

HEIG - PCO Laboratoire 6 - Producteur et Consommateur

Auteurs : Arthur Bécaud et Bruno Egremy

Étape 1 - *requestComputation* et *getWork*

Objectif

Cette étape consiste à gérer les producteurs et consommateurs de *Request*.

Strucure de données des *requests*

Pour commencer, il nous a fallu définir notre structure de données qui servira à stocker les *requests* parmi les trois types de *computation* en offrant la fonctionnalité d'une FIFO avec la possibilité de supprimer n'importe quel élément de la structure en tout temps (prévision pour les *aborts*).

Nous avons décidé d'utiliser un *vector* contenant des *deque* de *requests* (*std::vector<std::deque<Request>>*) pour chaque type de *computation*. Ainsi nous aurons une *FIFO* pour chaque type. La *deque* à l'index 0 de *vector* sera pour les *computations* de type *A*, l'index 1 pour les *B* et ainsi de suite.

Section critique et *monitoring*

Étant donné que notre structure de données sera utilisée par plusieurs *threads*, il est nécessaire de la protéger dans un *monitorIn()* et *monitorOut()*.

De plus, il existe deux conditions sur l'utilisation de la structure de données des *requests* :

Premièrement le nombre de *requests* par type est limité à 10 par défaut. Afin de gérer cette situation. Il faut pouvoir faire attendre le client s'il ajoute un calcul dans une *FIFO* pleine. Nous avons utilisé un *vector* de *PcoHoareMonitor::Condition* qui permettra de gérer le nombre de *requests* pour chacune des *FIFO*. Comme pour les *deques*, l'index 0 est pour les *A*, etc.

En deuxième il faut pouvoir mettre un calculateur en attente s'il demande du travail alors qu'aucune *request* d'un certain type n'est disponible. Nous avons donc utilisé un 2ème *vector* de *PcoHoareMonitor::Condition* pour distinguer les types de *computations* comme pour la première condition.

Implémentation

Attributs et déclarations :

```
std::vector<std::deque<Request>> requests;
std::vector<Condition> waitRequestType;
std::vector<Condition> waitQueuesFreeSpace;
size_t idCnt;
```

Nous retrouvons ici la structure de données des *requests*, les deux *vectors* de conditions et un compteur d'id pour assigner des identifiants uniques à chaque *computation*.

```
/*
 * @brief The ComputationType enum represents the abstract computation types that are available
 */
enum class ComputationType {A, B, C, NB_TYPE};
```

Nous avons ajouté un élément `NB_TYPE` en fin de l'`enum` pour indiquer le nombre de types. Ceci nous permet d'éviter des nombres magiques dans le code.

Constructeur :

```
ComputationManager::ComputationManager(int maxQueueSize):
    MAX_TOLERATED_QUEUE_SIZE(maxQueueSize),
    requests((size_t) ComputationType::NB_TYPE),
    waitRequestType((size_t) ComputationType::NB_TYPE),
    waitQueuesFreeSpace((size_t) ComputationType::NB_TYPE),
    idCnt(0) { }
```

Le constructeur initialise la constante du nombre maximum de `requests` dans chaque `FIFO` et les trois `vectors` avec le nombre de type dans l'`enum ComputationType`.

requestComputation :

```
int ComputationManager::requestComputation(Computation c) {
    monitorIn();

    // Verify if the appropriate queue has less than MAX_TOLERATED_QUEUE_SIZE requests.
    if (requests.at((int) c.computationType).size() == MAX_TOLERATED_QUEUE_SIZE) {
        wait(waitQueuesFreeSpace.at((size_t) c.computationType));
    }

    // Add a request to the appropriate queue.
    requests.at((int) c.computationType).push_back(Request(c, (int) idCnt));

    // Make a signal for the new request
    signal(waitRequestType.at((size_t) c.computationType));

    monitorOut();

    return (int) idCnt++;
}
```

La fonction monitorée commence par vérifier une fois si la `FIFO` du bon type de `computation` est pleine. Si c'est le cas, l'appel devient bloquant jusqu'à ce que la `FIFO` soit libérée d'une `request` via un appel à `getWork` par un calculateur du type correspondant.

