Lab 实验报告 III

Lab: Page Table

丁睿

dromniscience@gmail.com

更新：2020 年 12 月 7 日

目录

1 任务完成清单

Subtask	Done?	Time
Print a page table	Y	30min
A kernel page table per process	Y	7h
Simplify copyin/copyinstr	Y	3h

Grade: 66/66

2 详细情况 & 困难和收获

我的作业已开放在这个网页上。代码中的注释同时包括了出 bug 的历史代码，我相信很多同学会犯类似的错误。

这一次的练习比之前的更有挑战性，因为需要在原来 xv6 的框架下完整地实现进程私有的内核页表。我们将讲述设计的整体思想和设计层面的具体细节。

2.1 Print a page table

把`vmprint`写成一个递归调用是很方便的，而且非常灵活。注意到 xv6 页表的一个有趣属性可以当作递归终点 (也就是 Level 0 的页表) 的判断条件——指向页表页的页表条目只设置`PTE_V`位。可以仿照`freewalk`函数书写以下代码:

```
if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
    // this PTE points to a lower-level page table.
}
```

当然，由于此处还需要根据深度打印不同长度的前缀，而`vmprint`的参数列表又被限定了。简单起见，我们直接写一个三层循环嵌套打印。我个人实在不能理解`vmprint`当作调试工具的方便之处何在。在 debug 的整个过程钟我从未使用到它。

2.2 A kernel page table per process

这一阶段我们不必完成在内核页表中动态地维护用户空间的映射使之与用户页表同步。我们提供一个进程级页表和内核页表的逻辑映像 (图1)。由于在进程级内核线程中必须使用进程级内核页表，必要的映射区段至少包括 Trampoline 页 (存放用户级 trap 的统一入口的代码)、物理内存区域的直接映射 (包括存放内核数据和代码的页面) 以及硬件接口 (例如处理时钟中断时就必须使用该地址)。与内核页表不同的是，进程级内核页表仅需要记录自己的内核栈的条目，而内核页表有必要记录所有已分配的进程的内核栈。

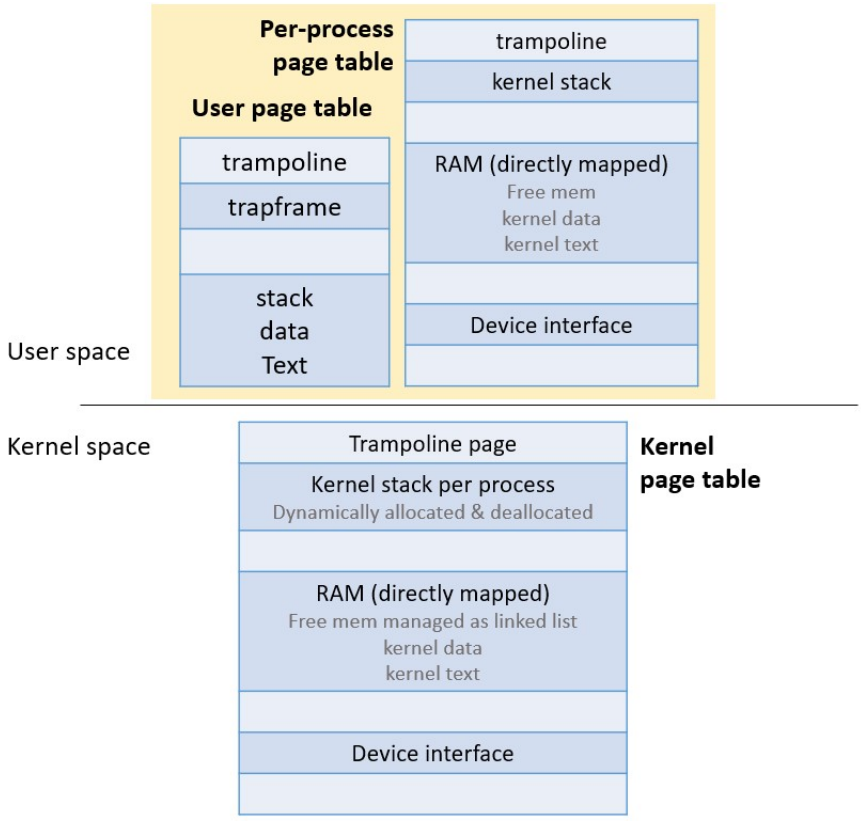

图 1: 进程级内核页表的总体逻辑映像

注意到除了内核栈，内核页表的其他内容的地址都是固定的，所以它们都可以在初始化进程 PCB 时设置 (也就是`allocproc`函数中)。具体工作如下:

