Introduction à la programmation concurrente

Moniteurs

Yann Thoma, Fiorenzo Gamba

Février 2021

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

Introduction

Introduction

- Concept de moniteur
  - Proposé par Hoare en 1974

Introduction

- Concept de moniteur
  - Proposé par Hoare en 1974
- Abstraction plus haut niveau que les sémaphores pour la synchronisation:
  - Implémente le *pattern* très courant d'un état partagé et d'opérations concurrentes conditionnées par cette état
  - Assure l'exclusion mutuelle sur les procédures du moniteur

Principe

- Les procédures du moniteur s’exécutent en exclusion mutuelle
- La synchronisation se fait via des *variables de condition*
    - En fait, il s’agit en quelque sorte d’une déclaration de type "condition concurrente"

Principe

- Les procédures du moniteur s'exécutent en exclusion mutuelle
- La synchronisation se fait via des *variables de condition*
  - En fait, il s'agit en quelque sorte d'une déclaration de type "condition concurrente"
- 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?
  - ...

Principe

- Les procédures du moniteur s'exécutent en exclusion mutuelle
- La synchronisation se fait via des *variables de condition*
  - En fait, il s'agit en quelque sorte d'une déclaration de type "condition concurrente"
- 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 condition permet de faire attendre le thread avec `wait()`

Principe

- Les procédures du moniteur s'exécutent en exclusion mutuelle
- La synchronisation se fait via des *variables de condition*
  - En fait, il s'agit en quelque sorte d'une déclaration de type "condition concurrente"
- 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 condition permet de faire attendre le thread avec `wait()`
- Un autre thread pourra réveiller un ou plusieurs threads avec `signal()`

Principe

- Les procédures du moniteur s'exécutent en exclusion mutuelle
- La synchronisation se fait via des *variables de condition*
  - En fait, il s'agit en quelque sorte d'une déclaration de type "condition concurrente"
- 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 condition permet de faire attendre le thread avec `wait()`
- Un autre thread pourra réveiller un ou plusieurs threads avec `signal()`
- Les procédures d'attente et de signalisation du moniteur sont thread-safe

Structure formelle d'un moniteur selon Hoare

monitor <*nom*>
  <*variables d'état*>
  <*variables conditions*>

  entry procedure *P1*(*paramètres*)
    <*variables locales*>
    begin <*code P1*> **end**
  entry procedure *P2*(*paramètres*)
    <*variables locales*>
    begin <*code P2*> **end**

  begin <*initialisation des variables d'état*>
  end

Seulement **un seul thread** à la fois accepté dans le moniteur

Variable de condition (VC)

- Soit la VC *cond* déclarée par

  condition *cond*;

- *cond.wait*
  - bloque inconditionnellement la tâche appelante
  - lui fait relâcher l'exclusion mutuelle sur le moniteur
  - la place dans une file associée à *cond*
- *cond.signal*
  - 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

Exemple: Verrou par moniteur

```
monitor MutexMonitor
    var locked: boolean;
    var access: condition;

    entry procedure Lock
        begin
            if locked then access.wait;
            locked := true;
        end Lock;

    entry procedure Unlock
        begin
            locked := false;
            access.signal;
        end Unlock;

    begin
        locked := false;
    end MutexMonitor;
```

Exemple: Producteur-consommateur

```
monitor Buffer
    var place: array [0..N-1] of ARTICLE;
    var head, tail, nbElements: integer;
    var notFull, notEmpty: condition;
    entry procedure put(a: ARTICLE)
        begin
            if nbElements = N then notFull.wait;
            nbElements := nbElements + 1;
            place[head] := a;
            head := (head + 1) mod N;
            notEmpty.signal;
        end put;
    entry procedure get(var a: ARTICLE)
        begin
            if nbElements = 0 then notEmpty.wait;
            a := place[tail];
            nbElements := nbElements - 1;
            tail := (tail + 1) mod N;
            notFull.signal;
        end get;
    begin nbElements := 0; tail := 0; head := 0; end Buffer;
```

Type de moniteur

- Moniteur de type Hoare
  - Le thread qui est réveillé par `signal` prend possession du mutex
  - Pour C++, implémentation avec des sémaphores

