Introduction à la programmation concurrente

Moniteurs

Yann Thoma, Jonas Chapuis

Reconfigurable and Embedded Digital Systems Institute
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Février 2018

Introduction

- Concept de moniteur
  - Proposé par Hoare en 1974
- Permet de résoudre la synchronisation des tâches
- Permet de regrouper des variables ainsi que les procédures agissant sur ces variables
- Assure l'exclusion mutuelle sur les procédures du moniteur
- Synchronisation assurée par des primitives
  - wait()
  - signal()

Introduction

- Les procédures du moniteur s'exécutent en exclusion mutuelle
- La synchronisation se fait via des *variables de condition*
- L'idée est que l'exécution d'une partie de code dépend d'une condition
  - Y-a-t-il une donnée à exploiter?
  - Est-ce que le thread Y a terminé son traitement?
  - ...
- Une variable de condition va permettre de faire attendre le thread jusqu'à ce qu'une condition C soit remplie
- Un autre thread pourra réveiller un ou plusieurs threads lorsqu'il aura garanti cette condition C

Structure d'un moniteur (Hoare)

monitor <*nom*>
  <*déclarations des variables rémanentes locales*>;
  <*déclarations des variables conditions*>;
  procedure *OpérationLocale*(*liste de paramètres*)
    <*déclarations des variables locales*>;
    begin <*code pour implémenter l'opération*>;
    end;
  entry procedure *OpérationVisible*(*liste de paramètres*)
    <*déclarations des variables locales*>;
    begin <*code pour implémenter l'opération*>;
    end;
  begin <*code pour initialiser les variables rémanentes*>;
  end;

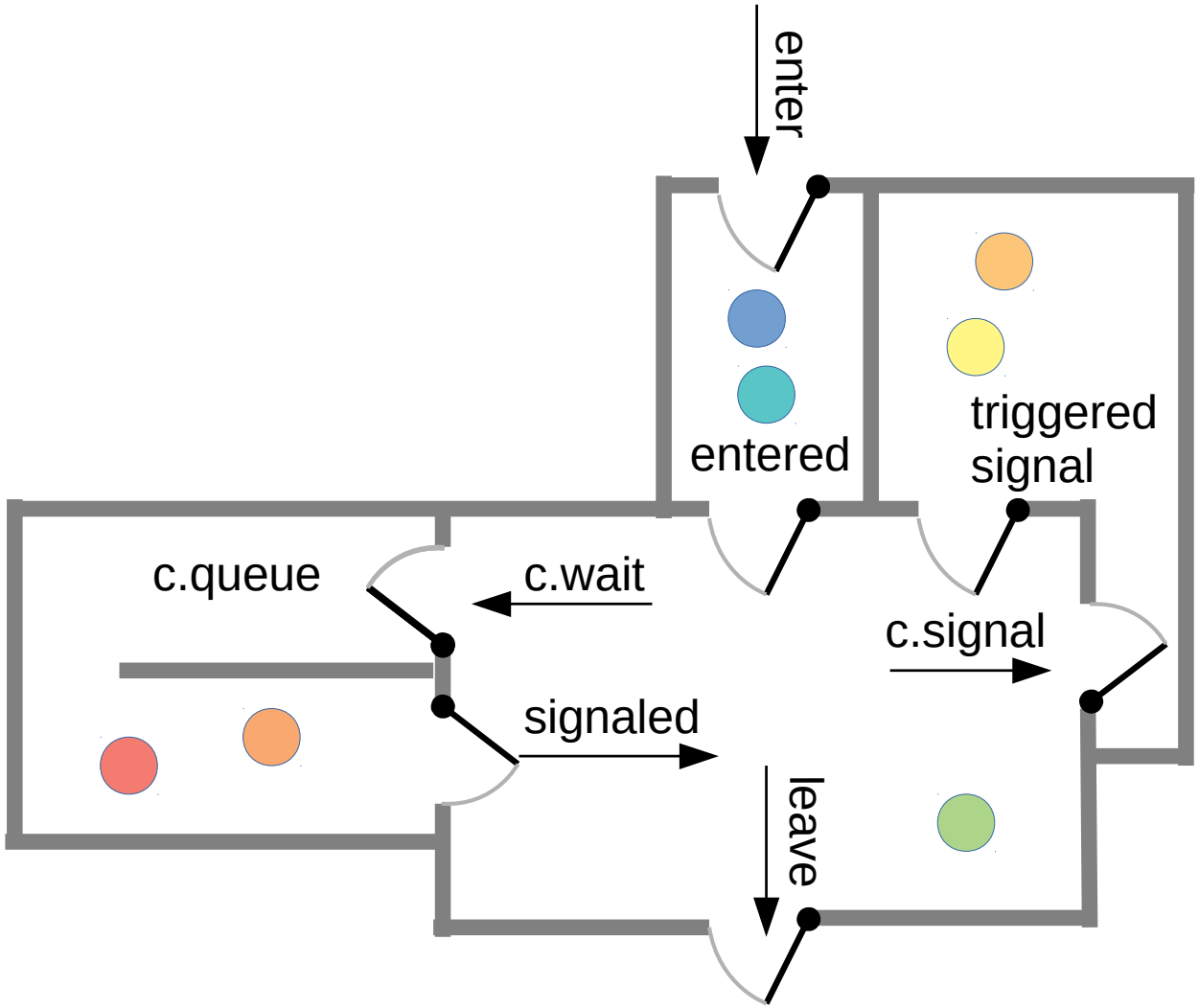
Variable de condition (Hoare)

