



# 프로젝트 #4 스택의 구조 콜스택 복원하기

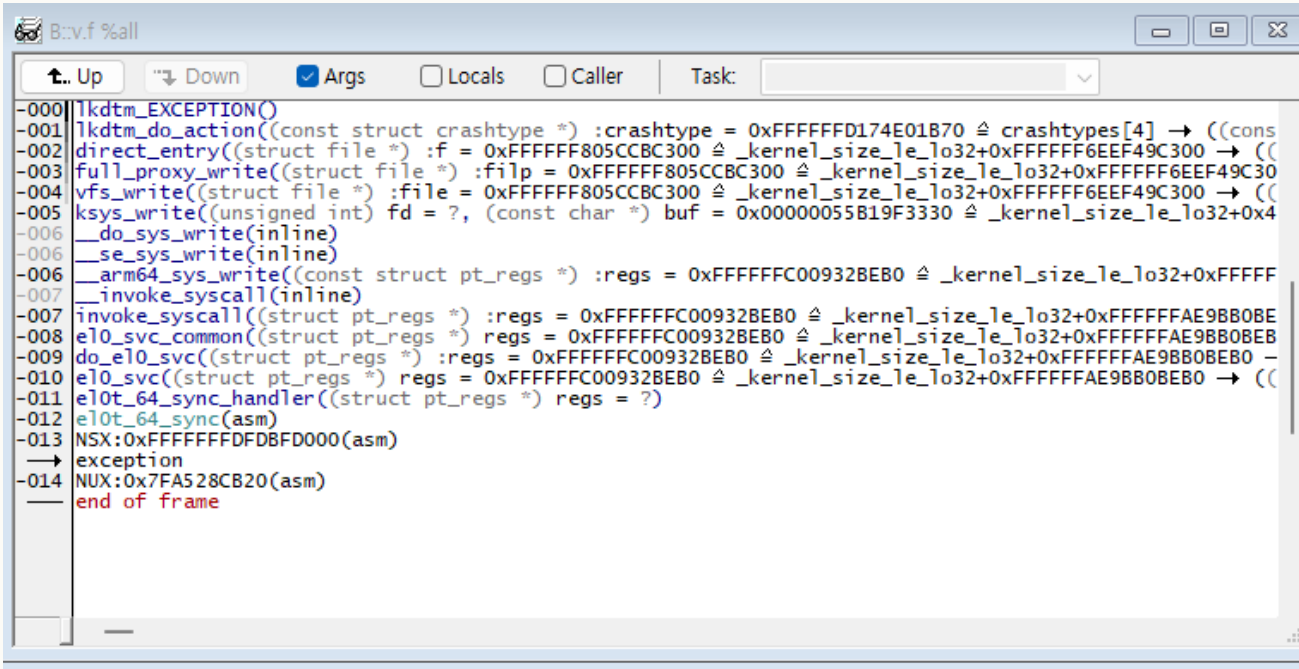
리눅스 시스템 및 커널 전문가 2기 김현성

# 목차



1. 프로젝트 주제
2. 프로세스와 스택
3. 콜스택 복원하기
4. Cmm 스크립트

# 1 프로젝트 주제



```
-000 tkdtm_EXCEPTION()
-001 tkdtm_do_action((const struct crashtype *) :crashtype = 0xFFFFFDD174E01B70 => crashtypes[4] -> ((cons
-002 direct_entry((struct file *) :f = 0xFFFFF805CCBC300 => _kernel_size_le_lo32+0xFFFFF6EEF49C300 -> ((
-003 full_proxy_write((struct file *) :filp = 0xFFFFF805CCBC300 => _kernel_size_le_lo32+0xFFFFF6EEF49C30
-004 vfs_write((struct file *) :file = 0xFFFFF805CCBC300 => _kernel_size_le_lo32+0xFFFFF6EEF49C300 -> ((
-005 ksys_write((unsigned int) fd = ?, (const char *) buf = 0x00000055B19F3330 => _kernel_size_le_lo32+0x4
-006 __do_sys_write(inline)
-006 __se_sys_write(inline)
-006 __arm64_sys_write((const struct pt_regs *) :regs = 0xFFFFFCC00932BEB0 => _kernel_size_le_lo32+0xFFFFF
-007 __invoke_syscall(inline)
-007 invoke_syscall((struct pt_regs *) :regs = 0xFFFFFCC00932BEB0 => _kernel_size_le_lo32+0xFFFFFAE9BB0BE
-008 e10_svc_common((struct pt_regs *) :regs = 0xFFFFFCC00932BEB0 => _kernel_size_le_lo32+0xFFFFFAE9BB0BEB
-009 do_e10_svc((struct pt_regs *) :regs = 0xFFFFFCC00932BEB0 => _kernel_size_le_lo32+0xFFFFFAE9BB0BEB0 -
-010 e10_svc((struct pt_regs *) :regs = 0xFFFFFCC00932BEB0 => _kernel_size_le_lo32+0xFFFFFAE9BB0BEB0 -> ((
-011 e10t_64_sync_handler((struct pt_regs *) :regs = ?)
-012 e10t_64_sync(asm)
-013 NSX:0xFFFFFFFDFBFD000(asm)
-> exception
-014 NUX:0x7FA528CB20(asm)
- end of frame
```

그림 1. 콜스택

“스택 덤프를 이동하며 콜스택 복원시”

- ✓ 프로세스 스택 Corruption 시 손쉽게 디버깅 가능
- ✓ 스택 오버플로우 디버깅
- ✓ 프로세스의 콜스택을 보며 함수의 실행 흐름을 파악

