Exclusion Mutuelle

Burkhalter - Lienhard

Version 1.0
3/16/2010 9:01:00 AM

Table des matières

Exclusion Mutuelle

Cette documentation décrit le programme Exclusion Mutuelle qui consiste à tester et comparer différents algorithmes d'exclusion mutuelle, en comparant le temps nécessaire à chacun d'eux pour s'exécuter. Cela a été mis en plage dans le laboratoire n°2 du cours PCO.

Version:
1.0

Introduction

Le but de ce programme consiste en la réalisation d'un programme permettant de comparer les performances de différents algorithmes permettant l'exclusion mutuelle. Les 3 algorithmes en question, sont:

- l'algorithme 2.2 du cours
- l'algorithme de Dekker
- l'algorithme de Peterson

Comparaison

Afin de comparer les différents algorithmes d'exclusion mutuelle, nous avons fait en sorte d'exécuter chacun d'entre eux l'un après l'autre et de manière indépendante, afin de pouvoir récupérer les temps nécessaires à chacun d'eux pour effectuer une opération critique (identique aux 3 algorithmes). Pour avoir des valeurs corrects et représentant uniquement le temps utile aux algorithmes et non pas avec les différentes tâches associées, tel que la création des threads, nous avons procédé de la manière suivante: Au début de chaque tâche, nous avons ajouté une partie permettant la synchronisation des deux tâches d'un algorithme entre elles. Seulement une fois les deux tâches prêtes, nous commençons l'exécution de l'algorithme et récupérons le temps de début. Une fois l'exécution finie, nous récupérons le temps de fin, et part la suite nous pouvons calculer le temps effectif de l'algorithme en soustrayons le temps de la première tâche ayant débuté l'algorithme au temps de fin de la dernière tâche.

Matériel

Les propriétés de la machine sur laquelle ont été réalisées les mesures suivante:

- Processeur: Intel Core 2 Duo T8100 2.1 GHz
- RAM : 4 Go
- Système d'exploitation: Windows 7 Professionnal sur 32 bits

Résultats

Les résultats que nous avons obtenus, et qui vont être présenté ci-dessous, portent sur 15 mesures par algorithmes. Nous avons décidé de prendre un suffisant de mesure afin d'éviter qu'une valeur atypique n'aille une influence trop importante sur les résultats finaux. Les résultats obtenus et représenté par les graphiques disponibles en annexe nous ont permit de faire ressortir les observations suivantes: Premièrement nous pouvons constater que malgré ça simplicité, l'algorithme de Peterson est le plus gourmand en temps. En moyenne, celui-ci nécessite environ 60% de temps en plus que l'algorithme de Dekker. Cela est sans doute du au fait que la boucle while exécutée par le thread attendant son tour contient deux conditions, contrairement aux deux autres algorithmes, qui eux n'en contiennent qu'une. Les graphiques nous permettent clairement de constater que l'algorithme de Dekker ainsi que l'algorithme 2.2

du cours nécessitent à peu près le même temps, se qui permet de mettre l'algorithme de Dekker en avant, du fait que celui-ci vérifie la règle 3 des propriétés des algorithmes, contrairement au second algorithme, qui oblige les deux tâches à fonctionner en même temps.

Index des fichiers

Liste des fichiers

Liste de tous les fichiers documentés avec une brève description :

