INFO-F404 Operating Systems II - Projet 1

Chapeaux Thomas
Dagnely Pierre

November 27, 2012

1 Introduction

Le but de ce projet était de construire en C++ un simulateur d'ordonnancement multiprocesseur d'un systèmes de tâches à temps réel via EDF Global ou EDF-k, ainsi qu'un générateur de systèmes de tâche permettant de spécifier le nombre de tâches du système ainsi que son utilisation globale.

Ensuite, il nous était demandé d'utiliser ces modules pour comparer les deux algorithmes (EDF Global et EDF-k) selon le nombre de préemptions, de migrations, de cœurs utilisés et d'instants oisifs.

2 Choix d'implémentation

2.1 Nature du simulateur

Notre simulateur utilise un temps discret (comme vu en cours) et est à pas constant. Il simule le système passé en paramètre avec un nombre de cœur choisi par l'utilisateur, mais contient aussi une fonction permettant de calculer ce nombre. La simulation durera le temps maximal nécessaire pour effectuer une période complète ($2 * PPCM(T_i) + max(O_i)$).

2.2 Nature du système

Comme demandé par l'énoncé, le simulateur accepte des systèmes périodiques à départ différé et à échéance contrainte. Il ne vérifie pas la faisabilité du système avant de commencer, mais s'arrête dés qu'un travail manque son échéance.

En pratique, les systèmes générés sont cependant à échéances implicites. En effet, nous avons dans ce cas des systèmes dont on peut calculer la faisabilité ($U_{max} \leq 1$) et le nombre de cœurs requis ($\lceil U_{globale} \rceil$) sans trop de complexité[1].

2.3 Génération de tâches

La génération du système de n tâches avec une utilisation $U_{globale}$ se fait en trois étapes.

[1] Précisons : Ceci est la solution proposée par Mr Goossens

Premièrement, on génère les n tâches avec des paramètres aléatoires (distribution uniforme entre deux bornes pour chaque paramètre). Le WCET est bien entendu borné par la valeur de l'échéance.

Ensuite, on calcule le facteur d'erreur $F_u = \frac{\Sigma U_i}{U_{globale}}$ et on l'applique à toutes les tâches soit en multipliant le WCET soit en divisant la période (une chance sur deux pour chaque tâche).

Ceci ne nous donne pas encore l'utilisation demandée pour deux raisons : premièrement, le système étant à temps discret, les résultats des opérations sont toujours arrondies à l'entier le plus proche ce qui crée des imprécisions. Ensuite, le facteur d'erreur est limité par les contraintes logiques des tâches ($wcet \geq 1$, $wcet \leq deadline$, etc).

On effectue donc une troisième étape dans laquelle on va, jusqu'à obtenir une utilisation acceptable, sélectionner une tâche au hasard et faire varier son wcet ou sa période de 1 pour faire augmenter ou diminuer son utilisation (en fonction de ce qui est nécessaire).

Ce système nous semble générer des systèmes assez variés en un temps raisonnable.

2.4 Structures de données

Il a été choisi d'utiliser la structure *deque*[2] de la STL pour le stockage des tâches (`tasks`), des travaux actifs, donc entrés dans le système et dont la deadline n'est pas dépassée (`jobs`) et pour les différents cœurs (`CPUs`, qui contient des pointeurs vers des éléments de `jobs`).

Les travaux actifs sont également référencés par deux files à priorité de pointeurs vers des travaux : la file `Running`, contenant les travaux en cours de calcul par un des cœurs (dont l'élément le plus prioritaire est le travail avec la plus $faible$ priorité), et la file `Ready`, contenant tous les autres travaux actifs (dont l'élément le plus prioritaire est le travail avec la plus *grande* priorité).

On notera qu'il existe deux structures de pointeurs vers les travaux en cours de calcul : la file `Running` et la deque `CPUs`. Ceci est dû au fait qu'une file à priorité ne permet pas facilement d'accéder à l'entierté des éléments ou de différencier les différents cœurs (pour le compteur de migration).

