Real-Time Systems

Steve Goddard
goddard@cse.unl.edu

Real-Time Systems

Conventional programming model
- Independent processes using independent processors
- Computation time does not affect correctness

Why was this model created?
- Simplified sharing a physical machine among many computations
- Improved average-case performance
- Simplified implementing operating system

http://www.cse.unl.edu/~goddard/Courses/CSCE855
Real-Time Systems

- Real-Time systems have different goals, require different assumptions, producing different designs & implementations
- In a real-time system, when the answer is produced is part of the answer’s correctness
  - Example: Can an air-traffic control solution created after the plane has crashed be correct?
    - Time dependent computation semantics
- Real-time computations must produce solutions which are logically correct and timely
  - Timeliness means completion by a deadline

Key Design Question:
- Must this system be able to guarantee that one or more computations will complete by a deadline
  - NO
  - No problem, this is not a real-time system
  - Conventional designs and approaches apply
- YES
  - OK, but we have a problem because guaranteeing that computations will complete by deadlines depends on worst case assumptions and behaviors
Real-Time Systems
Assumption Assault

♦ Decades of hardware and software design decisions have used average case performance metrics
  » Bad: explicit design assumptions can become invalid
  » Worse: implicit design assumptions can become invalid

♦ Explicit
  » Computation time doesn’t matter
  » Now we need to know worst case execution time
  » Computations can be treated independently
    ♦ They affect one another’s completion time
  » Algorithms treating computations fairly are good
    ♦ Unfair is preferred if it increases deadline satisfaction

♦ Implicit
  » Caching is good
    ♦ Not if it decreases average access time by increasing worst case access time
Real-Time Systems
Requirements

◆ Timeliness
  » System must ensure that real-time tasks satisfy their deadlines
◆ Simultaneity
  » More than one event may occur at the same time
  » Deadlines of computations serving events must be met
◆ Predictability
  » Real-time system must service all events predictably
◆ Adaptability to handle
  » Increased load (short term state changes)
  » Configuration changes (long term)

Real-Time Systems
Computation Characteristics

◆ Resource Use
  » CPU
  » Shared resources implying execution constraints
◆ Precedence Relations
  » Among components of a computation
◆ Concurrency Constraints
  » Arising from resource use or precedence relations
  » Should permit maximum concurrency
◆ Communication Relations
  » Time constrained communication among computations and components → precedence relations
Real-Time Systems
Classification

Computation Characteristics

- Importance
  - Different tasks have different levels of importance
  - Application semantics’ influence on scheduling
- Fault tolerance
  - Critical tasks must be fault tolerant
  - How critical and how tolerant must be specified and then dealt with appropriately
- Placement constraints
  - Hardware dependencies for device control
  - Separation on different HW elements for fault tolerance

Classification

- Deadline Classification
  - Hard: infinite cost for a missed deadline
  - Soft: non-zero but tolerable cost
  - Firm: non-zero and less tolerable cost
- Periodic/Aperiodic
  - Can a computation be handled with periodic attention
- Event triggered vs. Time triggered
  - Is the system best described as a set of computations scheduled at particular times or computations executing in response to external events
**Hard Real-Time**
- HRT computation failure causes terrible consequences
  - Air traffic control, fly-by-wire, machine controllers
  - Late results are useless
- Often low level operations and combined with fault tolerance requirements
- Often designed as separate components of a distributed system to simplify analysis
  - Isolate HRT components on dedicated resources
- Design Challenge
  - Correctly distinguish hard from not-so-hard computations
  - Redesign components to reduces “hardness”

**Soft Real-Time**
- Much less obvious temporal constraints
  - More complex cost/benefit tradeoff
  - “Fast Enough” is often heard but is not specific enough
- Rising cost (decreasing value) with lateness
  - Deadline violation rate
  - Value function: value of completed computation
- Examples: Vending Machines, Transaction Servers
- Continuum with “fast” conventional systems
- Often created by adding time-aware scheduling to a conventional system
  - Limited value when many sub-systems are designed for the average case (Solaris)
**Firm Real-Time**

- Emerging and growing class of systems
  - Most deadlines must be met accurately
  - Occasional misses can be handled
  - Fail-safe computation semantics required
- HRT/SRT compromise
  - Intermediate time constraint granularity
  - Intermediate deadline violation tolerance
- Examples
  - Video on demand and Multi-Media conferencing
  - Multi-player gaming
  - Automated manufacturing

