CIS 721 - Real-Time Systems
Lecture 6: Utilization-Based Tests

Mitch Neilsen
neilsen@cis.ksu.edu

Outline

- Commonly Used Approaches For Real-Time Scheduling (Ch. 4)
  - **Clock-Driven Scheduling (Ch. 5)**
  - Priority-Driven Scheduling
    - Periodic Tasks (Ch. 6)
      - Utilization-Based Test
      - Response Time Analysis
    - Aperiodic or Sporadic Tasks (Ch. 7)

Tools Used

- HI_PR – an efficient implementation of the push-relabel method to solve maximum flow problems – build using gcc toolchain + make
- Convert – home-grown app to convert text max flow output to digraph input format
- GraphViz Editor (GVEdit) or dot to generate graphical output
- Putty: ssh client to cislinux.cis.ksu.edu
- FileZilla: secure file transfer

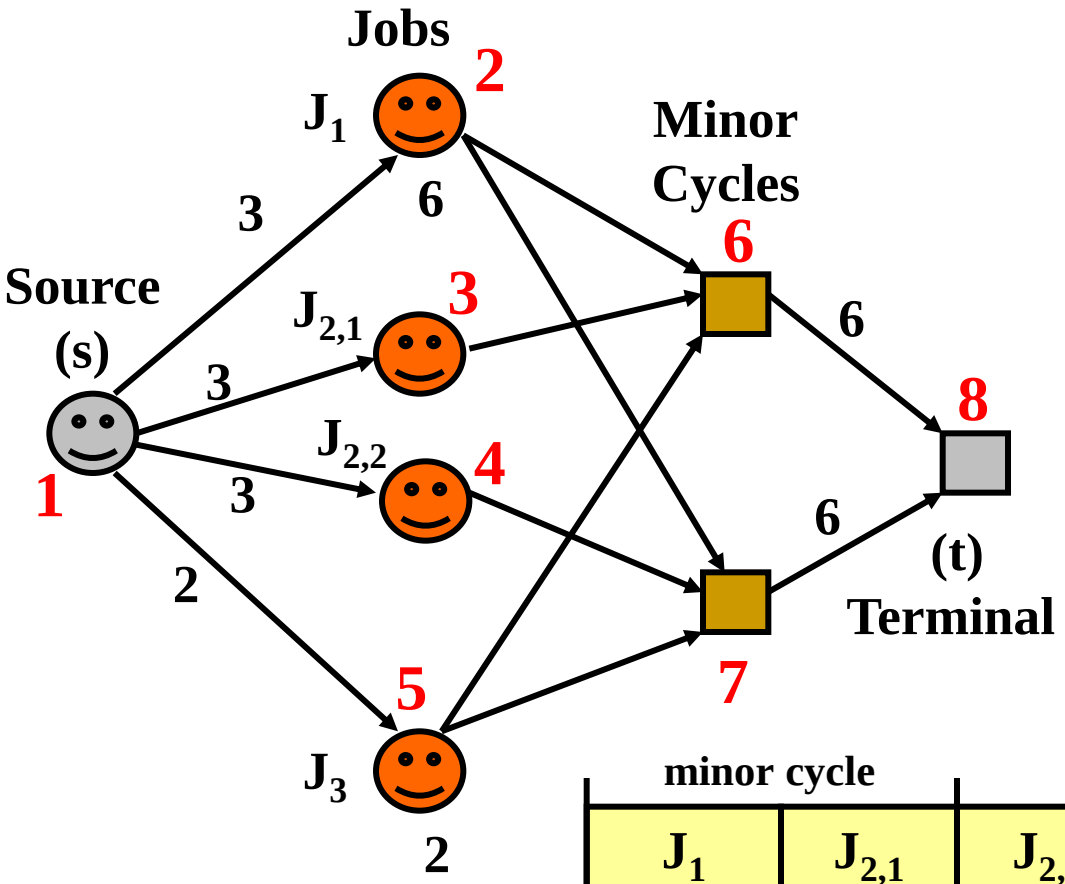
Clock-Driven Example

$ hi_pr < cyclic2.input > cyclic2.output

Tasks:
(12, 3), (6, 3), (12, 2)
Frame size: f = 6

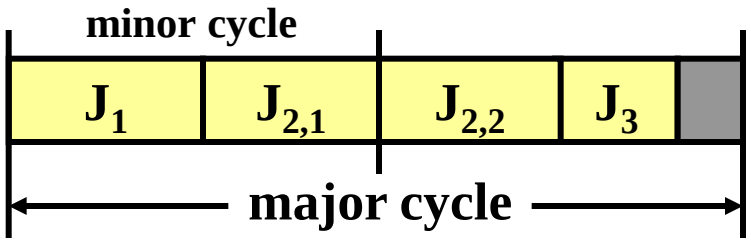
INPUT:

```
p max 8 12
n 1 s
n 8 t
a 1  2  3
a 1  3  3
a 1  4  3
a 1  5  2
a 2  6  6
a 2  7  6
a 3  6  6
a 4  7  6
a 5  6  6
a 5  7  6
a 6  8  6
a 7  8  6
```



OUTPUT:

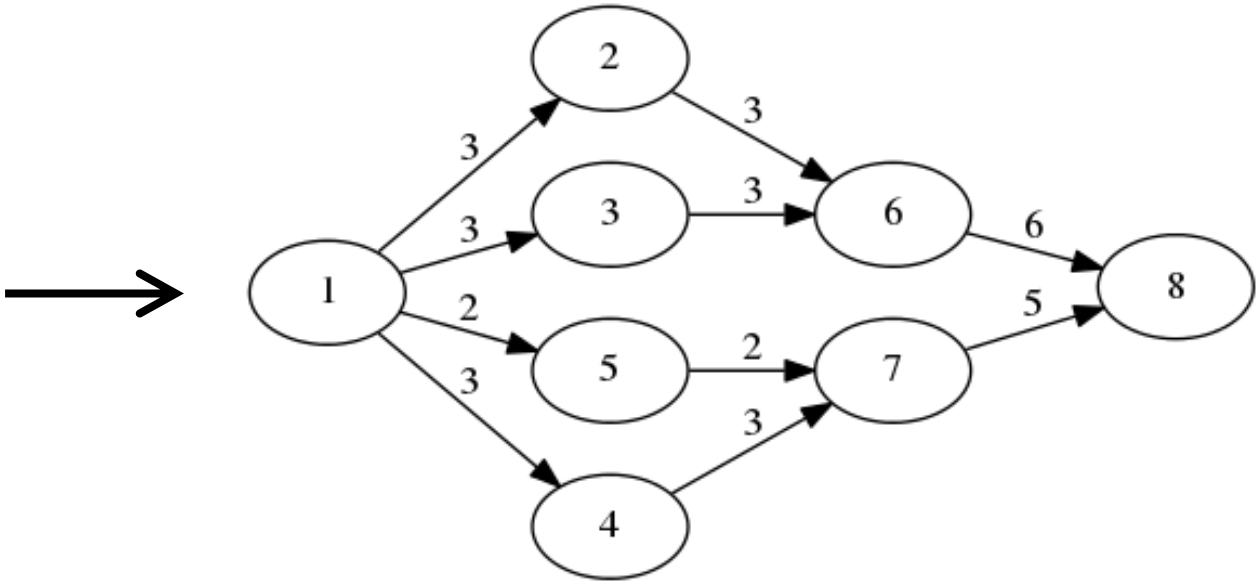
```
max flow:11
c flow values
f  1  2  3
f  1  4  3
f  1  3  3
f  1  5  2
f  2  6  3
f  2  7  0
f  3  6  3
f  4  7  3
f  5  7  2
f  5  6  0
f  6  8  6
f  7  8  5
c
```



Resulting Graphics File: cyclic2.gr

```
digraph g {
rankdir = LR;
1 -> 2 [label=3]
1 -> 4 [label=3]
1 -> 3 [label=3]
1 -> 5 [label=2]
2 -> 6 [label=3]
3 -> 6 [label=3]
4 -> 7 [label=3]
5 -> 7 [label=2]
6 -> 8 [label=6]
7 -> 8 [label=5]
{ rank=same;  2; 3; 4; 5;}
{ rank=same;  6; 7;}
}
```

```
$ convert1 cyclic2.output cyclic2.gr
$ dot -Tpng cyclic2.gr -o cyclic2.png
```



Outline

- Commonly Used Approaches For Real-Time Scheduling (Ch. 4)
  - Clock-Driven Scheduling (Ch. 5)
  - **Priority-Driven Scheduling**
    - **Periodic Tasks (Ch. 6)**
      - **Utilization-Based Test**
      - Response Time Analysis
    - Aperiodic or Sporadic Tasks (Ch. 7)

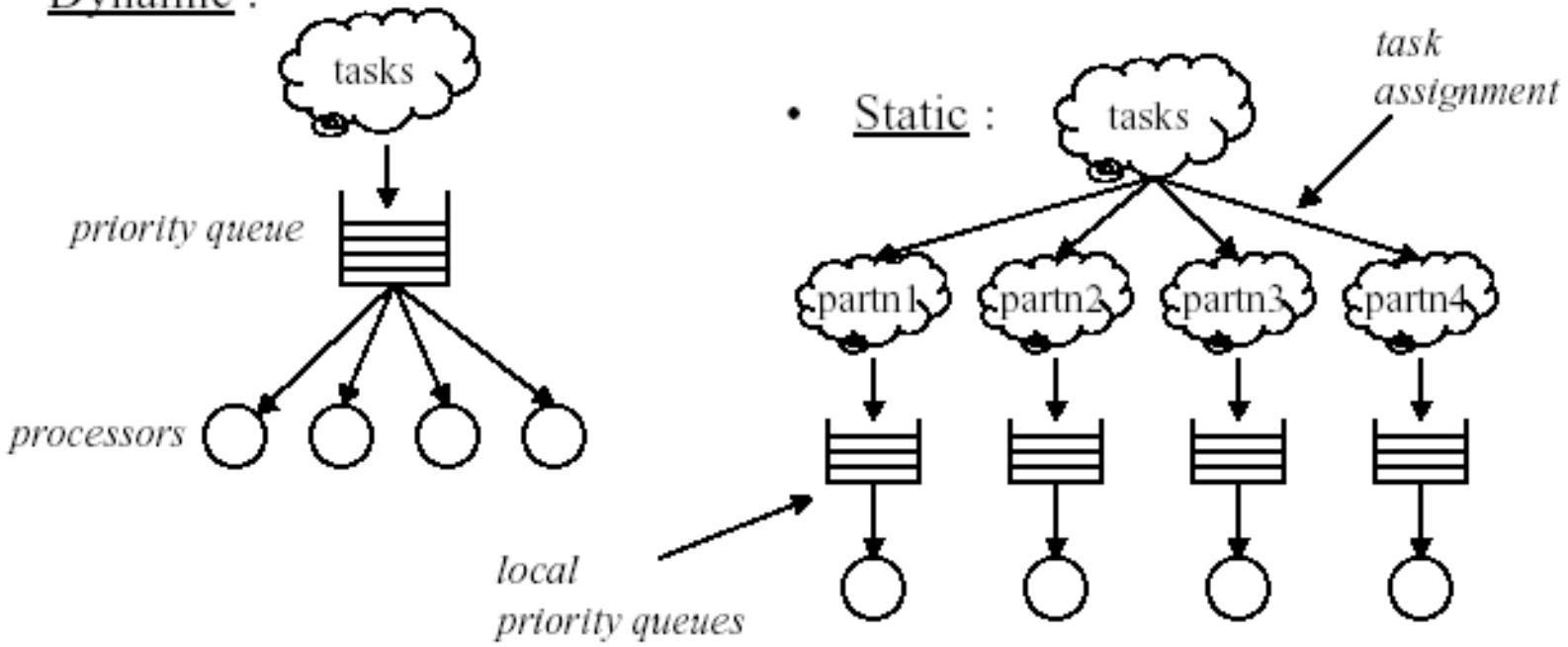
Scheduling of Periodic Tasks

- **Assumptions**
  - Tasks are independent
  - Preemption is allowed
  - All tasks are periodic
  - No sporadic or aperiodic tasks
  - Single processor

WHY A SINGLE PROCESSOR?

