8

Resource control

Uwe R. Zimmer – The Australian National University

References for this chapter

[Ada95RM] (link to on-line version)
Ada Working Group
ISO/IEC JTC1/SC 22/WG 9
Ada 95 Reference Manual
– Language and Standard Libraries
ISO/IEC 8652:1995(E) with COR.1:2000,
June 2001

[Bloom79]
Toby Bloom
Evaluating synchronization mechanisms
Proceedings of the seventh ACM Symposium on Operating systems principles, 1979

[Burns01]
Alan Burns and Andy Wellings
Real-Time Systems and Programming Languages
Addison Wesley, third edition, 2001

[Mercer97]
Clifford W. Mercer
Operating system resource reservation for real-time and multimedia applications
Ph.D. thesis CMU-CS-97-155, June 1997, Pittsburgh, Pennsylvania 15213-3890

[Murthy2001]
C. Siva Ram Murthy, G. Manimaran
Resource Management in Real-time Systems and Networks
MIT Press, Cambridge, Massachusetts, London, England

all references and links are available on the course page

Topics in real-time resource control

… from synchronization primitives and schedulers to resource management:

- Toby Bloom's evaluation criteria for synchronization primitives
- Resource atomicity, liveliness, and double interaction
- Resource reclaiming (C. Siva Ram Murthy, G. Manimaran)
- Resource reservation schemes (Clifford W. Mercer)

(*not covered here*: general dead-lock prevention / avoidance / detection / recovery algorithms
☞ operating systems course)

Categorizing resource/service requests

(based on Toby Bloom)

Service requests can be categorized by:

- their **type**

  (read requests might be treated very differently from update request)
- their **time** (often: by their **order** or **relative time** only)
- their **attributes**, **parameters**, and the **priority** of the calling process

  (this includes **timing constraints**)
- the **synchronization state** of the resource

  (states which refer to the synchronisation aspect – including **timing constraints**)
- the **internal state** of the resource

  (states which refer to the actual contents and available resources– including **timing constraints**)

Categorizing resource synchronization methods

(based on Toby Bloom)

Two (contradicting?) criteria:

Expressive power

- ☞ are all (required) forms of synchronization available?
- ☞ can all timing requirements be expressed?

Ease of use

- ☞ how error-prone are the constructs?
- ☞ how easy can basic methods be combined to complex resource control systems?

Evaluating synchronization mechanisms

Accepting or Avoiding?

Requests which cannot be fulfilled right now, can be handled via

Conditional wait

- accept all calls and suspend the threads internally

☞ all threads are immediately inside the synchronized server

☞ client threads are released from the server, only when the request is completed (can be overcome)

Avoidance synchronisation

- suspend tasks on the level of guards

☞ all threads are ‘at the borders’ of the synchronized server

☞ threads can easily revoke their requests

Handling resource requests

Required features:

• Handling request types by priorities	✔ (Ada95, Occam2)
• Handling threads by priorities	✔ (most rt-systems)
• Handling threads in order or by their timing constraints	✔ (most systems) ✔ (Real-time Java)
• Handling requests by client-attributes	✍ (mostly: call needs to be accepted first)
• Handling requests by server state	✔ (Ada95, Occam2)

Handling requests by types

```
WHILE TRUE
   PRI ALT
      ALT i=0 FOR max
         update [i] ? object
      ALT j=0 FOR max
         modify [j] ? object
```

```
pragma Queuing_Policy
   (Priority_Queing);

protected Resource_Manager is
   entry Update (…);
   entry Modify (…);
end Resource_Manager;
```

☞ serves clients with higher priority first

☞ serves entries in order of declaration

Handling requests by types

```
WHILE TRUE
   PRI ALT
      ALT i=0 FOR max
         update [i] ? object
      ALT j=0 FOR max
         modify [j] ? object
```

```
pragma Queuing_Policy
   (FIFO_Queing);

protected Resource_Manager is
   entry Update (…);
   entry Modify (…);
end Resource_Manager;

protected body Resource_Manager is

   entry Update (…) when … is …

   entry Modify (…) when …
          and Update'Count = 0 is …

end Resource_Manager;
```

- ☞ serves entries in defined order
- ☞ serves clients in FIFO-order (disregarding priorities)

how to control the order of requests regardless of their types?

how to control permission depending on call-parameters?

Handling requests by parameters

```
protected body resource_control is

   entry allocate(size : instances_of_resource)
      ~~when resources_free >= size is~~
   begin
      resource_free := resource_free - size;
   end allocate;

   procedure free(size : instances_of_resource) is
   begin
      resource_free := resource_free + size;
   end free;

end resource_control;
```

NOT VALID in ADA!

- ‘**SR**’ [Andrews and Olsson 1993] allows for such an direct access
- in most other synchronization environments: *accept all* and then *conditional wait* or *requeue*

Handling requests by parameters (using wrappers)

```
package Resource_Manager is

   Max_Resources : constant Integer := 100;
   type    Resource_Range        is new Integer    range 1..Max_Resources;
   subtype Instances_Of_Resource is Resource_Range range 1..50;

   procedure Allocate (Size : Instances_Of_Resource);
   procedure Free     (Size : Instances_Of_Resource);
end Resource_Manager;

package body Resource_Manager is

   task Manager is
      entry Sign_In  (Size : Instances_Of_Resource);
      entry Allocate (Instances_Of_Resource);
      entry Free     (Size : Instances_Of_Resource);
   end Manager;

   procedure Allocate (Size : Instances_Of_Resource) is begin
      Manager.Sign_In  (Size);
      Manager.Allocate (Size);
   end Allocate;
   procedure Free     (Size : Instances_Of_Resource) is begin
      Manager.Free     (Size);
   end Free;
```

Manager is informed about the request attributes first

entry family

double interaction is hidden

Handling requests by parameters (using wrappers)

```
package Resource_Manager is

   Max_Resources : constant Integer := 100;
   type    Resource_Range        is new Integer    range 1..Max_Resources;
   subtype Instances_Of_Resource is Resource_Range range 1..50;

   procedure Allocate (Size : Instances_Of_Resource);
   procedure Free     (Size : Instances_Of_Resource);
end Resource_Manager;

package body Resource_Manager is

   task Manager is
      entry Sign_In  (Size : Instances_Of_Resource);
      entry Allocate (Instances_Of_Resource);
      entry Free     (Size : Instances_Of_Resource);
   end Manager;

   procedure Allocate (Size : Instances_Of_Resource) is begin
      Manager.Sign_In  (Size);
      Manager.Allocate (Size);
   end Allocate;
   procedure Free     (Size : Instances_Of_Resource) is begin
      Manager.Free     (Size);
   end Free;
```

Manager can apply any policy to accept the '`Allocate`' entries

entry family

double interaction is hidden

Handling requests by types, attributes, and in a global order

Lack of expressive power (e.g. in Ada95) may 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 in parameters

- ☞ Requests are no longer atomic!
- ☞ Server deadlocked,
  when wrongly assuming that the client is going to make the second call
- ☞ Client deadlocked,
  when wrongly assuming that the client died and is not going to make the second call

Handling requests by types, attributes, and in a global order

Lack of expressive power (e.g. in Ada95) may lead to:

☞ Double Interactions

Ways out:

- Define the double interaction by means of atomic actions and make this known to the underlying synchronization methods.
- Assume that the client will never die during a double interaction sequence
- Eliminate the double interaction by means of a attributed, single request type and requeuing

Handling requests by types, attributes, and in a global order

```
type Request_Kinds         is (Allocate_Req, Expand_Req, Free_Req);
type Resource_Range        is …
type Resource_Range_Groups is (small, medium, large);

protected Resource_Control is

   entry Resource_Request (Kind : Request_Kinds; Amount : Resource_Range);

private
   entry Allocate_Sign_In  (Amount : Resource_Range);
   entry Allocate          (Resource_Range_Groups);

   entry Expand_Sign_In    (Amount : Resource_Range);
   entry Expand            (Resource_Range_Groups);

   entry Free              (Amount : Resource_Range);
end Resource_Control;
```

☞ Server has full control over the types, parameters, and orders

Handling requests by types, attributes, and in a global order

```
type Request_Kinds          is (Allocate_Req, Expand_Req, Free_Req);
type Resource_Range         is …
type Resource_Range_Groups  is (small, medium,

protected Resource_Control  is

   entry Resource_Request (Kind : Request_Kinds; Amount : Resource_Range);

private
   entry Allocate_Sign_In   (Amount : Resource_Range);
   entry Allocate           (Resource_Range_Groups);

   entry Expand_Sign_In     (Amount : Resource_Range);
   entry Expand             (Resource_Range_Groups);

   entry Free               (Amount : Resource_Range);
end Resource_Control;
```

