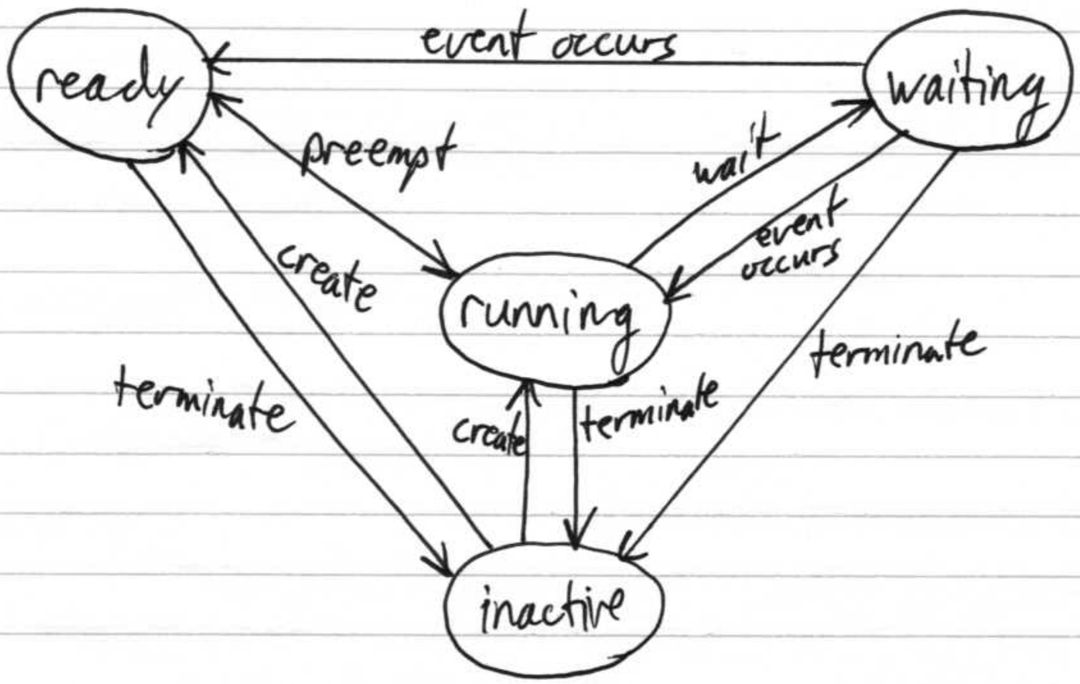
4) Scheduling [7.3]

4.1) Task States



- the scheduler runs when a task:
    - is created
    - terminates
    - yields e.g. osThreadYield() - tell OS to run another task
    - blocks e.g. osSignalWait(), osDelay()
    - unblocks e.g. signal received, delay ends
    - finishes its timeslice (default RTX timeslice = 5ms ⇒ 200Hz)

4.2) Task Types [7.2.4.1]

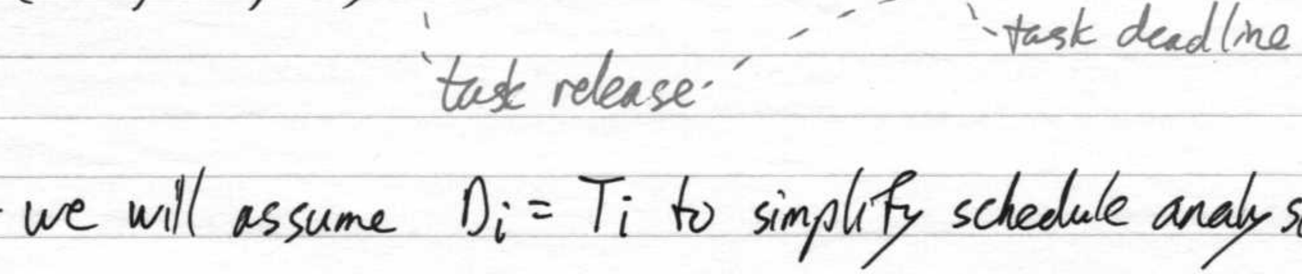
1. periodic - repeats regularly
   e.g. polling I/O
   kicking the watchdog timer

2. aperiodic - can occur any time
   - prohibits schedule analysis since it occur any time and cause indeterminate delays

3. sporadic - irregular release (execution) but with maximum frequency
   - permits schedule analysis
   e.g. axle rotation sensor

- tasks can be characterized by:
  - T period (or for sporadic task minimum inter-arrival time)
  - C worst-case execution time (WCET)
  - D deadline (relative to release time)

eg.



