Predictable response times in event-driven real-time systems

Artist2 Summer School in China 2008
Shanghai, July 2008

Michael González Harbour
mgh@unican.es
www.ctr.unican.es

1. Introduction
2. Basic scheduling
3. Mutual exclusion
4. Distributed systems
5. Hierarchical scheduling
6. Protection and flexible scheduling frameworks
7. Operating systems
8. Programming languages
9. Modeling and integration into the design process
10. Conclusions
Real-time systems

A Real-time system is a combination of a computer, hardware I/O devices, and special-purpose software, in which:

- there is a strong interaction with the environment
- the environment changes with time
- the system simultaneously controls and/or reacts to different aspects of the environment

As a result:

- timing requirements are imposed on software
- software is naturally concurrent

To ensure that timing requirements are met, the system’s timing behavior must be *predictable*
Real-Time Systems

“In Real-time applications, the correctness of computation depends upon not only the results of computation, but also the time at which outputs are generated.”

What’s important in real-time

Predictability of the response time

Criteria for real-time systems differ from that for time-sharing systems.

<table>
<thead>
<tr>
<th></th>
<th>Time-Share Systems</th>
<th>Real-Time Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>High throughput</td>
<td>Ability to meet timing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>requirements: Schedulability</td>
</tr>
<tr>
<td>Responsiveness</td>
<td>Fast average response</td>
<td>Ensured worst-case latency</td>
</tr>
<tr>
<td>Overload</td>
<td>Fairness</td>
<td>Stability of critical part</td>
</tr>
</tbody>
</table>

Worst case cannot be checked by testing
Options for managing time

Compile-time schedules:
- time triggered or cyclic executives
- predictability through static schedule
- logical integrity often compromised by timing structure
- difficult to handle aperiodic events & dynamic changes
- difficult to maintain

Run-time schedules:
- priority-based schedulers
- preemptive or non preemptive
- analytical methods needed for predictability
- separates logical structure from timing
- more flexibility

Example: Control in an Automobile

Activities to control:

- Speed measurement: C=4 ms, T=20 ms, D=5 ms
- ABS Control: C=10 ms, T=40 ms, D=40 ms
- Fuel Injection: C=40 ms, T=80 ms, D=80 ms

C = Execution time
T = Period
D = Deadline
Example: cyclic solution

A cyclic solution implies breaking the longest activity in several parts:

Example: concurrent solution

The concurrent solution is easier to design and maintain:

- **Speed measurement**
  ```
  loop
  Measure_Speed;
  next:=next+0.02;
  Sleep_until next;
  end loop;
  ```

- **ABS control**
  ```
  loop
  ABS_Control;
  next:=next+0.04;
  Sleep_until next;
  end loop;
  ```

- **Fuel injection**
  ```
  loop
  Fuel_Injection;
  next:=next+0.08;
  Sleep_until next;
  end loop;
  ```

- **Activities with no timing requirements**
  ```
  loop
  do work;
  ...
  end loop;
  ```
Example: maintenance

Suppose that we need to execute an aperiodic activity with a 1ms execution time. The minimum interarrival time is 80ms., but the deadline is 20 ms:

1. Cyclic solution:
   - Sample at least every 19 ms to check if an aperiodic event has arrived
   - This implies breaking ABS and FUEL in several parts each

2. Concurrent solution:
   - Add a new high-priority process and repeat the schedulability analysis
   - No modifications to existing code

Fixed-priority scheduling (FPS)

Fixed-priority preemptive scheduling is very popular for practical applications, because:

• Timing behavior is simpler to understand
• Behavior under transient overload is easy to manage
• A complete analytical technique exists
• High utilization levels may be achieved (typically 70% to 95% of CPU)
• Supported in standard concurrent languages or operating systems:
   - Ada 2005’s RT-annex, Java RTSJ
   - Real-time POSIX
Dynamic priority scheduling

In dynamic priority scheduling it is possible make better use of the available resources

We can find two kinds of dynamic priority policies

• **static job priority**: a static priority is assigned to each task job
  - e.g., *EDF* (earliest deadline first): each job is assigned a priority equal to its absolute deadline
  - the absolute deadline of a job does not change

• **dynamic job priority**: the priority of a task job may change before the job is finished
  - e.g., *LLF* (Least Laxity First): the priority of a job is inverse to the laxity: time left until the end of the job’s absolute deadline, minus the remaining computation time
  - the laxity changes with time

Dynamic priority scheduling

Static job priority policies are more common, because they are simpler

• EDF and its compatible variants are the most common
• treatment of transient overload is more complex
• not supported by standard operating systems
  - supported in Java RTSJ
  - recently added to Ada 2005

