DOC

应用程序基本执行环境

```
+-------------------------+
|       Application       |
+-------------------------+
  ^
  | function call
  |
+-------------------------+
|         std lib         |
+-------------------------+
  ^
  | syscall
  |
+-------------------------+
| Operating System/kernel |
+-------------------------+
  ^
  | ISA
  |
+-------------------------+
|    Hardware Platform    |
+-------------------------+


# graph-easy <<< 'graph{flow:north}[Hardware Platform] - ISA -> [Operating System/kernel]- syscall -> [std lib] - function call -> [Application]'
```

通用的操作系统中的执行环境栈，下层是上层的执行环境，保证上层程序正常运行。相邻上下层间通过接口完成调用。

其中函数标准库和操作系统/内核都是对下层资源的**抽象**。

> 对于用某种编程语言实现的应用程序源代码而言，编译器在将其通过编译、链接得到可执行文件的时候需要知道程序要在哪个**平台** (Platform) 上运行。这里平台主要是指 CPU 类型、操作系统类型和标准运行时库的组合。可以看出：
>
> 1. 如果用户态基于的内核不同，会导致系统调用接口不同或者语义不一致；
> 2. 如果底层硬件不同，对于硬件资源的访问方式会有差异。特别是如果 ISA 不同，则向软件提供的指令集和寄存器都不同。
>
> 它们都会导致最终生成的可执行文件有很大不同。需要指出的是，某些编译器支持同一份源代码无需修改就可编译到多个不同的目标平台并在上面运行。这种情况下，源代码是 **跨平台** 的。而另一些编译器则已经预设好了一个固定的目标平台。

在qemu上的启动流程

```
# BOARD
```

```
BOARD       ?= qemu
SBI         ?= rustsbi
BOOTLOADER  := ./bootloader/rustsbi-qemu.bin

QEMU = qemu-system-riscv64
QEMUOPTS = \
    -nographic \ # 设置为没有图形界面
    -machine virt \ # Hardware Platform
    -bios $(BOOTLOADER) \
    -kernel build/kernel    \

run: build/kernel
    $(QEMU) $(QEMUOPTS)

```

设置qemu模拟 `virt` 硬件平台，所以物理内存的起始物理地址为 `0x80000000` ~~死规定~~。上面代码中:

```
BOOTLOADER  := ./bootloader/rustsbi-qemu.bin
QEMUOPTS = \
    -bios $(BOOTLOADER)
```

就是将作为bootloader的 `rustsbi-qemu.bin` 加载到物理地址 `0x80000000` 处。

这节有一些待定/未定的代码，之后再完成这里的文档

程序的内存布局和编译流程

内存布局

当程序被编译为可执行文件后，该可执行文件的内容可以分为**代码**和**数据**2个部分。CPU执行代码部分，并将数据部分作为可读写的内存空间。但是现实中，往往进一步对这两个部分进行进一步的划分，我们称这些更小的单位为 `段` （`Section`）。

将不同 `段` 按一定的"约定"分配放置在内存的位置上，这就形成了**内存布局**。

```
            # high address
+----------+
|  stack   | # 栈
+----------+
|   ...    |
+----------+
|  heap    | # 堆
+----------+
|  .bss    | # 未初始化数据段
+----------+
|  .data   | # 可修改的全局数据
+----------+
| .rodata  | # 只读的全局数据
+----------+
|  .text   | # Code Memory 其他都属于Data Memory
+----------+
            # low address
```

```
# graph-easy <<< '[stack||...||heap||.bss||.data||.rodata||.text] '
```

编译流程

```
+-------------------------+
| application source code |
+-------------------------+
  |
  | compiler
  v
+-------------------------+
|        assembler        |
+-------------------------+
  |
  | assembler
  v
+-------------------------+
|       object code       |
+-------------------------+
  |
  | linker
  v
+-------------------------+
|       executables       |
+-------------------------+

# graph-easy <<< 'graph{flow:south}[application source code] - compiler ->
[assembler] - assembler -> [object code] - linker -> [executables]'
```

