CH4 : Programmation Concurrente

- La concurrence dans les L.P et le parallélisme au niveau Matériel sont deux concepts indépendants :
- Les opérations Matérielles se produisent en parallèle si elles se superposent dans le temps ;
- Les opérations dans le code d'un programme sont concurrentes si elles peuvent être, et pas nécessairement, exécutées en parallèles.

Exemple:

Non concurrent	concurrent
X:=5; Y:=2*X+4;	X:=A*B*C; Y:=3*A+7

Fig 1 code séquentiel et concurrent

- On peut avoir la concurrence sans le parallélisme matériel, et on peut avoir une exécution parallèle sans la concurrence dans le langage (programme exécuté sur monoprocesseur avec ou sans des sections concurrentes explicites);

En conclusion on peut dire que la concurrence se rapporte au potentiel du parallélisme.

Processus et Ressources : concepts fondamentaux de la programmation Concurrente

Processus, correspond à un traitement (calcul) séquentiel, muni de son thread de contrôle (cas du langage ADA).

le thread, d'un calcul séquentiel, est une séquence de points du programme qui sont atteints comme flow de contrôle a travers le code du programme.

La programmation concurrente permet à des processus multiples, de pouvoir partager des ressources (structures de données, ressources matériels, CPU, mémoire, dispositifs I/O).

Interaction entre processus : communication

La communication est un échange de données entre processus,

- soit par messages explicites (convient plus aux systèmes répartis qu'aux systèmes à mémoire partagée). Deux manières d'envoyer un message:

  Synchrone : permet autant la synchronisation que la communication. *le Processus Emetteur doit attendre que son message soit reçu; et il est possible qu'un processus Récepteur doive attendre qu'un message soit envoyé (CSP (Hoare 85, Rendez vous de ADA).* ***Asynchrone***: l'émetteur poursuit son exécution après l'envoi du message.

- soit à travers des variables partagées(visibles à chaque processus; convient aux systèmes à mémoire commune);
- ou bien par RPC (Remote Procedure Call).

Interaction entre processus : Synchronisation

La synchronisation lie le thread (de contrôle) d'un processus avec celui d'un autre processus;

Elle implique l'échange d'information de contrôle entre processus.

Exemples:

- un processus dépend des données produites par un autre; si les données ne sont pas disponibles, le processus doit attendre jusqu'à ce qu'elles soient disponibles.

- un processus peut changer son état à "Blocked"(Wait(Pi)), et peut signaler aux Processus "Blocked" qu'ils peuvent continuer (signal(Pi)).

Concurrence comme entrelacement (Interleaving)

Evénement = action atomique (non interruptible) choisie dans le langage (exécution d'une commande d'affectation, appel de procédure, évaluation d'une expression, etc..)

Entrelacement de threads: C'est un dispositif technique commode pour étudier l'exécution concurrente de processus.

Définition : un entrelacement de deux séquences s et t est une séquence U construite à partir des événements de s et de t, de telle sorte que les événements de s préservent leur ordre dans U ainsi que ceux de t.

Exemple :

Si a et z sont des événements concurrents, on considère le cas ou a se produit avant z ou bien z se produit avant a, mais on ignore le cas ou a et z se produisent en même temps.

Concurrence comme entrelacement : Exemple

Soient deux Processus:
A : a ; b
Z : x ; y ; z

L'exécution concurrente de A et Z, peut être étudiée en considérant les entrelacements possibles (10):

a b x y z

a x b y z

.....

x y z a b

L'entrelacement préserve l'ordre relatif dans le thread: a doit se produire avant b, mais x y z étant dans un autre thread peuvent se produire dans n'importe quel ordre relativement à a et b. Il en est de même pour x, y et z...

Entrelacement de threads: Tâches Concurrentes ADA

Modules: Packages

Processus: Tâches, avec leurs propres thread de contrôle.

