P4代码讲解

p4因为要开启虚存所以流程上有很大的不同，相对来讲较为繁复。RISC-V开启虚存就意味着所有的代码都要跑在虚存上，而内核代码一般使用0xffffffc000000000开头的虚地址，所以在开启虚存时，还要把内核搬到这个位置上。我们按照启动的流程进行讲解：

启动的引导和以前一样，从bootblock.S开始。但这次的镜像制作是有所变化的。以前的image=bootblock+main，这次的image变为了bootblock+kernelimage。kernelimage=(head.S+boot.c+string.c)+payload.c。前面三个文件是用于建立初始的内核页表，解析ELF格式的内核，并将内核ELF文件的segment搬到文件指示的虚地址上。payload.c是用elf2char生成的一个C文件，里面的char数组其实就是内核的ELF文件。相当于把内核的ELF文件嵌了进去。boot.c已经完整地给大家了，这里简单讲解一下：

```
int boot_kernel(unsigned long mhartid, uintptr_t riscv_dtb)
{
    // boot.c负责建立页表、加载内核最后跳转到内核入口
    if (mhartid == 0) {
        // 建立页表
        setup_vm();
        // 加载内核ELF文件。_elf_main和_length_main是payload.c中
        // 嵌入了内核ELF文件的数组。通过解析elf文件可以获得内核入口地址。
        // 注意这个地址已经是内核虚地址(0xffffffc开头的地址)了。
        start_kernel =
            (kernel_entry_t)load_elf(_elf_main, _length_main,
                                     PGDIR_PA, directmap);
    } else {
        enable_vm();
    }
    // 跳转到内核入口地址
    start_kernel(mhartid, riscv_dtb);
    return 0;
}
```

内核页表的建立过程需要大家仔细理解一下。后续用户页表的建立也是相似的。在前面的实验中，我们的内核管理了全部的物理地址。为了维持代码中的这个假设不变，我们的内核页表页映射了全部的物理地址。且内核虚地址和物理地址之间的偏移是固定的，线性对应的。相当于内核虚地址减去一个固定偏移就可以得到对应的物理地址。这样，后面我们的内核**管理内核虚地址，就相当于管理物理内存。访问内核虚地址，就相当于访问物理内存。**这一点特性在后面很多地方都会用到。

```
/* Sv-39 mode
 * 0x0000_0000_0000_0000-0x0000_003f_ffff_ffff is for user mode
 * 0xffff_ffc0_0000_0000-0xffff_ffff_ffff_ffff is for kernel mode
 */
void setup_vm()
{
    // 清空页目录的所有项（清零即可）
    clear_pgdir(PGDIR_PA);
    // 建立内核需要的二级页表，将0x50200000-0x60000000之间的地址按照2MB大小的
    // 页面映射到0xffffffc050200000-0xffffffc060000000。
    // map kernel virtual address(kva) to kernel physical
    // address(kpa) kva = kpa + 0xffff_ffc0_0000_0000 use 2MB page,
    // map all physical memory
```

```
    PTE *early_pgdir = (PTE *)PGDIR_PA;
    // 建立物理地址到虚地址的映射
    for (uint64_t kva = 0xffffffc050200000lu;
         kva < 0xffffffc060000000lu; kva += 0x200000lu) {
        map_page(kva, kva2pa(kva), early_pgdir);
    }
    // 这里需要额外把虚地址0x50000000~0x50400000映射到
    // 物理地址0x50000000~0x50400000。因为我们这个boot.c当前的
    // 地址在这个范围内。一旦开启虚存，所有的访存都会被认为是虚地址，
    // 所以为了boot.c能够正常运行完成，需要临时做一下这个映射。
    // 到内核正确启动后，由内核取消这个映射。
    // map boot address
    for (uint64_t pa = 0x50000000lu; pa < 0x50400000lu;
         pa += 0x200000lu) {
        map_page(pa, pa, early_pgdir);
    }
    enable_vm();
}
```

