CS343: Operating System

Scheduling and Threading

Lect15 : 29th Aug 2023

Dr. A. Sahu

Dept of Comp. Sc. & Engg.

Indian Institute of Technology Guwahati

Outline

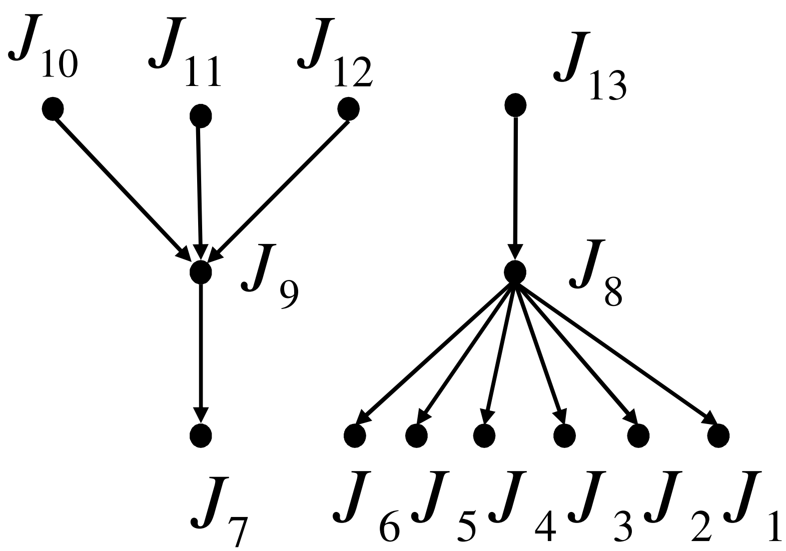
- List Scheduling of DAG
  - Graham's List scheduling
  - HU's CP Algorithm on Tree
  - CP Algorithm on DAG with pi=1
- MSF
- Practical Solution for $P||C_{max}$
  - Load balancing
  - In Distributed Setting
- Introduction :Threading and Synchronization

Graham's list scheduling

- **Set up a priority list L of jobs.**

How to set the priority ?

- **When a processor is idle, assign the first ready job to processor and remove it from the list L.**



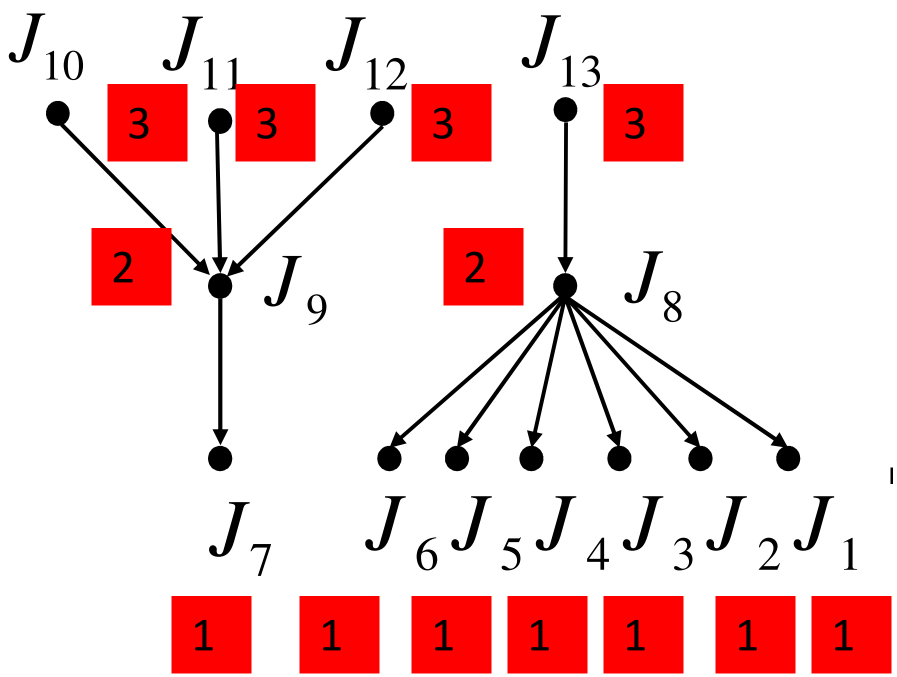
J_{11}	J_9	J_8	J_6	J_3
J_{10}	J_{13}	J_7	J_5	J_2
J_{12}			J_4	J_1

$$L=(J_9,J_8,J_7,J_6,J_5,J_{11},J_{10},J_{12},J_{13},J_4,J_3,J_2,J_1)$$

$Pm \mid prec, p_j = 1 \mid C_{max}$ (HLF/CP Algorithm)

- T. C. Hu (1961), **Critical Path (CP) /Hu's Highest Level First (HLF)** Algorithm
  - Assign a level h to each job.
    - **If job has no successors, h(j) equals 1.**
    - **Otherwise, h(j) equals one plus the maximum level of its immediate successors.**
  - Set up a priority list L by nonincreasing order of the jobs' levels.
  - Execute the list scheduling policy on this level based priority list L

HLF/CP algorithm



J_{10}	J_{13}	J_8	J_6	J_3
J_{11}	J_9	J_7	J_5	J_2
J_{12}			J_4	J_1

$$L=(\overbrace{J_{10},J_{11},J_{12},J_{13}}^{\text{Level 3}},\overbrace{J_9,J_8}^{\text{Level 2}},\overbrace{J_7,J_6,J_5,J_4,J_3,J_2,J_1}^{\text{Level 1}})$$

HLF/CP Algorithm

- **Time complexity**
  - $O(|V|+|E|)$ ($|V|$ is the number of jobs and $|E|$ is the number of edges in the precedence graph)
- **The HLF algorithm is Optimal for**
  - **$Pm \mid p_j = 1$, in-tree (out-tree) $\mid C_{max}$.**
  - **$Pm \mid p_j = 1$, in-forest (out-forest) $\mid C_{max}$.**

HLF/CP Algorithm

- The approximation ratio of HLF algorithm for the problem with **general precedence** constraints:

$$P_m \mid p_j = 1\text{ , prec} \mid C_{max}$$

Tight!

