**浙江大学实验报告**

课程名称： 计操作系统 实验类型： 综合

实验项目名称： RV64 虚拟内存管理

学生姓名： 专业： 计算机科学与技术 学号：

实验日期： 2023 年 11 月 1 日

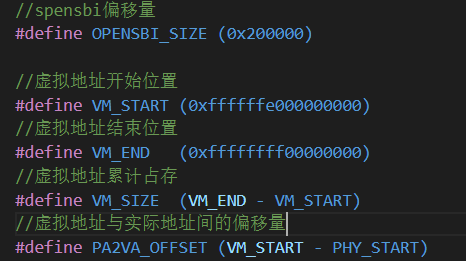
1. **实验内容**

在先前的实验中我们赋予了操作系统对多个线程调度以及并发执行的能力，由于目前这些线程都是内核线程，因此他们可以共享运行空间。如果我们需要线程相互隔离，就必须引入虚拟内存这个概念。虚拟内存可以为正在运行的进程提供独立的内存空间，同时虚拟内存到物理内存的映射也包含了对内存的访问权限，方便内核完成权限检查。

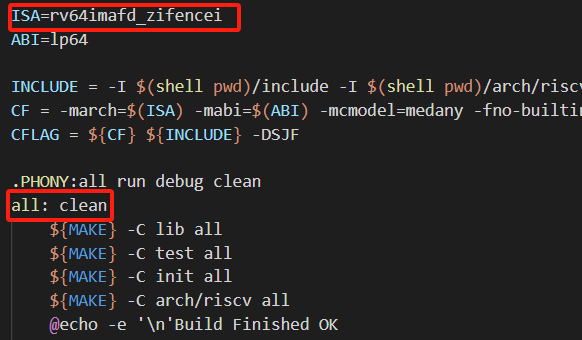
在本次实验中，我们需要关注内核如何开启虚拟地址以及通过设置页表来实现地址映射和权限控制。

1.1准备工程

在本实验中，我们设计的开展依赖于lab2已完成的代码。仅需在kernel文件夹下更换vmlinux.lds这一文件。在这一新的链接脚本中，ramv代表虚拟地址，ram代表物理地址。使用这一文件进行编译后，得到的System.map以及vmlinux文件中的符号采用的都是虚拟地址。虽然会方便之后的Debug过程，但是对linux内核的启动，也会带来一定的影响与挑战。

另外，因为虚拟地址的开启，我们需要在def.h文件中添加些许宏定义。

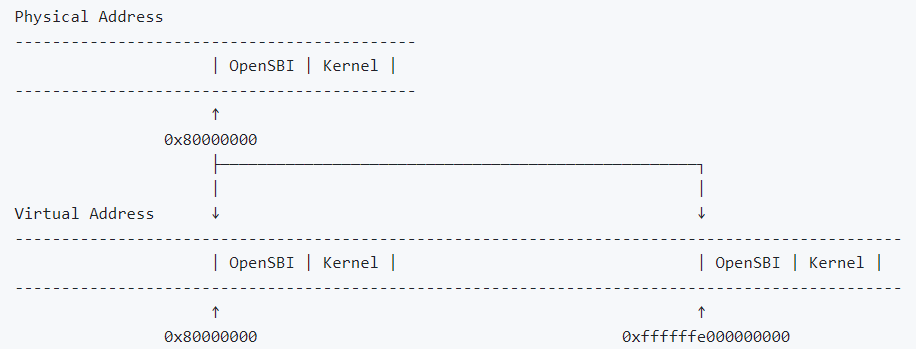
因为从本实验开始，我们需要使用刷新缓存的指令扩展，并自动在编译项目前执行clean任务来防止对头文件的修改无法触发编译任务。所以我们需要在项目顶层目录的Makefile中需要做如下更改：



