FAQ

p1的代码从写完到运行的完整流程是什么?

完整流程为：

1. 执行make命令编译，编译成功的效果类似于：

```
stu@stu:~/OSLab-RISC-V/p1$ make
riscv64-unknown-linux-gnu-gcc -O2  -fno-builtin -nostdlib -T riscv.lds -
Iinclude -Wall -mcmodel=medany -o bootblock bootblock.S -e main -
Ttext=0x50200000
riscv64-unknown-linux-gnu-gcc -O2  -fno-builtin -nostdlib -T riscv.lds -
Iinclude -Wall -mcmodel=medany -o kernel kernel.c head.S -Ttext=0x50201000
./createimage --extended bootblock kernel
0x50200000: bootblock
        segment 0
                offset 0x1000           vaddr 0x50200000
                filesz 0x0069           memsz 0x0069
                writing 0x0069 bytes
                padding up to 0x0200
0x50201000: kernel
        segment 0
                offset 0x1000           vaddr 0x50201000
                filesz 0x0194           memsz 0x01d0
                writing 0x01d0 bytes
                padding up to 0x0400
        segment 1
                offset 0x0000           vaddr 0x0000
                filesz 0x0000           memsz 0x0000
os_size: 1 sectors
```

2. 编辑 `run_qemu.sh` 脚本，设置正确的镜像位置，例如下面的设置：

```
#!/bin/bash
IMG_PATH=p1/image
sudo /home/stu/OSLab-RISC-V/qemu-4.1.1/riscv64-softmmu/qemu-system-riscv64 -
nographic -machine virt -m 256M -kernel /home/stu/OSLab-RISC-V/u-boot/u-boot
-drive if=none,format=raw,id=image,file=${IMG_PATH} -device virtio-blk-
device,drive=image -smp 2 -s
```

如果需要用gdb调试，想停在第一条语句处，也可以设置上`-S`（大写S），如果只想执行一下看看效果，或者不想在第一条语句就停下来，可以把`-S`（大写S）去掉。

3. 执行`run_qemu.sh`，显示出可以输入的命令行后，输入`loadboot`。

```
stu@stu:~/OSLab-RISC-V$ ./run_qemu.sh
[sudo] password for stu:
qemu-system-riscv64: warning: No -bios option specified. Not loading a
firmware.
qemu-system-riscv64: warning: This default will change in a future QEMU
release. Please use the -bios option to avoid breakages when this happens.
qemu-system-riscv64: warning: See QEMU's deprecation documentation for
details.


U-Boot 2019.07 UCAS_OS v2.0 (Aug 24 2021 - 07:56:20 +0000)

CPU:   rv64imafdcsu
Model: riscv-virtio,qemu
DRAM:  256 MiB
In:    uart@60000000
Out:   uart@60000000
Err:   uart@60000000
Net:   No ethernet found.
Hit any key to stop autoboot:  0

Device 0: QEMU VirtIO Block Device
            Type: Hard Disk
            Capacity: 0.0 MB = 0.0 GB (2 x 512)
... is now current device
** Invalid partition 1 **
No ethernet found.
No ethernet found.

virtio read: device 0 block # 0, count 2 ... 2 blocks read: OK
=> loadboot

Loading UCAS OS...

Hello OS!
bss check: t version: 1
```

看到效果后，按`Ctrl+a`之后按x，退出QEMU

4. 之后编辑Makefile，设置正确的设备。方法为：先在命令行中敲`ls /dev/sd*`，一般应该只能看到一个`/dev/sda`。插入SD卡和读卡器，在虚拟机上的USB控制器选项上勾选上Generic Mass Storage Device（这个就是SD卡），然后再次尝试`ls /dev/sd*`，此时应该能额外看到`/dev/sdb`。Linux对磁盘设备的编号一般是`sda`、`sdb`、`sdc`这样往上增长的，类似于windows的C、D、E盘这种感觉（这个类比不太精确，大概意会一下就行）。一般我们的虚拟机都

是`/dev/sdb`，我们发的start code应该是有意设置成了`/dev/sdc`，这是为了避免有些同学用自己的机器上的Linux，意外破坏自己的分区。

```
CFLAGS = -O2  -fno-builtin -nostdlib -T riscv.lds -Iinclude -Wall -
mcmodel=medany
DISK = /dev/sdb # 这里修改成正确的设备名

BOOTLOADER_ENTRYPOINT = 0x50200000
KERNEL_ENTRYPOINT = 0x50201000
```

