

操作课程实验报告

Lab0.5：比麻雀更小的麻雀（最小可执行内核）

网络空间安全学院 信息安全-法学双学位班

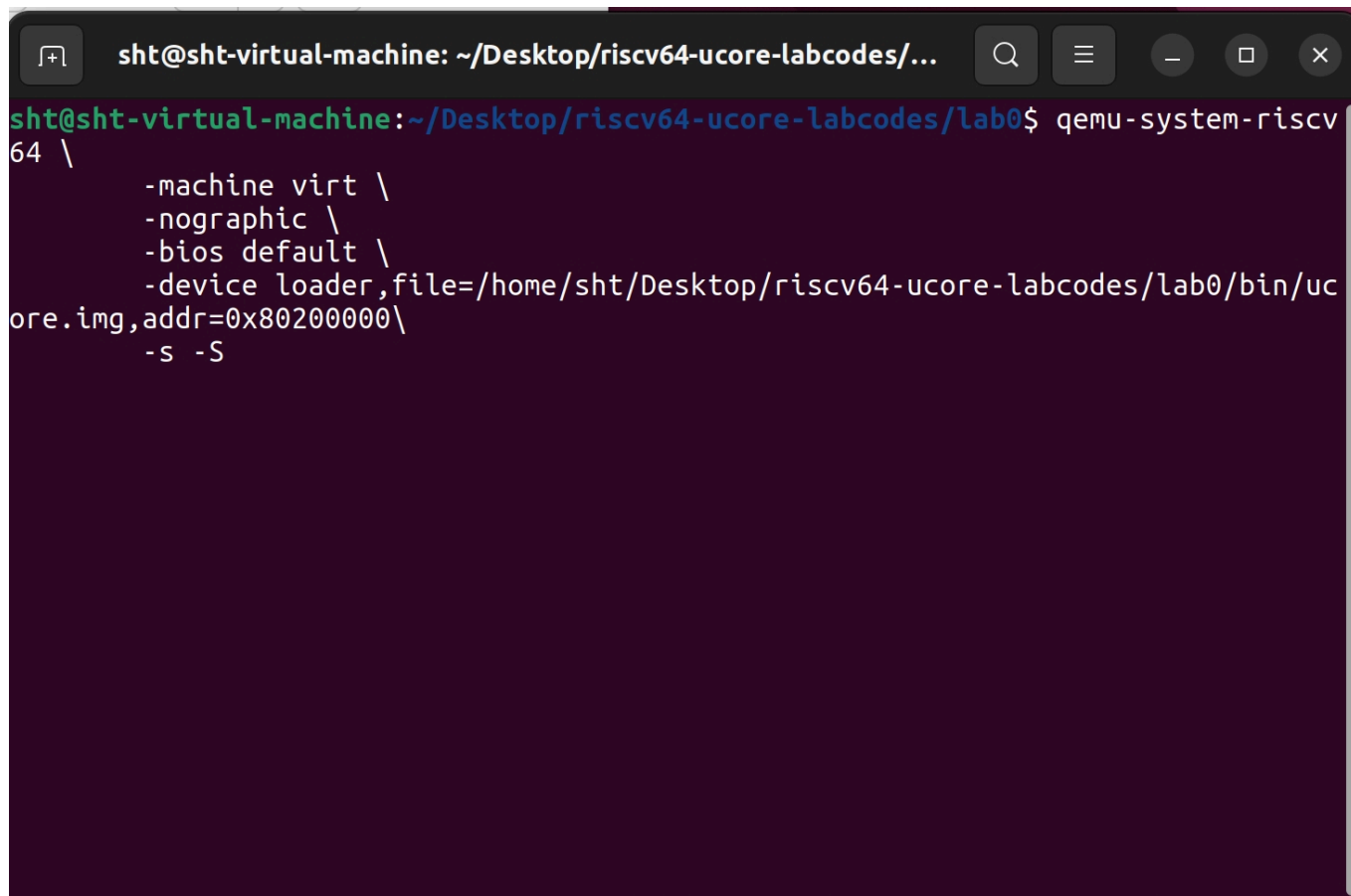
小组成员：苏浩天 苟辉铭 王崇宇

一、实验内容及目的

- 实验0.5主要讲解最小可执行内核和启动流程。我们的内核主要在 Qemu 模拟器上运行，它可以模拟一台 64 位 RISC-V 计算机。为了让我们的内核能够正确对接到 Qemu 模拟器上，需要了解 Qemu 模拟器的启动流程，还需要一些程序内存布局和编译流程（特别是链接）相关知识。
- 本章你将学到：
 - 使用 链接脚本 描述内存布局
 - 进行 交叉编译 生成可执行文件，进而生成内核镜像
 - 使用 OpenSBI 作为 bootloader 加载内核镜像，并使用 Qemu 进行模拟
 - 使用 OpenSBI 提供的服务，在屏幕上格式化打印字符串用于以后调试

