操作系统lab1

小组成员：方艾佳 杜子妍 谢闻星

一、实验涉及基本原理

（1）CPU上电过程

当 CPU 上电时，系统内存（DRAM）也同时上电。

上电瞬间：CPU 内所有寄存器的值通常为 0（部分特殊寄存器可能有固定值，例如 PC 的复位地址）；内存状态未初始化，处于不确定状态；因此系统此时的整体状态是非确定性的。

在上电阶段，CPU 的指令取决于 PC 的初值。PC 指向的地址必须是可以取到有效指令的位置，而由于普通内存掉电即丢失数据，因此通常指令不能直接从 DRAM 取得。

（2）上电后的指令获取流程

1. CPU 上电后首先从 ROM 中取指令。
2. ROM 中的指令通常是硬件初始化程序，使用固件引导加载程序，例如磁盘驱动程序，用于从硬盘或其他 storage 设备加载更多程序到内存。
3. Bootloader 执行完成后，将 PC 跳转到 DRAM 中操作系统内核的位置，使操作系统开始执行。

在 RISC-V 系统中的具体表现：

1. CPU 上电后会跳转到一个固定复位地址；
2. 该地址通常映射到 **OpenSBI 固件** 所在位置；
3. OpenSBI 固件逐条执行初始化指令（初始化寄存器、内存、外设等）；
4. 完成初始化后，OpenSBI 将控制权（PC）交给操作系统内核，操作系统开始运行。

> 可以理解为：CPU → OpenSBI → 内核

（3）自底向上构建I/O功能

根据学习和实验指导书的代码，发现封装标准库函数需要从底层到高级功能层层递进，从与硬件交互，到封装简单的功能，再到将简单功能封装成更高级的调用接口。我们以cprint函数为例，该函数可以格式化打印字符串。

1.在sbi.c文件中的sbi_call函数中，采用汇编内联，调用SBI的系统接口，该接口可以向终端输出一个字符。使用汇编语言定义了一个通用的 SBI 调用接口，需要使用汇编语言指定变量存放的寄存器，比如 `SBI` 功能编号（例如，`SBI_CONSOLE_PUTCHAR` = 1）放入指定的寄存器（通常是 a7 或 x17）。执行 `ecall` 指令，`CPU` 会 从S模式 到 M 模式，由 `OpenSBI` 固件处理请求。在S模式下是不能直接调用SBI接口的，需要权限升级。

代码块

```
1               __asm__ volatile (
2
3           "mv x17, %[sbi_type]\n"//存放sbi服务编号
4
5           "mv x10, %[arg0]\n"//参数入栈
6
7           "mv x11, %[arg1]\n"
8
9           "mv x12, %[arg2]\n"
10
11          "ecall\n"//S模式进入M模式
12
13          "mv %[ret_val], x10"
14
15          : [ret_val] "=r" (ret_val)
16
17          : [sbi_type] "r" (sbi_type), [arg0] "r" (arg0), [arg1] "r" (arg1),
    [arg2] "r" (arg2)
18
19          : "memory"
20
21      );
```

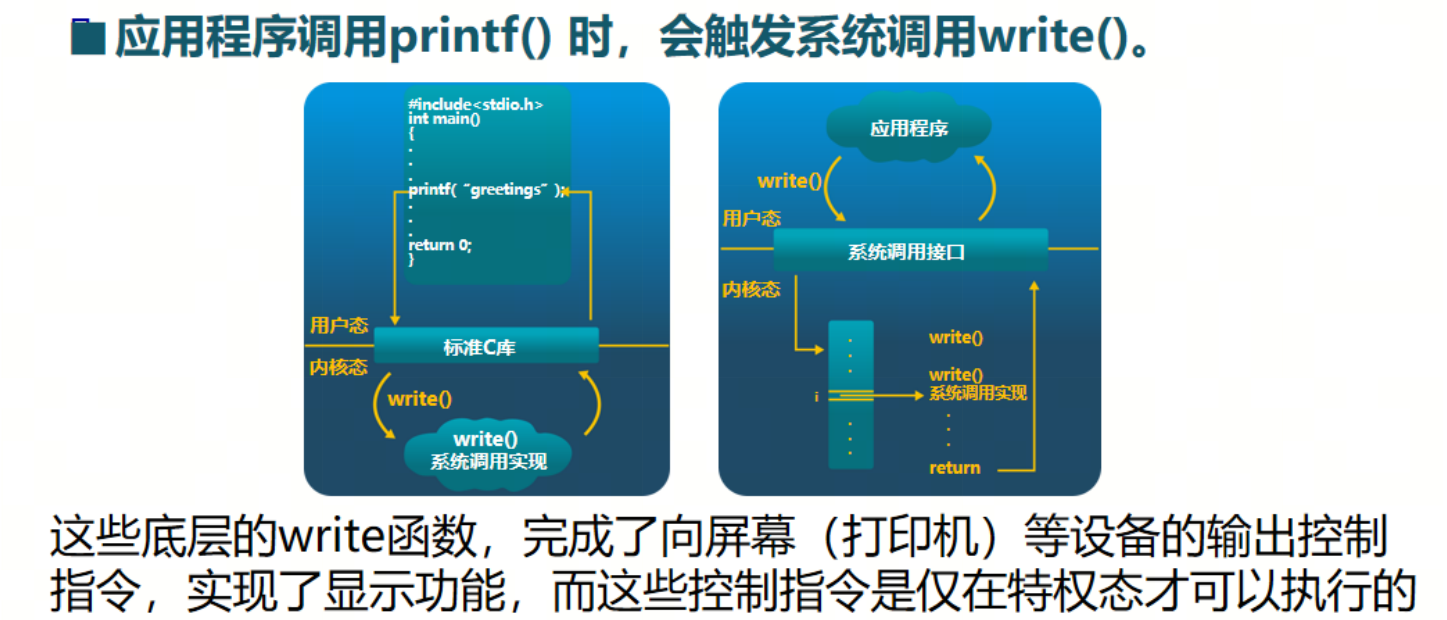
2.在sbi.c的sbi_console_putchar函数进一步封装底层SBI接口调用函数，提供一个标准函数，通过输入一个unsigned char即可调用该函数，用来让 S 模式通过 `ecall` 向 M 模式固件（比如 OpenSBI）请求输出一个字符到控制台。

