

微风轻拂

科技+艺术+人性=美好

目录视图

摘要视图

RSS 订阅

个人资料



MaxYou

访问：18259次

积分：591分

排名：千里之外

原创：42篇 转载：0篇

译文：0篇 评论：11条

文章搜索

文章分类

技术 (36)

其他 (0)

创业 (6)

文章存档

2014年03月 (2)

2014年02月 (1)

2014年01月 (1)

2013年12月 (3)

2013年11月 (2)

展开

阅读排行

Golang中new和make的 (2043)

Wifi P2p连接步骤整理 (1370)

Golang协程与通道整理 (1292)

QEMU虚拟机关键源代码 (1244)

Golang面向对象之类型、 (882)

OpenID和OAuth的区别 (793)

Android APP开发的弯路 (680)

Golang继承中字段及方法 (479)

有奖征资源，博文分享有内涵

社区问答：非酋iOS测试指南

专访阿里陶辉

2014 CSDN博文大赛

10月微软MVP申请

QEMU虚拟机关键源代码学习

分类：技术

2012-03-11 12:01

1244人阅读

评论(3)

收藏

举报

虚拟机

工作

数据结构

signal

parameters

<--- 以下是2007年做相关工作的学习笔记(留意，版本比较早了，其后可能有各种改动)。多年过去了，世事变迁，加上我的水平非常有限，现在翻出来仅供批评，勿认真，而且现在我自己也看不懂了。😁 --->

QEMU是一款虚拟机软件，模拟了包括CPU在内的各种硬件系统，包括：

1. 指令解释和执行
2. 异常、中断、时钟等CPU相关模块
3. 内存、网卡、硬盘，显示系统，以及键盘和鼠标输入

QEMU虚拟机的初始化过程：

1. 分配、初始化虚拟的CPU、内存等数据结构。
2. 分配、初始化其他硬件设备
3. 如指定了CPU状态和内存镜像则将两者加载
4. 启动时钟系统
5. 调用cpu_exec运行guest系统

以下分若干小节介绍具体工作流程。

指令的翻译和执行过程

Guest即被虚拟的系统，Guest的指令系统不一定跟Host机相同，比如可以在x86上虚拟出一台arm机器来。对guest指令的翻译和执行过程如下：

1. 根据guest的CPU状态得到当前指令指针的guest逻辑地址；
2. 根据mmu状态得到在guest物理地址；
3. 加上guest内存在host内存的基址得到待执行指令的host逻辑地址；
4. 取出待执行指令，在微操作函数库中查找待执行指令的微操作函数（一小段C代码，但是已经编译为二进制）并拷贝到执行缓冲区TB；
5. 根据指令执行流程，对下一条指令重复上面步骤，直到发现一条jump指令，或者一条修改CPU状态并无法预测结果的指令，或者发现TB满了；
6. 给执行缓冲区TB生成一段段落结束代码，其中包含jump的跳向；
7. 调用dyngen函数对上述生成的代码做重定位，使其能寻址到guest数据区；
8. 执行上述生成的代码；

以上是虚拟机工作的最核心过程。

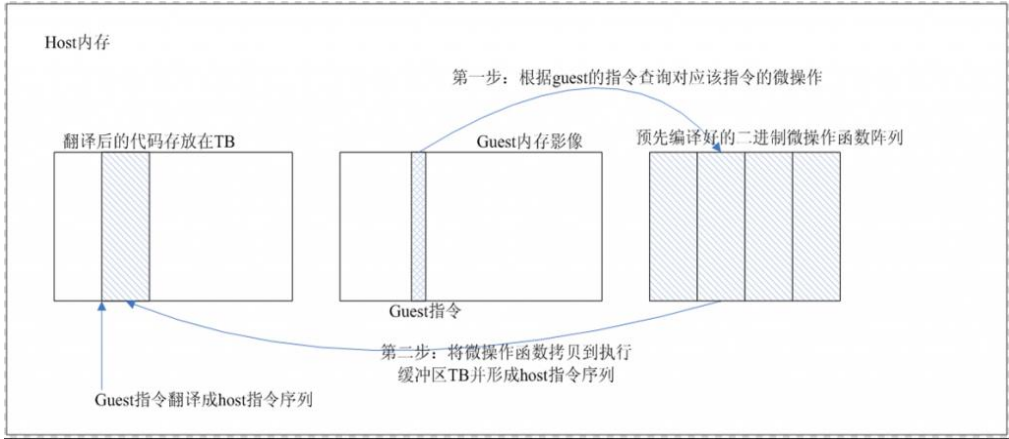
指令翻译过程的传送图

常用git命令整理	(435)
独立网站和互联网服务	(411)

