《HyperKernel 在 RISC-V 的移植与 CPU 结合验证》设计文档

陈宇 秦岳

目录

HV6 设计

hv6 在设计的过程中，为了使得设计出的操作系统可以被验证，使得 hv6 在设计之初就考虑到了在验证过程中可能会遇到的困难并且在设计操作系统的时候避开。具体来说，可被验证的操作系统主要有以下问题：

1. 系统调用的接口必须非常简单，并且操作系统执行的代码必须可以被符号化执行。为了解决这个困难，hv6 向用户程序开放了部分核心数据的访问权限，并且修改了一些系统调用的含义，使得系统调用在执行的时候不会出现循环和递归，而仅仅只是参数检查和数据的修改。
2. 内核数据和代码的访问不能经过虚实地址转换。这个问题困难的地方在于，我们很难对虚实地址转换进行建模，而且，即使建模成功，其符号化执行的复杂度也是非常巨大的。而如果不对虚实地址转换进行建模，就必须保证对于内核数据代码的地址转换是正确的，也就是说：**内核的正确性依赖于虚实地址转换的正确性**。但是，由于地址转换所使用的页是由内核分配的(alloc_page)，而如果内核的实现存在问题（比如，将分配给地址转换的页，又错误的分配给了用户进程），则无法保证地址转换的正确性，既：**虚实地址转换的正确性依赖于内核的正确性**。可以看出，地址转换的正确性和内核的正确性相互依赖，是无法被同时证明的。由于这个原因，hv6 对于内核数据代码的访问不经过地址转换，而将用户进程放在了虚拟机中，在虚拟机中可以存在单独的地址转换。在 x86 下，hv6 的内核运行在 host 的内核态(ring 0)，而用户进程则被运行在虚拟机中的内核态(ring 0)，在用户进程中可以直接访问页表，但是对于页表的所有修改都需要通过系统调用交由内核完成。
3. 第三个问题则是要求用于实现操作系统的代码必须便于被符号化执行。所以在 hv6 中选择 C 作为编写语言，其可以被转换为 llvm 格式的代码。

另外，为了简化验证，在进入系统调用的时候会关闭中断而在系统调用结束之后再开启中断，这样，在验证的时候就可以不考虑系统调用执行过程中被打断的情况，以此减少验证过程中的复杂度。

由于 hv6 将很多功能都交由用户程序完成，包括申请内存这样非常敏感的操作，所以 hv6 为用户进程提供了一套工具库 ulib，其主要的功能有两个：在进入用户进程之前完成一些初始化操作：包括将内核的敏感数据以只读的方式映射在固定位置，初始化虚拟机内部的

中断向量表等操作。ulib 还有一个功能就是提供对常用操作的封装以便用户进程调用。此外，ulib 还有一个特点值得一提：ulib 是以一个完整的 ELF 格式文件存在的，其本身可以不依赖用户程序而单独执行。而对于用户进程，其编译时也不是和 ulib 一起编译的：在生成 ulib 可执行文件之后，使用 nm 工具提取 ulib 中的函数符号和位置并构造一个跳转脚本 ustub.s，用户进程在编译的时候和 ustub.s 一起编译。也就是说，用户进程本身不包含 ulib 的任何代码，用户进程和 ulib 可以看做两个完全独立的程序，但是，hv6 必须同时在低地址载入用户进程并在高地址载入 ulib，只有在内存中同时存在用户进程和 ulib 的时候，用户进程才能够正常运行。

移植考量

为了将 hv6 移植到 riscv 平台上并且依然可以被验证正确性，我们同样要面对上面所述的 3 个问题。

首先是对于系统调用的考量。为了保证新系统依然可以被验证，所以我们同样需要保证系统调用的简洁性，包括无循环和递归。因此我们在移植过程中尽量重用了 hv6 的代码，经过我们分析和实验，总共有 54 个系统调用被 hv6 验证，其中 31 个系统调用无需修改，6 个系统调用需要移除，11 个系统调用需要小规模修改，6 个系统调用由于虚拟机的移除需要大规模修改。