用2MB的页映射物理内存的方式比较简单，就是严格按照RISC-V手册的要求填写页表。后面建立用户页表的时候流程也是相同的，只不过多查找一级而已。

```
// using 2MB large page
void map_page(uint64_t va, uint64_t pa, PTE *pgdir)
{
    va &= VA_MASK;
    uint64_t vpn2 =
        va >> (NORMAL_PAGE_SHIFT + PPN_BITS + PPN_BITS);
    uint64_t vpn1 = (vpn2 << PPN_BITS) ^
                    (va >> (NORMAL_PAGE_SHIFT + PPN_BITS));
    if (pgdir[vpn2] == 0) {
        // 如果页目录项没有对应的下一级页表，就分配一页并设置页目录项
        // alloc a new second-level page directory
        set_pfn(&pgdir[vpn2], alloc_page() >> NORMAL_PAGE_SHIFT);
        set_attribute(&pgdir[vpn2], _PAGE_PRESENT); // 设上valid位
        clear_pgdir(get_pa(pgdir[vpn2])); // 清空下一级页表
    }
    // 根据页目录项获取下一级页表的起始地址
    PTE *pmd = (PTE *)get_pa(pgdir[vpn2]);
    // 设置页表项
    set_pfn(&pmd[vpn1], pa >> NORMAL_PAGE_SHIFT);
    set_attribute(
        &pmd[vpn1], _PAGE_PRESENT | _PAGE_READ | _PAGE_WRITE |
                        _PAGE_EXEC | _PAGE_ACCESSED | _PAGE_DIRTY);
}
```

加载ELF格式内核的流程和之前createimage.c的过程相似。就不过多讲解了，这里只讲一些核心的代码：

```
/* prepare_page_for_kva should return a kernel virtual address */
static inline uintptr_t load_elf(
    unsigned char elf_binary[], unsigned length, uintptr_t pgdir,
    uintptr_t (*prepare_page_for_va)(uintptr_t va, uintptr_t pgdir))
{
    // ...
```

```
    while (ph_entry_count--) {
        phdr = (Elf64_Phdr *)ptr_ph_table;

        if (phdr->p_type == PT_LOAD) {
            // 只有PT_LOAD是类型的segment是需要加载的
            for (i = 0; i < phdr->p_memsz; i += NORMAL_PAGE_SIZE) {
                if (i < phdr->p_filesz) {
                    // 这个ELF.h是加载内核和加载用户程序通用的，所以分配
                    // 内存这里做了抽象。prepare_page_for_va的功能是
                    // 为虚地址va分配物理内存，并将虚地址到物理地址的映射
                    // 插入到pgdir处的页表中。函数返回分配的物理内存对应的
                    // 内核虚地址。这里就用到了前面说的特性，我们访问这个
                    // 内核虚地址其实就是在访问对应的物理内存。加载elf的时候，
                    // 当前生效的页表大概率不是我们正在加载的elf文件所使用的页表，
                    // 所以为了往这个elf文件所在地址空间写东西，我们只能
                    // 直接写对应的物理地址。
                    unsigned char *bytes_of_page =
                        (unsigned char *)prepare_page_for_va(
                            (uintptr_t)(phdr->p_vaddr + i), pgdir);
                    memcpy(
                        bytes_of_page,
                        elf_binary + phdr->p_offset + i,
                        MIN(phdr->p_filesz - i, NORMAL_PAGE_SIZE));
                    if (phdr->p_filesz - i < NORMAL_PAGE_SIZE) {
                        for (int j =
                                  phdr->p_filesz % NORMAL_PAGE_SIZE;
                             j < NORMAL_PAGE_SIZE; ++j) {
                            bytes_of_page[j] = 0;
                        }
                    }
                } else {
                    long *bytes_of_page =
                        (long *)prepare_page_for_va(
                            (uintptr_t)(phdr->p_vaddr + i), pgdir);
                    for (int j = 0;
                         j < NORMAL_PAGE_SIZE / sizeof(long);
                         ++j) {
                        bytes_of_page[j] = 0;
                    }
                }
            }
        }

        ptr_ph_table += ph_entry_size;
    }

    return ehdr->e_entry;
}
```

进入到内核以后，从start.S开始执行。这个start.S其实就是过去的head.S。不过需要大家注意一下，栈的地址之类的是否是内核虚地址。此时虚存已经完全开启了。内核的编译和以前也有所不同。以前内核是和测试程序等用户态程序编译在一起的。这次，我们的用户态程序是真正的用户程序了，和平时在一般的操作系统上编译的东西一样。所以这次内核（编译出来的文件名叫main）=内核部分+user_program.c。user_program.c和前面的payload.c一样，也是嵌入了ELF文件的C代码。所有的用户程序都通过elf2char嵌入到了user_program.c中。内核部分的执行和以前没有太大区别，只不过原来的物理地址都应该变成相应的内核虚地址了。

