操作系统课程实验报告

Ucore

Lab1：bootasm.S和bootmain.c实现了引导扇区中OS Boot loader的功能。从起始物理地址实现跳转。进入保护模式，应用分段存储管理机制。查看段寄存器中CPL字段值即可知道特权级。Elf头中描述了文件的组织，program header则描述了程序执行相关的目标文件结构信息。IDT负责描述中断，CPU通过IDT来寻找IDT的起始地址。其中，LIDT只能在特权0级别执行。相关描述在mmu.h文件中。又区分了任务门、中断门、陷阱门。通过iret恢复被中断执行的程序。

Lab2:初始化页表，通过起始地址建立空闲页链表。分配二级页表、删除二级页表。实现在pmm.c文件中。主要完成虚拟页与物理页之间的对应关系，使用Page数据结构来表示4kb的物理页。区分已经使用的物理页与空闲页，通过双向链表管理空闲页。Ucore使用二级页表管理，需要20kb空间。二级页表限制用户权限。通过引用数目确定是否收回页表。

Lab3:处理虚存。实现虚存的分配和收回等。应用先进先出策略，维持队列。完成按需分配功能。产生Page fault时，可获得虚存的地址，进一步判断是否需要换入换出操作。Mmap中存放有vma信息。如果权限争取，分配空的内存页给产生Page fault的查询。没有空页时，则需要释放队列头部的页。值得注意，换出操作时，需要确保磁盘上有物理页，保证换出有地方存。

Lab4:初始化进程控制块，pid赋值为-1，利用do\_fork激活进程。将进程信息复制，增加进程pid，添加入进程表，并唤醒进程。Proc\_run函数判断是否为新进程，如果是则将当前进程进行保留，将新进程的信息加载入栈，切换进程相关内容。用户进程需要维护自己的内存空间。进程创建过程中最重要的是写好进程控制块。里面包含了是否可以被cpu进行调度，以及运行时间、运行级别、内存管理信息、中断指针等关键信息。Ucore采用内核栈分配在不同位置的方法，内核态进程又共享内存，容易发生溢出污染。调度函数通过查找进程队列，决定是否需要进行进程切换。

Lab5:主要是实现系统调用，完成用户进程管理。进程管理的实现要比lab4复杂多。主要是进程管理和内存管理。不同的进程有不同的页表需要管理。调用kernel\_thread创建子进程。用户进程的执行代码和数据相互隔离。新进程先清空mm（用户态内存空间），然后再度ELF加载文件。重新设置进程中断帧，使之能够响应中断。Sys.exit操作先进入内核空间，释放用户空间，父进程处于等待子进程状态，由父进程对子进程进行回收。直到子进程释放了内核堆栈和进程控制块，回收完成。系统调用，由用户态到内核态，需要调用idt\_init初始化中断描述，调用中断门。所有系统调用操作均会使用syscall函数。调用时先保存现场到tapframe中，便于后续的恢复工作开展。

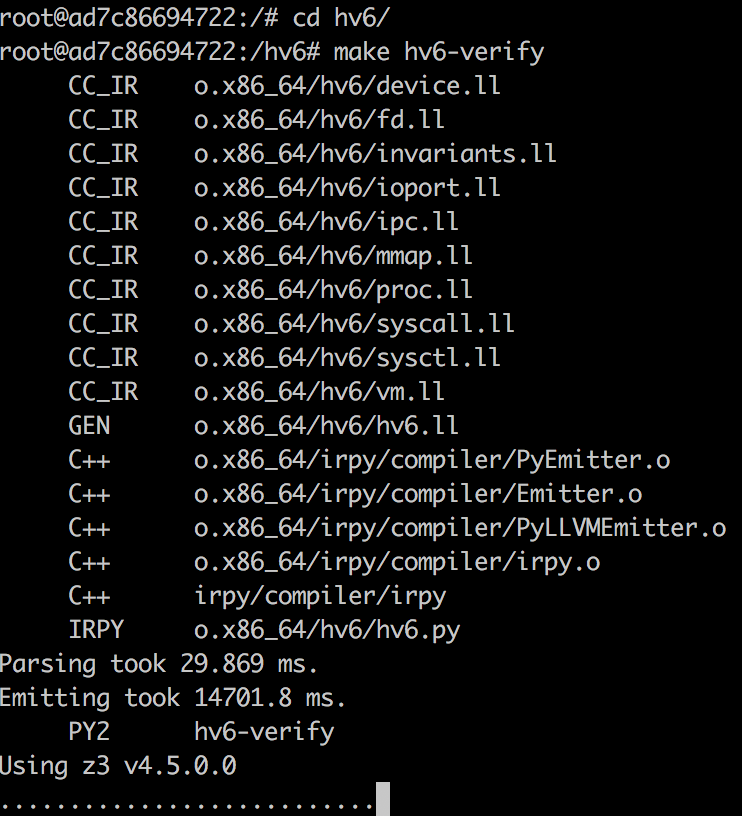
Lab6:调度实现了进程对CPU资源的抢占。当资源得不到满足时，就应该应用调度，主动放弃CPU的占用，给别的进程执行。Ucore中内核的资源是不能被抢占的。调度主要完成从就绪进程中移出、当前进程进入就绪进程等操作。使用time\_slice来感知进程的变化，选择合适的进程执行。进程的运行队列通过链表组织，完成队列初始化、删除、插入等操作。链表可以找到proc\_struct数据结构，其中记录着该进程的剩余时间、优先级等。RR算法采取轮询的方式调度，stride通过计算stride的大小并比较，来选取下一个将要执行的进程。

