Plan

1. Processus communicants
   - Principes
   - Désignation, alternatives
   - Architecture d'une application parallèle
2. Communication synchrone – CSP/CCS/Go
   - Principes
   - Recherche concurrente
   - Exemples d'objets de synchronisation
3. Rendez-vous étendu – Ada
   - Principe du rendez-vous
   - Mise en œuvre en Ada
   - Méthodologie par machine à états

Modèle Ada

Intérêt

- Modèle adapté à la répartition, contrairement aux sémaphores ou aux moniteurs, intrinsèquement centralisés.
- Similaire au modèle client-serveur.
- Contrôle plus fin du moment où les interactions ont lieu.

Vocabulaire : tâche = activité

Principe du rendez-vous

- Une tâche possède des points d'entrée de rendez-vous.
- Une tâche peut :
  - demander un rendez-vous avec une autre tâche désignée explicitement ;
  - attendre un rendez-vous sur un (ou plusieurs) point(s) d'entrée.
- Un rendez-vous est dissymétrique : tâche appelante ou cliente vs tâche appelée ou serveur.
- 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.

Rendez-vous – client en premier ♪♪♪

Rendez-vous – serveur en premier

♪♪♪

Principe du rendez-vous

- 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 indique qu'il est prêt à accepter un rendez-vous et qu'il n'y a pas de demandeur, il se bloque.
- En outre, l'appelant est bloqué pendant l'exécution du corps du rendez-vous.

Important : il est impossible d'accepter/refuser un rendez-vous selon la valeur des paramètres.

Déclaration d'une tâche

Déclaration

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

Exemple

```
task X is
    entry A;
    entry B (msg : in T);
    entry C (x : out T);
    entry D (a : in T1; b : out T2);
end X
```

Appel de rendez-vous

Appel de rendez-vous

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

Syntaxe identique à un appel de procédure, sémantique bloquante.

Exemple

```
X.A;
X.D(x,y);
```

Acceptation d'un rendez-vous

Acceptation

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

Exemple

```
task body X is
begin
  loop
      ...
     accept D (a : in Natural; b : out Natural) do
         if a > 6 then b := a / 4;
         else b := a + 2; end if;
     end D;
  end loop;
end X;
```

Acceptation parmi un ensemble

Alternative gardée

```
select
    when C1 =>
        accept E1 do
            ...
        end E1;
  or
    when C2 =>
        accept E2 do
            ...
        end E2;
  or
    ...
end select;
```

Producteurs/consommateurs

Déclaration du serveur

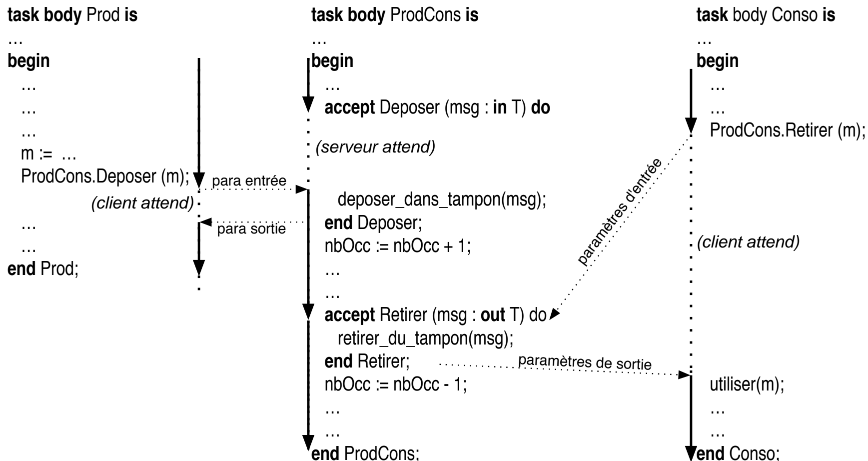
```
task ProdCons is
   entry Deposer (msg: in T);
   entry Retirer  (msg: out T);
end ProdCons;
```

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;
      end select;
   end loop;
end ProdCons;
```

Producteurs/consommateurs – un exemple d'exécution



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.
- `rdv'count` (attribut des entrées) donne le nombre de clients en attente sur une entrée donnée.
- 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 ;
  - plusieurs appels à un même service peuvent déclencher des traitements différents ;
  - le serveur peut être bloqué, tandis que des clients attendent.

Allocateur de ressources

Un système comporte des ressources critiques c'est-à-dire non partageables et non préemptibles, comme les pages mémoire. L'allocateur de ressources est un service qui permet à un processus d'acquérir par une seule action plusieurs ressources. On ne s'intéresse qu'à la synchronisation et on ne s'occupe pas de 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;
```