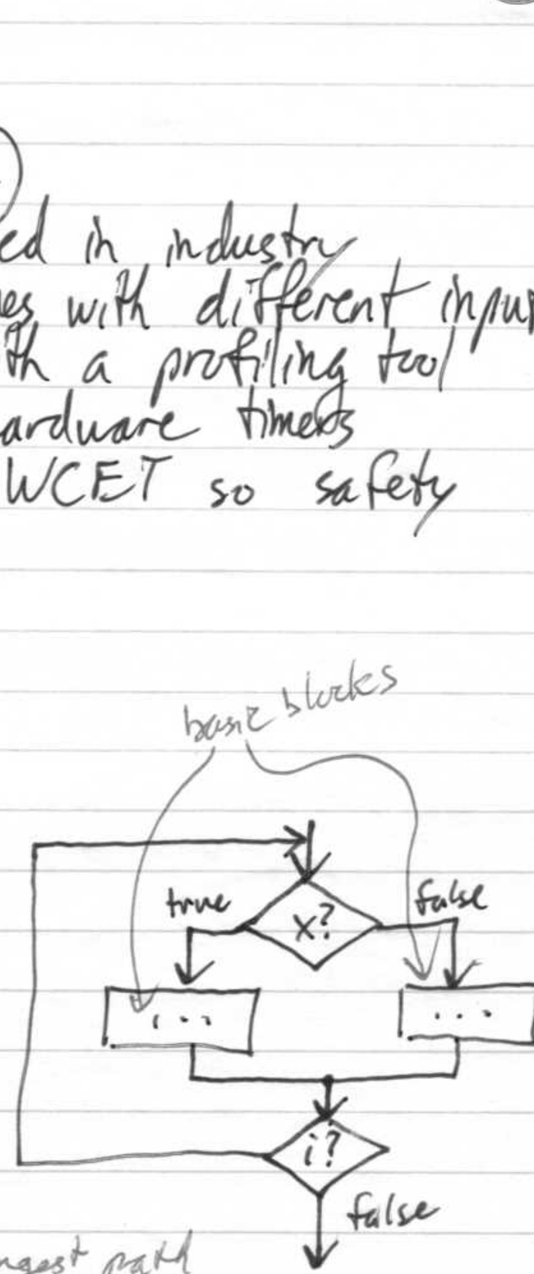
- we will assume $D_i = T_i$ to simplify schedule analysis

- WCET analysis methods

  1. measurement (dynamic)
     - most common method used in industry
     - run the program many times with different inputs
     - runtime is measured with a profiling tool such as gprof or hardware timers
     - this can underestimate WCET so safety margins are added

  2. static analysis
     - flow-control analysis

e.g.
```
do {
    if(x > 0) {
        ...
    } else {
        ...
    }
} while (i != j);
```
— estimate loop bounds and longest path



- use low-level analysis to estimate time of basic blocks or individual instructions
- flow-control and timing estimates are combined to calculate the WCET
- prone to over-estimating WCET

  3. hybrid of flow-analysis and measurement of basic-blocks

4.3) How to Evaluate a Scheduler

1. minimize missed deadlines or minimize lateness
2. maximize processor utilization

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$ percentage of time that the processor is busy

3. minimize scheduler overhead (time to decide next task to execute)

4.4) Non-preemptive Scheduling [7.3]

- advantage: minimizes scheduler overhead

4.4.1) Timeline Scheduling [7.3.1]

a.k.a. superloop

- tasks execute in a fixed order which repeats indefinitely
- the schedule is created offline

e.g. task set ($D_i = T_i$)

	T (period)	C (wcet)
τ_1	18	8
τ_2	30	10
τ_3	45	10

- check processor utilization

$$U = \frac{8}{18} + \frac{10}{30} + \frac{10}{45} = 1 \leq 1 ✓$$

- schedule length = LCM (least common multiple) of ~~task~~ task periods

- LCM calculation ~~#~~ techniques

  ① $LCM(a,b) = \frac{a * b}{GCD(a,b)}$ GCD = greatest common divisor

  $LCM(a,b,c) = LCM(LCM(a,b), c)$

  ② decompose into prime factors
   - find smallest prime divisor
   - divide period by divisor and repeat

  e.g.

