# Context Switching: The Role of The Kernel and Scheduler

Context switching is a critical function of a multitasking operating system. It allows the CPU to switch between tasks (or threads), ensuring that each task gets its fair share of computing time and that high-priority tasks are serviced in a timely manner. The context is the computational state that must be saved and restored when switching between tasks.

**- Start of Context Switch**

***Trigger***: A context switch begins when a trigger occurs, such as a timer interrupt signaling the end of a time slice, an I/O operation completing, a system call being made, a multithreading directive, or a hardware exception like a page fault.

**- Saving the Current Context**

*State Preservation*: The operating system pauses the current task and saves its state, including CPU registers, the program counter, memory management information, and other process-specific data.

**- Choosing the Next Task**

*Scheduler Activation*: The scheduler then evaluates which task to run next based on factors like priority, fairness, and scheduling policy.

**- Loading the New Context**

*State Restoration*: The CPU registers and program counter are updated with the saved state of the chosen task. Memory access rights and mappings are adjusted accordingly.

**- Execution of the New Task**

*Resumption*: The new task begins execution where it last left off, continuing until it is either completed or another context switch is triggered.

*The role of Scheduler During Context Switching:*

The scheduler assesses all ready-to-run processes in the system, which are typically kept in a queue or set of queues, and decides which one should be executed next. This decision is based on a combination of factors:

1. **Priority**: Processes are often assigned a priority level. The scheduler might choose the highest-priority process to run next. Some systems use dynamic priorities that can change over time to prevent starvation of low-priority processes.
2. **Fairness**: The scheduler ensures that each process gets a fair share of CPU time. This can be achieved through algorithms like round-robin, where each process gets to run for a fixed time slice in a cyclic order.
3. **Scheduling Policy**: Different policies dictate how the scheduler makes its decision. Common policies include:
   - **First-Come, First-Served (FCFS)**: Processes are scheduled according to their arrival time.
   - **Shortest Job First (SJF)**: Processes with the shortest estimated run time are chosen first.
   - **Multilevel Queue Scheduling**: Processes are divided into different queues based on their priority or type, and each queue has its own scheduling algorithm.
4. **Load Balancing (in Multi-core Systems)**: The scheduler may also consider the load on different CPU cores and attempt to balance it by assigning processes to less busy cores.

Once the scheduler selects a process, the context switch mechanism saves the state of the currently running process and loads the state of the selected process. The state includes register contents, program counter, and other necessary data to resume execution.

***The role of Kernel During Context Switching:***

1. **Task Management**: The kernel keeps track of all processes in the system, including their state and context information.
2. **Scheduler Invocation**: The kernel includes the scheduler, a subsystem that determines which process runs at any given time.
3. **Context Saving**: When the scheduler decides to switch the CPU from one process to another, the kernel saves the context (state) of the current process. This context includes the CPU register contents, the program counter, and process-specific information like open files and memory allocation.
4. **Process Selection**: Using the scheduling policy, the kernel's scheduler selects the next process to run. This can be based on:
    - **Priority**: The kernel may prioritize processes based on a static or dynamic priority level.
    - **Fairness**: The kernel's scheduler employs algorithms to ensure fair CPU time distribution, like round-robin scheduling.
    - **Scheduling Policy**: The kernel implements various scheduling algorithms to decide which process runs next.
    - **Load Balancing**: In multi-core systems, the kernel's scheduler will distribute processes across cores to balance the load effectively.
5. **Context Loading**: The kernel then loads the context of the chosen process, updating the CPU registers and program counter so that this process can resume execution.
6. **Process Execution**: The kernel transfers control to the selected process, which continues executing from where it left off.
7. **System Calls and Interrupt Handling**: The kernel is responsible for handling system calls from processes (e.g., for file access, network communication) and interrupts (from hardware or timers), which can also trigger context switches.
8. **Multi-threading and Concurrency**: The kernel manages the concurrency mechanisms, ensuring that threads are scheduled correctly and that context switching between threads happens efficiently.

The kernel is integral to managing the system's responsiveness and stability by overseeing how processes share the CPU, how they're prioritized, and how resources are allocated and protected. It ensures that the system's scheduling and context-switching mechanisms are fair, efficient, and capable of handling multiple tasks simultaneously without conflicts.

**What’s the main difference between the Kernel and Scheduler in this context?**

The kernel is the core part of an operating system. It acts as an intermediary between the computer's hardware and the applications running on it. The **kernel** has several responsibilities:

- Managing system resources such as the CPU, memory, and I/O devices.
- Providing system services to applications, such as process scheduling, memory management, and handling system calls.
- Ensuring security and access rights are enforced.
- Handling interrupts and exceptions.

The **scheduler** is a specific component within the kernel responsible for process scheduling. Its role is to:

- Decide which process or thread should run at any given moment.
- Implement scheduling algorithms (like round-robin, priority-based, etc.) to manage the execution order of processes.
- Ensure efficient CPU utilization by balancing the CPU's workload.
- Handle context switching between processes.

**Understanding Context switching with a Psudo Code:**

```
// Define a procedure to handle the signal
Procedure SignalHandler(signalNumber)
    Print "Interrupt received: Signal " + signalNumber + ". Pretend we're handling an interrupt."
    // After handling the interrupt, control will return to the main flow
EndProcedure

// Start of the main program
Begin
    // Register the SignalHandler as the handler for the SIGALRM signal
    RegisterSignalHandler(SIGALRM, SignalHandler)

    // Schedule an alarm to send SIGALRM after 5 seconds
    SetAlarm(5)

    Print "Main program doing work and waiting for an interrupt."

    // Main program loop - continues indefinitely
    While true
        // Simulate doing some work
        Sleep(1) // Sleep to simulate work and to yield the processor
        Print "Working..."
    EndWhile
End
```

When you run this program, after 5 seconds, the **alarm** system call will cause the operating system to send a **SIGALRM** signal to the process, which acts as an "interrupt". Here's how the context switching process would work once the signal is delivered:

1. **Pre-Interrupt State**: The main program is running, doing work in a loop. At this point, the CPU context is entirely dedicated to executing the main program's instructions.
2. **Signal Delivery (Interrupt)**: After 5 seconds, the **alarm** expires, and the kernel generates a **SIGALRM**. The signal delivery acts as an interrupt; it causes the CPU to immediately cease execution of the main program.
3. **Context Saving**: Before the signal handler (**signal_handler()**) is executed, the system saves the context of the main program, including the CPU registers and program counter. This information is typically stored on the stack.
4. **Handler Execution (Interrupt Service Routine)**: The CPU then loads the context required to run the signal handler (**signal_handler()**). This includes setting the program counter to the address of the signal handler function and preparing the stack and registers for its execution.
5. **Context Switching**: At this point, the context switch is complete. The CPU is now executing the signal handler code instead of the main program.
6. **Handler Completion**: Once the signal handler completes, the system will restore the previously saved context of the main program.
7. **Post-Interrupt State**: With the main program's context restored, the CPU resumes execution of the main program right where it left off, as if the interruption had never occurred.

During this whole process, the operating system's kernel manages the context saving and restoring, ensuring that the main program and the signal handler have the illusion of exclusive access to the CPU's resources. The kernel also takes care of switching back to the main program's context after the signal handler completes.

In multi-core processors, context switching also includes considerations for core affinity and load balancing across cores. The operating system may try to keep a process on the same core to maximize cache warmth, but it might also need to move processes between cores to balance the load.

***~~~ A Short Article By Yash ~~~***