EPFL

Programmation pour Ingénieur

Design pattern

ME 3e semestre

rev. 2025.1

Christophe Salzmann

Laboratoire d'Automatique

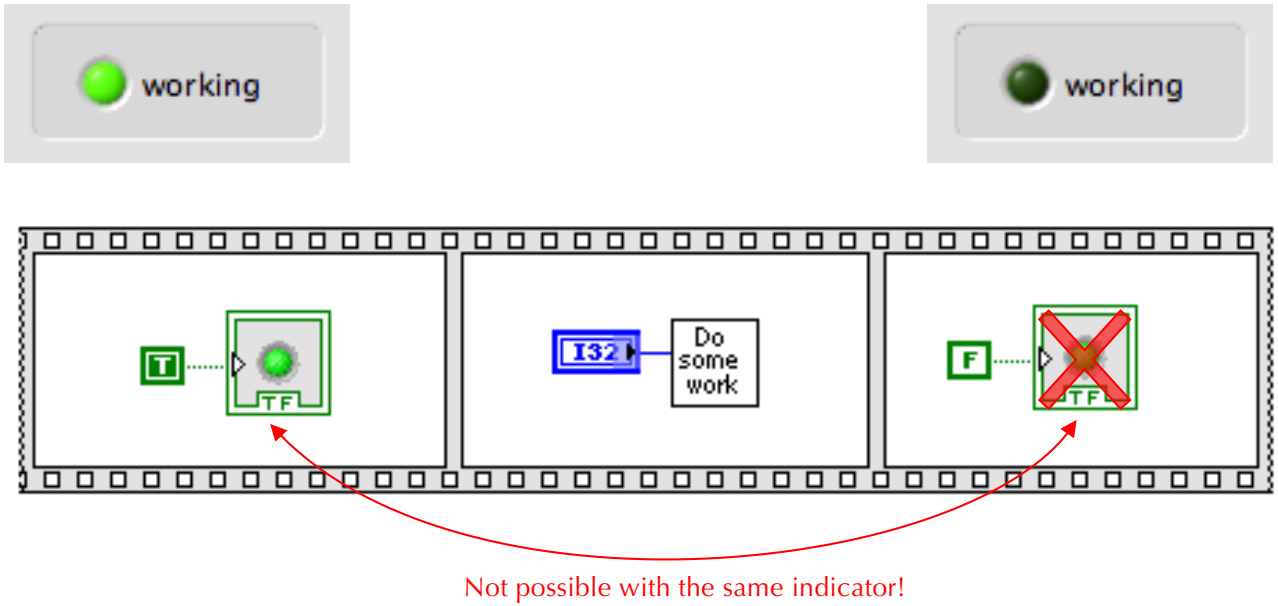
Rappel LabVIEW 3

- Boucles, indexing, shift-register
- String
- Array
- Cluster (err), Matrix
- Affichage: **chart** (1pt), **plot** (*n* pts) *-> click de droit pour configurer*
- Acquisition de signaux

LabVIEW Module 4

- Local and global
- Race condition
- Action Engine
- Machine d'états

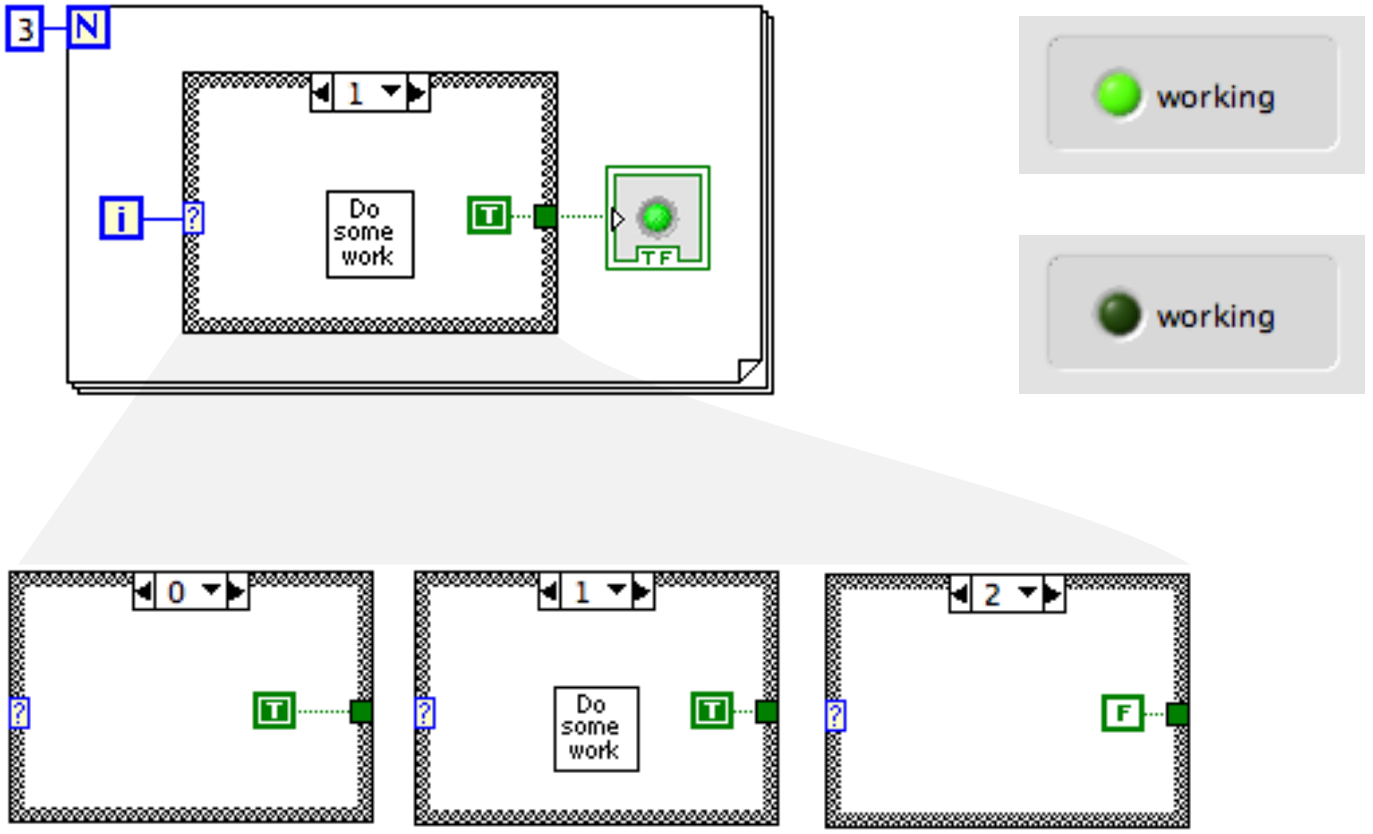
Locals (and globals)



Comment éteindre la LED *working* quand le VI s'arrête?

Locals (and globals)

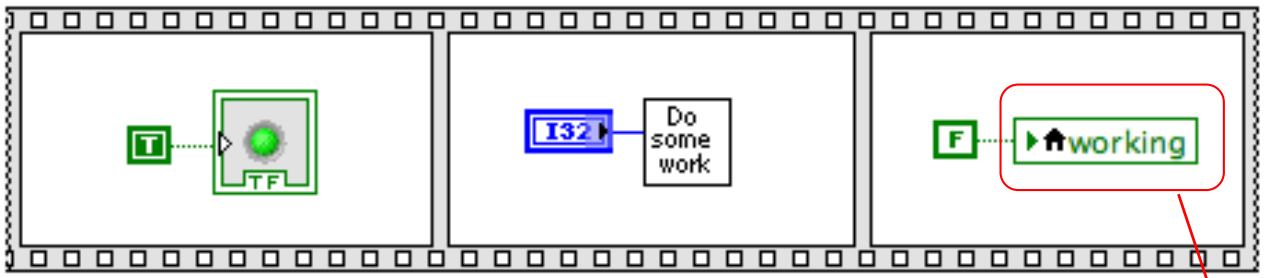
Idée: utiliser une *for loop* pour passer par les différents états stockés dans un *switch case*



Locals (and globals)

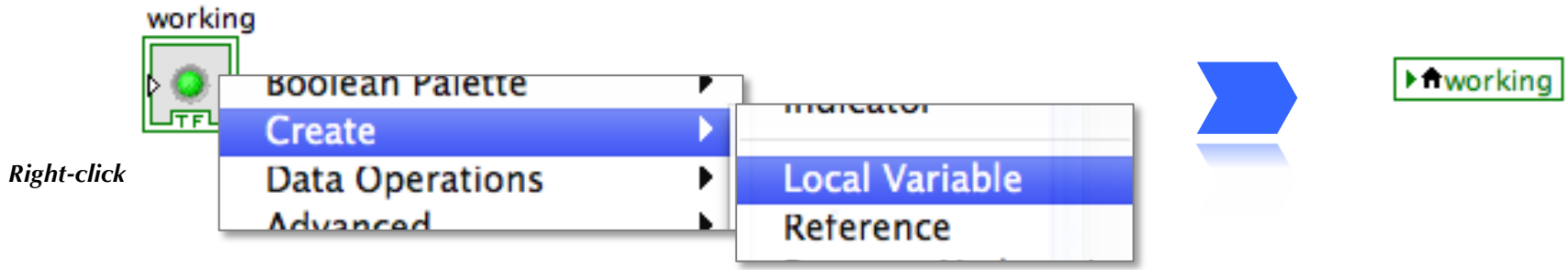
Lire/écrire dans une variable locale est comme lire/écrire dans un control/indicateur. Cela permet d'accéder aux controls/indicateurs depuis plusieurs endroits du *diagram*.

La variable locale *working* représente l'indicateur *working*.



