Task Scheduling Algorithms

Types of Tasks

- Real-time systems software is typically designed using distinct deliverables known as tasks or threads
- Fabrication of a schedule to determine whether the system can meet the deadline requirements
- This estimation can be very difficult, making it important to ensure reserve system capacity and further to prevent critical system failure
- Types of tasks:
  - periodic – repetitive with *hard* deadlines
  - aperiodic – asynchronous with *soft* deadlines
  - sporadic – asynchronous with *hard* deadlines

Types of Task Scheduling

- The array of categories of scheduling problems is rather broad:
  - aperiodic or periodic
  - preemptive or non-preemptive
  - precedence constraints or no constraints
  - synchronization among tasks or no synchronization
  - static or dynamic scheduling
  - deterministic or non- deterministic scheduling
- Many (most?) task scheduling problems are **NP**-hard
- Intuitive heuristic algorithms sometimes lead to unexpected and seeming paradoxical results

Aperiodic Task Scheduling

- Establish analysis techniques extended later to the more practical problem of scheduling periodic tasks
- The real time system considered here is assumed to have a fixed number of tasks
- Execute each task a single time - cost function is the overall execution time of the entire task set

Minimize the time T required to execute N tasks on M processors

or

Determine if the set of N tasks on M processors can be completed before a deadline D

Aperiodic - Non-Preemptive Scheduling With No Precedence Constraints

This "simple" scheduling problem Is

- NP-hard for **M = 2**,but with pseudo-polynomial time solution
- NP-hard for **M > 2**.

(If there are **N** tasks to be scheduled, the number of orderings-and thus the number of possible schedules is **N!** If **N = 20**,and if one candidate schedule could be checked In one *microsecond,* It would take *70,000* years to check all 201 schedules!)

An *heuristic* algorithm for solution of this problem Is the **Largest Processing Time** algorithm:

Whenever a processor is available, assign to it the available task with the largest execution time.

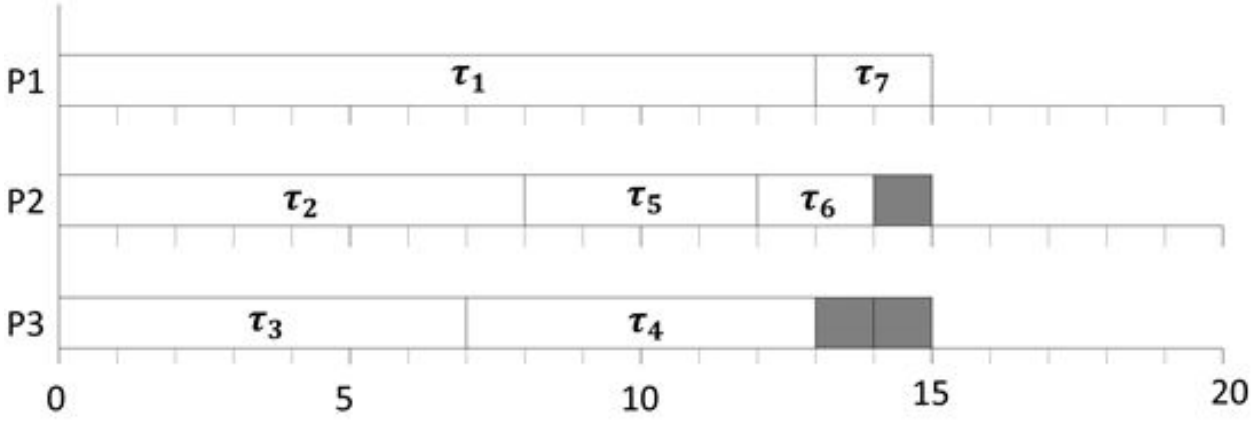
Aperiodic - Non-Preemptive Scheduling With No Precedence Constraints

The problem statement includes:

- **M**, the number of processors
- **N**, the number of tasks
- e_n, the execution time for each task.

Example: $M = 3 \quad e_1 = 13, \quad e_2 = 8, \quad e_3 = 7, \quad e_4 = 6, \quad e_5 = 4$

$N = 7 \quad e_6 = 2 \quad e_7 = 2$

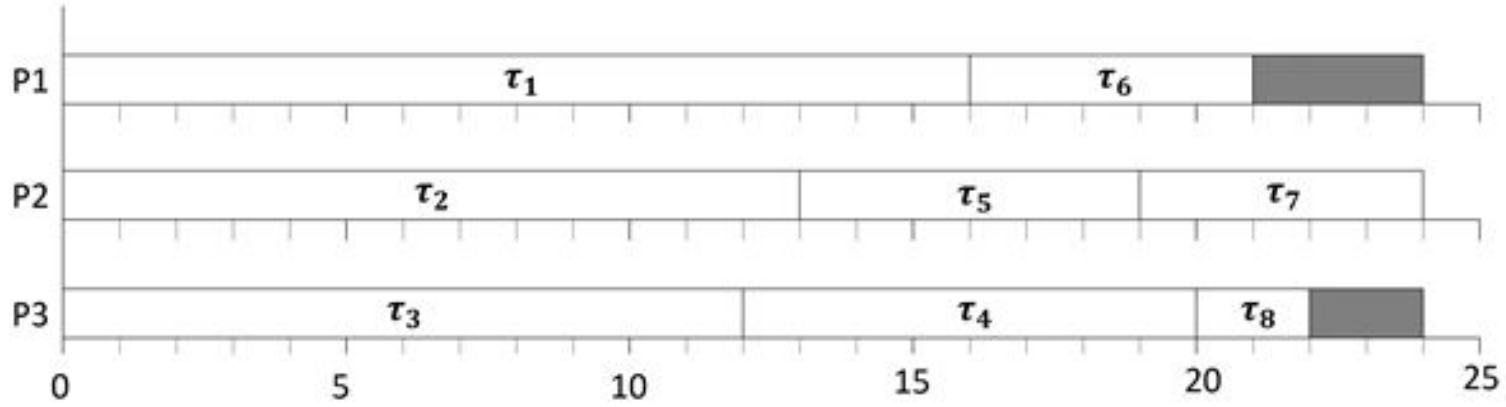


In this case an optimal schedule is produced!

Aperiodic - Non-Preemptive Scheduling With No Precedence Constraints

A second example of **Largest Processing Time** algorithm:

$$M = 3 \quad e_1 = 16, \quad e_2 = 13, \quad e_3 = 12\,, \quad e_4 = 8, \quad e_5 = 6$$
$$N = 8 \quad e_6 = 6\,, \quad e_7 = 5, \; e_8 = 2$$

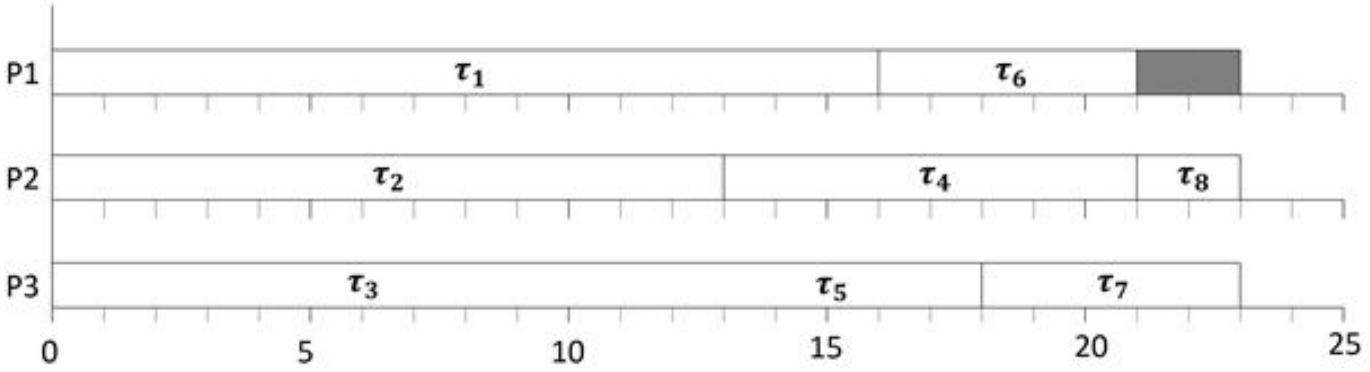


Here, T=24, but this is *not optimum*.

Aperiodic - Non-Preemptive Scheduling With No Precedence Constraints

Second example with reordered tasks:

$$M = 3 \quad e_1 = 16, \quad e_2 = 13, \quad e_3 = 12\,, \quad e_4 = 8, \quad e_5 = 6$$
$$N = 8 \quad e_6 = 5\,, \quad e_7 = 5, \; e_8 = 2$$



min T=23, this is *optimum*.

The problem with heuristic algorithms is that they are unpredictable

Aperiodic - Non-Preemptive Scheduling With No Precedence Constraints

The Largest Processing Time algorithm has a performance bound of

$$\frac{T}{minT} \leq \frac{4}{3} - \frac{1}{3M}$$

In this case $\boldsymbol{M = 3}$, this yields

$$\frac{T}{minT} \leq \frac{11}{9}$$

And the example problems , of course, meet this bound. The worst case occurs for large **M**

$$\frac{T}{minT} \leq \frac{4}{3}$$

Aperiodic - Non-Preemptive With Precedence Constraints

- Precedence constraints occur when one task or subtask *must* complete before another can begin execution.

- The problem statement then includes

  - **M**, the number of processors
  - **N**, the number of tasks
  - e_n, the execution time for each task
  - precedence constraint information for each task or subtask.

- This problem Is **NP**-hard In the strong sense for **M > 2**

Aperiodic - Non-Preemptive With Precedence Constraints

- Priority List Scheduling Algorithm:
  - There is an *a priori* priority list $\{\boldsymbol{\tau_1}, \boldsymbol{\tau_2}, \cdots, \boldsymbol{\tau_N}\}$ This list *is not* required to be consistent with the precedence constraints
  - Precedence constraints are specified in the form $\boldsymbol{\tau_i} \rightarrow \boldsymbol{\tau_j}$ (task $\boldsymbol{\tau_i}$ must complete before task $\boldsymbol{\tau_j}$ can start)
  - At any time **t** that a processor is available, the scheduler scans the priority and selects the highest priority task for which precedence constraints have been satisfied and assigns the available processor to that task
  - If two or more processors compete for a task, the tie is broken(arbitrarily) by assigning the task to the lowest indexed processor

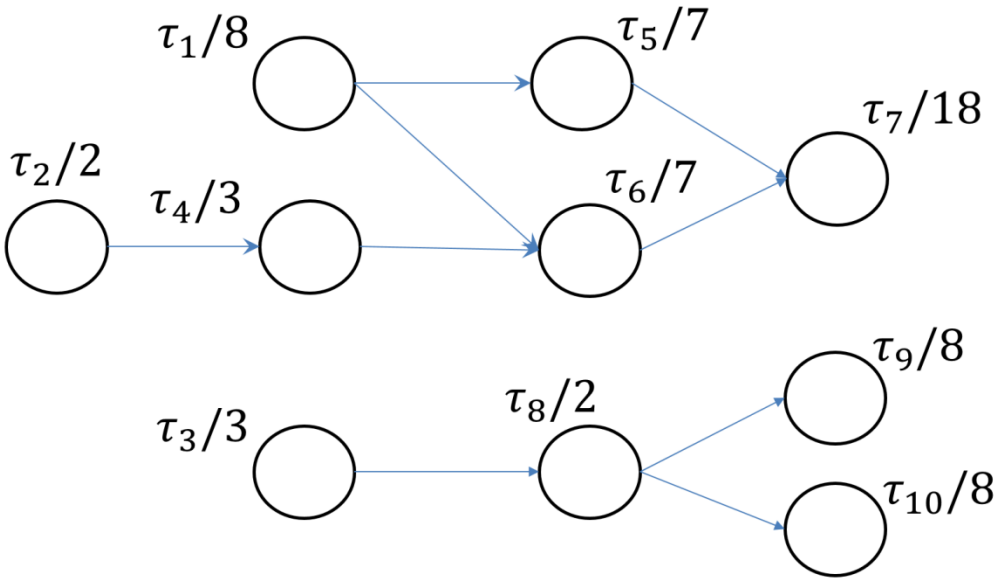
- If no task is available for a vacant processor, it idles

Aperiodic - Non-Preemptive With Precedence Constraints

Priority List Scheduling example:

$M = 2 \quad e_1 = 8, \quad e_2 = 2, \quad e_3 = 3\,, \quad e_4 = 3, \quad e_5 = 7$

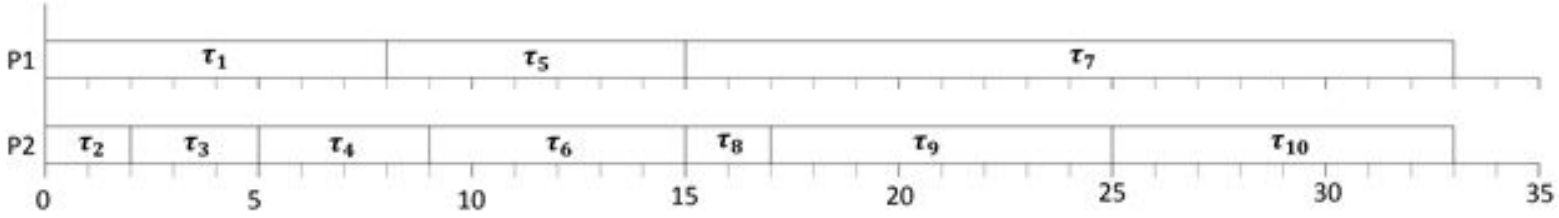
$N = 10 \quad e_6 = 7\,, \quad e_7 = 18, \; e_8 = 2\,, \quad e_9 = 8, \; e_{10} = 8$



Priority List = $\{\boldsymbol{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7, \tau_8, \tau_9, \tau_{10}}\}$

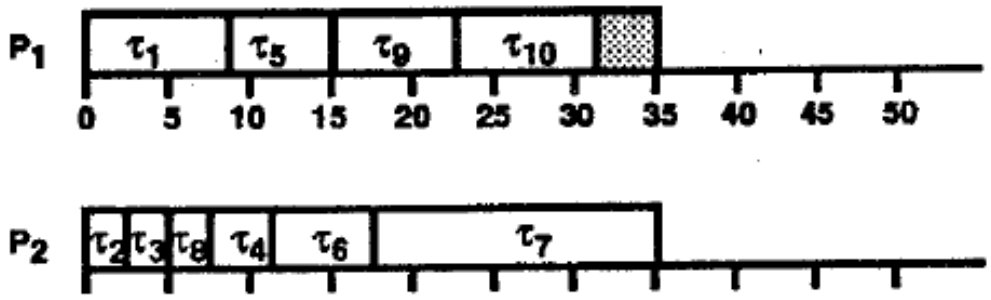
Aperiodic - Non-Preemptive With Precedence Constraints

The Priority List Scheduling is an heuristic algorithm. *It is subject to anomalous behavior.*



This schedule is obviously optimum, with **T = 33**

Change Priority List = $\{\tau_1, \tau_2, \tau_3, \tau_8, \tau_4, \tau_5, \tau_6, \tau_7, \tau_9, \tau_{10}\}$



T = 35, suboptimal

Aperiodic - Non-Preemptive With Precedence Constraints

- Decreasing execution times of tasks or increasing the number of processors is NOT guaranteed to produce a more optimal schedule using the Priority List Scheduling Algorithm

- Two characteristics of the Priority List Scheduling algorithm are responsible for such anomalous behavior:
  - No processor Idle time can be inserted to aid in scheduling
  - No preemption is allowed

Aperiodic - Non-Preemptive With Precedence Constraints

- There is a bound on these kinds of anomalous effects when Priority List Scheduling is used:
  - If a schedule is developed for a task system on **M** processors with a given set of execution times, precedence constraints, and task priorities, and producing schedule time **T** ; and a second schedule is developed on **M'** processors with a set of reduced execution times, a weaker set of precedence constraints, and a different priority list, and producing schedule time **T'**, then

$$\frac{T'}{T} \leq 1 + \frac{(M-1)}{M'}$$

  - This is a strange bound, in essence limiting the *damage* that can be done by *lessening* the systems requirements.
  - When **M = M'**, the bound is $\frac{T\prime}{T} \leq 2 - \frac{1}{M}$

Aperiodic - Non-Preemptive With Precedence Constraints

- One possible ordering of the priority list is in *decreasing order of task execution times. (recall the Largest Processing Time algorithm)*
- This process is termed *Decreasing Priority List Scheduling*.
  - When precedence constraints are present, this algorithm *can create the worst possible schedule* bounded by the factor

    $$2 - \frac{1}{M}$$

    (precedence constraints can prohibit tasks from executing in the order specified by the priority list)
  - When there are no precedence constraints, this algorithm exhibits a performance bound of

    $$\frac{4}{3} - \frac{1}{3M}$$ which is substantially better than $2 - \frac{1}{M}$
- This bound implies that the schedule time produced by Decreasing Priority List scheduling will never exceed the optimum value by more than a factor of 1/3

Aperiodic - Preemptive With No Precedence Constraints

- When there are no precedence constraints (or synchronization) this problem is solvable in polynomial time - that is, it is a *tractable problem*

- An algorithm for producing an optimum schedule:
  - Given the task system $\{\boldsymbol{\tau_1}, \boldsymbol{\tau_2}, \cdots, \boldsymbol{\tau_N}\}$ in priority order of decreasing execution times
  - Compute the minimum schedule length **T** from the expression

$$T = max\left\{max\{e_n\}, \frac{1}{M}\sum_{n=1}^{N} e_n\right\}$$

  - Starting with the first processor, assign tasks by the priority order. If the duration of the task exceeds **T**, assign the remaining time to the next processor.
  - Continue until all tasks have been assigned

