Reedos So Far

Ethan McDonald & Thomas Ulmer

March 19, 2023

Table of Contents

Pre-boot and Linker

Linker script: In kernel.ld

- Organize reedos ELF binary to QEMU expectations.
- Bound symbols to access at runtime.
- Provide alignment where needed.

Linking required to combine Rust code and Assembly.

```
OUTPUT_ARCH( "riscv" )
ENTRY( _entry )

MEMORY
{
  ram  (wxa) : ORIGIN = 0x80000000, LENGTH = 128M
}

SECTIONS
{
  .text : {[[…]]}

  PROVIDE(_global_pointer = .);

  .rodata : {[[…]]}
  .data : {[[…]]}

  /* lower guard page included in above */
  .stacks : {[[…]]}
  .intstacks : {[[…]]}
  . = . + 4096; /* guard page */
  /* stacks should start at stack end and alternate with guard pages going down */

  .bss : {[[…]]}
  PROVIDE(_end = .);
  PROVIDE(_memory_end = ORIGIN(ram) + LENGTH(ram));
}
```

Entry into Assembly

The first assembly to be run on boot. In src/asm/entry.S:16

- All harts jump to the same place.
- Set up primary stack and interrupt stack per hart.
- Prevent harts from colliding in memory.
- Jump to rust.

```
    .option norvc
    .section .text

    .global _entry
    _entry:

    .option push
    .option norelax
        # Linker position data relative to gp
    .extern _global_pointer
        la gp, _global_pointer
    .option pop
        # Set up stack per of hart ids according to linker script

        # Add 4k guard page per hart
        csrr a1, mhartid
        #sll a1, a1, 1 # Multiple hartid by 2 to get alternating pages
        li a0, 0x3000
        mul a1, a1, a0
    .extern _stacks_end # Linker supplied
        la a2, _stacks_end
        sub sp, a2, a1

        .extern _intstacks_end
                csrr a1, mhartid
                li a0, 0x4000
                mul a1, a1, a0
                la a2, _intstacks_end
                sub a2, a2, a1
                csrw mscratch, a2 # Write per hart mscratch pad
                li a0, 0x2000
                sub a2, a2, a0 # Move sp down by scratch pad page + guard page
                csrw sscratch, a2 # Write per hart sscratch pad

        # Jump to _start in src/main.rs
        .extern _start
        call _start
    spin:
        wfi
        j spin
```

Start into Rust

Setup for transition to supervisor mode In src/lib.rs:50

- Disable paging until `vm::init`
- Do setup that requires high privilege.
- ID harts in non-protected register.
- Begin firing M-mode timer interrupts.

```rust
/// This gets called from entry.S and runs on each hart.
/// Run configuration steps that will allow us to run the
/// kernel in supervisor mode.
#[no_mangle]
pub extern "C" fn _start() {
    // xv6-riscv/kernel/start.c
    let fn_main: *const () = main as *const ();

    // Set the *prior* privilege mode to supervisor.
    // Bits 12, 11 are for MPP. They are WPRI.
    // For sstatus we can write SPP reg, bit 8.
    let mut ms: u64 = read_mstatus();
    ms &= !MSTATUS_MPP_MASK;
    ms |= MSTATUS_MPP_S;
    write_mstatus(status: ms);

    // Set machine exception prog counter to
    // our main function for later mret call.
    write_mepc(addr: fn_main);

    // Disable paging while setting up.
    write_satp(pt: 0);

    // Delegate trap handlers to kernel in supervisor mode.
    // Write 1's to all bits of register and read back reg
    // to see which positions hold a 1.
    write_medeleg(med: 0xffff);
    write_mideleg(mid: 0xffff);
    //Supervisor interrupt enable.
    let sie: u64 = read_sie() | SIE_SEIE | SIE_STIE | SIE_SSIE;
    write_sie(ire: sie);

    // Now give sup mode access to phys mem.
    // Check 3.7.1 of riscv priv isa manual.
    write_pmpaddr0(addr: 0x3fffffffffffff_u64); // RTFM
    write_pmpcfg0(addr: 0xf); // 1st 8 bits are pmp0cfg

    // Store each hart's hartid in its tp reg for identification.
    let hartid: u64 = read_mhartid();
    write_tp(id: hartid);

    // Get interrupts from clock and set mtev handler fn.
    hw::timerinit();

    // Now return to sup mode and jump to main().
    call_mret();
} fn _start
```

