Hyperkernel: 一键运行的操作系统内核验证

Hyperkernel: Push-Button Verification of an OS Kernel

背景介绍

操作系统是一个计算机系统中非常关键的组成部分，它提供了对于各种硬件资源的抽象。同时操作系统的bug会给系统带来很大的正确性和安全性问题。然而操作系统的复杂程度使得人力无法处理。虽然一些现有的工作尝试对复杂的操作系统内核的正确性进行证明，但是耗费巨大的人力和物力，不是一个能广泛应用的方式。

该论文提出了一种较低运算量的方式来实现一个正确的操作系统。开发了基于xv6的 Hyperkernel操作系统，并对其正确性进行了形式化的验证。Hyperkernel使用了内核与用户分离的地址空间，其实现避免了次数无上限的循环和递归。Hyperkernel使用C语言开发，但在验证时，是将C语言编译到LLVM层面再进行验证的。

系统设计

接口定义

接口的定义必须在操作系统的易用性和证明的自动化之间做权衡。一方面，操作系统内核维护一系列数据结构来管理进程、虚拟内存、IO设备等各种资源，其接口需要一定的规范和约束来描述其正确的行为；另一方面，这些接口必须能以一种可自动证明的方式来实现。为此Hyperkernel的所有接口的被设计为有限的：所有的系统调用、中断和异常的处理都不能含有没有次数上限的循环或递归，使得每个处理程序的运行结果都能用一个有限长的表达式所表示。这里称这样的接口为有限的。

Hyperkernel的接口规范是基于已有的POSIX规范设计的，在一些为了简化验证的地方做了修改。例如，在Unix的设计中，每个进程都维护了一个文件描述符表，表中的一个槽指向了系统全局的文件表；在POSIX规范中，的定义是，复制一个文件描述符，放在数字最小的可用的文件描述符槽中，并返回这个数字。

POSIX的dup调用不是有限的，找到最小可用文件描述符需要从文件描述符表头开始，从头至尾扫描直到找到第一个可用槽为止。在最坏情况下，需要遍历整个文件描述符表。而该表的大小随着系统规模的增大会不断的增大，使得运行结果不能用一个有限长的表达式所表示。

Hyperkernel将POSIX的接口改为。这个接口要求用户自己选择一个新的文件描述符，若该文件描述符可用，则创建复制，并返回成功信息；否则返回失败信息。实现该接口只需要检查该文件描述符是否可用，可以在有限步数内完成，使得运行结果表达式的长度有限，易于形式化验证。

这样的设计也会降低操作系统的易用性。比如在系统调用中，用户往往也也不知道哪些文件描述符可以使用，除非用户自己记录下哪些文件描述符已经被使用；这时用户程序开发者往往会开发一些管理文件描述符的库，而这些库的正确性同样非常重要，而且包含没有次数上限的循环或递归，无法像这里这样简单地验证。

虚拟内存管理

在x86上常见的虚拟内存管理机制中，内核空间和用户空间在同一个地址空间，内核占上半空间，用户占下半空间。虚拟地址映射通常不是内射，两个不同的虚拟地址可能映射到同一个物理地址。而别名问题会给形式化证明带来很大的困难，更何况虚拟地址与物理地址的映射关系还可能会随着程序的运行而改变。为了避免给验证带来过多的困难，Hyperkernel所运行的内和空间与用户空间是分离的。为了实现分离的地址空间，Hyperkernel利用了x86上的硬件虚拟化技术（Intel VTx和AMD-V）：操作系统内核运行在host模式，而用户程序运行在guest模式，使用不同的页表。

形式化验证

验证的自动化

使用可满足性模理论（satisfiability modulo theories，SMT）求解器可以完成自动化定理证明。Hyperkernel使用Z3 SMT求解器来完成自动验证工作。

程序语言的语义

Hyperkernel使用C语言开发。众所周知，C语言的形式化表示非常复杂，要将C语言的语义建立符号化模型是非常困难的，因为C语言的指针运算和内存访问等底层操作很难用形式化语言精确地表示。更严重的是，为了让编译器实现更好的发掘高性能，C语言的语义在很多地方是未定义的。一些研究者声称，“不存在C语言标准可以保证不崩溃的C程序”。

验证程序在LLVM层进行验证。LLVM中间表示（internal representation, IR）具有比C语言简单得多的语义，且仍然足够高级到隐藏机器的细节。整个工作流程是先将C语言代码编译成LLVM中间表示，用验证程序对LLVM中间表示进行验证，最后将LLVM中间表示编译成机器码来部署、运行。

