http://blog.csdn.net/ustc\_dylan

[QEMU源码分析系列(一）](http://blog.csdn.net/ustc_dylan/article/details/6784876)

1 qemu概述

qemu是一种快速的多体系结构仿真器，通过动态翻译的技术达到了优异的仿真速度。目前，qemu支持两种操作模式：

全系统仿真模式。在这种模式下，qemu完整的仿真目标平台，此时，qemu就相当于一台完整的pc机，例如包括一个或多个处理器以及各种外围设备。这种模式可以用来运行不同的操作系统或调试操作系统的代码。

用户态仿真模式。在这种模式下，qemu能够运行不同于主机平台的其他平台的程序（比如，在x86平台上运行为arm平台编译的程序），其中典型的代表wine windows API emulator。另外，在这种模式下能够进行方便的交叉编译和调试。

对于全系统仿真模式，qemu目前可以支持的硬件列表如下：

x86 or x86\_64 体系结构处理器

ISA PC （没有PCI总线的PC）

PowerPC 处理器

32/64bit的SPARC 处理器

32/64bit的MIPS处理器

ARM体系结构的处理器

PXA 270、PXA 255

OMAP 310、OMAP 2420、OMAP 310

对于用户态仿真模式，qemu支持的硬件列表如下：x86 (32 and 64 bit), PowerPC (32 and 64 bit), ARM, MIPS (32 bit only), Sparc (32 and 64 bit), Alpha, ColdFire(m68k), CRISv32 and MicroBlaze CPUs are supported.

1.1 qemu 的本质

众所周知，Bochs 是一款可移植的IA-32仿真器，它利用模拟的技术来仿真目标系统，具体来说，将是将目标系统的指令分解，然后模拟分解后的指令以达到同样的效果。这种方法将每一条目标指令分解成多条主机系统的指令，很明显会大大降低仿真的速度。

qemu则是采用动态翻译的技术，先将目标代码翻译成一系列等价的被称为“微操作”（micro-operations）的指令，然后再对这些指令进行拷贝，修改，连接，最后产生一块本地代码。这些微操作排列复杂，从简单的寄存器转换模拟到整数/浮点数学函数模拟再到load/store操作模拟，其中load/store操作的模拟需要目标操作系统分页机制的支持。

qemu对客户代码的翻译是按块进行的，并且翻译后的代码被缓存起来以便将来重用。在没有中断的情况下，翻译后的代码仅仅是被链接到一个全局的链表上，目的是保证整个控制流保持在目标代码中，当异步的中断产生时，中断处理函数就会遍历串连翻译后代码的全局链表来在主机上执行翻译后的代码，这就保证了控制流从目标代码跳转到qemu代码。简单概括下：指定某个中断来控制翻译代码的执行，即每当产生这个中断时才会去执行翻译后的代码，没有中断时仅仅只是个翻译过程而已。这样做的好处就是，代码是是按块翻译，按块执行的，不像Bochs翻译一条指令，马上就执行一条指令。

1.2 qemu能够模拟的硬件

Bochs和qemu从非常低的层次对硬件进行模拟，对于像总线和外围设备如显卡，网卡，磁盘控制器等都有相对应的软件的表示，但是二者仅对有限的硬件集合进行精确的模拟，比如对中断控制器，总线驱动，磁盘驱动，键盘，鼠标，显卡以及网卡的模拟。随着时间的推移，可模拟的硬件集合将会扩展到客户操作系统能够支持的尽可能多的设备。Qemu和Bochs都利用运行在模拟机中的BIOSes来初始化硬件的某些部分，这种设计思想使得对设备的仿真忠于原始的硬件。

除了模拟之外，设备驱动利用主机的功能来提供模拟和用户要求的功能，下面来看几个例子：

帧缓冲区通过用户可选择的接口被暴露出来，对于qemu来说，帧缓冲有SDL window，VNC Server和无图形界面的输出三个可供选择的选项

qemu中的网络可以是被禁止的，可以是被桥接到主机的，可以使用虚拟以太网协议创建的Unix套接字，还可以是被在qemu中被完全模拟的

1.3 可移植性

qemu采用了模块化的设计思想，仿真器中与目标平台相关的部分被分离到它们自己的文件和目录中。对于核心部分，驱动部分和动态翻译器来说，所有目标平台都声明相同的接口，在整个qemu的111，000行代码中，目标平台相关的组件代码大约占了1/3，特别地，x86目标平台的大约不超过8000行。与Bochs不同的是，qemu对目标平台的描述非常的紧凑，因此，可以模拟大量的目标平台。

qemu要求公开有关编译执行的信息以便动态翻译器使用，幸运地是，这些信息中的绝大部分对于调试器，动态连接器和单独编译来说都是非常必要的。另外，qemu完全由C语言编写，在主机和目标平台环境之间创建了一个隔离层。值得一提的是动态翻译器使用了带GNU扩展的C编写，这种结构化的可移植性，再加上GCC对大量系统的支持，使得qemu在主机系统之间的可移植性大大增加。

本文仅仅简单介绍下了qemu的相关知识，下一篇文章将会深入分析qemu的实现原理。

希望对qemu/kvm或虚拟化技术的爱好者多多交流，我的邮箱:ustc.dylan@gmail.com。

[qemu源码分析系列(二)](http://blog.csdn.net/ustc_dylan/article/details/6807731)

 在qemu源码分析系列(一)简单介绍了下qemu相关的背景知识，本文将详细分析qemu的核心 -- 动态翻译器。

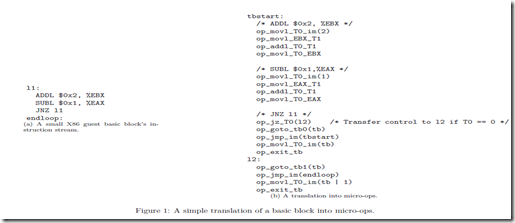
为了更容易理解动态翻译技术，我们暂时忽略掉qemu的其他模块，如用户交互模块，硬件模拟等模块，而是从数据结构的设计，数据结构之间的操作及其应用等方面来进行详细地分析，重点关注动态翻译器和微操作库(micro-ops library)的原理，至于细节的东西可以放在以后去深入分析。

qemu利用了一种可移植的动态代码翻译器以快速地完成客户代码的仿真。qemu本身并不能识别它主机体系结构的指令集，作为替代，每一个客户机指定一个c语言实现的微操作库以及一个客户机代码反汇编器和翻译器，用来将客户代码转换成微操作表，这些微操作可以被认为是一种虚拟机，尽管仅仅是对客户系统模拟的一种优化而已。另外，这些操作本身包括寄存器转化，显示的条件代码更新代码，按位操作，整型和浮点型数学函数，内存加载和存储操作等。

2.1 翻译

2.1.1 基本块的翻译

图1描述了一小段带条件跳转x86代码以及其对应的微操作指令表示，每一条op\_开头的微操作指令都将被拷贝到翻译缓冲区中，微操作指令中看起来像参数的常量被称为折叠常量(const folding)，在2.3.4接将会进行详细的解释，现在只要把它看成是qemu中定义的微操作指令的参数就可以了。微操作指令的参数中，tb是比较特殊的，它指向与当前翻译缓冲区相关的元数据。另外，JNZ指令对应的微操作指令看起来比较怪异，控制流处理相关的细节将会在2.2.3节和2.2.4节详细介绍。

[](http://blog.chinaunix.net/link.php?url=http://blog.chinaunix.net%2Fattachment%2F201109%2F23%2F20940095_1316789491n7Nz.png)

2.1.2 同步的错误安全舱口

严格的按照“基本块”（basic block）的方式译码将会使得翻译缓冲区中包含一条或多条客户机指令，同时几乎每一条指令都可能产生同步的fault(比方说MMU fault)。为了便于理解，在翻译的时候我们重点考虑比较直接的客户机控制流（比方说，分支跳转：条件跳转和非条件跳转），毕竟，同步的错误(fault)很少出现。对于同步错误(fault)需要一种恢复机制，用来处理翻译缓冲区中的fault。qemu使用longjmp()从翻译缓冲区中跳到仿真器核心代码中。这里描述的意思是：当在执行翻译缓冲区的代码时，遇到了fault，就需要将执行路径切换到qemu的代码中，另外，当qemu处理完fault后，会重新创建一个翻译缓冲区。

2.1 翻译缓冲区的高超技巧

qemu采用了一些不同寻常的技巧来提高性能：

* 冗长的、额外开销大的主机操作（如仿真客户的MMU操作）并不会被直接放进翻译缓冲区，而是以可以从微操作中调用的帮助函数的形式存在
  + 一方面减少了对翻译缓冲区的占用，另一方面简化了翻译的过程（qemu中事先保存某些固定指令的翻译结果，当动态翻译器碰到这些指令时，直接跳过去执行就行了）
* 翻译后的微操作指令序列的优化
* 启发式的译码和翻译

2.2.1 通过函数调用来达到节省翻译缓冲区的技术

目前操作系统中的MMU操作指令自身非常复杂。对于MMU操作会产生大量的访存操作，通常的解决方法是增加一个cache，但这进一步增加代码的复杂度。如果将每一次访存操作的这些复杂代码都放入翻译缓冲区中，其代价将会非常的昂贵。除了MMU操作指令，一些特殊的客户机指令也是非常复杂的，如CPUID指令，即使在简化的qemu的实现中，一条x86的CPUID指令都需要75行C代码来实现。与ARPL指令相比，CPUID对于不同的寄存器内容将会表现是完全不同的行为，因此，必须需要一大串冗长的微操作指令来实现CPUID的功能。

