Real-Time System: Definition

A real-time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period.

- Correctness depends not only on the logical result (function) but also the time it was delivered.
- Failure to respond is as bad as delivering the wrong result!
Types of Real-Time Systems

- Hard real-time systems
- Weakly-hard real-time systems
- Firm real-time systems
- Soft real-time systems
- Best-effort systems

Real-time systems typically deal with deadlines:
- A deadline is a time instant by which a response has to be completed
- A deadline is usually specified as relative to an event
  - The relative deadline is the maximum allowable response time
  - Absolute deadline: event time + relative deadline

Hard Real-Time Systems

- Deadline miss is "catastrophic"
  - safety-critical system: failure results in death, severe injury
  - mission-critical system: failure results in massive financial damage
- Steep and real "cost" function

Soft Real-Time Systems

- Deadline miss is undesired but tolerable
  - Frequently results on quality-of-service (QoS) degradation
  - eg audio, video rendering
  - Steep "cost" function
- Cost of deadline miss may be abstract

Firm Real-Time Systems

- Deadline miss makes computation obsolete
  - Typical examples are forecast systems
    - weather forecast
    - trading systems
- Cost may be loss of revenue (gain)
Weakly-Hard Real-Time Systems

- Tolerate a (small) fraction of deadline misses
  - Most feedback control systems (including life-supporting ones!)
    - occasionally missed deadline can be compensated at next event
    - system becomes unstable if too many deadlines are missed
  - Typically integrated with other fault tolerance
    - electro-magnetic interference, other hardware issues

Best-Effort Systems

- No deadlines, timeliness is not part of required operation
- In reality, there is at least a nuissance factor to excessive duration
  - response time to user input
- Again, “cost” may be reduced gain

Real-Time Operating System (RTOS)

- Designed to support real-time operation
  - Fast context switches, fast interrupt handling?
  - Yes, but predictable response time is more important
    - “Real time is not real fast”
  - Analysis of worst-case execution time (WCET)
- Support for scheduling policies appropriate for real time
- Classical RTOSes very primitive
  - single-mode execution
  - no memory protection
  - essentially a scheduler with a threads package
  - “real-time executive”
- Many modern uses require actual OS technology for isolation
  - generally microkernels

Approaches to Real Time

- Clock-driven (cyclic)
  - Typical for control loops
  - Fixed order of actions, round-robin execution
  - Statically determined (static schedule)
    - need to know all execution parameters at system configuration time
- Event-driven
  - Typical for reactive systems (sensors & actuators)
  - Static or dynamic schedules
Real-Time System Operation

- **Event-triggered**
  - timer interrupt
  - asynchronous events

- **Time-triggered**
  - Pre-defined temporal relation of events
  - event is not serviced until its defined release time has arrived

- **Rate-based**
  - activities get assigned CPU shares (“rates”)

Real-Time Task Model

- **Job**: unit of work to be executed
  - … resulting from an event or time trigger

- **Task**: set of related jobs which provide some system function
  - A task is a sequence of jobs (typically executing same function)
  - Job i+1 of a task cannot start until job i is completed/aborted

- **Periodic tasks**
  - Time-driven and all relevant characteristics known a priori
    - Task t characterized by period T, deadline, D, and execution time C
    - Applies to all jobs of task

- **Aperiodic tasks**
  - Event driven, characteristics are not known a priori
    - Task t characterized by period T, deadline D, and arrival distribution

- **Sporadic tasks**
  - Aperiodic but with known minimum inter-arrival time T
  - treated similarly to periodic task with period T

Standard Task Model

- **C**: Worst-case computation time (WCET)
- **T**: Period (periodic) or minimum inter-arrival time (sporadic)
- **D**: Deadline (relative, frequently D=T)
- **J**: Release jitter
- **P**: Priority: higher number is higher priority
- **B**: Worst-case blocking time
- **R**: Worst-case response time
- **U**: Utilisation; U=C/T

Task Constraints

- **Deadline constraint**: must complete before deadline
- **Resource constraints**:
  - Shared (R/O), exclusive (W-X) access
  - Energy
  - Precedence constraints:
    - t1 ⇒ t2: execution cannot start until t1 is finished
  - Fault-tolerance requirements
    - eg redundancy