而对于第三个问题，新系统同样采用 C 语言编写，所以也可以被符号化执行。

移植的关键点在于上一节中提到的第二个关键点。由于 riscv 并没有虚拟机，而对用户进程的虚实地址转换必不可少，所以对于内核数据代码的访问势必也要经过地址转换，但是由于加入地址转换会导致操作系统不可被验证，为了打破地址转换正确性和内核正确性以循环依赖，我们只能在移植过程中添加假设：**虚实地址转换是正确的**。为了简便，我们对内核数据代码采用恒等映射（虚地址等于实地址），而对于用户进程和 ulib 则按照分配的页进行映射。没有虚拟机造成的第二个困难是用户进程的上下文和中断帧需要我们新建，相应的与进程相关的操作（比如 fork）也需要修改。第三个困难是需要修改中断调用方式并且重新设置中断入口。

最后，我们将内核放在了 riscv 的 S 态（相当于 x86 的 ring 0），而将用户进程运行在 U 态（相当于 x86 的 ring 3）。

移植工作

1. 准备工作

为了能够移植到 riscv 平台下，我们首先要完成的是工具链的搭建。工具链源代码可以从 https://github.com/riscv/riscv-tools 下载，搭建教程可以从 https://github.com/oscourse-tsinghua/OS2018spring-projects-g09 中找到，同样也可以找到 RISCVEMU 的安装教程。

2. Boot Loader 替换

hv6 原有的 Boot Loader 由于有太多和硬件相关的代码，对于移植来说基本上是不可能的，经过调研，我们选择了 Berkeley Boot Loader（BBL）作为移植之后的 Boot Loader。

BBL 载入内核的方式是将内核代码通过汇编指令 incbin 包裹为自己的一部分，所以编译过程由以下几个过程组成：

1. 将内核编译为可执行文件
2. 使用 objcopy 将步骤 1 中的输出文件转化成二进制文件
3. 将步骤 2 中的输出文件和 BBL 一起编译

可以看到，上述存在两次编译过程，尤其是因为 BBL 需要和内核的二进制文件一起编译，导致内核的二进制文件被放置的位置是无法预知的，所以，在编译内核的时候必须编译成不使用绝对路径而全部使用相对路径的代码，在编译参数中添加-mcmodel=medany 即可实现，另外，内核中某些全局变量也需要修改，比如：

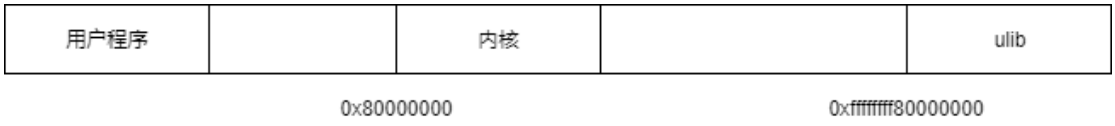
```
1.  void *syscalls[NR_syscalls] = {
2.          [0 ... NR_syscalls - 1] = sys_nop,
3.          [SYS_map_pml4] = sys_map_pml4,
4.          [SYS_map_page_desc] = sys_map_page_desc,
5.  }
```

需要被修改为：

```
1.  void *syscalls[NR_syscalls];
2.
3.  void init_syscalls()
```

```
{
    int i;
    for(i = 0; i < NR_syscalls; i ++) {
        syscalls[i] = sys_nop;
    }
    syscalls[SYS_map_pml4] = sys_map_pml4;
    syscalls[SYS_map_page_desc] = sys_map_page_desc;
}
```

在我们移植之后，hv6 的内存布局如下：



3. 建立页表

由于 x86 和 riscv 对于页表项标识位的含义和位置存在差异，并且 x86 和 riscv 对于页表项的中实地址的储存页不相同，所以我们首先需要修改的便是这部分内容（最麻烦的地方在于 x86 默认内存时可以执行的，只有将 PTE_NX 位置 1 才禁止代码执行，而 riscv 默认内存不可执行，只有 PTE_X 位置 1 才可执行，所以需要在每个地方都判断是否需要添加 PTE_X）。

