Sharif University of Technology
Department of Computer Science and Engineering

Lec. 8:

Task Mapping and Scheduling on Multicore Systems

Real-Time Computing

S. Safari

2023

Introduction

- We are going to consider multiprocessors, due to their widespread use in the form of multi-core systems.
- A large number of issues have to be considered during the transition from uniprocessors to multiprocessors.
- Initially, we assume having m identical processors (or “cores”).
- Assume dealing with a task system $\tau=\{\tau_1, ..., \tau_n\}$ where each task τ_i is characterized by its worst case execution time (WCET) C_i and -in case of periodic or sporadic tasks- its period T_i which is considered to also define the deadline unless otherwise noted.
- Whenever the periodic or sporadic nature of tasks is not relevant, we may also consider a set of jobs with explicit deadlines instead.

Introduction (2)

- For multiprocessors, it is not sufficient to decide when to execute tasks or their jobs.
- Rather, we must decide when to execute jobs and where to execute them.
- For *m* identical processors, obvious necessary conditions for schedulability are:

$$\forall i : u_i \leq 1$$

$$U_{sum} \leq m$$

Introduction (3)

- Scheduling for Independent Jobs on homogeneous multiprocessors
  - Partitioned Scheduling
  - Global Dynamic-Priority Scheduling
    - Proportional Fair (Pfair) Scheduling
  - Global Fixed-Job-Priority Scheduling
    - G-EDF Scheduling
    - EDZL Scheduling
  - Global Fixed-Task-Priority Scheduling
    - Global Rate Monotonic Scheduling
    - RMZL Scheduling
- Scheduling for dependent Jobs on homogeneous multiprocessors
  - As-Soon-as-Possible Scheduling
  - As-Late-as-Possible Scheduling
  - List Scheduling

Partitioned Scheduling

- Each task is allocated to a particular processor.
- Task migration is not allowed.
- Partitioned scheduling for synchronous arrival times can be done by bin packing, defined in a notation adjusted for real-time scheduling as follows:
- Definition of bin packing: Let $\tau = \{1, \ldots, n\}$ be a set of items, where each item $i \in \tau$ has a size $c_i \in (0, 1]$. Let $\pi = \{1, \ldots m\}$ be a set of bins with capacity one. The problem of finding an assignment $a : \tau \rightarrow \pi$ such that the number of nonempty bins $m \leq n$ is minimal and such that allocated sizes do not exceed the bin capacity is called the bin packing problem.
- Bin packing is known to be NP-hard. Hence, optimal algorithms need large run-times. Formalization of the scheduling problem as a bin packing problem aims at the minimization of the number of processors m.

Partitioned Scheduling (2)

- The heuristics are obtained by combining all possible combinations of two characteristics:
  1. The order in which tasks are considered: tasks can be considered in decreasing order of utilization (denoted by D), in increasing order of utilization (denoted by I), and in arbitrary order (denoted by an empty character).
  2. The search strategy for processor allocation: we consider processors to be ordered in some way. Then, the first fit strategy (FF) will allocate the first processor on which it fits. The worst fit strategy (WF) will allocate the processor with the largest remaining capacity. The best fit strategy (BF) will allocate the processor with the minimum remaining capacity on which it fits.
- There are no precedence, no preemption, and only identical processors.

Global Dynamic-Priority Scheduling

- Having unused processors in the presence of available jobs can be avoided with global scheduling.
- For global scheduling, the allocation of processors to tasks or jobs is dynamic.
- This gives us more flexibility, especially in the presence of changing workloads or processor availabilities.
- In the absence of execution constraints, upper bounds on the utilization is:

$$U_{sum} \leq m$$

- m is the number of processors (cores).