Une tâche est déclarée en deux parties (sa spécification, et son corps). On s'intéresse à un programme qui consiste en une procédure, ayant la structure:

Procedure <name> is

<declaration>

begin

<statement>

end<name>

Exemple:

```
with text_io; use text_io; -- utiliser importer la procédure
-- put_line depuis le module text_io
procedure hello is
begin
put_line("hello");
end hello;
```

Entrelacement de threads: Tâches Concurrentes ADA (suite)

```
with text_io; use text_io;
procedure identifier is
task p; -- spécification de la tache p
task body p is
begin
put_line ("p");
end p;
task q; -- spécification de la tache q
task body q is
begin
put_line ("q");
end q;
begin -- corps de la procédure, parent de p et q
put_line("r");
end identifier ;
```

- quand la procédure identifier est activée, les tâches déclarées à l'intérieur de la procédure sont aussi activées; les taches (p, q, et le corps de la procédure) s'exécutent de manière concurrente. Les appels à put_line dans la procédure et les deux tâches peuvent être entrelacées:

p p q q r r

q r p r p q

r q r p q p

Propriétés de vivacité

L'exactitude d'une exécution concurrente de processus se ramène à l'une des deux catégories de Problèmes:

- ***Sûreté (Safety*)**: un événement indésirable ne s'est pas produit pendant l'exécution (lié au Passé): le programme doit produire le résultat "exacte".

Exemple: l'accès à une ressource partagée comme l'imprimante nécessite que le processus utilisateur a un accès exclusif à la ressource (Exclusion Mutuelle);

- **Vi*vacité (liveness)*** : un événement souhaité arrivera nécessairement (lié au futur). Elle est liée à la progression du programme.

Exemple: accès à l'imprimante

Pas de processus qui empêchent d'autres à un éventuel accès à l'imprimante. Ainsi tout processus qui souhaite utiliser l'imprimante doit éventuellement avoir accès à l'imprimante.

Propriétés de vivacité : Le Modèle Producteur Consommateur

Cas où le Tampon circulaire :

Déclaration commune :

Type messages : suite de caractères ;

Variables communes :

Tampon : tableau [0..N-1] de messages ;

Processus Producteur	*Processus consommateur*
Var privée	Var privée
Mess1 : messages ;	Mess2 : messages ;
tête : entier init. à 0 ;	queue : entier init. à 0 ;
Répéter	Répéter
Produire (Mess1) ;	Mess2 := Tampon[queue] ;
Tampon[tête] := Mess1 ;	queue := queue + 1 mod N;
tête := tête +1 mod N;	Consommer (Mess2) ;
Jusqu'à faux ;	Jusqu'à faux ;

- *Problème de sûreté* est satisfait si: pas de chevauchement entre les deux pointeurs tête et queue ;
- *Problème de vivacité* est satisfait si: quand la queue contient un message, le consommateur doit être capable d'accéder à la queue, et quand la tête (Tampon) contient un espace pour un autre message, le producteur doit être capable d'accéder à la tête du tampon.

Propriétés de vivacité (suite)

La compétition sur les ressources impose des contraintes sur l'entrelacement des threads.

Exemple: soit R une ressource et A et Z deux processus et leur threads respectifs:

A: a verouiller_A (R) b c déverrouiller _A (R) d

Z: w verrouiller _Z(R) x y déverrouiller _Z (R) z

on obtient par exemple l'entrelacement suivant:

a w verrouiller_A (R) b c déverrouiller_A (R) verrouiller_Z(R) x d y déverrouiller_Z (R) z

Quelques Propriétés de vivacité: Le problème des Philosophes

Processus = Philo.

Ressources = Baguettes;

Spécification du Problème:

Règles de Politesse :

-Un philo. ne partage ses baguettes qu'avec ses voisins

-Un Philo. qui réfléchit n'utilise pas de baguette.

-Un Philo. Peut donc manger si ses deux voisins réfléchissent.

-Un Philo. est susceptible ; on ne peut lui enlever une baguette.

-Un Philo. ne mange que pendant un temps fini.

-Un Philo. ne se permet pas de manger avec ses doigts.

Cycle d'un Philo. :

Répeter

Penser ;

*Prendre les baguettes /*attente éventuelle*/*

Manger ;

*Remettre les baguettes ; /*réveil éventuel des voisins*/*

Jusqu'à fin_de_repas

Problème à résoudre:

Il consiste à synchroniser les Philo. pour que :

- il ne se produise pas de blocage (pas interblocage);
- si un Philo. désire manger, il puisse le faire au bout d'un temps fini (pas de famine).

