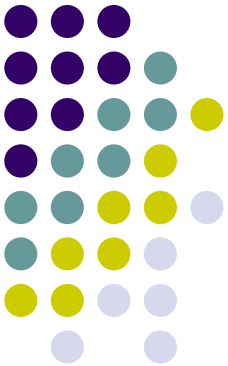
Priority-Driven Scheduling

Ingo Sander

ingo@kth.se



Based on Liu: Chapter 6

Outline

- Assumptions
- Fixed-Priority Algorithms
- Dynamic-Priority Algorithms
- Schedulable utilization and schedulability test
- Optimality of RMA and DMA
- Time demand analysis
- Practical factors

Assumptions

- Periodic tasks are scheduled on single processor
- Tasks are independent
- no aperiodic or sporadic tasks
- Jobs are ready for execution directly after release

Some of the assumptions will be relaxed later

Assumptions

- Scheduling decision are made directly upon job releases or completions
- Jobs can be pre-empted at any time
- Jobs do not suspend themselves
- Context overhead is negligibly small
- Number of priority levels is unlimited

Some of the assumptions will be relaxed later

Static Multiprocessor Systems

- Results from this lecture cannot be used for dynamic multiprocessor systems (jobs can be assigned to any processor)
- BUT they are applicable to each subsystem of a static multiprocessor system
- In static systems, the scheduler for each processor schedules its jobs independently of other schedulers
- Hard-real time systems are usually static, since there are almost no reliable validation techniques for dynamic systems

Priority-driven scheduling

- Priority-driven schedulers are online schedulers
  - Schedule is not pre-computed
  - Scheduler assigns priority to released jobs and places them in ready job queue in priority order
  - At each scheduling decision time, scheduler updates the ready job queue and schedules job with highest priority

Fixed- and Dynamic-Priority

- Task Level
  - Task-level fixed-priority: All jobs in a task have the same priority
  - Task-level dynamic-priority: Jobs in a task may have different priorities
- Job Level
  - Job-level fixed-priority: Once assigned, a priority of a job can never change (usual case)
  - Job-level dynamic-priority: Priority of an individual job may vary

Dynamic-Priority Algorithm

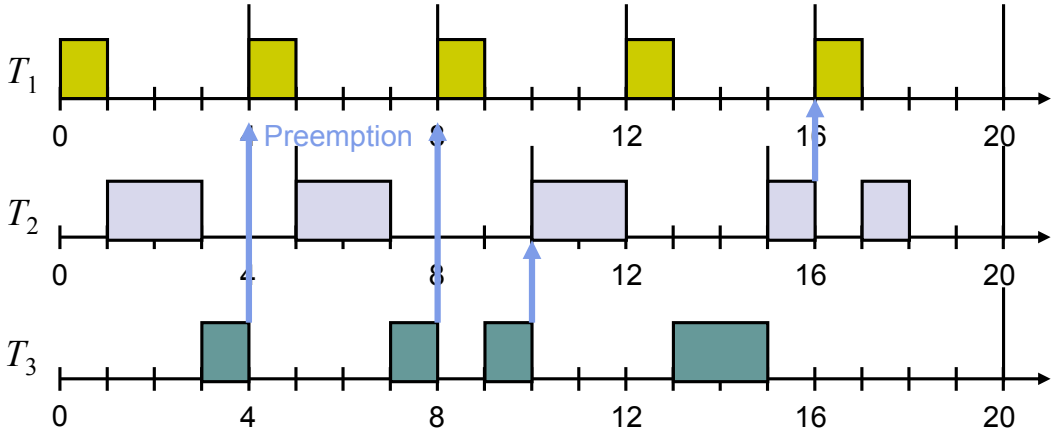
- Except when explicitly stated otherwise, the term *dynamic-priority* algorithm means a *task-level dynamic-priority* and a *job-level fixed-priority*

Rate-Monotonic Algorithm

- most used and studied fixed-priority algorithm
- Priority of tasks is assigned based on their periods
  - The shorter the period the higher the priority
  - *Rate* is inverse of period
- Algorithm is called rate-monotonic algorithm (RMA) and schedule produced by the algorithm rate-monotonic schedule (RMS)

Example Rate-Monotonic Algorithm

- Three tasks T_1 = (4, 1), T_2 = (5, 2), T_3 = (20,5)
- Priorities: T_1 > T_2 > T_3 (Period)
- Hyperperiod: 20



