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 moniteur (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 :

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

```
#define N 10

typedef struct Message;
typedef struct {
    Message msg;
} Case;

Case buffer[N];
unsigned int indVide;
unsigned int indPlein;

void faireDepot(Message msg){
    buffer[indVide].msg = msg;
    indVide = (indVide+1) % N
}
void faireRetrait(Message *msg){
    *msg = buffer[indPlein].msg;
    indPlein = (indPlein+1) % N
}
```

1.1.2 Exemples de Processus :

```
Processus Producteur {
    while(true)
        Prod_Conso.deposer(message)
}
Processus Consommateur {
    while(true)
        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 deposer un message.

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

- compteur du nombre de cases vides ;

- condition de bloquage des producteurs : $nb\_cases\_libres == 0$;
- condition de bloquage 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 {
    private unsigned nb_cases_libres;
    private Condition caseVide;
    private Condition casePleine;

    public void deposer(Message msg){
        if (nb_cases_libres == 0)
            caseVide.wait()
        // 0 < nb_cases_libres <= N
        faireDepot(msg);
        nb_cases_libres--;
        // 0 <= nb_cases_libres < N
        casePleine.signal()
    }
    public void retirer(Message *msg){
        if (nb_cases_libres == N)
            casePleine.wait()
        // 0 <= nb_cases_libres < N
        faireRetrait(msg);
        nb_cases_libres++;
        // 0 < nb_cases_libres <= N
        caseVide.signal();
    }

    public void initialiser() {
        indVide = 0;
        indPlein = 0;
        nb_cases_libres = N;
    }
};
```

1.2 Variante 2 - Message de deux types, dépôts alternés

1.2.1 Déterminer la Spécification :

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

```
#define N 10

typedef struct Message;
typedef struct {
    Message msg;
    bool type;
} Case;

Case buffer[N];
unsigned int indVide;
unsigned int indPlein;

void faireDepot(Message msg, bool type){
    buffer[indVide].msg = msg;
    buffer[indVide].type = type;
    indVide = (indVide+1) % N
}
void faireRetrait(Message *msg, bool *type){
    *msg = buffer[indPlein].msg;
    *type = buffer[indPlein].type;
    indPlein = (indPlein+1) % N
}
```

1.2.2 Exemples de Processus :

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

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 deposer 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 bloquage des producteurs :
  $nb\_cases\_libres == 0 \,||\, typeDernierMessage == type$;
- condition de bloquage 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 {
    private bool typeDernierMsg;
    private unsigned nb_cases_libres;
    private Condition CaseDispo[2]; // 0 pour True, 1 pour False
    private Condition MessageDispo;

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

    public void initialiser() {
        indVide = 0;
        indPlein = 0;
        nb_cases_libres = N;
        typeDernierMsg = false; //arbitrairement choisi
    }
};
```

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

1.3.1 Déterminer la Spécification :

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

```
#define N 1

typedef struct Message;
typedef struct {
    Message msg;
    bool type;
} Case;

Case buffer[N];
unsigned int indVide;
unsigned int indPlein;

void faireDepot(Message msg, bool type){
    buffer[indVide].msg = msg;
    buffer[indVide].type = type;
    indVide = (indVide+1) % N
}
void faireRetrait(Message *msg){
    *msg = buffer[indPlein].msg;
    indPlein = (indPlein+1) % N
}
bool getTypeProchainMessage(){
    return buffer[indPlein].type;
}
```

1.3.2 Exemples de Processus :

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

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 ou si le prochain message disponible n'est pas de type T ;
  → Si le buffer était vide, il est réveillé par un producteur du même type T qui vient de deposer un message.
  → Si le prochain message disponible n'était pas de type T, il est réveillé par un consommateur qui vient de récuper 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 bloquage des producteurs : $nb\_cases\_libres == 0$;
- condition de bloquage des consommateurs :
  $nb\_cases\_libres == N \ || \ typeProchainMessage \neq type$;

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

```
Moniteur Prod_Conso_V3 {
    // variables de synchronisation
    private unsigned nb_cases_libres;
    private Condition CaseDispo;
    private Condition MessageDispo[2]; // 0 pour True, 1 pour False

    public void deposer(Message msg, bool type){
        if (nb_cases_libres == 0)
            CaseDispo.wait();
        // 0 < nb_cases_libres <= N
        faireDepot(msg,type);
        nb_cases_libres--;
        // 0 <= nb_cases_libres < N
        if (nb_cases_libres == N-1){
            MessageDispo[type];
        }
    }
    public void retirer(Message *msg, bool type){
        if (nb_cases_libres == N || getTypeProchainMessage() != type){
            MessageDispo[type].wait()
        }
        faireRetrait(msg); // 0 <= nb_cases_libres < N && typeProchainMessage =
    type
        nb_cases_libres++; // 0 < nb_cases_libres <= N && typeProchainMessage =
    unknown
        if (nb_cases_libres != N){
            MessageDispo[getTypeProchainMessage()].signal()
        }
        CaseDispo.signal();
    }

    public void initialiser() {
        nb_cases_libres = N;
    }
};
```

2 Modèle des Lecteurs-Rédacteurs

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

2.0.1 Déterminer la Spécification :