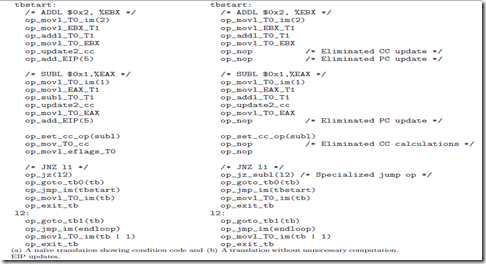
为了优化这种代价昂贵，需要冗长的微操作指令来实现的客户机指令，qemu采用能够实现类似函数调用功能的微操作指令的机制来实现，比如ldl\_kernel，实现了内核模式的大量数据的读功能；helper\_cpuid，这个微操作包含了CPUID指令的实现。

2.2.2 惰性赋值

每一条指令都会隐含性修改指令指针(EIP)，并且每一条指令都会修改处理器条件码（比如零标志位，溢出标志位、进位标志位等），但实际的情况是，只要指令按照正确的顺序执行，其实客户代码很少去关心这些状态的变化。对于这些条件码除了条件跳转状态位外，我们很少去关注其他的状态位。图1中一段小小的客户代码都会引起指令指针的修改和条件码修改的大量操作：

* ADDL操作会修改条件码的零和进位/借位标志位
* SUBL操作会修改条件码的零和进位/借位标志位
* ADDL和SUBL都会修改EIP，使其指向下一条指令

由于qemu是按照“基本代码块”为单位进行翻译的，所以只有在整个“基本代码块”翻译完成或显示的读取EIP的时候才会对EIP进行更新，这就避免了EIP的频繁更新的问题。在实际情况下，ADDL和SUBL都不会去读指令指针EIP，因此，可以优化掉翻译后的微操作指令中对EIP更新的微操作，具体描述如图2所示。图2中仅仅是在每个“基本代码块”的最后通过op\_jmp\_im微操作来进行EIP的更新，即每个“基本代码块”只做一次更新操作。



太困了，明天补上！

[QEMU源码分析系列（三）](http://blog.csdn.net/ustc_dylan/article/details/6827663)

从QEMU-0.10.0开始，TCG成为QEMU新的翻译引擎，使QEMU不再依赖于GCC3.X版本，并且做到了“真正”的动态翻译（从某种意义上说，旧版本是从编译后的目标文件中复制二进制指令）。TCG的全称为“**Tiny Code Generator**”，QEMU的作者Fabrice Bellard在TCG的说明文件中写到，TCG起源于一个C编译器后端，后来被简化为QEMU的动态代码生成器（Fabrice Bellard以前还写过一个很牛的编译器TinyCC）。实际上TCG的作用也和一个真正的编译器后端一样，主要负责分析、优化Target代码以及生成Host代码。

Target指令 ----> TCG ----> Host指令

**以下的讲述以X86平台为例（Host和Target都是X86）。**

我在上篇文章中讲到，**动态翻译的基本思想就是把每一条Target指令切分成为若干条微指令，每条微指令由一段简单的C代码来实现，运行时通过一个动态代码生成器把这些微指令组合成一个函数，最后执行这个函数，就相当于执行了一条Target指令。**

这种思想的基础是因为CPU指令都是很规则的，每条指令的长度、操作码、操作数都有固定格式，根据前面就可推导出后面，所以只需通过反汇编引擎分析出指令的操作码、输入参数、输出参数等，剩下的工作就是编码为目标指令了。

那么现在的CPU指令这么多，怎么知道要分为哪些微指令呢？其实CPU指令看似名目繁多，异常复杂，实际上多数指令不外乎以下几大类：

数据传送、算术运算、逻辑运算、程序控制；

例如，数据传送包括：传送指令（如MOV）、堆栈操作（PUSH、POP）等

程序控制包括：函数调用（CALL）、转移指令（JMP）等；

基于此，TCG就把微指令按以上几大类定义（见tcg/i386/tcg-target.c），例如：其中一个最简单的函数 tcg\_out\_movi 如下：

// tcg/tcg.c  
static inline void tcg\_out8(TCGContext \*s, uint8\_t v)  
{  
\*s->code\_ptr++ = v;  
}

static inline void tcg\_out32(TCGContext \*s, uint32\_t v)  
{  
\*(uint32\_t \*)s->code\_ptr = v;  
s->code\_ptr += 4;  
}

// tcg/i386/tcg-target.c  
static inline void tcg\_out\_movi(TCGContext \*s, TCGType type,  
int ret, int32\_t arg)  
{  
if (arg == 0) {  
/\* xor r0,r0 \*/  
tcg\_out\_modrm(s, 0x01 | (ARITH\_XOR << 3), ret, ret);  
} else {  
tcg\_out8(s, 0xb8 + ret); // 输出操作码，ret是寄存器索引  
tcg\_out32(s, arg); // 输出操作数  
}  
}

0xb8 - 0xbf 正是x86指令中的 mov R, Iv 系列操作的16进制码，所以，tcg\_out\_movi 的功能就是输出 mov 操作的指令码到缓冲区中。**可以看出，TCG在生成目标指令的过程中是采用硬编码的，因此，要让TCG运行在不同的Host平台上，就必须为不同的平台编写微操作函数。**

接下来，我还是以一条Target指令 jmp f000:e05b 来讲述它是如何被翻译成Host指令的。其中几个关键变量的定义如下：

gen\_opc\_buf： 操作码缓冲区  
gen\_opparam\_buf：参数缓冲区  
gen\_code\_buf： 存放翻译后指令的缓冲区  
gen\_opc\_ptr、gen\_opparam\_ptr、gen\_code\_ptr三个指针变量分别指向上述缓冲区。

jmp f000:e05b 的编码是：EA 5B E0 00 F0，

首先是disas\_insn()函数翻译指令，当碰到第1个字节EA，分析可知这是一条16位无条件跳转指令，因此依次从后续字节中得到offset和selector，然后分为如下微指令操作：

gen\_op\_movl\_T0\_im(selector);  
gen\_op\_movl\_T1\_imu(offset);  
gen\_op\_movl\_seg\_T0\_vm(R\_CS);  
gen\_op\_movl\_T0\_T1();  
gen\_op\_jmp\_T0();

这几个微指令函数的定义如下（功能可看注释）：

static inline void gen\_op\_movl\_T0\_im(int32\_t val)  
{  
tcg\_gen\_movi\_tl(cpu\_T[0], val); // 相当于 cpu\_T[0] = val  
}

static inline void gen\_op\_movl\_T1\_imu(uint32\_t val)  
{  
tcg\_gen\_movi\_tl(cpu\_T[1], val); // 相当于 cpu\_T[1] = val  
}

static inline void gen\_op\_movl\_seg\_T0\_vm(int seg\_reg)  
{  
tcg\_gen\_andi\_tl(cpu\_T[0], cpu\_T[0], 0xffff); // cpu\_T[0] = cpu\_T[0]&0xffff  
tcg\_gen\_st32\_tl(cpu\_T[0], cpu\_env,   
offsetof(CPUX86State,segs[seg\_reg].selector)); // the value of cpu\_T[0] store to the 'offset' of cpu\_env  
tcg\_gen\_shli\_tl(cpu\_T[0], cpu\_T[0], 4); // cpu\_T[0] = cpu\_T[0]<<4  
tcg\_gen\_st\_tl(cpu\_T[0], cpu\_env,   
offsetof(CPUX86State,segs[seg\_reg].base)); // the value of cpu\_T[0] store to the 'offset' of cpu\_env  
}

static inline void gen\_op\_movl\_T0\_T1(void)  
{  
tcg\_gen\_mov\_tl(cpu\_T[0], cpu\_T[1]); // cpu\_T[0] = cpu\_T[1]  
}

static inline void gen\_op\_jmp\_T0(void)  
{  
tcg\_gen\_st\_tl(cpu\_T[0], cpu\_env, offsetof(CPUState, eip)); // // the value of cpu\_T[0] store to the 'offset' of cpu\_env  
}

其中，cpu\_T[0]、cpu\_T[1]和前面讲过的T0、T1功能一样，都是用来临时存储的变量。在32位目标机上，tcg\_gen\_movi\_tl 就是 tcg\_gen\_op2i\_i32 函数，它的定义如下：

static inline void tcg\_gen\_op2i\_i32(int opc, TCGv\_i32 arg1, TCGArg arg2)  
{  
\*gen\_opc\_ptr++ = opc;  
\*gen\_opparam\_ptr++ = GET\_TCGV\_I32(arg1);  
\*gen\_opparam\_ptr++ = arg2;  
}

static inline void tcg\_gen\_movi\_i32(TCGv\_i32 ret, int32\_t arg)  
{  
tcg\_gen\_op2i\_i32(INDEX\_op\_movi\_i32, ret, arg);  
}

gen\_opparam\_buf 是用来存放操作数的缓冲区，它的存放顺序是：第1个4字节代表s->temps（用来存放目标值的数组，即输出参数）的索引，第2个4字节及之后字节代表输入参数，对它的具体解析过程可见 tcg\_reg\_alloc\_movi 函数，示例代码如下：

