Introduction to Real-Time Systems

Note: Slides are adopted from Lui Sha and Marco Caccamo

Overview

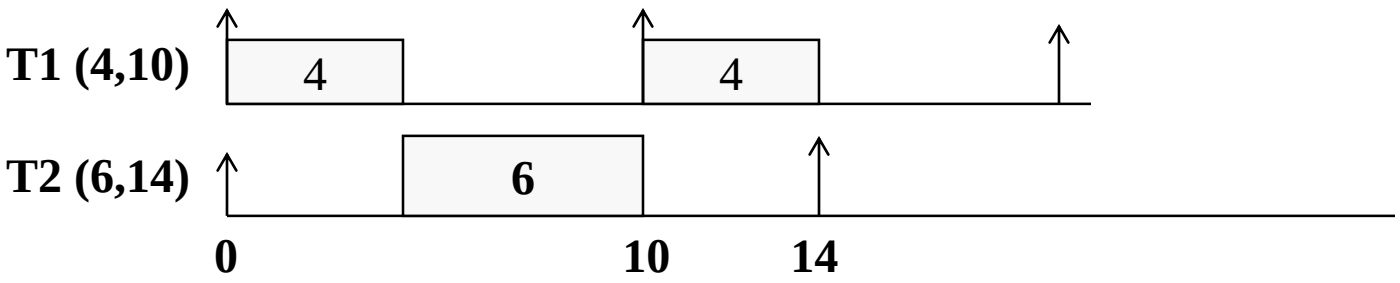
- Today: this lecture explains how to use Utilization Bound, it introduces the POSIX.4 scheduling interface and the exact analysis

- **To learn more on real-time scheduling**:
  - **see chapter 4 on "Hard Real-Time Computing Systems" book from G. Buttazzo**

- **To learn more on POSIX.4 scheduling interface**:
- Book: Programming for The Real World, Bill O. Gallmeister, O'Reilly&Associates, Inc.
  See pp.159-171 and 200-207 (available in the Lab)

- Basic tutorial at http://www.netrino.com/Publications/Glossary/RMA.html

RMS: Less Than 100% Utilization but not Schedulable



- **In this example, 2 tasks are scheduled under RMS, an optimal static priority method**
  4/10 + 6/14 = 0.83

- **The task set is schedulable but if we try to increase the computation time of task T1, the task set becomes unschedulable in spite of the fact that total utilization is 83%!**

- **<u>To achieve 100% utilization when using fixed priorities</u>, assign periods so that all tasks are harmonic. This means that for each task, its period is an exact multiple of every other task that has a shorter period.**

- **For example, a three-task set whose periods are 10, 20, and 40, respectively, is harmonic, and preferred over a task set with periods 10, 20, and 50**

The Liu & Layland Bound

C. Liu, J. Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment," JACM, 1973

- A set of n periodic tasks is schedulable if:

$$\frac{c_1}{p_1} + \frac{c_2}{p_2} + \ldots + \frac{c_n}{p_n} \leq n\left(2^{1/n} - 1\right)$$

- U(1) = 1.0 U(4) = 0.756 U(7) = 0.728
- U(2) = 0.828 U(5) = 0.743 U(8) = 0.724
- U(3) = 0.779 U(6) = 0.734 U(9) = 0.720

- For harmonic task sets, the utilization bound is U(n)=1.00 for all n. Otherwise, for large n, the bound converges to *ln 2* ~ 0.69.

- The L&L bound for rate monotonic algorithm is one of the most significant results in real-time scheduling theory. It allows to check the schedulability of a group of tasks with a single test! It is a sufficient condition; hence, it is inconclusive if it fails!

Sample Problem: Applying UB Test

	C	P	U
Task $\tau 1$:	**20**	**100**	**0.200**
Task $\tau 2$:	**40**	**150**	**0.267**
Task $\tau 3$:	**100**	**350**	**0.286**

- Are all the tasks schedulable?
- What if we double the execution time of task $\tau 1$?