Interblocage "Deadlock"

- C'est un problème lié à la sûreté.
- Un programme concurrent est dans un état d'interblocage (Deadlock) si un ensemble de processus sont privés de progresser à cause de leur demandes mutuelles de ressources additionnelles
- Solutions: prévention ou guérison

Exemple des Philosophes:
Baguettes = Ressources
prendre la baguette = verrouiller la ressource

Si chaque Philo. prend sa baguette de gauche, et attend la baguette de droite ; aucun Philo. ne peut progresser.

Famine ("starvation")

- C'est un problème lié à la vivacité : le système n'est pas dans une situation d'interblocage, mais aucun processus ne peut progresser.

Exemple: supposons que dans le problème des Philo. on impose: la baguette de gauche est remise dans le tas si la baguette de droite n'est pas disponible. Tous les Philo. effectuent la boucle:

prendre baguette de G;

remettre baguette de G;

prendre baguette de G;

remettre baguette de G;

Equité "fairness"

- Un processus qui veut progresser ne doit pas être bloqué indéfiniment.
- Exemple: une solution non équitable au problème des Philos est de laisser toujours les mêmes Philo. manger, et les autres attendent indéfiniment.

C'est une propriété délicate à assurer car en général on ne connaît pas la vitesse d'exécution des processus, et une solution par alternance n'est toujours pas la meilleure.

Sûreté d'Accès aux variables partagées

Explore la notion de sûreté pour des programmes concurrents avec des variables partagées.

Processus déterministes/non déterministes:

1.Un programme est déterministe si son évaluation sur les mêmes données en entrée produit toujours les mêmes sorties. La stratégie d'évaluation peut ne pas être toujours la même.

- *Exemple*: Les programmes séquentiels sont déterministes

2.Un programme est non déterministe si il a plus d'une stratégie d'évaluation, et différentes stratégies conduisent à des résultats différents.

- *Exemple*: les programmes concurrents peuvent être non déterministes puisque l'ordre et la vitesse d'exécution des processus ne sont pas prédictives.

Autres Exemples:

- deux processus P et Q envoient des impressions sur une imprimante;

- deux personnes réservent de manière concurrente une place (la dernière) dans un avion; seulement l'une d'entre elle pourra prendre l'avion ; le système de réservation ne s'intéresse pas au fait: *" laquelle des deux personnes a eu la place ?"*

Sections Critiques et Exclusion Mutuelle

Formulation de la notion de sûreté qui impose des contraintes sur l'entrelacement sans insister sur un unique résultat déterministe.

1. Une *section critique* dans un processus est une portion ou une section de code qui doit être traitée comme un événement atomique.

2. Deux sections critiques sont dites *mutuellement exclusives* car leur exécution ne doit pas être superposée.

Ainsi, un programme concurrent assure la propriété de sûreté s'il exécute la section critique de manière contiguë sans entrelacement.

Exclusion Mutuelle : Exemple

Soit le code :

x:=x+1; x:=x+2;

- En programmation séquentielle; cette affectation incrémente la valeur de x de 3.

- En programmation concurrente, on ne peut pas traiter l'affectation comme un événement atomique.

Ainsi, supposons que cette affectation est implémentée par deux processus comme suit:

Processus P	Processus Q
t:=x; x:=t+1;	u:=x; x:=u+2;

Si l'affectation de Q est entrelacée avec celle de P:

t:=x;

u:=x;

x:=u+2;

x:=t+1;

Alors x est incrémenté de 1 au lieu de 3 ; cet entrelacement ne peut être effectué que si on traite l'affectation comme une section critique.

Exclusion Mutuelle: Spécification du Problème

- Processus s'exécutent sur une machine mono/multiProcesseurs avec mémoire commune.
- Partager des variables communes pour :

  - coopérer et traiter un même problème

  - se partager des ressources.
- Le partage sans précaution particulière peut conduire à des résultats imprévisibles.