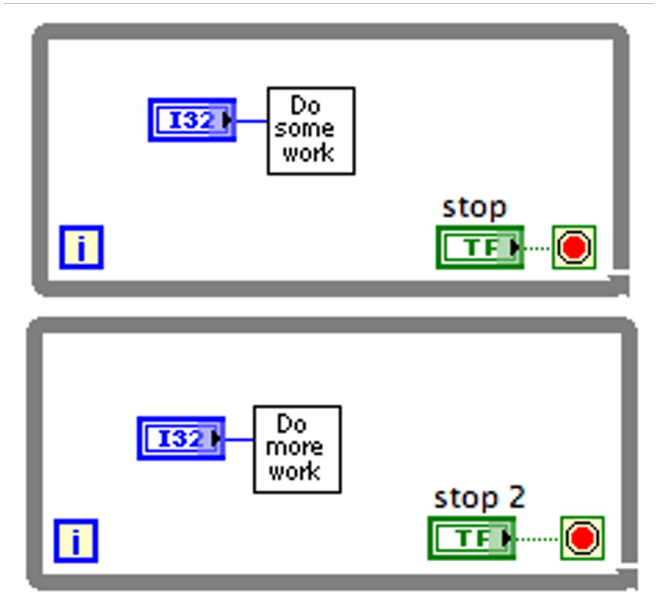
***working* local variable**

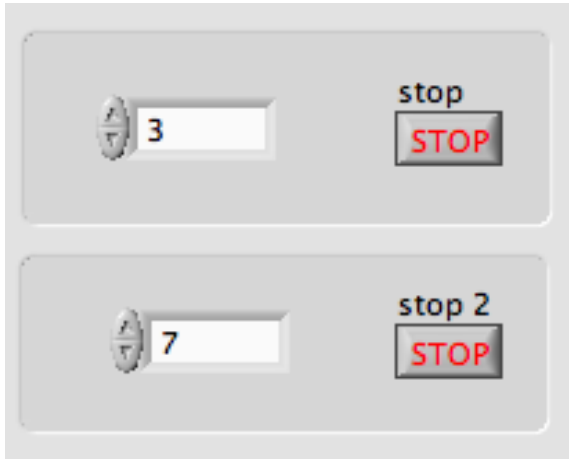
Create a local variable



Right-click

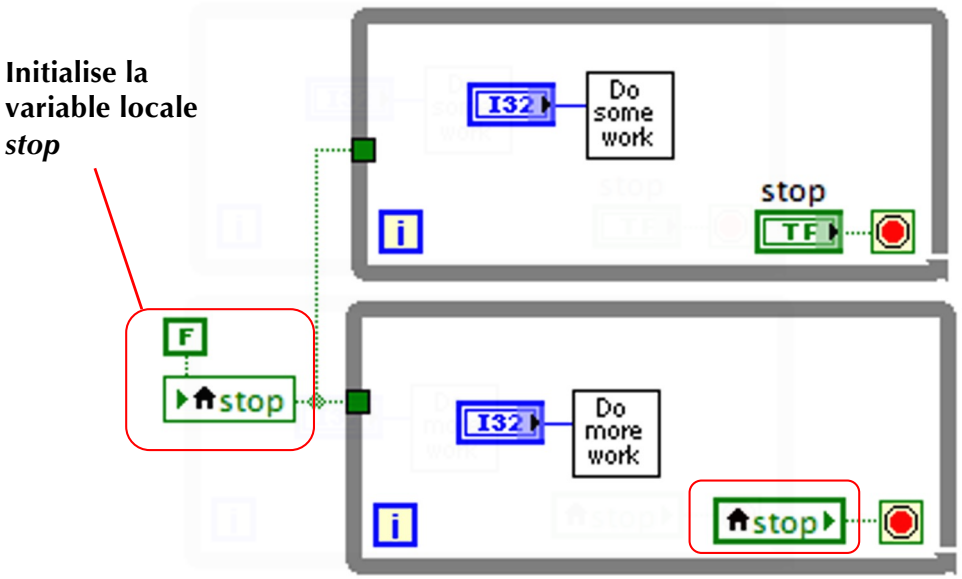
Locals (and globals)

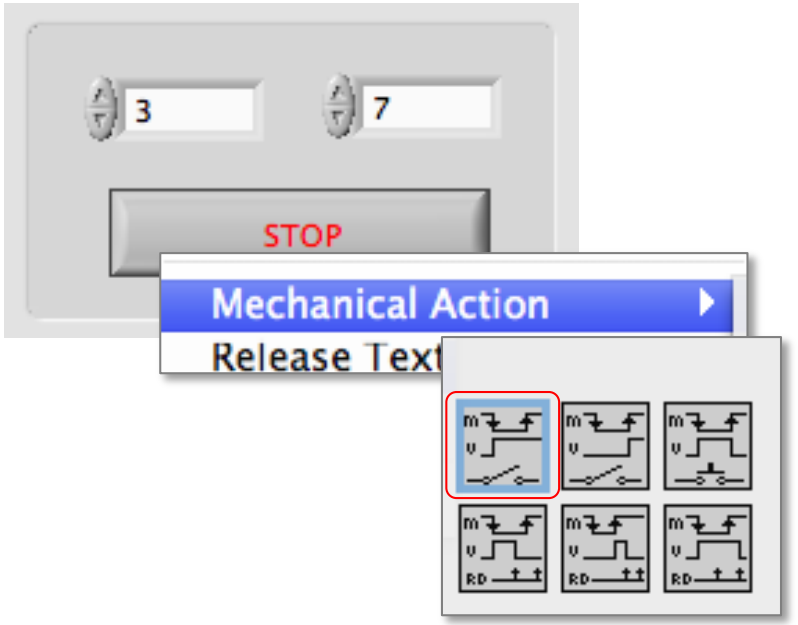




Comment stopper les deux boucles avec 1 seul *Stop* ?

Locals (and globals)

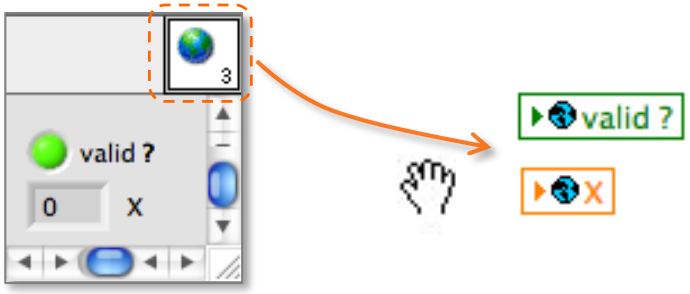




L'action méchanique (*latch*) par défaut du bouton *stop* n'est pas compatible avec l'utilisation d'une variable locale. Vous devez utiliser l'action *Switch when pressed*

Locals (and globals)

- La portée d'une variable **Local** est le **VI** dans lequel elle *vit*.
- La portée d'une variable **Global** est l'**application** LabVIEW.
- Les Globals sont des controls/indicators stockés dans un *front panel* spécifique (= VI sans diagram).



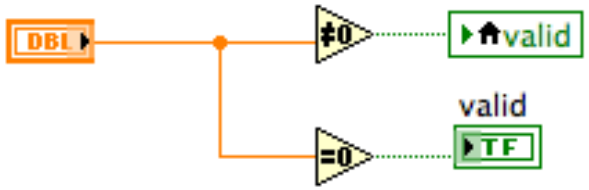
Drag in the diagram, then select

- L'utilisation de global/local peuvent ralentir l'exécution de votre VI.
- Lorsque vous utilisez des global/local, le flot normal des opérations est cassé! Il devient possible d'accéder à un *control/indicator* de deux endroits distincts. Qui sera le premier/deuxième à accéder la variable (race condition) ?

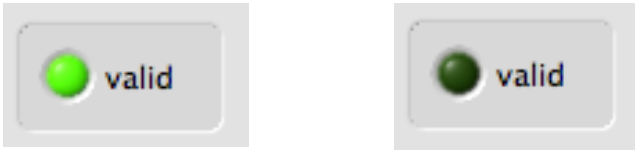
Race condition

Les opérations du *diagram* s'exécutent en parallèle dès que possible.

Quelle est la valeur de *valid* ?



valid ?



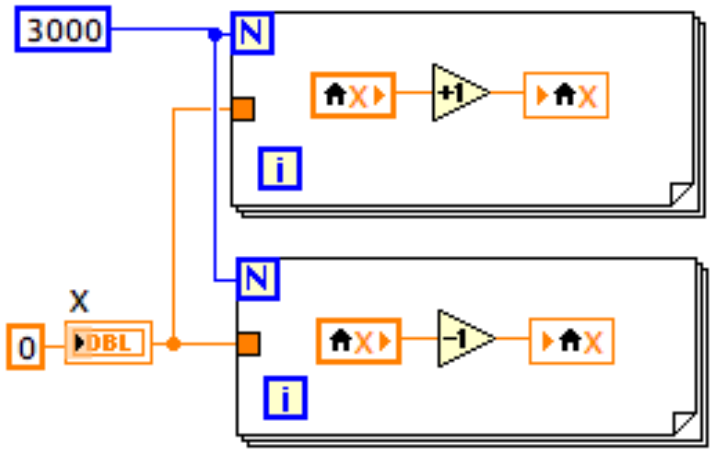
Impossible à dire!

*Il y a une course (**race**) entre l'indicateur et la variable locale pour modifier (**accéder**) la valeur de la ressource partagée (**shared resource**)* valid.

Race condition

Mettre X à 0, puis incrémenter X 3000 fois et en même temps le décrémenter 3000 foi. Si les opérations étaient séquentielles/sérialisées le résultat serait 0!

Mais LabVIEW est intrinsèquement parallèle et le résultat n'est pas 0! Dans les faits il est impossible de prédire la valeur de X…

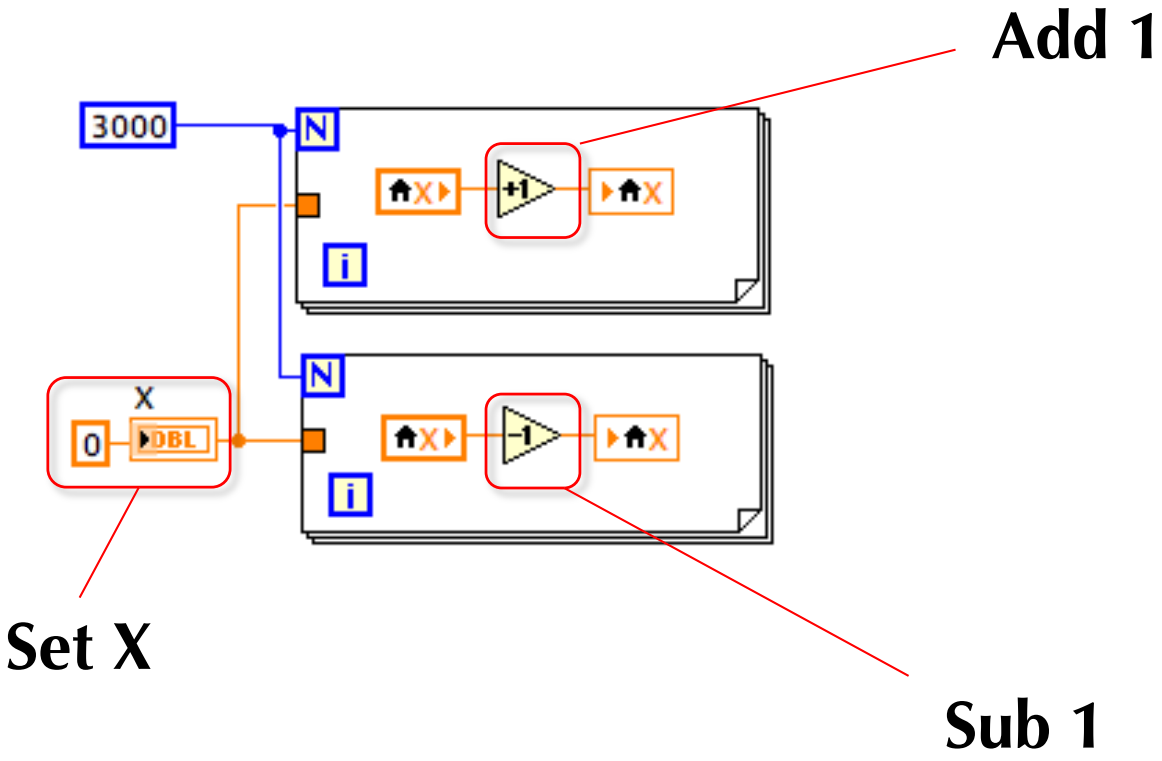


X = ?

Impossible à déterminer!

=> évitez ce cas de figure!

