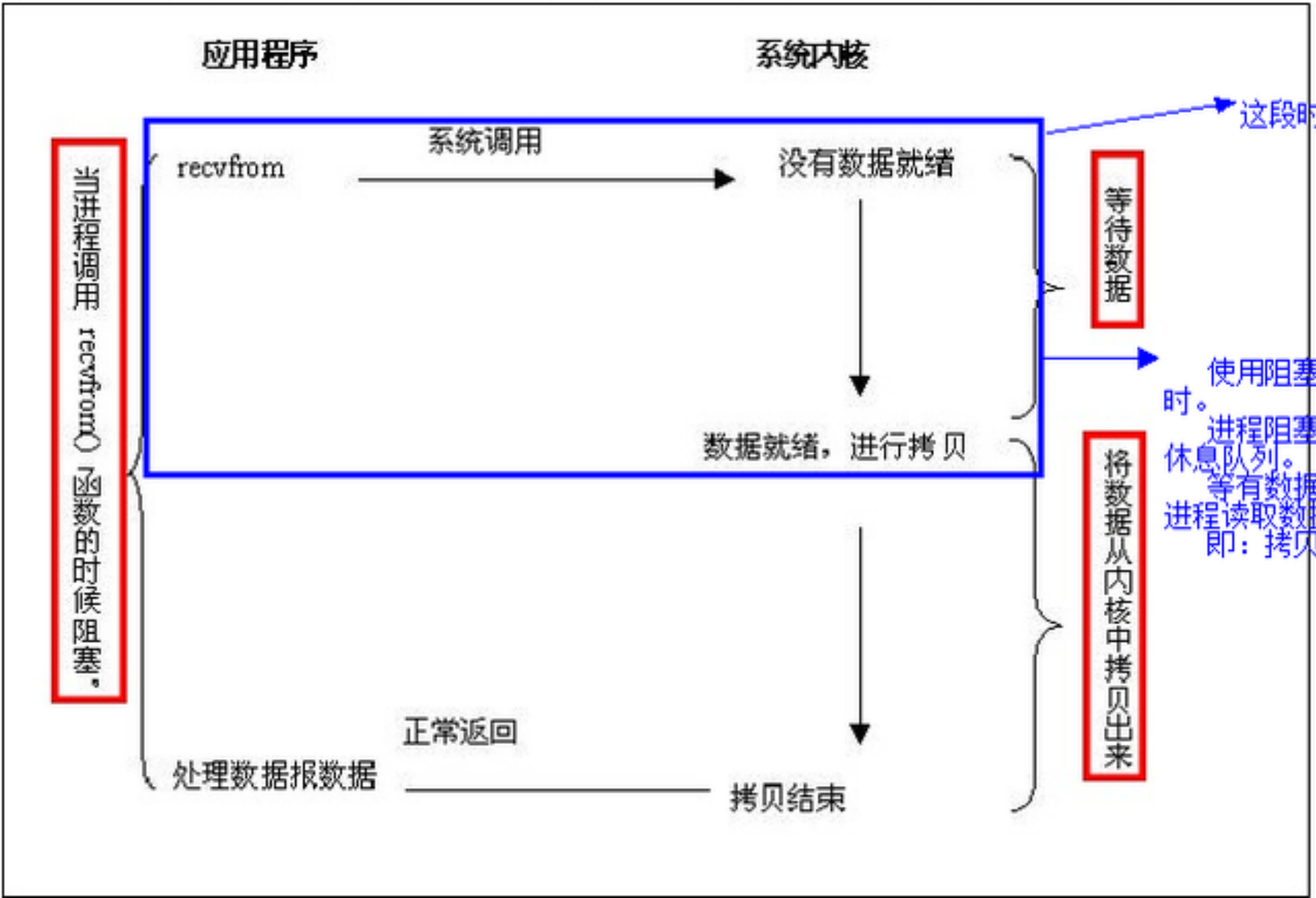
④ 信号驱动 I/O (SIGIO)

⑤ 异步 I/O

五种I/O 模式


IO模型
同步
(synchronous)
异步
(asynchronous)
阻塞(blocking)
非阻塞(non-blocking)
多路复用
(multiplexing)
信号驱动
(signal driven)
异步
(asynchronous)

