Travaux pratiques de Tolérance aux Fautes

L 'ensemble des fichiers nécessaires pour la séance sont rassemblés dans l'archive indiqué sur le site de l'UE.

Le TP est organisé en deux parties permettant d'illustrer les concepts suivants vus en cours :

1. Illustration de la nécessité d'appliquer élimination et tolérance aux fautes
2. Recovery Blocks / capture d'état et diversification pour la tolérance de fautes de développement non maitrisées

Partie I Fautes transientes, débuggage vs tolérance

Introduction

Cette partie a pour objectif de vous faire implémenter un petit programme jouet qui est tout de même représentatif des composants d'un systèmes asservi supervisé par un humain.

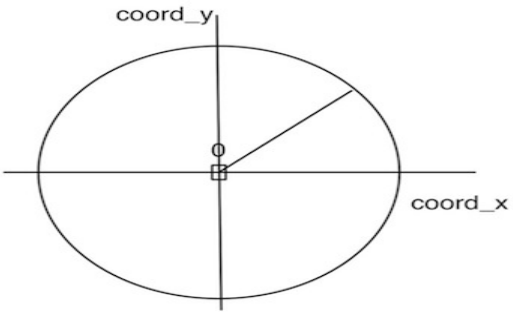
Nous allons durant cette partie voir éléments en rapport avec la tolérance aux fautes logicielle :

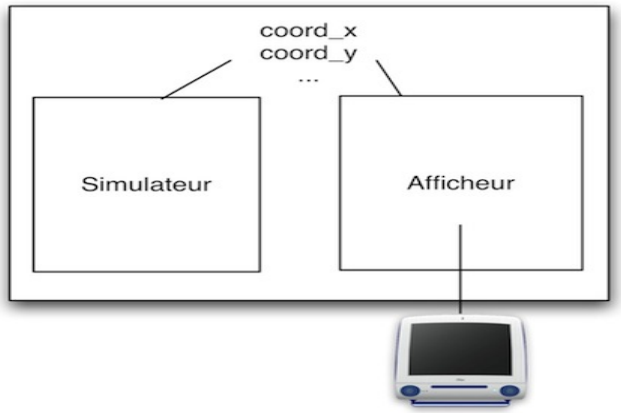
1. Un scénario d'élimination de faute qui consiste à modifier un code existant pour éviter éliminer une faute. (ce n'est pas de la tolérance, cela a pour but de clarifier la différence entre le débuggage et le tolérance)
2. Observer l'activation d'une faute engendrant des défaillances transitoires.
3. Mettre en place un mécanisme de tolérance aux fautes reposant sur un principe d'enveloppe pour un appel système de synchronisation

Allez dans le répertoire *synchro*

Description du programme étudié et du système modélisé

Nous allons étudier deux fonctions d'un système qui pilote et permet la supervision du déplacement d'un chariot dans le plan. Une première fonction est en charge de mettre à jour les coordonnées du chariot (le chariot se déplace aux coordonnées indiquées). Une seconde fonction s'exécutant en parallèle permet de visualiser un rapport des dernières coordonnées occupées par le chariot. Chaque rapport affiche une lecture de l'abscisse x, de l'ordonnée y, et de la valeur x^2+y^2. Le mouvement attendu du chariot est un déplacement le long du cercle de centre 0, et de rayon 1 (dans base dans laquelle les coordonnées sont définies).





Le programme `systeme.c` contient le code permettant de simuler le déplacement dans le plan d'un point sur un cercle de rayon 1 et de centre 0 (cf figure).

```
void simu_systeme(){
        long valeur_suivante;
        while(cont>0){
        valeur_suivante=(long)time((time_t*)NULL);
        coord_x=cos(sens*vitesse*(valeur_suivante-origine)/36.0*2*3.14);
        // temporisation pour simuler la difficulte du calcul
#ifdef TEMPO
        usleep (2*TEMPO);
#endif
        coord_y= sin(sens*vitesse*(valeur_suivante-origine)/36.0*2*3.14);
        usleep(30000);
        }}
```

La ligne surlignée en bleu calcul le temp écoulé depuis la dernière mise à jour de x et y. Cette valeur sert à calculer coord_x et coord_y en conséquence. On notera qu'une légère erreur est commise sur le calcul de coord_x et coord_y car du temps s'écoule entre les deux mises à jour. Ceci n'est pas un problème si ce temps est faible.