- Synchronisation des tâches grâce aux *variables conditions* (VC)
- Une VC offre 2 primitives:
  - *attente*
  - *signale*
- Soit la VC *cond* déclarée par

  condition *cond*;

- *cond.attente*
  - bloque inconditionnellement la tâche appelante
  - lui fait relâcher l'exclusion mutuelle sur le moniteur
  - la place dans une file associée à *cond*
- *cond.signale*
  - dépend de l'état de la file associée à *cond*
  - vide $\Rightarrow$ la tâche appelante poursuit son exécution et l'opération n'a aucun effet.
  - pas vide $\Rightarrow$ une des tâches bloquées est réactivée et reprend immédiatement son exécution

Illustration d'un moniteur (Hoare)



Exemple: Verrou par moniteur (Hoare)

```
monitor VerrouMoniteur
    var verrou: boolean;
    var acces: condition;

    entry procedure Verrouille
        begin
            if verrou then acces.attente;
            verrou := true;
        end Verrouille;

    entry procedure Deverrouille
        begin
            verrou := false;
            acces.signale;
        end Deverrouille;

    begin
        verrou := false;
    end VerrouMoniteur;
```

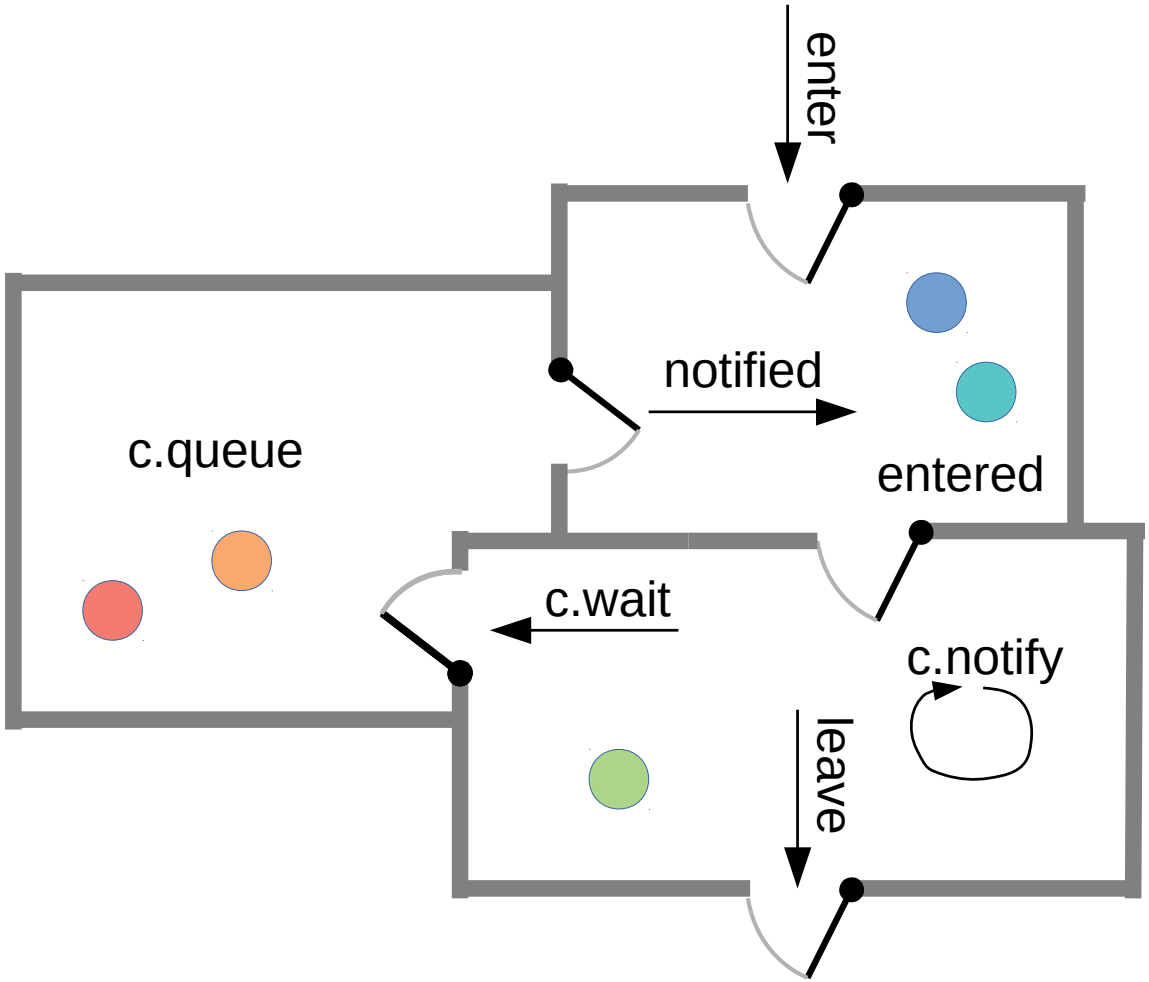
Exemple: Producteur-consommateur (Hoare)

```
monitor Tampons
    var place: array [0..N-1] of ARTICLE;
    var head, tail, size: integer;
    var notFull, notEmpty: condition;
    entry procedure deposer(a: ARTICLE)
        begin
            if size = N then notFull.attente;
            size := size + 1;
            place[head] := a;
            head := (head + 1) mod N;
            notEmpty.signale;
        end deposer;
