Lab 3: RV64 虚拟内存管理

1 实验目的

结合课堂学习的页式内存管理以及虚拟内存的相关知识，尝试在已有的程序上开启 MMU 并实现页映射，保证之前的进程调度能在虚拟内存下正常运行。

2 实验内容及要求

- 实现 Sv39 分配方案下的三级页表映射
- 了解页表映射的权限机制，设置不同的页表映射权限，完成对 section 的保护及相应的异常处理
- 理解在陷入和进程切换过程中，页表切换与栈切换的流程

请各小组独立完成实验，任何抄袭行为都将使本次实验判为0分。

请跟随实验步骤完成实验并根据文档中的要求记录实验过程，最后删除文档末尾的附录部分，并命名为"**学号1_姓名1_学号2_姓名2_lab3.pdf**"，你的代码请打包并命名为"**学号1_姓名1___学号2_姓名2_lab3**"，文件上传至学在浙大平台。

本实验以双人组队的方式进行，**仅需一人提交实验，**默认平均分配两人的得分（若原始打分为X，则可分配分数为2X，平均分配后每人得到X分）。如果有特殊情况请单独向助教反应，出示两人对于分数的钉钉聊天记录截图。单人完成实验者的得分为原始打分。

姓名	学号	分工	分配分数
秦立	3230104947	修改 head.S，实现映射机制 vm.c	50%
邱俊明	3230102363	修改进程初始化代码 task_manager.c，修改 entry.S，debug，编译及测试	50%

3 实验步骤

3.1 实验背景

3.1.1 建立映射

你可以复用上一次实验中映射好的文件夹，也可以重新创建一个容器做映射。

3.1.2 组织文件结构

你可以从 lab3.zip 下载本实验提供好的代码。

```
Lab3
├── Makefile
├── arch
│   └── riscv
│       ├── Makefile
│       └── kernel
│           ├── Makefile
│           ├── entry.S
│           ├── head.S
│           ├── main.c
│           ├── print.c
│           ├── sched.c
│           ├── test.c
│           ├── trap.c
│           ├── vm.c
│           └── vmlinux.lds
└── include
    ├── defs.h
    ├── riscv.h
    ├── sched.h
    ├── stddef.h
    ├── stdio.h
    ├── test.h
    └── vm.h
```

3.2 实验代码执行流程

为了避免我们的虚拟内存管理机制与 OpenSBI 冲突，后续的实验中，我们**不再使用 OpenSBI 来管理 M 态**，而是手动实现 M 态的管理。

本次实验中首先手动实现 OpenSBI 的部分功能，完成初始化后进入内核，然后将设置 satp 寄存器开启 Sv39 内存分配方案，随后设置物理地址与虚拟地址的映射，最后为不同 section 设置不同的映射权限，并实现相应的异常处理。为了实现映射，同学们需要分配内存空间给页表，并对页表项进行相应的设置。由于开启了 MMU，还需要对进程调度的相关代码进行一定的修改。

系统的运行流程大致如下：

1. 由于不使用 OpenSBI，机器将以 M 模式，从 `head.S: _start` 函数开始执行。
2. `_start` 函数开始执行。
    1. 首先关闭所有中断防止初始化过程受到干扰。
    2. 设置 M 模式的异常处理函数地址
    3. 将时钟中断，page fault 异常委托给 S 模式
    4. 利用 mscratch 存储 M 模式下的栈指针
    5. 清空 bss 段
    6. 打开中断使能

        7. 切换到 S 模式，进入 S 模式初始化函数
    3. S 模式下，进入 `_supervisor` 函数
        1. 关闭 MMU
        2. 初始化页表
        3. 开启 MMU
        4. 设置 S 模式下的中断处理函数 （因为 M 模式已经把一些中断委托给 S 模式了）
        5. 跳转到虚拟空间地址下的 `start_kernel` 函数
    4. `start_kernel` 函数
        1. 利用 ecall 中断设置第一次时钟中断
        2. 调用 `task_init` ，创建进程，分配空间和相关元信息
        3. `call_first_process` 进入第一个用户程序
        4. 发生时钟中断

3.3 虚拟内存管理

虚拟内存是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。

通过内存地址虚拟化，可以使得软件在没有访问某虚拟内存地址时不分配具体的物理内存，而只有在实际访问某虚拟内存地址时，操作系统再动态地分配物理内存，建立虚拟内存到物理内存的页映射关系，这种技术称为按需分页（demand paging）。把不经常访问的数据所占的内存空间临时写到硬盘上，这样可以腾出更多的空闲内存空间给经常访问的数据；当CPU访问到不经常访问的数据时，再把这些数据从硬盘读入到内存中，这种技术称为页换入换出（page swap in/out）。这种内存管理技术给了程序员更大的内存“空间”，从而可以让更多的程序在内存中并发运行。

另外，每个进程都拥有属于的虚拟内存（页表），这样各进程之间的内存访问还可以相互独立，互不影响。

总结来看，虚拟内存需要有如下几个特点：

- *透明*：应用开发者可以不必了解底层真实物理内存的硬件细节，且在非必要时也不必关心内核的实现策略， 最小化他们的心智负担；
- *高效*：这层抽象至少在大多数情况下不应带来过大的额外开销；
- *安全*：这层抽象应该有效检测并阻止应用读写其他应用或内核的代码、数据等一系列恶意行为。

我们回顾一下 **计算机组成原理** 课，当 CPU 取指令或者执行一条访存指令的时候，它都是基于虚拟地址访问属于当前正在运行的应用的地址空间。此时，CPU 中的 **内存管理单元** (MMU, Memory Management Unit) 自动将这个虚拟地址进行 **地址转换** (Address Translation) 变为一个物理地址，即这个应用的数据/指令的物理内存位置。也就是说，在 MMU 的帮助下，应用对自己虚拟地址空间的读写才能被实际转化为对于物理内存的访问。

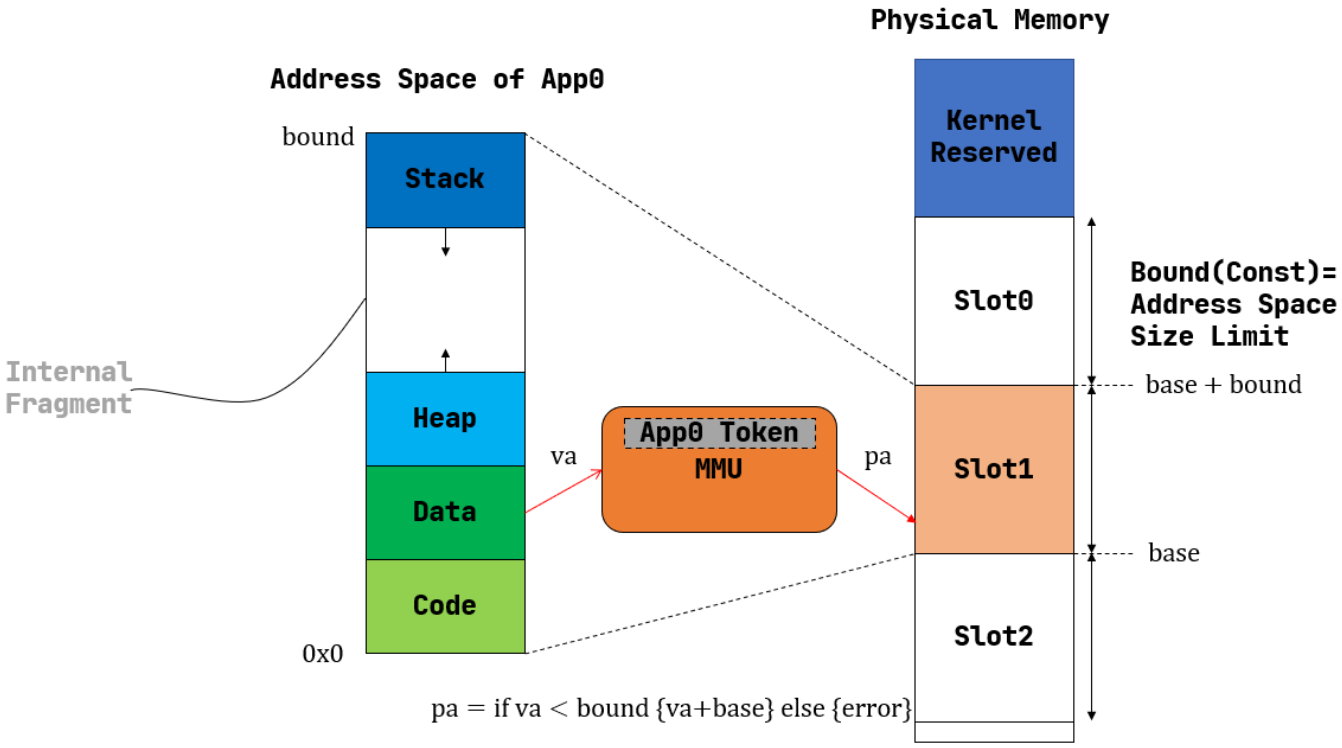
事实上，每个应用的地址空间都存在一个从虚拟地址到物理地址的映射关系。可以想象对于不同的应用来说，该映射可能是不同的，即 MMU 可能会将来自不同两个应用地址空间的相同虚拟地址转换成不同的物理地址。要做到这一点，就需要硬件提供一些寄存器（如satp），软件可以对它进行设置来控制 MMU 按照哪个应用的地址映射关系进行地址转换。于是，将应用的代码/数据放到物理内存并进行管理，建立好应用的地址映射关系，在任务切换时控制 MMU 选用应用的地址映射关系，则是**作为软件部分的内核需要完成的重要工作**。

我们把一个进程能够访问的虚拟内存区域，称为**地址空间**。地址空间只是一层抽象接口，具体的实现包括**如何分配与管理物理内存**和**如何创建物理内存与虚拟内存的映射**两方面。这两方面又有不同的实现策略，下面我们来简单介绍一下。

