Rapport ENSTA 2021 – ROB305 – Howard Wolowitz

# Mesure de temps et échantillonnage en temps

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed non risus. Suspendisse lectus tortor, dignissim sit amet, adipiscing nec, ultricies sed, dolor. Cras elementum ultrices diam. Maecenas ligula massa, varius a, semper congue, euismod non, mi.

## Gestion simplifiée du temps Posix

Nous avons implémenté un ensemble de fonctions facilitant l’utilisation de la structure timespec, ainsi que surchargé des opérateurs pour pouvoir les utiliser. Nous les avons ensuite testés.

## Timers avec callback

Nous avons implémenté un *callback* qui incrémente un compteur, comme on peut le voir ci-dessous.

void handler(int /\*sig\*/, siginfo\_t\* si, void\* /\*unused\*/)

{

int\* p\_counter = (int\*) si-> si\_value.sival\_ptr;

\*p\_counter += 1; // ou p\_counter[0]

std::cout << \*p\_counter << std::endl;

}

## Fonction simple consommant du CPU

Pour un compteur de 2000, les valeurs finales obtenues sont :

2000

time : 0,000075313s

## Mesure du temps d’exécution d’une fonction

En calibrant la fonction *incr()* de manière linéaire (ax+b), nous avons trouvé comme paramètres

a : 1.81389e+07

b : 18720

En faisant des vérifications entre le temps estimé et le temps réel, nous avons obtenu comme résultats :

time : 55,117513781s

expected time : 55,129140625s

diff time : -1,988373156s

Donc l’estimation est plutôt satisfaisante.

## Amélioration des mesures

todo

# Familiarisation avec l’API multitâches pthread

## Exécution sur plusieurs tâches sans mutex

Après plusieurs exécutions, nous remarquons que le compteur n’a jamais la valeur attendue,

elle est toujours inférieure. Un exemple est donné ci-dessous :

nLoops = 10000, nTasks = 50

counter = 86223

Les différents threads qui modifient le compteur se font suspendre pendant leurs modifications, donc la valeur du compteur n’est pas à jour lorsqu’un autre thread prend la main. Il faut ajouter un *mutex* sur le compteur pour résoudre ce problème.

## Mesure de temps d’exécution

Nous avons mesuré le temps d’exécution pour les paramètres demandés avec la priorité *SCHED\_RR*.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Ntasks \ nLoops | 1,00E+07 | 2,00E+07 | 3,00E+07 | 4,00E+07 |
| 1 | 0,8107 | 0,1596 | 1,2382 | 1,3163 |
| 2 | 0,387 | 1,7798 | 1,1568 | 2,5448 |
| 3 | 1,2236 | 3,4495 | 5,6759 | 7,9004 |
| 4 | 2,7276 | 5,3206 | 8,1586 | 10,1834 |
| 5 | 2,8067 | 6,1146 | 9,3115 | 12,6413 |
| 6 | 3,2901 | 7,287 | 11,8882 | 15,933 |

Nous en déduisons que l’architecture de notre processeur est todo.

## Exécution sur plusieurs tâches avec mutex

Nous avons protégé le compteur avec un mutex, et la valeur du compteur est alors celle qui est attendue. En voici un exemple ci-dessous :

nLoops = 10000000, nTasks = 5, protect = 1, schedPolicyInput = SCHED\_RR

counter = 5e+07

time : 29,059909937s

On peut remarquer en revanche que le temps d’exécution est très largement supérieur à ce qui a été observé à la question précédente. C’est dû au fait que les threads doivent maintenant attendre les uns sur les autres, au lieu de faire chacun leur tâche de leur côté.

# Classes pour la gestion du temps

## Classe Chrono

Nous avons créé une classe *Chrono* qui se comporte comme un chronomètre.

## Classe Timer

Dans les classes *Timer* et *PeriodicTimer*, les constructeurs, destructeurs et méthodes *start* et *stop* sont publiques parce que n’importe quel programme peut créer un timer, le démarrer et l’arrêter.

En revanche, la fonction *callback* est protégée parce que les programmes extérieurs n’ont pas à y avoir accès puisque son appel est géré quand le timer se termine, mais les classes qui en dérivent doivent l’implémenter.

La fonction *call\_callback* est privée parce qu’elle est appelée uniquement par une instance de *Timer* elle-même, au moment où le timer se termine.

Todo : pourquoi timer\_t tid est protégé

La méthode *call\_callback* sert à appeler la fonction à exécuter quand le timer se termine. Dans le lancement d’un timer posix, la fonction qui est appelée lorsque le timer se termine a des arguments précis, donc faire une fonction *call\_callback* permet plus de libertés sur la « vraie » fonction *callback*. Elle est statique parce que dans les arguments de cette fonction il ne peut pas y avoir de type Timer en premier argument.

La fonction *callback* doit être définie comme virtuelle puisque chaque classe qui en hérite doit définir quelles actions sont souhaitées lorsque le timer se termine.

## Calibration en temps d’une boucle

Nous avons implémenté les classes *Looper*, *Calibrator* et *CpuLoop*, et nous les avons testées grâce à la classe *Chrono*.

# Classes de base pour la programmation multitâche

## Classe Thread

Nous avons implémenté les classes *PosixThread* et *Thread* encapsulant les fonctions de gestion de thread Posix.

Pour les tester, nous avons implémenté la class *ThreadIncr* qui incrémente un compteur. Nous retrouvons les mêmes problèmes de compteur qu’au TP2a, ce qui est normal.

$ ./td4a.out 10000000 5

counter = 1.25398e+07

Todo : *Imaginez...*

## Classes Mutex et Mutex::Lock

Nous avons implémenté *Mutex*, *Mutex::Monitor, Mutex::Lock* et *Mutex::TryLock*. Pour les tester, nous avons réutilisé la même classe *ThreadIncr* qu’à la question précédente, agrémentée d’un mutex. Et cela résout les problèmes de compteur, comme au TP2c :

$ ./td4b.out 10000000 5

counter = 5e+07

## Classe Semaphore

Nous avons implémenté la classe *Semaphore* qui est un compteur de jetons et permet à un nombre fini de threads de s’exécuter.

## Classe Fifo multitâches

# Inversion de priorité