3.在console.c文件中的cons_putc进一步封装sbi_console_putchar函数实现向控制台输出，统一控制台接口。

4.在printmt.c中定义vprintmt函数，实现格式化字符串的输出，将单个字符输出函数cputch作为参数传入，统计已经输出的字符串数目，对格式化字符串进行处理。

5.在stdio.c文件中的cprint函数封装了格式化输出字符串功能的函数vprintmt，准备参数进行调用。

仔细学习该流程，为以后其他标准函数实现打下基础。



在课程里有问到调用标准库函数时发生了什么?

应用程序调用printf() 时，会触发系统调用write()，在lab1中就是cprint()函数调用，会触发系统调用sbi_call()。但是不同的是一个是从用户态到内核态，一个是从内核态到特权模式M。如果是在RISCV框架下，用户程序执行 I/O，CPU会从U模式→S模式，此时控制权转交给内核，S 模式处理系统调用，调用相应的内核函数，但S 模式本身也不能直接访问硬件，它必须请求 M 模式固件（OpenSBI）来执行，然后返回到U模式。用户程序不能直接访问硬件，必须通过系统调用来请求操作系统服务。Lab1通过SBI的ecall调用，展示了内核如何与更底层的固件交互，这与用户程序通过系统调用与内核交互的原理是相似的。

(4)权限

实验中涉及到权限交替：

1.在OS的初始化和加载中，在OpenSBI初始化时为M模式，跳转到OS时进入S模式。

2.在封装cprint函数时，从S模式进入M模式请求SBI的系统接口。

模式	职责	访问权限
M 模式	负责初始化硬件、内存、外设 设置中断、页表支持等 启动 OS（切换到 S 模式）	可访问所有寄存器、内存、I/O
S 模式	管理进程调度、内存分页、系统调用 响应中断、异常	可访问部分 CSR（受限寄存器）和内核空间
U 模式	执行用户应用程序	只能访问用户空间内存

	不能直接访问内核或硬件 只能通过系统调用（`ecall`）请求内核服务	

用户程序执行 `ecall` 指令从低层→高层；使用用 `mret` 指令M 模式 → S 模式 ；

和我们课程讲授的知识联系起来，这一部分是一致的，进入特权态才能执行特权指令、访问特权数据、以及执行存储在特权区域的代码，只有在M模式才能执行OpenSBI的代码，执行SBI接口调用指令等。

通过这样的机制确保程序、内核、固件、硬件之间的 安全隔离与受控交互，实现了 “分层控制、逐级授权”。

这也让我联想到老师上课提到的，当通过MMU虚拟地址转换器发现该虚拟也没有对应物理页时，由什么处理缺页异常？用户态访问虚拟地址触发 Page Fault ，进入内核态，控制权转移给操作系统中的页面故障处理程序，进行一系列操作，如果修复成功，返回用户态重新执行。

(5)内存布局与链接

链接脚本的重要性，它精确控制了内核代码段（.text）、数据段（.data）、只读数据段（.rodata）和未初始化数据段（.bss）在内存中的布局，并指定了内核的加载地址（0x80200000）和入口点（kern_entry）。同时，解释了相关地址和绝对地址，即内核代码在编译时就确定了绝对内存地址，因此必须加载到指定位置。对于内存布局，需要区分ELF文件和BIN文件，源文件先编译成目标文件，再通过链接脚本生成ELF文件，对于Qemu只能运行bin文件，需要将ELF转换为BIN文件以便Bootloader加载。 `.bss` 段在两者间有很大不同，在ELF文件中该段只需要声明大小，在BIN文件中需要用0填充该段，在kern_init()函数中也完成了这一步。

上课提到指令需要进入内存才能被处理器执行，与指令配套的数据也需要加载入内存，这些都隐含了对内存布局的需求。

链接脚本和内存布局是操作系统内核能够正确运行的先决条件,它是将高级语言编写的内核代码转化为机器可执行代码并放置在物理内存中的关键步骤。根据实验分析，Lab1的kernel.ld文件实现了内存布局设定和内核加载位置指定、内核入口确定。在 `kern_init()` 函数中也对 `.bss` 段进行处理，在bin文件中需要将.bss段用0填充。

(6)栈的初始化与交接

掌握 la sp, bootstacktop 指令在 entry.S 中的作用，即为 C 语言函数（如 kern_init）调用准备栈空间，并理解 RISC-V 栈向下增长的特性。接着，通过对比 OpenSBI 临时栈和内核新栈的内存内容，理解栈空间从固件到内核的平稳过渡，以及内核如何建立自己的独立运行环境。

(7) 固件的解释

它是一段在操作系统内核启动之前运行，并且在操作系统运行期间提供底层服务的关键软件。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，因为固件需要直接访问硬件。在计算机加电后、操作系统内核开始运行前，OpenSBI 负责承上启下。当操作系统需要执行一些只有最高权限（M-Mode）才能完成的操作时（例如：彻底关闭计算机、操作底层计时器等），它不能直接执行，因为它权限不够。这时，操作系统就会通过一个标准的调用约定（就是SBI），向仍然驻留在 M-Mode 的 OpenSBI 发起一个服务请求。

