INFO-F404 - Real-Time Operating Systems
Project 1: Global vs Partitioned DM

Anthony Caccia Jérôme Hellinckx

13 décembre 2016

1 Introduction

This project consists in studying performances of the *Deadline-monotonic scheduling* (DM) algorithm on systems with two different strategies : global and partitioned (best fit) scheduling. Systems have n periodic, asynchronous and independant tasks τ with constrained deadlines on a multiprocessor system.

DM is a *Fixed Task Priority Scheduler* (FTP) : tasks are ranked deterministically by their relative deadlines. The lower the relative deadline, the higher the priority. In case of equality, the priority still has to be deterministic. Therefore, in this implementation, the task id (which is unique) will decide of the priority.

1.1 Partitioned strategy

The main problem that has to be dealt with when using a partitioned strategy is to find an optimal partitioning such that each partition is schedulable on one processor while minimizing the number of partitions. Since this problem is known to be *NP-Complete*, this implementation uses the best-fit heuristic algorithm to find a partitioning. The *Best-Fit* strategy operates by assigning the current task to the processor for which the remaining utilization when adding said task utilization is minimum.

Once all tasks have been assigned to a partition, it cannot migrate and a uniprocessor scheduler is used on each partition.

1.2 Global strategy

Global strategy, on the other hand, permits to tasks to migrate between processors during their lifetime : they can then start their execution on a processor and resume on another one.

2 Code description

Three executables were asked in order to complete this project :

1. a simulator, taking as list of tasks, a number of processor and a strategy and simulate the system on the interval $I = [0; O_{max} + 2 * P]$;
2. a generator, which creates a list of n tasks with an utilization u ;
3. a study program, to test DM's performances.

2.1 Simulator

Aside from the essential data types - `Task` and `Job` which are described in the files with corresponding names - needed by the scheduler, the simulator is fully defined by one main basic class (`PCDSimulator`) and two other classes (`PDMSimulator` and `GDMSimulator`) derived from said main class. The idea behind the abstract super class is to offer tools to facilitate the scheduling of systems of Periodic tasks with Constrained Deadline (hence the name). This base class achieves that in the following ways :

- Being a template class permitting to specify a priority comparator between jobs. This flexibility is quite powerful because it means that one simply has to create a priority comparator between jobs and give it as template parameter to this class in order to define the desired priority assignment algorithm ;
- Regrouping the *ready* jobs into a priority queue where the priority is thus determined by the aforementioned class template argument such that the top of the queue is the job with highest priority ;
- Proposing a method checking if any deadline was missed ;
- Offering another method to generate jobs given a set of tasks ;
- Completely defining a `run` method which mimics the behavior of a multiprocessor system by sequentially generating the jobs, calling the scheduler, executing the jobs on each processor and finally checking if any deadline was missed.

`PDMSimulator` is the class defining the Partitioned strategy whereas `GDMsimulator` is the class describing the Global strategy using a Deadline Monotonic priority assignment. The fundamental contribution of these classes is, as their naming may suggest, to stipulate to which processor each top priority job must be assigned to. This is in practice realized by *overriding* the `schedule` method.

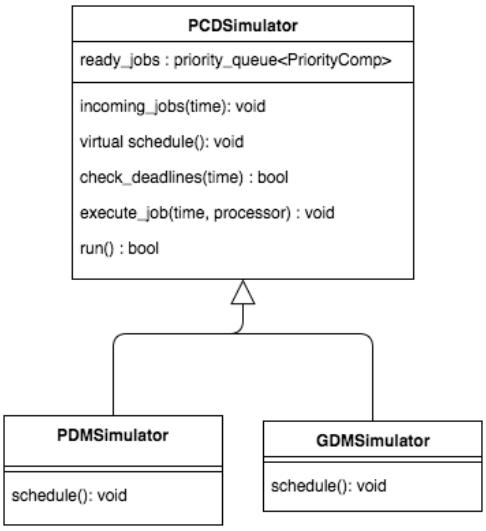


FIGURE 1 – Simulator class diagram

2.2 Generator

These are the primary constraints for tasks creation :

- there should be exactly n tasks ;
- the system utilisation should be very close to u
- each generated tasks τ_i should have $0 <= u_i <= 1$
- $\forall \tau_i . t_i \geq d_i \geq c_i$

The code is really simple : you just generate n numbers between 0 and 1, those numbers are then multiplied by the expected system utilisation u and divided by their sum : those numbers are now the u_i for each created tasks. We then convert u_i to fraction : the numerator will become the c_i and the denominator, the t_i. Now we can generate a d_i which is a random number from an uniform integer distribution between c_i and t_i.

All o_i can be generated randomly, we will just substract each of them by the minimal o_i so the minimal will be 0.

2.3 Visual output

A visual output of the scheduling is also created when running the simulators. This was done by writing a `python` program using `matplotlib` that generates a user-friendly output and by calling it with

`C++` bindings.