Race condition

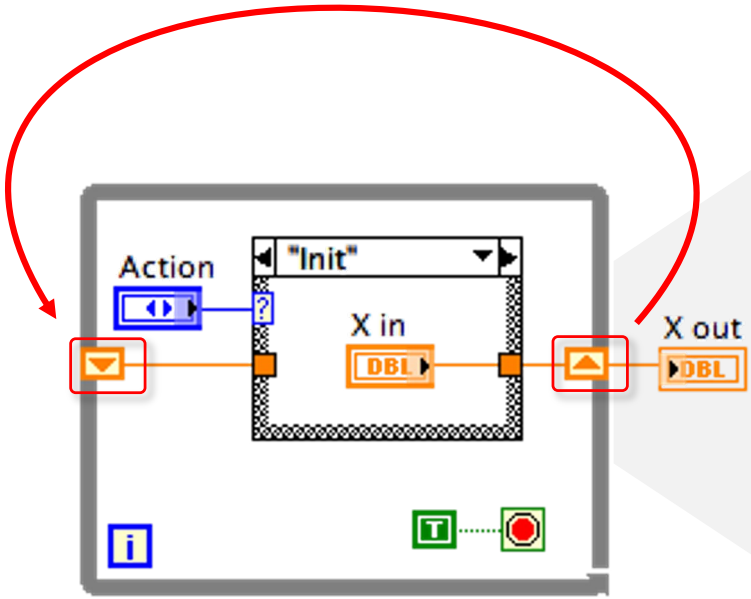


Idée: **un** VI gère ces 3 actions en les sérialisant, i.e. **un seul** accès à la resource partagée X

FGV
+/- 1

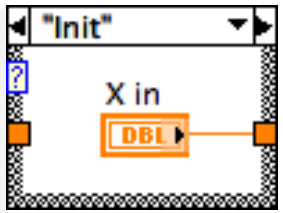
Race condition - sérialisation

Le shift register fait office de mémoire locale

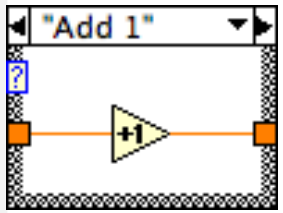


4 Actions possibles:

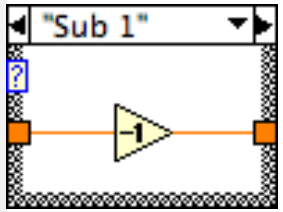
- Init
- Add 1
- Sub 1
- Get X



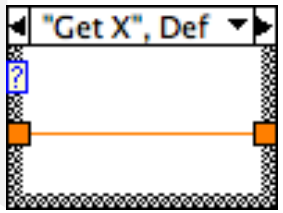
Init
Initialise le *shift register* à une X in



Add 1
Ajoute 1 au *shift register*

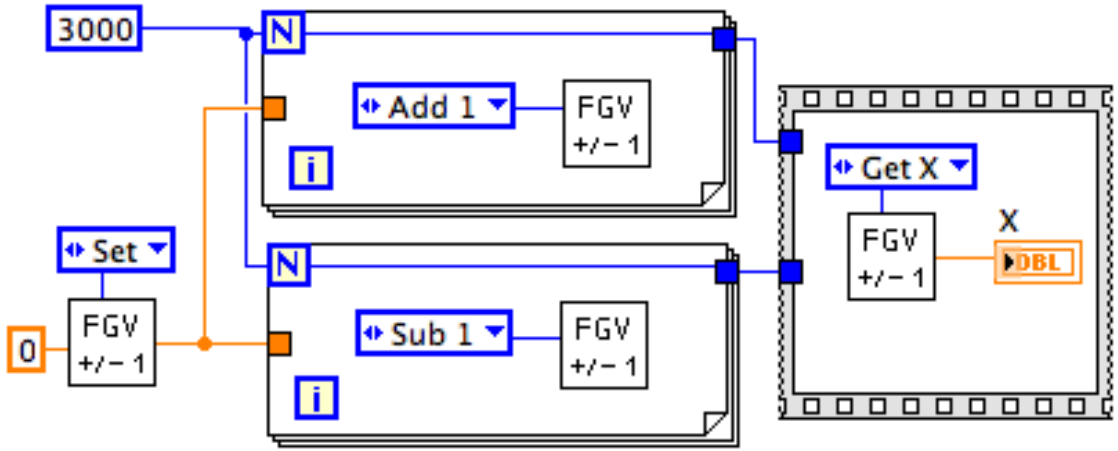


Sub 1
Soustrait 1 au *shift register*



Get X
Retourne la valeur du *shift register*

~~Race condition~~

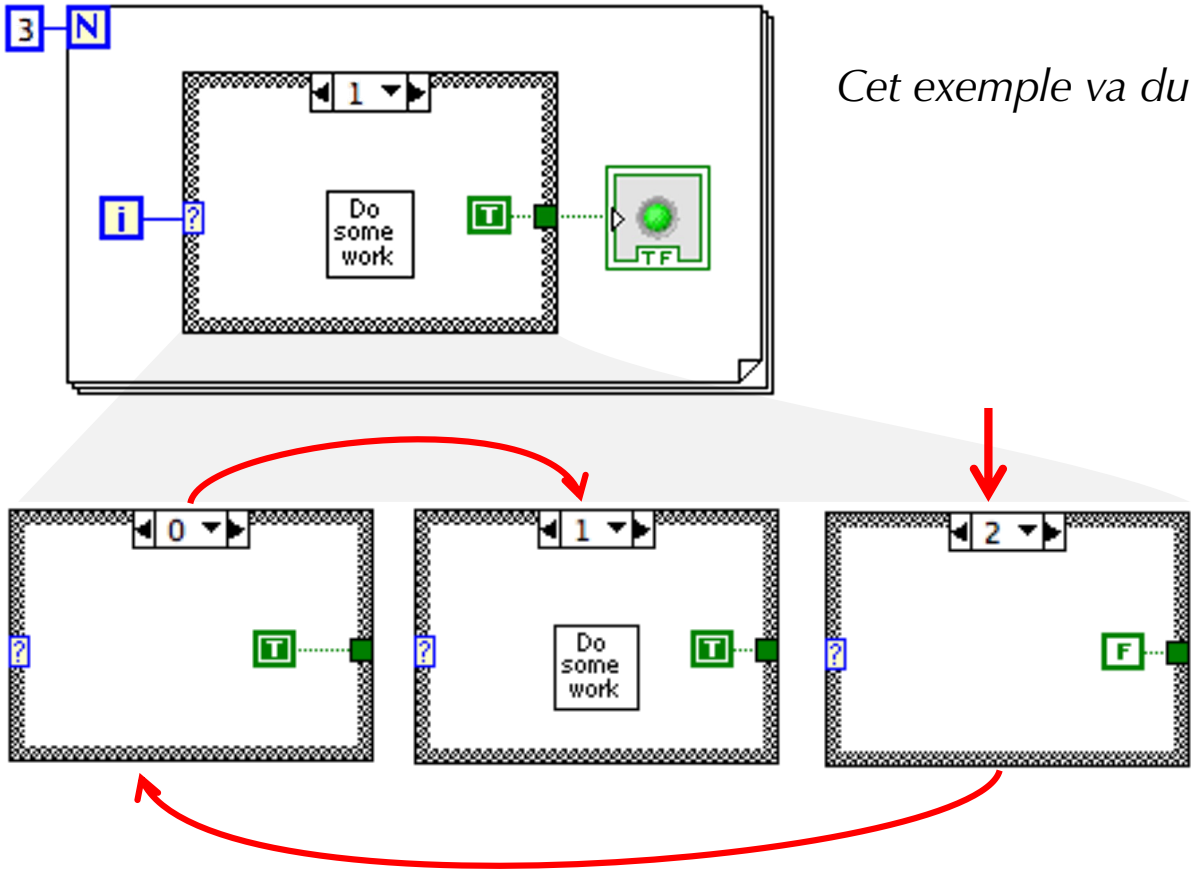


X = 0

Seul un *process (for loop)* à la fois peut accéder au VI *FGV +/- 1* vi. L'accès à la resource partagée est sérialisé.

Machine d'états (state machine)

- Exécute une séquence d'actions déterminées programatiquement

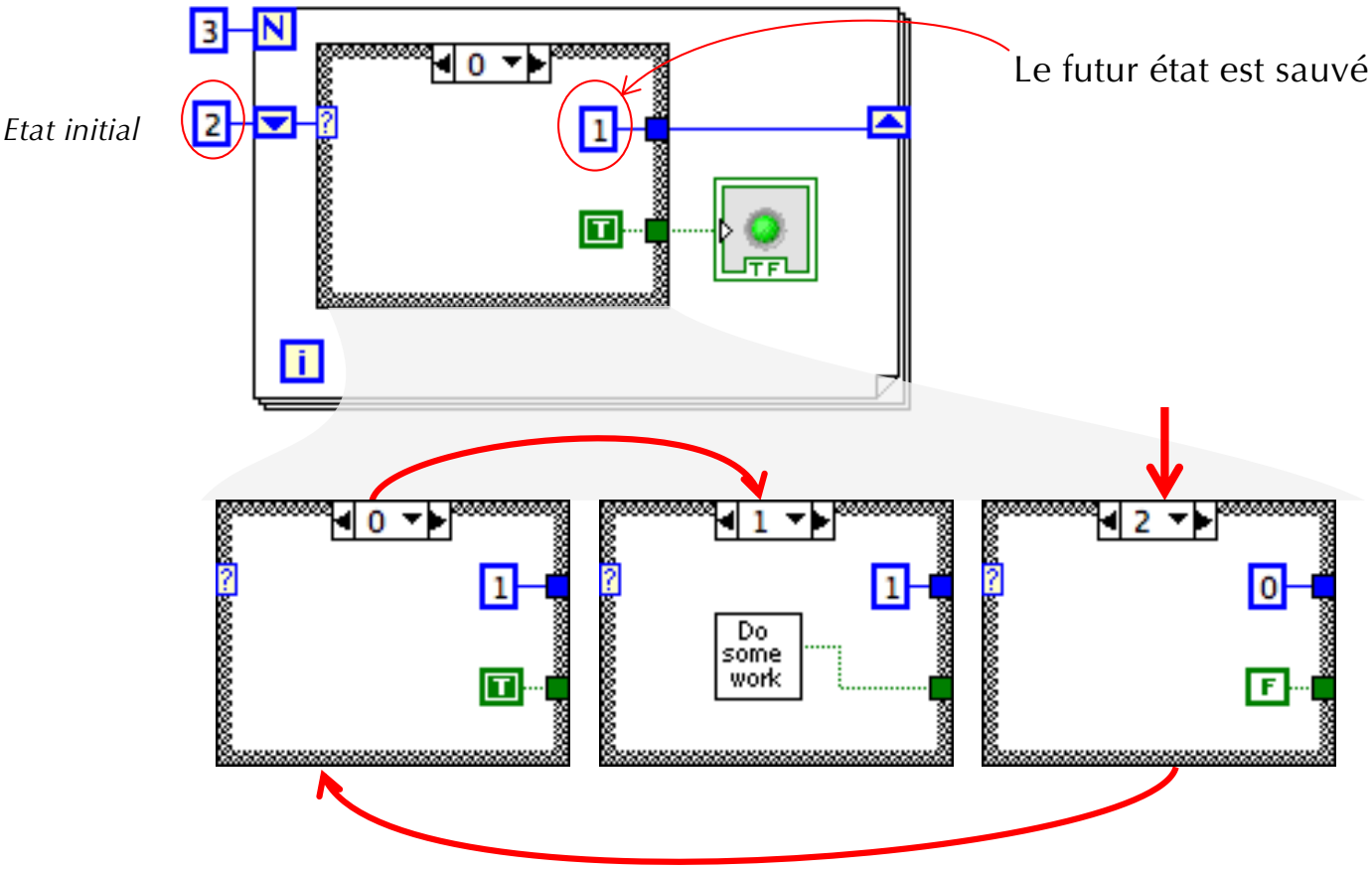


Cet exemple va du cas 0, puis 1 et enfin 2

Comment faire pour allez du cas 2, puis 0 et enfin 1 ?

