

从零构建操作系统：实验一报告

项目名称: riscv-os-exp1 日期: 2025年09月15日

一、系统设计部分

1.1 架构设计说明

本次实验的目标是构建一个能在QEMU virt 虚拟平台上运行的、最简化的64位RISC-V操作系统内核。其核心设计思想是**极简主义**，剥离所有复杂性，只保留引导和屏幕输出这两个最基本的功能，从而聚焦于操作系统启动的本质流程。

核心架构特性：

- 目标平台:** QEMU virt 模拟平台，RISC-V 64位 (RV64GC)。
- 运行模式:** 整个内核完全运行在**机器模式 (Machine Mode)**。这是RISC-V最高特权的模式，可以直接访问所有硬件资源。为了简化，没有像xv6那样切换到监督模式(Supervisor Mode)。
- 内存模型:** 采用**平坦物理内存模型**。内核直接运行在物理地址 **0x80000000**，不涉及虚拟内存、分页或内存保护。对硬件（如UART）的访问也是通过直接读写其固定的物理内存映射地址。
- 引导方式:** 依赖QEMU的 **-kernel** 参数，直接将编译好的ELF内核文件加载到内存中并执行。我们没有实现独立的bootloader。
- 驱动实现:** 包含一个基于**轮询 (Polling)** 方式的16550兼容UART串口驱动，用于向控制台输出字符。这是最简单的设备驱动模型，不涉及中断。
- 内核结构:** 单体内核，所有代码（启动汇编、C主函数、驱动）被链接成一个单一的ELF可执行文件。

1.2 关键数据结构

在本次极简实现中，没有复杂的C语言数据结构。关键的“数据”体现在内存布局和硬件接口上：

- 内核内存布局 (Kernel Memory Layout):** 由链接脚本 **kernel.ld** 定义，严格划分了程序在内存中的组织形式：

- .text:** 存放编译后的机器指令。
- .rodata:** 存放只读数据，如字符串常量。
- .data:** 存放已初始化的全局变量和静态变量。
- .bss:** 为未初始化的全局变量和静态变量预留的空间。在启动时由汇编代码清零。

- 启动栈 (Boot Stack):**

- 定义:** 在内核 **.bss** 段之后，通过链接脚本分配的一块固定大小 (4KB) 的连续物理内存区域。
- 作用:** 为即将运行的C语言环境提供栈空间。函数调用、参数传递、局部变量存储都依赖于此栈。栈指针 **sp** 在 **entry.S** 中被初始化为这块区域的最高地址。

- UART寄存器接口:**

- 虽然不是传统意义上的数据结构，但这些内存映射的寄存器是内核与硬件交互的接口。
- THR (Transmit Holding Register):** 地址 **0x10000000**，写入该寄存器即可发送一个字符。

- **LSR (Line Status Register)**: 地址 `0x10000005`，通过读取该寄存器的特定位（第5位），可以判断发送器是否空闲。

1.3 与xv6对比分析

虽然我们的目标是参考xv6，但为了实现最小化，我们的设计在多个核心方面对其进行了大幅简化。

特性	本次实现 (Minimal OS)	xv6-riscv	分析
特权级	仅使用机器模式 (M-Mode)	M-Mode (启动引导) → 监督模式 (S-Mode) (内核主体)	本项目简化了特权级切换，避免了处理 <code>mret</code> 、 <code>medeleg</code> 等复杂CSR。这使得我们可以专注于最核心的启动流程。
引导流程	无Bootloader，依赖QEMU直接加载ELF文件	包含一个独立的Bootloader，负责加载内核	简化了引导链，直接从内核入口 <code>_start</code> 开始，降低了初学难度。
多核支持	单核设计，只设置了一个全局栈	支持多核， <code>entry.S</code> 会为每个hart(核)分配独立的栈空间	本项目 <code>entry.S</code> 的栈设置逻辑非常简单。xv6必须处理多核启动时的同步与资源分配问题，复杂度远高于本项目。
内存管理	静态物理内存，无分页或虚拟内存	实现了完整的分页机制，使用页表进行地址转换	这是最显著的区别之一。我们的内核直接操作物理地址，而xv6构建了一个复杂的虚拟内存系统，这是现代操作系统的基础。
设备驱动	仅有一个极简的轮询模式UART驱动	驱动更完善，支持中断，设备抽象层次更高	我们的驱动仅用于输出 "Hello OS"。xv6的驱动需要处理中断、并发等问题，为上层提供统一的设备文件接口。
整体目标	验证核心启动流程，打印 "Hello OS"	一个功能完整的、类UNIX的教学操作系统（支持进程、文件系统等）	我们的目标是“从0到1”，而xv6的目标是“从1到100”，展示一个完整操作系统的全貌。

