

Parallélisme

TD

Concurrency (Moniteurs de Hoare)

Elana Courtines

courtines.e@gmail.com

<https://github.com/irinacake>

Séance 1 - 17 octobre 2022

Séance 1 - 24 octobre 2022

Séance 1 - 7 novembre 2022

Séance 1 - 14 novembre 2022

Amal Sayah - sayah@irit.fr

0 Rappel : Moniteur de Hoare

Qu'est-ce qu'un moniteur ?

Un moniteur est un objet ayant pour but de synchroniser des opérations entre différents processus. Celui-ci contient des méthodes toutes en exclusion mutuelles, ainsi que des variables d'état qui sont toutes "critiques".

Les moniteurs de Hoare respectent tous les 4 propriétés suivantes. Attention, il s'agit là d'une abstraction, et l'implémentation effective sur machine peut varier :

Propriété 1

À un instant donné, il n'y a qu'un seul processus actif dans le moniteur, ce qui garantit un accès exclusif aux variables d'état.

Propriété 2

Le processus actif au sein du moniteur peut être bloqué si une condition logique s'avère ne pas être vérifiée, grâce aux opérations `c.wait()` ou `wait(c)`, qui libèrent l'accès au moniteur.

Propriété 3

À un certain moment donné, un processus peut rendre vraie une condition logique attendue par un autre processus, grâce aux opérations `c.signal()` ou `signal(c)`, lui donnant ainsi immédiatement la main.

Note : malgré ce "passage de main", Hoare fait en sorte que la propriété 1 soit toujours respectée.

Propriété 4

Un processus ayant été suspendu par le processeur sera toujours prioritaire pour reprendre le contrôle.

1 Modèle des Producteurs-Consommateurs

1.1 Variante 1 - Buffer de N cases, messages d'un type unique

1.1.1 Déterminer la Spécification :

```
1 Moniteur Prod_Conso {  
    void deposer(Message msg);  
3    void retirer(Message *msg);  
    void initialiser() {}  
5};
```

```
#define N 10  
2  
typedef struct Message;  
4 typedef struct {  
    Message msg;  
6 } Case;  
  
8 Case buffer[N];  
unsigned int indVide;  
10 unsigned int indPlein;  
  
12 void faireDepot(Message msg){  
    buffer[indVide].msg = msg;  
14    indVide = (indVide+1) % N  
}  
16 void faireRetrait(Message *msg){  
    *msg = buffer[indPlein].msg;  
18    indPlein = (indPlein+1) % N  
}
```

1.1.2 Exemples de Processus :

```
1 Processus Producteur {  
    while(true)  
3        Prod_Conso.deposer(message)  
}  
5 Processus Consommateur {  
    while(true)  
7        Prod_Conso.retirer(&message)  
}
```

1.1.3 Expliciter les conditions de blocage et de réveil d'un processus et préciser dans quelles méthodes du moniteur cela se produit :

- Un producteur est bloqué dans `deposer()` si le buffer est plein ;
- Il est réveillé par un consommateur qui vient de retirer un message ;
- Un consommateur est bloqué dans `retirer()` si le buffer est vide ;
- Il est réveillé par un producteur qui vient de déposer un message.

1.1.4 En déduire les variables nécessaires au bon fonctionnement du moniteur :

- compteur du nombre de cases vides ;
- condition de blocage des producteurs : $nb_cases_libres == 0$;
- condition de blocage des consommateurs : $nb_cases_libres == N$;

1.1.5 Écrire le code du moniteur en utilisant (4) pour mettre en oeuvre (3) :

```
Moniteur Prod_Conso {
2   private unsigned nb_cases_libres;
   private Condition caseVide;
4   private Condition casePleine;

6   public void deposer(Message msg){
       if (nb_cases_libres == 0)
8           caseVide.wait()
           // 0 < nb_cases_libres <= N
10          faireDepot(msg);
           nb_cases_libres--;
12          // 0 <= nb_cases_libres < N
           casePleine.signal()
14      }
   public void retirer(Message *msg){
16       if (nb_cases_libres == N)
           casePleine.wait()
18          // 0 <= nb_cases_libres < N
           faireRetrait(msg);
           nb_cases_libres++;
20          // 0 < nb_cases_libres <= N
           caseVide.signal();
22      }
24
   public void initialiser() {
26       indVide = 0;
       indPlein = 0;
28       nb_cases_libres = N;
   }
30 };
```

