LIFPCA – Programmation Concurrente et Administration Système

Fascicule de Travaux Dirigés

Les sujets de TDs ont été réalisés au fil des années, par les différents responsables de LIFPCA :
Fabien Ricco, Yves Caniou, Matthieu Moy, Grégoire Pichon, Florence Zara

Table des matières

Compétences

A l'issue des TDs, voici les compétences à acquérir :

Compétences conceptuelles

- ☐ Comprendre les principes de la programmation concurrente.
- ☐ Distinguer les notions de processus, de thread et de tâche.
- ☐ Identifier les problèmes liés à l'accès concurrent aux ressources partagées.
- ☐ Expliquer les conditions d'apparition des conditions de compétition (*race conditions*).
- ☐ Comprendre les risques d'interblocage (*deadlock*), de famine et d'inversion de priorité.
- ☐ Comprendre le rôle de l'ordonnanceur du système d'exploitation dans l'exécution des processus et des threads.
- ☐ Expliquer les critères d'évaluation d'un algorithme d'ordonnancement (temps d'attente, temps de réponse, utilisation du processeur, équité).
- ☐ Expliquer le fonctionnement des principaux algorithmes d'ordonnancement des processus.

Compétences techniques

- ☐ Créer et gérer plusieurs threads dans un programme.
- ☐ Mettre en œuvre des sections critiques.
- ☐ Utiliser un mutex pour protéger une ressource partagée.
- ☐ Initialiser, verrouiller et déverrouiller correctement un mutex.
- ☐ Utiliser des sémaphores binaires et compteurs.
- ☐ Synchroniser plusieurs threads à l'aide de sémaphores.
- ☐ Choisir entre mutex et sémaphore selon le problème à résoudre.
- ☐ Concevoir un protocole de synchronisation garantissant la cohérence des données.

Compétences d'analyse

- ☐ Identifier les sections critiques dans un programme.
- ☐ Analyser les dépendances entre threads.
- ☐ Détecter les situations pouvant conduire à un interblocage.
- ☐ Vérifier la correction d'un algorithme de synchronisation.
- ☐ Évaluer les performances d'une solution concurrente (contention, parallélisme, coût de synchronisation).

TD 1 – Concurrence, mutex, sémaphores

1 Gestion de comptes en banque, et concurrence

Deux agences d'une banque veulent mettre à jour le même compte bancaire. Pour cela, l'agence de Nancy effectue :

```
1. courant = get_account(1867A)
2. nouveau = courant + 1000
3. update_account(1867A, nouveau)
```

et l'agence de Karlsruhe :

```
A. aktuelles = get_account(1867A)
B. neue = aktuelles - 1000
C. update_account(1867A, neue)
```

Q1. En supposant que l'agence de Nancy commence en premier, quel sera le montant à l'issue des transactions ?

2 Producteur consommateur

Le problème du producteur consommateur est un problème classique de synchronisation en programmation multi-thread. Par exemple, le problème du producteur/consommateur présente un ensemble de threads « producteurs » qui dialogue avec un ensemble de threads « consommateurs » qui dialoguent grâce à une file de données partagées. On peut par exemple envisager un thread « Maître » qui reçoit les connexions des clients et qui fournit les sockets de discussions à des threads « Esclaves » qui traitent leurs demandes. Vous avez déjà manipulé une forme de producteur-consommateur entre processus : le « pipe » unix (`cmd1 | cmd2`, où `cmd1` est producteur, et `cmd2` consommateur).

Pour éviter les problèmes d'accès concurrents à la liste de sockets, il faut protéger cette donnée. Le but de l'exercice est de programmer la liste sous la forme d'un moniteur.

Note

Pour les questions suivantes, vous ne donnerez que les algorithmes.

Q1. Quelles fonctions doit implémenter le moniteur ? Quelles sont celles qui doivent être protégées ?

Q2. Donnez la description de la structure de données qui permet de stocker cette file.

