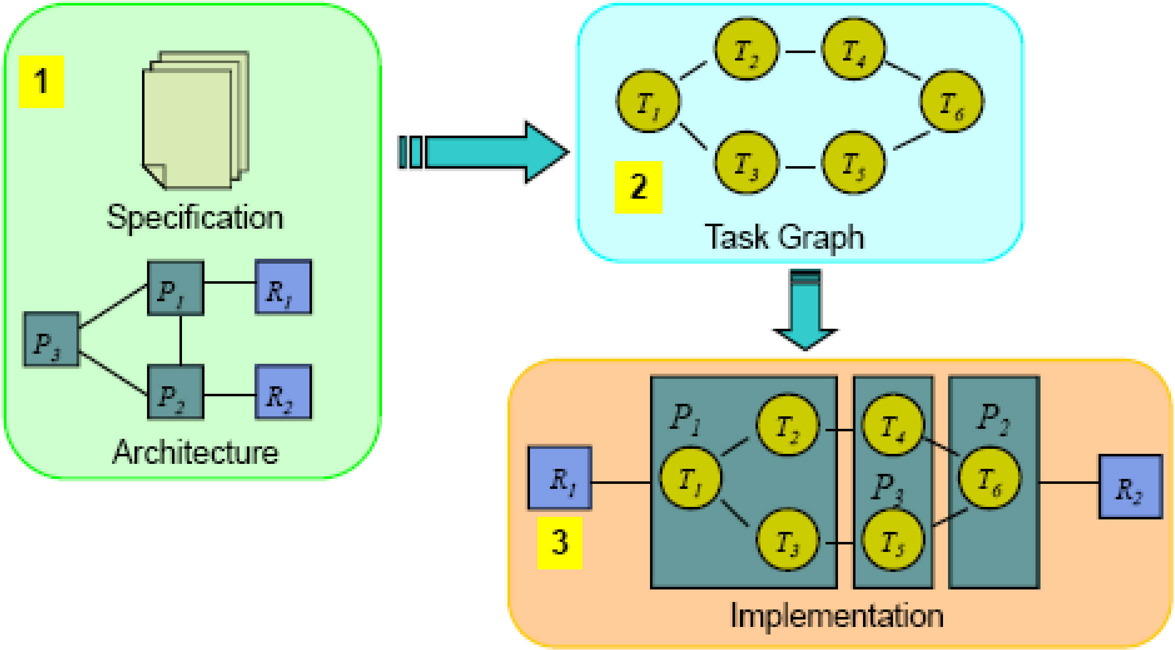
Simplified Design Flow



(a picture from Ingo Sander)

Hardware architecture

So far, we have talked about only single processor systems

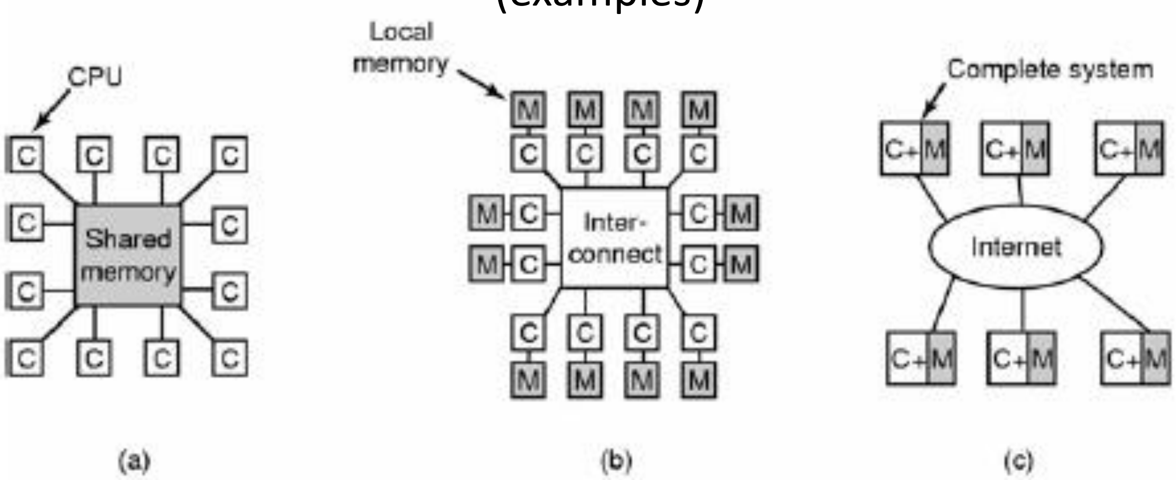
- **"Concurrency" implemented by "scheduling"**

RT systems often consist of several processors

- Multiprocessor systems
  - "Tightly connected" processors by a "high-speed" interconnect e.g. cross-bar, bus, NoC (network on chip) etc.
  - Single processor with multi-thread
- Distributed Systems
  - "Loosely connected" processors by a "low-speed" network e.g. CAN, Ethernet, Token Ring etc. --

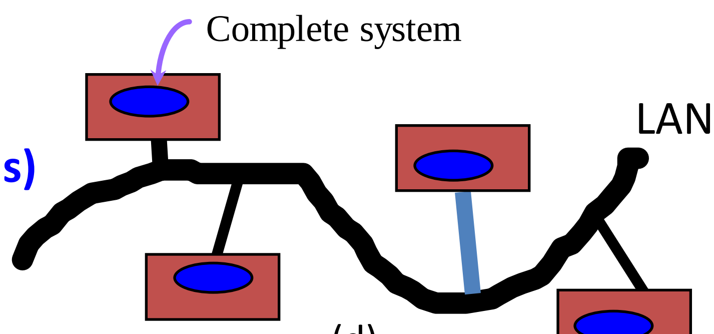
Multiprocessor vs.Distributed Systems

(examples)



– shared memory model
– message passing multiprocessor
– wide area distributed system

-- Local area distributed system (our focus)



(d)

