Septième partie

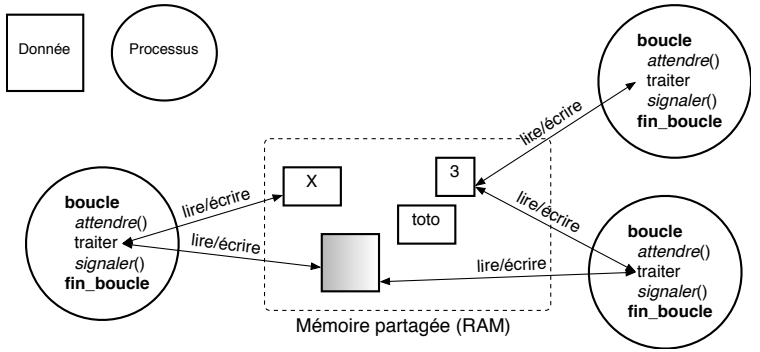
Processus communicants

Contenu de cette partie

- Panorama des modèles de programmation concurrente
- Présentation et caractéristiques du modèle des processus communicants
- Outils Ada pour la programmation concurrente
  - Le modèle des processus communicants en Ada : tâches et rendez vous
  - Démarche de conception d'applications concurrentes en Ada
    - Transposition de la démarche vue dans le cadre de la mémoire partagée (moniteurs)
    - Extension tirant parti des possibilités de contrôle fin offertes par Ada

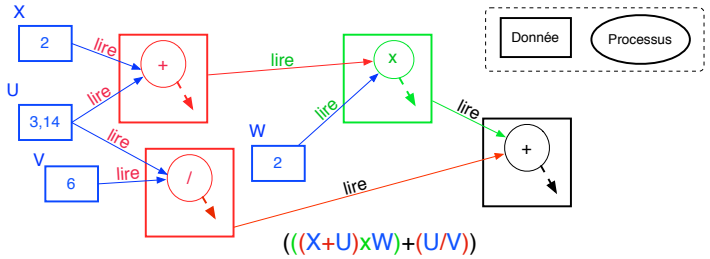
Plan

Modèles d'interaction : mémoire partagée



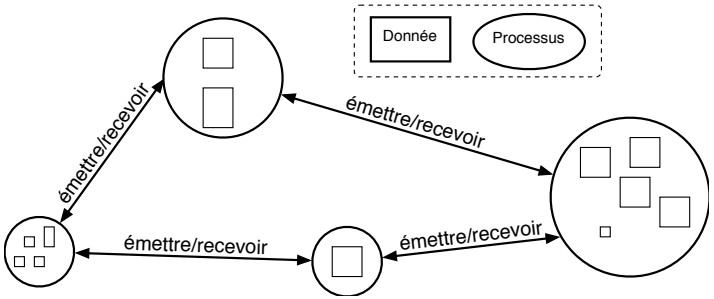
- Communication implicite
  - résulte de l'accès et de la manipulation des variables partagées
  - l'identité des processus n'intervient pas dans l'interaction
- Synchronisation explicite (et nécessaire)
- Architectures/modèles cibles
  - multiprocesseurs à mémoire partagée,
  - programmes multiactivités

Modèles d'interaction : données actives



- Chaque donnée est évaluée par un processus dédié
- Communication = lecture
- Synchronisation = attente de disponibilité des données
- Architectures/modèles cibles
  - parallélisme massif, à grain fin :
    - programmation par flots de données (streams Java 8),
    - programmation fonctionnelle (Scala)
  - objets immuables

Modèles d'interaction : processus communicants



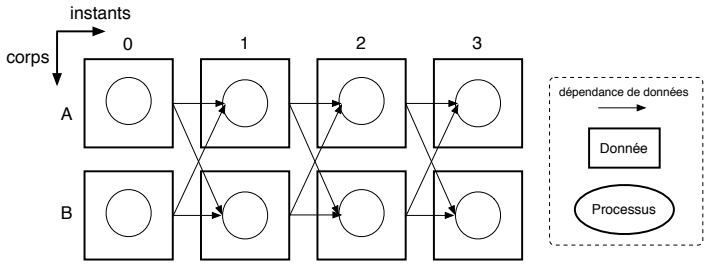
- Données encapsulées par les processus
- Communication nécessaire, explicite : échange de messages
  - Programmation et interactions plus lourdes
  - Visibilité des interactions → possibilité de trace/supervision
- Synchronisation implicite : attente de disponibilité des messages
- Les processus déterminent la granularité du parallélisme
- Architectures/modèles cibles
  - systèmes répartis : sites distants, reliés par un réseau
  - moniteurs, tâches Ada, API messages : sockets, MPI, JMS...

