Real-Time Operating Systems (0_KRI)
Real-Time Scheduling Models

Ivan Cibrario Bertolotti

IEIIT-CNR / Politecnico di Torino

Academic Year 2006-2007

Outline

Concurrent Programming and Real-Time

- In any concurrent program, the exact order in which processes execute is not specified.
- According to the concurrent programming theory, synchronization primitives are used to enforce the local ordering constraints needed to ensure that the semantics of the program is correct.
- For example, a semaphore can be used to ensure mutual exclusion, thus avoiding race conditions, when appropriate.

Despite these constraints, the general behavior of the program still exhibits significant non-determinism due to interleaving.

Concurrent Programming and Real-Time

- In any concurrent program, the exact order in which processes execute is not specified.
- According to the concurrent programming theory, synchronization primitives are used to enforce the local ordering constraints needed to ensure that the semantics of the program is correct.
- For example, a semaphore can be used to ensure mutual exclusion, thus avoiding race conditions, when appropriate.

Despite these constraints, the general behavior of the program still exhibits significant non-determinism due to interleaving.

Concurrent Programming and Real-Time

- In any concurrent program, the exact order in which processes execute is not specified.
- According to the concurrent programming theory, synchronization primitives are used to enforce the local ordering constraints needed to ensure that the semantics of the program is correct.
- For example, a semaphore can be used to ensure mutual exclusion, thus avoiding race conditions, when appropriate.

Despite these constraints, the general behavior of the program still exhibits significant non-determinism due to interleaving.

Concurrent Programming and Real-Time

- In any concurrent program, the exact order in which processes execute is not specified.
- According to the concurrent programming theory, synchronization primitives are used to enforce the local ordering constraints needed to ensure that the semantics of the program is correct.
- For example, a semaphore can be used to ensure mutual exclusion, thus avoiding race conditions, when appropriate.

Despite these constraints, the general behavior of the program still exhibits significant non-determinism due to interleaving.

Timings Come into Place

- While the concurrent program's output will be identical with all the possible interleavings (provided it is correct), the timing behavior may vary considerably.
- Hence, for example, if one of the concurrent processes has a tight deadline, only some of the interleavings will meet its temporal requirements, even if its results will have the right value anyway.

A real-time system needs to further restrict the non-determinism found within concurrent systems, because some interleavings that are correct with respect to semantics can be unacceptable with respect to timings.

Timings Come into Place

- While the concurrent program's output will be identical with all the possible interleavings (provided it is correct), the timing behavior may vary considerably.
- Hence, for example, if one of the concurrent processes has a tight deadline, only some of the interleavings will meet its temporal requirements, even if its results will have the right value anyway.

A real-time system needs to further restrict the non-determinism found within concurrent systems, because some interleavings that are correct with respect to semantics can be unacceptable with respect to timings.

Timings Come into Place

- While the concurrent program's output will be identical with all the possible interleavings (provided it is correct), the timing behavior may vary considerably.
- Hence, for example, if one of the concurrent processes has a tight deadline, only some of the interleavings will meet its temporal requirements, even if its results will have the right value anyway.

A real-time system needs to further restrict the non-determinism found within concurrent systems, because some interleavings that are correct with respect to semantics can be unacceptable with respect to timings.

The Role of the Scheduling Model

The main goal of a real-time scheduling model is to ensure that a concurrent program is correct with respect to timings, too.

In order to do this, it provides two main features:

1. A scheduling algorithm for ordering the use of system resources, the processor(s) in particular.
2. A means of predicting the worst-case behavior of the system, with respect to timings, when that scheduling algorithm is applied.

The Role of the Scheduling Model

The main goal of a real-time scheduling model is to ensure that a concurrent program is correct with respect to timings, too.

In order to do this, it provides two main features:

1. A scheduling algorithm for ordering the use of system resources, the processor(s) in particular.
2. A means of predicting the worst-case behavior of the system, with respect to timings, when that scheduling algorithm is applied.

The Role of the Scheduling Model

The main goal of a real-time scheduling model is to ensure that a concurrent program is correct with respect to timings, too.

In order to do this, it provides two main features:

1. A scheduling algorithm for ordering the use of system resources, the processor(s) in particular.
2. A means of predicting the worst-case behavior of the system, with respect to timings, when that scheduling algorithm is applied.

General-Purpose vs. Real-Time Scheduling

- When choosing a scheduling algorithm, it is often necessary to look for a compromise between optimizing the mean system performance and its determinism.
- Since they are less concerned with determinism, most general-purpose scheduling algorithms emphasize fairness and efficiency, and optimize the system throughput.
- On the other hand, real-time scheduling algorithms must put the emphasis on timings, even if this entails a greater overhead and the mean performance of the system becomes worse.
- In order to do this, the scheduling algorithms can take advantage of the greater amount of information that, on most real-time systems, is available on the processes to be executed.

General-Purpose vs. Real-Time Scheduling

- When choosing a scheduling algorithm, it is often necessary to look for a compromise between optimizing the mean system performance and its determinism.
- Since they are less concerned with determinism, most general-purpose scheduling algorithms emphasize fairness and efficiency, and optimize the system throughput.
- On the other hand, real-time scheduling algorithms must put the emphasis on timings, even if this entails a greater overhead and the mean performance of the system becomes worse.
- In order to do this, the scheduling algorithms can take advantage of the greater amount of information that, on most real-time systems, is available on the processes to be executed.

General-Purpose vs. Real-Time Scheduling

- When choosing a scheduling algorithm, it is often necessary to look for a compromise between optimizing the mean system performance and its determinism.
- Since they are less concerned with determinism, most general-purpose scheduling algorithms emphasize fairness and efficiency, and optimize the system throughput.
- On the other hand, real-time scheduling algorithms must put the emphasis on timings, even if this entails a greater overhead and the mean performance of the system becomes worse.
- In order to do this, the scheduling algorithms can take advantage of the greater amount of information that, on most real-time systems, is available on the processes to be executed.

General-Purpose vs. Real-Time Scheduling

- When choosing a scheduling algorithm, it is often necessary to look for a compromise between optimizing the mean system performance and its determinism.
- Since they are less concerned with determinism, most general-purpose scheduling algorithms emphasize fairness and efficiency, and optimize the system throughput.
- On the other hand, real-time scheduling algorithms must put the emphasis on timings, even if this entails a greater overhead and the mean performance of the system becomes worse.
- In order to do this, the scheduling algorithms can take advantage of the greater amount of information that, on most real-time systems, is available on the processes to be executed.

A Simple Process Model

Analyzing an arbitrarily complex concurrent program to predict its worst-case timing behavior is very difficult, hence it is necessary to impose some restrictions on the structure of real-time concurrent programs. The most basic model has the following characteristics:

1. The application consists of a fixed number of processes.
2. All processes are periodic, with known periods.
3. The processes are completely independent of each other.
4. All processes have hard deadlines, and the deadline of each process is equal to its period.
5. All processes have known and fixed worst-case execution times.
6. All system's overheads, for example context switch times, are negligible.

A Simple Process Model

Analyzing an arbitrarily complex concurrent program to predict its worst-case timing behavior is very difficult, hence it is necessary to impose some restrictions on the structure of real-time concurrent programs. The most basic model has the following characteristics:

1. The application consists of a fixed number of processes.
2. All processes are periodic, with known periods.
3. The processes are completely independent of each other.
4. All processes have hard deadlines, and the deadline of each process is equal to its period.
5. All processes have known and fixed worst-case execution times.
6. All system's overheads, for example context switch times, are negligible.

A Simple Process Model

Analyzing an arbitrarily complex concurrent program to predict its worst-case timing behavior is very difficult, hence it is necessary to impose some restrictions on the structure of real-time concurrent programs. The most basic model has the following characteristics:

1. The application consists of a fixed number of processes.
2. All processes are periodic, with known periods.
3. The processes are completely independent of each other.
4. All processes have hard deadlines, and the deadline of each process is equal to its period.
5. All processes have known and fixed worst-case execution times.
6. All system's overheads, for example context switch times, are negligible.

A Simple Process Model

Analyzing an arbitrarily complex concurrent program to predict its worst-case timing behavior is very difficult, hence it is necessary to impose some restrictions on the structure of real-time concurrent programs. The most basic model has the following characteristics:

1. The application consists of a fixed number of processes.
2. All processes are periodic, with known periods.
3. The processes are completely independent of each other.
4. All processes have hard deadlines, and the deadline of each process is equal to its period.
5. All processes have known and fixed worst-case execution times.
6. All system's overheads, for example context switch times, are negligible.

A Simple Process Model

Analyzing an arbitrarily complex concurrent program to predict its worst-case timing behavior is very difficult, hence it is necessary to impose some restrictions on the structure of real-time concurrent programs. The most basic model has the following characteristics:

1. The application consists of a fixed number of processes.
2. All processes are periodic, with known periods.
3. The processes are completely independent of each other.
4. All processes have hard deadlines, and the deadline of each process is equal to its period.
5. All processes have known and fixed worst-case execution times.
6. All system's overheads, for example context switch times, are negligible.

A Simple Process Model

Analyzing an arbitrarily complex concurrent program to predict its worst-case timing behavior is very difficult, hence it is necessary to impose some restrictions on the structure of real-time concurrent programs. The most basic model has the following characteristics:

1. The application consists of a fixed number of processes.
2. All processes are periodic, with known periods.
3. The processes are completely independent of each other.
4. All processes have hard deadlines, and the deadline of each process is equal to its period.
5. All processes have known and fixed worst-case execution times.
6. All system's overheads, for example context switch times, are negligible.

A Simple Process Model

Analyzing an arbitrarily complex concurrent program to predict its worst-case timing behavior is very difficult, hence it is necessary to impose some restrictions on the structure of real-time concurrent programs. The most basic model has the following characteristics:

1. The application consists of a fixed number of processes.
2. All processes are periodic, with known periods.
3. The processes are completely independent of each other.
4. All processes have hard deadlines, and the deadline of each process is equal to its period.
5. All processes have known and fixed worst-case execution times.
6. All system's overheads, for example context switch times, are negligible.

Model Generalizations

The basic model just introduced has a number of shortcomings, and will be generalized to make it more suitable to describe real-world systems.

In particular:

- The hypothesis of independence among processes is somewhat contrary to the motivations of concurrent systems, in which processes must interact with one another.
- The deadline of a process is not always related to its period, and is often shorter than it.
- Some processes are sporadic, that is, they are executed when an external event, for example an alarm, occurs.

Model Generalizations

The basic model just introduced has a number of shortcomings, and will be generalized to make it more suitable to describe real-world systems.

In particular:

- The hypothesis of independence among processes is somewhat contrary to the motivations of concurrent systems, in which processes must interact with one another.
- The deadline of a process is not always related to its period, and is often shorter than it.
- Some processes are sporadic, that is, they are executed when an external event, for example an alarm, occurs.

Model Generalizations

The basic model just introduced has a number of shortcomings, and will be generalized to make it more suitable to describe real-world systems.

In particular:

- The hypothesis of independence among processes is somewhat contrary to the motivations of concurrent systems, in which processes must interact with one another.
- The deadline of a process is not always related to its period, and is often shorter than it.
- Some processes are sporadic, that is, they are executed when an external event, for example an alarm, occurs.

Model Generalizations

The basic model just introduced has a number of shortcomings, and will be generalized to make it more suitable to describe real-world systems.

In particular:

- The hypothesis of independence among processes is somewhat contrary to the motivations of concurrent systems, in which processes must interact with one another.
- The deadline of a process is not always related to its period, and is often shorter than it.
- Some processes are sporadic, that is, they are executed when an external event, for example an alarm, occurs.

Further Generalizations

- For some applications and hardware architectures, scheduling and context switch times are far from being negligible.
- The behavior of some non-deterministic hardware components, for example caches, must sometimes be taken into account.
- Real-time systems may sometimes be overloaded, a critical situation in which the computational demand exceeds the system capacity during a certain time interval, so that not all processes can meet their deadline.

Further Generalizations

- For some applications and hardware architectures, scheduling and context switch times are far from being negligible.
- The behavior of some non-deterministic hardware components, for example caches, must sometimes be taken into account.
- Real-time systems may sometimes be overloaded, a critical situation in which the computational demand exceeds the system capacity during a certain time interval, so that not all processes can meet their deadline.

Further Generalizations

- For some applications and hardware architectures, scheduling and context switch times are far from being negligible.
- The behavior of some non-deterministic hardware components, for example caches, must sometimes be taken into account.
- Real-time systems may sometimes be overloaded, a critical situation in which the computational demand exceeds the system capacity during a certain time interval, so that not all processes can meet their deadline.

Basic Notation (I)

- A periodic real-time process is called a task and denoted by τ_i.
- Its j-th instance is sometimes called a job; accordingly, $\tau_{i,j}$ denotes the j-th instance of the i-th task.
- T_i is the period of task τ_i.
- D_i is the relative deadline of task τ_i.
- $d_{i,j}$ is the absolute deadline of the j-th instance of task τ_i.
- C_i is the worst-case execution time of task τ_i.
- R_i is the worst-case response time of task τ_i.
- $f_{i,j}$ is the response time of the j-th instance of task τ_i.

Basic Notation (I)

- A periodic real-time process is called a task and denoted by τ_i.
- Its j-th instance is sometimes called a job; accordingly, $\tau_{i,j}$ denotes the j-th instance of the i-th task.
- T_i is the period of task τ_i.
- D_i is the relative deadline of task τ_i.
- $d_{i,j}$ is the absolute deadline of the j-th instance of task τ_i.
- C_i is the worst-case execution time of task τ_i.
- R_i is the worst-case response time of task τ_i.
- $f_{i,j}$ is the response time of the j-th instance of task τ_i.

Basic Notation (I)

- A periodic real-time process is called a task and denoted by τ_i.
- Its j-th instance is sometimes called a job; accordingly, $\tau_{i,j}$ denotes the j-th instance of the i-th task.
- T_i is the period of task τ_i.
- D_i is the relative deadline of task τ_i.
- $d_{i,j}$ is the absolute deadline of the j-th instance of task τ_i.
- C_i is the worst-case execution time of task τ_i.
- R_i is the worst-case response time of task τ_i.
- $f_{i,j}$ is the response time of the j-th instance of task τ_i.