(8）固件和内核的加载地址

为何作为 bootloader 的 OpenSBI要被加载到 `0x80000000` 开头的区域上呢？加载到其他的地址空间可以吗？

简单来说，这是由QEMU 模拟器的启动约定所决定的。计算机加电后，CPU 首先执行 `0x1000` 地址的复位代码 。这段极其简单的代码只有一个核心任务：准备好环境，然后将控制权交给下一阶段的程序，也就是 bootloader (OpenSBI)。为了顺利完成交接，前一阶段的程序必须知道下一阶段的程序在哪里。在 QEMU 的这套实现中，复位代码被设计为**“默认去 `0x80000000` 地址找 OpenSBI”** 。因此，OpenSBI 必须被预先加载到这个双方约定好的地址上，否则复位代码就会扑空，启动流程就无法继续。所以，将 OpenSBI 加载到 `0x80000000` 并不是一个随意的选择，而是 QEMU 启动流程中，由第一阶段的复位代码所指定的、一个固定的交接地址 。

为什么内核一定要被加载到 0x80200000 的位置？加载到别的位置行不行呢？

对于我们实验中这个已经被编译好的内核文件来说，不行，它必须被加载到 `0x80200000` 。根本原因在于，这个内核被编译成了地址相关代码。

指导书中用一个很好的例子进行了解释：当编译器处理 C 语言代码如 `sum += 5;` 时，它需要知道变量 `sum` 的确切内存地址（比如 `0x55aa55aa` ） 。然后，编译器会将这个绝对地址直接硬编码到生成的机器指令中，例如生成一条类似 “将地址 `0x55aa55aa` 的值加载到寄存器” 的指令 。

这意味着，内核代码中的许多指令都包含了基于 `0x80200000` 这个起始地址计算出来的绝对地址 。整个内核程序就像一个被精确测量和切割好的家具，它被设计得只能刚刚好地放进一个特定尺寸和位置的房间里。

假如我们将这个内核加载到 `0x90000000` ，但代码内部的指令仍然要去访问那个基于 `0x80200000` 计算出的旧地址，那么它访问到的将是完全错误的数据或代码，程序必然会崩溃 。

因此，为了让 bootloader (OpenSBI) 和内核能够正确对接，双方必须遵守这个约定：OpenSBI 负责将内核加载到 `0x80200000` ，而内核在编译时就已经设定好自己将在这个地址上运行。

二、对应 OS 原理中的重要知识点：

1. 引导过程 ： 整个实验是对操作系统引导过程的实践和深入理解。从硬件上电到操作系统内核运行，涉及多阶段的加载和初始化，体现了操作系统启动的复杂性与层次性。

2. 程序与进程的静态与动态：实验中强调“程序只是静态文件；运行后 + 系统管理 → 进程”，通过 entry.S 和 kern_init 的执行，展示了静态代码如何被加载、初始化并最终成为一个动态运行的内核进程。
3. 上下文切换的萌芽：虽然实验尚未涉及完整的进程上下文切换，但 entry.S 中栈的初始化和 tail 跳转，已经包含了为后续 C 语言环境准备执行上下文的初步思想。
4. 系统调用与接口：OpenSBI 提供的 SBI 接口，是固件与操作系统之间进行交互的标准方式，这与操作系统向用户程序提供系统调用接口的原理是相通的，都是为了实现不同层级之间的安全、规范通信。
5. 内存地址的依赖差异：

   · 地址相关代码 (PDC)：代码中包含硬编码的绝对内存地址。实验中的内核在编译后就是地址相关代码。

   · 地址无关代码 (PIC)：代码中不包含绝对内存地址，所有地址引用都是相对的。

   · 差异：核心差异在于对内存地址的依赖性。PDC 依赖固定地址，PIC 则不依赖。在现代操作系统中，为了提高灵活性和安全性（如 ASLR），通常会使用 PIC。

三、实验内容

练习1：理解内核启动中的程序入口操作

（1）前置知识

阅读项目代码，结合操作系统内核启动流程，说明挑战1的问题。我们先根据自己的学习说明了内核加载执行过程，在此基础上进行回答。

1.加载OS和初始化

1.复位地址是 0x1000(这是由CPU设计规定的)，然后跳转到SBI固件入口地址0x80000000。

2.进入SBI固件入口地址，在M模式下，初始化OpenSBI，比如加载实验手册中提到的bootloader程序，解析由QEMU传入的设备树（DTB），了解模拟硬件的配置。准备进入S模式，然后跳转到 S 模式内核。

3.跳转到内核地址后，从符号 kern_entry 处开始执行。

2.程序间的关联

通过上述过程分析，在kernel.ld这个内核链接脚本中，说明了OS加载地址为 `BASE_ADDRESS = 0x80200000`，同时说明程序的入口地址为 `ENTRY(kern_entry)`。这两条说明为后续OS的正常加载和初始化提供指引，让CPU知道OS的位置，和跳转到OS后，从哪一条指令开始执行。同时在kernel.ld中还说明了将目标文件的不同段对应到ELF文件不同段的规则，并记录不同段开始和结束的位置。

3.分析entry.S文件

这是RISCV汇编语言文件，声明了代码段和数据段。

文件声明全局变量`globl kern_entry`,将该符号声明为全局可见，让它在链接阶段能被其他文件访问。根据编译原理可知，在链接前的阶段，对于外部的变量和函数由于不知道具体地址而使用替代符，在链接阶段将真实地址填充，kernel.ld中定义的`ENTRY(kern_entry)`会由真实入口指令地址填充。进入这个入口地址，便是执行如下两条指令`la sp, bootstacktop`和`ail kern_init`。数据段中声明了栈顶和栈底指针。

（2）阅读 `kern/init/entry.S` 内容代码，结合操作系统内核启动流程

> 指令 `la sp, bootstacktop` 完成了什么操作，目的是什么？

la 是 RISC-V 的伪指令（load address），它会把符号 `bootstacktop` 的绝对地址计算出来并写入到寄存器 `sp` （栈指针）

`la sp, bootstacktop`

- `la`： `load address` 指令，用于将符号地址加载到寄存器中
- `sp`：栈指针寄存器
- `bootstacktop`：一个符号，表示内核启动栈的顶端地址