3.3.1 如何分配与管理物理内存

（下文参考了rcore-tutorial-book-v3，感兴趣的同学可以自行了解）

3.3.1.1 分段内存管理



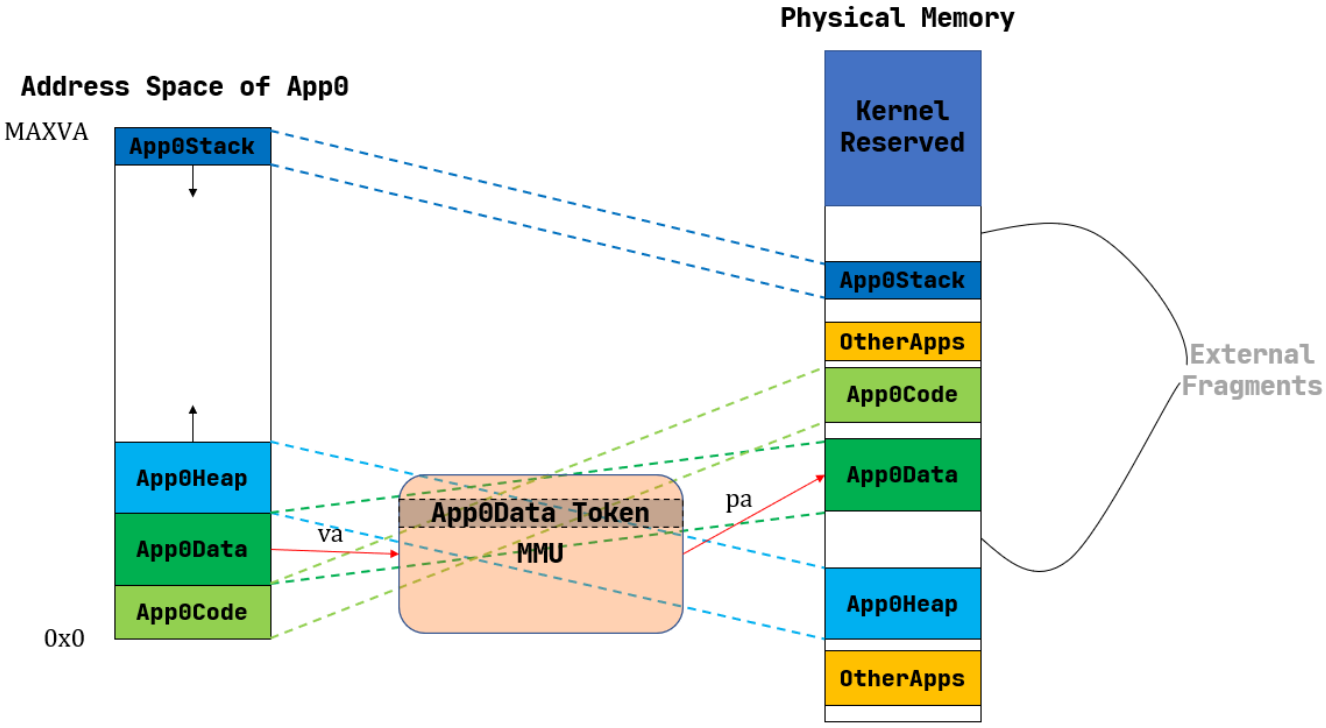
曾经的一种做法如上图所示：每个应用的地址空间大小限制为一个固定的常数 `bound`，也即每个应用的可用虚拟地址区间均为`[0, bound)`。随后，就可以以这个大小为单位，将物理内存除了内核预留空间之外的部分划分为若干个大小相同的 **插槽** (Slot)，每个应用的所有数据都被内核放置在其中一个插槽中，对应于物理内存上的一段连续物理地址区间，假设其起始物理地址为 `base`，则由于二者大小相同，这个区间实际为`[base, base+bound)`。因此地址转换很容易完成，只需检查一下虚拟地址不超过地址空间的大小限制（此时需要借助特权级机制通过异常来进行处理），然后做一个线性映射，将虚拟地址加上 `base` 就得到了数据实际所在的物理地址。

可以看出，这种实现极其简单：MMU 只需要 `base` 和 `bound` 两个寄存器，在地址转换进行比较或加法运算即可；而内核只需要在任务切换时完成切换 `base` 寄存器。在对 Slot 管理方面，可以用一个 **位图** (Bitmap) 来表示，随着应用的新增和退出对应置位或清空。

然而，它的问题在于：可能浪费的内存资源过多。注意到应用地址空间预留了一部分，它是用来让栈得以向低地址增长，同时允许堆往高地址增长（支持应用运行时进行动态内存分配）。每个应用对内存的需求都不同，内核只能按照在它能力范围之内的消耗内存最多的应用的标准来统一指定地址空间的大小，而其他内存需求较

低的应用根本无法充分利用内核给他们分配的这部分空间。但这部分空间又是一个完整的插槽的一部分，也不能再交给其他应用使用。这种在已分配/使用的地址空间内部无法被充分利用的空间就是**内碎片** (Internal Fragment)，它限制了系统同时共存的应用数目。如果应用的需求足够多样化，那么内核无论如何设置应用地址空间的大小限制也不能得到满意的结果。这就是固定参数的弊端：虽然实现简单，但不够灵活。

为了解决这个问题，一种**分段管理**的策略开始被使用，如下图所示：



注意到内核开始以更细的粒度，也就是应用地址空间中的一个逻辑段作为单位来安排应用的数据在物理内存中的布局。对于每个段来说，从它在某个应用地址空间中的虚拟地址到它被实际存放在内存中的物理地址中间都要经过一个不同的线性映射，于是 MMU 需要用一对不同的 `base, bound` 进行区分。这里由于每个段的大小都是不同的，我们也不再能仅仅使用一个 `bound` 进行简化。当任务切换的时候，这些 `base, bound` 对寄存器也需要被切换。

简单起见，我们这里忽略一些不必要的细节。比如应用在以虚拟地址为索引访问地址空间的时候，它如何知道该地址属于哪个段，从而硬件可以使用正确的一对 `base, bound` 寄存器进行合法性检查和完成实际的地址转换。这里只关注分段管理是否解决了内碎片带来的内存浪费问题。注意到每个段都只会在内存中占据一块与它实际所用到的大小相等的空间。堆的情况可能比较特殊，它的大小可能会在运行时增长，但是那需要应用通过系统调用向内核请求。也就是说这是一种**按需分配**，而不再是内核在开始时就给每个应用分配一大块很可能用不完的内存。由此，我们的内存中不再有内碎片了。

尽管内碎片被消除了，但内存浪费问题并没有完全解决。这是因为每个段的大小都是不同的（它们可能来自不同的应用，功能也不同），内核就需要使用更加通用、也更加复杂的连续内存分配算法来进行内存管理，而不能像之前的插槽那样以一个比特为单位。顾名思义，连续内存分配算法就是每次需要分配一块连续内存来存放一个段的数据。随着一段时间的分配和回收，物理内存还剩下一些相互不连续的较小的可用连续块，其中有一些只是两个已分配内存块之间的很小的间隙，它们自己可能由于空间较小，已经无法被用于分配，这就是**外碎片** (External Fragment)。

如果这时再想分配一个比较大的块，就需要将这些不连续的外碎片“拼起来”，形成一个大的连续块。然而这是

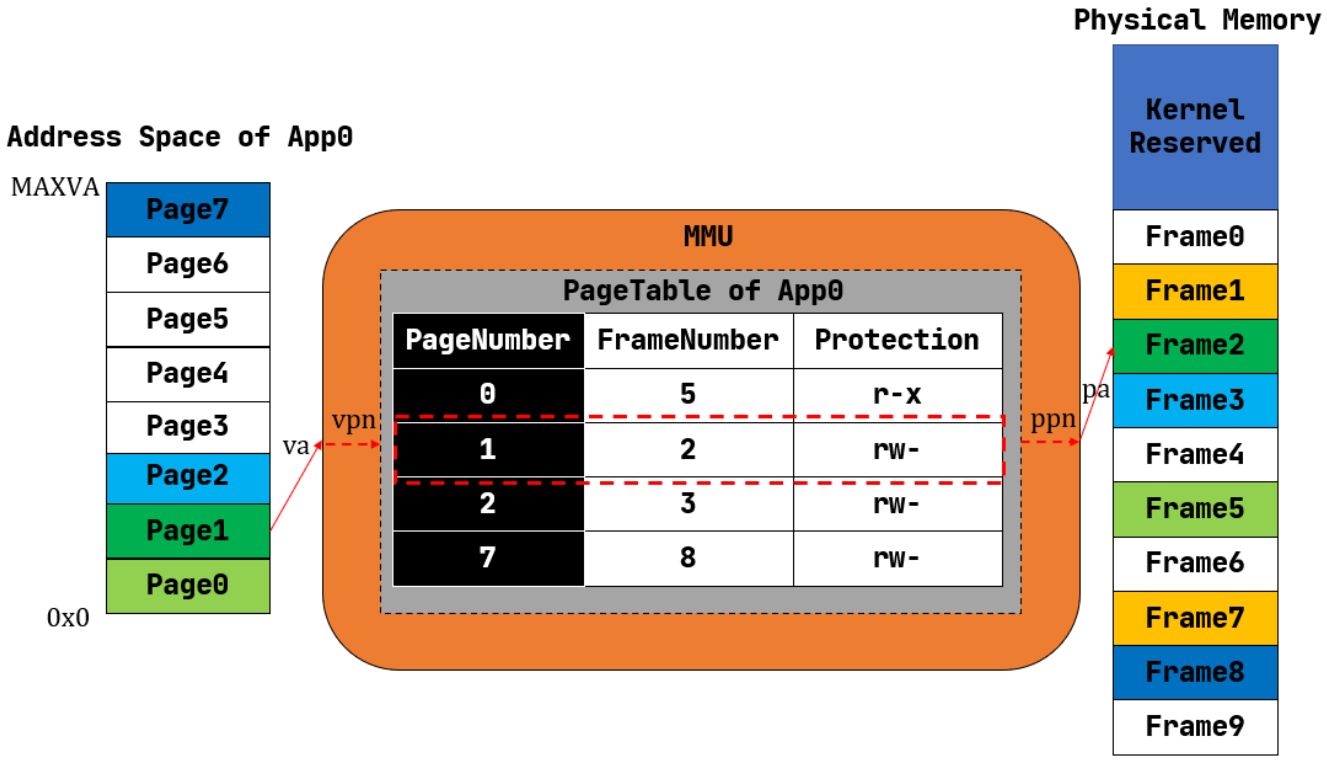
一件开销很大的事情，涉及到极大的内存读写开销。具体而言，这需要移动和调整一些已分配内存块在物理内存上的位置，才能让那些小的外碎片能够合在一起，形成一个大的空闲块。如果连续内存分配算法选取得当，可以尽可能减少这种操作。操作系统课上所讲到的那些算法，包括 `first-fit/worst-fit/best-fit` 或是 `buddy system`，其具体表现取决于实际的应用需求，各有优劣。