1.2开启虚拟映射

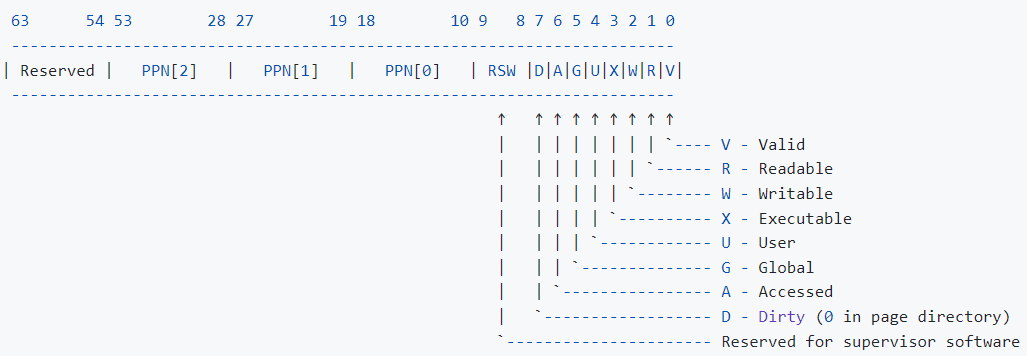
1.2.1 setup\_vm函数的实现

在setup\_vm函数中，我们要将从0x80000000开始的1GB区域进行两次映射，第一次为等值映射（使虚拟地址与物理地址相等），第二次则是将其映射到direct mapping area（经过偏移的虚拟地址位次）。图解如下：

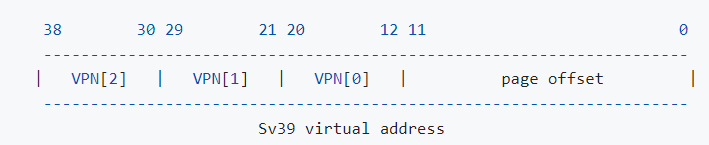


在本次实验中，我们的物理地址从0x80000000开始（OpenSBI与kernel运行的实际位置），而虚拟地址从0xffffffe000000000处开始。

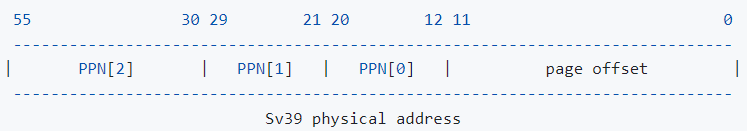
因为是1GB容量的等值映射，我们可以不使用多级页表。另外，Page Table Entry的具体结构如下。其中PTE的第0 - 9位用于存储控制信号信息，如第0位表示对映页表是否处于valid状态，第1/2位分别表示页表是否处于可读/可写状态（不同区域数据对映不同的控制需求）等等。在PTE的第10 - 53 位，需要放置对映页表的物理页号。



我们所使用的虚拟地址的address sample分析如下（仅39位，其中0 - 11 位为偏移量）。其中VPN表示当前虚拟地址所对映的多级页表信息，由2/1/0分别对映一级/二级/三级页表，page offset表示具体数据存放在对映数据页中的偏移量。