**RT System Characteristics**

- Often a mixed set of computations (hard, firm, soft)
  - One reason for distribution or Multi-CPU
- RT used to be limited to embedded applications
  - No longer
- Often motivated by desire to use a single CPU to support more than one computation
  - Move beyond embedded/dedicated model
  - How many CPUs are in a high end BMW?
    - 55 in 1990 (one for each wheel in ABS)
    - Move to shared bus and multi-processor architecture
    - Wiring cost more important than CPU cost
RT System Characteristics

- Fast Context Switch
  » Low system overhead

- Small size
  » Embedded application influence

- Minimal Functionality
  » Traditionally accepted to achieve small size
  » Generalizes to “configurable” OS where developer includes abilities required, leaving others out

- Fast Interrupt Service
  » Desired for typical embedded control applications
  » Generalizes to low latency event service

RT System Characteristics

- No Virtual Memory
  » Traditional for cost and speed
  » Combines VM and Logical Address Space concepts
  » No page faults makes sense
  » No MMU is not as sensible
    » MMUs now are cheap

- Able to lock code and data in memory
  » Related to no VM
  » Eliminates unpredictable page-fault latency

- Real-Time Clock
  » User computations often use absolute and elapsed time
RT System Characteristics

- System provides alarms and timeouts
  - User interface for the system’s real-time clock
- Tasks interface to describe scheduling requirements
- Traditional RT systems used methods which must become
  - More adaptive
  - More scalable
  - More complex
  - More dynamic
  - More distributed
- Major growth and employment opportunity

Misconceptions

- Sometimes arise from “sticker shock”
  - Profound nature and extent of changes required
  - Requirements can suddenly change when cost is known
- Real-time is about device drivers in assembly language on bare processors
  - Many years ago this was true
  - Real-time constraints are arising in a wide range of applications and device drivers now live inside systems
- Real-time is the same as fast
  - Must be able to predict behavior to guarantee a deadline
  - Fast computers often work OK for the wrong reasons
**Misconceptions**

- All I have to do is buy a fast enough computer
  - People (and managers) often want to simplify by drowning a problem in CPU cycles
  - Sometimes works
  - Leaves the system brittle since it can stop working abruptly and catastrophically if things change
    - Without deadline awareness everything can be late
  - Never a substitute for thought and understanding of the problem

- There will always be a fast enough computer
  - There are always problems where adequate resources exist without a sufficient surplus to permit sloppiness
    - Corollary: using existing resources well can often reveal a wide variety of new possibilities
  - We should get it working *logically* first and then worry about how fast it is
    - Evil - even backwards
    - Temporal constraints must be considered as first class design constraints
    - Otherwise many average case vs. worst case assumptions and vulnerabilities will creep in
**Misconceptions**

- Real-Time systems cannot use MMUs
  - Embedded systems traditionally use CPUs with extra device control, timers, and other features without MMU
  - Crucial distinction between VM and LM
    - Page faults are unpredictable and huge
  - Logical $\rightarrow$ Physical address mapping can be done predictably
    - Explicitly manage the TLB
    - “Innovation” in real-time systems
    - Process compilation and protection simplifications
  - Current RT systems commonly have a single huge physical address space $\rightarrow$ no protection

**Scheduling**

- Goal: Organize process (task) execution so that each completes before its deadline
- Notice that this is a difference performance metric than
  - Throughput
  - Fairness
  - Average response time
- Must consider: deadline, precedence, resource use
- Processor utilization is still an issue but we often must tolerate lower levels to ensure guarantees
  - Code (almost) never follows the worst case path
Scheduling

- Value or penalty function is often used (at least conceptually) to decrease task value after a deadline
  - HRT: step function
  - SRT: gentle slope
  - FRT: steep slope and often more complex constraints
- Miss deadlines of no more than 1 in N iterations
- The function describes how the "value" of completing a computation varies with time
  - Describe several important characteristics of a task

Scheduling Value Functions

- Hard RT
  - Decay period 0 and decays to 0
- Soft → Firm RT
  - Decay period extended and decays to 0

![Graph showing the decay period of a task value function](image-url)
Scheduling
Value Functions
◆ Theoretically we could use complex value functions
◆ Scheduler would have the job of maximizing the “value” produced by the system within various periods
◆ Classic Design Scenario
  » Theoretically attractive
  » Impractical for several reasons
◆ Problems
  » Value functions become too elaborate and expensive
  » Scheduler takes too long to evaluate situation
◆ Classic solution: Simple is better

Scheduling
◆ Schedulers assume some set of information about tasks
  » Deadline
  » WCET
  » Resource use (shared, exclusive)
  » Communication and precedence relations
Scheduling

- Scheduler characteristics
  - Preemptive and non-preemptive
  - Static and Dynamic
  - Centralized and Distributed