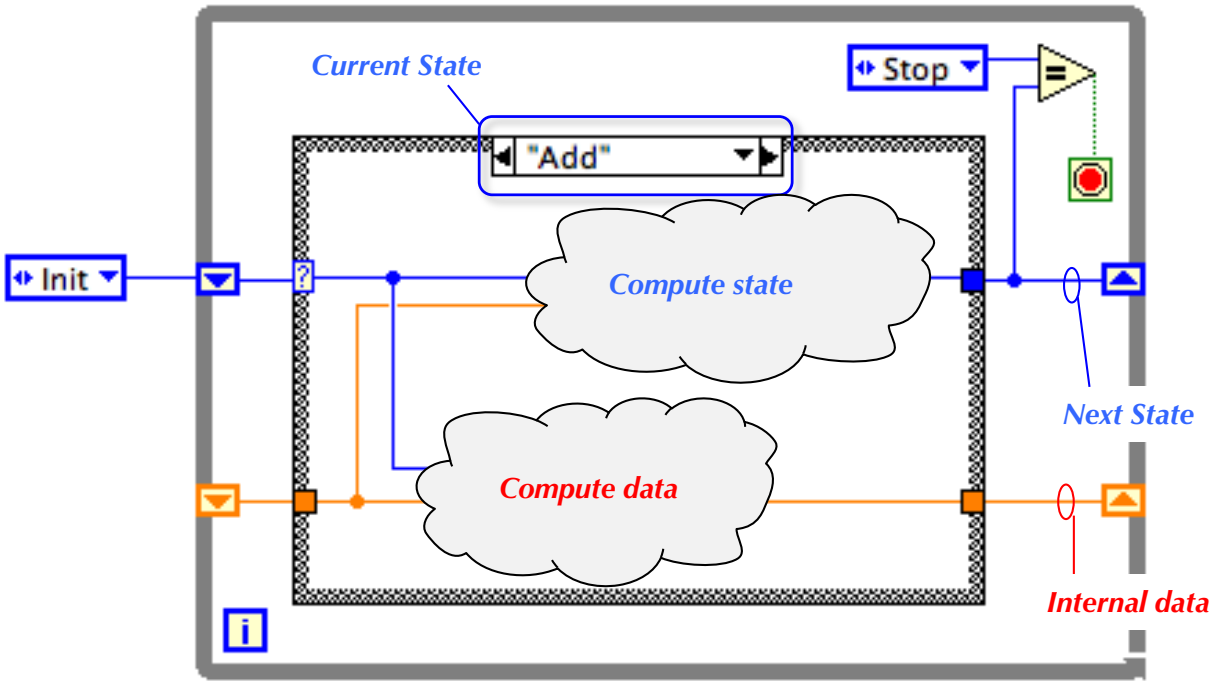
Machine d'états (state machine)

- Reprenons l'idée de l'*action engine*, mais cette fois l'**action est déterminée par l'état courant**



Machine d'états

- Similaire à l'Action Engine
- L'**action** devient l'**état**, l'**état suivant** est calculé dans l'**état courant**
- Il y a un *case* par état
- Des informations supplémentaires peuvent être stockées dans des *shift-register(s)*

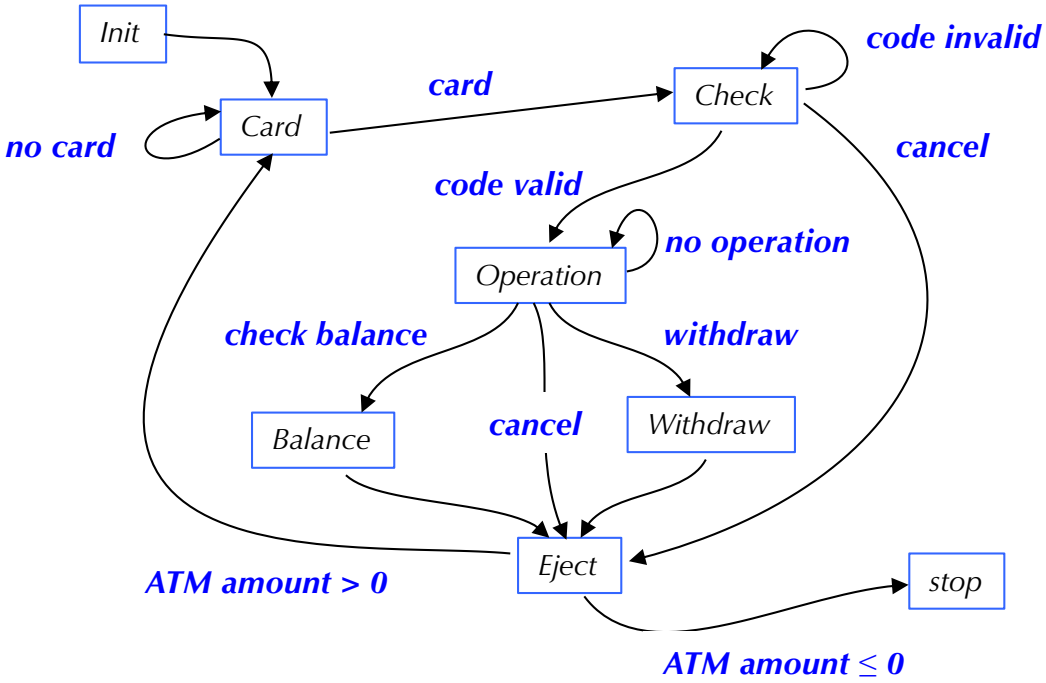


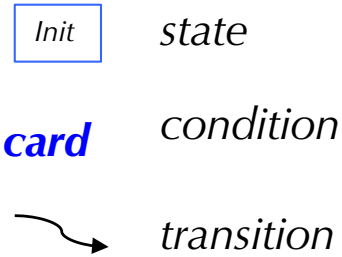

```
State = Init;
While (State!=Stop) {
  switch state {
    case Add: …
    case Sub: …
  }
  State = …
}
```

Machine d'états

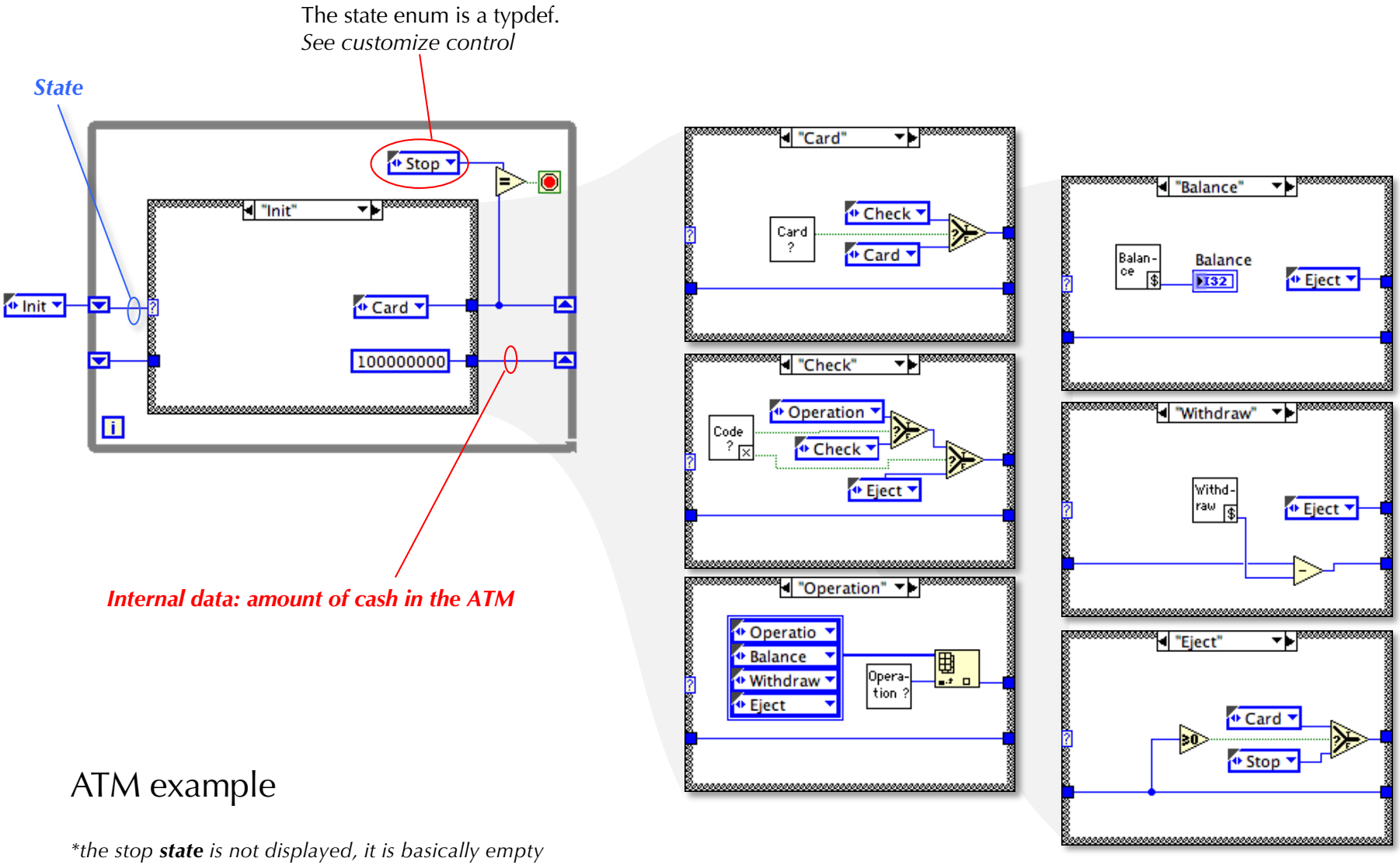
- Exécute une séquence d'opérations déterminées programmatiquement
- Le nouvel état est calculé en fonction de l'état courant et/ou d'autres conditions

Ex. ATM





Machine d'états



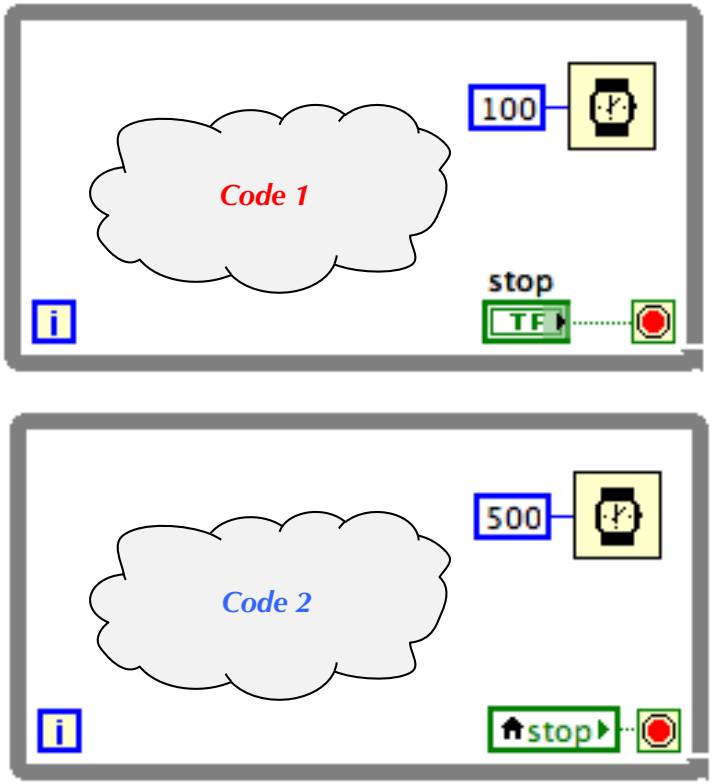
ATM example

*the stop **state** is not displayed, it is basically empty

Design patterns

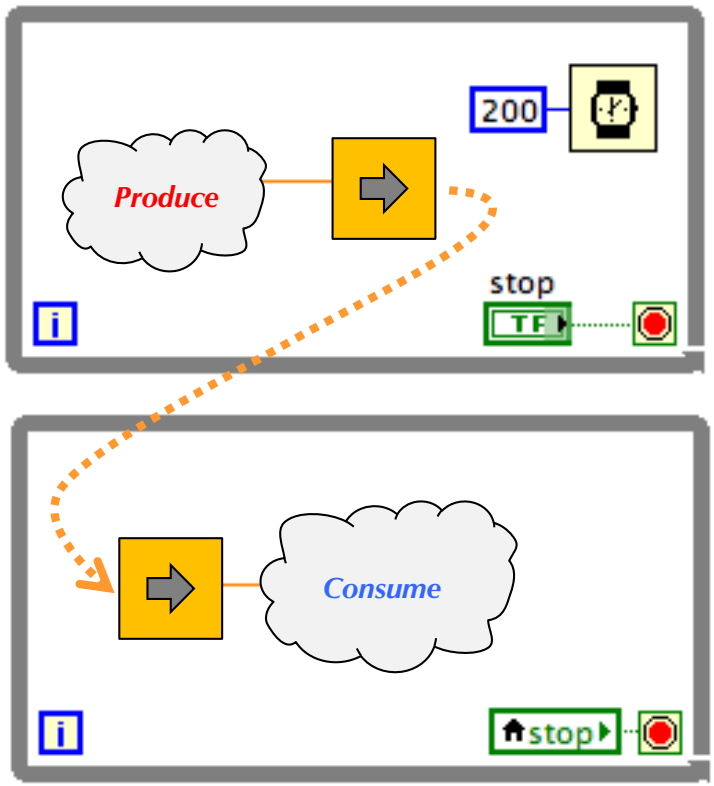
- One shot, ordered via error cluster
- One loop
- Functional global/action engine
- State machine
- Event loop
- Multiple loops = multi-threads ☺
- Producer/consumer

Classic double



- Pas d'echange de donnée -> pas de dépendence (à part le bouton *stop*)
- Les deux boucles sont exécutées à des fréquences différentes.

