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

Anthony Caccia Jérôme Hellinckx

12 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 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 scheduling, 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.

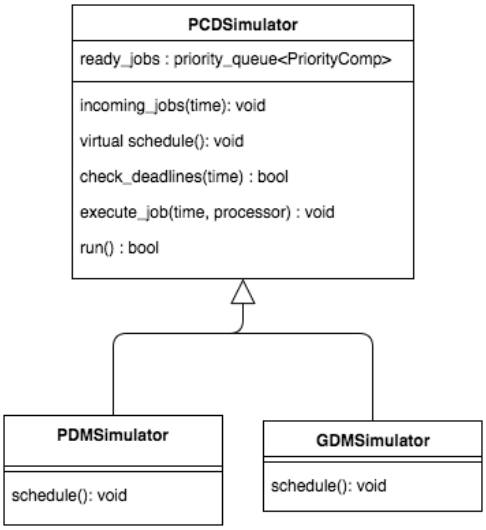

FIGURE 1 – Simulator class diagram

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.

3 Encoutered 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 (let us for example take a number of processors equal to the number of the tasks) for which the task set is schedulable. The minimum number of processors is computed 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 $s = \min(m, 1 - m)$.

3.3 Study

4 Comparison tests specification

5 Results