Mixed EDF/FPS schemes are possibly the best approach
Real-time system model

Real-time situation

Transaction

External Event

Activity

Internal Event

Event Handler

Activity

Event Handler

Timing Requirement

Transaction

...
Short History of Real-Time Analysis

Initial paper by Liu and Layland in 1973
- introduced Rate Monotonic and EDF scheduling
- only applicable to a very restrictive case with independent periodic tasks, and deadlines = periods
- optimal priority assignments (when deadlines are at end of period)
- analytic formulas to check schedulability (utilization tests)

Exact schedulability tests (response time analysis)
- developed by Harter (1984) and Joseph and Pandya (1986)

Extended to handle arbitrary deadlines
- Lehoczky (1990)

Extended to handle input jitter
- Tindell (1994)

Extensions to basic theory

Priority inversion and task synchronization
- Immediate priority ceilings
  - Lampson and Redell (1980)
  - Baker (1991)
- shared resources with priority inheritance
  - Sha, Rajkumar, Lehoczky (1990)
- multiprocessor systems

Aperiodic tasks
- Sporadic server, fixed priorities
  - Sprunt, Sha, Lehoczky (1989)
Multiprocessor and distributed systems, OS issues, etc.

Holistic Analysis.
- Tindell (1994) and Tindell and Clark (1994)
- Palencia et al (1997)

Offset Based Analysis
- Tindell (1994)
- Extended to distributed systems by Palencia and González (1998)
- Optimized by Palencia and González (1999), and by Redell (2003)

Priority assignment techniques

- **Rate Monotonic for deadlines equal to periods**
  - Liu and Layland (1973)

- **Deadline monotonic assignment for pre-period deadlines**

- **Deadlines larger than periods**
  - Audsley (1991)

- **Distributed systems**
  - Tindell, Burns, and Wellings (1992)
  - Gutiérrez García and González Harbour (1995)
Dynamic priorities

EDF response time analysis

- Single processor systems
  - Spuri (1996)

- Distributed systems

- Synchronization

- Aperiodic tasks
  - Constant bandwidth server, Abeni and Buttazzo (1998)

- Hierarchical scheduling

Influence in Standards

IEEE POSIX standards

- Real-time extensions (fixed priorities, priority inheritance protocols)
- Advanced real-time extensions (execution-time clocks, sporadic servers)

Ada

- 1983: fixed priorities
- 1995: priority ceilings in protected objects
- 2005: execution time clocks, EDF

Real-Time Java

- fixed priorities, EDF, priority inheritance, ...
Predictable response times in event-driven real-time systems

1. Introduction
2. Basic scheduling
3. Mutual exclusion
4. Distributed systems
5. Hierarchical scheduling
6. Protection and flexible scheduling frameworks
7. Operating systems
8. Programming languages
9. Modeling and integration into the design process
10. Conclusions

Activities in a basic real-time system: example

<table>
<thead>
<tr>
<th>Periodics</th>
<th>Shared Resources</th>
<th>Aperiodics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_1 ): control</td>
<td>( T = 100 \text{ ms} )</td>
<td>Sensor Data</td>
</tr>
<tr>
<td>( C=20 \text{ ms} )</td>
<td>2 ms</td>
<td>Emergency</td>
</tr>
<tr>
<td>Deadline 100 ms</td>
<td>20 ms</td>
<td>Minimum Interarrival Time = 50 msec</td>
</tr>
<tr>
<td>( \tau_2 ): sensing</td>
<td>( T = 150 \text{ ms} )</td>
<td>Commands</td>
</tr>
<tr>
<td>( C=40 \text{ ms} )</td>
<td>10 ms</td>
<td>Deadline = 6 msec after arrival</td>
</tr>
<tr>
<td>Deadline 130 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \tau_3 ): planning</td>
<td>( T = 350 \text{ ms} )</td>
<td>I/O-processing</td>
</tr>
<tr>
<td>( C=100 \text{ ms} )</td>
<td>10 ms</td>
<td>Average Interarrival Time = 40 msec</td>
</tr>
<tr>
<td>Deadline 350 ms</td>
<td></td>
<td>Desired Average Response Time = 20 ms</td>
</tr>
</tbody>
</table>
**Concepts and Definitions - Periodics**

**Periodic task**
- initiated at fixed intervals
- must finish before start of next cycle

**Task’s CPU utilization:** \[ U_i = \frac{C_i}{T_i} \]
- \( C_i \): compute time (execution time) for task \( \tau_i \)
- \( T_i \): period of task \( \tau_i \)
- \( P_i \): priority of task \( \tau_i \)
- \( D_i \): deadline of task \( \tau_i \)
- \( \phi_i \): phase of task \( \tau_i \)
- \( R_i \): response time of task \( \tau_i \)

**CPU utilization for a set of tasks:** \[ U = U_1 + U_2 + \ldots + U_n \]

---

**Basic principles of real-time analysis**

Two concepts help building the worst-case condition under fixed priorities:

- **Critical instant.** The worst-case response time for all tasks in the task set is obtained when all tasks are activated at the same time.

