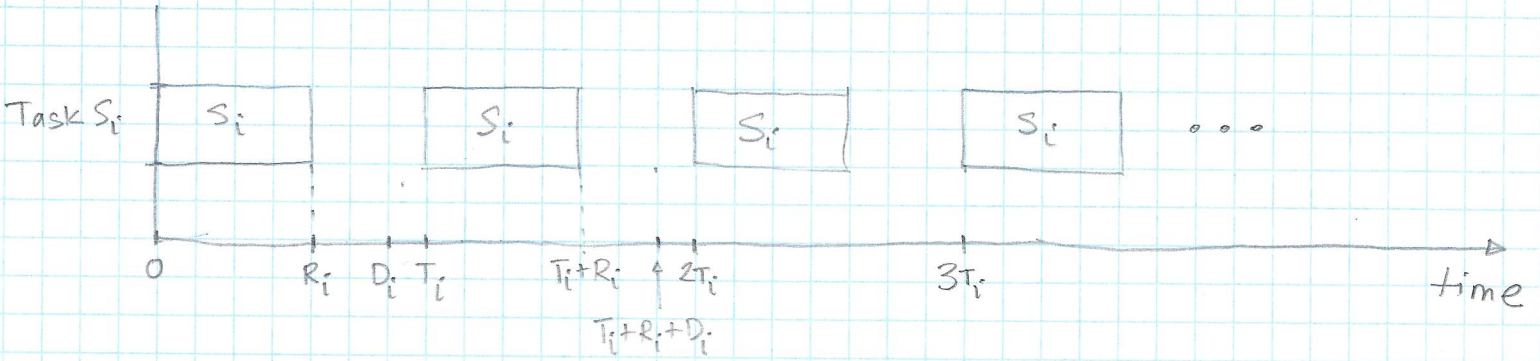
MORE MULTI-TASKING

- as we have seen, the conventional Linux kernel is not strict in the sense of task timing; Linux is designed as a "time sharing" system that attempts to provide CPU time for all users, rather than prioritize any given process.
- in a real-time OS, the scheduler is designed to be task-deadline driven, prioritizing certain tasks over others.
- hard real-time systems have time constraints which are absolute; that is, a total system failure may result from a single missed deadline. Examples of hard real-time systems include signal processing devices with fixed sampling periods, high-performance autonomous control systems found in robotics, cars and aircraft — if strict timing is not maintained, the consequences can be severe.
- soft real-time systems are those for which strict timing is desirable, but for which an occasional missed deadline has insignificant or mild consequences.
- in truth, it is difficult in many cases to prove that a system will never miss a deadline — reality presents situations that the designer may not have anticipated
- to help us to estimate the "real-time robustness" of an OS, we often assume certain conditions that simplify things in order to perform analysis. For example, we might assume deterministic (predictable) conditions, which preclude unexpected events. Tasks may be assumed to have fixed execution times and periods:

useful terminology follows...

- Suppose we have a number of tasks, each denoted by S_i, $i = 1, \ldots, N$, each with a corresponding deadline (D_i), release time (R_i), and task period (T_i).

  On a timeline, we could represent the task as:



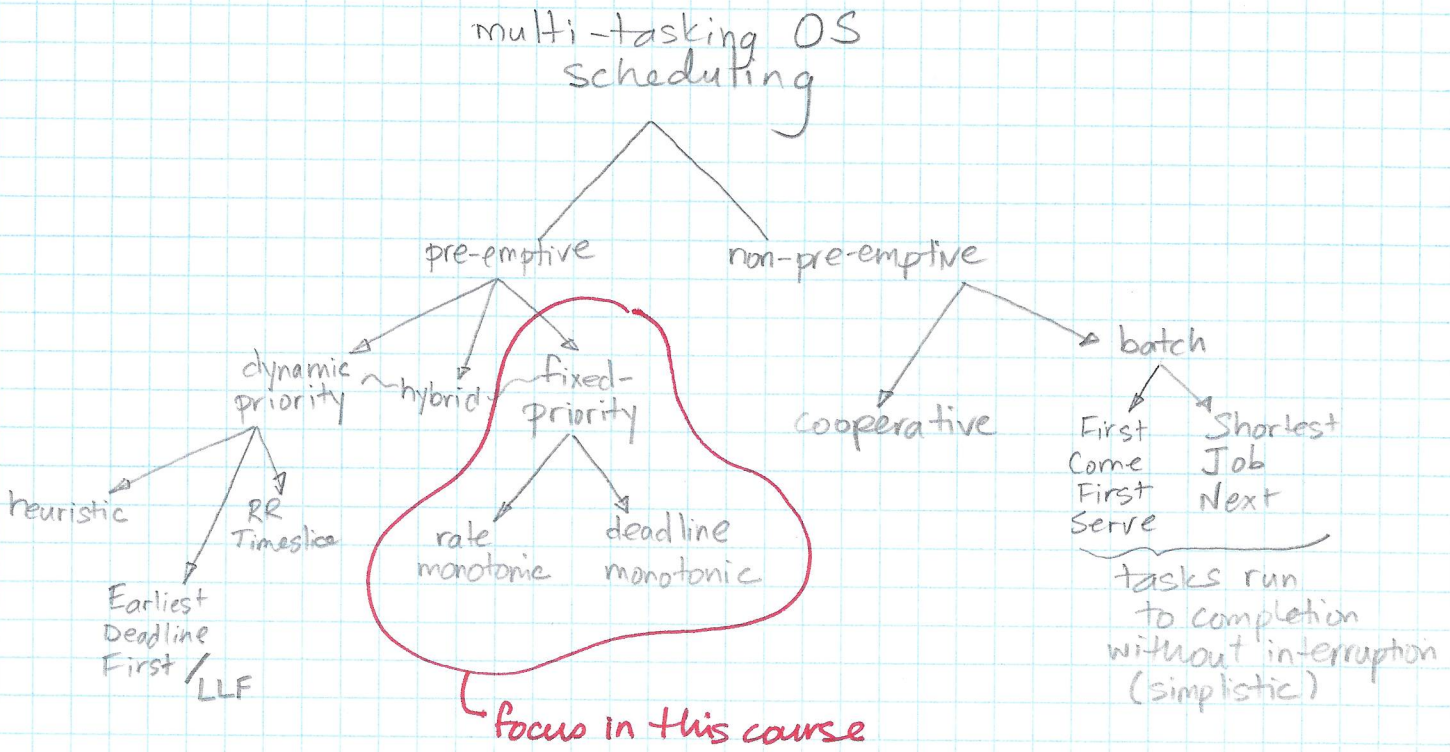
  - the deadline, D_i, is the amount of time the task S_i has to execute;
  - the release time R_i, is the time taken by the task, from when it first becomes active, until completion

  NOTE: in many examples, we often see that $D_i = R_i = T_i$ is used as a simplifying condition

  - T_i is the task period — it is sometimes helpful to assume a fixed amount of time between executions of a task, although in many real-world cases some deviation from periodic behaviour should be anticipated.

- Please bear in mind that:
  - real-time systems are often analyzed with deterministic assumptions (i.e., events are predictable);
  - service/task requests are periodic, and execution times of tasks are fixed (and known)
  - designers attempt to prove optimality (see below)

taxonomy of multi-tasking operating systems:



- fixed-priority design is at the foundation of pre-emptive operating systems
- Rate Monotonic (RM) and Deadline Monotonic (DM) two assignment policies in the fixed-priority category; both are optimal.
- An assignment policy is optimal if any set of services that can be scheduled with no missed deadlines can be scheduled with the assignment policy

→ optimality is a theoretical guarantee that a system is a hard real-time system ←

→ but bear in mind that many assumptions are made (described above) in any proof of optimality.

- in a **rate monotonic** (RM) policy higher priority is given to tasks with the highest request frequency (shortest task period) J_i;
- in a **deadline monotonic** (DM) policy higher priority is given to tasks with the shortest deadline;
- note that if $D_i = R_i = T_i$, RM and DM are equivalent.

EXAMPLE:

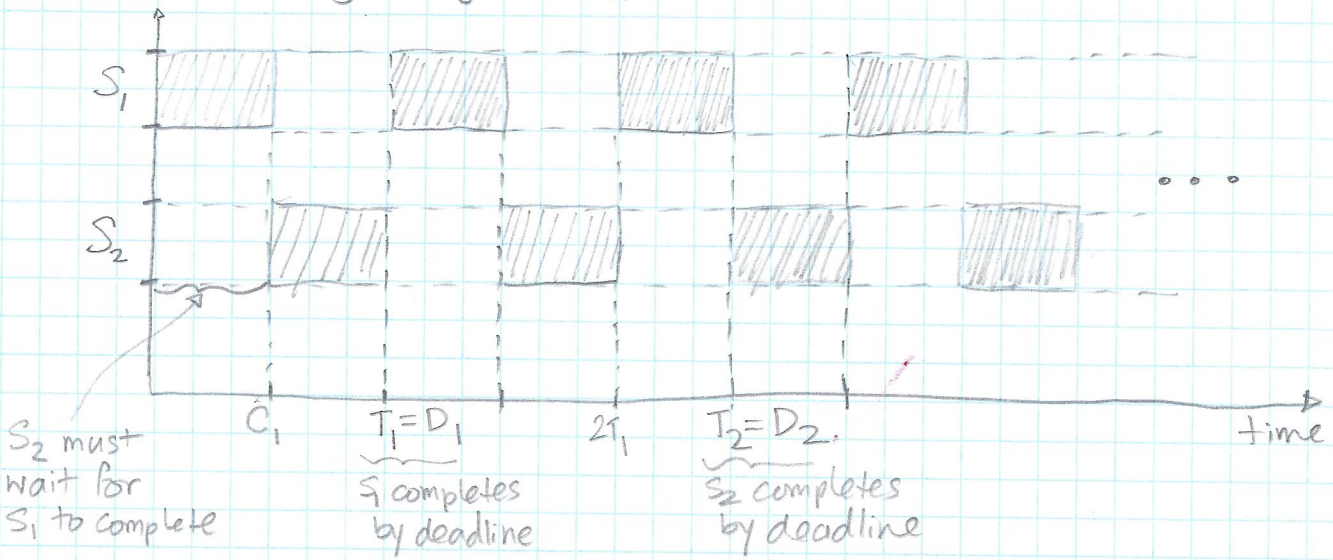
- given two tasks S_1 and S_2 with request periods $T_1 = 6$ and $T_2 = 15$ (in some unit of time), and execution times $C_1 = 3$ and $C_2 = 6$ (same time units), respectively, how does RM policy apply?

SOLN:

- because RM policy assigns higher priority to shorter task periods, we have that

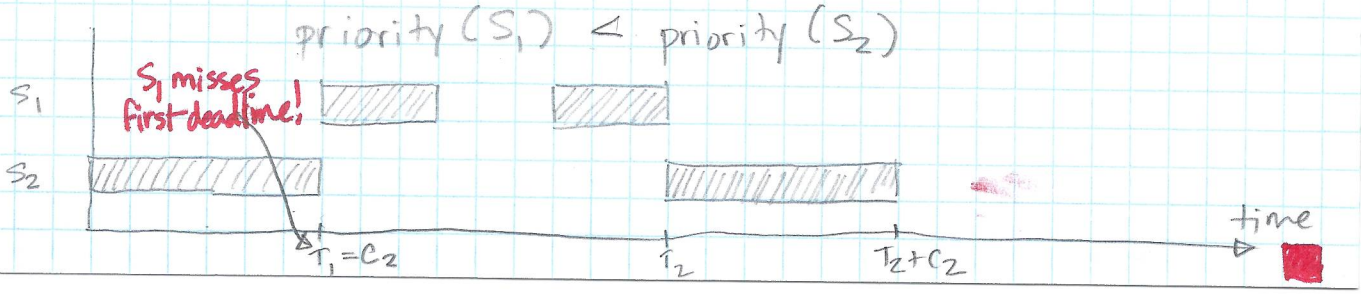
priority(S_1) > priority S_2

a timing diagram appears as follows:



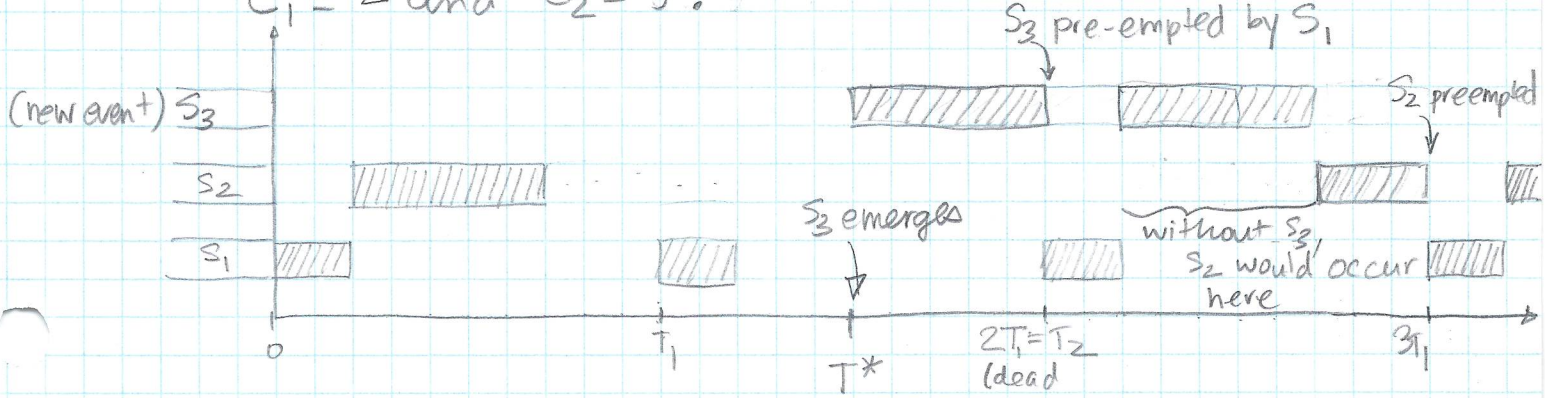
- if we adopt the anti-policy, i.e.,

priority(S_1) < priority(S_2)



- in dynamic-priority RTOS scenarios, events occur which demand on-the-fly priority assignments.
- according to earliest deadline first (EDF) policy, when a new event occurs, all priorities are adjusted such that those with earlier deadlines have higher priorities.
- EXAMPLE:

Suppose a real-time system is running w/ two tasks, S_1 and S_2; $T_1 = 10$, $T_2 = 20$, and $C_1 = 2$ and $C_2 = 5$:



- At time T^*, a third task emerges, S_3, with an execution time of 10 units, and a deadline of $T_2 + 2$.

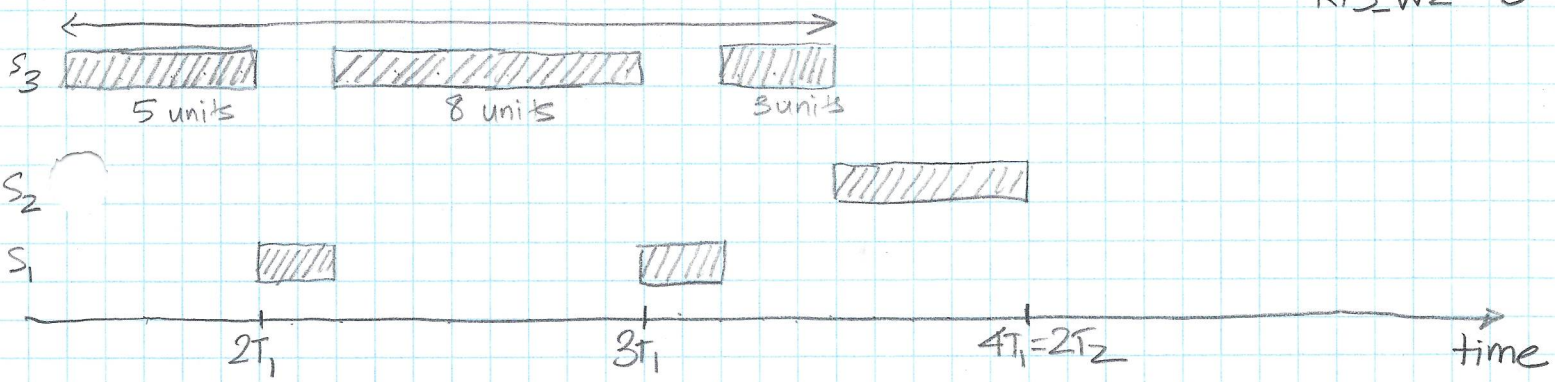
Accordingly, the task priorities are revised by EDF policy to:

$$\text{priority}(S_1) > \text{priority}(S_3) > \text{priority}(S_2)$$

and the above timing diagram results.

It appears that S_2 still makes its deadline ($4T_1$).

Q: How computationally expensive can S_3 be before S_2 misses its deadline?



∴ S_3 can take, at most, 5+8+3 = 16 units of execution time before S_2 misses its deadline ∎

- please download the online text,
  Mastering the FreeRTOS Real Time Kernel:
  A Hands-On Tutorial Guide

- Homework:
  1) read chapters 1-3 of the Guide
  2) order & receive your LPCXpresso LPC1769 eval kit → digikey.ca can often do next-day deliveries at no extra charge (see link on course github.com main page)
  3) download and install the MCUXpresso IDE from the NXP web site