Communication par passage de messages

Notion de canal de communication par mémoire partagée

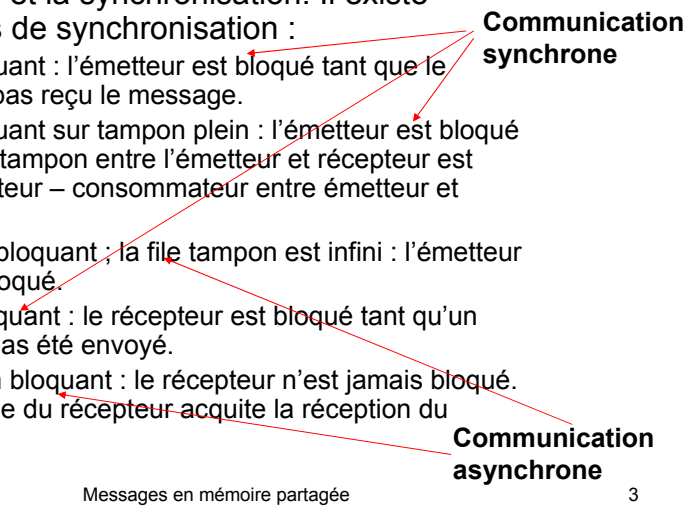
Canaux en mémoire partagée

- Envoi et réception de messages à travers un canal au lieu de lire ou écrire en mémoire partagée, protégée par des sémaphores ou par des moniteurs.
- Une classe *canal* peut être définie au niveau utilisateur pour les langages qui ne disposent pas en natif de l'objet canal.

```
channel requestChannel = new channel() ;
channel replyChannel = new channel() ;
```

flot1		flot2
requestChannel.send(request) ;	→	request = requestChannel.receive();
reply =replyChannel.receive();	←	replyChannel.send(reply);

- Un flot qui exécute un send() ou un receive() peut être bloqué ; ces opérations matérialisent à la fois la communication et la synchronisation. Il existe différents types de synchronisation :
  - Émetteur bloquant : l'émetteur est bloqué tant que le récepteur n'a pas reçu le message.
  - Émetteur bloquant sur tampon plein : l'émetteur est bloqué dès que la file tampon entre l'émetteur et récepteur est pleine (producteur – consommateur entre émetteur et récepteur).
  - Émetteur non bloquant ; la file tampon est infini : l'émetteur n'est jamais bloqué.
  - Récepteur bloquant : le récepteur est bloqué tant qu'un message n'a pas été envoyé.
  - Récepteur non bloquant : le récepteur n'est jamais bloqué. Une commande du récepteur acquite la réception du message.



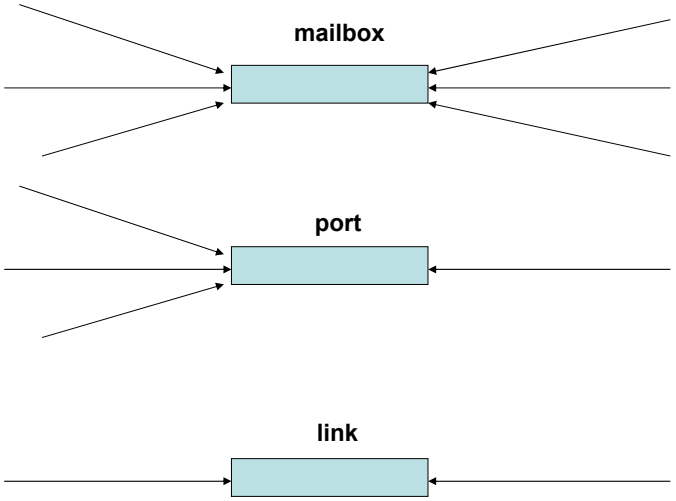
L'objet canal en Java

- Des canaux utilisateurs sont implémentés par des objets partagés.

```
Public abstract class channel {
    public abstract void send(Object m);   // envoi d'un objet message
    public abstract void send();           // signal vers un récepteur
    public abstract Object receive();      //réception d'un objet
```

- On définit trois types de canaux différentiés par le nombre de flots émetteur et receveur autorisé à accéder à un objet canal :
  - Boite_aux_lettres (mailbox) : plusieurs émetteurs – plusieurs récepteurs
  - Port : plusieurs émetteurs – un receveur
  - Lien (link) : un émetteur et un receveur

  Chaque type de canal a une version synchrone et une version asynchrone.



