

Lab3 最小化内核

一、实验背景

ucore的背景知识

2006年, MIT的Frans Kaashoek等人参考PDP-11上的UNIX Version 6写了一个可在x86指令集架构上运行的操作系统xv6 (基于MIT License)。

2010年, 清华大学操作系统教学团队参考MIT的教学操作系统xv6, 开发了在x86指令集架构上运行的操作系统ucore, 多年来作为操作系统课程的实验框架使用。

ucore麻雀虽小, 五脏俱全。在不超过5k的代码量中包含虚拟内存管理、进程管理、处理器调度、同步互斥、进程间通信、文件系统等主要内核功能, 充分体现了“小而全”的指导思想。

ucore的运行环境可以是真实的计算机(包括小型智能设备)。一开始ucore是运行在x86指令集架构上的, 到了如今, x86指令集架构的问题渐渐开始暴露出来。虽然在PC平台上占据绝对主流, 但出于兼容性考虑x86架构仍然保留了许多历史包袱, 用于教学的时候总有些累赘。另一方面, 为了更好的和目前5G、物联网技术的发展衔接, 将ucore移植到RISC-V架构势在必行。

相对于上百万行的现代操作系统(linux, windows), 几千行代码的ucore似乎很小。但是我们也一眼看不过来几千行代码, 所以我们对ucore进行简化, 首先构建出一个ucore的“骨架”, 然后一步一步完善功能, 最终实现一个完整的ucore。本次实验是整个操作系统实验的基础, 我们构建一个最小化内核, 这就是ucore的骨架, 它能够进行格式化输出, 然后进入死循环。

RISC-V背景知识

RISC发明者是美国加州大学伯克利分校教师David Patterson, RISC-V (拼做risk-five) 是第五代精简指令集, 也是由David Patterson指导的项目。2010年伯克利大学并行计算实验室(Par Lab) 的1位教授和2个研究生想要做一个项目, 需要选一种计算机架构来做。当时面临的的是选择X86、ARM, 还是其他指令集, 不管选择哪个都或多或少有些问题, 比如授权费价格高昂, 不能开源, 不能扩展更改等等。所以他们在2010年5月开始规划自己做一个新的、开源的指令集, 就是RISC-V。

RISC-V特点

1. 模块化的指令子集。RISC-V的指令集使用模块化的方式进行组织, 每一个模块使用一个英文字母来表示。RISC-V最基本也是唯一强制要求实现的指令集部分是由I字母表示的基本整数指令子集, 使用该整数指令子集, 便能够实现完整的软件编译器。其他的指令子集部分均为可选的模块, 具有代表性的模块包括M/A/F/D/C。
2. 规整的指令编码。RISC-V的指令集编码非常的规整, 指令所需的通用寄存器的索引(Index) 都被放在固定的位置。因此指令译码器(Instruction Decoder) 可以非常便捷的译码出寄存器索引然后读取通用寄存器组(Register File, Regfile)。
3. 优雅的压缩指令子集。基本的RISC-V基本整数指令子集(字母I表示) 规定的指令长度均为等长的32位, 这种等长指令定义使得仅支持整数指令子集的基本RISC-V CPU非常容易设计。但是等长的32位编码指令也会造成代码体积(Code Size) 相对较大的问题。为了满足某些对于代码体积要求较高的场景(譬如嵌入式领域), RISC-V定义了一种可选的压缩(Compressed) 指令子集, 由字母C表示, 也可以由RVC表示。RISC-V具有后发优势, 从一开始便规划了压缩指令, 预留了足够的编码空间, 16位长指令与普通的32位长指令可以无缝自由地交织在一起, 处理器也没有定义额外的状态。
4. 特权模式。RISC-V架构定义了三种工作模式, 又称特权模式(Privileged Mode): Machine Mode: 机器模式, 简称M Mode。Supervisor Mode: 监督模式, 简称S Mode。User Mode:

用户模式，简称U Mode。RISC-V架构定义M Mode为必选模式，另外两种为可选模式。通过不同的模式组合可以实现不同的系统。

5. 自定义指令扩展。除了上述阐述的模块化指令子集的可扩展、可选择，RISC-V架构还有一个非常重要的特性，那就是支持第三方的扩展。用户可以扩展自己的指令子集，RISC-V预留了大量的指令编码空间用于用户的自定义扩展，同时，还定义了四条Custom指令可供用户直接使用，每条Custom指令都有几个比特位的子编码空间预留，因此，用户可以直接使用四条Custom指令扩展出几十条自定义的指令。

RISCV手册中文版：见sakai资源

特权级架构简介：见sakai资源

RISCV汇编手册：[riscv-asm-manual/riscv-asm.md at master · riscv-non-isa/riscv-asm-manual \(github.com\)](https://github.com/riscv-asm-manual/riscv-asm.md)

二、实验目的

1. 源码是如何被编译成可执行文件的。
2. 编译成可执行文件后，计算机如何加载操作系统。
3. 加载以后，该从哪里去运行操作系统。
4. 操作系统是怎样输出信息的。

三、实验项目整体框架概述

. //lab0 目录

```
├── kern //kernel 文件夹，需要重点了解
│   ├── driver
│   │   ├── console.c //对opensbi调用的输入输出字符进行进一步封装
│   │   └── console.h
│   ├── init
│   │   ├── entry.S //内核入口点，需要重点了解
│   │   └── init.c //C语言层级上的入口点，需要重点了解
│   ├── libs
│   │   ├── stdio.c //调用console.c,实现简单的stdio库
│   │   └── mm //目前只定义了一些常量
│   │       ├── memlayout.h
│   │       └── mmu.h
├── libs //libraries 先了解大致功能，根据需求再去了解细节
│   ├── defs.h
│   ├── error.h
│   ├── printfmt.c //实现复杂的格式化输入输出函数
│   ├── readline.c
│   ├── riscv.h //定义了若干和riscv架构相关的宏，尤其是将一些内联汇编的代码封装成宏
│   ├── sbi.c //调用opensbi接口
│   ├── sbi.h
│   ├── stdarg.h
│   ├── stdio.h
│   ├── string.c
│   └── string.h
├── Makefile // Make 编译脚本，先观察make的运行结果，然后去了解具体含义。
└── tools
```

- └─ `function.mk` //Makefile的补充，被Makefile调用
- └─ `kernel.ld` //链接脚本

四、实验内容

1. 下载实验源码，切换到lab0目录下
2. 执行 `make` 指令，生成img镜像
3. 执行 `make qemu`，用qemu模拟器启动我们的最小化内核。

五、实验流程及相关知识点

第一步. 在blackboard上下载实验源码：

下载lab3代码并切换到代码目录

第二步. 使用 `make` 命令生成我们的模拟硬盘

从计算机组成开始：

计算机的组成：CPU，内存，硬盘，输入输出设备，总线

QEMU会帮助我们模拟一块riscv64的CPU，一块物理内存，还会借助你的电脑的键盘和显示屏来模拟命令行的输入和输出。虽然QEMU不会真正模拟一堆线缆，但是总线的通信功能也在QEMU内部实现了。所以还缺一块硬盘。

我们在lab0下面执行make命令，在bin目录下会生成ucore.bin文件，这是我们的硬盘。由于整个makefile文件比较复杂，我们不妨从makefile的运行结果开始，倒过来理解make之后发生了什么事情。