Basic Notation (I)

- A periodic real-time process is called a task and denoted by τ_i.
- Its j-th instance is sometimes called a job; accordingly, $\tau_{i,j}$ denotes the j-th instance of the i-th task.
- T_i is the period of task τ_i.
- D_i is the relative deadline of task τ_i.
- $d_{i,j}$ is the absolute deadline of the j-th instance of task τ_i.
- C_i is the worst-case execution time of task τ_i.
- R_i is the worst-case response time of task τ_i.
- $f_{i,j}$ is the response time of the j-th instance of task τ_i.

Basic Notation (I)

- A periodic real-time process is called a task and denoted by τ_i.
- Its j-th instance is sometimes called a job; accordingly, $\tau_{i,j}$ denotes the j-th instance of the i-th task.
- T_i is the period of task τ_i.
- D_i is the relative deadline of task τ_i.
- $d_{i,j}$ is the absolute deadline of the j-th instance of task τ_i.
- C_i is the worst-case execution time of task τ_i.
- R_i is the worst-case response time of task τ_i.
- $f_{i,j}$ is the response time of the j-th instance of task τ_i.

Basic Notation (I)

- A periodic real-time process is called a task and denoted by τ_i.
- Its j-th instance is sometimes called a job; accordingly, $\tau_{i,j}$ denotes the j-th instance of the i-th task.
- T_i is the period of task τ_i.
- D_i is the relative deadline of task τ_i.
- $d_{i,j}$ is the absolute deadline of the j-th instance of task τ_i.
- C_i is the worst-case execution time of task τ_i.
- R_i is the worst-case response time of task τ_i.
- $f_{i,j}$ is the response time of the j-th instance of task τ_i.

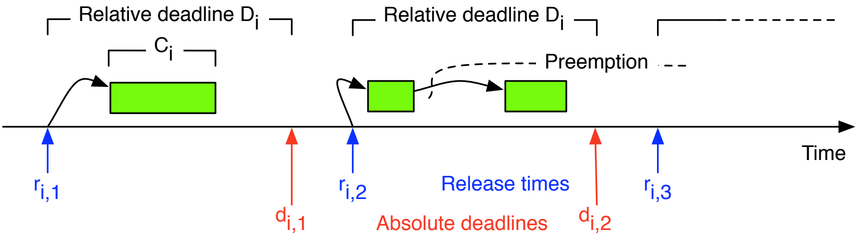
Basic Notation (I)

- A periodic real-time process is called a task and denoted by τ_i.
- Its j-th instance is sometimes called a job; accordingly, $\tau_{i,j}$ denotes the j-th instance of the i-th task.
- T_i is the period of task τ_i.
- D_i is the relative deadline of task τ_i.
- $d_{i,j}$ is the absolute deadline of the j-th instance of task τ_i.
- C_i is the worst-case execution time of task τ_i.
- R_i is the worst-case response time of task τ_i.
- $f_{i,j}$ is the response time of the j-th instance of task τ_i.

Basic Notation (I)

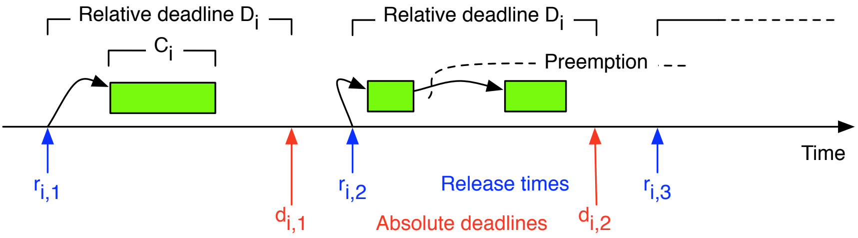
- A periodic real-time process is called a task and denoted by τ_i.
- Its j-th instance is sometimes called a job; accordingly, $\tau_{i,j}$ denotes the j-th instance of the i-th task.
- T_i is the period of task τ_i.
- D_i is the relative deadline of task τ_i.
- $d_{i,j}$ is the absolute deadline of the j-th instance of task τ_i.
- C_i is the worst-case execution time of task τ_i.
- R_i is the worst-case response time of task τ_i.
- $f_{i,j}$ is the response time of the j-th instance of task τ_i.

Basic Notation (II)



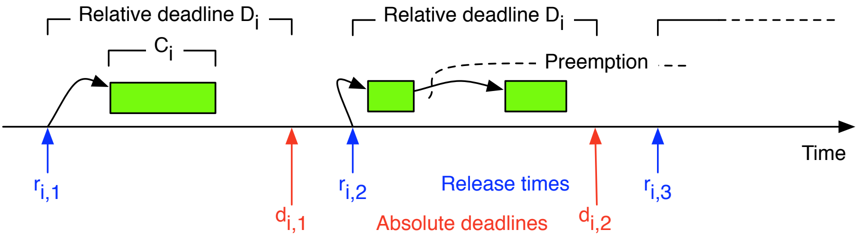
- $r_{i,j}$ denotes the release time of the j-th instance of task τ_i.
- The worst-case execution time (WCET) C_i is the time required to complete the task without any interference from other activities.
- The actual completion time — also known as response time — may be longer due, for example, to preemption.
- While relative deadlines (with respect to the task release time) are known and fixed for any given task, absolute deadlines depend on the task instance.

Basic Notation (II)



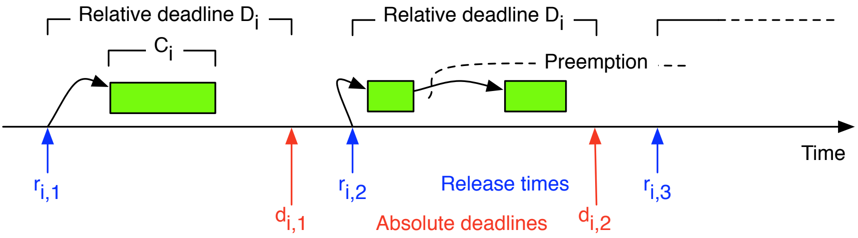
- $r_{i,j}$ denotes the release time of the j-th instance of task τ_i.
- The worst-case execution time (WCET) C_i is the time required to complete the task without any interference from other activities.
- The actual completion time — also known as response time — may be longer due, for example, to preemption.
- While relative deadlines (with respect to the task release time) are known and fixed for any given task, absolute deadlines depend on the task instance.

Basic Notation (II)



- $r_{i,j}$ denotes the release time of the j-th instance of task τ_i.
- The worst-case execution time (WCET) C_i is the time required to complete the task without any interference from other activities.
- The actual completion time — also known as response time — may be longer due, for example, to preemption.
- While relative deadlines (with respect to the task release time) are known and fixed for any given task, absolute deadlines depend on the task instance.

Basic Notation (II)



- $r_{i,j}$ denotes the release time of the j-th instance of task τ_i.
- The worst-case execution time (WCET) C_i is the time required to complete the task without any interference from other activities.
- The actual completion time — also known as response time — may be longer due, for example, to preemption.
- While relative deadlines (with respect to the task release time) are known and fixed for any given task, absolute deadlines depend on the task instance.

Basics of the Cyclic Executive

The cyclic executive, also known as timeline scheduling or cyclic scheduling, is one of the most widely used real-time scheduling methods.

- It is assumed that the basic model just introduced holds, hence there is a fixed set of periodic tasks.
- It is possible to lay out offline a complete schedule such that its repeated execution will cause all tasks to run at their correct rate and to meet their deadline.
- The cyclic executive is, essentially, a table of procedure calls where each call represents (part of) the code for a task.

Basics of the Cyclic Executive

The cyclic executive, also known as timeline scheduling or cyclic scheduling, is one of the most widely used real-time scheduling methods.

- It is assumed that the basic model just introduced holds, hence there is a fixed set of periodic tasks.
- It is possible to lay out offline a complete schedule such that its repeated execution will cause all tasks to run at their correct rate and to meet their deadline.
- The cyclic executive is, essentially, a table of procedure calls where each call represents (part of) the code for a task.

Basics of the Cyclic Executive

The cyclic executive, also known as timeline scheduling or cyclic scheduling, is one of the most widely used real-time scheduling methods.

- It is assumed that the basic model just introduced holds, hence there is a fixed set of periodic tasks.
- It is possible to lay out offline a complete schedule such that its repeated execution will cause all tasks to run at their correct rate and to meet their deadline.
- The cyclic executive is, essentially, a table of procedure calls where each call represents (part of) the code for a task.

Basics of the Cyclic Executive

The cyclic executive, also known as timeline scheduling or cyclic scheduling, is one of the most widely used real-time scheduling methods.

- It is assumed that the basic model just introduced holds, hence there is a fixed set of periodic tasks.
- It is possible to lay out offline a complete schedule such that its repeated execution will cause all tasks to run at their correct rate and to meet their deadline.
- The cyclic executive is, essentially, a table of procedure calls where each call represents (part of) the code for a task.

Major and Minor Cycles

- The complete table, known as the major cycle, is typically split into a number of slices called minor cycles, with a fixed duration.
- During execution, the cyclic executive switches from one minor cycle to another with the help of a periodic clock interrupt.
- As a consequence, only the activation of the tasks at the beginning of each minor cycle is synchronized with the real elapsed time.
- On the other hand, all the tasks belonging to the same minor cycle are simply activated in sequence.

Major and Minor Cycles

- The complete table, known as the major cycle, is typically split into a number of slices called minor cycles, with a fixed duration.
- During execution, the cyclic executive switches from one minor cycle to another with the help of a periodic clock interrupt.
- As a consequence, only the activation of the tasks at the beginning of each minor cycle is synchronized with the real elapsed time.
- On the other hand, all the tasks belonging to the same minor cycle are simply activated in sequence.

Major and Minor Cycles

- The complete table, known as the major cycle, is typically split into a number of slices called minor cycles, with a fixed duration.
- During execution, the cyclic executive switches from one minor cycle to another with the help of a periodic clock interrupt.
- As a consequence, only the activation of the tasks at the beginning of each minor cycle is synchronized with the real elapsed time.
- On the other hand, all the tasks belonging to the same minor cycle are simply activated in sequence.

Major and Minor Cycles

- The complete table, known as the major cycle, is typically split into a number of slices called minor cycles, with a fixed duration.
- During execution, the cyclic executive switches from one minor cycle to another with the help of a periodic clock interrupt.
- As a consequence, only the activation of the tasks at the beginning of each minor cycle is synchronized with the real elapsed time.
- On the other hand, all the tasks belonging to the same minor cycle are simply activated in sequence.

A Simple Example

If we have the following set of tasks...

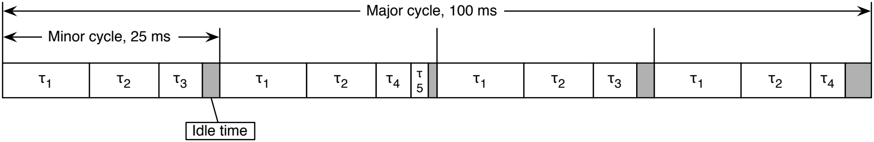
Task τ_i	Period T_i	Computation time C_i
τ_1	25	10
τ_2	25	8
τ_3	50	5
τ_4	50	4
τ_5	100	2

A Simple Example

If we have the following set of tasks...

Task τ_i	Period T_i	Computation time C_i
τ_1	25	10
τ_2	25	8
τ_3	50	5
τ_4	50	4
τ_5	100	2

...it can be scheduled on a single processor system, with a major cycle of 100 ms and a minor cycle of 25 ms, as follows:



The Scheduling Diagram

The diagram just shown, similar to a Gantt chart, is also known as the scheduling diagram.

- Its usage is (of course) not limited to the cyclic executive.
- It illustrates the job that each processor in the system is executing at any particular time.
- It is useful to visualize and understand how a scheduling algorithm works in a particular case.

The Scheduling Diagram

The diagram just shown, similar to a Gantt chart, is also known as the scheduling diagram.

- Its usage is (of course) not limited to the cyclic executive.
- It illustrates the job that each processor in the system is executing at any particular time.
- It is useful to visualize and understand how a scheduling algorithm works in a particular case.

The Scheduling Diagram

The diagram just shown, similar to a Gantt chart, is also known as the scheduling diagram.

- Its usage is (of course) not limited to the cyclic executive.
- It illustrates the job that each processor in the system is executing at any particular time.
- It is useful to visualize and understand how a scheduling algorithm works in a particular case.

The Scheduling Diagram

The diagram just shown, similar to a Gantt chart, is also known as the scheduling diagram.

- Its usage is (of course) not limited to the cyclic executive.
- It illustrates the job that each processor in the system is executing at any particular time.
- It is useful to visualize and understand how a scheduling algorithm works in a particular case.

Implementation of a Cyclic Executive

Provided an interrupt source with a 25 ms period is available, and the `intwait` function waits for an interrupt from this source, we can code the cyclic executive as:

```
while(1) {
  intwait();
    proc1(...); proc2(...); proc3(...);
  intwait();
    proc1(...); proc2(...); proc4(...); proc5(...);
  intwait();
    proc1(...); proc2(...); proc3(...);
  intwait();
    proc1(...); proc2(...); proc4(...);
}
```

where `proc1, ...proc5` are the procedures that contain the code of tasks $\tau_1, \ldots \tau_5$, respectively.

Implementation of a Cyclic Executive

Provided an interrupt source with a 25 ms period is available, and the `intwait` function waits for an interrupt from this source, we can code the cyclic executive as:

```
while(1) {
  intwait();
    proc1(...); proc2(...); proc3(...);
  intwait();
    proc1(...); proc2(...); proc4(...); proc5(...);
  intwait();
    proc1(...); proc2(...); proc3(...);
  intwait();
    proc1(...); proc2(...); proc4(...);
}
```

where `proc1, ...proc5` are the procedures that contain the code of tasks $\tau_1, \ldots \tau_5$, respectively.

Implementation of a Cyclic Executive

Provided an interrupt source with a 25 ms period is available, and the `intwait` function waits for an interrupt from this source, we can code the cyclic executive as:

```
while(1) {
  intwait();
    proc1(...); proc2(...); proc3(...);
  intwait();
    proc1(...); proc2(...); proc4(...); proc5(...);
  intwait();
    proc1(...); proc2(...); proc3(...);
  intwait();
    proc1(...); proc2(...); proc4(...);
}
```

where `proc1`, ...`proc5` are the procedures that contain the code of tasks $\tau_1, \ldots \tau_5$, respectively.

