Lab4 实验报告

游子诺

2019 年 5 月 15 日

1 实验思考题

Thinking 1

思考并回答下面的问题：

- 内核在保存现场的时候是如何避免破坏通用寄存器的？
- 系统陷入内核调用后可以直接从当时的 $a0-a3$ 参数寄存器中得到用户调用 msyscall 留下的信息吗？
- 我们是怎么做到让 sys 开头的函数“认为”我们提供了和用户调用 msyscall 时同样的参数的？
- 内核处理系统调用的过程对 Trapframe 做了哪些更改？这种修改对应的用户态的变化是？

1. 如何避免破坏通用寄存器？

   - 保存前：使用汇编语言执行保存现场的工作，防止编译器使用通用寄存器进行编译，只使用约定的 k0 和 k1 寄存器。
   - 保存时：，并用 k0 或 k1 寄存器将 sp 预先保存下来，而后把 sp 指向对应的异常/中断处理栈，并在那里保存其他通用寄存器的值。
   - 恢复时：即使用 sp 把所有的通用寄存器值恢复回来，再借助 k0 和 k1 把 sp 也恢复回来。
   - 恢复后：恢复完毕后，除 k0 和 k1 的通用寄存器（包括 sp）都回到中断异常前的值，但 k0 和 k1 并没有恢复（因而可以理解成一定有人牺牲才能保全其他人）。

2. 直接从 a0-a3 中获取信息？在本 lab 中，通过分析从使用 syscall 到内核调用的途中，可以发现 a0-a3 寄存器都没有被使用过，因此直接用实际是可行的。但是，随着系

统复杂度的提升，不排除 a0-a3 在临界过程中使用，因此理想方式是在 TrapFrame 中调取对应的 a0-a3 值。

3. 如何确保相同的参数？在 lib/syscall.S 中，使用了汇编函数确保即将跳转到的 sys 函数的参数和 mysyscall 参数相同，具体来说，就是确保 a0-a3 不变。

4. 对 TrapFrame 的修改？修改了 EPC 的值，使得从内核态返回时执行 syscall 下面一条语句；修改了 TramFrame 的 v0 寄存器的值，从而实现了系统调用时内核函数向用户函数返回值的传递。

Thinking 2

思考下面的问题，并对这两个问题谈谈你的理解：

- 子进程完全按照 fork() 之后父进程的代码执行，说明了什么？
- 但是子进程却没有执行 fork() 之前父进程的代码，又说明了什么？

- 子进程自恢复以来，拥有和父进程完全相同的寄存器环境、内存数据和 PC 值。
- 子进程的创建和普通进程通过加载 elf 文件是不同的，虽然是用 env_run() 使其跑起来，但是通过修改进程块中的 pc 值，其运行的起始点并不在文件入口，而是 fork() 这条语句，更准确地说，应该是 system_env_alloc() 中的 mysyscall 中的 syscall 语句后一句。

Thinking 3

关于 fork 函数的两个返回值，下面说法正确的是：

1. fork 在父进程中被调用两次，产生两个返回值。
2. fork 在两个进程中分别被调用一次，产生两个不同的返回值。
3. fork 只在父进程中被调用了一次，在两个进程中各产生一个返回值。
4. fork 只在子进程中被调用了一次，在两个进程中各产生一个返回值。

答案：第三项 C

Thinking 4

如果仔细阅读上述这一段话, 你应该可以发现, 我们并不是对所有的用户空间页都使用 duppage 进行了保护。那么究竟哪些用户空间页可以保护，哪些不可以呢，请结合 include/mmu.h 里的内存布局图谈谈你的看法。

首先，用户进程的复制只能涉及到用户空间的部分，因而所有可以 duppage 的页一定存在于 0 UTOP 这个范围，根据 mmu.h 所示的空间分布，逐个进行讨论：

- user exception stack: 不可 duppage，此部分恰用于用户进程处理 COW 时的临时栈空间，处理 COW 问题的内存空间肯定不能自己就是 COW，否则将会中断重入等问题。
- invalid stack: 选择性 duppage，此部分原则上不应该使用，但在本 lab 实现过程中，我将其用到了 pgfault 函数挪移页内存时的临时空间，因而其参与到了 COW 的过程，不可保护。
- normal user stack; 由用途，属于进程函数调用、临时变量等操作的位置，显然应该保护。
- text, data, bss: 应该 duppage 保护。
- 0 2*PDMAP: 暂时不清楚其用途。

