Technical Disclosure Commons

Defensive Publications Series

October 2020

Addressing Priority Inversion In Dynamic Priority Schedulers

Corey Edward Tabaka

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation
Tabaka, Corey Edward, "Addressing Priority Inversion In Dynamic Priority Schedulers", Technical Disclosure Commons, (October 29, 2020)
https://www.tdcommons.org/dpubs_series/3717

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.

Addressing Priority Inversion In Dynamic Priority Schedulers

ABSTRACT

Dynamic priority schemes may be employed in real-time schedulers in which tasks meet specific deadlines. For example, in an earliest deadline first (EDF) scheme, the task with the earliest deadline is considered the most important. As time passes, the relative priorities of tasks are dynamic, e.g., they change based on their proximity to their respective deadlines. Priority inversion, which takes place when a low-priority task preempts a high-priority task, e.g., by locking a resource needed by the high-priority task, is difficult to handle in dynamic priority schedulers. This disclosure describes techniques to forestall or mitigate priority inversion in dynamic priority schemes. Per the techniques, a low-priority task that blocks higher priority tasks from running due to its owning a lock on a resource needed by the higher priority tasks is granted a new deadline and capacity such that the lock owner has the same overall opportunity to run as the lock contenders, were they not blocked on the lock. In this way, the lock-owing task gets sufficient time to execute while avoiding a priority inversion.

KEYWORDS

- Priority inversion
- Real-time scheduling
- Deadline scheduling
- Task scheduling
- Operating system scheduler
- Earliest deadline first
- Priority inheritance
- Deadline inheritance

BACKGROUND

Priority inversion takes place when a low-priority task preempts a high-priority task, e.g., by locking a resource needed by the high-priority task. Priority inheritance is a standard solution to the priority inversion problem in task scheduling in operating systems. Most schedulers employ a static priority scheme, where threads are assigned a numeric value that indicates importance. Different priority values may be assigned to tasks to favor the execution of specific work; however, once configured, these values usually remain fixed. Moreover, the importance of a task doesn't change with respect to time as a function of the scheduling algorithm.

In a static priority scheduler, priority inheritance involves selecting the highest priority contender for a lock and elevating the owner of the lock to that priority. This prevents the owner of the lock from being starved by higher priority tasks and unnecessarily delaying contenders for the lock. This scheme is successfully implemented in many contemporary operating systems.

In contrast to static priority schemes, dynamic priority schemes may be used in real-time schedulers in which tasks are associated with specific deadlines. In a dynamic priority scheme, the importance of a task changes with respect to time as a function of the scheduling algorithm. An example of such an algorithm is the earliest deadline first (EDF). In this algorithm, the task with the earliest deadline is considered the most important. If a new task arrives with a deadline earlier than the currently executing task, the scheduler switches to the task with the earlier deadline. Additionally, if the deadline for the currently executing task passes, then the task with the next earliest deadline is selected to run.

Priority inheritance in a dynamic priority scheme is difficult compared to priority inheritance in static priority schedulers. Because the importance of a task evolves over time, it is

no longer a simple matter of selecting the lock contender with the highest priority for the lock owner to inherit; as time passes, the most important contender changes.

DESCRIPTION

This disclosure describes techniques to forestall or mitigate priority inversion in dynamic priority schemes. The techniques apply to a variety of scheduling algorithms, including the earliest deadline first algorithm.

Each of N contending tasks P_i ($1 \leq i \leq N$) has two parameters, the capacity C_i and the relative deadline, or period, D_i. These two parameters have time units that represent respectively how long a task may run per period and the period between runs. For example, if task P_1 has parameters C_1=1 ms and D_1=3 ms, then the task may run for at most 1 ms out of every 3 ms.

