Projet 1 : Mémoire transactionnelle

Taha KHERRAF GITHUB: https://github.com/tounbukto/tmpp

1. Présentation de la solution:

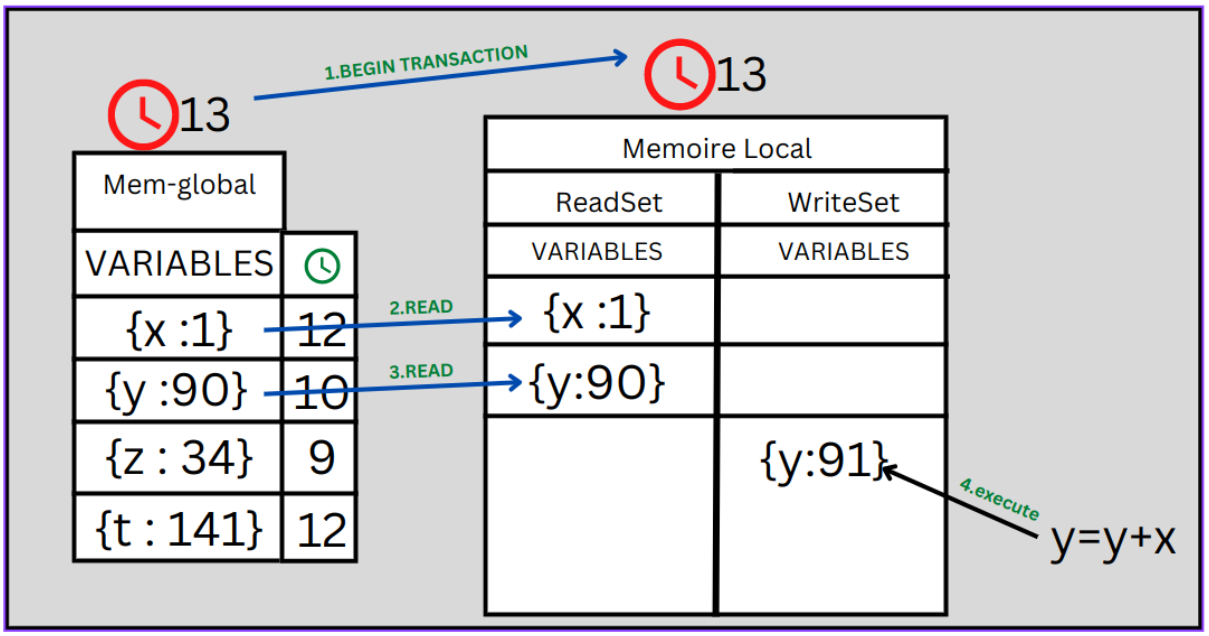
La mémoire transactionnelle se base sur le principe des transactions, pour cela une transaction à une mémoire locale où elle stocke toutes les variables de la mémoire globale dont elle a besoin, elle manipule ses variables locales "exécute des opérations", et à la fin de la transaction on commit.
Un commit est un écriture de la nouvelle valeur des variables de la mémoire locale en mémoire global, pour cela on lock la mémoire global pour assurer l'exclusion mutuelle

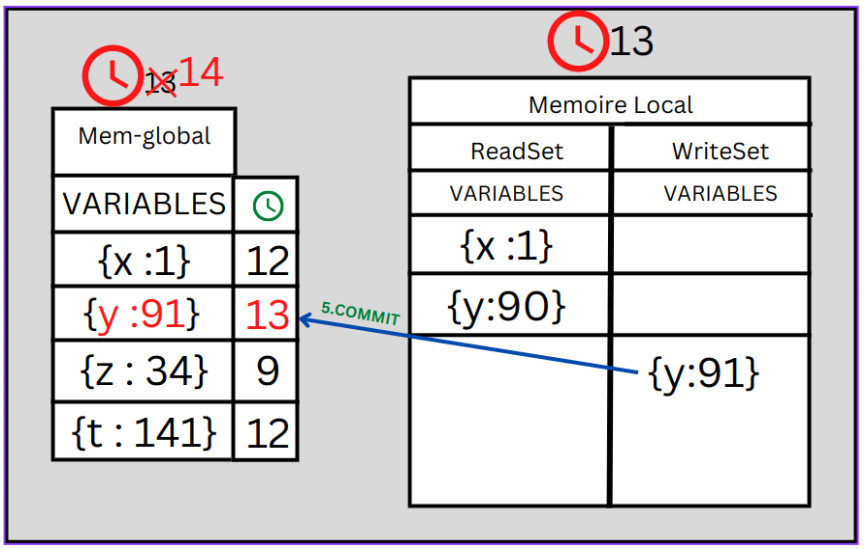
avant de commiter les changement, on vérifie qu'aucune autre transaction n'a changé la valeur les variables en mémoire globale, pour cela chaque variable a un compteur qui sera utilisé pour la vérification, avant d'écrire on vérifie que le compteur des variable dans la mémoire global est inférieur à l'horloge local de la transaction,
Pour chaque écriture on change aussi la valeur de ce compteur avec la valeur de l'horloge local de la transaction.
A la fin du commit on incrémente l'horloge de la mémoire globale.

