Research Centre in Real-Time Computing Systems
FCT Research Unit 608

Implementing Multicore Real-Time Scheduling Algorithms Based on Task Splitting Using Ada 2012

Björn Andersson and Luís Miguel Pinho

Ada-Europe 2010, Valencia, Spain
June, 15, 2010

IPP HURRAY!

isep instituto superior de engenharia do porto

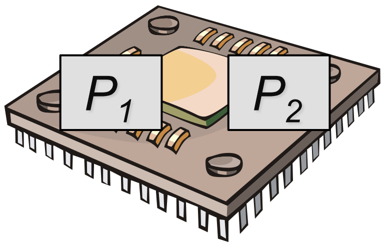
INSTITUTO POLITÉCNICO DO PORTO

Forewood

2000 2005 2010 2015 2020 time

Attempts to transition RM and EDF to multicores.

Development of
multicore scheduling using
the task-splitting
class of algorithms.



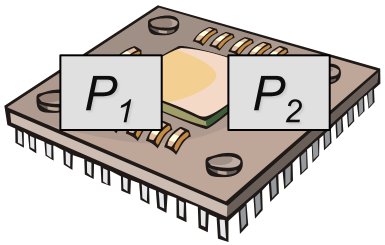
New language constructs for
multicore real-time scheduling
proposed

Forewood

2000 2005 2010 2015 2020 time

Attempts to transition RM and EDF to multicores.

Development of
multicore scheduling using
the task-splitting
class of algorithms.



Question: Can the new language constructs for supporting multicore real-time scheduling be used to implement previously published multicore scheduling algorithms based on task-splitting?

Outline

- System model and terminology
- Understanding task-splitting multiprocessor scheduling
- The new language constructs
- Implementing task-splitting multiprocessor scheduling with the new language constructs
- Discussion and Conclusions

System model

- *m* identical processors
- A task set τ composed of *n* tasks. $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$
- A task τ_i is characterized by T_i, C_i and D_i.
- A task τ_i generates a (potentially infinite) sequence of jobs with at least T_i time units between arrivals of two consecutive jobs.
- A job of τ_i must perform C_i units of execution at most D_i time units from its arrival.

Terminology

- For an implicit-deadline task set, it holds that: $\forall i$: $D_i = T_i$.
- For a constrained-deadline task set, it holds that: $\forall i$: $D_i \leq T_i$.
- For an arbitrary-deadline task set, there are no restrictions on D_i and T_i.
- The utilization of a task set is $U=(1/m)^{*}\times\sum_{i=1..n} C_i/T_i$
- The utilization bound of a scheduling algorithm A is the maximum number UB_A such that for each task set with utilization at most for UB_A and $\forall i{:}C_i \leq D_i$ it implies that all deadlines are met.

Design space of multiprocessor scheduling algorithms

		Priority restriction		
		task-static	job-static	dynamic
non-preemptive	Migration allowed	UB≤0.5		
	Migration not allowed			
preemptive	Migration allowed			
	Migration not allowed			

Design space of multiprocessor scheduling algorithms

		Priority restriction		
		task-static	job-static	dynamic
non-preemptive	Migration allowed			
	Migration not allowed		$UB \leq 0.5$	
preemptive	Migration allowed			
	Migration not allowed			

Task splitting algorithms are here.