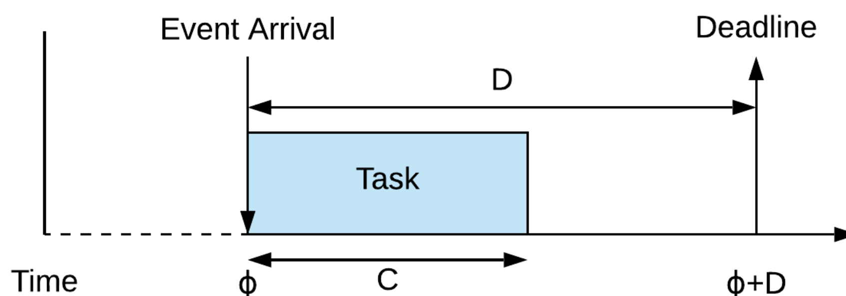

Fig. 1: The relationship between the capacity and the relative deadline of a task

Fig. 1 illustrates the relationship between the capacity and the relative deadline for a single activation of a task. The relative deadline D defines both how frequently a task may run (f=1/D) and when it must complete relative to its activation point (ϕ). In this diagram, the task runs immediately at the beginning of a period, however, the scheduler may place the task anywhere within the period as long as it will complete before the deadline. The ability to move a task within its period is how the scheduler accommodates multiple concurrent tasks.

The relationship between the capacity and the relative deadline of a task is constrained to $0 < C_i \leq D_i$ such that a task cannot be given a larger capacity than its period. However, a task may execute for multiple periods, which may or may not be back-to-back.

The ratio $U_i = C_i/D_i$ of the period of a task to its relative deadline is referred to as the worst-case utilization of the task (worst-case because the task may always complete before a time C_i has elapsed since launch). Worst-case utilization measures the percentage of processor-time a task may take when it uses its full capacity every period. In order to guarantee a set of tasks are successfully scheduled without missing deadlines, the sum U_{proc} of the worst-case utilizations of all tasks must be less than or equal to one:

$$U_{\text{proc}} \triangleq U_1 + U_2 + \ldots + U_N \leq 1.$$

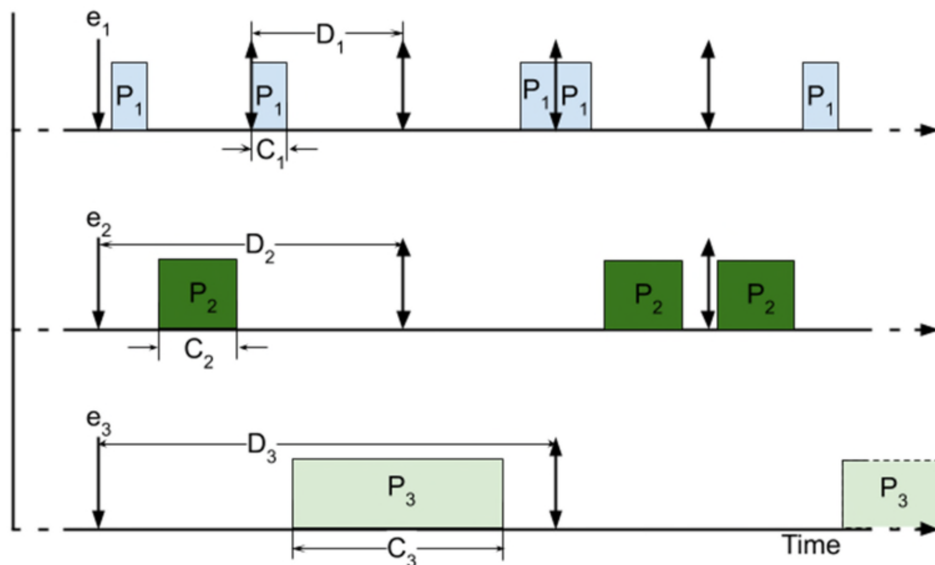

Fig. 2: Three tasks, their respective parameters, and how they complete prior to their respective deadlines while avoiding time-overlap with each other

Fig. 2 is an illustrative timeline with three tasks P_1, P_2, P_3, and their respective parameters. The labels e_1, e_2, and e_3 denote events that started the tasks running. After starting, the tasks continue to execute back-to-back periods. A downward arrow (↓) indicates the

activation (or launch) of a task, and an upward arrow (↑) indicates the deadline of a task. After the first period, the deadline of the previous task is coterminous with the activation of the next task; thus boundaries between periods are represented by two-sided arrows (↕=↑+↓). The scheduler acts to ensure that each task gets its allotted proportion of CPU time, completes before its deadline, and doesn't overlap with other tasks. The scheduler leverages the fact that the capacity C_i of a task is less than its deadline D_i to start tasks asynchronously, e.g., not necessarily at the beginning instant of its period, so that contending tasks do not overlap in time while each finishing before their respective deadlines.

