Université de Rouen Département Informatique

Master Sécurité des Systèmes d'Informations

Rapport de devoir

Devoir de Model-Checking

QuickSort

Auteur :

Alexis Osmont

Réfèrent :

M. Bedon Nicolas

19 décembre 2021

Table des matières

I - Rappel du problème

II - Détails implantatoires

III – Vérifications

IV – Difficultés

V – Conclusion

I – Rappel du problème

Le but du devoir était de vérifier, en utilisant SPIN, que le quicksort réalise bien ce qui lui est demande, c'est-à-dire, trier un tableau. L'algorithme utilise une méthode particulière pour le partitionnement du tableau sans swap (contrairement à d'autre algorithme de quicksort), bien connue car pédagogique, mais qui n'est pas la méthode de partitionnement originelle.

Algorithme de tri rapide :

```
// Sorts a (portion of an) array, divides it into partitions, then
sorts those
algorithm quicksort(A, lo, hi) is
  // If indices are in correct order
  if lo >= hi || lo < 0 then
    return

    // Partition array and get the pivot index
    p := partition(A, lo, hi)

    // Sort the two partitions
    quicksort(A, lo, p - 1) // Left side of pivot
    quicksort(A, p + 1, hi) // Right side of pivot

// Divides array into two partitions
algorithm partition(A, lo, hi) is
  pivot := A[hi] // Choose the last element as the pivot

  // Temporary pivot index
  i := lo - 1

  for j := lo to hi do
    // If the current element is less than or equal to the pivot
    if A[j] <= pivot then
      // Move the temporary pivot index forward
      i := i + 1

      // Swap the current element with the element at the temporary
pivot index
      swap A[i] with A[j]
  // Move the pivot element to the correct pivot position (between the
smaller and larger elements)
  i := i + 1
  swap A[i] with A[hi]
  return i // the pivot index
```

Il nous était demandé de montrer que la fonction de partitionnement effectue bien son travail, donc que les éléments à gauche du pivot sont plus petits ou égale à lui-même et que les éléments à droite du pivot sont plus grands ou égale à lui-même.

Le deuxième point était de vérifier que le tableau avait été bien trié en fin d'exécution du quicksort.

II – Détails implantatoires

Pour implanter l'algorithme de QuickSort j'ai donc suivi à la lettre l'algorithme donné sur le document de projet et sur le Wikipédia QuickSort. J'ai séparé le programme en 4 modules : le QuickSort, le partitionnement , la génération aléatoire du tableau et l'initialisation.

Variable Globales :

Comme variable globale j'ai choisi d'utiliser le tableau général (**tab**) pour permettre une plus grande liberté et le canal, (**canal_global**) car il est nécessaire pour permettre au QuickSort de ne commencer qu'une fois la génération terminée et pour que la vérification finale du tri du tableau (**VERIFY_SORT**) soit faite après la fin de tout appels récursifs du QuickSort.

Initialisation :

Pour cette partie j'ai commencé par un appel à la fonction de génération aléatoire que je détaillerais ci-dessous. Une fois cette génération terminé j'ai lancé un canal d'écoute qui permet d'attendre que la génération soit terminée, s'en suit donc l'appel de ma fonction QuickSort afin de trier ce tableau.

J'ai choisi de passer mon canal global en paramètre a mon quick sort pour permettre d'attendre que celui-ci se finisse puis d'afficher ma vérification de tri. Comme vous l'avez compris j'ai donc ma vérification de tri en fin d'**init**.

```
// Initialisation {init}
init
{
    // Fonction de generation aléatoire
    run random();
    canal_global?_;
    // Lancement du quick-sort
    printf("Lancement du quicksort ...\n\n");
    run quicksort(canal_global, 0, SIZE-1);
    canal_global?_;
```

Génération aléatoire :

Pour générer mon tableau aléatoirement j'ai attribué une valeur aléatoire a chaque élément du tableau grâce à l'élément **select(variable, borne_inferieure .. borne_supérieure).** L'élément variable correspond à l' élément généré, les éléments **borne_inferieure** et **borne_supérieure** sont les limites fixées pour les variables (Peuvent être changé grâce aux macros constantes).