Main

Initialize kernel subsystems on hart 0. In src/lib.rs:97

- Devices
- Exception and Interrupts traps
- Virtual memory subsystem

```rust
// Primary kernel bootstrap function.
// We ensure that we only initialize kernel subsystems
// one time by only doing so on hart0.
fn main() -> ! {
    // We only bootstrap on hart0.
    let id: u64 = read_tp();
    if id == 0 {
        uart::Uart::init();
        println!("{}", param::BANNER);
        log!(Info, "Bootstrapping on hart0...");
        trap::init();
        log!(Info, "Finished trap init...");
        let _ = vm::init();
        log!(Info, "Initialized the kernel page table...");
        log!(Debug, "Testing page allocation and freeing...");
        unsafe {
            vm::test_palloc();
        }
    } else {
        //Interrupt other harts to init kpgtable.
        trap::init();
    }

    loop {}
}
```

Uart Device

Treat serial port as streaming device at byte granularity. In src/device/uart.rs:31

- Initialize to match QEMU
- Protect with spinlock and hook with `print!`

```rust
impl Uart {
    pub fn init() {
        // https://mth.st/blog/riscv-qemu/AN-491.pdf <-- inclues 16650A ref
        let ptr: *mut u8 = UART_BASE as *mut u8;
        // Basic semantics:...
        unsafe {
            // Disable interrupts first.
            ptr.add(count: IER).write_volatile(val: 0x0);
            // Mode in order to set baud rate.
            ptr.add(count: LCR).write_volatile(val: 1 << 7);
            // baud rate of 38.4k
            ptr.add(count: 0).write_volatile(val: 0x03); // LSB (tx side)
            ptr.add(count: 1).write_volatile(val: 0x00); // MST (rx side)
                                                  // 8 bit words (no parity)
            ptr.add(count: LCR).write_volatile(val: 3);
            // Enabse and clear FIFO
            ptr.add(count: FCR).write_volatile(val: 1 << 0 | 3 << 1);
            // Enable tx and rx interrupts
            ptr.add(count: IER).write_volatile(val: 1 << 1 | 1 << 0);
        }
    } fn init

    pub const fn new() -> Mutex<Self> {
        Mutex::new(Uart {
            base_address: UART_BASE,
        })
    }

    pub fn put(&mut self, c: u8) {
        let ptr: *mut u8 = self.base_address as *mut u8;
        unsafe {
            ptr.add(count: 0).write_volatile(val: c);
        }
    }

    pub fn get(&mut self) -> Option<u8> {
        let ptr: *mut u8 = self.base_address as *mut u8;
        unsafe {
            if ptr.add(count: 5).read_volatile() & 1 == 0 {
                // The DR bit is 0, meaning no data
                None
            } else {
                // The DR bit is 1, meaning data!
                Some(ptr.add(count: 0).read_volatile())
            }
        }
    }
} impl Uart
```

Trap into Assembly

Middleman between rust and interrupting rust. In src/asm/trap.S:72

- Save registers to allow restoration of previous state.
- Make it safe to call rust, even if clobbered registers are in use.

```
    .section .text
# This is the machine mode trap vector(not really). It exists
# to get us into the rust handler
    .option norvc
    .align 4
    .global __mtrapvec
__mtrapvec:
        csrrw sp, mscratch, sp
        save_gp_regs

        .extern m_handler
    call m_handler

        load_gp_regs
        csrrw sp, mscratch, sp
    mret

# This is the supervisor trap vector, it just exists to get
# us into the rust handler elsewhere
    .option norvc
    .align 4
    .globl __strapvec
 __strapvec:
        csrrw sp, sscratch, sp
        save_gp_regs

    .extern s_handler
    call s_handler

        load_gp_regs
        csrrw sp, sscratch, sp
    sret
```

