

Systèmes concurrents

Philippe Quéinnec

ENSEEIHT
Département Sciences du Numérique

3 octobre 2022

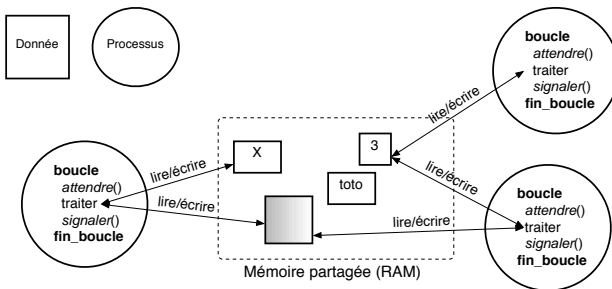
Septième partie

Processus communicants

Contenu de cette partie

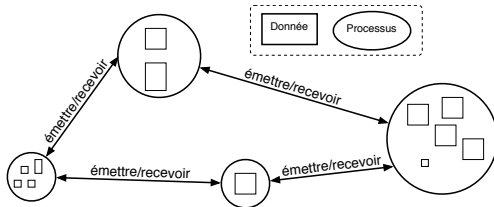
- Modèles de programmation concurrente
- Modèle des processus communicants
- Approche CSP/Go pour la programmation concurrente
 - Goroutine et canaux
 - Communiquer explicitement plutôt que partager implicitement
- Approche Ada pour la programmation concurrente
 - Tâches et rendez vous
 - Démarche de 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

Modèles d'interaction : mémoire partagée



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

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
 - Isolation des données
- **Synchronisation implicite** : attente de message
- Architectures/modèles cibles
 - systèmes répartis : sites distants, reliés par un réseau
 - moniteurs, CSP/Erlang/Go, tâches Ada

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

Processus communicants



Principes

- Communication inter-processus avec des **opérations explicites d'envoi / réception** de messages
- Synchronisation via ces primitives de communication **bloquantes** : envoi (bloquant) de messages / réception bloquante de messages
- Communicating Sequential Processes (CSP) / Calculus of Communicating Systems (CCS) / π -calcul / Erlang / Go
- Ada

Les principes détaillés des échanges et leur utilisation pour développer des applications sont vus dans le module « intergiciels ». On ne s'intéresse ici qu'à la synchronisation.

Quelle synchronisation ?



Réception

Réception bloquante : attendre un message

Émission

- Émission non bloquante ou asynchrone
 - Émission bloquante ou synchrone : bloque jusqu'à la réception du message = **rendez-vous** élémentaire entre l'activité émettrice et l'activité destinataire
 - Rendez-vous étendu : bloquant jusqu'à réception + réaction + réponse \approx appel de procédure
-
- Émission asynchrone \Rightarrow buffers (messages émis non reçus)
 - Synchrone \Rightarrow 1 case suffit



Désignation du destinataire et de l'émetteur



Nommage

- Direct : désignation de l'activité émettrice/destinataire
SEND message TO processName
RECV message FROM processName
- Indirect : désignation d'une boîte à lettres ou d'un canal de communication
SEND message TO channel
RECV message FROM channel

Multiplicité

$1 - 1$

Désignation de l'activité : 1 émetteur / 1 récepteur désignés

$n - 1$

Canal réservé en lecture (consommation) : envoi par n'importe quelle activité ; réception par une seule, propriétaire du canal

$n - m$

Canal avec envoi par n'importe qui, réception par n'importe qui :

- pas de duplication : un seul destinataire consomme le message
- ou duplication à tous les destinataires (diffusion)

En mode synchrone, la diffusion est complexe et coûteuse à mettre en œuvre (nécessite une synchronisation globale entre tous les récepteurs)



Alternative



Alternative en émission ou en réception = **choix** parmi un ensemble de communications possibles :

```
RECV msg FROM channel1 OR channel2
(SEND msg1 TO pid1) OR (SEND msg2 TO pid2)
(RECV msg1 FROM channel1) OR (SEND msg2 TO channel2)
```

- Si aucun choix n'est faisable \Rightarrow attendre
- Si un seul des choix est faisable \Rightarrow le faire
- Si plusieurs choix sont faisables \Rightarrow sélection non-déterministe (arbitraire)

Divers

Émission asynchrone \Rightarrow risque de buffers pleins



- perte de messages ?
- ou l'émission devient bloquante si plein ?

