

上海電力大學

SHANGHAI UNIVERSITY OF ELECTRIC POWER



FAULT TOLERANCE

題目: *Volt OS 故障注入实验报告书*
(*ODC Method*)

Under the on-campus supervision of

Dr. Man Xu, Dr. Zhen Wang

Under the off-campus supervision of

Dr. Wei Zhang

Tested by Lei Tang, Software Engineering

头

在软件工程领域，软件的可靠性和稳定性是衡量其质量的关键指标。正如著名软件工程专家 Watts S. Humphrey 所言：“软件的质量是构建出来的，而不是测试出来的，但严格的测试却是确保高质量软件必不可少的环节。”为了确保软件在实际运行中能够承受各种异常情况，故障注入测试成为了一项至关重要的技术手段。

软件故障注入已测试系统可靠性的技术，一般分为基于代码变异的缺陷注入和缺陷表现（错误）注入。诚如软件工程权威学者 Barry Boehm 所说：“对软件的深入理解和把控需要从代码的根源以及实际运行表现双管齐下。”基于代码变异的缺陷注入正是通过对软件代码进行特定的变异操作，模拟可能出现的代码缺陷，从根源上探寻软件潜在问题；而缺陷表现（错误）注入则是直接在软件运行过程中注入错误表现，以观察软件的应对能力，在实际运行层面来检验软件的健壮性。

在实际的软件开发过程中，软件编写人员总是不可避免地会写出与预期不符、不理想的代码。软件工程大师 Fred Brooks 曾强调：“人月神话告诉我们，软件开发中的复杂性和不可预测性无处不在，即使最优秀的程序员也难以完全避免失误。”这些潜在的问题代码可能在软件投入使用后引发各种故障，影响软件的正常运行。因此，为了确保代码质量，我们不得不对一个编码完成的系统进行故障注入测试。通过这种方式，可以提前发现软件中可能存在的问题，并及时进行修复，从而提高软件的可靠性和稳定性，为用户提供更加可靠的软件产品。

一、删除初始化 (OMVIV) 变异

变异前:

```
static int _top = 0;
static int _current = -1;
```

变异后:

```
static int _top;
static int _current;
```

变异前后的运行结果的打印报告如下图 1 所示:

图 1 OMVIV 代码变异后的运行对比

```
1 -----
2 Hello, VoltOS!
3 developed by Lei Tang, SUEP.
4 Mutated Mode:
5 -----welcome to Volt OS-----
6 page allocator initializing...
7 HEAP_START = 0x8000833c(aligned to 0x80009000),
8 HEAP_SIZE = 0x07ff7cc4,
9 num of reserved pages = 8, num of pages to be allocated :
10 TEXT: 0x80000000 -> 0x80004258
11 RODATA: 0x80004258 -> 0x8000481f
12 DATA: 0x80005000 -> 0x80005028
13 BSS: 0x80005030 -> 0x8000833c
14 HEAP: 0x80011000 -> 0x88000000
15 page initialized.
16 [VoltCommand]$ help
17 Unknown command. Type 'help' for available commands.
18
19 [VoltCommand]$ inj
20 Unknown command. Type 'help' for available commands.
21
22 [VoltCommand]$ gotimer interruption!
23 tick: 1
24 Task 0: Created!
25 Sync exceptions! Code = 8
26 System call from U-mode!
27 --> sys_gettid, arg0 = 0x80005898
28 system call returned!, hart id is 0
29 Task 0: Running...

1 -----
2 Hello, VoltOS!
3 developed by Lei Tang, SUEP.
4 Safty Mode:
5 -----welcome to Volt OS-----
6 page allocator initializing...
7 HEAP_START = 0x80008338(aligned to 0x80009000),
8 HEAP_SIZE = 0x07ff7cc8,
9 num of reserved pages = 8, num of pages to be allocated :
10 TEXT: 0x80000000 -> 0x80004258
11 RODATA: 0x80004258 -> 0x8000481f
12 DATA: 0x80005000 -> 0x8000502c
13 BSS: 0x80005030 -> 0x80008338
14 HEAP: 0x80011000 -> 0x88000000
15 page initialized.
16 [VoltCommand]$ gotimer interruption!
17 tick: 1
18 Task 1: Created!
19 Task 1: Running...
20 Task 1: Running...
21 timer interruption!
22 tick: 2
23 Task 0: Created!
24 Sync exceptions! Code = 8
25 System call from U-mode!
26 --> sys_gettid, arg0 = 0x80005898
27 system call returned!, hart id is 0
28 Task 0: Running...
29 Task 0: Running...
```

