RustOS

2018操作系统课程设计最终报告

计53 王润基

提纲

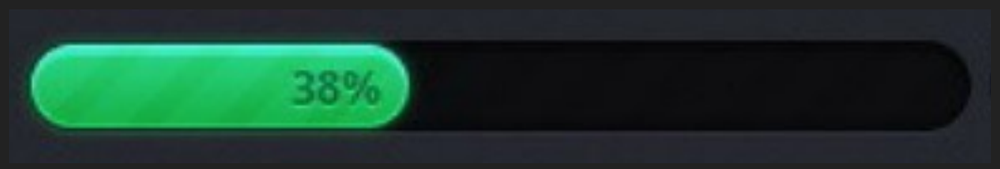
- 整体介绍
  - 目标和完成情况
  - 前期调研情况
  - 分工合作情况
- 实际完成的工作
  - 高地址内核
    - 设备初始化
    - 文件系统
    - 进程管理
    - 支线任务
- Rust语言分析
  - 安全哲学
    - 类型系统
    - 所有权机制
    - 模块化

实验目标和完成情况

使用Rust这一新兴的系统级编程语言来写OS

并利用它主推的内存和线程安全特性

进行SMP多核优化



前半段

移植uCore

后半段

~~SMP优化~~

完成情况

- 基础驱动：全部完成
- 内存管理：框架完成，一些算法没有实现
- 进程管理：近乎完成，可以正常运行大部分用户程序
- 同步互斥：没有完成
- 文件系统：作为单独模块完成，没有接入进程

起点

《Writing an OS in Rust》

- Bare Bones
  - A Minimal x86 Kernel
  - Entering Long Mode
  - Set Up Rust
  - Printing to Screen
- Exceptions
  - Handling Exceptions
  - Double Faults
- Memory Management
  - Allocating Frames
  - Page Tables
  - Remap the Kernel
  - Kernel Heap

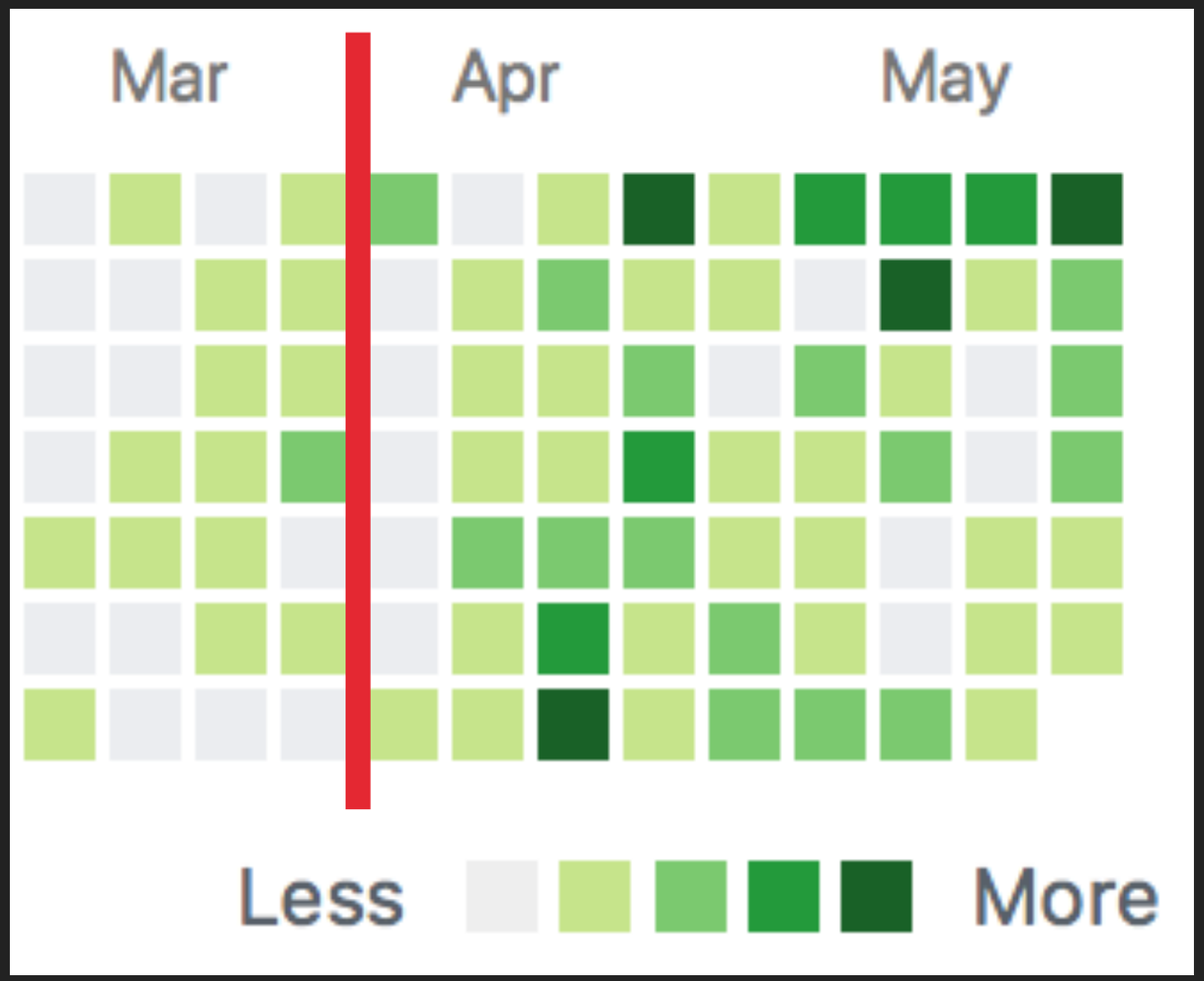
分工合作情况

- 组内“独占”所有工作
- 与其他Rust组各自维护独立仓库
- 互相借鉴，没有合并

实际完成的工作

工作量

uCore RustOS



150+h

180 commits

进度

- Week6：学习《Writing an OS in Rust》，尝试hack
- Week7：C语言互操作，Travis，设备初始化
- Week8：将内核移到高地址区，多核初始化，完成lab1
- Week9：划水（尝试独立的内存管理模块）
- Week10：进程初步，SFS文件系统模块
- Week11：将RustSFS链接到uCore
- Week12：可以运行用户态程序
- Week13：支持运行大部分uCore程序

