

# 操作系统实验报告

信息安全

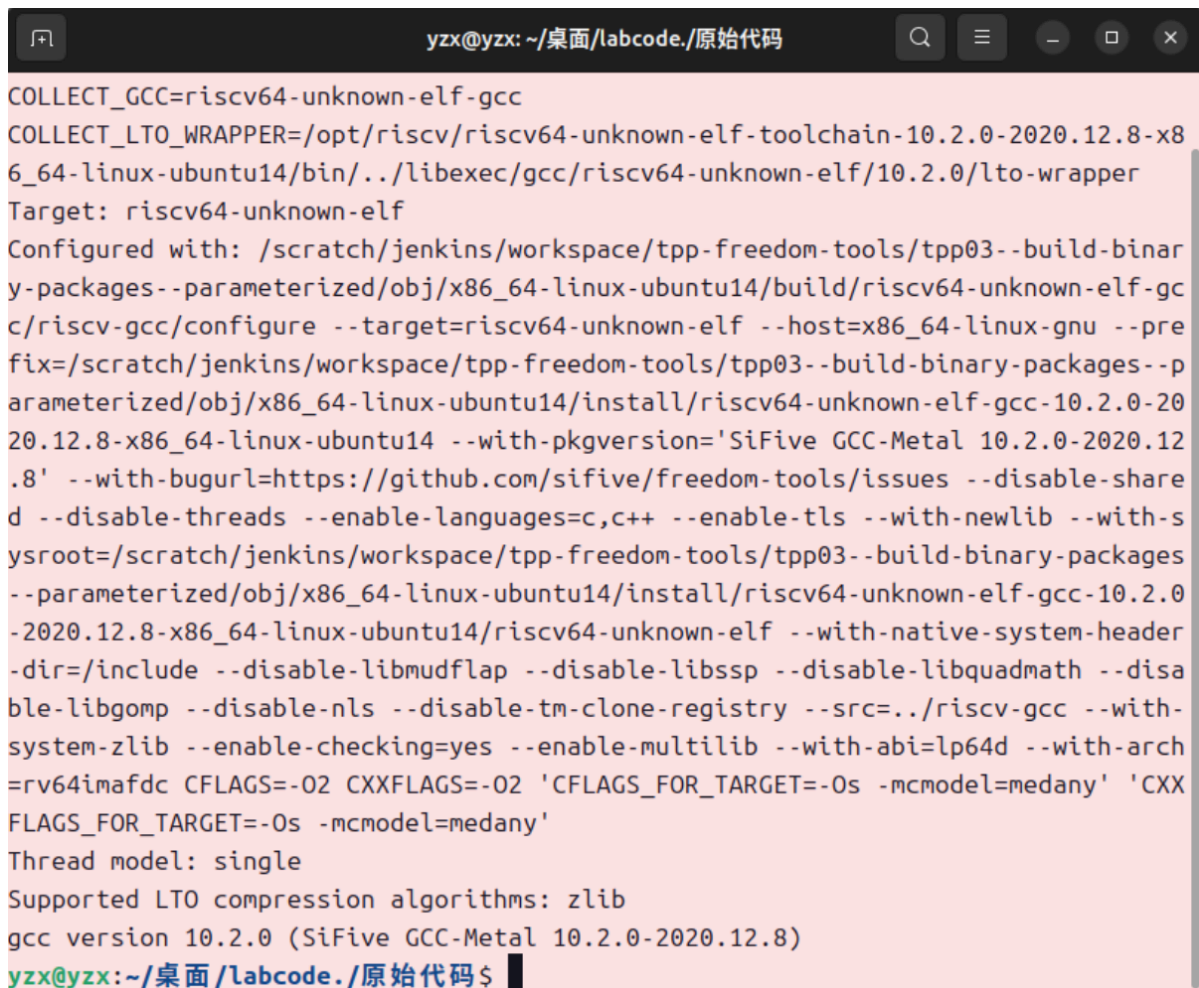
2313781 李胜林 2312796 张肇秋 2312323 杨中秀

## 一、配置实验环境

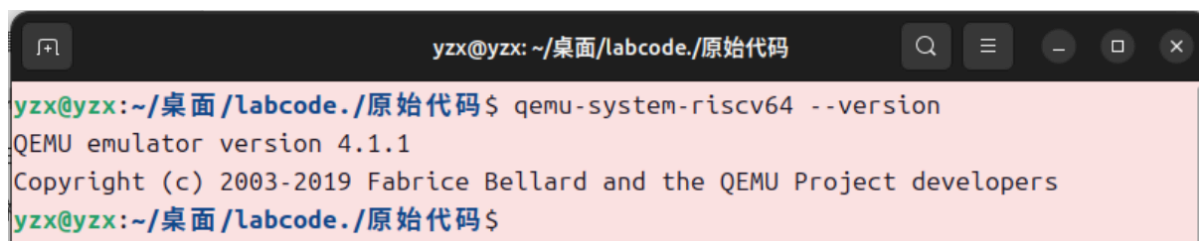
这个部分比较繁琐，但步骤相对简单，根据实验文档，遇到不会的就去查阅资料就可以了，因此在这里便不再赘述。在配置完环境之后，我们输入

```
1 riscv64-unknown-elf-gcc -v
2 qemu-system-riscv64 --version
```

两个指令即可查看gcc和qemu的版本，得到的版本信息如下：

A terminal window titled 'yzx@yzx: ~/桌面/labcode./原始代码' showing the output of the 'riscv64-unknown-elf-gcc -v' command. The output includes the compiler path, target architecture, and a detailed list of configuration options used to build the compiler, such as '--target=riscv64-unknown-elf', '--host=x86\_64-linux-gnu', and various optimization and feature flags. It also shows the thread model as 'single' and supported LTO compression algorithms as 'zlib'.

```
COLLECT_GCC=riscv64-unknown-elf-gcc
COLLECT_LTO_WRAPPER=/opt/riscv/riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86_64-linux-ubuntu14/bin/../libexec/gcc/riscv64-unknown-elf/10.2.0/lto-wrapper
Target: riscv64-unknown-elf
Configured with: /scratch/jenkins/workspace/tpp-freedom-tools/tpp03--build-binary-packages--parameterized/obj/x86_64-linux-ubuntu14/build/riscv64-unknown-elf-gcc/riscv-gcc/configure --target=riscv64-unknown-elf --host=x86_64-linux-gnu --prefix=/scratch/jenkins/workspace/tpp-freedom-tools/tpp03--build-binary-packages--parameterized/obj/x86_64-linux-ubuntu14/install/riscv64-unknown-elf-gcc-10.2.0-2020.12.8-x86_64-linux-ubuntu14 --with-pkgversion='SiFive GCC-Metal 10.2.0-2020.12.8' --with-bugurl=https://github.com/sifive/freedom-tools/issues --disable-shared --disable-threads --enable-languages=c,c++ --enable-tls --with-newlib --with-sysroot=/scratch/jenkins/workspace/tpp-freedom-tools/tpp03--build-binary-packages--parameterized/obj/x86_64-linux-ubuntu14/install/riscv64-unknown-elf-gcc-10.2.0-2020.12.8-x86_64-linux-ubuntu14/riscv64-unknown-elf --with-native-system-header-dir=/include --disable-libmudflap --disable-libssp --disable-libquadmath --disable-libgomp --disable-nls --disable-tm-clone-registry --src=../riscv-gcc --with-system-zlib --enable-checking=yes --enable-multilib --with-abi=lp64d --with-arch=rv64imafdc CFLAGS=-O2 CXXFLAGS=-O2 'CFLAGS_FOR_TARGET=-Os -mcmodel=medany' 'CXXFLAGS_FOR_TARGET=-Os -mcmodel=medany'
Thread model: single
Supported LTO compression algorithms: zlib
gcc version 10.2.0 (SiFive GCC-Metal 10.2.0-2020.12.8)
yzx@yzx:~/桌面/labcode./原始代码$
```

A terminal window titled 'yzx@yzx: ~/桌面/labcode./原始代码' showing the output of the 'qemu-system-riscv64 --version' command. The output displays the QEMU emulator version as 4.1.1, along with the copyright notice for 2003-2019 by Fabrice Bellard and the QEMU Project developers.

```
yzx@yzx:~/桌面/labcode./原始代码$ qemu-system-riscv64 --version
QEMU emulator version 4.1.1
Copyright (c) 2003-2019 Fabrice Bellard and the QEMU Project developers
yzx@yzx:~/桌面/labcode./原始代码$
```

## （一）运行ucore内核

```

yzx@yzx:~/桌面/labcode./原始代码/lab1$ make qemu

OpenSBI v0.4 (Jul  2 2019 11:53:53)

      /  _  \      /  _  \  _  \  _  \
| | | | | _ _ _ _ _ | ( _ _ | | _ | | | | |
| | | | | ' _ \ / _ \ ' _ \ \ _ \ | _ < | |
| | _ | | | _ ) | _ / | | | _ ) | | _ | |
 \ _ _ / | . _ / \ _ _ | | | _ _ / | _ _ / _ _ |
      | |
      | _ |

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: 0x0000000008000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffff (A,R,W,X)
(THU.CST) os is loading ...

