实验三 利用DPDK加速DNS查询(Part 1)

小组成员及贡献

- 邓龙　　40%
- 郭金涛　　30%
- 王海涵　　30%

实验过程

本地运行SimpleDNS

按照SimpleDNS仓库中的说明进行即可。

实现利用DPDK的收发包

设计思路的变动

在之前提交的本实验的设计思路中，我们认为在代码`Part2`部分需要自己从`mbuf_pool`中为应答包分配空间。当时的考虑是：一方面，应答包长度比查询包长（事实上应答的内容是额外添加在查询包后面的），所以没办法在查询包上原地构造应答包；另一方面，由于SimpleDNS本身实现中就有多次拷贝的开销（解析过程`buffer --> msg --> buffer`的两次转换），所以重新分配应答包内存并不会增加拷贝次数。

但最终我们还是决定在查询包上原地修改为应答包。理由如下：

- `rte_mbuf`空间的预分配允许我们安全地在查询包后添加信息。关于这一点在后文有更详细的说明。
- 无需手动分配空间。
- 包头的构造更为简单。
- 能够用事实说明实验要求中的问题2。

第二和第三点使得代码更为简洁。为了适应这个思路，我们也将`build_packet`函数的参数做了相应修改，现在只需要传入一个`buf`，代表查询包头部，我们在原地将其修改为应答包头。

另外，为了让程序能在服务器上正常运行，程序需要修改为一次接收多个包。于是我们对5个代码块以外的部分也进行了一定的修改。

实现细节 - 每次循环一个包

由于具体的DNS解析是交给SimpleDNS处理的，我们只需要将利用DPDK实现原SimpleDNS中`socket`的功能。这很简单，只需要将DPDK收到的包脱去头部交给SimpleDNS即可。如果每次循环仅处理一个包，大致流程如下：

```
rte_eth_rx_burst(port, 0, &query_buf, 1);   //接收一个包
uint8_t *data_addr = rte_pktmbuf_mtod(query_buf, void *);
eth_hdr = (struct ether_hdr *)data_addr;
ip_hdr = (struct ipv4_hdr *)(eth_hdr + 1);
udp_hdr = (struct udp_hdr *)(ip_hdr + 1);
buffer = (uint8_t *)(udp_hdr + 1);  //依次脱去各层头部
```

`buffer`即为去掉上层头的DNS查询包。

为了在真实环境中使用，我们还可以检查各层头部并丢弃无关的包。也可以增加IP地址等更多检查。

```
if(eth_hdr->ether_type != htons(0x800)){
    continue;   //仅允许IP协议
}
if(ip_hdr->next_proto_id != 0x11){
    continue;   //仅允许UDP协议
}
if(udp_hdr->dst_port != htons(9000u)){
    continue;   //仅允许9000号端口
}
```

SimpleDNS还需要`nbytes`变量，标识DNS查询包大小：

```
int hdr_len = (int)(buffer - data_addr);
int nbytes = query_buf[i]->data_len - hdr_len;
```

这样就把SimpleDNS需要的数据准备完毕了。

由于我们决定原地修改，所以Part1和Part2我们什么也不需要作，SimpleDNS会在原地将解析结果准备好。（我们假设了结果不会超出mbuf预分配的空间，在后文会提到这是合理的。但是这也是危险的，有可能导致缓冲区溢出攻击，如果需要用在生产环境中需要更严格的检查）

最后，我们调整包`rte_mbuf`结构的长度，并修改调用函数修改包头。最后发送包。

```
reply_buf[i] = query_buf[i];
rte_pktmbuf_append(reply_buf[i], buflen + hdr_len - reply_buf[i]->data_len);
build_packet(data_addr, buflen + hdr_len);
rte_eth_tx_burst(port, 0, &reply_buf, 1);
```