Sample Problem: Applying UB Test

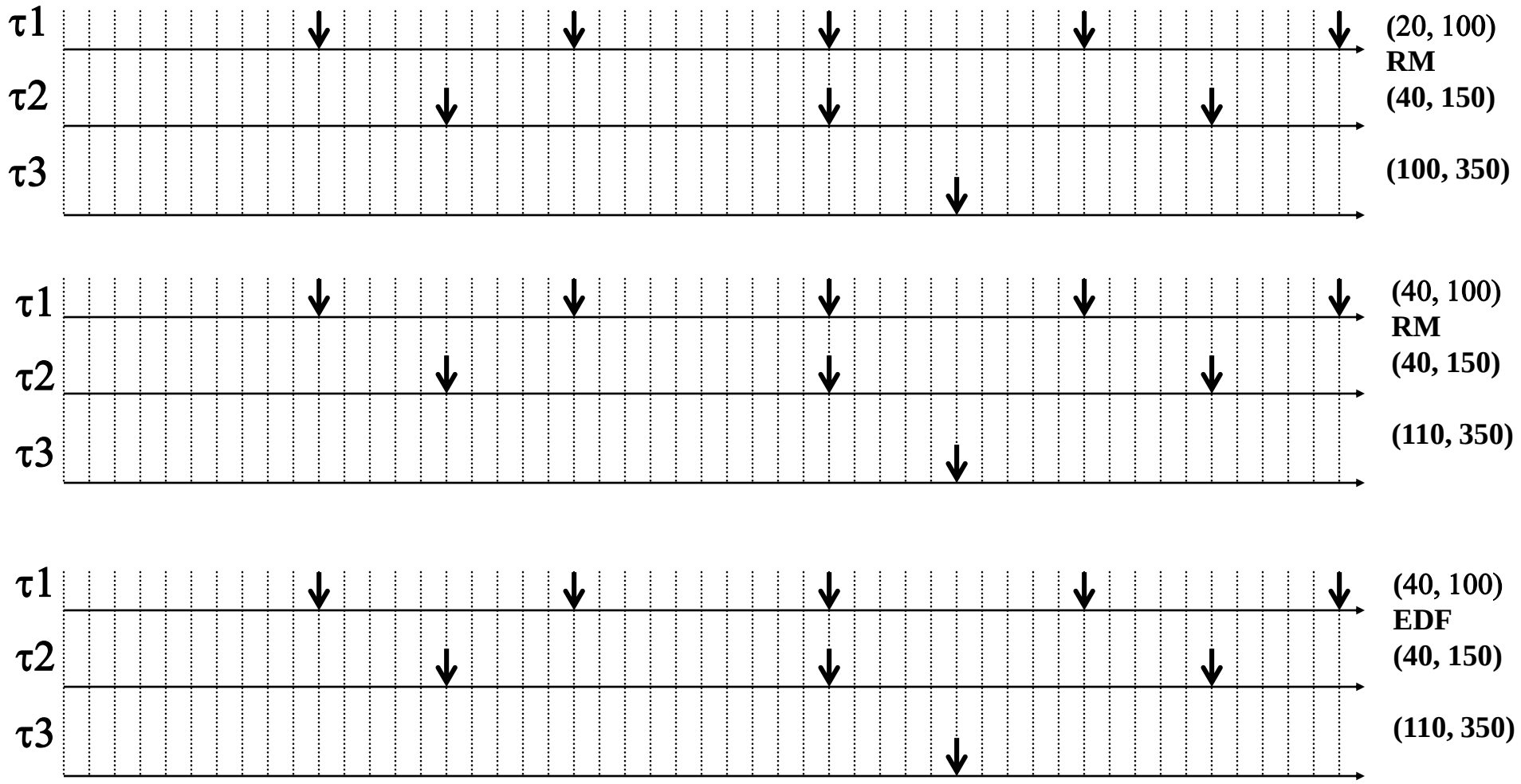
	C	P	U
Task $\tau 1$:	**20**	**100**	**0.200**
Task $\tau 2$:	**40**	**150**	**0.267**
Task $\tau 3$:	**100**	**350**	**0.286**

- Are all the tasks schedulable?
- Check the schedulability of T1, T2, and T3: $U_1 + U_2 + U_3 = 0.753 < U(3)$ ➔ **Schedulable!**