那么，分段内存管理带来的外碎片和连续内存分配算法比较复杂的问题可否被解决呢？

3.3.1.2 分页内存管理

仔细分析一下可以发现，段的大小不一是**外碎片**产生的根本原因。使用 Slot 进行内存管理，只需利用类似位图的数据结构维护一组插槽的占用状态，从逻辑上分配和回收都是以一个固定的大小为单位，自然也就不会存在外碎片了。但是这样粒度过大，不够灵活，又在地址空间内部产生了**内碎片**。

若要结合二者的优点的话，就需要内核始终**以一个同样大小的单位来在物理内存上放置应用地址空间中的数据**，这样内核就可以使用简单的插槽式内存管理，使得内存分配算法比较简单且不会产生外碎片；同时，**这个单位的大小要足够小**，从而其内部没有被用到的内碎片的大小也足够小，尽可能提高内存利用率。这便是我们将要介绍的分页内存管理。



如上图所示，内核以页为单位进行物理内存管理。每个应用的地址空间可以被分成若干个(虚拟)**页面** (Page)，而可用的物理内存也同样可以被分成若干个(物理)**页帧** (Frame)，虚拟页面和物理页帧的大小相同。每个虚拟页面中的数据实际上都存储在某个物理页帧上。相比分段内存管理，分页内存管理的粒度更小且大小固定，应用地址空间中的每个逻辑段都由多个虚拟页面组成。而且每个虚拟页面在地址转换的过程中都使用与运行的应用绑定的不同的线性映射，而不像分段内存管理那样每个逻辑段都使用一个相同的线性映射。

为了方便实现虚拟页面到物理页帧的地址转换，我们给每个虚拟页面和物理页帧一个编号，分别称为**虚拟页号** (VPN, Virtual Page Number) 和**物理页号** (PPN, Physical Page Number) 。每个应用都有一个表示地址映射关系

的**页表** (Page Table)，里面记录了该应用地址空间中的每个虚拟页面映射到物理内存中的哪个物理页帧，即数据实际被内核放在哪里。我们可以用页号来代表二者，因此如果将页表看成一个键值对，其键的类型为虚拟页号，值的类型则为物理页号。当 MMU 进行地址转换的时候，虚拟地址会分为两部分（虚拟页号，页内偏移），MMU 首先找到虚拟地址所在虚拟页面的页号，然后查当前应用的页表，根据虚拟页号找到物理页号；最后按照虚拟地址的页内偏移，给物理页号对应的物理页帧的起始地址加上一个偏移量，这就得到了实际访问的物理地址。

在页表中，还针对虚拟页号设置了一组保护位，它限制了应用对转换得到的物理地址对应的内存的使用方式。最典型的如 `rwx`， `r` 表示当前应用可以读该内存； `w` 表示当前应用可以写该内存； `x` 则表示当前应用可以从该内存取指令用来执行。一旦违反了这种限制则会触发异常，并被内核捕获到。通过适当的设置，可以检查一些应用在运行时的明显错误：比如应用修改只读的代码段，或者从数据段取指令来执行。

当一个应用的地址空间比较大的时候，页表中的项数会很多（事实上每个虚拟页面都应该对应页表中的一项，上图中我们已经省略掉了那些未被使用的虚拟页面），导致它的容量极速膨胀，已经不再是像之前那样数个寄存器便可存下来的了，CPU 内也没有足够的硬件资源能够将它存下来。因此它只能作为一种被内核管理的数据结构放在内存中，由内核管理。因此内核与硬件必须严格遵守页表的内存布局规范，才能正确的使用虚拟内存管理机制。

由于分页内存管理既简单又灵活，它逐渐成为了主流的内存管理机制，RISC-V 架构也使用了这种机制。后面我们会基于这种机制，自己动手从物理内存抽象出应用的地址空间来。

3.3.2 如何创建物理内存与虚拟内存的映射

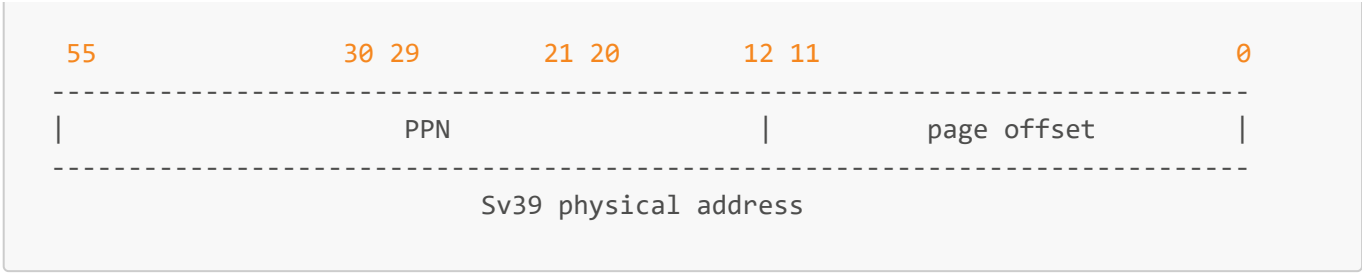
上文我们简单介绍了如何从虚拟地址转换成物理地址：基于虚拟页面到物理页帧的映射，加上偏移量。下面我们将详细介绍在本次试验中我们使用的虚拟地址规范 **Sv39**，请同学们结合【RISC-V 中文手册 10.6 节基于页面的虚拟内存】学习（ riscv特权手册 中详细介绍了 Sv39 的实现标准，感兴趣的同学可以自行阅读）。

Sv39 使用4KB大的基页，页表项的大小是 8 个字节，使用树形的三级页表，称作一级页表（根页表）、二级页表和三级页表。**一级页表的页表项指向一张二级页表，二级页表的页表项指向一张三级页表，三级页表的页表项为目标物理页帧地址。**注意页表是由硬件（MMU）使用的，因此每个页表项都是**物理地址**！在实现页表功能时要注意。

每张页表占据一个页（这里的"页表"指某个一/二/三级页表，而通常我们提到 "某个进程的页表" 时，指的是包含全部三级页表的整体），所以每个页表内可以存放 512 个页表项，全部的三级页表可以表示的虚拟空间大小为512*512*512*4 KB，即 512GB。

Sv39标准下的虚拟地址和物理地址格式如下：

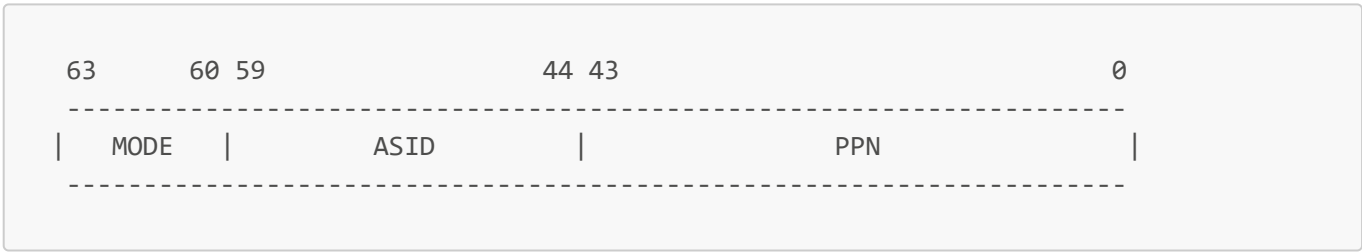
```
 38        30 29        21 20        12 11                          0
-------------------------------------------------------------------------
|   VPN[2]   |   VPN[1]   |   VPN[0]   |          page offset          |
-------------------------------------------------------------------------
                        Sv39 virtual address
```



Sv39 模式定义物理地址有 56 位，虚拟地址有 64 位。但是，虚拟地址的 64 位只有 39 位有效，63-39位在本次实验中（高地址映射）需要为 1 保证地址有效。Sv39 支持三级页表结构，`VPN` 代表虚拟页号，在虚拟地址转换过程中，将 `VPN` 分为三部分，分别对应每一级页表项的 id（详见下文），`PPN` 代表物理页号。物理地址和虚拟地址的低 12 位表示页内偏移（page offset）。**【需要说明的是，按照特权架构说明书，对于 Sv39 来说，需要 63-39 位都等于 38 位，才不会发生缺页异常，而一般规定下，这些位全为 1 时的虚拟内存空间给内核使用，全为 0 的虚拟内存空间给用户态使用】**

虚拟地址翻译为物理地址的完整过程请参考 Virtual Address Translation Process，建议仔细阅读，简化版内容如下：

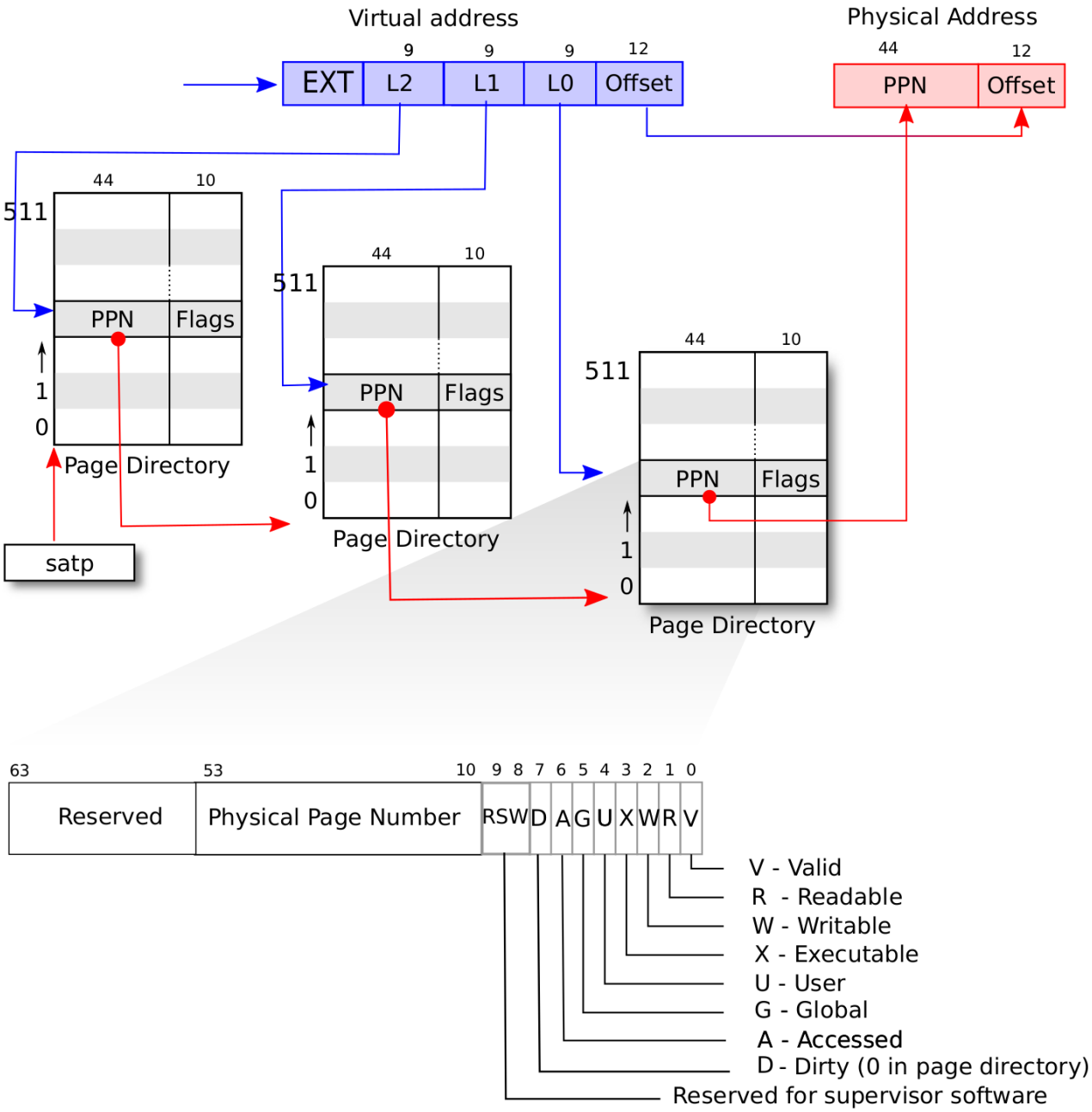
1. 从 satp 的 `PPN` 中获取根页表的物理地址，即 `PPN << 12` 。
   satp（Supervisor Address Translation and Protection，监管者地址转换和保护）是一个 S 模式控制状态寄存器，控制了分页系统，其内容如下所示：



