QEMU

TB 查找时的地址转换

20190905

Loongson Lab - Binary Translation

TranslationBlock 数据结构

target_ulong	pc
target_ulong	cs_base
uint16_t	size
uint8_t	*tc_ptr
tb_page_addr_t	page_addr[2]
struct TranslationBlock	*phys_hash_next
uint16_t	tb_next_offset[2]
uint16_t	tb_jmp_offset[2]
unsigned long	tb_next[2]
struct TranslationBlock	*tb_loopup_cache[N]

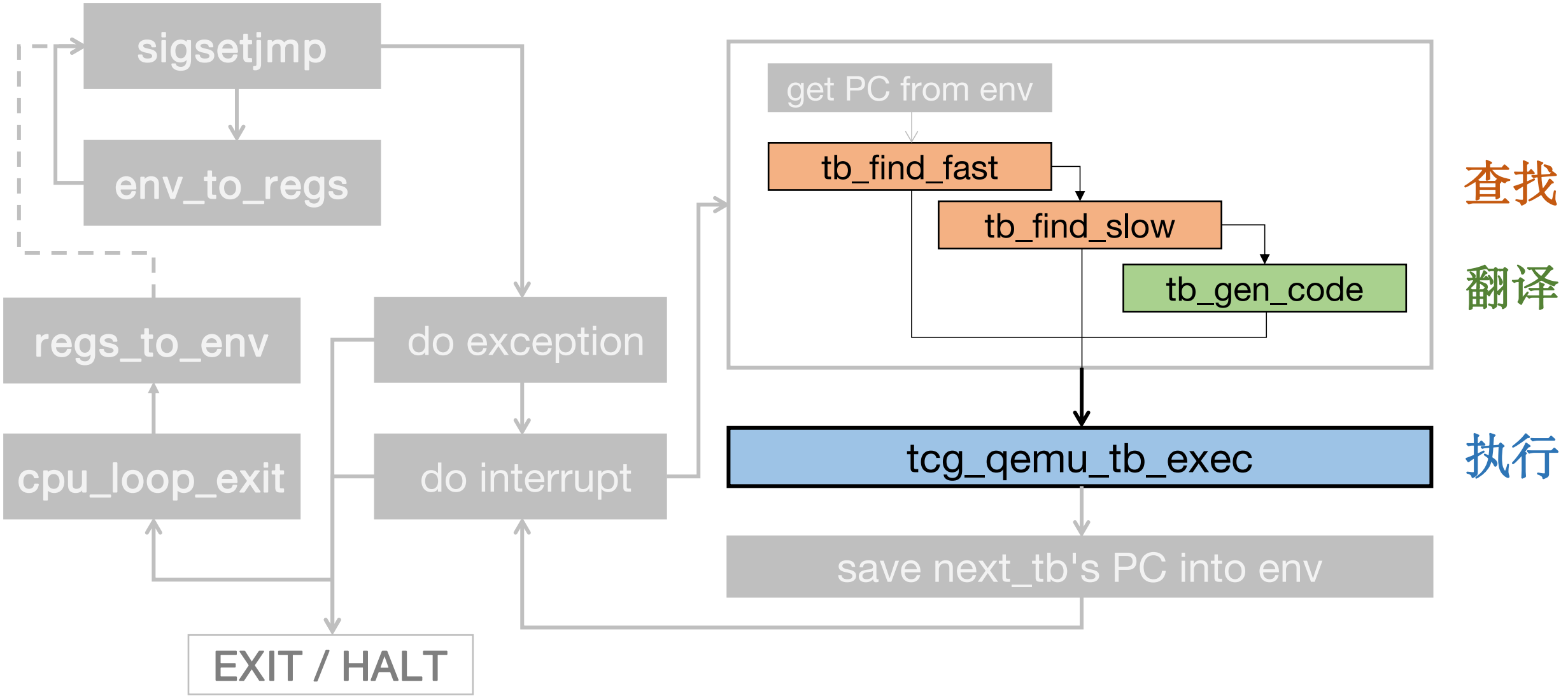
- target 代码
  - PC
  - 代码段基址 cs_base
  - 大小 size
- host 代码
  - 指针 tc_ptr
  - 所在 host 页的虚地址

