Virtual Machine Homework 2

組員：

- B04703001 蔡明宏
- B04705001 陳約廷

1 - Armv8 Architecture

Exception model

We start from understanding the Armv8-A exception model, splitting into different levels of priviledge. Level of priviledge changes only when an exception occurs. Priviledge levels are referred as exception levels.

- $EL0$: unprivileged
- $EL1$: OS kernel mode
- $EL2$: Hypervisor mode
- $EL3$: TrustZone® monitor mode

Uses exceptions to require for higher priviledges. (like interupt, page fault...)

- $EL0 \rightarrow EL1$: SVC (system call)
- $EL1 \rightarrow EL2$: HVC (hypervisor call)
- $EL2 \rightarrow EL3$: SMC (secure monitor call)

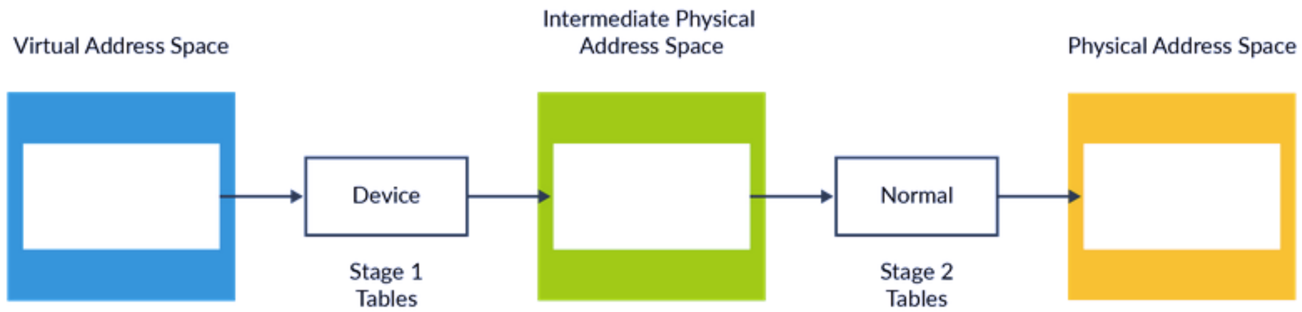
Memory Management

Armv8 manages memory in "**Memory Management Unit**" (MMU). A virtual address must be translated to a physical address before an memory access can take place.

Virtualization

This homework wants us to instrument the traps from guest to host in System VM. So we also need to understand the virtualization of Armv8-A.

Armv8 uses "**Stage 2 translation**" to allow hypervisor to control a view of memory in the VM.



Sensitive instruction

Guest needs a trap when sensitive instructions occur. Therefore we need to find the functions that calls it.

虛擬化指令(ARM)

問題指令(Problematic Instructions)

- Type I: 在user mode執行會產生未定義的指令異常
    - MCR、MRC: 需要依賴協處理器(coprocessor)
- Type II: 在user mode執行會沒有作用
    - MSR、MRS: 需要操作系統暫存器
- Type III: 在user mode執行會產生不可預測的行為
    - MOVS PC, LR: 返回指令，改變PC並跳回user mode，在user mode執行會產生不可預測的結果
- ARM 的敏感指令:
    - 存取協處理器: MRC / MCR / CDP / LDC / STC
    - 存取SIMD/VFP 系統暫存器: VMRS / VMSR
    - 進入TrustZone 安全狀態: SMC
    - 存取 Memory-Mapped I/O: Load/Store instructions from/into memory-mapped I/O locations
    - 直接存取CPSR: MRS / MSR / CPS / SRS / RFE / LDM (conditional execution) / DPSPC
    - 間接存取CPSR: LDRT / STRT – Load/Store Unprivileged ("As User")
    - 存取Banked Register: LDM / STM

In the picrure we can see that the traps interacting with the processor are `MRC/MCR/CDP/LDC/STC`.

2 - Trace code

Start to trace into the call of traps.

First we can see that TA already setted the instrument variables of trap count inside structure of `vcpu`.

```
#ifdef CONFIG_VM_HW2
    unsigned long cpu_trap_count;
    unsigned long mem_trap_count;
    unsigned long io_trap_count;
#endif
```

Next on we can see that inside `kvm_main.c` it, in TAG `KVM_RUN` it calls `kvm_arch_vcpu_ioctl_run`

```
    case KVM_RUN:
        r = -EINVAL;
        if (arg)
            goto out;
        r = kvm_arch_vcpu_ioctl_run(vcpu, vcpu->run);
        trace_kvm_userspace_exit(vcpu->run->exit_reason, r);
        break;
```

In `arm.c`, `kvm_arch_vcpu_ioctl_run` enters the guest, exits when `kvm_call_hyper`. Using the result from `kvm_call_hyper`, the `kvm` calls `handle_exit` to handle traps.

```
        /**************************************************************
         * Enter the guest
         */
        trace_kvm_entry(*vcpu_pc(vcpu));
        kvm_guest_enter();
        vcpu->mode = IN_GUEST_MODE;

        ret = kvm_call_hyp(__kvm_vcpu_run, vcpu);

        vcpu->mode = OUTSIDE_GUEST_MODE;
        vcpu->arch.last_pcpu = smp_processor_id();
        kvm_guest_exit();
        trace_kvm_exit(*vcpu_pc(vcpu));
        /*
         * We may have taken a host interrupt in HYP mode (ie
         * while executing the guest). This interrupt is still
         * pending, as we haven't serviced it yet!
         *
         * We're now back in SVC mode, with interrupts
         * disabled.  Enabling the interrupts now will have
         * the effect of taking the interrupt again, in SVC
         * mode this time.
         */
        local_irq_enable();

        /*
         * Back from guest
         *************************************************************/

        kvm_timer_sync_hwstate(vcpu);
        kvm_vgic_sync_hwstate(vcpu);

        ret = handle_exit(vcpu, run, ret);
#ifdef CONFIG_VM_HW2
        trace_kvm_vm_hw2(vcpu->cpu_trap_count, vcpu->mem_trap_count, vcpu->io_trap_count);
#endif
```