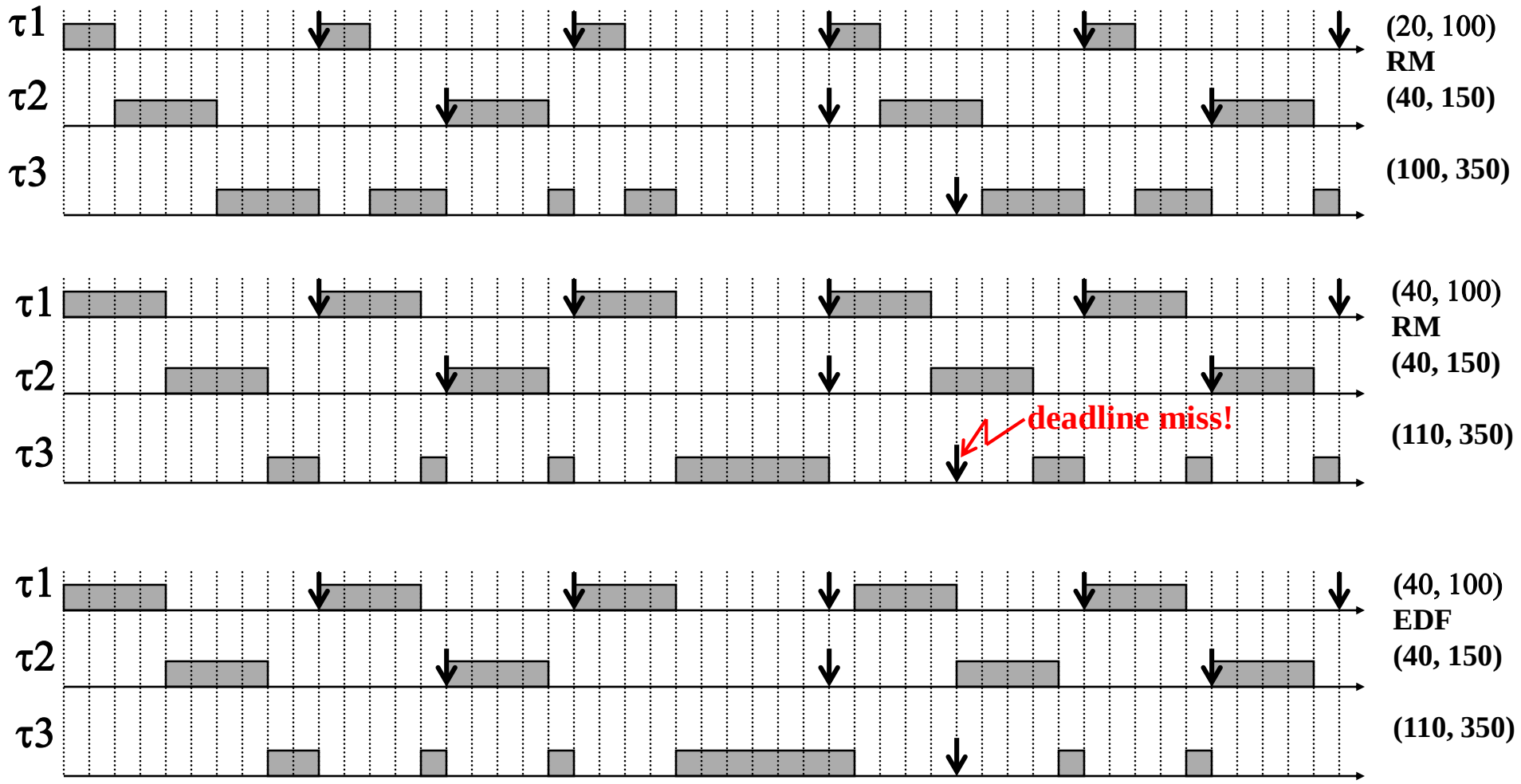
- What if we double the execution time of task $\tau 1$?
- Check schedulability of T1 and T2: $\frac{40}{100} + \frac{40}{150} = 0.4 + 0.27 = 0.67 < U(2)$ ➔ **Schedulable!**
- Check schedulability of T1, T2 and T3: $\frac{40}{100} + \frac{40}{150} + \frac{100}{350} = 0.953 > U(3) = 0.779$

- UB test is a sufficient condition and thus inconclusive if it fails!

Sample Problem: draw the schedule by using RM and EDF



Sample Problem: draw the schedule by using RM and EDF



Posix.4 scheduling interfaces

- The real-time scheduling interface offered by POSIX.4 (available on Linux kernel)

- Each process can run with a particular scheduling policy and associated scheduling attributes. Both the policy and the attributes can be changed independently. POSIX.4 defines three policies:

- SCHED_FIFO: preemptive, priority-based scheduling.
- SCHED_RR: Preemptive, priority-based scheduling with quanta.
- SCHED_OTHER: an implementation-defined scheduler

Posix.4 scheduling interfaces

- SCHED_FIFO: preemptive, priority-based scheduling.
- The available priority range can be identified by calling:
  sched_get_priority_min(SCHED_FIFO) ➔ Linux 2.6 kernel: 1
  sched_get_priority_max(SCHED_FIFO); ➔ Linux 2.6 kernel: 99

- *SCHED_FIFO* can only be used with static priorities higher than 0, which means that when a *SCHED_FIFO* process becomes runnable, it will always preempt immediately any currently running normal *SCHED_OTHER* process. *SCHED_FIFO* is a simple scheduling algorithm without time slicing.