Implementation of a Cyclic Executive

Provided an interrupt source with a 25 ms period is available, and the `intwait` function waits for an interrupt from this source, we can code the cyclic executive as:

```
while(1) {
  intwait();
    proc1(...); proc2(...); proc3(...);
  intwait();
    proc1(...); proc2(...); proc4(...); proc5(...);
  intwait();
    proc1(...); proc2(...); proc3(...);
  intwait();
    proc1(...); proc2(...); proc4(...);
}
```

where `proc1`, ...`proc5` are the procedures that contain the code of tasks $\tau_1, \ldots \tau_5$, respectively.

Choice of the Minor Cycle Length

- The minor cycle period is the smallest timing reference of the cyclic executive.
- All task periods must be an integer multiple of the minor cycle period.
- Otherwise, the schedule would be unable to guarantee that the tasks will be executed at their proper rate.

Optimum minor cycle length

It is easy to show that the minor cycle period must be equal to the Greatest Common Divisor (GCD) of the periods of the tasks to be scheduled.

Choice of the Minor Cycle Length

- The minor cycle period is the smallest timing reference of the cyclic executive.
- All task periods must be an integer multiple of the minor cycle period.
- Otherwise, the schedule would be unable to guarantee that the tasks will be executed at their proper rate.

Optimum minor cycle length

It is easy to show that the minor cycle period must be equal to the Greatest Common Divisor (GCD) of the periods of the tasks to be scheduled.

Choice of the Minor Cycle Length

- The minor cycle period is the smallest timing reference of the cyclic executive.
- All task periods must be an integer multiple of the minor cycle period.
- Otherwise, the schedule would be unable to guarantee that the tasks will be executed at their proper rate.

Optimum minor cycle length

It is easy to show that the minor cycle period must be equal to the Greatest Common Divisor (GCD) of the periods of the tasks to be scheduled.

Choice of the Minor Cycle Length

- The minor cycle period is the smallest timing reference of the cyclic executive.
- All task periods must be an integer multiple of the minor cycle period.
- Otherwise, the schedule would be unable to guarantee that the tasks will be executed at their proper rate.

Optimum minor cycle length

It is easy to show that the minor cycle period must be equal to the Greatest Common Divisor (GCD) of the periods of the tasks to be scheduled.

Choice of the Major Cycle Length

- By definition, the major cycle period is the maximum period that can be accommodated without secondary schedules.
- Secondary schedules are procedures that are called at each major cycle and, in turn, call a secondary procedure every *n* major cycles.
- Therefore, secondary schedules may be useful to accommodate tasks with long periods without enlarging the major cycle length too much.

Choice of the major cycle length

When secondary schedules are not used, the major cycle period must be equal to the Least Common Multiple (LCM) of the periods of the tasks to be scheduled.

Choice of the Major Cycle Length

- By definition, the major cycle period is the maximum period that can be accommodated without secondary schedules.
- Secondary schedules are procedures that are called at each major cycle and, in turn, call a secondary procedure every *n* major cycles.
- Therefore, secondary schedules may be useful to accommodate tasks with long periods without enlarging the major cycle length too much.

Choice of the major cycle length

When secondary schedules are not used, the major cycle period must be equal to the Least Common Multiple (LCM) of the periods of the tasks to be scheduled.

Choice of the Major Cycle Length

- By definition, the major cycle period is the maximum period that can be accommodated without secondary schedules.
- Secondary schedules are procedures that are called at each major cycle and, in turn, call a secondary procedure every *n* major cycles.
- Therefore, secondary schedules may be useful to accommodate tasks with long periods without enlarging the major cycle length too much.

Choice of the major cycle length

When secondary schedules are not used, the major cycle period must be equal to the Least Common Multiple (LCM) of the periods of the tasks to be scheduled.

Choice of the Major Cycle Length

- By definition, the major cycle period is the maximum period that can be accommodated without secondary schedules.
- Secondary schedules are procedures that are called at each major cycle and, in turn, call a secondary procedure every *n* major cycles.
- Therefore, secondary schedules may be useful to accommodate tasks with long periods without enlarging the major cycle length too much.

Choice of the major cycle length

When secondary schedules are not used, the major cycle period must be equal to the Least Common Multiple (LCM) of the periods of the tasks to be scheduled.

Properties of the Cyclic Executive

- No actual processes exist at run-time, because the minor cycles are just a sequence of procedure calls.
- These procedures share a common address space, hence they implicitly share data. On a single processor system, the shared data does not need to be protected, because concurrent access is not possible.
- Once a suitable cyclic executive has been constructed, its implementation is straightforward and efficient, because no scheduling activity takes place at run-time and overheads are very low.
- The sequence of tasks in the schedule is always the same, and can be easily visualized.
- The existence of a cyclic executive that meets the timing requirements of a set of tasks is a "proof by construction" of its schedulability.

Properties of the Cyclic Executive

- No actual processes exist at run-time, because the minor cycles are just a sequence of procedure calls.
- These procedures share a common address space, hence they implicitly share data. On a single processor system, the shared data does not need to be protected, because concurrent access is not possible.
- Once a suitable cyclic executive has been constructed, its implementation is straightforward and efficient, because no scheduling activity takes place at run-time and overheads are very low.
- The sequence of tasks in the schedule is always the same, and can be easily visualized.
- The existence of a cyclic executive that meets the timing requirements of a set of tasks is a “proof by construction” of its schedulability.

Properties of the Cyclic Executive

- No actual processes exist at run-time, because the minor cycles are just a sequence of procedure calls.
- These procedures share a common address space, hence they implicitly share data. On a single processor system, the shared data does not need to be protected, because concurrent access is not possible.
- Once a suitable cyclic executive has been constructed, its implementation is straightforward and efficient, because no scheduling activity takes place at run-time and overheads are very low.
- The sequence of tasks in the schedule is always the same, and can be easily visualized.
- The existence of a cyclic executive that meets the timing requirements of a set of tasks is a “proof by construction” of its schedulability.

Properties of the Cyclic Executive

- No actual processes exist at run-time, because the minor cycles are just a sequence of procedure calls.
- These procedures share a common address space, hence they implicitly share data. On a single processor system, the shared data does not need to be protected, because concurrent access is not possible.
- Once a suitable cyclic executive has been constructed, its implementation is straightforward and efficient, because no scheduling activity takes place at run-time and overheads are very low.
- The sequence of tasks in the schedule is always the same, and can be easily visualized.
- The existence of a cyclic executive that meets the timing requirements of a set of tasks is a “proof by construction” of its schedulability.

Properties of the Cyclic Executive

- No actual processes exist at run-time, because the minor cycles are just a sequence of procedure calls.
- These procedures share a common address space, hence they implicitly share data. On a single processor system, the shared data does not need to be protected, because concurrent access is not possible.
- Once a suitable cyclic executive has been constructed, its implementation is straightforward and efficient, because no scheduling activity takes place at run-time and overheads are very low.
- The sequence of tasks in the schedule is always the same, and can be easily visualized.
- The existence of a cyclic executive that meets the timing requirements of a set of tasks is a “proof by construction” of its schedulability.

Shortcomings of the Cyclic Executive

- The cyclic executive "processes" cannot be protected from each other, as regular processes are.
- It is difficult to incorporate sporadic processes efficiently without changing the task sequence.
- When the periods of the tasks are mutually prime, the cyclic executive scheduling table will be large.
- Tasks with long periods are also problematic, because secondary schedules are tricky. For example, their worst-case execution time is far from the mean execution time.

Shortcomings of the Cyclic Executive

- The cyclic executive “processes” cannot be protected from each other, as regular processes are.
- It is difficult to incorporate sporadic processes efficiently without changing the task sequence.
- When the periods of the tasks are mutually prime, the cyclic executive scheduling table will be large.
- Tasks with long periods are also problematic, because secondary schedules are tricky. For example, their worst-case execution time is far from the mean execution time.

Shortcomings of the Cyclic Executive

- The cyclic executive “processes” cannot be protected from each other, as regular processes are.
- It is difficult to incorporate sporadic processes efficiently without changing the task sequence.
- When the periods of the tasks are mutually prime, the cyclic executive scheduling table will be large.
- Tasks with long periods are also problematic, because secondary schedules are tricky. For example, their worst-case execution time is far from the mean execution time.

Shortcomings of the Cyclic Executive

- The cyclic executive “processes” cannot be protected from each other, as regular processes are.
- It is difficult to incorporate sporadic processes efficiently without changing the task sequence.
- When the periods of the tasks are mutually prime, the cyclic executive scheduling table will be large.
- Tasks with long periods are also problematic, because secondary schedules are tricky. For example, their worst-case execution time is far from the mean execution time.

Tasks with Sizable Execution Times

Any task with an execution time greater than the minor cycle time (and possibly other tasks as well, to avoid “monopolizing” a minor cycle) will need to be split into a number of smaller procedures.

- The split may cut across the structure of the code in a way that has nothing to do with the structure of the code itself.
- The schedule is sensitive to any change in the task characteristics, above all their periods, which requires the entire scheduling sequence to be reconstructed from scratch.

In addition, building a cyclic executive is mathematically hard by itself.

Tasks with Sizable Execution Times

Any task with an execution time greater than the minor cycle time (and possibly other tasks as well, to avoid "monopolizing" a minor cycle) will need to be split into a number of smaller procedures.

- The split may cut across the structure of the code in a way that has nothing to do with the structure of the code itself.
- The schedule is sensitive to any change in the task characteristics, above all their periods, which requires the entire scheduling sequence to be reconstructed from scratch.

In addition, building a cyclic executive is mathematically hard by itself.

Tasks with Sizable Execution Times

Any task with an execution time greater than the minor cycle time (and possibly other tasks as well, to avoid “monopolizing” a minor cycle) will need to be split into a number of smaller procedures.

- The split may cut across the structure of the code in a way that has nothing to do with the structure of the code itself.
- The schedule is sensitive to any change in the task characteristics, above all their periods, which requires the entire scheduling sequence to be reconstructed from scratch.

In addition, building a cyclic executive is mathematically hard by itself.

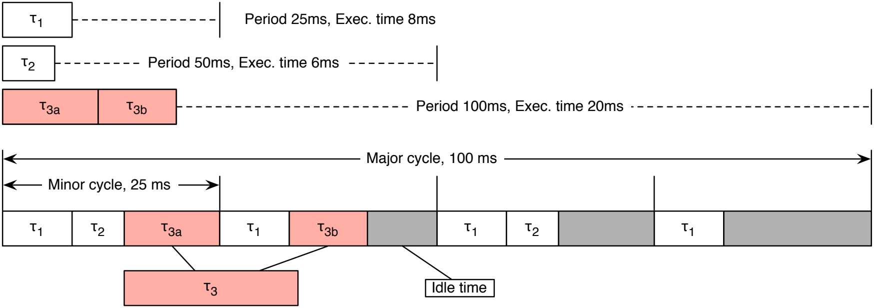
Tasks with Sizable Execution Times

Any task with an execution time greater than the minor cycle time (and possibly other tasks as well, to avoid “monopolizing” a minor cycle) will need to be split into a number of smaller procedures.

- The split may cut across the structure of the code in a way that has nothing to do with the structure of the code itself.
- The schedule is sensitive to any change in the task characteristics, above all their periods, which requires the entire scheduling sequence to be reconstructed from scratch.

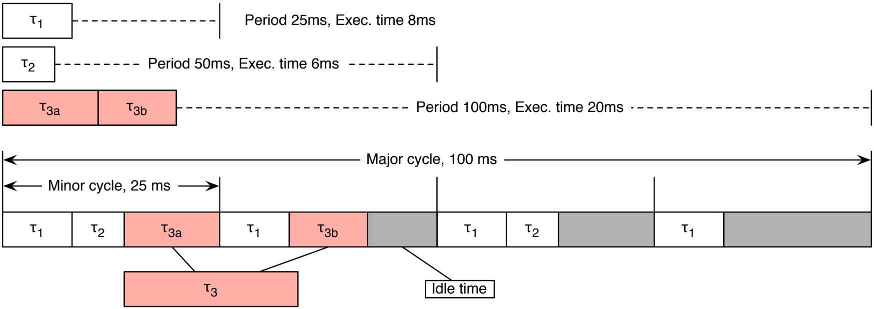
In addition, building a cyclic executive is mathematically hard by itself.

Tasks with Sizable Execution Times – An Example



- Task τ_3 could be executed entirely within a single minor cycle ($C_3 \leq 25$ ms), but it would interfere with the schedule of the other tasks, especially τ_1.
- Hence, for example, it must be split into two pieces τ_{3a} and τ_{3b}, so that $C_{3a} \leq 11$ ms and $C_{3b} \leq 17$ ms, respectively.

Tasks with Sizable Execution Times – An Example



- Task τ_3 could be executed entirely within a single minor cycle ($C_3 \leq 25$ ms), but it would interfere with the schedule of the other tasks, especially τ_1.
- Hence, for example, it must be split into two pieces τ_{3a} and τ_{3b}, so that $C_{3a} \leq 11$ ms and $C_{3b} \leq 17$ ms, respectively.

Constructing a Cyclic Executive is Hard

The construction of a cyclic executive can be put into analogy with the classical bin packing problem.

- With that problem, items of varying sizes (in just one dimension, like the task instances in this case) must be placed in the minimum number of bins (the minor cycles) so that no bin is overfull.
- The analogy does not take into account secondary schedules and the necessity of splitting “long” tasks into pieces.
- The bin-packing problem is known to be NP-hard, that is, its complexity is greater than a problem that can be solved in polynomial time by a nondeterministic Turing machine.
- For sizable problems, sub-optimal schemes, based on heuristics, must be used.

Constructing a Cyclic Executive is Hard

The construction of a cyclic executive can be put into analogy with the classical bin packing problem.

- With that problem, items of varying sizes (in just one dimension, like the task instances in this case) must be placed in the minimum number of bins (the minor cycles) so that no bin is overfull.
- The analogy does not take into account secondary schedules and the necessity of splitting “long” tasks into pieces.
- The bin-packing problem is known to be NP-hard, that is, its complexity is greater than a problem that can be solved in polynomial time by a nondeterministic Turing machine.
- For sizable problems, sub-optimal schemes, based on heuristics, must be used.

Constructing a Cyclic Executive is Hard

The construction of a cyclic executive can be put into analogy with the classical bin packing problem.

