qemu pwn-基础知识

raycp / 2019-10-21 08:50:52 / 浏览数 3692 安全技术 二进制安全 顶(0) 踩(0)

最近开始研究qemu，想看看qemu逃逸相关的知识，看了一些资料，学习pwn qemu首先要对qemu的一些基础知识有一定的掌握。

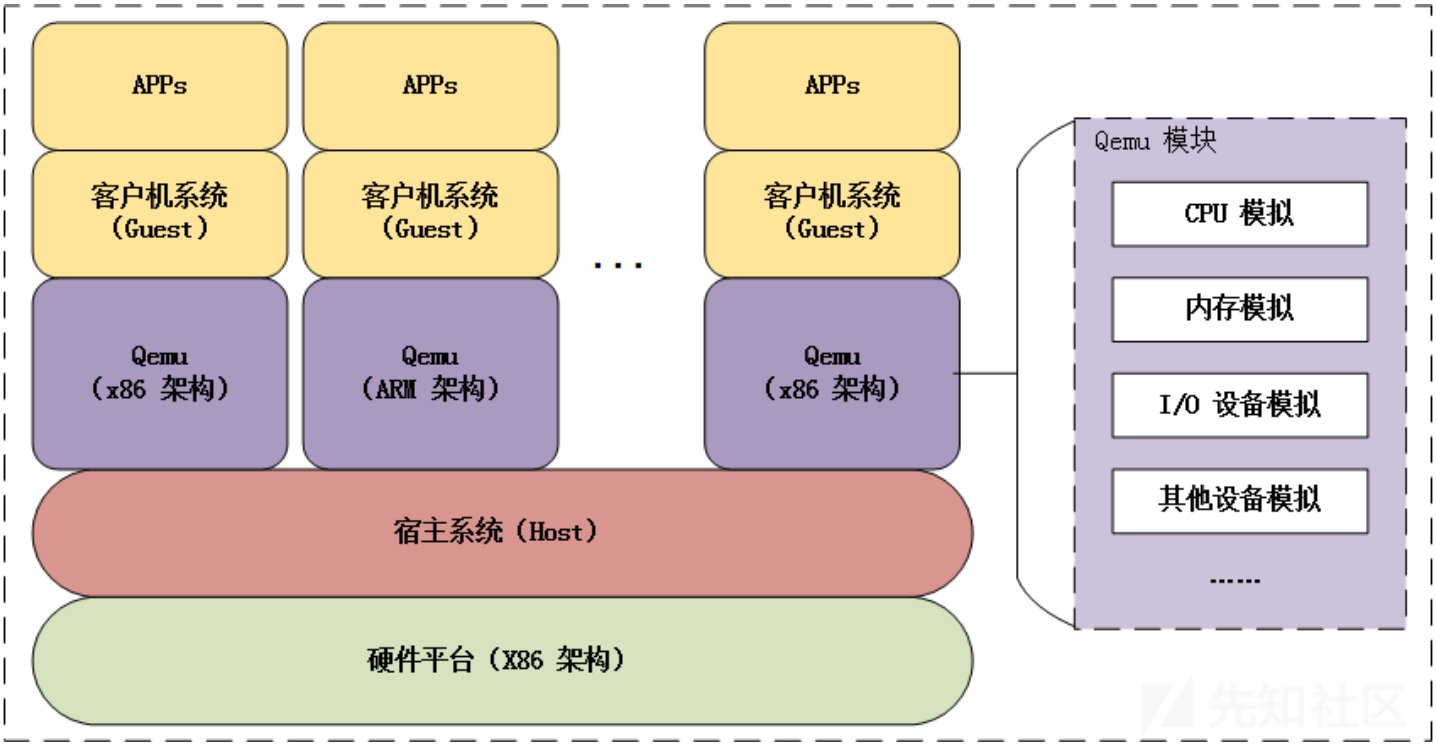
qemu
是纯软件实现的虚拟化模拟器，几乎可以模拟任何硬件设备。当然虚拟化因为性能的原因是无法直接代替硬件的，到那时它对于实验以及测试是非常方便的。

