

# Multiprocessor platforms for real-time systems

---

Why?

Models of multiprocessor systems

Scheduling policies for multiprocessor systems

Schedulability tests

# The advantages of multiprocessor systems

---

- Greater computational power (obviously!)
- Power savings
  - More slower processors when compared to a few fast (power-hungry) processors
  - Easier heat dissipation
- Reliability
  - Backups for critical tasks
  - Migrations when some processors fail
- Security and isolation
  - Critical tasks can be separated from non-critical tasks

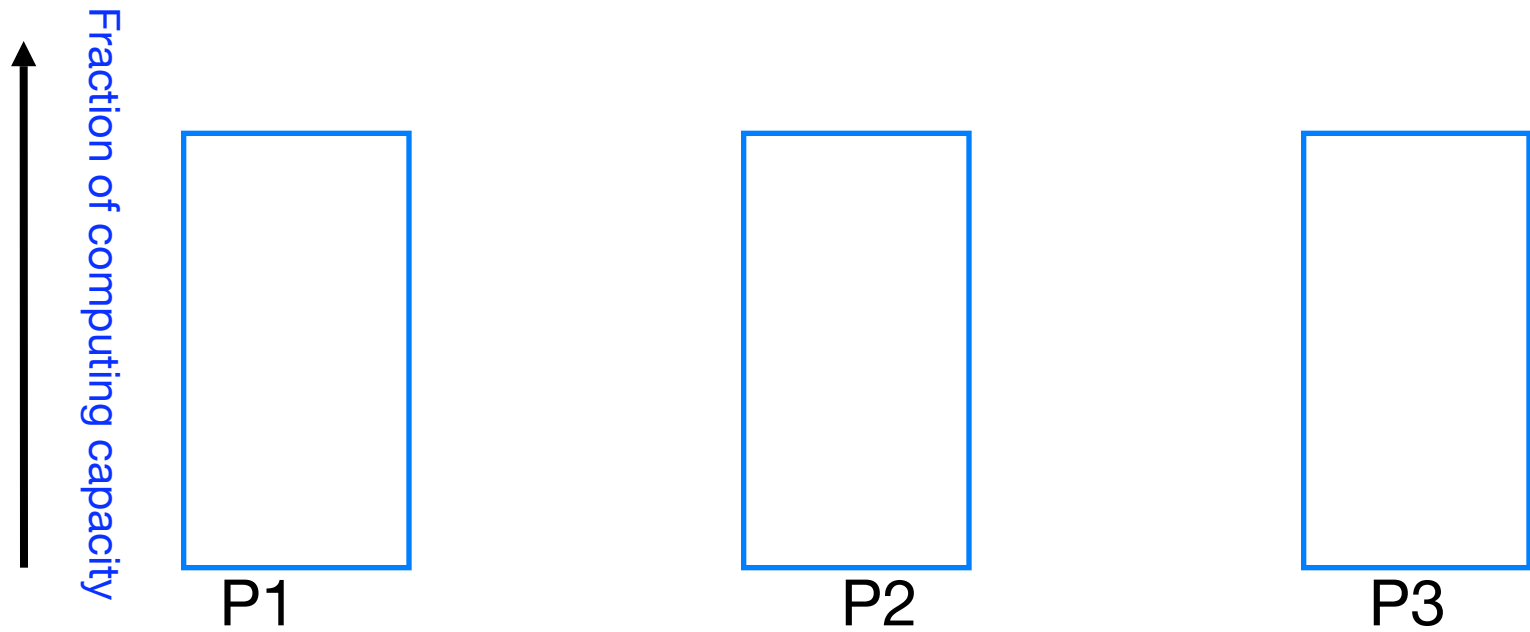
# Models of multiprocessor systems

---

- **Identical** multiprocessors
  - Each processor has the **same computing capacity**
- **Uniform** multiprocessors
  - Different processors have **different computing capacities**
  - The faster a processor is, the lower the execution time of a task
- **Heterogeneous** multiprocessors
  - Each **(task, processor) pair** may have a different computational attribute
  - Execution times of a task may vary from processor to processor but there is no well-defined relationship