5. 执行`make floppy`，写入磁盘。

```
stu@stu:~/OSLab-RISC-V/p1$ make floppy
sudo fdisk -l /dev/sdb
Disk /dev/sdb: 14.44 GiB, 15489564672 bytes, 30253056 sectors
Disk model: Storage Device
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x18c35dfa

Device     Boot Start      End  Sectors  Size Id Type
/dev/sdb1        2048    71679    69632   34M  c W95 FAT32 (LBA)
/dev/sdb2       71680 30253055 30181376 14.4G  c W95 FAT32 (LBA)
sudo dd if=image of=/dev/sdb2 conv=notrunc
2+0 records in
2+0 records out
1024 bytes (1.0 kB, 1.0 KiB) copied, 0.00291 s, 352 kB/s
```

6. 将SD卡拔出，插入到板卡上。在虚拟机的USB控制器选项上勾选Xilinx TUL，这个是我们的板卡。然后打开开关，之后执行`sudo minicom`。按板卡上的srst键重置板卡，应该就可以看到Nutshell的logo了。此时应该可以输入`loadboot`命令（可以按tab键补齐）。loadboot之后应该就能看到和qemu上相似的现象。
7. 在确认一个小阶段的目标完成后，提交到git并推送到gitlab上。

```
stu@stu:~/OSLab-RISC-V/p1$ git commit -a -m "finish XXXXXX"
# 这里的输出根据你提交的内容不同是不同的
[main ec7e6c6] update
 1 file changed, 1 insertion(+)
# 以下假定你之前是从远程git clone下来的
stu@stu:~/OSLab-RISC-V/p1$ git push
# 以下也是样例输出，具体情况不一定一样
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 6 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 268 bytes | 268.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To ssh://gitlab.agileserve.org.cn:8082/wangluming/oslab.git
   2293f43..ec7e6c6  main -> main
```

minicom如何退出?

minicom在连接到板卡的情况下，需要按Ctrl+a，然后单独按一个x，就会显示是否要退出minicom的对话框，选择是然后回车即可退出。

每次连接板卡都要重新设置虚拟机的USB选项，有什么简单方法吗?

在虚拟机的设置中，选择USB设备的选项卡，能够看到一个USB设备筛选器，点击添加一个USB设备筛选器。将板卡添加进去（板卡需要是已开电连接的状态）。如果需要，也可以把SD卡读卡器也添加进去。

命令行提示no left space一类的错误怎么处理?

部分安装了图形用户界面的同学会偶尔遇到这个问题。可以在命令行中输入如下命令确认：

```
df -h
```

如果观察到`/`目录已经用了100%了说明磁盘空间被全部占用了。占用以后，可以观察一下到底是哪个文件夹占用了过大的空间：

```
for i in `ls -a`
do
    du -s -h $i
done
```

目前遇到的一些同学是`~/.local/Trash`异常大，得有40多个G。验证方法是：

```
du -s -h ~/.local/Trash
```

能够看到该文件夹特别大。清理方法为，将其中的`files`和`info`两个子目录下的文件全删掉。

```
cd ~/.local/Trash
cd files
rm -rf *
cd ../info
rm -rf *
```

执行./run_qemu.sh或createimage等程序时显示Permission denied怎么办?

现象是为执行`./run_qemu.sh`或者make的时候会由make去执行./createimage导致报错：

```
stu@stu:~/OSLab-RISC-V/p0$ ./run_qemu.sh
-bash: ./run_qemu.sh: Permission denied
```

这个一般是由于没有设置可执行权限。有些将程序传入到虚拟机的方法可能无法把可执行权限设置上。需要自行设置可执行权限。设置命令为`chmod +x 可执行文件名`。例如`chmod +x ./createimage`或`chmod +x ./run_qemu.sh`

总之，基本上遇到要执行某个程序或者脚本，但是报Permission denied的错误，都可以通过用`chmod +x <可执行文件名>`增加可执行权限来解决。

make的时候在执行createimage时报Segmentation fault或者Falt error: glibc detected an invalid stdio handle等错误怎么办?