1.2 Variante 2 - Message de deux types, dépôts alternés

1.2.1 Déterminer la Spécification :

```
1 Moniteur Prod_Conso {  
    void deposer(Message msg, bool type);  
3    void retirer(Message *msg, bool *type);  
    void initialiser() {}  
5 };
```

```
#define N 10  
2  
typedef struct Message;  
4 typedef struct {  
    Message msg;  
6    bool type;  
} Case;  
8  
Case buffer[N];  
10 unsigned int indVide;  
unsigned int indPlein;  
12  
void faireDepot(Message msg, bool type){  
14     buffer[indVide].msg = msg;  
    buffer[indVide].type = type;  
16     indVide = (indVide+1) % N  
}  
18 void faireRetrait(Message *msg, bool *type){  
    *msg = buffer[indPlein].msg;  
20     *type = buffer[indPlein].type;  
    indPlein = (indPlein+1) % N  
22 }
```

1.2.2 Exemples de Processus :

```
Processus Producteur {  
2     while(true)  
        Prod_Conso_V2.deposer(message, type)  
4 }  
Processus Consommateur {  
6     while(true)  
        Prod_Conso_V2.retirer(&message,&type)  
8 }
```

1.2.3 Expliciter les conditions de blocage et de réveil d'un processus et préciser dans quelles méthodes du moniteur cela se produit :

- Un producteur de type T est bloqué dans deposer() si le buffer est plein ou si le type du dernier message est T ;
- Si le buffer était plein, il est réveillé par un consommateur qui vient de retirer un message à condition que le dernier message déposé était F ;
- Si le dernier message déposé était T, il est réveillé par un producteur de type F qui vient de faire un dépôt à condition que le buffer ne soit pas plein
- Un consommateur est bloqué dans retirer() si le buffer est vide ;

- Il est réveillé par un producteur de n'importe quel type qui vient de déposer un message.

1.2.4 En déduire les variables nécessaires au bon fonctionnement du moniteur :

- compteur du nombre de cases vides ;
- un variable pour conserver le dernier type déposé ;
- condition de blocage des producteurs :
 $nb_cases_libres == 0 \parallel typeDernierMessage == type$;
- condition de blocage des consommateurs : $nb_cases_libres == N$;

1.2.5 Écrire le code du moniteur en utilisant (4) pour mettre en oeuvre (3) :

```

Moniteur Prod_Conso_V2 {
2   private bool typeDernierMsg;
   private unsigned nb_cases_libres;
4   private Condition CaseDispo[2]; // 0 pour True, 1 pour False
   private Condition MessageDispo;

6
   public void deposer(Message msg, bool type){
8       if (nb_cases_libres == 0 || type == typeDernierMsg)
           CaseDispo[type].wait();
10      // 0 < nb_cases_libres <= N && type != typeDernierMsg
       faireDepot(msg, type);
12      nb_cases_libres--;
       typeDernierMsg = type;
14      // 0 <= nb_cases_libres < N && type = typeDernierMsg
       if (nb_cases_libres != 0) //only if the buffer is not full
16          CaseDispo[!type].signal();
       MessageDispo.signal();
18   }
   public void retirer(Message *msg, bool *type){
20       if (nb_cases_libres == N)
           MessageDispo.wait();
22      // 0 <= nb_cases_libres < N
       faireRetrait(msg, type);
24      nb_cases_libres++;
       // 0 < nb_cases_libres <= N
26      CaseDispo[!typeDernierMessage].signal();
   }

28
   public void initialiser() {
30       indVide = 0;
       indPlein = 0;
32       nb_cases_libres = N;
       typeDernierMsg = false; //arbitrairement choisi
34   }
};

