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 en multipliant le WCET.

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, la modification par le facteur d'erreur est limitée par les conditions d'existences 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 de 1 pour faire augmenter ou diminuer son utilisation (en fonction de ce qui est nécessaire).

Il est à noter qu'on aurait pu également effectuer ses changements sur la période, ou mélanger ces deux approches en modifiant parfois le WCET, parfois la période dans le but d'avoir des systèmes plus variés. C'est ce qui avait été fait au départ mais on a dû limiter les modifications au WCET pour garder un intervalle d'étude raisonnable (voir section 3.1).

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

2.5 Implémentation de EDF-k

Après avoir chargé le système (passé en paramètre ou lu sur un fichier), on calcule le nombre minimale de cœurs 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. section 3.2).
Ainsi quel que 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 renvoie l'intervalle d'étude, 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 ses paramètres.

1. `./globalEDFvsEDF-k -u utilisationVoulue -n nombreDeCœur -t nombreDeTests`
   On compare alors EDF-k et global EDF pour sur des systèmes correspondants.
   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 [sans parametres]` 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 vue en cours 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 rendu notre code trop complexe.

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 prioritaires, 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.

Nombre de cœurs
La simulation renvoie deux nombres de cœurs : le nombre théorique minimal qui a été utilisé et le nombre de cœurs réellement nécessaire (donc le nombre maximum de cœurs utilisés pendant un même instant).

Jusqu'à une utilisation de 100, EDF-k et global EDF utilisent peu ou prou le même nombre de cœurs.
A partir d'une utilisation de 150, EDF commence à nécessiter plus de cœurs que global EDF.

Parallèlement à cela, plus on a de tâches et plus le nombre de CPU utilisé est important par rapport au nombre de CPU vraiment nécessaire.

Nombre de préemptions et migrations

En règle générale global EDF nécessite plus de préemptions et migrations que EDF-k.
Et plus il y a de tâches, plus l'écart est grand.

Intervalles d'étude
Pour de petites utilisations, les intervalles d'études sont assez proches, mais pour de grandes utilisations (et nombre de tâches) on peut avoir des différences significatives, sans qu'une tendance ne se fasse jour.

Nombre d'instants oisifs
Jusqu'à une utilisation de 100, EDF-k et global EDF ont peu ou prou le même nombre d'instants oisifs.
A partir d'une utilisation de 150, EDF-k commence à nécessiter plus d'instants oisifs que global EDF, sauf pour un faible nombre de tâches (5).

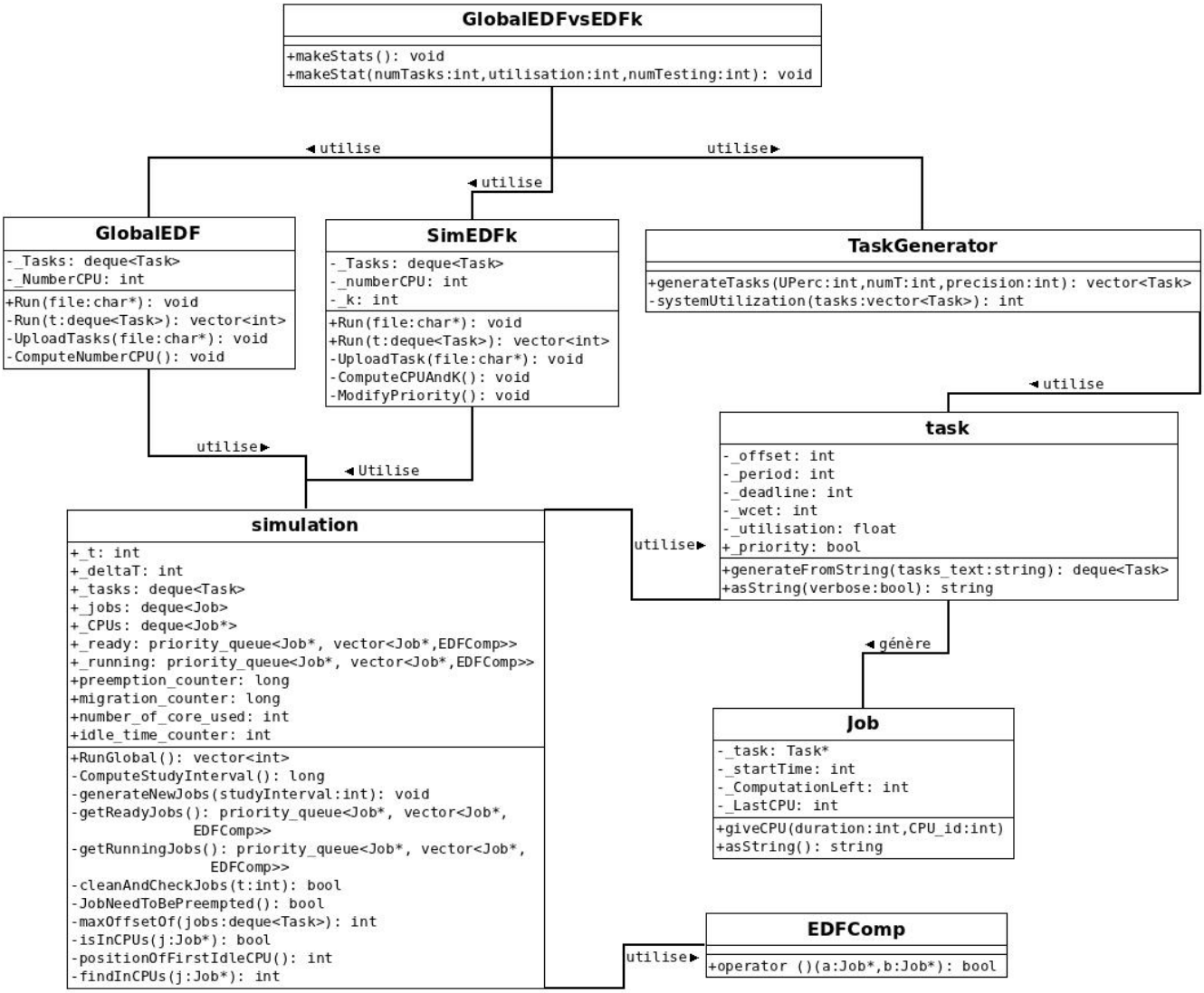
Nombre de systèmes ordonnancés
global EDF et EDF-k ordonnancent plus ou moins le même nombre de tâche, mais global EDF est souvent légèrement plus efficace.

