Lab4实验报告

19376273 陈厚伦

思考题

Thinking4.1

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?

保存现场时，除了k0,k1之外所有的寄存器在有修改前都已经被保存了，k0，k1在MIPS规则中是暂时，随意改变的寄存器，修改也没有关系。

可以直接获取，a0~a3寄存器没有被修改过，所以可以直接用。当然，从内核sp指向的栈区获取也可以。

调用sys开头函数前，人工把参数加载到了sys开头函数认为的位置。

EPC += 4，系统调用后的返回值写入了v0寄存器。系统调用结束后从syscall的下一条开始执行。

Thinking4.2

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?

子进程和父进程共享了代码，而且状态数据相同。

子进程拥有父进程的上下文以及状态，所以从分叉点继续执行。

Thinking4.3

关于 fork 函数的两个返回值，下面说法正确的是：

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

C

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。

UTOP以上的空间用户不可修改，2G系统空间对每个进程都是相同的，用户进程对系统空间没有修改权限，UTOP上面的空间不需要进行保护。

UXSTACKTOP-BY2PG到UXSTACKTOP的是用户进程的异常栈，如果允许写时复制可能导致出现死循环，不可被保护:

（进程异常栈被写-->触发写时复制缺页异常-->需要保存现场-->写进程异常栈--->触发写时复制缺页异常-->.....死循环）

USTACKTOP到USTACKTOP+BY2PG的空间在空间分布图上是invalid memory，不用保护。

UTEXT到USTACKTOP的空间要被保护。

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?

· 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?

vpt指向页表区域第一个页表项（是一个指针），**使用vpt时完全可以按照一级页表机制来使用（自映射机制保证）**，vpd指向一级页表（页目录）的第一个页表项（是一个指针）。可以当做数组，可以把*vpt，*vpd当成一个指针来用。

传入一个虚拟地址va，如果有效，`（*vpd）[va>>22（页目录的索引）]&(~0xfff)`表示二级页表的物理地址，如果有效，`(*vpt)[va >> 12]&(~0xfff)`为va对应的物理页面地址。

mmu.h中

```
extern volatile Pte* vpt[];//指针的指针
extern volatile Pde* vpd[];

.globl vpt
 vpt:
     .word UVPT
     //指向页表区域 二级页表的地址
.globl vpd
vpd:
     //体现自映射 页目录的地址
     .word (UVPT+(UVPT>>12)*4) // 等价(UVPT+UVPT>>10)
```

用户进程没有修改页表项的权限。

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”？

内核为什么需要将异常的现场Trapframe 复制到用户空间？

用户发生写时复制导致缺页中断并处理的过程中，有可能还会发生缺页，所以要“中断重入”，类似于函数调用的方式，要多重处理，直到不再缺页异常。

因为我们的MOS系统使用微内核设计，将缺页中断的处理交给了用户进程，所以用户进程需要读取Trapframe的值获得哪一条指令触发了缺页，从而得到缺的页面是哪一页，并进行调页。用户进程处理完毕恢复现场的时候也要使用Trapframe的数据。

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？

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？

减少内核代码的工作量，用户处理出现错误时还可以进行处理，如果内核出现错误，可能导致系统崩溃。

```
// 恢复除了sp寄存器
.macro RESTORE_SOME
.set    mips1
mfc0    t0, CP0_STATUS
ori     t0, 0x3
xori    t0, 0x3
mtc0    t0, CP0_STATUS
//修改cp0_status
lw      v0, TF_STATUS(sp)
li      v1, 0xff00
and     t0, v1
nor     v1, $0, v1
and     v0, v1
or      v0, t0
mtc0    v0, CP0_STATUS
lw      v1, TF_LO(sp)
mtlo    v1
lw      v0, TF_HI(sp)
lw      v1, TF_EPC(sp)
mthi    v0
mtc0    v1, CP0_EPC
// 用v0 v1寄存器恢复非通用寄存器
lw      $31, TF_REG31(sp)
lw      $30, TF_REG30(sp)
lw      $28, TF_REG28(sp)
lw      $25, TF_REG25(sp)
lw      $24, TF_REG24(sp)
lw      $23, TF_REG23(sp)
```

```
lw      $22, TF_REG22(sp)
lw      $21, TF_REG21(sp)
lw      $20, TF_REG20(sp)
lw      $19, TF_REG19(sp)
lw      $18, TF_REG18(sp)
lw      $17, TF_REG17(sp)
lw      $16, TF_REG16(sp)
lw      $15, TF_REG15(sp)
lw      $14, TF_REG14(sp)
lw      $13, TF_REG13(sp)
lw      $12, TF_REG12(sp)
lw      $11, TF_REG11(sp)
lw      $10, TF_REG10(sp)
lw      $9, TF_REG9(sp)
lw      $8, TF_REG8(sp)
lw      $7, TF_REG7(sp)
lw      $6, TF_REG6(sp)
lw      $5, TF_REG5(sp)
lw      $4, TF_REG4(sp)
lw      $3, TF_REG3(sp)
lw      $2, TF_REG2(sp)
lw      $1, TF_REG1(sp)
// 通过sp寄存器恢复通用寄存器
.endm

.macro RESTORE_ALL
RESTORE_SOME
lw      sp, TF_REG29(sp) /* Deallocate stack  sp已经到达高位，恢复sp，收回栈空间*/
.endm
```

