工程实践与科技创新Ⅲ-D 大作业五

陈文迪 519021910071

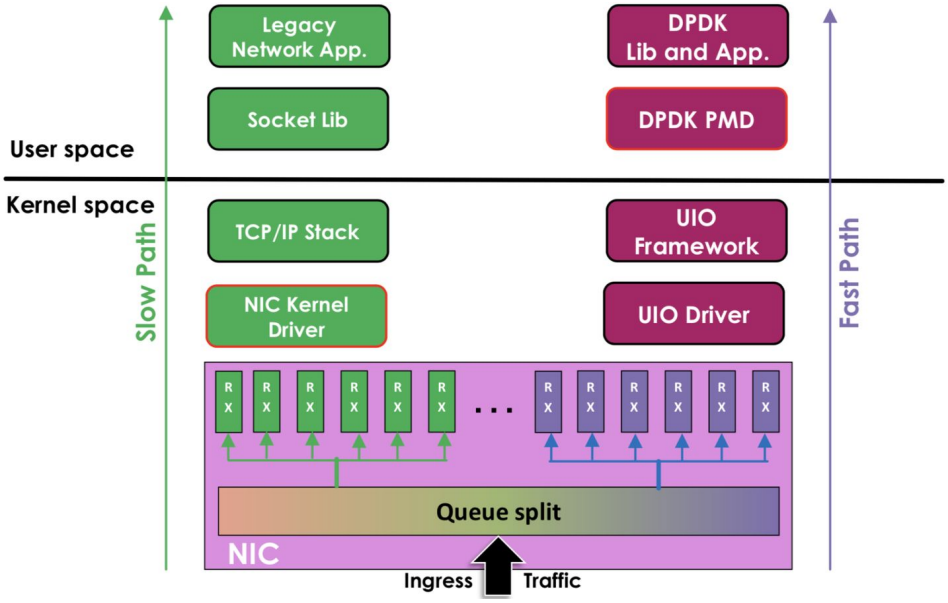
实验目标

- DPDK 性能测试
  - 在 KVM 上创建两个虚拟机
  - 分别在两台虚拟机上编译和安装DPDK库
  - 在 VM2 上编译并运行 DPDK 样例程序 l2fwd，接着在 VM1 上编译并运行 pktgen-dpdk。pktgen-dpdk 可以记录 VM1 所发送的包的大小和从 VM2 所接收的包的大小。l2fwd 会将它从 VM1 接收的包重新发回。
  - 评估 DPDK 在 L2 forwarding上的性能。

相关知识

DPDK

DPDK (Data Plane Development Kit) 是数据平面开发工具包，其中包含了大量可以加速CPU数据包处理的库。DPDK 相比于其他的包捕获工具来说，其通过 UIO+mmap、HugePages 等技术优化了传统数据包捕获受 Linus Kernel 限制的问题，极大地提高了数据处理性能，提高了数据平面应用程序的工作效率。 同时，DPDK 的环境抽象层向应用与函数库隐藏了底层环境的细节，因此其可以扩展到任何处理器上使用。



UIO 与 VFIO

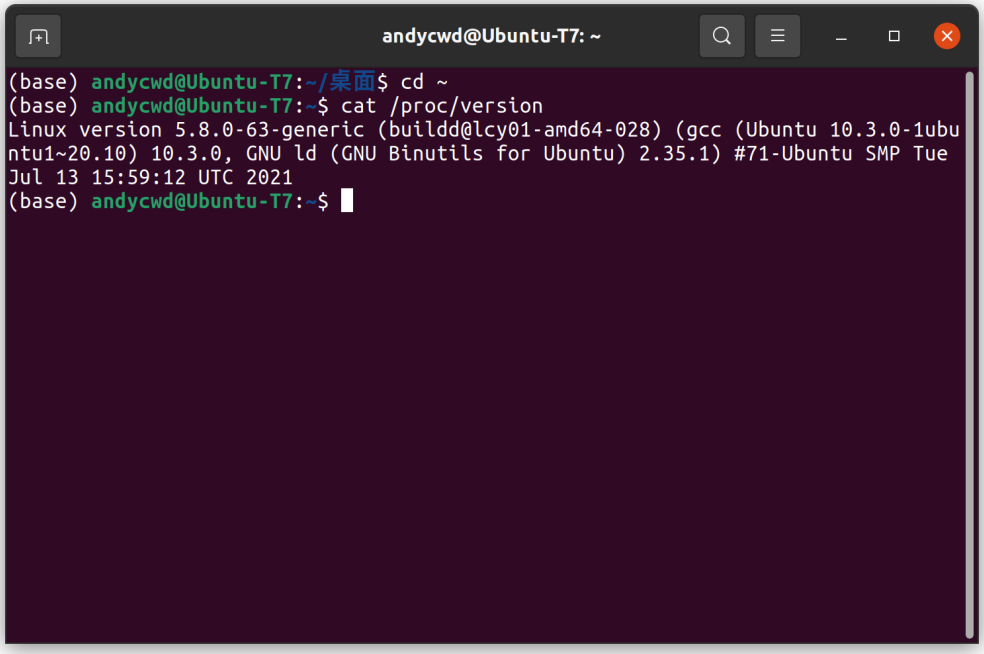
网卡想要与DPDK应用程序正常工作，需要加载相应的内核驱动程序，主要有UIO和VFIO两种。 UIO (Userspace I/O) 是运行在用户空间的 I/O 技术。它将驱动的很少一部分运行在内核空间，而在用户空间实现驱动的绝大部分功能。而 VFIO 则通过 IOMMU保护机制实现了更加鲁棒和安全的驱动。

l2fwd 与 pktgen-dpdk

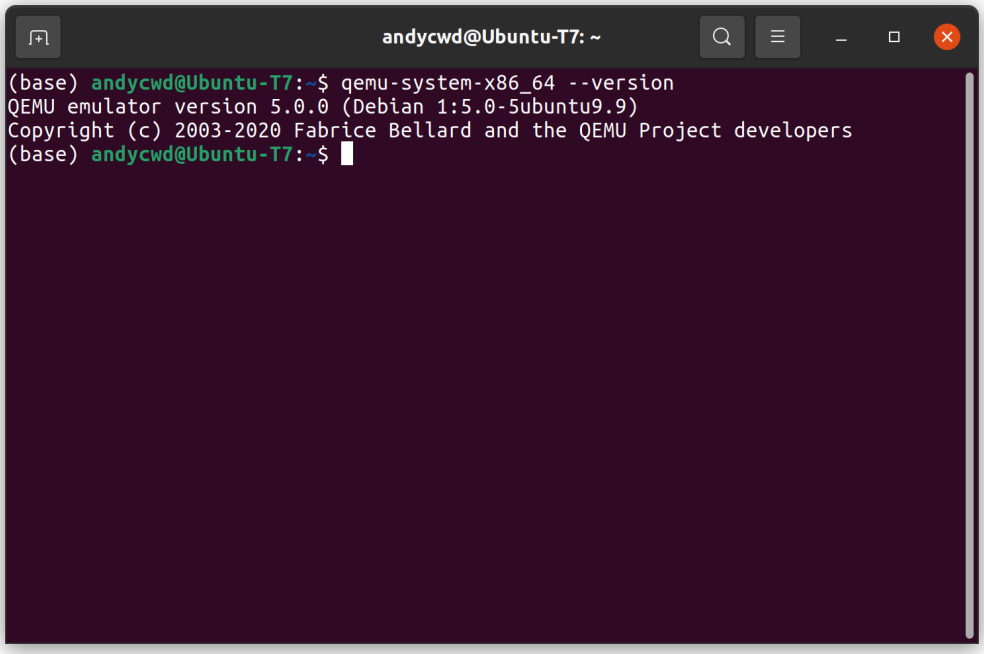
l2fwd是 DPDK 的一个示例程序，为 RX_PORT 上接收的每个数据包执行二层转发，转发的目标端口是启用端口掩码的相邻端口。pktgen-dpdk 则是一款用于对 DPDK 进行高速数据包测试的工具。

实验环境

本实验的宿主系统为 Ubuntu 20.10，直接运行在宿主机 Lenovo Yoga 14s 2021上。



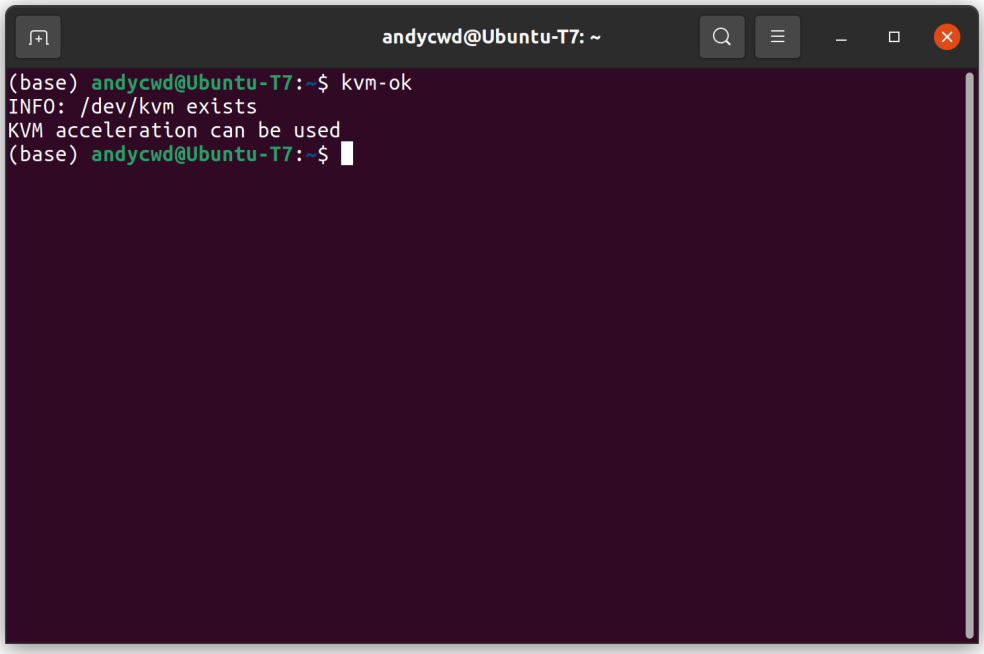
本机器已经安装了 QEMU 5.0.0。



实验过程

准备工作

- 执行 `kvm-ok` 确保硬件支持虚拟化扩展。从结果中看到，我们的主机支持虚拟化扩展。



创建虚拟机

在大作业二和三中，我们了解到虚拟机的创建主要有三种方式：直接使用QEMU安装虚拟机，使用 virt-manager 利用 GUI 安装或者使用 `virt-install` 利用命令行工具安装。

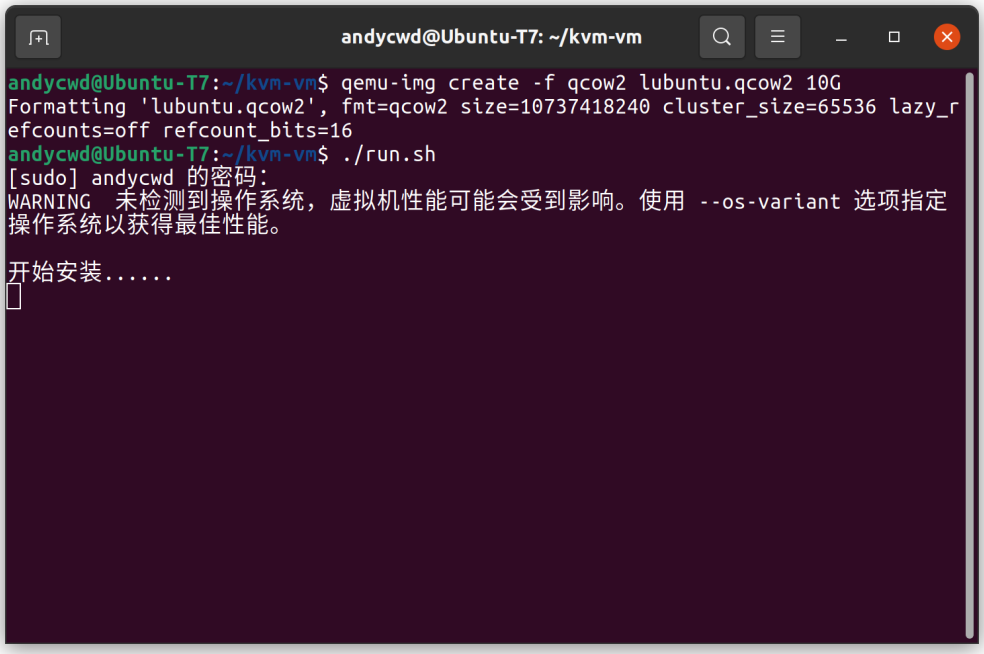
- 使用 QEMU 安装虚拟机分为两步。先执行 `qemu-img create -f qcow2 lubuntu.qcow2 10G` 来创建一个大小为10G，格式为qcow2的磁盘镜像。接着执行以下命令启动虚拟机并完成安装。

  ```
  sudo qemu-system-x86_64 -cdrom ./lubuntu-20.04.3-desktop-amd64.iso \
      -cpu host -enable-kvm -m 2048 -smp 2 \
      -drive file=./lubuntu.qcow2,format=qcow2
  ```

- 使用virt-manager 安装虚拟机的方法比较简单，只需要根据图形化界面的提示选择即可。

- 使用`virt-install`安装的过程与直接使用 QEMU 时类似，我们先执行`qemu-img create -f qcow2 lubuntu.qcow2 10G`来创建一个大小为10G，格式为qcow2的磁盘镜像。执行下列命令来安装指定配置的虚拟机，其中`--connect qemu:///system`表示我们指定 QEMU 作为 hypervisor。按照提示完成安装即可。

