UNIT-3

Priority-Driven Scheduling of Periodic Tasks

STATIC ASSUMPTION:

we recall that a multiprocessor priority-driven system is either dynamic or static. In a static system, all the tasks are partitioned into subsystems. Each subsystem is assigned to a processor, and tasks on each processor are scheduled by themselves. In contrast, in a dynamic system, jobs ready for execution are placed in one common priority queue and dispatched to processors for execution as the processors become available.

The dynamic approach should allow the processors to be more fully utilized on average as the workload fluctuates. Indeed, it may perform well most of the time. However, in the worst case, the performance of priority-driven algorithms can be unacceptably poor. A simple example demonstrates this fact. The application system contains $m + 1$ independent periodic tasks. The first m tasks T_i, for $i = 1, 2, \ldots m$, are identical. Their periods are equal to 1, and their execution times are equal to 2ε, where ε is a small number. The period of the last task T_{m+1} is $1+\varepsilon$, and its execution time is 1. The tasks are in phase. Their relative deadlines are equal to their periods.

It is arguable that the poor behavior of dynamic systems occurs only for some pathological system configurations, and some other algorithms may perform well even for the pathological cases. In most cases, the performance of dynamic systems is superior to static systems.

The more troublesome problem with dynamic systems is the fact that we often do not know how to determine their worst-case and best-case performance. The theories and algorithms presented in this and subsequent chapters make it possible for us to validate efficiently, robustly, and accurately the timing constraints of static real-time systems characterizable by the periodic task model.

For these reasons, most hard real-time systems built and in use to date and in the near future are static. In the special case when tasks in a static system are independent, we can consider the tasks on each processor independently of the tasks on the other processors. The problem of scheduling in multiprocessor and distributed systems is reduced to that of uniprocessor scheduling. In general, tasks may have data and control dependencies and may share resources on different processors

FIXED-PRIORITY VERSUS DYNAMIC-PRIORITY ALGORITHMS

Priority-driven algorithms differ from each other in how priorities are assigned to jobs. We classify algorithms for scheduling periodic tasks into two types: fixed priority and dynamic priority.

A ***fixed-priority*** algorithm assigns the same priority to all the jobs in each task. other words, the priority of each periodic task is fixed relative to other tasks.

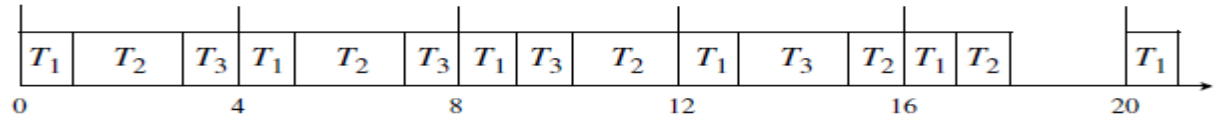
In contrast, a ***dynamic-priority*** algorithm assigns different priorities to the individual jobs in each task. Hence the priority of the task with respect to that of the other tasks changes as jobs are released and completed. This is why this type of algorithm is said to be "dynamic."

Rate-Monotonic and Deadline-Monotonic Algorithms:

A well-known fixed-priority algorithm is the *rate-monotonic* algorithm. This algorithm assigns priorities to tasks based on their periods: the shorter the period, the higher the priority. The *rate* (of job releases) of a task is the inverse of its period. Hence, the higher its rate, the higher its priority. We will refer to this algorithm as the RM algorithm for short and a schedule produced by the algorithm as an RM schedule.

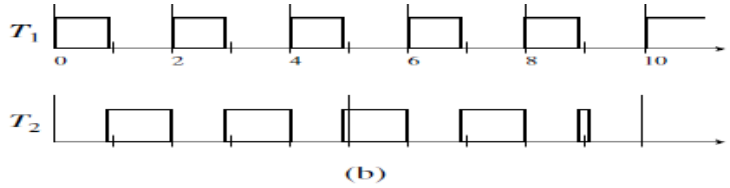
Example:

The system contains three tasks: $T1 = (4, 1)$, $T2 = (5, 2)$, and $T3 = (20, 5)$.