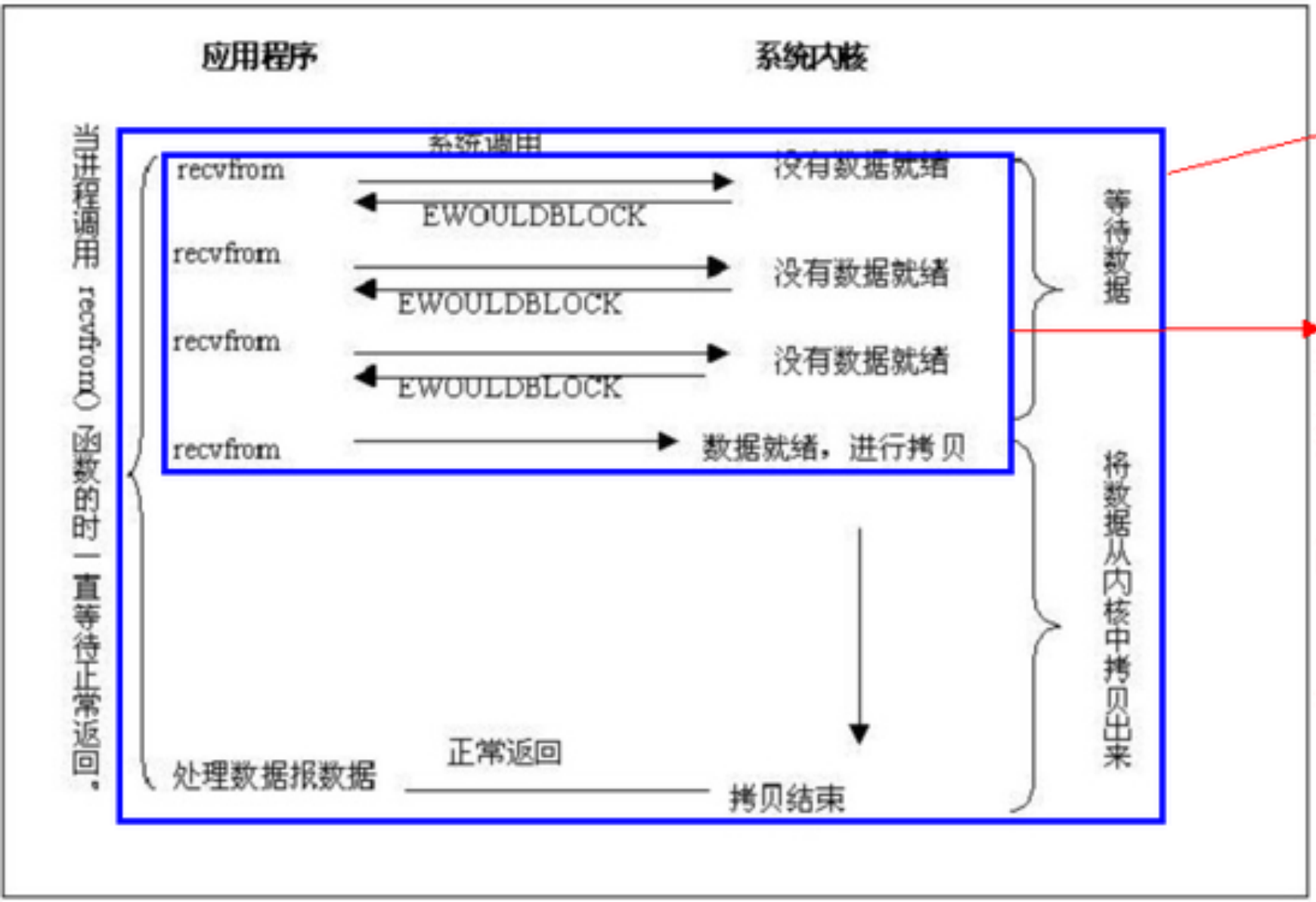
阻塞 I/O模式


应用程序
系统内核
当进程调用 recvfrom() 函数的时候阻塞。
系统调用
recvfrom
没有数据就绪
等待数据
这段时间是让出CPU的。
使用阻塞模式，没有数据时。
进程阻塞，让出CPU，进入休息队列。
等有数据是，内核会唤醒进程读取数据。
即：拷贝数据的过程。
数据就绪，进行拷贝
将数据从内核中拷贝出来
正常返回
处理数据报数据
拷贝结束

非阻塞 I/O模式

```
while true {
    for i in stream[] {
        if i has data
        read until unavailable
    }
}
```

放在一个大的循环中。



采用轮询的方法，测试是否有数据可以处理(读)。

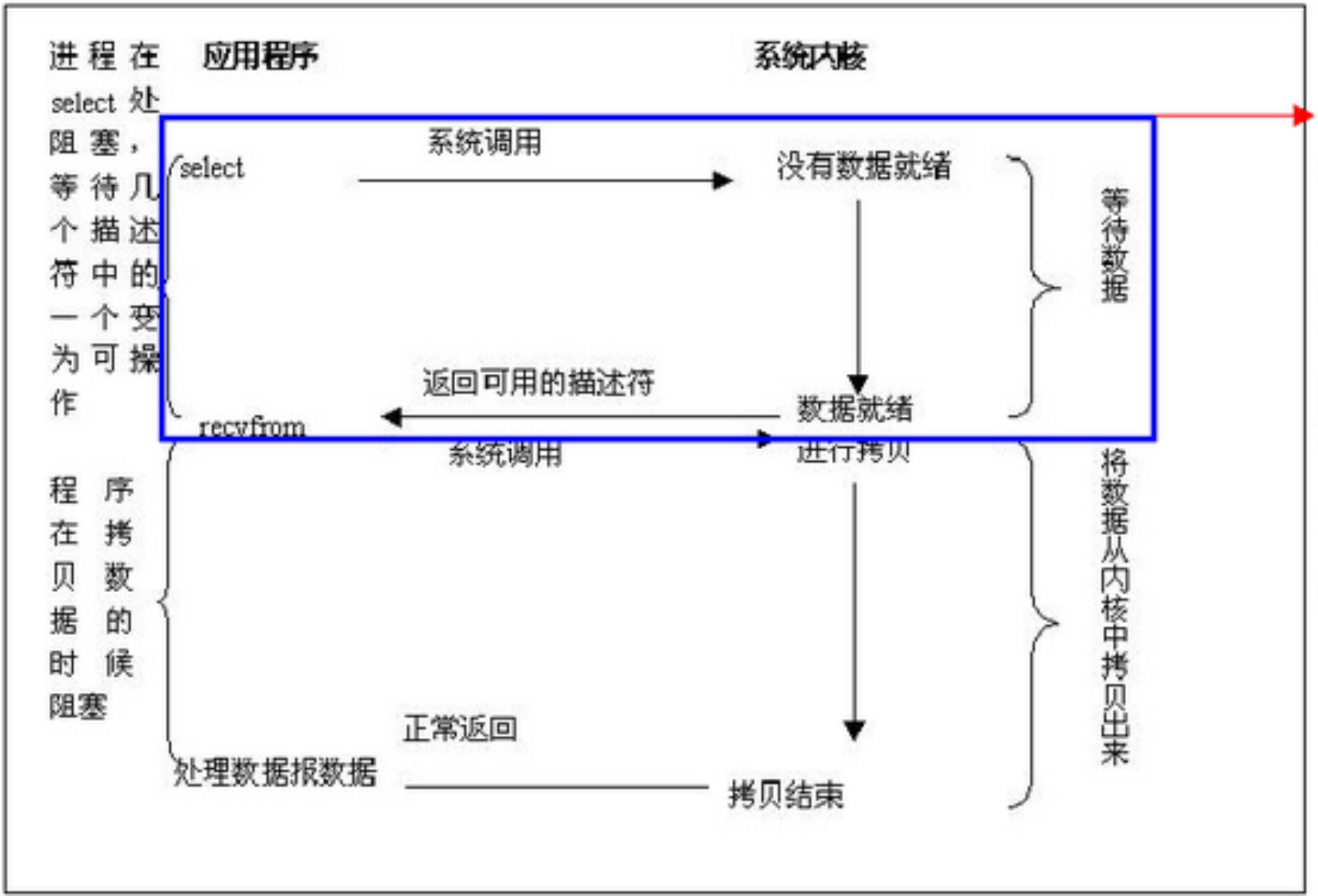
即不段的recvfrom，这个时候，进程占用这CPU，在不短的测试数据有没有准备好。