- Popular Methods
  - Earliest Deadline First (EDF)
  - Rate Monotonic
  - Explicit Plan

Preemptive vs. Non-Preemptive

- Can the execution of a task be stopped and restarted
- Preemption stops one process and starts another
  - This is the behavior assumption of a conventional OS
  - Usually done at I/O operations but also at time quantum
  - Consistent with “virtual time” assumption

- Consider resource use and synchronization
  - Preemption while holding a resource leaves it locked

- Good idea for average case behavior and fairness but RT systems do not care about average case or fairness
  - Still a good idea sometimes but care is required
  - Some task sets can only be scheduled preemptively
Preemptive vs. Non-Preemptive

- Generally, the highest priority task is run
- If a higher priority task arrives or makes the state transition Blocked → Runnable
  - Current lower priority task is preempted
- Running → Runnable
- Preempted tasks continue to hold all resources
- Scheduling decision is thus reduced to selecting the runnable process with the highest priority
  - O(N) operation to select maximum (best) value
  - Assumes a total order on the set of processes
- Attractive because it is familiar and simple
  - How do we know how to assign the priorities?

Schedulability

- RT system designers must constantly ask and answer:
  - Can this system meet all of its constraints?
- Conventional system designers do not face this question because execution time is not part of correctness
- It is for RT systems
  - Example: Event requiring 50 ms execution time occurs 30 times per second (33.3 ms period)
  - Get a (much) faster CPU
- This depends on the notion of guarantee
  - Must have sufficient CPU and other resources to meet worst case behavior
Schedulability

- Basic relationship makes the calculation on CPU cycles
  - Every task \( T_i \) has a period \( P_i \) and a computation time \( C_i \)
  - Utilization (\( \mu \)) of the processor(s) must be feasible
  - CPU utilization of a single task \( T_i \) is: \( \frac{C_i}{P_i} \)
  - For a set of \( m \) tasks on \( N \) processors satisfaction of the following equation is a necessary but not sufficient condition:
    \[
    \mu = \frac{\sum_{i=1}^{m} \frac{C_i}{P_i}}{N} \leq 1
    \]

- Preemption may be required
  - Consider a simple set of three tasks \( T_1, T_2, \) and \( T_3 \)
  - Assume that \( P_1 = 2P_2 = 4P_3 \)
    - This means that \( T_2 \) executes twice for every execution of \( T_1 \)
    - and \( T_3 \) executes four times for every execution of \( T_1 \)
  - Now consider what happens if:
    \( C_i > P_i - C_j - 2C_j \)
  - The task set is not schedulable unless the execution of \( T_i \) is split into two pieces through preemption
    - Because \( T_j \) cannot complete execution before \( T_i \) must begin executing again
**Schedulability**

- Note that this analysis provides a *lower bound* on the CPU resources required to support a task set.
- Ignores many sources of overhead, delay, and other constraints on scheduling:
  - Context switching
  - Interrupt service routines not associated with a task
  - Message transmission latency
  - Resource use
- Some increase CPU requirements, others constrain the minimum period of some computations
  - Constraints can be subtle

**Dynamic vs. Static**

- Dynamic scheduling algorithms make decisions at run time
- Static algorithms simply consult a predefined table to determine task context switches
  - Static algorithms clearly have lower overhead
- Conventional systems use priority driven preemptive dynamic scheduling with no priority re-computation
  - Familiar and very successful BUT
  - *Mechanism* not a *Policy*
- Static schedule satisfying all scheduling constraints
  - Is correct and sufficient
  - This is often lost in the complexity of design debates
Dynamic vs. Static

- Dynamic algorithms are familiar and attractive in theory because they are:
  - Simple
  - Provably optimal in uni-processor system
- They often do not take system overhead or resource use into account
  - When they do, they are not nearly as simple
- Common dynamic scheduling techniques include
  - Earliest Deadline First (EDF)
  - Least Laxity First (LLF)
  - Rate Monotonic (RM)

Optimality

- Important but dangerous term
  - Optimal means, colloquially, “as good as any and better than most”
  - No algorithm can produce better results
- Important questions
  - What is the performance metric?
    - Algorithms are optimal “with respect to” some measure
  - How much does this optimality cost?
  - How does it do with respect to other measures?
  - How close to optimal do simpler algorithms come?
  - How robust is the algorithm?
Earliest Deadline First (EDF)

