Série d'exercices #6

IFT-2245

10 février 2018

6.1 Lockfree 1

Soit le code suivant qui permet d'insérer des éléments dans un arbre binaire :

```
struct tree {
  int key;
  void *val;
  struct tree *smaller, *larger;
}

struct tree *tree_insert (struct tree *t, int key, void *val)
{
  struct tree *n = malloc (sizeof (struct tree));
  if (t == NULL) {
    n->key = key; n->val = val;
    n->smaller = NULL; n->larger = NULL;
  } else if (key == t->key) {
    n->key = key; n->val = val;
    n->smaller = t->smaller; n->larger = t->larger;
  } else if (key < t->key) {
    n->key = t->key; n->val = t->val;
    n->larger = t->larger;
    n->smaller = tree_insert (t->smaller, key, val);
    return n;
  } else { /* key > t->key */
    n->key = t->key; n->val = t->val;
    n->smaller = t->smaller;
    n->larger = tree_insert (t->larger, key, val);
  }
  return n;
}

void tree_set (struct tree **t, int key, void *val)
{
  *t = tree_insert (*t, key, val);
}
```

1. Trouver les conditions de course présentes si le code est utilisé dans une application à plusieurs *threads*.
2. Corriger ces conditions de course en ajoutant les verrous et opérations correspondantes nécessaires. Décrire clairement quelles données sont protégées par chaque verrou.
3. Corriger ces mêmes conditions de course sans utiliser de verrous, en utilisant à la place une approche de synchronisation optimiste, en utilisant l'opération `compare&swap`.

6.2 Lockfree 2

Soit le code suivant qui permet d'insérer des éléments dans un arbre binaire :

```
struct tree {
  int key;
  void *val;
  struct tree *smaller, *larger;
}

void tree_set (struct tree **t, int key, void *val)
{
  struct tree *n = *t;
  if (n == NULL) {
    n = malloc (sizeof (struct tree));
    n->key = key; n->val = val;
    n->smaller = NULL; n->larger = NULL;
    *t = n;
  } else if (key == n->key) {
    n->val = val;
  } else if (key < n->key) {
    tree_set (&n->smaller, key, val);
  }
  } else { /* key > t->key */
    tree_set (&n->larger, key, val);
  }
}
```

1. Trouver les conditions de course présentes si le code est utilisé dans une application à plusieurs *threads*.
2. Corriger ces conditions de course en ajoutant les verrous et opérations correspondantes nécessaires. Décrire clairement quelles données sont protégées par chaque verrou.
3. Corriger ces mêmes conditions de course sans utiliser de verrous, en utilisant à la place une approche de synchronisation optimiste, en utilisant l'opération `compare&swap`.

6.3 État sûr et interblocage

Développer un scénario où un système commence dans un état *unsafe* et où malgré cela, l'exécution des threads termine sans rencontrer d'interblocage.

6.4 Algorithme du banquier

Soit un système décrit par la table suivante :

	Allocation				*Max*			
	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2
P_1	1	0	0	0	1	7	5	0
P_2	1	3	5	4	2	3	5	6
P_3	0	6	3	2	0	6	5	2
P_4	0	0	1	4	0	6	5	6

Available			
A	B	C	D
1	5	2	0

Selon l'algorithme du banquier :

1. Quel est le contenu de la matrice *Needed* ?
2. Le système est-il dans un état *safe* ?
3. Si une requête arrive du processus P_1 pour les ressources $(0, 4, 2, 0)$, la requête peut-elle être acceptée immédiatement ?