3 Encountered problems and solutions

3.1 Simulator

From a technical point of view, the primary problem that we faced when writing the simulator was to understand which parts of the global and partitioned strategy could be generalized in a base class that could be thus shared between the two classes. Realizing that at the end of the day the only difference between these two strategies was only how they would assign, at time t, the highest priority jobs to a set of processors lead to the class structure seen in FIGURE 1. That is actually why these two classes are so *small*, since they mainly only override the `schedule` routine.

Moreover, we experienced some difficulties when trying to figure out what was really meant when stipulating that the simuator should output the number of processors *required*. To solve this issue, we organized the number of processors of a given task set into two categories :

1. The number of processors used. For example if the user entered 5 processors in the simulator but 3 of them were idle during the whole study interval, the number of processors used is 2.
2. The minimum number of processors needed such that the task set is schedulable.

The latter category needs to be further explained for each strategy. In the case of the partition strategy, the minimum number of processors needed may not be obtainable. This is because we use a best-fit heuristic that will always partition the tasks into the same partitions. Hence, even if we increase the number of processors, the partitioning will not change and the processor that observed a missed deadline will again miss the same deadline. That is why, if a partitioning is found, the minimum number of processors needed is equal to the number of processors used if the partitioning is schedulable. However, if we consider the global strategy, it is quite obvious that for all valid task sets, there exist a minimum number of processor for which the task set is schedulable (let us for example take a number of processors equal to the number of the tasks). Said minimum is computed by using a binary search algorithm defined in `PCDSimulator`.

3.2 Generator

There could be problems in asked conditions : if the asked u is bigger than $100 * n$, clearly it is impossible to generate n tasks with $0 < u_i \leq 1$. If this case happen, u is simply decreased to $100 * n$.

Contrarilly to what is stated in 2.2, generating the first numbers between 0 and 1 is a bad idea. If we take an extreme case where it is asked to take n jobs with a system utilisation $u = 100 * n$, it is logical that each task utilisation $\tau_u = 100$. We thus have to reduce the interval were the random numbers are taken. We can handle this by reducing this interval : we take $[m - s; m + s]$ where $m = u/n$ and $s = \min(m, 1 - m)$.

4 Comparison tests specifications

Let us first briefly describe the most important routine of the comparison tests. Said routine receives a task generator parametrized by an utilization and a number of tasks as well as a number of processors. The generator then produces a task set respecting its given parameters and the routine then computes the total load, schedulability and total number of required (minimum) processors depending on the strategy (it thus uses both simulators). This is done n times where n is the sample size. Note that the load and the schedulability is computed for the given number of processors (which is thus constant) and for the aforementioned *minimum* number of processors. Finally, the returned values are the mean of computed values for the sample size.

The comparison tests depending on the utilization and the number of tasks is then done by modifying the generator parameters given to the above routine. The utilization test fixes the number of tasks to 8, the number of processors to 4 and bounds the utilization between 5 and 300. It then iterates in this range with a step of 5 and calls the first routine with a generator set to the current utilization value. Similarly, the number of tasks test fixes the utilization to 70, the number of processors to 4 and bounds the number of tasks between 1 and 30 with a step of 1.

5 Results

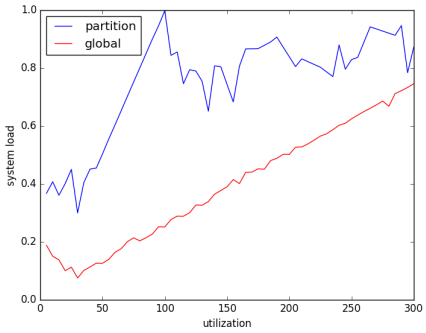
We observe in FIGURE 2 - (a) that for a fixed number of processors, the system load depending on utilization is considerably higher for the partition strategy. This observation is due to the best-fit partitioning strategy used that *compacts* the tasks into few partitions. An equivalent analysis could be made for FIGURE 2 - (b) where the minimum number of processors is considered. Moreover, we notice that the two partition graphs are identical. This is because, as mentioned earlier in this report, the minimum number of partitions is equal to the number of partitions used. Hence the schedules are identical. This is however not the case for the global strategy where the system load is obviously higher when using the minimum number of processors (FIGURE 2 - (b)) instead of the fixed one (FIGURE 2 - (a)).

FIGURE 3 indicates quite clearly that the system load directly depends on the number of tasks for the partition strategy but not so much for the global strategy. Indeed, if the tasks set is composed of only a few tasks, there is a higher chance than the best-fit heuristic will assign tasks to more processors, because each task would have an high utilization itself and would in consequence be *harder to fit* into an existing partition. If the tasks set is conversely composed of many tasks, then each task would be *easier to fit* into an existing partition thus increasing the total utilization of each partition therefore the system load is higher. The system load for the global strategy however doesn't appear to depend on the number of tasks and is quite understandably higher when considering the minimum number of processors than when considering a fixed number of processor.

