启动

概述

目的

1. 了解操作系统开发实验环境
2. 熟悉使用 类 Unix 操作系统
3. 熟悉命令行方式的编译、调试工程
4. 了解操作系统编译，启动流程

简述

操作系统编译和启动是操作系统开发中的两个关键流程，也是最基本的流程，下面简单介绍一下这两个流程。

操作系统一般由汇编语言和高级语言混合编写而成，编译过程会将代码转换为机器码，并生成可执行文件（通常是二进制格式）。具体的编译流程如下：

- 汇编语言编写引导程序
- 编写操作系统内核代码
- 编写链接脚本文件
- 将源代码编译成目标文件
- 链接目标文件，生成可执行文件

操作系统启动是指计算机从关机状态到开始运行用户应用程序的过程。一般来说，操作系统启动可以分成以下几个步骤：

- 加载引导程序：计算机通电后，BIOS 会加载引导设备上的引导程序，通常是位于硬盘、软盘或光盘上的 bootloader 程序。
- 执行引导程序：引导程序被加载到内存中后，CPU会跳转到引导程序的入口点，开始执行引导程序。
- 初始化系统：引导程序会初始化硬件设备，建立内存映射表等，并将控制权交给操作系统内核。
- 加载内核：操作系统内核被加载到内存中，然后进行初始化工作，例如初始化进程管理器、文件系统等。
- 运行用户应用程序：一旦操作系统初始化完成，就可以开始运行用户应用程序了，例如打开浏览器、编辑器等。

这些步骤会因为不同的操作系统而稍有不同，但通常都遵循类似的流程。

在本部分中，我们将会介绍操作系统编译，启动的相关知识。在实验部分，会引导学生配置开发环境，使用 Qemu 模拟器成功运行起 XV6 操作系统。

讲解

编译操作系统

编译操作系统的过程和编译普通软件的过程有很多相似之处，但也有一些不同。我们先专注于相似之处，最后再说不同之处。

操作系统一般由汇编语言和高级语言（C/C++）混合编写而成，编译过程会将代码转换为机器码，并生成二进制可执行文件。

而一个普通的 C 语言程序从源代码变成可执行程序要经过 4 个步骤：

1. 预处理
2. 编译
3. 汇编
4. 链接

操作系统从源代码变成可执行程序，也同样要经过这 4 个步骤，这一点和普通软件没有什么不同。

预处理

在这个阶段，编译器将源代码中的预处理指令（如 #include 、 #define等 ）替换为实际的代码或者内容。

为了编写跨平台的软件，通过预处理对代码进行一些选择，是常见的技巧。其常见的流程是：

1. 在编写代码时，可以根据不同的目标平台/不同的编译器，编写不同的代码。
2. 在预处理时，预处理器（编译器）会带上特定的宏（macro），去“筛选”代码。

以下是一个例子

```
// 根据硬件平台，选择不同的源代码，做不同的处理
void foo(){
#if defined(__x86_64__)
    // 在 64位x86 平台，我要这么做...
    A();
#elif defined(__riscv)
    // 在 RISC-V 平台，我要这么做...
    B();
#elif defined(__loongarch__ )
    // 在 龙芯 平台，我要这么做...
    C();
#endif
}

// 根据编译器种类 做选择
void bar(){
#if defined(_MSC_VER)
    // 使用 微软MSVC 编译器，我要这么做...
    A();
#elif defined(__GNUC__)
    // 使用 GCC 编译器，我要这么做...
    B();
#elif defined(__ICC )
    // 使用 英特尔ICC 编译器，我要这么做...
    C();
#endif
}
```

不同硬件平台/操作系统/种类的编译器，在预编译时，会附带一些预先定义的宏（predef），以对源代码进行“筛选”。

完整的预定义宏（predef）列表，可见以下链接。

> https://github.com/cpredef/predef

编译，汇编

编译则是将预处理文件转换成汇编代码文件（`.s` 文件）的过程，具体的步骤主要有：词法分析 -> 语法分析 -> 语义分析及相关的优化 -> 中间代码生成 -> 目标代码生成。

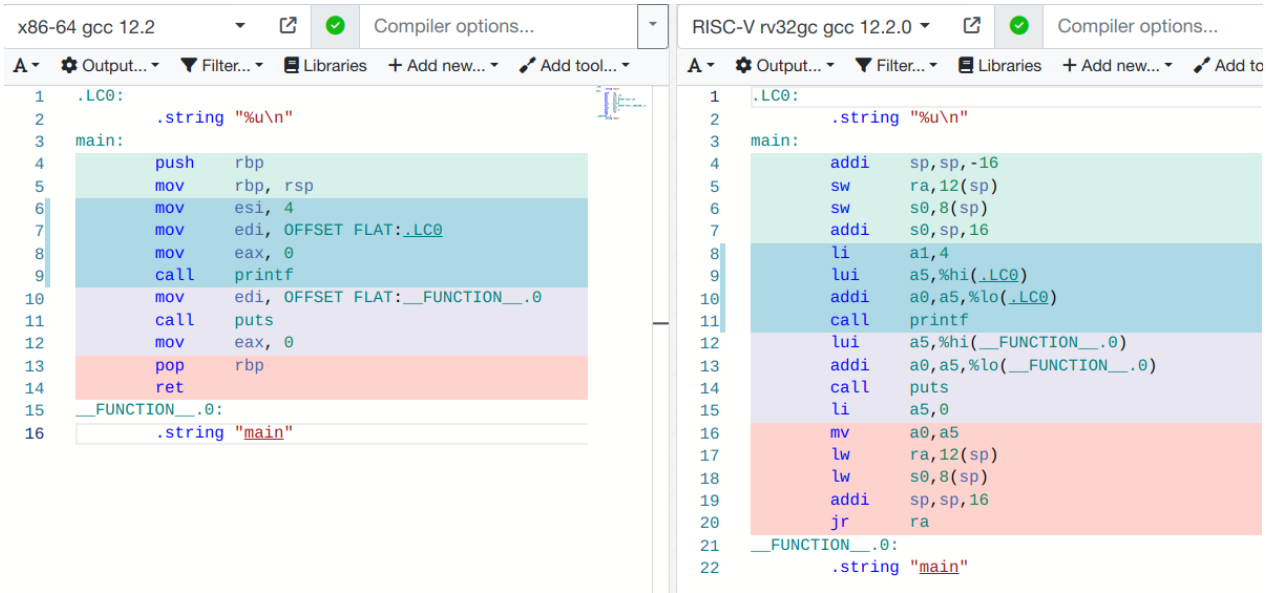
汇编阶段是将汇编代码编译成可重定位文件（Relocatable file）。汇编器（as）将汇编语言翻译成机器语言指令，把这些指令和其他附加的信息打包成可重定位文件（ `.o` 文件）。可重定位文件是一个二进制文件，它的字节编码是机器语言指令而不是字符。

从 C 语言代码变成汇编语言代码，是一个复杂的过程。

从汇编语言变成机器指令，是一个相对简单的过程。因为 *汇编语言* 是二进制指令的文本形式，与机器指令是一一对应的关系。从上图的右侧，可以明显的看出这种关系。

实际上，汇编语言的另一个名字是：机器指令助记符。汇编器（广义编译器的一部分）“忠实”地完成这一对一的替换过程，把汇编语言文本序列替换为二进制机器指令序列。

不同硬件平台，有着不同的机器指令，因此也就有着不同汇编语言。



上图是对同一段 C 代码编译的结果，左侧是 X86 平台，右侧是 RISC-V 平台，可以明显看到汇编指令的不同。

链接

任何大型软件都不可能只有一个源文件，通常有许多源代码文件，分别进行预处理，编译，汇编，在最后把他们链接到一起，形成一个可执行文件。

链接阶段就是把若干个可重定位文件(`.o` 文件)整合成一个可执行文件（Executable file）。上面已经提到，正是因为函数的实现可能分布在多个文件中，所以在链接阶段，我们需要提供函数实现的位置，然后最终生成可执行文件。链接阶段是通过链接器(ld)完成的，链接器将输入的可重定位文件加工后合成一个可执行文件。