```

1.3 Variante 3 - Messages de deux types, choix du type de message retiré

1.3.1 Déterminer la Spécification :

```
1 Moniteur Prod_Conso {  
    void deposer(Message msg, bool type);  
3    void retirer(Message *msg, bool type);  
    void initialiser() {}  
5 };
```

```
#define N 1  
2  
typedef struct Message;  
4 typedef struct {  
    Message msg;  
6    bool type;  
} Case;  
8  
Case buffer[N];  
10 unsigned int indVide;  
unsigned int indPlein;  
12  
void faireDepot(Message msg, bool type){  
14     buffer[indVide].msg = msg;  
    buffer[indVide].type = type;  
16     indVide = (indVide+1) % N  
}  
18 void faireRetrait(Message *msg){  
    *msg = buffer[indPlein].msg;  
20     indPlein = (indPlein+1) % N  
}  
22 bool getTypeProchainMessage(){  
    return buffer[indPlein].type;  
24 }
```

1.3.2 Exemples de Processus :

```
Processus Producteur {  
2    while(true)  
        Prod_Conso_V3.deposer(message, type)  
4 }  
Processus Consommateur {  
6    while(true)  
        Prod_Conso_V3.retirer(&message, type)  
8 }
```

1.3.3 Expliciter les conditions de blocage et de réveil d'un processus et préciser dans quelles méthodes du moniteur cela se produit :

- Un producteur (peu importe son type) est bloqué dans deposer() si le buffer est plein ;
- Il est réveillé par un consommateur (peu importe lequel) qui vient de retirer un message.
- Un consommateur de type T est bloqué dans retirer() si le buffer est vide ;
- Il est réveillé par un producteur de type T également qui vient de déposer un message.

- Un consommateur de type T est bloqué dans retirer() si le prochain message disponible n'est pas de type T ;
- Il est réveillé par un consommateur qui vient de récupérer un message à condition qu'il existe un nouveau message à retirer et que ce message soit de type T.

1.3.4 En déduire les variables nécessaires au bon fonctionnement du moniteur :

- compteur du nombre de cases vides ;
- un variable pour stocker le type du prochain message (ou une fonction) ;
- condition de blocage des producteurs : $nb_cases_libres == 0$;
- condition de blocage des consommateurs :
 $nb_cases_libres == N \parallel typeProchainMessage \neq type$;

1.3.5 Écrire le code du moniteur en utilisant (4) pour mettre en oeuvre (3) :

```

Moniteur Prod_Conso_V3 {
2   // variables de synchronisation
   private unsigned nb_cases_libres;
4   private Condition CaseDispo;
   private Condition MessageDispo[2]; // 0 pour True, 1 pour False
6
   public void deposer(Message msg, bool type){
8       if (nb_cases_libres == 0)
           CaseDispo.wait();
10      // 0 < nb_cases_libres <= N
       faireDepot(msg, type);
12      nb_cases_libres--;
       // 0 <= nb_cases_libres < N
14      if (nb_cases_libres == N-1){
           MessageDispo[type];
16      }
   }
18   public void retirer(Message *msg, bool type){
       if (nb_cases_libres == N || getTypeProchainMessage() != type){
20           MessageDispo[type].wait()
       }
       faireRetrait(msg); // 0 <= nb_cases_libres < N && typeProchainMessage =
22   type
       nb_cases_libres++; // 0 < nb_cases_libres <= N && typeProchainMessage =
24   unknown
       if (nb_cases_libres != N){
           MessageDispo[getTypeProchainMessage()].signal()
26       }
       CaseDispo.signal();
28   }

   public void initialiser() {
30       nb_cases_libres = N;
   }
32 }
};

```


2 Modèle des Lecteurs-Rédacteurs

Les étapes 1&2 de toutes ces variantes sont les mêmes :

2.0.1 Déterminer la Spécification :

```
1 Moniteur Lecteur_Redacteur {  
    void demanderEcriture();  
3    void terminerEcriture();  
    void demanderLecture();  
5    void terminerLecture();  
    void initialiser();  
7 };
```

2.0.2 Exemples de Processus :

```
Processus Lecteur() {  
2    while(true) {  
        Lecteur_Redacteur.demanderLecture();  
4        lire(f);  
        Lecteur_Redacteur.terminerLecture();  
6    }  
}  
8  
Processus Redacteur() {  
10    while(true) {  
        Lecteur_Redacteur.demanderEcriture();  
12        ecrire(f);  
        Lecteur_Redacteur.terminerEcriture();  
14    }  
}
```