1. 将内核移到高地址区

Boot32 → Boot64 → Rust64

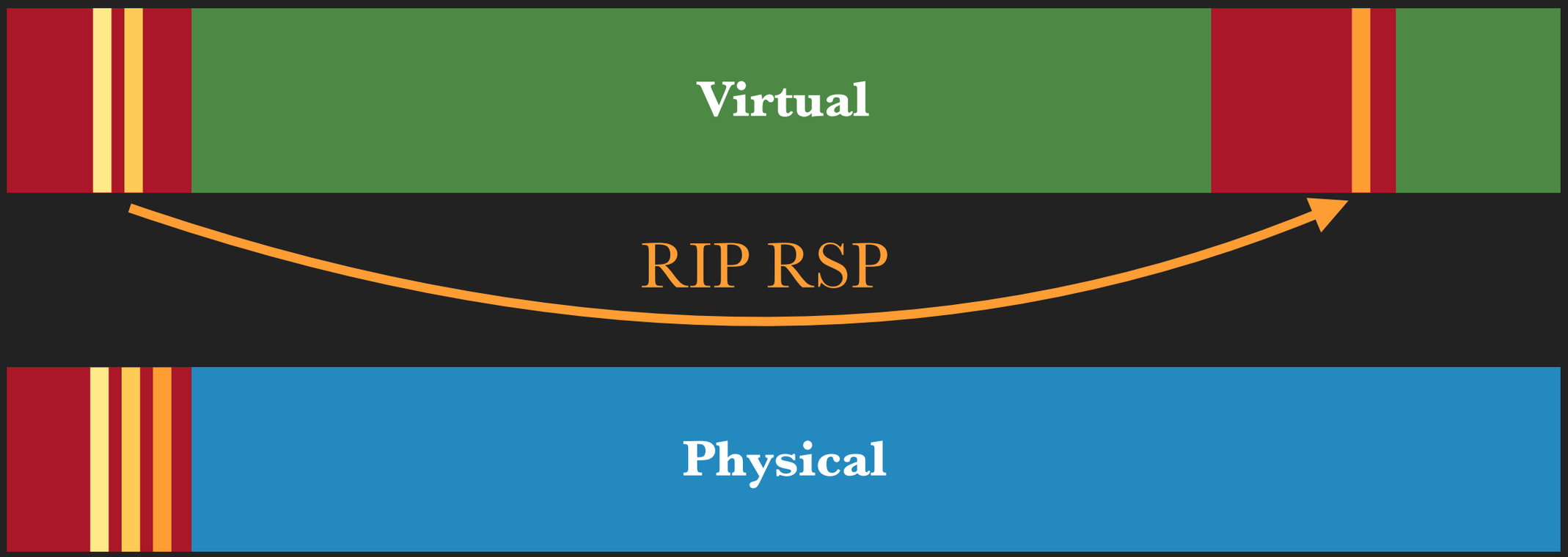
1. 将内核移到高地址区

Boot32 → Boot64 → Rust64

1. 将内核移到高地址区

Boot32 → Boot64 → Rust64

1. 将内核移到高地址区



Boot32 → Boot64 → Rust64

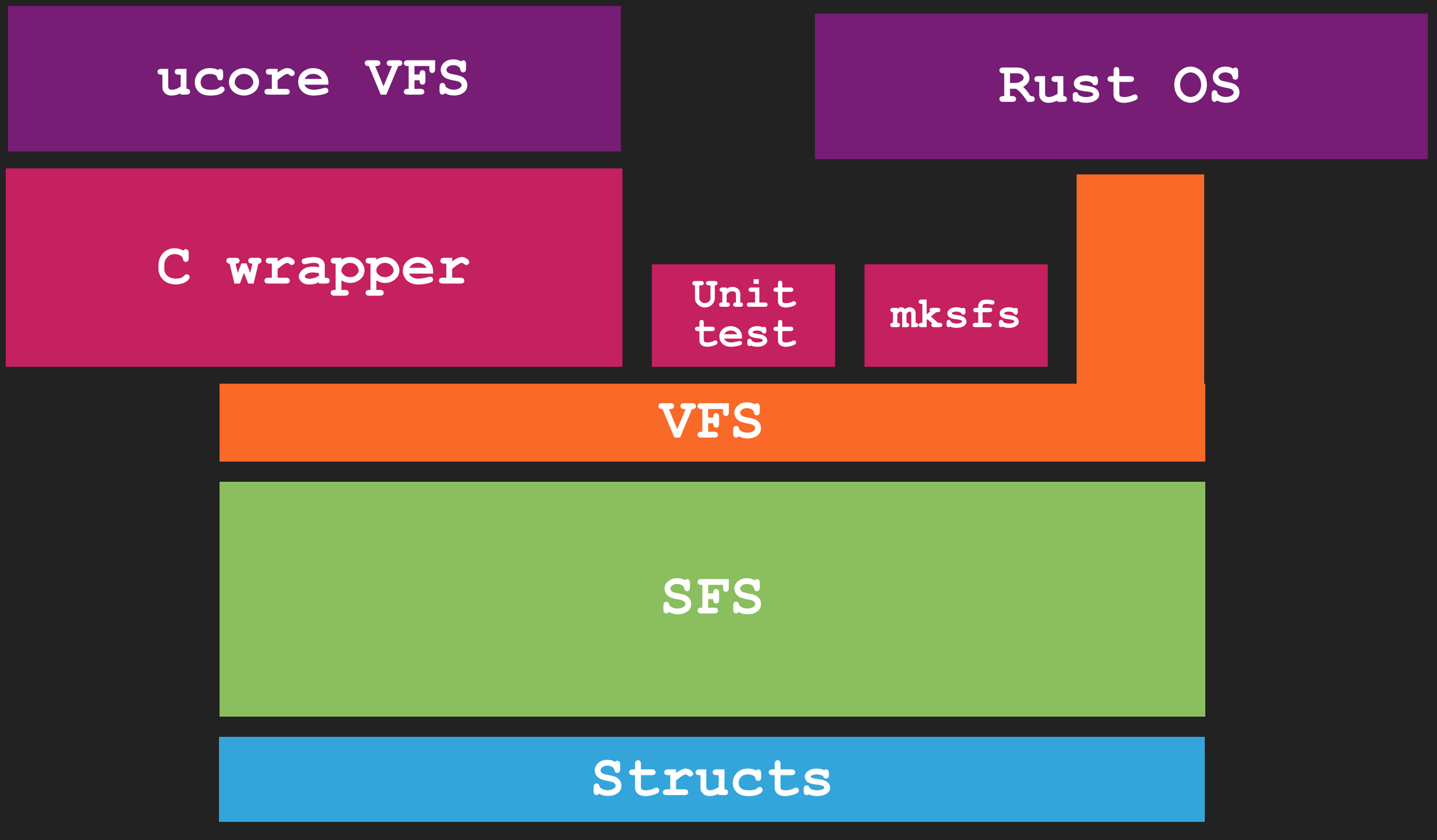
1. 将内核移到高地址区

Boot32 → Boot64 → Rust64

2. 设备和多核的初始化

- ACPI：参考xv6 x86_64，用Rust重写
- LocalAPIC：链接C代码
- IOAPIC：参考xv6 x86_64，用Rust重写
- PIT时钟：复制Redox代码
- IDE：后期借用驱动组的成果
- 键盘：链接C代码
- 串口：复制Redox代码
- VGA：blog_os写好了
- 启动多核：参考xv6 x86_64，用Rust重写

3. SFS文件系统



4. RE: 从零开始的进程管理

- 历时两周，分17个小任务完成
- 没有照搬xv6/uCore，用Rust从头写
- 最初以为平台依赖性很大，但实际发现还好-->模块化?

从零造进程17步

- 借用Redox的中断处理入口函数，保存下TrapFrame。
- 建立最简单的进程控制块Process和调度器Processor，为一个内核函数构造TrapFrame，通过直接改写中断时的tf，使之返回到新的内核线程。
- 为每个线程分配内核栈，中断时不再改写tf，而是在中断处理结束时修改rsp，来实现线程切换。
- 仿照Lab1 Challenge，新建两个软中断ToU/ToK，手动切换内核态和用户态。
- 将一个xv6二进制用户程序链接到Kernel，使用一个ELF解析库读出其各段信息。
- 在内存管理模块中新增类似mm和vma的内存描述结构MemorySet，并实现从ELF段信息的转换，和它在页表上的映射。
- 为用户程序构造TrapFrame和页表，反复调试直到可以执行用户程序。
- 实现一个最基础的系统调用，使得用户程序可以回到内核态。注意需要在每次进入用户态前在TSS中设置返回内核态时的内核栈rsp。
- 实现Fork系统调用，需要谨慎处理页表切换。
- 学习x86_64下的32位兼容模式，使得可以运行uCore的32位用户程序。

- 为了方便测试各个用户程序，把整个sfs.img链接进来，通过之前写好的SFS模块读取所有用户程序。
- 实现一个简易事件处理器，支持程序的睡眠和唤醒。
- 为了在用户程序发生异常时中止其运行，发现Redox版本的中断处理不统一，遂废弃之改用xv6/uCore的实现，修改后直接修复了一个长期阴魂不散的Bug。
- 将散落在各处的对rsp的修改统一到中断处理的最后。
- 发现sys_wait是一个异步操作，需要把(int*)store保存下来，等到某个程序exit后再赋值。修改后通过了waitpid函数的测试。