$18 \div \underline{2} = 9$	$30 \div \underline{2} = 15$	$45 \div \underline{3} = 15$
$9 \div \underline{3} = 3$	$15 \div \underline{3} = 5$	$15 \div \underline{3} = 5$
$3 \div \underline{3} = 1$	$5 \div \underline{5} = 1$	$5 \div \underline{5} = 1$
$18 = 2 \times 3^2$	$30 = 2 \times 3 \times 5$	$45 = 3^2 \times 5$

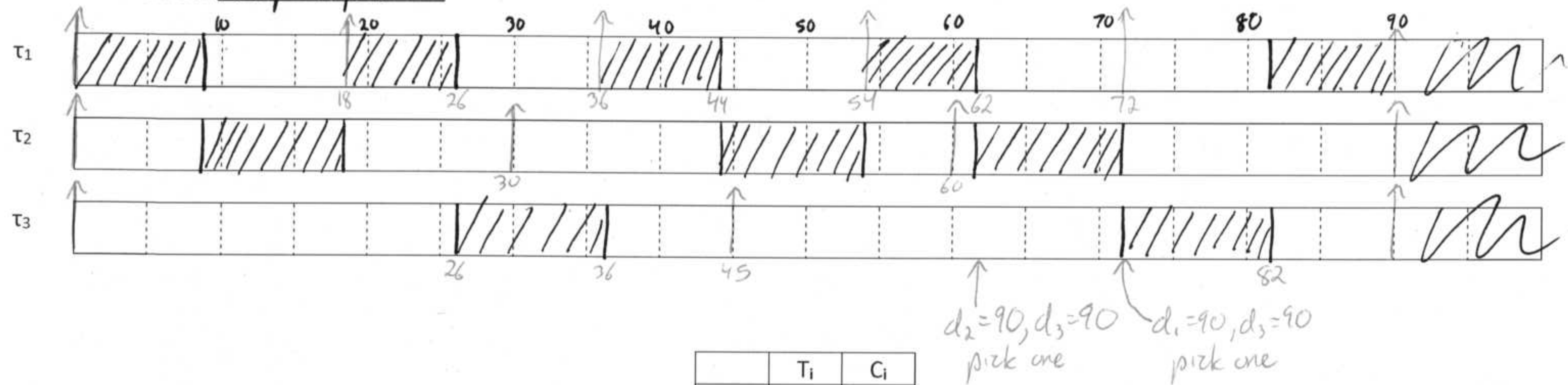
  then multiply the prime factors of highest power

  e.g. $LCM = 2 \times 3^2 \times 5 = \underline{90}$

- create schedule using Earliest Deadline First (see handout)

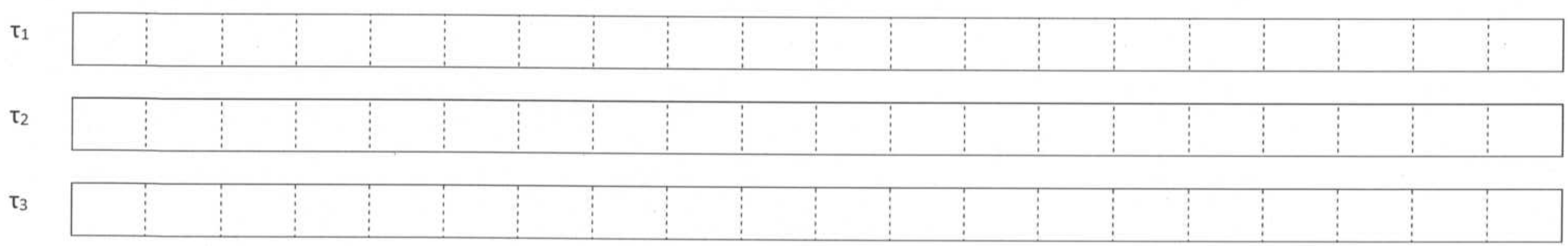
	T_i	C_i
τ_1	18	8
τ_2	30	10
τ_3	45	10

Method: non-preemptive EDF



	T_i	C_i
τ_1		
τ_2		
τ_3		

Method: ____________________



16a

- implementation: store the order in an array and the scheduler moves to the next element (task) each time
- scheduler overhead: $O(1)$
- array length $= \sum_{i=1}^{n} \frac{LCM}{T_i} = \frac{90}{18} + \frac{90}{30} + \frac{90}{45} = 10$
- disadvantage: the storage required for the schedule can be large

e.g. 5 tasks with deadlines $T_1 = 18$, $T_2 = 30$, $T_3 = 45$, $T_4 = 22$, $T_5 = 23$

schedule length $= 2 \times 3^2 \times 5 \times 11 \times 23 = 22770$

array length $= \frac{22770}{18} + \frac{22770}{30} + \frac{22770}{45} + \frac{22770}{22} + \frac{22770}{23}$
$= 4555$

