### QEMU的128位算法集合（基于QEMU2.0.0）

QEMU的128位算法集合，通常用于QEMU中GPA的地址运算，这里总结如下

|  |
| --- |
| /\*  \* 128位数据结构，由64bit组成，注意这是有符号数，所以高部分最多用63bit，表示127bit的数字  \*/ struct Int128 {     uint64\_t lo;     int64\_t hi; }; |

|  |  |
| --- | --- |
| **函数** | **意义** |
| Int128 int128\_make64(uint64\_t a) | 将 64BIT数字转换为 128bit 的格式, 放在128bit的低 64 为 |
| uint64\_t int128\_get64(Int128 a) | 得到 64bit 的值，也就是取得低 64位 |
| Int128 int128\_zero(void) | 0 |
| Int128 int128\_one(void) | 1 |
| Int128 int128\_and(Int128 a, Int128 b) | a & b |
| Int128 int128\_rshift(Int128 a, int n) | a >> n |
| Int128 int128\_add(Int128 a, Int128 b) | a + b |
| Int128 int128\_neg(Int128 a) | -a |
| Int128 int128\_sub(Int128 a, Int128 b) | a – b |
| int128\_nonneg(Int128 a) | a >= 0 |
| bool  int128\_eq(Int128 a, Int128 b) | a == b |
| bool int128\_ne(Int128 a, Int128 b) | a != b |
| bool int128\_ge(Int128 a, Int128 b) | a >= b |
| bool int128\_lt(Int128 a, Int128 b) | a < b |
| bool int128\_le(Int128 a, Int128 b) | a <= b |
| bool int128\_gt(Int128 a, Int128 b) | a > b |
| bool int128\_nz(Int128 a) | ａ != 0 |
| Int128 int128\_min(Int128 a, Int128 b) | a <= b ? a : b |
| Int128 int128\_max(Int128 a, Int128 b) | a >= b ? a : b |
| void int128\_addto(Int128 \*a, Int128 b) | a = a + b |
| void int128\_subfrom(Int128 \*a, Int128 b) | a = a – b |

### QEMU的AddrRange地址空间对象模型算法总结(QEMU2.0.0)

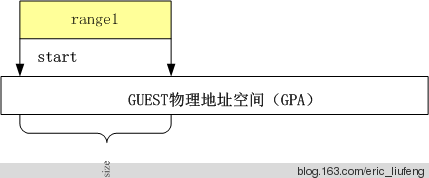
QEMU负责对分配给GUEST物理内存，需要一套数据结构对GUEST物理地址空间进行管理。

AddrRange 是用来管理地址空间计算的一个对象。QEMU通常使用 AddrRange 表示GUEST物理地址空间范围，所以需要注意的是通过AddrRange计算出来的值，一般代表了GUEST的物理地址空间，其中start代表了该段物理地址空间的其实地址，尤其注意的是，它是GPA

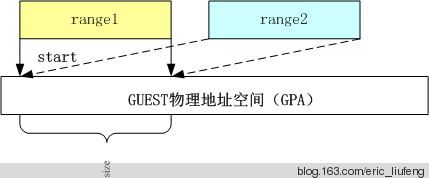
|  |
| --- |
| struct AddrRange {     Int128 start; /\*GPA\*/     Int128 size;  }; |

算法总结如下：

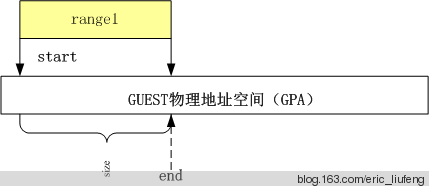
**AddrRange addrrange\_make(Int128 start, Int128 size)**： 定义一个AddrRange范围的GUEST地址空间，起始地址为start, 结束地址为size，如下图所示，黄色代表返回的range



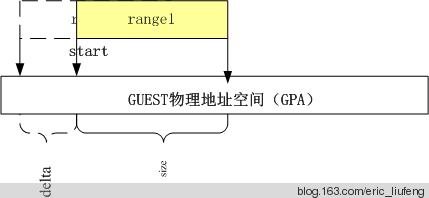
**bool addrrange\_equal(AddrRange r1, AddrRange r2)：** 判断r1和r2所代表的是同一个地址空间，如下图range1 和 range2代表的是同一个物理地址空间



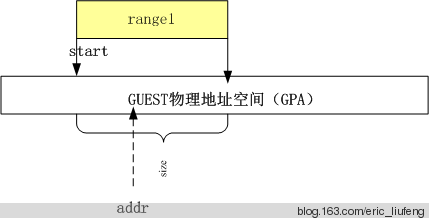
**Int128 addrrange\_end(AddrRange r)**： 返回range所代表的物理地址范围的结尾地址



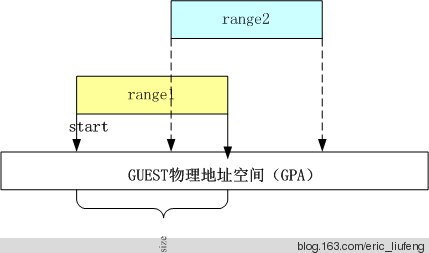
**AddrRange addrrange\_shift(AddrRange range, Int128 delta)**：将range移动delta距离，返回所代表的新的地址空间范围，delta可正可负，代表了移动的方向