## 2 프로세스와 스택

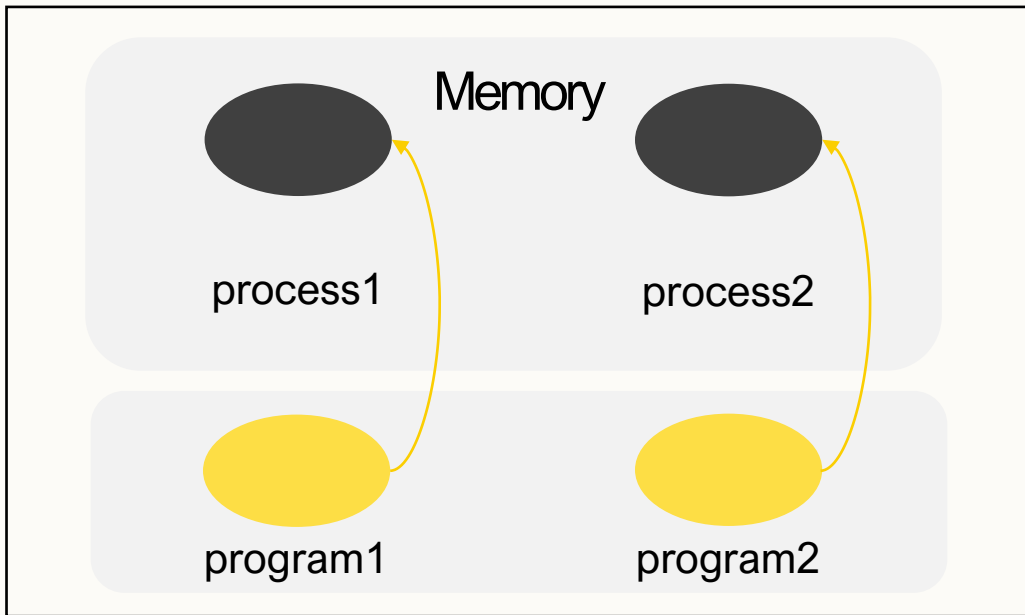


그림 2. 프로세스

### 프로그램 vs 프로세스

- ✓ 실행중인 프로그램
- ✓ 독립적인 실행
- ✓ 프로세스별 최소 1개의 스레드 생성

## 2 프로세스와 스택

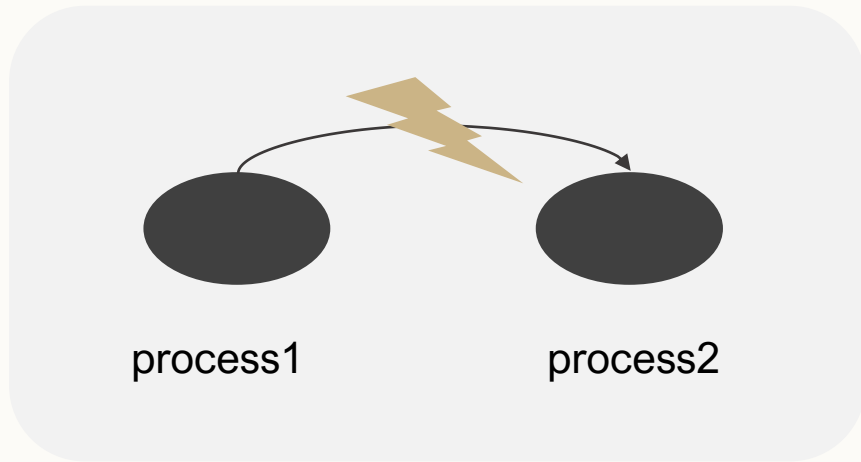


그림3. 프로세스간 충돌



문제해결

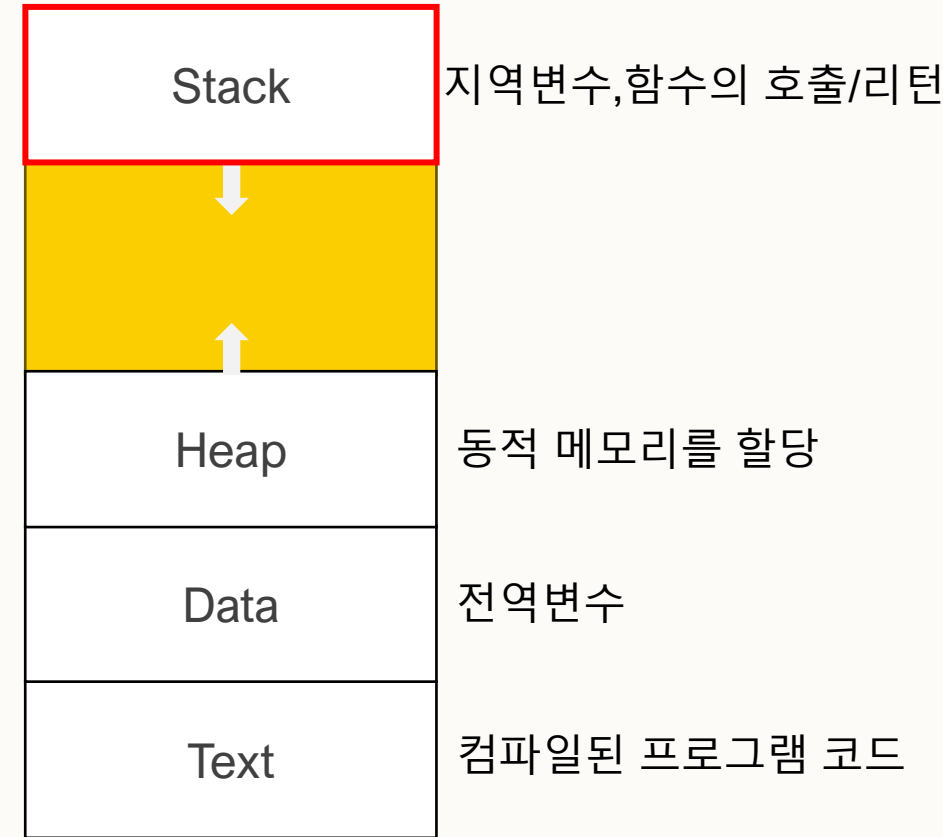


그림4. 프로세스의 주소공간

- OS는 개별 프로세스에게 독립적인 주소공간을 제공하여 프로세스간 충돌과 데이터 손실을 방지

## 2 프로세스와 스택

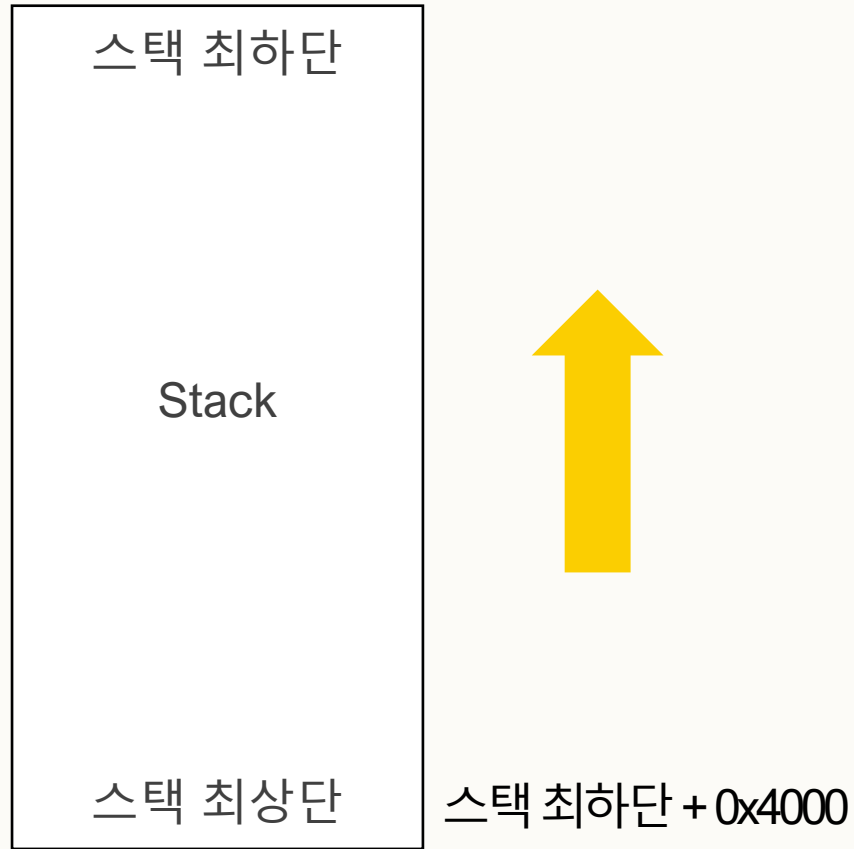


그림5. 스택

### 스택

“Key point”

- ✓ 스택은 높은 주소에서 낮은 주소로 자라난다.
- ✓ Armv8기반 리눅스 커널에서는 스택의 사이즈가 0x4000이다.

## 2 프로세스와 스택

### task\_struct 구조체

- /include/linux/sched.h
- 프로세스의 속성 정보를 표현하는 구조체
- task\_struct 구조체 필드의 stack에는 스택의 최하단 주소 값이다.

