RTOS: Scheduling Algorithms

Prof. P.H.Dave
DDU, Nadiad;
-
Prof. H.B.Dave
eInfochips, Ahmedabad.

March, 2015

Real-Time Scheduling algorithms

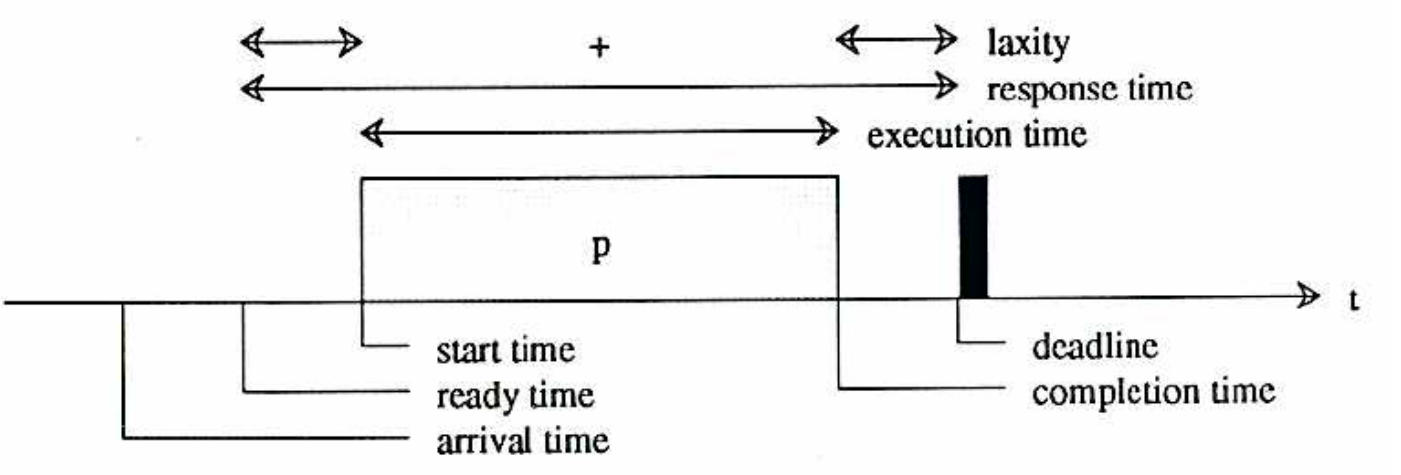
- Earliest Deadline First Scheduling (EDF)
- Least Laxity Time First Scheduling (LLF)
- Modified Least Laxity First (MLLF)
- Maximum Urgency First - (MUF)
- Modified Maximum Urgency First - (MMUF)
- Event-triggered, Sporadic Tasks

Scheduling parameters in RTOS

We have to know the basic scheduling parameters. A task i is characterized by:

- c_i: computation time
- s_i: start time
- d_i: deadline (relative to start time)
- p_i: period or minimum separation, i.e., Periodic vs. Aperiodic task
- **Laxity**: $l_i = d_i - c_i$ amount of time margin (*Laxity*) before Task must begin execution
- **Utilization factor**: $U = \sum_{i=1}^{n} \frac{c_i}{p_i}$, where n = number of tasks.
- **Schedulability Test**: $U \leq n(2^{1/n} - 1)$, also called Feasibility test, which means meeting timing constrains and resource req