Why Focus on Uniprocessor Scheduling?

- Dynamic *vs.* static multiprocessor scheduling:
  - Dynamic :
  - Static :

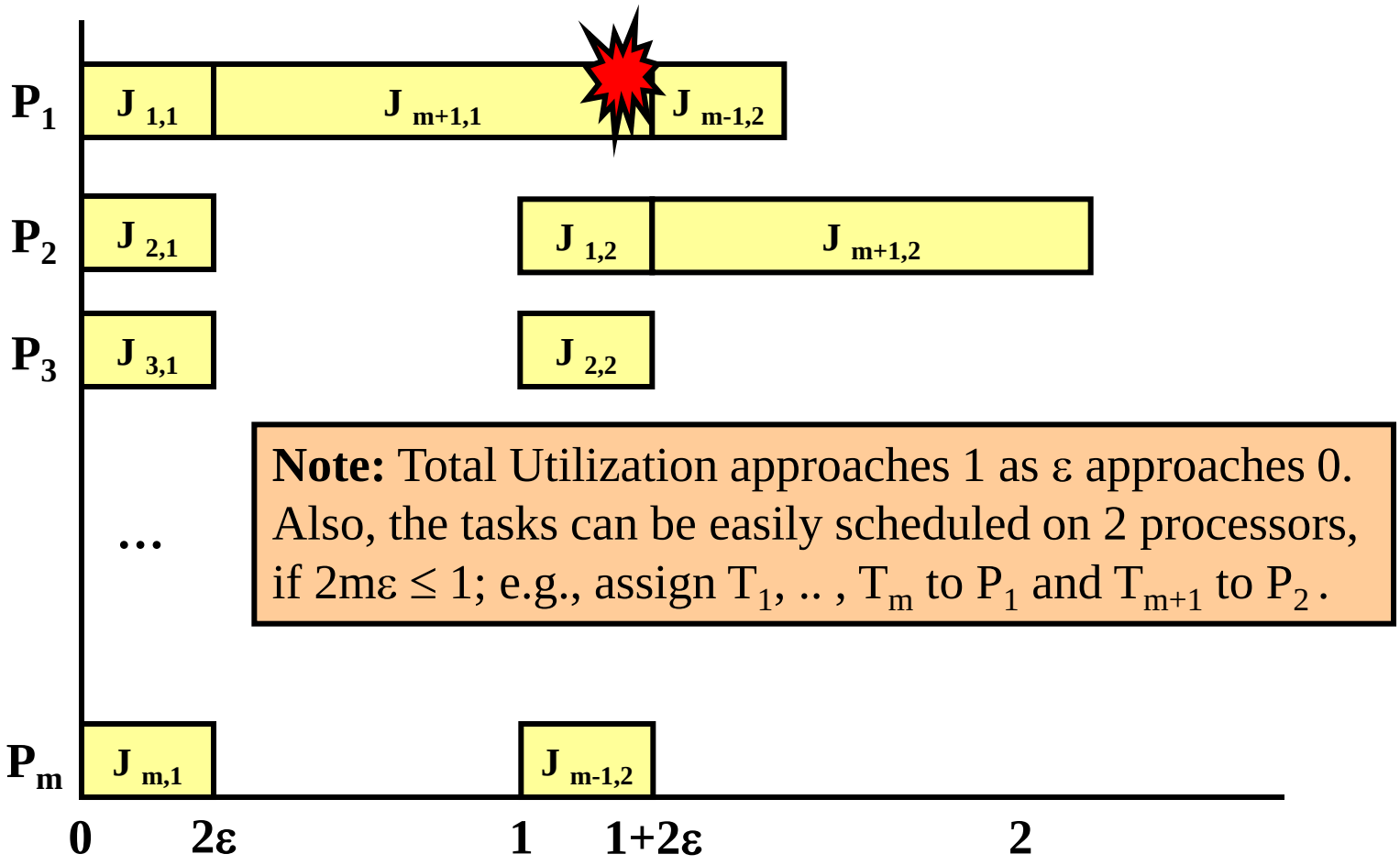


- Poor worst-case performance of priority-driven algorithms in dynamic environments.
- Difficulty in validating timing constraints.

Example of Poor Worst-Case Performance

- Here is an example to show that the performance of priority-driven algorithms with **dynamic processor assignment** can be **very poor**:
  - Number of processors = m
  - Number of independent periodic tasks = m+1
  - Small Tasks T_1 .. T_m are identical with $p_i = 1$, $e_i = 2\varepsilon$ for some small number ε
  - Large Task T_{m+1} has $p_{m+1} = \varepsilon + 1$, $e_{m+1} = 1$
  - Relative deadlines are equal to periods (for all tasks).
  - Apply a dynamic EDF algorithm to schedule the tasks on m processors.

Example (cont.)



Note: Total Utilization approaches 1 as ε approaches 0. Also, the tasks can be easily scheduled on 2 processors, if $2m\varepsilon \leq 1$; e.g., assign $T_1, .. , T_m$ to P_1 and T_{m+1} to P_2 .

Static vs. Dynamic Systems

- Although the poor behavior of **dynamic** systems occurs only for these types of pathological systems, the real problem is how to determine the **worst-case performance of dynamic systems**, other than by simulating and testing the system.
- Consequently, most hard real-time systems (for now and in the near future) are **static**. Well-grounded theories and algorithms can be used to validate efficiently, robustly, and accurately the timing constraints of static systems (as we shall see).
- Also, in a static system, uniprocessor algorithms can be easily **extended** to multiprocessor systems.

Issues in Fixed Priority Assignment On A Single Processor

- How to assign priorities?
- How to determine which assignment is the best; e.g., how to evaluate a priority assignment algorithm (method)?
- How to compare different priority assignment algorithms?

Rate-Monotonic Algorithm (RM)

- The **rate** of a task is the inverse of its period ($f_i = 1 / p_i$).
- Tasks with **higher rates (shorter periods)** are assigned **higher priorities**.
- C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment”, JACM, Vol. 20, No. 1, pages 46-61, 1973.

Deadline-Monotonic Algorithm (DM)

- Tasks with **shorter relative deadlines** are assigned **higher priorities**.
- When tasks have relative deadlines (D_i) equal to their periods (p_i), the rate-monotonic algorithm is the same as the deadline-monotonic algorithm.

Optimal Priority Assignment

- A given priority assignment algorithm is **optimal** if whenever a task set can be scheduled by some fixed priority assignment, it can also be scheduled by the given algorithm.
- Liu and Layland show that the rate-monotonic (RM) algorithm is optimal *for independent, preemptive, periodic tasks on a single processor.*

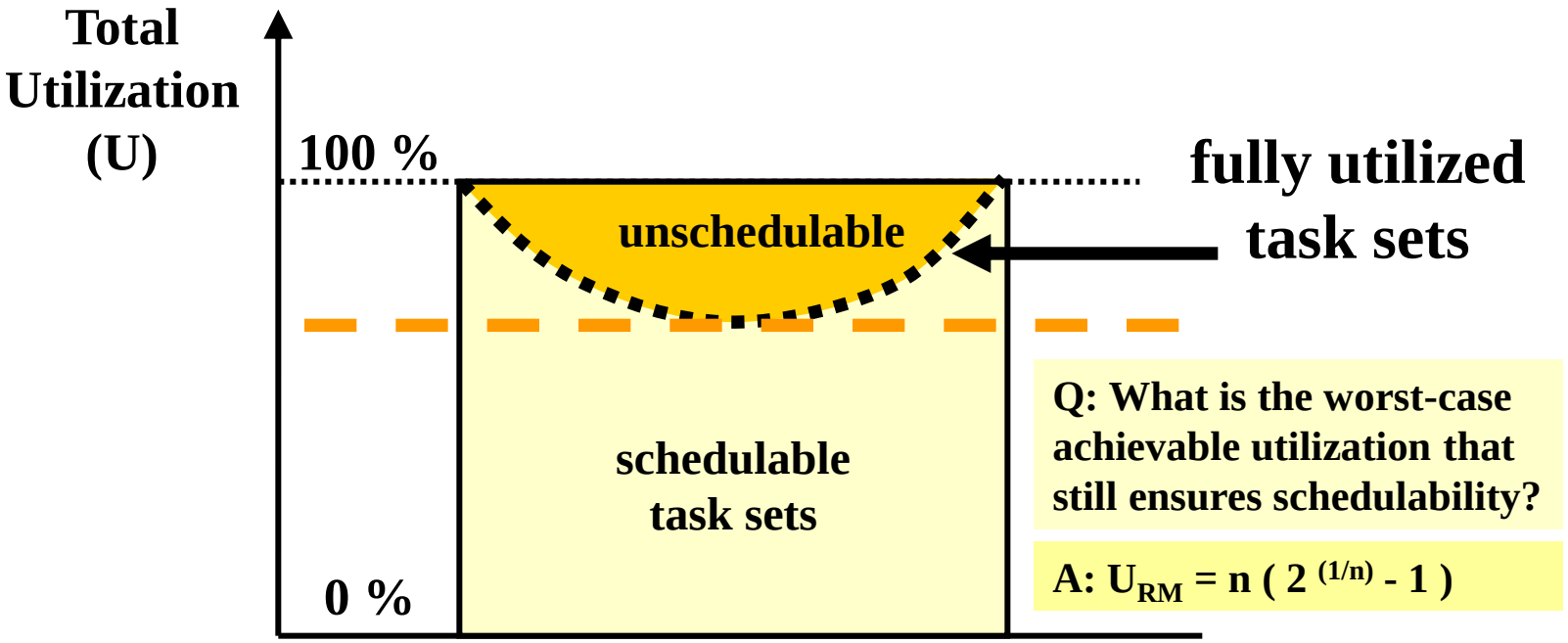
Processor Utilization

- Given a periodic task T_i , the ratio $u_i = e_i / p_i$ is called the **utilization of task T_i** .
- The **total utilization U** of all tasks in a system is the sum of the utilizations of all individual tasks:

$$U = \sum_{i=1}^{n} \frac{e_i}{p_i}$$

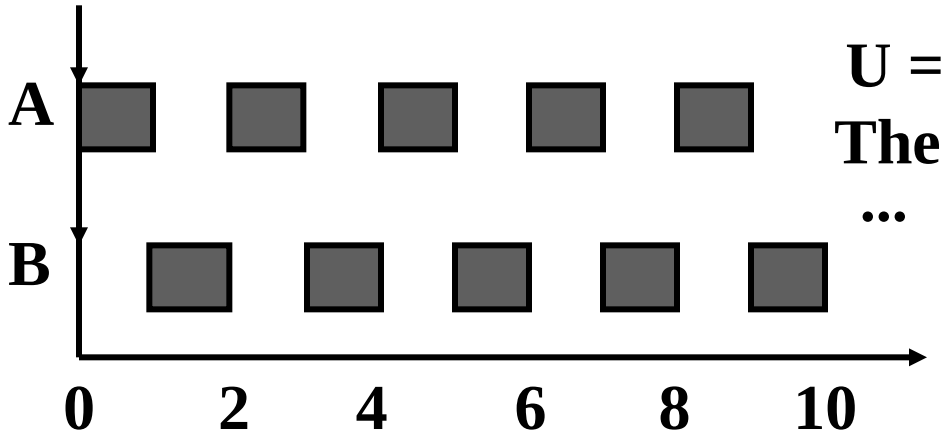
Fully Utilized Task Sets

A task set is **fully utilized** if any increase in run-time would result in a missed deadline.