Trap in Rust

Switch based on `mcause` or `scause` In src/trap.rs:32

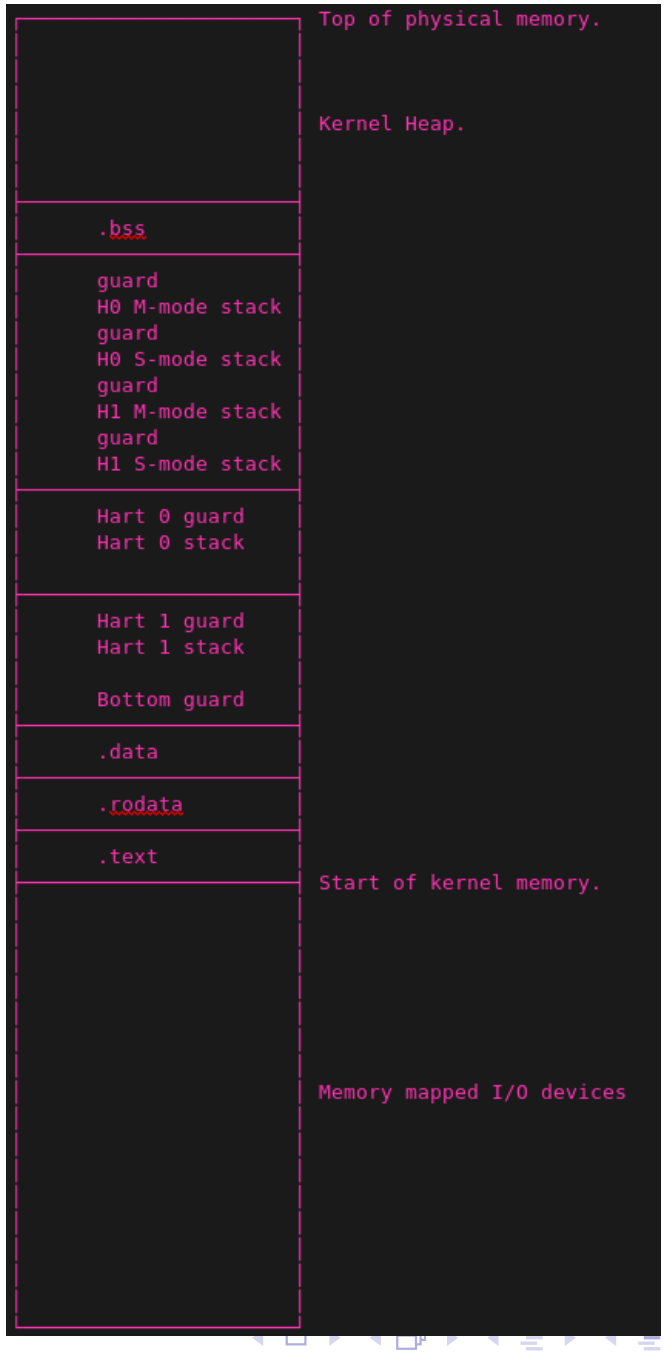
- Reset timer interrupt to make it regularly scheduled.
- Catch exceptions and halt execution.
- TODO: catch page faults.

```rust
/// Machine mode trap handler.
#[no_mangle]
pub extern "C" fn m_handler() {
    let mcause: u64 = riscv::read_mcause();

    match mcause {
        riscv::MSTATUS_TIMER => {
            // log::log!(Debug, "Machine timer interupt, hart: {}", ri
            clint::set_mtimecmp(interval: 10_000_000);
        }
        _ => {
            log::log!(
                Warning,
                "Uncaught machine mode interupt. mcause: 0x{:x}",
                mcause
            );
            panic!();
        }
    }
}

/// Supervisor mode trap handler.
#[no_mangle]
pub extern "C" fn s_handler() {
    let cause: u64 = riscv::read_scause();

    match cause {
        _ => {
            log::log!(
                Warning,
                "Uncaught supervisor mode interupt. scause: 0x{:x}",
                cause
            );
            panic!()
        }
    }
}
```

Virtual Memory Subsystem

Contains most memory abstractions.

