EECS 388: Embedded Systems

8. Real-Time Scheduling

Heechul Yun

Agenda

- Real-time operating systems
- Real-time CPU scheduling theory and practice

Real-Time Operating System

- Often refers to lightweight OS used in embedded systems
  - FreeRTOS, VxWorks, QNX, …
- Specialized to guarantee fast, deterministic real-time response to external events
  - Real-time (CPU) scheduling is key

Real-Time Operating System

- Thread scheduling
  - CPU scheduler
- Synchronization
  - Lock, semaphore, etc.
- Input and output
  - Device drivers
- …

A Scheduler

- Initialization
  - set up periodic timer interrupts;
  - create default thread data structures;
  - dispatch a thread (procedure call);
  - execute main thread (idle or power save, for example).
- Thread control block (TCB) data structure
  - copy of CPU state (machine registers)
  - address at which to resume executing the thread
  - status of the thread (e.g. blocked on mutex)
  - priority, WCET (worst case execution time), and other info to assist the scheduler

A Scheduler

- Timer interrupt service routine:
  - dispatch a thread.
- Dispatching a thread:
  - disable interrupts;
  - **determine which thread should execute (scheduling);**
  - if the same one, enable interrupts and return;
  - save state (registers) into current thread data structure;
  - save return address from the stack for current thread;
  - copy new thread state into machine registers;
  - replace program counter on the stack for the new thread;
  - enable interrupts;
  - return.

Real-Time Systems

- The correctness of the system depends on not only on the logical result of the computation but also on the time at which the results are produced
- **A correct value at a wrong time is a fault.**

- Two requirements
  - Logical correctness: correct outputs
  - Temporal correctness: outputs at the right time

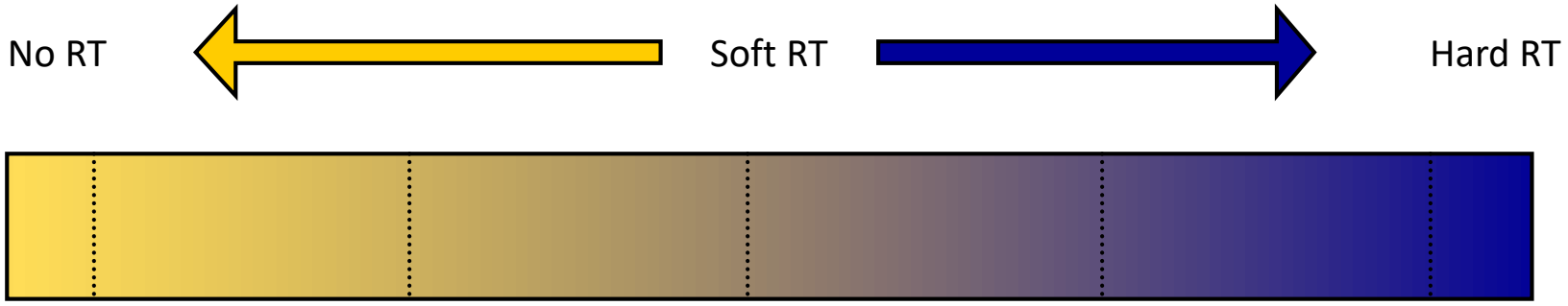
KU THE UNIVERSITY OF KANSAS

Soft vs. Hard Real-Time

- Soft real-time: missing deadlines is undesirable, but will not lead to catastrophic consequences
  - Related to the concept of "Quality of Service"
  - Typically interested in average-case response time (turnaround time)
  - Ex: reservation systems, media players, phones, etc.

- Hard real-time: missing deadlines is not an option
  - Interested in worst-case response time
  - Ex: airplanes, nuclear plants, military systems, etc.

Real-Time Spectrum



Jobs and Tasks

- A job is a unit of computation
  - E.g., one execution of a key event handling
- A task is a sequence jobs of the same type
  - E.g., the key event handling task

Periodic Tasks

- Time-triggered computation
- Task is activated periodically every T time units.
- Each instance of the task is called a job.
- Each job has the same relative deadline (usually = to period).
- E.g., most digital controllers.

Periodic Task Model

- Task τ_i (N tasks in the system, τ_1 to τ_N)
  - Execution time C_i (sometimes e_i)
  - Relative deadline D_i
  - Period T_i (sometimes p_i)

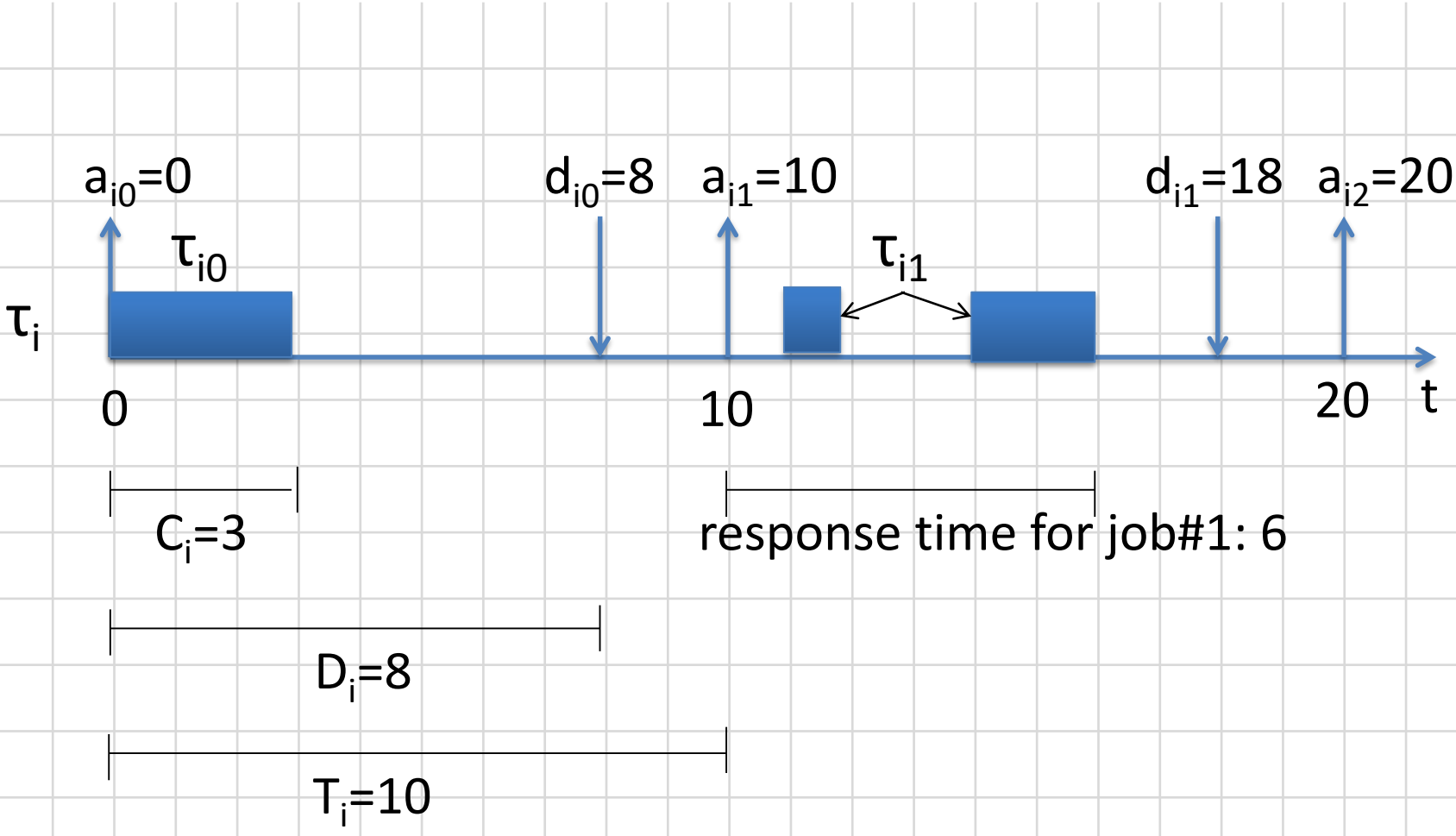
- Each job τ_{ij} of τ_i (first job: τ_{i0})
  - Activation time $a_{ij} = a_{ij-1} + T_i$ (usually with $a_{i0} = 0$)
  - Absolute deadline $d_{ij} = a_{ij} + D_i$

Periodic Task Model

- Release time: the instant at which the job becomes ready to execute
- Absolute deadline: specified in absolute time.
  - Ex: train and airlines schedules
- Relative deadline: related to the release time.
  - Ex: 8 msec after the release time.
- By convention, we will refer to an absolute deadline as “d”, and a relative deadline as “D”

