Unit - 2

Real time Scheduling

Preemptive scheduling Vs non preemptive scheduling:-

In preemptive scheduling the scheduler allows process preemptions. In non preemptive scheduling the scheduler does not allow for preemption.

Online Vs offline scheduling - It allocates resources for processor depending upon the current load it's online scheduling. While offline, all allocation is done before scheduling.

Types of scheduling Algorithm:-

1. Clock driven approach
2. Weighted Round-Robin approach
3. Priority based approach

1) clock-driven approach:- (Also called time-driven) decisions on what jobs execute at what times are made at specific time instant.

→ clock-driven scheduling is a static scheduling because scheduling decisions are made or compiled offline.

→ ~~Regular~~ Scheduled is computed offline and is stored for use at run time.

a) All parameters of hard real-time jobs are fixed and known.

(b) scheduling overhead during run-time can be minimized.

→ Scheduler makes decisions using a hardware timer. The timer is set to expire periodically without the intervention of the scheduler.

(a) when the system is initialized, the scheduler selects and schedules the job that will execute until the next scheduling decision time and then blocks itself waiting for the expiration of the timer.

(b) when the timer expires, the scheduler awakes and repeats these actions.

Weighted Round-Robin approach:- It's used for scheduling time-shared application. When jobs are scheduled on a round-robin basis, every job joins a first-in-first-out (FIFO) queue when it becomes ready for execution.

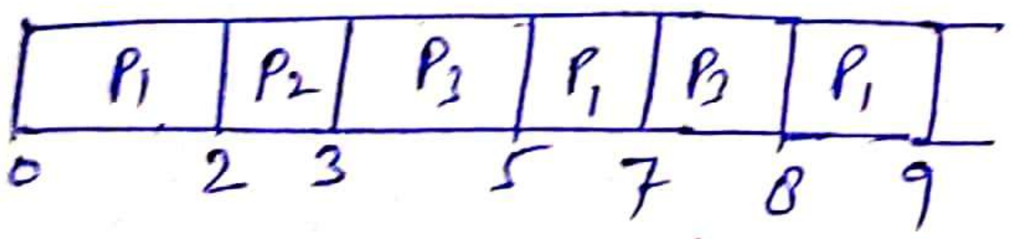
→ If the job does not complete by the end of the time slice, it is preempted and placed at the end of the queue to wait for its next turn.

In weighted Round Robin, Rather than giving all the ready jobs equal shares of the processor, different jobs may be given diffent weight. By adjusting the weights of Jobs we can speed up or retard the progress of each job towards its completion.

eg, Execution/Response Q = 2

P_1	5	3
P_2	1	0
P_3	3	0

— Gantt chart



— waiting Queue.

P_1	~~P_2~~	P_3	P_1	P_3	P_1

for exp.

New suppose deadline of $P_1 = 8$

Then WRR will be beneficial we can increase the weight of job J_1

Quantum time / weight

P_1	5	4
P_2	1	1
P_3	3	2

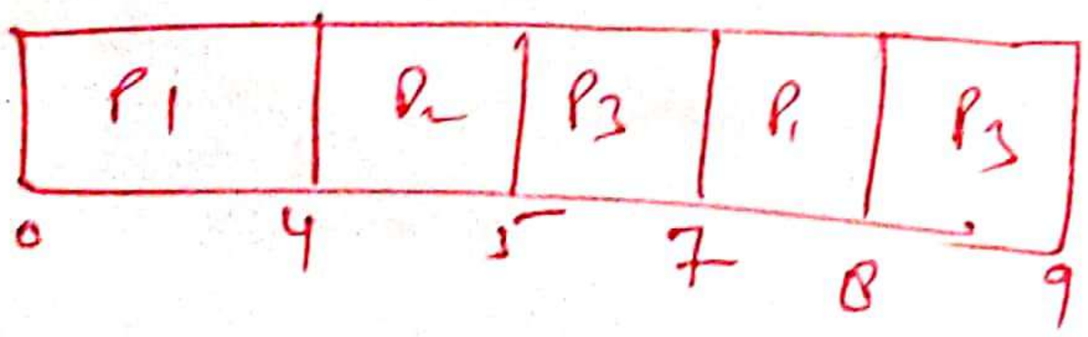
Suppose

P_1 —— 4

P_2 —— 1

P_3 —— 2

Here we check for w.r.t to P_1 only



more queues ordered by the priorities of the jobs. At any scheduling decision time, the jobs with the highest priorities are scheduled and executed on the available processors. Hence, a priority-driven scheduling algorithm is defined to a great extent by the list of priorities it assigns to jobs; the priority list and other rules, such as whether preemption is allowed, define the scheduling algorithm completely.

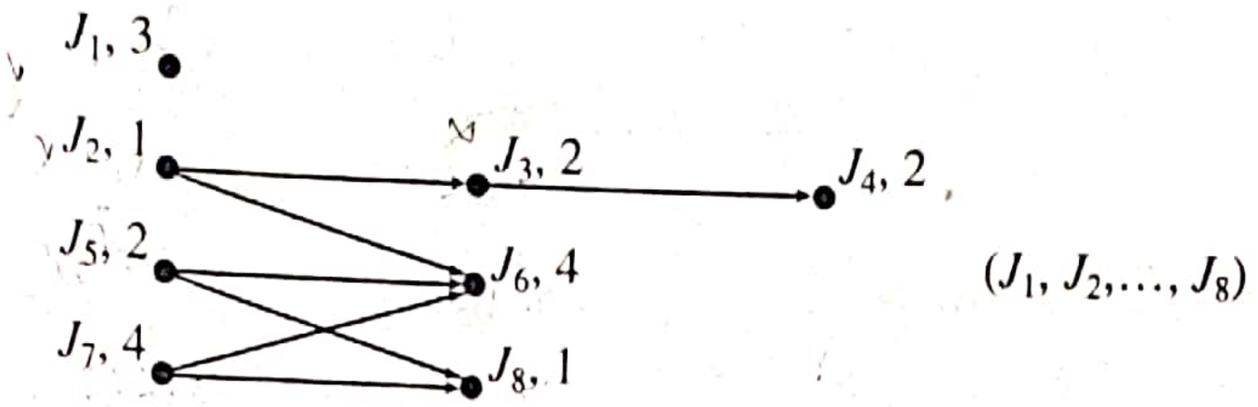
Most scheduling algorithms used in nonreal-time systems are priority-driven. Examples include the FIFO (First-In-First-Out) and LIFO (Last-In-First-Out) algorithms, which assign priorities to jobs according their release times, and the SETF (Shortest-Execution-Time-First) and LETF (Longest-Execution-Time-First) algorithms, which assign priorities on the basis of job execution times. Because we can dynamically change the priorities of jobs, even round-robin scheduling can be thought of as priority-driven: The priority of the executing job is lowered to the minimum among all jobs waiting for execution after the job has executed for a time slice.

