JudgePenguin

基于Linux的应用程序稳态测试系统

致理-信计01 单敬博 2020012711

2023/6/10

背景回顾

- 在编程竞赛中，希望对用户程序进行**稳定、准确、安全**的测试
  - 用户程序通常是单进程用户态程序，运行过程中受到时间、空间限制
- 测试系统的任务
  - 为用户程序提供**输入**，收集**输出**
  - 尽可能**准确**测量用户程序的**时间**、**空间**使用情况
  - 防止用户程序进行创建线程、连接网络、破坏系统等**非法或恶意的行为**

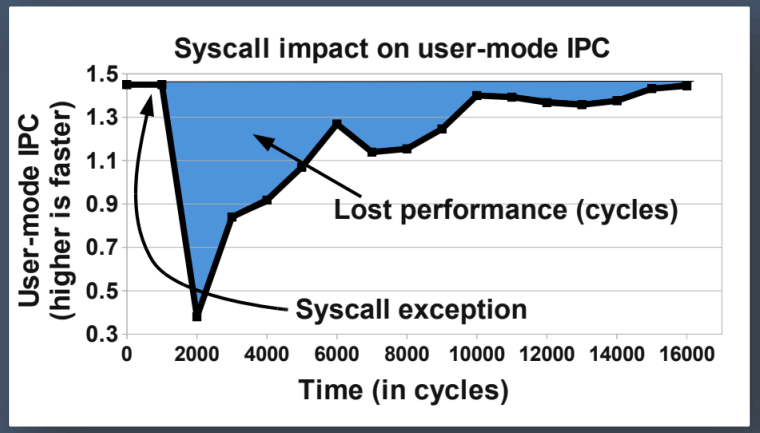
现有测试系统的不足

- 在系统中直接运行：Lemon, Cena, Arbiter, …
  - 无法有效防范用户程序的攻击 (直接在上下数级目录中查找答案文件)
- 基于docker/sandbox： LOJ, TUOJ, UOJ, …
  - 受虚拟化技术影响，时间测量结果波动较大 (🍪的误差可能高达100%!)

#	用户	题目	语言	状态	分数
57706	[redacted]	A	python3	Time Limit Exceeded	90
57706	[redacted]	A	python3	Accepted	100

现有测试系统的不足

- OS中断与调度
  - 用户程序执行过程中OS仍会收到来自外设、网络、时钟等的中断
  - 用户程序也可能因为OS调度而暂停执行
  - 处理中断、任务切换不增加user time，但会对cache及TLB造成影响



from FlexSC

现有测试系统的不足

- 内存分配不连续
  - 常见OS的内存分配结果难以预测，用户程序访存时cache命中率不同
- 与其他进程共享资源
  - 多核OS中其他进程会随机地对内存、L_3 cache等共享资源产生难以预料的影响