Par contre pour un petit nombre de tâches et pour des utilisations supérieures à 200, on voit que EDF-k ordonnance plus de systèmes que global EDF

Conclusion
EDF-k nous semble donc plus indiqué dans le cas de système avec peu de tâches (aux alentour de 5) et avec une forte utilisation (au-dessus de 200).
EDF-k ordonnance alors généralement plus de système et le fait avec moins d'instants oisifs mais utilise un petit peu plus de cœurs.

5 Appendice

5.1 Diagramme de classe



5.2 Output des programmes

```
pierre@Marvin:~/info-f404-project1$ ./taskGenerator -u 120 -n 8 -o task.txt
pierre@Marvin:~/info-f404-project1$ ./simGlobalEDF task.txt
statistics of the simulation :
Number of preemption = 554
Number of migration = 176
idle time  = 36516
studyInterval  = 33614
Core used = 2
Core necessary = 2
pierre@Marvin:~/info-f404-project1$ ./simEDF-k task.txt
statistics of the simulation :
Number of preemption = 468
Number of migration = 220
idle time  = 36516
studyInterval  = 33614
Core used = 2
Core necessary = 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 =          637.48    |      466.3
Average number of migration =           297.66    |      206.12
Average idle time  =                    53955     |      54171.6
Average study interval =                44927     |      44927
Average number of Core used =           2         |      2.04
Average number of Core necessary =      2         |      2.04
pierre@Marvin:~/info-f404-project1$
```

5.3 Statistiques du comparateur

Les résultats sont affichés sous la forme (résultat pour global EDF/ résultat pour EDF-k)
Ces statistiques sont générée en produisant (ou en essayant de produire) 50 systèmes respectant les paramètres correspondants.
Cependant il peut arriver que le générateur n'arrive pas à créer un système viable ou que le simulateur n'arrive pas à ordonnancer ce système.
C'est représenter par la ligne "Number of systems scheduled", dont les résultats ont la forme :
(nombre de système ordonnancer par global EDF / par EDF-k) sur le nombre de système créé par le générateur.

Si aucun système n'a été créer par le générateur ou n'est ordonnancable, alors les valeurs sont mises à -1.

Utilisation de 50

	5 tasks	10 tasks	15 tasks	20 tasks
Average number of preemption	(59.9231/62.1282)	(27/27)	(-1/-1)	(-1/-1)
Average number of migration	(0/0)	(0/0)	(-1/-1)	(-1/-1)
Average idle time	(3883.62/3883.62)	(48812.3/48812.3)	(-1/-1)	(-1/-1)
Average study interval	(5787.23/5787.23)	(71778.3/71778.3)	(-1/-1)	(-1/-1)
Average number of Core used	(1/1)	(1/1)	(-1/-1)	(-1/-1)
Average number of Core necessary	(1/1)	(1/1)	(-1/-1)	(-1/-1)
Number of systems scheduled	(39/39)sur 39	(3/3)sur 3	(0/0)sur 0	(0/0)sur 0