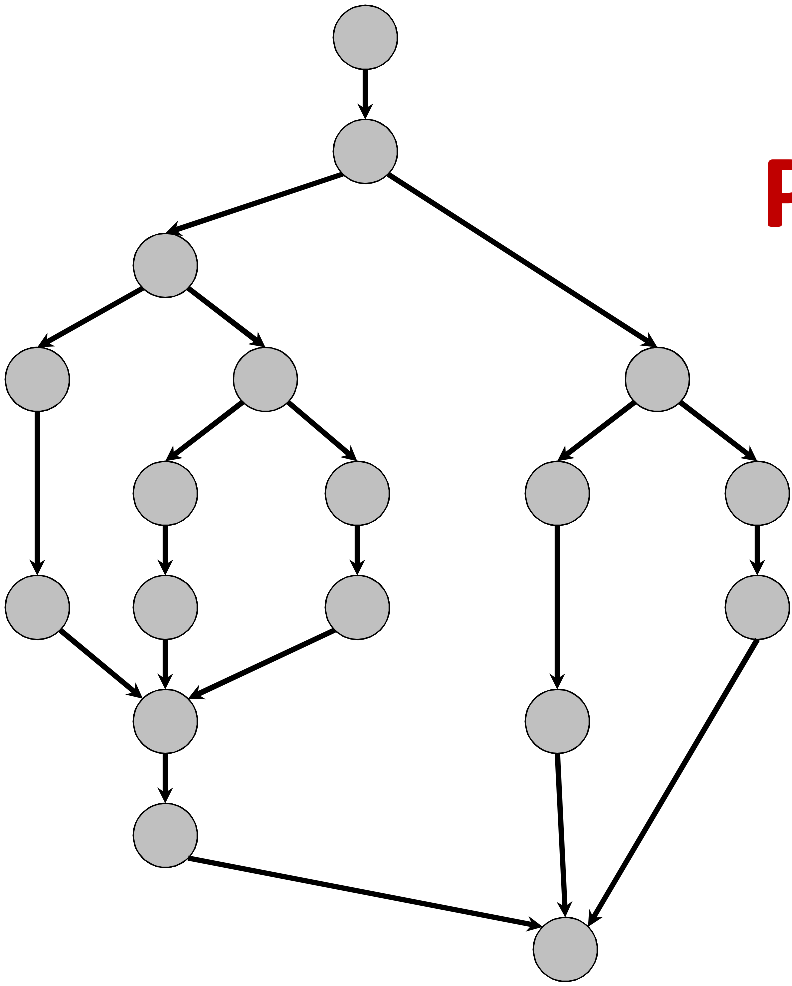
If $m = 2$, $\delta_{HLF} \leq 4/3$.

If $m \geq 3$, $\delta_{HLF} \leq 2 - 1/(m-1)$.