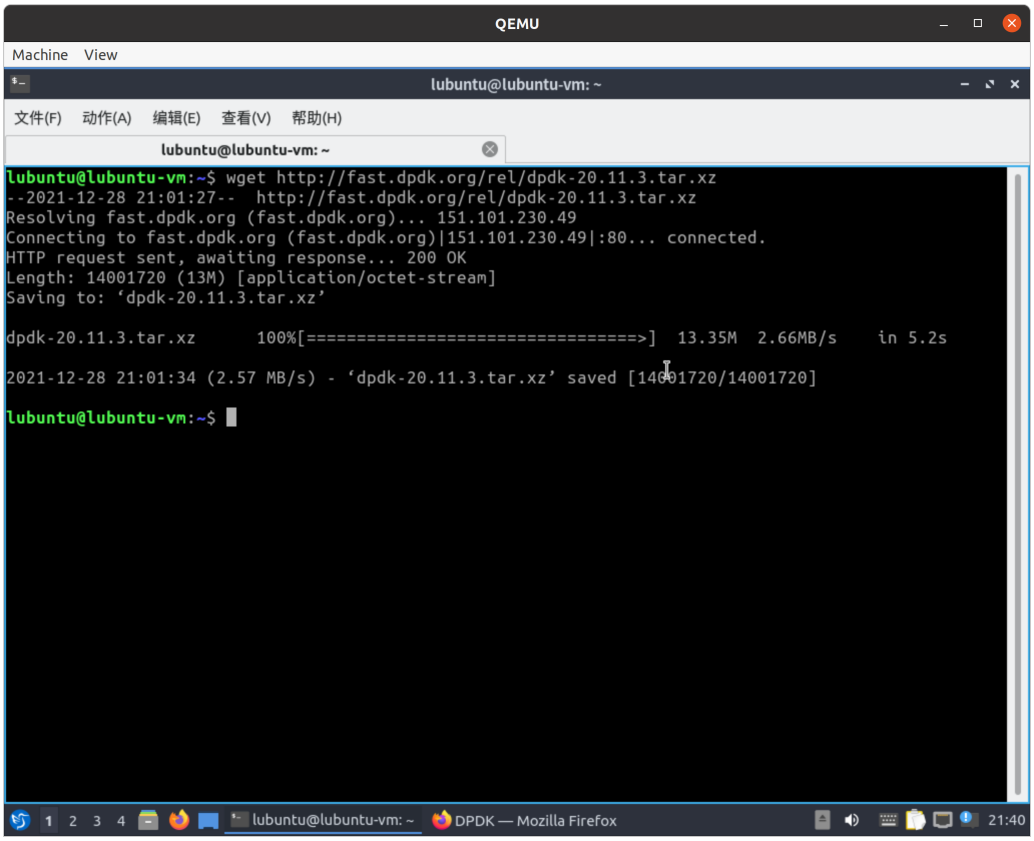
```
sudo virt-install --connect qemu:///system --name lubuntu \
    --vcpus 2 --ram 1024 --virt-type=kvm \
    --cdrom=./lubuntu-20.04.3-desktop-amd64.iso \
    --disk path=./lubuntu.qcow2 \
    --vnc
```



编译安装 DPDK

由于两台虚拟机均须要安装DPDK，因此一个比较好的方案是现在一台虚拟机上编译安装完成，再直接复制已经安装好DPDK的虚拟机镜像文件，作为第二台虚拟机。

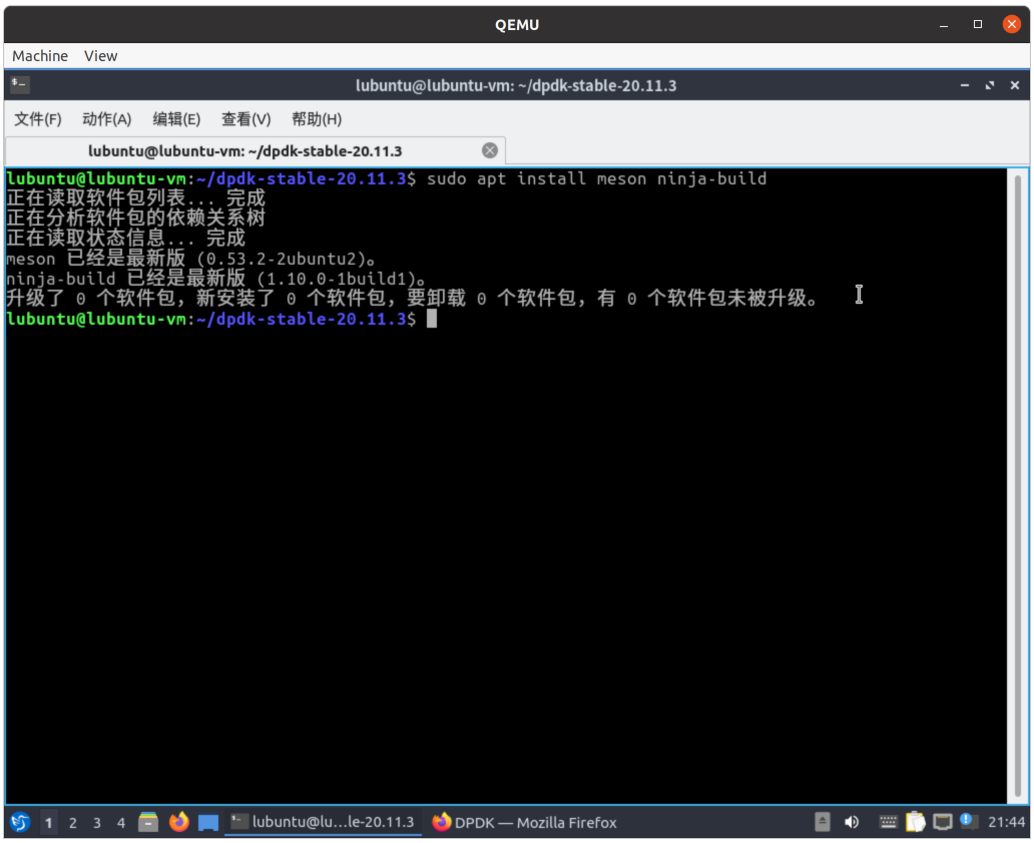
- 前往 DPDK 官网下载源码，这里我们选择 DPDK 20.11.3（LTS）版本。



- 执行以下命令解压并进入源码目录。

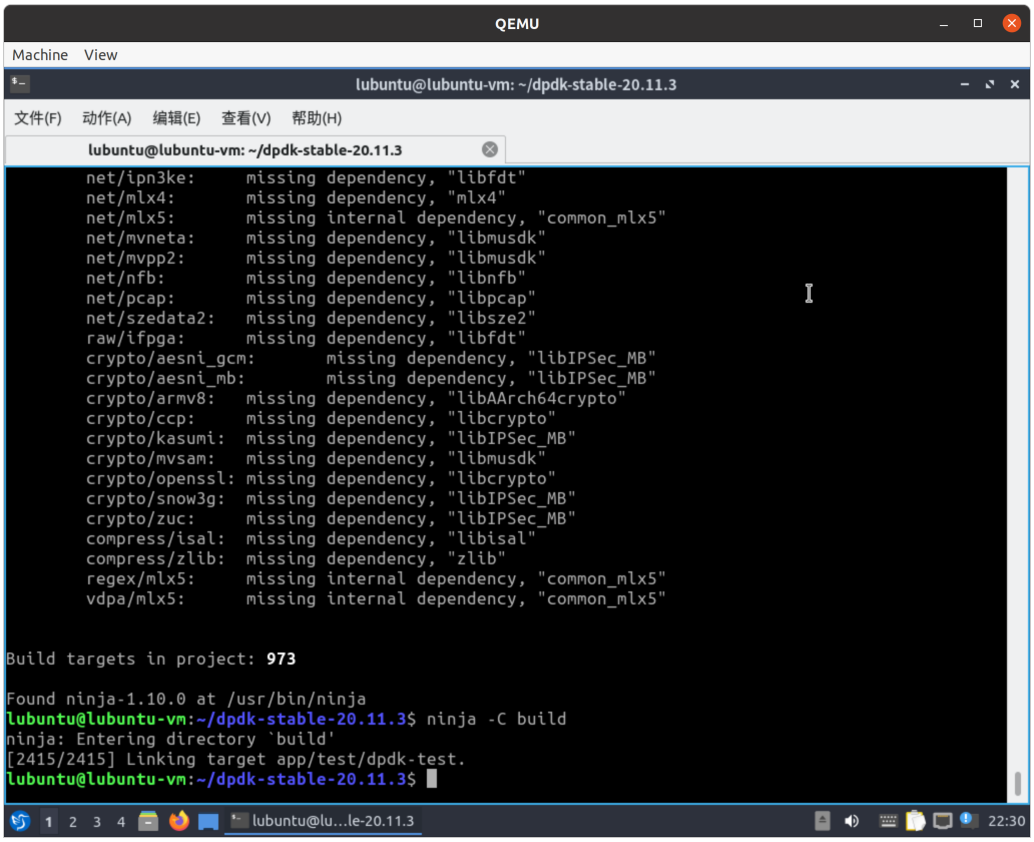
```
tar xf dpdk.tar.gz
cd dpdk
```

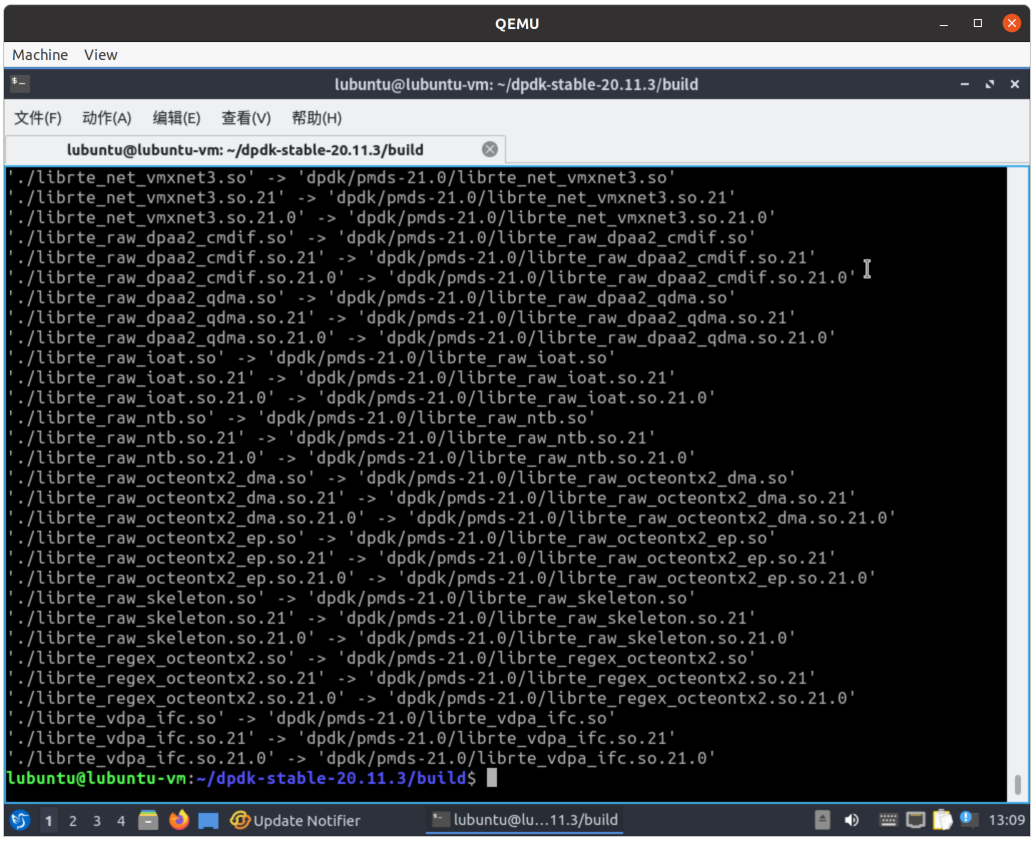
- 执行 `apt install meson ninja-build` 安装对应的构建工具。接着，执行 `apt install clang` 安装构建工具所需的编译器工具。



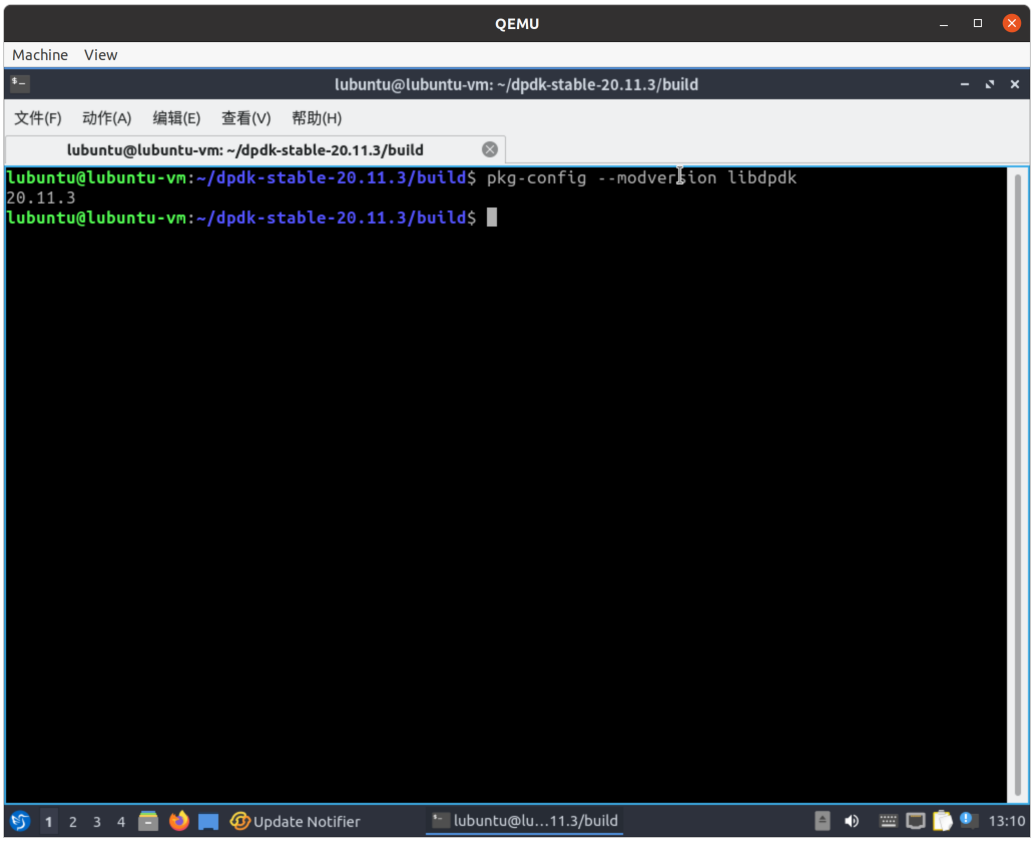
- 执行下列命令安装库、驱动和测试程序。注意：这里我们附加了 `-Dexamples=all` 来安装全部的示例程序。

```
meson -Dexamples=all build
ninja -C build
cd build
ninja install
```



