TDDD07 – Real-time Systems

Lecture 3: Scheduling and Resource sharing

Simin Nadjm-Tehrani

Real-time Systems Laboratory

Department of Computer and information Science

LINKÖPING UNIVERSITY

Recap from last lecture

- Cyclic scheduling
  - Offline
  - No priorities
- Rate Monotonic Scheduling
  - Online
  - (Fixed) priorities based on periods

Dynamic priorities

- Next: We look at regimes that change priorities dynamically

Priority-based scheduling

Earliest Deadline First

Earliest deadline first (EDF)

- Online decision
- Preemptive
- Dynamic priorities

Policy: Always run the process that is *closest* to its deadline

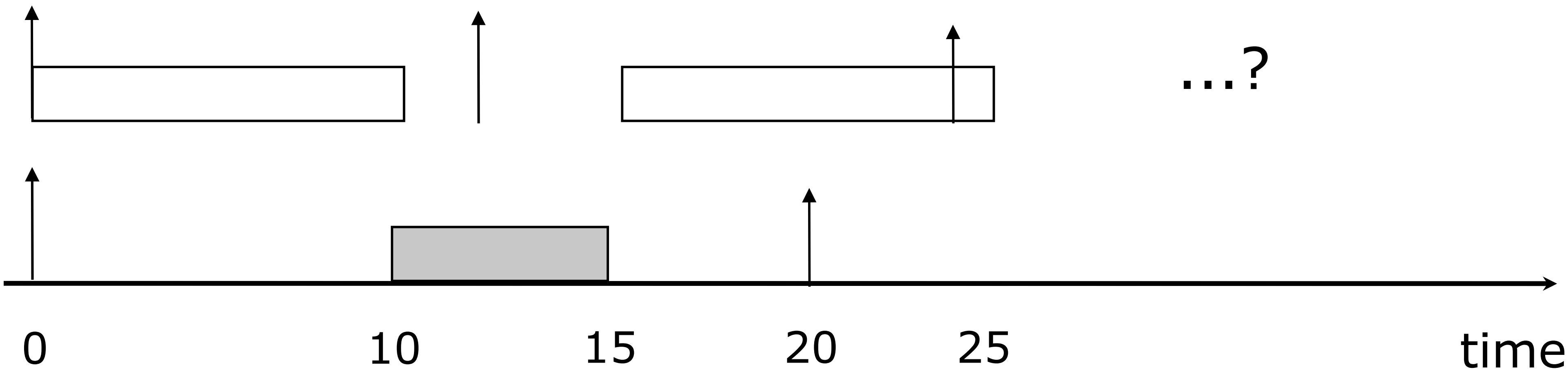
Assumptions on process set

- Event that leads to release of process P_i appears with minimum inter-arrival interval T_i
- Each P_i has a max computation time C_i
- The process must be finished before its relative deadline $D_i \leq T_i$
- Processes are independent (do not share resources other than CPU)

- **EDF:** The process with nearest absolute deadline (d_i) will run first

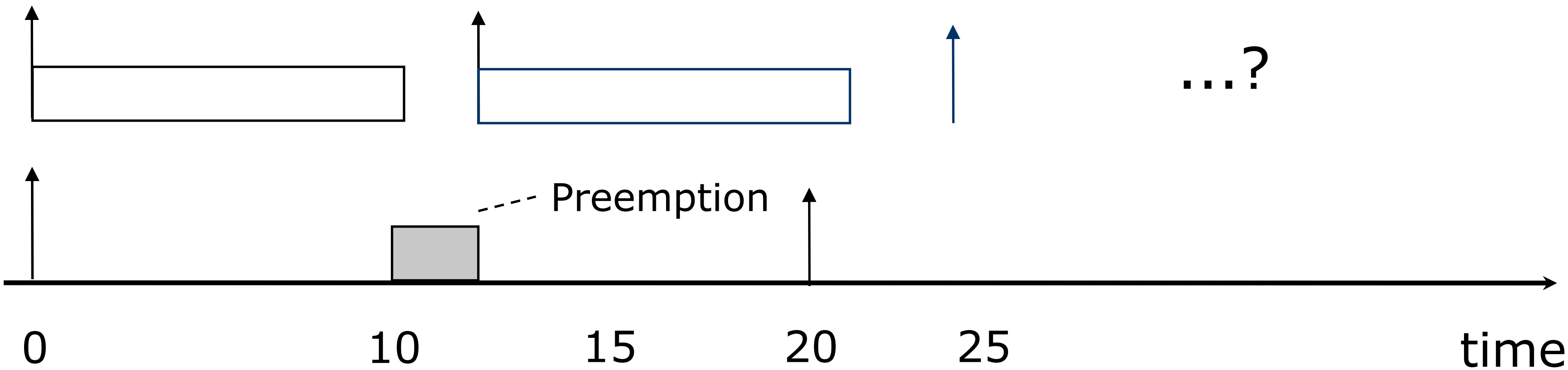
Example (6)

Consider following processes:	P_1	P_2
WCET (C_i)	5	10
Deadline ($D_i = T_i$)	20	12
Arrival times (r_i)	0, 20,...	0, 12,...