在这一阶段，我们可以通过 `.ld` 文件，手动地，更加精细地控制链接过程。

如图是 XV6 `kernel/kernel.ld` 的一部分。

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

SECTIONS
{
  /*
   * ensure that entry.S / _entry is at 0x80000000,
   * where qemu's -kernel jumps.
   */
  . = 0x80000000;        You, 3周前 • first commit

  .text : {
    *(.text .text.*)
    . = ALIGN(0x1000);
    _trampoline = .;
    *(trampsec)
    . = ALIGN(0x1000);
    ASSERT(. - _trampoline == 0x1000, "error: trampo
    PROVIDE(etext = .);
  }

  .rodata : {
    . = ALIGN(16);
    *(.srodata .srodata.*) /* do not need to disting
    . = ALIGN(16);
    *(.rodata .rodata.*)
  }
```

有几个要点，需要介绍一下：

- `. = 0x80000000`
- 链接生成的可执行程序，在运行时，要被放到什么内存里的什么位置。
  - 指定这一点，对于操作系统而言十分重要，其具体的值是由硬件平台决定的。

    - 在操作系统启动部分，这一点还会再讲。
- `. = ALIGN(0x1000)`
- 控制程序段的内存对齐（又称字节对齐）要求
    - 对齐，又称字节对齐，内存对齐（alignment）表示：某对象的起始地址必须整除该对象的对齐要求.
    - 为什么要对齐？ https://zhuanlan.zhihu.com/p/598708950
- `PROVIDE(etext = .)`
- 提供了一个变量符号，可以在编写代码时用到。
    - 变量 `etext` 其值就是该位置的内存地址
- `ENTRY( _entry )`
- 在第二行。
    - 程序的入口，也就是程序从哪里开始被执行。
    - 对于一个普通程序，你可能认为，程序从 `main` 函数开始执行，但实际并不是。`libc` 在 `main` 执行之前做了很多事情，最后才跳到 `main` 开始执行。具体可见链接 https://embeddedartistry.com/blog/2019/04/08/a-general-overview-of-what-happens-before-main/
    - 对于操作系统也是如此，在操作系统的 `main` 执行之前，也做了一些事情。

链接是一个复杂的内容，了解更多知识可以看以下书籍。

> 《程序员的自我修养（链接，装载，库）》

最终，完成这些过程，操作系统的编译就完成了。操作系统从众多源文件变成了一个二进制文件。

以 XV6 为例，其编译完成后，最终就生成了 `kernel/kernel` 这一个二进制文件，ELF 格式，RISC-V 平台，静态链接，大小为 260KiB。