TCGTemp \*ots;  
tcg\_target\_ulong val;

ots = &s->temps[args[0]];  
val = args[1];

ots->val\_type = TEMP\_VAL\_CONST;  
ots->val = val; // 把输入值暂时存放在ots结构中

接下来，根据 gen\_opc\_buf 保存的操作码列表，gen\_opparam\_buf 保存的参数列表，以及TCGContext结构，经过 tcg\_gen\_code\_common 函数调用，jmp f000:e05b 生成的最终指令如下：

099D0040 B8 00 F0 00 00 mov eax,0F000h   
099D0045 81 E0 FF FF 00 00 and eax,0FFFFh   
099D004B 89 45 50 mov dword ptr [ebp+50h],eax   
099D004E C1 E0 04 shl eax,4   
099D0051 89 45 54 mov dword ptr [ebp+54h],eax   
099D0054 B8 5B E0 00 00 mov eax,0E05Bh   
099D0059 89 45 20 mov dword ptr [ebp+20h],eax   
099D005C 31 C0 xor eax,eax   
099D005E E9 25 5D CA 06 jmp \_code\_gen\_prologue+8 (10675D88h) /\* 返回 \*/

从上面可以看出，生成的Host代码很简洁，对于Target机的JMP，Host没有去执行真正的跳转指令，而只是简单的将目标地址放到EIP中而已。

QEMU维护着一个称为 CPUState 的数据结构，这个结构包括了Target机CPU的所有寄存器，像EAX，EBP，ESP，CS，EIP，EFLAGS等。

它总是代表着Target机的当前状态，我用env变量来表示 CPUState 结构，

QEMU每次解析Target指令时，总是以 env.cs+env.eip 为开始地址的。   
  
像上面说的jmp f000:e05b指令，它分解为如下微操作：   
  
gen\_op\_movl\_T0\_im(selector);   
gen\_op\_movl\_T1\_imu(offset);   
gen\_op\_movl\_seg\_T0\_vm(R\_CS);   
gen\_op\_movl\_T0\_T1();   
gen\_op\_jmp\_T0();   
  
这几条微操作的意义概括起来很简单，就是把selector放到env.cs，把offset放到env.eip。在调试中，把QEMU执行Target指令的过程和Bochs比较是一件很有趣的事情，当然，这只是设计理念的不同，而并没有技术上的优劣之分。

[qemu源码分析之四--dyngen动态翻译技术](http://blog.csdn.net/ustc_dylan/article/details/6857014)

由于刚刚接触qemu，所以前面几篇文章仅仅是肤浅的介绍qemu的一些背景知识，今天突然感觉前面说的太没有条理了，而且大部分是读别人的文章，一知半解，没有自己的总结体会，今天感觉稍微有点心得，敬请指教。

1. 明确guest和host

    对于qemu而言，被仿真的平台成为guest或者说target；很明显，运行qemu的平台就称为host。

2. 了解qemu动态翻译技术的发展

    qemu运用动态翻译的技术将guest binary instructions动态翻译成host binary instructions，之后由host运行翻译后的指令。在qemu-0.9之前的版本都采用dyngen的动态翻译技术，而从qemu-0.10开始的版本开始采用TCG（Tiny  Code Generator）的翻译技术。

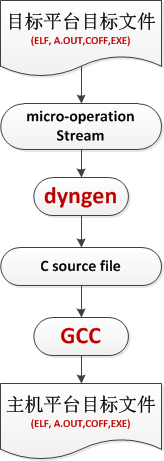
    采用dyngen  动态翻译技术的资料主要有以下两篇文章，是了解动态翻译技术入门的好文章（在后续的分析中，会简单介绍dyngen技术）：

* QEMU, a Fast and Portable Dynamic Translator
* Porting QEMU to Plan 9: QEMU Internals and Port Strategy

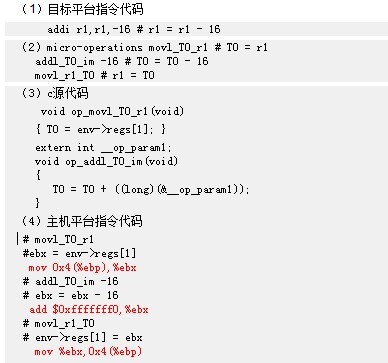
    介绍TCG技术的文章则相对较少，主要是阅读qemu源码和qemu官网上的相关资料。

3. dyngen简单介绍

     图1简单说明了qemu采用dyngen动态翻译技术将目标平台指令翻译成主机平台指令的简单过程。



假设target为PowerPC，host为x86，说明整个翻译过程：



**[html]** [view plaincopy](http://blog.csdn.net/ustc_dylan/article/details/6857014)

1. **<pre** style="BACKGROUND-COLOR: rgb(240,240,240); MARGIN: 4px 0px" class="html" name="code"**><pre** class="html" name="code"**><pre></pre>**
2. **<pre></pre>**
3. **<pre></pre>**
4. **<pre></pre>**
5. **<pre></pre>**
6. **<pre></pre>**
7. **<pre></pre>**
8. **<pre></pre>**
9. **<pre></pre>**
10. **<pre></pre>**
11. **<pre></pre>**
12. **<pre></pre>**
13. **<pre></pre>**
14. **<pre></pre>**
15. **<pre></pre>**
16. **<pre></pre>**
17. **<pre></pre>**
18. **</pre></pre>**

    dyngen在整个过程中扮演了非常重要的角色，其详细功能在这里不再详述，但是有一个疑问，希望与大家讨论：

在将目标平台指令集向micro-operations这一步的转化中，《QEMU, a Fast and Portable Dynamic Translator》一文中提到采用了hand coded code的方式，我理解是我们说的“硬编码”，即目标平台指令到micro-op是预先写好的一一映射的关系，我的疑问就是这种一一映射的关系是怎么实现的，因为没有看过qemu-0.9之前版本的代码，所以很想知道是怎么hand coded 的。另外，该文还提到“When QEMU first encounters a piece of target code, it translates it to host code ... ....”, 我的问题是qemu怎么处理目标平台的objective file的， 比方说qemu怎么分析一个ELF文件，怎么从中读取指令，怎么来进行后面的hand coded ？？？？

    在dyngen动态翻译技术中，还涉及到几个比较重要的地方，比如：

    （1）TBs，Translated Blocks

              qemu将TB定义为碰到下一个jump指令或修改CPU state的指令之前的所有代码称为一个TB  
    （2）寄存器分配

              target平台的寄存器被映射到host的固定寄存器或指定的内存地址

    （3）条件代码的优化

    （4）TB块以hash表的形式组织

    （5）mmap()系统调用仿真target的MMU

    （6）longjmp()实现异常仿真

    （7）异步轮询的方式实现中断的仿真

    至于（3）~（7）的具体实现方式，现在还比较模糊，希望与大家交流！！！！！！！！！

[qemu源码分析之五-- TCG动态翻译技术](http://blog.csdn.net/ustc_dylan/article/details/6868050)

1. TCG简单介绍

TCG（Tiny Code Generator）最早被用于C编译器的后端。在TCG相关的代码中，target指的是我们通常说的host，这一点需要注意，并不是我们理解的被仿真的平台。

2. TCG动态翻译技术的几个概念

（1）与dyngen一样，TCG的“function”与qemu的TBs（Translated Block）相对应，即以分支跳转指令结束的代码段。

（2）TCG中有三种变量：temporary， local temporary， global。这三种变量有着不同的生命周期，temporary变量的声明周期是TBs，local temporary变量的声明周期是functions，global变量的声明周期是所有的functions，类似C语言的全局变量。temporary和local temporary变量通常在function内定义，global变量通常在function外定义。全局变量通常被映射到某个内存地址或某个固定的寄存器。

3. TCG operations

就像dyngen动态翻译技术中的micro-operations一样，TCG也采用中间表示的形式(TCG instructions)，TCG至中间表示支持的三种变量有两种数据类型：32 bit整型和64 bit整型。另外，指针类型被实现为整型的别名。

TCG指令有固定的形式：

TCG 操作码 输出变量域，输入变量域，常量域

比较特殊的是call指令，其后只跟一个变量，同时作为输出和输入变量域。

举个例子：

add\_i32 t0, t1, t2 (t0 <- t1 + t2)

操作码 输出变量域 输入变量域 常量域（输入）

4. TCG是qemu的核心，主要实现了以下翻译流程：

guest binary instructions -> TCG IR -> host binary instructions TCG 定义了一组IR(intermediate representation),这些IR大致可以分为以下几类：

- Mov类操作: mov, movi, ...

- 逻辑操作: and, or, xor, shl, shr, ...

- 算术操作: add, sub, mul, div, ...

