一.KVM虚拟机创建的用户操作

对于用户或者管理员来说，虚拟机的创建有着很多的方法，例如：kvm自带命令行工

具、使用virsh命令来创建、使用具有图形界面的virt-manager等等。但是它们底层实现的原理都是一样的，而且它们基本上都是通过开源的虚拟化库Libvirt来开发的。下面就讲一讲三种用户可以创建虚拟机的方式。

1.1 利用kvm自带命令行工具进行创建

kvm常用命令如下：

| 参数 | 示例 | 说明 |
| --- | --- | --- |
| -hda | -hda /data/windows.img | 指定windows.img作为硬盘镜像 |
| -cdrom | -cdrom /data/windows.iso | 指定windows.iso作为光盘镜像 |
| -boot | -boot c | 从硬盘启动 |
|  | -boot d | 从光盘启动 |
| -m | -m 512 | 分配512M内存给虚拟系统 |
| -vnc | -vnc :0 | 作为vnc服务器 |
| -cpu | -cpu ? | 列出支持的CPU |
|  | -cpu core2duo | 指定CPU为core2duo |
| -smp | -smp 2 | 指定虚拟机有2个CPU |
| -net | -net nic | 为虚拟机网卡（默认为tap0） |
|  | -net tap | 系统分配tap设备（默认为tap0）[1](http://blog.csdn.net/wuyao721/archive/2010/08/05/5790961.aspx" \l "fn.1" \t "_blank) |
|  | -net nic -net tap | 将虚拟机的网卡eth0连接真机里的tap0 |

具体创建一个虚拟机的步骤如下：

（1）生成硬盘镜像文件

[root@host](mailto:root@host):kvm-img create -f rawtest.img 10G

其中“-f raw”指定镜像文件的格式为“raw”，“10G”指定镜像文件大小。

（2）从光盘启动虚拟机来安装操作系统

[root@host](mailto:root@host):kvm -boot d -hda test.img-cdrom test.iso -m 512

其中“-boot d”指定虚拟机从光盘启动，“-hda test.img”指定硬盘镜像的位置，“-cdrom test.iso”指定光盘镜像的位置，“-m 512”指定虚拟机的内存为512M。

（3）安装操作系统后便可直接从硬盘启动虚拟机

[root@host](mailto:root@host):kvm -boot c -hda test.img-m 512

1.2 利用virsh命令行工具进行创建

1.2.1 virsh工具简介

Virsh是由一个名叫libvirt的软件提供的管理工具，提供管理虚拟机比较高级的能力。Virsh可以管理KVM以及xen等虚拟机。

下面是virsh的一些常见的命令行选项：

| 命​令​ | Description |
| --- | --- |
| help | 打​印​基​本​帮​助​信​息​。​ |
| list | 列​出​所​有​客​户​端​。​ |
| dumpxml | 输​出​客​户​端​ XML 配​置​文​件​。​ |
| create | 从​ XML 配​置​文​件​生​成​客​户​端​并​启​动​新​客​户​端​。​ |
| start | 启​动​未​激​活​的​客​户​端​。​ |
| destroy | 强​制​客​户​端​停​止​。​ |
| define | 为​客​户​端​输​出​ XML 配​置​文​件​。​ |
| domid | 显​示​客​户​端​ ID。​ |
| domuuid | 显​示​客​户​端​ UUID。​ |
| dominfo | 显​示​客​户​端​信​息​。​ |
| domname | 显​示​客​户​端​名​称​。​ |
| domstate | 显​示​客​户​端​状​态​。​ |
| quit | 退​出​这​个​互​动​终​端​。​ |
| reboot | 重​新​启​动​客​户​端​。​ |
| restore | 恢​复​以​前​保​存​在​文​件​中​的​客​户​端​。​ |
| resume | 恢​复​暂​停​的​客​户​端​。​ |
| save | 将​客​户​端​当​前​状​态​保​存​到​某​个​文​件​中​。​ |
| shutdown | 关​闭​某​个​域​。​ |
| suspend | 暂​停​客​户​端​。​ |
| undefine | 删​除​与​客​户​端​关​联​的​所​有​文​件​。​ |
| migrate | 将​客​户​端​迁​移​到​另​一​台​主​机​中​。​ |
| 命​令​ | Description |
| setmem | 为​客​户​端​设​定​分​配​的​内​存​。​ |
| setmaxmem | 为​管​理​程​序​设​定​内​存​上​限​。​ |
| setvcpus | 修​改​为​客​户​端​分​配​的​虚​拟​ CPU 数​目​。​ |
| vcpuinfo | 显​示​客​户​端​的​虚​拟​ CPU 信​息​。​ |
| vcpupin | 控​制​客​户​端​的​虚​拟​ CPU 亲​和​性​。​ |
| domblkstat | 显​示​正​在​运​行​的​客​户​端​的​块​设​备​统​计​。​ |
| domifstat | 显​示​正​在​运​行​的​客​户​端​的​网​络​接​口​统​计​。​ |
| attach-device | 使​用​ XML 文​件​中​的​设​备​定​义​在​客​户​端​中​添​加​设​备​。​ |
| attach-disk | 在​客​户​端​中​附​加​新​磁​盘​设​备​。​ |
| attach-interface | 在​客​户​端​中​附​加​新​网​络​接​口​。​ |
| detach-device | 从​客​户​端​中​分​离​设​备​，使​用​同​样​的​ XML 描​述​作​为​命​令​attach-device。​ |
| detach-disk | 从​客​户​端​中​分​离​磁​盘​设​备​。​ |
| detach-interface | 从​客​户​端​中​分​离​网​络​接​口​。​ |
| 命​令​ | Description |
| version | 显​示​ virsh 版​本​ |
| nodeinfo | 有​关​管​理​程​序​的​输​出​信​息​ |