In the `handle_exit` function (in `handle_exit.c`), it calls `kvm_get_exit_handler` to get corresponding calls it need.

```c
/*
 * Return > 0 to return to guest, < 0 on error, 0 (and set exit_reason) on
 * proper exit to userspace.
 */
int handle_exit(struct kvm_vcpu *vcpu, struct kvm_run *run,
		       int exception_index)
{
	exit_handle_fn exit_handler;

	switch (exception_index) {
	case ARM_EXCEPTION_IRQ:
		return 1;
	case ARM_EXCEPTION_TRAP:
		/*
		 * See ARM ARM B1.14.1: "Hyp traps on instructions
		 * that fail their condition code check"
		 */
		if (!kvm_condition_valid(vcpu)) {
			kvm_skip_instr(vcpu, kvm_vcpu_trap_il_is32bit(vcpu));
			return 1;
		}

		exit_handler = kvm_get_exit_handler(vcpu);

		return exit_handler(vcpu, run);
```

Which are the following calls:

```c
static exit_handle_fn arm_exit_handlers[] = {
	[ESR_EL2_EC_WFI]	= kvm_handle_wfx,
	[ESR_EL2_EC_CP15_32]	= kvm_handle_cp15_32,          // CPU
	[ESR_EL2_EC_CP15_64]	= kvm_handle_cp15_64,          // CPU
	[ESR_EL2_EC_CP14_MR]	= kvm_handle_cp14_access,      // CPU
	[ESR_EL2_EC_CP14_LS]	= kvm_handle_cp14_load_store,  // CPU
	[ESR_EL2_EC_CP14_64]	= kvm_handle_cp14_access,      // CPU
	[ESR_EL2_EC_HVC32]	= handle_hvc,
	[ESR_EL2_EC_SMC32]	= handle_smc,
	[ESR_EL2_EC_HVC64]	= handle_hvc,
	[ESR_EL2_EC_SMC64]	= handle_smc,
	[ESR_EL2_EC_SYS64]	= kvm_handle_sys_reg,          // CPU
	[ESR_EL2_EC_IABT]	= kvm_handle_guest_abort,      // IO / Mem
	[ESR_EL2_EC_DABT]	= kvm_handle_guest_abort,      // IO / Mem
};
```

Also we can find there codes to lookup at the Armv8 manual:

```c
/* Hyp Prefetch Fault Address Register (HPFAR/HDFAR) */
#define HPFAR_MASK  (~0xFUL)

#define ESR_EL2_EC_UNKNOWN  (0x00)
#define ESR_EL2_EC_WFI      (0x01) // handle_wfx
#define ESR_EL2_EC_CP15_32  (0x03) // handle_cp15_32
#define ESR_EL2_EC_CP15_64  (0x04) // handle_cp15_64
#define ESR_EL2_EC_CP14_MR  (0x05) // handle_cp14_access
#define ESR_EL2_EC_CP14_LS  (0x06) // handle_cp14_load_store
#define ESR_EL2_EC_FP_ASIMD (0x07)
#define ESR_EL2_EC_CP10_ID  (0x08)
#define ESR_EL2_EC_CP14_64  (0x0C) // handle_cp14_access
#define ESR_EL2_EC_ILL_ISS  (0x0E)
#define ESR_EL2_EC_SVC32    (0x11)
#define ESR_EL2_EC_HVC32    (0x12) // handle_hvc
#define ESR_EL2_EC_SMC32    (0x13) // handle_smc
#define ESR_EL2_EC_SVC64    (0x15)
#define ESR_EL2_EC_HVC64    (0x16) // handle_hvc
#define ESR_EL2_EC_SMC64    (0x17) // handle_smc
#define ESR_EL2_EC_SYS64    (0x18)
#define ESR_EL2_EC_IABT     (0x20) // handle_guest_abort
#define ESR_EL2_EC_IABT_HYP (0x21)
#define ESR_EL2_EC_PC_ALIGN (0x22)
#define ESR_EL2_EC_DABT     (0x24) // handle_guest_abort
#define ESR_EL2_EC_DABT_HYP (0x25)
#define ESR_EL2_EC_SP_ALIGN (0x26)
#define ESR_EL2_EC_FP_EXC32 (0x28)
#define ESR_EL2_EC_FP_EXC64 (0x2C)
#define ESR_EL2_EC_SERROR   (0x2F)
#define ESR_EL2_EC_BREAKPT  (0x30)
#define ESR_EL2_EC_BREAKPT_HYP  (0x31)
#define ESR_EL2_EC_SOFTSTP  (0x32)
#define ESR_EL2_EC_SOFTSTP_HYP  (0x33)
#define ESR_EL2_EC_WATCHPT  (0x34)
#define ESR_EL2_EC_WATCHPT_HYP  (0x35)
#define ESR_EL2_EC_BKPT32   (0x38)
#define ESR_EL2_EC_VECTOR32 (0x3A)
#define ESR_EL2_EC_BRK64    (0x3C)
```

