**媒体流的处理（结束）**

原基于LINUX架构的系统移植到基于DPDK架构的系统，进行了哪些修改，罗列1，2，3，怎么解决的？

DPDK的优点：无锁队列，大数据。。。，怎么利用dpdk的优点进行了了哪些修改

1 dpdk-mbuf。

调用mbuf库进行申请mbuf的内存，内存如何分配，从核中直接拉起一段固定的大小，之后mbuf都从这里面获取，媒体流过来之后，从以太网口入口RTP处理层就进行申请mbuf，之后只是往这里面进行数据填充，最后由媒体流处理层进行mbuf的释放。这部分内存是申请在堆还是栈里面的？是固定获取，地址是连续的还是随机的？

2 媒体处理分成媒体控制消息处理和媒体流处理，其中控制消息会在这两个线程中同时调用，是如何通过无锁队列实现多线程调用处理问题的？需要看代码.

3 媒体流的具体格式有哪些

以太网头，udp头，rtp头，nbup头或者无nbup头，媒体流具体内容

4 媒体流通过封包时间控制发送媒体流的频率和长度

5 app绑定在那几个核上如何分配，DPDK的分配是连续的，需要如何分配，绑定。

示例代码怎么写的

申请mbuf 代码

解析网络报文代码 代码

多线程控制代码 代码

无锁代码实现业务消息控制 代码

通道定义

媒体流如何承载在网络报文上传输的，是怎样的格式，以怎样的速率进行封发，怎么确定的，媒体流在服务器上用了多少核，怎样的一个性能能力

在众多DPDK优势中，我关注过的是 零拷贝技术，无锁机制，多核多进程多线程技术

运行在多核处理器的服务器上，同一台服务器运行多个进程，同一个进程运行多个线程

这三个都在报文传输过程中得到实现

Rte\_mempool rte\_ring rte\_mbuf rte\_timer

零拷贝技术，（内存分配，报文传输，队列）,无锁机制,多核多进程多线程技术：



这个依存图上

我了解的是rte\_ring rte\_timer rte\_malloc rte\_mbuf rte\_mempool 这部分都在应用层

L2 mac

L3 ip

L4 udp/icmp/arp

上层应用：ping,媒体业务处理

关于UDP报文示意

网口报文上来-🡪pkt\_recv侧->udp\_recv侧-〉媒体处理侧->udp\_send侧-〉pkt\_snd侧-〉网口报文分发

**IPV6/IPV4（结束）**

相比IPV4，IPV6的优点：

1．128位的地址长度，16个字节，16X8=128位，提供几乎无限的地址空间

2．新的地址分类和地址分配方案 怎么个新法？

3．简化，高效的IP报文结构 怎么个简化法？

4．自动地址配置 怎么个自动配置？

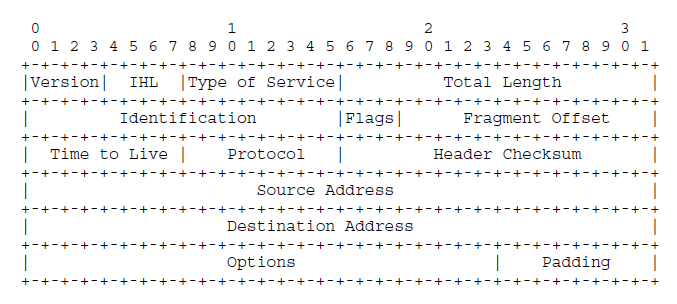
5．用组播代替广播又是怎么个意思？

6．更好的支持移动性

IPV6长度 128位

IPV4 32位？

* **IPv4报文格式**



IPV4头 20字节

IPV6格式

完全形式 类似 3FFE:FFFF:7654:1234:BA98:3210:4562

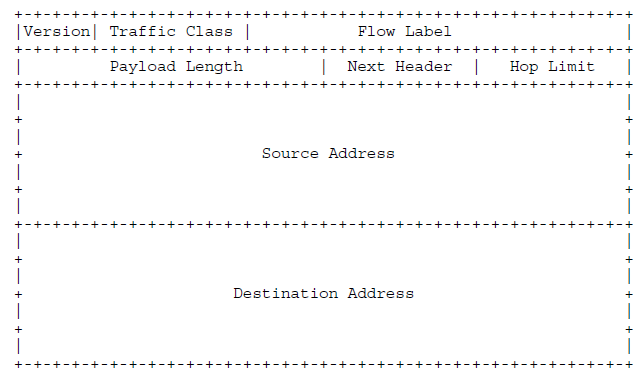
压缩形式 类似 FFED:0:0:0:0:BA97:3210:4562 压缩成 FFED::BA98:3210:4562