    entry procedure retirer(var a: ARTICLE)
        begin
            if size = 0 then notEmpty.attente;
            a := place[tail];
            size := size - 1;
            tail := (tail + 1) mod N;
            notFull.signale;
        end retirer;
    begin size := 0; tail := 0; head := 0;
    end Tampons;
```

Type de moniteur

- Moniteur de type Mesa
  - Le thread qui appelle `signal` garde le mutex
  - Java, pthread, Qt
- Moniteur de type Hoare
  - Le thread qui est réveillé par `signal` prend possession du mutex
  - Pour C++, implémentation avec des sémaphores

Illustration d'un moniteur Mesa



Moniteur Qt

- Un moniteur en Qt associe:
  - Un mutex, qui assure l'exclusion mutuelle
  - Une variable de condition, qui sert de point de signalisation
    - Classe **`QWaitCondition`**

Fonctions : Attente

```
bool QWaitCondition::wait(QMutex * lockedMutex,
                          unsigned long time = ULONG_MAX)
```

- Effectue les opérations suivantes de manière atomique:
  - Relâche le mutex `lockedMutex`
  - Attend que la variable de condition soit signalée.
- L'exécution du thread est suspendue (attente passive) jusqu'à ce que la variable condition soit signalée.
- Le mutex doit être verrouillé par le thread avant l'appel à **`wait`**.
- Au moment où la condition est signalée, **`wait`** re-verrouille automatiquement le mutex.

  ⚠ Attention, lors du re-verrouillage le thread est en compétition avec tous les threads demandant le verrou!!!
- Le paramètre `time` permet de borner l'attente en temps. Nous ne l'utiliserons pas dans ce cours.