3 - Modified code

Modified

- `kernel_code/mmu.c` : from `linux-kvm-arm/arch/arm/kvm/mmu.c`
- `kernel_code/sys_reg.c` : from `linux-kvm-arm/arch/arm64/kvm/sys_reg.c`

IO trap / Memory trap

We look inside the `handle_guest_abort` function (inside `mmu.c`). We can see that it deals with traps relating to the second stage translation table entry.

```c
/**
 * kvm_handle_guest_abort - handles all 2nd stage aborts
 * @vcpu:   the VCPU pointer
 * @run:    the kvm_run structure
 *
 * Any abort that gets to the host is almost guaranteed to be caused by a
 * missing second stage translation table entry, which can mean that either the
 * guest simply needs more memory and we must allocate an appropriate page or it
 * can mean that the guest tried to access I/O memory, which is emulated by user
 * space. The distinction is based on the IPA causing the fault and whether this
 * memory region has been registered as standard RAM by user space.
 */
```

The key function we see is `kvm_is_visible_gfn`, where GFN stands for "**Guest Frame Number**".

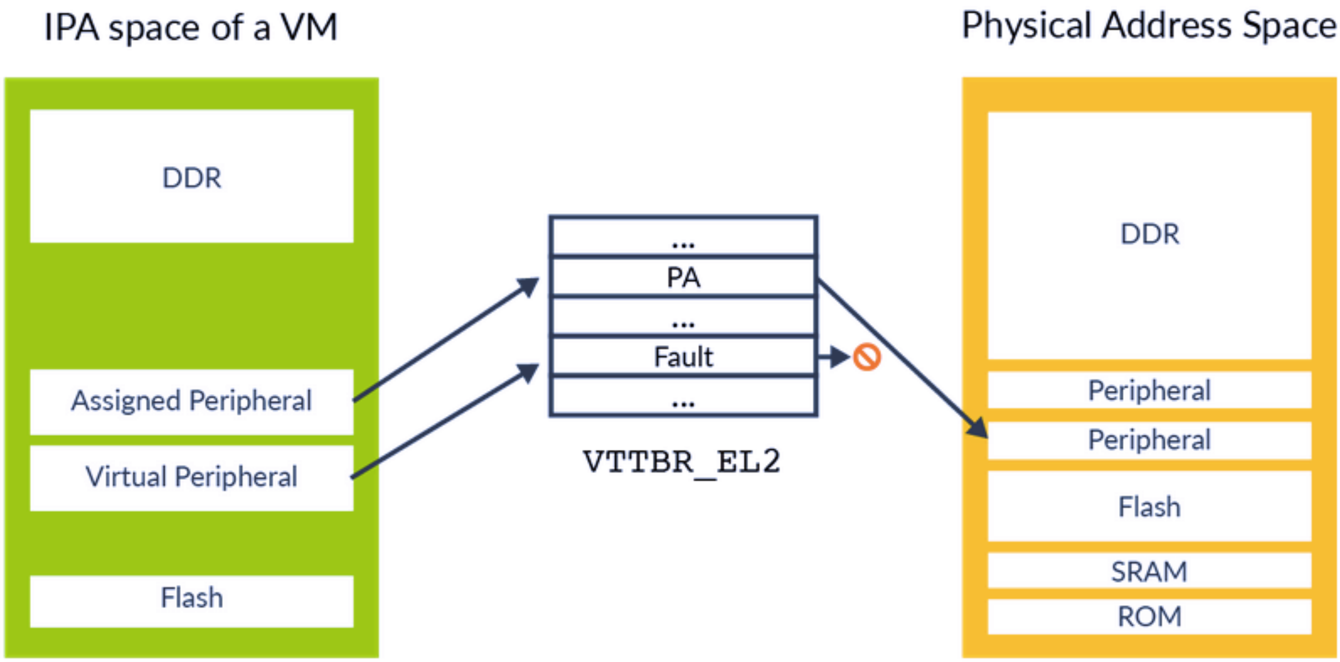
- When `kvm_is_visible_gfn` return `false`, the function tries to search for IO memory. (by the error log in the code) **We add `io_trap_count++` right here.**
- When `kvm_is_visible_gfn` returns `true`, it fetches for the corresponding memory slot. (fetched by `gfn_to_memslot`) **We add `mem_trap_count++` right here.**

```c
	idx = srcu_read_lock(&vcpu->kvm->srcu);

	gfn = fault_ipa >> PAGE_SHIFT;
	if (!kvm_is_visible_gfn(vcpu->kvm, gfn)) {
		if (is_iabt) {
			/* Prefetch Abort on I/O address */
			kvm_inject_pabt(vcpu, kvm_vcpu_get_hfar(vcpu));
			ret = 1;
			goto out_unlock;
		}
		vcpu->io_trap_count++;
		if (fault_status != FSC_FAULT) {
			kvm_err("Unsupported fault status on io memory: %#lx\n",
				fault_status);
			ret = -EFAULT;
			goto out_unlock;
		}

		/*
		 * The IPA is reported as [MAX:12], so we need to
		 * complement it with the bottom 12 bits from the
		 * faulting VA. This is always 12 bits, irrespective
		 * of the page size.
		 */
		fault_ipa |= kvm_vcpu_get_hfar(vcpu) & ((1 << 12) - 1);
		ret = io_mem_abort(vcpu, run, fault_ipa);
		goto out_unlock;
	}
	vcpu->mem_trap_count++;
	memslot = gfn_to_memslot(vcpu->kvm, gfn);
```

Translating from IPA (Intermediate Physical Address) to PA (Physical Address), kvm would need to check if the memory address shall be visible to the guest or not. This is what the function `kvm_is_visible_gfn` is doing.



