CIS 721 - Real-Time Systems
Lecture 3: Static Cyclic Scheduling

Mitch Neilsen
neilsen@ksu.edu

Outline

- **Approaches For Real-Time Scheduling (Ch. 4)**
  - **Clock-Driven (Static) Scheduling (Ch. 5)**
    - **Baker and Shaw, “The cyclic executive model and Ada”, IEEE Real-time Systems Symposium, pp. 120-129, 1988 (and on-line).**
    - **Liu textbook, Ch. 5**
  - Priority-Driven Scheduling (Ch.4, 6-7)

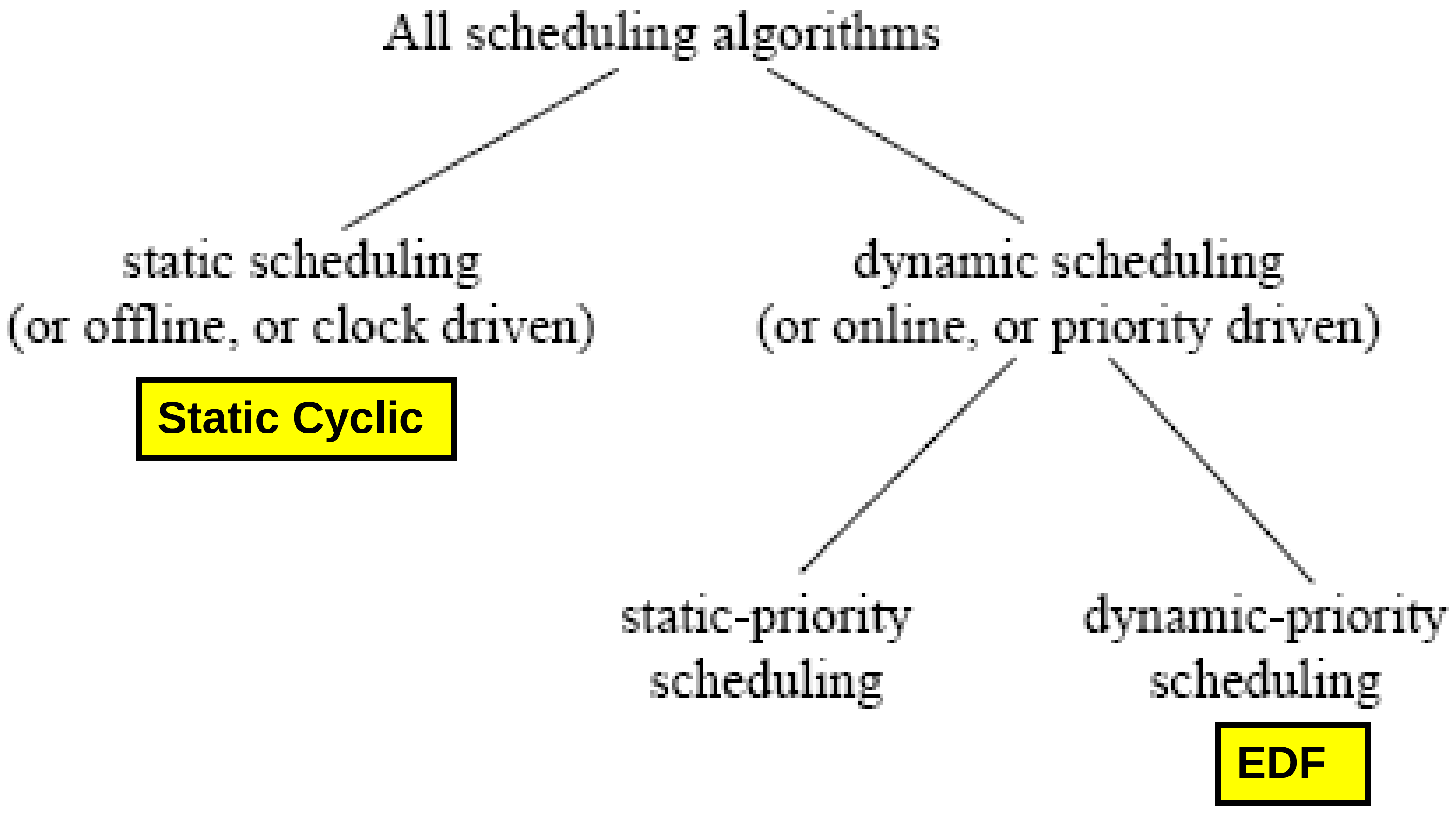
Temporal Parameters

- J_i : **job** – a unit of work
- T_i (or τ_i): **task** - a set of related jobs
- A **periodic task** is sequence of invocations of jobs with identical parameters.
- r_i: **release time** of job J_i
- d_i: **absolute deadline** of job J_i
- D_i: **relative deadline** (or just **deadline**) of job J_i
- e_i: (Maximum) **execution time** of job J_i

Schedules

- A **schedule** is an assignment of jobs to available processors. In a **feasible schedule**, every job starts at or after its release time and completes by its deadline in a hard real-time system.
- A scheduling algorithm is **optimal** if it always produces a feasible schedule if such a schedule exists.

Classification of Scheduling Algorithms



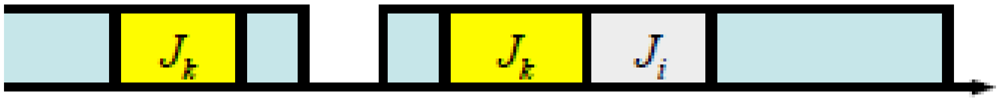
EDF Algorithm

- **Earliest-Deadline-First (EDF) algorithm:**
  - At any time, execute the available job with the **earliest deadline**.
- **Theorem: (Optimality of EDF):** In a system with **one processor** and **preemption** allowed, EDF is optimal; that is, EDF can produce a feasible schedule for a given job set J with arbitrary release times and deadlines, **if** a feasible schedule exists.
- **Proof:** Suppose that a feasible schedule S exists, then apply schedule transformations to S to generate an EDF schedule that is also feasible.

EDF proof (schedule transformations)

1. Any feasible schedule can be transformed into an EDF schedule
   - If J_i is scheduled to execute before J_k, but J_i's deadline is later than J_k's either:
     - The release time of J_k is after the J_i completes $\Rightarrow$ they're already in EDF order
     - The release time of J_k is before the end of the interval in which J_i executes:

- Swap J_i and J_k (this is always possible, since J_i's deadline is later than J_k's)



- Move any jobs following idle periods forward into the idle period

$\Rightarrow$ the result is an EDF schedule

2. So, if EDF fails to produce a feasible schedule, no feasible schedule exists
   - If a feasible schedule existed it could be transformed into an EDF schedule, contradicting the statement that EDF failed to produce a feasible schedule

EDF may not be optimal

- When preemption is not allowed:

$$
\begin{array}{cccccc}
 & & r_i & d_i & e_i & \\
J_1 & = (& 0, & 10, & 3 &) \\
J_2 & = (& 2, & 14, & 6 &) \\
J_3 & = (& 4, & 12, & 4 &)
\end{array}
$$

- When more than one processor is used:

$$
\begin{array}{cccccc}
 & & r_i & d_i & e_i & \\
J_1 & = (& 0, & 4, & 1 &) \\
J_2 & = (& 0, & 4, & 1 &) \\
J_3 & = (& 0, & 5, & 5 &)
\end{array}
$$

Clock-Driven Scheduling (Ch. 5)

- Idea: Compute a better **static schedule off-line** (e.g. at design time or during configuration) – note that we can afford to use expensive algorithms.
- Only **periodic tasks** are scheduled. Idle times can be used for aperiodic jobs.
- **Possible implementation: Table-driven scheduler**
  - Scheduling table has entries of type $\mathbf{(t_k, J(t_k))}$, where:
    - $\mathbf{t_k}$ is the decision time, and
    - $\mathbf{J(t_k)}$ is the set of jobs to start at time $\mathbf{t_k}$
- **Input:** Schedule $\mathbf{(t_k, J(t_k)), k = 0, 1, \ldots, N-1}$

Table-Driven Scheduling

Task Scheduler:

i := 0; k := 0;

<set timer to expire at time t_0>

BEGIN LOOP

<wait for timer interrupt, if an aperiodic job is executing, preempt it. >

i := i+1;

k:= i mod N;

<set timer to expire at time (i DIV N)*H + t_k >

IF J(t_{k-1}) is empty

THEN wakeup(aperiodic)

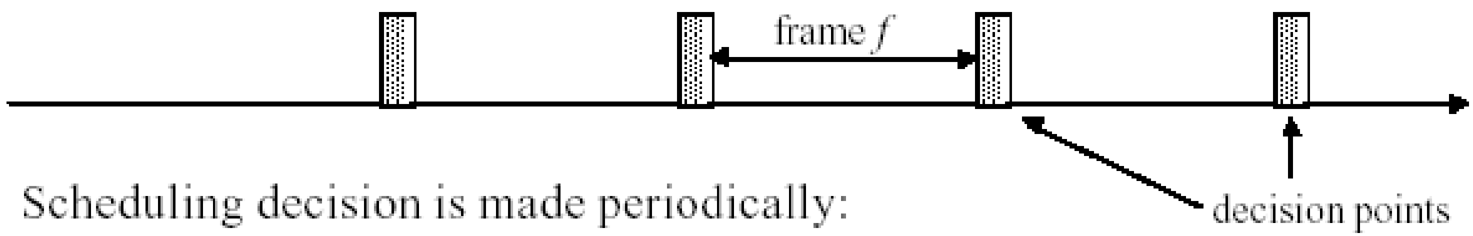
ELSE wakeup(J(t_{k-1}))

END LOOP

END Scheduler;

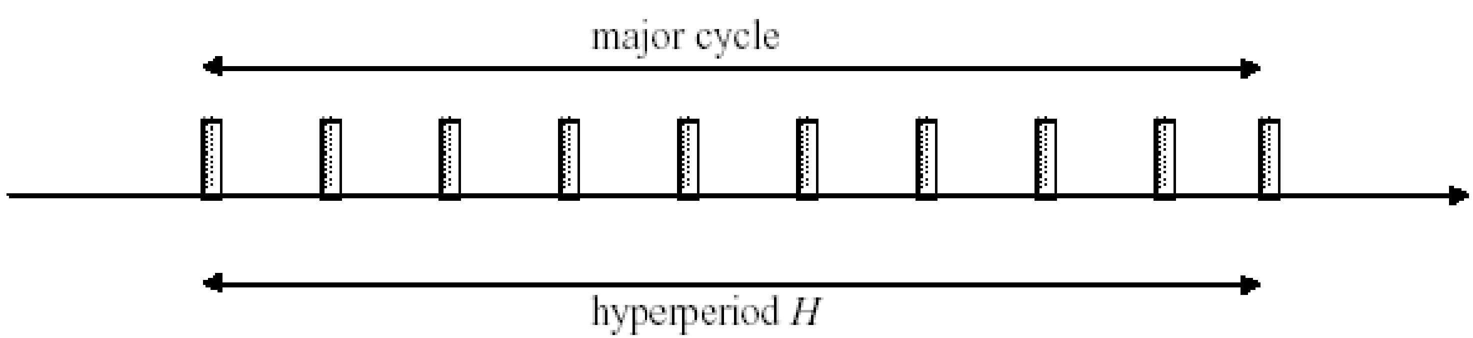
Cyclic Schedules: General Structure

- Scheduling decision is made periodically:



- Scheduling decision is made periodically:
  - choose which job to execute
  - perform monitoring and enforcement operations

- **Major Cycle**: Frames in a hyperperiod.