KU THE UNIVERSITY OF KANSAS

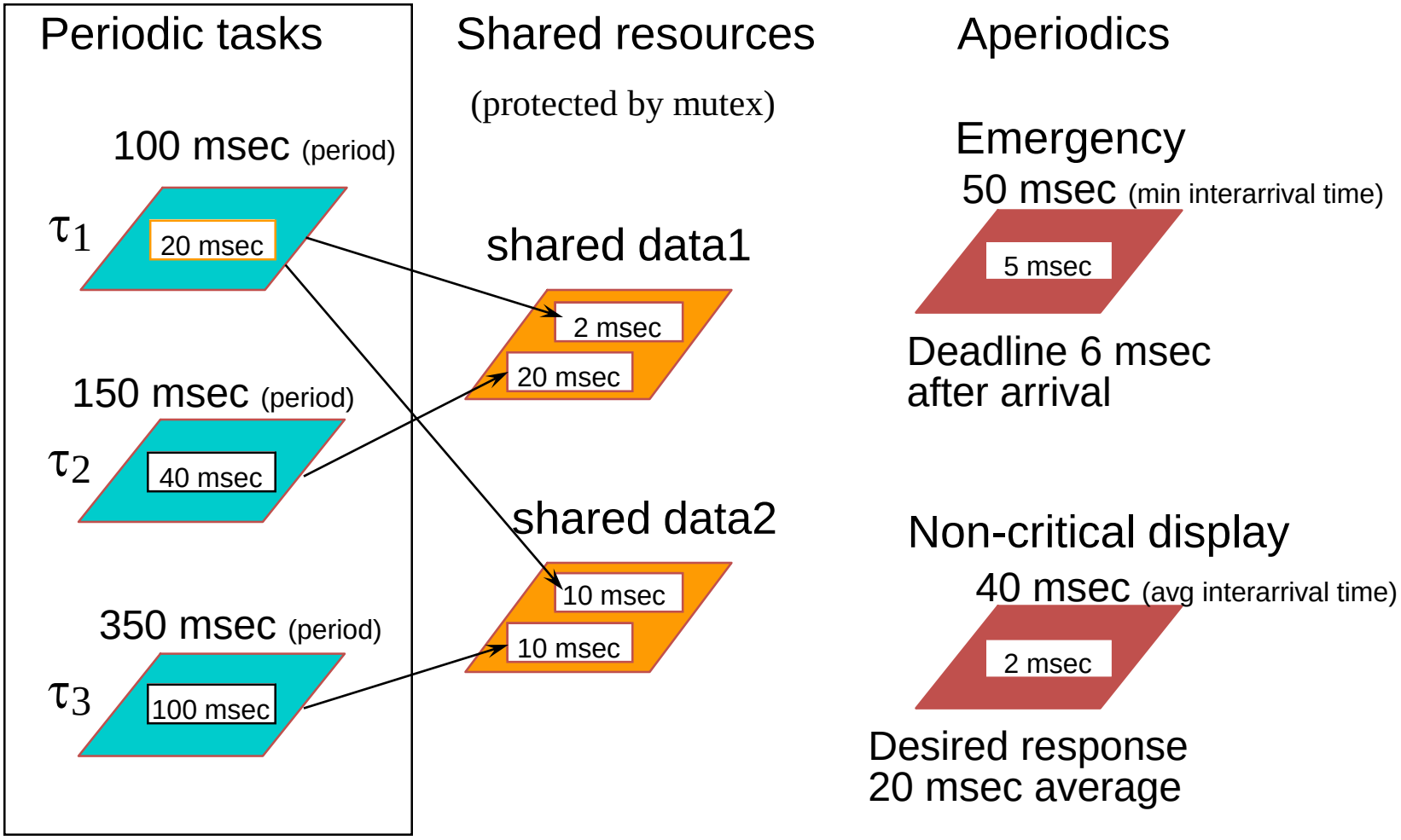
Periodic Task Model



Aperiodic Tasks

- Event-triggered computation.
- Task is activated by an external event.
- Task runs once to respond to the event.
- Relative deadline D: available time to respond to the event.
- Usually, minimal inter-arrival time T is assumed to be known

Example Real-Time System



Goal: guarantee that no real-time deadline is missed!!!

EECS 388: Embedded Systems

8. Real-Time Scheduling (Part 2)

Heechul Yun

Recap

- Job
  - A computation instance
- Task
  - A sequence of jobs
- Periodic task model
  - ti = (Ci, Ti) or (Ci, Ti, Di)

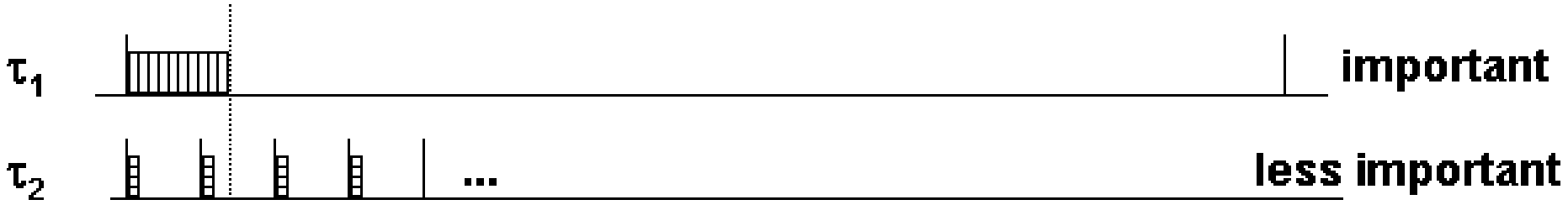
Real-Time Scheduling

- Scheduling
  - Pick which task to run next
- Priority-based scheduling
  - Pick the highest priority task among ready tasks

Priority and Criticality

- Priority
  - priority is the order we execute ready jobs.
- Criticality (Importance)
  - represents the penalty if a task misses a deadline (one of its jobs misses a deadline).
- Quiz
  - Which task should have higher priority?
  - Task 1: The most import task in the system: if it does not get done, serious consequences will occur
  - Task 2: A mp3 player: if it does not get done in time, the played song will have a glitch
  - If it is feasible, we would like to meet the real-time deadlines of both tasks
  - Answer: It depends…

Priority and Criticality



- If priorities are assigned according to importance, you can miss the deadline even if the system is mostly idle
- If p2 (period of t2) < C1 (execution time of t1), t2 will miss the deadline

Utilization

- A task's utilization (of the CPU) is the fraction of time for which it uses the CPU (over a long interval of time).
- A periodic task's utilization U_i (of CPU) is the ratio between its execution time and period: $U_i = C_i/p_i$
- Given a set of periodic tasks, the total CPU's utilization is equal to the sum of periodic tasks' utilization:

$$U = \sum_i \frac{C_i}{p_i}$$

- QUIZ: What is the obvious limit on U?

Real-Time Scheduling Algorithms

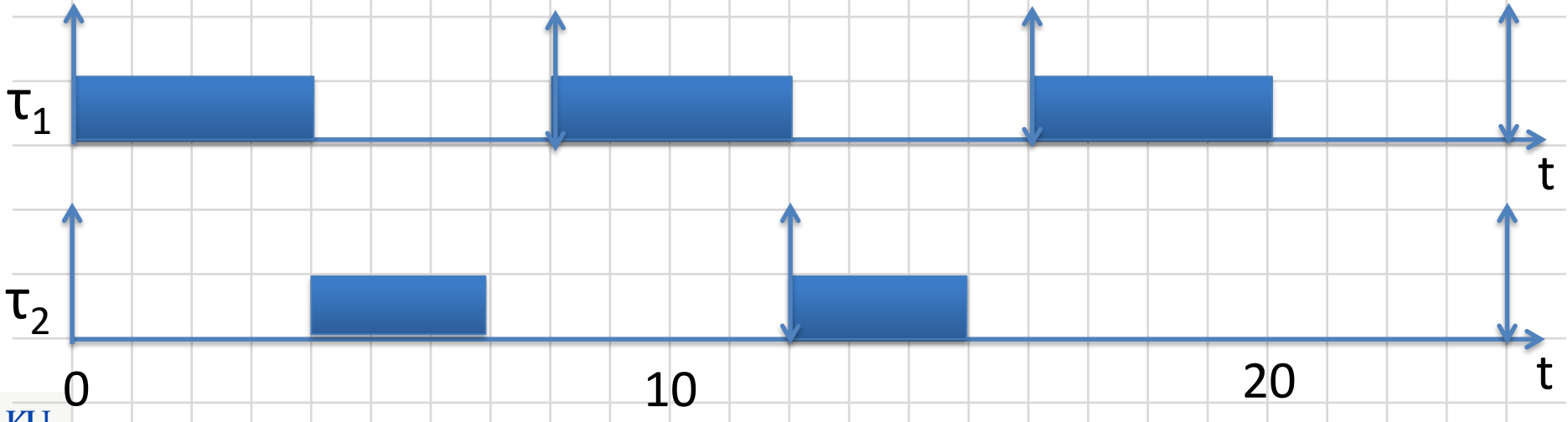
- Fixed-priority scheduling
  - All jobs of a task have the same priority
  - Rate Monotonic (RM)
- Dynamic priority scheduling
  - Different jobs of the same task may have different priorities
  - Earliest Deadline First (EDF)

