

昵称： Jessica程序猿
园龄： 5年10个月
粉丝： 544
关注： 27
+加关注

< 2020年4月 >						
日	一	二	三	四	五	六
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	1	2
3	4	5	6	7	8	9

搜索

找找看

谷歌搜索

- 常用链接
- 我的随笔

我的评论

我的参与

最新评论

我的标签

- 随笔分类 (988)
- C++(79)

C++ template(5)

C++ 容器(28)

C++构造函数(6)

C++面向对象编程(7)

C++训练(71)

C++重载与类型转换(9)

careercup(87)

dubbo(2)

Effective C++(3)

ext2文件系统系列(6)

flashcache(2)

KVM虚拟机IO处理过程(二) ----QEMU/KVM I/O 处理过程

接着KVM虚拟机IO处理过程中Guest Vm IO处理过程
(<http://blog.csdn.net/dashulu/article/details/16820281>),本篇文章主要描述IO从guest vm跳转到kvm和qemu后的处理过程.

首先回顾一下kvm的启动过程
(<http://blog.csdn.net/dashulu/article/details/17074675>).qemu通过调用kvm提供的一系列接口来启动kvm. qemu的入口为vl.c中的main函数,main函数通过调用kvm_init 和 machine->init来初始化kvm. 其中, machine->init会创建vcpu, 用一个线程去模拟vcpu, 该线程执行的函数为qemu_kvm_cpu_thread_fn, 并且该线程最终kvm_cpu_exec,该函数调用kvm_vcpu_ioctl切换到kvm中,下次从kvm中返回时,会接着执行kvm_vcpu_ioctl之后的代码,判断exit_reason,然后进行相应处理.

[cpp] view plain copy 在CODE上查看代码片 派生到我的代码片

```
1. int kvm_cpu_exec(CPUState *cpu)
2. {
3.     struct kvm_run *run = cpu->kvm_run;
4.     int ret, run_ret;
5.
6.     DPRINTF("kvm_cpu_exec()\n");
7.
8.     if (kvm_arch_process_async_events(cpu)) {
9.         cpu->exit_request = 0;
10.        return EXCP_HLT;
11.    }
12.
13.    do {
14.        if (cpu->kvm_vcpu_dirty) {
15.            kvm_arch_put_registers(cpu, KVM_PUT_RUNTIME_STATE);
16.            cpu->kvm_vcpu_dirty = false;
17.        }
18.
19.        kvm_arch_pre_run(cpu, run);
20.        if (cpu->exit_request) {
21.            DPRINTF("interrupt exit requested\n");
22.            /*
23.             * KVM requires us to reenter the kernel after IO exits to complete
24.             * instruction emulation. This self-signal will ensure that we
25.             * leave ASAP again.
26.             */
27.            qemu_cpu_kick_self();
28.        }
29.        qemu_mutex_unlock_iothread();
30.
31.        run_ret = kvm_vcpu_ioctl(cpu, KVM_RUN, 0);
32.
33.        qemu_mutex_lock_iothread();
34.        kvm_arch_post_run(cpu, run);
35.
36.        if (run_ret < 0) {
37.            if (run_ret == -EINTR || run_ret == -EAGAIN) {
38.                DPRINTF("io window exit\n");
39.                ret = EXCP_INTERRUPT;
40.                break;
41.            }
42.            fprintf(stderr, "error: kvm run failed %s\n",
43.                    strerror(-run_ret));
44.            abort();
45.        }
46.
47.        trace_kvm_run_exit(cpu->cpu_index, run->exit_reason);
```