- 模仿`kvminit`函数可以设置好除内核栈之外其他的部分。
- 注释掉`procinit`中初始化锁之外的内容。
- 在`allocproc`中分配一个内核栈的页面时同时将它注册到内核页中。

在进程的生命周期中，在目前的要求下，进程级用户页表的内容是完全固定的，不需改变。在释放时，我们需要修改`freeproc`完成以下工作:

- 释放用户页表以及其叶子结点的内存页。这可以原封不动地使用`proc_freepagetable`完成。这一步与下面两项的先后次序没有要求。
- 在释放进程级内核页表页之前必须释放内核栈并清空相应表项。
- 释放该进程集内核页表占用的内核页。这些页面都来自于图1中`free mem managed as linked list`包含的区域，因此直接使用`kfree`释放即可。代码可以仿照之前的`vmprint`编写。唯一需要注意的是，不应该释放叶子节点[1]的物理页面，因为它们已经在第一步中释放了。
- 安全起见，可以将 PCB 中记录进程级内核页表的字段置零。

最后提及页表切换的问题。注意 RISC-V xv6 的实际的上下文切换总是发生在内核级线程中，而且由`swtch`来完成。所以当一个进程结束运行时，`scheduler`将从`swtch`返回，然后继续轮

[1]这里叶子节点的含义是 Level 0 的页表条目指向的物理页，该术语与文献 [1.] 中的含义一致。

询进程表调度下一个进程。因此，只需紧接着`swtch`用内核级页表覆盖`satp`中进程级内核页表的值 (注意中断时 trampoline 已将用户页表切回进程级内核页表)。在调用`swtch`从`scheduler`返回前再换入新进程的进程级用户页表即可。

最开始对该 lab 进行 usertests 评测时一直超时，最后才发现是因为我的电脑未插电，默认开始节能模式。

2.3 Simplify copyin/copyinstr

现在我们要在进程级内核页表中完全复制用户级进程代码段、数据段和堆段的副本 (即完全相同的虚拟地址到物理地址的映射)。以下属性大大便利了我们的操作:

1. 用户虚存空间总是从地址 0 开始，而进程级内核页表的低地址都是空闲的[2](直到 PLIC 所在的 0x0c000000L)。
2. 用户虚存空间是总连续的。也就是我们可以通过简单地递增给进程级内核页表注册新的用户页表项以及释放他们，只需要了解操作前后`p->sz`分别是多少就足够了。

除了更新用户页的映射，余下内核页的设计是完全一致的 (如图2所示)。因此只需在所有可能更新用户页的内核代码中同步更新进程级用户页表:

`fork()` 进程级页表的其它内容无需改变，只需复制新进程的`[0, np->sz)`的用户页表即可。

`exec()` 由于已经包含了旧进程的用户页表项,必须先注销`[0, p->sz)`再根据新的用户页表复制`[0, np->sz)`。进程级页表的其它内容无需改变。

`growproc()` 如果是扩大内存，只需注册`[p->sz, np->sz + n)`的页表；如果缩小内存，只需注册`[p->sz + n, np->sz)`的页表。注意，这里需要判断用户虚存空间是否越过 PLIC 允许的上界。

`userinit()` 注意第一个用户进程不是通过`fork()`创建的，因此必须单独地拷贝用户页表项。

另一个容易忽略的问题是，指示进程虚存空间的`sz`参数不一定是按页对齐的。因此必须和源码遵循同样的舍入方式以保证不重不漏地更新页表项。分配`sz`时当然要向上舍入，这限制了更新时舍入的方式是唯一的。细节不做展开，我在源码中写了相应的注释。