- A process calling **sched_yield** will be put at the end of its priority list. No other events will move a process scheduled under the *SCHED_FIFO* policy in the wait list of runnable processes with equal static priority. A *SCHED_FIFO* process runs until either it is blocked by an I/O request, it is preempted by a higher priority process, it calls **sched_yield,** or it finishes.

Posix.4 scheduling interfaces

- SCHED_RR: preemptive, priority-based scheduling with quanta.
- The available priority range can be identified by calling:
  sched_get_priority_min(SCHED_RR) ➔ Linux 2.6 kernel: 1
  sched_get_priority_max(SCHED_RR); ➔ Linux 2.6 kernel: 99

- *SCHED_RR* is a simple enhancement of *SCHED_FIFO*. Everything described above for *SCHED_FIFO* also applies to *SCHED_RR*, except that each process is only allowed to run for a maximum time quantum. If a *SCHED_RR* process has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority.

- A *SCHED_RR* process that has been preempted by a higher priority process and subsequently resumes execution as a running process will complete the unexpired portion of its round robin time quantum. The length of the time quantum can be retrieved by **sched_rr_get_interval**.

Posix.4 scheduling interfaces

- SCHED_OTHER: an implementation-defined scheduler

- **Default Linux time-sharing scheduler**
- *SCHED_OTHER* can only be used at static priority 0. *SCHED_OTHER* is the standard Linux time-sharing scheduler that is intended for all processes that do not require special static priority real-time mechanisms. The process to run is chosen from the static priority 0 list based on a dynamic priority that is determined only inside this list.

- The dynamic priority is based on the nice level (set by the **nice** or **setpriority** system call) and increased for each time quantum the process is ready to run, but denied to run by the scheduler. This ensures fair progress among all *SCHED_OTHER* processes.

Posix.4 scheduling interfaces

- Child processes inherit the scheduling algorithm and parameters across a **fork**.

- Memory locking is usually needed for real-time processes to avoid paging delays, this can be done with **mlock** or **mlockall**.

- **Do not forget!!!!**

  ➔ a non-blocking end-less loop in a process scheduled under *SCHED_FIFO* or *SCHED_RR* will block all processes with lower priority forever, a software developer should always keep available on the console a shell scheduled under a higher static priority than the tested application. This will allow an emergency kill of tested real-time applications that do not block or terminate as expected.

- Since *SCHED_FIFO* and *SCHED_RR* processes can preempt other processes forever, only root processes are allowed to activate these policies under Linux.

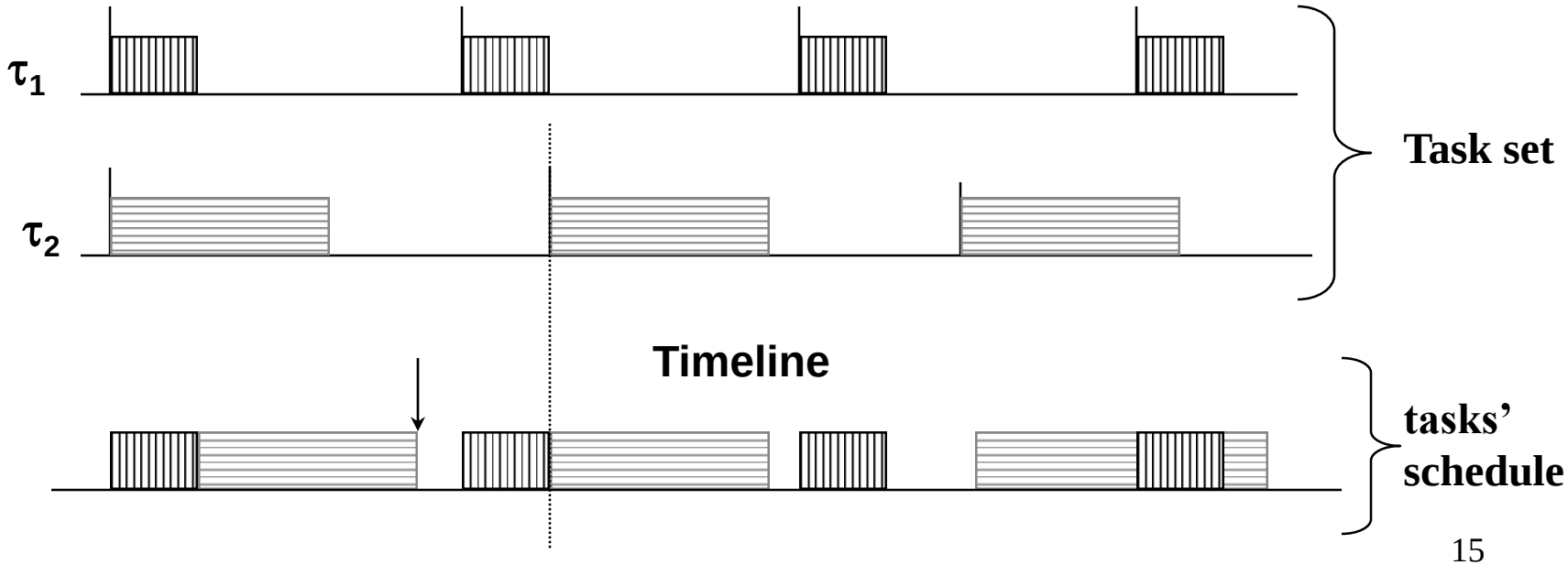
Posix.4 scheduling interfaces

```
#include <sched.h>
#include <sys/types.h>
#include <stdio.h>

int             fifo_min, fifo_max;
int             sched, prio, i;
pid_t           pid;
struct sched_param  attr;

main()
{
        fifo_min = sched_get_priority_min(SCHED_FIFO); fifo_max = sched_get_priority_max(SCHED_FIFO);

        printf("\n Scheduling informations: input a PID?\n");
        scanf("%d", &pid);
        sched_getparam(pid, &attr);
        printf("process %d uses scheduler %d with priority %d \n", pid,
        sched_getscheduler(pid), attr.sched_priority);

        printf("\n Let's modify a process sched parameters: Input the PID, scheduler type, and priority \n");
        scanf("%d %d %d", &pid, &sched, &prio);

        attr.sched_priority = prio;
        i = sched_setscheduler(pid, sched, &attr);
}
```