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?

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?

放在syscall_env_alloc之前只需要父进程执行就可以，如果放在syscall_env_alloc之后，父子进程都会执行一遍，没有必要。

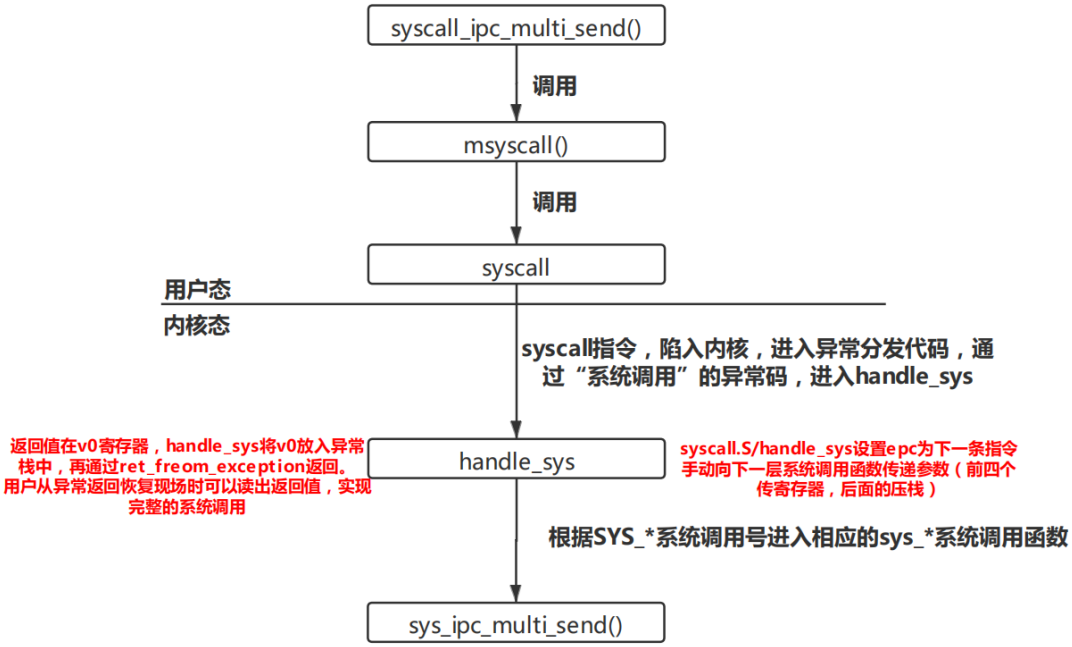
进程给__pgfault_handler变量赋值的时候会触发缺页中断，但是中断处理函数还没有安装，所以不能正常运行。

不用，父进程在fork执行syscall_env_alloc之前已经设置了__pgfault_handler，而且是 .global 的，子进程已经有了父进程中的__pgfault_handler的值。

实验难点图示

> 系统调用的过程

本单元系统调用是一个难点，很多函数长得非常像，但是用户态和内核态需要调用不同的函数。理解系统调用需要理解各个函数哪些由用户调用，哪些由内核调用，并且理清楚函数的调用顺序。下面结合课上测试题`syscall_ipc_multi_send()`系统调用，梳理一下系统调用的过程。