Illustration of Task Splitting

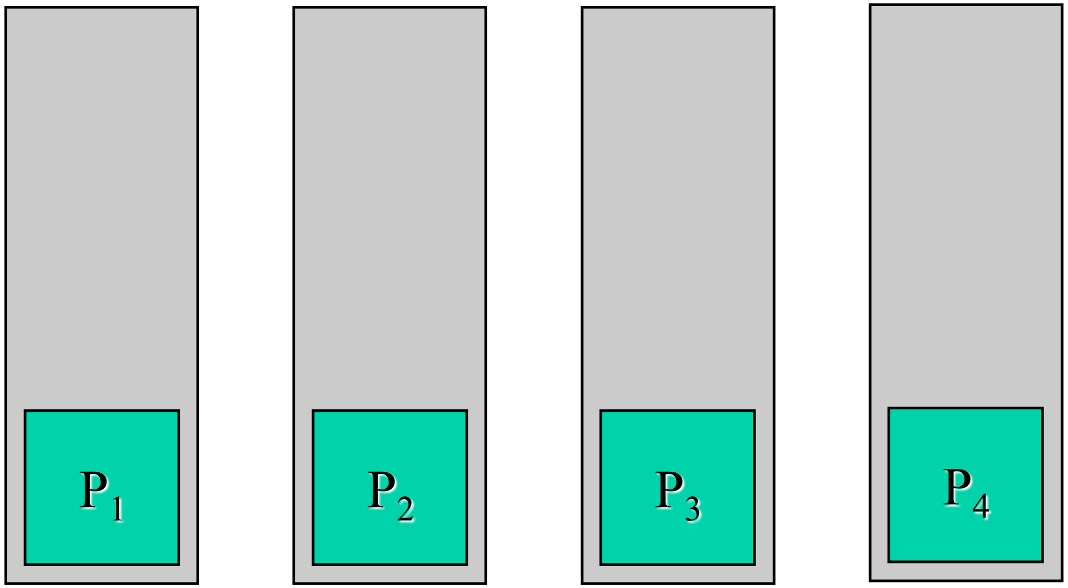


Illustration of Task Splitting

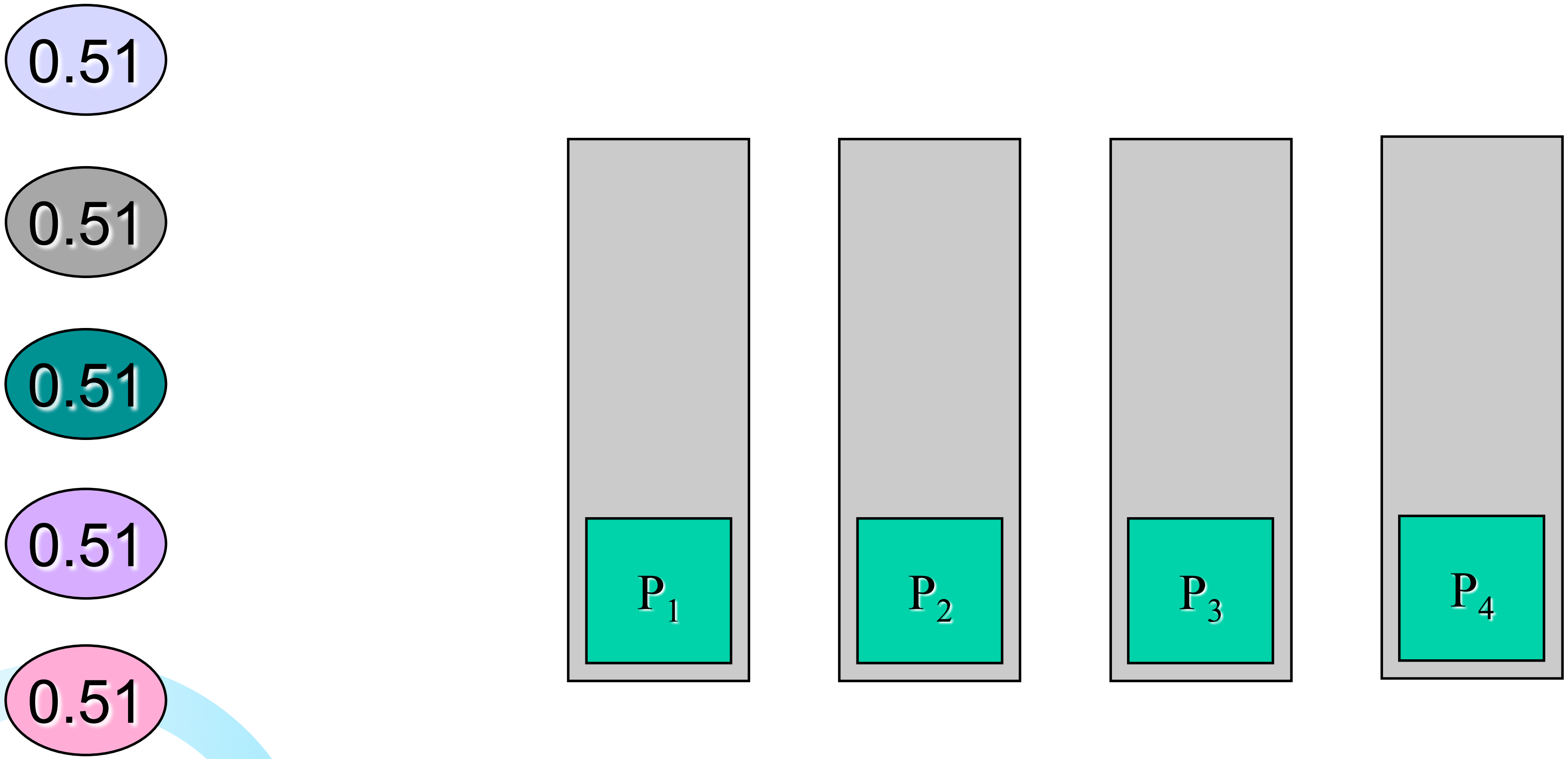


Illustration of Task Splitting

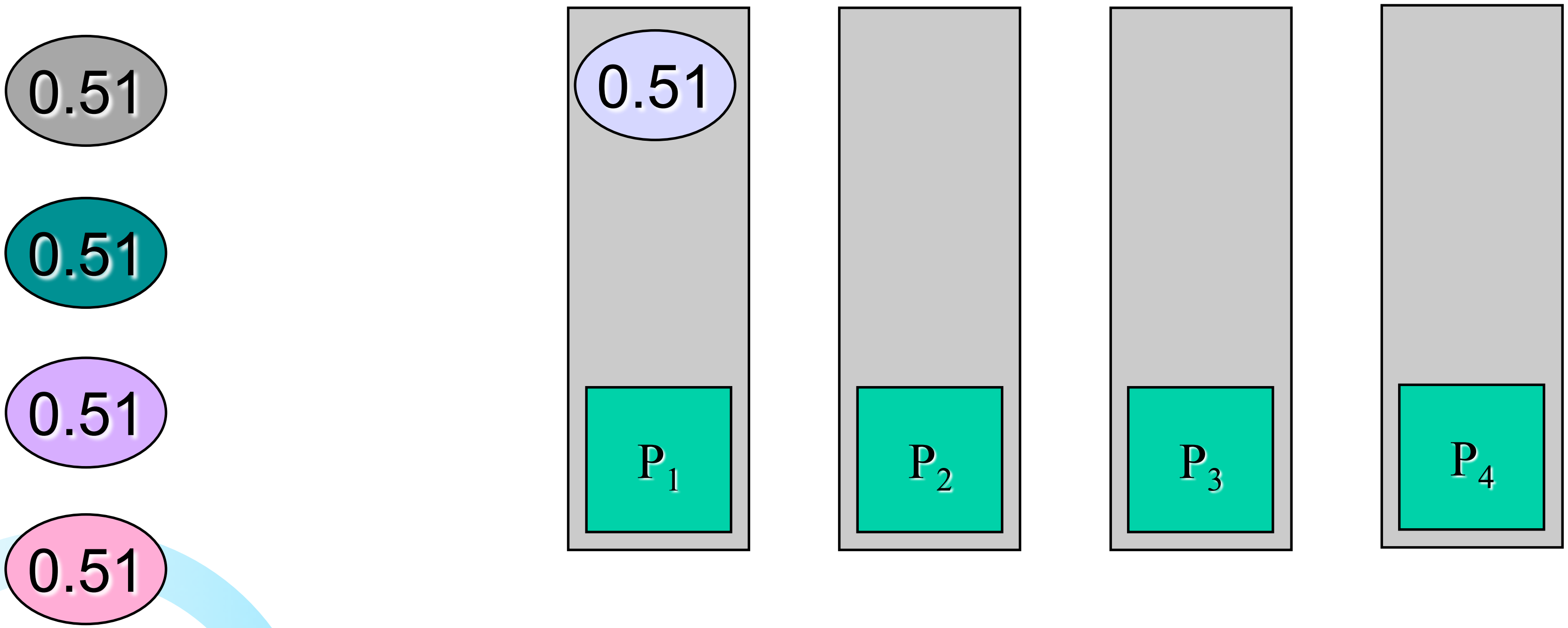


Illustration of Task Splitting

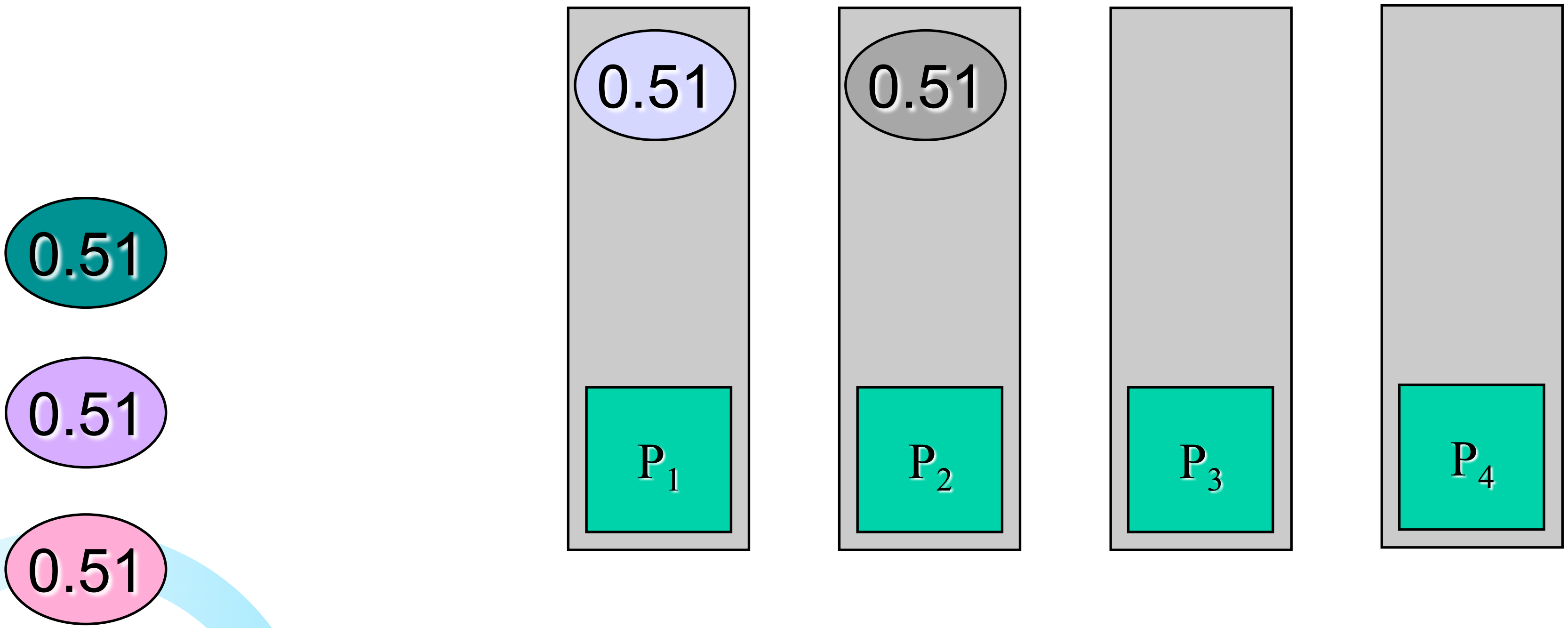