Documentation des fichiers

RÃ©fÃ©rence du fichier C:/Users/You/Arnaud/HEIG/2eme/PCO/Laboratoire/Labo2_Exlusion_mutuelle/main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdbool.h>
#include <time.h>
```

Fonctions

- void * **algo1_T0** (void *arg)
- void * **algo1_T1** (void *arg)
- void * **algoDekker_T0** (void *arg)
- void * **algoDekker_T1** (void *arg)
- void * **algoPeterson_T0** (void *arg)
- void * **algoPeterson_T1** (void *arg)
- int **main** ()

Variables

- bool **etat_algo1** [2] = {false, false}
- bool **etat_algoDekker** [2] = {false, false}
- bool **etat_algoPeterson** [2] = {false, false}
- int **tour_algo1** = 0
- int **tour_algoDekker** = 0
- int **tour_algoPeterson** = 0
- clock_t **start_tache0_algo1**
- clock_t **start_tache1_algo1**
- clock_t **end_algo1**
- clock_t **start_tache0_algoDekker**
- clock_t **start_tache1_algoDekker**
- clock_t **end_algoDekker**
- clock_t **start_tache0_algoPeterson**
- clock_t **start_tache1_algoPeterson**
- clock_t **end_algoPeterson**
- const int **NB_ITERATION** = 100000000
- int **cpt_algo1**
- int **cpt_algoDekker**
- int **cpt_algoPeterson**
- bool **readyTask0_algo1** = false
- bool **readyTask1_algo1** = false
- bool **readyTask0_algoDekker** = false
- bool **readyTask1_algoDekker** = false
- bool **readyTask0_algoPeterson** = false
- bool **readyTask1_algoPeterson** = false
- double **tempsExecution_algo1**
- double **tempsExecution_algoDekker**
- double **tempsExecution_algoPeterson**

Description détaillée

Auteur:
Steve Lienhard et Arnaud Burkhalter

Date:
16.03.2010

Version:
1.0

Description

Ce fichier définit les

Tests

Afin de tester que les algorithmes soient exécuté de manières correcte et contrôler que l'exclusion mutuelle aille bien lieu, nous avons affiché, à la fin de l'exécution de chaque thread, la variable globale sur laquelle les différentes tâches de chaque algorithme ont travaillées. Cela afin de vérifier que la variable aille été accédée un nombre correcte de fois, soit deux fois le nombre d'itérations d'une section critique. Dans notre cas, deux fois la constante NB_ITERATIONS, donc 200000000.

Documentation des fonctions

void* algo1_T0 (void * *arg*)

But : Fonction qui sera exécutée par un thread traitant la tâche 0 de l'algorithme de l'exemple 2.2 du support de cours. Dans une première phase, la fonction observera la seconde tâche de l'algorithme et attendra que celle-ci soit prête. Si celle-ci n'est pas encore prête, nous entrerons dans une boucle en attendant le feu vert de l'autre tâche, se qui sera fait à l'aide de la variable readyTask1_algo1. Une fois les deux tâches prêtes, nous commençons l'algorithme et récupérons le temps de début de celui-ci pour la tâche en cours. Lorsque l'algorithme est terminé, nous récupérons le temps de fin de celui-ci, que nous stockons dans end_algo1. Cette variable est identique pour les deux tâches car nous avons uniquement besoin du temps de la tâche finissant l'algorithme en dernier.

Paramètre(s): arg : pointeur passé à la fonction. Il est inutilisé, mais permet à la fonction de correspondre au prototype nécessaire pour que celle-ci puisse être utilisée par un thread.

void* algo1_T1 (void * *arg*)

But : Fonction qui sera exécutée par un thread traitant la tâche 1 de l'algorithme de l'exemple 2.2 du support de cours. Dans une première phase, la fonction observera la seconde tâche de l'algorithme et attendra que celle-ci soit prête. Si celle-ci n'est pas encore prête, nous entrerons dans une boucle en attendant le feu vert de l'autre tâche, se qui sera fait à l'aide de la variable readyTask0_algo1. Une fois les deux tâches prêtes, nous commençons l'algorithme et récupérons le temps de début de celui-ci pour la tâche en cours. Lorsque l'algorithme est terminé, nous récupérons le temps de fin de celui-ci, que nous stockons dans end_algo1. Cette variable est identique pour les deux tâches car nous avons uniquement besoin du temps de la tâche finissant l'algorithme en dernier.

Paramètre(s): arg : pointeur passé à la fonction. Il est inutilisé, mais permet à la fonction de correspondre au prototype nécessaire pour que celle-ci puisse être utilisée par un thread.

void* algoDekker_T0 (void * *arg*)

