同济大学操作系统课程设计 - Xv6 - Lab: page tables

【实验目的】

在操作系统中，系统调用（例如 `getpid()`）通常需要在用户空间（用户程序）和内核空间（操作系统核心）之间切换。这种切换是开销较大的操作，因为需要保存和恢复上下文，可能还涉及到数据的复制。

1. **共享只读区域**：
   - 为了减少切换开销，操作系统可以在用户空间和内核空间之间共享一个只读的内存区域。这意味着用户空间的程序可以直接访问内核空间的数据，而不需要进行数据复制。
   - 这种共享是通过页表来实现的。页表是操作系统用来管理内存映射的结构。
2. **映射页表**：
   - 操作系统在创建进程时，可以为每个进程分配一个只读的内存页，并将这个页通过页表映射到用户空间。这个只读页包含了如 `struct usyscall` 结构体等数据。
   - 这样，用户空间的程序可以直接访问这个只读页中的数据（比如 `PID`），而无需进入内核空间。
3. **优化 `getpid()` 系统调用**：
   - 在 `getpid()` 系统调用中，程序可以直接从这个映射的只读页中读取 `PID`，避免了频繁的用户空间和内核空间切换。
   - 这减少了上下文切换的开销，提高了系统调用的性能。

实验的目标是通过在 xv6 操作系统中实现这种优化，学习如何在页表中插入映射，从而加速 `getpid()` 等系统调用。总的来说，这种优化通过直接在用户空间访问内核中的数据（只读），避免了数据的复制和频繁的上下文切换，从而提高了性能。

【实验环境】

- **虚拟机：**VMare Workstation 17
- **操作系统：**Ubuntu-20.04.6
- **实验系统：**xv6-labs-2021

【实验内容】

要启动实验室，需要先切换到 pgtbl 分支：

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

- **定义结构体**：创建一个 `struct usyscall` 结构体，里面包含当前进程的 `PID`。
- **分配只读页**：在进程创建时，为每个进程分配一个只读的内存页，并在这个页中放置 `struct usyscall` 结构体的实例。
- **映射到用户空间**：通过页表将这个只读页映射到用户空间，这样用户空间的程序就可以直接访问这个数据，而无需切换到内核空间。

- **修改系统调用**：在 `getpid()` 系统调用中，直接从映射的页中读取 `PID`，而不需要每次都进行用户空间和内核空间的切换。

1 Speed up system calls

在传统的系统调用过程中，用户态程序需要陷入内核态以获取内核中的数据。这种用户态与内核态之间的频繁切换会引入显著的性能开销。尤其是对于简单的系统调用（如 `getpid()`），这种开销是非常不必要的。通过在用户空间和内核之间共享一个只读区域，数据可以直接从内核映射到用户空间，避免了数据的拷贝，从而减少了上下文切换带来的开销。

页表是操作系统管理虚拟内存的核心机制之一。通过页表，操作系统可以控制用户空间程序对物理内存的访问权限。我们可以利用页表，将内核中的某个只读页映射到用户空间中，从而实现数据的共享。在此实验中，我们将重点关注如何在 xv6 操作系统中实现这种页表映射。

以下是关于在 xv6 操作系统中通过页表机制优化 `getpid()` 系统调用的实现步骤的整理与总结：

1. 在 `kernel/proc.h` 的 `proc` 结构体中添加指针

   在 `proc` 结构体中添加一个指针用于保存共享页面的地址，以便在进程的生命周期内保持对该页的引用。

   ```
   struct usyscall *usyscallpage;
   ```

2. 在 `kernel/memlayout.h` 中完成页面映射工作
   - 在 `memlayout.h` 中定义用于共享页的虚拟地址位置。这个地址将会被映射到用户空间，并且进程可以通过该地址访问 `usyscall` 结构。

3. 在 `kernel/proc.c` 的 `allocproc()` 函数中分配和初始化共享页
   - 在新进程创建时，使用 `allocproc()` 函数为进程分配一个共享页，并初始化 `usyscall` 结构中的数据，比如进程的 `PID`。该页的地址将保存在 `proc` 结构体中的 `usyscallpage` 指针中。

     ```
     p->usyscallpage->pid = p->pid;
     ```

4. 在 `kernel/proc.c` 的 `proc_pagetable(struct proc *p)` 中完成页面分配
   - 为新创建的进程设置页表，将 `usyscall` 结构体所在的页映射到用户空间的指定地址，并设定权限为只读（`PTE_R | PTE_U`），确保用户态只能读取而无法修改该数据。

     ```
     if (mappages(p->pagetable, TRAMPOLINE - PGSIZE, PGSIZE, (uint64)p-
     >usyscallpage, PTE_R | PTE_U) != 0) {
         ...
     }
     ```

