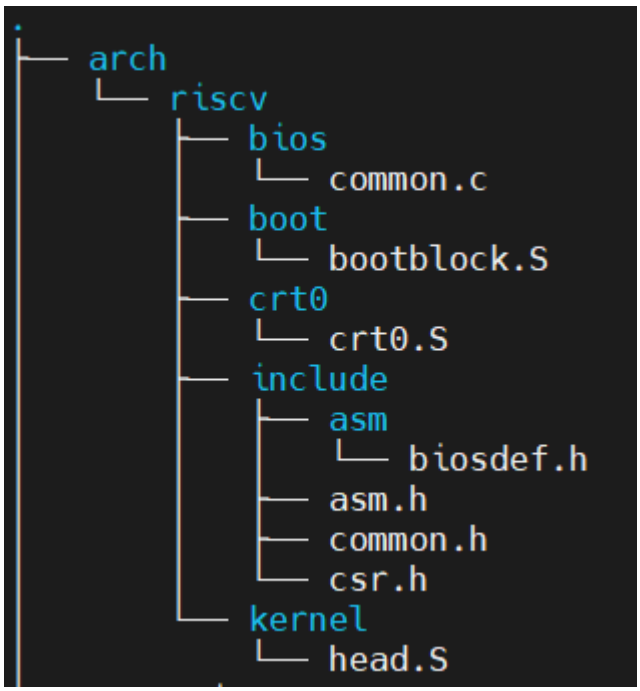
Design review

1.解释P1中的代码结构



arch文件夹

1. riscv/bios 下的common.c

```
static long call_bios(long which, long arg0,long arg1, long arg2, long arg3, lon
void port_write_ch(char ch)
void port_write(char *str)
int port_read_ch(void)
int sd_read(unsigned mem_address, unsigned num_of_blocks, unsigned block_id);
```

定义了一个总的 call_bios, 通过 bios 进行字符和字符串的打印，以及字符和block块的读。

2. boot下的bootblock.S

```
.equ os_size_loc, 0x502001fc
.equ kernel, 0x50201000
.equ bios_func_entry, 0x50150000
```

Bootblock (即 bootloader 相关的代码），定义了操作系统大小放置的位置，

kernel放置的位置，以及 bios函数的入口位置。

在这个函数中，完成BIOS的一些功能：打印字符，读取SD卡中的kernel,加载 task相关参数，并传递给kernel，最后跳转至kernel

3. crt0下的 crt0.S

设置用户程序的运行时环境，进入main 函数执行，执行完之后返回到 kernel

4. asm/biosdef.h

```
#define BIOS_PUTCHAR 1
#define BIOS_GETCHAR 2
#define BIOS_PUTSTR 9
#define BIOS_SDREAD 11
```

宏定义，传递到 riscv/bios 的 call_bios等的第一个参数 (long which),与串口读写相关的函数编号

5. Include 下的 asm.h/common.h/csr.h

asm.h: 设置汇编宏定义

common.h

```
void port_write_ch(char ch);
void port_write(char *buf);
int port_read_ch(void)
int sd_read(unsigned mem_address, unsigned num_of_blocks, unsigned block_id);
```

串口功能的打印，和 riscv/bios 中的函数是同一个，都是利用 bios 的函数调用来实现的(call_bios)

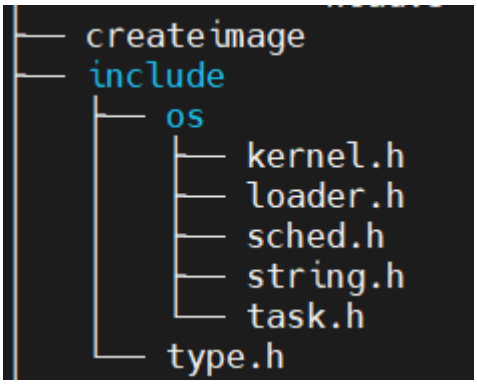
csr.h: 宏定义，设置状态寄存器标志，CSR寄存器，SIE(中断打开),SIP(中断挂起) 标志位

6. kernel/head.S

设置内核的初始地址

```
.section ".entry_function", "ax"
            csrw CSR_SIE, zero
            csrw CSR_SIP, zero
```

内核代码开始运行，关闭中断，需要清空BSS段，并且设置C的运行时环境



Createimage

把多个 elf 文件拼起来，变成镜像的工具

include文件夹

1. os/kernel.h

```
#define KERNEL_JMPTABLE_BASE 0x51ffff00
typedef enum{
    CONSOLE_PUTSTR,
    ...
}jmptab_idx_t;

static inline long call_jmptab(long which, long arg0, long arg1, long arg2,
long arg3,long arg4);

static inline void bios_putstr(char *str)
static inline void bios_putchar(int ch)
static inline int bios_getchar(void)
static inline int bios_sd_read(unsigned mem_address, unsigned num_of_blocks, \
                unsigned block_id);
```

