Real-time Systems

Joe Sventek
Objectives

- Explain the timing requirements of real-time systems
- Distinguish between hard and soft real-time systems
- Discuss the defining characteristics of real-time systems
- Describe scheduling algorithms for hard real-time systems
Overview of Real-time Systems

- A **real-time system** requires that results be produced within a specified deadline period.
- An **embedded system** is a computing device that is part of a larger system (i.e. automobile, airliner.)
- A **safety-critical system** is a real-time system with catastrophic results in case of failure.
- A **hard real-time system** guarantees that real-time tasks be completed within their required deadlines.
- A **soft real-time system** provides priority of real-time tasks over non real-time tasks.
System Characteristics

- Single purpose
- Small size
- Inexpensively mass-produced
- Specific timing requirements
System-on-Chip (SoC)

- Many real-time systems are designed using system-on-chip designs.
- SoC enables the CPU, memory, memory-management unit, and attached peripheral ports (e.g., USB) to be contained in a single integrated circuit.
Features of Real-time Kernels

- Most real-time systems do not provide the features found in a standard desktop system.
- Reasons include
  - Real-time systems are typically single-purpose.
  - Real-time systems often do not require interfacing with a user.
  - Features found in a desktop PC require more substantial hardware that what is typically available in a real-time system.
Virtual Memory in Real-time Systems

- Address translation may occur via:
  1. **Real-addressing mode** where programs generate actual physical addresses.
  2. **Relocation** register mode.
  3. Implementing full **virtual memory**.

- Most embedded real-time systems use option 1 above; option 3 is almost **NEVER** used.
Address Translation

![Diagram of address translation process]

- CPU
- L
- relocation register R
- P = L
- TLB
- page table
- physical memory
- process A
- process B
- kernel
Implementing Real-time Operating Systems

- In general, real-time operating systems must provide:
  - Preemptive, priority-based scheduling
  - Preemptive kernels
  - Latency must be minimized
Minimizing Latency

- **Event latency** is the amount of time from when an event occurs to when it is serviced.

![Diagram showing event latency](image-url)
Interrupt Latency

- **Interrupt latency** is the period of time from when an interrupt arrives at the CPU to when it is serviced.
Dispatch Latency

- **Dispatch latency** is the amount of time required for the scheduler to stop one process and start another.
Real-time CPU Scheduling

- Periodic processes require the CPU at specified intervals (periods)
- $p$ is the duration of the period
- $d$ is the deadline by when the process must be serviced
- $t$ is the processing time
Typical Real-Time Applications

- Divide RT applications into the following four types according to their timing attributes:
  - Purely cyclic: every task executes periodically; its demands in (computing, communication, and storage) resources do not vary significantly from period to period.
  - Mostly cyclic: most tasks execute periodically; the system must also respond to some external events (fault recovery and external commands) asynchronously.
  - Asynchronous and somewhat predictable: most tasks are not periodic; the durations between consecutive executions of a task may vary considerably, or the variations in resource utilization in different periods may be large; these variations have either bounded ranges or known statistics.
  - Asynchronous and unpredictable: applications that react to asynchronous events and have tasks with high run-time complexity.
Typical Real-Time Applications

- Examples
  - Purely cyclic – most digital controllers and real-time monitors
  - Mostly cyclic – modern avionics and process control systems
  - Asynchronous, predictable –
  - Asynchronous, unpredictable – intelligent real-time control systems
Definitions

- Each unit of work that is scheduled and executed by a system is a **job** – e.g. computation of a control-law, computation of an FFT on sensor data, transmission of a data packet, retrieval of a file

- A set of related jobs which jointly provide some system function is a **task** – e.g. the set of jobs that constitute the “maintain constant altitude” task – i.e. keep an airplane flying at a constant altitude