- MODE：可以开启分页并选择页表级数，**8 表示 Sv39 分配方案**，0 表示禁用虚拟地址映射。
- ASID (Address Space Identifier)： 用来区分不同的地址空间，用于硬件检测TLB是否过期。
- PPN (Physical Page Number)：保存了根页表的物理地址，注意 `PPN = physical address >> 12`。M 模式的程序在第一次进入 S 模式之前会把零写入 satp 以禁用分页，然后 S 模式的程序在初始化页表以后会初始化 `satp` 以开启页表。

2. 访问根页表中的第 `VPN[i]` 页表项（初始`i=2`），获取 PTE。
3. 检查PTE的 `V bit`，如果不合法，应该产生 page fault 异常。
4. 检查PTE的 `RWX` bits，如果全部为 0，则从 PTE 中的 PPN 得到的是下一级页表的 PPN （左移 12 位得到物理页帧地址），然后回到第二步，用 `VPN[i-1]` 来访问页表项。否则当前为最后一级页表，PPN 得到的是最终物理页的地址。
5. 将得到最终的物理页帧地址，与偏移地址相加，得到最终的物理地址。

SV39 地址转换的全过程图示如下（来源于 MIT 6.828 课程）：



可以自行尝试一遍以下两个过程，确保自己已理解。

- **【MMU的自动转换】：根据根页表首地址及虚拟地址找到页表项的地址，根据页表项的值转换为物理地址。即已知 satp，va，求相应的页表项地址，并根据页表项的值得到物理地址。**
- **【实验内容】：根据根页表基地址以及虚拟地址找到相应页表项的地址，根据物理地址及映射的权限设置页表项具体的值。即已知 satp，va，pa, 求相应的页表项地址，并根据物理地址设置页表项具体的值。**

> **多级页表**： 为什么我们要把页表分为三级然后使用树的格式来组织页表，而不是直接使用线性表呢？如果我们使用线性表，那么我们可以把虚拟地址到物理地址的映射直接保存在 Virtual_Addr + Offset 的内存中。但是这样的话，如果我们想表示同样大小的地址空间，就需要预留出 512*512*512*8 Byte 的空间来保存页表，这是 1GB！可以类比在C++中提前声明一个超大的数组，虽然读写的时间开销都是 O(1)，但是空间开销极大。有同学可能会想，那为什么不用 vector 的策略呢？因为这样会严重影响虚拟地址转换的效率（每次转换需要 O(n) 的查找时间），与我们“高效”的初衷相悖。所以我们使用树形的多级页表，即满足了按需分配，又得到了“不差”的读写效率。

3.3.3 本实验中的虚拟内存布局

在本实验中，大家需要建立如下内容的页表（具体的代码指导在后文，这里只是给出说明）：

1. 在开启页表后，进入用户进程之前，使用一份临时页表，用于支持内核完成初始化：
   - 将 `0xffffffc000000000` 开始的 16 MB 虚拟地址空间映射到起始物理地址为 `0x80000000` 的 16MB 物理地址空间，用于支持内核在虚拟空间下运行；
   - 对内核起始地址 `0x80000000` 的 16MB 空间做等值映射（将虚拟地址 `0x80000000` 开始的 16 MB 空间映射到起始物理地址为 `0x80000000` 的 16MB 空间），用于支持内核对物理地址进行读写（主要用于访问页表）。同时这一部分映射还支持了在**开启页表后，进入`0xffffffc000000000` 开始的虚拟地址空间前**指令的正常运行，请同学们思考：**这是指哪一部分指令？如果没有建立等值映射，执行这些指令时会发生什么？**
   - 将必要的硬件地址进行映射。
2. 在进程初始化时，为进程创建页表，然后在进入用户进程时切换到用户页表，以后就在不同的用户进程页表之间切换：
   - 将 `0x1001000` 开始的 `4KB` 映射到实际的用户栈物理地址（用户栈的定义与分配见下文）。
   - 将 `0x1000000` 开始的 `4KB` 映射到实际的用户应用程序物理地址。
   - 创建 1. 中的内核映射。

> **内核与应用地址空间的隔离** 目前我们的设计思路 A 是：让每个应用都有一个包含应用和内核的地址空间，并将其中的逻辑段分为内核和用户两部分，分别映射到内核/用户的数据和代码，且分别在 CPU 处于 S/U 特权级时访问。此设计中并不存在一个单独的内核地址空间。 另外的一种设计思路 B 是：对内核建立唯一的内核地址空间存放内核的代码、数据，同时对于每个应用维护一个它们自己的用户地址空间，因此在 Trap 的时候就需要进行地址空间切换（因为控制流从用户进程到了内核），而在任务切换的时候无需进行（因为这个过程全程在内核内完成）。 设计方式 A 的优点在于： Trap 的时候无需切换地址空间，而在任务切换的时候才需要切换地址空间。相对而言，设计方式A比设计方式B更容易实现，在应用高频进行系统调用的时候，采用设计方式A能够避免频繁地址空间切换的开销，这通常源于快表或 cache 的失效问题。 但是设计方式A也有缺点：即内核的逻辑段需要在每个应用的地址空间内都映射一次，这会带来一些无法忽略的内存占用开销，并显著限制了嵌入式平台（如 K210 ）的任务并发数。此外，设计方式 A 无法防御针对处理器电路设计缺陷的侧信道攻击（如 熔断 (Meltdown) 漏洞 ），使得恶意应用能够以某种方式间接"看到"内核地址空间中的数据，使得用户隐私数据有可能被泄露。将内核与地址空间隔离便是修复此漏洞的一种方法。

3.4 M态处理程序

3.4.1 异常中断委托机制

默认情况下，**任何特权级别的所有 Trap 都会在机器模式下处理**，当然机器模式处理程序可以使用 MRET 指令将**陷入**重定向回适当的模式级别。但是为了提高性能，RISC-V 提供了一种硬件机制，那就是**异常中断委托机制**。有了这个机制后，就不再需要软件程序上使用 MRET 指令将 Trap 重定向回想要的模式级别。请同学们仔细阅读 riscv 特权手册 3.1.8 节 和 riscv 中文手册 10.5 节，了解 riscv 的**异常中断委托机制**。

下面我们将以时钟中断处理为例，结合 M 态的代码来进一步理解一下这个过程。在 **lab1 的附录 B：S 模式下的异常**中，我们介绍了 S 模式下时钟中断的处理流程，推荐同学们在理解了下文的内容后，再回过头去看一下 lab1 的附录内容。

系统启动后，qemu 会跳转到 `head.S: _start` 处开始执行，此时系统处于 M 态，我们需要对 M 态的 CSR 寄存器做初始化。在 `head.S:line 40-41` 处，我们设置 mideleg 的第 5 位（从0开始）为 1，表示将编码为 5 的中断（interruption）委托给 S 态执行。当 mtime 大于 mtimecmp 时，时钟中断发生，由于 mtime 与 mtimecmp 是 M 态的寄存器，所以触发的是编码为 7 的 M 态时钟中断。而我们并没有将 M 态时钟中断委托给S态，所以我们会进入M态中断处理程序（`head.S: _mtrap`）。

在M态中断处理程序中，我们首先完成"现场保存"，然后进入用汇编代码写的中断异常处理程序，如果是时钟中断（注意此处是判断中断编码是否为 7，即 M 态时钟中断），则设置 `mip.stip = 1`，这样，我们在执行 `mret`后，硬件就会随即触发 S 态时钟中断，进而触发委托机制，进入 S 态处理。

```
_mtrap:
    # 交换 mscratch 和 sp
    # 相当于使用 M 模式的栈指针
    csrrw sp, mscratch, sp

    # 保存寄存器
    ...

    # 检查 mcause，用最高位判断是中断还是异常
    csrr t0, mcause
    srai t2, t0, 63
    bnez t2, is_int

    # 判断是不是 S 模式主动发出的异常
    li t1, 9
    beq t0, t1, encall_from_s
    j other_trap

    # 判断是不是 machine timer interrupt
    # 是 --> 跳转到 time_interupt
    # 否 --> 跳转到 other_trap
is_int:
    andi t0, t0, 0x7ff
    li t1, 7
    beq t0, t1, time_interupt
    j other_trap

time_interupt:
    # 禁用时钟中断
    li t1, 0x80
    csrc mie, t1

    # 设置 stip 为 1，这样才能进入 S 模式的时钟中断处理
    li t1, 0x20
    csrs mip, t1

    # 恢复寄存器 mepc 和 mstatus
    ld t0, 248(sp)
    ld t1, 256(sp)
    csrw mstatus, t0
    csrw mepc, t1
```

```
    j exit

encall_from_s:
    ...
    j exit


# 我们对其他异常不做任何处理
other_trap:
    ...

exit:
    #恢复寄存器
    ...

    # 交换 mscratch 和 sp
    # 相当于恢复到异常前的 sp
    csrrw sp,mscratch,sp
    mret
```

