**安装FEMU**

1. git 拷贝一份 Femu 源码

git clone <https://github.com/ucare-uchicago/femu.git>

1. 进入femu/build-femu，若没有这个文件夹，需自行在同级目录下面创建

cd femu

mkdir build-femu

cd build-femu

1. 将femu/femu-scripts文件夹下面的所有文件复制到build-femu文件夹下。

实际上不需要全部复制，在femu/femu-scripts/femu-copy-scripts.sh文件中制定了需要复制的文件，所以实际上只需要的复制femu-copy-scripts.sh这个脚本到build-femu文件夹下，然后执行即可。

cp ../femu-scripts/femu-copy-scripts.sh .

./femu-copy-scripts.sh .

另一种快捷的方法是直接复制：

cp ../femu-scripts/\*.sh ../build-femu

cp ../femu-scripts/ftk/\* ../build-femu

cp ../femu-scripts/vssd1.conf ../build-femu

1. 执行pkgdep.sh。因为额外需要安装的包和软件已经集合于此脚本中。

sudo ./pkgdep.sh

1. 运行femu-compile.sh

./femu-compile.sh

编译成功后FEMU的二进制文件会出现于x86\_64-softmmu/qemu-system-x86\_64。

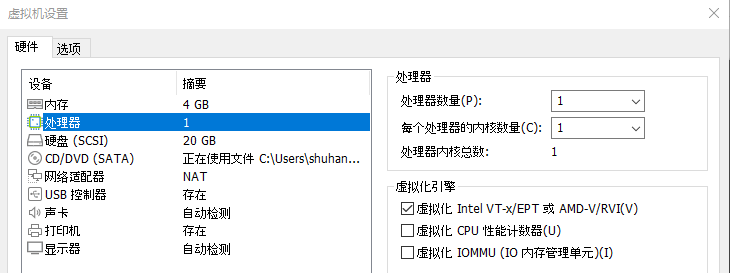
1. 创建虚拟机镜像，虚拟机镜像用来模拟虚拟机的硬盘。在启动虚拟机之前需要创建虚拟镜像文件。创建镜像后放于$HOME/images目录中（run-blackbox.sh/ run-whitebox.sh脚本决定）。

qemu-img create –f qcow2 u14s.qcow2 20G

-f选项用于指定镜像的格式，qcow2格式是QEMU最常用的镜像格式。u14s.qcow2是镜像文件的名字，20G是镜像文件的大小。

1. 镜像文件创建完成后，启动虚拟机并安装系统。

首先需要启用CPU的虚拟化功能（关机状态下勾选）：



然后，检查KVM是否可用：

QEMU使用KVM提升虚拟机性能，如果不启用KVM会导致性能损失。要使用KVM，首先要检查硬件是否有虚拟化支持：

grep -E ‘vmx|svm’ /proc/cpuinfo

因为刚刚已经启用CPU的虚拟化功能，此时会有输出表示硬件有虚拟化支持，

其次要检查KVM模块是否已经加载：

lsmod | grep kvm

若kvm\_intel/kvm\_amd及kvm模块都被显示出来。则KVM模块已经加载。

手动为镜像文件安装系统：

qemu-system-x86\_64 –m 2G –enable-kvm u14s.qcow2 –cdrom ubuntu18.04-beta2-desktop-amd64.iso

-m选项指定虚拟机内存大小；-enable-kvm使用KVM加速；

-cdrom添加u14s.qcow2的安装系统镜像。可在弹出的窗口中操作虚拟机安装操作系统。安装完成后重启虚拟机便会从硬盘（u14s.qcow2）启动。

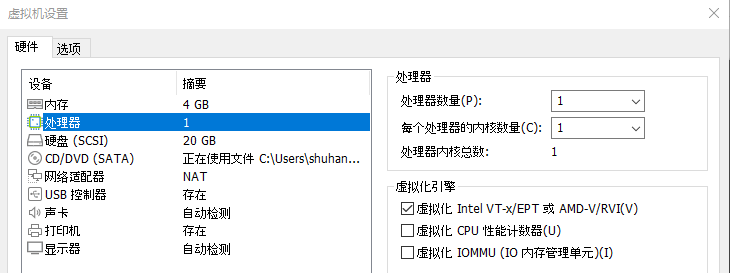


后续若需启动QEMU虚拟机只需执行：

qemu-system-x86\_64 -m 2G u14s.qcow2

PS:6.7.两步也可下载官方提供的femu-vm镜像放于$HOME/images目录之下，因其镜像已安装guest OS，无需手动安装操作系统，可直接执行脚本运行FEMU。

