ARMv8 异常处理研究

--杨森

一、项目目标

1.1 在用户态监控进程接管相关异常信号

1.2 在接管流程中回溯调用栈以及输出寄存器信息；更进一步发生 SIGSEGV 之后，是否可以让进程不退出，进行合理的异常处理后继续运行；

二、项目实施

2.1 异常现场输出

2.1.1 原理

2.1.1.1 内核对异常现场的保存

通过阅读 Linux 内核 arm64 的异常处理流程以及相关 arm64 体系资料得知，当进程发生同步异常（sigsegv，sigbus）时，内核会进行异常现场的保护，将处理器上下文存入栈中。具体操作如下

```
kernel_ventry   0, sync                 // Synchronous 64-bit EL0
kernel_ventry   0, irq                  // IRQ 64-bit EL0
kernel_ventry   0, fiq_invalid          // FIQ 64-bit EL0
kernel_ventry   0, error                // Error 64-bit EL0
```

图 2.1 异常向量表，内核通过 kernel_ventry 宏跳转到 sync

```
        .align  0
SYM_CODE_START_LOCAL_NOALIGN(el0_sync)
        kernel_entry 0
        mov     x0, sp
        bl      el0_sync_handler
        b       ret_to_user
SYM_CODE_END(el0_sync)
```

图 2.2 内核中 sync 进行的操作

```
        .macro kernel_ventry, el, label, regsize = 64
        .align 7
#ifdef CONFIG_UNMAP_KERNEL_AT_EL0
        .if     \el == 0
alternative_if ARM64_UNMAP_KERNEL_AT_EL0
        .if     \regsize == 64
        mrs     x30, tpidrro_el0
        msr     tpidrro_el0, xzr
        .else
        mov     x30, xzr
        .endif
alternative_else_nop_endif
        .endif
#endif

        sub     sp, sp, #S_FRAME_SIZE
#ifdef CONFIG_VMAP_STACK
        /*
         * Test whether the SP has overflowed, without corrupting a GPR.
         * Task and IRQ stacks are aligned so that SP & (1 << THREAD_SHIFT)
         * should always be zero.
         */
        add     sp, sp, x0                      // sp' = sp + x0
        sub     x0, sp, x0                      // x0' = sp' - x0 = (sp + x0) - x0 = sp
        tbnz    x0, #THREAD_SHIFT, 0f
        sub     x0, sp, x0                      // x0'' = sp' - x0' = (sp + x0) - sp = x0
        sub     sp, sp, x0                      // sp'' = sp' - x0 = (sp + x0) - x0 = sp
        b       el\()\el\()_\label

0:
```

图 2.3 kernel_ventry 宏

通过这几张图可以看到，内核在处理同步异常时，会先调用 kernel_ventry 宏，主要是将 sp 减去#S_FRAME_SIZE，为存储异常现场留出足够的栈空间，之后使用 b 指令提转到 el0_sync，而在 el0_sync 中，内核先调用 kernel_entry 宏将异常现场存入栈（具体的 kernel_entry 定义由于太大附在了末尾），再将异常现场的地址作为参数调用异常处理函数。

2.1.1.2 获取被保存的异常现场

通过上面的研究指导，异常现场是被内核保留在一个内存空间中的，我只需要在异常处理函数中获得这个内存空间的地址，就可以按照一定的结构获取异常现场处理器各寄存器的值，进而对异常进行一定的处理。

在陈炯前辈的指导下，我继续查阅 Linux 编程资料，发现在 Linux 系统编程更高级的信号编程 sigaction 中，信号处理函数有 3 个参数，其中第三个参数 void * context 是一个指向 ucontext_t 结构体的指针，而 ucontext_t 结构体中的成员 uc_mcontext，是一个包含异常现场的结构体，两个结构体在 aarch64 体系下的定义如下：

```
typedef struct
  {
    unsigned long long int __ctx(fault_address);
    unsigned long long int __ctx(regs)[31];
    unsigned long long int __ctx(sp);
    unsigned long long int __ctx(pc);
    unsigned long long int __ctx(pstate);
    /* This field contains extension records for additional processor
```

```
            state such as the FP/SIMD state.    It has to match the definition
            of the corresponding field in the sigcontext struct, see the
            arch/arm64/include/uapi/asm/sigcontext.h linux header for details.    */
        unsigned char __reserved[4096] __attribute__ ((__aligned__ (16)));
    } mcontext_t;

typedef struct ucontext_t
    {
        unsigned long __ctx(uc_flags);
        struct ucontext_t *uc_link;
        stack_t uc_stack;
        sigset_t uc_sigmask;
        mcontext_t uc_mcontext;
    } ucontext_t;
```