Exemple : Réservation avec un système temps partagé

```
Si Place_dispo 0
   alors Place_dispo := Place_dispo -1 ;
   répondre ('Place réservée') ;
   sinon répondre ('plus de place') ;
fsi.
```

Exclusion Mutuelle: Hypothèses d'exécution

- Les évolutions de chaque processus sont à priori indépendantes;
- Le délai entre deux instructions d'un processus est non nul, mais fini ;
- Deux accès à une même case mémoire ne peuvent être simultanés ;
- Les registres sont sauvegardés et restaurés à chaque commutation

Exemple : Incrémentation/décrementation d'un compteur

Processus1	Processus2
. LOAD R1 Compteur ADD R1 1 STORE R1 Compteur	. LOAD R1 Compteur SUB R1 1 STORE R1 Compteur

---

Sections Critiques (SC)

Sections critiques = ensemble de suites d'instructions qui peuvent produire des résultats imprévisibles lorsqu'elles sont exécutées simultanément par des processus différents.

- Une suite d'instructions sur des variables partagées est éventuellement une section critique relativement à d'autres suites d'instructions sur les mêmes variables partagées et non dans l'absolu.

- L'exécution simultanée de deux sections critiques appartenant à des ensembles différents et ne partagent pas de variable ne pose pas de problème.

***Problème de la détermination des sections critiques* :**

- l'existence implique l'utilisation de variables partagées, mais l'inverse n'est pas vrai.

- Pratiquement les sections critiques doivent être détectées par les concepteurs de programmes,

- et dès qu'il y a des variables partagées, il y a de forte chance de se trouver en présence de sections critiques.

Sections Critiques (suite)

- Les sections critiques doivent être exécutées en exclusion mutuelle :

  une section critique ne peut être commencée que si aucune autre section critique du même ensemble n'est en cours d'exécution.

- Avant d'exécuter une section critique, un processus doit s'assurer qu'aucun autre processus n'est en train d'exécuter une section critique du même ensemble ;

- Dans le cas contraire, il ne devra pas progresser plus avant, tant que l'autre processus n'aura pas terminé sa section critique.

- ***Structure des processus:***

  Début

  section non critique ;

  protocole d'entrée en SC

  SC

  protocole de sortie de SC

  section non critique

  Fin

- Protocole d'entrée en SC : un ensemble d'instructions qui permet cette vérification et la non-progression éventuelle;
- Protocole de sortie de SC : un ensemble d'instructions qui permet à un processus ayant terminé sa SC d'avertir d'autres processus en attente que le voie est libre

---

Exclusion Mutuelle : Propriétés

La résolution de l'exclusion mutuelle doit garantir trois propriétés :

- Au plus un seul processus en section critique ;
- Si la section critique est libre, un processus qui veut entrer en section critique doit pouvoir y entrer sans attente si aucun processus n'est en section critique ; i.e. il ne doit pas attendre qu'un autre processus passe avant lui pour avoir le droit d'y entrer ;
- un processus désirant entrer en section critique y entre au bout d'un temps fini ; il ne faut pas qu'il y est de privation d'entrée en section critique vis à vis d'un processus (pas d’attente de manière infinie).

Nous pouvons rajouter deux propriétés :

- la *tolérance aux défaillances*, i.e, si le processus en SC est détruit ou se termine anormalement, il faut pas qu'il bloque tout le système ;
- la *symétrie* i.e, les protocoles d'entrée et de sortie en SC doivent être identiques pour tous les processus et indépendants de leur nombre.

Exclusion Mutuelle : Propriétés (suite)

Pour monter qu'une résolution ne garantit pas l'exclusion mutuelle, il faut montrer l'une des assertions suivantes :

- un processus peut entrer en section critique alors qu'un autre s'y trouve déjà.
- un processus désirant entrer en SC ne peut pas y entrer alors qu'il n'y a aucun processus en SC ; il doit attendre qu'un autre y entre et sorte avant de pouvoir entrer.

-un processus désirant entrer en SC n'y entrera jamais parce qu'il ne sera jamais sélectionné lorsqu'il est en concurrence avec d'autres processus.

