13. Processus et échanges de messages

INF7235

Hiver 2017

Aperçu

Aperçu

Les deux principaux paradigmes de programmation concurrente

1. Communication par variables partagées

⇒ les ***threads*** partagent un espace commun

2. Communication par échange de messages

⇒ chaque **processus** possède sa propre mémoire, privée, inaccessible aux autres processus

Notion clé = **Canal** de communication

= lien de communication entre deux ou plusieurs processus

Il y a différents styles de programmation par échange de messages. . .

Définition implicite vs. explicite des canaux

Canaux implicites

Les canaux sont implicites, par ex., à un processus est implicitement associé un ou plusieurs canaux (e.g., MPI)

Canaux explicites

Les canaux sont explicites $=$ objets «de première classe»

Il y a différents styles de programmation par échange de messages. . .

Définition statique vs. dynamique des canaux

Canaux statiques

Un canal est créé de façon statique et crée un lien direct entre deux processus

Canaux dynamiques

De nouveaux canaux peuvent être créés en cours d'exécution et peuvent lier différents processus

Il y a différents styles de programmation par échange de messages. . .

Communication uni-directionnelle vs. bi-directionnelle

Canaux unidirectionnels

Transmission d'un émetteur vers un récepteur

Canaux bidirectionnels

Échange symétrique d'information entre processus

Il y a différents styles de programmation par échange de messages...

Envoi synchrone vs. asynchrone

Envoi synchrone

Les deux processus doivent être prêts à communiquer pour que l'échange ait lieu, sinon le premier attend

Envoi asynchrone

L'envoyeur peut envoyer, même si le récepteur n'est pas prêt à recevoir

Il y a différents styles de programmation par échange de messages...

Envoi synchrone vs. asynchrone

Envoi synchrone

Les deux processus doivent être prêts à communiquer pour que l'échange ait lieu, sinon le premier attend
$\Rightarrow$ Pas besoin de *buffer* pour conserver les messages

Envoi asynchrone

L'envoyeur peut envoyer, même si le récepteur n'est pas prêt à recevoir
$\Rightarrow$ Canal de communication $\approx$ *buffer* des messages

Il y a différentes sortes de canaux

Selon les langages, selon l'utilisation qu'on en fait, etc.

Boite aux lettres

Canal sur lequel n'importe quel processus peut envoyer des messages ou à partir du quel n'importe quel processus peut en recevoir

Port d'entrée (*input port*)

Canal sur lequel n'importe quel processus peut envoyer des messages mais un seul processus en reçoit

Lien entre processus (*link*)

Canal utilisé par un seul émetteur et un seul récepteur

Il y a différentes sortes de canaux

Mais certaines caractéristiques sont communes à toutes les formes/sortes de canaux

Soit `c` un canal de communication

Écritures multiples par un même processus P

Soit P qui exécute :

P: `c.send` m_1; `c.send` m_2

Alors : m_1 sera reçu avant m_2

Il y a différentes sortes de canaux

Mais certaines caractéristiques sont communes à toutes les formes/sortes de canaux

Soit `c` un canal de communication

Écritures multiples par un même processus P

Soit P qui exécute :

P : `c.send` m_1 ; `c.send` m_2

Alors : m_1 sera reçu avant m_2

Écritures multiples par des processus distincts P_1 et P_2

Soit P_1 et P_2 qui exécutent de façon concurrente :

P_1 : `c.send` m_1
P_2 : `c.send` m_2

Alors : l'ordre de réception de m_1 et m_2 est indéterminé!

Aperçu

2.1 La classe `PRuby::Channel`

Les principales méthodes de `PRuby::Channel`

Instance Method Summary

- `(self) close`

  Indique la fermeture d'un canal en lui transmettant la valeur speciale :EOS.

- `(Object) each(&block)`

  Permet d'executer un bloc pour chacun des elements obtenus d'un canal.

- `(Bool) empty?`

  Determine si le canal est presentement vide.

- `(Bool) eos?`

  Determine si la fin du flux a ete rencontree.

- `(Bool) full?`

  Determine si le canal est presentement plein.

- `(Object) get`

  Obtient l'element en tete du canal.

- `(Channel) initialize(name = nil, max_size = 0, contents = [])` constructor

  Constructeur de base.