我们可以在make后面加一个V=参数，这里make在执行时，会把具体执行的命令打印出来，其他方面的功能和make一样。

`make` 和 `make V=` 结果比较

```
oslab@oslab-virtual-machine:~/Desktop/lab0$ make
+ cc kern/init/entry.S
+ cc kern/init/init.c
+ cc kern/libs/stdio.c
+ cc kern/driver/console.c
+ cc libs/string.c
+ cc libs/printfmt.c
+ cc libs/readline.c
+ cc libs/sbi.c
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.bin
```

```
oslab@oslab-virtual-machine:~/Desktop/lab0$ make V=
+ cc kern/init/entry.S
riscv64-unknown-elf-gcc -Ikern/init/ -mcmodel=medany -std=gnu99 -Wno-unused -Werror -fno-builtin -Wall -O2 -nostdinc -fno-stack-protect
-Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/libs/ -Ikern/mm/ -Ikern/arch/ -c kern/init/entry.S -o obj/kern/init/entry.o
+ cc kern/init/init.c
riscv64-unknown-elf-gcc -Ikern/init/ -mcmodel=medany -std=gnu99 -Wno-unused -Werror -fno-builtin -Wall -O2 -nostdinc -fno-stack-protect
-Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/libs/ -Ikern/mm/ -Ikern/arch/ -c kern/init/init.c -o obj/kern/init/init.o
+ cc kern/libs/stdio.c
riscv64-unknown-elf-gcc -Ikern/libs/ -mcmodel=medany -std=gnu99 -Wno-unused -Werror -fno-builtin -Wall -O2 -nostdinc -fno-stack-protect
-Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/libs/ -Ikern/mm/ -Ikern/arch/ -c kern/libs/stdio.c -o obj/kern/libs/stdio.o
+ cc kern/driver/console.c
riscv64-unknown-elf-gcc -Ikern/driver/ -mcmodel=medany -std=gnu99 -Wno-unused -Werror -fno-builtin -Wall -O2 -nostdinc -fno-stack-protect
-Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/libs/ -Ikern/mm/ -Ikern/arch/ -c kern/driver/console.c -o obj/kern/driver/console.o
+ cc libs/string.c
riscv64-unknown-elf-gcc -Ilibs/ -mcmodel=medany -std=gnu99 -Wno-unused -Werror -fno-builtin -Wall -O2 -nostdinc -fno-stack-protector -f
libs/string.c -o obj/libs/string.o
+ cc libs/printfmt.c
riscv64-unknown-elf-gcc -Ilibs/ -mcmodel=medany -std=gnu99 -Wno-unused -Werror -fno-builtin -Wall -O2 -nostdinc -fno-stack-protector -f
libs/printfmt.c -o obj/libs/printfmt.o
+ cc libs/readline.c
riscv64-unknown-elf-gcc -Ilibs/ -mcmodel=medany -std=gnu99 -Wno-unused -Werror -fno-builtin -Wall -O2 -nostdinc -fno-stack-protector -f
libs/readline.c -o obj/libs/readline.o
+ cc libs/sbi.c
riscv64-unknown-elf-gcc -Ilibs/ -mcmodel=medany -std=gnu99 -Wno-unused -Werror -fno-builtin -Wall -O2 -nostdinc -fno-stack-protector -f
libs/sbi.c -o obj/libs/sbi.o
+ ld bin/kernel
riscv64-unknown-elf-ld -m elf64lriscv -nostdlib --gc-sections -T tools/kernel.ld -o bin/kernel obj/kern/init/entry.o obj/kern/init/init
ole.o obj/libs/string.o obj/libs/printfmt.o obj/libs/readline.o obj/libs/sbi.o
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.bin
```

提示：如果我们第二次执行 `make` 会显示 `make: Nothing to be done for 'TARGETS'`，这时候我们需要执行 `make clean` 命令，把我们生成的目标文件和中间文件给清除掉，才能执行 `make`。

我们可以看到，把 `entry.S` 和 `init.c`，`stdio.c` 等几个 `c` 文件编译成为 `.o` 目标文件，然后链接器[3]将 `.o` 文件链接成可执行文件 `kernel`（`elf` 文件），最后使用 `objcopy` 把 `elf` 文件转化成为 `ucore.bin`[2]，这是装我们最小化操作系统内核的二进制文件。接下来我们使用 `qemu` 中自带的 `OpenSBI`[1] 作为我们的 `bootloader`，启动我们的内核。