- With that problem, items of varying sizes (in just one dimension, like the task instances in this case) must be placed in the minimum number of bins (the minor cycles) so that no bin is overfull.
- The analogy does not take into account secondary schedules and the necessity of splitting "long" tasks into pieces.
- The bin-packing problem is known to be NP-hard, that is, its complexity is greater than a problem that can be solved in polynomial time by a nondeterministic Turing machine.
- For sizable problems, sub-optimal schemes, based on heuristics, must be used.

Constructing a Cyclic Executive is Hard

The construction of a cyclic executive can be put into analogy with the classical bin packing problem.

- With that problem, items of varying sizes (in just one dimension, like the task instances in this case) must be placed in the minimum number of bins (the minor cycles) so that no bin is overfull.
- The analogy does not take into account secondary schedules and the necessity of splitting “long” tasks into pieces.
- The bin-packing problem is known to be NP-hard, that is, its complexity is greater than a problem that can be solved in polynomial time by a nondeterministic Turing machine.
- For sizable problems, sub-optimal schemes, based on heuristics, must be used.

Constructing a Cyclic Executive is Hard

The construction of a cyclic executive can be put into analogy with the classical bin packing problem.

- With that problem, items of varying sizes (in just one dimension, like the task instances in this case) must be placed in the minimum number of bins (the minor cycles) so that no bin is overfull.
- The analogy does not take into account secondary schedules and the necessity of splitting “long” tasks into pieces.
- The bin-packing problem is known to be NP-hard, that is, its complexity is greater than a problem that can be solved in polynomial time by a nondeterministic Turing machine.
- For sizable problems, sub-optimal schemes, based on heuristics, must be used.

Process-Based Scheduling

- With the cyclic executive approach, the notion of process (or thread) is not preserved during execution.
- An alternative approach is to support process (or thread) execution directly, as is normally done in general-purpose operating systems.
- In this case, the process to be executed at any one time is determined by using one or more scheduling attributes.

With this approach, the process (or thread) becomes the unit of scheduling.

Process-Based Scheduling

- With the cyclic executive approach, the notion of process (or thread) is not preserved during execution.
- An alternative approach is to support process (or thread) execution directly, as is normally done in general-purpose operating systems.
- In this case, the process to be executed at any one time is determined by using one or more scheduling attributes.

With this approach, the process (or thread) becomes the unit of scheduling.

Process-Based Scheduling

- With the cyclic executive approach, the notion of process (or thread) is not preserved during execution.
- An alternative approach is to support process (or thread) execution directly, as is normally done in general-purpose operating systems.
- In this case, the process to be executed at any one time is determined by using one or more scheduling attributes.

With this approach, the process (or thread) becomes the unit of scheduling.

Process-Based Scheduling

- With the cyclic executive approach, the notion of process (or thread) is not preserved during execution.
- An alternative approach is to support process (or thread) execution directly, as is normally done in general-purpose operating systems.
- In this case, the process to be executed at any one time is determined by using one or more scheduling attributes.

With this approach, the process (or thread) becomes the unit of scheduling.

Scheduling Approaches

Assuming that process scheduling is being used, there are still many different scheduling approaches. Among them, we will consider:

1. Fixed-Priority Scheduling (FPS): each process has a fixed, static priority which is computed offline, before run-time. The *ready* processes are then scheduled according to their priority.
2. Earliest Deadline First (EDF): the *ready* processes are executed in the order determined by their absolute deadlines, the highest priority process being the one with the nearest deadline. Since absolute deadlines are computed at run-time, the priority assignment is dynamic.

Scheduling Approaches

Assuming that process scheduling is being used, there are still many different scheduling approaches. Among them, we will consider:

1. Fixed-Priority Scheduling (FPS): each process has a fixed, static priority which is computed offline, before run-time. The *ready* processes are then scheduled according to their priority.
2. Earliest Deadline First (EDF): the *ready* processes are executed in the order determined by their absolute deadlines, the highest priority process being the one with the nearest deadline. Since absolute deadlines are computed at run-time, the priority assignment is dynamic.

Scheduling Approaches

Assuming that process scheduling is being used, there are still many different scheduling approaches. Among them, we will consider:

1. Fixed-Priority Scheduling (FPS): each process has a fixed, static priority which is computed offline, before run-time. The *ready* processes are then scheduled according to their priority.
2. Earliest Deadline First (EDF): the *ready* processes are executed in the order determined by their absolute deadlines, the highest priority process being the one with the nearest deadline. Since absolute deadlines are computed at run-time, the priority assignment is dynamic.

Preemption or Non-Preemption?

Preemptive vs. non-preemptive scheduling

- In a preemptive scheduling scheme, if a high-priority process becomes ready during the execution of a lower priority one, there will be an immediate switch to the higher-priority process.
- Alternatively, if the scheme is non-preemptive, the lower-priority process will be allowed to complete.

+ Preemptive schemes make higher-priority processes more reactive.
+ They also prevent rogue processes from "monopolizing" the CPU.
– Preemption is responsible for most race conditions.
– For example, the cyclic executive avoids most race conditions by executing its tasks in a well-defined sequence and without preemption.

Preemption or Non-Preemption?

Preemptive vs. non-preemptive scheduling

- In a preemptive scheduling scheme, if a high-priority process becomes ready during the execution of a lower priority one, there will be an immediate switch to the higher-priority process.
- Alternatively, if the scheme is non-preemptive, the lower-priority process will be allowed to complete.

+ Preemptive schemes make higher-priority processes more reactive.
+ They also prevent rogue processes from "monopolizing" the CPU.

– Preemption is responsible for most race conditions.

– For example, the cyclic executive avoids most race conditions by executing its tasks in a well-defined sequence and without preemption.

Preemption or Non-Preemption?

Preemptive vs. non-preemptive scheduling

- In a preemptive scheduling scheme, if a high-priority process becomes ready during the execution of a lower priority one, there will be an immediate switch to the higher-priority process.
- Alternatively, if the scheme is non-preemptive, the lower-priority process will be allowed to complete.

+ Preemptive schemes make higher-priority processes more reactive.
+ They also prevent rogue processes from “monopolizing” the CPU.

– Preemption is responsible for most race conditions.

– For example, the cyclic executive avoids most race conditions by executing its tasks in a well-defined sequence and without preemption.

Preemption or Non-Preemption?

Preemptive vs. non-preemptive scheduling

- In a preemptive scheduling scheme, if a high-priority process becomes ready during the execution of a lower priority one, there will be an immediate switch to the higher-priority process.
- Alternatively, if the scheme is non-preemptive, the lower-priority process will be allowed to complete.

+ Preemptive schemes make higher-priority processes more reactive.
+ They also prevent rogue processes from “monopolizing” the CPU.
– Preemption is responsible for most race conditions.
– For example, the cyclic executive avoids most race conditions by executing its tasks in a well-defined sequence and without preemption.

Preemption or Non-Preemption?

Preemptive vs. non-preemptive scheduling

- In a preemptive scheduling scheme, if a high-priority process becomes ready during the execution of a lower priority one, there will be an immediate switch to the higher-priority process.
- Alternatively, if the scheme is non-preemptive, the lower-priority process will be allowed to complete.

+ Preemptive schemes make higher-priority processes more reactive.
+ They also prevent rogue processes from “monopolizing” the CPU.
– Preemption is responsible for most race conditions.
– For example, the cyclic executive avoids most race conditions by executing its tasks in a well-defined sequence and without preemption.

The Rate Monotonic Scheduler

- The basic process model just introduced is being used.
- Tasks have a static priority.
- They are scheduled preemptively and according to their priority.
- There is only one processor.

An optimum priority assignment scheme

Under these hypotheses, there exists a simple, optimum priority assignment scheme, due to Liu and Layland (1973) and known as Rate Monotonic (RM or RMS).

The Rate Monotonic Scheduler

- The basic process model just introduced is being used.
- Tasks have a static priority.
- They are scheduled preemptively and according to their priority.
- There is only one processor.

An optimum priority assignment scheme

Under these hypotheses, there exists a simple, optimum priority assignment scheme, due to Liu and Layland (1973) and known as Rate Monotonic (RM or RMS).

The Rate Monotonic Scheduler

- The basic process model just introduced is being used.
- Tasks have a static priority.
- They are scheduled preemptively and according to their priority.
- There is only one processor.

An optimum priority assignment scheme

Under these hypotheses, there exists a simple, optimum priority assignment scheme, due to Liu and Layland (1973) and known as Rate Monotonic (RM or RMS).

The Rate Monotonic Scheduler

- The basic process model just introduced is being used.
- Tasks have a static priority.
- They are scheduled preemptively and according to their priority.
- There is only one processor.

An optimum priority assignment scheme

Under these hypotheses, there exists a simple, optimum priority assignment scheme, due to Liu and Layland (1973) and known as Rate Monotonic (RM or RMS).

The Rate Monotonic Scheduler

- The basic process model just introduced is being used.
- Tasks have a static priority.
- They are scheduled preemptively and according to their priority.
- There is only one processor.

An optimum priority assignment scheme

Under these hypotheses, there exists a simple, optimum priority assignment scheme, due to Liu and Layland (1973) and known as Rate Monotonic (RM or RMS).

RM Priority Assignment

Each task is assigned a (fixed) priority that is inversely proportional to its period: the shorter the period, the higher the priority.

- This assignment is optimum in the sense that if any task set can be scheduled using a preemptive, fixed-priority scheduler...
- ...then the same task set can also be scheduled using the rate monotonic priority assignment.

RM Priority Assignment

Each task is assigned a (fixed) priority that is inversely proportional to its period: the shorter the period, the higher the priority.

- This assignment is optimum in the sense that if any task set can be scheduled using a preemptive, fixed-priority scheduler…
- …then the same task set can also be scheduled using the rate monotonic priority assignment.

RM Priority Assignment

Each task is assigned a (fixed) priority that is inversely proportional to its period: the shorter the period, the higher the priority.

- This assignment is optimum in the sense that if any task set can be scheduled using a preemptive, fixed-priority scheduler…
- …then the same task set can also be scheduled using the rate monotonic priority assignment.

RM Scheduling – An Example

If we have the following set of tasks:

Task τ_i	Period T_i	Computation time C_i	Priority
τ_1	20	7	
τ_2	50	13	
τ_3	25	6	

task τ_1 (which has the smallest period) will have the highest priority, followed by τ_3 and τ_2, in that order.

Caution!

Priorities are often expressed by integers, but there is no general agreement on whether higher values represent higher priorities or the other way round.

RM Scheduling – An Example

If we have the following set of tasks:

Task τ_i	Period T_i	Computation time C_i	Priority
τ_1	20	7	
τ_2	50	13	
τ_3	25	6	

task τ_1 (which has the smallest period) will have the highest priority, followed by τ_3 and τ_2, in that order.

Caution!

Priorities are often expressed by integers, but there is no general agreement on whether higher values represent higher priorities or the other way round.

RM Scheduling – An Example

If we have the following set of tasks:

Task τ_i	Period T_i	Computation time C_i	Priority
τ_1	20	7	High
τ_2	50	13	
τ_3	25	6	

task τ_1 (which has the smallest period) will have the highest priority, followed by τ_3 and τ_2, in that order.

Caution!

Priorities are often expressed by integers, but there is no general agreement on whether higher values represent higher priorities or the other way round.

RM Scheduling – An Example

If we have the following set of tasks:

Task τ_i	Period T_i	Computation time C_i	Priority
τ_1	20	7	High
τ_2	50	13	
τ_3	25	6	Medium

task τ_1 (which has the smallest period) will have the highest priority, followed by τ_3 and τ_2, in that order.

Caution!

Priorities are often expressed by integers, but there is no general agreement on whether higher values represent higher priorities or the other way round.

RM Scheduling – An Example

If we have the following set of tasks:

Task τ_i	Period T_i	Computation time C_i	Priority
τ_1	20	7	High
τ_2	50	13	Low
τ_3	25	6	Medium

task τ_1 (which has the smallest period) will have the highest priority, followed by τ_3 and τ_2, in that order.

Caution!

Priorities are often expressed by integers, but there is no general agreement on whether higher values represent higher priorities or the other way round.

RM Scheduling – An Example

If we have the following set of tasks:

Task τ_i	Period T_i	Computation time C_i	Priority
τ_1	20	7	High
τ_2	50	13	Low
τ_3	25	6	Medium

task τ_1 (which has the smallest period) will have the highest priority, followed by τ_3 and τ_2, in that order.

Caution!

Priorities are often expressed by integers, but there is no general agreement on whether higher values represent higher priorities or the other way round.

Simulating the Schedule (I)

In order to better understand how a scheduler (the rate monotonic scheduler in this case) works, and to draw some conclusions about its characteristics, we can simulate the behavior of the scheduler and build the corresponding scheduling diagram.

- To be meaningful, the simulation must be carried out for an amount of time that is “long enough” to cover all possible phase relations among the tasks.
- As for the cyclic executive, the right amount of time is the Least Common Multiple (LCM) of the task periods, unless we exploit some specific properties of the scheduling algorithm.
- Since we do not have any additional information about the tasks, we also assume that all tasks are simultaneously released at $t = 0$.

Simulating the Schedule (I)

In order to better understand how a scheduler (the rate monotonic scheduler in this case) works, and to draw some conclusions about its characteristics, we can simulate the behavior of the scheduler and build the corresponding scheduling diagram.

- To be meaningful, the simulation must be carried out for an amount of time that is "long enough" to cover all possible phase relations among the tasks.
- As for the cyclic executive, the right amount of time is the Least Common Multiple (LCM) of the task periods, unless we exploit some specific properties of the scheduling algorithm.
- Since we do not have any additional information about the tasks, we also assume that all tasks are simultaneously released at $t = 0$.

Simulating the Schedule (I)

In order to better understand how a scheduler (the rate monotonic scheduler in this case) works, and to draw some conclusions about its characteristics, we can simulate the behavior of the scheduler and build the corresponding scheduling diagram.

