TD 2 - ASR7 Programmation Concurrente

—

Thread et concurrence

Matthieu Moy, Fabien Rico, Adil Khalfa

Printemps 2018

I Retour sur le TD1

— Exercices non-faits pendant le TD1

II Pourquoi faire un programme multithread

On souhaite comparer l'efficacité d'un serveur de fichiers en mode mono ou multithread, même sur un ordinateur disposant d'un seul processeur monocoeur.

L'intérêt du multithread se trouve uniquement lors des accès disque car le thread qui demande l'accès à un fichier doit attendre pendant que les données sont lues sur le disque. Le serveur de fichiers dispose d'un cache en mémoire pour les fichiers les plus couramment lus. On suppose :

— la durée pour traiter une requête sans accès disque est de 15ms (récupérer la requête, chercher dans le cache, rendre le résultat) ;
— pour gérer le multithreading un surcoût de 5ms est nécessaire (changement de contexte et passage en mode noyau pour passer d'un thread à l'autre) ;
— si le fichier ne se trouve pas en cache il faut 75ms supplémentaires pour la lecture sur le disque ;
— en moyenne un fichier est disponible dans le cache dans 2/3 des cas.

Q.II.1) - Donner le nombre de requêtes traitées par seconde pour un serveur monothread

Pour un serveur monothread, dans 2/3 des cas le fichier est dans le cache et demande 15ms de traitement ; dans 1/3 des cas le fichier n'est pas en cache, il faut donc 75ms+15ms pour traiter la requête. En moyenne, il faut donc pour traiter une requête : 2/3*15 + 1/3*90 = 40ms. Le nombre de requêtes traitées par secondes est donc 1000/40 = 25.

Q.II.2) - Donner le nombre de requêtes traitées par seconde pour un serveur multithread utilisant des threads noyaux.

Dans le cas du multithreading, si on suppose que les requêtes sont bien réparties, on permet au processeur et au disque de travailler en parallèle donc on doit calculer séparément ce dont sont capable le processeur et le disque :

— Le processeur :
Dans 2/3 des cas le fichier est en cache, et la requête demande 15+5 = 20ms. Dans 1/3 des cas, le fichier n'est pas dans le cache, mais il est chargé en parallèle des traitements d'autres requêtes : dans le 1/3 des cas restants, il faut donc également 20ms pour traiter ces requêtes. En moyenne, cela donne donc une requête traitée toutes les 20ms, soit 50 requêtes traitée par seconde.
— Le disque :
Dans 1/3 des cas le temps nécessaire au disque est 75ms dans les autres cas c'est 0. Donc en moyenne c'est 25ms. donc le disque est capable de traiter 40 requêtes par seconde.

Au final, le plus lent impose son rythme donc le serveur peut traiter 40 requêtes par seconde.

Q.II.3) - Est-il intéressant d'utiliser des threads utilisateurs (green threads) ?

Faire rappel sur les threads utilisateurs, perf noyau et modèle programmation (N-M).
Les threads utilisateurs sont plus rapidement gérés (pas de surcoût pour la gestion des threads, ou très peu, car tout se passe en mode utilisateur sans changement de contexte). On peut donc enlever les 5ms. Par contre, le noyau ne voit pas le fait que le processus est multithread donc l'accès disque est bloquant. On ne gagne pas les 75ms d'accès disque. Donc le temps est identique à celui du serveur monothread, mais le programme est plus compliqué car c'est le programmeur qui doit gérer la file d'attente des connexions (via les threads) au lieu de laisser faire le système (via l'API des sockets).

III Le pont de Miralonde

Le pont de Miralonde est trop étroit pour que 2 voitures puissent se croiser. Vous devez mettre en place un système qui évitera tout incident. Le système se déclenchera automatiquement à l'arrivée d'une voiture à l'une des extrémités du pont. Il autorisera ou non le passage en fonction de la configuration sachant que :

— Si le pont est vide, la première voiture qui arrive peut passer.
— Si le pont contient une voiture qui va du nord au sud, seules les voitures circulant dans le même sens (donc arrivant au nord) peuvent passer.
— Inversement, si le pont contient une voiture qui va du sud au nord, seules les voitures arrivant au sud ont l'autorisation de passer.