所以，只需要对第三个参数进行指针类型转化，再对其内容进行访问就可以得到异常现场。虽然中间的过程并未弄明白，但感觉应该是内核将保存的异常现场地址通过这个参数传入到了异常处理函数。

2.1.2 异常现场输出

具体代码在附件 2 中，这里给出实施结果，如图 2-4

```
Parent process is still working
process[1610] is going to exit because of SIGSEGV
异常发生时程序现场如下:
寄存器X0的值为: 0x0
寄存器X1的值为: 0x1
寄存器X2的值为: 0x2
寄存器X3的值为: 0x3
寄存器X4的值为: 0x4
寄存器X5的值为: 0x4
寄存器X6的值为: 0x0
寄存器X7的值为: 0x0
寄存器X8的值为: 0xdc
寄存器X9的值为: 0xffffffffffff
寄存器X10的值为: 0x0
寄存器X11的值为: 0x0
寄存器X12的值为: 0x7f89e8d228
寄存器X13的值为: 0x0
寄存器X14的值为: 0x0
寄存器X15的值为: 0x6fffff4a
寄存器X16的值为: 0x10
寄存器X17的值为: 0x11
寄存器X18的值为: 0x7fe2cd0d40
寄存器X19的值为: 0x556dbc0c08
```

图 2-4 异常现场

为了对该异常输出的结果的正确性进行验证，我在触发异常的指令前使用内嵌汇编将 x0，x1，x2，x3，x4，x16，x17 的值设为 0，1，2，3，4，16，17。对照下，输出的异常现场正确。

2.2 栈回溯

2.2.1 原理

1. 在 arm64 体系结构中每次函数调用都会有函数自己的栈帧，被调用的函数会将父函数的栈顶以及在父函数的返回地址存入本函数栈帧的栈顶，再将 fp 与 sp 指向本函数栈帧的栈顶。具体情形如下图 2-5：

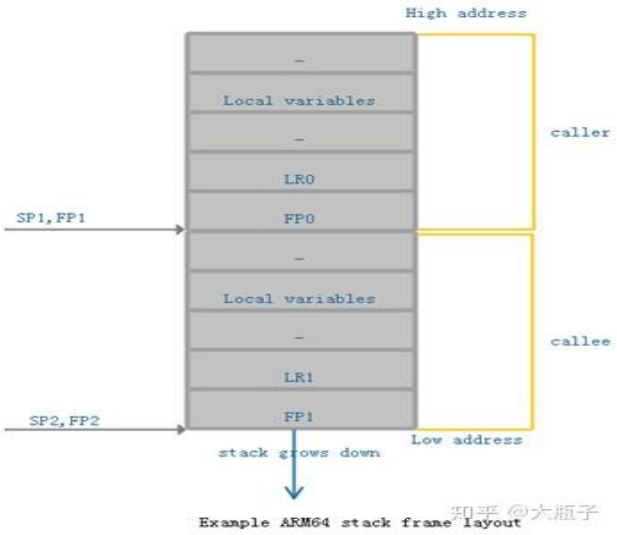

图 2-5 arm64 栈帧

因此，通过异常现场的 fp 与 sp 寄存器的值，不断获取父函数的栈帧与 lr，再对 lr 进行解析，就可以回溯异常函数的调用过程。

2. 并不是所有的函数都会有上述的栈帧结构，同时，上述所讲的方法是建立在进程的栈空间没有被破坏的情况下的，一旦栈被破坏，一切都无法实现。

对于被调用函数没有栈帧的情况，当一个函数是最后一个被调用的函数时，编译器会进行优化，不再将父函数栈顶与本函数返回地址存入栈中。具体情况如图 2-6

```
246 0000000000000b60 <sigsegv_error2>:
247      b60:       d10043ff        sub     sp, sp, #0x10
248      b64:       f90007ff        str     xzr, [sp, #8]
249      b68:       f94007e0        ldr     x0, [sp, #8]
```