最后，注意取消相应页的`PTE_U`标志，因为内核模式下 xv6 被设计成不能访问带有用户标记的代码。如果为了追求安全性，在分配新的用户页表项时选择检查`PTE_V`为零 (即确认该条目当前未注册) 再注册，那么必须取消进程级用户页表中 CLINT 的映射。因为运行某些用户进程时就将重新注册该表项的内容，这会导致一个 panic:remap 的错误。

[2]你会注意到实际上早在 0x2000000L 的位置就影射了 CLINT。但矛盾在于很多 xv6 程序需要的内存空间大于这个数，例如 usertests.c。因此要实现我们的目标就不得不对其进行覆盖。所幸覆盖它们不会造成灾难——必要时可以从内核页表中找到该信息。

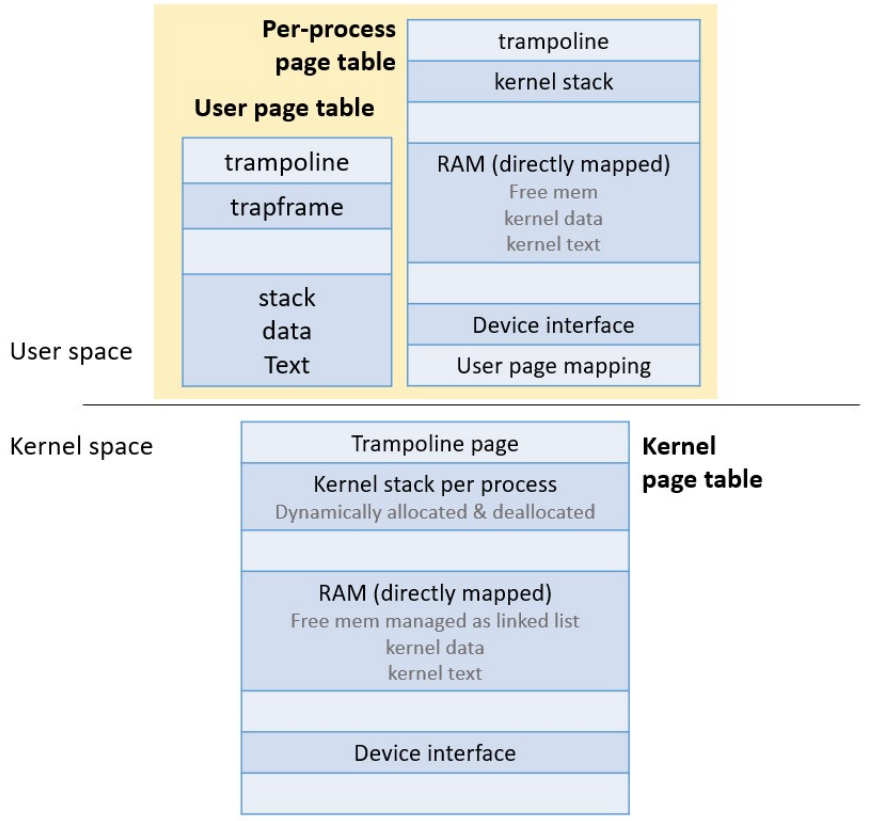

图 2: 映射了用户页的进程级内核页表

3 对课程和 lab 的建议

此 lab 非常值得一做。它要求完整地理解页表管理的各种细节，同时也展示了编写系统级代码时设计框架的重要性——既要追求安全，又要有效地使用代码复用完成功能模块。

熟悉并主动使用 xv6 源码中提供的功能接口，将极大地减少代码量并降低调试难度。

4 参考资料

1. *xv6: a simple, Unix-like teaching operating system*
   Russ Cox, Frans Kaashoek, Robert Morris *August 31, 2020*
2. *The RISC-V Reader: An Open Architecture Atlas*
   David Patterson, Andrew Waterman *1st Edition*
3. 现代操作系统 [M]
   A.S.Tanenbaum, H. Bos 著, 陈向群 马洪兵 译, 北京: 机械工业出版社, 2011: 47-95