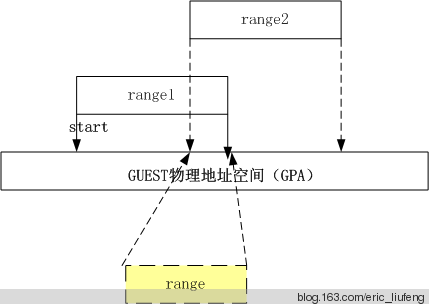
**bool addrrange\_contains(AddrRange range, Int128 addr)：** 判断addr是否在range所覆盖的范围内, addr在range范围内返回true，如下图所示的addr会返回true



**bool addrrange\_intersects(AddrRange r1, AddrRange r2)：** r1和r2是否有重叠区域部分，重叠返回true，如下图所示的r1和r2就是相交的情况，返回true



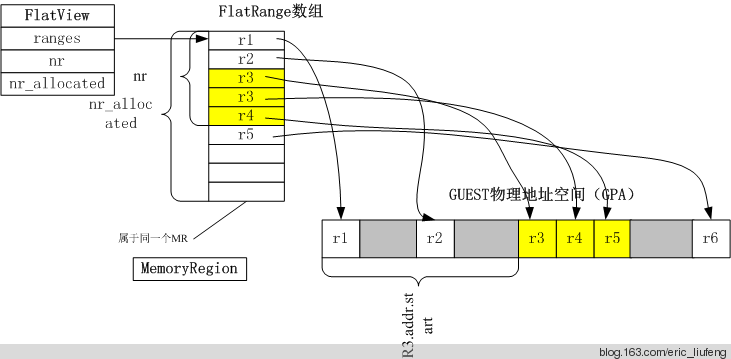
**AddrRange addrrange\_intersection(AddrRange r1, AddrRange r2)**：返回一个range, 代表r1和r2相交的地址空间部分，如下图黄色部分所示



### QEMU内存管理之FlatView模型（QEMU2.0.0）

在QEMU的内存管理中的FlatView描述了QEMU虚拟机内存平坦展开的情况。

首先看一下FlatView模型



**FlatView的原理：**

1. 首先FlatView模型是通过FlatView和FlatRange两个对象组成。

2. FlatView是该段内存的整体视图的管理结构，一个FlatView由一组FlatRange组成。

3. 每个FlatRange代表了虚拟机上的一段内存，多个FlagRange就组成了一个内存视图，这些FlatRange在物理地址空间上不一定是相邻的。

3. 每个FlatView代表了某一类内存的组合，用作特殊的用途（如系统内存空间，MMIO内存地址空间），通常一个FlatView同一个特定用途的Address\_Space进行关联

4. 每个FlatRange所界定的GUEST物理地址空间范围通过AddrRange所界定，有关AddrRange的内存，请参考[《QEMU的AddrRange地址空间对象模型算法总结(QEMU2.0.0)》](http://blog.163.com/eric_liufeng/blog/static/197382683201571713456649)

5. FlatRange数组在FlatView初始化的时候为0个，也就是没有分配数组。当进行flatview\_insert()的操作的时候，才会动态分配出来

6. 为了简化FlatView，通常将地址空间上连续的FlatRange进行合并，合并为1个FlatRange。如图中的r3,r4,r5就可以进行合并的区间，合并后都合并为r3。r4和r5的内容将被后面的FlatRange的数组元素覆盖掉

关于计算地址的方法和其他128bit的算法，请参考[《QEMU的128位算法集合（基于QEMU2.0.0）》](http://blog.163.com/eric_liufeng/blog/static/197382683201571711113246)