# Multiprocessor models

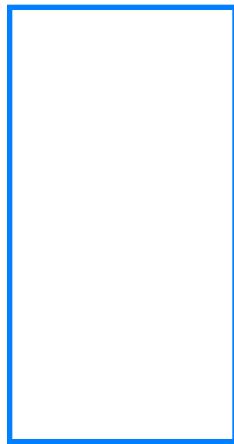
---



# Identical multiprocessors

---

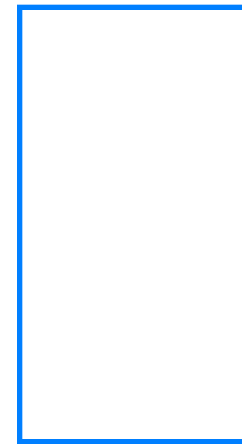
Task T1



P1



P2

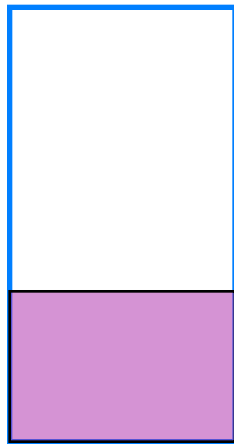


P3

# Identical multiprocessors

---

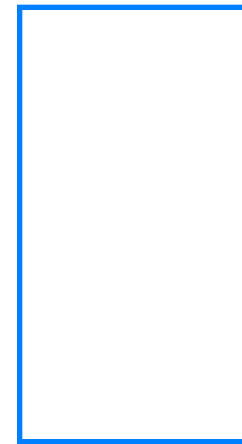
Task T1



P1



P2

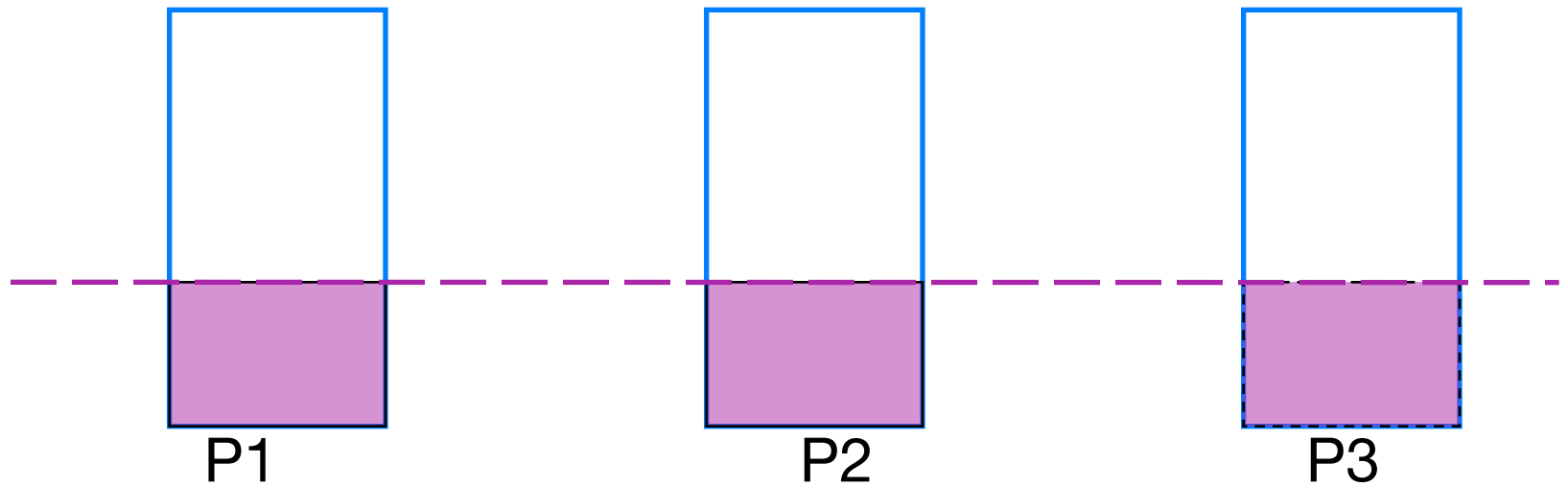


P3

# Identical multiprocessors

---

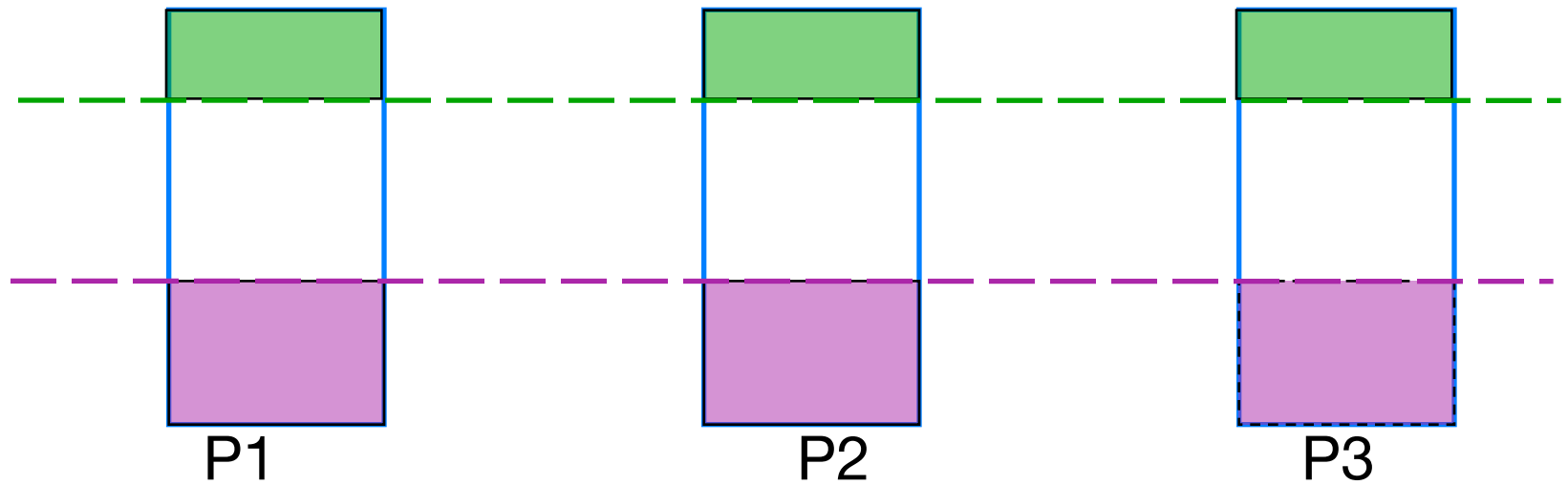
Task T1



# Identical multiprocessors

---

Task T1    Task T2

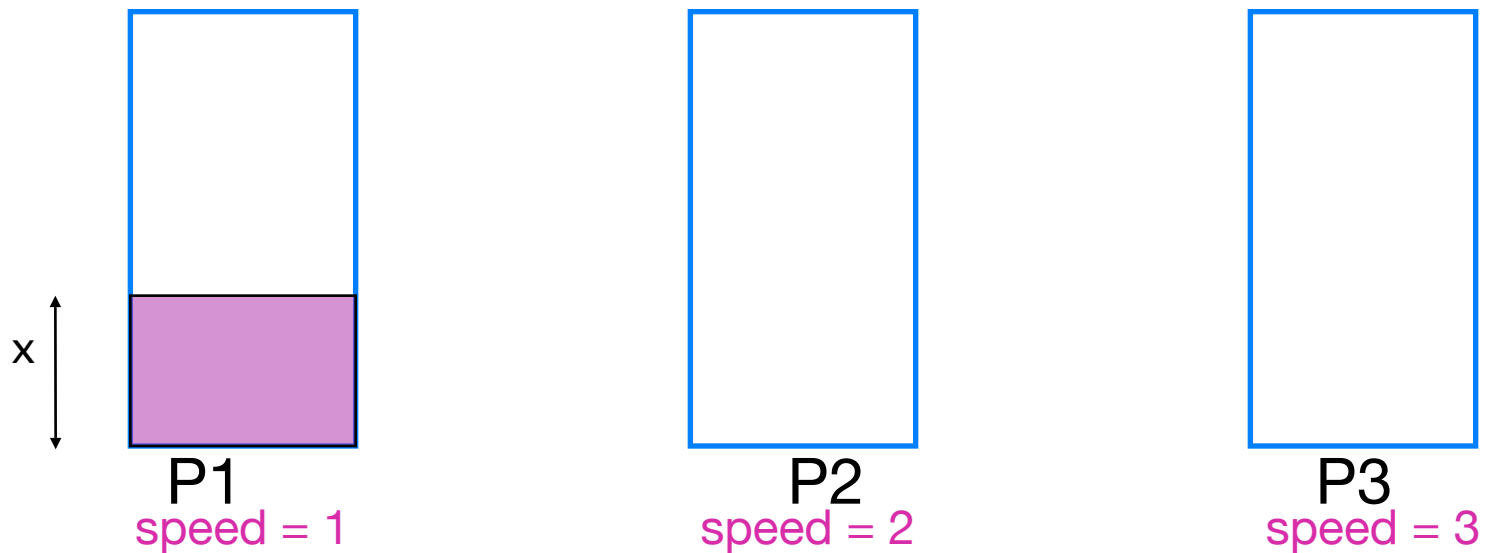




# Uniform multiprocessors

---

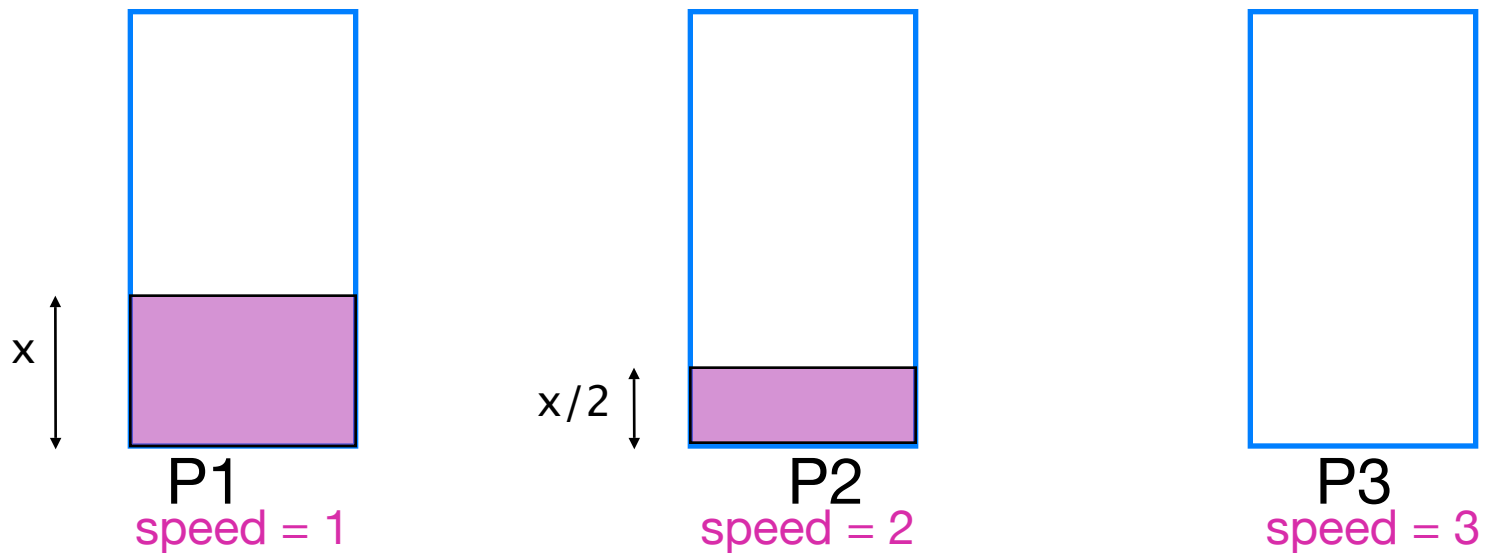
Task T1



# Uniform multiprocessors

---

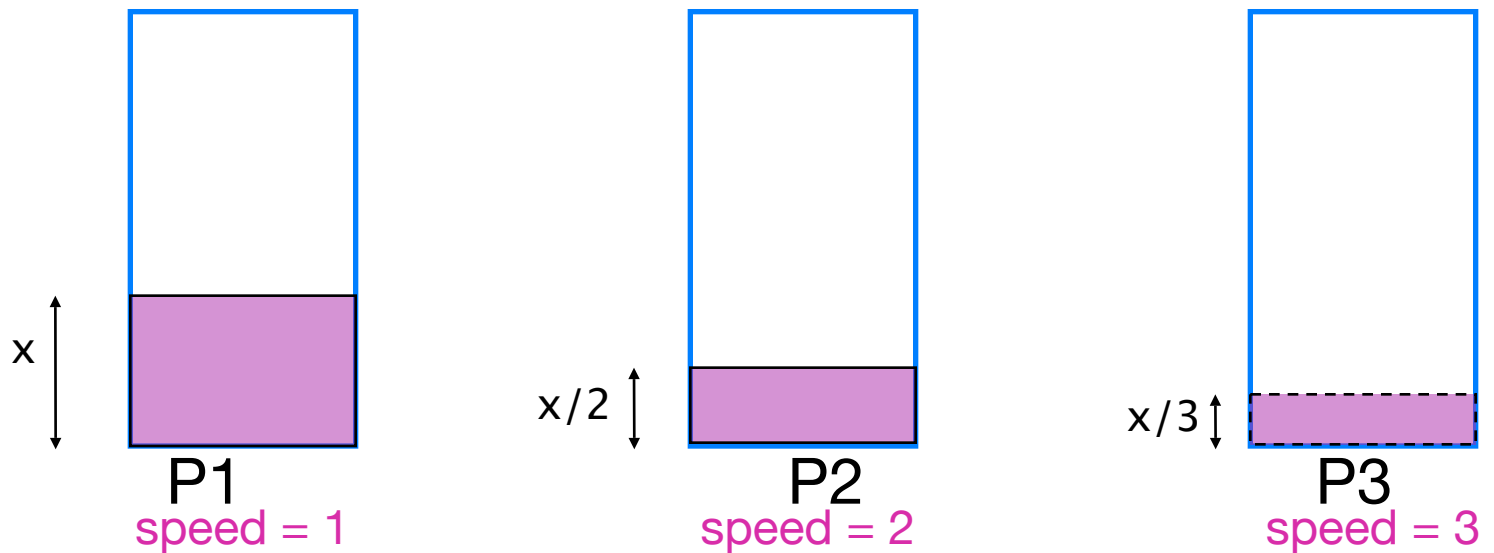
Task T1



# Uniform multiprocessors

---

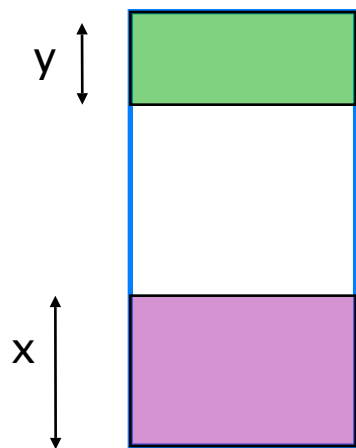
Task T1



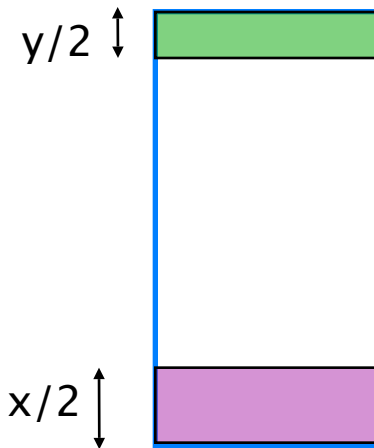
# Uniform multiprocessors

---

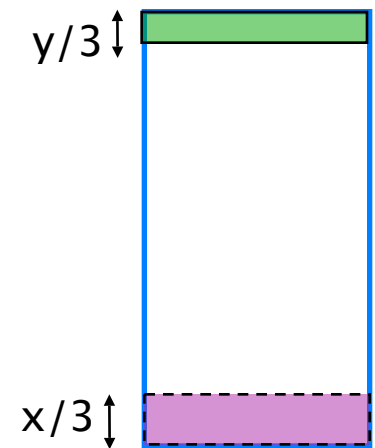
Task T1    Task T2



**P1**  
speed = 1



**P2**  
speed = 2

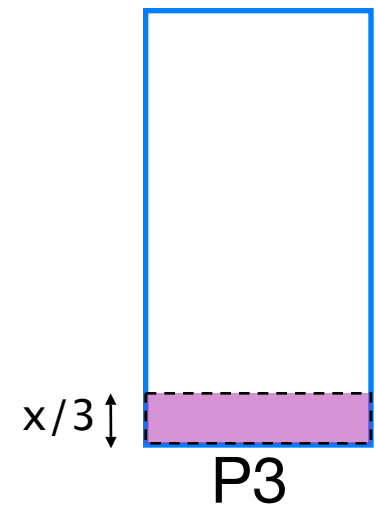
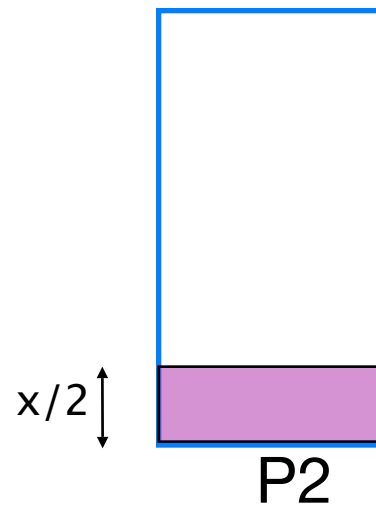
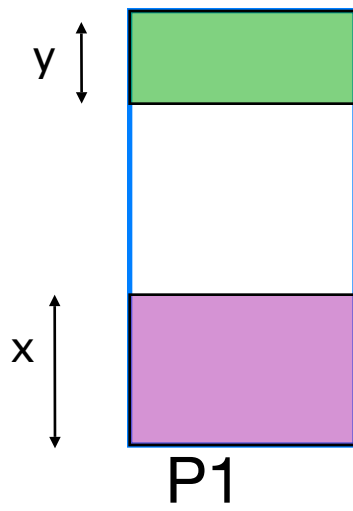