- **Checking only the deadlines in the worst-case busy period.**
  - for task: interval during which the processor is busy executing \( \tau_i \) or higher priority tasks.
Example of a critical instant

\[ \tau_1 \]
\[ 0 \quad 20 \quad 100 \quad 120 \quad 200 \quad 220 \quad 300 \quad 320 \]
\[ \cdots \]
\[ \tau_2 \]
\[ 0 \quad 20 \quad 60 \quad 150 \quad 190 \quad 300 \]
\[ \tau_3 \]
\[ 0 \quad 60 \quad 100 \quad 120 \quad 150 \quad 190 \quad 200 \quad 220 \quad 240 \quad 350 \]

Priority assignment

If \( D_i \leq T_i \), **deadline monotonic** assignment

- For a set of periodic independent tasks, with deadlines within the period, the optimum priority assignment is the deadline monotonic assignment:
  - Priorities are assigned according to task deadlines
  - A task with a shorter deadline is assigned a higher priority

If \( D_i > T_i \) for one or more tasks, Audsley’s algorithm

- iteratively apply analysis, successively ordering tasks by priority: \( O(n^2) \) times the analysis
**Utilization bound test (D=T) (Liu and Layland, 1973)**

**Utilization Bound Test:** A set of \( n \) independent periodic tasks, with deadlines at the end of the periods, scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasings, if

\[
\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} \leq U(n) = n \left( 2^{1/n} - 1 \right)
\]

<table>
<thead>
<tr>
<th>( n )</th>
<th>( U(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>0.828</td>
</tr>
<tr>
<td>3</td>
<td>0.779</td>
</tr>
<tr>
<td>4</td>
<td>0.756</td>
</tr>
<tr>
<td>5</td>
<td>0.743</td>
</tr>
<tr>
<td>6</td>
<td>0.734</td>
</tr>
<tr>
<td>7</td>
<td>0.728</td>
</tr>
<tr>
<td>8</td>
<td>0.724</td>
</tr>
<tr>
<td>( \infty )</td>
<td>0.693</td>
</tr>
</tbody>
</table>

For harmonic task sets, the utilization bound is \( U(n)=1.00 \) for all \( n \).

**Response time analysis (Harter, 1984; Joseph and Pandya, 1986)**

**Iterative test (pseudopolynomial time):**

\[
a_0 = C_1 + C_2 + \ldots + C_i
\]

\[
a_{k+1} = W_i(a_k) = \left\lfloor \frac{a_k}{T_1} \right\rfloor C_1 + \ldots + \left\lfloor \frac{a_k}{T_{i-1}} \right\rfloor C_{i-1} + C_i + B_i
\]

Finish when two consecutive results are the same
Elements that influence the response time

- Preemption
- Execution
- Blocking

Enhancements to response-time analysis (FPS)

Analysis with arbitrary deadlines (Lehoczky, 1990)
  - checking first deadline is not enough
  - repeat analysis for multiple jobs in a busy period

Analysis with release jitter (Tindell, 1992)

Analysis with overhead effects (context switch)

Methods for controlling input/output jitter
Analysis with arbitrary deadlines and jitter (Tindell, 1992):

We call the maximum release jitter of task \( i \), \( J_i \)

Worst case *response time* of a task: found in a busy period in which all tasks:
- are released at the beginning of the busy period,
- have experienced their maximum jitter on the first job
- and experience the minimum jitter on the following jobs that make them happen inside the busy period

Analysis with arbitrary deadlines and jitter (cont’d):

Iterative equation used for analysis of task-\( i \) in one resource:

\[
\begin{align*}
  w_i^{n+1}(p) &= \sum_{j \in h(p(i))} \left[ \frac{J_j + w_i^n(p)}{T_j} \right] C_j + pC_i + B_i \\
  \text{This is carried out for } p=1,2,3,\ldots, \text{ until } w_i(p) \leq pT_i
\end{align*}
\]
Modeling Task Switching

Two scheduling actions per preemptive task (preemption, and processor relinquishing)

- \( C_s = C_{s1} + C_{s2} \)
- \( C_i = C_i + 2C_s \)

Legend
- Task Execution
- Context Switch Activity

EDF Scheduling Policy

Each task \( i \) has a relative period, \( T_i \), and a relative deadline assigned: \( D_i \)

