Lab4 实验报告

19373135 田旗舰

一、实验思考题

Thinking 4.1 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？

利用 SAVE_ALL 将通用寄存器的值保存到内核栈中。

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？

可以，内核保存现场的过程中没有破坏 a0-a3 寄存器的值，只改变过 k0, k1, v0 的值。

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？

参数的传递依赖于 a0-a3 参数寄存器和栈，我们将前四个参数放在对应的寄存器上，后两个参数存在栈上的相同位置。

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？

修改了 v0 的值，传递函数返回值，修改了 epc，使返回用户态时可以执行下一条指令。

Thinking 4.2 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？

子进程的代码段与父进程相同。

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？

说明子进程开始执行的位置是 fork()，即恢复上下文的位置为 fork()。

Thinking 4.3 关于 fork 函数的两个返回值，下面说法正确的是：C

A. fork 在父进程中被调用两次，产生两个返回值

B. fork 在两个进程中分别被调用一次，产生两个不同的返回值

C. 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

D. 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因为

Thinking 4.4 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。

对从 0 到 USTACKTOP 的地址空间且不是共享的和只读的页面进行保护。

Thinking 4.5 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？

vpt 和 vpd 分别为页表项和页目录所在的虚拟地址。

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？

vpt 和 vpd 通过宏定义对应了内存中页表所在的虚拟地址。

它们是如何体现自映射设计的？

（UVPT+(UVPT>>12)*4)