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
    /*
     * For reasons of header soup (see current_thread_info()), this
     * must be the first element of task_struct.
     */
    struct thread_info      thread_info;
#endif
    unsigned int             __state;

    /* saved state for "spinlock sleepers" */
    unsigned int             saved_state;

    /*
     * This begins the randomizable portion of task_struct. Only
     * scheduling-critical items should be added above here.
     */
    randomized_struct_fields_start

    void                     *stack;
    refcount_t               usage;
    /* Per task flags (PF_*), defined further below: */
    unsigned int             flags;
    unsigned int             ptrace;
```

그림 6. struct task\_struct

### 3 콜스택 복원하기

crash64> runq -m

```
crash64> runq -m
CPU 0: [0 01:10:11.687] PID: 0      TASK: ffffffff17553fac0  COMMAND: "swapper/
0"
CPU 1: [0 01:10:11.687] PID: 0      TASK: ffffffff80402ble40  COMMAND: "swapper/
1"
CPU 2: [0 00:00:00.000] PID: 1591   TASK: ffffffff805d66dac0  COMMAND: "bash"
CPU 3: [0 01:10:11.686] PID: 0      TASK: ffffffff80402b5ac0  COMMAND: "swapper/
3"
```

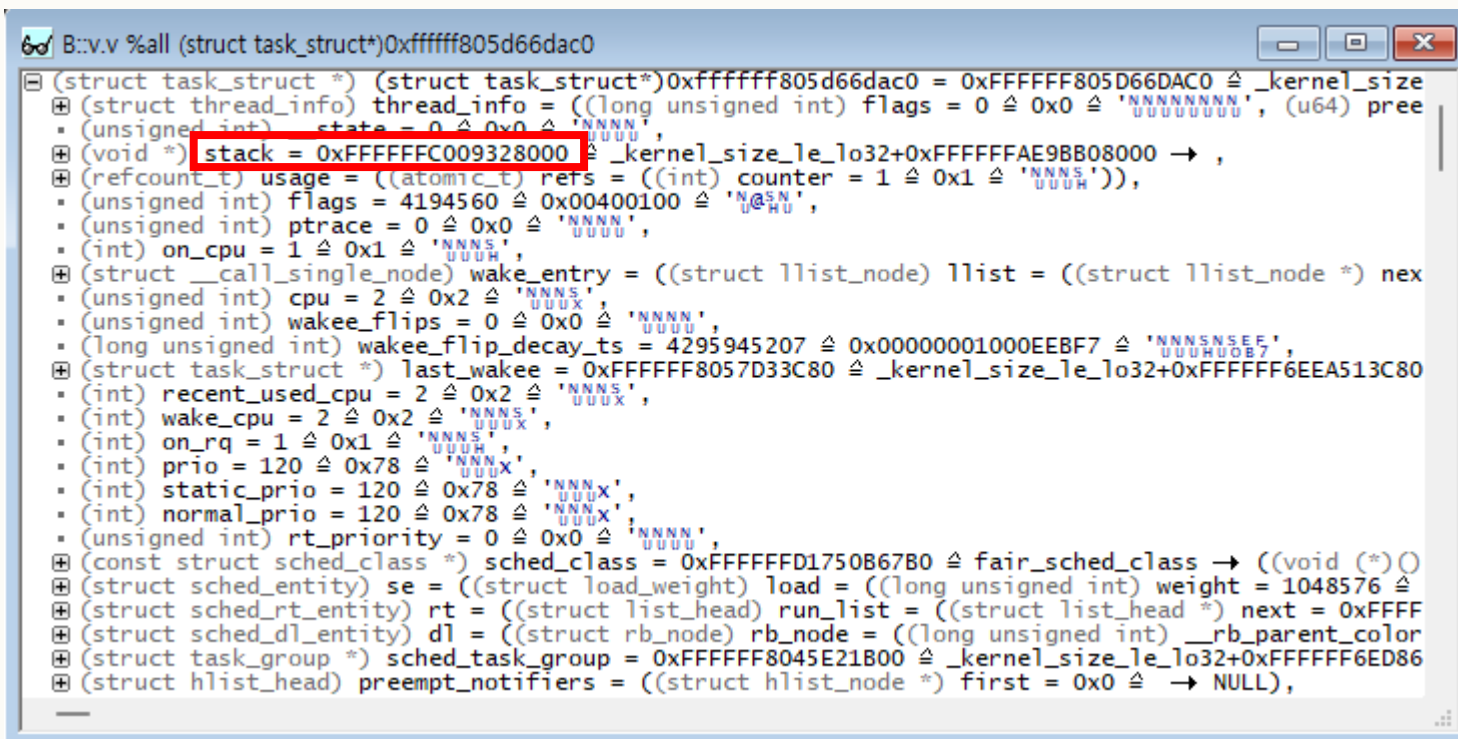
그림 7. runq-m

crash utility 프로그램에서 bash 프로세스의 태스크 디스크립터 주소 확인



### 3 콜스택 복원하기

TRACE32> v.v %all (struct task\_struct\*)0xfffff805d66dac0



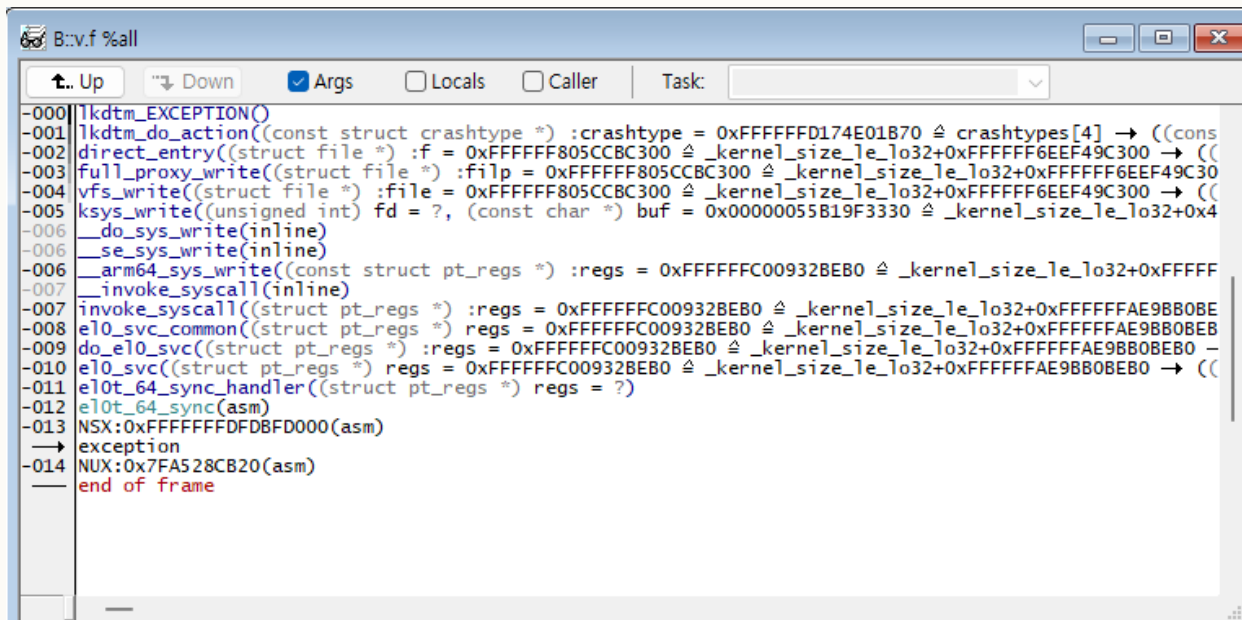
```
B::v.v %all (struct task_struct*)0xfffff805d66dac0
(struct task_struct *) (struct task_struct*)0xfffff805d66dac0 = 0xFFFFF805D66DAC0 ≙ _kernel_size
+ (struct thread_info) thread_info = ((long unsigned int) flags = 0 ≙ 0x0 ≙ 'NNNNNNNN', (u64) pree
  (unsigned int) state = 0 ≙ 0x0 ≙ 'NNNN',
+ (void *) stack = 0xFFFFF8009328000 ≙ _kernel_size_le_lo32+0xFFFFF8AE9BB08000 → ,
+ (refcount_t) usage = ((atomic_t) refs = ((int) counter = 1 ≙ 0x1 ≙ 'NNNS'),
  (unsigned int) flags = 4194560 ≙ 0x00400100 ≙ 'N@5N',
  (unsigned int) ptrace = 0 ≙ 0x0 ≙ 'NNNN',
  (int) on_cpu = 1 ≙ 0x1 ≙ 'NNNS',
+ (struct __call_single_node) wake_entry = ((struct llist_node) llist = ((struct llist_node *) nex
  (unsigned int) cpu = 2 ≙ 0x2 ≙ 'NNNS',
  (unsigned int) wakee_flips = 0 ≙ 0x0 ≙ 'NNNN',
  (long unsigned int) wakee_flip_decay_ts = 4295945207 ≙ 0x000000001000EEBF7 ≙ 'NNNSNSEE',
+ (struct task_struct *) last_wakee = 0xFFFFF8057D33C80 ≙ _kernel_size_le_lo32+0xFFFFF86EEA513C80
  (int) recent_used_cpu = 2 ≙ 0x2 ≙ 'NNNS',
  (int) wake_cpu = 2 ≙ 0x2 ≙ 'NNNS',
  (int) on_rq = 1 ≙ 0x1 ≙ 'NNNS',
  (int) prio = 120 ≙ 0x78 ≙ 'NNN',
  (int) static_prio = 120 ≙ 0x78 ≙ 'NNN',
  (int) normal_prio = 120 ≙ 0x78 ≙ 'NNN',
  (unsigned int) rt_priority = 0 ≙ 0x0 ≙ 'NNNN',
+ (const struct sched_class *) sched_class = 0xFFFFF8D1750B6780 ≙ fair_sched_class → ((void (*)())
+ (struct sched_entity) se = ((struct load_weight) load = ((long unsigned int) weight = 1048576 ≙
+ (struct sched_rt_entity) rt = ((struct list_head) run_list = ((struct list_head *) next = 0xFFFF
+ (struct sched_dl_entity) dl = ((struct rb_node) rb_node = ((long unsigned int) __rb_parent_color
+ (struct task_group *) sched_task_group = 0xFFFFF8045E21B00 ≙ _kernel_size_le_lo32+0xFFFFF86ED86
+ (struct hlist_head) preempt_notifiers = ((struct hlist_node *) first = 0x0 ≙ → NULL),
```

그림 8. task\_struct 구조체

TRACE32 프로그램에서 bash 프로세스의 task\_struct 구조체 압력 => 스택 주소 확인

# 3 콜스택 복원하기

TRACE32> v.f %all



