在上次的 P1 review 课堂上，我们针对大家存在的一些问题集中梳理了一下，以下为几个大家常见的弄不明白的问题，希望大家可以详细阅读，查漏补缺。

疑点一：内核入口地址确认是如何实现的？为什么正好在内核的开头？

在本次实验，内核的入口地址其实就是该函数：

```
void __attribute__((section(".entry_function"))) _start(void)
```

大家都知道，内核的放置位置在 **0xa0800200** 处，而我们跳转的地址也是这里，那么为什么内核的入口函数的位置就在这里呢？而不是其他的地方呢？有的同学可能会说，内核代码里就这么一个函数当然放在开头了，那么如果后来随着内核规模的增大，出现更多的函数，那么是否还能保证这个函数的位置就在这里呢？如果其他的函数被放到开头，那么我们还跳转到开头就会出错了，换而言之，如果内核的入口函数被随便放置，那我们其实也不知道从 bootloader 阶段到 OS 阶段往哪里跳了！

其实大家完全不必担心这个问题，因为我们在编译内核的时候已经通过某些方式告诉了编译器：“编译内核的时候一定要把内核入口函数放到开头”。所以每次跳到内核的开头这里一定会是入口函数。

具体做法就是在我们写代码的时候，我们会在_start 函数前加上如下标记：

```
__attribute__((section(".entry_function")))
```

这个标记并不是多余的，而是配合 ld.script 使用的，它的意思就是将_start 入口函数申明为名字叫”.entry_function”的 section，而在连接器脚本中，它会将这个名字叫”.entry_function”的 section 放到开头，具体做法如下：

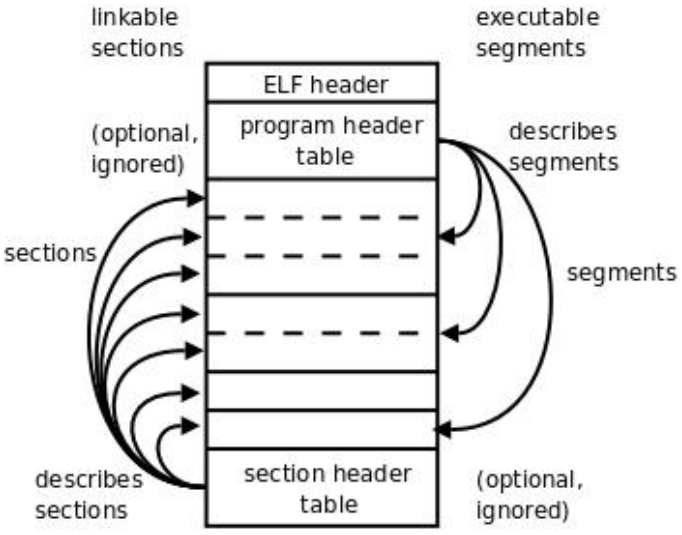
```
.text :
{
  _ftext = . ;
  *(.entry_function)
  *(.text)
  *(.rodata)
  *(.rodata1)
  *(.reginfo)
  *(.init)
  *(.stub)

  *(.gnu.warning)
} =0
```