Deadline-Monotonic Algorithm

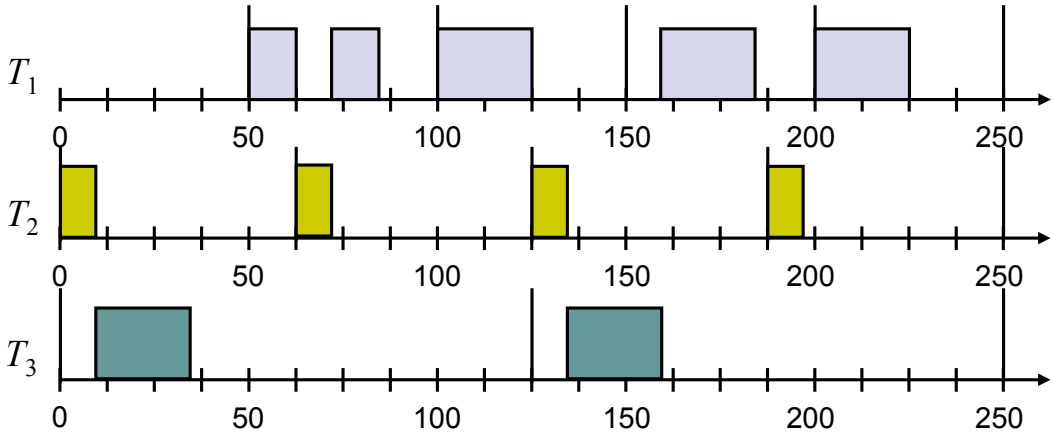
- another well-known fixed-priority algorithm
- Tasks are assigned priorities according to their *relative deadline*
  - shortest relative deadline has highest priority

Example
Deadline Monotonic Algorithm

- Three tasks
  - T_1 = (50, 50, 25, 100)
  - T_2 = (0, 62.5, 10, 20)
  - T_3 = (0, 125, 25, 50)
- Remember notation: (Φ_i, p_i, e_i, D_i)
- Priorities: T_2 > T_3 > T_1 (Relative Deadline)
- Hyperperiod: 250

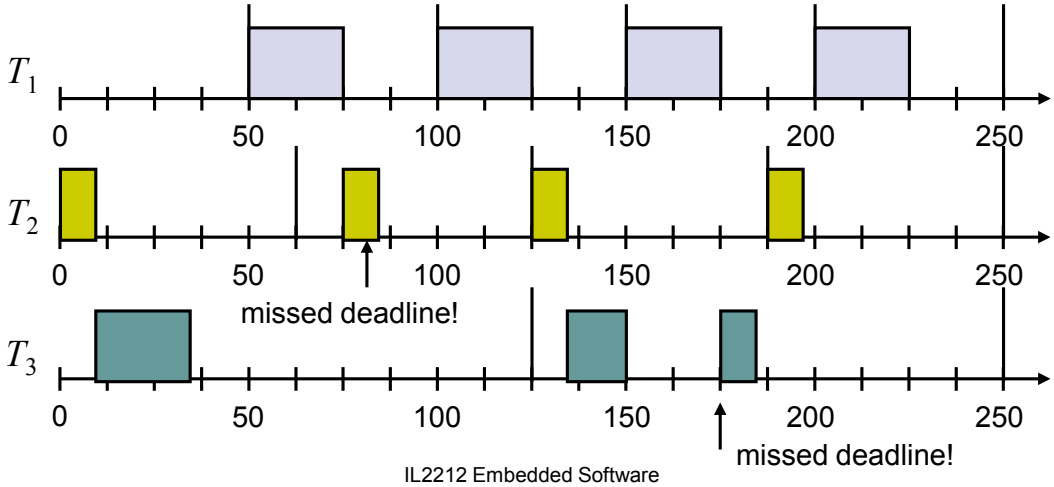
Example
Deadline-Monotonic Algorithm

- T_1 = (50, 50, 25, 100), T_2 = (0, 62.5, 10, 20), T_3 = (0, 125, 25, 50)
- Priorities: T_2 > T_3 > T_1 (Relative Deadline)



Example
Rate-Monotonic Algorithm

- T_1 = (50, 50, 25, 100), T_2 = (0, 62.5, 10, 20), T_3 = (0, 125, 25, 50)
- Priorities: T_1 > T_2 > T_3 (Period)