- A job executes or is executed by the operating system

- Every job executes on a processor
Definitions

- **Release time** – the instant of time at which a job becomes available for execution
- A job can be scheduled and executed at any time at or after its release time
- If all jobs are released when the system begins execution, then these jobs “have no release time”
- **Deadline** – the instant of time by which a job’s execution is required to be completed (also called *absolute deadline*)
- **Response time** – the length of time from the release time of the job to the instant when it completes
- **Relative deadline** – the maximum allowable response time of a job
  - absolute deadline = release time + relative deadline
- **Completion time** – the instant of time at which a job completes execution
- **Timing constraint** – any constraint imposed on the timing behaviour of a job
Temporal parameters

Many parameters of hard RT jobs and tasks are known at all times – otherwise, we cannot ensure that the system meets its hard RT requirements

- The number of hard RT tasks or jobs – a hard RT system may operate in different modes – the number of tasks/jobs is known for each mode – e.g. autopilot system is changed to standby
- Each job $J_i$ is characterized by its temporal (timing constraints) and functional (intrinsic properties of the job) parameters
A Reference Model of Real-Time Systems

- Temporal concepts
  - $r_i$ – release time of $J_i$
  - $d_i$ – absolute deadline of $J_i$
  - $D_i$ – relative deadline of $J_i$
  - $(r_i, d_i]$ – feasible interval for $J_i$
  - Often do not know exactly when a job is released, only that $r_i$ is in a range $[r_i^-, r_i^+]$ – this is known as release time jitter
  - If, for all practical purposes, we can approximate the actual release time of each job by its earliest or latest release time, then we say that the job has a fixed release time
A Reference Model of Real-Time Systems

- Execution time
  - $e_i$ is the execution time for $J_i$ – i.e. the amount of time required to complete the execution of $J_i$ when it executes alone and has all the resources it requires.
  - Value depends upon the complexity of the job and the speed of the processor upon which it is scheduled.
  - Execution time may vary for a variety of reasons:
    - Conditional branches
    - Cache memories and/or pipelines
    - Compression (e.g. MPEG video frames)
  - As for release time, usually we know $e_i$ is in the range $[e_i^-, e_i^+]$; we usually assume that we know this range for every hard RT job.
  - Often, we can validate a system by knowing $e_i^+$ for each job; therefore, $e_i$ often implies the maximum execution time.
A Reference Model of Real-Time Systems

- Periodic task model
  - Each computation or data transmission that is executed repeatedly at regular or semi-regular time intervals is modelled as a periodic task.
  - Each periodic task $T_i$ is a sequence of jobs.
  - The period $p_i$ of $T_i$ is the minimum length of all time intervals between release times of consecutive jobs.
  - The execution time $e_i$ of $T_i$ is the maximum of all jobs in the periodic task.
  - The period and execution time of every periodic task in the system are known at all times.
  - The accuracy of the periodic task model decreases with increasing release jitter and variations in execution times.
A Reference Model of Real-Time Systems

- Periodic task model
  - Individual jobs in $T_i$ are referred to as $J_{i,1}, J_{i,2}, \ldots$
  - The release time $r_{i,1}$ of $J_{i,1}$ in each task $T_i$ is called the phase of $T_i$, $\phi_i$
  - The hyperperiod $H$ of a set of periodic tasks is the least common multiple of $p_i$ for $i = 1 \ldots N$
  - The ratio $u_i = e_i/p_i$ is the utilization of task $T_i$ – i.e. the fraction of time a truly periodic task with period $p_i$ and execution time $e_i$ keeps a processor busy
  - Total utilization $U = \Sigma u_i$, where the sum is over all periodic tasks in the system
  - The relative deadline, $D_i$, is often simply the period, $p_i$
A Reference Model of Real-Time Systems

- Scheduling
  - Jobs are scheduled according to a chosen set of scheduling algorithms; a scheduler implements these algorithms
  - A scheduler specifically assigns jobs to processors
  - A schedule is an assignment of all jobs in the system on the available processors.
  - A **valid schedule** satisfies the following conditions:
    - Every processor is assigned to at most one job at any time
    - Every job is assigned at most one processor at any time
    - No job is scheduled before its release time
    - The total amount of processor time assigned to every job is equal to its maximum or actual execution time
A Reference Model of Real-Time Systems