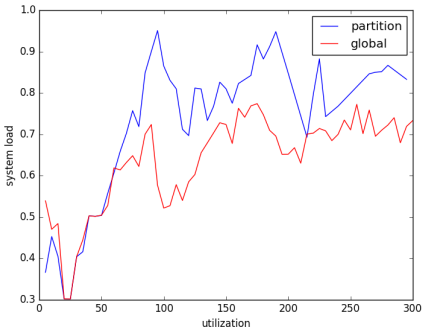
Let's also point out that the particular shape of the partition graph on FIGURE 4 - (a) intuitively informs on the heuristic used for the partitioning. Indeed the best-fit algorithm is quite conservative regarding the number of processors used since it tries to find the partition for which the utilization left is minimum after adding the current task. It will thus *compact* the tasks into as few partitions as possible. It will therefore have the same number of partitions for a range of utilization which explains why the number of processors required for the partition strategy has a stair-stepped graph. Logically, the step appears when the normalized total utilization is equal to the number of current partitions, because from that point adding one utilization would enforce the creation of a new partition since each partition cannot have an utilization greater than one. We additionally note from FIGURE 4 that the number of processors required for the global strategy is always greater than for the partition strategy.

Values shown in FIGURE 5 were obtained using a fixed number of 4 processors. We observe that the partition strategy begins to show serious trouble finding a schedule when the tasks set utilization is around 1 (100). This is because while the utilization is less than 1 the best-fit partitioning will assign all the tasks to the same partition. When the utilization then approaches 1, the partition gets harder and harder to schedule because tasks set with an utilization $\simeq 1$ are less likely to be schedulable on a single processor. The schedulability sinks in consequence hard for utilization values that are close to 1.

Combining the reasonings developed for FIGURE 3 and in the previous paragraph allows us to easily explains FIGURE 5 - (B). Fewer tasks means higher utilization for each task and thus higher probability of having more partitions with each a lower utilization therefore easier to schedule. Conversely, having more tasks leads to more packed partitions which are harder to schedule. In this example, the partition strategy schedules a maximum of 20% tasks set when the number of tasks is greater than 10 ! This figure also suggests that the schedulabilty of the global strategy does not depend on the number of tasks in the system.

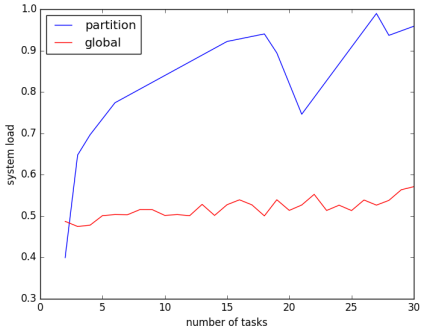


(a) Fixed number of processors

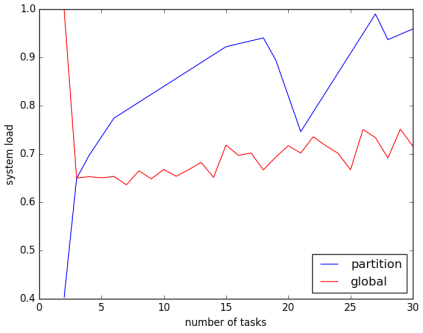


(b) Minimum number of processors

FIGURE 2 – Load depending on utilization

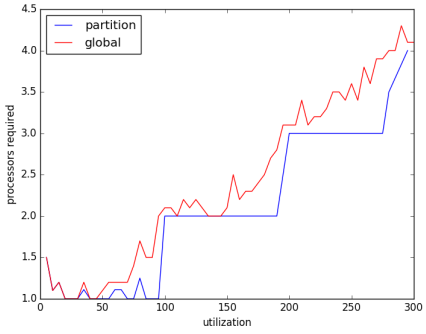


(a) Fixed number of processors

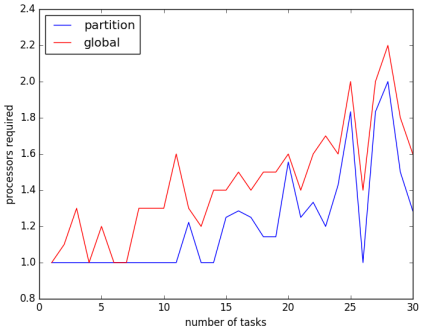


(b) Minimum number of processors

FIGURE 3 – Load depending on number of tasks

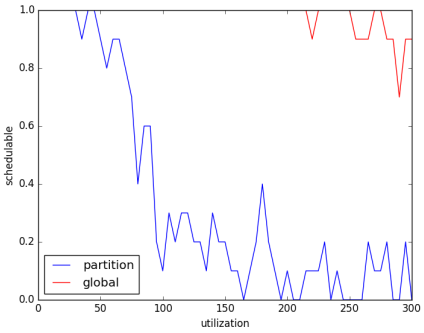


(a) Depending on utilization

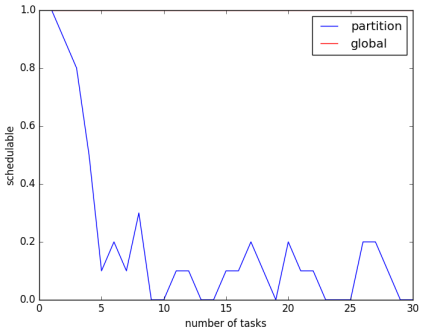


(b) Depending on number of tasks

FIGURE 4 – Number of processors



(a) Depending on utilization



(b) Depending on number of tasks

FIGURE 5 – Schedulability