RMA vs. DMA

- If relative deadline of every task is proportional to its period, RM and DM algorithm are identical
- When relative deadlines are arbitrary, DMA may produce a feasible schedule when RMA fails, but not vice versa

Dynamic-Priority Algorithms

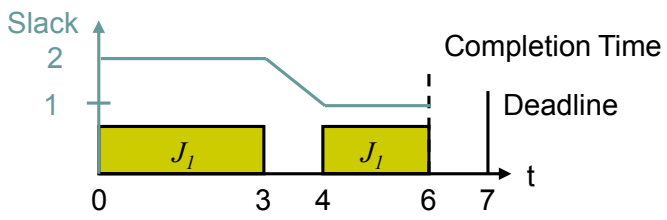
- Earliest Deadline First (EDF)
  - job priority according to their absolute deadlines
  - task-level dynamic, job-level fixed priority
- Least Slack Time First (LST)
  - jobs priority according to size of slack
  - task-level dynamic, job-level dynamic priority

Earliest-Deadline-First (EDF) Algorithm

- Assigns job priorities based on deadlines
- Earlier deadline gives higher priority
- Requires only knowledge of deadlines

Least-Slack-Time-First (LST) Algorithm

- uses the *slack* of a job for priorities (the smaller the slack, the higher the priority)
- at time t, the slack of a job with deadline d is equal to $d - t$ minus the time required to complete the remaining portion of the job
- more complex algorithm, requires also information on deadlines *and* execution times



Optimality of EDF and LST

- The EDF and LST algorithms are optimal under the following conditions
  - If preemption is allowed, the algorithm will always produce a feasible schedule of a set $\boldsymbol{J}$ of jobs with arbitrary release times and deadlines *on one processor*, if and only if $\boldsymbol{J}$ has feasible schedules

Non-Optimality of EDF and LST

- EDF and LST are not optimal, if jobs are non-preemptible or more than one processor is used

Dynamic-Priority Algorithms

- Two versions exist of LST
  - *non-strict* LST
    - Scheduling decisions are only made at release or completion times of jobs
  - *strict* LST
    - Slacks of all ready and executing jobs are monitored all the time and scheduling decisions can be made at any time
    - involves large overhead and is thus not used in practice!

Relative Merits

- Fixed- and dynamic-priority scheduling algorithms have different properties
- EDF and LST outperform RMA and DMA with respect to *schedulable utilization*

Relative Merits

- The behavior of dynamic algorithms is much more difficult to predict
  - difficult to predict, which task misses deadline due to overload
  - late job that misses may cause other jobs to be late
  - good management required for such instable systems

Schedulability Tests

- It is crucial that correctness of schedule can be validated
  - Simulating schedules is time consuming and error prone
  - Other faster methods are needed
- A *schedulability test* is a mechanism that proofs that all deadlines are met, when scheduling with a particular algorithm
  - useful also for acceptance of new online-jobs

Schedulable Utilization

- The *schedulable utilization* U_{ALG} of a scheduling algorithm is defined as follows:
  - A scheduling algorithm can feasibly schedule *any set* of periodic tasks on a processor if the total utilization of the tasks is equal to or less than the schedulable utilization of that algorithm!

Schedulable Utilization

- A periodic task is defined as a tuple (Φ_i, p_i, e_i, D_i) where $u_i = e_i / p_i$
- Total utilization is $U = \Sigma u_i$
- An algorithm is optimal, if $U_{\text{ALG}} = 1$
- Schedulability tests can be developed based on U_{ALG}

Schedulable Utilization

- A system of independent preemptable tasks with relative deadlines equal to its periods can be feasibly scheduled on one processor if and only if its total utilization is equal or less than 1 (Liu: Theorem 6.1)

EDF: Schedulable Utilization

- A system of independent preemptable tasks with relative deadlines equal to its periods can be feasibly scheduled on one processor if and only if its total utilization is equal or less than 1 (Liu: Theorem 6.1)
  - Since EDF and LST are optimal algorithms $U_{\text{EDF}} = 1$ and $U_{\text{LST}} = 1$ for such systems
  - Also $U_{\text{EDF}} = 1$ and $U_{\text{LST}} = 1$ for independent preemptable periodic tasks where $D_i \geq p_i$
  - Result is independent of phase Φ_i

EDF: Schedulable Utilization

- If $D_i < p_i$ for a system with $U \leq 1$ there might not be a feasible schedule any longer
- Test must be extended!