这里简单讲解一下Makefile的相应部分：

```
SRC_LIBC        = ./tiny_libc/printf.c ./tiny_libc/string.c
./tiny_libc/mthread.c ./tiny_libc/syscall.c ./tiny_libc/invoke_syscall.S \
                                ./tiny_libc/time.c ./tiny_libc/mailbox.c
./tiny_libc/rand.c ./tiny_libc/atol.c
SRC_LIBC_ASM    = $(filter %.S %.s,$(SRC_LIBC))
SRC_LIBC_C      = $(filter %.c,$(SRC_LIBC))
# 我们这次编译生成一个货真价实的C库。所有tinylibc下面的.c和.S文件，我们都编译成对应的.o文件，
然后用AR压缩到一起形成.a格式的库文件。
libtiny_libc.a: $(SRC_LIBC_C) $(SRC_LIBC_ASM) user_riscv.lds
        for libobj in $(SRC_LIBC_C); do ${CC} ${USER_CFLAGS} -c $$libobj -o
$${libobj/.c/.o}; done
        for libobj in $(SRC_LIBC_ASM); do ${CC} ${USER_CFLAGS} -c $$libobj -o
$${libobj/.S/.o}; done
        ${AR} rcs libtiny_libc.a $(patsubst %.c, %.o, $(patsubst %.S,
%.o,$(SRC_LIBC)))
```

之后，用户程序由crt0、c库还有相应的C代码编译出来：

```
SRC_USER        = ./test/test_shell.elf ./test/rw.elf ./test/fly.elf
./test/consensus.elf ./test/lock.elf ./test/mailbox.elf
# %.elf是通配符，相当于每个.elf都会由对应同名的.c通过同样的流程编译出来
$(ARCH_DIR)/crt0.o: $(ARCH_DIR)/crt0.S
        ${CC} ${USER_CFLAGS} -c $(ARCH_DIR)/crt0.S -o $(ARCH_DIR)/crt0.o

%.elf : %.c user_riscv.lds libtiny_libc.a $(ARCH_DIR)/crt0.o
        ${CC} ${USER_CFLAGS} $< ${USER_LDFLAGS} -o $@
```

最后，打包到user_program.c/.h中：

```
user: $(SRC_USER) elf2char generateMapping
        echo "" > user_programs.c
        echo "" > user_programs.h
        # 通过elf2char把所有的.elf文件嵌入到user_programs.c/.h中
        for prog in $(SRC_USER); do ./elf2char --header-only $$prog >>
user_programs.h; done
        for prog in $(SRC_USER); do ./elf2char $$prog >> user_programs.c; done
        ./generateMapping user_programs
        mv user_programs.h include/
        mv user_programs.c kernel/
# 把用户程序的elf文件和内核编译到一起。
main: $(SRC_MAIN) user riscv.lds
        ${CC} ${CFLAGS} -o main $(SRC_MAIN)  ./kernel/user_programs.c -
Ttext=${KERNEL_ENTRYPOINT}
```

我们用这种方式其实单纯是因为目前还没有文件系统。一般的操作系统的会直接从磁盘上通过文件系统读取ELF文件内容。我们只能暂时用这种将文件完整嵌入并编译到一起的方式临时替代文件系统。

用户进程的启动和以前大同小异，只是要建立页目录加载elf文件。加载elf的方式和加载内核elf类似，按照上面的讲解自行实现即可。建立页目录需要把之前内核的地址和用户的地址都映射进去。否则，切换到内核态以后，内核地址就没有对应页表了。这里有一个相对麻烦一点的点是传参。传参是在shell中解析出来的，但是要传到新建立的pcb的地址空间里面。此时当前的页表还是shell的页表，所以我们没有办法通过用户态的地址直接访问到新建立的pcb的地址空间（除非我们把页表切换成这个新的pcb的页表）。这时，就又要利用"访问内核虚地址等于访问对应物理地址"这种特性了。内核此时是知道新的pcb

