Evaluating synchronization mechanisms

**Categorizing resource/service requests**

Service requests can be categorized by:

- **Type**
  - (Read requests might be treated very differently from update requests)
- **Time**
  - (often: by their order or relative time only)
- **Attributes, parameters, and the priority of the calling process — including timing constraints**
- **Synchronization state of the resource**
  - (States which refer to the synchronization aspect — including timing constraints)
- **Internal state of the resource**
  - (States which refer to the actual contents and available resources — including timing constraints)

**Handling requests**

- by type and priority
  - (e.g., Ada, Occam2)
- by priority
  - (most rt-systems)
- in order
  - (most systems)
- by their timing constraints
  - (e.g., Real-time Java)
- by client-attributes
  - (mostly: call needs to be accepted first)
- by server state
  - (e.g., Ada, Occam2)

**Accepting or Avoiding?**

(based on Toby Bloom)

- **Conditional wait**
  - (Accept all calls and suspend the threads internally)
  - (e.g., Ada, Occam2)
- **Avoidance synchronization**
  - (Suspend tasks on the level of outer guards)
  - (Most systems)
  - (mostly: call needs to be accepted first)
- **Timing constraints**
  - (States which refer to the actual contents and available resources — including timing constraints)

**Ease of use**

- Expressive power
  - How complex are the constructs?
- How easy can basic methods be combined to complex resource control systems?

**Synchronization state**

- Server state
  - (Serves as the basis for the above)
- **Server state**
  - (Serves as the basis for the above)
- **Window state**
  - (Serves as the basis for the above)
- **System state**
  - (Serves as the basis for the above)

**Synchronization methods**

- by type and priority
  - (e.g., Real-time Java)
- by priority
  - (most rt-systems)
- in order
  - (most systems)
- by client-attributes
  - (mostly: call needs to be accepted first)
- by server state
  - (e.g., Ada, Occam2)

**References for this chapter**

- Tony Bloom
  - Evaluating synchronization mechanisms
  - Proceedings of the seventh ACM symposium on Operating systems principles (1989), pp. 32
- [Mercer97]
  - Clifford W. Mercer
  - Operating-system resource reservation for real-time and multimedia applications
- [Murthy2001]
  - C.R.Murthy, G.Manimaran
  - Resource Management in Real-time Systems and Networks

**Notes**

- All threads are immediately inside the synchronized server.
- Clients might not be able to revoke their requests.
- How easy can basic methods be combined to complex resource control systems?
- How complex are the constructs?
- How can they be combined to complex resource control systems?
Resource Control

Evaluating synchronization mechanisms

Handling requests by parameters

protected body resource_control is
entry allocate(size : instances_of_resource) when resource_free => size is
begin
resource_free := resource_free - size;
distribute;
free(size : instances_of_resource)
end;
free
end resource_control;

procedure allocate(size : instances_of_resource)
begin
if resource_free = resource_free + size
end;
free
end resource_control;

© 2015 Uwe R. Zimmer, The Australian National University page 850 of 961 (chapter 8: "Resource Control" up to page 901)

Resource Control

Handling requests by type and parameters

Lack of expressive power might lead to

Double Interactions

e.g. Register all requests first, then serve the individual types in a global order

e.g. Announce the parameters first, then serve the individual types based on parameters.

Potential implications:
e.g. Requests are no longer atomic!

deadlock

Resource Range

Servers as a trigger to the second call

when client died and is not going to make the second call.

Resource Range Groups

© 2015 Uwe R. Zimmer, The Australian National University page 851 of 961 (chapter 8: "Resource Control" up to page 901)

Resource Control

Handling requests by type and parameters

Lack of expressive power might lead to

Double Interactions

e.g. Register all requests first, then serve the individual types in a global order

e.g. Announce the parameters first, then serve the individual types based on parameters.

Ways-out:
e.g. Define the double interaction by means of atomic actions

and make this known to the underlying synchronization methods

e.g. Annotate or verify that the client will never die during a double interaction sequence

or Eliminate the double interaction by means of a distributed, single request type and requesting.

Resource Range Groups

Client has full control over the types, parameters, and orders.

© 2015 Uwe R. Zimmer, The Australian National University page 853 of 961 (chapter 8: "Resource Control" up to page 901)
Resource Control

Evaluating synchronization mechanisms

Handling requests by type and parameters

With a standard `request` statement:

- Any outstanding timeout is cancelled.
- The call is no longer resolvable.
- Clients losing control stemming from an AEC statement, or a timed entry-call.
- The server can rely on the client call not being revoked.

With a `request with abort` statement:

- All timeouts are reannounced.
- Allows the client to still revoke the call.
- Client side control is maintained.

Resource Reclaiming [Murthy2001]

Motivation for resource reclaiming

1. Worst case assumptions give schedulable systems ...
   but might leave only a few spare resources.
2. Some scheduled resources might not be actually used at run-time.
3. Some aspects of reliability in real-time systems rely directly on the amount of spare resources.