我们给的start code的Makefile是按照跑完整的三个任务写好的。但是在刚开始做第一个任务时需要做一些处理。第三个任务要求自己编写createimage.c，所以Makefile里面提供了构建createimage这个目标的相关指令。

直接执行make的话，会自动尝试构建`all`这个目标。构建`all`依赖于`createimage`和`image`，所以接下来make会首先尝试构建这两个目标。构建`createimage`目标时会根据写好的指令将`createimage.c`文件编译为createimage这个可执行文件。构建`image`则会先构建`bootblock`和`kernel`这两个可执行文件，然后再执行`./createimage --extended bootblock kernel`命令构建出image这个镜像文件。

在任务一时，需要用我们提供好的createimage。此时因为自己的createimage.c还是没写好的，所以不能让make去编译本地的createimage.c。否则编译出来的createimage会覆盖掉我们提供的createimage。后续构建image的时候执行`./createimage`指令就会出现Segmentation fault等各种奇怪的错误。因此，我们需要把all后面的createimage这个目标临时去掉。

```
all: createimage image
# 任务一应该去掉createimage这个目标，改为：
# all: image
# ！！！任务三的时候一定记得改回来！！！！！

bootblock: bootblock.S riscv.lds
        ${CC} ${CFLAGS} -o bootblock bootblock.S -e main -
Ttext=${BOOTLOADER_ENTRYPOINT}

kernel: kernel.c head.S riscv.lds
        ${CC} ${CFLAGS} -o kernel kernel.c head.S -Ttext=${KERNEL_ENTRYPOINT}

createimage: createimage.c
        ${HOST_CC} createimage.c -o createimage -ggdb -Wall
```

```
image: bootblock kernel createimage
	./createimage --extended bootblock kernel
```

另外请务必注意，**任务一去掉createimage这个目标后，做到任务三的时候一定记得改回来！！！**否则就一直用的是我们提供的createimage了。

另外，也请注意，`make clean`会删除createimage可执行文件，大家做任务一的时候可以自行根据情况处理一下。比如不要做make clean，或者修改clean这个目标使其不要删除createimage。

```
clean:
	rm -rf bootblock image kernel *.o createimage
# 可以将rm命令临时改为：rm -rf bootblock image kernel *.o
```

获取sbi_console_getchar后直接打印，输出很奇怪？

sbi_console_getchar()会立即检测键盘输入，如果此时键盘没有任何键被按下会返回-1。如果有某个键被按下则返回对应的ascii码。因此，在做键盘输入相关的动作时需要自己想办法处理掉-1的情况。避免将-1当作真正的输入直接使用。屏幕上看到很奇怪的输出很有可能是打印了`(char)-1`这个字符。

输入按键输入不进去（bbl那里都输入不进去）？

可能是minicom在设置时，hardware flow control没有选择为关闭。这里必须关闭才能输入进去。

RISC-V内存的哪些地址可以写？有没有哪些地方有重要的东西？

RISC-V内存的范围是0x50000000-0x60000000。其中，0x50000000-0x50200000的部分是BBL或者U-Boot使用的，如果覆盖掉会导致SBI调用出现问题。0x5f000000后面的一些地址被BBL或U-Boot用来放置一些数据或实现一些SBI相关的功能，建议也不要使用这段内存。中间的部分都没有被使用，可以供你编写的小操作系统自行管理使用。

在板子上将bootloader拷贝到其他地址，跳转过去以后执行会出现奇怪的问题。可能是什么原因？

板子上的RISC-V处理器是有L1 I-Cache的，但拷贝是通过L1 D-Cache进行的，所以有可能两个cache里的内容没同步。所以需要用fence.i指令将I-Cache彻底刷掉，让处理器从L2 Cache重新拉取数据，这样就能解决这种一致性问题。QEMU上由于没有模拟cache，所以应该是不会出现类似现象的。

刷I-Cache的这个指令不需要任何参数，直接写就行。

```
fence.i
```

bootblock.S卡死/循环输出/输出不了第二个字符串等可能是什么原因？

很多和bootblock.S相关的错误都和gp寄存器有关。bootblock.S中，编译器可能会生成使用了gp寄存器的代码。如果编译器生成了这类代码，那么需要在bootblock.S中初始化gp寄存器。否则可能造成bootblock.S运行时卡死。