目前qemu出问题比较多的地方以及比赛中出题目的形式都在在设备模拟中，因此后续也会将关注点主要放在设备模拟上。

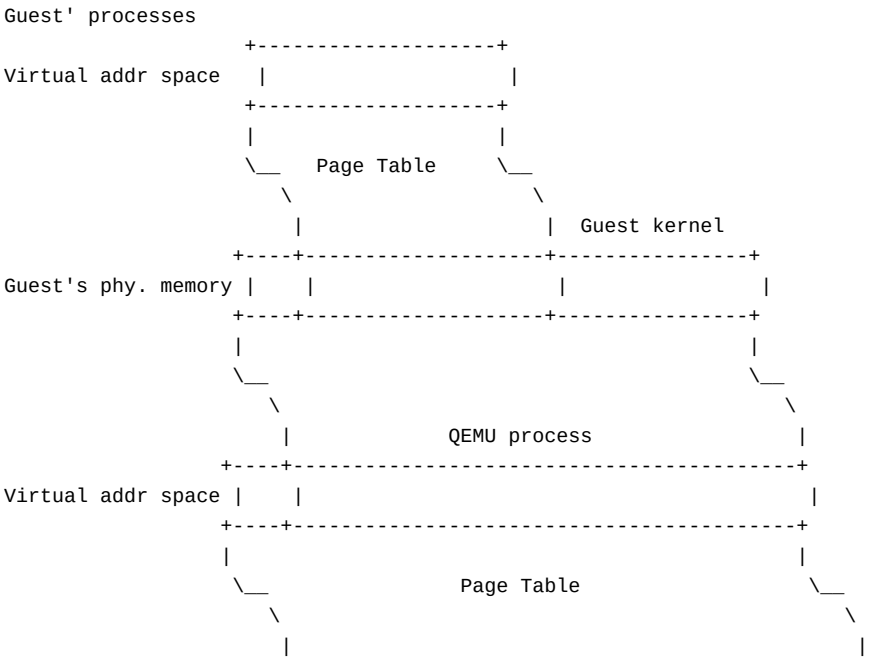
下一篇将主要是Blizzard CTF 2017 Strng的题解，所以本次的基础知识以该题代码以及qemu源码为例进行解释。

qemu概述

运行的每个qemu虚拟机都相应的是一个qemu进程，从本质上看，虚拟出的每个虚拟机对应 host 上的一个 qemu 进程，而虚拟机的执行线程（如 CPU 线程、I/O 线程等）对应 qemu 进程的一个线程。



其次我们需要知道的是，虚拟机所对应的内存结构。根据文章VM escape-QEMU Case Study，qemu虚拟机内存所对应的真实内存结构如下：



```
                 +----+-----------------------------------------------++
Physical memory    |    |                                                ||
                 +----+-----------------------------------------------++
```

qemu进行会为虚拟机mmap分配出相应虚拟机申请大小的内存，用于给该虚拟机当作物理内存（在虚拟机进程中只会看到虚拟地址）。

如strng启动的命令为：

```
./qemu-system-x86_64 \
   -m 1G \
   -device strng \
   -hda my-disk.img \
   -hdb my-seed.img \
   -nographic \
   -L pc-bios/ \
   -enable-kvm \
   -device e1000,netdev=net0 \
   -netdev user,id=net0,hostfwd=tcp::5555-:22
```

qemu虚拟机对应的内存为1G，虚拟机启动后查看qemu的地址空间，可以看到存在一个大小为0x40000000内存空间，即为该虚拟机的物理内存。

```
0x7fe37fafe000     0x7fe37fbfe000 rw-p   100000 0
   0x7fe37fbfe000     0x7fe37fbff000 ---p     1000 0
   0x7fe37fbff000     0x7fe37fcff000 rw-p   100000 0
   0x7fe37fcff000     0x7fe37fd00000 ---p     1000 0
   0x7fe37fd00000     0x7fe37fe00000 rw-p   100000 0
   0x7fe37fe00000     0x7fe3bfe00000 rw-p 40000000 0   //■■■■■■■■
```

如果我们在qemu虚拟机中申请一段内存空间，该如何才能在宿主机中找到该内存呢？

首先将qemu虚拟机中相应的虚拟地址转化成物理地址，该物理地址即为qemu进程为其分配出来的相应偏移，利用该地址加上偏移即是该虚拟地址对应在宿主机中的地址。

仍然是在strng虚拟机中，运行以下程序：

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <stdlib.h>
#include <fcntl.h>
#include <assert.h>
#include <inttypes.h>