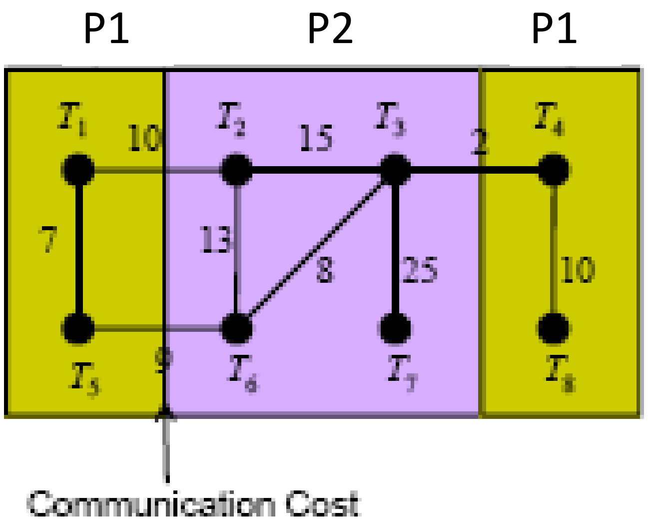
Task Assignment

- In the design flow:
  - First, the application is partitioned into tasks or task graphs.
  - At some stage, the execution times, communication costs, data and control dependencies of all the tasks become known.
- Task assignment determines
  - how many processors needed (bin-packing problem)
  - on which processor each task executes
- This is a very complex problem (NP-hard)
  - Often done off-line
  - Often heuristics work

Example of Task Assignment (or Task Partitioning)

i	T_i	u_i	i	T_i	u_i
1	(2,1)	0.50	5	(6,1)	0.17
2	(3,1)	0.33	6	(10,1)	0.10
3	(4,1)	0.25	7	(15,1)	0.07
4	(5,1)	0.20	8	(25,1)	0.04

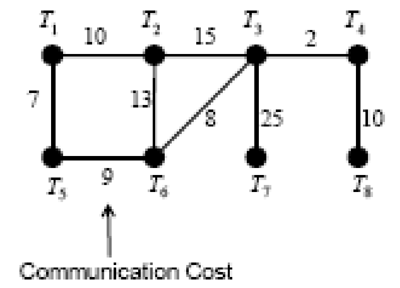
(EDF scheduling)



Objective is to find a partitioning, which is feasible at minimal costs (here interference cost is neglected!)

Task Assignment

- The task models used in task assignment can vary in complexity depending what considered/ignored:
  - Communication costs
  - Data and control dependencies
  - Resource requirements e.g. WCET, memory etc



- In multi-core platforms with shared caches, communication costs for tasks on the same chip may be very small
- It is often meaningful to consider the execution time requirement (WCET) and ignore communication in an early design phase

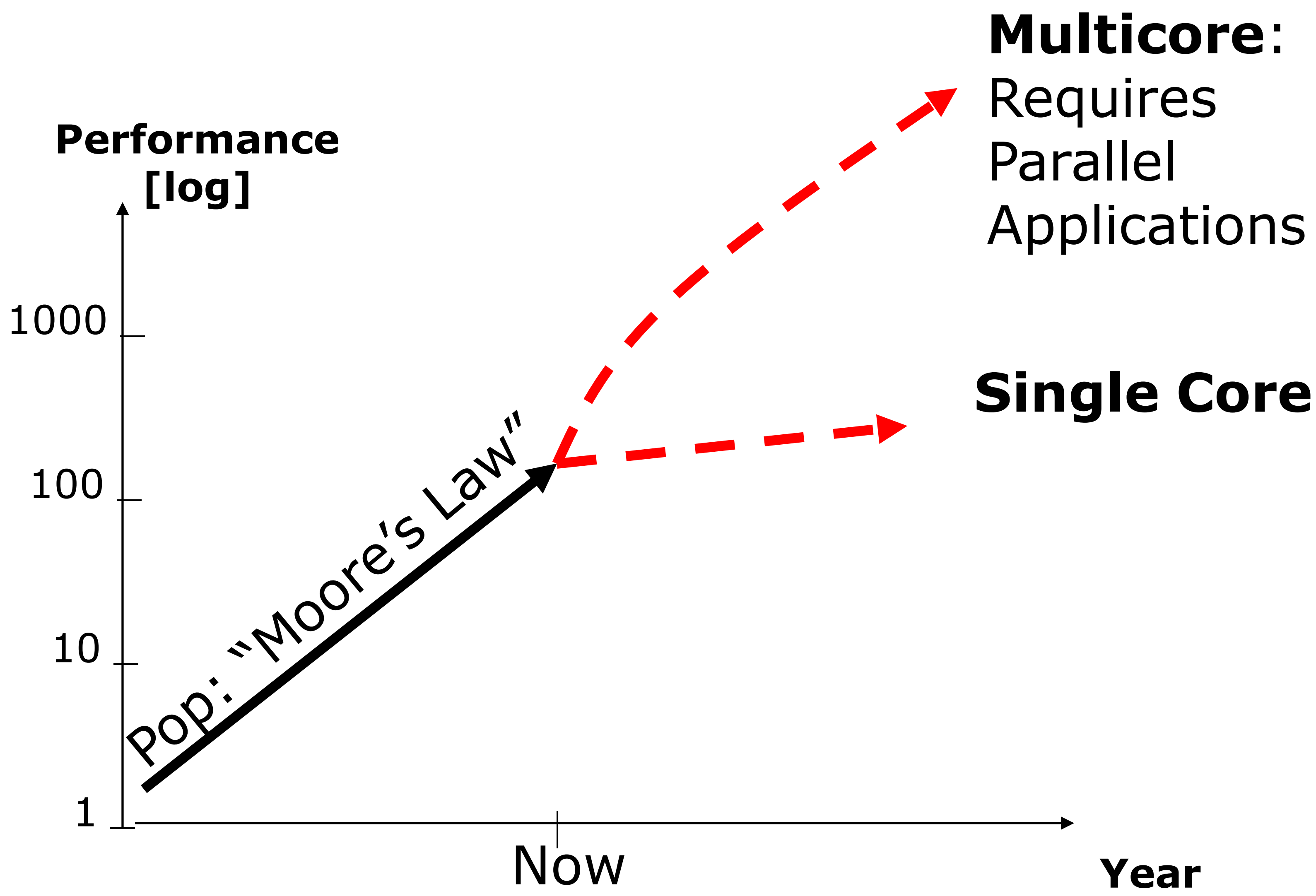
Today’s plan

- Why multiprocessor?
  - energy, performance and predictability
- What are multiprocessor systems
  - OS etc
- Design RT systems on multiprocessors
  - Task Assigment
- Multiprocessor scheduling
  - (semi-)partitioned scheduling
  - global scheduling

Why multiprocessor systems?