CPU trap

> The Armv8-A architecture has a family of exception-generating instructions: **SVC, HVC, and SMC**. These instructions are different from a simple invalid instruction, because they target different exception levels and are treated differently when prioritizing exceptions. These instructions are used to implement system call interfaces to allow less privileged code to request services from more privileged code.
> from arm.developer.com

These instructions are synchronous exceptions that does not involves in Hyp Mode.

CPU traps are those of sensitive instructions that interacts with the coprocessor (mentioned above) - `MRC/MCR/CDP/LDC/STC` .

We check it out in `sys_reg.c` , and they do deal with these sensitive instrucitons. So we add `cpu_trap_count++` in each of these functions.

```c
int kvm_handle_cp14_load_store(struct kvm_vcpu *vcpu, struct kvm_run *run)
{
	vcpu->cpu_trap_count++;
	kvm_inject_undefined(vcpu);
	return 1;
}

int kvm_handle_cp14_access(struct kvm_vcpu *vcpu, struct kvm_run *run)
{
	vcpu->cpu_trap_count++;
	kvm_inject_undefined(vcpu);
	return 1;
}

```

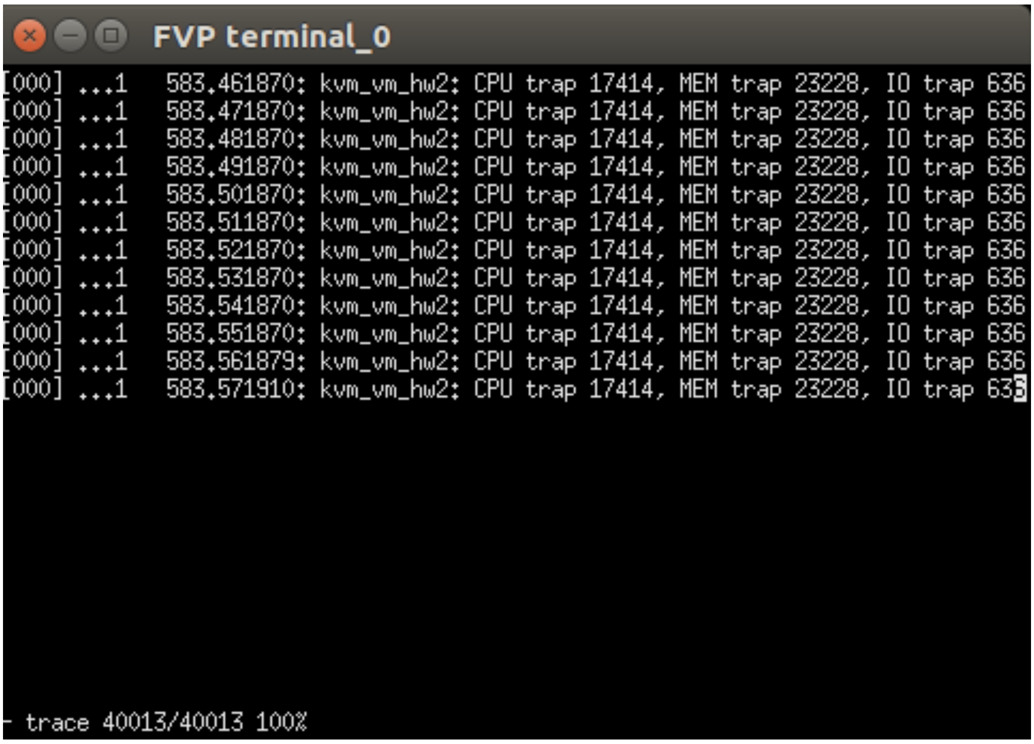
```c
/**
 * kvm_handle_cp15_64 -- handles a mrrc/mcrr trap on a guest CP15 access
 * @vcpu: The VCPU pointer
 * @run:  The kvm_run struct
 */
int kvm_handle_cp15_64(struct kvm_vcpu *vcpu, struct kvm_run *run)
```

```c
/**
 * kvm_handle_cp15_32 -- handles a mrc/mcr trap on a guest CP15 access
 * @vcpu: The VCPU pointer
 * @run:  The kvm_run struct
 */
int kvm_handle_cp15_32(struct kvm_vcpu *vcpu, struct kvm_run *run)
```

```c
/**
 * kvm_handle_sys_reg -- handles a mrs/msr trap on a guest sys_reg access
 * @vcpu: The VCPU pointer
 * @run:  The kvm_run struct
 */
int kvm_handle_sys_reg(struct kvm_vcpu *vcpu, struct kvm_run *run)
```