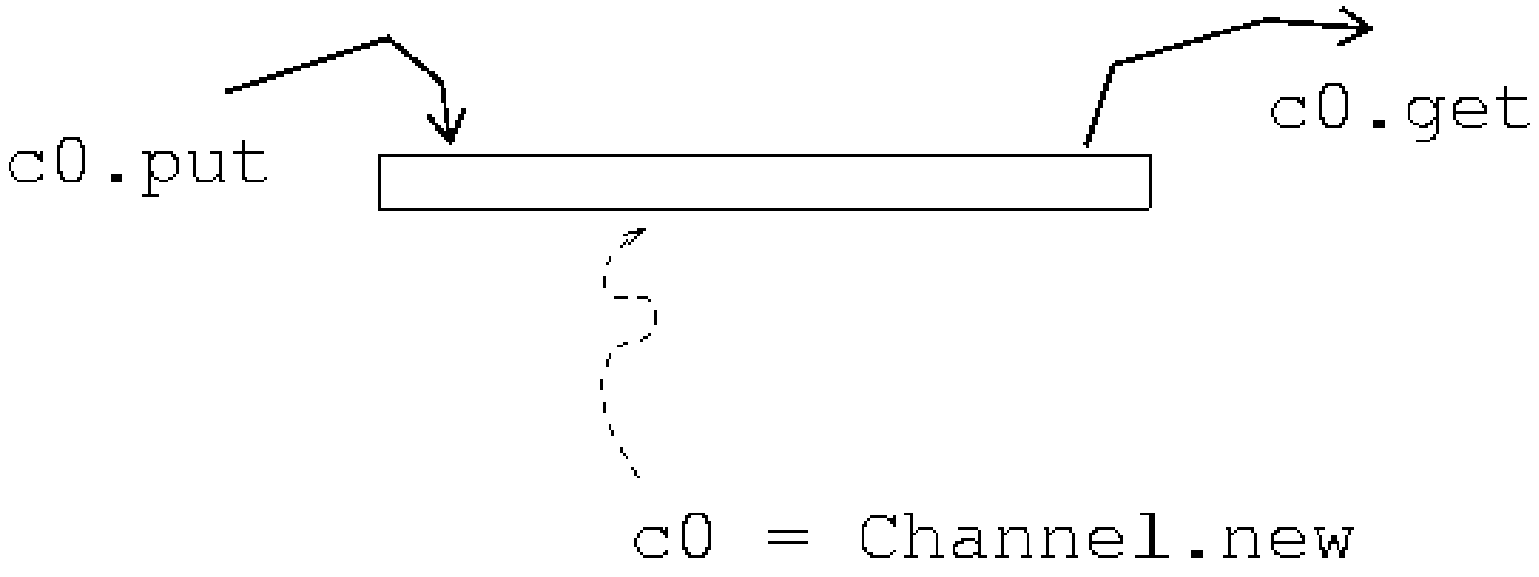
- `(Object) peek`

  Lit l'element en tete du canal, mais sans le retirer du canal.

- `(self) put(elem) (also: #<<)`

  Ajoute un element a la queue du canal.

Les principales méthodes de `PRuby::Channel`



Les principales méthodes de `PRuby::Channel`

Les méthodes de base

- `new` : Création d'un canal
- `put` : Envoi non bloquant — alias = «<<»
- `get` : Réception bloquante
- `close` : Fermeture du canal

Les principales méthodes de `PRuby::Channel`

Les méthodes de base

- `new` : Création d'un canal

  Par défaut ⇒ tampon non borné

- `put` : Envoi non bloquant — alias = «<<»

  Transmet `PRuby::EOS` pour indiquer la fermeture du flux

- `get` : Réception bloquante

  Retourne `PRuby::EOS` de façon persistente, dès que reçu

- `close` : Fermeture du canal

  Les appels à `put` ne sont plus possibles, mais les éléments encore présents seront obtenus par `get`

Les principales méthodes de `PRuby::Channel`

La méthodes `each` pour itérer sur les éléments d'un canal

```
class Channel
  def put( elem ); ...; end
  def get; ...; end
  def close; ...; end

  # Execute un bloc sur chaque element obtenu du canal.
  #
  # Termine quand EOS est rencontree.
  # Note: EOS n'est pas transmis au bloc.
  #
  def each
    while (v = get) != PRuby::EOS  # Bloquant!
      yield v
    end
  end

end
```