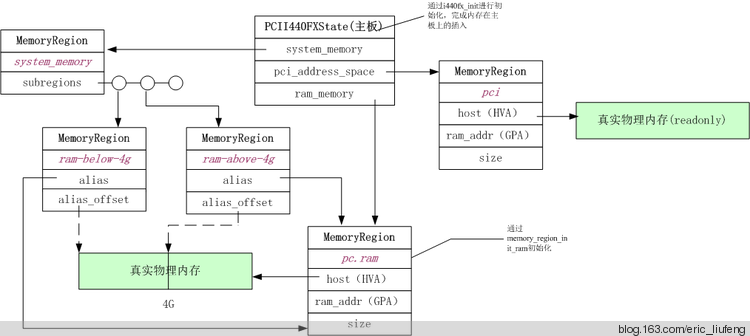
**了解了上面的原理，相关代码如下**

|  |
| --- |
| struct FlatRange {     MemoryRegion \*mr;     hwaddr offset\_in\_region;     AddrRange addr;     uint8\_t dirty\_log\_mask;     bool romd\_mode;     bool readonly; };  /\* \* FlatView是将树状AS的平行展开，可以想象为一个GUEST内存条，所以FlatView里面都是GPA相关的内容 \* FlatView里含有多个FlatRange，每个FlatRange代表了一段内存， \* 所有的FlatRange共同构成了FlatView的Guest内存条 \* 每个FlatRange通过AddrRange标记该段GUEST内存的大小和长度  \*/ struct FlatView {     unsigned ref;               /\*引用计数\*/     FlatRange \*ranges;          /\*指向FlatRange数组\*/     unsigned nr;               /\*已经使用了多少个FlatRange\*/     unsigned nr\_allocated;     /\*总共分配了多少FlatRange\*/ };  /\* \* 遍历FlatView下所有的FlatRange \*/ #define FOR\_EACH\_FLAT\_RANGE(var, view)          \     for (var = (view)->ranges; var < (view)->ranges + (view)->nr; ++var)  /\* \* 判断两个FlatRange是否代表相同的地址空间 \* AddrRange的判断请参见<QEMU的AddrRange地址空间对象模型算法总结(QEMU2.0.0)> \*/ static bool flatrange\_equal(FlatRange \*a, FlatRange \*b) {     ..... }  /\* \* 初始化一个FlatView，开始的时候，内部含有的0个FlatRange \*/ static void flatview\_init(FlatView \*view) {     view->ref = 1;     view->ranges = NULL;     view->nr = 0;     view->nr\_allocated = 0; } /\* \* 向Flatview的指定位置中插入一个FlatRange \*/ static void flatview\_insert(FlatView \*view, unsigned pos, FlatRange \*range) {      /\*      \* 如果FlatRange数组占满了，那么新分配已开内存(原来的2倍大小)      \* 注意这里用的是realloc，是扩大内存大小，不改变原来指针的位置，      \*/     if (view->nr == view->nr\_allocated) {         view->nr\_allocated = MAX(2 \* view->nr, 10);         view->ranges = g\_realloc(view->ranges,                                     view->nr\_allocated \* sizeof(\*view->ranges));     }      /\*      \* 将指定位置之后的内存都向后搬移一下，空出pos的位置      \*/     memmove(view->ranges + pos + 1, view->ranges + pos,             (view->nr - pos) \* sizeof(FlatRange));       /\*      \* 将新的FlatRange的内容记录到pos位置，完成插入      \*/     view->ranges[pos] = \*range;      /\* 每增加一个Range都会让mr的引用加1，因为FlatRange中mr成员会引用mr\*/     memory\_region\_ref(range->mr);      /\* 使用数量加一 \*/     ++view->nr; }  /\* \* 销毁一个FlatView \*/ static void flatview\_destroy(FlatView \*view) {     int i;       /\* 对MR引用计数减少1 \*/     for (i = 0; i < view->nr; i++) {         memory\_region\_unref(view->ranges[i].mr);     }            /\*释放FlatRange数组\*/     g\_free(view->ranges);       /\*释放FlatView\*/     g\_free(view); }  /\*增加FlatView引用计数\*/ static void flatview\_ref(FlatView \*view) {     atomic\_inc(&view->ref); }  /\*减少FlatView引用计数，减少为0的时候，销毁FlatView\*/ static void flatview\_unref(FlatView \*view) {     if (atomic\_fetch\_dec(&view->ref) == 1) {         flatview\_destroy(view);     } }  /\* \* 判断r1和r2是否可以合并，可以合并返回true \*  \* 可以合并的条件如下: \* 1. r1.end(GPA) == r2.start(GPA) \* 2. 属于同一个MR \* 3. r1.offset+size == r2.offset(在MR内的偏移量正好可以对接上) \* 4. dirty\_log相同，迁移使用 \* 5. 读写属性相同 \* 6. 读写模式相同 \*/ static bool can\_merge(FlatRange \*r1, FlatRange \*r2) {     ...... }  /\* \* 通过合并相邻的FlatRange，简化FlatView \*/ static void flatview\_simplify(FlatView \*view) {     unsigned i, j;      i = 0;      /\*遍历所有的FlatRange，进行range的合并\*/     while (i < view->nr) {         j = i + 1;           /\*           \* 将从i开始之后所有的能合并的内存都合并           \* 1. 所有能合并的都往 i 上做合并           \* 2. j指向可以合并的最后一个range的后一个           \*/         while (j < view->nr                && can\_merge(&view->ranges[j-1], &view->ranges[j])) {             /\*可以合并的话，增加i的大小\*/             int128\_addto(&view->ranges[i].addr.size, view->ranges[j].addr.size);             ++j;         }           /\* i指向了下一个待合并填充的位置\*/         ++i;           /\* 进行内存搬移，将合并了的内存覆盖掉 \*/         memmove(&view->ranges[i], &view->ranges[j],                 (view->nr - j) \* sizeof(view->ranges[j]));           /\*修改已用FlatRange的数量，减少已经合并的数量\*/         view->nr -= j - i;     } } |

### MemoryRegion模型原理，以及同FlatView模型的关系(QEMU2.0.0)

**一、MemoryRegion模型**

QEMU通过MemoryRegion来管理虚拟机内存，通过内存属性，GUEST物理地址等特点对内存分类，就形成了多个MemoryRegion，这些MemoryRegion 通过树状组织起来，挂接到根MemoryRegion下。每个MemoryRegion树代表了一类作用的内存，如系统内存空间或IO内存空间。MemoryRegion模型如下图所示，蓝色的部分代表了真实的物理内存



为了方便描述，我将MemoryRegion分为三类：

1. 根MemoryRegion： 不分配真正的物理内存，通过subregions将所有的子MemoryRegion管理起来，如图中的system\_memory