Producer - consumer



Master-slave behavior

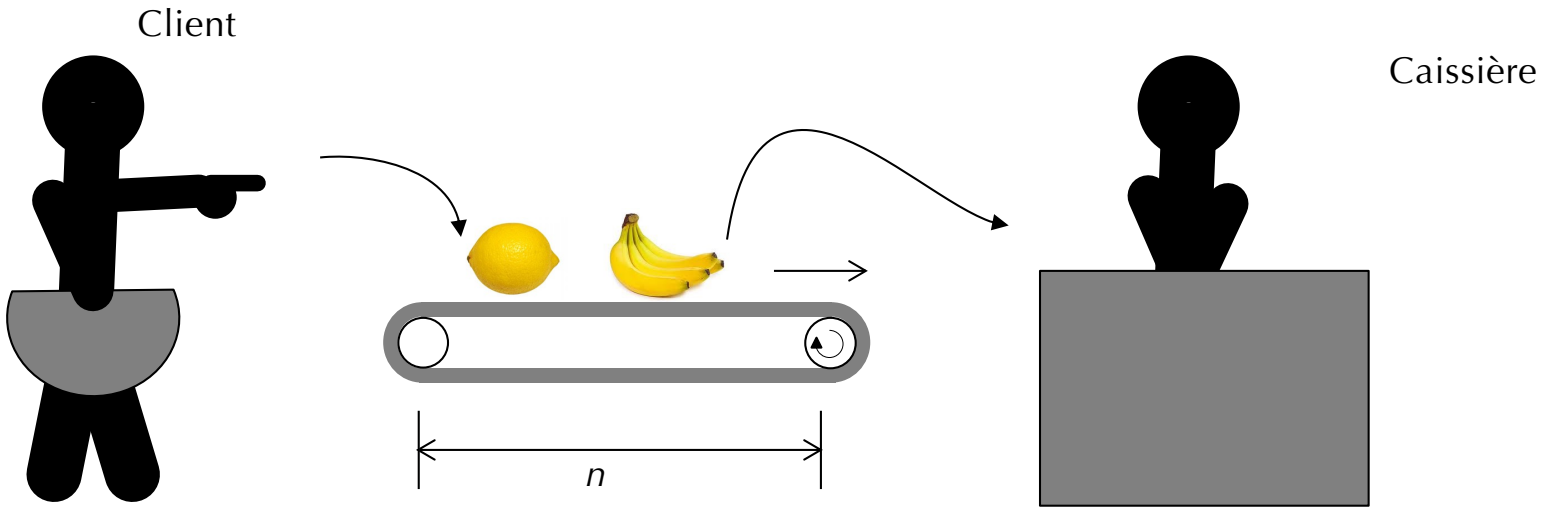
- Le producteur (**master**) genère des données à son rythme
- Le consomateur (**slave**) consomme les données et attend (éternellement?) jusqu'à ce que de nouvelles données soient disponibles

- Comment échanger les données et garantir qu'elles sont transférées en intégralité? -> **queues** *(prochains slides)*

- Ou **Tunnel** >LabVIEW 2016

Buffer/tampon

Exemple: tapis roulant à la caisse d'un supermarché.

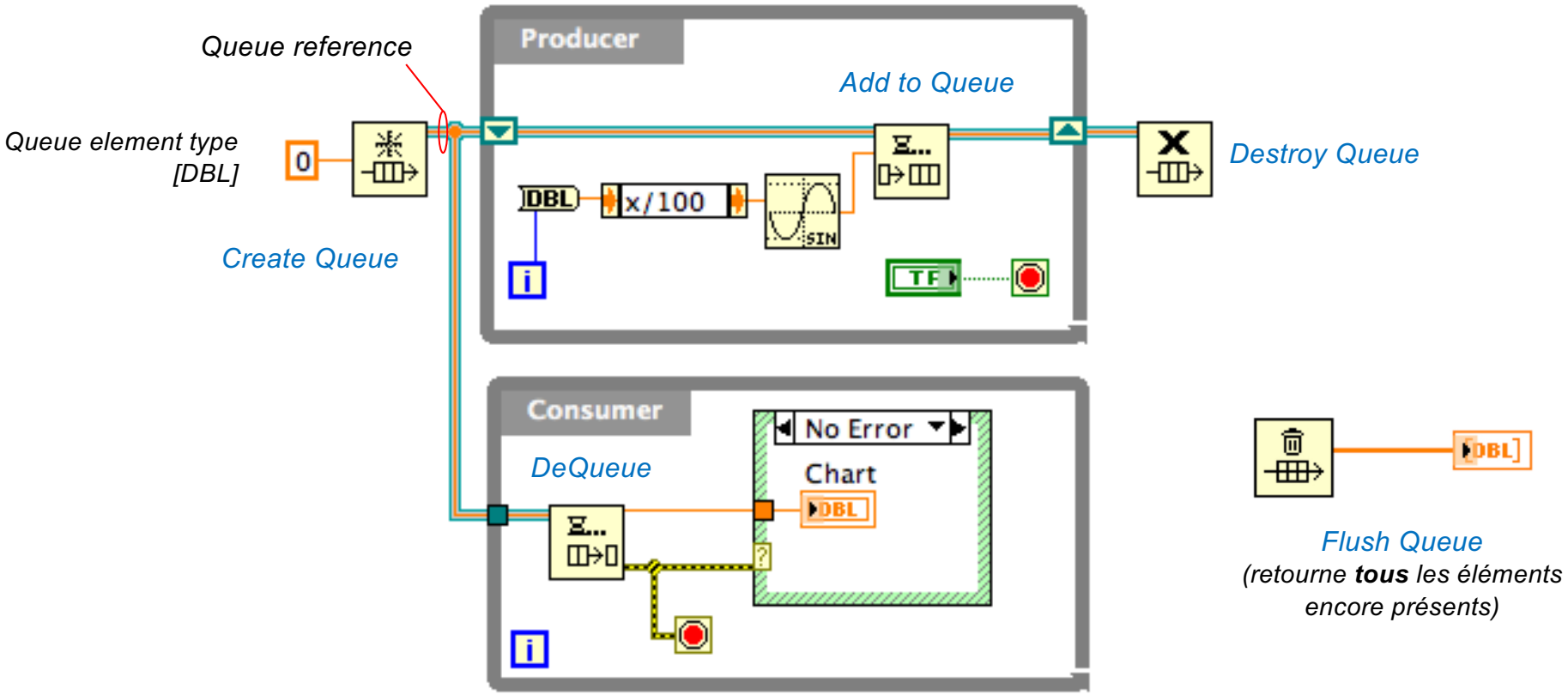


Le/la client pose ses achats sur le tapis roulant.
Le/la caissière prend les achats du tapis roulant un à un et les enregistres.
Le tapis roulant fait office de *buffer*.
Il peut contenir *n* éléments (achats) à la fois.

Queues

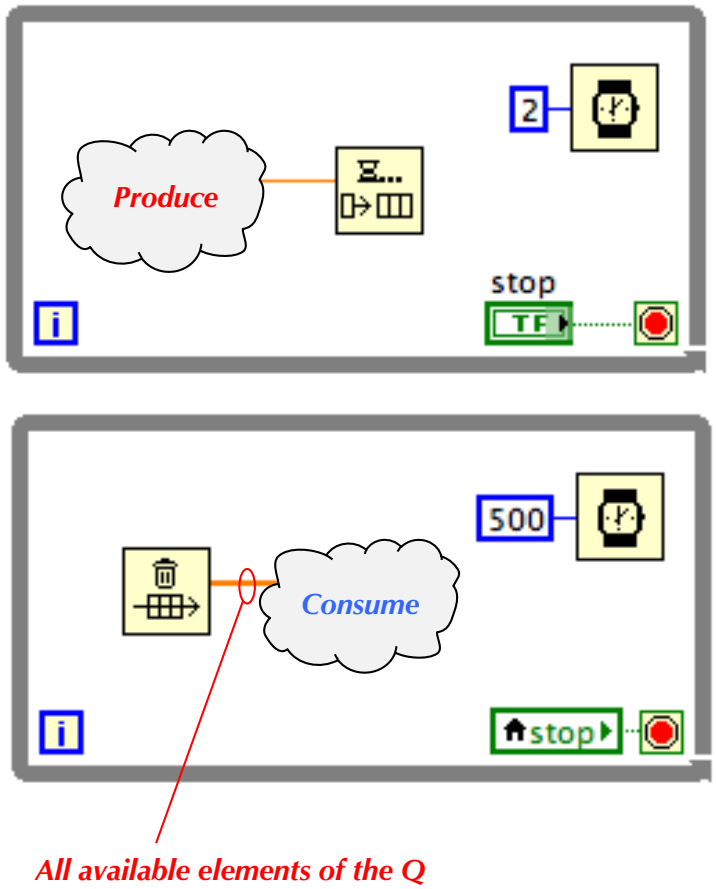
- Le concept de queues est similaire au tapis roulant que l'on trouve aux caisses des magasins
- Ce que l'on dépose d'un coté est enlevé de l'autre
- Le premier élément déposé est le premier enlevé (FIFO- First in First out)
- La vitesse à la quelle on dépose les éléments sur le tapis roulant n'est pas forcément la même que celle à laquelle on enlève les éléments
- Plus le tapis roulant est grand, plus on peut mettre des éléments dessus, le tapis roulant est un *buffer* temporaire ayant un capacité donnée
- Si l'on enlève les éléments trop lentement le tapis déborde, au contraire si les éléments sont déposés trop lentement le tapis est vide

Producer - consumer



- *Which loop is the master ? Why ?*
- *You can have many producers and many consumers*

Producer - consumer



Multi-rate

- Asynchronous execution, the two loops run at different pace
- Data are exchanged via **queue**

- ex. one loop acquire data point at high rate, the other one process the available points