Illustration of Task Splitting

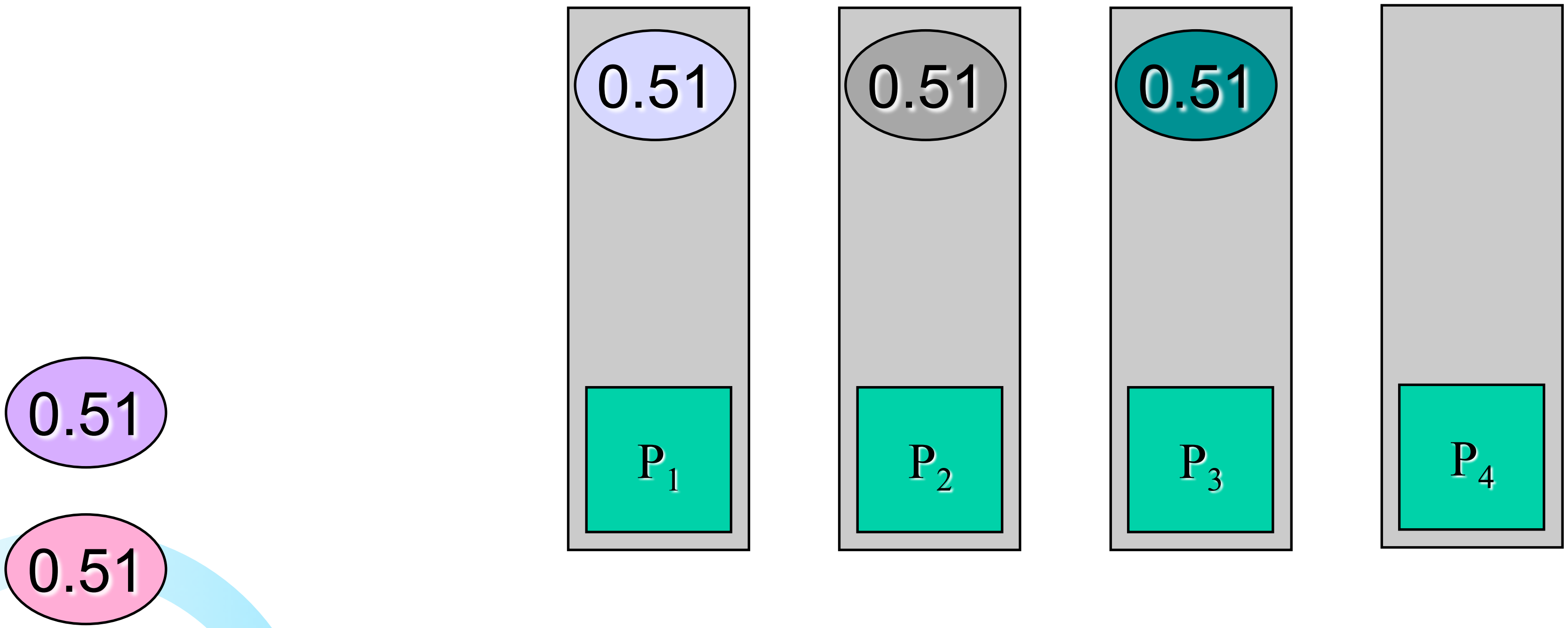


Illustration of Task Splitting

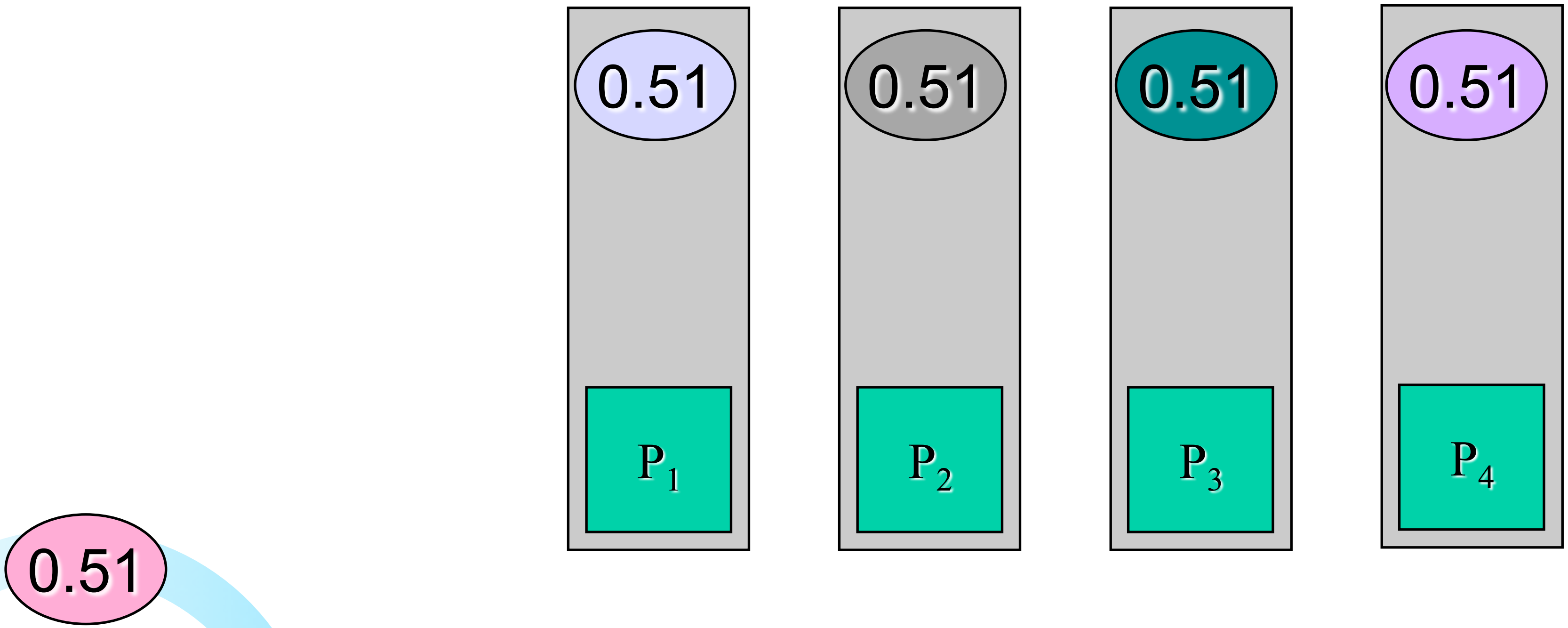


Illustration of Task Splitting

We can split it

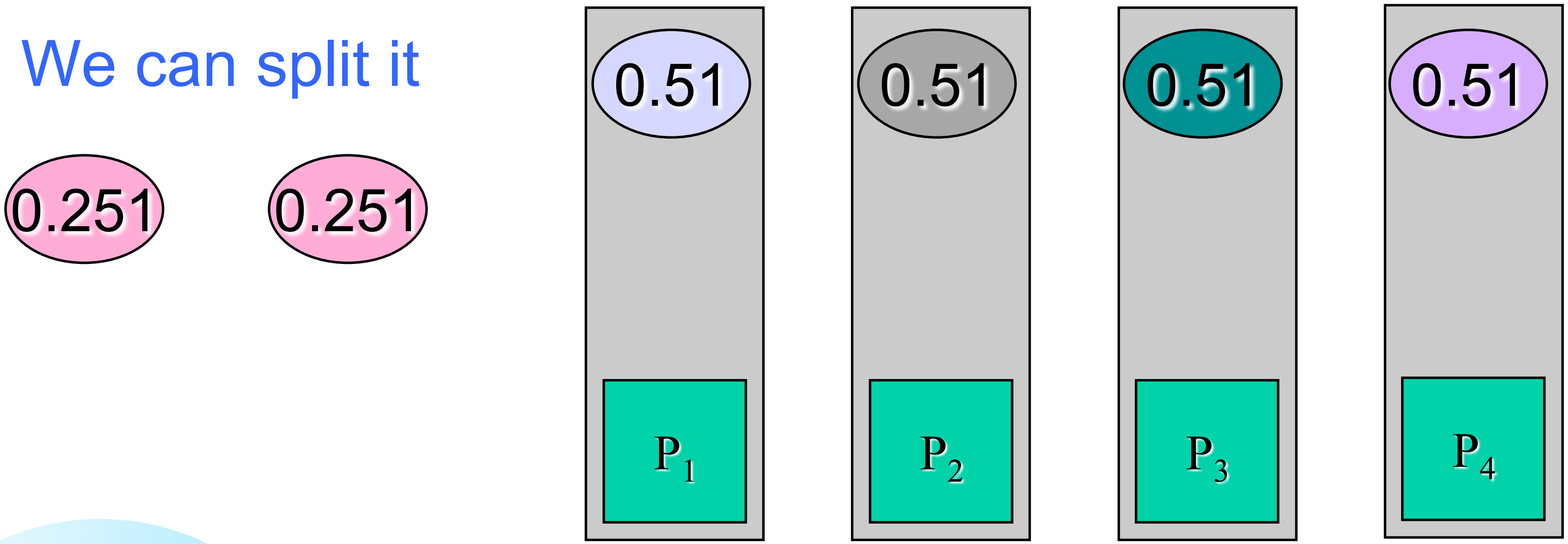


Illustration of Task Splitting

And now it is possible to allocate the task(s)