Example #6

Task T_i		Period p_i	Deadline D_i	Run-Time e_i	Phase φ_i
A	(High Priority)	2	2	1	0
B	(Low Priority)	2	2	1	0



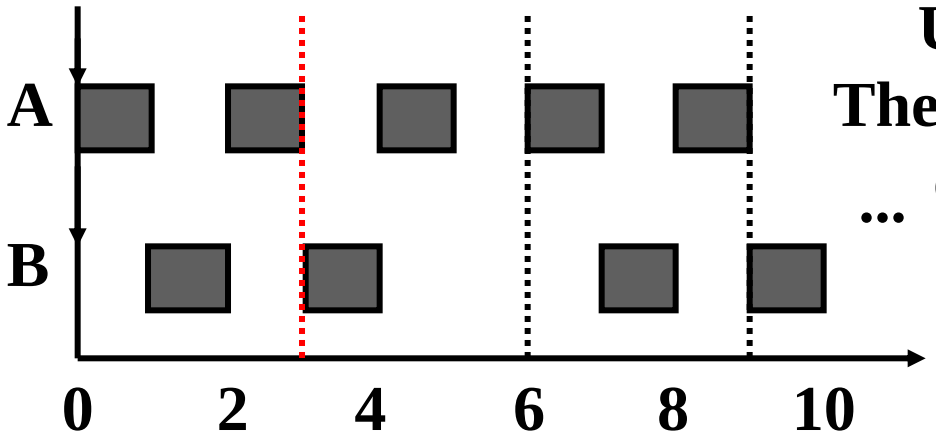
U = 1/2 + 1/2 = 1.0 = 100 %

The task set is fully utilized.

…

Example #7

Task T_i		Period p_i	Deadline D_i	Run-Time e_i	Phase φ_i
A	(High Priority)	2	2	1	0
B	(Low Priority)	3	3	1	0

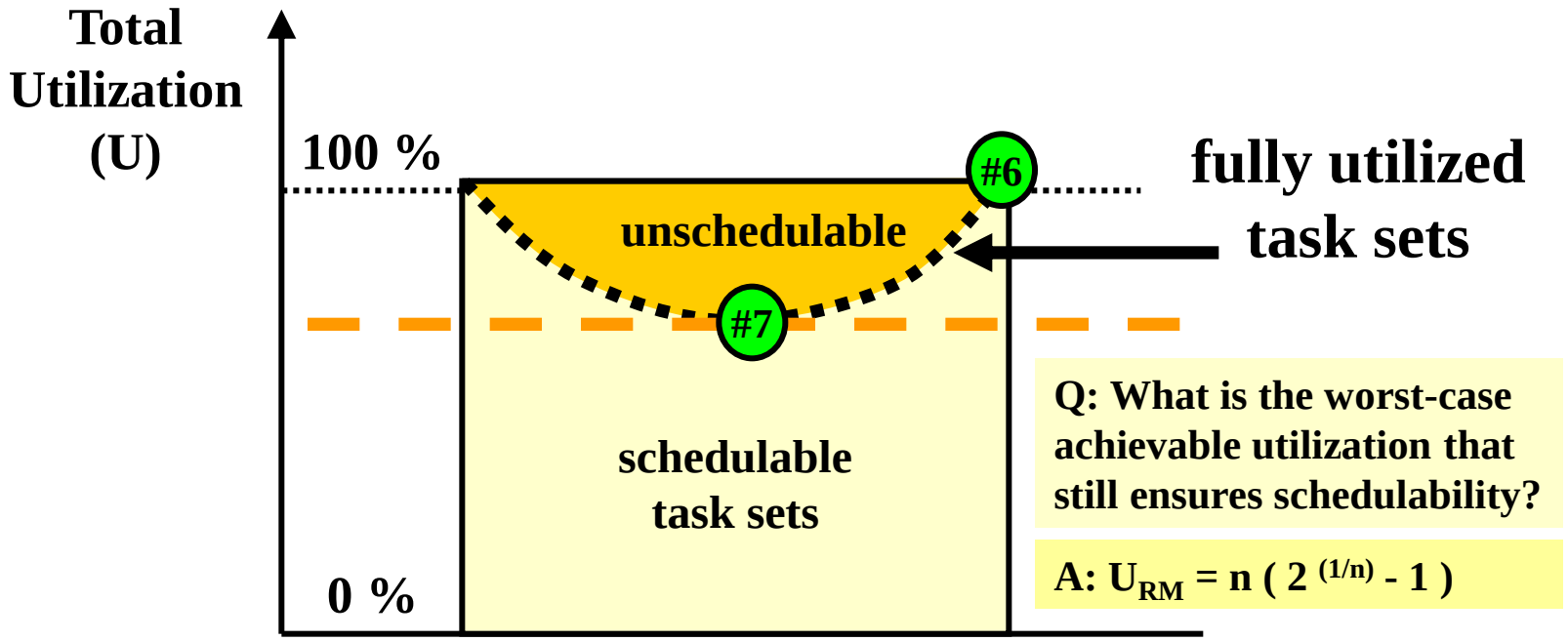


U = 1/2 + 1/3 = 0.8333

The task set is fully utilized, even though U < 1.0.

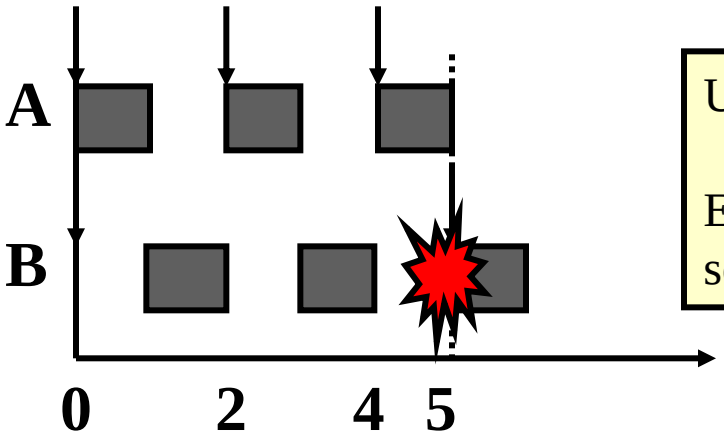
Fully utilized task sets

A task set is **fully utilized** if any increase in run-time would result in a missed deadline.



Example #8

Task T_i		Period p_i	Deadline D_i	Run-Time e_i	Phase φ_i
A	(High Priority)	2	2	1	0
B	(Low Priority)	5	5	2.1	0

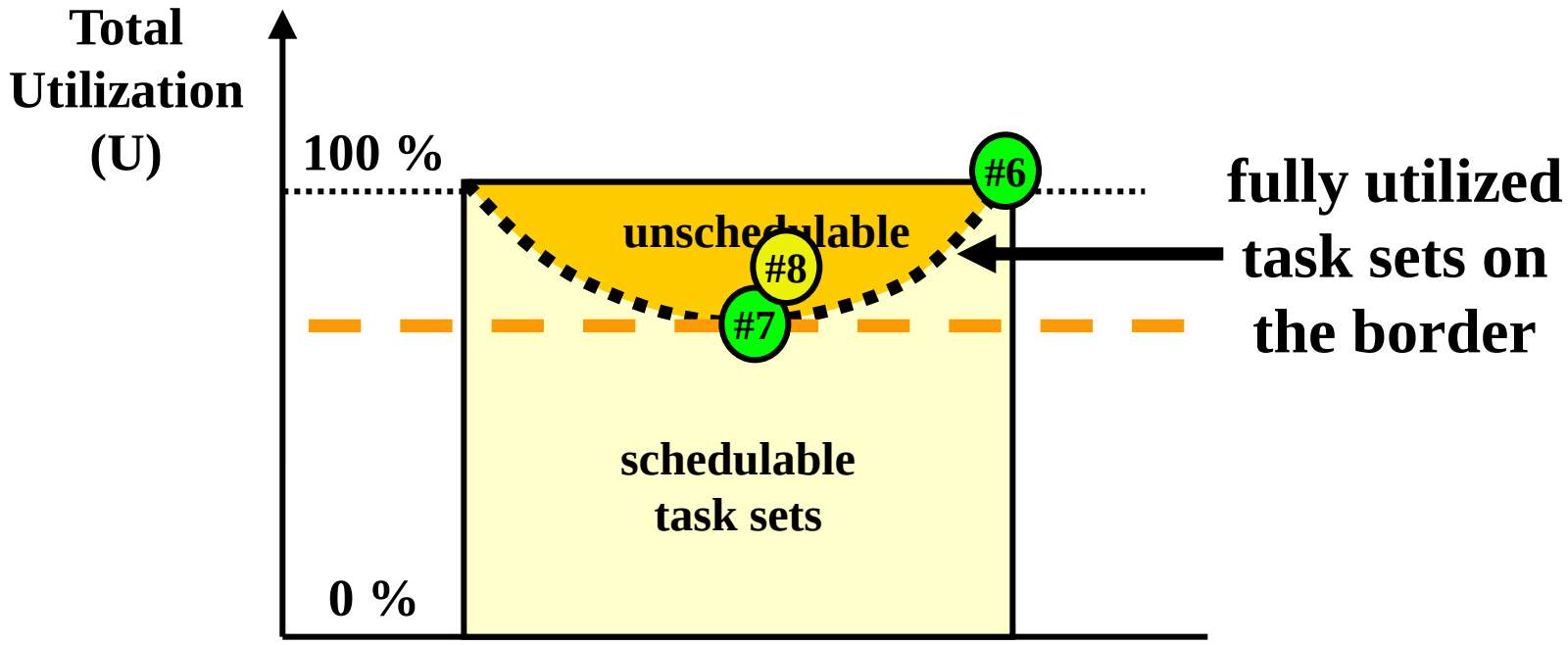


$U = C_1 / T_1 + C_2 / T_2 = 1 / 2 + 2.1 / 5 = 0.92$

Even if $U < 1$, a task set may not be schedulable using fixed priority scheduling.

Fully utilized task sets

A task set is **fully utilized** if any increase in run-time would result in a missed deadline.



Schedulable Utilization

- Every set of periodic tasks with total utilization less than or equal to the **schedulable utilization** of a scheduling algorithm can be feasibly scheduled using that algorithm.
- Schedulable utilization is always less than or equal to 1.0 = 100%.
- In a sense, the higher the schedulable utilization, the better the algorithm.

Examples

- First-In, First-Out (FIFO): $U_{FIFO} = 0$
- Earliest-Deadline-First (EDF): $U_{EDF} = 1$
- Rate-Monotonic Algorithm (RM):

  $U_{RM} = n (2^{(1/n)} - 1)$, where n = number of tasks

Utilization-Based Test

- A **sufficient, but not necessary, test** that can be used to test the schedulability of a task set that is assigned priorities using the given algorithm.
- Compute **total task utilization** U(n) = U.
- Compare with worst-case utilization bound (also called **schedulable utilization**) of a given scheduling algorithm (SA): $U_{SA}(n) = U_{SA}$:
  - If **U > 1**, then the task set is not schedulable.
  - If **$U \leq U_{SA}$**, then the task set is schedulable.
  - Otherwise, no conclusion can be made.

Examples

- **First-In, First-Out (FIFO): $U_{FIFO} = 0$**
- Earliest-Deadline-First (EDF): $U_{EDF} = 1$
- Rate-Monotonic Algorithm (RM):

  $U_{RM} = n (2^{(1/n)} - 1)$, where n = number of tasks

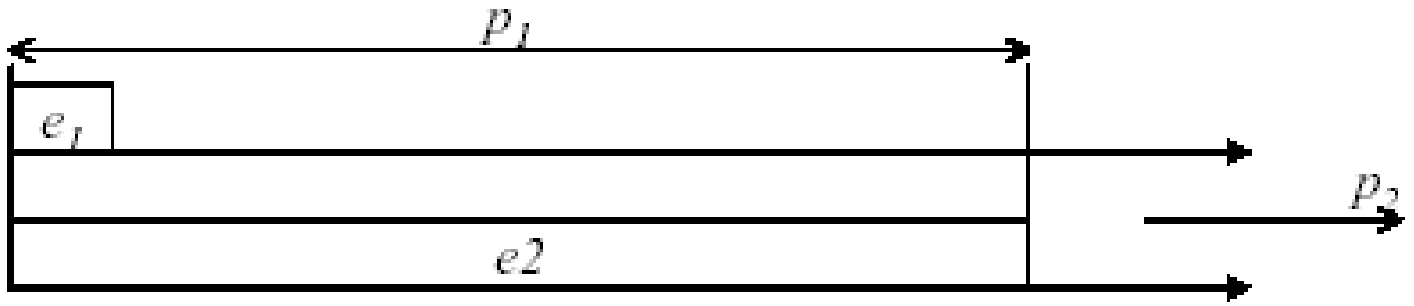
Theorem: $U_{FIFO} = 0$

Proof:

Given any utilization level $\varepsilon > 0$, we can find a task set, with utilization ε, which may not be feasibly scheduled according to FIFO.