2.1 Variante 1

2.1.3 Expliciter les conditions de blocage et de réveil d'un processus et préciser dans quelles méthodes du moniteur cela se produit :

- Un Redacteur est bloqué dans demanderEcriture() si un autre Redacteur est en train d'écrire ;
- Il est débloqué par le précédent redacteur dans terminerEcriture() après que celui-ci ait terminé ;
- Un Redacteur est bloqué dans demanderEcriture() si un ou plusieurs Lecteurs sont en train de lire ;
- Il est débloqué par un lecteur dans terminerLecture() si celui-ci a terminé et qu'il n'y a aucun autre lecteur en train de lire ;
- Un Lecteur est bloqué dans demanderLecture() si un Redacteur est en train d'écrire ;
- Il est débloqué par un Redacteur dans terminerEcriture() si celui-ci a terminé d'écrire et qu'aucun autre redacteur n'attend pour écrire ;
- Il peut aussi être débloqué par un lecteur qui vient d'avoir accès à la lecture (réveil en cascade).

2.1.4 En déduire les variables nécessaires au bon fonctionnement du moniteur :

- compteur du nombre de Lecteurs actifs ;
- booléen indiquant si un Rédacteur est en train d'écrire ;
- condition de blocage des Rédacteurs : un Rédacteur est en train d'écrire, ou au moins un Lecteur est en train de lire ;
- condition de blocage des Lecteurs : un Rédacteur est en train d'écrire.

2.1.5 Écrire le code du moniteur en utilisant (4) pour mettre en oeuvre (3) :

```
1 Moniteur Lecteur_Redacteur {
    private Condition redacteurs;
3    private Condition lecteurs;
    private unsigned int nb_lecteurs_actifs;
5    private bool redacteur_actif;

7    void demanderEcriture() {
        if (redacteur_actif || nb_lecteurs_actifs > 0) {
9            redacteurs.wait();
        }
11       redacteur_actif = true;
    }
13    void terminerEcriture() {
        redacteur_actif = false;
15        if (!redacteurs.isEmpty()) {
            redacteurs.signal();
17        } else {
            lecteurs.signal();
19        }
    }
21    void demanderLecture() {
        if (redacteur_actif) {
23            lecteurs.wait();
        }
25        nb_lecteurs_actifs++;
        lecteurs.signal(); // réveil en cascade
27    }
29    void terminerLecture() {
        nb_lecteurs_actifs--;
        if (nb_lecteurs_actifs == 0) {
31            redacteurs.signal();
        }
33    }

35    void initialiser() {
        nb_lecteurs_actifs = 0;
37        redacteur_actif = false;
    }
39 };
```

2.1.6 Problèmes posés par cette version :

- famine des lecteurs : s'il y a constamment un redacteur en attente pendant qu'un autre est encore en train d'écrire ;
- famine des redacteurs : s'il y a constamment des lecteurs en train de lire

2.2 Variante 2 - Priorité des rédacteurs sur les lecteurs

2.2.3 Expliciter les conditions de blocage et de réveil d'un processus et préciser dans quelles méthodes du moniteur cela se produit :

- Un Redacteur est bloqué dans demanderEcriture() si un autre Redacteur est en train d'écrire ;
- Il est débloqué par le précédent redacteur dans terminerEcriture() après que celui-ci ait terminé ;
- Un Redacteur est bloqué dans demanderEcriture() si un ou plusieurs Lecteurs sont en train de lire ;
- Il est débloqué par un lecteur dans terminerLecture() si celui-ci a terminé et qu'il n'y a aucun autre lecteur en train de lire ;
- Un Lecteur est bloqué dans demanderLecture() si un Redacteur est en train d'écrire ou en attente d'accès ;
- Il est débloqué par un Redacteur dans terminerEcriture() si celui-ci a terminé d'écrire et qu'aucun autre redacteur n'attend pour écrire ;
- Il peut aussi être débloqué par un lecteur qui vient d'avoir accès à la lecture (réveil en cascade).