Task Time Parameters

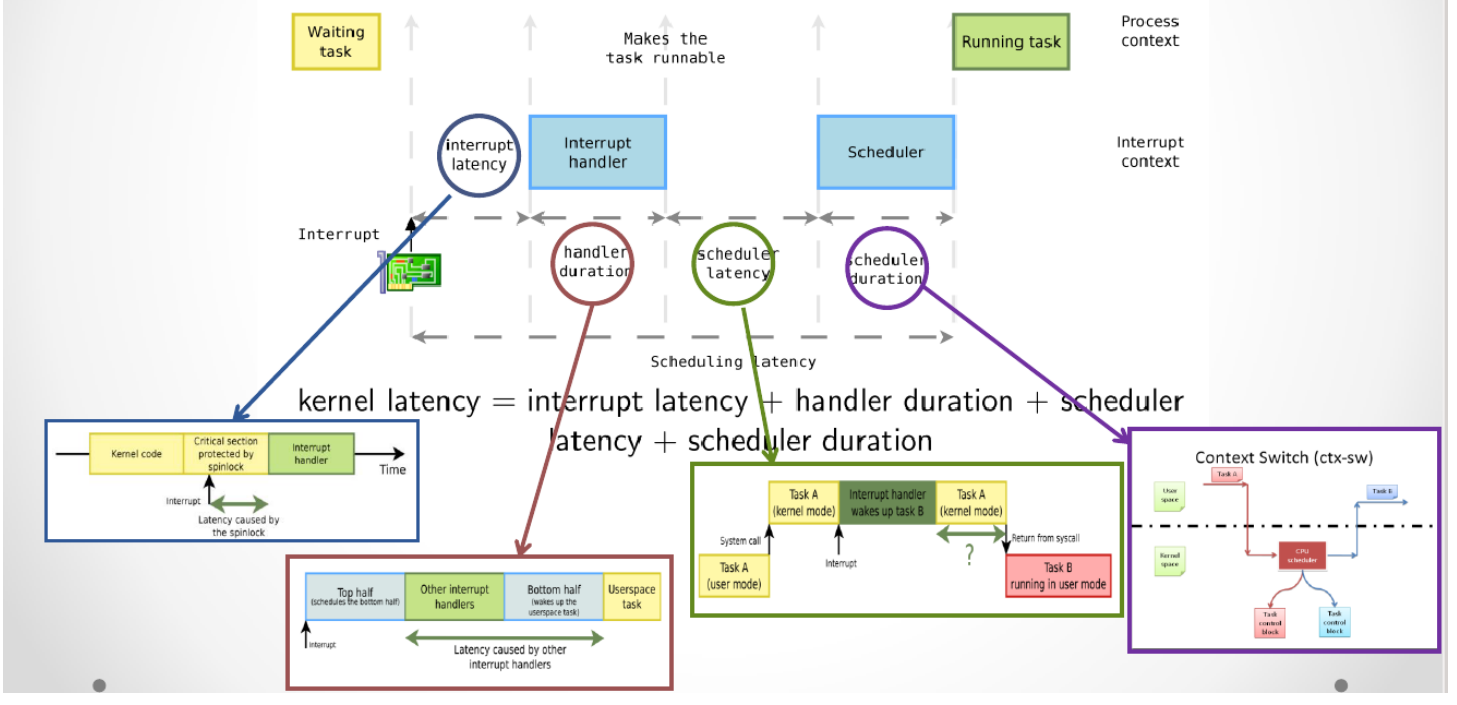


Interrupt Latency

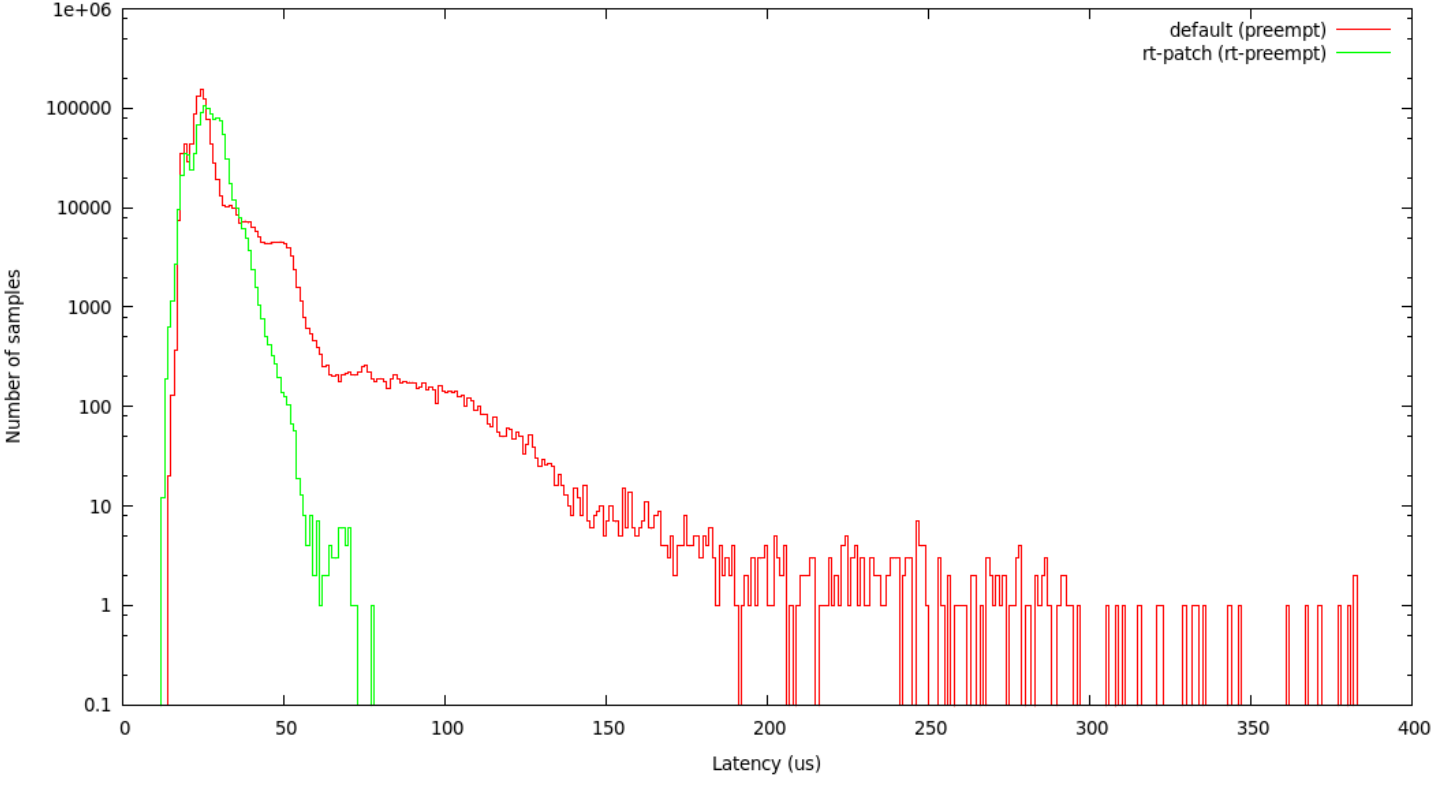
- Interrupt Latency is the time that elapses from the time of occurrence of the interrupt to time of service of the interrupt.
- Interrupt latency is an important parameter for a particular processor being used in an embedded system.
- It contributes to the Real-time performance of the system.
- Micro-controllers are designed to keep the interrupt latency as small as possible.
- Factors: nature of interrupt controller, current interrupt masking, O/S interrupt handling methods.
- The Linux Kernel provides a typical arrangement: the ISR for a device performs only the bare minimum steps necessary to keep the device functioning, leaving all other data processing to a **tasklet**

- **Tasklets** are run by the Kernel after all pending interrupts have been serviced.
- Newly arriving interrupts will pre-empt currently running tasklets, ensuring system responsiveness.
- Tasklets are scheduled on a higher priority than user-level jobs, ensuring that the data is processed in a timely manner.