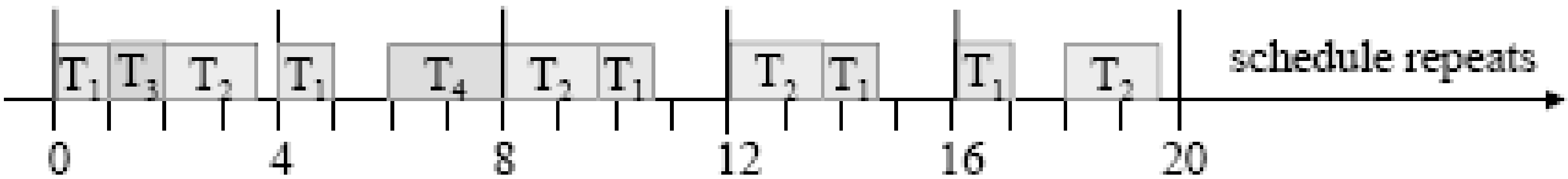
Example

Example

Consider a system of four tasks, $T_1 = (4, 1)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1)$ $T_4 = (20, 2)$.

(period = 4, execution time = 1)

Consider the following static schedule:



The first few table entries would be: $(0, T_1)$, $(1, T_3)$, $(2, T_2)$, $(3.8, I)$, $(4, T_1)$, …

Static Cyclic Scheduling

- Jobs in a periodic task are statically assigned to fixed time intervals in a cycle.
- A single **major cycle** can be divided into several smaller **minor cycles** of equal length.
- Given a task set and possible major and minor cycle lengths, **maximum network flow** algorithms can be used to determine if a feasible cyclic schedule exists.

Minor Cycle (Frame) Size Constraints

- Let ***f*** denote the **frame size**.
- Ideally, every job should be able to start and complete its execution within a single frame:

$$f \geq \max_{1 \leq i \leq n}(e_i)$$

Minor Cycle (Frame) Size Constraints

- To keep the length as short as possible, *f* should divide the hyperperiod ($H = lcm(p_1, .. , p_n)$); that is, *f* should divide the period of at least one task:

$$\lfloor p_i / f \rfloor - p_i / f = 0 \text{ for some } 1 \le i \le n$$

- Let ***F = H/f*** , then there are *F* minor cycles of length *f* in a major cycle of length *H.*

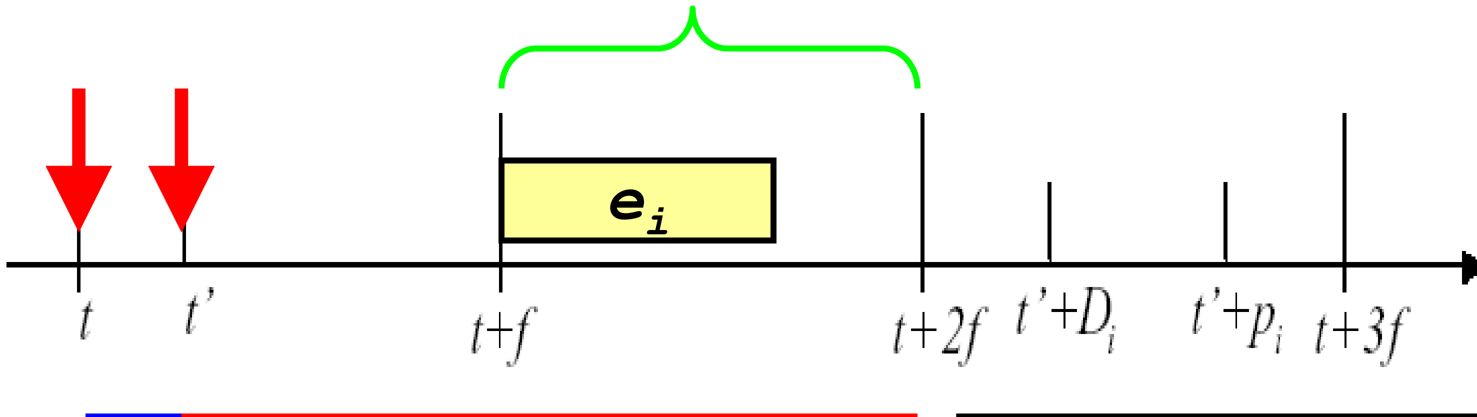
Scheduling Time Constraint

- Since scheduling decisions are made at the beginning of each frame, and periodic jobs are not preempted within a frame, to make it possible for the scheduler to determine if a job completes by its deadline, there should be at least one frame between the release and deadline of each job:

$$\boxed{2f - \mathrm{gcd}(p_i, f) \le D_i \text{ for each } 1 \le i \le n}$$

Scheduling Time Constraint

For monitoring purposes, frames must be sufficiently small that between release time and deadline of every job there is at least one frame:



$$2f - (t' - t) \le D_i$$

$$t' - t \ge \gcd(p_i, f)$$

$$(3) \qquad 2f - \gcd(p_i, f) \le D_i$$

Two Cases:

- a job J_i arrives at time t, or
- a job J_i arrives between time t and time $t+f$

Example #1

Task τ_i	Period p_i	Deadline D_i	Run-Time C_i
τ_1	4	4	1
τ_2	5	5	1.8
τ_3	20	20	1
τ_4	20	20	2

- **Hyperperiod = 20**
- **First Constraint => *f* is at least 2.**
- **Second Constraint => *f* divides 20; so, *f* is 2, 4, 5, 10, or 20.**
- **Third Constraint => *f* is 2.**

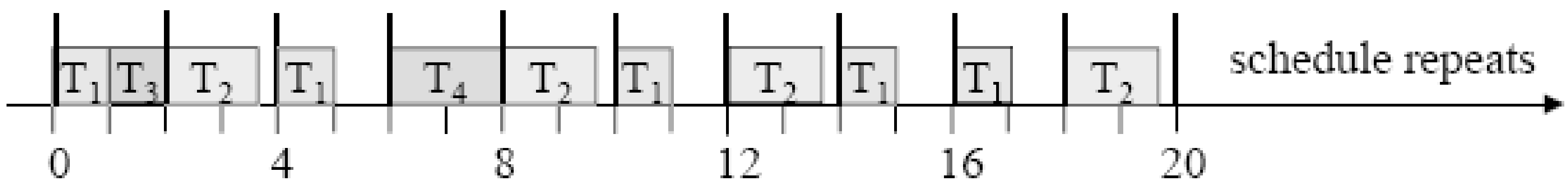
Example

Consider a system of four tasks, $T_1 = (4, 1)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1)$ $T_4 = (20, 2)$.

By first constraint, $f \geq 2$.

Hyperperiod is 20, so by second constraint, possible choices for f are 2, 4, 5, 10, and 20.

Only $f = 2$ satisfies the third constraint. The following is a possible cyclic schedule.



Example #2

Task τ_i	Period T_i	Deadline D_i	Run-Time C_i
τ_1	15	14	1
τ_2	20	26	2
τ_3	22	22	3

- **First Constraint => *f* is at least 3.**
- **Second Constraint => *f* is 3, 4, 5, 10, 11, 15, 20 or 22.**
- **Third Constraint => *f* is 3, 4, or 5.**

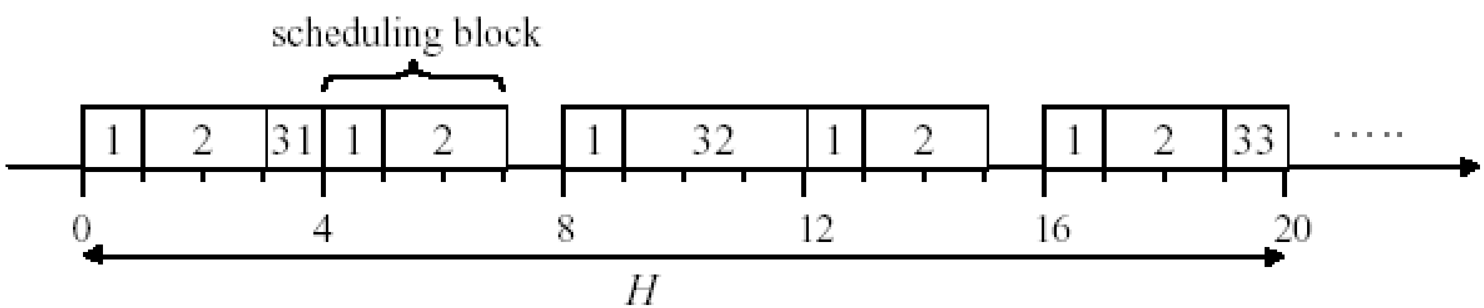
Slicing and Scheduling Blocks

- Slicing

$$
\begin{array}{llllll}
 & & p_i & e_i & D_i & \\
T_1 & = (& 4, & 1, & 4 &) \\
T_2 & = (& 5, & 2, & 5 &) \\
T_3 & = (& 20, & 5, & 20 &)
\end{array}
\qquad
\left.\begin{array}{lll}
(1) & \Rightarrow & f \geq 5 \\
(3) & \Rightarrow & f \leq 4
\end{array}\right\} ?!
$$

slice T_3

$$
\begin{array}{llllll}
T_1 & = (& 4, & 1, & 4 &) \\
T_2 & = (& 5, & 2, & 5 &) \\
T_{31} & = (& 20, & 1, & 20 &) \\
T_{32} & = (& 20, & 3, & 20 &) \\
T_{33} & = (& 20, & 1, & 20 &)
\end{array}
\qquad
\left.\begin{array}{lll}
(1) & \Rightarrow & f \geq 3 \\
(3) & \Rightarrow & f \leq 4
\end{array}\right\} f = 4
$$



Cyclic Executives

- *L(k)* = list of periodic jobs to be scheduled in the k^{th} **scheduling block**.
- *F* = number of frames per major cycle = *H/f*.
- *f* = minor frame size (in fixed time units; e.g., msec.)

Cyclic Executive

Input: Stored schedule: L(k) for k = 0,1,…,F-1;
Aperiodic job queue.

```
TASK CYCLIC_EXECUTIVE:
  k = 0;    /* current frame */
  BEGIN LOOP
    accept clock interrupt at time k*f;
    IF <the last job is not completed> take action;
    CurrentBlock := L(k);
    k            := k+1 mod F;
    IF <any slice in CurrentBlock is not released> take action;
    WHILE <CurrentBlock is not empty>
      execute the first slice in it;
      remove the first slice from CurrentBlock;
    END WHILE;
    WHILE <the aperiodic job queue is not empty>
      execute the first job in the queue;
      remove the just completed job;
    END WHILE;
  END LOOP;
END CYCLIC_EXECUTIVE;
```

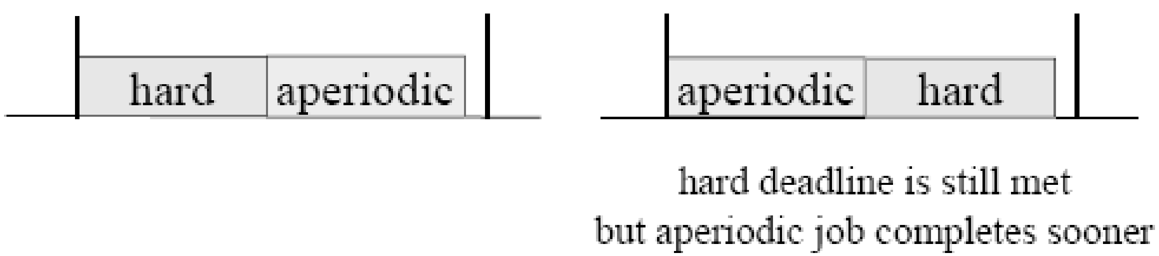
just a procedure call

Summary of Design Decisions

- Choose an appropriate frame size
- Partition jobs into slices (if necessary)
- Place slices into frames