Rate Monotonic (RM)

- Fixed-priority scheduling algorithm
- Assigns priorities to tasks on the basis of their periods
- **Shorter period = higher priority.**

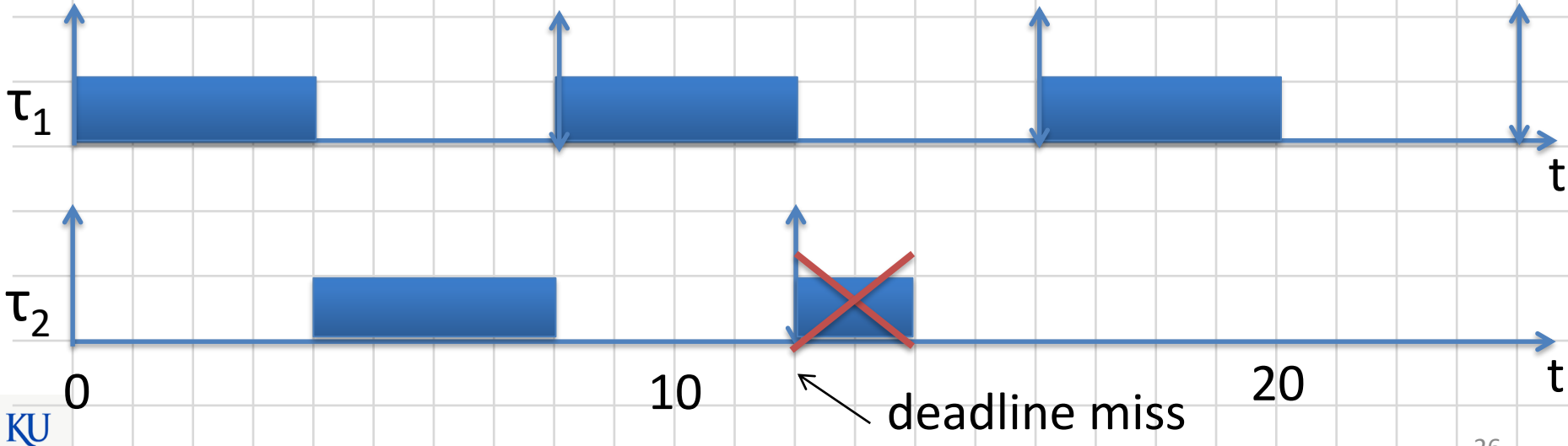
RM Example

- Case#1
  - τ_1 (C1 = 4, T1 = 8), high prio
  - τ_2 (C2 = 3, T2 = 12), low prio
  - Utilization: U = 4/8 + 3/12 = 0.75



RM Example

- Case#2
  - τ_1 (C1 = 4, T1 = 8), high prio
  - τ_2 (C2 = 6, T2 = 12), low prio
  - Utilization: U = 4/8 + 6/12 = 1

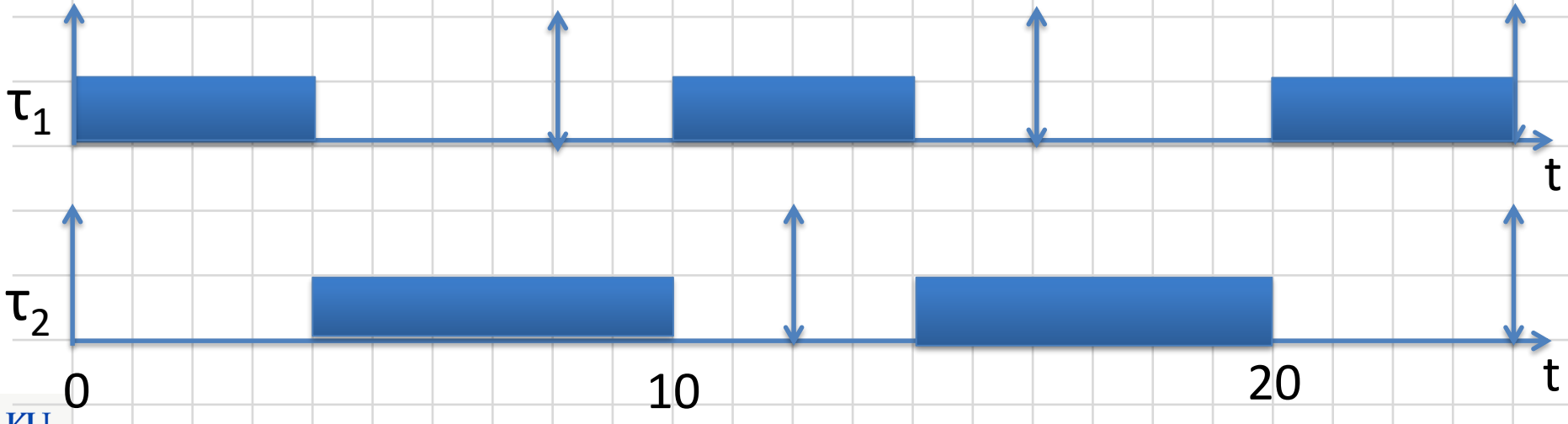


Earliest Deadline First (EDF)

- Dynamic-Priority Scheduling Algorithm
- Task priority is inversely proportional to its current absolute deadline
- **Earlier deadline = higher priority**
- Each job of a task has a different deadline, hence a different priority.

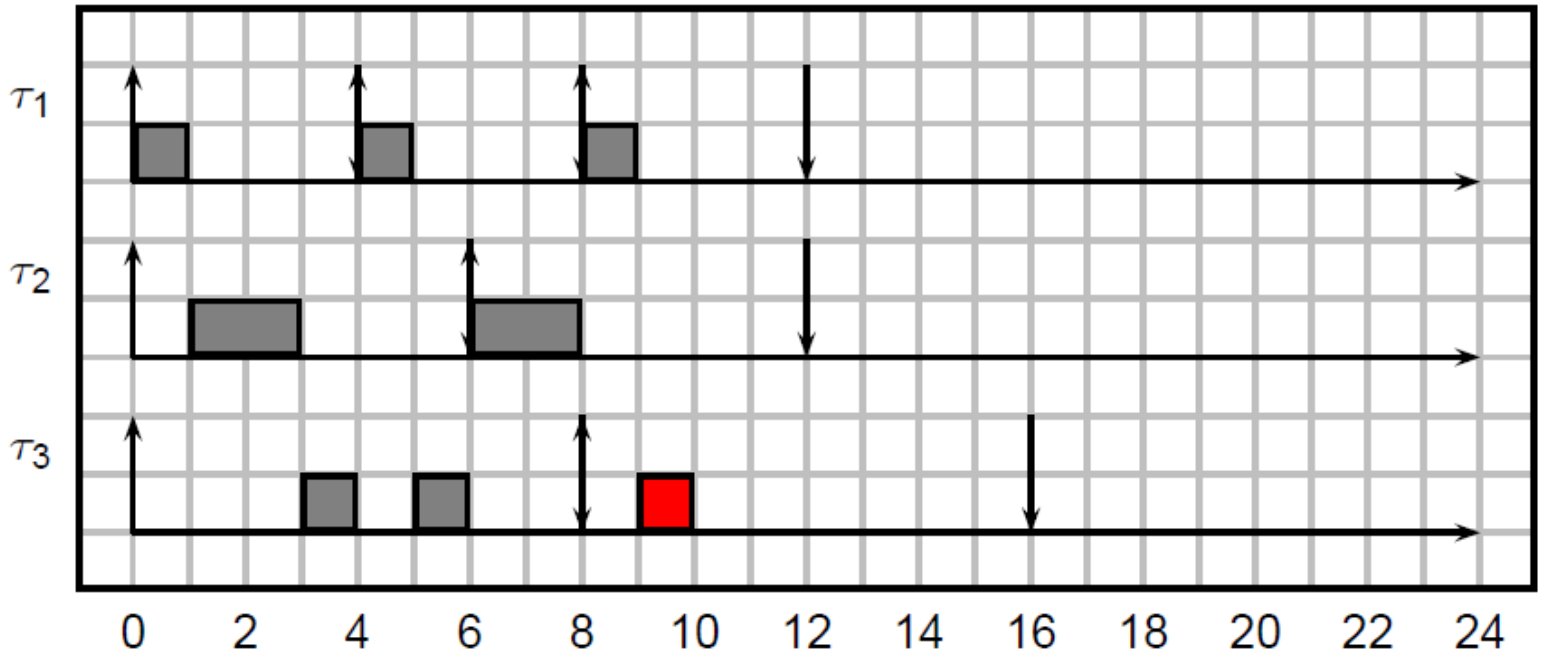
EDF Example

- Case#2:
  - τ_1 (C1 = 4, T1 = 8), τ_2 (C2 = 6, T1 = 12)
  - Utilization: U = 4/8 + 6/12 = U_b = 1



