```
mkdir riscv-os && cd riscv-os
git init
mkdir -p kernel/{boot,mm,trap,proc,fs,net} include scripts
```

5. 验证环境

创建测试文件验证交叉编译：

```
echo 'int main(){ return 0; }' > test.c
riscv64-unknown-elf-gcc -c test.c -o test.o
file test.o  # 应显示RISC-V 64-bit
```

实验1：RISC-V引导与裸机启动

实验目标

通过参考xv6的启动机制，理解并实现最小操作系统的引导过程，最终在QEMU中输出"Hello OS"。

核心学习资料

xv6关键文件分析

- `kernel/entry.S` - 启动汇编代码
  - 重点理解：栈设置、BSS清零、跳转到C代码
  - 思考：为什么需要设置栈？栈应该多大？
- `kernel/kernel.ld` - 链接脚本
  - 重点理解：入口点设置、段的组织、符号定义
  - 思考：各个段的作用和排列顺序
- `kernel/uart.c` - 串口驱动
  - 重点理解：UART寄存器操作、字符输出实现
  - 思考：如何简化为最小实现？

RISC-V启动机制

- **特权级规范第3.1节**：机器模式启动状态
- **QEMU virt平台**：内存布局和设备地址

```
# 查看QEMU设备树
qemu-system-riscv64 -machine virt,dumpdtb=virt.dtb -nographic
dtc -I dtb -O dts virt.dtb | grep -A5 -B5 "uart\|memory"
```

任务列表

任务1：理解xv6启动流程

学习方法：

1. 阅读 `kernel/entry.S`，回答：
   - 为什么第一条指令是设置栈指针？
   - `la sp, stack0` 中的stack0在哪里定义？
   - 为什么要清零BSS段？
   - 如何从汇编跳转到C函数？
2. 分析 `kernel/kernel.ld`，思考：
   - `ENTRY(_entry)` 的作用是什么？
   - 为什么代码段要放在0x80000000？
   - `etext`、`edata`、`end` 符号有什么用途？

深入思考：

- xv6支持多核，你的单核系统可以如何简化？
- xv6的内存管理很复杂，最小系统需要哪些部分？

任务2：设计最小启动流程

设计要求：

1. 绘制你的启动流程图
2. 确定内存布局方案
3. 列出必需的硬件初始化步骤

关键问题：

- 栈应该放在内存的哪个位置？需要多大？
- 是否需要清零BSS段？为什么？
- 最简串口输出需要配置哪些寄存器？

任务3：实现启动汇编代码

参考xv6实现思路，但要大幅简化：

实现步骤：

1. 创建 `kernel/entry.S`
2. 设置入口点和栈指针
3. 清零BSS段（如果需要）
4. 跳转到C主函数

调试检查点：

```
# 在关键位置插入调试代码
_start:
    li t0, 0x10000000    # UART基地址
    li t1, 'S'           # 启动标记
```

```
    sb t1, 0(t0)          # 输出字符S表示启动

    # 设置栈后再输出一个字符验证
    la sp, stack_top
    li t1, 'P'            # 栈设置完成标记
    sb t1, 0(t0)
```

任务4：编写链接脚本

参考xv6的基本结构，简化复杂部分：

设计考虑：

1. 确定起始地址（通常是0x80000000）
2. 组织代码段、数据段、BSS段
3. 定义必要的符号供C代码使用

验证方法：

```
# 编译后检查内存布局
riscv64-unknown-elf-objdump -h kernel.elf
riscv64-unknown-elf-nm kernel.elf | grep -E "(start|end|text)"
```

任务5：实现串口驱动

参考xv6的uart.c，实现最小功能：

学习要点：

1. UART 16550的基本寄存器：
   - THR (Transmit Holding Register): 0x10000000
   - LSR (Line Status Register): 0x10000005
2. 输出一个字符的完整流程
3. 为什么需要检查LSR的THRE位？

实现策略：

```
// 先实现最基本的字符输出
void uart_putc(char c);

// 成功后实现字符串输出
void uart_puts(char *s);
```

调试建议：

- 先在汇编中直接写UART验证硬件工作
- 再在C函数中实现相同功能
- 最后实现完整的字符串输出

任务6：完成C主函数

设计考虑：

- 函数名可以不是main，与链接脚本保持一致
- 程序结束后应该做什么？死循环还是关机？
- 如何防止程序意外退出导致系统重启？

调试策略

分阶段调试法

1. **硬件验证阶段**：在汇编中直接写UART
2. **启动验证阶段**：验证能跳转到C函数
3. **功能验证阶段**：实现完整的Hello输出

常见问题诊断

问题：QEMU启动后无任何输出

- 检查链接脚本的起始地址
- 验证UART基地址是否正确
- 确认程序是否被正确加载

问题：输出乱码或不完整

- 检查UART初始化是否充分
- 验证字符发送间隔是否太快
- 确认字符串是否正确终止

GDB调试技巧

```
# 启动调试环境
make qemu-gdb  # 在一个终端
gdb-multiarch kernel/kernel.elf  # 在另一个终端
(gdb) target remote :1234
(gdb) b _start
(gdb) c
(gdb) layout asm
(gdb) si  # 单步执行汇编
```

思考题

1. **启动栈的设计**：
   - 你如何确定栈的大小？考虑哪些因素？
   - 如果栈太小会发生什么？如何检测栈溢出？
2. **BSS段清零**：
   - 写一个全局变量，不清零BSS会有什么现象？
   - 哪些情况下可以省略BSS清零？
3. **与xv6的对比**：

- 你的实现比xv6简化了哪些部分？
- 这些简化在什么情况下会成为问题？

4. **错误处理**：
   - 如果UART初始化失败，系统应该如何处理？
   - 如何设计一个最小的错误显示机制？

实验2：内核printf与清屏功能实现

实验目标

通过深入分析xv6的输出系统，理解格式化字符串处理原理，独立实现功能完整的内核printf和清屏功能。

核心学习资料

xv6输出系统架构分析

- `kernel/printf.c` - 格式化输出实现
  - 重点函数：`printf()`，`printint()`，`printptr()`
  - 学习要点：可变参数处理、数字转字符串算法
- `kernel/uart.c` - 硬件抽象层
  - 重点函数：`uartputc()`，`uartinit()`
  - 理解：设备驱动的抽象设计
- `kernel/console.c` - 控制台抽象层
  - 重点函数：`consputc()`，`consolewrite()`
  - 思考：为什么需要这个中间层？

相关技术规范

- **ANSI转义序列**：https://en.wikipedia.org/wiki/ANSI_escape_code
  - 重点：清屏、光标控制、颜色设置
- **C语言可变参数**：stdarg.h的使用方法
  - 关键宏：va_start, va_arg, va_end

任务列表

任务1：深入理解xv6输出架构

分析重点：

1. 研读 `printf.c` 中的核心函数：