内核启动时需要为内核代码分配栈空间， `la sp, bootstacktop` 将 `bootstacktop` 的地址加载到 `sp` 寄存器(起点),RISC-V 栈向下增长。当执行 `la sp, bootstacktop` 时， `sp` 寄存器最初指向的是栈区域的最高地址。随着函数调用和数据压栈， `sp` 的值会逐渐减小，向低地址方向移动。

> `tail kern_init` 完成了什么操作，目的是什么？

tail 也是 RISC-V 的伪指令，无返回地跳转到目标函数入口（这里是 kern_init），这条指令通过tail实现直接跳转到kern_init函数位置，并开始执行该函数的代码。注意，j和call是有区别的，这里等同于j指令。

目的是从底层汇编引导代码转向更高级的C语言初始化函数 kern_init。OpenSBI将OS加载到指定位置，并跳转到 S 模式内核，启动OS。从内核的指定入口位置开始执行代码，准备kern_init函数的栈空间，然后跳转到C语言主体部分kern_init。

总的来说QEMU 把内核镜像加载到链接脚本指定的基地址（0x80200000），PC 从入口 `kern_entry` 开始执行。第一件事“la sp, bootstacktop”：建立内核栈，使得接下来 C 代码可以安全运行。第二件事“tail kern_init”：不设置返回地址，直接跳到 C 的初始化函数，完成 BSS 清零、打印信息等初始化工作，然后按照设计进入死循环。

练习2: 使用GDB验证启动流程

为了熟悉使用 QEMU 和 GDB 的调试方法，请使用 GDB 跟踪 QEMU 模拟的 RISC-V 从加电开始，直到执行内核第一条指令（跳转到 0x80200000）的整个过程。通过调试，请思考并回答：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？请在报告中简要记录你的调试过程、观察结果和问题的答案。

> **tips:**
>
> 启动流程可以分为以下几个阶段：
>
> 1. CPU 从复位地址（`0x1000`）开始执行初始化固件（OpenSBI）的汇编代码，进行最基础的硬件初始化。
> 2. SBI 固件进行主初始化，其核心任务之一是将内核加载到 `0x80200000`。可以使用 `watch *0x80200000` 观察内核加载瞬间，避免单步跟踪大量代码。
> 3. SBI 最后跳转到 0x80200000，将控制权移交内核。使用 `b *0x80200000` 可在此中断，验证内核开始执行

我们采用远程调试，让内核在 **QEMU** 这个“带监控的模拟电脑”里运行。然后我们从外部用 **GDB** 这个“监控终端”连接上去，从而实现对内核的完全控制和观察。

1.运行 `make gdb` 指令

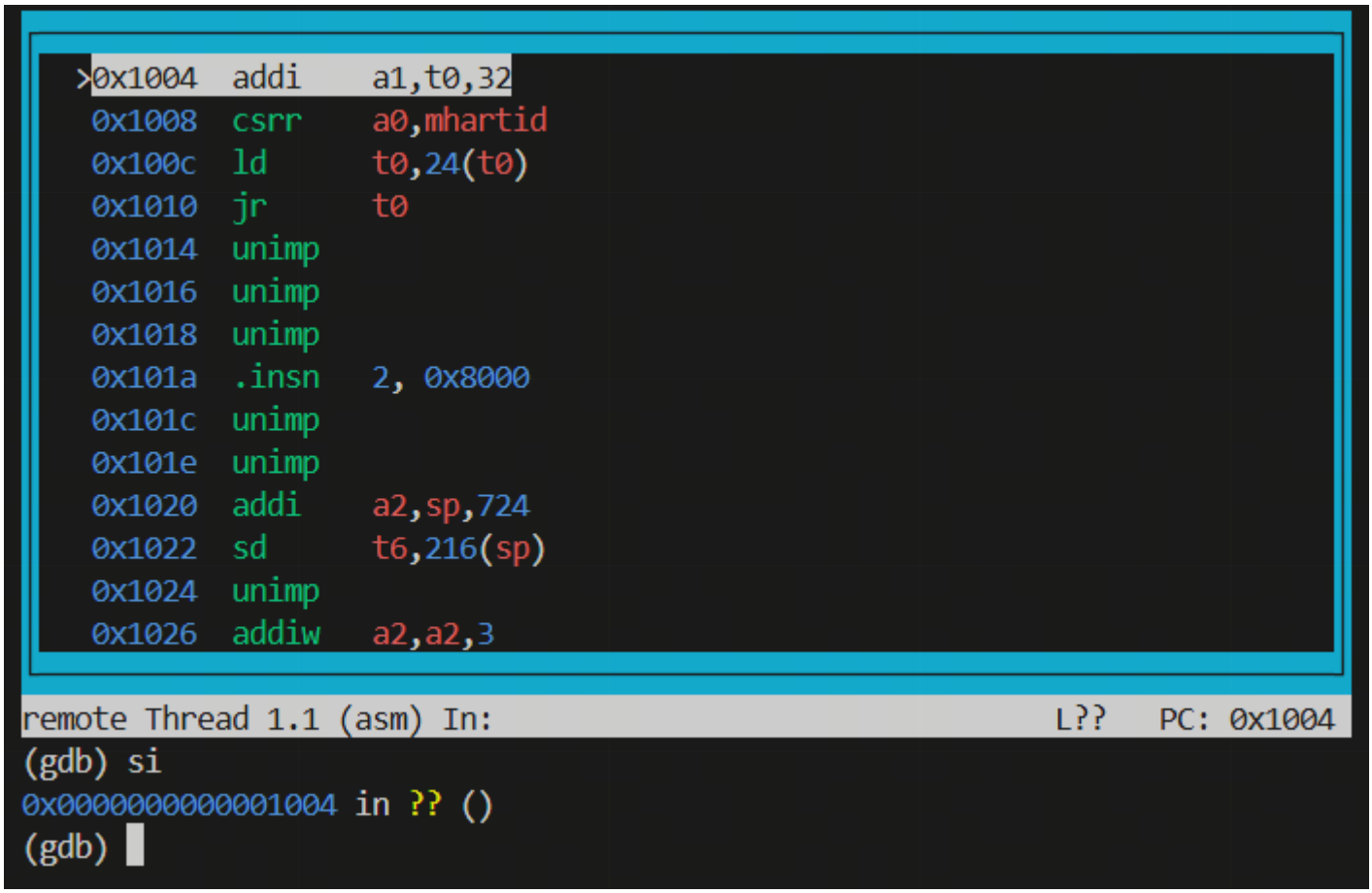
```
The target architecture is set to "riscv:rv64".
Remote debugging using localhost:1234
0x0000000000001000 in ?? ()
```