2.2.4 En déduire les variables nécessaires au bon fonctionnement du moniteur :

- compteur du nombre de Lecteurs actifs ;
- booléen indiquant si un Rédacteur est en train d'écrire ;
- condition de blocage des Rédacteurs : un Rédacteur est en train d'écrire, ou au moins un Lecteur est en train de lire ;
- condition de blocage des Lecteurs : un Rédacteur est en train d'écrire, ou un Rédacteur est en attente d'accès.

2.2.5 Écrire le code du moniteur en utilisant (4) pour mettre en oeuvre (3) :

```
1 Moniteur Lecteur_Redacteur {
    private Condition redacteurs;
3    private Condition lecteurs;
    private unsigned int nb_lecteurs_actifs;
5    private bool redacteur_actif;

7    void demanderEcriture() {
        if (redacteur_actif || nb_lecteurs_actifs > 0) {
9            redacteurs.wait();
        }
11       redacteur_actif = true;
    }
13    void terminerEcriture() {
        redacteur_actif = false;
15        if (!redacteurs.isEmpty()) {
            redacteurs.signal();
17        } else {
            lecteurs.signal();
19        }
    }
21    void demanderLecture() {
        if (redacteur_actif || !redacteur.isEmpty()) {
23            lecteurs.wait();
        }
25        nb_lecteurs_actifs++;
        lecteurs.signal(); // réveil en cascade
27    }
    void terminerLecture() {
29        nb_lecteurs_actifs--;
        if (nb_lecteurs_actifs == 0) {
31            redacteurs.signal();
        }
33    }

35    void initialiser() {
        nb_lecteurs_actifs = 0;
37        redacteur_actif = false;
    }
39 };
```

2.2.6 Problèmes posés par cette version :

Cette version 2 règle le problème de famine des rédacteurs, mais ne règle pas le problème de famine des lecteurs

2.3 Variante 3 - Gestion prioritaire des accès

2.3.3 Expliciter les conditions de blocage et de réveil d'un processus et préciser dans quelles méthodes du moniteur cela se produit :

- Un Redacteur est bloqué dans demanderEcriture() si un autre Redacteur est en train d'écrire ;
- Il est débloqué par le précédent redacteur dans terminerEcriture() après que celui-ci ait terminé à condition qu'il n'y ait aucun lecteur en attente ;
- Un Redacteur est bloqué dans demanderEcriture() si un ou plusieurs Lecteurs sont en train de lire ;
- Il est débloqué par un lecteur dans terminerLecture() si celui-ci a terminé et qu'il n'y a aucun autre lecteur en train de lire ;
- Un Lecteur est bloqué dans demanderLecture() si un Redacteur est en train d'écrire ou en attente d'accès ;
- Il est débloqué par un Redacteur dans terminerEcriture() si celui-ci a terminé d'écrire et qu'aucun autre redacteur n'attend pour écrire ;
- Il peut aussi être débloqué par un lecteur qui vient d'avoir accès à la lecture (réveil en cascade).

2.3.4 En déduire les variables nécessaires au bon fonctionnement du moniteur :

- compteur du nombre de Lecteurs actifs ;
- booléen indiquant si un Rédacteur est en train d'écrire ;
- condition de blocage des Rédacteurs : un Rédacteur est en train d'écrire, ou au moins un Lecteur est en train de lire ;
- condition de blocage des Lecteurs : un Rédacteur est en train d'écrire, ou un Rédacteur est en attente d'accès.

2.3.5 Écrire le code du moniteur en utilisant (4) pour mettre en oeuvre (3) :

```
1 Moniteur Lecteur_Redacteur {
    private Condition redacteurs;
3    private Condition lecteurs;
    private unsigned int nb_lecteurs_actifs;
5    private bool redacteur_actif;

7    void demanderEcriture() {
        if (redacteur_actif || nb_lecteurs_actifs > 0) {
9            redacteurs.wait();
        }
11       redacteur_actif = true;
    }
13    void terminerEcriture() {
        redacteur_actif = false;
15        if (lecteurs.isEmpty()) {
            redacteurs.signal();
17        } else {
            lecteurs.signal(); // Cascade automatique si au moins 1 en attente
19        }
    }
21    void demanderLecture() {
        if (redacteur_actif || !redacteur.isEmpty()) {
23            lecteurs.wait();
        }
25        nb_lecteurs_actifs++;
        lecteurs.signal(); // réveil en cascade
27    }
    void terminerLecture() {
29        nb_lecteurs_actifs--;
        if (nb_lecteurs_actifs == 0) {
31            redacteurs.signal();
        }
33    }

35    void initialiser() {
        nb_lecteurs_actifs = 0;
37        redacteur_actif = false;
    }
39 };
```