2.2 Une extension de la classe `Proc` pour des processus simples

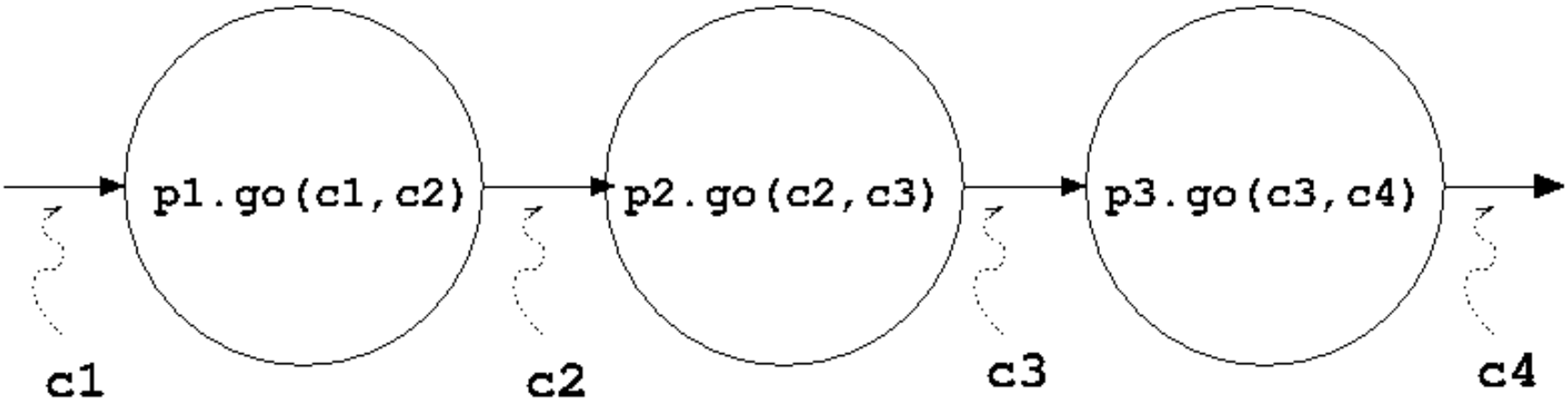
Extension de la classe `Proc`, classe des `lambda`s

```
class Proc
  def go( *canaux )
    Thread.new { call *canaux }
  end
end
```

Aperçu

3.1 Un petit pipeline avec trois processus

Représentation graphique du pipeline (linéaire) avec trois processus et quatre canaux



Note : C'est le programme (*thread*) principal qui crée les canaux, lance les processus, et amorce le traitement — en écrivant dans `c1` — puis imprime le résultat final — en lisant `c4`

Un petit pipeline

Les trois (3) processus sous forme de `lambda` (objets `Proc`)

```
p1 = lambda do |cin, cout|
  n = cin.get
  (1..n).each { |i| cout << i }
  cout.close
end

p2 = lambda do |cin, cout|
  cin.each { |v| cout << 10 * v }
  cout.close
end

p3 = lambda do |cin, cout|
  r = 0
  cin.each { |v| r += v }
  cout << r
  cout.close
end
```

Un petit pipeline

Le «programme principal» qui crée quatre (4) canaux et active les processus

```
# Creation des canaux.
c1, c2, c3, c4 = Array.new(4) { PRuby::Channel.new }

# Activation des processus.
p1.go( c1, c2 )
p2.go( c2, c3 )
p3.go( c3, c4 )

# Ecriture initiale => amorce le flux des donnees.
c1 << 10

# Reception du resultat.
puts c4.get # => 550
```

3.2 Trois façons de calculer le mininum et le maximum d'une série de valeurs

Le problème : Calcul distribué du `min` et du `max`

- On a `n` processus.
- Au départ :
  Chaque processus possède une valeur initiale privée.
- À la fin :
  Tous les processus doivent connaître les valeurs minimum et maximum parmi toutes les valeurs.