Improving Response Times for Aperdiodic Jobs

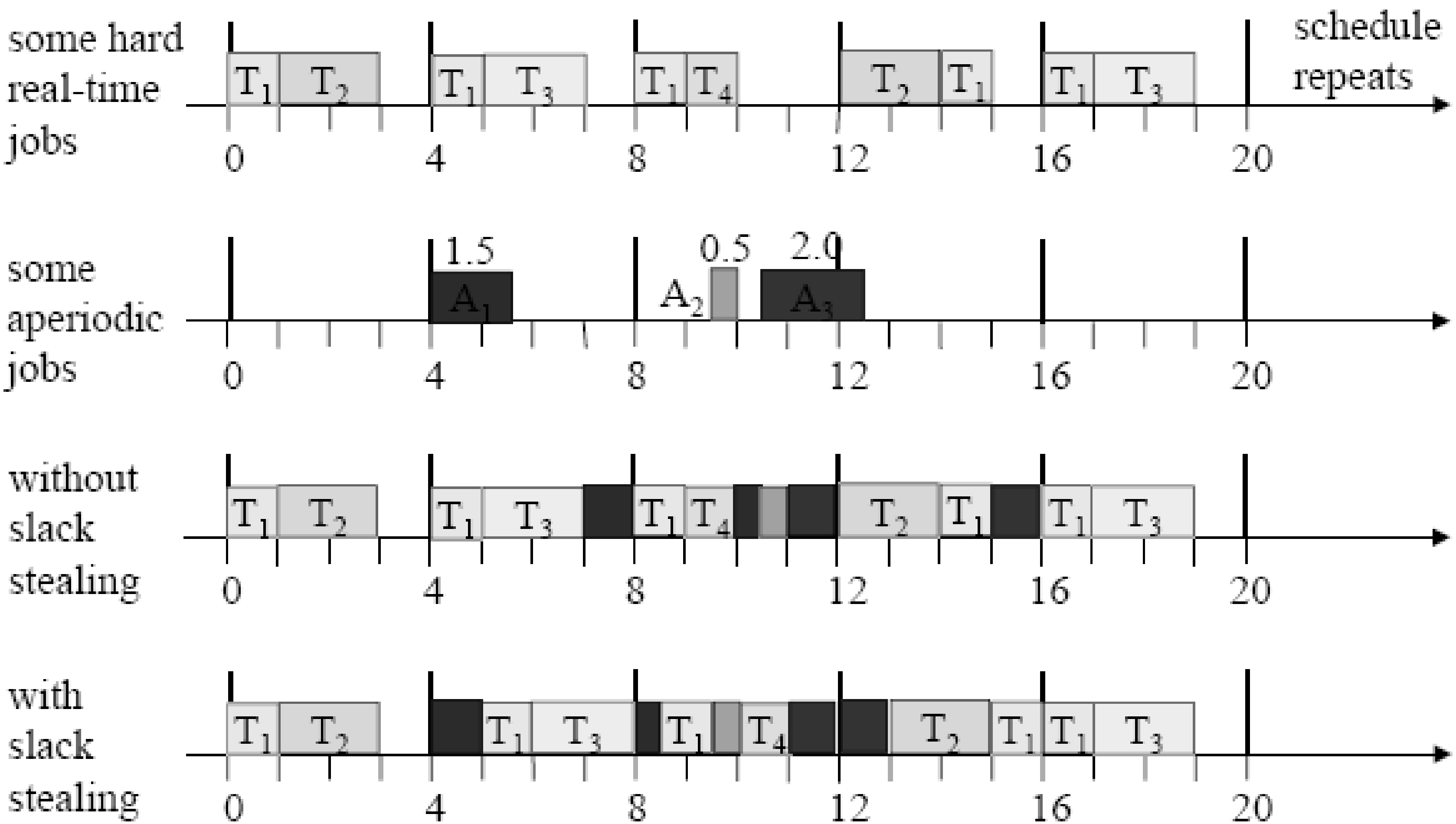
- Intuitively it makes sense to schedule the hard real-time periodic tasks first.
- However, this may lengthen the response time of aperiodic jobs, and there is no point in scheduling a hard real-time job first as long as it completes by its deadline.



Slack Stealing

- Let the total amount of time allocated to slices scheduled in frame k be $\boldsymbol{x_k}$.
- ***Def.*** The **slack time** or **slack** available at the beginning of frame k is $\boldsymbol{f - x_k}$.
- Change to scheduler: If the aperiodic job queue is non-empty and there is non-zero slack time, then schedule the aperiodic job at the front of the queue.

Example



some hard real-time jobs
schedule repeats
T1 T2 T1 T3 T1 T4 T2 T1 T1 T3
0 4 8 12 16 20
some aperiodic jobs
1.5 0.5 2.0
A1 A2 A3
0 4 8 12 16 20
without slack stealing
T1 T2 T1 T3 T1 T4 T2 T1 T1 T3
0 4 8 12 16 20
with slack stealing
T1 T2 T1 T3 T1 T4 T2 T1 T1 T3
0 4 8 12 16 20

Static Cyclic Scheduling

- Jobs in a periodic task are statically assigned to fixed time intervals in a cycle.
- A single **major cycle** can be divided into several smaller **minor cycles** of equal length.
- Given a task set and possible major and minor cycle lengths, **maximum network flow algorithms** can be used to determine if a feasible cyclic schedule exists.

Network Flow Algorithm for Computing Static Schedules

- Initialization: Compute all possible frame sizes based on the second two constraints:

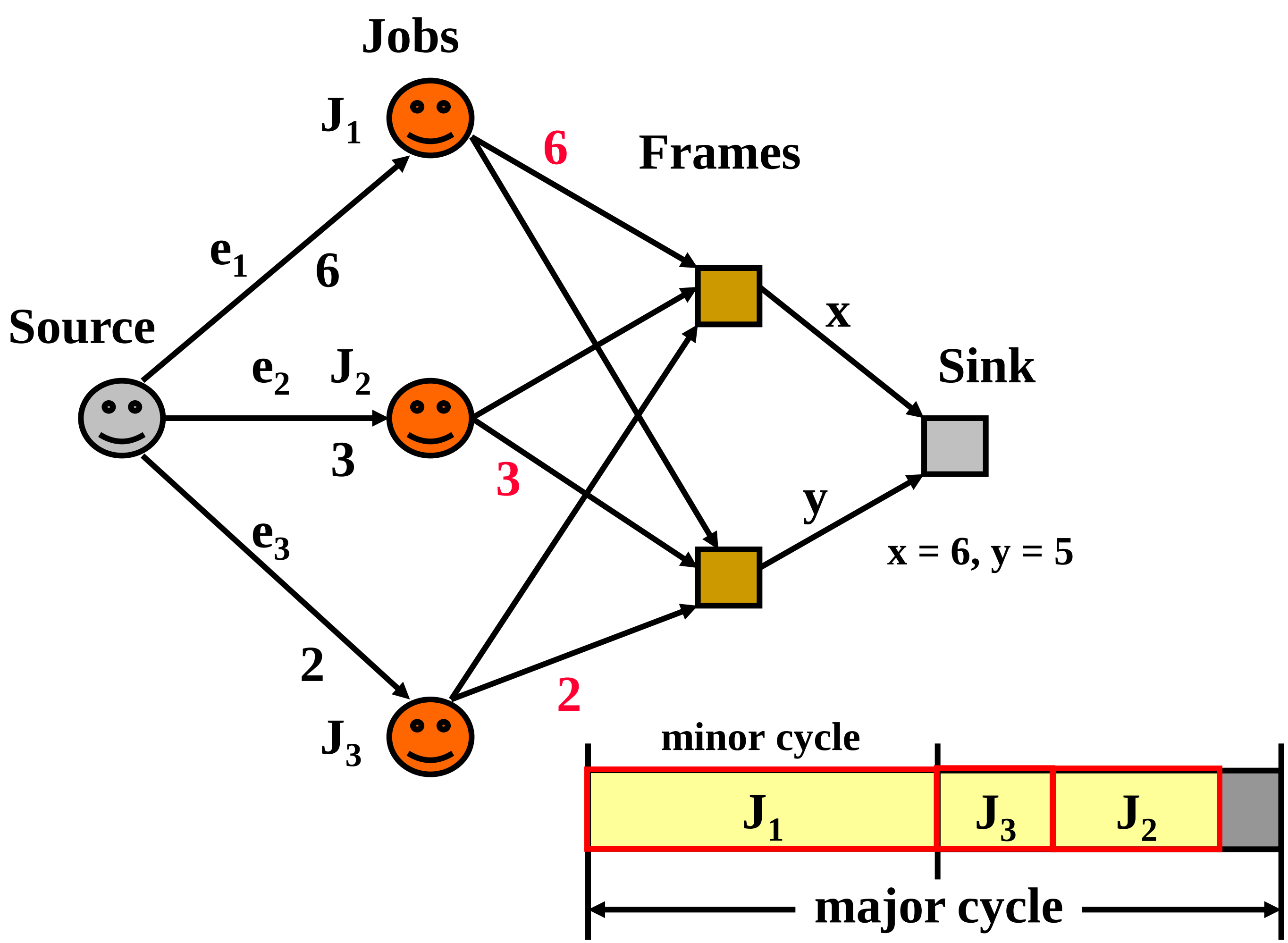
$$\lfloor p_i/f \rfloor - p_i/f = 0 \qquad\qquad 2f - \gcd(p_i, f) \le D_i$$

- Thus, we may need to slice a task into subtasks; e.g., start with the largest computed frame size to minimize slicing.
- For each possible frame size, construct a **flow graph** and run a max-flow algorithm to see if all tasks can be scheduled.

Flow Graph

- One vertex for each job in hyperperiod.
- One vertex for each frame (minor cycle).
- One source and one sink node.
- An edge from source to each job J_i vertex with weight equal to the job's run-time e_i. The maximum attainable flow is the **sum of the run-times**.
- An edge from each job vertex to each frame vertex with the edge weight equal to the amount that can be scheduled in each frame.
- An edge from each frame vertex to the sink with edge weight equal to the frame size.

Example Flow Graph – Max. Flow = 11



Maximum Network Flow Algorithms

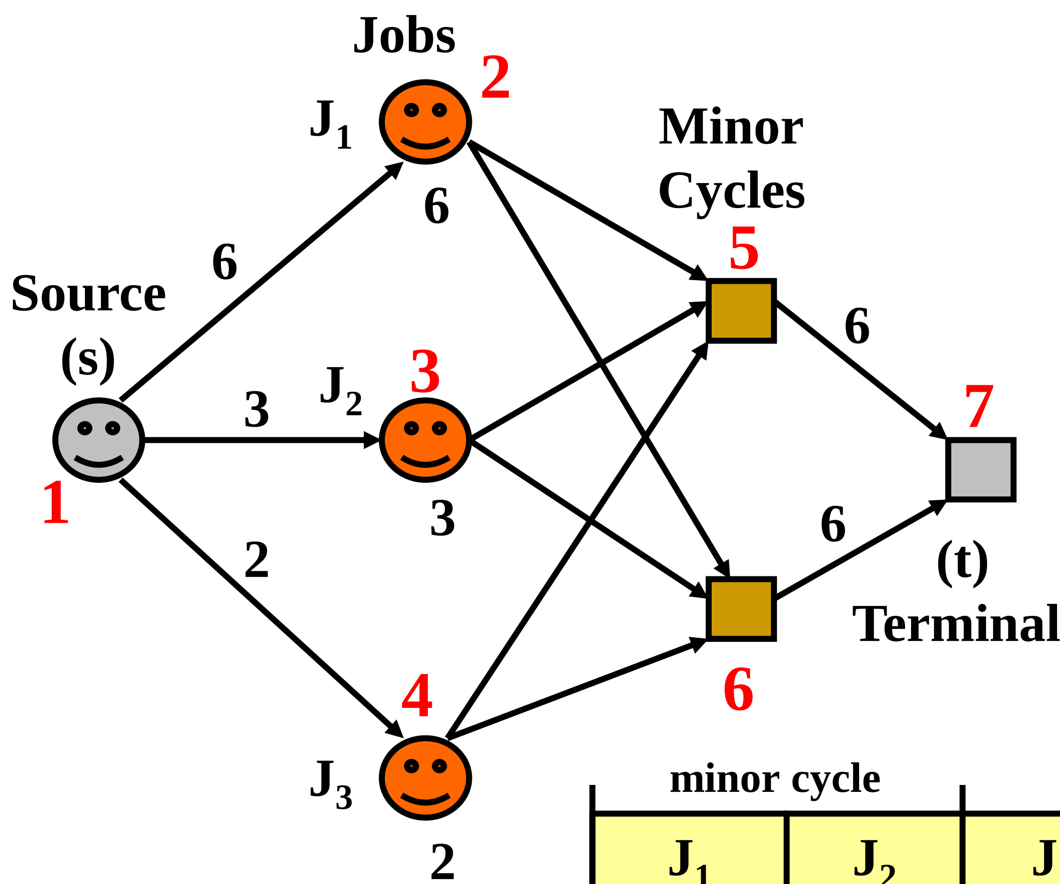
- **Ford Fulkerson** Algorithm
- **PRF** Algorithm = Push-Relabel method for max. Flow/min. cut problems, Goldberg, et al., *Algorithmica*, Vol. 19 (1997), pp. 390-410.
- Andrew Goldberg's software library: http://avglab.com/andrew/soft.html
- HIPR = High-Level Variant of PRF.

