练习 1：理解内核启动中的程序入口操作

内核入口与初始化分析

操作系统内核的生命周期始于计算机引导加载程序（Bootloader）将内核镜像加载到内存中，并将控制权移交给内核的入口点。在本实验中，这个入口点是 kern_entry，定义于汇编文件 kern/init/entry.S 中。

kern_entry 的主要职责是为 C 语言编写的内核初始化函数提供一个最小化且有效的执行环境。这通常包括建立内核栈、清空 BSS 段、以及设置必要的寄存器状态。完成这些底层准备工作后，程序将控制权转交给 C 语言函数 kern_init()，以开始内核的主要初始化过程。

1.1lasp,bootstacktop

指令操作：

la 是 RISC-V 架构中的伪指令，意为加载地址。

sp 表示栈指针寄存器，用于指向当前栈顶的位置。

bootstacktop 是在本文件末尾定义的标签，指向一块为内核引导阶段预留的栈空间的顶部地址。该栈空间通过以下语句分配：

.section.data

.alignPGSHIFT

.globalbootstack

bootstack:

.spaceKSTACKSIZE#分配 KSTACKSIZE 字节的内核栈空间

.globalbootstacktop

bootstacktop:

因此，lasp,bootstacktop 的作用是将符号 bootstacktop 的内存地址加载到寄存器 sp 中，即初始化内核栈指针，使其指向栈空间的顶部位置。

这条指令的主要目的是为即将执行的 C 语言函数建立一个合法的栈环境。

在操作系统内核中，栈空间用于：

函数调用机制：保存返回地址和传递函数参数；
局部变量存储：为函数体内的局部变量提供存储空间；
中断与异常处理：保存现场上下文。

由于 RISC-V 架构的栈是从高地址向低地址增长的，将 sp 指向栈空间的最高地址 bootstacktop，就为后续的函数调用提供了可用的栈空间。
这是从纯汇编世界进入 C 语言世界前的关键一步，为 C 函数的执行奠定了运行时环境的基础。

1.2tailkern_init

指令操作：
tail 也是一条伪指令，其语义为无返回跳转。
它通常会被汇编器翻译为一条无条件跳转指令，例如：
jkern_init
其中，kern_init 是定义在 kern/init/init.c 文件中的 C 语言函数，是整个内核的主入口。
因此，这条指令的实际行为是直接跳转到 kern_init 函数开始执行，不再返回 kern_entry。

执行到此处时，汇编部分的任务（如栈初始化）已经全部完成。
接下来，系统需要进入更高级的 C 语言初始化阶段，执行如：
清空.bss 段；
初始化页表与内存管理；
建立中断机制；
初始化控制台输出与设备驱动。

tailkern_init 的使用体现了一种“尾调用优化”的思想：
与 call 或 jal 指令不同，tail 不会保存返回地址，也不会在函数结束后返回。
这正符合当前情形——kern_entry 的使命已结束，控制权应永久交由 kern_init。

同时，这样的跳转方式避免了不必要的压栈与返回操作，使启动过程更加简洁、高效。

1.3 小结

指令	含义	主要作用	目的
lasp,bootstacktop	将内核栈顶地址加载到栈指针寄存器	初始化栈指针	为即将执行的 C 语言代码建立安全的函数调用环境
tailkern_init	无返回跳转到内核初始化函数	控制权转移	将控制权从汇编引导代码转交给 C 语言主函数，开始内核初始化过程