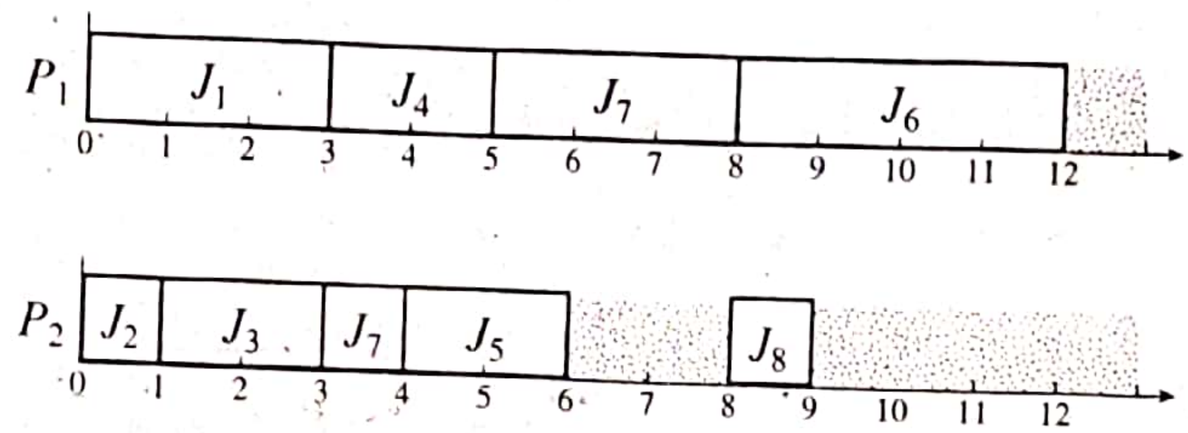
Figure 4–2 gives an example. The task graph shown here is a classical precedence graph; all its edges represent precedence constraints. The number next to the name of each job is its execution time. J_5 is released at time 4. All the other jobs are released at time 0. We want to schedule and execute the jobs on two processors P_1 and P_2. They communicate via a shared memory. Hence the costs of communication among jobs are negligible no matter where they are executed. The schedulers of the processors keep one common priority queue of ready jobs. The priority list is given next to the graph: J_i has a higher priority than J_k if $i < k$. All the jobs are preemptable; scheduling decisions are made whenever some job becomes ready for execution or some job completes.

Figure 4–2(a) shows the schedule of the jobs on the two processors generated by the priority-driven algorithm following this priority assignment. At time 0, jobs J_1, J_2, and J_7 are ready for execution. They are the only jobs in the common priority queue at this time. Since J_1 and J_2 have higher priorities than J_7, they are ahead of J_7 in the queue and hence are scheduled. The processors continue to execute the jobs scheduled on them except when the following events occur and new scheduling decisions are made.

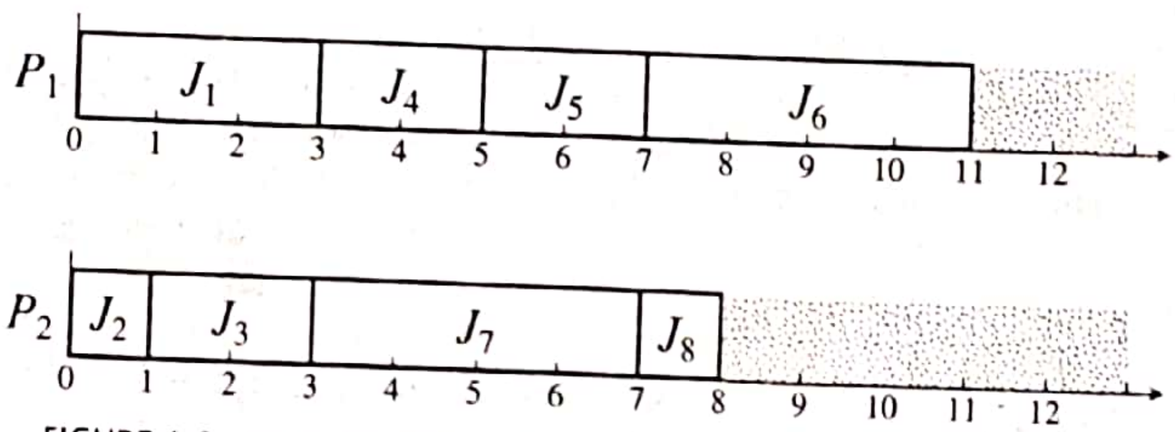
- At time 1, J_2 completes and, hence, J_3 becomes ready. J_3 is placed in the priority queue ahead of J_7 and is scheduled on P_2, the processor freed by J_2.
- At time 3, both J_1 and J_3 complete. J_5 is still not released. J_4 and J_7 are scheduled.
- At time 4, J_5 is released. Now there are three ready jobs. J_7 has the lowest priority among them. Consequently, it is preempted. J_4 and J_5 have the processors.
- At time 5, J_4 completes. J_7 resumes on processor P_1.
- At time 6, J_5 completes. Because J_7 is not yet completed, both J_6 and J_8 are not ready for execution. Consequently, processor P_2 becomes idle.
- J_7 finally completes at time 8. J_6 and J_8 can now be scheduled and they are.

