比麻雀更小的麻雀（最小可执行内核）

实验目的

- 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
- **多语言支持**: 中英文界面切换

实验过程

环境配置

我们执行 make qemu 命令以验证编译系统和内核镜像的正确性。

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

运行后成功输出了预期信息，这表明我们的 Makefile 配置及内核初始化流程均工作正常。在此基础之上，我们正式进入内核调试阶段。

练习一 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?

指令 `la sp, bootstacktop` 用于在内核启动过程中完成栈指针的初始化

> 其作用是将代码中在 `.data` 段定义的符号 `bootstacktop`（对应内核启动栈的顶部地址）作为有效物理地址加载到栈指针寄存器 `sp` 中。在 RISC-V 架构中，栈遵循向下生长规则，即压栈时栈指针减小，出栈时栈指针增大，该指令的执行为后续内核代码的执行建立了可用的栈空间基础。

目的是为后续函数调用提供合法栈空间

> 在操作系统内核启动初期，硬件处于无栈可用的初始状态——CPU 刚从引导程序跳转至内核入口点，尚未建立任何栈结构。然而，后续即将执行的 C 语言函数必须依赖合法的栈空间来完成关键操作，包括保存返回地址 `ra`、传递函数参数、存储局部变量等。若未预先初始化栈指针，直接进行函数调用将导致非法内存访问或栈溢出等严重错误。因此，通过 `la sp, bootstacktop` 指令显式设置栈指针，正是为后续的函数调用提供必需的合法栈空间，确保内核能够安全地切换到 C 语言执行环境。

指令 tail kern_init作用为尾调用跳转到内核核心初始化函数

> 在 RISC-V 汇编中，`tail` 是一条尾调用指令，其功能是直接跳转到指定的目标函数执行。与常规的 `call` 调用指令不同，`call` 指令会在跳转前自动将当前函数的返回地址保存到 `ra` 寄存器中，并在被调用函数执行结束后通过 `ret` 指令返回到原调用点继续执行后续指令；而 `tail` 指令作为一种优化手段，在跳转时不会保存当前函数的返回地址，这是因为调用者在执行 `tail` 调用后已无任何后续指令需要执行，被调用函数执行完毕后可以直接返回到当前调用者的上一级调用者（例如系统引导程序），从而实现了调用链的简化与栈空间的节省。

目的是从汇编入口过渡到 C 语言初始化

> 在操作系统内核的启动流程中，汇编入口代码（entry.S）承担着硬件级的基础初始化工作——仅执行最必要的底层操作（如初始化栈指针、关闭中断等），这是由于汇编语言更贴近硬件特性，能够直接处理与CPU架构相关的底层细节。而C语言初始化例程（kern_init）则负责实现更复杂的内核逻辑——包括初始化各类内核子系统、设置内存管理机制、加载设备驱动以及创建初始进程等高级功能。 这条`tail kern_init`指令的本质在于完成从汇编到C语言的执行环境过渡——它将内核初始化的控制权从汇编入口点平稳移交至C代码入口。通过采用尾调用优化机制，既避免了无意义的返回地址保存，又有效减少了栈空间的开销，使得被调用的kern_init函数在执行完毕后能够直接返回到更上层的调用者，从而实现执行流程的高效转换。

练习二 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。

Lab1中，将借助 GDB 对 QEMU 模拟的 RISC-V 计算机进行调试，以观察系统从加电启动，直至执行到应用程序第一条指令（即跳转至地址 t0）的完整过程。

在示例代码的 Makefile 中，定义了以下部分内容

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

    .section .text,"ax",%progbits
    .globl kern_entry
```

```
kern_entry:
    la sp, bootstacktop

    tail kern_init

.section .data
    # .align 2^12
    .align PGSHIFT
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:
```

> 以通过 `make debug` 和 `make gdb` 命令对 QEMU 进行调试。具体操作流程： -->在一个终端中执行 `make debug` 命令。该命令会完成内核镜像的生成和加载，并启动 QEMU。同时，QEMU 会打开 GDB 监听端口，并暂停代码运行，等待 GDB 连接，从而为调试提供一个明确的起点。 -->在另一个终端中执行 `make gdb` 命令。该命令会启动 GDB 并连接到 QEMU 的监听端口，建立远程调试会话。 -->GDB 连接成功后，即可开始进行内核调试工作。

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