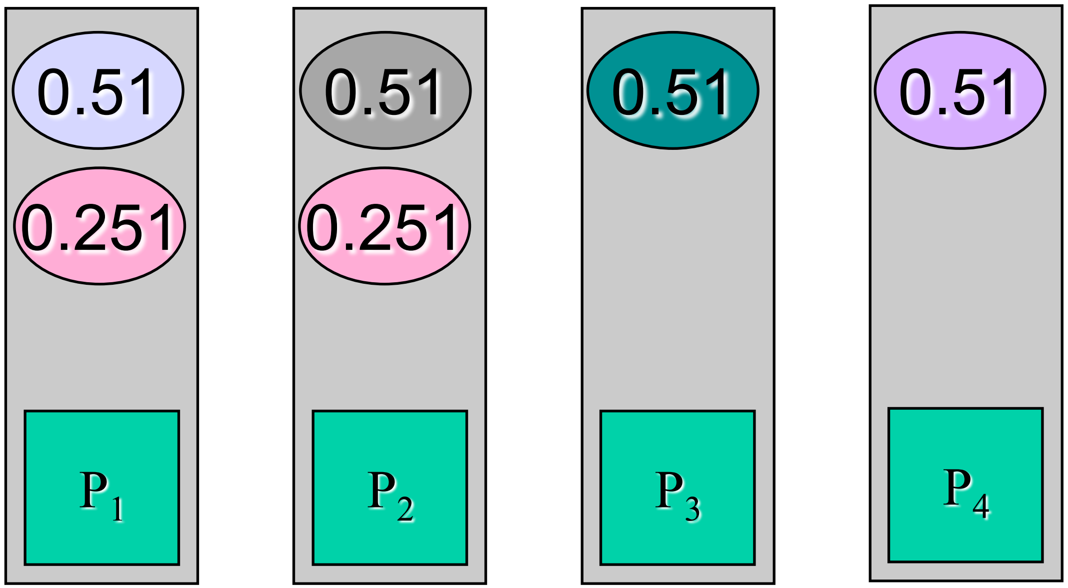
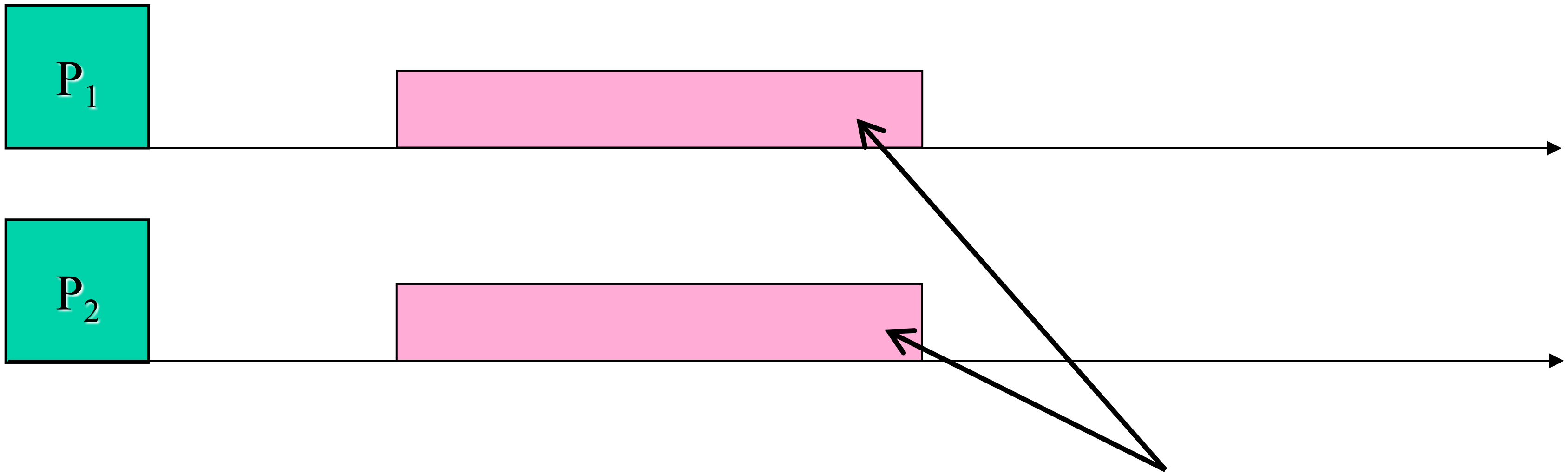


Illustration of Task Splitting



It can happen that two pieces of a split task executes simultaneously.

We need a dispatcher that avoids this.

Different types of split-task dispatching

- Slot-based split-task dispatching
- Job-based split-task dispatching
- Suspension-based split-task dispatching

Different types of split-task dispatching

- Slot-based split-task dispatching
- Job-based split-task dispatching
- Suspension-based split-task dispatching

These types of algorithms have requirement sets where they are superior.

Different types of split-task dispatching

- Slot-based split-task dispatching
- Job-based split-task dispatching
- Suspension-based split-task dispatching

This type of algorithms has no requirement set where it is superior.

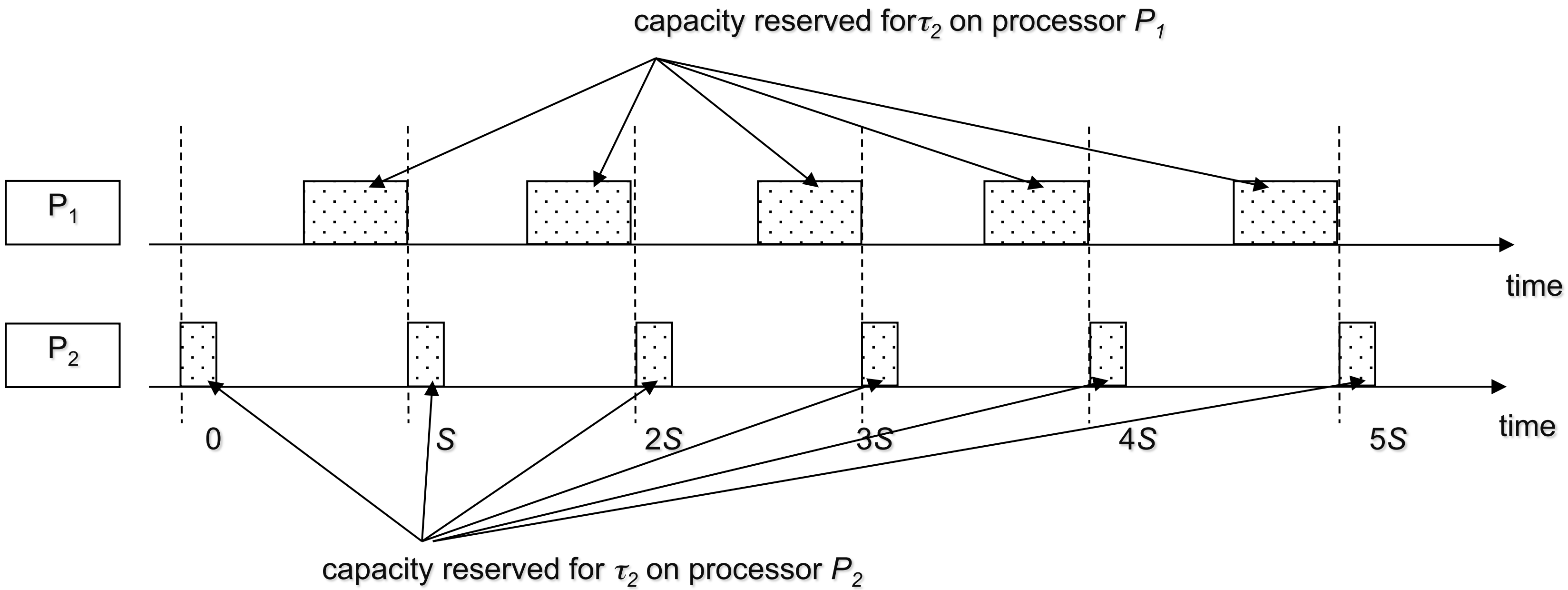
Different types of split-task dispatching

- Slot-based split-task dispatching
- Job-based split-task dispatching
- Suspension-based split-task dispatching

We will only discuss these.