然后，需要对内核代码建立恒等映射，并且禁止用户程序访问：

```
static void setup_kernel_map(pid_t pid)
{
    assert(KERNSTART % PAGE_SIZE == 0, "KERNBASE % PAGE_SIZE must be 0");
    uintptr_t va;
    pte_t perm = PTE_P | PTE_W | PTE_R | PTE_X;
    for(va = KERNSTART; va < KERNEND; va += PAGE_SIZE) {
        pn_t pt = page_walk(pid, va);
        assert((va - (uintptr_t)pages) % PAGE_SIZE == 0, "(va - pages) % PAGE_SIZE != 0");
        pte_t* pt_page = get_page(pt);
        pt_page[PT_INDEX(va)] = ((va >> PAGE_SHIFT) << PTE_PFN_SHIFT) | perm;
        enum page_type type = get_page_desc(get_proc(pid)->page_table_root)->type;
        if (type == PAGE_TYPE_X86_PML4 || type == PAGE_TYPE_X86_PDPT || type == PAGE_TYPE_X86_PD || type == PAGE_TYPE_X86_PT) {
```

```
            pt_page[PT_INDEX(va)] = ((va >> PAGE_SHIFT) << PTE_PFN_SHIFT) | perm | PTE_U;
        }
    }
}
```

在 hv6 原来的设计中，用户进程对页表具有访问权限而不具有修改权限，所以在对内核代码进行映射的时候也需要开启页表页对于用户进程的可读权限。

另外，由于在原来的页表中并不存在对于内核代码的映射，所以在 exec 等相关需要回收内存的处理的时候，hv6 简单的将低一半的地址全部回收了，但是由于内核代码实际上是位于低一半的地址的，所以需要修改相应代码，对内核代码段进行特判。

4. 系统调用

在 hv6 中，使用 vmcall 在虚拟机中调用主机中的函数，但是在 riscv 中并没有虚拟机的支持，只能使用传统的 ecall（对应 x86 中的 int 指令，但是不能指定中断号）。

经过分析 riscv 平台下 C 语言编译出的汇编代码，发现参数是通过 a0,a1,a2~a7 传递的，最多可以传递 8 个 64 位的整数。在我们的设计中，将需要调用的中断号保存在 a7 中，将参数保存在 a0~a6 中，然后使用 ecall 调用系统函数。由于 a7 被拿来传递中断号，所以系统调用的参数不能超过 7 个，好在经过检查，已有的系统调用都不超过 7 个参数，故此方法可行。

前文提到，ulib 的工作之一是进行初始化，包括将内核关键数据以只读的方式映射到指定地方，在我们的实现中为了简单，在进行 ulib 的初始化时是位于内核态（内核载入 ulib 和用户程序之后直接跳到 ulib 中），而 ulib 为了完成初始化需要调用内核的相关函数（比如建立内存映射），这就需要 ulib 的处理函数可以直接跳转到内核代码，于此同时，用户进程调用 ulib 的函数的时候使用的是完全相同的代码，这就需要在 ulib 中判断当前的进程权限，如果是内核态则直接跳转到内核代码，如果是用户态则使用 ecall 调用内核函数。为此，我们在进入 ulib 的时候将系统调用地址数组传递过去，并且在 ulib 中设置一个全局变量描述当前的权限。

以下是 ulib 的入口部分代码：

```
void** ulib_syscalls;
int is_user_mode;
```

```
uintptr_t cr3_value;

void ulib_init(void* _ulib_syscalls[], void (*fn)())
{
    ulib_syscalls = _ulib_syscalls;
    is_user_mode = 0;
    cr3_value = rcr3();
```