Pour éviter tout problème, votre système dispose de barrières contrôlées par un ordinateur. Vous devez écrire le programme de contrôle qui tourne sur cet ordinateur en utilisant les outils classiques (mutex, variables de conditions, ...). À chaque fois qu'une voiture arrive à l'extrémité nord du pont, le système appelle automatiquement la fonction `EntreeNS()` puis lorsque cette voiture sort du pont, la fonction `SortieNS()` est appelée. Inversement, les fonctions `EntreeSN()` et `SortieSN()` servent à gérer les voitures qui circulent dans l'autre sens. Toutes ces fonctions peuvent être appelées en concurrence.

Nous supposerons que si la fonction `EntreeNS()` (ou `EntreeSN()`) se termine, le véhicule est autorisé à passer, alors que si elle bloque, le véhicule est aussi bloqué (par exemple, on peut imaginer un système qui détecte l'arrivée d'une voiture et appelle `EntreeNS()` quand la voiture arrive, lève la barrière d'entrée quand la fonction termine, et la redescend immédiatement après le passage de la voiture).

Q.III.1) - Donnez un algorithme des fonctions `EntreeNS()`, `EntreeSN()`, `SortieNS()` et `SortieSN()` en utilisant le principe du moniteur de Hoare.

```
1   /**
2    * Simple version, but one direction may starve if cars come often
3    * enough in the other direction.
4    */
5   class Miralonde_Starve {
6   public:
7           void EntreeNS() {
8                   m.lock();
9                   cout << "EntreeNS: request" << endl;
10                  while (nb_cars_SN != 0) {
11                          c.wait(m);
12                  }
13                  nb_cars_NS++;
14                  cout << "EntreeNS: pass" << endl;
15                  m.unlock();
16          }
17          void EntreeSN() {
18                  m.lock();
19                  cout << "                       EntreeSN: request" << endl;
20                  while (nb_cars_NS != 0) {
21                          c.wait(m);
22                  }
23                  nb_cars_SN++;
24                  cout << "                       EntreeSN: pass" << endl;
25                  m.unlock();
26          }
27          void SortieNS() {
28                  m.lock();
29                  cout << "SortieNS" << endl;
30                  nb_cars_NS--;
31                  c.notify_all();
32                  m.unlock();
33          }
34          void SortieSN() {
35                  m.lock();
36                  cout << "                       SortieSN" << endl;
37                  nb_cars_SN--;
38                  c.notify_all();
39                  m.unlock();
40          }
41  private:
42          int nb_cars_NS = 0;
43          int nb_cars_SN = 0;
44          mutex m;
45          condition_variable_any c;
46  };
```

Q.III.2) - Montrez la propriété de sûreté : il n'y a jamais à la fois une voiture venant du nord et une venant du sud sur le pont.

Q.III.3) - Montrez que cet algorithme n'a pas de problème de *deadlock* (inter-blocage).

Q.III.4) - L'algorithme pose-t-il un problème de famine ? Si oui, donnez un exemple puis proposez un nouvel algorithme.

La propriété de sûreté est garantie par la boucle `while` : dans `EntreeNS()`, on n'autorise le passage qu'en sortant de la boucle `while`, donc quand `nb_cars_SN == 0`, et réciproquement pour `EntreeSN()`. Donc une voiture ne peut entrer que quand il n'y a pas de voiture dans l'autre sens.

L'absence de deadlock est assez facile à montrer :

— Chaque `m.lock()` correspond à une portion de code dont le temps d'exécution est borné : on atteint soit un `c.wait(m)` soit un `m.unlock()` sans opération bloquante ni boucle non-bornée.

— Quand une voiture appelle `c.wait()`, c'est qu'il y a des voitures dans l'autre sens, donc ces voitures sortiront forcément du pont à un moment et appelleront `c.notify_all()`.

Par contre, l'algorithme pose un problème de famine. Une famine est une attente pendant une durée non-bornée, donc un exemple de famine est une exécution infinie pendant laquelle un processus n'est pas servi (on ne peut pas montrer une famine sur une exécution finie).

On considère l'exécution suivante :

— La première voiture qui arrive vient du sud, elle rentre sur le pont.

— Pour chaque voiture venant du sud, avant que cette voiture ne sorte du pont, une autre voiture arrive du sud et rentre également sur le pont (elle peut le faire vu qu'il y a une voiture dans le sens sud-nord donc aucune dans le sens nord-sud).