Le problème : Calcul distribué du `min` et du `max`

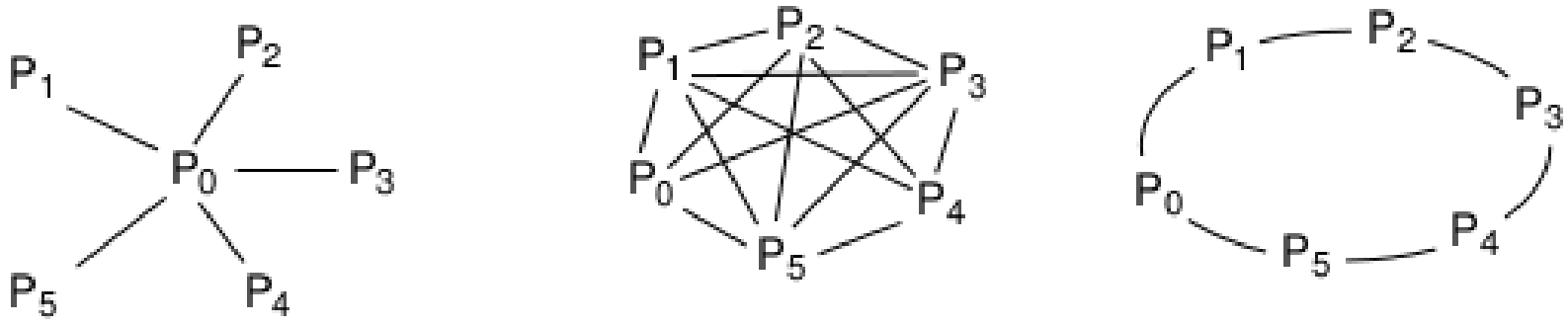
Trois (3) solutions

(a) Solution centralisée

(b) Solution symétrique (SPMD)

(c) Solution quasi-symétrique avec anneau de processus

Le problème : Calcul distribué du `min` et du `max`
Trois (3) solutions



(a) Centralized solution (b) Symmetric solution (c) Ring solution

Figure 7.14 Communication structures of the three programs.

Source : G.R. Andrews, «*Foundations of Multithreaded, Parallel, and Distributed Programming*», Addison-Wesley, 2000.

(a) Solution centralisée : Définition du processus maitre

C'est le processus maitre qui fait tout le travail, i.e., les comparaisons

```
maitre = lambda do |donnees, *resultats|
  val = rand(MAX_VAL)

  le_min, le_max = val, val
  (n-1).times do
    x = donnees.get
    le_min = [le_min, x].min
    le_max = [le_max, x].max
  end

  resultats.each do |canal|
    canal.put [le_min, le_max]
  end
end
```

(a) Solution centralisée : Définition des autres processus

Les autres processus ne font qu'envoyer leur valeur et recevoir le résultat

```
travailleurs = (1...n).map do
  lambda do |donnees, resultat|
    val = rand(MAX_VAL)

    donnees.put val
    le_min, le_max = resultat.get
  end
end
```

(a) Solution centralisée : Création des canaux et activation des processus

```
# Canal pour recevoir les donnees des processus.
donnees = PRuby::Channel.new

# Canaux pour retourner les resultats
# aux autres processus.
resultats = Array.new(n-1) { PRuby::Channel.new }

# On active les processus.
maitre.go( donnees, *resultats )
(0...n-1).each do |i|
  travailleurs[i].go( donnees, resultats[i] )
end
```

(b) Solution symétrique : Définition des processus, qui font tous exactement la même chose

C'est-à-dire tous les processus font tout le travail!

```
procs = (0...n).map do |i|
  lambda do |mon_canal, *autres_canaux|
    val = rand(MAX_VAL)

    # On transmet la valeur aux autres processus.
    autres_canaux.each { |canal|  canal.put val }

    # On recoit les valeurs des autres processus.
    le_min, le_max = val, val
    (n-1).times do
      autre_val = mon_canal.get
      le_min = [le_min, autre_val].min
      le_max = [le_max, autre_val].max
    end
  end
end
```

(b) Solution symétrique : Création des canaux et activation des processus

```
# Les canaux, tous utilises de la meme facon.
canaux = Array.new(n) { PRuby::Channel.new }

# On lance les processus.
(0...n).each do |i|
  procs[i].go( canaux[i],
               *canaux[0...i],
               *canaux[i+1..-1] )
end
```

(c) Solution en anneau... avec deux passes : Définition du premier processus

C'est le premier processus qui amorce la ronde autour de l'anneau

```
procs = []  # Tableau des divers processus.

# Initie la circulation autour de l'anneau.
procs << lambda do |gauche, droite|
  val = rand(MAX_VAL)

  # Premiere passe.
  droite.put [val, val]

  # Deuxieme passe.
  le_min, le_max = gauche.get
  droite.put [le_min, le_max]
end
```

(c) Solution en anneau... avec deux passes : Définition des autres processus

Les autres processus

```
(1...n).each do |i|
  procs << lambda do |gauche, droite|
    val = rand(MAX_VAL)

    # Premiere passe.
    le_min, le_max = gauche.get
    le_min = [le_min, val].min
    le_max = [le_max, val].max
    droite.put [le_min, le_max]

    # Deuxieme passe.
    le_min, le_max = gauche.get
    droite.put [le_min, le_max]
  end
end
```