可以看到, 对于本测试 (OMVIV), 删除掉初始化的赋值语句, 我们发现 HEAP 的大小减小了 4Byte。

对于减少的 4Bytes 有可能会如下几个问题: 如果对于频繁申请内存的程序会导致内存溢出问题。

而对于 HEAP 减小后的内存空间, 在整体上会显得紧凑。越是紧凑, 发生哈希碰撞从而对于原始数据进行破坏性的数据覆盖等问题。

二、删除常数赋值语句

变异前：

```
#define MAX_TASKS 10
#define STACK_SIZE 1024
```

变异后：

```
// #define MAX_TASKS 10
#define STACK_SIZE 1024
```

对于上述的删除常数赋值语句的变异，我们最后把 c 源文件代码提交编译器时，出现了编译错误。编译错误报告如下：

```
sched.c:12:49: error: 'MAX_TASKS' undeclared here (not in a function)
    12 | uint8_t __attribute__((aligned(16))) task_stack[MAX_TASKS][STACK_SIZE];
      |                                                     ~~~~~
sched.c: In function 'task_create':
sched.c:63:1: warning: control reaches end of non-void function [-Wreturn-type]
    63 | }
      | ^
```

对于该类型代码变异，我们可以发现其不能正常通过编译。对于常数赋值，有两种状态：一是在程序中运用到了这个常数，那么如果对于即将用到的常数如果删除后，编译会直接不通过。

另一个状态就是就是程序中并没有用到这个常数，只是声明了此变量。一般地，如 GCC 编译后，会抛出警告：“该变量在程序中没有被用到过”以提示编程人员可能存在把这个变量遗漏的现象。在编程人员检查了该错误后，会有两种动作：一是可能把遗漏掉的使用补回去，二是直接删除该声明语句。

做完上述两个动作后，可以发现待编译程序的状态又回到了原先的状态中。所以这个状态是安全的。因为该程序根本通不过编译，自然不会生成可执行码，便不会在 HART 上执行。

三、删除表达式赋值

变异前：

```
_current = (_current + 1) % _top;
struct context *next = &(ctx_tasks[_current]);
```

变异后：

```
// _current = (_current + 1) % _top;
struct context *next = &(ctx_tasks[_current]);
```

于是我们发现系统崩溃了。

系统运行的报告如下图 2 所示：

```

1 -----
2 Hello, VoltOS!
3 developed by Lei Tang, SUEP.
4 Mutated Mode:
5 -----welcome to Volt OS-----
6 page allocator initializing...
7 HEAP_START = 0x80008338(aligned to 0x80009000),
8 HEAP_SIZE = 0x07ff7cc8,
9 num of reserved pages = 8,
10 num of pages to be allocated for heap = 32751
11 TEXT: 0x80000000 -> 0x80004238
12 RODATA: 0x80004238 -> 0x800047ff
13 DATA: 0x80005000 -> 0x8000502c
14 BSS: 0x80005030 -> 0x80008338
15 HEAP: 0x80011000 -> 0x88000000
16 page initialized.
17 [VoltCommand]$ go
18 Sync exceptions! Code = 7
19 panic: OOPS! What can I do!
20
1 -----
2 Hello, VoltOS!
3 developed by Lei Tang, SUEP.
4 Safty Mode:
5 -----welcome to Volt OS-----
6 page allocator initializing...
7 HEAP_START = 0x80008338(aligned to 0x80009000),
8 HEAP_SIZE = 0x07ff7cc8,
9 num of reserved pages = 8,
10 num of pages to be allocated for heap = 32751
11 TEXT: 0x80000000 -> 0x80004258
12 RODATA: 0x80004258 -> 0x8000481f
13 DATA: 0x80005000 -> 0x8000502c
14 BSS: 0x80005030 -> 0x80008338
15 HEAP: 0x80011000 -> 0x88000000
16 page initialized.
17 [VoltCommand]$ gotimer interruption!
18 tick: 1
19 Task 1: Created!
20 Task 1: Running...