Proportional Fair (Pfair) Scheduling

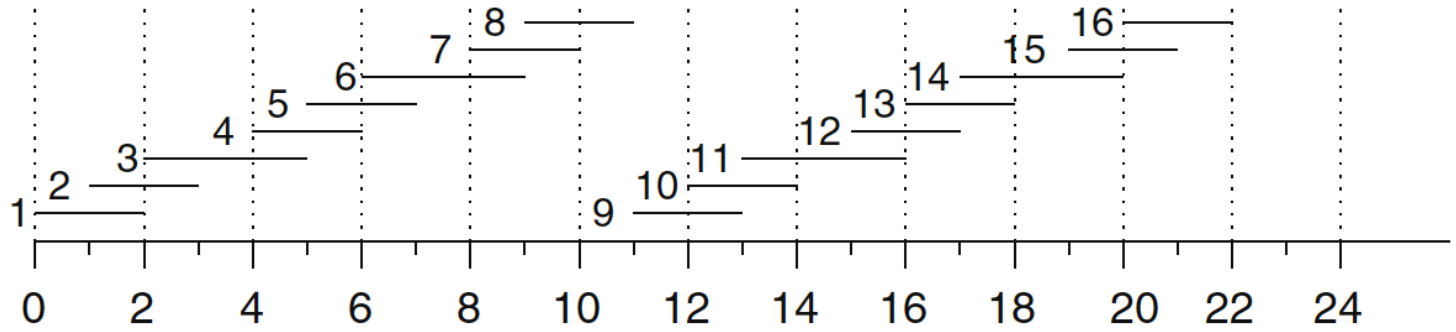
- The key idea of proportional fair (pfair) scheduling is to execute each task at a rate corresponding to its utilization.
- For pfair scheduling, we assume that time is quantized and enumerated with integers. Also, C_i and T_i parameters are represented by integers.
- For pfair scheduling, we divide each task τ_i into subtasks τ_i^j, where j enumerates the execution intervals.
  - For each subtask, we define a pseudo-release time and a pseudo-deadline:

$$r(\tau_i^j) = \left\lfloor \frac{j-1}{u_i} \right\rfloor$$

$$d(\tau_i^j) = \left\lceil \frac{j}{u_i} \right\rceil$$

Example of Proportional Fair (Pfair) Scheduling

- Consider a task τ_i with C_i = 8 and T_i = 11. Possible intervals for the number of allocated execution slots for each j are shown:



$$r(\tau_i^6) = \left\lfloor \frac{6-1}{8/11} \right\rfloor = \left\lfloor \frac{55}{8} \right\rfloor = 6 \qquad d(\tau_i^6) = \left\lceil \frac{6}{8/11} \right\rceil = \left\lceil \frac{66}{8} \right\rceil = 9$$

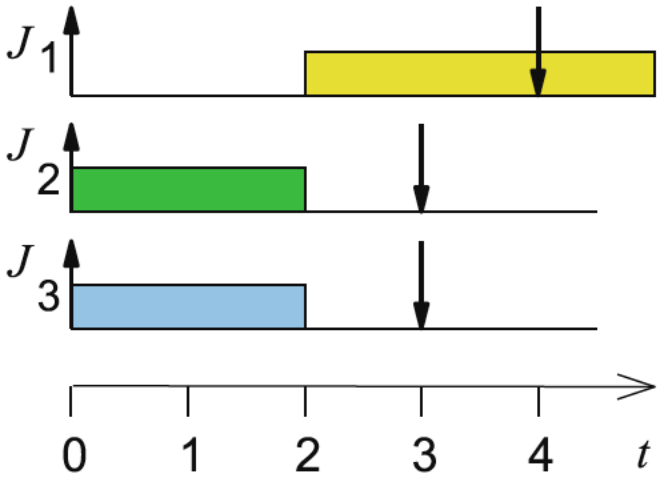
- We can apply EDF to pseudo-deadlines, or we can modify EDF by defining rules which are applied in case of ties.
- Pfair scheduling potentially suffers from a large number of migrations between processors.