Fonctions : Signalisation

```
void QWaitCondition::wakeOne();
```

- réveille un des threads en attente sur la variable condition:
  - si aucun thread n'est en attente, la fonction n'a aucun effet ;
  - si plusieurs threads sont en attente, un seul est réveillé (choisi par l'ordonnanceur).

```
void QWaitCondition::wakeAll();
```

- réveille tous les threads en attente sur cond:
  - si aucun thread n'est en attente, la fonction n'a aucun effet ;
  - les threads réveillés continuent leur exécution chacun à leur tour, car le mutex ne peut être repris que par un thread à la fois (l'ordre est imprévisible et dépend de l'ordonnanceur).

Pour la culture: Fonctions Posix

Fonction	Description
`int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)`	Initialisation d'une variable condition
`int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`	Attente sur la variable
`int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime)`	Attente avec échéance
`pthread_cond_signal(pthread_cond_t *cond)`	Réveil potentiel d'un thread
`pthread_cond_broadcast(pthread_cond_t *cond)`	Réveil de tous les threads en attente

Moniteurs en C++/Qt

- Le concept de moniteur se prête bien à une implémentation OO

Définition de la classe

```
class MyMonitor {

protected:
  QMutex mutex;
  QWaitCondition cond;
  bool uneCondition;

public:
  MyMonitor() {};
  virtual ~MyMonitor() {};

void oneFunction();
void anotherFunction();
}
```

Structure type des points d'entrée d'un moniteur en Qt

Méthodes

```
void MyMonitor::oneFunction() {
  mutex.lock();

  // Evaluer uneCondition
  while (!uneCondition)
    cond.wait(&mutex);

  // Exécuter la fonction demandée

  mutex.unlock();
}


void MyMonitor::anotherFunction() {
  mutex.lock();

  // Faire quelque chose

  // Modifier la condition (passer à true)
  uneCondition = true;
  cond.wakeOne();

  mutex.unlock();
}
```

- Pourquoi une boucle **`while`**?

Exemple de producteurs/consommateurs en C++/Qt

```
class Buffer {
private:
    QMutex mutex;
    QWaitCondition isFree, isFull;
    char *buffer;

public:
    Buffer() {
      buffer = 0;
    }

    void put(char *msg);
    char *get(void);
};
```

Exemple de producteurs/consommateurs en C++/Qt

```
/* Depose le message msg (qui est dupliqué) et bloque tant que le
** tampon est plein. */
void Buffer::put(char *msg) {
    mutex.lock();
    while (buffer != NULL)
       isFree.wait(&mutex);
    if ((buffer = (char *)malloc(strlen(msg) + 1)) != NULL) {
       strcpy(buffer,msg);
       isFull.wakeOne();
    }
    mutex.unlock();
}

/* Renvoie le message du tampon et bloque tant que le tampon est vide.
** La libération de la mémoire contenant le message est à la charge de
** l'appelant. */
char *Buffer::get(void) {
    char *result;
    mutex.lock();
    while (buffer == NULL)
        isFull.wait(&mutex);
    result = buffer;
    buffer = NULL;
    mutex.unlock();
    isFree.wakeOne();
    return result;
}
```

Moniteur de Hoare à partir de sémaphores

- Si les variables de condition ne sont pas offertes

⇒ Possibilité de créer un moniteur à base de sémaphores

- En respectant la sémantique initiale de Hoare

⇒ Lorsqu'un thread signale une condition, si un thread est réveillé c'est lui qui obtient le droit de s'exécuter

⇒ Lorsqu'un thread quitte le moniteur il doit en priorité laisser l'accès au moniteur à un thread en attente suite à un signalement

Implémentation de moniteurs à partir de sémaphores (1)

Etape 1: Une classe étant exploitée comme un moniteur a besoin des éléments suivants :

```
typedef struct {
    QSemaphore mutex;
    QSemaphore signale;    // file bloquante des signaleurs
    unsigned   nbSignale; // tâches en attente dans signal
} T_Moniteur;

T_Moniteur mon;
```

Implémentation de moniteurs à partir de sémaphores (2)

Etape 2: Chaque procédure constituant un point d'entrée du moniteur est encadrée par :

```
mon.mutex.acquire();

// <code de la procédure>

if (mon.nbSignale > 0)
    mon.signale.release();
else
    mon.mutex.release();
```

Implémentation de moniteurs à partir de sémaphores (3)