Type de moniteur

- Moniteur de type Hoare
  - Le thread qui est réveillé par `signal` prend possession du mutex
  - Pour C++, implémentation avec des sémaphores
- Moniteur de type Mesa
  - Le thread qui appelle `signal` garde le mutex
  - C++11, Java, pthread, Qt

Moniteur Mesa en PCO

Moniteur PCO

- Un moniteur s'implémente à l'aide de:

Moniteur PCO

- Un moniteur s'implémente à l'aide de:
  - un mutex, qui assure l'exclusion mutuelle

Moniteur PCO

- Un moniteur s'implémente à l'aide de:
  - un mutex, qui assure l'exclusion mutuelle
  - une variable de condition de type `PcoConditionVariable`

Fonctions : Attente

```
void PcoConditionVariable::wait(PcoMutex * lockedMutex)
```

- Effectue les opérations suivantes de manière atomique:
  - Relâche le mutex `lockedMutex`
  - Attend que la variable de condition soit signalée.

Fonctions : Attente

```
void PcoConditionVariable::wait(PcoMutex * lockedMutex)
```

- 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.

Fonctions : Attente

```
void PcoConditionVariable::wait(PcoMutex * lockedMutex)
```

- 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()`.

Fonctions : Attente

```
void PcoConditionVariable::wait(PcoMutex * lockedMutex)
```

- 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!!!

Fonctions : Signalisation

```
void PcoConditionVariable::notifyOne();
```

- 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
    - Pas de liste d'attente FIFO

    $\Rightarrow$ Choisi par l'ordonnanceur

Fonctions : Signalisation

```
void PcoConditionVariable::notifyAll();
```

- Réveille tous les threads en attente sur la variable condition:
  - 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).

Moniteurs de Mesa

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

Définition de la classe

```
class MyMonitor {

protected:
    PcoMutex mutex;
    PcoConditionVariable cond;
    bool theCondition;

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

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

Structure type des points d'entrée d'un moniteur de Mesa

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

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

    // Faire quelque chose

    mutex.unlock();
}

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

    // Faire quelque chose

    // Modifier la condition (passer à true)
    theCondition = true;
    cond.notifyOne();

    mutex.unlock();
}
```

Structure type des points d'entrée d'un moniteur de Mesa

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

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

    // Faire quelque chose

    mutex.unlock();
}

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

    // Faire quelque chose

    // Modifier la condition (passer à true)
    theCondition = true;
    cond.notifyOne();

    mutex.unlock();
}
```

- Pourquoi une boucle `while`?

Exemple de producteurs/consommateurs

```
class Buffer {
private:
    PcoMutex mutex;
    PcoConditionVariable isFree, isFull;
    std::string buffer;
    bool bufferFull;

public:
    Buffer() : bufferFull(false) {}

    void put(std::string msg);
    std::string get(void);
};
```

Exemple de producteurs/consommateurs

```
// Depose le message msg (qui est dupliqué) et bloque tant que le tampon est plein.
void Buffer::put(std::string msg) {
    mutex.lock();
    while (bufferFull)
        isFree.wait(&mutex);
    buffer = msg;
    bufferFull = true;
    isFull.notifyOne();
    mutex.unlock();
}