Exemple : simulation discrète de corps en interaction

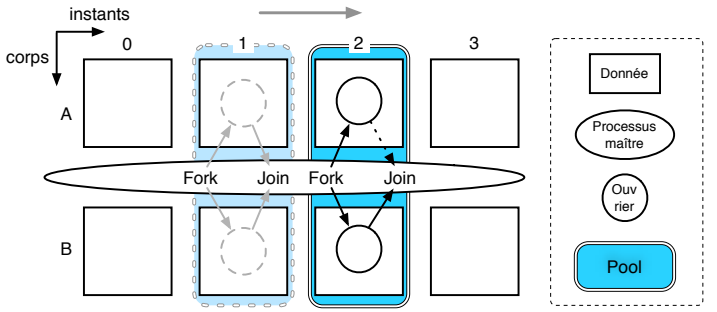
Problème des N corps

Calculer la position de N corps C_1, $C_2 \ldots C_N$ à une suite d'instants successifs t_0, t_1, $\ldots t_k$. La position d'un corps à l'instant t_i est déterminée par la position de l'ensemble des corps à l'instant t_{i-1}.

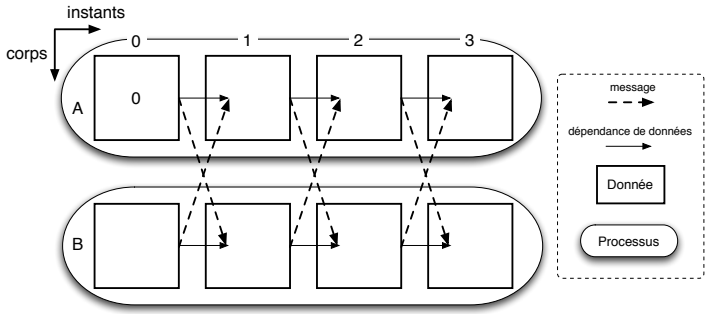
- **Parallélisme/résultat** (Flots de données/Structures actives)
  - matrice corps x instants
  - un processus d'évaluation par élément de la matrice



- **Parallélisme planifié** (Structures partagées + pool d'ouvriers)
  - données partagées : positions aux instants précédents
  - un maître coordonne les ouvriers (schéma fork/join)
  - N ouvriers : chaque ouvrier évalue l'élément de la colonne courante qui lui est affecté
  - progression colonne par colonne : le maître attend qu'une colonne soit complètement évaluée (`join`), avant de lancer l'évaluation de la suivante (`fork`)
  - peut être vu comme une réalisation du parallélisme résultat



- **Parallélisme/spécialistes** (Messages/Clients-Serveur)
  - un processus par corps
  - accès aux positions passées des autres corps → messages



Plan

Processus communicants

Synchronisation obtenue via des primitives de communication bloquantes : envoi (bloquant) de messages / réception bloquante de messages

- Communicating Sequential Processes (CSP) / Calculus of Communicating Systems (CCS) / π-calcul
- Ada

Les principes détaillés des échanges et leur utilisation pour développer des applications sont vus dans l'UE « intergiciels ».
On intéresse ici avant tout à la synchronisation.

Opérations

- Emettre(message, destination)
- Recevoir(message, source)

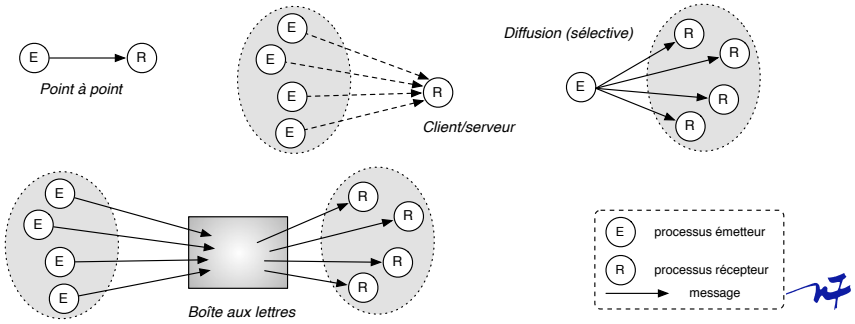
Synchronisation liée aux opérations *du modèle des proc. communicants*