2.4 Variante 4 - Gestion FIFO des accès

2.4.3 Expliciter les conditions de blocage et de réveil d'un processus et préciser dans quelles méthodes du moniteur cela se produit :

- Un Rédacteur est bloqué à la queue de la file normale dans `demanderEcriture()` si un autre Redacteur est en train d'écrire ;
- Un Rédacteur est bloqué à la queue de la file normale dans `demanderEcriture()` si au moins un lecteur est en train de lire ;
- Un Rédacteur est bloqué en file prioritaire dans `demanderEcriture()` s'il vient d'être réveillé mais que des lecteurs sont encore en train de lire ;
- Un Rédacteur est débloqué par un rédacteur dans `terminerEcriture()` si celui-ci a terminé d'écrire à condition que ce rédacteur soit le premier processus en attente dans la file normale ;
- Un Rédacteur est débloqué par un lecteur dans `terminerLecture()` si celui-ci a terminé de lire et qu'il est le dernier lecteur à condition que ce rédacteur soit le premier processus en attente dans la file normale ;
- Un Rédacteur est débloqué par un lecteur dans `terminerLecture()` si celui-ci a terminé de lire et qu'il est le dernier lecteur actif à condition que ce rédacteur soit dans la file d'attente prioritaire ;
- Un Lecteur est bloqué à la queue de la file normale dans `demanderLecture()` si un Rédacteur est en train d'écrire ou si au moins un processus est en attente ;
- Un Lecteur est débloqué par un Rédacteur dans `terminerEcriture()` si celui-ci a terminé d'écrire à condition que ce lecteur soit le premier processus en attente ;
- Un Lecteur est débloqué par un lecteur qui vient d'avoir accès à la lecture (réveil en cascade).

2.4.4 En déduire les variables nécessaires au bon fonctionnement du moniteur :

- compteur du nombre de Lecteurs actifs ;
- booléen indiquant si un Rédacteur est en train d'écrire ;
- condition de blocage en file normale des Rédacteurs : un Rédacteur est en train d'écrire, ou au moins un Lecteur est en train de lire ;
- condition de blocage en file prioritaire des Rédacteurs : au moins un Lecteur est en train de lire ;
- condition de blocage des Lecteurs : un Rédacteur est en train d'écrire, ou un processus (peut importe lequel) est en attente d'accès.

2.4.5 Écrire le code du moniteur en utilisant (4) pour mettre en oeuvre (3) - version "File avec priorité" :

```
1 #define PRIORITAIRE 0
2 #define NORMAL 0
3
4 Moniteur Lecteur_Redacteur {
5     private Condition acces;
6     private unsigned int nb_lecteurs_actifs;
7     private bool redacteur_actif;
8
9     void demanderEcriture() {
10         if (redacteur_actif || nb_lecteurs_actifs > 0) {
11             acces.wait(NORMAL);
12         }
13         if (nb_lecteurs_actifs > 0) {
14             acces.wait(PRIORITAIRE);
15         }
16         redacteur_actif = true;
17     }
18     void terminerEcriture() {
19         redacteur_actif = false;
20         acces.signal();
21     }
22     void demanderLecture() {
23         if (redacteur_actif || !acces.isEmpty()) {
24             acces.wait(NORMAL);
25         }
26         nb_lecteurs_actifs++;
27         acces.signal(); // lecteur ou rédacteur
28     }
29     void terminerLecture() {
30         nb_lecteurs_actifs--;
31         if (nb_lecteurs_actifs == 0) { // dernier lecteur
32             acces.signal(); // rédacteur obligatoirement
33         }
34     }
35
36     void initialiser() {
37         nb_lecteurs_actifs = 0;
38         redacteur_actif = false;
39     }
40 };
```