的栈所对应的物理地址的（毕竟页表什么的都是由内核管理的）。于是，想往这个新的pcb的地址空间中拷贝东西，只需要查询它的虚地址所对应的物理地址，再通过相应的内核虚地址写内容，就相当于往这个新的pcb的地址空间中拷贝东西了。

FAQ

pgtable.h中的函数都是什么作用？

pgtable.h中有一些需要自己理解实现的函数，这里明确一下它们的作用：

```
/* 将内核虚地址转换为物理地址。
 * note: 内核虚地址和物理地址之间是线性映射的。因此，可以利用
 *       内核虚地址空间和物理地址空间的差值是个固定值的特性，
 *       直接计算出来。
 * 参数说明:
 *  kva: 内核虚地址，是一个0xffff....开头的内核使用的虚地址
 * 返回值: kva对应的物理地址。
 */
static inline uintptr_t kva2pa(uintptr_t kva);

/* 物理地址转换成内核虚地址，和kva2pa的功能正好相反。
 * 参数pa为物理地址。
 */
static inline uintptr_t pa2kva(uintptr_t pa);

/* entry为PTE项的内容，返回值为PTE中存放的物理页号对应的物理页的起始地址。 */
static inline uint64_t get_pa(PTE entry);

/* 查询pgdir_va处存放的页表，获取虚地址va对应的物理地址。
 * 最后将这个物理地址转换成内核虚地址返回。 */
```

```
static inline uintptr_t get_kva_of(uintptr_t va, uintptr_t pgdir_va)

/* 设置/获取`entry`的物理页框号。注意，这里是物理页框号，而不是物理页的地址 */
static inline long get_pfn(PTE entry);
static inline void set_pfn(PTE *entry, uint64_t pfn);

/* 获取/设置`entry`中的属性位 */
static inline long get_attribute(PTE entry, uint64_t mask);
static inline void set_attribute(PTE *entry, uint64_t bits);

/* 将整个页目录/页表清零 */
static inline void clear_pgdir(uintptr_t pgdir_addr);
```

本次内核和用户程序的启动流程是什么？

由于存在虚存，本次启动流程与之前有明显变化。启动流程及涉及的文件如下：

1. bootblock.S：bootloader，和以往的实验一致，负责将SD卡上的镜像加载到内存。
2. head.S：负责设置C语言运行环境，跳转到boot.c中的`boot_kernel`。
3. boot.c：负责设置初始的内核页表，开启虚存机制，加载内核(ELF格式)到内核虚地址上。跳转到内核ELF格式所指示的入口处。
4. start.S：即以前的head.S，初始化内核的BSS，设置内核栈等，最后跳入到内核入口处。
5. main.c：内核C语言代码入口。

用户程序的启动流程是：

1. crt0.S：相当于内核的head.S，代码基本上和head.S相似，按照注释写即可，最后跳转到main
2. XXX.c：测试程序的C代码（XXX代表相应的文件名）。

与前面的实验有所不同的是，本次是bootblock和kernelimage一起传入到createimage中，形成一个镜像。kernelimage是由head.S/boot.c等启动程序和payload.c编译而成。payload.c是用elf2char工具将和前面实验中类似的内核镜像main转换成char数组嵌在c文件里面形成的。具体过程可以看Makefile。
和内核一样，用户程序也会使用elf2char等工具把elf文件内嵌在C文件里面。
user_programs.h/user_programs.c就是嵌入了用户程序elf的C文件及其对应的头文件。

SV-39是三级页表，为什么任务书中说Kernel是二级页表？是否需要设置成别的模式？

RISC-V的模式其实规定的只是最高查找多少级页表。根据RISC-V手册的说明，只要rwx位都是0，处理器就会认为当前的页表项中的物理地址代表的是下一级页目录的物理地址。然后处理器会去查找下一级页目录。直到查找到一个r=1或者x=1的页目录项为止。SV39代表最多查找三级。如果处理器查到了两级就找到了一个r=1或x=1的项，它就会按照两级来做地址翻译。