- 分支跳转操作: jmp, br, brcond

- 函数调用: call - 内存操作: ld, st

- QEMU的特殊操作: tb\_exit, goto\_tb, qemu\_ld/qemu\_st

这里仅对TCG中间表示做一下简单分类，至于每条TCG指令的具体用法，参见qemu源码tcg/readme。

5. TCG 动态翻译过程

前面也提到TCG主要实现以下翻译过程：

guest binary instructions -> TCG IR -> host binary instructions

在qemu源码中，target-ARCH/\* 定义了如何将guest binary instructions 反汇编成 TCG IR，tcg/ARCH 定义了如何將 TCG IR 翻译成 host binary instructions。

在下一篇文章中，将会从qemu源码的角度详细分析x86-->x86平台的TCG动态翻译技术的执行过程。

[qemu源码分析之Makefile](http://blog.csdn.net/ustc_dylan/article/details/6963256)

通过分析qemu的Makefile可以了解qemu代码的组织方式以及qemu功能模块的划分，一方面，有助于理解qemu源代码设计思路，另一方面，有助于根据需求裁剪qemu代码，生成精简的符合制定要求的qemu。  
    为了更好的理解qemu的Makefile的设计，对于不熟悉makefile语法规则的同学建议阅读下面两篇文章：  
    1. 《[Makefile常用函数分析](http://blog.chinaunix.net/link.php?url=http://blog.csdn.net%2Fustc_dylan%2Farticle%2Fdetails%2F6963248)》  
    2. 《跟我一起写makefile》  
  
----------------------------------------------------------------------------------------  
/\*qemu的Makefile的依赖目标之间的关系\*/  
p, li { white-space: pre-wrap; }

all:build-all                                           (1)

build-all:$(TOOLS) recurse-all                          (2)

recurse-all: $(SUBDIR\_RULES) $(ROMSUBDIR\_RULES)         (3)

p, li { white-space: pre-wrap;SUBDIR\_RULES=$(patsubst %,subdir-%, $(TARGET\_DIRS))     (4)

ROMSUBDIR\_RULES=$(patsubst %,romsubdir-%, $(ROMS))      (5)

subdir-%: $(GENERATED\_HEADERS)                          (6)

$(call quiet-command,$(MAKE) $(SUBDIR\_MAKEFLAGS) -C $\* V="$(V)" TARGET\_DIR="$\*/" all,)

romsubdir-%:                                            (7)

$(call quiet-command,$(MAKE) $(SUBDIR\_MAKEFLAGS) -C pc-bios/$\* V="$(V)" TARGET\_DIR="$\*/",)

上面即是qemu简化版的Makefile,下面就来详细分析下上面规则的含义。  
-------------------------------------------------------------------------------------------  
p, li { white-space: pre-wrap; }

SRC\_PATH=/home/src/qemu-0.15.1

TOOLS=qemu-ga qemu-nbd qemu-img qemu-io  /\*规则（2）中TOOLS的定义\*/

TARGET\_DIRS=i386-softmmu                 /\*规则（4）中TARGET\_DIRS的定义\*/

ROMS=optionrom                           /\*规则（5）中ROMSUBDIR\_RULES的定义\*/

--------------------------------------------------------------------------------------------

上述规则中最重要的规则就是两个函数的调用：p, li { white-space: pre-wrap; }

subdir-%: $(GENERATED\_HEADERS)

$(call quiet-command,$(MAKE) $(SUBDIR\_MAKEFLAGS) -C $\* V="$(V)" TARGET\_DIR="$\*/" all,)

上面规则等价于：

make --no-print-directory -C i386-softmmu V="" TARGET\_DIR="i386-softmmu/" all

回想make -C的常用格式:

make -C path 参数

则上面规则的含义为：执行i386-softmmu下的Makefile中的依赖目标all, 传递的参数是V=""和TARGET\_DIR="i386-softmmu/"

romsubdir-%:

$(call quiet-command,$(MAKE) $(SUBDIR\_MAKEFLAGS) -C pc-bios/$\* V="$(V)" TARGET\_DIR="$\*/",)

与subdir-i386-softmmu一样，上面规则的含义就是：执行pc-bios/下的所有依赖目标，传递的参数为 V="$(V)" TARGET\_DIR="$\*/"，需要注意的是这里的$\*代表的是不包含扩展名的目标文件的名称。

自此，Makefile就进入了i386-softmmu和pc-bios两个目录下，执行相对应的Makfile

---------------------------------------------------------------------------------------------

从i386-softmmu/Makefile中能够清晰的找出哪些代码完成了qemu哪些功能模块的仿真。

1. cpu功能模块的仿真

1. libobj-y = exec.o translate-all.o cpu-exec.o translate.o
2. libobj-y += tcg/tcg.o
3. libobj-y += fpu/softfloat.o
4. libobj-y += op\_helper.o helper.o
5. ifeq ($(TARGET\_BASE\_ARCH), i386)
6. libobj-y += cpuid.o
7. endif
8. libobj-$(CONFIG\_NEED\_MMU) += mmu.o
9. libobj-$(TARGET\_ARM) += neon\_helper.o iwmmxt\_helper.o
10. libobj-y += disas.o

2. 硬件设备的仿真

1. # Hardware support
2. obj-i386-y += vga.o
3. obj-i386-y += mc146818rtc.o i8259.o pc.o
4. obj-i386-y += cirrus\_vga.o sga.o apic.o ioapic.o piix\_pci.o
5. obj-i386-y += vmport.o
6. obj-i386-y += device-hotplug.o pci-hotplug.o smbios.o wdt\_ib700.o
7. obj-i386-y += debugcon.o multiboot.o
8. obj-i386-y += pc\_piix.o
9. obj-i386-$(CONFIG\_KVM) += kvmclock.o
10. obj-i386-$(CONFIG\_SPICE) += qxl.o qxl-logger.o qxl-render.o

3. qemu中平台无关的代码（平台共用代码）

1. translate.o: translate.c cpu.h
2. translate-all.o: translate-all.c cpu.h
3. tcg/tcg.o: cpu.h
4. # HELPER\_CFLAGS is used for all the code compiled with static register
5. # variables
6. op\_helper.o user-exec.o: QEMU\_CFLAGS += $(HELPER\_CFLAGS)
7. # Note: this is a workaround. The real fix is to avoid compiling
8. # cpu\_signal\_handler() in user-exec.c.
9. signal.o: QEMU\_CFLAGS += $(HELPER\_CFLAGS)

4. kvm相关代码

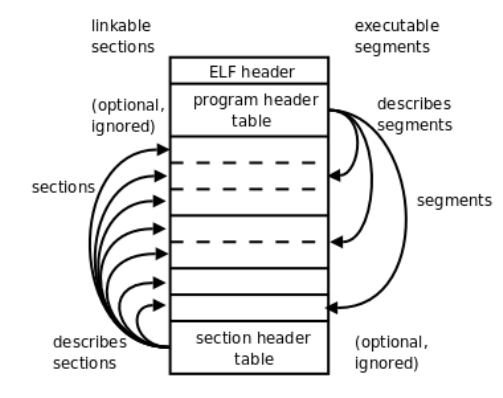
1. obj-$(CONFIG\_KVM) += kvm.o kvm-all.o
2. obj-$(CONFIG\_NO\_KVM) += kvm-stub.o

根据./configure时不能模块的使能，此Makefile还有其他功能模块，这些功能模块对应的代码在这里一目了然，能够有效的提高对qemu代码理解和裁剪的效率。

[qemu中ELF文件的加载](http://blog.csdn.net/ustc_dylan/article/details/7068190)

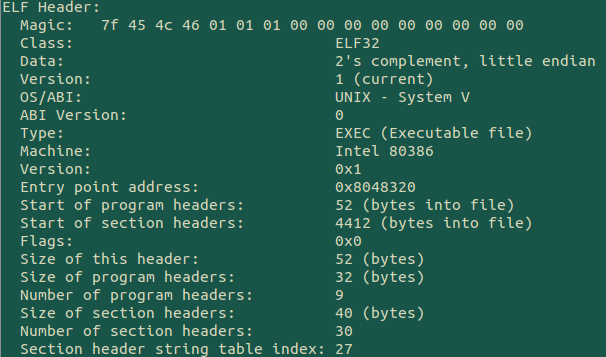
前段时间分析了qemu中ELF文件的加载过程，个人感觉通过这个分析不但可以加深对ELF文件格式的理解，而且能够从侧面了解操作系统加载器的工作过程。  
 **一、ELF相关的背景知识**  
1. ELF格式文件相关概念  
ELF格式文件主要包括以下三种类型的文件：

* 可重定位的目标文件(.o文件)      --> 用于链接生成可执行文件或动态链接库文件(.so)
* 可执行文件                      --> 进程映像文件
* 共享库(.so文件)                 --> 用于链接

    从链接和执行的角度来讲，ELF文件存在两种视图：链接视图和执行视图。为了区分两种视图，只需记住链接视图由多个section组成，而执行视图由多个segment组成即可。另外，section是程序员可见的，是给链接器使用的概念，汇编文件中通常会显示的定义.text，.data等section，相反，segment是程序员不可见的，是给加载器使用的概念。下图形象的描述了ELF文件两种不同的视图的结构以及二者之间的联系。  
[](http://blog.chinaunix.net/attachment/201112/13/20940095_1323775969amxX.png)  
    二者之间的联系在于：一个segment包含一个或多个section。  
    注意：Section Header Table和Program Header Table并不是一定要位于文件的开头和结尾,其位置由ELF Header指出,上图这么画只是为了清晰。-- ELF文件每个部分的详细介绍参见《ELF 文件格式分析》。[[http://blog.chinaunix.net/blog/image/attachicons/common.gif](http://blog.chinaunix.net/attachment/attach/20/94/00/9520940095749b425507abb1ce08ec3fde95896db3.pdf) TN05.ELF.Format.Summary.pdf](http://blog.chinaunix.net/attachment/attach/20/94/00/9520940095749b425507abb1ce08ec3fde95896db3.pdf)    
  