以下是 ulib 中调用内核函数的代码，需要进行权限特判：

```
#define SYSCALL(sym)        \
.global sys_##sym;          \
sys_##sym:                  \
    ld      a7, is_user_mode; \
    bnez    a7, ecall_sys_##sym; \
    j       direct_sys_##sym; \
ecall_sys_##sym:            \
    li      a7, SYS_##sym;  \
    ecall;                  \
    ret;                    \
direct_sys_##sym:           \
    ld      a7, ulib_syscalls;  \
    addi    a7,a7,8*SYS_##sym;  \
    ld      a7, 0(a7);          \
    jr      a7;                 \

SYSCALL(map_page_desc)
SYSCALL(map_pml4)
```

在内核中也需要设置相关的中断处理函数，由于代码太过复杂，所以不贴上来了，中断的入口代码在 hv6/hv6/trap_entry.S 中，相应的 C 处理程序在 hv6/hv6/trap.c 中。

5. 进程管理移植

进程管理模块中，关于进程调度的部分几乎不需要修改，主要修改的部分是上下文切换以及 fork 部分。

对于上下文切换，使用 context 对象保存上下文：

```
struct context {
    uintptr_t ra;
    uintptr_t sp;
```

```
    uintptr_t s0;
    uintptr_t s1;
    uintptr_t s2;
    uintptr_t s3;
    uintptr_t s4;
    uintptr_t s5;
    uintptr_t s6;
    uintptr_t s7;
    uintptr_t s8;
    uintptr_t s9;
    uintptr_t s10;
    uintptr_t s11;
};
```

在 context 数据结构中，并没有储存全部的寄存器，这是由于在 C 语言编程约定中，只有这些寄存器是需要被调用者保存的，我们只需要将这些寄存器保存起来即可。

对于 fork，首先需要处理一些外围工作，包括创建并复制页表，创建内核栈等操作，然后需要复制父进程的上下文，并且设定返回值 a0 和返回地址 ra，以下是 fork 的关键代码：

```
void riscv_hvm_copy(void *_dst, void *_src, pid_t pid)
{
    hvm_t* dst = (hvm_t*)_dst;
    hvm_t* src = (hvm_t*)_src;
    *dst = *src;
    dst->pid = pid;

    dst->tf = (struct trap_frame *)(get_page(get_proc(dst->pid)->stack) + PAGE_SIZE) - 1;
    *(dst->tf) = *src->tf;
    dst->tf->gpr.a0 = 1;

    dst->tf->gpr.sp = src->tf->gpr.sp;
    dst->context.sp = dst->tf;
    dst->context.ra = forkret;
}
```

新系统正确性验证

1. 正确性验证概述[1]

在原 hv6 中，验证 hv6 的正确性需要完成三部分工作：

1) 在进入用户程序之前验证操作系统不变量的正确性

2) 每个系统调用必须符合规约和不变量的要求

3) 从内核代码的任何一个函数开始执行都不会崩栈

对于第一部分，在 hv6 中采用的是动态检查的方式，hv6 在进入用户程序之前，会执行一个 `check_invariants` 函数（在 `hv6/hv6/invariants.c`）中，在该函数中会检查当前系统的不变量是否满足约束。

对于第二部分，则是正确性验证的重点，在 hv6 中，先使用 clang 将内核源代码编译为 llvm 代码，然后使用 llvm-link 工具将这部分代码链接为 hv6.ll，然后使用 irpy 工具将 hv6.ll 转换为 hv6.py，然后使用 z3 工具将 hv6.py 和标准的约束 spec 做比对，实现对每个系统调用的正确性证明。

对于第三部分，则是采用静态检查的方式：使用工具生成 hv6 内核代码函数之间的调用图，并计算出每个函数需要占用多少字节的栈空间，从而可以计算出最长的函数调用路径所需的栈空间，将之与内核最大栈空间 4k 做比较即可。由于 hv6 的内核函数之间的调用不会形成环，所以此方法是可行的。

在我们移植的新系统中，同样需要完成这三部分验证，考虑到第二部分的验证工作最是困难，所以我们的主要工作即是将第二部分由原来 x86 下对于 hv6 的验证移植到 riscv 下对于我们新系统的验证。