The clients are providing all information

☞ Server has full control over the types, parameters, and orders

Handling requests by types, attributes, and in a global order

```
type Request_Kinds          is (Allocate_Req, Expand_Req, Free_Req);
type Resource_Range         is …
type Resource_Range_Groups  is (small, medium,

protected Resource_Control  is

   entry Resource_Request (Kind : Request_Kinds; Amount : Resource_Range);

private
   entry Allocate_Sign_In     (Amount : Resou
   entry Allocate             (Resource_Range

   entry Expand_Sign_In       (Amount : Resou
   entry Expand               (Resource_Range

   entry Free                 (Amount : Resource_Range);
end Resource_Control;
```

The clients are providing all information

The protected object is arranging the suspending queues accordingly (**requeue**-facility)

☞ Server has full control over the types, parameters, and orders

Handling requests by types, attributes, and in a global order

```
type Request_Kinds          is (Allocate_Req, Expand_Req, Free_Req);
type Resource_Range
type Resource_Range_Group                       rge);

protected Resource_Contro

   entry Resource_Request (Kind : Request_Kinds; Amount : Resource_Range);

private
   entry Allocate_Sign_In  (Amount : Resource_Range);
   entry Allocate          (Resource_Range_Groups);

   entry Expand_Sign_In    (Amount : Resource_Range);
   entry Expand            (Resource_Range_Groups);

   entry Free              (Amount : Resource_Range);
end Resource_Control;
```

Is the client going
to loose all control?

☞ Server has full control over the types, parameters, and orders

Handling requests by types and in a global order

requeue with abort

With a standard `requeue` statement:

- any outstanding timeout is cancelled
- the thread is no longer abortable

☞ clients losing control stemming from an ATC statement, or a timed entry-call

☞ the server can rely on the client thread no being revoked.

With a `requeue with abort` statement:

- all timeouts are maintained
- allows the client to still revoke the call

☞ maintains client side control

requeue can also lead to external entries!

☞ aborts need to be considered carefully

Categorizing resource/service requests

(based on Toby Bloom)

Service requests can be categorized by:

- their **type**
- their **time** (often: by their **order** or **relative time** only)
- their **attributes**, **parameters**, and the **priority** of the calling process
- the **synchronization state** of the resource
- the **internal state** of the resource

The real-time perspective:

☞ take special care of failing tasks (atomic actions, deadlocks)

☞ determine and handle timing constraints in resource requests

Resource Reclaiming *[Murthy2001]*

Motivation for resource reclaiming

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

☞ Resource reclaiming **may** enhance the system's reliability

Resource reclaiming properties

- **Correctness**:
    - maintain the feasibility!
- **Inexpensiveness**:
    - resource reclaiming overhead need to be small in comparison to the possible gains
- **Bounded complexity**:
    - resource reclaiming should be included in the task's worst case computation time
      ☞ complexity needs to be bound by a constant
- **Effectiveness**:
    - improve the system's actual reliability,
      thus e.g. more failures can be handled by applying resource reclaiming

Resource Reclaiming [Murthy2001]

Expanded task-model

Each task t_i has the following attributes:

- T_i: **cycle time**
- E_i: **ready time**
- D_i: **deadline**
- C_i: **worst case computation time**
- C_i': **actual computation time**
- R_i: **worst case response time**
- a set of **resource conflicts**: $t_i \otimes t_j$, i.e. t_i or t_j requires a resource exclusively.
- a set of **precedence constraints**: $t_i < t_j$, i.e. t_i completes always before t_j may start.

Further assumptions:

- n processors available
- tasks cannot migrate
- at most one task per processor
- task-queues are in shared memory
- tasks are not pre-empted

More terminology

- **Feasible (prerun) schedule *S***:

  taking into account timing, resource, precedence constraints, and worst case computation times.

- **Postrun schedule *S*′**:

  starting from S and considering the actual computation times into account.

- **Start and finish times**:

  the scheduled start st_i and finish times ft_i as from the feasible prerun schedule S, and the actual start st_i' and finish times ft_i' as depicted in the postrun schedule S' of the task t_i.

- **Correct postrun schedule**:

  a postrun schedule is considered correct iff $\forall t_i \in Q: (st_i' \leq st_i) \wedge (ft_i' \leq d_i)$.

- **Passing tasks**:

  a task t_i passed a task t_j iff $(st_i' < st_j') \wedge (ft_j < st_i)$, i.e. the strict order in S is not maintained.

Resource reclaiming algorithms

Two extreme versions:

- Dispatching according to the feasible prerun schedule S,
  i.e. no reclaiming at all – resource reclaiming cost is zero.
- Global re-scheduling, whenever reclaiming is requested, or at each release of a resource,
  i.e. optimal reclaiming
  – can be applied only, if the reclaiming cost is smaller than the gained resources

Optimal scheduling of
dynamically arriving non-pre-emptive tasks on a multi-processor environment

☞ **NP-hard**

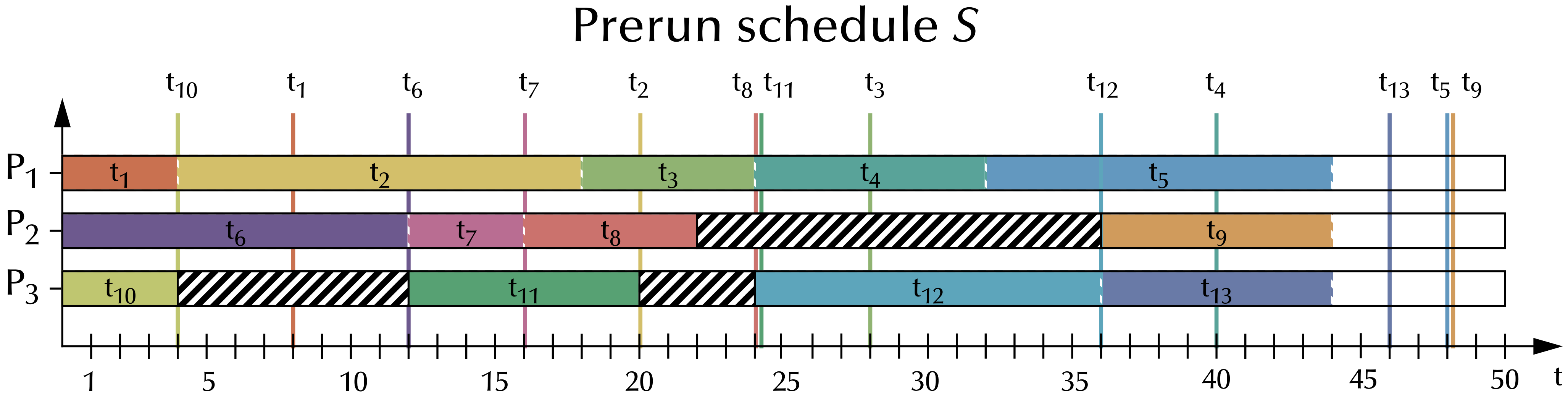
☞ all practical re-scheduling algorithms are approximating. The come in two classes:

- Algorithms without passing ☞ bounded complexity
- Algorithms with passing ☞ in general: $O(log\ n)$, but bounded with restricted passing

Resource Reclaiming [Murthy2001]

Resource reclaiming from independent tasks

☞ trivial: apply a greedy strategy, which dispatches tasks, whenever there are runable tasks.

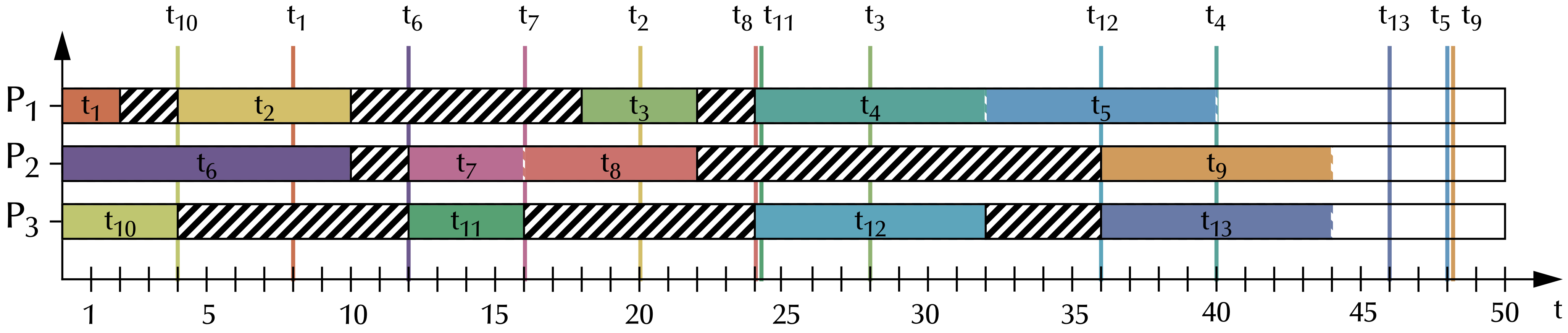


- Feasible prerun schedule S

Resource reclaiming from independent tasks

☞ trivial: apply a greedy strategy, which dispatches tasks, whenever there are runable tasks.

