Embedded System Design and Modeling

IX. Scheduling basics

Scheduling

- Scheduling = the process of arranging the execution of a set of **tasks** which need to be run on the same processing device
  - i.e. decide which task is run when, for how long, etc.
- Encountered in multi-tasking systems
- *Note*: Slides are heavily based on Prabal Dutta & Edward A. Lee, Berkeley 2017

Task

- A **task** is a set of operations which have:
  - release (arrival) time: earliest time when it can be run
  - **start time**: actual starting time
  - **finish time**: actual ending time
  - execution time: actual running time, excluding any interruptions
  - **deadline**: latest time by which a task must be completed
- Tasks may be interrupted by higher priority tasks, when priorities are defined
- Tasks may be periodic (e.g. every 10ms) or aperiodic

Task

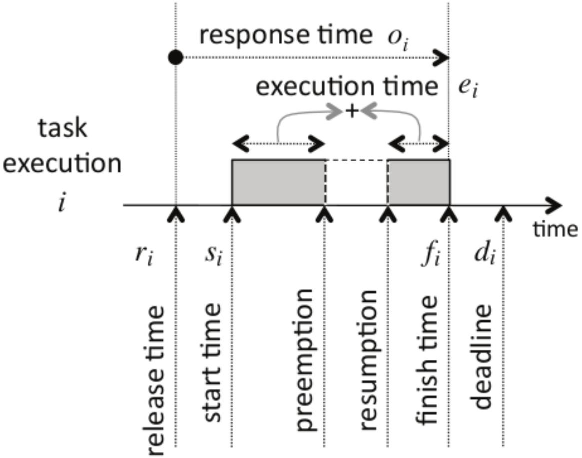

Figure 12.1: Summary of times associated with a task execution.

Figure 1: Task attributes

Scheduling

- How to decide which task to run when?

Considerations:

- Preemptive vs. non-preemptive scheduling
- Periodic vs. aperiodic tasks
- Fixed priority vs. dynamic priority
- Priority inversion anomalies
- Other scheduling anomalies

Preemptive vs. non-preemptive

- Non-premmptive: once started, no task can be interrupted until it finishes
- Preemptive: a task can be interrupted
  - the kernel decides when
- Preemptive scheduling:
  - Every task has a **priority**
  - At any instant, the task with the **highest priority** is executed
  - Any high priority task takes precedence over a low priority task

Rate Monotonic Scheduling (RMS)

- Given N **periodic** tasks
- **Rate Monotonic Scheduling (RMS)**: assign task priority by period: smaller period has higher priority

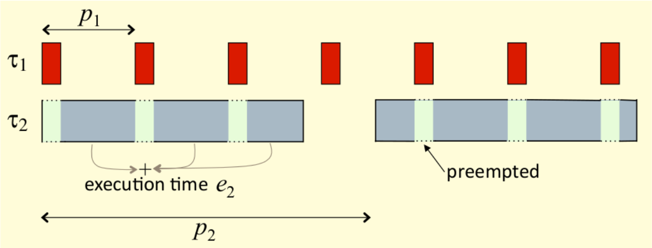

Figure 12.3: Two periodic tasks $T = \{\tau_1, \tau_2\}$ with a preemptive schedule that gives higher priority to τ_1.

1

[1] image from Lee&Sheshia book

Optimality of RMS

- A **feasible schedule** = all task finish times are before their deadlines
  - no deadline is exceeded
- **Theorem**: If the set of N tasks can be arranged to form a feasible schedule, then the RMS scheduling is feasible.

Earliest Deadline First (EDF)

- Given N non-periodic independent tasks with arbitrary arrival times and deadlines
- **Earliest Deadline First** (EDF) scheduling: execute the task with the earliest deadline among all available tasks
- Note: If a new task that just arrived can interrupt the current task, in case it has an earlier deadline

Earliest Deadline First (EDF)

	arrival	duration	deadline
T_1	0	10	33
T_2	4	3	28
T_3	5	10	29

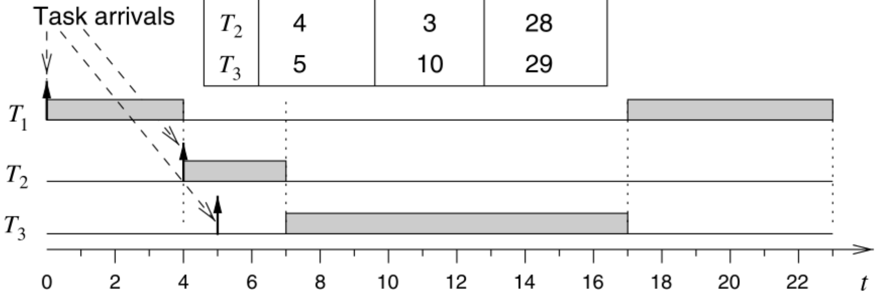

Figure 6.6. EDF schedule

[2]

[2] image from "Embedded System Design" 2nd edition, Peter Marwedel, Springer 2011

Optimality of EDF

- **Theorem**: EDF scheduling minimizes the maximum lateness of the tasks
- The **maximum lateness** of a set of N tasks is:

  $$L_{max} = \max(f_i - d_i)$$

  i.e. the maximum exceeding of a deadline

  (the maximum lateness can be negative, i.e. when no task deadline is exceeded, and in this case it acts as a safety time margin)
- EDF makes the maximum exceeding of deadline as small as possible, or, if no deadline is exceeded, EDF maximizes the safety margin between the finish time and the dealine

Priority Inversion

- Although simple, some complicated and undesired effects might happen (scheduling anomalies)
- **Priority Inversion**: scheduling anomaly where high-priority task is blocked while unrelated lower-priority tasks execute
- Can cause serious problems, such as system resets and data loss
  - Example: Mars Pathfinder mission in 1997.