评论排行	
Android APP开发的弯路	(5)
QEMU虚拟机关键源代码	(3)
独立网站和互联网服务	(2)
常用git命令整理	(1)
最简整理Binder Driver	(0)
另类角度思考HTML5和原	(0)
Golang中new和make的	(0)
从场景和需求梳理JNI接I	(0)
Golang面向对象之类型、	(0)
Golang协程与通道整理	(0)

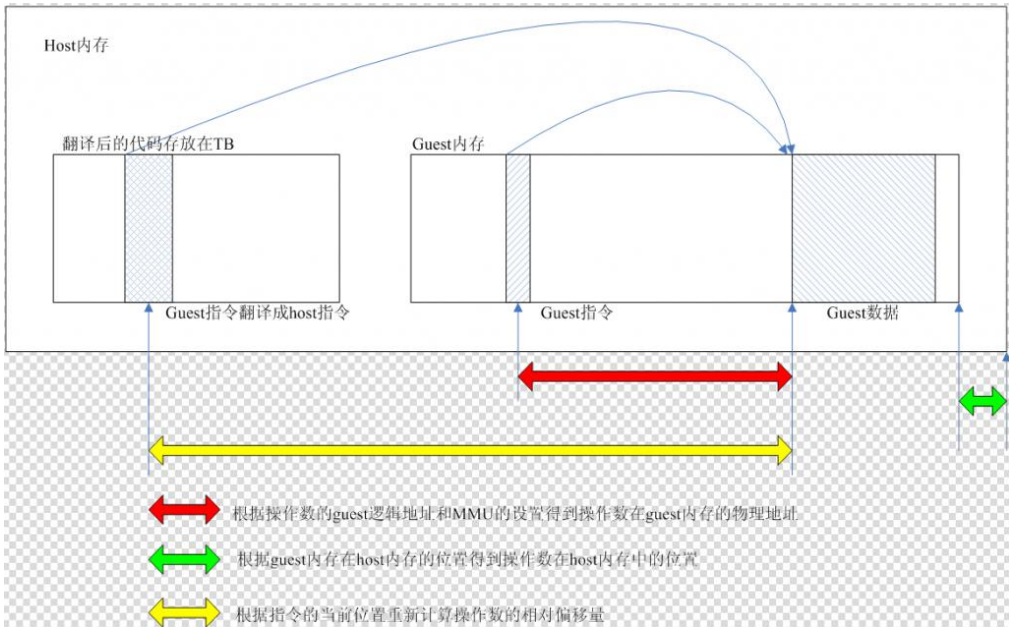
推荐文章	
------	--

最新评论	
常用git命令整理 yhshenghuang: fffffssddf	
QEMU虚拟机关键源代码学习 MaxYou: @tjhd1989:谢谢指出哦~，我去文章前面添加一个提醒。	
QEMU虚拟机关键源代码学习 tjhd1989: 写得非常不错，不过要注意的是这里研究的qemu版本为0.9.1，之后的版本有较大变动，指令的翻译转向...	
QEMU虚拟机关键源代码学习 trumpchi519: 学习了，谢谢，最近在研究这个	
Android APP开发的弯路教训 GEOLO: 但是做起类似微信一样的玩意就很多臭虫了！	
Android APP开发的弯路教训 MaxYou: @GEOLO:我以前参加过JavaEE培训，顺带有一点Android培训，但现在我倾向Android...	
Android APP开发的弯路教训 GEOLO: 还是走到J2EE吧，了解框架搭建，了解jdbc，了解ico，了解aop，了解ejb先！好多...	
Android APP开发的弯路教训 MaxYou: 哦，谢谢指教，回头我看看。	
Android APP开发的弯路教训 michael_yy: 使用 android 快速开发框架 afinal你讲在开发过程中过得很快快乐 哈哈	
独立网站和互联网服务 MaxYou: @simontam:嗯嗯，除非必要才建立独立网站。	



