Lab5: page table

在本实验室中，您将探索页表并对其进行修改，以简化将数据从用户空间复制到内核空间的函数。

> Attention
>
> 开始编码之前，请阅读xv6手册的第3章和相关文件：
>
> - *kernel/memlayout.h*，它捕获了内存的布局。
> - *kernel/vm.c*，其中包含大多数虚拟内存（VM）代码。
> - *kernel/kalloc.c*，它包含分配和释放物理内存的代码。

要启动实验，请切换到pgtbl分支：

```
$ git checkout pgtbl
$ make clean
```

Print a page table (easy)

为了帮助您了解RISC-V页表，也许为了帮助将来的调试，您的第一个任务是编写一个打印页表内容的函数。

YOUR JOB

定义一个名为 `vmprint()` 的函数。它应当接收一个 `pagetable_t` 作为参数，并以下面描述的格式打印该页表。在 `exec.c` 中的 `return argc` 之前插入 `if(p->pid==1) vmprint(p->pagetable)`，以打印第一个进程的页表。如果你通过了 `pte printout` 测试的 `make grade`，你将获得此作业的满分。

现在，当您启动xv6时，它应该像这样打印输出来描述第一个进程刚刚完成 `exec()` ing `init` 时的页表：

```
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
.. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
.. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
.. .. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
```

第一行显示 `vmprint` 的参数。之后的每行对应一个PTE，包含树中指向页表页的PTE。每个PTE行都有一些“`..`”的缩进表明它在树中的深度。每个PTE行显示其在页表页中的PTE索引、PTE比特位以及从PTE提取的物理地址。不要打印无效的PTE。在上面的示例中，顶级页表页具有条目0和255的映射。条目0的下一级只映射了索引0，该索引0的下一级映射了条目0、1和2。

您的代码可能会发出与上面显示的不同的物理地址。条目数和虚拟地址应相同。

一些提示：

- 你可以将 `vmprint()` 放在*kernel/vm.c*中
- 使用定义在*kernel/riscv.h*末尾处的宏
- 函数 `freewalk` 可能会对你有所启发
- 将 `vmprint` 的原型定义在*kernel/defs.h*中，这样你就可以在 `exec.c` 中调用它了
- 在你的 `printf` 调用中使用 `%p` 来打印像上面示例中的完成的64比特的十六进制PTE和地址

A kernel page table per process (hard)

Xv6有一个单独的用于在内核中执行程序时的内核页表。内核页表直接映射（恒等映射）到物理地址，也就是说内核虚拟地址 `x` 映射到物理地址仍然是 `x` 。Xv6还为每个进程的用户地址空间提供了一个单独的页表，只包含该进程用户内存的映射，从虚拟地址0开始。因为内核页表不包含这些映射，所以用户地址在内核中无效。因此，当内核需要使用在系统调用中传递的用户指针（例如，传递给 `w`

`rite()` 的缓冲区指针）时，内核必须首先将指针转换为物理地址。本节和下一节的目标是允许内核直接解引用用户指针。

YOUR JOB

你的第一项工作是修改内核来让每一个进程在内核中执行时使用它自己的内核页表的副本。修改 `struct proc` 来为每一个进程维护一个内核页表，修改调度程序使得切换进程时也切换内核页表。对于这个步骤，每个进程的内核页表都应当与现有的的全局内核页表完全一致。如果你的 `usertests` 程序正确运行了，那么你就通过了这个实验。

阅读本作业开头提到的章节和代码；了解虚拟内存代码的工作原理后，正确修改虚拟内存代码将更容易。页表设置中的错误可能会由于缺少映射而导致陷阱，可能会导致加载和存储影响到意料之外的物理页存页面，并且可能会导致执行来自错误内存页的指令。

提示：

- 在 `struct proc` 中为进程的内核页表增加一个字段
- 为一个新进程生成一个内核页表的合理方案是实现一个修改版的 `kvminit`，这个版本中应当创造一个新的页表而不是修改 `kernel_pagetable`。你将会考虑在 `allocproc` 中调用这个函数
- 确保每一个进程的内核页表都关于该进程的内核栈有一个映射。在未修改的XV6中，所有的内核栈都在 `procinit` 中设置。你将要把这个功能部分或全部的迁移到 `allocproc` 中
- 修改 `scheduler()` 来加载进程的内核页表到核心的 `satp` 寄存器(参阅 `kvminithart` 来获取启发)。不要忘记在调用完 `w_satp()` 后调用 `sfence_vma()`
- 没有进程运行时 `scheduler()` 应当使用 `kernel_pagetable`
- 在 `freeproc` 中释放一个进程的内核页表
- 你需要一种方法来释放页表，而不必释放叶子物理内存页面。
- 调式页表时，也许 `vmprint` 能派上用场
- 修改XV6本来的函数或新增函数都是允许的；你或许至少需要在*kernel/vm.c*和*kernel/proc.c*中这样做（但不要修改*kernel/vmcopyin.c*, *kernel/stats.c*, *user/usertests.c*, 和*user/stats.c*）
- 页表映射丢失很可能导致内核遭遇页面错误。这将导致打印一段包含 `sepc=0x00000000XXXXXXXX` 的错误提示。你可以在*kernel/kernel.asm*通过查询 `XXXXXXXX` 来定位错误。

Simplify `copyin/copyinstr`（hard）

内核的`copyin`函数读取用户指针指向的内存。它通过将用户指针转换为内核可以直接解引用的物理地址来实现这一点。这个转换是通过在软件中遍历进程页表来执行的。在本部分的实验中，您的工作是将用户空间的映射添加到每个进程的内核页表（上一节中创建），以允许`copyin`（和相关的字符串函数`copyinstr`）直接解引用用户指针。

YOUR JOB

将定义在*kernel/vm.c*中的`copyin`的主题内容替换为对`copyin_new`的调用（在*kernel/vmcopyin.c*中定义）；对`copyinstr`和`copyinstr_new`执行相同的操作。为每个进程的内核页表添加用户地址映射，以便`copyin_new`和`copyinstr_new`工作。如果`usertests`正确运行并且所有`make grade`测试都通过，那么你就完成了此项作业。

此方案依赖于用户的虚拟地址范围不与内核用于自身指令和数据的虚拟地址范围重叠。Xv6使用从零开始的虚拟地址作为用户地址空间，幸运的是内核的内存从更高的地址开始。然而，这个方案将用户进程的最大大小限制为小于内核的最低虚拟地址。内核启动后，在XV6中该地址是`0xC000000`，即PLIC寄存器的地址；请参见*kernel/vm.c*中的`kvminit()`、*kernel/memlayout.h*和文中的图3-4。您需要修改xv6，以防止用户进程增长到超过PLIC的地址。

一些提示：

- 先用对`copyin_new`的调用替换`copyin()`，确保正常工作后再去修改`copyinstr`
- 在内核更改进程的用户映射的每一处，都以相同的方式更改进程的内核页表。包括`fork()`，`exec()`，和`sbrk()`.
- 不要忘记在`userinit`的内核页表中包含第一个进程的用户页表
- 用户地址的PTE在进程的内核页表中需要什么权限？(在内核模式下，无法访问设置了`PTE_U`的页面）
- 别忘了上面提到的PLIC限制

Linux使用的技术与您已经实现的技术类似。直到几年前，许多内核在用户和内核空间中都为当前进程使用相同的自身进程页表，并为用户和内核地址进行映射以避免在用户和内核空间之间切换时必须切换页表。然而，这种设置允许边信道攻击，如Meltdown和Spectre。

> 该实验不用回答问题，直接提交实验截图即可