现有测试系统的不足

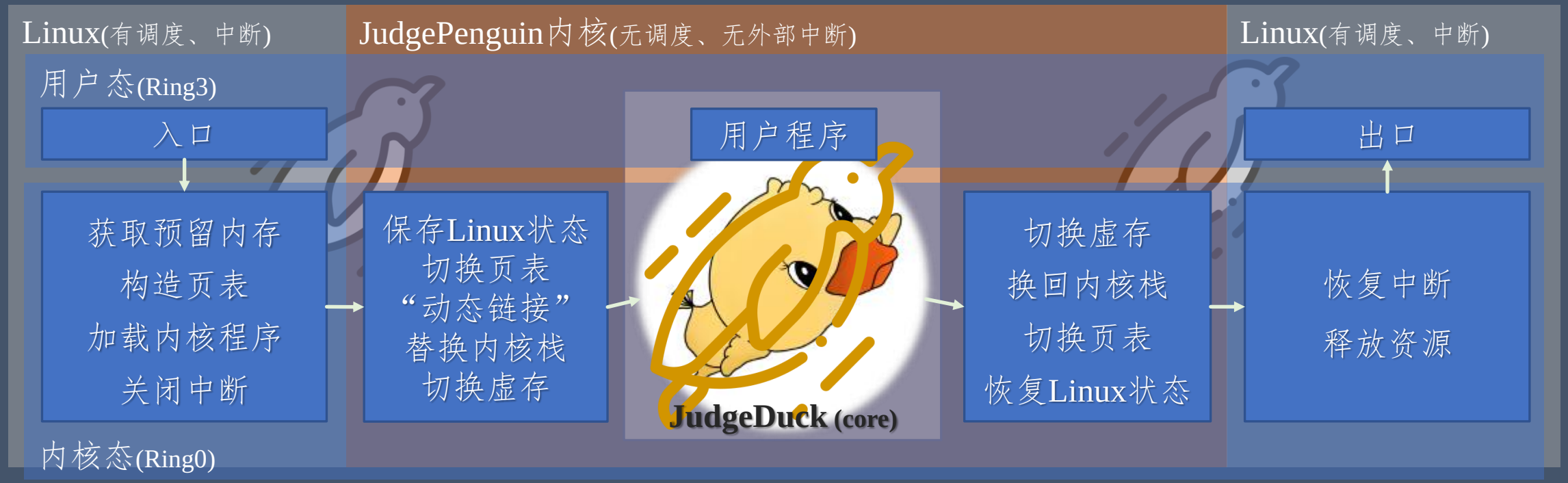
- 自研操作系统：JudgeDuck-OS
  - 屏蔽全部外部中断；为用户程序分配连续而确定的内存；用户程序独占全部系统资源
  - 硬件驱动需要自行编写，依赖特定硬件 (JudgeDuck-OS依赖e1000网卡)

项目设计

- 稳定无干扰的环境只需在运行被测用户程序时提供
- 加载用户程序与数据、处理测试结果可以在任意环境中进行

- 借助**Linux内核模块**机制：
  - 进行测试时阻塞Linux，在JudgePenguin内核中实现稳定、准确测试
  - 测试完成后恢复到Linux，在Linux中完成后续处理

整体架构


Linux(有调度、中断)
JudgePenguin内核(无调度、无外部中断)
Linux(有调度、中断)
用户态(Ring3)
入口
获取预留内存
构造页表
加载内核程序
关闭中断
保存Linux状态
切换页表
“动态链接”
替换内核栈
切换虚存
用户程序
JudgeDuck (core)
切换虚存
换回内核栈
切换页表
恢复Linux状态
出口
恢复中断
释放资源
内核态(Ring0)

已经实现的功能

- 通过Linux内核模块关闭中断，实现单核独占运行
- 在Linux中预留物理内存，在内核模块中获取并使用
- 构造x86-64 4级页表，实现页表与内核栈切换，修复内存基址
  → 在地址确定的稳定环境中运行包含libc的内核态程序
  运行结束后能够顺利返回Linux
- 完成了JudgeDuck-OS (core) 的部分迁移 (暂时还不能正确测试用户态程序)
- 通过gdb远程调试在QEMU Linux中运行的程序，正确加载符号、显示源码

关闭中断 消除外设、OS调度的影响

- local_irq_save：保存状态并关闭中断
  - 其中间接执行了preempt_disable关闭抢占调度
- on_each_cpu：逐个核心执行
- local_irq_restore：恢复状态并打开中断

```
kernel: Hello world 1.
kernel: online_cpus: 4
kernel: disable interrupt on cpu 0.
kernel: disable interrupt on cpu 3.
kernel: disable interrupt on cpu 2.
kernel: disable interrupt on cpu 1.
kernel: disable interrupt on all cpus.
kernel: restore interrupt on cpu 0.
kernel: restore interrupt on cpu 2.
kernel: restore interrupt on cpu 1.
kernel: restore interrupt on cpu 3.
```

单核独占运行 避免多核间资源共享带来的影响

- 在on_each_cpu中使用atomic_t使所有核执行到相同位置

```
atomic_inc(&call_done);
while (atomic_read(&call_done) != cpus)
    cpu_relax();
```