```
-000|lkdtm_EXCEPTION()
-001|lkdtm_do_action((const struct crashtype *) :crashtype = 0xFFFFFFFFD174E01B70 &= crashtypes[4] → ((cons
-002|direct_entry((struct file *) :f = 0xFFFFFFFF805CCBC300 &= _kernel_size_le_lo32+0xFFFFFFFF6EEF49C300 → ((
-003|full_proxy_write((struct file *) :filep = 0xFFFFFFFF805CCBC300 &= _kernel_size_le_lo32+0xFFFFFFFF6EEF49C30
-004|vfs_write((struct file *) :file = 0xFFFFFFFF805CCBC300 &= _kernel_size_le_lo32+0xFFFFFFFF6EEF49C300 → ((
-005|ksys_write(unsigned int) fd = ?, (const char *) buf = 0x000000055B19F3330 &= _kernel_size_le_lo32+0x4
-006|__do_sys_write(inline)
-006|__se_sys_write(inline)
-006|__arm64_sys_write((const struct pt_regs *) :regs = 0xFFFFF0009328EB0 &= _kernel_size_le_lo32+0xFFFFF
-007|__invoke_syscall(inline)
-007|invoke_syscall((struct pt_regs *) :regs = 0xFFFFF0009328EB0 &= _kernel_size_le_lo32+0xFFFFF0009328EB0
-008|el0_svc_common((struct pt_regs *) :regs = 0xFFFFF0009328EB0 &= _kernel_size_le_lo32+0xFFFFF0009328EB0
-009|do_el0_svc((struct pt_regs *) :regs = 0xFFFFF0009328EB0 &= _kernel_size_le_lo32+0xFFFFF0009328EB0 → ((
-010|el0_svc((struct pt_regs *) :regs = 0xFFFFF0009328EB0 &= _kernel_size_le_lo32+0xFFFFF0009328EB0 → ((
-011|el0t_64_sync_handler((struct pt_regs *) :regs = ?)
-012|el0t_64_sync(asm)
-013|NSX:0xFFFFF0009328EB0(asm)
→ exception
-014|NEX:0x7FA528CB20(asm)
---end of frame
```

그림 9. 콜스택

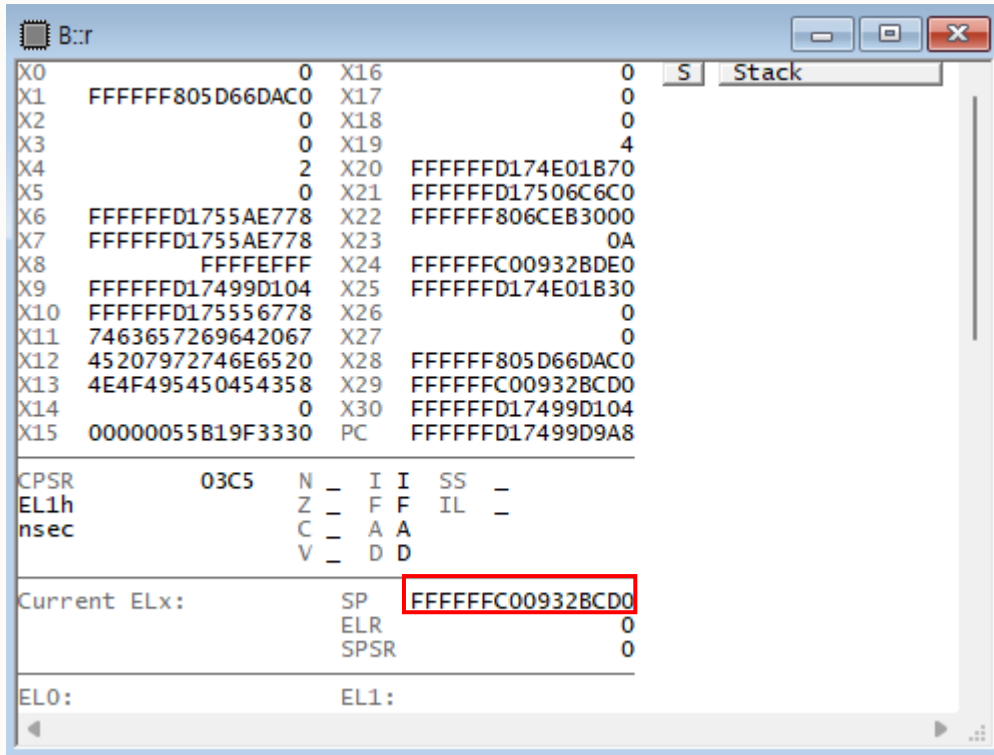
v.f로 확인한 콜스택은 다음과 같다.



그림 10. 콜스택 정리

### 3 콜스택 복원하기

TRACE32> v.f %all



X0	0	X16	0
X1	FFFFFF805D66DAC0	X17	0
X2	0	X18	0
X3	0	X19	4
X4	2	X20	FFFFFFFD174E01B70
X5	0	X21	FFFFFFFD17506C6C0
X6	FFFFFFFD1755AE778	X22	FFFFFFF806CEB3000
X7	FFFFFFFD1755AE778	X23	0A
X8	FFFFFFEFFF	X24	FFFFFFFC00932BDE0
X9	FFFFFFFD17499D104	X25	FFFFFFFD174E01B30
X10	FFFFFFFD175556778	X26	0
X11	7463657269642067	X27	0
X12	45207972746E6520	X28	FFFFFFF805D66DAC0
X13	4E4F495450454358	X29	FFFFFFFC00932BCD0
X14	0	X30	FFFFFFFD17499D104
X15	00000055B19F3330	PC	FFFFFFFD17499D9A8

CPSR	03C5	N	-	I	I	SS	-
EL1h		Z	-	F	F	IL	-
nsec		C	-	A	A		
		V	-	D	D		

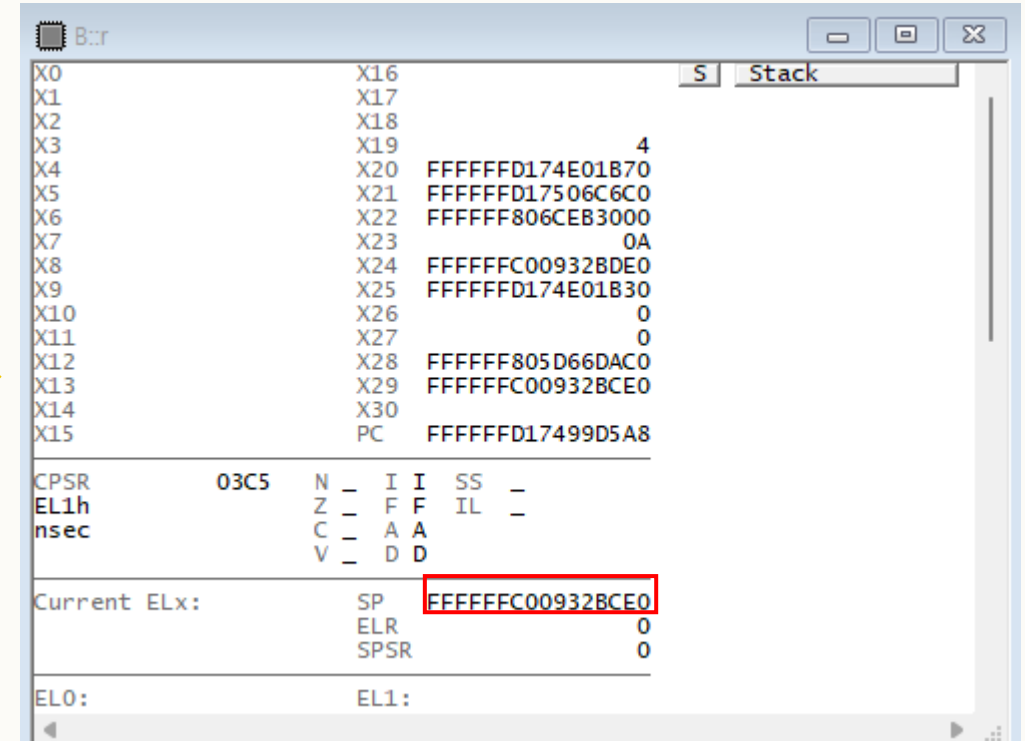
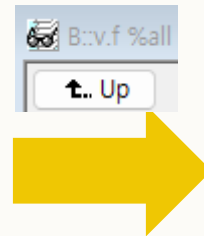
  

Current ELx:	SP	FFFFFFFC00932BCD0
	ELR	0
	SPSR	0

EL0:	EL1:
------	------

그림 11. sp 레지스터



X0		X16	
X1		X17	
X2		X18	
X3		X19	4
X4		X20	FFFFFFFD174E01B70
X5		X21	FFFFFFFD17506C6C0
X6		X22	FFFFFFF806CEB3000
X7		X23	0A
X8		X24	FFFFFFFC00932BDE0
X9		X25	FFFFFFFD174E01B30
X10		X26	0
X11		X27	0
X12		X28	FFFFFFF805D66DAC0
X13		X29	FFFFFFFC00932BCE0
X14		X30	
X15		PC	FFFFFFFD17499D5A8

CPSR	03C5	N	-	I	I	SS	-
EL1h		Z	-	F	F	IL	-
nsec		C	-	A	A		
