TD 2 - ASR7 Programmation Concurrente

—

Thread et concurrence

Matthieu Moy + l'équipe ASR7 (cf. page du cours)

Printemps 2020

I Retour sur le TD1

— Exercices non-faits pendant le TD1

II Algorithme de Dijkstra

Voici une tentative (incorrecte) d'algorithme d'exclusion mutuelle :

```
1   int locked = false;
2
3   void lock() {
4       while (locked) {
5           /* wait */
6       }
7       locked = true;
8   }
9
10  void unlock() {
11      locked = false;
12  }
```

Q.II.1) - Montrer que cet algorithme ne garantie pas l'exclusion mutuelle.

> Deux threads peuvent exécuter le `while`, trouver la variable `locked` à false, puis tous les deux écrire la valeur true et entrer en section critique.

Q.II.2) - Si on vous fournit une fonction `test_and_set()` qui permet atomiquement de lire la valeur d'une variable et d'y écrire une autre valeur, pouvez-vous faire mieux (en pratique les processeurs modernes fournissent en général une instruction permettant de faire ceci) ?

```
1   void lock() {
2       while (test_and_set(&locked, true)) {
3           /* wait */
4       }
5   }
```

L'article donnant le premier algorithme de résolution du mutex à n processus est donné page 9. Il date de 1965. Il suppose n processus sur N processeurs, et s'exécute en **mémoire partagée** : Il suppose que K est accessible en lecture par tous et peut être mis à jour par tous.

Avant d'entrer dans les détails, on peut remarquer que le code est mal indenté et qu'il est difficile de voir comment les begin/end se correspondent. Par ailleurs, une boucle est codée avec un if/then/else + goto. Voici une version un peu plus claire :

```
1   Li0: b[i] := false; // Phase 1
2   Li1: while k != i
3   Li2: begin
4            c[i] := true;
5   Li3:     if b[k] then k := i;
6        end
7   Li4: // Phase 2
8        c[i] := false;
9        for j := 1 step 1 until N
10       begin
11           if j != i and not c[j] then goto Li1
12       end;
13       critical section;
14       c[i] : = true; b[i] : = true;
15       remainder of the cycle in which stopping is allowed;
16       goto Li0;
```

Q.II.3) - Que sont I, K ?

I est le numéro du proc exécutant l'algorithme. K est le numéro du proc voulant entrer en Section Critique.

Q.II.4) - Comment sont initialisés les tableaux B et C ?

B et C doivent être dans le même état après la section critique qu'avant, donc on peut déduire qu'ils sont initialisés à true au départ (c'est écrit dans l'article juste au dessus du code).

Q.II.5) - Montrer que deux processus ne peuvent entrer en section critique en même temps.

- Remarque préalable : On peut noter que 2 proc peuvent être rendus en Li4 car l'un peut avoir exécuté la séquence d'instruction où il affecte I à K puis le test $(k! = i)$ alors qu'un autre a effectué le test $(B[K])$ et n'exécute sa prochaine instruction, c-à-d $K = I$ qu'après que le premier soit rendu à Li4.
  Il faut donc départager les processus qui ont demandé et sont passés à la ligne Li4 ensemble. On voit qu'ils affectent C[I] à false. Dans la boucle suivante, on attend que tous les C[J], dans l'ordre donné par la boucle, soient un à un à true. C'est cette boucle (l'ordre induit !) qui permet d'assurer l'exclusion mutuelle.
- En fait, la phase 2 est suffisante pour assurer l'exclusion mutuelle (premier paragraphe de la preuve dans le papier). On procède par un raisonnement par l'absurde :
  Supposons qu'il y ait 2 proc a et b $(a \neq b)$ en SC, alors lors du passage dans la boucle on a :
  - pour x, tous les c[x] valent true pour $x \neq a$, en particulier c[b] = true
  - pour x, tous les c[x] valent true pour $x \neq b$, en particulier c[a] = true

  Pourtant si a arrive à Li4, il commence par affecter false à C[a], donc le processus b a forcément testé C[a] avant l'affectation C[a] = false, donc le processus b est entré en section critique avant a. Le même raisonnement permet de montrer que a est entré avant b : on obtient une incohérence qui conclue la preuve.
  Donc 2 procs ne peuvent être en SC au même instant.

