操作系统专题训练期末报告

张炯辰 2021011781

axnet 性能测量和分析

测量方法

接口层

直接构造链路层报文给接口层发送，为理论最高速率。

```
make A=apps/net/bwbench SMP=1 NET=y LOG=debug run
```

传输层

```
make A=apps/c/iperf SMP=1 NET=y BLK=y ARCH=x86_64 LOG=error run
```

在WSL中：

```
iperf3 -c 127.0.0.1 -p 5555 -R
iperf3 -uc 127.0.0.1 -p 5555 -b 1000M -l 1472 -R
```

真实网卡

测试结构：

```
   WSL     172.26.218.77    172.26.208.1   Laptop    192.168.1.3
192.168.1.2
Starry-OS <------------------------------> Windows <---100Mbps--->  Router <-----
--------> Armbian
  iperf3       :5555                      portproxy -> :22223
```

Windows:

```
netsh interface portproxy add v4tov4 listenaddress=0.0.0.0 listenport=22223
connectaddress=172.26.218.77 connectport=5555
```

测量结果

架构	x86_64	riscv64
接口层	~1000Mbps	~800Mbps
本地发送	157Mbps	104Mbps
实际发送TCP	99.4Mbps（受限于网卡）	79Mbps

- 本地发送的性能衰减源自包处理
- riscv的性能衰减来自模拟器

代码分析

以下是TCP和UDP中多路复用IO的核心代码：

```
fn block_on<F, T>(&self, mut f: F) -> AxResult<T>
where
    F: FnMut() -> AxResult<T>,
{
    if self.is_nonblocking() {
        f()
    } else {
        loop {
            SOCKET_SET.poll_interfaces();
            match f() {
                Ok(t) => return Ok(t),
                Err(AxError::WouldBlock) => axtask::yield_now(),
                Err(e) => return Err(e),
            }
        }
    }
}
```

任务会若完成则返回值，未完成则返回WouldBlock。non-blocking模式下，子任务未完成，父任务立即返回WouldBlock；blocking模式下，子任务未完成，父任务轮询到其完成为止。轮询中交替检查网络包状态和任务状态。

与真正异步的区别：

- 多次重试失败，上下文切换次数较多
- 函数返回WouldBlock的控制流必须可重复执行

Embassy-net的实现中，将socket接口改为异步；注册唤醒函数，只在完成时唤醒。若在axnet中做这种修改，需要修改上层syscall中的实现。同时，我怀疑只在内部使用异步的可行性。这是因为传输层的性能下降主要来源于包处理过程，而这一过程主要是CPU密集，异步可能并不会提升性能。

smoltcp 补丁

Bug发现

在使用iperf时，发送特定大小UDP包导致内核panic。例如如下的指令，令iperf发送长度为1488的udp包，就会产生异常：

```
iperf3 -uc 127.0.0.1 -p 5555 -b 1000M -l 1480 -R
```

```
Accepted connection from 10.0.2.2, port 59688
[  5] local 0.0.0.0 port 5555 connected to 10.0.2.2 port 36445
[  5.841111 0 axruntime::lang_items:5] panicked at crates/axnet/src/smoltcp_impl/mod.rs:307:51:
called `Result::unwrap()` on an `Err` value: InvalidParam
[  5.847252 0 axbacktrace:43] Call trace:
[  5.848636 0 axbacktrace::riscv:89] 0xFFFFFFC0802559DC
[  5.850298 0 axbacktrace::riscv:89] 0xFFFFFFC08025240C
[  5.851346 0 axbacktrace::riscv:89] 0xFFFFFFC0802194BA
```

进一步测试发现，不同长度的udp包会产生不同的行为：

- 长度<1483的udp包正常发出
- 长度1483-1494的udp包导致内核panic
- 长度>1494的udp包无法发出，表现为接收端测量的带宽为0

漏洞定位

没有分片?

由于udp协议是ip协议的简单封装，异常的现象又和包长度相关，我自然联想到ip分片机制可能是导致异常的原因。同时我注意到，axnet使用`smoltcp`包时，没有打开分片机制的选项。是不是没有分片导致的呢？当我启用了分片机制后，发送长度>1494的udp包的行为也正常了，但是长度1483-1494的udp包仍然会导致panic。

Cargo.toml M

Cargo.toml

```
[dependencies.smoltcp]
git = "https://github.com/rcore-os/smoltcp.git"
rev = "8bf9a9a"
default-features = false
features = [
  "alloc", "log",   # no std
  "medium-ethernet",
  "medium-ip",
  "proto-ipv4",
  "proto-ipv6",
  "socket-raw", "socket-icmp", "socket-udp", "socket-tcp", "socket-dns", "proto-igmp",
  "fragmentation-buffer-size-65536", "proto-ipv4-fragmentation",
  "reassembly-buffer-size-65536", "reassembly-buffer-count-32",
  "assembler-max-segment-count-32",
]

```

此时一个现象误导了我。我观察到，当包长在特定长度时（典型值是2900 octets），会产生大量丢包，我一度以为这和panic有关，因此通过`tcpdump`抓包分析。但是抓包结果表明，发送过程很正常，由此可见丢包现象和bug无关。（从事后来看，这种丢包可能是发包速度过快造成的）

缓冲区过小?

