Process Scheduling :
Linux uses process scheduling to switch between multiple processes quickly, achieving apparent simultaneous execution. This involves choosing when and which process to run.
Scheduling Policy:
Linux scheduling balances objectives like fast response time, good throughput, and avoiding starvation. It's based on time-sharing, dividing CPU time into slices for each process.

Scheduling Policy:
- Linux ranks processes by priority and adjusts them dynamically based on their CPU usage.
- Processes are categorized as "I/O-bound" (waiting for I/O) or "CPU-bound" (needing a lot of CPU time).
Linux's scheduling ensures efficient use of CPU resources for various types of processes.

Processes in Linux can be classified into three categories based on their behavior:
1. **Interactive Processes:** These constantly interact with users, waiting for input like keypresses and mouse actions. To maintain responsiveness, the average delay should be between 50 and 150 ms with limited variance. Examples include command shells, text editors, and graphical applications.

2. **Batch Processes:** These run in the background without user interaction. They don't require immediate responsiveness and are often lower-priority. Examples include programming language compilers, database search engines, and scientific computations.

3. **Real-Time Processes:** Real-time processes have strict scheduling requirements. They should never be blocked by lower-priority processes, need a quick response time, and minimal variation in response time. Examples include video and sound applications, robot controllers, and sensor data collection.

While these classifications are somewhat independent, Linux prioritizes I/O-bound processes to ensure responsiveness for interactive applications. It explicitly recognizes real-time processes but doesn't distinguish between interactive and batch processes in scheduling.

Process Preemtation
In Linux, processes are preemptive, meaning they can be interrupted and switched out for another process in the TASK_RUNNING state with higher priority or when their time quantum expires. This ensures responsiveness and efficient use of CPU time.
- Preemption occurs when a higher-priority process becomes runnable.
- For example, if an interactive text editor has a higher priority than a compiler, it can be suspended during pauses but quickly resumed when the user presses a key.
- Preempted processes remain in TASK_RUNNING state but don't use the CPU.
Linux's kernel is not preemptive in Kernel Mode, simplifying synchronization. Preemption only happens in User Mode.

The duration of a quantum (time slice) is crucial:
- Too short quantum leads to high overhead from frequent task switches.
- Too long quantum can make processes seem non-concurrent, but it doesn't typically affect interactive applications' response time due to their higher priority.
- However, excessively long quantum can make the system appear unresponsive, especially when a CPU-bound process delays an interactive one.

The Scheduling Algorithm
The Linux scheduling algorithm divides CPU time into epochs, each with a specific time quantum for every process. Here's how it works:

1. **Time Quantum and Epochs:** Within each epoch, each process gets a time quantum, which is the maximum CPU time it can use. Different processes can have different quantum durations. When a process exhausts its quantum, it gets preempted, and another runnable process is chosen.

2. **Repeating Epochs:** A process can be selected multiple times in one epoch if it doesn't use up its entire quantum. The epoch ends when all runnable processes have used up their quantum. Then, the scheduler recalculates quantum durations for all processes, and a new epoch begins.

3. **Base Time Quantum:** Each process has a base time quantum, which is assigned by the scheduler when it exhausts its quantum in the previous epoch. Users can modify this base quantum using system calls like nice() and setpriority(). New processes inherit their parent's base quantum.

4. **Priority Types:** There are two types of priorities:
   - **Static Priority:** Assigned by users to real-time processes, ranging from 1 to 99. It remains fixed and is not changed by the scheduler.
   - **Dynamic Priority:** Applies to conventional processes. It's the sum of the base time quantum (base priority) and the remaining CPU time ticks in the current epoch.

5. **Priority Hierarchy:** Real-time processes always have higher priority than conventional ones. The scheduler runs conventional processes only when no real-time process is in a TASK_RUNNING state.

Data Structures Used by the Scheduler
The Linux scheduler uses various data structures and fields to manage processes and decide which one to run next. Here's an overview of these data structures and fields used by the scheduler:

1. **Process Lists:** Processes are linked together using process descriptors, and runnable processes are linked in a runqueue list. The `init_task` process descriptor acts as a list header.