		V	-	D	D		

Current ELx:	SP	FFFFFFFC00932BCE0
	ELR	0
	SPSR	0

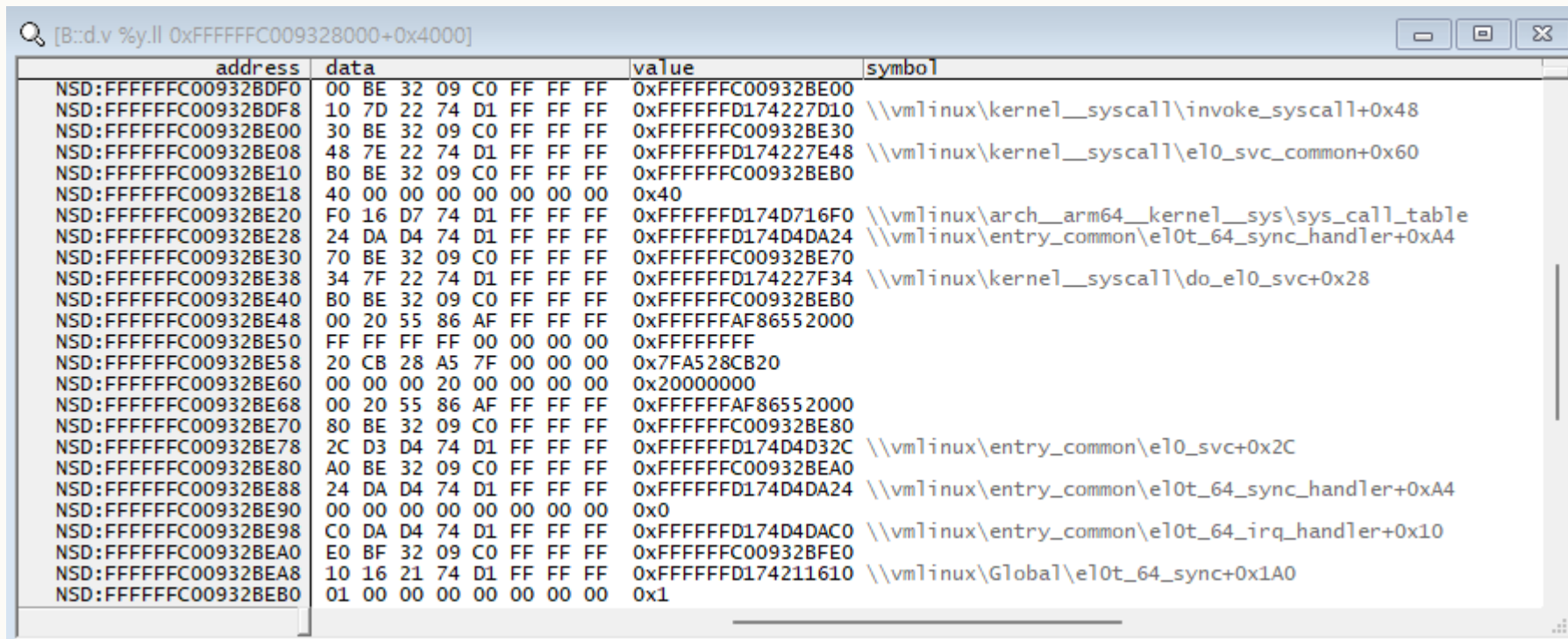
EL0:	EL1:
------	------

그림 12. sp 레지스터

콜스택을 up 으로 이동시켰을 시 SP 레지스터 값이 변경된다.

### 3 콜스택 복원하기

TRACE32> d.v %y.ll 0xFFFFFFFFC009328000+0x4000



[B::d.v %y.ll 0xFFFFFFFFC009328000+0x4000]

	address	data	value	symbol
NSD:	FFFFFFFFC009328DF0	00 BE 32 09 C0 FF FF FF	0xFFFFFFFFC009328E00	
NSD:	FFFFFFFFC009328DF8	10 7D 22 74 D1 FF FF FF	0xFFFFFFFFD174227D10	\\vmlinux\kernel__syscall\invoke_syscall+0x48
NSD:	FFFFFFFFC009328E00	30 BE 32 09 C0 FF FF FF	0xFFFFFFFFC009328E30	
NSD:	FFFFFFFFC009328E08	48 7E 22 74 D1 FF FF FF	0xFFFFFFFFD174227E48	\\vmlinux\kernel__syscall\e10_svc_common+0x60
NSD:	FFFFFFFFC009328E10	80 BE 32 09 C0 FF FF FF	0xFFFFFFFFC009328EB0	
NSD:	FFFFFFFFC009328E18	40 00 00 00 00 00 00 00	0x40	
NSD:	FFFFFFFFC009328E20	F0 16 D7 74 D1 FF FF FF	0xFFFFFFFFD174D716F0	\\vmlinux\arch__arm64__kernel__sys\sys_call_table
NSD:	FFFFFFFFC009328E28	24 DA D4 74 D1 FF FF FF	0xFFFFFFFFD174D4DA24	\\vmlinux\entry_common\e10t_64_sync_handler+0xA4
NSD:	FFFFFFFFC009328E30	70 BE 32 09 C0 FF FF FF	0xFFFFFFFFC009328E70	
NSD:	FFFFFFFFC009328E38	34 7F 22 74 D1 FF FF FF	0xFFFFFFFFD174227F34	\\vmlinux\kernel__syscall\do_e10_svc+0x28
NSD:	FFFFFFFFC009328E40	80 BE 32 09 C0 FF FF FF	0xFFFFFFFFC009328EB0	
NSD:	FFFFFFFFC009328E48	00 20 55 86 AF FF FF FF	0xFFFFFFFFAF86552000	
NSD:	FFFFFFFFC009328E50	FF FF FF FF 00 00 00 00	0xFFFFFFFF	
NSD:	FFFFFFFFC009328E58	20 CB 28 A5 7F 00 00 00	0x7FA528CB20	
NSD:	FFFFFFFFC009328E60	00 00 00 20 00 00 00 00	0x20000000	
NSD:	FFFFFFFFC009328E68	00 20 55 86 AF FF FF FF	0xFFFFFFFFAF86552000	
NSD:	FFFFFFFFC009328E70	80 BE 32 09 C0 FF FF FF	0xFFFFFFFFC009328E80	
NSD:	FFFFFFFFC009328E78	2C D3 D4 74 D1 FF FF FF	0xFFFFFFFFD174D4D32C	\\vmlinux\entry_common\e10_svc+0x2C
NSD:	FFFFFFFFC009328E80	A0 BE 32 09 C0 FF FF FF	0xFFFFFFFFC009328EA0	
NSD:	FFFFFFFFC009328E88	24 DA D4 74 D1 FF FF FF	0xFFFFFFFFD174D4DA24	\\vmlinux\entry_common\e10t_64_sync_handler+0xA4
NSD:	FFFFFFFFC009328E90	00 00 00 00 00 00 00 00	0x0	
NSD:	FFFFFFFFC009328E98	C0 DA D4 74 D1 FF FF FF	0xFFFFFFFFD174D4DAC0	\\vmlinux\entry_common\e10t_64_irq_handler+0x10
NSD:	FFFFFFFFC009328EA0	E0 BF 32 09 C0 FF FF FF	0xFFFFFFFFC009328FE0	
NSD:	FFFFFFFFC009328EA8	10 16 21 74 D1 FF FF FF	0xFFFFFFFFD174211610	\\vmlinux\Global\e10t_64_sync+0x1A0
NSD:	FFFFFFFFC009328EB0	01 00 00 00 00 00 00 00	0x1	

그림 13. 스택 덤프

d.v명령어를 입력하여 스택 주소의 덤프값을 확인

### 3 콜스택 복원하기

TRACE32> r.init

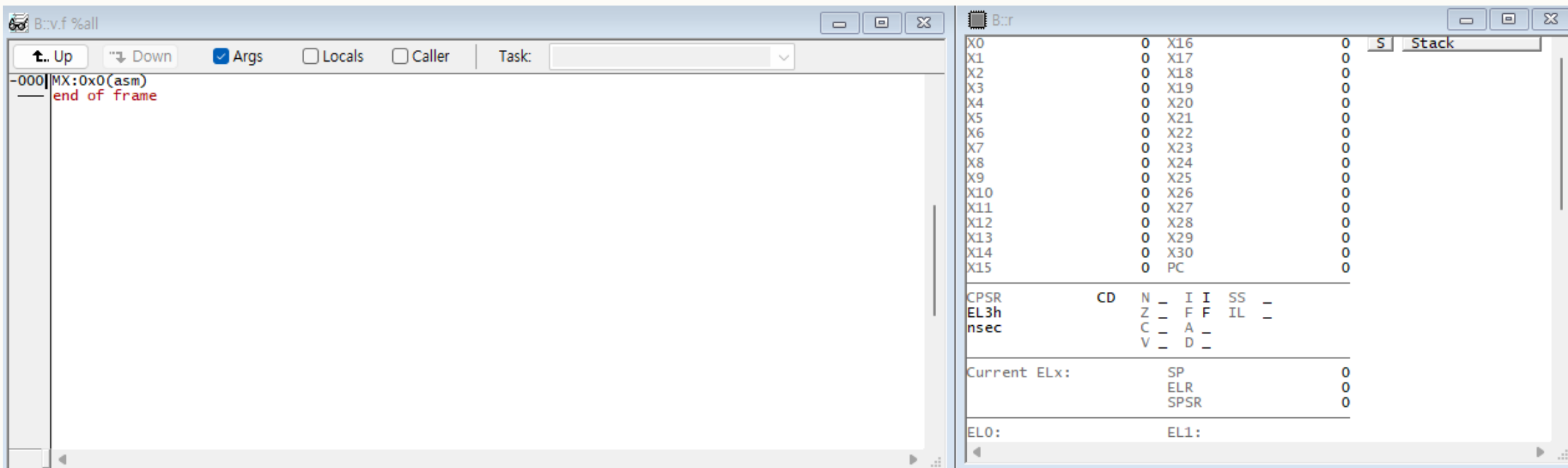


그림 14. 레지스터 초기화

콜스택 복원 실습을 위해 레지스터 값을 초기화 시켜주어 콜스택을 삭제

### 3 콜스택 복원하기