2 Approx from CLR Algorithm Book

CP Algo: CLR Book Page 779-783

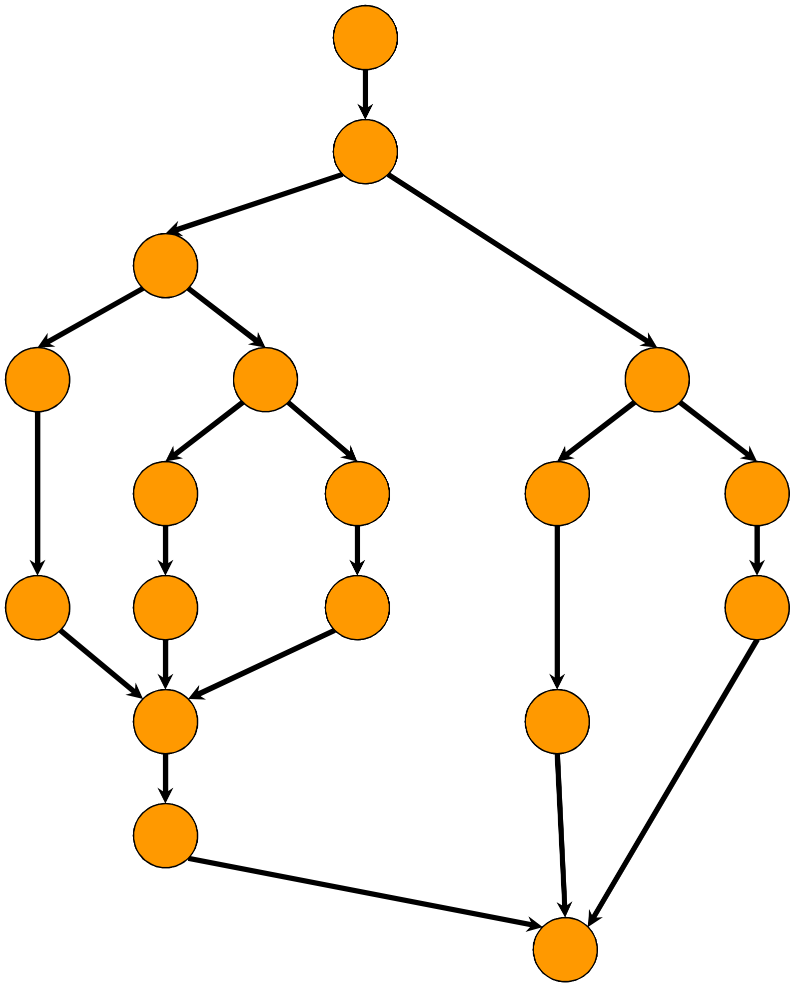
T_P = execution time on P processors



$Pm \mid p_j = 1 , prec \mid C_{max}$

CP Algorithms

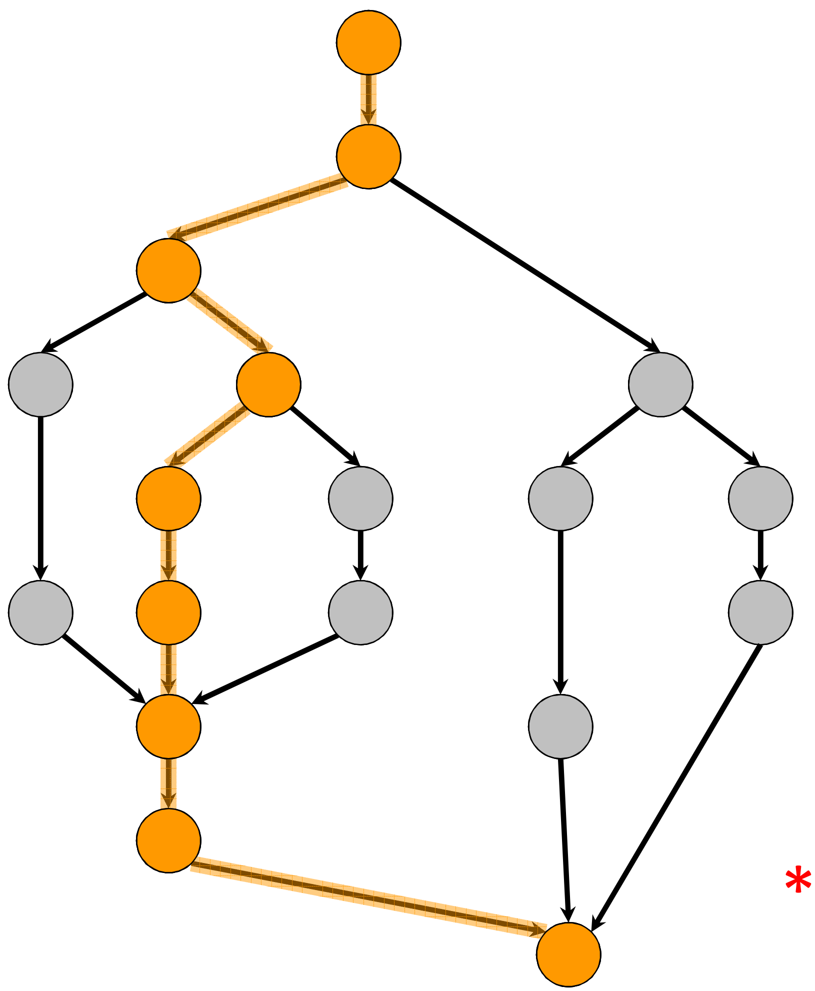
T_P = execution time on P processors



T_1 = ***work***

CP Algorithms

T_P = execution time on P processors



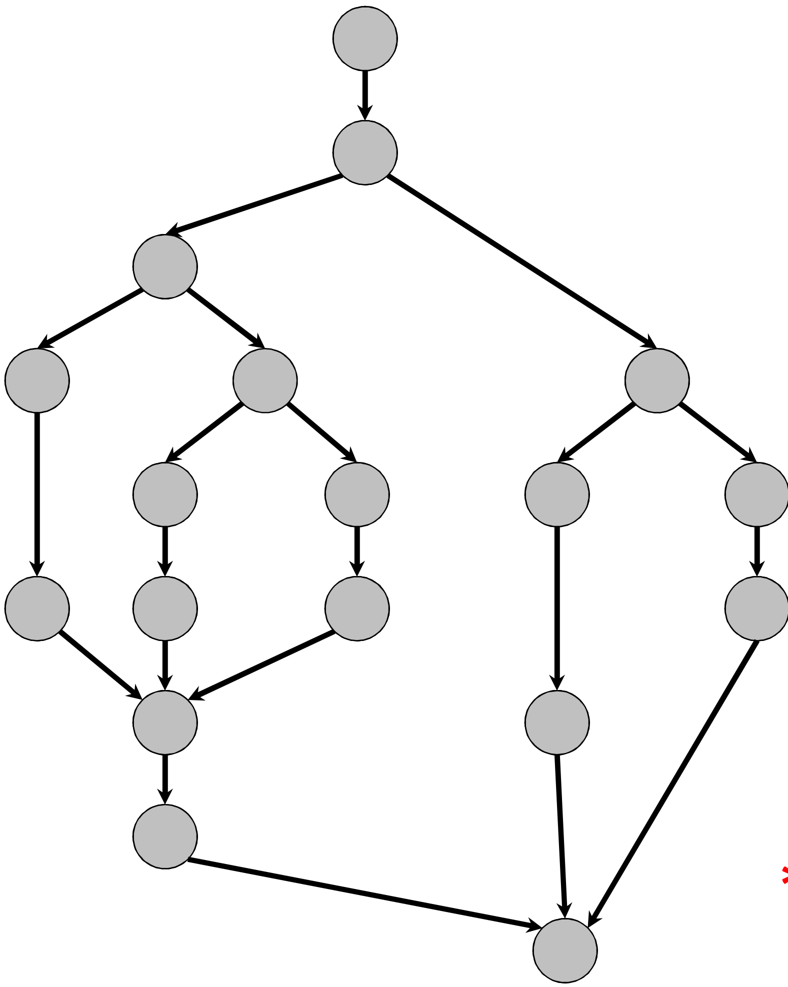
T_1 = ***work***

T_∞ = ***span****

* Also called ***critical-path length*** or ***computational depth***.

CP Algorithms

T_P = execution time on P processors



T_1 = ***work***

T_∞ = ***span****

LOWER BOUNDS

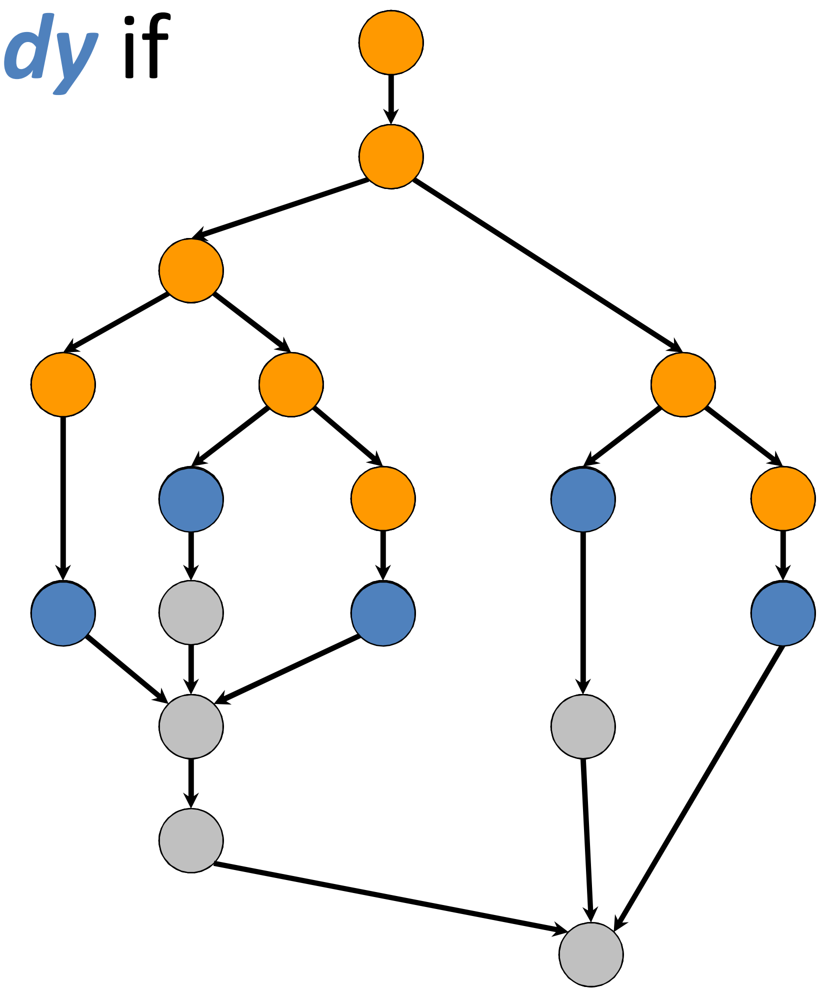
- $T_P \geq T_1/P$
- $T_P \geq T_\infty$