需要注意的是，仍需要启用CPU的虚拟化功能（关机状态下勾选）：



然后，也需要检查KVM是否可用：

QEMU使用KVM提升虚拟机性能，如果不启用KVM会导致性能损失。要使用KVM，首先要检查硬件是否有虚拟化支持：

grep -E ‘vmx|svm’ /proc/cpuinfo

因为刚刚已经启用CPU的虚拟化功能，此时会有输出表示硬件有虚拟化支持，

其次要检查KVM模块是否已经加载：

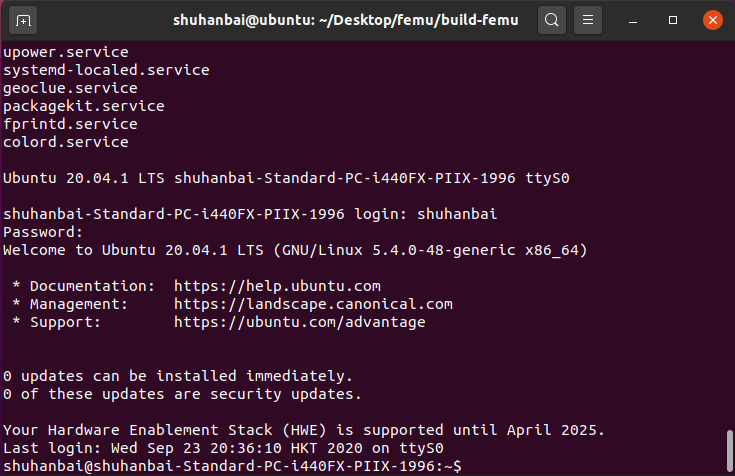
lsmod | grep kvm

若kvm\_intel/kvm\_amd及kvm模块都被显示出来。则KVM模块已经加载。

1. 标盘下运行FEMU

./run-blackbox.sh

运行后会自动启动FEMU：（下图为自建镜像界面）

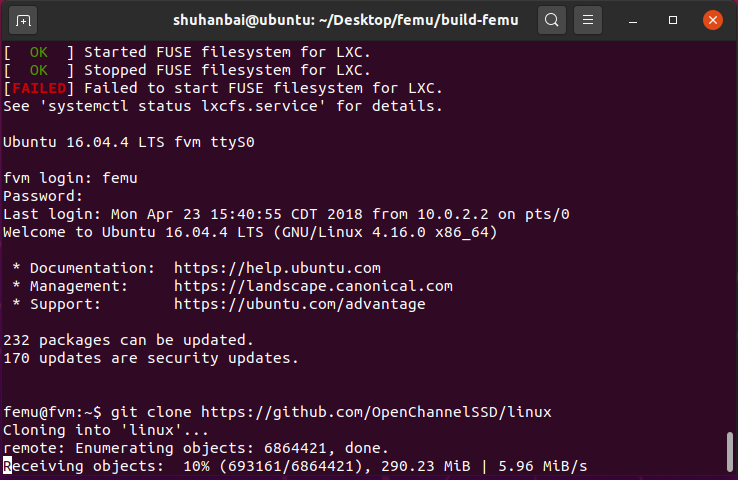


1. OCSSD下运行FEMU

./run-whitebox.sh

(由于vmware虚拟机创建时内存设置为4G，此内存无法全部用于FEMU的运行（这也是给femu镜像安装系统时指令设置内存为2G的原因），需将run-whitebox.sh脚本中-m 4G行改为更小的内存(此例中改为-m 2G，使之与femu镜像设置的内存保持一致)，否则支持LightNVM的内核会编译失败)