1.4 设计决策理由

- **决策：为何只在机器模式下运行？**
 - **理由：为了简化。** 实验一的核心目标是理解从硬件加电到执行第一行C代码的全过程。引入特权级切换（M-mode到S-mode）会涉及到对 `mstatus`, `mepc`, `medeleg` 等多个控制状态寄存器(CSR)的复杂配置，以及 `mret` 指令的使用。这会分散学习重点。停留在M-mode可以让我们用最少的代码实现目标。
- **决策：为何不实现Bootloader？**

- **理由：为了聚焦。** 同样是为了简化学习路径。QEMU的`-bios none -kernel <elf_file>`选项完美地模拟了一个已将内核加载到内存的环境，让我们可以直接从内核的第一行指令开始调试和分析，而无需关心加载过程本身的复杂性。
 - **决策：为何栈大小选择4KB？**
 - **理由：这是一个安全且常规的选择。** 4KB是RISC-V体系结构中一个常见的页大小。对于我们这个没有深层函数调用和巨大局部变量的极简内核来说，4KB的栈空间绰绰有余，可以有效避免栈溢出。
 - **决策：为何内核起始地址是 0x80000000？**
 - **理由：遵循QEMU virt平台的内存布局约定。** 在QEMU的virt机器模型中，物理内存（DRAM）从0x80000000开始。将内核放在这个地址是标准做法，确保了硬件能正确找到并执行我们的代码。
-

二、实验过程部分

2.1 实现步骤记录

1. **环境搭建 (实验0):** 安装`qemu-system-riscv64`和`riscv64-unknown-elf-gcc`工具链，并验证其可用性。
2. **项目初始化:** 创建`riscv-os`目录，使用`git init`初始化仓库，并创建`kernel`, `Makefile`, `README.md`等文件和目录结构。
3. **编写链接脚本 (`kernel.ld`):**
 - 定义入口点为 `_start`。
 - 设置程序基地址为 `0x80000000`。
 - 依次定义 `.text`, `.rodata`, `.data`, `.bss` 段。
 - 在 `.bss` 段前后定义了 `sbss` 和 `end` 符号，用于后续的BSS清零操作。
 - 在最后分配了4KB空间作为栈，并定义了 `stack_top` 符号。
4. **实现启动汇编 (`kernel.entry.S`):**
 - 创建全局可见的 `_start` 标签。
 - 第一步，使用 `la sp, stack_top` 指令将链接脚本中定义的栈顶地址加载到栈指针 `sp` 寄存器。
 - 第二步，实现BSS清零循环。使用 `la` 加载 `sbss` 和 `end` 的地址，通过循环和 `sd` (store doubleword) 指令将这块内存区域逐字（8字节）清零。
 - 第三步，使用 `call kmain` 指令，无条件跳转到C语言主函数。
 - 最后，添加了一个 `_hang: j _hang` 的死循环，防止 `kmain` 意外返回后CPU“跑飞”。
5. **实现UART驱动 (`kernel/uart.c`):**
 - 通过宏定义了UART的基地址和关键寄存器偏移。
 - 实现了 `uart_putc()` 函数。该函数通过一个while循环，不断读取LSR寄存器的值，判断发送器是否空闲。一旦空闲，就将字符写入THR寄存器。这里的关键是使用了`volatile`关键字，防止编译器优化掉对内存地址的读写。
 - 基于 `uart_putc()` 实现了 `uart_puts()`，用于输出整个字符串。
6. **实现C主函数 (`kernel/main.c`):**
 - 编写 `kmain` 函数。
 - 在函数内部，调用 `uart_puts("Hello OS from RISC-V!\n")` 来输出信息。
 - 函数最后进入 `while(1)` 死循环，使内核保持运行状态。
7. **编写Makefile:**