第三步. 使用 `make qemu` 启动我们的内核

当我们执行 `make qemu` 时，对应于

```
qemu-system-riscv64 \
  -machine virt \
  -nographic \
  -bios default \
  -device loader,file=bin/ucore.bin,addr=0x80200000
```

这条指令相当于给我们的模拟计算机插电，然后 `qemu` 会调用内置的 `OpenSBI` 作为我们的 `bootloader`。`OpenSBI` 所做的一件事情就是把 `CPU` 从 `M Mode` 切换到 `S Mode`，接着跳转到一个固定地址 `0x80200000`，开始执行内核代码。

知识点

1 OpenSBI

我们需要硬盘上的程序和数据。比如崭新的 `windows` 电脑里 `C` 盘已经被占据的二三十 `GB` 空间，除去预装的应用软件，还有一部分是 `windows` 操作系统的内核。在插上电源开机之后，就需要运行操作系统的内核，然后由操作系统来管理计算机。

问题在于，操作系统作为一个程序，必须加载到内存里才能执行。而“把操作系统加载到内存里”这件事情，不是操作系统自己能做到的，就好像你不能拽着头发把自己拽离地面。

因此我们可以想象，在操作系统执行之前，必然有一个其他程序执行，他作为“先锋队”，完成“把操作系统加载到内存”这个工作，然后他功成身退，把 `CPU` 的控制权交给操作系统。

这个“其他程序”，我们一般称之为 `bootloader`。很好理解：他负责 `boot` (开机)，还负责 `load` (加载 `OS` 到内存里)，所以叫 `bootloader`。

在 `QEMU` 模拟的 `riscv` 计算机里，我们使用 `QEMU` 自带的 `bootloader`: `OpenSBI` 固件。

知识点

在计算机中，**固件(firmware)**是一种特定的计算机软件，它为设备的特定硬件提供低级控制，也可以进一步加载其他软件。固件可以为设备更复杂的软件（如操作系统）提供标准化的操作环境。对于不太复杂的设备，固件可以直接充当设备的完整操作系统，执行所有控制、监视和数据操作功能。在基于 x86 的计算机系统中, BIOS 或 UEFI 是固件；在基于 riscv 的计算机系统中，OpenSBI 是固件。OpenSBI运行在**M态 (M-mode)**，因为固件需要直接访问硬件。

RISCV有四种**特权级 (privilege level)**。

Level	Encoding	全称	简称
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	
3	11	Machine	M

粗略的分类：

U-mode是用户程序、应用程序的特权级，S-mode是操作系统内核的特权级，M-mode是固件的特权级。

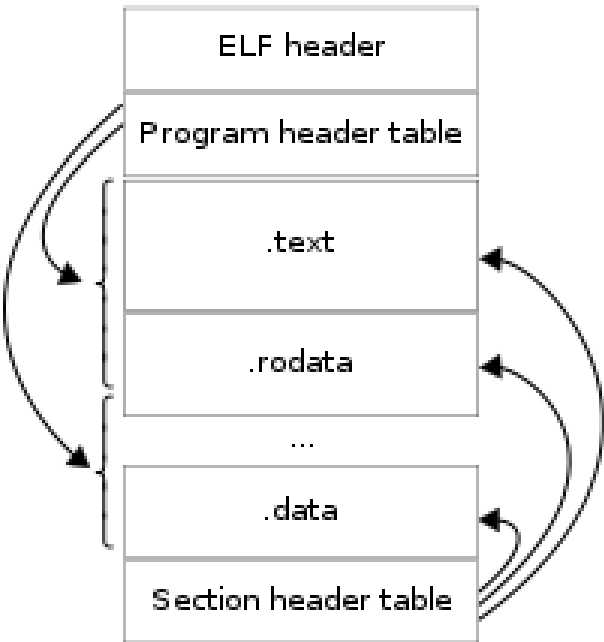
详细内容请自行查阅RISC-V手册。

2 ELF 和 BIN 文件

在Opensbi工作完成后，总会把program counter跳到 0x80200000 这个内存地址开始执行，所以我们要把操作系统镜像放在这个位置上。通过qemu命令中可以看到，我们加载的是bin文件。这里为什么不直接加载生成的ELF可执行文件呢？