Exclusion Mutuelle à l'aide des Sémaphores

- Introduit par Dijkstra en 1965 pour résoudre le problème d'exclusion mutuelle.
- Permettent l'utilisation de m ressources identiques (exemple imprimantes) par n processus

Définitions

Un sémaphore est une variable globale protégée, c'est à dire on peut y accéder qu'au moyen des trois procédures :

I(S, x) : initialiser le sémaphore S à une certaine valeur x ;

P(S) ; Peut-on passer ?/peut-on continuer?

V(S) ; libérer?/vas y?

Un sémaphore est une structure contenant deux champs :

Struct {

n : entier ;

en_attente : file de processus

}

Un *sémaphore binaire* est un sémaphore dont la valeur peut prendre que deux valeurs positives possibles : en générale 1 et 0.

Un *sémaphore de comptage* : la valeur peut prendre plus de deux valeurs positives possibles. Il est utile pour allouer une ressource parmi plusieurs exemplaires identiques : la valeur est initialisée avec le nombre de ressources.

Les Sémaphores

Réalisations logicielles des primitives P et V

P et V sont des primitives plutôt que des procédures car elles sont non interruptibles (possible sur monoprocesseur par masquage d'Interruption).

Une première implantation logicielle:

```
I(S, x)
{S.n :=x ;
}
P(S) /*S.n est tjs modifié par P(S)*/
{ S.n :=S.n -1 ;
    Si S.n < 0 alors bloquer le processus en fin de S.en_attente ;
}
V(S) /*S.n est tjs modifié par V(S)*/
{ S.n :=S.n + 1 ;
Si S.n <= 0 alors débloquer le processus en tête de S.en_attente ;
}
```

Ainsi, pour résoudre le problème de l'exclusion mutuelle, il suffit d'initialiser S à 1, et la procédure d'entrée est P(S), et la procédure de sortie est V(S)

Remarque:

- L'initialisation dépend du nombre de processus pouvant effectuer en même temps une même " section critique " : m, si on a m imprimantes identiques par exemple;
- Cette implémentation donne à chaque fois dans S.n le nombre de ressources libres ; lorsque S.n est négative, sa valeur absolue donne le nombre de processus dans la file.

Une deuxième implantation logicielles:

Traduction directe de la spécification fonctionnelle, équivalente à la première implantation

```
P(S) {
Si S.n > 0 alors S.n =S.n -1 ;
Sinon bloquer le processus en fin de S.en_attente ;
}
```

V(S)

{

Si S.en_attente non-vide alors débloquer le processus en tête de S.en_attente ;

sinon S.n :=S.n +1 ;

}

Les Sémaphores (suite)

Sémaphore d'exclusion Mutuelle

Var mutex : sémaphore init. à 1

Processus Pi

Début

...

P(mutex)

SC

V(mutex)

...

Fin

Sémaphore de synchronisation

Rappel du Principe : un processus doit attendre un autre pour continuer (ou commencer) son exécution.

Var sem initialisée à 0;

Processus 1	**Processus 2**
1 er travail V(sem)//réveil processus 2	P(sem)//attente processus1 2eme travail

Attention : Problème d'Interblocage

Processus 1	**Processus 2**
P(semA);	P(semB);
P(semB);	P(semA);
<SC	<SC
V(semA);	V(semB);
V(semB);	V(semA);

Exemple: Producteur/consommateur

Problème à résoudre: déposer un message alors que le consommateur n'a pas retiré le précédent (cas où la taille du tampon est égale à 1), ou retirer un message alors que le producteur n'a rien déposé.

Solution naive: Attente active

```
Déclaration communes :
Type messages : suite de caractères ;
Variables communes :
Tampon : tableau [1..nb] de messages ;
tête, queue: entier init. à 1
```

Processus Producteur	**Processus consommateur**
Var privée	Var privée
Mess1 : messages ;	Mess2 : messages ;
Répéter	Répéter
Produire (Mess1) ;	Tant que tête = queue faire attente
Tampon[tête] :=Mess1;	Mess2 := Tampon[queue] ;
tête := tête +1 ;	queue := queue + 1 ;
Jusqu'à faux ;	Consommer (Mess2) ;
	Jusqu'à faux ;

Problèmes **:**

- Attente active (monopolise le processeur),
- Pas d'exclusion Mutuelle sur les variables communes tête et queue.