```
    0x1000:      auipc   t0,0x0     # t0 = pc + (0 << 12) = 0x1000
    0x1004:      addi    a2,t0,40   # a2 = t0 + 40 = 0x1028
    0x1008:      csrr    a0,mhartid # a0 = mhartid
    0x100c:      ld      a1,32(t0)  # a1 = [t0 + 32] = [0x1020]
    0x1010:      ld      t0,24(t0)  # t0 = [t0 + 24] = [0x1018]
    0x1014:      jr      t0         # 跳转到地址 t0
    0x1018:      unimp
    0x101a:      .insn   2, 0x8000
    0x101c:      unimp
    0x101e:      unimp
```

其中在地址为0x1014的指令处会跳转跳转到 0x80000000 执行 OpenSBI 程序，故实际执行的为以下指令

```
    0x1000:      auipc   t0,0x0
    0x1004:      addi    a2,t0,40
    0x1008:      csrr    a0,mhartid
    0x100c:      ld      a1,32(t0)
    0x1010:      ld      t0,24(t0)
    0x1014:      jr      t0
```

输入si单步执行，使用形如info r t0的指令查看涉及到的寄存器结果：

```
(gdb) si
0x0000000000001004 in ?? ()
(gdb) info r t0
```

```
t0             0x1000   4096
(gdb) si
0x0000000000001008 in ?? ()
(gdb) info r t0
t0             0x1000   4096
(gdb) si
0x000000000000100c in ?? ()
(gdb) info r t0
t0             0x1000   4096
(gdb) si
0x0000000000001010 in ?? ()
(gdb) info r t0
t0             0x1000   4096
(gdb) si
0x0000000000001014 in ?? ()
(gdb) info r t0
t0             0x80000000   2147483648
(gdb) si
0x0000000080000000 in ?? ()                 #跳转到地址0x80000000
```

当程序跳转到地址 `0x80000000` 继续执行时，可以通过在 GDB 中输入 `x/10i 0x80000000` 命令来查看该地址处的 10 条指令。此处加载的是作为 bootloader 的 OpenSBI.bin，其主要作用是加载操作系统内核并启动操作系统的执行。

显示的代码如下：

```
(gdb) x/10i 0x80000000
=> 0x80000000:  csrr    a6,mhartid      # 读取当前硬件线程ID到a6寄存器
   0x80000004:  bgtz    a6,0x80000108   # 如果a6 > 0 (非0号核心)，跳转到0x80000108
   0x80000008:  auipc   t0,0x0          # 将当前PC的高20位与0左移12位相加，存入t0
   0x8000000c:  addi    t0,t0,1032      # t0 = t0 + 1032 = 0x80000008 + 1032 =
0x80000410
   0x80000010:  auipc   t1,0x0          # 将当前PC的高20位与0左移12位相加，存入t1
   0x80000014:  addi    t1,t1,-16       # t1 = t1 - 16 = 0x80000010 - 16 =
0x80000000
   0x80000018:  sd      t1,0(t0)        # 将t1的值(0x80000000)存储到t0指向的地址
(0x80000410)
   0x8000001c:  auipc   t0,0x0          # 将当前PC的高20位与0左移12位相加，存入t0
   0x80000020:  addi    t0,t0,1020      # t0 = t0 + 1020 = 0x8000001c + 1020 =
0x80000418
   0x80000024:  ld      t0,0(t0)        # 从t0指向的地址(0x80000418)加载一个双字到t0
(gdb)
```

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，输出如下：

```
(gdb) break kern_entry
Breakpoint 1 at 0x80200000: file kern/init/entry.S, line 7.
```

在操作系统的内核加载过程中，地址 `0x80200000` 是由链接脚本 `kernel.ld` 中定义的 `BASE_ADDRESS`（内核加载基地址）所确定的，而标签 `kern_entry` 则是在同一链接脚本中通过 `ENTRY` 指令显式声明的程序入口点。

该入口点所对应的汇编代码及其功能如下：

- `la sp, bootstacktop`：将符号 `bootstacktop` 所代表的地址加载到栈指针寄存器 `sp` 中，从而为内核建立起初始的栈空间；
- `tail kern_init`：使用 RISC-V 的尾调用指令跳转至 `kern_init` 函数，实现控制流的无缝转移，同时避免不必要的返回地址保存。

在调试器中输入指令 `x/10i 0x80200000`，查看该地址起始处的十条汇编指令，进一步验证入口代码的实际内容。