报文的头部主要修改是交换各层头的地址和重新计算`checksum`，大部分数据不需要变化，十分简洁。

```
static void
build_packet(uint8_t *buf, uint16_t pkt_size)
{
    struct ether_hdr *eth_hdr = (struct ether_hdr *)buf;
    struct ipv4_hdr *ip_hdr = (struct ipv4_hdr *)(eth_hdr + 1);
    struct udp_hdr *udp_hdr = (struct udp_hdr *)(ip_hdr + 1);
    uint8_t *pkt_end = buf + pkt_size;
    //Part 4. 原地修改

    //ether_hdr
    uint8_t *d_addr = eth_hdr->d_addr.addr_bytes;
    uint8_t *s_addr = eth_hdr->s_addr.addr_bytes;
    memswap(d_addr, s_addr, 6);

    //ipv4
    uint32_t tmp_ip_addr;
    ip_hdr->total_length = htons((uint16_t)(pkt_end - (uint8_t*)ip_hdr));
    ip_hdr->packet_id = 0;
    ip_hdr->fragment_offset = 0;
    ip_hdr->time_to_live = 255;
    ip_hdr->hdr_checksum = 0;
    tmp_ip_addr = ip_hdr->src_addr;
    ip_hdr->src_addr = ip_hdr->dst_addr;
```

```
    ip_hdr->dst_addr = tmp_ip_addr;
    ip_hdr->hdr_checksum = rte_ipv4_cksum(ip_hdr);

    //udp
    uint16_t tmp_port;
    tmp_port = udp_hdr->src_port;
    udp_hdr->src_port = udp_hdr->dst_port;
    udp_hdr->dst_port = tmp_port;
    udp_hdr->dgram_len = htons((uint16_t)(pkt_end - (uint8_t*)udp_hdr));
    udp_hdr->dgram_cksum = 0;

    udp_hdr->dgram_cksum = rte_ipv4_udptcp_cksum(ip_hdr, udp_hdr);
}
```

这样，每个循环一个包的处理就完毕了。使用`dig`命令本地测试，即可收到返回的查询结果。

实现细节 - 批处理

然而我们需要每次循环处理多个包。由于SimpleDNS每次仍然只处理一个包，我们只需要一次收发多个包（增加对应的检查），内层循环将每个包交由SimpleDNS处理即可。

大致结构如下：

```
int nb_rx, nb_tx;   //收发包的数量
nb_rx = rte_eth_rx_burst(port, 0, query_buf, BURST_SIZE);
if(nb_rx == 0){
    continue;   //未收到包直接跳过
}
for(int i = 0; i < nb_rx; i ++){
    //...
    //对每个包query_buf[i]进行相同处理
}
nb_tx = rte_eth_tx_burst(port, 0, reply_buf, nb_rx);
if (unlikely(nb_tx < nb_rx)) {
    uint16_t buf;
    for (buf = nb_tx; buf < nb_rx; buf++)
        rte_pktmbuf_free(query_buf[buf]);
}
//发送处理完毕的包，并释放未发送的包
```

附：DPDK mbuf 空间分配的问题

实验一中问到了关于DPDK中`rte_mbuf`结构的问题。当时我们认为`rte_mbuf`分配的空间是在`rte_pktmbuf_alloc`的参数中指定的且动态分配的。在这种情况下，我们难以对收到的包大小进行修改，也就难以原地修改得到应答包。

但是，事实上DPDK为每个`rte_mbuf`所分配空间是一致的，每个`rte_mbuf`创建时，DPDK都静态的分配了一段固定大小的空间（一般是4KB），而需要改变大小时，`rte_pktmbuf_append`仅仅是简单的对`rte_mbuf`中的字段进行检查和修改。所以我们可以在不知道最终大小时，先往后写入内容，最后再修改`rte_mbuf`中`data_len`和`pkt_len`。这使得我们可以交由SimpleDNS原地生成应答包。

4KB的大小已经足够绝大部分DNS查询，在本次实验中的查询更是远不及这个大小，所以不进行检查也很安全。如果在生产环境中，应该在DNS解析过程中同时检查大小，对过大的包进行特殊处理。

另外，我们实验一报告中对`headroom`和`tailroom`的理解也有误。两段空间仅仅是为了快速的实现`prepend`和`append`功能。

测试过程

将程序放到服务器上进行测试。为了将原版SimpleDNS和DPDK实现进行对比，需要对两个实现知道解析的包数量。实验说明上给出的方案是每次解析包后直接记录并打印数量。

但是在测试中，我们发现这个方案存在一定问题。因为打印输出的效率是很低的，如果每次打印，可能会对程序性能造成影响，无法对比两个程序真正的解析效率。事实上，据我们测试，原SimpleDNS中的输出就已经对程序性能造成极大影响。去掉输出后，原SimpleDNS的性能为未去掉输出时的6倍左右。