Static Cyclic Clock-Driven Scheduling

$ hi_pr < example1.input > example1.output

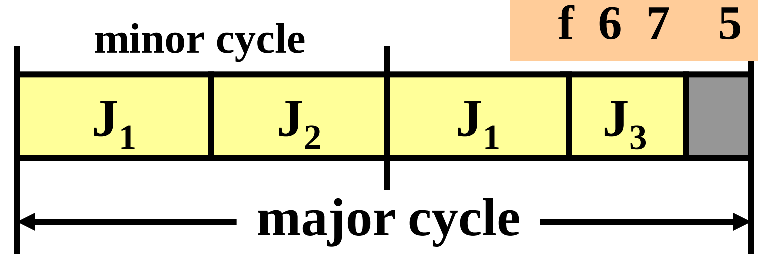
INPUT:

```
p max 7 11
n 1 s
n 7 t
a 1 2 6
a 1 3 3
a 1 4 2
a 2 5 6
a 2 6 6
a 3 5 6
a 3 6 6
a 4 5 6
a 4 6 6
a 5 7 6
a 6 7 6
```



OUTPUT:

```
max flow:11
…
f 1 2 6
f 1 4 2
f 1 3 3
f 2 5 3
f 2 6 3
f 3 5 3
f 3 6 0
f 4 6 2
f 4 5 0
f 5 7 6
f 6 7 5
```

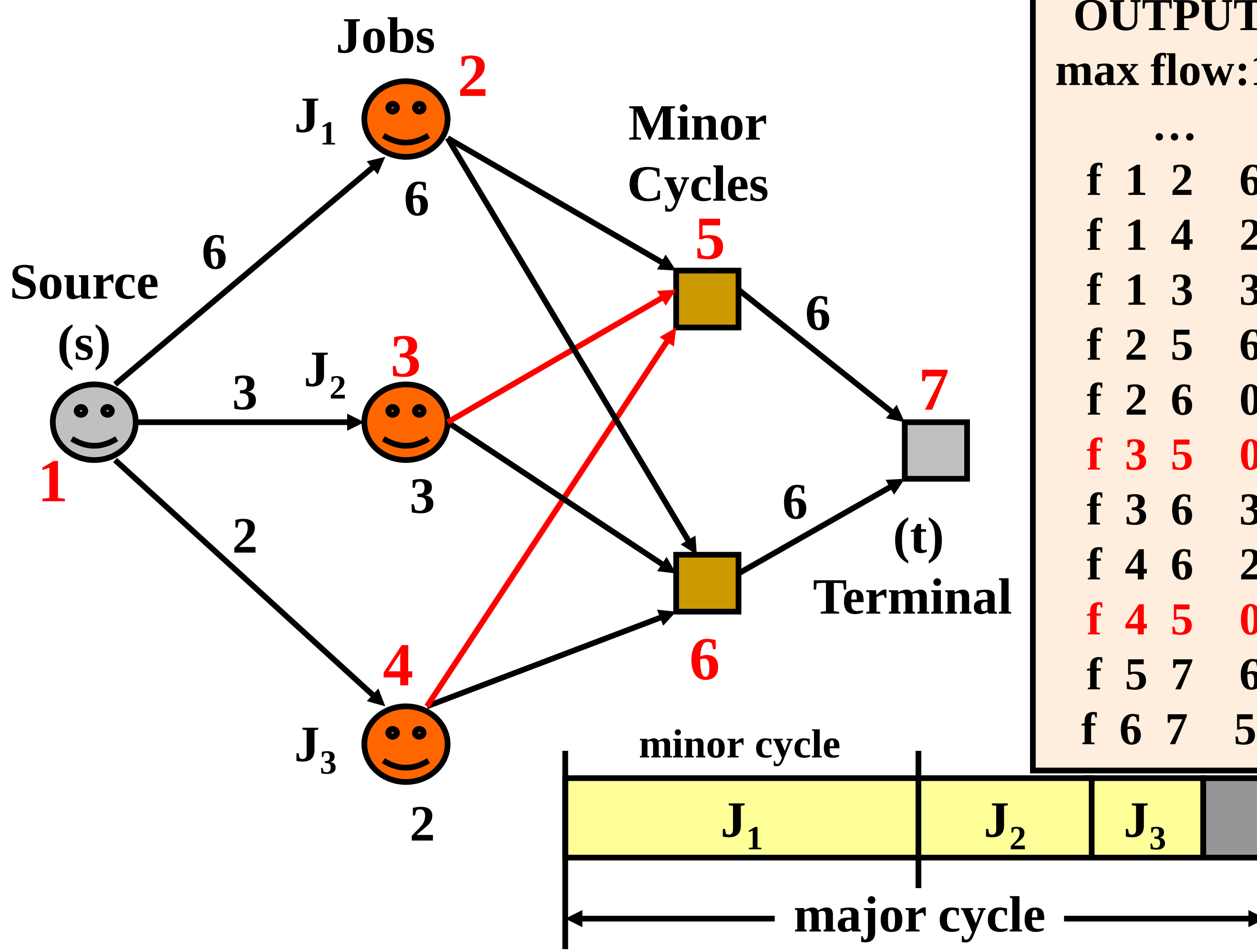


Static Cyclic Clock-Driven Scheduling

$ hi_pr < example2.input > example2.output

INPUT:

```
p max 7 11
n 1 s
n 7 t
a 1 2  6
a 1 3  3
a 1 4  2
a 2 5  6
a 2 6  6
a 3 5  0
a 3 6  6
a 4 5  0
a 4 6  6
a 5 7  6
a 6 7  6
```



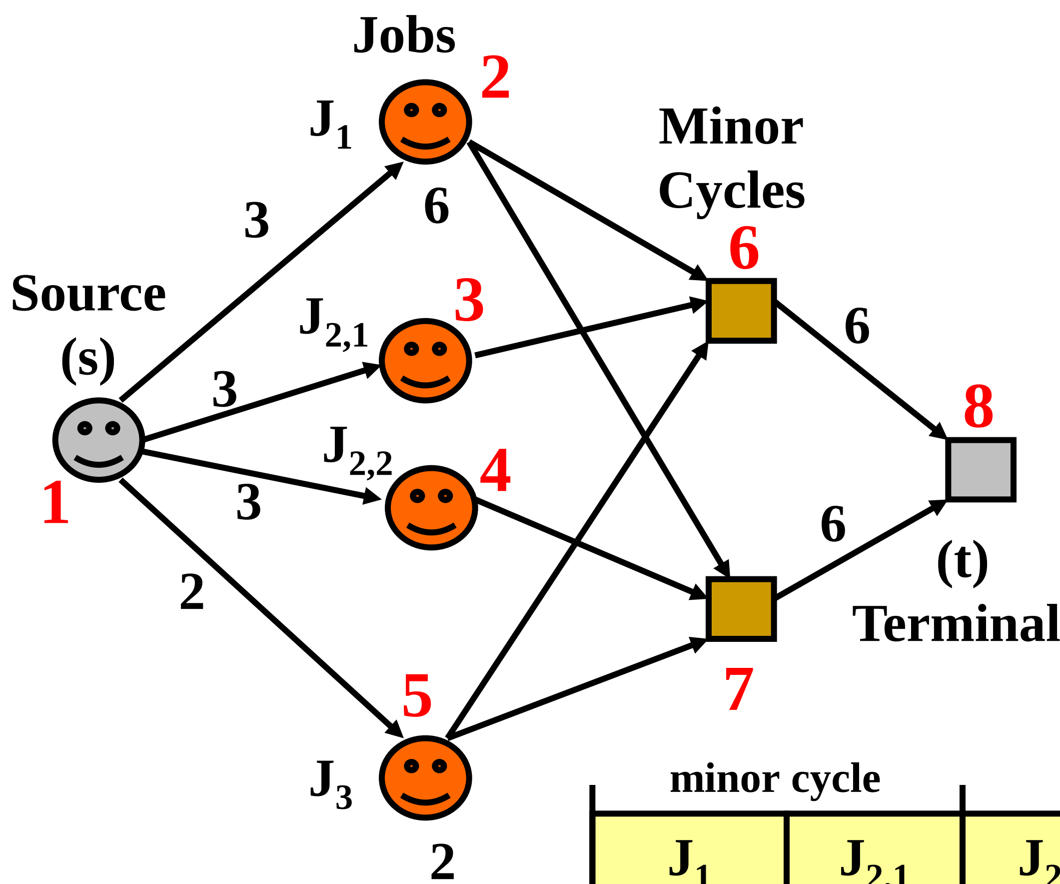
OUTPUT:

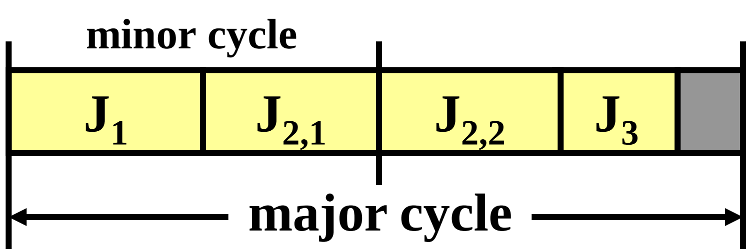
```
max flow:11
…
f 1 2  6
f 1 4  2
f 1 3  3
f 2 5  6
f 2 6  0
f 3 5  0
f 3 6  3
f 4 6  2
f 4 5  0
f 5 7  6
f 6 7  5
```

Clock-Driven Example

$ hi_pr < cyclic2.input > cyclic2.output

INPUT:
p max 8 12
n 1 s
n 8 t
a 1 2 3
a 1 3 3
a 1 4 3
a 1 5 2
a 2 6 6
a 2 7 6
a 3 6 6
a 4 7 6
a 5 6 6
a 5 7 6
a 6 8 6
a 7 8 6



