**Context Switching in Operating System**

* Context Switching in an operating system is a critical function that allows the CPU to efficiently manage multiple processes.
* By saving the state of a currently active process and loading the state of another, the system can handle various tasks simultaneously without losing progress.
* This switching mechanism ensures optimal use of the CPU, enhancing the system's ability to perform multitasking effectively.

**Need for Context Switching**?

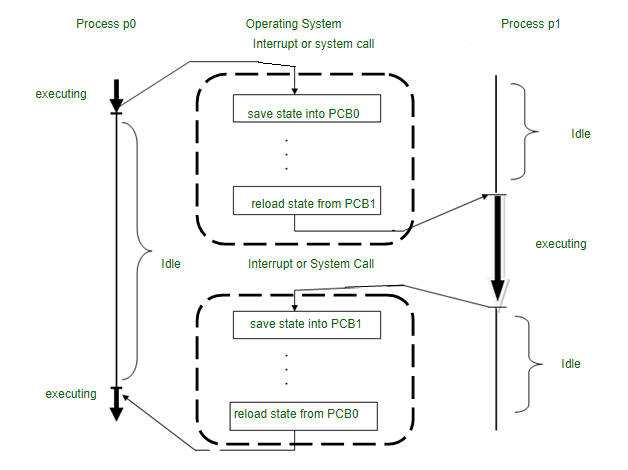
* **Efficient CPU Sharing**: Context switching allows multiple processes to share a single CPU, enabling multitasking without additional hardware.
* **Save and Resume Process State**: It saves the current state of a process so it can resume later from the exact same point without losing data or progress.
* **Managed by Operating System**: The OS controls the switching between processes, ensuring smooth and fair CPU usage—processes don’t directly switch themselves.
* **Supports Multitasking**: Even with one CPU, multiple processes can run “in parallel” by quickly switching between them, making the system responsive.
* **Reduces Need for Multiple CPUs**: With context switching, a single processor can handle many tasks, eliminating the need for extra processors to run multiple applications.

**Context Switching Triggers – 3 Main Types**

1. **Interrupts**
   * When the CPU is interrupted (e.g., by hardware like a disk), it needs to switch tasks to handle the event.
   * The OS saves the current process and switches to the interrupt handler.
2. **Multitasking**
   * To allow multiple processes to run on a single CPU, the OS switches between them.
   * It saves the current process state and loads another, enabling multitasking.
3. **User/Kernel Mode Switch**
   * When a process needs a service from the OS (like I/O), it switches from **user mode to kernel mode**.
   * The OS switches context to safely execute system-level operations.

**Working of Context Switching (Step-by-Step)**

1. **Save Current Process State**: The OS saves the current process’s state (like register values, program counter, etc.) into its **Process Control Block (PCB)**.
2. **Store Process Information**: The saved information includes system-level details (like credentials, priority, etc.) which are stored in **kernel memory** or a special OS file.
3. **Prepare PCB Handle**: A **handle (pointer or reference)** is created/updated so the OS can quickly locate and manage the process again when needed.
4. **Select Next Process to Run**: The OS **stops (aborts)** the current process and selects the **next process** to run from the **ready queue**, based on **priority or scheduling algorithm**.
5. **Load New Process State**: The selected process’s **program counter** and saved CPU state are **loaded** from its PCB.
6. **Resume Process Execution**: The CPU now **starts executing** the new process from where it last stopped.



**Priority Scheduling in Operating System**

* Priority scheduling is one of the most common scheduling algorithms used by the operating system to schedule processes based on their priority.
* Each process is assigned a priority value based on criteria such as memory requirements, time requirements, other resource needs, or the ratio of average I/O to average CPU burst time.
* The process with the highest priority is selected for execution first. If there are multiple processes sharing the same priority, they are scheduled in the order they arrived, following a First-Come, First-Served approach.

**Non-Preemptive Priority Scheduling**

In Non-Preemptive Priority Scheduling, the CPU is not taken away from the running process. Even if a higher-priority process arrives, the currently running process will complete first.