1.2.2 virsh命令来创建虚拟机步骤

（1）生成硬盘镜像文件

[root@host](mailto:root@host):kvm-img create -f rawtest.img 10G

（2）编写xml配置文件，这一步在1.2.3节具体介绍

（3）创建并运行虚拟机

[root@host](mailto:root@host):virsh create test.xml

其中“test.xml”指定步骤（2）中创建的xml文件

       这样一个虚拟机便创建起来了。

1.2.3 xml配置文件的编写

利用virsh工具创建虚拟机必须编写xml配置文件，该文件指定虚拟机的各项参数，比如虚拟机名称、磁盘镜像的位置、内存大小、显示配置等等。下面给出一个简单的配置文件的例子。

#test.xml

<domain type='qemu'>

  <name>windowsXP</name>

  <uuid></uuid>

  <memory>500000</memory>

  <currentMemory>500000</currentMemory>

  <vcpu>1</vcpu>

  <os>

    <type arch='i686'machine='pc'>hvm</type>

    <boot dev='hd'/>

    <boot dev='cdrom'/>

  </os>

  <devices>

    <emulator>/usr/bin/qemu-system-x86\_64</emulator>

    <disk type='file' device='cdrom'>

      <sourcefile='/home/turnupthesun/kvm/windowsXP.iso'/>

      <target dev='hdc'/>

      <readonly/>

    </disk>

    <disk type='file' device='disk'>

      <sourcefile='/home/turnupthesun/kvm/windowsXP.img'/>

      <target dev='hda'/>

    </disk>

       <graphicstype='vnc' port='14' listen='127.0.0.1'/>

  </devices>

</domain>

下面介绍其中几个比较重要的元素及属性。

（1）<domain>元素的type属性指定运行域的虚拟机管理器，针对kvm应当选择“qemu”。

（2）<name>元素的内容指定域的名字。

（3）<memory>元素和<currentMemory>元素的内容非别指定启动时为域分配的最大内存和实际分配的内存。

（4）<os></os>元素之间的内容用来指定操作系统启动的一些信息。其中重复的<boot>元素形成了一个启动顺序表，比如例子中先从磁盘启动，磁盘无法启动再从光盘启动。

（5）<disk>元素的device属性指明不同的设备，<source>标签的file属性指明这些设备的位置。

1.3 如何通过图形化界面virt-manager来创建虚拟机

   Virt-manger既虚拟机管理器，是创建和管理虚拟客户端的图形工具。具体的操作步骤为：

①   从控制台窗口启动这个工具，从root身份输入virt-manager命令，点击file菜单

的”新建”选项。

②   virt-manager显示两种虚拟化方法：Qemu/KVM或者Xen，这里选择Qemu/KVM作

为hypervisor。

③  选择虚拟机名称和指定一种安装方法，通过网络安装服务器或者本地CD/DVD驱动包括本地ISO文件，在此我用本地ISO的安装方法。

④ 输入本地ISO文件路径和文件名(假设本地ISO的路径就在根目录下，名称为Mini-BT3.6.1.iso)