- 阅读xv6文档后惊讶地发现，它是通过switch函数直接在内核态切换线程来实现调度的（这意味着我前四周ucore也没学明白），在这种机制下上面的问题就不再是问题了。于是我又引入了switch机制，修改了新进程的初始内核栈内容，中断处理时就不再修改rsp了。
- 参考uCore的调度模块，实现了RRScheduler和StrideScheduler。

文本

Terminal

```
    |
    = note: #[warn(unconditi
    = help: a `loop` may exp

    Finished dev [unoptimize
WARNING: Image format was no
         Automatically detec
         Specify the 'raw' f
Hello World!
Hello world! from CPU 1!
Hello world! from CPU 2!
Hello world! from CPU 3!
sleep 1 x 100 slices.
sleep 2 x 100 slices.
sleep 3 x 100 slices.
sleep 4 x 100 slices.
sleep 5 x 100 slices.
sleep 6 x 100 slices.
sleep 7 x 100 slices.
sleep 8 x 100 slices.
sleep 9 x 100 slices.
sleep 10 x 100 slices.
use 1018 msecs.
sleep pass.
QEMU: Terminated
```

sleep

Terminal

```
   --> src/lib.rs:158:9
    |
158 |         fn stack_overflow() {
    |         ^^^^^^^^^^^^^^^^^^^ cannc
159 |             stack_overflow(); //
    |             ---------------- recu
    |
    = note: #[warn(unconditional_recurs
    = help: a `loop` may express intent

    Finished dev [unoptimized + debugir
WARNING: Image format was not specifiec
         Automatically detecting the fc
         Specify the 'raw' format expli
Hello World!
Hello world! from CPU 1!
Hello world! from CPU 2!
Hello world! from CPU 3!
I am the parent. Forking the child...
I am parent, fork a child pid 3
I am the parent, waiting now..
I am the child.
waitpid 3 ok.
exit pass.
QEMU: Terminated
```