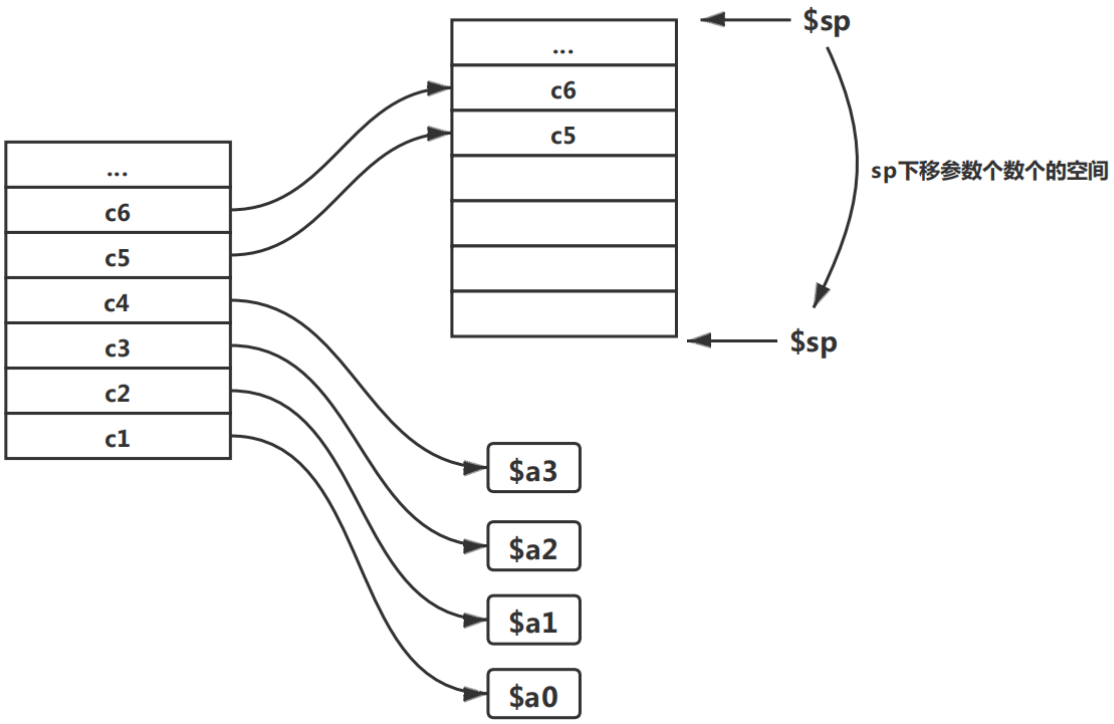


那么，handle_sys是如何通过系统调用号找到内核中相应的系统调用函数呢？（这也是课上测试卡我许久的地方），各个系统调用函数的入口地址构成系统调用表，以系统调用号为索引在这个数组中就可以找到相应的系统调用函数，并跳转即可。

```
sys_call_table:                         // Syscall Table
.align 2
    .word sys_putchar
    .word sys_getenvid
    .word sys_yield
    .word sys_env_destroy
    .word sys_set_pgfault_handler
    .word sys_mem_alloc
    .word sys_mem_map
    .word sys_mem_unmap
    .word sys_env_alloc
    .word sys_set_env_status
    .word sys_set_trapframe
    .word sys_panic
    .word sys_ipc_can_send
    .word sys_ipc_recv
    .word sys_cgetc
    // 添加新的系统调用函数后要在这里声明函数地址的存储空间
    .word sys_ipc_multi_send
```

> 函数参数的传递过程

调用方向被调用方传递参数时，栈指针下移参数数量个的空间，其中前四个栈空间没有传值，其前四个参数传递到`a0~a3`寄存器中，第五个及后面的参数压入栈中。这个过程中C语言调用函数时，已经自动实现了这个过程，但是在汇编函数中，我们调用函数时要根据约定，手工把参数放到合理的位置。



> fork全过程

fork函数是最难的函数，一个进程调用fork，在两个进程中得到两个返回值，与sys_env_alloc函数密切相关。这里的难点有两个：产生两个返回值的机理，fork的流程。

以下是fork函数与sys_env_alloc函数的简略版。要注意到，父进程fork中执行了系统调用syscall_env_alloc，需要从系统调用中恢复现场；子进程被创建，但是没有被调度，需要在调度的时候恢复现场。