- 定义了工具链、编译参数 `CFLAGS` 和链接参数 `LDFLAGS`。
- 编写了链接生成 `kernel.elf` 的规则。
- 编写了将 `.c` 和 `.S` 文件编译为 `.o` 文件的通用规则。
- 添加了 `make qemu` 伪目标，用于自动化启动QEMU。
- 添加了 `make clean` 用于清理生成文件。

8. **编译与测试:** 执行 `make qemu`，观察到QEMU窗口输出预期的字符串，实验成功。

2.2 问题与解决方案

• 问题1: QEMU启动后黑屏，没有任何输出。

◦ 排查过程:

1. 首先怀疑链接脚本地址问题。使用 `riscv64-unknown-elf-objdump -h kernel.elf` 检查各段的起始地址，确认 `.text` 段确实在 `0x80000000`。
2. 其次怀疑UART地址错误。查阅QEMU `virt machine` 文档，确认UART基地址 `0x10000000` 无误。
3. 最终发现是QEMU启动命令问题。最初忘记添加 `-bios none` 参数，导致QEMU默认加载了它内置的OpenSBI固件，该固件接管了硬件初始化和控制权，我们的内核代码没有得到执行。

- **解决方案:** 在 `Makefile` 的 `qemu` 命令中添加 `-bios none` 参数，让QEMU直接执行 `-kernel` 指定的ELF文件。

• 问题2: 编译时链接器报错 "undefined reference to `_start`".

- **排查过程:** 这个错误很明确，链接器找不到 `ENTRY` 点。检查 `kernel.ld` 文件，`ENTRY(_start)` 书写正确。检查 `kernel/entry.S` 文件，发现 `_start` 标签前缺少了 `.globl _start` 声明。
- **解决方案:** 在 `entry.S` 中的 `_start:` 标签前添加 `.globl _start`，将该符号声明为全局可见，这样链接器就能找到它了。

2.3 源码理解总结 (回答任务中的问题)

任务一

• 问：为什么第一条指令是设置栈指针？

- **答：** 因为后续的 `call kmain` 指令以及 `kmain` 函数本身都需要使用栈。`call` 指令会把返回地址压入栈中，C函数会使用栈来存储局部变量、传递参数和保存寄存器。如果不在调用C函数前设置好一个合法可用的栈，`call` 指令和后续的C代码会因访问无效内存地址而立刻崩溃。**先建立环境，再运行程序**，这是基本原则。

• 问：`la sp, stack0` (对应我们代码中的 `la sp, stack_top`) 中的 `stack0` 在哪里定义？

- **答：** 它在链接脚本 `kernel.ld` 中定义。在我们的实现中，对应的符号是 `stack_top`。我们通过 `PROVIDE(stack_top = .);` 这行代码，将内核文件末尾加上一个栈大小（4KB）之后的地址赋值给 `stack_top` 这个符号。汇编代码通过这个符号，就能知道栈的最高地址在哪里。