```
(gdb) x/10i 0x80200000
   0x80200000 <kern_entry>:     auipc   sp,0x3
    # sp = pc + (0x3 << 12) = 0x80200000 + 0x3000 = 0x80203000
   0x80200004 <kern_entry+4>:   mv      sp,sp
    # sp = sp (空操作，实际sp值不变)
   0x80200008 <kern_entry+8>:   j       0x8020000a <kern_init>
    # 跳转到kern_init函数
   0x8020000a <kern_init>:      auipc   a0,0x3
    # a0 = pc + (0x3 << 12) = 0x8020000a + 0x3000 = 0x8020300a
   0x8020000e <kern_init+4>:    addi    a0,a0,-2
    # a0 = a0 - 2 = 0x8020300a - 2 = 0x80203008
   0x80200012 <kern_init+8>:    auipc   a2,0x3
    # a2 = pc + (0x3 << 12) = 0x80200012 + 0x3000 = 0x80203012
   0x80200016 <kern_init+12>:   addi    a2,a2,-10
    # a2 = a2 - 10 = 0x80203012 - 10 = 0x80203008
   0x8020001a <kern_init+16>:   addi    sp,sp,-16
    # sp = sp - 16 = 0x80203000 - 16 = 0x80202ff0
   0x8020001c <kern_init+18>:   li      a1,0
    # a1 = 0 (立即数0加载到a1寄存器)
   0x8020001e <kern_init+20>:   sub     a2,a2,a0
    # a2 = a2 - a0 = 0x80203008 - 0x80203008 = 0
(gdb)
```

> 可以看到在kern_entry之后，紧接着就是kern_init

输入continue执行直到断点，debug输出如下：

```
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: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
```

> 然后使用c继续执行至断点, 此时在左侧看到成功启动了OpenSBI

接着输入指令break kern_init，输出如下：

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

在调试过程中，设置的断点准确地定位到了之前通过反汇编显示的 `<kern_init>` 符号对应的地址 `0x8020000c`。 为了更好地理解执行上下文，有必要了解几个关键寄存器的功能：

- `ra` 寄存器用于保存函数返回地址
- `sp` 寄存器指向当前栈顶位置
- `gp` 寄存器作为全局指针用于优化数据访问而
- `tp` 寄存器作为线程指针用于线程局部存储

当在调试器中输入 `continue` 命令后，程序将继续执行直至遇到断点，随后可以通过 `disassemble kern_init` 命令查看 `kern_init` 函数的完整反汇编代码，从而深入分析该函数的指令级实现细节。

```
0x000000008020000a <+0>:     auipc   a0,0x3
  # a0 = pc + (0x3 << 12) = 0x8020000a + 0x3000 = 0x8020300a
0x000000008020000e <+4>:     addi    a0,a0,-2
  # a0 = a0 - 2 = 0x8020300a - 2 = 0x80203008
0x0000000080200012 <+8>:     auipc   a2,0x3
  # a2 = pc + (0x3 << 12) = 0x80200012 + 0x3000 = 0x80203012
0x0000000080200016 <+12>:    addi    a2,a2,-10
  # a2 = a2 - 10 = 0x80203012 - 10 = 0x80203008
0x000000008020001a <+16>:    addi    sp,sp,-16
  # sp = sp - 16 (在栈上分配16字节空间)
0x000000008020001c <+18>:    li      a1,0
  # a1 = 0 (立即数0)
0x000000008020001e <+20>:    sub     a2,a2,a0
  # a2 = a2 - a0 = 0x80203008 - 0x80203008 = 0
0x0000000080200020 <+22>:    sd      ra,8(sp)
  # 将返回地址ra保存到栈中[sp+8]位置
0x0000000080200022 <+24>:    jal     0x80200490 <memset>
  # 跳转到memset函数, 同时ra = 0x80200026
0x0000000080200026 <+28>:    auipc   a1,0x0
  # a1 = pc + 0 = 0x80200026 + 0 = 0x80200026
```

