Interrupt Handling and Context Switching in Kernel

Efficient multitasking and hardware responsiveness in modern operating systems are enabled by two key mechanisms in the kernel: **interrupt handling** and **context switching**. These are fundamental to how the kernel orchestrates processes, handles external events, and maintains CPU utilization.

2. Interrupt Handling in the Kernel

When an interrupt occurs:

1. **Interrupt Context**:
2. The kernel enters a special context where regular process scheduling is suspended.
3. It saves the state of the current task (registers, stack pointer).
4. **Top Half and Bottom Half**:
5. **Top Half (ISR):** Runs immediately; does minimal work (acknowledges interrupt).
6. **Bottom Half:** Deferred work using mechanisms like:
    - SoftIRQs
    - Tasklets
    - Workqueues

These bottom halves are scheduled to run later when it is safe to sleep or schedule tasks.

4. Context Switching and the Scheduler

The **Linux scheduler** (CFS – Completely Fair Scheduler) decides which task runs next based on priorities and fairness.

Interrupts can:

- **Preempt the current task**, forcing a context switch.
- **Schedule deferred work**, which may also cause a switch.

Involuntary Switch:

Triggered by:

- Timer interrupts (preemption)
- I/O wait
- Kernel yields

Voluntary Switch:

Triggered by:

- Process calling `sleep()`, `wait()`, `sched_yield()`.

Summary

Feature	Interrupt Handling	Context Switching
Trigger	Asynchronous (HW/SW event)	Scheduler decision or blocking call
Purpose	Handle external/ internal events	Switch execution between tasks
Context	Runs in interrupt context	Runs in process/thread context
Latency Critical?	Yes (especially in ISRs)	No (usually)

Feature	Interrupt Handling	Context Switching
Preemptive?	Yes	Yes

Conclusion

Interrupt handling and context switching are core kernel functions that allow responsive, multitasking operating systems. The balance between fast interrupt response and efficient task scheduling is crucial for system performance and reliability.