Embedded Systems Task Scheduling Algorithms and Deterministic Behavior

Prepared By

Khalil Mustafa Yousef
Nabeel Barakat

Supervised by

Dr. Lo'ai Tawalbeh

Dept. of Computer Engineering
Faculty of Computer and Information Technology
Jordan University of Science and Technology
Contents

• Why need scheduling?
• Task scheduling
• Taxonomy of Real-Time Scheduling
• Real-Time Scheduling
• Sporadic Tasks
• “Real Time” and Determinism
  – Deterministic systems
  – Non- Deterministic systems
• Conclusions and Summary
Why Need Scheduling?

• Industry:
  • some of the most stable operating systems are for embedded systems because criticality tasks (the controllers in nuclear power plants, military systems, life support equipment, etc.) must never crash.

• But: Embedded operating systems have fewer resources to work with.

• So: A good embedded operating system should
  1. take up only a small portion of code memory
  2. use only a small amount of time to handle important requests or switch between processes,
  3. and be predictable.
  4. What’s else??
Why Need Scheduling?

4. meeting specific time deadlines for tasks to occur in.

- **Difficulty**: it must be able to guarantee that the worst case response time for the operating system to give control to a process that needs attention is short enough that the process has time to handle events.

- **One aid** to ensuring sufficient response time is to prioritize processes so that more important processes always receive processor attention if they need it,

- Thus the idea of scheduling based on priorities
Task Scheduling

• **The task scheduler**  
  – the part of the operating system that responds to the requests by programs and interrupts for processor attention and gives control of the processor to those processes.

• **Scheduling algorithm**: The algorithm followed to decide who gets next turn on CPU.

• The program that does this is called the **Scheduler**

• **In real-time systems**,  
  • every real-time task has a **deadline** before or at which it must be completed.  
  • the scheduling algorithms of these systems must be timely and predictable, Due to the criticality of the tasks,

• **Task scheduling**:  
  • Tasks could either be periodic or aperiodic.  
  • Real-time scheduling algorithms may assign priorities statically, dynamically, or in a **hybrid manner**, which are called fixed, dynamic and mixed scheduling algorithms, respectively.  
  • These algorithms may allow preemptions to occur or may impose a non-preemptive method.
Taxonomy of Real-Time Scheduling

- Pre-emption
  - Saving CPU-context and moving execution

- Preemptive permits one task to preempt another one of lower priority

- Preemptive
- Non-preemptive

- Dynamic
- Static

- Soft
- Hard

- Computed at run-time based
- done at compile time for all possible tasks
Recall From Previous Lecture

(a) Round-robin Preemptive Scheduler

(b) Priority-Driven Nonpreemptive Scheduler
Recall From Previous lecture

(c) Priority-Driven Preemptive Scheduler on Preemption Points

Real-time process preempts current process and executes immediately

(d) Immediate Preemptive Scheduler
Scheduling Parameters

• **Characterization of a task (i):**
  - $c_i$ computation time
  - $s_i$ start time (task dependencies)
  - $d_i$ deadline (relative to start time)
  - $p$ period or minimum separation (Periodic vs. Aperiodic)

• **Handy values:**
  - **Laxity** $l_i = d_i - c_i$ (amount of Laxity time before Task must begin execution)
  - **Utilization factor** $u_i = \sum_{i=1}^{n} \frac{C_i}{P_i}$, $n$: Number of tasks
  - **Schedulability Test:** $U \leq n(2^{1/n} - 1)$
Deadline Scheduling

- **Real-time** applications are **not concerned with speed but with completing tasks**

- In this approach, *scheduler is provided with information about task deadlines.*

  **Information used**
  - Ready time
  - Starting deadline
  - Completion deadline
  - Processing time
  - Resource requirements
  - Priority
  - Subtask scheduler

- **The deadline scheduler** gets the tasks to run before they miss their deadlines, by preemptively “borrowing” running time from tasks that normally have higher priorities.

  • How to implement? **But Before, Let’s Introduce RM**
Rate Monotonic Scheduling Algorithm

• Rate Monotonic (RM)
  – Static fixed priority scheduler based on Task Periods
    • Assigns priorities to tasks on the basis of their periods
    • Highest-priority task is the one with the shortest period
  – Immediately pre-empts any running task with a higher priority task
  – Negligible context-switching time,
  – Periodic tasks
  – No precedence constraints

• Advantages
  – Easy to implement (most widely used)
  – Low system overhead
  – Optimal among other static priorities algorithms

• Disadvantages
  – Requires static prioritization before run-time

Static prioritization can be difficult since it is not certain what task may be more critical at a given time
Feasibility (Schedulability) test

- **Feasibility means:**
  - Meeting timing constrains
  - Meeting resource requirements

- \( U = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \)

- The **least upper bound of** \( U \): for all task sets whose \( U \) is below this bound, \( \exists \) a fixed priority assignment which is feasible

- \( \text{LUB-}U=n(2^{1/n}-1) \)
  - \( n \): periodic tasks

- \( \left( \frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \right) \leq n(2^{1/n}-1) \)
Example