```
0x000000008020002a <+32>:    addi    a1,a1,1154
  # a1 = a1 + 1154 = 0x80200026 + 1154 = 0x802004a8
0x000000008020002e <+36>:    auipc   a0,0x0
  # a0 = pc + 0 = 0x8020002e + 0 = 0x8020002e
0x0000000080200032 <+40>:    addi    a0,a0,1178
  # a0 = a0 + 1178 = 0x8020002e + 1178 = 0x802004c8
0x0000000080200036 <+44>:    jal     0x80200054 <cprintf>
  # 跳转到cprintf函数，同时ra = 0x8020003a
0x000000008020003a <+48>:    j       0x8020003a <kern_init+48>
  # 无限循环，程序停在此处
```

> 从反汇编代码可以观察到，`kern_init` 函数的最后一条指令是 `j 0x8020003a <kern_init+48>`，这是一个指向自身的无条件跳转指令。这种设计形成了一个无限循环结构，当程序执行到此处时将不断地跳转回当前地址，所以代码在此处永久循环而无法继续向后执行。

输入continue，debug窗口出现以下输出：

```
(THU.CST) os is loading ...
```

练习二问题回答

a. 执行起点 RISC-V 硬件加电后，CPU 的程序计数器（PC）会从固定的复位向量地址 `0x1000` 开始执行。这是 RISC-V virt 机器约定的启动入口。

b. 核心任务与参数准备 位于 `0x1000` 的初始指令序列的核心任务，是为后续的固件初始化函数准备参数并完成跳转：

- **准备参数**：
  - `a0` 寄存器：通过 `csrr a0, mhartid` 指令，读取并存储当前硬件线程的 ID。
  - `a1` 寄存器：从内存地址 `0x1020` 处加载一个配置参数。
  - `a2` 寄存器：被设置为地址 `0x1028`，指向一个包含更多启动参数的结构体。
- **跳转执行**： 指令从地址 `0x1018` 处加载固件核心初始化函数的入口地址到 `t0` 寄存器，随后通过 `jr t0` 指令跳转到该函数。
- **阶段意义** 这一步骤是系统启动流程中的关键一环，它完成了从"硬件复位"到"固件核心初始化"的过渡，为最终将控制权移交给位于更高地址的操作系统内核做好了必要的准备工作。

实验问题解决

环境搭建说明

我们选择了桌面版 Ubuntu 作为开发环境。为确保组件兼容性，我们进行了如下配置：

- 组件版本管理：由于因其他课程需要而安装的 QEMU 7.0 版本较高，其内建的 OpenSBI 与实验要求存在兼容性问题，我们选择并编译了 QEMU 4.1.1 版本。该版本内置了兼容的 OpenSBI

- 环境隔离：为避免影响系统其他部分，我们将 QEMU 4.1.1 安装到独立的目录中

```
yao@LAPTOP-KL96VSNC:/mnt/c/Users/姚文广/Desktop/OS/qemu-4.1.1$ /opt/qemu4.1.1/bin/qemu-system-riscv64 --version
QEMU emulator version 4.1.1
Copyright (c) 2003-2019 Fabrice Bellard and the QEMU Project developers
```

- 项目配置：我们修改了项目的 Makefile 文件，确保其正确指向我们自定义目录下的 QEMU 4.1.1

```
ifndef QEMU
QEMU := /opt/qemu4.1.4/bin/qemu-system-riscv64
endif
```

实验中的重要知识点：

1. 实验中通过链接脚本定义内核代码、数据等段在内存中的分布及加载基地址（如0x80200000），对应OS原理中对内存空间的规划，包括内核区、用户区等划分。
2. 实验中ROM存储上电初始程序，SBI作为固件加载内核，对应OS原理中ROM因非易失性存储启动程序，固件负责硬件初始化和内核加载。差异：实验侧重QEMU中ROM和SBI的地址配置，课程讲述的原理侧重其通用功能和硬件特性。
3. 实验中初始化栈为后续C函数调用提供空间，尾调用实现从汇编到C的过渡，而课程讲述的OS原理中内核启动需建立执行环境，实现不同语言代码的衔接。