**P3**  
speed = 3

# Heterogeneous multiprocessors

---

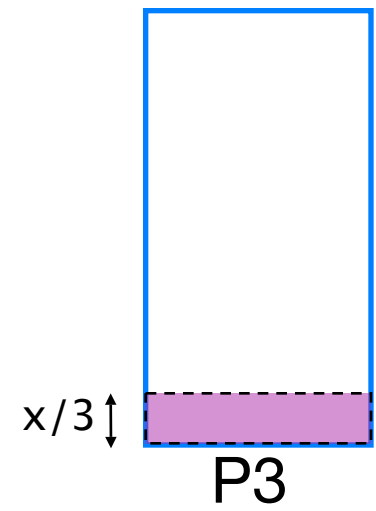
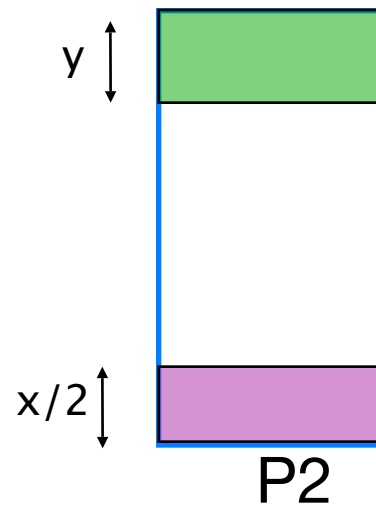
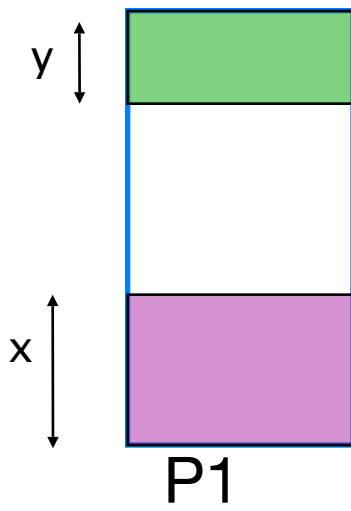
Task T1    Task T2



# Heterogeneous multiprocessors

---

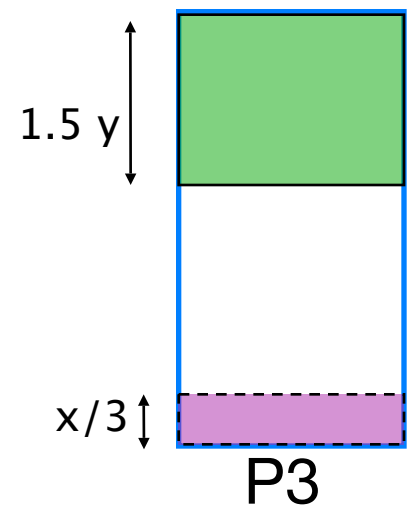
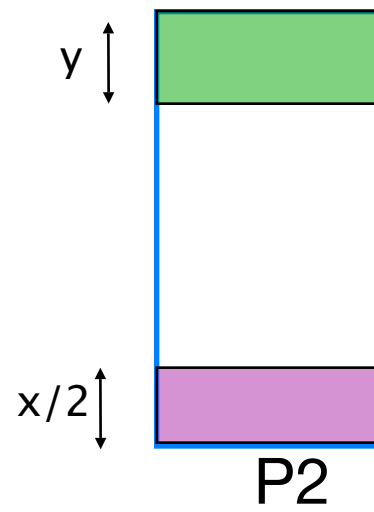
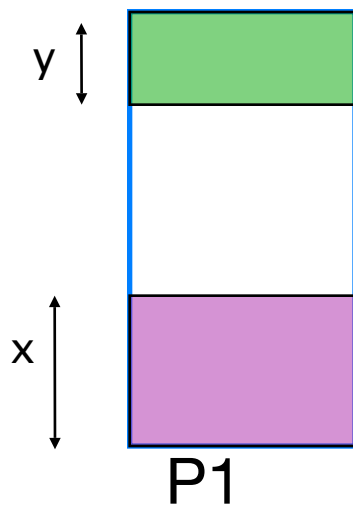
Task T1    Task T2



# Heterogeneous multiprocessors

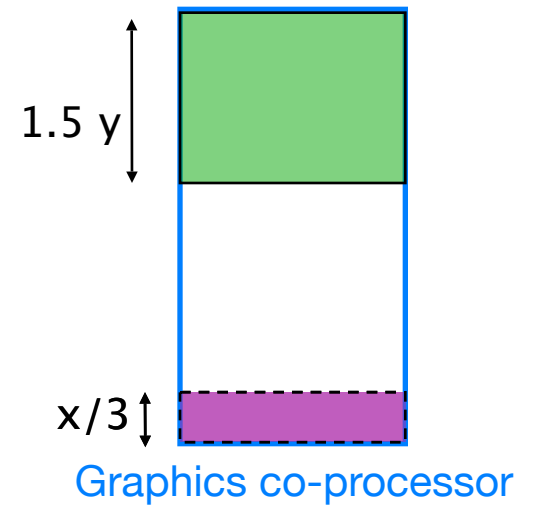
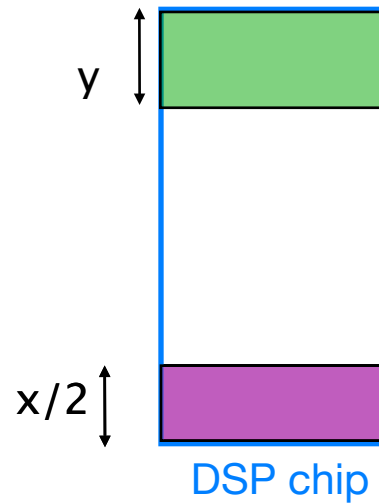
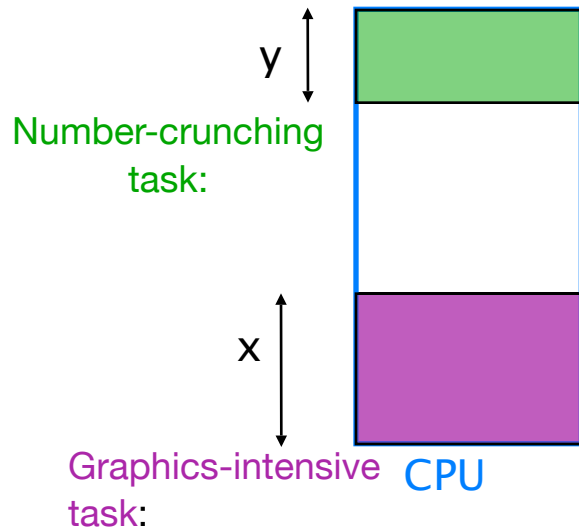
---

Task T1    Task T2



# Heterogeneous multiprocessors

---

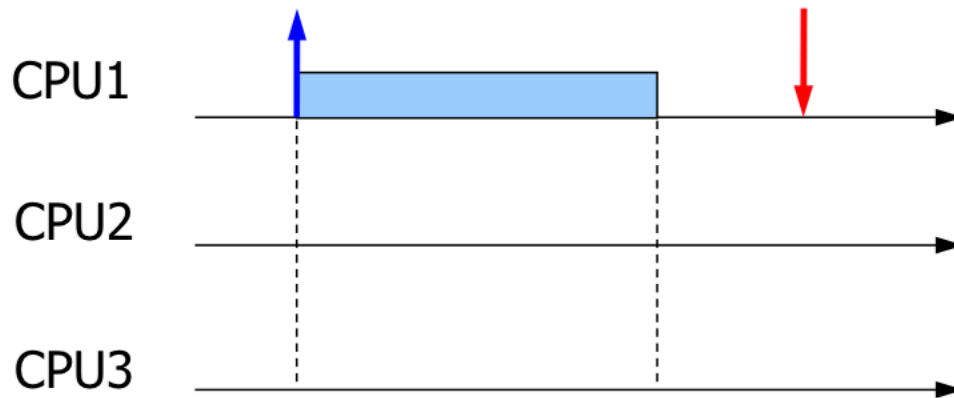




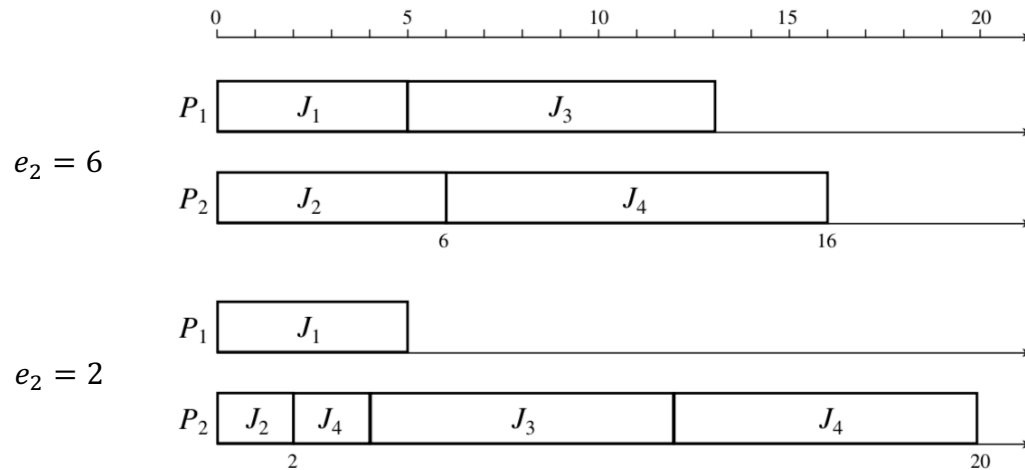
# Multiprocessor scheduling is difficult!