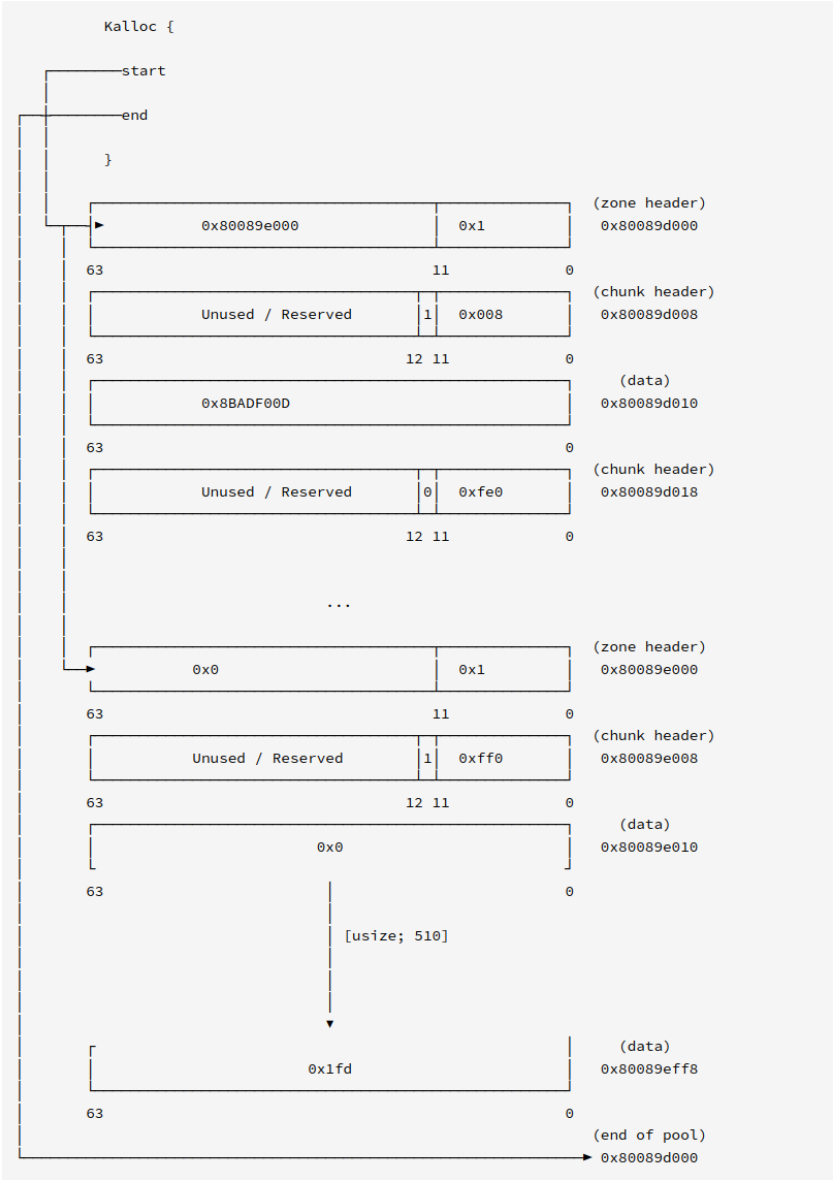
- Page allocation.
- General allocation.
- Virtual memory for kernel and processes.
- Kernel page table maps all of memory with correct permissions.



Allocation

Memory allocation has two forms:

- `palloc` gives physically contiguous pages.
- `vmalloc` gives sub-page chunks like `malloc`.
- Global alloc... Sound familiar?



Allocation Example

Allocation on the kernel heap.

In src/vm.rs:143

```
use alloc::collections;
{
    // Simple test. It works!
    let mut one = Box::new(5);
    let a_one: *mut u32 = one.as_mut();
    assert_eq!(*one, *a_one);

    // Slightly more interesting... it also works! Look at GDB
    // and watch for the zone headers + chunk headers indicating 'in us
    // 'chunk size'. Then watch as these go out of scope.
    let mut one_vec: Box<collections::VecDeque<u32>> = Box::default();
    one_vec.push_back(555);
    one_vec.push_front(111);
    let _a_vec: *mut collections::VecDeque<u32> = one_vec.as_mut();
}
```

Memory state while in use.

```
(gdb) x/16g 0x800ac000
0x800ac000:     0x3     0x1008
0x800ac010:     0x5     0x1020
0x800ac020:     0x4     0x800ac048
0x800ac030:     0x3     0x2
0x800ac040:     0x1010  0x22b
0x800ac050:     0x6f00000000    0xfa0
0x800ac060:     0x0     0x0
0x800ac070:     0x0     0x0
```

Memory state after drop.

```
(gdb) x/16g 0x800ac000
0x800ac000:     0x0     0x8
0x800ac010:     0x5     0xfe0
0x800ac020:     0x4     0x800ac048
0x800ac030:     0x3     0x2
0x800ac040:     0x0     0x22b
0x800ac050:     0x6f00000000    0x0
0x800ac060:     0x0     0x0
0x800ac070:     0x0     0x0
```

GlobalAlloc

Implementing the GlobalAlloc trait with reedos memory allocation tools.