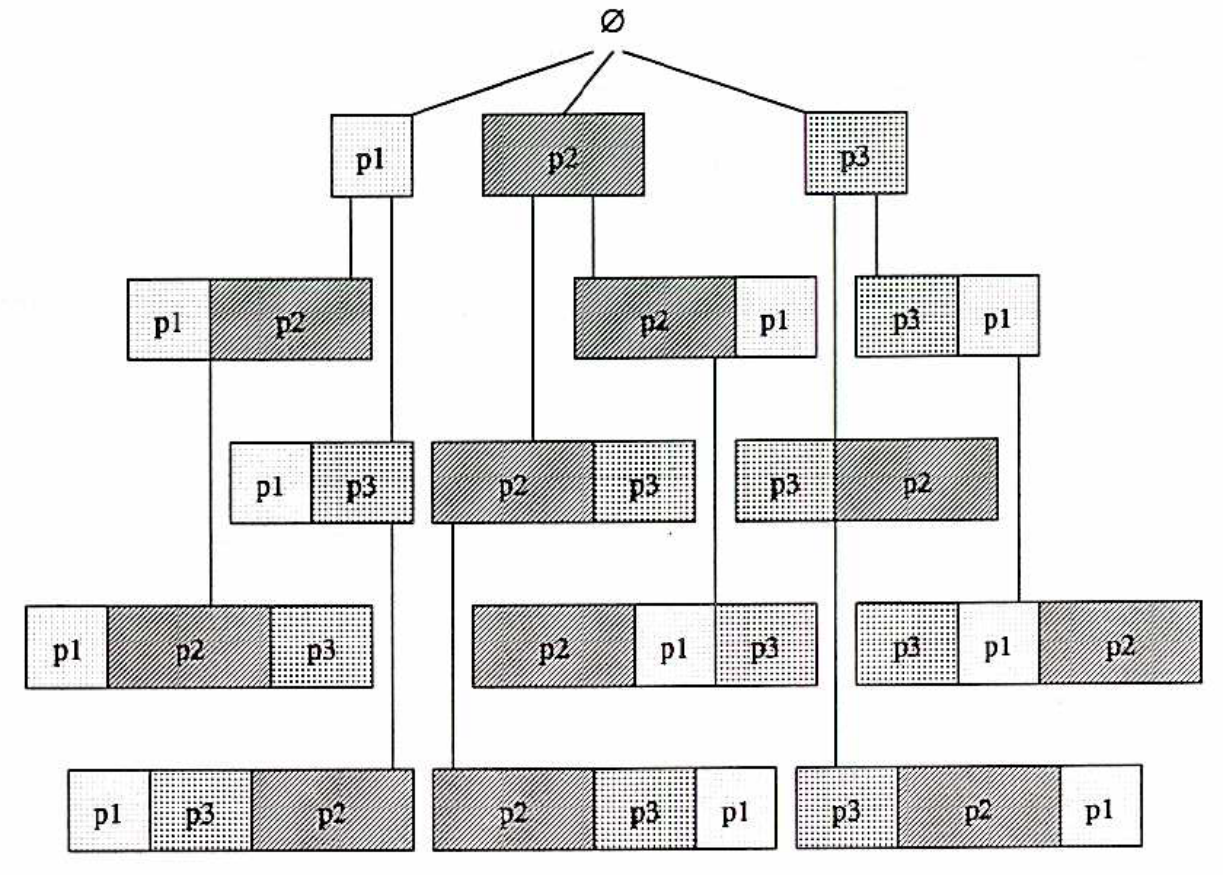
Components of Linux Kernel Interrupt Latency



Comparision of Interrupt Latency: Normal vs. PREMPT_RT



Scheduling is Tree Search of Permutations of Tasks



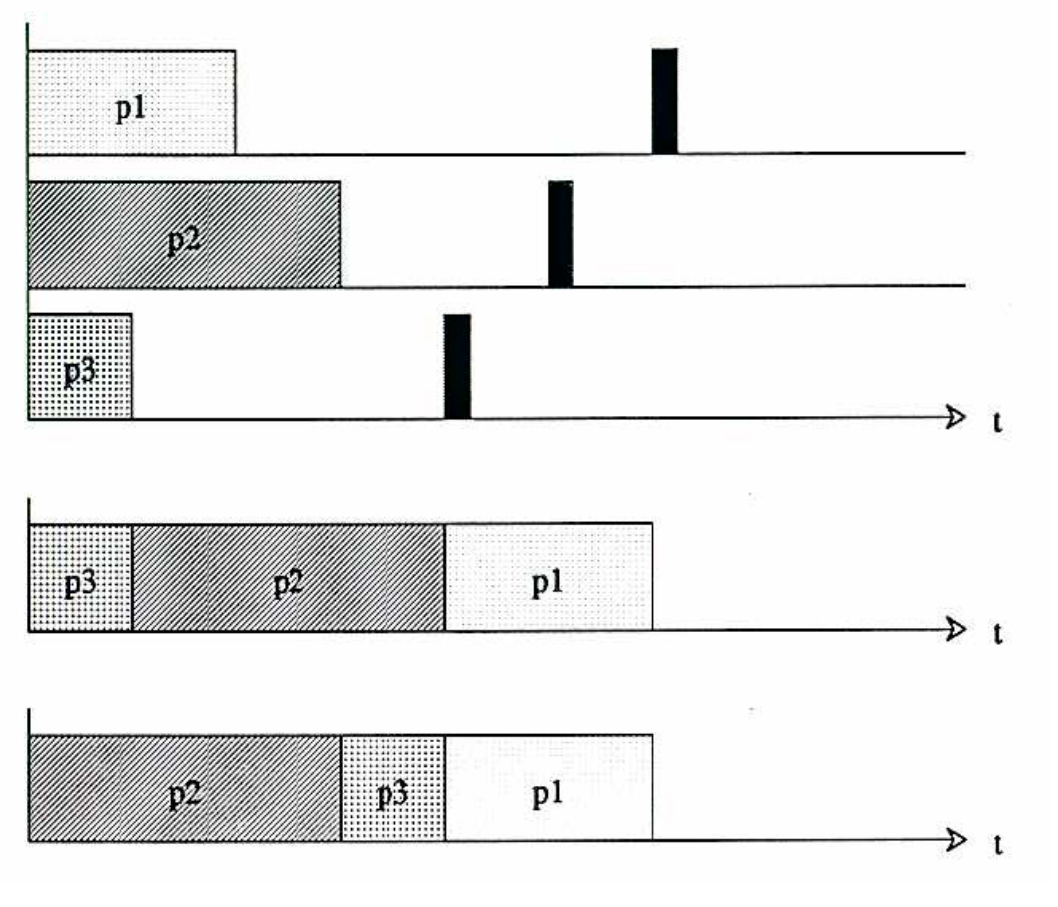
Earliest Deadline First Scheduling (EDF)