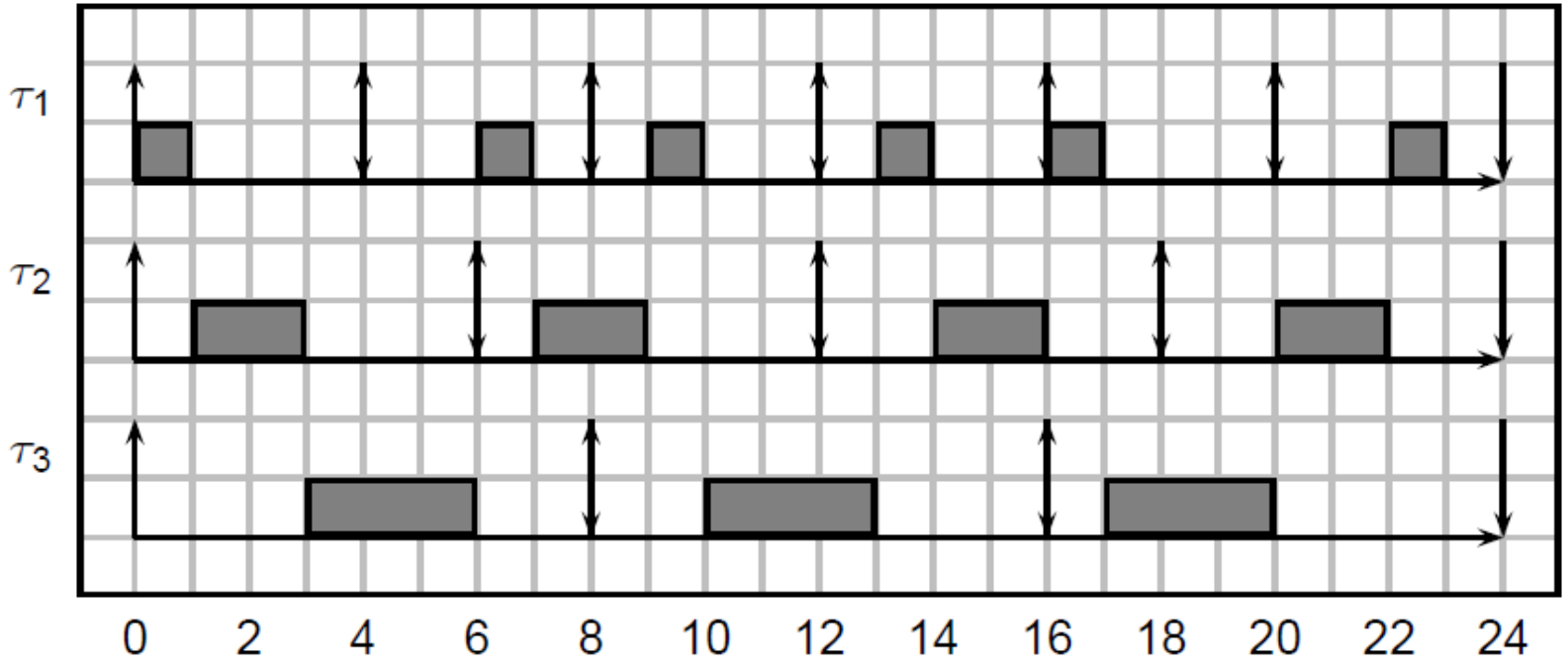
RM vs. EDF

- τ_1 (C1 = 1, T1 = 4), τ_2 (C2 = 2, T1 = 6), τ_3 (C3 = 3, T3 = 8)
- Utilization: U = 1/4 + 2/6 + 3/8 = 23/24
- Schedulable in RM?
  - No



RM vs. EDF

- τ_1 (C1 = 1, T1 = 4), τ_2 (C2 = 2, T1 = 6), τ_3 (C3 = 3, T3 = 8)
- Utilization: U = 1/4 + 2/6 + 3/8 = 23/24
- Schedulable in EDF?
  - Yes



Key Results

- For periodic tasks with relative deadlines equal to their periods:

- Rate monotonic scheduling is the optimal fixed-priority priority policy
  - No other static priority assignment can do better
  - Yet, it cannot achieve 100% CPU utilization

- Earliest deadline first scheduling is the optimal dynamic priority policy
  - EDF can achieve 100% CPU utilization

RM vs. EDF

- In practice, industrial systems heavily favor RM over EDF. Why?

- RM is easier to implement
  - Task priority never changes.
- RM is more transparent and robust
  - easier to understand what is going on if something goes wrong (ex: overload).
  - if a task executes for longer than its prescribed worst-case time, higher priority tasks will be left untouched.

EECS 388: Embedded Systems

9. Real-Time Scheduling (Part 3)

Heechul Yun

So far

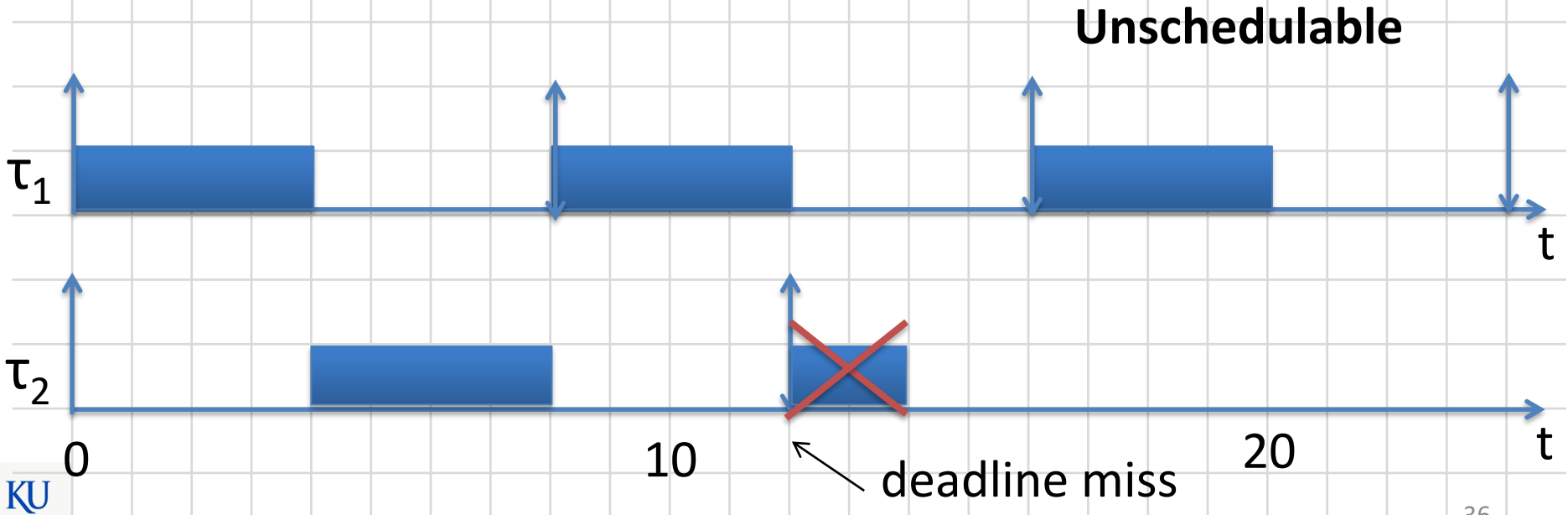
- Job, Task
- Periodic task model
  - ti = (Ci, Pi) or (Ci, Pi, Di)
- Static/dynamic priority scheduling:
  - RM
  - EDF
- Utilization
  - Ui = Ci / Pi $U = \sum_i \frac{C_i}{p_i}$

Agenda

- Utilization Bound
- Exact Schedulability analysis
- POSIX scheduling interface

Recall: RM Example

- τ_1 (C1 = 4, T1 = 8), high prio
- τ_2 (C2 = 6, T1 = 12), low prio
- Utilization: U = 4/8 + 6/12 = 1