#define PAGE_SHIFT  12
#define PAGE_SIZE   (1 << PAGE_SHIFT)
#define PFN_PRESENT (1ull << 63)
#define PFN_PFN     ((1ull << 55) - 1)

int fd;

uint32_t page_offset(uint32_t addr)
{
   return addr & ((1 << PAGE_SHIFT) - 1);
}

uint64_t gva_to_gfn(void *addr)
{
   uint64_t pme, gfn;
   size_t offset;
   offset = ((uintptr_t)addr >> 9) & ~7;
   lseek(fd, offset, SEEK_SET);
   read(fd, &pme, 8);
   if (!(pme & PFN_PRESENT))
       return -1;
   gfn = pme & PFN_PFN;
   return gfn;
}

uint64_t gva_to_gpa(void *addr)
{
   uint64_t gfn = gva_to_gfn(addr);
   assert(gfn != -1);
```

```
   return (gfn << PAGE_SHIFT) | page_offset((uint64_t)addr);
}

int main()
{
   uint8_t *ptr;
   uint64_t ptr_mem;

   fd = open("/proc/self/pagemap", O_RDONLY);
   if (fd < 0) {
       perror("open");
       exit(1);
   }

   ptr = malloc(256);
   strcpy(ptr, "Where am I?");
   printf("%s\n", ptr);
   ptr_mem = gva_to_gpa(ptr);
   printf("Your physical address is at 0x%"PRIx64"\n", ptr_mem);

   getchar();
   return 0;
}
```

其中gva_to_gpa即是虚拟地址转化为相应的物理地址的函数，具体原理可以去搜索地址转化去了解。

由于strng虚拟机为32位的，所以编译的命令为：

```
gcc -m32 -O0 mmu.c -o mmu
```

使用命令`scp -P5555 mmu ubuntu@127.0.0.1:/home/ubuntu`将其传至虚拟机中，最后运行得到结果：

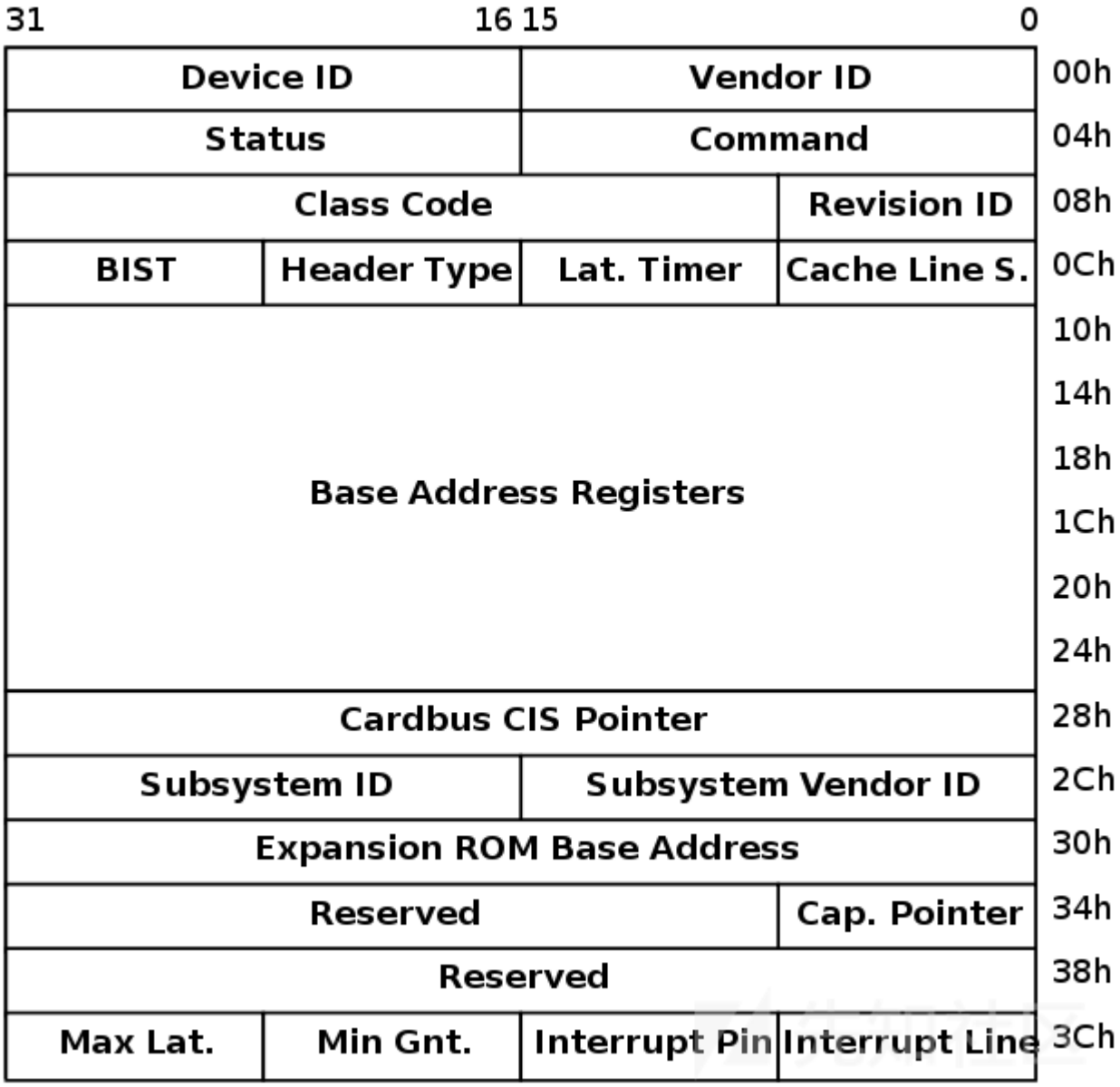
```
ubuntu@ubuntu:~$ sudo ./mmu
Where am I?
Your physical address is at 0x33cd6008
```

从上面我们知道了虚拟机对应的内存地址在qemu进程中的地址为`0x7fe37fe00000`到`0x7fe3bfe00000`，因此相应的字符串地址为`0x7fe37fe00000+0x33cd6008`，在