```

图 2 OMVAE 代码变异体运行对比

我们发现通过对表达式 `_current = (_current + 1) % _top;` 的删除，整体的系统是出于崩溃状态的。因为原先的设计是：键入“go”命令后，操作系统应该开始一个 task0 任务的调度。但是这里却陷入了崩溃，打印输出了“OOPS! WHAT CAN I DO!”。

四、 删除 if 语句（保留内部结构）

变异前：

```

if (_top <= 0) {
    panic("Num of task should be greater than zero!");
    return;
}

```

变异后：

```

// if (_top <= 0) {
    panic("Num of task should be greater than zero!");
    return;
// }

```

通过对 if 结构的变异，我们得到了如下图 3 的运行报告：

```

1 -----
2 Hello, VoltOS!
3 developed by Lei Tang, SUEP.
4 Mutated Mode:
5 -----welcome to Volt OS-----
6 page allocator initializing...
7 HEAP_START = 0x80008338(aligned to 0x80009000),
8 HEAP_SIZE = 0x07ff7cc8,
9 num of reserved pages = 8,
10 num of pages to be allocated for heap = 32751
11 TEXT: 0x80000000 -> 0x80004208
12 RODATA: 0x80004208 -> 0x800047cf
13 DATA: 0x80005000 -> 0x8000502c
14 BSS: 0x80005030 -> 0x80008338
15 HEAP: 0x80011000 -> 0x88000000
16 page initialized.
17 [VoltCommand]$ gopanic:
18 Num of task should be greater than zero!
19
1 -----
2 Hello, VoltOS!
3 developed by Lei Tang, SUEP.
4 Safty Mode:
5 -----welcome to Volt OS-----
6 page allocator initializing...
7 HEAP_START = 0x80008338(aligned to 0x80009000),
8 HEAP_SIZE = 0x07ff7cc8,
9 num of reserved pages = 8,
10 num of pages to be allocated for heap = 32751
11 TEXT: 0x80000000 -> 0x80004258
12 RODATA: 0x80004258 -> 0x8000481f
13 DATA: 0x80005000 -> 0x8000502c
14 BSS: 0x80005030 -> 0x80008338
15 HEAP: 0x80011000 -> 0x88000000
16 page initialized.
17 [VoltCommand]$ gotimer interruption!
18 tick: 1
19 Task 1: Created!

```

图 3 OMIA 变异后的运行对比

可以发现，程序崩溃了。其背后的崩溃逻辑是调度器会首先检查任务队列里是否存在任务。如果存在任务，才可以调度进行。而这里，则是直接跳过了检查，直接执行该段逻辑，所以崩溃了。

五、删除 if 结构（包括其内部语句）

变异前：

```
void schedule()
{
    if (_top <= 0) {
        panic("Num of task should be greater than zero!");
        return;
    }

    _current = (_current + 1) % _top;
    struct context *next = &(ctx_tasks[_current]);
    switch_to(next);
}
```

变异后：

```
void schedule()
{
    // if (_top <= 0) {
    //     panic("Num of task should be greater than zero!");
    //     return;
    // }

    _current = (_current + 1) % _top;
    struct context *next = &(ctx_tasks[_current]);
    switch_to(next);
}
```