二、熟悉相关命令和操作

- （一）首先我们使用命令来启动一个RISC-V架构的QEMU虚拟机：
 - qemu-system-riscv64: 启动QEMU模拟器，指定目标架构为RISC-V 64位。
 - machine virt: 指定使用“virt”虚拟机类型，这是一个通用的RISC-V虚拟机。
 - nographic: 禁用图形界面，所有输出都将在控制台中显示，适用于没有图形界面的环境。
 - bios default: 使用默认的BIOS文件，通常用于引导系统。
 - device loader,file=\$(UCOREIMG),addr=0x80200000: 加载一个指定的镜像文件（\$(UCOREIMG)是一个环境变量，表示镜像文件的路径），并将其加载到内存地址0x80200000。
 - s: 启用GDB调试，等待GDB连接。
 - S: 在启动时暂停虚拟机，这样可以在GDB连接后手动继续执行。
- 结合起来，这个命令通常用于开发和调试RISC-V系统，方便开发者进行深入的分析 and 调试。



```
sht@sht-virtual-machine: ~/Desktop/riscv64-ucore-labcodes/...
sht@sht-virtual-machine:~/Desktop/riscv64-ucore-labcodes/lab0$ qemu-system-riscv64 \
    -machine virt \
    -nographic \
    -bios default \
    -device loader,file=/home/sht/Desktop/riscv64-ucore-labcodes/lab0/bin/ucore.img,addr=0x80200000\
    -s -S
```

- (二) 随后我们启动RISC-V架构的GDB调试器：
 - riscv64-unknown-elf-gdb: 启动GDB，指定使用RISC-V 64位的交叉编译工具链。
 - ex 'file bin/kernel': 在GDB启动时执行该命令，将调试的可执行文件设置为bin/kernel。这个文件通常是编译后的内核映像。
 - ex 'set arch riscv:rv64': 在GDB启动时执行该命令，设置架构为RISC-V 64位，以便GDB能够正确处理与该架构相关的指令和数据。
 - ex 'target remote localhost:1234': 在GDB启动时执行该命令，连接到在本地主机（localhost）上运行的GDB远程目标，使用端口1234。通常，这个端口是QEMU的GDB调试监听端口，用于与虚拟机进行调试连接。
- 结合起来，这个命令是用来启动GDB并连接到正在运行的QEMU虚拟机，以便进行内核调试。

```
sht@sht-virtual-machine:~/Desktop/riscv64-ucore-labcodes/lab0$ riscv64-unknown-elf-gdb \
    -ex 'file bin/kernel' \
    -ex 'set arch riscv:rv64' \
    -ex 'target remote localhost:1234'
GNU gdb (SiFive GDB-Metal 10.1.0-2020.12.7) 10.1
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 "--host=x86_64-linux-gnu --target=riscv64-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://github.com/sifive/freedom-tools/issues>.
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 bin/kernel...
The target architecture is set to "riscv:rv64".
Remote debugging using localhost:1234
```

- （三）使用make gdb调试，输入指令x/10i \$pc查看即将执行的10条汇编指令，其中在地址为0x1010的指令处会跳转，故实际执行的为以下指令：

```
(gdb) x/10i $pc
=> 0x1000:      auipc      t0,0x0
    0x1004:      addi      a1,t0,32
    0x1008:      csrr      a0,mhartid
    0x100c:      ld        t0,24(t0)
    0x1010:      jr        t0
    0x1014:      unimp
    0x1016:      unimp
    0x1018:      unimp
    0x101a:      0x8000
    0x101c:      unimp
(gdb)
```

- （四）使用make gdb调试，输入指令x/10i 0x80000000查看显示 0x80000000 处的10条汇编指令。该地址处加载的是作为bootloader的OpenSBI.bin，该处的作用为加载操作系统内核并启动操作系统的执行：

```
(gdb) x/10i 0x80000000
0x80000000:      csrr      a6,mhartid
0x80000004:      bgtz      a6,0x80000108
0x80000008:      auipc      t0,0x0
0x8000000c:      addi      t0,t0,1032
0x80000010:      auipc      t1,0x0
0x80000014:      addi      t1,t1,-16
0x80000018:      sd        t1,0(t0)
0x8000001c:      auipc      t0,0x0
0x80000020:      addi      t0,t0,1020
0x80000024:      ld        t0,0(t0)
```