Q3. Donnez l'algorithme des fonctions qui permettent d'assurer l'ajout et le retrait d'un élément (l'élément sera un simple `int`). Dans un premier temps on pourra renvoyer une erreur quand on tente d'insérer dans une file pleine ou de retirer un élément d'une file vide.

Q4. Modifiez ces fonctions de manière à assurer l'attente (passive) en cas de file pleine ou vide.

Note

Lors du TP, vous implémenterez ces fonctions avec la bibliothèque `thread C++`. A cette fin, il ne faudra pas oublier les fonctions d'initialisation et de libération de ressources.

3 Gestion d'ordre avec des sémaphores

On considère un système où s'exécutent trois processus (légers ou lourds) P1, P2 et P3 qui ont les caractéristiques suivantes :

— P1 exécute en boucle les tâches A puis B ;

— P2 exécute en boucle les tâches U puis V ;

— P3 exécute en boucle la tâche X.

De plus, les contraintes suivantes doivent être respectées :

— La tâche A de P1 produit un élément nécessaire à la tâche X de P3. Cela signifie qu'une occurrence de X ne peut pas démarrer avant la fin d'une occurrence de A.

— Les tâches B et U ne peuvent s'exécuter en même temps.

On note dA_i et fA_i respectivement le début et la fin de la tâche A. On fait de même pour toutes les tâches. Répondez aux questions suivantes :

Q1. Les ordres d'exécutions suivant sont-ils possibles ? Si non, quelles parties posent problème :

a) $dA_1 fA_1 dX_1 dB_1 dU_1 fX_1 fU_1 fB_1 dA_2 dX_2 dV_1 fV_1 fX_2 fA_2 dU_2 dB_2 fU_2 fB_2 dA_3 fA_3 dB_3 fB_3$

b) $dA_1 fA_1 dX_1 dB_1 fB_1 dA_2 dU_1 fA_2 fX_1 fU_1 dX_2 dV_1 fV_1 fX_2 dU_2 fU_2 dB_2 fB_2 dA_3 fA_3 dB_3 fB_3$

Q2. Donnez le graphe de précédence et d'exclusion mutuelle.

Q3. Gérez le problème entre P1 et P2 avec des sémaphores.

Q4. Gérez le problème entre P1 et P3 avec des sémaphores.

Q5. Peut-on utiliser le ou les mêmes sémaphores pour les questions **Q3.** et **Q4.** ?

TD 2 – Thread et concurrence

Note

Finir les exercices du TD1

1 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 `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).

Q1. Donnez les algorithmes des fonctions `EntreeNS()`, `EntreeSN()`, `SortieNS()` et `SortieSN()` en utilisant le principe du **moniteur de Hoare**.

Q2. 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.

Q3. Montrez que cet algorithme n'a **pas de problème de *deadlock* (inter-blocage)**.

Q4. L'algorithme pose-t-il un **problème de famine ?**
Si oui, donnez un exemple, puis proposez un nouvel algorithme.

2 Intérêt d'un programme multi-threadé

On souhaite comparer l'efficacité d'un serveur de fichiers en mode mono ou multithread, 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.

Voici nos hypothèses :

- la durée pour traiter une **requête sans accès disque est de 15ms** (récupérer la requête, chercher dans le cache, rendre le résultat) ;
- la **gestion du multithreading engendre un surcoût de 5ms** (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**.

Q1. Donner le nombre de requêtes traitées par seconde pour un serveur monothread.

Q2. Donner le nombre de requêtes traitées par seconde, pour un serveur multithread utilisant des threads noyaux.

Q3. Est-il intéressant d'utiliser des threads utilisateurs (*green threads*) ?

TD 3 – Exclusion mutuelle, algorithme de Dijkstra

1 Exclusion mutuelle : premier algorithme

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

```
int locked = false;

void lock() {
    while (locked) {
        /* wait */
    }
    locked = true;
}

void unlock() {
    locked = false;
}
```

Q1. Montrer que cet algorithme ne garantit pas l'exclusion mutuelle, c'est-à-dire qu'il ne garantit pas qu'il n'y ait qu'un seul thread à un instant donné en section critique.