```

为更细化地了解ucore内核启动过程中执行的操作，我们接下来使用GDB调试，一步步监控过程中寄存器的变化与执行的语句的情况。

在使用gdb工具前，我们需要安装一个tmux工具用于查看两个界面，输入 `sudo qpt install tmux` 即可完成安装。

随后我们开始进行调试，首先在左侧界面输入 `make debug`，右侧界面输入 `make gdb`，即可进入以下界面：

```
yzx@yzx: ~/桌面/labcode./原始代码/lab1$ make debug
yzx@yzx:~/桌面/labcode./原始代码/lab1$ make gdb
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
0x0000000000001000 in ?? ()
(gdb) █
```

## 查看硬件加电最初指令的位置

这说明我们的调试工作开始了，随后我们输入 `target remote:1234` 和 `info registers pc`，从而建立GDB与Qemu调试器的链接并显示pc寄存器的值，然后我们先输入 `x/10i $pc` 以查看从现在开始往后的10条汇编指令，这些指令便是硬件加电后最初执行的几条指令的位置，其结果如下图所示：

其中比较重要的就是前5条汇编指令

```
1  auipc    tp,0x0
2  addi     a1,to,32
3  csrr     a0,mhartid
4  ld       to,24(t0)
5  jr       t0
```

随后我们会进行说明，现在我们着重展现调试过程。

## 查看一些其他的信息

### 查看所有寄存器的信息

随后我们输入 `b* kern_entry` 添加断点，随后输入 `c` 执行到断点的位置，我们发现debug界面也发生了变化，如图所示：



```
15 | 0x000000008020003a <+48>: j      0x8020003a <kern_init+48> ; 死循环，内核初始
    | 化完成后挂起
```

## 单步调试并查看函数调用栈

我们在执行到断点的位置后，便可以进行单步调试 `si` 并查看函数调用栈中的信息 `bt` 了，如下所示：

```
(gdb) si
0x0000000080200004 in kern_entry () at kern/init/entry.S:7
7      la sp, bootstacktop
(gdb) bt
#0  0x0000000080200004 in kern_entry () at kern/init/entry.S:7
(gdb) si
9      tail kern_init
(gdb) bt
#0  kern_entry () at kern/init/entry.S:9
(gdb) si
kern_init () at kern/init/init.c:8
8      memset(edata, 0, end - edata);
(gdb) bt
#0  kern_init () at kern/init/init.c:8
#1  0x0000000080000a02 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```

我们可以看到，随着代码的执行，函数调用栈中的信息从无到有有一个 `kern_init()` 再到有另一个仅有地址的位置函数。

## 切换函数调用栈

在上一步单步执行到一定程度之后，我们变可以通过 `frame 0`、`frame 1` 等切换函数调用栈，如下图所示：

```
(gdb) frame 1
#1  0x0000000080000a02 in ?? ()
(gdb) frame 0
#0  0x0000000080200016 in kern_init () at kern/init/init.c:8
8      memset(edata, 0, end - edata);
(gdb) frame 2
No frame at level 2.
```

我们可以看到仅能切换两个函数调用栈（因为仅有调用了两个函数），与我们之前单步执行时相同观察到的函数调用栈中的信息相同。

## 查看栈中的内容

我们输入 `x/10x $sp` 可以以十六进制显示栈指针后的10个字，如下图所示：