2. 前期准备：验证工具链

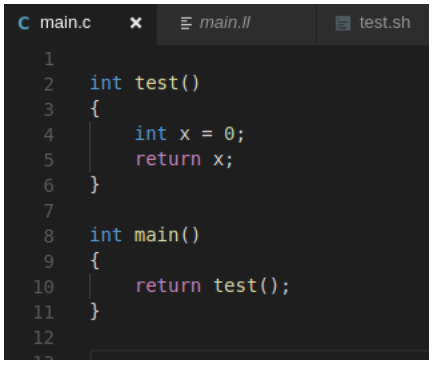
为了完成第二部分验证，我们首先需要做的是搭建一套完整的工具链，由于在之前的移植过程中我们已经有了完整的编译部分的工具链，现在我们还需要准备的就是生成 llvm 代码以及将 llvm 代码转成 python 代码的 irpy 工具。

我们先按照 https://github.com/lowRISC/riscv-llvm 教程尝试编译 llvm 工具，同时

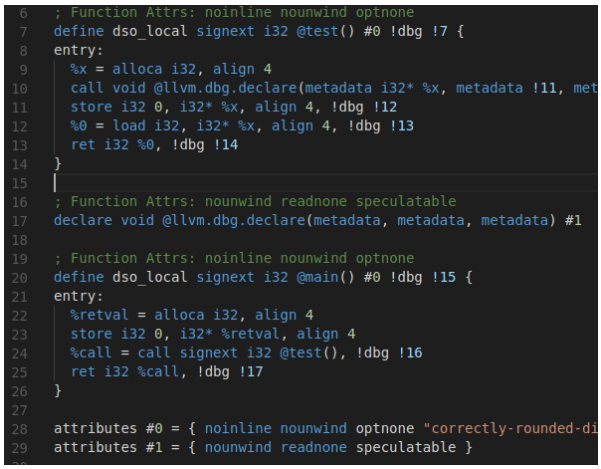
[1] Hyperkernel: Push-Button Verification of an OS Kernel

编译相对应的 irpy 工具，但是可能是由于我们使用的 clang 版本不正确（我们下载的版本是 7.0，原 hv6 使用的是 5.0），导致可以由 C 代码生成对应的 llvm 文件，但是却无法将 llvm 文件转成 python 代码。

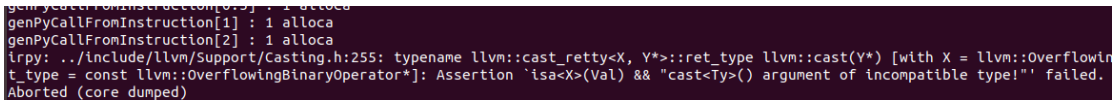
我们的测试 C 代码如下：



将其转换为对应的部分 llvm 代码如下：



但是在将上述 llvm 代码使用 irpy 工具转化为 python 的过程中出现了如下报错信息：



出现上述错误的原因不明，并且，我们发现我们需要验证的代码并没有包含具体的底层汇编代码，所以最终我们选择在 x86 的工具链下进行验证（Z3 v4.6.3.0，LLVM 5.0.0）。

3.14/50：无修改，直接验证

测试结果在 `hv6/before.txt` 中，数字 0 表示验证通过。

首先，我们将原 hv6 的测试套在新系统的代码上进行测试，没有修改 spec 也没有修改新系统的代码。一共对 50 个系统调用进行了测试，其中有 14 个系统调用通过测试，36 个

系统调用没有通过测试。考虑到我们在移植 hv6 的过程中仅仅修改了少量的系统调用，所以这次的测试结果和我们预想相符。

4. 30/50：在 spec 中修改/添加函数定义

测试结果在 `hv6/after1.txt` 中，数字 0 表示验证通过。

在验证过程中，并没有将全部的源代码编译成 llvm 并生成 python 代码，仅仅是处理了与系统调用较为相关的几个文件，具体文件如下：

```
HV6_LLS :=                          \
        $(O)/hv6/device.ll          \
        $(O)/hv6/fd.ll              \
        $(O)/hv6/invariants.ll      \
        $(O)/hv6/ioport.ll          \
        $(O)/hv6/ipc.ll             \
        $(O)/hv6/mmap.ll            \
        $(O)/hv6/proc.ll            \
        $(O)/hv6/syscall.ll         \
        $(O)/hv6/sysctl.ll          \
        $(O)/hv6/vm.ll              \
```