这是因为elf文件比较复杂，包含一个文件头(ELF header)和冗余的调试信息，指定程序每个section的内存布局，需要解析program header才能知道各段(section)的信息。如果我们已经有一个完整的操作系统来解析elf文件，那么elf文件可以直接执行。但是对于OpenSBI来说还无法完成对于elf文件的解析。所以我们需要将复杂的ELF文件转换为简单的BIN二进制可执行文件。

ELF文件是linux系统上的主要可执行文件的格式(图片来自wiki)。BIN文件是二进制可执行文件。



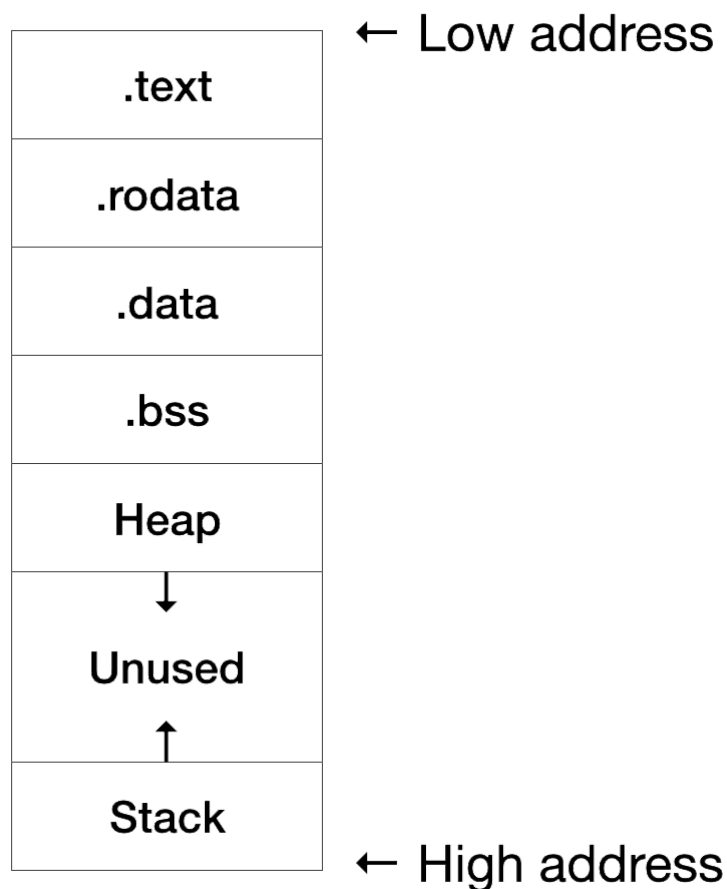
我们举一个例子解释elf和bin文件的区别：bss段是一个初始化为零的一个大数组，在elf文件里是bss数据段的一部分，只需要记住这个数组的起点和终点就可以了，等到加载到内存里的时候分配那一段内存。但是在bin文件里，那个数组有多大，有多少个字节的0，bin文件就要对应有多少个零。所以如果一个程序里声明了一个大全局数组（默认初始化为0），那么可能编译出来的elf文件只有几KB，而生成bin文件之后却有几MB，这是很正常的。实际上，可以认为bin文件会把elf文件指定的每段的内存布局都映射到一块线性的数据里，这块线性的数据（或者说程序）加载到内存里就符合elf文件之前指定的布局。

