Supporting Time-Sensitive Applications on a Commodity OS (implementation of time Sensitive linux (TSL))

Time Sensitive applications: Multimedia applications, and soft real-time applications in general, are driven by real-world demands and are characterized by timing constraints that must be satisfied for correct operation; for this reason, we call these applications time-sensitive.

- **This paper shows that there are three important requirements for achieving timely resource allocation**

Time-Sensitive Requirements most important requirement is resources must be allocated at “appropriate” times. These appropriate times are determined by events of interest to the application.
In response to these events, the application is scheduled or activated.
We call the latency between the event and its activation, kernel latency. Kernel latency should be low in a time-sensitive system

To reduce these components of kernel latency, there are three requirements:

1) Timing Mechanism:

FIRM TIMERS:
Combination of Timers, One shot timer and soft timer, these techniques are complimentary to each other

TYPES OF TIMERS:

1. PERIODIC TIMER:
   - Traditionally used timers in commodity OS, normally implemented with periodic timer interupts. But this can cause latency when the

period is large, solution to this is keeping the period small but this causes problem of high frquncy of interupts.

2. ONE SHOT TIMER:

- To reduce the overhead of timers, it is necessary to move from a periodic timer interrupt model to a one-shot timer interrupt model where interrupts are generated only when needed. Consider two tasks with periods 5 and 7 ms. With periodic timers and a period of 1 ms, the maximum timer latency would be 1 ms. In addition, in 35 ms, 35 interrupts would be generated. With one-shot timers, interrupts will be generated at 5 ms, 7 ms, 10 ms, etc., and the total number of interrupts in 35 ms is 11.
- One-shot timers can provide high accuracy, but, unlike periodic timers, they require reprogramming at each activation.
- One-shot timers generate a timer interrupt at the next timer expiry, to reprogramming the timer. So there are two problems:
  - Timer reprogramming: This problem is resolved by modern systems
  - Fielding Timer Interupt: To resolve this we need soft timers.

3. SOFT TIMER:

- They poll for expired timers at strategic points in the kernel such as at system call, interrupt, and exception return paths.
- While soft timers reduce the costs associated with interrupt handling, they introduce two new problems:
  - Cost of polling: This cost can be ammortized if the timer completes a certain number of check each time it is fired.
  - Timer latency: when the checks occur infrequently or the distribution of the checks and the timer deadlines are not well matched--> this problem is resolved by one-shot timers with soft timers by exposing a system-wide timer overshoot parameter. With this parameter, the one-shot timer is programmed to fire an overshoot amount of time after the next timer expiry.
  - The timer overshoot parameter allows making a trade-off between accuracy and overhead. A small value of timer overshoot provides high timer resolution but increases overhead since the soft timing component of firm timers are less likely to be effective. Conversely, a large value

decreases timer overhead at the cost of increased maximum timer latency. The overshoot value can be changed dynamically

FIRM TIMER IMPLEMETATION:

- **Timer Queue:** TSL maintains a timer queue for each processor. This queue keeps track of timers and sorts them based on when they are set to expire.

- **APIC Timer:** The one-shot APIC timer is a hardware timer that generates an interrupt when a timer expires. The interrupt handler checks the timer queue and executes callback functions associated with expired timers.
- **Handling Timers:** Expired timers are removed from the queue, while periodic timers are rescheduled to trigger again after their specified time period.
- **Timer Accuracy:** The accuracy of timers depends on the APIC timer. In theory, it can provide accuracy up to 10 nanoseconds, but in practice, the time taken to handle timer interrupts limits this accuracy.
- **Soft Timers:** Soft timers are introduced with a non-zero timer overshoot value. They add some delay after the next timer event, improving timer accuracy.
- **Global Overshoot Value:** The current implementation uses a single global overshoot value, but it can be extended to allow each timer or application to specify its desired overshoot or timing accuracy.
- **Data Structure Efficiency:** Periodic timers are more efficient than one-shot timers in terms of data structures. Periodic timers use calendar queues, which operate quickly, while one-shot timers require priority heaps, which take longer to process.
- **Combining Timing Mechanisms:** To benefit from the efficiency of periodic timers, firm timers combine periodic and one-shot timing mechanisms. Long timeouts are handled with periodic timers followed by setting the one-shot APIC timer.

2) Responsive Kernel:

An accurate timing mechanism is not sufficient for reducing latency. For example, even if a timer interrupt is generated by the hard-ware at the correct time, the activation could occur much later because the kernel is unable to interrupt its current activity. This problem occurs because either the interrupt might be disabled or the ker-nel is in a non-preemptible section.

The solution for improving kernel response is to reduce the size of such non-preemptible sections.

NOTE: One solution that was initially thought in the paper was to explicitly put preemption points in the code.

This done using the way we do in GATE, using spinlocks, but this cases issue when they are held for a long time, the solution is to explicitly preempt when they are held for long time.

3) Appropriate CPU Scheduling algorithm:

Assumption of the scheduling algorithm is that the kernel is preemtive, this works beacause of the two conditions we have satisfied above, A responsive kernel with an accurate timing mechanism

In this paper, we use two different real-time scheduling algorithms: a proportion-period scheduler and a priority-based scheduler.

PROPORTIONAL-PERIOD SCHEDULER:

- The proportion-period allocation model automatically provides temporal protection because each task is allocated a fixed proportion of the CPU at each task period.
- We implemented a proportion-period CPU scheduler in Linux by using real-time scheduling techniques (EDF priority assignment) and by allowing a task to execute as a real-time task for a time Q every period T. The re-served time Q is equal to the product of the task's pro-portion P and its period T, and the end of each period is used as a deadline. After executing for a time Q, the task

is blocked (or scheduled as a non real-time task) until its next period.

PRIORITY BASED SCHEDULER:

- In the priority model, real-time priorities are assigned to time-sensitive tasks based on application needs [10]. Since the fixed priority model does not provide temporal protection, TSL schedules fixed priority tasks in the background with respect to proportion-period tasks. The only exception to this rule is with shared server tasks because they can cause priority inversion
- We use a variant of the priority ceiling protocol [19] called the highest locking priority (HLP) protocol to cope with priority inversion. The HLP protocol works as follows: when a task acquires a resource, it automatically gets the highest priority of any task that can acquire this resource

CONCLUSION:

This paper describes the design and implementation of a Time-Sensitive Linux (TSL) system that can support applications requiring fine-grained resource allocation and low-latency response. The three key techniques that we have investigated in the context of TSL are firm timers for accurate timing, fine-grained kernel preemptibility for improving kernel responsiveness and proportion-period scheduling for providing precise allocations to tasks. Our experiments show that integrating these techniques helps provide allocations to time-sensitive tasks