Aperiodic - Preemptive With No Precedence Constraints

An example: Case 1: $N = number\ of\ tasks = 5$

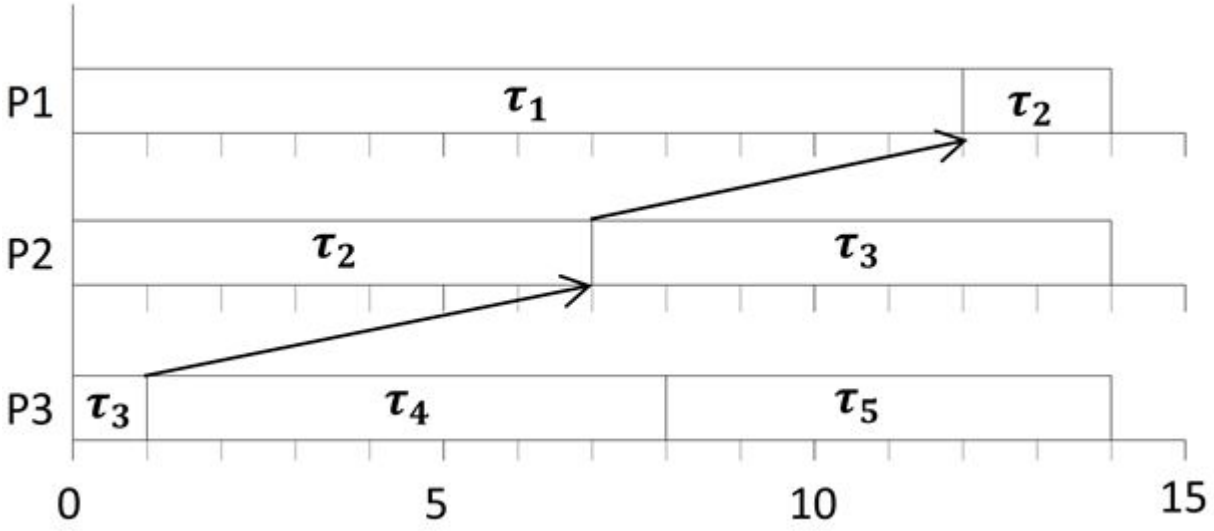
$$M = number\ of\ processors = 3$$

$$Precedence\ constraints - none$$

$$e_1 = 12, \quad e_2 = 9, \quad e_3 = 8, \quad e_4 = 7, \quad e_5 = 6$$

Calculate **T** = max {12,14} = 14

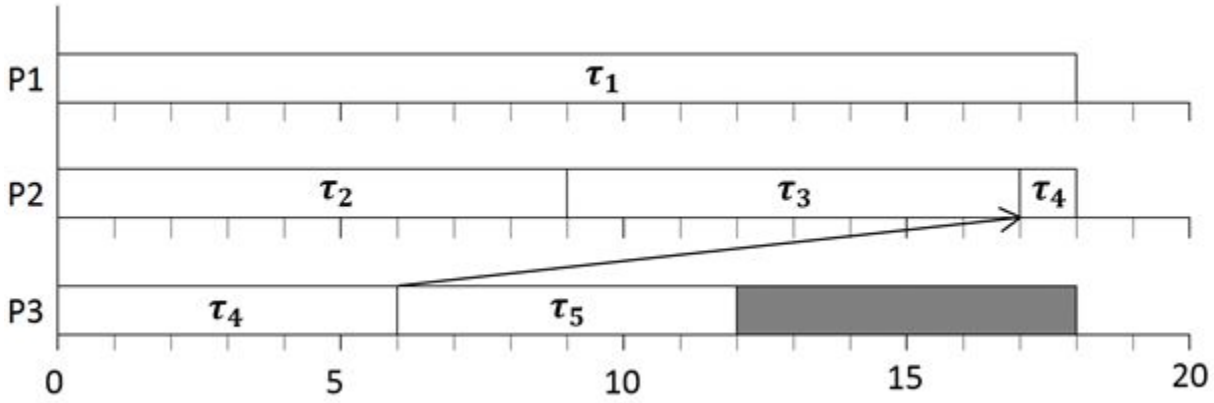
Processor utilization will be always 100% - if $max\{e_n\} < \frac{1}{M}\sum_{n=1}^{N} e_n$



Aperiodic - Preemptive With No Precedence Constraints

Example, Case 2: Increase $\mathbf{e_1}$ to 18

Calculate T =max { 18,16} = 18



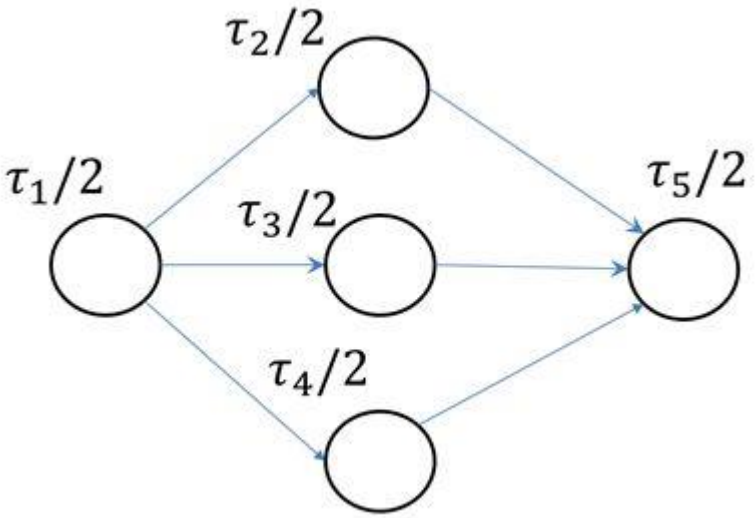
Processor utilization = 89%

Aperiodic – Processor Sharing

- A conceptual view - a processor, rather than being considered a discrete unit, is treated as though it comprises a unit of processing capability to be divided into arbitrary fractions
- Processor sharing is a useful concept when dealing with precedence constraints. A conceptual processor sharing schedule can be converted to a realizable preemptive schedule
- Simple Example

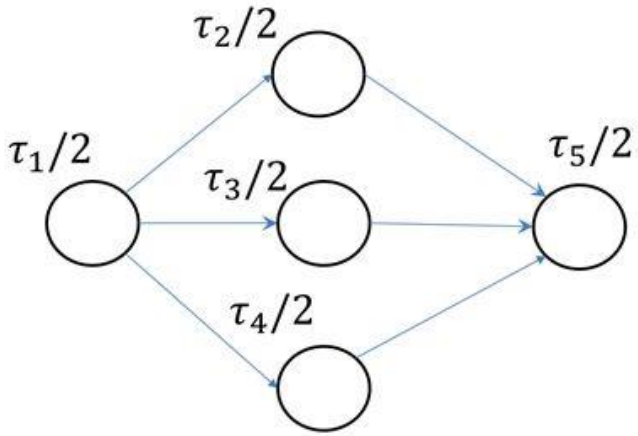
M = number of processors = 2

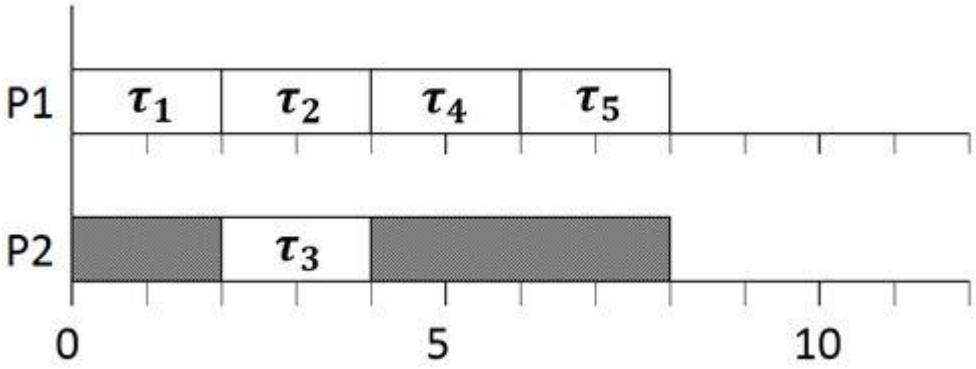
N = number of tasks = 5



Aperiodic – Processor Sharing

- Without processor sharing, the optimum schedule is:

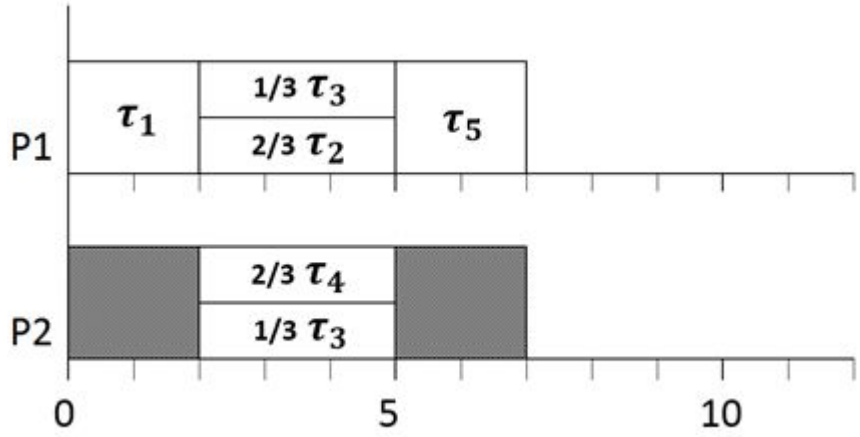




T=8, processor utilization = 5/8

Aperiodic – Processor Sharing

- With processor sharing, the optimum schedule is:

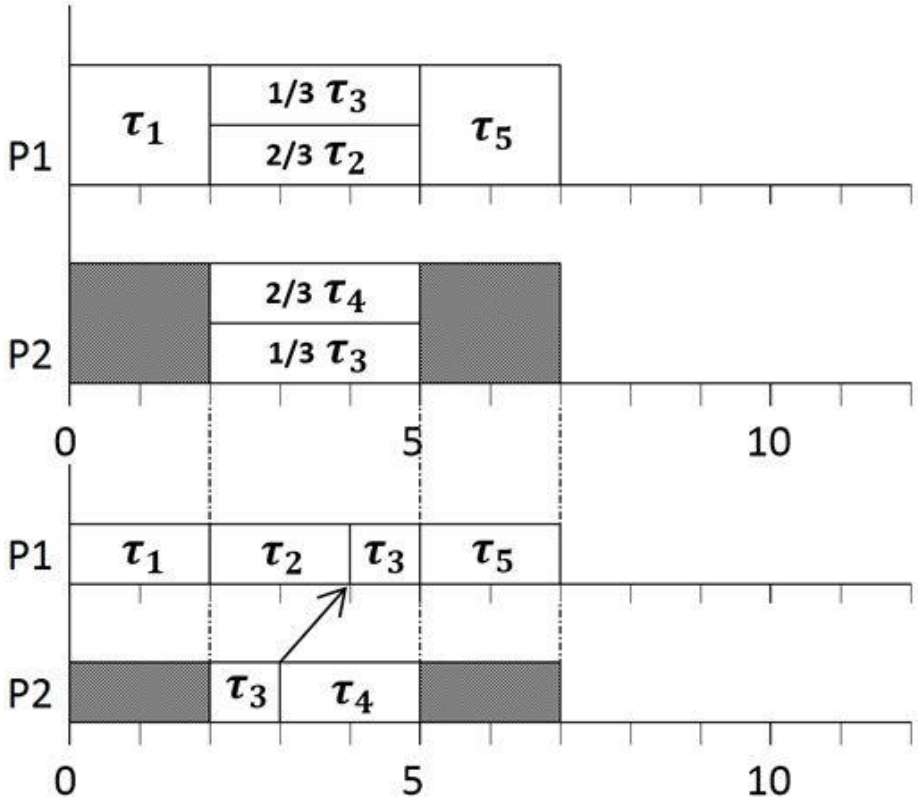


T=7, processor utilization = 5/7

The translation is not unique, since the assignments to processors can be made in several ways as long as a single task does not execute on two processors simultaneously

Aperiodic – Processor Sharing

- The processor sharing schedule can be converted to a preemptive schedule:



It is obvious that the process of preemption is needed to realize a processor sharing schedule.

Aperiodic - Preemptive With Precedence Constraints

- This problem has a polynomial time solution if all task execution times are equal and
  - **M** ≤ 2, or
  - the precedence graph is a tree (each task has at most one immediate *successor)*

- In these two special cases, a technique known as **Critical Path Scheduling** produces an optimal result.

- In any other case the problem is **NP**-hard, in which case the Critical Path Scheduling technique is an heuristic.

Aperiodic - Preemptive With Precedence Constraints

Critical Path Scheduling Algorithm:

- Assign tasks to processors according to their relative urgency, starting at the highest level in the precedence graph. If there is a tie among a number of tasks A for the last B processors ($B < A$), then assign the A tasks to the B processors using processor sharing, i.e. assign the B/A of a processor to each task.
- Re-assign the tasks to processors as described above, when either
  - a task is completed, or
  - a time is reached at which one of the tasks is executing at a rate higher than that at which a task of higher relative urgency is executing,
- The algorithm ensures that task reassignment occurs continuously in a manner that ensures the most critical tasks will always be executing.

Aperiodic - Preemptive With Precedence Constraints

Relative urgency

For a task relative urgency is defined as the maximum of the sums of the execution times along the various processing chains headed by the task in the yet unexecuted part of the precedence graph.

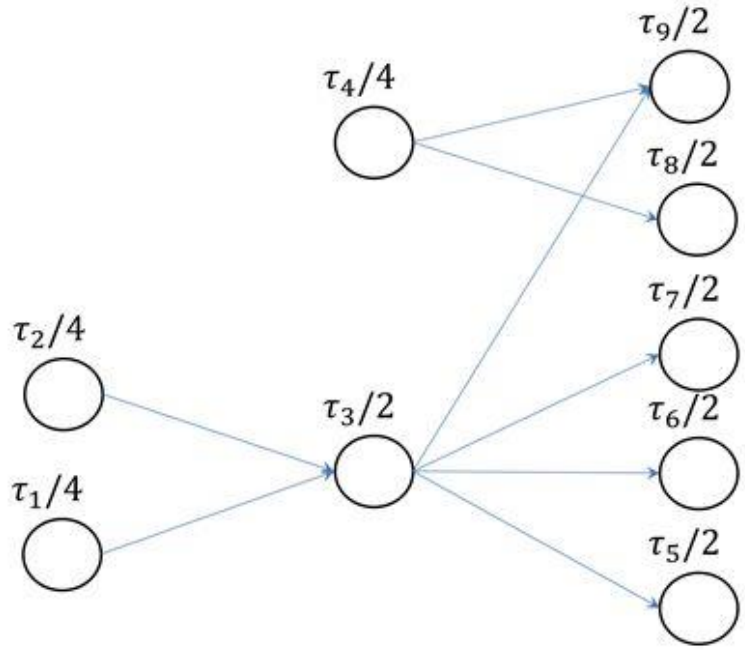
Relative urgency for each task changes as the schedule is executed.

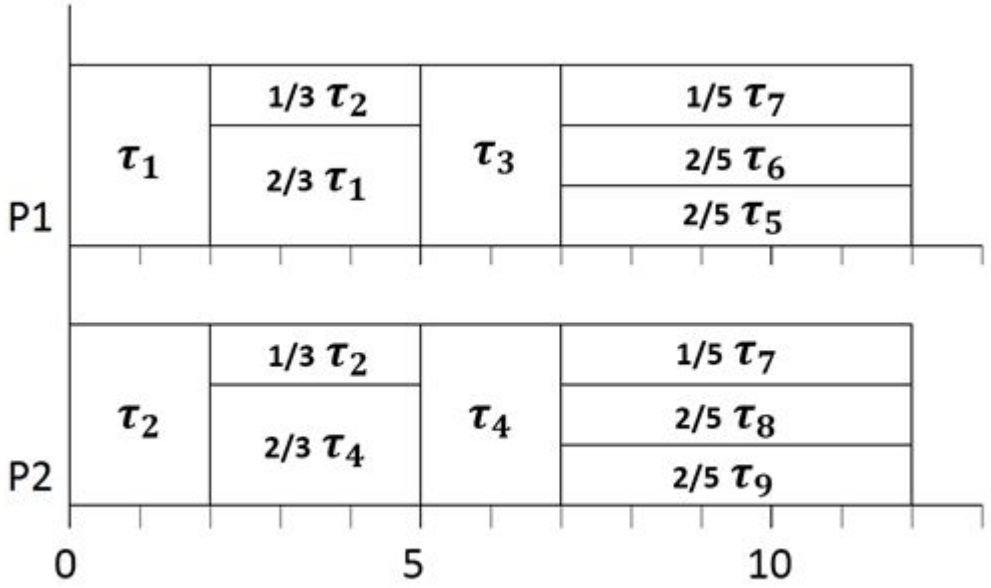
Relative urgency for each task can be computed at every instant in the schedule. Practically, it can be computed periodically.

If the schedule is computed prior to run-time this does not constitute an overhead.

