实验三 页表

目标

学习页表的实现机制，用户拷贝数据到内核态的方法

实验要求

作业一

在xv6中，如果用户态调用系统调用，就会切换到内核态，这中间一定是有开销的，至少CPU要保存用户态进程的上下文，然后CPU被内核占有，系统调用完成后再切换回来。作业一本质是加速`getpid()`，思路是为每一个进程多分配一个虚拟地址位于`USYSCALL`的页，然后这个页的开头保存一个`usyscall`结构体，结构体中存放这个进程的`pid`

- 创建每个进程时，在 `USYSCALL` 处映射一个只读页面（在`memlayout.h`中定义的 VA ）。
- 在此页面的开头，存储一个`struct usyscall` （也在`memlayout.h`中定义），并初始化它以存储当前进程的 `PID`。
- 对于本实验，`ugetpid()`已在用户空间端提供，并将自动使用 `USYSCALL` 映射。
- 要求运行`pgtbltest`时`ugetpid`测试用例通过。

作业二

递归遍历页表，碰到有效的就遍历进下一层页表，打印页表内容

- 定义一个名为`vmprint()`的函数。它应该使用`pagetable_t`参数，并以下面描述的格式打印该页表。

```
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
 .. .. ..509: pte 0x0000000021fdd813 pa 0x0000000087f76000
 .. .. ..510: pte 0x0000000021fddc07 pa 0x0000000087f77000
 .. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
```

- 在 `exec.c` 中 `return argc` 之前插入`if(p->pid==1) vmprint(p->pagetable)`，以打印第一个进程的页表，并通过`make Grade`的`PTE`打印输出测试。

实现说明

作业一

some hints:

- 可以在`kernel/proc.c`中的`proc_pagetable()`中执行映射。
- 选择允许用户空间仅读取页面的权限位。
- `mappages()`创建新的映射关系。

- `allocproc()` 中分配和初始化页面。
- `freeproc()` 中释放页面。

coding：

- `USYSCALL` 页是独立于进程页表的一个页，把定义加到 `proc.h` 中
- 在 `proc.c` 中初始化这个页面，`allocproc()` 中会分配一些页，在这个函数里面分配出需要的 `usyscall`
- 在 `proc_pagetable()` 中加入映射逻辑
- 映射完成了之后就要对其进行初始化，回到 `allocproc()`，在最后加入

```
static struct proc * allocproc(void){
    ...
    p->usyscall->pid = p->pid;
    return p;
}
```

- 在进程回收的时候，该页面也要一起释放，在 `freeproc()` 中加入释放即可
- 在 `proc_freepagetable()` 中解除映射，不然启动的时候会报 `freewalk panic`

作业二

some hints：

- 将 `vmprint()` 放在 `kernel/vm.c` 中。
- 使用文件 `kernel/riscv.h` 末尾的宏。
- 学习 `freewalk` 函数释放页表内存。
- 在 `kernel/defs.h` 中定义 `vmprint`，以便可以从 `exec.c` 调用它。
- 在 `printf` 调用中使用 `%p` 打印出完整的 64 位十六进制 `PTE` 和地址。