• Scheduling
  ◦ A valid schedule is a **feasible schedule** if every job meets its timing constraints.
  ◦ A hard real time scheduling algorithm is **optimal** if the algorithm always produces a feasible schedule if a given set of jobs has a feasible schedule.
  ◦ lateness = completion time – deadline
tardiness = max[0, lateness]
  ◦ Miss rate – the percentage of jobs that are executed but completed too late
  ◦ Loss rate – the percentage of jobs that are not executed at all
Priority-driven Approach

- Priority-driven algorithms NEVER intentionally leave any resource idle.
- Scheduling decisions are made when events such as releases and job completions occur; hence, such algorithms are **event-driven**
- Also called greedy scheduling (makes locally optimal decisions)
- Locally optimal scheduling decisions are often NOT globally optimal
Priority-driven Approach

Most scheduling algorithms used in non real-time systems are priority-driven

- First-In-First-Out (Based upon release times)
- Last-In-First-Out
- Shortest-Execution-Time-First (Based upon execution times)
- Longest-Execution-Time-First
Priority-driven Scheduling of Periodic Tasks

- Focus on well-known priority-driven algorithms for scheduling periodic tasks on a processor
- Assume a restricted periodic task model
  1. The tasks are independent
  2. There are no aperiodic or sporadic tasks
- Other assumptions made:
  1. Every job is:
     - Ready for execution as soon as it is released
     - Can be preempted at any time
     - Never suspends itself
  2. Scheduling decisions are made immediately upon job releases and completions
  3. Context switch overhead is negligibly small compared with execution times of the jobs
  4. The number of priority levels is unlimited
Priority-driven Scheduling of Periodic Tasks

- Additional assumptions
  - The period of a task means the minimum interrelease time of jobs in the task
  - A fixed number of periodic tasks
  - The addition of another task to the system requires the scheduler to perform an acceptance test – i.e. the task will be added to the system only if the new task and all other existing tasks can be feasibly scheduled
  - Focus on scheduling on uniprocessor systems

- Recall
  - Priority-driven algorithms NEVER intentionally leave any resource idle.
  - Scheduling decisions are made when events such as releases and job completions occur; hence, such algorithms are **event-driven**
  - Locally optimal scheduling decisions are often NOT globally optimal
Priority-driven Scheduling of Periodic Tasks

- Fixed-priority vs. Dynamic-priority Algorithms
  - A priority-driven scheduler is an on-line scheduler
    - It does NOT precompute a schedule of tasks/jobs
    - It assigns priorities to jobs when they are released and places them on a ready job queue in priority order
    - When preemption is allowed, a scheduling decision is made whenever a job is released or completed
    - At each scheduling decision time, the scheduler updates the ready job queue and then schedules and executes the job at the head of the queue
  - A fixed-priority algorithm assigns the same priority to all the jobs in a task.
  - A dynamic-priority algorithm assigns different priorities to the individual jobs in a task.
  - The priority of a job is usually assigned upon its release and does not change
  - Three categories of algorithms:
    - Task-level fixed-priority
    - Task-level dynamic-priority and job-level fixed-priority
    - Job-level dynamic-priority
Priority-driven Scheduling of Periodic Tasks

Fixed-priority Algorithms

• Rate-monotonic algorithm (RM)
  ◦ Assigns priorities to tasks based on their periods – the shorter the period, the higher the priority
  ◦ Since the rate is \((\text{period})^{-1}\), the higher the rate, the higher the priority

• Deadline-monotonic algorithm (DM)
  ◦ Assigns priorities to tasks according to their relative deadlines – the shorter the relative deadline, the higher the priority

• When the relative deadline of every task is proportional to its period, the RM and DM algorithms give identical results

• When the relative deadlines are arbitrary, the DM algorithm performs better in the sense that it can sometimes produce a feasible schedule when RM fails, while RM always fails when DM fails
Priority-driven Scheduling of Periodic Tasks

- Rate-monotonic example, $T = (φ, p, e[, D])$
  - $T_1 = (0, 4, 1); T_2 = (0, 5, 2); T_3 = (0, 20, 5)$
  - Relative priorities: $T_1 > T_2 > T_3$