Q.II.6) - Montrez qu'un processus peut entrer en section critique lorsque c'est possible, c-à-d lorsqu'elle est libre.

Si personne ne demande à entrer en SC, alors tous les C[x] sont à true lorsqu'un processus fait sa demande. Le processus l'obtient donc sans attendre.
Pour être rigoureux il faudrait aussi montrer que des processus ne peuvent pas boucler indéfiniment (situation « après vous/après vous » mentionnée dans l'article). On ne le fera pas dans les détails, mais les arguments essentiels sont 1) un processus réussira toujours à affecter k := i, et 2) une fois que k est affecté à i, les processus qui entrent dans la boucle ne peuvent pas modifier k jusqu'à ce que k ait terminé sa section critique. k finira toujours par entrer en SC car les autres finiront par affecter leur c[i] à true.

Q.II.7) - Que vient-on de montrer avec ces 2 propriétés ?

La vivacité et la sûreté, donc que l'algo permet d'assurer une exclusion mutuelle :

— Une propriété de *sûreté* (*safety*) est une propriété du type « une erreur ne se produit jamais » (autrement dit, les états d'erreurs ne sont pas atteignables). Ici, une erreur serait d'avoir les deux processus en même temps en section critique. Mathématiquement, une propriété de sûreté est du type $\forall ..., \neg \text{error}$.

— Une propriété de *vivacité* (*liveness*) est une propriété du type « quelque chose arrivera forcémement en temps fini ». Mathématiquement, c'est une propriété du type $\exists ..., \text{propriete}$. Ici, la propriété qui nous intéresse est qu'un processus qui demande la section critique finit par l'obtenir.

On peut remarquer qu'il est en fait très facile de garantir la sûreté :

```
void lock() {
    while (true) {}; // Boucle infinie
}
void unlock() {
    /* Rien */
}
```

Avec cette implémentation du mutex, on n'a jamais deux processus en même temps dans la section critique, car il n'y a jamais personne dans la section critique ! Bien sûr, c'est trivialement incorrect, car ça ne garantie pas la vivacité.

Q.II.8) - Donner un inconvénient à l'algorithme.

L'attente active ! C'est la raison des sémaphores et des solutions sollicitant l'ordonnanceur du système d'exploitation.

Q.II.9) - Pour ceux voulant aller plus loin, vous pouvez lire `http://jakob.engbloms.se/archives/65`. Le titre de l'article, « Dekker's Algorithm Does not Work, as Expected », devrait vous mettre la puce à l'oreil (sachant que l'algorithme de Dijkstra est une généralisation de Dekker) ;-).

On ne rentre pas dans les détails avec les étudiants, mais l'algorithme fait l'hypothèse que les accès aux variables sont « atomiques », au sens où un les écritures fait par un thread seront vues dans l'ordre où elles ont été faites par les autres threads par exemple. Sur une un processeur moderne, ce n'est pas toujours le cas à cause du cache, de ré-ordonnancement

d'instructions, ... d'où la nécessité d'avoir des opérations atomiques dans les langages comme C++11 et C11, mais on ne les verra pas dans ce cours.

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

Faire rappel sur les threads utilisateurs, perf noyau et modèle programmation threads multiplexés (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).