GDB确认它已经将目标架构设置为64位RISC-V，成功通过TCP连接到了 本地主机：1234 上的远程目标（即QEMU）。即qemu作为被调试目标，gdb通过端口观测qemu的行为。

GDB成功地在QEMU模拟器刚刚加电的时刻暂停了CPU，PC寄存器指向即将要执行的下一条指令的内存地址0x0000000000001000，即CPU的复位地址。in？？（）意味着GDB无法将地址 0x1000 映射到任何已知的函数名。因为地址 0x1000 处的代码并不属于编译的内核 。这部分代码是QEMU模拟的硬件固化在ROM中的一小段引导程序。加载的 bin/内核文件的符号表里自然没有关于这块内存地址的信息。

2.从0x1000单步调试直到进入OpenSBI

通过指令 `layout asm` 进入反汇编查看0x1000内存位置对应的汇编代码，如下



在RISCV中寄存器数量有限，寄存器有明确的分工，比如a0,a1寄存器作为函数的参数寄存器。`addi a1,to,32` 汇编指令执行的是计算 `t0` 寄存器的值加上立即数 `32`，然后将结果存入 `a1` 寄存器，准备传递给OpenSBI的参数-DTB文件的地址。

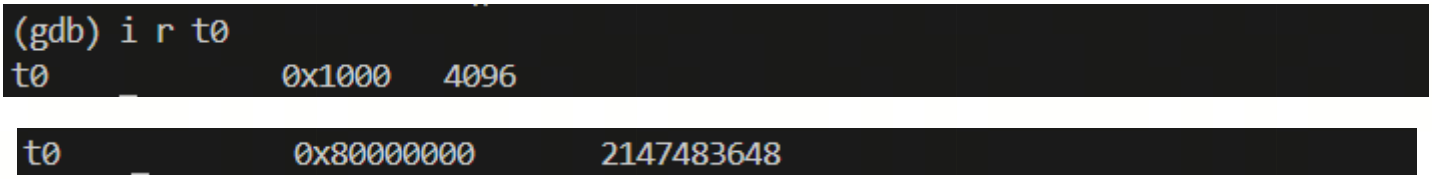
`csrr ao,mhartid` 读取名为 `mhartid` 的特殊寄存器的值（`mhartid` 存放的是当前硬件线程（CPU核心）的唯一ID），并将其存入 `a0` 寄存器，这条指令正在准备传递给OpenSBI的第一个参数-当前CPU核心的ID。

`ld t0,24(t0)` 执行的是从内存中加载一个64位的值到 `t0` 寄存器。加载的内存地址由 `t0` 寄存器的值加上偏移量 `24` 得到,即0x1000+24=0x1018。根据汇编代码可知，在0x1018处，加载到t0的值为0x8000。