---

*“The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors” [Liu 1969]*



# Multiprocessor scheduling: Anomaly



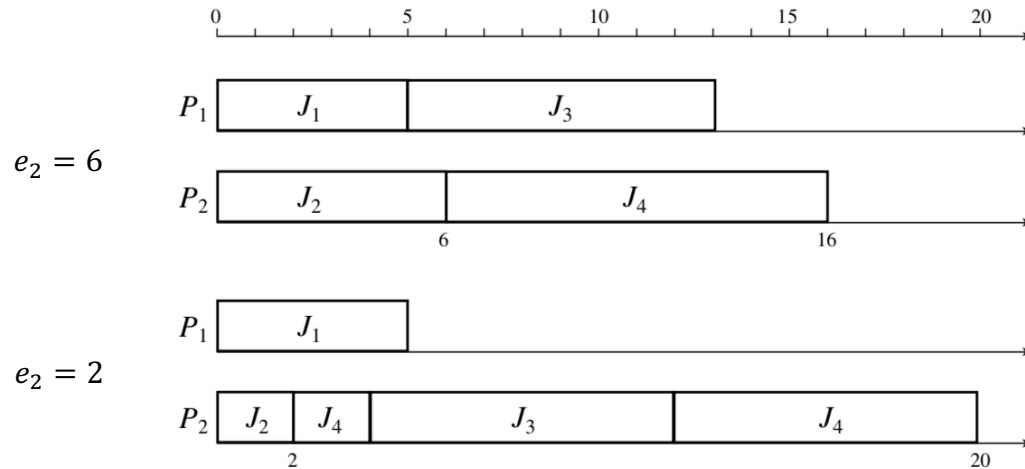
|       | $r_i$ | $d_i$ | $[e_i^-, e_i^+]$ |
|-------|-------|-------|------------------|
| $J_1$ | 0     | 10    | 5                |
| $J_2$ | 0     | 10    | [2, 6]           |
| $J_3$ | 4     | 15    | 8                |
| $J_4$ | 0     | 20    | 10               |

Jobs cannot migrate across processors but can be preempted on the processor to which they are assigned

Jobs are scheduled according to their priorities:

$$\pi_1 > \pi_2 > \pi_3 > \pi_4$$

# Multiprocessor scheduling: Anomaly



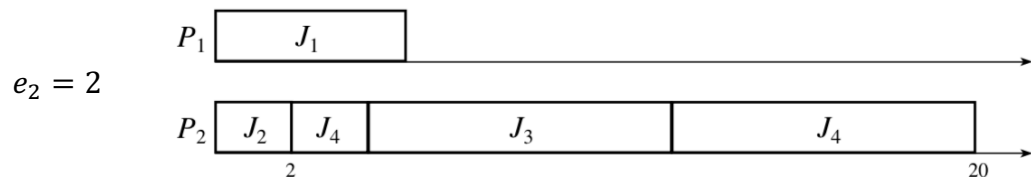
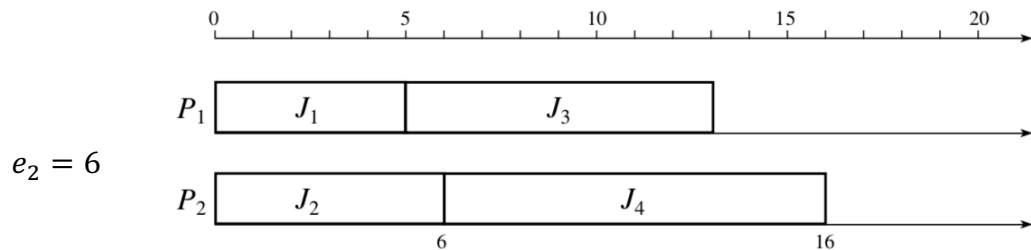
|       | $r_i$ | $d_i$ | $[e_i^-, e_i^+]$ |
|-------|-------|-------|------------------|
| $J_1$ | 0     | 10    | 5                |
| $J_2$ | 0     | 10    | [2, 6]           |
| $J_3$ | 4     | 15    | 8                |
| $J_4$ | 0     | 20    | 10               |

Is this sufficient to conclude that all jobs meet their deadlines?

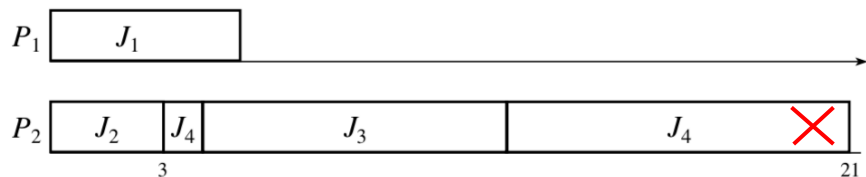
Jobs cannot migrate across processors but can be preempted on the processor to which they are assigned

Jobs are scheduled according to their priorities:  
 $\pi_1 > \pi_2 > \pi_3 > \pi_4$

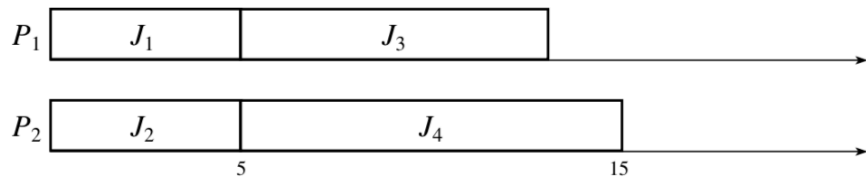
# Multiprocessor scheduling: Anomaly



**Worst case for  $J_4$**   
is when  $e_2 = 3$



**Best case for  $J_4$**   
is when  $e_2 = 5$

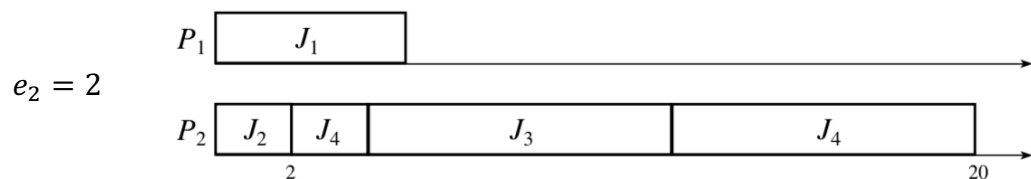
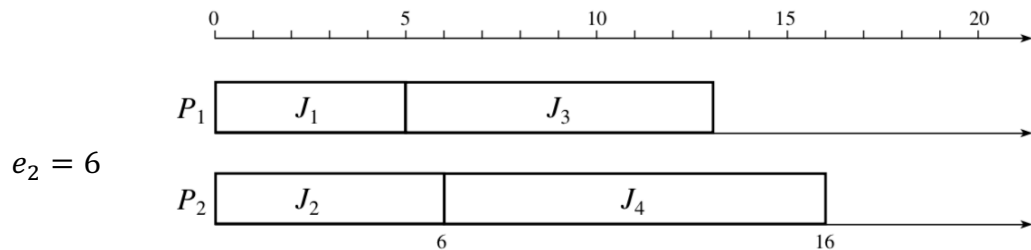


|       | $r_i$ | $d_i$ | $[e_i^-, e_i^+]$ |
|-------|-------|-------|------------------|
| $J_1$ | 0     | 10    | 5                |
| $J_2$ | 0     | 10    | [2, 6]           |
| $J_3$ | 4     | 15    | 8                |
| $J_4$ | 0     | 20    | 10               |

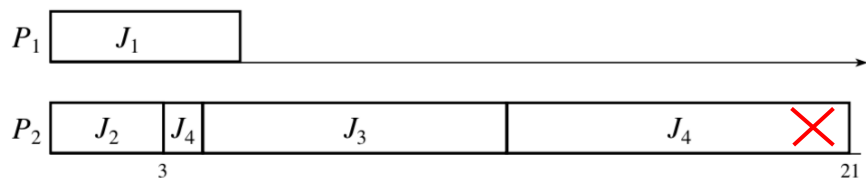
Jobs cannot migrate across processors but can be preempted on the processor to which they are assigned

Jobs are scheduled according to their priorities:  
 $\pi_1 > \pi_2 > \pi_3 > \pi_4$

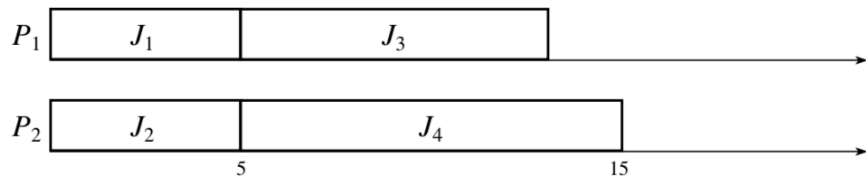
# Multiprocessor scheduling: Anomaly



**Worst case for  $J_4$**   
is when  $e_2 = 3$



**Best case for  $J_4$**   
is when  $e_2 = 5$



|       | $r_i$ | $d_i$ | $[e_i^-, e_i^+]$ |
|-------|-------|-------|------------------|
| $J_1$ | 0     | 10    | 5                |
| $J_2$ | 0     | 10    | [2, 6]           |
| $J_3$ | 4     | 15    | 8                |
| $J_4$ | 0     | 20    | 10               |

Jobs cannot migrate across processors but can be preempted on the processor to which they are assigned

Jobs are scheduled according to their priorities:  
 $\pi_1 > \pi_2 > \pi_3 > \pi_4$

If  $X = \max_{i \in [n]} (e_i^+ - e_i^-)$ , then will need  $O(X^n)$  time to verify schedulability!