Each task-\( i \) job \( j \) has an absolute activation time, \( a_{i,j} \), and an absolute deadline, \( d_{i,j} \), used as the inverse of the job priority
Optimality of EDF

In a system with only periodic independent tasks, with a preemptive scheduler, executing in a single processor

- The EDF scheduling policy is optimal (Liu & Layland, 1973)
- 100% utilization may be achieved
- EDF utilization bound test is exact:

\[ U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1 \]

Response time analysis for EDF

**Busy period**: interval during which processor is not idle

Worst case **response time** of a task: found in a busy period in which all other tasks:

- are released at the beginning of the busy period,
- and have experienced their maximum jitter

Differences with fixed priorities:

- The task under analysis does not necessarily start with the busy period
- The busy period is longer
Response Time Analysis for EDF

Worst contribution of task $\tau_i$ to the busy period at time $t$, when the deadline of the analyzed task, $\tau_a$, is $D$:

$$W_i(t, D) = \min\left(\left[\frac{t + J_i}{T_i}\right], \left[\frac{J_i + D - d_i}{T_i}\right] + 1\right)_0 \cdot C_i$$

Worst completion time of activation $p$, if first activation at $A$:

$$w_a^A(p) = pC_a + \sum_{i \neq a} W_i(w_a^A(p), D^A(p))$$

Worst response time if first activation is $A$:

$$R^A(p) = w_a^A(p) - A + J_a - (p - 1)T_a$$

Response time analysis in a single resource (cont’d)

Set of potential critical instants; $L$ is the longest busy period:

$$\Psi = \bigcup\{(p - 1)T_i - J_i + d_i\} \quad \forall p = 1 \ldots \left[\frac{L - J_a}{T_a}\right], \forall i \neq a$$

Values of $A$ to check:

$$\Psi^* = \{\psi_x \in \Psi | (p - 1)T_a - J_a + d_a \leq \psi_x < pT_a - J_a + d_a\}$$

$$A = \psi_x - [(p - 1)T_a - J_a + d_a]$$

Worst-case response time

$$R_a = \max[R^A(p)] \quad \forall p = 1 \ldots \left[\frac{L - J_a}{T_a}\right], \forall A \in \Psi^*$$
Handling aperiodic activities

The main problem is guaranteeing predictability in systems with unbounded aperiodic events.
Scheduling aperiodic tasks

Periodic Polling:

Direct Event Processing:

Background:

Sporadic Server:

Legend
- Event Processing
- Replenishment
- Event Arrival

Periodic servers

Sporadic server (FPS)
- Characterized by two parameters:
  - initial execution capacity ($C_R$)
  - replenishment period ($T_R$)
- It guarantees
  - a minimum bandwidth for aperiodic events
  - bounded preemption on lower priority tasks

Servers also exist for EDF scheduling
- Constant bandwidth servers (CBS)
Constant bandwidth server

The CBS is assigned a service time, $Q_s$, and a period, $T_s$. It assigns each job an initial deadline

• If an event arrives at $r_k$, it is assigned a deadline:

$$d_k = \max(r_k, d_{k-1}) + T_s$$

When a task requires more time than what is reserved for it, the absolute deadline is re-evaluated by adding $T_s$ to it.

The CBS can be used to bring predictability even if the execution times are unknown or when they exceed the estimated worst-case value.

Predictable response times in event-driven real-time systems

1. Introduction
2. Basic scheduling
3. Mutual exclusion
4. Distributed systems
5. Hierarchical scheduling
6. Protection and flexible scheduling frameworks
7. Operating systems
8. Programming languages
9. Modeling and integration into the design process
10. Conclusions
Example: synchronization

Unbounded priority inversion

\[ \tau_1: \{...P(S1)...V(S1)\} \]
\[ \tau_3: \{...P(S1)...V(S1)\} \]
Synchronization protocols

Protocols that prevent unbounded priority inversion:

- **No preemption**
  - equivalent to raising the priority of the critical section to the highest level

- **Immediate priority ceiling protocol**
  - raises the priority of the critical section to a value called the "ceiling"
  - lowest blocking time, no extra context switches
  - best for static environments

- **Priority inheritance**
  - raises the priority only if somebody is waiting
  - best for dynamic environments

---

**Immediate Priority Ceiling (IPC)**

\[
\tau_2: \{ \ldots P(S1) \ldots V(S1) \ldots \} \\
\tau_4: \{ \ldots P(S1) \ldots V(S1) \ldots \}
\]

\[\tau_1^{(H)}\]

\[\tau_2\]

\[\tau_3\]

\[\tau_4^{(L)}\]