5. 在 `kernel/proc.c` 的 `freeproc()` 函数中释放页面
   - 在进程结束时，使用 `freeproc()` 函数释放之前分配的共享页，以避免内存泄漏。确保在释放该页之前，解除与页表的映射。

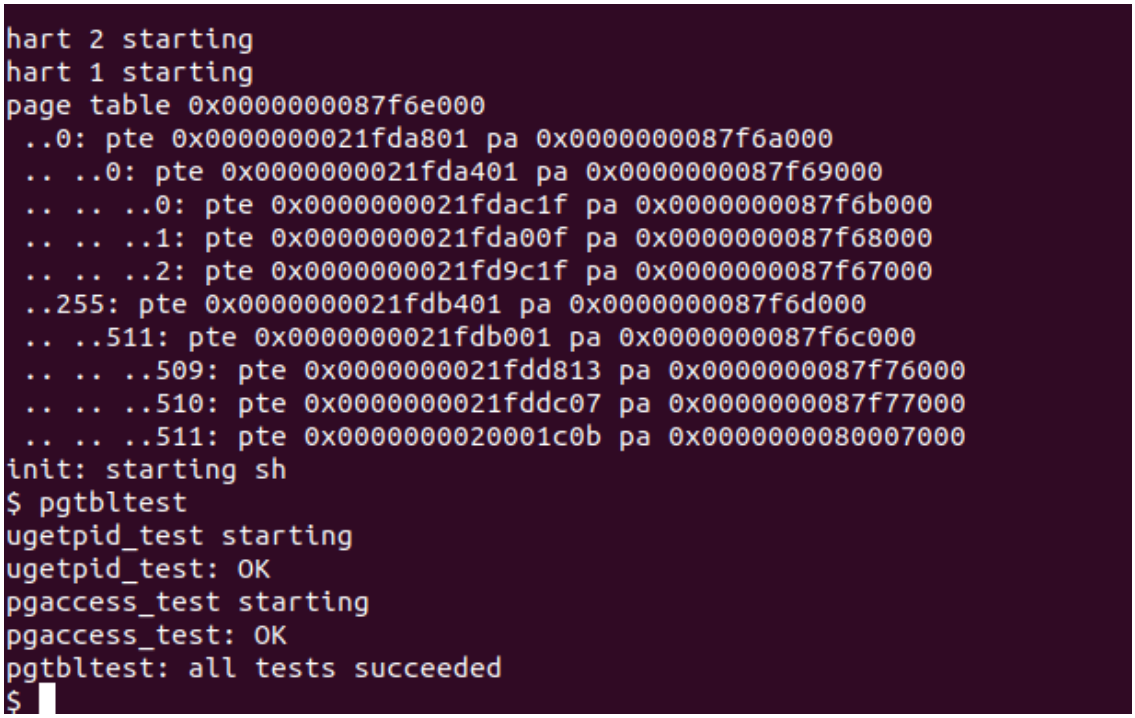
     ```
     kfree((void *)p->usyscallpage);
     p->usyscallpage = 0;
     ```

6. 在 `kernel/proc.c` 的 `proc_freepagetable(pagetable_t pagetable, uint64 sz)` 中释放页表项
    - 进一步清理进程退出时的资源，确保页表中对应的页表项被正确释放，以避免潜在的内存问题。

    ```
    uvmFRE(pagetable, sz);
    ```

7. 使用 `make qemu` 指令运行 xv6
8. 在命令行中输入 `pgtbltest`

    在命令行中输入 `pgtbltest`，以检查页面表映射的正确性，并测试 `getpid()` 系统调用的优化效果：



2 Print a page table

1. **在 `kernel/exec.c` 中插入打印页表的代码**
    - 找到 `exec()` 函数中 `return argc;` 之前的位置。
    - 插入条件语句以在执行 `init` 进程时打印页表。

    ```
    if (p->pid == 1) {
        vmprint(p->pagetable);  // 打印 init 进程的页表
    }
    ```

2. **在 `kernel/defs.h` 中定义 `vmprint` 原型**

    在 `defs.h` 中添加 `vmprint` 函数的原型，以便其他文件调用。

    ```
    void vmprint(pagetable_t pagetable);
    ```

3. **编写 `vmprintwalk()` 函数**
    - 仿照 `kernel/vm.c` 的 `freewalk()` 函数，编写 `vmprintwalk()` 函数。

- `vmprintwalk()` 函数递归地逐层打印页表的内容，包括每个 `PTE` 的索引、16 进制表示和从 `PTE` 中提取的物理地址

```
void vmprint(pagetable_t pagetable) {
    ...
    // 递归遍历页表并打印内容
    ...
}
```