SMT求解器会从LLVM中间表示提取符号化模型。在这个过程中，SMT求解器会检查LLVM中所有的未定义行为，如可能的除零、使用未初始化的变量等。

系统规范与模型

系统的规范包括如下几个部分：

1. 系统状态规范。系统状态包括一系列定宽整数、定长映射表。比如：
   1. 整数current表示当前正在运行的进程编号；
   2. 定长映射表proc\_fd\_table表示每个进程的文件描述符表，将一个进程和一个文件描述符映射一个文件；
   3. 定长映射表proc\_nr\_fds将一个进程映射到该进程所使用的文件描述符个数；
   4. 定长映射表file\_nr\_fds将一个文件映射到引用该文件的文件描述符个数。
2. 状态转换规范。大多数系统调用的规范都遵循类似的模式：首先检查参数的有效性，若参数有效将原系统状态转换到一个新的系统状态；否则返回错误码，系统状态不变。每个系统调用将得到一个有效性表达式和一个新状态。以为例，其有效性表达式为

其状态转换表达式为

其中

状态的其它部分不变

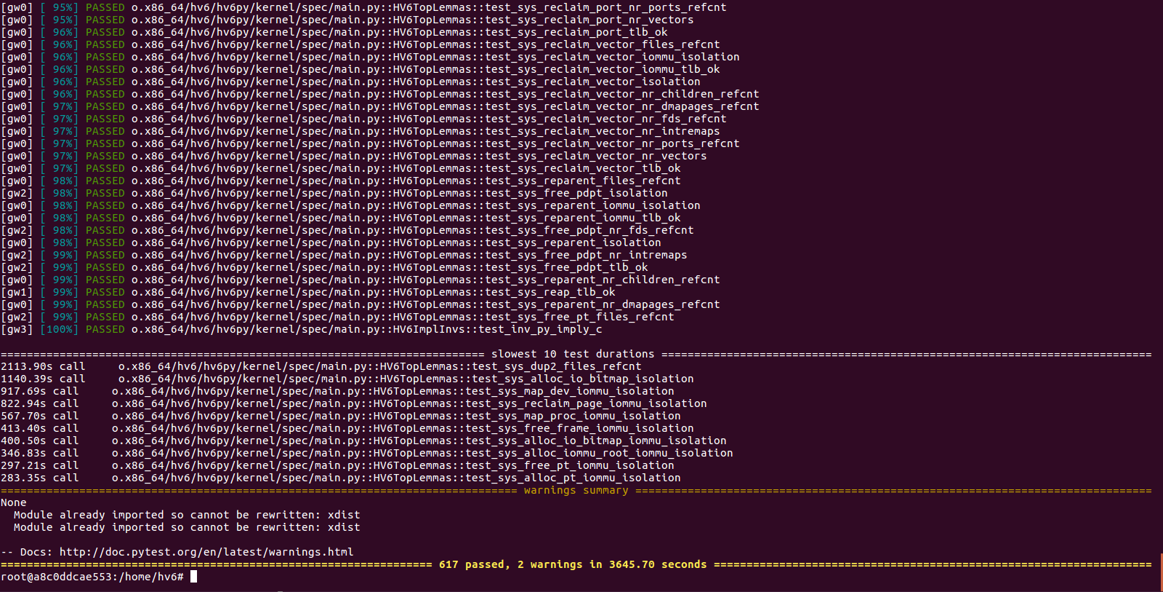
1. 声明式规范。以上两种规范还没有涉及一些实现细节，而为了发现实现细节中如内存布局、计数器溢出等bug，再对状态机加一些约束。比如说对于任意，都有
2. 不变式。任何一个系统调用在运行时都会首先检查用户传入的参数的有效性，因为用户是不可信任的，但是系统状态的有效性却往往是被默认的，而且额外的检查也带来额外的性能开销。为了保证系统状态的有效性，还需要规定一些在系统任何状态下都成立的不变式。在验证时，得到每个接口的调用结果之后，要检查不变式是否仍然成立。比如对当前正在运行的进程号，有如下不变式