To get high performance and to reduce energy consumption

Hardware: Trends



Theoretically you may get:

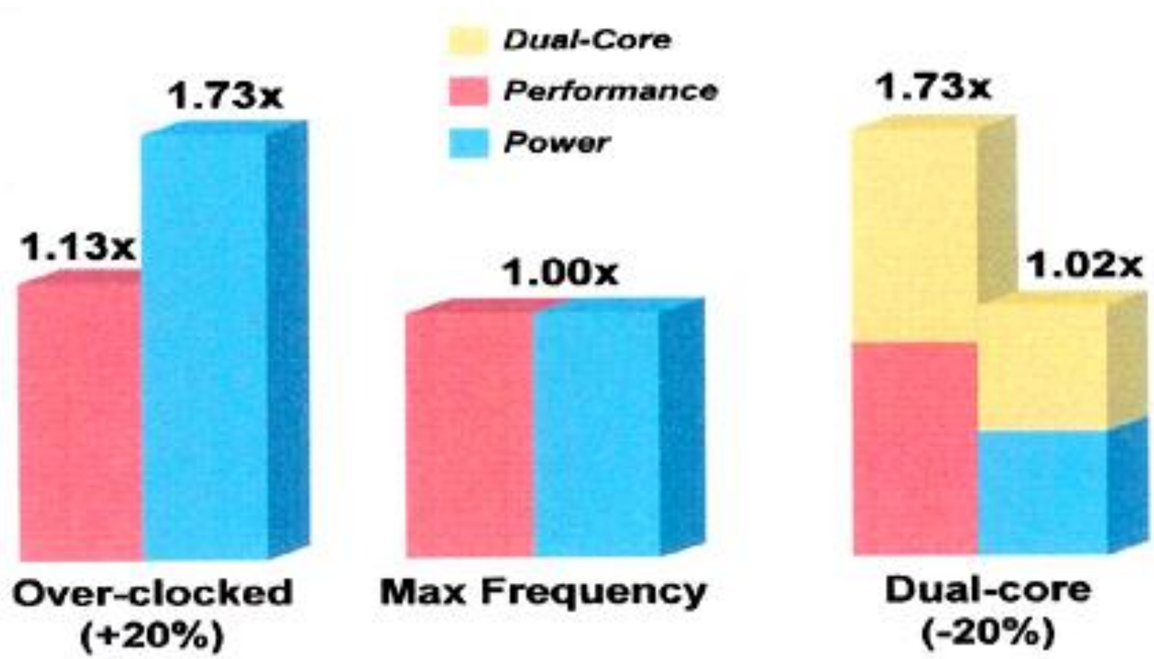
- Higher Performance
  - Increasing the cores -- unlimited computing power ∞ !
- Lower Power Consumption
  - Increasing the cores, decreasing the frequency
    - Performance (IPC) = Cores * F ➔ 2* Cores * F/2 ➔ Cores * F
    - Power = $C * V^2 * F$ ➔ $2 * C * (V/2)^2 * F/2$ ➔ $C * V^2 /4 * F$

➔ Keep the "same performance" using ¼ of the energy (by doubling the cores)

This sounds great for embedded & real-time applications!

CPU frequency vs Power consumption

1. Standard processor over-clocked 20%
2. Standard processor
3. Two standard processors each under-clocked 20%



What’s happening now?

- General-Purpose Computing

  (Symposium on High-Performance Chips, Hot Chips 21, Palo Alto, Aug 23-25, 2009)
  - 4 cores in notebooks
  - 12 cores in servers
    - AMD 12-core Magny-Cours will consume less energy than previous generations with 6 cores
  - 16 cores for IBM servers, Power 7
- Embedded Systems
  - 4 cores in ARM11 MPCore embedded processors

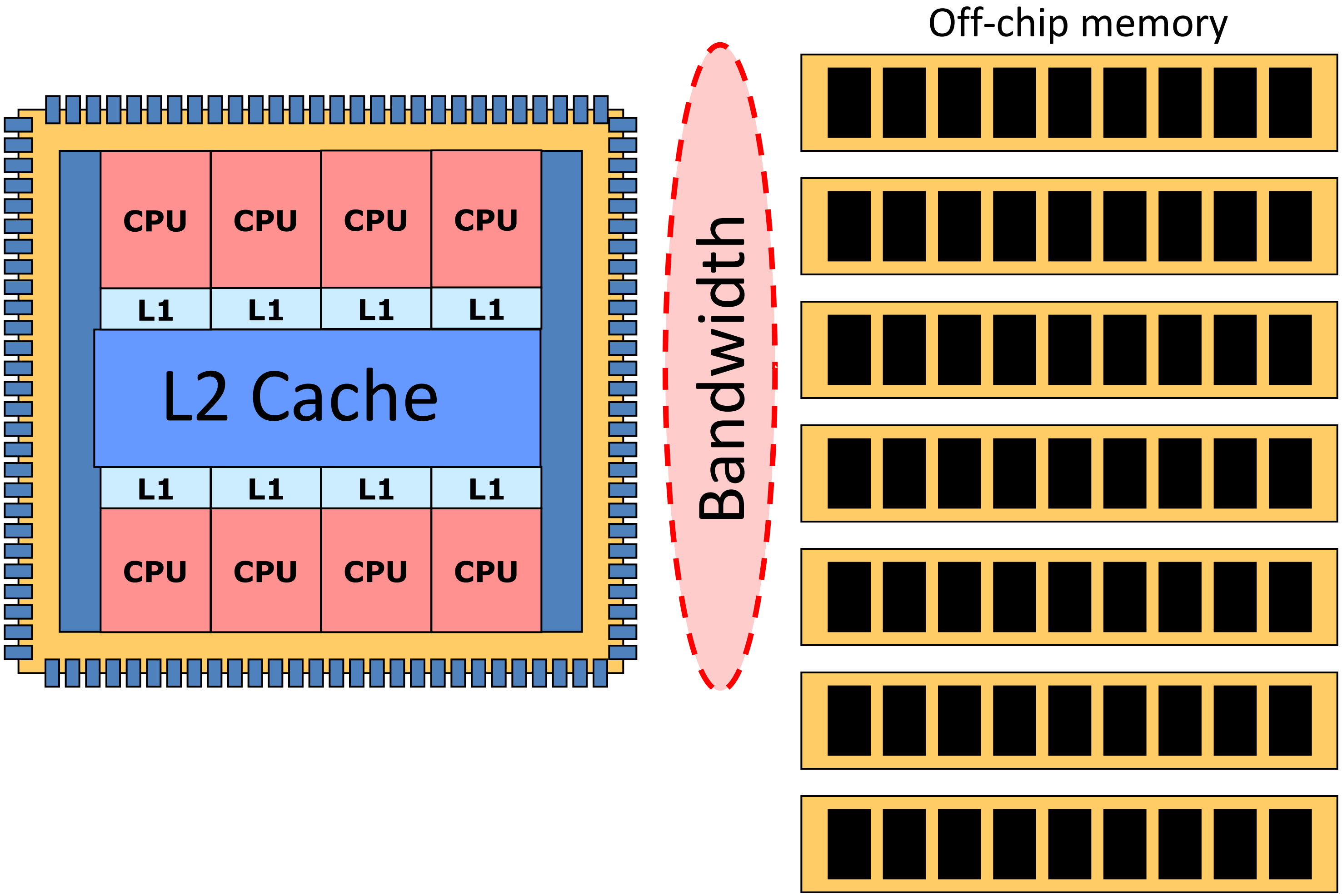
What next?