- to shorten schedule and array length, you can <u>reduce</u> some task periods, as long as it is still schedulable ($U \le 1$)

e.g. $T_1 = 18$, $T_2 = 30$, $T_3 = 45$, $T_4 = \underline{20}$, $T_5 = 20$
$\quad$ $L = 2^2 \times 5$

schedule length $= 2^2 \times 3^2 \times 5^1 = 180$

array length $= 180\left(\frac{1}{18} + \frac{1}{30} + \frac{1}{45} + \frac{1}{20} + \frac{1}{20}\right)$
$= 38$

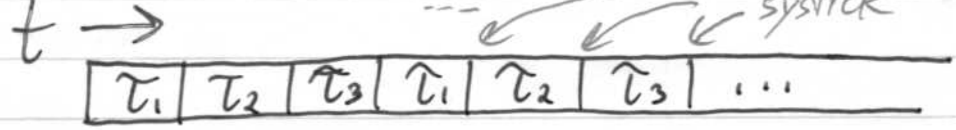
4.5) Preemptive Schedulers [7.4]

- tasks are suspended in mid-execution to allow another task to run

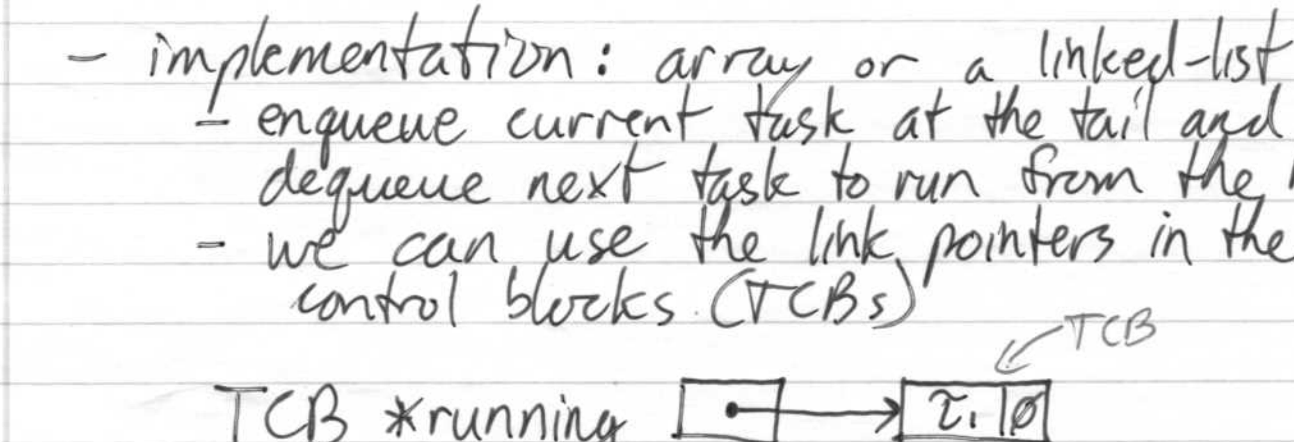
4.5.1) Round-Robin Preemptive

ruban rond

- all tasks have equal priority
- tasks switch ~~on the SysTick interrupt~~ at the end of ~~their~~ each timeslice



- implementation: array or a linked-list queue
  - enqueue current task at the tail and dequeue next task to run from the head
  - we can use the link pointers in the task control blocks (TCBs)