- 接下来主核(cpu==0)独自执行后续任务，其他核cpu_relax忙等

```
if (cpu) {
    while (atomic_read(&call_done) != 0)
        cpu_relax();
} else {
    ret = jpenguin_main(data);
    pr_info("Leaving Steady Mode.\n");
    atomic_set(&call_done, 0);
}
```

加载内核程序

- 向Linux注册一个设备/dev/JudgePenguin
- 将内核程序binary作为设备的固件安装到 /lib/firmware 中
- 使用request_firmware加载内核程序

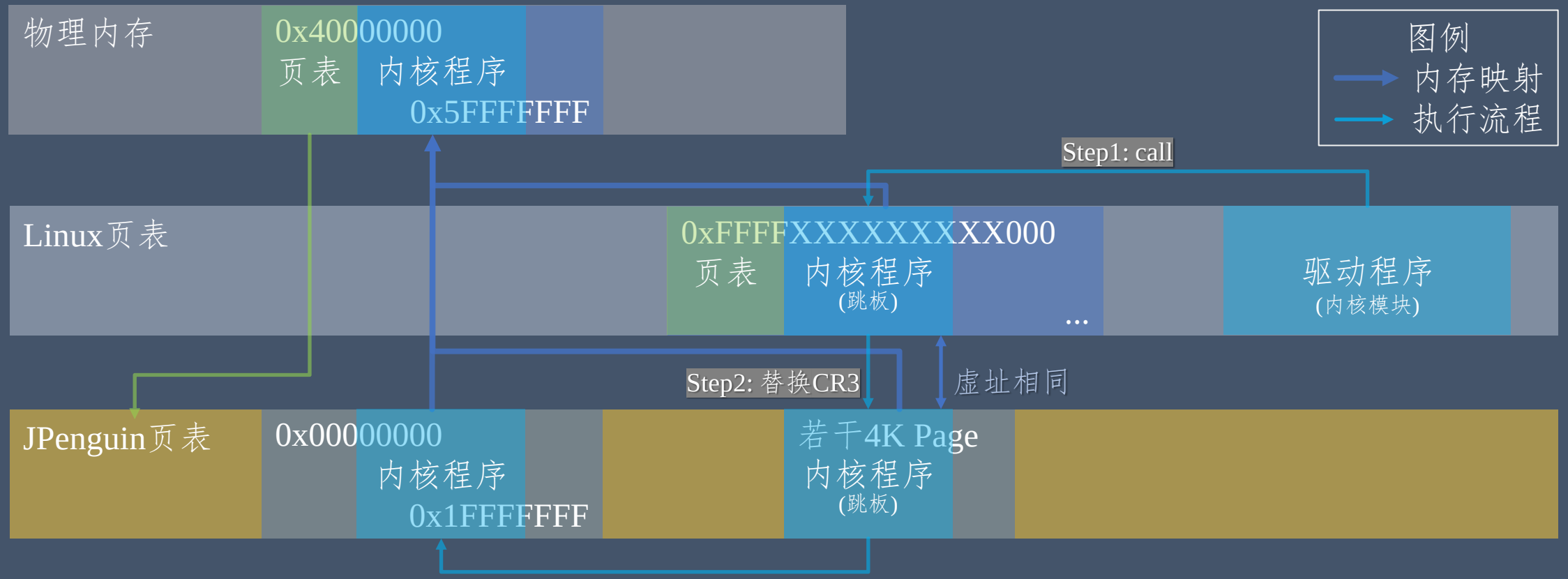
```
MODULE_FIRMWARE(FIRMWARE_NAME);

int load_firmware(void) {
    pr_info("load firmware begin.\n");
    int err;

    const struct firmware *kernel;
    err = request_firmware(&kernel, FIRMWARE_NAME, jpenguin_dev);
```

注册设备之后，还可以使用ioctl与内核模块进行通信(暂未完全实现)