# Resource management for real-time systems

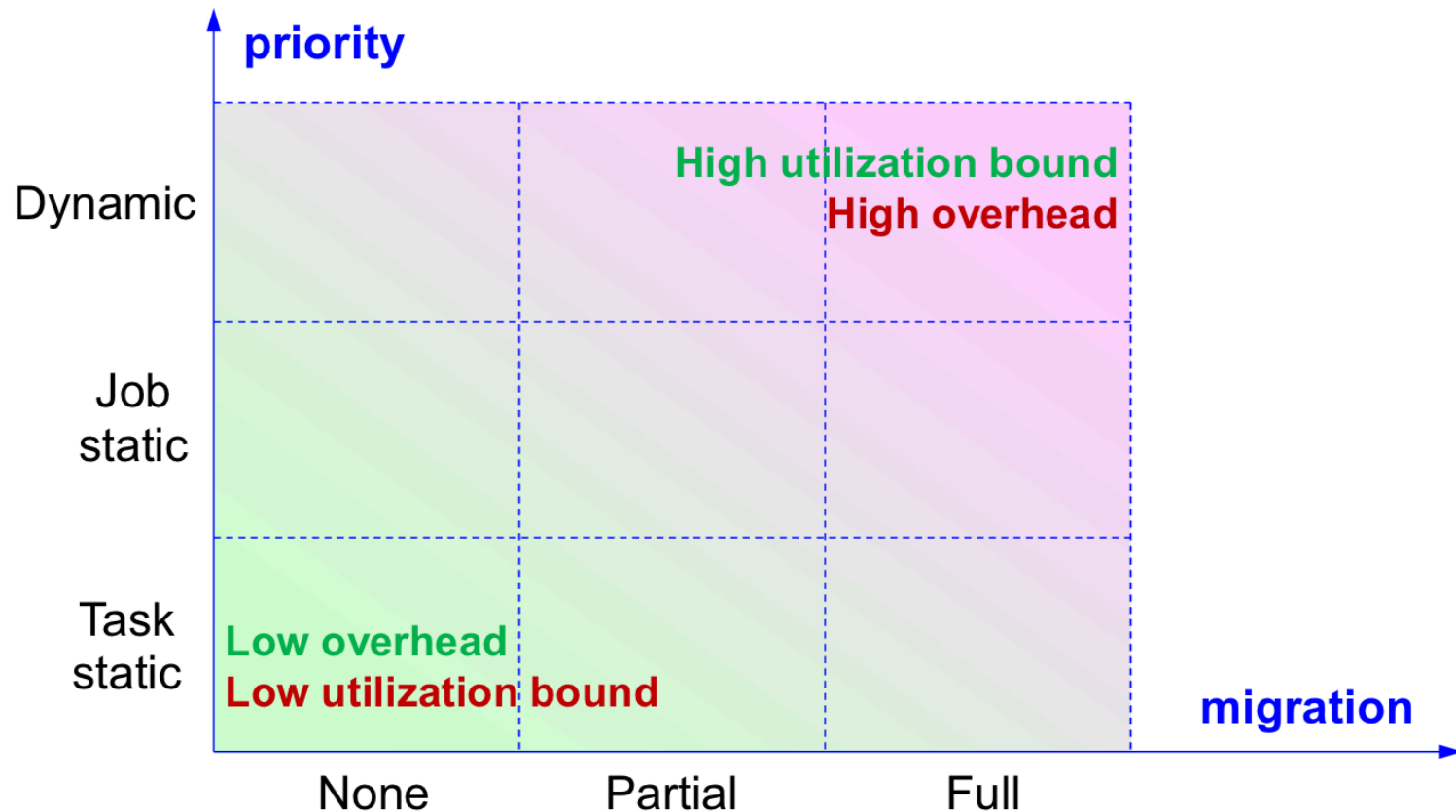
---

- Given a multiprocessing platform and a set of recurring tasks with deadlines, can the tasks be scheduled to meet their deadlines on the platform?
- Standard recurring task model
  - Tasks  $\{T_i\}$
  - Periodic tasks with periods  $\{P_i\}$
  - Execution times of the tasks  $\{e_i\}$
  - Known deadlines

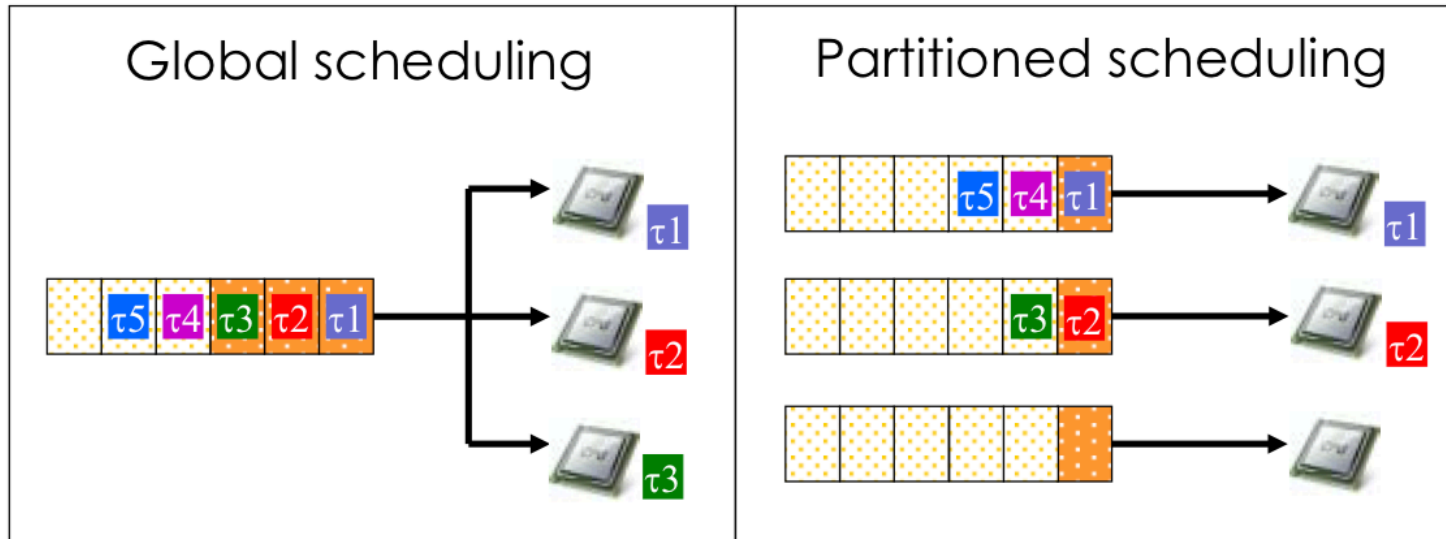
# Classification of MP scheduling approaches

---

Multiprocessor scheduling algorithms can be classified according to two orthogonal criteria:



# Global vs. partitioned scheduling



Bin-packing  
problem

+

Uniprocessor  
scheduling  
problem

NP-hard in the  
strong sense;  
various heuristics  
adopted

Well-known



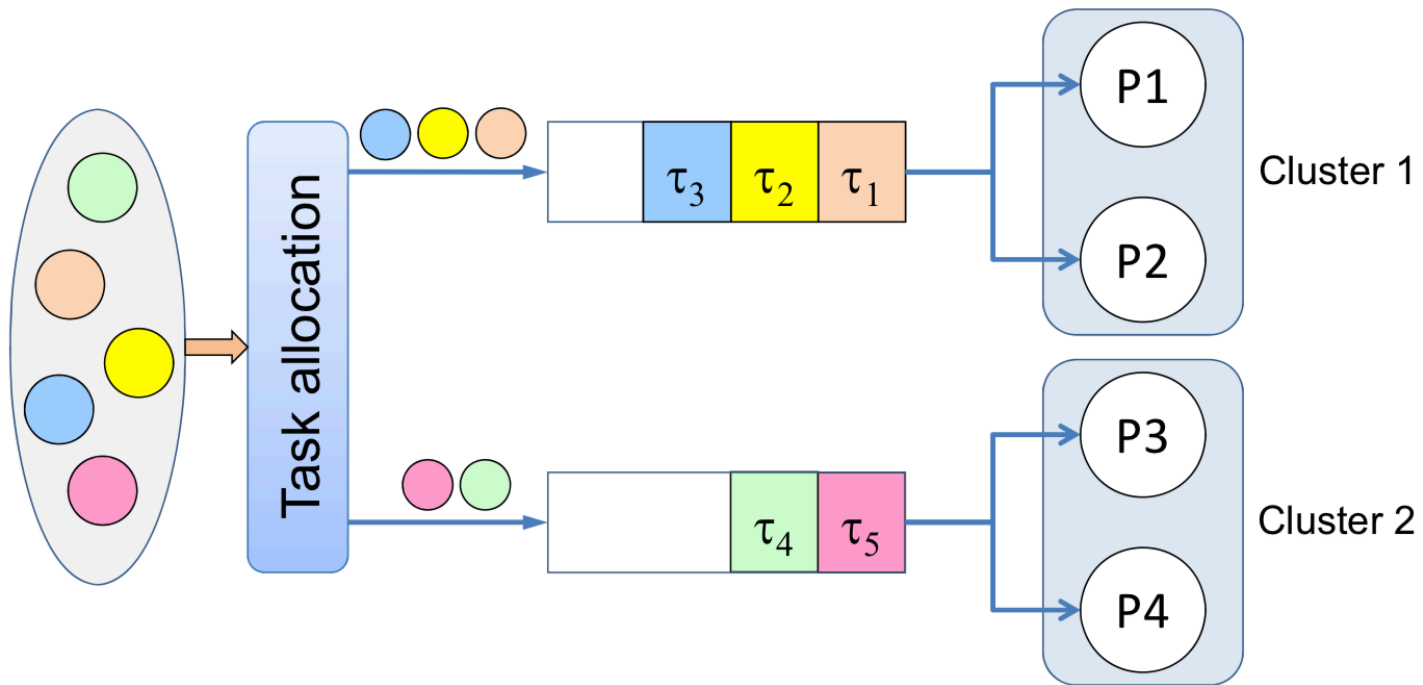
# Partitioned scheduling vs. global scheduling

---

- Partitioned scheduling is easier to implement and reason about
  - Once tasks are assigned to a processor, we can apply known schedulability tests
  - Without migration it is easier to maintain context information
  - When processors are on different chips migration requires context transfer, cache problems, etc.
- Global scheduling, however, is more flexible
  - Allowing migration improves schedulability
  - On-chip multiprocessing minimizes some of the overhead of job migration

# Clustered scheduling

- A task can only migrate within a predefined subset of processors (cluster)



# Partitioned Scheduling

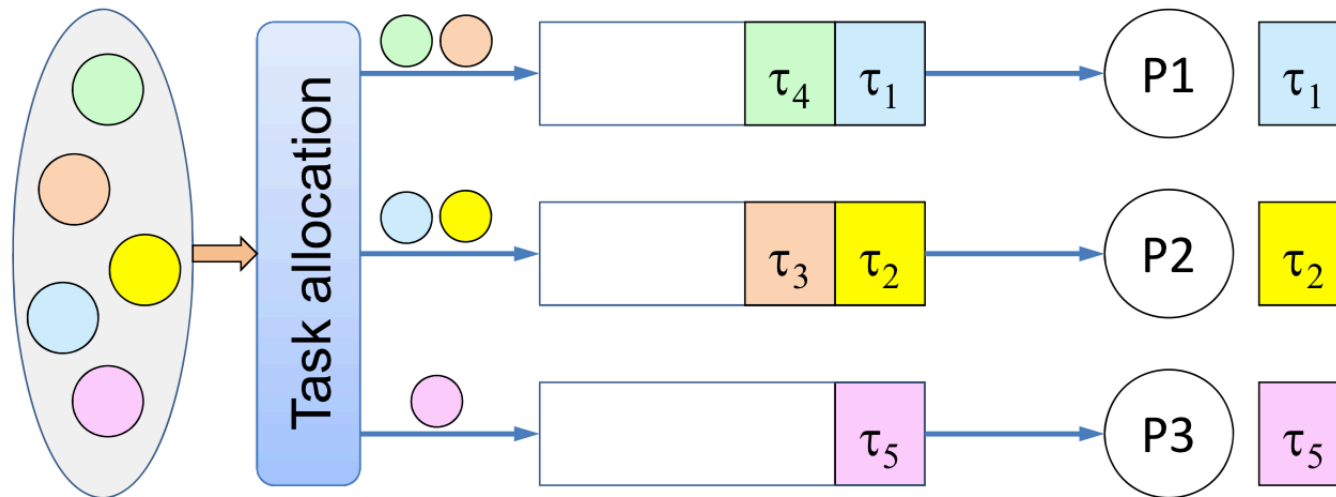
---

- Partitioned scheduling problem has two separate dimensions:
  - **Spatial dimension** (task to processor allocation)
    - Which processor should a task execute on?
  - **Temporal Dimension** (local scheduling policy (per processor))
    - Once tasks are pinned to processors, how do we schedule the tasks on every processor?