Figure 4–2(b) shows a nonpreemptive schedule according to the same priority assignment. Before time 4, this schedule is the same as the preemptive schedule. However, at time 4 when J_5 is released, both processors are busy. It has to wait until J_4 completes (at time 5) before it can begin execution. It turns out that for this system this postponement of the higher priority job benefits the set of jobs as a whole. The entire set completes 1 unit of time earlier according to the nonpreemptive schedule.





(a)



(b)

FIGURE 4–2 Example of priority-driven scheduling. (a) Preemptive (b) Nonpreemptive.

In general, however, nonpreemptive scheduling is not better than preemptive scheduling. A fundamental question is, when is preemptive scheduling better than nonpreemptive scheduling and vice versa? It would be good if we had some rule with which we could determine from the given parameters of the jobs whether to schedule them preemptively or nonpreemptively. Unfortunately, there is no known answer to this question in general. In the special case when jobs have the same release time, preemptive scheduling is better when the cost of preemption is ignored. Specifically, in a multiprocessor system, the minimum makespan (i.e., the response time of the job that completes last among all jobs) achievable by an optimal preemptive algorithm is shorter than the makespan achievable by an optimal nonpreemptive algorithm. A natural question here is whether the difference in the minimum makespans achievable by the two classes of algorithms is significant, in particular, whether the theoretical gain in makespan achievable by preemption is enough to compensate for the context switch overhead of preemption. The answer to this question is only known for the two-processor case. Coffman and

Difference b/w Static and Dynamic Real time Scheduling -

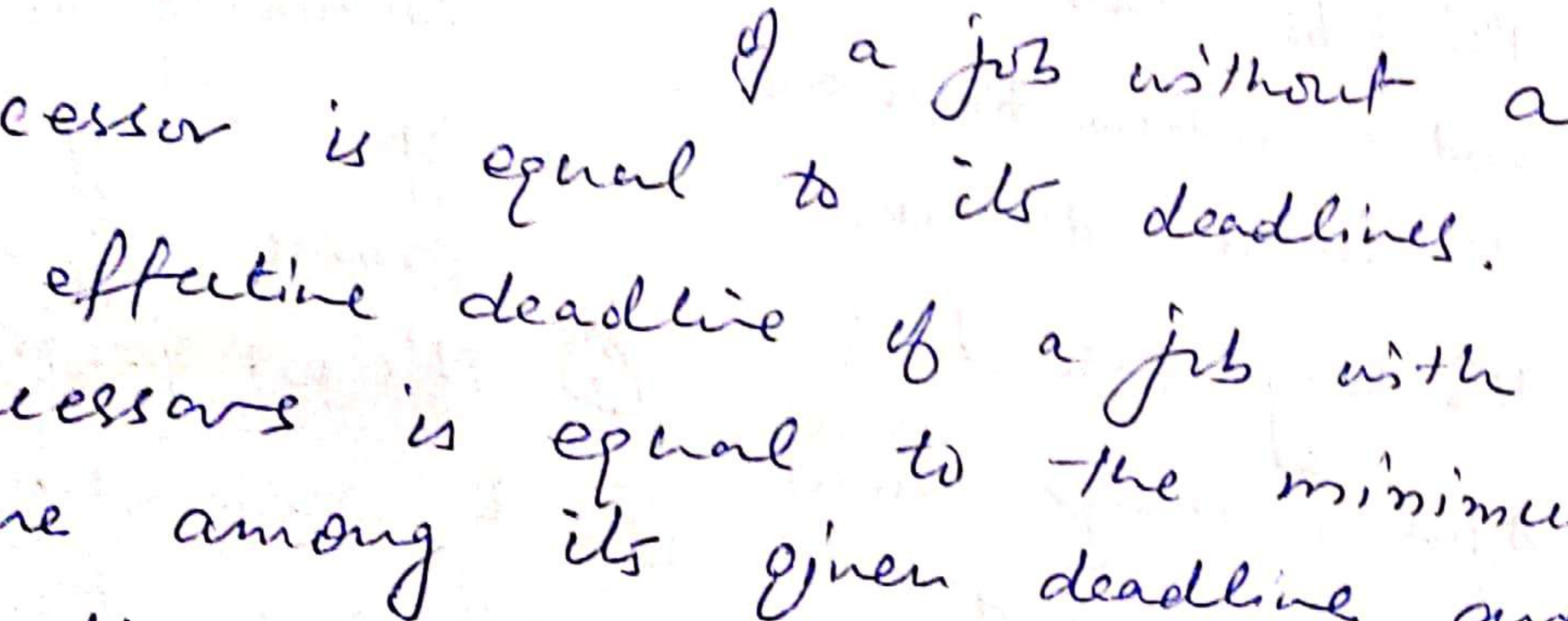
Static Real time Scheduling	Dynamic real time Scheduling
(1) All scheduling decision at compile time.	(1) All scheduling decision at run time.
(2) It determine schedule offline.	(2) It determine schedule online.
(3) All jobs, their arrival and execution times are known in advance.	(3) Priorities are assign statically as well as dynamically to the process.
(4) It's very predictable	(4) It's not predictable.
(5) Dynamic changes can not be accomodated They are clock driven	(5) It accomodate all dynamic changes.
Exp - Round-Robin Scheduling	Exp Static priority Scheduling — RM (Rate monotonic), DM (Deadline Monotonic) Dynamic priority Scheduling — EDF (Earliest deadline first), LLF or LST (least laxity or least slack Time)

Effective Release Time → The effective release time of a job without predecessors is equal to its given release time.

→ The effective release time of a job with predecessors is equal to the maximum value among its given release time and the effective release times of all of its predecessors.

Effective deadline :→ The effective deadline of a job without successor is equal to its deadlines.

→ The effective deadline of a job with successors is equal to the minimum value among its given deadline and the effective deadlines of all of its successors.