Aperiodic - Preemptive With Precedence Constraints

Critical Path Scheduling Algorithm Example with **M** = 2:



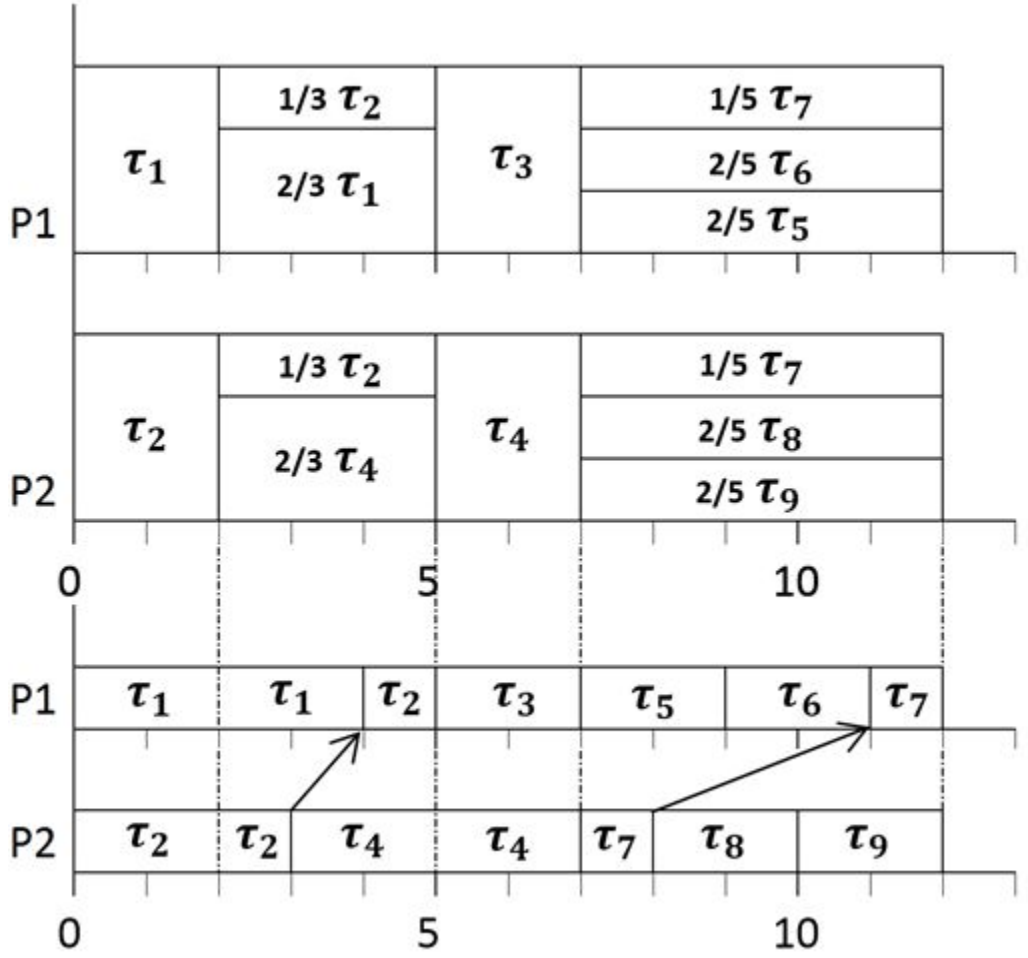


Aperiodic - Preemptive With Precedence Constraints

Critical Path Scheduling Algorithm

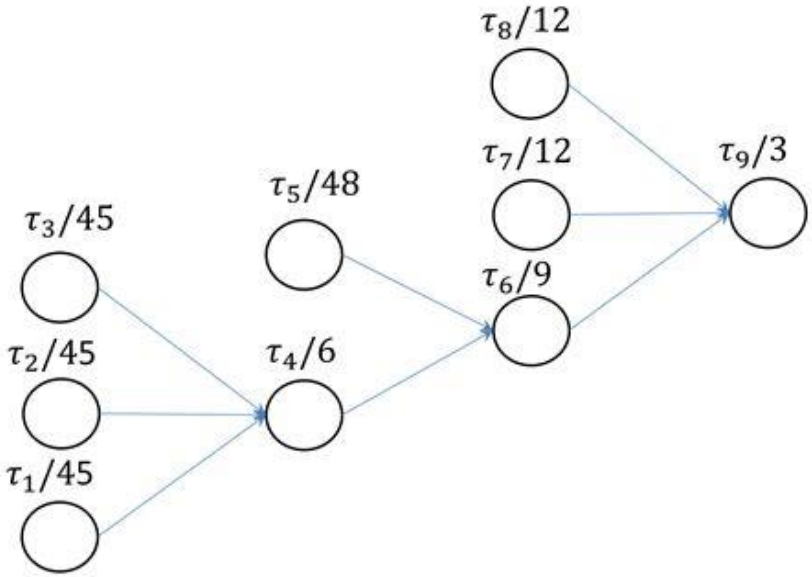
Example with **M** = 2:

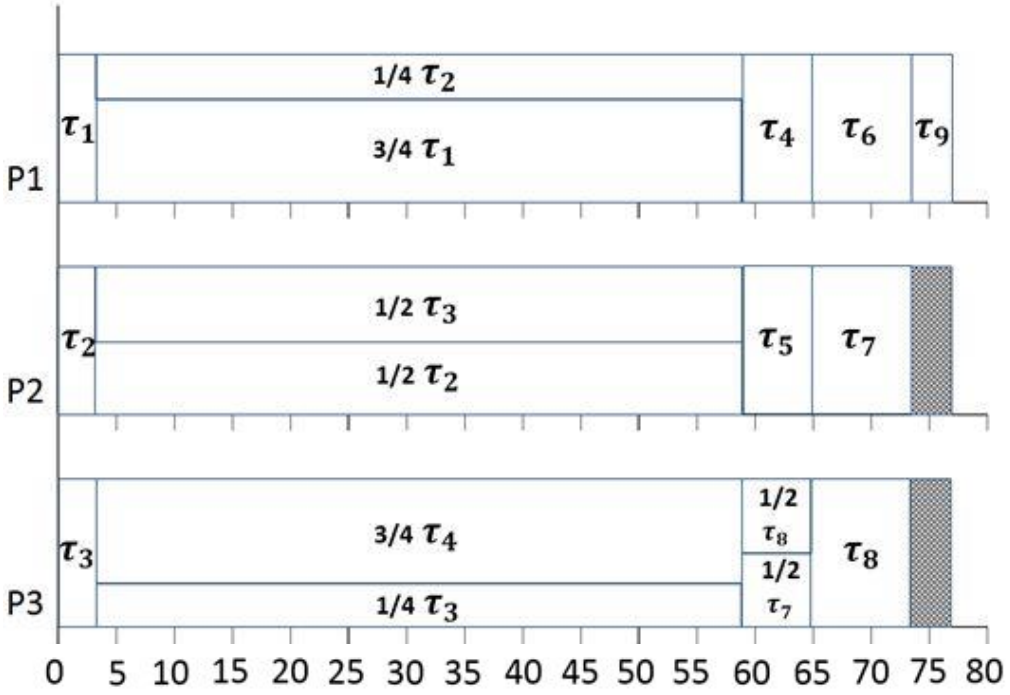
Conversion to a Realizable Schedule



Aperiodic - Preemptive With Precedence Constraints

Critical Path Scheduling Algorithm Example with **M** = 3:





Tree-like precedence graph

T = 25.66

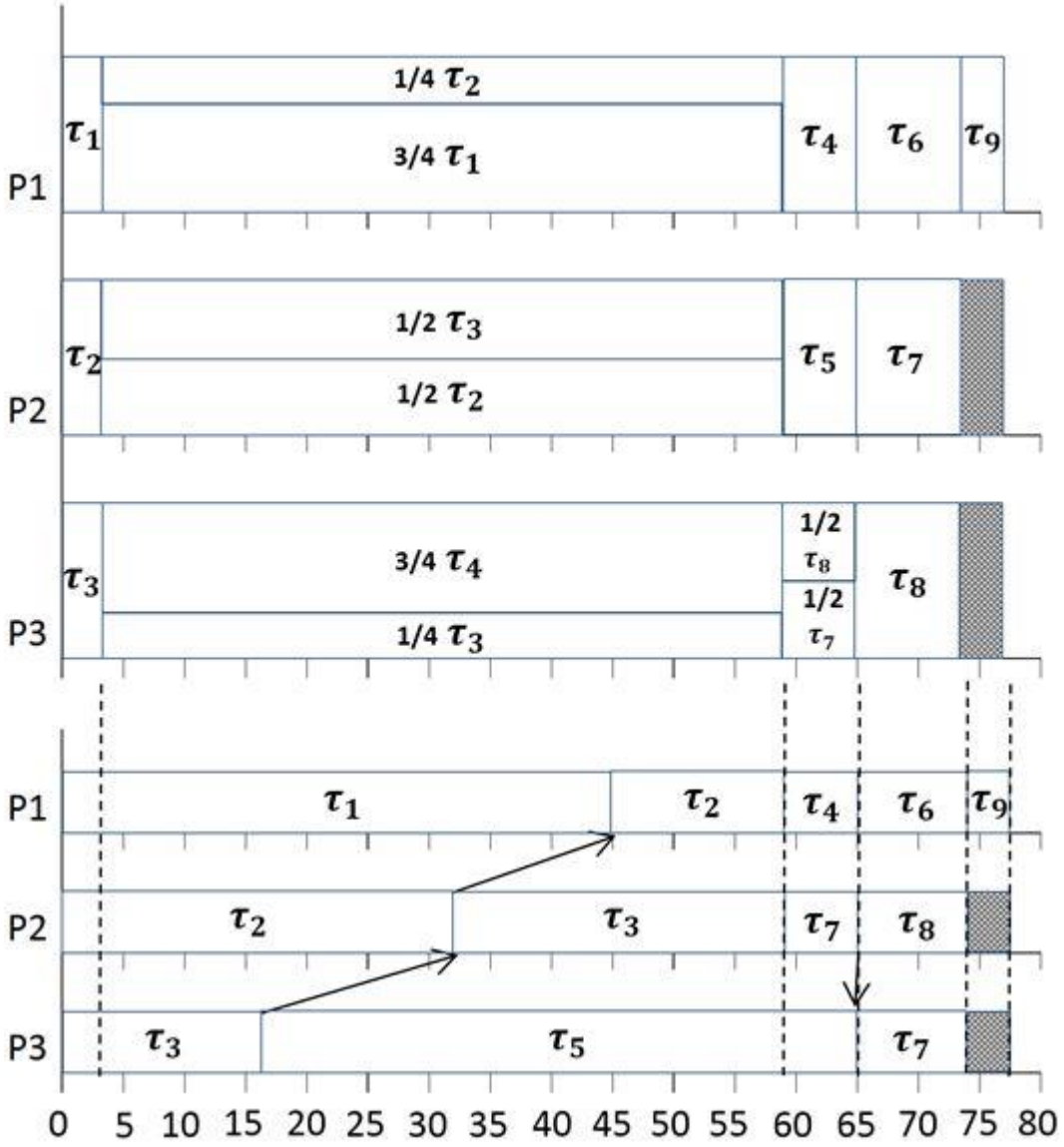
Processor utilization = 97%

Aperiodic - Preemptive With Precedence Constraints

Critical Path Scheduling Algorithm

Example with **M** = 3:

Conversion to a Realizable Schedule



Aperiodic Task Scheduling Summary

We have covered the following **aperiodic task** scheduling algorithms:

- Largest Processing Time Algorithm (no precedence constraints, no preemption)
- Priority List Scheduling Algorithm (with precedence constraints, no preemption)
- Decreasing Priority List Scheduling Algorithm (with precedence constraints, no preemption, LPT rule for priority)
- Optimal Preemptive Scheduling Algorithm (no precedence constraints, with preemption, without processor sharing)
- Processor Sharing Algorithm (with precedence constraints, with preemption, with processor sharing)
- Critical Path Scheduling (with precedence constraints, with preemption, with processor sharing)

Aperiodic Task Scheduling Summary

- The assumptions and characteristics were:
  - Tasks were aperiodic
  - Tasks have deterministic execution times
  - In most cases, the scheduling problem is intractable
  - Heuristics play an important part
  - Scheduling is static
  - Release times and deadlines have not been considered
  - The single processor case is usually trivial

When tasks are periodic, the scheduling problem becomes much more complex.

(For example, the single processor case can become very complex)

Periodic Task Scheduling

- Real-time systems typically respond to a number of external or internal stimuli that are frequently periodic
- Periodic task scheduling model becomes an important issue
- Sets of periodic tasks in the presence of asynchronous requirements
- Formalized as system parameters
  - **Task Periods:** Periodic tasks have *periods* which are defined as the time interval after which it must be repeated. Task periods are determined by the system requirements and may vary based upon the task specifics.
  - **Precedence and/or synchronization constraints**: Tasks do not typically execute independently of one another, and one task may generate data used by another task, or wait for it to complete.

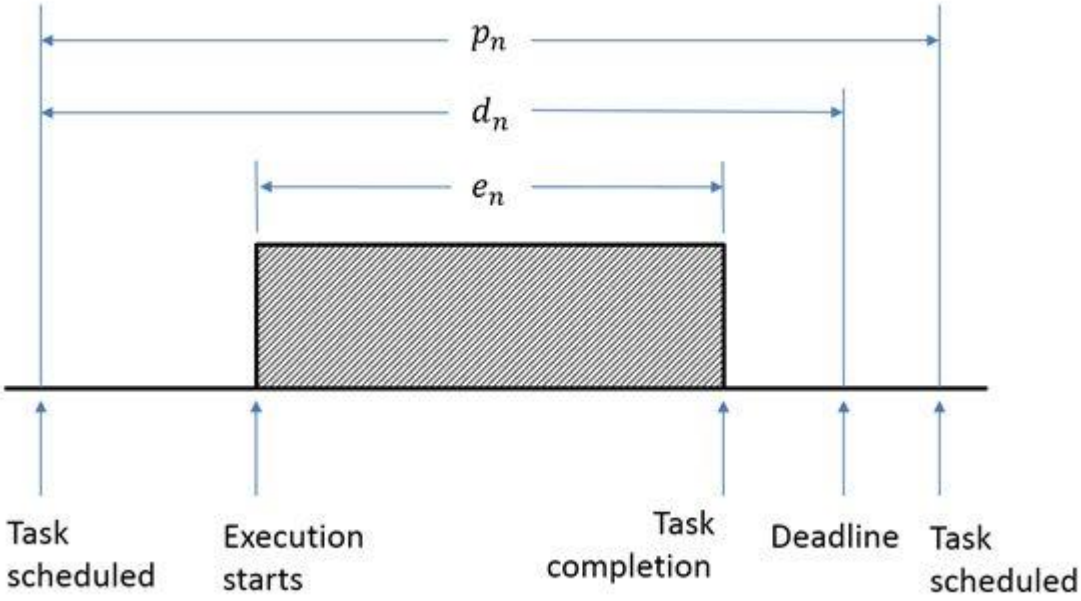
Periodic Task Scheduling

- **Task Deadlines**: The time by which task execution must be completed is termed a *deadline*. Usually a periodic task must be completed by the time it is again scheduled for execution (deadline= period), but a task may sometimes have a deadline shorter than the period.

- **Task execution times**: The amount of time a task requires to complete is the *execution time* and knowledge of these are required in order to develop a schedule.

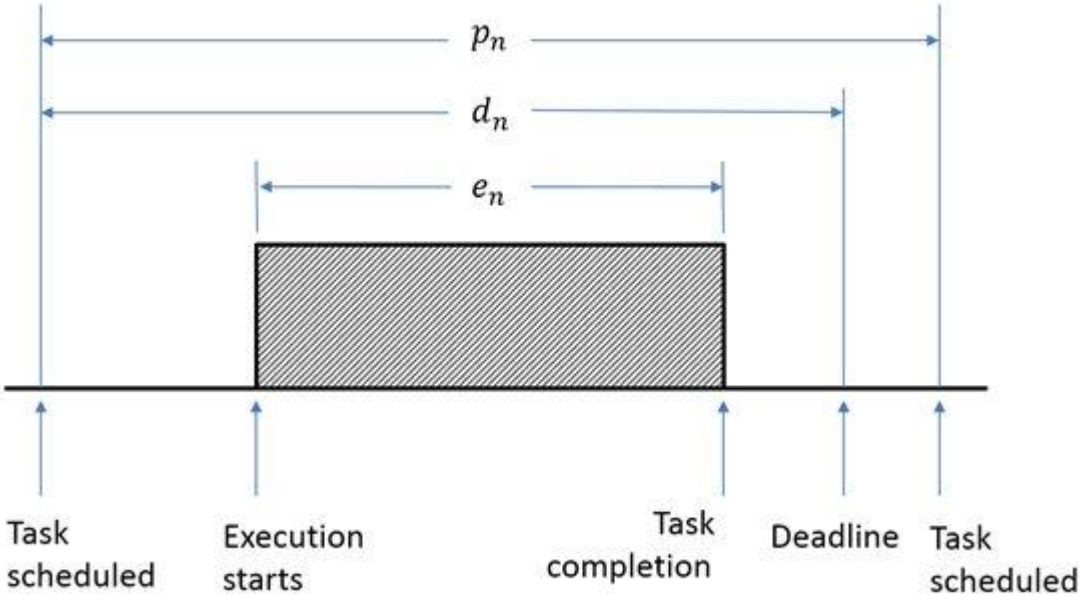
- The execution time of a task may vary from one execution to the next. This is known as *jitter* and is due to differing execution paths taken through the code. Jitter can constitute a major problem in periodic task schedules, notably those implemented by deterministic, timer driven, cyclic executive software.

Task Model



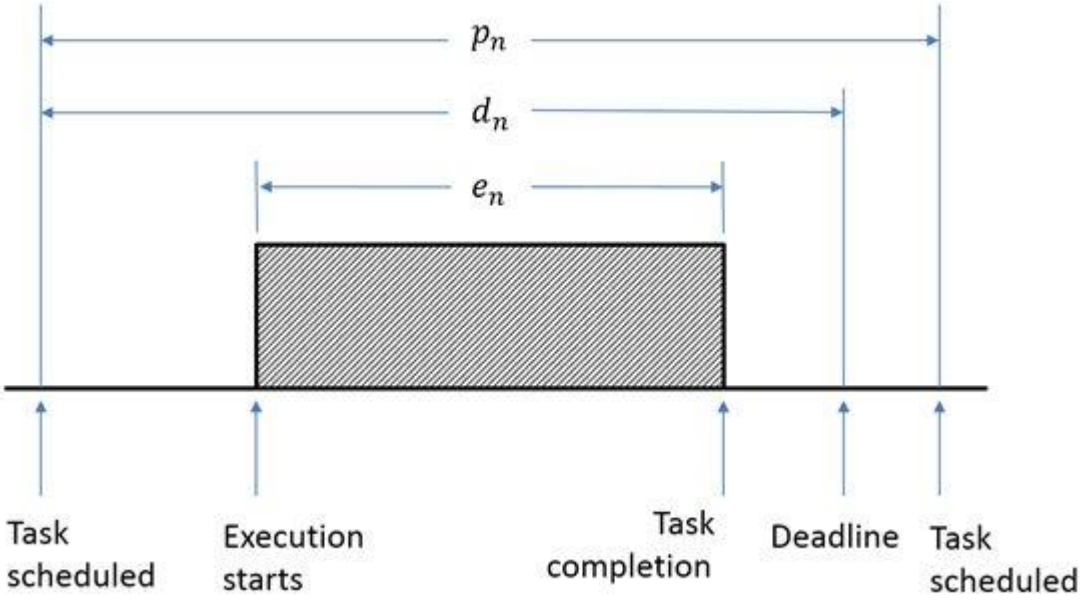
- Task Number: The *n*-th task is referenced as $\boldsymbol{\tau_n}$.
- Release Time: The *release time* of the *n*-th task is labeled $\boldsymbol{r_n}$. The release time is the time at which a task is scheduled to execute. It may begin execution time any time after its release time, but it may not begin execution *before* its release time.