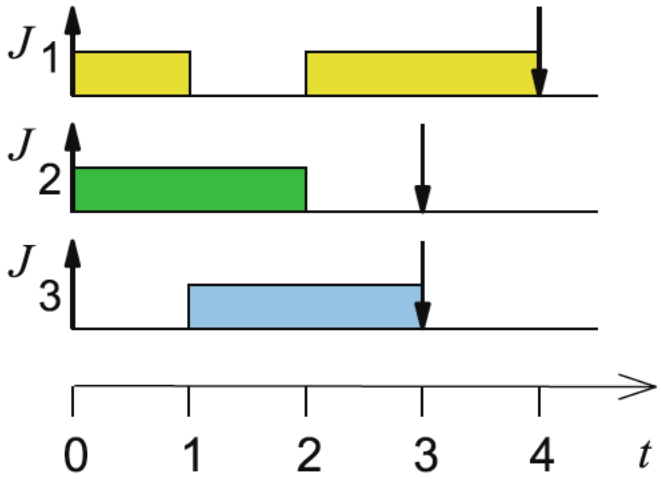
Global Fixed-Job-Priority Scheduling: (1) G-EDF Scheduling

- Solving the two-dimensional problem with extensions of uniprocessor scheduling algorithms.
- For example, we could use global EDF (G-EDF).
  - G-EDF, just like EDF, defines job priorities based on the closeness of the next deadlines.
- If m processors are available, those m jobs having the highest priorities among all available jobs are executed.
- Obviously, such priorities are job-dependent and not just task-dependent.
- In a global scheduling strategy, we would like to keep preemptions and task migrations to a minimum.

Example of G-EDF Scheduling

- Suppose m=2 and C_1=3, D_1=4, C_2=2, D_2=3, C_3=2, and D_3=3.
- G-EDF schedules J_2 and J_3 first, due to their earlier deadline.
- J_1 misses its deadline.
- However, a schedule is feasible, as shown in the following figure (right).





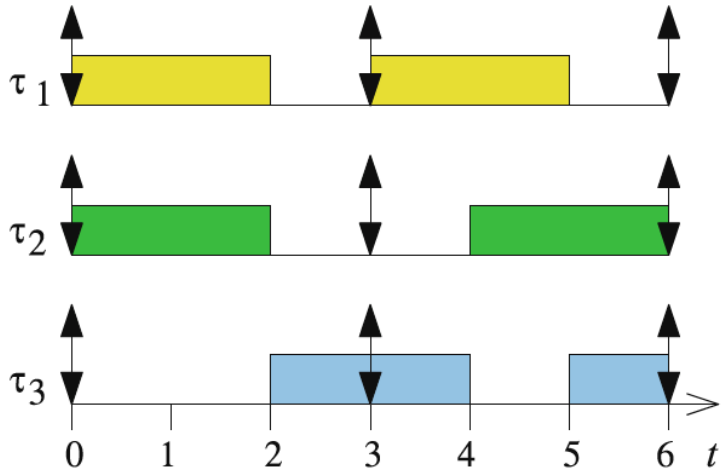
Global Fixed-Job-Priority Scheduling: (2) EDZL Scheduling

- Obviously, G-EDF can miss deadlines for task sets that are schedulable.
- We can improve G-EDF by adding a consideration of laxity: the EDZL (Earliest Deadline Zero Laxity) algorithm applies G-EDF as long as the laxity of jobs is greater than zero.
- However, whenever the laxity of a job becomes zero, the job gets the highest priority among all jobs, even including currently executing jobs.
- The literature proved the following utilization bound for EDZL:

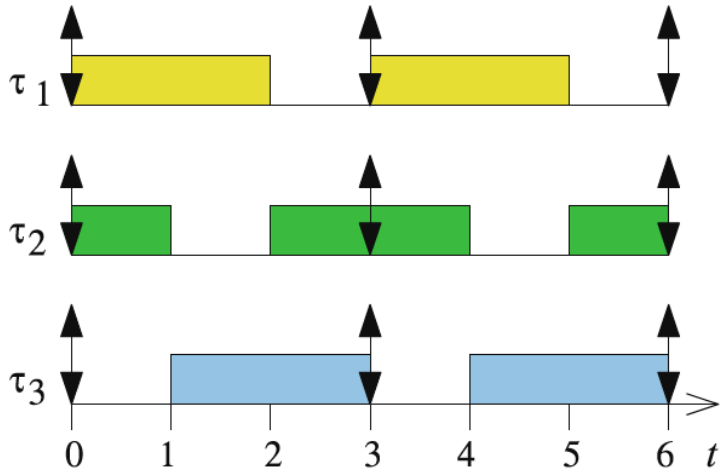
$$U_{sum} \leq \frac{m+1}{2}$$

Example of EDZL Scheduling

- Consider the parameters as follows: n=3, m=2, T_1=T_2=T_3=3, and C_1=C_2=C_3=2.
- For this example, G-EDF misses the deadlines for τ_3 at times t=3n for n=1, 2, 3.., as can be seen in the following figure (left).
- However, EDZL keeps the deadlines as can be seen in the following figure (right). The detailed behavior depends somewhat on the processor allocation used by EDZL.



