

第 2 讲：中断、异常与系统调用

第三节：kernel mode 操作系统

向勇、陈渝、李国良

清华大学计算机系

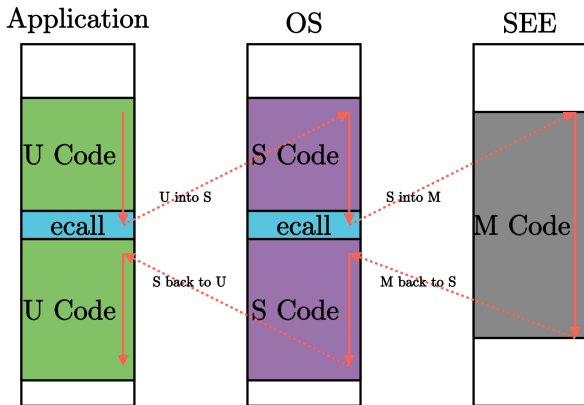
xyong,yuchen,liguoliang@tsinghua.edu.cn

2021 年 9 月 16 日

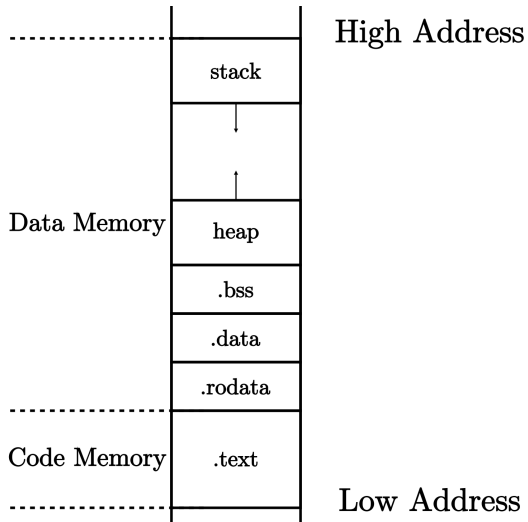
提纲

- 用户态应用程序：控制逻辑，地址空间，系统调用
- kernel-mode OS：启动，初始化，创建执行应用，系统调用服务

kernel mode OS: RISC-V CPU 特权级切换



kernel mode OS: U-Mode 应用程序




kernel mode OS: U-Mode 应用程序的执行环境

应用程序的执行环境的建立是编译器，库和 OS 的共同完成

```
1  #[no_mangle]
2  #[link_section = ".text.entry"]
3  pub extern "C" fn _start() -> ! {
4      clear_bss();
5      exit(main());
6      panic!("unreachable after sys_exit!");
7  }
```

kernel mode OS: U-Mode 应用程序内存布局

- 将程序的起始地址调整为某个地址，程序都会被加载到这个地址上运行
- 将 `_start` 所在的 `.text.entry` 放在整个程序的开头，也就是说只要在加载之后跳转到 `0x80040000` 就已经进入了用户库的入口点，完成初始化
- 提供了执行文件的 `.bss` 段的起始和终止地址，方便 `clear_bss` 函数使用
- 提供栈 (stack) 的地址空间



```
299 //===== userapp main =====
300 #[no_mangle]
301 #[link_section = ".text.entry"]
302 extern "C" fn usr_app_main() {
303     uprintln!("Uusrapp: Hello, world!");
304     sys_exit( xstate: 9);
305 }
```

kernel mode OS: U-Mode 应用程序系统调用

```
/// 功能：将内存中缓冲区中的数据写入文件。  
/// 参数：`fd` 表示待写入文件的文件描述符；  
///       `buf` 表示内存中缓冲区的起始地址；  
///       `len` 表示内存中缓冲区的长度。  
/// 返回值：返回成功写入的长度。  
/// syscall ID: 64  
fn sys_write(fd: usize, buf: *const u8, len: usize) -> isize;  
  
/// 功能：退出应用程序并将返回值告知批处理系统。  
/// 参数：`xstate` 表示应用程序的返回值。  
/// 返回值：该系统调用不应该返回。  
/// syscall ID: 93  
fn sys_exit(xstate: usize) -> !;
```