- （五）使用make gdb调试，输入指令x/10xw 0x80000000显示 0x80000000 处的10条数据，格式为16进制32bit，可以看到寄存器中存放的具体数据值：

```
(gdb) x/10xw 0x80000000
0x80000000: 0xf1402873 0x11004263 0x00000297 0x40828293
0x80000010: 0x00000317 0xff030313 0x0062b023 0x00000297
0x80000020: 0x3fc28293 0x0002b283
```

- （六）输入si单步执行，先执行三次，参照先前的整体代码，再执行一次，使用形如info r t0的指令查看涉及到的寄存器结果，由于在上一步的执行过程中t0 = [t0 + 24] = 0x80000000，值变化了，所以t0寄存器里的值也改变了：

```
(gdb) si
0x00000000000001004 in ?? ()
(gdb) si
0x00000000000001008 in ?? ()
(gdb) si
0x0000000000000100c in ?? ()
(gdb) info r t0
t0 0x1000 4096
(gdb) si
0x00000000000001010 in ?? ()
(gdb) info r t0
t0 0x80000000 2147483648
```

- （七）随后我们考虑熟悉断点相关的功能。首先需要先查看一下程序kern_entry的结构：

```
Open  [icon]
1 #include <mmu.h>
2 #include <memlayout.h>
3
4     .section .text,"ax",%progbits
5     .globl kern_entry
6 kern_entry:
7     la sp, bootstacktop
8
9     tail kern_init
10
11 .section .data
12     # .align 2^12
13     .align PGSIZE
14     .global bootstack
15 bootstack:
16     .space KSTACKSIZE
17     .global bootstacktop
18 bootstacktop:
```

- （八）设置点前，我们输入指令x/10i 0x80200000查看显示 0x80200000 处的10条汇编指令，方便与后续操作进行对应：

```
(gdb) x/10i 0x80200000
0x80200000 <kern_entry>: auipc    sp,0x3
0x80200004 <kern_entry+4>: mv      sp,sp
0x80200008 <kern_entry+8>: j      0x8020000a <kern_init>
0x8020000a <kern_init>: auipc    a0,0x3
0x8020000e <kern_init+4>: addi    a0,a0,-2
0x80200012 <kern_init+8>: auipc    a2,0x3
0x80200016 <kern_init+12>: addi    a2,a2,-10
0x8020001a <kern_init+16>: addi    sp,sp,-16
0x8020001c <kern_init+18>: li      a1,0
0x8020001e <kern_init+20>: sub     a2,a2,a0
```

- （九）接着输入指令 `break kern_entry`，在目标函数 `kern_entry` 的第一条指令处设置断点，输入 `continue` 执行直到断点：

```
(gdb) x/10i 0x80200000
0x80200000 <kern_entry>: auipc    sp,0x3
0x80200004 <kern_entry+4>: mv      sp,sp
0x80200008 <kern_entry+8>: j      0x8020000a <kern_init>
0x8020000a <kern_init>: auipc    a0,0x3
0x8020000e <kern_init+4>: addi    a0,a0,-2
0x80200012 <kern_init+8>: auipc    a2,0x3
0x80200016 <kern_init+12>: addi    a2,a2,-10
0x8020001a <kern_init+16>: addi    sp,sp,-16
0x8020001c <kern_init+18>: li      a1,0
0x8020001e <kern_init+20>: sub     a2,a2,a0
```

- 根据上图我们可以看到，地址 `0x80200000` 由 `kernel.ld` 中定义的 `BASE_ADDRESS`（加载地址）所决定，标签 `kern_entry` 是在 `kernel.ld` 中定义的 `ENTRY`（入口点）。
- `kernel_entry` 标志的汇编代码及解释如下：
 - `la sp, bootstacktop`：将 `bootstacktop` 的地址赋给 `sp`，作为栈
 - `tail kern_init`：尾调用，调用函数 `kern_init`
- 可以看到在 `kern_entry` 之后，紧接着就是 `kern_init`。
- 另一个命令行窗口 Debug 的输出说明 OpenSBI 此时已经启动。
- （十）随后对于程序 `kern_init` 设置断点。首先需要先查看一下程序 `kern_init` 的结构：

```

Open  ▾  [🔍]
1 #include <stdio.h>
2 #include <string.h>
3 #include <sbi.h>
4 int kern_init(void) __attribute__((noreturn));
5
6 int kern_init(void) {
7     extern char edata[], end[];
8     memset(edata, 0, end - edata);
9
10    const char *message = "(THU.CST) os is loading ...\n";
11    cprintf("%s\n\n", message);
12    while (1)
13        ;
14 }