⑤ 设置虚拟机使用的内存容量和处理器数量。

 ⑥  配置虚拟机的存储方法。对于存储后端有两种选择：物理存储设备或者使用之前建立的磁盘文件。如果处于简单测试，创建文件作为存储后端。当创建虚拟磁盘时，默认为10GB。

 ⑦ 网络配置，在这里选择NAT方式。

这样一个虚拟机就开始启动起来了，将会出现启动界面，最后出现虚拟机中操作系统的界面。

二．libvirt函数库如何实现虚拟机创建

2.1 virsh工具”create”命令源码

在libvirt软件包安装完成之后，就可以看到libvirt的源码，这个源码实现了很多的开发虚拟化软件的用户接口，也就是开发的API。里面也实现了工具virsh，这个工具也实现了很多的功能。在/tools下面有一个virsh.c，这个文件里面实现virsh的功能，这里就具体把创建这部分代码选取出来。

     /\*

 \* "create" command

 \*/

static const vshCmdInfo info\_create[] ={

   {"help", N\_("create a domain from an XML file")},

   {"desc", N\_("Create a domain.")},

   {NULL, NULL}

};

static const vshCmdOptDef opts\_create[]= {

   {"file", VSH\_OT\_DATA, VSH\_OFLAG\_REQ, N\_("file containingan XML domain description")},

#ifndef WIN32

   {"console", VSH\_OT\_BOOL, 0, N\_("attach to console after creation")},

#endif

   {"paused", VSH\_OT\_BOOL, 0, N\_("leave the guest pausedafter creation")},

  {"autodestroy", VSH\_OT\_BOOL, 0, N\_("automatically destroythe guest when virsh disconnects")},

   {NULL, 0, 0, NULL}

};

static bool

cmdCreate(vshControl \*ctl, const vshCmd\*cmd)

{

   virDomainPtr dom;

   const char \*from = NULL;

   bool ret = true;

   char \*buffer;

#ifndef WIN32

   int console = vshCommandOptBool(cmd, "console");

#endif

   unsigned int flags = VIR\_DOMAIN\_NONE;

   if (!vshConnectionUsability(ctl, ctl->conn))

       return false;

   if (vshCommandOptString(cmd, "file", &from) <= 0)

       return false；

   if (virFileReadAll(from, VIRSH\_MAX\_XML\_FILE, &buffer) < 0)

       return false;

   if (vshCommandOptBool(cmd, "paused"))

       flags |= VIR\_DOMAIN\_START\_PAUSED;

   if (vshCommandOptBool(cmd, "autodestroy"))

       flags |= VIR\_DOMAIN\_START\_AUTODESTROY;

   dom = virDomainCreateXML(ctl->conn, buffer, flags);

   VIR\_FREE(buffer);

   if (dom != NULL) {

       vshPrint(ctl, \_("Domain %s created from %s\n"),

                 virDomainGetName(dom), from);

#ifndef WIN32

       if (console)

           cmdRunConsole(ctl, dom,NULL);

#endif

       virDomainFree(dom);

   } else {

       vshError(ctl, \_("Failed to create domain from %s"), from);

       ret = false;

   }

   return ret;

}

代码的讲解：

  ⑴ typedef struct{

       const char \*name;

       const char \*data;

     }vshCmdInfo;

  上面这个结构体是关于命令的键值对信息，命令一般包含两个名称：命令的名字和命令的描述信息。

  ⑵ typedef struct{

        const char \*name;

        vshCmdOptType type;

        unsigned int flags;

        const char \*help;

     }vshCmdOptDef;

  上面这个结构体是关于命令选项的定义，该结构体一般包括四个字段：选项的名称，选项类型，标志，帮助信息。其中选项类型包括：bool类型，字符串类型，整型，字符数据，剩余的参数。

  ⑶ 在cmdCreate主程序中有一个特别重要的函数：virDomainCreateXML()，这个函数的最初原型是: virDomainPtr virDomainCreateXML (virConnectPtr conn,const char\*xmlDesc,unsigned int flags),这个函数是基于一个指定的XML文件来创建一个虚拟机，其中conn表示一个指向hypervisor的连接，xmlDesc表示一个XML文件，flags表示命令选项的标志。