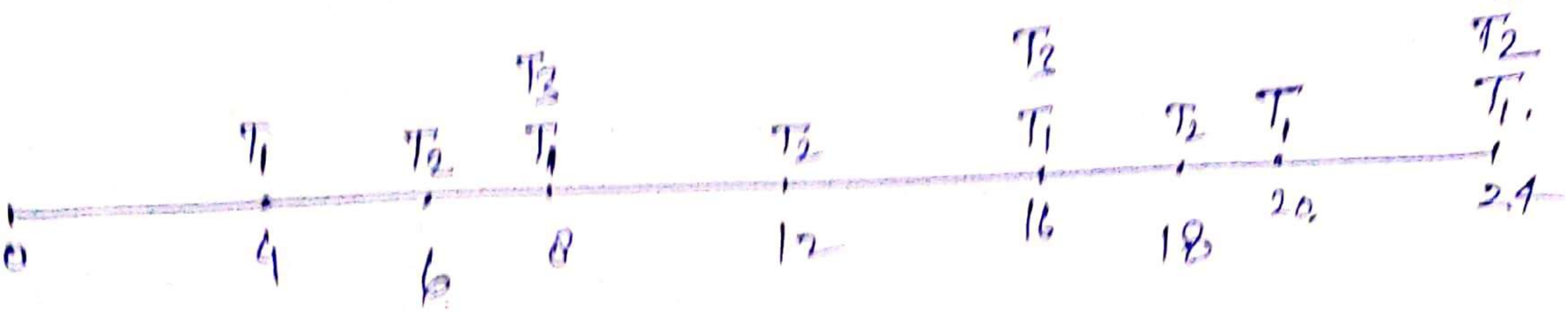


	r_i	p	e	D
T_1	0	4	1	4
T_2	0	6	2	6
T_3	0	8	3	8

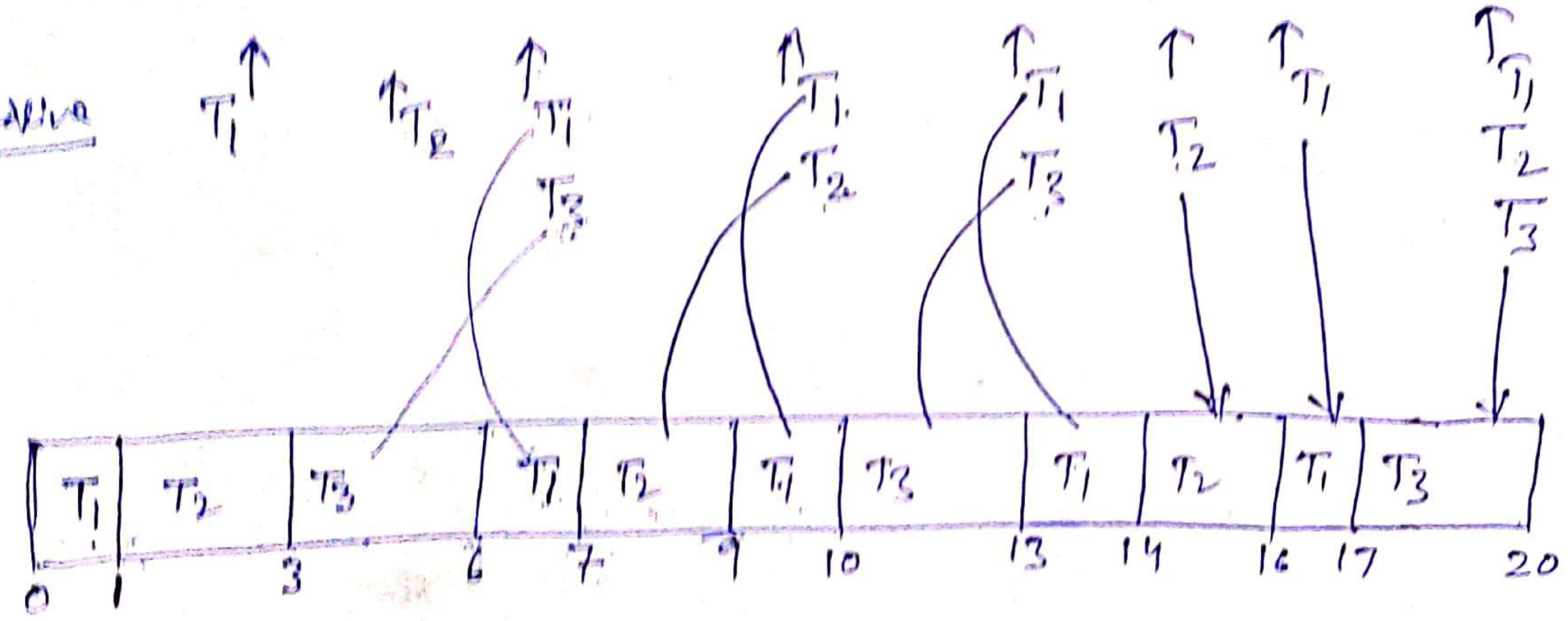
EDF

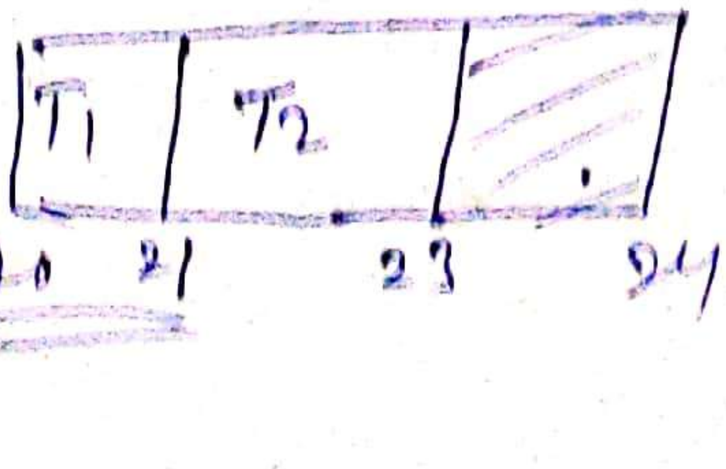
Deadline

2 | 4, 6, 8
3 | 2, 3, 4
1, 3, 2



Deadline





$u_1 = \frac{1}{4}$

Effective Release Time

$T_1 = 2$

$T_2 = 0$

$T_3 = 2$

$T_4 = 4$

$T_5 = 2$

$T_6 = 4$

$T_7 = 6$

Effective deadline (4)