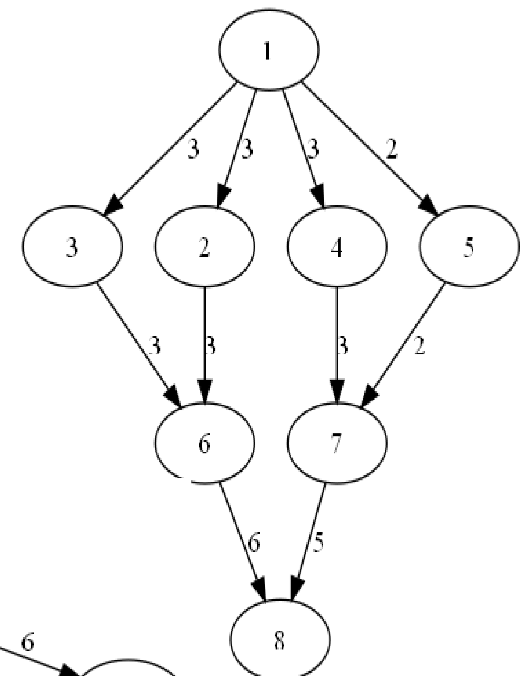


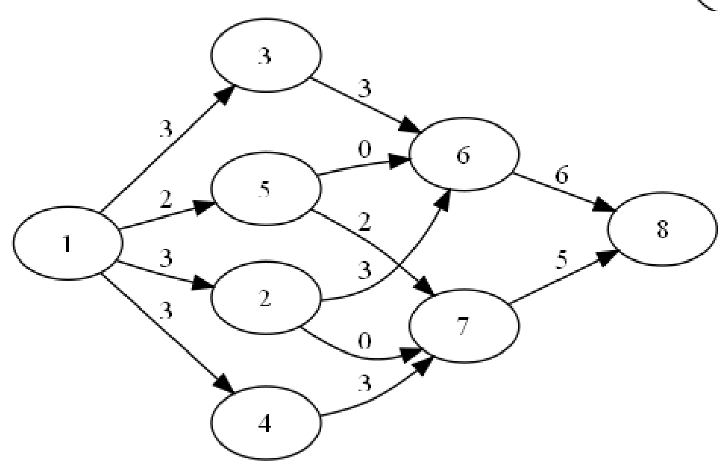
OUTPUT:
max flow:11
c flow values
f 1 2 3
f 1 4 3
f 1 3 3
f 1 5 2
f 2 6 3
f 2 7 0
f 3 6 3
f 4 7 3
f 5 7 2
f 5 6 0
f 6 8 6
f 7 8 5
c

Resulting Graphics File

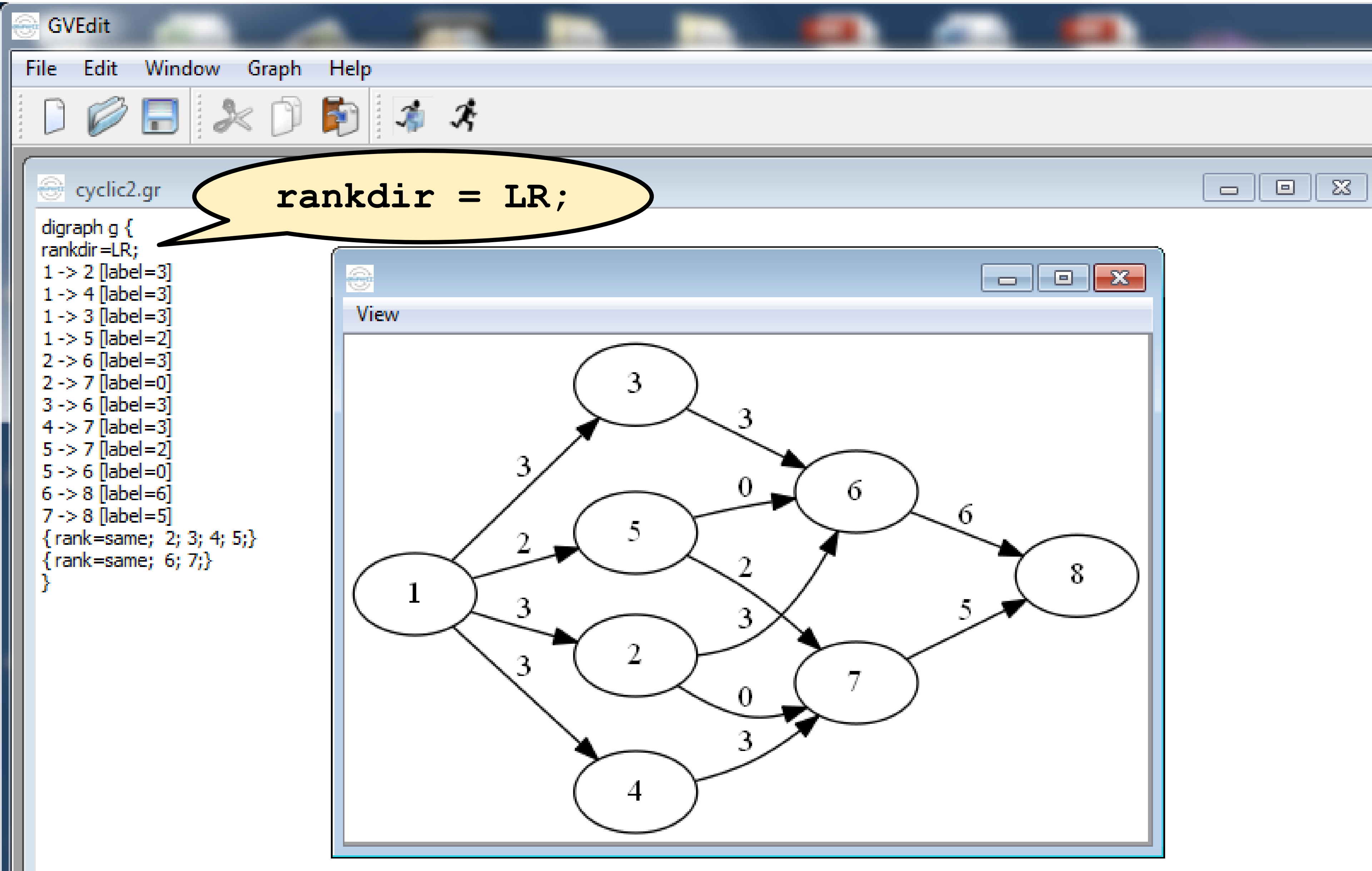
```
digraph g {
1 -> 2 [label=3]
1 -> 4 [label=3]
1 -> 3 [label=3]
1 -> 5 [label=2]
2 -> 6 [label=3]
3 -> 6 [label=3]
4 -> 7 [label=3]
5 -> 7 [label=2]
6 -> 8 [label=6]
7 -> 8 [label=5]
{ rank=same;  2; 3; 4; 5;}
{ rank=same;  6; 7;}
}
```

`dot -Tpng fn.gr -o fn.png`

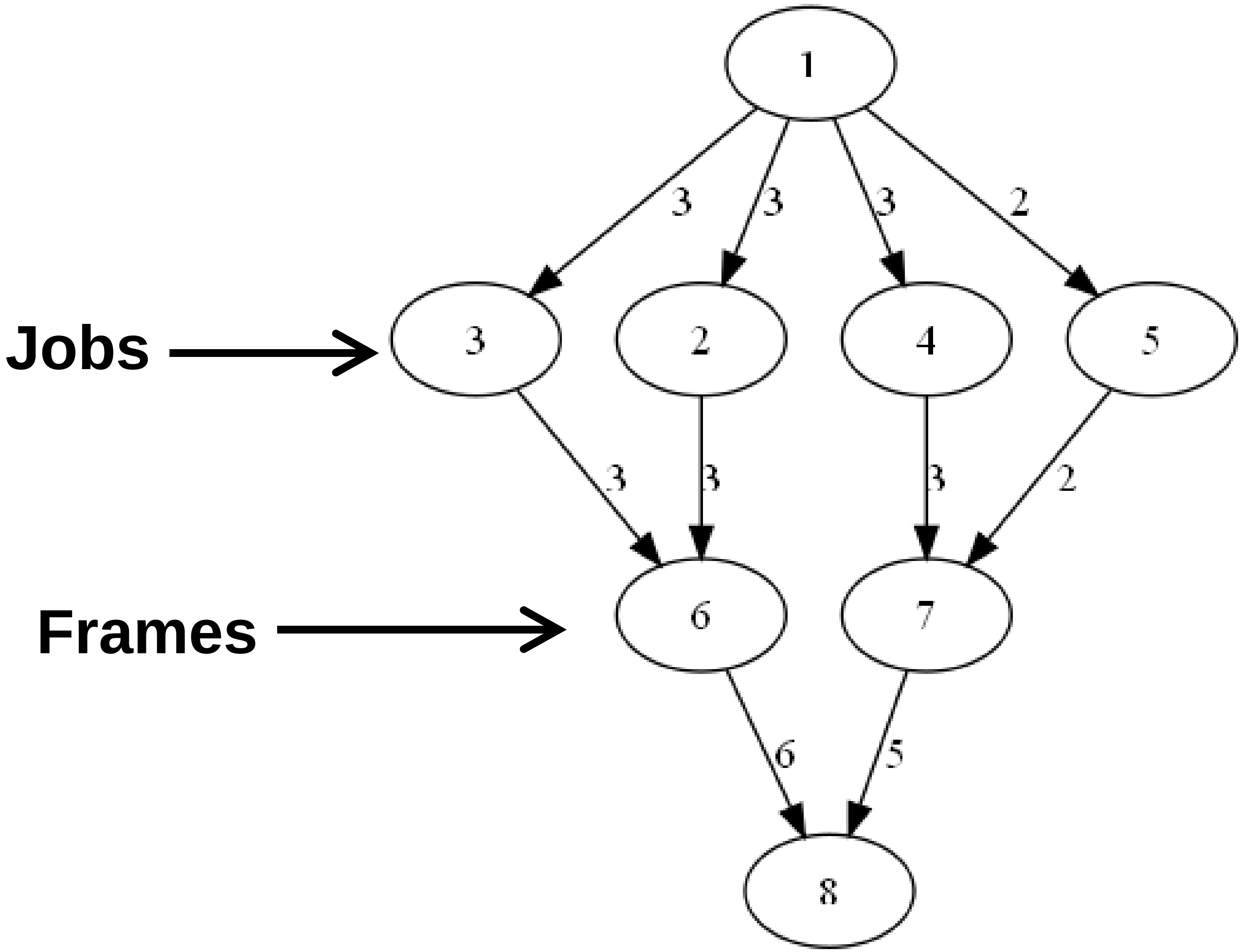




GraphViz Editor: GVEdit



Output generated



Example #3

Task τ_i	Period p_i	Deadline D_i	Run-Time C_i
τ_1	4	4	1
τ_2	5	5	1.8
τ_3	20	20	1
τ_4	20	20	2

- **Hyperperiod = 20**
- **First Constraint => *f* is at least 2.**
- **Second Constraint => *f* divides 20; so, *f* is 2, 4, 5, 10, or 20.**
- **Third Constraint => *f* is 2.**

Example #3 – Scaled (x10)

Task τ_i	Period p_i	Deadline D_i	Run-Time C_i
τ_1	40	40	10
τ_2	50	50	18
τ_3	200	200	10
τ_4	200	200	20

- **Hyperperiod = 200**
- **First Constraint => *f* is at least 20.**
- **Second Constraint => *f* divides 200; so, *f* is 20, 40, ..**
- **Third Constraint => *f* is 20.**

Example #3 – Scaled (x10)