Postrun schedule *S′*



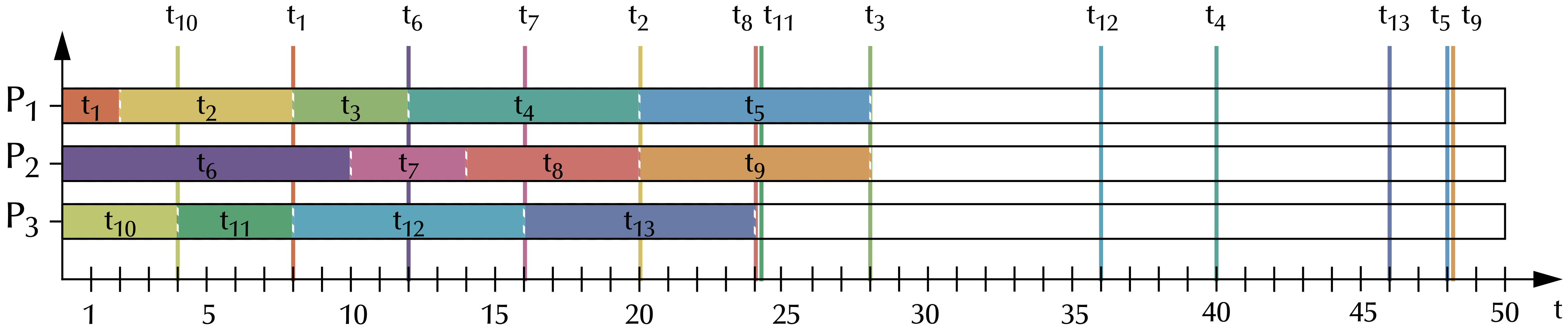
- Postrun schedule *S′* without resource reclaiming

Resource Reclaiming [Murthy2001]

Resource reclaiming from independent tasks

☞ trivial: apply a greedy strategy, which dispatches tasks, whenever there are runable tasks.

Reclaimed resources

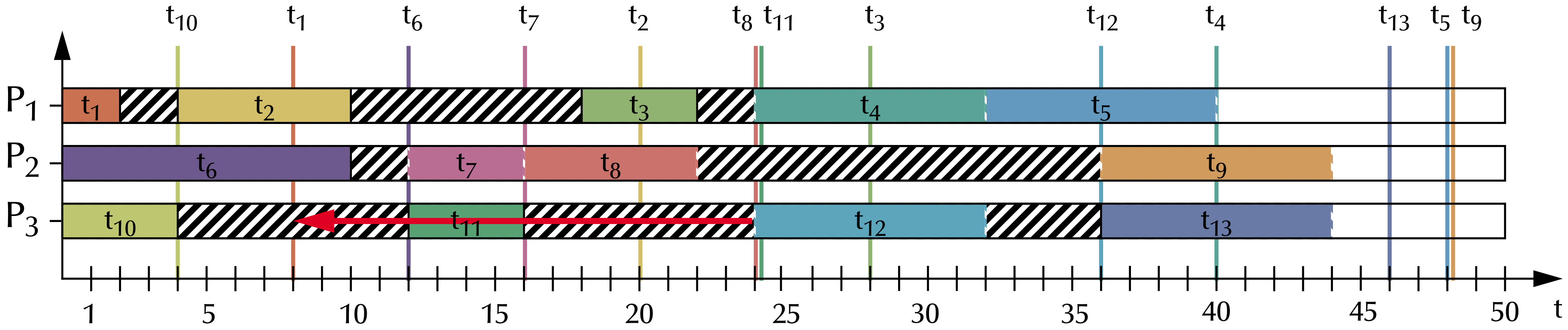


- Postrun schedule S' with resource reclaiming for independent tasks

Resource reclaiming from interdependent tasks

☞ greedy reclaiming

Postrun schedule S'



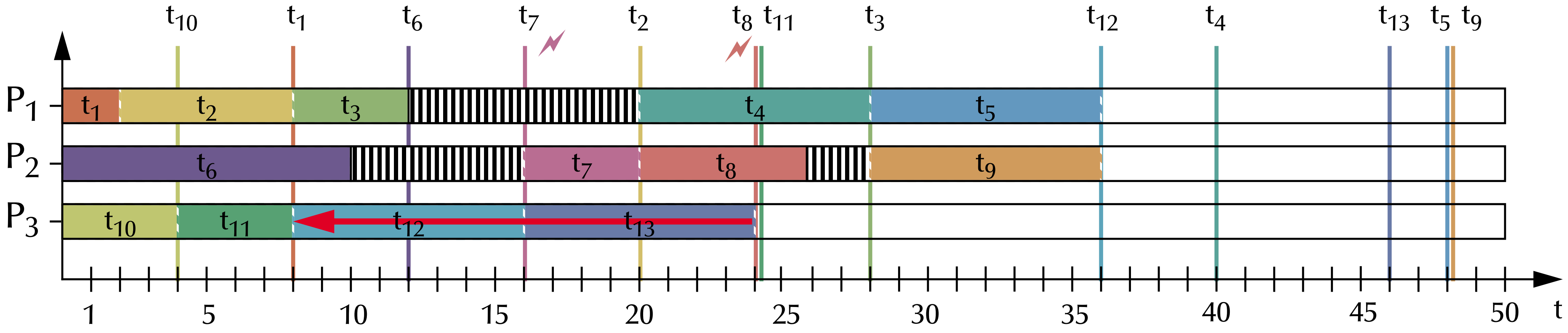
- Postrun schedule S' without resource reclaiming
- Tasks $t_4 \otimes t_7$; $t_4 \otimes t_9$; $t_7 \otimes t_9$; $t_7 \otimes t_{12}$; $t_9 \otimes t_{12}$ have conflicting resource locks

Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

☞ greedy reclaiming ⚡

Runtime anomaly



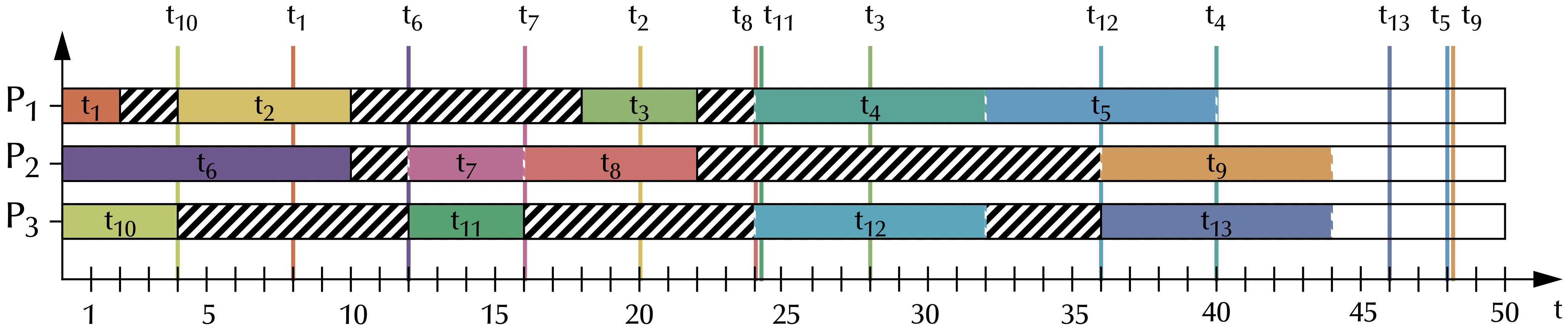
- Postrun schedule S' without greedy resource reclaiming
- Tasks $t_4 \otimes t_7$; $t_4 \otimes t_9$; $t_7 \otimes t_9$; $t_7 \otimes t_{12}$; $t_9 \otimes t_{12}$ have conflicting resource requests ⚡

Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

☞ basic reclaiming: look for simultaneous idling

Postrun schedule S'

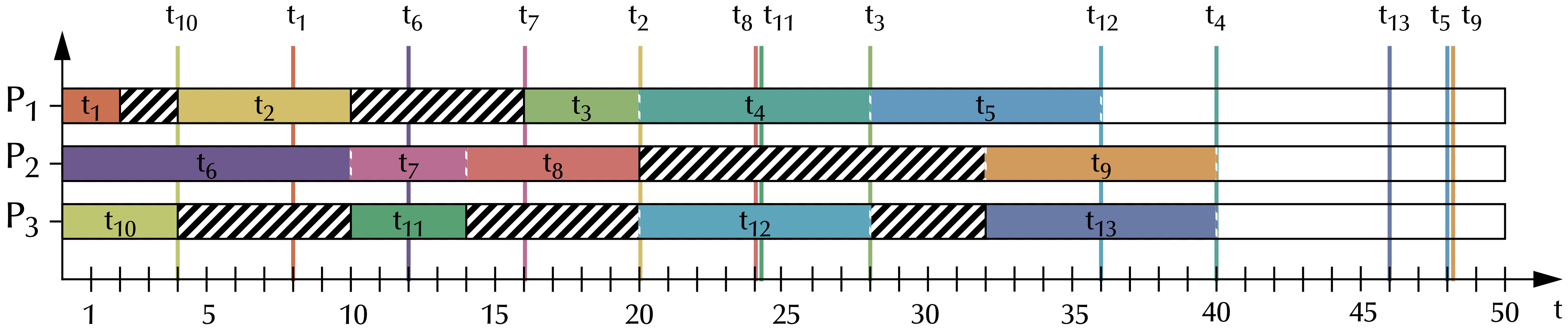


- Postrun schedule S' without resource reclaiming
- Tasks $t_4 \otimes t_7$; $t_4 \otimes t_9$; $t_7 \otimes t_9$; $t_7 \otimes t_{12}$; $t_9 \otimes t_{12}$ have conflicting resource locks

Resource reclaiming from interdependent tasks

☞ basic reclaiming: look for simultaneous idling ✔

Basic reclaiming



- Postrun schedule S' without basic resource reclaiming
- Tasks $t_4 \otimes t_7$; $t_4 \otimes t_9$; $t_7 \otimes t_9$; $t_7 \otimes t_{12}$; $t_9 \otimes t_{12}$ have conflicting resource locks

Resource reclaiming from interdependent tasks

☞ early start algorithm

- Detect overlaps in the prerun schedule S:

$$t_{<i} = \{t_j | ft_j < st_i\}$$

$$t_{>i} = \{t_j | st_j > ft_i\}$$

$t_{\sim i} = \{t_j | ((t_j \notin t_{<i}) \wedge (t_j \notin t_{>i}))\}$ ☞ all tasks which overlap with t_i in S

- Detect tasks overlapping with t_i on processor k and order all sets

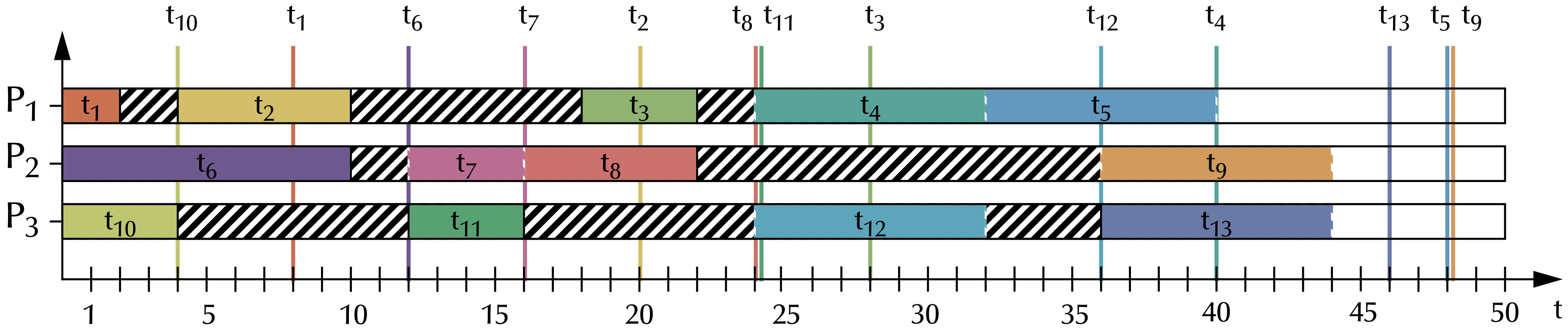
☞ Allow tasks in $t_{\sim i}$ to be executed simultaneously
and ensure that they do not overlap with tasks out of $t_{<i}$ or $t_{>i}$.

☞ Complexity $O(m^2)$; with m processors.

Resource reclaiming from interdependent tasks

☞ early start algorithm

Postrun schedule S'

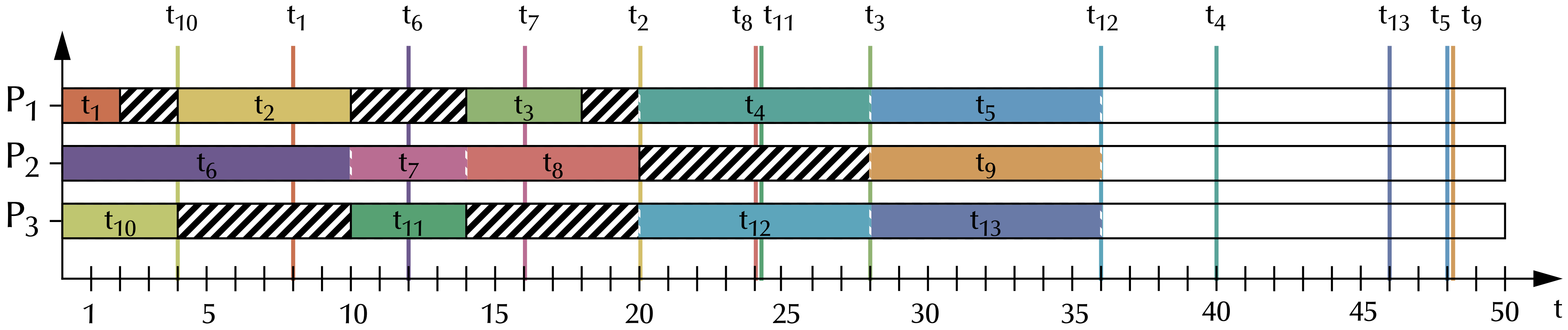


- Postrun schedule S' without resource reclaiming
- Tasks $t_4 \otimes t_7$; $t_4 \otimes t_9$; $t_7 \otimes t_9$; $t_7 \otimes t_{12}$; $t_9 \otimes t_{12}$ have conflicting resource locks

Resource reclaiming from interdependent tasks

☞ early start algorithm ✔

Early start resource reclaiming



- Postrun schedule S' without early start resource reclaiming
- Tasks $t_4 \otimes t_7$; $t_4 \otimes t_9$; $t_7 \otimes t_9$; $t_7 \otimes t_{12}$; $t_9 \otimes t_{12}$ have conflicting resource locks

Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm (Shen, Ramamritham, Stankovic, ‘93)

- **Restriction vector (RV):**

$$RV_i[j] = \begin{cases} t_k \in t_{<i}(j) \,\middle|\, (\neg\exists t_l \in t_{<i}(j) \,|\, st_l > st_k) & \text{if } j = proc(i) \\ t_m \in t_{<i}(j) \,\middle|\, (t_m < t_i \vee t_m \otimes t_i) \wedge (\neg\exists t_l \in t_{<i}(j) \,|\, (st_l > st_m) \wedge (t_l < t_i \vee t_l \otimes t_i)) & \text{if } j \neq proc(i) \\ \text{—} & \text{no such task} \end{cases}$$

- **Completion bit matrix (CBM):**

$$CMB[i, j] = \begin{cases} 1 & \text{iff task } t_i \text{ has completed its scheduled execution in processor } j \\ 0 & \text{otherwise} \end{cases}$$

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm (Shen, Ramamritham, Stankovic, '93)

- compute the $RV_i(j)$ by checking the k most recent tasks in $t_{<i}(j)$
- for any task t_i next to be scheduled on processor j:
    - fetch the most recent CBM
    - if $\forall t_l \in RV_i(j) \big| CBM(i, j) = 1$ then start t_i else idle until the next CBM update.

