2021219113 2021213595 沈尉林

part 1：函数说明

1. `DEFINE_IDTENTRY(exc_divide_error)`：

```c
DEFINE_IDTENTRY(exc_divide_error)

{

    do_error_trap(regs, 0, "divide error", X86_TRAP_DE,
SIGFPE,

                  FPE_INTDIV, error_get_trap_addr(regs));

}
```

- 这个宏定义了中断描述符表（IDT）的入口，特定的异常（在这里是除零异常）将引发一个名为 `exc_divide_error` 的处理函数。
- 接下来将调用 `do_error_trap` 函数来处理除零异常。

2. `do_error_trap(struct pt_regs *regs, long error_code, char *str, unsigned long trapnr, int signr, int sicode, void __user *addr)`：

```c
static void do_error_trap(struct pt_regs *regs, long
error_code, char *str,

    unsigned long trapnr, int signr, int sicode, void __user
*addr)

{

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



```

```
    if (notify_die(DIE_TRAP, str, regs, error_code, trapnr,
signr) !=

            NOTIFY_STOP) {

        cond_local_irq_enable(regs);

        do_trap(trapnr, signr, str, regs, error_code, sicode,
addr);

        cond_local_irq_disable(regs);

    }

}
```

- 这个函数用于处理各种异常，在这里处理除零异常。
- 然后，它调用 `notify_die` 来通知有关异常的处理。如果 `notify_die` 返回 `NOTIFY_STOP`，则异常处理被停止，否则它会继续处理异常。
- 如果异常继续处理，它会启用中断，然后调用 `do_trap` 函数，用于实际处理异常。

3. `do_trap(int trapnr, int signr, char *str, struct pt_regs *regs, long error_code, int sicode, void __user *addr)`:

```c
static void

do_trap(int trapnr, int signr, char *str, struct pt_regs
*regs,

    long error_code, int sicode, void __user *addr)

{

    struct task_struct *tsk = current;



    if (!do_trap_no_signal(tsk, trapnr, str, regs,
error_code))

```

```
17
18          return;
19
20
21
22      show_signal(tsk, signr, "trap ", str, regs, error_code);
23
24
25
26      if (!sicode)
27
28          force_sig(signr);
29
30      else
31
            force_sig_fault(signr, sicode, addr);

    }
```

- 这个函数用于处理异常。
- 它首先检查是否需要发送信号，如果不需要发送信号，则直接返回。
- 如果需要发送信号，它会调用 `show_signal` 函数来显示有关异常的信息，然后根据 `sicode` 的值来调用 `force_sig` 或 `force_sig_fault` 函数来发送信号。

4. `do_trap_no_signal(struct task_struct *tsk, int trapnr, const char *str, struct pt_regs *regs, long error_code)`:

```c
static nokprobe_inline int

do_trap_no_signal(struct task_struct *tsk, int trapnr, const
char *str,

          struct pt_regs *regs, long error_code)

{

    if (v8086_mode(regs)) {

        /*

```

```
14
15          * Traps 0, 1, 3, 4, and 5 should be forwarded to
16  vm86.
17
18          * On nmi (interrupt 2), do_trap should not be called.
19
20          */
21
22         if (trapnr < X86_TRAP_UD) {
23
24             if (!handle_vm86_trap((struct kernel_vm86_regs *)
25  regs,
26
27                                   error_code, trapnr))
28
29                 return 0;
30
31         }
32
33     } else if (!user_mode(regs)) {
34
35         if (fixup_exception(regs, trapnr, error_code, 0))
36
37             return 0;
38
39
40
41         tsk->thread.error_code = error_code;
42
43         tsk->thread.trap_nr = trapnr;
44
45         die(str, regs, error_code);
46
47     }
48
49
50
51     /*
52
53      * We want error_code and trap_nr set for userspace faults
54
```

```
55  and
56
57       * kernelspace faults which result in die(), but not
58
59       * kernelspace faults which are fixed up.  die() gives the
60
61       * process no chance to handle the signal and notice the
62
63       * kernel fault information, so that won't result in
64  polluting
65
66       * the information about previously queued, but not yet
67
68       * delivered, faults.  See also exc_general_protection
69  below.
70
71       */
72
73      tsk->thread.error_code = error_code;

        tsk->thread.trap_nr = trapnr;



        return -1;

    }
```