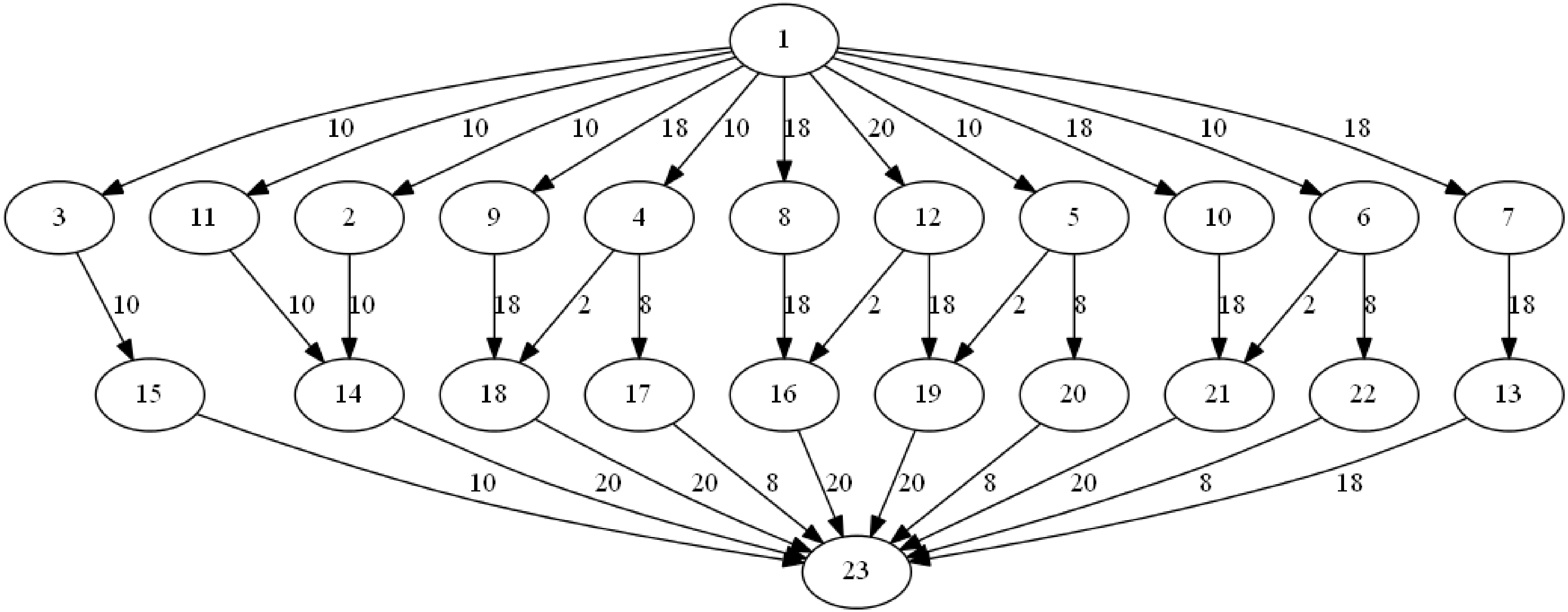
Task τ_i	Period p_i	Deadline D_i	Run-Time C_i	Number of jobs
τ_1	40	40	10	200/40 = 5
τ_2	50	50	18	200/50 = 4
τ_3	200	200	10	200/200 = 1
τ_4	200	200	20	200/200 = 1

- Hyperperiod = 200, in one hyperperiod, 5+4+1+1 = 11 jobs.
- Number of frames/hyperperiod = 200/20 = 10 frames.
- Frame size = $f = 20$.
- Network flow graph has 1 + 11 + 10 + 1 = 23 nodes (vertices), and 11 + 5*2 + 4*2 (or 3) + 1*10 + 1*10 + 10 = 59 edges (arcs).
- Max. possible flow = 5*10+4*18+10+20 = 152.

Example #3 – Schedule Generated

Frame	Jobs (time)	Frame Node
1	$\mathbf{J_{2,1}}$ **(18)**	13
2	$J_{1,1}$ (10), $J_{3,1}$ (10)	14
3	$J_{1,2}$ (10)	15
4	$\mathbf{J_{2,2}}$ **(18)**, $\mathbf{J_{4,1}}$ **(2)**	16
5	$J_{1,3}$ (8)	17
6	$J_{1,3}$ (2), $\mathbf{J_{2,3}}$ **(18)**	18
7	$J_{1,4}$ (2), $\mathbf{J_{4,1}}$ **(18)**	19
8	$J_{1,4}$ (8)	20
9	$J_{1,5}$ (2), $\mathbf{J_{2,4}}$ **(18)**	21
10	$J_{1,5}$ (8)	22

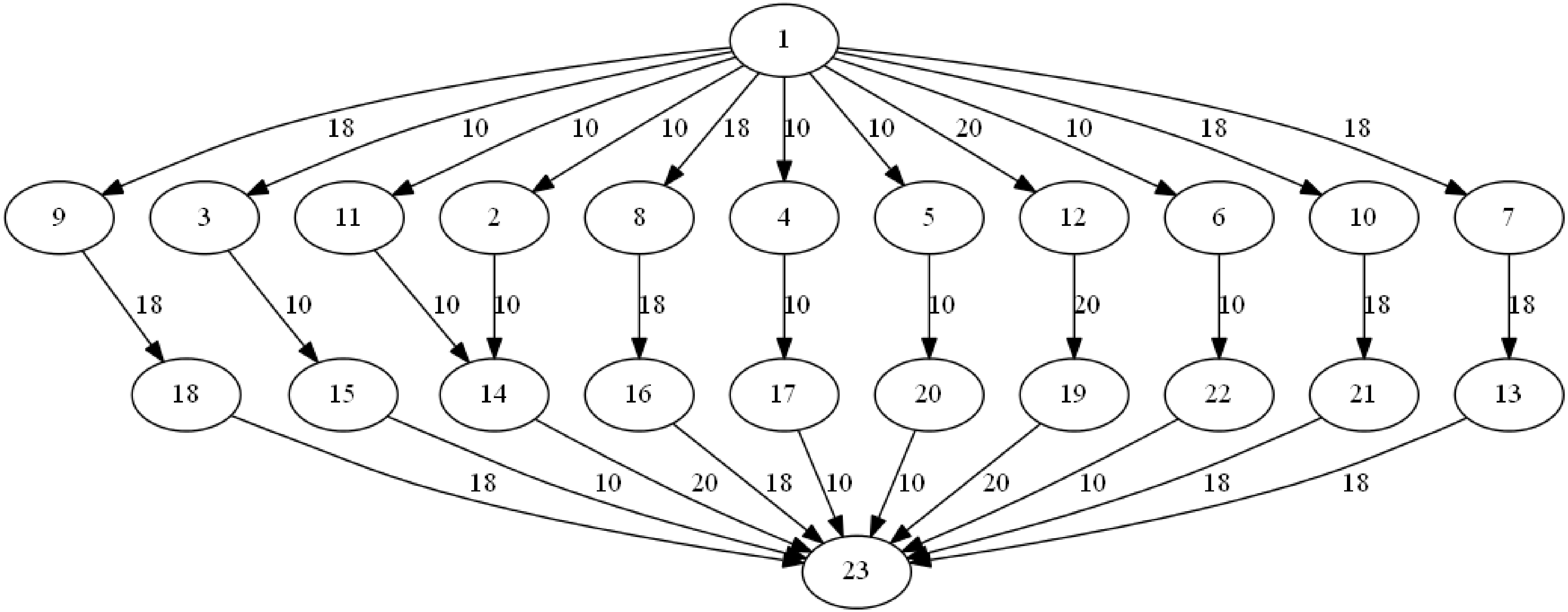
More Examples



Example #4 – Schedule Generated

Frame	Jobs (time)	Frame Node
1	**$J_{2,1}$ (18)**	13
2	$J_{1,1}$ (10), $J_{3,1}$ (10)	14
3	$J_{1,2}$ (10)	15
4	**$J_{2,2}$ (18)**	16
5	$J_{1,3}$ (10)	17
6	**$J_{2,3}$ (18)**	18
7	**$J_{4,1}$ (20)**	19
8	$J_{1,4}$ (10)	20
9	**$J_{2,4}$ (18)**	21
10	$J_{1,5}$ (10)	22

More Examples



Clock Driven Scheduling Example

Consider the following periodic task set:

1. (0, 500, 30.3671, 500)
2. (0, 500, 30.3671, 500)
3. (0, 2000, 30.1913, 2000)
4. (0, 2000, 50.1122, 2000)
5. (0, 6000, 400.823, 6000)

Set $f = 500$, $H = 6000$.

Then, there are 45 nodes: 1 source, 1 sink, 31 job nodes, 12 frame nodes, and 31+72 = 103 arcs, and max possible flow = 1370.5439.

File exampleInput.txt

- Five Tasks: (phase, period, run-time, deadline)
  - (0, 500, 30.3671, 500)
  - (0, 500, 30.3671, 500)
  - (0, 2000, 30.1913, 2000)
  - (0, 2000, 50.1122, 2000)
  - (0, 6000, 400.823, 6000)

Max Flow Input File exampleData.txt

- **Problem Max-Flow with 45 nodes and 103 arcs (edges):**

```
p max 45 103
n 1 s
n 45 t
a 1 2 303671
a 2 33 5000000
a 1 3 303671
a 3 34 5000000
a 1 4 303671
a 4 35 5000000
a 1 5 303671
a 5 36 5000000
a 1 6 303671
a 6 37 5000000
..
```

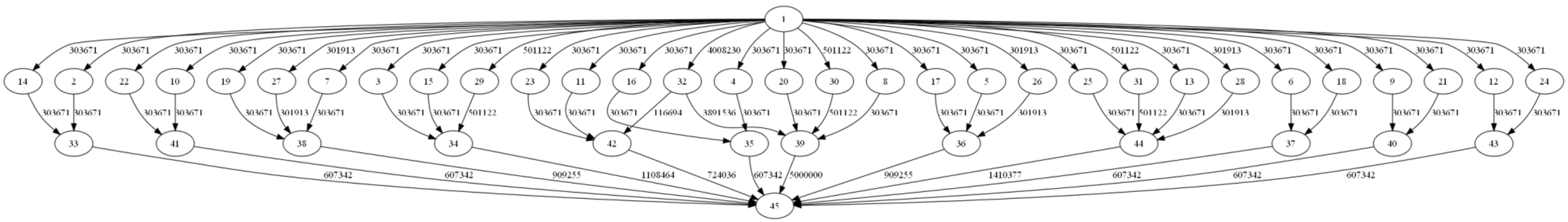
Max Flow Output File exampleData.out

```
c
c hi_pr version 3.6
c Copyright C by IG Systems, igsys@eclipse.net
c
c nodes:                  45
c arcs:                  103
c
c flow:         13705439.0
c
c Solution checks (feasible and optimal)
c
c pushes:                 96
c relabels:               32
c updates:                 1
c gaps:                    0
c gap nodes:               0
c
c flow values
f        1        2        303671
f        1       23        303671
f        1       10        303671
f        1       27        301913
f        1        3        303671
```