页表切换 内核程序在确定的低地址上运行



替换内核栈 内核程序的所有访存都与Linux无关

- 从Linux内核栈分两步切换到JudgePenguin低地址内核栈
  - 驱动程序将JP高地址栈顶指针作为参数传递给内核程序，进入内核程序后立即从Linux内核栈切换到JP高地址内核栈
  - 切换页表后JP低地址生效，将rsp减去地址偏移，切换到低地址内核栈(实际上对应相同物理内存)，随后jmp到低地址中的duck::kern_loader

```
// switch kernel stack pointer to JPenguin low virtual address space
addq %rdi, %rsp
// calculate the address of the duck kernel loader in JPenguin low virtual address space
leaq kern_loader(%rip), %r15
addq %rdi, %r15
jmp *%r15
```

“动态链接” 链接musl-libc，运行时修复基址偏移

- 直接链接内核程序时生成了一些位置相关内容
- 人工检查汇编源码、链接时添加`-pie`选项后解决了大部分问题
- JudgeDuck链接了musl-libc，其中一些指针无法被`-pie`正确处理
- 驱动程序加载的是binary而非ELF，似乎难以实现真正的动态链接
- 于是驱动程序传入基址偏移，内核程序在进入duck前手动修复

```
extern _IO_FILE __stdout_FILE;
extern _IO_FILE __stderr_FILE;
extern _IO_FILE *stdout;
extern _IO_FILE *stderr;
extern _IO_FILE *__stdout_used;
extern _IO_FILE *__stderr_used;
extern "C" void (*const __init_array_start)(void);
extern "C" void (*const __init_array_end)(void);
extern "C" void (*const __fini_array_start)(void);
extern "C" void (*const __fini_array_end)(void);
extern void *__traps[256];
```

gdb调试QEMU

- 内核程序独立于Linux运行，在调好串口之前无法打印日志
- ASLR 对 Linux 虚存地址进行随机偏移，在 grub cmdline 中添加 `noaslr nokaslr` 关闭 ASLR
- 使用gdb连接到QEMU进行调试
  - 使用`symbol-file`指令加载内核程序符号表，需要使用`-o`设置偏移
  - 使用`-d`选项提供源码目录
  - 这样就可以`b entry`打断点，进入内核程序触发断点，开始调试

gdb调试QEMU

```
Source
 73      // extern const char _binary_hello32_elf_start[];
 74      // extern const char _binary_hello32_elf_end[];
 75      // run_test(_binary_hello32_elf_start, _binary_hello32_elf_end);
 76  }
 77
!78  int main() {
 79      print_hello();
 80
 81      Logger::init();
 82      // TODO: disable debug logging for production
Stack
[0] from 0x0000000000139b70 in main()+0 at duck/kern/kern_main.cpp:78
```

启动JudgeDuck

- ✓ printf, <cmath>
- ✓ PIC::init()
- ✓ Time::init()
- ✓ Trap::init()
- ✓ load_elf64()

- × LAPIC::init()
  出现assertion failed
- × Memory::init()
  PTE_U还没设置
- × ……

```
[  959.610586] JudgePenguin: main begin
[  959.610727] header magic: deadbeef
[  959.611022] now call kernel entry with kernel stack top: [vL]0xffffc90040400000
Hello world!
e = 2.718281828459046
pi = 2 * atan2(1, 0) = 3.141592653589793
[tsc 3076895114926][DEBUG] void PIC::init() start
[tsc 3076896743502][INFO] Enabled Interrupts: 2
[tsc 3076897240846][DEBUG] void PIC::init() done
[tsc 3076897617774][DEBUG] void Timer::init() start
[tsc 3076898050894][DEBUG] CPU brand string: [GenuineIntel]
[tsc 3076898686798][WARN] Assume clk_freq Hz = round(tsc_freq, 100M)
[0.000005][DEBUG] tsc_freq = 3600000000, ext_freq = 1000000000
[0.000318][DEBUG] void Timer::init() done
[0.000420][DEBUG] void Memory::init() start
[0.000601][INFO] Kernel memory used: 2.1 MiB
[0.000731][INFO] vaddr_break = 20000000 (512.0 MiB)
[0.000902][DEBUG] void Memory::init() done
[0.000995][DEBUG] void Trap::init() start
[0.001167][DEBUG] void Trap::init() done
[0.001266][INFO] Running tests
[0.001344][DEBUG] start = 0x15f0b8, len = 202472
[0.001475][INFO] Loading ELF64
[0.002545][INFO] Load ELF ok!
[0.002707][INFO] ========================================
[0.002808][INFO] Welcome to JudgeDuck-OS-64 !!!
[0.002893][INFO] ABI Version 0.04
[  959.622025] kernel exit with code 233
[  959.622390] header magic: abcd1234
[  959.625159] JudgePenguin: main end
```