2.2 通过libvirt创建虚拟机的关键API

通过分析2.1中的virsh源码我们可以看出，使用libvirt进行虚拟机创建要调用两个关键的API-- virFileReadAll和virDomainCreateXML，下面分别进行说明。

2.2.1 virFileReadAll

    该函数原型为intvirFileReadAll(const char \*path, int maxlen, char \*\*buf)，功能是将参数“path”指定路径的文件内容读到一个缓冲区中，并将缓冲区地址记录在参数“\*buf”中，而参数“maxlen”指定文件的最大长度。利用该API，我们可以将xml配置文件都到一个缓冲区中，以方便接下来的使用。

2.2.2virDomainCreateXML

该函数原型为[virDomainPtr](http://libvirt.org/html/libvirt-libvirt.html#virDomainPtr)    virDomainCreateXML      ([virConnectPtr](http://libvirt.org/html/libvirt-libvirt.html#virConnectPtr)conn,  const char \* xmlDesc,  unsigned int flags)，功能是根据参数“xmlDesc”定义的配置方式创建一个域并返回该域的指针。参数“conn”是指向虚拟机管理器的指针，而通过设置不同的“flags”标志，可以使创建的域具有不同的属性。

三． 利用libvirt库编写自己的虚拟机创建程序

   Virsh命令用来创建虚拟机的命令是：virsh create,这个命令主要是从给定的XML文件生成客户端并启动客户端。

  下面用一个测试例子来说明如何通过virsh命令来创建虚拟机的。

   具体的操作实践步骤是：

首先需要创建虚拟硬盘，为了放置操作系统的地方，命令是：kvm-img create

701.img10G,也就是创建一个大小为10G的虚拟硬盘。

   2.  编写一个xml文件，这个文件里面包含启动操作系统的一些特征，比如：内存容量，操作系统位置，虚拟硬盘位置等等，其实有很多的字段，可以简写一个xml文件，如果有些字段没有定义，那么系统就会默认，下面给出一个xml文件，命名为701.xml，程序为：

      <domain type='qemu'>

      <name>linux10.0421</name>

      <uuid></uuid>

     <memory>512000</memory>

     <currentMemory>512000</currentMemory>

     <vcpu>1</vcpu>

      <os>

        <type arch='i686' machine='pc'>hvm</type>

        <boot dev='cdrom'/>

        <boot dev='hd'/>

     </os>

        <devices>

           <emulator>/usr/bin/qemu-system-x86\_64</emulator>

           <disk type='file' device='cdrom'>

           <source file='/usr/src/ubuntu-10.04-desktop-i386.iso'/>

        <target dev='hdc'/>

        <readonly/>

      </disk>

        <disk type='file' device='disk'>

         <sourcefile='/var/lib/libvirt/images/701.img'/>

        <target dev='hda'/>

      </disk>

           <graphics type='vnc' port='5901'listen='127.0.0.1'/>

       </devices>

     </domain>

    3.  接着编写一个c文件,名称为701.c这个文件主要实现的功能就是调用这个xml文件来创建并启动虚拟机。这个c程序代码为：

     #include<stdio.h>

     #include<stdlib.h>

     #include<memory.h>

     #include<libvirt/libvirt.h>

     const char \*from=NULL;

     static virConnectPtr conn=NULL;

     #define VIRSH\_MAX\_XML\_FILE 10\*1024\*1024

     void closeConn()

     {

         if(conn!=NULL)

         virConnectClose(conn);

     }

     int cmdCreate()

     {

        virDomainPtr dom;

        char \*buffer;

        unsigned int flags=VIR\_DOMAIN\_NONE;

       conn=virConnectOpen("qemu:///system");

        if(conn==NULL)

        {

           fprintf(stderr,"failed to connect tohypervisor/n");

           closeConn();

           return 0;

        }

       if(virFileReadAll(from,VIRSH\_MAX\_XML\_FILE,&buffer)<0)

         return 0;

       dom=virDomainCreateXML(conn,buffer,flags);

        memset(buffer,0,sizeof(buffer));

        if(dom!=NULL){

           fprintf(stdout,"Domain %screated from %s\n",virDomainGetName(dom),from);

           virDomainFree(dom);

       }

      else{

          fprintf(stdout,"Failed to createdomain from %s",from);

        }

      }

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

       {

         if(argc<2){

         fprintf(stdout,"there are too fewparameters,should has two more parameters!");