(a)

Figure (a) is for The priority of $T1$ is the highest because its rate is the highest (or equivalently, its period is the shortest). Each job in this task is placed at the head of the priority queue and is executed as soon as the job is released. $T2$ has the next highest priority. Its jobs execute in the background of $T1$. For this reason, the execution of the first job in $T2$ is delayed until the first job in $T1$ completes, and the third job in $T2$ is preempted at time 16 when the fourth job in $T1$ is released. Similarly, $T3$ executes in the background of $T1$ and $T2$; the jobs in $T3$ execute only when there is no job in the higher-priority tasks ready for execution. Since there is always at least one job ready for execution until time 18, the processor never idles until that time.

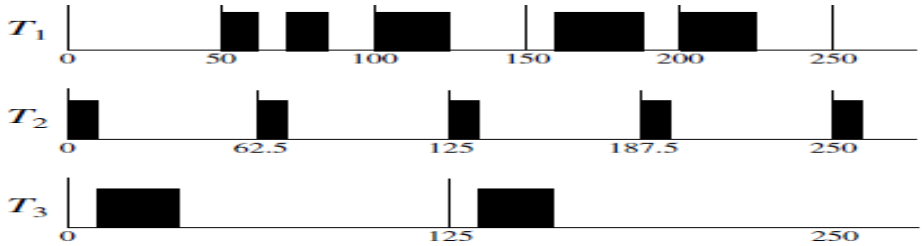


(b)

The schedule in Figure (b) is for the tasks $T1 = (2, 0.9)$ and $T2 = (5, 2.3)$. The tasks are in phase. Here, we represent the schedule in a different form. Instead of using onetime line, sometimes called a Gantt chart, to represent a schedule on a processor as we have done thus far, we use a time line for each task. Each time line is labeled at the left by the name of a task; the time line shows the intervals during which the task executes. According to the RM algorithm, task $T1$ has a higher-priority than task $T2$. Consequently, every job in $T1$ is scheduled and executed as soon as it is released. The jobs in $T2$ are executed in the background of $T1$.

Another well-known fixed-priority algorithm is the ***deadline-monotonic algorithm***, called the DM algorithm hereafter. This algorithm assigns priorities to tasks according their relative deadlines: the shorter the relative deadline, the higher the priority.

Example: The system consists of three tasks. They are $T1 = (50, 50, 25, 100)$, $T2 = (0, 62.5, 10, 20)$, and $T3 = (0, 125, 25, 50)$. Their utilizations are 0.5, 0.16, and 0.2, respectively. The total utilization is 0.86. According to the DM algorithm, $T2$ has the highest priority because its relative deadline 20 is the shortest among the tasks. $T1$, with a relative deadline of 100, has the lowest priority. The resultant DM schedule is shown in Below Figure. According to this schedule, all the tasks can meet their deadlines.



When the relative deadline of every task is proportional to its period, the RM and DM algorithms are identical. When the relative deadlines are arbitrary, the DM algorithm performs better in the sense that it can sometimes produce a feasible schedule when the RM algorithm fails, while the RM algorithm always fails when the DM algorithm fails.

Well-Known Dynamic Algorithms

The EDF algorithm assigns priorities to individual jobs in the tasks according to their absolute deadlines; it is a dynamic-priority algorithm

Another well-known dynamic-priority algorithm is the *Least-Slack-Time-First* (LST) algorithm. You recall that at time t , the slack of a job whose remaining execution time (i.e., the execution of its remaining portion) is x and whose deadline is d is equal to $d - t - x$. The scheduler checks the slacks of all the ready jobs each time a new job is released and orders the new job and the existing jobs on the basis of their slacks: the smaller the slack, the higher the priority

Coincidentally, the schedule of $T1$ and $T2$ in the above example produced by the LST algorithm happens to be identical to the EDF schedule

Relative Merits

Algorithms that do not take into account the urgencies of jobs in priority assignment usually perform poorly. Dynamic-priority algorithms such as FIFO and LIFO are examples.

The criterion we will use to measure the performance of algorithms used to schedule periodic tasks is the schedulable utilization. The *schedulable utilization* 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 the algorithm*.