Task Model



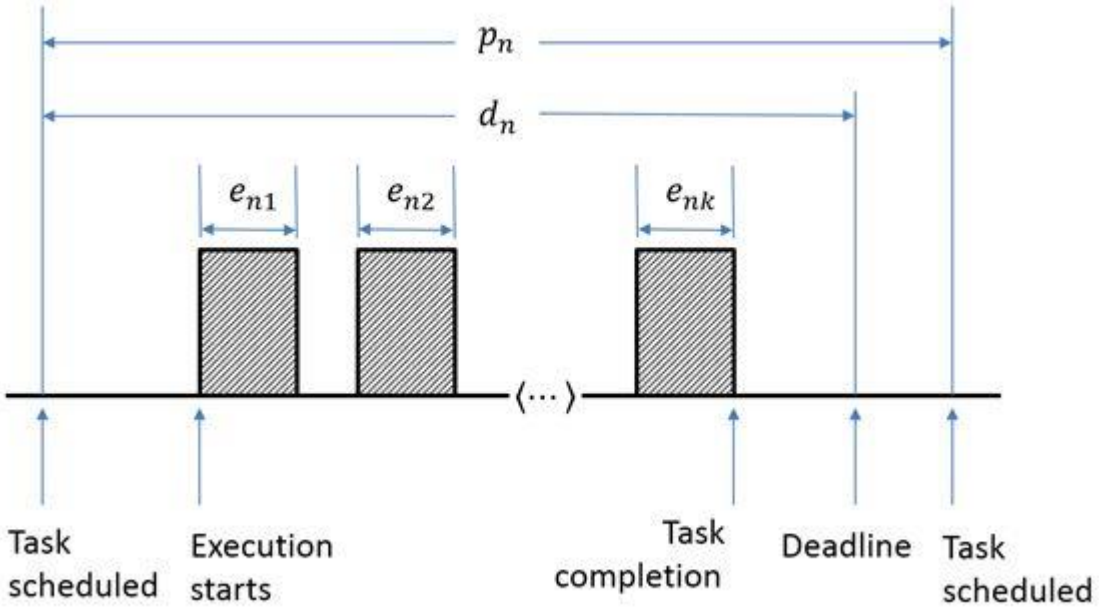
- Period: The *period* of the *n*-th task in $\boldsymbol{p_n}$ and it is the time interval between successive release times of the task.
- Deadline: The *deadline* of the *n*-th task is designated as $\boldsymbol{d_n}$. The deadline is a time period following the release time of a task within which the task must execute to completion.

Task Model



- Execution Time: The execution time of the *n*-th task is $\boldsymbol{e_n}$. This is the time during which the task is actually executing.
- The above task model does not require the actual execution of the task to be periodic, but rather that it is the release times that occur in a periodic manner.

Preemptive Task Model



- This kind of task execution may occur as a result of the development of a cyclic executive schedule in which the task is broken into subtasks in order to fit it into the schedule.
- Alternatively, it may occur dynamically as a RTOS preempts and resumes the task during execution. In either case, the task cannot begin execution prior to its release time, and it must complete prior to its deadline.

Periodic Task Scheduling

These choices affect the scheduling process.

- **Static or Dynamic Scheduling**: Static schedules do not change during real-time operation. They are generally implemented using a cyclic executive design driven by an interrupt timer. On the other hand, dynamic schedules may change during system operation and are implemented using a multitasking executive (RTOS) and priority-driven scheduling.
- **Preemption:** Tasks may or may not be allowed to be preempted. For designs based on cyclic executives, preemption means that the tasks can be arbitrarily divided into subtasks to achieve synchronization, or in order to "*fit*" them into the schedule. For priority-driven systems, preemption implies that the real-time operating system is free to arbitrarily suspend an executing subtask in order for a higher priority subtask to execute.
- **Asynchronous processing**: Most systems will have requirements represented by aperiodic tasks. The arrival of aperiodic tasks is usually signaled and handled by an interrupt, but normal interrupt processing may not always be sufficient and special server techniques must be applied.

Periodic Task Scheduling

- The primary design question is whether or not a given task system with a specified set of task priorities can be scheduled in a manner that ensures all tasks meet their deadlines.

- The general scheduling problem remains very difficult. Consequently, heuristic methods are often used.

- An effective heuristic design method for generating cyclic schedules is to assign task priorities according to some algorithm, and then use these priorities to generate the schedule.

Periodic Task Scheduling

- One method of performing the schedulability analysis of a dynamic, priority driven, task scheduling process is to build the schedule that would result if all tasks executed for precisely the execution time specified for the task.

- Applies both to cyclic executive schedule design and to the schedulability analysis of dynamic, priority-driven systems

- The arguments presented above imply that it does not matter whether the goal is to analyze the schedulability of dynamically scheduled, priority-driven systems using static task priorities, or to design a static cyclic schedule using an algorithm that assigns tasks according to a set of static priorities. The process is the same. A similar statement holds for the case of dynamic task priorities.

Static Task Scheduling

- Static task scheduling refers to the situation in which a cyclic schedule is predetermined and the scheduler, in this case a cyclic executive, executes the tasks according to this schedule.
  - Static task scheduling with *static task priorities* refers to the situation in which a predetermined schedule is developed using an algorithm that assigns priorities to the tasks and then assigns tasks according to these priorities. The priorities are *static*, in that their values do not vary during the development of the schedule.
  - Static task scheduling with *dynamic task priorities* refers to the situation in which a predetermined schedule is developed using an algorithm that assigns priorities to the tasks and then schedules tasks according to these priorities. The priorities are *dynamic*, in that their values vary during the development of the schedule in accordance with the algorithm used for their determination.

Dynamic Task Scheduling

- Dynamic task scheduling refers to the situation in which tasks are scheduled dynamically by a multitasking executive as the program executes.
  - Dynamic task scheduling with *static task priorities* refers to the situation in which the scheduler of a multitasking executive dynamically schedules tasks in accordance with a set of *static* task priorities. Static priorities, by definition, are preset and remain fixed throughout the execution of the program.
  - Dynamic task scheduling with *dynamic task priorities* refers to the situation in which the scheduler of a multitasking executive dynamically schedules tasks in accordance with a set of *dynamic* task priorities. Dynamic priorities, by definition, can be modified as the program executes. The manner in which these modifications take place is defined by the particular priority assignment algorithm.

Schedulability Analysis

- The term *schedulability analysis* refers to those techniques used to determine if a given task system can be scheduled on one or more processors.

  - A task system is termed ***synchronous*** if the initial release times of all tasks are identical. Without loss of generality, this common release time can be taken to be zero. A task system that is not synchronous is said to be ***asynchronous***. The complexity of a scheduling problem varies considerably depending on whether the task system is synchronous or asynchronous.

  - A schedule is said to be ***valid*** for a specific task system if the schedule provides for meeting the deadlines for all task requests. The term ***all task requests*** includes the infinity of requests that exist for a periodic task, and the question that is immediately raised concerns the problem of guaranteeing deadlines for an infinite time.

Schedulability Analysis

- A task system is said to be ***feasible*** on M identical processors if there is a *valid* schedule for the task system on M identical processors.

- A task system is said to be ***schedulable*** on M identical processors if there is a *valid* schedule on M identical processors produced by a *static* task priority assignment. The priority assignment may be used by a multitasking executive to dynamically schedule the tasks, or it may be used as a scheduling algorithm in developing a static task schedule.

- A scheduling algorithm is said to be ***optimal*** if it always produces a *valid* schedule for every task system that is *feasible*. This use of the term *optimal* deserves particular attention. It implies that, if a task system is feasible, an optimal algorithm will produce a valid - but not necessarily uniquely so task schedule.

Schedulability Analysis

- *Processor utilization* $\boldsymbol{U}$, generalized to the M processor case, produces the following inequality as a *necessary but not sufficient* condition for a task system to be feasible on M identical processors

$$U = \sum_{n=1}^{N} \frac{e_n}{p_n} \leq M$$

(EQ 5.3.2)

Schedulability Analysis

- *Deadline utilization* $\boldsymbol{D}$ leads to the following inequality as a *sufficient but not necessary* condition for a task system to be feasible on M identical processors

$$D = \sum_{n=1}^{N} \frac{e_n}{d_n} \leq M$$

(EQ 5.3.3)

Schedulability Analysis

- For synchronous task systems in which all task deadlines are equal to the corresponding periods, *i.e.*

$$d_n = p_n \text{ for all } \tau_n$$

the inequality of the first equation above is both necessary and sufficient.

YES/NO question "is the task system feasible?"

Schedulability Analysis

- We need to determine whether a schedule produced by a particular algorithm for *static* priority assignment is *valid*

  - The *response time* of a task is the time interval between the release time of the task and the completion time of the task. The response time is different from the execution time of the task. While the execution time is the total of the actual time the task spends executing, the response time is comprised of the actual execution time plus any suspension time.

  - A *critical instant* of a task is a time at which the task is released and has the largest response time of all task releases.

Schedulability Analysis

- The "*critical instant*" concept necessarily defines the extreme condition under which a task will meet its deadline under the worst case scenario.
- It is easy to visualize that the task deadline will be met *for all release times* of the task *if and only if* the deadline is met when the release time occurs at a critical instant.
- Under *static* task priorities, a critical instant will occur for a particular task if it is released simultaneously with all higher priority tasks in the system. Furthermore, if the task system is *synchronous,* the initial releases of all the tasks occur at $t = 0$.

Schedulability Analysis

- The following important result can be stated:

  For scheduling a synchronous task system on a single processor, there is a pseudo-polynomial time algorithm for deciding if the schedule produced by a given static priority assignment is valid. The algorithm is the construction of the schedule from a critical instant through the end of the longest period.

- Instead of the graphical construction of the schedule it is sometimes more convenient to have analytical techniques to investigate task system schedulability on a single processor.

Schedulability Analysis

- Analytical methods such as utilization bounds are sometimes more convenient to determine schedulability of a task system than creating a graphic schedule.
  - The *execution time* of the task itself is the actual time the task spends executing its code.
  - The *preemption time* is the time the task is suspended because of preemption by *higher* priority tasks.
  - The *blocking time* is the time the task is blocked by *lower* priority tasks.

Schedulability Analysis

- Analytical methods such as utilization bounds are sometimes more convenient to determine schedulability of a task system than creating a graphic schedule.
  - The *execution time* of the task itself is the actual time the task spends executing its code.
  - The *preemption time* is the time the task is suspended because of preemption by *higher* priority tasks.
  - The *blocking time* is the time the task is blocked by *lower* priority tasks.

Schedulability Analysis

- The concept of processor utilization that hitherto was based solely on task execution time has to be extended to include some additional effects:
  - Preemption by tasks with priorities *greater than or equal to* that of task τ_n, but with periods *greater than or equal to* the deadline of task τ_n. The set of tasks with priorities higher than that of task τ_n is designated as S_n. The subset of S_n containing those tasks with periods greater than or equal to the deadline of task τ_n is designated as S_{ng} with the number of tasks in the subset being N_{ng}. Tasks in subset S_{ng} can preempt task τ_n *only once* before its deadline.

Schedulability Analysis

- Preemption by tasks with priorities *greater than or equal to* that of task τ_n, but with periods *less than* the deadline of task τ_n. The subset of S_n containing those tasks with periods less than the deadline of task τ_n is designated as S_{nl} with the number of tasks in the subset being N_{nl}. Tasks in subset S_{nl} can preempt task τ_n *multiple times* before its deadline.
- Blocking delays produced by *lower* priority tasks. Blocking due to any cause is included in the value of worst-case blocking time b_n.

- Now we will determine Effective Utilization and compare it to the Utilization Bound

Schedulability Analysis

- The effective task utilization based on all these possible delays can now be defined as:

$$E_n = \sum_{S_{nl}} \frac{e_i}{p_i} + \frac{1}{p_n}\left[e_n + b_n + \sum_{S_{ng}} e_i\right]$$

(EQ 5.3.4)

- Using this equation, the effective utilization of each task τ_n is computed and compared to a worst-case utilization bound.

Schedulability Analysis

- The worst-case utilization bound for task τ_n is determined as follows:

$$UB_n = \begin{cases} \dfrac{d_n}{p_n}, & \dfrac{d_n}{p_n} \leq \dfrac{1}{2} \\ [N_{nl}+1]\left[\left\{\left(\dfrac{2d_n}{p_n}\right)^{\frac{1}{N_{nl}+1}} - 1\right\} + 1 - \dfrac{d_n}{p_n}\right], & \dfrac{d_n}{p_n} > \dfrac{1}{2} \end{cases}$$

(EQ 5.3.5)

Schedulability Analysis

- If $d_n = p_n$ the equation reduces to

$$UB_n = (N_{nl} + 1)\left(2^{\frac{1}{N_{nl}+1}} - 1\right)$$

(EQ 5.3.6)

- If E_n is less than or equal to the utilization bound value, the task will meet its first deadline and therefore all future deadlines.

- Each task in the system must satisfy this test for a comprehensive conclusion.

Schedulability Analysis

- On the other hand, if the effective utilization of a task exceeds the bound, no information is conveyed.
- This is, a worst-case bound, by definition, is a *sufficient but not necessary* condition.
- In the event that the utilization bound test is inconclusive, the analysis must resort to schedule building or alternative equivalent technique.

Utilization Bound Test

Example 5.3.1: Using the utilization bound test.

The utilization bound test can be applied to the following task system.

$N = number\ of\ tasks = 5$

M = number of processors = 1

Period	Deadline	Execution Time	Blocking Time
$p_1 = 8$	$d_1 = 2$	$e_1 = 1$	$b_1 = 0$
$p_2 = 60$	$d_2 = 60$	$e_2 = 16$	$b_2 = 0$
$p_3 = 36$	$d_3 = 28$	$e_3 = 4$	$b_3 = 0$
$p_4 = 50$	$d_4 = 30$	$e_4 = 2$	$b_4 = 1$
$p_5 = 30$	$d_5 = 30$	$e_5 = 2$	$b_5 = 0$

Priority list = $\{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$

The utilization bound test is required to be applied to each task separately.

Utilization Bound Test

Test task τ_5:

Determine S_{5g}= $\{\tau_2, \tau_3, \tau_4\}$. This is the set of tasks with priorities greater than or equal to that of task τ_5, and with periods greater than or equal to the deadline of task τ_5. The number of tasks in this set is $N_{5g} = 3$.

Determine S_{5l}= $\{\tau_1\}$. This is the set of tasks with priorities greater than or equal to that of task τ_5, and with periods less than the deadline of taskτ_5. The number of tasks in this set is $N_{5l} = 1$.

Calculate $\frac{d_5}{p_5} = 1$ and use EQ 5.3.5 to calculate the utilization bound

$$UB_5 = (1+1)\left(2^{\frac{1}{1+1}} - 1\right) = 0.828$$

This value will be compared to the effective utilization of task $\tau_5$calculated as follows:

$$E_5 = \frac{1}{8} + \frac{1}{30}[2 + 16 + 4 + 2] = 0.925$$

Utilization Bound Test

The effective utilization of task τ_5 is larger than the utilization bound. This result implies only that the utilization bound test cannot conclusively determine if task τ_5 will meet its first deadline. It does not imply that task τ_5 will miss its first deadline.

Obviously some other test must be applied if a determination of the schedulability of task τ_5 is to be made. Such a test will be described in the following example, but first the utilization bound test will be applied to the remaining tasks.

Test task τ_4:

$S_{4g} = \{\tau_2, \tau_3\}$ $\quad N_{4g} = 2$

$S_{4l} = \{\tau_1\}$ $\quad N_{4l} = 1$

$$UB_4 = 2[2(0.6^{0.5} - 1) + 1 - 0.6] = 0.591$$

$$E_4 = \frac{1}{8} + \frac{1}{50}[2 + 1 + 16 + 4] = 0.585$$

Since the effective utilization is less that the utilization bound, task τ_4 will meet its first deadline.

Utilization Bound Test

Test task τ_3:

$S_{3g} = \{\tau_2\}$ $\quad N_{3g} = 1$

$S_{3l} = \{\tau_1\}$ $\quad N_{3l} = 1$

$$UB_3 = 2[(1.55^{0.5} - 1) + 1 - 0.77] = 0.716$$

$$E_3 = \frac{1}{8} + \frac{1}{36}[4 + 12] = 0.569$$

Since the effective utilization is less that the utilization bound, task τ_3 will meet its first deadline.

Utilization Bound Test

Test task τ_2:

$S_{2g} = \{\ \}$ $\quad N_{2g} = 0$

$S_{2l} = \{\tau_1\}$ $\quad N_{2l} = 1$

$$UB_2 = (1+1)\left(2^{\frac{1}{1+1}} - 1\right) = 0.828$$

$$E_2 = \frac{1}{8} + \frac{1}{60}[12] = 0.325$$

Since the effective utilization is less that the utilization bound, task τ_2 will meet its first deadline.

Utilization Bound Test

Test task τ_1:

$S_{1g} = \{\quad\}$ $\qquad N_{1g} = 0$

$S_{1l} = \{\quad\}$ $\qquad N_{1l} = 0$

$$UB_1 = \frac{d_1}{p_1} = \frac{2}{8} = 0.25$$

$$E_1 = \frac{1}{8} = 0.125$$

Since the effective utilization is less that the utilization bound, task τ_1 will meet its first deadline.

Utilization Bound Test

The results of the application of the utilization bound tests are summarized here:

- The given priorities will produce a valid schedule for tasks τ_1, τ_2, τ_3, and τ_4, since the effective utilization of each of these tasks does not exceed the corresponding utilization bound.

- In the case of task τ_5 the utilization bound test failed to show that the given priorities will produce a valid schedule. The test did not show that a valid schedule could not be produced.