The Exact Schedulability Test

Critical instant theorem: If a task meets its first deadline when all higher priority tasks are started at the same time, then this task's future deadlines will always be met. The exact test for a task checks if this task can meet its first deadline[Liu73].

It holds only for fixed priority scheduling!



Exact Schedulability Test (Exact Analysis)

$$r_i^{k+1} = c_i + \sum_{j=1}^{i-1} \left\lceil \frac{r_i^k}{p_j} \right\rceil c_j, \quad \text{where } r_i^0 = \sum_{j=1}^{i} c_j$$

Test terminates when $r_i^{k+1} > p_i$ (not schedulable)
or when $r_i^{k+1} = r_i^k \leq p_i$ (schedulable).

Tasks are ordered according to their priority: T_1 is the highest priority task.

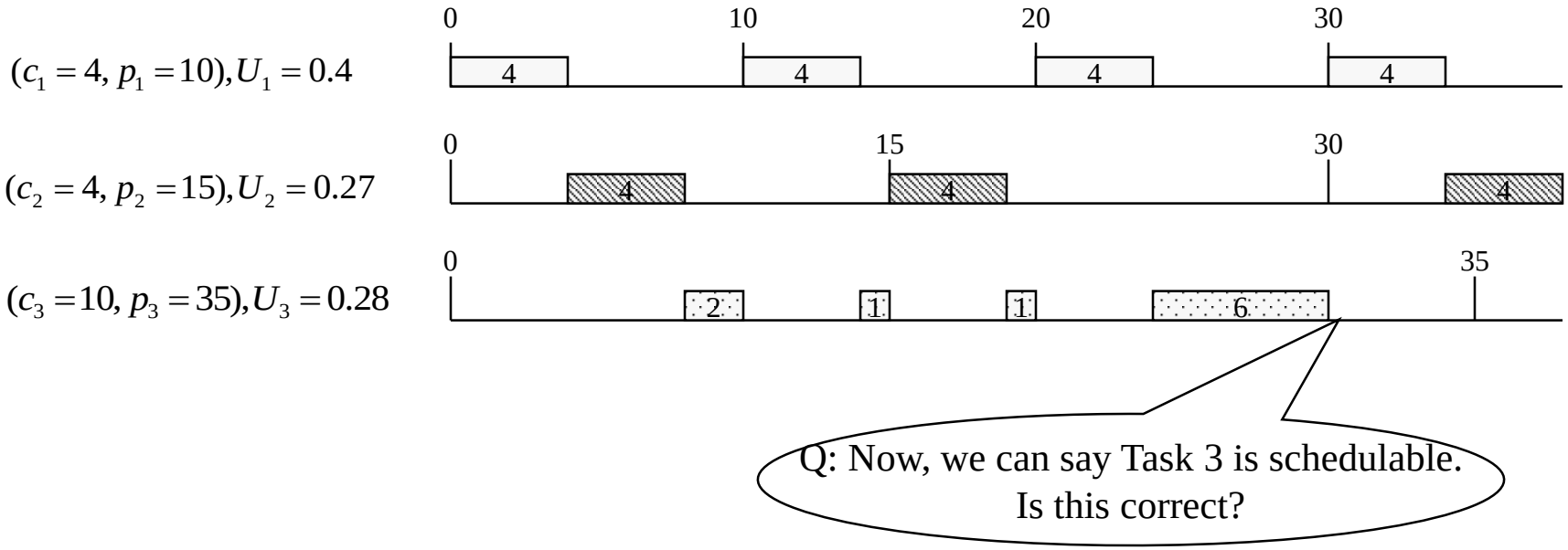
The superscript *k* indicates the number of iterations in the calculation.
The index *i* indicates it is the *ith* task being checked.