Lab7:通过信号量等方法实现同步互斥。在等待队列中选取就绪进程执行。以时钟为基本调度单位，等待函数最终的目的是让等待进程关联的等待项同信号量绑定，实现信号量控制进程的调度。信号量主要涉及P、V操作，先关中断再操作信号量value的值。管程则是相对局限的实现方式。只有条件得到满足，进程才能从睡眠中释放继续执行。管程中执行的进程数为1。管程相比于信号量，更为严格，不易发生死锁。

Lab8:文件系统的实现主要涉及管理目录、数据、元数据。包括超级块、inode、dentry、file。通过超级块找到inode，再通过inode找到dentry，进而访问到文件。在进程初始化过程中还需要添加入文件指针信息。Ucore内核通过调用接口实现对文件和目录的访问。与block一样，文件在文件抽象层inode中也会记录被引用的数量等信息。在SFS中使用块为基本单位，可从查询第0块superblock开始依次查询，直到目标文件。设备与文件系统的连接是通过双向链表实现的。读写设备文件，需要驱动函数的支持。文件的操作需要每一层函数层层调用。

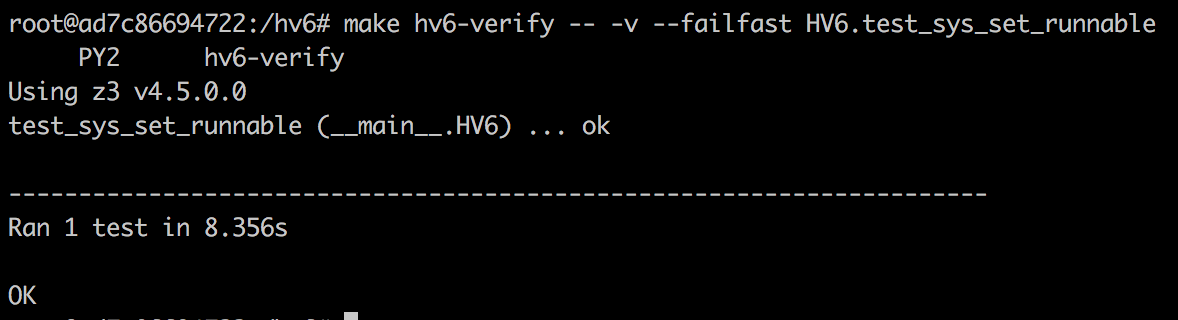
Hyperkernel：

使用docker pull mancanfly53373931/docker\_hv6:hv6获取实验镜像。然后运行docker，docker run -t -i -d mancanfly53373931/docker\_hv6:hv6 /bin/bash，获取hv6，git clone https://github.com/mancanfly-1/hv6.git，运行测试，验证。将内核构建到LLVM IR中，使用Irpy将内核转换到Python，调用/spec/main.py。实验结果如图：