QuickSort :

En ce qui concerne le QuickSort, comme dit précédemment j'ai implanté l'algorithme QuickSort comme définit dans l'énoncé du devoir. Cependant j'ai dû ajouter des canaux afin d'éviter toute concurrence entre les appels récursif, j'ai par conséquent définit un canal au sein de **QuickSort**, ce canal est transmis en paramètre de la fonction de partitionnement (**partition**), qui elle émet depuis ce canal pour signifier que le partitionnement c'est bien fini. Le canal est en mode écoute avant le premier appel récursif (attend que le partitionnement se finisse), la fonction récursive a droite effectue son travail. Il est fait de même pour la partie récursivité a gauche même si celle-ci attend que les appels récursifs à droite soient entièrement finis pour commencer et éviter les chevauchements et la concurrence.

Une fois tous ces appels effectués, le canal global envoie en mode émission un élément pour qu'un déblocage s'effectue et laisse la vérification avoir lieu.

Partitionnement :

En ce qui concerne la partie partition, comme dit précédemment j'ai implanté l'algorithme comme définit dans l'énoncé du devoir, en ajoutant une émission dans le canal donné en argument pour que celui-ci se termine avant d'effectuer de nouveaux appels récursifs .

```
// Fonction de partition
proctype partition(chan canal_param; int lo, hi)
{
    int pivot,tmp,j;
    short i = lo -1;
    pivot = tab[hi];
    for (j : lo .. hi) {
        if
        :: tab[j] <= pivot ->
        i = i+1;
        // Swap
        tmp = tab[j];
        tab[j] = tab[i];
        tab[i] = tmp;
        :: else
        fi
    }
    // Envoie du pivot au quick sort
    canal_param!i;
}
```

```
proctype quicksort(chan canal_param; int low, high)
{
    // Initialisation canal local
    chan canal_local = [0] of {int}
    int p;
    if
    :: ((low >= 0) && (high >= 0) && (low < high)) ->
        // Partition
        run partition(canal_local, low, high);
        // Récuperation du p grâce au canal de partition
        canal_local?p;

        run quicksort(canal_local, low, p-1);
        // Bloquage grâce au canal pour ne pas faire les deux quicksort en même temps
        canal_local?_;
        run quicksort(canal_local, p+1, high);
        // Debloquage grâce au canal
        canal_local?_;
    :: else
    fi
    canal_param!0;
}
```

III – Vérifications

En ce qui concerne les vérifications de partition et de tri, j'ai effectué l'une une fois le tri entièrement fini, en fin d'**init** et l'autre au milieu de la fonction partition pour vérifier quelle faisait bien ce qui lui été demandé.

Pour ces vérifications j'ai fait appel à des assertions car, plus simple à mettre en place et tout aussi efficace qu'une LTL.

VERIFY_PARTITION :

J'ai simplement testé si chaque élément du tableau était inferieur ou égal au pivot si son indice l'était aussi, et supérieur ou égal au pivot si son indice l'était également.

```
//----------------VERIFY_PARTITION----------------
#ifdef VERIFY_PARTITION
    printf("VERIFY_PARTITION pivot:%d -> ",i);
    do
    :: i == 0 ->
        for (j : i .. hi-1) {
            assert(tab[i] <= tab[j+1]);
        };
        printf("ok\n");
        break;
    :: i == SIZE-1 ->
        for (j : i .. lo+1) {
            assert(tab[i] >= tab[j-1]);
        };
        printf("ok\n");
        break;
    :: else ->
        for (j : i .. hi-1) {
            assert(tab[i] <= tab[j+1]);
        };
        for (j : i .. lo+1) {
            assert(tab[i] >= tab[j-1]);
        }
        printf("ok\n");
        break;
    od
#endif
```

VERIFY_SORT :

J'ai simplement testé si chaque élément du tableau était inferieur ou égal à l'élément d'indice i + 1 (i étant l'indice courant).

```
//----------------VERIFY_SORT-------------------
#ifdef VERIFY_SORT
    printf("VERIFY_SORT !\n\n");
    int h;
    for(h : 1 .. SIZE-1) {
        assert(tab[h] >= tab[h-1]);
    }
#endif
//----------------------------------------------
```

IV – Difficultés & Données

La difficulté ici aura été l'utilisation des canaux et l'optimisation de l'algorithme, en effet ayant peu pratiqué les canaux au cours des TP il ne m'a pas été simple de les appréhender ou plutôt de les utiliser dans notre cas précis.