<table>
<thead>
<tr>
<th>Time</th>
<th>Ready to run</th>
<th>Scheduled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$J_{1,1}; J_{2,1}; J_{3,1}$</td>
<td>$J_{1,1}$</td>
</tr>
<tr>
<td>1</td>
<td>$J_{2,1}; J_{3,1}$</td>
<td>$J_{2,1}$</td>
</tr>
<tr>
<td>3</td>
<td>$J_{3,1}$</td>
<td>$J_{3,1}$</td>
</tr>
<tr>
<td>4</td>
<td>$J_{1,2}; J_{3,1}$</td>
<td>$J_{1,2}$</td>
</tr>
<tr>
<td>5</td>
<td>$J_{2,2}; J_{3,1}$</td>
<td>$J_{2,2}$</td>
</tr>
<tr>
<td>7</td>
<td>$J_{3,1}$</td>
<td>$J_{3,1}$</td>
</tr>
<tr>
<td>8</td>
<td>$J_{1,3}; J_{3,1}$</td>
<td>$J_{1,3}$</td>
</tr>
</tbody>
</table>
Priority-driven Scheduling of Periodic Tasks

<table>
<thead>
<tr>
<th>Time</th>
<th>Ready to run</th>
<th>Scheduled</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>J_{3,1}</td>
<td>J_{3,1}</td>
</tr>
<tr>
<td>10</td>
<td>J_{2,3} ; J_{3,1}</td>
<td>J_{2,3}</td>
</tr>
<tr>
<td>12</td>
<td>J_{1,4} ; J_{3,1}</td>
<td>J_{1,4}</td>
</tr>
<tr>
<td>13</td>
<td>J_{3,1}</td>
<td>J_{3,1}</td>
</tr>
<tr>
<td>15</td>
<td>J_{2,4}</td>
<td>J_{2,4}</td>
</tr>
<tr>
<td>16</td>
<td>J_{1,5} ; J_{2,4}</td>
<td>J_{1,5}</td>
</tr>
<tr>
<td>17</td>
<td>J_{2,4}</td>
<td>J_{2,4}</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Priority-driven Scheduling of Periodic Tasks

Dynamic-priority algorithms

- **Earliest-deadline-first (EDF)**
  - The job queue is ordered by earliest deadline

- **Least-slack-time-first (LST)**
  - The job queue is ordered by least slack time
  - Nonstrict – scheduling decisions are made only when jobs are released or completed
  - Strict – scheduling decisions are made also whenever a queued job’s slack time becomes smaller than the executing job’s slack time – huge overheads, not used

- **First-in-first-out (FIFO)**
  - Job queue is first-in-first-out by release time

- **Last-in-first-out (LIFO)**
  - Job queue is last-in-first-out by release time
Priority-driven Scheduling of Periodic Tasks

- Compare RM, EDF, LST, FIFO
- \( T_1 = (0, 2, 1); T_2 = (0, 5, 2.5) \)
- The total utilization is 1.0
- Expect some of these algorithms to lead to missed deadlines!
Priority-driven Scheduling of Periodic Tasks

<table>
<thead>
<tr>
<th>Rate-monotonic</th>
<th>Earliest-deadline-first</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(J_{1,1}; J_{2,1})</td>
</tr>
<tr>
<td>1</td>
<td>(J_{2,1})</td>
</tr>
<tr>
<td>2</td>
<td>(J_{1,2}; J_{2,1})</td>
</tr>
<tr>
<td>3</td>
<td>(J_{2,1})</td>
</tr>
<tr>
<td>4</td>
<td>(J_{1,3}; J_{2,1})</td>
</tr>
<tr>
<td>5</td>
<td>(J_{2,1}; J_{2,2})</td>
</tr>
<tr>
<td>5.5</td>
<td>(J_{2,2})</td>
</tr>
<tr>
<td>6</td>
<td>(J_{1,4}; J_{2,2})</td>
</tr>
<tr>
<td>7</td>
<td>(J_{2,2})</td>
</tr>
<tr>
<td>8</td>
<td>(J_{1,5}; J_{2,2})</td>
</tr>
<tr>
<td>9</td>
<td>(J_{2,2})</td>
</tr>
</tbody>
</table>