While by the criterion of schedulable utilization, optimal dynamic-priority algorithms outperform fixed-priority algorithms, an advantage of fixed-priority algorithms is predictability. The timing behavior of a system scheduled according to a fixed-priority algorithm is more predictable than that of a system scheduled according to a dynamic-priority algorithm.

The EDF algorithm has another serious disadvantage. We note that a late job which has already missed its deadline has a higher-priority than a job whose deadline is still in the future.

MAXIMUM SCHEDULABLE UTILIZATION:

A system is *schedulable* by an algorithm if the algorithm always produces a feasible schedule of the system. A system is schedulable (and *feasible*) if it is schedulable by some algorithm, that is, feasible schedules of the system exist.

Schedulable Utilizations of the EDF Algorithm:

We first focus on the case where the relative deadline of every task is equal to its period. The following theorem Describes that any such system can be feasibly scheduled if its total utilization is equal to or less than one, no matter how many tasks there are and what values the periods and execution times of the tasks are. In the proof of this and later theorems, we will use the following terms. At any time *t* , the *current period* of a task is the period that begins before *t* and ends at or after *t* .We call the job that is released in the beginning of this period the *current job*.

THEOREM 1: A system *T* of independent, preemptable tasks with relative deadlines equal to their respective periods can be feasibly scheduled on one processor if and only if its total utilization is equal to or less than 1

The following facts follow straightforwardly from this theorem.
1. A system of independent, preemptable periodic tasks with relative deadlines longer than their periods can be feasibly scheduled on a processor as long as the total utilization is equal to or less than 1.
2. The schedulable utilization UEDF (n) of the EDF algorithm for n independent, preemptable Periodic tasks with relative deadlines equal to or larger than **their periods are equal to 1.**
THEOREM 2. A system **T** of independent, preemptable tasks can be feasibly scheduled on one processor if its density is equal to or less than 1.
Proof: The condition given by this theorem is not necessary for a system to be feasible. A system may nevertheless be feasible when its density is greater than 1. The system consisting of *(*2, 0.6, 1*)* and *(*5, 2.3*)* is an example. Its density is larger than 1, but it is schedulable according to the EDF algorithm.

Schedulability Test for the EDF Algorithm:

We call a test for the purpose of validating that the given application system can indeed meet all its hard deadlines when scheduled according to the chosen scheduling algorithm a *schedulability test*. If a schedulability test is efficient, it can be used as an on-line acceptance test.

Checking whether a set of periodic tasks meet all their deadlines is a special case of the validation problem that can be stated as follows: We are given

1. The period p_i , execution time e_i , and relative deadline D_i of every task T_i in a system $\mathbf{T} = \{T_1, T_2, \ldots T_n\}$ of independent periodic tasks, and

2. A priority-driven algorithm used to schedule the tasks in **T** preemptively on one processor.

OPTIMALITY OF THE RM AND DM ALGORITHMS

In fixed-priority scheduling, we index the tasks in decreasing order of their priorities except where stated otherwise. In other words, the task T_i has a higher priority than the task T_k if $i < k$.

In Fixed priority they assign fixed priorities to tasks, fixed-priority algorithms cannot be optimal: Such an algorithm may fail to schedule some systems for which there are feasible schedules. To demonstrate this fact, we consider a system which consists of two tasks: $T_1 = (2, 1)$ and $T_2 = (5, 2.5)$. Since their total utilization is equal to 1, we know from Theorem1 that the tasks are feasible. $J_{1,1}$ and $J_{1,2}$ can complete in time only if they have a higher priority than $J_{2,1}$. In other words, in the time interval $(0, 4]$, T_1 must have a higher-priority than T_2. However, at time 4 when $J_{1,3}$ is released, $J_{2,1}$ can complete in time only if T_2 (i.e., $J_{2,1}$) has a higher priority than T_1 (i.e., $J_{1,3}$). This change in the relative priorities of the tasks is not allowed by any fixed priority algorithm.

While the RM algorithm is not optimal for tasks with arbitrary periods, it is optimal in the special case when the periodic tasks in the system are simply periodic and the deadlines of the tasks are no less than their respective periods. 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 . An example of a simply periodic task system is the flight control system.The RM algorithm is optimal among all fixed-priority algorithms whenever the relative deadlines of the tasks are proportional to their periods.