由于非阻塞，所以，应用程序采用轮询方式，不断测试，浪费了大量的CPU资源。

所以非阻塞方式并不常用。

但他在实时性方面，确实比较厉害。

支持I/O复用的系统调用




```
while true {
    select(streams[])
    for i in streams[] {
        if i has data
     read until unavailable
    }}
```

调用select,poll或epoll等系统调用，
等待指定的时间，

从批量被监控的描述符中，

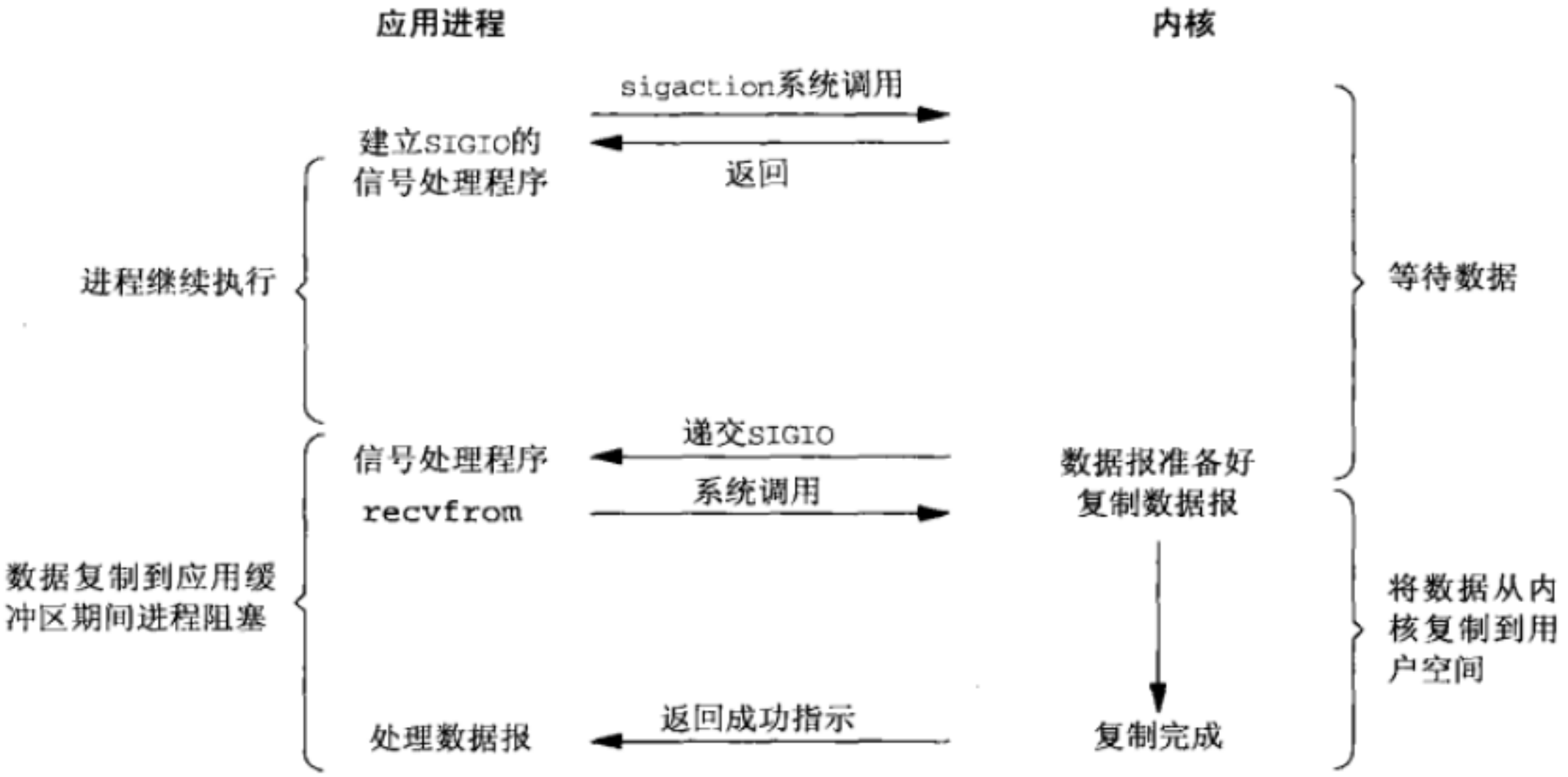
获取可以操作的描述符，（即：数据准备好的描述符）

然后，下面对这些描述符进行操作时，因为有数据，所以不会出现阻塞。

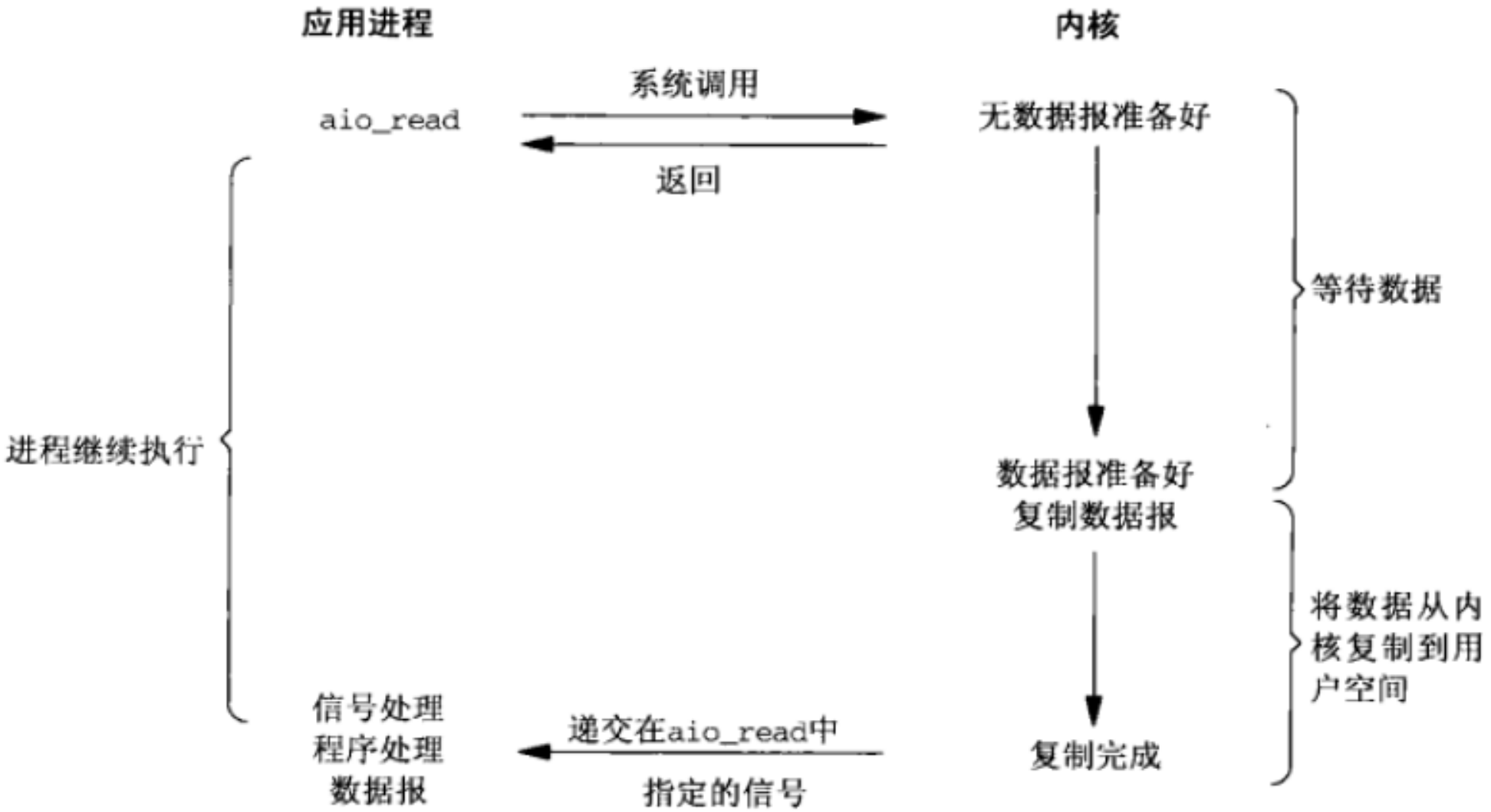
充分说明了::
I/O多路复用，应用于大量文件描述符的场景，不使用于单个文件描述符的场景。

```
while true {
    active_stream[] = epoll_wait(epollfd
    for i in active_stream[] {
        read or write till
    }
}
```