Après la vérification ou l'attente, nous créons une nouvelle `request` à partir du `computation` fourni et l'ajoutons dans la bonne `FIFO`.

Il faut ensuite signaler qu'une nouvelle `request` est disponible pour possiblement débloquer un calculateur en attente de travail.

La fonction termine par retourner l'id de la `request` puis incrémenter le compteur d'id.

getWork :

```
Request ComputationManager::getWork(ComputationType computationType) {
    monitorIn();

    // Verify if there is a request in the appropriate queue.
    if (requests.at((size_t) computationType).empty()) {
        wait(waitRequestType.at((size_t) computationType));
    }
```

```

// Retrieve and remove the request from the appropriate queue.
Request request = requests.at((size_t) computationType).front();
requests.at((size_t) computationType).pop_front();

// Signal that a request was retrieved
signal(waitQueuesFreeSpace.at((size_t) computationType));

monitorOut();

return request;
}

```

La fonction monitorée vérifie premièrement si une *request* est disponible dans le type de *computation* demandé. Si aucune n'est disponible, l'appel devient bloquant jusqu'à l'arrivée d'une *request* du bon type.

Après la vérification ou l'attente, il faut récupérer et supprimer la *request* de sa *FIFO*.

Puis signaler que la *FIFO* de la *request* possède à présent une place pour une nouvelle *request* avant de retourner la *request* récupérée.

Étape 2 - *getNextResult* et *provideResult*

Objectif

Cette étape consiste à gérer les producteurs et consommateurs de *Result*.

Strucure de données des *results*

Afin de stocker les *results* des calculateurs, il nous faut choisir une structure de données qui nous permettra de garder l'ordre d'arrivée des *computations* avec la possibilité de supprimer une *computation* en tout temps sans impacter l'intégrité de la structure (prévision pour les *aborts*).

Pour remplir ces conditions, nous avons choisi d'utiliser une *list* de *results* (*std::list*) et une *map* de paire id, itérateur de *list* (*std::map::iterator*).

La *list* permet de stocker les *results* au fil de leur arrivée.

La *map* sert à lister tous les *results* attendus, grâce à des itérateurs de *list*, dans l'ordre pour le client.

Section critique et *monitoring*

Étant donné que notre structure de données sera utilisée par plusieurs *threads*, il est nécessaire de la protéger dans un *monitorIn()* et *monitorOut()*.

De plus il faut ajouter une condition au client pour vérifier si un *result* existe lorsqu'il essaye d'en récupérer un.

Implémentation

Attributs et déclarations :

```

typedef size_t id;
typedef std::list<Result>::iterator listIndex;

std::list<Result> results;
std::map<id, listIndex> resultsIndex;
Condition waitResult;

```

Nous retrouvons ici la structure de données des *results*, avec une utilisation de *typedef* pour simplifier la paire de la *map*, et la condition lors des appels du client.

requestComputation :

```
int ComputationManager::requestComputation(Computation c) {  
  
    monitorIn();  
  
    // Verify if the appropriate queue has less than MAX_TOLERATED_QUEUE_SIZE requests.  
    if (requests.at((int) c.computationType).size() == MAX_TOLERATED_QUEUE_SIZE) {  
        wait(waitQueuesFreeSpace.at((size_t) c.computationType));  
    }  
  
    // Add a request to the appropriate queue.  
    requests.at((int) c.computationType).push_back(Request(c, (int) idCnt));  
  
    // Add a placeholder result. // NEW!  
    resultsIndex.insert(std::pair<id, listIndex>(idCnt, results.end())); // NEW!  
  
    // Make a signal for the new request  
    signal(waitRequestType.at((size_t) c.computationType));  
  
    monitorOut();  
  
    return (int) idCnt++;  
}
```

Nous avons modifié cette fonction en ajoutant la réservation d'un espace dans la *map* avec l'id du *computation* et un itérateur sur la fin de la *list* des *results*. La *map* utilisera les identifiants pour trier les paires, ce qui permet d'obtenir l'ordre des *results* pour le client.

