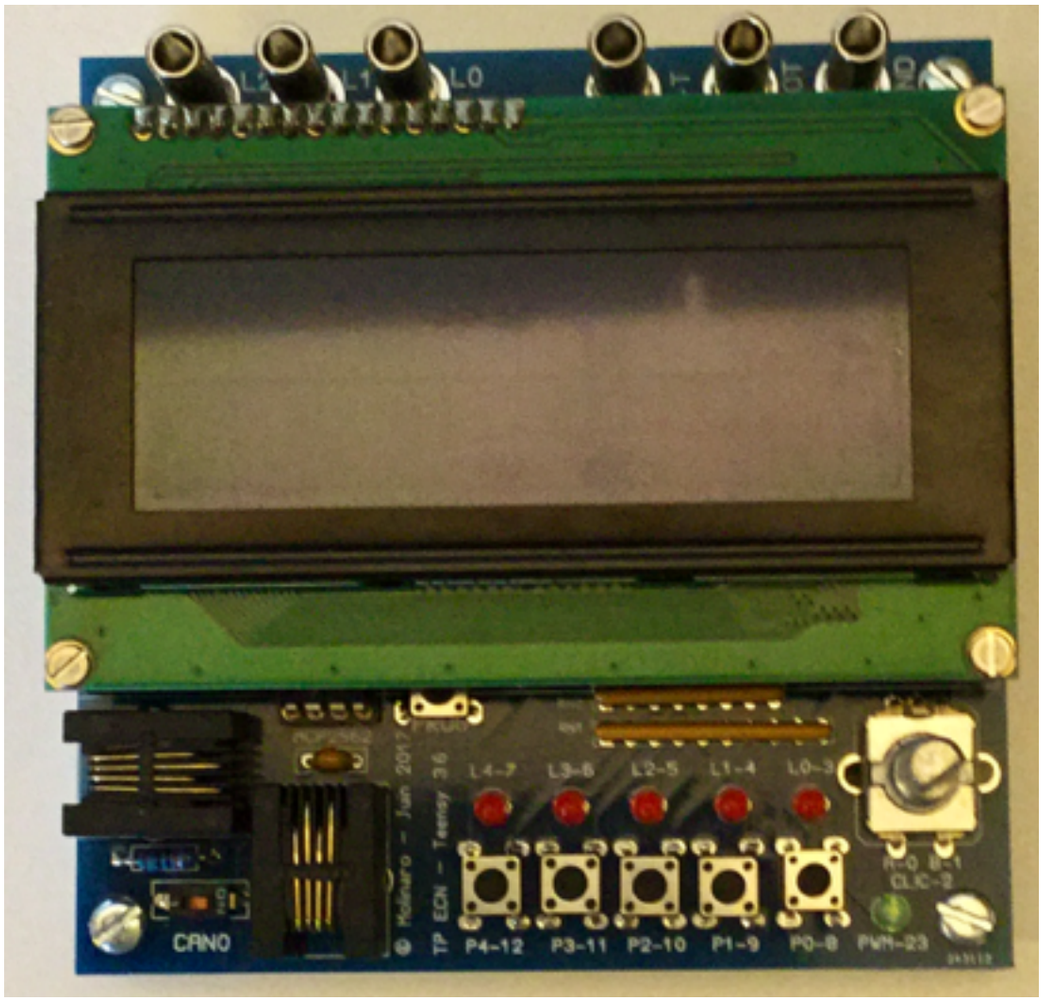
Temps Réel



Programme *18-guarded-commands*

Pierre Molinaro

Juin 2018

Description de cette étape

L'attente temporisée permet d'exprimer une attente sur le premier événement qui se déclenche, soit le passage par une primitive pouvant être bloquante, soit lorsqu'une date est atteinte.

Les commandes gardées sont une généralisation qui permet d'exprimer une attente sur des événements quelconques.

Imaginées par Dijkstra, les commandes gardées ont été étendues par Hoare pour le langage *Communicating Sequential Processes* avec des commandes de communication basées sur le rendez-vous. Dans la suite, nous décrivons comment toute commande susceptible d'être bloquante peut apparaître en garde (exemple typique : la primitive **P** d'un sémaphore).

Liens :

https://en.wikipedia.org/wiki/Guarded_Command_Language

https://en.wikipedia.org/wiki/Communicating_sequential_processes

Les commandes gardées de Dijkstra

Commande gardée. Une commande gardée a pour syntaxe :

```
expression_booléenne -> liste d'instructions
```

L'expression booléenne est appelée *garde*. La liste d'instructions qui suit est dite *associée à cette garde*.

État d'une garde. À un moment donné de l'exécution, une garde est fausse ou vraie, selon la valeur de son expression booléenne.