Slot-based split-task dispatching: assign reserves for the split tasks

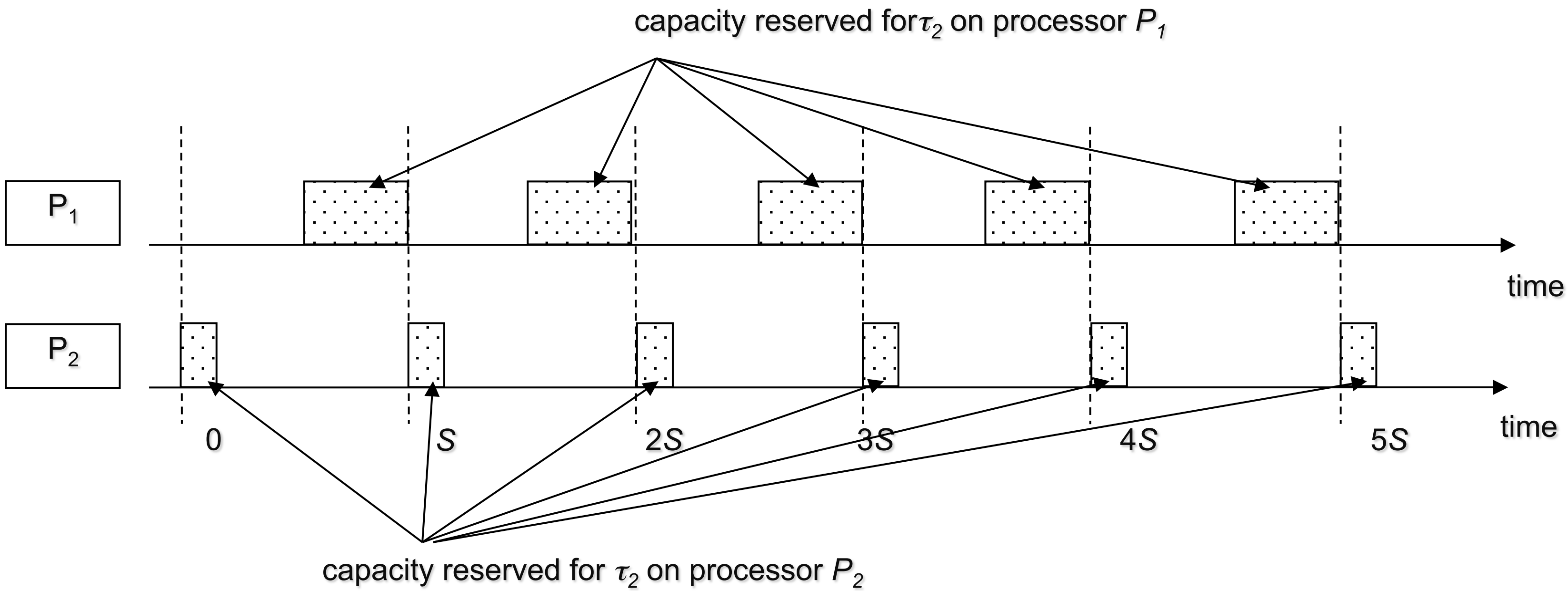
Let τ_2 denote a task that is split between processor 1 and processor 2.



A split task is only allowed to execute in its reserve.

Slot-based split-task dispatching: assign reserves for the split tasks

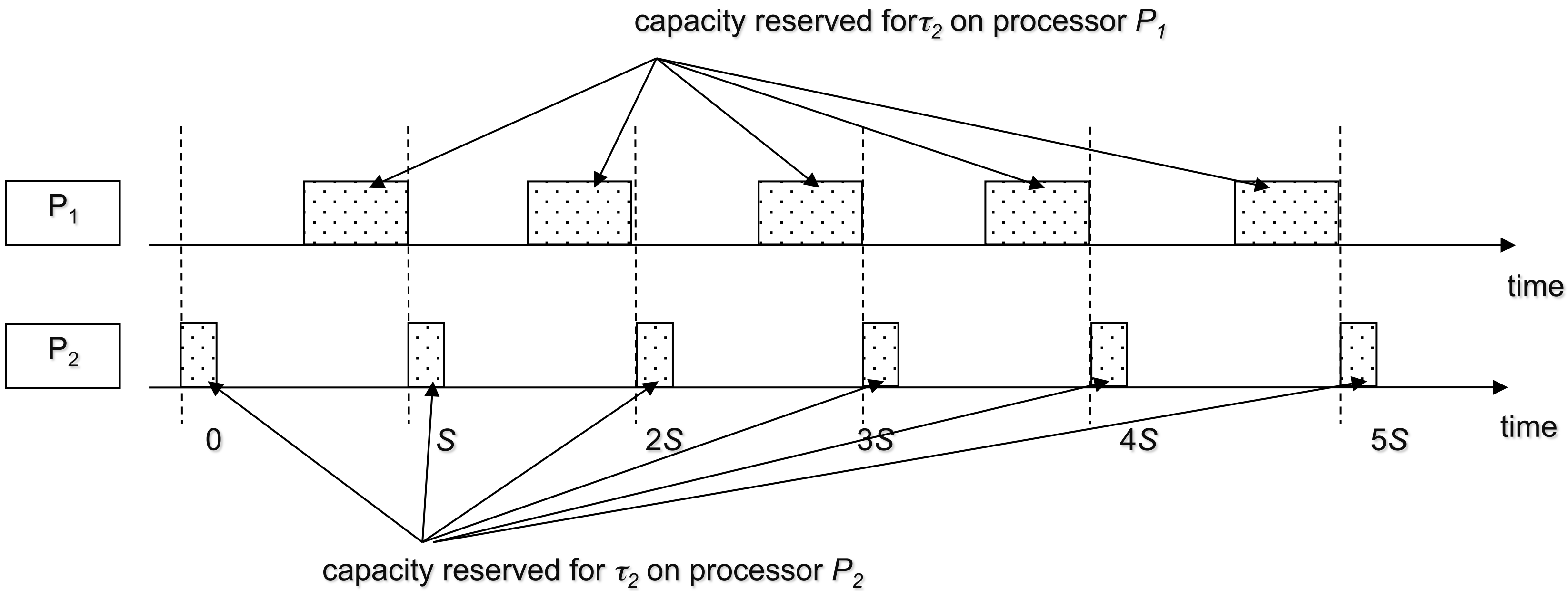
Let τ_2 denote a task that is split between processor 1 and processor 2.



A split task executes with the highest priority in its reserve.

Slot-based split-task dispatching: assign reserves for the split tasks

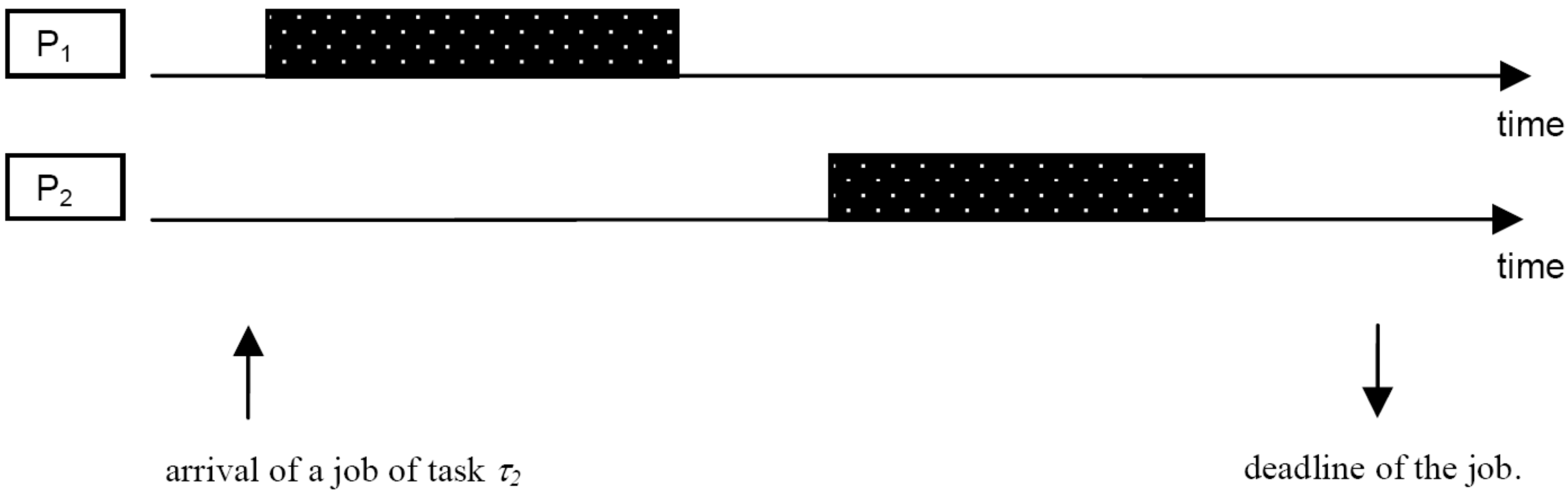
Let τ_2 denote a task that is split between processor 1 and processor 2.