- Manycores (>100's of cores) predicted to be here in a few years – e.g. Ericsson

What are multiprocessor systems?

"Tightly connected" processors by a "high-speed" interconnect e.g. cross-bar, bus, NoC etc.

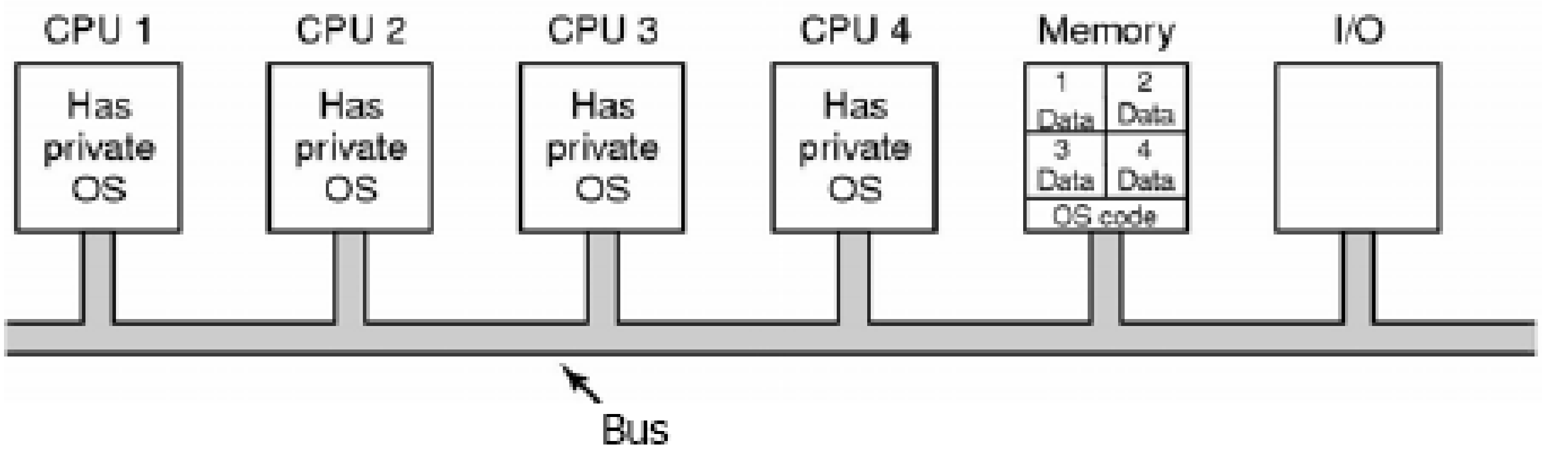
Typical Multicore Architecture



Single processor vs. multiprocessor OS

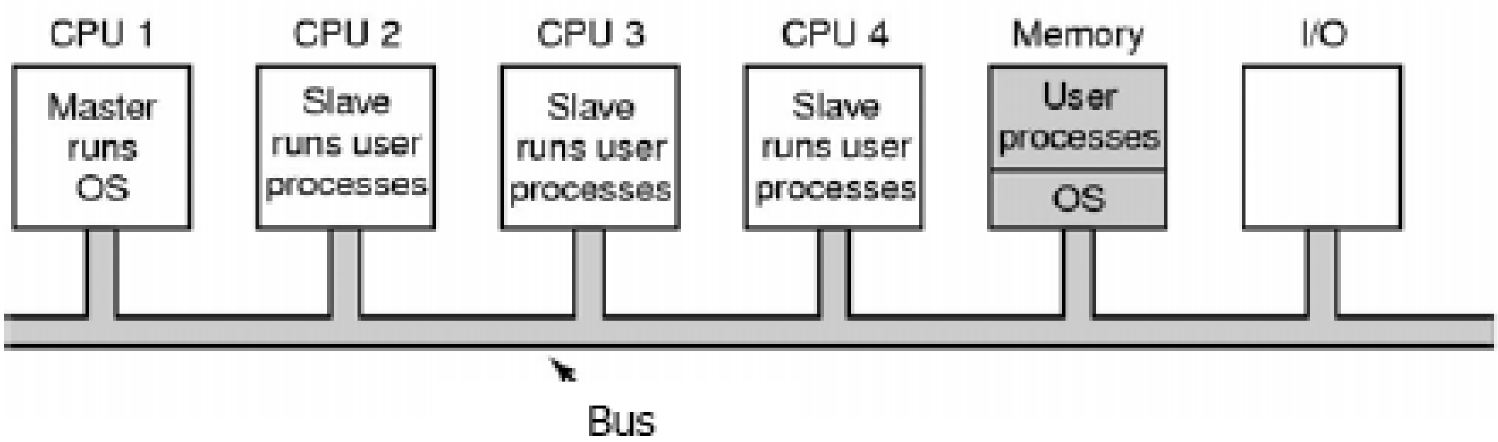
- Single-processor OS
  - easier to support kernel synchronization - why?
    - disabling interrupts to prevent concurrent executions
    - fine-grained locking vs. coarse-grained locking
  - easier to perform scheduling
    - which to run, not where to run
- Multi-processor OS
  - OS structure
  - synchronization
  - scheduling