Émission non bloquante \rightarrow émission bloquante



introduire un acquittement

```
(SEND m TO ch; RECV _ FROM ack)
```

```
|| (RECV m FROM ch; SEND _ TO ack)
```

Émission bloquante \rightarrow émission non bloquante



introduire une boîte intermédiaire qui accepte immédiatement tout message et le stocke dans une file.

```
(SEND m TO ch1)
```

```
|| boucle (RECV m FROM ch1; insérer m dans file)
```

```
|| boucle (si file non vide alors extraire et SEND TO ch2)
```

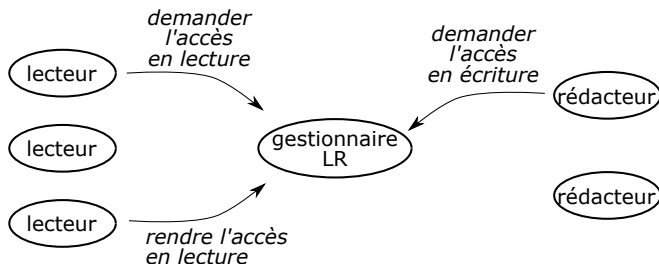
```
|| (RECV FROM ch2)
```



Architecture



La résolution des problèmes de synchronisation classiques (producteurs/consommateurs. . .) ne se fait plus en synchronisant directement les activités via des données partagées, mais indirectement via une **activité de synchronisation**.



Activité arbitre pour un objet partagé



Interactions avec l'objet partagé

Pour chaque opération Op ,

- émettre un message de requête vers l'arbitre
- attendre le message de réponse de l'arbitre

(\Rightarrow se synchroniser avec l'arbitre)

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 message correspondant)

Note : en communication synchrone, on peut se passer du message de réponse s'il n'y a pas de contenu à fournir.



Intérêt

- + découplage entre les activités clientes : l'interface partagée est celle de l'activité de synchronisation
- + réalisation **centralisée et répartie**
- + transfert explicite d'information : traçage
- + pas de données partagées ⇒ **pas de protection nécessaire**
- + contrôle fin des interactions
- + schéma naturel côté client : question/réponse = appel de fonction
- multiples recopies (mais optimisations possibles)
- parallélisation du service : au cas par cas

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

Go language



Principes de conception

- Syntaxe légère inspirée du C
- Typage statique fort avec inférence
- Interfaces avec extension et polymorphisme (typage structurel / duck typing à la Smalltalk)
- Ramasse-miettes

Concepts pour la concurrence

- Descendant de CSP (Hoare 1978), cousin d'Erlang
- Goroutine ~ activité/thread
 - une fonction s'exécutant indépendant (avec sa pile)
 - très léger (plusieurs milliers sans problème)
 - gérée par le noyau Go qui alloue les ressources processeurs
- Canaux pour la communication et la synchronisation



Go – canaux



Canaux

- Création : `make(chan type)` ou `make(chan type, 10)`
(synchrone / asynchrone avec capacité)
- Envoi d'une valeur sur le canal `chan` : `chan <- valeur`
- Réception d'une valeur depuis `chan` : `<- chan`
- Canal transmissible en paramètre ou dans un canal :
`chan chan int` est un canal qui transporte des canaux
(transportant des entiers)

Go – canaux



Alternative en réception et émission

```
select {  
  case v1 := <- chan1:  
    fmt.Printf("received %v from chan1\n", v1)  
  case v2 := <- chan2:  
    fmt.Printf("received %v from chan2\n", v2)  
  case chan3 <- 42:  
    fmt.Printf("sent %v to chan3\n", 42)  
  default:  
    fmt.Printf("no one ready to communicate\n")  
}
```

Exemple élémentaire



```
func boring(msg string, c chan string) {  
    for i := 0; ; i++ {  
        c <- fmt.Sprintf("%s %d", msg, i)  
        time.Sleep(time.Duration(rand.Intn(4)) * time.Second)  
    }  
}
```

```
func main() {  
    c := make(chan string)  
    go boring("boring!", c)  
    for i := 0; i < 5; i++ {  
        fmt.Printf("You say: %q\n", <- c)  
    }  
    fmt.Println("You're boring; I'm leaving.")  
}
```



Moteur de recherche



Objectif : agrégation de la recherche dans plusieurs bases

```
func Web(query string) Result
func Image(query string) Result
func Video(query string) Result
```

Moteur séquentiel

```
func Google(query string) ( results [] Result) {
    results = append(results, Web(query))
    results = append(results, Image(query))
    results = append(results, Video(query))
    return
}
```

exemple tiré de <https://talks.golang.org/2012/concurrency.slide>



Recherche concurrente



Moteur concurrent

```
func Google(query string) ( results [] Result) {  
    c := make(chan Result)  
    go func() { c <- Web(query) } ()  
    go func() { c <- Image(query) } ()  
    go func() { c <- Video(query) } ()  
  
    for i := 0; i < 3; i++ {  
        result := <- c  
        results = append(results, result )  
    }  
    return  
}
```