EDF: Schedulable Utilization

- *Density* of task T_i is defined as $\delta_i = e_i / \min(D_i, p_i)$
- *Density* of a system is $\Delta = \Sigma \delta_I$
- Theorem 6.2 (Liu): A system $\mathbf{T}$ of independent, preemptable tasks can feasibly be scheduled, if its density is equal or less to 1
- Theorem 6.2 is a sufficient, but not necessary condition
  - System may nevertheless be feasible when its density is greater than 1 (Simulation can be used to validate schedule)

Schedulable Utilization of RMA

- Theorem 6.11(Liu): A system of n independent, preemptable periodic tasks with relative deadlines equal to their respective periods can be feasibly scheduled on a processor according to the RM algorithm if its total utilization U is less than or equal to

$$U_{\mathrm{RM}}(n) = n(2^{1/n} - 1)$$

Schedulable Utilization of RMA

- For different n:
  - $U_{\mathrm{RM}}(1) = 1$
  - $U_{\mathrm{RM}}(2) = 0.83$
  - $U_{\mathrm{RM}}(3) = 0.78$
  - $U_{\mathrm{RM}}(\infty) = 0.69$
- $U \leq U_{\mathrm{RM}}(n)$ is a sufficient, but not necessary condition!

Optimality of RM and DM

- RM and DM algorithms cannot be optimal, since they assign fixed priorities
  - Not all systems with $U \leq 1$ can be scheduled by RM or DM
- But RM and DM are optimal for *simply periodic* systems
  - A system of periodic tasks is *simply periodic*, if for every pair of tasks T_i and T_k in the system and $p_i < p_k$, p_k is an integer multiple of p_i

Optimality of RM and DM

- Theorem 6.3 (Liu): A system of simply periodic, independent, preemptable tasks whose deadlines are equal or larger than their periods is schedulable on one processor according to the RM algorithm, if and only if its total utilization is equal or less than 1
- Example: **T** = {(2,1), (4,1), (8,2)} is schedulable according to Theorem 6.3

Optimality of DM

- Theorem 6.4 (Liu): A system $\mathbf{T}$ of independent, preemptable tasks that are in phase and have relative deadlines equal or less than their respective periods can be feasibly scheduled on one processor according to the DM algorithm whenever it can feasibly scheduled according to any fixed-priority algorithm

Fixed-Priority Tasks Schedulability Test

- Fixed priority algorithms are predictable
  - Formal analysis possible
- Assumption
  - response time is smaller than or equal to their respective periods
- Analyze the worst case and draw conclusions

Fixed-Priority Tasks Schedulability Test

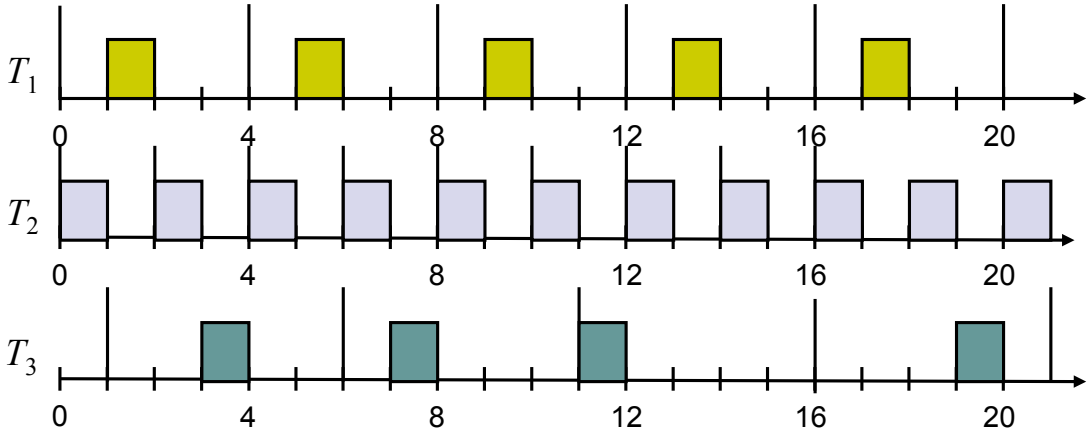
1. Find *critical instant* when system is most loaded and has worst response time
2. Use *time demand analysis* to determine if the system is schedulable at the critical time instant
3. Prove that, if a fixed-priority system is schedulable at the critical instant, it is always schedulable