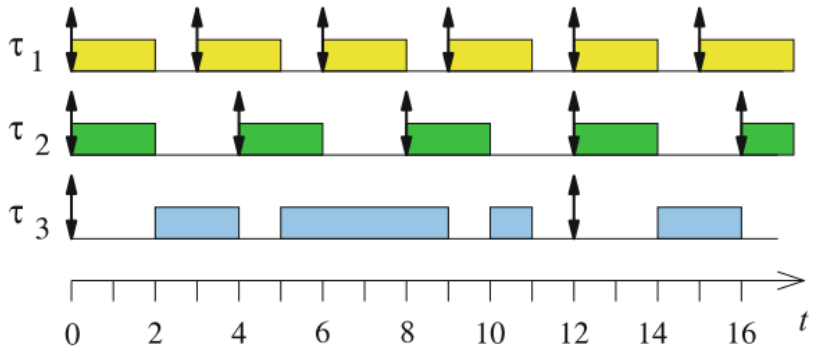
G-EDF scheduling



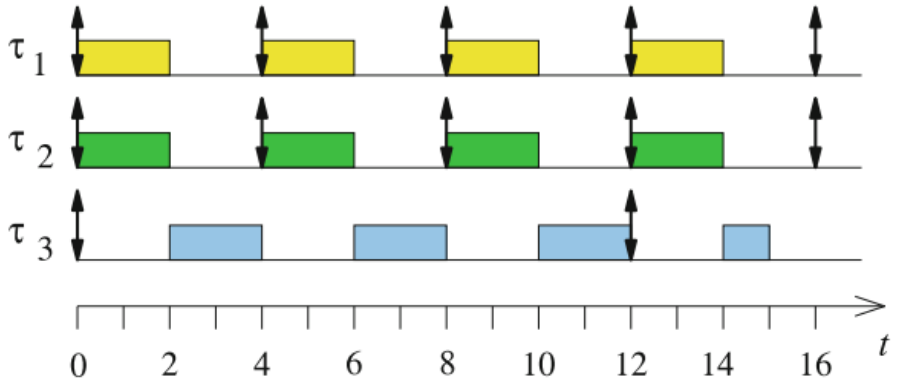
EDZL scheduling

Global Fixed-Task-Priority Scheduling: Global Rate Monotonic Scheduling

- Consider the case m = 2, n = 3, T1 = 3, C1 = 2, T2 = 4, C2 = 2, T3 = 12, and C3 = 7.



- For G-RM, there is an anomaly concerning relaxed schedules.
  - If we extend the period of T1 to T1 = 4, τ3 will miss its deadline



RMZL Scheduling

- G-RM might miss deadlines for task sets that are schedulable, and we can consider improvements.
- One such improvement is RMZL scheduling.
- For RMZL scheduling, we use (G-)RM scheduling as long as the current laxity is larger than zero.
- However, when the laxity becomes zero for one of the jobs, we raise its priority to the highest.
- RMZL scheduling is superior to RM scheduling, since schedules are changed only when RM scheduling could have missed a deadline.