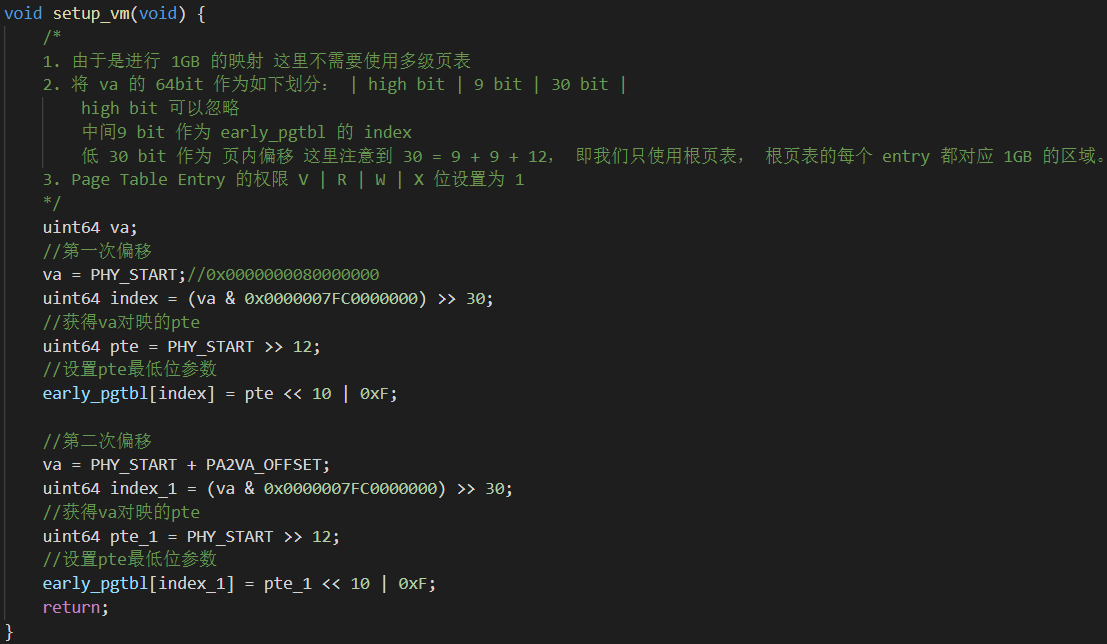


另外，本实验使用的物理地址的address sample分析如下（仅56位，其中0 - 11 位为偏移量）。其中PPN表示每一页表中对映的物理页号，2/1/0分别对映于一级/二级/三级页表中所查询得到的地址信息。PPN或记录下一级页表的地址信息，或记录对映数据页实际的物理地址信息，详细的转化及使用见后。另外，page offset表示具体数据存放在对映数据页中的偏移量

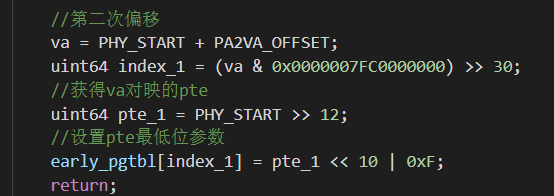


第一次直接映射部分的具体实现见下。在直接映射中，我们仅需要一份页表即可。因而对于虚拟地址而言，结构表现为high bit（本实验可忽略）+ 9 bit + 30bit的形式，等价于只需要VPN[2]中的内容即可。

其中，VPN[2]只是对映页表的查询参数，我们在此用数组的形式仿效页表的存在。则页表中对映位置所需的数据，我们需要用具体的物理地址进行一定的偏移操作及转换后才可获得。



对于第二次映射，我们的实现如下：



首先，我们需要偏移至对映的虚拟地址，通过虚拟地址来获得对映页表的VPN查询参数。而对于页表中所需的数值，则需要使用实际的物理地址通过转化获得，实际的转化过程同第一次直接映射。

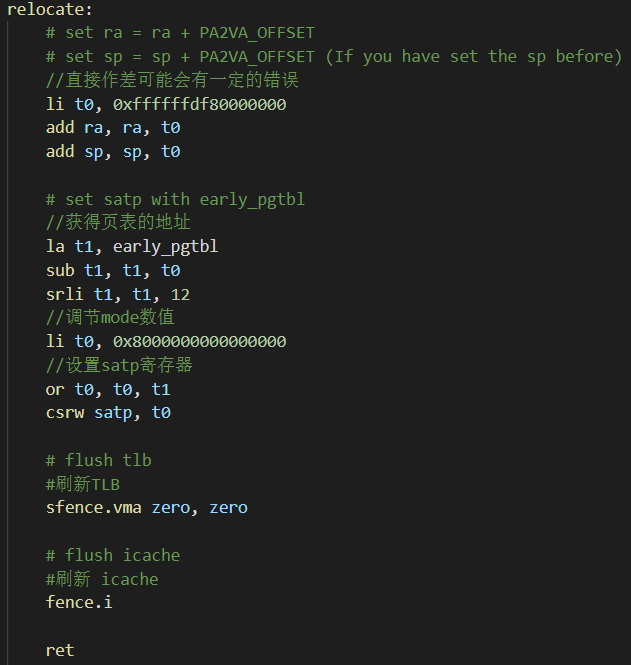
在实现了setup\_vm函数后，我们要在head.s文件的适当位置调用其。并调用相关的relocate函数，来设置satp寄存器。