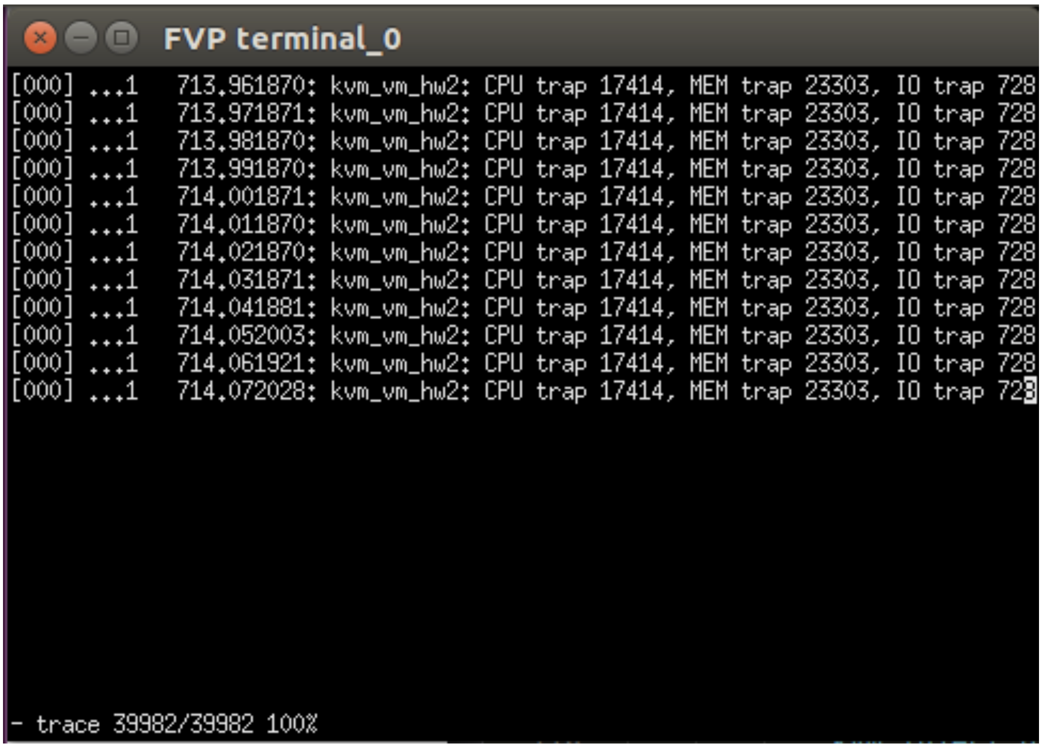
4 - Experiments

This section are of experiments that the homework requires.



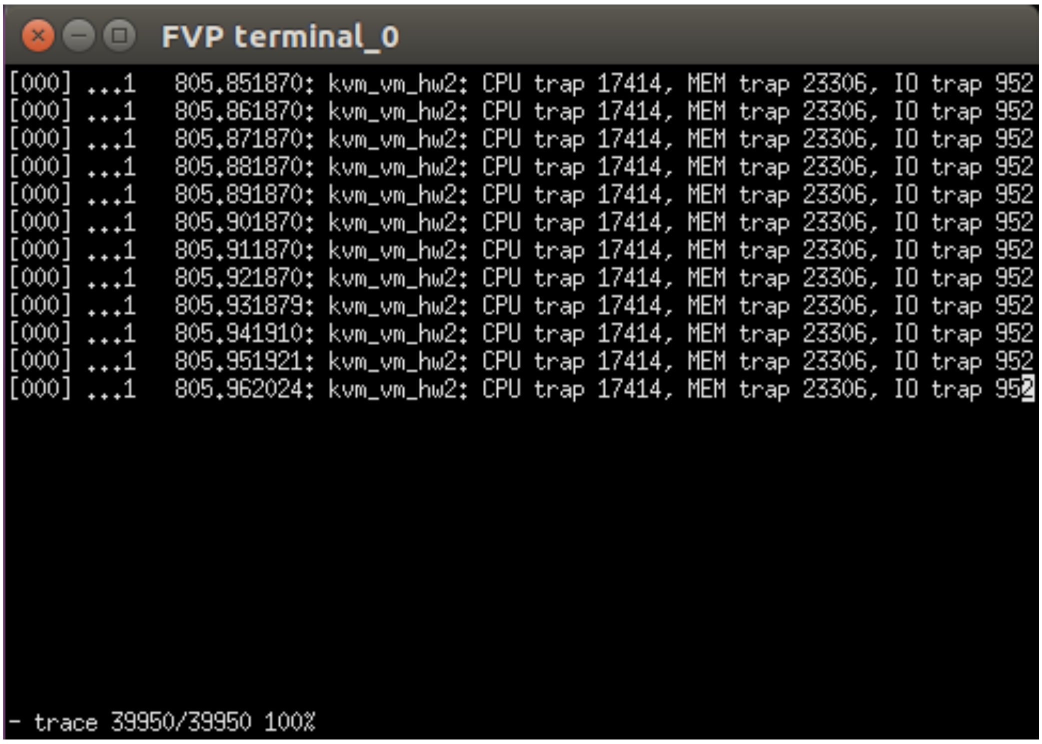
The above picture is the trace right after booting the guest.

- CPU trap comes from calls of interaction that requires the coprocessor (cache maintenance, TLB mainenance...)
- MEM trap comes from for address needed for daemons when boot
- IO trap comes from the output on boot logs on to the screen



The above picture is the trace right after executing `home/root/mem`.