The utilization bound tests constitute sufficient but not necessary conditions. They do not conclusively answer the question of the schedulability of the task system. Schedule construction or some other conclusive method is required for this purpose.

Completion Time Test

- The obvious ***conclusive test*** is to construct the schedule, but schedule construction is tedious.
- An analytical method exists as an alternative to the actual graphical construction of the schedule.
- It can be easily implemented as a software algorithm on a computer.
- The algorithm involves the solution of an iterative equation for the first completion times of the tasks in a task system.
- If the first completion time of a task does not exceed the associated deadline, then the task meets its first deadline and consequently meets all deadlines.

Completion Time Test

- The iterative equation for the completion time of a task is given by

$$C_n(i+1) = e_n + b_n + \sum_{j=1}^{n-1} \left\lceil \frac{C_n(i)}{p_j} \right\rceil e_j$$

(EQ 5.3.7)

$C_n()$ is the completion time computed iteratively until a stable value is attained, e_i is the task execution time, b_n is the worst case task blocking time, p_j is the task period and the operator $\left\lceil \frac{a}{b} \right\rceil$ means the smallest integer greater than or equal to $\frac{a}{b}$.

The initial value is given by

$$C_n(0) = b_n + \sum_{j=1}^{n} e_j$$

(EQ 5.3.8)

Utilization Bound Test

Example 5.3.1: Using the utilization bound test.

The utilization bound test can be applied to the following task system.

$N = number\ of\ tasks = 5$

M = number of processors = 1

Period	Deadline	Execution Time	Blocking Time
$p_1 = 8$	$d_1 = 2$	$e_1 = 1$	$b_1 = 0$
$p_2 = 60$	$d_2 = 60$	$e_2 = 16$	$b_2 = 0$
$p_3 = 36$	$d_3 = 28$	$e_3 = 4$	$b_3 = 0$
$p_4 = 50$	$d_4 = 30$	$e_4 = 2$	$b_4 = 1$
$p_5 = 30$	$d_5 = 30$	$e_5 = 2$	$b_5 = 0$

Priority list = $\{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$

The utilization bound test is required to be applied to each task separately.

Completion Time Test

Example 5.3.2: Calculating precise completion times for periodic tasks.

Apply the completion time computation to task τ_5. For the initial value we obtain,

$$C_5(0) = 0 + \sum_{j=1}^{5} e_j = 1 + 16 + 4 + 2 + 2 = 25$$

Going through the iterations:

$i = 1$:

$$C_5(1) = e_5 + b_5 + \sum_{j=1}^{4} \left\lceil \frac{C_5(0)}{p_j} \right\rceil e_j = 2 + 0 + 4 + 16 + 4 + 2 = 28$$

$i = 2$:

$$C_5(2) = e_5 + b_5 + \sum_{j=1}^{4} \left\lceil \frac{C_5(1)}{p_j} \right\rceil e_j = 2 + 0 + 4 + 16 + 4 + 2 = 28$$

The iteration has converged and indicates that the first completion of task τ_5 will occur at time $t = 28$, which is less than the deadline value $d_5 = 30$

Completion Time Test

- The calculation of the completion time of a task by the above method is precise as no approximations or worst-case bounding values are involved.
- To use the completion time test to show that a task system is schedulable using a given priority list, the completion time of each task must be determined.
- An easier approach to the problem of determining the schedulability of a task system is to first apply the utilization bound test to all tasks, and then to apply the completion time method to those tasks for which the result of the utilization bound test is inconclusive.
- This is essentially the method applied to the example problem.

Task Priority Assignment

- Schedulability analysis presumes the existence of a task priority list
- The highest priority tasks should be executing at any particular time assuming that they can be executed (not waiting on a resource)
- Tasks are preemptively assigned to processors in accordance with this priority list in a manner that ensures this rule
- How to choose task priority??
- We need a RULE or ALGORITHM for this purpose

Task Priority Assignment

- Schedulability analysis presumes the existence of a task priority list
- The highest priority tasks should be executing at any particular time assuming that they can be executed (not waiting on a resource)
- Tasks are preemptively assigned to processors in accordance with this priority list in a manner that ensures this rule
- How to choose task priority??
- We need a RULE or ALGORITHM for this purpose

Task Priority Assignment

- It is important to recall that a scheduling algorithm is said to be *optimal* if it always produces a *valid* schedule for every task system that is *feasible*

- The known optimal task scheduling algorithms are all priority list algorithms. Such an algorithm defines the basis upon which a priority list of the tasks is constructed.

- The *intrinsic importance* of a task (some measure of how vital the service performed by the task is to the overall functioning of the real-time system) has no bearing on the assigned priority of the task (Why?)

Task Priority Assignment

- Each of these algorithms is optimal for a certain class of task scheduling problem

  - The *deadline-monotonic algorithm* assigns task priorities in order of increasing deadlines. This is a *static* priority assignment algorithm.

  - The *rate-monotonic algorithm* assigns task priorities in order of increasing periods, *i.e.* decreasing rates (frequencies). This is a *static* priority assignment algorithm.

  - The *earliest deadline algorithm* schedules tasks in the following manner: at each instant of time task priorities are assigned in increasing order of currently impending deadlines. This is a *dynamic* priority assignment algorithm, since the priority assignments vary as the execution proceeds.

Deadline Monotonic Scheduling

- Assigns priorities to tasks in the order of *increasing deadlines* d_n
- Task priorities are static and do not change once they are assigned, since a task deadline is a fixed parameter of the task system, determined *a priori,* by the system engineers
- Static scheduling case - the designer applies the algorithm to create a task priority list and then creates a schedule by preemptively allocating tasks to processors based on their priority
- Dynamic scheduling case - the designer applies the algorithm to create a task priority list and then creates task control blocks or instances in the software based on these priorities (in a modern RTOS task priority is parameter supplied as a part of the system call when the task is created)

Deadline Monotonic Scheduling

- It is an optimal static priority assignment algorithm for *synchronous task systems* executing on *one processor*
- An optimal task scheduling algorithm is one that will result in a valid task schedule if the task system is schedulable (even though the solution is not guaranteed to be unique)
- Therefore one merely needs to assign task priorities in accordance with the deadline-monotonic algorithm and then construct the schedule through the longest period. If all tasks meet their first deadlines the task system is feasible. Otherwise, it is not.
- In place of schedule construction, analytical methods can be used.

Deadline Monotonic Scheduling

Example 5.4.1: The use of deadline monotonic scheduling.

The use of deadline-monotonic scheduling will be illustrated by the construction of a schedule for the task system shown below.

$$N = number\ of\ tasks = 3$$

Period	Deadline	Execution Time
$p_1 = 4$	$d_1 = 2$	$e_1 = 1$
$p_2 = 6$	$d_2 = 4$	$e_2 = 2$
$p_3 = 10$	$d_3 = 10$	$e_3 = 3$

The processor utilization is determined to be

$$U = \frac{1}{4} + \frac{2}{6} + \frac{3}{10} = 0.883 < 1.$$

Deadline Monotonic Scheduling

The deadline utilization is determined to be

$$D = \frac{1}{2} + \frac{2}{4} + \frac{3}{10} = 1.3 > 1.$$

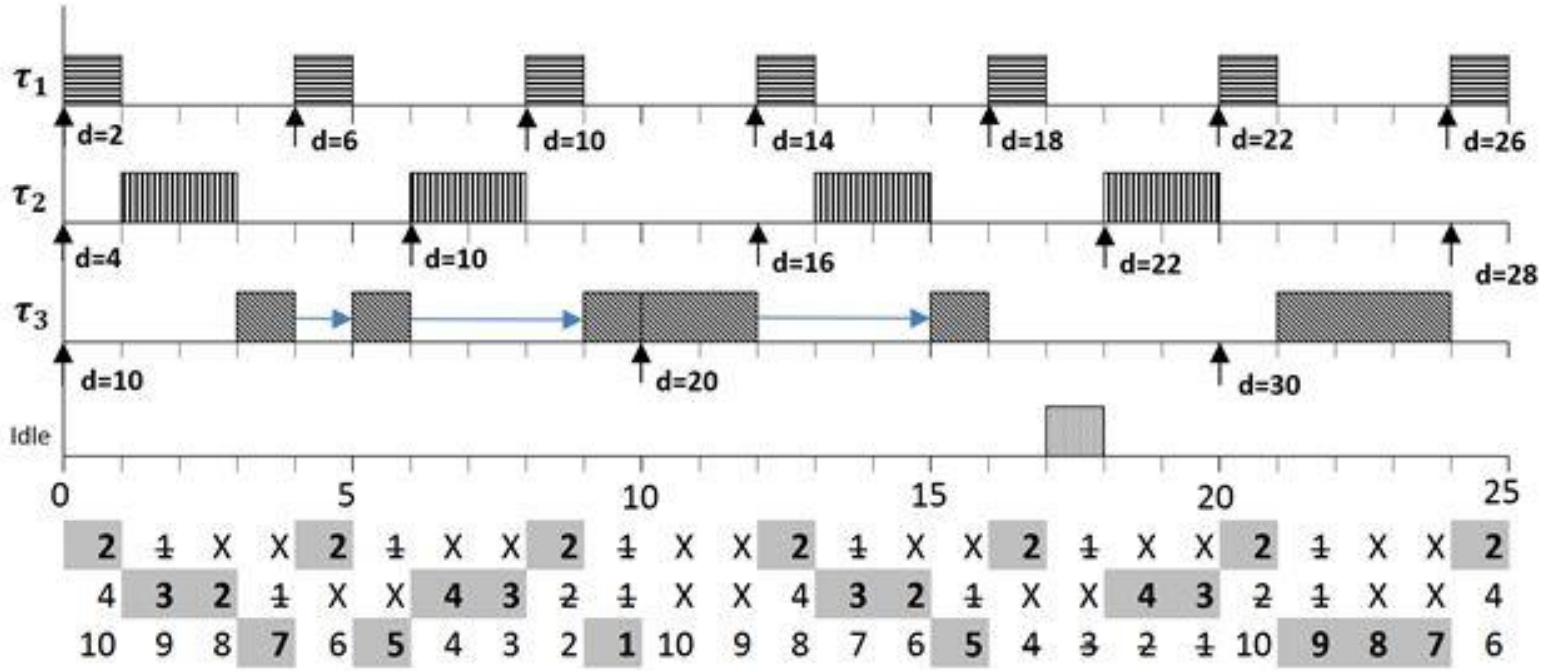
The first result implies that task system can possibly be scheduled.

The second result implies that it cannot be determined conclusively whether task system can be scheduled.

The only known algorithm for answering the YES/NO question, “is this task system schedulable?,” is to construct the schedule through the longest period using an optimal static priority assignment.

If all tasks meet their first deadline, they will meet their deadlines for all task releases, and the task system is schedulable.

Deadline Monotonic Scheduling



Each indication of a release time is accompanied by the associated deadline. Since the tasks are arbitrarily preemptable, the algorithm is free to suspend an executing task at any point so that a higher priority task can execute. That is precisely what happens to task τ_3 at times 4 and 6.

The schedule shows that all tasks meet their first deadlines, thus proving that the task system is schedulable.

Deadline Monotonic Scheduling

As an alternative to graphical schedule construction, analytical techniques can be applied, since application of these analytical techniques is the equivalent of schedule construction.

Example 5.4.2: Application of the completion time method.

It is informative to apply the completion time calculation to this task scheduling problem. The above iterative equation must be applied:

Calculate the completion time for task τ_3:

$$C_3(0) = 1 + 2 + 3 = 6$$

$$C_3(1) = 3 + \left\lceil \frac{6}{6} \right\rceil 2 + \left\lceil \frac{6}{4} \right\rceil 1 = 7$$

$$C_3(2) = 3 + \left\lceil \frac{7}{6} \right\rceil 2 + \left\lceil \frac{7}{4} \right\rceil 1 = 9$$

Deadline Monotonic Scheduling

$$C_3(3) = 3 + \left\lceil \frac{9}{6} \right\rceil 2 + \left\lceil \frac{9}{4} \right\rceil 1 = 10$$

$$C_3(4) = 3 + \left\lceil \frac{10}{6} \right\rceil 2 + \left\lceil \frac{10}{4} \right\rceil 1 = 10$$

The iteration bas converged, indicating that task τ_3 will complete at time $t = 10$, and this is verified by examination of the task schedule of Figure 5.18.

Calculate the completion time of task τ_2:

$$C_2(0) = 1 + 2 = 4$$

$$C_2(1) = 2 + \left\lceil \frac{2}{4} \right\rceil 1 = 3$$

The iteration has converged, indicating that task τ_2 will complete at time $t = 3$, and this is verified by examination of the task schedule of Figure 5.18.

This example demonstrates the equivalency of schedule construction and the analytical solution for completion times.

Deadline Monotonic Scheduling

A summary of deadline-monotonic task scheduling is presented below.

- The deadline-monotonic algorithm requires a task set that is preemptable and independent, and it uses static task priorities.
- The deadline-monotonic algorithm is optimal for synchronous task systems on one processor.
- There is no simple condition that is both necessary and sufficient for determining if a task system is schedulable by deadline-monotonic scheduling, but a utilization bound can be obtained using the equations in the previous section.
- A synchronous task system on one processor can be tested by constructing the schedule through the longest task period, and this can be accomplished in pseudo-polynomial time. Either graphical or analytical methods can be used.

Rate-Monotonic Scheduling

- The rate-monotonic priority assignment algorithm is a static priority assignment algorithm that assigns task priorities in the order of increasing periods.
- For task systems in which deadlines equal periods ($d_n = p_n$ for all tasks τ_n), the rate-monotonic and deadline-monotonic algorithms are equivalent, which results in the following conclusion.

  For task systems in which $d_n = p_n$ for all tasks τ_n, rate-monotonic priority assignment is an optimal static assignment algorithm for synchronous task systems on one processor

- Using the concept of a critical instant, a worst-case performance bound for rate monotonic scheduling can be developed.

Rate-Monotonic Scheduling

- The set of tasks S_{ng} consisting of those tasks with priorities *greater than or equal to* that of task τ_n, and with periods *greater than or equal to* the deadline of task τ_n is empty. That is,$S_{ng} = \{\}$, and $N_{ng} = 0$.

- The set of tasks S_{nl} consisting of those tasks with priorities *higher* than that of task τ_n, but with *periods less than* the deadline of task τ_n contains all tasks with priorities greater than that of task τ_n. That is, $S_{nl} = \{\tau_1, \tau_2, \cdots, \tau_{n-1}\}$ and $N_{nl} = n - 1$.

- The effective utilization of task τ_n, then reduces to

$$E_n = \sum_{i=1}^{N} \frac{e_i}{p_i} + \frac{b_n}{p_n} \qquad \text{(EQ 5.5.1)}$$

Rate-Monotonic Scheduling

- The utilization bound becomes simply

$$E_n = n\left(2^{\frac{1}{n}} - 1\right) \qquad \text{(EQ 5.5.2)}$$

Furthermore, in the situation in which there is no blocking ($b_n = 0$), if task τ_n meets its deadline then so also do all higher priority tasks.

- The rate-monotonic scheduling algorithm will produce a valid schedule for a synchronous system of N independent tasks if -but only if-the following inequality holds.

$$U \leq N\left(2^{\frac{1}{N}} - 1\right) \qquad \text{(EQ 5.5.3)}$$

- In the limit as N grows large, the inequality approaches $In\ 2 = 0.692$, implying that processor utilization can be limited to values as low as 69% solely by the scheduling process.

Rate-Monotonic Scheduling

- The inequality of Equation 5.5.3 is a worst-case bound, and hence can serve to determine that a task system is schedulable. It cannot be used to determine if a task system is *not* schedulable.

Example 5.5.1: A simple example of rate-monotonic task scheduling.

A simple task system consisting of three periodic tasks is specified below.

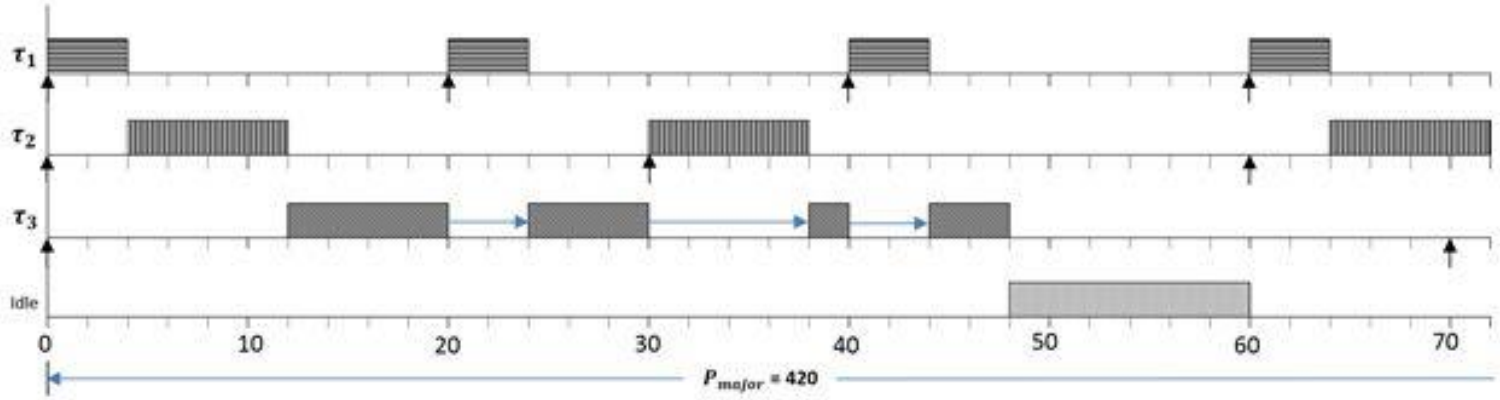
$$N = number\ of\ tasks = 3$$

Period	Execution Time
$p_1 = 20$	$e_1 = 4$
$p_2 = 30$	$e_2 = 8$
$p_3 = 70$	$e_3 = 20$

The processor utilization is determined to be $U = \frac{4}{20} + \frac{8}{30} + \frac{20}{70} = 0.752.$

The utilization bound is $3\left(2^{\frac{1}{3}} - 1\right) = 0.780.$

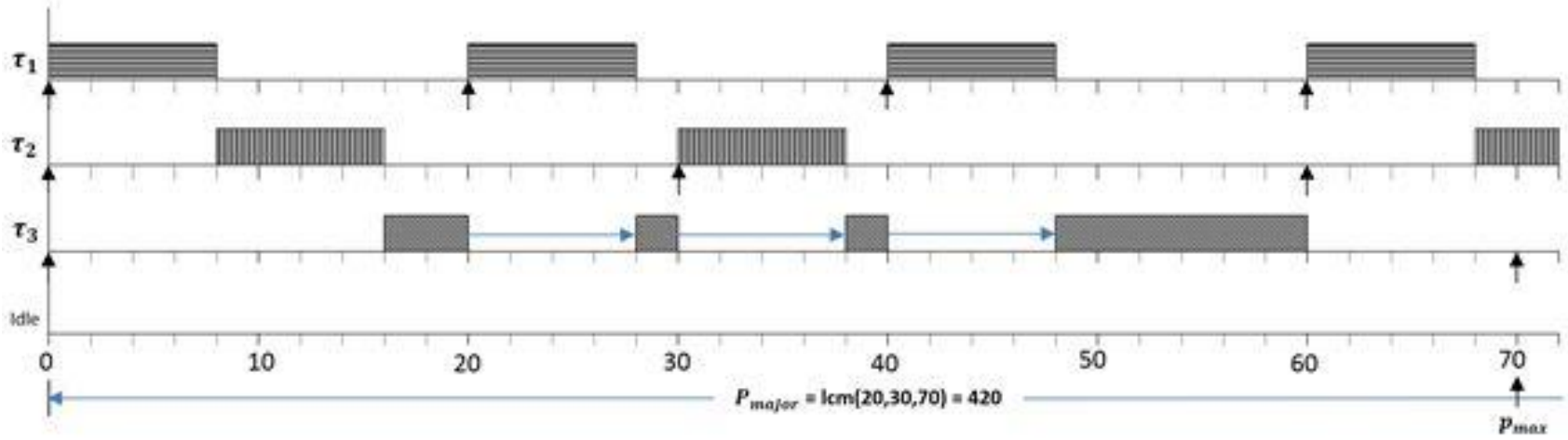
Rate-Monotonic Scheduling



- Since the processor utilization is less than the utilization bound, this task system is schedulable using rate-monotonic priority assignment.