从JudgeDuck退出

- 向syscall_handler添加SYS_exit
- 跳转到duck_exit
- 恢复rsp(高地址)，ret，搞定

```
duck_entry:
    // callee saved registers
    pushq %rbx
    pushq %rbp
    pushq %r12
    pushq %r13
    pushq %r14
    pushq %r15
    movq %rsp, .wrapper_rsp(%rip)

    // switch kernel stack pointer
    addq %rdi, %rsp
    // calculate the address of th
    leaq kern_loader(%rip), %r15
    addq %rdi, %r15
    jmp *%r15

    .globl duck_exit
duck_exit:
    movq .wrapper_rsp(%rip), %rsp
    popq %r15
    popq %r14
    popq %r13
    popq %r12
    popq %rbp
    popq %rbx
    movq %rdi, %rax
    ret
```

已经实现的功能

- 通过Linux内核模块关闭中断，实现单核独占运行
- 在Linux中预留物理内存，在内核模块中获取并使用
- 构造x86-64 4级页表，实现页表与内核栈切换，修复内存基址
  → 在地址确定的稳定环境中运行包含libc的内核态程序
  运行结束后能够顺利返回Linux
- 完成了JudgeDuck-OS (core) 的部分迁移 (暂时还不能正确测试用户态程序)
- 通过gdb远程调试在QEMU Linux中运行的程序，正确加载符号、显示源码

rdtsc测试

- 使用rdtsc指令连续多次读取CPU时间戳，记录最大间隔
- 若没有中断影响，最大值应稳定取得较小的值 (≤100)
- 未关闭中断：$t_{max} > 10^4$ 在10^8次中出现超过10^3次
- 关闭中断：总有$t_{max} \leq 100$

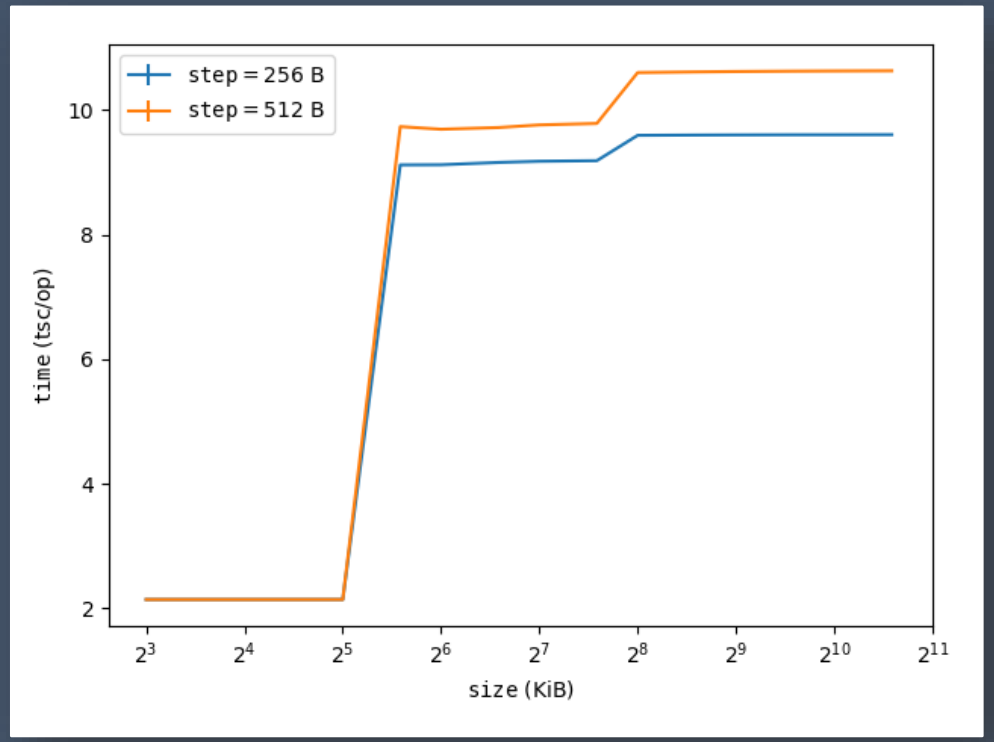
```
kernel: test_rdtsc on cpu 0 for 100000000 rounds.
kernel: avg=21 max=100 bad_count=0.
kernel: test_rdtsc pass.
kernel: test_rdtsc on cpu 0 for 100000000 rounds.
kernel: avg=21 max=98 bad_count=0.
kernel: test_rdtsc pass.
```