(::)只能出现一次

混合形式 IPV4嵌入IPV6中 n:n:n:n:n:n:d.d.d.d

IPV6地址指定给网络接口，而不是主机，地址分为单播，多播和任意播。

Ipv6



Ipv6头字节长度40字节，内容：

Version 4位

Traffic Class 8位 通信类别及优先级

Flow Label 20位新增字段 标记需要特殊处理的数据流

PayloadLength 16位负载长度 包括首部和上层PDU

NextHeader 8位 识别 紧跟IPv6头后的首部类型，如某个传输层协议头（诸如TCP，UDP或者ICMPV6，SCTP等）

HopLimit 8位，用包转发生命期

SourceAddress 128位

DestinationAddress128位

我涉及到的只有 控制消息的转发；

解析从要跳转 NEXTHEADER 跳转到下一个消息头紧连的部分，

Nextheader 是udp，那就是下一个消息头紧连的部分是udp

网络报文解析

第一步 14字节的以太网头

第二步 20字节的Ip报文头 / 40字节的IP报文头

第三步 UDP报文

第四步 RTP报文

**SIP协议（未完成今天下午）**

基本呼叫的流程，补充业务呼叫的流程，网域之间的拼接

注册，呼叫，订阅等

呼叫 invite 180 200 reinvite bye

订阅 subscribe

注册 register

关键字

头域关联 确认同一个呼叫一张话单XDR

Callid

回家找，忘记找了，苍天啊

Sdcp

Sip过来第一步 粗解析，分发到不同的线程，进行解析，分发到不同流程处理模块，通过XX进行哈希关联，

1 消息过来的分发过程，关联过程，哈希过程，合成xdr,,输出。

**写代码上的内容**

**1 头文件中使用**

**#ifndef \_SYSDATA\_H**

**#define \_SYSDATA\_H**

**#endif**

**目的：**防止头文件被多次包含的预处理技术

在编译项目时，编译每个.cpp文件（通常）。简单来说，这意味着编译器会把你的.cpp文件，打开任何文件#included，将它们全部连接成一个海量文本文件，然后执行语法分析，最后将它转换成一些中间代码，优化/执行其他任务，最后生成目标架构的汇编输出。因此，如果#included一个.cpp文件下的文件多次，则编译器将附加文件内容两次，因此如果该文件中有定义，你会收到一个编译器错误，告诉你重新定义了一个变量。FILE\_H当编译过程中的预处理器步骤处理文件时，首次到达其内容时，前两行将检查是否已为预处理器定义。如果没有，它将定义FILE\_H并继续处理它和指令之间的代码#endif。下一次该文件的内容被预处理器看到时，检查FILE\_H将是假的，所以它将立即扫描#endif并继续。这样可以防止重新定义错误。它将定义并继续处理它和指令之间的代码。

**2 #Pragma Pack(n)与内存分配**

之所以用这个，是因为似乎从32位移到64位，出现了对齐问题，导致程序出现了异常？

解释一：

每个特定平台上的编译器都有自己的默认“对齐系数”(也叫对齐模数)。程序员可以通过预编译命令#pragma pack(n)，n=1,2,4,8,16来改变这一系数，其中的n就是你要指定的“对齐系数”。

　　规则：

　　1、数据成员对齐规则：结构(struct)(或联合(union))的数据成员，第一个数据成员放在offset为0的地方，以后每个数据成员的对齐按照#pragma pack指定的数值和这个数据成员自身长度中，比较小的那个进行。

　　2、结构(或联合)的整体对齐规则：在数据成员完成各自对齐之后，结构(或联合)本身也要进行对齐，对齐将按照#pragma pack指定的数值和结构(或联合)最大数据成员长度中，比较小的那个进行。