Utilisation de 100

	5 tasks	10 tasks	15 tasks	20 tasks
Average number of preemption	(189.041/160.271)	(3648.48/2904.5)	(3821.73/8191.2)	(-1/-1)
Average number of migration	(1.18367/0.708333)	(710.522/816.667)	(1492.91/1327.6)	(-1/-1)
Average idle time	(1791.76/1853.6)	(193438/203715)	(411227/451745)	(-1/-1)
Average study interval	(3953.63/3815.69)	(158312/159417)	(414541/400554)	(-1/-1)
Average number of Core used	(1.42857/1.5)	(1.69565/1.79167)	(1.81818/1.9)	(-1/-1)
Average number of Core necessary	(1.40816/1.47917)	(1.69565/1.79167)	(1.81818/1.9)	(-1/-1)
Number of systems scheduled	(49/48)sur 49	(23/24)sur 24	(11/10)sur 11	(0/0)sur 0

Utilisation de 150

	5 tasks	10 tasks	15 tasks	20 tasks
Average number of preemption	(264.44/167.735)	(4239.54/899.63)	(11553.1/8129.87)	(12206/2044)
Average number of migration	(175.32/106.265)	(1999.98/460.783)	(5584.04/2967.83)	(6035.5/1579.33)
Average idle time	(9653.8/10808.4)	(87089.4/100436)	(263831/448749)	(310073/314085)
Average study interval	(13116.4/12478.7)	(93994.7/73871.9)	(276830/287663)	(400698/164659)
Average number of Core used	(2/2.28571)	(2/2.47826)	(2/2.78261)	(2/3)
Average number of Core necessary	(1.92/2.08163)	(2/2.45652)	(2/2.73913)	(2/3)
Number of systems scheduled	(50/49)sur 50	(48/46)sur 48	(26/23)sur 26	(4/3)on 4

Utilisation de 200

	5 tasks	10 tasks	15 tasks	20 tasks
Average number of preemption	(235.744/37.9565)	(2513.02/332.98)	(16853.8/4168.02)	(13711.4/6131.28)
Average number of migration	(105.233/26.2174)	(1471.6/203.592)	(9071.81/2645)	(7561/3534.72)
Average idle time	(3834.4/17427.1)	(35433.4/63629.3)	(318775/478492)	(591950/802179)
Average study interval	(6460.56/13205.2)	(40400.6/35567.6)	(273696/253813)	(387086/387086)
Average number of Core used	(2.02326/2.93478)	(2.4/3.2449)	(2.67442/3.33333)	(2.88/3.36)
Average number of Core necessary	(1.97674/2.69565)	(2.4/3.18367)	(2.67442/3.2619)	(2.88/3.36)
Number of systems scheduled	(43/46)sur 50	(50/49)sur 50	(43/42)sur 43	(25/25)sur 25

Utilisation de 250

Average number of preemption	(22.6/4.2)	(2459.2/247)	(1240.9/266.111)	(3507.56/1037.56)
Average number of migration	(10.7/4.2)	(1800.9/145.6)	(791.6/172.222)	(2609/712)
Average idle time	(4389.4/5542.5)	(95505.3/174328)	(34511/58417.1)	(184539/303331)
Average study interval	(4212.8/4212.8)	(78773.9/78773.9)	(27258.2/26551.4)	(125192/125192)
Average number of Core used	(3/3.5)	(3/4.1)	(3/3.88889)	(3/4)
Average number of Core necessary	(2.9/3.3)	(3/4)	(3/3.66667)	(3/4)
Number of systems scheduled	(10/10)sur 10	(10/10)sur 10	(10/9)sur 10	(9/9)sur 9

Utilisation de 300

Average number of preemption	(69.3333/6.5)	(3052.5/30.8)	(11485.5/525.8)	(5966.5/389.125)
Average number of migration	(31/6.5)	(2300.2/27.4)	(7565.3/331.2)	(4390.6/317.625)
Average idle time	(9093.67/18936.1)	(140933/308773)	(168359/509329)	(1.05e+06/1.32e+06)
Average study interval	(9135.17/10868.4)	(102546/102546)	(185730/185730)	(463026/370878)
Average number of Core used	(3/4)	(3.3/5.1)	(3.4/4.9)	(3.9/5)
Average number of Core necessary	(2.83333/3.5)	(3.3/5)	(3.4/4.9)	(3.9/5)
Number of systems scheduled	(6/8)sur 10	(10/10)sur 10	(10/10)sur 10	(10/8)sur 10

Utilisation de 350

Average number of preemption	(0/28.2222)	(1048.4/12.4)	(5334.4/20.8889)	(17629.8/1867)
Average number of migration	(0/28.2222)	(759.3/10.4)	(3766/7.77778)	(13106.1/1416.25)
Average idle time	(48625.8/48733.8)	(88985.5/146287)	(288914/617550)	(989216/1.7846e+06)
Average study interval	(33097.4/33097.4)	(49165.8/49165.8)	(148626/153938)	(512874/467838)
Average number of Core used	(4/4.11111)	(4/5.8)	(4/5.88889)	(4/5.75)
Average number of Core necessary	(3.33333/3.33333)	(4/5.4)	(4/5.77778)	(4/5.75)
Number of systems scheduled	(9/9)on 10	(10/10)on 10	(10/9)on 10	(10/8)on 10