getNextResult :

```
Result ComputationManager::getNextResult() {  
  
    monitorIn();  
  
    // Verify if the next result is available.  
    while (resultsIndex.empty() || resultsIndex.begin()->second == results.end()) {  
        wait(waitResult);  
    }  
  
    // Retrieve the result from the list.  
    Result result = * (resultsIndex.begin()->second);  
  
    // Remove the result from the list and the result entry from the map.  
    results.erase(resultsIndex.begin()->second);  
    resultsIndex.erase(resultsIndex.begin());  
  
    monitorOut();  
  
    return result;  
}
```

La fonction monitorée débute en vérifiant en boucle si la *map* est vide ou si l'itérateur du premier élément de la *map* itère sur la fin de la *list*. Si c'est le cas, l'appel devient bloquant jusqu'à ce qu'un nouveau *result* soit fourni. Il est possible que le nouveau *result* ne soit pas celui attendu, c'est pour ça qu'il faut effectuer la vérification en boucle jusqu'à l'arrivée du bon *result*.

Après la vérification, nous récupérons le *result* grâce à l'itérateur dans la paire. Puis nous supprimons le *result* de la *list* et la paire de la *map* avant de retourner le *result* à l'appelant.

provideResult :

```

void ComputationManager::provideResult(Result result) {
    monitorIn();

    // Add the result to the vector.
    results.push_back(result);

    // Add the iterator to the result in the list 'results'.
    resultsIndex[result.getId()] = --(results.end());

    // Signal that a new result is available.
    signal(waitResult);

    monitorOut();
}

```

La fonction monitorée ajoute le nouveau *result* dans la *list*. Puis donne l'itérateur du nouveau *result* à la pair avec l'id correspondant dans la *map*. Après nous signalons qu'un nouveau *result* est disponible pour possiblement débloquer le *thread* du client lors d'un appel à *getNextResult*.

Étape 3 - *abortComputation* et *continueWork*

Objectif

Cette étape ajoute la fonctionnalité d'annulation des *computation* que le client a demandé.

Section critique et *monitoring*

Les deux fonctions vont devoir manipuler les structures de données. Il est ainsi nécessaire d'utiliser *monitorIn()* et *monitorOut()* dans chacune des fonctions.

Implémentation

abortComputation :

```

void ComputationManager::abortComputation(int id) {
    monitorIn();

    auto iterator = resultsIndex.find(id);

    // Verify if the id exist.
    if (iterator == resultsIndex.end()) {
        monitorOut();
        return;
    }

    // Verify if a result already exist.
    if (iterator->second != results.end()) {
        results.erase(iterator->second);
    } else {
        // If not,
        // try to find the request using the given id.
        // If the request is found, removes it
        // from the requests deque and from the 'resultsIndex' map.
        for (size_t i = 0; i < (size_t) ComputationType::NB_TYPE; i++) {
            for (size_t j = 0; j < requests.at(i).size(); j++) {
                if (requests.at(i).at(j).getId() == id) {
                    requests.at(i).erase(requests.at(i).begin() + j);

                    // Signal that a request was deleted.
                    // There is now room for a new one.
                    signal(waitQueuesFreeSpace.at(i));
                }
            }
        }
    }
}

```

```

        monitorOut();
        return;
    }
}

// Remove the resultIndex in all cases.
resultsIndex.erase(id);

monitorOut();
}

```

La fonction monitorée commence par chercher l'id donné dans la *map* pour stocker l'itérateur de la paire dans une variable locale. Ceci pour éviter de chercher cet itérateur à plusieurs reprise dans la *map*.

Ensuite nous vérifions si nous avons trouvé l'id dans la *map* grâce à la valeur de l'itérateur pour prendre en compte le cas où le client nous donnerait un id inexistant. Si l'id n'existe pas, nous fermons le moniteur et quittons la fonction.

Après ça, nous devons voir si un résultat existe déjà dans la *list*. Pour ça il nous suffit de regarder où itère l'itérateur dans la paire. S'il n'itère pas à la fin de la liste cela veut dire qui faut supprimer un *result* de la *list*. Dans le cas contraire où il itère à la fin de la liste, cela indique que la *request* n'a pas encore été consommée. Il faut ainsi la chercher dans la liste, la supprimer et ensuite signaler qu'un nouvel emplacement est disponible dans la *FIFO* correspondante.