If processor *p* does not execute a split task at time *t* then it executes at time *t* the non-split task assigned to processor *p* with the highest priority at time *t*.

Job-based split-task dispatching: assign subdeadlines and offsets to “pieces” of the split tasks

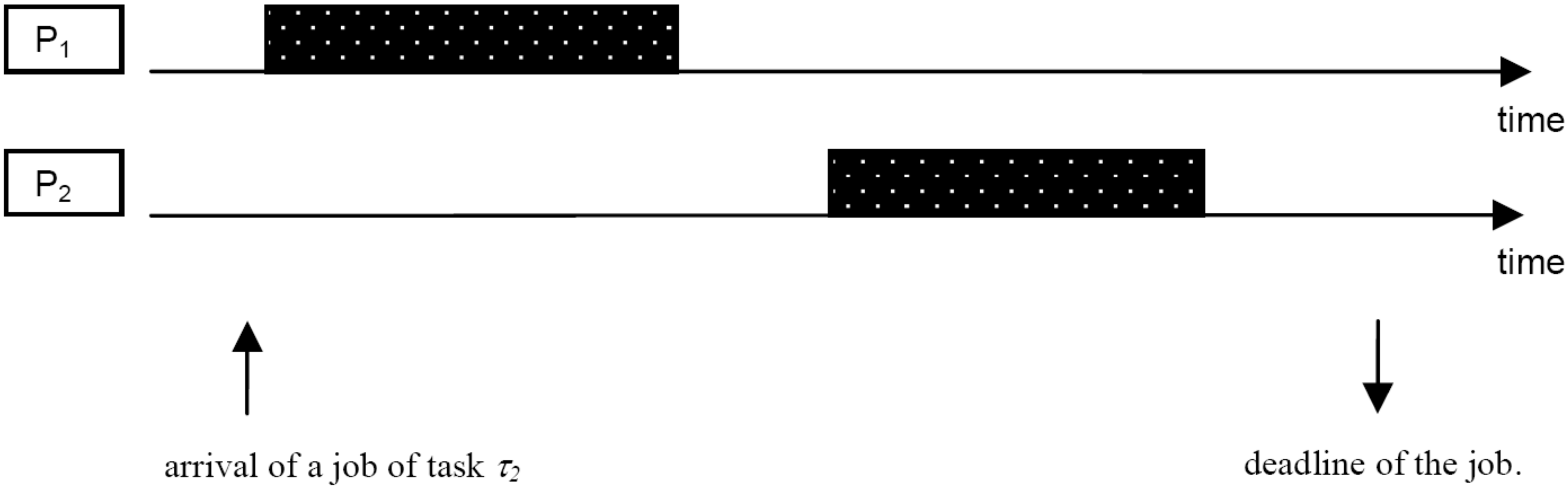
τ_2 is a split task. When a job of τ_2 arrives, it executes on processor 1 and then it migrates to processor 2.



We let C_2' and D_2' denote the execution time of the “piece” of τ_2 that is assigned to P_1.

Job-based split-task dispatching: assign subdeadlines and offsets to "pieces" of the split tasks

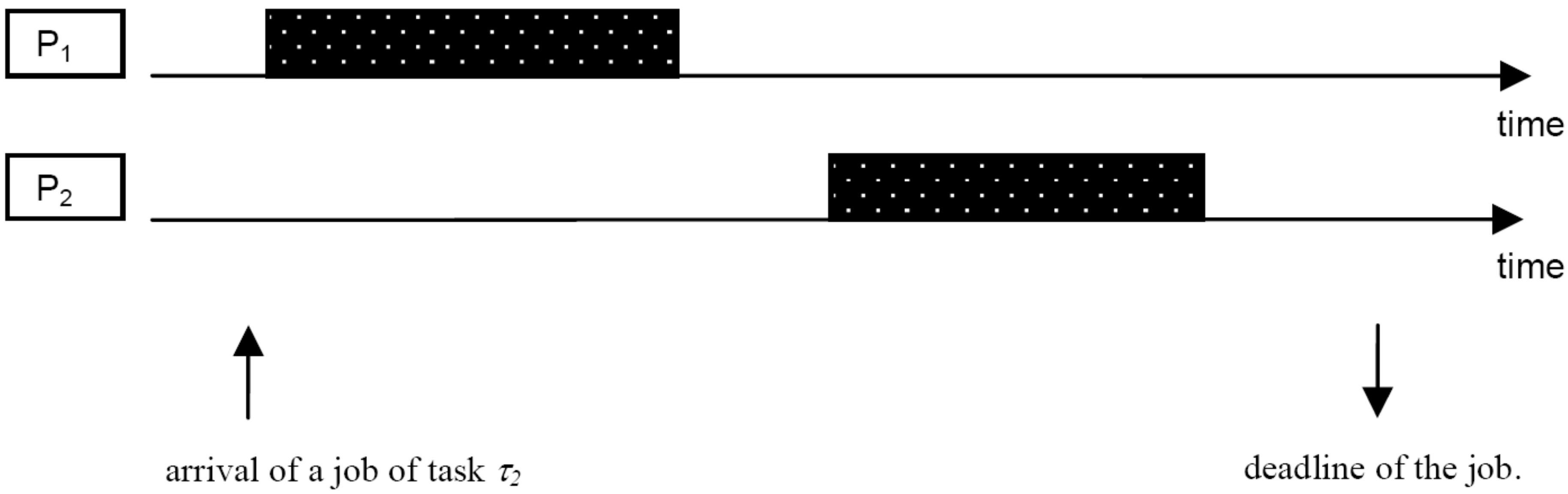
τ_2 is a split task. When a job of τ_2 arrives, it executes on processor 1 and then it migrates to processor 2.



We let C_2'' and D_2'' denote the execution time of the "piece" of τ_2 that is assigned to P_2.

Job-based split-task dispatching: assign subdeadlines and offsets to “pieces” of the split tasks

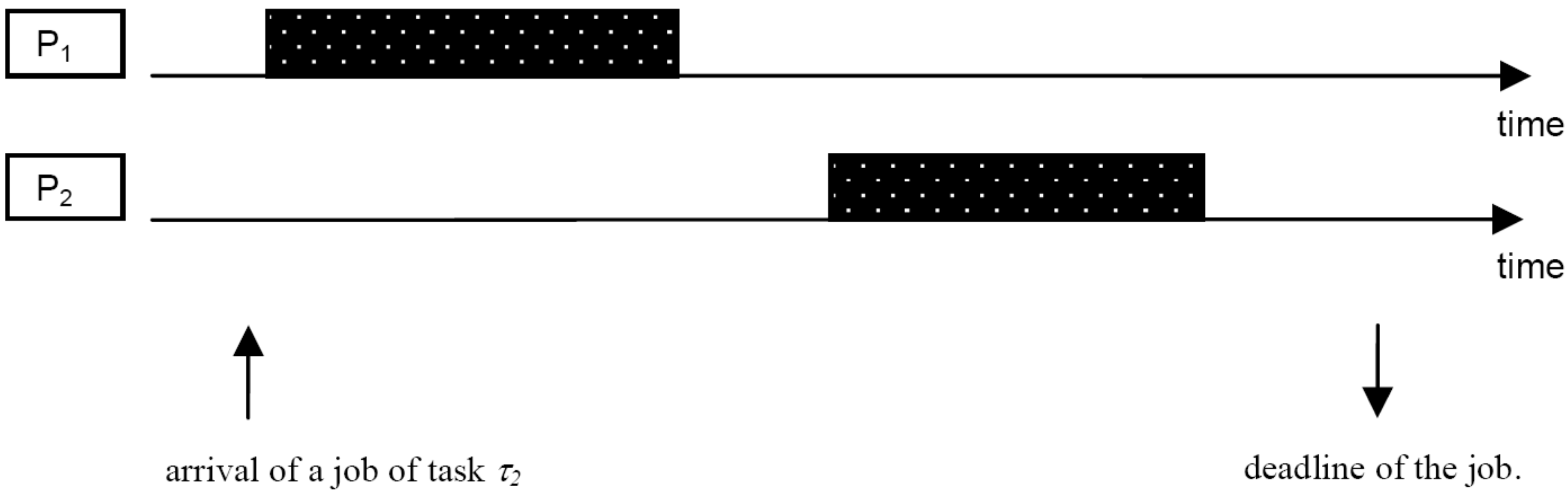
τ_2 is a split task. When a job of τ_2 arrives, it executes on processor 1 and then it migrates to processor 2.



