1

编写模块fp，使用了浮点运算

```c
static int __init fp_init(void)
{
    float b = 1.1;
    printk("a: %d\n", a + (int)b);
    return 0;
}
```

硬浮点

编译选项中添加CFLAGS_fp.o=-mhard-float（为避免浮点部分被优化，设置-O0）

```makefile
obj-m += fp.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) CFLAGS_fp.o="-mhard-float -O0" modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

反汇编结果：

可以看到汇编中使用了flds, fstps等x87 FPU指令

```
root@ubuntu:~/lab07/fp# objdump -d fp.o

fp.o:     file format elf64-x86-64


Disassembly of section .init.text:

0000000000000000 <init_module>:
   0:   e8 00 00 00 00          callq  5 <init_module+0x5>
   5:   55                      push   %rbp
   6:   48 89 e5                mov    %rsp,%rbp
   9:   48 83 ec 10             sub    $0x10,%rsp
   d:   d9 05 00 00 00 00       flds   0x0(%rip)        # 13 <init_module+0x13>
  13:   d9 5d fc                fstps  -0x4(%rbp)
  16:   d9 45 fc                flds   -0x4(%rbp)
  19:   d9 7d f6                fnstcw -0xa(%rbp)
  1c:   0f b7 45 f6             movzwl -0xa(%rbp),%eax
  20:   80 cc 0c                or     $0xc,%ah
  23:   66 89 45 f4             mov    %ax,-0xc(%rbp)
  27:   d9 6d f4                fldcw  -0xc(%rbp)
  2a:   db 5d f0                fistpl -0x10(%rbp)
  2d:   d9 6d f6                fldcw  -0xa(%rbp)
  30:   8b 55 f0                mov    -0x10(%rbp),%edx
  33:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 39 <init_module+0x39>
  39:   01 d0                   add    %edx,%eax
  3b:   89 c6                   mov    %eax,%esi
  3d:   48 c7 c7 00 00 00 00    mov    $0x0,%rdi
  44:   e8 00 00 00 00          callq  49 <init_module+0x49>
  49:   b8 00 00 00 00          mov    $0x0,%eax
  4e:   c9                      leaveq
  4f:   c3                      retq

Disassembly of section .exit.text:

0000000000000000 <cleanup_module>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   90                      nop
   5:   5d                      pop    %rbp
   6:   c3                      retq
```

软浮点

编译选项中添加CFLAGS_fp.o="-msoft-float"

```
obj-m += fp.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) CFLAGS_fp.o="-msoft-float -O0" modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

make时有warning："__fixsfsi" undefined

```
root@ubuntu:~/lab07/fp# make
make -C /lib/modules/4.19.0+/build M=/home/lzzz/lab07/fp CFLAGS_fp.o="-msoft-float -O0" modules
make[1]: Entering directory '/home/lzzz/linux'
  CC [M]  /home/lzzz/lab07/fp/fp.o
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: "__fixsfsi" [/home/lzzz/lab07/fp/fp.ko] undefined!
  LD [M]  /home/lzzz/lab07/fp/fp.ko
make[1]: Leaving directory '/home/lzzz/linux'
```

反汇编结果：

原来应该出现浮点调用指令的地方变成了一个函数调用callq，由于没有链接调用地址未知，由warning提示这个函数调用应该是__fixsfsi，功能是将float转换为int。与编写的模块内容相符。

```
root@ubuntu:~/lab07/fp# objdump -d fp.o

fp.o:     file format elf64-x86-64


Disassembly of section .init.text:

0000000000000000 <init_module>:
   0:   e8 00 00 00 00          callq  5 <init_module+0x5>
   5:   55                      push   %rbp
   6:   48 89 e5                mov    %rsp,%rbp
   9:   48 83 ec 08             sub    $0x8,%rsp
   d:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 13 <init_module+0x13>
  13:   89 45 fc                mov    %eax,-0x4(%rbp)
  16:   8b 45 fc                mov    -0x4(%rbp),%eax
  19:   50                      push   %rax
  1a:   e8 00 00 00 00          callq  1f <init_module+0x1f>
  1f:   48 83 c4 08             add    $0x8,%rsp
  23:   89 c2                   mov    %eax,%edx
  25:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 2b <init_module+0x2b>
  2b:   01 d0                   add    %edx,%eax
  2d:   89 c6                   mov    %eax,%esi
  2f:   48 c7 c7 00 00 00 00    mov    $0x0,%rdi
  36:   e8 00 00 00 00          callq  3b <init_module+0x3b>
  3b:   b8 00 00 00 00          mov    $0x0,%eax
  40:   c9                      leaveq
  41:   c3                      retq

Disassembly of section .exit.text:

0000000000000000 <cleanup_module>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   90                      nop
   5:   5d                      pop    %rbp
   6:   c3                      retq
```

分析

从上面的结果可以看出，使用硬浮点编译的代码会使用FPU指令，需要机器有支持协处理器，而使用软浮点选项编译的程序会使用glibc的函数来模拟浮点指令，不需要FPU支持。