程序的内存布局

一般来说，一个程序按照功能不同会分为下面这些段：

- .text 段：代码段，存放汇编代码
- .rodata 段：只读数据段，顾名思义里面存放只读数据，通常是程序中的常量
- .data 段：存放被初始化的可读写数据，通常保存程序中的全局变量
- .bss 段：存放被初始化为 0 的可读写数据，与 .data 段的不同之处在于我们知道它要被初始化为 0，因此在可执行文件中只需记录这个段的大小以及所在位置即可，而不用记录里面的数据，也不会实际占用二进制文件的空间
- Stack：栈，用来存储程序运行过程中的局部变量，以及负责函数调用时的各种机制。它从高地址向低地址增长
- Heap：堆，用来支持程序**运行过程中**内存的**动态分配**，比如说你要读进来一个字符串，在你写程序的时候你也不知道它的长度究竟为多少，于是你只能在运行过程中，知道了字符串的长度之后，再在堆中给这个字符串分配内存

内存布局，也就是指这些段各自所放的位置。一种典型的内存布局如下：



3 链接脚本

gnu工具链中，包含一个链接器 `ld`。

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。

我们在链接脚本里把操作系统的入口点定义为 `kern_entry`，所以程序里需要有一个名称为 `kern_entry` 的符号。我们在 `kern/init/entry.s` 编写了一段汇编代码，作为整个内核的入口点。

这里为什么不能直接用C语言呢，因为在C语言层级上，无法进行寄存器操作，所以涉及到寄存器操作的部分我们都会使用汇编语言。在下一节中断保存现场使用汇编进行寄存器的保存，同样是这个道理。

```
#include <mmu.h>
#include <memlayout.h>

.section .text, "ax", %progbits
.globl kern_entry
kern_entry:
    la sp, bootstacktop

    tail kern_init
    #调用kern_init，这是我们要用C语言编写的一个函数，tail是riscv伪指令，作用相当于调用函数（跳转）
.section .data
    # .align 2^12
    .align PGSIZE
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:
```

里面有很多符号和指令，定义的符号，在头文件查看具体信息。关于指令，就需要去查阅相关资料和手册。

这里调用了唯一的一个函数，下一步我们去查看这个函数。

第四步. 找到C语言层级上的入口点

我们在 `kern/init/init.c` 编写函数 `kern_init`，作为“真正的”内核入口点。为了让我们能看到一些效果，我们希望它能在命令行进行格式化输出。

如果我们在linux下运行一个C程序，需要格式化输出，一般想法是我们应该 `#include<stdio.h>`。于是我们在 `kern/init/init.c` 也这么写一句。但是，linux下，当我们调用C语言标准库的函数时，实际上依赖于 `glibc` 提供的运行时环境，也就是一定程度上依赖于操作系统提供的支持。可是我们并没有把 `glibc` 移植到ucore里！

怎么办呢？只能自己动手，丰衣足食。QEMU里的OpenSBI固件提供了输入一个字符和输出一个字符的接口，我们一会把这个接口一层层封装起来，提供 `stdio.h` 里的格式化输出函数 `cprintf()` 来使用。这里格式化输出函数的名字不使用原先的 `printf()`，强调这是我们在ucore里重新实现的函数。

第五步. 使用OpenSBI提供的接口在屏幕上进行输出字符

OpenSBI作为运行在M态的软件（或者说固件），提供了一些接口供我们编写内核的时候使用。

我们可以通过 `ecall` 指令(environment call)调用OpenSBI。通过寄存器传递给OpenSBI一个“调用编号”，如果编号在 0-8 之间，则由OpenSBI进行处理，否则交由我们自己的中断处理程序处理（暂未实现）。有时OpenSBI调用需要像函数调用一样传递参数，这里传递参数的方式也和函数调用一样，按照riscv的函数调用约定(calling convention)把参数放到寄存器里。可以阅读[SBI的详细文档](#)。

知识点

ecall(environment call)，当我们在 S 态执行这条指令时，会触发一个 `ecall-from-s-mode-exception`，从而进入 M 模式中的中断处理流程（如设置定时器等）；当我们在 U 态执行这条指令时，会触发一个 `ecall-from-u-mode-exception`，从而进入 S 模式中的中断处理流程（常用来进行系统调用）。