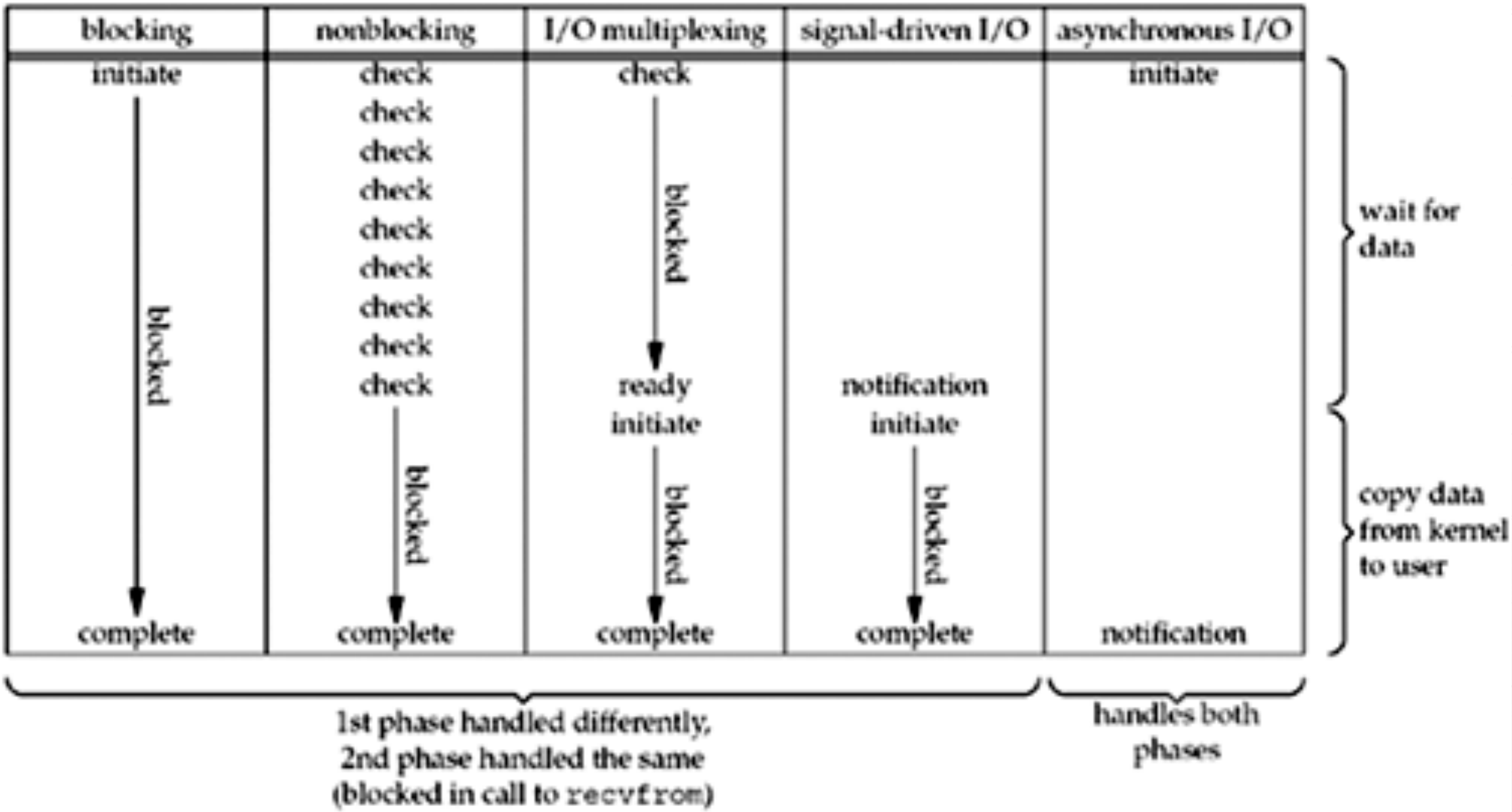
信号驱动IO模型


应用进程
内核
sigaction系统调用
建立SIGIO的
信号处理程序
返回
进程继续执行
等待数据
递交SIGIO
信号处理程序
数据报准备好
系统调用
recvfrom
复制数据报
数据复制到应用缓
冲区期间进程阻塞
将数据从内
核复制到用
户空间
返回成功指示
处理数据报
复制完成

异步IO模型


应用进程
内核
系统调用
aio_read
无数据报准备好
返回
等待数据
进程继续执行
数据报准备好
复制数据报
将数据从内核复制到用户空间
信号处理程序处理数据报
递交在aio_read中指定的信号
复制完成

5种IO模型的比较


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

- **同步与异步IO**
  - **Posix**定义这两个术语如下
    - 同步**I/O**操作引起请求进程阻塞，知道**I/O**操作完成；
    - 异步**I/O**操作不引起请求进程阻塞；
  - **IO的两个阶段都没有被阻塞，才能被称为异步IO**
  - 因此，阻塞**I/O**模型、非阻塞**I/O**模型、 **I/O**复用模型和信号驱动模型都是同步**I/O**模型；最后一种才是异步**IO**
- **实际编程中，这几种IO模型经常根据需要混用**
  - 首先要避免被阻塞，但是阻塞都不好？
  - 在事件的检查点要阻塞
  - 避免采用信号机制，主要原因是信号是异步的，将使程序变得很难编写。

IO复用与Select函数

- IO复用技术能够使能应用程序同时等待多个事件的发生
- IO复用技术的引入，使得网络编程的模式发生了巨大变化
- 常用的IO复用函数有
  - select/poll/pselect
  - epoll (Linux)
  - kqueue (FreeBSD)

SELECT

- **int select (int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);**
- **select** 函数监视的文件描述符分**3**类，分别是**writefds**、**readfds**、和**exceptfds**。调用后**select**函数会阻塞，直到有描述副就绪（有数据 可读、可写、或者有**except**），或者超时（**timeout**指定等待时间，如果立即返回设为**null**即可），函数返回。当**select**函数返回后，可以 通过遍历**fdset**，来找到就绪的描述符。
- **select**目前几乎在所有的平台上支持，其良好跨平台支持也是它的一个优点。**select**的一个缺点在于单个进程能够监视的文件描述符的数量存在最大限制，在**Linux**上一般为**1024**或**2048**，可以通过修改宏定义甚至重新编译内核的方式提升这一限制，但 是这样也会造成效率的降低。

select()函数

```
#include <sys/select.h>
#include <sys/time.h>
struct timeval  {
  long    tv_sec;          /* seconds
 */
  long      tv_usec;              /*
 microseconds */
};
```

```
int select
(
 int maxfdp1,
 fd_set *readset,
 fd_set *writeset,
 fd_set *exceptset,
 const struct timeval *timeout
);
Returns:
positive count of ready descriptors, 0 on
 timeout
-1 on error
```