这样做带来的好处是只需要处理与底层汇编无关的代码，但是其存在的问题是仅仅由这部分代码并不能够生成完整的内核，比如函数 `panic` 的定义并不在其中，为此，需要在 `hv6/spec/kernel/main.py` 中添加上响应的函数说明，如此才能够正常的进行验证。

而在我们的移植过程中，我们引入了一些函数用于调试，这些函数同样不在上诉文件中，所以我们需要在 `hv6/spec/kernel/main.py` 中添加上其定义：

```python
def libs_cprintf(ctx, *args, **kwargs):
    pass
libs_cprintf.read = lambda *args: libs_cprintf

def newctx():
    ### ...
    ctx.globals['@libs_cprintf'] = libs_cprintf
```

并且，我们对于 hvm_switch 的实现也与原来 hv6 的实现不同，需要将：

```
def hvm_switch(ctx, *args, **kwargs):
    raise util.NoReturn()
```

修改为：

```
def hvm_switch(ctx, *args, **kwargs):
    pass
```

经过以上修改，我们能够成功地通过 30 个系统调用的测试。

5. 38/50：在 spec 中修改页表标识位

测试结果在 `hv6/after2.txt` 中，数字 0 表示验证通过。

X86 与 riscv 最大的差异之一,就在于四级页表标志位的不同,而在我们的验证过程中，对于页表的相关为进行了检查，这样当我们将 hv6 从 x86 移植到 riscv 下，就需要对页表项的相关标识位进行修改，同样在 spec 中也需要修改对应的常量。

具体来说，在 `hv6/spec/kernel/datatypes.py` 定义了 spec 所需的各种常量，我们需要将其中的页表项标识位代码：

```
PTE_P = BIT64(0)                                # present
PTE_W = BIT64(1)                                # writable
PTE_U = BIT64(2)                                # user
PTE_PWT = BIT64(3)                              # write through
PTE_PCD = BIT64(4)                              # cache disable
PTE_A = BIT64(5)                                # accessed
PTE_D = BIT64(6)                                # dirty
PTE_PS = BIT64(7)                               # page size
PTE_G = BIT64(8)                                # global
PTE_AVL = BIT64(9) | BIT64(10) | BIT64(11)      # available for software use
PTE_NX = BIT64(63)                              # execute disable
PTE_PERM_MASK = PTE_P | PTE_W | PTE_U | PTE_PWT | PTE_PCD | PTE_AVL | PTE_NX
```

修改为 riscv 下正确的标识位代码：

```
PTE_P = BIT64(0)                                # present
PTE_R = BIT64(1)
PTE_W = BIT64(2)                                # writable
PTE_U = BIT64(4)                                # user
# PTE_PWT = BIT64(3)                              # write through
```

```
# PTE_PCD = BIT64(4)                                # cache disable
# PTE_A = BIT64(5)                                  # accessed
# PTE_D = BIT64(6)                                  # dirty
# PTE_PS = BIT64(7)                                 # page size
# PTE_G = BIT64(8)                                  # global
# PTE_AVL = BIT64(9) | BIT64(10) | BIT64(11)  # available for software use
PTE_X = BIT64(3)                                # execute disable
PTE_PERM_MASK = PTE_P | PTE_W | PTE_U | PTE_X | PTE_R
```