Example task set:

$$\left.\begin{array}{lll} T_1 & : & e_1 = \dfrac{\varepsilon}{2} p_1 \\ T_2 & : & p_2 = \dfrac{2}{\varepsilon} p_1 \\ & & e_2 = p_1 \end{array}\right\} \Rightarrow U = \varepsilon$$



Examples

- First-In, First-Out (FIFO): $U_{FIFO} = 0$
- **Earliest-Deadline-First (EDF): $U_{EDF} = 1$**
- Rate-Monotonic Algorithm (RM):

  $U_{RM} = n (2^{(1/n)} - 1)$, where n = number of tasks

Examples

- First-In, First-Out (FIFO): $U_{FIFO} = 0$
- Earliest-Deadline-First (EDF): $U_{EDF} = 1$
- **Rate-Monotonic Algorithm (RM):**

$$\mathbf{U_{RM} = n (2^{(1/n)} - 1),}$$

where n = number of tasks

Deadline Monotonic Algorithm

Theorem 6-4: A system **T** of independent, preemptable periodic tasks that are in phase and have relative deadlines at most their respective periods can be feasibly scheduled on one processor according to the DM algorithm whenever it can be feasibly scheduled according to any fixed-priority algorithm.

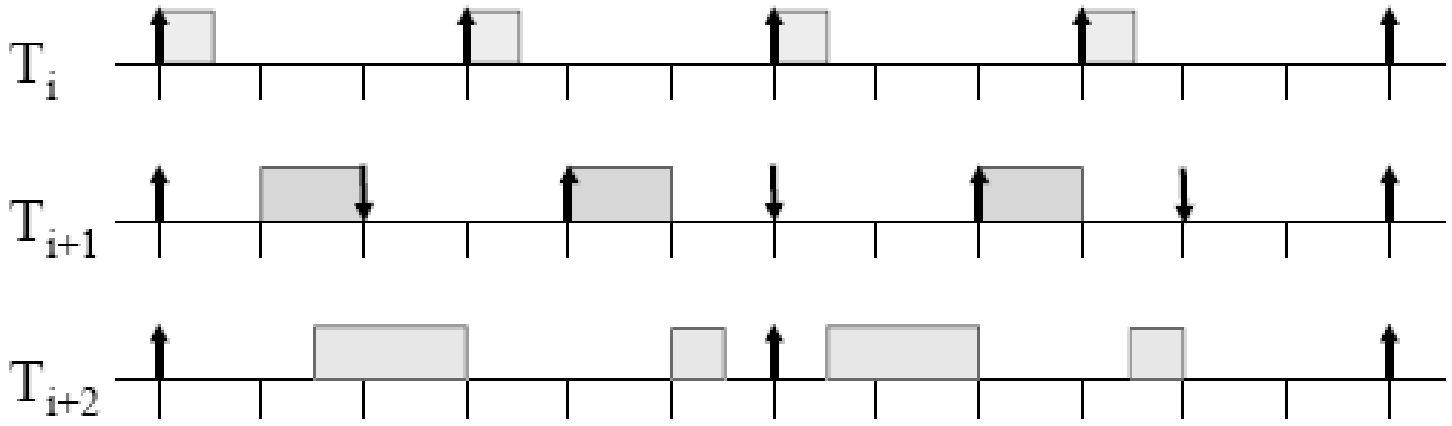
Corollary: The RM algorithm is optimal among all fixed-priority algorithms whenever the relative deadlines of all tasks are proportional to their periods.

Proof – Given a feasible static priority assignment

Suppose $T_1, \ldots, T_i$ are prioritized in accordance with DM.

Suppose T_i has a longer relative deadline than T_{i+1}, but T_i has a higher priority than T_{i+1}.

Then, we can interchange T_i and T_{i+1} and adjust the schedule accordingly by swapping "pieces" of T_i with "pieces" of T_{i+1}.

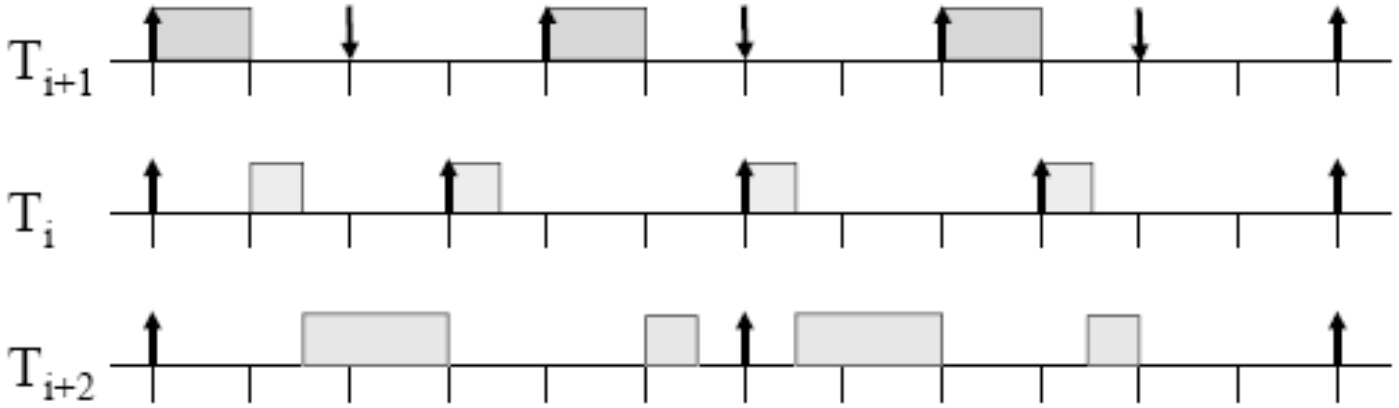


Proof (cont.)

Suppose $T_1, \ldots, T_i$ are prioritized in accordance with DM.

Suppose T_i has a longer relative deadline than T_{i+1}, but T_i has a higher priority than T_{i+1}.

Then, we can interchange T_i and T_{i+1} and adjust the schedule accordingly by swapping "pieces" of T_i with "pieces" of T_{i+1}.



By induction, we can correct all such situations.

Liu and Layland's Utilization-Based Test

Theorem 6-11: [Liu and Layland] A system of n independent, preemptable periodic tasks with relative deadlines equal to their respective periods can be feasibly scheduled on a processor according to the RM algorithm if its total utilization U is at most

$$U_{RM}(n) = n(2^{1/n} - 1)$$

Note that this is only a **sufficient** schedulability test.

Proof Sketch

- W.O.L.O.G., w.m.a. that all priorities are distinct, $p_1 < p_2 < .. < p_n$.
- First, we consider the case where $p_n \leq 2^* p_1$.
- Then, we remove that restriction.

Difficult-To-Schedule (DTS) System

Definition: A system is **difficult-to-schedule** if it is schedulable according to the RM algorithm, but it fully utilizes the processor for some interval of time so that any increase in the execution time or decrease in the period of some task will make the system unschedulable.

We seek the **most difficult-to-schedule** system, i.e., the system whose utilization is smallest among all difficult-to-schedule systems.

The proof for the special case $p_n \leq 2p_1$ consists of **four steps**, described next.

Proof Steps

- **Step 1:** Identify the phases in the most difficult-to-schedule system.
- **Step 2:** Define the periods and execution times for the most difficult-to-schedule system.
- **Step 3:** Show that any difficult-to-schedule system whose parameters are not like in Step 2 has utilization that is at least that of the most difficult-to-schedule system.
- **Step 4:** Compute an expression for $U_{RM}(n)$.

Critical Instant

Definition: A **critical instant** of a task T_i is a time instant such that:

(1) the job of T_i released at this instant has the maximum response time of all jobs in T_i, if the response time of every job of T_i is at most D_i, the relative deadline of T_i, and

(2) the response time of the job released at this instant is greater than D_i of the response time of some jobs in T_i exceeds D_i.

Informally, a critical instant of T_i represents a worst-case scenario from T_i's standpoint.

Theorem 6-5

Theorem 6-5: [Liu and Layland] In a fixed-priority system where every job completes before the next job of the same task is released, a critical instant of any task T_i occurs when one of its job $J_{i,c}$ is released at the same time with a job of every higher priority task.

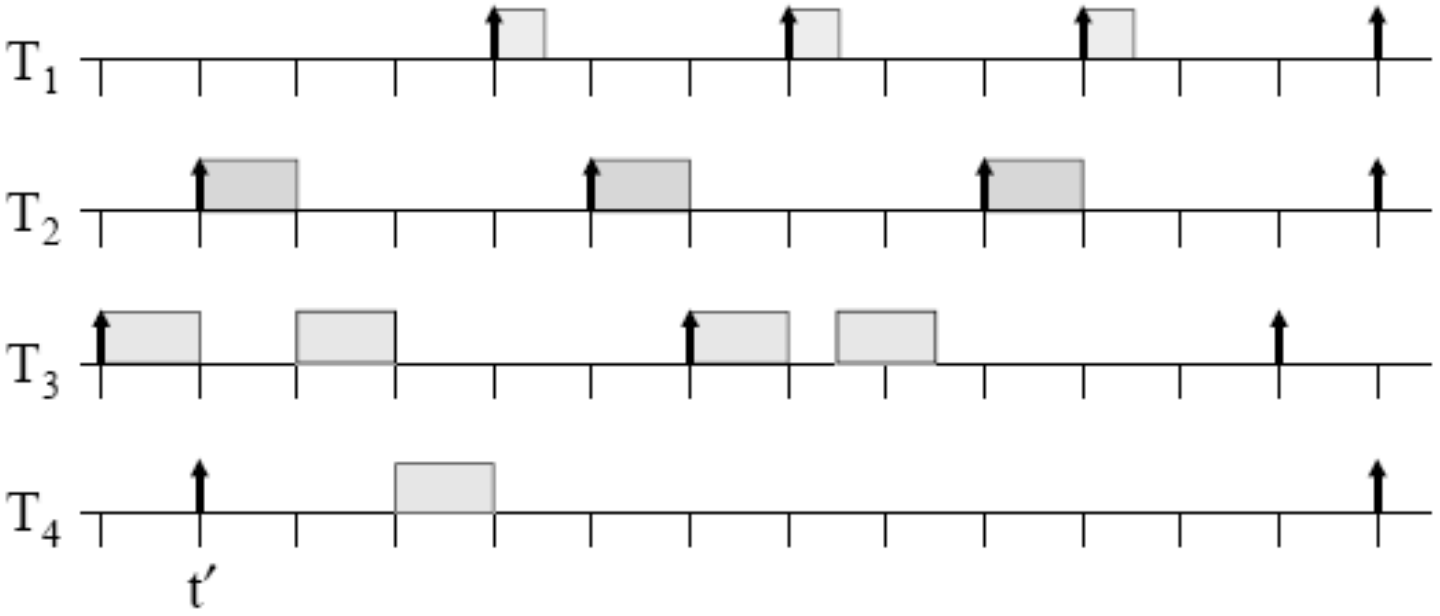
We are not saying that $T_1, \ldots, T_i$ will all necessarily release jobs at the same time, but if this does happen, we are claiming that the time of release will be a critical instant for T_i.