2. 实体MemoryRegion：  这种MemoryRegion中真正的分配物理内存，最主要的就是pc.ram和pci。分配的物理内存的作用分别是内存、PCI地址空间以及fireware空间。QEMU是用户空间代码，分配的物理内存返回的是HVA，被保存到host域。同时这个结构还会为本段虚拟机内存分配虚拟机物理地址空间起始地址，该起始地址(GPA)保存到ram\_addr域，该段内存大小为size。通过实体MemoryRegion就可以将HOST地址HVA和GUEST地址GPA对应起来，这种实体MemoryRegion起到了转换的作用。

3. 别名MemoryRegion： 这种MemoryRegion中不分配物理内存，代表了实体MemoryRegion的一个部分，通过alias域指向实体MemoryRegion，alias\_offset代表了该别名MemoryRegion所代表内存起始GPA相对于实体MemoryRegion所代表内存起始GPA的偏移量，通常用来计算别名MemoryRegion对应的物理内存的HVA值：HVA = 起始HVA + alias\_offset。如图中的ram\_above\_4g和ram-below-4g。

**所有实体MemoryRegion都会被插在主板上，如上图pc.ram就被插在I440FX主板的ram\_memory成员中。**

i440fx\_init() {

…

f->system\_memory = address\_space\_mem;

f->pci\_address\_space = pci\_address\_space;

f->ram\_memory = ram\_memory;

…

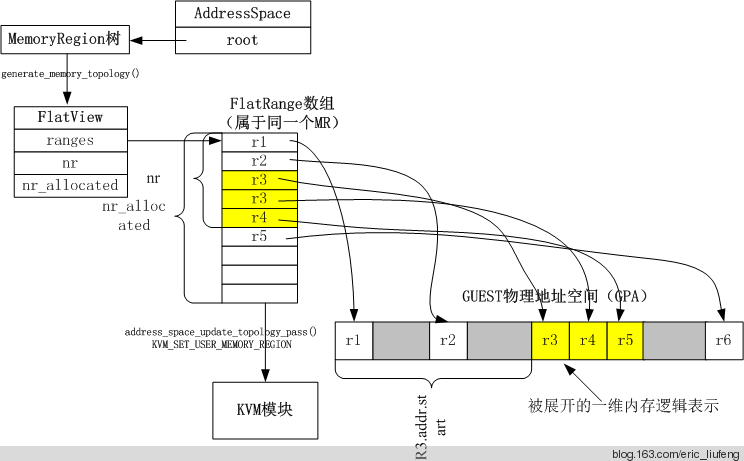
}

**二、FlatView模型**

参考博文[《QEMU内存管理之FlatView模型（QEMU2.0.0）》](http://blog.163.com/eric_liufeng/blog/static/197382683201571755153732)

**三、MemoryRegion模型和FlatView模型的关系**

MemoryRegion是QEMU管理内存的树状结构，便于按照功能、属性分类；但这只是管理结构。但虚拟机的内存需要通过KVM\_SET\_USER\_MEMORY\_REGION，将HVA和GPA的对应关系注册到KVM模块的memslot，才可以生效成为EPT。如果QEMU直接使用MemoryRegion进行注册，那么注册的过程将会很麻烦，也容易不断的出现重叠判断等。所以在通过KVM\_SET\_USER\_MEMORY\_REGION注册前，加了一层转换机制，先将树状的MemoryRegion展开物理内存样子的一维区间结构，然后再通过KVM\_SET\_USER\_MEMORY\_REGION将这个展开的物理内存注册到KVM内核模块中，就方便了许多。这个转换机制就是FlatView模型。整个转换过程请参见函数address\_space\_update\_topology，后面会再写博客进行详细分析。FlatView模型见下图所示



 address\_space\_update\_topology将指定的AddressSpace下的MemoryRegion树进行展平，形成了对应一维内存逻辑表示的FlatView，然后再address\_space\_update\_topology\_pass中将FlatView模型通过KVM\_SET\_USER\_MEMORY\_REGION注册到KVM模块中。

### 通过KVM\_SET\_USER\_MEMORY\_REGION操作虚拟机内存（Kernel 3.10.0 & qemu 2.0.0）

先界定几个术语

GPA：Guest 物理地址

HVA：HOST 虚拟地址

MR：Memory Region，QEMU虚拟机中地址空间的管理结构

在QEMU用户态进程中，会调用KVM\_SET\_USER\_MEMORY\_REGION来修改虚拟机的内存，kvm\_set\_user\_memory\_region就是内核操作KVM\_SET\_USER\_MEMORY\_REGION的接口

例如：

|  |
| --- |
| static int kvm\_set\_user\_memory\_region(KVMState \*s, KVMSlot \*slot)  {      struct kvm\_userspace\_memory\_region mem;      .......     /\*通过KVM\_SET\_USER\_MEMORY\_REGION进入Kernel\*/      mem.guest\_phys\_addr = slot->start\_addr;      mem.userspace\_addr = (unsigned long)slot->ram;      mem.memory\_size = slot->memory\_size;      return kvm\_vm\_ioctl(s, KVM\_SET\_USER\_MEMORY\_REGION, &mem);  } |