图 2-6 最后调用的函数没有一致的栈帧结构

对于这种情况，我设想的处理很简单，只需要在进行栈回溯前进行判断，如果异常现场的 sp 与 fp 不相等，说明该函数没有上述的栈帧结构，对这种情况下进行栈回溯，调整一下循环的次序就行了。

3. 但是在进行功能检测时发现，当异常函数是最后一个被调用时，异常处理函数在接管程序后，进程的栈空间总是被破坏，具体如图 2-7。进行栈回溯前，可以看到异常现场 fp 与 sp 不相同，说明，fp 与 lr 未入栈，但这时再根据 fp 进行栈回溯时，得到了乱码。

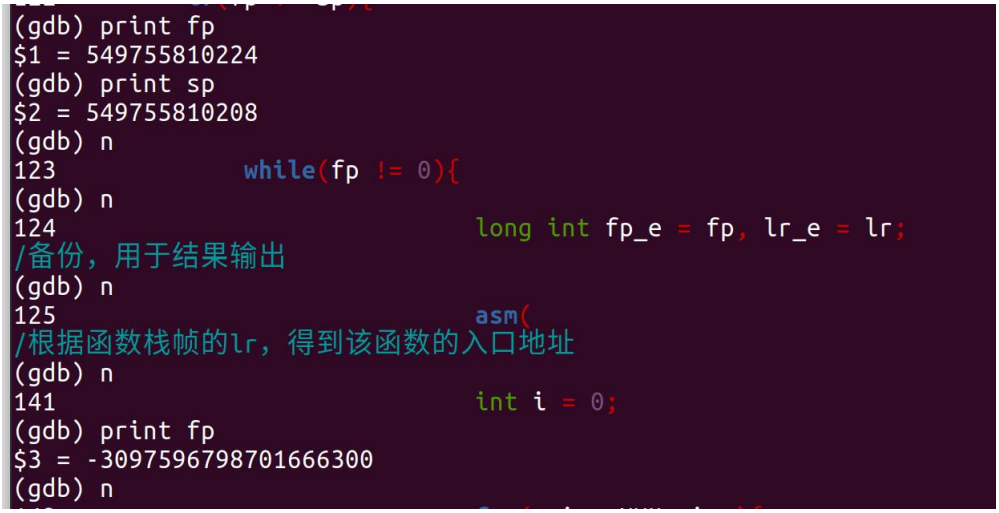

图 2-7 异常函数最后调用，栈被破坏

4. 为了保证程序能进行栈回溯，需要在编译时加上 -fstack-protector-all 选项，但是在加上这个选项后，即使异常函数时最后一个被调用的，其 fp 和 sp 也会入栈，我设计的 fp ！= sp 的情况分支完全没了作用，但是仍然能实现两种情况的栈回溯。

2.2.2 实现

具体代码在附件 2，具体结果需要与地址解析模块一起给出。

2.3 地址解析

2.3.1 解析原理

在上述的基于 fp 的栈回溯中，我可以通过一层一层的 fp 向上找出每个函数的返回地址，同时也异常函数现场的 pc，我需要对这些地址进行解析，得到每个函数的名称和跳转位置在本函数中的偏移，真正实现对异常函数的 backtrack。

方案一：

1. 最初我想的是在异常处理函数中为每个函数名称和其地址建立字典，再将 lr 地址减 4，得到 bl 指令的地址，根据 bl 指令编码和 lr 得到运行时将要跳转的函数的地址（这里不用担心地址会匹配不上，都是运行时的函数地址），通过查阅字典得到对应函数名，再通过计算得到跳转位置或者错误位置再本函数的偏移。在我自己实验的程序中成功实现了输出函数名与偏移，但是很明显，在稍微复杂的程序中根本行不通，一是函数多起来时，建立起字典太复杂。二是这个方案得到函数地址的过程，只能对 bl 指令进行解析，在大型项目中，如果跳转距离过远，使用了 blr 指令（如 main 函数的调用），根本无法解析。这个方案只是我在不了解 elf 文件时自己摸索的一种替代。