→ 用于哈希表链接的指针

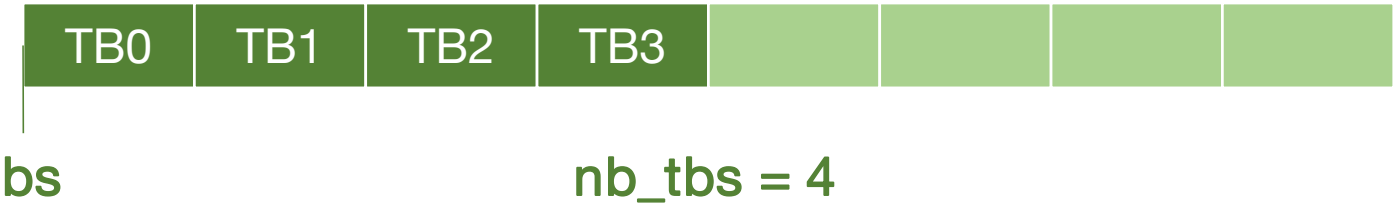
运行时动态绑定 TB 间跳转关系

→ 运行时快速查找下一个 TB

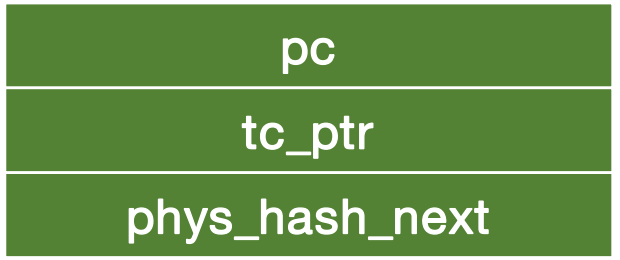
主循环： cpu_exec(env)



