CMPSCI 577 Operating Systems Design and Implementation **Spring 2020**

Lecture 14: 12 March 2020

Lecturer: ***Ahmed Ali-Eldin*** *Scribe:* ***Kanchi Masalia (1st year Masters)***

14.1 System Calls in Linux

It is the interface between the user and kernel. The system calls are compliant with POSIX and SUSV3. All system calls have return type *long*. All system calls have a number associated with them which are architecture dependent. Linux tries to minimize the number of system calls.
There is only slight difference between system calls in Minix and Linux. In Minix each system call has a server which exposed it to the user, it is not so in Linux. *SYSCALL_DEFINE0* is a macro which defines system call where 0 means zero parameters are used to invoke this system call. Each system call has a defined handler.

14.1.1 Mechanism

System calls result in an interrupt which causes an exception and the control is switched to the kernel space. Depending on the parameters it is decided which system call is to be invoked. The system call number is passed to kernel via **eax** register.

14.1.2 Why not implement system call in Linux?

You need a system call number officially assigned to you, which is very difficult to get. You need backward compatibility for them, so once you add you can't remove them. Thus, making it a maintenance issue. Hence, Linux unlike Windows tries to maintain low number of system calls.
Alternative: Linux Kernel Modules

14.2 Interrupts and Interrrupt Handlers

14.2.1 Why interrupts?

Processors can run many instructions in seconds. While communicating with slow hardware, it not ideal for the processor to wait for the response from the hardware.
In order to solve this variable speed problem are 2 ways are:
Polling : Processor will wait for some pre-decided time and then check if the hardware has something that it has to process.
Interrupts

14.2.2 Interrupts

Enables the hardware to signal the processor once it completes the task. Hardware devices generate interrupts asynchronously. Interrupt controller is a chip which holds a mapping table for each signal from the hardware and address of what action is to be taken. This enables the processor to communicate with all types of hardware. Each interrupt has a unique value. (In reality, not all interrupts have a value.) Interrupt values are known as IRQ (Interrrupt Request Lines).

14.2.3 IRQ

Each IRQ line is assigned a unique number. IRQ zero is the timer interrupt IRQ one is the keyboard Interrupt. A specific interrupt is associated with a specific device. Unlike system calls, all IRQ numbers aren't defined. At times, IRQ numbers are dynamically assigned.

14.2.4 Difference between Interrupts and Exceptions

Origin of interrupts is hardware whereas origin of exceptions is software.

14.2.5 Interrupt Handlers

While installing a device driver, we teach it how to handle interrupts. Device drivers can be implemented in Kernel or Linux Kernel Module. They are written in C. They run in special kernel context called interrupt context or atomic context meaning it can not be blocked and thus runs quickly.

14.2.6 Top Halves vs Bottom Halves

Interrupt handling is split in 2 halves. Top Half is for critical work which run immediately and bottom half are for non urgent work.

14.2.7 Registering Device Driver

Each devices has one associated driver. Each driver registers one handler using *request_irq()* which returns 0 on success. Common error is -EBUSY which means the irq is already in use.

14.2.8 Handler Flags

When there is an interrupt, everything is blocked. This is achieved using handler flags.
IRQF_DISABLED : When set it instructs the kernel to disable all interrupts when executing this interrupt handler. When unset, interrupt handlers run with all interrupts except their own enabled. Most commonly, it is unset.
IRQF_TIMER: This is for actual system handler. It indicates something is of very high priority and should run right away.
IRQF_SHARED and dev: Enables multiple devices to share same same interrupt number.

IRQF_SAMPLE_RANDOM: There are no true random number generators. Random numbers are needed for crypto libraries in kernel, etc. So it uses metadata to generate them. This flag is set/unset depending on whether this metadata can be used for random number generator.

14.2.9 Reentrancy and Interrupt Handlers

When an interrupt handler is executing, it can not run on any other processor as well. This is needed to prevent the interrupt from being overridden by another interrupt of same type.

14.2.10 Why interrupt sharing?

Limited number of pins but infinite devices to attach. So we want a way to overbook the pins. This can be achieved by sharing interrupt handler numbers and having hardware and driver support. Hardware should understand it's sharing interrupt handler, the kernel should be able to figure which particular device is raising the interrupt out of all the ones sharing the interrupt handler number.
In the interrupt vector table, the interrupts ranging from 0 - 19 can not to shared. They are non maskable.

14.2.11 Interrupt Control Methods in the kernel

There are ways to mask interrupts on a single processor for synchronization purpose. In multiprocessor system, each processor has it's own interrupt handler. There is no way for disabling interrupts on all cores.

14.3 Bottom Halves

Performs most work. It will run when system is less busy. They are run after the handler, but with all interrupts enabled. If the work is time sensitive, hardware dependent or requires no interruption it must be performed in interrupt handler. Rest all things are part of bottom half.
There are different mechanisms used to implement bottom-half:
1. Task Queues
2. Softirqs and Tasklets
3. Threaded Interrupts

14.3.1 Task Queues

Similar to priority queues. There are many queues, each one contains linked list of functions to call.
Issue : Not light weight
Now replaced with work-queues.

14.3.2 Softirqs and Tasklets

Softirqs are statistically defined Bottom Halves which can run simultaneously on different processors even if they are of same type. They also have a number table. Softirqs offer high performance.
Tasklets are built on top of softirqs. Thus, tasklets are softirqs. Two same type of tasklets can not run on

different processor simultaneously.
Tasklets are for high priority and softirqs are for low priority.

14.3.3 Threaded Interrupts

Instead of implementing top halves and bottom halves, top half is just for acknowledging and clearing hardware. Everything else is executed in kernel thread.

14.3.4 ksoftirqd

If there are too many softirqs it hogs the system. Also, a softirq can raise it self so it runs again and can lead to starvation. To mitigate, ksoftirqd - per processor kernel thread are there. If the number of softirqs grows excessively, the kernel wakes up kernel threads to handle the load with lowest priority.

14.3.5 Work Queues

If the deffered work needs sleep Work Queues are used else softirqs or tasklets are used. Useful for allocating memory, perform block I/O.