```
pwndbg> x/s 0x7fe37fe00000+0x33cd6008
0x7fe3b3ad6008: "Where am I?"
```

pci设备地址空间

PCI设备都有一个配置空间（PCI Configuration
Space），其记录了关于此设备的详细信息。大小为256字节，其中头部64字节是PCI标准规定的，当然并非所有的项都必须填充，位置是固定了，没有用到可以填充0。前1



比较关键的是其6个BAR(Base Address Registers)，BAR记录了设备所需要的地址空间的类型，基址以及其他属性。BAR的格式如下：

Memory Space BAR Layout

31 - 4	3	2 - 1	0
16-Byte Aligned Base Address	Prefetchable	Type	Always 0

I/O Space BAR Layout

31 - 2	1	0
4-Byte Aligned Base Address	Reserved	Always 1

设备可以申请两类地址空间，memory space和I/O space，它们用BAR的最后一位区别开来。

当BAR最后一位为0表示这是映射的I/O内存，为1是表示这是I/O端口，当是I/O内存的时候1-2位表示内存的类型，bit 2为1表示采用64位地址，为0表示采用32位地址。bit1为1表示区间大小超过1M，为0表示不超过1M。bit3表示是否支持可预取。

而相对于I/O内存，当最后一位为1时表示映射的I/O端口。I/O端口一般不支持预取，所以这里是29位的地址。

通过memory space访问设备I/O的方式称为memory mapped I/O，即MMIO，这种情况下，CPU直接使用普通访存指令即可访问设备I/O。

通过I/O space访问设备I/O的方式称为port I/O，或者port mapped I/O，这种情况下CPU需要使用专门的I/O指令如`IN/OUT`访问I/O端口。

关于MMIO和PMIO，维基百科的描述是：

