目录

[Dpdk学习总结 2](#_Toc90235160)

[一、DPDK线程初始化 2](#_Toc90235161)

[二、recvmsg与sendmsg 2](#_Toc90235162)

[基本定义 2](#_Toc90235163)

[数据结构 3](#_Toc90235164)

[操作宏 4](#_Toc90235165)

[函数使用 4](#_Toc90235166)

[处理流程 4](#_Toc90235167)

[参数与返回值 5](#_Toc90235168)

[flags 5](#_Toc90235169)

[Sendmsg 5](#_Toc90235170)

[Recvmsg 6](#_Toc90235171)

[二、dpdk中断机制 6](#_Toc90235172)

[中断初始化 6](#_Toc90235173)

[中断注册 6](#_Toc90235174)

Dpdk学习总结

一、DPDK线程初始化

1. 在每个slave核上创建线程。

RTE\_LCORE\_FOREACH\_SLAVE(lcore\_id) {

rte\_eal\_remote\_launch(user\_loop, param, lcore\_id);

}

2. 在除了master核外其他核上创建线程。

rte\_eal\_mp\_remote\_launch(user\_loop, param, SKIP\_MASTER);

创建线程实际时唤醒对应lcore的worker线程，通过cpu\_layout.py可以查看当前cpu的核心分布。

二、recvmsg与sendmsg

基本定义

#include <sys/types.h>

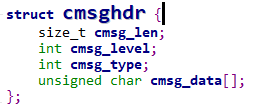
#include <sys/socket.h>

ssize\_t sendmsg(int sockfd, const struct msghdr \*msg, int flags);

ssize\_t recvmsg(int sockfd, struct msghdr \*msg, int flags);

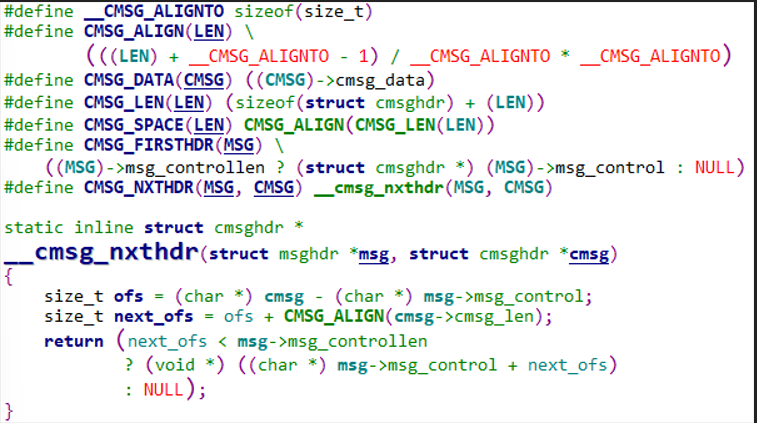
支持一般数据的发送和接收，还支持多缓冲区的报文发送和接收，还可以在报文中带辅助数据。这些功能是常用的**send、recv等接口无法完成的**。

## 数据结构



1. **msg\_name** 主要用于保存当前使用的协议的地址，比如使用了tcp协议、udp协议、UNIX domain协议等
2. **msg\_namelen**指明了p这个套接字地址的长度
3. 多缓冲区的发送和接收处理就是通过一个struct iovec的数组，每个成员的**io\_base**都指向了不同的buffer的地址。struct iovec中的**iov\_len**是指该buffer中的数据长度。
4. struct msghdr中的**msg\_iovlen**是指buffer缓冲区的个数，即iovec数组的长度
5. **msg\_control**通常指向一个控制消息头部
6. **cmsg\_len**表示辅助数据的字节数，包含这个cmsghdr结构体头部大小，加上cms\_data数据区域的长度，由CMSG\_LEN()宏计算得到。
7. **cmsg\_level**主要包含IPPROTO\_IP(IPv4)、IPPROTO\_IPV6(IPv6)、SOL\_SOCKET(Unix Domain)。
8. **cmsg\_type**是根据上述的类型有分别有不同的内容，比如SOL\_SOCKET中主要包含以下两个：SCM\_RIGHTS(发送接收描述符)，SCM\_CREDS(发送接收用户凭证，是一个包含证书信息的结构)。
9. msg\_flags，主要用于在接收消息的过程中用来获取消息的相关属性

## 操作宏



1. **CMSG\_FIRSTHDR**得到第一个辅助数据的cmsghdr结构。如果不存在辅助数据，则返回NULL
2. **CMSG\_NXTHDR**得到当前辅助数据的下一个cmsghdr结构。如果不存在下一个，则返回NULL。
3. **CMSG\_LEN**计算cmsghdr及其附属数据的大小。
4. **CMSG\_SPACE**计算cmsghdr及其附属数据的大小，以及对其字段和可能的结尾填充字段的大小。这个宏与CMSG\_LEN()的区别是，CMSG\_LEN()计算得到的值并不包含结尾的填充字段大小。CMSG\_SPACE()宏对于确定所需要的缓存大小非常有用。需要注意的是，如果有多个辅助数据，一定要同时使用多个CMSG\_SPACE()宏进行计算。
5. **CMSG\_DATA**返回cmsghdr头后面的辅助数据的地址