`Resource reclaiming may enhance the system's reliability.`

Resource Reclaiming [Murthy2001]

Expanded task-model

Each task has the following attributes:

- \( T_i \): Cycle time.
- \( Z_i \): Ready time.
- \( D_i \): Deadline.
- \( C_{\text{CI}} \): Worst case computation time.
- \( C_{\text{AI}} \): Actual computation time.
- \( R_i \): Worst case response time.

Further assumptions:

- There are \( n \) processors available.
- Tasks cannot migrate once started.
- At most one task per processor.
- Tasks-queues are in shared memory.
- Tasks are not pre-empted.

Resource reclaiming algorithms

Two extreme versions:

- Dispatching according to the feasible preemtive schedule, i.e., the reclaiming-resource reclaiming costs are zero.
- Greedy scheduling, whenever reclaiming is requested, or at each release of a resource, i.e., optimal reclaiming.

Optimal scheduling of dynamically arriving, non-preemptive tasks in a multiprocessor environment is \( NP \)-hard.

Thus all practical re-scheduling algorithms are approximating and come in two classes:

- Algorithms without passing or bounded complexity.
- Algorithms with passing or in general \( O(n^3) \), but bounded with restricted passing.

Resource reclaiming for independent tasks

Greedy method

Prerun schedule \( S \)

Resource reclaiming for independent tasks

Greedy method

Prerun schedule \( S \) with deadlines

Correctness:

- Resource reclaiming used to maintain the feasibility!
- Inexpensiveness:
  - Resource reclaiming overhead need to be small in comparison to the possibilities.
- Bounded complexity:
  - Resource reclaiming should be included in the task's worst case computation time. Completeness shall be bound by a constant.
- Effectiveness:
  - Improves the system's actual reliability, e.g., more failures can be handled by applying resource reclaiming.

Expanded scheduling-model

- Feasible (prerun) schedule \( S \):
  - Accounting for timing, resource, precedence constraints, and worst case computation times.
- Prerun schedule \( S' \):
  - Starting from \( S \) and taking the actual computation times into account.
- Start and finish times:
  - The scheduled start \( s_i \) and finish times \( f_i \) as from the feasible preemtive schedule \( S \) and the actual start \( s_i \) and finish times \( f_i \) as depicted in the prerun schedule \( S' \) of task \( i \).
- Correct postrun schedule:
  - A postrun schedule is considered correct iff \( \forall i \in V \exists \exists (s'_i \leq s_i < f'_i \leq f_i) \).
- Passing tasks:
  - A task \( i \) passed a task \( j \) iff \( (s'_i < s_j < f'_i < f_j) \), i.e., the strict order in \( S \) is not maintained.

Correctness:

- Resource reclaiming used to maintain the feasibility!
- Inexpensiveness:
  - Resource reclaiming overhead need to be small in comparison to the possibilities.
- Bounded complexity:
  - Resource reclaiming should be included in the task's worst case computation time. Completeness shall be bound by a constant.
- Effectiveness:
  - Improves the system's actual reliability, e.g., more failures can be handled by applying resource reclaiming.

Expanded scheduling-model

- Feasible (prerun) schedule \( S \):
  - Accounting for timing, resource, precedence constraints, and worst case computation times.
- Prerun schedule \( S' \):
  - Starting from \( S \) and taking the actual computation times into account.
- Start and finish times:
  - The scheduled start \( s_i \) and finish times \( f_i \) as from the feasible preemtive schedule \( S \) and the actual start \( s_i \) and finish times \( f_i \) as depicted in the prerun schedule \( S' \) of task \( i \).
- Correct postrun schedule:
  - A postrun schedule is considered correct iff \( \forall i \in V \exists \exists (s'_i \leq s_i < f'_i \leq f_i) \).
- Passing tasks:
  - A task \( i \) passed a task \( j \) iff \( (s'_i < s_j < f'_i < f_j) \), i.e., the strict order in \( S \) is not maintained.
Resource reclaiming for interdependent tasks

Early start: Check concurrent task groups

1. Compute RVs: Check for simultaneous ending
   Postrun schedule $S'$ with early start reclaiming

Restriction vectors: Shen, Ramamritham, Stankovic, '93

Pre-run schedule $S$ with deadlines

Exclusion: $t_2 \notin \{t_1, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}\}

Preorder: $t_1 < t_2 < t_3 < t_4 < t_5 < t_6 < t_7 < t_8 < t_9 < t_{10} < t_{11} < t_{12} < t_{13}$
Resource reclaiming for interdependent tasks

Restriction vectors: Shen, Ramanathan, Stallkamp, ‘93

Proof of correctness

Lemma: Given a feasible prerun schedule $S$ and $x_i > y_i$, then passing must have occurred.

Proof: Assume that passing occurred.

1. All $t_j$ have been dispatched before $t_i$ and
2. If $t_j$ are only dispatched after $t_i$ completed.

By definition of a feasible schedule all $t_j$ do not interfere with $t_i$ and can thus by no means delay the execution of $t_i$.

Therefore $a_i \leq a_j$, $i \neq j$.