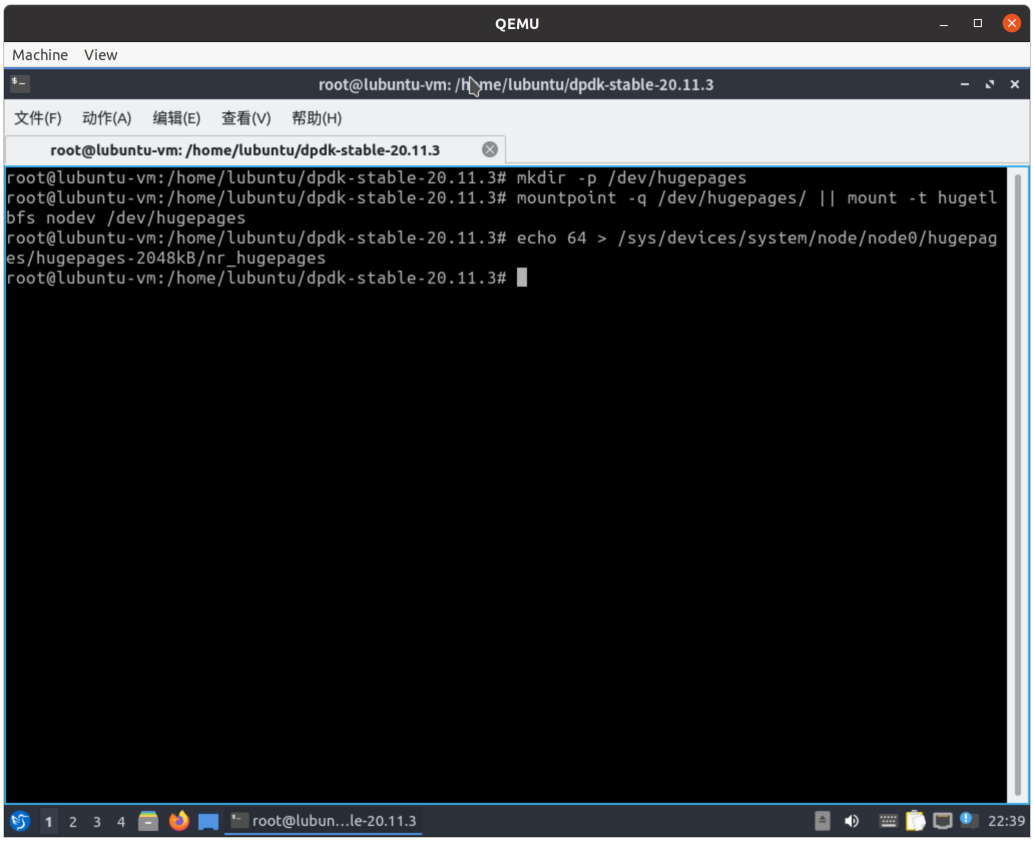


- 执行 `pkg-config --modversion libdpdk` 检查安装。可以看到，指定版本的 DPDK 安装成功。

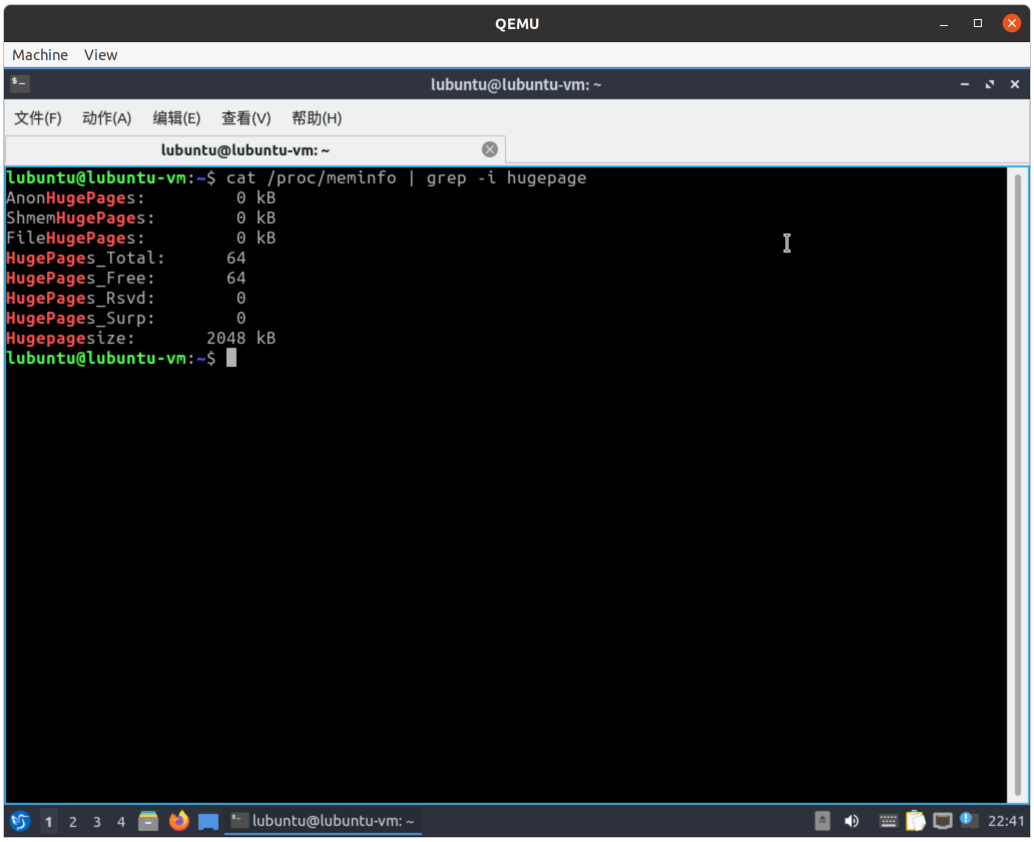


- 执行下列指令保留制定大小的巨页内存。注意：此处我们需要切换到root用户，否则会遇到权限问题。

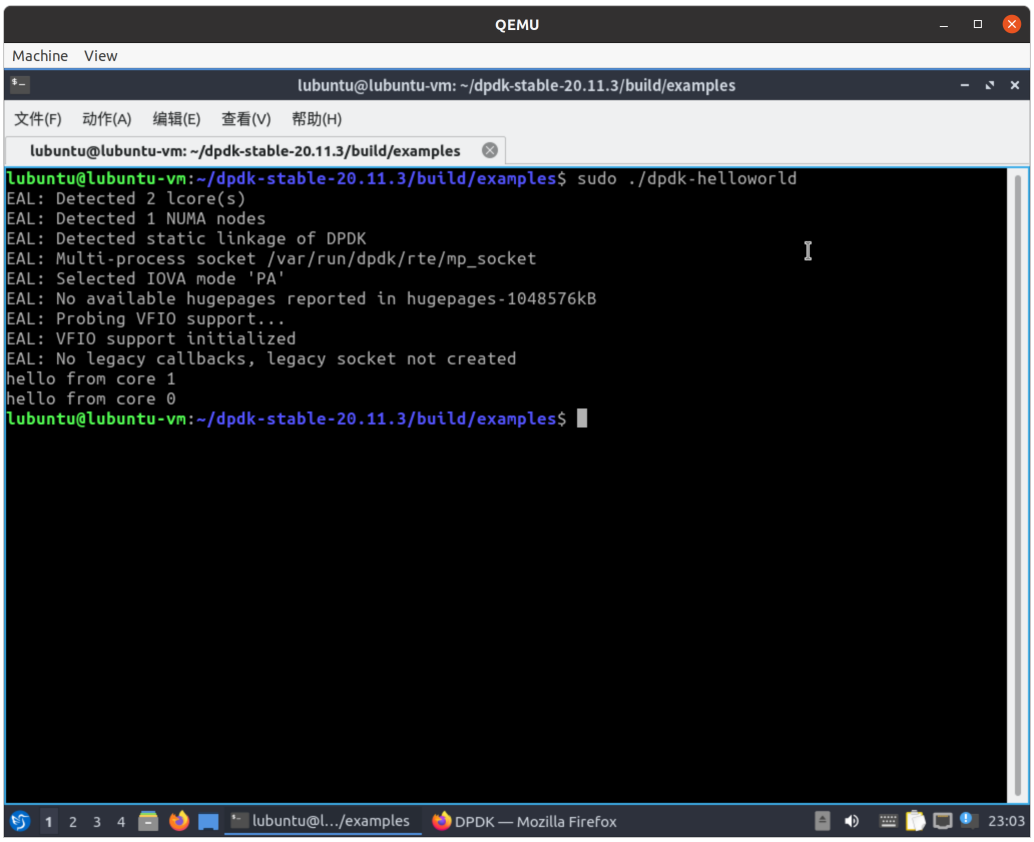
```
mkdir -p /dev/hugepages
mountpoint -q /dev/hugepages || mount -t hugetlbfs nodev /dev/hugepages
echo 64 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
```



从结果中可以看到，我们开启了指定大小的巨页。



- 运行 dpdk-helloworld 示例程序来验证安装。



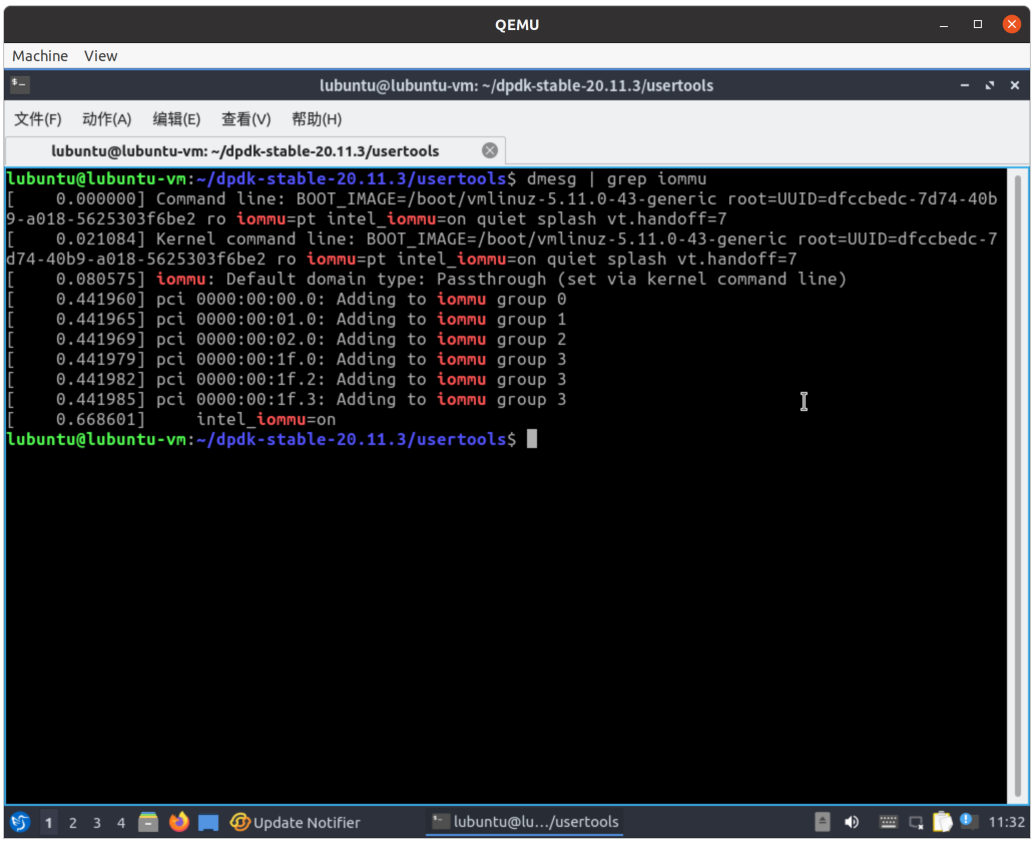
DPDK 驱动加载

这里我们使用更为强大和安全的VFIO用作演示。

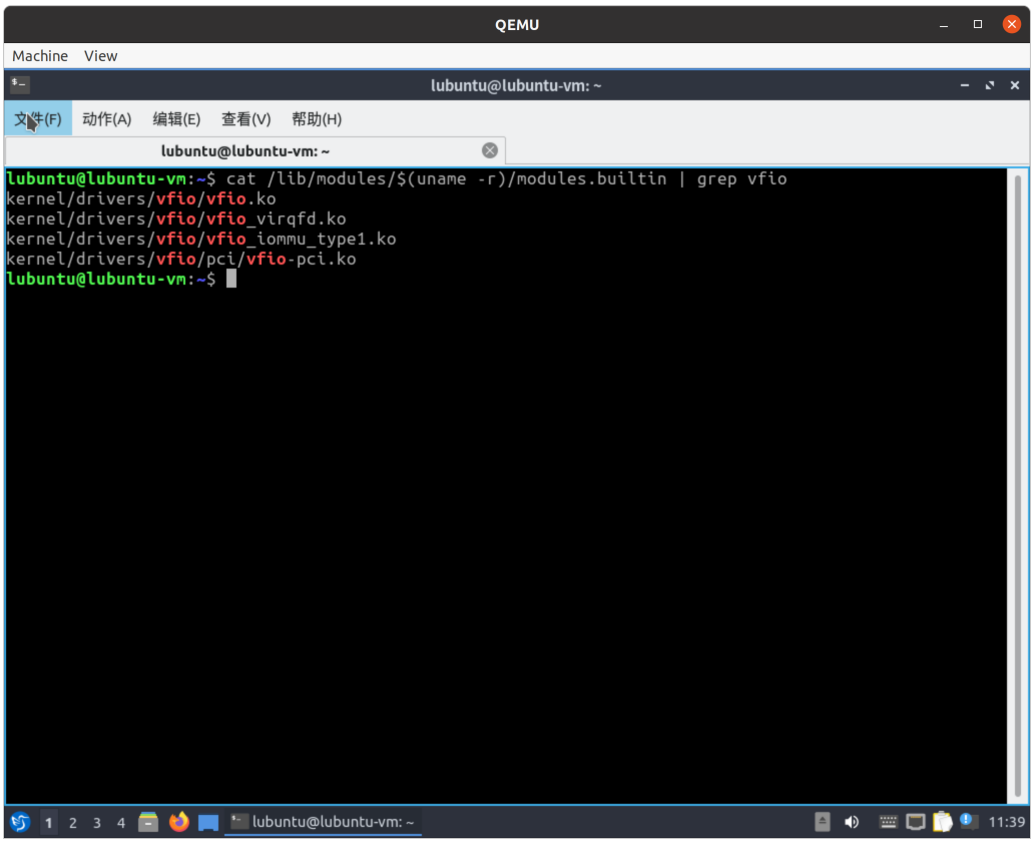
- 要使用VFIO驱动绑定到NIC，我们需要使能iommu。在 QEMU 启动参数中增加 `-machine q35 -device intel-iommu,intremap=off` 字段，来在虚拟机中启用 vIOMMU 设备。通过修改grub文件中的以下字段，即可使能iommu。

```
...
GRUB_CMDLINE_LINUX="iommu=pt intel_iommu=on"
...
```