Critical Instant

- Theorem 6.5 (Liu): In a fixed-priority system where every job completes before the next job in the same task is released, a *critical instant* of any task T_i occurs when one of its job $J_{i,c}$ is released at the same time with a job in every higher-priority task, that is $r_{i,c} = r_{k,l(k)}$ for some $l(k) = 1, 2, \ldots, i-1$

Example
Critical Instant

- Example: T_1 = (4,1), T_2 = (2,1), T_3 = (1,5,1,5)



Critical Instance is 16 for task T_3 and 0, 4, 8, … for task T_1

Time Demand Analysis

- Time demand analysis uses critical instants in order to conduct a schedulability test
  1. Compute a the *total demand for processor time* by a job released at a critical instant of the task and by the higher-priority tasks
  2. Check then if the demand can be met before the deadline of the job

Time Demand Analysis

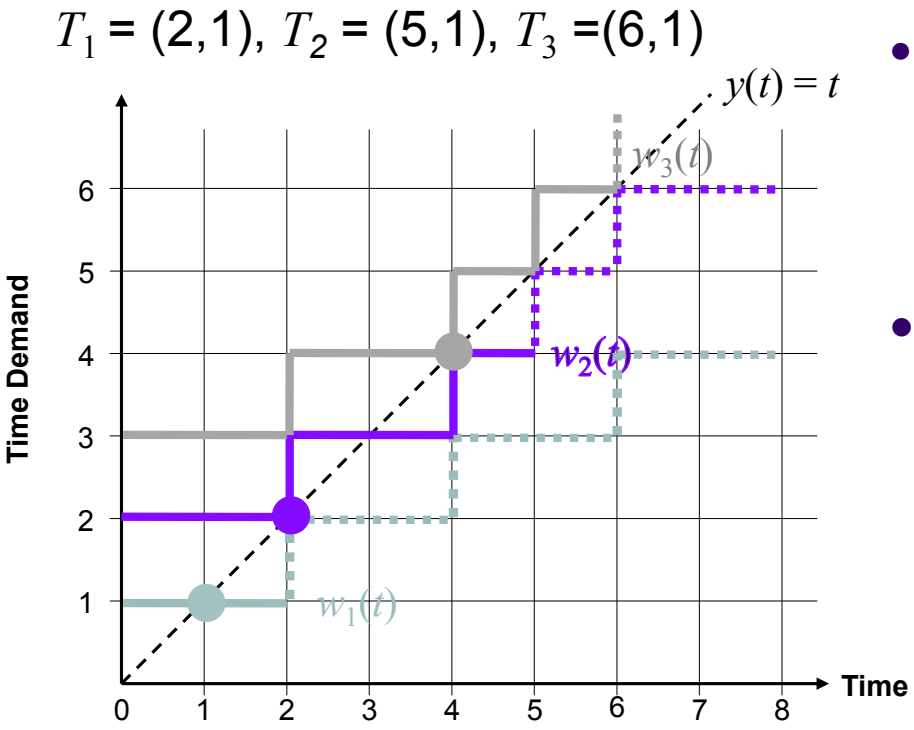
- Consider tasks in descending order
- Focus on job J_i in T_i where the release time t_0 is a critical instant of T_i
- At time $t+t_0$ for $t \geq 0$, the total processor demand $w_i(t)$ of this job and all the higher-priority jobs released in $[t_0, t]$ is given by

$$w_i(t) = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k, \quad \text{for } 0 \leq t \leq p_i$$

Time Demand Analysis

- Compare time demand $w_i(t)$ with available time t
  - If $w_i(t) \leq t$ for some $t \leq D_i$ where $D_i \leq p_i$, all jobs in T_i can complete before its deadline
  - If $w_i(t) > t$ for all $0 \leq t \leq D_i$ where $D_i \leq p_i$, this job in T_i cannot complete by its deadline
    - This is valid under the assumption that the critical instant occurs, but this must not be the case (can be proved by simulation or analysis)
- Check time demand for all tasks
- If all tasks meet their time demand at their critical instant, the system is schedulable

RMA
Time Demand Analysis



- Maximal Response Times
  - $W_1 = 1$
  - $W_2 = 2$
  - $W_3 = 4$
- All tasks meet their deadlines!