exit

Terminal

```
    = help: a `loop` may express intentic

    Finished dev [unoptimized + debuginfc
WARNING: Image format was not specified f
         Automatically detecting the form
         Specify the 'raw' format explici
Hello World!
Hello world! from CPU 1!
Hello world! from CPU 2!
Hello world! from CPU 3!
priority process will sleep 400 ticks
main: fork ok,now need to wait pids.
child pid 6, acc 492000, time 1401
child pid 7, acc 616000, time 1403
child pid 5, acc 332000, time 1407
child pid 4, acc 256000, time 1414
child pid 3, acc 128000, time 1415
main: pid 3, acc 128000, time 1416
main: pid 4, acc 256000, time 1417
main: pid 5, acc 332000, time 1418
main: pid 6, acc 492000, time 1419
main: pid 7, acc 616000, time 1421
main: wait pids over
stride sched correct result: 1 2 3 4 5
QEMU: Terminated
```

priority

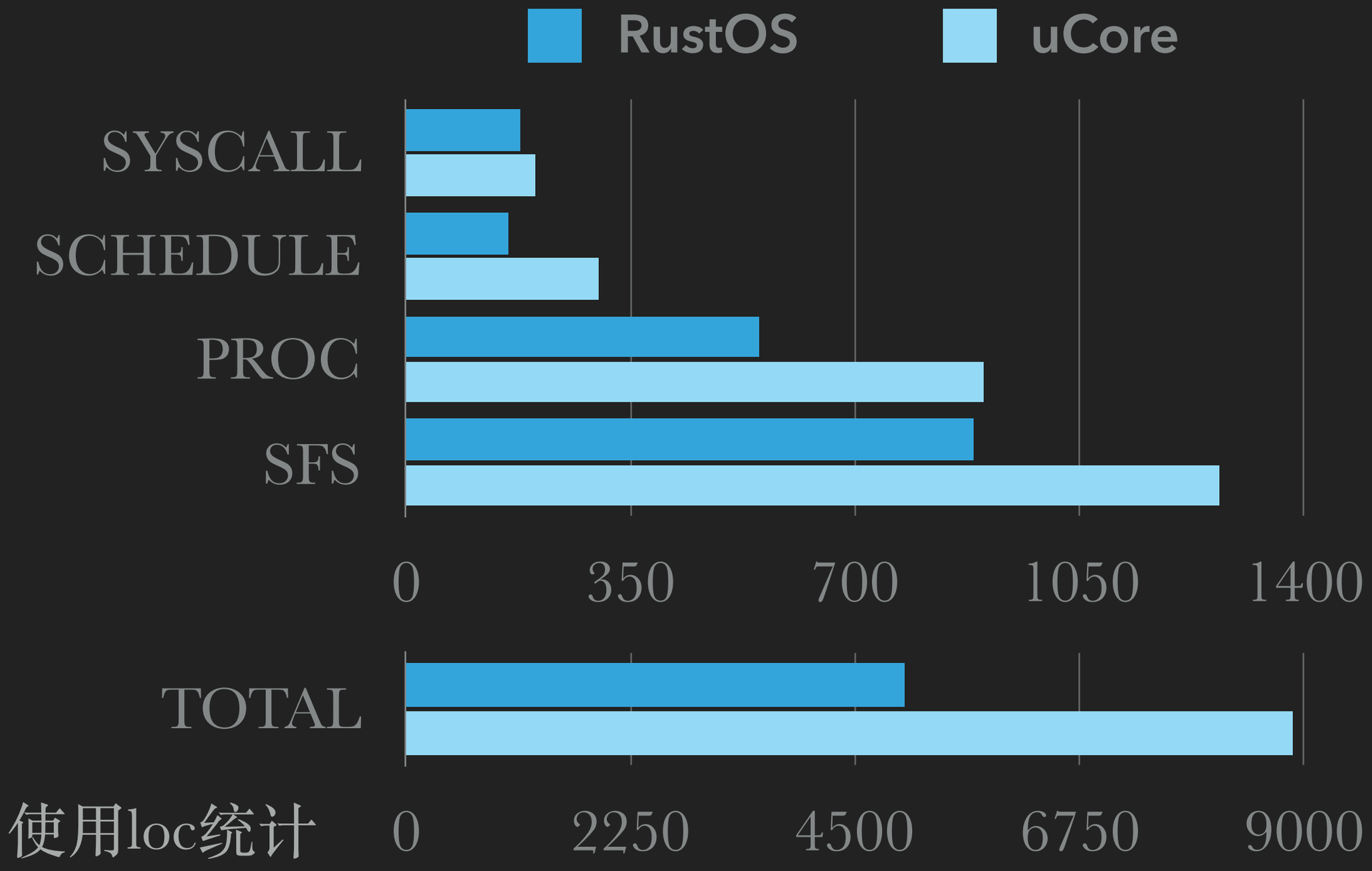
uCore32位用户程序支持清单

- ○ badarg
- ○ badsegment
- ✓ divzero
- ✓ exit
- ✓ faultread
- ✓ faultreadkernel
- ✓ forktest
- ✓ forktree
- ✓ hello
- ○ ls
- ✓ matrix
- ○ pgdir
- ✓ priority
- ○ sh
- ✓ sleep
- ✓ sleepkill
- ✓ softint
- ✓ spin
- ✓ testbss
- ✓ waitkill
- ✓ yield

xv6 64位用户程序支持清单

- ○ cat
- ○ chmod
- ○ echo
- ○ forktest
- ○ grep
- ○ init
- ○ kill
- ○ ln
- ○ ls
- ○ mkdir
- ○ rm
- ○ sh
- ○ stressfs
- ○ usertests
- ○ wc
- ○ zombie