- **Legend**
  - S1 locked
  - Executing
  - Blocked

- **Critical section executes at highest locker’s priority**
**Basic Inheritance Protocol (BIP)**

\[ \tau_2 = \{ ...P(S1)...V(S1)... \} \]
\[ \tau_4 = \{ ...P(S1)...V(S1)... \} \]

---

**Comparison of Synchronization Protocols**

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Non preemption</th>
<th>IPC</th>
<th>BIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires precalculating ceilings</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Blocked at most once</td>
<td>Yes(^1)</td>
<td>Yes(^2)</td>
<td>No</td>
</tr>
<tr>
<td>Avoids deadlock</td>
<td>Yes(^1)</td>
<td>Yes(^1)</td>
<td>No</td>
</tr>
<tr>
<td>Avoids unnecessary blocking</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Average-case response time</td>
<td>Equal to worst-case</td>
<td>Equal to worst-case</td>
<td>Very low</td>
</tr>
<tr>
<td>Worst-case context switch overhead</td>
<td>0</td>
<td>0</td>
<td>2(C_s) per critical section</td>
</tr>
</tbody>
</table>
Summary of properties

**Dynamic** systems (no precalculated ceilings):
- **BIP** has less blocking than Non-Preemption

**Static** systems (ceilings are precalculated):
- **IPC** has less overhead than PCP
- Critical sections are designed to not suspend themselves
  - this is desirable in any case

Analyzing the Blocking Times

For the no preemption, immediate priority ceiling, or priority ceiling protocols:
- \( B_i \) is the maximum of all the critical sections of lower priority tasks whose ceiling is \( \geq P_i \)

For the basic priority inheritance protocol:
- The critical sections that may cause blocking are those of lower priority tasks whose ceiling is \( \geq P_i \); ceiling may be affected by transitive or recursive inheritance.
- Only one independent (non overlapped) critical section per semaphore may cause blocking (pick the longest)
- Only one independent critical section per lower priority task may cause blocking (pick the longest)
Synchronisation in EDF

Blocking effects similar to unbounded priority inversion exist

Synchronisation protocols

• (dynamic) priority inheritance
• Baker’s protocol
  - applicable to many scheduling policies
  - immediate priority ceiling is a special case
  - similar properties

SRP Protocol

A preemption level, $\pi_i$, is assigned to each task $i$

• in reverse order relative to the relative deadlines, $D_i$

Each resource $R$ has a preemption ceiling, $C_R$. For binary resources (with mutual exclusion)

$$C_R = \max(\pi_i | \tau_i \text{ uses } R)$$

The dynamic system preemption ceiling is defined as

$$\Pi = \max(C_R | R \text{ is locked})$$
SRP Protocol

The scheduler is modified to add a new preemption rule:
- a task is only allowed to execute if its job priority is the highest of all the executable tasks, and $\pi_j > \Pi$

The properties of SRP are the same as for the immediate priority ceiling. Assuming no suspension inside critical sections, and a single-processor system:
- if a task blocks, it does it only before starting the execution of its job
- a task can only be blocked once per job
- there are no deadlocks caused by SRP resources

Elements that influence the response time
Lessons learned from basic real-time theory

In fixed-priority systems, the worst-case response time of a task is a function of:

- **preemption**: time waiting for higher-priority tasks
- **execution**: time to do its own work
- **blocking**: time delayed by lower-priority tasks, usually because of mutual exclusion synchronization

Deadlines are missed when utilization is over 100% (seems obvious, but happens in reality):

- **Solutions**:
  - change the computation requirements
  - change the periods or interarrival times
  - buy a faster CPU

Lessons learned from basic real-time theory (cont’d)

Deadlines are missed because there is too much preemption:

- **A task or a task’s portion runs at a higher priority than it should**
  - this happens typically with interrupt service routines

- **There are not enough priorities**

- **Solutions**:
  - preemption can be minimized by choosing an optimum priority assignment
  - sometimes a task or an ISR must be split into parts whose priorities can be independently assigned
  - buy a system with user-defined priorities for I/O drivers
  - buy a system with at least 32 priority levels
Lessons learned from basic real-time theory (cont'd)

Deadlines are missed because there is too much blocking:

- Normal semaphores suffer from unbounded priority inversion
  - this causes horrendous worst-case blocking times

- Solutions
  - disable preemption in the critical sections (may still have too much blocking)
  - priority inheritance protocol
  - priority ceiling or Baker’s protocol
  - split long critical sections

Predictable response times in event-driven real-time systems

1. Introduction
2. Basic scheduling
3. Mutual exclusion
4. Distributed systems
5. Hierarchical scheduling
6. Protection and flexible scheduling frameworks
7. Operating systems
8. Programming languages
9. Modeling and integration into the design process
10. Conclusions
Real-time networks

