Luoye's blog - Up Step by Step

路漫漫其修远兮...

7月 12 2014

KVM Run Process之KVM核心流程

在“KVM Run Process之Qemu核心流程”一文中讲到Qemu通过*KVM_RUN*调用KVM提供的API发起KVM的启动，从这里进入到了内核空间运行，本文主要讲述内核中KVM关于VM运行的核心调用流程,所使用的内核版本为linux3.15。

KVM核心流程

KVM RUN的准备

当Qemu使用kvm_vcpu_ioctl(env, KVM_RUN, 0);发起*KVM_RUN*命令时，ioctl会陷入内核，到达kvm_vcpu_ioctl()；

```
kvm_vcpu_ioctl()     file: virt/kvm/kvm_main.c, line: 1958
    --->kvm_arch_vcpu_ioctl_run()    file: arch/x86/kvm, line: 6305
        --->__vcpu_run()  file: arch/x86/kvm/x86.c, line: 6156
```

在__vcpu_run()中也出现了一个while(){}主循环；

```
1  static int __vcpu_run(struct kvm_vcpu *vcpu)
2  {
3          ......
4          r = 1;
5          while (r > 0) {
6                  if (vcpu->arch.mp_state == KVM_MP_STATE_RUNNABLE && !vcpu->arch.apf.halted)
```

```
7                       r = vcpu_enter_guest(vcpu);
8               else {
9                       ......
10                      }
11              }
12              if (r <= 0)     <--------当r小于0时会跳出循环体
13                      break;
14              ......
15      }
16      return r;
17 }
```

我们看到当KVM通过__vcpu_run()进入主循环后，调用vcpu_enter_guest(),从名字上看可以知道这是进入guest模式的入口；
当r大于0时KVM内核代码会一直调用vcpu_enter_guest()，重复进入guest模式；
当r小于等于0时则会跳出循环体，此时会一步一步退到当初的入口kvm_vcpu_ioctl(),乃至于退回到用户态空间Qemu进程中,具体的地方可以参看上一篇文章，这里也给出相关的代码片段：

```
1  int kvm_cpu_exec(CPUArchState *env)
2  {
3      do {
4          run_ret = kvm_vcpu_ioctl(env, KVM_RUN, 0);
5          switch (run->exit_reason) {     <----------Qemu根据退出的原因进行处理，主要是IO相关方面的操作
6          case KVM_EXIT_IO:
7                  kvm_handle_io();
8                          ......
9          case KVM_EXIT_MMIO:
10                 cpu_physical_memory_rw();
11                         ......
12         case KVM_EXIT_IRQ_WINDOW_OPEN:
13                 ret = EXCP_INTERRUPT;
14             ......
15         case KVM_EXIT_SHUTDOWN:
16                 ret = EXCP_INTERRUPT;
17             ......
18         case KVM_EXIT_UNKNOWN:
19                 ret = -1
20             ......
21         case KVM_EXIT_INTERNAL_ERROR:
22                 ret = kvm_handle_internal_error(env, run);
23             ......
24         default:
```

```
25                 ret = kvm_arch_handle_exit(env, run);
26             ......
27         }
28     } while (ret == 0);
29     env->exit_request = 0;
30     return ret;
31 }
```

Qemu根据退出的原因进行处理，主要是IO相关方面的操作，当然处理完后又会调用kvm_vcpu_ioctl(env, KVM_RUN, 0)再次RUN KMV。我们再次拉回到内核空间，走到了static int vcpu_enter_guest(struct kvm_vcpu *vcpu)函数,其中有几个重要的初始化准备工作：

```
1  static int vcpu_enter_guest(struct kvm_vcpu *vcpu)  file: arch/x86/kvm/x86.c, line: 5944
2  {
3          ......
4          kvm_check_request();     <-------查看是否有guest退出的相关请求
5          ......
6          kvm_mmu_reload(vcpu);    <-------Guest的MMU初始化，为内存虚拟化做准备
7          ......
8          preempt_disable();       <-------内核抢占关闭
9          ......
10         kvm_x86_ops->run(vcpu);  <-------体系架构相关的run操作
11         ......                   <-------到这里表明guest模式已退出
12         kvm_x86_ops->handle_external_intr(vcpu);  <-------host处理外部中断
13         ......
14         preempt_enable();        <-------内核抢占使能
15         ......
16         r = kvm_x86_ops->handle_exit(vcpu);  <------根据具体的退出原因进行处理
17         return r;
18         ......
19 }
```

