The scheduler is not common for kernel-space and user-space processes, though they are related.

**1. User-Space Process Scheduling**

* The kernel scheduler manages user-space processes (also called threads or tasks) using scheduling policies like **CFS (Completely Fair Scheduler)**, **RT (Real-Time) Scheduler**, and **Deadline Scheduler**. It decides which process gets CPU time and when to switch between them.

**2. Kernel-Space Process Scheduling**

* The kernel also has its own threads, called **kernel threads (kthreads)**, which run in the kernel space. These threads are scheduled using the same kernel scheduler but with different priority handling. Kernel tasks (such as interrupt handling, workqueues, and softirqs) have their own mechanisms that are separate from regular process scheduling.

**Key Differences:**

|  |  |  |
| --- | --- | --- |
| **Feature** | **User-Space Scheduling** | **Kernel-Space Scheduling** |
| **Context Switch** | Full process switch (includes MMU, registers) | Lightweight, may not involve full MMU switch |
| **Preemption** | Can be preempted by a higher-priority task | Kernel threads can be preempted if preemption is enabled |
| **Priority** | Uses nice values and scheduler classes | Uses static kernel priorities |
| **Execution Mode** | Runs in user mode (ring 3) | Runs in kernel mode (ring 0) |

In a uniprocessor system, only **one process or thread** runs at any given moment because there is only one CPU core. However, the kernel **time-shares** the CPU among multiple processes using **context switching**, creating the illusion of parallel execution.

A user-space process can block kernel-space processes in certain conditions:

1. **Blocking Due to CPU Scheduling:**
   * user-space process can occupy the CPU for a long time (if it has a high priority) and **delay kernel threads** from executing. However, modern schedulers ensure fair CPU distribution, and **kernel threads usually have higher priority** than normal user-space tasks.
2. **Blocking Due to System Calls:**
   * A user-space process making a **blocking system call** (e.g., read(), write(), sleep(), wait())puts itself into a waiting state. While waiting, the kernel can schedule other tasks, including kernel threads.
3. **Blocking Due to Resource Contention:**
   * If a user-space process holds a mutex that a kernel thread is waiting for, the kernel thread cannot proceed until the user-space process releases it.
4. **Interrupts and Preemption:**
   * In preemptive kernels, a **kernel thread (or interrupt handler) can preempt a running user-space process** at any time.

When a **user-space process makes a system call**, the process itself does **not** get elevated privileges. Instead, the following happens:

**1. Mode Switch (User Mode → Kernel Mode)**

* The user-space process executes a system call (e.g., read(), write(), open()). This triggers a **trap** (software interrupt), causing the CPU to switch from **user mode (ring 3)** to **kernel mode (ring 0)**.
* The currently running process remains the same, but now it is executing in **kernel mode** instead of user mode. The process is now in **process context** within the kernel.

**2. Execution in Process Context**

* The system call executes within the context of the same user-space process (not a new thread).
* The kernel executes the system call **on behalf of the calling process** using the same kernel stack allocated for that process.
* The kernel can **access and modify user-space memory** (with proper validation) to return results.
* **the kernel does not create a new thread** for each system call. The system call runs in the same process context but in **privileged mode (kernel mode)**. However, if the system call needs to perform background work (like I/O), the kernel may:
  + **Use worker threads** (e.g., kernel threads handling async tasks).
  + **Schedule interrupts or workqueues** to complete the request asynchronously.

**4. Returning to User Mode**

* Once the system call is complete, the kernel restores the **previous CPU state** (registers, stack, etc.). The CPU switches back to **user mode**, and execution continues in the user-space process.

**Key Takeaways**

|  |  |
| --- | --- |
| **Aspect** | **What Happens?** |
| **Does user-space process get elevated privilege?** | No, it remains in user mode; only the CPU switches to kernel mode. |
| **Does the kernel create a new thread?** | No, the system call runs in the same process context. |
| **Where does the system call execute?** | In the **kernel** within the context of the calling process. |
| **How does execution return to user mode?** | The kernel restores process state and switches the CPU back to user mode. |

In ARM architecture, raising a **software interrupt (softirq)** is typically done using the **Supervisor Call (SVC) instruction** (previously known as SWI in older ARM versions). However, in Linux, **softirqs are not triggered using an assembly instruction directly**; they are typically scheduled within the kernel.