kernel mode OS: U-Mode 应用程序系统调用实现

- 约定寄存器 a0-a6 保存系统调用的参数，a0-a1 保存系统调用的返回值。寄存器 a7 用来传递 syscall ID

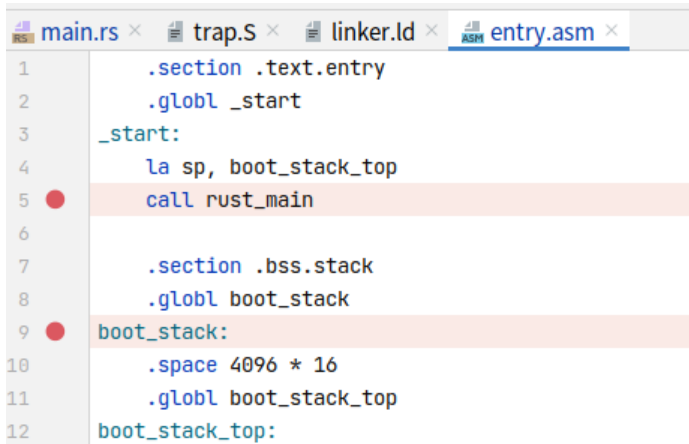
```
1 // user/src/syscall.rs
2
3 fn syscall(id: usize, args: [usize; 3]) -> isize {
4     let mut ret: isize;
5     unsafe {
6         llvm_asm!("ecall"
7             : "={x10}" (ret)
8             : "{x10}" (args[0]), "{x11}" (args[1]), "{x12}" (args[2]), "{x17}" (id)
9             : "memory"
10            : "volatile"
11        );
12    }
13    ret
14 }
```


提纲

- 用户态应用程序：控制逻辑，地址空间，系统调用
- kernel-mode OS：启动，初始化，创建执行应用，系统调用服务

kernel mode OS: 内核启动

- 启动：设置栈，跳转到 rust_main 函数



```
1      .section .text.entry
2      .globl _start
3      _start:
4      la sp, boot_stack_top
5      call rust_main
6
7      .section .bss.stack
8      .globl boot_stack
9      boot_stack:
10     .space 4096 * 16
11     .globl boot_stack_top
12     boot_stack_top:
```

图：内核启动

kernel mode OS: 内核初始化

初始化:

- 初始化系统调用处理 (也可被中断和异常处理共享)
- 初始化和创建应用程序执行环境
- 特权级切换并返回到应用程序的环境中去执行



The screenshot shows a code editor with four tabs: `main.rs` (active), `trap.S`, `linker.ld`, and `entry.asm`. The `main.rs` tab displays the following code:

```
239 //===== kernel main =====
240 #[no_mangle]
241 #[link_section = ".text.entry"]
242 extern "C" fn rust_main() {
243     kprintln!("Kernel: Hello, world!");
244     trap_init();
245     run_usrapp();
246 }
```

kernel mode OS: 内核中的系统调用入口

- 设置系统调用处理的入口位置



The screenshot shows a code editor with four tabs: `main.rs`, `trap.S`, `linker.ld`, and `entry.asm`. The `main.rs` tab is active, displaying the following Rust code:

```
155 global_asm!(include_str!("trap.S"));
156
157 pub fn trap_init() {
158     extern "C" {
159         fn __alltraps(); //in trap.S
160     }
161     unsafe {
162         stvec::write( addr: __alltraps as usize, mode: TrapMode::Direct);
163     }
164 }
165
```

进入 traphandler 前的硬件状态

- 发生异常的指令 PC 被存入 sepc, 且 PC 被设置为 stvec
- scause 根据异常设置类型, stval 被设置为出错的地址或者异常相关信息字
- 把 sstatus CSR 中的 SIE 置零, 屏蔽中断, 且 SIE 之前的值被保存在 SPIE 中
- 发生例外前的特权模式被保存在 sstatus 的 SPP 域, 然后设置当前特权模式为 S 模式

kernel mode OS: 内核系统调用处理框架

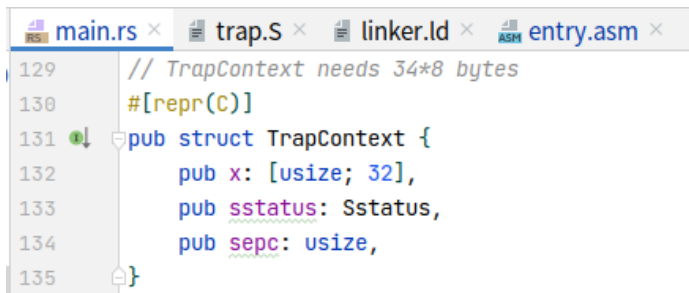
- 进入 traphandler 前的硬件状态

```
# The entry procedure for a interrupt I /exception E into supervisor mode
# (Assuming mideleg[I] /mideleg[E] is set):
# I: scause.interrupt = 1; scause.exception_code = I
# E: scause.interrupt = 0; scause.exception_code = E
# E: stval = faulting virtual address WHEN a hardware breakpoint is triggered,
#           or an instruction, load, or store address-misaligned, access-fault,
#           or page-fault exception occurs; other ZERO
# sstatus.spie = 1, save previous interrupt enable. (See ISR stack.)
# sstatus.sie = 0, interrupts are disabled unless the ISR re-writes this.
# sstatus.spp = Previous privilege mode (S:1 or U:0). set privilege mode to S
# sepc = Interrupted PC, save the return address.
# IF stvec.mode == Direct THEN; PC = (stvec.Base & ~0xF) + (I/E * 4)
```

kernel mode OS: 内核系统调用处理框架

traphandle 处理框架

- 保存应用的中断上下文 (trapcontext): `__alltraps`
- 执行系统调用服务
- 恢复应用的中断上下文 (trapcontext): `__restore`
- `sret` 返回到应用程序继续执行



