Problem (1): A hard real-time system is required to run three periodic tasks with periods of 10 ms, 25 ms and 45 ms, and execution times of 3 ms, 6 ms, and 19 ms per period respectively. The deadline of each task is equal to its period.

a) Is the above task set schedulable using the RM algorithm? Explain your answer.
b) If the task set is not RM schedulable, can this be changed by increasing the processor speed? If your answer is yes, find the minimum factor by which the processor speed must be increased to achieve schedulability (Assuming that periods will not be changed).

Problem (2): A hard real-time system is required to run three periodic tasks with periods of 10, 25, and 45 ms, and execution times of 3, 6, and 20 ms per period respectively. The deadline of each task is equal to its period.

a) Is the above task set RM schedulable? Explain your answer.
b) Repeat part (a) for the case of EDF scheduling.
c) If the execution time or period of the first task can be changed:
i) find the least reduction in the execution time to achieve RM schedulability.
ii) find the least increase in the period to achieve RM schedulability.

Problem (3): A hard real-time system runs two independent periodic tasks with periods of 50, and 20 ms, and execution times of 10 and 5 ms per period respectively. A third task with period of 70 ms is required also to run on the system.

a) Find the maximum possible execution time for the third task under RM scheduling, assuming that the deadline of each task is equal to its period.
b) Repeat part (a) assuming EDF scheduling.
c) Use response-time analysis to find if the execution time obtained in part (a) is still possible using Deadline Monotonic scheduling if third task has a deadline of 45 ms (with its period still 70 ms).

Real-Time Scheduling with Multiple Processors

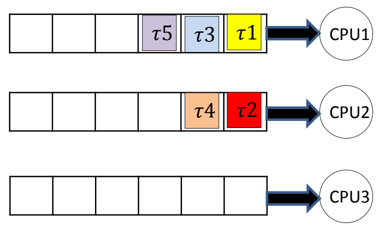
It may be impossible to satisfy the requirements of a real-time system using a single processor. Multiple processors (or cores) will then be needed to implement the system.

New problems will arise, e.g. assignment of tasks to processors, effect of communication delays, effect of shared cache and coordination of processor schedules.

Multiprocessor system may be:

- Homogeneous: using identical processors
- Uniform: similar processors working at different supply voltages/ frequencies.
- Heterogeneous: processors have different instruction sets.

Partitioned vs. Global Scheduling



In partitioned scheduling, each task is assigned to one processor selected offline. The task is executed on the same processor each time it is requested (no "task migration").

A real-time scheduling policy is used to schedule tasks of each processor.

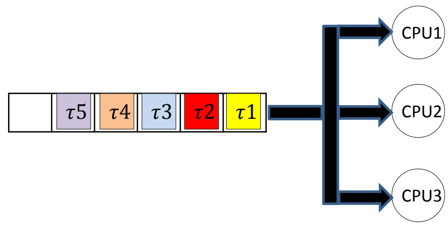
Partitioned vs. Global Scheduling

This approach is equivalent to multiple uniprocessor systems. Many results of uniprocessor scheduling are thus still useful.

However, partitioned scheduling may result in low processor utilization: a processor may stay idle while a task waits for another processor.

Assigning and scheduling problems with multiple processors are computationally intractable (NP-complete problem), and thus heuristic approaches are often employed.

Partitioned vs. Global Scheduling



In global scheduling, all ready tasks are kept in a common queue that is shared among the processors. Whenever a processor becomes idle, a task from the queue is selected for execution on that processor.

Partitioned vs. Global Scheduling

Task assignment is determined on-line with task migration allowed. Thus, a task may run on a different processor when released again or after being pre-empted.

Better usage of processors is expected. However, few of the results obtained for a single processor generalize directly to the multiple processor case.

Partitioned Scheduling

In the following, we assume partitioned scheduling on identical processors.

If tasks are independent, periodic, and pre-emptable; and the deadline of each task is equal to its period, each processor can use RM or EDF scheduling.

The problem of minimizing the required number of processors is equivalent to the well-known problem of *bin-packing*. Heuristics for solving this problem can be applied (e.g. first-fit assignment).

EDF with First-Fit decreasing assignment

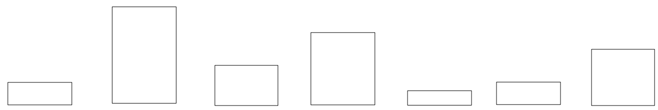
The first step in task assignment is to find the minimum number of needed processors.