- **readset/writeset/exceptset**为读/写/异常集合，这些集合分别表示我们关注的读/写/异常事件集合
  - 这些集合内包含了某个文件描述符上读/写/异常事件
  - 集合通常是以**bitmap**的形式实现
  - 例如：
  - 一个读集合**{1,4,7}**表示我们对文件描述符为**1**，**4**，**7**的文件上的读事件感兴趣
  - **{1,4,7}**集合的**bitmap**为：**1001 0010B**

- **maxfdp1**表示读/写/异常集合中，最大描述符**+1**
  - **+1，why？**
  - 集合通常以**bitmap**实现，这些**bitmap**都有一个固定的大小
  - 为了加快**bitmap**中搜索，所以给出了一个搜索的上限，不然的话需要枚举**bitmap**中所有**bit**位，这个上限就是**maxfdp1**的作用
  - 注意到文件描述符是以**0**开始的，所以上限应该是所有集合中最大描述符**+1**

- **timeout**是一个时间间隔
  - 当在**timeout**的时间间隔内没有事件发生，**select**会退出阻塞态
  - 通常**Unix**的时间分片设计为**10ms**，所以**select**一般最小的粒度为**10ms**，设定更高的时间精度没有意义

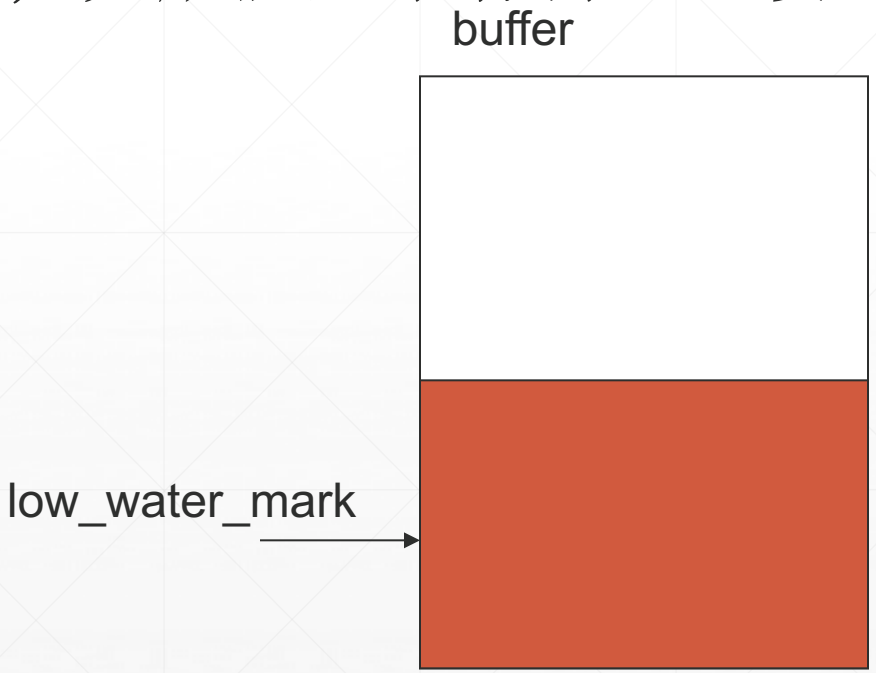
- 由于引入是**timeout**，所以**select**函数的动作有三种
  - 当**timeout**指针为**NULL**，**select**会一直等待事件发生
  - 当**timeout**为一个非**0**的设定值，并且在这个时间范围内没有事件发生，**select**函数在时间到时返回
  - 当**timeout**设定为**0**，**select**不阻塞，直接返回，这称为**poll**

- bitmap集合的操纵宏

```
#define FD_SET(fd, fdsetp)   __FD_SET (fd, fdsetp)
#define FD_CLR(fd, fdsetp)   __FD_CLR (fd, fdsetp)
#define FD_ISSET(fd, fdsetp) __FD_ISSET (fd, fdsetp)
#define FD_ZERO(fdsetp)      __FD_ZERO (fdsetp)
```

  - 注意：在向**select**传递读/写/异常集合之前一定要调用**FD_ZERO**对集合清零
  - 另外，如果对读/写/异常事件不感兴趣，可以向**select**函数传递**NULL**指针

- 读事件
  - 当收到的数据字节数超过 **buffer** 的 **low_water_mark**
  - 对端发送**FIN**，**read**将返回**0**
  - 有新的客户连接上来，完成队列中有新建连接
  - **socket**出错



- 写事件
  - 写**buffer**中的空闲空间超过**low_water_mark**，并且是**1**）是已经建连；**2**）或者是无连接的
  - 写半关闭，触发写事件。如果继续通过该**socket**的文件描述符向对端写数据，会触发写事件，同时也会触发**SIGPIPE**信号
  - 在非阻塞条件下，**connect**建连成功或建连失败
  - **socket**出错
    - **socket**出错会同时出现在读事件/写事件集合中

- 异常事件
  - 接收到带外数据
  - 接收到伪终端传递过来的控制字符

select函数事件触发表

Condition	readable	writable	exception
buffer内数据可读	yes		
读半关闭	yes		
可以accept	yes		
buffer空闲空间可写		yes	
写半关闭		yes	
非阻塞下connect成功		yes	
socket错误	yes	yes	
收到带外数据			yes
收到伪终端控制命令			yes

select/poll

- select函数的主要缺陷
  - FD_SETSIZE
    - <linux/posix_types.h>

```
#undef __FD_SETSIZE
#define __FD_SETSIZE    1024
```