Etape 3: Pour chaque variable condition `cond` du moniteur, créer un enregistrement:

```
typedef struct {
    QSemaphore attente;
    unsigned nbAttente; // tâches en attente
} T_Condition;

T_Condition cond;
```

Implémentation de moniteurs à partir de sémaphores (4)

Etape 4: Dans toutes les procédures du moniteur, substituer `cond.attente` par :

```
cond.nbAttente += 1;
if (mon.nbSignale > 0)
    // Avant de se mettre en attente, libérer un thread ayant émis
    // un signal
    mon.signale.release();
else
    // Avant de se mettre en attente, libérer le mutex pour laisser
    // un autre thread rentrer dans le moniteur
    mon.mutex.release();
cond.attente.acquire();
cond.nbAttente -= 1;
```

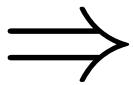
Implémentation de moniteurs à partir de sémaphores (5)

Etape 5: Dans toutes les procédures du moniteur, substituer `cond.signale` par :

```
if (cond.nbAttente > 0) {
    mon.nbSignale += 1;
    cond.attente.release();
    mon.signale.acquire(); // Pour laisser la priorité au thread réveillé
    mon.nbSignale -= 1;
}
```

Exemple: producteurs/consommateurs

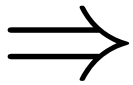
```
monitor Tampons
    var place: array [0..N-1] of ARTICLE;
    var head, tail, size: integer;
    var notFull, notEmpty: condition;
    begin size := 0; tail := 0; head := 0;
end Tampons;
```



```
class ProdConsSem {
  T_Moniteur mon;
  ARTICLE place[0..N-1];
  int head, tail, size;
  T_Condition notFull, notEmpty;
```

Exemple: producteurs/consommateurs

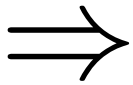
```
monitor Tampons
    entry procedure deposer(a: ARTICLE)
        begin
            if size = N then notFull.attente;
            size := size + 1;
            place[head] := a;
            head := (head + 1) mod N;
            notEmpty.signale;
        end deposer;
    end Tampons;
```



```
void deposer(ARTICLE a) {
  mon.mutex.acquire();
  if (size == N) {
     notFull.nbAttente += 1;
     if (mon.nbSignale > 0)
        mon.signale.release();
     else
        mon.mutex.release();
     notFull.attente.acquire();
     notFull.nbAttente -= 1;
  }
  size += 1;
  place[head] = a;
  head = (head + 1) % N;
  if (notEmpty.nbAttente > 0) {
     mon.nbSignale += 1;
     notEmpty.attente.release();
     mon.signale.acquire();
     mon.nbSignale -= 1;
  }
  if (mon.nbSignale > 0)
     mon.signale.release();
  else
     mon.mutex.release();
}
```

Exemple: producteurs/consommateurs

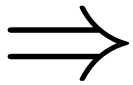
```
monitor Tampons
    entry procedure retirer(var a: ARTICLE)
        begin
            if size = 0 then notEmpty.attente;
            a := place[tail];
            size := size - 1;
            tail := (tail + 1) mod N;
            notFull.signale;
        end retirer;
    end Tampons;
```



```
void retirer(ARTICLE *a) {
  mon.mutex.acquire();
  if (size == 0) {
     notEmpty.nbAttente += 1;
     if (mon.nbSignale > 0)
        mon.signale.release();
     else
        mon.mutex.release();
     notEmpty.attente.acquire();
     notEmpty.nbAttente -= 1;
  }
  a = place[tail];
  size -= 1;
  tail = (tail + 1) % N;
  if (notFull.nbAttente > 0) {
     mon.nbSignale += 1;
     notFull.attente.release();
     mon.signale.acquire();
     mon.nbSignale -= 1;
  }
  if (mon.nbSignale > 0)
     mon.signale.release();
  else
     mon.mutex.release();
}
```

Optimisation (1)