```
❯ file kernel/kernel
kernel/kernel: ELF 64-bit LSB executable, UCB RISC-V, RVC, double-float ABI, version 1 (SYSV), statically linked, with debug_info, not stripped
❯ du -sh kernel/kernel
260K    kernel/kernel
```

这就是操作系统，在编译之后，运行之前的最终样子，是不是和一个普通软件很类似。

但上文，我们也提到，操作系统和普通软件的编译过程存在一些区别。

和普通软件编译过程的区别

其最主要的区别就是，我们编写普通软件的平台和运行普通软件的平台很可能是相同的，而编写操作系统的平台和运行操作系统的平台很可能是不同的。

这也就引入了一个概念：交叉编译。

交叉编译是指在一台计算机上生成另一种架构或操作系统下的可执行文件或库的过程。它通常用于开发跨平台应用程序或在嵌入式设备中使用。

在传统编译中，源代码被编译成针对本地计算机架构和操作系统的可执行文件或库。但当需要将程序在另一种操作系统或架构上运行时，就需要进行交叉编译。这时，编译器需要生成针对目标架构或操作系统的二进制代码，而不是本地计算机的二进制代码。

例如，在一个 x86架构的 Linux 计算机上编写了一个 C++程序，要在 ARM 架构的嵌入式设备上运行，就需要进行交叉编译。在这种情况下，需要使用 ARM 架构的编译器来生成可在 ARM 设备上运行的二进制代码。

我们在编译 XV6 时就用到了 `riscv64-linux-gnu-gcc` ，该软件就是运行在 X86 平台上，但目标平台是 RISC-V 平台，也就是说生成的汇编语言，机器指令都是对应 RISC-V 平台的。

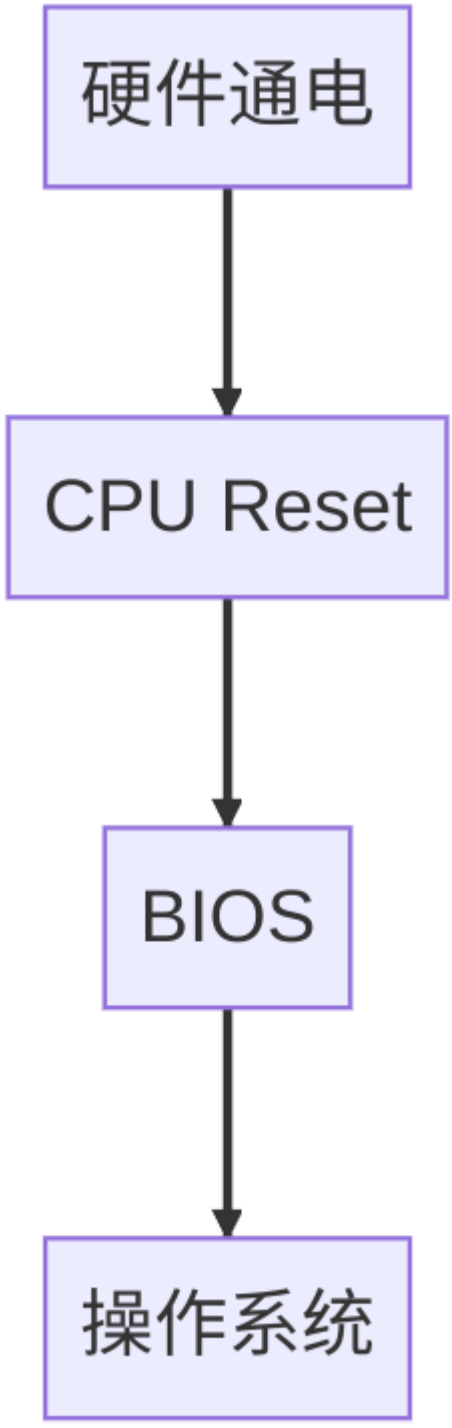
另一个区别是，操作系统是运行在裸机（Bare metal）上的，操作系统运行的时候，什么都没有，没有 `printf` ，没有 `malloc` 等等 … 这些东西需要我们自己编写代码实现。

但编译器在默认情况下，总是认为我们在写普通软件，会链接上 `libc` 库。因此，我们在编译的时候要额外的告诉编译器：不要带上这些东西（通过编译选项 `-mcmodel=medany -ffreestanding -fno-common -nostdlib` ）。

加载操作系统

以 XV6 为例，上文提到，编译完成后我们得到了一个二进制可执行文件 `kernel/kernel` 。接下来就来讲讲，操作系统是如何被硬件平台加载的。

其大致流程如下图



硬件通电

插上电源，按下开机按钮。

CPU Reset