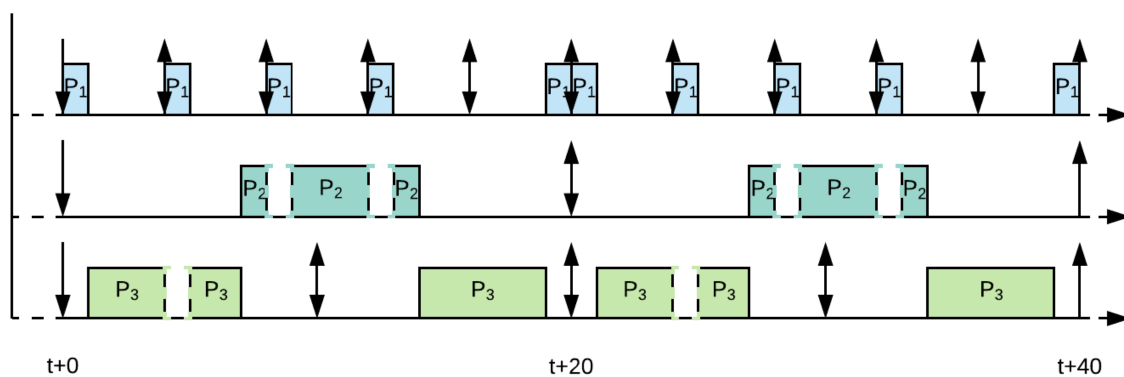

Fig. 3: Each of three tasks complete prior to their deadlines. Tasks P_2 and P_3 are temporarily preempted to enable task P_1 to meet its deadline

Fig. 3 illustrates another example of three tasks coordinated by a scheduler such that each task gets its allotted proportion of CPU time, completes before its deadline, and doesn't overlap with other tasks. In this example, the task P_2 is preempted on two occasions to enable task P_1, which has an earlier deadline, to complete. A task has its context saved upon preemption and reloaded upon restarting. Similarly, task P_3 is also preempted on two occasions to enable task P_1 to complete before its deadline.

A problem arises when we consider lock contention in a dynamic priority scheduler. Suppose that tasks P_2 and P_3 both contend on a lock (or mutually-exclusive resource) owned by another task P_s. In such a situation, the task P_s should get enough time to execute while avoiding a priority inversion. In a static priority scheduler, where tasks have a single comparable numeric priority, the choice, as mentioned before, is simple: find the maximum priority among the contenders and raise the priority of the lock owner to that priority. In dynamic priority schedulers, there is no immediate way to compare priorities, since tasks do not have comparable numeric priorities. The relative priorities of tasks can change depending on how close they are to their respective deadlines. Unlike a static scheduler, the parameters C_2, D_2 of task P_2 are not comparable to the parameters C_3, D_3 of task P_3.

Per the techniques of this disclosure, instead of choosing from among the contenders the best set of parameters for task P_s to inherit, a new set of parameters is derived that provide the same overall opportunity, or bandwidth, to run as all the contenders would have had were they not blocked on the lock. Priority inversion is thereby effectively avoided.

The new set of parameters for the task P_s is determined as follows. The worst-case utilization for the task P_s is set to the sum of the utilizations of the contending tasks. The deadline (period, D_s) of the task P_s is the minimum of the contending tasks. The capacity C_s of the task P_s is given by $C_s = U_s D_s$. In mathematical notation,