Pour finir, il faut dans tous les cas supprimer la paire de la *map*.

continueWork :

```

bool ComputationManager::continueWork(int id) {
    monitorIn();

    // Try to found resultIndex is found with the given id.
    bool canContinueToWork = resultsIndex.find(id) != resultsIndex.end();

    monitorOut();

    return canContinueToWork;
}

```

La fonction monitorée va simplement vérifier si l'id donné existe dans la *map* puis retourner ce résultat. S'il n'existe pas, cela indique que l'id est invalide ou que la *computation* a été avortée.

provideResult :

```

void ComputationManager::provideResult(Result result) {

    monitorIn();

    // If the computation was aborted, there is no need to provide any result. //NEW!
    if (resultsIndex.find(result.getId()) == resultsIndex.end()) { //NEW!
        monitorOut(); //NEW!
        return; //NEW!
    } //NEW!

    // Add the result to the vector.
    results.push_back(result);

    // Add the iterator to the result in the list 'results'.
    resultsIndex[result.getId()] = --(results.end());

    // Signal that a new result is available.
    signal(waitResult);
}

```

```
    monitorOut();
}
```

Nous avons modifié cette fonction en ajoutant une vérification de l'existance du *result* en paramètre dans la *map*. Si il existe, Nous fermons le moniteur et quittons la fonction. Nous devons ajouter cette fonctionnalité, car il est possible qu'un calculateur essaye de donner un *result* avorté.

Étape 4 - stop

Objectif

Mettre en place la gestion de la terminaison du buffer.

Implémentation

Attributs et déclarations :

```
std::vector<size_t> nbComputerWaiting;
```

Nous avons dû ajouter des compteurs pour les calculateurs en attente de travail lors des appels à la fonction *getWork*. Sans ça, nous ne pouvons pas signaler tous les calculateurs en attente lors d'un appel à *stop*.

stop :

```
void ComputationManager::stop() {
    monitorIn();
    stopped = true;

    for (size_t i = 0; i < (size_t) ComputationType::NB_TYPE; i++) {
        auto nbToSignal = nbComputerWaiting.at(i);
        for (size_t j = 0; j < nbToSignal; j++) {
            signal(waitRequestType.at(i));
        }
        signal(waitQueuesFreeSpace.at(i));
    }
    signal(waitResult);

    monitorOut();
}
```

La fonction monitorée va simplement changer la valeur de la variable *stopped* à *true* puis signaler tous les threads possiblement en attente. Voici la liste des appels bloquant à signaler :

- Il faut signaler tous les calculateurs en attente dans *getWork*. Nous utilisons le *vector* *nbComputerWaiting* pour compter les threads en attentes à cette endroit.
- Le client peut possiblement être en attente d'un emplacement libre dans l'une des *FIFO* après un appel à *requestComputation*. Nous devons signaler qu'un nouvel emplacement est libre dans chaque *FIFO*.
- Le client peut aussi être en attente d'un résultat dans *getNextResult*.

leaveMonitorIfStopped :

```
void ComputationManager::leaveMonitorIfStopped() {
    // Use 'throwStopException' if the 'stop()' function was called.
    if (stopped == true) {
        monitorOut();
```

```

        throwStopException();
    }
}

```

Nous avons ajouté cette fonction *leaveMonitorIfStopped* qui permet de quitter une fonction monitorée via un appel à *throwStopException* après un appel à *stop*. Nous utilisons cette fonction avant et après chaque appel à *PcoHoareMonitor::wait(...)*

getWork :

```

Request ComputationManager::getWork(ComputationType computationType) {

    monitorIn();

    // Verify if there is a request in the appropriate queue.
    if (requests.at((size_t) computationType).empty()) {

        leaveMonitorIfStopped(); // NEW!

        ++nbComputerWaiting.at((size_t) computationType); // NEW!
        wait(waitRequestType.at((size_t) computationType));
        --nbComputerWaiting.at((size_t) computationType); // NEW!

        leaveMonitorIfStopped(); // NEW!
    }

    // Retrieve and remove the request from the appropriate queue.
    Request request = requests.at((size_t) computationType).front();
    requests.at((size_t) computationType).pop_front();

    // Signal that a request was retrieved and removed from the appropriate queue.
    signal(waitQueuesFreeSpace.at((size_t) computationType));

    monitorOut();

    return request;
}