## 函数使用

#include <sys/types.h>

#include <sys/socket.h>

ssize\_t sendmsg(int sockfd, const struct msghdr \*msg, int flags);

ssize\_t recvmsg(int sockfd, struct msghdr \*msg, int flags);

**发送时**不使用msghdr中的msg\_flags，直接使用传入的flags，**接收时**使用msg\_flags，此时会先将flags复制到msg\_flags，由内核读取进行接收处理。

## 处理流程

1. 两个进程之间创建unix socket，父子进程可以用socketpair实现流管道，各自拥有一个fd。普通进程之间使用基本的unix socket，服务端（bind、listen、accept）返回套接字，客户端connect返回套接字，相互关联。
2. 发送进程调用函数打开一个新描述符。
3. 发送进程创建msghdr数据结构，将待传递的描述符作为辅助数据发送，调用sendmsg跨越1中获得的套接字发送描述符。完成后关闭新描述符不影响接收进程描述符，每次发送，新描述符的引用计数+1.
4. 接收进程调用1中获得的fd（可能与发送进程的fd数值不同，与open对同一文件打开原理一样）。在接收过程中如果数据不带报文，无法分辨是接收结束或只有辅助数据，因此发送辅助数据的时候至少带一个字节数据，该数据不做处理。

## 参数与返回值

### flags

flags可以为0，或者包含以下标记：

* **MSG\_CONFIRM**：告诉链路层，从对端收到一个成功的响应。如果链路层没有获取到该消息，则它会重新探测neighbor（如通过广播ARP包）。该标记只支持SOCK\_DGRAM和SOCK\_RAW类型的套接字有效，且目前只支持IPv4和IPv6。
* **MSG\_DONTROUTE**：表示不通过路由发送数据，只有在直连网络的情况下才发送。该标记通常只在诊断模式或者路由程序中使用。该标记只对路由家族协议定义，数据包socket不使用。
* **MSG\_DONTWAIT**：启用非阻塞操作。如果操作可能阻塞，则返回EAGAIN 或 EWOULDBLOCK。该选项也可以使用函数fcntl()使用O\_NONBLOCK标记实现。
* **MSG\_EOR**：终止一条记录（如果该标记支持，则对应的套接字类型为SOCK\_SEQPACKET）。
* **MSG\_MORE**：调用者有更多数据要发送，该标记与TCP套接字一起使用，可以达到TCP\_CORK类型套接字选项相同的效果，不同的是该标记可以在每次调用时才设置。从linux内核2.6开始，该标记也支持UDP套接字，会通知内核将所有通过该标记发送的数据打包成一个datagram，当函数调用没有指定该标记时，一次性把这些数据发送出去。（可以参考UDP\_CORK套接字类型）
* **MSG\_NOSIGNAL**：当另一端断开链接时，当面向流的套接字发生错误时，请求不发送SIGPIPE信号。EPIPE错误仍被返回。
* **MSG\_OOB**：在支持该选项套接字上（如SOCK\_STREAM类型）发送Out-of-band数据，底层的协议也必须支持out-of-band数据。

### Sendmsg

成功返回发送字节数，否则返回-1.设置errno。

### Recvmsg

成功返回接收的字节数，如果消息长度大于缓存，多余的数据可能丢弃（socket类型决定）。

如果socket没有数据会一直等待，除非设置为非阻塞模式。非阻塞模式下无数据会返回-1，errno设置EAGAIN或EWOULDBLOCK，如果对端关闭返回0.

二、dpdk中断机制

dpdk通过线程中使用epoll模型，监听UIO设备事件，模拟系统中断处理。

## 中断初始化

rte\_eal\_init() –> rte\_eal\_intr\_init()

1. 初始化全局中断源队列TAILQ\_INIT(&intr\_sources);
2. 创建管道，用于epoll模型的消息通知pipe(intr\_pipe.pipefd)，控制中断处理线程更新监听事件。
3. 创建中断处理线程rte\_ctrl\_thread\_create(&intr\_thread, "eal-intr-thread", NULL, eal\_intr\_thread\_main, NULL);

处理函数eal\_intr\_thread\_main()，该函数的主体是一个for(;;)无限循环，在每一次循环内会创建epoll模型，遍历intr\_sources链表，注册监听事件。eal\_intr\_handle\_interrupts((pfd, numfds)监听已注册的所有UIO设备的中断事件，如果监听到中断，则调用eal\_intr\_process\_interrupts函数，调用对应UIO设备的中断处理函数

## 中断注册

int rte\_intr\_callback\_register(const struct rte\_intr\_handle \*intr\_handle,

      rte\_intr\_callback\_fn cb, void \*cb\_arg)

在网卡初始化的时候，会调用  
rte\_eth\_dev\_init() —> eth\_igb\_dev\_init() —> rte\_intr\_callback\_register()

rte\_intr\_callback\_register()函数的主要工作如下：

1. 分配回调函数内存并初始化（函数指针+入参arg）
2. 遍历中断队列，若已经注册中断，直接将回调函数注册到该中断的中断回调列表中。
3. 若没有注册该中断，则分配一个struct rte\_intr\_source，并注册更新。
4. 第一次注册中断wake\_thread置位，通知epoll\_wait重建等待列表，中断epoll\_wait死循环，使得中断处理线程重新添加监听事件。