Ex: A high-priority process must wait until the currently running process finishes.

**Example of Non-Preemptive Priority Scheduling:**

Consider the following table of arrival time and burst time for three processes P1, P2 and P3:

Note: Lower number represents higher priority.

| **Process** | **Arrival Time** | **Burst Time** | **Priority** |
| --- | --- | --- | --- |
| **P1** | 0 | 4 | 2 |
| **P2** | 1 | 2 | 1 |
| **P3** | 2 | 6 | 3 |

**Step-by-Step Execution:**

* **At Time 0:** Only P1 has arrived.P1 starts execution as it is the only available process, and it will continue executing till t = 4 because it is a non-preemptive approach.
* **At Time 4:**P1 finishes execution. Both P2 and P3 have arrived. Since P2 has the highest priority (Priority 1), it is selected next.
* **At Time 6:**P2 finishes execution. The only remaining process is P3, so it starts execution.
* **At Time 12:** P3 finishes execution.

**Preemptive Priority Scheduling**

In **Preemptive Priority Scheduling**, the CPU can be taken away from the currently running process if a new process with a higher priority arrives.

Ex: A low-priority process is running, and a high-priority process arrives; the CPU immediately switches to the high-priority process.

**Example of Preemptive Priority Scheduling (Same Arrival Time)**

Consider the following table of arrival time and burst time for three processes P1, P2 and P3:

Note: Higher number represents higher priority.

| **Process** | **Arrival Time** | **Burst Time** | **Priority** |
| --- | --- | --- | --- |
| **P1** | 0 | 7 | 2 |
| **P2** | 0 | 4 | 1 |
| **P3** | 0 | 6 | 3 |

**Step-by-Step Execution:**

* **At Time 0:** All processes arrive at the same time.**P3** has the highest priority (Priority 3), so it starts execution.
* **At Time 6: P3** completes execution. Among the remaining processes, **P1** (Priority 2) has a higher priority than **P2**, so **P1** starts execution.
* **At Time 13: P1** completes execution. The only remaining process is **P2** (Priority 1), so it starts execution.
* **At Time 17: P2** completes execution. All processes are now finished.

**Example of Preemptive Priority Scheduling (Different Arrival Time)**

Consider the following table of arrival time and burst time for three processes P1, P2 and P3:

| **Process** | **Arrival Time** | **Burst Time** | **Priority** |
| --- | --- | --- | --- |
| **P1** | 0 | 6 | 2 |
| **P2** | 1 | 4 | 3 |
| **P3** | 2 | 5 | 1 |

**Step-by-Step Execution:**

* **At Time 0:** Only P1 has arrived, so it starts execution.
* **At Time 1:**P2 arrives with a higher priority (Priority 3) than P1.P1 is preempted, and P2 starts execution.
* **At Time 5:**P2 completes execution. Both P1 and P3 are available. P1 has the higher priority (Priority 2), so it starts execution.
* **At Time 10:**P1 completes execution. P3 resumes execution to finish its remaining burst time.
* **At Time 15:**P3 completes execution. All processes are now finished.

**Earliest Deadline First (EDF) Scheduling**

**Overview:**

Earliest Deadline First is a **dynamic scheduling algorithm** commonly used in real-time systems. In EDF, the scheduler assigns the **highest priority to the process with the closest (earliest) deadline**. As deadlines change over time, so do the task priorities. This makes EDF **adaptive and optimal** for many real-time scenarios.

**Key Characteristics:**

* **Dynamic Priority**: Task priorities are not fixed. They are recalculated as deadlines approach.
* **Real-Time Friendly**: Suitable for both **hard** and **soft real-time systems**.
* **Preemptive**: If a task with an earlier deadline arrives, it can **interrupt** (preempt) a running task.

**Working:**

1. Each task specifies its **deadline** and **execution time**.
2. The scheduler always picks the task with the **earliest deadline** to run first.
3. If a new task arrives with a **sooner deadline**, it immediately preempts the running task.
4. The system tries to meet all deadlines by always favoring the most urgent tasks.