We give a different (probably more hand-waving) proof of Theorem 6-5 than that found in Liu.

Proof

Consider a system such that T_1, …, T_i all release jobs together at some time instant t. Suppose t is not a critical instant for T_i, i.e., T_i has a job released at another time t′ that has a longer response time than its job released at t.

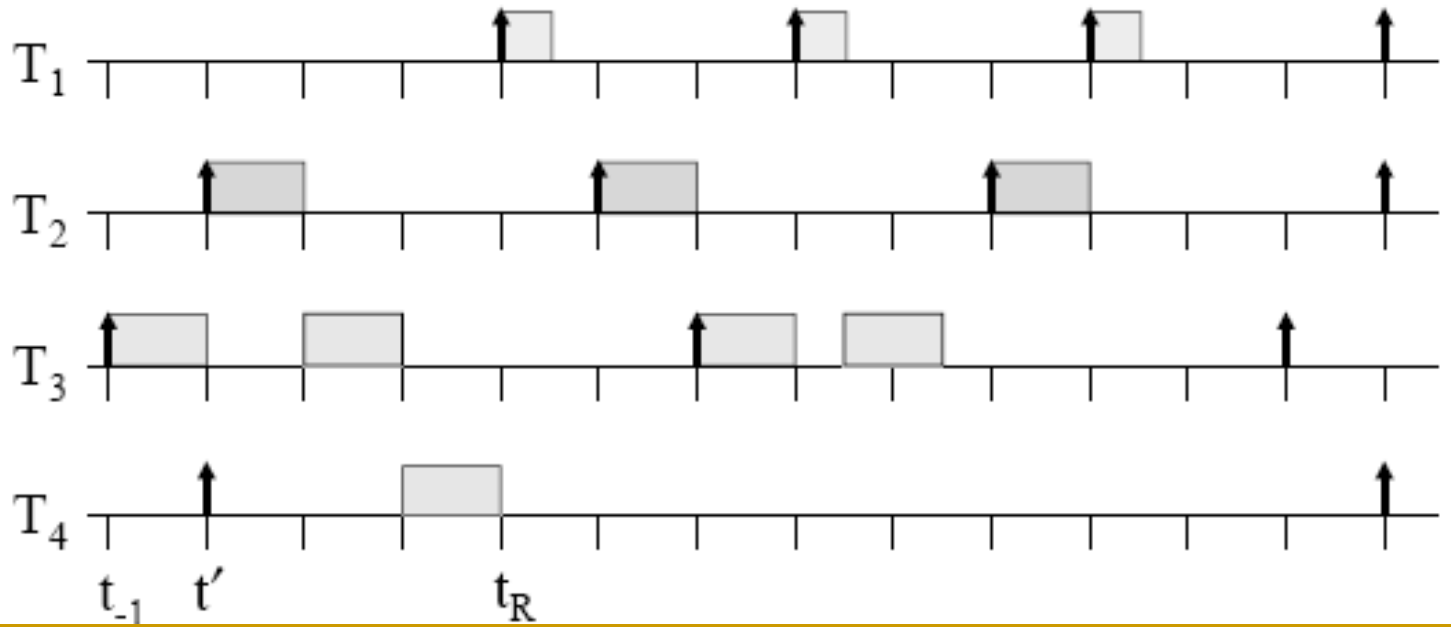
Example:



Proof (cont.)

Let t_{-1} be the latest "idle instant" for $T_1, \ldots, T_i$ at or before t'.
Let J be T_i's job released at t'.
Let t_R denote the time instant when J completes.

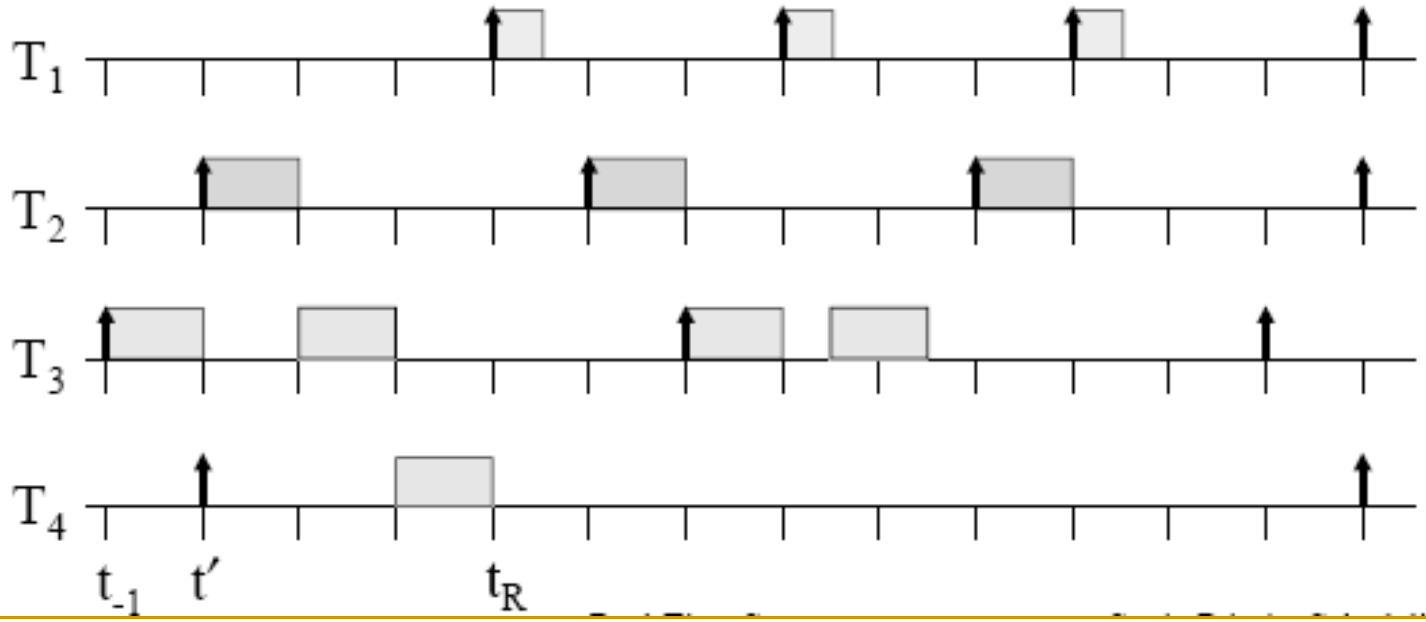
Example:



Proof (cont.)

If we (artificially) redefine J's release time to be t_{-1}, then t_R remains unchanged (but J's response time may increase).

Example:

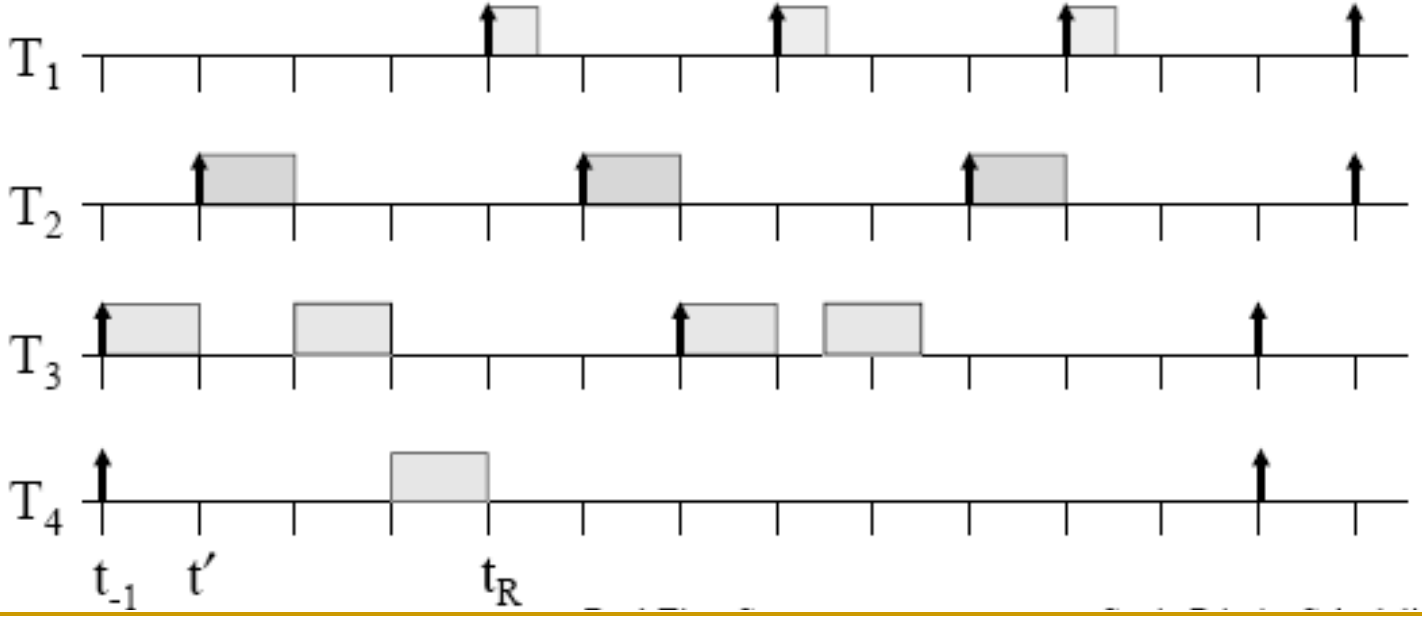


Proof (cont.)

Starting with T_1, let us "left-shift" any task whose first job is released after t_{-1} so that its first job is released at t_{-1}.

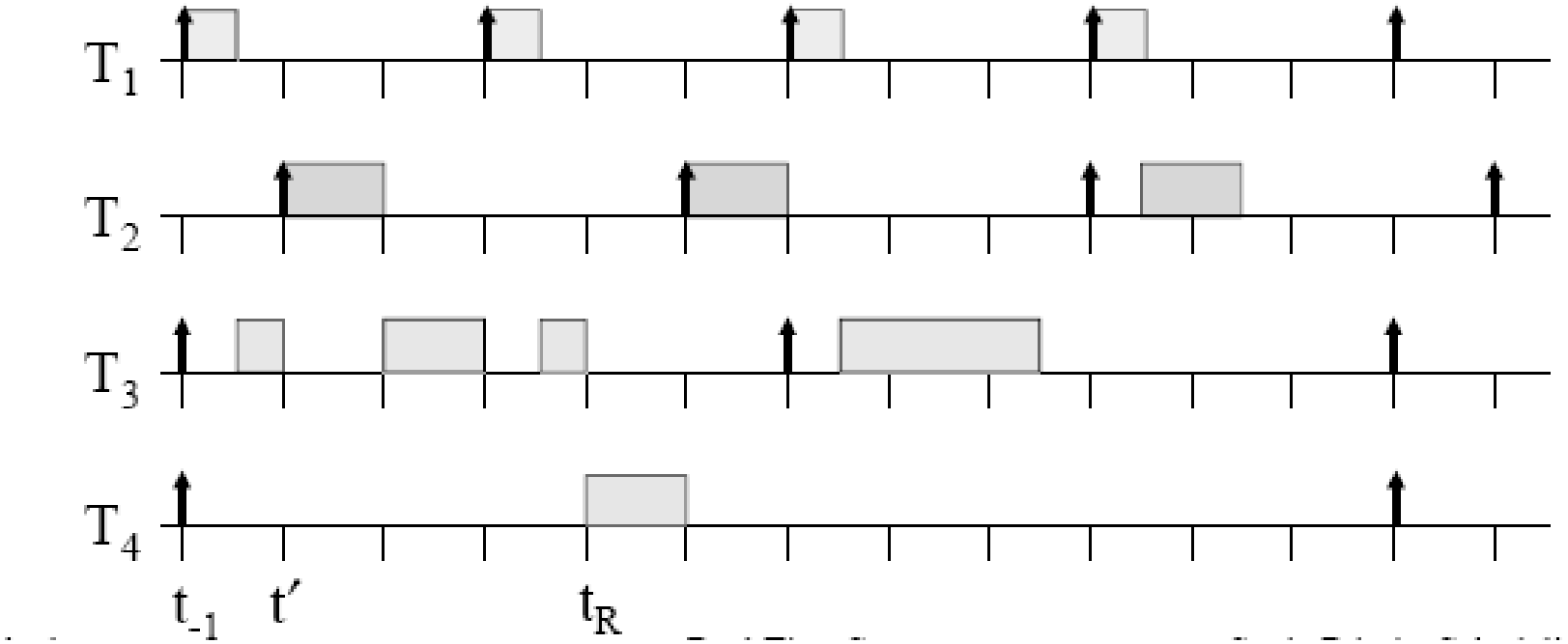
With each shift, T_i's response time does not decrease. **Why?**

Example: **Shift over T_1 ...**



Proof (cont.)