       }

      from=\*++argv;

      cmdCreate();

      return 0;

      }

    4. 在命令窗口中先执行gcc -lvirt -o 701 701.c  ,然后执行./701 701.xml,就可以看到这个虚拟机被创建并启动起来了。

四．KVM内核如何实现底层虚拟机创建功能

4.1 KVM虚拟机创建和运行虚拟机的流程

   开源的Lbvirt库实现了很多的虚拟化API，这些API的实现还是要靠底层的KVM内核的实现，下面重点讲讲KVM内核中是如何实现虚拟机创建和运行功能的操作系统层的实现。

    KVM虚拟机创建和运行虚拟机分为用户态和核心态两个部分，用户态主要提供应用程序接口，为虚拟机创建虚拟机上下文环境，在libkvm中提供访问内核字符设备/dev/kvm的接口；内核态为添加到内核中的字符设备/dev/kvm，模块加载进内核后，即可进行接口用户空间调用创建虚拟机。在创建虚拟机过程中，kvm字符设备主要为客户机创建kvm数据结构，创建该虚拟机的虚拟机文件描述符及其相应的数据结构以及创建虚拟机处理器及其相应的数据结构。kvm创建虚拟机的流程如下图：

    根据上图就可以大致知道虚拟机创建和运行的流程了。首先申明一个kvm\_context\_t变量用以描述用户态虚拟机上下文信息，然后调用kvm\_init()函数初始化虚拟机上下文信息；函数kvm\_create()创建虚拟机实例，该函数通过ioctl系统调用创建虚拟机相关的内核数据结构并且返回文件描述符给用户态kvm\_context\_t数据结构；创建完内核虚拟机数据结构后，再创建内核pit以及mmio等外设模拟设备，然后调用kvm\_create\_vcpu()函数来创建虚拟处理器，kvm\_create\_vcpu()函数通过系统调用向由vm\_fd文件描述符指向的虚拟文件调用创建虚拟处理器，并将虚拟处理器的文件描述符返回给用户态程序，供以后的调度使用；创建完虚拟处理器后，由用户态的QEMU程序申请客户机用户空间，用以加载和运行客户机代码；为了使得客户虚拟机正确执行，必须要在内核中为客户机建立正确的内存映射关系，即影子页表信息。因此，申请客户机内存地址空间之后，调用函数kvm\_create\_phys\_mem()创建客户机内存映射关系，该函数主要通过ioctl系统调用向vm\_fd指向队的虚拟文件调用设置内核数据结构中客户机内存映射关系，主要建立影子页表信息；当创建好虚拟处理器和影子页表后，即可读取客户机到指定分配的空间中，然后调度虚拟处理器运行。调度虚拟机的函数为kvm\_run()，该函数通过ioctl系统调用调用由虚拟处理器文件描述符指向的虚拟文件调度处理函数kvm\_run()调度虚拟处理器的执行，该系统调用将虚拟处理器vcpu信息加载到物理处理器中，通过vm\_entry执行进入客户机执行。在客户机正常运行期间kvm\_run()函数不返回，只有发生以下两种情况时，函数返回：1，发生了I/O事件，如客户机发出读写I/O的指令；2，产生了客户机和内核KVM都无法处理的异常。I/O事件处理完毕后，通过重新调用KVM\_RUN()函数继续调度客户机的执行。

4.2 KVM虚拟机创建和运行虚拟机的主要函数分析以及流程

    1.函数kvm\_init():该函数在用户态创建一个虚拟机上下文，用以在用户态保存基本的虚拟机信息，这个函数是创建虚拟机的第一个需要调用的函数，函数返回一个kvm\_context\_t结构体。该函数原型的实现在libkvm.c中，该函数原型是：

kvm\_context\_t kvm\_init(struct kvm\_callbacks\*callbacks,void \*opaque);

参数：callbacks为结构体kvm\_callbacks变量，该结构体包含指向函数的一组指针，用于在客户机执行过程中因为I/O事件退出到用户态的时候处理的回调函数。参数opaque一般未使用。

    函数执行基本过程：打开字符设备dev/kvm，申请虚拟机上下文变量kvm\_context\_t空间，初始化上下文的基本信息：设置fd文件描述符指向/dev/kvm，禁止虚拟机文件描述符vm\_fd(-1),设置I/O事件回调函数结构体，设置IRQ和PIT的标志位以及内存页面记录的标志位。