Thinking 5

在遍历地址空间存取页表项时你需要使用到 vpd 和 vpt 这两个“指针的指针”，请思考并回答这几个问题：

- vpt 和 vpd 的作用是什么？怎样使用它们？
- 从实现的角度谈一下为什么能够通过这种方式来存取进程自身页表？
- 它们是如何体现自映射设计的？
- 进程能够通过这种存取的方式来修改自己的页表项吗？

- 通过 grep 查找 vpt 和 vpd，我们发现其被定义在 UVPT 内存空间段。vpt 和 vpd 其实充当指针的作用，当使用 * 运算即可取出其地址中的值。例如要访问页目录的第 i 项，可以写成 (* vpd)[x] / (* ((Pte*)vpd + i))
- UENVS 是内核程序在 env 块初始化时，给予 env 的能够读但不能写的虚拟内存管理，其对应的实际是内核中 envs 数组的内容，但是无法写操作。
- 自映射设计一定是创造者和使用者共同遵守才能够形成的。

  在构造 env 时：其所对应的页目录里 pgdir[PDX(UVPT)] = cr3 | perm，这样使得加载此页目录为 mCONTEXT 后 tlb 访问 UVPT 部分空间时使用的一级页表和二级页表其实是同一个物理页。

  在定义 vpd 时：页目录的起始地址为 (UVPT +(UVPT » 12)*4)，是从 UVPT 中取出页目录的，因此是自映射结构。
- 不能！

Thinking 6

page_fault_handler 函数中，你可能注意到了一个向异常处理栈复制 Trapframe 运行现场的过程，请思考并回答这几个问题：

- 这里实现了一个支持类似于“中断重入”的机制，而在什么时候会出现这种“中断重入”?
- 内核为什么需要将异常的现场 Trapframe 复制到用户空间？

- 首先经过分析，在内核态的 page_fault_handler 运行中，应该是不会再次出现 COW 的写异常导致中断嵌套。此处“中断重入”所处的情景，应该是把内核态的 COW 缺页中断处理函数和用户态的 pgfault 处理函数合并考虑，当内核态已经跳转至用户态的 pgfault 进行缺页中断时，这时如果再次写 COW 标志的页，则又触发了缺页中断，从而实现了“重入”。
- 因为遵循微内核的思想，pgfault 的实际拷贝处理是交给用户态进程来处理的，用户进程处理完 pgfault 完后就要恢复到陷入异常的状态，而此时用户态显然不能依赖于内核栈来恢复环境，因而只能“恳请内核”预先把恢复现场所需要的东西准备在自己能够访问的空间中。

Thinking 7

到这里我们大概知道了这是一个由用户程序处理并由用户程序自身来恢复运行现场的过程，请思考并回答以下几个问题：

- 用户处理相比于在内核处理写时复制的缺页中断有什么优势？
- 从通用寄存器的用途角度讨论用户空间下进行现场的恢复是如何做到不破坏通用寄存器的？

- 使内核更轻量，内核能够有时间处理更多其他的任务。缺页问题其实仅与一个进程是否顺利运行有关，因此缺页的处理是能够被打断的，而要实现能够被调度打断只能在用户态中。
- 参考 user/entry.S 中的恢复现场函数。首先，内核函数把异常前的现场都赋值到 UXSTACKTOP 的栈空间中；而后用户态进程进行 pgfault 处理；当用户态处理完毕后，回到 entry.S 中根据“和内核签下的现场内容保存位置协议”，从 UXSTACKTOP 中进行现场的恢复。

  换一种思路，用户态的 pgfault 处理过程其实是介于用户态和内核态之间的一个”伪内核态“，其用用户态的权限运行，但是又有专用的不会出现 pgfault 的栈空间使得操作顺利进行（效果如同在内核中操作）。