$T_1 = 8$

$T_2 = 7$

$T_3 = 8$

$T_4 = 9$

$T_5 = 8$

$T_6 = 20$

$T_7 = 21$

EDF (Earliest Deadline first)

closest deadline → higher Priority

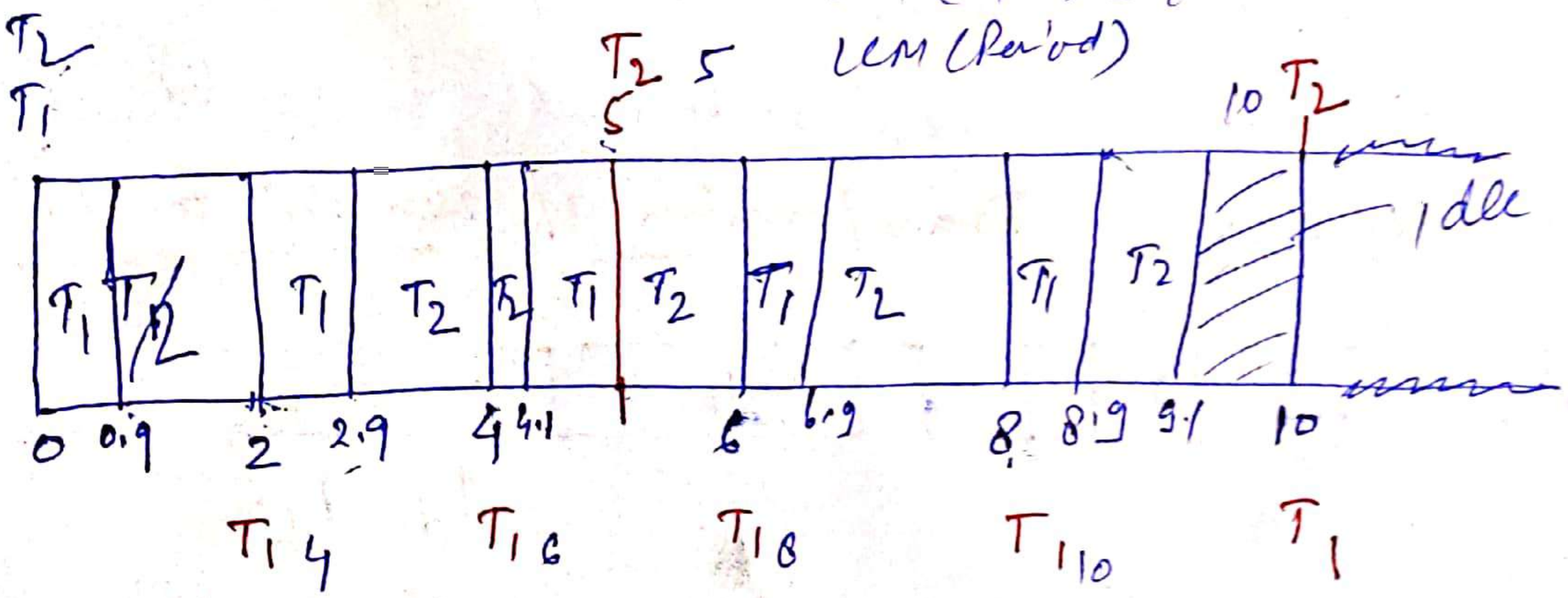
Eg:

	P	C	D
T_1	2	0.9	2
T_2	5	2.3	5

Initially Release Time for T_1 & T_2 is zero

LCM (2, 5) = 10

LCM (Period)



① $2 - 0.9 = 1.1$

$2.3 - 1.1 = 1.2$

~~2.5 1.2 at~~ 0

② $4 - 2.9 = 1.1$

$1.2 - 1.1 = 0.1$

~~2.5 1.5 0.2~~ 0

Optimality of the EDF

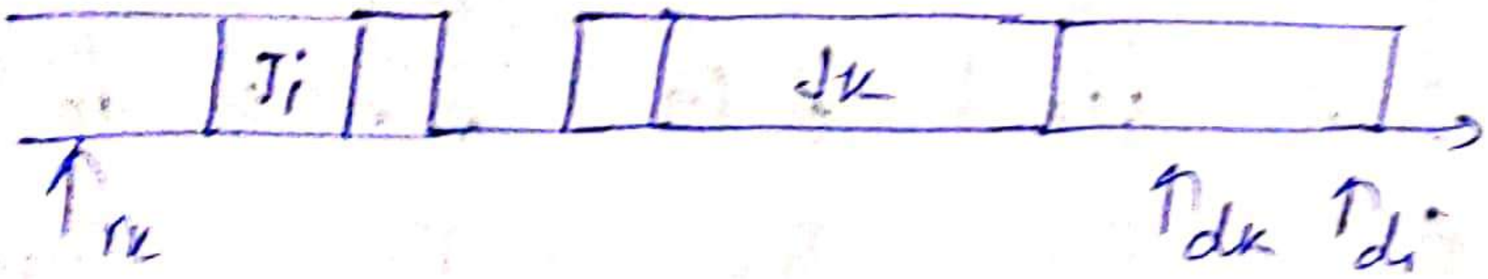
(1a)

Theorem: when preemption is allowed and jobs do not contend for resources, the EDF algorithm can produce a feasible schedule of a set J of jobs with arbitrary release times and deadlines on a processor if and only if J has feasible schedules.

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 ⇒ They are already in EDF order.