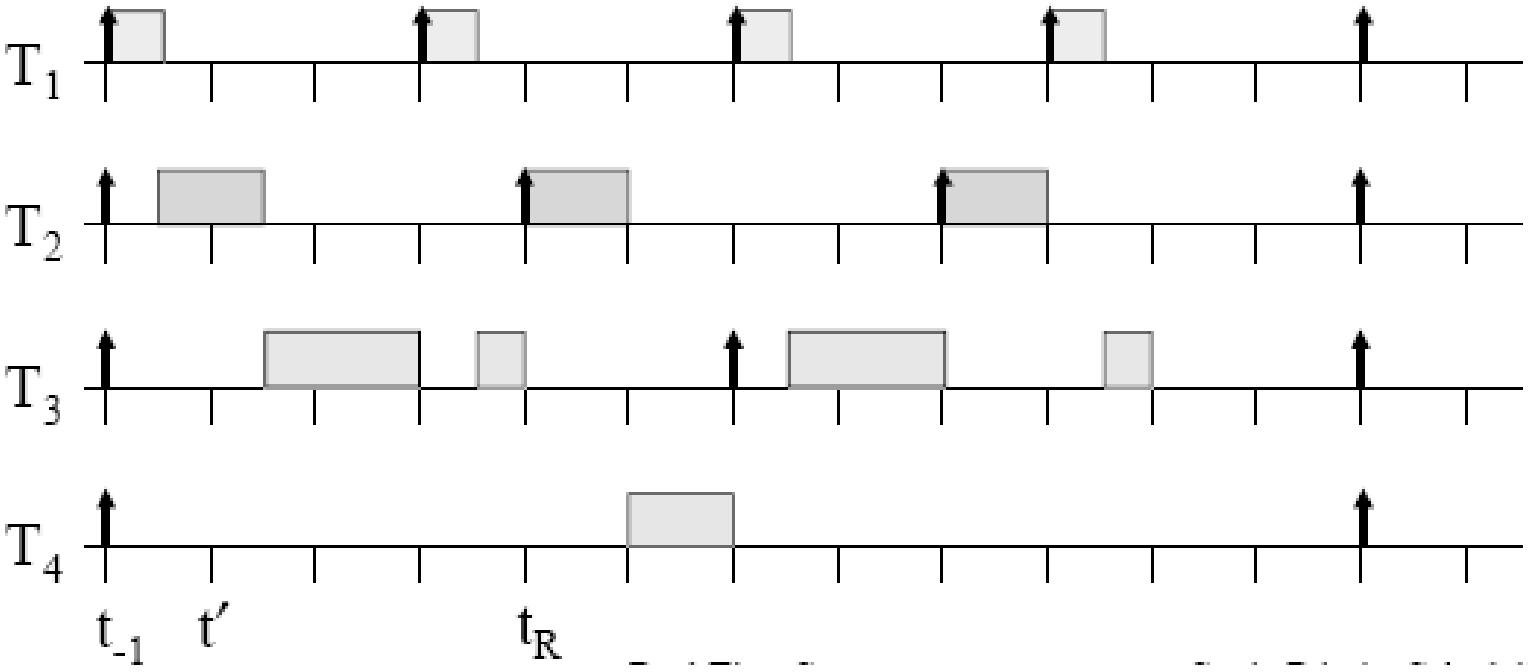
Example: **Shift over T_2 ...**



Proof (cont.)

With each shift, T_i's response time does not decrease. **Why?**

Example:



Proof (cont.)

We have constructed a portion of the schedule that is identical to that which occurs at time t (when T_1, …, T_i all release jobs together).

Moreover, the response time of T_i's job released at t is at least that of T_i's job released at t'.

This contradicts our assumption that T_i's job released at t' has a longer response time than T_i's job released at t.

Thus, t is a critical instant.

Proof of Theorem 6-11 – Step 1

- Back to the proof of Theorem 6-11…
- Recall that Step 1 is to identify the phases in the most difficult-to-schedule system.
- By Theorem 6-5, we can assume that each task in the most difficult-to-schedule system releases its first job at time 0.

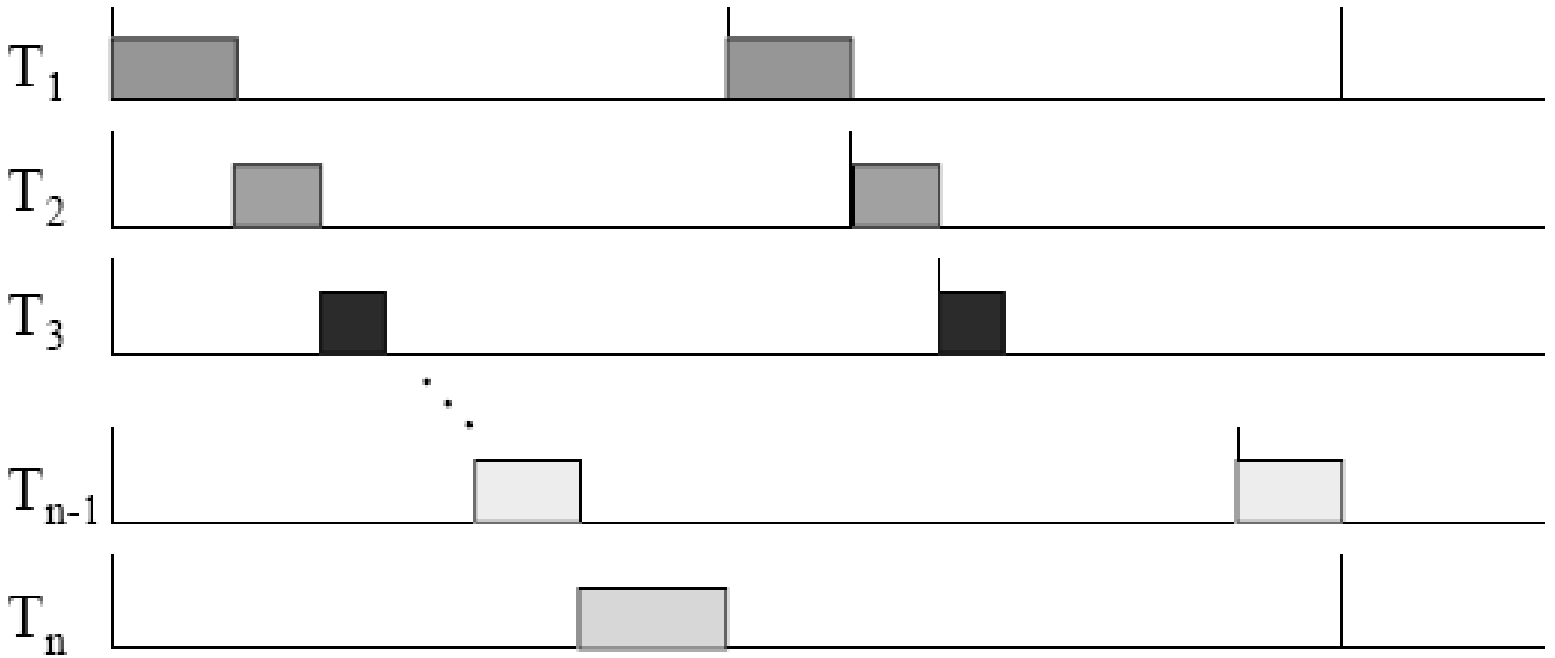
Step 2

- By Theorem 6-5, we can limit attention to the first period of each task.
- We need to make sure that each task's first job completes by the end of its first period.
- We will define the system's parameters so that the tasks keep the processor busy from time 0 until at least p_n, the end of the first period of the lowest-priority task.

Step 2 (cont.)

Let us define

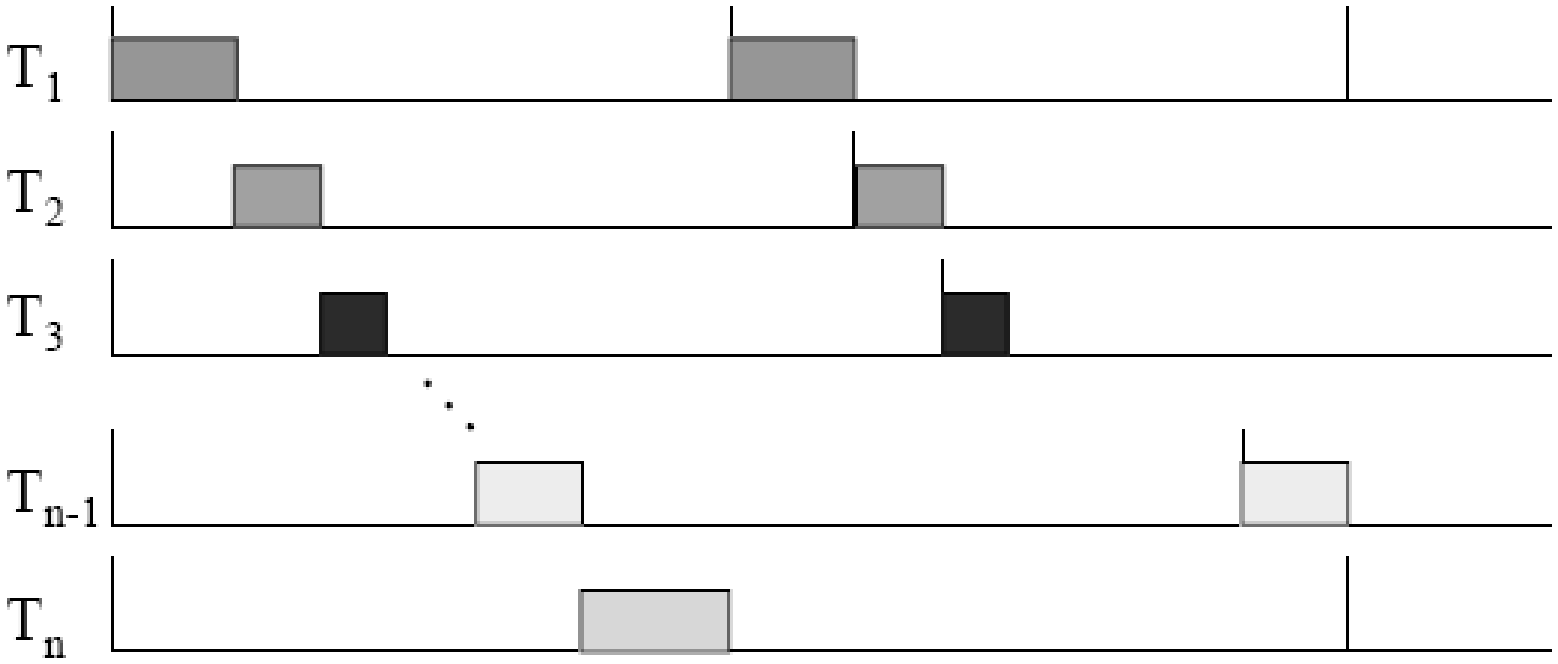
$$e_k = p_{k+1} - p_k \quad \text{for } k = 1, 2, \ldots, n-1$$
$$e_n = p_n - 2\sum_{k=1,\ldots,n-1} e_k.$$



Step 2 (cont.)