```

- (十一) 接着输入指令 `break kern_init`, 输出如下:

```

0x80200000 <kern_entry>:  outpc    sp,0x3
0x80200004 <kern_entry+4>:  mv        sp,sp
0x80200008 <kern_entry+8>:  j         0x8020000a <kern_init>
0x8020000a <kern_init>:    auipc     a0,0x3
0x8020000e <kern_init+4>:    addi     a0,a0,-2
0x80200012 <kern_init+8>:    auipc     a2,0x3
0x80200016 <kern_init+12>:   addi     a2,a2,-10
0x8020001a <kern_init+16>:   addi     sp,sp,-16
0x8020001c <kern_init+18>:   li       a1,0
0x8020001e <kern_init+20>:   sub      a2,a2,a0
(gdb) continue
Continuing.

Breakpoint 1, kern_entry () at kern/init/entry.S:7
7      la sp, bootstacktop
(gdb) break kern_init
Breakpoint 2 at 0x8020000a: file kern/init/init.c, line 8.
(gdb) 

```

```

Open  ▾  [🔍]
1 #include <stdio.h>
2 #include <string.h>
3 #include <sbi.h>
4 int kern_init(void) __attribute__((noreturn));
5
6 int kern_init(void) {
7     extern char edata[], end[];
8     memset(edata, 0, end - edata);
9
10    const char *message = "(THU.CST) os is loading ...\n";
11    cprintf("%s\n\n", message);
12    while (1)
13        ;
14 }

```

- 这里就指向了之前显示为 `<kern_init>` 的地址 `0x8020000c`
 - `ra`: 返回地址
 - `sp`: 栈指针
 - `gp`: 全局指针
 - `tp`: 线程指针
- (十二) 随后我们输入 `continue`, 接着输入 `disassemble kern_init` 查看反汇编代码:


```
(gdb) disassemble kern_init
Dump of assembler code for function kern_init:
=> 0x000000008020000a <+0>:      auipc    a0,0x3
0x000000008020000e <+4>:      addi      a0,a0,-2 # 0x80203008
0x0000000080200012 <+8>:      auipc    a2,0x3
0x0000000080200016 <+12>:     addi      a2,a2,-10 # 0x80203008
0x000000008020001a <+16>:     addi      sp,sp,-16
0x000000008020001c <+18>:     li        a1,0
0x000000008020001e <+20>:     sub       a2,a2,a0
0x0000000080200020 <+22>:     sd        ra,8(sp)
0x0000000080200022 <+24>:     jal      ra,0x802004b6 <memset>
0x0000000080200026 <+28>:     auipc    a1,0x0
0x000000008020002a <+32>:     addi      a1,a1,1186 # 0x802004c8
0x000000008020002e <+36>:     auipc    a0,0x0
0x0000000080200032 <+40>:     addi      a0,a0,1210 # 0x802004e8
0x0000000080200036 <+44>:     jal      ra,0x80200056 <cprintf>
0x000000008020003a <+48>:     j        0x8020003a <kern_init+48>
End of assembler dump.
(gdb)
```

- 可以看到这个函数最后一个指令是 `j 0x8020003a <kern_init+48>`，也就是跳转到自己，所以代码会在这里一直循环下去。
- （十三）最后我们输入 `continue`，debug窗口出现以下输出：

```
0x000000008020001a <+16>:      addi      sp,sp,-16
0x000000008020001c <+18>:      li        a1,0
0x000000008020001e <+20>:      sub       a2,a2,a0
0x0000000080200020 <+22>:      sd        ra,8(sp)
0x0000000080200022 <+24>:      jal      ra,0x802004b6 <memset>
0x0000000080200026 <+28>:      auipc    a1,0x0
0x000000008020002a <+32>:      addi      a1,a1,1186 # 0x802004c8
0x000000008020002e <+36>:      auipc    a0,0x0
0x0000000080200032 <+40>:      addi      a0,a0,1210 # 0x802004e8
0x0000000080200036 <+44>:      jal      ra,0x80200056 <cprintf>
0x000000008020003a <+48>:      j        0x8020003a <kern_init+48>
End of assembler dump.
(gdb) continue
Continuing.
```