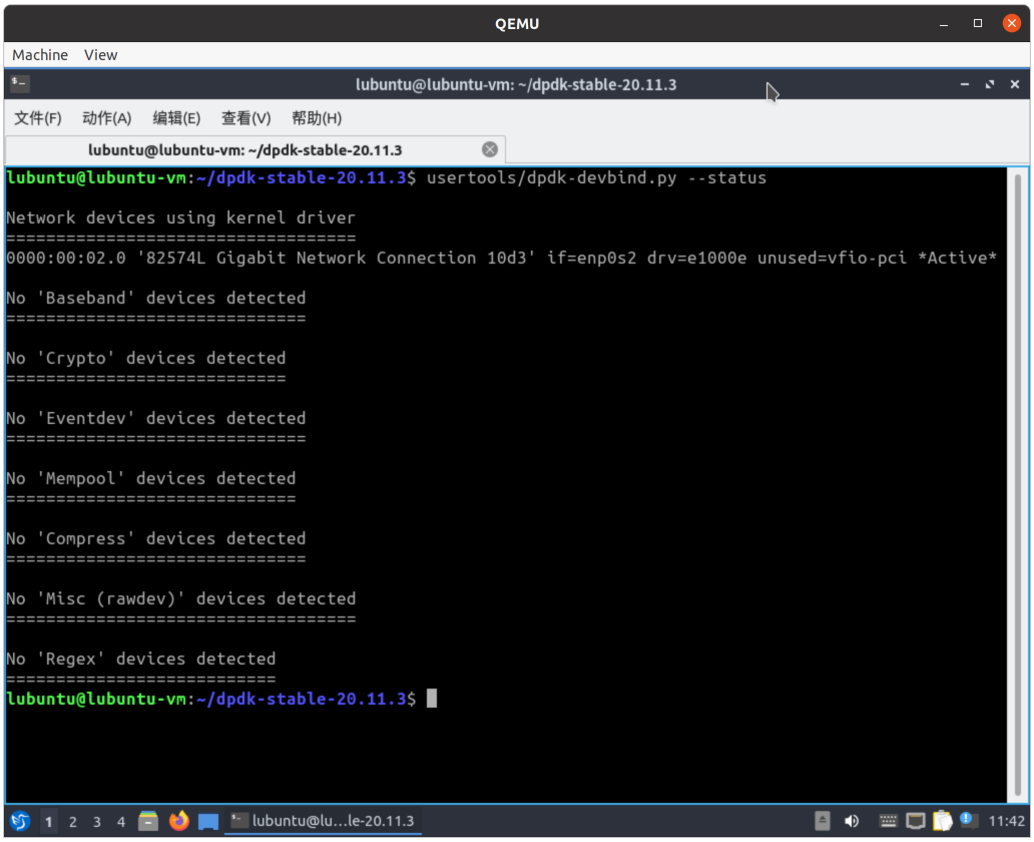
接着，执行 `sudo update-grub` 更新grub文件。重启后，可以通过 `dmesg | grep iommu` 可以看到 iommu 使能成功。



- 由于Ubuntu 20.10的内核已经构建了VFIO驱动，所以我们不需要手动开启。使用 `cat /lib/modules/$(uname -r)/modules.builtin | grep vfio` 可以确定内核模块已构建。



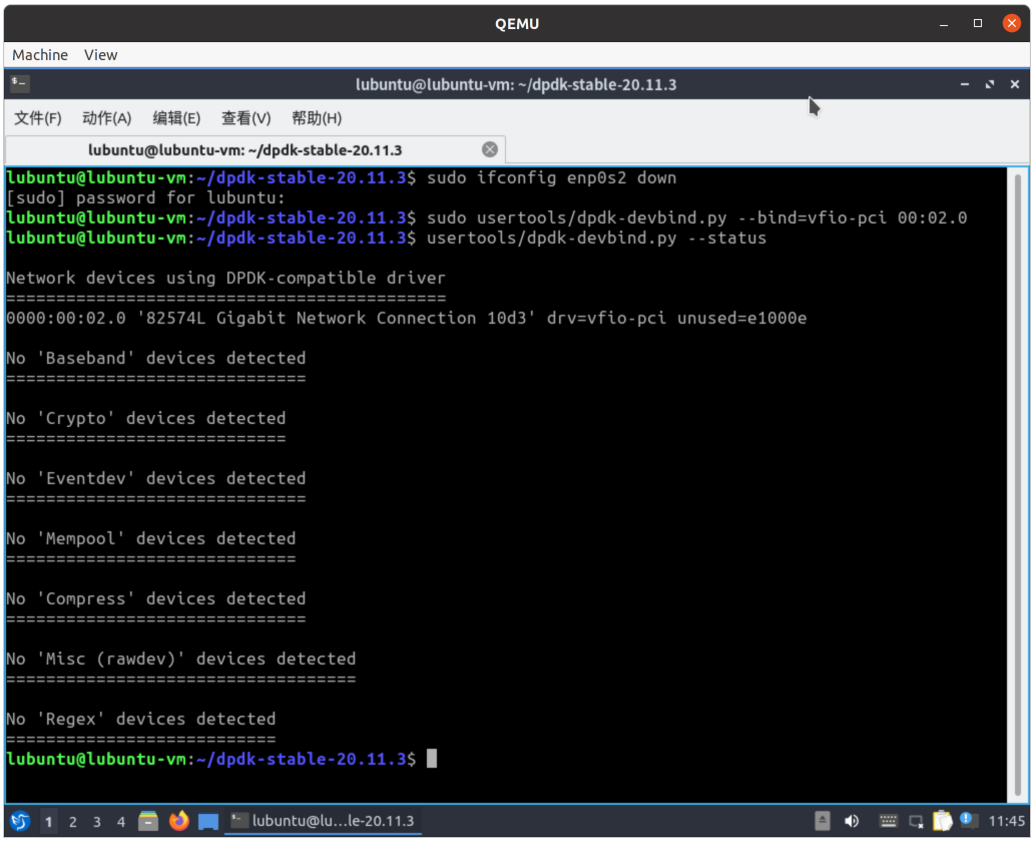
- 在绑定网卡之前，我们先使用 `usertools/dpdk-devbind.py --status` 查看绑定状态。可以看到，目前网络设备被绑定到内核驱动。



接着，执行下列命令来关闭网卡并绑定VFIO驱动和网卡。

```
sudo ifconfig enp0s2 down
usertools/dpdk-devbind.py --bind=vfio-pci 00:02.0
```

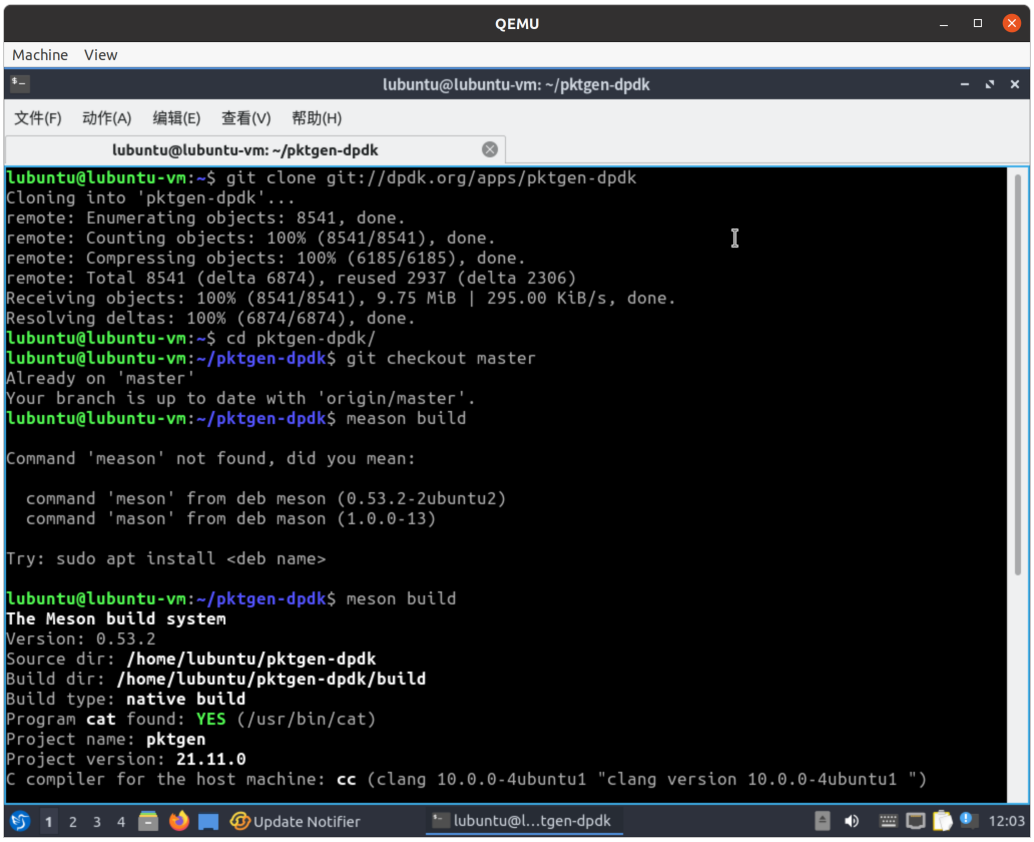
使用 `usertools/dpdk-devbind.py --status` 查看绑定状态，可以看到VFIO驱动和网卡成功绑定。



编译运行pktgen-dpdk

- 执行下列命令下载源码仓库并使用构建工具进行构建。

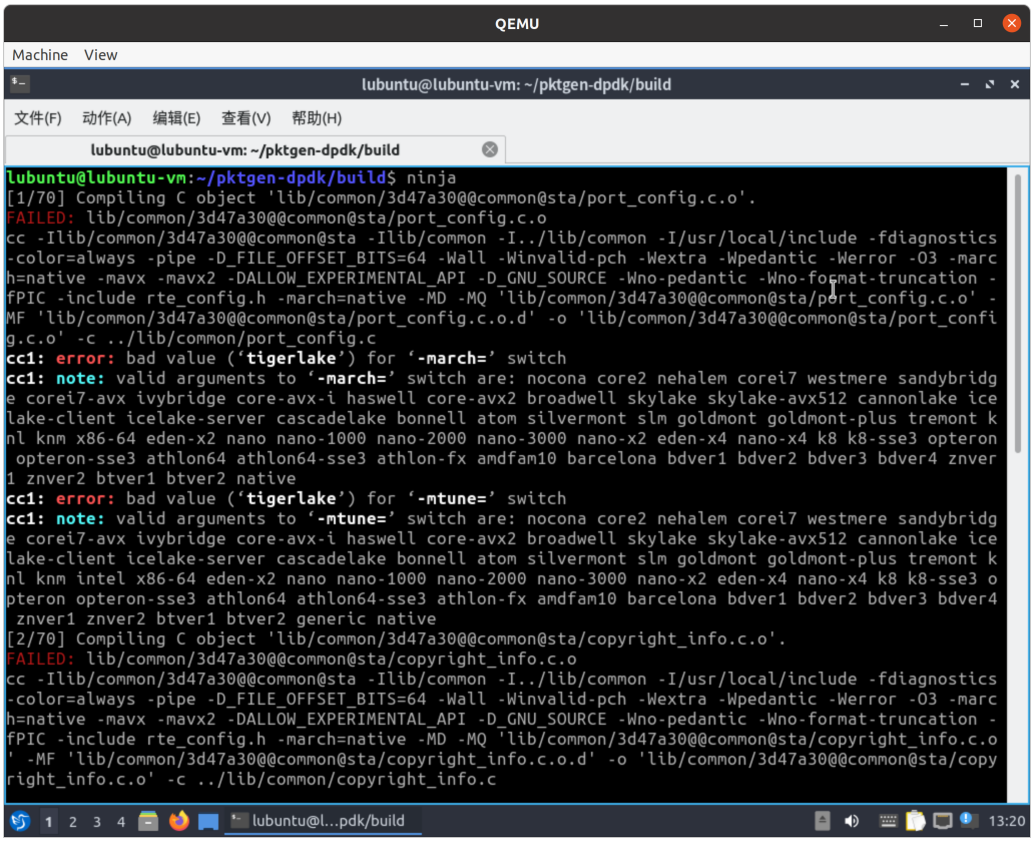
```
git clone git://dpdk.org/apps/pktgen-dpdk
cd pktgen-dpdk
git checkout master
meson build
cd build
ninja
# 或直接执行下列命令安装
./tools/pktgen-build.sh build
```



- 在过程中，我们可能遇到 C 语言库缺失的问题。执行下列命令安装即可。

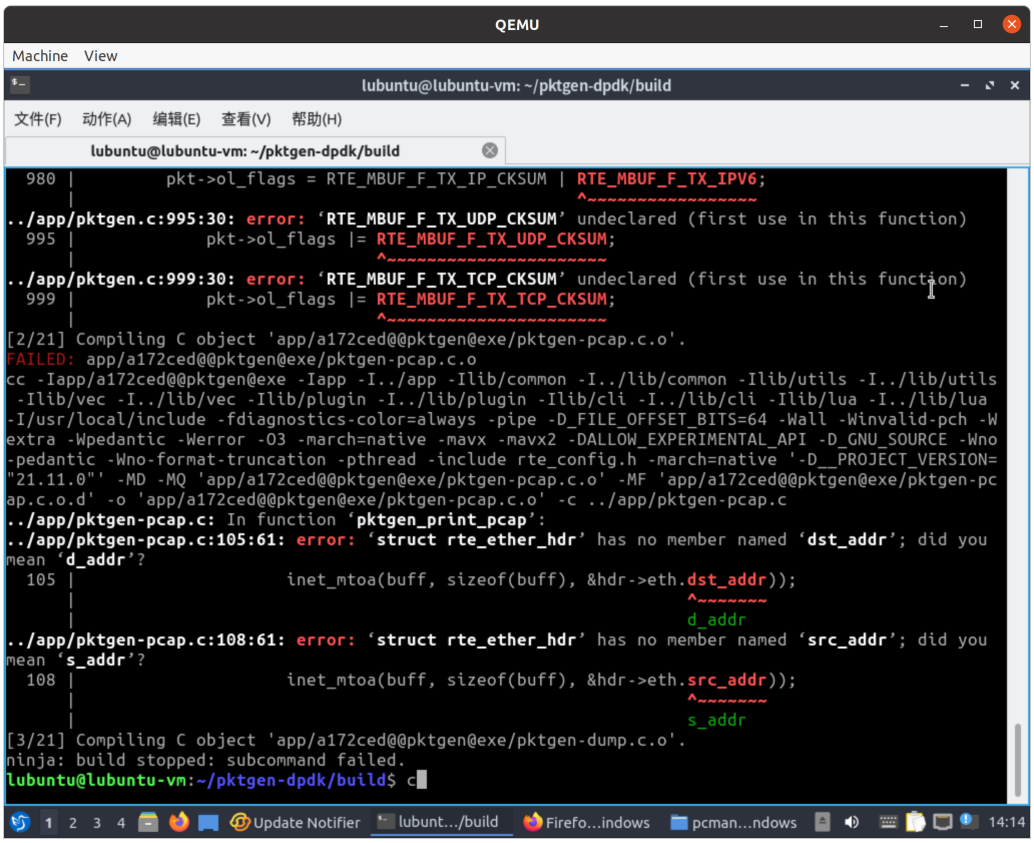
```
sudo apt install libnuma-dev libpcap-dev
```

