Real-Time Software

The Not So Simple Process Model

René Rydhof Hansen

9. november 2010

Today's Goals

- To be able to use RTA to determine schedulability
  - Both for the simple process model
  - As well as for relevant extensions

Simple Process Model

- The application has a fixed set of processes
- All processes are periodic, with known periods
- All processes have a deadline
- The processes are independent of each other
- All processes have a worst-case execution time
- All context-switching costs etc. are ignored

Sporadic Processes

- Use minimum inter-arrival time (MIT) as period
- Require $D < T$
- Response time analysis applies
- Deadline monotonic priority assignment is optimal
  - Equivalent to rate monotonic priority assignment for $D = T$

Aperiodic Processes

- No minimum inter-arrival time
- Can run at a lowest priority
- Alternative: use a server
  - Period T
  - Capacity C
  - Often with highest priority

Simple Process Model

- The application has a fixed set of processes
- Sporadic and periodic processes have known T
- All processes have a deadline
- The processes are independent of each other
- All processes have a worst-case execution time
- All context-switching costs etc. are ignored

Simple Process Model

- The application has a fixed set of processes
- Sporadic and periodic processes have known T
- All processes have a deadline
- The processes are independent of each other
- All processes have a worst-case execution time
- All context-switching costs etc. are ignored

Process Interactions: Blocking

- A process waiting for a lower-priority process suffers priority inversion
- The process is blocked

Example

Periodic processes: a, b, c, and d

Resources: Q and V

Process	Priority	Execution Sequence	Release Time
a	1	EQQQQE	0
b	2	EE	2
c	3	EVVE	2
d	d	EEQVE	4

Remember: 1 is lowest priority

Priority Inheritance

Task P suspended waiting for Q: priority of Q is raised to mathc priority of P

Calculating Blocking

Worst Case Blocking Time

$$B_i = \sum_{k=1}^{K} usage(k, i) C(k)$$

where K is the number of critical regions and

$$usage(k, i) = \begin{cases} 1 & \text{if } \exists P_j \text{ s.t. } P_j \text{ uses } k \text{ and } pri(P_j) < pri(P_i) \\ 0 & \text{otherwise} \end{cases}$$

Worst Case Response Time

$$R_i = C_i + B_i + I_i$$

Note

Assumes simple priority inheritance protocol

Priority Ceiling Protocols

Two forms

- Original ceiling priority protocols (OCPP)
- Immediate ceiling priority protocol (ICPP)

Original Ceiling Priority Protocol (OCPP)

- Each process has a static default priority
- Each resource has a static ceiling: maximum priority of the processes that use it
- A process' dynamic priority is maximum of its own static priority and inherited priorities (from blocked higher-priority processes)
- A process can only lock a resource if its dynamic priority is higher then the ceiling of any currently locked resource (excluding any that it has already locked itself)

Calculating Worst Case Blocking Time under OCPP

$$B_i = \max_{k=1}^{K} usage(k, i) C(k)$$

Immediate Ceiling Priority Protocol (ICPP)

- Each process has a static default priority
- Each resource has a static ceiling: the maximum priority of the processes that use it
- A process has a dynamic priority: maximum of its own static priority and the ceiling of any resources it has locked
- A process will only suffer a block at the very beginning of its execution

Calculating Worst Case Blocking Time under ICPP

$$B_i = \max_{k=1}^{K} usage(k, i) C(k)$$

Note: same as for OCPP

Real-Time Java

ICPP is called *priority ceiling emulation*

Properties of priority ceiling protocols

On a single processor

- A high-priority process is blocked at most once duing its execution by lower-priority processes
- Deadlocks are prevented
- Transitive blocking is prevented
- Mutual exclusive access to resources is ensured by the protocol itself

Comparing Priority Ceiling Protocols

OCPP versus ICPP

- Worst-case behaviour is identical (from a scheduling view point)
- ICPP is easier to implement than the original (OCPP) as blocking relationships need not be monitored
- ICPP blocks prior to first execution: fewer context switches
- ICPP requires more priority movements as this happens with all resource usage
- OCPP changes priority only if an actual block has occurred

Extending Response-Time Analysis for FPS

- Release jitter
- Arbitrary deadlines
- Fault tolerance
- Interrupts
- Context switches

Release Jitter

Definition (Release jitter)

Maximum variation in a tasks' release is called release jitter

Response Time Analysis with Release Jitter

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_i}{T_i} \right\rceil C_j$$

- Does not generally occur for periodic tasks
- Mainly for sporadic tasks
- May also occur when restricting granularity of system clock:

$$R_i^{periodic} = R_i + J_i$$

where $R_i^{periodic}$ is the response time relative to the "real" release time

Arbitrary Deadlines

Assumption

A task is allowed to complete before it is released again

Overlapping releases are analysed in separate windows:

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j$$

where "stable q" can be found by iteration (fix-point computation)

$$R_i(q) = w_i^n(q) - qT_i$$

Number of iterations bounded by $\min\{q | R_i(q) \leq T_i\}$

Worst Case Response Time with Arbitrary Deadlines

$$R_i = \max_{q=0,1,2,\ldots} R_i(q)$$

Fault Tolerance

Definition (Fault Model)

In a RTS deadlines should be met even when a certain level of faults occur, this level is called the fault model.

Response Time Analysis with Simple Fault Model

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} F C_k^f$$

where C_k^f is the extra computation time resulting from an error in process i and F the maximum number of faults tolerated.

Can be changed to model minimum inter-arrival time between faults

Assumption

Error recovery runs at same priority as faulty process

Interrupts (drivers)

Allowing drivers to interrupt

Idea: treat drivers as sporadic processes and add their worst-case response time to the worst-case response time of all processes.

Worst Case Response Time with Interrupts

$$R_i = C_i + I_i + B_i + \underbrace{\sum_{d \in Drivers} \left\lceil \frac{R_i}{T_d} \right\rceil C_d}_{\text{RT for } d}$$

Context Switches

Nothing is free

Must take cost of context switching into account

Worst Case Response Time with Context Switches

CS^1: cost of switching *to* the task; CS^2: cost of switching *from* the task

$$R_i = C_i + I_i + B_i + CS^1$$

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + C_j + CS^2)$$

Summary

- Response time analysis is flexible and caters for:
  - Periodic and sporadic processes
  - Blocking caused by IPC
  - Release jitter
  - Arbitrary deadlines
  - Fault tolerance
  - Interrupts
  - Context switches
  - ...