$$U_s = U_2 + U_3 = C_2/D_2 + C_3/D_3 \,,$$

$$D_s = \min(D_2, D_3) \,, \quad \text{and}$$

$$C_s = U_s D_s$$

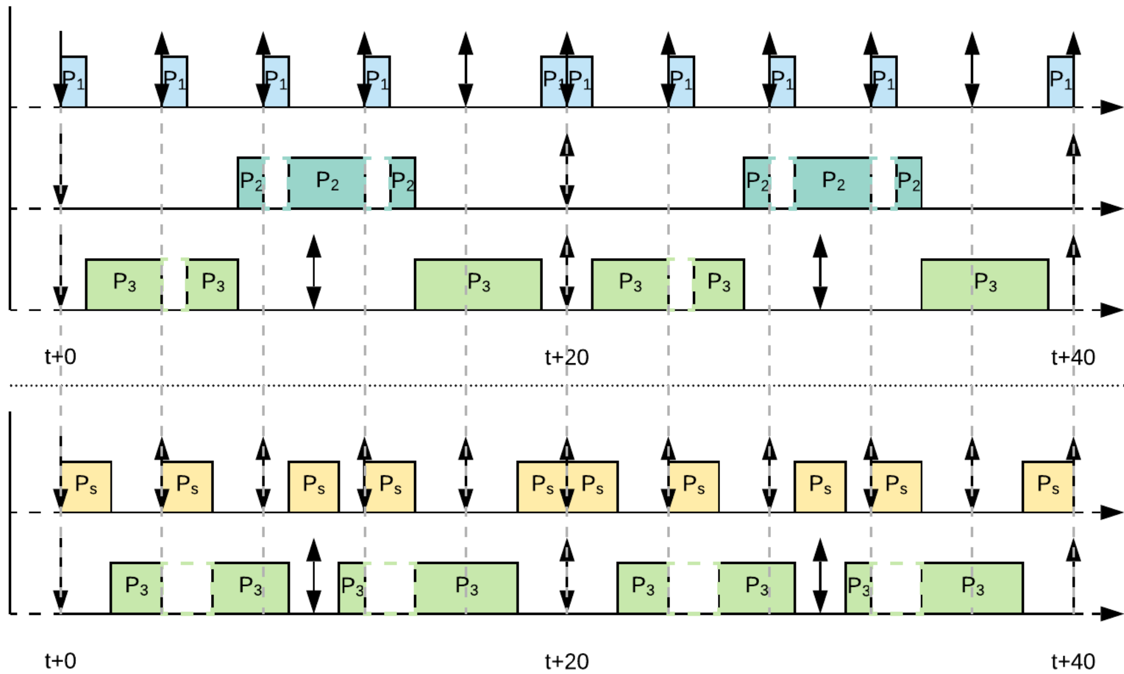

Fig. 4: Preventing priority inversion in dynamic priority schedulers

Fig. 4 illustrates how the new parameters for the process P_s enables the tasks to meet the same deadlines and produce the same total utilization. Effectively, the task P_s executes with the same absolute bandwidth and minimum relative deadline as constituent tasks P_1 and P_2. The top half Fig. 4 represents a possible timeline for the tasks if they had not blocked on the lock. The three tasks have utilizations $U_1 = 1/4$, $U_2 = 5/20$, and $U_3 = 5/10$, such that $U_{\text{proc}} = U_1 + U_2 + U_3 = 1$. The bottom half of the diagram shows the alternate timeline, where task P_s inherits the new set of parameters, e.g.,

$$
\begin{aligned}
U_s &= U_1 + U_2 &&= 5/10; \\
D_s &= \min(D_1, D_2) &&= 4; \\
C_s &= U_s D_s &&= 20/10;
\end{aligned}
$$

and runs in place of tasks P_1 and P_2. The gray dashed lines show where the original deadlines fall in the alternate timeline. It is observed that task P_s always completes at least as much work as the constituent tasks P_1 and P_2 would have by each of their respective deadlines. For example, at

time t+20, task P_s completes 10 units of work, which is the sum of the work tasks P_1 and P_2 would have completed by that time. Also, because task P_s has the same relative deadline, but greater capacity and lower laxity (difference between D_s and C_s) than the tightest task P_1, task P_s receives more service in the same interval and can meet the same demands. Therefore, priority inversion is avoided since task P_s can meet all of the same timing and utilization requirements as the contending tasks that are waiting for it to release the lock. Effectively, the task that holds a lock and blocks the other tasks from running is given bandwidth from the blocked tasks in order to complete faster and release the lock.

The techniques apply to any number of contending tasks as follows. The period D_s is computed as the minimum of the periods of the contending tasks and the capacity C_s is computed based on the sum-of-utilizations formula

$$U_s = C_s/D_s = U_x + U_y + \ldots + U_z \,,$$

where x, y, z are indices to the contending tasks.

CONCLUSION

This disclosure describes techniques to forestall or mitigate priority inversion in dynamic priority schemes. Per the techniques, a low-priority task that blocks higher priority tasks from running due to its owning a lock on a resource needed by the higher priority tasks is granted a new deadline and capacity such that the lock owner has the same overall opportunity to run as the lock contenders, were they not blocked on the lock. In this way, the lock-owing task gets sufficient time to execute while avoiding a priority inversion.

REFERENCES

[1] https://www.kernel.org/doc/html/latest/scheduler/sched-deadline.html, accessed Oct. 18, 2020.