*Also called ***critical-path length*** or ***computational depth***.

CP: Greedy Scheduling

IDEA: Do as much as possible on every step.

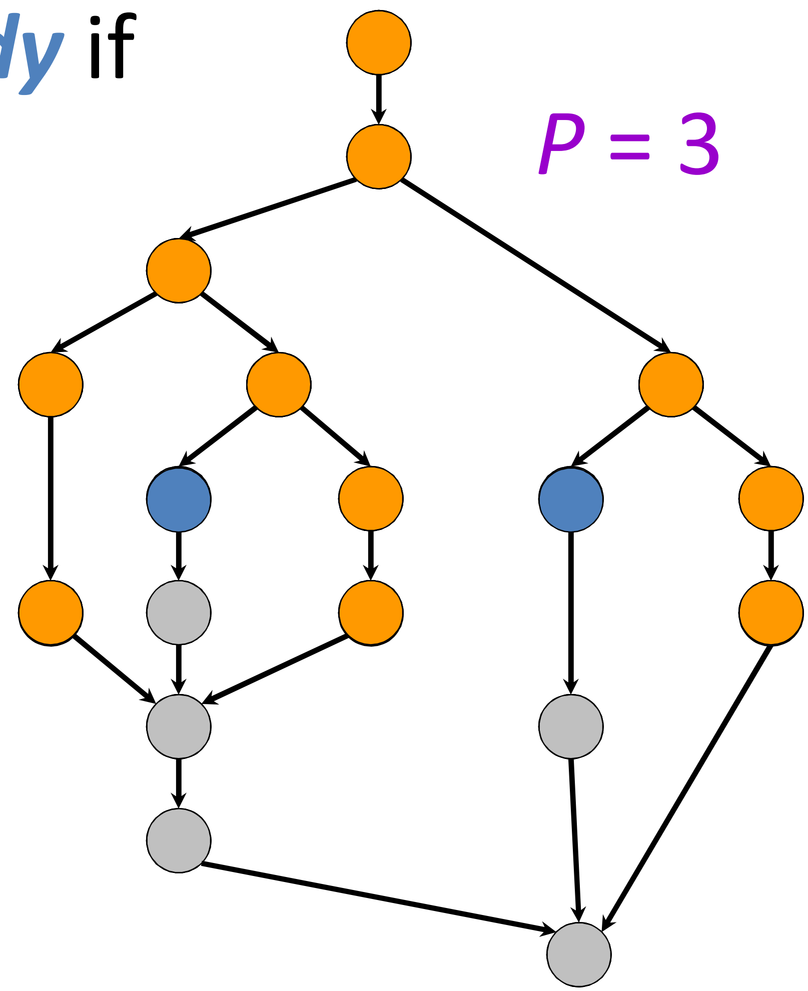
Definition: A node is ***ready*** if all its predecessors have ***executed***.



CP : Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition: A node is ***ready*** if all its predecessors have ***executed***.



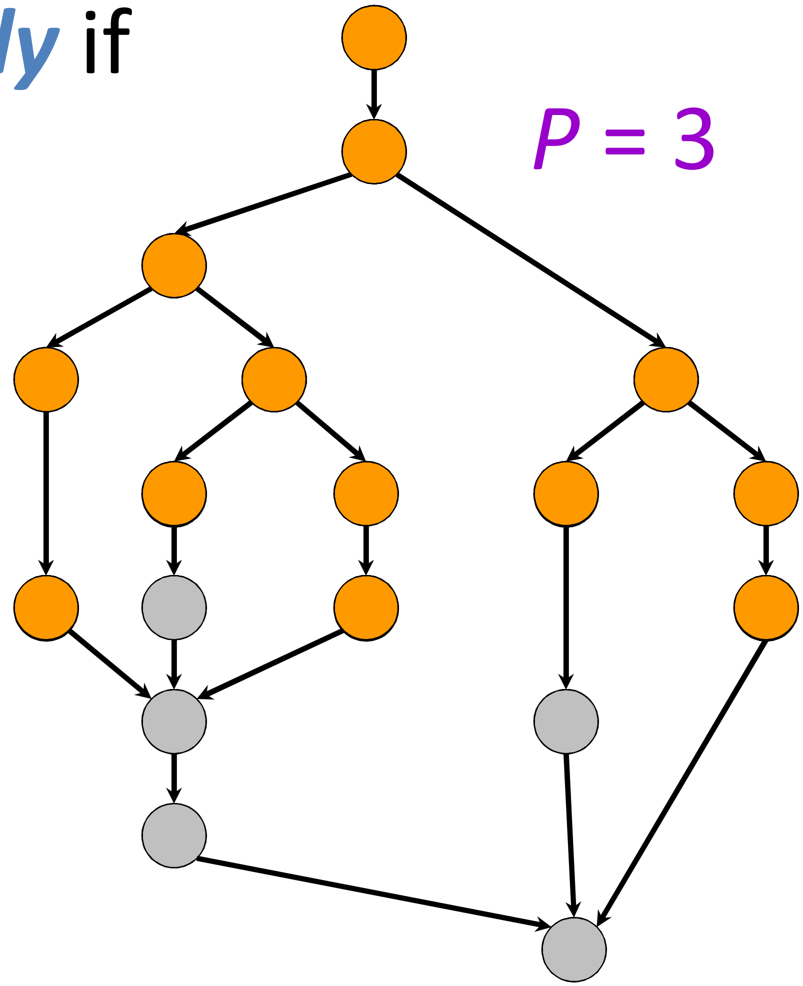
Complete step

- #ready task $\geq P$ cores.
- Run any P.