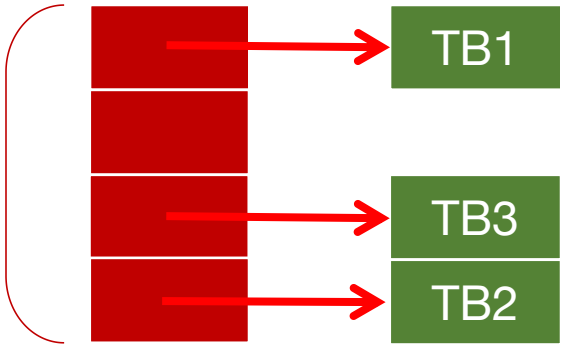
tb_find_fast 索引是 pc



TranslationBlock



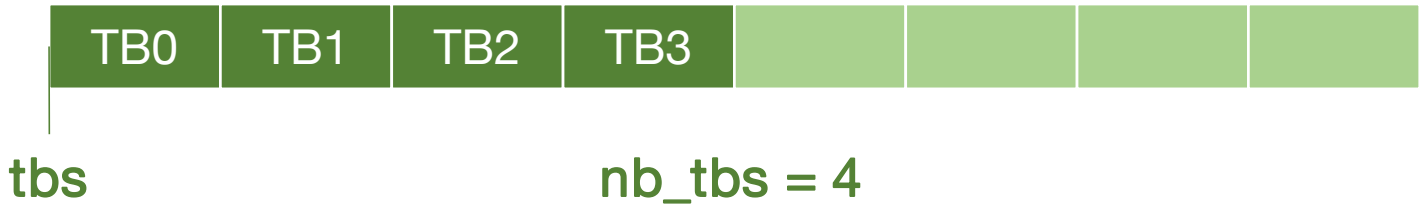
索引
hash_func(pc)
通过 tb->pc 计算

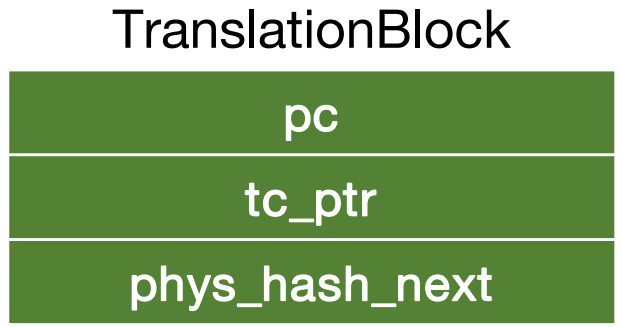


env->tb_jmp_cache

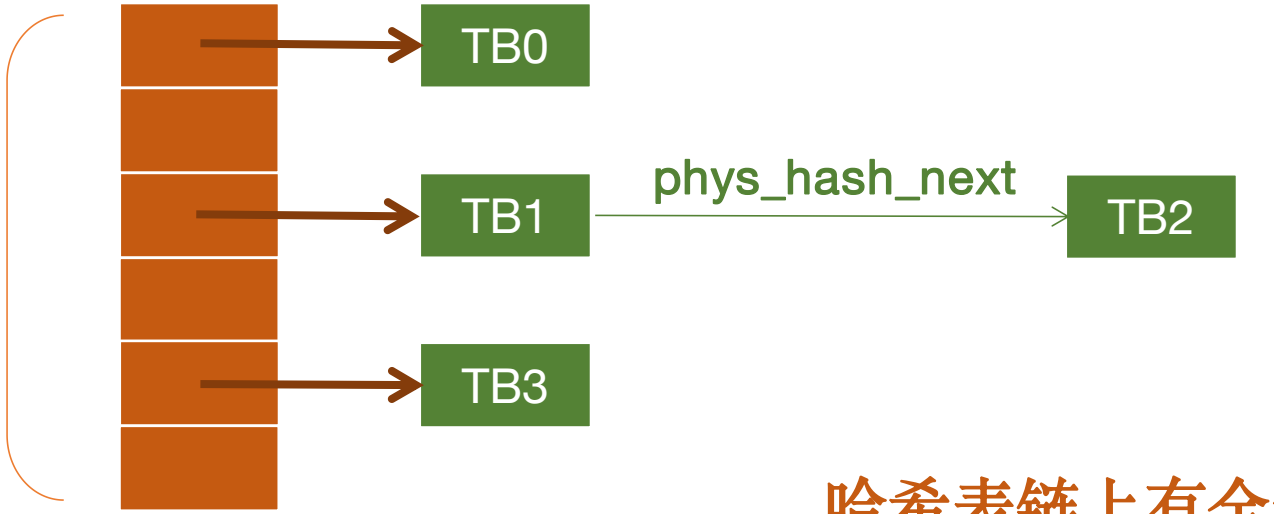
cache存有部分或全部的TB
相当于每个node只有一个TB的哈希表

tb_find_slow 索引是 phys_pc



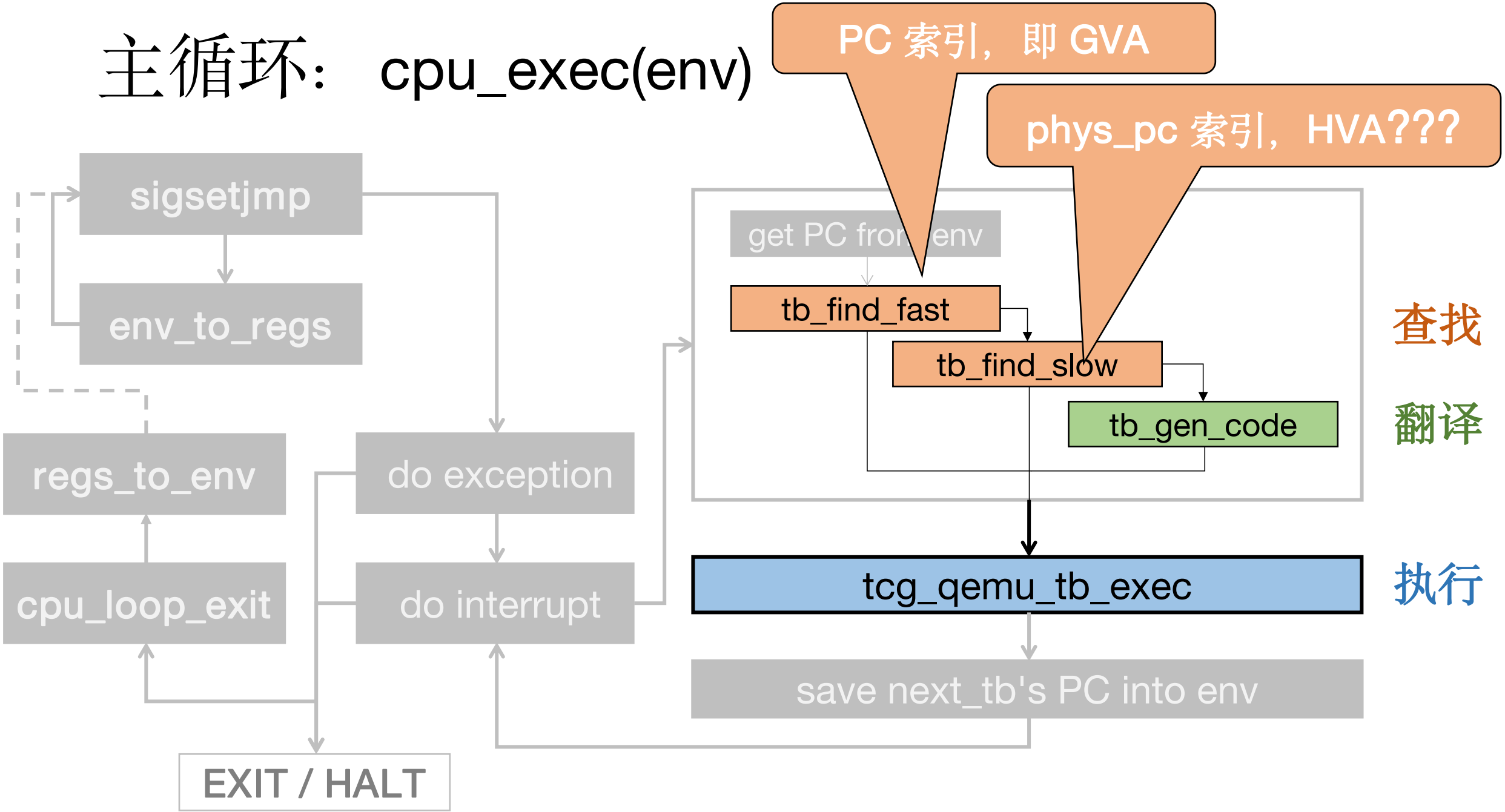


索引
hash_func(phys_pc)
通过物理地址计算



哈希表链上有全部的TB

主循环： cpu_exec(env)



查找TB： tb_find_slow(pc,cs_base,flags)

- cpu-exec.c:177 TranslationBlock *tb_find_slow(pc,cs_base,flags)
  - **phys_pc = get_page_addr_code(env, pc);**
  - phys_page1 = phys_pc & TARGET_PAGE_MASK; phys_page2 = -1;
  - hash = tb_phys_hash_func(phys_pc); ptb1 = &tb_phys_hash[hash];
  - for(;;)
    - tb = *ptb1
    - if(!tb) goto not_found;
    - // check pc & cs_base & flags & tb->page_addr[0:1] // 连续两页 or 只有一页
    - check OK goto found;
    - **ptb1 = &tb->phys_hash_next;**
  - not_found: tb = tb_gen_code(env, pc, cs_base, flags, 0);