# Partitioned scheduling

---

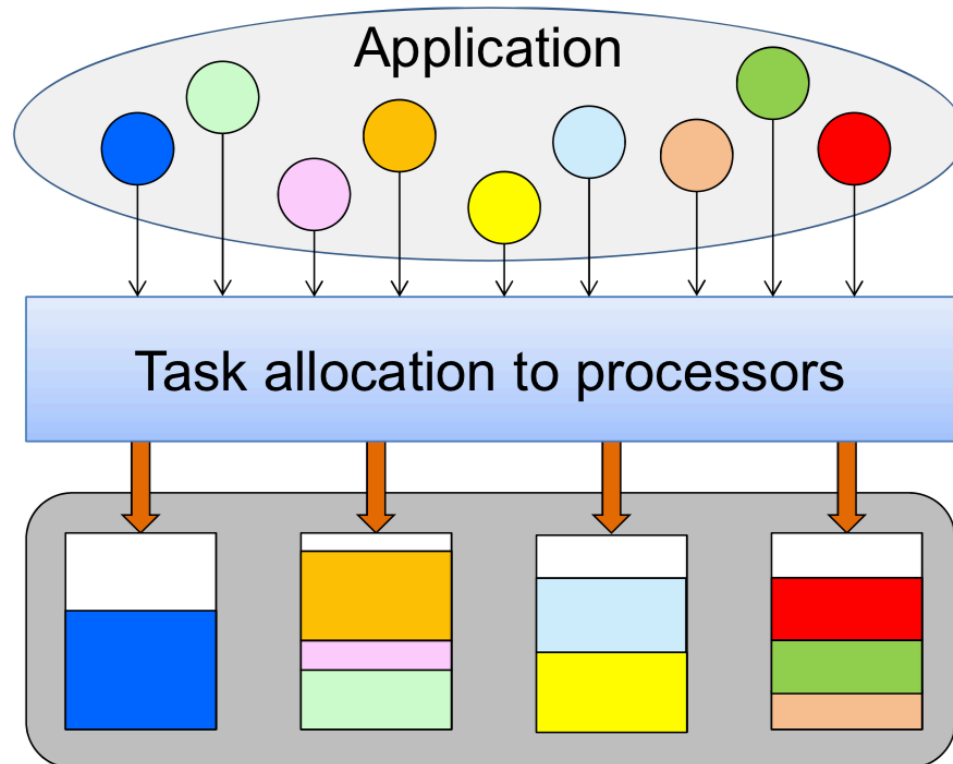
- Each processor manages its own ready queue
- The processor for each task is determined off-line
- The processor cannot be changed at run time



# Partitioned scheduling

---

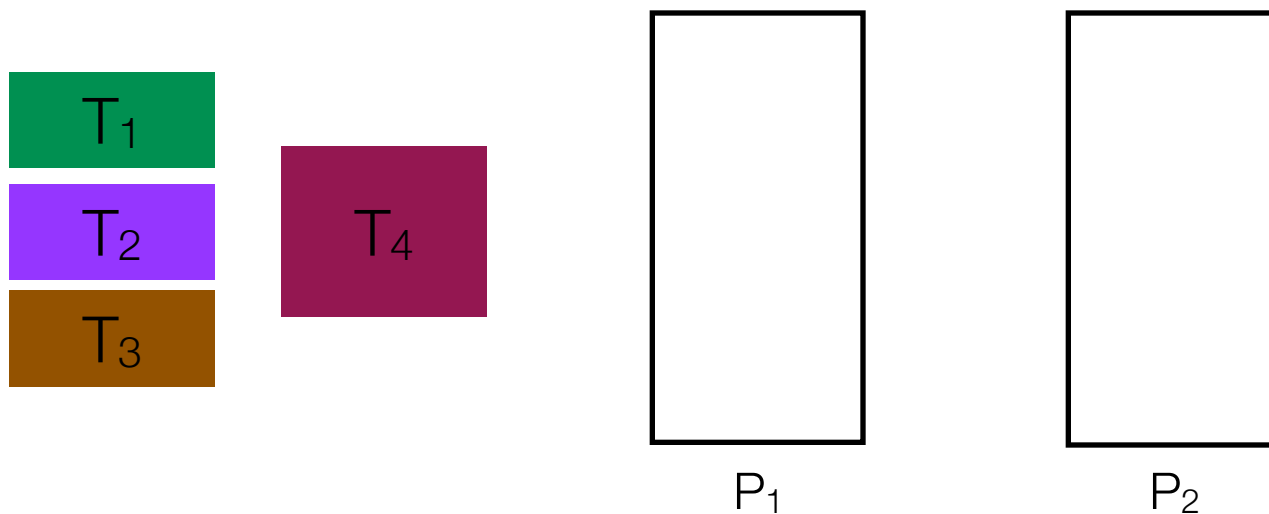
Once tasks are allocated to processors, they can be handled by [uniprocessor scheduling](#) algorithms:



# Partitioned scheduling

---

- We can use either fixed priority (rate monotonic) or dynamic priority (EDF) policies
- Need to assign tasks to processors such that the utilization bound (or other schedulability condition) is satisfied
  - For simplicity we will assume that any task can be allocated to any processor
  - This may not always be the case because of resource requirements and so on



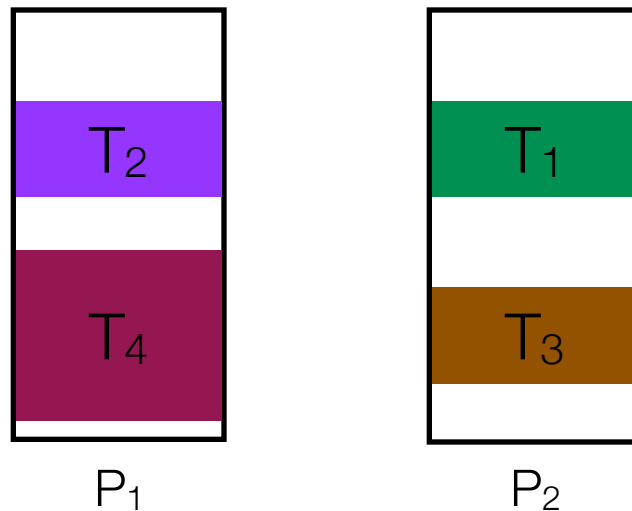
# Partitioned scheduling

---

- We can use either fixed priority (rate monotonic) or dynamic priority (EDF) policies
- Need to assign tasks to processors such that the utilization bound (or other schedulability condition) is satisfied
  - For simplicity we will assume that any task can be allocated to any processor
  - This may not always be the case because of resource requirements and so on

Any assignment of tasks to processors is suitable as long as utilization bounds are not violated.

Is closely related to the **bin packing** problem, which is NP-Hard.



# Spatial dimension (task to processor allocation)

---

- This is the task partitioning problem
- Consider the **identical** multiprocessor case (homogeneous)

Given  $n$  tasks with utilizations  $u_1, \dots, u_n$  and a set of  $m$  processors, each running a local scheduling algorithm with utilization bound (capacity, volume)  $U_b$ , is there an assignment of tasks to processors so that the utilization bound of each processor is not violated?

*If such assignment exists, how do we compute it?*



# Spatial dimension (task to processor allocation)

---

- This is the task partitioning problem
- Consider the **identical** multiprocessor case (homogeneous)

Given  $n$  tasks with utilizations  $u_1, \dots, u_n$  and a set of  $m$  processors, each running a local scheduling algorithm with utilization bound (capacity, volume)  $U_b$ , is there an assignment of tasks to processors so that the utilization bound of each processor is not violated?

*If such assignment exists, how do we compute it?*

- Analogy with the **BIN-PACKING** problem
  - **Optimization version:** Given  $n$  items with sizes  $u_1, \dots, u_n$ , what is the minimum number of bins, each having capacity  $C$ , that are needed to pack all items so that each bin's capacity is not exceeded?

# Bin Packing – Practical Examples

---

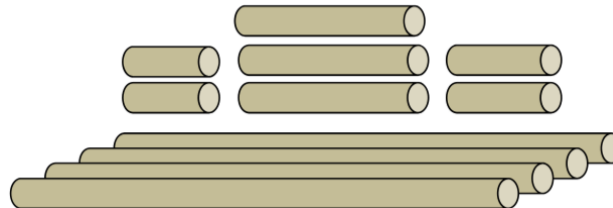
- How to store files into CDs



- How to fill minibuses with groups of people that must stay together



- How to cut pieces of pipes from pipes of given length to minimize wastes.



# Spatial dimension (task to processor allocation)

---

- This is the task partitioning problem
- Consider the **identical** multiprocessor case (homogeneous)

Given  $n$  tasks with utilizations  $u_1, \dots, u_n$  and a set of  $m$  processors, each running a local scheduling algorithm with utilization bound (capacity, volume)  $U_b$ , is there an assignment of tasks to processors so that the utilization bound of each processor is not violated?

*If such assignment exists, how do we compute it?*

- Analogy with the **BIN-PACKING** problem
  - **Optimization version:** Given  $n$  items with sizes  $u_1, \dots, u_n$ , what is the minimum number of bins, each having capacity  $C$ , are needed to pack all items so that each bin's capacity is not exceeded?
  - **Decision version:** Given  $n$  items with sizes  $u_1, \dots, u_n$ , is there a packing of the items into at most  $m$  bins each of capacity  $C$  each so that the capacity of each bin is not exceeded?