Commande répétitive. Une *commande répétitive* est constituée d'une ou plusieurs commandes gardées. Son exécution est la suivante :

- si une ou plusieurs gardes sont vraies, une garde est choisie *au hasard*, sa liste d'instructions est exécutée, puis la commande répétitive est exécutée de nouveau ;
- si toutes les gardes sont fausses, la commande répétitive est terminée, l'exécution se poursuit par celle de l'instruction suivante.

Voici un exemple de commande répétitive, qui trie x1, x2 et x3 par ordre croissant :

```
*[ x1 > x2 -> x1, x2 := x2, x1 // Échange x1, x2
 | x2 > x3 -> x2, x3 := x3, x2 // Échange x2, x3
 ]
```

Commande alternative. Une *commande alternative* est constituée d'une ou plusieurs commandes gardées. Son exécution est la suivante :

- si une ou plusieurs gardes sont vraies, une garde est choisie *au hasard*, sa liste d'instructions est exécutée, puis l'exécution se poursuit par celle de l'instruction qui suit la commande alternative ;
- le cas où toutes les gardes sont fausses est considéré comme une erreur d'exécution.

Garde avec une primitive de synchronisation

Garde avec une primitive de synchronisation. La syntaxe d'une garde est étendue, en acceptant la présence d'une primitive de synchronisation susceptible d'être bloquante :

```
expression_booléenne ; synchro -> liste d'instructions
```

État d'une garde. À un moment donné de l'exécution :

- si l'expression booléenne est fausse, la garde est **fausse** ;
- si l'expression booléenne est vraie :
  - si la primitive de synchronisation n'est pas bloquante, la garde **vraie** ;
  - si la primitive de synchronisation est bloquante, la garde est **neutre**.

Commande répétitive. Une *commande répétitive* est constituée d'une ou plusieurs commandes gardées. Son exécution est la suivante :

- si une ou plusieurs gardes sont vraies, une garde est choisie *au hasard*, sa liste d'instructions est exécutée, puis la commande répétitive est exécutée de nouveau ;
- si toutes les gardes sont fausses, la commande répétitive est terminée, l'exécution se poursuit par celle de l'instruction suivante ;
- si aucune garde n'est vraie, et il y a une plusieurs gardes neutres, l'exécution est suspendue en attendant qu'une ou plusieurs gardes deviennent vraies, ou qu'elles deviennent toutes fausses.

Commande alternative. Son fonctionnement est étendu de façon analogue.

Résumé : les trois formes d'une garde

Garde booléenne pure.

```
expression_booléenne -> liste d'instructions
```

Garde constituée d'une expression booléenne et une primitive de synchronisation.

```
expression_booléenne ; synchro -> liste d'instructions
```

Garde constituée d'une primitive de synchronisation.

```
synchro -> liste d'instructions
```

Cette forme est sémantiquement équivalente à la deuxième forme dans laquelle l'expression booléenne est toujours vraie.

L'expression booléenne ne peut être constituée que de variables privées de la tâche. Il y a plusieurs raisons à cela :

- cette règle ne restreint pas l'expressivité des commandes gardées, et en facilite même la compréhension, puisque les gardes ne peuvent évoluer que par l'action de la tâche courante ;
- implémenter les commandes gardées avec des variables susceptibles d'être modifiées par d'autres tâches est très compliqué (voir les implémentations de Ada…)