- A Dynamic priority scheduler
- The scheduler adjusts the priority as per approach of a deadline.
- Highest priority is assigned to the task with the nearest deadline from the current time.
- Uses preemptive scheduling.
- Advantages are: theoretically superior to RM and guaranteed Schedulability if CPU utilization 100% or less.
- Disadvantages are:
  - More difficult to implement
  - Higher system overhead

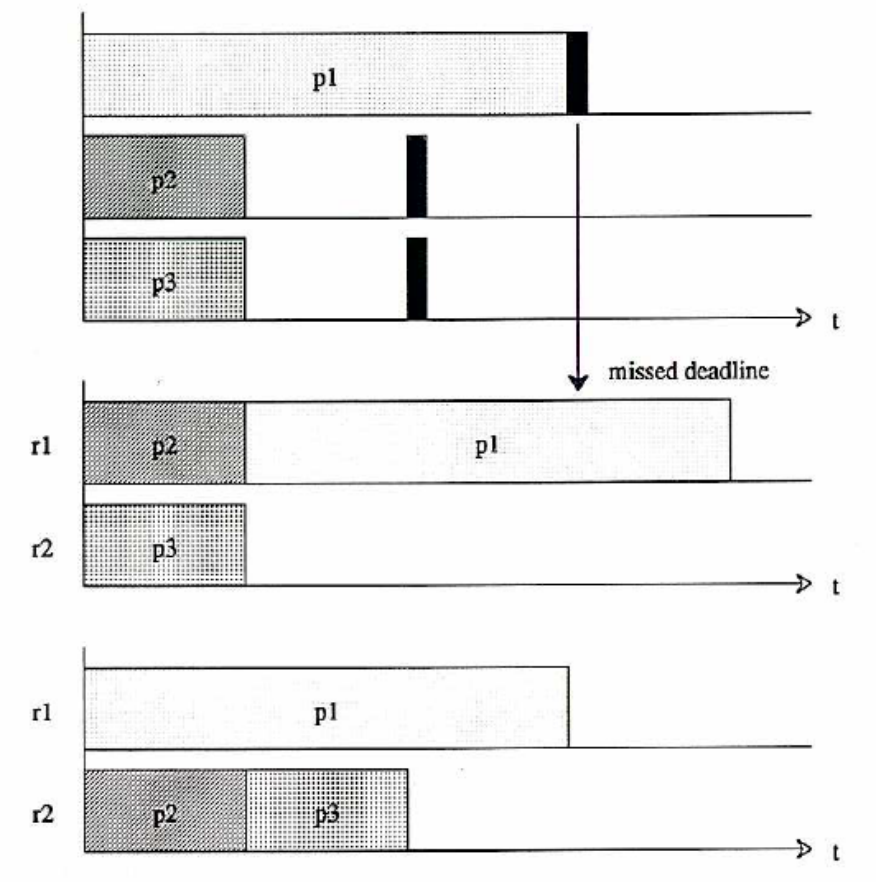
- Overloaded system is unpredictable
- Non-critical tasks may be scheduled before critical tasks
- Does not guarantee to execute a set of higher priority tasks before their deadlines

For example, consider three tasks in the Ready-queue T2(12,4), T3(10,4), T1(5,3) – where the values in the parentheses are (deadline, compute time). With FIFO scheduling, the task will be executed in order T2, T3, T1 and T1 will miss its deadline. With an EDF scheduler, the tasks will be executed in order T1, T3, T2 and all will meet their deadline.