The screenshot shows a code editor with four tabs: `main.rs`, `trap.S`, `linker.ld`, and `entry.asm`. The `trap.S` tab is active, displaying Rust code for a `TrapContext` struct. The code is as follows:

```
129 // TrapContext needs 34*8 bytes
130 #[repr(C)]
131 pub struct TrapContext {
132     pub x: [usize; 32],
133     pub sstatus: Sstatus,
134     pub sepc: usize,
135 }
```

kernel mode OS: 内核系统调用处理框架

traphandle 处理框架

- sret 返回到应用程序继续执行

```
# The exit procedure for a supervisor interrupt/exception when sret is executed.  
# sstatus.sie = sstatus.spie, restore interrupt enable  
# Privilege mode = sstatus.spp, restore privilege  
# sstatus.spp = sstatus.spie = 0  
# PC = sepc
```


kernel mode OS: 内核创建和初始化应用

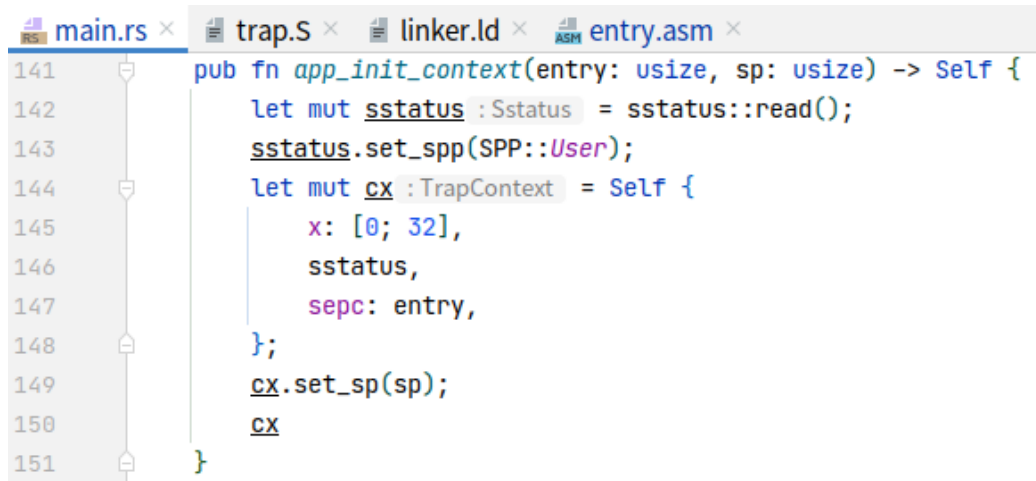
- 模拟应用发出系统调用时的中断上下文



```
227 pub fn run_usrapp() -> ! {
228     extern "C" {
229         fn __restore(cx_addr: usize); //in trap.S
230     }
231     unsafe {
232         __restore( cx_addr: KERNEL_STACK.push_context( cx: TrapContext::app_init_context(
233             entry: usr_app_main as usize,
234             sp: USER_STACK.get_sp(),
235             )) as *const _ as usize);
236     }
```

kernel mode OS: 内核创建和初始化应用

- 初始化应用的中断上下文



The screenshot shows a code editor with four tabs: `main.rs`, `trap.S`, `linker.ld`, and `entry.asm`. The `entry.asm` tab is active, displaying Rust code for the `app_init_context` function. The code is as follows:

```
141 pub fn app_init_context(entry: usize, sp: usize) -> Self {
142     let mut sstatus : Sstatus = sstatus::read();
143     sstatus.set_spp(SPP::User);
144     let mut cx : TrapContext = Self {
145         x: [0; 32],
146         sstatus,
147         sepc: entry,
148     };
149     cx.set_sp(sp);
150     cx
151 }
```

小结

- 了解 kernel-mode OS 如何创建应用程序
- 了解 kernel-mode OS 如何让应用程序执行
- 了解应用程序如何得到 kernel-mode OS 的服务
- 读懂汇编代码的含义和功能