```

Nous avons ajouté les appels à *leaveMonitorIfStopped()* et l'incrémentation/décrémentation des compteurs avant et après l'appel à *wait*. Ceci pour permettre au nombre de threads appelant d'être interrompu proprement.

requestComputation :

```

int ComputationManager::requestComputation(Computation c) {

    monitorIn();

    // Verify if the appropriate queue has less than MAX_TOLERATED_QUEUE_SIZE requests.
    if (requests.at((int) c.computationType).size() == MAX_TOLERATED_QUEUE_SIZE) {

        leaveMonitorIfStopped(); // NEW!

        wait(waitQueuesFreeSpace.at((size_t) c.computationType));

        leaveMonitorIfStopped(); // NEW!
    }

    // Add a request to the appropriate queue.
    requests.at((int) c.computationType).push_back(Request(c, (int) idCnt));

    // Add a placeholder result.
    resultsIndex.insert(std::pair<id, listIndex>(idCnt, results.end()));

    // Make a signal for the new request
    signal(waitRequestType.at((size_t) c.computationType));
}

```

```

    monitorOut();

    return (int) idCnt++;
}

```

Nous avons ajouté un appel à `leaveMonitorIfStopped()` avant et après l'appel à `wait` pour permettre au thread appelant d'être interrompu proprement.

getNextResult :

```

Result ComputationManager::getNextResult() {

    monitorIn();

    // Verify if the next result is available.
    while (resultsIndex.empty() || resultsIndex.begin()->second == results.end()) {

        leaveMonitorIfStopped();

        wait(waitResult);

        leaveMonitorIfStopped();
    }

    // Retrieve the result from the list.
    Result result = * (resultsIndex.begin()->second);

    // Remove the result from the list and the result entry from the map.
    results.erase(resultsIndex.begin()->second);
    resultsIndex.erase(resultsIndex.begin());

    monitorOut();

    return result;
}

```

Nous avons ajouté un appel à `leaveMonitorIfStopped()` avant et après l'appel à `wait` pour permettre au thread appelant d'être interrompu proprement.