The algorithm is heuristic in the sense that it is only checking the k most recent tasks in $t_{<i}(j)$

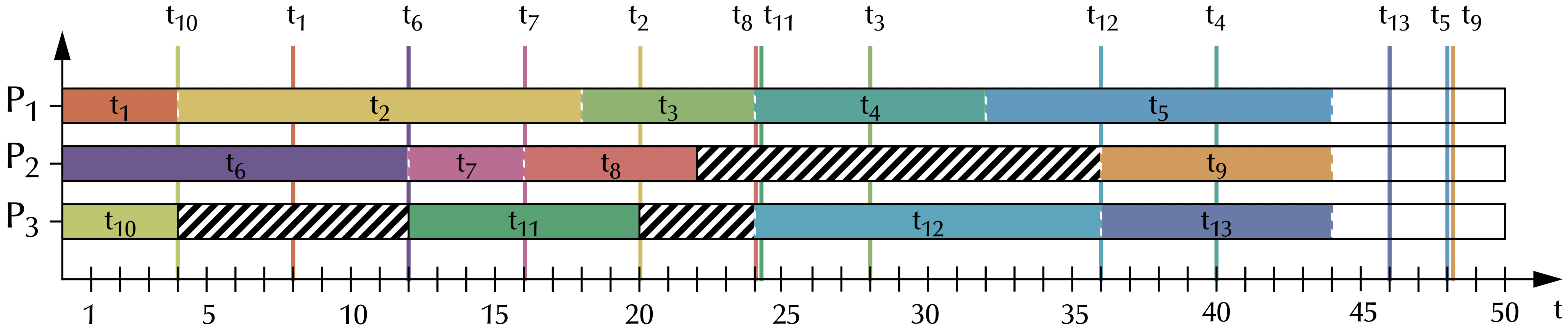
The complexity is $O(m^2)$ since m processors need to check m *RV*-entries ☞ bounded

Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm (Shen, Ramamritham, Stankovic, '93)

Prerun schedule *S*

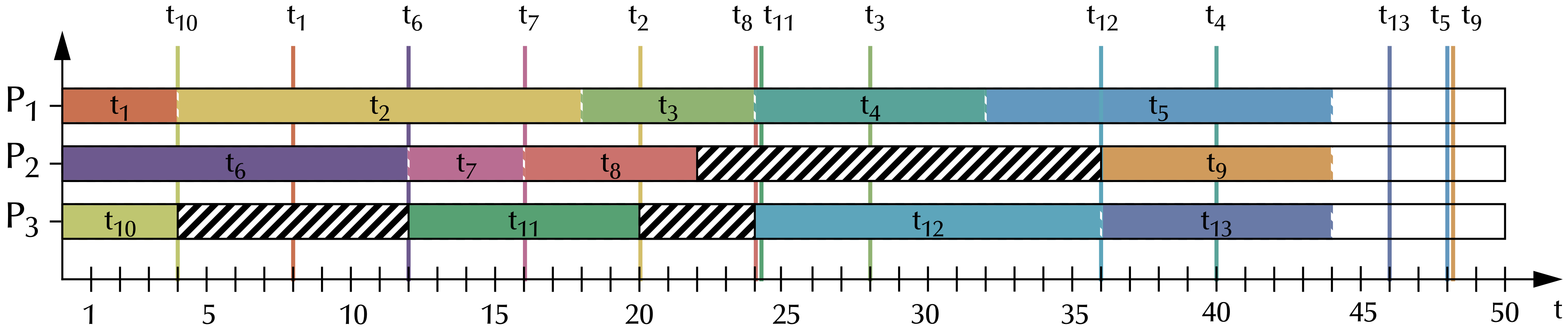


- Tasks $t_4 \otimes t_7$; $t_4 \otimes t_9$; $t_7 \otimes t_9$; $t_7 \otimes t_{12}$; $t_9 \otimes t_{12}$ have conflicting resource locks.
- Tasks $t_{10} < t_8$; $t_{10} < t_4$; $t_8 < t_9$; $t_8 < t_{13}$; $t_1 < t_2$; $t_1 < t_3$; $t_2 < t_{12}$; $t_3 < t_{12}$; $t_{11} < t_{12}$ have precedence relations.

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm (Shen, Ramamritham, Stankovic, ‘93)

Prerun schedule S



☞ *RVs*: t_1: [-, -, -]; t_2: [t_1, -, -]; t_3: [t_2, -, -]; t_4: [t_3, t_7, t_{10}]; t_5: [t_4, -, -];

t_6: [-, -, -]; t_7: [-, t_6, -]; t_8: [-, t_7, t_{10}]; t_9: [t_4, t_8, t_{12}]; t_{10}: [-, -, -];

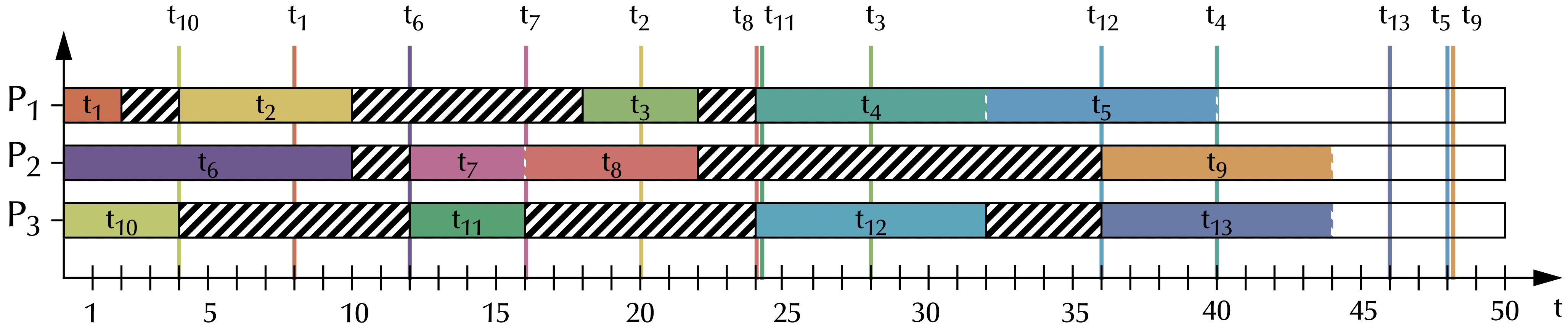
t_{11}: [-, -, t_{10}]; t_{12}: [t_3, t_7, t_{11}]; t_{13}: [-, t_8, -]

Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm (Shen, Ramamritham, Stankovic, '93)

Postrun schedule S'



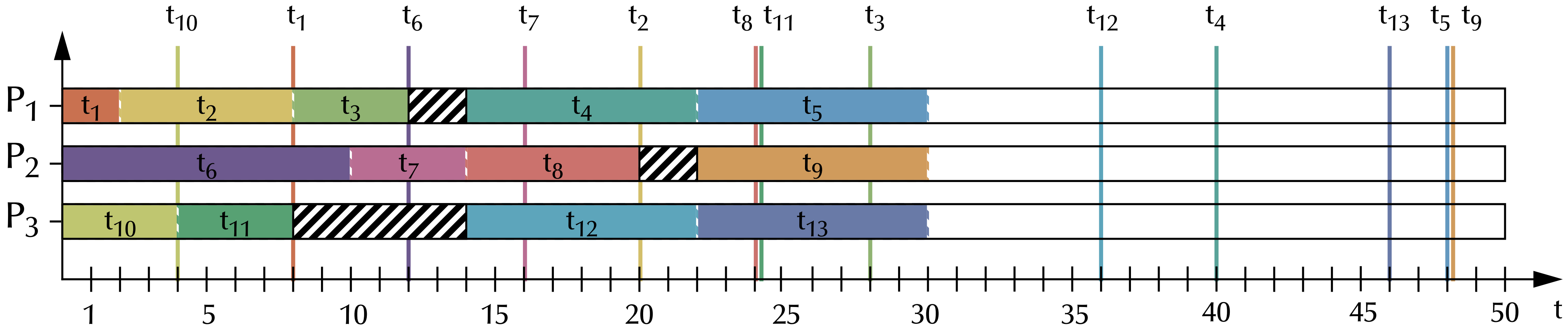
☞ *RVs*: t_1: [-, -, -]; t_2: [t_1, -, -]; t_3: [t_2, -, -]; t_4: [t_3, t_7, t_{10}]; t_5: [t_4, -, -];
t_6: [-, -, -]; t_7: [-, t_6, -]; t_8: [-, t_7, t_{10}]; t_9: [t_4, t_8, t_{12}]; t_{10}: [-, -, -];
t_{11}: [-, -, t_{10}]; t_{12}: [t_3, t_7, t_{11}]; t_{13}: [-, t_8, -]

Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm (Shen, Ramamritham, Stankovic, '93)

Restriction vector based resource reclaiming



☞ *RVs:* t_1: [-, -, -]; t_2: [t_1, -, -]; t_3: [t_2, -, -]; t_4: [t_3, t_7, t_{10}]; t_5: [t_4, -, -];
t_6: [-, -, -]; t_7: [-, t_6, -]; t_8: [-, t_7, t_{10}]; t_9: [t_4, t_8, t_{12}]; t_{10}: [-, -, -];
t_{11}: [-, -, t_{10}]; t_{12}: [t_3, t_7, t_{11}]; t_{13}: [-, t_8, -]

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm (Shen, Ramamritham, Stankovic, '93)

Proof of Correctness

Lemma: Given a feasible prerun schedule S: if $\exists t_i \big| (st_i' > st_i)$ then passing must have occurred.