# Spatial dimension (task to processor allocation)

---

- **BIN-PACKING** is NP-Complete in the strong sense
  - And so is the task partitioning problem
- Exact solution methods are exponential-time (unless  $P = NP$ )
  - Integer Linear Programming, Branch & Bound, etc...
- Our next best option: *Approximation Schemes*
  - Those are algorithms that produce approximate solutions but with **provable** guarantees on the quality of the solutions they return
  - How do we quantify approximation error (optimality gap)? What is a suitable metric? (next: either utilization bounds or processor speed-up factors)
- **Assumption:** Local scheduling policy is EDF so that  $U_b = 1$  for all processors

# Bin Packing Heuristics

---

- **Next Fit (NF)**  
Place each item in the same bin as the last item. If it does not fit, start a new bin.
- **First Fit (FF)**  
Place each item in the first bin that can contain it.
- **Best Fit (BF)**  
Places each item in the bin with the smallest empty space.
- **Worst Fit (WF)**  
Places each item in the used bin with the largest empty space, otherwise starts a new bin.

# Bin Packing Heuristics

---

The performance of each algorithm strongly depends on the input sequence

However,

**NF** has a poor performance since it does not exploit the empty space in the previous bins

**FF** improves the performance by exploiting the empty space available in all the used bins.

**BF** tends to fill the used bins as much as possible.

**WF** tends to balance the load among the used bins.

# The First Fit Decreasing (FFD) heuristic

---

- Most used heuristic: First Fit Decreasing (FFD)
  - Maintain a list of “opened” processors so far (initially empty)

# The First Fit Decreasing (FFD) heuristic

---

- Most used heuristic: First Fit Decreasing (FFD)
  - Maintain a list of “opened” processors so far (initially empty)
  - Sort the jobs in decreasing order of utilizations  $u_1 > u_2 > \dots > u_n$



# The First Fit Decreasing (FFD) heuristic

---

- Most used heuristic: First Fit Decreasing (FFD)
  - Maintain a list of “opened” processors so far (initially empty)
  - Sort the jobs in decreasing order of utilizations  $u_1 > u_2 > \dots > u_n$
  - In this order, pack  $u_i$  into the the oldest (earliest opened) processor into which it fits

# The First Fit Decreasing (FFD) heuristic

---

- Most used heuristic: First Fit Decreasing (FFD)
  - Maintain a list of “opened” processors so far (initially empty)
  - Sort the jobs in decreasing order of utilizations  $u_1 > u_2 > \dots > u_n$
  - In this order, pack  $u_i$  into the the oldest (earliest opened) processor into which it fits
  - If none of currently open processors have enough remaining capacity (utilization) to accommodate  $u_i$ , open a new processor if we have not exhausted all  $m$  processor and put  $u_i$  in it

# The First Fit Decreasing (FFD) heuristic

---

- Most used heuristic: First Fit Decreasing (FFD)
  - Maintain a list of “opened” processors so far (initially empty)
  - Sort the jobs in decreasing order of utilizations  $u_1 > u_2 > \dots > u_n$
  - In this order, pack  $u_i$  into the the oldest (earliest opened) processor into which it fits
  - If none of currently open processors have enough remaining capacity (utilization) to accommodate  $u_i$ , open a new processor if we have not exhausted all  $m$  processor and put  $u_i$  in it
  - If we used all  $m$  processors, declare the task set UNSCHEDULABLE

# Utilization bounds for partitioned scheduling with EDF

---

- When tasks are allocated to processors using the FFD heuristic, we can derive a utilization upper-bound for a uniform multiprocessor system that guarantees schedulability
- Let  $m$  be the number of processors: then the maximum possible utilization of the system is  $m$  (each processor can have a utilization up to 1)

$$U_b = \frac{m + 1}{2}$$

- Proof sketch
  - Consider a task set with  $m+1$  tasks, each task having utilization  $1/2$ ; this task set is schedulable
  - If each task has utilization slightly greater than  $1/2$ , the task set is not schedulable

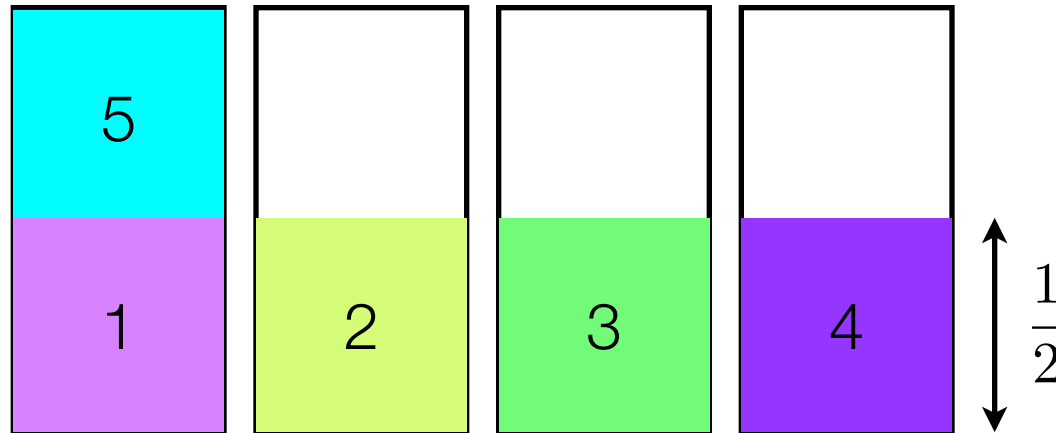
# Utilization bounds for partitioned scheduling with EDF

---

- When tasks are allocated to processors using the FFD heuristic, we can derive a utilization upper-bound for a uniform multiprocessor system that guarantees schedulability
- This is a sufficient condition but not necessary

$$U_b = \frac{m + 1}{2}$$

Schedulable



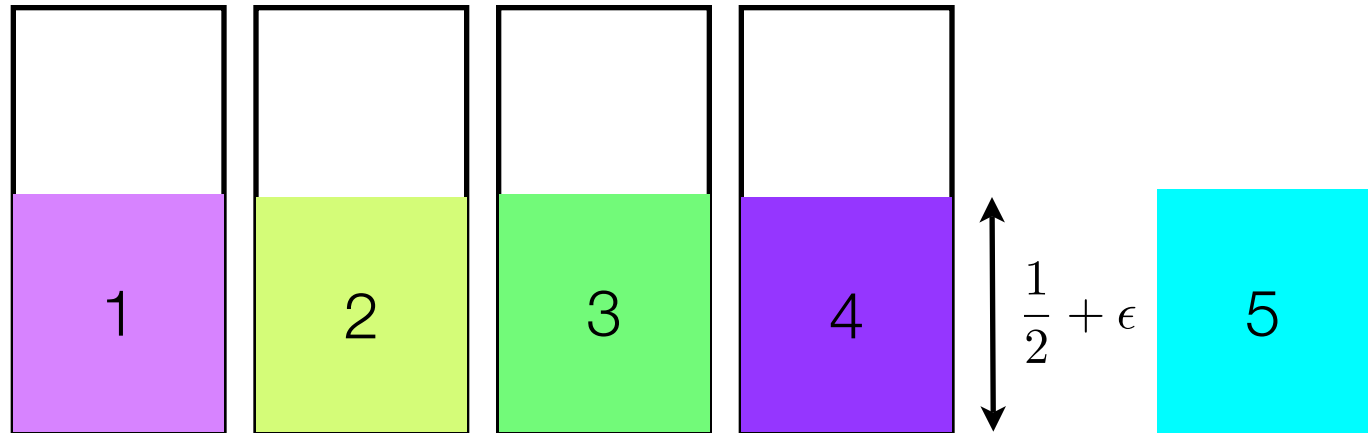
# Utilization bounds for partitioned scheduling with EDF

---

- When tasks are allocated to processors using the FFD heuristic, we can derive a utilization upper-bound for a uniform multiprocessor system that guarantees schedulability
- This is a sufficient condition but not necessary

$$U_b = \frac{m + 1}{2}$$

Not  
schedulable



# Utilization bounds for partitioned scheduling with EDF

---

- When tasks are allocated to processors using the FFD heuristic, we can derive a utilization upper-bound for a uniform multiprocessor system that guarantees schedulability
- This is a sufficient condition but not necessary

$$U_b = \frac{m + 1}{2}$$

- The utilization can be rather poor even though we have many processors
- The overall utilization is as low as 50%!

# Is the utilization bound a good metric?

- T consists of  $m+1$  tasks, each of utilization  $0.5+\epsilon$
- $U = (m+1)(0.5+\epsilon)$
- $U$  is larger than the  $(m+1)/2$  utilization bound
- According to utilization bounds, it is deemed unschedulable
- For **no**  $\epsilon > 0$  is T is schedulable on  $m$  processors by any algorithm, not even by optimal
- For T one pays the penalty of a utilization loss of  $(m - (m + 1)/2)$  as a consequence of choosing to do partitioned scheduling, regardless of which particular partitioning algorithm we use.
- T' consists of  $2(m+1)$  tasks, each of utilization  $(0.5+\epsilon)/2$
- $U' = (m+1)(0.5+\epsilon)$
- $U'$  larger than the  $(m+1)/2$  utilization bound
- According to utilization bounds, it is deemed unschedulable
- There **is**  $\epsilon > 0$  for which T' is schedulable on  $m$  processors! Can you come up with such  $\epsilon$ ?
- Any utilization loss arises from *the choice of partitioning algorithm*, not merely the decision to go with partitioned (as opposed to global) scheduling.



# Is the utilization bound a good metric?

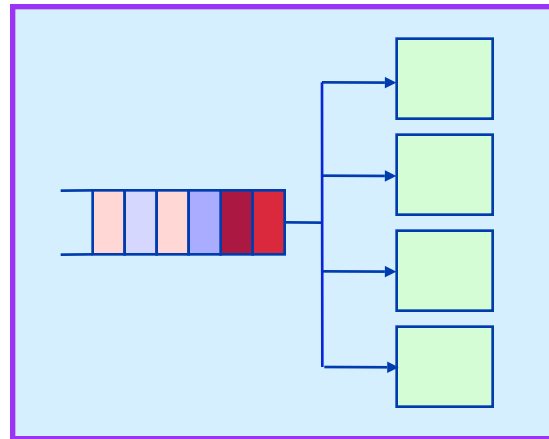
- T consists of consists of  $m+1$  tasks, each of utilization  $0.5+\epsilon$
- $U = (m+1)(0.5+\epsilon)$
- $U$  is larger than the  $(m+1)/2$  utilization bound
- According to utilization bounds, it is deemed unschedulable
- For **no**  $\epsilon > 0$  is T is schedulable on  $m$  processors by any algorithm, not even by optimal
- For T one pays the penalty of a utilization loss of  $(m - (m + 1)/2)$  as a consequence of choosing to do partitioned scheduling, regardless of which particular partitioning algorithm we use.
- T' consists of  $2(m+1)$  tasks, each of utilization  $(0.5+\epsilon)/2$
- $U' = (m+1)(0.5+\epsilon)$
- $U'$  larger than the  $(m+1)/2$  utilization bound
- According to utilization bounds, it is deemed unschedulable
- There **is**  $\epsilon > 0$  for which T' is schedulable on  $m$  processors! Can you come up with such  $\epsilon$ ?
- Any utilization loss arises from *the choice of partitioning algorithm*, not merely the decision to go with partitioned (as opposed to global) scheduling.