为了更便于大家理解，这里摘一段原文的说法："Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv39 supports 2 MiB megapages and 1 GiB gigapages, each of which must be virtually and physically aligned to a boundary equal to its size."，翻译过来就是任意一级PTE都可能是最后一级PTE项(leaf PTE)，所以Sv39除了支持4KB的页以外，还同时支持2MB或1GB的页。每一页都必须按照它的大小，对齐到相应的边界上（比如4K的页的起始地址必须是4K对齐的）。

因为内核的虚地址是线性映射的，所以为了简单我们建议就直接用2MB的内存大小

什么叫做线性映射？内核和物理地址的对应关系究竟是什么？

在之前的实验中，内核是直接管理所有物理地址的。在本次实验中，内核虽然运行在虚拟地址上，但我们依旧希望内核能管理所有的物理地址。为了实现这一目的，我们采用了一个小技巧，就是把所有物理地址都线性地映射到一个内核虚拟地址上。比如0x50000000就映射到0xffffffc050000000上，0x50200000就映射到0xffffffc050200000，其他同理。这样，我们管理内核的虚拟内存，就等价与管理真实的物理内存。后面无论是需要往特定的物理内存里面写东西，还是给用户分配物理内存，或者其他的类似的工作，都可以借助于内核的虚存来完成。

用户态的虚存地址无法被内核访问是什么原因？

sstatus的SUM位控制了用户态的虚存能否被内核访问。之前的entry.s中有可能把这个位关闭了。如果需要可以自行打开。

Linux的做法是需要的时候临时打开，不需要了再立刻关掉。这是为了避免因代码错误等意外不小心改动了用户态的东西导致一些奇怪的现象。大家在做的时候无论是直接打开还是模仿Linux的方式等用到了再临时打开都可以。请根据自己的设计和实际情况自行权衡。

如何理解PTE的A和D位？

PTE中的A代表Access，D代表Dirty。这两个位是为了追踪某个页是否被访问/写入过而设置的。所以一开始这两位设置位0，然后当某个页被访问或写入时，A或D位设置为1。这里有一个问题在于：由谁负责在页面被第一次访问/写入时将A或D设置为1？RISC-V标准规定了两种可行的实现：1. 由软件设置。即当处理器发现要访问/写入的页的A/D位为0（第一次访问），则产生一个page fault（中断处理程序就可以设置这两位了）。2. 硬件发现A/D为0后自动设置。按照RISC-V标准的要求，系统软件需要同时兼容这两种实现。比较巧的是，**QEMU是第2种实现，我们的板卡是第1种实现。**所以这里QEMU和板卡的行为不一致，需要格外注意。

对于需要软件设置A/D位的情况，可以根据例外的类型来决定如何设置这两位。例如，load page fault是读指令触发的，遇到该例外可以设置Access位。store page fault是写指令触发的，遇到该例外可以设置A和D位。

在我们的实验中，我们希望boot.c不产生中断。所以，在设置内核页表时，就直接将A和D位设置上，避免产生中断。

用户态程序或多核遇到莫名其妙的和大内核锁相关的问题可能是什么原因？

需要注意global pointer的加载。在start code中给了global pointer的加载代码。编译器在找很多全局变量的时候喜欢利用gp寄存器来寻址，所以gp中必须放置global pointer的地址。大内核锁如果是采用C代码实现的，需要中加载完global pointer之后再上锁，避免编译器利用错误的gp来寻址全局变量。

在本次实验中，A-CORE及以上由于单独编译出了用户程序，所以用户态和内核态的gp是不一样的，这一点需要特别注意。如果用户态出现奇怪的问题也需要检查是否正确地初始化/保存/恢复了用户态的gp寄存器。

设置页表都有哪些需要注意的？

1. 页表中的是PPN，也就是物理页框号。物理页框号的计算方式是物理页的地址/4096
2. 无论是页目录项还是页表项都需要设置Valid位
3. 为了在boot时不触发任何例外，页表项需要设置A和D位
4. 页目录项需要保证RWX都是0
5. satp寄存器中的也是物理页框号
6. 对于用户进程需要设置User位

有一种取巧一点的调试方式是，先只用一级页表跑通。如果能跑通再来构建二级页表。一个一级页表项可以映射1GB的大页，比较好调一点。弄对一级页表说明理解正确了，再去按照同样的理解搞二级页表。

bootblock无法正确读入数据或有其他奇怪的问题？

因为本次start code提供的加载的地址和以前有点不一样，所以可能有的同学的Makefile或者createimage有问题。需要做对应的调整。或者Makefile中的KERNEL_ENTRYPOINT等需要调整。