2. ELF文件主要数据结构  
    上面讲了ELF的相关概念，但是要想用计算机语言（C语言）来实现，必须对应相应的数据结构。linux下通过三个数据结构描述了ELF文件的相关概念。  
(1) ELF Header  
ELF Header描述了体系结构和操作系统等基本信息,并指出Section Header Table和Program Header Table在文件中的什么位置，每个成员的解释参见注释及附件。

**[cpp]** [view plaincopy](http://blog.csdn.net/ustc_dylan/article/details/7068190)

1. #define EI\_NIDENT 16
2. **typedef** **struct**{
3. /\*ELF的一些标识信息，固定值\*/
4. unsigned **char** e\_ident[EI\_NIDENT];
5. /\*目标文件类型：1-可重定位文件，2-可执行文件，3-共享目标文件等\*/
6. Elf32\_Half e\_type;
7. /\*文件的目标体系结构类型：3-intel 80386\*/
8. Elf32\_Half e\_machine;
9. /\*目标文件版本：1-当前版本\*/
10. Elf32\_Word e\_version;
11. /\*程序入口的虚拟地址，如果没有入口，可为0\*/
12. Elf32\_Addr e\_entry;
13. /\*程序头表(segment header table)的偏移量，如果没有，可为0\*/
14. Elf32\_Off e\_phoff;
15. /\*节区头表(section header table)的偏移量，没有可为0\*/
16. Elf32\_Off e\_shoff;
17. /\*与文件相关的，特定于处理器的标志\*/
18. Elf32\_Word e\_flags;
19. /\*ELF头部的大小，单位字节\*/
20. Elf32\_Half e\_ehsize;
21. /\*程序头表每个表项的大小，单位字节\*/
22. Elf32\_Half e\_phentsize;
23. /\*程序头表表项的个数\*/
24. Elf32\_Half e\_phnum;
25. /\*节区头表每个表项的大小，单位字节\*/
26. Elf32\_Half e\_shentsize;
27. /\*节区头表表项的数目\*/
28. Elf32\_Half e\_shnum;
29. /\*\*/
30. Elf32\_Half e\_shstrndx;
31. }Elf32\_Ehdr;

下面通过一个具体实例来说明ELF header中每个数据成员对应的值，下面是hello world的ELF文件头，在linux下可以通过"readelf -h ELF文件名"来获得。  
[](http://blog.chinaunix.net/attachment/201112/13/20940095_1323777039e68I.png)  
ELF Header用数据结构Elf32\_Ehdr来表示，描述了操作系统是UNIX,体系结构是80386。Section Header Table中有30个Section Header,从文件地址4412开始,每个Section Header占40字节,Segment Header Table中有9个segment，每个segment header占32个字节，此ELF文件的类型是可执行文件(EXEC)，入口地址是0x8048320。  
  
(2) Section Header Table Entry

从ELF Header中可知，每个ELF文件有个Section Header Table，其中每一个表项对应一个section，由数据结构Elf32\_Shdr来描述，每个成员的含义参见注释及附件。在linux下可以通过"readelf -S ELF文件名"来查看。

**[cpp]** [view plaincopy](http://blog.csdn.net/ustc_dylan/article/details/7068190)

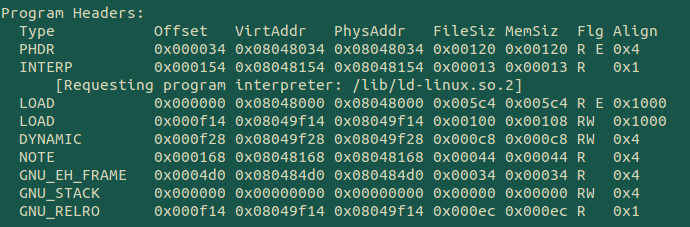
1. **typedef** **struct**{
2. /\*节区名称\*/
3. Elf32\_Word sh\_name;
4. /\*节区类型：PROGBITS-程序定义的信息，NOBITS-不占用文件空间(bss),REL-重定位表项\*/
5. Elf32\_Word sh\_type;
6. /\*每一bit位代表一种信息，表示节区内的内容是否可以修改，是否可执行等信息\*/
7. Elf32\_Word sh\_flags;
8. /\*如果节区将出现在进程的内存影响中，此成员给出节区的第一个字节应处的位置\*/
9. Elf32\_Addr sh\_addr;
10. /\*节区的第一个字节与文件头之间的偏移\*/
11. Elf32\_Off sh\_offset;
12. /\*节区的长度，单位字节，NOBITS虽然这个值非0但不占文件中的空间\*/
13. Elf32\_Word sh\_size;
14. /\*节区头部表索引链接\*/
15. Elf32\_Word sh\_link;
16. /\*节区附加信息\*/
17. Elf32\_Word sh\_info;
18. /\*节区带有地址对齐的约束\*/
19. Elf32\_Word sh\_addralign;
20. /\*某些节区中包含固定大小的项目，如符号表，那么这个成员给出其固定大小\*/
21. Elf32\_Word sh\_entsize;
22. }Elf32\_Shdr;

(3) Program Header Table Entry

从ELF Header中可知，每个ELF文件有个Program Header Table，其中每一个表项对应一个segment，由数据结构Elf32\_phdr来描述，每个成员的含义参见注释及附件。在linux下可以通过"readelf -l ELF文件名"来查看。

**[cpp]** [view plaincopy](http://blog.csdn.net/ustc_dylan/article/details/7068190)

1. **typedef** **struct**
2. {
3. /\*segment的类型：PT\_LOAD= 1 可加载的段\*/
4. Elf32\_Word p\_type;
5. /\*从文件头到该段第一个字节的偏移\*/
6. Elf32\_Off p\_offset;
7. /\*该段第一个字节被放到内存中的虚拟地址\*/
8. Elf32\_Addr p\_vaddr;
9. /\*在linux中这个成员没有任何意义，值与p\_vaddr相同\*/
10. Elf32\_Addr p\_paddr;
11. /\*该段在文件映像中所占的字节数\*/
12. Elf32\_Word p\_filesz;
13. /\*该段在内存映像中占用的字节数\*/
14. Elf32\_Word p\_memsz;
15. /\*段标志\*/
16. Elf32\_Word p\_flags;
17. /\*p\_vaddr是否对齐\*/
18. Elf32\_Word p\_align;
19. } Elf32\_phdr;

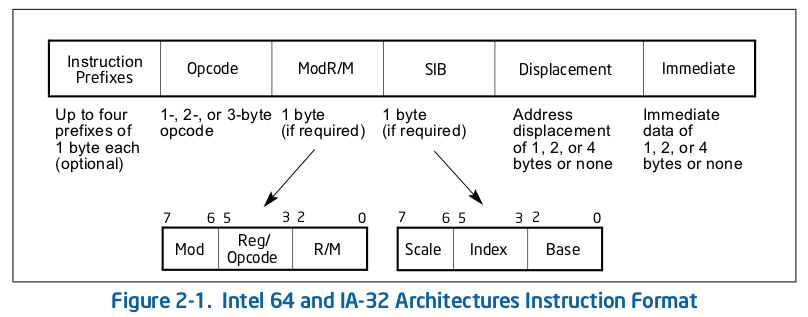
二、qemu中ELF文件的加载过程  
    在了解了ELF文件的基本结构之后，大体可以想到ELF文件的加载过程就是一个查表的过程，即通过ELF Header得到ELF文件的基本信息-Section Header Table和program Header Table,然后再根据Section Header Table和program Header Table的信息加载ELF文件中的相应部分。上面也提到过，section是从链接器的角度来讲的概念，所以，ELF文件的加载过程中，只有segment是有效的，加载器根据program Header Table中的信息来负责ELF文件的加载。  
    首先，从感性上认识一下segment，还是以上面的hello world为例，其对应的program header table如下。  
[](http://blog.chinaunix.net/attachment/201112/13/20940095_1323778084m588.png)  
    第一列type即每个segment的类型，每个类型的具体含义参见附件。通常我们之关心程序的代码段(.text section)和数据段(.date section)，这两个section组成LOAD类型的segment。  
    Offset：当前segment加载到的地址的偏移  
    VirAddr：当前segment加载到的虚拟地址  
    PhysAddr：当前segment加载到的物理地址（x86平台上，此值没有意义，并不指物理地址）  
    FileSiz：当前segment在ELF文件中的偏移  
    MemSiz:当前segment在内存页中的偏移  
    Flg：segment的权限，R-可读，W-可写， E-可执行  
    Align：x86平台内存页面的大小  
  