我们看一下，内核是处理如何操作的

kvm\_vm\_compat\_ioctl

-> kvm\_vm\_ioctl (case KVM\_SET\_USER\_MEMORY\_REGION)

-> kvm\_vm\_ioctl\_set\_memory\_region

-> kvm\_set\_memory\_region

-> \_\_kvm\_set\_memory\_region

int \_\_kvm\_set\_memory\_region(struct kvm \*kvm, struct kvm\_userspace\_memory\_region \*mem)  
{  
 struct kvm\_memory\_slot old, new/\*待添加的slot的内容\*/;   
   
 ......   
 /\*设置memslot的base\_gfn、npages等域\*/  
 base\_gfn = mem->guest\_phys\_addr >> PAGE\_SHIFT;  
 /\*memory\_size为0，就说明npages = 0, npages 在后面会被标记为KVM\_MR\_DELETE，说明是内存删除动作\*/  
 npages = mem->memory\_size >> PAGE\_SHIFT;   
   
  
 /\*判断操作的类型\*/  
 r = -EINVAL;  
 if (npages) { /\*非删除\*/  
 if (!old.npages) /\*老的页面数为0，说明是新增\*/  
 change = KVM\_MR\_CREATE;  
 else { /\* Modify an existing slot. \*/  
 if (base\_gfn != old.base\_gfn) /\*新旧插槽只有gfn不同，那么就是MOVE\*/  
 change = KVM\_MR\_MOVE;  
 else if (new.flags != old.flags)   
 change = KVM\_MR\_FLAGS\_ONLY;  
 else { /\* Nothing to change. \*/  
 r = 0;  
 goto out;  
 }  
 }  
 } else if (old.npages) { /\*待设定的页面数位0，说明是删除\*/  
 /\*memory\_size = 0\*/  
 change = KVM\_MR\_DELETE;  
 } else /\* Modify a non-existent slot: disallowed. \*/  
 goto out;  
  
 /\*  
 \* 处理和已经存在的memslots的重叠，发现重叠返回-EEXIST  
 \*/  
 if ((change == KVM\_MR\_CREATE) || (change == KVM\_MR\_MOVE)) {  
 /\* Check for overlaps \*/  
 r = -EEXIST;  
 kvm\_for\_each\_memslot(slot, kvm->memslots) {  
 if ((slot->id >= KVM\_USER\_MEM\_SLOTS) ||  
 (slot->id == mem->slot)/\*当前要加入的slot，不管，直接跳过\*/)  
 continue;  
 /\* new\_end > slot\_base && new\_base < slot\_end，说明已经有覆盖该段内存了\*/  
 if (!((base\_gfn + npages <= slot->base\_gfn) ||  
 (base\_gfn >= slot->base\_gfn + slot->npages)))  
 goto out;  
 }  
 }  
  
 if (change == KVM\_MR\_CREATE) {  
 /\*初始化HVA的内容\*/  
 new.userspace\_addr = mem->userspace\_addr;  
 /\*设定memslot中arch相关的内容\*/  
 if (kvm\_arch\_create\_memslot(&new, npages))  
 goto out\_free;  
 }  
  
 if ((change == KVM\_MR\_DELETE) || (change == KVM\_MR\_MOVE)) {  
 r = -ENOMEM;  
 slots = kmemdup(kvm->memslots, sizeof(struct kvm\_memslots),GFP\_KERNEL);  
 if (!slots)  
 goto out\_free;  
 slot = id\_to\_memslot(slots, mem->slot);  
 slot->flags |= KVM\_MEMSLOT\_INVALID;  
  
 old\_memslots = install\_new\_memslots(kvm, slots, NULL);  
  
 /\* slot was deleted or moved, clear iommu mapping \*/  
 kvm\_iommu\_unmap\_pages(kvm, &old);  
 /\* From this point no new shadow pages pointing to a deleted,  
 \* or moved, memslot will be created.  
 \*  
 \* validation of sp->gfn happens in:  
 \* - gfn\_to\_hva (kvm\_read\_guest, gfn\_to\_pfn)  
 \* - kvm\_is\_visible\_gfn (mmu\_check\_roots)  
 \*/  
 kvm\_arch\_flush\_shadow\_memslot(kvm, slot);  
 slots = old\_memslots;  
 }  
  
 ......  
}

从上面的代码看来：

如果memory\_size大于0，那么这次内存操作的动作为“向虚拟机中添加内存或者更新内存布局”

如果memory\_size等于0，那么这次内存操作的动作为“将guest\_phys\_addr 为起始地址的那块内存块从虚拟机中删除”

OK，知道了上面的内容，让我们来理解一个QEMU里面的关键函数，这个函数会在后面讲解QEMU内存初始化流程的博文中再次提到。它就是kvm\_set\_phys\_mem，通过该函数，QEMU可以修改虚拟机注册在内核中的内存，达到为虚拟机添加/删除/移动内存的目的