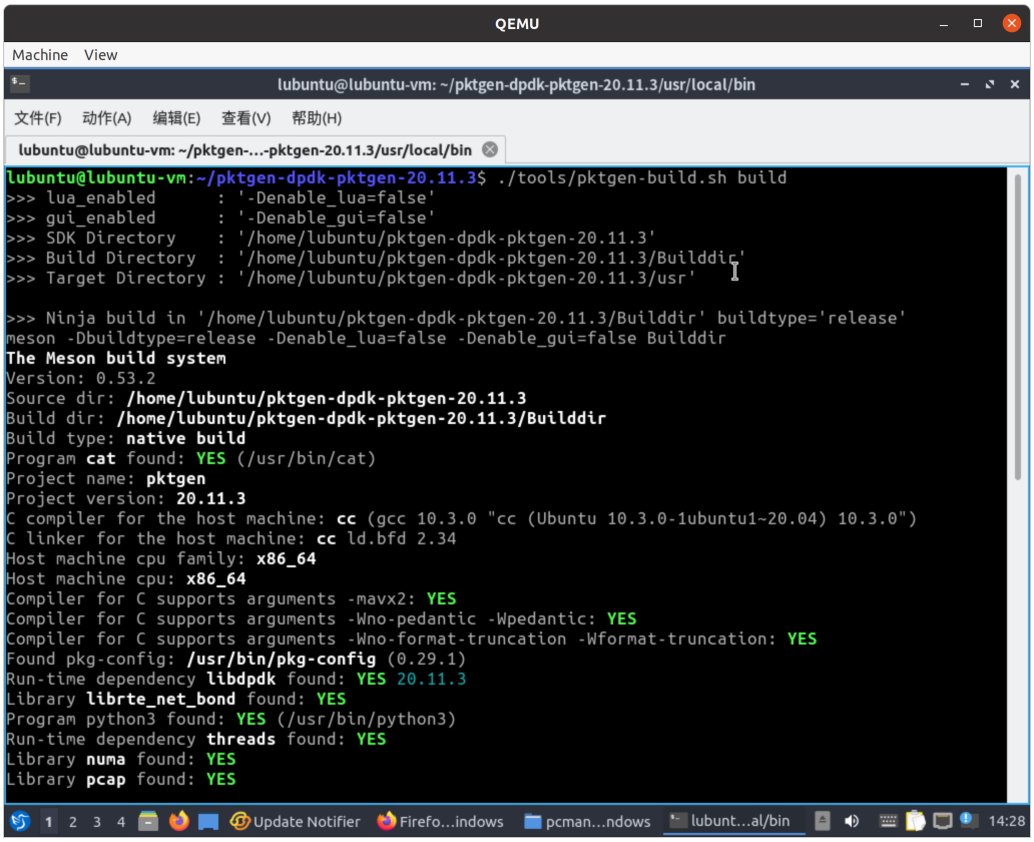
- 在执行 `ninja` 时出现如下问题，查阅相关资料后发现，是 gcc 版本不够导致的，执行下列命令升级 gcc 版本后解决。



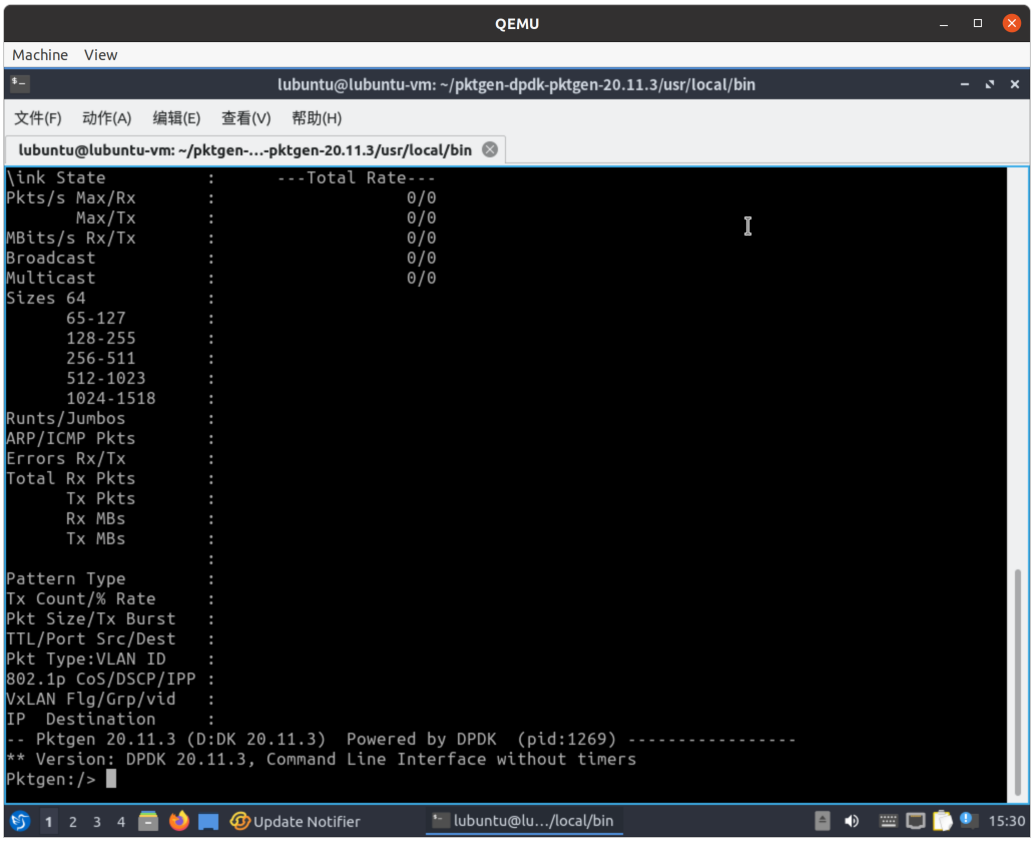
```
sudo apt install gcc-10 g++-10
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-10 100 --slave /usr/bin/g++ g++ /usr/bin/g++-10 --slave /usr/bin/gcov gcov /usr/bin/gcov-10
```

- 继续安装，遇到如下问题。经检查后发现时pktgen-dpdk版本过高导致的，使用 pktgen-dpdk-20.11.3后问题解决。





- 绑定网卡后可以顺利执行 pktgen 程序。



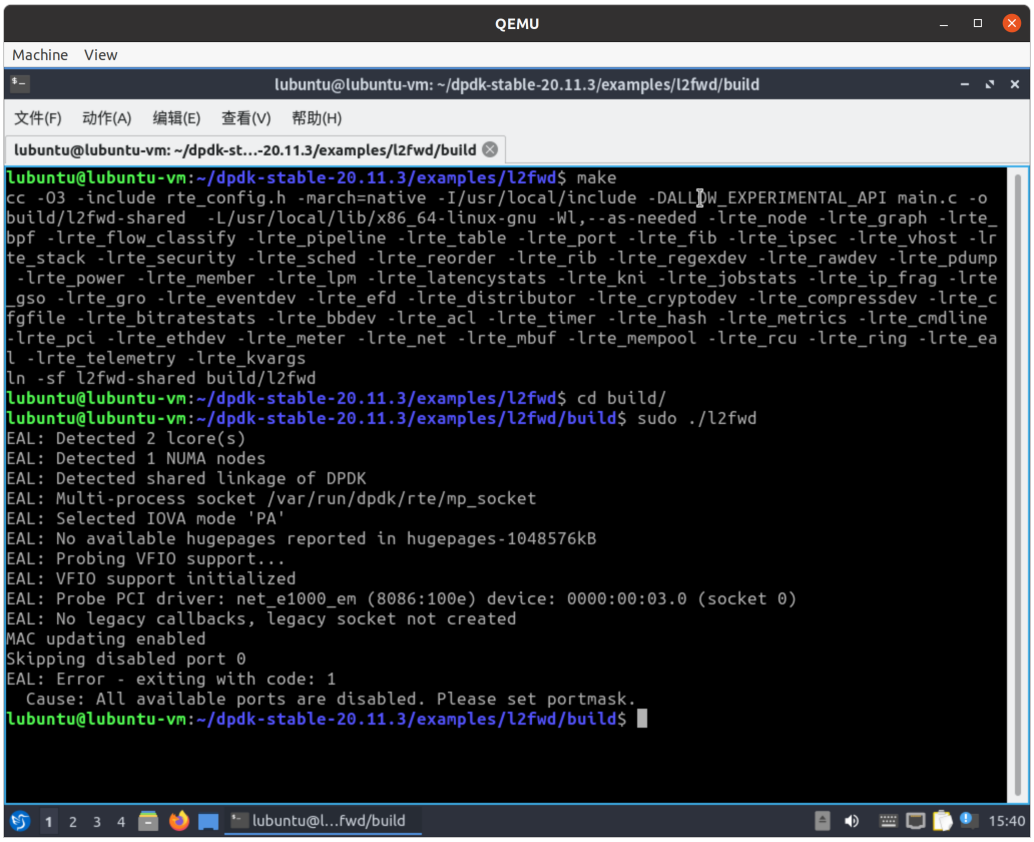
- 在后续实验中发现，我们还需要修改 `pktgen-build.sh`，开始 Lua 脚本功能。并执行以下命令安装 Lua 相关依赖。

```
sudo apt install lua5.3 liblua5.3-dev
```

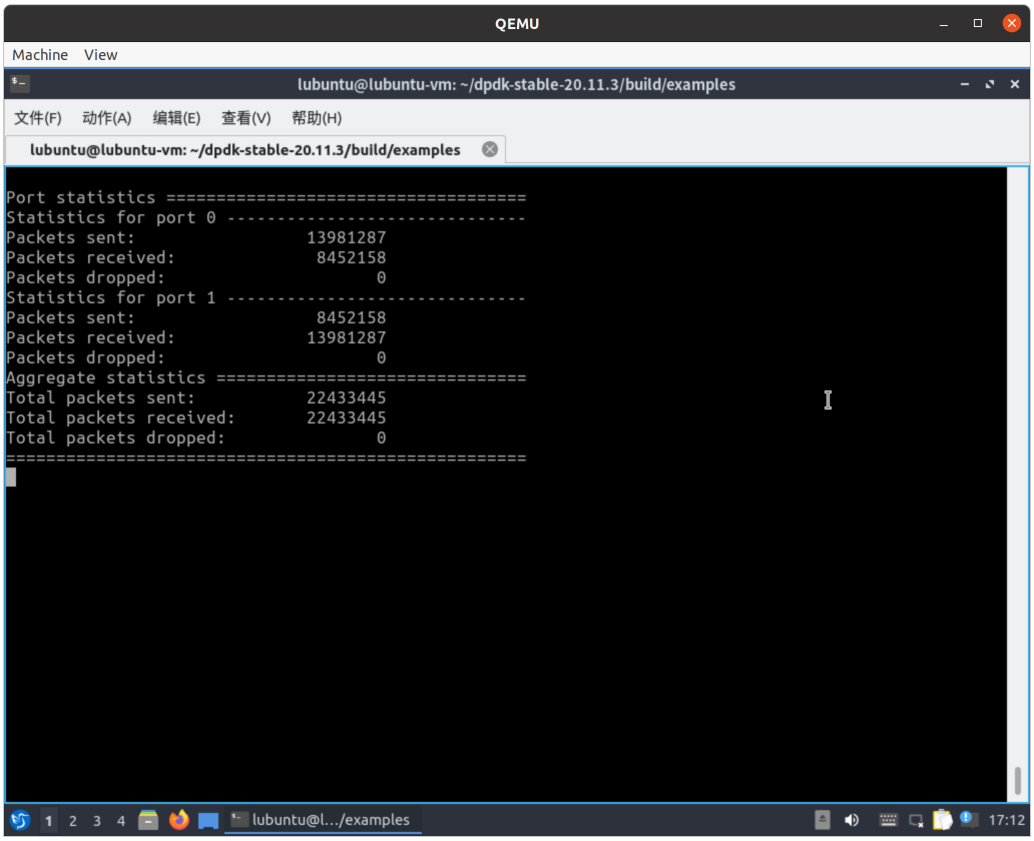
编译运行l2fwd

事实上我们在编译 DPDK 时，已经增加了 `-Dexamples=all` 参数，因此包括 l2fwd 在内的所有示例程序都已经编译完毕。当然，我们可以执行以下命令来手动编译并运行。从结果可以看出，l2fwd 可以正常运行。

```
make
cd build
sudo ./l2fwd
```

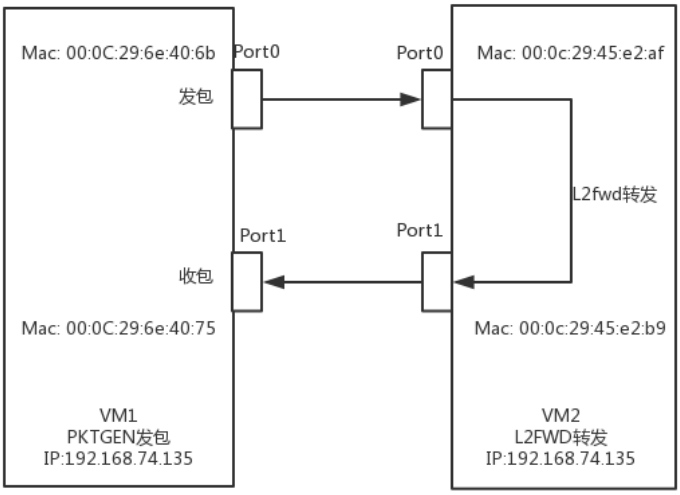


添加两块网卡并绑定驱动后，执行 `sudo ./l2fwd -n 1 - c 0x3 -- -q 4 -p 0x3`。可以看到 l2fwd 正常运行。



连接 VM1 与 VM2 并配置实验环境

在本次大作业中，我们会使用两种网络拓扑结构。一种是在pktgen-dpdk端（也即VM1端）使用一块网卡用于收发，在l2fwd端（也即VM2端）使用两块网卡，总共三块网卡通过一个网桥相连。第二种拓扑结构则在VM1端也使用两块网卡分别用于收发，并通过两个网桥与VM2相连。第二种网络拓扑结构如图所示：