与标盘情况相同，运行后会自动启动FEMU。（下图为官方镜像界面）



**下面为OCSSD LightNVM做准备。**

1. 在FEMU whitebox中编译支持LightNVM的内核（pblk-tool）

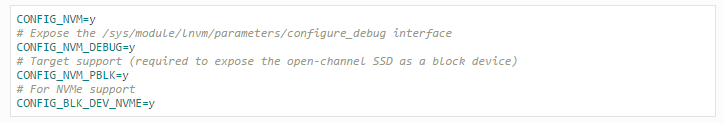
git clone <https://github.com/OpenChannelSSD/linux.git>

cd linux

git checkout for-next

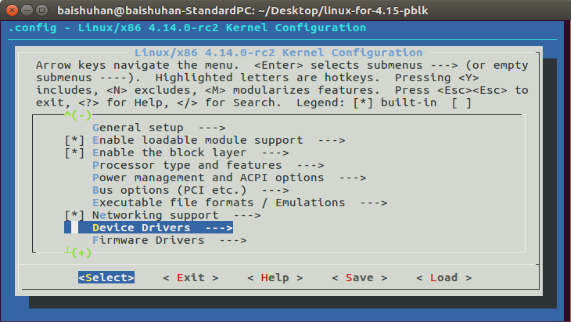
make menuconfig

完成下图对应选项的设置并保存

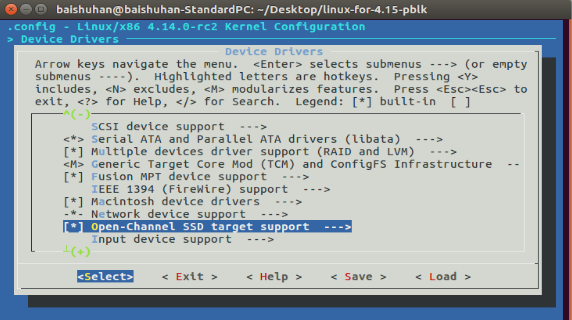


即：

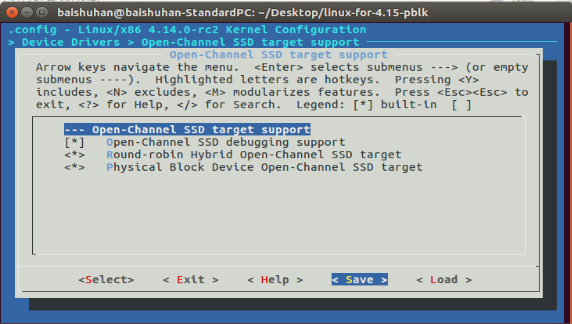
①



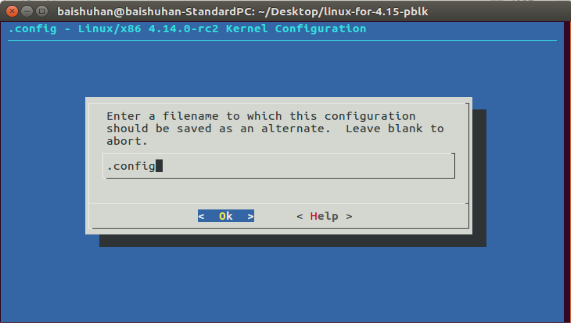
②



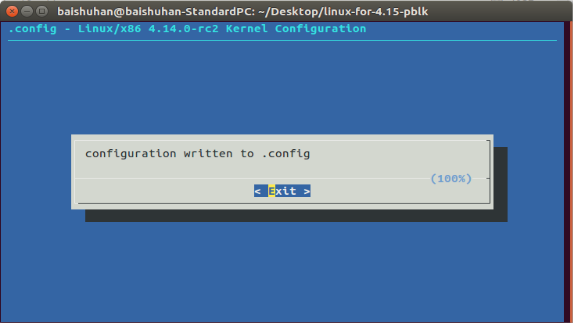
③全部改为Y并保存



④



⑤完成



sudo make -j4 (建议采用多线程编译加快编译速度←命令中-j4表示4线程编译)