父进程执行系统调用syscall_env_alloc，执行完恢复现场时得到系统调用的返回值，即子进程的进程号，从syscall_env_alloc()的下一句执行。而子进程被创建后，复制了父进程的现场信息，但是需要修改两个地方，将env->tf中的v0设置0，将pc设置为epc（epc已经从父进程拷贝而来），这样子进程被调度，恢复现场时，也从fork的syscall_env_alloc的下一句执行，此时子进程v0寄存器也有"返回值"，不过这个返回值是假的，只不过看起来与父进程相似的形式，似乎是从函数返回的结果，其实不然，子进程并没有从任何函数中返回，而是人为修改。

```
syscall
jal ra  --> 子进程被调度时，要恢复现场，pc指向这一句，从这一句执行
nop
```

```
int
fork(void)
{
        // ...
        set_pgfault_handler(pgfault);
        r = syscall_env_alloc();
        if (r < 0) {
            return r;
        }
        if (r == 0) {
            // 子进程执行部分
        }
```

```
        // 父进程执行部分
        return newenvid;

}

int sys_env_alloc(void)
{
        // ...
        r = env_alloc(&e,curenv->env_id);
        if (r!=0) {return r;}
        // 新进程相关属性的设置与现场的保存...
        e->env_tf.pc = e->env_tf.cp0_epc;
        e->env_tf.regs[2] = 0;
        return e->env_id;
}
```

关于fork的流程，指导书已经给出了非常详细的流程图。多阅读代码，对fork逻辑，对函数作用熟悉后，按照流程图写，难度就不是很大了。

> 写时复制与缺页处理

这里要明确几个函数：

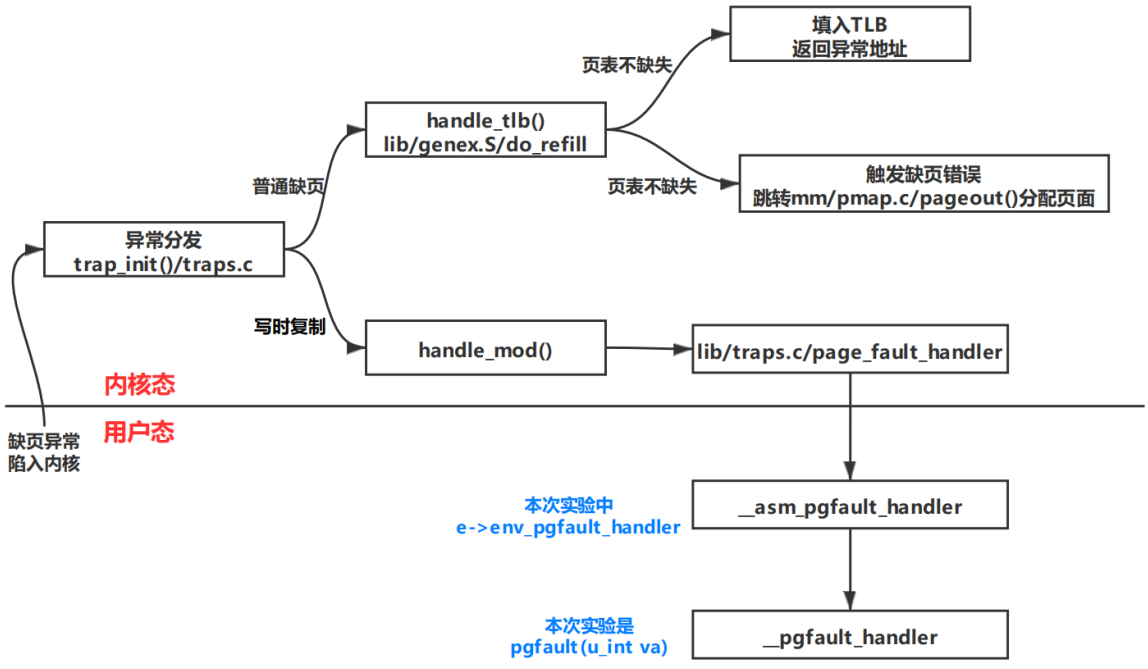
lib/traps.c/page_fault_handler：

这个函数是内核态函数，将现场保存在异常处理栈，设置epc的值为env_pgfault_handler。

user/entry.S/__asm_pgfault_handler：用户态的一个处理缺页的汇编函数，进程的env_pgfault_handler域设置为它。它可以完成缺页处理并恢复现场。

user/entry.S/__pgfault_handler：用户态中缺页处理函数的地址。__asm_pgfault_handler中jal __pgfault_handler中可以看出，这个函数才是真正的缺页处理函数。fork中set_pgfault_handler()也是在设置__pgfault_handler。