- This conclusion is verified by the rate monotonic schedule depicted in Figure 5.19, in which the low priority task τ_3 is preempted twice but still completes prior to its deadline at $t = 70$.

Rate-Monotonic Scheduling



- If the execution time of task τ_1 is increased from its value of 4 to a value of 8, the processor utilization increases to a value $U = \frac{8}{20} + \frac{8}{30} + \frac{20}{70} = 0.952$. Since this value is greater than that of the utilization bound, no conclusion can be drawn concerning the schedulability of the task system.
- The schedulability question can be answered by constructing the schedules using the rate-monotonic priority assignment. The resulting schedule is shown in Figure 5.20 and shows that all tasks meet their first deadlines, and hence that the task system is schedulable.

Rate-Monotonic Scheduling

- It should be noted that the schedule of Figure 5.20 shows no processor idle time, yet the processor utilization was computed to be 0.952.
- The cyclic schedules produced by these algorithms repeat at the major cycle period. Yet, the schedule construction concern only the first release time of each task. The schedule need be constructed only through the longest period.
- The reason it is not necessary to examine the entire major cycle is that, for synchronous task systems, if a task meets its first deadline it will meet all deadlines for all releases. It is entirely possible for the processor utilization over the first duration of the longest period to be different from that over a complete major cycle.

Rate-Monotonic Scheduling

Example 5.5.2: A second example of rate monotonic scheduling.

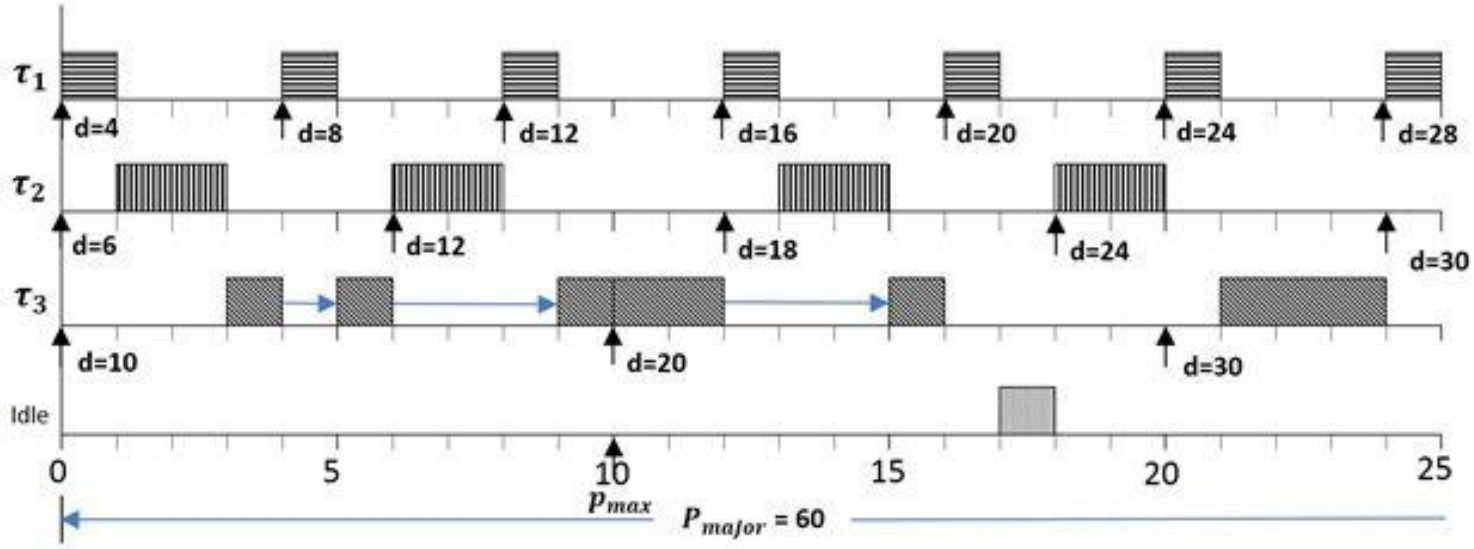
A second example of rate-monotonic scheduling-and one that illustrates the phenomenon described above will be examined. In the treatment of this problem, the schedule over the longest period will be compared to the schedule as it exists over the complete major cycle. Since the pattern of task execution is periodic at the major cycle rate, the task utilization over the major cycle determines the long term utilization. The task system to be considered is described below.

$N = number\ of\ tasks = 3$

Period	Execution Time
$p_1 = 4$	$e_1 = 1$
$p_2 = 6$	$e_2 = 2$
$p_3 = 10$	$e_3 = 3$

The processor utilization is determined to be $U = \frac{1}{4} + \frac{2}{6} + \frac{3}{10} = 0.883.$

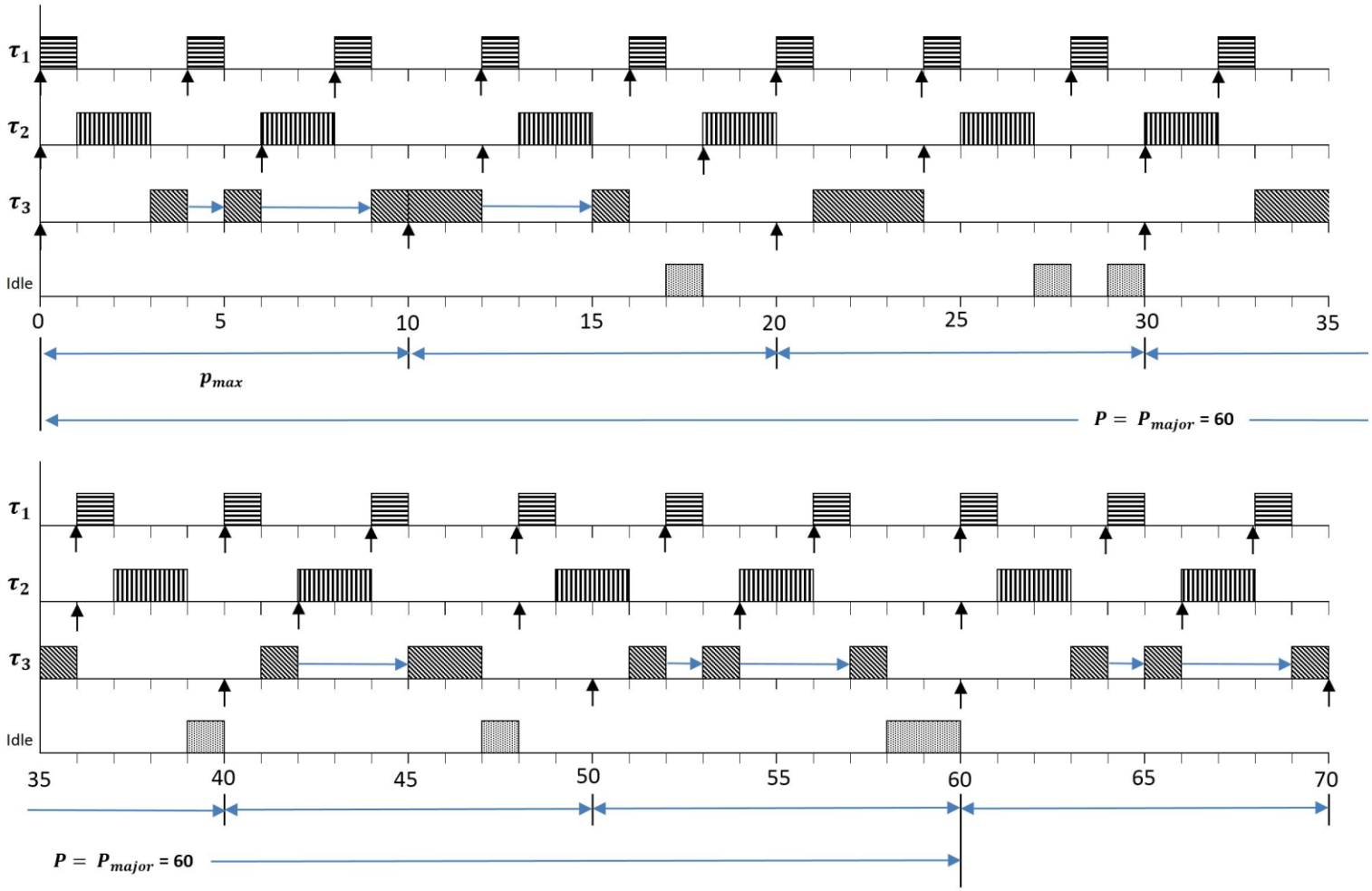
Rate-Monotonic Scheduling



- The utilization bound for three tasks using rate-monotonic scheduling is 0.780, making the utilization bound test inconclusive. Schedule construction over the longest period yields the result depicted in Figure 5.21 and indicates that the task system is schedulable.

- There is no processor idle time indicated in this schedule, even though the processor utilization was calculated to be 0.883.

Rate-Monotonic Scheduling



A complete major cycle is depicted in Figure 5.22 and shows processor idle time appearing over the remainder of the major cycle.

Rate-Monotonic Scheduling

- A summary of rate-monotonic task scheduling is presented below.
  - The rate-monotonic algorithm requires a task set that is preemptable and independent, and uses static task priorities.
  - Under the condition that deadlines equal periods, the rate-monotonic priority assignment is an optimal static priority assignment for *synchronous task systems on one processor*.
  - There is no simple condition that is both necessary and sufficient for determining if a task system is schedulable by rate-monotonic scheduling, but a utilization bound can be simply stated as in Equation 5.5.3.
  - A *synchronous task system on one processor* can be tested by constructing the schedule through the longest task period, and this can be accomplished in pseudo polynomial time. Alternatively, analytic completion time algorithms can be used.

Earliest Deadline Scheduling

- The deadline-monotonic and rate-monotonic algorithms are static priority assignment algorithms. This means that the task priorities do not change during system operation when the task system is executing under control of a multitasking executive (RTOS).
- In the case of a cyclic executive, a static schedule is constructed in advance for the system and the task priorities do not change as the schedule is being developed.
- The earliest deadline algorithm, on the other hand, is a dynamic priority assignment algorithm, which means that the assigned priorities can change during execution by an RTOS, or during the development of a static cyclic schedule.

Earliest Deadline Scheduling

- The earliest deadline priority assignment is made as follows.

  At each instant of time, task priorities are assigned in increasing order of currently impending deadlines.

  The analogy to the Critical Path Scheduling algorithm from the aperiodic scheduling section is evident.

- Application of the algorithm requires a continuous determination of the "*time-to-deadline*" for each task in the task system, and task priorities are continuously assigned according to these times.

- What does it mean for a real-time operating system? the scheduler, takes these decisions (after re-computing task priorities) upon each *tick of the real time clock*.

Earliest Deadline Scheduling

- The earliest deadline algorithm has optimal properties similar to the prior algorithms (always produces a *valid* schedule for every task system that is *feasible)*.

  The earliest deadline algorithm is an optimal dynamic priority assignment algorithm for scheduling periodic tasks on a single processor.

- That the earliest deadline algorithm is optimal is particularly significant in view of the fact that

$$U = \sum_{n=1}^{N} \frac{e_n}{p_n} \leq M$$

  represents a *necessary and sufficient* condition for a synchronous task system to be feasible (if there is a *valid* schedule for the task system) when deadlines equal periods.

Earliest Deadline Scheduling

For synchronous task systems in which deadlines equal periods, a necessary and sufficient condition for a task system to be feasible is $U \leq 1$ *. If these conditions are met, earliest deadline scheduling will produce a valid schedule.*

- The term *feasible* is extremely important, implying no requirement for static priorities, as opposed to the term *schedulable* (which requires a schedule based on static priorities).

- Further, for *arbitrary task systems* that are not synchronous and also with task deadlines that differ from task periods, the earliest deadline algorithm is still optimal with respect to dynamic priority assignment on a single processor.

Earliest Deadline Scheduling

Example 5.6.1: An application of earliest deadline scheduling.

The application of earliest deadline scheduling to the task system specified below will be used to determine if the task system is feasible.

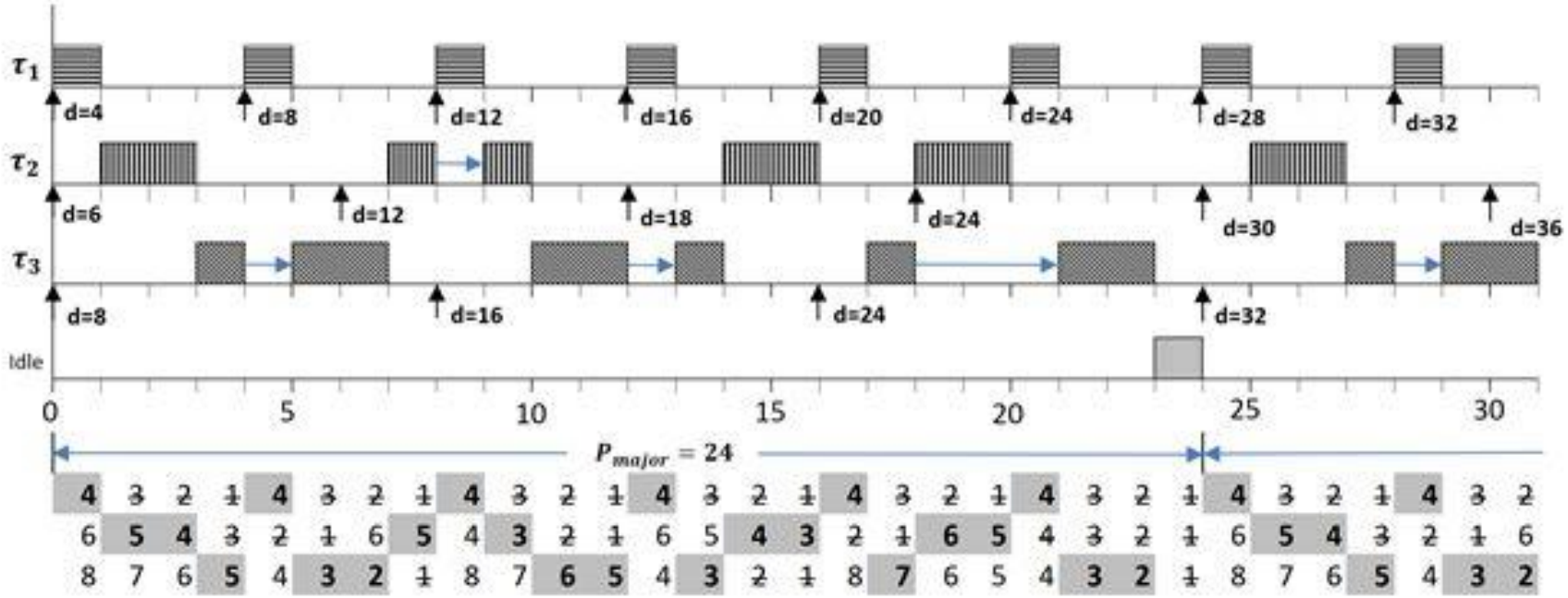
$$N = number\ of\ tasks = 3$$

Period	Deadline	Execution Time
$p_1 = 4$	$d_1 = 4$	$e_1 = 1$
$p_2 = 6$	$d_2 = 6$	$e_2 = 2$
$p_3 = 8$	$d_3 = 8$	$e_3 = 3$

The processor utilization is determined to be $U = \frac{1}{4} + \frac{2}{6} + \frac{3}{8} = 0.958 < 1.$

The rate-monotonic utilization bound for three tasks is 0.780; the test is inconclusive. Since the earliest deadline algorithm is an optimal dynamic priority assignment, application of this algorithm will produce a valid schedule if the task system is feasible.

Earliest Deadline Scheduling



The earliest deadline schedule is depicted in Figure 5.23 and indicates that all tasks meet their first deadline.

This task system is therefore *feasible* on a single processor - it can be scheduled on a single processor - but is not schedulable on a single processor.