由于选择的程序部位相对特殊，在本次变异后显性条件下似乎是很难发现缺陷的。如下图 4 所示。

```
1 -----
2 Hello, VoltOS!
3 developed by Lei Tang, SUEP.
4 Mutated Mode:
5 -----welcome to Volt OS-----
6 page allocator initializing...
7 HEAP START = 0x80008338 (aligned to 0x80009000),
8 HEAP_SIZE = 0x07ff7cc8,
9 num of reserved pages = 8,
10 num of pages to be allocated for heap = 32751
11 TEXT: 0x80000000 -> 0x80004240
12 RODATA: 0x80004240 -> 0x800047db
13 DATA: 0x80005000 -> 0x8000502c
14 BSS: 0x80005030 -> 0x80008338
15 HEAP: 0x80011000 -> 0x88000000
16 page initialized.
17 [VoltCommand]$ gotimer interruption!
18 tick: 1
19 Task 1: Created!
20 Task 1: Running...
21 Task 1: Running...
22 timer interruption!
23 tick: 2
24 Task 0: Created!
25 Sync exceptions! Code = 8
26 System call from U-mode!
27 --> sys_gethid, arg0 = 0x80005898
28 system call returned!, hart id is 0
29 Task 0: Running...
```

图 4 OMIFS 代码变异后的运行对比

对于任务队列里有实时计算任务的系统而言，看上去是正常工作的。但是，当我使用组合故障手段的时候（结合删除函数调用变异，即“OMFC”），便发生了严重的、灾难性的系统

崩溃。如下图 5 和图 6 所示：

```
void os_main(void)
{
    task_create(user_task0);
    task_create(user_task1);
}
```




```
void os_main(void)
{
    // task_create(user_task0);
    // task_create(user_task1);
}
```

图 5 OMFC 代码变异前后

可以看到，os_main(void)是一个任务调度函数。函数体里面的任务遵循 FIFO 原则。对于 OMFC 变异后，该模块是十分敏感的，直接导致了 kernel 调度器的崩溃。崩溃如下图 6 所示：

```
1 -----
2 Hello, VoltOS!
3 developed by Lei Tang, SUEP.
4 Mutated Mode:
5 -----welcome to Volt OS-----
6 page allocator initializing...
7 HEAP_START = 0x80008338(aligned to 0x80009000),
8 HEAP_SIZE = 0x07ff7cc8,
9 num of reserved pages = 8,
10 num of pages to be allocated for heap = 32751
11 TEXT: 0x80000000 -> 0x80004220
12 RODATA: 0x80004220 -> 0x800047bb
13 DATA: 0x80005000 -> 0x8000502c
14 BSS: 0x80005030 -> 0x80008338
15 HEAP: 0x80011000 -> 0x88000000
16 page initialized.
17 [VoltCommand]$ goSync exceptions! Code = 7
18 panic: OOPS! What can I do!
19
20
21
22
```



```
1 -----
2 Hello, VoltOS!
3 developed by Lei Tang, SUEP.
4 Safty Mode
5 -----welcome to Volt OS-----
6 page allocator initializing...
7 HEAP_START = 0x80008338(aligned to 0x80009000),
8 HEAP_SIZE = 0x07ff7cc8,
9 num of reserved pages = 8,
10 num of pages to be allocated for heap = 32751
11 TEXT: 0x80000000 -> 0x80004258
12 RODATA: 0x80004258 -> 0x8000481f
13 DATA: 0x80005000 -> 0x8000502c
14 BSS: 0x80005030 -> 0x80008338
15 HEAP: 0x80011000 -> 0x88000000
16 page initialized.
17 [VoltCommand]$ gotimer interruption!
18 tick: 1
19 Task 1: Created!
20 Task 1: Running...
21 Task 1: Running...
22 timer interruption!
```