链接器

```
/*调整内存布局，链接脚本调整链接器的行为*/
OUTPUT_ARCH(riscv)
ENTRY(_entry)       /*程序入口点*/
BASE_ADDRESS = 0x80200000;      /*内核初始化代码放置地址*/

/*.text, .rodata .data, .bss 从低地址到高地址按顺序放置*/
SECTIONS
{
    . = BASE_ADDRESS;  /*对.进行赋值来调整接下来的段放在哪里*/
    skernel = .;       /*记录当前地址*/

    s_text = .;        // 存放代码段开始的地址
    .text : {          /*花括号内按照放置顺序描述将所有输入目标文件的哪些段放在这个段中
*/
        *(.text.entry)
        *(.text .text.*)/*指定名称以“.text”开头的所有符号。包含在“.text”部分*/
        . = ALIGN(0x1000);/*对齐*/
        *(trampsec)
        . = ALIGN(0x1000);
    }
    . = ALIGN(4K);
    e_text = .;         // 存放代码段结束的地址
```

```
22
23      s_rodata = .;
24      .rodata : {
25          *(.rodata .rodata.*)
26      }
27
28      . = ALIGN(4K);
29      e_rodata = .;
30
31      s_data = .;
32      .data : {
33          *(.data.apps)
34          *(.data .data.*)
35          *(.sdata .sdata.*)
36      }
37      . = ALIGN(4K);
38      e_data = .;
39
40      .bss : {
41          *(.bss.stack)
42          s_bss = .;
43          *(.bss .bss.*)
44          *(.sbss .sbss.*)
45      }
46      . = ALIGN(4K);
47      e_bss = .;
48      ekernel = .;
49
50      /DISCARD/ : {
51          *(.eh_frame)
52      }
53  }
```

> 冒号前面表示最终生成的可执行文件的一个段的名字，花括号内按照放置顺序描述将所有输入目标文件的哪些段放在这个段中，每一行格式为 `<ObjectFile>(SectionName)`，表示目标文件 `ObjectFile` 的名为 `SectionName` 的段需要被放进去。我们也可以使用通配符来书写 `<ObjectFile>` 和 `<SectionName>` 分别表示可能的输入目标文件和段名。

问：`*(.text .text.*)`第一个星号是什么? 括号中第一个`.text` 和 第二个 `.text.*`又各自代表什么?

- 第一个星号 `*` 表示所有要链接的目标文件。
- 括号中的第一个`.text`表示链接器将所有名称为`.text`的段放置在`.text`代码段中。通常，代码段包括程序的指令、函数、只读常量等内容。
- 括号中的第二个`.text.*`表示链接器将名称以`.text.`开头的所有段都放置在`.text`段中，例如`.text.init`，`.text.hot`等类型的段。

下面是链接的例子：

Before

```
1  +-----------+ # 0x0
2  |  .text.1  |
3  +-----------+ # 0x2000
4  | .rodata.1 |
```

```
+-----------+ # 0x5000
|  .data.1  |
+-----------+ # 0x7000
+-----------+ # 0x0
|  .text.2  |
+-----------+ # 0x1000
| .rodata.2 |
+-----------+ # 0x3000
|  .data.2  |
+-----------+ # 0x6000

# graph-easy <<< '[.text.1||.rodata.1||.data.1]
[.text.2||.rodata.2||.data.2]'
```

After

```

+-----------+ # 0x0
|  .text.1  |
+-----------+ # 0x2000
|  .text.2  |
+-----------+ # 0x3000
| .rodata.1 |
+-----------+ # 0x6000
| .rodata.2 |
+-----------+ # 0x8000
|  .data.1  |
+-----------+ # 0xa000
|  .data.2  |
+-----------+ # 0xd000

# graph-easy <<<
'[.text.1||.text.2||.rodata.1||.rodata.2||.data.1||.data.2]'

                           +-----------+     +-----------------+
                           |  .text.1  | --> | sections from 1 | <+
                           +-----------+     +-----------------+  |
    +-----------------+    |           |       ^                  |
  +> | sections from 2 | <- |  .text.2  |       |                  |
  |  +-----------------+    +-----------+       |                  |
  |    ^                    |           |       |                  |
  |    |                    | .rodata.1 | ------+------------------+
  |    |                    +-----------+       |
  |    |                    |           |       |
  +----+------------------- | .rodata.2 |       |
       |                    +-----------+       |
       |                    |           |       |
       |                    |  .data.1  | ------+
       |                    +-----------+
       |                    |           |
       +------------------- |  .data.2  |
                            +-----------+