```
(gdb) x/10x $sp
0x80203000 <SBI_CONSOLE_PUTCHAR>:      0x00000001      0x00000000
0x00000000      0x00000000
0x80203010:      0x00000000      0x00000000      0x00000000      0x0
00000000
0x80203020:      0x00000000      0x00000000
```

## 内核启动

我们继续输入 `c` 即可使程序绕过断点继续执行，在这个时候便会输出 `(THU.CST) os is loading`。随后便与实验指导文档中所说的一样，成功启动了内核

### (三) 练习解答

#### 练习1：理解内核启动中的程序入口操作

阅读 `kern/init/entry.S` 内容代码，结合操作系统内核启动流程，说明指令 `la sp, bootstacktop` 完成了什么操作，目的是什么？`tail kern_init` 完成了什么操作，目的是什么？

我认为在系统内核启动的过程中，`la sp, bootstacktop` 是将 `bootstacktop` 的地址加载到栈指针寄存器 `sp`，而 `tail kern_init` 是跳转到 `kern_init` 函数，这里的 `tail` 与普通的 `call` 不同，它不会保存返回地址和建立栈帧。这段代码的目的是设置好内核栈指针，保存函数调用的返回地址、局部变量和函数参数等状态信息，在内核正式执行前准备好执行环境，以确保后续的 `kern_init` 等 C 函数能够正确运行，然后直接跳转到内核初始化函数，实现了控制权从底层汇编代码向内核初始化的转移。

#### 练习2：使用GDB验证启动流程

RISC-V 硬件加电后最初执行的几条指令位于什么地址？它们主要完成了哪些功能？

RISC-V 硬件加电后第一阶段执行的指令从 `0x1000` 处开始，如下图所示，完成了硬件上电、初始化和固件启动的工作：

```
(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)
```

其主要代码为：

1	<code>auipc</code>	<code>tp,0x0</code>	#将当前PC值的高20位与立即数0组合，存入tp寄存器
2	<code>addi</code>	<code>a1,to,32</code>	#将tp的值加上32存入a1寄存器
3	<code>csrr</code>	<code>a0.mhartid</code>	#读取mhartid CSR（控制和状态寄存器），获取当前硬件线程ID
4	<code>ld</code>	<code>to,24(t0)</code>	#从tp+24的内存地址加载64位值到t0寄存器
5	<code>jr</code>	<code>t0</code>	#跳转到t0寄存器指定的地址

这几条指令完成了RISC-V系统加电后的硬件初始化，并跳转到主引导程序继续执行。



RISC-V硬件加电后的第二阶段执行的指令从0x80000000处开始，其部分代码如下图所示，完成了OpenSBI初始化与内核加载等工作：

```
(gdb) x/10i 0x80000000
0x80000000: add    s0,a0,zero
0x80000004: add    s1,a1,zero
0x80000008: add    s2,a2,zero
0x8000000c: jal    ra,0x80000548
0x80000010: add    a6,a0,zero
0x80000014: add    a0,s0,zero
0x80000018: add    a1,s1,zero
0x8000001c: add    a2,s2,zero
0x80000020: li     a7,-1
0x80000022: beq    a6,a7,0x8000002a
```

该阶段主要完成了OpenSBI在RISC-V机器模式下的运行、初始化处理器运行环境和系统控制状态寄存器（同时保存a0、a1、a2等参数随后进行函数调用），设置中断处理、内存保护等底层机制，准备加载操作系统内核的环境，加载内核镜像至0x80200000等工作。

RISC-V硬件加电后的第三阶段执行的指令从0x80200000处开始，其代码如下图所示，完成了内核启动执行等工作：

```
(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
(gdb) b* kern_init
Breakpoint 2 at 0x8020000a: file kern/init/init.c, line 8.
(gdb) c
Continuing.

Breakpoint 2, kern_init () at kern/init/init.c:8
8          memset(edata, 0, end - edata);
```

其主要代码为：

1	auipc	sp,0x3	#计算栈顶地址，调整栈指针
2	mv	sp,sp	#或许是手动加上的阻塞以确保流水线效率
3	j	0x8020000a <kern_init>	#跳转到kern_init函数，进行c代码的内存初始化
4	auipc	a0,0x3	#传参，a0<=-PC+0x3000
5	addi	a0,a0,-2	#得到edata的地址
6	auipc	a2,0x3	#传参，a2<=-PC+0x3000
7	addi	a2,a2,0d-10	#end的地址（0x80203008）
8	addi	sp,sp,-16	#开辟栈帧
9	li	a1,0	#设置参数初始值
10	sub	a2,a2,a0	

上述代码主要实现了从 kern\_entry 开始执行内核代码，设置内核栈指针，为C语言函数调用准备栈空间，跳转到 kern\_init 函数进入内核主初始化流程，并对内核数据段进行必要的初始化。

以上三级启动的流程实现了操作系统从最底层的硬件初始化到高层操作系统内核的平滑过渡，每一级都为下一级准备了必要的执行环境。