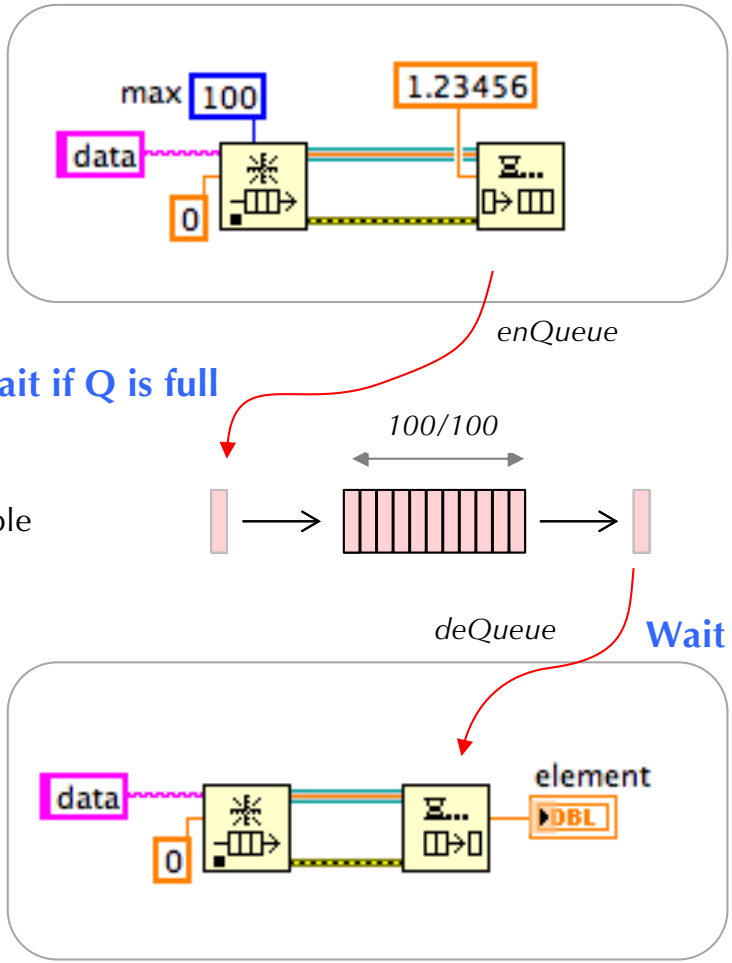
Recap

- Local and global
  - Local for VI, Global for application
  - May slow down the VI execution
  - Prone to race condition
- Race condition
  - Protect access to shared resource with semaphore, FGV
- Queue
  - to transfer data efficiently between loops
- Design patterns
  - FGV, State machine, producer-consumer, events loop

Additional material

- Race condition – sémaphores
- Mutli-threaded code
- Queue
- Event loop
- Real-time loop

Queues

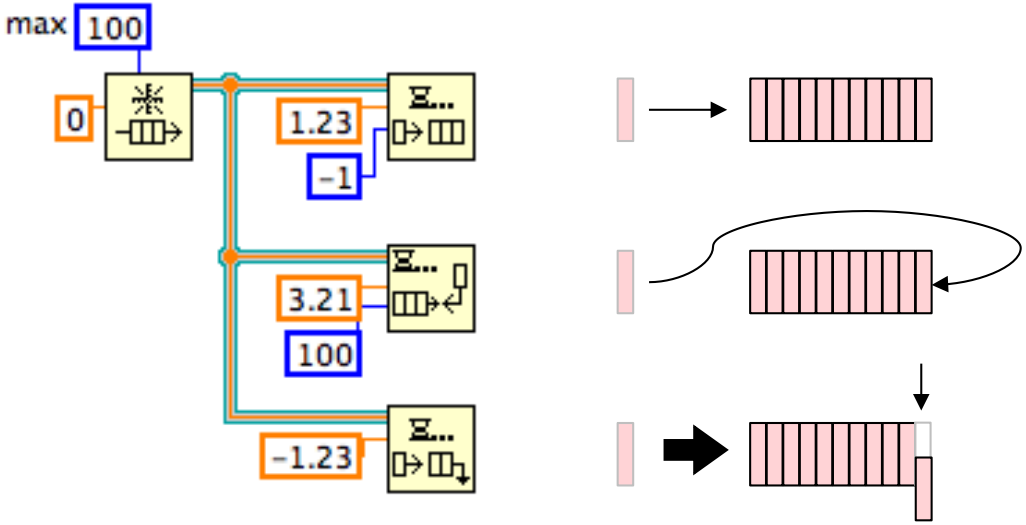


Queue name: “data”
Queue element type: double
Max nbr. elements: 100
FIFO access

```
#define qmax 100
typedef Queue_e double;
Queue_e data[qmax];

Thread 0
Err = EnQueue(data,
              1.23456,
              wait);

Thread 1
Err = DeQueue(data,
              &element,
              wait);
```

Queues & stacks

Create un-named queue of 100 double



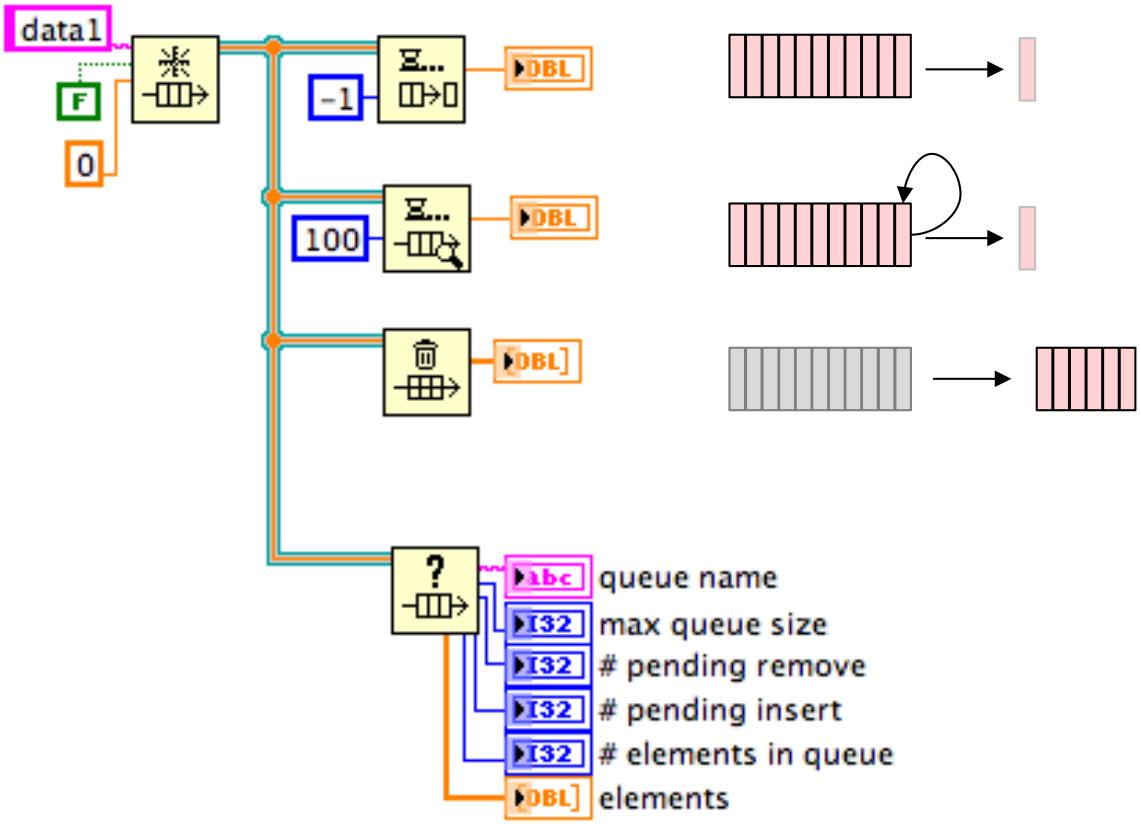
Wait forever if Q is full

EnQueue element in the front (**LIFO**)
Wait 100 *ms* if Q is full, then discard
-> then queue behaves like a stack

Force enQueue **without** delay,
if Q is full, discard the front element

Queues

Find the queue named “data1”



DeQueue, wait forever if Q is full

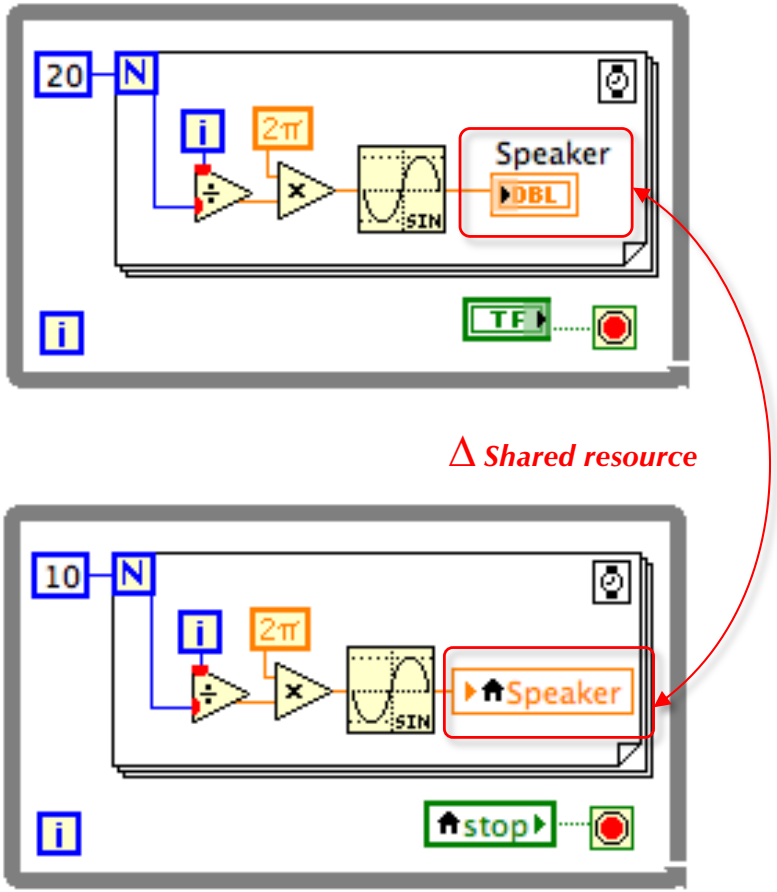
Preview element, don’t deQueue
Wait 100 *ms* if Q is empty, then discard

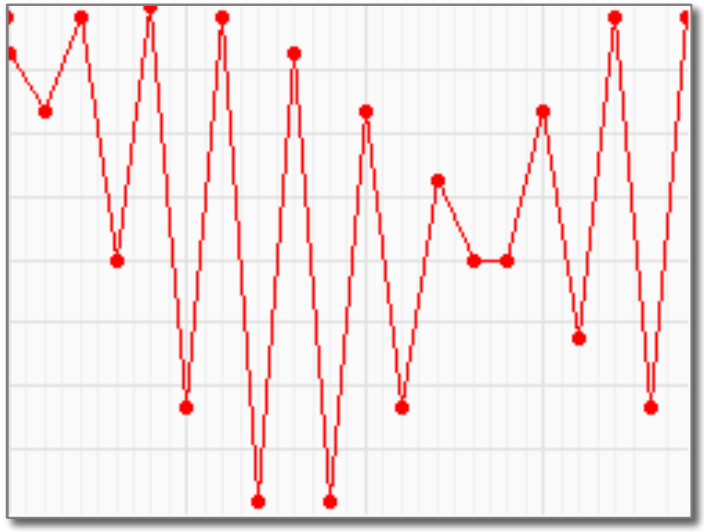
Flush the Queue without delay

Get info about the Queue,
Does not alter its content

Race condition

Imagine an hardware resource not serialized, ex. speaker.