解释二：

n 字节的对齐方式 VC 对结构的存储的特殊处理确实提高 CPU 存储变量的速度，但是有时候也带来 了一些麻烦，我们也屏蔽掉变量默认的对齐方式，自己可以设定变量的对齐方式。 VC 中提供了#pragma pack(n)来设定变量以 n 字节对齐方式。n 字节对齐就是说 变量存放的起始地址的偏移量有两种情况：

第一、如果 n 大于等于该变量所占用的字 节数，那么偏移量必须满足默认的对齐方式。

第二、如果 n 小于该变量的类型所占用 的字节数，那么偏移量为 n 的倍数，不用满足默认的对齐方式。结构的总大小也有个 约束条件，分下面两种情况：如果 n 大于所有成员变量类型所占用的字节数，那么结 构的总大小必须为占用空间最大的变量占用的空间数的倍数； 否则必须为 n 的倍数。

下面举例说明其用法。 #pragma pack(push) //保存对齐状态

 #pragma pack(4)//设定为 4 字节对齐

struct test { char m1; double m4; int m3; }; #pragma pack(pop)//恢复对齐状态 以上结构体的大小为 16：

下面分析其存储情况，首先为 m1 分配空间，其偏移量 为 0，满足我们自己设定的对齐方式（4 字节对齐），m1 大小为 1 个字节。接着开始 为 m4 分配空间，这时其偏移量为 1，需要补足 3 个字节，这样使偏移量满足为 n=4 的倍数（因为 sizeof(double)大于 4）,m4 占用 8 个字节。接着为 m3 分配空间，这时 其偏移量为 12，满足为 4 的倍数，m3 占用 4 个字节。这时已经为所有成员变量分配 了空间，共分配了 16 个字节，满足为 n 的倍数。如果把上面的#pragma pack(4)改为 #pragma pack(8)，那么我们可以得到结构的大小为 24。

大家看了这些文字描述头也一定会发麻吧，我坚持读完后，然后自己编写了一个程序：

#pragma pack(4)

struct node{

  int e;  
  char f;  
  short int a;  
  char b;

};

struct node n;

printf("%d\n",sizeof(n));

我自己算的结果是16，结果实际结果是：



然后结构体内部数据成员变动一下位置：

#pragma pack(4)

struct node{

  char f;  
  int e;  
  short int a;  
  char b;};

struct node n;

printf("%d\n",sizeof(n));



将对齐位数强制定位2

#pragma pack(2)

struct node{

  char f;  
  int e;  
  short int a;  
  char b;};

struct node n;

printf("%d\n",sizeof(n));



将对齐位数强制定位1

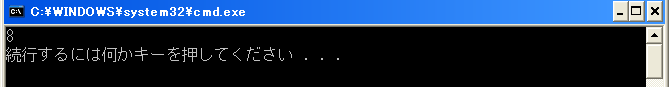
#pragma pack(1)

struct node{

  char f;  
  int e;  
  short int a;  
  char b;};

struct node n;

printf("%d\n",sizeof(n));



看着输出结果和文字描述有点晕，下面简单说一下俺的判定规则吧：

其实之所以有内存字节对齐机制，就是为了最大限度的减少内存读取次数。我们知道CPU读取速度比内存读取速度快至少一个数量级，所以为了节省运算花费时间，只能以牺牲空间来换取时间了。

下面举例说明如何最大限度的减少读取次数。

#pragma pack(1)

struct node{

  char f;  
  int e;  
  short int a;  
  char b;};

struct node n;

printf("%d\n",sizeof(n));

这里强制按照1字节进行对齐，可以理解成所有的内容都是按照1字节进行读取（暂且这样理解，因为这样可以很好的理解内存对其机制），其他所有的数据成员都是1字节的整数倍，所以也就不用进行内存对其，各个成员在内存中就按照实际顺序进行排列，结构体实际长度为8

#pragma pack(2)

struct node{

  char f;  
  int e;  
  short int a;  
  char b;};

struct node n;

printf("%d\n",sizeof(n));

这里强制按照2字节进行对齐。如果内存分布仍然是连续的话，那么int e就得三次才能读到CPU中，所以为了“讲究”int e的读取，所以在char f之后预留1BYTE，最后的char b也是如此，所以长度为10