Each node is a "complete system" running its own OS but sharing the memory



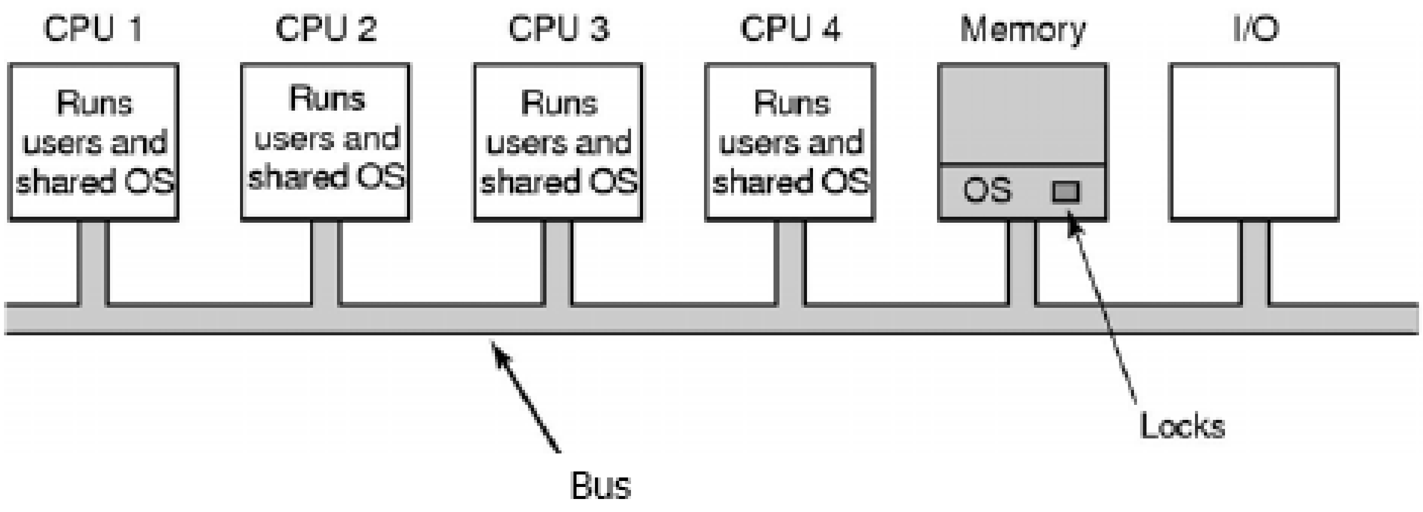
- Each CPU has its own operating system
  - quick to port from a single-processor OS
- Disadvantages
  - difficult to share things (processing cycles, memory, buffer cache)

One master node running the OS



- *All operating system functionality goes to one CPU*
  - no multiprocessor concurrency in the kernel
- Disadvantage
  - *OS CPU consumption may be large so the OS CPU becomes the bottleneck (especially in a machine with many CPUs)*

All nodes are “sharing” the OS kernel: Symmetric Multiprocessing (SMP)



- All CPUs run a single OS instance
- The OS itself must handle multiprocessor synchronization
  - have a big kernel lock – only one processor can execute in the kernel at a time
  - support fine-grain synchronization

Multiprocessor scheduling

- *"Given a set J of jobs where job j_i has length l_i and a number of processors m_i, what is the minimum possible time required to schedule all jobs in J on m processors such that none overlap?"* – Wikipedia
  - That is, design a schedule such that the response time of the last tasks is minimized

  (Alternatively, given M processors and N tasks, find a mapping from tasks to processors such that all the tasks are schedulable)

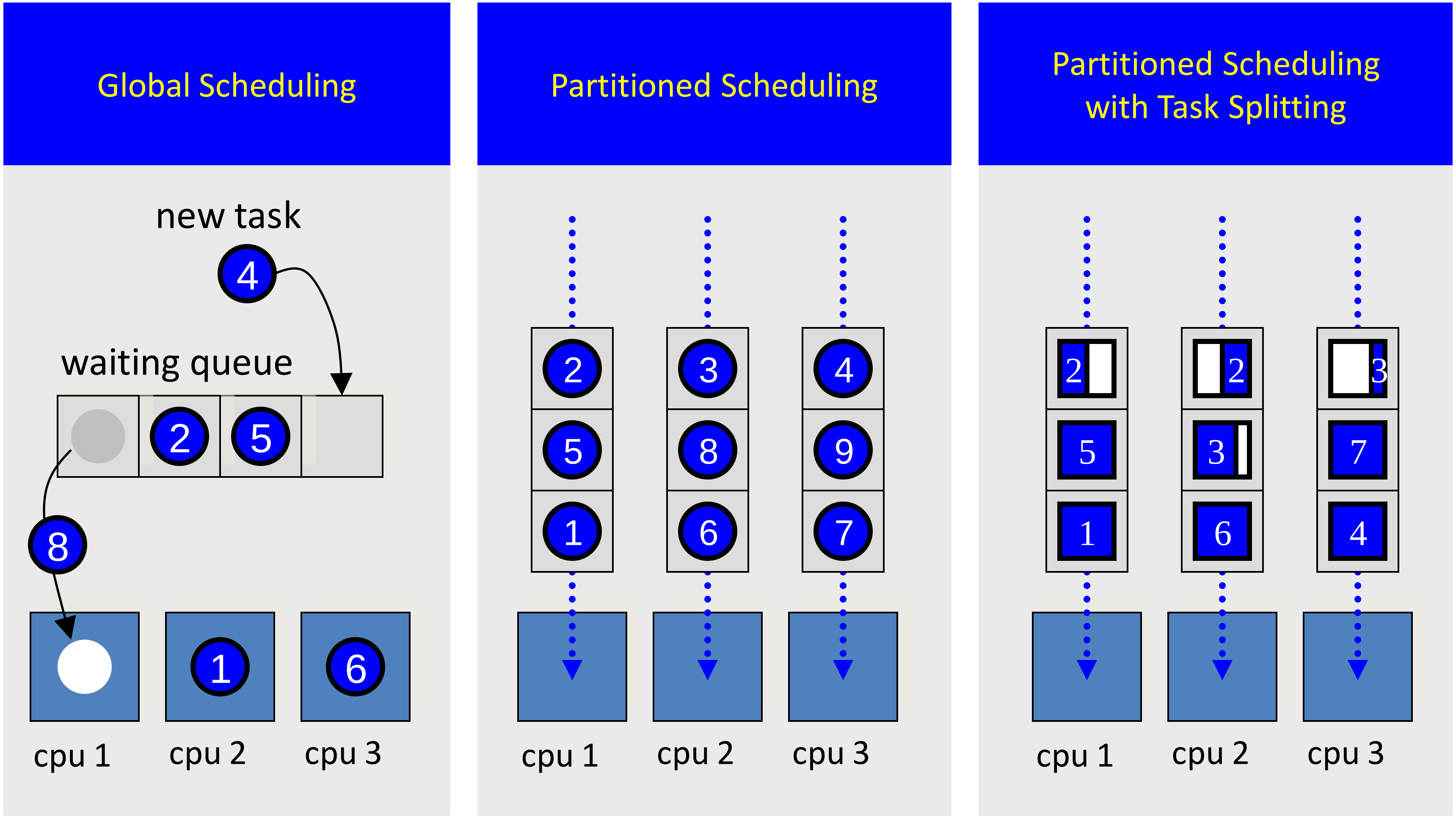
- The problem is NP-complete
- It is also known as the "load balancing problem"