• Consider $\tau_1$ & $\tau_2$ with $T_1=2$, $T_2=5$, & $C_1=1$, $C_2=1$
• $\tau_1$ has higher priority than $\tau_2$
  • priority assignment is feasible
  • can increase $C_2$ to 2 and still be able to schedule
Earliest Deadline First Scheduling Algorithm

• Earliest Deadline First (EDF)
  – Dynamic priority scheduler
  – Highest priority is assigned to the task with the nearest deadline
  – Preemptive

• Advantages
  – EDF theoretically superior to RM
  – Guaranteed scheduleability if CPU utilization 100% or less

• Disadvantages
  – 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 the set of higher priority tasks before their deadlines
EDF vs FIFO (non-real-time scheduler)

Example: task s d c
1 0 5 3
2 0 12 4
3 0 10 4

FIFO schedule: 2, 3, 1 (task 1 misses deadline)
EDF schedule: 1, 3, 2 (all tasks meet deadlines)
Earliest Deadline First (EDF)

- Example:
  - $T_1 = 50, \quad C_1 = 25; \quad T_2 = 62.5, \quad C_2 = 10; \quad T_3 = 125, \quad C_3 = 25.$
Example Two Tasks

Figure: Scheduling of Periodic Real-time Tasks with Completion Deadline

Table 1: Execution Profile of Two Periodic Tasks

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Execution Time</th>
<th>Ending Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1)</td>
<td>0</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>A(2)</td>
<td>20</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>A(3)</td>
<td>40</td>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>A(4)</td>
<td>60</td>
<td>10</td>
<td>80</td>
</tr>
<tr>
<td>A(5)</td>
<td>80</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>B(1)</td>
<td>0</td>
<td>25</td>
<td>50</td>
</tr>
<tr>
<td>B(2)</td>
<td>50</td>
<td>25</td>
<td>100</td>
</tr>
</tbody>
</table>
Least Laxity Time First Scheduling Algorithm

• Least laxity Time First
  – Dynamic priority scheduler
  – Highest priority given to the task with the least laxity time (Laxity = time To Deadline – execution Time Left)
  – It takes into account both a task’s deadline and its processing load.
  – When the LL scheduler has evaluated the laxity for all tasks, it finds the task with the smallest current value of laxity – and that is the task that needs to be scheduled to run next

• Advantages
  – Intuitively logical
Least Laxity Time First Scheduling Algorithm

Example:
- \( T1 = 50 \), \( C1 = 25 \); \( T2 = 62.5 \), \( C2 = 10 \); \( T3 = 125 \), \( C3 = 25 \).
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 laxities.
- **MLLF** solve the problem by reducing the number of context switches.
- **How it works:**
  - As long as there is no laxity tie, MLLF schedules the task the same as the **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 the 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.
MUF – Maximum Urgency First Scheduling.

- 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 LL 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.

- **How it works:**
  - 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**.

- **Shortcoming:** rescheduling operation is performed whenever a task is arrived to the ready queue **which may cause a critical task to fail.**
MUF

The MUF algorithm assigns priorities in two phases.

– Phase One concerns the assignment of static priorities to tasks. (assigned once and do not change)
– Phase Two deals with the run-time behavior of the MUF scheduler as it is clarified later.

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

In the second phase, the MUF follows an algorithm to select a task for execution.
This algorithm is executed whenever a new task is arrived to 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.
MUF Example

MUF will select the task with minimum laxity ($T_1$) at time zero. The remaining execution time of task $T_1$ is greater than remaining time to $T_2$’s laxity. This selection will cause task $T_2$ to miss its deadline.

Fig. 1 Schedule generated by the MUF scheduling algorithm
**Modified Maximum Urgency First (MMUF) scheduling algorithm**

- **Shortcoming:** rescheduling operation is performed whenever a task is arrived to the ready queue which may cause a critical task to fail.
- **MMUF** is a preemptive mixed priority algorithm for predictable scheduling of periodic real-time tasks.
- With this algorithm,
  - 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 trivial that the task with the shortest request period is not necessarily the most important one.
  - With the MMUF algorithm, either EDF or MLLF can be used to define the dynamic priority.
MMUF

The MMUF algorithm consists of two phases:

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

1. **Sorting the tasks** from the **most importance** to the least importance
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.

- **A)** If there is **at least one critical task in the ready queue**
  1. 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**

- **B)** If there is **no critical task in the ready queue**
  1. Select the task with earliest deadline (EDF algorithm) **if there is no tie**
  2. 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**
Sporadic Tasks

- Tasks that are released **irregularly**, often in response to some event in the environment
  - no periods associated
  - but must have some maximum release rate (**minimum inter-arrival time**)
    - Otherwise no limit on workload!