Time Demand Analysis

- Schedulability test based on time demand analysis is more complex than the schedulability test based on utilization, but is more general
- Works for any priority-based algorithm
- Can be extended to cover other cases

Alternative Approach

- Simulate behavior of tasks at their critical instants up to the largest period of the tasks
  - Gives same results as time demand analysis

Practical Factors

- So far we have assumed
  - every job is preemptable at any time
  - released jobs never suspend themselves
  - scheduling and context switch overhead is negligible
  - scheduler is event-driven and acts immediately upon event occurrences
  - tasks have distinct priorities
  - priorities in a fixed-priority system never change

Practical Factors

- In practice these assumptions are very often not valid
- Section 6.8 discusses practical factors in some detail
  - Read this!
  - Lecture gives only an overview

Blocking and Priority Inversion

- A (ready) job is blocked when it is prevented from execution by a lower priority job
- Priority inversion occurs whenever a lower priority job executes while some higher-priority job waits

Non-preemptability of Jobs

- Several possibilities why jobs are not preemptable
  - Job accesses a shared resource and shall not be interrupted
  - Job makes a non-preemptable system call
  - Preemption may sometimes be very costly, that is accessed through initialization must be initialized again

Non-preemptability of Jobs

- Priority inversion may occur since high-priority job must wait on non-preemptable low-priority job
- For schedulability test not only the jobs with higher priority, but also the blocking time due to non-preemptable low-priority jobs must be considered

Self-Suspension

- Self-suspension or (self-blocking) can occur when a job invokes an external resource and has to wait until it completes
- Task is then no longer strictly periodic
- Self-suspension time must be taken into account when analyzing the schedule

Blocking
Time Demand Analysis

- Time Demand Analysis can take blocking and self-suspension into account by an additional term blocking time b_i
- $b_i = b_i(np) + b_i(ss)$, where
  - $b_i(np)$: blocking time due to non-preemptability
  - $b_i(ss)$: blocking time due to self-suspension

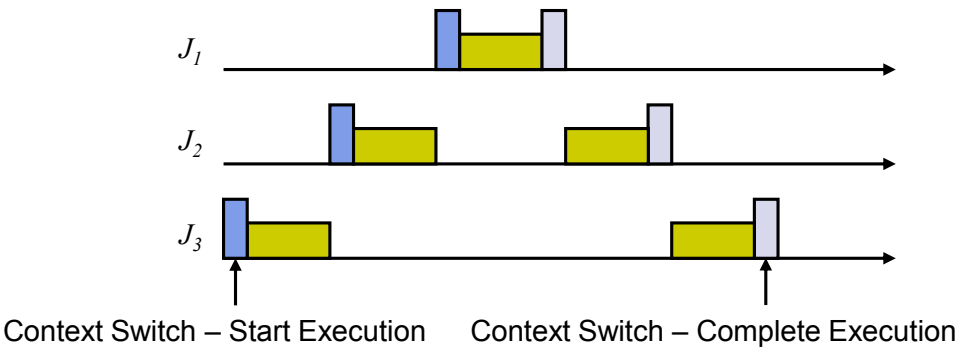
$$w_i(t) = e_i + b_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k, \quad \text{for } 0 \le t \le p_i$$

Context Switch Times

- Context switches take their time according to context switch overhead
- To account for context-switch, the time spend for context-switch time can be added to the execution time of a job