```
TRACE32> r.s pc 0xFFFFFD174D4DA24
          r.s sp 0xFFFFFC00932BEA0
          r.s x29 0xFFFFFC00932BEA0
```

The screenshot shows the TRACE32 debugger interface. The left pane displays a call stack with the following entries:

- 000| el0t\_64\_sync\_handler((struct pt\_regs \*) regs = ?)
- 001| el0t\_64\_sync(asm)
- 002| NSX:0xFFFFFDFDBFD000(asm)
- exception
- 003| NUX:0x7FA528CB20(asm)
- end of frame

The right pane shows the current state of registers and stack:

Register	Value
X0	0
X1	0
X2	0
X3	0
X4	0
X5	0
X6	0
X7	0
X8	0
X9	0
X10	0
X11	0
X12	0
X13	0
X14	0
X15	0
X16	0
X17	0
X18	0
X19	0
X20	0
X21	0
X22	0
X23	0
X24	0
X25	0
X26	0
X27	0
X28	0
X29	0
X30	0
PC	FFFFFD174D4DA24

Below the register list, the CPSR (Current Program Status Register) is shown with fields N, Z, C, V, I, F, A, D, SS, IL, and IL. The current ELx (Exception Level) is shown as SP, ELR, and SPSR. The stack pointer (SP) is shown as FFFFFFFC00932BEA0.

그림 15. 콜스택 복원

Arm 아키텍처 calling convention 규약에 맞게 pc, sp, x29 레지스터에 값 입력

## 4 Cmm 스크립트

### 전역변수를 선언하는 소스코드