Multiprocessor scheduling

– static and dynamic task assignment

- Partitioned scheduling
  - Static task assignment
    - Each task may only execute on a fixed processor
    - No task migration
- Semi-partitioned scheduling
  - Static task assignment
    - Each instance (or part of it) of a task is assigned to a fixed processor
    - task instance or part of it may migrate
- Global scheduling
  - Dynamic task assignment
    - Any instance of any task may execute on any processor
    - Task migration

Multiprocessor Scheduling



Multiprocessor (multicore) Scheduling

- Significantly more difficult:
  - Timing anomalies
  - Hard to identify the worst-case scenario
  - Bin-packing/NP-hard problems
  - Multiple resources e.g. caches, bandwidth
  - … …

Underlying causes

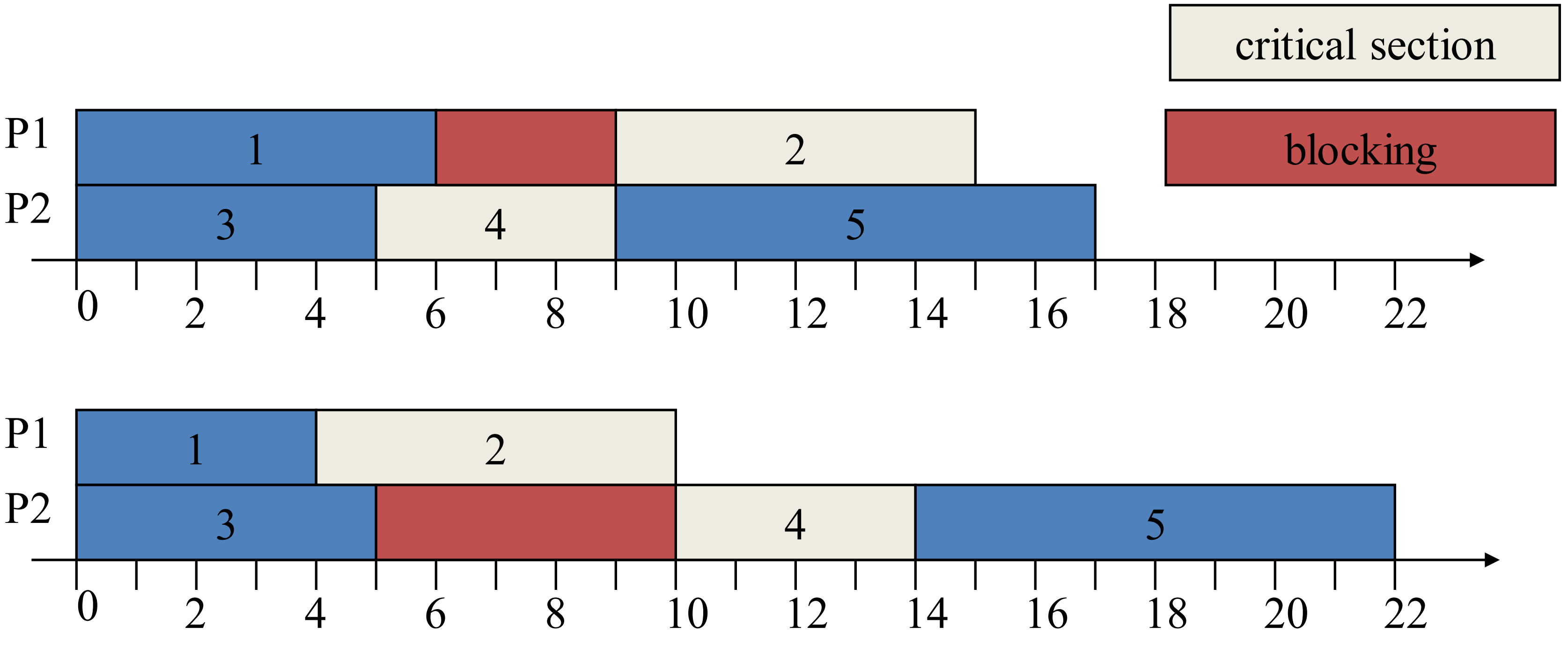
– The ”root of all evil” in global scheduling: (Liu, 1969):

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.