Proof: *Assuming that no passing occurred,*

then all $t \in t_{<i}$ have been dispatched before t_i and all $t \in t_{>i}$ are only dispatched after t_i completed.

By definition of a feasible schedule all $t \in t_{\sim i}$ do not interfere with t_i and can thus by no means delay the execution of t_i.

Therefore $st_i' \leq st_i$ ↯

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm (Shen, Ramamritham, Stankovic, '93)

Proof of Correctness

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

Proof: By the above lemma, passing occurred if S' is incorrect, i.e.

$$\exists t_i, t_j \,\middle|\, (st_j > ft_i) \wedge (st_j' < st_i') \wedge (st_i' > st_i).$$

Two cases need to be distinguished:

case 1: t_i and t_j have resource or precedence conflicts, then t_i is directly or transitively included in the restriction vector RV_j. Therefore this case of passing is prevented by the RV-algorithm.

case 2: t_i and t_j have no resource or precedence conflicts. In this case t_j cannot delay the execution of t_i by means of passing and the postrun schedule S' would be correct still.

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

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, '97)

- **Restriction vector (RV) with static processor assignment:**

$$RV_i[j] = \begin{cases} t_k \in t_{<i}(j) \,\middle|\, (\neg \exists t_l \in t_{<i}(j) \,|\, st_l > st_k) & \text{if } j = proc(i) \\ t_m \in t_{<i}(j) \,\middle|\, (t_m < t_i \vee t_m \otimes t_i) \wedge (\neg \exists t_l \in t_{<i}(j) \,|\, (st_l > st_m) \wedge (t_l < t_i \vee t_l \otimes t_i)) & \text{if } j \neq proc(i) \\ \text{---} & \text{no such task} \end{cases}$$

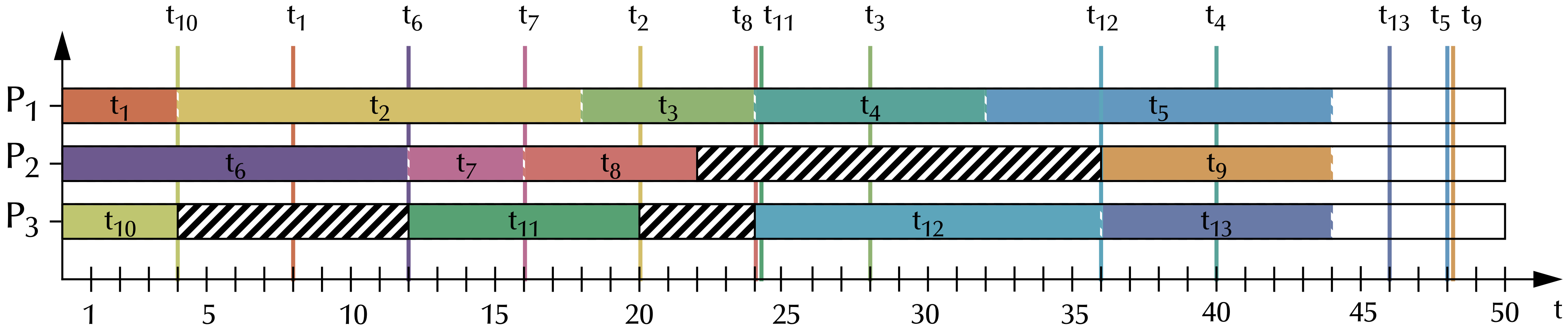
- **Restriction vector (RV) with dynamic processor assignment:**

$$RV_i[j] = \begin{cases} t_m \in t_{<i}(j) \,\middle|\, (t_m < t_i \vee t_m \otimes t_i) \wedge (\neg \exists t_l \in t_{<i}(j) \,|\, (st_l > st_m) \wedge (t_l < t_i \vee t_l \otimes t_i)) & \text{if } t_m \text{ exists} \\ \text{---} & \text{no such task} \end{cases}$$

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, ‘97)

Prerun schedule *S*



☞ *RVs*:

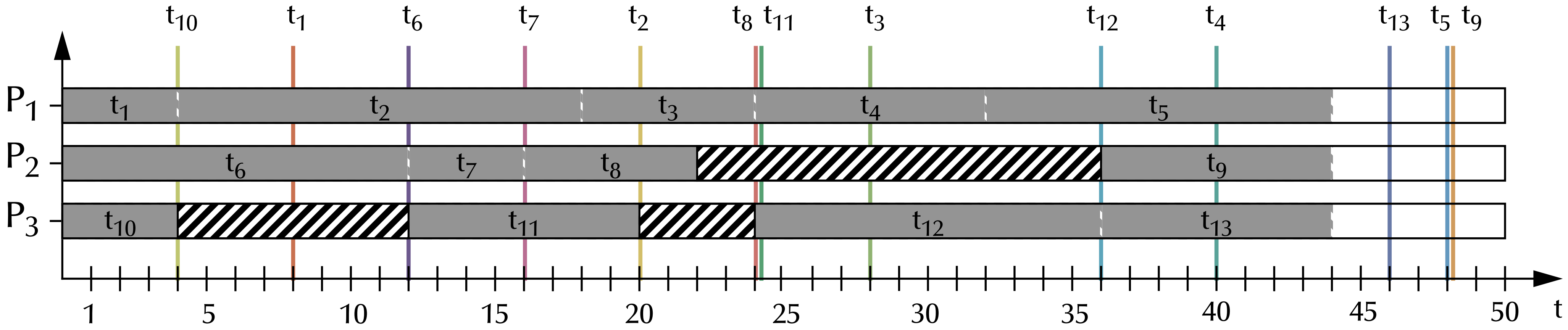
t_1: [-, -, -]; t_2: [t_1, -, -]; t_3: [t_1, -, -]; t_4: [-, t_7, t_{10}]; t_5: [-, -, -];
t_6: [-, -, -]; t_7: [-, -, -]; t_8: [-, -, t_{10}]; t_9: [t_4, t_8, t_{12}]; t_{10}: [-, -, -];
t_{11}: [-, -, -]; t_{12}: [t_3, t_7, t_{11}]; t_{13}: [-, t_8, -]

Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

- Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, '97)

Prerun schedule S

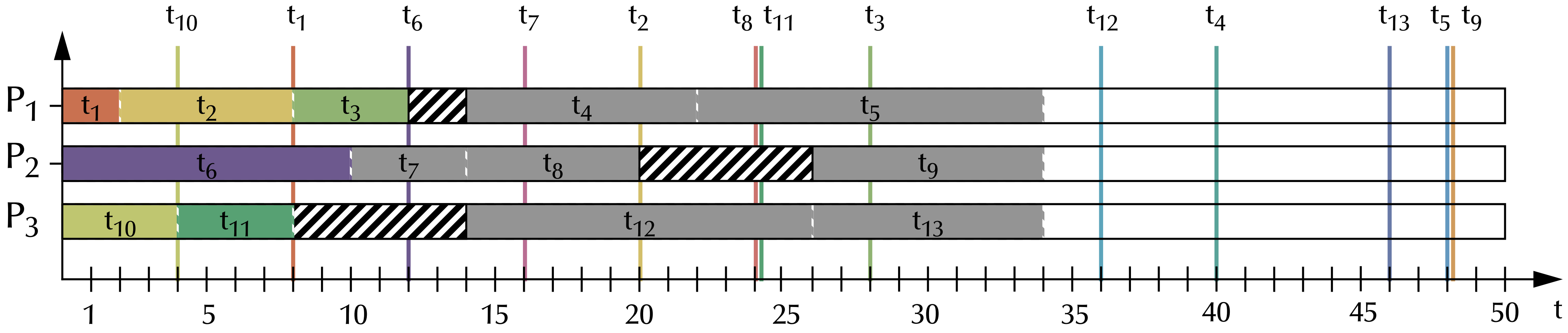


- *RVs*: t_1: [-, -, -]; t_2: [t_1, -, -]; t_3: [t_1, -, -]; t_4: [-, t_7, t_{10}]; t_5: [-, -, -];
  t_6: [-, -, -]; t_7: [-, -, -]; t_8: [-, -, t_{10}]; t_9: [t_4, t_8, t_{12}]; t_{10}: [-, -, -];
  t_{11}: [-, -, -]; t_{12}: [t_3, t_7, t_{11}]; t_{13}: [-, t_8, -]