- **Scheduler’s job to ensure that constraints are met!**
Scheduling

- Preemptive vs non-preemptive
- Static (fixed, off-line) vs dynamic (on-line)
- Clock-driven vs priority-based
  - Clock-driven is static, only works for very simple systems
  - Priorities can be static (pre-computed and fixed) or dynamic
  - Dynamic priority adjustment can be at task-level (each job has fixed prio) or job-level (jobs change prio)

Clock-Driven (Time-Triggered) Scheduling

- Typically implemented as time “frames” adding up to “base rate”
- Advantages
  - Fully deterministic
  - “Cyclic executive” is trivial
    - Loop waiting for timer tick, followed by function calls to jobs
    - Minimal overhead
- Disadvantage:
  - Big latencies if event rate doesn’t match base rate (hyper-period)
  - Inflexible

Non-Preemptive Scheduling

- Minimises context-switching overhead
  - Significant cost on modern processors (pipelines, caches)
- Easy to analyse timeliness
- Drawbacks:
  - Larger response times for “important” tasks
  - Reduced utilisation, schedulability
    - In many cases cannot produce schedule despite plenty idle time
- Only used in very simple systems

Fixed-Priority Scheduling (FPS)

- Real-time priorities are absolute:
  - Scheduler always picks highest-priority job
- Fixed priorities obviously easy to implement, low overhead
- Drawbacks: inflexible, sub-optimal
  - Cannot schedule some systems which are schedulable preemptively
Rate-Monotonic (RM) Scheduling

- RM: Standard approach to fixed priority assignment
  - \( T_i < T_j \Rightarrow P_i > P_j \)
  - \( 1/T \) is the "rate" of a task
- RM is optimal (as far as fixed priorities go)
- Schedulability test: RM can schedule \( n \) tasks with \( D=T \) if
  \[
  U = \sum \frac{C_i}{T_i} \leq n(2^{1/n}-1)
  \]
  - sufficient but not necessary condition
- If \( D < T \) replace by deadline-monotonic (DM):
  - \( D_i < D_j \Rightarrow P_i > P_j \)
- DM is also optimal (but schedulability bound is more complex)

Earliest Deadline First (EDF)

- Dynamic scheduling policy
- Job with closest deadline executes
- Preemptive EDS with \( D=T \) is optimal: \( n \) jobs can be scheduled iff
  \[
  U = \sum \frac{C_i}{T_i} \leq 1
  \]
  - necessary and sufficient condition
  - no easy test if \( D \neq T \)
Overload: FPS vs EDF

On overload, (by definition!) \textit{lowest-prio jobs miss deadlines}

- Result is well-defined and -understood for FPS
  - Treats highest-prio task as “most important”
  - … but that may not always be appropriate!
  - Under transient overload may miss deadlines of higher-priority tasks

- Result is unpredictable (apparently random) for EDF
  - May result in all tasks missing deadlines!
  - Under constant overload will scale back all tasks
  - No concept of task “importance”
  - “EDF behaves badly under overload”
  - Main reason EDF is unpopular in industry

Why Have Overload?

- Faults (software, EMI, hardware)
- Incorrect assumptions about environment
- Optimistic WCET
  - Computing WCET of non-trivial programs is hard!
  - Safe WCET bounds tend to be highly pessimistic (orders of magnitude!)
  - WCET often very unlikely and orders of magnitude worse than “normal”
    - thanks to caches, pipelines, under-specified hardware
    - requires massive over-provisioning
  - Some systems have effectively unbounded execution time
    - e.g. object tracking
WCET Analysis

Why Have Overload?

- Faults (software, EMI, hardware)
- Incorrect assumptions about environment
- Optimistic WCET
  - Computing WCET of non-trivial programs is hard!
  - Safe WCET bounds tend to be highly pessimistic (orders of magnitude!)
  - WCET often very unlikely and orders of magnitude worse than “normal”
    - thanks to caches, pipelines, under-specified hardware
    - requires massive over-provisioning

Way out?