```
9 Global &g_init_task_addr
10 Global &g_init_task_list_addr
11 Global &g_task_struct_start_addr
12 Global &g_task_struct_name
13 Global &g_task_struct_state
14 Global &g_task_struct_thread // added
15
16 Global &g_task_struct_list_next
17 Global &g_task_struct_list_next_temp
18 Global &g_task_struct_list_offset
19 Global &g_task_struct_name_offset
20 Global &g_task_struct_state_offset
21 Global &g_task_struct_thread_offset // added
```

그림 16. 전역변수

## 4 Cmm 스크립트

### 각 필드의 오프셋을 저장하는 코드

```
23 ; find the offset
24 &g_task_struct_list_offset=address.offset(v.address(((struct task_struct)0x0).tasks))
25 &g_task_struct_name_offset=address.offset(v.address(((struct task_struct)0x0).comm))
26
27 &g_task_struct_state_offset=address.offset(v.address(((struct task_struct)0x0).__state))
28 &g_task_struct_thread_offset=address.offset(v.address(((struct task_struct)0x0).thread))
```

그림 17. 오프셋 값 저장

각 필드들의 오프셋을 해당 전역변수에 저장한다.



## 4 Cmm 스크립트

### Linkedlist를 순회하며 show thread 함수로 이동

```
36 while &g_init_task_list_addr!=&g_task_struct_list_next
37 (
38     &g_task_struct_start_addr=&g_task_struct_list_next-&g_task_struct_list_offset
39
40     &g_task_struct_name=&g_task_struct_start_addr+&g_task_struct_name_offset
41     &g_task_struct_state=&g_task_struct_start_addr+&g_task_struct_state_offset
42
43     print "process name: " data.string(d:&g_task_struct_name)
44
45     gosub show_thread &g_task_struct_start_addr
46
47     &g_task_struct_list_next_temp=data.quad(d:&g_task_struct_list_next)
48     &g_task_struct_list_next=&g_task_struct_list_next_temp // for debugging purpose
49 )
```

그림 18. Linked list 순회

While문을 돌며 task라는 필드의 포인팅 하고 있는 Linked list를 순회하며 프로세스 이름을 출력하며, gosub 구문을 통해 show\_thread 함수로 이동한다.

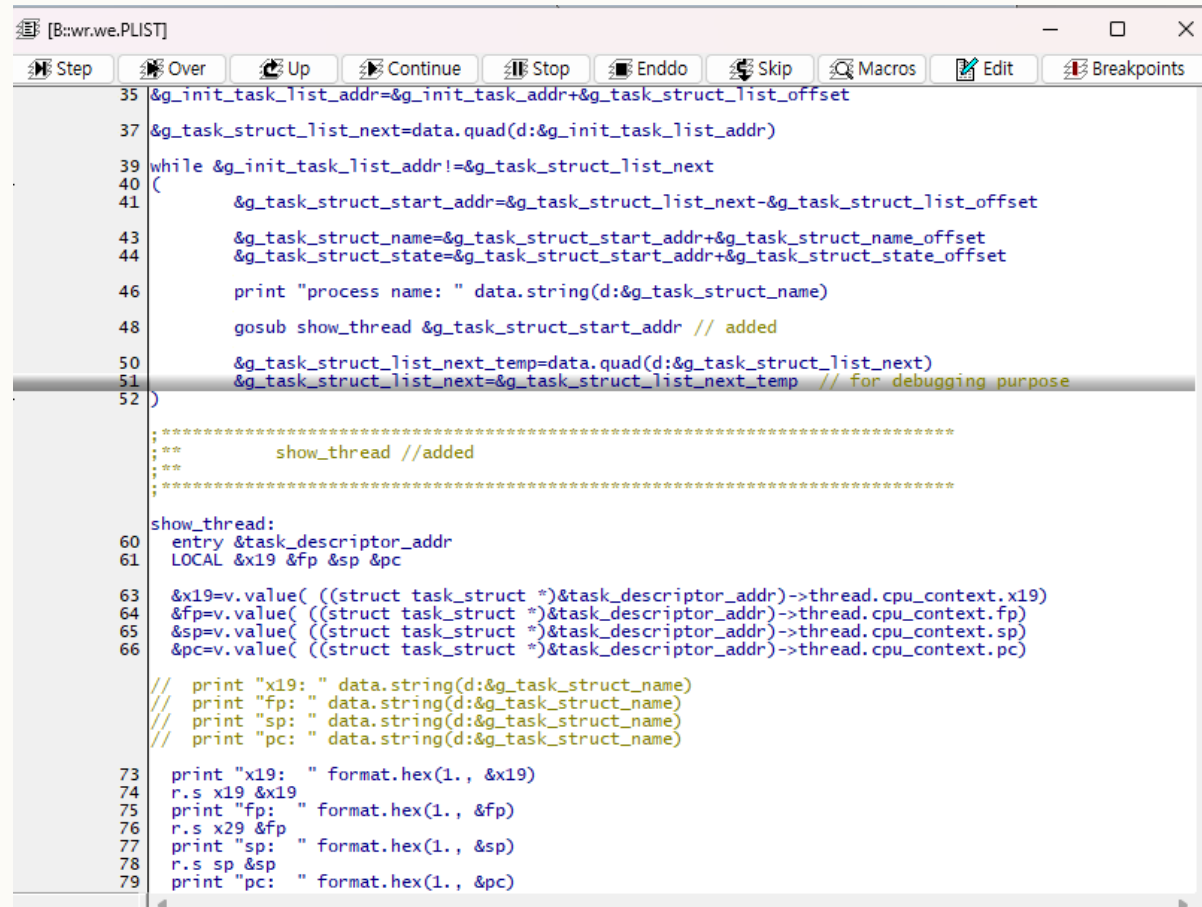
## 4 Cmm 스크립트