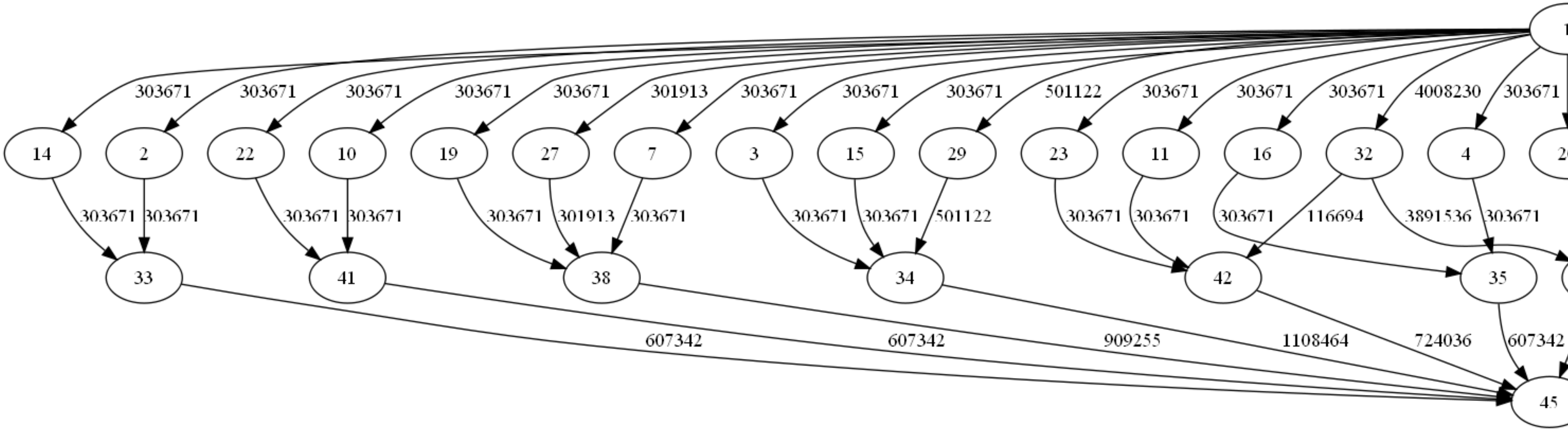
Graphics File exampleData.gr

```
digraph g {
1 -> 2 [label=303671]
1 -> 23 [label=303671]
1 -> 10 [label=303671]
1 -> 27 [label=301913]
1 -> 3 [label=303671]
1 -> 19 [label=303671]
1 -> 14 [label=303671]
1 -> 29 [label=501122]
1 -> 4 [label=303671]
1 -> 32 [label=4008230]
..
```

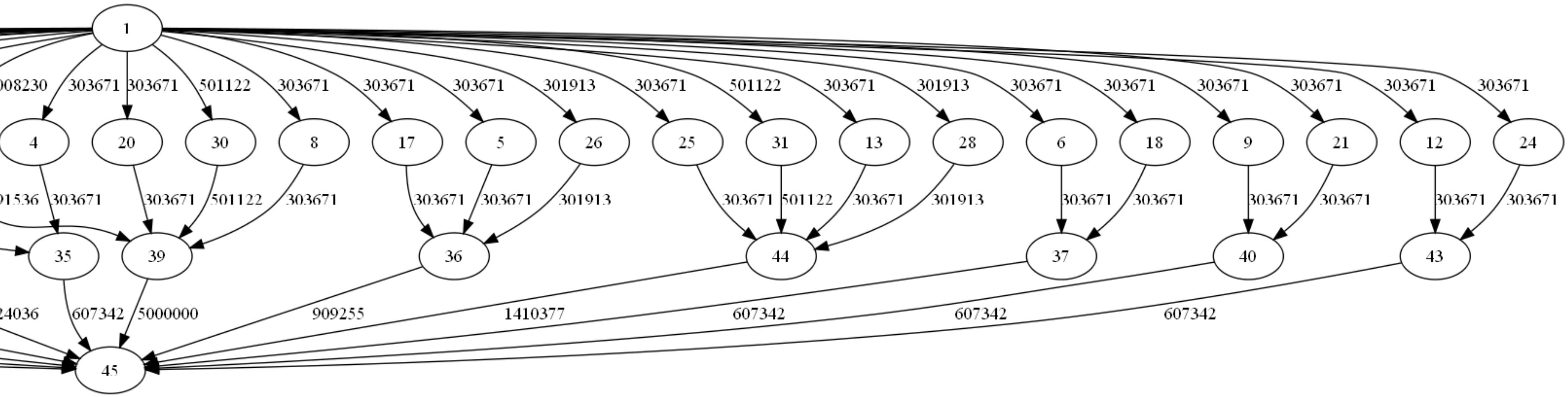
Output generated



Output generated



Output generated



Static Cyclic Scheduling Paper

N. Audsley, K. Tindell, and A. Burns, “The end of the line for static cyclic scheduling?”, In Proc. of the 21st Euromicro Conference, 1995.

- Shows how static priority assignments and priority based scheduling can be used in place of static cyclic scheduling.

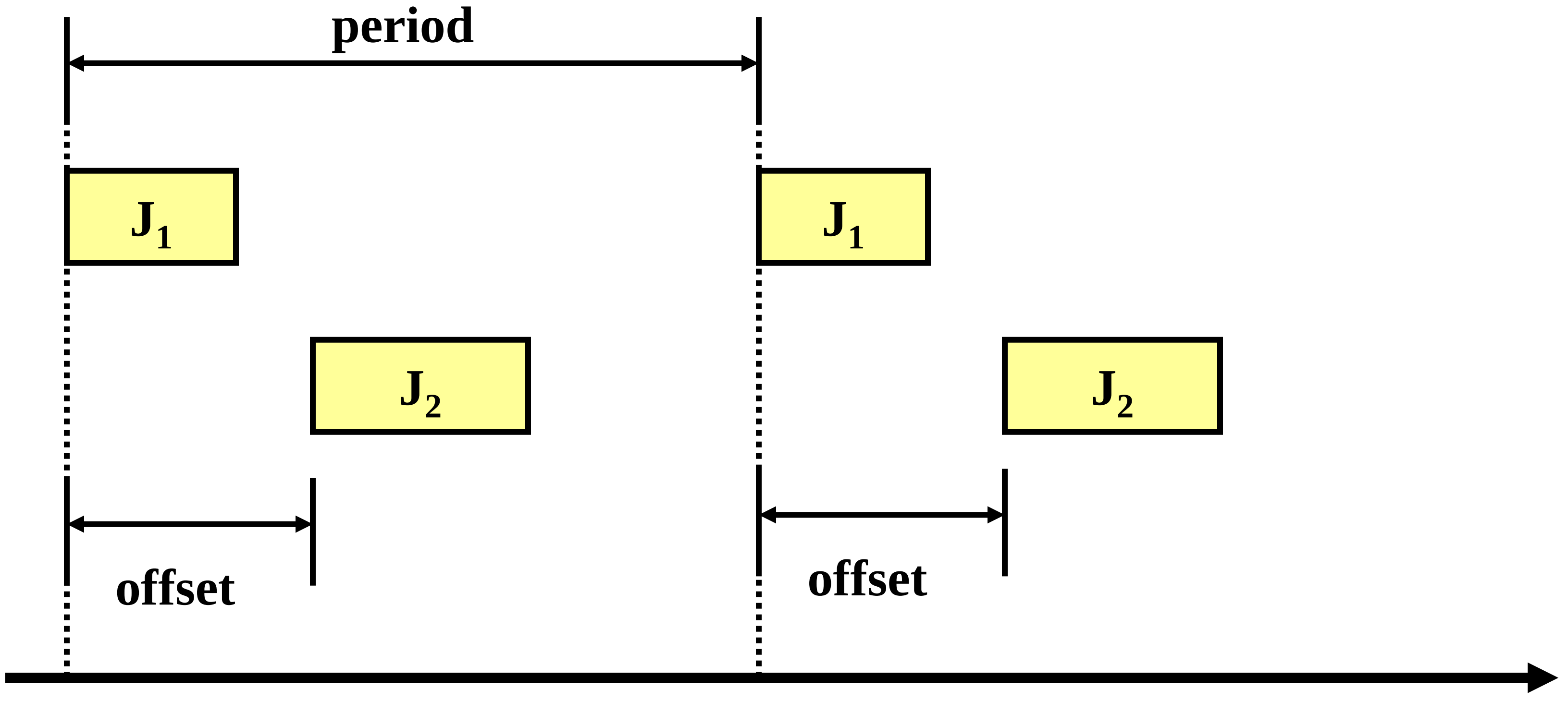
Computational Model

- A fixed number of transactions are assigned to each **processor**.
- A **transaction** consists of a fixed number of tasks.
- Each **task T_i** (or τ_i) requires a bounded amount of computation time e_i (or C_i).
- Transactions may arrive either **periodically** or **sporadically**, but there is a minimum amount of time between subsequent arrivals.

Task Model

- Tasks are **released** (put in a priority-ordered ready queue) at a fixed offset relative to transaction arrival time.
- Tasks are assigned **static priorities**.
- Task may have **arbitrary deadlines** and **release jitter**.

Example



Motivation for Offsets

- Precedence constraints can be modeled using offsets; e.g., J_1 must complete before J_2.
- Offsets can also be used to avoid the need for resource access control mechanisms; e.g., semaphores, etc.
- Offsets can be used to permit tight jitter bounds and express complex timing patterns; e.g., break the job into two parts -- input and output.

Optimal Priority Ordering

- Assign priorities from lowest to highest.
- Let 1 = highest priority (note, incorrectly listed as 0 in the paper).
- Let N = lowest priority and number of tasks.

Algorithm

```
ordered := N
repeat
    finished := false
    failed := true
    j := ordered
    repeat
        insert task at priority j (from unsorted list) into sorted list at priority ordered
        if task j is feasible then
            ordered := ordered - 1
            failed := false
            finished := true
        else
            remove j from sorted list and return to old priority - 1 in unsorted list
        end if
        j := j - 1
    until finished or j = 0
until ordered = 0 or failed
```

Time complexity: $O((N^2 + N) E)$ where E is the complexity of the feasibility test.

Observations

- At all times, the sorted list is schedulable.
- The sorted list increases in size until either
  - all tasks are schedulable, or
  - none of the top $n \leq N$ tasks are schedulable at priority n.
- Since decreasing the priority of a task cannot lead to a decrease in worst-case response time, if none of the top n tasks are schedulable at priority n, then no feasible priority assignment exists.

Priority vs. Cyclic Scheduling

- Unrelated strictly periodic tasks, with the same period, can be incorporated into the same transaction with offsets between tasks.
- Tasks with different periods can be transformed into tasks sharing the same period by choosing a **common period** smaller than the original periods, or by **adding multiple instances** of the same task with offsets between them.
- Note that individual instances can be assigned different priorities to improve the feasibility of a task set.

Precedence Constraints

- Precedence Constraints can be incorporated into the Priority Assignment Algorithm.
  - Task B is constrained to run only when Task A has finished.
  - Task A and Task B exclude each other.

Outline

- Commonly Used Approaches For Real-Time Scheduling (Ch. 4)
  - Clock-Driven Scheduling (Ch. 5)
  - **Priority-Driven Scheduling (Ch. 6-7)**
    - **Periodic Tasks (Ch. 6)**
    - Aperiodic or Sporadic Tasks (Ch. 7)

Temporal Parameters

- J_i : **job** – a unit of work
- T_j (or τ_i): **task** - a set of related jobs
- A **periodic task** is sequence of invocations of jobs with identical parameters.
- r_i: **release time** of job J_i
- d_i: **absolute deadline** of job J_i
- D_i: **relative deadline** (or just **deadline**) of job J_i
- e_i: (Maximum) **execution time** of job J_i

Periodic Task Model

- **Tasks:** T_1, ….., T_n
- Each consists of a set of **jobs**: $T_i = \{J_{i1}, J_{i2}, \ldots\ldots\}$
- ϕ_i: p**hase** of task T_i = time when its first job is released
- p_i: p**eriod** of T_i = minimum inter-release time
- H: h**yperperiod** $H = \mathrm{lcm}(p_1, \ldots.., p_n)$
- e_i: e**xecution time** of T_i
- u_i: **utilization** of task T_i is given by $u_i = e_i / p_i$
- D_i: (relative) **deadline** of T_i, typically $D_i = p_i$

Periodic Task

- We refer to a periodic task T_i with phase ϕ_i, period p_i, execution time e_i, and relative deadline D_i by the 4-tuple(ϕ_i, p_i, e_i, D_i).
- Example: (1, 10, 3, 6)
- By default, the phase of each task is 0, and its relative deadline is equal to its period.
- Example: (0, 10, 3, 10) = (10, 3).