- **How to deal with them?**
  - consider them as periodic with a period equal to the minimum inter-arrival time
  - other approaches…
Handling Sporadic Tasks: Approach 1

• 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
  • if no sporadic task awaiting service, **processor is idle**
• Outside this time the processor attends to periodic tasks
• **Problem**: wasteful!
Handling Sporadic Tasks
Approach 2 (Deferred Server)

• Less wasteful…
• 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 sporadic task
• Schedulability?
Dealing with Transient Overload

- Transient system overload may cause some deadlines to be missed
- Lower priority tasks are likely to miss their deadlines first in an overload situation
- But, the *more important task* may have been assigned a lower priority: priority $\neq$ importance
- One could assign priorities according to importance
  - e.g. by artificially reducing smaller deadline
  - but... reduces schedulability
Example

• Consider two tasks:
  Task $\tau_1$: $C_1 = 3.5; T_1 = 10; D_1 = 10$; less important
  Task $\tau_2$: $C_2 = 7; T_2 = 14; D_2 = 13$; critical task
• $\tau_2$ will have lower priority
  • completion time test shows that $\tau_2$ is not schedulable
  • but is important and must be guaranteed!
• Making priority $\tau_2$ of $\tau_1$ will make unschedulable
A Better Approach: Period Transformation

• One could transform period of $\tau_2$ to 7, yielding a modified task set
  Task $\tau_1$: $C_1 = 3.5$; $T_1 = 10$; $D_1 = 10$
  Task $\tau_{2a}$: $C_{2a} = 3.5$; $T_{2a} = 7$; $D_{2a} = 6$

• Note: in period transformation, the real deadline is at the last transformed period
  • deadline at the second transformed period of $\tau_{2a}$ is at most 6 ($7+6=13$)
  • Now, $\tau_{2a}$ has higher priority, and the task set is schedulable!
Using Period Transformation to Improve Schedulability

• Consider two tasks:
  Task \( \tau_1: C_1=5; T_1=10 \)
  Task \( \tau_2: C_2=5; T_2=15 \)

• These two tasks are just schedulable with utilization 83.3%

• If we transform,
  Task \( \tau_{1a}: C_{1a}=2.5; T_{1a}=5 \)
  Task \( \tau_2: C_2=5; T_2=15 \)
  the utilization bound becomes 100%
“Real Time” and Determinism

• What exactly is “real time”?  
  – “Real time” is defined as a system with deterministic results.  
    • This means that you can guarantee that when an action occurs, *the time for the reaction has an defined upper bound.*

• Determinism  
  – Operations are performed at fixed, predetermined times or within predetermined time intervals  
  – Equivalent to *predictability*  
  – Lack of RT determinism can result in missed deadlines

• Example: An interrupt occurs, the handler must run within some bounded time frame or bad things happen (system failure, damaged product, injury, etc)
Determinism

• Non-deterministic:
  – Time required to collect is indeterminate
  – Not meeting Timing constrains

• **Timing constraint**: constraint imposed on timing behavior of a job: Hard and soft constraints
  – *Timing constraints* derived from *physical* impact of controlling systems activities
Deterministic systems

• A **deterministic system** will react within bound delay under all conditions.
  - Concerned with how long the operating system delays before acknowledging an interrupt and there is sufficient capacity to handle all the requests within the required time.

• A **deterministic system** can be defeated by external causes:
  - (failure of a device, severing of communication line),
  - but this is considered as an accepted exceptional situation for which reaction is foreseen.

• **Determinism implies** : reservation of all resources (bus, memory space, ...) needed to complete the task timely.

• **Determinism** is closely related to the principle of cyclic operation.
Non-deterministic systems

• A non-deterministic system
  – can fail to meet its deadline because of internal causes (congestion, waiting on resource), without any external cause.

• Computers and communication may introduce non-deterministic delays, due to internal and external causes:
  – response to asynchronous events from the outside world (interrupts)
  – access to shared resources: computing power, memory, network driver, ...
  – use of devices with non-deterministic behavior (hard-disk sector position)

• Non-determinism is especially caused by:
  – Operating system with preemptive scheduling (UNIX, Windows,..) or virtual memory (in addition, their scheduling algorithm is not parametrizable)
  – Programming languages with garbage collection (Java, C#, ...)
  – Communication systems using a shared medium with collision (Ethernet)
  – Queues for access to the network (ports, sockets)

• Non-determinism is closely related to on-demand (event-driven) operation
Hard- and Soft real time

hard real-time (deterministic)

- probability
- bound!
- the probability of the delay to exceed an arbitrary value is zero under normal operating conditions, including recovery from error conditions
- meet all time constraints exactly (hard real-time)

soft real-time (non-deterministic)

- probability
- unbound!
- the probability of the delay to exceed an arbitrary value is small, but non-zero under normal operating conditions, including recovery from error conditions
- meet timing constraints most of the time (soft real-time)
Summary and Conclusion

• Task characteristics
• Periodic task scheduling algorithms
• Aperiodic task scheduling algorithms
• Transient Overload
• Task Synchronization
• “Real Time” and Determinism
Questions ?
References

• TRANSACTIONS ON ENGINEERING, COMPUTING AND TECHNOLOGY V9 NOVEMBER 2005 ISSN 1305-5313, “A Modified Maximum Urgency First Scheduling Algorithm for Real-Time Tasks”, Vahid Salmani, Saman Taghavi Zargar, and Mahmoud Naghibzadeh