CP : Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition: A node is ***ready*** if all its predecessors have ***executed***.



Complete step

- #ready task $\geq P$ cores.
- Run any P.

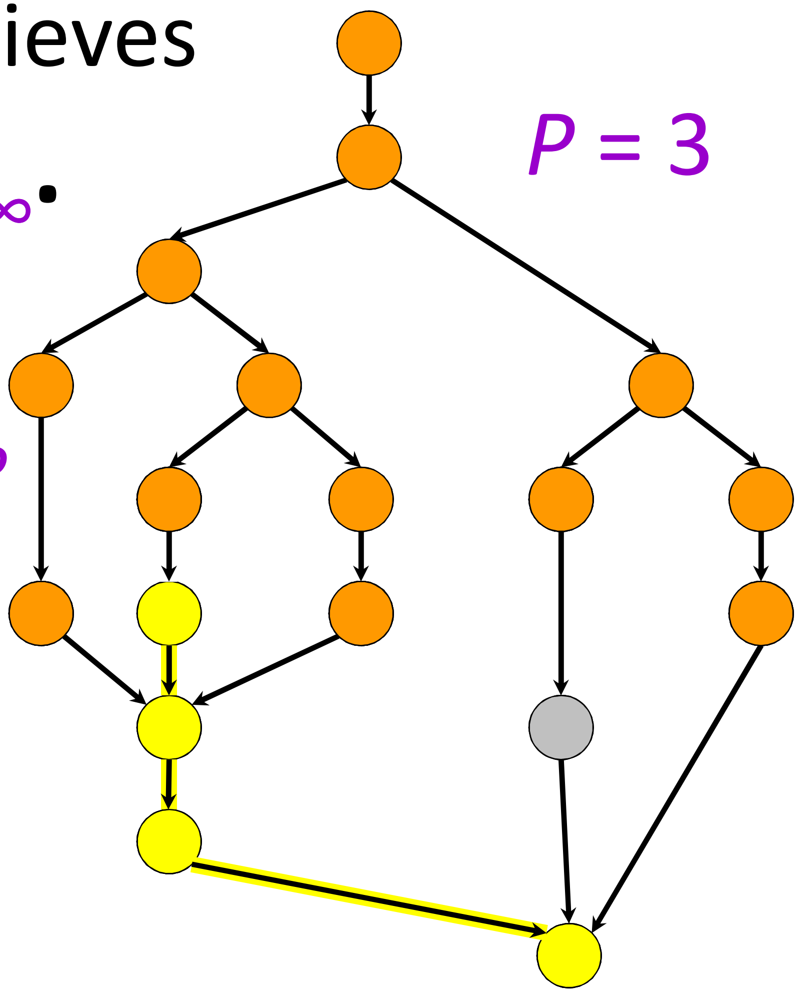
Incomplete step

- # ready task$< P$ cores.
- Run all of them.

CP: Greedy-Scheduling Theorem

Theorem [Graham '68 & Brent '75].
Any greedy scheduler achieves

$$T_P \le T_1/P + T_\infty.$$



$P = 3$

Proof.

- # complete steps $\le T_1/P$, since each complete step performs P work.
- # incomplete steps $\le T_\infty$, since each incomplete step reduces the span of the unexecuted dag by 1. ■

CP: Optimality of Greedy

Corollary. Any greedy scheduler achieves within a factor of 2 of optimal.

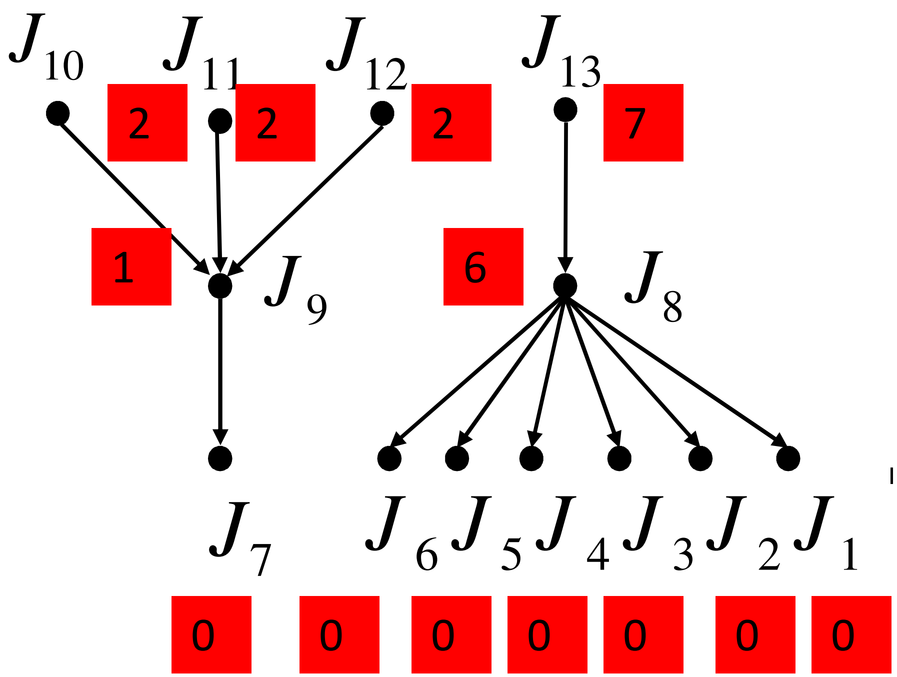
Proof. Let T_P^* be the execution time produced by the optimal scheduler. Since $T_P^* \geq \max\{T_1/P, T_\infty\}$ (lower bounds), we have

$$
\begin{aligned}
T_P &\leq T_1/P + T_\infty \\
&\leq 2 .\max\{T_1/P, T_\infty\} \\
&\leq 2T_P^* . \quad \blacksquare
\end{aligned}
$$

Most Successors First (MSF) Algo.

- Set up a priority list L by nonincreasing order of the jobs' successors numbers.
  - (i.e. the job having more successors should have a higher priority in L than the job having fewer successors)
- Execute the list scheduling policy based on this priority list L.