图 6 OMIFS 结合 OMFC 两种变异手段后崩溃对比

六、删除 if 结构（包括内部语句，但是保留 else 部分）

变异前：

```
if (_top < MAX_TASKS) {
    ctx_tasks[_top].sp = (reg_t) &task_stack[_top][STACK_SIZE];
    ctx_tasks[_top].pc = (reg_t) start_routine;
    _top++;
    return 0;
} else {
    return -1;
}
```

变异后：

```
int task_create(void (*start_routine)(void))
{
    // if (_top < MAX_TASKS) {
    //     ctx_tasks[_top].sp = (reg_t) &task_stack[_top][STACK_SIZE];
    //     ctx_tasks[_top].pc = (reg_t) start_routine;
    //     _top++;
    //     return 0;
    // } else {
        return -1;
    // }
}
```

可以看到，我们对于 task_create() 模块的函数体注释掉了逻辑判断部分。对于该函数而言，注释掉了主干部分后，就只剩下了返回 -1 的值。

通过实验，我们发现系统也崩溃了。其崩溃的系统运行报告如下图 7 所示：

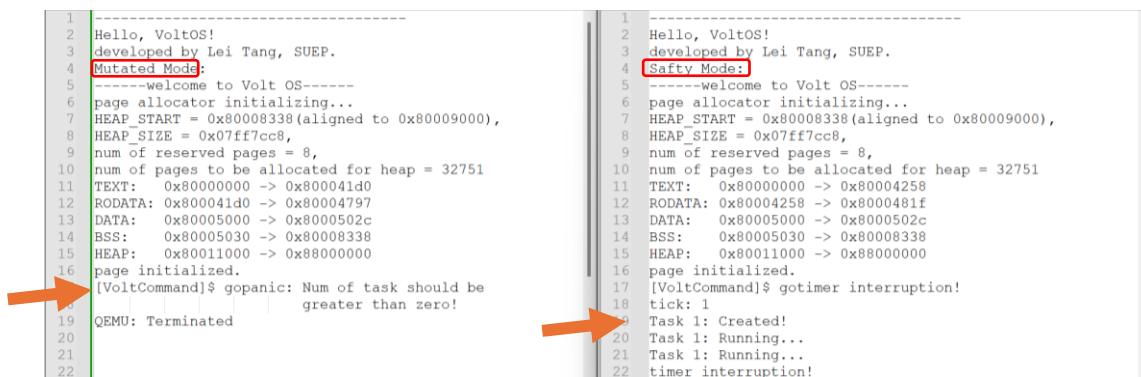


图 7 OMIEB 代码变异后的运行报告对比

七、删除“逻辑与”或“逻辑或”以及其中一个操作数

变异前:

```
void sched_init()
{
    w_mscratch(0);

    /* enable machine-mode software interrupts. */
    w_mie(r_mie() | MIE_MSIE);
}
```

变异后:

```
void sched_init()
{
    w_mscratch(0);

    /* enable machine-mode software interrupts. */
    // w_mie(r_mie() | MIE_MSIE);
}
```

可以看到, Machine 模式下的软件中断使能信号被删除了。但是庆幸的是, 对于下面的测试实验中, 软件中断使能的变异并没有导致系统崩溃。系统测试报告如下图 8:

<pre>1 ----- 2 Hello, VoltOS! 3 developed by Lei Tang, SUEP. 4 Mutated Mode: 5 -----welcome to Volt OS----- 6 page allocator initializing... 7 HEAP_START = 0x80008338(aligned to 0x80009000), 8 HEAP_SIZE = 0x07ff7cc8, 9 num of reserved pages = 8, 10 num of pages to be allocated for heap = 32751 11 TEXT: 0x80000000 -> 0x800041f4 12 RODATA: 0x800041f4 -> 0x800047bb 13 DATA: 0x80005000 -> 0x8000502c 14 BSS: 0x80005030 -> 0x80008338 15 HEAP: 0x80011000 -> 0x88000000 16 page initialized. 17 [VoltCommand]\$ gotimer interruption! 18 tick: 1 19 Task 1: Created! 20 Task 1: Running... 21 Task 1: Running... 22 timer interruption! 23 tick: 2 24 Task 0: Created! 25 Sync exceptions! Code = 8 26 System call from U-mode! 27 --> sys_gethid, arg0 = 0x80005898 28 system call returned!, hart id is 0 29 Task 0: Running...</pre>	<pre>1 ----- 2 Hello, VoltOS! 3 developed by Lei Tang, SUEP. 4 Safty Mode: 5 -----welcome to Volt OS----- 6 page allocator initializing... 7 HEAP_START = 0x80008338(aligned to 0x80009000), 8 HEAP_SIZE = 0x07ff7cc8, 9 num of reserved pages = 8, 10 num of pages to be allocated for heap = 32751 11 TEXT: 0x80000000 -> 0x80004258 12 RODATA: 0x80004258 -> 0x8000481f 13 DATA: 0x80005000 -> 0x8000502c 14 BSS: 0x80005030 -> 0x80008338 15 HEAP: 0x80011000 -> 0x88000000 16 page initialized. 17 [VoltCommand]\$ gotimer interruption! 18 tick: 1 19 Task 1: Created! 20 Task 1: Running... 21 Task 1: Running... 22 timer interruption! 23 tick: 2 24 Task 0: Created! 25 Sync exceptions! Code = 8 26 System call from U-mode! 27 --> sys_gethid, arg0 = 0x80005898 28 system call returned!, hart id is 0 29 Task 0: Running...</pre>
---	---

图 8 OMLC 代码变异系统运行报告对比

分析其原因: 在 RISC-V 架构中, MIE 寄存器控制着机器模式下的中断使能。MIE_MSIE 位是 MIE 寄存器中的一个位, 用于启用机器模式的软件中断。而对于实时型 OS 我们对于分配给系统令其执行的任务是放在 USER 态的。所以对于整体的功能性而言, 其输出的内容是我们所期待的。

八、删除连续语句

变异前:

```
void task_delay(volatile int count)
{
    count *= 50000;
    while (count--);
}
```

变异后:

```
void task_delay(volatile int count)
{
    // count *= 50000;
    // while (count--);
}
```

同样地，由于其位置的特殊性，对于本系统基本没有破坏性，仅仅使调度器的 task_delay() 失效。对于 delay 模块的失效不怕破坏其功能性。

其运行结果与未故障注入的系统版本是相同的。但是，我们同样发现了一些弊端：系统对于任务的高并发处理可能会导致我们的系统不能原子性地完成特殊任务。如果要实现原子性，可以通过自定义 RISC v 的指令扩展以实现。如下图 9:

```
1 Task 1: Running...
2 Task 1: Running...
3 Task 1: Running...
4 Task 1: Running...
5 Task 1: Running...
6 Task 1: Running...
7 Task timer interruption!
8 tick: 16
9 ...
10 Task 0: Running...
11 Task 0: Running...
12 Task 0: Running...
13 Task 0: Running...
14 Task 0: Running...
15 Task 0: Running...
16 Task 0: Running...
17
```