- The release time of J_k is before the end of the interval in which J_i executes: Transformation of Non EDF to EDF



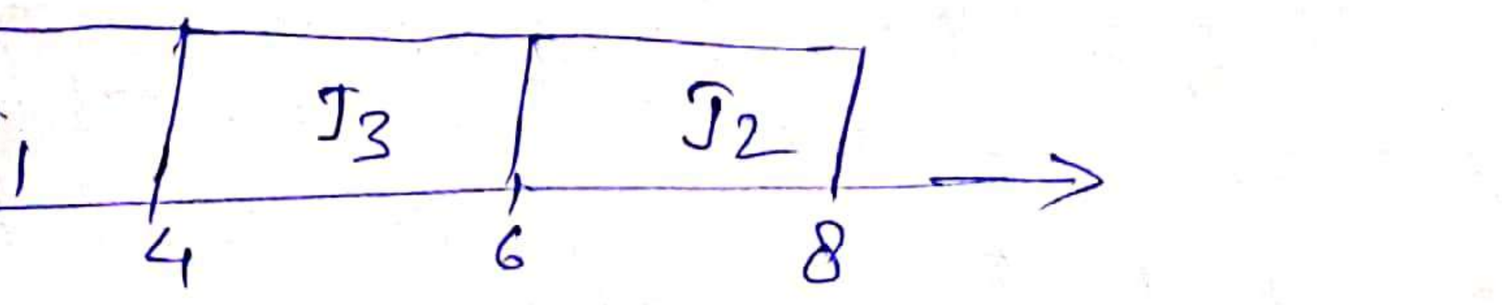
- Swap J_i and J_k (J_i's deadline is later than J_k's

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

the result is an EDF schedule.
So if EDF fails to produce a feasible schedule, no feasible schedule exists.

LRT (Latest Release Time) - (Reverse EDF algorithm) — This algorithm treats times as release time and schedule the jobs as deadline and deadlines backwards. starting from the latest dead-line of all jobs.

the priorities are based on the release times of jobs: the later the release time, the highest priority.



Example illustrating the LRT

* The latest deadline among all jobs is 8. Hence time starts at 0 and goes backwards to zero (0).

Least - Slack - Time - first (LST) - Also

Called the Minimum - Laxity - First (MLF) algorithm.

priorities to jobs based on their slacks

Smaller the slack value → Higher the priority (↓) (↑)

Slack - At any time t, the slack of a job with deadline at d is equal to d - t - e' (e': time required to complete the remaining portion of job).

Eg:

	Φ	C	D	P
T1	0	3	7	20
T2	0	2	4	5
T3	0	2	8	10

Slack time (D - t - e')

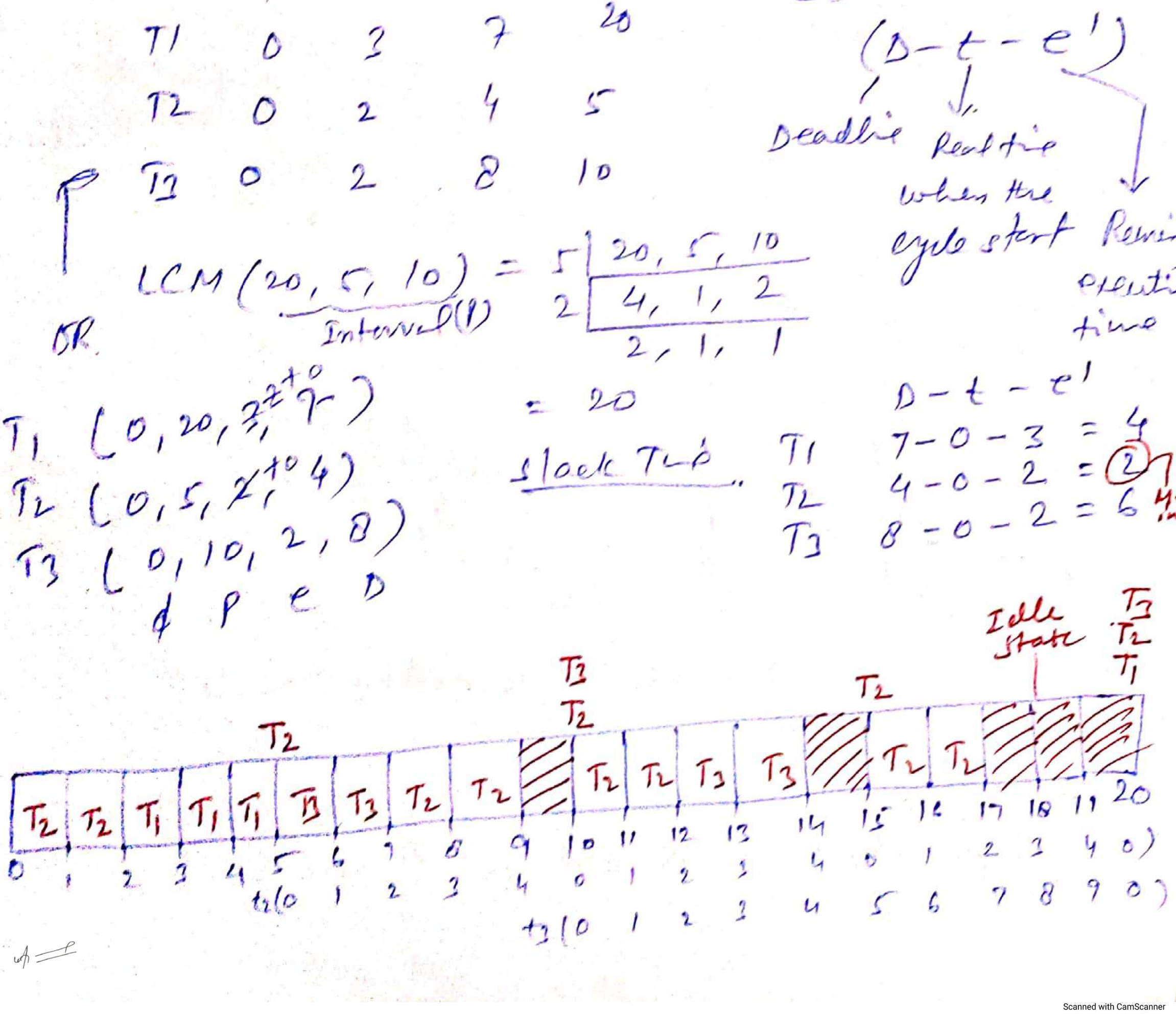
D: Deadline, t: Real time when the cycle start, e': Remaining execution time

OR. LCM (20, 5, 10) = 5 | 20, 5, 10 ; 2 | 4, 1, 2 ; 2, 1, 1 = 20
Interval (I)

Slack Table

	D - t - e'
T1	7 - 0 - 3 = 4
T2	4 - 0 - 2 = 2
T3	8 - 0 - 2 = 6

T1 (0, 20, 7, 4)
T2 (0, 5, 4, 2)
T3 (0, 10, 8, 6)



(2)

D - t - e

$T_1 \rightarrow 7-1-3 = 3$

$T_2 \rightarrow 4-1-1 = (2)$ Min

$T_3 \rightarrow 8-1-2 = 5$

D - t - e

$T_1 \rightarrow 7-3-2 = [2]$ Min

$T_3 \rightarrow 8-3-2 = 3$

$T_1 \rightarrow 7-2-3 = (2) \rightarrow$ Min (13)

$T_2 \rightarrow 4-2-0 = 2$ ✗ (Execution is Complete No need to Calculate Slack time)

$T_3 \rightarrow 8-2-2 = 4$

$T_1 \rightarrow 7-4-1 = 2$ (Tie)

$T_3 \rightarrow 8-4-2 = 2$

T_1 execution time completes its within interval 20. we will not considered for the T_1.