gdb调试(2)
git(3)
hbase(1)
iscsi(4)
JAVA(16)
JVM虚拟机(1)
Leetcode(169)
linux内存管理(11)
linux内核(23)
Linux内核分析及编程(9)
Linux内核设计与实现(1)
maven
nginx(2)
python(1)
shell 编程(20)
Spring(2)
SQL(5)
STL源码剖析(11)
UNIX 网络编程(25)
unix环境高级编程(24)
Vim(2)
web(10)
阿里中间件(5)
操作系统
测试(13)
程序员的自我修养(7)
大数据处理(3)
动态内存和智能指针(4)
泛型编程(2)
分布式(1)
概率题(2)
论文中的算法(11)
面试(82)

```
48.     switch (run->exit_reason) {
49.     case KVM_EXIT_IO:
50.         DPRINTF("handle_io\n");
51.         kvm_handle_io(run->io.port,
52.                        (uint8_t *)run + run->io.data_offset,
53.                        run->io.direction,
54.                        run->io.size,
55.                        run->io.count);
56.
57.         ret = 0;
58.         break;
59.     case KVM_EXIT_MMIO:
60.         DPRINTF("handle_mmio\n");
61.         cpu_physical_memory_rw(run->mmio.phys_addr,
62.                                run->mmio.data,
63.                                run->mmio.len,
64.                                run->mmio.is_write);
65.
66.         ret = 0;
67.         break;
68.     case KVM_EXIT_IRQ_WINDOW_OPEN:
69.         DPRINTF("irq_window_open\n");
70.         ret = EXCP_INTERRUPT;
71.         break;
72.     case KVM_EXIT_SHUTDOWN:
73.         DPRINTF("shutdown\n");
74.         qemu_system_reset_request();
75.         ret = EXCP_INTERRUPT;
76.         break;
77.     case KVM_EXIT_UNKNOWN:
78.         fprintf(stderr, "KVM: unknown exit, hardware reason %" PRIx64 "\n",
79.                (uint64_t)run->hw.hardware_exit_reason);
80.
81.         ret = -1;
82.         break;
83.     case KVM_EXIT_INTERNAL_ERROR:
84.         ret = kvm_handle_internal_error(cpu, run);
85.         break;
86.     default:
87.         DPRINTF("kvm_arch_handle_exit\n");
88.         ret = kvm_arch_handle_exit(cpu, run);
89.         break;
90.     }
91. } while (ret == 0);
92.
93. if (ret < 0) {
94.     cpu_dump_state(cpu, stderr, fprintf, CPU_DUMP_CODE);
95.     vm_stop(RUN_STATE_INTERNAL_ERROR);
96. }
97.
98. cpu->exit_request = 0;
99. return ret;
100. }
```

kvm_vcpu_ioctl执行时,调用的kvm函数是virt/kvm/kvm-main.c中的kvm_vcpu_ioctl.c函数.当传入参数为KVM_RUN时,最终会调用到vcpu_enter_guest函数, vcpu_enter_guest函数中调用了kvm_x86_ops->run(vcpu),在intel处理器架构中该函数对应的实现为vmx_vcpu_run, vmx_vcpu_run设置好寄存器状态之后调用VM_LAUNCH或者VM_RESUME进入guest vm, 一旦发生vm exit则从此处继续执行下去.

[cpp] [view plain](#) [copy](#)  在CODE上查看代码片  派生到我的代码片

```
1. static void __noclone vmx_vcpu_run(struct kvm_vcpu *vcpu)
2. {
3.     struct vcpu_vmx *vmx = to_vmx(vcpu);
4.     unsigned long debugctlmsr;
5.
6.     /*...此处省略n行代码...*/
7.     vmx->__launched = vmx->loaded_vmcs->launched;
8.     asm(
9.         /* Store host registers */
10.        "push %%_ASM_DX "; push %%_ASM_BP ";
11.        "push %%_ASM_CX " \n\t" /* placeholder for guest rcx */
12.        "push %%_ASM_CX " \n\t"
13.        "cmp %%_ASM_SP ", %c[host_rsp](%0) \n\t"
14.        "je 1f \n\t"
```

软件安装(31)
设计模式(12)
深度探索C++对象模型(17)
深入理解Linux内核(1)
数据结构与算法(60)
搜索引擎(2)
算法 每日一练(7)
算法导论(27)
文件系统(20)
虚拟化(11)
杂项(23)

随笔档案 (974)