2.5 Implémentation de EDF-k

Après avoir chargé le système (passé en paramètres ou lu sur un fichier), on calcule le nombre minimale de cœur et le nombre de tâches prioritaires avec la formule : $m_{min}(T) = \min_{k=1}^{n}\{(k-1) + \lceil \frac{U^{(T^{(k+1)})}}{1-U(T_k)} \rceil\}$.

On lance ensuite un simulateur global EDF qu'on a légèrement adapté pour qu'il puisse gérer les jobs prioritaires (c.f. chap 3.2).

[2] `http://www.cplusplus.com/reference/deque/deque/`

Ainsi quelque soit l'état du système ceux-ci passeront devant les autres jobs et ils auront l'impression d'avoir un processeur chacun.

2.6 Implémentation de global EDF

Après avoir chargé le système (passé en paramètres ou lu sur un fichier), on calcule le nombre minimale de cœur avec la formule : $m_{min} = \lceil U_{globale} \rceil$.

On lance ensuite le simulateur global EDF. C'est le même que pour EDF-k, mais tous les jobs ayant la même priorité (nulle), celui-ci se comporte alors en "pure" global EDF.

2.7 Implémentation du simulateur

Voici une version simplifiée de la boucle principale de notre simulateur :

```
1  while (_t < studyInterval) {
2    generateNewJobs(_t);
3    // scheduling : assign which jobs goes to which CPUs
4    while (not _readyJobs.empty()
5    and (availableCPUs or JobNeedToBePreempted() )) {
6    // The earliest-deadline active job should get a CPU
7      if (availableCPUs)
8      {
9        Job* newJob = _readyJobs.pop();
10       _CPUs[firstIdleCPU] = newJob;
11       _runningJobs.push(newJob);
12     }
13     else // all CPUs are busy, preempt the one with the latest deadline
14     {
15       Job* oldJob = _runningJobs.pop();
16       Job* newJob = _readyJobs.pop();
17       _CPUs[posOldJob] = newJob;
18       _readyJobs.push(oldJob);
19       _runningJobs.push(newJob);
20       preemption_counter++;
21     }
22   }
23
24   // CPUs
25   for (unsigned int i = 0; i < _CPUs.size(); ++i)
26   {
27     if (_CPUs[i] != NULL)
28     {
29       // compute jobs in CPUs
30       if (_CPUs[i]->getLastCPU_Id() != (int) i)
31         ++migration_counter;
32       _CPUs[i]->giveCPU(_deltaT, i);
33
34       // check if a job is done
35       if (_CPUs[i]->getComputationLeft() == 0)
36         _CPUs[i] = NULL;
37       else
38         ++idle_time_counter;
39     }
40   }
41   // advance time
42   _t += _deltaT;
43   cleanAndCheckJobs(_t);
44 }
```

Récoltes de statistiques
Notre simulateur renvoi le nombre de préemptions, d'instants oisifs et de migrations apparues au cours de la simulation.

2.8 Implémentation du comparateur

Nous avons crée un comparateur lançant deux types de comparaisons, en fonction de ces paramètres.

1. ./globalEDFvsEDF-k -u $utilisationVoulue$ -n $nombreDeCœur$ -t $nombreDeTestes$
   On compare alors EDF-k et global EDF pour le système correspondant.
   On lance donc successivement TaskGenerator puis simGlobalEDF et simEDF-k, mais sans passer par des fichiers textes, ils se passent le système et les différents paramètres directement.

2. ./globalEDFvsEDF-k
   On compare ici EDF-k et global EDF pour différentes valeurs d'utilisation et de nombre de tâches à des fins de statistiques.
   On essaye des utilisations de (30-50-70-90-110-130-150) pour des systèmes de (5-10-15-20) tâches.
   Le résultat est affiché et stocké dans un fichier "statistics.txt"

3 Difficultés rencontrées

3.1 Intervalle d'étude

Les périodes des tâches sont générées aléatoirement. Hors, elles vont avoir une très grande influence sur l'intervalle d'étude du simulateur vu que celui-ci est égal à $PPCM(T_i) + max(O_i)$. Des périodes tirées aléatoirement peuvent très vite rendre le PPCM relativement grand, jusqu'au point où le temps de calcul devient inacceptable.