Example:

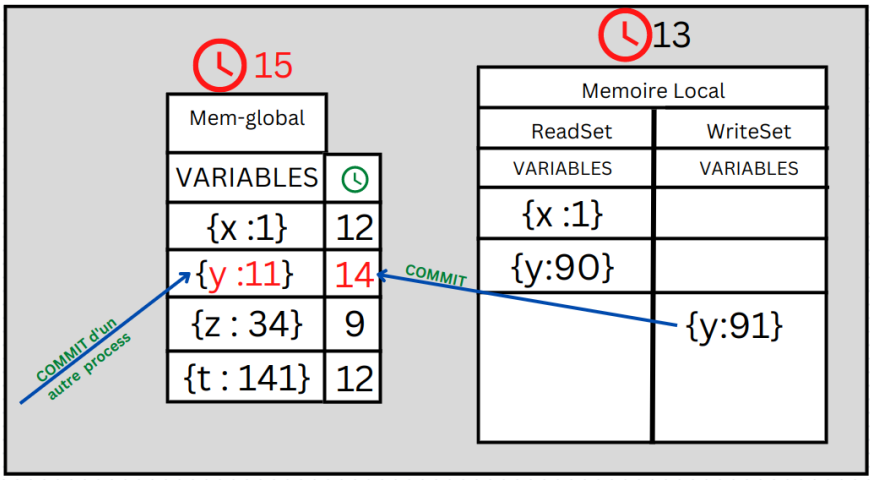
```
Transaction  {//begin (1)
        read x (2)
        read y (3)
        y = y+x (4)
}// commit (5)
```

Cas d'un commit sans problème:

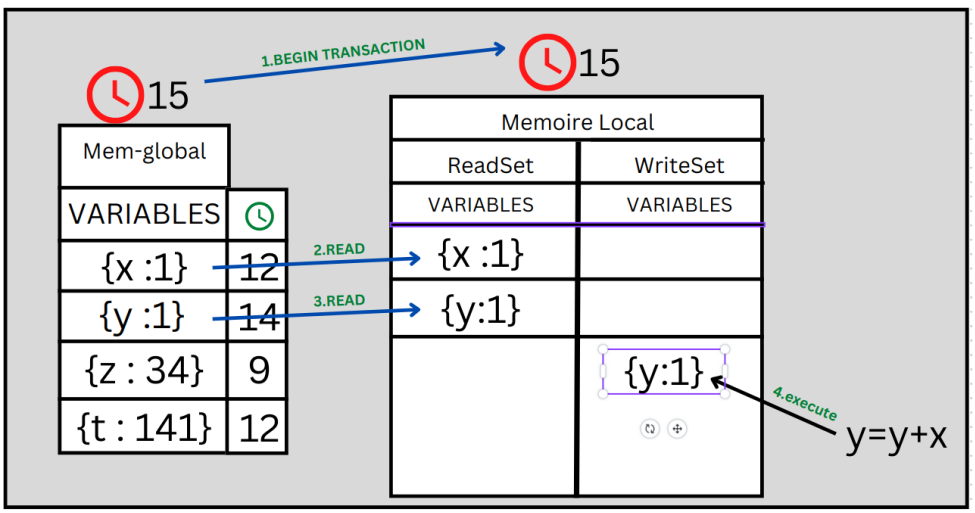




Cas d'un rollback:



ce commit est impossible puisque le compteur de la variable y en memoire global est supérieur à l'horloge local de la transaction donc on roll back et on recommence :



2.implémentation:

Memory:

représente la mémoire global, elle a deux attributs, Values qui est une list de Value et une horloge global qui est un entier

Value:

class qui représente les variables de la mémoire, elle a comme attributs value de type entier et counter de type entier

ValueSet:

représente une collection de value
la classe définit des fonctions pour ajouter des variables a la collection ‘set’, ou les pour la recherche ‘get’, aussi une fonction pour vider la collection de values ‘clean’,
ainsi qu’une fonction qui vérifie que toutes les variable sont prete pour etre merge sur la memoire globale ‘checkWrite’

```
public boolean checkWrite(int clock){
    for(int i=0 ; i<n_elements ; ++i){
        if(Memory.memory.values[elements[i]].counter >= clock){
            return false;
        }
    }
    return true;
}
```