8625d4fa2735a5c8ad005337bc5bd21

satp寄存器具体结构如下：



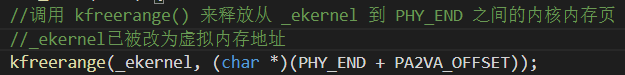
其中，MODE为该内核所使用的address模式，本实验中，我们使用Sv39模式，即Page-based 39 bit virtual addressing，因而我们需要将MODE位设置为10。ASID位在本实验中直接设0即可，我们进行简化考虑。而PPN位则用于记录顶级页表的物理页号。Relocate函数的具体实现如下：



如果我们先前设置了sp寄存器，则我们需要为sp寄存器附加虚拟地址产生的映射偏移；这样可以使后续对sp寄存器的使用合乎虚拟地址的分布。另外，我们可以通过early\_pgtbl标签来获得顶级页表的物理地址，通过偏移取或，可以用于设置satp寄存器的具体数值。最后，我们使用“sfence.vma zero, zero”指令刷新虚拟内存的TLB缓存，并使用“fence.i”指令保证指令流水线中的指令按照程序顺序执行。

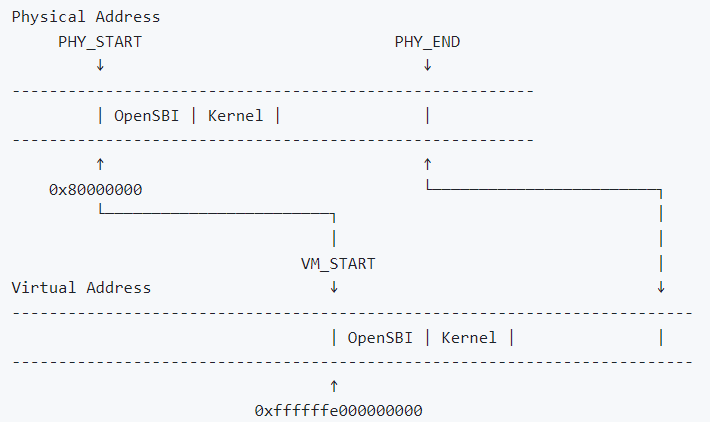
1.2.2 setup\_vm\_final函数的实现

在本实验中，我们需要在setup\_vm\_final函数中申请页面的接口，因而需要在其之前完成内存管理初始化。这可能需要修改mm.c中的代码，使其中的初始化的函数接收的起始结束地址需要调整为虚拟地址。具体的调整结果如下：