// Renvoie le message du tampon et bloque tant que le tampon est vide.
std::string Buffer::get() {
    std::string result;
    mutex.lock();
    while (!bufferFull)
        isFull.wait(&mutex);
    result = buffer;
    bufferFull = false;
    mutex.unlock();
    isFree.notifyOne();
    return result;
}
```

Illustration d’un moniteur Mesa

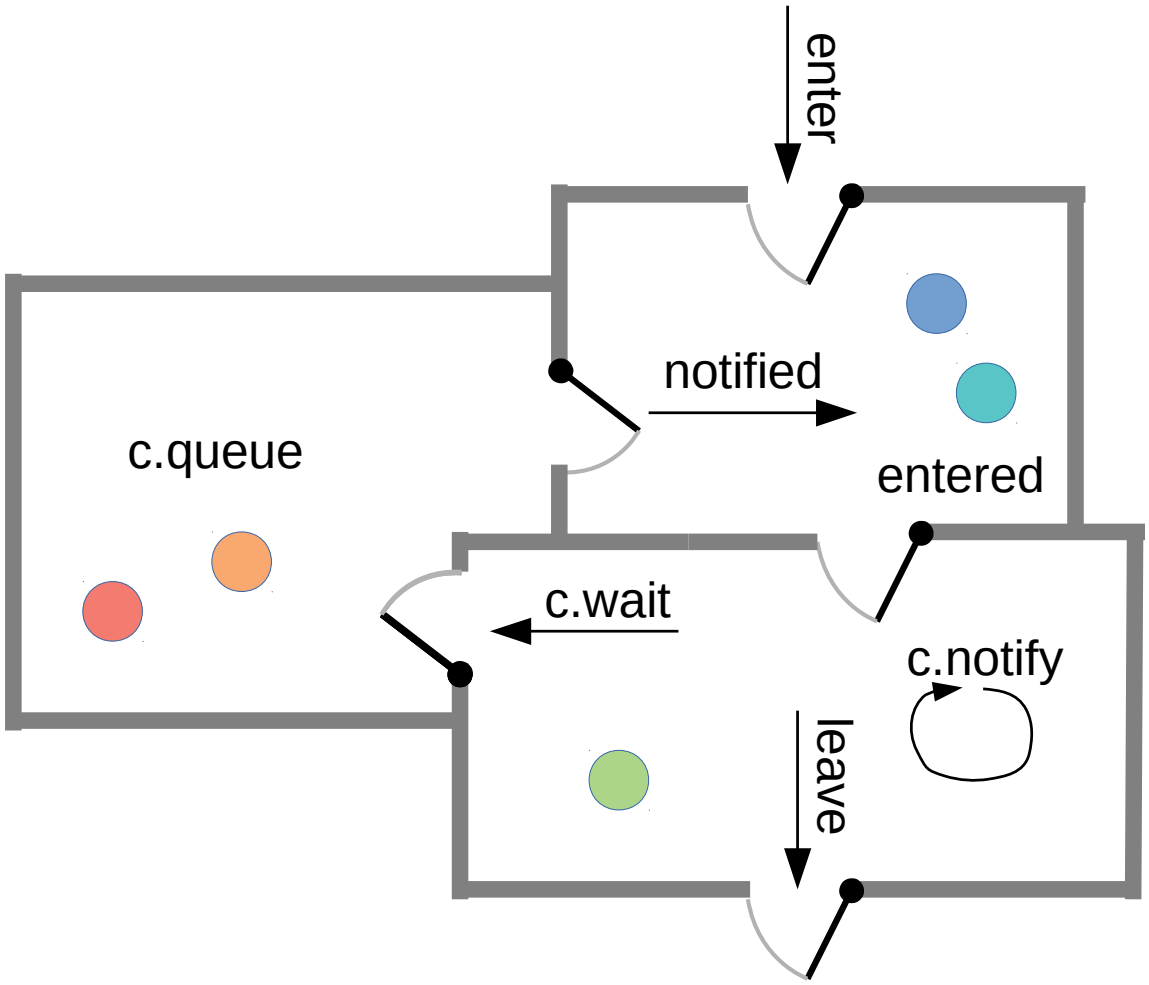
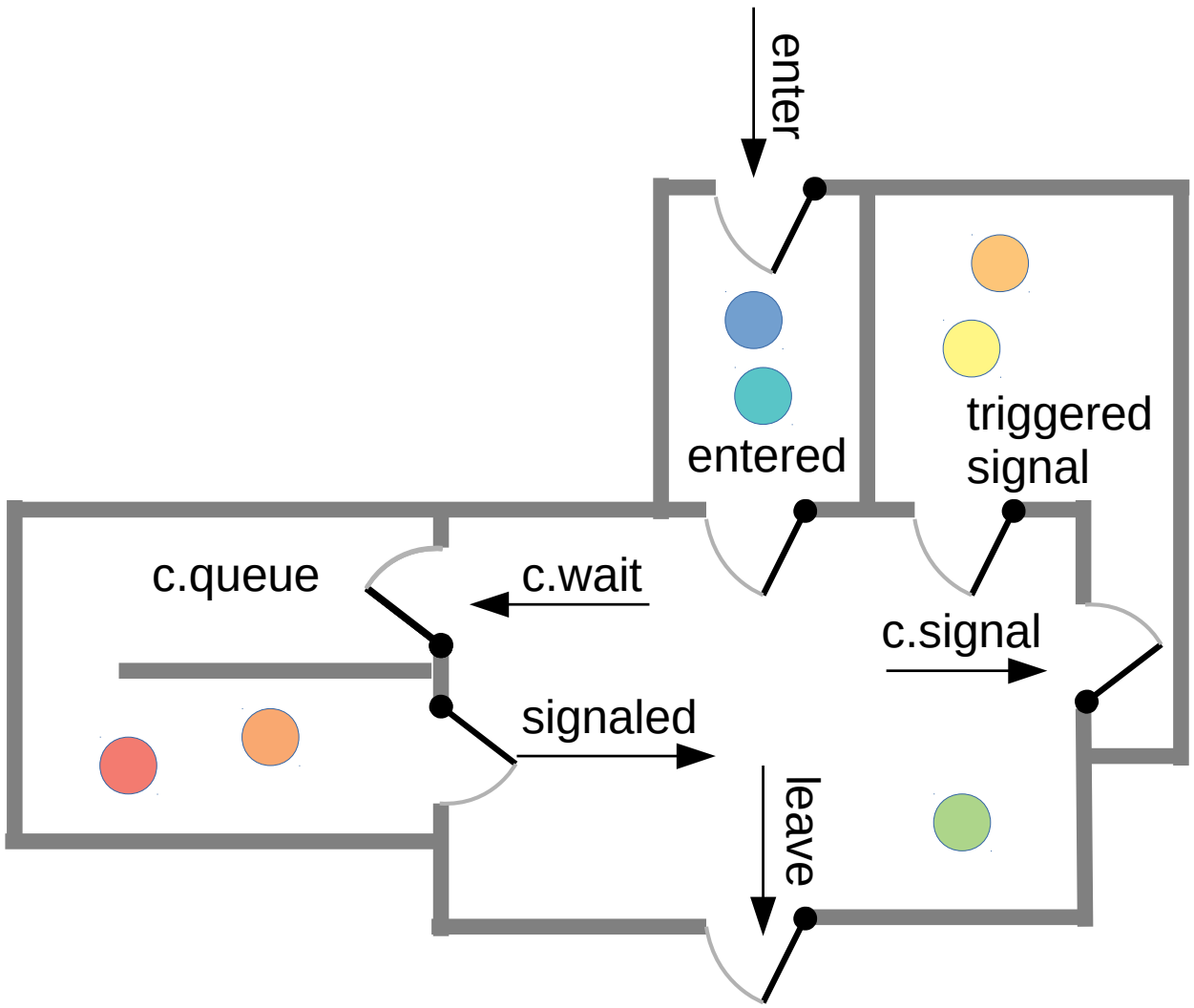


Illustration d'un moniteur Hoare



Moniteur de Hoare à base de sémaphores

Moniteur de Hoare à partir de sémaphores

- Si les variables de condition ne sont pas offertes

$\Rightarrow$ Possibilité de créer un moniteur à base de sémaphores

- En respectant la sémantique initiale de Hoare

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

$\Rightarrow$ 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 {
    PcoSemaphore mutex;
    PcoSemaphore signaled;   // file bloquante des signaleurs
    unsigned     nbSignaled; // nb 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.nbSignaled > 0)
    mon.signaled.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 {
    PcoSemaphore waitingSem;
    unsigned nbWaiting; // nb 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.wait` par :

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

Implémentation de moniteurs à partir de sémaphores (5)

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