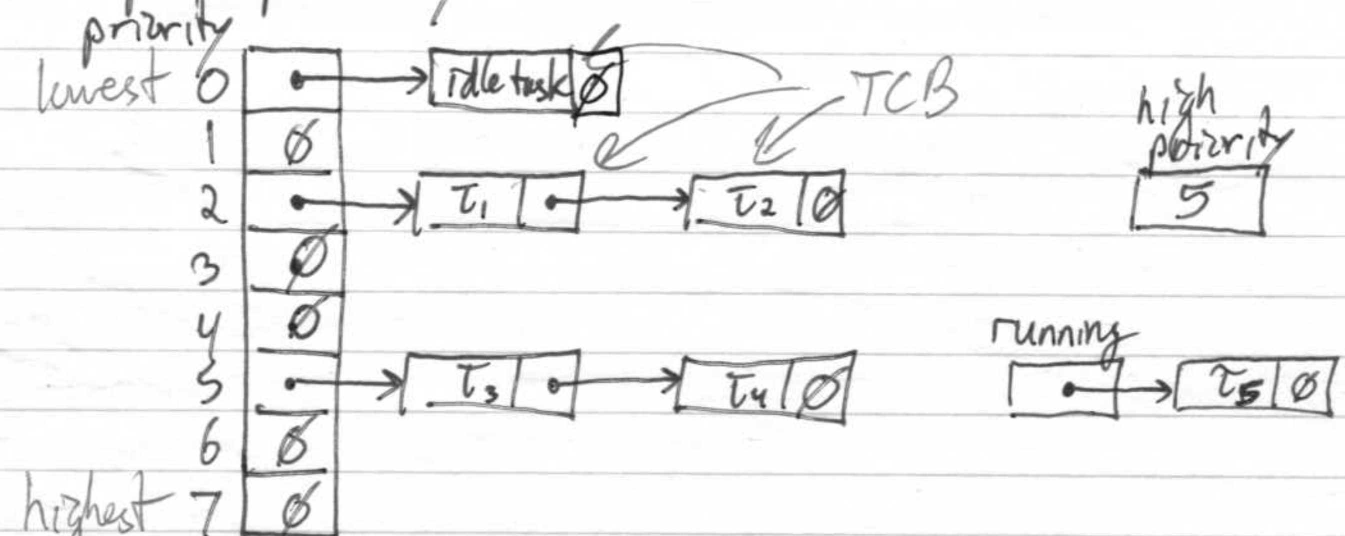


- scheduling overhead is $O(1)$
- disadvantages: no priorities and no deadlines

4.5.2) Fixed-Priority Preemptive Scheduling [7.4.5]

- task priorities are static (don't change)
- run the ready tasks with highest priority in round-robin fashion

- implementation: an array of linked-list queues, one per priority level



- when a timeslice (sys tick) ends, enqueue the current task at the back of the queue and deque next front the front
- adding a task is $O(1)$
- what happens when the high-priority queue empties?
  1. linear scan for next non-empty queue $= O(n)$

  or 2. use a bit vector to represent non-empty queues

e.g.

0	0	1	0	0	1	0	1
7	6	5	4	3	2	1	0

- the first set bit can be found with an instruction such as ARM CLZ (count leading zeroes) instruction
- can also be done in software in constant time using deBruijn sequences

- advantage: $O(1)$ scheduling overhead
  $\Rightarrow$ widely used
- disadvantages: - no deadlines
  - frequent scheduling / context switches

- Rate Monotonic scheduling (RM) [7.4.5.2]
  - a way to assign fixed priorities based on deadlines
    priority $\propto$ execution rate of task
    (~~shorter period~~ higher frequency $\Rightarrow$ higher priority)

- schedulability test:
  a set on n tasks are schedulable by RM if

$$\prod_{i=1}^{n}\left(1+\frac{C_i}{T_i}\right) \leq 2$$

- ~~this~~ the test to accept a new task can be performed in $O(1)$ (advantage)

4.5.3) Rate Monotonic Scheduling [7.4.5.2]

- it is a way to assign ~~prop~~ priorities for F.P.P. based on deadlines
    - priority is relative to the task frequency (shorter period ⇒ higher priority)

- schedulability test:
  a set of n tasks are schedulable by RM if

$$\prod_{i=1}^{n}\left(1+\frac{C_i}{T_i}\right) \leq 2$$

- the test to accept a new task can be performed in $O(1)$ time

e.g. $(D_i = T_i)$

	T_i	C_i
τ_1	10	2
τ_2	10	3
τ_3	6	1

- check processor utilization

$$U = \frac{2}{10} + \frac{3}{10} + \frac{1}{6} = 0.67 \leq 1 \checkmark$$

- RM schedulability

$$\prod_{i=1}^{n}\left(1+\frac{C_i}{T_i}\right) = \left(1+\frac{2}{10}\right)\left(1+\frac{3}{10}\right)\left(1+\frac{1}{6}\right) = 1.82 \leq 2 \checkmark$$