Exemple : une classe *mailbox* synchrone

```
Public class mailbox extends channel {
    private Object message = null ;
    private final Object sending = new Object() ;
    private final Object receiving = new Object() ;
    private final binarySemaphore sent = new binarySemaphore(0) ;
    private final binarySemaphore received = new binarySemaphore(0) ;
    public final void send(Object sentMsg) {
        if (sentMsg == null) {throw new NullPointerException(''message
    vide'');
        }
        synchronized(sending){
        message = sentMsg ;
        sent.V();                    // signal que le message est disponible
        received.P();                //attendre le signal de l'arrivée du message
        }
    }
```

```
public final void send() {
        synchronized(sending) {
                message = new Object() ;  // envoi d'un message vide
                sent.V();                 // signale que le message est dispo
                received.P() ;            //attendre l'arrivée du message
        }
}


 public final void Object receive() {
        Object receivedMessage = null ;
        synchronized(receiving) {
                sent.P();                 // attente de l'envoi d'un message
                receivedMessage = message ;
                received.V() ;            // signal à l'émetteur que le message
                                          est bien arrivé
        }
        return receivedMessage ;
   }
}
```


Les classes *link* et *port*

Les méthodes send() pour les classes ports et mailbox sont identiques
Les méthodes receive() pour les classes port et link sont identiques.

Un seul flot peut exécuter une opération receive sur un port ou un lien.
La nouvelle méthode pour mettre en œuvre plusieurs receveurs:

```
public final Object receive() {
        synchronized(receiving) {
        if (receiver == null)          //identifie le 1er thread à appeler receive
          receiver = Thread.currentThread() ;
// si currentThread() n'est pas le premier thread à appeler le récepteur, lever
d'une exception
        if(Thread.currentThread() != receiver) throw new
        InvalidLinkUsage("tentative d'accès avec de multiples récepteurs") ;
        Object receivedMessage = null ;
        sent.P() ;          // attente que le message soit expédié
        receivedMessage = message ;
        received.V() ;      // signal vers l'émetteur que le message a été reçu
        return receivedMessage ;
        }
 }
```

Exemple2 : Une classe mailbox asynchrone

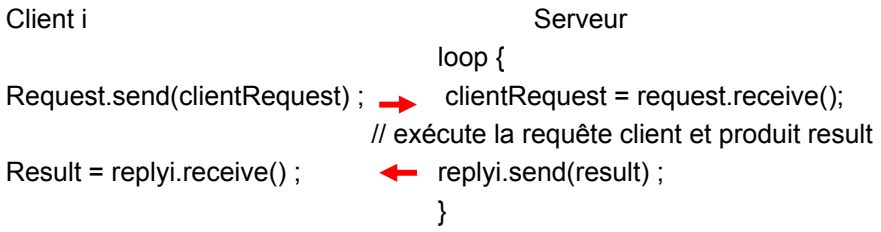
```
public final class asynchMailbox extends channel {
        private final int capacity = 100 ;
        private Object messages[ ] = new Object[capacity] ;    //tampon de messages
        private countingSemaphore messageAvailable = new countingSemaphore(0);
        private countingSemaphore slotAvailable = new countingSemaphore (capacity);
        private binarySemaphore senderMutex = new binarySemaphore(1);
        private binarySemaphore receiverMutex = new binarySemaphore(1);
        private int i = 0, out =0 ;

        public final void send(Object sent Message) {
                if (sentMessage == null) {
                throw new NullPointerException (''message null à envoyer'') ;
                }
                slotAvailable.P() ;
                senderMutex.P() ;
                messages[in] = sentMessage ;
                in = (in+1)%capacity ;
                senderMutex.V() ;
                messageAvailable.V() ;
        }
```



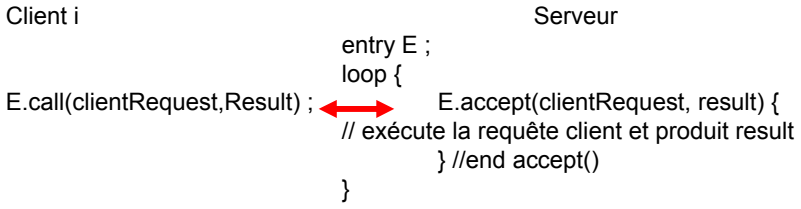
```
        public final void send() {
                        slotAvailable.P() ;
                        senderMutex.P() ;
                        messages[in] = new Object() ;
                        in = (in+1)%capacity ;
                        senderMutex.V() ;
                        messageAvailable.V() ;
                }


        public final Object receive() {
                messageAvailable.P() ;
                receiverMutex.P() ;
                Object receivedMessage = messages [out] ;
                out = (out+1) % capacity ;
                receiverMutex.V() ;
                slotAvailable.V() ;
                return receivedMessage ;
                }
        }
```

Le paradigme du rendez vous

- Utile dans un environnement client-serveur
  - Appel depuis un client d'une entrée dans une tâche serveur
  - Synchronisation entre client et serveur (Rendez vous)
  - Retour vers le client de l'information demandée.