2020年1月(1)
2019年2月(2)
2019年1月(1)
2018年8月(1)
2018年7月(1)
2018年6月(6)
2018年5月(7)
2018年4月(2)
2018年2月(3)
2017年12月(3)
2017年11月(1)
2017年9月(2)
2017年5月(2)
2017年4月(2)
2017年3月(3)
2017年2月(3)
2016年5月(4)
2016年4月(12)
2016年3月(3)
2016年2月(2)
2016年1月(6)

```
15.     "mov %% _ASM_SP ", %[host_rsp](%0) \n\t"
16.     __ex(ASM_VMX_VMWWRITE_RSP_RDX) "\n\t"
17.     "1: \n\t"
18.     /* Reload cr2 if changed */
19.     "mov %[cr2](%0), %% _ASM_AX " \n\t"
20.     "mov %%cr2, %% _ASM_DX " \n\t"
21.     "cmp %% _ASM_AX ", %% _ASM_DX " \n\t"
22.     "je 2f \n\t"
23.     "mov %% _ASM_AX", %%cr2 \n\t"
24.     "2: \n\t"
25.     /* Check if vmlaunch of vmresume is needed */
26.     "cmpl $0, %[launched](%0) \n\t"
27.     /* Load guest registers. Don't clobber flags. */
28.     "mov %[rax](%0), %% _ASM_AX " \n\t"
29.     "mov %[rbx](%0), %% _ASM_BX " \n\t"
30.     "mov %[rdx](%0), %% _ASM_DX " \n\t"
31.     "mov %[rsi](%0), %% _ASM_SI " \n\t"
32.     "mov %[rdi](%0), %% _ASM_DI " \n\t"
33.     "mov %[rbp](%0), %% _ASM_BP " \n\t"
34. #ifdef CONFIG_X86_64
35.     "mov %[r8](%0), %%r8 \n\t"
36.     "mov %[r9](%0), %%r9 \n\t"
37.     "mov %[r10](%0), %%r10 \n\t"
38.     "mov %[r11](%0), %%r11 \n\t"
39.     "mov %[r12](%0), %%r12 \n\t"
40.     "mov %[r13](%0), %%r13 \n\t"
41.     "mov %[r14](%0), %%r14 \n\t"
42.     "mov %[r15](%0), %%r15 \n\t"
43. #endif
44.     "mov %[rcx](%0), %% _ASM_CX " \n\t" /* kills %0 (ecx) */
45.
46.     /* Enter guest mode */
47.     "jne 1f \n\t"
48.     __ex(ASM_VMX_VMLAUNCH) "\n\t"
49.     "jmp 2f \n\t"
50.     "1: " __ex(ASM_VMX_VMRESUME) "\n\t"
51.     "2: "
52.     /* Save guest registers, load host registers, keep flags */
53.     "mov %0, %[wordsize](%% _ASM_SP ") \n\t"
54.     "pop %0 \n\t"
55.     "mov %% _ASM_AX ", %[rax](%0) \n\t"
56.     "mov %% _ASM_BX ", %[rbx](%0) \n\t"
57.     __ASM_SIZE(pop) " %[rcx](%0) \n\t"
58.     "mov %% _ASM_DX ", %[rdx](%0) \n\t"
59.     "mov %% _ASM_SI ", %[rsi](%0) \n\t"
60.     "mov %% _ASM_DI ", %[rdi](%0) \n\t"
61.     "mov %% _ASM_BP ", %[rbp](%0) \n\t"
62. #ifdef CONFIG_X86_64
63.     "mov %%r8, %[r8](%0) \n\t"
64.     "mov %%r9, %[r9](%0) \n\t"
65.     "mov %%r10, %[r10](%0) \n\t"
66.     "mov %%r11, %[r11](%0) \n\t"
67.     "mov %%r12, %[r12](%0) \n\t"
68.     "mov %%r13, %[r13](%0) \n\t"
69.     "mov %%r14, %[r14](%0) \n\t"
70.     "mov %%r15, %[r15](%0) \n\t"
71. #endif
72.     "mov %%cr2, %% _ASM_AX " \n\t"
73.     "mov %% _ASM_AX ", %[cr2](%0) \n\t"
74.
75.     "pop %% _ASM_BP "; pop %% _ASM_DX " \n\t"
76.     "setbe %[fail](%0) \n\t"
77.     ".pushsection .rodata \n\t"
78.     ".global vmx_return \n\t"
79.     "vmx_return: " _ASM_PTR " 2b \n\t"
80.     ".popsection"
81.     : : "c"(vmx), "d"((unsigned long)HOST_RSP),
82.     [launched]"i"(offsetof(struct vcpu_vmx, __launched)),
83.     [fail]"i"(offsetof(struct vcpu_vmx, fail)),
84.     [host_rsp]"i"(offsetof(struct vcpu_vmx, host_rsp)),
85.     [rax]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_RAX])),
86.     [rbx]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_RBX])),
87.     [rcx]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_RCX])),
88.     [rdx]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_RDX])),
89.     [rsi]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_RSI])),
```