1. Op.c文件编译为obj格式的微操作函数库。
2. 每条guest指令对应一个微操作序列。
3. 拷贝guest指令对应的微操作序列到TB中，形成host指令序列。

Guest和host的寻址转换关系



说明：

1. Disas_insn(): 对guest代码逐条翻译，并存放在TB中。
2. Dyngen_code(): 重定向翻译后的代码。

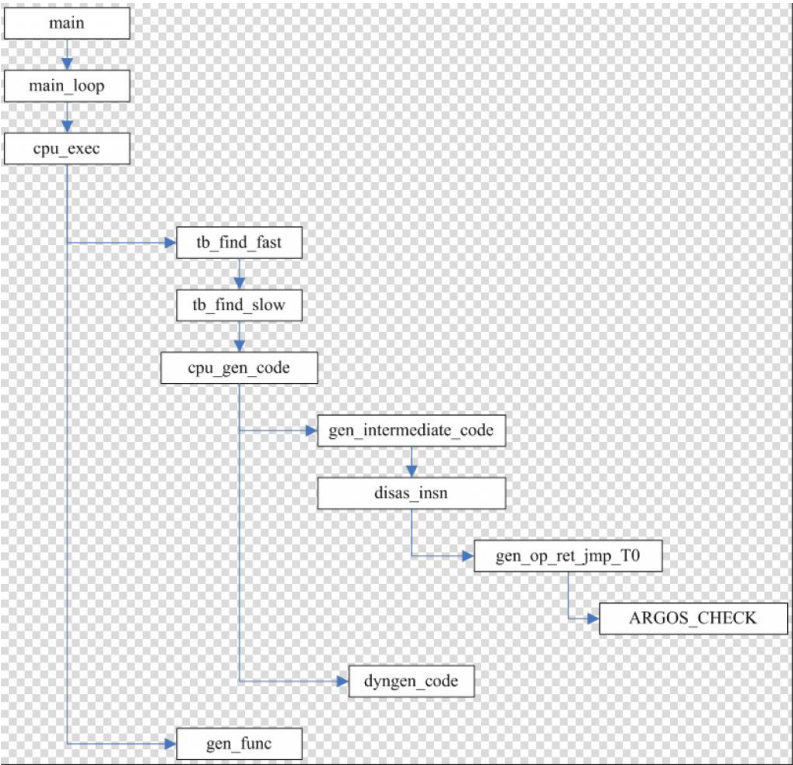
TB相关数据结构

1. TB是一个数组：TranslationBlock tbs[CODE_GEN_MAX_BLOCKS];
2. 使用一个计数器来指向空闲TB：int nb_tbs;
3. 使用一个hash索引的指针数组来指向各个TB：TranslationBlock
*tb_phys_hash[CODE_GEN_PHYS_HASH_SIZE];

从地址索引到TB的过程如下：

1. phys_pc = get_phys_addr_code(env, pc); //从逻辑地址得到物理地址
2. h = tb_phys_hash_func(phys_pc); //取低位的hash操作;
3. ptb1 = &tb_phys_hash[h]; //用提炼的h做hash表索引得到指针

函数调用主路径



几个关键函数：

1. Cpu_exec(): 放入CPU状态，然后看TB表中有没有翻译好的代码，没有则调用tb_find_fast()去生成翻译好的代码，其后调用gen_func()执行翻译好的代码。
2. Disas_insn(): 对guest代码逐条翻译。
3. Dyngen_code(): 重定向翻译后的代码。
4. gen_op_ret_jump_T0(): 翻译后的跳转指令，在其中插入argos检查代码。

对guest代码逐条翻译

函数disas_insn()对guest代码进行逐条的翻译。翻译的过程如下：

1. 首先处理指令前部分的prefix；
2. 解析操作码；
3. 调用gen_op_*()函数生成一段微操作代码；
4. 返回（此刻指令指针指向下一条指令）；

最典型的一个指令解析如下：

```
[cpp]
01. case 0x89: /* mov Gv, Ev */
02.     if ((b & 1) == 0)         ot = OT_BYTE;
03.     else                     ot = dflag + OT_WORD;
04.     modrm = ldub_code(s->pc++);
05.     reg = ((modrm >> 3) & 7) | rex_r;
06.     gen_ldst_modrm(s, modrm, ot, reg, 1);
07.     break;
```

但是涉及到跳转指令的时候有些变化，即此刻的指令解析应该告一个段落，返回到TB并执行。

特点---动态翻译、重定位

Qemu文档中声称的不同于其他虚拟机的特点：

1. *QEMU is a dynamic translator. The basic idea is to split every x86 instruction into fewer simpler instructions. (target-i386/op.c)* 把一条guest指令分解为功能相同的一小段host代码，也即指令的微操作函数
2. *A key idea to get optimal performances is that constant parameters can be passed to the simple operations. For that purpose, dummy ELF relocations are generated with gcc for each constant parameter. Then, the tool ('dyngen') can locate the relocations and generate the appropriate C code to resolve them when building the dynamic code.* 常量参数可以带入微操作函数，dyngen可以根据重定位信息并生成恰当的动态代码供执行

指令翻译的优化过程

将多个微操作串接，这样可以省略每个微操作的调用和返回开支

确保只有一个出口，这样可以消除入口和出口的开销。如下的出入口部分就可以消除了：—entrance—body—exit—entrance—body—exit—entrance—body—exit—