> 实际上，riscv 标准不建议把 M 态时钟中断委托给 S 态处理。因此触发时钟中断后，先进入 M 态，再进入 S 态，这个看似简单绕了一圈的实现方式是符合 riscv 标准的。 *"Bits 3, 7, and 11 of sip and sie correspond to the machine-mode software, timer, and external interrupts, respectively. Since most platforms will choose not to make these interrupts delegatable from M-mode to S-mode, they are shown as 0 in Figures 1.6 and 1.7.*" —— *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*
> 在有些 riscv 版本中，会存在 mtimecmp 在其他特权级下的版本，如 stimecmp，因此可以直接触发 S 态时钟中断。在我们使用的 riscv 版本中不存在 stimecmp。

3.5 栈切换与页表切换

3.5.1 M态栈切换

在 lab2 中我们提到过，尽管我们在 lab2 已经实现了 U 态和 S 态，但是我们**并没有将内核栈和用户栈进行分离**（sp寄存器始终指向同一个栈），用户进程完全可以访问到内核栈的信息。

```
lab2中各个用户进程的栈分布：

-------------------------------------------------------------------------
|             |   Task0   |   Task1   |   Task2   |   Task3   |   Task4   |
|   Kernel    |           |           |           |           |           |
|             |   Space   |   Space   |   Space   |   Space   |   Space   |
-------------------------------------------------------------------------
^             ^           ^           ^           ^           ^
0x80200000    0x80210000  0x80211000  0x80212000  0x80213000  0x80214000
```

现在我们要实现虚拟地址，虚拟地址为不同应用提供了良好的**隔离**，即内核和用户进程不再能够访问彼此的地址空间，因此我们必须为内核与用户进程分配不同的栈。现在我们来关注一下如果使用了不同的栈，如何在切

换地址空间时进行换栈。我们以**发生时钟中断，从 U 态进入 M 态时的换栈流程**为例：

首先我们要理解为什么需要栈切换：

假设我们已经实现了虚拟地址，现在在 U 态执行用户程序，sp 寄存器指向用户栈，此时发生了时钟中断，我们需要进入 M 态中断异常处理程序。此时的页表是用户程序的页表（即 satp 寄存器指向该用户进程的根页表），而 M 态根本不考虑虚拟地址，只使用物理地址，因此**地址空间发生了变换**，导致我们的程序无法再通过 sp 寄存器访问到刚刚的用户栈（因为此时sp指向用户地址空间的某个地址，而这个地址在实际的物理地址空间**大概率是不存在的，即使可以访问也没有意义**）。因此我们必须换栈，即将 sp 设置为在当前地址空间内有意义的地址。

> 注意！上述原因只是描述了需要换栈的必要性，但在 os 的设计理念里，不是因为有了虚拟地址才需要换栈，恰恰相反，正是因为换栈不足以保证安全性，才出现了虚拟地址技术。栈分离和虚拟地址技术都是为了保证系统的安全而存在的，希望同学们理解这个逻辑。

栈切换用到了两个重要的 csr 寄存器，`mscratch/sscratch`，这两个寄存器没有明确的作用，而是作为中转寄存器出现的。在时钟中断发生时，我们需要将 Trap 上下文保存在 M 态的栈上，因此需要一个寄存器暂存 M 态栈地址，并以它作为基地址指针来依次保存 Trap 上下文的内容。但是所有的通用寄存器都不能够用作基地址指针，因为它们都需要被保存，如果覆盖掉它们，就会影响后续应用控制流的执行。事实上我们缺少了一个重要的中转寄存器，而 `mscratch/sscratch` CSR 正是为此而生。

在进入 `_start` 后，我们将 stack_top 保存在 mscratch 中。

```
// head.S
35  # 用 mscratch 存储 M 模式下的栈指针
36  la t1, stack_top
37  csrw mscratch, t1
```

> 关于stack_top的地址： 有同学可能会奇怪，明明我们在 `vmlinux.lds` 中定义基地址是 `Line 14: . = 0xffffffc000000000`，这说明在编译过程中，所有的地址都应该在虚拟地址空间中，为什么在这里访问 `stack_top` 得到的是物理地址呢？ （如果你没有觉得这里奇怪的话，那你更应该仔细理解一下这段内容） 这是因为在编译器对代码进行编译时，采用了相对寻址的寻址方式，即根据当前的 pc 值加上一个偏移量计算每个地址的实际位置。通过指令 `riscv64-unknown-linux-gnu-objdump -d head.o` 对 `head.o` 反汇编可以看出，计算 `stack_top` 的地址时，实际使用的指令是 20: 00000317 auipc t1,0x0 24: 00030313 mv t1,t1 28: 34031073 csrw mscratch,t1 由于对于`head.o`单个文件来说，此时还没有进行链接，所以编译器也无法确定需要偏移多少才能得到`stack_top` 的地址，所以 `auipc` 指令的偏移量为 0（这是编译原理的相关内容，感兴趣的同学可以自行了解）。但是我们可以从上面的反汇编结果看出，编译器确实是通过**相对寻址**得到 `stack_top` 的地址。 那么现在我们可以简单的梳理一下为什么我们可以得到 stack_top 的物理地址。注意我们在 lab3 中没有OpenSBI，所以我们把整个OS（包括 M 态管理程序）都放在了 `0x80000000` 起始的物理地址上。编译器在编译链接时，根据vmlinux.lds，基地址是`0xffffffc000000000`，即第一条指令的地址是`0xffffffc000000000`，因此编译器通过相对寻址的方式计算出了通过 `pc` 得到 `stack_top` 地址的偏移量`offset`。然而在实际运行时，我们的第一条指令在 `0x80000000`，因此当实际计算 `stack_top` 的地址时，我们通过`0x8000xxxx` （`auipc` 指令的地址）+ `offset` 得到了 `stack_top` 以 `0x80000000` 为基地址的“相对地址”，而这“恰好”就是 `stack_top` 的物理地址。为什么对“恰好”加引号呢？那是因为使用相对寻址的情况下，不论我们把代码加载到哪块地址

> 上，都可以正确计算出每个符号的地址。 只是这样的话，就需要同学们注意，在 pc 不同的情况下，框架中的符号的值也会不同。比如，在创建页表并开启虚拟地址之前，`stack_top`是物理地址，但是在开启虚拟地址之后（更确切地说，是进入以`0xffffffc000000000` 为基地址的虚拟地址空间后），如果在代码中再次访问 `stack_top`，得到的就是虚拟地址了。如果同学在完成 lab3 的过程中需要用到某些预定义好的符号，一定要注意此时访问该符号得到的是物理地址还是虚拟地址！

通过 `_mtrap` 的代码可以看出，在进入 M 态异常中断处理程序后，我们首先交换 `sp` 和 `mscratch` 的值。这相当于把 `sp` 从用户栈指向了 M 态的栈，而把原来用户栈的栈顶地址保存在了 mscratch 中。

```
// head.S
128  _mtrap:
129      # 交换 mscratch 和 sp
130      # 相当于使用 M 模式的栈指针
131      csrrw sp, mscratch, sp
```

在退出 `_mtrap` 时，我们再次交换 `mscratch` 与 `sp`，就完成了用户栈与M态栈的切换。

```
// head.S
276      # 交换 mscratch 和 sp
277      # 相当于恢复到异常前的 sp
278      csrrw sp,mscratch,sp
279      mret
```

> 为什么不用硬编码的方式保存每个应用的栈地址与内核栈地址呢？因为我们需要的不是栈底地址，而是栈顶地址，而栈顶地址是变化的。

请同学们仔细理解这一部分内容，如果有疑问及时询问或者和同学讨论。搞清楚上面的栈切换逻辑之后，我们开始介绍内核栈与用户栈的切换过程。

3.5.2 用户栈与内核栈

我们以**进程调度过程**为例，解释在实现了内核栈与用户栈后，代码的执行逻辑与 `lab2` 发生了哪些变化。

在 lab2 中，我们在不同的用户进程之间切换时，在 `__swtich_to` 函数中通过切换 `sp` 完成了栈的切换。实际上我们此时完成的是“内核栈”的切换，因为在一个进程被调出时，控制流是在内核中的，此时的栈上保存的是该进程的内核状态（即 `__switch_to` 的前半部分所做的）。因此当一个进程被调入时，也应该恢复目标进程在内核中的控制流（即`__switch_to`的后半部分所做的），然后再由内核控制流切换回用户进程的控制流（即执行`sret`）。

不同点1：在实现了内核栈与用户栈后，与从 U 态进入 M 态需要换栈一样，从 U 态进入 S 态也需要换栈。我们借助`sscratch` 寄存器完成这一步。进入内核时，我们交换 `sp` 与 `sscratch`，交换后 `sp` 指向该进程的内核栈，而 `sscratch` 保存着该进程的用户栈栈顶地址。从内核进入用户进程时，再次交换，就完成了栈切换。

不同点2：在进行进程切换时，我们需要在切换 sp 的同时切换 sscratch，这样才能在从 S 态进入 U 态时正确恢复目标进程的控制流。因此我们在 task_struct 中添加了一项为sscratch，用于进程切换时保存sscratch信息。

```
// task_manager.h

/* 进程数据结构 */
struct task_struct {
  ...
  uint64_t sscratch; // 保存 sscratch
  ...

};
```

> 借用一下数学中的数学归纳法的思想，如果在一个进程进入内核时，sscratch 确实存的是它对应的内核栈地址，那么我们上面的描述就都是成立的。那么在进程第一次启动时，sscratch 应该存的是什么呢？sp 呢？在进程初始化时，我们需要把 sp 指向该进程的内核栈，sscratch 指向该进程的用户栈即可。

> 在虚拟地址空间中，每个进程的用户栈地址都是相同的，在我们的框架中，我们把虚拟栈地址定义为 0x1001000 开始的 4KB 地址空间。这就是虚拟地址的一个重要作用，对于任何用户进程，它都好像是独占了CPU一样。再次强调，每个用户态进程都占有这些相同的虚拟内存地址是不会冲突的，因为每个用户态进程都有属于自己的页表（即 satp 寄存器在不同用户态程序下是不同的），这样虽然每个用户态进程都占有相同的虚拟地址空间，但可以通过不同的页表来使得虚拟内存地址映射到的物理内存地址不发生冲突。

3.5.3 页表切换

我们提到过，在我们的虚拟内存管理机制下，在 Trap 时不需要进行页表切换，因为内核和用户进程实际上使用同一张页表，而在进程调度时需要从当前进程页表切换到目标进程页表。