Q2. Supposez que vous avez à disposition une fonction `bool test_and_set(bool* ptr)` permettant atomiquement de lire la valeur à l'adresse ptr, d'y écrire la valeur `true` si `*ptr == false` et de renvoyer l'ancienne valeur de `*ptr` (en pratique les processeurs modernes fournissentune instruction permettant de faire ceci). Pouvez-vous faire mieux ?

2 Algorithme de Dijkstra

L'article donnant le premier algorithme de résolution du mutex à n processus est donné à la fin du TD. Il date de 1965. Il suppose n processus sur N processeurs (numérotés de 1 à N), et il s'exécute en **mémoire partagée**. Il suppose que la variable k est accessible en lecture par tous et peut être mis à jour par tous.

Algorithme initial

Avant d'entrer dans les détails, on peut remarquer que le code de l'article 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 de l'algorithme initial :

```
Li0: b[i] := false; // Phase 1
Li1: while (k != i)
Li2: begin
         c[i] := true;
Li3:     if b[k] then k := i;
     end
Li4: // Phase 2
     c[i] := false;
     for j := 1 step 1 until N
     begin
         if j != i and not c[j] then goto Li1
     end;
     critical section;
     c[i]:= true; b[i]:= true;
     remainder of the cycle in which stopping is allowed;
     goto Li0;
```

Q1. Que représentent les variables i et k ?

Q2. Comment sont initialisés les tableaux b et c ?

Algorithme remanié

Pour clarifier encore plus l'algorithme, nous allons changer les noms des variables.

— Le tableau `b[]` donne un marqueur pour dire qu'on souhaite entrer en Section Critique (SC). `b[i]` est mis à `false` (LiO). Dans la SC il est à `false`, et il est mis à `true` (ligne 14) quand on sort de la SC. Le tableau `b[]` est renommé `wantToBeInSC[]`.
`wantToBeInSC[i]` indique que le processus i souhaite entrer en SC.

— Ligne 11, pour tous les processus j différents de i, on fait le test `not c[j]`, c'est-à-dire qu'on souhaite que `not c[j]` soit vrai, cad que `c[j]` soit faux, cad que j ne soit pas en SC.
Le tableau `c[]` est renommé `inSC[]`. Mais cela ne signifie pas que le processus est déjà en SC, mais plutôt "Je suis candidat final à l'entrée dans la section critique".
Autrement dit, le processus est dans la phase 2 de l'algorithme.

— Les tableaux `b[]` et `wantToBeInSC[]` sont initialisés à **false** au départ. On inverse donc les différents tests / valeurs des booléens employés dans l'algorithme.

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

Q1. Analyser ligne après ligne cet algorithme.

Q2. Montrer que deux processus ne peuvent entrer en Section Critique en même temps.

Q3. Montrez qu'un processus peut entrer en Section Critique lorsque c'est possible, c'est-à-dire lorsque la section critique est libre.

Q4. Que vient-on de montrer avec ces 2 propriétés ?

Q5. Donner un inconvénient relatif à cet algorithme.

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.

TD 4 – Ordonnancement et travail à la chaîne

1 Ordonnancement avec des mutex

Nous considérons un ordonnancement préemptif avec priorité avec comme hypothèse que plus la valeur de priorité est importante plus la tâche est prioritaire. Nous considérons un jeu de tâches mélangeant des tâches périodiques et des tâches ponctuelles. De plus, les tâches B et D se partagent un mutex.

Tâche	Date(s) d'arrivée(s)	Priorité	Durée	Remarques
A	0, 6, 12, 18, 24, 30	10	1	Périodique (multiple de temps=6)
B	0, 10, 20, 30	8	4	Périodique (multiple de temps=10), à chaque itération, la tâche doit acquérir le mutex à la fin de la 1re unité de temps et la libérer à la fin de la 3e.
C	18	5	6	Ponctuelle
D	0	1	8	Ponctuelle, à la fin de la 6e unité de temps, la tâche acquiert le mutex et le conserve jusqu'à la fin de la 8e unité.

Q1. Faire l'ordonnancement de ces tâches sur 32 unités de temps.