- Need explicit notion of importance: criticality
- Expresses effect of failure on the system mission
  - Catastrophic, hazardous, major, minor, no effect
- Orthogonal to scheduling priority

Mixed Criticality

- A mixed-criticality system supports multiple criticalities concurrently
  - Eg in avionics: consolidation of multiple functionalities
  - Higher criticality requires more pessimistic analysis, higher certification
  - Needs more than just scheduling support: strong OS-level isolation
- In overload scheduler drops lowest criticality
  - Current research issue

Must handle

<table>
<thead>
<tr>
<th>Criticality</th>
<th>T</th>
<th>(U_{\text{worst}})</th>
<th>(U_{\text{expected}})</th>
<th>(U_{\text{average}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>10</td>
<td>50%</td>
<td>50%</td>
<td>0.05%</td>
</tr>
<tr>
<td>Medium</td>
<td>1</td>
<td>(200%)</td>
<td>10%</td>
<td>2.5%</td>
</tr>
<tr>
<td>Low</td>
<td>100</td>
<td>(1000%)</td>
<td>20%</td>
<td>10%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>over!</td>
<td>80%</td>
<td>12.55%</td>
</tr>
</tbody>
</table>

Execution-Time Servers

- Scheduling model which
  - Allows dealing with jobs with unknown (or untrusted) deadlines
  - Allows integrating sporadic, asynchronous and soft tasks
  - Core concept is a “server” which hands out time to jobs
    - effectively a simple (FIFO) sub-scheduler
  - Popular: Constant bandwidth server (CBS) [Abeni & Buttazzo ’98]

- Idea: server schedules a certain utilisation (“bandwidth”)
  - server has a period, \(T\) and a budget, \(Q = U \times T\)
  - generates appropriate absolute EDF deadlines on the fly
  - when executing a job, budget is consumed
  - when budget goes to zero, new deadline is generated with new budget
Constant Bandwidth Server

- Idea: server schedules a certain utilisation ("bandwidth")
  - server has a period, $T$ and a budget, $Q = U \times T$
  - generates appropriate absolute EDF deadlines on the fly
  - when executing a job, budget is consumed
  - when budget goes to zero, new deadline is generated with new budget

Message-Based Synchronisation

- Tasks may communicate via messages
  - blocking IPC
- Enforces precedence relations
- Tag deadlines onto messages

Shared Resources

Concurrent access to shared resources

t\_low() {
    ....
    \textbf{wait} (sem);
    /* critical section */
    signal (sem);
    ...
}

High-priority job is blocked, waiting for low-priority job

Priority Inversion

- High-priority job is blocked for a long time by a low-prio job
- Long wait chain: $t_1 \rightarrow t_3 \rightarrow t_2$
- Worst-case blocking time of $t_1$ bounded only by $C_2 + C_3 + C_4$
- Must find a way to do better!
Priority Inheritance

- If \( t_1 \) blocks on a resource held by \( t_2 \), and \( P_1 > P_2 \), then
  - \( t_2 \) is temporarily given priority \( P_1 \)
  - when \( t_1 \) releases the resource, its priority reverts to \( P_2 \)

Transitive Inheritance

Deadlock!
Priority Inheritance Protocol (PIP)

- If \( t_1 \) blocks on a resource held by \( t_2 \), and \( P_1 > P_2 \), then
  - \( t_2 \) is temporarily given priority \( P_1 \)
  - when \( t_1 \) releases the resource, its priority reverts to \( P_2 \)
- Transitive inheritance
  - potentially long blocking chains
  - potential for deadlock
- Frequently blocks much longer than necessary

Priority Ceiling Protocol (PCP)

- Purpose: ensure job can block at most once on a resource
  - avoid transitivity, potential for deadlocks
- Idea: associate a ceiling priority with each resource
  - equal to the highest priority of jobs that may use the resource
  - when job accesses its resource, immediately bump prio to ceiling!
- Also called:
  - immediate ceiling priority protocol (ICPP)
  - ceiling priority protocol (CPP)
  - stack-based priority-ceiling protocol
    - because it allows running all jobs on the same stack
- Improved version of the original ceiling priority protocol (OCPP)
  - … which is also called the basic priority ceiling protocol

Priority Ceiling Protocol

- Purpose: ensure job can block at most once on a resource
  - avoid transitivity, potential for deadlocks
- Idea: associate a ceiling priority with each resource
  - equal to the highest priority of jobs that may use the resource
  - when job accesses its resource, immediately bump prio to ceiling!
- Properties:
  - Blocking time is limited to the duration of one critical section
  - Deadlock-free
  - Fewer context switches than OCPP
- Implementation:
  - Each task must declare all resources at admission time
    - System must maintain list of tasks associated with resource
    - Priority ceiling derived from this list
    - For EDF the “ceiling” is the floor of relative deadlines