通电后，主板向 CPU 发出 reset 信号。所谓的 Reset 其实就是 CPU 把所有寄存器都赋了一个初始值。

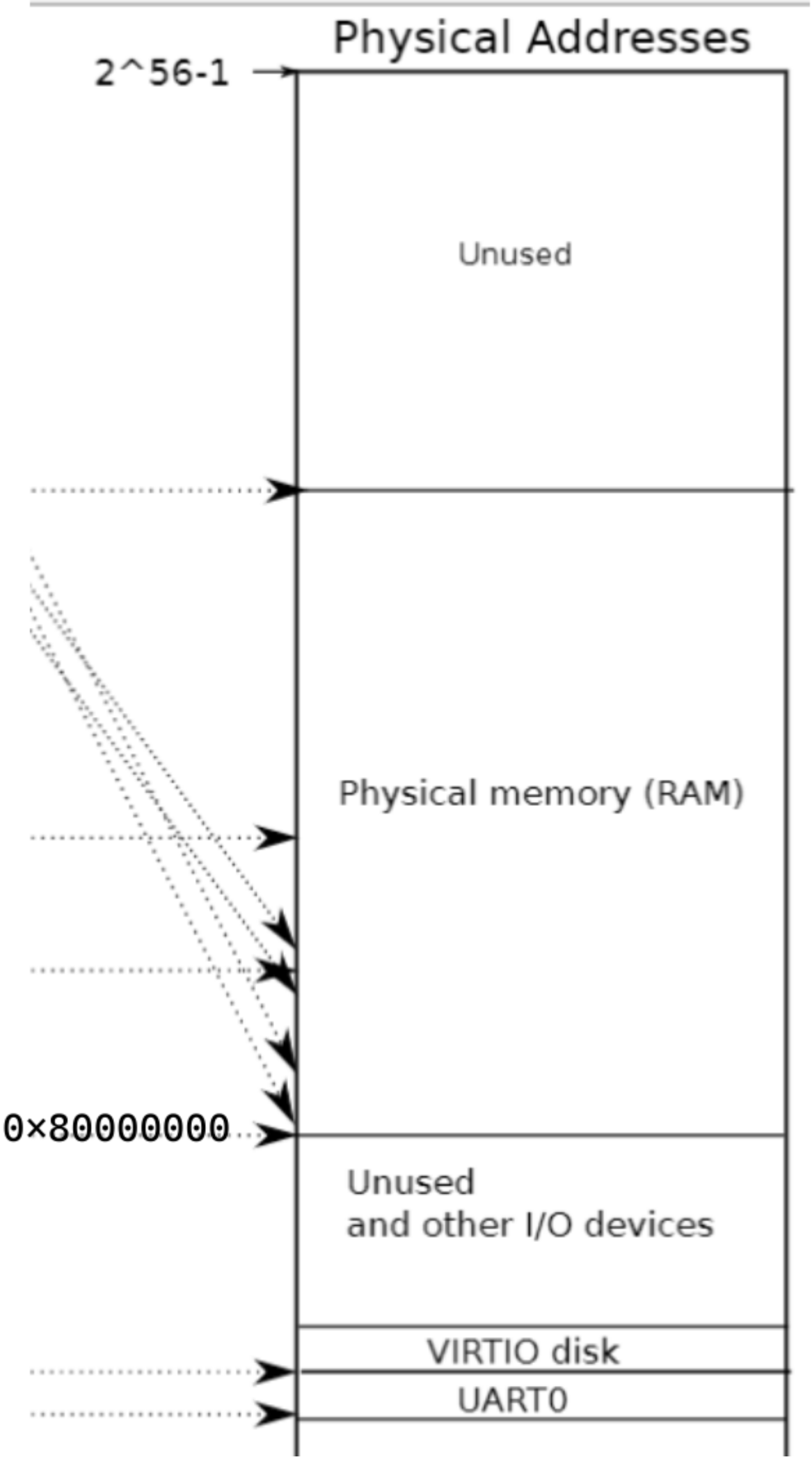
我们主要关注 `pc` 寄存器的变化， `pc` 寄存器指向 CPU 将要执行的代码的地址。

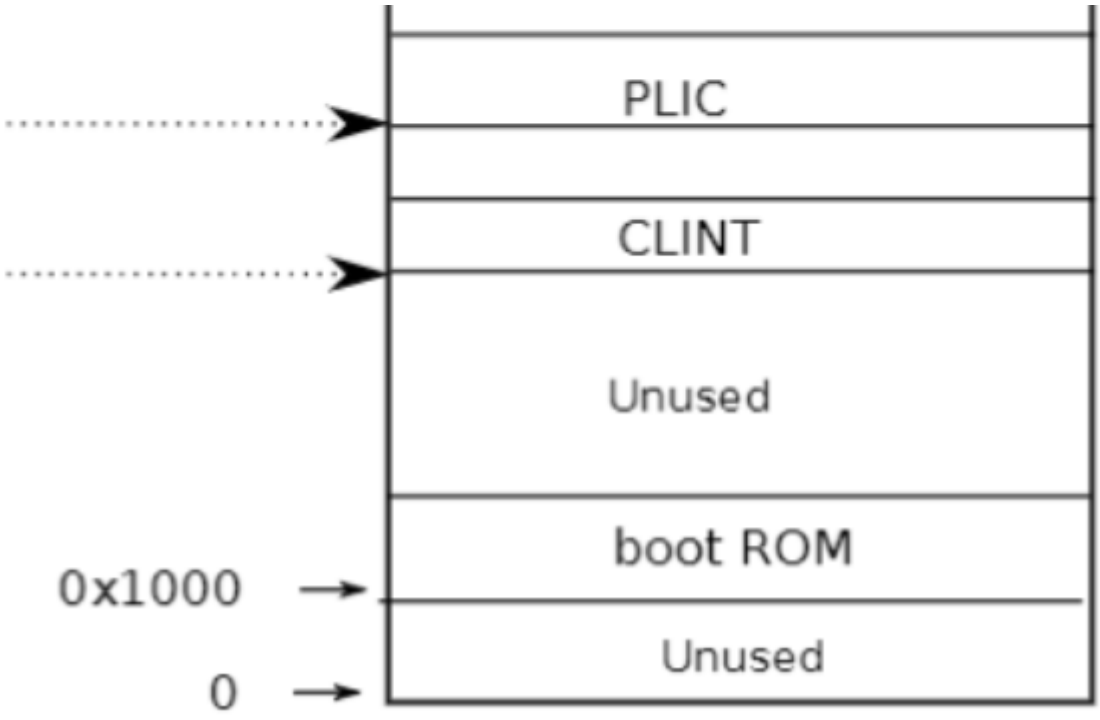
- 在 X86 平台上，CPU 在 Reset 后会跳到 `0xFFFFFFF0` 地址去执行。
- 在 RISC-V 平台上，CPU 在 Reset 后会跳到 `0x1000` 地址去执行。

回到 XV6，通过 `kernel.ld` 我们把操作系统放到了 `0x80000000` 地址。

RISC-V 的 CPU Reset 之后，其 `pc` 寄存器是 `0x1000`，这离操作系统的 `0x80000000` 还远。

Qemu 模拟 RISC-V 的内存地址空间如下图


Physical Addresses
2^56-1
Unused
Physical memory (RAM)
0×80000000
Unused
and other I/O devices
VIRTIO disk
UART0



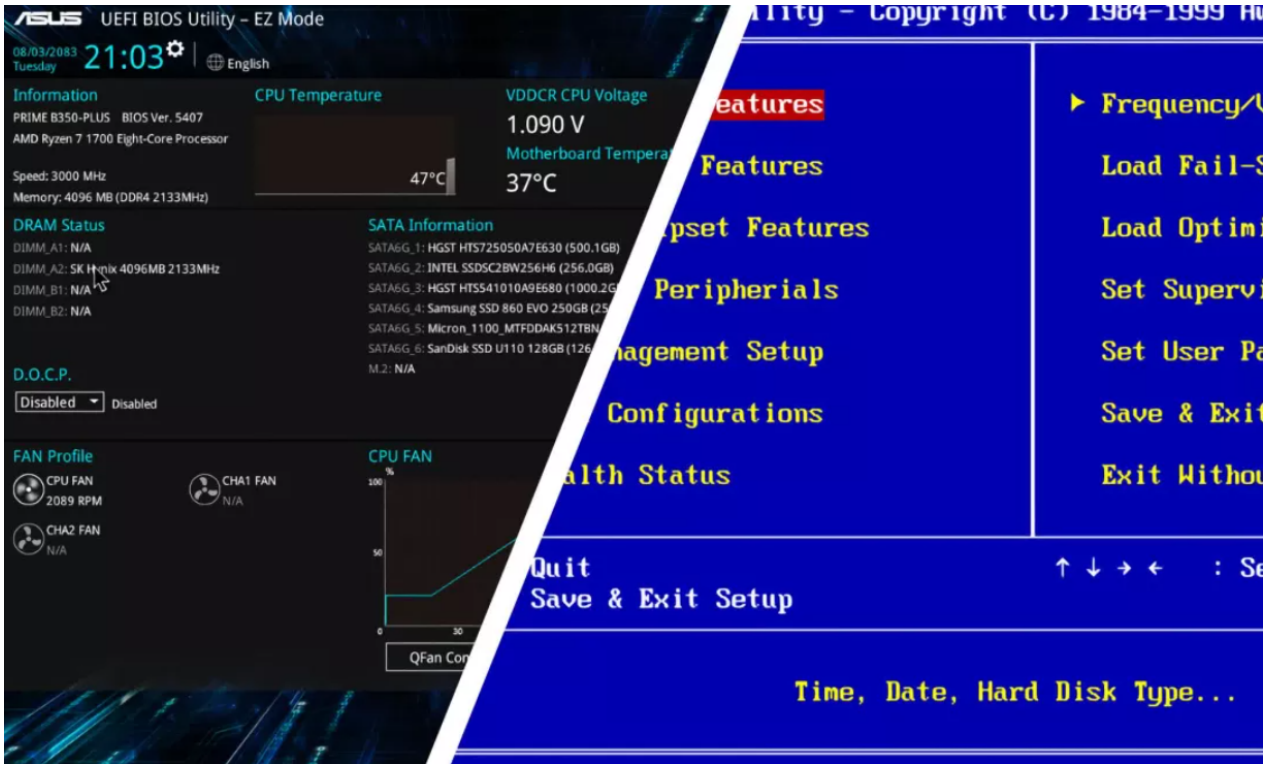
BIOS

RISC-V 的 CPU Reset 之后，其 `pc` 寄存器是 `0x1000`。而在内存地址 `0x1000` 放着的程序就是 BIOS，这一段程序通常放在主板的 ROM 里。

BIOS（Basic Input/Output System，基本输入输出系统）是计算机主板上的一种固件程序，它在计算机启动时第一个运行并负责初始化硬件设备、执行 POST（Power-On Self-Test，自检）和启动 Boot Loader。

BIOS最早出现在IBM PC及其兼容机中，它是一种低级别的软件，常常被称为“系统引导程序”的前身。BIOS通常存储在主板上的ROM芯片中，因此也被称为“ROM BIOS”。

当计算机开机时，CPU 会首先跳转到 BIOS 程序所在的地址，并且 BIOS 会按照预先设定的顺序和设置，初始化硬件设备，如内存、CPU、硬盘、显卡等，并完成 POST 过程，以确认硬件设备是否正常工作。之后 BIOS 会加载 Boot Loader 到内存中，让操作系统开始运行。



上图是 PC 机上典型的 BIOS 界面，BIOS 因主板而异。

我们使用 Qemu 模拟 RISC-V 平台，运行 XV6，来看看 Qemu 是如何实现 BIOS 的。

> https://github.com/slavaim/riscv-notes/blob/master/bbl/boot.md