方案二：

通过读取本程序的 elf 文件，结合 elf 文件中的.symtab 和.strtab 对 lr 和 pc 解析得到函数名称与函数地址，再得到偏移。读取 elf 文件的原理参照了附件 3 里的文章。具体操作如下：

1. 父函数中建立文件描述符，打开本程序的 elf 文件。

2. 建立 Elf64_Ehdr 结构体，读取本 elf 头部存入该结构体。

3. 根据 elf 头部中的 e_shoff 成员确定 elf 段表的位置，再结合 e_shnum 成员和 sizeof(Elf64_Shdr)得到段表大小，分配空间存储本 elf 的段表。

4. 根据 elf 头部中的 e_shstrndx 成员找到.shstrtab 对应的段表中的下标，找到.shtrtab 节并分配空间存储。

5. 便利段表，根据 Elf64_Shdr 结构体中的 sh_name（节名称在 shstrtab 中的偏移）和存储的 shstrtab 找到.symtab 和.strtab 在段表中的下标，分配空间存储 symtab 和 strtab。

6. 根据栈回溯得到的 pc 值，与 symtab 表中每一项的长远 st_value 和 st_value+st_size 比对，根本不行，栈回溯得到的地址是程序实际运行的地址，而 elf 文件中的地址都是相对于本 elf 文件头的偏移，所以 pc 还需要减去装载地址 base 才能正常比较。

7. 对于 base 的获得，由于编写的函数都是在.text 段，其装载地址都相同，我将 main 函数运行地址减去 main 函数在 elf 文件中的偏移，即得到了 base。

8. 遍历 symtab 表，将栈回溯得到的 pc 与 base+st_value 和 base+st_size+st_value 比较，在其范围内，则说明调用的是该项对应的函数。

9. 根据该项的 st_name，在 strtab 中找到函数名，再经过计算得到再本函数的偏移。

10. 输出，完成 bt。

```
//elf 文件头
typedef struct
{
  unsigned char    e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf64_Half   e_type;              /* Object file type */
  Elf64_Half   e_machine;           /* Architecture */
  Elf64_Word e_version;             /* Object file version */
  Elf64_Addr  e_entry;        /* Entry point virtual address */
  Elf64_Off    e_phoff;       /* Program header table file offset */
  Elf64_Off    e_shoff;       /* Section header table file offset */
  Elf64_Word e_flags;         /* Processor-specific flags */
  Elf64_Half   e_ehsize;      /* ELF header size in bytes */
  Elf64_Half   e_phentsize;         /* Program header table entry size */
  Elf64_Half   e_phnum;             /* Program header table entry count */
  Elf64_Half   e_shentsize;         /* Section header table entry size */
  Elf64_Half   e_shnum;             /* Section header table entry count */
  Elf64_Half   e_shstrndx;          /* Section header string table index */
} Elf64_Ehdr;
//段表项
typedef struct
{
  Elf64_Word sh_name;       /* Section name (string tbl index) */
  Elf64_Word sh_type;       /* Section type */
  Elf64_Xword       sh_flags;       /* Section flags */
  Elf64_Addr  sh_addr;        /* Section virtual addr at execution */
  Elf64_Off    sh_offset;           /* Section file offset */
  Elf64_Xword       sh_size;        /* Section size in bytes */
  Elf64_Word sh_link;         /* Link to another section */
```

```
    Elf64_Word sh_info;         /* Additional section information */
    Elf64_Xword       sh_addralign;         /* Section alignment */
    Elf64_Xword       sh_entsize;           /* Entry size if section holds table */
} Elf64_Shdr;

//符号表项
typedef struct
{
    Elf64_Word st_name;         /* Symbol name (string tbl index) */
    unsigned char     st_info;          /* Symbol type and binding */
    unsigned char st_other;             /* Symbol visibility */
    Elf64_Section     st_shndx;         /* Section index */
    Elf64_Addr  st_value;         /* Symbol value */
    Elf64_Xword       st_size;          /* Symbol size */
} Elf64_Sym;
```