Notes:

- This task system is difficult-to-schedule. **(Why?)**
- The processor is fully utilized up to p_n.



Step 3 – Showing it's the most D-T-S

We still need to show that the system from Step 2 is the *most* difficult-to-schedule system.

We must show that other difficult-to-schedule systems have equal or higher utilization.

Other difficult-to-schedule systems can be obtained from the one in Step 2 by systematically increasing or decreasing the execution times of some of the tasks.

We show that any small increase or decrease results in a utilization that's at least as big as that of the original task system.

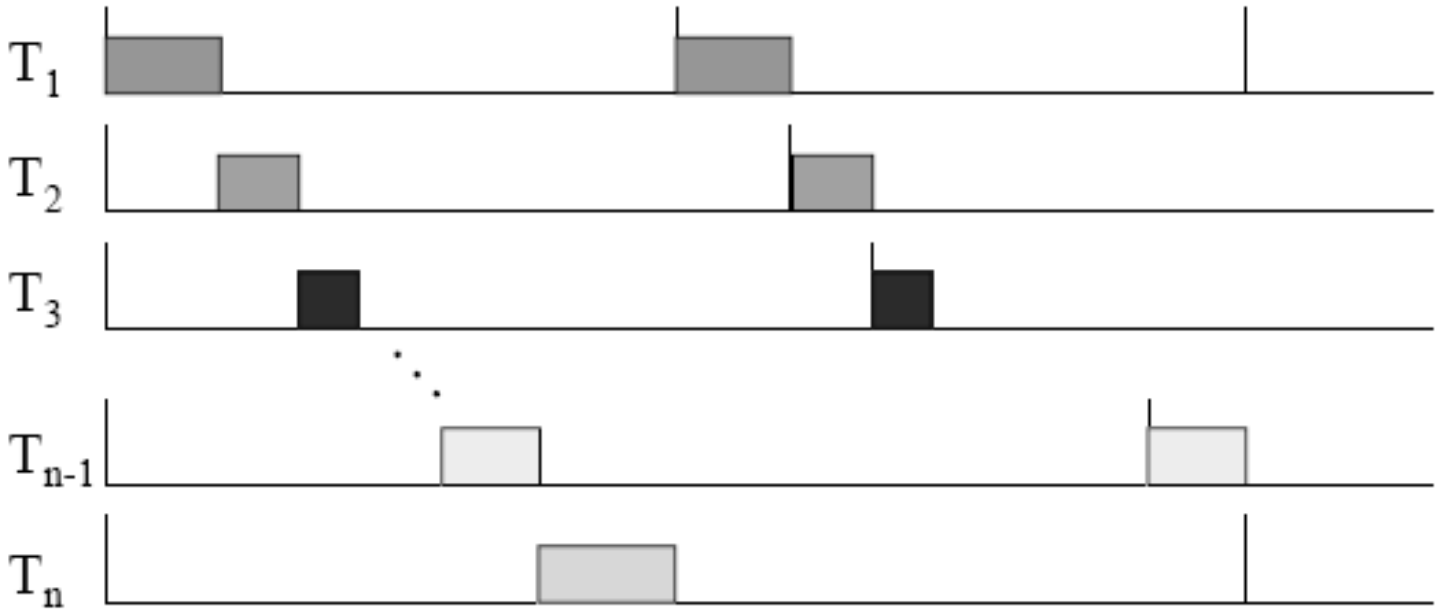
Step 3 (cont.)

Let's **increase** the execution of some task, say T_1, by ε, i.e.,

$$e'_1 = p_2 - p_1 + \varepsilon = e_1 + \varepsilon.$$

We can keep the processor busy until p_n by decreasing some T_k's, $k \neq 1$, execution time by ε:

$$e'_k = e_k - \varepsilon.$$

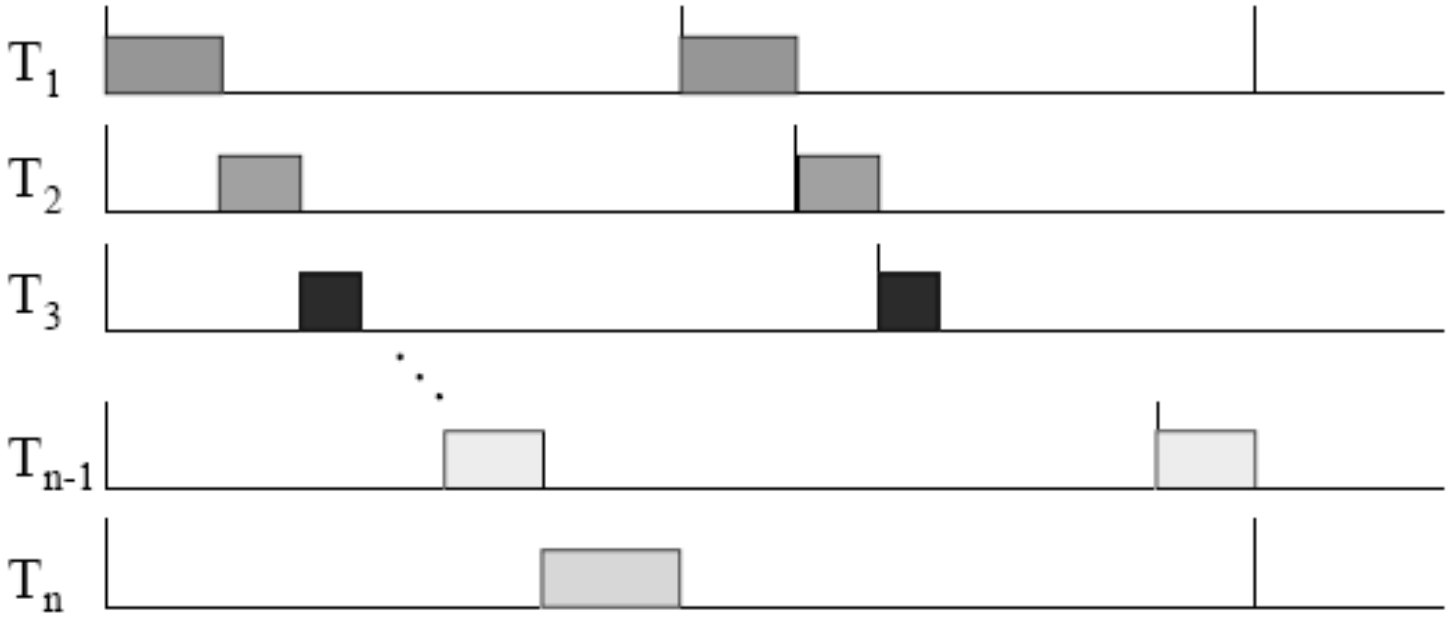


Step 3 (cont.)

The **difference in utilization** is: $U' - U = \frac{e_1'}{p_1} + \frac{e_k'}{p_k} - \frac{e_1}{p_1} - \frac{e_k}{p_k}$

$$= \frac{\varepsilon}{p_1} - \frac{\varepsilon}{p_k}$$

$$> 0 \quad \text{since } p_1 < p_k$$



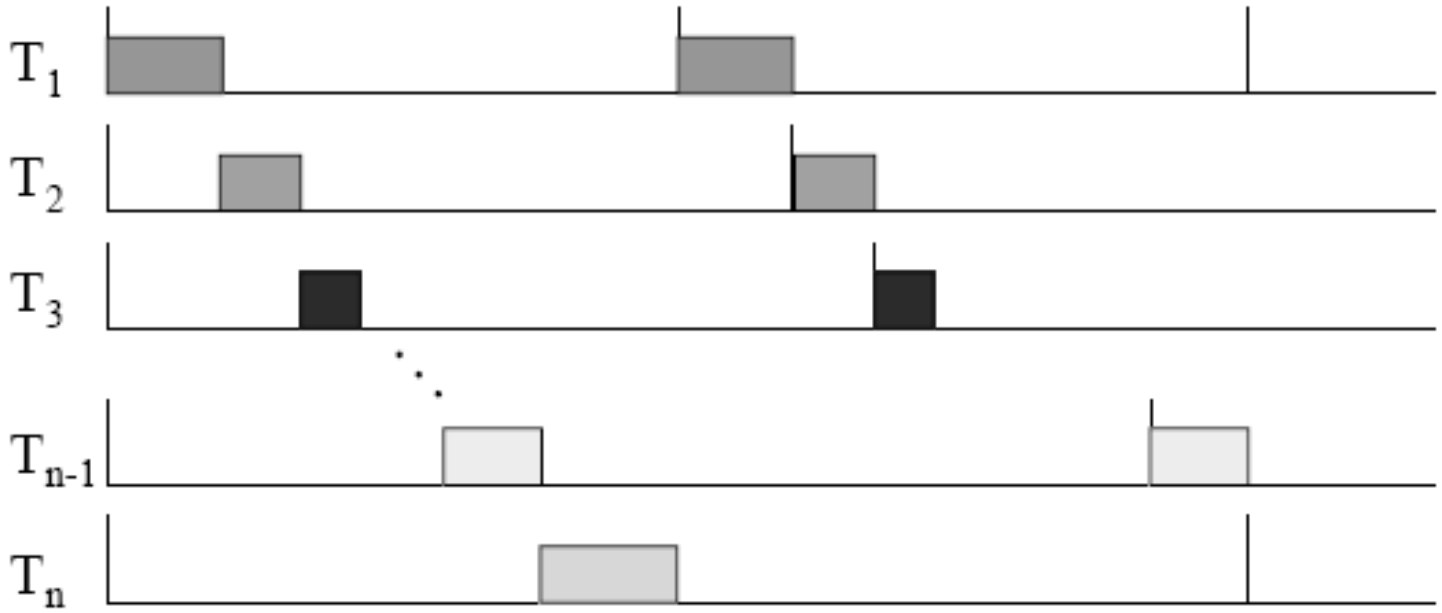
Step 3 (cont.)

Let's **decrease** the execution of some task, say T_1, by ε, i.e.,

$$e''_1 = p_2 - p_1 - \varepsilon.$$

We can keep the processor busy until p_n by increasing some T_k's, $k \neq 1$, execution time by 2ε:

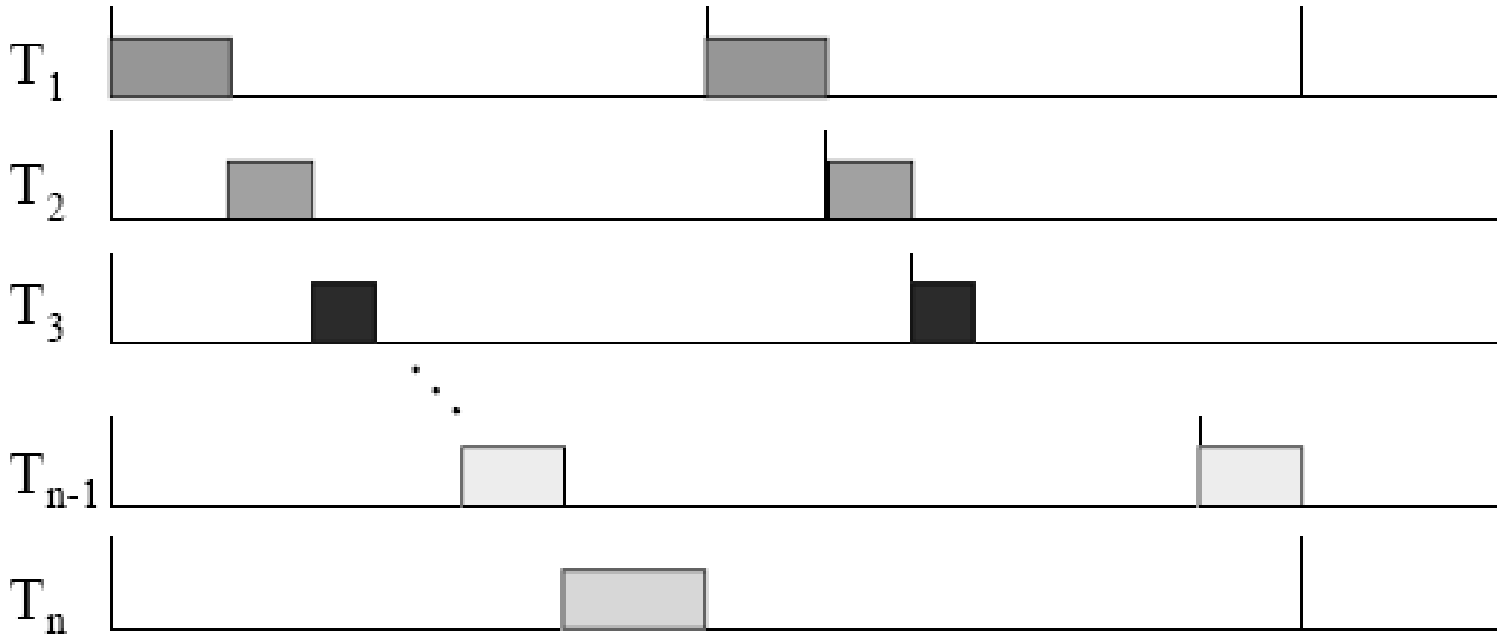
$$e''_k = e_k + 2\varepsilon.$$



Step 3 (cont.)