处理流程：



我们的微内核设计下，将缺页处理与恢复线程的工作交给了用户。所以与一般的系统调用不太一样。当触发系统调用时，将现场保存下来，也保存了触发缺页时的epc，系统调用结束后，并不会向一般的系统调用，从触发异常的epc执行，而是把epc寄存器设为用户态进程的env_pgfault_handler域的函数__asm_pgfault_handler，结束异常从内核返回时，执行用户态进程的异常处理函数，最终执行完才恢复现场，从触发缺页异常的位置开始执行。

> duppage的顺序问题

duppage中需要分别给父进程与子进程相关的页，设置PTE_COW位，要先给子进程设置，再给父进程设置。因为子进程现在是不可运行的，而父进程是可运行的。

如果先给父进程某页设置了PTE_COW，然后父进程可能修改这一页，此时触发写时复制，父进程重新分配一页，但是这一页不再存在共享的冲突，没有PTE_COW，这个时候map子进程，子进程相应地址也指向这一页，但是有PTE_COW。此后，如果父进程再修改这一页，由于没有PTE_COW，不会触发写时复制，这一共享页被修改了，但是对于子进程来说，不应该被修改，于是执行出现错误。

先给子进程映射则不会出现问题，因为子进程当前是ENV_NOT_RUNNABLE，不会修改这一页。

> 跳来跳去的page_fault处理过程

这内核到用户，跳来跳去，确实，水平，太高了！看了许久，悟透一点玄机。

KERNEL_SP是内核的异常栈的位置，handle_mod中sp指针指向内核栈的保存的现场结构体，这个时候调用page_fault_handler把这个内核的异常的结构体原封不动的保存在用户的异常栈，然后将内核的现场的sp与epc修改，sp用用户异常栈的栈顶，epc为用户的异常处理函数的地址，从handle_mod的系统调用返回值，返回到不是触发异常的地点，而是因为修改了epc跳转到用户处理函数，此时sp指向用户异常栈的保存的现场结构体，用户处理完根据用户异常栈保存的现场返回，这个现场保存的sp与epc才是最初的，才能回到最初触发异常的地方。

体会与感想

本次实验课下部分大概花费了3~4天完成，前期与后期主要都是在理解代码。写代码过程花费时间不是很多，但是写代码的过程也比较迷糊，之后右重新梳理了这一部分的代码，对这部分的理解才更加清楚。

lab4难度较大，有大量功能不同的函数，而且用户态与内核态执行的函数不同，花了许多时间理解了这一点。系统调用与缺页处理这两个过程的流程比较复杂，有多处跳转，还有许多地方需要人为操作寄存器，涉及了许多汇编代码，理解起来有难度。

lab4集大成者了，需要大量之前我们写的函数。在之前的几个lab中，我们都是写的内核代码，分别实现一个个原子功能，在lab4中我们通过系统调用将用户与内核联系起来，使得用户可以通过系统调用实现相应的功能。写这一部分需要对前面几个lab写的函数功能及其调用关系非常熟悉，而且这一lab检测出了前面许多bug，~~真令人头秃~~。

不过仍然收获颇丰！

指导书反馈

`lib/syscall_all.c`中`sys_mem_map`注释似乎有问题，`Perm has the same restrictions as in sys_mem_alloc`，分配内存的时候显然是不能有PTE_COW，但是映射内存时是需要有PTE_COW的，而且`user/fork.c`中函数`duppage`中使用

```
r = syscall_mem_map(syscall_getenvid(),addr,envid,addr,perm);
```

来映射子进程，对于有效可写且非LIBRARY的页需要通过`sys_mem_map`加上PTE_COW位，所以`sys_mem_map`函数中不能够检查PTE_COW位。

残留难点

pgfault中临时虚拟地址可以自定义，我使用了USTACKTOP，但是仍然不是很清楚原因，对MMU图还是理解的不是很透彻。

syscall.S中handle_sys中提到了内核栈指针与用户栈指针，对于两个指针到底指向哪里，不是很清楚。

__asm_pgfault_handler恢复现场与其余中断异常恢复现场不同，要跳转到epc的地址，处理pgfault回到用户，再从用户恢复，跳转次数多，有点混乱。