扩展名为.elf文件的生成规则在哪里？我在Makefile里面没找到？

生成规则位于Makefile中的%.elf:%.c那一行。

%代表通配符。这里的含义是任意的.elf文件都会依赖对应的.c，编译的过程是其后面的几行定义的。

riscv_dtb是个什么东西？

这个是历史遗留代码，建议直接删除。bbl会将一个fdt的首指针放置在a1寄存器中。fdt中包含了网卡地址、处理器主频等平台相关的信息。但今年实验为了简化，将和它相关的东西全部封装为了一个sbi，所以不需要自行解析了。相关的代码都可以直接安全删除掉。

遇到printk等输出语句会出现问题？

有可能是sbi_console_putstr有问题。由于machine态不认得虚存，所以p4 start code中新给了一个循环调用sbi_console_putchar实现的sbi_console_putstr，而不是像以前一样真正调用了sbi_console_putstr的sbi。p4 start的补丁应该会自动修改sbi.h，如果发现sbi.h没被修改，可以自行改动一下。

为什么明明没有memcpy但是编译报错说找不到memcpy？

gcc有可能会自动生成对于memcpy的调用，所以可能明明看着代码里没有调用memcpy，但是链接器说有memcpy。遇到这种情况，建议可以把以前内核里面的kmemcpy改成memcpy（注意参数的类型也要和编译器想要的一致）。或者也可以额外添加一个memcpy进去。除了memcpy以外有可能有其他函数，都照类似的方法改就行。

如何理解PTE的G位?如何设置ASID？

ASID是用于区分进程的，设置成进程pid就行。我们都知道，TLB相当于是页表的Cache。那么，当进程切换的时候，我们要切换页目录，那么原来TLB里面的内容就需要都丢弃掉。这是因为TLB缓存的是之前的进程的虚地址和物理地址的对应关系。但切换了进程之后，同样的虚地址在当前进程中可能就对应了不同的物理地址。因此，为了避免出错，需要把之前的TLB中缓存的映射关系都丢弃掉。

将TLB全刷掉，等之后切换回原来的进程时，又要重新缓存页表的内容。这会造成某种程度的性能损伤。为了缓解这一问题，TLB中设置了ASID，以允许不同进程的页表项都被缓存在TLB中。每次进行地址翻译时，不但对比虚地址本身，还对比ASID。这样就可以在允许不同进程的页表项同时存在于TLB中，无需每次在切换进程时都刷掉TLB中的内容。

G位的含义与上面讲的原理有关。G代表Global，也就是在各个地址空间上都有的映射。比如说，内核在每个进程的虚空间中都有，所以为了这种需求，RISC-V设计了G位。对于标了G位的表项，TLB在比较时不会比较ASID。因此，对于内核虚地址这种在所有地址空间上都有的地址，可以设置上G位。

但是我们实验用的处理器比较简单，所以实际上不支持ASID及相关功能。所以大家就简单地把pid设置成ASID，然后不用设置G位即可。**每次切换进程时，必须刷新TLB，以保证TLB的内容正确。**

用户栈地址为什么的位置有什么特殊要求吗？

没有，只要是用户态地址就行。

用户态地址是指第39位及更高的位都是0的地址。例如我们给的位置：0xf00010000lu，第37位到第64位都是0，所以它是用户地址。

手册中的ppn被划分为了ppn0、ppn1和ppn2，有什么特殊意义吗？

没有特别的意义，只是单纯为了看着和VPN那边对应。大家填写ppn的时候只需要把物理页的地址/4096填进去就可以，不用特别在意ppn0、ppn1和ppn2。

head.S中的pid0_pcb等无法链接上该怎么办？

在本次实验中，内核的入口从原来的head.S变成了start.S。head.S变成了为boot.c建立执行环境的汇编代码。因此，需要针对性的做一些修改。例如，对于boot.c来说，pid0_pcb是毫无意义的，因为这是内核里才有的一个概念。head.S中仅需要为boot.c的执行初始化bss、设置栈空间即可。