We select C_2' and C_2'' as $C_2'+C_2''=C_2$.

Job-based split-task dispatching: assign subdeadlines and offsets to “pieces” of the split tasks

τ_2 is a split task. When a job of τ_2 arrives, it executes on processor 1 and then it migrates to processor 2.



We select D_2' and D_2'' as $D_2'+D_2''=D_2$.

New language constructs (recalling previous presentation)

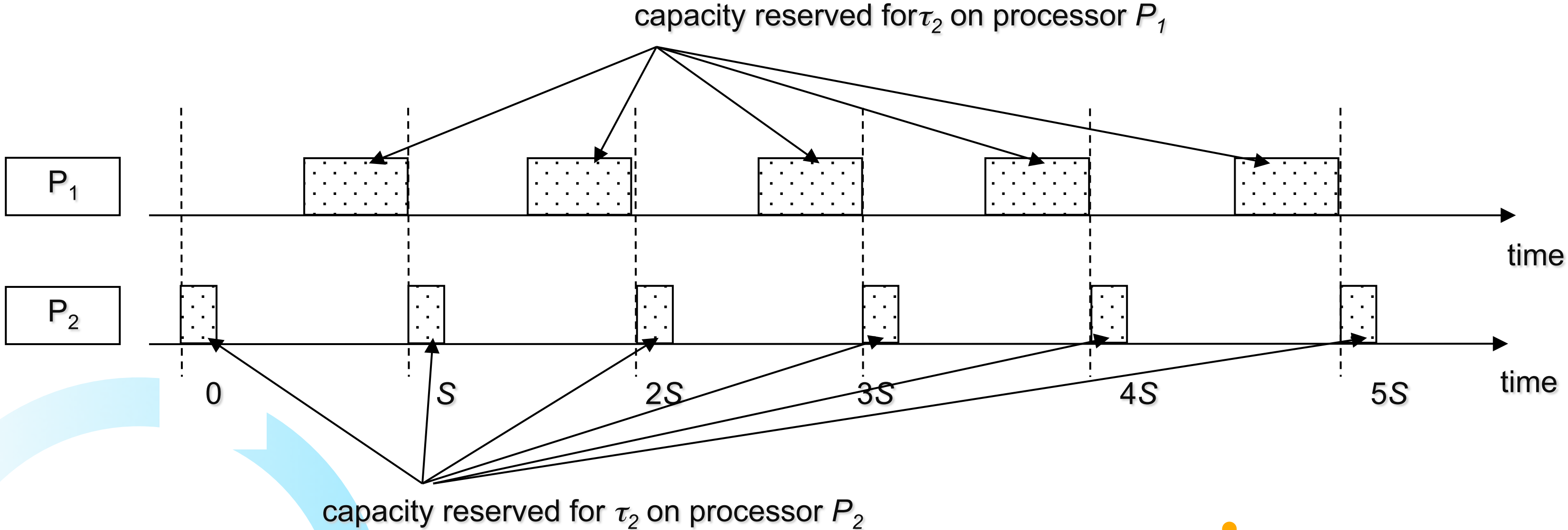
The extension defines packages for handling the CPUs available, and the creation of dispatching domains.

We are deadling with a single domain so our main interest is in Set_CPU and Delay_Until_And_Set_CPU

```
package Ada.Dispatching is
   type Dispatching_Domain_Policy is private;
   -- other declared types and subprograms not shown here
end Ada.Dispatching;

package Ada.Dispatching.Domains is
   type Dispatching_Domain is  private;
   System_Dispatching_Domain: Dispatching_Domain;

   -- other declared subprograms not shown here

   procedure Set_CPU(P : in CPU_Range;
                     T : in Task_Id := Current_Task);

   procedure Delay_Until_And_Set_CPU(
                     Delay_Until_Time : in Ada.Real_Time.Time;
                     P : in CPU_Range);
end Ada.Dispatching_Domains;
```

Implementing split-task multiprocessor scheduling: slot-based split-task dispatching

- As seen in the example
  - A high priority band is used for the split tasks' slots
  - Asynchronous task control is used to suspend a task if it has reached the end of left slot
  - Timing events manage the dispatching points
  - Management encapsulated in a Protected Object



Implementing split-task multiprocessor scheduling: slot-based split-task dispatching

```
pragma Priority_Specific_Dispatching (EDF_Across_Priorities, 1, 10) ;
pragma Priority_Specific_Dispatching (FIFO_Within_Priorities, 11, 12);

…

protected type Sporadic_Switcher is
  pragma Priority(12);
  procedure Register(ID : Task_ID; Phase_1_CPU, Phase_2_CPU: CPU_Range;
                     Phase_1_Reserve, Phase_2_Reserve : Time_Span);
  procedure Handler(TM :in out Timing_Event);
  procedure Release_Task;
  procedure Finished;
  entry Wait;
private
   -- private data
end Sporadic_Switcher;
```

Implementing split-task multiprocessor scheduling: slot-based split-task dispatching

```
procedure Release_Task is          -- called by someone else or by interrupt
begin
-- decide if release or not depending of phase
    if Release_Time >= Slot_Start and Release_Time < End_of_Phase_1 then
      Set_CPU(Client_Phase_1_CPU, Client_ID);
      Switch_Timer.Set_Handler(End_of_Phase_1, Handler'Access);
      Client_Current_Phase := Phase_1;
      Released := True;
    elsif Release_Time >= Start_of_Phase_2 and Release_Time < End_of_Slot then
      Set_CPU(Client_Phase_2_CPU, Client_ID);
      Switch_Timer.Set_Handler(End_of_Slot, Handler'Access);
      Client_Current_Phase := Phase_2;
      Released := True;
    else
      Client_Current_Phase := Not_Released;
      Switch_Timer.Set_Handler(Start_of_Phase_2, Handler'Access);
    end if;
  end Release_Task;
```

Implementing split-task multiprocessor scheduling: slot-based split-task dispatching

```
procedure Handler(TM :in out Timing_Event) is
 begin
   case Client_Current_Phase is
     when Not_Released =>
       Set_CPU(Client_Phase_2_CPU, Client_ID);
       Switch_Timer.Set_Handler(End_of_Slot, Handler'Access);
       Client_Current_Phase := Phase_2;  Released := True;
     when Phase_1 =>
       Client_Current_Phase := Suspended;
       Switch_Timer.Set_Handler(Start_of_Phase_2, Handler'Access);
       Hold(Client_ID);
    when Suspended =>
       Set_CPU(Client_Phase_2_CPU, Client_ID);
       Switch_Timer.Set_Handler(End_of_Slot, Handler'Access);
       Client_Current_Phase := Phase_2;  Continue(Client_ID);
     when Phase_2 =>
       Set_CPU(Client_Phase_1_CPU, Client_ID);
       Switch_Timer.Set_Handler(End_of_Phase_1, Handler'Access);
       Client_Current_Phase := Phase_1;
   end case;
 end Handler;
```