Thinking 8

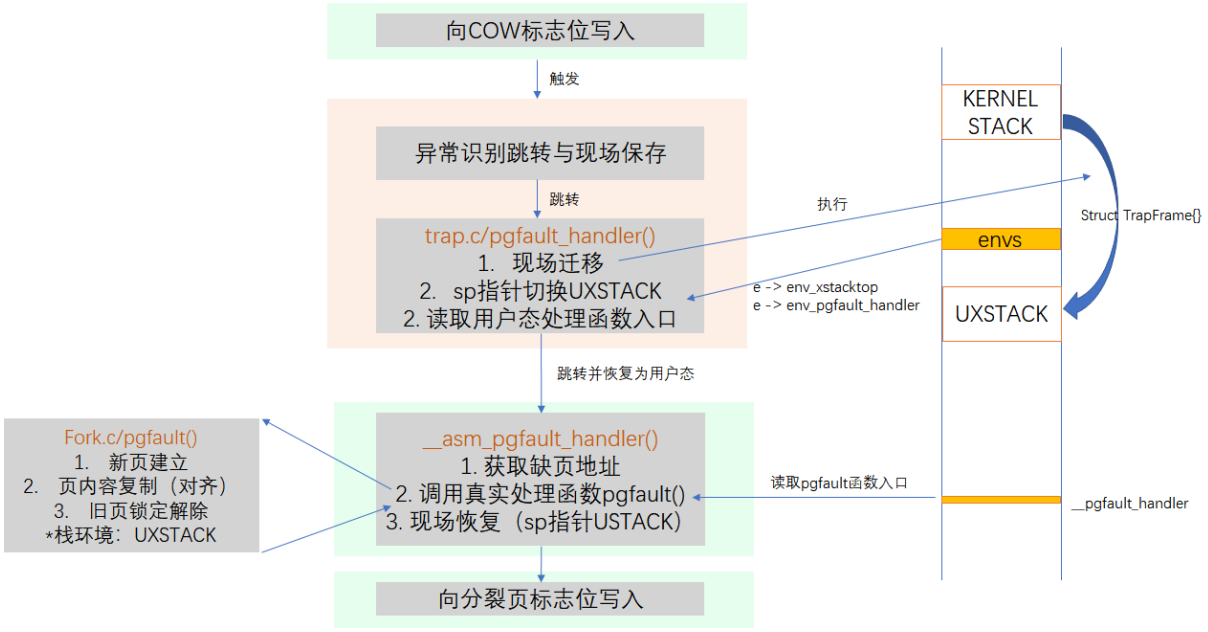
请思考并回答以下几个问题：

- 为什么需要将 set_pgfault_handler 的调用放置在 syscall_env_alloc 之前？
- 如果放置在写时复制保护机制完成之后会有怎样的效果？
- 子进程需不需要对在 entry.S 定义的字 ___pgfault_handler 赋值？

- 在 fork 函数后面我们看到父进程已经帮助子进程设置了子进程结构体的相关内容和分配了 uxstacktop 的空间，如果把这个函数放在 syscall_env_alloc 之后，那么子进程被 run 以后就会执行这个函数，运行不必要的过程而且可能会发生错误。
- 在 duppage 之后父进程的就已经被 COW 保护了，而调用 set_pgfault_handler 是调用函数，会涉及到使用栈，而此时的 pgfault 并没有配置好，因此会有问题。
- 不需要，子进程在 alloc 时就拥有父进程这部分空间，因此无需赋值。

2 实验难点图示

在 lab4 中，我遇到的实验难点主要是 COW 缺页中断的初始化与处理流程，其中涉及到的进程状态、函数和内存数据比较繁复，在此进行图示梳理：



其中以下几点需要注意一下几点：

- TrapFrame 复制的原因与工作原理，这点在 lab3 已有明确的探讨说明，弄清 TrapFrame 及其复制迁移过程对如何从用户态的 pgfault 恢复到陷入异常的现场非常重要。
- 各个函数运行时栈空间指针的变化，在整个流程中共有内核栈、用户异常栈和用户栈三个栈，内核栈与用户异常栈的切换在**由内核态到用户态**，用户异常栈到用户栈的切换在 **asm 汇编函数中的现场恢复**。
- 从内存中取得一些数据：第一是从 envs 结构体中获取 xstacktop 和 asm_handler 的程序入口；第二是从用户空间中的 ___pgfault_handler 获取的 pgfault 程序的入口。