在讲解前，我们先了解一下什么是overlap。

overlap就是重叠的意思，见图1

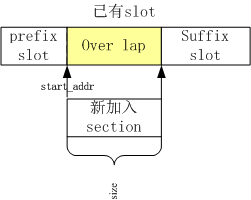


                            图1: overlap内存

所谓slot可以理解为物理机上的一个内存插槽上插入的内存，已有slot，就是虚拟机中已经有的内存

当新加入的section管理的地址范围，完全落入已有slot的范围内（见图1），这样就叫做overlap，注意是完全落入哦，重叠一部分不算，参加函数kvm\_lookup\_overlapping\_slot

|  |
| --- |
| /\* \* 在KVMSlot插槽数组中查找同制定范围重叠的KVMSlot \* 注意: 这里完全覆盖才算找到 \*/ static KVMSlot \*kvm\_lookup\_overlapping\_slot(KVMState \*s,  hwaddr start\_addr, hwaddr end\_addr) {      for (i = 0; i < s->nr\_slots; i++) {         KVMSlot \*mem = &s->slots[i];        .....           /\*找到有地址空间重叠的KVMSlot区域\*/         if (end\_addr > mem->start\_addr &&             start\_addr < mem->start\_addr + mem->memory\_size) {             found = mem;         }     }     return found; } |

那么QEMU会怎么处理这种overlap的情况呢，是这样的：

1.  将已有slot分为三个部分，prefix slot + overlap + suffix slot

2.  将已有的slot内存，通过KVM\_SET\_USER\_MEMORY\_REGION，从内核KVM模块中删除

3.  通过KVM\_SET\_USER\_MEMORY\_REGION，向内核kvm模块注册添加prefix slot内存

4.  通过KVM\_SET\_USER\_MEMORY\_REGION，向内核kvm模块注册添加suffix slot内存

5.  通过KVM\_SET\_USER\_MEMORY\_REGION，向内核kvm模块注册添加overlap内存，起始overlap内存就是本次这次的section的内存

OK，这样就将内存加入进去了。

一定有人会问，为什么不直接利用原来的内存，为什么要新分配呢？

我的理解是这样的，kvm\_set\_phys\_mem是一个操作内存的统一接口，添加内存和删除内存都通过该接口，如果是删除overlap部分，kvm\_set\_phys\_mem的分段工作就不言而喻了。而且如果添加的section的属性变了，如从RAM变成了ROM，那么重新进行添加也是必要的。