\(J_{x,y}\) denotes the task with priority \(x\) and ID \(y\).
# Priority-driven Scheduling of Periodic Tasks

<table>
<thead>
<tr>
<th>Least-sack-time-first</th>
<th>First-in-first-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>J₁,₁[1]; J₂,₁[2.5]</td>
</tr>
<tr>
<td>1</td>
<td>J₂,₁[1.5]</td>
</tr>
<tr>
<td>2</td>
<td>J₁,₂[1]; J₂,₁[1.5]</td>
</tr>
<tr>
<td>3</td>
<td>J₂,₁[0.5]</td>
</tr>
<tr>
<td>4</td>
<td>J₂,₁[0.5]; J₁,₃[1]</td>
</tr>
<tr>
<td>4.5</td>
<td>J₁,₃[0.5]</td>
</tr>
<tr>
<td>5</td>
<td>J₁,₃[0.5]; J₂,₂[2.5]</td>
</tr>
<tr>
<td>5.5</td>
<td>J₂,₂[2]</td>
</tr>
<tr>
<td>6</td>
<td>J₁,₄[1]; J₂,₂[2]</td>
</tr>
<tr>
<td>7</td>
<td>J₂,₂[1]</td>
</tr>
<tr>
<td>8</td>
<td>J₁,₅[1]; J₂,₂[1]</td>
</tr>
<tr>
<td>9</td>
<td>J₂,₂[0]</td>
</tr>
</tbody>
</table>
Priority-driven Scheduling of Periodic Tasks

- **FIFO**
  - J_{1,1} | J_{2,1} | J_{1,2} | J_{1,3} | J_{2,2} | J_{1,4} | J_{1,5}

- **LST**
  - J_{1,1} | J_{2,1} | J_{1,2} | J_{2,1} | J_{2,1} | J_{1,3} | J_{2,2} | J_{1,4} | J_{2,2} | J_{1,5} | J_{2,2}

- **EDF**
  - J_{1,1} | J_{2,1} | J_{1,2} | J_{2,1} | J_{2,1} | J_{1,3} | J_{2,2} | J_{1,4} | J_{2,2} | J_{1,5} | J_{2,2}

- **RM**
  - J_{1,1} | J_{2,1} | J_{1,2} | J_{2,1} | J_{1,3} | J_{2,1} | J_{2,2} | J_{1,4} | J_{2,2} | J_{1,5} | J_{2,2}
Priority-driven Scheduling of Periodic Tasks

- Relative merits
  - Algorithms that do not take into account the urgencies of jobs in priority assignment usually perform poorly (FIFO, LIFO)
  - Algorithms are ranked by their ability to maximize the utilization of the system in terms of meeting job deadlines – maximum value of 1 – EDF is optimal in this sense, while RM and DM are not
  - EDF continues to give high priority to jobs that have already missed their deadlines relative to a job whose deadline is in the future; therefore, EDF is not particularly suitable to systems where overload conditions are unavoidable
Priority-driven Scheduling of Periodic Tasks

- Recall that a periodic task $T_i$ is defined by the 4-tuple $(\phi_i, p_i, e_i, D_i)$ and that the utilization of $T_i$, $u_i = e_i / p_i$
- For a system of periodic tasks $\mathcal{T} = \{T_i, i = 1..n\}$, the total utilization, $U(\mathcal{T}) = u_1 + u_2 + \ldots + u_n$
- A scheduling algorithm can feasibly schedule any system $\mathcal{T}$ of periodic tasks on a processor if $U(\mathcal{T})$ is equal to or less than the schedulable utilization of the algorithm
- Schedulable utilization is an attribute of a scheduling algorithm; we will denote it as $U_{\text{ALG}}$
- If $U_{\text{ALG}} = 1$, the algorithm is optimal
- Why is the knowledge of $U_{\text{ALG}}$ important?
  - Validation of your system then depends upon simply showing that for your system $\mathcal{T}$, $U(\mathcal{T}) \leq U_{\text{ALG}}$
Priority-driven Scheduling of Periodic Tasks