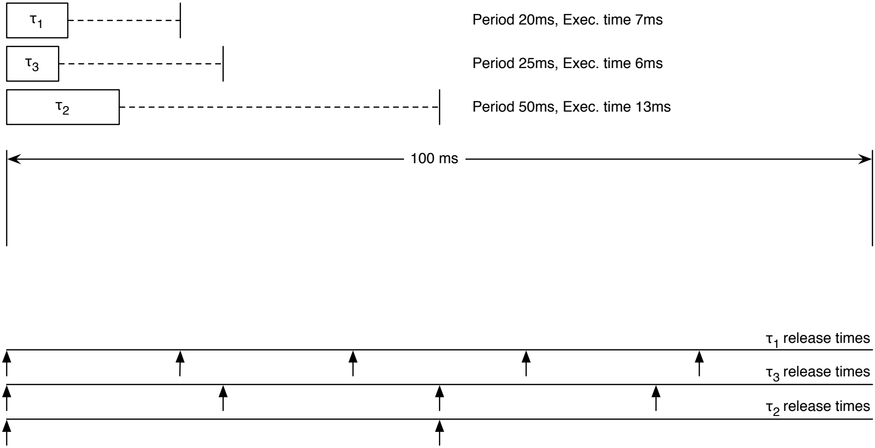
- To be meaningful, the simulation must be carried out for an amount of time that is “long enough” to cover all possible phase relations among the tasks.
- As for the cyclic executive, the right amount of time is the Least Common Multiple (LCM) of the task periods, unless we exploit some specific properties of the scheduling algorithm.
- Since we do not have any additional information about the tasks, we also assume that all tasks are simultaneously released at $t = 0$.

Simulating the Schedule (I)

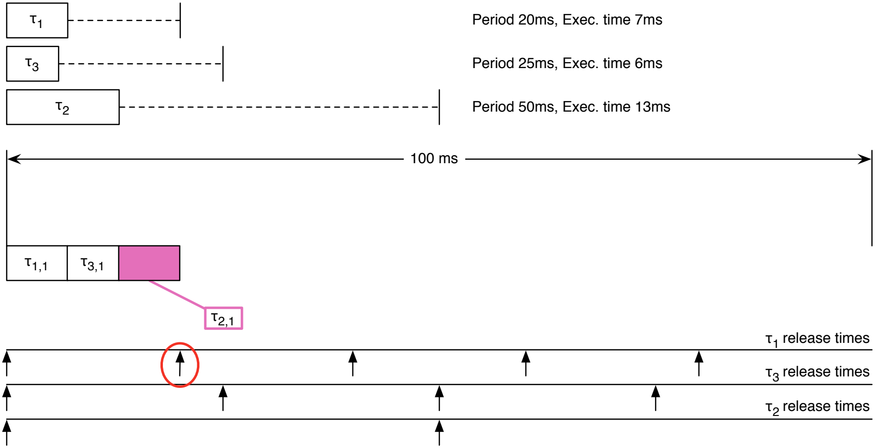
In order to better understand how a scheduler (the rate monotonic scheduler in this case) works, and to draw some conclusions about its characteristics, we can simulate the behavior of the scheduler and build the corresponding scheduling diagram.

- To be meaningful, the simulation must be carried out for an amount of time that is “long enough” to cover all possible phase relations among the tasks.
- As for the cyclic executive, the right amount of time is the Least Common Multiple (LCM) of the task periods, unless we exploit some specific properties of the scheduling algorithm.
- Since we do not have any additional information about the tasks, we also assume that all tasks are simultaneously released at $t = 0$.

Simulating the Schedule (II)

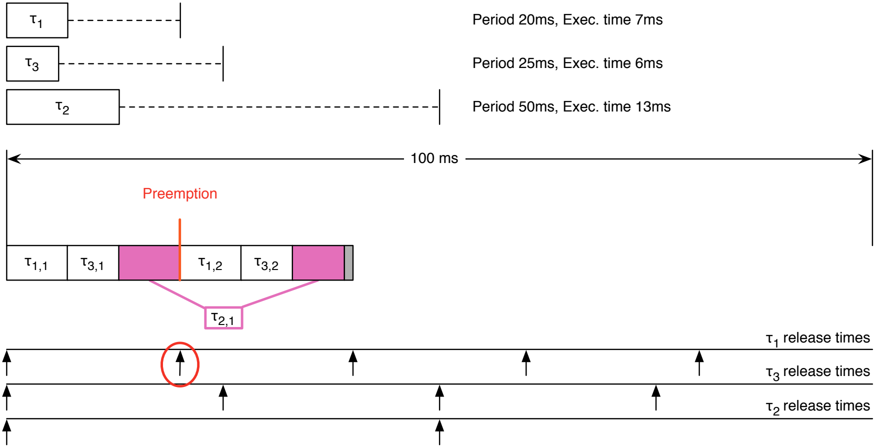


Simulating the Schedule (II)



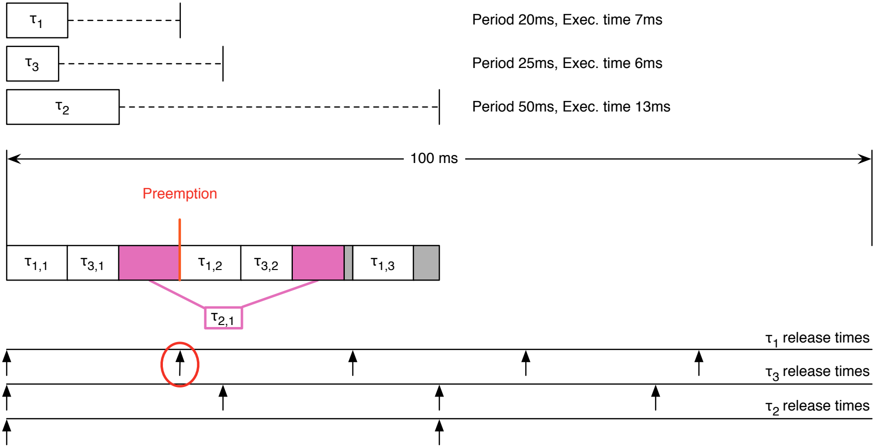
At $t = 0$, all tasks are ready: the first one to be executed is τ_1 then, at its completion, τ_3. At $t = 13$, τ_2 finally starts but, at $t = 20$, τ_1 is released again.

Simulating the Schedule (II)



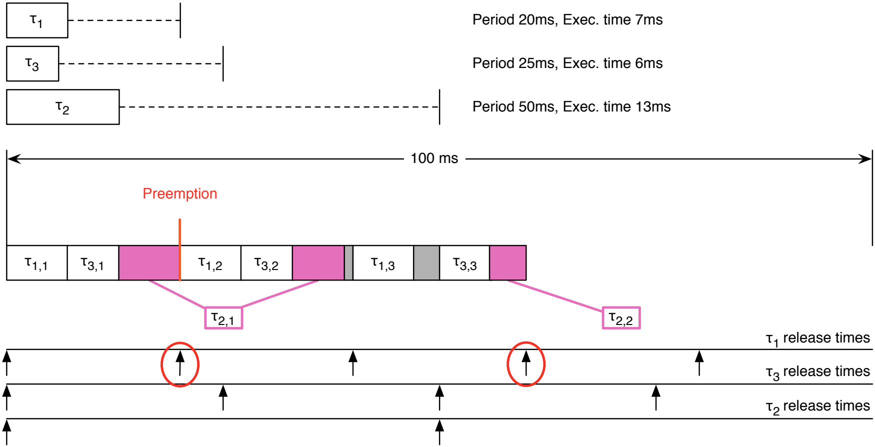
Hence, τ_2 is preempted in favor of τ_1. While τ_1 is executing, τ_3 is released, but this does not lead to a preemption: τ_3 is executed after τ_1 has finished. Finally, τ_2 is resumed and then completed at $t = 39$.

Simulating the Schedule (II)



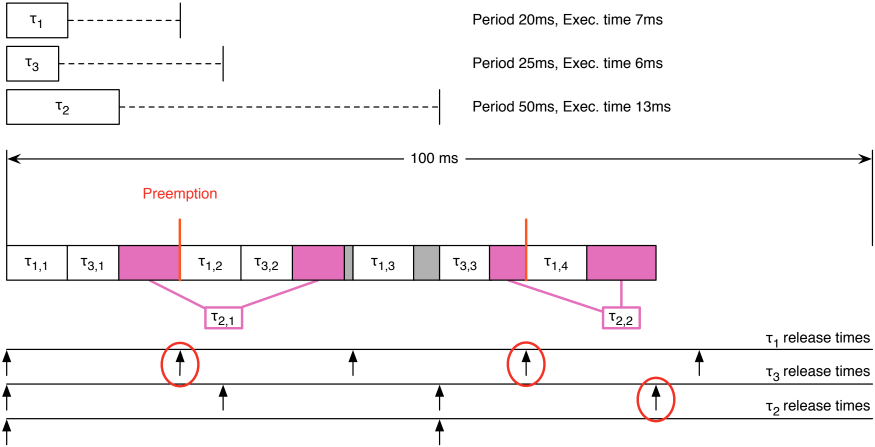
At $t = 40$, after 1 ms of idling, task τ_1 is released. Since it is the only *ready* task, it is executed immediately, and completes at $t = 47$.

Simulating the Schedule (II)



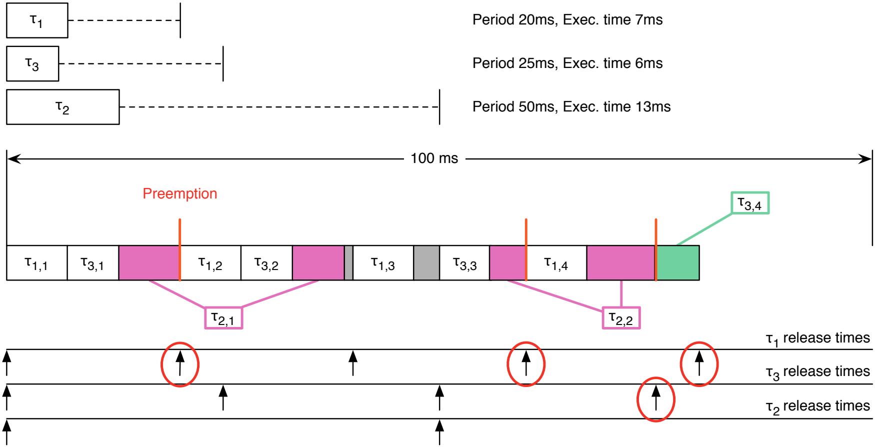
At $t = 50$, both τ_3 and τ_2 become *ready* simultaneously. τ_3 is run first, then τ_2 starts and runs for 4 ms. However, at $t = 60$, τ_1 is released again.

Simulating the Schedule (II)



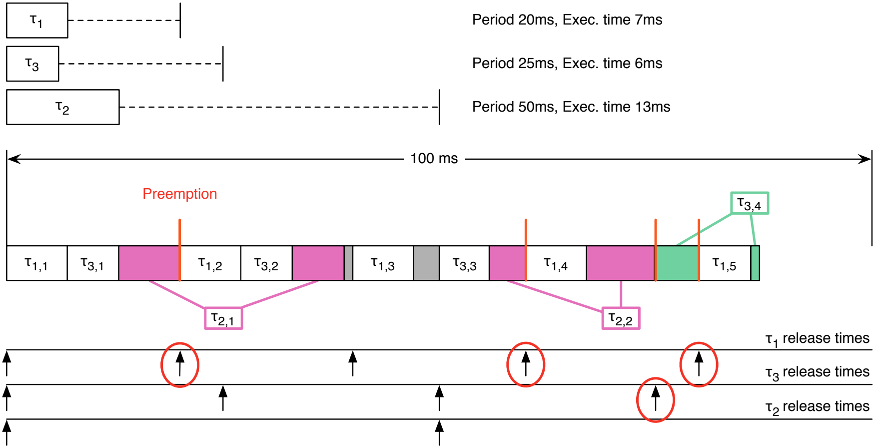
As before, this leads to the preemption of τ_2 and τ_1 runs to completion. Then, τ_2 is resumed and runs for 8 ms, until τ_3 is released.

Simulating the Schedule (II)



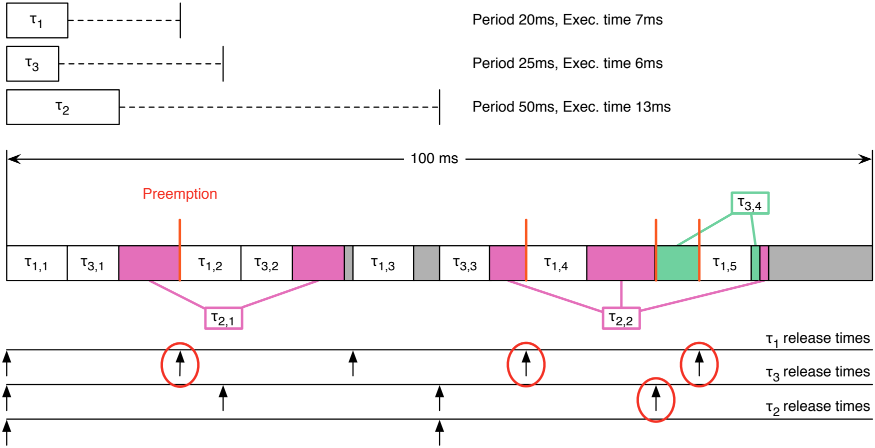
τ_2 is preempted again to run τ_3. The latter runs for 5 ms but at, $t = 80$, τ_1 is released for the fifth time.

Simulating the Schedule (II)



τ_3 is preempted, too, to run τ_1. After the completion of τ_1, both τ_3 and τ_2 are *ready*. τ_3 runs for 1 ms, then completes.

Simulating the Schedule (II)



Finally, τ_2 runs and completes its execution cycle by consuming 1 ms of CPU time. After that, the system stays idle until $t = 100$, where the whole cycle starts again.

Optimality of Rate Monotonic – Definitions

- According to the simple process model, the relative deadline of a task is equal to its period, that is, $D_i = T_i \;\forall i$.
- Hence, for each task instance, the absolute deadline is the time of its next release, that is, $d_{i,j} = r_{i,j+1}$.
- We say that there is an overflow at time t if t is the deadline of a job that misses the deadline.
- A scheduling algorithm is feasible for a given set of task if they are scheduled so that no overflows ever occur.

Definition of critical instant and critical time zone

- A critical instant for a task is an instant at which the release of the task will produce the largest response time.
- A critical time zone for a task is the interval between a critical instant and the end of the task response.