    在了解了segment的相关信息后，分析下qemu代码中ELF文件的加载过程，印证下上面提到的ELF文件的加载的思想。

**[cpp]** [view plaincopy](http://blog.csdn.net/ustc_dylan/article/details/7068190)

1. ret = loader\_exec(filename, target\_argv, target\_environ, regs,info,&bprm);
3. filename：要加载的ELF文件的名称
4. target\_argv：qemu运行的参数，在这里即hello（hello是生成的可执行文件名， $qemu hello）
5. target\_environ：执行qemu的shell的环境变量
6. regs,info,bprm是ELF文件加载过程中涉及的三个重要数据结构，下面会详细分析。
8. er\_exec函数的功能及含义参见代码注释。
10. **int** loader\_exec(**const** **char**\* filename, **char**\*\* argv, **char**\*\* envp,
11. **struct** target\_pt\_regs \* regs, **struct** image\_info\*infop,
12. **struct** linux\_binprm \*bprm)
13. {
14. **int** retval;
15. **int** i;
16. bprm->p= TARGET\_PAGE\_SIZE\*MAX\_ARG\_PAGES-**sizeof**(unsignedint);    /\*MAX\_ARG\_PAGES= 33\*/
17. memset(bprm->page, 0, **sizeof**(bprm->page));
18. retval = open(filename, O\_RDONLY);                                /\*返回打开文件的fd\*/
19. **if** (retval< 0)
20. **return** retval;
21. bprm->fd= retval;
22. bprm->filename= (**char** \*)filename;
23. bprm->argc= count(argv);
24. bprm->argv= argv;
25. bprm->envc= count(envp);
26. bprm->envp= envp;
27. /\*1. 要加载文件的属性判断：是否常规文件，是否可执行文件，是否ELF文件； 2. 读取ELF文件的前1024个字节\*/
28. retval = prepare\_binprm(bprm);
29. **if**(retval>=0){            /\*prepare\_binrpm函数已经读出了目标文件的前1024个字节，先判断下这个文件是否是ELF文件，即前4个字节\*/
30. **if** (bprm->buf[0]== 0x7f
31. && bprm->buf[1]== 'E'
32. && bprm->buf[2]== 'L'
33. && bprm->buf[3]== 'F'){
34. retval = load\_elf\_binary(bprm, regs, infop);
35. #if defined(TARGET\_HAS\_BFLT)
36. } elseif (bprm->buf[0]== 'b'
37. && bprm->buf[1]== 'F'
38. && bprm->buf[2]== 'L'
39. && bprm->buf[3]== 'T'){
40. retval = load\_flt\_binary(bprm,regs,infop);
41. #endif
42. } **else**{
43. fprintf(stderr,"Unknown binary format\n");
44. **return** -1;
45. }
46. }
47. **if**(retval>=0){
48. /\* success. Initialize important registers\*/
49. do\_init\_thread(regs, infop);
50. **return** retval;
51. }
52. /\* Something went wrong, return the inodeand free the argument pages\*/
53. **for** (i=0; i<MAX\_ARG\_PAGES; i++){
54. g\_free(bprm->page[i]);
55. }
56. **return**(retval);
57. }
59. **int** load\_elf\_binary(**struct** linux\_binprm\* bprm, **struct** target\_pt\_regs\* regs,
60. **struct** image\_info \* info)
61. {
62. **struct** image\_info interp\_info;
63. **struct** elfhdr elf\_ex;
64. **char** \*elf\_interpreter = NULL;
65. info->start\_mmap= (abi\_ulong)ELF\_START\_MMAP;        /\*ELF\_START\_MMAP= 0x80000000\*/
66. info->mmap= 0;
67. info->rss= 0;
68. /\*主要工作就是初始化info，申请进程虚拟地址空间，将ELF文件映射到这段虚拟地址空间上\*/
69. load\_elf\_image(bprm->filename, bprm->fd, info,
70. &elf\_interpreter, bprm->buf);
72. ... ... ... ...