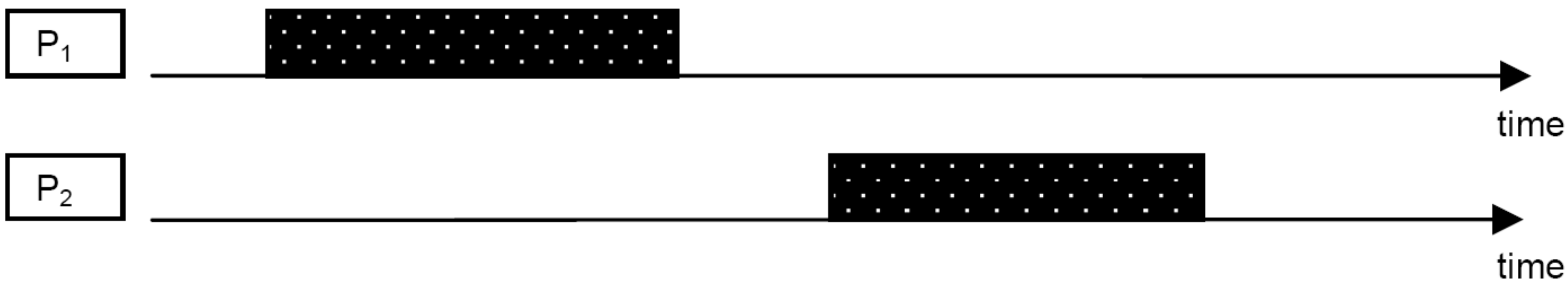
Implementing split-task multiprocessor scheduling: slot-based split-task dispatching

```
task body Task_2 is

begin    My_Switcher.Register(Current_Task,
                 CPU_2, CPU_1,
                 Reserve_Phase_1_Task_2,
                 Reserve_Phase_2_Task_2);
  loop

    My_Switcher.Wait;

    -- Code of application

    My_Switcher.Finished;

  end loop;
end Task_2;
```

Implementing split-task multiprocessor scheduling: job-based split-task dispatching #1

- Simpler
  - Uses Priorities
  - Timing Event to change CPU in the end of phase 1
  - Management encapsulated in a Protected Object

τ_2 is a split task. When a job of τ_2 arrives, it executes on processor 1 and then it migrates to processor 2.



Implementing split-task multiprocessor scheduling: job-based split-task dispatching #1

```
Priority_Task1_First_Phase  : constant Priority := 20;
Priority_Task1_Second_Phase : constant Priority := 19;

Priority_Task2 : constant Priority := 18;
Priority_Task3 : constant Priority := 17;

protected type Job_Based_Switcher is
procedure Register(IID : Task_ID; Phase_1_CPU, Phase_2_CPU: CPU_Range;
                   Phase_1_C, Phase_2_C, Phase_1_D, Phase_2_D: Time_Span;
                   Phase_1_Prio, Phase_2_Prio: Priority);
  procedure Handler(TM :in out Timing_Event);
  procedure Release_Task;
  procedure Finished;
  entry Wait;
private
   -- private data
end Sporadic_Switcher;
```

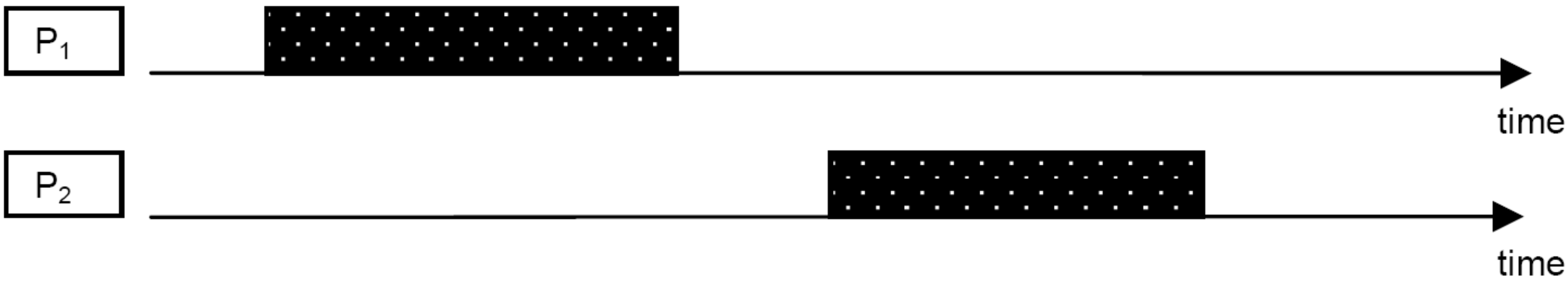
Implementing split-task multiprocessor scheduling: job-based split-task dispatching #1

```
procedure Handler(TM :in out Timing_Event) is
begin
   -- in this algorithm, handler is just called in the end of phase 1
   Set_CPU(Client_Phase_2_CPU, Client_ID);
   Set_Priority(Client_Phase_2_Prio, Client_ID);
end Handler;

procedure Release_Task is
begin
   -- calculate parameters
   -- set first phase parameters
   Set_CPU(Client_Phase_1_CPU, Client_ID);
   Set_Priority(Client_Phase_1_Prio, Client_ID);
   -- set timer
   Switch_Timer.Set_Handler(End_of_Phase_1, Handler'Access);
   -- release
   Released := True;
end Release_Task;
```

Implementing split-task multiprocessor scheduling: job-based split-task dispatching #2

- The second algorithm is also job-based split-task
- However, it uses EDF for scheduling tasks, and
- Migration is in dependent on actual execution time
  - So a executon time timer is used

τ_2 is a split task. When a job of τ_2 arrives, it executes on processor 1 and then it migrates to processor 2.



Implementing split-task multiprocessor scheduling: job-based split-task dispatching #2

```
protected body My_Job_Based_Switcher is
  procedure Register(ID : Task_ID; Phase_2_CPU: CPU_Range)  …

  procedure Budget_Expired(T : in out Ada.Execution_Time.Timers.Timer) is
  begin
    -- similarly to previous section, handler just called in the end of phase 1
    Set_CPU(Client_Phase_2_CPU, Client_ID);
  end Budget_Expired;

end My_Job_Based_Switcher;
```

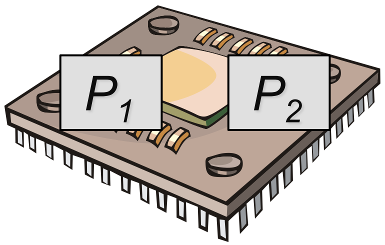
Implementing split-task multiprocessor scheduling: job-based split-task dispatching #2

```
task body Task_2 is
   …
begin
   My_Job_Based_Switcher.Register( ... );
   Next := Ada.Real_Time.Clock;
   loop
      Delay_Until_and_Set_Deadline( Next, Deadline_Task_2);
      Set_CPU(Phase_1_CPU, My_ID);
      Ada.Execution_Time.Timers.Set_Handler(The_Timer, C_First_Phase,
                        My_Job_Based_Switcher.Budget_Expired'Access);
      -- Code of application
      Ada.Execution_Time.Timers.Cancel_Handler(The_Timer, Cancelled);
      Next := Next + Period_Task_2;
   end loop;
end Task_2;
```

Discussion

Attempts to transition RM and EDF to multicores.

Development of
multicore scheduling using
the task-splitting
class of algorithms.



Question: Can the new language constructs for supporting multicore real-time scheduling be used to implement previously published multicore scheduling algorithms based on task-splitting?

2000

A

time

Conclusion

The new language constructs for supporting multicore real-time scheduling **can** be used to implement previously published multicore scheduling algorithms based on task-splitting.

Question: Can the new language constructs for supporting multicore real-time scheduling be used to implement previously published multicore scheduling algorithms based on task-splitting?

2000

A

time

Conclusion

The new language constructs for supporting multicore real-time scheduling **can** be used to implement previously published multicore scheduling algorithms based on task-splitting.

Open Question

Do the new language constructs for supporting multicore real-time scheduling **allow efficient/strict** implementations of previously published multicore scheduling algorithms based on task-splitting?

Question: Can the new language constructs for supporting multicore real-time scheduling be used to implement previously published multicore scheduling algorithms based on task-splitting?

Discussion

- There are a few practical imperfections
- Code executing in the wrong processor
  - Handlers and release procedures
  - Should we specify in which CPU timing event and execution time handlers execute?
    - Setting in a different CPU may need to reschedule so we need more experience with implementations
- In particular, a potential source of priority/deadline inversion
  - Task 2 in the last example
  - Also, periodic tasks in slot-based approaches must be via a timer
- Should we defer changing CPU and Deadline?
  - Instead a lot of Delay_Until_And_Set_X_And_Y_And_Z (and do not forget Yield_And_Set_Deadline?)

2000

A

time

Conclusion

The new language constructs for supporting multicore real-time scheduling **can** be used to implement previously published multicore scheduling algorithms based on task-splitting.

Open Question

Do the new language constructs for supporting multicore real-time scheduling **allow efficient/strict** implementations of previously published multicore scheduling algorithms based on task-splitting?

Question: Can the new language constructs for supporting multicore real-time scheduling be used to implement previously published multicore scheduling algorithms based on task-splitting?

Thank You

Questions?