为了解决这个问题，我们首先将两个程序循环中所有输出全部取消。并重新设计了输出计数的方式。

对于DPDK，由于`rte_eth_rx_burst`函数并不阻塞程序，而是在当前没有收到包时立即返回0，所以计数较为简单。每当收到包数为0时，打印计数即可。为了清晰起见，连续收到0个包时，仅在第一个打印计数。实现如下：

```
//初始时output_flag = 0
nb_rx = rte_eth_rx_burst(port, 0, query_buf, BURST_SIZE);
if(nb_rx == 0){
    if(output_flag == 0){
        output_flag = 1;
        printf("Send: %d, Resolved: %d\n", send_count, resolved_count);
    }
    continue;
}
output_flag = 0;
```

但是原SimpleDNS中，`recvfrom`会阻塞至收到包，这导致我们无法判段测试结束从而在合适时间打印数字。经过查询，使用`setsockopt`函数可以设置`socket`的超时时间，利用该函数，将超时时间设置为1秒。即可利用类似DPDK的方法进行计数。

设置超时时间：

```
struct timeval tv;
tv.tv_sec = 1;
tv.tv_usec = 0;
if (setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv)) < 0) {
    printf("Set Timeout Error.");
}
```

打印计数：

```
nbytes = recvfrom(sock, buffer, sizeof(buffer), 0, (struct sockaddr *)
&client_addr, &addr_len);
if (nbytes < 0){
    if(output_flag == 0){
        printf("Count: %d\n", count);
        output_flag = 1;
    }
    continue;
}
```

对于DPDK的测试，另一个问题是计数标准。我们在内存循环中依次解析了收到的包，但是其中可能会有解析失败的包。最后我们将所有包都使用`rte_eth_tx_burst`发送出去，末尾可能有发送失败的包（即可能有解析失败的包被发送，也可能有解析成功的包未被发送）。这导致成功解析的包的数量可能和成功发送的包数量不一致。我们将这两个数量都进行了统计。

最后，我们用`pktgen`每次发送10000个包，间歇性发送10次，作为一组测试。对于每个程序，我们进行了两组测试。

测试结果

SimpleDNS

测试结果如下：

```
[ldeng@localhost SimpleDNS]$ ./main
Listening on port 9000.
Count: 0
Count: 1029
Count: 2060
Count: 3121
Count: 4156
Count: 5209
Count: 5997
Count: 6756
Count: 7756
Count: 8343
Count: 9142

[ldeng@localhost SimpleDNS]$ ./main
Listening on port 9000.
Count: 0
Count: 1077
Count: 1954
Count: 3005
Count: 4067
Count: 5122
Count: 6161
Count: 6945
Count: 7996
Count: 8937
Count: 10260
```

SimpleDNS每10000个包大约能成功解析并应答1000个。

SimpleDNS with DPDK

```
[ldeng@localhost lab3]$ sudo ./build/server-main
SimpleDNS (using DPDK) is running...
Send: 0, Resolved: 0
Send: 6556, Resolved: 6556
Send: 13244, Resolved: 13244
Send: 19932, Resolved: 19932
Send: 26620, Resolved: 26620
Send: 33340, Resolved: 33340
Send: 40060, Resolved: 40060
```

```
Send: 46780, Resolved: 46780
Send: 53500, Resolved: 53500
Send: 60156, Resolved: 60156
Send: 66844, Resolved: 66844

[ldeng@localhost lab3]$ sudo ./build/server-main
SimpleDNS (using DPDK) is running...
Send: 0, Resolved: 0
Send: 6492, Resolved: 6492
Send: 13180, Resolved: 13180
Send: 19804, Resolved: 19804
Send: 26524, Resolved: 26524
Send: 33244, Resolved: 33244
Send: 39964, Resolved: 39964
Send: 46684, Resolved: 46684
Send: 46702, Resolved: 46702
Send: 53308, Resolved: 53308
Send: 60028, Resolved: 60028
Send: 66748, Resolved: 66748
```

使用DPDK后，每10000个包能处理约6500个。

性能提升十分明显。另外可以看到DPDK解析成功的包数和发送成功的包数相等。说明解析发送成功率很高。

值得注意的问题

- 要保证客户端`pktgen`使用的网卡和服务端使用的是直接相连的两块。最简单的方法是，（使用DPDK时）保证两台服务器都仅有一块网卡使用uio驱动。

问题

1. 根据你得到的性能对比测试结果，DPDK是否提升了SimpleDNS的性能?

显然，DPDK显著提升了SimpleDNS的性能。

2. 直接在查询包上原地修改，得到应答包，这样做可行吗?

可行，事实上我们的实现方案就是这样做的。