```
if (cond.nbWaiting > 0) {
    mon.nbSignaled += 1;
    cond.waitingSem.release();
    // Pour laisser la priorité au thread réveillé
    mon.signaled.acquire();
    mon.nbSignaled -= 1;
}
```

Exemple: producteurs/consommateurs

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

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

Exemple: producteurs/consommateurs

```
monitor Buffer
    entry procedure put(a: ARTICLE)
        begin
            if nbElements = N then notFull.wait;
            nbElements := nbElements + 1;
            place[head] := a;
            head := (head + 1) mod N;
            notEmpty.signal;
        end put;
    end Buffer;
```

```
void put(ARTICLE a) {
  mon.mutex.acquire();
  if (nbElements == N) {
     notFull.nbWaiting += 1;
     if (mon.nbSignaled > 0)
        mon.signaled.release();
     else
        mon.mutex.release();
     notFull.waitingSem.acquire();
     notFull.nbWaiting -= 1;
  }
  nbElements += 1;
  place[head] = a;
  head = (head + 1) % N;
  if (notEmpty.nbWaiting > 0) {
     mon.nbSignaled += 1;
     notEmpty.attente.release();
     mon.signaled.acquire();
     mon.nbSignaled -= 1;
  }
  if (mon.nbSignaled > 0)
     mon.signaled.release();
  else
     mon.mutex.release();
}
```

Exemple: producteurs/consommateurs

```
monitor Buffer
    entry procedure get(var a: ARTICLE)
        begin
            if nbElements = 0 then notEmpty.wait;
            a := place[tail];
            nbElements := nbElements - 1;
            tail := (tail + 1) mod N;
            notFull.signal;
        end get;
    end Buffer;
```

```
void get(ARTICLE *a) {
  mon.mutex.acquire();
  if (nbElements == 0) {
     notEmpty.nbWaiting += 1;
     if (mon.nbSignaled > 0)
        mon.signaled.release();
     else
        mon.mutex.release();
     notEmpty.waitingSem.acquire();
     notEmpty.nbWaiting -= 1;
  }
  a = place[tail];
  nbElements -= 1;
  tail = (tail + 1) % N;
  if (notFull.nbWaiting > 0) {
     mon.nbSignaled += 1;
     notFull.attente.release();
     mon.signaled.acquire();
     mon.nbSignaled -= 1;
  }
  if (mon.nbSignaled > 0)
     mon.signaled.release();
  else
     mon.mutex.release();
}
```

Optimisation (1)

```
void put(ARTICLE a) {
  mon.mutex.acquire();
  if (nbElements == N) {
     notFull.nbWaiting += 1;
     if (mon.nbSignaled > 0)
        mon.signaled.release();
     else
        mon.mutex.release();
     notFull.waitingSem.acquire();
     notFull.nbWaiting -= 1;
  }
  nbElements += 1;
  place[head] = a;
  head = (head + 1) % N;
  if (notEmpty.nbWaiting > 0) {
     mon.nbSignaled += 1;
     notEmpty.attente.release();
     mon.signaled.acquire();
     mon.nbSignaled -= 1;
  }
  if (mon.nbSignaled > 0)
     mon.signaled.release();
  else
     mon.mutex.release();
}
```

Optimisation (1)

```
void put(ARTICLE a) {
  mon.mutex.acquire();
  if (nbElements == N) {
     notFull.nbWaiting += 1;
     if (mon.nbSignaled > 0)
        mon.signaled.release();
     else
        mon.mutex.release();
     notFull.waitingSem.acquire();
     notFull.nbWaiting -= 1;
  }
  nbElements += 1;
  place[head] = a;
  head = (head + 1) % N;
  if (notEmpty.nbWaiting > 0) {
     mon.nbSignaled += 1;
     notEmpty.attente.release();
     mon.signaled.acquire();
     mon.nbSignaled -= 1;
  }
  if (mon.nbSignaled > 0)
     mon.signaled.release();
  else
     mon.mutex.release();
}
```

```
void put(ARTICLE a) {
  mon.mutex.acquire();
  if (nbElements == N) {
     notFull.nbWaiting += 1;
     if (mon.nbSignaled > 0)
        mon.signaled.release();
     else
        mon.mutex.release();
     notFull.waitingSem.acquire();
     notFull.nbWaiting -= 1;
  }
  nbElements += 1;
  place[head] = a;
  head = (head + 1) % N;
  if (notEmpty.nbWaiting > 0)
     notEmpty.attente.release();
  else if (mon.nbSignaled > 0)
     mon.signaled.release();
  else
     mon.mutex.release();
}
```

Optimisation (2)