2. **Fields in Process Descriptors:** Each process descriptor includes several fields related to scheduling:

   - `need_resched`: A flag checked to decide if scheduling needs to occur.
   - `policy`: Specifies the scheduling class (SCHED_FIFO, SCHED_RR, SCHED_OTHER).
   - `rt_priority`: The static priority of real-time processes.
   - `priority`: The base time quantum or priority of the process.
   - `counter`: The number of ticks of CPU time left before the quantum expires.

3. **Base Time Quantum:** Processes have a base time quantum, inherited from their parent or modified using system calls like `nice()` and `setpriority()`.

4. **Scheduling Priority:** Processes have static (real-time) or dynamic (conventional) priorities. Real-time processes always have higher priority.

5. **Scheduling Algorithm Actions:**

- **Direct Invocation:** The scheduler is invoked directly when a process must be blocked immediately due to resource unavailability.
- **Lazy Invocation:** The scheduler can be invoked lazily by setting the `need_resched` field of the current process.

6. **Schedule Function:** The `schedule()` function is responsible for selecting the next process to run based on priority and other factors.

7. **Actions Performed by `schedule()`:**
- It runs functions left by other kernel control paths in various queues.
- It executes active unmasked bottom halves.
- It selects the next process to run by comparing priorities.
- If the previous process used up its quantum, it assigns a new quantum and repositions it in the runqueue.
- If the previous process had pending signals and was in an interruptible state, it wakes it up.
- If the previous process is not in the TASK_RUNNING state, it removes it from the runqueue.
- If no suitable process is found, a new epoch begins with fresh quantum assignments for all processes.

8. **Context Switch:** If a different process is selected, a process switch occurs, and the `context_swtch` statistic is updated.

Overall, the scheduler manages processes' quantum durations, priorities, and selects the next process to run based on a set of rules and criteria.

System Calls Related to Real-Time Processes

The provided text describes several system calls related to real-time processes and their scheduling policies in Linux. Here's a summary of these system calls:

1. **sched_getscheduler():**
- This system call queries the scheduling policy currently applied to the specified process (identified by its PID).
- If the PID is set to 0, it retrieves the policy of the calling process.
- Returns the scheduling policy (SCHED_FIFO, SCHED_RR, or SCHED_OTHER).

2. **sched_setscheduler():**
- This system call sets both the scheduling policy and the associated parameters for the specified process (identified by its PID).
- If the PID is set to 0, it sets the scheduling parameters of the calling process.
- It checks the validity of the policy and static priority specified and the permissions of the caller (requires CAP_SYS_NICE capability or superuser rights).
- Sets the policy and real-time priority fields of the process descriptor.
- If the process is currently runnable, it may need to be moved in the runqueue.
- It sets `need_resched` for the current process, ensuring the scheduler is invoked.

3. **sched_yield():**
- This system call allows a process to voluntarily relinquish the CPU without being suspended.
- The process remains in a TASK_RUNNING state but is placed at the end of the runqueue list, giving other processes with the same dynamic priority a chance to run.
- Mainly used by SCHED_FIFO processes.

- Sets the SCHED_YIELD flag in the policy field if the process is SCHED_OTHER.
- Sets `need_resched` for the current process.
- Moves the process to the end of the runqueue list.

4. **sched_get_priority_min() and sched_get_priority_max():**
- These system calls return the minimum and maximum real-time static priority values that can be used with the specified scheduling policy.
- Always returns 1 for minimum priority if the current process is a real-time process.
- Always returns 99 for maximum priority if the current process is a real-time process.

5. **sched_rr_get_interval():**
- This system call is supposed to retrieve the round-robin time quantum for a named real-time process.
- However, it does not operate as expected and always returns a 150-millisecond value in the timespec structure.
- This system call is effectively unimplemented in Linux.

These system calls provide ways for processes to query and modify their scheduling policies and priorities, as well as to yield the CPU voluntarily when needed.