2

分解DO_INVALID_OP宏

traps.c中定义了DO_ERROR宏来统一建立一些错误信号的错误处理函数，这里将do_invalid_op分离出来。

修改如图：

```
  #define DO_ERROR(trapnr, signr, str, name)                          \
  dotraplinkage void do_##name(struct pt_regs *regs, long error_code) \
  {                                                                    \
          do_error_trap(regs, error_code, str, trapnr, signr);        \
  }

+ #define DO_INVALID_OP(trapnr, signr, str)                           \
+ dotraplinkage void do_invalid_op(struct pt_regs *regs, long error_code)\
+ {                                                                    \
+         do_error_trap(regs, error_code, str, trapnr, signr);    \
+ }
+
  DO_ERROR(X86_TRAP_DE,     SIGFPE,  "divide error",           divide_error)
  DO_ERROR(X86_TRAP_OF,     SIGSEGV, "overflow",               overflow)
~ // DO_ERROR(X86_TRAP_UD,     SIGILL,  "invalid opcode",       invalid_op)
+ DO_INVALID_OP(X86_TRAP_UD,SIGILL,  "invalid opcode");
  DO_ERROR(X86_TRAP_OLD_MF, SIGFPE,  "coprocessor segment overrun",coprocessor_segment_overrun)
  DO_ERROR(X86_TRAP_TS,     SIGSEGV, "invalid TSS",            invalid_TSS)
  DO_ERROR(X86_TRAP_NP,     SIGBUS,  "segment not present",    segment_not_present)
  DO_ERROR(X86_TRAP_SS,     SIGBUS,  "stack segment",          stack_segment)
  DO_ERROR(X86_TRAP_AC,     SIGBUS,  "alignment check",        alignment_check)

```

3

编写普通程序：

```c
#include <stdio.h>

int main()
{
        printf("Hello world!\n");
        return 0;
}

```

编译生成a.out，反汇编查看main函数位置之后，在二进制指令中加上一个非法指令FF，再反汇编查看结果：

```
000000000000063a <main>:
 63a:   55                      push   %rbp
 63b:   48 89 e5                mov    %rsp,%rbp
 63e:   48 8d 3d 9f 00 00 00    lea    0x9f(%rip),%rdi        # 6e4 <_IO_stdin_used+0x4>
 645:   e8 c6 fe ff ff          callq  510 <puts@plt>
 64a:   ff                      (bad)
 64b:   b8 00 00 00 00          mov    $0x0,%eax
 650:   c3                      retq
 651:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 658:   00 00 00
 65b:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

可以看到其中多出了一条指令：ff　　(bad)

执行的结果是：

```
Hello world!
Illegal instruction (core dumped)
```

程序在执行printf之后，在ff指令处发生invalid op

修改do_error_trap函数：

在发生SIGILL时打印Never Giveup，由于trap之后回到原来指令，所以将ip加1，跳过这条指令，然后返回。

```
  static void do_error_trap(struct pt_regs *regs, long error_code, char *str,
  			unsigned long trapnr, int signr)
  {
  	siginfo_t info;

  	RCU_LOCKDEP_WARN(!rcu_is_watching(), "entry code didn't wake RCU");

  	/*
  	 * WARN*()s end up here; fix them up before we call the
  	 * notifier chain.
  	 */
  	if (!user_mode(regs) && fixup_bug(regs, trapnr))
  		return;

+ 	if (signr == SIGILL) {
+ 		printk("Never Giveup");
+ 		regs->ip += 1;
+ 		return;
+ 	}
+
  	if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, signr) !=
  			NOTIFY_STOP) {
  		cond_local_irq_enable(regs);
  		clear_siginfo(&info);
  		do_trap(trapnr, signr, str, regs, error_code,
  			fill_trap_info(regs, signr, trapnr, &info));
  	}
  }

```

结果：

```
lzzz@ubuntu:~/lab07/sig$ dmesg
lzzz@ubuntu:~/lab07/sig$ ./a.out
Hello world!
lzzz@ubuntu:~/lab07/sig$ dmesg
[  132.839667] Never Giveup
lzzz@ubuntu:~/lab07/sig$
```

4

编写模块task，想法是将已经遍历已经存在的用户进程，对某个用户的进程设置cpuaffinity；然后劫持系统调用sys_execve，对之后建立的进程设置cpuaffinity。

在include/linux/cpumask.h中查看cpumask接口，在kernel/sched/core.c中EXPORT_SYMBOL内核中的sched_setaffinity

```
  }
+ EXPORT_SYMBOL(sched_setaffinity);

```

模块内容如下：

```c
#include <linux/sched.h>
#include <asm/uaccess.h>
#include <linux/cpumask.h>

#define CR0_WP 0x00010000
MODULE_LICENSE("GPL");
extern void *sys_call_table[];

static int uid;
static int cpunum;
module_param(uid, int, 0644);
module_param(cpunum, int, 0644);

unsigned long cr0;

// extern long sched_setaffinity(pid_t, const struct cpumask *);

asmlinkage int(*original_call) (const char *, char *const *, char *const *);



asmlinkage int my_execve(const char *filename, char *const argv[], char *const envp[])
{
	if (current_uid().val == uid) {
		struct cpumask cpumask;
		cpumask_clear(&cpumask);
		cpumask_set_cpu(cpunum, &cpumask);
		sched_setaffinity(current->pid, &cpumask);
	}
	return original_call(filename, argv, envp);
}


int init_module()
{
	struct cpumask cpumask;
	struct task_struct *ts;
	cpumask_clear(&cpumask);
	cpumask_set_cpu(cpunum, &cpumask);
	for_each_process(ts) {
		if (task_uid(ts).val == uid)
			sched_setaffinity(ts->pid, &cpumask);
	}

	cr0 = read_cr0();
	write_cr0(cr0 & ~CR0_WP);
	original_call = sys_call_table[__NR_execve];
	sys_call_table[__NR_execve] = my_execve;
	write_cr0(cr0);
	return 0;
}


/*
```

编译之后insmod成功.