- 这个函数用于检查是否需要发送信号，如果不需要，则返回0，否则返回-1。
- 首先，它检查是否在虚拟8086模式下（v8086_mode）。如果是的话，它会根据异常类型和错误码来决定是否将异常交给虚拟8086模式处理。
- 否则，如果不是用户态，即内核态（kernel_mode），它会尝试修复异常，如果修复成功，则返回0。
- 如果无法修复，它会设置进程的错误码和陷阱号，并调用 `die` 函数，通常会导致进程终止。

5. `int fixup_exception(struct pt_regs *regs, int trapnr, unsigned long error_code, unsigned long fault_addr)`:

```c
int fixup_exception(struct pt_regs *regs, int trapnr, unsigned
long error_code,

             unsigned long fault_addr)

{

    const struct exception_table_entry *e;

    ex_handler_t handler;



    e = search_exception_tables(regs->ip);

    if (!e)

        return 0;



    handler = ex_fixup_handler(e);

    return handler(e, regs, trapnr, error_code, fault_addr);

}
```

- 这个函数用于尝试修复异常，如果能够修复异常，则返回1，否则返回0。
- 它首先通过 `search_exception_tables` 函数查找异常表，以确定是否存在与 `regs->ip`（指令指针）匹配的异常处理程序。
- 如果找到匹配项，它会获取异常处理程序（`ex_fixup_handler`），并调用该处理程序来尝试修复异常。

part 2 操作系统如何处理异常

以除零异常为例，我们可以通过上述代码中的函数调用链来解释异常处理过程：

1. 当 CPU 执行一条除以零的指令时，会触发除零异常（divide error）。
2. 异常处理从硬件一直传递到操作系统内核。

3. 内核中的中断描述符表（IDT）会指向 `exc_divide_error` 函数作为异常处理程序的入口点。
4. `exc_divide_error` 函数会被调用，它接收一些参数，如寄存器状态 `regs`，错误码 `error_code`，异常名称字符串 `str`，异常号 `trapnr`，信号编号 `signr`，以及用户空间地址 `addr`。
5. `do_error_trap` 函数被 `exc_divide_error` 调用，这个函数用于处理各种异常，包括除零异常。
6. 在 `do_error_trap` 函数中，首先检查了一些与异常处理相关性不高的条件，比如 RCU 相关的检查，然后调用 `notify_die` 函数来通知内核有异常发生。这一步可以用于外部观察或处理异常。
7. 如果 `notify_die` 返回值不是 `NOTIFY_STOP`，说明异常处理未被停止，继续执行下一步。
8. `do_error_trap` 启用了中断（`cond_local_irq_enable(regs)`），然后调用 `do_trap` 函数，传递异常号 `trapnr`，信号编号 `signr`，异常名称字符串 `str`，寄存器状态 `regs`，错误码 `error_code`，异常编码 `sicode` 和用户空间地址 `addr` 给 `do_trap` 函数。
9. `do_trap` 函数用于实际处理异常。它首先检查是否需要发送信号，如果需要发送信号，它会调用 `show_signal` 函数来显示异常信息，然后根据异常编码 `sicode` 的值来调用 `force_sig` 或 `force_sig_fault` 函数来发送信号给当前进程或线程。
10. 最终，异常处理过程完成，控制权返回到用户空间或继续内核执行，具体取决于异常的性质和处理方式。

part 3 操作系统在内核态和用户态处理异常的区别

在内核态下处理除零异常：

1. 当在内核态下发生除零异常时，操作系统会首先捕获这个异常。
2. 异常处理会根据异常类型和错误码来确定如何处理。在这里，我们关注的是除零异常（X86_TRAP_DE）。
3. 如果内核发现可以修复除零异常，它会尝试执行修复操作。这通常包括检查除数是否为零，如果除数为零，可能会将其设置为一个非零值，以避免除零错误。这是一个非常危险的操作，因为它可能导致不可预测的结果。在一般情况下，内核不会尝试修复除零异常，而是将异常传递给正在执行的进程。
4. 如果内核决定不修复异常，它会将异常信息传递给当前正在执行的进程。这可能涉及向进程发送信号，通知进程发生了除零异常。

在用户态下处理除零异常：

1. 当在用户态下发生除零异常时，操作系统会捕获异常并将控制权切换到内核态。

2. 异常处理会根据异常类型和错误码来确定如何处理。在这里，我们关注的是除零异常（X86_TRAP_DE）。
3. 内核会检查当前发生异常的进程是否有合适的异常处理程序（信号处理程序）来处理除零异常。如果有，内核将执行该处理程序，并传递异常信息给它。
4. 如果进程没有注册对除零异常的处理程序，或者处理程序返回后，内核会根据默认行为处理异常。通常，这会导致终止进程，因为除零异常通常表示了一个严重的错误。