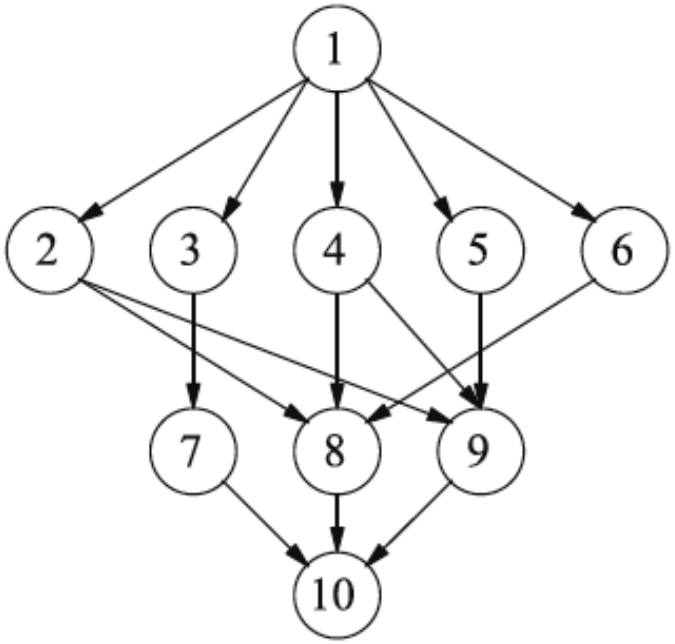
Dependent Jobs on Homogeneous Multiprocessors

As Soon As Possible (ASAP) scheduling

- The simplest type of scheduling occurs when we wish to optimize the overall latency of the computation and do not care about the number of resources required.
- This can be achieved by simply starting each operation (task) in a DAG as soon as its predecessors have completed.
  - Compute the set of unscheduled tasks for which all predecessors have finished their computation
  - Schedule these tasks to start at the current time step.
- ASAP scheduling considers a mapping of tasks to integer start times.
- Allocation to specific processors has to be performed after ASAP scheduling.
- Preemptions are not allowed.

As Soon As Possible (ASAP) scheduling: Example (1)

```
for (t=0; there are unscheduled tasks; t++)   {
   τ′={all tasks for which all predecessors finished};
   set start time of all tasks in τ′ to t;
}
```

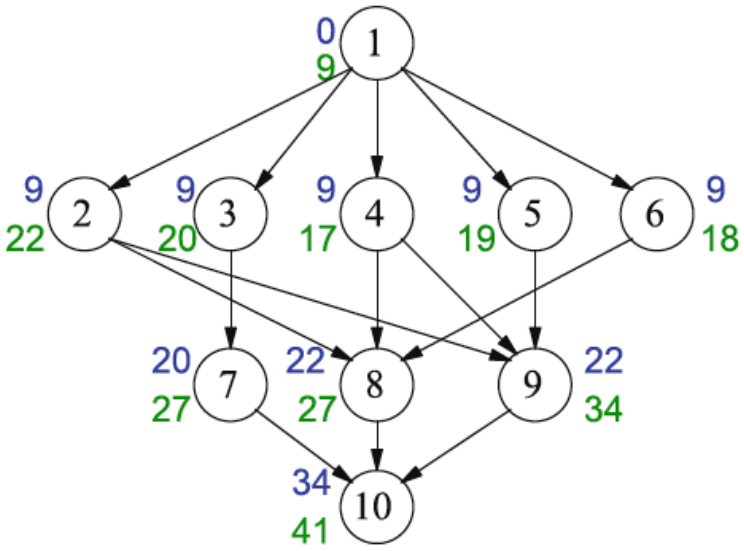


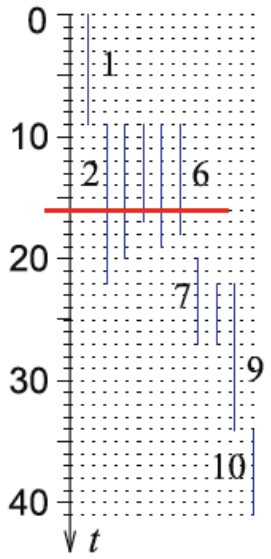
Task	C_i
1	9
2	13
3	11
4	8
5	10
6	9
7	7
8	5
9	12
10	7

As Soon As Possible (ASAP) scheduling: Example (2)

```
for (t=0; there are unscheduled tasks; t++)   {
   τ′={all tasks for which all predecessors finished};
   set start time of all tasks in τ′ to t;
}
```