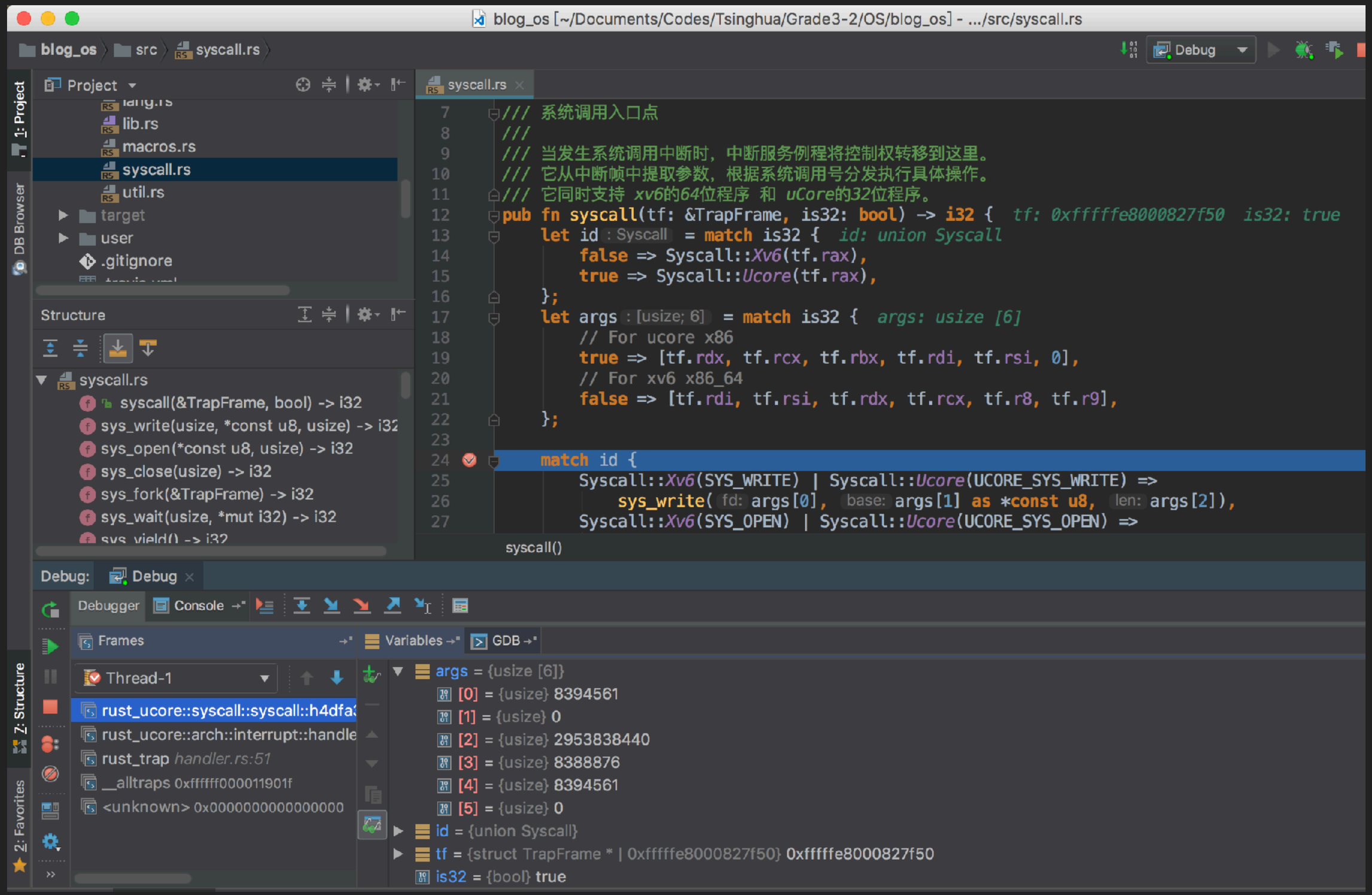
代码量统计


RustOS
uCore
SYSCALL
SCHEDULE
PROC
SFS
0
350
700
1050
1400
TOTAL
0
2250
4500
6750
9000
使用loc统计