```
(gdb) x/2i 0x1000
   0x1000:  auipc   t0,0x7ffff
   0x1004:  jr  t0
```

> `auipc` instruction moves its immediate value 12 bits to the left and adds to the current PC , so `t0 = (0x7ffff<<12)+ 0x1000 = 0x80000000`

经过 `boot rom` 的跳转， `pc` 变成了 `0x80000000` ，也就是 XV6 操作系统内核所在的位置。

此时，CPU 真正开始去执行操作系统的代码，xv6从 `entry.S` 开始 OS 的第一行代码。

可以说，Qemu 的 BIOS 什么也没做，只是简单的跳到了 `0x80000000` 位置。

而真实世界中的流程要复杂的多，从 Reset 到把控制权交给 OS，要经过众多流程，下图是一个简要说明。

RISC-V 介绍

在介绍操作系统是如何运行起来的之前，必须要介绍下 RISC-V 平台。

RISC-V（Reduced Instruction Set Computing Five）是一种基于精简指令集（RISC）架构的开放式指令集架构 (ISA)，它具有以下几个特点：

1. 开放性：RISC-V 的 ISA 是开放的，任何人都可以使用它来设计、制造和销售处理器芯片。这与其他指令集架构如 x86、ARM 等商业和专有 ISA 不同。

2. 可扩展性：RISC-V 支持可扩展性，允许用户添加自定义指令来满足某些应用场合的需要。这使得 RISC-V 对于特定应用场景的优化更加容易。
3. 简单性：RISC-V 的指令集非常简洁，同时保持了必要的灵活性和功能。这使得 RISC-V 易于实现、易于调试、易于测试和易于教授。
4. 兼容性：在 RISC-V ISA 中，每个指令都有一个唯一的编号，这意味着软件可以在不同的 RISC-V 处理器上运行而无需针对特定的硬件进行编译或调整。
5. 高性能：RISC-V 支持各种先进的技术和功能，如超标量执行、乱序执行、分支预测、向量处理等，这使得它在高性能计算领域中也具有很大的应用潜力。

RISC-V 的开放式设计和可扩展性使其成为适用于各种应用场景的理想处理器架构。同时，由于其简单性和兼容性，它也非常适合嵌入式系统和物联网设备等资源受限的领域。

RISC-V Assembly

> https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md

RISC-V 体系结构中有 32 个通用寄存器（GPR），它们的编号为 x0 到 x31。其中，x0 寄存器始终包含值 0，不可被写入。

这些 GPR 可以用于存储程序计数器 PC、函数调用参数、局部变量和临时变量等。除了 x0 寄存器之外，其余 31 个寄存器在程序执行过程中可以任意修改和使用。在函数调用期间，一些寄存器可能会被用作参数传递和返回值存储。

此外，RISC-V 架构定义了一些控制和状态寄存器（Control and Status Register，CSR），用于配置或记录一些运行的状态。CSR 寄存器是处理器核内部的寄存器，使用其自己的地址编码空间和存储器寻址的地址区间完全无关系。

CSR 寄存器的访问采用专用的 CSR 指令，包括 CSRRW、CSRRS、CSRRC、CSRRWI、CSRRSI 以及 CSRRCI 指令。

- 在硬件中还有一个寄存器叫做程序计数器（Program Counter Register）。
- 还有一堆控制 CPU 工作方式的寄存器，比如 SATP（Supervisor Address Translation and Protection）寄存器，它包含了指向 page table 的物理内存地址。
- 还有一些对于今天讨论非常重要的寄存器，比如 STVEC（Supervisor Trap Vector Base Address Register）寄存器，它指向了内核中处理 trap 的指令的起始地址。
- SEPC（Supervisor Exception Program Counter）寄存器，在 trap 的过程中保存程序计数器的值。
- SSRATCH（Supervisor Scratch Register）寄存器，这也是个非常重要的寄存器。
- 。。。

常见汇编指令：

- `lui`（Load Upper Immediate）：将 `rd` 设置为 32 位值，其中低 12 位为 0，高 20 位来自 U 类型立即数。
- `auipc`（Add Upper Immediate to Program Counter）：将 `rd` 设置为当前 PC 和 32 位值之和，其中低 12 位为 0，高 20 位来自 U 类型立即数。
- `li` 伪指令用于加载立即数值。
- `la` 伪指令用于加载符号地址。
- `l{b|h|w|d} <rd>, <symbol>`：从全局加载字节、半字、字或双字。
- `s{b|h|w|d} <rd>, <symbol>, <rt>`：向全局存储字节、半字、字或双字。
- `call <symbol>`：调用远程子例程。
- `call <rd>, <symbol>`：调用远程子例程。
    - 与 `call <symbol>` 相似，但使用 `<rd>` 保存返回地址。

- `jump <symbol>, <rt>`：跳转到远程例程。
- `jal`：跳转到地址并将返回地址放在 GPR 中。
- `jr`：跳转到地址。
- `jalr`：跳转到地址并将返回地址放在 GPR 中。

RISC-V CSRs (Control and Status Registers)

> https://balancetwk.github.io/2020/12/05/hexo-blog/RISC_V_Note/RISC-V%20%E7%89%B9%E6%9D%83%E6%A8%A1%E5%BC%8F%EF%BC%9A%E6%9C%BA%E5%99%A8%E6%A8%A1%E5%BC%8F/

CSR 寄存器控制着当前系统的状态，包括是否开中断，设置页表，设置特权模式等等。

例如：

- `stvec`：内核在这里写入其陷阱处理程序的地址；RISC-V 跳转到这里处理陷阱。
- `sepc`：当发生陷阱时，RISC-V 会在这里保存程序计数器 `pc`（因为 `pc` 会被 `stvec` 覆盖）。`sret`（从陷阱返回）指令会将 `sepc` 复制到 `pc`。内核可以写入 `sepc` 来控制 `sret` 的去向。
- `scause`： RISC-V 在这里放置一个描述陷阱原因的数字。
- `sscratch`：内核在这里放置了一个值，这个值在陷阱处理程序一开始就会派上用场。
  - Typically, `sscratch` is used to hold a pointer to the hart-local supervisor context while the hart is executing user code. At the beginning of a trap handler, `sscratch` is swapped with a user register to provide an initial working register.
- `sstatus`：其中的**SIE**位控制设备中断是否启用。如果内核清空**SIE**，RISC-V 将推迟设备中断，直到内核重新设置**SIE**。**SPP**位指示陷阱是来自用户模式还是管理模式，并控制 `sret` 返回的模式。
- ...

RISC-V Privilege modes

RISC-V 架构定义了几种工作模式，又称特权模式（Privileged Mode）：

- Machine Mode：机器模式，简称M Mode。
- Supervisor Mode：监督模式，简称S Mode。
- User Mode：用户模式，简称U Mode。

M,S 模式都有对应的 CSR 寄存器，如 `mepc` `sepc` ...

操作系统启动时跑在 machine mode，然后进入 supervisor mode，user mode。此后有中断时，又从 user mode 进入 supervisor mode。

可以理解为 OS 视角的内核态对应着 Supervisor mode，用户态对应着 User mode。

有一些指令，称为特权指令，这些指令仅能在 S-mode 下运行。

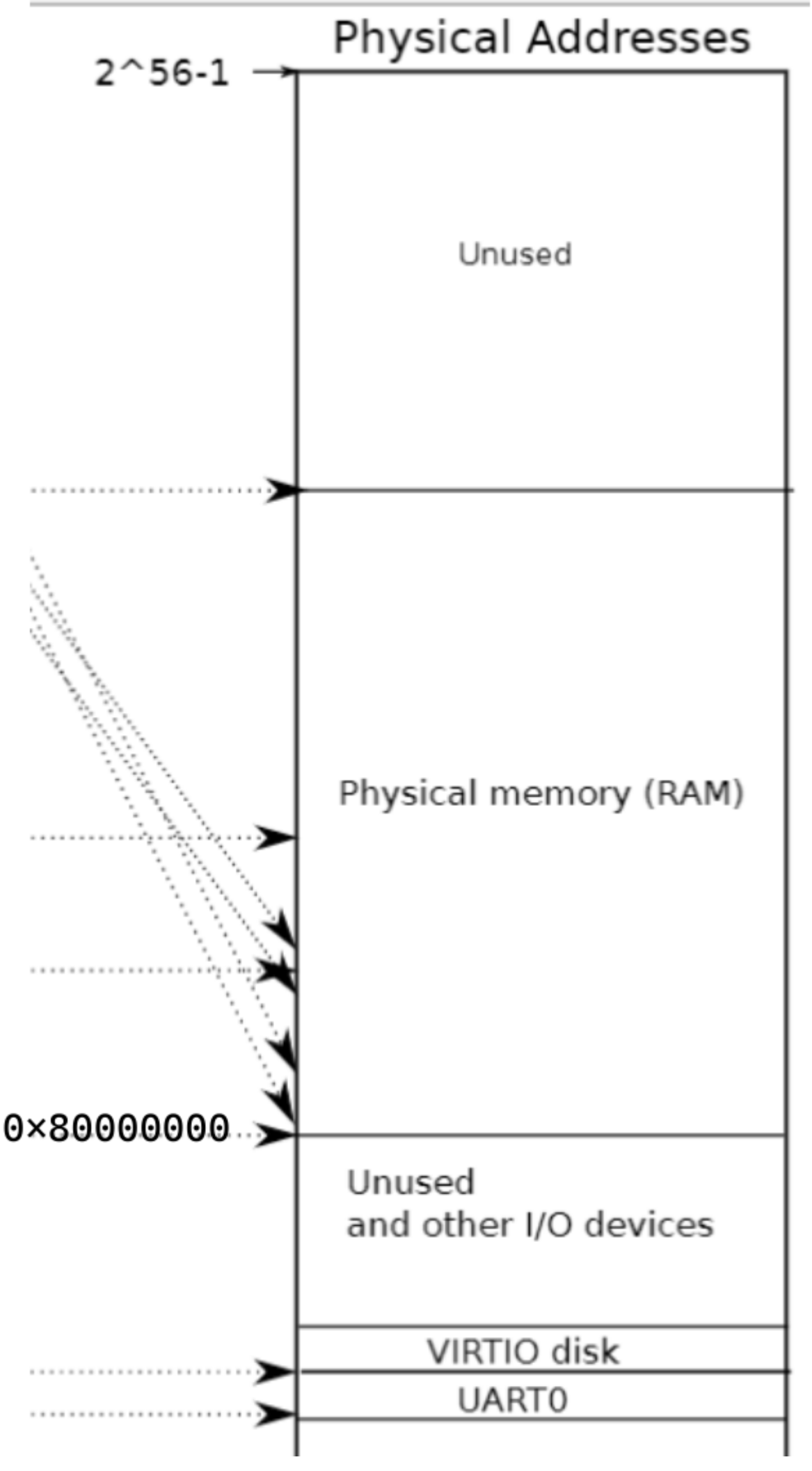
Memory-mapped IO

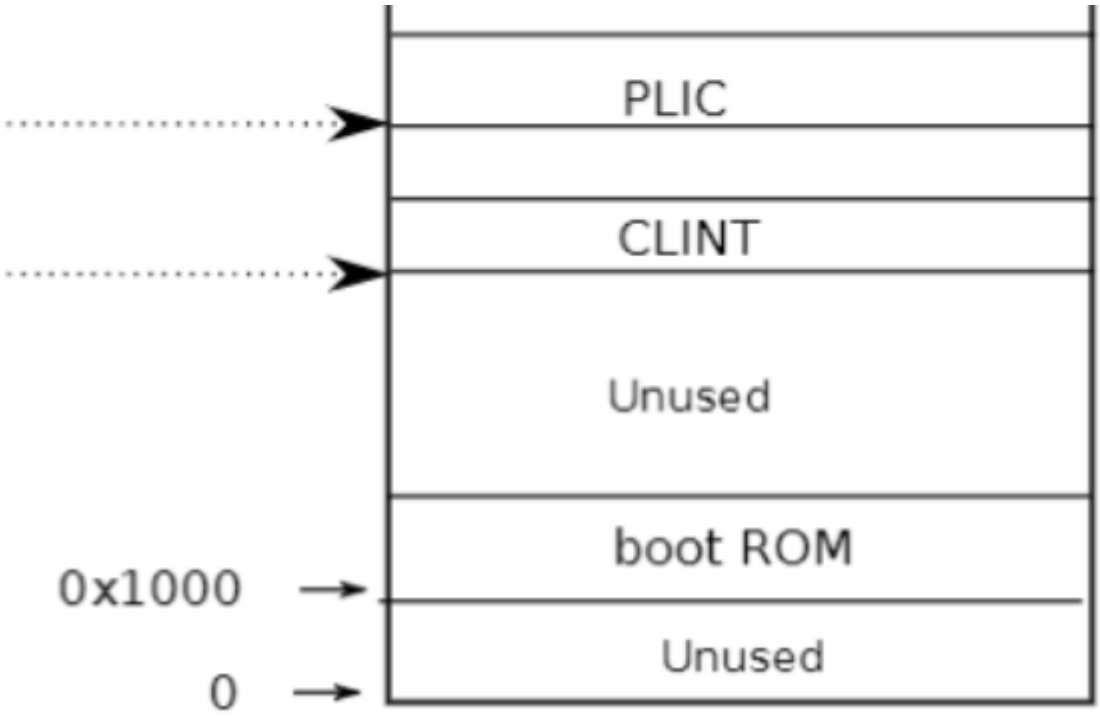
现代的体系结构大多使用 MMIO 的形式，即使用相同的地址总线来寻址内存和输入输出设备（简称 I/O 设备），前提是 I/O 设备上的设备内存和寄存器都已经被映射到内存空间的某个地址。这样当 CPU 访问某个地址的时候，可能是要访问某一部分物理内存，也可能是要访问 I/O 设备上的内存。因此，设备内存也可以通过内存访问指令来完成读写。

与之相对的是类似 8086 上，使用专门的 `IN` `OUT` 指令进行 IO 操作。

下图是 `qemu-riscv` 的物理地址映射图，完整的地址映射可见 riscv/virt.c#L75 。

其中，**ROM 被映射到 `0x1000`，DRAM 被映射到 `0x80000000` 地址**，这两个地址之间的为 IO 设备的映射。


Physical Addresses
2^56-1
Unused
Physical memory (RAM)
0×80000000
Unused
and other I/O devices
VIRTIO disk
UART0



下图是 Intel Xeon 的 MMIO 地址映射图，作为拓展了解。

运行操作系统

现在 `PC` 寄存器为 `0x80000000`，也就是我们操作系统内核所在的位置，CPU 即将执行 `_entry`（`_entry` 位于 `kernel/entry.S`）。

XV6 操作系统从 `entry.S` 开始拿到 CPU 控制权，到最终 `main.c` 里执行完初始化，跑起 shell 结束，算是完成了 OS 的启动。

`entry.S` --> `start.c` --> `main.c`

`entry.S`

Run in Machine Mode

xv6定义了 `NCPU=8` 核心处理器，每个核心(risc-v 称之为 hart[hardware thread])都并行的执行，都从 `Boot Rom` 跳转到 `entry.S`

此时，只能还只能执行汇编代码，我们还没有为每个CPU开辟stack，因此暂时不能执行C代码。

为每个核心设置好对应 `sp` 寄存器（即设置好stack），就可以跳转到 `start.c` 去执行C代码了。

注意，RISC-V要求stack地址向下增长，并且 `sp` 始终对齐到16字节。

> In the standard RISC-V calling convention, the stack grows downward and the stack pointer is always kept 16-byte aligned.

> From RISC-V Calling Convention

因此，`start.c`中的栈空间（实际就是个大数组）定义如下

`__attribute__ ((aligned (16))) char stack0[4096 * NCPU];`

每个核心有4KB大的执行栈。

至于，这个栈在内存中的真正地址是由编译器决定的，内核被加载到`0x80000000`地址，栈一定在这个地址之后，我这里是在`80008890 <_GLOBAL_OFFSET_TABLE_+0x8>`地址，它的真正地址并不需要关心。