因此我们需要在进程切换的过程中完成页表的切换。和 sscratch 一样，我们需要在 task_struct 中添加一项 satp，用于进程切换时保存 satp 信息。

```
/* 进程数据结构 */
struct task_struct {
  ...
  uint64_t satp;     // 保存 satp

};
```

注意页表切换的时机：在 __switch_to 函数中，我们就已经完成了控制流的转换，因此页表切换需要在 __switch_to 中完成。

> 内核虚拟地址空间的“平滑切换”： 我们在每一个用户进程的页表中都建立了**相同**的内核地址空间映射，这样我们就可以保证，在切换用户进程的页表后，能够“平滑”地继续运行内核。可以想象一下，如果每个进程的页表中，内核的地址空间映射不同，那么在我们执行完页表切换指令后，因为 pc 还指向原来的地址空间，下一条指令（pc+4）就无效了。 这是在内核实现过程中必须要考虑的问题。如果我们采用设计模式 B（即内核与用户进程使用不同的页表），那么也需要在内核和用户进程页表中，留出一部分建立相同的映射，在这个相同的地址空间中完成页表的切换。这通常被称为“跳板机制”。

3.6 内存布局

在实现了页表和用户栈/内核栈之后，我们的内存布局如下图所示：

```
lab3中各个部分**在物理空间中**的分布：

-------------------------------------------------------------------------
|             |     User     |        |   Kernel  |   M mode  |  OS Init  |
|   Kernel    |              |  .bss  |   Init    |           |   Page    |
--->
|             |    Program   |        |   Stack   |   Stack   |   Table   |
-------------------------------------------------------------------------
^             ^              ^                    ^           ^
0x80000000    0x80004000     0x8000a000          0x8000c000 0x8000d000


        ---------------------------------------------------------------------------
-----
        |   Task0   |   Task0   |   Task0   |  Task1   |   Task1   |   Task1   |
 --->   |   Kernel  |   User    |   Page    |  Kernel  |   User    |   Page    |
... ...
        |   Space   |   Space   |   Table   |  Space   |   Space   |   Table   |
        ---------------------------------------------------------------------------
------


注：在实际的操作系统中，通常会在可写的数据段与其他段之间加入guard page，其实就是一个空页，但是这个空页不在页表中，因此一旦访问就会触发异常，这样就对不同的段起到了保护作用。
```

从 `0x80000000` 开始，到 `0x80004000` 是内核的代码和数据段，具体来说是 `.text`、`.rodata` 和 `.data` 段，关于这几个段的内容与其含义，可以参考这篇博客，~~不过这在我们的os课上不重要，是编译原理的知识~~。

在 lab3 中，我们把用户应用程序的代码单独拿出来（而不是和内核代码混在一起），放在了 `0x80004000` 到 `0x8000a000` 的 `20KB` 空间中（感兴趣的同学可以看一下 `vmlinux.lds` 文件和 `users.S` 文件，看看我们做了什么），每个用户应用程序占用 `4KB (0x1000 Byte)` 的内存。之后的 `4KB` 是内核的 `.bss` 段。

然后从 `0x8000b000` 到 `0x8000c000` 是我们为内核初始化分配的栈，在代码中，大家需要在进入S态后手动将 `sp` 指向这块区域的高地址（栈底在高地址，栈顶在低地址），用于支持内核初始化过程中的函数调用。

从 `0x8000c000` 到 `0x8000d000` 是我们为M态分配的栈。

`0x8000d000` 之后，我们用了几个页来分配 os 初始化过程中使用的页表。之后的空间我们用于分配用户应用程序的页表、应用的内核栈与用户栈。这部分需要同学们写代码完成。

3.7 修改 head.S（30%）

3.7.1 修改 S 模式下系统启动部分代码

默认情况下 MMU 未被使能（启用），此时无论 CPU 位于哪个特权级，访存的地址都会作为一个物理地址交给对应的内存控制单元来直接访问物理内存。我们可以通过修改 S 特权级的一个名为 `satp` 的 CSR 来启用分页模式，在这之后 S 和 U 特权级的访存地址会被视为一个虚拟地址，它需要经过 MMU 的地址转换变为一个物理地址，再通过它来访问物理内存；而 M 特权级的访存地址，我们可设定是内存的物理地址。

> M 特权级的访存地址被视为一个物理地址还是一个需要经历和 S/U 特权级相同的地址转换的虚拟地址取决于硬件配置，在这里我们不会进一步探讨。

- 在 `_supervisor` 开头先设置 `satp` 寄存器为0，暂时关闭 MMU
- 设置 `sp` 的值为 `init_stack_top` 的物理地址
- 调用 `paging_init` 函数建立页表
- 设置 `satp` 的值以打开 MMU，注意我们是从 `&_end` 开始分配的页表，所以此时根页表的地址为 `_end`。此时可以不用考虑 `satp` 的 `asid` 域，保持 `asid=0` 即可，可以想一下为什么?
- 执行 `sfence.vma` 指令同步虚拟内存相关映射。这条命令的作用为刷新`TLB`。
- 设置 `stvec` 为异常处理函数 `trap_s` 在**虚拟地址空间下**的地址
  提示： `vmlinux.lds` 中规定将内核放在物理内存 `0x80000000`、虚拟内存 `0xffffffc000000000` 的位置，因此物理地址空间下的地址 `x` 在虚拟地址空间下的地址为 `x - 0x80000000 + 0xffffffc000000000`。
- 记录 `start_kernel` 在**虚拟地址空间下**的地址，加载到 s0 寄存器中
- 设置 `sp` 的值为**虚拟地址空间**下的 `init_stack_top`
- 设置 `sstatus.spp = 0`，使得进入用户程序时 CPU 处于U态
- 使用 `jr` 指令跳转到 `start_kernel`（读取之前记录在寄存器中的值）

请注意，在执行完 `jr` 以后，我们的内核就已经运行在虚拟地址空间中了，即 `0xffffffc000000000` 的位置，而不应该再回到 `0x80000000` 的位置。

请在下方代码框中补充完整你的代码：

```
_supervisor:
    # 1. 在 _supervisor 开头先设置 satp 寄存器为0，暂时关闭 MMU
    csrw satp, x0

    # 2. 设置 sp 的值为 init_stack_top 的物理地址
    la sp, init_stack_top

    # 3. 调用 paging_init 函数建立页表
    call  paging_init
```

```
    # 4. 设置 satp 的值以打开 MMU
    # 提示：从 _end 开始为页表分配物理空间，即根页表的地址为 _end
    # satp [63:60 mode] [59:44 ASID] [43:0 PPN]
    la t0, _end
    srli t0, t0, 12
    li t1, 8
    slli t1, t1, 60
    or t0, t1, t0
    csrw satp, t0

    # 5. 执行 sfence.vma 指令同步虚拟内存相关映射
    sfence.vma

    # 6. 设置 stvec 为异常处理函数 trap_s 在虚拟地址空间下的地址
    # 提示：vmlinux.lds 中规定将内核放在物理内存 0x80000000、虚拟内存
0xffffffc000000000 的位置，因此物理地址空间下的地址 x 在虚拟地址空间下的地址为 x -
0x80000000 + 0xffffffe00000000。
    li t3, 0xffffffc000000000
    li t2, 0x80000000
    sub t3, t3, t2                          #store diff in t3
    la t0, trap_s
    add t0, t0, t3
    csrw stvec, t0

    # 7. 记录 start_kernel 在虚拟地址空间下的地址，加载到 s0 寄存器中
    la t0, start_kernel
    add s0, t0, t3

    # 8. 设置 sp 的值为虚拟地址空间下的 init_stack_top
    la t0, init_stack_top
    add sp, t0, t3

    # 9. 设置sstatus.spp = 0
    li t0, 256
    csrc sstatus, t0

    # 9. 使用 jr 指令跳转到 start_kernel（读取之前记录在寄存器中的值）
    jr s0
```

注意：你也许使用 GDB 打印出 `&_end` 的值是 `0xffffffc00000X000`，这是因为调试符号的地址是在编译时确定的，所以无论什么时候，用 GDB 查看符号值，都是在虚拟地址空间中的值。但是这不一定是它的“真实值”！

3.7.2 阅读 M 模式下异常处理代码 `_mtrap`

该小节代码已提供，仅需理解。下面简介 M 模式的栈空间是如何工作的。

- M 模式下依然使用物理地址，使用虚拟地址将导致内存访问错误。
- `mscratch` 寄存器是 M 模式下专用的临时寄存器。通常，它就用于保存 M mode 下上下文物理空间的地址。lds 文件中分配出了 1 个 page 的空间用于储存进程上下文，其顶部标记为 `stack_top`，在 head.S 进入 S mode 之前的适当位置，将 `mscratch` 寄存器设置为 `stack_top` 的物理地址。

- 在 M 模式异常处理函数 `trap_m` 的开头，将 `mscratch` 与 `sp` 寄存器的值交换，使用预先分配好的栈空间保存 `x1-x31` 寄存器。
- 在 `trap_m` 返回前，恢复保存好的寄存器，之后将 `mscratch` 与 `sp` 寄存器的值重新交换回来。

3.8 实现映射机制 `vm.c`（40%）

3.8.1 创建映射（30%）

为了增加一个虚拟地址到物理地址的映射，根据 Sv39 分配方案中的地址翻译规则，首先需要**根据根页表基地址以及虚拟地址找到相应页表项的地址，并在该过程中为页表分配物理页面，随后根据物理地址及映射的权限设置页表项具体的值**，使得后续 MMU 工作时能够根据根页表基地址和虚拟地址得到正确的物理地址。

在 `vm.c` 中编写函数 `create_mapping(uint64 *pgtbl, uint64 va, uint64 pa, uint64 sz, int perm)`，用作建立页表映射的统一接口，其中参数作用如下：

- `pgtbl` 为根页表的基地址
- `va`，`pa` 分别为需要映射的虚拟、物理地址的基地址
- `sz` 为映射的大小，单位为字节
- `perm` 为映射的读写权限

我们从根页表出发，每一次创建映射时要依次建立二级页表和三级页表，如果二级页表和三级页表不存在，那就分配一块空间给二级页表和三级页表使用即可。本实验中可以自由管理分配 `_end` 地址之后的物理内存。

我们在代码中提供了相关的函数和宏供大家使用：

1. `alloc_page()`：分配一个物理页帧并返回该页的**物理地址**。
2. `alloced_page_num()`：返回已经分配的物理页帧数量。
3. `PHYSICAL_ADDR(x)`：将 x 转换为物理地址（如果已经是物理地址，则不变）。
4. `VIRTUAL_ADDR(x)`：将 x 转换为 `0xffffffc000000000` 地址空间下的虚拟地址。
5. `PAGE_SIZE`：unsigned long 4096，表示一个页的大小。
6. `PTE_V,PTE_R,PTE_W,PTE_X,PTE_U`：Sv39 规范下页表项的控制位，可以通过 `PTE | PTE_?` 设置页表项权限。