Very few networks guarantee real-time response
• many protocols allow collisions
• no priorities or deadlines in the protocols

Some solutions
• CAN bus and other field busses
• Priority-based token passing
• Time-division multiplexed access
• Point to point networks

Distributed system model

Linear Action: \( e_{j-1,j} \rightarrow a_j \rightarrow e_{j,j+1} \quad T_{j-1,j} = T_j = T_{j,j+1} \)

Linear Response to an Event:

CPU-1   Network   CPU-2
![Diagram showing linear response to an event]
Release jitter in distributed systems

In the example below, actions $a_2, a_3$ and $a_5, a_6, a_7, a_8$ have jitter, even if $a_1$ and $a_4$ are purely periodic:

The problem

Release jitter in one resource depends on the response times in other resources

Response times depend on release jitter
Holistic analysis technique

Developed at the University of York

Each resource is analyzed separately (CPU’s and communication networks):

• the analysis is repeated iteratively, until a stable solution is achieved
• the method converges because of the monotonic relation between jitters and response times

Analysis provides guarantees on schedulability, but is very pessimistic
• independent critical instants may not be possible in practice

Pessimism in holistic analysis

Holistic analysis technique assumes independent task activations in each resource
Independent task analysis

Execution timeline for task 22 in previous example:

- task-1
- task 21
- task 22

Response time for task 2:
- includes times for task 21, m1, task 4, m2 and task 22
- total is 270

Using offsets to reduce pessimism

Execution timeline for analysis of task 2:

- task-1
- task 2
- serial line
- task 3
- task 4

Arrows indicate offsets with t = 145
Offset-based techniques

They reduce the pessimism of the worst-case analysis in multiprocessor and distributed systems:
- by considering offsets in the analysis

Exact analysis is intractable:
- The task that generates the worst-case contribution of a given transaction is unknown
- The analysis has to check all possible combinations of tasks

Upper bound approximation
- This technique is pessimistic, but polynomial
- In 93% of the tested cases, the response times were exact

Example

RMT: Teleoperated Robot

Teleoperation Station

Ethernet Network

Local Controller

GUI

Trajectory Planner

Reporter

Command Message

Status Message

Command Manager

Data Sender

Servo Control

1sec

50ms

1sec

5ms
Priority assignment techniques

No known optimum priority assignment for distributed systems

*Simulated Annealing*
- Standard optimization technique

*HOPA*
- Heuristic algorithm based on successively applying the analysis
- Much faster than simulated annealing
- Usually finds better solutions

None of them guarantees finding the solution

---

Summary

<table>
<thead>
<tr>
<th>Kind of Analysis</th>
<th>Deadlines</th>
<th>Number of processors</th>
<th>Fixed Priorities</th>
<th>EDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utilization test</td>
<td>D=T</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Response Time Analysis</td>
<td>Arbitrary</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Holistic Analysis</td>
<td>Arbitrary</td>
<td>Many</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Offset-Based Analysis</td>
<td>Arbitrary</td>
<td>Many</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Distributed model also applicable to:
- signal & wait synchronization
- activities that suspend themselves (i.e., delays, I/O, ...)
Predictable response times in event-driven real-time systems

1. Introduction
2. Basic scheduling
3. Mutual exclusion
4. Distributed systems
5. Hierarchical scheduling
6. Protection and flexible scheduling frameworks
7. Operating systems
8. Programming languages
9. Modeling and integration into the design process
10. Conclusions

Mixing EDF and fixed priorities

EDF scheduler

Multimedia processing system

Task 1  Task 2  Task 3  Task 4  Task 5
Mixing EDF and fixed priorities

**Hardware scheduler**

Interrupt service routines

ISR 1  ISR 2

**EDF scheduler**

Multimedia processing system

Task 1  Task 2  Task 3  Task 4  Task 5

Mixing EDF and fixed priorities

**Hardware scheduler**

Interrupt service routines

ISR 1  ISR 2

**Fixed priority scheduler**

Control tasks

Task 1  Task 2  Task 3

**EDF scheduler**

Multimedia processing system

Task 1  Task 2  Task 3  Task 4  Task 5
Mixing EDF and fixed priorities

RTA for EDF-within-priorities

Techniques have developed to obtain a worst-case response time analysis in systems with hierarchical schedulers

- underlying fixed priority scheduler
- EDF schedulers running tasks at a given priority level

We call this approach “EDF within fixed priorities”
Predictable response times in event-driven real-time systems

1. Introduction
2. Basic scheduling
3. Mutual exclusion
4. Distributed systems
5. Hierarchical scheduling
6. Protection and flexible scheduling frameworks
7. Operating systems
8. Programming languages
9. Modeling and integration into the design process
10. Conclusions