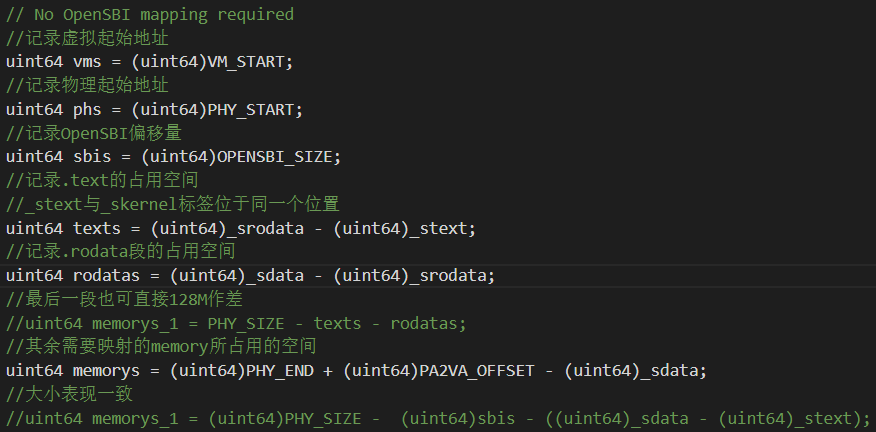


其中，因为vmlinux.lds文件的修改，\_ekernel标签对映的地址已经被修改为虚拟地址；因而只需要改变结束地址即可。

Setup\_vm\_final函数实现的地址映射如下：

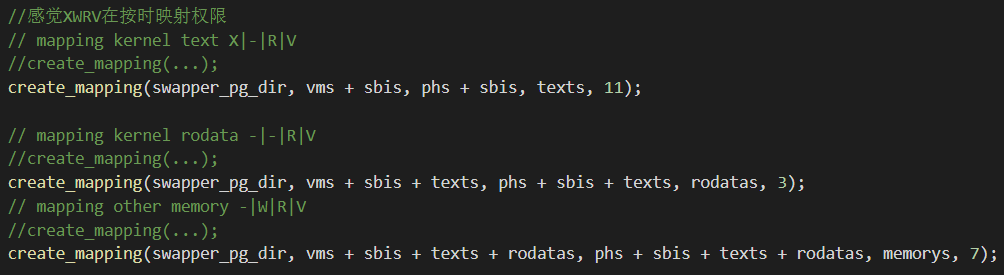


在实现setup\_vm\_final函数之前，我们先定义了其物理地址与虚拟地址间对映的顶级页表swapper\_pg\_dir。我们先使用部分uint64变量，记录各全局标签的实际地址信息及.text、.rodata等不同段实际的占用空间。



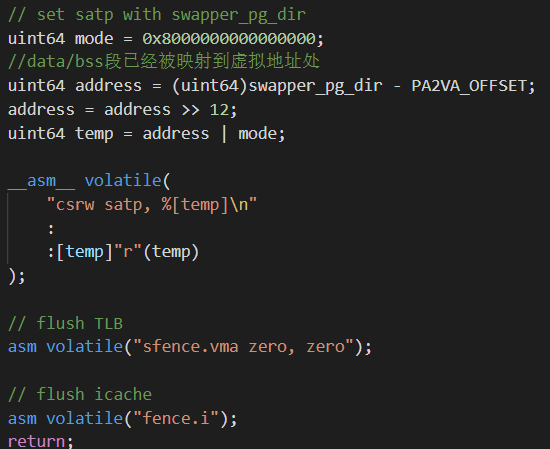
因为在本实验中OpenSBI不会映射到虚拟地址区域，所以在其余区域的映射过程中，我们需要考虑其对映的偏移量，所以需要在此先作记录转化。

接下来我们只需要使用create\_mapping函数，进行对映区域的物理地址/虚拟地址映射即可（create\_mapping函数具体实现见后）。



我们先假设当前的128M地址被我们成功从物理地址映射至虚拟地址中。此后，我们需要效仿head.s文件中的步骤，对satp寄存器进行设置，并进行TBL刷新等操作。在具体的实现过程中，只有satp数值设置时使用的物理地址数据与前不同。

这里需要特别注意，程序已经进入虚拟地址空间，因而我们所请求的swapper\_pg\_dir地址已经为虚拟地址，需要减去相应的虚拟地址偏移量。



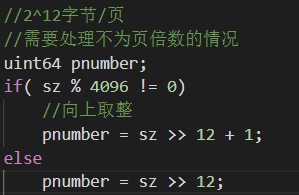
接下来，我们需要介绍create\_mapping函数的具体实现细节。首先，我们需要介绍其函数接受的相关参数。

c641f53f4af0e06d422f783a5b6ffda

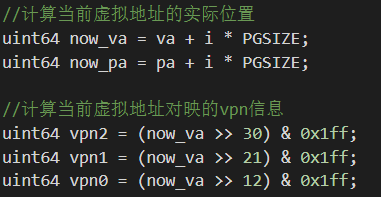
其中pgtbl记录根页表的基地址；va、pa分别记录需要映射的虚拟地址与物理地址；sz记录当前需要进行映射的内存大小（单位为字节，尽量保持4096B的整数倍 --- 即页的单位）；perm表示需要映射的权限（X|W|R|V）。

在create\_mapping函数中，我们可以在创建多级页表的时候使用 kalloc函数来获取一页作为页表目录。注意，这里的kalloc函数不再获得空闲的物理地址，而是申请获得空闲的虚拟地址，方便我们的直接操作。最后，我们可以在创建多级列表的过程中使用V bit（最低位）来判断页表项是否存在。