• 问：为什么要清零BSS段？

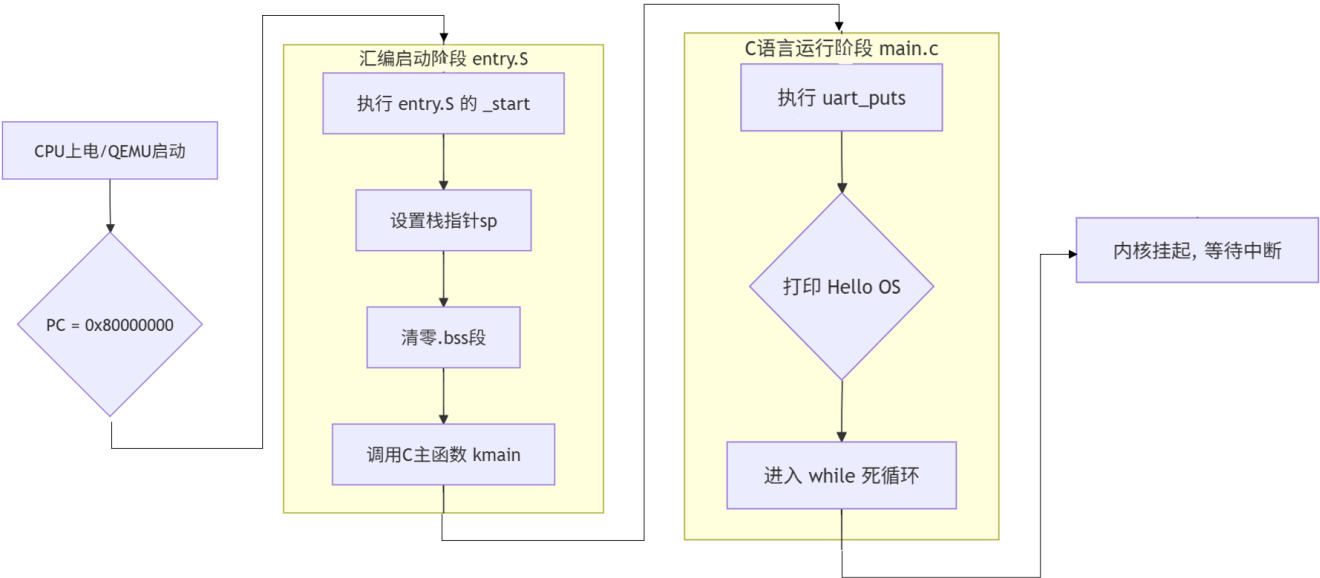
- **答：** 这是C语言标准的要求。标准规定，任何未被显式初始化的静态变量或全局变量，其初始值必须为0。编译器在生成代码时，会将这类变量放入BSS段。由于可执行文件本身不存储这些0（为了

节省空间），所以操作系统内核必须在执行C代码前，手动将BSS段对应的内存区域全部清零，以确保C语言环境的正确性。

- 问：如何从汇编跳转到C函数？
 - 答：使用 `call` 指令。`call kmain` 这条指令做了两件事：1) 将下一条指令的地址（即返回地址）存入 `ra` (Return Address) 寄存器；2) 无条件跳转到 `kmain` 标签所在的地址执行。
- 问： `ENTRY(_entry)` (对应 `ENTRY(_start)`) 的作用是什么？
 - 答：它告诉链接器(LD)，当这个程序被加载到内存并执行时，第一条需要执行的指令位于 `_start` 这个标签处。它设定了整个程序的入口点。
- 问：为什么代码段要放在 `0x80000000`？
 - 答：因为这是QEMU `virt` 模拟平台物理内存（DRAM）的起始地址。当QEMU加载我们的内核时，它会把ELF文件的内容映射到这个基地址上。CPU的程序计数器(PC)也会被设置为这个地址，从而开始执行我们的代码。
- 问： `etext`、 `edata`、 `end` 符号有什么用途？
 - 答：这些是由链接器在链接过程中自动生成的符号，用于标记各个段的结束位置，它们在C和汇编代码中非常有用。
 - `etext`: 标记代码段 `.text` 的结束位置。
 - `edata`: 标记已初始化数据段 `.data` 的结束位置。
 - `end`: 标记未初始化数据段 `.bss` 的结束位置，也通常被认为是整个内核静态镜像的结束位置。在我们的代码中， `end` 和 `sbss` 一起被用来确定需要清零的BSS内存区域。

任务二

流程图



内存设计方案

--



必需的硬件初始化步骤

1. 设置栈指针 (sp): 这是最关键的一步。我们必须在汇编代码中, 将一个有效的、可读写的内存地址 (即我们规划的栈区顶部地址stack_top) 加载到sp寄存器中。这是所有函数调用能够正常工作的前提。

2. 准备内存区域 (清零BSS): 虽然这是对内存的操作, 但它也是为了满足C语言运行环境的硬件状态要求, 确保特定内存区域的内容是确定的 (全为0)。

3. 串口设备: 无需显式初始化配置。QEMU virt平台模拟的16550 UART设备在上电后有默认的配置 (如波特率等), 这些默认配置足以让我们直接向其发送字符。我们只需要在使用时遵循其工作协议 (检查状态再写入) 即可。

- 问: 栈应该放在内存的哪个位置? 需要多大?