```
_entry:
        # set up a stack for C.
        # stack0 is declared in start.c,
        # with a 4096-byte stack per CPU.
        # sp = stack0 + (hartid * 4096)
        la sp, stack0 # load address of stack0 to sp register
        li a0, 1024*4 # per hart has 4kB space of stack.
        csrr a1, mhartid # get current hart id to a1
        addi a1, a1, 1 # the stack grows downward, so `sp` of hart0 is stack0 +
4096*(0+1)
        mul a0, a0, a1
        add sp, sp, a0 # sp of hart0 is stack0 + 4096*1 , of hart1 is stack0 +
4096*2
        # jump to start() in start.c
        call start # Entry C language world, say bye to assembly
```

总而言之，`entry.S`做了一件事就是设置执行栈空间，以执行C代码。

`start.c`

Run in Machine Mode

1. 设置MSTATUS的MPP位
2. 设置MEPC为`main`地址
3. 暂时禁用页表翻译
4. 设置PMP(Physical Memory Protection)
5. 初始化时钟中断
    1. 中断部分最复杂了😭
6. 将hartid存到tp寄存器里
    1. `mhartid`寄存器是个M-mode寄存器，在S-mode下无法读取。而操作系统跑在S-mode下，因此OS要想知道是哪个hart在执行，就要用某种方式维护这个信息。XV6的做法是：在M-mode时将hartid存到`tp`寄存器里，此后对这个寄存器只读不写。
7. 执行`mret`变成S-mode，并进入`main.c`

总而言之，设置CSRs，为进入S-Mode做好准备。

`main.c`

Run in Supervisor Mode

所有 hart 都会运行到这，但只有 hart0去执行各部件的初始化。

> 在这些初始化的函数中，现在需要关注的是`userinit()`。在这个函数内，会完成第一个`User Mode`程序的创建。

> 这个函数的操作就是手动创建了一个用户进程，将这个进程设置为 `RUNNABLE` 状态，以便 cpu 进行调度执行。这里有个有意思的地方， `userinit()` 创建用户进程的方式，是直接使用程序的二进制形式来创建。
>
> 这个用户程序的二进制代码定义直接硬编码在 `initcode` 这个数组里，相应的可执行程序是 `user/initcode` 。
>
> `initcode` 的定义在 user/initcode.S。这个程序就是 Xv6 所创建的第一个用户态程序，从 `userinit()` 中可以看出，第一个用户态程序只能以靠手动的形式来创建，所以会要求它足够的**简单**。因此， `initcode` 更像一个**楔子**，用来引出真正的带有逻辑的用户程序。
>
> 从代码中可以看出， `initcode` 就是利用 `exec` 这个系统调用来创建出一个更为复杂的可执行程序。它会将 `init` 这个程序加载到 `a0` ，这个便是需要创建的程序的程序名，而后将参数 `argv` 加载到 `a1` 。 `exec` 这个系统调用的调用号是 7，所以需要将 7 加载到 `a7` 上。相关的操作完成后，就会调用 `ecall` 将控制权交回给操作系统。
>
> 通过上述的流程，可以正确地创建出真正意义上的第一个用户态程序， `init` 。这个程序的源代码在 `user/init.c` 。从代码里可以看出，这个程序唯一目的就是修改文件描述符，将 0 和 1 强制设定为标准输入输出。并且维持 `sh` 的运行。

`initcode init sh` 全都跑在用户态下