2.3.2 地址解析实现

具体代码在附件 3，这里仅给出结果

```
error address: 0x0
*****************backTrace**********************
调用次序        函数地址        函数名称        偏移
0:              (0x55573f0d40)  sigsegv_error2  (+0x28)
fp: 7ffa813ee0 sp: 7ffa813ec0
1:              (0x55573f0ce4)  sigsegv_error1  (+0x20)
fp: 7ffa813f10 sp: 7ffa813ee0
2:              (0x55573f0c8c)  func1   (+0x28)
fp: 7ffa813f30 sp: 7ffa813f10
3:              (0x55573f0bbc)  main    (+0x8c)
fp: 7ffa813ff0 sp: 7ffa813f30
进程[1579]在上述函数调用中出现了SIGSEGV错误，将退出进程
```

图 2-8 elf 解析栈回溯

2.4 偏移解析

在错误定位中，还是找到错误在源文件的行数比较方便，可以通过上述 backtrack 的结果得到具体错误或函数调用在源文件地行数。具体如图 2-9

```
yangsen@raspberrypi:~/armv8_irq $ nm final|grep -i sigsegv_error2
0000000000000d40 T sigsegv_error2
yangsen@raspberrypi:~/armv8_irq $ addr2line -e final 0xd68
/home/yangsen/vscode/dj_armv8_irq/sigsegv.c:71
```

图 2-9 找到错误在源文件的行数

其中 final 是本 elf 文件，0xd68 是第一条指令得到的地址加上栈回溯的偏移。

三、 收获

本次的暑期项目实训带给我的收获是匪浅的，学习到的内容大致可分为两个方向。

1. 对 armv8 体系架构的学习和了解

本次项目的实验与操作都是基于 armv8 的 64 位体系结构进行的，与在学校中学习的 32 位的 S3C2440A 有相同也有不同，通过暑期对 aarch64 的学习与在项目中的实践，我也了解并熟练掌握了 aarch64 的 ABI，同时本项目在交叉开发平台上的实验也对我的嵌入式学习有着很大的帮助。

2. 对 Linux 系统以及 Linux 下程序运行的更深层次理解

虽然我大二一直是嵌入式方向的学生，但在本次项目后我才明白我所学的嵌入式还差得远，在之前的校园学习中，我主要的重心仅是在 IDE 下的代码编写以及对嵌入式处理器的了解与简单实验，甚至 GDB 也很少使用，对 Linux 平台以及 Linux 下程序运行原理的掌握对于一个我这个嵌入式方向的学生远远不够。而在此次项目中，通过阅读 Linux 内核 aarch64 体系中异常处理部分的源码，查阅学习 elf 文件，我对 Linux 下程序的运行原理以及 Linux 内核的运行有了更深的了解与体会。

我在本项目中学到的不止是上面所说的理论知识，更有在遇到陌生问题时解决问题的能力，通过在网上搜寻资料，自己理解，辨别和学习来解决遇到的问题。这也是我在本项目中所提升能力的一方面。

本次项目最让我明白的是我对计算机知识掌握的匮乏，因为见的少，掌握的知识点不够多，在遇到一个问题时明明有非常简便的方式但由于不了解只能采用最笨和效力最低的方法，有时甚至根本没有思路。