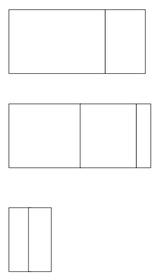
If all processors use EDF, the total utilization of tasks assigned to each processor cannot exceed 1. The following algorithm is known to result in sub-optimal, but acceptable results:

- Arrange tasks in non-increasing order of utilizations.
- Assign next task to the *first processor that will not be overloaded* by this task. If no such processor exists, add a new processor.

EDF with First-Fit decreasing assignment

τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7
0.2	0.7	0.25	0.5	0.1	0.2	0.4





Processor 1	τ_2, τ_3	$\sum u_i = 0.95$
Processor 2	τ_4, τ_7, τ_5	$\sum u_i = 1$
Processor 3	τ_1, τ_6	$\sum u_i = 0.4$

EDF with First-Fit decreasing assignment

For large number of processors, the above algorithm is known to result in no more than 1.22 of the minimum number of processors.

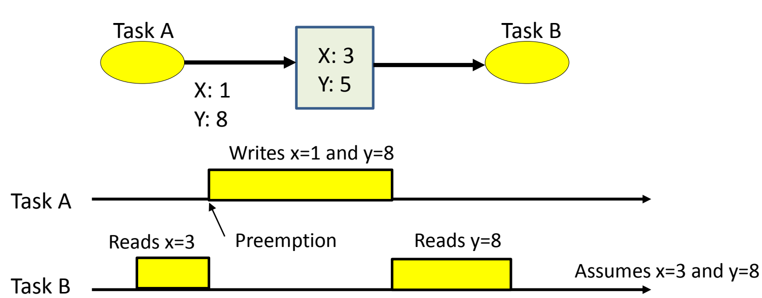
With RM scheduling, we have a *variable-size* bin-packing problem, which is more complex than the uniform–size case.

Process Interaction and communication

In many cases, running processes (or threads) will not run independently of each other:

- ❑ They may cooperate within a given application and need to exchange information.
- ❑ They may need to synchronize their operation in some way, e.g. to avoid using a certain data item before being set up properly.
- ❑ There may be a need to avoid problems resulting from competition on shared resources (e.g. in "critical sections").

Process Interaction and communication



Task should lock the resource while accessing it. A task trying to access a locked resource cannot run until the resource becomes available (Mutual exclusion of critical sections).

Mutual Exclusion

What are the disadvantages of the shown simple solution?

```
Shared int lock=0;
………
while (lock){};
lock =1;
Access_data_item();
lock =0;
```


Semaphores

A semaphore is a shared integer variable on which the following two *system* functions are defined

Down(s)

```
If s ≥ 1 then s:= s-1
else block the calling process
```

Up(s)

```
If there are processes blocked by down(s)
then unblock one of them
else s:= s+1;
```

down( ) and up( ) are executed *without interruption*.

Mutual Exclusion with Semaphores

We can enforce mutual exclusion using semaphores as follows: (m initially=1)

Process 1

```
……
<non-critical>
down(m);
critical section;
up(m);
<non-critical>
……
```

Process 2

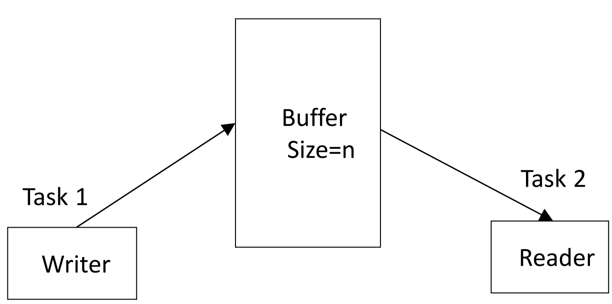
```
……
<non-critical>
down(m);
critical section;
up(m);
<non-critical>
……
```

Process 3

```
……
<non-critical>
down(m);
critical section;
up(m);
<non-critical>
……
```


The Reader/Writer Problem

Task 1 prepares data items and stores them in a buffer that can hold up to n items. If it tries to write while buffer is full, it should be blocked. Task 2 independently read and remove items from buffer. If buffer is empty, task 2 should be blocked.



The Reader/Writer Problem

We use two semaphores, empty (initially = n) and full (initially =0).

Writer code

```
..
..
down(empty);
Write_item ;
up(full);
..
..
```

Reader code

```
..
..
down(full);
Read_item ;
up(empty);
..
..
```


Signaling using Semaphores

<u>Task 1</u>	<u>Task 2</u>	<u>Task 3</u>
...	...	...
fn1()	fn2()	fn3()
...	...	...

How to use semaphores such that fn2() in task 2 is not executed until fn1() ends execution, and fn3() in task 3 is not executed until fn2() ends execution.