Compare to RMS

For the same task set:	P_1	P_2
WCET (C_i)	5	10
Deadline ($D_i = T_i$)	20	12
Arrival times (r_i)	0, 20,…	0, 12,…



Theorem

A set of *periodic* tasks $P_1,\ldots,P_n$ for which $D_i = T_i$ is schedulable with EDF **iff**

$$U= C_1/T_1+\ldots+C_n/T_n \leq 1$$

For Example 6:

$$U=C_1/T_1 + C_2/T_2 = 5/20 + 10/12 = 1.08!$$

Example (7)

Consider following task set:	P_1	P_2
WCET (C_i)	2	4
Deadline ($D_i = T_i$)	5	7

Is it schedulable?

$$U = 2/5 + 4/7 = 0.97$$

Yes!

EDF vs. RMS

- EDF gives higher processor utilisation (Example 7 not schedulable with RMS!)
- EDF has simpler exact analysis for the mentioned type of task sets
- But suffers from domino effect...
- RMS can be implemented to run faster at run-time (if we ignore the time for context switching)

2 Bonus points!

[Deeper analysis of RMS and EDF based on Buttazzo 2005 article!] **$B**

Next…

- We remove the assumption that all tasks are independent!

Resource sharing

Sharing resources other than CPU

- Assume that processes synchronise using semaphores
- We schedule the processes with *fixed* priorities but relax the independence requirement

Priority Inversion

- A low priority process (P_1) locks the resource
- A high priority process (P_2) has to wait on the semaphore (blocked state)
- A medium priority process (P_3) preempts P_1 and runs to completion before P_2!

How to avoid it?

- When P_2 is blocked by P_1 one raises the priority of P_1 to the same level as P_2 temporarily
- Afterwards, when the semaphore is released by P_1, it goes back to its prior priority level
- P_3 can not interrupt P_1 any more!

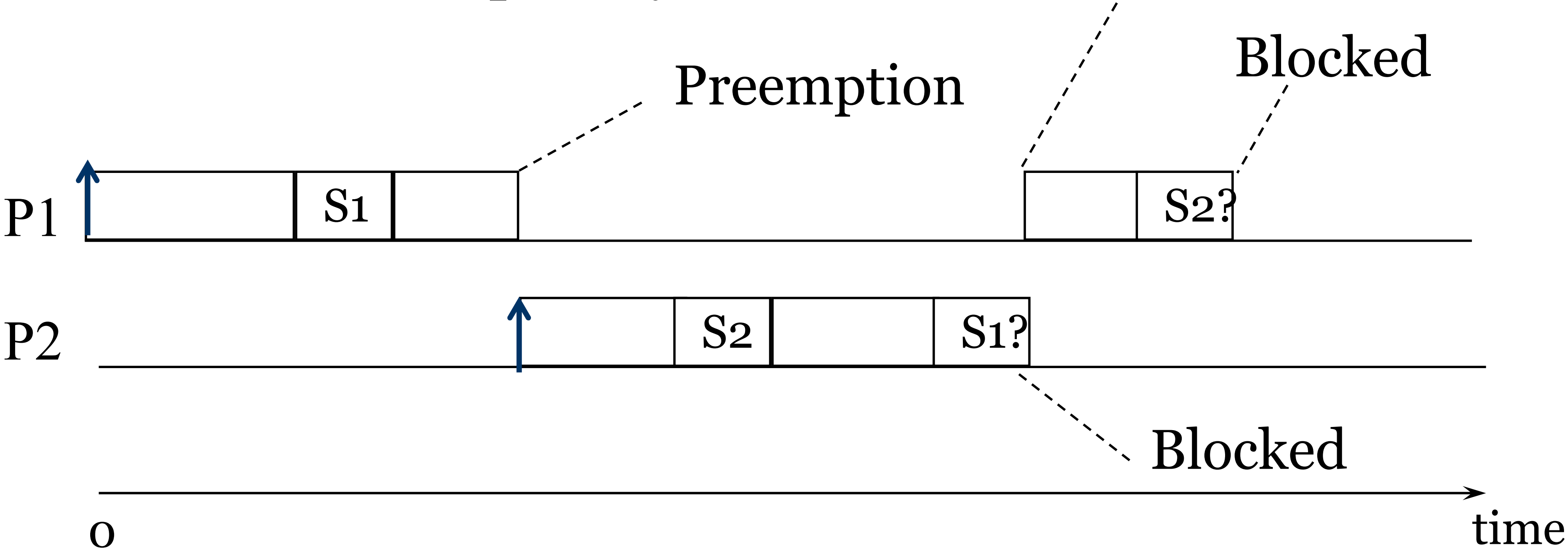
Priority inheritance

- Is transitive
- Can compute maximum blocking time for each resource (high priority process P_2 is blocked only under the time that P_1 uses the resource)
- As long as the resource is released!