`jr t0` 跳转指令，跳转到寄存器中的地址，也就是 `0x80000000`

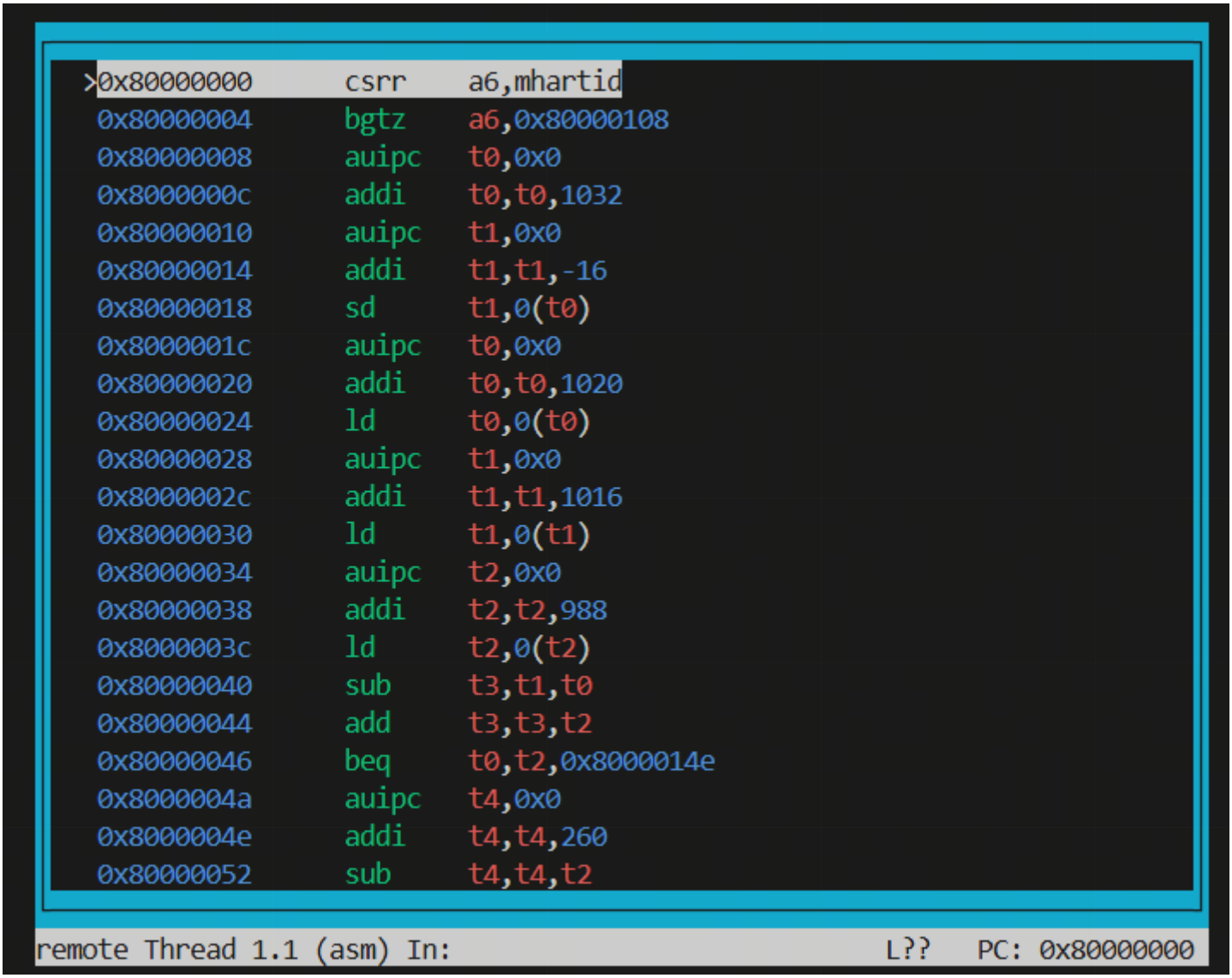
控制权被正式移交给了OpenSBI固件，这个MROM的任务到此结束。

如下通过 `i r t0` 指令观察t0寄存器值的变化：执行 `ld` 指令后，t0的值从0x1000变为0x80000000



3.进入OpenSBI初始化

通过 `jr t0` 指令进入OpenSBI初始化，CPU 跳转到 `0x80000000` 处继续运行。



在OpenSBI初始化与内核加载部分，①在 RISC-V 的最高特权级（M 模式）下，该阶段负责初始化处理器的运行环境，②准备进入S模式，③执行跳转，PC跳转到0x80200000内核处，权限级别从M模式切换到 S模式。

使用watch*0x80200000设置观察点，运行程序，发现终端没有输出，运行到了死循环。即在OpenSBI初始化阶段内存位置0x80200000没有变化。后来我们又设置实验，在QEMU模拟器刚刚加电的时刻暂停了CPU，通过指令`disassemble 0x80200000, 0x80200100`观察0x80200000位置的汇编代码如下图。在此时OS操作系统已经加载到指定位置。查阅资料发现，在真机中CPU 复位后执行的是 ROM 里的固件；这个固件或 Bootloader 会去读取 磁盘 / Flash / SD 卡 / 网络 / 串口等地方的 OS 镜像；再把它复制（load）到内存指定地址；最后把 CPU 的 PC 设置为内核入口地址，然后跳转执行。但在模拟器中，qemu将直接把BIN 镜像加载到内存，在CPU复位时就可以观察到指定位置已经加载内核。那么在 Qemu 开始执行任何指令之前，首先要将作为 bootloader 的 OpenSBI.bin 加载到物理内存以物理地址 0x80000000 开头的区域上，然后将操作系统内核加载到QEMU模拟的硬盘中。

```
(gdb) disassemble 0x80200000, 0x80200100
Dump of assembler code from 0x80200000 to 0x80200100:
   0x0000000080200000 <kern_entry+0>:   auipc   sp,0x3
   0x0000000080200004 <kern_entry+4>:   mv      sp,sp
   0x0000000080200008 <kern_entry+8>:   j       0x8020000a <kern_init>
   0x000000008020000a <kern_init+0>:    auipc   a0,0x3
   0x000000008020000e <kern_init+4>:    addi    a0,a0,-2 # 0x80203008
   0x0000000080200012 <kern_init+8>:    auipc   a2,0x3
   0x0000000080200016 <kern_init+12>:   addi    a2,a2,-10 # 0x80203008
   0x000000008020001a <kern_init+16>:   addi    sp,sp,-16 # 0x80202ff0
   0x000000008020001c <kern_init+18>:   li      a1,0
```

但是此时mepc寄存器(存放OS入口地址)的值为0

```
(gdb) i r mepc
mepc            0x0      0
```

我们使用指令`(gdb) find /b 0x80000000, +0x20000, 0x73, 0x00, 0x20, 0x30`寻找机器码0x30200073即mret对应指令的位置，并且在每一个mret位置设置断点。执行到第一个断点处，观察寄存器mepc的值，发现变为0x80200000。并且在该处再单步调试，进入kern_entry()。我们进入这个阶段的汇编代码查看，并且查看寄存器的值。