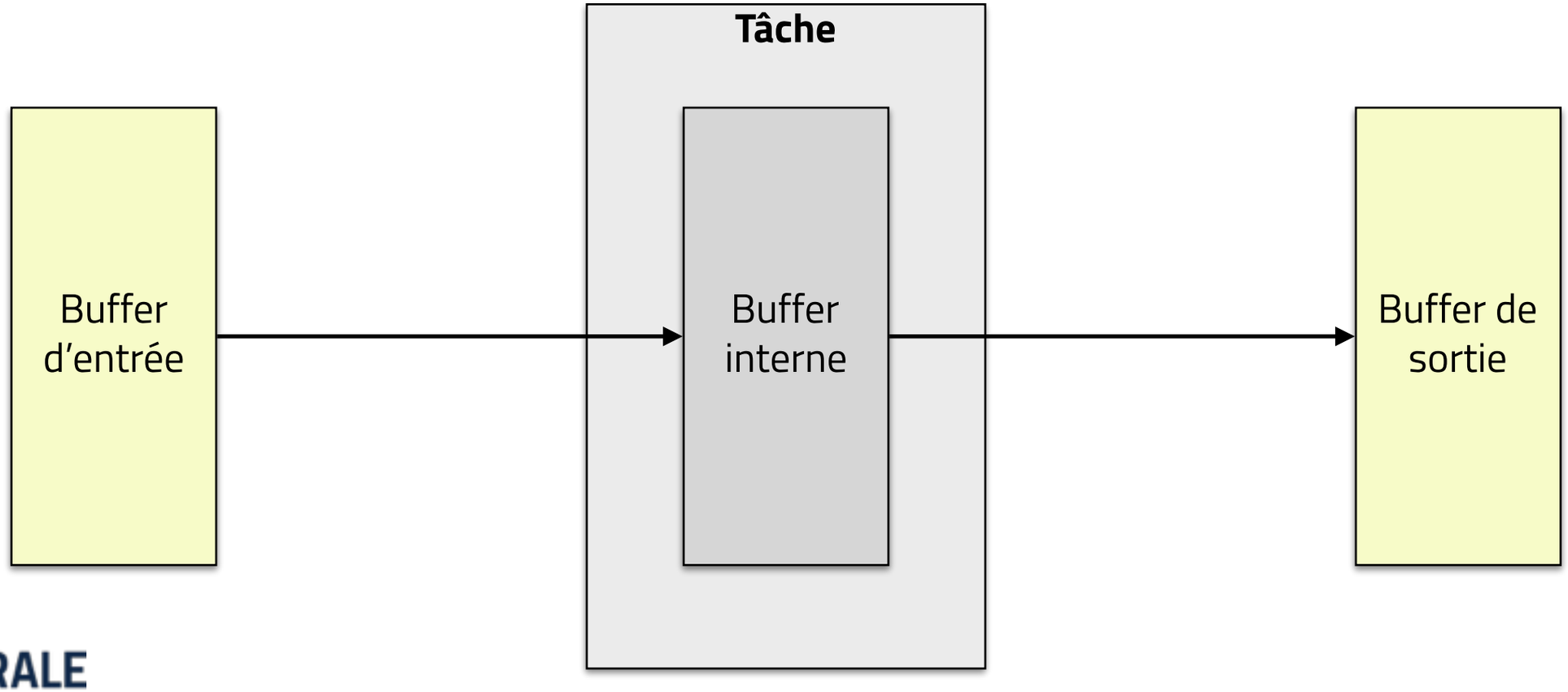
Un exemple

Une tâche reçoit des données en provenance d'un *buffer d'entrée*, les stocke dans son *buffer interne*, pour les retransmettre dans le *buffer de sortie*.

Les buffers d'entrée et de sortie sont des outils de synchronisation, le buffer interne est une simple structure de données.

Ainsi, la tâche réalise répétitivement :

- si le buffer interne n'est pas plein et que le buffer d'entrée contient une donnée, effectuer le transfert ;
- si le buffer interne n'est pas vide et que le buffer de sortie n'est pas plein, effectuer le transfert.



CENTRALE NANTES

Écriture des commandes gardées en C++

Garde booléenne pure.

`if (guard_booleanExpression (MODE_ expression_booléenne)) { liste d'instructions }`

Une garde booléenne pure a un effet de bord, aussi il faut appeler la fonction dédiée **guard_booleanExpression** qui fait les opérations nécessaires.

Garde constituée d'une expression booléenne et une primitive de synchronisation.

`if (expression_booléenne && outil.guarded_op (MODE)) { liste d'instructions }`

Attention, il faut toujours utiliser l'opérateur &&, qui n'évalue pas l'expression de droite si celle de gauche est vraie.

Garde constituée d'une primitive de synchronisation.

`if (outil.guarded_op (MODE)) { liste d'instructions }`

État d'une garde. L'expression d'un **if** ne pouvant prendre que deux valeurs (**true** ou **false**), on peut pas rendre compte des trois états possibles (*faux*, *neutre* ou *vrai*) d'une garde. Aussi :

- `outil.guarded_op(MODE)` renvoie le booléen **true** si la commande gardée est *vraie* ;
- `outil.guarded_op(MODE)` renvoie le booléen **false** si la commande gardée est *neutre* ;
- l'expression d'un **if** est **true** si la garde est vraie, et **false** si elle est *neutre* ou *fausse*.

CENTRALE NANTES

Écriture d'une commande répétitive en C++

```
bool loop = true ;
while (loop) {
  if (bufferInterneNonPlein && bufferEntrée.guarded_out (MODE_ data)) {
    Entrer data dans le buffer interne
  }else if (bufferInterneNonVide && bufferSortie.guarded_in (MODE_ dataBufferInterne)) {
    Retirer la donnée envoyée du buffer interne
  }else{
    loop = guard_waitForChange (MODE) ;
  }
}
```

Noter que l'ordre d'examen des commandes gardées est déterministe et figé.