- Réception bloquante : attente d'un message
- L'émission peut être, selon les modèles
  - non bloquante (comm. asynchrone) : l'émission termine dès la prise en charge du message par le medium de communication.
  - bloquante (communication synchrone) : l'émetteur doit attendre jusqu'à la réception effective du message
    → rendez-vous entre l'émetteur et le destinataire

Désignation des activités sources et destinataires

La source ou le destinataire peuvent être une ou plusieurs activités → différents schémas de communication :

- communication directe, « point à point » (1-1) entre activités
- diffusion (sélective) (1-n)
- communication indirecte : via une « boite aux lettres » (un « canal ») attachée à un processus (n-1 : port), ou partagée (n-m)



Exemple : le langage CSP (Communicating Sequential Processes)

Caractéristiques

- émission bloquante
- échanges de messages via des canaux explicitement désignés, attachés ou non à un processus.

Opérations

- envoi d'un message `msg` sur le canal `c` : $c!msg$
- réception d'un message `msg` sur `c` : $c?msg$

Rendez-vous

L'émission et la réception sont bloquantes : chaque communication est un rendez-vous entre un proc. émetteur et un proc. récepteur :
$proc\ p\ \{c!m\}\ \|\ proc\ q\ \{c?m\}$

Alternative

Définit un choix à effectuer parmi un ensemble de communications possibles :

- choix entre deux réceptions : $c_1?m_1[]c_2?m_2$
- choix entre deux émissions : $c_1!m_1[]c_2!m_2$
- une alternative peut comporter plus de deux choix (sic)
- une alternative peut comporter des émissions et des réceptions
- les choix peuvent être gardés par une condition g : $g \rightarrow s?m$

Evaluation

Les choix dont les gardes sont fausses sont éliminés. Ensuite :

- Aucun des choix restants n'est possible immédiatement → attendre que l'un d'eux le devienne
- Un seul choix possible → le faire
- Plusieurs choix possibles → sélection non-déterministe (arbitraire)

Synchronisation des processus communicants

- La démarche de résolution des problèmes de synchronisation vue pour le modèle de la mémoire partagée est basée sur la définition et le contrôle de l'interaction avec un objet partagé.
- Le modèle des processus communicants fournit une base à l'encapsulation des données/ressources :

  le seul moyen d'accéder aux données locales à un processus est d'échanger des messages avec ce processus

→ La démarche vue pour le modèle de la mémoire partagée se transpose simplement dans le contexte des processus communicants :
Définir un processus arbitre (ou « serveur »), encapsulant l'objet partagé, pour contrôler et réaliser les opérations sur celui-ci.

Mise en œuvre d'un processus arbitre pour un objet partagé

Interactions avec l'objet partagé (protocole requête/réponse) :
Pour chaque opération $\mathcal{O}p$,

- émettre un message de requête vers l'arbitre
- attendre le message de réponse de l'arbitre
  ($\Rightarrow$ se synchroniser avec l'arbitre)

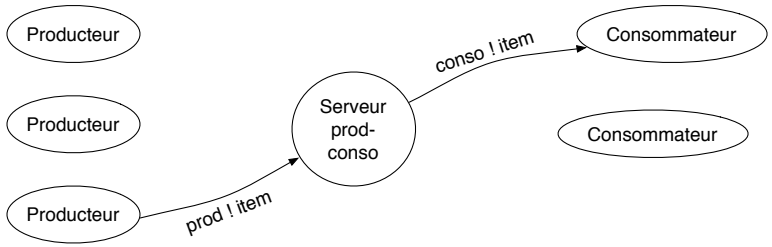
$\rightarrow$ en CSP :

- messages échangés via un canal $\mathcal{C}\_\mathcal{O}p$ associé à l'opération $\mathcal{O}p$,
- interaction = rendez-vous entre le client et l'arbitre, sur $\mathcal{C}\_\mathcal{O}p$

Schéma de fonctionnement de l'arbitre

- L'arbitre exécute une boucle infinie contenant une alternative
- Cette alternative possède une branche par opération fournie
- Chaque branche est gardée par la condition d'acceptation de l'opération (suivie de l'attente du rendez-vous correspondant)