Ce problème a été réglé en donnant 19 valeurs de période possibles (dont le PPCM est connu et acceptable) et en choisissant pour chaque tâche une de ses valeurs.

Les valeurs choisies sont : 2, 3, 5, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 22, 24, 25, 28, 30, 32, leur PPCM est 554400.

3.2 Priorité en EDF-k

La principale difficulté de EDF-k est la gestion des tâches prioritaires.
La méthode vu en cour consistait à mettre les deadlines de ces tâches à $-\infty$. mais notre implémentation du simulateur utilise fréquemment ces deadlines pour divers calculs (tester l'ordonnancabilité du système, ...) et la gestion de deadline à $-\infty$ aurait trop complexifié notre code.

Il nous a donc semblé plus simple de rajouter un paramètre booléen "priority" aux tâches (et donc à leurs jobs), ainsi on ne doit se préoccuper de la priorité qu'à deux endroits :

1. Lors du tri des priority_queue ready et running, en modifiant le comparateur EDFcomp pour que les jobs prioritaires arrivent au sommet de ready (ainsi ils sont toujours exécuté en premier) et en bas de running (ainsi on regarde prioritairement les autres jobs en cas de préemption).

2. Lors du choix des jobs a préempter, on utilise la méthode "JobNeedToBePreempted()" qui regarde les sommets de ready et running pour définir si une préemption est nécessaire.
   S'il y a des jobs prioritaires, ils sont alors privilégiés (ont leur attribue un cœur ou les laisse terminer leur job), sinon on se base sur les deadlines.

Ainsi le système peut fonctionner avec des jobs prioritaires, ou sans jobs prioritaire, devenant alors un simulateur global EDF.

4 Résultats

En se basant sur notre comparateur, on peut déjà analyser assez finement les différences entre global EDF et EDF-k.
Mais on ne prend pas en compte le nombre de système non-ordonnancable rencontré par le comparateur, normalement EDF-k ordonnance beaucoup moins de systèmes que global EDF, mais cela n'est pas analysé ici.

Nombre de cœurs
Quelque soit le nombre de tâches, jusqu'à une utilisation de 70, ils utilisent tout deux un seul cœur. Mais passez ce cap, en moyenne global EDF commence à utiliser de plus en plus de cœurs tandis que EDF-k reste stable.

EDF-k est donc plus intéressant pour des système avec une plus grosse utilisation.

Nombre de préemptions et migrations
Quelque soit l'utilisation et le nombre de tâches, EDF-k nécessite en moyenne plus de préemptions, mais moins (voir pas) de migrations que global EDF.

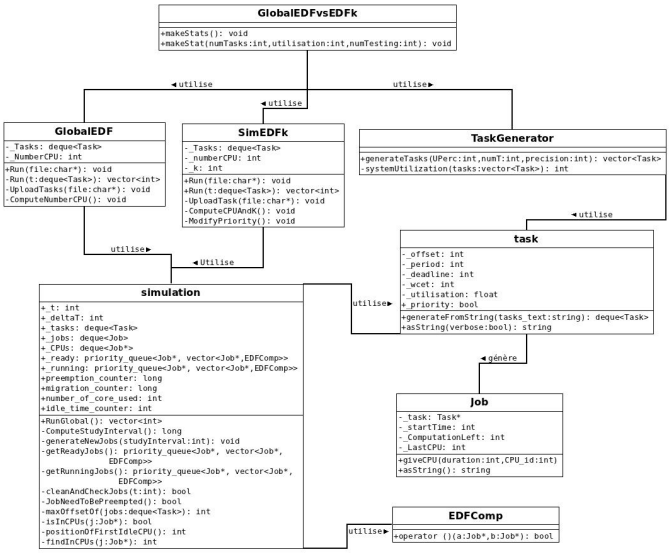
Nombre d'instants oisifs
Quelque soit l'utilisation et le nombre de tâches, EDF-k et global EDF ont peu ou prou le même nombre d'instants oisifs.