Si une garde est *vraie*, la liste d'instructions associée est exécutée, et la commande répétitive est exécutée de nouveau.

L'exécution parvient à la branche **else** si toutes les expressions booléennes sont toutes **false**, ce qui correspond à des gardes *neutres* ou *fausses*. La primitive **guard_waitForChange** exploite les effets de bord de l'appel aux primitives **garded_xxx** :

- si toutes les gardes sont *fausses*, la primitive n'est pas bloquante et renvoie **false** (ce qui fait quitter la commande répétitive) ;
- sinon la primitive est bloquante, et sera débloquée en renvoyant **true** dès que l'état d'une primitive **garded_xxx** a changé.

CENTRALE NANTES

Écriture d’une commande alternative en C++

```
bool loop = true ;
while (loop) {
  if (une_commande_gardée) {
    liste d’instructions
    loop = false ;
  }else if (autre_commande_gardée) {
    liste d’instructions
    loop = false ;
  }else{
    loop = guard_waitForChange (MODE) ;
    if (!loop) {
      Erreur, toutes les gardes sont fausses
    }
  }
}
```

Le code ci-dessus fait apparaître deux modifications par rapport à une commande répétitive :

- après chaque *liste d’instructions associée*, le booléen `loop` est mis à **false** pour sortir de la boucle ;
- évaluer toutes les gardes à l’état *faux* constitue une erreur.

Attente temporelle en garde : `guard_waitUntil`

```
bool guard_waitUntil (USER_MODE_ const uint32_t inDeadlineMS) ;
```

Elle renvoie :

- **true** si l'échéance est atteinte (garde *vraie*) ;
- **false** si l'échéance n'est pas atteinte (garde *neutre*).

La primitive **guard_waitUntil** est une commande de synchronisation particulière, elle peut être précédée d'une expression booléenne.

Noter que l'argument est une date et non pas un délai. En effet, utiliser un délai provoquerait des comportements imprévisibles (voir l'exemple de la page suivante).

Nouvelle écriture de `P_until`

La primitive **P_until** peut-être maintenant vue comme un cas particulier : c'est une commande alternative constituée de deux gardes :

```
bool loop = true ;
bool response = false ;
while (loop) {
  if (sémaphore.guarded_P (MODE)) {
    loop = false ;
    response = true ;
  }else if (guard_waitUntil (MODE_ échéance)) {
    loop = false ;
  }else{
    guard_waitForChange (MODE) ; // Inutile de tester la valeur retournée
  }
}
```

Pourquoi il n'existe pas d'attente de délai en garde. Les gardes peuvent être évaluées un nombre imprévisible de fois, à des dates imprévisibles : l'échéance effective serait recalculée à chaque fois et serait « fuyante ».

Écriture d’une primitive apparaissant en garde

```
bool loop = true ;
while (loop) {
  if (une_commande_gardée) {
    liste d’instructions
    loop = false ;
  }else if (autre_commande_gardée) {
    liste d’instructions
    loop = false ;
  }else{
    loop = guard_waitForChange (MODE) ;
    if (!loop) {
      Erreur, toutes les gardes sont fausses
    }
  }
}
```

Ajouter la prise en compte des commandes gardées dans un outil de synchronisation nécessite l’utilisation d’une classe complémentaire et de deux fonctions.

La classe GuardList.

Exemple : le sémaphore et guarded_P

```
bool loop = true ;
while (loop) {
  if (une_commande_gardée) {
    liste d'instructions
    loop = false ;
  }else if (autre_commande_gardée) {
    liste d'instructions
    loop = false ;
  }else{
    loop = guard_waitForChange (MODE) ;
    if (!loop) {
      Erreur, toutes les gardes sont fausses
    }
  }
}
```

Le code ci-dessus fait apparaître deux modifications par rapport à une commande répétitive :

- après chaque *liste d'instructions associée*, le booléen `loop` est mis à **false** pour sortir de la boucle ;
- évaluer toutes les gardes à l'état *faux* constitue une erreur.

Travail à faire

Dupliquer le projet de l'étape précédente et renommez-le **18-guarded-commands**.

Il y a trop de modifications à faire pour les présenter simplement. Prendre l'archive **18-files.tar.bz2** qui contient les nouvelles versions des fichiers :

- **task-list--32-tasks.h** et **task-list--32-tasks.cpp** ;
- **time.h** et **time.cpp** ;
- **xtr.h** et **xtr.cpp**.

Expérimenter les commandes gardées en mettant en œuvre une chaîne de transmission de données à travers plusieurs tâches. Les données sont produites par une routine d'interruption temps-réel.