Le temps sans interruption



Crée un canal sur lequel un message sera envoyé après la durée spécifiée.

time.After

```
func After(d time.Duration) <-chan bool {  
    // Returns a receive-only channel  
    // A message will be sent on it after the duration  
    c := make(chan bool)  
    go func() {  
        time.Sleep(d)  
        c <- true  
    }()  
    return c  
}
```

Recherche concurrente en temps borné



Moteur concurrent avec timeout

```
c := make(chan Result)
go func() { c <- Web(query) } ()
go func() { c <- Image(query) } ()
go func() { c <- Video(query) } ()

timeout := time.After(80 * time.Millisecond)
for i := 0; i < 3; i++ {
    select {
        case result := <-c:
            results = append(results, result)
        case <-timeout:
            fmt.Println("timed out")
            return
    }
}
return
```


Recherche répliquée



Utiliser plusieurs serveurs répliqués et garder la réponse du premier qui répond.

Recherche en parallèle

```
func First(query string, replicas ... Search) Result {  
    c := make(chan Result)  
    searchReplica := func(i int) { c <- replicas[i](query) }  
    for i := range replicas {  
        go searchReplica(i)  
    }  
    return <-c  
}
```

Recherche répliquée



Moteur concurrent répliqué avec timeout

```
c := make(chan Result)
go func() { c <- First(query, Web1, Web2, Web3) } ()
go func() { c <- First(query, Image1, Image2) } ()
go func() { c <- First(query, Video1, Video2) } ()
timeout := time.After(80 * time.Millisecond)
for i := 0; i < 3; i++ {
    select {
        case result := <-c:
            results = append(results, result)
        case <-timeout:
            fmt.Println("timed out")
            return
    }
}
return
```

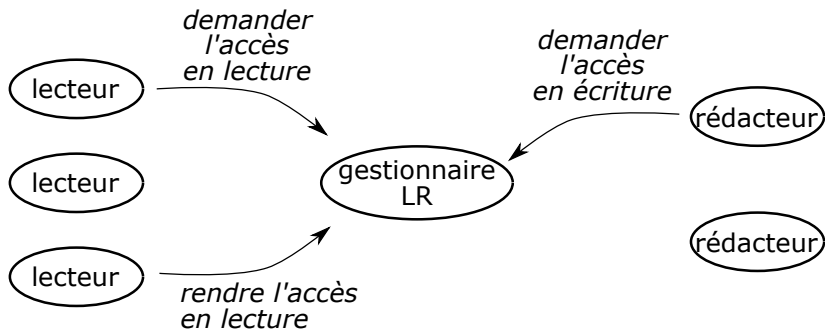
Bilan

- Création ultra-légère de goroutine : penser concurrent
- Pas besoin de variables partagées
⇒ Pas de verrous
- Pas besoin de variable condition / sémaphore pour synchroniser
- Pas besoin de callback ou d'interruption

Don't communicate by sharing memory, share memory by communicating.

(la bibliothèque Go contient *aussi* les objets usuels de synchronisation pour travailler en mémoire partagée : verrous, sémaphores, moniteur...)

Lecteurs/rédacteurs



- Un canal pour chaque type de requête : DL, TL, DE, TE
- Émission bloquante \Rightarrow accepter un message (une requête) uniquement si l'état l'autorise

Lecteurs/rédacteurs



Utilisateur

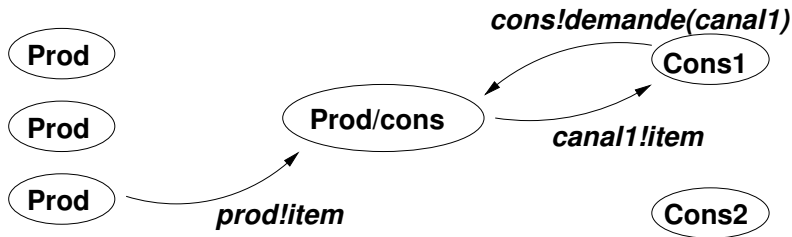
```
func Utilisateur () {  
  nothing := struct {} {}  
  for {  
    DL <- nothing; // demander lecture  
    ...  
    TL <- nothing; // terminer lecture  
    ...  
    DE <- nothing; // demander écriture  
    ...  
    TE <- nothing; // terminer écriture  
  }  
}
```