Exemple 1 (CSP) : producteurs/consommateurs (1/3)



Objet partagé :

Tampon borné de taille N

Opérations sur le tampon partagé :

- Déposer $\rightarrow$ canal associé : *prod* (sens producteur $\rightarrow$ serveur)
- Retirer $\rightarrow$ canal associé : *conso* (sens serveur $\rightarrow$ consommateur)

Producteurs/consommateurs (2/3)

Producteur

```
Processus producteur
  boucle
    ...
    item := ...
    prod!item
    ...
  finboucle
```

Consommateur

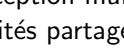
```
Processus consommateur
  boucle
    ...
    conso?item
    utiliser item
    ...
  finboucle
```


Producteurs/consommateurs (3/3)

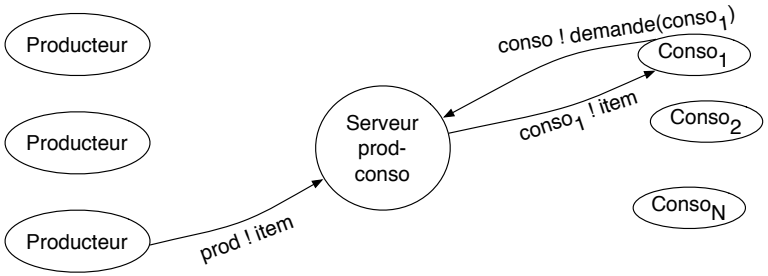
Activité de synchronisation (serveur)

```
Processus prod-conso
  var
    int nbocc := 0;
    Item m;
    File<Item> tampon;
  boucle
     nbocc < N → prod?m; nbocc++; tampon.ranger(m);
  □
     nbocc > 0 → ; conso!tampon.extraire(); nbocc--;
  finboucle
```

- Nécessite l'alternative mixte
  (plus complexe à réaliser que l'alternative en réception)
- Nécessite la réception multiple
  (plusieurs activités partagent le même canal *conso*)



[Producteurs/consommateurs : variante (1/3)]



- Un canal *prod* pour les demandes de dépôt
- Un canal *conso* pour les demandes de retrait
- Pour chaque activité $Conso_i$ demandant un retrait, un canal $conso_i$, pour la réponse à la demande de retrait

[Producteurs/consommateurs : variante (2/3)]

Producteur

```
Processus producteur
  boucle
    ...
    item := ...
    prod!item
    ...
  finboucle
```

Consommateur

```
Processus consommateur
Canal moi
  boucle
    ...
    conso!demande(moi);
    moi?item;
    utiliser item
  finboucle
```

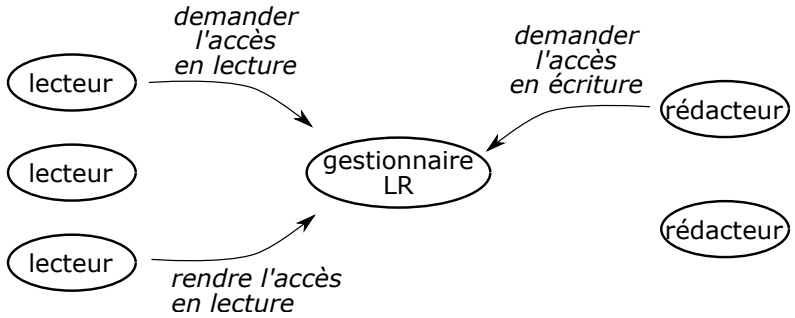

[Producteurs/consommateurs : variante (3/3)]

Processus de synchronisation

```
Processus prod-conso
  var
    int nbocc := 0;
    Item m;
    File<Item> tampon;
    Canal c;
  boucle
     nbocc < N → prod?m; nbocc++; tampon.ranger(m)
  □
     nbocc > 0 → conso?demande(c); m := tampon.extraire();
                 c!m; nbocc--;
  finboucle
```

Nécessite des variables *Canal*, pouvant être passées en paramètre.