在此也要对郭俊辉学长和陈炯老师对我的悉心指导以及大疆对培育青年学生的社会责任感表示真挚的感谢。

附件 1　Kernel_entry 宏

```
        .macro  kernel_entry, el, regsize = 64
        .if     \regsize == 32
        mov     w0, w0                                  // zero upper 32 bits of x0
        .endif
        stp     x0, x1, [sp, #16 * 0]
        stp     x2, x3, [sp, #16 * 1]
        stp     x4, x5, [sp, #16 * 2]
        stp     x6, x7, [sp, #16 * 3]
        stp     x8, x9, [sp, #16 * 4]
        stp     x10, x11, [sp, #16 * 5]
        stp     x12, x13, [sp, #16 * 6]
        stp     x14, x15, [sp, #16 * 7]
        stp     x16, x17, [sp, #16 * 8]
        stp     x18, x19, [sp, #16 * 9]
        stp     x20, x21, [sp, #16 * 10]
        stp     x22, x23, [sp, #16 * 11]
        stp     x24, x25, [sp, #16 * 12]
        stp     x26, x27, [sp, #16 * 13]
        stp     x28, x29, [sp, #16 * 14]

        .if     \el == 0
        clear_gp_regs
        mrs     x21, sp_el0
        ldr_this_cpu tsk, __entry_task, x20
        msr     sp_el0, tsk

        /*              * Ensure MDSCR_EL1.SS is clear, since we can unmask debug exceptions   * when
scheduling.   */
        ldr     x19, [tsk, #TSK_TI_FLAGS]
        disable_step_tsk x19, x20

        /* Check for asynchronous tag check faults in user space */
        check_mte_async_tcf x19, x22
        apply_ssbd 1, x22, x23

        ptrauth_keys_install_kernel tsk, x20, x22, x23

        scs_load tsk, x20
        .else
        add     x21, sp, #S_FRAME_SIZE
        get_current_task tsk
```

```
	/* Save the task's original addr_limit and set USER_DS */
	ldr	x20, [tsk, #TSK_TI_ADDR_LIMIT]
	str	x20, [sp, #S_ORIG_ADDR_LIMIT]
	mov	x20, #USER_DS
	str	x20, [tsk, #TSK_TI_ADDR_LIMIT]
	/* No need to reset PSTATE.UAO, hardware's already set it to 0 for us */
	.endif /* \el == 0 */
	mrs	x22, elr_el1
	mrs	x23, spsr_el1
	stp	lr, x21, [sp, #S_LR]

	/*	 * In order to be able to dump the contents of struct pt_regs at the	 * time the exception was taken (in case we attempt to walk the call	 * stack later), chain it together with the stack frames.	 */
	.if \el == 0
	stp	xzr, xzr, [sp, #S_STACKFRAME]
	.else
	stp	x29, x22, [sp, #S_STACKFRAME]
	.endif
	add	x29, sp, #S_STACKFRAME
#ifdef CONFIG_ARM64_SW_TTBR0_PANalternative_if_not ARM64_HAS_PAN
	bl	__swpan_entry_el\elalternative_else_nop_endif#endif

	stp	x22, x23, [sp, #S_PC]

	/* Not in a syscall by default (el0_svc overwrites for real syscall) */
	.if	\el == 0
	mov	w21, #NO_SYSCALL
	str	w21, [sp, #S_SYSCALLNO]
	.endif

	/* Save pmr */alternative_if ARM64_HAS_IRQ_PRIO_MASKING
	mrs_s	x20, SYS_ICC_PMR_EL1
	str	x20, [sp, #S_PMR_SAVE]alternative_else_nop_endif

	/* Re-enable tag checking (TCO set on exception entry) */#ifdef CONFIG_ARM64_MTEalternative_if ARM64_MTE
	SET_PSTATE_TCO(0)alternative_else_nop_endif#endif

	/*	 * Registers that may be useful after this macro is invoked:	 *	 * x20 - ICC_PMR_EL1	 * x21 - aborted SP	 * x22 - aborted PC	 * x23 - aborted PSTATE */
	.endm
```