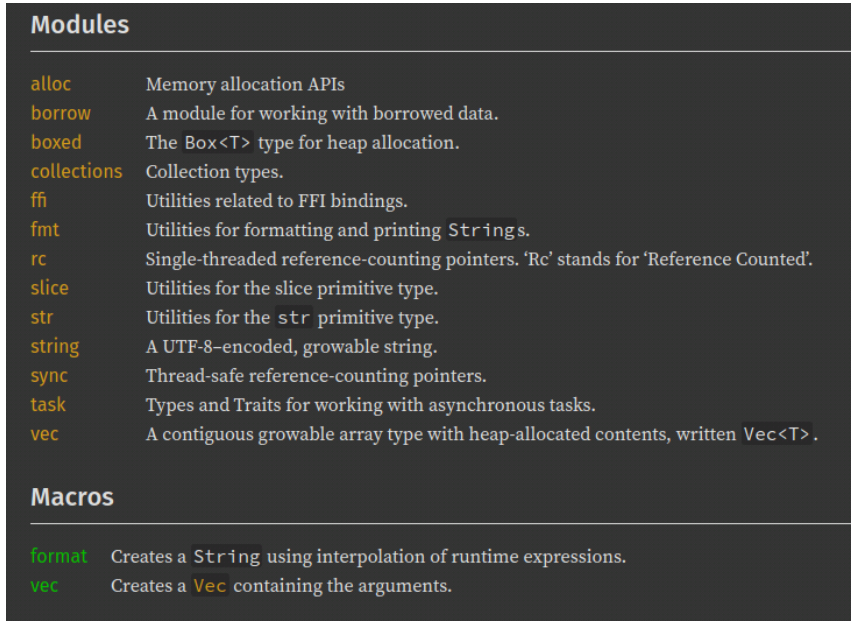
In src/vm.rs:20

```rust
#[global_allocator]
static mut GLOBAL: GlobalWrapper = GlobalWrapper {
    inner: OnceCell::new(),
};

struct GlobalWrapper {
    inner: OnceCell<Galloc>,
}

unsafe impl GlobalAlloc for GlobalWrapper {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        self.inner.get().unwrap().alloc(layout)
    }

    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
        self.inner.get().unwrap().dealloc(ptr, layout)
    }

    unsafe fn alloc_zeroed(&self, layout: Layout) -> *mut u8 {
        self.inner.get().unwrap().alloc_zeroed(layout)
    }

    unsafe fn realloc(&self, ptr: *mut u8, layout: Layout, new_size: usize) -> *mut u8 {
        self.inner.get().unwrap().realloc(ptr, layout, new_size)
    }
```

Useful tools

Within reedos:

- The benefits of General Allocation: $\rightarrow$
- `log` for logging with severity via uart.
- `core::assert` for unsafe/runtime checking.



Modules

alloc	Memory allocation APIs
borrow	A module for working with borrowed data.
boxed	The `Box<T>` type for heap allocation.
collections	Collection types.
ffi	Utilities related to FFI bindings.
fmt	Utilities for formatting and printing `String`s.
rc	Single-threaded reference-counting pointers. 'Rc' stands for 'Reference Counted'.
slice	Utilities for the slice primitive type.
str	Utilities for the `str` primitive type.
string	A UTF-8–encoded, growable string.
sync	Thread-safe reference-counting pointers.
task	Types and Traits for working with asynchronous tasks.
vec	A contiguous growable array type with heap-allocated contents, written `Vec<T>`.

Macros

format	Creates a `String` using interpolation of runtime expressions.
vec	Creates a `Vec` containing the arguments.

Useful tools

Outside of reedos:

- GNU toolchain guides.
- Specifically GDB.
- Trust me use GDB.

```
Thread 1.1 hit Hardware watchpoint 2: *reedos::vm::KPGTABLE

Old value = reedos::vm::ptable::PageTable {
  base: 0x8008c000
}
New value = reedos::vm::ptable::PageTable {
  base: 0x80089ee0
}
reedos::lock::mutex::Mutex<reedos::device::uart::Uart>::lock<reedos::device::uart::Uart> (self=0x80089ee0) at src/lock
/mutex.rs:64
64              while self.lock_state.swap(1, Ordering::Acquire) == 1 {}
(gdb)
```

What you could do / next steps

Project Ideas and Stubs

- UART input and nice wrappers.
- File system (+ shell?).
- Device drivers.
- Page Fault handling (+ swap?).
- I/O device buffers.
- Syscalls.
- Key/Value (`alloc`?) / Page Cache.

Prototyped outside(?)

- `DataSource` for whole disk partition.
- L4-style synchronous IPC.
- Condition variables (Async I/O or `alloc::task`).

Our Short Term

- Hello World from userspace.
- Process loading + scheduling.