Exemple 2 (CSP) : lecteurs/rédacteurs (1/3)



- Un canal pour chaque opération : *DE*, *TE*, *DL*, *TL*
- Émission bloquante ⇒ contrôle des requêtes par l'arbitre : un message n'est accepté que si l'état de l'objet partagé l'autorise

Lecteurs/rédacteurs (2/3)

Utilisateur

```
Processus utilisateur
  boucle
    DL!_;  // demander lecture
    ...
    TL!_;  // terminer lecture
    ...
    DE!_;  // demander écriture
    ...
    TE!_;  // terminer écriture
  finboucle
```

Lecteurs/rédacteurs (3/3)

Processus de synchronisation

```
Processus SynchroLR
  var
    int nblec = 0;
    boolean ecr = false;
  boucle
     nblec = 0 ∧ ¬ecr → DE?_; ecr := true;
  □
     ¬ecr → DL?_; nblec++;
  □
     TE?_; ecr := false;
  □
     TL?_; nblec--;
  finboucle
```

Plan

Modèle Ada

Modèle orienté vers une organisation en termes de processus communicants

- Application (processus lourd) = ensemble de *tâches* (activités) concurrentes
- Interactions entre tâches privilégiant le schéma client-serveur.
- Contrôle fin de l'ordonnancement des requêtes au niveau du serveur.
- Transposable simplement à un environnement réparti.

Déclaration d'une tâche

Schéma d'interaction privilégié : client-serveur
→ toute tâche exporte une interface (*points d'entrée*)

Exemple

```
task ProdCons is
   entry Deposer (msg: in T);
   entry Retirer (msg: out T);
end ProdCons;
```

Syntaxe

```
task [type] <nom> is
   { entry <point d'entrée> (<param formels>); }+
end <nom> ;
```


Modularité → interface et implémentation (*corps*) déclarées séparément

Exemple

```
task body ProdCons is
   Libre : integer := N;
begin
     ⋮
     accept Deposer (msg : in T) do
          deposer_dans_tampon(msg);
     end Deposer;
     Libre := Libre - 1;
     ⋮
end ProdCons;
```

Syntaxe

```
task body <nom> is
  [ < déclaration des variables locales > ]
begin
   [ < instructions > ]
end < nom > ;
```

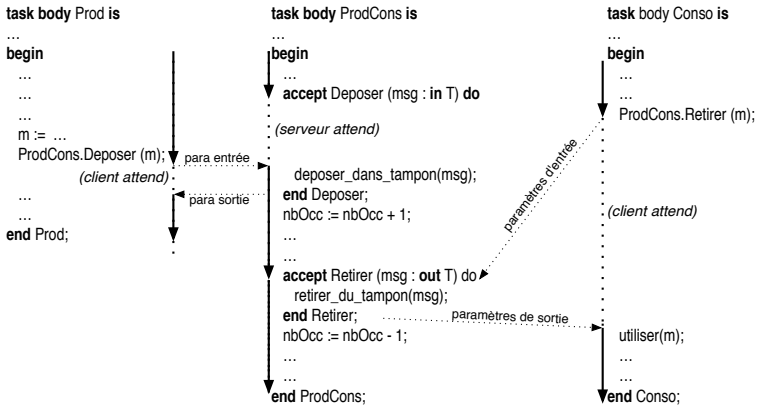

Création (activation) des tâches

Une tâche peut être activée :

- statiquement : une `task T` déclarée directement, ou comme instance d'un type de tâche, est créée au démarrage du programme, avant l'initialisation des tâches qui utilisent `T.entry`.
- dynamiquement :
  - déclaration par `task type T`
  - activation par allocation : `var t is access T := new T;`
  - possibilité d'activer plusieurs tâches d'interface `T`.

Interaction client/serveur : rendez-vous Ada

- tâche cliente : requête de service = appel d'un point d'entrée
  Exemple : `tampon.déposer(ln)`
- tâche serveur : prise en charge d'une requête → instruction accept



Protocole

- Si un client demande un rendez-vous alors que le serveur n'est pas prêt à l'accepter, le client se bloque en attente de l'acceptation.
- Si un serveur est prêt à accepter un rendez-vous et qu'il n'y a pas de demandeur, il se bloque.
- L'appelant est bloqué pendant l'exécution du corps du rendez-vous.

Côté client : appel d'entrée (≈ demande de rendez-vous)

```
<nom tâche>.<nom d'entrée> (<param effectifs>);
```

Similaire à un appel de procédure.