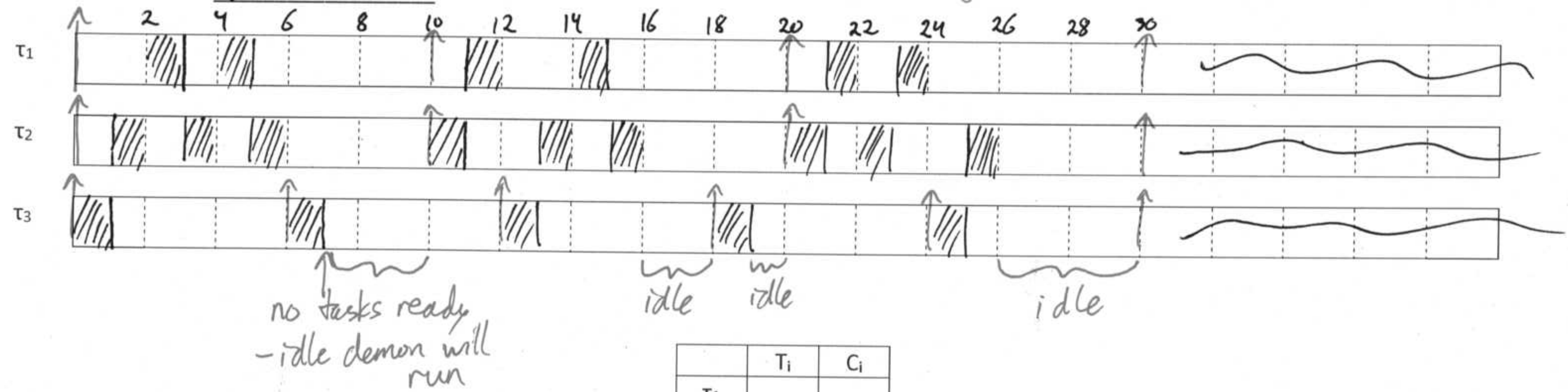
	T_i	C_i
τ_1	~~20~~ 10	2
τ_2	10	3
τ_3	6	1

} tied for lowest priority.

— highest priority

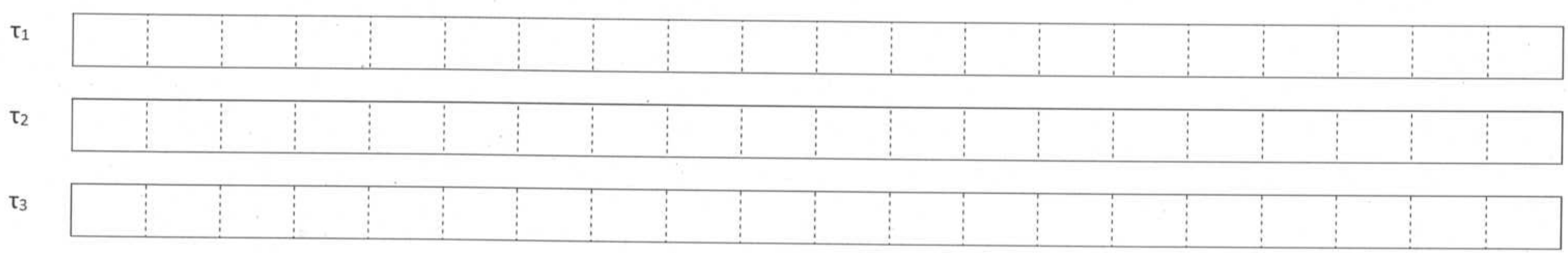
-assume timeslice = 1 time unit

Method: RM with FPP



	T_i	C_i
τ_1		
τ_2		
τ_3		

Method: ____________



21a

4.5.4) Preemptive EDF [7.4.4.1]

- priorities are dynamic (deadline is the priority)
- the scheduler runs the task with the closest deadline
- if a task is released with an earlier deadline it preempts the current task
- no timeslices - no SysTick interrupt

- implementation: two priority queues
  1. "ready" queue - all ready tasks, sorted by deadline
  2. "waiting" queue - waiting tasks sorted by next release

  - the scheduler overhead is O(log n) assuming that minheaps are used

  - advantages:
    - optimal scheduling algorithm
    - has the fewest scheduler invocations and fewest context switches

  - disadvantages:
    - has a complex schedulability test (which is needed to decide if a new task can be admitted)