NOVA system call internals

Michal Sojka

Czech Technical University in Prague,
Email: michal.sojka@cvut.cz

December 19, 2017

Execution context

- In NOVA, execution context (Ec) represents a thread of execution (similar to tasks in other OSes).
- Data stored in the execution context:

```
class Ec {
        void         (*cont)();  // Continuation address
        Exc_regs     regs;       // Registers
        static Ec * current;     // Currently running Ec
};
```

  - Ec::regs stores user space registers (i.e. syscall parameters)
  - Ec::current is a (global) pointer to the currently executing Ec.
- First Ec is created in bootstrap(), init.cc:

```
// Create a new Ec with Ec::root_invoke as entry point
Ec::current = new Ec (Ec::root_invoke, addr);
// Start executing the new "task" (in kernel space)
Ec::current->make_current();
UNREACHED; // This is never executed.
```

Context switch in the kernel

```
void Ec::make_current()
{
    current = this;

    Tss::run.sp0 = reinterpret_cast<mword>(exc_regs() + 1);

    asm volatile ("mov %0, %%esp;"
                  "jmp *%1"
                  :
                  : "g" (KSTCK_ADDR + PAGE_SIZE), "rm" (cont)
                  : "memory");
    UNREACHED;
}
```

1. Remember which Ec is current
2. Set kernel stack pointer (See “CPU initialization” and “Syscall entry” slides)
3. Reset stack and jump to Ec::cont.

Kernel side of system calls

- CPU initialization
- Kernel entry code
- Syscall handler
- Kernel exit code

CPU initialization

- Set Model-Specific Registers (MSR) to tell the CPU what to do when user space invokes the `sysenter` instruction (see `init.cc`, `init()`)

```
Msr::write<mword>(Msr::IA32_SYSENTER_CS,
        SEL_KERN_CODE);
Msr::write<mword>(Msr::IA32_SYSENTER_ESP,
        reinterpret_cast<mword>(&Tss::run.sp0));
Msr::write<mword>(Msr::IA32_SYSENTER_EIP,
        reinterpret_cast<mword>(&entry_sysenter));
```

- CS (code segment) register will be set to kernel code segment
  - Note that code segment descriptor determines the privilege level of executing code.
- ESP (stack pointer) will point to `sp0` member of `Tss::run` global variable (see `tss.h`)
- EIP (instruction pointer) will be set to `entry_sysenter` (see `entry.S`)

Syscall entry

```
entry_sysenter:
  cld
  pop     %esp
  lea     -44(%esp), %esp
  pusha
  mov     $(KSTCK_ADDR + PAGE_SIZE), %esp
  jmp     syscall_handler
```

3. Set ESP to the point behind address of `Ec::current->regs` (see `Ec::make_current()` in `ec.h`).
4. Decrease ESP to skip 11 registers that are used only during exception handling (`Exc_regs`)
5. **Store 8 general purpose registers (syscall arguments) to** `Ec::current->regs`
6. Set ESP to the top of kernel stack
7. Jump to `Ec::syscall_handler`

Syscall implementation

- `Ec::syscall_handler` – A C++ function implementing the syscalls
- Where do we get the `number` argument?

```
void Ec::syscall_handler (uint8 number)
{
    switch(number) {
    case 0: ...
    case 1: ...
    }
    ret_user_sysexit();
    UNREACHED; // Tell the compiler to not generate
               // function epilog
}
```

Returning to user space

```
void Ec::ret_user_sysexit()
{
    asm volatile ("lea %0, %%esp;"
                  "popa;"
                  "sti;"
                  "sysexit"
                  : : "m" (current->regs) : "memory");
    UNREACHED;
}
```

3. Set ESP to point `Ec::current->regs`.
4. **Restore 8 general purpose registers from there.**
5. Enable interrupts.
6. Return to user space.

sysenter/sysexit

- Faster alternative to `int 0x80` and `iret`.
- Does not use stack to store return address.
- `sysexit` sets EIP←EDX, ESP←ECX and decreases the privilege level.
- Therefore the user space syscall wrapper must be different from the "`int 0x80`" variant:

```
unsigned syscall1 (unsigned w0) {
  asm volatile (
    "   mov %%esp,%%ecx;"
    "   mov $1f,%%edx;" // set edx to the addr. of label 1:
    "   sysenter;"
    "1:"                // continue here after sysexit
    : "+a" (w0) : : "ecx", "edx", "memory");
  return w0;
}
```