**Moral:** Utilization bounds do not distinguish between both cases!

# Global scheduling

---

- Is an alternative to partitioning
- Tasks may migrate among processors
- Appropriate for tightly-coupled systems



# Global vs. Partitioned: Pros & Cons

---

## □ **Global** scheduling

- ✓ Automatic load balancing
- ✓ Lower avg. response time
- ✓ Simpler implementation
- ✓ *Optimal* schedulers exist
- ✓ More efficient reclaiming
- ✗ Migration costs
- ✗ Inter-core synchronization
- ✗ Loss of cache affinity
- ✗ Weak scheduling framework

## □ **Partitioned** scheduling

- ✓ Supported by automotive industry (e.g., AUTOSAR)
- ✓ No migrations
- ✓ Isolation between cores
- ✓ Mature scheduling framework
- ✗ Cannot exploit unused capacity
- ✗ Rescheduling not convenient
- ✗ NP-hard allocation

# The Dhall Effect

---

- Consider an implicit-deadline sporadic task system of  $(m + 1)$  tasks to be scheduled upon an  $m$ -processor platform
  - Tasks  $T_1, \dots, T_m$  have parameters  $(e_i = 1, P_i = P)$
  - Task  $T_{m+1}$  has parameters  $e_{m+1} = P_{m+1} = P + 1$

# The Dhall Effect

---

- Consider an implicit-deadline sporadic task system of  $(m + 1)$  tasks to be scheduled upon an  $m$ -processor platform
  - Tasks  $T_1, \dots, T_m$  have parameters  $(e_i = 1, P_i = P)$
  - Task  $T_{m+1}$  has parameters  $e_{m+1} = P_{m+1} = P + 1$
- $U = m \left( \frac{1}{P} \right) + \frac{P+1}{P+1} = \frac{m}{P} + 1 \quad \left[ \lim_{P \uparrow \infty} U = 1 \right]$

# The Dhall Effect

---

- Consider an implicit-deadline sporadic task system of  $(m + 1)$  tasks to be scheduled upon an  $m$ -processor platform
  - Tasks  $T_1, \dots, T_m$  have parameters  $(e_i = 1, P_i = P)$
  - Task  $T_{m+1}$  has parameters  $e_{m+1} = P_{m+1} = P + 1$
- $U = m \left( \frac{1}{P} \right) + \frac{P+1}{P+1} = \frac{m}{P} + 1 \quad \left[ \lim_{P \uparrow \infty} U = 1 \right]$
- Is this task set schedulable with Global EDF if all tasks are released simultaneously?

# The Dhall Effect

---

- Consider an implicit-deadline sporadic task system of  $(m + 1)$  tasks to be scheduled upon an  $m$ -processor platform
  - Tasks  $T_1, \dots, T_m$  have parameters  $(e_i = 1, P_i = P)$
  - Task  $T_{m+1}$  has parameters  $e_{m+1} = P_{m+1} = P + 1$
- $U = m \left( \frac{1}{P} \right) + \frac{P+1}{P+1} = \frac{m}{P} + 1 \quad \left[ \lim_{P \uparrow \infty} U = 1 \right]$
- Is this task set schedulable with Global EDF if all tasks are released simultaneously?
- What happens if we increase the number of processors?

# The Dhall Effect

---

- Consider an implicit-deadline sporadic task system of  $(m + 1)$  tasks to be scheduled upon an  $m$ -processor platform
  - Tasks  $T_1, \dots, T_m$  have parameters  $(e_i = 1, P_i = P)$
  - Task  $T_{m+1}$  has parameters  $e_{m+1} = P_{m+1} = P + 1$
- $U = m \left( \frac{1}{P} \right) + \frac{P+1}{P+1} = \frac{m}{P} + 1 \quad \left[ \lim_{P \uparrow \infty} U = 1 \right]$
- Is this task set schedulable with Global EDF if all tasks are released simultaneously?
- What happens if we increase the number of processors?

This task system is not EDF-schedulable despite having a utilization close to 1

The utilization bound of global EDF is **very poor**: it is arbitrarily close to one regardless of the number of processors.



# The Dhall Effect

---

- Consider an implicit-deadline sporadic task system of  $(m + 1)$  tasks to be scheduled upon an  $m$ -processor platform
  - Tasks  $T_1, \dots, T_m$  have parameters  $(e_i = 1, P_i = P)$
  - Task  $T_{m+1}$  has parameters  $e_{m+1} = P_{m+1} = P + 1$
- $U = m \left( \frac{1}{P} \right) + \frac{P+1}{P+1} = \frac{m}{P} + 1 \quad \left[ \lim_{P \uparrow \infty} U = 1 \right]$
- Is this task set schedulable with Global EDF if all tasks are released simultaneously?
- What happens if we increase the number of processors?

This task system is not EDF-schedulable despite having a utilization close to 1

The utilization bound of global EDF is **very poor**: it is arbitrarily close to one regardless of the number of processors.

- **Question:** Is this task set partitioned-schedulable?

# The Dhall Effect

---

- Consider an implicit-deadline sporadic task system of  $(m + 1)$  tasks to be scheduled upon an  $m$ -processor platform
  - Tasks  $T_1, \dots, T_m$  have parameters  $(e_i = 1, P_i = P)$
  - Task  $T_{m+1}$  has parameters  $e_{m+1} = P_{m+1} = P + 1$
- $U = m \left( \frac{1}{P} \right) + \frac{P+1}{P+1} = \frac{m}{P} + 1 \quad \left[ \lim_{P \uparrow \infty} U = 1 \right]$
- Is this task set schedulable with Global EDF if all tasks are released simultaneously?
- What happens if we increase the number of processors?

This task system is not EDF-schedulable despite having a utilization close to 1

The utilization bound of global EDF is **very poor**: it is arbitrarily close to one regardless of the number of processors.

- **Question:** Is this task set partitioned-schedulable?
  - **Answer:** Yes! for example, when  $m < P$ , we need only two processors!

# The Dhall Effect

---

- Dhall's Effect shows the limitation of global EDF and RM: both utilization bounds tend to 1, independently of the value of  $m$ .
- Researchers lost interest in global scheduling for ~25 years, since late 1990s.
- Such a limitation is related to EDF and RM, not to global scheduling in general

# Global Scheduling: Negative Results




---

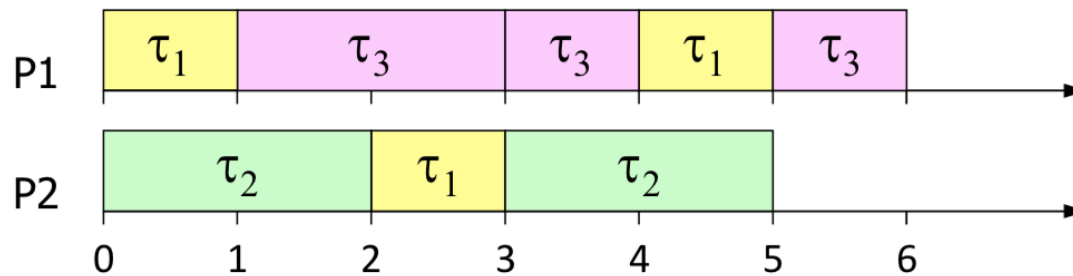
- **Weak theoretical framework**
  - Unknown critical instant
  - Global EDF is not optimal
  - Any global job-fixed (or task-dynamic) priority scheduler is not optimal
  - Optimality only for implicit deadlines
  - Many sufficient tests (most of them incomparable)

# Global vs. Partitioned

- There are tasks that are schedulable only with a **global** scheduler!

**Example:**

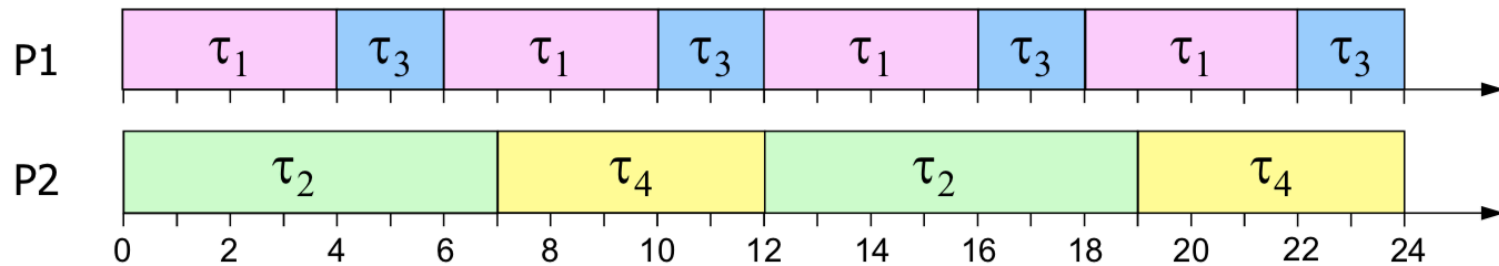
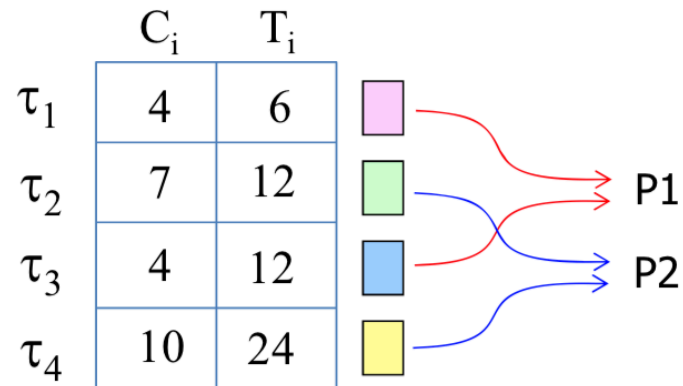
|          | $C_i$ | $T_i$ |                                                                                     |
|----------|-------|-------|-------------------------------------------------------------------------------------|
| $\tau_1$ | 1     | 2     |  |
| $\tau_2$ | 2     | 3     |  |
| $\tau_3$ | 2     | 3     |  |



# Global vs. Partitioned

- But there are also task sets that are schedulable only with a **partitioned** scheduler





**Example:**



All  $4! = 24$  global priority assignments lead to deadline miss.

# Global vs. Partitioned

- Example of an unfeasible global schedule with  $\pi_1 > \pi_2 > \pi_3 > \pi_4$

|          | $C_i$ | $T_i$ |                                                                                     |
|----------|-------|-------|-------------------------------------------------------------------------------------|
| $\tau_1$ | 4     | 6     |  |
| $\tau_2$ | 7     | 12    |  |
| $\tau_3$ | 4     | 12    |  |
| $\tau_4$ | 10    | 24    |  |