  - found: move tb to the head of the list in tb_phys_hash
  - env->tb_jmp_cache[tb_jmp_cache_hash_func(pc)] = tb;

地址翻译

addr 即 tb->pc

- exec-all.h:389 **get_page_addr_code**(*env1, **addr**)
  - page_index = (**addr** >> PAGEBITS) & (CPU_TLB_SIZE - 1)
  - mmu_idx = cpu_mmu_index(env1)
  - if(unlikely(env1->tlb_table[mmu_idx][page_index].addr_code != (addr & PAGE_MASK))
    - ldub_code(addr)
  - **p** = **addr** + env1->**tlb_table**[mmu_idx][page_index].**addend**;
  - return **qemu_ram_addr_from_host_nofail**(**p**);

p = tb->pc + addend

地址翻译：TLB组织

- target-i386/cpu.h:616 struct CPUX86State {
  - CPU_COMMON
- cpu-defs.h:155 #define CPU_COMMON
  - CPU_COMMON_TLB
- cpu-defs.h:111 #define CPU_COMMON_TLB
  - CPUTLBEntry tlb_table[NB_MMU_MODES][CPU_TLB_SIZE];
- cpu-defs.h:89 struct CPUTLBEntry {
  - target_ulong addr_read;
  - target_ulong addr_write;
  - target_ulong addr_code;
  - unsigned long addend; // add to VA to get host address