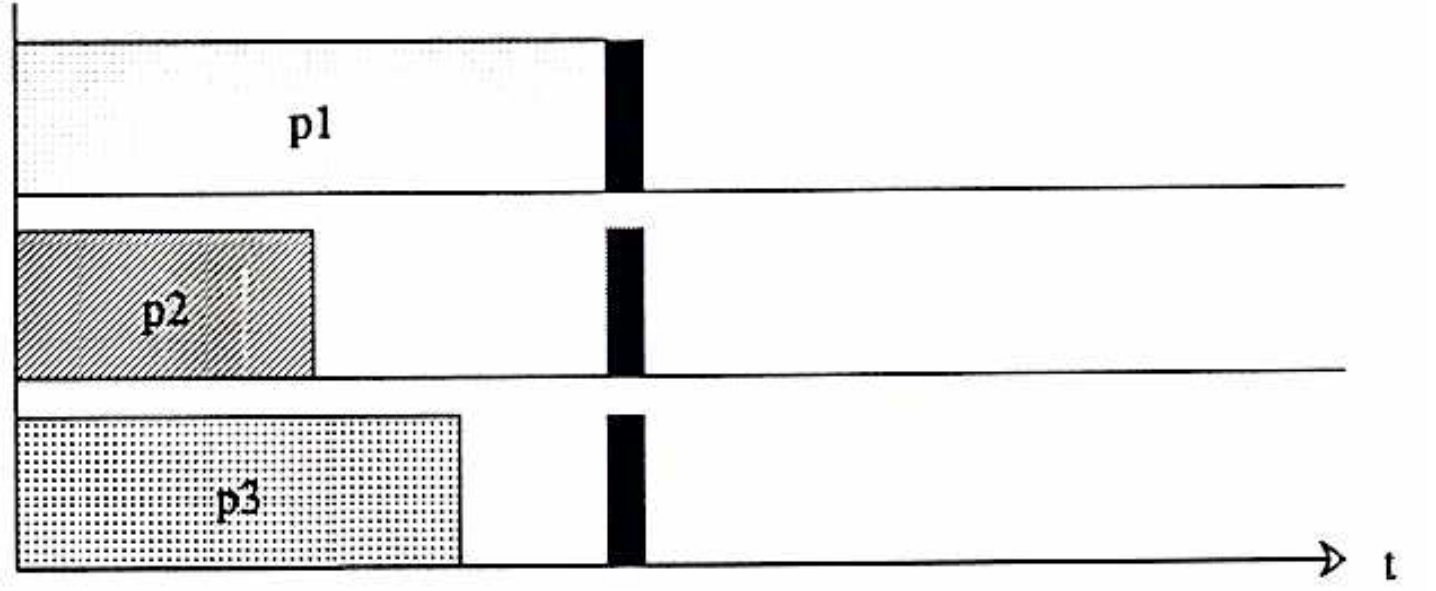
Optimal Scheduling EDF and LLF



Two Processors: EDF Fails, LLF schedules



Why Do we need to consider LLF Scheduling?



Least Laxity Time First Scheduling (LLF)

- A Dynamic priority scheduler.
- Highest priority is given to the task with the least *laxity* time, where Laxity = time to deadline - execution time left, $l_i = d_i - c_i$.
- It takes into account both a task's deadline and its processing load.
- When the LLF scheduler has evaluated the laxity for all tasks, it finds the task with the smallest current value of laxity
- That is the task that needs to be scheduled to run next.
- Advantage of LLF is that it is Intuitively logical.

Modified Least Laxity First (MLLF)

LLF is impractical to implement because laxity ties result in poor system performance due to the frequent context switches among the tasks. A laxity tie occurs when two or more tasks have the same laxity. MLLF solves the problem by reducing the number of context switches.

- As long as there is no laxity tie, MLLF schedules the task sets the same as LLF algorithm.
- If the laxity tie occurs, the running task continues to run with no preemption as far as the deadlines of other tasks are not missed.
- The MLLF algorithm defers the context switching until necessary and it is safe even if a laxity tie occurs.