MSF Algorithm



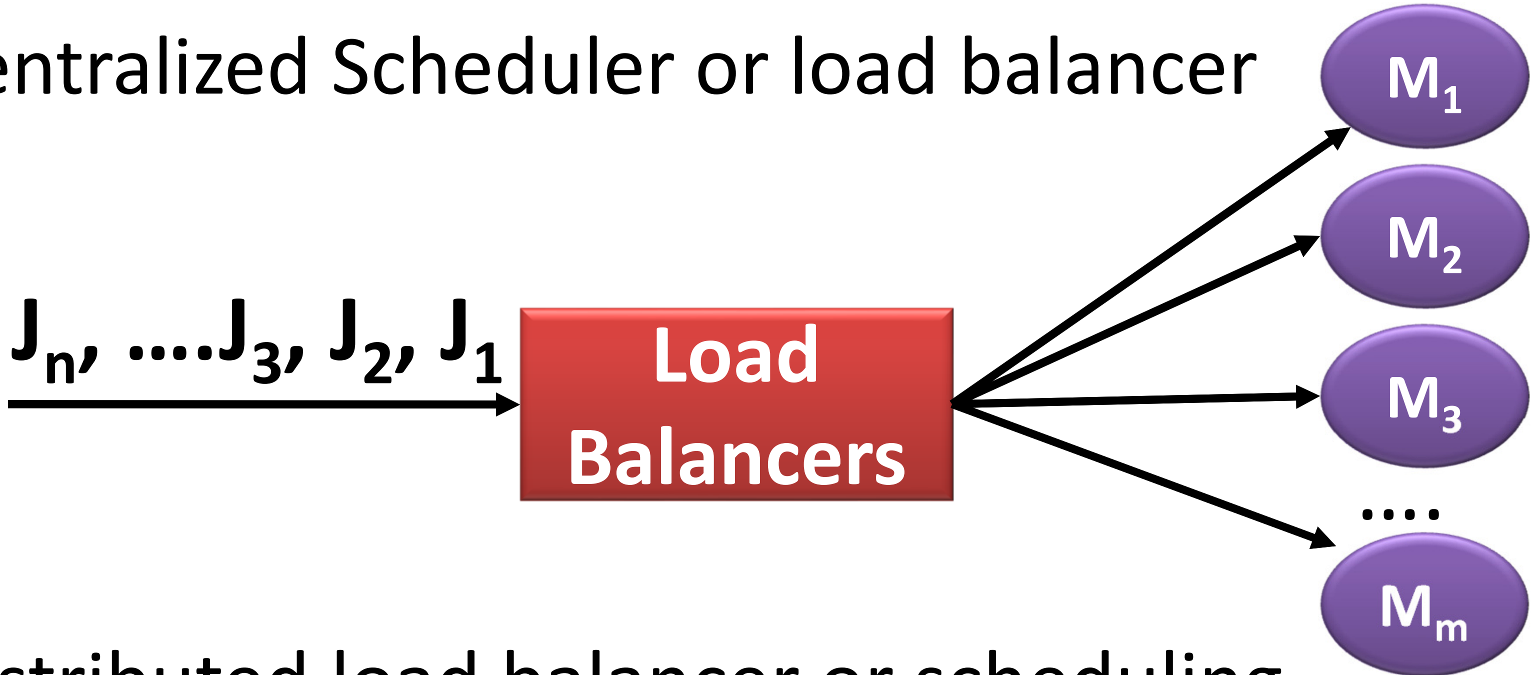
J_{13}	J_{10}	J_9	J_7	J_2
J_{12}	J_8	J_6	J_4	J_1
J_{11}		J_5	J_3	