```
Moniteur Lecteur_Redacteur {
    void demanderEcriture();
    void terminerEcriture();
    void demanderLecture();
    void terminerLecture();
    void initialiser();
};
```

2.0.2 Exemples de Processus :

```
Processus Lecteur(){
    while(true){
        Lecteur_Redacteur.demanderLecture();
        lire(f);
        Lecteur_Redacteur.terminerLecture();
    }
}

Processus Redacteur(){
    while(true){
        Lecteur_Redacteur.demanderEcriture();
        ecrire(f)
        Lecteur_Redacteur.terminerEcriture();
    }
}
```

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, ou si un ou plusieurs Lecteurs sont en train de lire ;
  → S'il était bloqué par un autre rédacteur en train d'écrire, il est debloqué par ce dernier dans terminerEcriture() après qu'il ait terminé ;
  → S'il était bloqué par des Lecteurs en train de lire, il est debloqué par un de ces lecteurs 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 debloqué 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 debloqué 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 bloquage des Rédacteurs : $redacteurActif == True \mid\mid nbLecteursActifs > 0$;
- condition de bloquage des Lecteurs : $redacteurActif == True$.

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

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

    void demanderEcriture(){
        if (redacteur_actif || nb_lecteurs_actifs > 0){
            redacteurs.wait();
        }
        redacteur_actif = true;
    }
    void terminerEcriture(){
        redacteur_actif = false;
        if (!redacteurs.isEmpty()) {
            redacteurs.signal()
        } else {
            lecteurs.signal();
        }
    }
    void demanderLecture(){
        if (redacteur_actif){
            lecteurs.wait();
        }
        nb_lecteurs_actifs++;
        lecteurs.signal(); // réveil en cascade
    }
    void terminerLecture(){
        nb_lecteurs_actifs--;
        if(nb_lecteurs_actifs == 0){
            redacteurs.signal();
        }
    }

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

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, ou si un ou plusieurs Lecteurs sont en train de lire ;
  → S'il était bloqué par un autre rédacteur en train d'écrire, il est debloqué par ce dernier dans terminerEcriture() après qu'il ait terminé ;
  → S'il était bloqué par des Lecteurs en train de lire, il est debloqué par un de ces lecteurs 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 debloqué 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 debloqué 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 bloquage des Rédacteurs : $redacteurActif == True \,||\, nbLecteursActifs > 0$;
- condition de bloquage des Lecteurs : $redacteurActif == True \,||\, redacteurEnAttente == True$.

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

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

    void demanderEcriture(){
        if (redacteur_actif || nb_lecteurs_actifs > 0){
            redacteurs.wait();
        }
        redacteur_actif = true;
    }
    void terminerEcriture(){
        redacteur_actif = false;
        if (!redacteurs.isEmpty()) {
            redacteurs.signal()
        } else {
            lecteurs.signal();
        }
    }
    void demanderLecture(){
        if (redacteur_actif || !redacteur.isEmpty()){
            lecteurs.wait();
        }
        nb_lecteurs_actifs++;
        lecteurs.signal(); // réveil en cascade
    }
    void terminerLecture(){
        nb_lecteurs_actifs--;
        if(nb_lecteurs_actifs == 0){
            redacteurs.signal();
        }
    }

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

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, ou si un ou plusieurs Lecteurs sont en train de lire ;
  → S'il était bloqué par un autre rédacteur en train d'écrire, il est debloqué par ce dernier dans terminerEcriture() après qu'il ait terminé à condition qu'il n'y ait aucun lecteur en attente ;
  → S'il était bloqué par des Lecteurs en train de lire, il est debloqué par un de ces lecteurs 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 debloqué 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 debloqué 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 bloquage des Rédacteurs : un Rédacteur est en train d'écrire, ou au moins un Lecteur est en train de lire ;
- condition de bloquage 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) :

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

    void demanderEcriture(){
        if (redacteur_actif || nb_lecteurs_actifs > 0){
            redacteurs.wait();
        }
        redacteur_actif = true;
    }
    void terminerEcriture(){
        redacteur_actif = false;
        if (lecteurs.isEmpty()){
            redacteurs.signal();
        } else {
            lecteurs.signal(); // Cascade automatique si au moins 1 en attente
        }
    }
    void demanderLecture(){
        if (redacteur_actif || !redacteur.isEmpty()){
            lecteurs.wait();
        }
        nb_lecteurs_actifs++;
        lecteurs.signal(); // réveil en cascade
    }
    void terminerLecture(){
        nb_lecteurs_actifs--;
        if(nb_lecteurs_actifs == 0){
            redacteurs.signal();
        }
    }

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

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 bloquage en file normale des Rédacteurs : un Rédacteur est en train d'écrire, ou au moins un Lecteur est en train de lire ;
- condition de bloquage en file prioritaire des Rédacteurs : au moins un Lecteur est en train de lire ;
- condition de bloquage 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" :