地址翻译

addr 即 tb->pc

- exec-all.h:389 **get_page_addr_code**(*env1, **addr**)
  - page_index = (**addr** >> PAGEBITS) & (CPU_TLB_SIZE - 1)
  - mmu_idx = cpu_mmu_index(env1)
  - if(unlikely(env1->tlb_table[mmu_idx][page_index].addr_code != (addr & PAGE_MASK))
    - ldub_code(addr)
  - **p** = **addr** + env1->**tlb_table**[mmu_idx][page_index].**addend**;
  - return **qemu_ram_addr_from_host_nofail**(**p**);

p = tb->pc + addend，就是 HVA !

地址翻译

pc + addend，是对应的 HVA

???

- exec.c:3357 qemu_ram_addr_from_host(*ptr, *ram_addr)
  - RAMBlock *block;
  - uint8_t *host = ptr;
  - QLIST_FOREACH(block, &ram_list.blocks, next)

```
if ( host - block->host < block->length )
    *ram_addr = block->offset + ( host - block->host )
```

- cpu-all.h:870 struct RAMBlock {
  - uint8_t *host;
  - ram_addr_t offset;
  - ram_addr_t length;

RAMBlock是啥

从遥远的 main 函数讲起

- ./qemu -L pc-bios **-m 32** ~/dos.img
- int main(argc, argv, envp) { // vl.c:1987
  - QEMUMachine *machine;
  - machine = find_default_machine(); // 遍历全局变量 first_machine 链表
  - machine->**init**(**ram_size**, boot_devices,
    kernel_filename, kernel_cmdline,
    initrd_filename, cpu_model);
- struct QEMUMachine { // hw/boards.h:15
  - **QEMUMachineInitFunc *init;**
- **void pc_init_pci(ram_size, ...)** // hw/pc_piix.c:189

从遥远的 main 函数讲起

- static void pc_init_pci(ram_size, ...) // hw/pc_piix.c:189
  - pc_init1(ramsize, ...)
- static void pc_init1(ram_size, ...) // hw/pc_piix.c:63
  - pc_memory_init(ram_size, ...)
- void pc_memory_init(ram_size, ...) // hw/pc.c:959
  - ram_addr = qemu_ram_alloc(...
  - cpu_register_physical_memory(...

- ram_addr_t qemu_ram_alloc(...) // define in cpu-common.h:51

从遥远的 main 函数讲起

- qemu_ram_alloc_from_ptr(*dev, *name, size, *host) // exec.c:3189
  - RAMBlock *new_block, *block;
  - new_block = qemu_mallocz(sizeof(*new_block))
  - **new_block->host = qemu_vmalloc(size);**
  - qemu_madvise(new_block->host, size, ...);
  - **new_block->offset = find_ram_offset(size);**
  - new_block->length = size;