2.4.6 Écrire le code du moniteur en utilisant (4) pour mettre en oeuvre (3) - version "avec Sas" :

```
1 Moniteur Lecteur_Redacteur {
    private Condition acces;
3    private Condition sas;
    private unsigned int nb_lecteurs_actifs;
5    private bool redacteur_actif;

7    void demanderEcriture(){
        if (redacteur_actif || nb_lecteurs_actifs > 0){
9            acces.wait();
        }
        if (nb_lecteurs_actifs > 0){
11            sas.wait();
        }
        redacteur_actif = true;
13    }
15    void terminerEcriture(){
        redacteur_actif = false;
        acces.signal();
17    }
19    void demanderLecture(){
        if (redacteur_actif || !acces.isEmpty() || !sas.isEmpty()){
21            acces.wait();
        }
        nb_lecteurs_actifs++;
23        acces.signal(); // réveil en cascade
    }
25    void terminerLecture(){
        nb_lecteurs_actifs--;
        if(nb_lecteurs_actifs == 0){
27            if (sas.isEmpty()){
                acces.signal();
31            } else {
                sas.signal();
33            }
        }
35    }
37    void initialiser() {
        nb_lecteurs_actifs = 0;
        redacteur_actif = false;
39    }
41 };
```

3 Traitement de commandes

3.1 Déterminer la Spécification :

```
1 Moniteur GererCmdes {  
    void commander();  
3    void commencerEtape();  
    void terminerEtape();  
5    void initialiser();  
};
```

3.2 Exemples de Processus :

```
1 void appliquerEtape(NumeroEtape numEtape, Commande *uneCommande);  
  
3 Processus Client {  
    // Se rendre au magasin  
5    GererCmdes.commander(listeDeCourse);  
    // Rentrer chez soi, satisfait :widepeepohappy:  
7 }  
  
9 Processus Employe (NumeroEtape etapeAppliquee) {  
    while (1) {  
11        GererCmdes.commencerEtape(etapeAppliquee, &cmdeATraiter, &guichetOrigine  
    );  
        appliquerEtape(etapeAppliquee, cmdeATraiter);  
13        GererCmdes.terminerEtape(cmdeATraiter, guichetOrigine);  
    }  
15 }
```

3.3 Préciser les conditions de blocage et de réveil d'un processus Client et d'un processus Employe :

- Un Client est bloqué au début de commander() si aucun guichet n'est disponible ;
- Un Client qui était bloqué au début de commander() est débloquent par n'importe quel Client dans commander() après que ce dernier ait récupéré sa commande() ;
→ un tel Client sera alors à un guichet G ;
- Un Client est bloqué (en attente au guichet G) à la fin de commander() après avoir passé une commande ;
- Un Client qui était bloqué à la fin de commander() est débloquent par un Employe dans terminerEtape() s'il traite la dernière étape (nbEtape-1) et qu'il a terminé de traiter la commande du guichet auquel le Client a passé la commande ;
- Un Employe d'étape E est bloqué dans commencerEtape() s'il n'y a aucune commande en attente d'être traitée à l'étape E ;
Il est réveillé par :
 - un Client qui vient de passer sa commande dans commander() si cet Employe traite l'étape 0 ;
 - un Employe qui vient d'appliquer son étape dans terminerEtape() si cet Employe traite une étape !=0 ;

3.4 En déduire les variables nécessaires au bon fonctionnement du moniteur :

- un compteur du nombre de guichets libres ;
- condition de blocage en début de commande des Clients : aucun guichet n'est libre ;
- condition de blocage en fin de commande des Clients : la commande n'a pas terminé d'être traité ;
- condition de blocage des Employés : aucune commande n'ayant besoin d'être traitée est à son étape.
- *et peut-être bien plus encore !*

3.5 Écrire le code du moniteur en utilisant (4) pour mettre en oeuvre (3) :



Note : seul le (3) a été traité en TD. Le (4) n'a été que partiellement commencé, le (5) n'a pas été traité du tout.