(c) Solution en anneau... avec deux passes : Définition des autres processus

Les autres processus... pour que tous les canaux soient vides lorsqu'on termine

```
(1...n).each do |i|
  procs << lambda do |gauche, droite|
    val = rand(MAX_VAL)

    # Premiere passe.
    le_min, le_max = gauche.get
    le_min = [le_min, val].min
    le_max = [le_max, val].max
    droite.put [le_min, le_max]

    # Deuxieme passe.
    le_min, le_max = gauche.get
    droite.put [le_min, le_max]  unless i == n-1
  end
end
```

(c) Solution en anneau... avec deux passes : Création des canaux et activation des processus

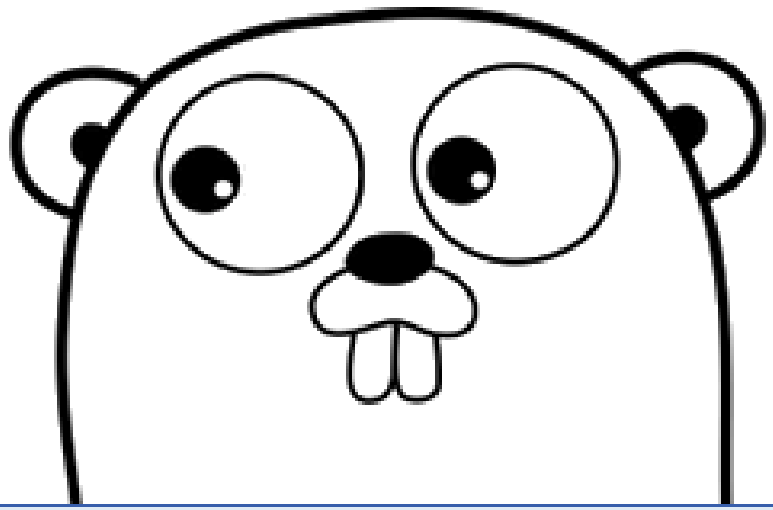
```
# Les canaux.
canaux = Array.new(n) { PRuby::Channel.new }

# On active les processus.
(0...n).each do |i|
  procs[i].go( canaux[i],                          # Gauche
               canaux[i == n-1 ? 0 : i+1] )# Droite
end
```

Aperçu

Le langage Go

https://golang.org/

Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.



Download Go

Binary distributions available for
Linux, Mac OS X, Windows, and more.

Le langage Go

Go est un langage de programmation compilé et concurrent inspiré de C et Pascal. Ce langage a été développé par Google à partir d'un concept initial de Robert Griesemer, Rob Pike et Ken Thompson.

Source :
`https://fr.wikipedia.org/wiki/Go_%28langage%29`

Le langage Go : La concurrence

Go intègre directement, comme Java, les traitements de code en concurrence. Le mot clé go permet à un appel de fonction de s'exécuter en concurrence avec le thread courant. Ce code exécuté en concurrence se nomme une goroutine par analogie lointaine avec les coroutines [—] pas forcément [exécuté] dans un nouveau thread [. . .].

Les goroutines communiquent entre elles par passage de messages, en envoyant ou en recevant des messages sur des canaux.

Source :
`https://fr.wikipedia.org/wiki/Go_%28langage%29`

Le langage Go : La concurrence

Go intègre directement, comme Java, les traitements de code en concurrence. Le mot clé go permet à un appel de fonction de s'exécuter en concurrence avec le thread courant. Ce code exécuté en concurrence se nomme une goroutine par analogie lointaine avec les coroutines [—] pas forcément [exécuté] dans un nouveau thread [. . .].

Les goroutines communiquent entre elles par passage de messages, en envoyant ou en recevant des messages sur des canaux.

Source :
`https://fr.wikipedia.org/wiki/Go_%28langage%29`

Le langage Go : Canaux et opérations de base

- `c chan int`
- `<- c`
- `c <- v`
- `range c`
- `close(c)`

Le langage Go : Canaux et opérations de base

- `c chan int` :
  Déclaration d'un canal `c` pour transmettre des `int`s.

- `<- c` :
  Lecture d'une valeur sur `c` — ≈ `c.get` en PRuby.