```

```
# echo "graph{flow:east}
[.text.1||.text.2||.rodata.1||.rodata.2||.data.1||.data.2]
[.text.1.text.2.rodata.1.rodata.2.data.1.data.2.0] -> [sections from 1]
[.text.1.text.2.rodata.1.rodata.2.data.1.data.2.1] -> [sections from 2]
[.text.1.text.2.rodata.1.rodata.2.data.1.data.2.2] -> [sections from 1]
[.text.1.text.2.rodata.1.rodata.2.data.1.data.2.3] -> [sections from 2]
[.text.1.text.2.rodata.1.rodata.2.data.1.data.2.4] -> [sections from 1]
[.text.1.text.2.rodata.1.rodata.2.data.1.data.2.5] -> [sections from 2]" >
graph.txt
# graph-easy ./graph.txt

```

系统的链接脚本(kernel.ld)决定了elf程序的内存地址的虚拟映射，**程序中的绝对地址在链接时确定**。我们需要把系统放置在物理地址0x80200000处，需要进行这步操作的原因是在这个运行时刚进入S态，并没有激活虚存机制。~~我还不知道为什么是这个地址~~。

entry.s

```
# os/entry.s
    .section .text.entry #该 段为内核入口点，放在段最低地址; 在下一个新的部分结束，即.section .bss.stack
    .globl _entry # 声明全局符号_entry
_entry: # 声明入口点的开始
    la sp, boot_stack_top # (sp) = (boot_stack_top)
    call main # 调用main函数，进入Cの世界 ^_^

    .section .bss.stack # 堆栈未初始化数据段
    .globl boot_stack
boot_stack:
    .space 4096 * 16 # 2^16B == 4KB * 16 == 64KB
                     # 定义堆栈空间大小需要满足大小是4KB的整数倍。
    .globl boot_stack_top
boot_stack_top:

```

> 这是一个汇编语言源文件，可以作为可执行程序的启动代码。下面是逐行解释：
>
> 1. `.section .text.entry`: 这行代码表示将下面的代码放置在名为 ".text.entry" 的代码段中。
> 2. `.globl _entry`: 这行代码定义了一个全局符号 "_entry"，该符号是程序的入口点，在程序启动时执行。
> 3. `_entry:`: 这行代码是 _entry 符号的标签，代表程序的入口点。
> 4. `la sp, boot_stack_top`: 这行代码使用 la 命令生成代码，将 boot_stack_top 符号的地址加载到 sp 寄存器中，将栈指针设置为堆栈的顶部。
> 5. `call main`: 这行代码通过 call 指令调用名为 "main" 的函数。
> 6. `.section .bss.stack`: 这行代码将下面的代码放置在名为 ".bss.stack" 的未初始化数据段中。
> 7. `.globl boot_stack`: 这行代码定义了一个全局符号 "boot_stack"，用于表示堆栈的起始位置。
> 8. `boot_stack:`: 这行代码是 boot_stack 符号的标签，表示变量存储在此处。
> 9. `.space 4096 * 16`: 这行代码使用 space 命令在内存中分配 4096 * 16 字节的空间。这是堆栈的空间大小。

> 10. `.globl boot_stack_top`: 这行代码定义了一个全局符号 "boot_stack_top"，该符号用于表示堆栈的顶部位置，即栈指针初始值。
> 11. `boot_stack_top:`: 这行代码是 boot_stack_top 符号的标签，表示指针存储在此处。

小知识点

- `.globl`是什么?

`.globl`指示符表示一个全局符号/标签；例：`.globl _entry`: 这行代码定义了一个全局符号 "_entry"

- 以下代码中`la`是什么意思?

```
1 la sp, boot_stack_top # (sp) = (boot_stack_top)
```

"la"是RISC-V指令集中的一个伪指令(pseudo-instruction），它的全称为“load address”，通常用于将地址加载到寄存器中。

下面是"la"伪指令的语法格式：

```
1 la rd, symbol
```

其中，"rd"表示目标寄存器，"symbol"表示要加载的地址符号。

例如，下面的代码将符号"array"的地址加载到寄存器x1中：

```
1 la x1, array
```

当汇编器处理这个伪指令时，它将使用类似于以下实际指令的指令序列来替代它：

```
1 auipc x1, %pcrel_hi(array)
2 addi x1, x1, %pcrel_lo(array)
```

这个指令序列使用了两个新的伪指令：%pcrel_hi和%pcrel_lo。这两个伪指令用于计算符号的高位和低位偏移，在这个例子中，将符号"array"的地址作为立即数加载到寄存器x1中。