Task	C_i
1	9
2	13
3	11
4	8
5	10
6	9
7	7
8	5
9	12
10	7





As Late As Possible (ALAP) scheduling

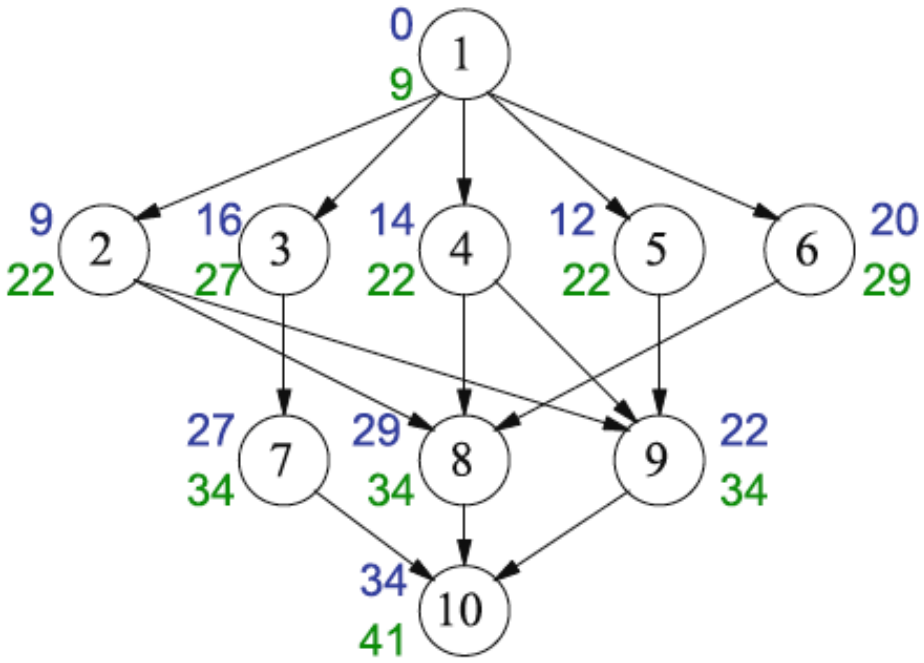
- The algorithm starts with tasks on which no other task depends.
- These tasks are assumed to finish at time 0.
- Their start time is then computed from their execution time.
- The loop then iterates backward over time steps. Whenever we reach a time step, at which a task should finish the latest, its start time is computed, and the task is scheduled.
- After finishing the loop, all times are shifted toward positive times such that the first task starts at time 0.

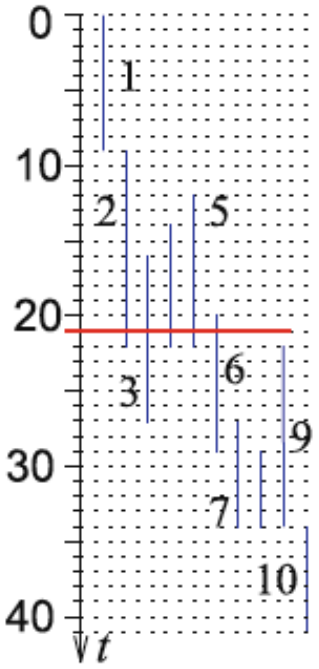
* Generate a list, starting at its end

As Late As Possible (ALAP) scheduling: Example

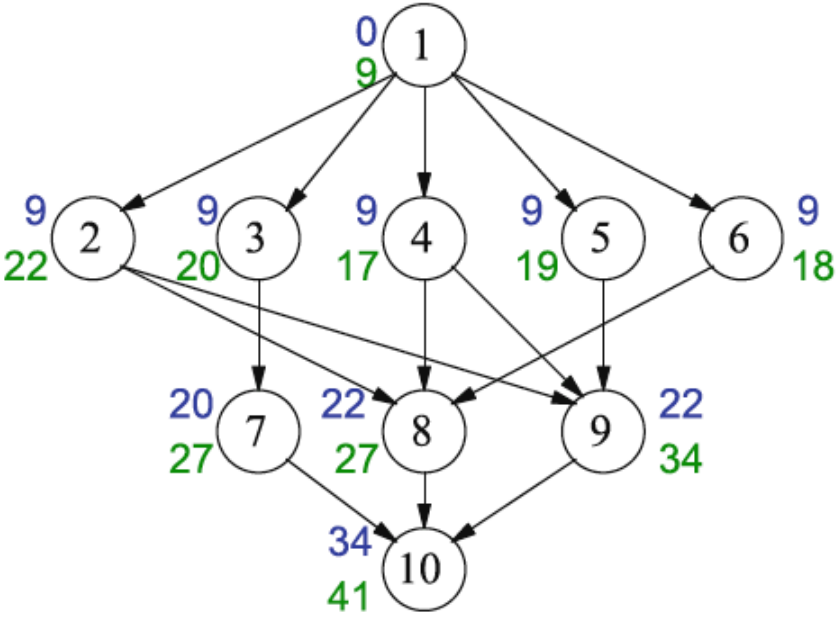
```
for (t=0; there are unscheduled tasks; t--) {
     τ'={all tasks on which no unscheduled task depends};
     set start time of all tasks in τ' to (t - their execution time);
}
Shift all times such that the first tasks start at t=0.
```