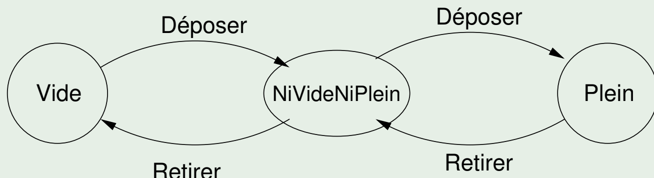
```
task body Allocateur is
   nbDispo : integer := N;
begin
   loop
      select
         accept Demander (nbDemandé : in natural) do
            while nbDemandé > nbDispo loop
               accept Rendre (nbRendu : in natural) do
                  nbDispo := nbDispo + nbRendu;
               end Rendre;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

Méthodologie par machine à états

Construire un automate fini à états :

- identifier les états du système
- un état est caractérisé par les rendez-vous acceptables
- un rendez-vous accepté change (éventuellement) l'état

Producteurs/consommateurs à 2 cases



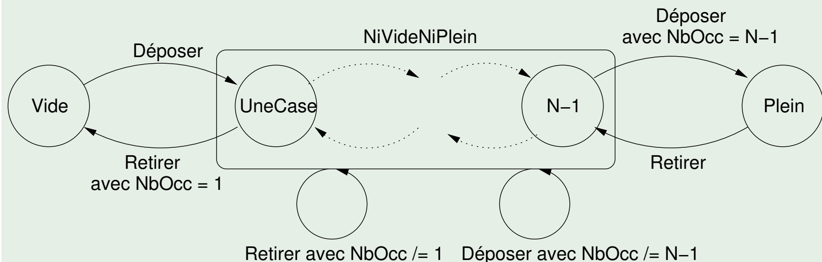
```
task body ProdCons is
   type EtatT is (Vide, NiVideNiPlein, Plein);
   etat : EtatT := Vide;
begin
   loop
      if etat = Vide then
            select
               accept Deposer (msg : in T) do
                  deposer_dans_tampon(msg);
               end Deposer;
               etat := NiVideNiPlein;
            end select;
      elsif etat = NiVideNiPlein then
            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;
```

```
    elsif etat = Plein then
        select
            accept Retirer (msg : out T) do
                msg := retirer_du_tampon();
            end Retirer;
            etat := NiVideNiPlein;
        end select;
    end if;
  end loop;
end ProdCons;
```

Automate paramétré

Représenter un *ensemble d'états* comme un unique état *paramétré*. Les valeurs du paramètre différenciant les états de l'ensemble peuvent être utilisées pour étiqueter les transitions.

Producteurs/consommateurs à N cases



Lecteurs/rédacteurs

Lecteurs/rédacteurs

Libre
DL
DE
1 lect
TL
TE
1 réd
DL
2 lect
TL

DL = demander lecture
DE = demander écriture
TL = terminer lecture
TE = terminer écriture

Lecteurs/rédacteurs priorité rédacteurs

Lecteurs/rédacteurs priorité rédacteurs

Libre

lect(s)

1 réd

DL si
DE'count = 0

DE

TL

TE

DL si
DE'count = 0

TL

```
task body LRprioRed is
   type EtatT is (Libre, Lect, Red);
   etat : EtatT := Libre;
   nblect : Natural := 0;
begin
  loop
    if etat = Libre then
      select
        when DE'count = 0 => accept DL; etat := Lect; nblect := 1;
      or
        accept DE; etat := Red;
      end select;
    elsif etat = Lect then
      select
        when DE'count = 0 => accept DL; nblect := nblect + 1;
      or
        accept TL; nblect := nblect - 1;
         if nblect = 0 then etat := Libre; else etat := Lect; end if;
      end select;
    elsif etat = Red then
      accept TE;
      etat := Libre;
    end if;
  end loop;
end LRprioRed;
```

Dynamicité : activation de tâche

Une tâche peut être activée :

- statiquement : chaque **task** T, déclarée explicitement, est activée au démarrage du programme, avant l'initialisation des modules 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.

Dynamicité :Terminaison

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.

La terminaison est difficile !

Bilan processus communicants

+ Pas de partage implicite de la mémoire (→ isolation)
+ Transfert explicite d'information (→ traçage)
+ Réalisation centralisée et répartie
+ Contrôle fin des interactions

∼ Méthodologie

− Performance (copies)
− Quelques schémas classiques, faire preuve d'invention (→ attention aux doigts)