- CPU traps didn't occur
- MEM traps increase because `./mem` calls for memory allocation
- IO trap slightly increases due to typing on the guest screen to execute `./mem`



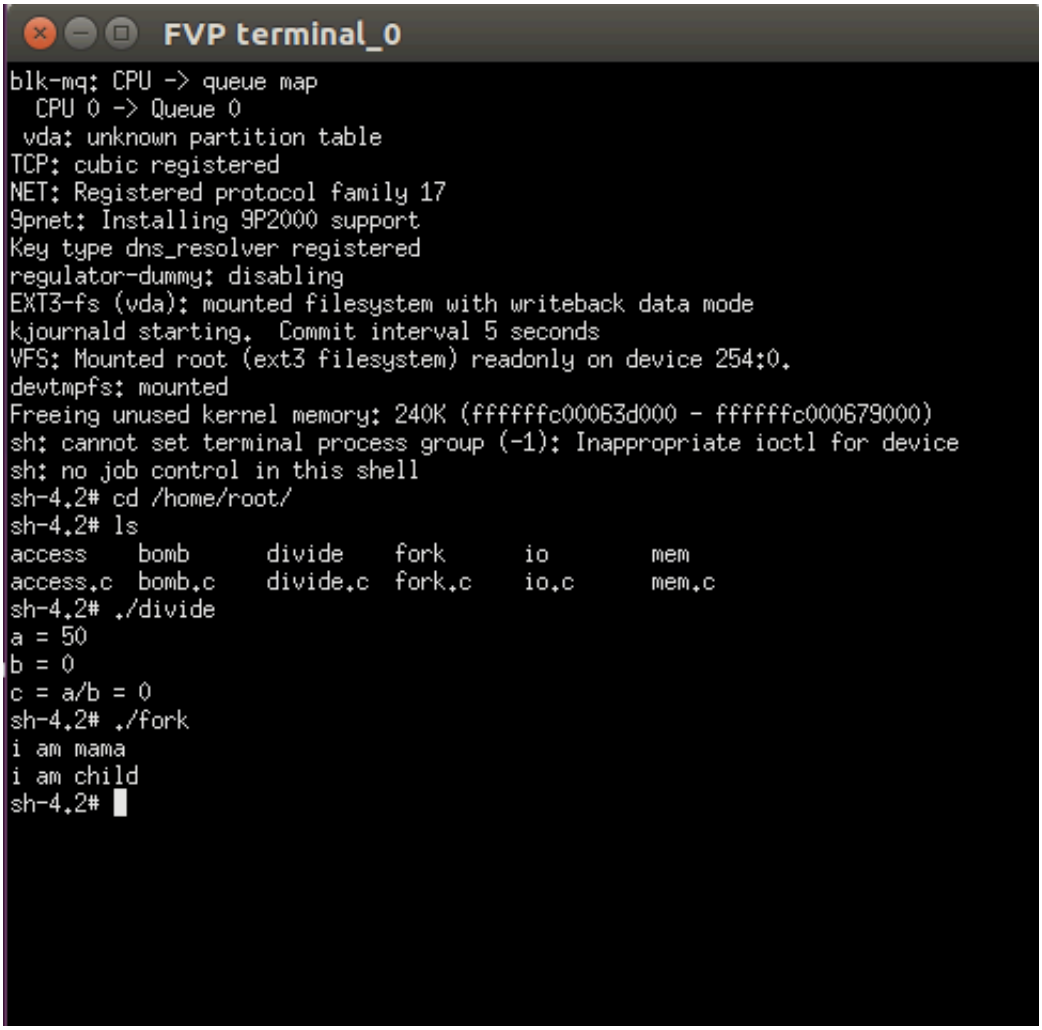
The above picture is the trace right after executing `home/root/io`.

- CPU traps didn't occur
- MEM trap slightly increases because the IPA may not contain everything needed when executing and require some fetch from PA.
- IO trap increases because of the IO calls in `home/root/io`

5 - Other Experiments

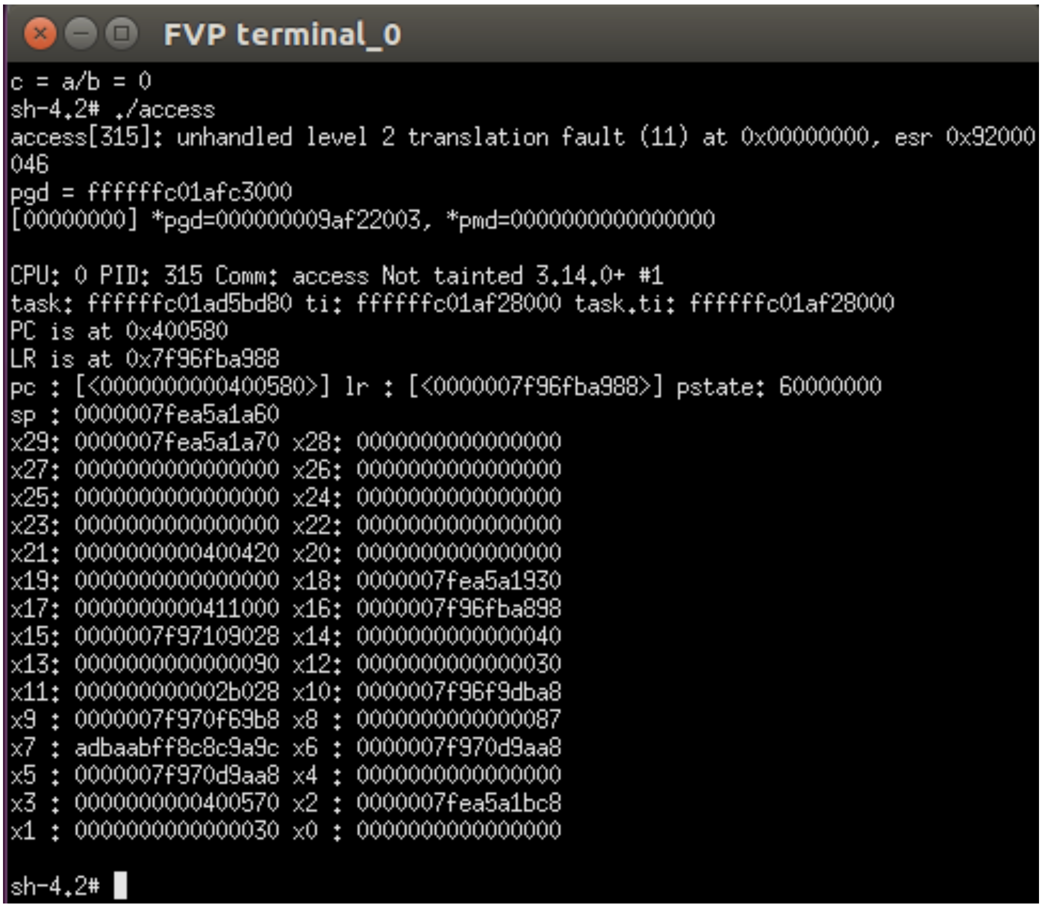
divide by zero (`test_program/divide.c`)