But : Fonction qui sera exécutée par un thread traitant la tâche 0 de l'algorithme de Dekker. Dans une première phase, la fonction observera la seconde tâche de l'algorithme et attendra que celle-ci soit prête. Si celle-ci n'est pas encore prête, nous entrerons dans une boucle en attendant le feu vert de l'autre tâche, se qui sera fait à l'aide de la variable readyTask1_algoDekker. Une fois les deux tâches prêtes, nous commençons l'algorithme et récupérons le temps de début de celui-ci pour la tâche en cours. Lorsque l'algorithme est terminé, nous récupérons le temps de fin de celui-ci, que nous stockons dans end_algoDekker. Cette variable est identique pour les deux tâches car nous avons uniquement besoin du temps de la tâche finissant l'algorithme en dernier.

Paramètre(s): arg : pointeur passé à la fonction. Il est inutilisé, mais permet à la fonction de correspondre au prototype nécessaire pour que celle-ci puisse être utilisée par un thread.

void* algoDekker_T1 (void * *arg*)

But : Fonction qui sera exécutée par un thread traitant la tâche 1 de l'algorithme de Dekker. Dans une première phase, la fonction observera la seconde tâche de l'algorithme et attendra que celle-ci soit prête. Si celle-ci n'est pas encore prête, nous entrerons dans une boucle en attendant le feu vert de l'autre tâche, se qui sera fait à l'aide de la variable readyTask0_algoDekker. Une fois les deux tâches prêtes, nous commençons l'algorithme et récupérons le temps de début de celui-ci pour la tâche en cours. Lorsque l'algorithme est terminé, nous récupérons le temps de fin de celui-ci, que nous stockons dans end_algoDekker. Cette variable est identique pour les deux tâches car nous avons uniquement besoin du temps de la tâche finissant l'algorithme en dernier.

Paramètre(s): arg : pointeur passé à la fonction. Il est inutilisé, mais permet à la fonction de correspondre au prototype nécessaire pour que celle-ci puisse être utilisée par un thread.

void* algoPeterson_T0 (void * *arg*)

But : Fonction qui sera exécutée par un thread traitant la tâche 0 de l'algorithme de Peterson. Dans une première phase, la fonction observera la seconde tâche de l'algorithme et attendra que celle-ci soit prête. Si celle-ci n'est pas encore prête, nous entrerons dans une boucle en attendant le feu vert de l'autre tâche, se qui sera fait à l'aide de la variable readyTask1_algoPeterson. Une fois les deux tâches prêtes, nous commençons l'algorithme et récupérons le temps de début de celui-ci pour la tâche en cours. Lorsque l'algorithme est terminé, nous récupérons le temps de fin de celui-ci, que nous stockons dans end_algoPeterson. Cette variable est identique pour les deux tâches car nous avons uniquement besoin du temps de la tâche finissant l'algorithme en dernier.

Paramètre(s): arg : pointeur passé à la fonction. Il est inutilisé, mais permet à la fonction de correspondre au prototype nécessaire pour que celle-ci puisse être utilisée par un thread.

void* algoPeterson_T1 (void * *arg*)

But : Fonction qui sera exécutée par un thread traitant la tâche 1 de l'algorithme de Peterson. Dans une première phase, la fonction observera la seconde tâche de l'algorithme et attendra que celle-ci soit prête. Si celle-ci n'est pas encore prête, nous entrerons dans une boucle en attendant le feu vert de l'autre tâche, se qui sera fait à l'aide de la variable readyTask0_algoPeterson. Une fois les deux tâches prêtes, nous commençons l'algorithme et récupérons le temps de début de celui-ci pour la tâche en cours. Lorsque l'algorithme est terminé, nous récupérons le temps de fin de celui-ci, que nous stockons dans end_algoPeterson. Cette variable est identique pour les deux tâches car nous avons uniquement besoin du temps de la tâche finissant l'algorithme en dernier.

Paramètre(s): arg : pointeur passé à la fonction. Il est inutilisé, mais permet à la fonction de correspondre au prototype nécessaire pour que celle-ci puisse être utilisée par un thread.

int main ()

But : Fonction principale qui a pour but de tester les différents algorithmes d'exclusion mutuelle. Pour chaque algorithme deux threads sont créés et exécutés, un pour chaque tâche. Après l'exécution de chaque algorithme, nous calculons le temps qu'il a fallu aux tâches pour s'exécuté, et affichons celui-ci.

Résultat: EXIT_SUCCESS : int qui indique si le programme s'est terminé de manière correct ou non.

Index