- Allows the laxity inversion where a task with the least laxity may not be scheduled immediately.
- Laxity inversion applies to the duration that the currently running task can continue running with no loss in Schedulability, even if there exist a task (or tasks) whose laxity is smaller than the current running task

Maximum Urgency First - (MUF)

It defines a critical set of tasks that is guaranteed to meet all its deadlines during a transient overload. It is a mixture of some LLF deadline scheduling, with some traditional priority-based pre-emptive scheduling. It combines the advantages of fixed and dynamic scheduling to provide the dynamically changing systems with flexible scheduling.
In MUF each task is given an urgency:

- combination of two fixed priorities (*criticality* and user priority) and a dynamic priority that is inversely proportional to the laxity.
- The criticality has higher precedence over the dynamic priority while user priority has lower precedence than the dynamic priority.
- A disadvantage: rescheduling operation is performed whenever a task arrives to the Ready-queue which may cause a critical task to fail.
- It assigns priorities in two phases:

1. Phase One: concerns the assignment of static priorities to tasks, which is assigned once and does not change.
2. Phase Two: deals with the run-time behavior of the MUF scheduler as it is given below.

- **First Phase**:

1. Sorting tasks from shortest period to the longest period.
2. Defining the critical set as the first N tasks such that total CPU load factor does not exceed 100%. These tasks are guaranteed not to fail even during a transient overload.
3. Critical set tasks are assigned high criticality. The remaining tasks are considered to have low criticality.
4. Every task in the system is assigned an optional unique user priority.

- **Second Phase**: follows an algorithm to select a task for execution. This algorithm is executed whenever a new task has arrived into the Ready-queue.

  1. If there is only one highly critical task, pick it up and execute it.

  2. If there are more than one highly critical task, select the one with the highest dynamic priority. Here, the task with the least laxity is considered to be the one with the highest priority.

  3. If there is more than one task with the same laxity, select the one with the highest user priority.

Modified Maximum Urgency First - (MMUF)

It is a pre-emptive mixed priority algorithm for predictable scheduling of periodic Real-time tasks. Significant features are:

- use a unique *importance* parameter, instead of using tasks request intervals, to create the critical set.
- The importance parameter is a fixed priority which can be defined as user priority or any other optional parameter which expresses the degree of the task's criticalness.
- It is obvious that the task with the shortest request period is not necessarily the most important one.
- Either EDF or MLLF can be used to define the dynamic priority.

The MMUF algorithm consists of two phases:

- **Phase 1**: In this phase fixed priorities are defined only once as follows, not changed during execution time.

  1. Sorting the tasks from the most important to the least important.
  2. Add the first N tasks to the critical set such that the total CPU load factor does not exceed 100%.

- **Phase 2**: This phase calculates the dynamic priorities at every scheduling event and selects the task to be executed next.

  1. If there is at least one critical task in the ready queue: Select the critical task with the earliest deadline (EDF algorithm) if there is no tie.

2. If there are two or more critical tasks with the same earliest deadline: if any of these critical tasks is already running select it to continue running; otherwise, select the critical task with the highest importance.

3. If there is no critical task in the ready queue: Select the task with earliest deadline (EDF algorithm) if there is no tie.

4. If there are two or more tasks with the same earliest deadline: If any of these tasks is already running select it to continue running; Otherwise, select the task with the highest importance.