Optimality of Rate Monotonic – Definitions

- According to the simple process model, the relative deadline of a task is equal to its period, that is, $D_i = T_i \ \forall i$.
- Hence, for each task instance, the absolute deadline is the time of its next release, that is, $d_{i,j} = r_{i,j+1}$.
- We say that there is an overflow at time t if t is the deadline of a job that misses the deadline.
- A scheduling algorithm is feasible for a given set of task if they are scheduled so that no overflows ever occur.

Definition of critical instant and critical time zone

- A critical instant for a task is an instant at which the release of the task will produce the largest response time.
- A critical time zone for a task is the interval between a critical instant and the end of the task response.

Optimality of Rate Monotonic – Definitions

- According to the simple process model, the relative deadline of a task is equal to its period, that is, $D_i = T_i \ \forall i$.
- Hence, for each task instance, the absolute deadline is the time of its next release, that is, $d_{i,j} = r_{i,j+1}$.
- We say that there is an overflow at time t if t is the deadline of a job that misses the deadline.
- A scheduling algorithm is feasible for a given set of task if they are scheduled so that no overflows ever occur.

Definition of critical instant and critical time zone

- A critical instant for a task is an instant at which the release of the task will produce the largest response time.
- A critical time zone for a task is the interval between a critical instant and the end of the task response.

Optimality of Rate Monotonic – Definitions

- According to the simple process model, the relative deadline of a task is equal to its period, that is, $D_i = T_i \;\forall i$.
- Hence, for each task instance, the absolute deadline is the time of its next release, that is, $d_{i,j} = r_{i,j+1}$.
- We say that there is an overflow at time t if t is the deadline of a job that misses the deadline.
- A scheduling algorithm is feasible for a given set of task if they are scheduled so that no overflows ever occur.

Definition of critical instant and critical time zone

- A critical instant for a task is an instant at which the release of the task will produce the largest response time.
- A critical time zone for a task is the interval between a critical instant and the end of the task response.

Optimality of Rate Monotonic – Definitions

- According to the simple process model, the relative deadline of a task is equal to its period, that is, $D_i = T_i \ \forall i$.
- Hence, for each task instance, the absolute deadline is the time of its next release, that is, $d_{i,j} = r_{i,j+1}$.
- We say that there is an overflow at time t if t is the deadline of a job that misses the deadline.
- A scheduling algorithm is feasible for a given set of task if they are scheduled so that no overflows ever occur.

Definition of critical instant and critical time zone

- A critical instant for a task is an instant at which the release of the task will produce the largest response time.
- A critical time zone for a task is the interval between a critical instant and the end of the task response.

Where Are the Critical Instants?

Theorem (Liu and Layland, 1973)

A critical instant for any task occurs whenever it is released simultaneously with the release of all higher priority tasks.

- Let $\tau_1, \ldots, \tau_m$ be a set of tasks, listed in order of decreasing priority, and consider the task with the lowest priority, τ_m.
- If τ_m is released at t_1, between t_1 and $t_1 + T_m$, the time of the next release of τ_m, other tasks with an higher priority will possibly be released and interfere with the execution of τ_m, because of preemption.
- Now, consider one of the interfering tasks, τ_i, with $i < m$ and suppose that, in the interval between t_1 and $t_1 + T_m$, it is released at $t_2, t_2 + T_i, \ldots, t_2 + kT_i$, with $t_2 \geq t_1$.

Where Are the Critical Instants?

Theorem (Liu and Layland, 1973)

A critical instant for any task occurs whenever it is released simultaneously with the release of all higher priority tasks.

- Let $\tau_1, \ldots, \tau_m$ be a set of tasks, listed in order of decreasing priority, and consider the task with the lowest priority, τ_m.
- If τ_m is released at t_1, between t_1 and $t_1 + T_m$, the time of the next release of τ_m, other tasks with an higher priority will possibly be released and interfere with the execution of τ_m, because of preemption.
- Now, consider one of the interfering tasks, τ_i, with $i < m$ and suppose that, in the interval between t_1 and $t_1 + T_m$, it is released at $t_2, t_2 + T_i, \ldots, t_2 + kT_i$, with $t_2 \geq t_1$.

Where Are the Critical Instants?

Theorem (Liu and Layland, 1973)

A critical instant for any task occurs whenever it is released simultaneously with the release of all higher priority tasks.

- Let $\tau_1, \ldots, \tau_m$ be a set of tasks, listed in order of decreasing priority, and consider the task with the lowest priority, τ_m.
- If τ_m is released at t_1, between t_1 and $t_1 + T_m$, the time of the next release of τ_m, other tasks with an higher priority will possibly be released and interfere with the execution of τ_m, because of preemption.
- Now, consider one of the interfering tasks, τ_i, with $i < m$ and suppose that, in the interval between t_1 and $t_1 + T_m$, it is released at $t_2, t_2 + T_i, \ldots, t_2 + kT_i$, with $t_2 \geq t_1$.

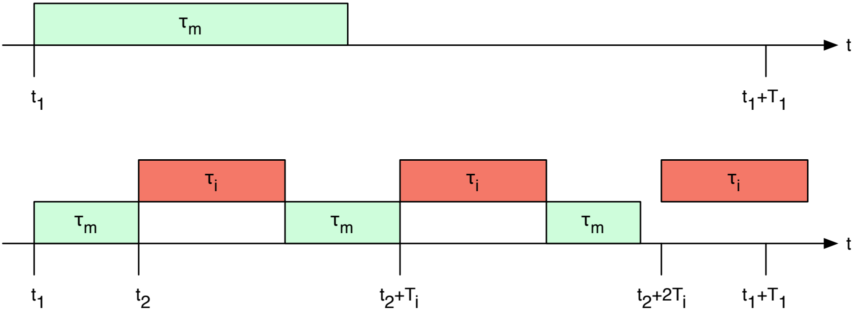
Where Are the Critical Instants?

Theorem (Liu and Layland, 1973)

A critical instant for any task occurs whenever it is released simultaneously with the release of all higher priority tasks.

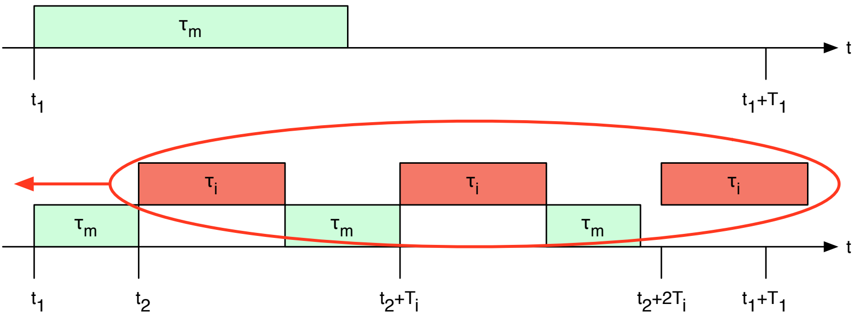
- Let $\tau_1, \ldots, \tau_m$ be a set of tasks, listed in order of decreasing priority, and consider the task with the lowest priority, τ_m.
- If τ_m is released at t_1, between t_1 and $t_1 + T_m$, the time of the next release of τ_m, other tasks with an higher priority will possibly be released and interfere with the execution of τ_m, because of preemption.
- Now, consider one of the interfering tasks, τ_i, with $i < m$ and suppose that, in the interval between t_1 and $t_1 + T_m$, it is released at $t_2, t_2 + T_i, \ldots, t_2 + kT_i$, with $t_2 \geq t_1$.

Moving t_2 around



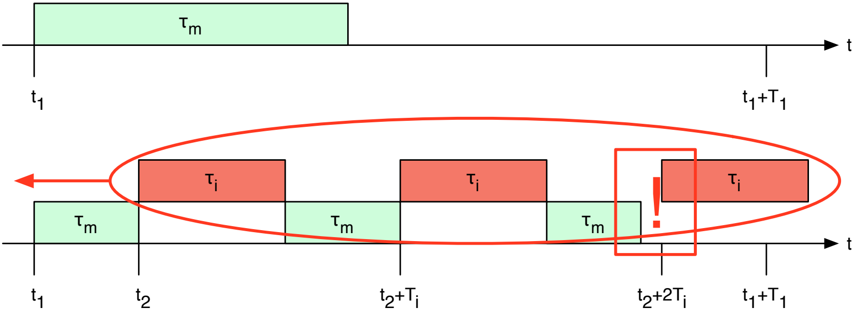
- The preemption of τ_m by τ_i will cause a certain amount of delay in the completion of the instance of τ_m being considered, unless it has already been completed before t_2.

Moving t_2 around



- The amount of delay depends on the relative placement of t_1 and t_2.
- However, moving t_2 towards t_1 will never decrease the completion time of τ_m.

Moving t_2 around



- Hence, the completion time of τ_m will be either unchanged or further delayed, due to additional interference, by moving t_2 towards t_1.
- The delay is largest when $t_1 = t_2$, that is, when the tasks are released simultaneously.

Final Remarks

- The argument just set out can be repeated for all tasks $\tau_i, \;\; 1 \leq i < m$, thus proving the theorem.

$\square$

Corollary

- Under the hypotheses of the theorem, it is possible to check whether or not a given priority assignment scheme will yield a feasible scheduling algorithm, without simulating it for the LCM of the periods.
- If all tasks fulfill their deadlines when they are released simultaneously, that is, at their critical instant, then the scheduling algorithm is feasible.
- This is a sufficient feasibility condition because, depending on the relative phases of the tasks, the critical instant may never occur in the actual schedule.

Final Remarks

- The argument just set out can be repeated for all tasks $\tau_i, \;\; 1 \leq i < m$, thus proving the theorem.

□

Corollary

- Under the hypotheses of the theorem, it is possible to check whether or not a given priority assignment scheme will yield a feasible scheduling algorithm, without simulating it for the LCM of the periods.
- If all tasks fulfill their deadlines when they are released simultaneously, that is, at their critical instant, then the scheduling algorithm is feasible.
- This is a sufficient feasibility condition because, depending on the relative phases of the tasks, the critical instant may never occur in the actual schedule.

Final Remarks

- The argument just set out can be repeated for all tasks $\tau_i, \ 1 \le i < m$, thus proving the theorem.

□

Corollary

- Under the hypotheses of the theorem, it is possible to check whether or not a given priority assignment scheme will yield a feasible scheduling algorithm, without simulating it for the LCM of the periods.
- If all tasks fulfill their deadlines when they are released simultaneously, that is, at their critical instant, then the scheduling algorithm is feasible.
- This is a sufficient feasibility condition because, depending on the relative phases of the tasks, the critical instant may never occur in the actual schedule.

Final Remarks

- The argument just set out can be repeated for all tasks $\tau_i, \; 1 \leq i < m$, thus proving the theorem.

$\square$

Corollary

- Under the hypotheses of the theorem, it is possible to check whether or not a given priority assignment scheme will yield a feasible scheduling algorithm, without simulating it for the LCM of the periods.
- If all tasks fulfill their deadlines when they are released simultaneously, that is, at their critical instant, then the scheduling algorithm is feasible.
- This is a sufficient feasibility condition because, depending on the relative phases of the tasks, the critical instant may never occur in the actual schedule.

Optimality of RM

- Starting from the previous theorem and its corollary, the optimality of RM is proved by showing that if a task set is schedulable by an arbitrary (but fixed) priority assignment, then it is also schedulable by RM.
- This result also implies that if RM cannot schedule a certain task set, no other fixed priority assignment algorithm can schedule it.
- We will start with a simpler lemma, that only involves two tasks.

Lemma

If the set of two tasks τ_1, τ_2 is schedulable by any arbitrary, but fixed, priority assignment, then it is schedulable by RM as well.

Optimality of RM

- Starting from the previous theorem and its corollary, the optimality of RM is proved by showing that if a task set is schedulable by an arbitrary (but fixed) priority assignment, then it is also schedulable by RM.
- This result also implies that if RM cannot schedule a certain task set, no other fixed priority assignment algorithm can schedule it.
- We will start with a simpler lemma, that only involves two tasks.

Lemma

If the set of two tasks τ_1, τ_2 is schedulable by any arbitrary, but fixed, priority assignment, then it is schedulable by RM as well.

Optimality of RM

- Starting from the previous theorem and its corollary, the optimality of RM is proved by showing that if a task set is schedulable by an arbitrary (but fixed) priority assignment, then it is also schedulable by RM.
- This result also implies that if RM cannot schedule a certain task set, no other fixed priority assignment algorithm can schedule it.
- We will start with a simpler lemma, that only involves two tasks.

Lemma

If the set of two tasks τ_1, τ_2 is schedulable by any arbitrary, but fixed, priority assignment, then it is schedulable by RM as well.

Optimality of RM

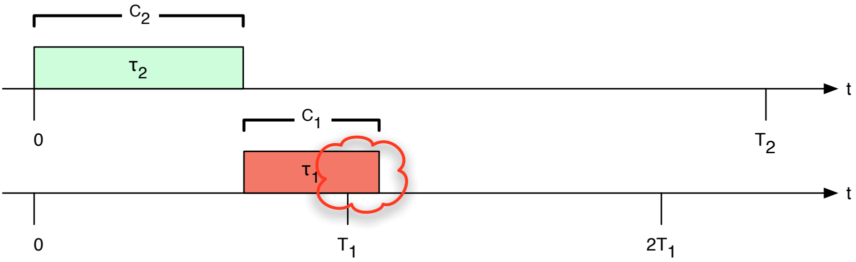
- Starting from the previous theorem and its corollary, the optimality of RM is proved by showing that if a task set is schedulable by an arbitrary (but fixed) priority assignment, then it is also schedulable by RM.
- This result also implies that if RM cannot schedule a certain task set, no other fixed priority assignment algorithm can schedule it.
- We will start with a simpler lemma, that only involves two tasks.

Lemma

If the set of two tasks τ_1, τ_2 is schedulable by any arbitrary, but fixed, priority assignment, then it is schedulable by RM as well.