cache测试 体系结构课缓存测量作业的正确打开方式

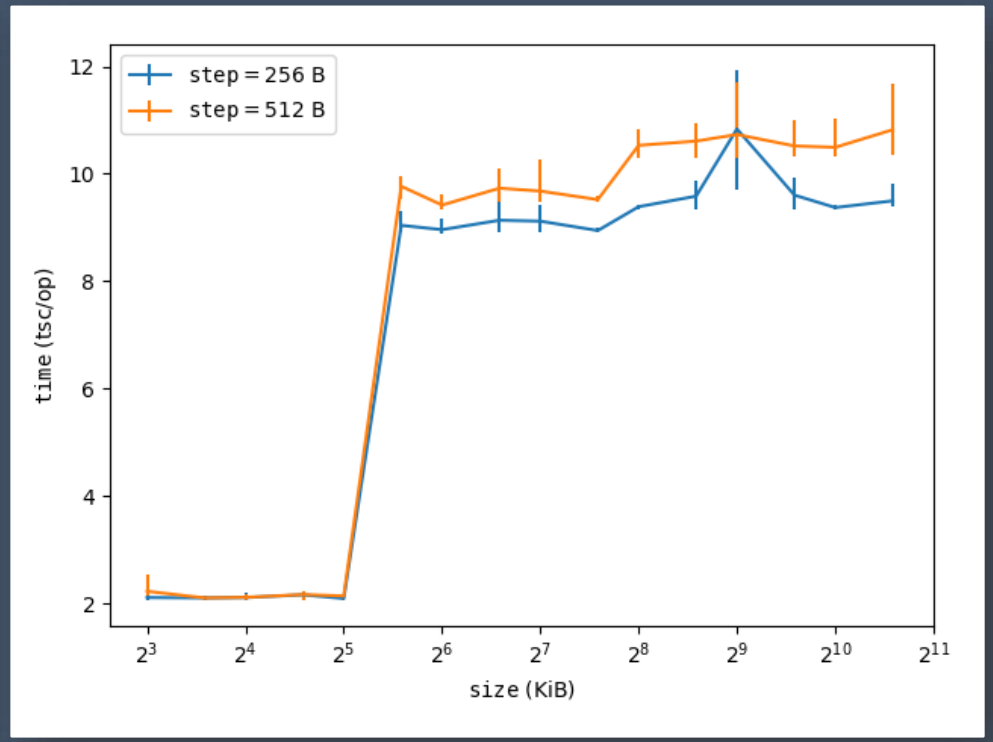
- 以step为步长，循环访问大小为size的内存N次，测量tsc间隔T
- 单次访存耗时为 $T/N-\varepsilon$，在每级cache大小附近将出现跳变