    用户态数据结构kvm\_context\_t用以描述虚拟机实例的用户态上下文信息。在kvm\_common.h文件里面有kvm\_context的结构体定义。

structkvm\_context {

    /// Filedescriptor to /dev/kvm

    int fd;

    int vm\_fd;

    int vcpu\_fd[MAX\_VCPUS];

    struct kvm\_run \*run[MAX\_VCPUS];

    /// Callbacks that KVM uses to emulatevarious unvirtualizable functionality

    struct kvm\_callbacks \*callbacks;

    void \*opaque;

    /// A pointer to the memory used as thephysical memory for the guest

    void \*physical\_memory;

    /// is dirty pages logging enabled for allregions or not

    int dirty\_pages\_log\_all;

    /// memory regions parameters

    struct kvm\_memory\_regionmem\_regions[KVM\_MAX\_NUM\_MEM\_REGIONS];

    /// do not create in-kernel irqchip if set

    int no\_irqchip\_creation;

    /// in-kernel irqchip status

    int irqchip\_in\_kernel;

};

   各个数据域的解释为：

int fd :指向内核标准字符设备/dev/kvm的文件描述符。

int vm\_fd:指向所创建的内核虚拟机数据结构相关文件的文件描述符。

intvcpu\_fd[MAX\_VCPUS]:指向虚拟机所有的虚拟处理器的文件描述符数组。

struct kvm\_run\*run[MAX\_VCPUS]：指向虚拟机运行环境上下文的指针数组。

struct kvm\_callbacks\*call\_backs: 回调函数结构体指针，该结构体用于处理用户态I/O事件。

void \*opaque:指针(还未弄清楚)

int dirty\_page\_log\_all:设置是否记录脏页面的标志。

int no\_ira\_creation: 用于设置是否再kernel里设置irq芯片。

int\_irqchip\_in\_kernel:内核中irqchip的状态

structkvm\_callbacks：该结构体用于在用户态中处理I/O事件，在KVM中调用KVM\_QEMU实现，主要包含的数据域为：

int (\*inb)(void \*opaque, uint16\_t addr,uint8\_t \*data)：用于模拟客户机执行8位的inb指令。

int (\*inw)(void \*opaque, uint16\_t addr,uint16\_t \*data)：用于模拟客户机执行16位的inw指令。

int (\*inl)(void \*opaque, uint16\_t addr,uint32\_t \*data)：用于模拟客户机执行32位的inl指令。

int (\*outb)(void \*opaque, uint16\_t addr,uint8\_t data)：用于模拟客户机执行8位的outb指令。

int (\*outw)(void \*opaque, uint16\_t addr,uint16\_t data)：用于模拟客户机执行16位的outw指令。

int (\*outl)(void \*opaque, uint16\_t addr,uint32\_t data)：用于模拟客户机执行32位的outl指令。

int (\*mmio\_read)(void \*opaque, uint64\_taddr, uint8\_t \*data,int len)：用于模拟客户机执行mmio读指令。

int (\*mmio\_write)(void \*opaque, uint64\_taddr, uint8\_t \*data,int len)：用于模拟客户机执行mmio写指令。

int (\*debug)(void \*opaque, void \*env,struct kvm\_debug\_exit\_arch \*arch\_info)：用户客户机调试的回调函数。

int (\*halt)(void \*opaque, int vcpu)：用于客户机执行halt指令的响应。

int (\*shutdown)(void \*opaque, void \*env)：用于客户机执行shutdown指令的响应。

int (\*io\_window)(void \*opaque)：用于获得客户机io\_windows。

int (\*try\_push\_interrupts)(void \*opaque)：用于注入中断的回调函数。

void (\*push\_nmi)(void \*opaque)：用于注入nmi中断的函数。

void (\*post\_kvm\_run)(void \*opaque, void\*env);用户得到kvm运行状态函数。

int (\*pre\_kvm\_run)(void \*opaque, void\*env);用于获得kvm之前运行状态的函数

int (\*tpr\_access)(void \*opaque, int vcpu,uint64\_t rip, int is\_write);获得tpr访问处理函数

int (\*powerpc\_dcr\_read)(int vcpu, uint32\_tdcrn, uint32\_t \*data);用于powerpc的dcr读操作

nt (\*powerpc\_dcr\_write)(int vcpu, uint32\_tdcrn, uint32\_t data);用于powerpc的dcr写操作

int (\*s390\_handle\_intercept)(kvm\_context\_tcontext, int vcpu,struct kvm\_run \*run);用于s390的中断处理。

int (\*s390\_handle\_reset)(kvm\_context\_tcontext, int vcpu,struct kvm\_run \*run);用于s390的重设处理。