Earliest Deadline Scheduling

- That is, there is no *static* priority assignment that will produce a valid schedule. That the task system is not schedulable may be verified by attempting to apply the rate-monotonic algorithm - it will not result in a valid schedule.
- Since the rate-monotonic algorithm is an optimal static priority assignment algorithm, if it does not produce a valid schedule, neither will any other static priority algorithm.

- For *asynchronous* task systems, the complexity of the task scheduling problem increases considerably. *There is no optimal static priority assignment*, as shown in the case of synchronous task systems. The deadline-monotonic algorithm, is not optimal for asynchronous task systems for *any* number of processors.

Multiprocessor Scheduling

- When periodic tasks are to be scheduled on more than one processor, the utilization bound equations apply and are repeated here as Equations 5.7.1 and 5.7.2.
- Equation 5.7.1 is a *necessary but not sufficient* condition, while Equation 5.7.2 is a *sufficient but not necessary* condition.
- When deadlines equal periods, these inequalities are equivalent and represent a necessary and sufficient condition for feasibility.

$$U = \sum_1^N \frac{e_n}{p_n} \leq M \qquad \text{(EQ 5.7.1)}$$

$$D = \sum_1^N \frac{e_n}{d_n} \leq M \qquad \text{(EQ 5.7.2)}$$

- If the feasibility of a task system is not evident from the utilization bound of Equation 5.7.2, the only known manner in which feasibility can be shown is construction of a valid schedule (There is no optimal algorithm!!).

Multiprocessor Scheduling

- The usual approach to schedule construction is the use of some systematic technique as a heuristic method, such as rate-monotonic scheduling, deadline-monotonic scheduling, earliest deadline scheduling, or some other technique.

- Two possible methods for determining which tasks will be assigned to which processors.
  - In the *non-partitioning method* the M processors are treated as a single resource, and the M tasks with the highest priorities are always executing, with the priorities assigned by whatever heuristic method has been chosen.
  - In the *partitioning method*, tasks are partitioned into separate groups, possibly in some optimal or near optimal manner, and each group is then scheduled on a single processor. Tasks are assigned static priorities within a group using whatever heuristic method has been chosen.

Multiprocessor Scheduling

- The question of just how much of the infinite periodic schedule need be constructed to prove feasibility, must be addressed. For task systems with *static* priorities.
  - For *synchronous* task systems the schedule is periodic with period $P = lcm(p_1, p_2, \ldots, p_N)$, so that the schedule must be constructed through time $t = P$.
  - For *asynchronous* task systems the schedule becomes periodic at time $r_{max} + P$, where r_{max} is equal to the largest of the initial release times of all the tasks in the system. Thus, it is necessary to construct the schedule through time $t = r_{max} + 2P$.
- For synchronous task systems on one processor it is necessary to construct the schedule only through the longest period. The additional complexity of multiprocessor scheduling requires schedule construction over the longer base period P.

Multiprocessor Scheduling

Example 5.7.1: Scheduling tasks on two processors by the partitioning and non-partitioning methods.

$N = number\ of\ tasks = 4$

M = number of processors = 2

Period	Deadline	Execution Time
$p_1 = 2$	$d_1 = 2$	$e_1 = 1$
$p_2 = 3$	$d_2 = 3$	$e_2 = 2$
$p_3 = 4$	$d_3 = 4$	$e_3 = 2$
$p_4 = 6$	$d_4 = 6$	$e_4 = 2$

The processor utilization is determined to be

$$U = \frac{1}{2} + \frac{2}{3} + \frac{2}{4} + \frac{2}{6} = 2.0 \leq 2.$$

Even though the task system is known to be feasible, on the basis that deadlines equal periods and that it satisfies this utilization bound, there is no known algorithm guaranteed to produce a valid schedule. Heuristic algorithms can be applied but with no guarantee of success.

Multiprocessor Scheduling

If deadline-monotonic scheduling is selected as the heuristic algorithm, then either a partitioning or non-partitioning approach must be used.

Partitioning Approach

In constructing this schedule, the following partitioning was used.

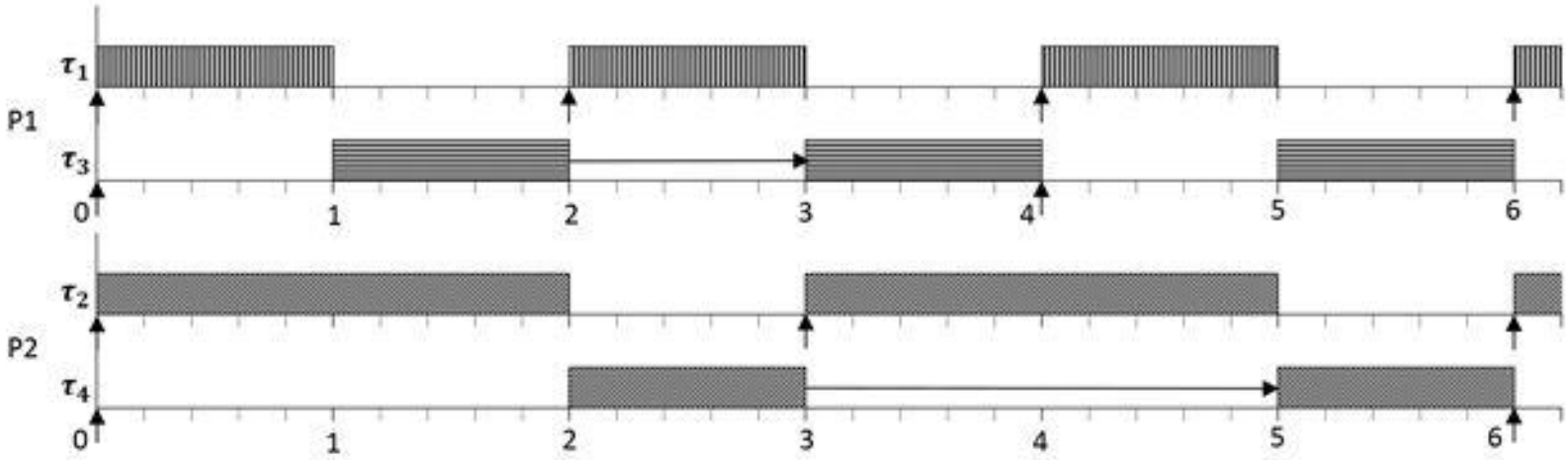
- Tasks τ_1 and τ_3 are assigned to processor P_1 for execution.
- Tasks τ_2 and τ_4 are assigned to processor P_2 for execution.

That is, tasks τ_1 and τ_3 are scheduled on processor P_1 using deadline-monotonic scheduling

independently of the scheduling of tasks τ_2 and τ_4 on processor P_2, which is also accomplished using deadline-monotonic scheduling.

(since deadlines equal periods in this example, deadline-monotonic and rate-monotonic priority assignments are equivalent)

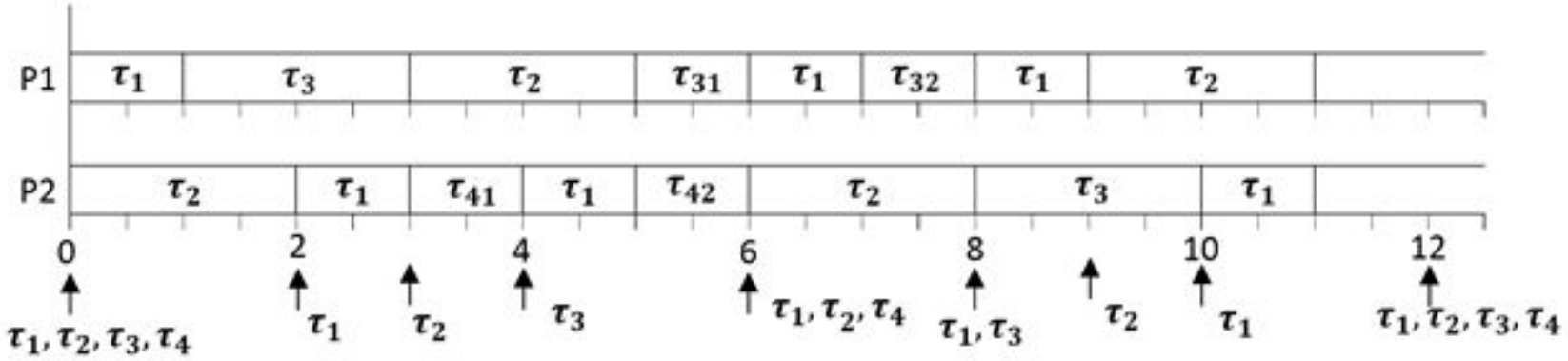
Multiprocessor Scheduling



- The general requirement of a valid schedule for a synchronous task system on multiple processors is that all tasks meet their deadlines in the interval $P = lcm(2,3,4,6) = 12$, the base period of the task system.
- Two synchronous task systems that are independently scheduled on the two processors, the single processor requirement for schedule validity can be used - schedule need only be constructed through the longest period - time $t = 4$ for processor P_1, and time $t = 6$ for processor P_2. All tasks meet their first deadlines, and hence the schedule is valid.

Multiprocessor Scheduling

Non-Partitioning Method



- The combination of the two processors is considered a common pool of computational resources which are assigned to execute the tasks. Two highest priority tasks are executing at any particular time.
- The deadline-monotonic priority assignment fails to produce a valid schedule. At time $t = 6$, task τ_4 is released, but higher priority tasks τ_1, τ_2, and τ_3 consume all execution time until time $t = 11$, at which point there is not enough time on either processor for τ_4 to complete by its deadline at time $t = 12$.

Multiprocessor Scheduling

Example 5.7.2: A second example of scheduling tasks on two processors.

The second example of the scheduling of tasks on multiple processors using both the partitioning and non-partitioning methods involves the following task system.

$N = number\ of\ tasks = 4$

M = number of processors = 2

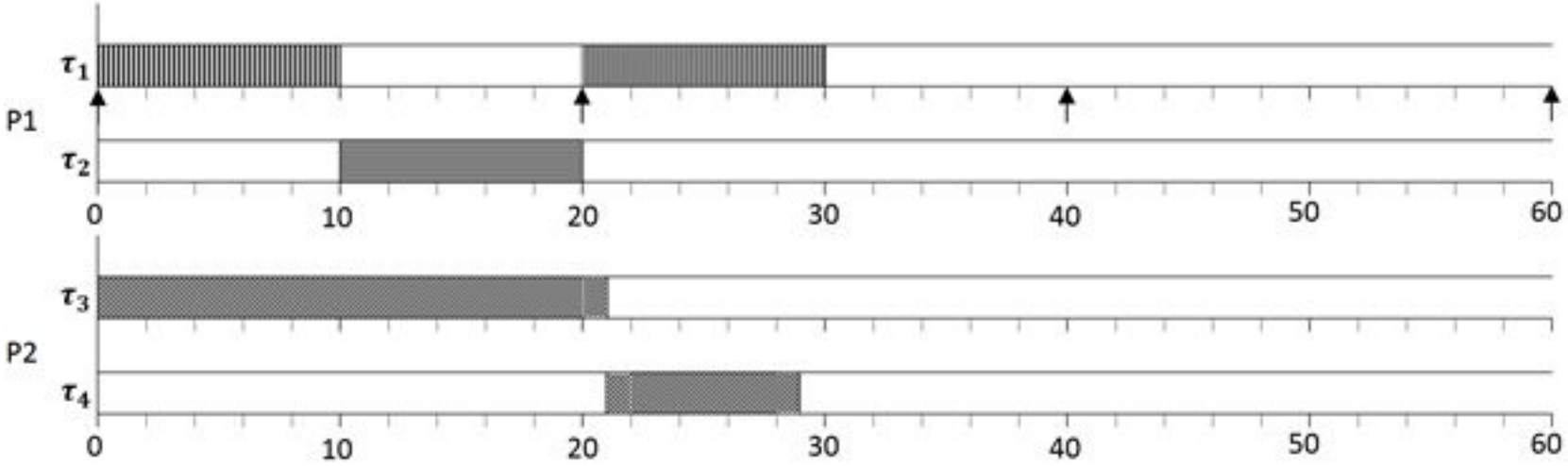
Period	Deadline	Execution Time
$p_1 = 20$	$d_1 = 20$	$e_1 = 10$
$p_2 = 30$	$d_2 = 30$	$e_2 = 11$
$p_3 = 30$	$d_3 = 30$	$e_3 = 21$
$p_4 = 40$	$d_4 = 40$	$e_4 = 8$

The processor utilization is determined to be

$$U = \frac{10}{20} + \frac{11}{20} + \frac{21}{30} + \frac{8}{40} = 1.767 \leq 2.$$

Since deadlines = periods, this result is a necessary and sufficient condition for feasibility of the task system on two processors. Use Deadline-monotonic scheduling

Multiprocessor Scheduling



Tasks τ_1 and τ_2 are assigned to processor P_1 for execution.

Tasks τ_3 and τ_4 are assigned to processor P_2 for execution.

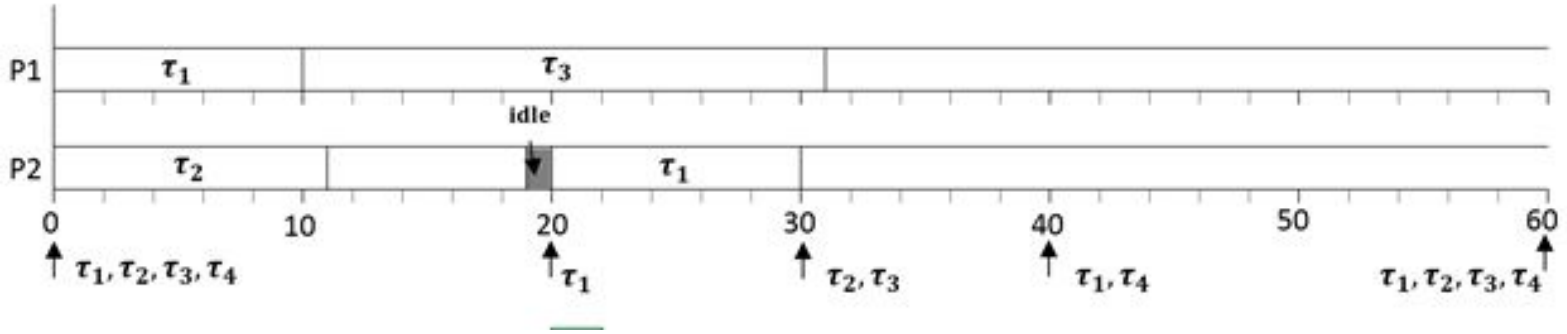
The individual processor utilizations are

$$U_1 = \frac{10}{20} + \frac{11}{20} = 0.867 \leq 1.$$

$$U_2 = \frac{21}{30} + \frac{8}{40} = 0.9 \leq 1.$$

Schedule fails when task τ_2 misses its first deadline at time $t = 30$

Multiprocessor Scheduling

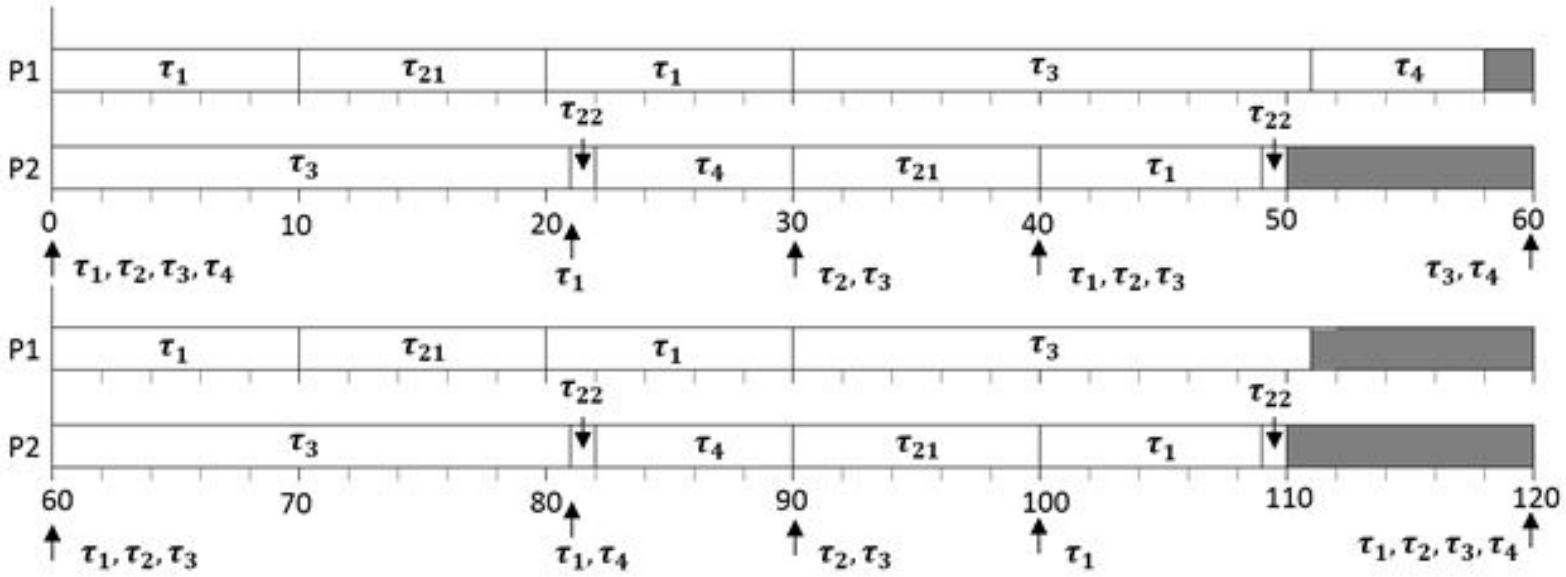


The deadline-monotonic scheduling algorithm allows ties to be arbitrarily broken, and in this case the tie is broken to produce the priority list $(\tau_1, \tau_2, \tau_3, \tau_{4)}$.

The resulting schedule attempt is shown in the diagram of Figure 5.27, which shows task τ_3 missing its first deadline at time $t = 30$.

It would appear that deadline-monotonic scheduling used as a heuristic algorithm fails in this case.