The index *j* runs from 1 to i-1, i.e. all the higher priority tasks. Recall from the convention - task 1 has a higher priority than task 2 and so on.

<u>We check the schedulability of a single task at the time!!!</u>

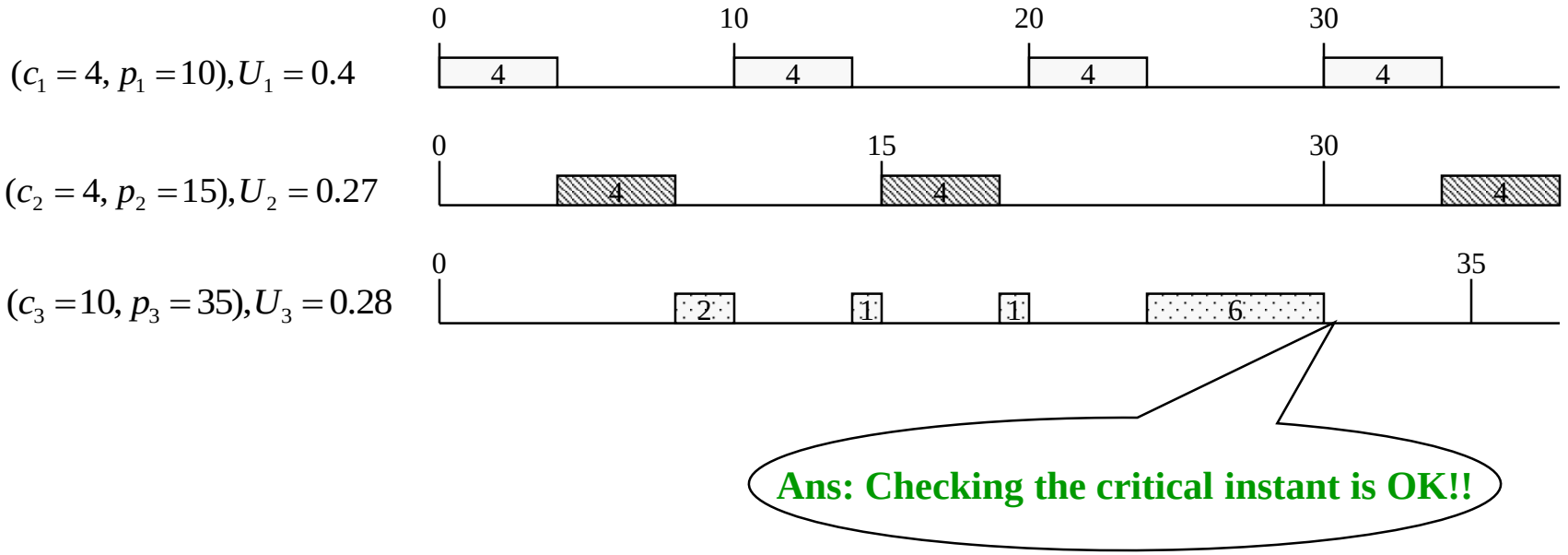
The Exact Schedulability Test

- Basically, "*Enumerate"* the schedule
- "Task by Task" schedulability test



How long should we enumerate the schedule?

Critical instant theorem: If a task meets its first deadline when all higher priority tasks are started at the same time, then this task's future deadlines will always be met. The exact test for a task checks if this task can meet its first deadline[Liu73].