Unlike gcc in linux, aarch64 does not call for a floating point error when divided-by-zero occurs.



access bad memory (`test_program/access.c`)

Accessing bad memory will create an error and MEM trap will increase.



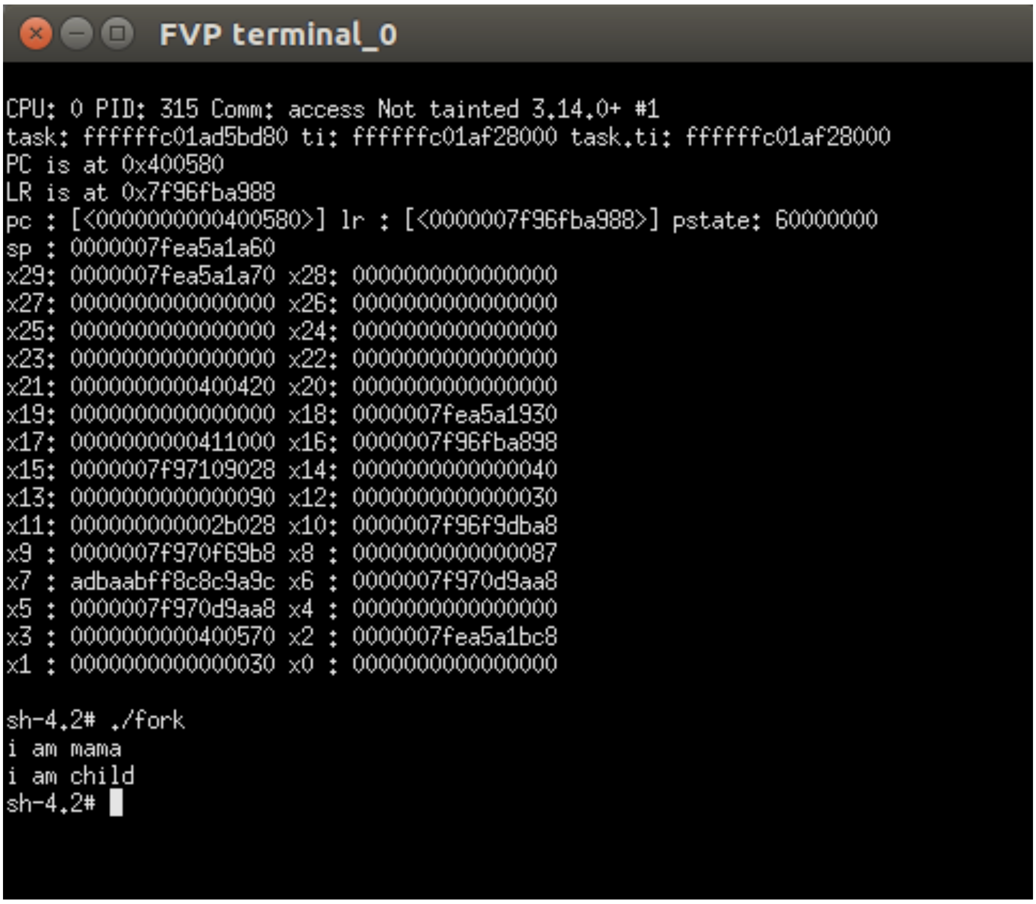
```
c = a/b = 0
sh-4.2# ./access
access[315]: unhandled level 2 translation fault (11) at 0x00000000, esr 0x92000
046
pgd = ffffffc01afc3000
[00000000] *pgd=000000009af22003, *pmd=0000000000000000

CPU: 0 PID: 315 Comm: access Not tainted 3.14.0+ #1
task: ffffffc01ad5bd80 ti: ffffffc01af28000 task.ti: ffffffc01af28000
PC is at 0x400580
LR is at 0x7f96fba988
pc : [<0000000000400580>] lr : [<0000007f96fba988>] pstate: 60000000
sp : 0000007fea5a1a60
x29: 0000007fea5a1a70 x28: 0000000000000000
x27: 0000000000000000 x26: 0000000000000000
x25: 0000000000000000 x24: 0000000000000000
x23: 0000000000000000 x22: 0000000000000000
x21: 0000000000400420 x20: 0000000000000000
x19: 0000000000000000 x18: 0000007fea5a1930
x17: 0000000000411000 x16: 0000007f96fba898
x15: 0000007f97109028 x14: 0000000000000040
x13: 0000000000000090 x12: 0000000000000030
x11: 000000000002b028 x10: 0000007f96f9dba8
x9 : 0000007f970f69b8 x8 : 0000000000000087
x7 : adbaabff8c8c9a9c x6 : 0000007f970d9aa8
x5 : 0000007f970d9aa8 x4 : 0000000000000000
x3 : 0000000000400570 x2 : 0000007fea5a1bc8
x1 : 0000000000000030 x0 : 0000000000000000

sh-4.2#
```