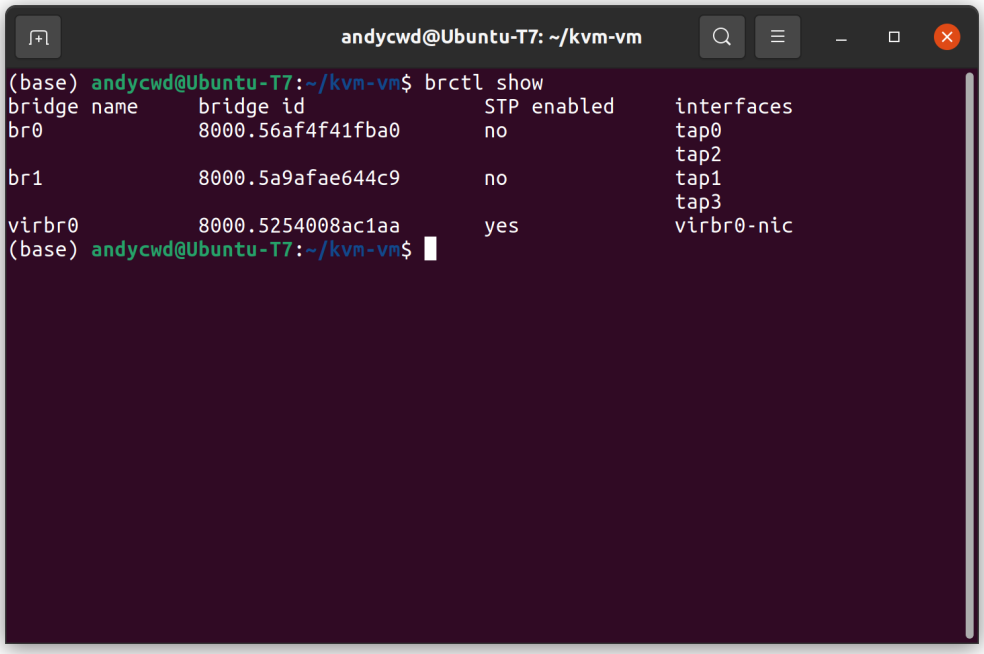


这里，我们以第二种拓扑结构为例子展示如何搭建实验环境。

- 按照大作业二中的方法添加两个网桥，并创建四个tap接口共虚拟机网卡连接。

```
sudo brctl addbr br0
sudo brctl addbr br1
sudo tunctl -t tap0 -u 'andycwd'
sudo tunctl -t tap1 -u 'andycwd'
sudo tunctl -t tap2 -u 'andycwd'
sudo tunctl -t tap3 -u 'andycwd'
sudo brctl addif br0 tap0
sudo brctl addif br1 tap1
sudo brctl addif br0 tap2
sudo brctl addif br1 tap3
sudo ip link set dev br0 up
sudo ip link set dev br1 up
sudo ip link set dev tap0 up
sudo ip link set dev tap1 up
sudo ip link set dev tap2 up
sudo ip link set dev tap3 up
```

  执行 `brctl show` 可以看到网桥与tap接口成功创建。



- 使用下列代码启动虚拟机，以在每个虚拟机上创建两个网卡并连接到对应的tap接口。

```
sudo qemu-system-x86_64 -machine q35 -cpu host -enable-kvm -m 2048 \
    -smp 2 -drive file=./lubuntu.qcow2,format=qcow2 \
    -device intel-iommu  \
    -netdev tap,id=mynet0,ifname=tap0,script=no,downscript=no \
    -device e1000,netdev=mynet0,mac=52:55:00:d1:55:01 \
    -netdev tap,id=mynet1,ifname=tap1,script=no,downscript=no \
    -device e1000,netdev=mynet1,mac=52:55:00:d1:55:02
```

- 启动虚拟机后，按此前的步骤开启大页、绑定驱动。
- 使用如下命令在 VM1 上启用 pktgen-dpdk。

```
sudo ./pktgen -l 0-2 -n 2 -- -P -m "1.0 2.1"
```

其中 `-l 0-2 -n` 表示使用0-2核心，内存通道数为2，`-P -m "1.0 2.0"` 表示开启混杂模式，并指定1号核心控制端口0，2号核心控制端口1。

- 使用如下命令在 VM2 上启动 l2fwd。

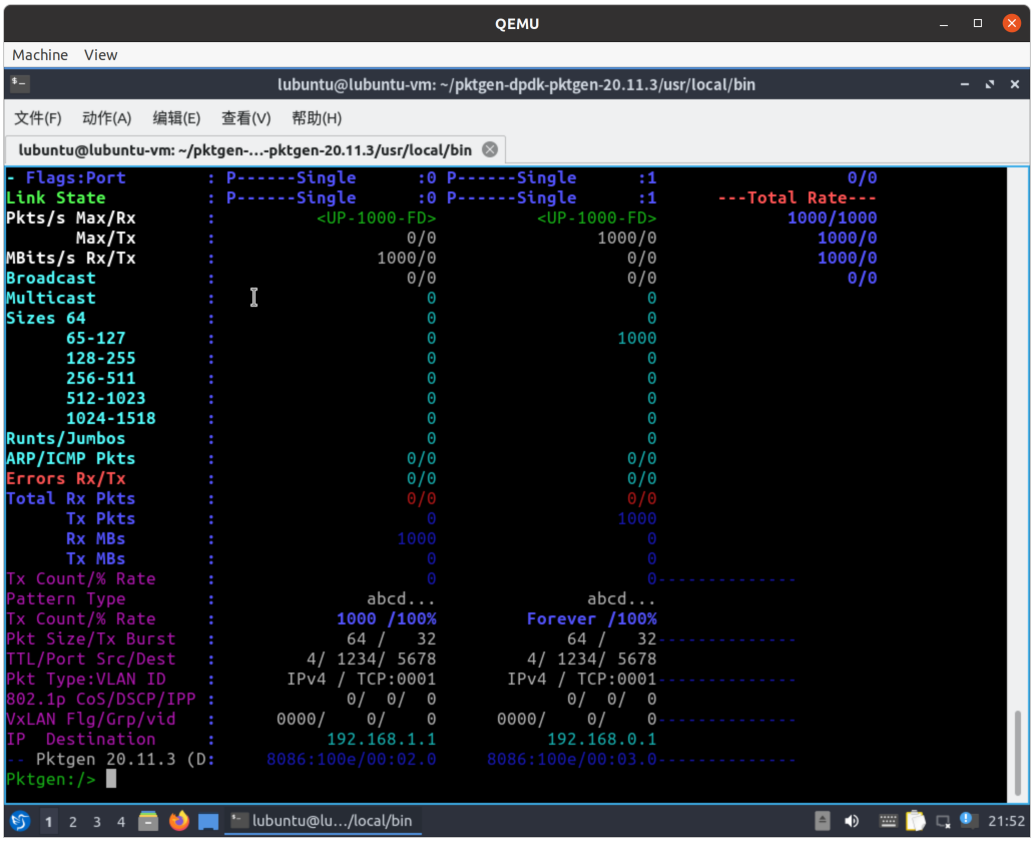
```
sudo ./l2fwd -n 2 -c 0x3 -- -q 4 -p 0x3
```

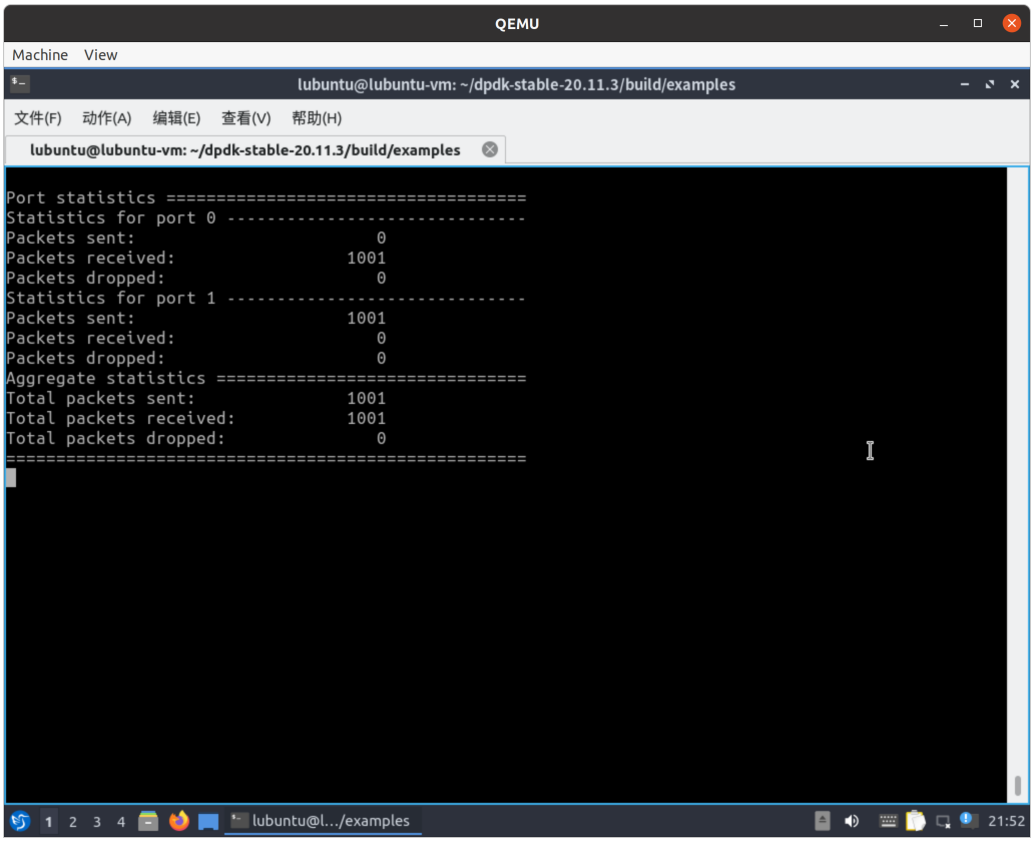
其中 `-n 1 -c 0x3` 表示使用起始的两个核心，内存通道数为2，`-q 4 -p 0x3` 表示分配给每个逻辑内核4个收发队列，开启起始的两个端口。

- 在 VM1 的 pktgen 命令行中输入如下命令，将发送包的目标 MAC 地址指向 VM2 的端口0，开启测试样例。

```
# 设置端口0的发送配置
# l2fwd是第二层转发，不需要IP地址
set 0 dst mac 52:55:00:d1:55:01
# 设置发送数量
set 0 count 1000
# 设置发送包大小
set 0 size 16
start 0
```

从结果中可以看到，pktgen 发出的包被 VM2 顺利转发，并且 pktgen 的端口1也顺利收到了发回的包。





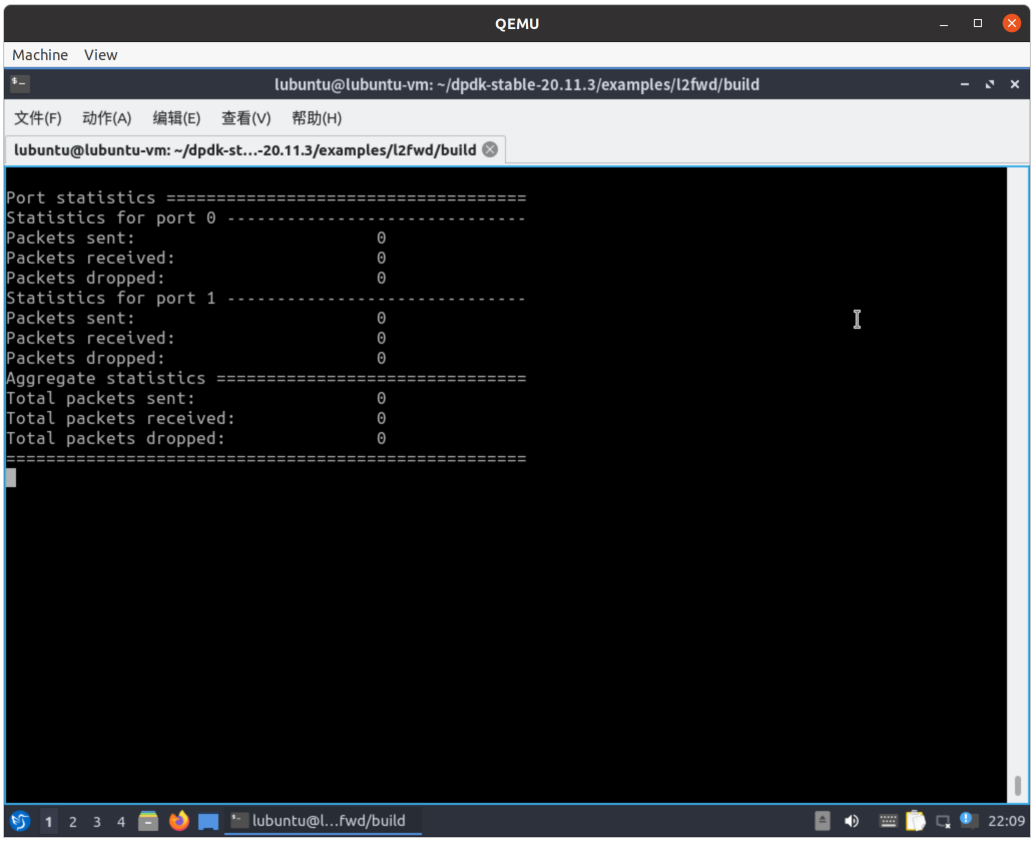
修改 l2fwd 源码

虽然在上节中，我们设置了 pktgen 的目标 MAC 地址，但是由于 l2fwd 本身是默认开启混杂模式的，VM2 的端口0其实是接收了全部流经它的包，这为我们的实验引入了误差。对 l2fwd 源码的修改非常简单，我们只需要注释掉如下代码。

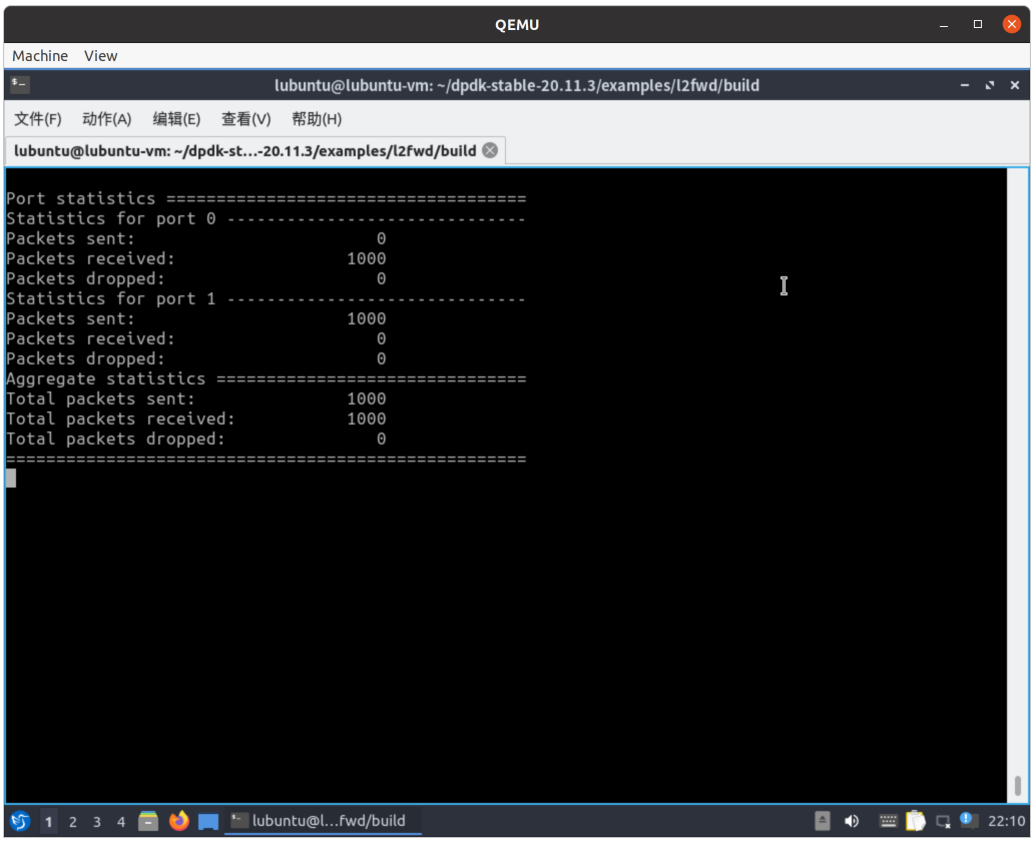
```
rte_eth_promiscuous_enable(portid);
```