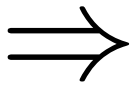
```
void deposer(ARTICLE a) {
  mon.mutex.acquire();
  if (size == N) {
     notFull.nbAttente += 1;
     if (mon.nbSignale > 0)
        mon.signale.release();
     else
        mon.mutex.release();
     notFull.attente.acquire();
     notFull.nbAttente -= 1;
  }
  size += 1;
  place[head] = a;
  head = (head + 1) % N;
  if (notEmpty.nbAttente > 0) {
     mon.nbSignale += 1;
     notEmpty.attente.release();
     mon.signale.acquire();
     mon.nbSignale -= 1;
  }
  if (mon.nbSignale > 0)
     mon.signale.release();
  else
     mon.mutex.release();
}
```



```
void deposer(ARTICLE a) {
  mon.mutex.acquire();
  if (size == N) {
     notFull.nbAttente += 1;
     if (mon.nbSignale > 0)
        mon.signale.release();
     else
        mon.mutex.release();
     notFull.attente.acquire();
     notFull.nbAttente -= 1;
  }
  size += 1;
  place[head] = a;
  head = (head + 1) % N;
  if (notEmpty.nbAttente > 0)
     notEmpty.attente.release();
  else if (mon.nbSignale > 0)
     mon.signale.release();
  else
     mon.mutex.release();
}
```

Optimisation (2)

```
void deposer(ARTICLE a) {
  mon.mutex.acquire();
  if (size == N) {
     notFull.nbAttente += 1;
     if (mon.nbSignale > 0)
        mon.signale.release();
     else
        mon.mutex.release();
     notFull.attente.acquire();
     notFull.nbAttente -= 1;
  }
  size += 1;
  place[head] = a;
  head = (head + 1) % N;
  if (notEmpty.nbAttente > 0)
     notEmpty.attente.release();
  else if (mon.nbSignale > 0)
     mon.signale.release();
  else
     mon.mutex.release();
}
```



```
void deposer(ARTICLE a) {
  mon.mutex.acquire();
  if (size == N) {
     notFull.nbAttente += 1;
     mon.mutex.release();
     notFull.attente.acquire();
     notFull.nbAttente -= 1;
  }
  size += 1;
  place[head] = a;
  head = (head + 1) % N;
  if (notEmpty.nbAttente > 0)
     notEmpty.attente.release();
  else
     mon.mutex.release();
}
```

Optimisation (3)→solution

```
void deposer(ARTICLE a) {
  mon.mutex.acquire();
  if (size == N) {
     notFull.nbAttente += 1;
     mon.mutex.release();
     notFull.attente.acquire();
     notFull.nbAttente -= 1;
  }
  size += 1;
  place[head] = a;
  head = (head + 1) % N;
  if (notEmpty.nbAttente > 0)
     notEmpty.attente.release();
  else
     mon.mutex.release();
}
```

```
void retirer(ARTICLE *a) {
  mon.mutex.acquire();
  if (size == 0) {
     notEmpty.nbAttente += 1;
     mon.mutex.release();
     notEmpty.attente.acquire();
     notEmpty.nbAttente -= 1;
  }
  a = place[tail];
  size -= 1;
  tail = (tail + 1) % N;
  if (notFull.nbAttente > 0)
     notFull.attente.release();
  else
     mon.mutex.release();
}
```

Moniteur de Hoare en C++

```
class HoareMonitor{
protected:
    class Condition
    {
        friend HoareMonitor;
    public:
        Condition();
    private:
        QSemaphore waitingSem;
        int nbWaiting;
    };
    HoareMonitor();
    void monitorIn();
    void monitorOut();
    void wait(Condition &cond);
    void signal(Condition &cond);

private:
    QSemaphore monitorMutex;
    QSemaphore monitorSignale;
    int monitorNbSignale;
};
```

Moniteur de Hoare en C++

```
HoareMonitor::Condition::Condition() : waitingSem(0), nbWaiting(0) {}

HoareMonitor::HoareMonitor() :
    monitorMutex(1), monitorSignale(0), monitorNbSignale(0) {}

void HoareMonitor::monitorIn() {
    monitorMutex.acquire();
}

void HoareMonitor::monitorOut() {
    if (monitorNbSignale > 0)
        monitorSignale.release();
    else
        monitorMutex.release();
}
```

Moniteur de Hoare en C++

```
void HoareMonitor::wait(Condition &cond) {
    cond.nbWaiting += 1;
    if (monitorNbSignale > 0)
        monitorSignale.release();
    else
        monitorMutex.release();
    cond.waitingSem.acquire();
    cond.nbWaiting -= 1;
}

void HoareMonitor::signal(Condition &cond) {
    if (cond.nbWaiting>0) {
        monitorNbSignale += 1;
        cond.waitingSem.release();
        monitorSignale.acquire();
        monitorNbSignale -= 1;
    }
}
```