Goroutine de synchronisation

```
func when(b bool, c chan struct{}) chan struct{ } {  
    if b { return c } else { return nil }  
}
```

```
func SynchroLR() {  
    nblec := 0;  
    ecr := false;  
    for {  
        select {  
            case <- when(nblec == 0 && !ecr, DE):  
                ecr := true;  
            case <- when(!ecr, DL):  
                nblec++;  
            case <- TE:  
                ecr := false;  
            case <- TL:  
                nblec--;  
        }  
    }  
}
```

Producteurs/consommateurs : architecture



- Un canal pour les demandes de dépôt
- Un canal pour les demandes de retrait
- Un canal par activité demandant le retrait (pour la réponse à celle-ci)

(exercice futile : `make(chan T, N)` est déjà un tampon borné = un prod/cons de taille N)



Producteurs/consommateurs



Programme principal

```
func main() {  
    prod := make(chan int)    // un canal portant des entiers  
    cons := make(chan chan int) // un canal portant des canaux  
    go prodcons(prod, cons)  
    for i := 1; i < 10; i++ {  
        go producteur(prod)  
    }  
    for i := 1; i < 5; i++ {  
        go consommateur(cons)  
    }  
    time.Sleep (20*time.Second)  
    fmt.Println ("DONE.")  
}
```



Producteurs/consommateurs



Producteur

```
func producteur(prod chan int) {  
  for {  
    ...  
    item := ...  
    prod <- item  
  }  
}
```

Consommateur

```
func consommateur(cons chan chan int) {  
  moi := make(chan int)  
  for {  
    ...  
    cons <- moi  
    item := <- moi  
    // utiliser item  
  }  
}
```

Variables « canal », passables en paramètre et en message.

Producteurs/consommateurs



Goroutine de synchronisation

```
func prodcons(prod chan int, cons chan chan int) {  
    nbocc := 0;  
    queue := make([]int, 0)  
    for {  
        if nbocc == 0 {  
            m := <- prod; nbocc++; queue = append(queue, m)  
        } else if nbocc == N {  
            c := <- cons; c <- queue[0]; nbocc--; queue = queue[1:]  
        } else {  
            select {  
            case m := <- prod: nbocc++; queue = append(queue, m)  
            case c := <- cons:  
                c <- queue[0]; nbocc--; queue = queue[1:]  
            }  
        }  
    }  
}
```

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.

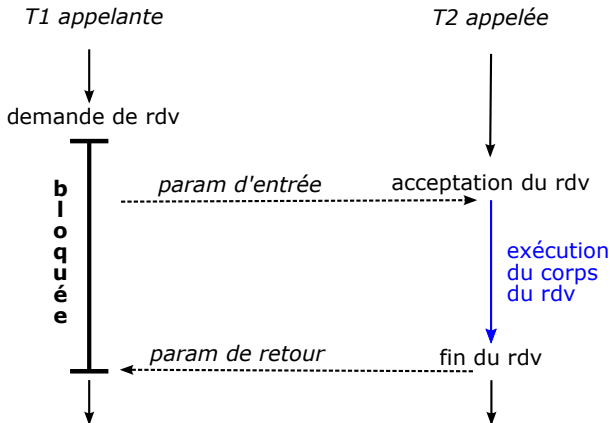
Vocabulaire : tâche = activité

Principe du rendez-vous

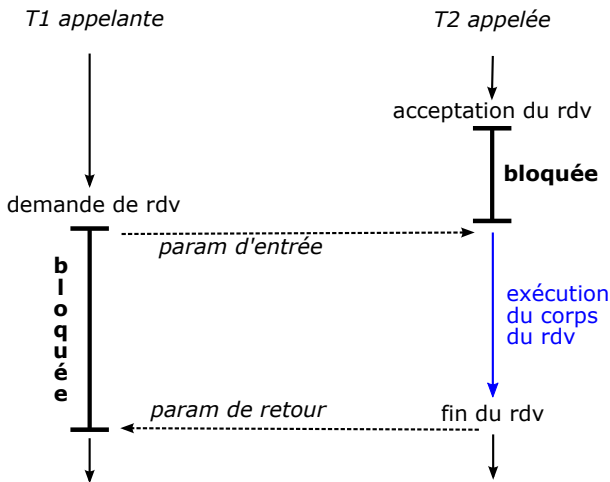


- Une tâche possède des **points d'entrée de rendez-vous**.
- Une tâche peut :
 - demander un rendez-vous avec une autre tâche désignée explicitement ;
 - attendre un rendez-vous sur un (ou plusieurs) point(s) d'entrée.
- Un rendez-vous est **dissymétrique** : tâche appelante ou cliente vs tâche appelée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