En parallèle de ce programme nous exécutons le code suivant de supervision :

```
void afficheur () {
  while(cont>0){
    printf("X : %f;",coord_x);
#ifdef TEMPO
    usleep(TEMPO);
#endif
    printf(" Y : %f;",coord_y);
    printf(" R : %f;\n",coord_x*coord_x+coord_y*coord_y);
    usleep(1000000);
  }
}
```

Ce code doit permettre d'afficher périodiquement un rapport de la position du chariot (a priori toujours sur le cercle).

Le code présent dans le main permet de lancer les thread exécutant chaque programme vous n'avez la possibilité que d'ajouter des lignes dans cette partie du code.

Exercice 1 : absence de synchronisation, élimination de faute

Compilez-le code par la commande `make V1` qui génère `sv1`, puis exécutez le. Utilisez ensuite la commande `make VT1` qui génère le fichier `sv1tempo` et ré-exécuter le programme. La seule

différence réside dans l'ajout de pauses dans l'exécution du programme. La faute à l'origine de ce problème est une faute de développement se manifestant de manière transitoire : identifiez son origine (ici, c'est ce que l'on appelle communément un "heisen" bug car il ne se manifeste pas de manière déterministe.

Indice : la défaillance désynchronise l'affichage de x et y ...).

Proposez une correction du code en introduisant une synchronisation reposant sur un sémaphore (ceci est une contrainte de réalisation imposée). Nommez le nouveau fichier systeme_synch.c

(PS: on vous demande de ne pas supprimer de ligne dans le code du main.c).

Le problème de data race concerne la portion de code modifiant coord_x et coord_y et celle lisant ces valeurs. L'usage du sémaphore doit encadrer ces portions de code pour en assurer l'exécution non entrelacée.

Sur internet ou en ligne de commande tapez `man semaphore.h` et parcourez la description des fonctions disponibles. En priorité, vous utiliserez `sem_init()`, `sem_wait()` et `sem_post`.

Laissez intactes les macros "TEMPO", ces macros nous permettront de vérifier que l'exclusion mutuelle est bien assurée. vous pourrez compiler votre programme avec la commande `make VT1`.

Exercice 2 Tolérance aux fautes

Le problème est-il réglé ?

- Lancez le programme ainsi corrigé dans un terminal (ne l'arrêtez pas)
- Ouvrez un nouveau terminal, placez vous dans le répertoire de l'exercice en cours. Puis, lancez `sh benchit.sh sv1tempo`.
- Observez le comportement de l'application pendant environ 30 secondes.

Origine du comportement particulier du programme

Un appel système est un service qui intègre la notion de mode de défaillance. Il faut bien lire et analyser les "codes de retour d'erreur" des appels systèmes car ils identifient les modes et causes de défaillances des différents services utilisés.

En consultant le contenu du script utilisé pour "stresser le programme", proposez une explication permettant de justifier le comportement du programme.

La défaillance dure-t-elle dans le temps ?

Proposez une approche par enveloppe de l'appel à sem_wait permettant de tolérer l'activation de la faute de synchronisation.

Partie I

Implémentation du concept de Recovery block en C vs Java

Nous souhaitons que vous implémentiez un ensemble de fonction permettant de mettre à disposition le principe de recovery block pour un code séquentiel de temps d'exécution borné. Cette implémentation devra être fournie en C dans un premier temps, puis en Java)

Le principe est le suivant :

- Hypothèse : les fonctions ont la signature long <name > (long)
- Vous considérerez les deux implémentations fact1 et fact2 de la fonction factorielle (l'une

est défaillante, l'autre non (cas de défaillance par dépassement de capacité).

- Le recovery block exécute fact1 puis fact2 si la représentation ne passe pas le test résultat >0
- La réexécution doit restaurer la pile d'exécution
- Les deux implémentation mémorisent dans la variable globale rec_call le nombre d'appels récursifs réalisés. Vous devez vous assurer que ce nombre retourne bien le nombre d'appels récursifs correctement exécutés.

Vous pourrez utiliser setjmp et longjmp pour implémenter la capture de l'état de la pile. Il era nécessaire d'assure la restauration de l'état de la mémoire manuellement. L'implémentation du récovery block prendra comme paramètre un tableau de pointeur de fonction et le paramètre d'appel.