C语言并不能直接调用 `ecall`，需要通过内联汇编来实现。

```
// libs/sbi.c
#include <sbi.h>
#include <defs.h>

//SBI编号和函数的对应
uint64_t SBI_SET_TIMER = 0;
uint64_t SBI_CONSOLE_PUTCHAR = 1;
uint64_t SBI_CONSOLE_GETCHAR = 2;
uint64_t SBI_CLEAR_IPI = 3;
uint64_t SBI_SEND_IPI = 4;
uint64_t SBI_REMOTE_FENCE_I = 5;
uint64_t SBI_REMOTE_SFENCE_VMA = 6;
uint64_t SBI_REMOTE_SFENCE_VMA_ASID = 7;
uint64_t SBI_SHUTDOWN = 8;
//sbi_call函数是我们关注的核心，封装ecall操作
uint64_t sbi_call(uint64_t sbi_type, uint64_t arg0, uint64_t arg1, uint64_t
arg2) {
    uint64_t ret_val;
    __asm__ volatile (
        "mv x17, %[sbi_type]\n"
        "mv x10, %[arg0]\n"
        "mv x11, %[arg1]\n"
        "mv x12, %[arg2]\n"    //mv操作把参数的数值放到寄存器里
        "ecall\n"             //参数放好之后，通过ecall，交给OpenSBI来执行
        "mv %[ret_val], x10"
        //OpenSBI按照riscv的calling convention,把返回值放到x10寄存器里
        //我们还需要自己通过内联汇编把返回值拿到我们的变量里
        : [ret_val] "=r" (ret_val)
        : [sbi_type] "r" (sbi_type), [arg0] "r" (arg0), [arg1] "r" (arg1),
[arg2] "r" (arg2)
        : "memory"
    );
    return ret_val;
}

void sbi_console_putchar(unsigned char ch) {
    sbi_call(SBI_CONSOLE_PUTCHAR, ch, 0, 0); //注意这里ch隐式类型转换为int64_t
}

void sbi_set_timer(unsigned long long stime_value) {
    sbi_call(SBI_SET_TIMER, stime_value, 0, 0);
}
```


知识点

函数调用与calling convention

我们知道，编译器将高级语言源代码翻译成汇编代码。对于汇编语言而言，在最简单的编程模型中，所能够利用的只有指令集中提供的指令、各通用寄存器、CPU 的状态、内存资源。那么，在高级语言中，我们进行一次函数调用，编译器要做哪些工作利用汇编语言来实现这一功能呢？

显然并不是仅用一条指令跳转到被调用函数开头地址就行了。我们还需要考虑：

- 如何传递参数？
- 如何传递返回值？
- 如何保证函数返回后能从我们期望的位置继续执行？

等更多事项。通常编译器按照某种规范去翻译所有的函数调用，这种规范被称为 [calling convention](#)。值得一提的是，为了实现函数调用，我们需要预先分配一块内存作为 **调用栈**，后面会看到调用栈在函数调用过程中极其重要。你也可以理解为什么第一章刚开始我们就要分配栈了。

可以参考[riscv calling convention](#)

现在可以输出一个字符了，有了第一个，就会有第二个第三个.....第无数个。

这样我们就可以通过 `sbi_console_putchar()` 来输出一个字符。接下来我们要做的事情就像月饼包装，把它封了一层又一层。

`console.c` 只是简单地封装一下

```
// kern/driver/console.c
#include <sbi.h>
#include <console.h>
void cons_putc(int c) {
    sbi_console_putchar((unsigned char)c);
}
```

`stdio.c`里面实现了一些函数，注意我们已经实现了ucore版本的puts函数: `cputs()`

```
// kern/libs/stdio.c
#include <console.h>
#include <defs.h>
#include <stdio.h>

/* HIGH level console I/O */

/* *
 * cputch - writes a single character @c to stdout, and it will
 * increace the value of counter pointed by @cnt.
 * */
static void cputch(int c, int *cnt) {
    cons_putc(c);
    (*cnt)++;
}

/* cputchar - writes a single character to stdout */
void cputchar(int c) { cons_putc(c); }

int cputs(const char *str) {
    int cnt = 0;
    char c;
```

```

while ((c = *str++) != '\0') {
    cputch(c, &cnt);
}
cputch('\n', &cnt);
return cnt;
}