2015年11月(3)
2015年10月(7)
2015年9月(15)
2015年8月(10)
2015年7月(11)
2015年6月(2)
2015年5月(38)
2015年4月(60)
2015年3月(71)
2015年2月(3)
2015年1月(3)
2014年12月(119)
2014年11月(180)
2014年10月(69)
2014年9月(18)
2014年8月(109)
2014年7月(67)
2014年6月(85)
2014年5月(37)

文章分类 (0)

metaq
netty
UML

linux

阮一峰的网络日志
淘宝内核组
阿里核心系统团队博客

算法牛人

v_JULY_v
分布式文件系统测试
acm之家
Linux开发专注者



```
90.         [rdi]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_RDI])),
91.         [rbp]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_RBP])),
92. #ifdef CONFIG_X86_64
93.         [r8]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_R8])),
94.         [r9]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_R9])),
95.         [r10]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_R10])),
96.         [r11]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_R11])),
97.         [r12]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_R12])),
98.         [r13]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_R13])),
99.         [r14]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_R14])),
100.        [r15]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_R15])),
101. #endif
102.        [cr2]"i"(offsetof(struct vcpu_vmx, vcpu.arch.cr2)),
103.        [wordsize]"i"(sizeof(ulong))
104.        : "cc", "memory"
105. #ifdef CONFIG_X86_64
106.        , "rax", "rbx", "rdi", "rsi"
107.        , "r8", "r9", "r10", "r11", "r12", "r13", "r14", "r15"
108. #else
109.        , "eax", "ebx", "edi", "esi"
110. #endif
111.        );
112.
113.        /* MSR_IA32_DEBUGCTLMSR is zeroed on vmexit. Restore it if needed */
114.        if (debugctlmsr)
115.            update_debugctlmsr(debugctlmsr);
116.
117. #ifndef CONFIG_X86_64
118.        /*
119.         * The sysexit path does not restore ds/es, so we must set them to
120.         * a reasonable value ourselves.
121.         *
122.         * We can't defer this to vmx_load_host_state() since that function
123.         * may be executed in interrupt context, which saves and restore segments
124.         * around it, nullifying its effect.
125.         */
126.        loadsegment(ds, __USER_DS);
127.        loadsegment(es, __USER_DS);
128. #endif
129.
130.        vcpu->arch.regs_avail = ~( (1 << VCPU_REGS_RIP) | (1 << VCPU_REGS_RSP)
131.                                   | (1 << VCPU_EXREG_RFLAGS)
132.                                   | (1 << VCPU_EXREG_CPL)
133.                                   | (1 << VCPU_EXREG_PDPTR)
134.                                   | (1 << VCPU_EXREG_SEGMENTS)
135.                                   | (1 << VCPU_EXREG_CR3));
136.        vcpu->arch.regs_dirty = 0;
137.
138.        vmx->idt_vectoring_info = vmcs_read32(IDT_VECTORING_INFO_FIELD);
139.
140.        if (is_guest_mode(vcpu)) {
141.            struct vmcs12 *vmcs12 = get_vmcs12(vcpu);
142.            vmcs12->idt_vectoring_info_field = vmx->idt_vectoring_info;
143.            if (vmx->idt_vectoring_info & VECTORING_INFO_VALID_MASK) {
144.                vmcs12->idt_vectoring_error_code =
145.                    vmcs_read32(IDT_VECTORING_ERROR_CODE);
146.                vmcs12->vm_exit_instruction_len =
147.                    vmcs_read32(VM_EXIT_INSTRUCTION_LEN);
148.            }
149.        }
150.
151.        vmx->loaded_vmcs->launched = 1;
152.
153.        vmx->exit_reason = vmcs_read32(VM_EXIT_REASON);
154.        trace_kvm_exit(vmx->exit_reason, vcpu, KVM_ISA_VMX);
155.
156.        vmx_complete_atomic_exit(vmx);
157.        vmx_recover_nmi_blocking(vmx);
158.        vmx_complete_interrupts(vmx);
159.    }
```

酷壳
C++11 中值得关注的几大变化 (详解)
iTech's Blog
Not Only Algorithm，不仅仅是算法，关注数学、算法、数据结构、程序员笔试面试以及一切涉及计算机编程之美的内容。