$$L=(J_{13},J_8,J_{12},J_{11},J_{10},J_9,J_7,J_6,J_5,J_4,J_3,J_2,J_1)$$

7 6 2 2 2 1 0 0 0 0 0 0 0

P||Cmax In Practice

- Centralized Scheduler or load balancer

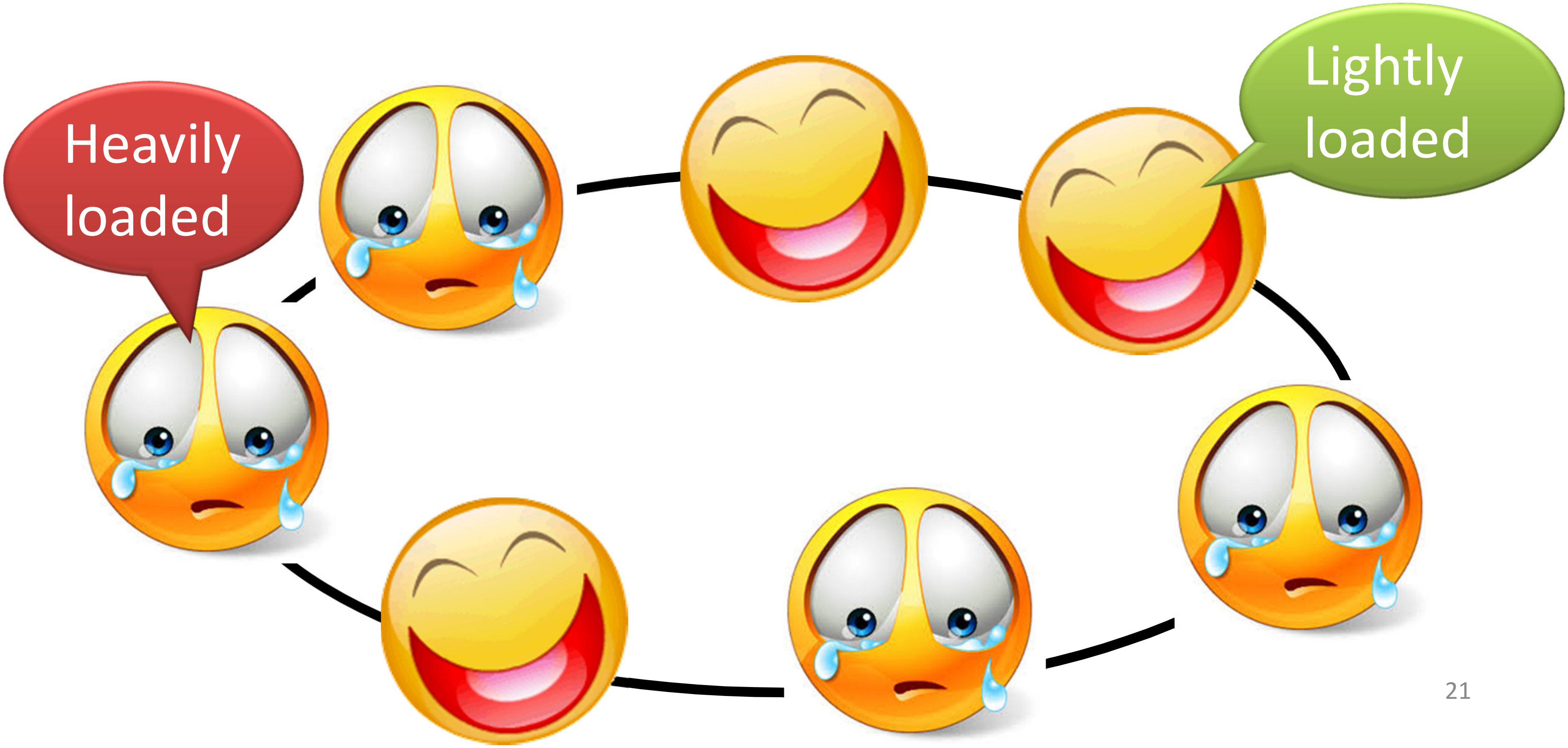


- Distributed load balancer or scheduling
  – Every one act as peers
  – Every one participate in load balancing

Distributed Scheduling

- Resource management component of a system
  - Which moves jobs around the processors
  - To balance load
  - And maximize overall performance
- Needed because of uneven distribution of tasks on individual processors
  - Can be due to several reasons
  - Can even make sense for homogeneous systems with (on average) even loads.

Load Sharing



Load: How does one characterize ?

- Performance : average response time
- Load:
  - It has been shown that queue lengths for resources (e.g. CPUs) can be a good indicator.
  - How does one handle the delay of transfer when systems are unevenly loaded and we seek to rectify that ?
    - Timeouts, holddowns
  - Queue length not very appropriate for (nor correlated with) CPU utilization for some tasks (e.g. interactive).

Load Balancing Approaches

- Static
  - Decisions are “hard wired”
  - A-priori into the system based on designers understanding
- Dynamic
  - Maintain state information for the system
  - And make decisions based on them
  - Better than static, but have more overhead.
- Adaptive
  - A subtype of dynamic, they can change the parameters they analyze based on system load.

Load Balancing Approaches

- Load balancing vs. Load sharing
  - Centralized: Balancing typically involves more transfers.
  - Distributed: Sharing algorithms transfer in anticipation

Load Sharing

- Sender Initiated
- Receiver Initiated
- Symmetrically Initiated

Sender Initiated Algorithms

- The overloaded node attempts to send tasks to lightly loaded node
- Transfer Policy
  - If new Tasks takes you above threshold, become sender.
  - If receiving task will not lead to crossing over threshold, then become receiver

Sender Initiated Algorithms

- Selection Policy: Newly arrived tasks
- Location Policy
  - Random – still better than no sharing. Constrain by limiting the number of transfers
  - Threshold – chose nodes randomly but poll them before sending task. Limited no. of polls. If process fails execute locally.
  - Shortest – Poll all randomly selected nodes and transfer to least loaded. Doesn’t improve much over threshold.

Receiver initiated Algorithms

- Receiver
  - Load sharing process initiated by a lightly loaded node
- Transfer Policy
  - Threshold based
- Selection Policy
  - Can be anything

Receiver initiated Algorithms

- Location Policy
  - Receiver selects upto N nodes and polls them, transferring task from the first sender.
  - If none are found, wait for a predetermined time, check load and try again
- Stability
  - At high loads, few polls needed since senders easy to find.
  - At low loads, more polls but not a problem. However, transfers will tend to be preemptive.

Symmetric Algorithms

- Simple idea
  - Combine the previous two
  - One works well at high loads, the other at low loads.
- Above Average Algorithm
  - Keep load within a range
- Transfer Policy
  - Maintain 2 thresholds equidistant from average
  - Nodes with load > upper are senders, Nodes with load < lower are receivers.

Multithreading
and
Multiprocessing

Doing Work Collaboratively

- Many application run multiple process/thread to do a collaborative work
- **Multi-process Apps**
  - Example: Chrome Browser, IE9, Adobe PDF,
- **Multi-threaded Apps**
  - Core Video studio, Adobe Phtoshop, MS Excel
- From user point of view all are same..
  - they can take benefit of multicore

Multithreading

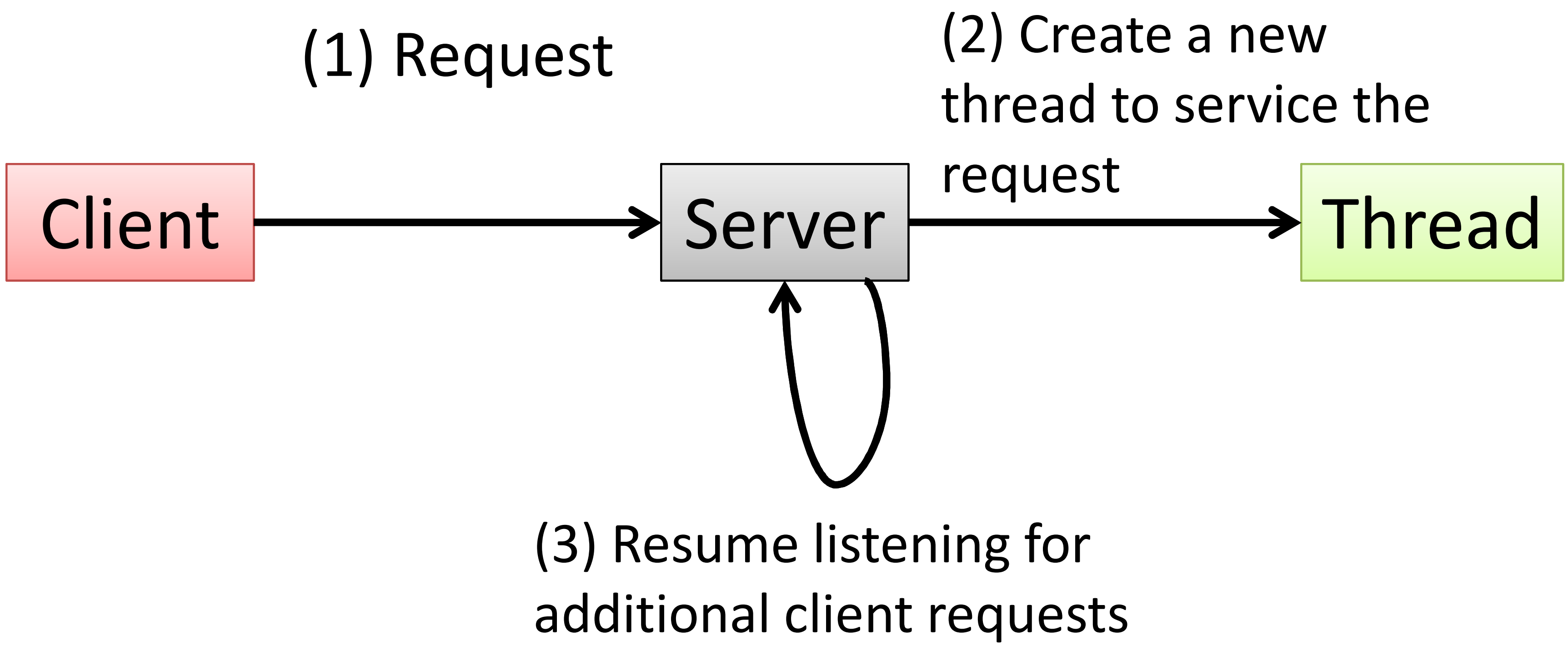
Thread: Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display, Fetch data, Spell checking, Answer a network request