进程能够通过这种存取的方式来修改自己的页表项吗？

不能，没有权限

Thinking 4.6 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”？

当标记为 PTE_COW 的页面被修改时。

内核为什么需要将异常的现场 **Trapframe** 复制到用户空间？

微内核结构在用户态处理异常

Thinking 4.7 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？

微内核结构，内核更加精简、安全，处理也更加灵活。

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？

存入到异常处理栈 UXSTACKTOP 中。

Thinking 4.8 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？

syscall_env_alloc 过程中也可能需要进行异常处理。

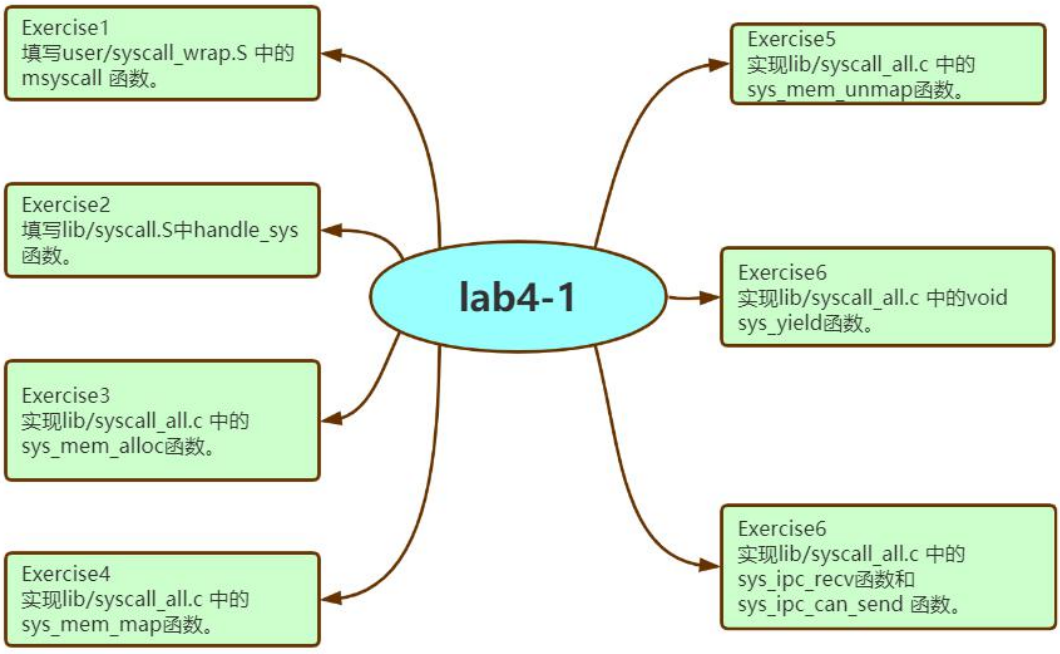
如果放置在写时复制保护机制完成之后会有怎样的效果？

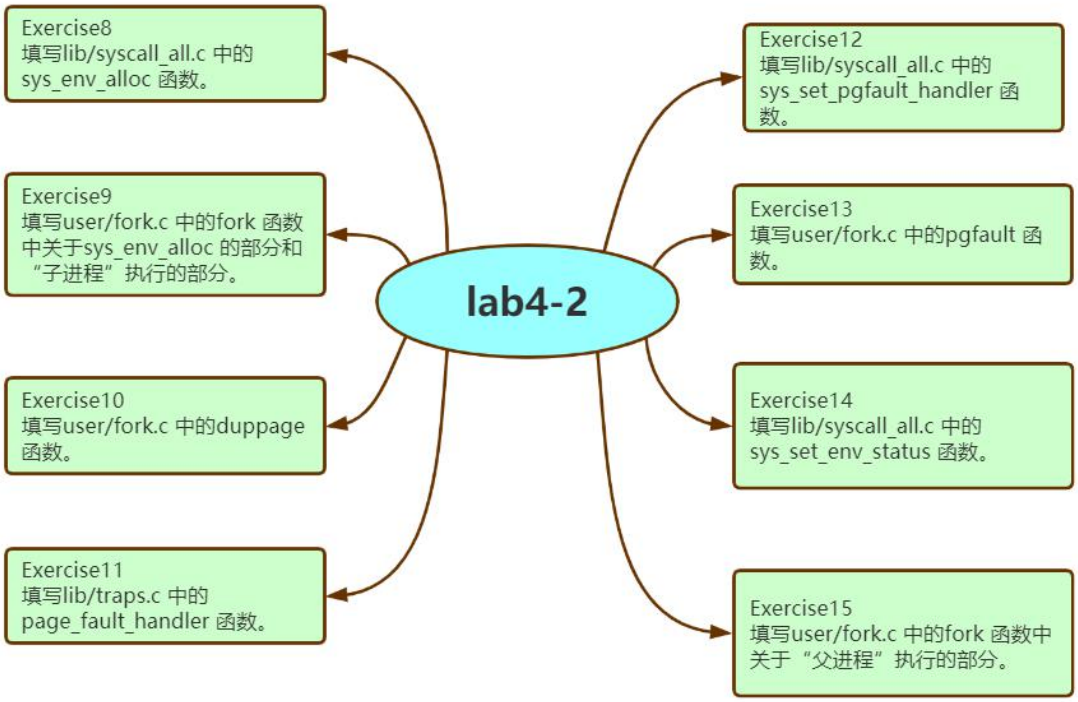
此时进程给__pgfault_handler 变量赋值时就会触发缺页中断，但中断处理没有设置好，故无法进行正常处理。

子进程需不需要对在 entry.S 定义的字__pgfault_handler 赋值？

不需要，子进程复制了父进程中__pgfault_handler 变量值。

二、实验难点图示





本次实验主要包括系统调用、进程间通信和 fork()函数三大部分。其中，进程间通信和 fork()函数都要依靠系统调用。虽然系统调用理解起来没有 fork()难，但却是后面的基础，因此系统调用的实现是本次实验的重点，需要确保所实现函数的正确性，否则会导致后面出错。此外，对于 fork()函数调用一次返回两个返回值的具体实现是一个难点，要理解 sys_env_alloc 函数之后父子进程各自执行。

具体的实验难点如下：

1.handle_sys 函数参数传递的实现，理解好栈的结构，确保每一个参数能够传递到对应的位置、

2.sys_env_alloc，sys_mem_alloc，sys_mem_map 等函数的实现，需要判断异常情况，并返回相应的异常值，并保证其功能能够实现。

3.父子进程各自执行的理解，两个进程共同执行同一段代码，因此需要通过返回值利用分支语句决定父子进程所执行的代码段。

4.写时复制的实现，在 duppage 函数中明确哪些页面需要被保护，哪些无需保护，并在 pgfault 函数中实现缺页时的对页面的复制。

三、体会与感想

本次实验给我带来了很大的挑战，虽然完成实验的时间并不是很长，但由于出现了 bug，导致用了大量时间进行调试，总时间大概要超过 50 小时，花费了整整一个周末。出现 bug 的原因主要有两个，一个是 sys_yield 函数中调度算法出现了问题，没有判断下一个进程是否为 RUNNABLE（在我的实现中，没有将阻塞的进程移出调度队列，而是设置为 NOT_RUNABLE），但这个 bug 比较明显，很容易调试，因此没有花费太长时间；而另一个 bug 则花费了大量时间，主要表现为子进程修改页面时，虽然能够进入并执行缺页的异常处理，但实际上并没有完成 TLB 的重填，导致返回后重新抛出缺页异常，从而出错。一开始我以为是 pgfault 函数或其中相关的系统调用出错，但经过反复检查后发现并没有错误，最终经过助教提示，在以前的 lab 中逐个检查与 TLB 填写相关的函数，发现是 lab2 中 tlb_out 函数中 tlbp 这一汇编语句没有填写所致，补充填写后，bug 即解决。这次 debug 的经历可谓是相当痛苦，不过也给了我一定的启示，不要被课程组划分的 lab 所拘束，而要建立各个执行过程、函数调用次序的整体观念，出现 bug 也要按照整体去检查，不能简单地认为是新写的代码所致。

四、指导书反馈

在 syscall_all 中各个系统调用函数中对于 envid2env 函数的 checkperm 参数并没有限制，但实际上 checkperm 并不能随意填写为 0 或 1，而与后面的是否只由父子进程调用有关，但按照教程的顺序，只有在完成整个实验后才能意识到这一点，因此极易导致一开始填写错误，最后忘记修改的情况。希望能够在教程中提示 envid2env 的 checkperm 参数与后面的部分有关，不能随意填写。

五、残留难点

内核捕获的缺页异常是常规的缺页异常还是写时复制的缺页异常是在何处判断的？是通过 CPU 进行判断，保存在 CP0_CAUSE 中吗？