continueWork :

```

bool ComputationManager::continueWork(int id) {
    monitorIn();

    // Verify if the 'stop()' function was called. // NEW!
    if (stopped == true) { // NEW!
        monitorOut(); // NEW!
        return false; // NEW!
    } // NEW!

    // Try to found resultIndex is found with the given id.
    bool canContinueToWork = resultsIndex.find(id) != resultsIndex.end();

    monitorOut();

    return canContinueToWork;
}

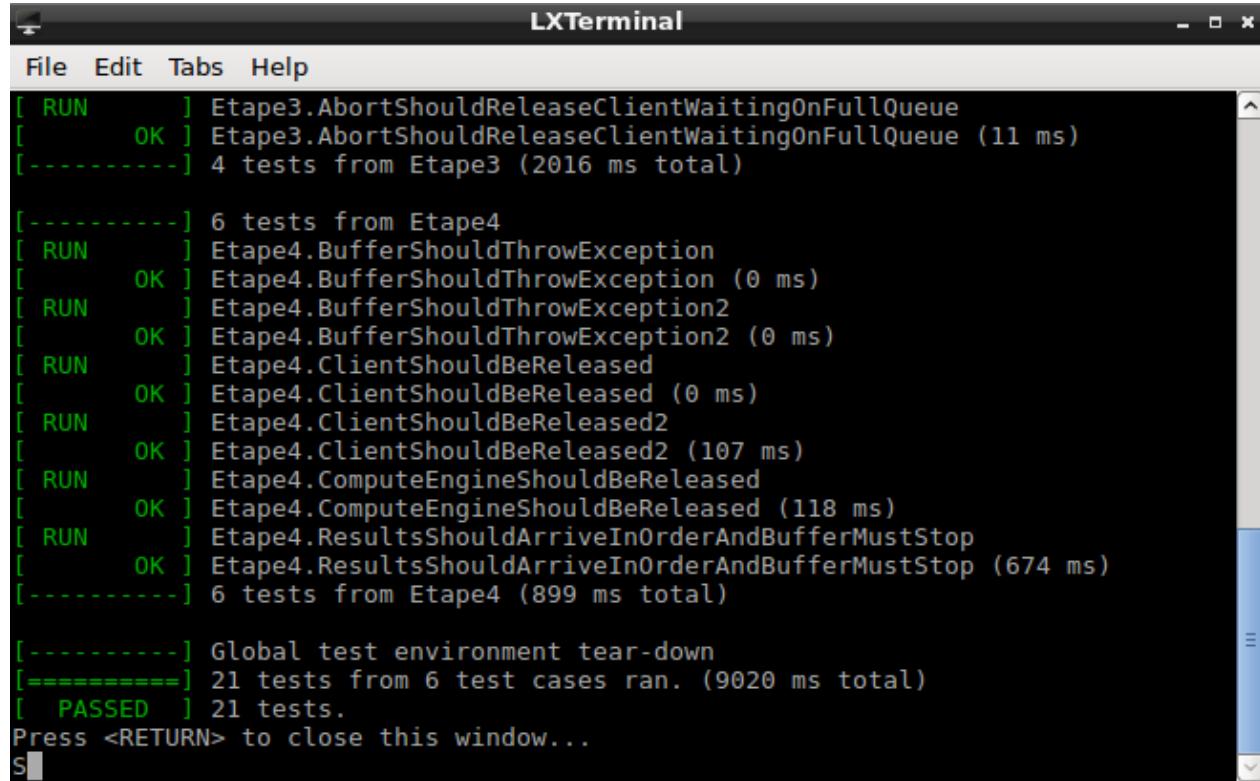
```

La fonction `continueWork` retourne à présent `false` dans tous les cas après un appel à `stop`.

Tests

Afin de vérifier si notre implémentation était fonctionnelle. Nous avons principalement utilisé la configuration *labo6_tests* fournit dans le projet *QT Creator*. Ceci nous a permis de tester l'application à chaque étape.

Nous voyons avec la capture suivante que notre implémentation semble correcte.



The screenshot shows the output of a test run in an LXTerminal window. The terminal title is "LXTerminal". The menu bar includes "File", "Edit", "Tabs", and "Help". The main area displays the following test results:

```
[ RUN      ] Etape3.AbortShouldReleaseClientWaitingOnFullQueue
[      OK  ] Etape3.AbortShouldReleaseClientWaitingOnFullQueue (11 ms)
[-----] 4 tests from Etape3 (2016 ms total)

[-----] 6 tests from Etape4
[ RUN      ] Etape4.BufferShouldThrowException
[      OK  ] Etape4.BufferShouldThrowException (0 ms)
[ RUN      ] Etape4.BufferShouldThrowException2
[      OK  ] Etape4.BufferShouldThrowException2 (0 ms)
[ RUN      ] Etape4.ClientShouldBeReleased
[      OK  ] Etape4.ClientShouldBeReleased (0 ms)
[ RUN      ] Etape4.ClientShouldBeReleased2
[      OK  ] Etape4.ClientShouldBeReleased2 (107 ms)
[ RUN      ] Etape4.ComputeEngineShouldBeReleased
[      OK  ] Etape4.ComputeEngineShouldBeReleased (118 ms)
[ RUN      ] Etape4.ResultsShouldArriveInOrderAndBufferMustStop
[      OK  ] Etape4.ResultsShouldArriveInOrderAndBufferMustStop (674 ms)
[-----] 6 tests from Etape4 (899 ms total)

[-----] Global test environment tear-down
[=====] 21 tests from 6 test cases ran. (9020 ms total)
[ PASSED ] 21 tests.
Press <RETURN> to close this window...
S|
```

À côté de ça, nous avons aussi testé notre implémentation via l'interface graphique.