注意要确保这个出口在最后。一个普通函数的出口不一定在最后，即使在C代码的最后，也不一定是编译后的最后一条指令。比如：

```
[cpp]
01.  int global, global2;
02.  void quux(int a)
03.  {
04.      if (a > 0) {
05.          global = 0;
06.          goto out;
07.      } else {
08.          global = 1;
09.          goto out;
10.      }
11.  out:
12.      global2 = 1;
13.      return;
14.  }
```

可能编译为如下：

```
[cpp]
01.  quux      CMPL a+0x0(FP), $0x0
02.  quux+0x5  JLE quux+0x1c(SB)
03.  quux+0x7  MOVL $0x0, global(SB)
04.  quux+0x11 MOVL $0x2, global2(SB)
05.  quux+0x1b RET
06.  quux+0x1c MOVL $0x1, global(SB)
07.  quux+0x26 JMP quux+0x11(SB)
```

重定位原理---常量

立即数指令在翻译时的麻烦：

1. 微操作二进制段是直接拷贝到TB的，本身无法带参数；
2. 只能先拷贝过去，然后修改指令的立即数部分；

问题：待翻译指令中有立即数，可否拷贝之后从待翻译指令中取立即数？

常数重定位的意义在于，无需堆栈或者变量来中转，直接生成了host的立即数指令

1. 首先obj文件解析为符号表，重定位条目，代码区；
2. 每个微操作码的常量参数使用特殊的符号名：__op_paramN，N是个序列号，这样就可以在重定位条目中找到对应项；
3. 编译后的微操作码被拷贝到指定位置，然后从重定位条目中找到重定位信息，实现运行时的正确重定向；

```
[cpp]
```

```

01.  case INDEX_op_addl_T0_im:
02.  {
03.      long param1;
04.      extern void op_addl_T0_im();
05.      memcpy(gen_code_ptr, (char *)&op_addl_T0_im+0, 6);
06.      param1 = *opparam_ptr++;
07.      *(uint32_t *)(gen_code_ptr + 2) = param1;
08.      gen_code_ptr += 6;
09.      break;
10.  }

```

其中opparam_ptr指向运行时参数列表，在本例中该值被带入gen_code_ptr指向的指令立即数位置。

重定位原理---变量和子函数

微操作可能需要引用helper类子函数来实现复杂功能，引用变量的重定位也一样，只要需要修改相对偏移量即可。

```

[cpp]
01.  {
02.      extern void op_cpuid();
03.      extern char helper_cpuid;
04.      memcpy(gen_code_ptr, (void *)((char *)&op_cpuid+0), 13);
05.      *(uint32_t *)(gen_code_ptr + 5) = (long)&helper_cpuid - (long)
(gen_code_ptr + 5) + -4;
06.      gen_code_ptr += 13;
07.  }

```

以上参见dyngen.c的gen_code()函数，解释如下：

1. 前面两条语句说明微操作op_cpuid及子函数helper_cpuid都在host内存的进程空间中，是随qemu编译加载的，里面的偏移量是对的，即op_cpuid能调用到helper_cpuid。
2. 第三条语句拷贝op_cpuid代码段到TB区，由于op_cpuid是相对寻址helper_cpuid，由于调用者自身移动了，故这个拷贝版本op_cpuid不能寻址到helper_cpuid了。
3. 第四条语句修改op_cpuid的call指令的参数（被调子函数的相对偏移量），使之正确指向helper_cpuid，这样op_cpuid就可以调用到helper_cpuid了。
4. 最后一条语句移动下面一块空地，准备接收翻译的下一条指令。

重定位实际过程

指令重定位实际过程

1. 调用dyngen_labels(gen_labels, nb_gen_labels, gen_code_buf, gen_opc_buf)设置某些标签，为下一步编译做准备；
2. 参见函数dyngen_code(uint8_t *gen_code_buf, uint16_t *label_offsets, uint16_t *jmp_offsets, const uint16_t *opc_buf, const uint32_t *opparam_buf, const long *gen_labels);
3. 最后生成的适合在host上跑的文件存放在TB里面；

Dyngen函数

```

[cpp]
01.  int dyngen_code(uint8_t *gen_code_buf, uint16_t *label_offsets, uint16_t *jmp_offsets,
02.                  const uint16_t *opc_buf, const uint32_t *opparam_buf, const long *gen_lab
03.  {
04.      for(;;) {
05.          switch(*opc_ptr++) {
06.              {在gen_code(name, sym->st_value, sym->st_size, outfile, 1)输出如下信息（不全）：
07.                  extern char __dot_%s __asm__("%s\");
08.                  memcpy(gen_code_ptr, (void *)((char *)&sym->st_value, %d); //是拷贝二进制的地方
09.                  /* 输出偏移量信息 emit code offset information */
10.                  label_offsets[%d] = %ld + (gen_code_ptr - gen_code_buf);
11.                  param%d = *opparam_ptr++;
12.                  jmp_offsets[%d] = %d + (gen_code_ptr - gen_code_buf);