最新评论
1. Re:编写一个程序，从标准输入中读取若干string对象并查找连续重复出现的单词。所谓连续重复出现的意思是：一个单词后面紧跟着这个单词本身。要求记录连续重复出现的最大次数以及对应的单词
输入how cow，两个不同的单词，统计会出现问题。结果应该是没有连续出现的单词。
--想撸串的红杉树
2. Re:linux内核内存管理(zone_dma zone_normal zone_highmem)
你好，我想咨询您一个问题，32位系统中，用户进程可以访问物理内存的大小为3G，那么这3G有具体的空间范围吗？是物理内存的0-3G还是什么，因为我看0-896M是被内核线性映射，所以肯定表示0-3G，所...
--CV学习者
3. Re:spring中bean配置和bean注入
tql
--那么小晨呐
4. Re:如何使用jstack分析线程状态
thanks
--andyFly2016
5. Re:迪杰斯特拉算法介绍
不过有图挺好的，但我看到那里看不懂了。
--何所倚



阅读排行榜
1. spring中bean配置和bean注入(138538)

介绍完初始化的流程,可以介绍IO在kvm和qemu中的处理流程了. 当Guest Vm进行IO操作需要访问设备时,就会触发vm exit 返回到vmx_vcpu_run, vmx保存好vmcs并且记录下VM_EXIT_REASON后返回到调用该函数的vcpu_enter_guest, 在vcpu_enter_guest函数末尾调用了r = kvm_x86_ops->handle_exit(vcpu), 该函数对应于vmx_handle_exit函数(intel cpu架构对应关系可以查看vmx.c文件中static struct kvm_x86_ops vmx_x86_ops), vmx_handle_exit调用kvm_vmx_exit_handlers[exit_reason](vcpu),该语句根据exit_reason调用不同的函数,该数据结构定义如下:

[cpp] [view plain](#) [copy](#) 在CODE上查看代码片 派生到我的代码片

```
1. static int (*const kvm_vmx_exit_handlers[])(struct kvm_vcpu *vcpu) = {
2.     [EXIT_REASON_EXCEPTION_NMI]           = handle_exception,
3.     [EXIT_REASON_EXTERNAL_INTERRUPT]       = handle_external_interrupt,
4.     [EXIT_REASON_TRIPLE_FAULT]            = handle_triple_fault,
5.     [EXIT_REASON_NMI_WINDOW]              = handle_nmi_window,
6.     [EXIT_REASON_IO_INSTRUCTION]          = handle_io,
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_rdmr,
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,
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_VMWWRITE]                 = handle_vmwwrite,
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_WBINVD]                   = handle_wbinvd,
30.    [EXIT_REASON_XSETBV]                   = handle_xsetbv,
31.    [EXIT_REASON_TASK_SWITCH]              = handle_task_switch,
32.    [EXIT_REASON_MCE_DURING_VMENTRY]      = handle_machine_check,
33.    [EXIT_REASON_EPT_VIOLATION]            = handle_ept_violation,
34.    [EXIT_REASON_EPT_MISCONFIG]            = handle_ept_misconfig,
35.    [EXIT_REASON_PAUSE_INSTRUCTION]        = handle_pause,
36.    [EXIT_REASON_MWAIT_INSTRUCTION]        = handle_invalid_op,
37.    [EXIT_REASON_MONITOR_INSTRUCTION]      = handle_invalid_op,
38.    };
```

如果是因为IO原因导致的vm_exit,则调用的处理函数为handle_io,handle_io的处理可以查看(<http://blog.csdn.net/fanwenyi/article/details/12748613>), 该过程结束之后需要qemu去处理IO,这时候会返回到qemu, 在kvm_cpu_exec中继续执行下去,看上面kvm_cpu_exec的代码,如果是因为IO原因返回到qemu,会调用kvm_handle_io函数.

[cpp] [view plain](#) [copy](#) 在CODE上查看代码片 派生到我的代码片

```
1. switch (run->exit_reason) {
2.     case KVM_EXIT_IO:
3.         DPRINTF("handle_io\n");
4.         kvm_handle_io(run->io.port,
5.                       (uint8_t *)run + run->io.data_offset,
6.                       run->io.direction,
7.                       run->io.size,
8.                       run->io.count);
9.         ret = 0;
10.        break;
```

kvm_handle_io调用cpu_outb, cpu_outw等指令处理IO操作.