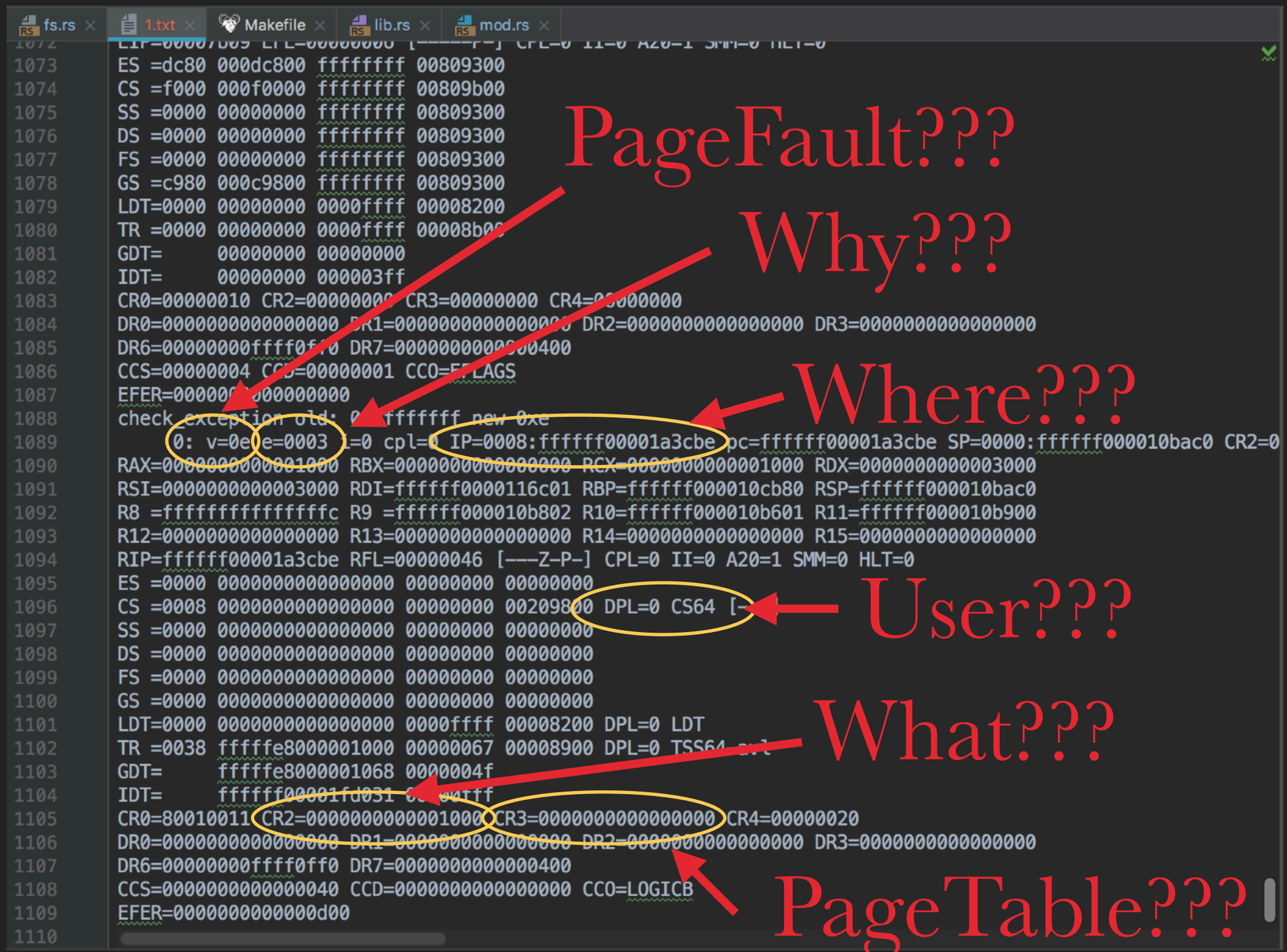
5. 支线任务

- C语言互操作
- TravisCI
- 未被整合的内存管理模块
- Copy-on-write
- CLion配合gdb调试 & 调试经验
- 日志模块和彩色输出

5.5 CLion配合gdb调试



5.5 然而更多的时候……


fs.rs
1.txt
Makefile
lib.rs
mod.rs
ES =dc80 000dc800 ffffffff 00809300
CS =f000 000f0000 ffffffff 00809b00
SS =0000 00000000 ffffffff 00809300
DS =0000 00000000 ffffffff 00809300
FS =0000 00000000 ffffffff 00809300
GS =c980 000c9800 ffffffff 00809300
LDT=0000 00000000 0000ffff 00008200
TR =0000 00000000 0000ffff 00008b00
GDT= 00000000 00000000
IDT= 00000000 000003ff
CR0=00000010 CR2=00000000 CR3=00000000 CR4=00000000
DR0=0000000000000000 DR1=0000000000000000 DR2=0000000000000000 DR3=0000000000000000
DR6=00000000ffff0ff0 DR7=0000000000000400
CCS=00000004 CCD=00000001 CCO=EFLAGS
EFER=0000000000000000
check_exception old: 0xffffffff new 0xe
0: v=0e e=0003 i=0 cpl=0 IP=0008:ffffff00001a3cbe pc=ffffff00001a3cbe SP=0000:ffffff000010bac0 CR2=0
RAX=0000000000001000 RBX=0000000000000000 RCX=0000000000001000 RDX=0000000000003000
RSI=0000000000003000 RDI=ffffff0000116c01 RBP=ffffff000010cb80 RSP=ffffff000010bac0
R8 =fffffffffffffffc R9 =ffffff000010b802 R10=ffffff000010b601 R11=ffffff000010b900
R12=0000000000000000 R13=0000000000000000 R14=0000000000000000 R15=0000000000000000
RIP=ffffff00001a3cbe RFL=00000046 [---Z-P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0000 0000000000000000 00000000 00000000
CS =0008 0000000000000000 00000000 00209800 DPL=0 CS64 [-
SS =0000 0000000000000000 00000000 00000000
DS =0000 0000000000000000 00000000 00000000
FS =0000 0000000000000000 00000000 00000000
GS =0000 0000000000000000 00000000 00000000
LDT=0000 0000000000000000 0000ffff 00008200 DPL=0 LDT
TR =0038 ffffffe8000001000 00000067 00008900 DPL=0 TSS64-avl
GDT= ffffffe8000001068 0000004f
IDT= ffffff00001fd031 0000fff
CR0=80010011 CR2=0000000000001000 CR3=0000000000000000 CR4=00000020
DR0=0000000000000000 DR1=0000000000000000 DR2=0000000000000000 DR3=0000000000000000
DR6=00000000ffff0ff0 DR7=0000000000000400
CCS=0000000000000040 CCD=0000000000000000 CCO=LOGICB
EFER=0000000000000d00
PageFault???
Why???
Where???
User???
What???
PageTable???

5.5 这才是OS的日常

Terminal

```
ffffff00001a3cae:       eb 22                   jmp    ffffff00001a3cd2 <_ZN1
ffffff00001a3cb0:       48 8d 3d 4a 2f f7 ff    lea    -0x8d0b6(%rip),%rdi
ffffff00001a3cb7:       b8 00 10 00 00          mov    $0x1000,%eax
ffffff00001a3cbc:       89 c1                   mov    %eax,%ecx

        unsafe { *(0x1000 as *mut u8) = 2; }
ffffff00001a3cbe:       c6 01 02                movb   $0x2,(%rcx)
        assert_eq!(RC_MAP.read_count(&frame), 1);
ffffff00001a3cc1:       e8 6a 77 ff ff          callq  ffffff000019b430 <_ZN8
7d80e729d24cc6E>
ffffff00001a3cc6:       48 89 85 30 f2 ff ff    mov    %rax,-0xdd0(%rbp)
ffffff00001a3ccd:       e9 ba 00 00 00          jmpq   ffffff00001a3d8c <_ZN1
ffffff00001a3cd2:       48 8d 35 07 bf f9 ff    lea    -0x640f9(%rip),%rsi
daf2fbb859d7bE>
        page_table.map_to_shared(Page::of_addr(0x3000), frame.clone(), EntryF
        assert_eq!(RC_MAP.read_count(&frame), 1);
        assert_eq!(RC_MAP.write_count(&frame), 2);
        assert_eq!(unsafe { *(0x1000 as *const u8) }, 1);
        assert_eq!(unsafe { *(0x2000 as *const u8) }, 1);
        assert_eq!(unsafe { *(0x3000 as *const u8) }, 1);
ffffff00001a3cd9:       48 8b bd 10 f7 ff ff    mov    -0x8f0(%rbp),%rdi
ffffff00001a3ce0:       e8 ab b7 f9 ff          callq  ffffff000013f490 <_ZN4
ffffff00001a3ce5:       48 89 85 28 f2 ff ff    mov    %rax,-0xdd8(%rbp)
ffffff00001a3cec:       48 89 95 20 f2 ff ff    mov    %rdx,-0xde0(%rbp)
ffffff00001a3cf3:       48 8d bd a8 f6 ff ff    lea    -0x958(%rbp),%rdi
:
```