```

```

13.         *(uint32_t *) (gen_code_ptr + %d) = %s + %d;
14.         *(uint32_t *) (gen_code_ptr + %d) = %s - (long)(gen_code_ptr + %d) + %d;
15.         gen_code_ptr += %d;
16.         *gen_opparam_ptr++ = param%d;
17.         *gen_opc_ptr++ = INDEX_%s;
18.     }
19.     case INDEX_op_nop:                break;
20.     case INDEX_op_nop1:                opparam_ptr++;                break;
21.     case INDEX_op_nop2:                opparam_ptr += 2;                break;
22.     case INDEX_op_nop3:                opparam_ptr += 3;                break;
23.     default:                goto the_end;
24. }
25. }
26. the_end:
27.     flush_icache_range((unsigned long)gen_code_buf, (unsigned long)gen_code_ptr);
28.     return (gen_code_ptr - gen_code_buf);
29. }

```

从上面的代码看，似乎dyngen()函数的功能是为前面生成的二进制微操作码生成重定位辅助结构。

在dyngen.c中用fprintf()打印dyngen()函数为文件，命令行上可以确定文件名。

invalidate指令

某些指令在执行的过程中会修改自身

1. 该指令在TB中有一份翻译后的，在guest内存中有一份原始的，修改指令自身是修改在guest内存中的那一份（这里可能有疑问）；
2. 此类情况发生后则立即invalidate，也即从内存往TB更新，始终保证TB中含有最新的指令；
3. 给指令所在页面做了bitmap，注意该页面可能也有数据区，该页面被修改时可以根据bitmap判定究竟是指令被修改了还是数据被修改了，只invalidate指令。

Invalidate更新流程

1. 将指令区域设置为不可写，在signal信号处理函数中添加对应的写错误函数，在该函数中invalidate对应的TB区；
2. 自修改指令运行后将写TB区，导致signal信号处理函数被调用；

异常和中断处理

在每个小循环中处理中断，在每个大循环中处理异常。

```

[cpp]
01. For(;;)
02. {
03.     如果有异常则处理，入口是do_interrupt ();
04.     For(;;)
05.     {
06.         如果有中断则处理，入口和异常相同： do_interrupt()
07.         tb = tb_find_fast(); gen_func(); 执行逐个TB里面的代码；
08.         怎样跳出并到上级循环的？不清楚。
09.     }
10. }

```

Guest的异常通常用host的信号来模拟

上一篇 JVM和QEMU虚拟机的对比学习

下一篇 OpenID和OAuth的区别

顶 1

踩 0

主题推荐 虚拟机 源代码 qemu 数据结构 二进制

猜你在找

- qemu虚拟机与外部网络的通信

【虚拟机】虚拟化技术以及KVM、QEMU与libvirt介

qemu 启动虚拟机 常用命令(2)---drive 的使用

qemu 启动虚拟机 常用命令(1)

Ubuntu12.10 下搭建基于KVM-QEMU的虚拟机环境
- CPU常识+Qemu虚拟机CPU配置

KVM虚拟机IO处理过程(二) ----QEMU/KVM I/O 处


qemu虚拟机通过tun/tap上网

KVM虚拟机代码揭秘——QEMU代码结构分析


Ubuntu 下安装kvm, qemu, libvirt, 并新建虚拟机的过

查看评论


- 2楼 [tjhd1989](#) 2013-08-02 16:31发表



写得非常不错，不过要注意的是这里研究的qemu版本为0.9.1，之后的版本有较大变动，指令的翻译转向tcg了。
- Re: [MaxYou](#) 2013-08-04 02:05发表



回复tjhd1989：谢谢指出哦~，我去文章前面添加一个提醒。
- 1楼 [trumpchi519](#) 2012-12-09 22:52发表



学习了，谢谢，最近在研究这个

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题

Java VPN Android iOS ERP IE10 Eclipse CRM JavaScript Ubuntu NFC

WAP jQuery 数据库 BI HTML5 Spring Apache Hadoop .NET API HTML SDK

IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails

QEMU KDE Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6

Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby

Hibernate ThinkPHP Spark HBase Pure Solr Angular Cloud Foundry Redis Scala

Django Bootstrap