Event-triggered, Sporadic Tasks

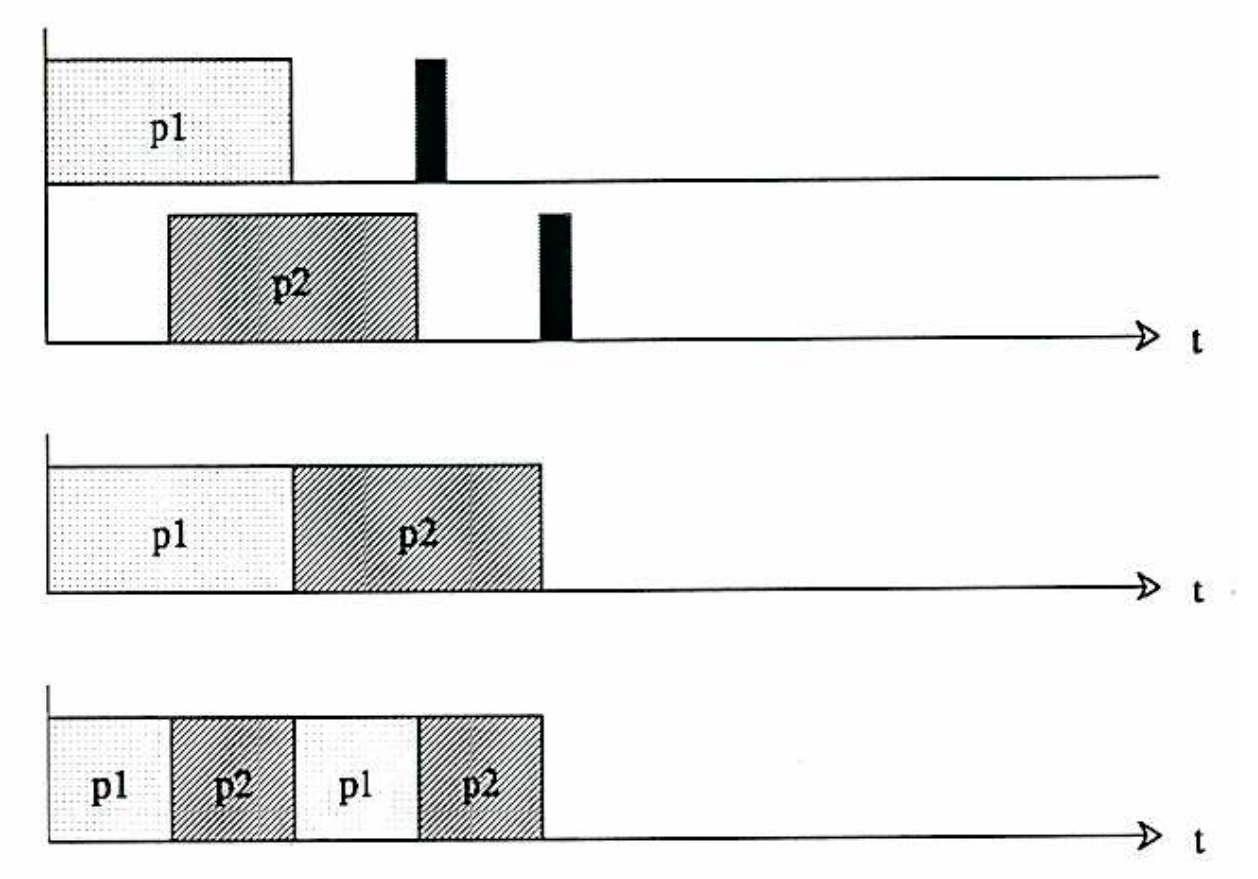
These are tasks that are released irregularly, often in response to some event in the environment. There are no periods associated with them, but must have some maximum release rate, i.e., minimum inter-arrival time. In order to deal with them we can consider them as periodic with a period equal to the minimum inter-arrival time. However there are other approaches. The first approach is a naive approach:

- Define fictitious periodic task of highest priority and of some chosen execution period
- During the time this task is scheduled to run, the processor can run any sporadic task that is awaiting service and if no sporadic task is awaiting service, the processor is idle.
- Apart from this time the processor attends to periodic tasks.
- This arrangement can be wasteful.

This second approach is known as Deferred Server:

- It is less wasteful than the above approach.
- Whenever the processor is scheduled to run sporadic tasks, and finds no such tasks awaiting service, it starts executing other, periodic, tasks in order of priority.
- However, if a sporadic task arrives, it preempts the periodic task and can occupy a total time up to the time allotted for a sporadic task.

Reduce the Number of Task Switches to Reduce Power



Power Management in Embedded Systems

A large percentage of Embedded Systems use batteries or other low power sources like Solar energy, Super-capacitor, etc. It is necessary to limit use of energy from such sources. Two basic approaches seems viable:

- **Clock Frequency control**: as almost all the Embedded Systems use CMOS technology, the power consumption is directly proportional to the clock frequency. The Operating Environment should arrange to reduce the clock frequency whenever CPU load is reduced due to less compute heavy work being done.
  - The developer programmer usually visualizes the software as a FSM, with state transitions depending upon various external and internal inputs received. For a state which does not present heavy computational load, the programmer can reduce clock frequency at entry point.

- The Operating Environment itself can continuously monitor the short-term utilization factor of the CPU and reduce the clock rate accordingly.

- **Using proper Scheduling algorithms**: such that the task switches are kept minimum possible. Each task switch is a fast event and the CPU is expected to run at full speed during the switch, but the switched-in task may not require all that speed from the CPU.