  - fd_set需要拷贝到内核，更关键的是：fd_set将进程的所有描述符都罗列出来，内核与应用程序需要逐个检查相应的bit位。这个复杂度为O(N)。在一次select设定/响应过程，fd_set至少要经过三次这种枚举才能完成遍历。
  - select关注的事件类型较少，其它类型的事件呢？如文件删除、进程启动...
  - Select每次从阻塞态退出后，都需要重新设置一次读/写集合

poll()函数

- int poll (struct pollfd *fds, unsigned int nfds, int timeout);
- 不同于select使用三个位图来表示三个fdset的方式，poll使用一个 pollfd的指针实现。

```
struct pollfd {
    int fd; /* file descriptor */
    short events; /* requested events to watch */
    short revents; /* returned events witnessed */
};
```

- **poll呢？**
  - 情况好点，它没有**FD_SETSIZE**的限制，但是效率仍然很低
- **综上，select、poll对于重负荷的服务器而言，效率比较低**
- **虽然select/poll存在上述弊端，但是select/poll提供了一种全新的编程模型**
  - 对网络报文的收发，采用了事件触发的方式
  - 推而及广，**select**提供了一种**IO**事件触发的编程方式
  - **select**的这种模型非常高效，可以在一个线程里支撑起重负荷的应用
  - **select**的这种复用方式，使得它成为进一步改进的对象

SELECT&POLL的缺点

- 支持的文件描述符数量太小了，默认是1024或2048
- 每次调用select&poll都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大
- 每次调用select&poll，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大

高级polling技术

- 高级polling技术没有标准，各个操纵系统的实现不一样
  - Solaris的/dev/poll / FreeBSD的kqueue / Linux的epoll技术
- 本章以epoll为例介绍高级polling技术
- epoll的特点
  - epoll是对poll的改进
  - 增量的事件添加与删除
  - 支持边沿触发（edge trigger、ET）和水平触发（level trigger、LT）
  - 同步的事件处理
  - 基于文件描述操作，epoll是一个特殊的文件

EPOLL

- Epoll是Linux内核为处理大批量句柄而作了改进的poll，是Linux下多路复用IO接口select/poll的增强版本，它能显著减少程序在大量并发连接中只有少量活跃的情况下的系统CPUs利用率。