Platform Name	: QEMU Virt Machine
Platform HART Features	: RV64ACDFIMSU
Platform Max HARTs	: 8
Current Hart	: 0
Firmware Base	: 0x80000000
Firmware Size	: 112 KB
Runtime SBI Version	: 0.1

```
PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
```

- 至此我们完成了Lab0.5的内容，熟悉了相关gdb指令的调试方法以及调试的整体流程。

三、练习一问题回答

- 1、在RISC-V硬件加电后，执行的几条指令位于内存地址0x1000至0x1010。
- 2、这些指令的功能如下：

- `auipc t0,0x0` (地址：0x1000):

功能：将当前程序计数器（PC）加上偏移量0x0的值存入寄存器t0。由于PC在指令执行时指向0x1000，因此t0的值将为0x1000。

原因：该指令用于获取当前地址，以便后续操作使用。

- `addi a1,t0,32` (地址：0x1004):

功能：将t0的值加上32，并将结果存入寄存器a1。计算后，a1的值为0x1020。

原因：为后续操作准备参数。

- csrr a0,mhartid (地址：0x1008):

功能：将当前硬件线程的ID (mhartid) 加载到寄存器a0。在本例中，a0将被设置为0，表示当前线程的ID。

原因：用于标识当前执行的硬件线程。

- ld t0,24(t0) (地址：0x100c):

功能：从t0加上偏移量24的地址（即 $0x1000 + 24 = 0x10018$ ）中加载一个字（通常是64位）到t0。这里假设地址0x10018处的内容为0x80000000。

原因：加载下一个要跳转到的地址。

- jr t0 (地址：0x1010):

功能：跳转到t0指向的地址0x80000000。

原因：执行到此，程序将跳转到下一阶段的代码或应用程序，通常是内核或应用的入口点。

三、本实验重要知识点

相互关联的

- **最小可执行内核的设计与实现**：主要涉及构建一个极简的操作系统内核，能够独立启动并运行基本任务。这包括引导加载程序的工作原理，以及内核如何初始化和管理的硬件资源并且命令如何启动虚拟机。这种设计帮助理解内核的基本职责和与硬件的交互方式，通过实践掌握内核启动过程中的关键步骤。但是与os课程中不同的是，实验中的内核设计较为简化，帮助理解基本概念；而完整操作系统更复杂，功能更全面。
- **操作系统的加载**：本实验中，主要涉及了OpenSBI怎样知道把操作系统加载到内存的什么位置，实际上，系统使用了一个elf文件和bin文件，得到内存布局合适的elf文件，然后把它转化成bin文件（这一步通过objcopy实现），然后加载到QEMU里运行。
- **调试程序**：在本实验中，调试相关的知识点包括使用GDB进行内核调试的具体步骤和技巧。首先，通过命令启动RISC-V架构的QEMU虚拟机，并设置GDB连接，以便进行调试。使用make gdb命令可以启动GDB，并连接到QEMU虚拟机，从而调试内核。通过调试命令的学习，这些命令帮助开发者深入了解程序的执行过程，尤其是在内核引导和初始化阶段。通过设置断点和单步执行，可以逐步观察程序的状态变化，便于定位问题和优化代码。
- **链接器**：链接器的作用是把输入文件(往往是.o文件)链接成输出文件(往往是elf文件)。一般来说，输入文件和输出文件都有很多section，链接脚本(linker script)的作用，就是描述怎样把输入文件的section映射到输出文件的section，同时规定这些section的内存布局。
- **调用接口**：实验中的接口主要是指OpenSBI的接口一层层封装到格式化输入输出函数，而在os课程中的接口是指用户态与内核态的转换。

未涉及的

- **进程管理**：涉及多任务的创建、调度和切换。需要实现进程调度算法，理解进程生命周期的各个阶段，以及上下文切换的机制。这有助于掌握进程间的资源共享与隔离，确保多个进程能够有效地同时运行，进一步理解操作系统如何在资源有限的情况下优化性能。

- **内存管理**：系统内存管理涉及通过分页或分段技术来组织和分配内存，以实现高效的资源利用和保护。操作系统通过内存管理单元（MMU）将虚拟地址转换为物理地址，从而实现进程隔离和内存共享。内存管理还包括内存分配策略和回收机制，确保不同进程能够安全、有效地使用内存资源。这种管理方式提高了系统的稳定性和性能。