判断编译器是否生成了gp寄存器相关指令的方法为，编译后在命令行中执行下面的代码对bootblock进行反汇编

```
riscv64-unknown-linux-gnu-objdump -d bootblock
```

这样就能看到bootblock的汇编代码，如果这里有使用到gp寄存器，那么就需要初始化gp寄存器。初始化的代码与head.S中我们给出的相同：

```
/* Load the global pointer */
.option push
.option norelax
la gp, __global_pointer$
.option pop
```

在使用gp寄存器之前进行初始化即可。

初始化bss到底该怎么做？

在head.S中初始化bss段的思路：riscv.lds中存在**bss_start和**BSS_END_*两个符号。标志着bss段的起始和结束。（相信聪明的同学们一看riscv.lds就知道为什么^^）*。在汇编或C代码中，lds里面定义的符号是可以直接引用的。但是请注意，只有这个符号的地址是有意义的（因为这个地址并没有存什么有意义的值）。所以在汇编中，你需要用la来加载它的地址到寄存器。

OS代码出错后看不出来问题在哪儿该怎么办？

请参照预备课的内容，用riscv的gdb连接到QEMU上，通过单步执行、打印变量等调试手段，对你的OS代码进行调试。在Project 1中，这种方法应该是足以应对可能出现的问题的。如果板子上有问题，但QEMU没问题，可能是由于QEMU会自动将所有寄存器和内存清零。所以可以从初始化等方面进行考虑。

执行qemu以后输出大量truly_illegal_insn怎么办？

qemu输出`truly_illegal_insn`是因为执行到了非法指令。大概率是处理器的PC跑到了奇怪的位置，处理器读取到的指令是非法指令。

处理器跑到奇怪位置的原因可能有很多种。一种是某些跳转指令跳转的地址不对，导致处理器跳到了错误的地址。还有可能是处理器把kernel_main中的东西都执行完了，顺着往下执行，而下面的内存中没有存放任何合法指令，所以导致触发了异常。

对于前者，可以用gdb的si指令跟踪，仔细检查每次跳转到的地址是否正确。对于后者，可以考虑在`kernel_main`或者`head.S`（反正就是内核执行结束的位置）中添加死循环，避免处理器继续往后执行无效指令。

为什么需要用createimage做一个image才能运行？直接把bootblock和kernel连接成一个文件不能执行吗？

bootblock和kernel都是编译器编译出来的ELF可执行文件。和我们平时在操作系统上所执行的那些可执行文件的格式是一样的。例如：在Linux上，我们双击执行一个程序，OS的加载器会帮我们把程序加载好，然后再去执行。

操作系统中的加载器会负责读入可执行文件，然后按照ELF文件中给出的每个segment的内存起始地址、所需占用的内存空间等信息，将文件中的内容加载到内存的相应位置。之后，再跳转到ELF文件头中记录的程序入口地址。这样处理器就开始执行这个可执行文件了。

但是我们现在编写的是操作系统，所以没有加载器帮我们加载，故而需要createimage这样的工具。按照约定，BIOS只会将SD卡的头512B拷贝到内存中并执行。整个过程没有任何类似于操作系统的加载的过程。另外，bootblock只有512B，难以进行ELF的解析的动作，所以一般也是单纯将后续数据直接拷入到内存中。因此，我们的SD卡中的内容，就是程序在内存中实际运行时的内存布局的镜像。
createimage实际上就类似于做了一个加载器的功能。只不过一般操作系统是读入ELF文件，然后拷贝到内存中。createimage是读入ELF文件，然后写入到image文件中。image文件中的内容即是ELF文件加载到内存后的样子。这样BIOS/bootblock只需要把数据拷贝到内存中即可直接执行。

直接把bootblock/kernel连接成一个文件的话，仍会有ELF文件头。且未初始化的全局变量等数据，在ELF文件中只记录了它的大小，没有真的在文件中给它留位置。因此，ELF文件中的数据和程序在真实内存中的样子仍有一定差别。BIOS/bootblock又没有解析ELF文件的功能，因此，直接连接成一个文件是无法正确执行的。