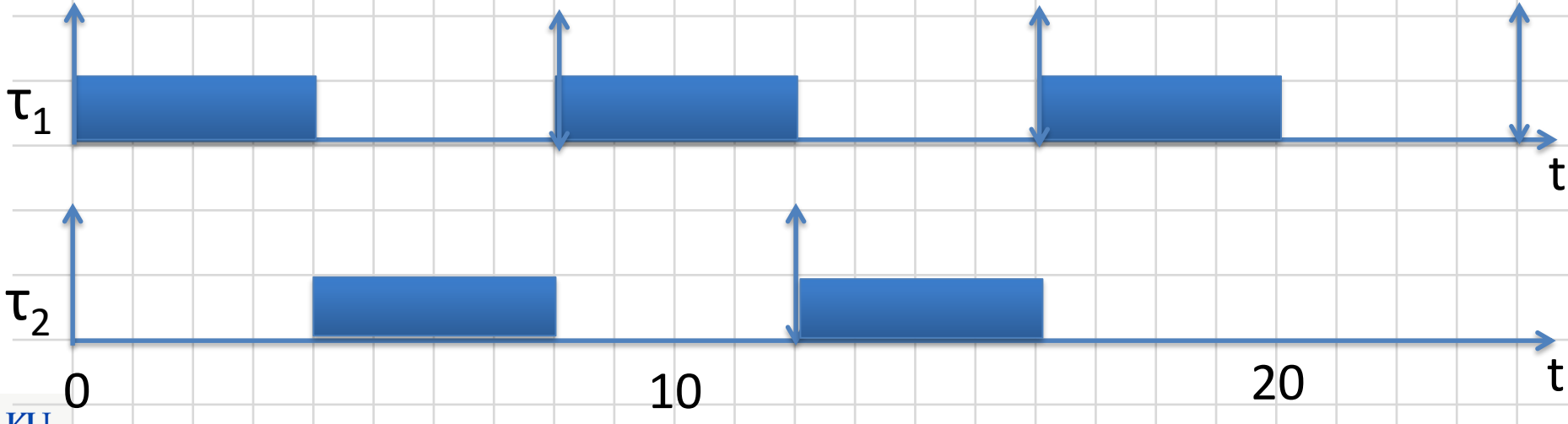


RM Example

- τ_1 (C1 = 4, T1 = 8), high prio
- τ_2 (C2 = **4**, T1 = 12), low prio
- Utilization: U = 4/8 + **4**/12 = 10/12 = 0.83

Is there an easy way to know whether a taskset is schedulable or not?

able!



Liu & Layland, JACM, Jan. 1973

Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment

C. L. LIU

Project MAC, Massachusetts Institute of Technology

AND

JAMES W. LAYLAND

Jet Propulsion Laboratory, California Institute of Technology

ABSTRACT. The problem of multiprogram scheduling on a single processor is studied from the viewpoint of the characteristics peculiar to the program functions that need guaranteed service. It is shown that an optimum fixed priority scheduler possesses an upper bound to processor utilization which may be as low as 70 percent for large task sets. It is also shown that full processor utilization can be achieved by dynamically assigning priorities on the basis of their current deadlines. A combination of these two scheduling techniques is also discussed.

KEY WORDS AND PHRASES: real-time multiprogramming, scheduling, multiprogram scheduling, dynamic scheduling, priority assignment, processor utilization, deadline driven scheduling

CR CATEGORIES: 3.80, 3.82, 3.83, 4.32

Liu & Layland Bound

- A set of n periodic tasks is schedulable if

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

  - UB(1) = 1.0
  - UB(2) = 0.828
  - UB(3) = 0.779
  - …
  - UB(n) = ln(2) = **~0.693**

Q. If it isn't, does that mean the taskset is unschedulable?

A. Not necessarily.
It's a sufficient condition, but not necessary one.

Sample Problem

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

L&L Bound

UB(1) = 1.0
UB(2) = 0.828
UB(3) = 0.779
UB(n) = 0.693

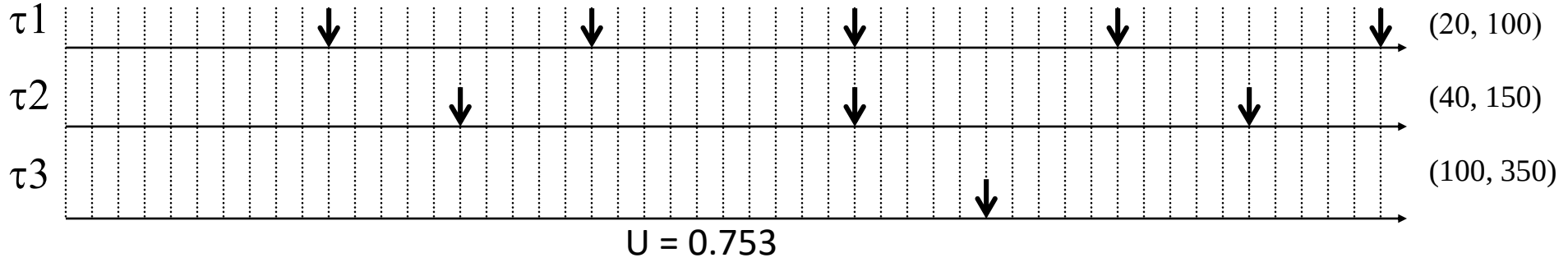
- Are all tasks schedulable?
  - $U_1 + U_2 + U_3 = 0.753 < U(3)$ ➔ **Schedulable!**
- What if we double the C of $\tau 1$
  - 0.2*2 + 0.267+ 0.286 = 0.953 > UB(3) = 0.779
  - **We don't know yet.**