```
54 ;*****
55 ;**      show_thread //added
56 ;**
57 ;*****
58
59 show_thread:
60     entry &task_descriptor_addr
61     LOCAL &x19 &fp &sp &pc
62
63     &x19=v.value( ((struct task_struct *)&task_descriptor_addr)->thread.cpu_context.x19)
64     &fp=v.value( ((struct task_struct *)&task_descriptor_addr)->thread.cpu_context.fp)
65     &sp=v.value( ((struct task_struct *)&task_descriptor_addr)->thread.cpu_context.sp)
66     &pc=v.value( ((struct task_struct *)&task_descriptor_addr)->thread.cpu_context.pc)
67
68     // print "x19: " data.string(d:&g_task_struct_name)
69     // print "fp: " data.string(d:&g_task_struct_name)
70     // print "sp: " data.string(d:&g_task_struct_name)
71     // print "pc: " data.string(d:&g_task_struct_name)
72
73     print "x19: " format.hex(1., &x19)
74     r.s x19 &x19
75     print "fp: " format.hex(1., &fp)
76     r.s x29 &fp
77     print "sp: " format.hex(1., &sp)
78     r.s sp &sp
79     print "pc: " format.hex(1., &pc)
80     r.s pc &pc
81
82 RETURN
83
84 enddo
```

Task 디스크립터의 시작 주소를 인자로 받아 thread.cpu\_context의 레지스터의 값을 직접적으로 접근

# 4 Cmm 스크립트

TRACE32> pedit C:\troubleshooting\_vmcore\EXCEPTION\scripts/\*



```
[B::wr.we.PLIST]
Step Over Up Continue Stop Enddo Skip Macros Edit Breakpoints
35 &g_init_task_list_addr=&g_init_task_addr+&g_task_struct_list_offset
37 &g_task_struct_list_next=data.quad(d:&g_init_task_list_addr)
39 while &g_init_task_list_addr!=&g_task_struct_list_next
40 (
41     &g_task_struct_start_addr=&g_task_struct_list_next-&g_task_struct_list_offset
43     &g_task_struct_name=&g_task_struct_start_addr+&g_task_struct_name_offset
44     &g_task_struct_state=&g_task_struct_start_addr+&g_task_struct_state_offset
46     print "process name: " data.string(d:&g_task_struct_name)
48     gosub show_thread &g_task_struct_start_addr // added
50     &g_task_struct_list_next_temp=data.quad(d:&g_task_struct_list_next)
51     &g_task_struct_list_next=&g_task_struct_list_next_temp // for debugging purpose
52 )
;*****
;**      show_thread //added
;**
;*****
show_thread:
60 entry &task_descriptor_addr
61 LOCAL &x19 &fp &sp &pc
63 &x19=v.value( ((struct task_struct *)&task_descriptor_addr)->thread.cpu_context.x19)
64 &fp=v.value( ((struct task_struct *)&task_descriptor_addr)->thread.cpu_context.fp)
65 &sp=v.value( ((struct task_struct *)&task_descriptor_addr)->thread.cpu_context.sp)
66 &pc=v.value( ((struct task_struct *)&task_descriptor_addr)->thread.cpu_context.pc)
// print "x19: " data.string(d:&g_task_struct_name)
// print "fp: " data.string(d:&g_task_struct_name)
// print "sp: " data.string(d:&g_task_struct_name)
// print "pc: " data.string(d:&g_task_struct_name)
73 print "x19: " format.hex(1., &x19)
74 r.s x19 &x19
75 print "fp: " format.hex(1., &fp)
76 r.s x29 &fp
77 print "sp: " format.hex(1., &sp)
78 r.s sp &sp
79 print "pc: " format.hex(1., &pc)
```

그림 20. 디버깅 진행

# 4 Cmm 스크립트

Cmm 스크립트를 수행하여 콜스택이 복원된 모습

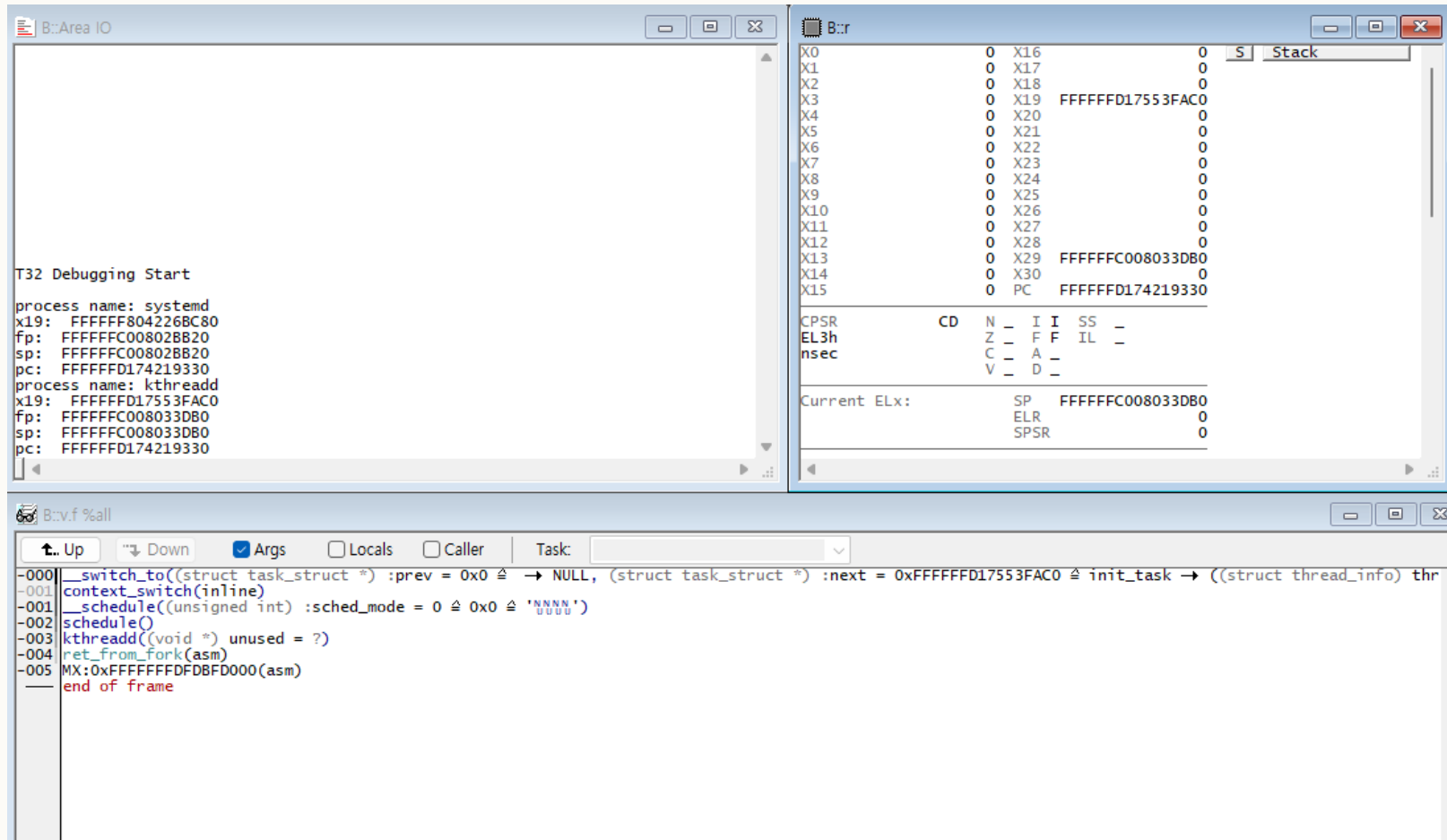


그림 21. 콜스택 복원

**Thank you**