Exemple

```
tampon.déposer(ln)
```

Acceptation (côté serveur)

```
accept <nom d'entrée> (<param formels>)
     [ do
         { <instructions> }+
       end <nom d'entrée> ]
```

Exemple

```
accept Deposer (msg : in T) do
     deposer_dans_tampon(msg);
     Libre := Libre - 1;
end Deposer;
```

Remarques

- les `accept` ne peuvent figurer que dans le corps des tâches
- `accept` sans corps → synchronisation « pure »
- une file d'attente (FIFO) est associée à chaque entrée
  l'attribut `'COUNT` donne pour chaque entrée, la taille de la file
- la gestion et la prise en compte des appels diffèrent par rapport aux moniteurs
  - la prise en compte d'un appel au service est déterminée par le serveur → serveur « actif »
  - plusieurs appels à un même service peuvent déclencher des traitements différents
  - le serveur peut être bloqué, tandis que des clients attendent
- échanges de données :
  - lors du début du rendez-vous, de l'appelant vers l'appelé ;
  - lors de la fin du rendez-vous, de l'appelé vers l'appelant.

Plan

Construction d'une tâche serveur

Alternative gardée (`select`)

Ada permet à un serveur d'attendre un appel à un point d'entrée quelconque parmi un ensemble donné de points d'entrée.

```
select
    when Libre > 0 =>
       accept Deposer (msg : in T) do
          deposer_dans_tampon(msg);
       end Deposer;
        ...
  or
    when Libre < N =>
       accept Retirer (msg : out T) do
          msg := retirer_du_tampon();
       end Retirer;
        ...
  or ...
end select;
```

Evaluation du `select`

- évaluer les conditions vraies → branches ouvertes
- branche ouverte et appel en attente → branche franchissable
- si des branches sont franchissables, choisir (arbitrairement) une branche parmi celles-ci
- sinon attendre qu'une branche ouverte soit franchissable

Compléments

- `when` omis = true
- clauses/branches particulières
  - clause `else` possible, comme dernière possibilité d'un `select`
    - exécutée si aucune branche n'est franchissable
    - évite le blocage en réception
    - clause `else` omise et aucune branche ouverte → exception `program_error`
  - `when <condition> => delay <nbDeSecondes>` exécutée si ouverte et aucune branche n'est franchissable à l'issue du délai
  - `when <condition> => terminate` termine le serveur en l'absence de tâches clientes potentielles.

Structure d'un serveur

Serveur ≪ type ≫

`select` dans une boucle sans fin
→ tâche (passive) dédiée à la réalisation de services

Remarque

Similitude entre cette structure de tâche et les moniteurs

→ possibilité de réutiliser/transposer directement directement la démarche de conception et les techniques d'optimisation vues pour les moniteurs.

Terminaison du serveur

Une tâche T est potentiellement appelante de T' si

- T' est une tâche statique et le code de T contient au moins une référence à T',
- ou T' est une tâche dynamique et (au moins) une variable du code de T référence T'.

Une tâche se termine quand :

- elle atteint la fin de son code,
- ou elle est bloquée en attente de rendez-vous sur un `select` avec clause `terminate` et toutes les tâches potentiellement appelantes sont terminées.

Exemple 1 : producteurs/consommateurs

Client : utilisation

```
begin
   -- engendrer le message m1
   ProdCons.Deposer (m1);
   -- ...
   ProdCons.Retirer (m2);
   -- utiliser m2
end
```


```
task body ProdCons is
   Libre : integer := N;
begin
   loop
      select
         when Libre > 0 =>
            accept Deposer (msg : in T) do
               deposer_dans_tampon(msg);
            end Deposer;
            Libre := Libre - 1;
      or
         when Libre < N =>
            accept Retirer (msg : out T) do
               msg := retirer_du_tampon();
            end Retirer;
            Libre := Libre + 1;
      or
         terminate;
      end select;
   end loop;
end ProdCons;
```

Exemple 2 : allocateur de ressources multiples

Un système comporte des ressources critiques c'est-à-dire non partageables et non préemptibles, comme des pages mémoire. Le service d'allocation de ressources multiples permet à un processus d'acquérir par une seule action plusieurs ressources. L'exemple ne traite que la synchronisation et ne présente pas la gestion effective des (identifiants de) ressources.