KU THE UNIVERSITY OF KANSAS

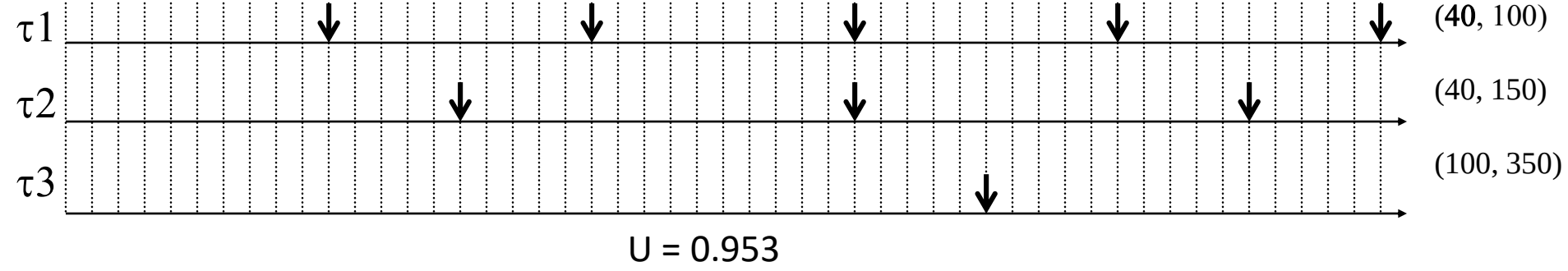
L&L Bound

UB(3) = 0.779
UB(n) = 0.693

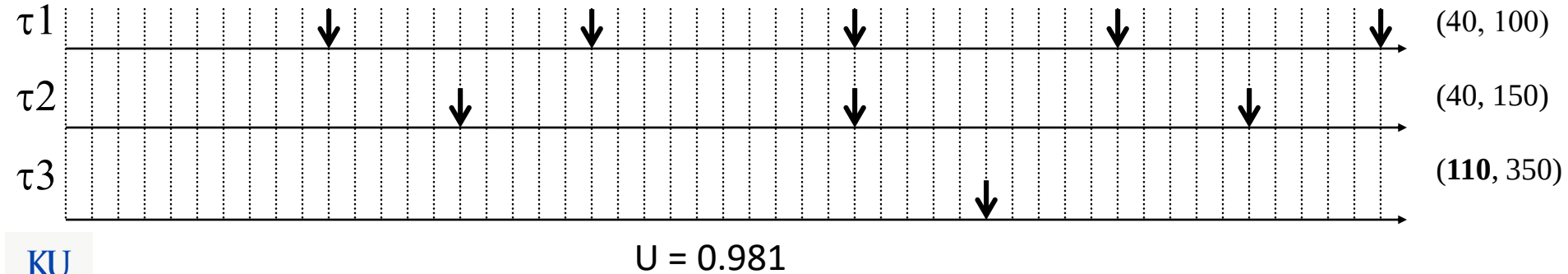
Sample Problem



U = 0.753



U = 0.953

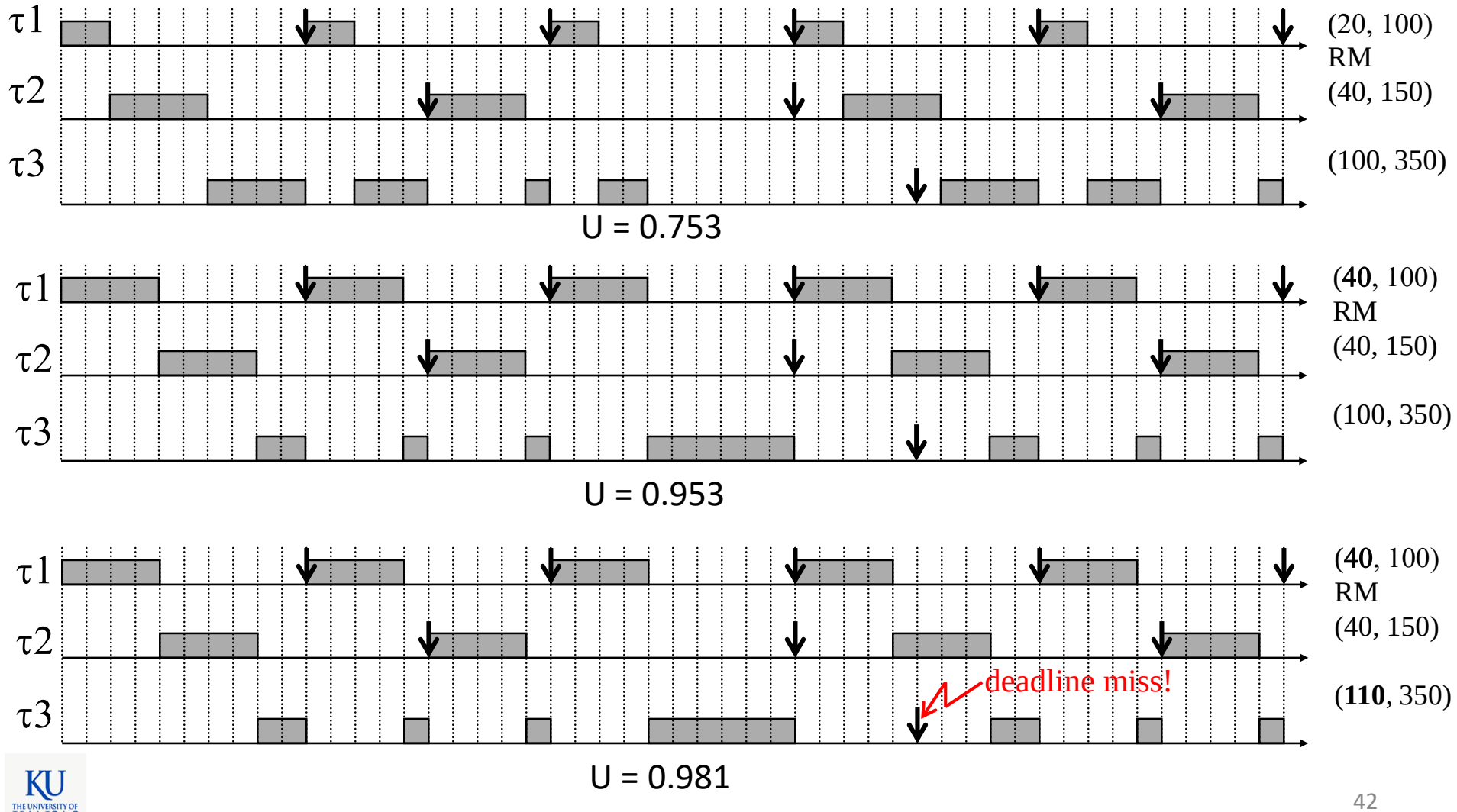


U = 0.981

Sample Problem

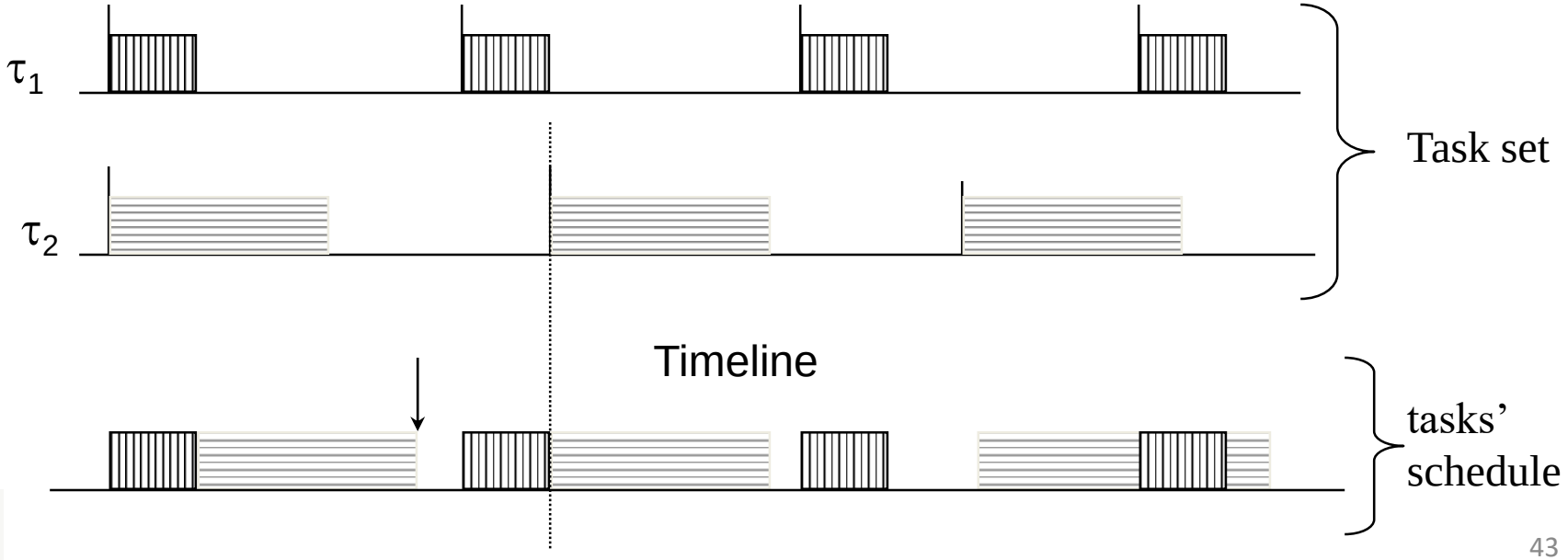
L&L Bound

UB(3) = 0.779
UB(n) = 0.693



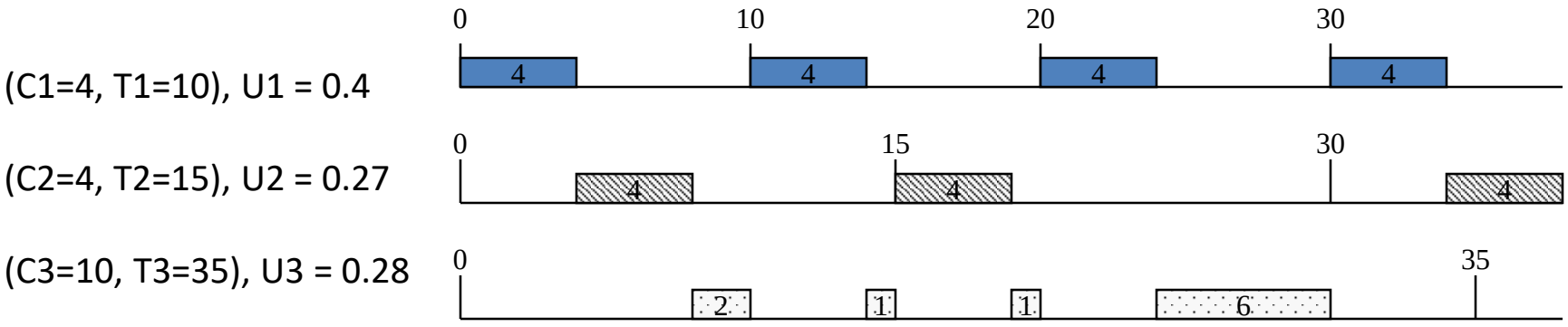
Critical Instant Theorem

- 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.