文本

Exceptions

Exceptions as described in this article are generated by the CPU when an 'error' occurs. Some exceptions are not really errors in most cases, such as page faults. Exceptions are a type of interrupt.

Exceptions are classified as:

- **Faults**: These can be corrected and the program may continue as if nothing happened.
- **Traps**: Traps are reported immediately after the execution of the trapping instruction.
- **Aborts**: Some severe unrecoverable error.

Some exceptions will push a 32-bit "error code" on to the top of the stack, which provides additional information about the error. This value must be pulled from the stack before returning control back to the currently running program. (i.e. before calling IRET)

Name	Vector nr.	Type	Mnemonic	Error code?
Divide-by-zero Error	0 (0x0)	Fault	#DE	No
Debug	1 (0x1)	Fault/Trap	#DB	No
Non-maskable Interrupt	2 (0x2)	Interrupt	-	No
Breakpoint	3 (0x3)	Trap	#BP	No
Overflow	4 (0x4)	Trap	#OF	No
Bound Range Exceeded	5 (0x5)	Fault	#BR	No
Invalid Opcode	6 (0x6)	Fault	#UD	No
Device Not Available	7 (0x7)	Fault	#NM	No
Double Fault	8 (0x8)	Abort	#DF	Yes (Zero)
~~Coprocessor Segment Overrun~~	9 (0x9)	Fault	-	No
Invalid TSS	10 (0xA)	Fault	#TS	Yes

Navigation

Main Page
Forums
FAQ
OS Projects
Random page

About

This site
Joining
Editing help
Recent changes

Toolbox

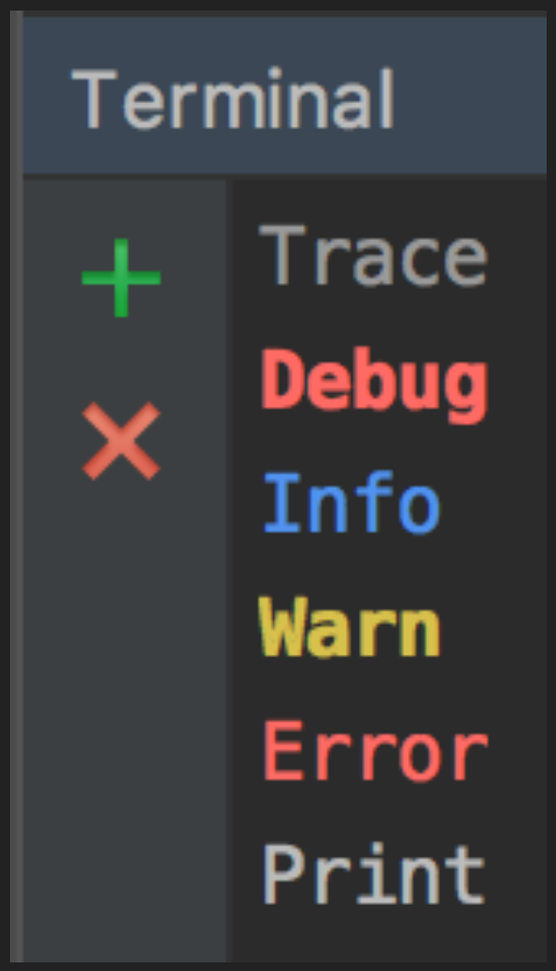
What links here
Related changes
Special pages
Printable version
Permanent link

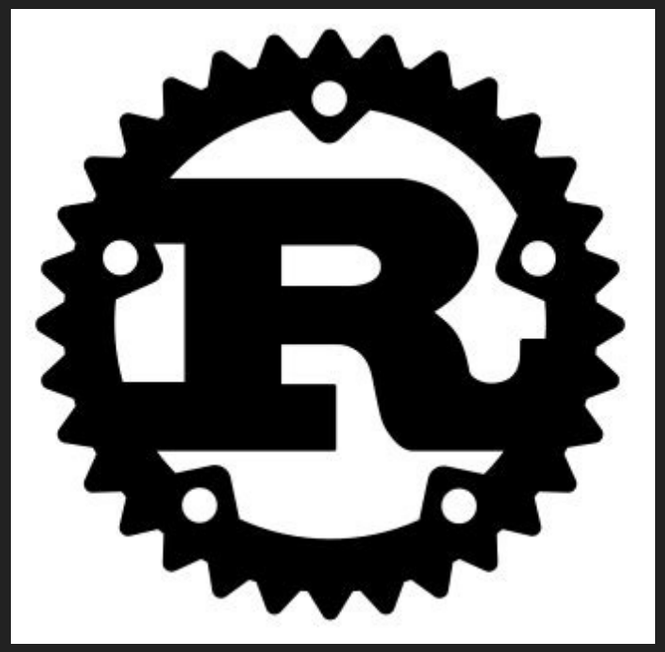
In other languages

Deutsch

5.6 LOG 模块 & 彩色输出

```rust
trace!("Trace");
debug!("Debug");
info!("Info");
warn!("Warn");
error!("Error");
println!("Print");
```