完成了上诉修改，可以通过 38 个系统调用的验证。

6. 43/50：修改 spec 的相关页表，修改 hv6 相关页表函数

测试结果在 `hv6/after3.txt` 中，数字 0 表示验证通过。

riscv 与 x86 对于一个页表项中的实地址提取以不相同，对于 x86，一个页表项有 64 位，其中低 12 位用于描述各种标识位（包括是否可写，访问标识等），高 52 位用于标识物理地址（实际上并非全部如此，某些情况下只有 48 位地址线），而对于 riscv，一个页表项的低 10 位用来描述各种标识位，剩下的高 54 位标识物理地址，这个差异导致了在 spec 中某些从页表项中提取物理地址的验证是错误的，具体来说，需要在 `hv6/spec/kernel/spec/specs.py` 中修改如下代码：

```
def sys_protect_frame(old, pt, index, frame, perm):
    cond = z3.And(
        # ...
        # the entry in the pt must be the frame
        z3.Extract(63, 40, z3.UDiv(old.pages_ptr_to_int,
                                   util.i64(dt.PAGE_SIZE)) + frame) == z3.BitVecVal(0, 24),
        z3.Extract(39, 0, z3.UDiv(old.pages_ptr_to_int, util.i64(
            dt.PAGE_SIZE)) + frame) == z3.Extract(51, 12, old.pages[pt].data(index)),
        #...
    )
```

修改为：

```
def sys_protect_frame(old, pt, index, frame, perm):
    cond = z3.And(
        # ...
        # the entry in the pt must be the frame
```

```
        z3.Extract(63, 40, z3.UDiv(old.pages_ptr_to_int,
                                   util.i64(dt.PAGE_SIZE)) + frame) == z3.BitVecVal(0, 24),
        z3.Extract(39, 0, z3.UDiv(old.pages_ptr_to_int, util.i64(
            dt.PAGE_SIZE)) + frame) == z3.Extract(49, 10, old.pages[pt].data(index)),
        # ...
    )
```

在 hv6/spec/kernel/spec/specs.py 中还有几个类似的函数也需要修改。

此外，令我们在实验中感到困惑不解的是，原 hv6 从页表项中提取物理地址并不是如我们想象的那样提取高 52 位，而是提取中间的 40 位（低 12 位到 51 位）再左移 12 位：

```
// 原 hv6，include/uapi/machine/mmu.h
#define PTE_ADDR(pte) ((physaddr_t)(pte)&BITMASK64(51, 12))
```

而我们将 hv6 移植到 riscv 是直接提取高 54 位作为物理地址，为了和 spec 保持一致，我们修改了新系统的相关代码，将：

```
// 新 hv6，include/uapi/machine/mmu.h
#define PTE_ADDR(pte) (((physaddr_t)(pte) >> PTE_PFN_SHIFT) << PAGE_SHIFT)
```

修改为：

```
// 新 hv6，和 spec 保持一致，include/uapi/machine/mmu.h
#define PTE_ADDR(pte) ((((physaddr_t)(pte) >> PTE_PFN_SHIFT) << PAGE_SHIFT)&BITMASK64(51, 12))
```

经过上述修改，我们总共通过了 43 个系统调用的验证。

7. 46/50：增加 spec 中对于 XWR 标志位的约束

测试结果在 hv6/after4.txt 中，数字 0 表示验证通过。

原 hv6（x86 下）和新的 hv6（riscv 下）都采用 4 级页表，但是 riscv 平台下对于 4 级页表中间的部分存在约束：页表的非叶子节点的执行位（X）、可写位（W）和可读位（R）必须均为 0。而在 x86 下不存在这个约束，为了符合 riscv 的规定，在我们的新系统中规定了非叶子节点的 XWR 为必须为 0，为此，需要修改 spec 使其符合 riscv 的规定。

首先，在 hv6/spec/kernel/datatypes.py 添加对于 XWR 的约束：

```
PTE_XWR_MASK = PTE_W | PTE_X | PTE_R
```

然后，在 hv6/spec/kernel/spec/specs.py 中有三个函数（sys_alloc_pdpt，sys_alloc_pd，sys_alloc_pt）需要加上 XWR 的约束，以 sys_alloc_pdpt 为例，将：

```
def sys_alloc_pdpt(old, pid, frm, index, to, perm):
    return alloc_page_table(old, pid, frm, index, to, perm,
                            dt.page_type.PAGE_TYPE_X86_PML4, dt.page_type.PAGE_TYPE_X86_PDPT)
```