再使用 `make` 编译源码。

此时，若我们不指定目标 MAC 地址，则 l2fwd 无法收到 VM1 所发出的包。



使用 `set 0 dst mac 52:55:00:d1:55:01` 指定了目标 MAC 地址后，l2fwd 可以正常接收包。



实验结果

控制变量测试

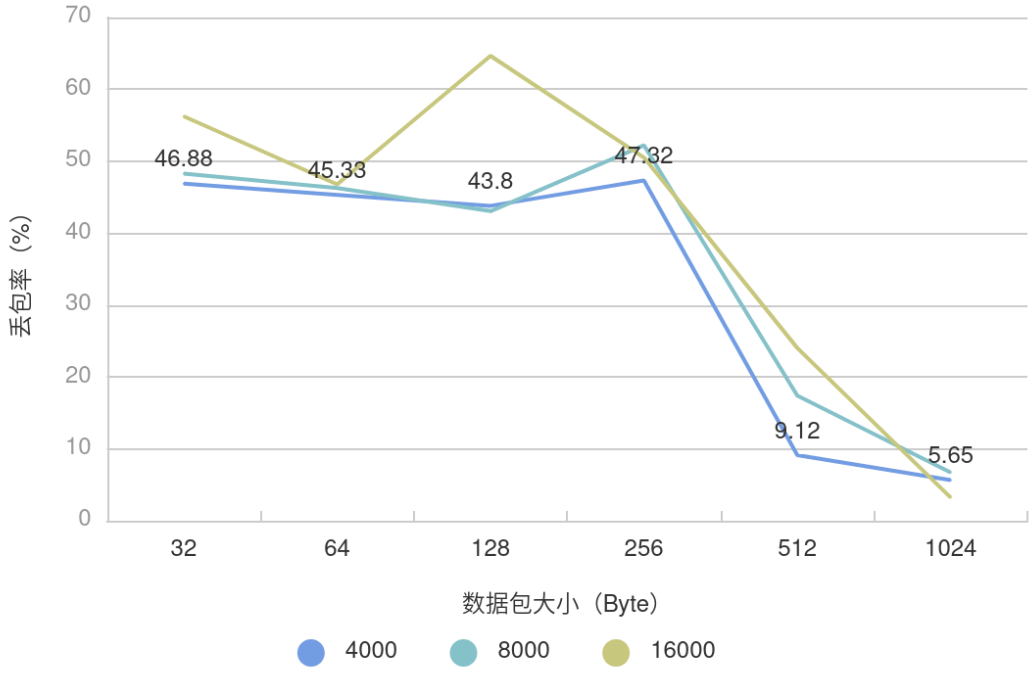
在本次实验中，有如下几个变量会影响网络性能：l2fwd 端启用的核心个数和收发队列的个数以及pktgen端包的数量和包的大小。为了更好地理解各因素对性能的具体影响，在本节中，我们将对这四种因素进行控制变量实验。如果我们手动调整实验参数，实验将相当繁琐，受到RFC2544性能测试的启发，我编写了的Lua脚本（见附录）来自动调整实验参数并记录实验结果。我们使用在两台虚拟机上分别启用两张e1000网卡，并对每组参数进行5次测试来减少实验误差。

- 结果记录：

l2fwd 端启用的核心数	收发队列个数	VM1 发包个数	数据包大小（Byte）	丢包率（%）
2	1	4000	32	46.88
			64	45.33
			128	43.8
			256	47.32
			512	9.12
			1024	5.65
		8000	32	48.28
			64	46.27
			128	43.06
			256	52.22
			512	17.4
			1024	6.7
		16000	32	56.25
			64	46.78
			128	64.65
			256	50.51
			512	24.02
			1024	3.21
	4	4000	32	50.71
			64	35.92
			128	28.12
			256	25.36
			512	18.27
			1024	0
		8000	32	59.79
			64	36
			128	43.3
			256	22.52
			512	16.05
			1024	0
		16000	32	46.7
			64	37.5
			128	47.25
			256	38.66
			512	11.05
			1024	0.04

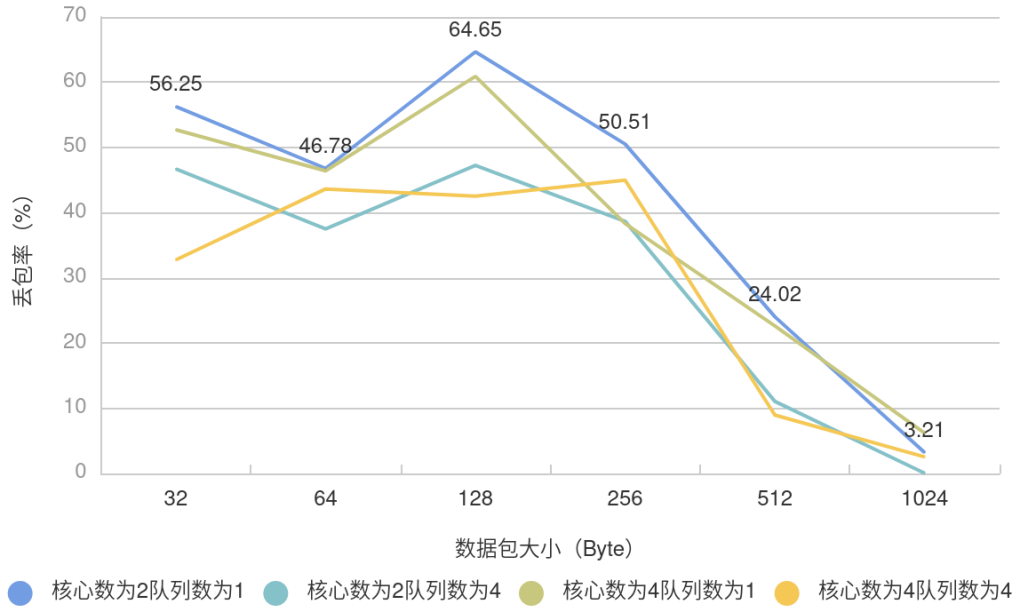
l2fwd 端启用的核心数	收发队列个数	VM1 发包个数	数据包大小（byte）	丢包率（%）
4	1	4000	32	45.17
			64	35.48
			128	39.24
			256	44.45
			512	15.29
			1024	4.83
		8000	32	54.1
			64	47.53
			128	53.83
			256	39.16
			512	19.84
			1024	4.53
		16000	32	52.71
			64	46.4
			128	60.9
			256	38.31
			512	22.62
			1024	6.18
	4	4000	32	46.325
			64	22.44
			128	25.08
			256	26.46
			512	3.06
			1024	0
		8000	32	23.76
			64	43.8
			128	36.94
			256	18.32
			512	17.11
			1024	0
		16000	32	32.75
			64	43.63
			128	42.52
			256	44.98
			512	8.96
			1024	2.53

- 数据可视化：
  - 首先，我们将l2fwd 端启用的核心数和收发队列的个数均设定为1，改变包的数量和大小，所得结果如下图：



可以观察到：当包大小增大时，丢包率将逐渐下降并趋向于0；当数据包的个数增多时，丢包率会有一定的上升。

- 接着，我们固定包的数量为16000，包的大小为1024 Byte，比较不同的核心数和队列个数对性能造成的影响，结果如下图：



可以看到：通过增加CPU核心数和队列个数，可以缓解丢包现象。

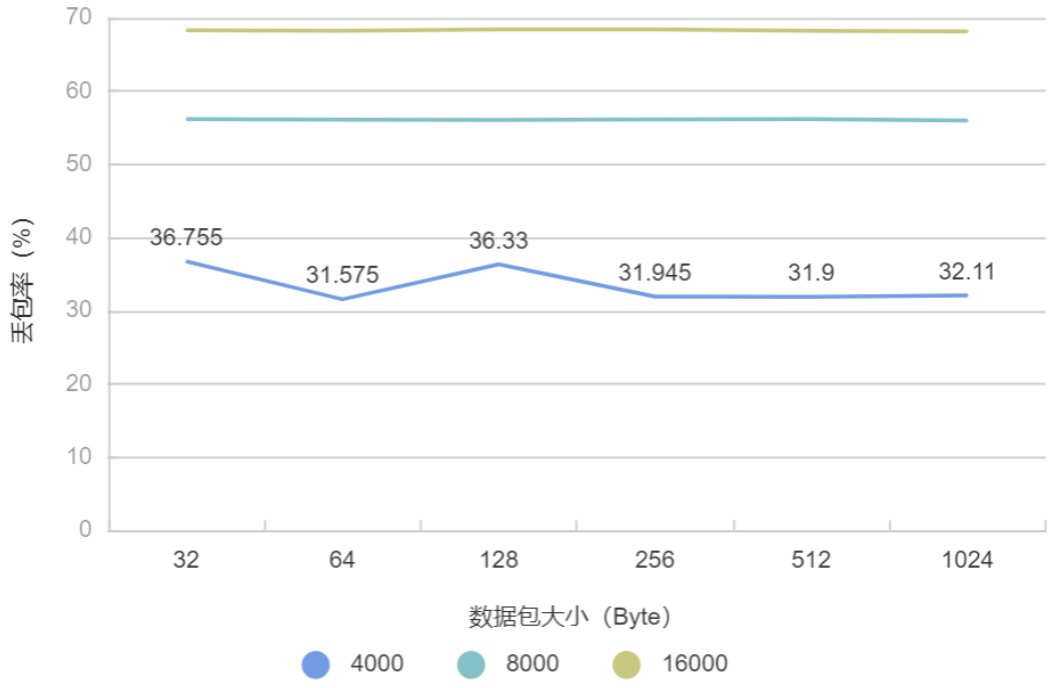
更多实验

受到大作业二的启发，我们思考虚拟网卡的类型是否也会对丢包率造成影响。于是，我将两台虚拟机的网卡更换为半虚拟化的virtio，并将核心数设置为2，队列个数设置4，重复上述实验。

- 结果记录：

l2fwd 端启用的核心数	收发队列个数	VM1 发包个数	数据包大小（byte）	丢包率（%）
2	4	4000	32	36.755
			64	31.575
			128	36.33
			256	31.945
			512	31.9
			1024	32.11
		8000	32	56.18
			64	56.09
			128	56.08
			256	56.16
			512	56.18
			1024	55.99
		16000	32	68.31
			64	68.26
			128	68.44
			256	68.43
			512	68.25
			1024	68.18

- 从实验结果我们可以得到发包个数、数据包大小与丢包率的关系图：



结果还是比较意外的，我们依旧可以观察到随着数据包数量的增加，丢包率依次提高，但是性能更好的virtio网卡却导致了更大的丢包率。

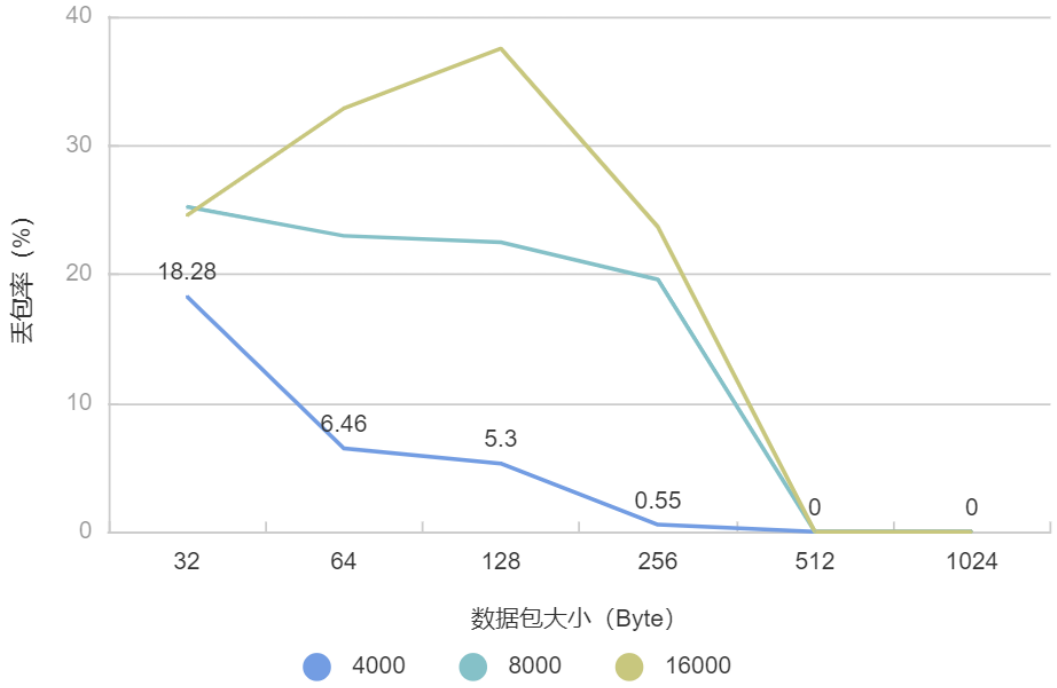
接着，我们改变网络的拓扑结构，还是使用e1000网卡，不同的是VM1端此时仅用一张网卡同时用作收和发。此外，我们还需要更改Lua测试脚本，将收发端口均修改为端口0。

```
local sendport      = "0";
local recvport      = "0";
```

- 结果记录：

l2fwd 端启用的核心数	收发队列个数	VM1 发包个数	数据包大小（byte）	丢包率（%）
2	4	4000	32	18.28
			64	6.46
			128	5.3
			256	0.55
			512	0
			1024	0
		8000	32	25.23
			64	22.98
			128	22.49
			256	19.59
			512	0
			1024	0
		16000	32	24.56
			64	32.87
			128	37.53
			256	23.68
			512	0
			1024	0