综上，kern_entry 作为操作系统内核的起点，其核心任务是建立运行环境与完成控制权移交。

当 tailkern_init 执行完毕，系统已从引导阶段进入真正的内核执行阶段，标志着操作系统启动流程的正式开始。

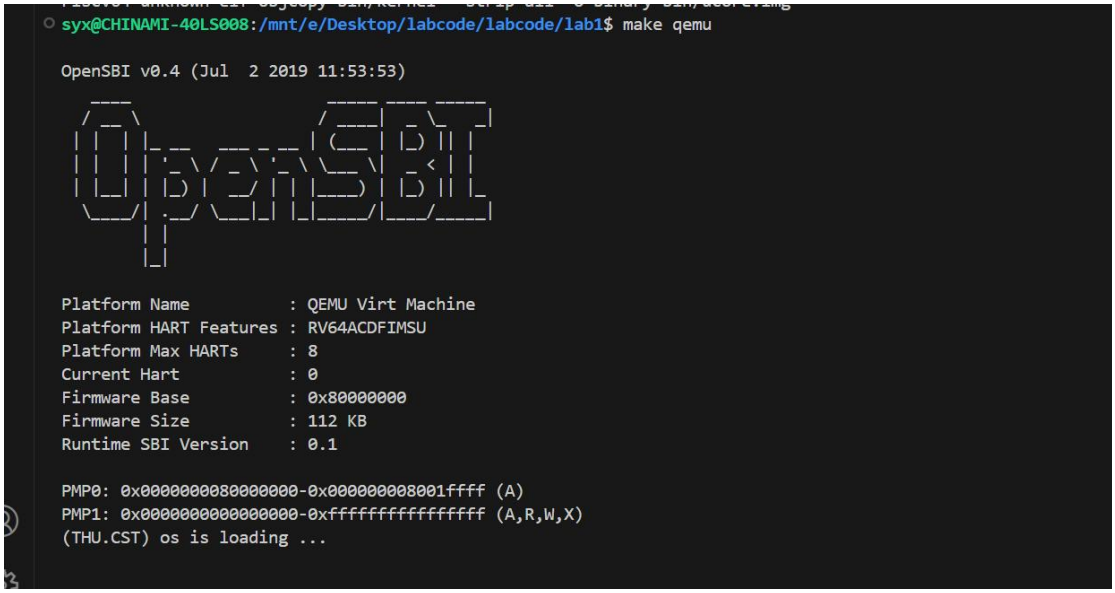
练习 2:使用 GDB 验证启动流程

本节记录了使用 GDB 调试工具跟踪 QEMU 模拟的 RISC-V 平台从加电到执行内核第一条指令的完整过程。

2.1 调试准备与过程

启动 QEMUforGDB 连接调试：首先，我们需要让 QEMU 在启动后暂停，并监听 GDB 的连接。这可以通过在 Makefile 中指定的 makedebug 命令实现。

而在另一个终端中，我们通过 Makefile 中指定的 makegdb 启动 RISC-V 版本的 GDB，并连接到 QEMU。



设置断点并跟踪：连接成功后，我们在内核的第一条指令地址 0x80200000 处设置一个断点，然后让程序继续执行。

(gdb)b*0x80200000

Breakpoint1at0x80200000

(gdb)c

Continuing.

程序会停在我们在 kern_entry 设置的断点处，证明内核代码即将开始执行。

2.2 观察与分析加电后的指令

1.初始状态：为了探究加电后真正执行的第一条指令，我们需要在 GDB 连接后，不设置断点，而是直接使用 si(StepInstruction)命令进行单步调试。

当 GDB 首次连接到 QEMU 时，可以看到终端中显示 0x0000000000001000in??()。这表明，RISC-V 硬件加电后，执行的第一条指令位于地址 0x1000。

2.0x1000 地址处的代码：这个地址位于 QEMU 的固化引导 ROM(BootROM)中。此处的代码是 QEMU 模拟的硬件固件，是真正的“第一阶段引导程序”。它的功能非常基础，主要完成以下工作：

设置设备树(DeviceTreeBlob,DTB)：它会找到 DTB 的位置，并将其地址放入 a1 寄存器中，以便后续的引导加载程序或内核能够了解当前的硬件布局。

跳转到下一阶段加载程序：它会跳转到一个固定的内存地址，通常是 0x80000000。这个地址上存放的是第二阶段的引导加载程序，在我们的实验环境中通常是 OpenSBI。

3.OpenSBI 的角色：代码执行权跳转到 0x80000000 后，OpenSBI 开始执行。OpenSBI 是一个开源项目，它为运行在 M-Mode（机器模式）下的操作系统内核提供了一个标准的接口，使得内核可以请求底层硬件的服务（如操作定时器、发送核间中断等）。OpenSBI 完成了更复杂的初始化，包括：

初始化 M-Mode 下的中断和异常处理。

设置 S-Mode 的运行环境。

最后，它会跳转到操作系统内核的入口点，也就是我们指定的 0x80200000。

通过 inforegisters 命令可以查看到程序计数器 pc 以及其他寄存器的值。

4. kern_entry的角色:代码执行权跳转到 0x80000000 后,kern_entry开始执行，kern_entry 是操作系统内核的入口。它的作用是设置内核栈指针，为C 语言函数调用分配栈空间，准备C 语言运行环境，然后按照RISC-V 的调用约定跳转到kern_init()函数。
在kern_init()函数处设置断点并使用 si 命令进行单步调试查看汇编代码。
同样在断点处通过 inforegisters 命令可以查看到寄存器的值。

5. kern_init 的角色:最后，kern_init()调用 cprintf()输出一行信息，表示内核启动成功。

2.3 结论

通过GDB 调试，我们可以清晰地梳理出从加电到内核启动的完整链条：
硬件加电：CPU 从复位向量地址 0x1000 开始执行。
BootROM(地址 0x1000)：执行 QEMU 的固件代码，进行最基础的硬件探测，并准备好 DTB 指针。
跳转到 OpenSBI：BootROM 将控制权转移到位于 0x80000000 的 OpenSBI。
OpenSBI 初始化：OpenSBI 为内核准备好监管者模式（S-Mode）的运行环境。
跳转到内核：OpenSBI 将控制权最终交给位于 0x80200000 的操作系统内核入口 kern_entry。

至此，硬件和固件的引导任务完成，操作系统的生命周期正式开始。

总结

本次实验通过代码分析和GDB 调试,完整地再现了操作系统内核从加电到执行第一行C 代码的完整启动链条。OpenSBI 的存在，定义了一套标准的二进制接口(SBI)。内核通过这个接口向M-Mode 的固件请求服务，而无需关心底层硬件的具体实现。同时GDB 调试揭示了从0x1000(BootROM)到0x80000000(OpenSBI)，再到0x80200000(Kernel)的跳转过程。这体现了操作系统设计中重要的分层思想，

这种特权级分离是现代操作系统安全模型的基石，它将内核与底层固件隔离，保护了硬件，也为内核提供了一个稳定的运行环境。