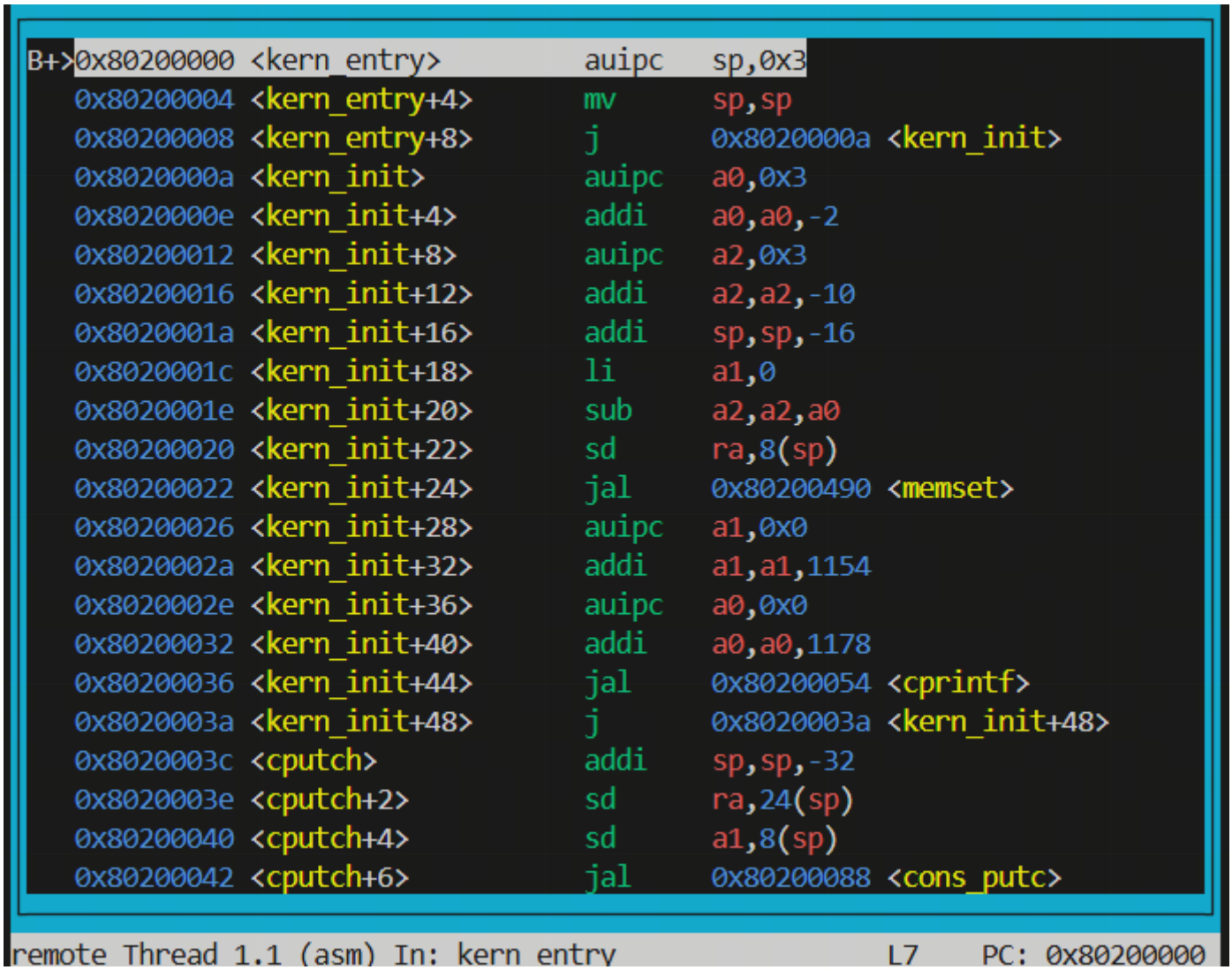
mepc的值已变为入口地址，mstatus的值为1，对应S模式。执行mret指令时，先从mstatus确定下一个模式，然后CPU跳转到mepc地址处，最后清理寄存器的值。

但是进入S模式后，便无法再访问mepc、mstatus寄存器的值，不具有权限。

```
   0x0000000080005006:  csrw    mstatus,a5
   0x000000008000500a:  csrw    mepc,a2
   0x000000008000500e:  li      a5,1
   0x0000000080005010:  beq     a3,a5,0x80005026
   0x0000000080005014:  bnez    a3,0x80005022
   0x0000000080005016:  csrw    0x5,a2
   0x000000008000501a:  csrwi   0x40,0
   0x000000008000501e:  csrwi   0x4,0
   0x0000000080005022:  mret
   0x0000000080005026:  csrw    stvec,a2
   0x000000008000502a:  csrwi   sscratch,0
   0x000000008000502e:  csrwi   sie,0
   0x0000000080005032:  csrwi   satp,0
=> 0x0000000080005036:  mret
   0x000000008000503a:  addi    sp,sp,-32
   0x000000008000503c:  sd      ra,24(sp)
   0x000000008000503e:  sd      s0,16(sp)
   0x0000000080005040:  sd      s1,8(sp)
   0x0000000080005042:  addi    s0,sp,32
   0x0000000080005044:  mv      s1,a0
   0x0000000080005046:  auipc   a0,0x6
   0x000000008000504a:  addi    a0,a0,10 # 0x8000b050
   0x000000008000504e:  jal     0x80003244
   0x0000000080005052:  auipc   a4,0x6
   0x0000000080005056:  addi    a4,a4,-10 # 0x8000b048
   0x000000008000505a:  ld      a3,0(a4)
   0x000000008000505c:  li      a5,1
   0x000000008000505e:  sll     a5,a5,s1
End of assembler dump.
(gdb) i r mepc
mepc           0x80200000       2149580800
(gdb) i r mstatus
mstatus        0x8000000000006800       SD:1 VM:00 MXR:0 PUM:0 MPRV:0 XS:0 F
S:3 MPP:1 HPP:0 SPP:0 MPIE:0 HPIE:0 SPIE:0 UPIE:0 MIE:0 HIE:0 SIE:0 UIE:0
(gdb) i r a5
a5             0x1      1
```

4.进入entry.S

在OpenSBI阶段代码过多，通过设置断点 `b* kern_entry` 跳过OpenSBI阶段，进入OS内核的汇编引导代码。在调试过程中，曾遇到一直卡住没踩中断点的问题，通过 GDB 观察发现 QEMU 版本不兼容。在重装 QEMU 后，问题得到解决，能够正常进行调试。



通过0x80200000处的汇编代码可以知道，进入到内核加载位置，即为可执行代码部分的入口点kern_entry。这是因为链接脚本将 `entry.S` 中的代码段放在内核镜像的最开始位置。

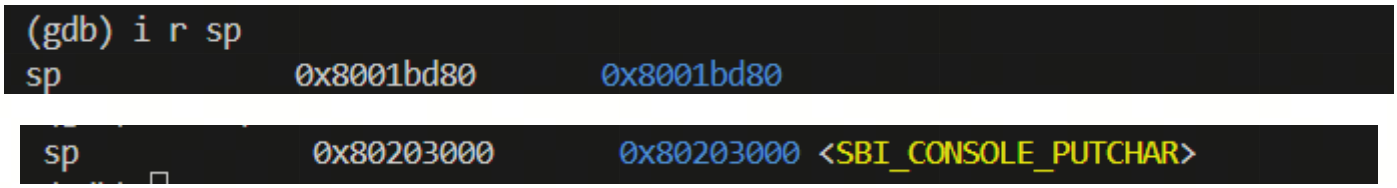
代码块

```
1       .text : {
2           *(.text.kern_entry .text .stub .text.* .gnu.linkonce.t.*)
3       }
4
```