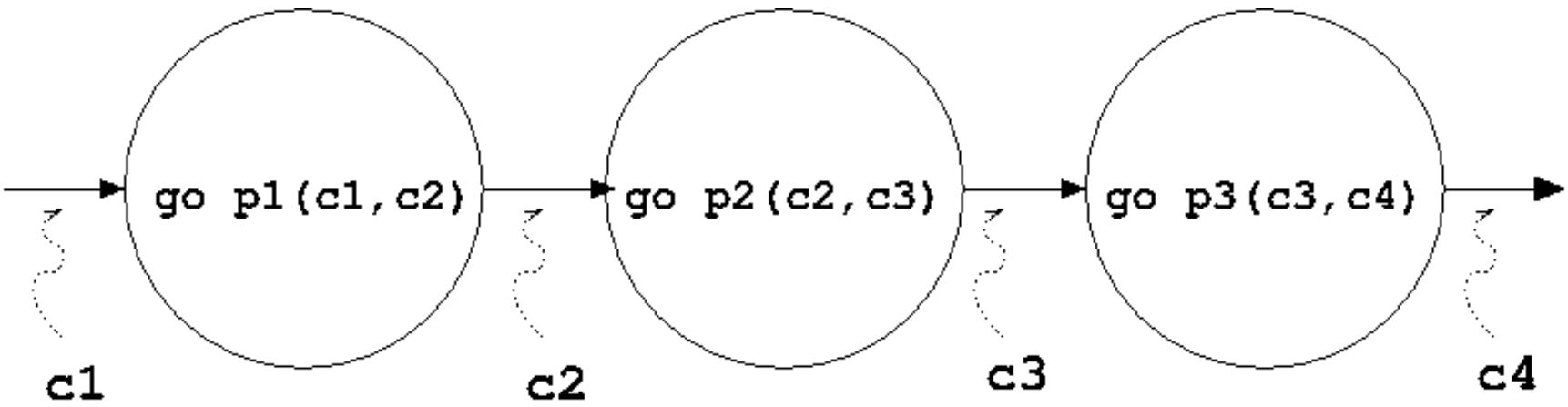
- `c <- v` :
  Écriture de `v` sur `c` — ≈ `c.put v` en PRuby.

- `range c` :
  Énumération des éléments de `c` — ≈ `c.each` en PRuby.

- `close(c)` :
  Fermeture du canal `c`.

4.1 Un petit pipeline avec trois processus

Représentation graphique du pipeline (linéaire) avec trois processus et quatre canaux



Un petit pipeline
Les trois (3) processus

```
func p1( cin chan int, cout chan int ) {
  n := <- cin
  for i := 1; i <= n; i++ { cout <- i }
  close( cout )
}

func p2( cin chan int, cout chan int ) {
  for v := range cin { cout <- 10 * v }
   close( cout )
}

func p3( cin chan int, cout chan int ) {
   r := 0
   for v := range cin { r +=  v }
   cout <- r
   close( cout )
}
```

Un petit pipeline
Le programme principal

```
func main() {
  c1 := make( chan int )
  c2 := make( chan int )
  c3 := make( chan int )
  c4 := make( chan int )

  go p1( c1, c2 )
  go p2( c2, c3 )
  go p3( c3, c4 )

  c1 <- 10
  fmt.Printf( "%d\n", <- c4 )
}
```

4.2 Quelques autres éléments du langage Go

Une caractéristique fondamentale des canaux en Go

Les canaux Go sont toujours bornés

- Par défaut, si aucune taille n'est spécifiée, alors le tampon est de taille nulle

  $\Rightarrow$

  l'envoi et la réception se font de façon synchrone = rendez-vous

Une caractéristique fondamentale des canaux en Go

Les canaux Go sont toujours bornés

- Par défaut, si aucune taille n'est spécifiée, alors le tampon est de taille nulle

  $\Rightarrow$

  l'envoi et la réception se font de façon synchrone $=$ rendez-vous

Les canaux PRuby peuvent être non bornés

- Par défaut, si aucune taille n'est spécifiée, alors le tampon associé est non-borné

L'opération `select` obtient un élément d'un canal sélectionné parmi plusieurs

```
select {
  case r1 := <- ch1:
    ...; return

  case r2 := <- ch2:
    ...; return

  case <-time.After(100 * time.Millisecond):
    ...; return
}
```

- Va prendre un élément sur `ch1` ou sur `ch2`, à moins que le *time-out* ne survienne auparavant.