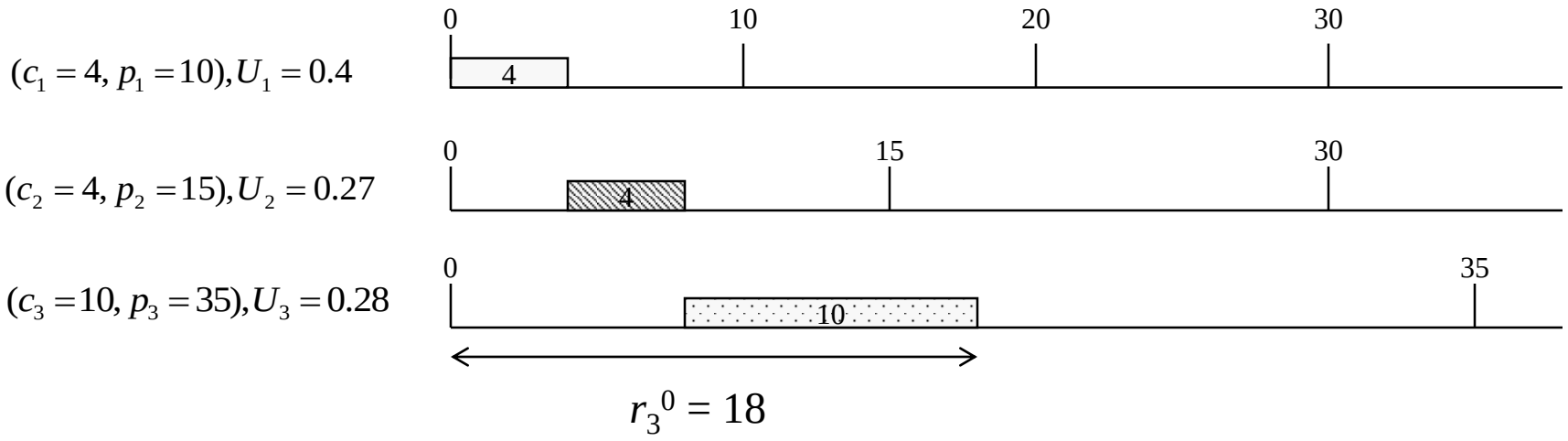


Intuitions of Exact Schedulability Test

- Obviously, the response time of task 3 should be larger than or equal to $c_1+c_2+c_3$

$$r_3^0 = \sum_{j=1}^{3} c_j = c_1 + c_2 + c_3 = 4 + 4 + 10 = 18$$

Intuitions of Exact Schedulability Test



Intuitions of Exact Schedulability Test

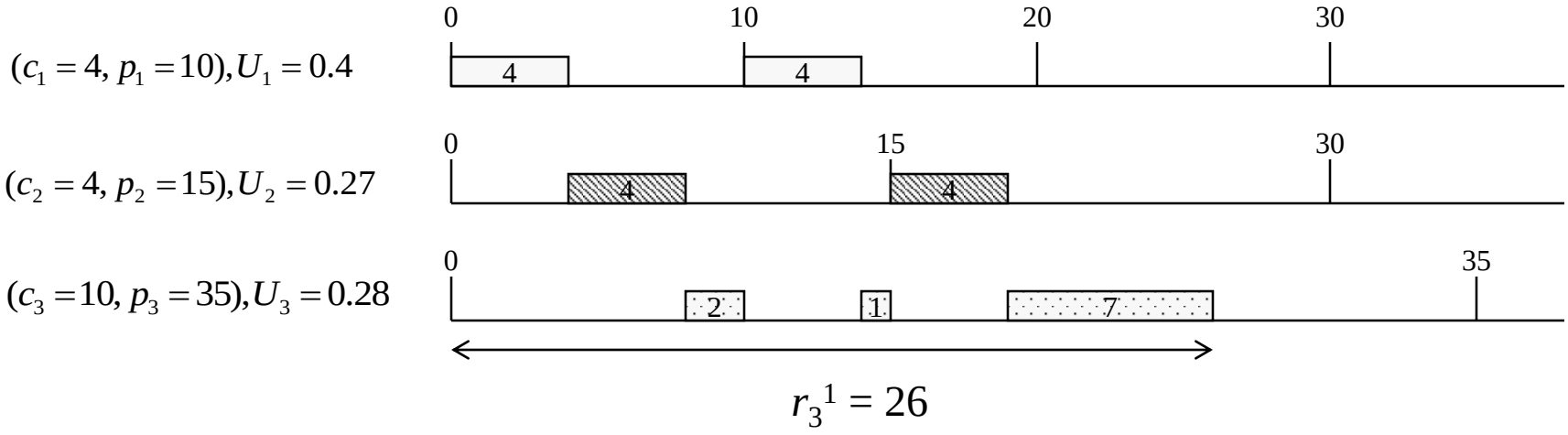
- Obviously, the response time of task 3 should be larger than or equal to $c_1+c_2+c_3$

$$r_3^0 = \sum_{j=1}^{3} c_j = c_1 + c_2 + c_3 = 4 + 4 + 10 = 18$$

- The high priority jobs released before r_3^0, should lengthen the response time of task 3

$$r_3^1 = c_3 + \sum_{j=1}^{2} \left\lceil \frac{r_3^0}{p_j} \right\rceil \cdot c_j = 10 + \left\lceil \frac{18}{10} \right\rceil 4 + \left\lceil \frac{18}{15} \right\rceil 4 = 26$$