```
Memory-mapped I/O (MMIO) and port-mapped I/O (PMIO) (which is also called
isolated I/O) are two complementary methods of performing input/output (I/O)
between the central processing unit (CPU) and peripheral devices in a computer.
An alternative approach is using dedicated I/O processors, commonly known as
channels on mainframe computers, which execute their own instructions.
```

在MMIO中，内存和I/O设备共享同一个地址空间。
MMIO是应用得最为广泛的一种I/O方法，它使用相同的地址总线来处理内存和I/O设备，I/O设备的内存和寄存器被映射到与之相关联的地址。当CPU访问某个内存地址时，

在PMIO中，内存和I/O设备有各自的地址空间。
端口映射I/O通常使用一种特殊的CPU指令，专门执行I/O操作。在Intel的微处理器中，使用的指令是IN和OUT。这些指令可以读/写1,2,4个字节（例如：`outb`, `outw`,
`outl`）到IO设备上。I/O设备有一个与内存不同的地址空间，为了实现地址空间的隔离，要么在CPU物理接口上增加一个I/O引脚，要么增加一条专用的I/O总线。由于I/O地
I/O)。

qemu中查看pci设备

下面通过在qemu虚拟机中查看pci设备来进一步增进理解，仍然是基于strng这道题的qemu虚拟机。

`lspci`命令用于显示当前主机的所有PCI总线信息，以及所有已连接的PCI设备信息。

pci设备的寻址是由总线、设备以及功能构成。如下所示：

```
ubuntu@ubuntu:~$ lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 03)
00:02.0 VGA compatible controller: Device 1234:1111 (rev 02)
00:03.0 Unclassified device [00ff]: Device 1234:11e9 (rev 10)
00:04.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 03)
```

`xx:yy:z`的格式为■■:■■:■■的格式。

可以使用lspci命令以树状的形式输出pci结构：

```
ubuntu@ubuntu:~$ lspci -t -v
-[0000:00]-+-00.0  Intel Corporation 440FX - 82441FX PMC [Natoma]
           +-01.0  Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
           +-01.1  Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
           +-01.3  Intel Corporation 82371AB/EB/MB PIIX4 ACPI
           +-02.0  Device 1234:1111
           +-03.0  Device 1234:11e9
           \-04.0  Intel Corporation 82540EM Gigabit Ethernet Controller
```

其中`[0000]`表示pci的域， PCI域最多可以承载256条总线。 每条总线最多可以有32个设备，每个设备最多可以有8个功能。

总之每个 PCI 设备有一个总线号, 一个设备号, 一个功能号标识。PCI 规范允许单个系统占用多达 256 个总线, 但是因为 256 个总线对许多大系统是不够的, Linux 现在支持 PCI 域。每个 PCI 域可以占用多达 256 个总线. 每个总线占用 32 个设备, 每个设备可以是 一个多功能卡(例如一个声音设备, 带有一个附加的 CD-ROM 驱动)有最多 8 个功能。

PCI 设备通过`VendorIDs`、`DeviceIDs`、以及`Class Codes`字段区分：

```
ubuntu@ubuntu:~$ lspci -v -m -n -s 00:03.0
Device: 00:03.0
Class:  00ff
Vendor: 1234
Device: 11e9
SVendor:        1af4
SDevice:        1100
PhySlot:        3
Rev:    10

ubuntu@ubuntu:~$ lspci -v -m -s 00:03.0
Device: 00:03.0
Class:  Unclassified device [00ff]
Vendor: Vendor 1234
Device: Device 11e9
SVendor:        Red Hat, Inc
SDevice:        Device 1100
PhySlot:        3
```

```
Rev:    10
```

也可通过查看其config文件来查看设备的配置空间，数据都可以匹配上，如前两个字节1234为vendor id：

```
ubuntu@ubuntu:~$ hexdump /sys/devices/pci0000\:00/0000\:00\:03.0/config
0000000 1234 11e9 0103 0000 0010 00ff 0000 0000
0000010 1000 febf c051 0000 0000 0000 0000 0000
0000020 0000 0000 0000 0000 0000 0000 1af4 1100
0000030 0000 0000 0000 0000 0000 0000 0000 0000
```

查看设备内存空间：

```
ubuntu@ubuntu:~$ lspci -v -s 00:03.0 -x
00:03.0 Unclassified device [00ff]: Device 1234:11e9 (rev 10)
       Subsystem: Red Hat, Inc Device 1100
       Physical Slot: 3
       Flags: fast devsel
       Memory at febf1000 (32-bit, non-prefetchable) [size=256]
       I/O ports at c050 [size=8]
00: 34 12 e9 11 03 01 00 00 10 00 ff 00 00 00 00 00
10: 00 10 bf fe 51 c0 00 00 00 00 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 f4 1a 00 11
30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