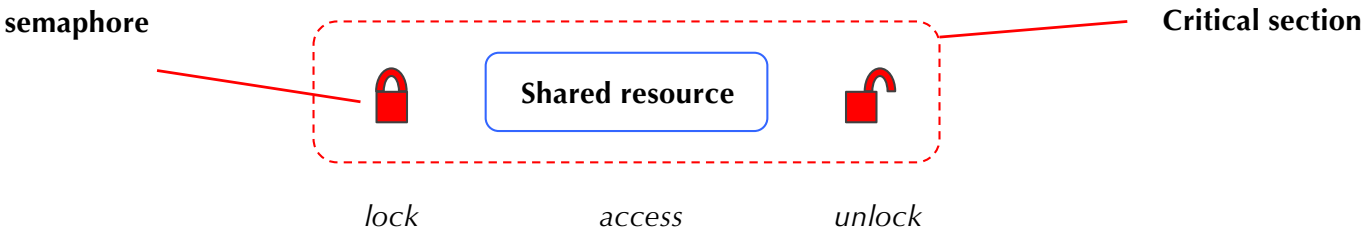


Simulated speaker

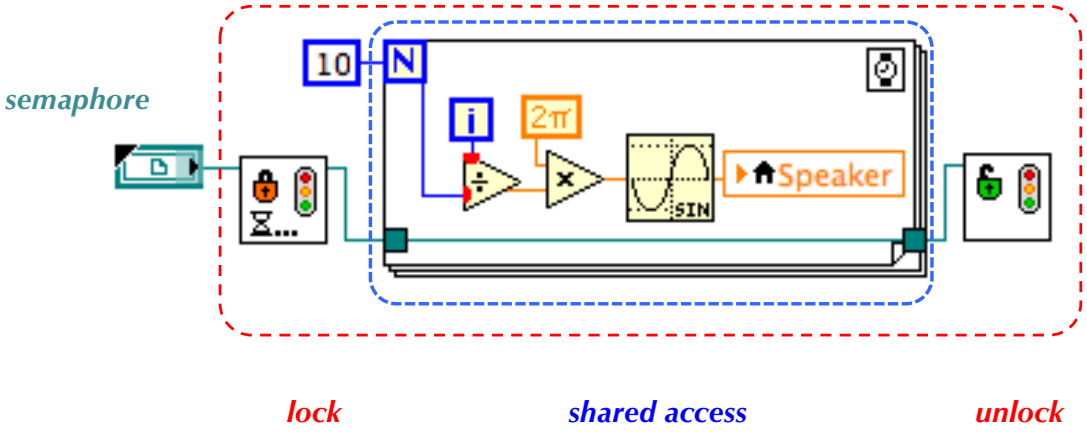
Critical section

- When a resource can be accessed (read/write) from more than one place in the diagram it becomes a critical section that has to be protected to ensure cohenrency, but how ?

- Solution: serialize the access to the shared resource
  - Use semaphore (it is like a lock) or similar mechnisms
  - Use functional globals (to replace global/local)
  - If appropriate, use another structure to store data (queue ?)



Semaphore

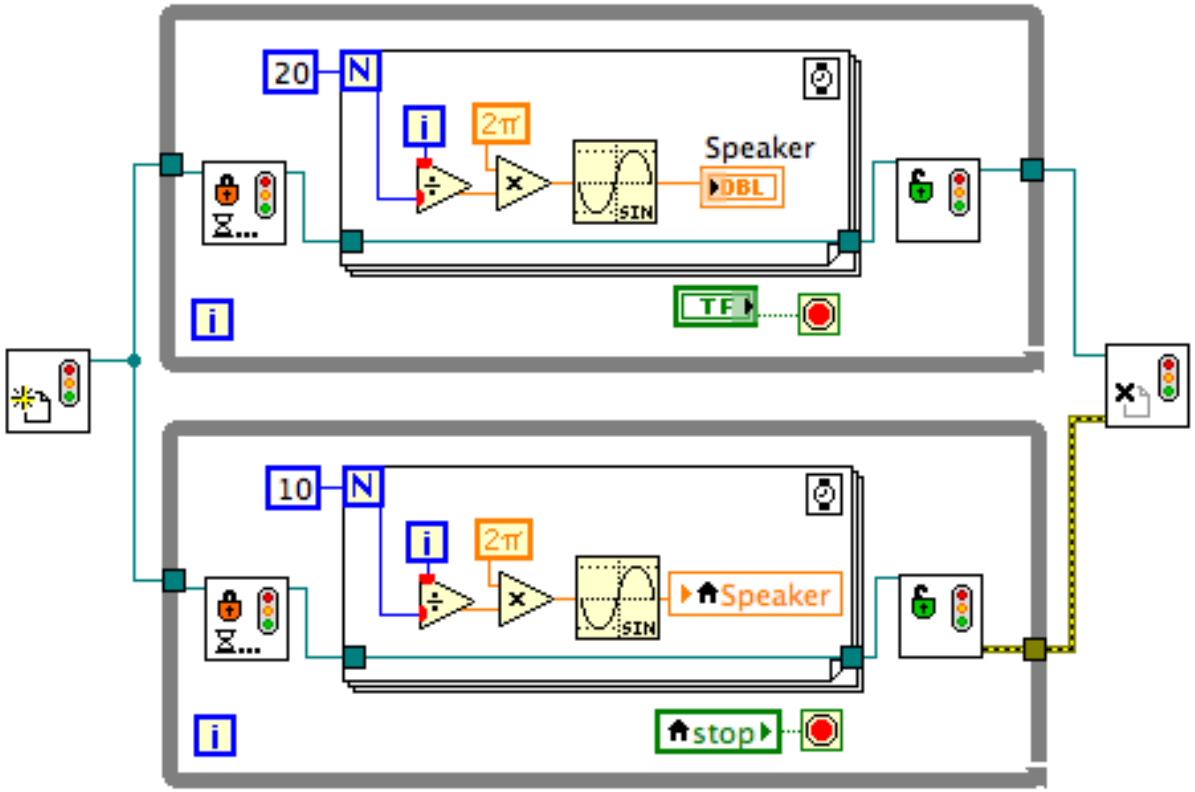


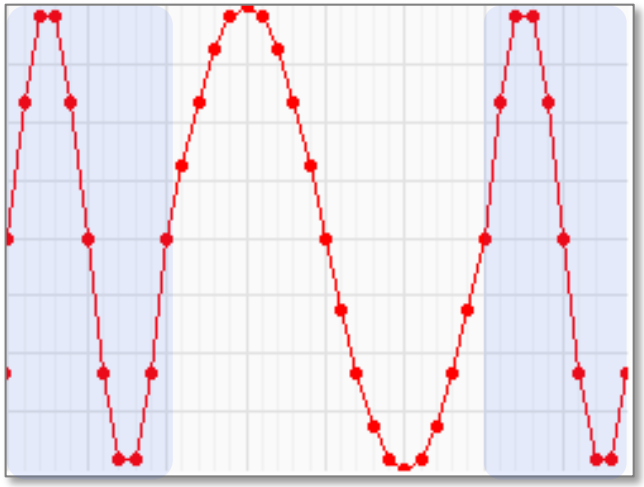
Acquire semaphore, once acquire no one can access it

Release semaphore, next in line can access

```
#include <pthread.h>
pthread_mutex_t s;


pthread_mutex_lock(&s);

for (i=0;i<10;i++)
    speaker =
sin(i/10*2*pi);

pthread_mutex_unlock(&s);
```

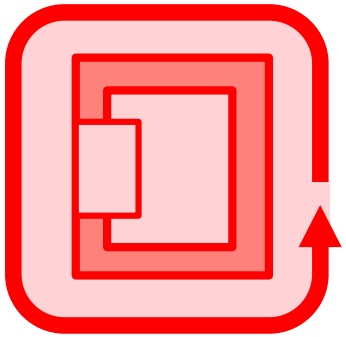
~~Race condition~~



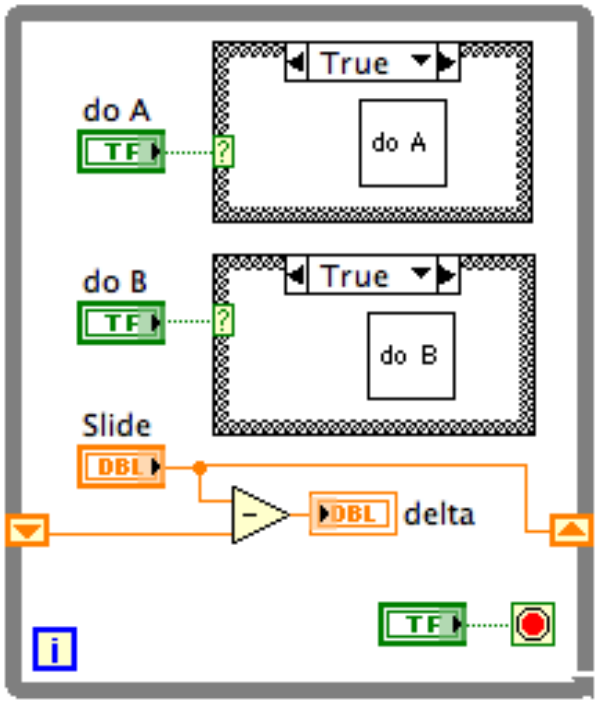


The two processes write to the simulated speaker one after the other.

Event loop



Are you actively pooling for user actions ?



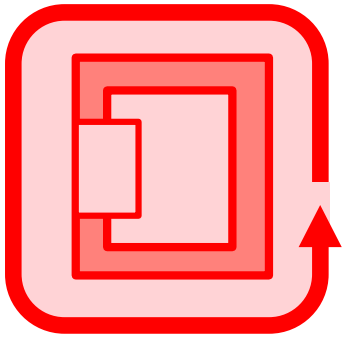
Events order ?

Did you catch all events?

How's your CPUs ?

```
While (!bStop) {
     if (do_A)
                doA();
      if (do_B)
                doB();

      delta = slide - prev_slide;
      prev_slide = slide;
}
```

Event loop



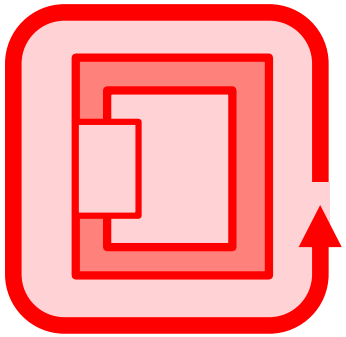
Event-driven programming

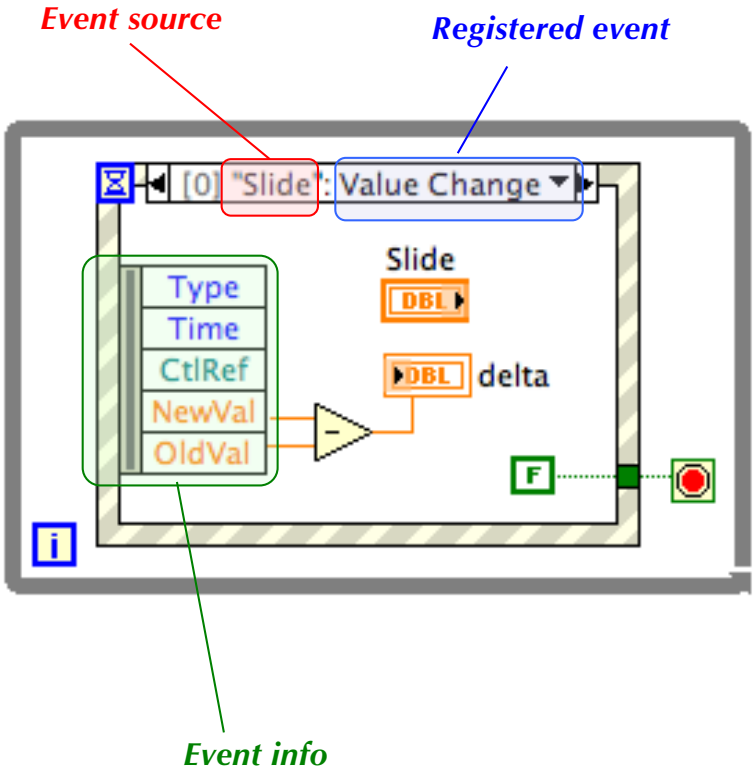
- Events (ex. Button pressed) are broadcasted by the OS (or LabVIEW)
- Registered events are captured by the event structure
- Event structure returns information about event to case
- Event structure enqueues events that occur while it's busy