A SCHEDULABILITY TEST FOR FIXED-PRIORITY TASKS WITH SHORT RESPONSE TIMES

Every job completes before the next job in the same task is released. We will consider the general case where the response times may be larger than the periods in the next section. Since no system with total utilization greater than 1 is schedulable, we assume hereafter that the total utilization U is
equal to or less than 1.

Critical Instants:

critical instant of a task T_i is a time instant which is such that

1. The job in T_i released at the instant has the maximum response time of all jobs in T_i, if the response time of every job in T_i is equal to or less than the relative deadline D_i of T_i, and

2. The response time of the job released at the instant is greater than D_i if the response time of some jobs in T_i exceeds D_i . We call the response time of a job in T_i released at a critical instant the *maximum* (*possible*) *response time* of the task and denote it by W_i . The following theorem gives us the condition under which a critical instant of each task T_i occurs

Time-Demand Analysis

To determine whether a task can meet all its deadlines, we first compute the total demand for processor time by a job released at a critical instant of the task and by all the higher-priority tasks as a function of time from the critical instant. We then check whether this demand can be met before the deadline of the job. For this reason, we name this test a *time-demand analysis*.

To carry out the time-demand analysis on **T**, we consider one task at a time, starting from the task $T1$ with the highest priority in order of decreasing priority. To determine whether a task T_i is schedulable after finding that all the tasks with higher priorities are schedulable, we focus on a job in T_i , supposing that the release time $t0$ of the job is a critical instant of T_i . At time $t0 + t$ for $t \geq 0$, the total (processor) time demand $w_i(t)$ of this job and all the higher-priority jobs released in $[t0, t]$ is given by

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

This job of T_i can meet its deadline $t0 + D_i$ if at some time $t0 + t$ at or before its deadline, the supply of processor time, which is equal to t , becomes equal to or greater than the demand $w_i(t)$ for processor time. In other words, $w_i(t) \leq t$ for some $t \leq D_i$, where D_i is equal to or less than p_i . Because this job has the maximum possible response time of all jobs in T_i , we can conclude that all jobs in T_i can complete by their deadlines if this job can meet its deadline.

Alternatives to Time-Demand Analysis:

Instead of carrying out a time-demand analysis, we can also determine whether a system of independent preemptable tasks is schedulable by simply simulating this condition and observing whether the system is then schedulable. In other words, a way to test the schedulability of such a system is to construct a schedule of it according to the given scheduling algorithm. In this construction, we assume that the tasks are in phase and the actual execution times and inter release times of jobs in each task T_i are equal to e_i and p_i , respectively.

SCHEDULABILITY TEST FOR FIXED-PRIORITY TASKS WITH ARBITRARY RESPONSE TIMES

The time-demand analysis method developed by Lehoczky to determine the schedulability of tasks whose relative deadlines are larger than their respective periods. Since the response time of a task may be larger than its period, it may have more than one job ready for execution at any time. Ready jobs in the same task are usually scheduled on the FIFO basis

Busy Intervals

We will use the term level-πi busy interval. A *level-πi busy interval* $(t0, t]$ begins at an instant $t0$ when (1) all jobs in $\mathbf{T}i$ released before the instant have completed and (2) a job in $\mathbf{T}i$ is released. The interval ends at the first instant t after $t0$ when all the jobs in $\mathbf{T}i$ released since $t0$ are complete. In other words, in the interval $(t0, t]$, the processor is busy all the time executing jobs with priorities πi or higher, all the jobs executed in the busy interval are released in the nterval, and at the end of the interval there is no backlog of jobs to be executed afterwards. Hence, when computing the response

times of jobs in $\mathbf{T}i$, we can consider every level-πi busy interval independently from other level-πi busy intervals.