Priority Inversion

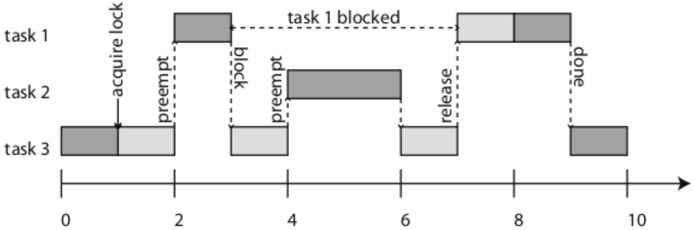

Figure 12.9: Illustration of priority inversion. Task 1 has highest priority, task 3 lowest. Task 3 acquires a lock on a shared object, entering a critical section. It gets preempted by task 1, which then tries to acquire the lock and blocks. Task 2 preempts task 3 at time 4, keeping the higher priority task 1 blocked for an unbounded amount of time. In effect, the priorities of tasks 1 and 2 get inverted, since task 2 can keep task 1 waiting arbitrarily long.

[3]

[3] image from Lee&Sheshia book

Avoiding Priority Inversion

- Options for avoiding priority inversion:
  - Priority inheritance
  - Priority ceiling
  - Priority boosting

Priority inheritance protocol:

- When a task blocks while trying to acquire a lock, the task holding the lock **inherits** the priority of the blocked task.
- This ensures that the task holding the lock cannot be preempted by a task with lower priority than the blocked task.

Priority inheritance

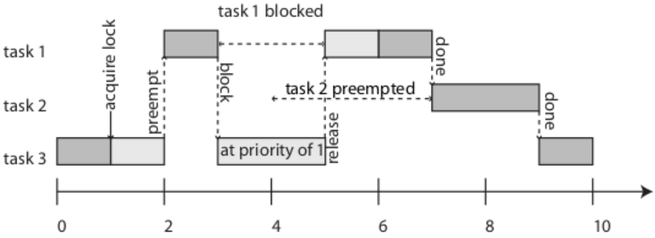

Figure 12.10: Illustration of the priority inheritance protocol. Task 1 has highest priority, task 3 lowest. Task 3 acquires a lock on a shared object, entering a critical section. It gets preempted by task 1, which then tries to acquire the lock and blocks. Task 3 inherits the priority of task 1, preventing preemption by task 2.

[4]

[4] image from Lee&Sheshia book

Deadlock

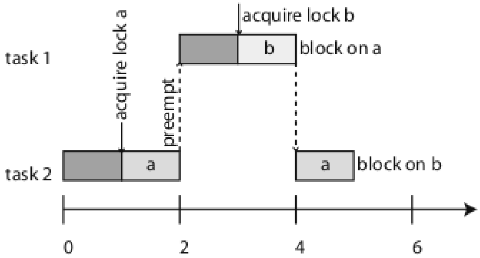

Figure 12.11: Illustration of deadlock. The lower priority task starts first and acquires lock a, then gets preempted by the higher priority task, which acquires lock b and then blocks trying to acquire lock a. The lower priority task then blocks trying to acquire lock b, and no further progress is possible.

5

[5] image from Lee&Sheshia book

Deadlock

- A task holds a lock A and blocks waiting for a lock B
- Another task holds the lock B and blocks waiting for a lock A
- Result: both tasks are blocked for ever

Possible solutions:

- Priority ceiling
- Lock ordering

Priority Ceiling Protocol

- Priorities can be used to prevent certain types of deadlocks
- **Priority ceiling protocol**:
  - Every lock is assigned a priority ceiling, equal to the priority of the highest-priority task that can lock it.
  - A task can acquire a lock only if its priority is **strictly higher** than the priority ceilings of all locks **currently held** by other tasks
- What happens:
  - Suppose all locks can be acquired by any task, so priority ceiling of all tasks is equal to P_{max} = maximum priority among all tasks
  - Suppose one task holds a lock A, so priority ceiling of all locks **currently held** is P_{max}
  - Another task cannot hold another lock B unless its prio is **strictly higher** than P_{max} => impossible => can't lock B => no deadlock

Priority Ceiling Protocol

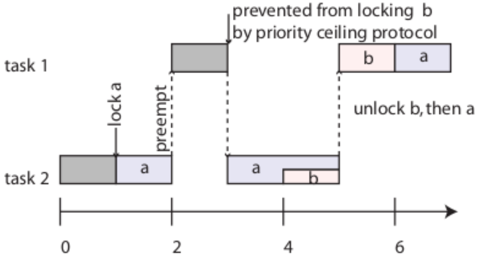

Figure 12.12: Illustration of the priority ceiling protocol. In this version, locks a and b have priority ceilings equal to the priority of task 1. At time 3, task 1 attempts to lock b, but it cannot because task 2 currently holds lock a, which has priority ceiling equal to the priority of task 1.

6

[6] image from Lee&Sheshia book

Priority Ceiling Protocol

Drawbacks:

- Implementing the priority ceiling protocol requires being able to determine in advance which tasks acquire which locks

Lock ordering

Assign each lock a unique numerical value, and require that locks be acquired in increasing order

Example:

- A system with three locks, A, B, and C, and two threads, T1 and T2.
- We have deadlock if T1 holds A and needs B, and T2 holds B and needs A

Solution with lock ordering:

- Assign each lock a unique numerical value: A=1, B=2, and C=3, and require that locks be acquired in increasing order
- T2 is not allowed to acquire B before A, so "*and T2 holds B and needs A*" is impossible
- T2 must first acquire A before it can get B, so it waits for T1 to release A
- Deadlock is avoided