#pragma pack(4)

struct node{

  char f;  
  int e;  
  short int a;  
  char b;};

struct node n;

printf("%d\n",sizeof(n));

这里强制按照4字节进行对齐。所以char f后要预留3BYTE，而short int a 和 char b可以一次读取到CPU(按照4字节读取)，所以长度为12

如果#pramga pack(n)中的n大于结构体成员中任何一个成员所占用的字节数，则该n值无效。编译器会选取结构体中最大数据成员的字节数为基准进行对其

**3 在写程序时看到有重新封装C的库函数的，不知道原因所以上网搜了一下**

看到原因 一是方便扩展，二是保证代码安全性可靠性

接着看到了lib库，又从lib库看到了dll和lib的解释，这里记录一下

Dll 是动态链接库 dll 给的是一个描述，编译调用时（并没有内存拷贝），当程序被夹在到内存里开始运行的时候，系统会在底层创建DLL和应用程序之间的连接关系，当执行期间需要调用DLL函数时，系统才会真正根据链接的定位信息去执行DLL中的函数代码

Lib 是静态链接库把函数或过程直接链接到可执行文件中，成为可执行程序中的一部分，当多个程序调用同样的函数时，内存里就会有这个函数的多个拷贝，浪费内存资源

**4 在写程序时又看到了一写法 func(参数,…) 参数用省略号的形式，上网百度了一下**

在 ANSI C 中，如果一个函数有可变参数，那么在该可变参数前必须有一个明确定义的参数

语言中的标准函数printf便使用这种机制。在声明不确定形参的函数时，形参部分可以使用省略号“…”代替。“…”告诉编译器，在函数调用时不检查形参类型是否与实参类型相同，也不检查参数个数。

对于可变参数的函数，需要进行特殊的处理。首先需要引用 <stdarg.h> 头文件，然后利用va\_list类型和va\_start、va\_arg、va\_end 3个宏读取传递到函数中的参数值。

**例如：**

#include <stdarg.h>

void

err\_ret(const char \*fmt, ...)

{

va\_list ap;

va\_start(ap, fmt);

err\_doit(1, LOG\_INFO, fmt, ap);

va\_end(ap);

return;

}

**LINUX-调试命令，搭建（未完成）**

LINUX的基本调试命令

gdb

p

grep

bt

ps - if 看程序进程号

pstack 后加进程ID既可以看出 函数的调用堆栈关系。

strace命令。

gdb 前台运行 set args -f

在GDB敲RUN命令之前敲set args -f

直接敲r 7也可以前段运行p x/cx

tcpdump -i eth3 host 10.9.25.52 -w /home/dpdk.cap

ps aux|grep dpiApp

scp -r srmpApp [root@10.8.29.32:/home/zry/vsrmp/srmp-pcm-20171031](mailto:root@10.8.29.32:/home/zry/vsrmp/srmp-pcm-20171031)

查找并输出

**C-数据结构，指针（未完成 今天下午开始弄，上班不弄，更待何时 5月前搞定，5月投）**

1 按照固定格式输出文本文件；

2 根据性能进行排序选择单板

3 从话单中筛选出符合条件的结果

4 简历哈希表，便利哈希表，快速查找。

5 根据pcap报文，写Demo收到网络报文，进行便宜，解出应用报文

6 多线程

7 线程分配

8 线程调用

9 互斥处理

10 队列操作

11 指针操作

12 无锁操作

13 内存操作

14 rte\_mbuf (alloc append rtem clone 等 mbuf的结构熟悉)

15 rte\_timer(set kill)

16 rte\_ring

17 rte\_mempool

1. SDCP+ODAS程序 合成1个程序；

( 整体架构设计，详细模块设计，写代码，调试，测试）

实现 网络报文解析，多线程处理，LINUX运行

分类处理

包括 媒体流报文 / 信令报文流 分线程处理

媒体流报文实现放音 RTP流

信令报文流实现SIP 关联合成XDR功能，以及统一格式化处理；

1. dpdk基础上网络收发包业务控制处理程序；
2. C++程序使用，将第一个程序重新以c++的进行开发