Q2. Comment calcule-t-on le temps de réponse (ou latence) d'une tâche ?

Q3. Donnez le temps de réponse (ou latence) de chacune des tâches.

Q4. Que se passe-t-il sur la période de temps 21 à 27 ?

2 Fabrication de Doubitchous

Nous allons considérer un programme simulant le travail à la chaîne de l'usine de Preskovitch S.A. à Sofia pour la fabrication des Doubitchous (cf. Le père Noël est une ordure...). Il y aura naturellement plusieurs Doubitchous à fabriquer, soit `nb_tours = 10`.

La confection d'un Doubitchou est ainsi décomposé en 6 étapes constituant le travail à la chaîne :

1. Mélanger la farine et le cacao ;
2. Pétrir la pâte avec la margarine et l'huile ;
3. Ajouter le sucre et la cannelle ;
4. Ajouter un morceau de chocolat ;
5. Rouler le Doubitchou ;
6. Cuire le Doubitchou.

Pour fabriquer les Doubitchous, nous écrivons un programme multi-threadé, **chaque thread étant en charge de l'une des étapes** de fabrication. Le programme lancera ainsi le nombre de threads (numérotés à partir de 0) nécessaires à la fabrication de tous les Doubitchous, soit `nb_etape = 6` threads lancés. A la fin de l'exécution, le programme affichera le temps passé à la confection de chacun des Doubitchous, c'est-à-dire la somme des temps passés par chacun des threads dans chacune des étapes.

2.1 Mise en place du travail à la chaine

La fabrication d'un Doubitchou se fait ainsi à la chaîne. Par exemple, le thread en charge de la cuisson (se trouvant en position 6 de la chaîne) ne peut le faire tant que le thread à la position précédente (position 5 de la chaîne) n'a pas correctement roulé le Doubitchou. Pour mettre en place ce travail à la chaîne, nous ferons passer un "jeton" d'une étape à l'autre.

Pour cela nous disposons d'un tableau d'entiers **`nb_doubs[]`** (de taille `nb_etape+1`) représentant le nombre de Doubitchous avant chacune des étapes de la chaîne de travail :

— **`nb_doubs[0]`** représente le nombre de Doubitchous dont la fabrication n'a pas encore démarré (initialisé dans notre cas à `nb_tours = 10`).

— **`nb_doubs[nb_etape]`** représente le nombre de Doubitchous terminés (initialisé à 0).

— Les cases des autres indices sont toutes initialisées à 0.

Q1. Quel mécanisme de programmation concurrente va être mis en place ? Que représente ce tableau ?

Q2. Quel est l'intérêt d'initialiser la première case du tableau à `nb_tours = 10` ?

Q3. Quel thread aura accès à la dernière case du tableau ?

Q4. Expliciter l'évolution de l'état de ce tableau au cours de l'exécution du programme.

Q5. Que faut-il mettre en place pour ce tableau ? Et de manière générale pour les threads ?

2.2 Implémentation du programme

Pour synchroniser les threads et faire le travail à la chaîne, nous implémentons un **moniteur de Hoare**.

```
1  class chaine {
2  private:
3    unsigned nb_etapes; // les 6 etapes de notre travail
4    vector<int> nb_doubs; // nb Doubitchous dispo en entree des etapes
5    mutex m;  // mutex pour protection
6    vector<condition_variable_any> t_cond; // variables de condition
7
8  public:
9    chaine(int nb_etapes, int nb_doubichous); // constructeur
10   void affiche(); // uniqument pour du debug
11   void attend(int pos);
12   void transmet(int pos);
13   void termine();
14 };
15
16 // constructeur de la classe
17 chaine::chaine(int nb_etapes, int nb_doubichous) :
18   nb_etapes(nb_etapes),
19   nb_doubs (nb_etapes+1, 0),
20   m(),
21   t_cond(nb_etapes)
22 {
23   nb_doubs[0] = nb_doubichous;
24 }
```