```
void put(ARTICLE a) {
  mon.mutex.acquire();
  if (nbElements == N) {
     notFull.nbWaiting += 1;
     if (mon.nbSignaled > 0)
        mon.signaled.release();
     else
        mon.mutex.release();
     notFull.waitingSem.acquire();
     notFull.nbWaiting -= 1;
  }
  nbElements += 1;
  place[head] = a;
  head = (head + 1) % N;
  if (notEmpty.nbWaiting > 0)
     notEmpty.attente.release();
  else if (mon.nbSignaled > 0)
     mon.signaled.release();
  else
     mon.mutex.release();
}
```

Optimisation (2)

```
void put(ARTICLE a) {
  mon.mutex.acquire();
  if (nbElements == N) {
     notFull.nbWaiting += 1;
     if (mon.nbSignaled > 0)
        mon.signaled.release();
     else
        mon.mutex.release();
     notFull.waitingSem.acquire();
     notFull.nbWaiting -= 1;
  }
  nbElements += 1;
  place[head] = a;
  head = (head + 1) % N;
  if (notEmpty.nbWaiting > 0)
     notEmpty.attente.release();
  else if (mon.nbSignaled > 0)
     mon.signaled.release();
  else
     mon.mutex.release();
}
```

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

Optimisation (3)→solution

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

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

Moniteur de Hoare en C++

Moniteur de Hoare en C++

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

private:
    PcoSemaphore monitorMutex;
    PcoSemaphore monitorSignaled;
    int monitorNbSignaled;
};
```

Moniteur de Hoare en C++ - Implémentation (1)

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

PcoHoareMonitor::PcoHoareMonitor() :
    monitorMutex(1), monitorSignaled(0), monitorNbSignaled(0) {}

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

void PcoHoareMonitor::monitorOut() {
    if (monitorNbSignaled > 0)
        monitorSignaled.release();
    else
        monitorMutex.release();
}
```

Moniteur de Hoare en C++ - Implémentation (2)

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

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

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

```
class ProdConsoHoare : public PcoHoareMonitor {
  ITEM items[0..N-1];
  int head, tail, nbElements;
  Condition notFull, notEmpty;
```

```
void put(ITEM item) {
  monitorIn();
  if (nbElements == N) {
    wait(notFull);
  nbElements += 1;
  items[head] = item;
  head = (head + 1) % N;
  signal(notEmpty);
  monitorOut();
}
```

```
void get(ITEM *item) {
  monitorIn();
  if (nbElements == 0) {
     wait(notEmpty);
  }
  item = items[tail];
  nbElements -= 1;
  tail = (tail + 1) % N;
  signal(notFull);
  monitorOut();
}
```

Moniteurs en Java

Moniteurs en Java

- Le mécanisme de synchronisation natif à Java est le moniteur

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

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:

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

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()`

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()`

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

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 (faible) en Java

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

Attente

Réveil d'une tâche

Moniteurs en Java

- Il est également possible de ne synchroniser qu'une partie d'une méthode

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

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:
  - Des `Lock` qui offrent le même type de fonctionnement que les verrous
  - Des `Condition` qui offrent des variables de conditions

Remarques finales

- Avantages des moniteurs
  - Protection associée au moniteur (exclusion mutuelle);
  - Moins de code que solution pure sémaphores
  - Classique et disponible presque partout
- Mais... Encore assez bas niveau
  - Variations sémantiques des implémentations dans les divers langages qui les supportent.
    - Dans le cas de *pthread* ou de C++11, il n'y a aucune garantie que les variables partagées soient effectivement accédées uniquement depuis les points d'entrée du moniteur qui devraient les protéger;
    - Détail d'implémentation non-explicite qui rend l'utilisation d'une boucle while nécessaire

Code source

```
run:../code/7-moniteurs/simpleMonitor
run:../code/7-moniteurs/simpleMonitor
run:../code/7-moniteurs/prodConsumerMonitor
```