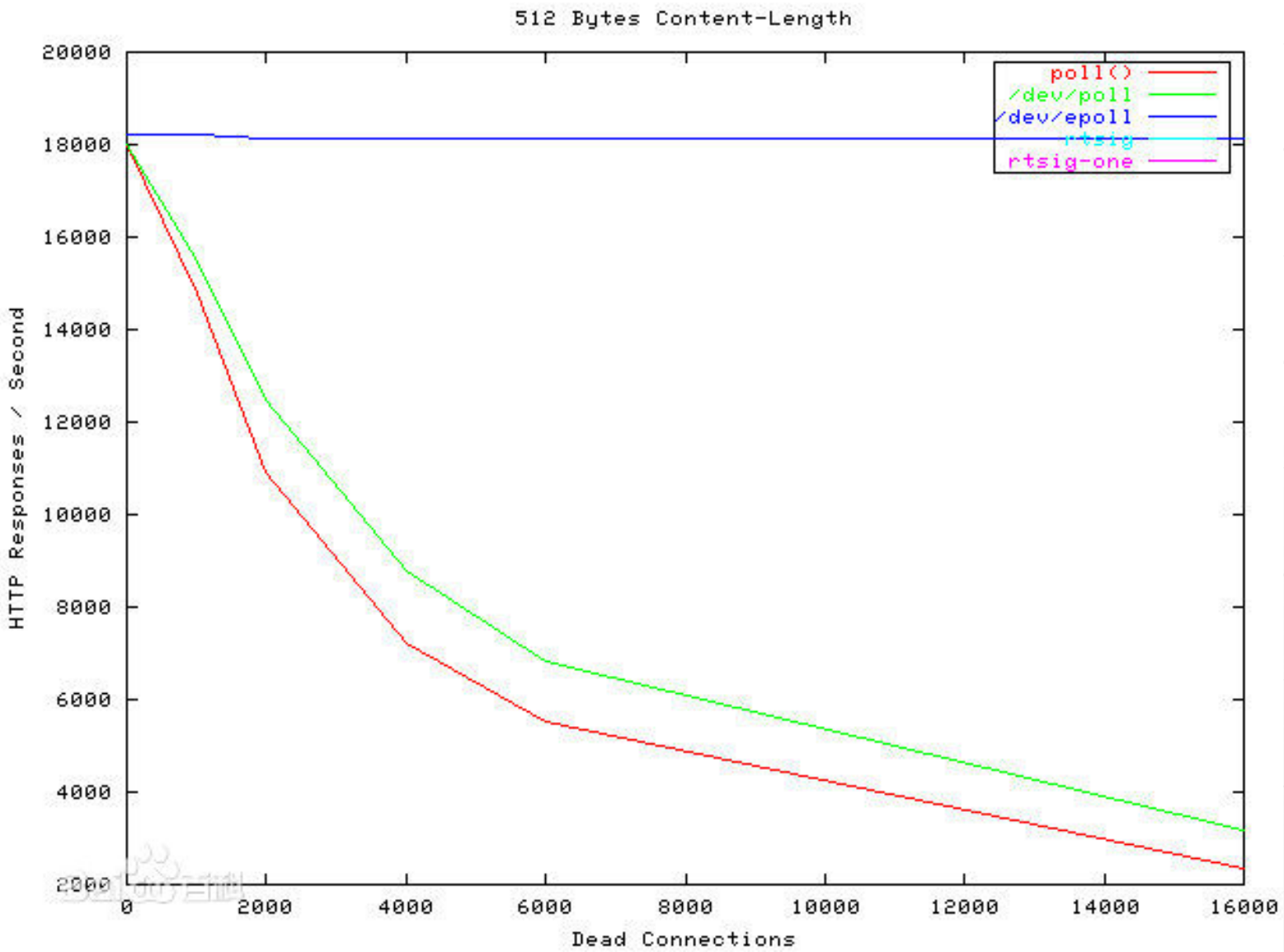
EPOLL使用举例

- 设想一下如下场景：
  - 有100万个客户端同时与一个服务器进程保持着TCP连接。而每一时刻，通常只有几百上千个TCP连接是活跃的(事实上大部分场景都是这种情况)。如何实现这样的高并发？
  - 在select/poll时代，服务器进程每次都把这100万个连接告诉操作系统(从用户态复制句柄数据结构到内核态)，让操作系统内核去查询这些套接字上是否有事件发生，轮询完后，再将句柄数据复制到用户态，让服务器应用程序轮询处理已发生的网络事件，这一过程资源消耗较大，因此，select/poll一般只能处理几千的并发连接。

Epoll的优点

- 能支持打开更大数目的socket描述符，select一般限制为1024或2048
- 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

Select，Poll，Epoll性能对比



Epoll的操作调用

- epoll_create
- epoll_ctl
- epoll_wait

epoll_create

- **epoll_create(int size);**
  - 创建一个**epoll**的句柄，**size**用来告诉内核这个监听的数目一共有多大。需要注意的是，当创建好**epoll**句柄后，它就是会占用一个**fd**值，在**linux**下如果查看**/proc/**进程**id/fd/**，是能够看到这个**fd**的，所以在使用完**epoll**后，必须调用**close()**关闭，否则可能导致**fd**被耗尽。

epoll_ctl

- epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
- epoll_ctl是事件注册函数，它不同与select()是在监听事件时告诉内核要监听什么类型的事件，而是在这里先注册要监听的事件类型。
- 第一个参数是epoll_create()的返回值，
- 第二个参数表示动作，用三个宏来表示：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；
- 第三个参数是需要监听的fd，第四个参数是告诉内核需要监听什么事，

struct epoll_event结构如下

```
struct epoll_event {
  __uint32_t events;
/* Epoll events */
  epoll_data_t data;
/* User data variable */
};
```

epoll_wait

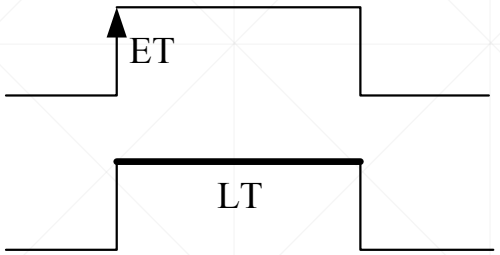
- epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
- 等待事件的产生，类似于select()调用。
- 参数events用来从内核得到事件的集合maxevents告之内核events有多大，这个maxevents的值不能大于创建epoll_create()时的size，参数timeout是超时时间（0会立即返回，-1将不确定，也有说法说是永久阻塞）。
- 该函数返回需要处理的事件数目，如返回0表示已超时

Epoll的事件类型

- EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；
- EPOLLOUT：表示对应的文件描述符可以写；
- EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；
- EPOLLERR：表示对应的文件描述符发生错误；
- EPOLLHUP：表示对应的文件描述符被挂断；
- EPOLLET： 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。
- EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

Epoll的的两种工作模式

- LT模式
- ET模式



- **ET与LT**
  - 在**ET**模式下，事件只在**epoll**返回时通知应用程序。即是说，如果一次不干净地处理完通知事件，以后该事件再也不会通知应用程序
  - 而在**LT**模式下，应用程序可以只处理一部分事件，剩下的事件，在下次**epoll**返回后，仍然还会通知用户
  - 例子
    - 一个**pipe**，**writer**向其中写入**4KB**
    - **reader**只读取**1KB**
    - 在**ET**模式下，剩下的**3KB**将不再通知
    - 而在**LT**模式下，该事件仍然会通知应用程序
- **ET的效能更高，但是对程序员的要求也更高。在ET模式下，我们必须一次干净而彻底地处理完所有事件**

LT模式

- LT（level triggered）是缺省的工作方式，并且同时支持block和no-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。传统的select/poll都是这种模型的代表。

ET模式

- ET模式是种高速工作方式，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，直到你做了某些操作导致那个文件描述符不再为就绪状态了.
- 例如:客服端发送了一个10KB的数据，这时服务端收到一个读取事件但只读取了5KB的数据也就是说缓存区还有5kB未读取但因为采用的是ET模式服务端不会再收到一个读取事件这就导致还有5KB的数据不会被读取.

ET模式

- 读数据的时候要考虑当recv()返回的大小,如果等于请求的大小，那么很有可能是缓冲区还有数据未读完，也意味着该次事件还没有处理完，所以还需要再次读取。

- ```
  while(rs)
  {
    buflen = recv(activeevents[i].data.fd, buf, sizeof(buf), 0);
    if(buflen < 0)
    {
      /* 由于是非阻塞的模式,所以当errno为EAGAIN时,表示当前缓冲区        已无数据可读 */
      if(errno == EAGAIN)
        break;
      else
        return;
    }
    else if(buflen == 0)
    {
      /* 对方端口的socket已正常关闭*/
    }
    if(buflen == sizeof(buf)
      rs = 1;   /* 需要再次读取 */
    else
      rs = 0;
  }
  ```
  ```

编程注意事项

1. 使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。
2. epoll工作在ET模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。