◦ 答:

▪ 位置: 栈被放置在内核镜像的末尾, 即**.bss段**之后。这是一种简单而安全的策略, 因为它确保了栈空间与内核的代码和数据区完全分离, 避免了栈向下增长时意外覆盖内核本身。

▪ 大小: 我们为栈分配了**4KB**大小。这是一个常规且安全的尺寸, 对于我们这个函数调用层次很浅、没有大型局部变量的微型内核来说绰绰有余, 可以有效防止栈溢出 (Stack Overflow) 的发生。

• 问: 是否需要清零BSS段? 为什么?

◦ 答: 是, 必须清零。

◦ 原因有两点:

1. 遵循C语言标准: C语言规范明确要求, 任何未被程序员显式初始化的全局变量或静态变量, 其在程序开始运行时的值必须为0。

2. **可执行文件格式的优化**: 为了减小可执行文件（如ELF文件）的体积，编译器和链接器不会在文件中存储一大块0来代表BSS段。它们只记录BSS段的起始位置和大小。因此，当加载器（这里是QEMU）将程序加载到内存时，BSS段对应的内存区域里是随机的“脏数据”。我们必须在`_start`启动代码中，手动将这块内存区域逐字节清零，才能为后续的C代码提供一个符合预期的、干净的运行环境。**不清零BSS是导致C代码行为诡异和出现“幽灵bug”的常见原因。**

- **问：最简串口输出需要配置哪些寄存器？**

- **答：**

- **配置 (Configuration)**: 对于QEMU `virt`平台，实现最简单的输出**不需要配置任何寄存器**。我们可以直接使用其上电后的默认状态。
- **操作 (Operation)**: 在进行输出操作时，必须**使用**以下两个寄存器：
 1. **LSR (Line Status Register, 偏移地址 +5)**: 这是一个状态寄存器，我们需要**读取**它。特别是要检查它的第5位（从0开始计数），即**THRE (Transmitter Holding Register Empty)**位。只有当此位为1时，才表示串口的发送缓冲区已空，可以接收并发送下一个字符。
 2. **THR (Transmit Holding Register, 偏移地址 +0)**: 这是一个数据寄存器，我们需要向它**写入**数据。当确认**LSR**的**THRE**位为1后，把要发送的字符的ASCII码写入**THR**，硬件就会自动将该字符通过串口发送出去。

任务五

- **问：如何简化为最小实现？**

- **答：**

1. **忽略初始化**: xv6的 `uartinit` 会配置波特率、中断等。QEMU的UART默认配置已经可用，所以我们完全省略了初始化函数。
2. **只实现发送**: 我们只实现了 `uart_putc` 和 `uart_puts`，没有实现接收字符的功能。
3. **使用轮询**: 我们没有使用中断，而是通过一个死循环不断查询状态寄存器来判断是否可以发送。这是最简单但效率最低的方式，对于我们的目标已经足够。
4. **硬编码地址**: 我们直接将物理地址 `0x10000000` 硬编码在代码中，没有通过设备树或更复杂的硬件发现机制来获取。

- **问：为什么需要检查LSR的THRE位？**

- **答**: THRE (Transmitter Holding Register Empty) 位是UART硬件提供的一个状态标志。当这一位为1时，表示UART的发送缓冲已经空了，硬件已经准备好接收下一个要发送的字符。如果不检查这个位就直接向 **THR** 寄存器写入数据，可能会覆盖掉上一个尚未发送完毕的字符，导致数据丢失或输出乱码。**检查THRE位本质上是一种流控制，确保我们发送数据的速度不会超过硬件处理的速度。**

任务六

- **问：程序结束后应该做什么？死循环还是关机？**

- **答**: 程序设计为在内核主函数 `kmain` 打印完欢迎信息后，进入一个无限循环 `while (1)`

- **问：如何防止程序意外退出导致系统重启？**

。答：

1. **设计原则：** 操作系统内核不应有“退出”的概念，其主函数应永远不会返回。
2. **防御性编程：** 在汇编入口点 `entry.S` 中，`call kmain` 之后紧跟着一个死循环 (`_hang`)。这是一种“安全网”或“容错处理”，可以捕获任何从 `kmain` 返回的意外情况，从而防止系统崩溃。