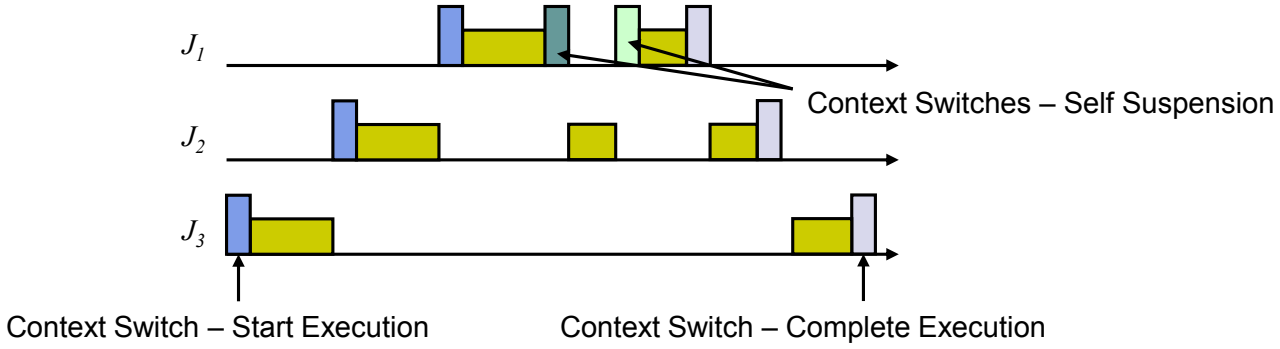
Context Switch Times

- If self-suspension is not considered
  - each job suffers at most one context switch when its *starts execution* and one when it *completes*
  - compensate for this by increasing the execution time of each task to, $e_{CS,i} = e_i + 2CS$



Context Switch Times

- If self-suspension is considered
  - If a job self-suspends, the job incurs two more context switches each time it self-suspends
  - compensate for K_i self-suspensions by increasing the execution time to $e_{CS,i} = e_i + 2(K_i + 1)CS$



Tick Scheduling

- Assumption was that scheduling of a job was event-driven
  - a scheduling event like the release or completion of a job could immediately cause an execution of the scheduler
- Schedulers are often implemented with a hardware timer using timer interrupts
  - scheduler decisions are only made at certain time instances

Tick Scheduling

- Additional factors
  - Scheduler notices ready job first at the next timer interrupt => Execution will be delayed
  - Jobs that get ready between timer interrupts have to be put into special *pending job* queue
    - When scheduler executes, it moves jobs from pending queue to ready queue
  - These additional times can change the timely behavior a lot

Tick Scheduling

- The effects of tick scheduling can be integrated into time demand analysis
- Scheduler can be modeled as a periodic task $T_0 = (p_0, e_0)$
  - p_0 denotes the *tick size* (time between clock interrupts)
  - CS_0 denotes the maximum amount of time that the scheduler needs to move a job from the pending job queue to the ready job queue

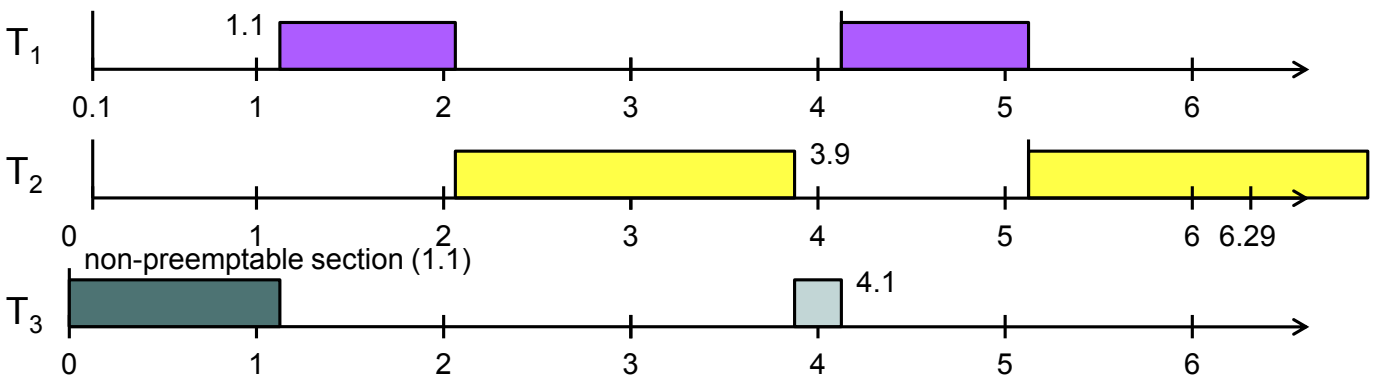
Tick Scheduling

- Assumption
  - Here we assume that if jobs are already in the ready queue, they can be scheduled in priority order without intervention of the scheduler

Example Tick Scheduling (Liu, p. 169)