3 体会与感想

在本次 lab 中，我有以下体会：

3.1 系统调用与接口

系统调用从面向对象的角度来看，有点类似于类的 public 方法。首先，内核通过定义固定的公开方法，从而使用户程序只能运行有限的合乎常理和安全性的操作，令自身内部的大部分数据结构得以保护，不被用户程序非法的操作而破坏。其次，接口形式隔离了外部操作和内部操作，如果需要调整内核态的一些数据结构和类型，仅需要调整内核态内部的运行过程，而用户进程是不需要发生变化的。

3.2 用户态的 COW 缺页中断

我认为，在 lab4 中理解清晰 COW 缺页中断的处理脉络是最为重点的之一。下面我将从缺页中断的处理脉络和用户态的缺页中断处理两个方向来谈谈自己的想法：

缺页中断的处理重点分为**缺页中断处理程序的设置**和**缺页中断的处理跳转**:

- **中断处理程序的设置**：明确 pgfault.c 函数中的 set_pgfault_handler 功能与 syscall_set_pgfault_handler 二者之间的差异。在实际运行时，父进程运行前者而子进程运行后者，从函数分析上，前者其实包括了后者，而差别在于：第一，前者设置了 pgfault 用户处理函数的实际位置；第二，前者的使用对象必须是当前运行的进程而后者指定 env_id 即可（这也是对子进程的 handler 设置几乎要把前者重复一遍，因为前者不支持对其他进程设置）。

- **中断处理程序的处理跳转**：整个 COW 中断处理跳转其实是内核 + 用户的运行过程：在内核阶段要留意的核心函数是 trap.c 中的 pgfault_handler，这里将看到用户异常栈的设置和将内核返回的 EPC 设置为”神奇的地方-___asm_pgfault_handler“；在用户态阶段，重点则关注 ___asm_pgfault_handler 这个函数，这个函数是用户态处理 COW 的 pgfault 的枢纽函数，首先，其功能在于调取预先被设置好的 fock.c/pgfault 函数真正处理问题；其次，既然之前内核态设置的 EPC 是一个”神奇的值“，那么跳转肯定回不到发成 COW 中断的语句，因此 asm 函数也肩负恢复真正现场的责任。

用户态的缺页中断处理　其实通过观察 asm_pgfault_handler 这个汇编函数，我们发现这个运行在**用户态**的函数使用到了 k0,k1 这些一般由内核使用的寄存器，因此处理 pgfault 的用户态程序其实可以被看做**偏向于内核态的，介于内核和用户的中间态程序**，之所以这样做，我目前认为主要是为了减轻内核的负担，因为一个用户进程发生的 COW 缺页中断仅对其自己产生影响，这个中断处理时是可以被打断和延后的，就像多

线程编程中对一个类的上百个方法都只使用 synchronized 一把锁，但其实这个类某些数据是可以并行操作的，这时候要用多把锁隔离。

4 指导书残留难点与反馈

4.1 pgfault 部分的填充函数

在指导有关 pgfault 缺页中断的函数的完成顺序是比较混乱的，并且各个函数的名字很相似让我最初编程的时候很难琢磨清楚各个函数在运行中的功能。初步看来有两条脉络：内核与用户，初始化与执行，个人比较倾向于采用**初始化与执行**的脉络指导编程：

- 首先阐述 COW 型 pgfault 的处理方式是用户态运行 -> 内核态处理 -> 用户态处理 -> 用户态运行的过程，而后具体梳理几个过程：
- 在内核态部分介绍异常运行到 trap.c/pgfault_handler 的过程，并**围绕 pgfault_handler** 主讲重要的几个知识点：异常用户处理栈、回到用户态时 epc 的设置。
- 在用户态处理程序部分，**将 asm 汇编程序作为处理程序的中心枢纽重点解释**，解释如何跳转到实质的处理函数和如何在用户态恢复为用户态的过程，最后引出 pgfault 的具体处理过程。
- 最后，才是从运行跳回到初始化的过程，与 pgfault 初始化的程序有两个，这两个函数的功能总的有：分配异常处理栈、结构体中的入口定义和 pgfault 实际处理函数的定义，围绕着这三类对象再次温习一下他们在异常处理过程中出现的时刻与作用，从而明确其意义。