在具体实现的过程中，我们先要根据sz的输入，计算出当前需要进行创建页表的数据页个数。当sz不为4KB的整数倍时，我们需要进行向上取整操作。



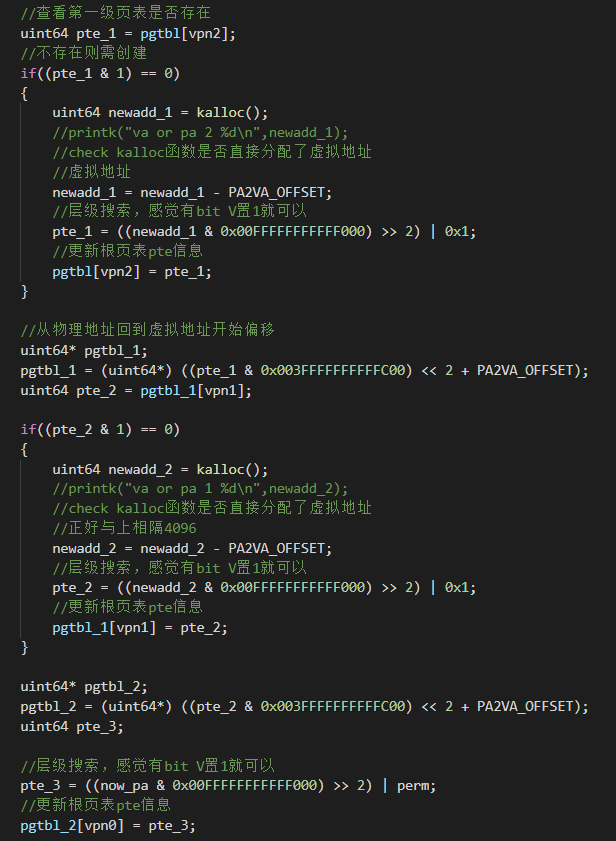
接着，我们需要使用循环来对每一个数据页进行页表的创建或填充。我们可以根据当前数据页的序号数，先计算获得它们需要对映的物理地址与虚拟地址。在拥有虚拟地址之后，我们可以便捷的计算得到其对映的VPN信息。



在获得VPN信息之后，我们需要进行层级列表是否存在的判断及申请填充操作。具体的实现见后图。首先，我们根据获得的VPN[2]信息，查询顶级页表；由此我们可以获得对映的PTE信息，若PTE信息的最低位，即V bit为1，则说明对映的二级页表已存在，因而我们继续层级搜索。若V bit为0，则说明当前二级页表并不存在，我们需要使用对映的操作进行申请与设置。

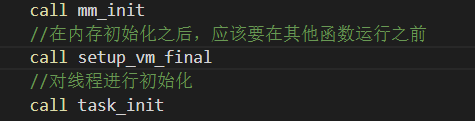
对此，我们可以先使用kalloc函数进行虚拟地址页的申请，因为获得的是虚拟地址信息，因而我们需要减去相应的偏移量。在获得二级页表对映的地址后，我们可以根据其对映的物理地址，进行一级页表的PTE信息更改。在记录物理地址信息之外，我们还需要将V bit置1，表示二级页表已建立。

在此之后，我们可以根据已有的PTE信息，进行数值偏移以获得二级页表的（数组）起始地址，并借助VPN[1]的寻址，进行二级页表PTE数值的获得。此后操作均类似于一级页表的处理。



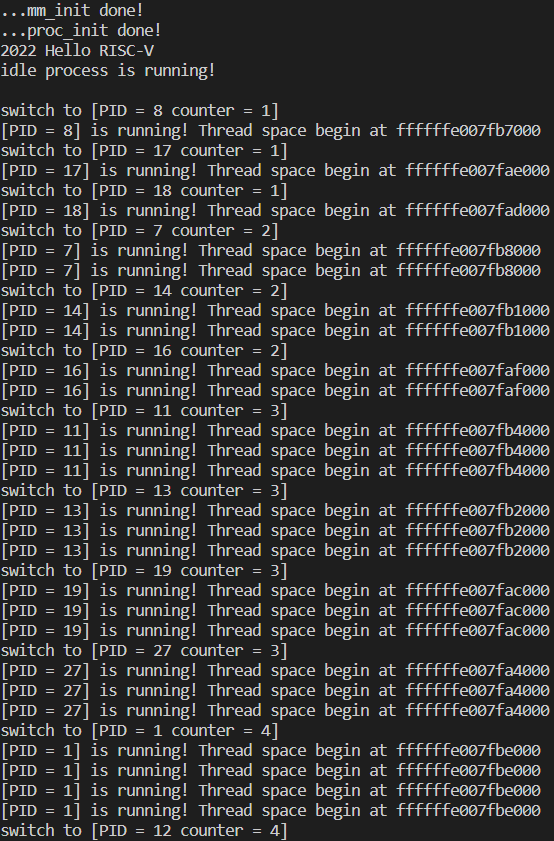
但对于三级页表而言，我们不再进行V bit的校验。因为本实验只实现三级页表的功能，从而三级页表的PTE信息，应该记录对映数据页物理地址的相关数值。因而，我们只需要对实际的物理地址进行偏移并将其与perm输入进行或操作即可。获得的结果即为当前页表PTE所需填入的数值。