所有 bios 相关的读写都通过 call_jmptab 来实现，比 port_write 等多了一层封装。

在 os中，希望使用串口的读写功能，则直接调用 os/kernel.h 里定义的函数，这些函数利用 jump table 来实现，记录了这些 bios 函数的入口地址

2. os/loader.h

```
uint64_t load_task_img(int taskid);
```

加载 task 信息，可以通过 taskid 加载 app

3. os/sched.h

定义寄存器的上下文信息，switchto时保存寄存器的信息，task的四种状态（blocked, running, ready, exited）,进程的pcb，ready队列，sleep队列等。

定义进程调度相关的函数，切换/调度/睡眠/阻塞/取消阻塞

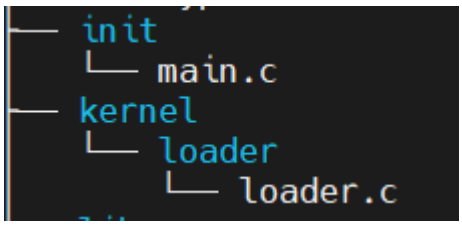
4. os/string.h os/task.h

string.h: 字符串相关操作

task.h: 定义 task_info, app 调度时自己的 task_info_t

5. type.h

定义相关的字节、字、双字等的最大最小值



Init 和 kernel

1. init/main.c

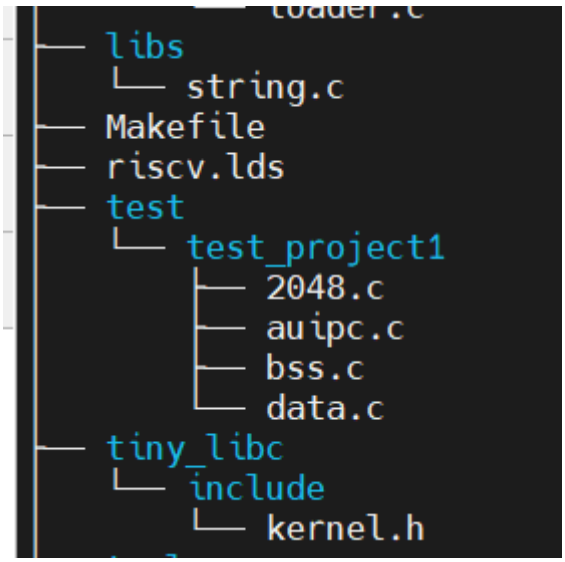
进入 kernel 之后，进行 bss 是否清空的检查， jump table 是否初始化的检查，task_info 是否初始化的检查