执行两条汇编指令，①分配栈帧空间，修改sp寄存器的值②跳转到kern_init位置执行代码。

可以发现进入OS内核后，多了很多调试信息。

观察sp寄存器值变化：有0x8001bd80变为0x80203000。黄色字体意味着当前栈指针（sp）寄存器里的这个地址 0x80203000正好是全局变量 SBI_CONSOLE_PUTCHAR 在内存中的地址。



5.进入C语言函数 `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
0x80200020 <kern_init+22>       sd      ra,8(sp)
>0x80200022 <kern_init+24>      jal     0x80200490 <memset>
0x80200026 <kern_init+28>       auipc   a1,0x0
0x8020002a <kern_init+32>       addi    a1,a1,1154
0x8020002e <kern_init+36>       auipc   a0,0x0
0x80200032 <kern_init+40>       addi    a0,a0,1178
0x80200036 <kern_init+44>       jal     0x80200054 <cprintf>
0x8020003a <kern_init+48>       j       0x8020003a <kern_init+48>
0x8020003c <cputch>             addi    sp,sp,-32
0x8020003e <cputch+2>           sd      ra,24(sp)
0x80200040 <cputch+4>           sd      a1,8(sp)
0x80200042 <cputch+6>           jal     0x80200088 <cons_putc>
```

`在kern_init函数中主要完成memset和cprint函数的调用，将.bss段用0填充，格式化打印字符串 "(THU.CST) os is loading ...\n"`，最后陷入死循环。

6.问题的答案

通过调式和反汇编，发现RISC-V 硬件加电后最初执行的几条指令位于0x1000到0x100c。它们主要完成了哪些功能是准备进入OpenSBI初始化的参数，读取OpenSBI的开始地址，跳转到OpenSBI 被加载到的物理内存地址 `0x80000000` 。

四、实验中很重要但是没有体现的知识点

1. **文件系统和存储设备的初始化：**
   - 文件系统是操作系统管理存储设备（如硬盘、SSD）上文件和目录的机制。操作系统的启动通常需要从文件系统加载更多的模块或用户程序。文件系统就像是操作系统的仓库目录。系统启动后，内核要从这个“仓库”里取出各种组件（驱动、程序、配置）， 才能继续运行、让用户使用。
   - Lab1中内核在加载和初始化时仅涉及到简单的操作，还没有涉及文件系统的概念。在实际的操作系统中，启动时会识别并挂载根文件系统，以便访问存储设备上的文件。同时也会进行一系列的初始化工作，包括设置内存布局（kern_init有涉及到.bss的处理）、初始化设备驱动、建立中断和异常处理机制等。
2. **虚拟内存管理单元（MMU）的初始化：**
   - **重要性**：现代操作系统普遍采用虚拟内存技术，通过MMU将程序的虚拟地址转换为物理地址，实现内存保护、地址空间隔离和更大的地址空间。MMU的初始化是启用虚拟内存的关键一步。

- **未涉及原因**：Lab1中内核直接运行在物理地址上，没有启用虚拟内存。在更复杂的操作系统中，启动初期会配置页表，启用MMU，将内核自身映射到虚拟地址空间，并为后续的用户进程创建独立的虚拟地址空间做准备。

3. **进程控制块 (PCB) 的完整结构与管理：**

   PCB作为操作系统管理进程的核心数据结构，包括其内容（ID、状态、上下文、内存信息、I/O状态等）以及PCB表的组织方式。Lab1虽然在概念上提到了进程的“存档”和寄存器保存，但并未具体实现或展示PCB的完整结构和管理机制。

4. **虚拟内存与页表：**

   实验中内核是地址相关代码，直接操作物理地址。但现代操作系统普遍采用虚拟内存机制，为每个进程提供独立的虚拟地址空间，并通过页表将虚拟地址映射到物理地址。这解决了地址相关代码的灵活性问题，并提供了内存保护、内存共享等高级功能。实验中对固定物理地址的依赖，正是引入虚拟内存的动机之一。

5. **中断与异常处理：**

   操作系统能够响应外部事件（如 I/O 完成、时钟中断）和内部异常（如缺页、除零），这依赖于完善的中断和异常处理机制。在内核启动后，这些机制是操作系统正常运行、实现多任务和资源管理的基础。实验中 OpenSBI 传递的设备树信息，也暗示了硬件中断的配置。

6. **进程调度：**

   实验仅涉及内核的启动，尚未进入多任务环境。一旦内核启动完成，就需要进程调度器来管理多个进程的执行，决定“何时切换”和“切换谁”。这包括 FCFS、SJF、HRRN、多级反馈队列等经典算法，以及 CFS、PELT、EAS 等现代 Linux 调度器。实验中对 entry.S 和 kern_init 的理解，是后续理解进程概念和调度机制的基础。

7. **并发与同步：**

   在多核环境下，多个 CPU 核心可能同时访问共享资源。为了避免数据竞争和不一致性，操作系统需要提供同步机制（如锁、信号量）。实验中虽然没有直接体现，但 OpenSBI 和内核的启动过程本身就可能涉及对共享资源的访问，为后续理解并发编程和同步机制打下伏笔。

8. **内存分配算法：**

   在内核启动并运行后，它需要动态地分配和回收内存。这涉及到伙伴系统、位图、链表等内存管理数据结构，以及最佳适配、最先适配等分配算法。实验中对栈的静态分配，是更复杂动态内存管理的前奏。

如果内容有理解错误，积极接受指正