- But … it does not avoid deadlock!

Example (8)

Let P1 have lower priority than P2.



Here $\boxed{\text{Si}}$ denotes the process locks semaphore Si.

Terminology

Note that:

- *blocked* – when waiting due to a resource (other than CPU)
- *not dispatched* or *preempted* - when waiting for CPU

Ceiling Protocols

e.g. Immediate priority Ceiling Protocol (ICP):

- A process that obtains a resource inherits the *resource's ceiling priority* - the highest priority among all processes that can possibly claim that resource
- Dynamic priority for a process is the max of own (fixed) priority and the ceiling values of all resources it has locked
- When a resource is released, the process priority returns to the normal level (or to another engaged resource's ceiling)

ICP and deadlocks/starvation

LINKÖPING
UNIVERSITY

Properties

- The blocking delay for process Pi is a function of the length of all critical sections
  – We need to compute this (Bi) for each process!
- Do not even need to use semaphores!
- A process is blocked max once by another process with lower priority

Let’s prove that!

ICP & Deadlock-related issues

- The ICP prevents deadlocks (How?)
- ICP prevents starvation (How?)

Recall: Coffman conditions

1. **Mutual exclusion**
   Access to resource is limited to one (or a limited number of) process(es) at a time

2. **Hold & wait**
   Processes hold allocated resources and wait for another resource at the same time

Coffman conditions

3. Voluntary release

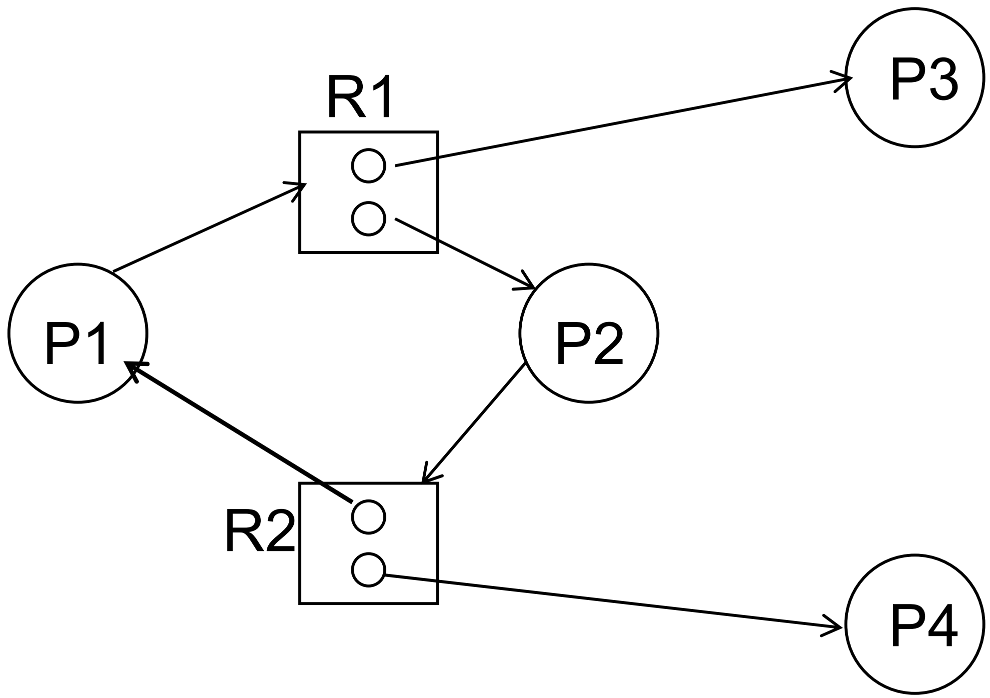
Resources can only be released by a process voluntarily

4. Circular wait

There is a chain of processes where each process holds a resource that is required by another process

Recall: Resource allocation graphs

Recall from the OS course: A dynamic snapshot of which resources are allocated, and which resources are wished



ICP & Deadlock

- The ICP prevents deadlocks (How?)
- We need to show that a set of n processes using FP scheduling and ICP cannot end up in a deadlock
- Use proof by contradiction!

ICP & Starvation

- Show that an arbitrary process that is waiting will not wait for a resource indefinitely
- First, recall that it will not wait for a chain of waiting processes indefinitely
- Second, show that waiting for a running process is bounded by the combined impact of interference and blocking, which can be computed
- A process that waits indefinitely will only do so if its response time is beyond its deadline

Questions?

www.ida.liu.se/~TDDD07