Priority-Driven Scheduling Algorithms

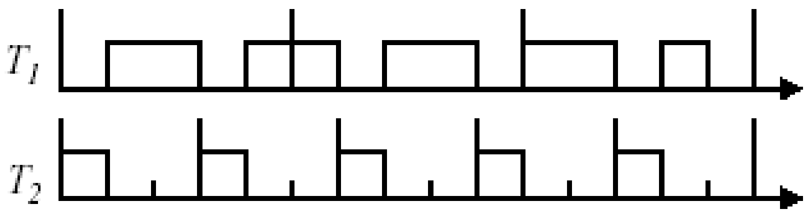
- **Static-(or Fixed-)Priority** - assigns the same priority to all jobs in a task.
- **Dynamic-Priority** – may assign different priorities to individual jobs within each task; e.g., earliest-deadline-first (EDF) algorithm, etc.

Static-Priority *vs.* Dynamic Priority

- **Static-Priority**: All jobs in task have same priority.
- example:

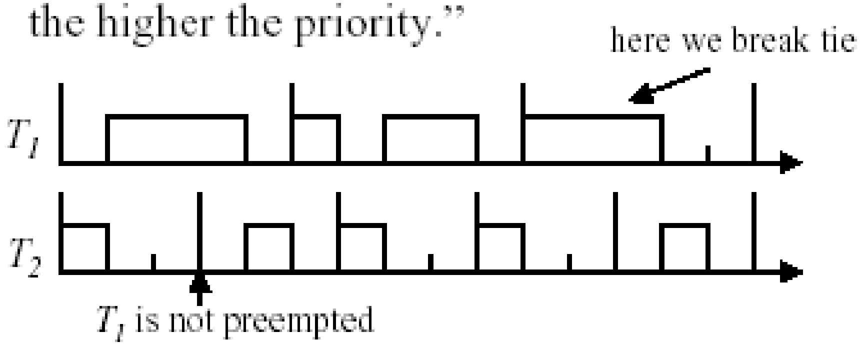
  Rate-Monotonic: "The shorter the period, the higher the priority."

$$T_1 = (\ 5,\ 3,\ 5\ )$$
$$T_2 = (\ 3,\ 1,\ 3\ )$$



- **Dynamic-Priority**: May assign different priorities to individual jobs.
- example:

  Earliest-Deadline-First: "The nearer the absolute deadline, the higher the priority."



Scheduler

- A **scheduler** assigns jobs to processors.
- A **schedule** is an assignment of all jobs in the system on available processors (produced by scheduler).
- The **execution time** (or **run-time**) of a job is the amount of time required to complete the execution of a job once it has been scheduled (e_i or C_i).
- A constraint imposed on the timing behavior of a job is called a **timing constraint**.

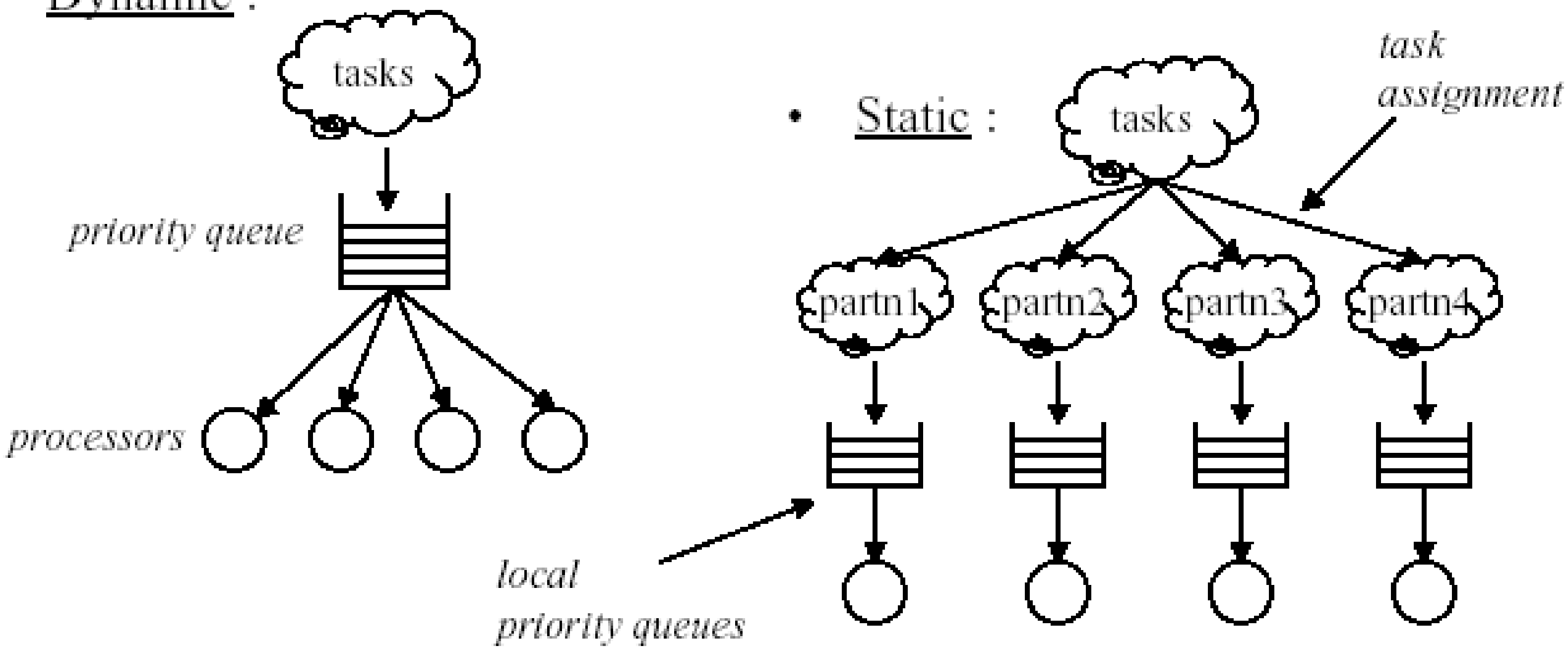
Scheduling of Periodic Tasks

- **Assumptions**
  - Tasks are independent
  - Preemption is allowed
  - All tasks are periodic
  - No sporadic or aperiodic tasks
  - Single processor

WHY A SINGLE PROCESSOR?

Why Focus on Uniprocessor Scheduling?

- Dynamic *vs*. static multiprocessor scheduling:
  - <u>Dynamic</u> :
  - <u>Static</u> :

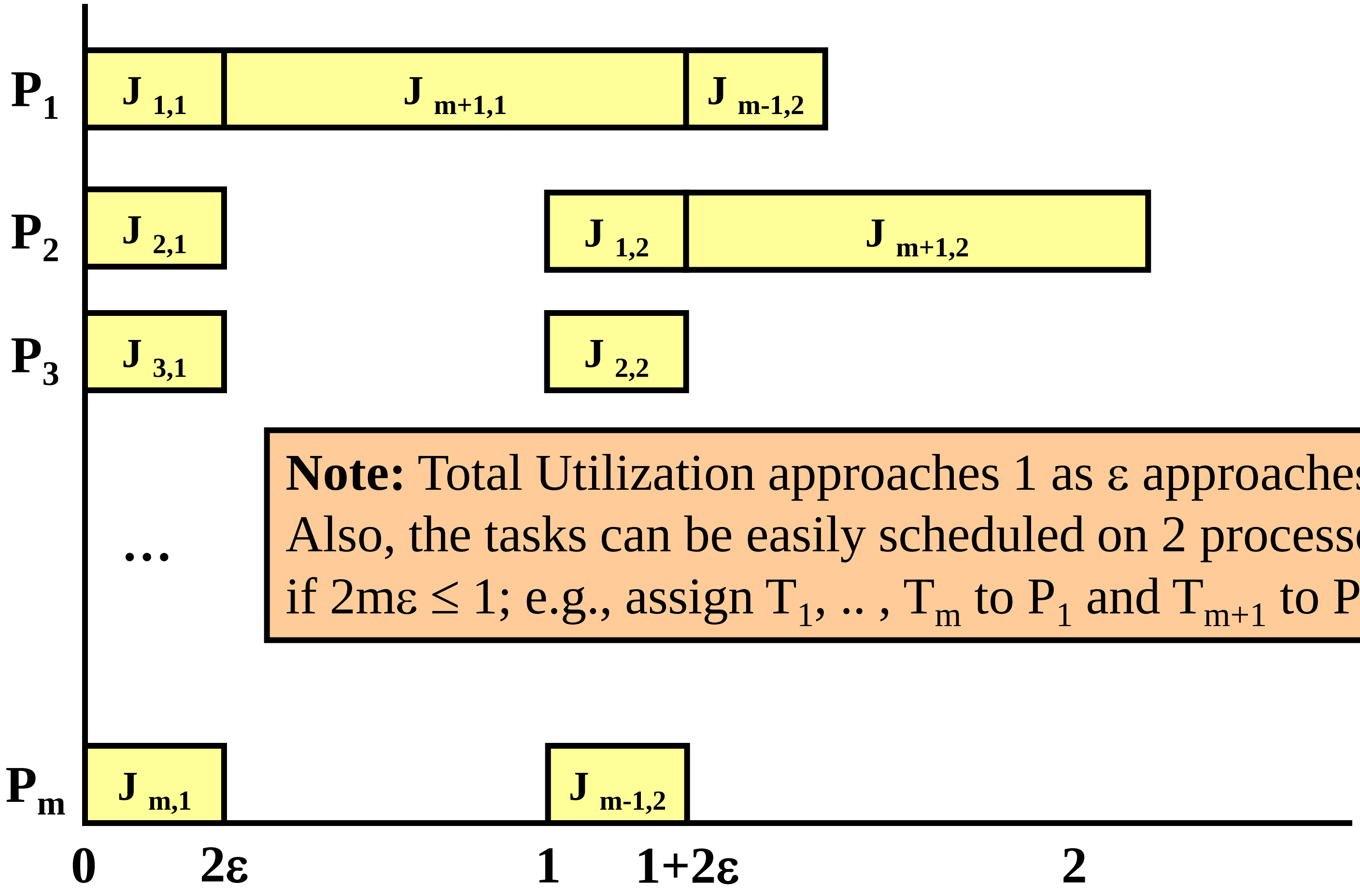


- Poor worst-case performance of priority-driven algorithms in dynamic environments.
- Difficulty in validating timing constraints.

Example

- Here is an example to show that the performance of priority-driven algorithms with **dynamic processor assignment** can be **very poor**:
  - Number of processors = m
  - Number of independent periodic tasks = m+1
  - Small Tasks T_1 .. T_m are identical with $p_i = 1$, $e_i = 2\varepsilon$ for some small number ε
  - Large Task T_{m+1} has $p_{m+1} = \varepsilon + 1$, $e_{m+1} = 1$
  - Relative deadlines are equal to periods (for all tasks).
  - Apply a dynamic EDF algorithm to schedule the tasks on m processors.

Example (cont.)



Note: Total Utilization approaches 1 as ε approaches 0. Also, the tasks can be easily scheduled on 2 processors, if $2m\varepsilon \leq 1$; e.g., assign $T_1, .. , T_m$ to P_1 and T_{m+1} to P_2 .

Static vs. Dynamic Systems

- The poor behavior of **dynamic** systems occurs only for these types of pathological systems, but the real problem is how to determine the **worst-case performance of dynamic systems**, other than by simulating and testing the system.
- Consequently, most hard real-time systems (for now and in the near future) are **static**. Well-grounded theories and algorithms can be used to validate efficiently, robustly, and accurately the timing constraints of static systems (as we shall see).
- Also, in a static system, uniprocessor algorithms can be easily **extended** to multiprocessor systems.

Summary

- Read Ch. 4-7.
- Homework #1.