The **difference in utilization** is: $U'' - U = \frac{2\varepsilon}{p_k} - \frac{\varepsilon}{p_1}$

≥ 0 since $p_k \leq 2p_1$



Step 4 – Compute $U_{RM}(n)$

Let $U(n) = \sum_{k=1}^{n} \frac{e_k}{p_k}$ denote the utilization of the system in Step 2.

Define $q_{k,i} = p_k / p_i$. Then,

$$U(n) = q_{2,1} + q_{3,2} + \cdots + q_{n,(n-1)} + \frac{2}{q_{2,1} q_{3,2} \cdots q_{n,(n-1)}} - n.$$

To find the minimum, we take the partial derivative of U(n) with respect to each adjacent period ratio $q_{k+1,k}$ and set the derivative to zero. This gives us the following n - 1 equations

$$1 - \frac{2}{q_{2,1} q_{3,2} \cdots q_{(k+1),k}^{2} \cdots q_{n,(n-1)}} = 0 \quad \text{for all } k = 1, 2, \ldots, n-1.$$

Solving these equations for $q_{(k+1),k}$, we find that U(n) is at its minimum when all the $n-1$ adjacent period ratios $q_{k+1,k}$ are equal to $2^{1/n}$. Thus,

$$U(n) = n(2^{1/n} - 1).$$

Removing the period ratio restriction

Definition: The ratio $q_{n,1} = p_n/p_1$ is the **period ratio** of the system.

We have proven Theorem 6-11 only for systems with period ratios of at most 2.

To deal with systems with period ratios larger than 2, we show the following

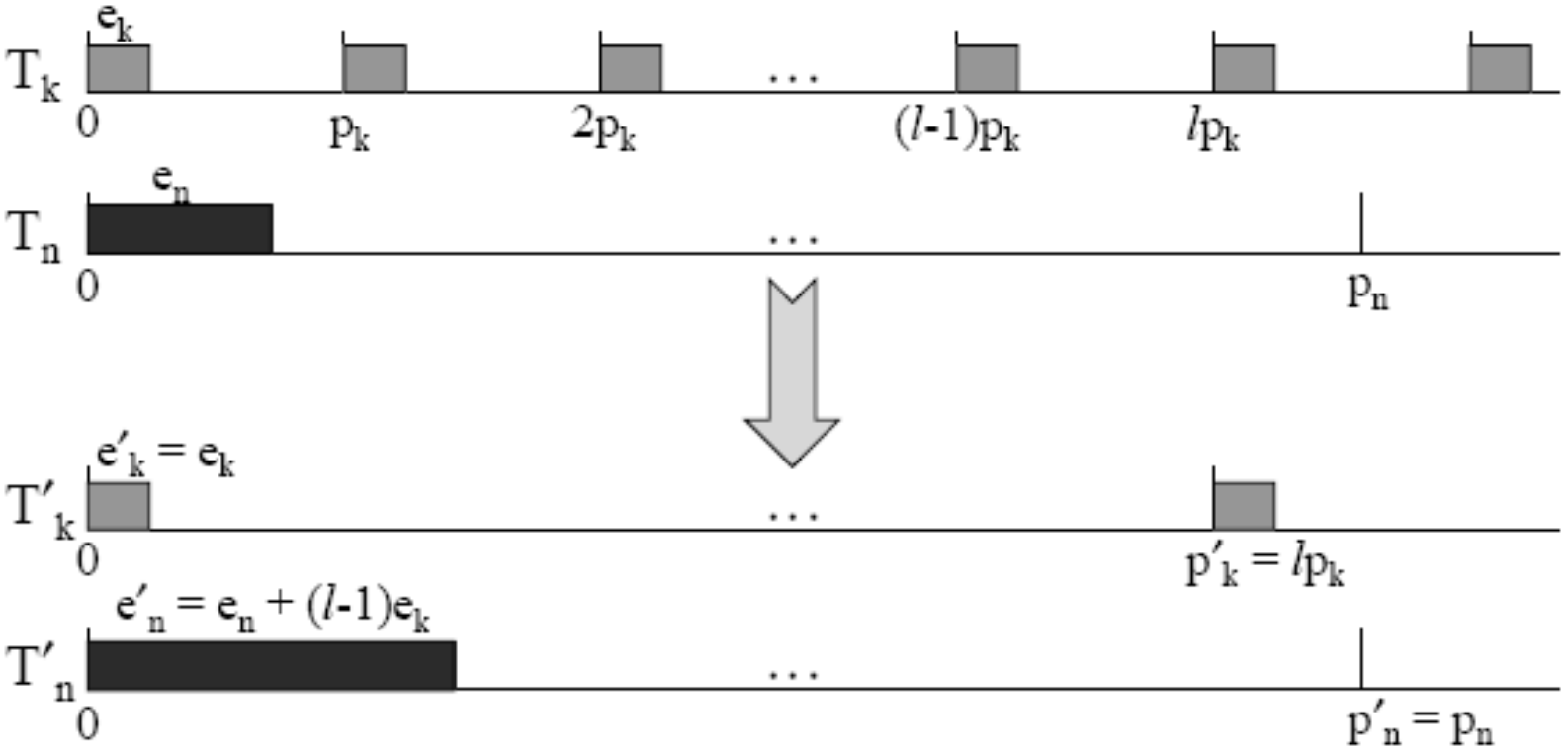
(1) Corresponding to every difficult-to-schedule n-task system **T** whose period ratio is larger than 2 there is a difficult-to-schedule n-task system **T′** whose period ratio is at most 2, and

(2) **T**'s utilization is at least **T′**'s.

Proof of (1)

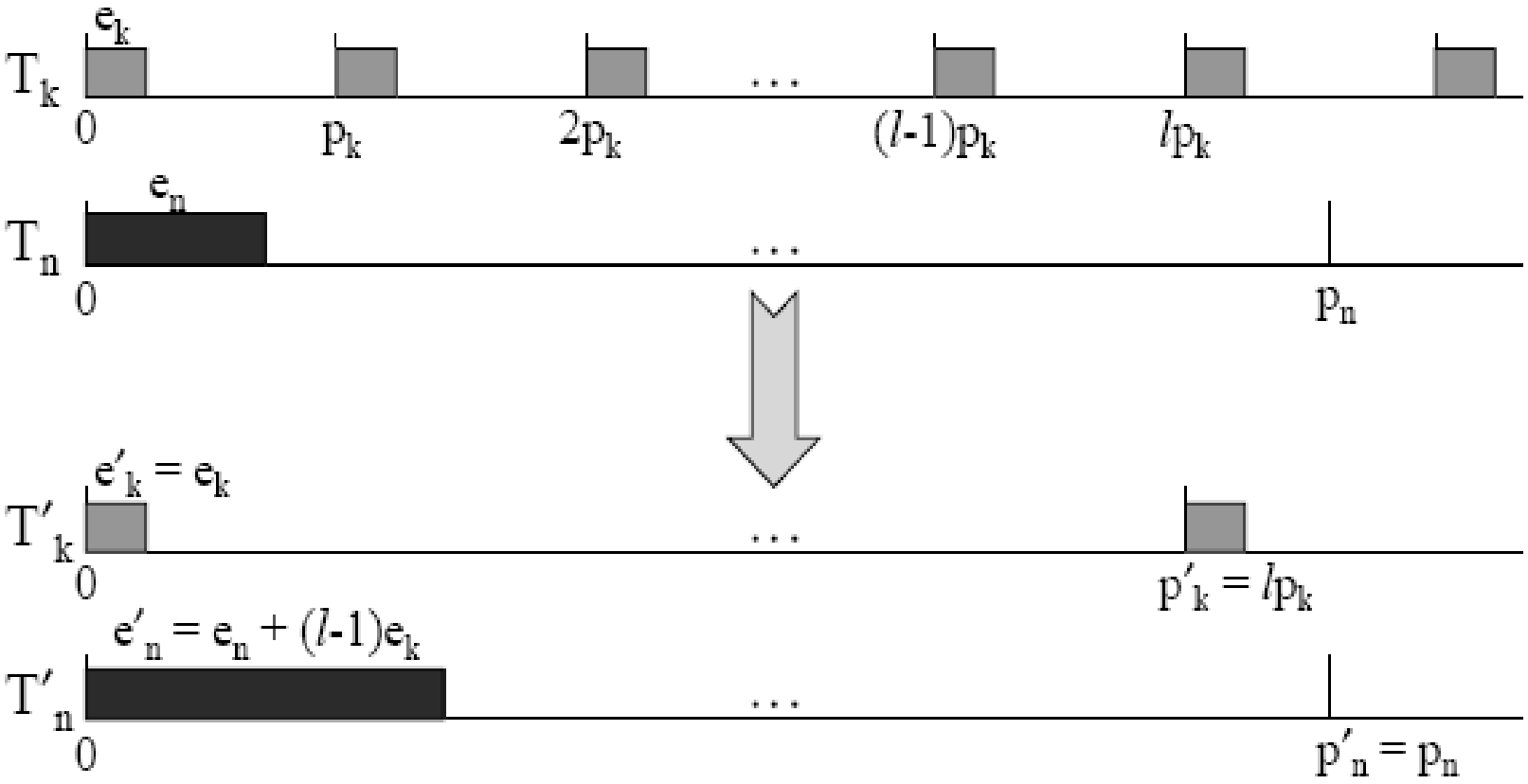
We show we can transform **T** step-by-step to get **T′**.

At each step, we find a task T_k whose period is such that $lp_k < p_n \leq (l+1)p_k$, where l is an integer that is at least 2. We modify (only) T_k and T_n as follows.



Proof of (1)

- Clearly it follows that:
  - The resulting system is difficult-to-schedule, and
  - We will eventually get a system with a period ratio of at most 2.



Proof of (2)

It suffices to look at the difference between the utilization of the old and new system when one of the steps in the proof of (1) is applied.

This difference is:

$$\frac{e_k}{p_k} - \frac{e_k}{lp_k} - \frac{(l-1)e_k}{p_n}$$

$$= \left(\frac{1}{lp_k} - \frac{1}{p_n} \right) (l-1)e_k$$

$$> 0 \qquad \text{because } lp_k < p_n.$$

This concludes the proof of (2) and (finally!) the proof of Theorem 6-11.

Summary

- Read Ch. 4-7 + Liu and Layland's, and Devi's papers.
- Homework #1.