Lecture 22: Operating System Kernel Case Study

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/16-modular-compilation.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

Here we discuss a last and third use case where using C is appropriate: the implementation of an operating system (OS) kernel.

System Call and Context Switch

Because of its proximity with the hardware and performance, C is heavily used in OS kernels. Here we will study how the **system call** and **context switch** operations are implemented by an OS. We already talked about invoking system calls from user space in the previous lecture regarding the C standard library. Here we'll get an overview of how they are handled on the kernel side. Regarding Context switches, it corresponds to the action of replacing a task running on the CPU by another one. We'll get an overview of how the kernel manages that.

System Calls in a Nutshell

When a user space task is executing on the processor, the CPU state of this task consists in the values present in the processor registers. We have general purpose registers used for computations, but also special-purpose ones such as stack and instruction pointers. The stack pointer points to the stack, a contiguous memory area used to store various things such as local variables. The instruction pointer points to the instruction currently executed by the CPU, located inside the program code in memory.

When the system call instruction is issued (e.g. `syscall` for x86-64), the CPU switches to privileged mode and jumps to a predefined handler in the kernel. It also switches to the kernel stack. The kernel starts by saving the application state by pushing all registers on the kernel stack. Then the kernel determines which system call is invoked (e.g. on x86-64 by looking at the system call identifier in `%rax`) and starts processing it.

When the system call has been processed, the kernel restores the task state by popping all the registers from the kernel stack. And then user space execution resumes at the next instruction following the system call invocation instruction.

Context Switches in a Nutshell

Let's assume a single core CPU, with a task running (scheduled) on the CPU, we'll call it task A. We also assume a task B that is currently sleeping, i.e. it is not scheduled. Each task has its code and data (including its stack) in memory. In memory, we also have the kernel code, as well as a kernel stack for each task A and B.

Context switch should be done by the kernel, so let's assume at some point there is a trap to the kernel, either due to a system call or a hardware interrupt. After the interrupt is treated, in many OSes the scheduler will check if there is a task ready with a higher priority than the one currently running. Let's assume task B is ready, and its

priority is higher than the currently running task A. The scheduler then decides to run B instead of A and a context switch is needed.

The context switch operation starts by saving the state of the task getting scheduled out of the CPU, A, pushing all of its registers' values on its kernel stack. Next the state of task getting scheduled, B, is restored from its own kernel stack, where it was previously saved the last time that task was scheduled out. Then the execution of B can resume, in kernel mode first, then the kernel returns to user space, and B resumes.

System Call and Context Switch Implementations in a Real OS

We have seen in a nutshell how the syscall handling and context switch operation look like. Now let's have a look at some code. We'll study the code of the HermiTux[1] kernel. It is a minimal OS for cloud applications, an adaptation of an existing OS named HermitCore[2]. The good thing about HermiTux is that it is very small and simple which is great for learning purposes.

Interrupts

Recall that **context switch and system call handling are privileged operations that can only be done by the kernel**. An important thing to note is that the only way to enter the kernel is through an **interrupt**:

- Either a **hardware interrupt** such as a tick from the timer, a notification from the network card that a packet has arrived, etc.
- Or a **software interrupt**, also called exception, for example division by 0, or a syscall.

When there is an interrupt, the CPU stops executing user code and traps to the kernel. It is during kernel entry that the state of the task that was interrupted is saved.

Example Scenario

Here we will study what happens in the HermiTux OS code when a task executes the `sched_yield` system call. This system call is made when a task wants to voluntarily relinquish the CPU. This scenario combines a system call and a context switch.

Assume we have task A running. It invokes the `sched_yield` system call, there is a trap to the kernel and the CPU switches from user mode to kernel mode. The system call is processed by the kernel and the scheduler is called to find another task to run, let's call it task B. Then there is a context switch, task A is removed from the CPU and the kernel schedules task B. When returning to user space, there is a switch from kernel to user mode and task B resumes execution.

The Code

Let's have a look in the code. When a syscall is invoked by user code it traps to a predefined entry point int the kernel. This part is in assembly, in HermiTux it corresponds to the `isyscall` label in `arch/x86/kernel/entry.asm`:

```
global isyscall
isyscall:
    cli                 ; disable interrupts
    push rax            ; push the task's user state on its stack (in HermiTux the user land
    push rcx            ; and kernel share the same stack
    push rdx
    push rbx
    ; more push operations here ...


    mov rdi, rsp        ; set pointer at "struct state" as first argument
    sti                 ; enable interrupts
```

[1] https://ssrg-vt.github.io/hermitux/
[2] https://github.com/hermit-os/libhermit

```
    extern syscall_handler
    call syscall_handler
```

The interrupted task's state is saved on its stack, with as all the registers being pushed there. Then the kernel jumps to `syscall_handler`, which is a function implemented in C in `arch/x86/kernel/isrs.c`.

```
void syscall_handler(struct state *s) {
    switch(s->rax) {
        // ... one "case" for each syscall number
        case 24:                          // sched_yield's number is 24
            s->rax = sys_sched_yield();
            break;
        /* ... */
    }
}
```

In `syscall_handler`, the kernel looks at the value of the saved `%rax` register on the stack to determine which system call is invoked, with a large `switch` statement. For sched_yield the syscall number is 24[3]. And then the kernel calls the function implementing that system call: `sys_sched_yield` in `kernel/syscalls/sched_yield.c`. Its return value will overwrite the saved value of `%rax` on the stack, that will be restored once we resume the task execution.

`sys_sched_yield` just calls `check_scheduling` which is in `kernel/tasks.c`:

```
void check_scheduling(void)
{
    /* ... */
    uint32_t prio = get_highest_priority();
    task_t* curr_task = per_core(current_task);
    if (prio > curr_task->prio) {
       reschedule();
    /* ... */
}
```

`check_scheduling` looks for the task ready to run with the highest priority and, if this priority is higher than the currently running task (i.e. the one that invoked the system call), a context switch needs to be performed. In that case `reschedule` is called, and this function calls `switch_context`, which is implemented in assembly in `arch/x86/kernel/entry.asm`:

```
switch_context:
    push rax ; ... start pushing the state of the scheduled out task on the stack
    push rcx
    push rdx
    ; ...
    jmp common_switch
    ; ...
common_switch:
    ; ...
    call get_current_stack      ; get new rsp
    mov rsp, rax                ; switch stack to the new task's one
    ; ...
    pop r15                     ; start restoring the scheduled in task state (reverse order)
    pop r14
    ; ...
    add rsp, 16 ; at that point the stack pointer points to the saved instruction pointer
    iretq ; restore instruction pointer from the stack, i.e. jumps there
```

At that point we are in the context switch logic. We save the kernel state of the task that is scheduled out by pushing all its registers on the stack. Then we switch to the stack of the scheduled in task. We restore its kernel

[3] https://filippo.io/linux-syscall-table/

state (that was saved the last time this task was scheduled out) by popping all the registers and with iretq we jump to the saved instruction pointer, i.e. we resume the task in kernel mode.

We are now executing in the context of the scheduled in task. We take the return path in the kernel, returning from whatever syscall or interrupt triggered the scheduling out of the task in the past. Let's assume it was a syscall:

```
isyscall:
    ; ...
    call syscall_handler
    ; ...
    pop r15   ; start restoring the userland state of the task
    pop r14
    pop r13
    ; ...
    sysret    ; return from syscall handler to userspace
```

We go back to the assembly syscall handling code. Restore all registers that correspond to the user state of the task. And with `sysret` we jump back to user code.