Une voiture arrivant du nord face à cette série de voiture sera bloquée indéfiniment.

Une solution est d'interdire à une voiture de s'engager sur le pont quand une voiture est en attente dans l'autre sens. Mais il faut alors faire attention aux deadlocks ! Par exemple :

— Voiture 1 : `EntreeNS()` (passe)

— Voiture 2 : `EntreeSN()` (attend, car il y a quelqu'un engagé dans le sens opposé)

— Voiture 3 : `EntreeNS()` (attend, car il y a une voiture en attente de l'autre côté)

— Voiture 1 : `SortieNS()`

Avec un algorithme naïf, les voitures 2 et 3 seraient bloquées l'une par l'autre (deadlock). On peut résoudre ce problème en dé-symétrisant la situation : lorsque le pont est libre, une voiture A n'attendra que si une voiture attend B de l'autre côté *et* que la dernière voiture engagée sur le pont était dans le même sens que A. Cette version a les propriétés attendues :

— Quand le pont est libre, s'il n'y a pas de voiture en attente d'un côté, alors les voitures arrivant dans l'autre sens peuvent passer (pas d'attente inutile).

— Quand le pont est libre et qu'il y a des voitures en attente des deux côtés, alors l'un des deux côtés est bloqué pendant qu'une voiture dans l'autre sens peut s'engager (pas de deadlock).

— Il ne peut pas y avoir d'attente non-bornée : quand une voiture A est bloquée d'un côté, c'est soit par une voiture B engagée dans l'autre sens, soit par une voiture B en attente dans l'autre sens. Si B est engagée, alors B finira par libérer le pont et par laisser passer A. Si B est en attente, c'est qu'il y a une voiture engagée dans le sens de A (sinon A aurait pu s'engager), et alors B pourra s'engager une fois cette voiture sortie du pont. Une fois B engagée, les autres voitures dans le même sens que B devront attendre (vu que A est en attente de l'autre côté). Quand B libérera le pont, A ne sera plus bloquée car A est dans le sens opposé à celui de B qui est la dernière voiture engagée sur le pont (pas de famine).

Au final, l'algorithme est (le code complet est disponible sur la page du cours) :

```
1  /**
2   * Starvation-free version of Miralonde.
3   *
4   * Cars can go if:
5   * - nobody takes the other direction
6   * - and :
7   *      - nobody's waiting on the other side, or
8   *      - previous car went the oposite way
9   */
10 class Miralonde_Starve_Free {
11 public:
12         void EntreeNS() {
13                 m.lock();
```

```
14                  cout << "EntreeNS: request" << endl;
15                  nb_cars_waiting_N++;
16                  while (not(nb_cars_SN == 0 &&
17                             (nb_cars_waiting_S == 0 ||
18                              current_direction == SN))) {
19                          c.wait(m);
20                  }
21                  nb_cars_waiting_N--;
22                  nb_cars_NS++;
23                  current_direction = NS;
24                  cout << "EntreeNS: pass" << endl;
25                  m.unlock();
26          }
27          void EntreeSN() {
28                  m.lock();
29                  cout << "                    EntreeSN: request" << endl;
30                  nb_cars_waiting_S++;
31                  while (not(nb_cars_NS == 0 &&
32                             (nb_cars_waiting_N == 0 ||
33                              current_direction == NS))) {
34                          c.wait(m);
35                  }
36                  nb_cars_waiting_S--;
37                  nb_cars_SN++;
38                  cout << "                    EntreeSN: pass" << endl;
39                  current_direction = SN;
40                  m.unlock();
41          }
42          void SortieNS() {
43                  m.lock();
44                  cout << "SortieNS" << endl;
45                  nb_cars_NS--;
46                  c.notify_all();
47                  m.unlock();
48          }
49          void SortieSN() {
50                  m.lock();
51                  cout << "                    SortieSN" << endl;
52                  nb_cars_SN--;
53                  c.notify_all();
54                  m.unlock();
55          }
56  private:
57          enum direction {
58                  NS,
59                  SN,
60          };
61          direction current_direction = NS;
62          int nb_cars_waiting_N = 0;
63          int nb_cars_waiting_S = 0;
64          int nb_cars_NS = 0;
65          int nb_cars_SN = 0;
66          mutex m;
67          condition_variable_any c;
68  };
```