当然，prefix slot 和 suffix slot 的长度可以为0，不用多解释了，就是正好覆盖到起始或结束位置。

好了，有了上面的理解，我们就来可以分析一下kvm\_set\_phys\_mem函数了

主要流程如图2所示，就不多废话了

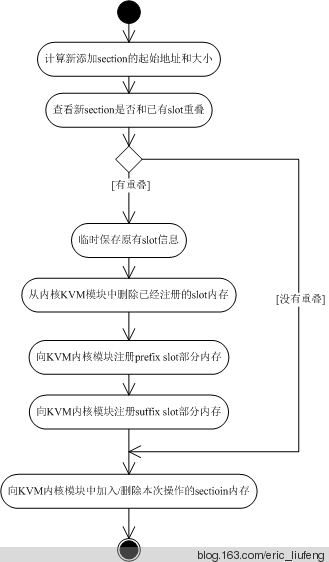


图2

下面是代码分析

|  |
| --- |
| /\*设定虚拟机物理内存\*/ static void kvm\_set\_phys\_mem(MemoryRegionSection \*section, bool add) {     KVMState \*s = kvm\_state;     KVMSlot \*mem, old;     int err;     MemoryRegion \*mr = section->mr;     bool log\_dirty = memory\_region\_is\_logging(mr);     bool writeable = !mr->readonly && !mr->rom\_device;     bool readonly\_flag = mr->readonly || memory\_region\_is\_romd(mr);     hwaddr start\_addr = section->offset\_within\_address\_space; /\*start\_addr为该段section内存GPA的起始地址\*/     ram\_addr\_t size = int128\_get64(section->size); /\*start\_addr为该段section内存的大小\*/     void \*ram = NULL;     unsigned delta;      /\* kvm works in page size chunks, but the function may be called        with sub-page size and unaligned start address. \*/     /\*页面对齐，对页面起始地址和长度进行微调\*/     delta = TARGET\_PAGE\_ALIGN(size) - size;     if (delta > size) {         return;     }     start\_addr += delta;     size -= delta;     size &= TARGET\_PAGE\_MASK;     if (!size || (start\_addr & ~TARGET\_PAGE\_MASK)) {         return;     }       /\*如果注册的内存是ROM，进行权限校验\*/     if (!memory\_region\_is\_ram(mr)) {         if (writeable || !kvm\_readonly\_mem\_allowed) {             return;         } else if (!mr->romd\_mode) {             /\* If the memory device is not in romd\_mode, then we actually want              \* to remove the kvm memory slot so all accesses will trap. \*/             add = false;         }     }       /\*      \* memory\_region\_get\_ram\_ptr 是MR管理的内存块的HVA      \* HVA + section->offset\_within\_region + delta; 就是section所在HVA      \* ram 就是section所在HVA      \*/     ram = memory\_region\_get\_ram\_ptr(mr) + section->offset\_within\_region + delta;      while (1) {           /\*查找和制定区间重叠的KVMSLOT\*/         mem = kvm\_lookup\_overlapping\_slot(s, start\_addr, start\_addr + size);           /\* 如果没有重叠，就跳出循环, 直接添加内存\*/         if (!mem) {             break;         }            /\*从这里开始，说明找到了mem和制定内存返回重叠了\*/          if (add && start\_addr >= mem->start\_addr &&             (start\_addr + size <= mem->start\_addr + mem->memory\_size) &&             (ram - start\_addr == mem->ram - mem->start\_addr)) {             /\* The new slot fits into the existing one and comes with              \* identical parameters - update flags and done. \*/             kvm\_slot\_dirty\_pages\_log\_change(mem, log\_dirty);             return;         }            /\*临时保存原有slot信息\*/         old = \*mem;          if (mem->flags & KVM\_MEM\_LOG\_DIRTY\_PAGES) {             kvm\_physical\_sync\_dirty\_bitmap(section);         }            /\*通过将memory\_size设置为0，通过KVM\_SET\_USER\_MEMORY\_REGION从内核KVM模块中删除已经注册的内存\*/         /\* unregister the overlapping slot \*/         mem->memory\_size = 0;           /\*设定虚拟机物理内存\*/         err = kvm\_set\_user\_memory\_region(s, mem);         if (err) {             fprintf(stderr, "%s: error unregistering overlapping slot: %s\n",                     \_\_func\_\_, strerror(-err));             abort();         }          /\* Workaround for older KVM versions: we can't join slots, even not by          \* unregistering the previous ones and then registering the larger          \* slot. We have to maintain the existing fragmentation. Sigh.          \*          \* This workaround assumes that the new slot starts at the same          \* address as the first existing one. If not or if some overlapping          \* slot comes around later, we will fail (not seen in practice so far)          \* - and actually require a recent KVM version. \*/           /\*           \* 一般不会走这里，因为broken\_set\_mem\_region为0，暂时不看           \*/         if (s->broken\_set\_mem\_region &&             old.start\_addr == start\_addr && old.memory\_size < size && add) {             mem = kvm\_alloc\_slot(s);             mem->memory\_size = old.memory\_size;             mem->start\_addr = old.start\_addr;             mem->ram = old.ram;             mem->flags = kvm\_mem\_flags(s, log\_dirty, readonly\_flag);              err = kvm\_set\_user\_memory\_region(s, mem);             if (err) {                 fprintf(stderr, "%s: error updating slot: %s\n", \_\_func\_\_,                         strerror(-err));                 abort();             }              start\_addr += old.memory\_size;             ram += old.memory\_size;             size -= old.memory\_size;             continue;         }            /\*           \* 通过KVM\_SET\_USER\_MEMORY\_REGION，向KVM内核模块注册prefix slot部分内存           \* 通过临时保存的old，计算prefix slot的起始地址和大小                     \*/         /\* register prefix slot \*/         if (old.start\_addr < start\_addr) {             mem = kvm\_alloc\_slot(s);             mem->memory\_size = start\_addr - old.start\_addr;             mem->start\_addr = old.start\_addr;             mem->ram = old.ram;             mem->flags =  kvm\_mem\_flags(s, log\_dirty, readonly\_flag);              err = kvm\_set\_user\_memory\_region(s, mem);             if (err) {                 fprintf(stderr, "%s: error registering prefix slot: %s\n",                         \_\_func\_\_, strerror(-err)); #ifdef TARGET\_PPC                 fprintf(stderr, "%s: This is probably because your kernel's " \                                 "PAGE\_SIZE is too big. Please try to use 4k " \                                 "PAGE\_SIZE!\n", \_\_func\_\_); #endif                 abort();             }         }            /\*           \* 通过KVM\_SET\_USER\_MEMORY\_REGION，向KVM内核模块注册suffix slot部分内存           \* 通过临时保存的old，计算suffix slot的起始地址和大小                     \*/         /\* register suffix slot \*/         if (old.start\_addr + old.memory\_size > start\_addr + size) {             ram\_addr\_t size\_delta;              mem = kvm\_alloc\_slot(s);             mem->start\_addr = start\_addr + size;             size\_delta = mem->start\_addr - old.start\_addr;             mem->memory\_size = old.memory\_size - size\_delta;             mem->ram = old.ram + size\_delta;             mem->flags = kvm\_mem\_flags(s, log\_dirty, readonly\_flag);              err = kvm\_set\_user\_memory\_region(s, mem);             if (err) {                 fprintf(stderr, "%s: error registering suffix slot: %s\n",                         \_\_func\_\_, strerror(-err));                 abort();             }         }     }      /\* in case the KVM bug workaround already "consumed" the new slot \*/     if (!size) {         return;     }     if (!add) {         return;     }       /\*新分配一个slot，将section的信息填入其中，并通过KVM\_SET\_USER\_MEMORY\_REGION向KVM内核模块中加入/删除内存\*/     mem = kvm\_alloc\_slot(s);     mem->memory\_size = size;     mem->start\_addr = start\_addr;     mem->ram = ram;      mem->flags = kvm\_mem\_flags(s, log\_dirty, readonly\_flag);      err = kvm\_set\_user\_memory\_region(s, mem);     if (err) {         fprintf(stderr, "%s: error registering slot: %s\n", \_\_func\_\_,                 strerror(-err));         abort();     } } |