Feasibility Conditions (I)

Let us consider two tasks, τ_1 and τ_2, with $T_1 < T_2$. If their priorities are not assigned according to RM, then τ_2 will have a priority higher than τ_1. At a critical instant, their situation is:

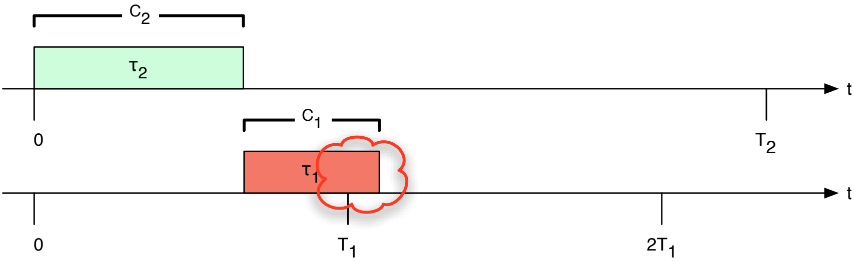


The schedule is feasible if (and only if) the following inequality is satisfied:

$$C_1 + C_2 \leq T_1$$

Feasibility Conditions (I)

Let us consider two tasks, τ_1 and τ_2, with $T_1 < T_2$. If their priorities are not assigned according to RM, then τ_2 will have a priority higher than τ_1. At a critical instant, their situation is:



The schedule is feasible if (and only if) the following inequality is satisfied:

$$C_1 + C_2 \leq T_1$$

Feasibility Conditions (II)

If priorities are assigned according to RM, then task τ_1 will have a priority higher than τ_2.

If we let F be the number of periods of τ_1 entirely contained within T_2:

$$F = \left\lfloor \frac{T_2}{T_1} \right\rfloor \quad ,$$

then, in order to determine the feasibility conditions, we must consider two cases (which cover all possible situations):

1. The execution time C_1 is "short enough" so that the all the instances of τ_1 within the critical zone of τ_2 are completed before the next release of τ_2.
2. The execution of the last instance of τ_1 that starts within the critical zone of τ_2 overlaps the next release of τ_2.

Feasibility Conditions (II)

If priorities are assigned according to RM, then task τ_1 will have a priority higher than τ_2.

If we let F be the number of periods of τ_1 entirely contained within T_2:

$$F = \left\lfloor \frac{T_2}{T_1} \right\rfloor \quad ,$$

then, in order to determine the feasibility conditions, we must consider two cases (which cover all possible situations):

1. The execution time C_1 is "short enough" so that the all the instances of τ_1 within the critical zone of τ_2 are completed before the next release of τ_2.
2. The execution of the last instance of τ_1 that starts within the critical zone of τ_2 overlaps the next release of τ_2.

Feasibility Conditions (II)

If priorities are assigned according to RM, then task τ_1 will have a priority higher than τ_2.

If we let F be the number of periods of τ_1 entirely contained within T_2:

$$F = \left\lfloor \frac{T_2}{T_1} \right\rfloor ,$$

then, in order to determine the feasibility conditions, we must consider two cases (which cover all possible situations):

1. The execution time C_1 is "short enough" so that the all the instances of τ_1 within the critical zone of τ_2 are completed before the next release of τ_2.
2. The execution of the last instance of τ_1 that starts within the critical zone of τ_2 overlaps the next release of τ_2.

Feasibility Conditions (II)

If priorities are assigned according to RM, then task τ_1 will have a priority higher than τ_2.

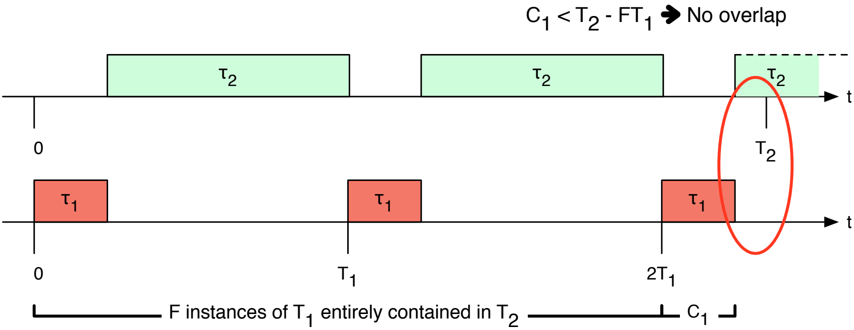
If we let F be the number of periods of τ_1 entirely contained within T_2:

$$F = \left\lfloor \frac{T_2}{T_1} \right\rfloor ,$$

then, in order to determine the feasibility conditions, we must consider two cases (which cover all possible situations):

1. The execution time C_1 is “short enough” so that the all the instances of τ_1 within the critical zone of τ_2 are completed before the next release of τ_2.
2. The execution of the last instance of τ_1 that starts within the critical zone of τ_2 overlaps the next release of τ_2.

First Case



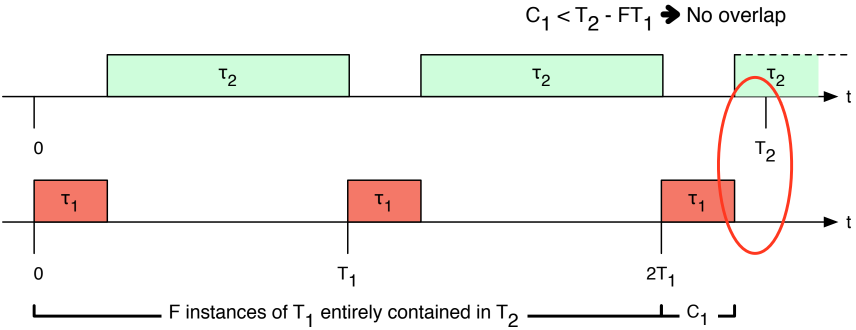
- The first case occurs when:

$$C_1 < T_2 - FT_1$$

- From the figure, we can that the task set is schedulable if and only if:

$$(F + 1)C_1 + C_2 \leq T_2$$

First Case



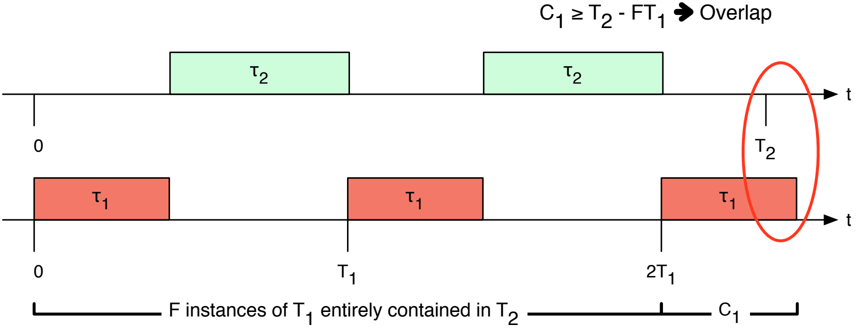
- The first case occurs when:

$$C_1 < T_2 - FT_1$$

- From the figure, we can that the task set is schedulable if and only if:

$$(F+1)C_1 + C_2 \leq T_2$$

Second Case



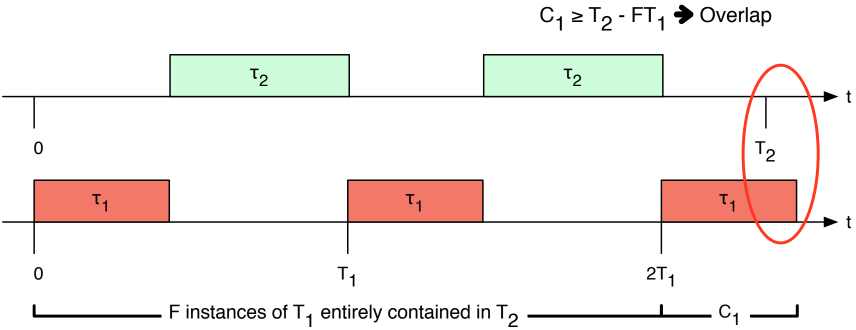
- The second case occurs when:

$$C_1 \geq T_2 - FT_1$$

- From the figure, we can see that the task set is schedulable if and only if:

$$FC_1 + C_2 \leq FT_1$$

Second Case



- The second case occurs when:

$$C_1 \geq T_2 - FT_1$$

- From the figure, we can see that the task set is schedulable if and only if:

$$FC_1 + C_2 \leq FT_1$$

Summary

Given a set of two tasks, τ_1 and τ_2, with $T_1 < T_2$:

1. When priorities are assigned according to RM, the set is schedulable if and only if:
   - $(F+1)C_1 + C_2 \leq T_2$, when $C_1 < T_2 - FT_1$.
   - $FC_1 + C_2 \leq FT_1$, when $C_1 \geq T_2 - FT_1$
2. When priorities are assigned otherwise, the set is schedulable if and only if: $C_1 + C_2 \leq T_1$.

To prove the lemma we must show that the following two implications hold:

1. When $C_1 < T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow (F+1)C_1 + C_2 \leq T_2$
2. When $C_1 \geq T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FT_1$.

Summary

Given a set of two tasks, τ_1 and τ_2, with $T_1 < T_2$:

1. When priorities are assigned according to RM, the set is schedulable if and only if:
   - $(F+1)C_1 + C_2 \leq T_2$, when $C_1 < T_2 - FT_1$.
   - $FC_1 + C_2 \leq FT_1$, when $C_1 \geq T_2 - FT_1$
2. When priorities are assigned otherwise, the set is schedulable if and only if: $C_1 + C_2 \leq T_1$.

To prove the lemma we must show that the following two implications hold:

1. When $C_1 < T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow (F+1)C_1 + C_2 \leq T_2$
2. When $C_1 \geq T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FT_1$.

Summary

Given a set of two tasks, τ_1 and τ_2, with $T_1 < T_2$:

1. When priorities are assigned according to RM, the set is schedulable if and only if:
   - $(F + 1)C_1 + C_2 \leq T_2$, when $C_1 < T_2 - FT_1$.
   - $FC_1 + C_2 \leq FT_1$, when $C_1 \geq T_2 - FT_1$
2. When priorities are assigned otherwise, the set is schedulable if and only if: $C_1 + C_2 \leq T_1$.

To prove the lemma we must show that the following two implications hold:

1. When $C_1 < T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow (F + 1)C_1 + C_2 \leq T_2$
2. When $C_1 \geq T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FT_1$.

Summary

Given a set of two tasks, τ_1 and τ_2, with $T_1 < T_2$:

1. When priorities are assigned according to RM, the set is schedulable if and only if:
   - $(F+1)C_1 + C_2 \leq T_2$, when $C_1 < T_2 - FT_1$.
   - $FC_1 + C_2 \leq FT_1$, when $C_1 \geq T_2 - FT_1$
2. When priorities are assigned otherwise, the set is schedulable if and only if: $C_1 + C_2 \leq T_1$.

To prove the lemma we must show that the following two implications hold:

1. When $C_1 < T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow (F+1)C_1 + C_2 \leq T_2$
2. When $C_1 \geq T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FT_1$.

Summary

Given a set of two tasks, τ_1 and τ_2, with $T_1 < T_2$:

1. When priorities are assigned according to RM, the set is schedulable if and only if:
   - $(F + 1)C_1 + C_2 \leq T_2$, when $C_1 < T_2 - FT_1$.
   - $FC_1 + C_2 \leq FT_1$, when $C_1 \geq T_2 - FT_1$
2. When priorities are assigned otherwise, the set is schedulable if and only if: $C_1 + C_2 \leq T_1$.

To prove the lemma we must show that the following two implications hold:

1. When $C_1 < T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow (F + 1)C_1 + C_2 \leq T_2$
2. When $C_1 \geq T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FT_1$.

Proving the First Implication

When $C_1 < T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow (F+1)C_1 + C_2 \leq T_2$

- If we multiply both members of $C_1 + C_2 \leq T_1$ by F and then add C_1, we obtain:

$$(F+1)C_1 + FC_2 \leq FT_1 + C_1$$

- We know that $F \geq 1$ (otherwise, it would not be $T_1 < T_2$), hence:

$$FC_2 \geq C_2$$

- Moreover, from the hypothesis we have:

$$FT_1 + C_1 < T_2$$

- As a consequence, we have:

$$(F+1)C_1 + C_2 \leq (F+1)C_1 + FC_2 \leq FT_1 + C_1 \leq T_2$$

Proving the First Implication

When $C_1 < T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow (F+1)C_1 + C_2 \leq T_2$

- If we multiply both members of $C_1 + C_2 \leq T_1$ by F and then add C_1, we obtain:

$$(F+1)C_1 + FC_2 \leq FT_1 + C_1$$

- We know that $F \geq 1$ (otherwise, it would not be $T_1 < T_2$), hence:

$$FC_2 \geq C_2$$

- Moreover, from the hypothesis we have:

$$FT_1 + C_1 < T_2$$

- As a consequence, we have:

$$(F+1)C_1 + C_2 \leq (F+1)C_1 + FC_2 \leq FT_1 + C_1 \leq T_2$$

Proving the First Implication

When $C_1 < T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow (F+1)C_1 + C_2 \leq T_2$

- If we multiply both members of $C_1 + C_2 \leq T_1$ by F and then add C_1, we obtain:

$$(F+1)C_1 + FC_2 \leq FT_1 + C_1$$

- We know that $F \geq 1$ (otherwise, it would not be $T_1 < T_2$), hence:

$$FC_2 \geq C_2$$

- Moreover, from the hypothesis we have:

$$FT_1 + C_1 < T_2$$

- As a consequence, we have:

$$(F+1)C_1 + C_2 \leq (F+1)C_1 + FC_2 \leq FT_1 + C_1 \leq T_2$$

Proving the First Implication

When $C_1 < T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow (F+1)C_1 + C_2 \leq T_2$

- If we multiply both members of $C_1 + C_2 \leq T_1$ by F and then add C_1, we obtain:

$$(F+1)C_1 + FC_2 \leq FT_1 + C_1$$

- We know that $F \geq 1$ (otherwise, it would not be $T_1 < T_2$), hence:

$$FC_2 \geq C_2$$

- Moreover, from the hypothesis we have:

$$FT_1 + C_1 < T_2$$

- As a consequence, we have:

$$(F+1)C_1 + C_2 \leq (F+1)C_1 + FC_2 \leq FT_1 + C_1 \leq T_2$$

Proving the First Implication

When $C_1 < T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow (F+1)C_1 + C_2 \leq T_2$

- If we multiply both members of $C_1 + C_2 \leq T_1$ by F and then add C_1, we obtain:

$$(F+1)C_1 + FC_2 \leq FT_1 + C_1$$

- We know that $F \geq 1$ (otherwise, it would not be $T_1 < T_2$), hence:

$$FC_2 \geq C_2$$

- Moreover, from the hypothesis we have:

$$FT_1 + C_1 < T_2$$

- As a consequence, we have:

$$(F+1)C_1 + C_2 \leq (F+1)C_1 + FC_2 \leq FT_1 + C_1 \leq T_2$$

Proving the First Implication

When $C_1 < T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow (F+1)C_1 + C_2 \leq T_2$

- If we multiply both members of $C_1 + C_2 \leq T_1$ by F and then add C_1, we obtain:

$$(F+1)C_1 + FC_2 \leq FT_1 + C_1$$

- We know that $F \geq 1$ (otherwise, it would not be $T_1 < T_2$), hence:

$$FC_2 \geq C_2$$

- Moreover, from the hypothesis we have:

$$FT_1 + C_1 < T_2$$

- As a consequence, we have:

$$(F+1)C_1 + C_2 \leq (F+1)C_1 + FC_2 \leq FT_1 + C_1 \leq T_2$$

Proving the Second Implication

When $C_1 \geq T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FT_1$.