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, '97)

Postrun schedule S' with task migration



☞ *RV*s: t_1: [-, -, -]; t_2: [t_1, -, -]; t_3: [t_1, -, -]; t_4: [-, t_7, t_{10}]; t_5: [-, -, -];

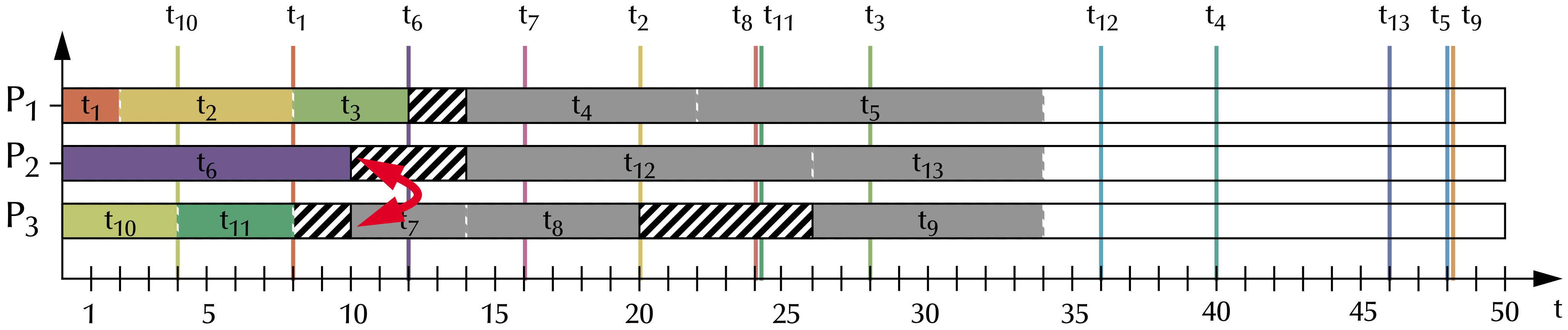
t_6: [-, -, -]; t_7: [-, -, -]; t_8: [-, -, t_{10}]; t_9: [t_4, t_8, t_{12}]; t_{10}: [-, -, -];

t_{11}: [-, -, -]; t_{12}: [t_3, t_7, t_{11}]; t_{13}: [-, t_8, -]

Resource reclaiming from interdependent tasks

- ☞ Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, ‘97)

Postrun schedule S' with task migration



☞ *RVs:*

t_1: [-, -, -]; t_2: [t_1, -, -]; t_3: [t_1, -, -]; t_4: [-, t_7, t_{10}]; t_5: [-, -, -];

t_6: [-, -, -]; t_7: [-, -, -]; t_8: [-, -, t_{10}]; t_9: [t_4, t_8, t_{12}]; t_{10}: [-, -, -];

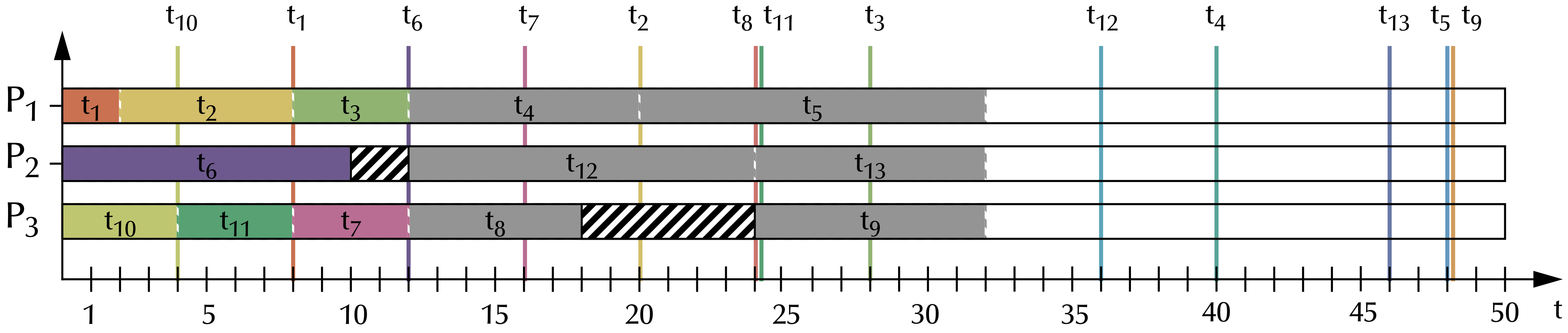
t_{11}: [-, -, -]; t_{12}: [t_3, t_7, t_{11}]; t_{13}: [-, t_8, -]

Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, '97)

Postrun schedule S' with task migration

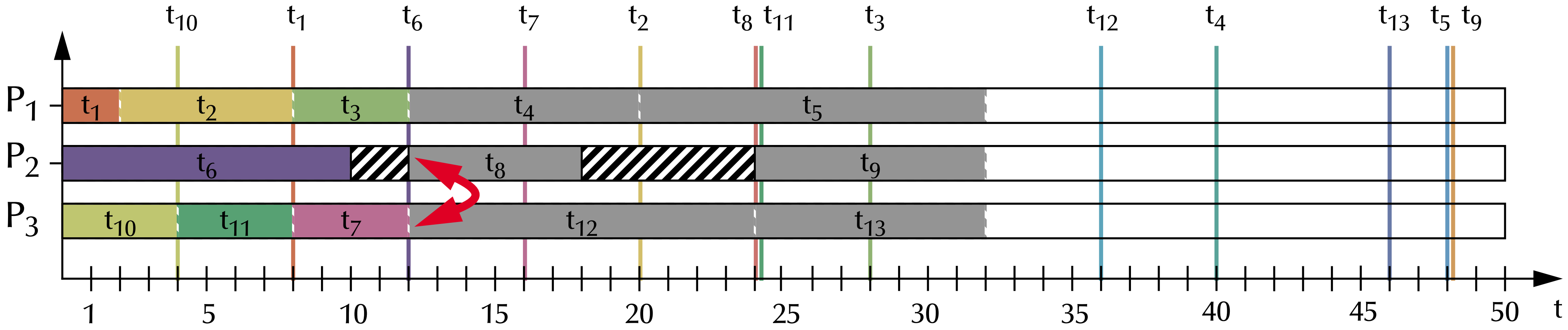


☞ *RVs*: t_1: [-, -, -]; t_2: [t_1, -, -]; t_3: [t_1, -, -]; t_4: [-, t_7, t_{10}]; t_5: [-, -, -];
t_6: [-, -, -]; t_7: [-, -, -]; t_8: [-, -, t_{10}]; t_9: [t_4, t_8, t_{12}]; t_{10}: [-, -, -];
t_{11}: [-, -, -]; t_{12}: [t_3, t_7, t_{11}]; t_{13}: [-, t_8, -]

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, ‘97)