之后打印 Hello OS, 完成对 task 的加载后，执行 while(1){等待中断发生}

2. kernel/loader/loader.c

```
1 uint64_t load_task_img(int taskid)
```

负责根据 task id 或者 task name 加载 task, 并返回 task 入口地址



test文件夹

1. libs/string.c

实现string.h 里的字符串操作

2. Makefile

通过 make 执行后续的编译、链接、运行、gdb

3. riscv.lds

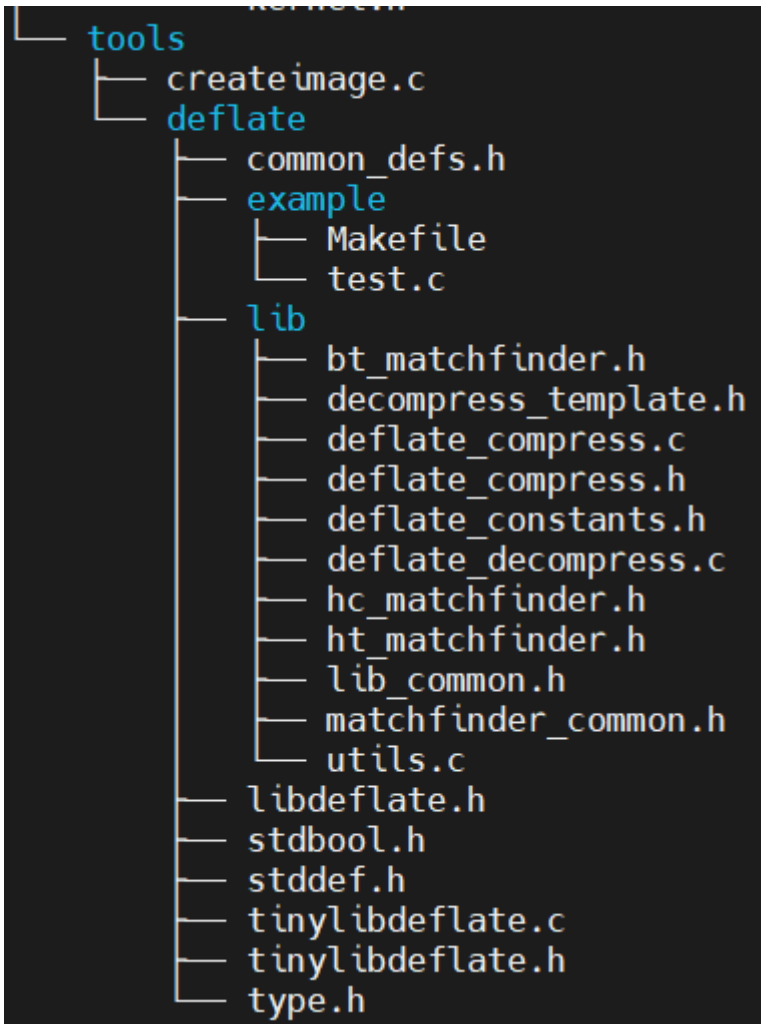
Linker script: 链接器脚本，链接器把文件链接成目标的执行文件

4. test文件夹

进行 bss 段的检测和 data 的输入输出检测，运行一个 2048 游戏

5. kernel.h

和 os/kernel.h 中的文件一致



1. createimage.c

完成读取 elf header, program header, 获得入口地址，获得file size, memory size,

写 segment, 写 padding, 写 image info 等功能，完成创建镜像的功能

需要添加自定义的 task_info 的初始化功能

2. tool

实现对文件的压缩和解压缩

2.你的kernel如何加载 app, 并令它运行

根据内存布局，bootblock在镜像的第一个扇区，kernel在镜像的第二个扇区开始，规定都占15个扇区。

由于 task id 和 image 中第几个用户程序一一对应，所以假设需要加载的 task app 为

x, 则实际其所占的扇区号为:

1 [2 - 16] [17 - 31]

2 + （x-1)*15

X * 15 + 1

起始扇区: 2 + (x-1)*15, 结束扇区: x * 15 + 1

根据扇区大小固定为 512B,计算出 App 的入口地址。

1. 通过 BIOS 的BBL提供的与底层硬件相关的API，调用 BIOS API,

BIOS的各个函数的入口地址在 0x50150000, bios_func_entry

汇编语言层面调用

2. 使用该函数

```
1 uintptr_t bios_sdread(unsigned mem_address, unsigned num_of_blocks,
2 unsigned block_id)
3 // 读取第 x 个 App, 从SD卡的第 block_id = 2 + (x - 1) * 15个扇区开始,
4 // 读取 15 个扇区，放入内存 0x520 (X+1)0000 处
```

3. 特别地(没有任务3设定的15个扇区)，扇区的数据需要通过读头一个扇区的倒数第4个字节的位置

需要 lh a0, 0x502001fc 等

lh a0, 0x50200(X+1)fc 等读取扇区数目

后面完成 write_img_info() 时，能够读取用户程序的数目，以及kernel所占的扇区数，在上电后读取。

4. 进行初始化操作，从 0x50500000处设置栈地址空间，清空bss段

3.你的设计中，task_info_t 的结构是什么? 如何初始化 task_info_t ?

```c
typedef struct {
    uint64_t task_id;
    uint64_t task_begin_sector;    // task 起始扇区
    uint64_t task_size;           // task 占据的扇区数
    char *task_name;             // task name
}task_info_t;
```

从 write_img_info() 里获取 task_begin_sector 和 task_size

task name 为 *files

4.解释你设计的 image 文件中的内容和它们的offset

Image 文件中的内容:

```
[        Bootblock (1 个扇区)       ]
[        Kernel    (15 个扇区)       ]
[        App Volume                  ] }
[        App Volume                  ] }
[        App Volume                  ] }
[        App Info                    ]
```

设置 App Volume，每个 App Volume 的前8个字节，4个字节用来存储起始地址，4个地址用来存储 App 的大小。

总结

image是存放在SD卡里的，紧密排列最好。

在内存里，每个 App 对应的内存地址是定好的，按照对应地址进行放置。

Objdump -S 可以查看反汇编的代码

a0寄存器通常用于返回值，a0寄存器开始也用于函数传递的参数。

task_info_t 构成了一个全局数组，放在bss段(初始化为0的全局数组当中)