假设虚拟机是用raw格式的磁盘,则IO在qemu中处理时经过的函数栈如下所示:

[cpp] [view plain](#) [copy](#) 在CODE上查看代码片 派生到我的代码片

- 2. 如何使用jstack分析线程状态(89973)
- 3. maven快照版本和发布版本(29243)
- 4. C++ stringstream介绍, 使用方法与例子(28518)
- 5. Linux用户空间与内核空间（理解高端内存）(27348)

评论排行榜

- 1. KVM虚拟机IO处理过程(一) ----Guest VM I/O 处理过程(9)
- 2. careercup-递归和动态规划 9.10(7)
- 3. 如何使用jstack分析线程状态(6)
- 4. spring中bean配置和bean注入(6)
- 5. 蜻蜓fm面试(6)

推荐排行榜

- 1. 如何使用jstack分析线程状态(20)
- 2. spring中bean配置和bean注入(19)
- 3. 数据库索引原理及优化(12)
- 4. maven快照版本和发布版本(8)
- 5. linux下的僵尸进程处理SIGCHLD信号(8)

```
1. #0 bdrv_aio_writev (bs=0x55555629e9b0, sector_num=870456,
2. qiov=0x555556715ab0, nb_sectors=1,
3. cb=0x55555670161b <ide_sector_write_cb>, opaque=0x5555567157b8)
4. at block.c:3408
5. #1 0x0000555556701960 in ide_sector_write (s=0x5555567157b8)
6. at hw/ide/core.c:798
7. #2 0x00005555567047ae in ide_data_writev (opaque=0x555556715740, addr=496,
8. val=8995) at hw/ide/core.c:1907
9. #3 0x0000555556709e4c in portio_write (opaque=0x5555565c0670, addr=0,
10. data=8995, size=2) at /home/dashu/kvm/qemu/qemu-dev-zwu/ioport.c:174
11. #4 0x000055555670e13d5 in memory_region_write_accessor (mr=0x5555565c0670,
12. addr=0, value=0x7fffb4dbd528, size=2, shift=0, mask=65535)
13. at /home/dashu/kvm/qemu/qemu-dev-zwu/memory.c:440
14. #5 0x000055555670e151d in access_with_adjusted_size (addr=0,
15. value=0x7fffb4dbd528, size=2, access_size_min=1, access_size_max=4,
16. access=0x55555670e1341 <memory_region_write_accessor>, mr=0x5555565c0670)
17. at /home/dashu/kvm/qemu/qemu-dev-zwu/memory.c:477
18. #6 0x000055555670e3dfb in memory_region_dispatch_write (mr=0x5555565c0670,
19. addr=0, data=8995, size=2)
20. at /home/dashu/kvm/qemu/qemu-dev-zwu/memory.c:984
21. #7 0x0000555556707384 in io_mem_write (mr=0x5555565c0670, addr=0, val=8995,
22. size=2) at /home/dashu/kvm/qemu/qemu-dev-zwu/memory.c:1748
23. #8 0x0000555556708a18e in address_space_rw (as=0x555556216d80, addr=496,
24. buf=0x7fffb4dbd670 "##", len=2, is_write=true)
25. at /home/dashu/kvm/qemu/qemu-dev-zwu/exec.c:1968
26. #9 0x0000555556708a474 in address_space_write (as=0x555556216d80, addr=496,
27. buf=0x7fffb4dbd670 "##", len=2)
28. at /home/dashu/kvm/qemu/qemu-dev-zwu/exec.c:2030
29. #10 0x00005555567098c9 in cpu_outw (addr=496, val=8995)
30. at /home/dashu/kvm/qemu/qemu-dev-zwu/ioport.c:61
```

bdrv_aio_writev最终调用bdrv_co_aio_rw_vector函数, 该函数调用co = qemu_coroutine_create(bdrv_co_do_rw) 创建一个协程去执行bdrv_co_do_rw函数,bdrv_co_wd_rw函数的函数栈如下:

[cpp] [view plain](#) [copy](#) 在CODE上查看代码片 派生到我的代码片

```
1. #1 0x0000555556706353c in paio_submit (bs=0x5555562a13d0, fd=10, sector_num=2,
2.
3. qiov=0x555556715ab0, nb_sectors=1,
4. cb=0x555556028b1 <bdrv_co_io_em_complete>, opaque=0x555556964e30, type=1)
5. at block/raw-posix.c:825
6. #2 0x000055555670636659 in raw_aio_submit (bs=0x5555562a13d0, sector_num=2,
7. qiov=0x555556715ab0, nb_sectors=1,
8. cb=0x555556028b1 <bdrv_co_io_em_complete>, opaque=0x555556964e30, type=1)
9. at block/raw-posix.c:853
10. #3 0x0000555556706366c9 in raw_aio_readv (bs=0x5555562a13d0, sector_num=2,
11. qiov=0x555556715ab0, nb_sectors=1,
12. cb=0x555556028b1 <bdrv_co_io_em_complete>, opaque=0x555556964e30)
13. at block/raw-posix.c:861
14. #4 0x00005555567029b8 in bdrv_co_io_em (bs=0x5555562a13d0, sector_num=2,
15. nb_sectors=1, iov=0x555556715ab0, is_write=false) at block.c:4038
16. #5 0x0000555556702a49 in bdrv_co_readv_em (bs=0x5555562a13d0, sector_num=2,
17. nb_sectors=1, iov=0x555556715ab0) at block.c:4055
18. #6 0x000055555670fed61 in bdrv_co_do_readv (bs=0x5555562a13d0, sector_num=2,
19. nb_sectors=1, qiov=0x555556715ab0, flags=0) at block.c:2547
20. #7 0x000055555670fee03 in bdrv_co_readv (bs=0x5555562a13d0, sector_num=2,
21. nb_sectors=1, qiov=0x555556715ab0) at block.c:2573
22. #8 0x0000555556707d8c in raw_co_readv (bs=0x55555629e9b0, sector_num=2,
23. nb_sectors=1, qiov=0x555556715ab0) at block/raw.c:47
24. #9 0x000055555670fed61 in bdrv_co_do_readv (bs=0x55555629e9b0, sector_num=2,
25. nb_sectors=1, qiov=0x555556715ab0, flags=0) at block.c:2547
26. #10 0x00005555567023af in bdrv_co_do_rw
```

最终在paio_submit中会往线程池中提交一个请求thread_pool_submit_aio(pool, aio_worker, acb, cb, opaque), 由调度器去执行aio_worker函数,aio_worker是真正做IO操作的函数,它通过pwrite和pread去读取磁盘.

当qemu完成IO操作后,会在kvm_cpu_exec函数的循环中,调用kvm_vcpu_ioctl重新进入kvm.

以上阐述了IO操作在kvm和qemu中处理的整个过程.

参考资料:

1. kvm代码解析连载(二):io的虚拟化:<http://blog.csdn.net/fanwenyi/article/details/12748613>

分类: [虚拟化](#)

好文要顶

关注我

收藏该文



Jessica程序猿

关注 - 27

粉丝 - 544

+加关注

0

0

« 上一篇: [KVM虚拟机IO处理过程\(一\) ---- Guest VM I/O 处理过程](#)

» 下一篇: [文件作为块设备访问](#)

posted @ 2015-07-30 19:58 Jessica程序猿 阅读(1067) 评论(0) 编辑 收藏

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论, 请 [登录](#) 或 [注册](#), [访问](#) 网站首页。

【推荐】超50万行VC++源码: 大型组态工控、电力仿真CAD与GIS源码库

【推荐】腾讯云产品限时秒杀, 爆款1核2G云服务器99元/年!

相关博文:

- [qemu到kvm的处理, 再到vm的运行](#)
- [kvm和qemu交互处理io流程](#)
- [KVM初始化过程](#)
- [KVM的初始化过程](#)
- [QEMU-KVM 介绍2 概述](#)
- » [更多推荐...](#)

最新 IT 新闻:

- [再下一城 寒武纪科创板IPO进入“已问询”状态](#)
- [小鹏汽车成立贸易公司, 注册资本1000万元](#)
- [微软开始推送Windows 10 V2004: 修复大量错误、Bug](#)
- [美团外卖回应佣金话题: 每单平台利润不到2毛钱 将长期帮助商户](#)
- [三星和谷歌合作打造下一代Pixel智能手机 最早可能今年推出](#)
- » [更多新闻...](#)