Task	C_i
1	9
2	13
3	11
4	8
5	10
6	9
7	7
8	5
9	12
10	7

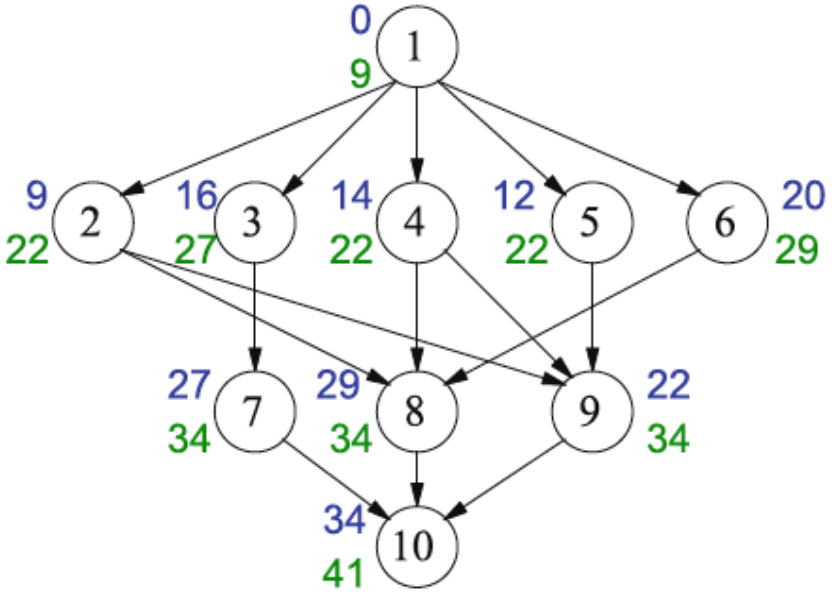




Resulting ASAP and ALAP Scheduling



ASAP scheduling



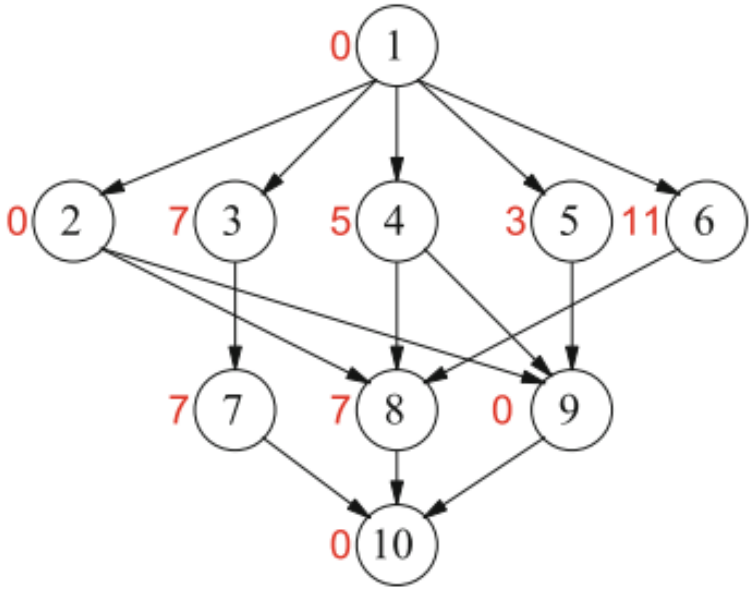
ALAP scheduling

List Scheduling

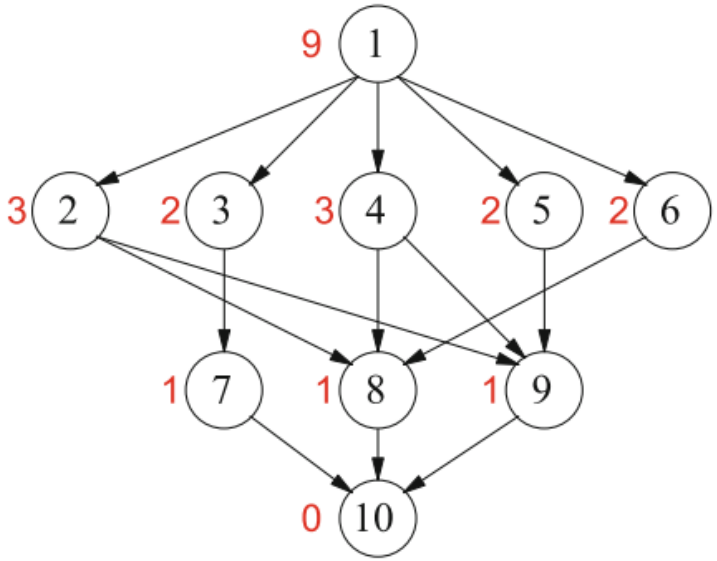
- List scheduling: extension of ALAP/ASAP method
- We assume that we have a set L of processor types.
- List scheduling requires the availability of a priority function reflecting the urgency of scheduling a certain task.
  - Possible priorities:
    - Mobility = τ (ALAP schedule)- τ (ASAP schedule)
    - Number of nodes below task τ_i in the tree