Transaction:

représente la transaction elle a 2 ValueSet qui sont le writeSet et le ReadSet
en plus elle a une horloge locale
la classe définit les fonctions suivante:

begin:

le début de la transaction “copier l’horloge global dans la locale”

```
2 usages
public void begin() { this.clock = Memory.memory.clock; }
```

abort:

dans le cas du rollback cette fonction est appelée pour flush les 2 ValueSet
cette fonction throw une exception pour pouvoir la catch est ré-exécuter la transaction a nouveau

```
2 usages
public void abort() throws TransactionAbort {
    commitAbort++;
    readSet.clear();
    writeSet.clear();
    throw new TransactionAbort();
}
```

read:
permet lire la valeur de la variable dans un index, si la variable est dans le writeSet en retourne la valeur qui est dans le writeSet sinon on cherche la variable dans la mémoire global; on vérifie que son compteur est inférieur à l'horloge local puis on retourne la sa valeur

```
public int read(int idx) throws TransactionAbort {
    if(this.writeSet.containsIdx(idx)) {
        return this.writeSet.map[idx].value;
    }
    Value value = Memory.memory.values[idx];
    if(value.counter > this.clock) abort();
    this.readSet.set(idx, value);
    return value.value;
}
```

Write:
si on a déjà la variable dans le writeSet on change sa valeur, sinon on ajoute une copie avec la nouvelle valeur dans le writeSet dans le même index que la mémoire global cela nous permet après de commiter les changement dans la mémoire globale

```
public void write(int idx, int newVal) throws TransactionAbort {
    if(this.writeSet.containsIdx(idx)) {
        this.writeSet.map[idx].value = newVal;
        return;
    }
    Value value = new Value(newVal);
    this.writeSet.set(idx, value);
}
```

Commit:
dans cette fonction on verifie que toutes les variable dans le writeSet "qui sont les variables qu'on a change" peuvent etre merger sur la memoire global, pour cela on verifie que leur compteur en memoire global est inferieur a la clock local de la transaction,
apres ecriture dans la memoire global on incremente l'horloge global et on clean le read et le write set

```
public void commit() throws TransactionAbort {
    synchronized(Memory.memory) {
        if(!writeSet.checkWrite(clock)) {
            abort();
        }
        for(int i=0 ; i< writeSet.n_elements ; ++i){
            Memory.memory.values[writeSet.elements[i]].value = writeSet.map[writeSet.elements[i]].value;
            Memory.memory.values[writeSet.elements[i]].counter = Memory.memory.clock;
        }
        writeSet.clear();
        readSet.clear();
        commitSuccess++;
        Memory.memory.clock++;
    }
}
```

j ai pris le choix d'implémenter le checkWrite comme une fonction qui retourne un true si tout est bien et que toutes les variable sont ok pour le merge sur la mémoire global, et false si c'est pas le cas, et sur la fonction commit je boucle sur ces variable de la writeSet et je

commit les changement en mémoire globale des valeures ainsi que les compteurs des variable , après je clean le writeSet et le ReadSet, et l'incrément l'horloge global
l'incrémentation de l'horloge global va causer le rollback de toutes les autres transaction parallèles qui utilisent l'une des variables que cette transaction a modifié

3. exemple d'usage

Exemple proposé
j'ai proposé un exemple d'usage pour générer un tableau de taille N qui a la forme suivante [1,2,3,4,.....N-2,N-1,N,N-1,N-2.....4,3,2,1]

pour cela j'ai ajouté un attribut a' la class MAIN qui est le idProcess:

```
class Main extends Thread {
    2 usages
    private int idProcess = 0;
```

qui va être utilisé pour définir la partie du tableau que ce processus va modifier.

```
public void test1(int idProcess) {
    int n = 0;
    final int chunk = Memory.memory.values.length/2;
    try {
        Transaction transaction = Transaction.Transaction.get();

        transaction.begin();
        for(int i= idProcess ; i< Memory.memory.values.length-idProcess; ++i){
            int val = transaction.read(i);
            transaction.write(i, newVal: val + 1);
        }

        transaction.commit();

    }catch(TransactionAbort abort) {
        System.out.println("abooort");
        n = delay(n);
        test1(idProcess);
    }
}
```

un exemple avec N=12

```
abooort
abooort
1 - 2 - 3 - 4 - 5 - 6 - 6 - 5 - 4 - 3 - 2 - 1 -

--> (6, 2)
```

pour N=1024
--> (512, 1708) 512:success et 1708 failure
on remarque que sur cette exemple y'a toujours bcp de roll back ce qui est normal car les processus utilise plus qu'une variable en commun en plus le temps d'une transaction et plus lent

exemple de base :

```
--> 1000000 (1000000, 124)
```

124 failure et 1M de success qui est le nombre de transaction
même si toute les transactions utilisent la même variable de la mémoire global on a que 124 rollback
et cela est dû au temps d'exécution des transactions qui est trop court et en plus de la fonction delay, retarder la ré-exécution de la transaction