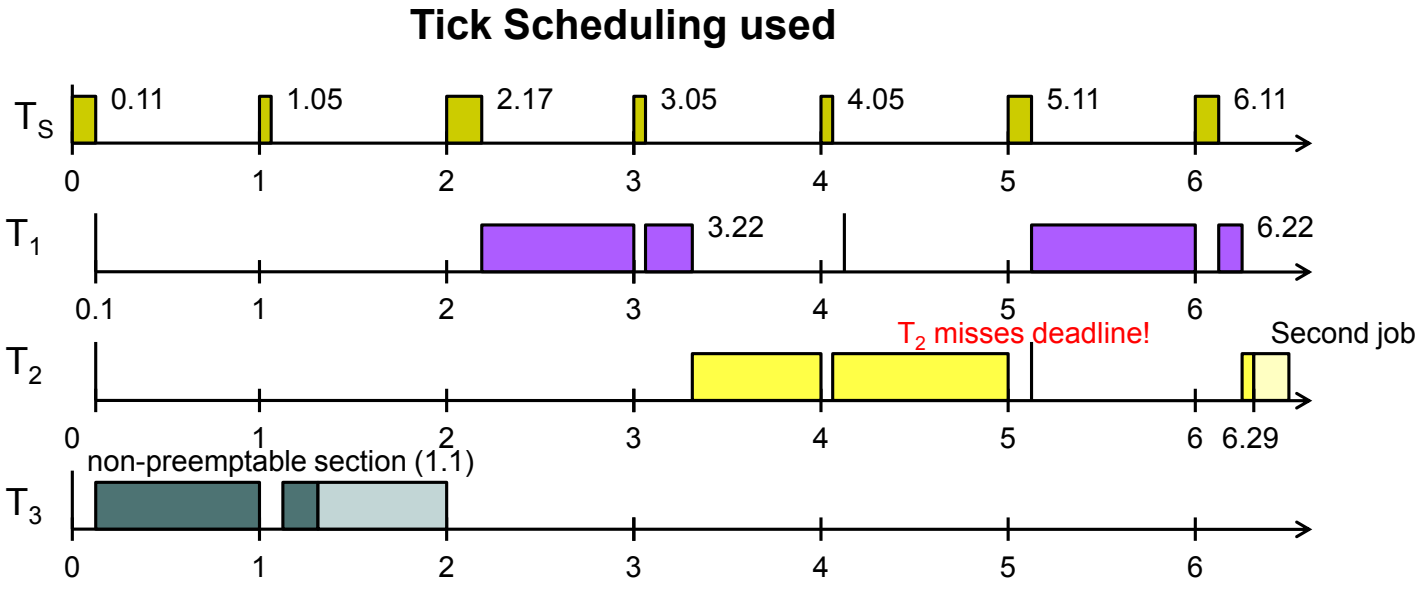
- T_1=(0.1, 4,1,4), T_2=(0.1, 5, 1.8, 5), T_3=(20,5)

Tick Scheduling not used



Example Tick Scheduling (Liu, p. 169)

T_1=(0.1, 4, 1, 4), T_2=(0.1, 5, 1.8, 5), T_3=(20,5), p_0=1, e_0=0.05, CS_0= 0.06



Tick Scheduling

- For fixed-priority systems the time-demand function of a task T_i is modified as follows:
  1. include the task T_0 = (p_0, e_0) in the set of higher-priority tasks
  2. add $(K_k + 1)CS_0$ to the execution time e_k of every higher-priority task ($k = 1,2,3,\ldots$), where K_k is the number of times T_k may self-suspend

Tick Scheduling

3. for every lower-priority task $T_{k,}$ $k = i+1,\ldots, n$ add a task (p_k, CS_0) in the set of higher-priority tasks
4. make the blocking time $b_i(np)$ due to non-preemptibility of T_i equal to

$$\left(\left\lceil \max_{i+1 \le k \le n} \theta_k / p_0 \right\rceil + 1\right) p_0$$

where θ_k is the maximum execution time of non-preemptable sections of the lower–priority task of T_k

Time Demand Analysis is pessimistic!

Summary

- Dynamic-priority algorithms have a higher schedulable utilization than fixed-priority algorithms
- Behavior of dynamic priority algorithms is difficult to predict
- Schedulability tests allow to validate algorithms without simulation

Summary

- Time Demand Analysis can be used as schedulability test for fixed-priority systems
- Practical factors have to be taken into account, since they can change the results of a theoretical fixed-priority schedule considerably