    - Path length for a task

List Scheduling with mobility/number of nodes below each task

- Mobility is defined as the difference between the start times for the ASAP and ALAP schedule.
- Obviously, scheduling is urgent for the four tasks on the critical path for which mobility is zero.



Mobility

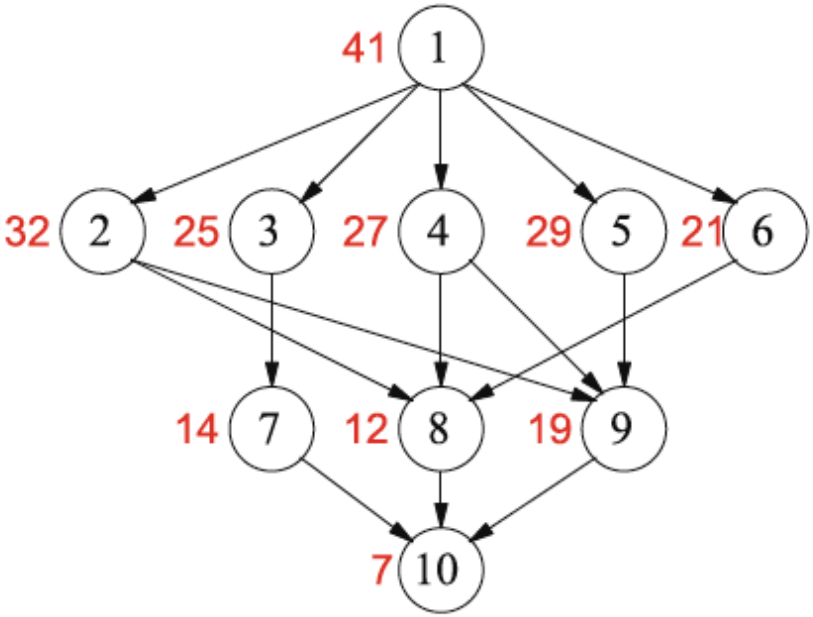


Number of nodes below each task

List Scheduling with Longest Path: Example

- The path length for a task is defined as the length of the path from starting at τ_i to finishing the entire graph G.

Task	C_i
1	9
2	13
3	11
4	8
5	10
6	9
7	7
8	5
9	12
10	7



We have just three processors