Déclaration du serveur

```
task Allocateur is
   entry demander (nbDemandé: in natural;
                   id : out array of RessourceId);
   entry rendre (nbRendu: in natural;
                 id : in array of RessourceId);
end Allocateur;
```


```
task body Allocateur is
   nbDispo : integer := N;
begin
   loop
      select
          accept Demander (nbDemandé : in natural) do
              while nbDemandé > nbDispo loop
                  accept Rendre (nbRendu : in natural) do
                      nbDispo := nbDispo + nbRendu;
                  end Rendre;
              end loop;
              nbDispo := nbDispo - nbDemandé;
          end Demander;
      or
          accept Rendre (nbRendu : in natural) do
              nbDispo := nbDispo + nbRendu;
          end Rendre;
      or
          terminate;
      end select;
   end loop;
end Allocateur;
```

Extension de la méthodologie : définition d'un automate

Idée

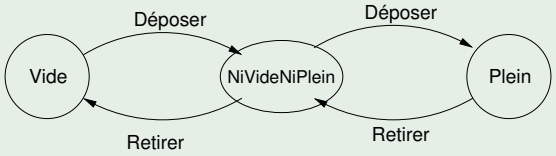
Affiner la définition des utilisations cohérentes d'un objet partagé par un ensemble de processus concurrents.

- Démarche moniteurs basée sur la définition de l'ensemble des états possibles (invariant du moniteur)
- Compléter cette définition par une fonction de transition, précisant, à partir de chaque état possible, quelles sont les actions (transitions) possibles, et quel est l'état résultat
  - → définition d'un automate
  - → le serveur traite les requêtes conformément à cet automate

Démarche de construction de l'automate

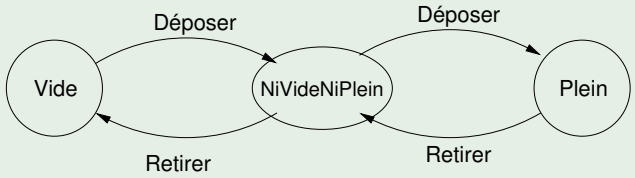
- identifier les états de l'objet partagé géré par le serveur
- pour chaque état, identifier les rendez-vous acceptables
- un rendez-vous accepté change (éventuellement) l'état

Producteurs/Consommateurs à 2 cases




```
task body ProdCons is
   type EtatTampon is (Vide, NiVideNiPlein, Plein);
   etat : EtatTampon := Vide;
begin
   loop
     case etat is
     when NiVideNiPlein =>
            select
               accept Deposer (msg : in T) do
                  deposer_dans_tampon(msg);
               end Deposer;
               etat := Plein;
            or
               accept Retirer (msg : out T) do
                  msg := retirer_du_tampon();
               end Retirer;
               etat := Vide;
            end select;
```

Producteurs/Consommateurs à 2 cases (suite)

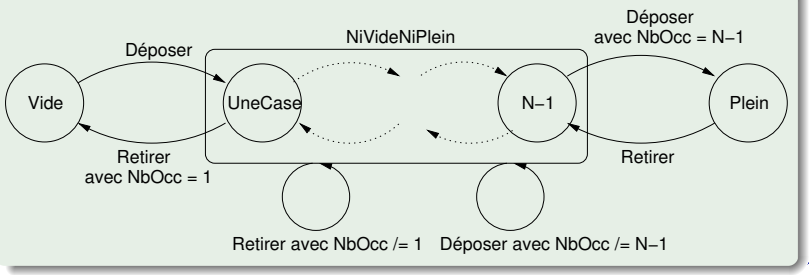



```
   --
     when Vide =>
           accept Deposer (msg : in T) do
              deposer_dans_tampon(msg);
           end Deposer;
           etat := NiVideNiPlein;
     when Plein =>
           accept Retirer (msg : out T) do
              msg := retirer_du_tampon();
           end Retirer;
           etat := NiVideNiPlein;
     end case;
   end loop;
end ProdCons;
```


Automate paramétré

Un ensemble d'états peut être représenté comme un état *paramétré*
Les valeurs du paramètre différenciant les états de l'ensemble peuvent être utilisées pour étiqueter les transitions, lorsqu'elles diffèrent selon l'état de l'ensemble.

Producteurs/Consommateurs à N cases



Exercice

Réaliser un serveur (basé sur un automate) gérant les accès à un fichier partagé selon le schéma lecteurs rédacteurs, avec priorité aux lecteurs.

Indication : automate développé pour le schéma L/R, sans priorités