L'autre difficulté aura été la gestion de type générique , en effet j'ai utilisé des macros constantes nous permettant de gérer le type des valeurs, la taille du tableau et les bornes des valeurs du tableau.

```
#define SIZE 7
#define VERIFY_PARTITION
//#define VERIFY_SORT
#define ELEMENT_TYPE int
#define BORNE_INF 0
#define BORNE_SUP 9
```

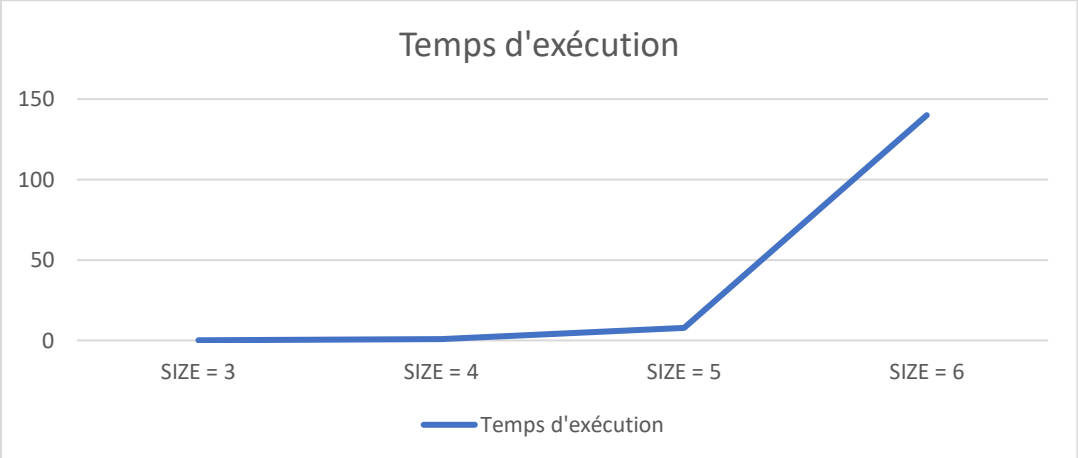
➔ Spin quicksort.pml

```
alexis@alexis-Lenovo-Yoga710-14ISK:~/Documents/M1/Model-checking/projet$ spin quicksort.pml | sed 's/[ ]\+/ /g'
 Génération du tableau aléatoire ... tab[ 9 1 5 3 1 3 ];

 Lancement du quicksort ...

 VERIFY_PARTITION pivot:3 -> ok
 VERIFY_PARTITION pivot:1 -> ok
 VERIFY_PARTITION pivot:5 -> ok
 Tableau après tri : tab[ 1 1 3 3 5 9 ];

12 processes created
```

Après avoir exécuté a de multiples reprise l'algorithme en suivant cette méthode :

➔ Spin -a quicksort.pml
➔ gcc pan.c
➔ ./a.out

J'ai pu obtenir cette courbe de temps d'exécution :



V – Conclusion & Question subsidiaire

J'ai effectué la partie subsidiaire en modifiant mes canaux, en effet il m'a fallu placer deux canaux d'écoute (réception) entre mes deux appels récursifs de quicksort pour que mes quicksort se fasse de manière concurrente. Après ma vérification spin il en est ressorti que mettre en concurrence les appels récursifs ne marche pas car cette méthode ne trie pas correctement le tableau et génère une erreur spin.

Pour conclure notre algorithme de vérification spin a bien fait ce qui lui était demandé, c'est-à-dire vérifier qu'un tableau est correctement trié grâce au QuickSort.