- Un canal que le serveur peut utiliser pour recevoir les requêtes des clients et un canal pour chaque réponse aux clients.
- Spécification de ce paradigme



- Le serveur utilise un nouveau type de canal appelé *entry*

Propriétés du rendez vous

- Un seul flot peut faire *accept* d'un appel sur une entrée donnée : quand un serveur exécute l'*accept* sur l 'entrée E alors :
  - *S'il n'y a pas d'appel pendant alors le serveur attend*
  - *Si un ou plusieurs appels sont arrivés le serveur accepte l'appel et exécute le corps de l'accept ; à la fin de l'exécution du corps, l'entrée retourne vers le client l'information demandée, le client et le serveur poursuivent leurs exécutions.*
- Mécanisme de communication synchrone
- Natif en Ada ; des classes implémentant le paradigme sont possibles en Java.

L'activation sélective de tâches

- Le concept d'activation sélective permet à une tâche de choisir un rendez vous parmi plusieurs possibles. La sélection peut dépendre de conditions associées à chaque alternative.

```
entry E ; entry F ;
loop
        select
          E.accept(client1Request, result1) {
          result1 = …;
          }
        or
        F.accept(client2request,  result2) {
        result2 = … ;
        end select ;
end loop ;
```

```
entry E ;
loop
        select
        E.accept(signal_reveil);
        or
        delay 30.0 ;
        stop_the_train ;
        end select ;
end loop ;
```

```
entry E ;
loop
        select
        E.accept(signal_reveil);
        else
        delay 30.0 ;
        stop_the_train ;
        end select ;
end loop ;
```

```
entry E ; entry F ;
loop
        select
        E.accept(demarrer);
        or
        terminate ;
        end select ;
end loop ;
```


Exemple avec le producteur-consommateur

Déclarations

```
entry deposer ;
entry retirer ;
```

Producteurs

```
{
Envoi_message : Message ;

Envoi_message = preparation_message ;
buffer.deposer(Envoi_message) ;
}
```

Consommateurs

```
{
retrait_message : Message ;

buffer.retirer(Retait_message) ;
exploitation_message(Retrait_message);
}
```

Synchro

```
{
int tampon[ ]  = new int[n] ;
int i=0, j=0, nombre = 0;
serveur {
 loop {
   select {
        when nombre < n {
        deposer.accept(M);
        tampon[i] = M ;
        }
        i = (i+1) % n ;
        nombre = nombre +1 ;
   or
        when nombre >0 {
        retirer.accept(L) ;
        L = tampon[j] ;
        }
        j = (j +1) % n ;
        nombre = nombre -1 ;
  or
        terminate ;
    }
   }
 }
}
```

Exemple Lecteurs-Rédacteurs (syntaxe Ada)

Une tâche synchro, N tâches Proc.

```
procedure Main is
  task Synchro is
    entry AvantLire ;
    entry ApresLire ;
    entry AvantEcrire ;
    entry ApresEcrire ;
  end Synchro ;
  task body Synchro is
    NL : Integer := 0 ;
  begin – le 1er accès est pour écrire
    accept AvantEcrire ; accept ApresEcrire ;
    loop
      select
          when NL = 0 =>
             accept AvantEcrire ; Accept ApresEcrire;
      or
          accept AvantLire ; NL = NL+1 ;
      or
          when NL >0 => accept ApresLire ; NL = NL-1 ;
      or
          terminate
      end select
    end loop
  end synchro
```


Exemple Lecteurs-Rédacteurs (suite)

Une tâche synchro, N tâches Proc.

```
task type Proc ;
Appli : array (1..N) of Proc ;

task body Proc is
begin

-- la tâche agissant comme rédacteur
Synchro.avantEcrire ;
EcritureDansLeFichier ;
Synchro.ApresEcrire ;

-- la tâche agissant comme lecteur
Synchro.AvantLire ;
LectureDansLeFichier ;
Synchro.ApresLire ;

end Proc ;

begin
null ;
end Main ;
```


Cours suivant