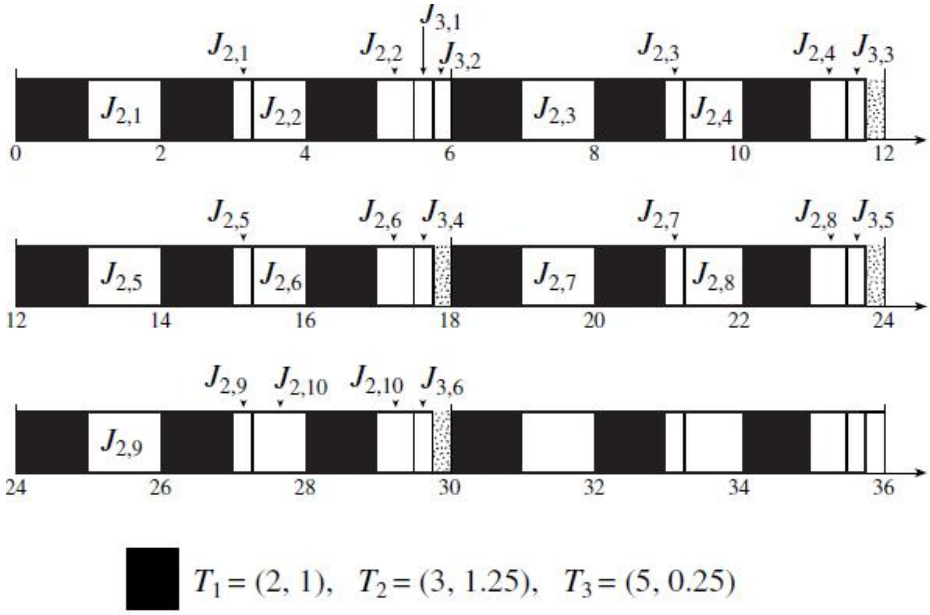
With a slight abuse of the term, we say that a level-πi busy interval is *in phase* if the first jobs of all tasks that have priorities equal to or higher than priority πi and are executed in this interval have the same release time. Otherwise, we say that the tasks have arbitrary phases in the interval.

Example: the schedule of three tasks $T1 = (2, 1)$, $T2 = (3, 1.25)$, and $T3 = (5, 0.25)$ in the first hyperperiod. The filled rectangles depict where jobs in $T1$ are scheduled. The first busy intervals of all levels are in phase. The priorities of the tasks are $\pi 1 = 1$, $\pi 2 = 2$, and $\pi 3 = 3$, with 1 being the highest priority and 3 being the lowest priority. As expected, every level-1 busy interval always ends 1 unit time after it begins.

For this system, all the level-2 busy intervals are in phase. They begin at times 0, 6, and so onwhich are the least common multiples of the periods of tasks $T1$ and $T2$. The lengths of these intervals are all equal to 5.5. Before time 5.5, there is at least one job of priority 1 or 2 ready for execution, but immediately after 5.5, there are none.

Hence at 5.5, the first job in $T3$ is scheduled. When this job completes at 5.75, the second job in $T3$ is scheduled. At time 6, all the jobs released before time 6 are completed; hence, the first level-3 busy interval ends at this time.

The second level-3 busy interval begins at time 6. This level-3 busy interval is not in phase since the release times of the first higher-priority jobs in this interval are 6, but the first job of $T3$ in this interval is not released until time 10. The length of this level-3 busy interval is only 5.75. Similarly, all the subsequent level-3 busy intervals in the hyperperiod have arbitrary phases.



General Schedulability Test

The general schedulability test described below relies on the fact that when determining the schedulability of a task Ti in a system in which the response times of jobs can be larger than their

respective periods, it still suffices to confine our attention to the special case where the asks are in phase.

General Time-Demand Analysis Method

Test one task at a time starting from the highest priority task $T1$ in order of decreasing priority. For the purpose of determining whether a task Ti is schedulable, assume that all the tasks are in phase and the first level-πi busy interval begins at time 0.

Correctness of the General Schedulability Test

The general schedulability test described above makes a key assumption: The maximum response time of some job Ji, j in an in-phase level-πi busy interval is equal to the maximum possible response time of all jobs in Ti . Therefore, to determine whether Ti is schedulable, we only need to check whether the maximum response times of all jobs in this busy interval are no greater than the relative deadline of Ti . This subsection presents several lemmas as proof that this assumption is valid.