Motivation for time protection

Response-time analysis is heavily based on knowing the worst-case execution times (WCET) of each individual task

- tools based on precise processor models
- tools based on measurement + probabilistic methods

Precise WCET is very different from average case

Time protection helps by detecting execution-time overruns

Independently developed components sharing resources require time protection
Contract-based scheduling

The FRESCOR flexible scheduling framework project

A contract specifies:
- the minimum resource usage requirements
- how to make use of spare resources

FRESCOR Increases the level of abstraction for RT services
- automatic admission test
- independence of scheduling algorithm
- dynamic reclamation of unused resources
- coexistence of hard real-time, quality of service

Temporal encapsulation of subsystems

Closes the gap between theory and practice
Example: Periodic task with budget and deadline control

With OS API

set priority
create budget signal handler
create deadline signal handler
create budget timer
create deadline timer

while (true) {
  reset deadline timer
  set budget timer
  do useful things
  reset budget timer
  set deadline timer
  wait for next period
}

With FRSH API

create contract with \{C,T\}
negotiate the contract

while (true) {
  do useful things
  frsh_timed_wait
}

Example: 3-version algorithm

create contract with
  \{\{C1,T\},\{C2,T\},\{C3,T\}\}
negotiate the contract

while (1) {
  if (current_budget<C2) {
    do_version_1
  } else if (current_budget<C3) {
    do_version_2
  } else {
    do_version_3
  }
  frsh_timed_wait
}
Predictable response times in event-driven real-time systems

1. Introduction
2. Basic scheduling
3. Mutual exclusion
4. Distributed systems
5. Hierarchical scheduling
6. Protection and flexible scheduling frameworks
7. Operating systems
8. Programming languages
9. Modeling and integration into the design process
10. Conclusions

Real-time operating systems

In the past, many real-time systems did not need an operating system

Today, many applications require operating system services such as:

- concurrent programming, communication networks, file system, etc.

The timing behavior of a program depends strongly on the operating system’s behavior

POSIX definition of real-time in operating systems:
“The ability of the operating system to provide a required level of service in a bounded response time.”
What is POSIX?

Portable Operating System Interface

Based on UNIX operating systems

The goal is portability of
- applications (at the source code level)
- programmers

Sponsored by the IEEE and The Open Group

Real-Time Features in POSIX

<table>
<thead>
<tr>
<th>Real-time Features</th>
<th>Support in POSIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority preemptive scheduler</td>
<td>SCHED_FIFO scheduling policy</td>
</tr>
<tr>
<td>Sufficient number of priorities</td>
<td>At least 32 priority levels</td>
</tr>
<tr>
<td>Priorities modifiable at run-time</td>
<td>Yes</td>
</tr>
<tr>
<td>Known context switch times, etc.</td>
<td>Not provided</td>
</tr>
<tr>
<td>Synchronization primitives free of unbounded priority inversion</td>
<td>PRIO_INHERIT and PRIO_PROTECT for mutexes</td>
</tr>
<tr>
<td>Periodic task activation</td>
<td>Yes</td>
</tr>
<tr>
<td>Execution-time budgets</td>
<td>Execution time clocks and timers</td>
</tr>
<tr>
<td>Sporadic server</td>
<td>SCHED_SPORADIC scheduling policy</td>
</tr>
<tr>
<td>Synchronization primitives free of remote blocking</td>
<td>Immediate priority protocol for mutexes</td>
</tr>
</tbody>
</table>
POSIX real-time profiles (POSIX.13)

The POSIX Standard:
• Allows writing portable real-time applications
• Very large: inappropriate for embedded real-time systems

POSIX.13:
• Defines four real-time system subsets (profiles)
• C and Ada language options

Summary of RT Profiles
System tick and other OS events

Can be measured with a program that constantly reads the clock and records differences large than usual

Measuring context switches

High prio task

Low prio task

sets a flag

read the clock
Delay expirations

System timer ISR

High prio task

Low prio tasks

↑ task activation

Predictable response times in event-driven real-time systems

1. Introduction
2. Basic scheduling
3. Mutual exclusion
4. Distributed systems
5. Hierarchical scheduling
6. Protection and flexible scheduling frameworks
7. Operating systems
8. Programming languages
9. Modeling and integration into the design process
10. Conclusions
Programming languages for real-time systems

Some languages do not provide support for concurrency or real-time

- C, C++
- support achieved through OS services (POSIX)

Other languages provide support in the language itself

- Java (concurrency) and RTSJ (real-time)
  - API based
- Ada: concurrency and real-time
  - integrated into the core language