首先从发生panic处开始寻找。Panic的直接原因是，接口层缓冲区无法容纳过长的数据，返回了`Err`，而上层直接又使用了`unwrap()`。此时怀疑接口层的缓冲区是否过小。但经过一番计算，缓冲区大小为1526，MTU=1500，+14以太网头+10元信息头，应当是足够的。此时我仍然怀疑是axnet的实现问题，因为没有限制udp包的大小，导致缓冲区大小不够。

没有分片！

在重新学习了ip报文的分片机制后，我纠正了上面的看法。因为udp作为传输层协议不应该限制包大小，而是交给网络层处理。真正的问题应该在于分片机制异常。在详细计算了各个层的包大小后，终于发现了端倪：当发送1482的udp包时，此时ipv4包长为1502，超出链路层MTU，应当分片，但是没有被分片；而又由于缓冲区有2字节的冗余，链路层也没有做校验，这一现象没有暴露出来，成功发出了长度超出MTU设定的ipv4报文。

最终，我在smoltcp中找到了漏洞的来源：

src/iface/interface/mod.rs

```
@@ -1201,7 +1201,7 @@ impl InterfaceInner {
1201 1201                #[cfg(feature = "proto-ipv4")]
1202 1202                IpRepr::Ipv4(repr) => {
1203 1203                    // If we have an IPv4 packet, then we need to check if we need to fragment it.
1204      -              if total_ip_len > self.caps.max_transmission_unit {
     1204 +              if total_ip_len > self.caps.ip_mtu() {
1205 1205                    #[cfg(feature = "proto-ipv4-fragmentation")]
1206 1206                    {
1207 1207                        net_debug!("start fragmentation");
```

这段代码的逻辑是：判断原始ipv4包是否超过MTU，如果没有直接发送；如果超过进入分片过程。但是它混淆了两种MTU。常见的MTU是指接口层最大负载的长度，对于以太网为1500 octets；而这里的`caps.max_transmission_unit`则是接口层最大传输单元，也就是说包含以太网头的14 octets。

那么解决这个漏洞也很简单，将`caps.max_transmission_unit`换成`caps.ip_mtu()`即可。

补丁

补丁的Commit，以及Pull request：

[Fix the error of specific length IP packets not being fragmented by Jc0x7D3 · Pull Request #1008 · smoltcp-rs/smoltcp]

这个commit首先改掉了这个bug，另外加了一个复现这个bug的测例。

测例在ip层工作，发送了各个长度的udp报文（特别是触发bug的长度），并在接口层判断长度是否正确。

```
last year       fix(791): wrong payload length o...
2 years ago     Split interface into multiple files
last year       fix(791): wrong payload length o...
2 years ago     Split interface into multiple files
last week       fix bug
2 years ago     Split interface into multiple files
```

```
1200            match &mut ip_repr {
1201                #[cfg(feature = "proto-ipv4")]
1202                IpRepr::Ipv4(repr) => {
1203                    // If we have an IPv4 packet, then we need to check if we need to fragme
1204                    if total_ip_len > self.caps.ip_mtu() {
1205                        #[cfg(feature = "proto-ipv4-fragmentation")]
1206                        {
1207                            net_debug!("start fragmentation");
1208
1209                            // Calculate how much we will send now (including the Ethernet h
1210                            let tx_len = self.caps.max_transmission_unit;
1211
1212                            let ip_header_len = repr.buffer_len();
1213                            let first_frag_ip_len = self.caps.ip_mtu();
1214
```

反思

为什么这么简单的问题这么久无人发现？这个bug可以说是非常明显的，因为只需要静态地看语义就可以发现问题。但是事实就是，从2年前这句代码被写下开始，就没有人注意到这个漏洞。一个明显的、低级的错误，在一个收获近4k stars，有数百贡献者的开源项目中存在了近两年，是相当令人意外的。我认为可能有如下原因和问题：

- **单元测试覆盖率不足**

  如果在bug语句前打上panic!()就会发现，上百个测试中，仅有两个测试点运行到了bug语句。而且这两个测试点都是进出组播组相关的，报文长度很短，根本不可能触发bug或分片。也就是说，这条语句没有被单元测试覆盖到，是长久没有发现bug的主要原因。

- **完全没有集成测试**

  这个项目测试的另一个问题是集成测试。其对分片机制仅局限于fragmenter本身，而缺乏从包创建、分片、发送、组装、解析这一完整流程的测试。因此，当问题出现在创建之后、分片之前，判断是否需要分片的分支语句时，就难以被发现。

- **分片特性应用不广，不受重视**

  分片特性是一个很尴尬的特性，有性能问题、安全问题、路由器支持性问题，ipv6甚至只在扩展包头中支持分片，因此其不受欢迎和重视是一个客观的事实。

- **项目作者管理比较随意?**

  虽然我反复推敲了新增的测试代码，但是第一次提交的代码通常是需要修改才会被合入的。但是我提交了pr后，第二天项目作者就直接merge了。

总结

尽管这半个学期并没有完成原定的对axnet进行异步优化的目标，但是完成了初步的可行性分析，并且还阴差阳错地发现和解决了开源项目中的一个bug。在解决bug的过程中，我深入探索了协议栈的实现，加深了对TCP/IP协议、协议实现以及操作系统的理解。这次大实验的经历对我来说的确是印象深刻，也让我收获颇丰。