task body Prod is

```
...  
begin  
  ...  
  ...  
  ...  
  m := ...  
  ProdCons.Deposer (m);  
  (client attend)  
  ...  
end Prod;
```

task body ProdCons is

```
...  
begin  
  ...  
  accept Deposer (msg : in T) do  
    (serveur attend)  
    deposer_dans_tampon(msg);  
  end Deposer;  
  nbOcc := nbOcc + 1;  
  ...  
  ...  
  accept Retirer (msg : out T) do  
    retirer_du_tampon(msg);  
  end Retirer;  
  nbOcc := nbOcc - 1;  
  ...  
end ProdCons;
```

task body Conso is

```
...  
begin  
  ...  
  ...  
  ProdCons.Retirer (m);  
  (client attend)  
  ...  
  utiliser(m);  
  ...  
end Conso;
```

para entrée

para sortie

paramètres d'entrée

paramètres de sortie

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 Ressourceld);  
  entry Rendre (nbRendu: in natural;  
               id : in array of Ressourceld);  
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;

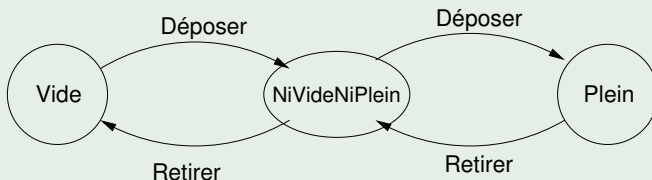
Méthodologie par machine à états



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;

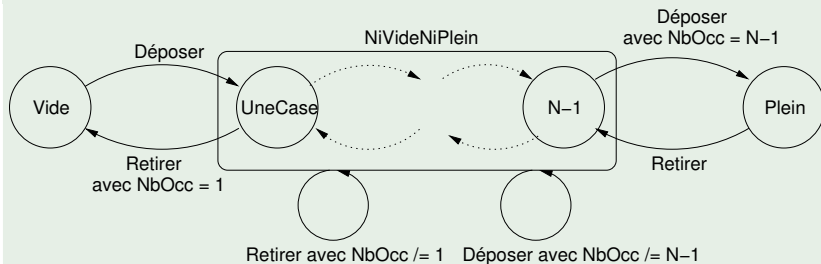
```
    elsif etat = Plein then  
        select  
            accept Retirer (msg : out T) do  
                msg := retirer_du_tampon();  
            end Retirer;  
            etat := NiVideNiPlein;  
        end select;  
    end if;  
end loop;  
end ProdCons;
```

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.

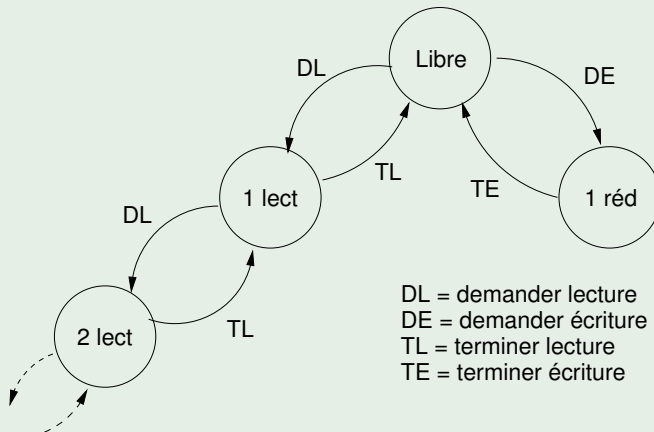
Producteurs/consommateurs à N cases



Lecteurs/rédacteurs



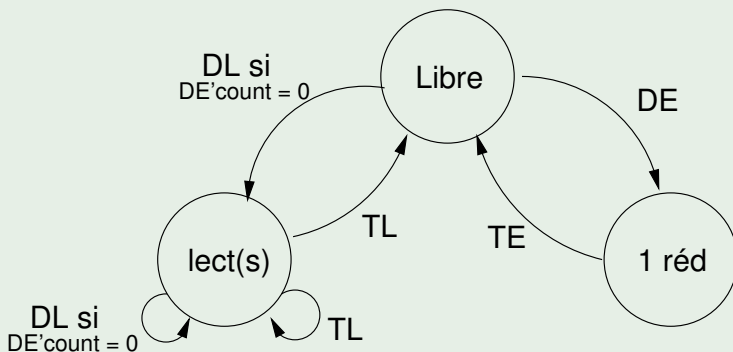
Lecteurs/rédacteurs



Lecteurs/rédacteurs priorité rédacteurs



Lecteurs/rédacteurs priorité rédacteurs



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;

Dynamacité : 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.

Dynamacité :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)