Moniteur de Hoare en C++: Exemple d'utilisation

```
class ProdConsoHoare : public HoareMonitor {
  ARTICLE place[0..N-1];
  int head, tail, size;
  Condition notFull, notEmpty;
```

```
void deposer(ARTICLE a) {
  monitorIn();
  if (size == N) {
    wait(notFull);
  size += 1;
  place[head] = a;
  head = (head + 1) % N;
  signal(notEmpty);
  monitorOut();
}
```

```
void retirer(ARTICLE *a) {
  monitorIn();
  if (size == 0) {
    wait(&notEmpty);
  }
  a = place[tail];
  size -= 1;
  tail = (tail + 1) % N;
  signal(notFull);
  monitorOut();
}
```

Moniteurs en Java

- Le mécanisme de synchronisation natif à Java est le moniteur
- Dans une classe, les méthodes peuvent être déclarées `synchronized`
  - Le mot-clé `synchronized` garanti l'exclusion mutuelle sur ces méthodes
  - Identique à l'idée d'avoir un mutex dans la classe, verrouillé en début de méthode et relâché en fin

Exemple

```
class UniqueID {
    private int id;

    public synchronized int getID() {
        return id ++;
    }

    public synchronized void decrID() {
        id --;
    }
}
```

Moniteurs en Java

- La synchronisation est ensuite développée à l'aide de 3 méthodes:

- **`wait`**`()` : Suspend la tâche appelante jusqu'au réveil de l'objet courant
- `notify()` : Réveille une tâche bloquée sur un **`wait`**`()`
- `notifyAll()` : Réveille toutes les tâches bloquées sur un **`wait`**`()`

- Attention, l'ordre de réveil n'est pas défini, il peut être quelconque
- Et une tâche réveillée doit réacquérir l'exclusion mutuelle

Moniteurs en Java

Exemple: un sémaphore en Java

```
public class Semaphore {
    private int value; /* la valeur du semaphore */
    private int nbWait = 0; /* nb en attente */
    public Semaphore(int initVal) {
        value = initVal;
    }
    public synchronized void wait() {
        while (value <= 0) {
            nbWait++;
            wait();
        }
        value--;
    }
    public synchronized void post() {
        value++;
        if (nbWait > 0) {
            nbWait--;
            notify();
        }
    }
}
```

Attente

Réveil d'une tâche

Moniteurs en Java

Exemple: un sémaphore en Java (plus simple)

```
public class Semaphore {
    private int value; /* la valeur du semaphore */
    public Semaphore(int initVal) {
        value = initVal;
    }
    public synchronized void wait() {
        while (value <= 0) {
            wait();
        }
        value--;
    }
    public synchronized void post() {
        value++;
        notify();
    }
}
```

Attente

Réveil d'une tâche

Moniteurs en Java

- Il est également possible de ne synchroniser qu'une partie d'une méthode
- Pour avoir des files d'attente de type FIFO il faut effectuer leur gestion de manière explicite
- Java offre également (depuis Java 7):
  - Des `Lock` qui offrent le même type de fonctionnement que les verrous
  - Des `Condition` qui offrent des variables de conditions
- Il y a dès lors plus de flexibilité à disposition

Remarques finales

- Avantages des moniteurs
  - une protection associée au moniteur (exclusion mutuelle);
  - une souplesse d'utilisation des primitives *attente* et *signale*;
  - une efficacité de ces mécanismes.
- Inconvénients des moniteurs
  - un risque de manque de lisibilité qui est partiellement dû à des variations sémantiques des implémentations dans les divers langages qui les supportent.
    - Dans le cas de *pthread* ou de Qt, il n'y a aucune garantie que les variables partagées sont effectivement accédées uniquement depuis les points d'entrée du moniteur qui devrait les protéger;
  - les variables condition sont de bas niveau;
  - l'impossibilité d'imposer un ordre total ou partiel dans l'exécution des procédures ou fonctions exportées.

Code source

```
http://reds.heig-vd.ch/share/cours/PCO/cours/code/7-moniteurs/simpleMonitor.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/7-moniteurs/prodConsumerMonitor.tar.gz
```