通过二者的“合作”，最终我们可以将内核入口函数放到编译完成的二进制文件开头。

疑点二：createimage 工具所创造的镜像（image）到底和 ELF 文件有什么关系？

在进行 review 的时候，很多同学对 createimage 的具体功能存在疑惑，对任务书的一些内容存在疑问，在这里简单的为大家说明一下。

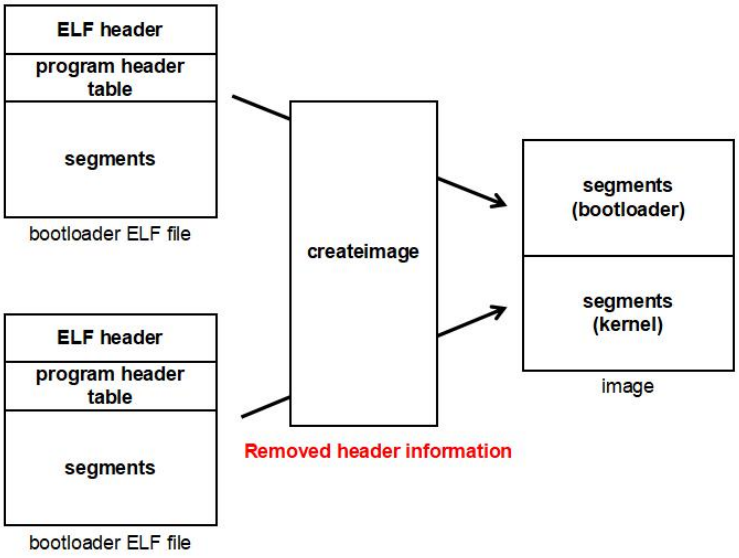
大家都知道 ELF 文件的格式大致如下：



其中 ELF header 和 program header table、section header table 都是用来指示 ELF 文件中的代码和数据的放置位置的，对于一个 ELF 文件，如果想让它在我们现有的操作系统（比如 Ubuntu），那可能需要完整的信息（包括各种 header），因为进行./[ELF 文件]去运行一个可执行文件时，操作系统会从 header 中分析代码和数据的位置，把它们加载的内存中，

然后去执行。

但在我们的内核镜像中，这些 header 都是没有用的，我们在制作镜像的时候并不需要它们，说的更直白一些，这些 header 是需要 createimage 去移除的。我们生成的最终 image 只需要保留 segments 部分，而不需要 ELF header、program header。如下图：



当然，如何 createimage 中如何找到 segment 实际上是通过 ELF header 和 program header 进行的，具体实现大家了解完 ELF 格式后编程实现。

疑点三：关于 bonus 的一些提示

在之前的任务中，我们将内核放在 0xa0800200 处进行执行，内核的代码段基地址为 0xa0800200，确立一个可执行文件的代码段基地址是在编译中加入-Ttext 参数决定的，如下：

```
kernel: kernel.c
    ${CC} -G 0 -O2  -fno-pic -mno-abicalls -fno-builtin -nostdinc -mips3 \
    -Ttext=0xffffffffa0800200 -N -o kernel kernel.c -nostdlib -Wl,-m -Wl,elf32ltsmip \
    -T ld.script
```

可以看到，我们将 kernel 的段基地址设置为了 0xa0800200，我们将编译后的 kernel 反编译出来结果如下：

```
 7  a0800200 <_start>:
 8  a0800200:   27bdffe8    addiu   sp,sp,-24
 9  a0800204:   3c04a080    lui a0,0xa080
10  a0800208:   3c028007    lui v0,0x8007
11  a080020c:   afbf0014    sw  ra,20(sp)
12  a0800210:   3442b980    ori v0,v0,0xb980
13  a0800214:   0040f809    jalr    v0
14  a0800218:   24840258    addiu   a0,a0,600
15  a080021c:   08200087    j   a080021c <_start+0x1c>
16  a0800220:   00000000    nop
```

可以看到，内核的_start 入口函数地址为 0xa0800200，并且在汇编中寻址和跳转操作的偏移量都是相对于 0xa0800200 进行的，那么这段代码加载到虚存为 0xa0800200 的位置的确是可以运行的，实际上，我们也的确是让大家把它加载到这个位置。

那么如果我们要将内核加载到 0xa0800000 的地方去运行呢？如果内核的代码基地址还是 0xa0800200 那是否会出错呢？相信大家已经有了自己的想法，在这里就不做过多的说明了~

反编译指令：

在后续的任务中，大家可以使用反汇编指令去做一些调试，它可以将可执行文件反汇编成汇编代码，不但如此，其中还会包含一些其他的信息，比如上述就显示了指令在虚存中的信息等，具体实现很简单，只需要一条指令：

```
mipsel-linux-objdump -d kernel > kernel.txt
```

其中 kernel 为可执行文件，kernel.txt 是反汇编出的信息输入文件。