Postrun schedule S' with task migration

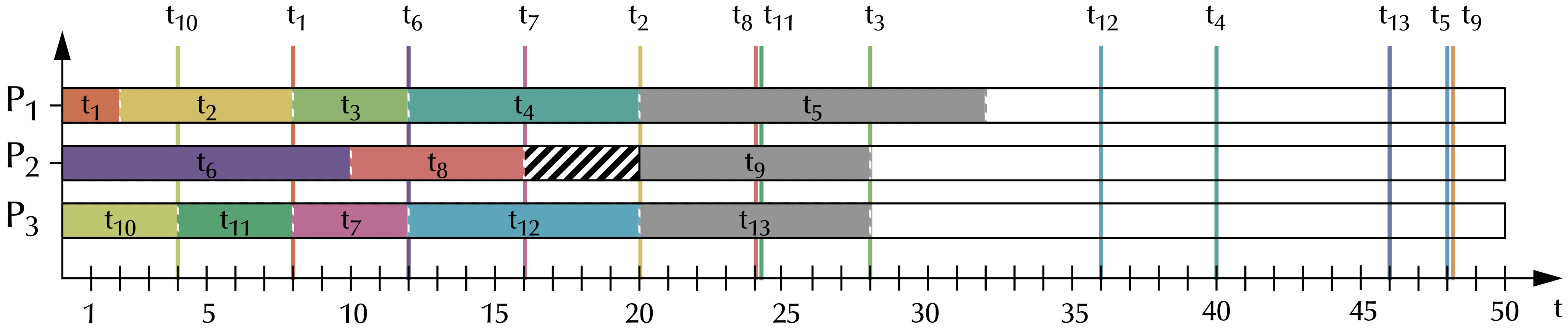


☞ *RVs*:
t_1: [-, -, -]; t_2: [t_1, -, -]; t_3: [t_1, -, -]; t_4: [-, t_7, t_{10}]; t_5: [-, -, -];
t_6: [-, -, -]; t_7: [-, -, -]; t_8: [-, -, t_{10}]; t_9: [t_4, t_8, t_{12}]; t_{10}: [-, -, -];
t_{11}: [-, -, -]; t_{12}: [t_3, t_7, t_{11}]; t_{13}: [-, t_8, -]

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, '97)

Postrun schedule S' with task migration



☞ *RVs*:

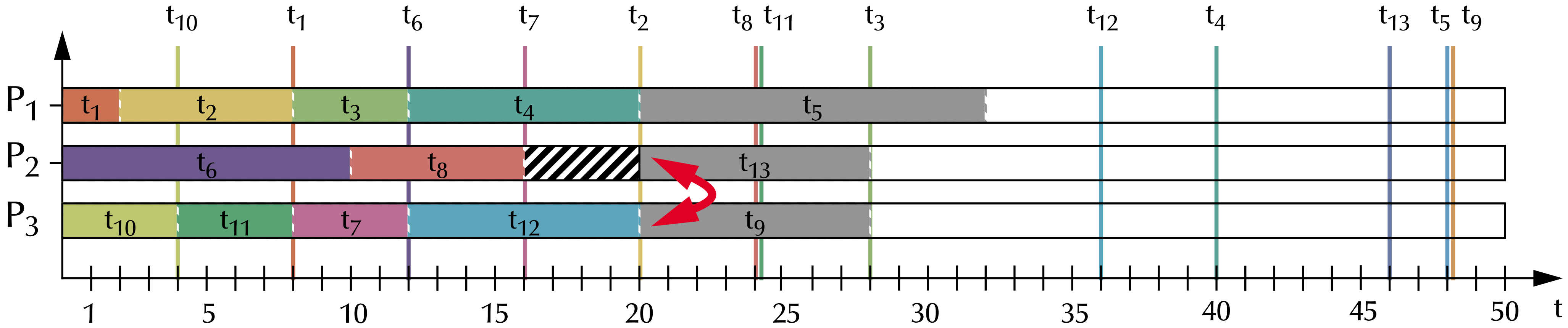
t_1: [-, -, -];	t_2: [t_1, -, -];	t_3: [t_1, -, -];	t_4: [-, t_7, t_{10}];	t_5: [-, -, -];
t_6: [-, -, -];	t_7: [-, -, -];	t_8: [-, -, t_{10}];	t_9: [t_4, t_8, t_{12}];	t_{10}: [-, -, -];
t_{11}: [-, -, -];	t_{12}: [t_3, t_7, t_{11}];	t_{13}: [-, t_8, -]		

Resource Reclaiming [Murthy2001]

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, '97)

Postrun schedule S' with task migration

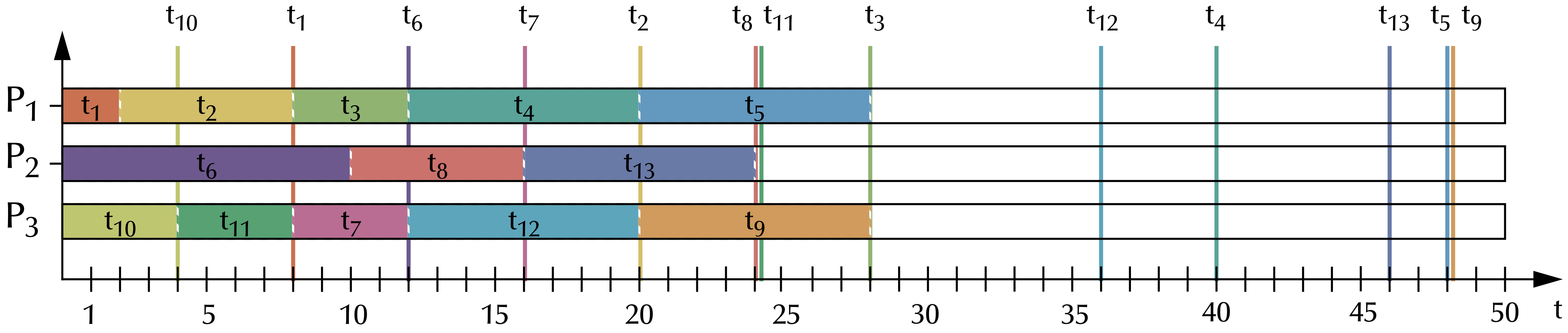


☞ *RVs*:
t_1: [-, -, -]; t_2: [t_1, -, -]; t_3: [t_1, -, -]; t_4: [-, t_7, t_{10}]; t_5: [-, -, -];
t_6: [-, -, -]; t_7: [-, -, -]; t_8: [-, -, t_{10}]; t_9: [t_4, t_8, t_{12}]; t_{10}: [-, -, -];
t_{11}: [-, -, -]; t_{12}: [t_3, t_7, t_{11}]; t_{13}: [-, t_8, -]

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, ‘97)

Postrun schedule S' with task migration



☞ *RVs*:

t_1: [-, -, -]; t_2: [t_1, -, -]; t_3: [t_1, -, -]; t_4: [-, t_7, t_{10}]; t_5: [-, -, -];

t_6: [-, -, -]; t_7: [-, -, -]; t_8: [-, -, t_{10}]; t_9: [t_4, t_8, t_{12}]; t_{10}: [-, -, -];

t_{11}: [-, -, -]; t_{12}: [t_3, t_7, t_{11}]; t_{13}: [-, t_8, -]

Resource reclaiming from interdependent tasks

☞ Restriction vectors (RV) algorithm with task migration (Manimaran, Murthy, '97)

Correctness of the migration process

To ensure that the swapping of dispatching queues $DQ_x \leftrightarrow DQ_y$ between processor P_x and P_y does not interfere with the correctness of the postrun schedule S', swapping is permitted only if:

$$st_i \geq ft_j$$

the currently blocked task t_i is not further delayed

(where task t_i is next to be scheduled on the idling P_x and task t_j is currently executing on P_y).

The unrestricted and executable task t_k, which is next to be scheduled on P_y is started earlier by transferring it to the idling P_x.

☞ no task is delayed by swapping these dispatching queues.

Resource reclaiming evaluated

Some additional observables:

- **Task graph density** $P_p \rightarrow [0 \ldots 1]$, where zero indicates and independent and one a fully dependent task-set.
- **aw-ratio**: C_i' / C_i (actual to worst case ratio)
- **mig-attempts**: number of checks on dispatch queues by the RV with migration algorithm

RC computational costs (from Manimaran, Murthy, Vijay, Ramamritham '97):

- $C_{RC\text{-}basic} = 1$
- $C_{RC\text{-}early\text{-}start} = mC_{RC\text{-}basic}$; with m the number of processors
- $C_{RC\text{-}RV} = C_{RC\text{-}early\text{-}start} + C_{RV}$; with C_{RV} the cost for the calculation of the *RV*s.
- $C_{RC\text{-}RV\text{-}migration} = C_{RC\text{-}RV} + f(mig\text{-}attempts, C_{RC\text{-}early\text{-}start})$

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 communication/synchronization overhead can reach noticeable levels.
- There need to be a high degree of dependencies in the task-set (P_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]*

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 exceeds 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 control

- **Resource synchronization primitives**
    - evaluation criteria for resource synchronisation methods
    - atomicity, liveliness, and double interaction
- **Resource reclaiming schemes**
    - basic reclaiming, early start, and restriction vector algorithms
    - resource reclaiming with task migration
- **Real-time resource control**
    - policy and run-time issues to be considered