Guest的进入

kvm_x86_ops是一个x86体系相关的函数集，定义位于file: arch/x86/kvm/vmx.c, line: 8693

```
1 static struct kvm_x86_ops vmx_x86_ops = {
2         ......
3         .run = vmx_vcpu_run,
```

```
4         .handle_exit = vmx_handle_exit,
5         ......
6 }
```

vmx_vcpu_run()中一段核心的汇编函数的功能主要就是从ROOT模式切换至NO ROOT模式，主要进行了：

1. Store host registers：主要将host状态上下文存入到VM对应的VMCS结构中；
2. Load guest registers：主要讲guest状态进行加载；
3. Enter guest mode: 通过ASM_VMX_VMLAUNCH指令进行VM的切换，从此进入另一个世界,即Guest OS中；
4. Save guest registers, load host registers： 当发生VM Exit时，需要保持guest状态，同时加载HOST；

当第4步完成后，Guest即从NO ROOT模式退回到了ROOT模式中，又恢复了HOST的执行生涯。

Guest的退出处理

当然Guest的退出不会就这么算了，退出总是有原因的，为了保证Guest后续的顺利运行，KVM要根据退出原因进行处理，此时重要的函数为：vmx_handle_exit();

```
1  static int vmx_handle_exit(struct kvm_vcpu *vcpu)    file: arch/x86/kvm/vmx.c, line: 6877
2  {
3          ......
4          if (exit_reason < kvm_vmx_max_exit_handlers
5              && kvm_vmx_exit_handlers[exit_reason])
6                  return kvm_vmx_exit_handlers[exit_reason](vcpu);     <-----根据reason调用对应的注册函数处理
7          else {
8                  vcpu->run->exit_reason = KVM_EXIT_UNKNOWN;
9                  vcpu->run->hw.hardware_exit_reason = exit_reason;
10         }
11         return 0;      <--------若发生退出原因不在KVM预定义的handler范围内，则返回0
12 }
```

而众多的exit reason对应的handler如下：

```
1  static int (*const kvm_vmx_exit_handlers[])(struct kvm_vcpu *vcpu) = {
2          [EXIT_REASON_EXCEPTION_NMI]           = handle_exception,     <------异常
3          [EXIT_REASON_EXTERNAL_INTERRUPT]      = handle_external_interrupt,  <------外部中断
4          [EXIT_REASON_TRIPLE_FAULT]            = handle_triple_fault,
```

```
5           [EXIT_REASON_NMI_WINDOW]                  = handle_nmi_window,
6           [EXIT_REASON_IO_INSTRUCTION]          = handle_io,      <------io指令操作
7           [EXIT_REASON_CR_ACCESS]               = handle_cr,
8           [EXIT_REASON_DR_ACCESS]               = handle_dr,
9           [EXIT_REASON_CPUID]                   = handle_cpuid,
10          [EXIT_REASON_MSR_READ]                = handle_rdmsr,
11          [EXIT_REASON_MSR_WRITE]               = handle_wrmsr,
12          [EXIT_REASON_PENDING_INTERRUPT]       = handle_interrupt_window,
13          [EXIT_REASON_HLT]                     = handle_halt,
14          [EXIT_REASON_INVD]                          = handle_invd,
15          [EXIT_REASON_INVLPG]                    = handle_invlpg,
16          [EXIT_REASON_RDPMC]                   = handle_rdpmc,
17          [EXIT_REASON_VMCALL]                  = handle_vmcall,     <-----VM相关操作指令
18          [EXIT_REASON_VMCLEAR]                 = handle_vmclear,
19          [EXIT_REASON_VMLAUNCH]                = handle_vmlaunch,
20          [EXIT_REASON_VMPTRLD]                 = handle_vmptrld,
21          [EXIT_REASON_VMPTRST]                 = handle_vmptrst,
22          [EXIT_REASON_VMREAD]                  = handle_vmread,
23          [EXIT_REASON_VMRESUME]                = handle_vmresume,
24          [EXIT_REASON_VMWRITE]                 = handle_vmwrite,
25          [EXIT_REASON_VMOFF]                   = handle_vmoff,
26          [EXIT_REASON_VMON]                    = handle_vmon,
27          [EXIT_REASON_TPR_BELOW_THRESHOLD]     = handle_tpr_below_threshold,
28          [EXIT_REASON_APIC_ACCESS]             = handle_apic_access,
29          [EXIT_REASON_APIC_WRITE]              = handle_apic_write,
30          [EXIT_REASON_EOI_INDUCED]             = handle_apic_eoi_induced,
31          [EXIT_REASON_WBINVD]                  = handle_wbinvd,
32          [EXIT_REASON_XSETBV]                  = handle_xsetbv,
33          [EXIT_REASON_TASK_SWITCH]             = handle_task_switch,     <----进程切换
34          [EXIT_REASON_MCE_DURING_VMENTRY]      = handle_machine_check,
35          [EXIT_REASON_EPT_VIOLATION]               = handle_ept_violation,   <----EPT缺页异常
36          [EXIT_REASON_EPT_MISCONFIG]           = handle_ept_misconfig,
37          [EXIT_REASON_PAUSE_INSTRUCTION]       = handle_pause,
38          [EXIT_REASON_MWAIT_INSTRUCTION]       = handle_invalid_op,
39          [EXIT_REASON_MONITOR_INSTRUCTION]     = handle_invalid_op,
40          [EXIT_REASON_INVEPT]                  = handle_invept,
41  };
```