有些文件无法include进去该怎么办？

这次的我们的内核和用户程序是完全分开的。因此，用户程序只使用tinylibc的include里面的文件。而内核基本上只使用include里面的文件。如果实在需要在用户程序中用某些内核的参数，可以将相应的定义复制到用户的文件中。

另外，如果之前在设计的时候，不慎将部分内核用的结构或者定义放在了tinylibc下，需要自行调整代码结构，把内核的东西和用户的东西完全区分开。

为什么Makefile里面的入口地址是0x50301000？

这个是和写参考答案代码的时候的一些情况有关。大家只要自己理顺自己的bootblock、head.S和内核地址的关系即可，无需按照这个地址来设计。

参考答案代码的设计思路是，把0x50202000留给head.S当作栈了。所以就把代码和数据等放在了0x50301000上，bootblock也把镜像加载到了这个地址上。之后，加载内核后，把内核放在了0x50401000的位置上。但这种设计并不好，浪费了很多不必要的空间。大家自己合理设计即可。

主要需要考虑的问题是要注意给boot.c的栈不要和内核镜像本身重叠，导致运行的时候栈把后面的镜像覆盖了。

多线程情况下，新线程的gp怎样设置？

实现多线程的相对简单的方式就是复用原来的进程的相关机制，实现一个**和父进程共享地址空间**的特殊的进程。这样其实就实现了多线程。但在这样实现的过程中，比较容易出现的问题是，gp指针的设置问题。典型的错误现象是发现多线程的时候用户传递的某些变量或者用户使用的某些变量的地址突然变成了个内核地址。

在只有用户进程的情况下，我们设置gp指针的方式是在crt0.S中用la命令加载gp。需要在crt0.S中加载gp的原因是，本次我们内核和用户程序是分开编译的。用户有用户的gp，内核有内核的，两个值是完全不一样的（虽然内核里面gp的符号和用户相同，但是因为分开编译所以这个符号的地址完全不一样）。

但多线程的时候存在一个麻烦，在开启新线程的时候，新线程是不会从crt0.S开始执行的。新线程会直接从用户指定的函数入口开始，没有经过初始化gp的过程。为了解决这一问题，需要在开启新线程的时候，把新线程的gp设置成和“父进程”的gp相同的值。父进程的gp在我们保存上下文的时候已经保存下来了，可以从父进程的上下文里面获取到这个gp的值。

编译shell时遇到undefined reference to 'main'?

shell需要仿照其他测试程序，将入口函数名改为main。

调用flush_tlb_all函数卡死？

`flush_tlb_all`函数存在问题，刷tlb的时候请使用`local_flush_tlb_all`函数。暂时无需考虑多线程情况下

，需要刷别的核的TLB的情况。理论上我们的测试应该不会触发必须刷掉其他核TLB才能保证正确性的情况。

argv应该怎么放？

建议argv直接放在新的用户进程的栈上，由内核在`do_exec`的时候将从shell中解析出来的参数拷贝到新进程的栈上。拷贝的时候需要注意的是，我们此时仍处于调用`do_exec`系统调用的进程的地址空间下，所以无法直接通过用户虚地址读写新的用户程序的栈。因此，需要通过内核虚地址写新的用户程序的虚地址。

用户程序应该如何调试？

编译出的elf格式的用户程序位于`test`目录下，扩展名为.elf。以test_shell.elf为例，想查看其反汇编代码需要手工进行反汇编：

```
riscv64-unknown-linux-gnu-objdump -d test_shell.elf
```

在gdb中想要调试用户代码需要加载用户程序符号表，可以在启动gdb的时候将`.elf`文件作为参数传入，也可以在gdb运行的过程中用`file`或`symbol-file`命令加载`.elf`文件。

QEMU上sbi_sd_write发生error怎么办？

QEMU上的sbi_sd_write只能写镜像文件大小的范围。超过了镜像文件大小就会报错。因此，需要用第一节课上讲的命令将image扩展成一个足够大的磁盘文件。

```
# count代表追加多少个sector，请自行调节数目
dd if=/dev/zero of=image oflag=append conv=notrunc bs=512 count=65
```

为了省事，也可以把上面的命令直接写到makefile里面。

```
image: bootblock kernelimage createimage
        ./createimage --extended bootblock kernelimage
        dd if=/dev/zero of=image oflag=append conv=notrunc bs=512 count=65
```