fork (`test_program/fork.c`)

A single fork can be successfully executed.

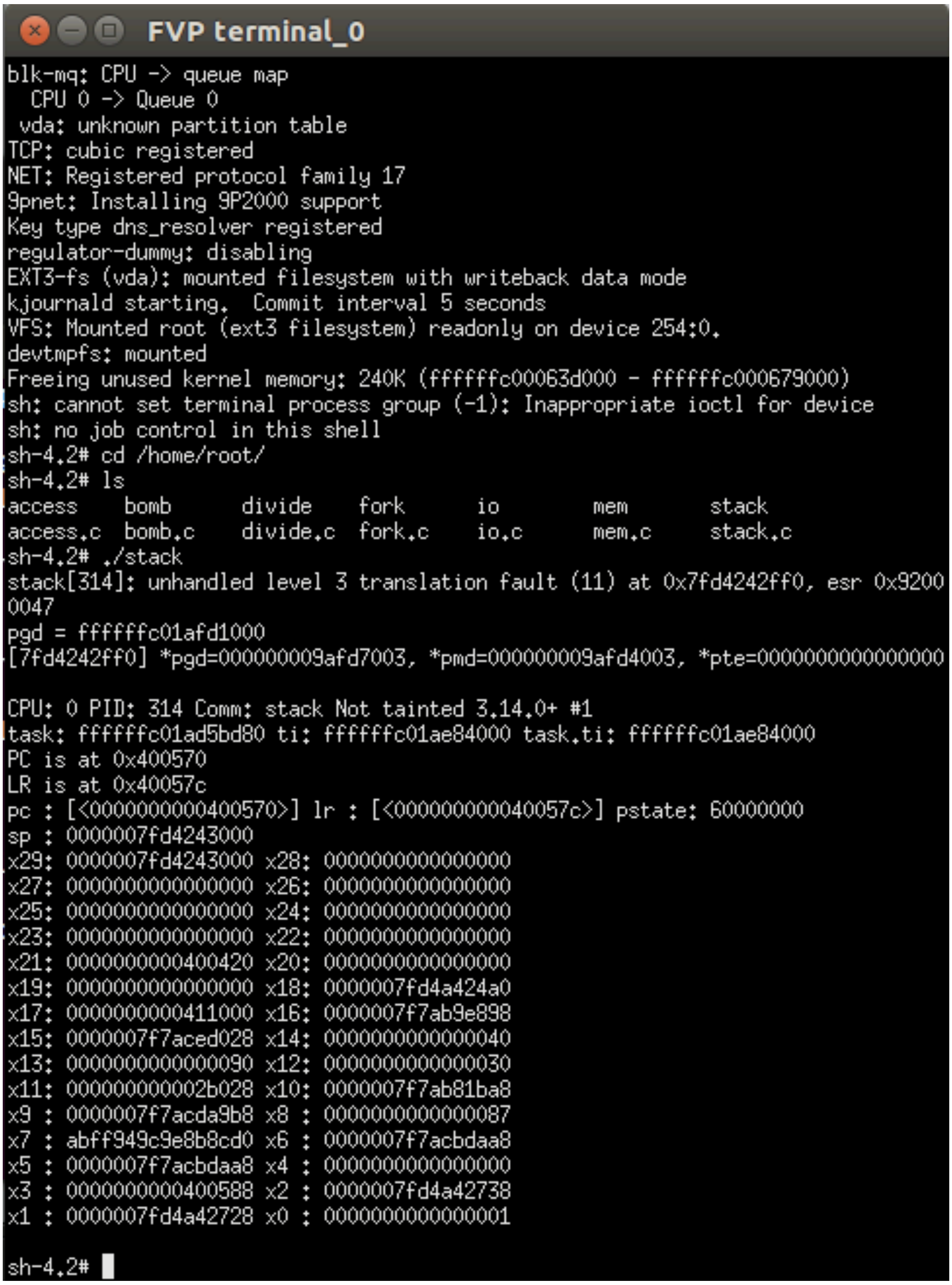


fork bomb (`test_program/bomb.c`)

The hypervisor didn't add limit the numbers of process the guest can create. Simple fork bomb program gradually paralyzes the host too ~ QAQ Making it really slow and have to be reboot.

stack overflow (`test_program/stack.c)

Stack overflow also creates memory trap.



6 - Reference

- developer.arm.com - Exception Model
- developer.arm.com - Memory Management
- developer.arm.com - Armv8-A Virtualization
- 成大資工wiki - XVisor