**Example:**

Imagine three tasks:

|  |  |  |
| --- | --- | --- |
| **Task** | **Execution Time** | **Deadline** |
| A | 2 ms | 5 ms |
| B | 1 ms | 3 ms |
| C | 3 ms | 8 ms |

Even if Task A started first, Task B will preempt it when it arrives because it has an earlier deadline.

**Advantages:**

* **Maximizes CPU Utilization**: Can utilize up to 100% of CPU without missing deadlines (under ideal conditions).
* **Optimal for uniprocessor systems**: If any scheduling can meet all deadlines, EDF will.
* **Supports both periodic and aperiodic tasks**.

**Disadvantages:**

* **Overhead**: Due to constant priority recalculations and preemptions.
* **Unpredictable behavior** in overloaded situations; system may crash or behave inconsistently if tasks start missing deadlines.
* Harder to implement and **analyze for safety-critical systems** compared to static scheduling.

**Rate Monotonic Scheduling (RMS)**

**Overview:**

Rate Monotonic Scheduling is a **static priority preemptive scheduling algorithm** used primarily in **hard real-time systems**. It assigns fixed priorities based on task frequency: the **shorter the period of a task, the higher its priority**.

**Key Characteristics:**

* **Static Priority**: Priorities are fixed at compile time and do not change.
* **Period-Based**: Task with the **shortest period** (i.e., most frequent execution) gets the highest priority.
* **Suitable for Periodic Tasks**: Best for systems where tasks repeat at regular intervals.

**Working:**

1. All tasks are assumed to be **periodic**, with known execution times and periods.
2. The system assigns priorities based on their **periods**.
3. At runtime, the task with the **highest priority** that is ready to run gets the CPU.
4. If a high-priority task becomes ready while a lower-priority task is running, it preempts the running task.

**Example:**

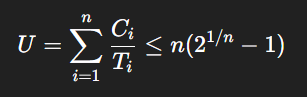
Three tasks:

|  |  |  |  |
| --- | --- | --- | --- |
| **Task** | **Execution Time** | **Period** | **Priority** |
| T1 | 1 ms | 4 ms | Highest |
| T2 | 2 ms | 8 ms | Medium |
| T3 | 1 ms | 16 ms | Lowest |

Since T1 runs most frequently, it is given the highest priority.

**Processor Utilization Bound:**

RMS guarantees all deadlines are met only if CPU utilization is within the bound:



For large n, this approaches **69.3%**. Beyond this, RMS cannot guarantee deadline satisfaction.

**Advantages:**

* **Predictable**: Easy to analyze and test for correctness.
* **Simple to implement** in embedded and real-time systems.
* **Reliable** for critical systems where timing guarantees are essential.

**Disadvantages:**

* **Not optimal**: Can't use 100% of CPU. Limited to 69.3% utilization (for many tasks).
* Not suitable for **aperiodic or sporadic tasks**.
* **Lower-priority tasks may suffer starvation**.

**Comparison Table: EDF vs RMS**

|  |  |  |
| --- | --- | --- |
| **Feature** | **EDF** | **RMS** |
| Priority Type | Dynamic | Static |
| Basis of Priority | Deadline | Task Period |
| Supports Aperiodic Tasks | Yes | No |
| CPU Utilization (Guaranteed) | Up to 100% | Up to 69.3% |
| Implementation Complexity | High | Moderate |
| Predictability | Lower | Higher |
| Usage | Soft and hard real-time | Hard real-time only |

**When to Use**

* **EDF** is ideal for systems where:
  + Task deadlines change or are dynamic.
  + High CPU utilization is required.
  + Tasks may arrive irregularly.
* **RMS** is ideal for systems where:
  + All tasks are periodic and known in advance.
  + Timing predictability and safety are more important than utilization.
  + The system is resource-constrained (e.g., microcontrollers in embedded systems).