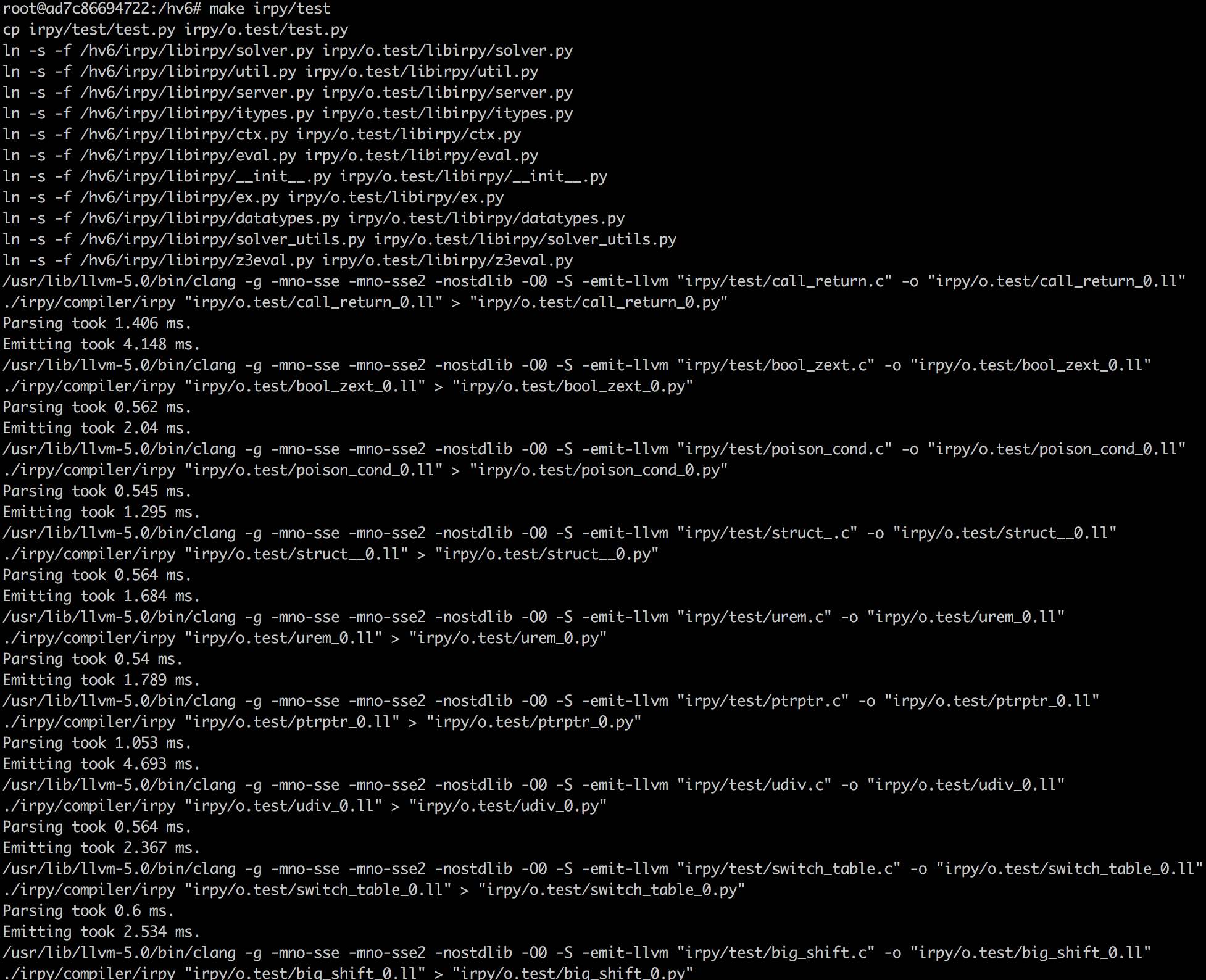


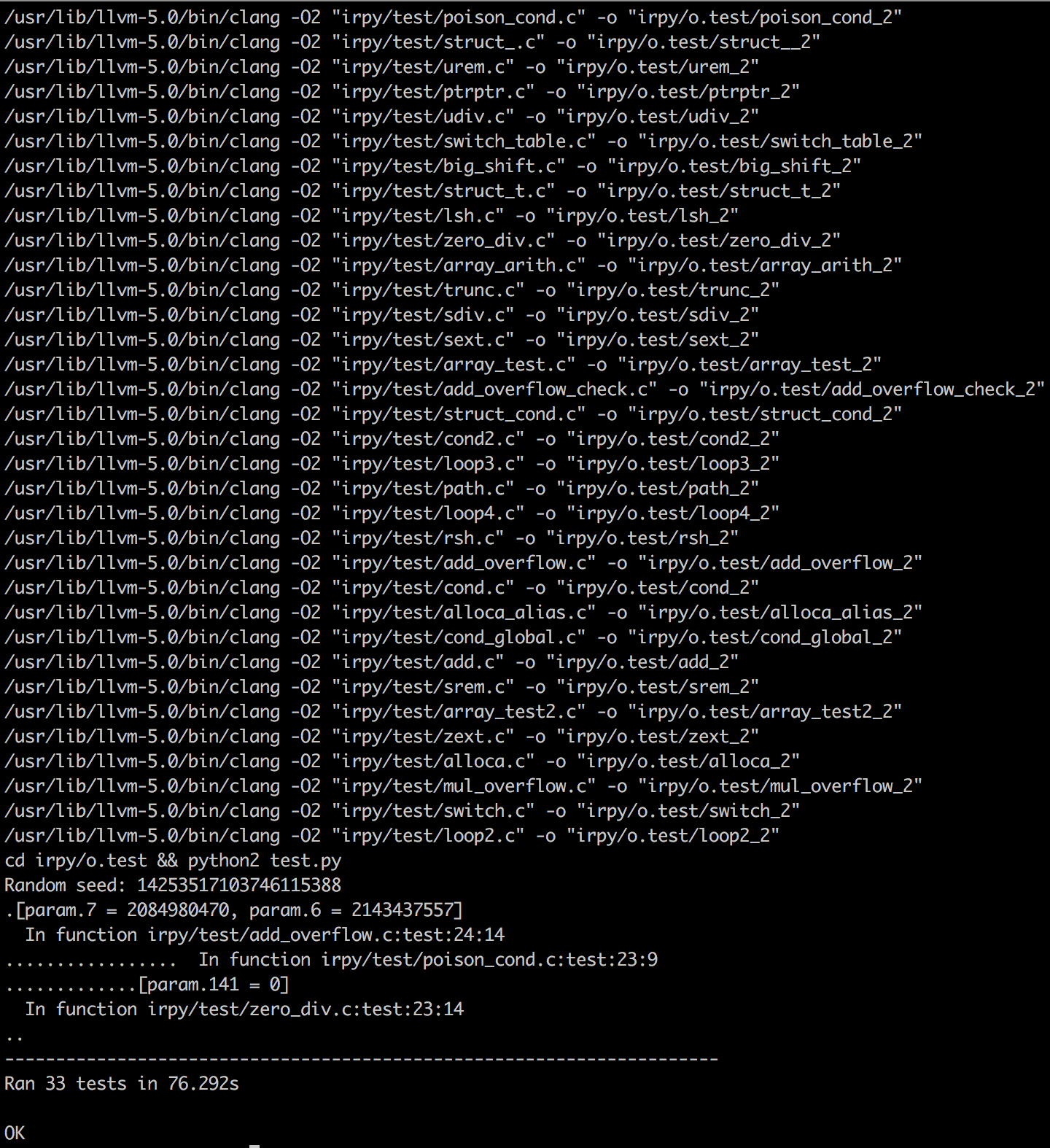
其中，由于是单核运行，z3进度非常的慢，运行超过两个半小时还没有结果。

尝试单个测试，结果如下图所示：



运行Irpy测试套件，将该套件符号执行结果与运行C代码进行比较。实验结果如下图所示：





分析源码，对比测试过程主要在下面过程中实现，对于z3如何实现形式化描述还不甚清楚：

def test\_preempt(self):

with self.assertRaises(util.NoReturn):

self.ctx.call('@preempt')

pid = util.FreshBitVec('pid', dt.pid\_t)

newstate = spec.switch\_proc(self.state, pid)[1]

self.\_prove(z3.Exists([pid], spec.state\_equiv(self.ctx, newstate)))

def test\_fault(self):

with self.assertRaises(util.NoReturn):

self.ctx.call('@fault')

pid = util.FreshBitVec('pid', dt.pid\_t)

s = self.state.copy()

s.procs[s.current].killed = z3.BoolVal(True)

newstate = spec.switch\_proc(s, pid)[1]

self.\_prove(z3.Exists([pid], spec.state\_equiv(self.ctx, newstate)))

def \_syscall\_generic(self, name):

args = syscall\_spec.get\_syscall\_args(name)

res = self.ctx.call('@' + name, \*args)

cond, newstate = getattr(spec, name)(self.state, \*args)

model = self.\_prove(z3.And(spec.state\_equiv(self.ctx, newstate),

cond == (res == util.i32(0))),

pre=z3.And(self.\_pre\_state, z3.BoolVal(True)),

return\_model=INTERACTIVE)

if INTERACTIVE and model:

from ipdb import set\_trace

set\_trace()