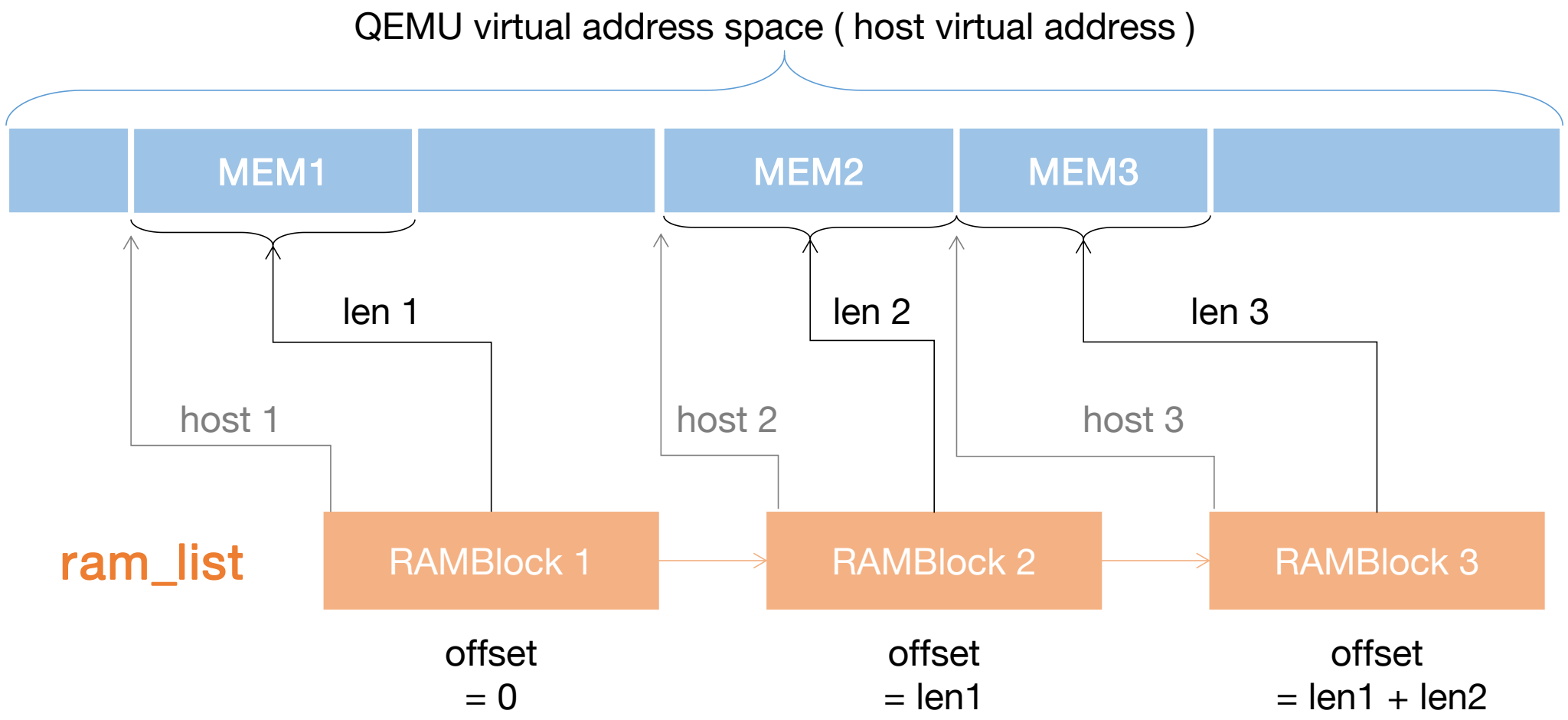
  - return new_block->offset;

从遥远的 main 函数讲起

- void *qemu_vmalloc(size) // oslib-posix.c:64
  - qemu_memalign(PAGESIZE, size)
- void *qemu_memalign(alignment, size) // oslib-posix.c:43
  - ret = **posix_memalign**(**&ptr**, alignment, size)
  - trace_qemu_memalign(alignment, size, ptr);
  - return **ptr**;
- posix_memalign(**memptr, alignment, size) // stdlib
  - 是 stdlib 里的一个基本函数，申请动态内存
  - 申请的内存的地址存放在 **memptr** 中返回

- **RAMBlock->host 是这块内存空间的起始地址，即 HVA**

RAMBlock 的组织



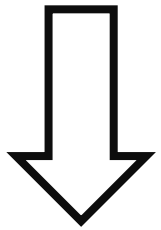
RAMLIST地址空间：所有 RAMBlock 组成的地址空间为 [0, SUM - 1]

用来查找 TB 的 phys_pc 到底是什么

*ram_addr = block->offset + (host - block->host)

phys_pc

host = pc + addend



ram_list

offset = 0

offset = len1

offset = len1 + len2

RAMLIST地址空间：所有 RAMBlock 组成的地址空间为 [0, SUM - 1]

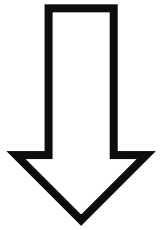
用来查找 TB 的 phys_pc 到底是什么

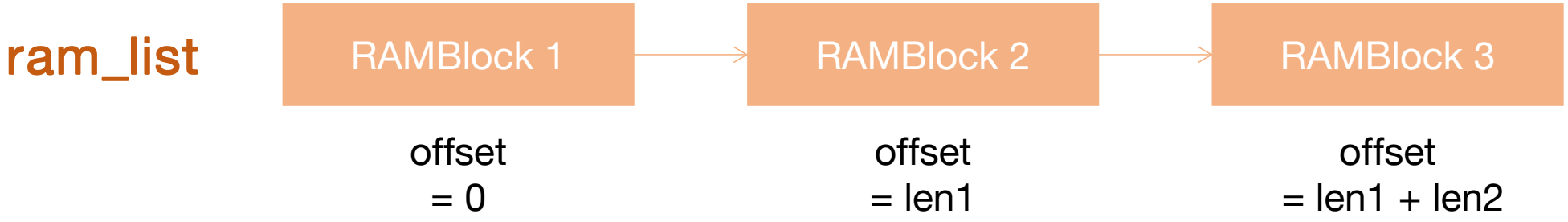
*ram_addr = block->offset + (host - block->host)