D - t - e

$T_2 \rightarrow 4-0-2 = 2$

$T_3 \rightarrow 8-5-2 = (1)$ Min

$T_2 \rightarrow 4-1-2 = 1$ (Tie)

$T_3 \rightarrow 8-6-1 = 1$

T_2 (0, 5, 2, 4)

T_3 (0, 10, 2, 8)

T_3 complete the execution in the mentioned (10) interval so we can execute now T_2. (No need to calculate the Slack time)

D - t - e

$T_2 \rightarrow 4-0-2 = (2)$ min

$T_3 \rightarrow 8-0-2 = 6$

$T_2 \rightarrow 4-1-1 = 2$

$T_3 \rightarrow 8-1-2 = 5$

T_2 (0, 5, 2, 4)

T_3 (0, 10, 2, 8)

T_2 complete the execution time.

No need to Calculate Slack time because only T_2 is left for the execution rest T_1 & T_3 already completes their execution.

T_2 (0, 5, 2, 4)

Rate Monotonic Algorithm (RMA)

It's well known fixed priority algorithm. priorities are assigned to task according to their periods.

Shorter the period, the higher the priority.

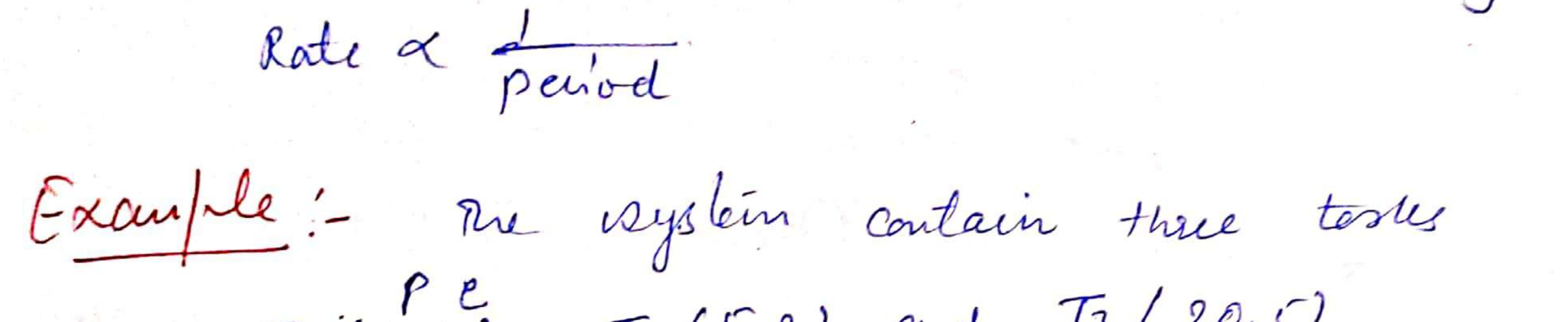
$$\text{Rate} \propto \frac{1}{\text{period}}$$