Language support helps in

- increased reliability
- more expressive power
- reduced development and maintenance costs
Ada Real-Time Features

<table>
<thead>
<tr>
<th>Real-time Features</th>
<th>Support in Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority preemptive scheduler</td>
<td>Default policy in the RT Annex</td>
</tr>
<tr>
<td>Sufficient number of priorities</td>
<td>At least 31 priority levels defined in the Real-Time Annex</td>
</tr>
<tr>
<td>Priorities modifiable at run-time</td>
<td>Yes, in the Real-Time Annex</td>
</tr>
<tr>
<td>Known context switch times, etc.</td>
<td>Not provided</td>
</tr>
<tr>
<td>Synchronization primitives free of unbounded priority inversion</td>
<td>Immediate priority ceiling protocol for protected objects</td>
</tr>
<tr>
<td>Periodic task activation</td>
<td>Absolute delay (delay until)</td>
</tr>
<tr>
<td>Execution-time budgets</td>
<td>Ada.Execution_Time</td>
</tr>
<tr>
<td>Sporadic server</td>
<td>Not provided</td>
</tr>
<tr>
<td>Synchronization primitives free of remote blocking</td>
<td>Immediate priority ceiling protocol for protected objects</td>
</tr>
</tbody>
</table>

Predictable response times in event-driven real-time systems

1. Introduction
2. Basic scheduling
3. Mutual exclusion
4. Distributed systems
5. Hierarchical scheduling
6. Protection and flexible scheduling frameworks
7. Operating systems
8. Programming languages
9. **Modeling and integration into the design process**
10. Conclusions
Motivation

The latest schedulability analysis techniques are difficult to apply by hand

Need to integrate all the techniques in a single tool suite
  • schedulability analysis tools
  • priority assignment
  • sensitivity analysis

A model of the real-time behavior is necessary

MAST environment
The MAST_RT_View in the “4+1 View”

- **Logical View**: (Functionality)
- **Component View**: (Software management)
- **Use Case View**: (Understandability, Usability)
- **Process View**: (Performance, Scalability, Throughput)
- **Deployment View**: (System Distribution)

**Integration into the design process**

- **Translation**
  - Detailed Design
  - Object analysis
  - System Engineering
- **Design**
  - Architectural Design
  - Map real-time properties to subsystems
- **Analysis**
  - Sensitivity analysis
- **Testing**
  - Unit Testing
  - Integration and Test
  - Validation
- **Party**
  - Requirements Analysis
- **Identification of real-time situations**
  - Transactions
  - Timing requirements
  - Work loads

- **WCET evaluation**
- **Generation of detailed real-time models**
- **Schedulability analysis**
- **Priority Assignment**
- **Sensitivity analysis**
- **Concurrency patterns**
- **Synchronization patterns**
- **Architectural real-time models**
- **Scheduling policies**
Predictable response times in event-driven real-time systems

1. Introduction
2. Basic scheduling
3. Mutual exclusion
4. Distributed systems
5. Hierarchical scheduling
6. Protection and flexible scheduling frameworks
7. Operating systems
8. Programming languages
9. Modeling and integration into the design process

10. Conclusions

Conclusions

Priority-based run-time scheduling supports

• predictability of response time, via static analysis
• flexibility and separation of concerns
• dynamic and fixed priorities can be combined together

Real-time theory is now capable of analyzing complex real-time systems

Tools are available to

• automatically analyze a system
• provide sensitivity analysis
• design space exploration
Conclusions (cont’d)

Flexible scheduling frameworks
• integrate hard real-time with quality of service requirements
• increased use of system resources
• higher level of abstraction

Future goals

Integration of schedulability analysis with design tools and methods
• component-based approaches

Improved methods for WCET estimation

New networks and protocols that use real-time scheduling

Integration of real-time response into general-purpose operating systems
• kernel support, including I/O drivers
• flexible scheduling frameworks
**URLs**

**MAST**
- http://mast.unican.es

**FRESCOR**
- http://frescor.org

**MaRTE OS**
- http://marte.unican.es

---

**References**

**Books**


**Real-Time analysis in single processor systems**

References (continued)


References (continued)


References (continued)


References (continued)


Aperiodic Scheduling


References (continued)

Distributed Real-Time Systems


References (continued)


Synchronization


References (continued)

Priority assignment


References (continued)

Initial work on RMA


References (continued)

Real-Time operating systems and networks


References (continued)

Ada


 POSIX: Standard Operating System Interface


References (continued)


References (concluded)

MAST


FRESCOR project

Part of the material in this tutorial is extracted from the tutorial on Rate Monotonic Analysis by the Software Engineering Institute of Carnegie Mellon University.