注意：页表中的页表项指向的地址始终是**物理地址**。

请在下方代码框中补充完整你的代码：

```
void create_mapping(uint64_t *pgtbl, uint64_t va, uint64_t pa, uint64_t sz, int perm){
  // pgtbl 为根页表的基地址
  // va, pa 分别为需要映射的虚拟、物理地址的基地址
  // sz 为映射的大小，单位为字节
  // perm 为映射的读写权限
  // pgtbl 是根页表的首地址，它已经占用了一页的大小，不需要再分配
  //  (调用create_mapping前需要先分配好根页表)
    while(sz>0){
```

```
        uint64_t* pgtbl2;
        uint64_t* pgtbl3;
        // 1. 通过 va 得到一级页表项的索引
        // 2. 通过 va 得到二级页表项的索引
        // 3. 通过 va 得到三级页表项的索引
        uint64_t VPN1 = (va>>30)&(0x1FF);
        uint64_t VPN2 = (va>>21)&(0x1FF);
        uint64_t VPN3 = (va>>12)&(0x1FF);
        // 4. 如果一级页表项不存在，分配一个二级页表
        // 5. 设置一级页表项的内容
        uint64_t entry;
        entry = pgtbl[VPN1];
        if(!(entry&PTE_V)){
          pgtbl2 = (uint64_t*)alloc_page();
          pgtbl[VPN1] = (((uint64_t)pgtbl2>>12)<<10)|PTE_V;
        }else{
          pgtbl2 = (uint64_t*)((entry>>10)<<12);
        }
        // 6. 如果二级页表项不存在，分配一个三级页表
        // 7. 设置二级页表项的内容
        entry = pgtbl2[VPN2];
        if(!(entry&PTE_V)){
          pgtbl3 = (uint64_t*)alloc_page();
          pgtbl2[VPN2] = (((uint64_t)pgtbl3>>12)<<10)|PTE_V;
        }else{
          pgtbl3 = (uint64_t*)((entry>>10)<<12);
        }
        // 8. 设置三级页表项的内容
        pgtbl3[VPN3] = ((pa>>12)<<10)|(PTE_V|(perm));
        sz-=PAGE_SIZE;
        va+=PAGE_SIZE;
        pa+=PAGE_SIZE;
    }
}
```

3.8.2 设置映射（10%）

在 `vm.c` 中编写 `paging_init` 函数。

- 创建内核的虚拟地址空间，调用 `create_mapping` 函数将虚拟地址 `0xffffffc000000000` 开始的 `16 MB` 空间映射到起始物理地址为 `0x80000000` 的 `16MB` 空间，`PTE_V | PTE_R | PTE_W | PTE_X` 为映射的读写权限
- 对内核起始地址 `0x80000000` 的`16MB`空间做等值映射（将虚拟地址 `0x80000000` 开始的 `16 MB` 空间映射到起始物理地址为 `0x80000000` 的 `16MB` 空间），`PTE_V | PTE_R | PTE_W | PTE_X` 为映射的读写权限。
- 修改对内核空间不同 section 所在页属性的设置，完成对不同 section 的保护，其中 `text` 段的权限为 `r-x`, `rodata` 段为 `r--`, 其他段为 `rw-`，注意上述两个映射都需要做保护。
- 将必要的硬件地址（如 `0x10000000` 为起始地址的 UART ）进行等值映射 (可以映射连续 1MB 大小)，无偏移，`PTE_V | PTE_R | PTE_W` 为映射的读写权限

> 提示：lds文件中定义的变量，在C代码中使用时应该使用&取址。

请在下方代码框中补充完整你的代码：

```
void paging_init()
{
  // 在 vm.c 中编写 paging_init 函数，该函数完成以下工作：
  // 1. 创建内核的虚拟地址空间，调用 create_mapping 函数将虚拟地址 0xffffffc000000000 开始的 16 MB 空间映射到起始物理地址为 0x80000000 的 16MB 空间，PTE_V | PTE_R | PTE_W | PTE_X 为映射的读写权限。
  // 2. 对内核起始地址 0x80000000 的16MB空间做等值映射（将虚拟地址 0x80000000 开始的 16 MB 空间映射到起始物理地址为 0x80000000 的 16MB 空间），PTE_V | PTE_R | PTE_W | PTE_X 为映射的读写权限。
  // 3. 修改对内核空间不同 section 所在页属性的设置，完成对不同section的保护，其中text 段的权限为 r-x，rodata 段为 r--，其他段为 rw-，注意上述两个映射都需要做保护。
  // 4. 将必要的硬件地址（如 0x10000000 为起始地址的 UART ）进行等值映射（ 可以映射连续 1MB 大小 ），无偏移，PTE_V | PTE_R | PTE_W 为映射的读写权限

  // 注意：paging_init函数创建的页表只用于内核开启页表之后，进入第一个用户进程之前。进入第一个用户进程之后，就会使用进程页表，而不再使用 paging_init 创建的页表。

  uint64_t *pgtbl = alloc_page();
  create_mapping(pgtbl, 0xffffffc000000000, 0x80000000, 16*1024*1024, PTE_V | PTE_R | PTE_W | PTE_X);
  create_mapping(pgtbl, 0x80000000, 0x80000000, 16*1024*1024, PTE_V | PTE_R | PTE_W | PTE_X);
  create_mapping(pgtbl, 0x10000000, 0x10000000, 1*1024*1024, PTE_V | PTE_R | PTE_W);
  create_mapping(pgtbl, (uint64_t)&text_start, (uint64_t)&text_start, (uint64_t)&rodata_start-(uint64_t)&text_start, PTE_R|PTE_X);
  create_mapping(pgtbl, (uint64_t)&rodata_start, (uint64_t)&rodata_start, (uint64_t)&data_start-(uint64_t)&rodata_start, PTE_R);
  create_mapping(pgtbl, (uint64_t)&data_start, (uint64_t)&data_start, (uint64_t)&user_program_start - (uint64_t)&data_start, PTE_R|PTE_W);
}
```

3.8.3 测试

如果需要使用 gdb 调试，请参考附录【断点不生效解决方法】修改 `vmlinux.lds`，完成本阶段的调试。注意完成调试后需要把 `vmlinux.lds` 改回原来的值。

此时我们的 os 应该可以执行进程调度前的代码了，执行 `make run`，如果正确输出了 `ZJU OSLAB 3 学号 姓名`、进程初始化信息以及第一次调度信息，则说明页表建立没有问题。

```
ZJU OSLAB 3 学号 姓名
task init...
[PID = 0] Process Create Successfully!
```

```
[PID = 1] Process Create Successfully!
[PID = 2] Process Create Successfully!
[PID = 3] Process Create Successfully!
[PID = 4] Process Create Successfully!
task[0]: counter = 1, priority = 4 <-- next
task[1]: counter = 4, priority = 5
task[2]: counter = 3, priority = 2
task[3]: counter = 4, priority = 1
task[4]: counter = 5, priority = 4
Instruction page fault! epc = 0x0000000001000000
```

3.9 修改进程初始化代码 `task_manager.c`（10%）

lab3 与 lab2 不同的地方在于我们区分了用户栈和内核栈，并实现了虚拟内存管理。因此在进程初始化时，我们需要完成下面几个步骤：

1. 为进程分配内核栈（已完成）
   在lab2中我们通过 `Kernel_Page+Offset` 的方式得到了进程的栈地址，现在我们可以直接调用 `alloc_page()`。

```
// task_manager.c
26    struct task_struct* new_task =(struct task_struct*)
(VIRTUAL_ADDR(alloc_page()));
27    new_task->state = TASK_RUNNING;
28    new_task->counter = 0;
29    new_task->priority = 5;
30    new_task->blocked = 0;
31    new_task->pid = i;
32    task[i] = new_task;
33    task[i]->thread.sp = (uint64_t)task[i] + PAGE_SIZE; // 内核栈的栈底
34    task[i]->thread.ra = (uint64_t)__init_sepc;
```

2. 为进程分配用户栈
3. 为进程建立页表
   - 建立用户栈映射：从 `0x1001000` 映射到用户栈的物理地址，大小为`4KB`。
   - 建立代码段映射：从 `0x1000000` 映射到用户代码的物理地址，大小为`4KB`。用户代码位置已经由 `task_addr` 计算出来。
   - 建立内核映射。为什么这里我们还需要对内核进行等值映射呢？与初始化内核时建立等值映射的原因相同吗？在初始化内核时建立等值映射是为了执行**“开启页表后，进入`0xffffffc000000000`地址空间前”**的代码；为用户进程页表建立内核的等值映射，是为了方便页表的建立。