Exact Schedulability Test

- For each task, checks if it can meet its first deadline



Exact Schedulability Test

- For each task, checks if it can meet its first deadline

ceiling function

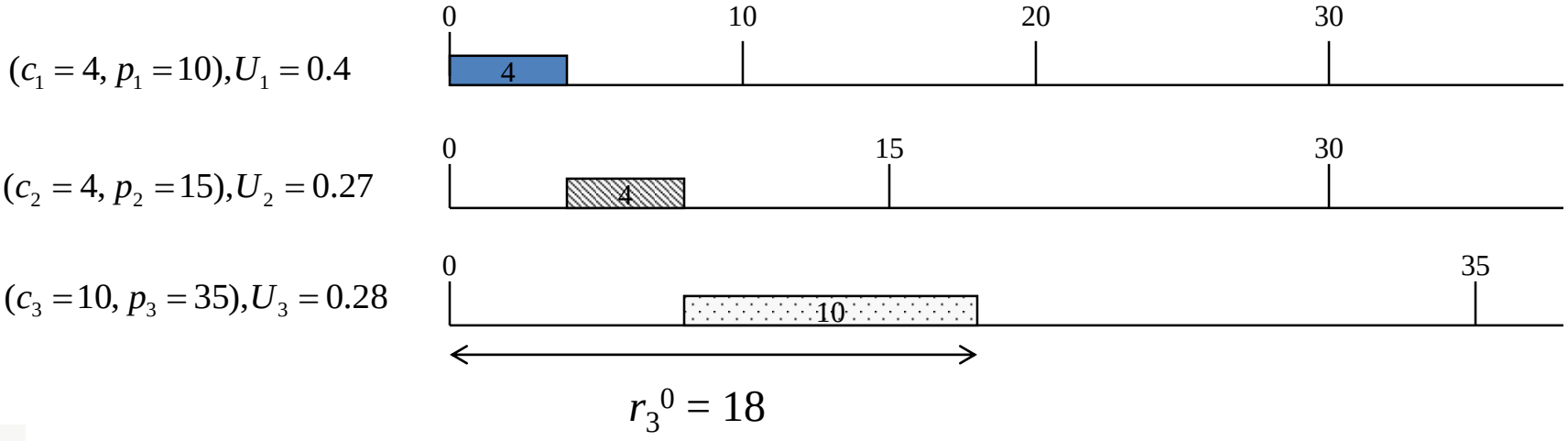
$$r_i^{k+1} = c_i + \sum_{j=1}^{i-1} \left\lceil \frac{r_i^k}{p_j} \right\rceil c_j, \quad \text{where } r_i^0 = \sum_{j=1}^{i} c_j$$

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

Exact Schedulability Test

- For task 3
  - First iteration

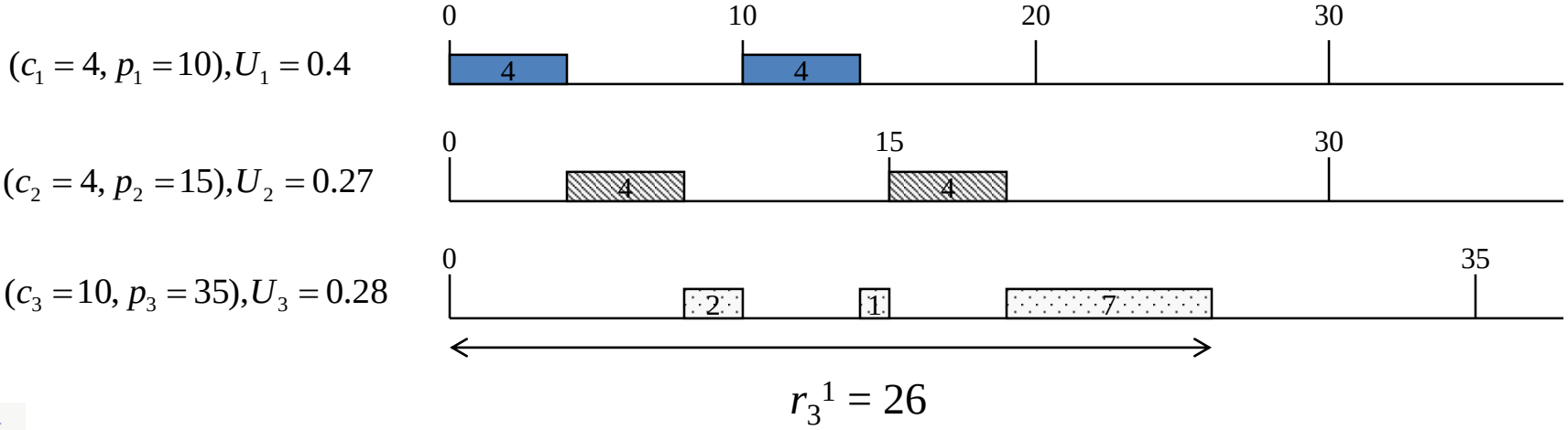
$$r_3^0 = \sum_{j=1}^{3} c_j = c_1 + c_2 + c_3 = 4 + 4 + 10 = 18$$



Exact Schedulability Test

- For task 3
  - Second iteration

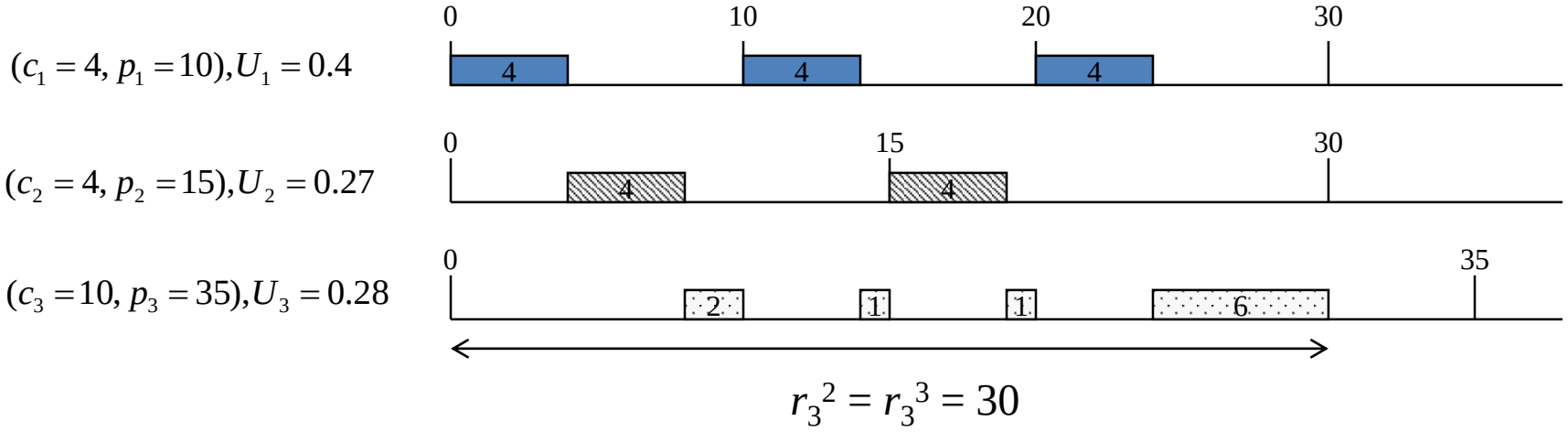
$$r_3^1 = c_3 + \sum_{j=1}^{2} \left\lceil \frac{r_3^0}{p_j} \right\rceil \cdot c_j = 10 + \left\lceil \frac{18}{10} \right\rceil 4 + \left\lceil \frac{18}{15} \right\rceil 4 = 26$$