Pour réaliser le programme, vous disposez de la fonction **`double travail_chaine(...)`** effectuant la conception des Doubitchous, c'est-à-dire toutes les étapes du travail à la chaîne pour chacun des Doubitchous. Cette fonction est exécutée par chacun des threads. Elle prend en paramètres :

— **c** : structure de donnée nécessaire à la synchronisation (le moniteur de Hoare) ;

— **pos** : position du thread dans la chaîne ;

— **nb_tours** : nombre de tours / Doubitchous demandés ;

— **nb_etape** : nombre d'étapes nécessaires à la préparation (dans notre cas = 6).

```
1  double travail_chaine(chaine *c, int pos, int nb_tours, int nb_etape) {
2    int i;
3    double temps = 0;
4    for (i=0; i<nb_tours; i++) { // boucle sur le nb de Doubitchous
5      c->attend(pos);           // attente travail sur position pos
6      temps += travail(pos, i); // effectue travail de la position pos
7      if (pos != nb_etape-1) {
8        c->transmet(pos+1);     // transmet travail vers pos+1
9      } else {
10       c->termine();
11       fprintf(stdout, "doub %d : le doubitchou est fini\n", i);
12     }
13   }
14   return temps;
15 }
```

L'algorithme de cette fonction est simple. Pour chacun des Doubitchous à confectionner, le thread :

— attend que le thread précédent ait fini sa préparation pour avoir du travail à la position `pos` : procédure **`void attend(int pos)`** ;

— effectue sa tâche : fonction **`double travail(int pos, int num_tours)`** qui effectue le travail pour le thread de la position **`pos`** dans la chaîne. Cette fonction mesure le temps nécessaire à la réalisation de sa tâche et le retourne. Elle affiche aussi un message permettant de connaître l'étape réalisée. Par exemple, durant la confection du 3e Doubitchou (le numéro 2), par le thread numéro 5 (i.e. le thread se trouvant à la position 6 de la chaine), la fonction affiche : "`2 : le thread 5 cuit le doubitchou`".

— s'il n'est pas le dernier, il transmet son travail au suivant : procédure **`void transmet(int pos_suiv)`**.

— s'il est le dernier, il pose le Doubitchou terminé sur le plat (qui correspond à la dernière case du tableau **`nb_doubs`**), en appelant la procédure **`void termine()`**.

Q1. Donnez le code des procédures `attend()` (qui est appelée avec `pos` en paramètre) et `transmet()` (qui est appelée avec `pos+1` en paramètre).

Q2. Donnez le code de la fonction `termine()`.

Pour finaliser notre programme, voici le code de la fonction exécutée par chacun des threads, ainsi qu'un extrait du code du programme principal lançant les threads et récupérant leur résultat.

```
void fct_thread(chaine *c, int pos, int nb_tours, int nb_etape, double *
    temps) {
  *temps = travail_chaine(c, pos, nb_tours, nb_etape);
}
```

Extrait du programme principal :

```
  chaine c(NB_ETAPES, nb_tours);

  vector<thread> ths;
  double tab_temps[nb_threads];

  c.affiche();

  for (i=0; i<nb_threads; i++) {
    ths.push_back(thread(fct_thread, &c, i, nb_tours, NB_ETAPES, &tab_temps
        [i]));
  }

  c.affiche();

  for (i=0; i<nb_threads; i++) {
    ths[i].join();
  }

  c.affiche();

  double somme = 0;
  for (i=0; i<NB_ETAPES; i++) {
    somme += tab_temps[i];
  }
  cout << "Le temps total est de " << somme << " secondes"<< endl;
```

La difficulté est de récupérer le temps renvoyé par travail_chaine afin de l'affecter à un paramètre passé par référence, puisqu'on ne peut pas récupérer la valeur de retour d'un `std::thread`.