4. **在终端中编译并运行**
   - 保存修改后，使用 `make qemu` 指令编译并运行 xv6。
   - 确保按照格式打印出每个 `PTE` 的索引、16 进制表示和提取的物理地址：

```
hart 1 starting
hart 2 starting
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
init: starting sh
```

3 Detecting which pages have been accessed

1. 在 `kernel/riscv.h` 中定义 `PTE_A`
   - `PTE_A` 是 RISC-V 架构中定义的访问位，用于标记某个页面是否已被访问。我们首先在 `riscv.h` 中定义它。

```
#define PTE_A (1L << 6)
```

这条定义语句将 PTE_A 设置为一个移位的常量，表示访问位在页表项中的位置。

2. 在 `defs.h` 中声明 `walk` 函数
   - `walk` 函数用于在页表中查找特定的页表项。我们需要在 `defs.h` 文件中声明这个函数，以便在其他文件中使用它。

```
pte_t *walk(pagetable_t pagetable, uint64 va, int alloc);
```

通过声明 `walk` 函数，可以在 `sys_pgaccess()` 系统调用中使用它来查找并操作页表项。

3. 在 `kernel/sysproc.c` 中实现 `sys_pgaccess()`

   `sys_pgaccess()` 是我们要实现的系统调用，用于检测页面的访问状态。以下是实现的步骤：

   1. **获取系统调用参数：**
      - 我们需要获取三个参数：起始虚拟地址、要检查的页面数、以及用于存储结果的缓冲区地址。
   2. **遍历页面并检查访问位：**

- 使用 `walk` 函数遍历页表项，检查每个页面的 PTE_A 位是否被设置。
- 如果某个页面的 PTE_A 位被设置，意味着该页面已被访问。

3. **重置访问位：**
   - 为了后续的检测，我们可能需要重置这些访问位。
4. **将结果存储到用户空间：**
   - 使用位运算将每个页面的访问状态置入一个临时位向量中，最后将这个位向量复制到用户内存中的指定地址。

```
uint64
   sys_pgaccess(void)
   {
       uint64 start_va;
       int num_pages;
       uint64 user_buf;
       uint64 access_bits = 0;
       pte_t *pte;

       // 获取系统调用参数
       if (argaddr(0, &start_va) < 0 || argint(1, &num_pages) < 0 ||
argaddr(2, &user_buf) < 0)
           return -1;

       for (int i = 0; i < num_pages; i++) {
           // 获取页表项
           pte = walk(myproc()->pagetable, start_va + i * PGSIZE, 0);
           if (pte == 0)
               return -1;

           // 检查 PTE_A 位
           if (*pte & PTE_A) {
               access_bits |= (1L << i);
               // 重置访问位
               *pte &= ~PTE_A;
           }
       }

       // 将结果存储到用户缓冲区
       if (copyout(myproc()->pagetable, user_buf, (char *)&access_bits,
sizeof(access_bits)) < 0)
           return -1;

       return 0;
   }
```

4. 编译并运行 xv6
   - 保存所有修改后，在终端中执行 `make qemu`，编译并运行 xv6 操作系统。
5. 在命令行中输入 `pgtbltest`

   在 xv6 的命令行中输入 `pgtbltest`，以测试 `pgaccess()` 系统调用的功能是否正常工作。这个测试将会验证你实现的页面访问跟踪功能。

```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
$
```

【分析讨论】

- 通过对内存管理相关函数的替换与修改，深刻理解了用户页表和内核页表的作用以及它们之间的关系。特别是在页表复制和权限管理方面，获得了宝贵的实际经验。
- 实际编写和调试了 `vmcopypage` 函数，并在多个关键函数中集成了新的内存处理逻辑。这增强了我在系统级编程中的实践能力。
- 遇到和解决了一些在系统调用及内存管理中实际出现的问题，这不仅提高了我的调试技巧，也让我学会了如何有效地追踪和解决系统级编程中的复杂问题。
- 通过优化内存管理函数，提高了系统性能的实际效果，这让我认识到优化操作系统性能的重要性，并理解了在内存管理方面进行精细调控的必要性。

【实验验证】

新建`time.txt`，输入自己做实验的用时，运行`make grade`进行评分：

```
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (0.7s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests ==
$ make qemu-gdb
Timeout! (300.3s)
== Test   usertests: all tests ==
  usertests: all tests: FAIL
    ...
         test bigfile: OK
         test dirfile: OK
         test iref: OK
         test forktest: OK
         test bigdir: qemu-system-riscv64: terminating on signal 15 from pid 12
6402 (make)
    MISSING '^ALL TESTS PASSED$'
== Test time ==
time: OK
Score: 36/46
make: *** [Makefile:336: grade] Error 1
yzz@ubuntu:~/Desktop/xv6/xv6-2021/lab3$
```