- Simple and Fast
  - Keep a list of tasks sorted by deadline
  - Always run the task with the earliest (lowest) deadline
- Optimal for a single CPU and tasks with no ordering or mutual exclusion (exclusive resource use) constraints
  - Many RT systems meet these criteria
- Ignores context switching costs
- Brittle with respect to assumption violation
  - If any WCET or period assumption is violated the whole system can crash → no tasks meet their deadlines
  - Every task almost makes it

Least Laxity First (LLF)

- Also simple and fast
  - Laxity is the difference between the time remaining until the deadline and the computation time
  - Interesting because this metric combines aspects of deadline and computation time
  - Execute the task with least laxity at any given moment
- Optimal for single CPU and independent tasks
- Brittle
  - Assumption violation can leave all tasks almost finishing
  - When problems occur it can also be difficult to figure out why they happened → cascade failures
**Rate Monotonic (RM)**

- Classic result by Liu and Layland assigns priorities according to the task period
  - A task \( T_i \) has WCET \( C_i \) and a period \( P_i \)
  - Tasks with shorter periods get better priorities
- Result is classic because
  - Proved optimal for single CPU and independent tasks
  - Provides a utilization bound
    - Roughly 0.69 in theory but higher in practice
  - Uses familiar priority driven scheduling
- Brittle with respect to assumption violation
  - Difficult failure analysis and cascade failure

**Rate Monotonic (RM)**

- RM is among the most popular RT scheduling algorithms
  - Software Engineering Institute support and documentation
- Provides an easy way to adapt essentially conventional systems to real-time
- Important extensions for
  - Aperiodic event server
  - Handling tasks which use resources creating mutual exclusion scheduling constraints
  - Even distributed systems
- Good, popular, and has equations
  - Not a law of the universe
Rate Monotonic (RM)

- Resource use in real-time priority driven systems makes things more complicated
- Resource use in exclusive mode creates execution constraints which the priority driven scheduler cannot see
- Sharing of a mutual exclusion resource among tasks with different priorities can lead to priority inversion
  » A lower priority task can block the execution of a higher priority task
- Handling priority inversion substantially increases system complexity
  » Implementation, analysis, and performance evaluation

Rate Monotonic Priority Inversion Example

- Consider three tasks \( T_1, T_2, \) and \( T_3 \)
  » \( T_1 \) has the shortest period and thus the highest priority
  » \( T_3 \) has the longest period and thus the lowest priority
- \( T_1 \) and \( T_3 \) share a resource \( R \)
- \( T_2 \) holds \( R \) when \( T_2 \) becomes runnable
  » Scheduler preempts \( T_3 \) to execute \( T_2 \)
- \( T_1 \) then becomes runnable preempting \( T_2 \) but \( T_1 \) blocks when it tries to get \( R \) because \( T_3 \) still holds \( R \)
- \( T_1 \) blocking makes \( T_2 \) the highest priority process
  » \( T_2 \) thus keeps \( T_1 \) from running and thus freeing \( R \)
  » \( T_2 \) thus keeps \( T_1 \) from running \( \rightarrow \) Priority Inversion
Rate Monotonic 
Priority Inheritance

- Priority Inversion is handled by implementing priority inheritance
  - We assume we know resource use by each task
  - Preprocessing is performed on the set of tasks after priorities are assigned to determine what lower priority tasks can potentially block higher priority tasks
  - Table of resource priorities is constructed
  - Records highest priority use of each resource
  - System raises priority of a task to the resource priority while it uses the resource
  - Lower priority task inherits a higher priority
- Significantly complicates schedulability analysis

Explicit Plan Scheduling

- Classic scheduling algorithms are often called myopic because they make decisions based on limited information
  - They are nearsighted
- Important to realize that all scheduling algorithms are NP-Complete for multiple CPU/Distributed systems
  - Optimality and theoretical advantage evaporates
Explicit Plan Scheduling

- Simply pre-compute when tasks will execute
  » Ability to find such a schedule is not guaranteed
  » When you have one you are done
  » Searching for a feasible schedule is NP-Complete
  » Heuristics are used
- Plan can be constructed using any of a number of methods and can consider all task constraints
  » Resource use - mutual exclusion
  » Precedence Relations
  » Communication relations
  » Context switching and other system overhead

Disadvantage is that we have no guarantee that we can find a feasible schedule
  » Cannot distinguish infeasible task set from failure to find a feasible schedule
- More of a theoretical than a practical problem
  » Off-line schedule search task can run for a long time
Explicit Plan Scheduling

- Spring system at Umass-Amherst
  - Used explicit plan scheduling
    - Task precedence relations
    - Resource use (shared, exclusive)
    - Explicit delay
    - Communication relations
- Computations written as groups of interacting processes
  - Scheduled as sets of tasks with known WCET, resource use, precedence and communication relations
  - Compiler extensively analyzed process representation during compilation and constructed a task representation of the process execution time behavior