图 9 OMLPA 代码变异后的运行结果

九、用 0xFF 对变量进行 XOR

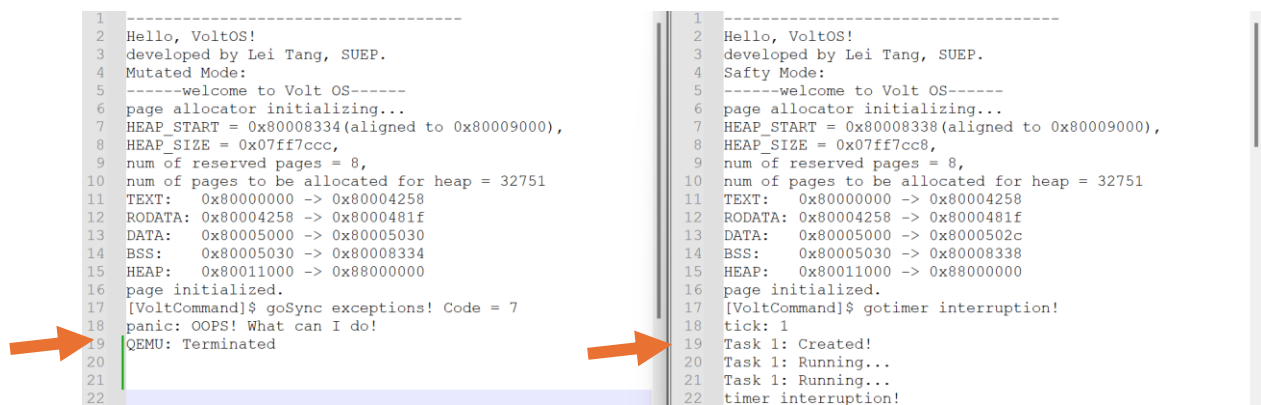
变异前：

```
static int _top = 0;
static int _current = -1;
```

变异后：

```
static int _top = 0 ^ 0xFF;
static int _current = -1;
```

接下来，我们运行程序。其运行结果如下图 10 所示：



```
1 -----
2 Hello, VoltOS!
3 developed by Lei Tang, SUEP.
4 Mutated Mode:
5 -----welcome to Volt OS-----
6 page allocator initializing...
7 HEAP_START = 0x80008334(aligned to 0x80009000),
8 HEAP_SIZE = 0x07ff7ccc,
9 num of reserved pages = 8,
10 num of pages to be allocated for heap = 32751
11 TEXT: 0x80000000 -> 0x80004258
12 RODATA: 0x80004258 -> 0x8000481f
13 DATA: 0x80005000 -> 0x80005030
14 BSS: 0x80005030 -> 0x80008334
15 HEAP: 0x80011000 -> 0x88000000
16 page initialized.
17 [VoltCommand]$ goSync exceptions! Code = 7
18 panic: OOPS! What can I do!
19 QEMU: Terminated
20
21
22
```

```
1 -----
2 Hello, VoltOS!
3 developed by Lei Tang, SUEP.
4 Safty Mode:
5 -----welcome to Volt OS-----
6 page allocator initializing...
7 HEAP_START = 0x80008338(aligned to 0x80009000),
8 HEAP_SIZE = 0x07ff7cc8,
9 num of reserved pages = 8,
10 num of pages to be allocated for heap = 32751
11 TEXT: 0x80000000 -> 0x80004258
12 RODATA: 0x80004258 -> 0x8000481f
13 DATA: 0x80005000 -> 0x8000502c
14 BSS: 0x80005030 -> 0x80008338
15 HEAP: 0x80011000 -> 0x88000000
16 page initialized.
17 [VoltCommand]$ gotimer interruption!
18 tick: 1
19 Task 1: Created!
20 Task 1: Running...
21 Task 1: Running...
22 timer interruption!
```

图 10 OWWAV 代码变异后运行结果报告

可以看到，对于一个用于标记任务上下文且被初始化为 0 的 top 的操作进行异或操作是致命的，Kernel 再一次崩溃了。

这两个变量在任务调度和上下文切换中起着关键作用。当一个任务被调度执行时，_current 指向该任务的上下文，而当任务完成或被阻塞时，内核会更新 _current 以指向下一个要执行的任务。同时，_top 确保了任务上下文的存储区域不会超出系统的限制。