- The schedulable utilization of the EDF algorithm
  - A system $\mathcal{T} = \{T_i, i = 1..n\}$ of independent, preemptable periodic tasks with $D_i = p_i$ can be feasibly scheduled on one processor if and only if $U(\mathcal{T}) \leq 1$.
  - Corollaries:
    - $U_{EDF}(n)$ for $n$ independent, preemptable periodic tasks with $D_i \geq p_i$ is 1.
    - $U_{LST}(n)$ for $n$ independent, preemptable periodic tasks with $D_i \geq p_i$ is 1.
  - Note that all of these results are independent of $\phi_i$
Priority-driven Scheduling of Periodic Tasks

- Schedulability testing
  - A test for the purpose of validating that the given application system meets all its hard deadlines when scheduled according to a particular scheduling algorithm is a **schedulability test**.
  - If a schedulability test is efficient, then it can be used as an on-line acceptance test.

- Schedulability test for EDF
  - $U(T) \leq 1$
  - This test is robust – i.e. the test holds true if some jobs execute for less than their maximum execution times; it also holds true if the interrelease times of jobs in a task are longer than the period (minimum interrelease time).
Priority-driven Scheduling of Periodic Tasks

- How can you use this in practice?
  - Assume that you have decided to use EDF to schedule multiple periodic tasks
    - You know the execution times for jobs in each periodic task
    - You can choose the periods for the tasks such that the schedulability test is met
  - Example – digital robot controller
    - Control-law computation takes 8 ms
    - Desired sampling rate is 100 Hz => period is 10 ms => utilization is .8
    - Suppose that a built-in self test task is to be included, and that the computation of the test takes 50 ms; schedulability test tells us that as long as the period for this task is 250 ms or more, the total utilization remains ≤ 1; if a period of 1 sec is chosen, utilization is 0.05
    - If we now wish to add another task, as long as its utilization is ≤ 0.15, the system is still feasible
Priority-driven Scheduling of Periodic Tasks

- Optimality of the RM and DM algorithms
  - We’ve already seen examples wherein these fixed-priority algorithms are not optimal
  - In fact, if the periods of the tasks in the system are related appropriately, then the RM and DM algorithms are optimal
  - A system of periodic tasks is **simply periodic** if for every pair of tasks \( T_i \) and \( T_k \) in the system and \( p_i < p_k \), \( p_k \) is an integer multiple of \( p_i \).
  - Theorem: A system of simply periodic, independent, preemptable tasks whose relative deadlines are \( \geq \) their periods is schedulable on one processor according to the RM algorithm iff its total utilization is \( \leq 1 \).
  - Corollary: The same is true for the DM algorithm.
  - Since fixed-priority algorithms are more constrained, why would one choose to use them?
    - They often lead to more predictable and stable systems
Priority-driven Scheduling of Periodic Tasks

- Schedulable utilization of the RM algorithm
  - Assume $D_i = p_i$ for all $I$
  - Arbitrary relationships between relative deadlines of tasks
  - $U_{RM}(n) = n \left( 2^{1/n} - 1 \right)$
  - For large $n$, approaches $\ln 2$ (0.693)
  - $U(\mathcal{T}) \leq U_{RM}(n)$ is a sufficient condition – i.e. if $U(\mathcal{T}) > U_{RM}(n)$, the RM algorithm (or better yet, the DM algorithm) may be able to find a feasible schedule

<table>
<thead>
<tr>
<th>$n$</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>18</th>
<th>20</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_{RM}(n)$</td>
<td>0.828</td>
<td>0.757</td>
<td>0.735</td>
<td>0.724</td>
<td>0.718</td>
<td>0.714</td>
<td>0.710</td>
<td>0.708</td>
<td>0.707</td>
<td>0.705</td>
<td>0.696</td>
</tr>
</tbody>
</table>