- It we multiply both members of of $C_1 + C_2 \leq T_1$ by F, we obtain:

$$FC_1 + FC_2 \leq FT_1$$

- We know that $F \geq 1$ (otherwise, it would not be $T_1 < T_2$), hence:

$$FC_2 \geq C_2$$

- As a consequence, we have:

$$FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1$$

- This concludes the proof of the lemma.

$\square$

Proving the Second Implication

When $C_1 \geq T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FT_1$.

- It we multiply both members of of $C_1 + C_2 \leq T_1$ by F, we obtain:

$$FC_1 + FC_2 \leq FT_1$$

- We know that $F \geq 1$ (otherwise, it would not be $T_1 < T_2$), hence:

$$FC_2 \geq C_2$$

- As a consequence, we have:

$$FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1$$

- This concludes the proof of the lemma.

□

Proving the Second Implication

When $C_1 \geq T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FT_1$.

- It we multiply both members of of $C_1 + C_2 \leq T_1$ by F, we obtain:

$$FC_1 + FC_2 \leq FT_1$$

- We know that $F \geq 1$ (otherwise, it would not be $T_1 < T_2$), hence:

$$FC_2 \geq C_2$$

- As a consequence, we have:

$$FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1$$

- This concludes the proof of the lemma.

$\square$

Proving the Second Implication

When $C_1 \geq T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FT_1$.

- It we multiply both members of of $C_1 + C_2 \leq T_1$ by F, we obtain:

$$FC_1 + FC_2 \leq FT_1$$

- We know that $F \geq 1$ (otherwise, it would not be $T_1 < T_2$), hence:

$$FC_2 \geq C_2$$

- As a consequence, we have:

$$FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1$$

- This concludes the proof of the lemma.

□

Proving the Second Implication

When $C_1 \geq T_2 - FT_1$, $C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FT_1$.

- It we multiply both members of of $C_1 + C_2 \leq T_1$ by F, we obtain:

$$FC_1 + FC_2 \leq FT_1$$

- We know that $F \geq 1$ (otherwise, it would not be $T_1 < T_2$), hence:

$$FC_2 \geq C_2$$

- As a consequence, we have:

$$FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1$$

- This concludes the proof of the lemma.

$\square$

Extension to n Tasks

Theorem (Liu and Layland, 1973)

If the task set $\tau_1, \ldots, \tau_n$ (n tasks) is schedulable by any arbitrary, but fixed, priority assignment, then it is schedulable by RM as well.

- Let τ_i and τ_j be two tasks of adjacent priorities, τ_i being the higher priority one, and suppose that $T_i > T_j$.
- Having adjacent priorities, both τ_i and τ_j are affected in the same way by the interferences coming from the higher priority tasks (and not at all by the lower priority ones).
- Hence, we can apply the result just obtained and state that if we interchange the priorities of τ_i and τ_j the set is still schedulable.
- Last, we notice that the RM priority assignment can be obtained from any other priority assignment by a sequence of pairwise priority reorderings as above.

Extension to n Tasks

Theorem (Liu and Layland, 1973)

If the task set $\tau_1, \ldots, \tau_n$ (n tasks) is schedulable by any arbitrary, but fixed, priority assignment, then it is schedulable by RM as well.

- Let τ_i and τ_j be two tasks of adjacent priorities, τ_i being the higher priority one, and suppose that $T_i > T_j$.
- Having adjacent priorities, both τ_i and τ_j are affected in the same way by the interferences coming from the higher priority tasks (and not at all by the lower priority ones).
- Hence, we can apply the result just obtained and state that if we interchange the priorities of τ_i and τ_j the set is still schedulable.
- Last, we notice that the RM priority assignment can be obtained from any other priority assignment by a sequence of pairwise priority reorderings as above.

□

Extension to n Tasks

Theorem (Liu and Layland, 1973)

If the task set $\tau_1, \ldots, \tau_n$ (n tasks) is schedulable by any arbitrary, but fixed, priority assignment, then it is schedulable by RM as well.

- Let τ_i and τ_j be two tasks of adjacent priorities, τ_i being the higher priority one, and suppose that $T_i > T_j$.
- Having adjacent priorities, both τ_i and τ_j are affected in the same way by the interferences coming from the higher priority tasks (and not at all by the lower priority ones).
- Hence, we can apply the result just obtained and state that if we interchange the priorities of τ_i and τ_j the set is still schedulable.
- Last, we notice that the RM priority assignment can be obtained from any other priority assignment by a sequence of pairwise priority reorderings as above.

□

Extension to n Tasks

Theorem (Liu and Layland, 1973)

If the task set $\tau_1, \dots, \tau_n$ (n tasks) is schedulable by any arbitrary, but fixed, priority assignment, then it is schedulable by RM as well.

- Let τ_i and τ_j be two tasks of adjacent priorities, τ_i being the higher priority one, and suppose that $T_i > T_j$.
- Having adjacent priorities, both τ_i and τ_j are affected in the same way by the interferences coming from the higher priority tasks (and not at all by the lower priority ones).
- Hence, we can apply the result just obtained and state that if we interchange the priorities of τ_i and τ_j the set is still schedulable.
- Last, we notice that the RM priority assignment can be obtained from any other priority assignment by a sequence of pairwise priority reorderings as above.

Extension to n Tasks

Theorem (Liu and Layland, 1973)

If the task set $\tau_1, \ldots, \tau_n$ (n tasks) is schedulable by any arbitrary, but fixed, priority assignment, then it is schedulable by RM as well.

- Let τ_i and τ_j be two tasks of adjacent priorities, τ_i being the higher priority one, and suppose that $T_i > T_j$.
- Having adjacent priorities, both τ_i and τ_j are affected in the same way by the interferences coming from the higher priority tasks (and not at all by the lower priority ones).
- Hence, we can apply the result just obtained and state that if we interchange the priorities of τ_i and τ_j the set is still schedulable.
- Last, we notice that the RM priority assignment can be obtained from any other priority assignment by a sequence of pairwise priority reorderings as above.

□

The Earliest Deadline First Scheduler

Can we do any better than RM?

We can do better than RM if we abandon the hypothesis of using static priorities.

- The basic process model is still being used.
- Task priorities are assigned dynamically.
- They are scheduled preemptively.
- There is only one processor.

An optimum scheduling algorithm

Under these hypotheses, there exists an optimum scheduling algorithm, due to Liu and Layland (1973) and known as Earliest Deadline First (EDF).

The Earliest Deadline First Scheduler

Can we do any better than RM?

We can do better than RM if we abandon the hypothesis of using static priorities.

- The basic process model is still being used.
- Task priorities are assigned dynamically.
- They are scheduled preemptively.
- There is only one processor.

An optimum scheduling algorithm

Under these hypotheses, there exists an optimum scheduling algorithm, due to Liu and Layland (1973) and known as Earliest Deadline First (EDF).

The Earliest Deadline First Scheduler

Can we do any better than RM?

We can do better than RM if we abandon the hypothesis of using static priorities.

- The basic process model is still being used.
- Task priorities are assigned dynamically.
- They are scheduled preemptively.
- There is only one processor.

An optimum scheduling algorithm

Under these hypotheses, there exists an optimum scheduling algorithm, due to Liu and Layland (1973) and known as Earliest Deadline First (EDF).

The Earliest Deadline First Scheduler

Can we do any better than RM?

We can do better than RM if we abandon the hypothesis of using static priorities.

- The basic process model is still being used.
- Task priorities are assigned dynamically.
- They are scheduled preemptively.
- There is only one processor.

An optimum scheduling algorithm

Under these hypotheses, there exists an optimum scheduling algorithm, due to Liu and Layland (1973) and known as Earliest Deadline First (EDF).

The Earliest Deadline First Scheduler

Can we do any better than RM?

We can do better than RM if we abandon the hypothesis of using static priorities.

- The basic process model is still being used.
- Task priorities are assigned dynamically.
- They are scheduled preemptively.
- There is only one processor.

An optimum scheduling algorithm

Under these hypotheses, there exists an optimum scheduling algorithm, due to Liu and Layland (1973) and known as Earliest Deadline First (EDF).

The Earliest Deadline First Scheduler

Can we do any better than RM?

We can do better than RM if we abandon the hypothesis of using static priorities.

- The basic process model is still being used.
- Task priorities are assigned dynamically.
- They are scheduled preemptively.
- There is only one processor.

An optimum scheduling algorithm

Under these hypotheses, there exists an optimum scheduling algorithm, due to Liu and Layland (1973) and known as Earliest Deadline First (EDF).

EDF Priority Assignment

The EDF algorithm selects tasks according to their absolute deadlines. Namely, at each instant, tasks with earlier deadlines will receive higher priorities.

- The absolute deadline $d_{i,j}$ of the j-th instance of task τ_i is:

$$d_{i,j} = \Phi_i + jT_i + D_i \ ,$$

  where Φ_i is the phase of τ_i, that is, the release time of its first instance (for which $j = 0$).
- Hence, the priority of each task is assigned dynamically, instance by instance, but the priority of each instance (job) is still fixed.
- EDF is optimum in the sense that if any task set is schedulable by any scheduling algorithm, under the hypotheses just set out, then it is also schedulable by EDF.

EDF Priority Assignment

The EDF algorithm selects tasks according to their absolute deadlines. Namely, at each instant, tasks with earlier deadlines will receive higher priorities.

- The absolute deadline $d_{i,j}$ of the j-th instance of task τ_i is:

$$d_{i,j} = \Phi_i + jT_i + D_i \quad ,$$

where Φ_i is the phase of τ_i, that is, the release time of its first instance (for which $j = 0$).

- Hence, the priority of each task is assigned dynamically, instance by instance, but the priority of each instance (job) is still fixed.
- EDF is optimum in the sense that if any task set is schedulable by any scheduling algorithm, under the hypotheses just set out, then it is also schedulable by EDF.

EDF Priority Assignment

The EDF algorithm selects tasks according to their absolute deadlines. Namely, at each instant, tasks with earlier deadlines will receive higher priorities.

- The absolute deadline $d_{i,j}$ of the j-th instance of task τ_i is:

$$d_{i,j} = \Phi_i + jT_i + D_i \quad ,$$

  where Φ_i is the phase of τ_i, that is, the release time of its first instance (for which $j = 0$).
- Hence, the priority of each task is assigned dynamically, instance by instance, but the priority of each instance (job) is still fixed.
- EDF is optimum in the sense that if any task set is schedulable by any scheduling algorithm, under the hypotheses just set out, then it is also schedulable by EDF.

EDF Priority Assignment

The EDF algorithm selects tasks according to their absolute deadlines. Namely, at each instant, tasks with earlier deadlines will receive higher priorities.

- The absolute deadline $d_{i,j}$ of the j-th instance of task τ_i is:

  $$d_{i,j} = \Phi_i + jT_i + D_i \quad ,$$

  where Φ_i is the phase of τ_i, that is, the release time of its first instance (for which $j = 0$).
- Hence, the priority of each task is assigned dynamically, instance by instance, but the priority of each instance (job) is still fixed.
- EDF is optimum in the sense that if any task set is schedulable by any scheduling algorithm, under the hypotheses just set out, then it is also schedulable by EDF.

FPS Versus EDF

- FPS is easier to implement than EDF, as the scheduling attribute (priority) is static.
- EDF requires a more complex run-time system, which will typically have a higher overhead.
- During overload situations, the behavior of FPS is easier to predict (the lower-priority processes will miss their deadlines first).
- EDF is less predictable, and can experience a domino effect in which a large number of tasks unnecessarily miss their deadline.
- As for processor utilization, EDF is always able to exploit the full processor capacity, whereas FPS in the worst case does not.

FPS Versus EDF

- FPS is easier to implement than EDF, as the scheduling attribute (priority) is static.
- EDF requires a more complex run-time system, which will typically have a higher overhead.
- During overload situations, the behavior of FPS is easier to predict (the lower-priority processes will miss their deadlines first).
- EDF is less predictable, and can experience a domino effect in which a large number of tasks unnecessarily miss their deadline.
- As for processor utilization, EDF is always able to exploit the full processor capacity, whereas FPS in the worst case does not.

FPS Versus EDF

- FPS is easier to implement than EDF, as the scheduling attribute (priority) is static.
- EDF requires a more complex run-time system, which will typically have a higher overhead.
- During overload situations, the behavior of FPS is easier to predict (the lower-priority processes will miss their deadlines first).
- EDF is less predictable, and can experience a domino effect in which a large number of tasks unnecessarily miss their deadline.
- As for processor utilization, EDF is always able to exploit the full processor capacity, whereas FPS in the worst case does not.

FPS Versus EDF

- FPS is easier to implement than EDF, as the scheduling attribute (priority) is static.
- EDF requires a more complex run-time system, which will typically have a higher overhead.
- During overload situations, the behavior of FPS is easier to predict (the lower-priority processes will miss their deadlines first).
- EDF is less predictable, and can experience a domino effect in which a large number of tasks unnecessarily miss their deadline.
- As for processor utilization, EDF is always able to exploit the full processor capacity, whereas FPS in the worst case does not.

FPS Versus EDF

- FPS is easier to implement than EDF, as the scheduling attribute (priority) is static.
- EDF requires a more complex run-time system, which will typically have a higher overhead.
- During overload situations, the behavior of FPS is easier to predict (the lower-priority processes will miss their deadlines first).
- EDF is less predictable, and can experience a domino effect in which a large number of tasks unnecessarily miss their deadline.
- As for processor utilization, EDF is always able to exploit the full processor capacity, whereas FPS in the worst case does not.