Rapport ENSTA 2021 – ROB305 – Madeleine BECKER, Iad ABDUL RAOUF

# Mesure de temps et échantillonnage en temps

## 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 une *callback*, appelée par un timer, 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) avec seulement deux mesures (nombre minimum pour l’estimation de ces deux paramètres), 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

D’une mesure à l’autre les temps d’exécution varie légèrement pour un même nombre d’itérations. Pour mieux calibrer a et b, on peut donc faire n mesures de temps au lieu de deux avec n idéalement très grand puis faire une régression linéaire pour trouver les meilleurs paramètres a et b. C’est d’ailleurs ce que nous avons fait, dans le td3c au sein de la classe *Calibrator*.

# 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

L’accès concurrent au même espace mémoire conduit à ignorer certaines incrémentations. En effet un thread peut incrémenter une valeur obsolète ne correspondant plus à la nouvelle valeur du compteur déjà incrémenté par un autre thread. 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 |

Si on fixe le nombre d’itération par tâche, il semblerait que le temps d’exécution augmente significativement avec le nombre de tâches. Or si le processeur était multicœurs, ce temps ne devrait pas ou peu augmenter.

Nous en déduisons que notre processeur n’a qu’un cœur.

## 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 remarque 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 tâches se retrouvent souvent dans l’état bloqué, en attente de mutex, au lieu d’incrémenter counter chacune 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 pour que n’importe quel programme puisse 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. Son appel est géré quand le timer arrive a échéance et ce sont les classes qui en dérivent qui doivent l’implémenter.

La fonction *call\_callback* est privée celle ci n’existe que pour appeler *callback* tout en repectant le type utilisé par la libraire time.h. Il n’y a aucune raison de la modifer dans les classes dérivé ou de l’appeler de l’exterieur. De plus,

L’identifiant du timer tid est protégé car la classe dérivée PeriodicTimer a besoin d’y accéder afin de remplir ses champs tel qu’un timer récurrent soit lancé, au lieu du timer à échéance unique de la classe Timer.

Pour comprendre la raison d’être de *call\_callback*, il faut s’interroger sur la possibilité de fournir directement *callback* au timer. Malheureusement, ce n’est pas possible car une méthode non statique est d’un type particulier. Le type de cette fonction inclut dans ses paramètres la classe qui la contient. Il n’est pas non plus possible de définir *callback* comme une méthode static parce que cela rendrait inaccessible les membres de la classe, qui sont potentiellement nécessaire aux calculs réalisés à l’intérieur de la fonction.

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

Remarque générale sur le TD-4 : Ils est demandé à plusieurs reprise de coder des fonctions renvoyant un booléen en fonction du succès ou de l’échec en cas de *timeout* notamment. Cependant la valeur de ce booléen n’est pas précisée. Nous avons donc dû faire un choix que nous avons conservé pour toutes les fonctions pour avoir une certaine cohérence. En C la gestion d’erreur se fait en renvoyant 0 dans le cas normal et une valeur non nulle dans le cas anormal. Or false est associé à la valeur 0, donc nos fonctions renvoient false lorsque tout s’est déroulé comme attendu.

## 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

Nous n’avons pas implémenté la modification de thread avec le l’ajout de started car nous n’avons pas compris la différence entre isActive (déjà présent dans la classe PosixThread) et started.

## 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. Celle ci a aussi été testé à l’aide des classe TokenConsumer et TokenProducer. La verification de la modification du compteur a été faite en affichant (par std ::cout) sur la console la valeur du compteur à chaque appelle de give ou take. On peut aussi vérifier que la valeur finale du compteur doit être égale à nProd – nCons (car chacun des thread ne produit ou ne concomme qu’un jeton). De même on verifie que si nCons est plus grand que nProd alors le main est dans un état bloqué à cause des appels bloquant de join sur des tâches consomatrices qui ne finnessent jamais.

## Classe Fifo multitâche

Nous avons implémenté la classe *Fifo* qui encapsule une FIFO, et nous l’avons testé en la remplissant et vidant des suite d’entiers consécutifs à l’aide des classe intConsumer et intProducer. IntProducer produit dans l’ordre les entier 0, 1, …, n tandis que IntConsumer les attends aussi dans l’ordre. Il y a evidement plusieurs instances de ces classe accedant à la fifo de manière simultanée. IntConsumer ne sait pas à l’avance si l’entier récuéré via la méthode pop est celui attendu à l’instant présent. Dans le cas ou il s’agit d’un entier trop grand, ou d’un entier déjà consomé, celui ci est réinjecté dans la fifo pour pouvoir être récupéré plus tard (part lui même ou par un autre IntConsomer). Certe c’est extremement inefficace, mais l’objectif ici est de tester la classe Fifo. IntConsomer et IntProducer n’ont aucune autre utilié et le temps ecxecif nécéssaire à la réalisation de ce simple programme n’est pas un problème.

# Inversion de priorité

Nous n’avons pas eu le temps d’aborder ce TP.