Le Modèle Producteur Consommateur : Résolution par les sémaphores

Cas 1 : Tampon infini

Déclaration commune :

Type messages : suite de caractères ;

Variables communes :

Tampon : tableau [0…;] de messages ;

NPLEIN : sémaphore initialisé à 0 ;

Processus Producteur	**Processus consommateur**
Var privée	Var privée
Mess1 : messages ;	Mess2 : messages ;
tête : entier init. à 0 ;	queue : entier init. à 0 ;
Répéter	Répéter
Produire (Mess1) ;	*P(NPLEIN) ;*
Tampon[tête] := Mess1;	Mess2 := Tampon[queue] ;
tête := tête +1 ;	queue := queue + 1 ;
V(NPLEIN) ;	Consommer (Mess2) ;
Jusqu'à faux ;	Jusqu'à faux ;

Le Modèle Producteur Consommateur: résolution par les sémaphores

Cas 2 : Tampon circulaire :

Déclaration commune:

Type messages : suite de caractères ;

Variables communes :

Tampon : tableau [0..N-1] de messages ;

NPLEIN : sémaphore initialisé à 0 ;

NVIDE : sémaphore initialisé à N ;

Processus Producteur	**Processus consommateur**
Var privée	Var privée
Mess1 : messages ;	Mess2 : messages ;
tête : entier init. à 0 ;	queue : entier init. à 0 ;
Répéter	Répéter
Produire (Mess1) ;	*P(NPLEIN) ;*
P(NVIDE) ;	Mess2 := Tampon[queue] ;
Tampon[tête] := Mess1 ;	queue := queue + 1 mod N;
tête := tête +1 mod N;	*V(NVIDE) ;*
V(NPLEIN) ;	Consommer (Mess2) ;
Jusqu'à faux ;	Jusqu'à faux ;

Synchronisation par Rendez Vous

- La meilleure solution pour éviter tous les problèmes dû aux accès simultanés à des variables partagées est précisément de n'en avoir aucune. Ce qui est le cas dans les systèmes distribués.
- La synchronisation ne consiste pas, pour un processus, à attendre qu'une variable donnée ait une certaine valeur, mais à attendre qu'un second processus veuille communiquer avec le premier.

Ceci nous a conduit à la notion de rendez-vous développée dans le langage CSP (pour Communicating Sequential Processes) [C.A.R. Hoare] puis reprise avec d'autres structures modernes dans le langage ADA.

Primitives de Synchronisation (ADA)

Déclaration de processus :

TASK *<id_processus* ***IS*** *<corps_du_processus* ***END;***

Activation d'un processus :

INITIATE *<id_processus;* -- à vérifier sur certaines versions d'ADA'

Déclaration de RDV dans le corps d'un processus proc1 :

ENTRY <id_rendez-vous (liste de paramètres formels);

-- chacun des paramètres doit avoir un type et être spécifié IN ou OUT

Attente de rendez-vous par proc1 :

ACCEPT *<id_rendez-vous(liste des paramètres)*

DO

<instructions

END;

-- le nombre et le type des paramètres doivent coïncider

Demande de RDV de proc1 par un autre processus proc2:

proc1.<id_rendez-vous(liste de paramètres d'appel);

Remarque: Le rendez-vous ne peut être réalisé que si proc1 est prêt à exécuter ACCEPT et si proc2 est prêt à exécuter la demande. Si l'un des deux n'est pas prêt alors l'autre doit attendre qu'il le soit.

Primitives de Synchronisation ADA (suite)

Attente multiple :

SELECT

ACCEPT *<id_RV(...)* ***DO*** *<liste d'instructions* ***END;***

<liste d'instructions;

OR

ACCEPT *<id_RV(...)* ***DO*** *<liste d'instruction* ***END;***

<liste d'instructions;

END;

Attente conditionnelle à l'intérieur d'un SELECT :

SELECT

ACCEPT *<id_RV(...)* ***DO ... END;***

...;

OR

ACCEPT *<id_RV(...)* ***DO ... END;***

...;

OR

WHEN *<condition* ***= ACCEPT****<id_RV(...)* ***DO ... END;***

...;

ELSE *<liste d'instructions;*

END;

Sémantique de l'instruction SELECT

* Toutes les conditions apparaissant dans les WHEN sont évaluées. Chaque ACCEPT dont la condition est vraie est dit ouvert. Un ACCEPT qui n'est pas précédé d'un WHEN est toujours ouvert.