可以看到该设备有两个空间：BAR0为MMIO空间，地址为febf1000，大小为256；BAR1为PMIO空间，端口地址为0xc050，大小为8。

可以通过查看resource文件来查看其相应的内存空间：

```
ubuntu@ubuntu:~$ ls -la /sys/devices/pci0000\:00/0000\:00\:03.0/
...
-r--r--r--  1 root root 4096 Aug  1 03:40 resource
-rw-------  1 root root  256 Jul 31 13:18 resource0
-rw-------  1 root root    8 Aug  1 04:01 resource1
...
```

resource文件包含其它相应空间的数据，如resource0（MMIO空间）以及resource1（PMIO空间）：

```
ubuntu@ubuntu:~$ cat /sys/devices/pci0000\:00/0000\:00\:03.0/resource
0x00000000febf1000 0x00000000febf10ff 0x0000000000040200
0x000000000000c050 0x000000000000c057 0x0000000000040101
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 0x0000000000000000
```

每行分别表示相应空间的起始地址（start-address）、结束地址（end-address）以及标识位（flags）。

qemu中访问I/O空间

存在mmio与pmio，那么在系统中该如何访问这两个空间呢？访问mmio与pmio都可以采用在内核态访问或在用户空间编程进行访问。

访问mmio

编译内核模块，在内核态访问mmio空间，示例代码如下：

```
#include <asm/io.h>
#include <linux/ioport.h>

long addr=ioremap(ioaddr,iomemsize);
readb(addr);
readw(addr);
readl(addr);
readq(addr);//qwords=8 btyes

writeb(val,addr);
writew(val,addr);
writel(val,addr);
writeq(val,addr);
iounmap(addr);
```

还有一种方式是在用户态访问mmio空间，通过映射resource0文件实现内存的访问，示例代码如下：

```
#include <assert.h>
#include <fcntl.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <unistd.h>
#include<sys/io.h>


unsigned char* mmio_mem;

void die(const char* msg)
{
   perror(msg);
   exit(-1);
}



void mmio_write(uint32_t addr, uint32_t value)
{
   *((uint32_t*)(mmio_mem + addr)) = value;
}

uint32_t mmio_read(uint32_t addr)
{
   return *((uint32_t*)(mmio_mem + addr));
}




int main(int argc, char *argv[])
{

   // Open and map I/O memory for the strng device
   int mmio_fd = open("/sys/devices/pci0000:00/0000:00:04.0/resource0", O_RDWR | O_SYNC);
   if (mmio_fd == -1)
       die("mmio_fd open failed");

   mmio_mem = mmap(0, 0x1000, PROT_READ | PROT_WRITE, MAP_SHARED, mmio_fd, 0);
   if (mmio_mem == MAP_FAILED)
       die("mmap mmio_mem failed");

   printf("mmio_mem @ %p\n", mmio_mem);

   mmio_read(0x128);
       mmio_write(0x128, 1337);

}
```

访问pmio

编译内核模块，在内核空间访问pmio空间，示例代码如下：

```
#include <asm/io.h>
#include <linux/ioport.h>

inb(port);  //■■■■■
inw(port);  //■■■■■
inl(port);  //■■■■■

outb(val,port); //■■■■
outw(val,port); //■■■■
outl(val,port); //■■■■
```

用户空间访问则需要先调用iopl函数申请访问端口，示例代码如下：

```
#include <sys/io.h >

iopl(3);
inb(port);
inw(port);
inl(port);

outb(val,port);
outw(val,port);
outl(val,port);
```

QOM编程模型

QEMU提供了一套面向对象编程的模型——QOM（QEMU Object Module），几乎所有的设备如CPU、内存、总线等都是利用这一面向对象的模型来实现的。

由于qemu模拟设备以及CPU等，既有相应的共性又有自己的特性，因此使用面向对象来实现相应的程序是非常高效的，可以像理解C++或其它面向对象语言来理解QOM。

有几个比较关键的结构体，`TypeInfo`、`TypeImpl`、`ObjectClass`以及`Object`。其中ObjectClass、Object、TypeInfo定义在include/qom/object.h中，TypeImpl定义