Advantage

- Execution determined at run-time
- Waits for events (ex. Button pressed) to occur without consuming CPU
- Remembers order of multiple events
- Possibility to filter events before they are processed
- Possibility to define dynamic events

Event loop

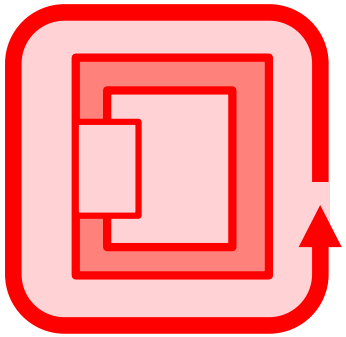


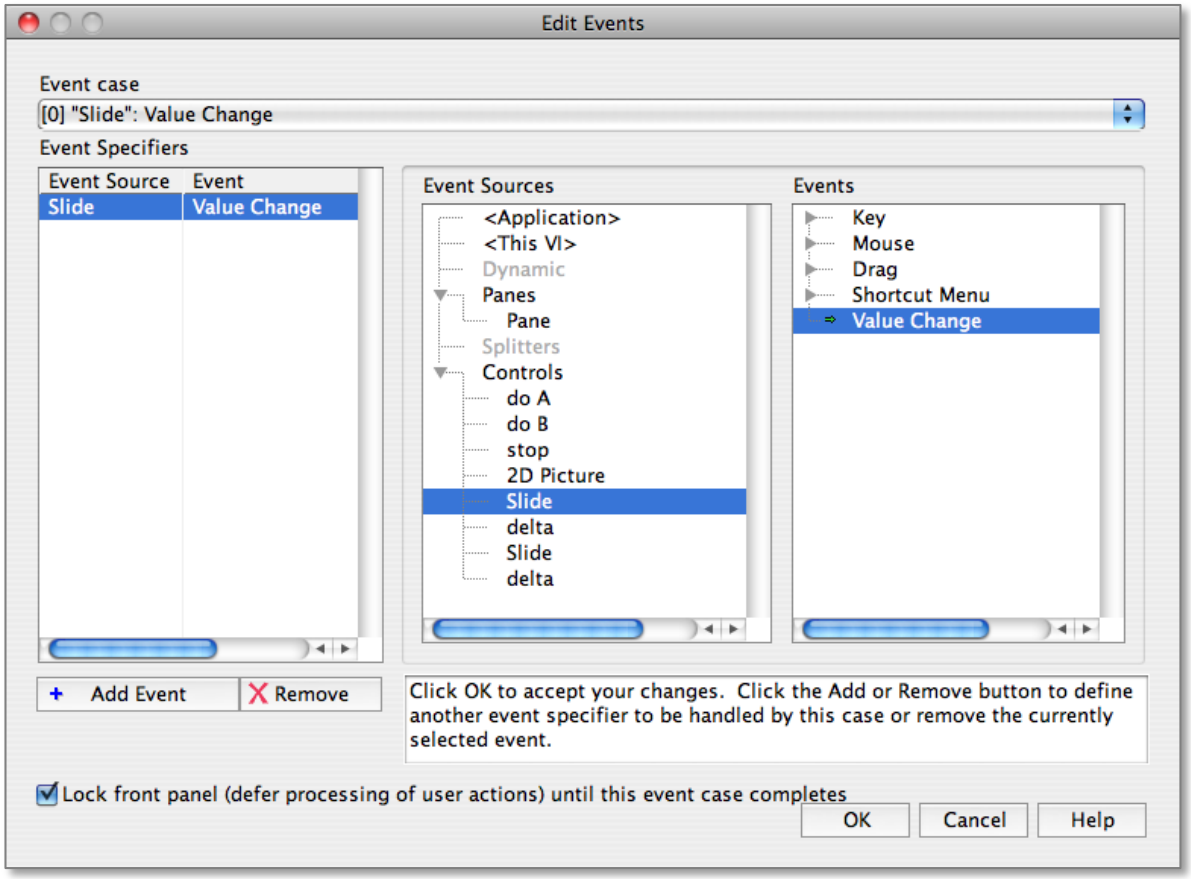


- ***Always nested within a while loop***
- ***Only 1 event structure per application!***

- Registered events are enqueued
- Blocking until event registered or timeout
- One case for each registered event
- Event information are available (left)

Event loop

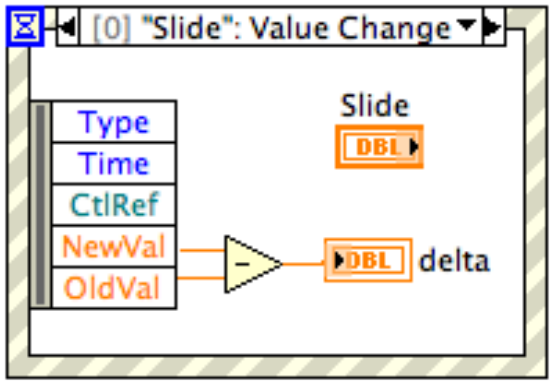




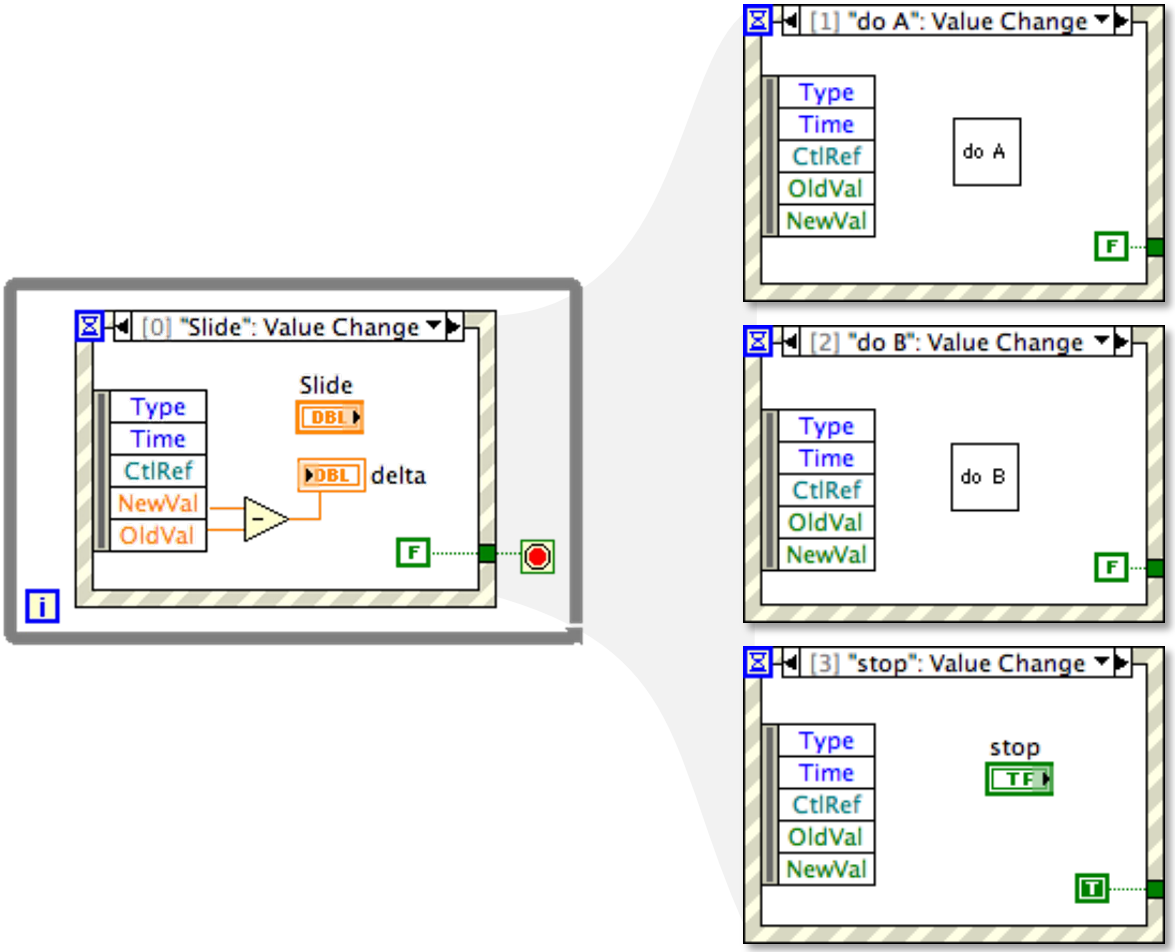
Browse controls

Browse events per control

- Green arrow: notify
- Red arrow: filter

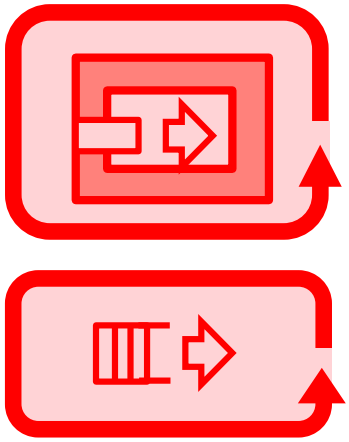


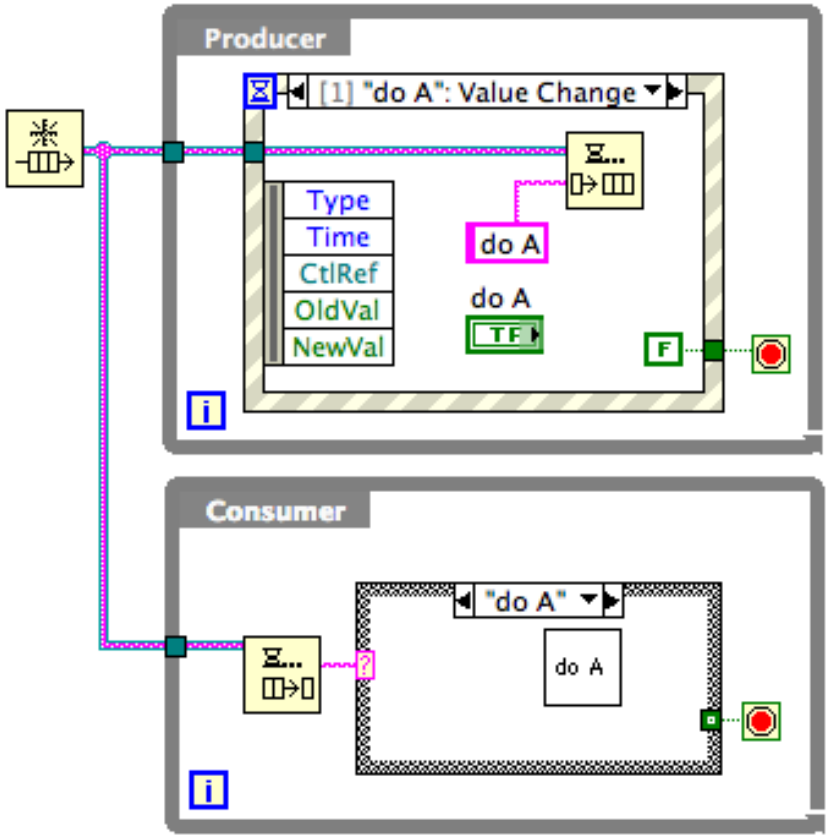
Event loop




```
While (!done) {
    event = waitnextevent();
    swtich (event.what) {
      case doA_valChng: doA();
      case doB_valChng: doB();
      case slide_valChng:
           delta = event.newVal -
                        event.OldVal;
      case stop_valChng: done=true;
}
```

Producer consumer Event loop





- Allow to defer the event associated code execution
- Like in real (complex) application

- The event loop produces events
- The consumer loop(s) handle sent events