```
 [hardcode]      exec()                        exec()
user/initcode ------------>   kernel/init    ----------->   user/sh
```

这里请注意 `initcode.S` 的注释， `initcode` 是跑在用户态的。

```
# Initial process that execs /init.
# This code runs in user space.
```

那么，问题来了，在 `main.c` 的时候，XV6跑在S-mode，而 `user/initcode` 是第一个跑在U-mode的程序，那XV6是什么时候从S-mode进入U-mode的呢？

XV6第一次进入用户态

首先了解下，RISC-V的calling convention，与x86把返回地址压在栈里的方式不同，RISC-V把返回地址放到 `ra` 寄存器中。

在整个过程中，我们将追踪 `ra` 返回地址寄存器， `sp` 栈指针寄存器。

在 `userinit ()` 函数中，将创建 `initcode` 的进程，并设置该进程的一些信息，如下

```
  p->context.ra = (uint64)forkret;
  p->context.sp = p->kstack + PGSIZE;

  p->sz = PGSIZE;
  p->trapframe->epc = 0;      // user program counter
  p->trapframe->sp = PGSIZE; // user stack pointer
```

同为用户进程，但XV6对于这个 `initcode` 第一个用户进程有一些特别的处理， `uvmfirst(p->pagetable, initcode, sizeof(initcode))` ，XV6为 `initcode` 这个进程仅分配一个page的空间。

`initcode` 的地址空间如下，也就是说该进程的地址空间中没有 `heap` 段，且 `stack` 与 `.text` 段加在一起共占1个page（因为这个程序非常小，只是个胶水代码，这样做没问题）。

```
---------
trampoline
---------
trapframe
--------- 4096
stack
--------- ...
.text
--------- 0
```

至此， `initcode` 进程设置完成，在 `scheduler()` 中会真正调度到CPU上。

在 `swtch` 中，会用 `p->context` 中的内容替换当前寄存器。即 `ra=forkret` ， `sp=p->kstack + PGSIZE` ，也就是说执行完 `swtch` ，会跳到 `forktret()` 函数，使用的是内核栈。 *任何用户进程在初次启动时都会从这个forkret()开始*

> 学生提问：所以当swtch函数返回时，CPU会执行forkret中的指令，就像forkret刚刚调用了swtch函数并且返回了一样？
>
> Robert教授：是的，从switch返回就直接跳到了forkret的最开始位置。
>
> 学生提问：因吹斯听，我们会在其他场合调用forkret吗？还是说它只会用在这？
>
> Robert教授：是的，它只会在启动进程的时候以这种奇怪的方式运行。

在 `forkret()` 中会调用 `usertrapret()` 。

在 `usertrapret()` 中，会设置 `sepc` 寄存器为 `p->trapframe->epc` ，也就是 `0` 。然后将page table地址作为参数，调用 `trampoline.S中的userret` 函数。

在 `userret` 中，设置页表寄存器 `satp` ，也就是现在已经从内核地址空间切换到了用户进程地址空间（内核地址空间和用户地址空间的trampoline的虚拟地址相同，因此可以继续执行代码而不会出错）。并从 `p->trapframe` 加载寄存器， `ra=p->trapframe->ra` 是一个没有意义的值， `sp=p->trapframe->sp = PGSIZE` ，此时切换到了initcode进程的用户栈（见 `initcode` 的地址空间）。最后执行 `sret` 。

在 `userret` 中执行 `sret` ，会从S-mode进入U-mode，并用 `sepc` 寄存器值替换 `pc` 寄存器。此时 `pc=0` ，XV6进入了用户态，并从头开始执行 `initcode` 中的代码。

这就是XV6第一次从内核态进入用户态的过程。

`initcode` 使用 `exec()` 系统调用，调用 `kernel/init` 程序， `kernel/init` 调用 `user/sh` 程序，就是我们进入XV6看到的shell程序。

```
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img
,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

init: starting sh
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2305
cat            2 3 32048
echo           2 4 31016
forktest       2 5 15464
grep           2 6 35664
init           2 7 31560
```

实验

实验介绍

1. 准备 Linux 环境
2. 配置开发环境
3. 拉取代码
4. 启动 XV6
5. 配置 Vscode 调试环境

实验内容 - 配置环境

准备 Linux 环境

大概有以下几种选择：

- 实机安装 Linux 发行版，如 Arch,Ubuntu 等
- 在 Windows 系统下
- 使用 `VirtualBox`
    - 开源，免费，比较好用
    - 使用 `VMware` 虚拟机
    - player 版免费，pro 版付费，很好用
    - 使用 `Hyper-V` 虚拟机
    - Windows 家庭版默认没有，比较好用
    - 使用 `WSL2`
    - 默认没有 GUI 界面，很好用

Clone xv6 repo

下载xv6源代码

> https://jihulab.com/xv6n/xv6n

```
git clone https://jihulab.com/xv6n/xv6n.git

cd xv6n
```

Install Toolchains

需要的Tools如下:

- git
- make
- gcc
- perl
- qemu-system-riscv64
- riscv64-unknown-elf-gcc 或者 riscv64-linux-gnu-gcc
- riscv64-unknown-elf- 或者 riscv64-linux-gnu-
    - ld
    - objcopy
    - objdump
    - ... (这套东西一般叫binutils-riscv64-.....)

- gdb-multiarch

以ubuntu为例

```
sudo apt install binutils-riscv64-linux-gnu
sudo apt install gcc-riscv64-linux-gnu
sudo apt install gdb-multiarch
sudo apt install qemu-system-misc opensbi u-boot-qemu qemu-utils
```

实际上,你只需要跟着 `make qemu` 的报错,需要什么装什么就行了

make qemu

此时,在 `xv6n` 目录下,执行 `make qemu` ,正常的话就进入了 xv6.

退出qemu:

1. first press Ctrl + A (A is just key a, not the alt key),
2. then release the keys,
3. afterwards press X.

完成这一步,才能继续

make qemu-gdb

此时,在 `xv6n` 目录下,执行 `make qemu-gdb` ,进程会阻塞.