修改为：

```
def sys_alloc_pdpt(old, pid, frm, index, to, perm):
    return alloc_page_table(old, pid, frm, index, to, perm & (dt.MAX_INT64 ^ dt.PTE_XWR_MASK),
                            dt.page_type.PAGE_TYPE_X86_PML4, dt.page_type.PAGE_TYPE_X86_PDPT)
```

进过上诉修改，总共通过了 46 个系统调用的验证。

8. 50/50：修改 spec 中 iommu 的页表标识位

测试结果在 hv6/after5.txt 中，数字 0 表示验证通过。

在目前为止，只有与 iommu 相关的几个函数还没有通过验证，由于 riscv 中并没有对于 iommu 的相关支持，所以理论上这部分代码可以删除掉而无需验证，但是由于担心删除代码会导致其他更严重的问题，而且 iommu 的验证代码对于写新的验证代码有一定的参考意义，所以我们最终选择保留了 iommu 的相关验证。

与 iommu 相关的几个系统调用实际上是在设置 DMA 的页表，从代码中可以看出 iommu 和用户程序使用的页表在结构上是相同的，所以我们只需要修改 iommu 所使用的页表项标识位即可。

将 hv6/spec/kernel/datatypes.py 中的：

```
DMAR_PTE_W = BIT64(1)     # Write
```

修改为：

```
DMAR_PTE_W = BIT64(2)     # Write
```

到此为止，我们已经完成了全部系统调用的验证。

9. 51/50：在 spec 中添加新系统调用的约束

我们在将 hv6 从 x86 移植到 riscv 的过程中，由于虚拟机的移除，导致我们需要对内核代码也建立页表映射，而 fork 需要建立和自己页表完全相同的页表，基于以上原因，我们需要增加一个系统调用 `sys_map_page`，其代码实现在 `hv6/vm.c` 中。

为了能够使用 spec 的框架能够验证我们新增的系统调用，我们需要增加两部分代码。

第一部分是对于系统调用的输入参数的描述，在 `hv6/spec/kernel/syscall_spec.py` 中：

```
class SyscallSpec(object):
    def sys_map_page(self):
        pid = util.FreshBitVec('pid', dt.pid_t)
        frm = util.FreshBitVec('from', dt.pn_t)
        index = util.FreshBitVec('index', dt.size_t)
        pa = util.FreshBitVec('pa', dt.size_t)
        perm = util.FreshBitVec('perm', dt.pte_t)
        from_type = util.FreshBitVec('from_type', dt.uint64_t)
        return (pid, frm, index, pa, perm, from_type)
```

第二部分则是对于该系统调用的详细约束，主要包括对于输入数据的检查和对系统状态的修改，代码在 `hv6/spec/kernel/spec/specs.py` 中。

最后在 `hv6/spec/kernel/main.py` 的 `_syscalls` 变量的最后添加 `sys_map_page`，然后执行命令：`make hv6-verify -- -v --failfast HV6.test_sys_map_page` 即可验证该系统调用的正确性。

总结

我们将 hv6 从 x86 平台上移植到了 riscv 平台上，通过修改相关的内核代码与约束，成功地验证了全部的 50 个系统调用，并且对于我们新添加的系统调用也做了验证。

但是，验证通过并不代表我们的新 hv6 是正确的，只能说明它和约束所描述的表现一致并且也满足不变量的要求，但是我们对于不变量的约束是否足够以及约束所描述的表现是否是正确的尚无定论。

在我们修改验证的过程中，使用了“二分删除代码”调试方法，也既：对于一个系统调

用，删除全部的 spec 和 C 代码，查看验证是否通过，如果通过则添加一半的代码查看是否还是通过，依次进行直到找到错误点。但是此方法并不一直有效，原因在于一个系统调用对于操作系统状态的修改应该是一个连续的整体，是不可被分割的。