附件 2 异常处理代码（注意先要函数声明，并将 fd 初始化）

```
// 用于读取 elf 的，解析 symtab 的变量
int fd = -1;      //读取本 elf 文件的文件描述符

int symtab_ind = -1, strtab_ind = -1;          //记录在段表中的下标
Elf64_Ehdr elfhd;                //拷贝 elf 文件头
Elf64_Shdr * section_header_ptr = NULL;    //指向 elf 段表
Elf64_Sym * symtab_ptr = NULL;         //用于存储解析函数明所需要的各节内容
char * strtab_ptr = NULL;
unsigned int sym_num = 0;

void sig_handler(int signo, siginfo_t * sig, void * context){
      pid_t cur_pid = getpid();
      ucontext_t * p = (ucontext_t *)context;       //获取异常现场指针
      long int fp = p->uc_mcontext.regs[29], lr = p->uc_mcontext.regs[30], \
       sp = p->uc_mcontext.sp, pc = p->uc_mcontext.pc, ret_lr, ret_sp;          //获取栈回溯所需的值
      long int addr = 0;                                  //存储栈回溯得到的函数地址



      //输出异常时各寄存器的值
      printf("异常发生时程序现场如下:\n");

      for(int i = 0; i < 30; i++){
            printf("寄存器 X%d 的值为：0x%llx\n", i, p->uc_mcontext.regs[i]);
      }

      printf("寄存器 LR 的值为：0x%llx\n", p->uc_mcontext.regs[30]);
      printf("寄存器 SP 的值为：0x%llx\n", p->uc_mcontext.sp);
      printf("寄存器 PC 的值为：0x%llx\n", p->uc_mcontext.pc);
      printf("寄存器 PSTATE 的值为：0x%llx\n", p->uc_mcontext.pstate);
      printf("error address：0x%llx\n", p->uc_mcontext.fault_address);

      get_elf_head();
      get_section_head();
      get_tab();

      int main_ind = 0;
      for(int j = 0; j < sym_num; j++){
            if(strcmp(&strtab_ptr[symtab_ptr[j].st_name], "main") == 0){
                  main_ind = j;
```

```
                break;
            }
        }
        long int base = (long int)main - symtab_ptr[main_ind].st_value;

        printf("*****************backTrace**********************\n");
        printf("调用次序\t 函数地址\t 函数名称\t 偏移\n");
        //栈回溯
        int count = 0;
        if(fp != sp){
            while(fp != 0){
                char func[SIZE];            //用于暂存函数名
                long int fp_e = fp,    lr_e = lr, offset;                    //备份，用于结果输出

                asm(                                    //根据函数栈帧的 lr，得到该函数的入口
地址
                    "mov x1, %2\n"
                    "sub x1, x1, #4\n"
                    "ldr w2, [x1]\n"                    //取得跳转入该函数的 bl 指令编码
                    "and w2, w2,#0x03ffffff\n"
                    "mov x3, #0\n"
                    "lsl w3, w2, #2\n"                  //由 bl 指令编码得到偏移地址
                    "add %0, x1, x3\n"                  //得到该函数地址
                    "ldr %2, [%1, #8]\n"                //得到父函数的栈帧的 fp 和 lr
                    "ldr %1, [%1]\n"
                    :"=r"(addr),"+r"(fp), "+r"(lr)
                    :
                    :"memory"
                );

                offset = pc - addr;
                pc = lr_e - 4;

                int i = 0;
                for(; i < sym_num; i++){
                    if(symtab_ptr[i].st_info & 0xf == 2 && symtab_ptr[i].st_value == addr){
                            strncpy(func, &strtab_ptr[symtab_ptr[i].st_name], SIZE - 1);
                            printf("%d\t%s(%lx): fp: %lx sp: %lx\n", count, func, offset, fp_e,
sp);

                            break;
                    }

                }
```

```
                if(i >= sym_num) printf("%d unknown\n", count);

                ++count;
                sp = fp_e;
            }
        }else{
            char func[SIZE];            //用于暂存函数名
            long int    offset;

            while(*(long int *)fp != 0){
                //输出该次函数调用的信息
                int i = 0;
                for(; i < sym_num; i++){