```
delta@ubuntukvm ~/xv6-riscv (riscv)> make qemu-gdb
*** Now run 'gdb' in another window.
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img
,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -S -gdb tcp::26000
```

另开一个终端,在 `xv6n` 目录下,执行 `gdb-multiarch kernel/kernel`

可能的问题

```
delta@ubuntukvm ~/xv6-riscv (riscv)> gdb-multiarch kernel/kernel
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from kernel/kernel...
warning: File "/home/delta/xv6-riscv/.gdbinit" auto-loading has been declined by your `auto-load safe-path' set
 to "$debugdir:$datadir/auto-load".
To enable execution of this file add
        add-auto-load-safe-path /home/delta/xv6-riscv/.gdbinit
line to your configuration file "/home/delta/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/home/delta/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
(gdb)
```

`make qemu-gdb` 在当前目录生成了 `.gdbinit` ,具体见 `Makefile` Line 166.

`gdb` 会默认首先执行当前目录下的 `.gdbinit`

上图中这一步被阻止了.

```
warning: File "/home/delta/xv6-riscv/.gdbinit" auto-loading has been declined by
your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
```

同样,正如他所说将

```
add-auto-load-safe-path /home/delta/xv6-riscv/.gdbinit
```

这一行加到 `/home/delta/.gdbinit` 这个文件里即可.

退出gdb快捷键 `ctrl+d` .

执行

```
echo "add-auto-load-safe-path /home/delta/xv6-riscv/.gdbinit" >>
"/home/delta/.gdbinit"
```

或者vim手动编辑也行.

之后再次运行 `gdb-multiarch kernel/kernel`

```
delta@ubuntukvm ~/xv6-riscv (riscv)> gdb-multiarch kernel/kernel
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.htm
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from kernel/kernel...
The target architecture is assumed to be riscv:rv64
0x0000000080000c70 in pop_off () at kernel/riscv.h:60
60        asm volatile("csrw sstatus, %0" : : "r" (x));
(gdb)
```

一切正常

配置Vscode

安装拓展 `C/C++`

> https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools

在 `.vscode` 目录下,创建以下两个文件

launch.json

```
// xv6n/.vscode/launch.json
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "xv6debug",
```

```
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceFolder}/kernel/kernel",
            "stopAtEntry": true,
            "cwd": "${workspaceFolder}",
            "miDebuggerServerAddress": "127.0.0.1:26000", //见.gdbinit 中 target
remote xxxx:xx
            "miDebuggerPath": "/usr/bin/gdb-multiarch", // which gdb-multiarch
            "MIMode": "gdb",
            "preLaunchTask": "xv6build"
        }
    ]
}
```

tasks.json

```
// xv6n/.vscode/tasks.json
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "xv6build",
            "type": "shell",
            "isBackground": true,
            "command": "make qemu-gdb",
            "problemMatcher": [
                {
                    "pattern": [
                        {
                            "regexp": ".",
                            "file": 1,
                            "location": 2,
                            "message": 3
                        }
                    ],
                    "background": {
                        "beginsPattern": ".*Now run 'gdb' in another window.",
                        // 要对应编译成功后,一句echo的内容. 此处对应 Makefile Line:170
                        "endsPattern": "."
                    }
                }
            ]
        }
    ]
}
```

按F5即可开始调试.

可能会发现报错如下

或者

```
> Executing task: make qemu-gdb <

*** Now run 'gdb' in another window.
qemu-system-riscv64 -machine virt -bios none -kernel kernel/ke
rnel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,form
at=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmi
o-bus.0 -S -gdb tcp::26000
qemu-system-riscv64: QEMU: Terminated via GDBstub

终端将被任务重用，按任意键关闭。
```

> https://stackoverflow.com/questions/5550555/qemu-terminated-via-gdbstub-error
> GDB automagically loads ~/.gdbinit.
> so when you load .gdbinit via --command=~/.gdbinit
> it runs the script twice,
> when it gets to the 2nd invocation of target remote localhost:1234
> gdb hangs up its initial connection, qemu quits,
> then gdb fails to reconnect to it because it is no longer running.

这是因为 `.gdbinit` 中有 `target remote 127.0.0.1:26000` ,这个文件会被gdb最先执行一遍.

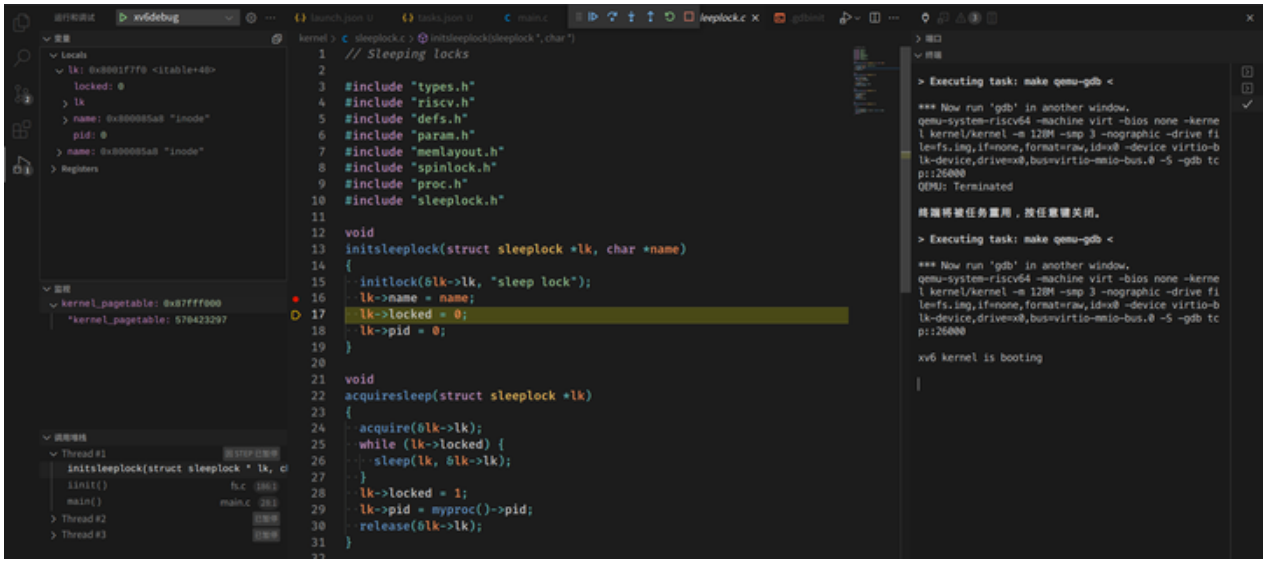
此外,我们在 `launch.json` 中指定了 `"miDebuggerServerAddress": "127.0.0.1:26000"` ,同一个remote address被配置了两次.

只需要把 `.gdbinit` 中的注释掉即可

```
set confirm off
set architecture riscv:rv64
@REM target remote 127.0.0.1:26000
symbol-file kernel/kernel
set disassemble-next-line auto
set riscv use-compressed-breakpoints yes
```

`@REM` 就是 `.gdbinit` 的注释符号.

要是你使用命令行gdb,记得再改回来.




```
./
├── .dir-locals.el
```

```
├── .editorconfig
├── .gdbinit.tmpl-riscv
├── .git
│   ├── ......
├── .gitignore
├── kernel
│   ├── bio.c
│   └── ......
├── LICENSE
├── Makefile
├── mkfs
│   └── mkfs.c
├── README
├── user
│   ├── cat.c
│   └── ......
└── .vscode
    ├── launch.json
    └── tasks.json
```

拓展实验内容

> 配置好环境，生成一个 Docker 镜像，方便分享给大家用

Tips：

- Docker 中文文档
- https://dockerdocs.cn/
- code-server 镜像, 网页版的Vscode
- https://hub.docker.com/r/linuxserver/code-server
    - 也许可以在此基础上，配置 XV6 开发环境