三、测试验证部分

3.1 功能测试结果

- **测试用例：** 编译并运行内核。
- **预期结果：** QEMU启动后，在串行控制台（即当前终端）输出字符串 "Hello OS from RISC-V!" 并换行。
- **实际结果：** 实际输出与预期结果完全一致。
- **结论：** 内核的引导流程、BSS清零、栈指针设置、C函数调用以及UART轮询驱动功能均正常工作。

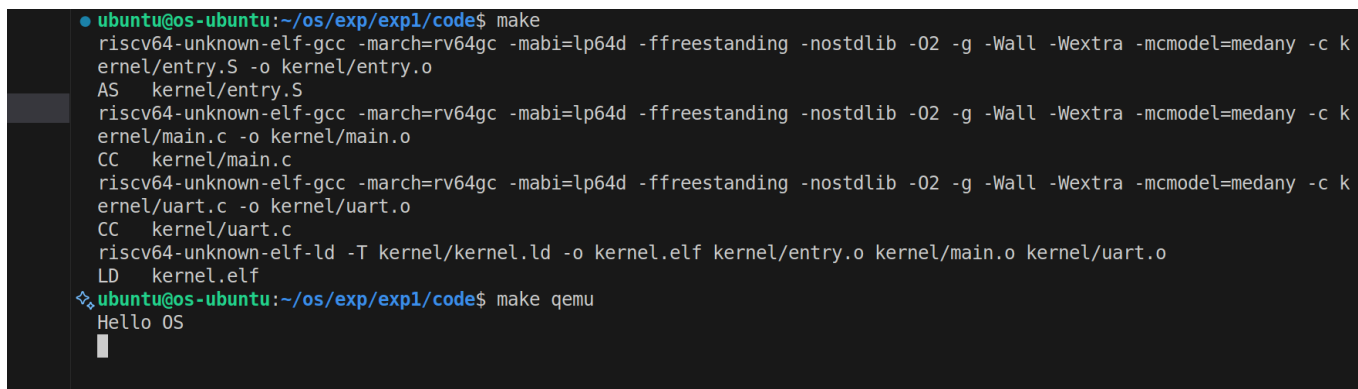
3.2 性能数据

本次实验为功能验证性质，不涉及性能测试。内核启动到打印信息完成的时间在毫秒级别，几乎是瞬时的。

3.3 异常测试

- **测试用例1：** 在 `main.c` 的 `kmain` 函数中移除 `while(1)` 循环，使其能够返回。
- **预期结果：** `kmain` 函数返回后，执行流会回到 `entry.S` 中的 `call kmain` 的下一条指令，即 `_hang: j _hang` 死循环。系统不会崩溃，但会“卡住”，并且不会有新的输出。
- **实际结果：** QEMU表现与预期一致，系统稳定地停留在死循环中，没有崩溃或重启。
- **结论：** `entry.S` 中的容错设计有效。

3.4 运行截图/录屏



```
ubuntu@os-ubuntu:~/os/exp/exp1/code$ make
riscv64-unknown-elf-gcc -march=rv64gc -mabi=lp64d -ffreestanding -nostdlib -O2 -g -Wall -Wextra -mcmodel=medany -c kernel/entry.S -o kernel/entry.o
AS   kernel/entry.S
riscv64-unknown-elf-gcc -march=rv64gc -mabi=lp64d -ffreestanding -nostdlib -O2 -g -Wall -Wextra -mcmodel=medany -c kernel/main.c -o kernel/main.o
CC   kernel/main.c
riscv64-unknown-elf-gcc -march=rv64gc -mabi=lp64d -ffreestanding -nostdlib -O2 -g -Wall -Wextra -mcmodel=medany -c kernel/uart.c -o kernel/uart.o
CC   kernel/uart.c
riscv64-unknown-elf-gcc -march=rv64gc -mabi=lp64d -ffreestanding -nostdlib -O2 -g -Wall -Wextra -mcmodel=medany -c kernel/uart.c -o kernel/uart.o
LD   kernel.elf
ubuntu@os-ubuntu:~/os/exp/exp1/code$ make qemu
Hello OS
```

(截图说明：上图显示了在终端中执行 `make qemu` 命令后的输出结果。首先是编译过程的日志，随后QEMU启动，并成功打印出 "Hello OS " 字符串，验证了实验的成功。)