}

当客户机执行I/O事件或者停机操作等事件时，KVM会交给用户态的QEMU模拟外部I/O事件，调用这个结构体指向的相关的函数进行处理。

Struct kvm\_run: 用于KVM运行时一些的一些状态信息。主要包含的数据域为：

\_\_u8 request\_interrupt\_window;

\_\_u8 padding1[7];

\_\_u32 exit\_reason;

\_\_u8 ready\_for\_interrupt\_injection;

\_\_u8 if\_flag;

\_\_u8 padding2[2];

/\* in (pre\_kvm\_run), out (post\_kvm\_run) \*/

\_\_u64 cr8;

\_\_u64 apic\_base;

union {

/\* KVM\_EXIT\_UNKNOWN \*/

struct {

\_\_u64 hardware\_exit\_reason; 记录退出原因

} hw;

/\* KVM\_EXIT\_FAIL\_ENTRY \*/  客户机执行过程中执行VM\_ENTRY失败。

struct {

       \_\_u64hardware\_entry\_failure\_reason;

} fail\_entry;

/\* KVM\_EXIT\_EXCEPTION \*/  客户机因为异常退出

struct {

       \_\_u32exception;

       \_\_u32error\_code;

} ex;

/\* KVM\_EXIT\_IO \*/   客户机因为IO事件退出。

struct kvm\_io {

#define KVM\_EXIT\_IO\_IN  0

#define KVM\_EXIT\_IO\_OUT 1

\_\_u8 direction;

\_\_u8 size; /\* bytes \*/

\_\_u16 port;

\_\_u32 count;

\_\_u64 data\_offset; /\* relative to kvm\_runstart \*/

} io;

struct {

struct kvm\_debug\_exit\_arch arch;

} debug;

/\* KVM\_EXIT\_MMIO \*/ 客户机因为MMIO退出

struct {

\_\_u64 phys\_addr;

\_\_u8 data[8];

\_\_u32 len;

\_\_u8 is\_write;

} mmio;

/\* KVM\_EXIT\_HYPERCALL \*/ 客户机退出的超调用参数。

struct {

\_\_u64 nr;

\_\_u64 args[6];

\_\_u64 ret;

\_\_u32 longmode;

\_\_u32 pad;

} hypercall;

              /\*KVM\_EXIT\_TPR\_ACCESS \*/ 客户机退出访问TPR参数

struct {

       \_\_u64rip;

       \_\_u32is\_write;

       \_\_u32pad;

} tpr\_access;

/\* KVM\_EXIT\_S390\_SIEIC \*/  和S390相关数据

struct {

\_\_u8 icptcode;

\_\_u64 mask; /\* psw upper half \*/

\_\_u64 addr; /\* psw lower half \*/

\_\_u16 ipa;

\_\_u32 ipb;

} s390\_sieic;

/\* KVM\_EXIT\_S390\_RESET \*/

#define KVM\_S390\_RESET\_POR       1

#define KVM\_S390\_RESET\_CLEAR     2

#define KVM\_S390\_RESET\_SUBSYSTEM 4

#define KVM\_S390\_RESET\_CPU\_INIT  8

#define KVM\_S390\_RESET\_IPL       16

\_\_u64 s390\_reset\_flags;

/\* KVM\_EXIT\_DCR \*/

struct {

       \_\_u32dcrn;

       \_\_u32data;

       \_\_u8  is\_write;

} dcr;

/\* Fix the size of the union. \*/

char padding[256];

2. 函数kvm\_create()：该函数主要用于创建一个虚拟机内核环境。该函数原型为：

int kvm\_create(kvm\_context\_t kvm,unsignedlong phys\_mem\_bytes, void \*\*phys\_mem);

参数：kvm\_context\_t 表示传递的用户态虚拟机上下文环境，phys\_mem\_bytes表示需要创建的物理内存的大小，phys\_mem表示创建虚拟机的首地址。这个函数首先调用kvm\_create\_vm()分配IRQ并且初始化为0,设置vcpu[0]的值为-1，即不允许调度虚拟机执行。然后调用ioctl系统调用ioctl(fd,KVM\_CREATE\_VM,0)来创建虚拟机内核数据结构struct kvm。