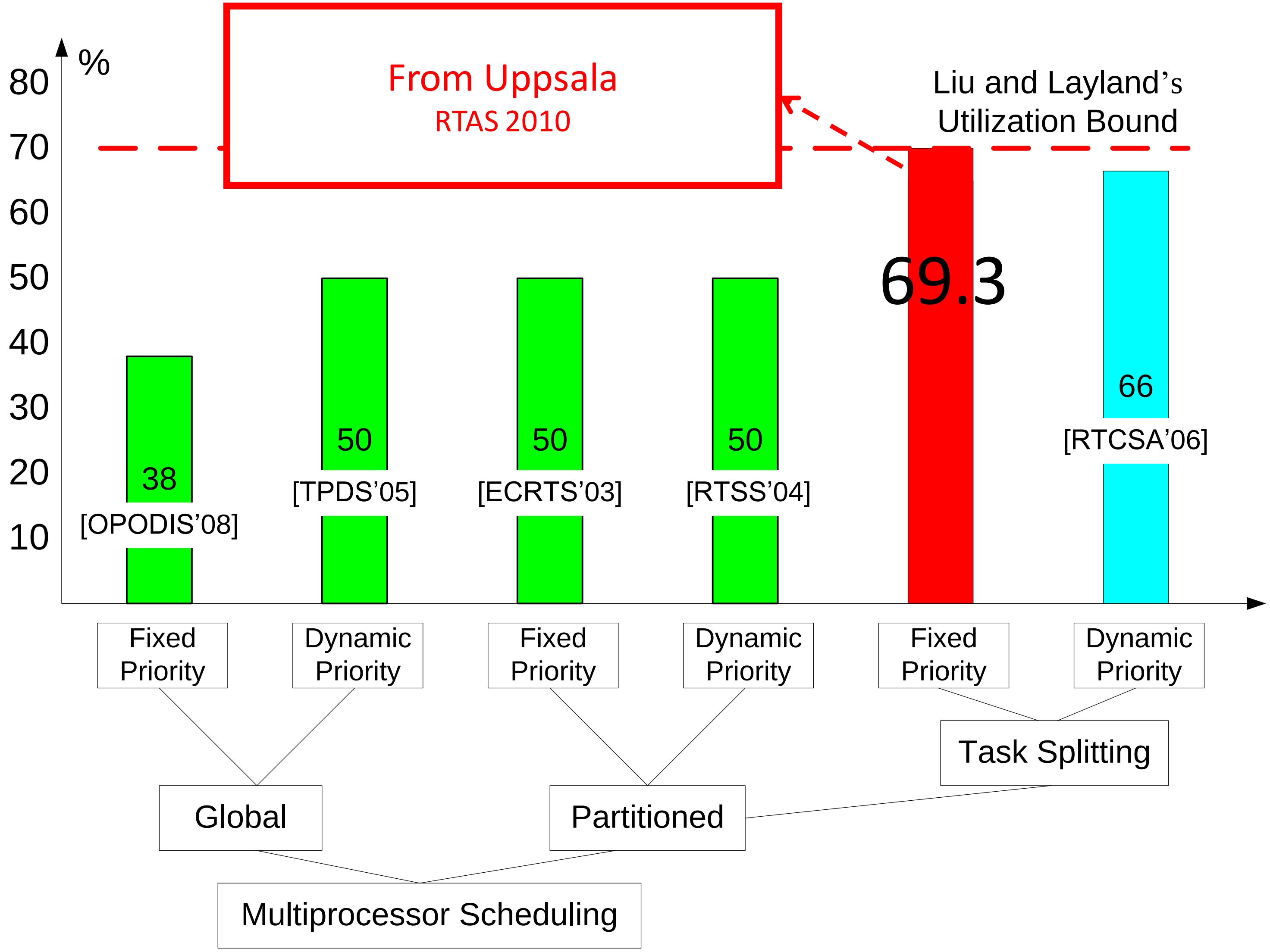
– Dhall’s effect: with RM, DM and EDF, some low-utilization task sets can be un-schedulable regardless of how many processors are used.
– Hard-to-find critical instant: a critical instant does not always occur when a task arrives at the same time as all its higher-priority tasks.

Example: Anomali under Resource constraints

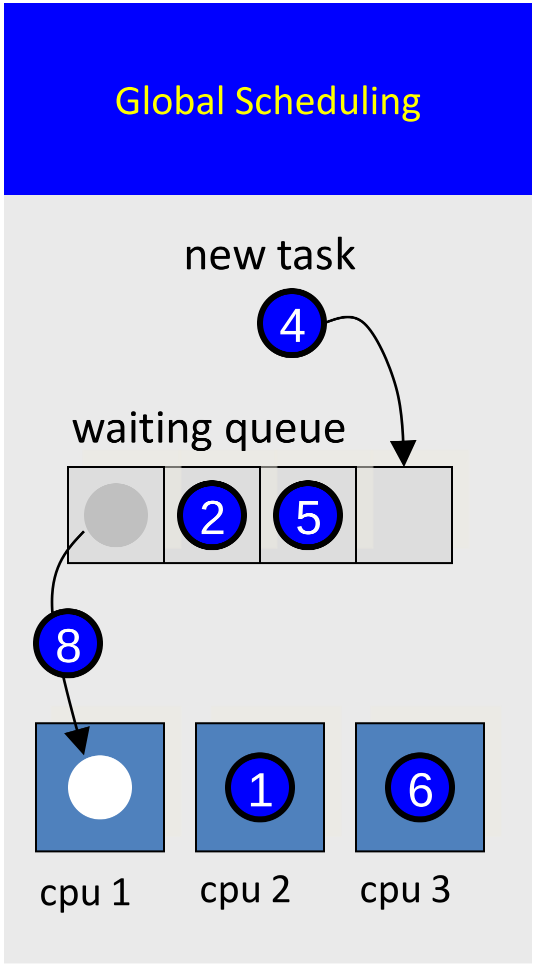
- 5 tasks on 2 CPUs, sharing 1 resource
- Static assignment T1, T2 on P1 and T3, T4, T5 on P2
- Reducing the computation time of T1 will increase the response time!