5 Appendice

5.1 Output des programmes

```
pierre@Marvin:~/info-f404-project1$ ./taskGenerator -u 120 -n 8 -o tasks.txt
pierre@Marvin:~/info-f404-project1$ ./simEDF-k tasks.txt
statistics of the simulation :
Number of preemption = 1142
Number of migration = 160
idle time  = 4517
Core used = 2
pierre@Marvin:~/info-f404-project1$ ./simGlobalEDF tasks.txt
statistics of the simulation :
Number of preemption = 682
Number of migration = 270
idle time  = 4517
Core used = 2
pierre@Marvin:~/info-f404-project1$ ./globalEDFvsEDF-k -u 120 -n 8 -t 50
statistics of the simulation :   Global EDF       |      EDF-k
-----------------------------------------------------------------------
Average number of preemption =          1185.24  |      580.08
Average number of migration =           703.66   |      335.26
Average idle time  =                    22632    |      22632.1
Average number of Core used =           2.02     |      2.04
pierre@Marvin:~/info-f404-project1$
```

5.2 Diagramme du programme



5.3 Statistiques du comparateur

```
The result are display with this form (stat for global EDF / stat for EDF-k)
If the value is -1, it means that the generator has not happened to create systems with these settings
statistics of the simulation :          5 tasks          |  10 tasks      |   15 tasks    |   20 tasks
-----------------
Utilisation of 30
-----------------------------------------------------------------------------------------------------------------
Average number of preemption =          (16.8182/40.8636)|     (0/0)      |     (-1/-1)   |     (-1/-1)
Average number of migration =           (0/0)            |     (0/0)      |     (-1/-1)   |     (-1/-1)
Average idle time  =                    (184.136/184.136)|     (0/0)      |     (-1/-1)   |     (-1/-1)
Average number of Core used =           (1/1)            |     (1/1)      |     (-1/-1)   |     (-1/-1)
-----------------
Utilisation of 50
-----------------------------------------------------------------------------------------------------------------
Average number of preemption =          (138/145.786)    |     (160/2266.67)|  (-1/-1)   |     (-1/-1)
Average number of migration =           (0/0)            |     (0/0)        |  (-1/-1)   |     (-1/-1)
Average idle time  =                    (1097.83/1097.83)|     (711/711)    |  (-1/-1)   |     (-1/-1)
Average number of Core used =           (1/1)            |     (1/1)        |  (-1/-1)   |     (-1/-1)
-----------------
Utilisation of 70
-----------------------------------------------------------------------------------------------------------------
Average number of preemption =          (115.619/88.0952)|     (2031.36/2446.36)|     (-1/-1)  |     (-1/-1)
Average number of migration =           (0/0)            |     (0/0)            |     (-1/-1)  |     (-1/-1)
Average idle time  =                    (1235.69/1235.79)|     (10927.7/10927.7)|     (-1/-1)  |     (-1/-1)
Average number of Core used =           (1/1)            |     (1/1)            |     (-1/-1)  |     (-1/-1)
-----------------
Utilisation of 90
-----------------------------------------------------------------------------------------------------------------
Average number of preemption =          (206.214/205.286)|     (1949.57/2996.57)|     (217.5/1656.75)|       (-1/-1)
Average number of migration =           (1.71429/0)      |     (60.2857/0)      |     (121.5/0)       |       (-1/-1)
Average idle time  =                    (2333.67/2333.79)|     (10745.5/10745.4)|     (5962.5/5962.5)|       (-1/-1)
Average number of Core used =           (1.33333/1)      |     (1.61905/1)      |     (2/1)           |       (-1/-1)
-----------------
Utilisation of 110
```