phys_pc = len1 + offset

host = pc + addend

phys_pc 是 GVA 映射到RAMLIST地址空间中的地址，仅用来当作 TB 哈希表的索引。





RAMLIST地址空间：所有 RAMBlock 组成的地址空间为 [0, SUM - 1]

总结

- TB 查找中的地址转换
  - tb_find_fast 根据 pc（GVA）索引 TB 缓存
  - tb_find_slow 根据 phys_pc（RAMLIST 地址空间）索引 TB 哈希表
  - get_page_addr_code 仅在此处使用，意味着 RAMLIST 地址空间仅仅是用来作为 TB 哈希表索引的地址空间，而不是使用真正的 HVA 来索引
- 后续工作
  - tlb_table 中的 addend 直接完成 GVA->GPA->HVA 的转换，是如何做到的
  - tlb_table 是如何维护的？何时更新表项？
  - 继续调试 QEMU

完整性补充

- machine的由来：只是为了确认 init 函数指针

从遥远的 main 函数讲起

- ./qemu -L pc-bios -m 32 ~/dos.img
- int main(argc, argv, envp) { // vl.c:1987
  - QEMUMachine *machine;
  - machine = find_default_machine(); // 遍历全局变量 first_machine 链表
  - machine->init(ram_size, boot_devices,
    kernel_filename, kernel_cmdline,
    initrd_filename, cpu_model);
- struct QEMUMachine { // hw/boards.h:15
  - QEMUMachineInitFunc *init;
- void pc_init_pci(ram_size, ...) // hw/pc_piix.c:189

machine的由来

- static QEMUMachine **pc_machine** = { // hw/pc_piix.c:215
  - .name = “pc-0.14”
  - .init = **pc_init_pci**
- static void **pc_machine_init**(void){ // hw/pc_piix.c:379
  - qemu_register_machine(&pc_machine);
- static QEMUMachine ***first_machine**; // vl.c:1112
- int qemu_register_machine(*m) // vl.c:1115
  - pm = &**first_machine**;
  - while(*pm != NULL) pm = &(*pm)->next;
  - *pm = m;
- machine_init(**pc_machine_init**); // hw/pc_piix.c:389
  - 这是个宏!

machine的由来

- #define machine_init(**function**) // module.h:32
  - module_init(function, MODULE_INIT_MACHINE)
- #define module_init(**function**, type) \ // module.h:18
  - static void __attribute__((**constructor**)) do_qemu_init_ ## function() { \
    - register_module_init(**function**, type);

- static void do_qemu_init_pc_machine_init()
  - 作为 constructor 这个函数会在 main 之前执行!

machine的由来

- static ModuleTypeList **init_type_list**[MAX]; // module.c:27
- void register_module_init(void (*fn)(void), type) // module.c:57
  - ModuleEntry *e;
  - ModuleTypeList *l;
  - **e->init = fn; // fn 即 pc_machine_init**
  - **l = find_type(type); // 从全局变量 init_type_list 中寻找**
  - QTAILQ_INSERT_TAIL(l, e, node); // 插入到链表中
    // 从而可以通过全局变量
    // init_type_list 来访问

machine的由来：总结

- constructor 在 main 之前执行，初始化 init_tyoe_list
- int main(argc, argv, envp) // vl.c:1987
  - module_call_init(MODULE_INIT_MACHINE); // vl.c:2151
  - machine = find_default_machine(); // 全局变量 first_machine 链表
- void module_call_init(type)
  - l = find_type(type); // 全局变量 init_type_list
  - QTAILQ_FOREACH(e, l, node)
    - e->init(); // 即 pc_machine_init
      // 从而将静态变量 QEMUMachine pc_machine
      // 加入到全局链表 first_machine