Rust语言分析

RUST的安全哲学


Rust 和 C++ 有哪些优劣?
匿名用户
rust：编译时想撞墙。
c++：调试时想跳楼。
发布于 2016-01-20
102
6 条评论

RUST的安全哲学

- 尝试把系统正确性证明整合到语言本身当中
- Rust从不会放松对安全的要求，但出于实际考虑，它可以允许把编译时约束转移到运行时（例如Mutex，RefCell），也允许把安全保证甩锅给程序员（unsafe块）。
- 显式地指出不安全，并使用安全封装和管理不安全
- unsafe块是一个精妙的设计，它总是在你想偷懒破坏安全性的时候给你带来小小的骚扰

类型系统

- 类似Haskell的强大类型系统
- 鼓励开发者使用自定义类型封装各种概念
- 以页表为例，详见报告

所有权机制和资源管理

RefCell<T>

.borrow()

.borrow_mut()

Mutex<T>

.lock()

Dirty<T>

.borrow()

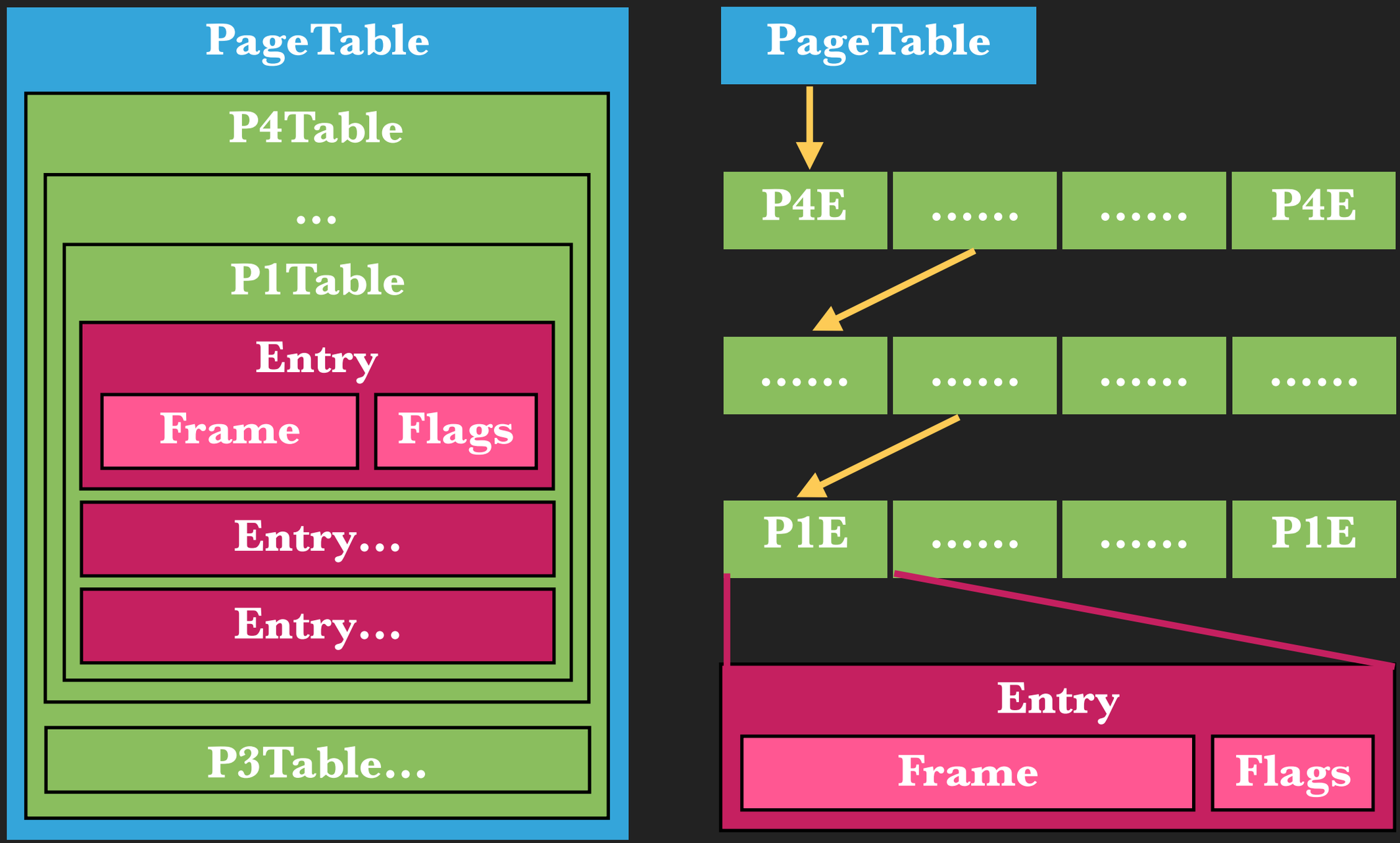
.borrow_mut()

Rc<T>

.clone()

.drop()

Rust vs C


PageTable
P4Table
...
P1Table
Entry
Frame
Flags
Entry...
Entry...
P3Table...
PageTable
P4E
.....
.....
P4E
.....
.....
.....
.....
P1E
.....
.....
P1E
Entry
Frame
Flags

模块化

- 本次实验中进行的模块化尝试
  - 文件系统：😀
  - 进程管理：😏
  - 内存管理：🤪
- 《Writing an OS in Rust》作者起了好头
  - Rust OSDev

尝试和ARM组合并

把进程管理和内存管理模块化

学习借鉴sv6搞SMP优化

完成xv6所有功能的移植，主要是多核运行程序

提供完整的文档，达到和uCore同样的可用度和可读性

完成uCore所有8个lab功能的移植

后期计划

成为更好的uCore

感谢聆听

Q&A