sudo make modules\_install

sudo make install

1. 安装用户空间库liblightnvm

git clone <https://github.com/OpenChannelSSD/liblightnvm>.git

cd liblightnvm

make configure

make

sudo make install

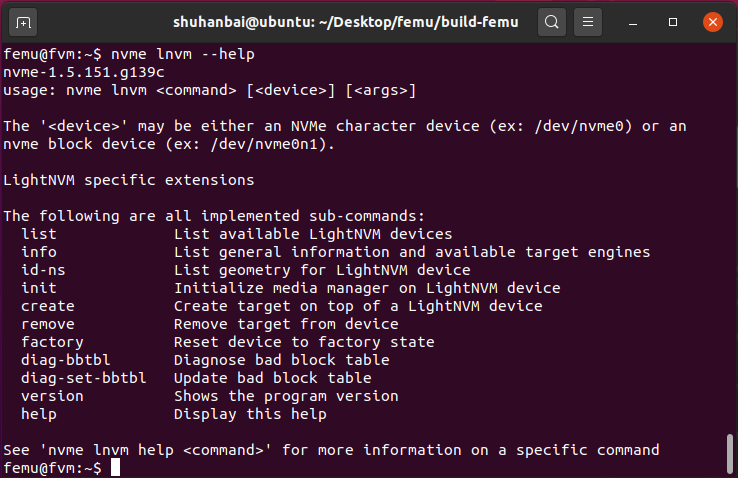
1. 安装于用户空间中管理和控制lightnvm设备的工具nvme-cli

sudo apt install nvme-cli

**命令行测试(nvme-cli/liblightnvm)**

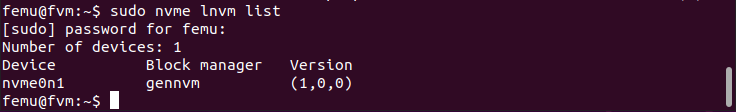
1. LightNVM can be interfaced through the nvme cli tool(nvme-cli).It allows us to create,delete, and manage LightNVM targets and devices.After nvme cli has been installed,its commands can be listed with:

**nvme lnvm --help**



1. To list devices and target types registered with LightNVM.
2. To list registered devices:

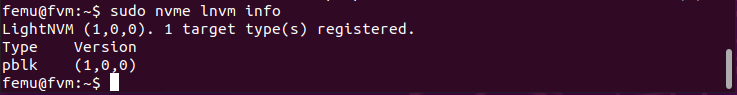
**sudo nvme lnvm list**



查询到已有注册的设备nvme0n1，其dev\_path是/dev/nvme0n1。

1. To list available target type(s) and their version:

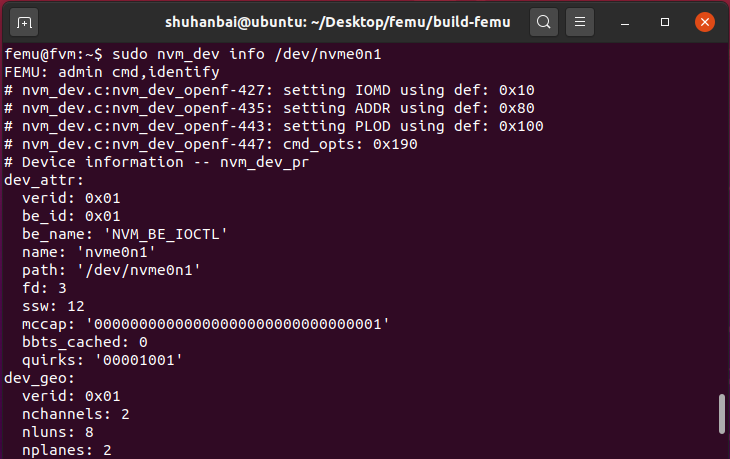
**sudo nvme lnvm info**

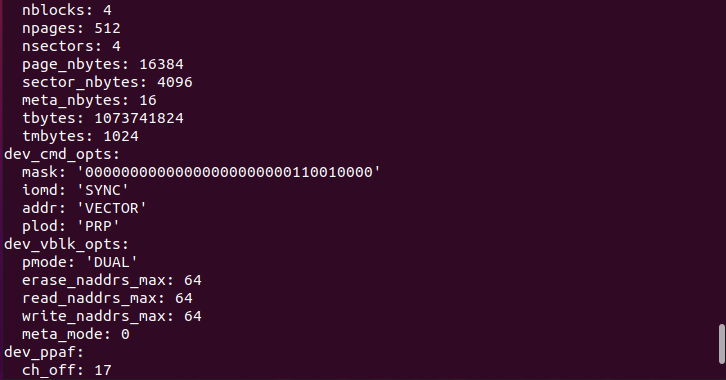


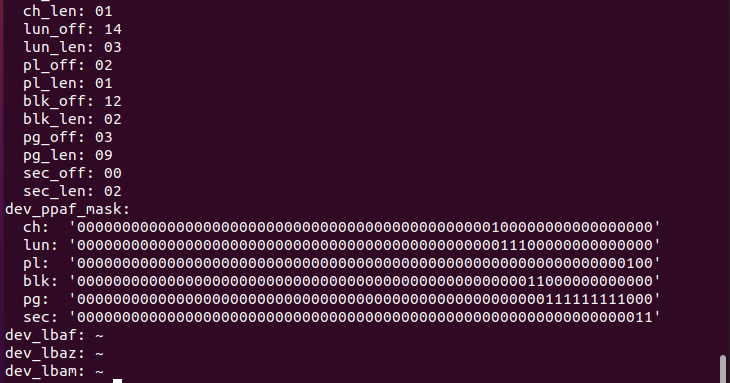
因为上述第10步已经安装pblk-tool，故此时有已注册pblk类target。

1. Retrieve device information from the Open-Channel SSD at the path /dev/nvme0n1：

**sudo nvm\_dev info /dev/nvme0n1**







(dev\_ppaf指设备几何结构的二进制表示。xxx\_len指表示此结构需要的比特数，xxx\_off表示在整体结构的二进制表示中，表示此结构的比特的偏移(结合dev\_ppaf\_mask示例理解))

查询得，OCSSD几何结构如下：

channel 2

lun 8/ch 16 226B=64MB/lun

plane 2/lun 32 225B=32MB/pl

block 4/pl 128 223B=8MB/blk

page 512/blk 216 214B=16KB/pg

sector 4/pg 218 212B=4KB/sec total bytes 230B=1GB=1024MB

1. A device must first have a media registered when it is first uesd, i.e., initialize LightNVM device with media manager:

**1nvme lnvm init -d nvme0n1**

1. To add a target on top of a device registered with the gennvm media manager:

**nvme lnvm create -d $DEVICE -n $TARGET\_NAME -t $TARGET\_TYPE -b $LUN\_BEGIN -e $LUN\_END**

※noted:

1)$DEVICE: Backend device. lnvm devices to list available devices.

2)$TARGET\_NAME: Name of the target to be exposed -> /dev/$TARGET\_NAME

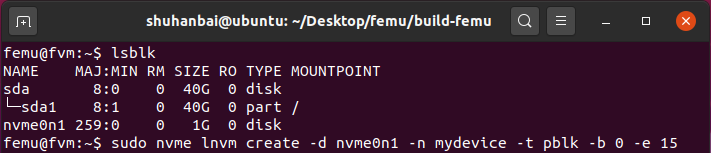
3)$TARGET\_TYPE: Target type. Targets need to be compiled individually before they can be instantiated at run-time. For now, pblk is the available implementations.

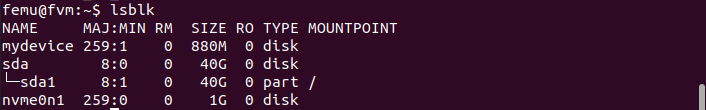
4)$LUN\_BEGIN: Lower bound of the LUN range allocated to the target.

5)$LUN\_END: Higher bound of the LUN range allocated to the target.