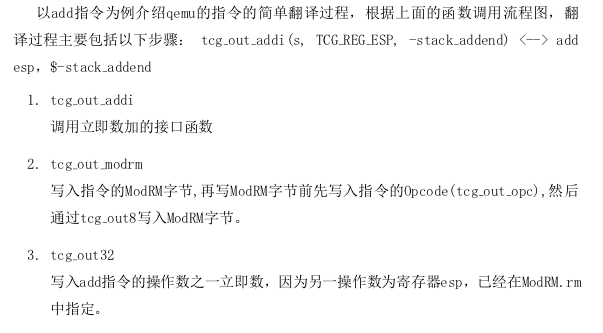
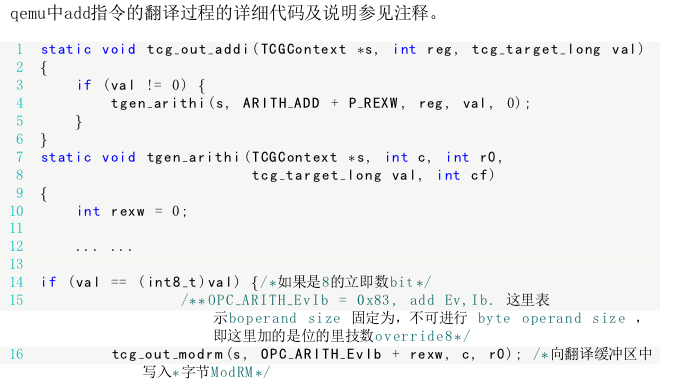
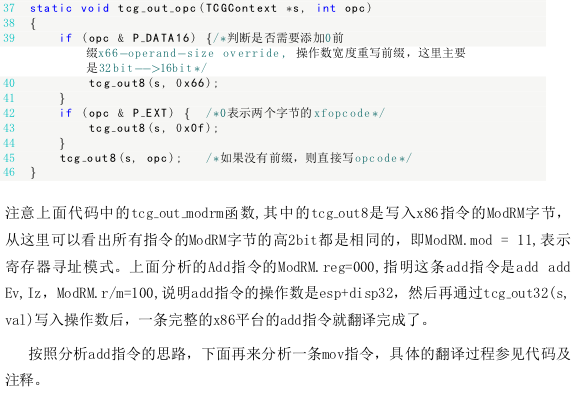
75. **return** 0;
76. }
78. **static** **void** load\_elf\_image(**const** **char**\*image\_name,**int** image\_fd,
79. **struct** image\_info \*info, **char**\*\*pinterp\_name,
80. **char** bprm\_buf[BPRM\_BUF\_SIZE])
81. {
82. **struct** elfhdr \*ehdr = (**struct** elfhdr \*)bprm\_buf;
83. **struct** elf\_phdr \*phdr;
84. abi\_ulong load\_addr, load\_bias, loaddr, hiaddr,error;
85. **int** i, retval;
86. **const** **char** \*errmsg;
87. /\* First of all, some simple consistency checks\*/
88. errmsg = "Invalid ELF image for this architecture";
89. **if** (!elf\_check\_ident(ehdr)){/\*ELF头检查\*/
90. **goto** exit\_errmsg;
91. }
92. bswap\_ehdr(ehdr);    /\*当前为空，是不是主机和目标机大小尾端不一致时才会swap\*/
93. **if** (!elf\_check\_ehdr(ehdr)){
94. **goto** exit\_errmsg;
95. }
96. /\*下面的代码即读出ELF文件的程序头表，首先判断下是否已经被完全读出\*/
97. i = ehdr->e\_phnum\* **sizeof**(**struct** elf\_phdr);    /\*program header 表的大小\*/
98. **if** (ehdr->e\_phoff+ i <= BPRM\_BUF\_SIZE){
99. phdr = (**struct** elf\_phdr \*)(bprm\_buf+ ehdr->e\_phoff);
100. } **else**{
101. phdr = (**struct** elf\_phdr \*) alloca(i);    /\*申请i个程序头部\*/
102. retval = pread(image\_fd, phdr, i, ehdr->e\_phoff);    /\*从文件image\_id的偏移为ehdr->e\_phoff处读取i个字节到phdr中，即phdr存放program header\*/
103. **if** (retval!= i){
104. **goto** exit\_read;
105. }
106. }
107. bswap\_phdr(phdr, ehdr->e\_phnum);
108. #ifdef CONFIG\_USE\_FDPIC
109. info->nsegs= 0;
110. info->pt\_dynamic\_addr= 0;
111. #endif
112. /\* Find the maximum size of the imageand allocate an appropriate
113. amount of memory to handle that.\*/
114. loaddr = -1, hiaddr= 0;
115. **for** (i= 0; i < ehdr->e\_phnum;++i){/\*遍历每一个program header\*/
116. **if** (phdr[i].p\_type== PT\_LOAD){
117. abi\_ulong a = phdr[i].p\_vaddr;
118. **if** (a < loaddr){        /\*loaddr= -1而且是unsigned 类型的，所以loaddr是个很大的数\*/
119. loaddr = a;            /\*loaddr记录segment的起始地址\*/
120. }
121. a += phdr[i].p\_memsz;    /\*这个segment在内存中的偏移地址\*/
122. **if** (a > hiaddr){        /\*hiaddr记录segment的结束地址\*/
123. hiaddr = a;
124. }
125. #ifdef CONFIG\_USE\_FDPIC
126. ++info->nsegs;
127. #endif
128. }
129. }
130. load\_addr = loaddr;        /\*计算出来的需要加载的起始地址\*/
131. **if** (ehdr->e\_type== ET\_DYN){    /\*共享目标文件(.so)\*/
132. /\* The image indicates that it can be loaded anywhere. Find a
133. location that can hold the memoryspace required.If the
134. image is pre-linked, LOADDR will be non-zero. Since wedo
135. not supply MAP\_FIXED here we'll use that addressif and
136. only if it remains available.\*/
137. load\_addr = target\_mmap(loaddr, hiaddr- loaddr, PROT\_NONE,
138. MAP\_PRIVATE | MAP\_ANON| MAP\_NORESERVE,
139. -1, 0);
140. **if** (load\_addr== -1) {
141. **goto** exit\_perror;
142. }
143. } elseif (pinterp\_name!= NULL) {
144. /\* Thisis the main executable. Make sure that the low
145. address does not conflict with MMAP\_MIN\_ADDRor the
146. QEMU application itself. \*/
147. probe\_guest\_base(image\_name, loaddr, hiaddr);
148. }
149. load\_bias = load\_addr - loaddr;
150. #ifdef CONFIG\_USE\_FDPIC
151. {
152. **struct** elf32\_fdpic\_loadseg \*loadsegs= info->loadsegs=
153. g\_malloc(**sizeof**(\*loadsegs)\* info->nsegs);
154. **for** (i= 0; i < ehdr->e\_phnum;++i){
155. **switch** (phdr[i].p\_type){
156. **case** PT\_DYNAMIC:
157. info->pt\_dynamic\_addr= phdr[i].p\_vaddr+ load\_bias;
158. **break**;
159. **case** PT\_LOAD:
160. loadsegs->addr= phdr[i].p\_vaddr+ load\_bias;
161. loadsegs->p\_vaddr= phdr[i].p\_vaddr;
162. loadsegs->p\_memsz= phdr[i].p\_memsz;
163. ++loadsegs;
164. **break**;
165. }
166. }
167. }
168. #endif
169. info->load\_bias= load\_bias;    /\*真实的加载地址和计算出来(读ELF头信息)的加载地址之差\*/
170. info->load\_addr= load\_addr;    /\*真实的加载地址\*/
171. info->entry= ehdr->e\_entry+ load\_bias;    /\*重新调整下程序的入口\*/
172. info->start\_code= -1;
173. info->end\_code= 0;
174. info->start\_data= -1;
175. info->end\_data= 0;
176. info->brk= 0;
177. **for** (i= 0; i < ehdr->e\_phnum; i++){
178. **struct** elf\_phdr \*eppnt = phdr + i;
179. **if** (eppnt->p\_type== PT\_LOAD){
180. abi\_ulong vaddr, vaddr\_po, vaddr\_ps, vaddr\_ef, vaddr\_em;
181. **int** elf\_prot = 0;
182. /\*记录PT\_LOAD类型segment的权限：读/写/可执行\*/
183. **if** (eppnt->p\_flags& PF\_R) elf\_prot= PROT\_READ;
184. **if** (eppnt->p\_flags& PF\_W) elf\_prot|= PROT\_WRITE;
185. **if** (eppnt->p\_flags& PF\_X) elf\_prot|= PROT\_EXEC;
186. vaddr = load\_bias + eppnt->p\_vaddr;
187. vaddr\_po = TARGET\_ELF\_PAGEOFFSET(vaddr);/\*((vaddr)& ((1<< 12)-1))，目的是取页内偏移\*/
188. vaddr\_ps = TARGET\_ELF\_PAGESTART(vaddr);    /\*((vaddr)& ~(unsigned long)((1<< 12)-1))，向下页对齐，目的取页对齐的地址\*/
189. /\*将ELF文件映射到进程地址空间中\*/
190. error= target\_mmap(vaddr\_ps, eppnt->p\_filesz+ vaddr\_po,    /\*映射的时候从页内偏移vaddr\_po开始映射，即保持原来的偏移量\*/
191. elf\_prot, MAP\_PRIVATE| MAP\_FIXED,
192. image\_fd, eppnt->p\_offset- vaddr\_po);
193. **if** (error ==-1) {
194. **goto** exit\_perror;
195. }
196. vaddr\_ef = vaddr + eppnt->p\_filesz;
197. vaddr\_em = vaddr + eppnt->p\_memsz;
198. /\*If the load segment requests extra zeros (e.g. bss), map it.\*/
199. **if** (vaddr\_ef < vaddr\_em){
200. zero\_bss(vaddr\_ef, vaddr\_em, elf\_prot);
201. }
202. /\* Find the full program boundaries.\*/
203. **if** (elf\_prot & PROT\_EXEC){
204. **if** (vaddr < info->start\_code){
205. info->start\_code= vaddr;    /\*代码段的起始虚拟地址（页对齐的地址）\*/
206. }
207. **if** (vaddr\_ef > info->end\_code){
208. info->end\_code= vaddr\_ef;    /\*代码段的结束虚拟地址（页对齐的地址）\*/
209. }
210. }
211. **if** (elf\_prot & PROT\_WRITE){
212. **if** (vaddr < info->start\_data){
213. info->start\_data= vaddr;    /\*数据段的起始虚拟地址\*/
214. }
215. **if** (vaddr\_ef > info->end\_data){
216. info->end\_data= vaddr\_ef;    /\*数据段的起始虚拟地址（包括bss段的大小）\*/
217. }
218. **if** (vaddr\_em > info->brk){
219. info->brk= vaddr\_em;    /\*程序内存映像的顶端（代码段+数据段+bss段）\*/
220. }
221. }
222. } elseif (eppnt->p\_type== PT\_INTERP&& pinterp\_name){/\*内部解释程序名称：/lib/ld-linux.so.2\*/
223. **char** \*interp\_name;
224. **if** (\*pinterp\_name){
225. errmsg = "Multiple PT\_INTERP entries";
226. **goto** exit\_errmsg;
227. }
228. interp\_name = malloc(eppnt->p\_filesz);
229. **if** (!interp\_name){
230. **goto** exit\_perror;
231. }
232. **if** (eppnt->p\_offset+ eppnt->p\_filesz<= BPRM\_BUF\_SIZE){
233. memcpy(interp\_name, bprm\_buf+ eppnt->p\_offset,
234. eppnt->p\_filesz);
235. } **else** {
236. retval = pread(image\_fd, interp\_name, eppnt->p\_filesz,
237. eppnt->p\_offset);
238. **if** (retval != eppnt->p\_filesz){
239. **goto** exit\_perror;
240. }
241. }
242. **if** (interp\_name[eppnt->p\_filesz- 1] != 0){
243. errmsg = "Invalid PT\_INTERP entry";
244. **goto** exit\_errmsg;
245. }
246. \*pinterp\_name = interp\_name;
247. }
248. }
249. **if** (info->end\_data== 0){
250. info->start\_data= info->end\_code;
251. info->end\_data= info->end\_code;
252. info->brk= info->end\_code;
253. }
254. **if** (qemu\_log\_enabled()){
255. load\_symbols(ehdr, image\_fd, load\_bias);
256. }
257. close(image\_fd);
258. **return**;
259. exit\_read:
260. **if** (retval>= 0){
261. errmsg = "Incomplete read of file header";
262. **goto** exit\_errmsg;
263. }
264. exit\_perror:
265. errmsg = strerror(errno);
266. exit\_errmsg:
267. fprintf(stderr,"%s: %s\n", image\_name, errmsg);
268. exit(-1);
269. }

[qemu指令翻译过程（1）-- ADD指令](http://blog.csdn.net/ustc_dylan/article/details/7085200)

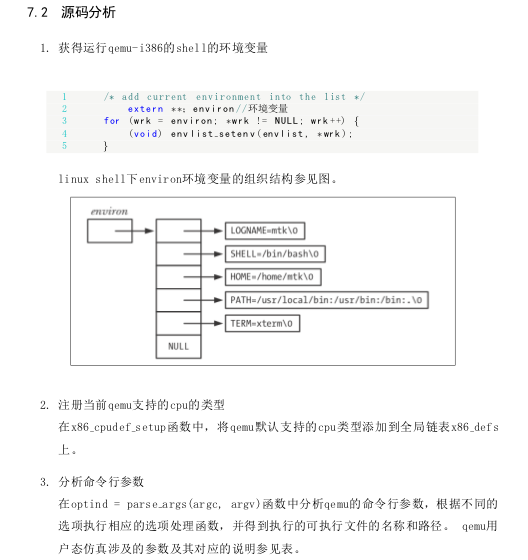
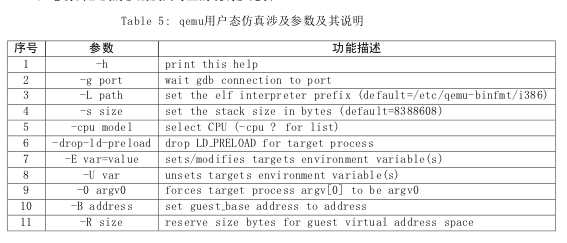
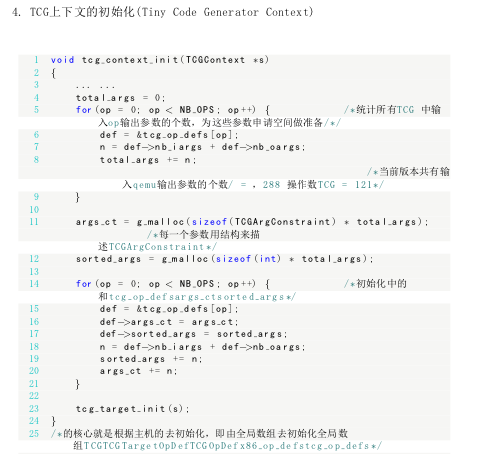
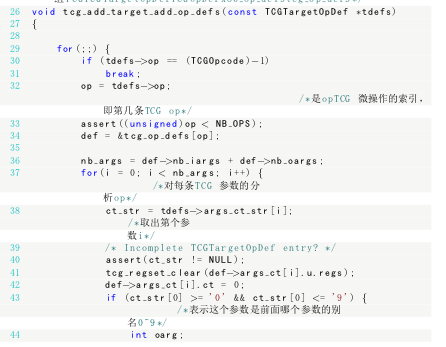
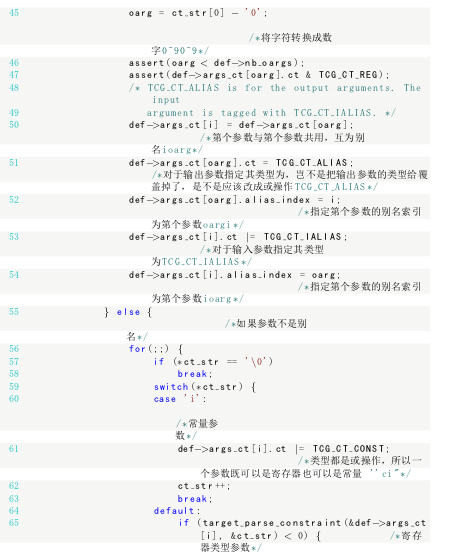
本文通过qemu中add指令的翻译过程来加深了解向x86通用指令的结构以及qemu指令翻译的基本通过。  
  