Restriction vectors with task migration: Manimaran, Murthy, ‘97

Proof of correctness

Theorem: The RV algorithm gives a correct postrun schedule $S'$.

Proof: By the above lemma, passing occurred if $x_i > y_i$, i.e.,

$|a_j - b_j| > |a_i - b_i| > (a_j > b_j)$

Two cases need to be distinguished:

Case 1: $t_i$ and $t_j$ have resource or precedence conflicts.

If $t_i$ directly or transitively included in the restriction vector $\gamma_j$, then passing must have occurred.

Therefore the case of passing is prevented by the RV algorithm.

Case 2: $t_i$ and $t_j$ have no resource or precedence conflicts, then $t_j$ cannot delay the execution of $t_i$.

Therefore the RV algorithm allows for restricted forms of passing only which does not corrupt the correctness of the postrun schedule $S'$.

Resource reclaiming with task migration: Manimaran, Murthy, ‘97
Resource Reclaiming [Murthy2001]

Resource reclaiming for interdependent tasks
Restriction vectors with task migration: Manimaran, Murthy, '97

Postrun schedule S' with RV reclaiming with task migration

Proof of correctness

Assume that task t_i is next to be scheduled, yet blocked on the dispatching queue D_Q_i on processor P_x and task t_j is immediately executing on processor P_y. To ensure that the swapping of dispatching queues D_Q_i = D_Q_j between processors P_x and P_y does not interfere with correctness of the postrun schedule S', swapping is permitted only if:

\[ s_j = R_t_i \]

i.e., the currently blocked task t_j is not further delayed.

The unrestricted and movable task t_i, which is next to be scheduled on processor P_x is started earlier by swapping it to the scheduling queue D_Q_x.

where zero indicates an independent and one a fully dependent task-set.

Assume that task $t_i$ is next to be scheduled, yet blocked on the dispatching queue $D_Q_i$ on processor $P_x$ and task $t_j$ is currently executing on processor $P_y$. To ensure that the swapping of dispatching queues $D_Q_i = D_Q_j$ between processors $P_x$ and $P_y$ does not interfere with correctness of the postrun schedule $S'$, swapping is permitted only if:

\[ s_j = R_{t_i} \]

i.e., the currently blocked task $t_j$ is not further delayed.

The unrestricted and movable task $t_i$, which is next to be scheduled on processor $P_x$ is started earlier by swapping it to the scheduling queue $D_Q_x$. Where zero indicates an independent and one a fully dependent task-set.

No task is delayed by swapping the dispatching queues $D_Q_i = D_Q_j$. Where zero indicates an independent and one a fully dependent task-set.

Resource reclaiming evaluated

Some additional observations:

- Task graph density $g = [0, 1]$
- $A_{cc}$ actual to worst case ratio
- $M_{ig \_attempts}$ number of checks on dispatch queues by the RV with migration algorithm.
- $IC$ computational cost (Manimaran, Murthy, Vijay Ramamurthi '97):
  
  - $C_{ic \_part} = 1$
  
  \[ C_{ic \_part} = m \cdot C_{ic \_part} \] with $m$ processors
  
  \[ C_{ic \_part} = C_{IC_{part}} + C_{IP_{part}} \] with $C_{IP_{part}}$ the cost for the calculation of the RVs.
  
  \[ C_{IC_{part}} = C_{IC_{part}} \cdot \frac{M_{ig \_attempts}}{C_{IC_{part}}} \]
Resource reclaiming evaluated

**Practical measurements:**
- There is a continuous improvement in terms of gained resources by applying basic - early-start - RV-reclaiming - RV-reclaiming-with-task-migration - algorithms.
- In case of RV-reclaiming with task migration, the extended communications/synchronization overhead can reach noticeable levels.
- There need to be a high degree of dependencies in the task-set \( P \), in order to justify the application of RV-reclaiming with task migration.

Reclaiming in the introduced sense is applicable only to **real-time systems** which:
- Allow for earlier task start times.
- Allow for task migration.
- And where all dependencies can be expressed in terms of the introduced formalism.

Real-time Resource Control [Mercer97]

**Resource Control Issues**

- **Policies:**
  - *Priority assignment problem*: The mapping of the known and arising timing constraints and reliability considerations to linear priorities.
  - *Overload problem*: Predicting and protecting the system from overload conditions.
  - *Flexibility problem*: Locally adjusting the system behaviour to the current timing constraints.

- **Run-time environment:**
  - *Enforcement problem*: Handling tasks and resources which exceed their anticipated worst case limits.
  - *Measurement problem*: Recording all relevant information in a sufficient resolution and frequency.
  - *Coordination problem*: Synchronizing system components which are organized according to different policies.

Summary

- **Resource synchronization primitives**
  - Evaluation criteria for resource synchronization methods.
  - Atomicity, liveliness, and double interaction.

- **Resource reclaiming schemes**
  - Basic reclaiming
  - Early-start algorithm
  - Restriction vector
  - Resource reclaiming with task migration

- **Real-time resource control**
  - Policy and run-time issues to be considered