`TypeInfo`是用户用来定义一个Type的数据结构，用户定义了一个TypeInfo，然后调用`type_register(TypeInfo )`或者`type_register_static(TypeInfo )`函数，就会生成相应的`TypeImpl`实例，将这个`TypeInfo`注册到全局的`TypeImpl`的hash表中。

```
struct TypeInfo
{
   const char *name;
   const char *parent;
   size_t instance_size;
   void (*instance_init)(Object *obj);
   void (*instance_post_init)(Object *obj);
   void (*instance_finalize)(Object *obj);
   bool abstract;
   size_t class_size;
   void (*class_init)(ObjectClass *klass, void *data);
   void (*class_base_init)(ObjectClass *klass, void *data);
   void (*class_finalize)(ObjectClass *klass, void *data);
   void *class_data;
   InterfaceInfo *interfaces;
};
```

`TypeImpl`的属性与`TypeInfo`的属性对应，实际上qemu就是通过用户提供的TypeInfo创建的TypeImpl的对象。

如下面定义的`pci_test_dev`：

```
static const TypeInfo pci_testdev_info = {
       .name          = TYPE_PCI_TEST_DEV,
       .parent        = TYPE_PCI_DEVICE,
       .instance_size = sizeof(PCITestDevState),
       .class_init    = pci_testdev_class_init,
};
TypeImpl *type_register_static(const TypeInfo *info)
{
   return type_register(info);
}
TypeImpl *type_register(const TypeInfo *info)
{
   assert(info->parent);
   return type_register_internal(info);
}
static TypeImpl *type_register_internal(const TypeInfo *info)
{
   TypeImpl *ti;
   ti = type_new(info);
   type_table_add(ti);
   return ti;
}
```

当所有qemu总线、设备等的`type_register_static`执行完成后，即它们的`TypeImpl`实例创建成功后，qemu就会在`type_initialize`函数中去实例化其对应的`Obje`

每个`type`都有一个相应的`ObjectClass`所对应，其中ObjectClass是所有类的基类

```
struct ObjectClass
{
   /*< private >*/
   Type type;
   GSList *interfaces;
   const char *object_cast_cache[OBJECT_CLASS_CAST_CACHE];
   const char *class_cast_cache[OBJECT_CLASS_CAST_CACHE];
   ObjectUnparent *unparent;
   GHashTable *properties;
};
```

用户可以定义自己的类，继承相应类即可：

```
/* include/qom/object.h */
typedef struct TypeImpl *Type;
typedef struct ObjectClass ObjectClass;
struct ObjectClass
{
       /*< private >*/
       Type type;       /* points to the current Type's instance */
       ...
/* include/hw/qdev-core.h */
typedef struct DeviceClass {
       /*< private >*/
       ObjectClass parent_class;
       /*< public >*/
       ...
/* include/hw/pci/pci.h */
typedef struct PCIDeviceClass {
       DeviceClass parent_class;
       ...
```

可以看到类的定义中父类都在第一个字段，使得可以父类与子类直接实现转换。一个类初始化时会先初始化它的父类，父类初始化完成后，会将相应的字段拷贝至子类同时将

```
static void pci_testdev_class_init(ObjectClass *klass, void *data)
{
       DeviceClass *dc = DEVICE_CLASS(klass);
       PCIDeviceClass *k = PCI_DEVICE_CLASS(klass);
       k->init = pci_testdev_init;
       k->exit = pci_testdev_uninit;
       ...
       dc->desc = "PCI Test Device";
       ...
}
```

最后一个是Object对象：

```
struct Object
{
   /*< private >*/
   ObjectClass *class;
   ObjectFree *free;
   GHashTable *properties;
   uint32_t ref;
   Object *parent;
};
```

Object对象为何物？Type以及ObjectClass只是一个类型，而不是具体的设备。TypeInfo结构体中有两个函数指针：instance_init以及class_init。class_ini

the Object constructor and destructor functions (registered by the respective Objectclass constructors) will now only get call
Object■■■■■■■■■■■■■■Objectclass■■■■■■■■■■■■■■■■■-device■■■■■■■■■■■■■■■■■■■■■■■■■■■■PCI■■■■■