For example, to allocate LUNs 0 to 15 to a pblk instance using the NVMe device nvme0n1:

(由上图设备信息中dev\_geo可知LUN数为16)

**nvme lnvm create -d nvme0n1 -n mydevice -t pblk -b 0 -e 15**

A target instance can be removed again by:

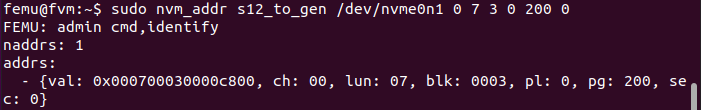
nvme lnvm remove -n $TARGET\_NAME

1. Physical Addressing: Most of the library takes one or more physical addresses as parameter.
2. Generic format: The physical addresses are represented in generic format by the data-structure struct nvm\_addr. One can construct an address by specifying the relative location within the device geometry down to the granularity of a sector:

**sudo nvm\_addr s12\_to\_gen dev\_path ch lun blk pl pg sec**

For example, construct an address for sector 0 within page 200 in block 3 on plane 0 of LUN 7 in channel 0:

**sudo nvm\_addr s12\_to\_gen /dev/nvme0n1 0 7 3 0 200 0**

****

注意结构的顺序为：channel→lun→block→plane→page→sector，且在generic format中各自所占用的比特数分别为：2/2/4/2/4/2。

1. Device format: As the output from the device information shows, then there is a notion of a device format. The library user need not be concerned with the device format as the translation to device format is handled by the library for every part of the interface with the exception of the low-level command-interface nvm\_cmd. However, if one needs an address on device format for nvm\_cmd or another tool such as nvme-cli, then the generic-format can be converted to device format using:

**sudo nvm\_addr gen2dev /dev/nvme0n1 0x000700030000c800**

8

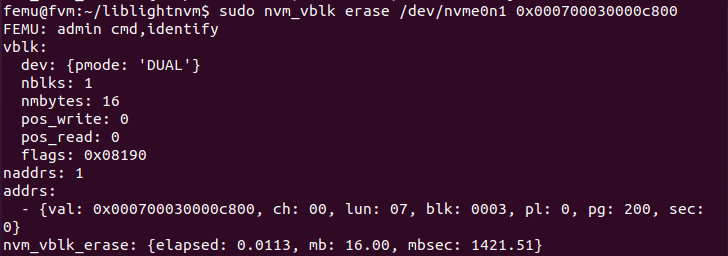
1. Vectorized IO to NAND media

With the basics of obtaining device information and constructing addresses in place one can dive into the task of constructing commands for doing vectorized IO. As the section on background information describes, then there are handful of constraints to handle for IOs to NAND media to succeed.

1. Erase before write

The first constraint to handle is that a block must be erased before it can be written. With the addresses we can now construct a single command with one address and send of the erase:

**sudo nvm\_vblk erase /dev/nvme0n1 0x000700030000c800**

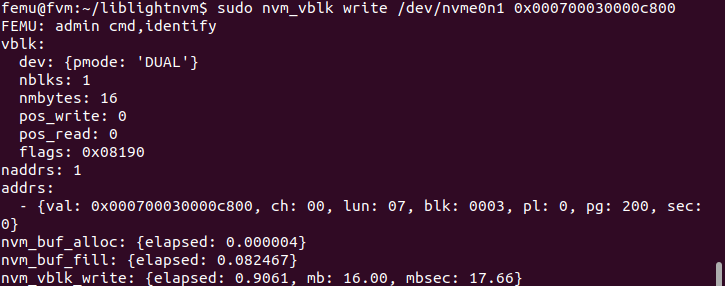


If an erase fails as above then it is because a block is bad. The bad-block-table interface (nvm\_bbt) is provided to communicate and update media state. Introduction to the bad-block-table is given in a later section.

1. Write

The two primary constraints for issuing writes are that they must be at the granularity of a full flash page and contiguous within a block.

**sudo nvm\_vblk write /dev/nvme0n1 0x000700030000c800**



1. Read

We can read a single sector, the data read from device can be written to a file system using the -o FILE option,and the payload is then available by inspection, e.g. with hexdump:(要先写入数据才有读取输出，否则报错)

**sudo nvm\_vblk read /dev/nvme0n1 0x000700030000c800 -o /tmp/dump.bin**

**Hexdump /tmp/dump.bin -C -n 128**