                    if((base + symtab_ptr[i].st_value) <= pc && pc <= (base + symtab_ptr[i].st_value + symtab_ptr[i].st_size)){
                        strncpy(func, &strtab_ptr[symtab_ptr[i].st_name], SIZE - 1);
                        offset = pc - symtab_ptr[i].st_value - base;

                        printf("%d:\t\t(0x%lx)\t%s\t(+0x%lx)\n",  count,(base + symtab_ptr[i].st_value), func, offset);
                        break;
                    }

                }
                if(i >= sym_num)
                    printf("%d unknown\n", count);

                asm(
                    "ldp %0, %1, [%0]"
                    :"+r"(fp), "=r"(lr)
                    :
                    :
                );

                printf("fp: %lx sp: %lx\n", fp, sp);

                pc = lr - 4;

                ++count;
                sp = fp;
            }
        }

        switch(signo){
```

```
            case 11:
                printf("进程[%d]在上述函数调用中出现了 SIGSEGV 错误，将退出进程\n",
cur_pid);
                exit(11);
                break;

            case 4:
                printf("进程[%d]在上述函数调用中出现了 SIGILL 错误，将退出进程\n", cur_pid);
                exit(4);
                break;

            case 7:
                printf("进程[%d]在上述函数调用中出现了 SIGBUS 错误，将退出进程\n",
cur_pid);
                exit(7);
                break;
        }

        exit(0);
}

void get_elf_head(){
        read(fd, elfhd.e_ident, EI_NIDENT);
        read(fd, &elfhd.e_type, sizeof(elfhd.e_type));
        read(fd, &elfhd.e_machine, sizeof(elfhd.e_machine));
        read(fd, &elfhd.e_version, sizeof(elfhd.e_version));
        read(fd, &elfhd.e_entry, sizeof(elfhd.e_entry));
        read(fd, &elfhd.e_phoff, sizeof(elfhd.e_phoff));
        read(fd, &elfhd.e_shoff, sizeof(elfhd.e_shoff));
        read(fd, &elfhd.e_flags, sizeof(elfhd.e_flags));
        read(fd, &elfhd.e_ehsize, sizeof(elfhd.e_ehsize));
        read(fd, &elfhd.e_phentsize, sizeof(elfhd.e_phentsize));
        read(fd, &elfhd.e_phnum, sizeof(elfhd.e_phnum));
        read(fd, &elfhd.e_shentsize, sizeof(elfhd.e_shentsize));
        read(fd, &elfhd.e_shnum, sizeof(elfhd.e_shnum));
        read(fd, &elfhd.e_shstrndx, sizeof(elfhd.e_shstrndx));
}

void get_section_head(){
        section_header_ptr = (Elf64_Shdr *)calloc(elfhd.e_shnum, sizeof(Elf64_Shdr));

        lseek(fd, elfhd.e_shoff, SEEK_SET);

        read (fd, section_header_ptr, elfhd.e_shnum * sizeof(Elf64_Shdr));
```

```
}

void get_tab(){
	int shstr_ind = elfhd.e_shstrndx;
	char shstr_ptr[section_header_ptr[shstr_ind].sh_size];

	lseek(fd, section_header_ptr[shstr_ind].sh_offset, SEEK_SET);

	read(fd, shstr_ptr, section_header_ptr[shstr_ind].sh_size);

	for(int i = 0; i < elfhd.e_shnum; i++){
		if(strcmp(&shstr_ptr[section_header_ptr[i].sh_name], ".symtab") == 0)
			symtab_ind = i;

		if(strcmp(&shstr_ptr[section_header_ptr[i].sh_name], ".strtab") == 0)
			strtab_ind = i;
	}

	sym_num = section_header_ptr[symtab_ind].sh_size / sizeof(Elf64_Sym);

//获取 symtab
	symtab_ptr = calloc(sym_num, sizeof(Elf64_Sym));
	lseek(fd, section_header_ptr[symtab_ind].sh_offset, SEEK_SET);
	read(fd, symtab_ptr, section_header_ptr[symtab_ind].sh_size);

//获取 strtab
	strtab_ptr = (char *)malloc(section_header_ptr[strtab_ind].sh_size);
	lseek(fd, section_header_ptr[strtab_ind].sh_offset, SEEK_SET);
	read(fd, strtab_ptr, section_header_ptr[strtab_ind].sh_size);

}
```

附件 3

Elf 文件读取原理文章：
https://bbs.pediy.com/thread-223569.htm
https://bbs.pediy.com/thread-255670.htm