Object示例如下所示：

```
/* include/qom/object.h */
typedef struct Object Object;
struct Object
{
       /*< private >*/
       ObjectClass *class; /* points to the Type's ObjectClass instance */
       ...
```

```
/* include/qemu/typedefs.h */
typedef struct DeviceState DeviceState;
typedef struct PCIDevice PCIDevice;
/* include/hw/qdev-core.h */
struct DeviceState {
       /*< private >*/
       Object parent_obj;
       /*< public >*/
       ...
/* include/hw/pci/pci.h */
struct PCIDevice {
       DeviceState qdev;
       ...
struct YourDeviceState{
       PCIDevice pdev;
       ...
```

（QOM will use instace_size as the size to allocate a Device Object, and then it invokes the instance_init ）

QOM会为设备Object分配instace_size大小的空间，然后调用instance_init函数（在Objectclass的class_init函数中定义）：

```
static int pci_testdev_init(PCIDevice *pci_dev)
{
       PCITestDevState *d = PCI_TEST_DEV(pci_dev);
       ...
```

最后便是PCI的内存空间了，qemu使用MemoryRegion来表示内存空间，在include/exec/memory.h中定义。使用MemoryRegionOps结构体来对内存的操作进行表示

```
static const MemoryRegionOps pci_testdev_mmio_ops = {
       .read = pci_testdev_read,
       .write = pci_testdev_mmio_write,
       .endianness = DEVICE_LITTLE_ENDIAN,
       .impl = {
               .min_access_size = 1,
               .max_access_size = 1,
       },
};

static const MemoryRegionOps pci_testdev_pio_ops = {
       .read = pci_testdev_read,
       .write = pci_testdev_pio_write,
       .endianness = DEVICE_LITTLE_ENDIAN,
       .impl = {
               .min_access_size = 1,
               .max_access_size = 1,
       },
};
```

首先使用memory_region_init_io函数初始化内存空间（MemoryRegion结构体），记录空间大小，注册相应的读写函数等；然后调用pci_register_bar来注册BAR

```
/* hw/misc/pci-testdev.c */
#define IOTEST_IOSIZE 128
#define IOTEST_MEMSIZE 2048

typedef struct PCITestDevState {
       /*< private >*/
       PCIDevice parent_obj;
       /*< public >*/

       MemoryRegion mmio;
       MemoryRegion portio;
       IOTest *tests;
       int current;
} PCITestDevState;

static int pci_testdev_init(PCIDevice *pci_dev)
{
       PCITestDevState *d = PCI_TEST_DEV(pci_dev);
       ...
       memory_region_init_io(&d->mmio, OBJECT(d), &pci_testdev_mmio_ops, d,
```

```
                                          "pci-testdev-mmio", IOTEST_MEMSIZE * 2);
    memory_region_init_io(&d->portio, OBJECT(d), &pci_testdev_pio_ops, d,
                                          "pci-testdev-portio", IOTEST_IOSIZE * 2);
    pci_register_bar(pci_dev, 0, PCI_BASE_ADDRESS_SPACE_MEMORY, &d->mmio);
    pci_register_bar(pci_dev, 1, PCI_BASE_ADDRESS_SPACE_IO, &d->portio);
```

到此基本结束了，最后可以去看strng的实现去看一个设备具体是怎么实现的，它的相应的数据结构是怎么写的。

小结

介绍了qemu虚拟机的内存结构以及它的地址转化；以及pci设备的配置空间，比较重要的是BAR信息还有PMIO以及MMIO；最后是QOM模型，如何通过QOM对象来实现

相关文件以及脚本的链接

参考链接

1. QEMU Internals: Big picture overview
2. VM escape-QEMU Case Study
3. 内存映射IO (MMIO) 简介
4. 浅谈内存映射I/O(MMIO)与端口映射I/O(PMIO)的区别
5. PCI设备的地址空间
6. [PCI 设备详解一]
7. Essential QEMU PCI API
8. Writing a PCI Device Driver, A Tutorial with a QEMU Virtual Device
9. QEMU中的对象模型——QOM（介绍篇）
10. How QEMU Emulates Devices
11. Blizzard CTF 2017: Sombra True Random Number Generator (STRNG)