* Un ACCEPT ouvert ne peut être exécuté que si un autre processus a appelé l'ENTRY correspondante. Si plusieurs ACCEPT peuvent être exécutés alors un seul, choisi arbitrairement, l'est. Si aucun ne peut être exécuté alors le ELSE, s'il y en a un, l'est. S'il n'y en a pas, alors le processus attend qu'un ACCEPT ouvert puisse être exécuté.

* Si aucun ACCEPT n'est ouvert et s'il y a un ELSE alors il est exécuté. S'il n'y en a pas alors il se produit une erreur d'exécution.

* Le OR est exclusif : un seul ACCEPT peut être exécuté à la fois et pour en exécuter un autre il faut re-exécuter le SELECT.

* S'il y a plusieurs processus appelant une ENTRY, un seul est choisi pour exécuter le rendez-vous.

* Au début de l'exécution d'un ACCEPT, le processus acceptant reçoit du processus appelant les paramètres spécifiés IN. Ce dernier est ensuite bloqué jusqu'à la réception des paramètres spécifiés OUT qui ne lui sont transmis par l'acceptant qu'après l'exécution des instructions figurant entre DO et END.

* Après l'exécution du ACCEPT, chacun des deux processus reprend son exécution de façon autonome : l'instruction suivant l'appel pour l'appelant; les instructions qui suivent la clause DO ... END, s'il y en a, puis l'instruction suivant le SELECT pour l'acceptant.

L'intérêt du SELECT est de permettre de pouvoir attendre des événements dans un ordre non prédéterminé.

Synchronisation : Exemple

1.Réalisation d'une Exclusion Mutuelle

```
TASK MUTEX IS -- partie visible de la
ENTRY ENTREE; -- déclaration qui peut
ENTRY SORTIE; -- être connue par les
END MUTEX; -- utilisateurs
TASK BODY MUTEX IS -- partie invisible de la
BEGIN -- déclaration qui n'est
LOOP -- connue que de celui qui
    ACCEPT ENTREE; -- programme MUTEX
    ACCEPT SORTIE; -- et le met à la disposition
END LOOP; -- des utilisateurs
END MUTEX;

TASK BODY PROCESSUS IS
BEGIN
...
MUTEX.ENTREE;
section_critique;
MUTEX.SORTIE;
...
END PROCESSUS;
```

Synchronisation : Exemple (2)

2.Producteur/Consommateur

```
TASK BODY MAGASINIER IS
ENTRY DEPOT (MESS: IN MESSAGE);
ENTRY RETRAIT(MESS: OUT MESSAGE);
TYPE MESSAGE;
 TAMPON ARRAY(0…N-1) OF MESSAGE
NOMBRE : INTEGER := 0;
TETE, QUEUE : INTEGER RANGE 0 .. N-1 := 0;
BEGIN
LOOP
SELECT
   WHEN NOMBRE < N =>
      ACCEPT DEPOT(MESS1: IN MESSAGE)
         DO TAMPON(TETE + 1) := MESS1 END;
         TETE := (TETE + 1) MOD N;
      NOMBRE := NOMBRE + 1;
OR
   WHEN NOMBRE > 0 =>
      ACCEPT RETRAIT(MESS2: OUT MESSAGE)
         DO MESS2 := TAMPON(QUEUE + 1) END;
         QUEUE := (QUEUE + 1) MOD N ;
         NOMBRE := NOMBRE - 1;
END SELECT;
END LOOP;
END MAGASINIER;
```

```
TASK BODY PRODUCTEUR IS
BEGIN
MAGASINIER.DEPOT(MON_MESS);
END PRODUCTEUR;
```

```
TASK BODY CONSOMMATEUR IS
BEGIN
MAGASINIER.RETRAIT(SON_MESS);
END CONSOMMATEUR;
```