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

Q.IV.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  };
```

Solution of a Problem in Concurrent Programming Control

E. W. Dijkstra
Technological University, Eindhoven, The Netherlands

A number of mainly independent sequential-cyclic processes with restricted means of communication with each other can be made in such a way that at any moment one and only one of them is engaged in the "critical section" of its cycle.

Introduction

Given in this paper is a solution to a problem for which, to the knowledge of the author, has been an open question since at least 1962, irrespective of the solvability. The paper consists of three parts: the problem, the solution, and the proof. Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in computer coupling will appreciate the significance of the fact that this problem indeed can be solved.

The Problem

To begin, consider N computers, each engaged in a process which, for our aims, can be regarded as cyclic. In each of the cycles a so-called "critical section" occurs and the computers have to be programmed in such a way that at any moment only one of these N cyclic processes is in its critical section. In order to effectuate this mutual exclusion of critical-section execution the computers can communicate with each other via a common store. Writing a word into or nondestructively reading a word from this store are undividable operations; i.e., when two or more computers try to communicate (either for reading or for writing) simultaneously with the same common location, these communications will take place one after the other, but in an unknown order.

The solution must satisfy the following requirements.

(a) The solution must be symmetrical between the N computers; as a result we are not allowed to introduce a static priority.

(b) Nothing may be assumed about the relative speeds of the N computers; we may not even assume their speeds to be constant in time.

(c) If any of the computers is stopped well outside its critical section, this is not allowed to lead to potential blocking of the others.

(d) If more than one computer is about to enter its critical section, it must be impossible to devise for them such finite speeds, that the decision to determine which one of them will enter its critical section first is postponed until eternity. In other words, constructions in which "After you"-"After you"-blocking is still possible, although improbable, are not to be regarded as valid solutions.

We beg the challenged reader to stop here for a while and have a try himself, for this seems the only way to get a feeling for the tricky consequences of the fact that each computer can only request one one-way message at a time. And only this will make the reader realize to what extent this problem is far from trivial.

The Solution

The common store consists of:

"**Boolean array** b, $c[1{:}N]$; **integer** k"

The integer k will satisfy $1 \leq k \leq N$, $b[i]$ and $c[i]$ will only be set by the ith computer; they will be inspected by the others. It is assumed that all computers are started well outside their critical sections with all Boolean arrays mentioned set to **true**; the starting value of k is immaterial.

The program for the ith computer ($1 \leq i \leq N$) is:

```
"integer j;
Li0: b[i] := false;
Li1: if k ≠ i then
Li2: begin c[i] := true;
Li3: if b[k] then k := i;
     go to Li1
     end
       else
Li4: begin c[i] := false;
       for j := 1 step 1 until N do
         if j ≠ i and not c[j] then go to Li1
     end;
     critical section;
     c[i] := true; b[i] := true;
     remainder of the cycle in which stopping is allowed;
     go to Li0"
```

The Proof

We start by observing that the solution is safe in the sense that no two computers can be in their critical section simultaneously. For the only way to enter its critical section is the performance of the compound statement $Li4$ without jumping back to $Li1$, i.e., finding all other c's **true** after having set its own c to **false**.

The second part of the proof must show that no infinite "After you"-"After you"-blocking can occur; i.e., when none of the computers is in its critical section, of the computers looping (i.e., jumping back to $Li1$) at least one—and therefore exactly one—will be allowed to enter its critical section in due time.

If the kth computer is not among the looping ones, $b[k]$ will be **true** and the looping ones will all find $k \neq i$. As a result one or more of them will find in $Li3$ the Boolean $b[k]$ **true** and therefore one or more will decide to assign "$k := i$". After the first assignment "$k := i$", $b[k]$ becomes **false** and no new computers can decide again to assign a new value to k. When all decided assignments to k have been performed, k will point to one of the looping computers and will not change its value for the time being, i.e., until $b[k]$ becomes **true**, viz., until the kth computer has completed its critical section. As soon as the value of k does not change any more, the kth computer will wait (via the compound statement $Li4$) until all other c's are **true**, but this situation will certainly arise, if not already present, because all other looping ones are forced to set their c **true**, as they will find $k \neq i$. And this, the author believes, completes the proof.