TD 5 – Administration système

1 Des droits et des fichiers

Sur le système considéré dans cet exercice, on trouve notamment les trois utilisateurs suivants :

— `a1400` qui fait partie des groupes `etudiant` et `user` ;

— `mmoy` qui fait partie des groupes `prof` et `user` ;

— `ycaniou` qui fait partie des groupes `prof`, `user` et `ycaniou`

Dans un répertoire ayant tous les droits d'accès, la commande `ls -l` retourne ce qui suit :

```
srw-------  1  ycaniou   ycaniou  0   déc.  3 06:24 kdeinit4__0
lrwxrwxrwx  1  ycaniou   ycaniou 33    mai  2  2014 rap.tex -> rapport.tex
drwxr-xr-x 27  mmoy     user    27  janv.  2 11:06 public/
drwx--x--- 70  mmoy     prof  4096  avril 22 17:21 prive/
-rw-rw-r--  1  a1400     etu   5349   déc. 10  2008 rapport.tex
-rw-rw----  1  mmoy     prof   336  avril 19 12:45 public/sujet.tex
-rw-rw-r--  1  mmoy     prof   336  avril 19 12:45 public/sujet.pdf
-rw-r--r--  1  ycaniou   prof   336  avril 19 18:32 public/correction.pdf
-rw-r--r--  1  mmoy     prof   336  avril 19 18:32 prive/notes.ods
```

Q1. Donnez le sens de chacune des colonnes.

Q2. Que représente le fichier `kdeinit4__0` ? Et le fichier `rap.tex` ?

Q3. Les droits sur `rap.tex` sont-ils normaux ?

Q4. Représentez dans une matrice les possibilités d'accès des fichiers et répertoires pour chaque utilisateur.

Q5. Qu'est-ce qu'une telle « configuration » des droits sur `public/` et `prive/` permet de faire ?

2 Le bit « setuid »

On considère maintenant cette situation (le `$` est l'invite de commande, ou « prompt ») :

```
$ sudo ls /root
$ cat test-setuid.c
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
        int f = open("/root/i-can-haz-write-access-to-r00t.txt",
                     O_WRONLY | O_CREAT);
        if (f < 0) {
                perror("Can't open file");
                exit(1);
```

```
        }
        write(f, "pwned", strlen("pwned"));
        close(f);
}
$ gcc test-setuid.c -o test-setuid
$ ls -l test-setuid
-rwxr-xr-x 1 moy moy 8808 nov.  30 17:27 test-setuid*
$ ./test-setuid
Can't open file: Permission denied
$ sudo chown root test-setuid
$ ./test-setuid
Can't open file: Permission denied
$ sudo chmod +s test-setuid
$ ls -l test-setuid
-rwsr-sr-x 1 root moy 8808 nov.  30 17:27 test-setuid*
$ ./test-setuid
$ sudo ls /root
i-can-haz-write-access-to-r00t.txt
$
```

Q1. Pourquoi les premières exécutions de `test-setuid` échouent-elles ? Pourquoi la dernière réussit-elle ?

3 Administration Système

Lorsque la ou les commandes sont propres à une distribution, nous supposerons qu'il s'agit d'une distribution Debian ou dérivée (Ubuntu, comme votre VM Openstack).

Q1. Rappelez quelques variables d'environnement importantes. Comment toutes les lister avec leur affectation ?

Q2. Quelle est la différence entre sourcer un script (`source mon-script.sh` ou `. mon-script.sh`) et l'exécuter (`./mon-script.sh`) ?

Q3. Comment mettre en place la complétion *intelligente* ? À quoi sert-elle ?

Q4. Où sont les scripts permettant de lancer les serveurs ?

Q5. Quel est le pseudo système de fichiers permettant de configurer dynamiquement le fonctionnement du noyau ?

Les commandes suivantes peuvent se faire avec `apt`, un front-end qui se veut convivial pour gérer les paquets Debian. Mais `apt` ne fournit pas encore toutes les choses des autres outils, comme la recherche du package auquel appartient un fichier donné.

Q6. Comment lister tous les packages installés ?

Q7. Comment savoir si le package `apache2` est installé ?

Q8. Où modifier les *repositories* d'où la liste des packages et leur version, ainsi que les packages eux-mêmes seront téléchargés ?

Q9. Comment savoir quelle version d'un package sera installée si on décide de l'installer ?

Q10. Comment mettre à jour l'ensemble du système ?