操作系统原理课程设计报告

姓　　名：　　张钧玮

学　　院：　　计算机学院

专　　业：　　数据科学与大数据专业

班　　级：　　大数据 2102 班

学　　号：　　U202115520

指导教师：　　周正勇

分数	
教师签名	

2024 年 1 月 3 日

目　　录

1 挑战实验 1 （lab1_challenge1_backtrace）

1.1 实验目的

打印用户程序调用栈

对以下给定应用「user/app_print_backtrace.c」,应该实现打印用户程序调用栈的功能。既从理论上推断该程序的函数调用关系应该是「main -> f1 -> f2 -> f3 -> f4 -> f5 -> f6 -> f7 -> f8」，而 print_backtrace(7)的作用就是将以上用户程序的函数调用关系，从最后 f8 向上打印 7 层

```c
/*
 * Below is the given application for lab1_challenge1_backtrace.
 * This app prints all functions before calling print_backtrace().
 */

#include "user_lib.h"
#include "util/types.h"

void f8() { print_backtrace(7); }
void f7() { f8(); }
void f6() { f7(); }
void f5() { f6(); }
void f4() { f5(); }
void f3() { f4(); }
void f2() { f3(); }
void f1() { f2(); }

int main(void) {
  printu("back trace the user app in the following:\n");
  f1();
  exit(0);
  return 0;
}
```

uer/app_print_backtrace.c 代码

1.2 实验内容

1. 梳理用户程序的依赖和函数调用关系，定义 print_backtrace 函数：

从应用出发可以发现用户程序所调用的函数都在 user_lib.h 中定义，user_lib.c 中实现，最后转换成对 do_user_call 函数的调用。因为应用程序中使用的 print_backtrace 函数还没有导入，因此需要在 user_lib.h 中声明 print_backtrace 函数，然后在 user_lib.c 中将函数转换成 do_user_call 这样的操作系统函数调用，观察其他实现的

用户函数例如 printu 都有一个系统宏作为 do_user_call 的函数参数,因此设置传入参数为系统宏 SYS_print_backtrace。

```
// @lab1_challenge1 add
int print_backtrace(int depth);
```

user/user_lib.h 代码增加行

```
// @lab1_challenge1 add
int print_backtrace(int depth) {
return do_user_call(SYS_print_backtrace, depth, 0, 0, 0, 0, 0, 0);
}
```

user/user_lib.c 代码增加行

2. 声明系统宏和系统调用号：

重新执行./obj/app_print_backtrace 发现不再报错但是显示为未止的系统调用。这提示我们需要修改增加内核代码，使得 backtrace 可以正常被调用。首先需要在 kernel/syscall.h 中声明系统宏和系统调用号，然后在 kernel/syscall.c 中实现系统调用。

```
// @lab1_challenge1 add
//
// [a0]: the syscall number; [a1] ... [a7]: arguments to the syscalls.
// returns the code of success, (e.g., 0 means success, fail for otherwise)
//
long do_syscall( long a0, long a1, long a2, long a3, long a4, long a5, long a6,long a7 ) {
    switch ( a0 ) {
    case SYS_user_print:
        return sys_user_print( (const char *) a1, a2 );
    case SYS_user_exit:
        return sys_user_exit( a1 );
    case SYS_print_backtrace:
        return sys_backtrace( a1 );
    default:
        panic( "Unknown syscall %ld \n", a0 );
    }
}
```

kernel/syscall.c 代码增加行

3. 参照 sys_user_print 函数设计 sys_user_backtrace 函数：

首先需要为 sys_user_backtrace 函数引入 elf_get_funname 函数，这个函数在 elf.h 中定义，作用是通过传入的函数地址获取函数名。sys_user_backtrace 的实现思路是通过当前进程的 trapframe 获取当前函数的栈指针，然后通过栈指针获取当前函数的

函数地址，再通过调用 elf_get_funname 函数根据函数地址获取函数名。通过循环获取函数名就可以实现打印函数调用栈调用栈的功能。设计是如果遍历到 mian 函数返回当前函数调用栈的层数，sys_user_backtrace 的正常返回值是 depth。

```
// added @lab1_challenge1
ssize_t sys_user_backtrace( uint64 depth ) {
    int i, off;
    uint64 fun_sp = current->trapframe->regs.sp + 32;
    uint64 fun_pa = fun_sp + 8;
    i = 0;
    while ( i < depth ) {
        if ( elf_get_funname( *(uint64 *) fun_pa ) == 0 ) return i;
        fun_pa += 16;
        i++;
    }
    return i;
}
```

kernel/syscall.c 代码增加行

因为 sys_user_backtrace 要求只能在 syscall.c 内部进行调用，因此不需要再在 syscall.h 中进行声明，这是操作系统的保护性措施，目的在于防止用户程序直接调用系统调用。

4. elf_get_funname 函数的实现

根据传入的函数地址在 elf 文件的符号表中通过遍历符号表查找对应函数，如果地址位于符号表的某个函数的地址范围内，则说明传入地址对应的就是这个函数，打印函数名并返回 1。如果发现传入地址是 main 函数则返回 0.

```
// added @lab1_challenge1
int elf_get_funname( uint64 ret_addr ) {
    int len_count = sym_count;
    for ( int i = 0; i < sym_count; i++ ) {

        if ( ret_addr >= symbols[ i ].st_value &&
           ret_addr < symbols[ i ].st_value + symbols[ i ].st_size ) {
            sprint( "%s\n", sym_name_pool[ i ] );
           if ( strcmp( sym_name_pool[ i ], "main" ) == 0 ) return 0;
            return 1;
        }
    }
    return 1;
}
```

elf_get_funname 函数实现

1.3 实验调试与心得

实验的要求的知识点特别多。

在线程调用中，因为 current->trapframe->regs.sp 指向的是当前进程的栈指针，通过观察 sp 结构可以以及使用 sprint("%dn",res)之类的工具进行打印调试知道通过 sp 指向地址和 main 函数地址差了 32 个字节，因此 uint64 fun_sp = current->trapframe->regs.sp + 32 可以获取到 main 函数的地址，而 uint64 fun_pa = fun_sp + 8 就可以获得第一次调用 backtrace 函数的函数的地址，然后根据这个地址依次因为栈帧的长度是 16 字节，+16 字节遍历 elf 的符号表就可以正确地遍历函数的调用栈。

另外打印符号表同样是一个难点，因为我们需要知道对应函数的起始地址，而这事实上与 PKE 的大小端策略有关，如果是小端则起始地址是从小地址开始，如果是大端则计算出的 pa 的起始地址还需要增加一个偏移值才是正确的函数地址。通过阅读 elf.h，发现 elf_header_t 里贴心地给我标了注释，PKE 使用的是小端模式。

事实上关于这关存在两种思路，一种是通过追踪函数调用的栈帧来获取函数地址再在 elf 里面查表，一种是读取 elf 的 Section Header Table 来追踪函数。后者因为需要深入 elf 文件系统，我能力有限经过短暂的尝试选择了放弃，但是前者同样是一种讨巧的方法，因为栈帧的长度不是固定的，理论上需要设计算法获取栈帧的长度来遍历函数调用栈，但是我选择设计每次传入地址以后在获取函数名的函数内部对地址进行反复自增直到获取到新的函数名为止，这种方式同样实现了函数调用栈的遍历，但是不需要设计算法获取栈帧的长度，因此更加简单。最后，事实上本题栈帧的长度是固定的，因为函数的调用栈是用来存放传入参数的，而测试用的程序因为不存在参数，长度固定是 16 字节。所以无论是否对栈帧长度进行计算，都可以正确地遍历函数调用栈。