```
#define PRIORITAIRE 0
#define NORMAL 0

Moniteur Lecteur_Redacteur {
    private Condition acces;
    private unsigned int nb_lecteurs_actifs;
    private bool redacteur_actif;

    void demanderEcriture(){
        if (redacteur_actif || nb_lecteurs_actifs > 0){
            acces.wait(NORMAL);
        }
        if (nb_lecteurs_actifs > 0){
            acces.wait(PRIORITAIRE);
        }
        redacteur_actif = true;
    }
    void terminerEcriture(){
        redacteur_actif = false;
        acces.signal();
    }
    void demanderLecture(){
        if (redacteur_actif || !acces.isEmpty()){
            acces.wait(NORMAL);
        }
        nb_lecteurs_actifs++;
        acces.signal(); // lecteur ou rédacteur
    }
    void terminerLecture(){
        nb_lecteurs_actifs--;
        if(nb_lecteurs_actifs == 0){ // dernier lecteur
            acces.signal(); // rédacteur obligatoirement
        }
    }

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

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

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

    void demanderEcriture(){
        if (redacteur_actif || nb_lecteurs_actifs > 0){
            acces.wait();
        }
        if (nb_lecteurs_actifs > 0){
            sas.wait();
        }
        redacteur_actif = true;
    }
    void terminerEcriture(){
        redacteur_actif = false;
        acces.signal();
    }
    void demanderLecture(){
        if (redacteur_actif || !acces.isEmpty() || !sas.isEmpty()){
            acces.wait();
        }
        nb_lecteurs_actifs++;
        acces.signal(); // réveil en cascade
    }
    void terminerLecture(){
        nb_lecteurs_actifs--;
        if(nb_lecteurs_actifs == 0){
            if (sas.isEmpty()){
                acces.signal();
            } else {
                sas.signal();
            }
        }
    }

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

3 Traitement de commandes

3.1 Déterminer la Spécification :

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

3.2 Exemples de Processus :

```
void appliquerEtape(NumeroEtape numEtape, Commande *uneCommande);

Processus Client {
    // Se rendre au magasin
    GererCmdes.commander(listeDeCourse);
    // Rentrer chez soi, satisfait :widepeepohappy:
}

Processus Employe (NumeroEtape etapeAppliquee) {
    while (1) {
        GererCmdes.commencerEtape(etapeAppliquee, &cmdeATraiter, &guichetOrigine
    );
        appliquerEtape(etapeAppliquee, cmdeATraiter);
        GererCmdes.terminerEtape(cmdeATraiter, guichetOrigine);
    }
}
```

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 débloqué 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 débloqué 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 (initialisé à nbMaxGuichets) ;
- condition de bloquage en début de commande des Clients : aucun guichet n'est libre (Condition guichetLibre) ;
- condition de bloquage en fin de commande des Clients : la commande n'a pas terminé d'être traité (condition à l'intérieur de la structure du Guichet ci-dessous) ;
- condition de blocage des Employés : aucune commande n'ayant besoin d'être traitée est à son étape (Condition commandeALEtape[nbNaxEtapes]) ;
- on pourra définir une structure pour les Guichets (stoqués dans un tableau) contenant :
  - Condition finCommande ;
  - Commande commande ;
  - unsigned int numEtape (initialisé à 0) ;
  - EtatGuichet etatCourant (initialisé à LIBRE) ;
- avec : enum EtatGuichet {LIBRE, EN_ATTENTE, EN_COURS} ;
- une fonction int trouverCommande(unsigned int numEtape) qui renvoie le numéro du premier guichet à l'étape demandé qui soit EN_ATTENTE, ou -1 s'il n'y en a aucun.

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

Note : seuls les (3) et (4) ont été traités, le (5) n'a pas été traité.