- 从实验结果我们可以得到发包个数、数据包大小与丢包率的关系图：



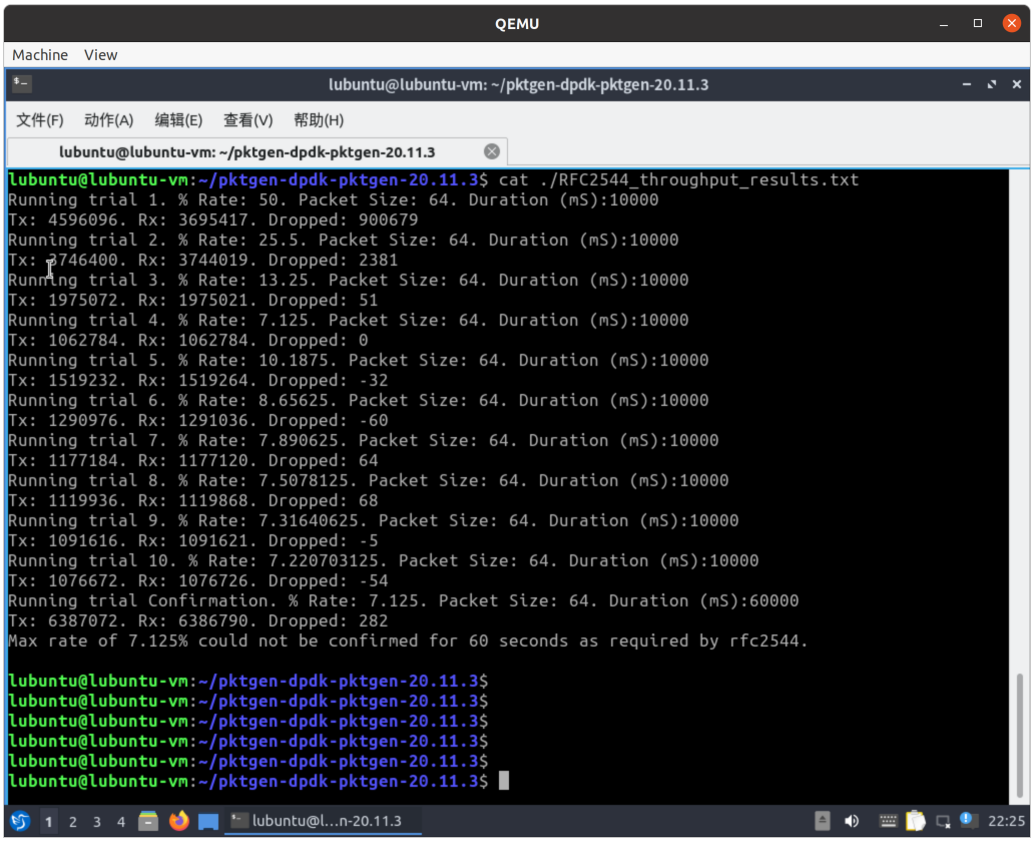
我们同样复现了包大小减小与包数量增加会导致丢包率提升这个结论，并且仅使用一个网卡时的丢包率要远远低于使用两张网卡，在大部分情况下都可以实现0丢包。

扩展实验

为了进一步地探明网络性能的影响因素，我查阅了RFC2544性能测试的原始代码。从中，我了解到网卡的可以通过控制网卡的发送速率（Rate）来实现0丢包，而0丢包时能达到的最大发送速率也可以用来评测网络设备的性能。这里，我们通过修改RFC2544性能测试的源码，来指定收包的MAC地址并限定只使用 64 Bytes大小的包用于测试。

```
local pktSizes      = {64};
pktgen.set_mac(sendport,'dst', "52:55:00:d1:55:01");
```

我们将核心个数指定为2，队列个数指定为4，网卡使用e1000，采用双网卡拓扑结构执行测试，考察包大小为 64 Bytes 时的最大发送速率。实验结果如下：



可以看到，随着发包速率逐步降低，丢包数和丢包率都得以下降，最终发包速率稳定在 7.2%左右。该指标便是目前的网络配置在尽可能保证不丢包的情况下所能达到的最大速率。

结果分析

现象总结

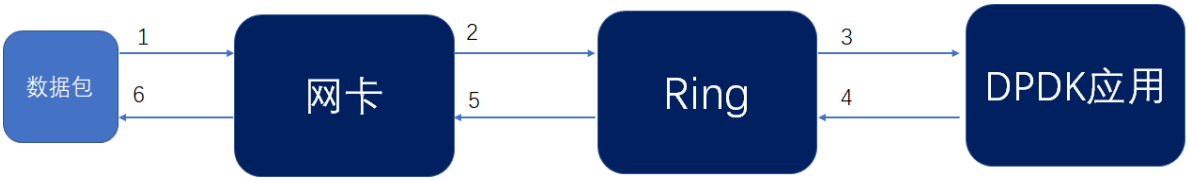
我们先对目前的实验现象进行一个简单的总结：

- 包丢失主要发生在 pktgen 发包的过程中，l2fwd 与 pktgen 收包过程并没有发生丢包，也即包的丢失发生在 `pktgen - br0 - l2fwd` 这个过程中。
- 增大包的个数或是减小包的数量都会导致丢包率增加
- 增加启用核心个数或是增加收发队列数量均可以缓解丢包现象
- virtio的丢包率远大于e1000
- pktgen端使用单网卡用于收发可以缓解丢包现象
- 通过减少pktgen网卡的发送速率（Rate）同样可以缓解丢包

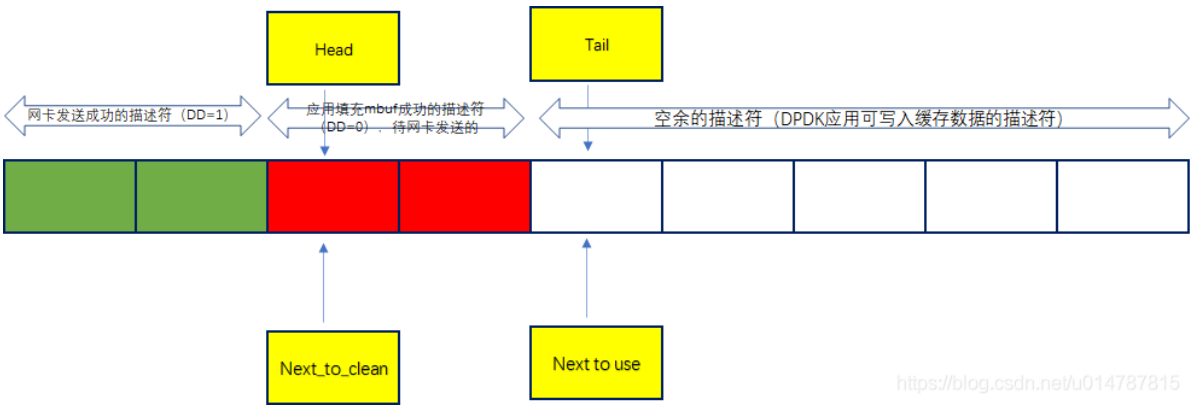
原理分析

为了分析丢包现象的根本原因，我调研了 DPDK 收发包的具体过程。根据参考资料 [17]，我们可以了解到，数据包接收时需要经过以下流程：

- 数据包到达网卡
- 网卡经过DMA操作将数据包从网卡拷贝到收包队列
- DPDK 应用从收包队列中取包

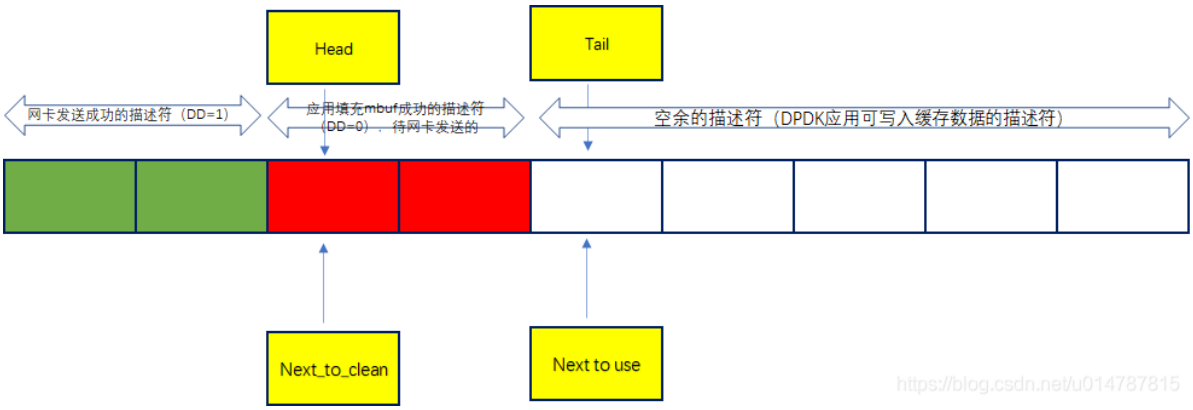


具体来说，收包队列具有以下结构：



可以看到 CPU 并不是在 Ring 描述符中填充完 DMA 地址后立马就对 tail 寄存器的更新，而是等 DMA 可用描述符低于一定阈值时才执行写寄存器更新tail，因此整个过程是异步的。

数据包发送时的流程是类似的，而发包队列具有以下结构：



理论分析

结合前置知识和我们的实验结果，在与同学经过讨论后，我们发现可以用如下公式来描述丢包率：

$$loss = 1 - \frac{v_r t + nb}{v_t t} = 1 - \frac{v_r}{v_t} - \frac{nb}{v_t t}$$

其中，v_r 与 v_t 分别是 l2f2d 收包和 pktgen 发包的速率，而 n 与 b 分别是缓冲区个数和缓冲区大小，t 是发包时间。公式的含义很简单，接收端能处理的最大包个数是发包时间内处理的个数加缓冲区内的包个数。

综合分析

结合原理与公式，我们便可以针对以上实验结果分别给出解释：

- 之所以丢包现象发生在 `pktgen - br0 - l2fwd` 过程中，是因为对于 l2fwd 而言，其转发过程需要历经两次 Ring buffer 并经由 DPDK处理，由于异步更新机制的存在，整个处理时延较长。而双网卡情况下 pktgen 则可以直接通过发包队列高性能发包。也即 v_r 较小而 v_t 较大，这便增加了丢包率。
- 在实验过程中通过 pktgen 的统计界面也可以观察到，增大包的数量或者减少包的个数都会导致 $\frac{v_r}{v_t}$ 变化，当 $\frac{v_r}{v_t}$ 便小时，丢包率便增大。
- 在接收端启用更多核心可以加快接收端的处理速度 v_r，而增大缓冲区个数相当于增加 n 的大小。这都可以缓解丢包现象。

- 由于接收端的瓶颈不在于网卡，而发送端的瓶颈与网卡有关，在启用 virtio 网卡后，增大了 v_t，导致丢包现象更加严重。
- pktgen端使用同一网卡用于收发，减缓了发包速率 v_t，因此丢包率有所下降。
- 发送速率直接改变了发包速率 v_t，因此自然可以缓解丢包现象。

实验感想

本次大作业是本学期虚拟化课程的最后一次大作业，不仅内容较为复杂，也涉及到了前几次作业中所涉及到的网桥搭建、网卡虚拟化和巨页创建的内容。可以说，本次大作业一是对本学期所学习的内容进行了一个系统的复习，二是对我们能灵活运用所学知识的能力进行了一次考验。具体来讲，在本次大作业中，我了解了 DPDK 这一强大的工具，对二层转发机制进行了一些探索。同时，我自身查阅资料、设计实验与自主分析的能力也得到了进一步的锻炼。

经过本学期的课程，我对虚拟化这一本不熟悉的领域有了初步的了解。我相信，无论我们未来从事哪个领域，虚拟化和云计算的大趋势是不变的，而本次课程中所收获的知识与经验则会成为我们面对云时代的宝贵财富。

在此，我想对管老师和各位助教学长表示由衷的感谢。感谢你们准备丰富的课程内容并在课后不厌其烦地为我们解答疑惑。同时，也要感谢陈文浩、窦铱明、叶航宇等同学与我在课下交流讨论。

参考资料

[1] DPDK:Quick Start Guide

[2] DPDK-20.11.3在CentOS8.4上编译运行

[3] DPDK:Linux Drivers

[4] QEMU:Features/VT-d

[5] dpdk-20.11编译和安装

[6] dpdk-pktgen 安装和基本使用

[7] GitHub-pktgen/Pktgen-DPDK

[8] L2 Forwarding Tests

[9] DPDK L2FWD

[10] DPDK的L2FWD实践

[11] Running DPDK Forwarding Applications With Pktgen-DPDK

[12] The Pktgen Application-Document

[13] 3. Environment Abstraction Layer

[14] 使用pktgen-dpdk和l2fwd测试RFC2544

[15] DPDK解析

[16] 在Linux（CentOS）上部署DPDK

[17] DPDK收发包流程分析

附录：测试所用Lua脚本

```
package.path = package.path ..";?.lua;test/?.lua;app/?.lua;../?.lua"

require "Pktgen";

-- define packet sizes to test
local pkt_sizes     = { 32, 64, 128, 256, 512, 1024};
local pkt_nums          = {4000, 8000, 16000};

-- Time in seconds to transmit for
local duration      = 2000;
local pauseTime     = 2000;

-- define the ports in use
local sendport      = "0";
local recvport      = "1";

local initialRate   = 50 ;

local function setupTraffic()
    pktgen.set_mac(sendport,'dst', "52:55:00:d1:55:01");
    pktgen.set(sendport, "count", 0);
end

local function runTrial(pkt_size, pkt_num, rate, duration)
    local num_tx, num_rx, num_dropped;

    pktgen.clr();
    -- pktgen.set(sendport, "rate", rate);
    pktgen.set(sendport, "size", pkt_size);
    pktgen.set(sendport, "count", pkt_num);

    pktgen.start(sendport);

    pktgen.delay(duration);
    pktgen.stop(sendport);

    pktgen.delay(pauseTime);

    statTx = pktgen.portStats(sendport, "port")[tonumber(sendport)];
    statRx = pktgen.portStats(recvport, "port")[tonumber(recvport)];
    num_tx = statTx.opackets;
    num_rx = statRx.ipackets;
    num_dropped = num_tx - num_rx;

    print("Tx: " .. num_tx .. ". Rx: " .. num_rx .. ". Dropped: " ..
num_dropped);
    file:write("Tx: " .. num_tx .. ". Rx: " .. num_rx .. ". Dropped: " ..
num_dropped .. "\n");
    pktgen.delay(pauseTime);
    return num_dropped;
end

local function runThroughputTest(pkt_size, pkt_num)
    local num_temp, num_dropped, trial_rate, drop_rate;
    trial_rate = initialRate;
```

```
    num_dropped = 0;

    for count=1, 5, 1
    do
        num_temp = runTrial(pkt_size, pkt_num, trial_rate, duration);
        num_dropped = num_dropped + num_temp;
    end

    drop_rate = num_dropped / (5*pkt_num);
    print("Drop rate of " .. pkt_num .." packages of " ..pkt_size.. " bytes is "
..drop_rate..'\n\n');
    file:write("Drop rate of "..pkt_num.." packages of " ..pkt_size.. " bytes is
"..drop_rate..'\n\n');
end

function main()
    file = io.open("drop_test_result.txt", "w");
    setupTraffic();
    for _,num in pairs(pkt_nums)
    do
        for _,size in pairs(pkt_sizes)
        do
            runThroughputTest(size, num);
        end
    end
    file:close();
end

main();
```