- 在普通环境中：高级别缓存被多任务共享，size较大时测量结果波动很大，甚至掩盖跳变现象
- 在JudgePenguin稳态环境中：所有资源独占使用，结果非常稳定

cache测试



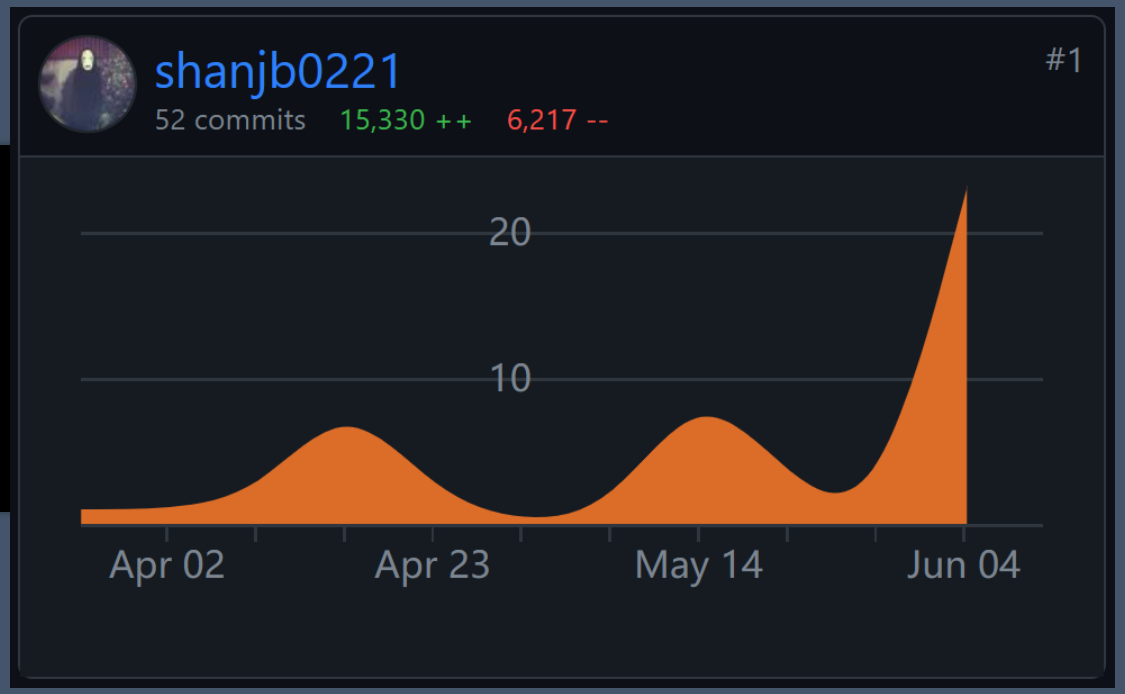
进入稳态后



进入稳态前

代码量

```
wc *.c *.*pp *.S *.h
   .                : 7958
   driver           : 718
   kernel/wrapper:  : 446
```



未完待续

- 完成JudgeDuck-OS的剩余迁移工作
  - 主要困难在于Linux状态保存与PIC改造
- 设计、完善用户接口，使项目更加简单易用 (基于ioctl)
- 提升项目对不同版本Linux内核的兼容性 (目前需要<5.7.0)

> Unfortunately, since Linux v5.7 `kallsyms_lookup_name` is also unexported,

- 在更多真机上部署项目 (更多数量&更多型号)
- --→ 在真实比赛中使用本项目提供评测服务

致谢

- 感谢**陈渝**老师、**向勇**老师、**任炬**老师、**贾跃凯**助教的指导
- 感谢**王逸松**学长对我从选题到开发过程中的指导与帮助
- 感谢**操作系统课程**大实验环节为我提供了这样一份挑战
- 最后，感谢各位的聆听

感谢聆听 & 欢迎提问

致理-信计01 单敬博 2020012711