Multiprocessor Scheduling



If the priority list is changed to $(\tau_1, \tau_3, \tau_2, \tau_4)$, the valid schedule shown in Figure 5.28 results.

The fact that one priority list produces a valid schedule while the other does not merely points out that the deadline-monotonic priority assignment is not optimal in this situation. **Will a dynamic priority algorithm work?**

Multiprocessor Scheduling

Example 5.7.3: Application of the earliest deadline method to the scheduling of tasks on two processors.

The algorithm is applied in a manner that treats the multiple processors as a common pool of computational resources, *i.e.* the non-partitioning method is used. The task system for this example is specified below.

$N = number\ of\ tasks = 3$

M = number of processors = 2

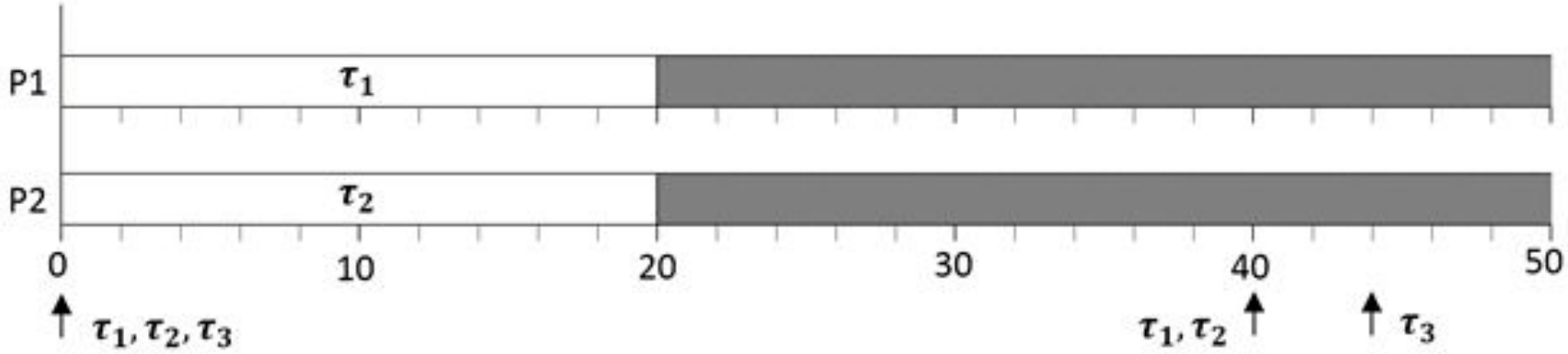
Period	Deadline	Execution Time
$p_1 = 40$	$d_1 = 40$	$e_1 = 20$
$p_2 = 40$	$d_2 = 40$	$e_2 = 20$
$p_3 = 44$	$d_3 = 44$	$e_3 = 40$

The processor utilization is determined to be

$$U = \frac{20}{40} + \frac{20}{40} + \frac{40}{44} = 1.909 \leq 2.$$

The since deadlines = periods, this condition is both necessary and sufficient, and hence the task system is feasible. In an attempt to find a valid schedule, the earliest deadline priority assignment algorithm will be used.

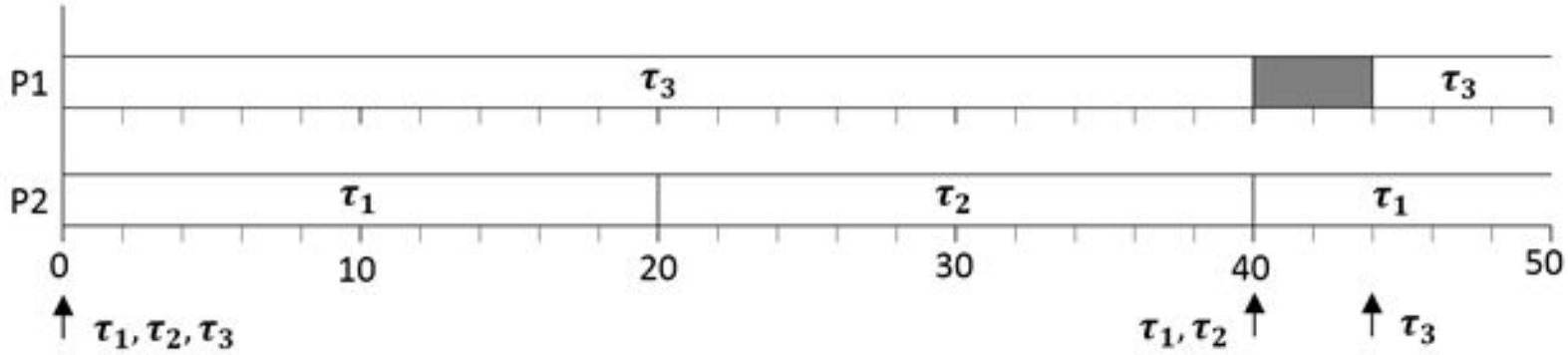
Multiprocessor Scheduling



The result is shown in Figure 5.29, where it can be seen that task τ_3 misses its first deadline.

Thus, the earliest deadline priority assignment algorithm fails to produce a valid schedule, even though the task system is feasible.

Multiprocessor Scheduling



A valid schedule for this task set is easily obtainable by assigning task τ_3 to processor P_1 and tasks τ_1 and τ_2 to processor P_2, as illustrated in the task schedule shown in Figure 5.30.

If the earliest deadline priority assignment algorithm were optimal, it would have produced a valid schedule.

Even if a task system is known to be feasible on M processors, there is no known algorithm guaranteed to produce a valid schedule, *i.e.* there is no known optimal algorithm.

Least Slack Time Priority Algorithm

Another dynamic priority assignment algorithm will be described: the *least slack time* priority assignment algorithm. This algorithm requires that task priorities be assigned in the following manner.

At any instant of time, tasks are assigned priorities in increasing order of slack times. The slack time of a task τ_n at time t is defined as

$$t_s = t_d - e(t) - t \qquad \text{(EQ 5.7.3)}$$

where t_d is the time of the impending deadline of task $\tau_n$and $e(t)$ is execution time remaining in order to complete task τ_n.

The slack time t_d is thus the difference between the time available to meet the impending deadline and the execution time required to complete the task.

Least Slack Time Priority Algorithm

The least slack time algorithm assigns tasks to processors using the concept of *processor sharing* introduced earlier.

The least slack time algorithm is not an optimal algorithm for scheduling periodic tasks on multiple processors (remember there is no such polynomial time scheduling algorithm) but has some desirable features

- If the earliest deadline algorithm produces a valid schedule, so also does the least slack time algorithm.
- In the single processor case, the least slack time algorithm is optimal.
- For multiple processors there are task systems for which the least slack time algorithm will produce a valid schedule when the earliest deadline algorithm will not.
- The length of schedule construction required to determine if a schedule produced by the least slack time algorithm is valid is identical to that required of the earliest deadline algorithm or of a static priority algorithm.

Least Slack Time Priority Algorithm

Example 5.7.4: Least slack time scheduling.

Processor sharing by definition treats the multiple processors as a common pool of computational resources, *i.e.* the non-partitioning method is used. The task system for this example is specified below.

$N = number\ of\ tasks = 3$

M = number of processors = 2

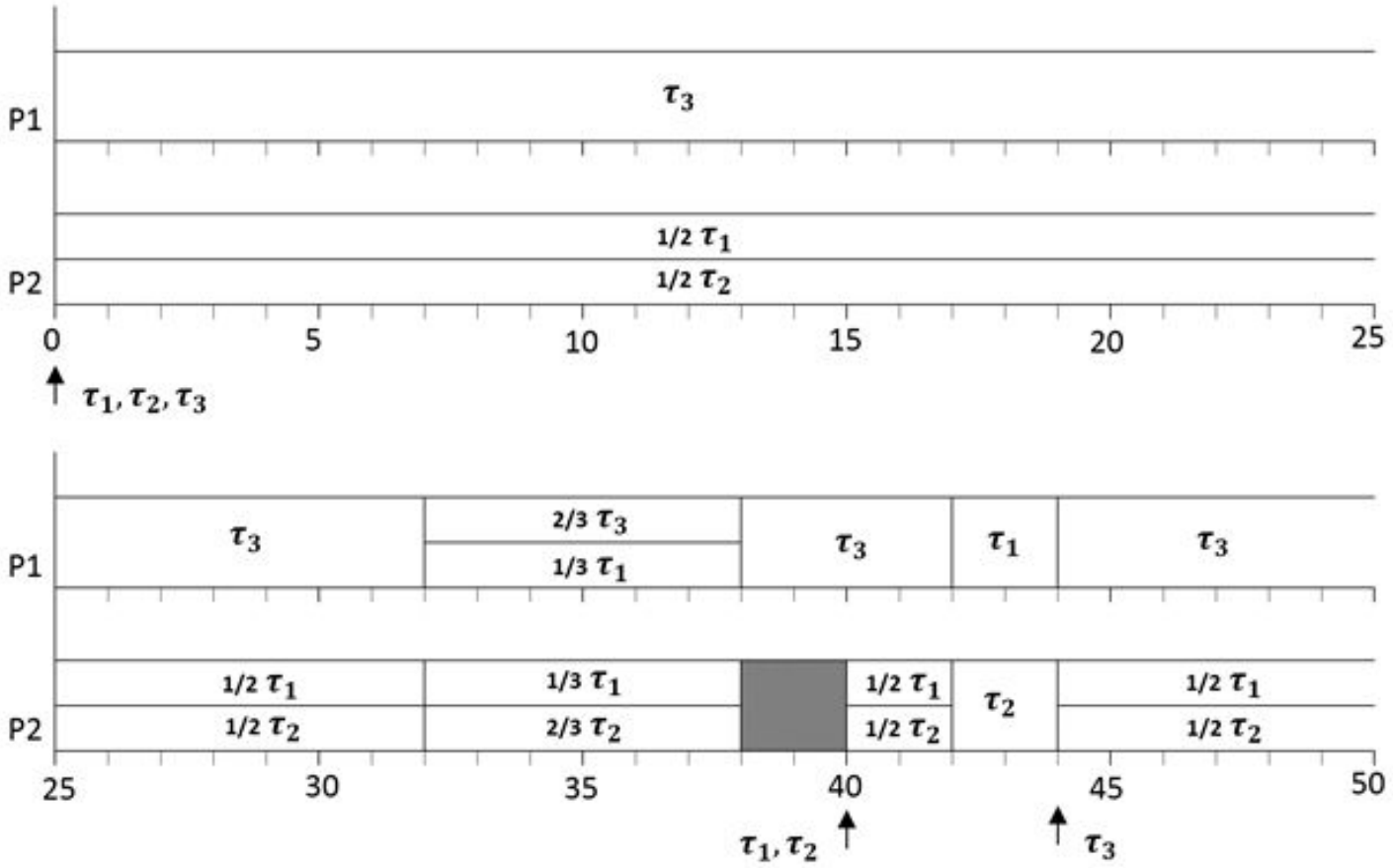
Period	Deadline	Execution Time
$p_1 = 40$	$d_1 = 40$	$e_1 = 20$
$p_2 = 40$	$d_2 = 40$	$e_2 = 20$
$p_3 = 44$	$d_3 = 44$	$e_3 = 40$

The processor utilization is determined to be

$$U = \frac{20}{40} + \frac{20}{40} + \frac{40}{44} = 1.909 \leq 2.$$

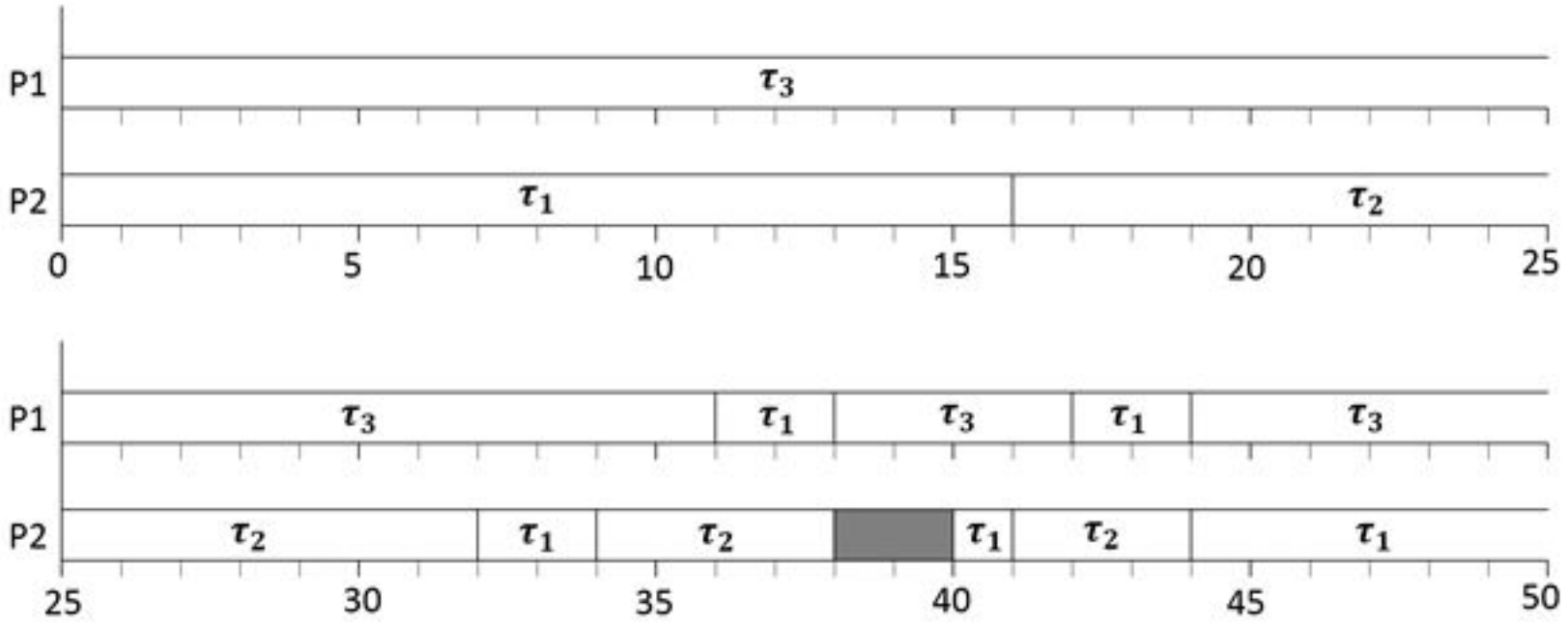
The since deadlines = periods, this condition is both necessary and sufficient, and hence the task system is feasible.

Least Slack Time Priority Algorithm



Processor sharing schedule using least slack time scheduling

Least Slack Time Priority Algorithm



Equivalent realizable schedule.

The significance is that the least slack time algorithm produced a valid schedule for this task system when the earliest deadline algorithm failed to do so.

Periodic Task Scheduling Summary

The scheduling of periodic tasks is an important component of the design of real-time systems software.

Periodic task systems can be either preemptive or non-preemptive, and this characteristic has a profound effect on the ability to produce task schedules that meet periodic timing requirements.

- In general, a task system can be characterized by specifying-for each task: a period, a deadline, and an execution time. If blocking is possible, then a worst case blocking time must also be specified. Execution times will normally vary from one release of a task to another, a phenomenon known as jitter. As a result, the execution times of tasks are often expressed as worst-case execution times.
- Tasks can be scheduled statically using a cyclic executive approach. Static task schedules are predetermined through the use of task scheduling algorithms that employ the concept of task priority to determine how the various tasks are assigned to the processor(s) as a function of time.

Periodic Task Scheduling Summary

- The resulting schedule exhibits periodic behavior at a period called the major cycle. Each major cycle is identical to the next. The major cycle is comprised of some number of minor cycles. The processing within different minor cycles is different, and the actual code executed within any given minor cycle is called a frame.

- Synchronization requirements are met by dividing tasks into subtasks at the synchronization points, and then scheduling the subtasks in such a manner that the synchronization requirements are met. Cyclic executive systems are scheduled by periodic, timer-driven interrupts.

- Tasks can be scheduled dynamically through use of a real-time multitasking executive (RTOS). Tasks are assigned priorities and are dynamically scheduled as the task system executes by the RTOS, which ensures that the highest priority pending task is always selected for execution.

Periodic Task Scheduling Summary

- Schedulability analysis is conducted by assuming a fixed execution time for each of the tasks and applying appropriate analysis methods. This is necessary for ensuring that a given task system will meet all deadlines for all task releases. In many situations, the only known analysis method is schedule construction.

- Synchronization requirements are met through the use of primitive operations, such as semaphores, provided by the real-time operating system.

- In general, producing a static schedule for a cyclic executive system or equivalently, determining if a task system is schedulable using a specified priority assignment in a dynamically scheduled system can be a difficult problem.

- Arbitrary task preemption increases scheduling flexibility. Arbitrary preemption is a requirement in those few situations in which efficient schedulability analysis is possible.

Periodic Task Scheduling Summary

- Synchronous task systems with arbitrary task preemption exhibit a pseudo polynomial time algorithm for determining if a particular priority assignment will produce a valid schedule on a single processor. This algorithm consists of constructing the schedule through the longest task period-or of the analytical equivalent of schedule construction.

- In many cases of practical importance, upper bounds on processor utilization relating to schedulability can be formulated and effectively used. Meeting the bound requirement is a sufficient but not necessary condition of schedulability for a task system.

- In many situations of practical importance, the only known algorithm for determining if a given task system is feasible or schedulable is to construct the schedule using a known optimal priority assignment. For scheduling on multiple processors there is no known optimal priority assignment.

Periodic Task Scheduling Summary

- There are optimal priority assignments for some classes of task systems. An optimal priority assignment is a priority assignment that will produce a valid schedule if one exists.
  - The deadline-monotonic priority assignment is an optimal static priority assignment for synchronous task systems on one processor.
  - The rate-monotonic priority assignment is an optimal static priority assignment for synchronous task systems on one processor when deadlines equal periods.
  - The earliest deadline priority assignment is an optimal dynamic priority assignment for scheduling tasks, synchronous or asynchronous, on one processor.

- Alternatively, priority assignments used in schedule construction for statically scheduled systems, or those assigned in order to analyze task behavior in a dynamically scheduled system, are used as heuristic approaches to the problem.