Intuitions of Exact Schedulability Test



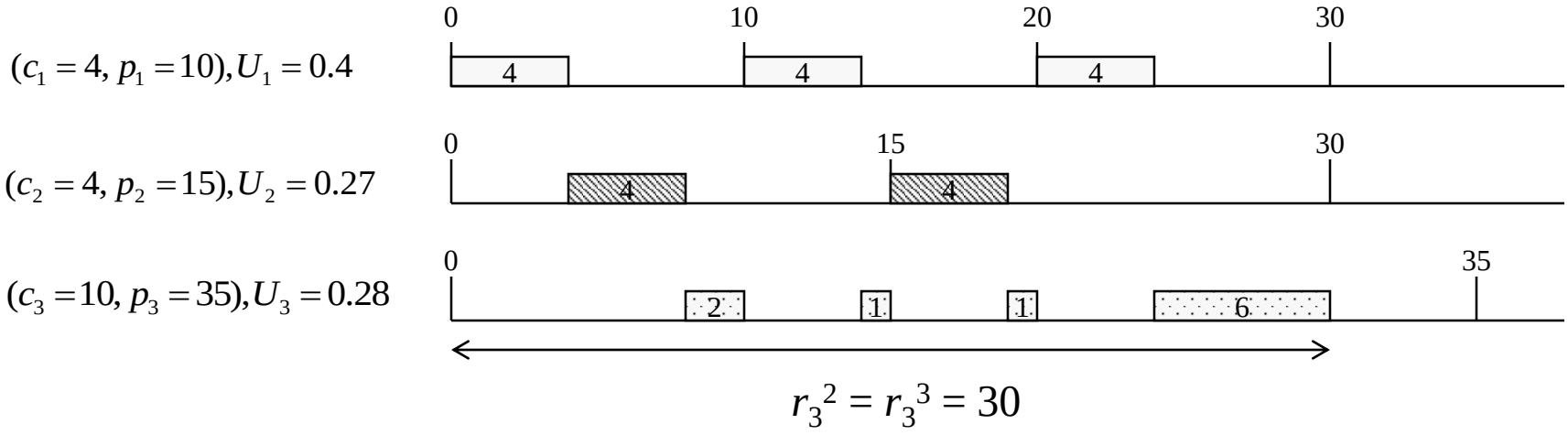
Intuitions of Exact Schedulability Test

- Keep doing this until either r_3^k no longer increases or $r_3^k > p_3$

$$r_3^2 = c_3 + \sum_{j=1}^{2} \left\lceil \frac{r_3^1}{p_j} \right\rceil \cdot c_j = 10 + \left\lceil \frac{26}{10} \right\rceil 4 + \left\lceil \frac{26}{15} \right\rceil 4 = 30$$

$$r_3^3 = c_3 + \sum_{j=1}^{2} \left\lceil \frac{r_3^2}{p_j} \right\rceil \cdot c_j = 10 + \left\lceil \frac{30}{10} \right\rceil 4 + \left\lceil \frac{30}{15} \right\rceil 4 = 30$$ **Done!**

Intuitions of Exact Schedulability Test



Intuition for the Exact Schedulability Test

- Suppose we have n tasks, and we pick a task, say i, to see if it is schedulable.
- We initialize the testing by assuming all the higher priority tasks from 1 to i-1 will only preempt task i once.
- Hence, the initially presumed finishing time for task i is just the sum of C_1 to C_i, which we call r^0.
- We now check the actual arrival of higher priority tasks within the duration r^0 and then presume that it will be all the preemption task i will experience. So we compute r^1 under this assumption.
- We will repeat this process until one of the two conditions occur:
  - 1. The r^n eventually exceeds the deadline of task i. In this case we terminate the iteration process and conclude that task i is not schedulable.
  - 2. The series r^n converges to a fixed point (i.e., it stops increasing). If this fixed point is less than or equal to the deadline, then the task is schedulable and we terminate the schedulability test.

Assumptions under UB & Exact Analysis

- Both the Utilization Bound and the Exact schedulability test make the following assumptions:

  - All the tasks are periodic
  - Tasks are scheduled according to RMS
  - All tasks are independent and do not share resources (data)
  - Tasks do not self-suspend during their execution
  - Scheduler overhead (context-switch) is negligible