实验

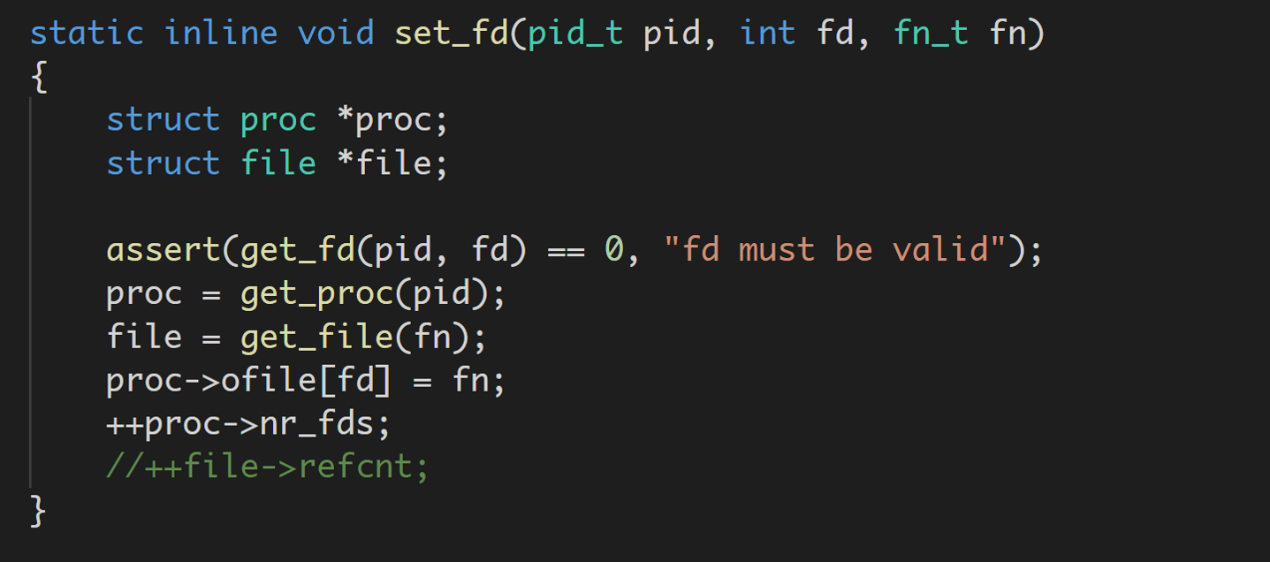
经过验证与修复，除了初始化和一些胶水代码以外，Hyperkernel在LLVM层已经做到了完全正确，只要LLVM编译器和所运行的硬件符合其标准规范，那么Hyperkernel就不会出现不符合规范的错误。

xv6在2016年到2017年上半年一共修复了十个bug。其中有两个与锁相关，因为Hyperkernel运行在单核机器上，不会出现这类bug；还有三个在ELF的装载上，由于Hyperkernel对ELF的装载时在用户空间完成的，尽管这部分代码可能有bug，但不对内核产生安全性危害。而剩下的五个bug都可以在验证过程中被发现。

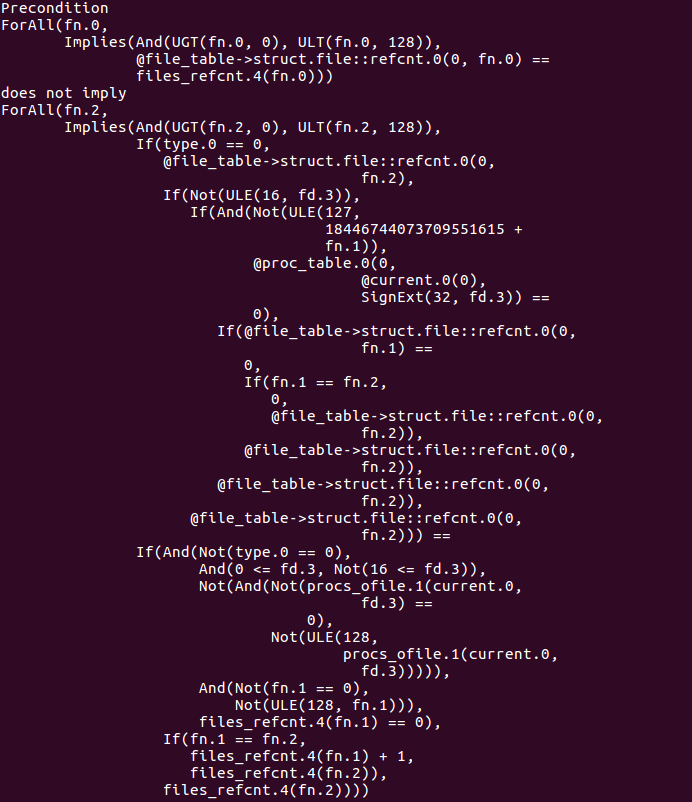
我们重复了这一实验，用4核Intel i5-6300运行了一个小时，验证了Hyperkernel中617条定理。



修改代码，制造bug，可以发现SMT求解器能成功的找出bug



SMT求解器发现文件的引用计数器的值与实际引用数不相等



结论

Hyperkernel是一个经过高度自动化、低计算量的形式化验证的操作系统内核。通过SMT求解器自动提取LLVM IR的语义并建模来达到证明的自动化；通过有限接口设计、利用硬件虚拟化技术等思想来降低操作系统内核的复杂性，从而简化验证。Hyperkernel为正确的操作系统内核和其他底层软件的设计、以及系统与自动化验证的协同设计提供了一个可靠的方向。

这种设计从某种程度上降低了操作系统的易用性，实际生产环境中的系统通常需要有无上限的循环或递归，这使得Hyperkernel给出的解决方案只能应用于操作系统的一部分。另外，操作系统中常见的一类bug出在多核处理器引起的数据竞争上，Hyperkernel是运行在单核机器上的，对于这类bug也没有提供很好的解决方案。