3. 系统调用函数ioctl(fd,KVM\_CREATE\_VM,0)，用于在内核中创建和虚拟机相关的数据结构。该函数原型为：

Static long kvm\_dev\_ioctl(struct file \*filp，unsigned intioctl， unsignedlong arg);其中ioctl表示命令。这个函数调用kvm\_dev\_ioctl\_create\_vm()创建虚拟机实例内核相关数据结构。该函数首先通过内核中kvm\_create\_vm()函数创建内核中kvm上下文struct kvm，然后通过函数

Anno\_inode\_getfd(“kvm\_vm”,&kvm\_vm\_fops,kvm,0)返回该虚拟机的文件描述符，返回给用户调用函数，由2中描述的函数赋值给用户态虚拟机上下文变量中的虚拟机描述符kvm\_vm\_fd。

4. 内核创建虚拟机kvm对象后，接着调用kvm\_arch\_create函数用于创建一些体系结构相关的信息，主要包括kvm\_init\_tss、kvm\_create\_pit以及kvm\_init\_coalsced\_mmio等信息。然后调用kvm\_create\_phys\_mem创建物理内存，函数kvm\_create\_irqchip用于创建内核irq信息，通过系统调用ioctl(kvm->vm\_fd,KVM\_CREATE\_IRQCHIP)。

5，函数kvm\_create\_vcpu()：用于创建虚拟处理器。该函数原型为：

int kvm\_create\_vcpu(kvm\_context\_t kvm, intslot)；

参数：kvm表示对应用户态虚拟机上下文，slot表示需要创建的虚拟处理器的个数。

该函数通过ioctl系统调用ioctl(kvm->vm\_fd,KVM\_CREATE\_VCPU，slot)创建属于该虚拟机的虚拟处理器。该系统调用函数：

Static init kvm\_vm\_ioctl\_create\_vcpu(struct\*kvm, n) 参数kvm为内核虚拟机实例数据结构，n为创建的虚拟CPU的数目。

6，函数kvm\_create\_phys\_mem()用于创建虚拟机内存空间，该函数原型：

Void \* kvm\_create\_phys\_mem(kvm\_context\_tkvm,unsigned long phys\_start,unsigned len,int log,int writable);

参数：kvm 表示用户态虚拟机上下文信息，phys\_start为分配给该虚拟机的物理起始地址，len表示内存大小，log表示是否记录脏页面，writable表示该段内存对应的页表是否可写。

该函数首先申请一个结构体kvm\_userspace\_memory\_region 然后通过系统调用KVM\_SET\_USER\_MEMORY\_REGION来设置内核中对应的内存的属性。该系统调用函数原型：

Ioctl(int kvm->vm\_fd，KVM\_SET\_USER\_MEMORY\_REGION，&memory)；

参数：第一个参数vm\_fd为指向内核虚拟机实例对象的文件描述符，第二个参数KVM\_SET\_USER\_MEMORY\_REGION为系统调用命令参数，表示该系统调用为创建内核客户机映射，即影子页表。第三个参数memory表示指向该虚拟机的内存空间地址。系统调用首先通过参数memory通过函数copy\_from\_user从用户空间复制struct\_user\_momory\_region 变量，然后通过kvm\_vm\_ioctl\_set\_memory\_region函数设置内核中对应的内存域。该函数原型：

Int kvm\_vm\_ioctl\_set\_memory\_region(struct\*kvm,struct kvm\_usersapce\_memory\_region \*mem,int user\_alloc);该函数再调用函数kvm\_set\_memory\_resgion()设置影子页表。当这一切都准备完毕后，调用kvm\_run()函数即可调度执行虚拟处理器。

7，函数kvm\_run()：用于调度运行虚拟处理器。该函数原型为：

Int kvm\_run(kvm\_context\_t kvm,int vcpu,void \*env) 该函数首先得到vcpu的描述符，然后调用系统调用ioctl(fd,kvm\_run,0)调度运行虚拟处理器。Kvm\_run函数在正常运行情况下并不返回，除非发生以下事件之一：一是发生了I/O事件，I/O事件由用户态的QEMU处理；一个是发生了客户机和KVM都无法处理的异常事件。KVM\_RUN()中返回截获的事件，主要是I/O以及停机等事件。