Thread: Motivation

- Process creation is heavy-weight while thread creation is light-weight
  - Thread as Light Weight Process
- Can simplify code, increase efficiency
- **Kernels are generally multithreaded**

Multithreaded Server Architecture



Benefits of multithreading

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than **shared memory or message passing (to be discussed IPC/send/pipe)**
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

Multicore Programming

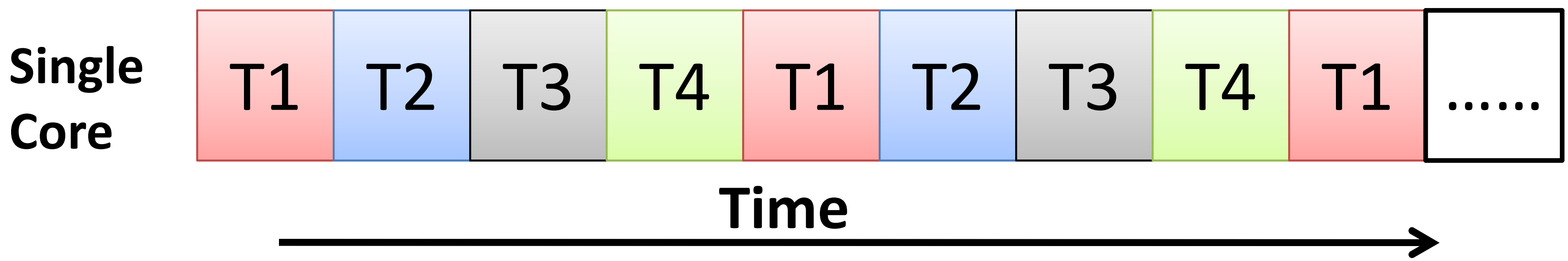
- **Multicore or multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities, Balance**
  - **Data splitting, Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Multicore Programming (Cont.)

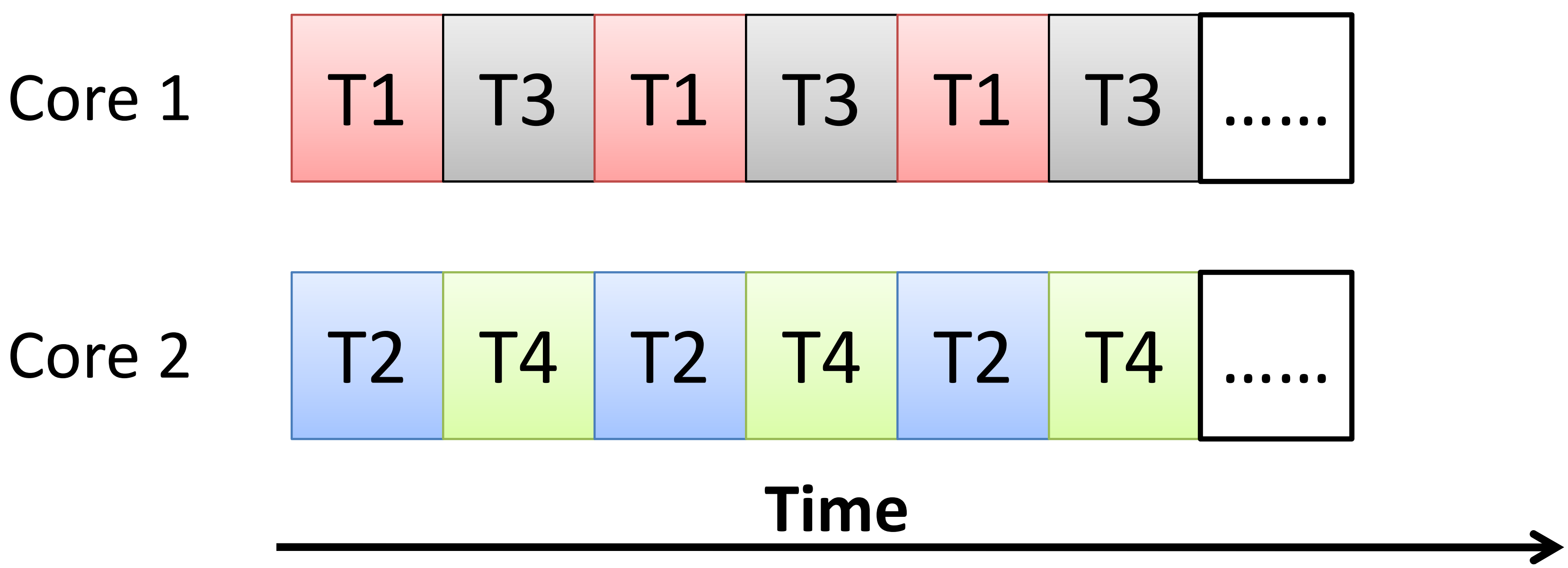
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - **AMD thread ripper** with 128 cores, and 2 hardware threads per core
  - **Intel Core i7**: 8 cores, 2 thread per core

Concurrency vs. Parallelism

- Concurrent execution on single-core



- Parallelism on a multi-core system:



Single and Multithreaded Processes