请在下方代码框中补充完整你的代码：

```
```

```
// TODO: 完成用户栈的分配，并创建页表项，将用户栈映射到实际的物理地址
    // 1. 为用户栈分配物理页面，使用alloc_page函数
    uint64_t stack_pa = alloc_page();
    // 2. 为用户进程分配根页表，使用alloc_page函数
    uint64_t root_pt = alloc_page();
    // 3. 将task[i]->sscratch指定为虚拟空间下的栈地址，即0x1001000 + PAGE_SIZE（注意
栈是从高地址到低地址使用的）
    task[i]->sscratch = 0x1001000 + PAGE_SIZE;
    // 4. 正确设置task[i]->satp，注意设置ASID
    task[i]->satp = ((uint64_t)8<<60)|(root_pt>>(uint64_t)12)|((uint64_t)(i&0xFFF)
<<44);
    // 5. 将用户栈映射到实际的物理地址，使用create_mapping函数
    create_mapping((uint64_t *)root_pt,0x1001000,stack_pa,PAGE_SIZE,PTE_V | PTE_R
| PTE_W | PTE_U );
    // 6. 将用户程序映射到虚拟地址空间，使用create_mapping函数
    create_mapping((uint64_t *)root_pt, 0x1000000,task_addr,PAGE_SIZE, PTE_V |
PTE_R | PTE_W | PTE_X | PTE_U);
    // 7. 将虚拟地址 0xffffffc000000000 开始的 16 MB 空间映射到起始物理地址为
0x80000000 的 16MB 地址空间，注意此时 &rodata_start、... 得到的是虚拟地址还是物理地
址？我们需要的是什么地址？
    create_mapping((uint64_t *)root_pt, 0xffffffc000000000, 0x80000000,
16*1024*1024, PTE_V | PTE_R | PTE_W | PTE_X);
    // 8. 对内核起始地址 0x80000000 的16MB空间做等值映射（将虚拟地址 0x80000000 开始的
16 MB 空间映射到起始物理地址为 0x80000000 的 16MB 空间），PTE_V | PTE_R | PTE_W |
PTE_X 为映射的读写权限。
    create_mapping((uint64_t *)root_pt, 0x80000000, 0x80000000, 16*1024*1024,
PTE_V | PTE_R | PTE_W | PTE_X);
    // 9. 修改对内核空间不同 section 所在页属性的设置，完成对不同section的保护，其中
text段的权限为 r-x，rodata 段为 r--，其他段为 rw-，注意上述两个映射都需要做保护。
    create_mapping((uint64_t *)root_pt,(uint64_t)&text_start ,
PHYSICAL_ADDR((uint64_t)&text_start),
                   (uint64_t)&rodata_start-(uint64_t)&text_start, PTE_R|PTE_X);
    create_mapping((uint64_t *)root_pt, (uint64_t)&rodata_start,
PHYSICAL_ADDR((uint64_t)&rodata_start),
                   (uint64_t)&data_start-(uint64_t)&rodata_start, PTE_R);
    create_mapping((uint64_t *)root_pt, (uint64_t)&data_start,
PHYSICAL_ADDR((uint64_t)&data_start),
                   (uint64_t)&user_program_start - (uint64_t)&data_start,
PTE_R|PTE_W);
    // 10. 将必要的硬件地址（如 0x10000000 为起始地址的 UART ）进行等值映射（ 可以映射
连续 1MB 大小 ），无偏移，PTE_V | PTE_R | PTE_W 为映射的读写权限
    create_mapping((uint64_t *)root_pt, 0x10000000, 0x10000000, 1*1024*1024, PTE_V
| PTE_R | PTE_W);
```

> 之前已经提到过一些符号的值在不同的地址空间算出来是不一样的，什么时候需要使用 PHYSICAL_ADDR，请仔细考虑！

3.10 修改 entry.S

结合 3.5 节内容，完成用户栈和内核栈的换栈汇编代码。

1. 在 `trap_s` 中加入换栈操作
   请在下方代码框中补充完整你的代码：

```
trap_s:
    # TODO: swap sp and sscratch, now sp point to kernel stack, sscratch point to
user stack
    csrr t0, sscratch
    csrw sscratch, sp
    mv sp, t0

...

    # TODO: swap sp and sscratch, now sp point to user stack, sscratch point to
kernel stack
    csrr t0, sscratch
    csrw sscratch, sp
    mv sp, t0

    sret
```

2. 在 `__switch_to` 中加入对 `sscratch` 的保存和换页表操作
   请在下方代码框中补充完整你的代码：

```
.globl __switch_to
__switch_to:

...

    # TODO: save sscratch into prev->sscratch
    csrr t0, sscratch
    sd t0, 14*reg_size(a3)
    # TODO: save satp into prev->satp
    csrr t0, satp
    sd t0, 15*reg_size(a3)
...

    # TODO: load sscratch from next->sscratch
    ld t0, 14*reg_size(a4)
    csrw sscratch, t0
    # TODO: load satp from next->satp
    ld t0, 15*reg_size(a4)
    csrw satp, t0

    # return to ra
    ret
```

3. 在 `__init_sepc` 中加入换栈操作
   请在下方代码框中补充完整你的代码：

```
.globl __init_sepc
__init_sepc:
  # DONE: Set sepc to test(in virtual memory)
    li t0, 0x1000000
    csrw sepc, t0
    # TODO: swap sp and sscratch, now sp point to user stack, sscratch point to
kernel stack
    csrr t0, sscratch
    csrw sscratch, sp
    mv sp, t0

    sret
```

3.11 编译及测试

执行 `make run`，对 `main.c` 做修改，确保输出本组成员的学号与姓名。本次实验使用的调度算法为`SJF`。

请在此附上你的代码成功运行的运行结果截图，可以贴多张图
答:

```
ZJU OSLAB 3 3230104947 秦立 / 3230102363 邱俊明
task init...
[PID = 0] Process Create Successfully!
[PID = 1] Process Create Successfully!
[PID = 2] Process Create Successfully!
[PID = 3] Process Create Successfully!
[PID = 4] Process Create Successfully!
task[0]: counter = 1, priority = 4 <-- next
task[1]: counter = 4, priority = 5
task[2]: counter = 3, priority = 2
task[3]: counter = 4, priority = 1
task[4]: counter = 5, priority = 4
[*PID = 0] Context Calculation: counter = 1,priority = 4
task[0]: counter = 0, priority = 4
task[1]: counter = 4, priority = 5
task[2]: counter = 3, priority = 2 <-- next
task[3]: counter = 4, priority = 1
task[4]: counter = 5, priority = 4
[*PID = 2] Context Calculation: counter = 3,priority = 2
[*PID = 2] Context Calculation: counter = 2,priority = 2
[*PID = 2] Context Calculation: counter = 1,priority = 2
task[0]: counter = 0, priority = 4
task[1]: counter = 4, priority = 5
task[2]: counter = 0, priority = 2
task[3]: counter = 4, priority = 1 <-- next
task[4]: counter = 5, priority = 4
[*PID = 3] Context Calculation: counter = 4,priority = 1
```

```
[*PID = 3] Context Calculation: counter = 3,priority = 1
[*PID = 3] Context Calculation: counter = 2,priority = 1
[*PID = 3] Context Calculation: counter = 1,priority = 1
task[0]: counter = 0, priority = 4
task[1]: counter = 4, priority = 5 <-- next
task[2]: counter = 0, priority = 2
task[3]: counter = 0, priority = 1
task[4]: counter = 5, priority = 4
[*PID = 1] Context Calculation: counter = 4,priority = 5
[*PID = 1] Context Calculation: counter = 3,priority = 5
[*PID = 1] Context Calculation: counter = 2,priority = 5
[*PID = 1] Context Calculation: counter = 1,priority = 5
task[0]: counter = 0, priority = 4
task[1]: counter = 0, priority = 5
task[2]: counter = 0, priority = 2
task[3]: counter = 0, priority = 1
task[4]: counter = 5, priority = 4 <-- next
[*PID = 4] Context Calculation: counter = 5,priority = 4
[*PID = 4] Context Calculation: counter = 4,priority = 4
[*PID = 4] Context Calculation: counter = 3,priority = 4
[*PID = 4] Context Calculation: counter = 2,priority = 4
[*PID = 4] Context Calculation: counter = 1,priority = 4
task[0]: counter = 3, priority = 1
task[1]: counter = 2, priority = 1
task[2]: counter = 4, priority = 1
task[3]: counter = 1, priority = 1
task[4]: counter = 1, priority = 2 <-- next
[*PID = 4] Context Calculation: counter = 1,priority = 2
task[0]: counter = 3, priority = 1
task[1]: counter = 2, priority = 1
task[2]: counter = 4, priority = 1
task[3]: counter = 1, priority = 1 <-- next
task[4]: counter = 0, priority = 2
[*PID = 3] Context Calculation: counter = 1,priority = 1
task[0]: counter = 3, priority = 1
task[1]: counter = 2, priority = 1 <-- next
task[2]: counter = 4, priority = 1
task[3]: counter = 0, priority = 1
task[4]: counter = 0, priority = 2
[*PID = 1] Context Calculation: counter = 2,priority = 1
[*PID = 1] Context Calculation: counter = 1,priority = 1
task[0]: counter = 3, priority = 1 <-- next
task[1]: counter = 0, priority = 1
task[2]: counter = 4, priority = 1
task[3]: counter = 0, priority = 1
task[4]: counter = 0, priority = 2
[*PID = 0] Context Calculation: counter = 3,priority = 1
```

```
[*PID = 0] Context Calculation: counter = 3,priority = 1
[*PID = 0] Context Calculation: counter = 2,priority = 1
[*PID = 0] Context Calculation: counter = 1,priority = 1
task[0]: counter = 0, priority = 1
task[1]: counter = 0, priority = 1
task[2]: counter = 4, priority = 1 <-- next
task[3]: counter = 0, priority = 1
task[4]: counter = 0, priority = 2
[*PID = 2] Context Calculation: counter = 4,priority = 1
[*PID = 2] Context Calculation: counter = 3,priority = 1
[*PID = 2] Context Calculation: counter = 2,priority = 1
[*PID = 2] Context Calculation: counter = 1,priority = 1
task[0]: counter = 1, priority = 1
task[1]: counter = 1, priority = 1
task[2]: counter = 1, priority = 1
task[3]: counter = 1, priority = 1
task[4]: counter = 1, priority = 1 <-- next
[*PID = 4] Context Calculation: counter = 1,priority = 1
task[0]: counter = 1, priority = 1
task[1]: counter = 1, priority = 1
task[2]: counter = 1, priority = 1
task[3]: counter = 1, priority = 1 <-- next
task[4]: counter = 0, priority = 1
[*PID = 3] Context Calculation: counter = 1,priority = 1
task[0]: counter = 1, priority = 1
task[1]: counter = 1, priority = 1
task[2]: counter = 1, priority = 1 <-- next
task[3]: counter = 0, priority = 1
task[4]: counter = 0, priority = 1
[*PID = 2] Context Calculation: counter = 1,priority = 1
task[0]: counter = 1, priority = 1
task[1]: counter = 1, priority = 1 <-- next
task[2]: counter = 0, priority = 1
task[3]: counter = 0, priority = 1
task[4]: counter = 0, priority = 1
[*PID = 1] Context Calculation: counter = 1,priority = 1
task[0]: counter = 1, priority = 1 <-- next
task[1]: counter = 0, priority = 1
task[2]: counter = 0, priority = 1
task[3]: counter = 0, priority = 1
task[4]: counter = 0, priority = 1
[*PID = 0] Context Calculation: counter = 1,priority = 1
test end
ticks: 330
```

4 讨论和心得

本次实验作为小组实验来说工作量还算可以，难度适中，主要是部分代码写起来较为繁琐，需要比较细心地考虑到各种情况，比如页表项的建立、页表的切换等，不过好在部分代码可以根据注释很顺利地完成。本次实验有一个小问题是pgtbl2 = (uint64_t*)alloc_page(); 这句代码，会不会有所浪费？后来通过思考发现，一个表理论上有 2^9项，一项的大小是 2^3 bytes, page大小是2^12 Bytes, 刚刚好不会浪费。
实验不足之处依旧是注释等文字部分依然有一些小瑕疵，其他都挺好的。