Example :- The system contain three tasks

$T_1(\overset{P}{4}, \overset{e}{1})$, $T_2(5, 2)$ and $T_3(20, 5)$

priority — $T_1 > T_2 > T_3$ fixed Priority

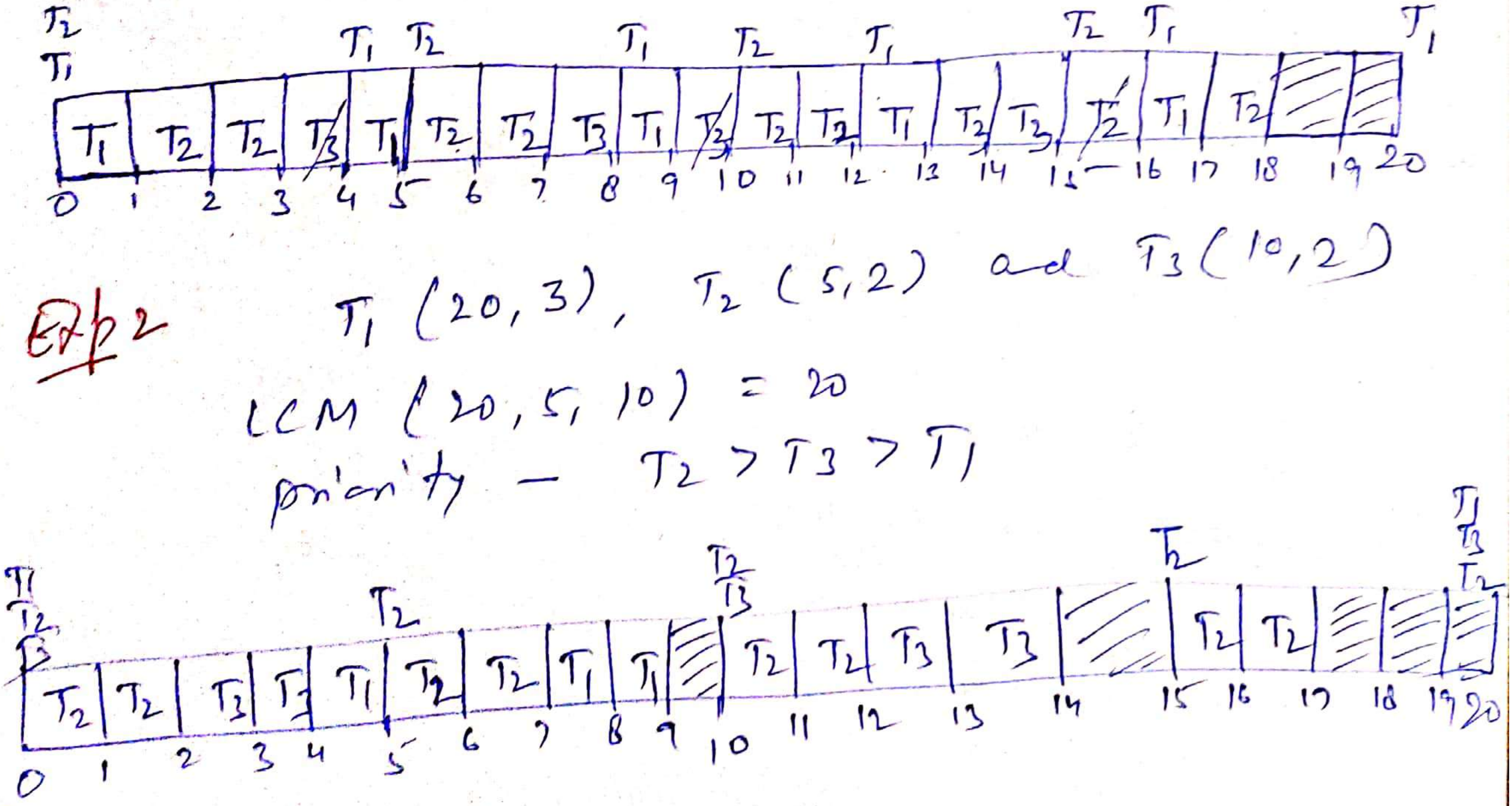
LCM (4, 5, 20) = 20



Ex 2 $T_1(20, 3)$, $T_2(5, 2)$ and $T_3(10, 2)$

LCM (20, 5, 10) = 20

priority — $T_2 > T_3 > T_1$



(4) Schedulability Test for RMA. To test whether a set of periodic tasks can be feasibly be scheduled under RMA.

(a) Necessary Condition: A set of periodic real time tasks would not be RMA schedulable unless they satisfy the necessary condition

$$\sum_{i=1}^{n} \frac{e_i}{p_i} = \sum_{i=1}^{n} U_i \leq 1$$

Here e_i : execution Time
p_i : period of Task T_i
n : total no. of Task
U_i : utilization due to task T_i

(b) Sufficient Condition Liu and Layland's Test

$$\sum_{i=1}^{n} U_i \leq n\left(2^{1/n} - 1\right)$$

If set of task satisfies the sufficient condition, then the set of tasks would be RMA schedulable.
If fails, we should not conclude that it's not schedulable under RMA.

Further checking is needed, the test name is Lehoczky's Test

Lehoczky's Test : A set of periodic real-Time is RMA schedulable under any task phasing if all the task meet their respective first deadline.

Task T_i could meet its first deadline if

$$e_i + \sum_{k=1}^{i-1} \left\lceil \frac{d_i}{d_k} \right\rceil \times e_k \leq d_i$$

Ex 1 Check the following set of periodic real-time task is schedulable under RMA on a uniprocessor.

	C	P
T_1	20	100
T_2	30	150
T_3	60	200

Sol'n CPU utilization U_i (Necessary Condition)

$$\sum_{i=1}^{n=3} \frac{C_i}{P_i} = \sum_{i=1}^{n=3} U_i = \frac{20}{100} + \frac{30}{150} + \frac{60}{200}$$

$$= 0.7$$

which is less than 1 means the necessary condition for schedulability of the task is satisfied.

Sufficient condition –

$$\sum_{i=1}^{n} U_i \leq n(2^{1/n} - 1)$$

$$0.7 \leq 3(2^{1/3} - 1)$$

$$0.7 \leq 0.78 \quad \text{(Condition True)}$$

under sufficient condition here 0.7 is total utilization of Task. Task set is RMA schedulable.

If fails then we will check for the next i.e., Test Lehoczky's Test. But here no need to check because 2nd Test satisfied.

Ex 2. Show the periodic task (10, 2) (15, 5) (25, 9) are schedulable by the RMS.

already task scheduled as per the period in term.

$T_1 > T_2 > T_3$

Task	P	e	d=p
T_1	10	2	10
T_2	15	5	15
T_3	25	9	25

first check the necessary condition

$$\sum_{i=1}^{3} U_i \leq 1 = \frac{2}{10} + \frac{5}{15} + \frac{9}{25} = 0.893 \leq 1$$

Sufficient condition

$$\sum_{i=1}^{3} U_i \leq n(2^{1/n} - 1)$$

$$0.893 \leq 3(2^{1/3} - 1)$$

$0.893 \leq 0.78$ condition false

then apply Lechozy Test

for T_1, $P_1 = 10$, $e_1 = 2$ $\quad 2 < 10$

so it meet its deadline.

for T_2, $e_2 + \sum_{k=1}^{1} \frac{15}{10} \times e_1 = 5 + \frac{15}{10} \times 2 < 15$

$= 7 \leq 15$

so its meet its deadline.

for $i = 3$,

$$T_3 = e_3 + \sum_{k=1}^{2} \frac{P_i}{P_k} \times e_k \leq P_i$$

$$= e_3 + \frac{P_3}{P_1} \times e_1 + \frac{P_3}{P_2} \times e_2 \leq P_3$$

$$= 9 + \frac{25}{10} \times 2 + \frac{25}{15} \times 5 \leq 25$$

$$= 9 + 3 \times 2 + 2 \times 5 \leq 25$$

$$= 25 \leq 25$$

So T_3 meet its deadline.

So T_1 T_2 T_3 meet their deadline so they are RMA schedulable.

Diff^n b/w Offline / Online Scheduling -

Offline	Online
• Schedule computed before the system begin execution	• Priority driven algorithm
• Inflexibility ← Disadvantage	• flexibility ← advantage
• Runtime overload ← Advantage (low)	• can not be used resources at best/optimal.