Best Known Results



Global Scheduling



Global scheduling

- All ready tasks are kept in a global queue
- When selected for execution, a task can be dispatched to any processor, even after being preempted

Global scheduling Algorithms

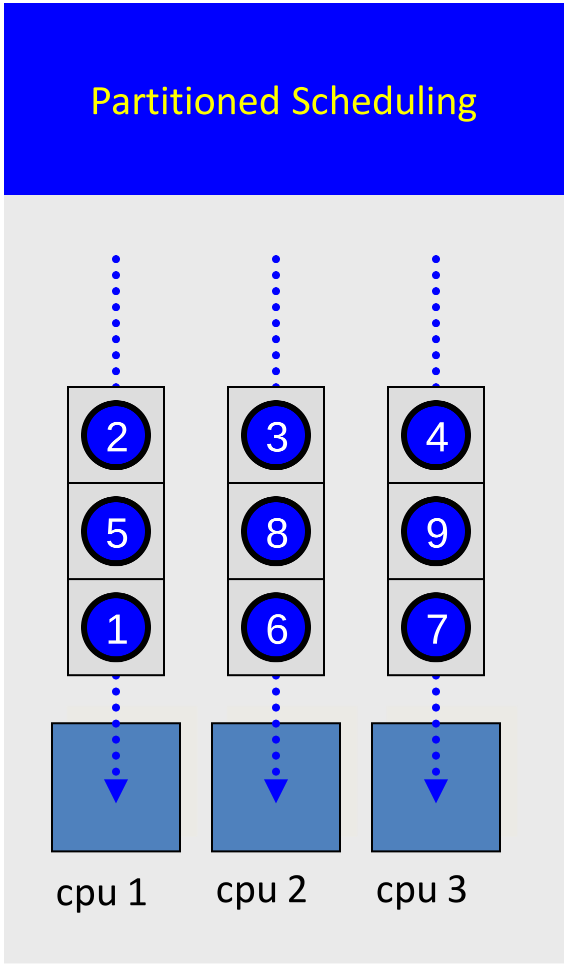
- EDF – Unfortunately not optimal!
  - No simple schedulability test known (only sufficient)
- Fixed Priority Scheduling e.g. RM
  - Difficult to find the optimal priority order
  - Difficult to check the schedulability

- Any algorithm for single processor scheduling may work, but schedulability analysis is non-trivial.

Global Scheduling: + and -

- Advantages:
  - Supported by most multiprocessor operating systems
    - Windows NT, Solaris, Linux, ...
  - Effective utilization of processing resources (if it works)
    - Unused processor time can easily be reclaimed at run-time (mixture of hard and soft RT tasks to optimize resource utilization)
- Disadvantages:
  - Few results from single-processor scheduling can be used
  - No “optimal” algorithms known except idealized assumption (Pfair sch)
  - Poor resource utilization for hard timing constraints
    - No more than 50% resource utilization can be guaranteed for hard RT tasks
  - Suffers from scheduling anomalies
    - Adding processors and reducing computation times and other parameters can actually decrease optimal performance in some scenarios!

Partition-Based Scheduling



Partitioned scheduling

- Two steps:
  - Determine a mapping of tasks to processors
  - Perform run-time scheduling
- Example: Partitioned with EDF
  - Assign tasks to the processors such that no processor's capacity is exceeded (utilization bounded by 1.0)
  - Schedule each processor using EDF

Bin-packing algorithms

- The problem concerns packing objects of varying sizes in boxes (”bins”) with the objective of minimizing number of used boxes.
  – Solutions (Heuristics): Next Fit and First Fit
- Application to multiprocessor systems:
  – Bins are represented by processors and objects by tasks.
  – The decision whether a processor is ”full” or not is derived from a utilization-based schedulability test.

Rate-Monotonic-First-Fit (RMFF): [Dhall and Liu, 1978]

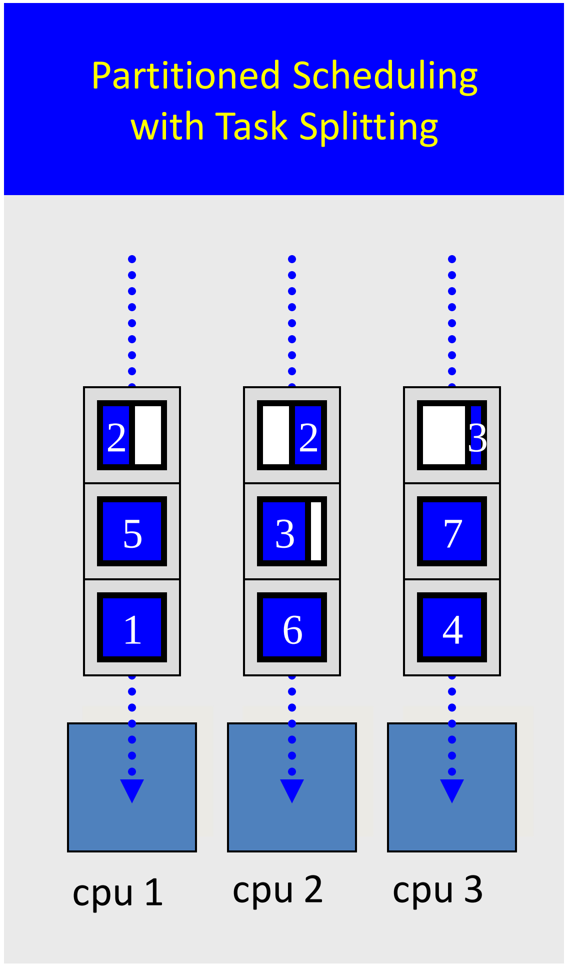
- First, sort the tasks in the order of increasing periods.
- Task Assignment
  - All tasks are assigned in the First Fit manner starting from *the task with highest priority*
  - A task can be assigned to a processor if all the tasks assigned to the processor are RM-schedulable i.e.
    - the total utilization of tasks assigned on that processor is bounded by $n(2^{1/n}-1)$ where n is the number of tasks assigned.

      (One may also use the Precise test to get a better assignment!)
  - Add a new processor if needed for the RM-test.

Partitioned scheduling

- Advantages:
  - Most techniques for single-processor scheduling are also applicable here
- Partitioning of tasks can be automated
  - Solving a bin-packing algorithm
- Disadvantages:
  - Cannot exploit/share all unused processor time
  - May have very low utilization, bounded by 50%

Partition-Based Scheduling with Task-Splitting



Partition-Based scheduling with Task Splitting

- High resource utilization
- High overhead (due to task migration)

Fixed-Priority Multiprocessor Scheduling