Explicit Plan Scheduling

- Less popular for no clear reason
  - Strength of CMU and SEI reputation and advocacy of RMS
  - Lure of mathematical analysis and optimality
    - Largely illusory
- Considerable duality in these methods
  - RM analysis effectively constructs a “worst case” execution plan
  - The task set is thus feasible even in the worst case
  - Texas Instruments then used this as an explicit schedule
- All explicit schedules satisfying execution constraints are solutions to the scheduling problem - regardless of source
Periodicity and Guarantees

- Mathematically based methods (RMS) are often popular because of perceived reliability and optimality
  - Often optimal in that they succeed if any method succeeds, not that they have the best CPU utilization
- All methods are based on behavioral assumptions
  - WCET
  - Period
  - Resource use
  - Communication patterns

Periodicity and Guarantees

- Customers, and designers, often want to combine issues
  - Guarantee and best effort
- Priority driven scheduling is attractive because it is familiar and because the highest priority task is always run
  - BUT: the guarantee of system correctness is based on an assumption about every process being periodic
  - Fairness and minimizing response time are not relevant
  - Executing every task according to the periodic assumptions must be OK or the analysis is bogus
Periodicity and Guarantees

- Many developers simplify their problem by providing periodic servers for all events
  - Then executing these according to a specific plan
  - Minimize aperiodic ISR execution time
- This approach must be OK or everything is nonsense

Language and Compiler Support

- All approaches to RT scheduling assume non-trivial information about tasks is available
  - WCET
  - Resource use
  - Precedence relations
  - Various attributes depending on scheduling method
- None of this information is known or used \textit{a priori} by conventional systems
Language and Compiler Support

- Language and compiler support are required to enable the compiler to provide required information about task behavior and to have that information be reliable
  » Reliable execution behavior predictions
- As RT constraints become more and more important to a wider range of applications the ability to express time and behavior constraints and to make predictions will become more and more important

Language and Compiler Support

- RT semantics are creeping into many applications without the developers or users realizing the implications
- CORBA researchers and developers are considering applications with RT constraints
  » Adding behavioral assertions and constraints to the IDL
- Opportunity because RT is likely to become important “suddenly” from the point of view of many industry segments and types of users
  » Those positioned to help will benefit greatly
Network Support

- Real-time applications are increasingly distributed
  » Distributed applications exhibit RT constraints with increasing frequency
- Network support is a component of distributed computations
  » Predictability of network behavior thus affects the predictability of computation behavior
- Networks are traditionally designed to reduce cost through
  » Statistical multiplexing
  » Probabilistic resource allocation → paradigm conflict
    - Significant source of difficulty
    - Providers are having figuring out how to support new services economically

Real-Time Communication

- Different from communication in other distributed systems
- High performance is nice, but predictability and determinism are required!
  » Ethernet does not provide a known upper bound on transmission time.
  » Token ring and Time Division Multiple Access (TDMA) protocols do.
Real-Time Communication

- Communication protocols are often very different from other distributed systems.
  - QoS specification is common
  - Time-Triggered Protocol (TTP)

- Unusual properties of TTP
  - Detection of a lost packet implies failed sender
  - CRC on the packet plus global state
  - Automatic group communication membership protocol
  - The way clock synchronization is achieved

Real-Time Communication

Time-Triggered Protocol

- Used in MARS real-time system
  - Consists of a single layer that handles
    - End-to-end data transport,
    - Clock synchronization, and
    - Membership management.

- All nodes are connected by two reliable and independent TDMA broadcast networks
- All packets are sent on both networks in parallel
- Expected loss rate is one packet every 30 million years!
Current Trends

◆ Time constraints are emerging in more and more areas
  » Not from specialized to general computations
  » But from general applications to real-time
◆ Distribution is becoming more and more common
◆ COTS hardware is developing such a dominant price/performance ration that it may dominate
  » wearables.stanford.edu
  » Matchbox size 66 MHz 486 w/16 MB
  » KU Real-Time modifications to Linux
◆ Distributed virtual environments and multimedia may be sufficient to drive networks toward RT - maybe not

Emerging Applications

◆ Time constrained transaction systems
◆ Multimedia
  » On-demand video/audio
  » Multi-media conferencing (harder because of lower latency constraint) → Games
◆ Smart appliances
◆ Complex distributed control
  » Houses, Cars
  » Traffic control
  » Cars, Trains, Ships, Planes, Elevators → turbo lifts
  » Aegis Cruisers