当该众多的handler处理成功后，会得到一个大于0的返回值，而处理失败则会返回一个小于0的数；则又回到__vcpu_run()中的主循环中；
vcpu_enter_guest() > 0时： 则继续循环，再次准备进入Guest模式；
vcpu_enter_guest() <= 0时： 则跳出循环，返回用户态空间，由Qemu根据退出原因进行处理。

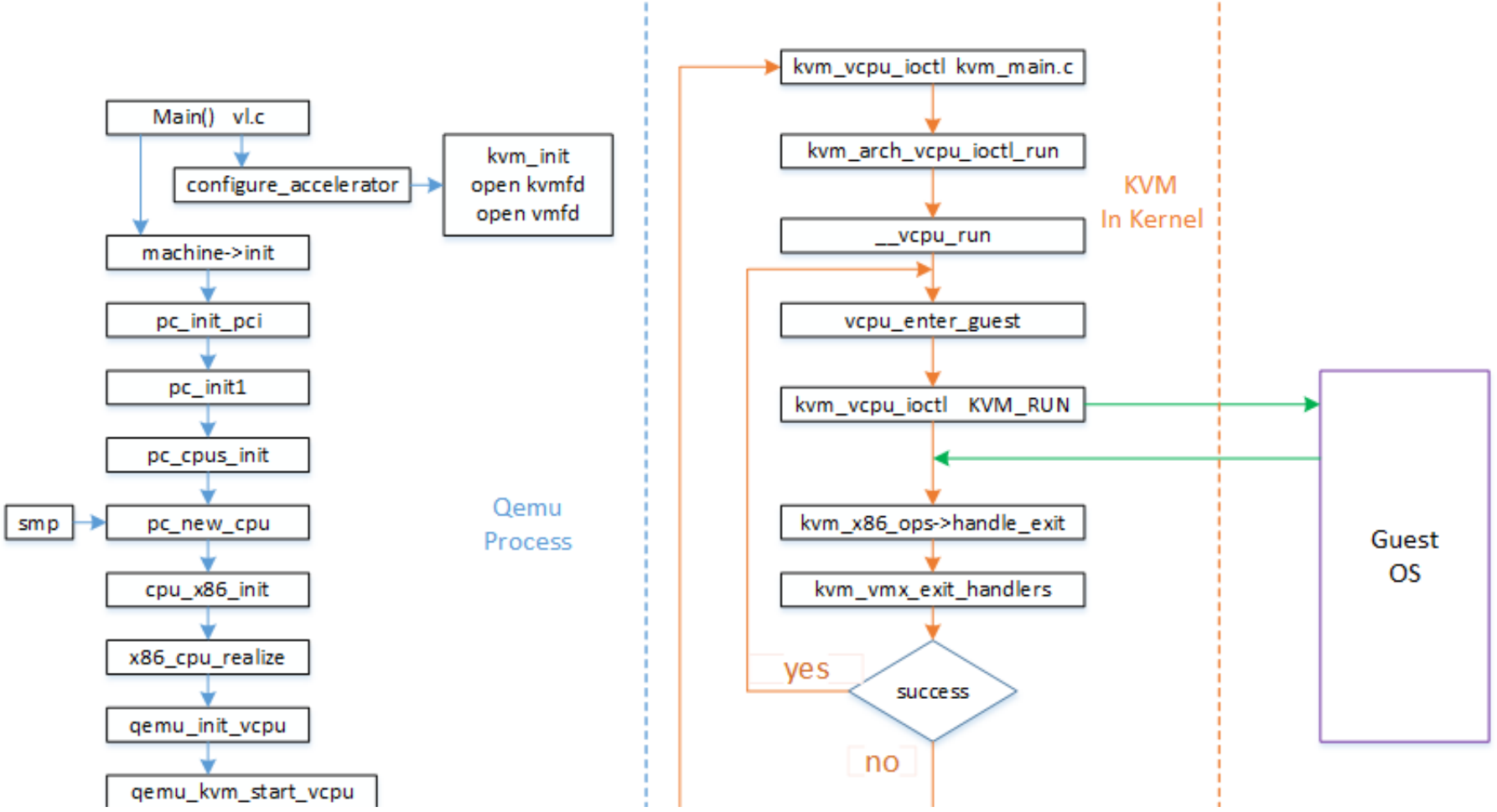
Conclusion

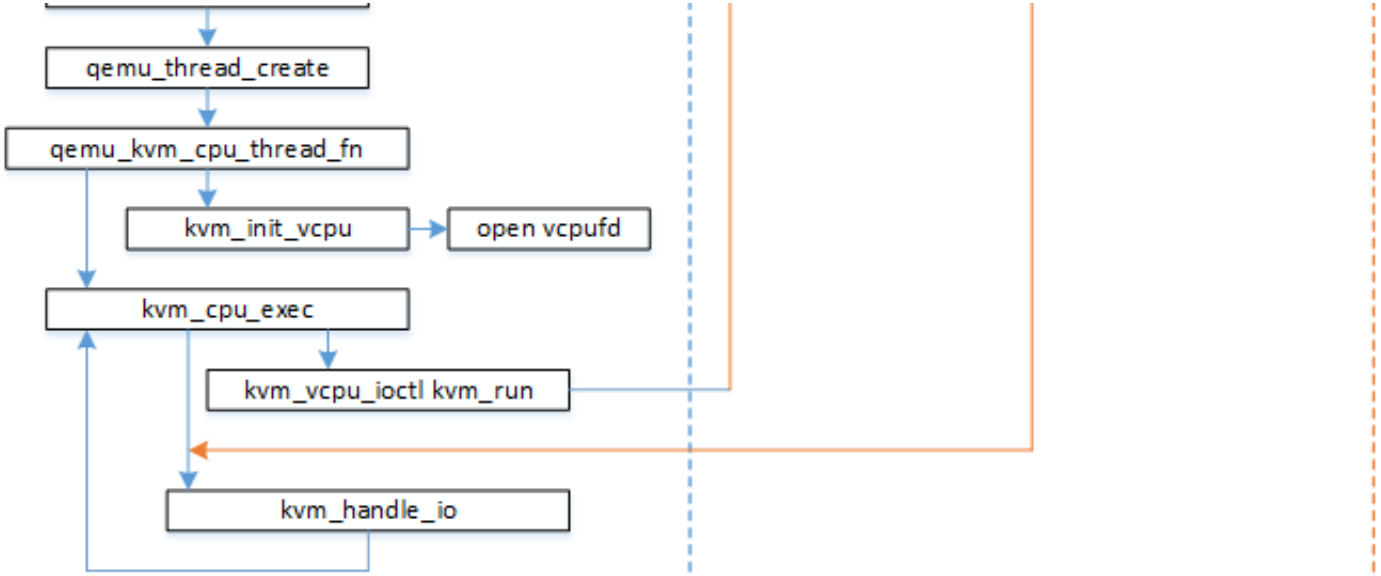
至此，KVM内核代码部分的核心调用流程的分析到此结束，从上述流程中可以看出，KVM内核代码的主要工作如下：

1. Guest进入前的准备工作；
2. Guest的进入；
3. 根据Guest的退出原因进行处理，若kvm自身能够处理的则自行处理；若KVM无法处理，则返回到用户态空间的Qemu进程中进行处理；

总而言之，KVM与Qemu的工作是为了确保Guest的正常运行，通过各种异常的处理，使Guest无需感知其运行的虚拟环境。

附图：





KVM
Virtualization KVM QEMU
上一页 下一页

分类

- KVM18

标签云

EPTVirtualization KVM QEMUvhostvirtio