**1. x86通用指令的基本结构(摘自intel 手册)**  
[](http://blog.chinaunix.net/attachment/201112/19/20940095_1324289672dG5G.png)

x86/x64 通用指令编码的核心是：**Opcode**，**ModRM** 以及 **SIB**

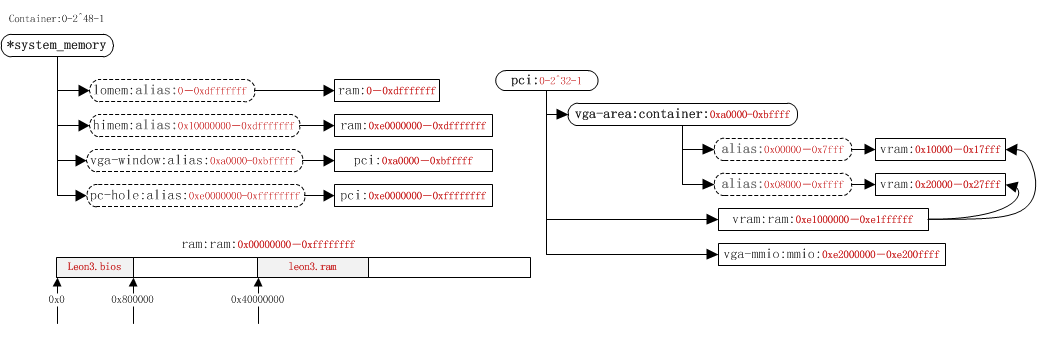
* Opcode 提供指令的操作码
* ModRM 提供操作数的寻址，以及对 opcode 进行辅助补充
* SIB 提供更丰富的 memory 寻址

**2. qemu中add指令的翻译过程**(摘自我的笔记《qemu源码分析》)  
[](http://blog.chinaunix.net/attachment/201112/19/20940095_1324290465w8VU.png)  
[](http://blog.chinaunix.net/attachment/201112/19/20940095_13242903710PEF.png)  
  
[](http://blog.chinaunix.net/attachment/201112/19/20940095_13242903572V2e.png)  
  
  
[](http://blog.chinaunix.net/attachment/201112/19/20940095_1324290403m485.png)  
[http://blog.chinaunix.net/attachment/201112/19/20940095_1324290417eHEo.png](http://blog.chinaunix.net/attachment/201112/19/20940095_1324290417eHEo.png)  
  
  
至于，如何分析操作码和操作数，最笨的方法就是参考intel手册，不过，还是有许多技巧的，在以后的blog中，将会详细讲解如何分析x86指令集的操作码和操作数（熟能生巧!）。

[qemu用户态仿真代码分析](http://blog.csdn.net/ustc_dylan/article/details/7100439)

**1、 基本理论分析**  
    qemu实现了两种模式的仿真：qemu系统仿真和qemu用户程序仿真。简单的理解，qemu系统仿真可以理解为仿真了一台PC，可以在上面安装OS，qemu用户程序仿真仅仅是对CPU的仿真，即可以运行不同体系结构的应用程序，可以理解为仿真了应用程序的运行环境。  
    下面通过实际的代码来分析qemu用户程序仿真的实现过程，在此之前，首先思考下可执行文件的加载执行过程，理解了这个过程有助于对qemu用户程序仿真实现的理解。  
    以ELF文件为例，其执行过程为：  
    加载器分析ELF文件的header，按照一定的规则将ELF文件的指定segment加载到进程地址空间的某个地址处，linux下通常为0x804800,设置应用程序执行需要的相关条件，如环境变量，需要的其他动态链接库等，一切准备工作都完成后，将cs：ip的值设置为ELF文件的入口即可。  
    从本质上来讲，qemu要实现用户程序的仿真，也无非做些类似的工作。  
  
2、 详细代码分析  
 [](http://blog.chinaunix.net/attachment/201112/24/20940095_1324736204Npp8.png)  
[](http://blog.chinaunix.net/attachment/201112/24/20940095_13247362205JtQ.png)  
[](http://blog.chinaunix.net/attachment/201112/24/20940095_1324736236Cw19.png)  
[](http://blog.chinaunix.net/attachment/201112/24/20940095_13247362475MXO.png)  
[](http://blog.chinaunix.net/attachment/201112/24/20940095_1324736255wyXJ.png)  
未完，待续 ,.. ...

[qemu 内存模型（内存仿真概念）](http://blog.csdn.net/ustc_dylan/article/details/7326900)

qemu内存API仿真了qemu的内存,IO总线以及对应的控制器,主要包括以下部分的仿真:  
    • 常规内存  
    • IO映射内存(MMIO)  
    • 内存控制器(将物理内存动态的映射到不同的虚拟地址空间)  
  
qemu内存模型主要包括以下功能:  
    • 跟踪目标机内存的变化  
    • 为kvm建立共享内存(coalesced memory)  
    • 为kvm建立ioeventfd regions  
    qemu的内存以MemoryRegion对象为单位被组织成无环的树型结构,树的根是从CPU的角度(the system bus)可见的内存(system memory),树中的节点表示其他总线,内存控制器以及被重新映射过的内存区域,叶子节点表示真正的RAM Regions和MMIO Regions。qemu中包含四种类型的Memory Regions,通过c数据结构struct MemoryRegion来表示:  
    (1) RAM Region:目标机(GM)可用的主机(HM)上的一段虚拟地址空间  
    (2) MMIO Region:注册了read和write回调函数(callbacks functions)的一段目标机(HM)地址空间,对这段空间的读写操作将会调用主机上的回调函数  
    (3) Container:多个Memory Regions的集合,每个MR在Container中有不同的offset  
    (4) Alias:某个MR的subsection,alias类型的MR可以指向任何其他类型的MR  
     Memory Regions的name通过每个MR的构造函数进行赋值,对于大多数的MR来说,其name仅仅用作调试只用,但有时也用来定位在线迁移的内存。每个MR通过构造函数memory\_region\_init\*()来创建,并通过析构函数memory\_region\_destrory()来销毁,然后,通过memory\_region\_add\_subregion()将其添加到目标系统的地址空间中,并通过memory\_region\_del\_subregion()从地址空间中删除,另外,每个MR的属性在任何地方都  
可以被改变。通常来说,不同的MR不会重叠,但是有时候,MR的重叠是很有用的,目标系统可用通过memory\_region\_add\_subregion\_overlap()允许同一个container中的两个MR的地址空间重叠,重叠的MR具有优先级的属性(priority),用来标识当前哪个MR是可见的。  
    当目标系统访问某个地址空间时,qemu内存管理系统按照如下规则选择一个MR:  
    • 从根节点按照降序的优先级进行匹配  
    • 如果当前的MR是叶子节点,搜索过程终止  
    • 如果当前MR是Container,相同的算法在Container中搜索  
    • 如果当前MR是Alias,搜索从Alias指向的MR继续进行  
[](http://blog.chinaunix.net/attachment/201203/6/20940095_1331043305chhA.png)  
    上图是简单的PC内存映射图,4G的RAM地址空间通过两个Alias MR被映射到目标系统的地址空间中,其中,lomem采用一一映射共映射了4G地址空间的前3.5G,himem映射到剩下的0.5G的地址空间(上图中被称为pci-hole)。内存控制器将640k-768k的RAM地址重新映射到PCI地址空间,命名为vga-window,并且比原来RAM中的这段地址空间有更高的优先级,保证了访问这段地址空间是访问的PCI地址空间中的这段地址空间。  
上文中提到,只有system memory管理的地址空间是CPU可见的,pci地址空间并不是system memory的孩子节点,通过创建了vga-window和pci-hole两个PCI地址空间中两个子空间的别名的方式,使得PCI地址空间中的部分Region对CPU是可见的。Memory Region的属性包括:read-only,dirty logging,coalesced mmio,ioeventfd等,MMIO类型的MR提供了read()和write()两个回调函数,另外还附加了一些限制条件用来控制对两个回调函数的调用。