```

六、增加GDB调试(optional)

在这里，我们提供GDB调试ucore的方法：

安装gdb：

```

sudo apt install gdb=9.1-0ubuntu1

sudo apt install gdb-multiarch

```

第一步 修改Makefile

在CFLAGS 中添加 -g 参数

```

CC      := $(GCCPREFIX)gcc
CFLAGS  := -mmodel=medany -std=gnu99 -wno-unused -werror
CFLAGS += -fno-builtin -Wall -O2 -nostdinc $(DEFS)
CFLAGS += -fno-stack-protector -ffunction-sections -fdata-sections
CTYPE   := c S

#在以上代码的CFLAGS中添加`-g`参数
CC      := $(GCCPREFIX)gcc
CFLAGS  := -mmodel=medany -std=gnu99 -wno-unused -werror -g
CFLAGS += -fno-builtin -Wall -O2 -nostdinc $(DEFS)
CFLAGS += -fno-stack-protector -ffunction-sections -fdata-sections
CTYPE   := c S

```

在qemu 对应的命令下面添加

```

.PHONY: qemu
qemu: $(UCOREIMG) $(SWAPIMG) $(SFSIMG)
# $(V)$(QEMU) -kernel $(UCOREIMG) -nographic
$(V)$(QEMU) \
    -machine virt \
    -nographic \
    -bios default \
    -device loader,file=$(UCOREIMG),addr=0x80200000

# ----添加以下内容-----
.PHONY: qemu-gdb
qemu-gdb: $(UCOREIMG) $(SWAPIMG) $(SFSIMG)
$(V)$(QEMU) \
    -machine virt \
    -nographic \
    -bios default \
    -device loader,file=$(UCOREIMG),addr=0x80200000
    -S -gdb tcp::26000

```

第二步 在项目的目录下新建 `.gdbinit` 文件并添加以下内容

```
set confirm off
set architecture riscv:rv64
target remote 127.0.0.1:26000
symbol-file bin/kernel
set disassemble-next-line auto
set riscv use-compressed-breakpoints yes
```

第三步 在/home/oslab(这里换成自己的用户名)/.gdbinit (如无请新建) 下面添加

```
add-auto-load-safe-path /home/oslab/Desktop/lab/.gdbinit
//这里要换成你自己的路径
```

否则运行时会出现以下错误:

```
warning: File "/home/oslab/Desktop/lab0/.gdbinit" auto-loading has been declined
by your `auto-load safe-path' set to "$debugdir:$datadir/auto-
load:/home/oslab/Desktop/xv6-riscv/.gdbinit:/home/oslab/Desktop/lab2/.gdbinit".
To enable execution of this file add
    add-auto-load-safe-path /home/oslab/Desktop/lab0/.gdbinit
line to your configuration file "/home/oslab/.gdbinit".
To completely disable this security protection add
    set auto-load safe-path /
line to your configuration file "/home/oslab/.gdbinit".
```

第四步 运行gdb

```
# 在项目文件夹下面 运行
make qemu-gdb

# 在另一个terminal 运行
gdb-multiarch
```

关于GDB的使用, 请自行阅读GDB手册以及相关的博客。

如果有使用VSCode进行调试的需求, 请在上面的基础上, 参照以下链接完成配置:

<https://zhuanlan.zhihu.com/p/501901665>

六、本节知识点回顾

本次lab你需要理解以下概念:

- 1.操作系统启动过程
- 2.bootloader的角色
- 3.ELF文件和BIN文件
- 4.内存布局是什么
- 5.链接脚本的作用
- 6.entry.S的作用
- 7.C语言的入口点

七、下一实验简单介绍

在下一实验中，我们将在最小化内核的基础上，完成基本的中断处理。