这里补充说明一个关键数据结构MemoryRegionSection，这个结构指定了一段内存，这段内存将被注册到内核KVM模块，进行加入或删除操作。

不多说了，上代码

|  |
| --- |
| struct MemoryRegionSection {     MemoryRegion \*mr;     AddressSpace \*address\_space;      /\*      \* 相当于在region内偏移量，region上面挂载了一块从HOST上分配的内存，通过这个offset，就可以计算这个section在HOST内存上的HVA了      \*/     hwaddr offset\_within\_region;         /\*该段内存的大小\*/     Int128 size;      /\*      \* AS内偏移量，该值是GPA，相当于从GUEST物理地址0处开始的偏移量，也就是说，这个值是该段内存GPA的起始地址      \* 这很好理解，如果AS代表的是系统内存，那么AS内的偏移量当然是物理地址      \*/     hwaddr offset\_within\_address\_space;     /\*指明是ROM还是RAM\*/     bool readonly; }; |

### 如何查看系统中都注册了哪些MemoryRegion(QEMU2.0.0)

1. 修改代码

${QEMU\_SRC}/memory.c

1. 修改代码

${QEMU\_SRC}/memory.c

static void render\_memory\_region(FlatView \*view,  
 MemoryRegion \*mr,  
 Int128 base,  
 AddrRange clip,  
 bool readonly){

....

int alias = 0;  
 int sub = 0;

.....

if (mr->alias) {  
 alias = 1;  
 //printf("Liufeng: render\_memory\_region 1\n");  
 int128\_subfrom(&base, int128\_make64(mr->alias->addr));  
 int128\_subfrom(&base, int128\_make64(mr->alias\_offset));  
 render\_memory\_region(view, mr->alias, base, clip, readonly);  
 return;  
 }  
 ......  
 /\* Render subregions in priority order. \*/  
 QTAILQ\_FOREACH(subregion, &mr->subregions, subregions\_link) {  
 sub = 1;  
 //printf("Liufeng: render\_memory\_region 2\n");  
 render\_memory\_region(view, subregion, base, clip, readonly);  
 }

.....

/\*这里加入是因为不用重复打印alias和sub, 因为它们是递归调用处理的\*/

if(alias == 0 && sub == 0){  
 printf("Liufeng: render\_memory\_region mr.name = %s, offset\_in\_region = 0x%x\n",mr->name,offset\_in\_region);  
 }

.....

}

2.  运行QEMU，

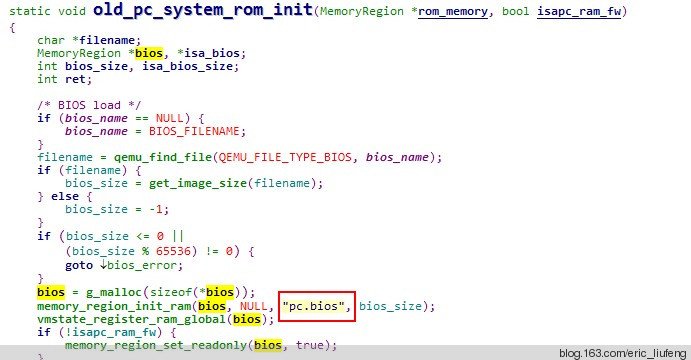
输出结果实例：

Liufeng: render\_memory\_region mr.name = vga.vram, offset\_in\_region = 0x8000  
Liufeng: render\_memory\_region mr.name = vga.vram, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = cirrus-low-memory, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = hpet, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = kvm-ioapic, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0xc0000000  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = msix-pba, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = msix-table, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = msix-pba, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = msix-table, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = cirrus-mmio, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = vga.vram, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = cirrus-bitblt-mmio, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = cirrus-linear-io, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = vga.vram, offset\_in\_region = 0x8000  
Liufeng: render\_memory\_region mr.name = vga.vram, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = cirrus-low-memory, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = pc.rom, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = pc.bios, offset\_in\_region = 0x20000  
Liufeng: render\_memory\_region mr.name = pc.bios, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = kvm-apic-msi, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0xcb000  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0xec000  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0xe8000  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0xe4000  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0xe0000  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0xdc000  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0xd8000  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0xd4000  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0xd0000  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0xcc000  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0xc8000  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0xc4000  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0xc0000  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0xf0000  
Liufeng: render\_memory\_region mr.name = vga.vram, offset\_in\_region = 0x8000  
Liufeng: render\_memory\_region mr.name = vga.vram, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = cirrus-low-memory, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = hpet, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = kvm-ioapic, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0xc0000000  
Liufeng: render\_memory\_region mr.name = pc.ram, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = msix-pba, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = msix-table, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = msix-pba, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = msix-table, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = cirrus-mmio, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = vga.vram, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = cirrus-bitblt-mmio, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = cirrus-linear-io, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = vga.vram, offset\_in\_region = 0x8000  
Liufeng: render\_memory\_region mr.name = vga.vram, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = cirrus-low-memory, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = pc.rom, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = pc.bios, offset\_in\_region = 0x20000  
Liufeng: render\_memory\_region mr.name = pc.bios, offset\_in\_region = 0x0  
Liufeng: render\_memory\_region mr.name = kvm-apic-msi, offset\_in\_region = 0x0

3. 在按照名字找各个memoryRegion在哪里注册的

如：





### QEMU 中关于CPU初始化的重要函数调用栈

<blog>