SUFFICIENT SCHEDULABILITY CONDITIONS FOR THE RM AND DM ALGORITHMS

When we know the periods and execution times of all the tasks in an application system, we can use the schedulability test described in the last section to determine whether the system is schedulable according to the given fixed-priority algorithm. However, before we have completed the design of the application system, some of these parameters may not be known. In fact, the design process invariably involves the trading of these parameters against each other. We may want to vary the periods and execution times of some tasks within some range of values for which the system remains feasible in order to improve some aspects of the system.

Schedulable Utilization of the RM Algorithm for Tasks with *Di = pi* Specifically, the following theorem from [LiLa] gives us a schedulable utilization of the RM algorithm. We again focus on the case when the relative deadline of every task is equal to its period. For such systems, the RM and DM algorithms are identical.

THEOREM: 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

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

URM(n) is the schedulable utilization of the RM algorithm when $Di = pi$ for all $1 \leq k \leq n$.

Generalization to Arbitrary Period Ratios. The ratio $qn,1 = pn/p1$ is the *period ratio* of the system. To complete the proof of Theorem 6.11, we must show that any n-task system whose total utilization is no greater than *URM(n)* is schedulable rate-monotonically, not just systems whose period ratios are less than or equal to 2. We do so by showing that the following two facts are true.

1. Corresponding to every difficult-to-schedule n-task system whose period ratio is larger than 2 there is a difficult-to-schedule n-task system whose period ratio is less than or equal to 2.

2. The total utilization of the system with period ratio larger than 2 is larger than the total utilization of the corresponding system whose period ratio is less than or equal to 2.

Schedulable Utilization of RM Algorithm as Functions of Task Parameters

When some of the task parameters are known, this information allows us to improve the schedulable utilization of the RM algorithm.We now give several schedulable utilizations that are larger than *URM(n)* for independent, preemptive periodic tasks whose relative deadlines are equal to their respective periods. These schedulable utilizations are expressed in terms of known parameters of the tasks, for example, the utilizations of individual tasks, the number *nh* of disjoint subsets each containing simply periodic tasks, and some functions of the periods of the tasks. The general schedulable utilization *URM(n)* of the RM algorithm is the minimum value of these specific schedulable utilizations. Because they are larger than *URM(n)*, when applicable, these schedulable utilizations are more accurate criteria of schedulability

Schedulable Utilization of Fixed Priority Tasks with Arbitrary Relative Deadlines

A system of *n* tasks with a total utilization *URM(n)* may not be schedulable ratemonotonically when the relative deadlines of some tasks are shorter than their periods. On the other hand, if the relative deadlines of the tasks are larger than the respective task periods, we expect the schedulable utilization of the RM algorithm to be larger than *URM(n)*.

Schedulable Utilization of the RM Algorithm for Multiframe Tasks

The *multiframe task model* developed by Mok and Chen [MoCh] is a more accurate model and leads to more accurate schedulability tests. The example used by Mok and Chen to motivate the multiframe task model is a system of two tasks: *T*1 and *T*2. *T*2 is a task with period 5 and execution time 1. The period of *T*1 is 3. The maximum execution time of *J*1,*k* is equal to 3, if *k* is odd and is equal to 1 if *k* is EVEN. The relative deadlines of the tasks are equal to their respective periods. We can treat

*T*1 as the periodic task *(*3, 3*)*, but if we were to do so, we would conclude that the system is not schedulable. This conclusion would be too pessimistic because the system is in fact schedulable. Indeed, as we will see shortly, by modeling *T*1 more accurately as a multiframe task, we can reach the correct conclusion.

Specifically, in the multiframe task model, each (multiframe) task *Ti* is characterized by a 4-tuple $(p_i, \xi_i, e_i^p, e_i^n)$ In the 4-tuple, *pi* is the period of the task and has the same meaning as the period of a periodic task. Jobs in *Ti* have either one of two possible maximum execution times: *ei p* and *ei n*, where $e_i^p \geq e_i^n$. . The former is its *peak execution time*, and the latter is its *normal execution time*. Each period which begins at the release time of a job with the peak execution time is called a *peak frame*, and the other periods are called *normal frames*. Each peak frame is followed by $\xi_i - 1$ normal frames, which in turn are followed by a peak frame and so on.