如下示意图 11：

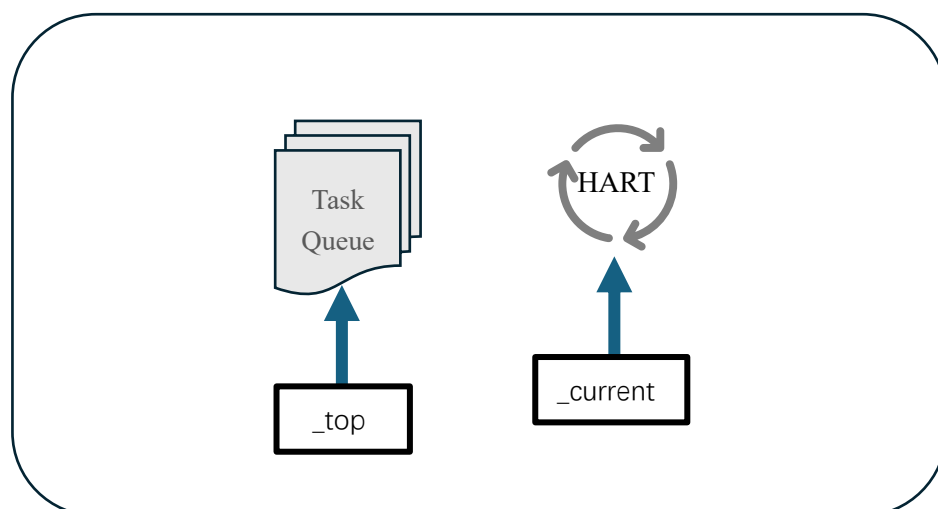


图 11 top 和 current 的工作示意图

十、用 0XFF 对函数调用参数中的中的变量进行 XOR

变异前:

```
void task_delay(volatile int count)
{
    count *= 50000;
    while (count--);
}
```

变异后:

```
void task_delay(volatile int count)
{
    count ^= 0xff;
    count *= 50000;
    while (count--);
}
```

对于这个函数模块而言，对于其参数的污染对于整体系统的鲁棒性基本没有破坏。其运行报告对比图如下图 12 所示:

<pre>1 ----- 2 Hello, VoltOS! 3 developed by Lei Tang, SUEP. 4 Mutated Mode: 5 -----welcome to Volt OS----- 6 page allocator initializing... 7 HEAP_START = 0x80008338(aligned to 0x80009000), 8 HEAP_SIZE = 0x07ff7cc8, 9 num of reserved pages = 8, 10 num of pages to be allocated for heap = 32751 11 TEXT: 0x80000000 -> 0x80004264 12 RODATA: 0x80004264 -> 0x8000482b 13 DATA: 0x80005000 -> 0x8000502c 14 BSS: 0x80005030 -> 0x80008338 15 HEAP: 0x80011000 -> 0x88000000 16 page initialized. 17 [VoltCommand]\$ gotimer interruption! 18 tick: 1 19 Task 1: Created! 20 Task 1: Running... 21 Task 1: Running... 22 timer interruption! 23 tick: 2 24 Task 0: Created! 25 Sync exceptions! Code = 8 26 System call from U-mode! 27 --> sys_gettid, arg0 = 0x80005898</pre>	<pre>1 ----- 2 Hello, VoltOS! 3 developed by Lei Tang, SUEP. 4 Safty Mode: 5 -----welcome to Volt OS----- 6 page allocator initializing... 7 HEAP_START = 0x80008338(aligned to 0x80009000), 8 HEAP_SIZE = 0x07ff7cc8, 9 num of reserved pages = 8, 10 num of pages to be allocated for heap = 32751 11 TEXT: 0x80000000 -> 0x80004258 12 RODATA: 0x80004258 -> 0x8000481f 13 DATA: 0x80005000 -> 0x8000502c 14 BSS: 0x80005030 -> 0x80008338 15 HEAP: 0x80011000 -> 0x88000000 16 page initialized. 17 [VoltCommand]\$ gotimer interruption! 18 tick: 1 19 Task 1: Created! 20 Task 1: Running... 21 Task 1: Running... 22 timer interruption! 23 tick: 2 24 Task 0: Created! 25 Sync exceptions! Code = 8 26 System call from U-mode! 27 --> sys_gettid, arg0 = 0x80005898</pre>
--	--

图 12 OWPFV 代码变异前后运行报告对比