Exact Schedulability Test

- For task 3
  - Third iteration

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



Exact Schedulability Test

- For task 3
  - Fourth iteration … is the same as the 3rd

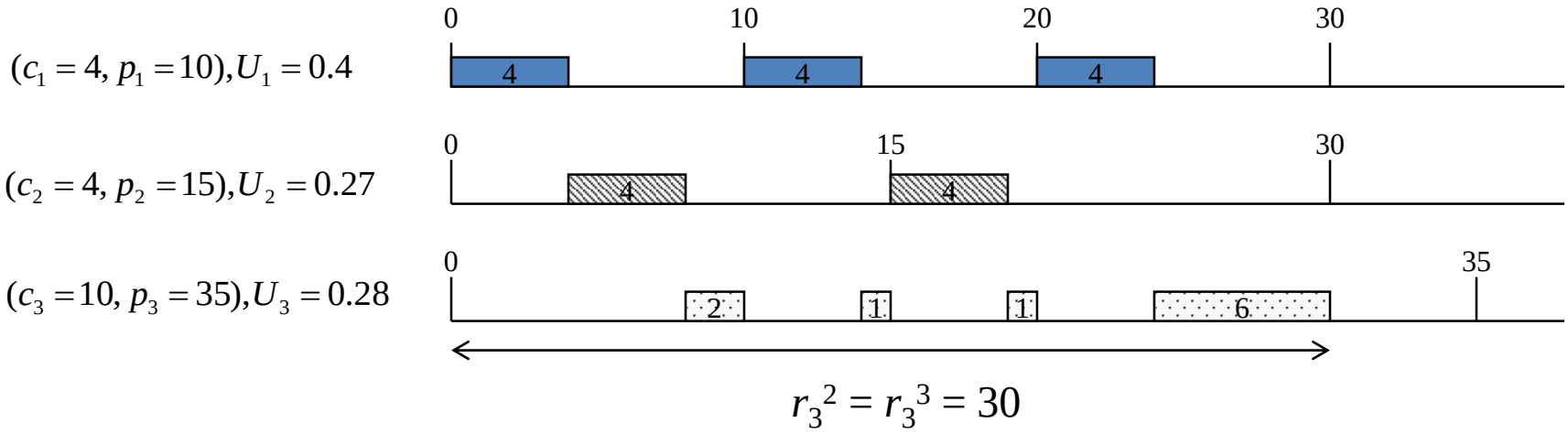
$$r_3^3 = c_3 + \sum_{j=1}^{2} \left\lceil \frac{r_3^2}{p_j} \right\rceil \cdot c_j = 10 + \left\lceil \frac{30}{10} \right\rceil 4 + \left\lceil \frac{30}{15} \right\rceil 4 = 30$$

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

  - **Done!**

Exact Schedulability Test

- All tasks meet their deadlines → schedulable



Caveats: Assumptions

- So far the theories assume
  - 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

POSIX Scheduling Interface

- POSIX.4 Real-Time Extension support real-time scheduling policies
- 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 defined policies
  - SCHED_FIFO: preemptive, priority-based scheduling.
  - SCHED_RR: Preemptive, priority-based scheduling with quanta.
  - SCHED_OTHER: an implementation-defined scheduler → Linux's default scheduler (CFS)

SCHED_FIFO

- Preemptive, priority-based scheduling.
- Priority ranges: 1 (lowest) – 99 (highest)
- 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.

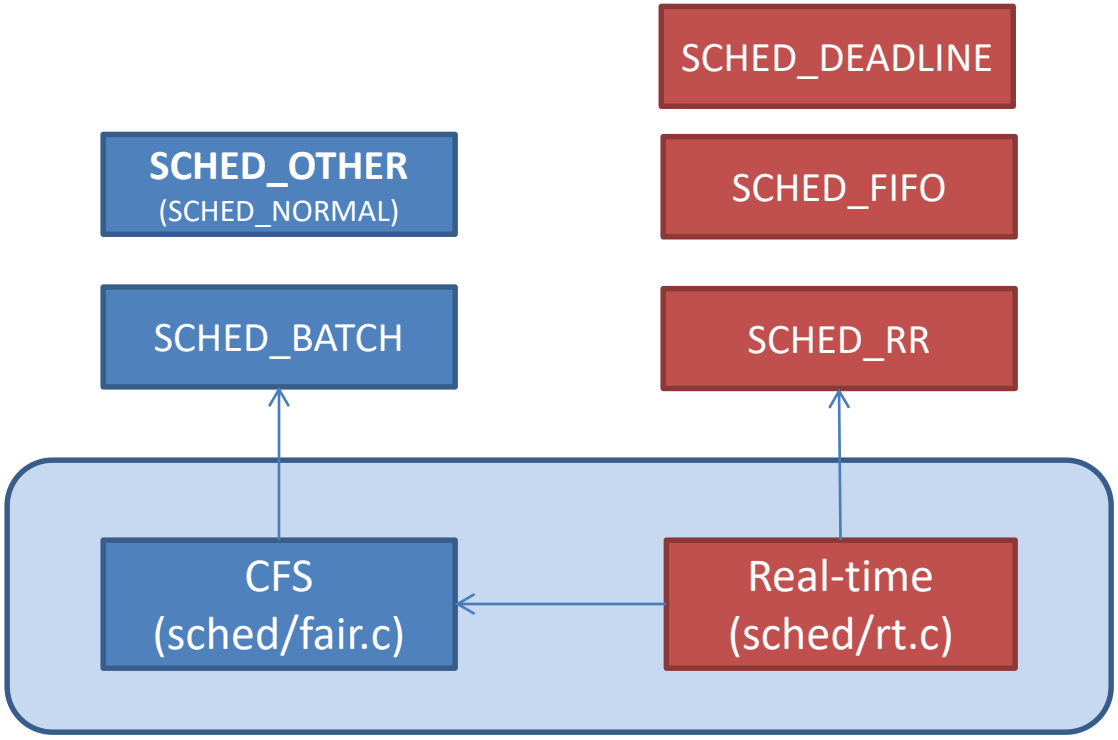
SCHED_RR

- Same as SCHED_FIFO except the following.
- Time slicing among the same priority tasks:
  - 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**.

SCHED_OTHER

- An implementation defined scheduler, not defined by POSIX.4
- In Linux, this class is the default CFS scheduler.

Linux Scheduling Framework

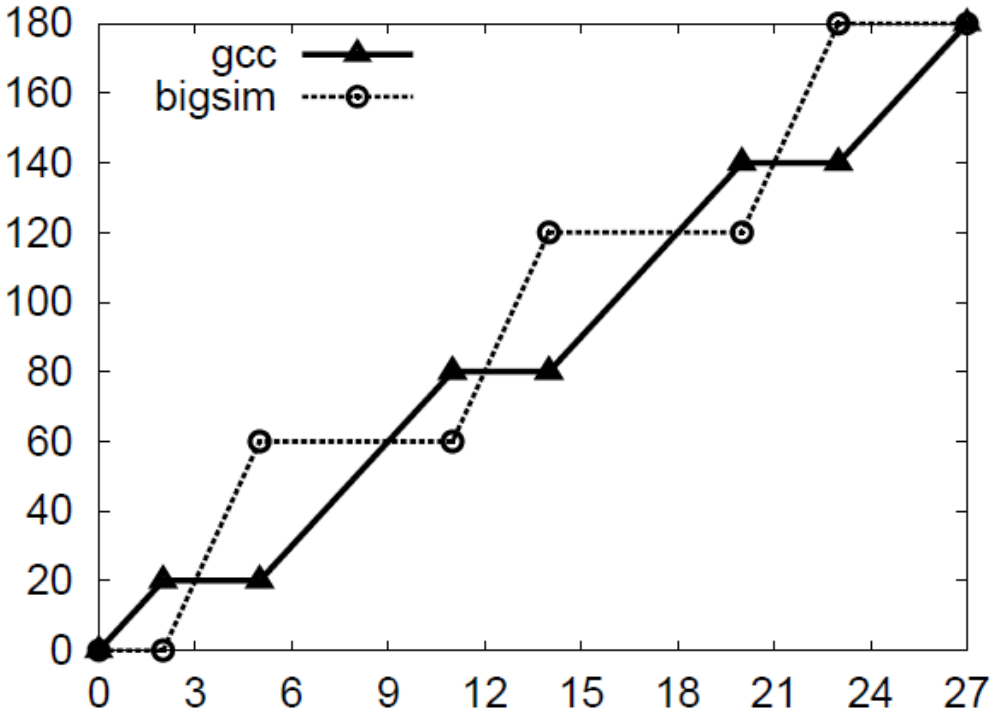


- Completely Fair Scheduler (CFS) ← for general purpose
- Real-time Schedulers ← for real-time apps.
- **Why not to create a single scheduler for both?**

Completely Fair Scheduler (CFS)

- SCHED_OTHER class
- Linux's default scheduler, focusing on **fairness**
- Each task owns a fraction of CPU time share
  - E.g.,) A=10%, B=30%, C=60%
- Scheduling algorithm
  - Each task maintains its virtual runtime
    - Virtual runtime = executed time (x 1 / weight)
  - Pick the task with the **smallest virtual runtime**
    - Tasks are sorted according to their virtual times

CFS Example



Weights: gcc = 2/3, bigsim=1/3

X-axis: mcu (tick), Y-axis: virtual time

Fair in the long run

kernel/sched/fair.c (CFS)

- Priority to CFS weight conversion table
  - Priority (Nice value): -20 (highest) ~ +19 (lowest)
  - kernel/sched/core.c

```
const int sched_prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

Summary

- Utilization Bound
- Exact Schedulability analysis
- POSIX scheduling interface

Acknowledgements

- These slides draw on materials developed by
  - Lui Sha and Marco Caccamo (UIUC)
  - Rodolfo Pellizzoni (U. Waterloo)
  - Edward A. Lee and Prabal Dutta (UCB) for EECS149/249A

KU THE UNIVERSITY OF KANSAS