最后，我们需要在合适的地方调用setup\_vm\_final函数。



1.3 调试与运行结果

在进行调试并更改所需输出内容后，我们在NR\_TASKS = 32时，可以获得以下的实验程序运行结果。



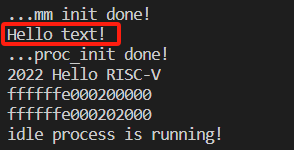
我们可以看到，程序顺利运行了mm\_init初始化内存函数，以及task\_init线程初始化函数；并正常的进入了main.c函数中。

接着，实验程序开始了正常的线程运行与调度步骤。在本次测试中，我使用的是最短作业优先调度方法；可以见到，时间片小的进程确实优先被内核调度并运行。在进程运行的每一时间片中，我们让其输出一次对映的地址信息；从结果可见，输出信息的数量与时间片大小保持一致，且输出的地址信息为虚拟地址（ffffff开头---特别注意，需要使用lx控制符介导虚拟地址输出），说明线程运行在虚拟内存中，在一定程度上证明了我们实验设计的正确性。

**二、思考题**

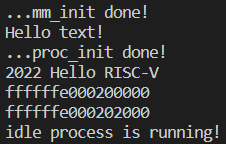
1.验证.text，.rodata段的属性是否成功设置，给出截图。

首先，程序能够正常运行，并输出正确结果，很大程度上说明了.text段以及.rodata段属性已被成功设置。另外，我们可以写在.text段设置call hello\_text函数，来判断.text段的读数据与执行代码的功能。



Qemu程序输出了我们在.text段设置的call hello\_text函数内容，从而可以证明.text段可读可执行的功能设置。

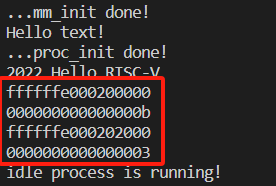
接下来，我们在.rodata段也放置类似的全局标签以及call hello\_rodata函数，来验证.rodata段的属性设置。



在输出结果中，并没有出现“Hello rodata！”的字样，说明.rodata段的程序并没有被我们执行，说明其没有执行的功能。另外，在system.map文件中，我们查询到了在.rodata段设置的标签信息，说明该段读功能正常。

d0178843c12809b7baeee592843d806

最后我们输出.text段与.rodata段对映的虚拟地址信息以及三级页表下对映的PTE后四位（X|W|R|V）的设置情况，从根本上判断这两段的属性是否被成功设置。输出结果如下图。由结果可见，.text段的后四位为B，对映于1011，意味着可执行可读；而.rodata段的后四位为3，对映于0011，则表示仅可读。与我们前述的验证性实验具有一致性。



1. 为什么我们在setup\_vm中需要做等值映射?

等值映射是指将虚拟地址空间的一部分直接映射到物理地址空间，这意味着虚拟地址和物理地址在这个范围内是相等的。

在setup\_vm函数中，我们既做了等值映射，又做了基于一定偏移量的映射。如果我们不做等值映射，则在head.s文件中执行relocate函数时，会发生访问异常，从而导致qemu的重启。

因为在relocate函数中，我们会将satp寄存器赋予正确的数值，这意味着我们将打开虚拟地址空间。打开虚拟地址空间后，整段代码将从物理地址区域搬运到了虚拟地址区域。这一操作对于大多数代码而言不会产生问题，但是对于relocate函数后续的三条指令“sfence.vma zero, zero”“fence.i”“ret”而言，它们也会被搬运到虚拟地址处。但是此时的PC依然位于物理地址处，如果没有等值映射，此时就会触发越界访问，导致程序不能正常运行。因而，我们需要等值映射，使虚拟地址与物理地址可以进行等值对映，介导上述三条语句的正常运行。

