目录

[HV6设计 1](#_Toc515523184)

[移植考量 2](#_Toc515523185)

[移植工作 3](#_Toc515523186)

[1. 准备工作 3](#_Toc515523187)

[2. Boot Loader替换 3](#_Toc515523188)

[3. 建立页表 4](#_Toc515523189)

[4. 系统调用 5](#_Toc515523190)

[5. 进程管理移植 7](#_Toc515523191)

# 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 的适合，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

## 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。

## 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

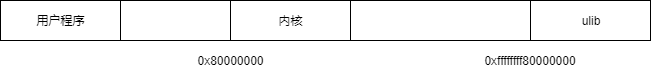
可以看到，上述存在两次编译过程，尤其是因为BBL需要和内核的二进制文件一起编译，导致内核的二进制文件被放置的位置是无法预知的，所以，在编译内核的时候必须编译成不使用绝对路径而全部使用相对路径的代码，在编译参数中添加-mcmodel=medany即可实现，另外，内核中某些全局变量也需要修改，比如：

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];
3. **void** init\_syscalls()
4. {
5. **int** i;
6. **for**(i = 0; i < NR\_syscalls; i ++) {
7. syscalls[i] = sys\_nop;
8. }
9. syscalls[SYS\_map\_pml4] = sys\_map\_pml4;
10. syscalls[SYS\_map\_page\_desc] = sys\_map\_page\_desc;
11. }

在我们移植之后，hv6的内存布局如下：



## 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：

1. **static** **void** setup\_kernel\_map(pid\_t pid)
2. {
3. assert(KERNSTART % PAGE\_SIZE == 0, "KERNBASE % PAGE\_SIZE must be 0");
4. **uintptr\_t** va;
5. pte\_t perm = PTE\_P | PTE\_W | PTE\_R | PTE\_X;
6. **for**(va = KERNSTART; va < KERNEND; va += PAGE\_SIZE) {
7. pn\_t pt = page\_walk(pid, va);
8. assert((va - (**uintptr\_t**)pages) % PAGE\_SIZE == 0, "(va - pages) % PAGE\_SIZE != 0");
9. pte\_t\* pt\_page = get\_page(pt);
10. pt\_page[PT\_INDEX(va)] = ((va >> PAGE\_SHIFT) << PTE\_PFN\_SHIFT) | perm;
11. **enum** page\_type type = get\_page\_desc(get\_proc(pid)->page\_table\_root)->type;
12. **if** (type == PAGE\_TYPE\_X86\_PML4 || type == PAGE\_TYPE\_X86\_PDPT || type == PAGE\_TYPE\_X86\_PD || type == PAGE\_TYPE\_X86\_PT) {
13. pt\_page[PT\_INDEX(va)] = ((va >> PAGE\_SHIFT) << PTE\_PFN\_SHIFT) | perm | PTE\_U;
14. }
15. }
16. }

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。

## 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：

1. **void**\*\* ulib\_syscalls;
2. **int** is\_user\_mode;
3. **uintptr\_t** cr3\_value;
5. **void** ulib\_init(**void**\* \_ulib\_syscalls[], **void** (\*fn)())
6. {
7. ulib\_syscalls = \_ulib\_syscalls;
8. is\_user\_mode = 0;
9. 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：

1. #define SYSCALL(sym)        \
2. .global sys\_##sym;          \
3. sys\_##sym:                  \
4. ld      a7, is\_user\_mode; \
5. bnez    a7, ecall\_sys\_##sym; \
6. j       direct\_sys\_##sym; \
7. ecall\_sys\_##sym:            \
8. li      a7, SYS\_##sym;  \
9. ecall;                  \
10. ret;                    \
11. direct\_sys\_##sym:           \
12. ld      a7, ulib\_syscalls;  \
13. addi    a7,a7,8\*SYS\_##sym;  \
14. ld      a7, 0(a7);          \
15. jr      a7;                 \
17. SYSCALL(map\_page\_desc)
18. 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。

## 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：

1. **struct** context {
2. **uintptr\_t** ra;
3. **uintptr\_t** sp;
4. **uintptr\_t** s0;
5. **uintptr\_t** s1;
6. **uintptr\_t** s2;
7. **uintptr\_t** s3;
8. **uintptr\_t** s4;
9. **uintptr\_t** s5;
10. **uintptr\_t** s6;
11. **uintptr\_t** s7;
12. **uintptr\_t** s8;
13. **uintptr\_t** s9;
14. **uintptr\_t** s10;
15. **uintptr\_t** s11;
16. };

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：

1. **void** riscv\_hvm\_copy(**void** \*\_dst, **void** \*\_src, pid\_t pid)
2. {
3. hvm\_t\* dst = (hvm\_t\*)\_dst;
4. hvm\_t\* src = (hvm\_t\*)\_src;
5. \*dst = \*src;
6. dst->pid = pid;
8. dst->tf = (**struct** trap\_frame \*)(get\_page(get\_proc(dst->pid)->stack) + PAGE\_SIZE) - 1;
9. \*(dst->tf) = \*src->tf;
10. dst->tf->gpr.a0 = 1;
12. dst->tf->gpr.sp = src->tf->gpr.sp;
13. dst->context.sp = dst->tf;
14. dst->context.ra = forkret;
15. }