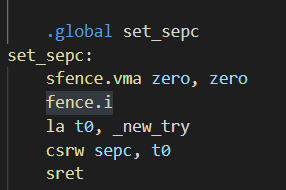
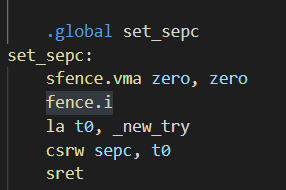
1. 在Linux中，是不需要做等值映射的。请探索一下不在setup\_vm中做等值映射的方法。

在linux的真实实现中，linux内核并没有使用等值映射的方法。如果我们在本次实验中不使用等值映射的方法，就会遇到思考题2中所述的越界访问问题。但是，我们可以效仿linux内核的实现，使用中断处理来完成这个跳转操作。

我们的问题是，一方面，如果设置了satp寄存器，我们的代码将离开实际的物理地址，但我们剩余的指令依旧需要运行在物理地址上；另一方面，我们需要ret指令，介导我们跳转到虚拟地址去，但没有satp寄存器的设置，我们的虚拟地址并没有被打开，从而产生了矛盾。

在此，可以通过设置stvec寄存器，将其对映的数值设置为任意标签。只要stvec寄存器在satp寄存器设置之前被设置了，我们就有一个通道前往虚拟地址。在设置satp寄存器后，我们不需要执行任何语句，此时会触发越界访问，因而系统会跳至stvec寄存器中设置的数值。

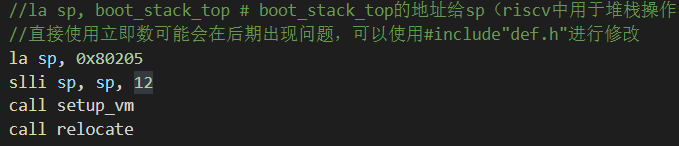
在那里，我们可以继续执行思考题2中阐述的前两条语句，并像一般中断处理程序一样，设置sepc寄存器的数值，为程序的返回挑选合适的标签（在本实验中可以新设置一个标签），再继续执行代码，即可进行跳转。因为这一系列操作均在satp寄存器设置后开展，因而跳转的地址均为虚拟地址；因而我们借助于stvec与sepc寄存器完成了从物理地址到虚拟地址的跳转。具体的实验如下。

****

**三、讨论、心得**

其实在完成程序的各项设计之后，我们也不能直接执行lab3的代码。若我们直接执行，程序也不会出现任何打印信息。经探究，其实是一些危险的越界操作，引发了qemu的不断重启；进而进入死循环，无法输出任何有效信息。

首先是call setup\_vm函数的问题。因为vmlinux.lds文件的存在，boot\_stack\_top标签已经被对映至虚拟内存中。若我们不先为sp寄存器赋予栈顶地址，则在后续使用sp寄存器时，会产生越界的现象，导致qemu重启；但我们若为sp寄存器赋予boot\_stack\_top标签的地址，则sp会得到一个虚拟的地址，这对于内核运行来说，也是越界访问（还没有开启虚拟地址），从而也会引起qemu的不断重启。因而，我们需要在call setup\_vm函数前，手动的设置sp寄存器所指向的地址。



其中使用立即数进行赋值的方法未必安全（标签可能在不同程序中地址表示不同）；在本实验中可以根据boot\_stack\_top的虚拟地址，再减去虚拟地址偏移量而得到（需要include def.h文件，可能需要一定的修改）。

在此之后，程序依然未必能正常运行。在relocate函数中，我们需要得到根页表的首地址，并使其参与satp寄存器的数值设定，但early\_pgtbl标签已被对映至虚拟地址处，而我们在设置satp寄存器前并没有开启虚拟地址，因而我们需要将early\_pgtbl标签对映的地址减去相应的偏移量。由此才能正确的完成satp寄存器的设定工作。

完成以上两步操作后，我们才能正常的运行该程序。