Synchronisation des processus

N. Hameurlain

Plan

- Spécification du problème par un exemple;
- Synchro à l'aide d'événements;
- Synchro à l'aide des sémaphores;
- Synchro par les données : la boîte aux lettres;
- Synchro forte par les données : le RdV ADA;
- Synchro par les données à l'aides des moniteurs.

Spécification du problème

- Consiste à construire un mécanisme, indépendant de la vitesse d'exécution, permettant à un processus actif
  - De « débloquer » un autre processus;
  - De se bloquer lui-même en attente d'un événement;
- L'E.M est un cas particulier de la synchronisation: un processus ne peut entrer en SC qu'après le « départ » du processus précédent.

Exemple

- Un processus *Clavier* qui lit une chaîne de caractères tapée au clavier et qui la transmet ensuite à un processus *Ecran* qui l'affiche. Le buffer correspond à une chaîne de cc de taille un

Var C: char /* partagée*/

Processus Clavier	*Processus Ecran*
Var car_lu: char;	*Var car_à_écrire: char;*
Début	*Début*
Lire(car_lu); (a)	*car_à_écrire:=C; (a')*
C:= car_lu; (b)	*afficher (car_à_écrire); (b')*
Fin.	*Fin.*

- Règle: empêcher l'*Ecran* d'exécuter (a') avant que le *Clavier* exécute (b),i.e. il faut qu'il y ait un caractère disponible.

Synchro à l'aide d'événements

- Définition: Un événement est une variable de type:

```
Type événement = enregistrement
    Survenu: booléen;
    en-attente: liste de processus
    Fin_enreg;
```

- Remarque: la structure d'un événement est identique à celle d'un verrou, la différence concerne les primitives de manipulant le structures:

Évènements (2)

```
Procedure attendre (var e: évènement)
Début
    Si e.survenu = faux alors bloquer le proc. dans e. en_attente
    Fsi
Fin
Procedure réinitialisation(var e: évènement)
Début
    e.survenu = faux;
Fin
Procedure déclencher (var e: évènement)
Début
    e.survenu = vrai;
    débloquer tous les proc. dans e.en_attente;
Fin
```

- Un événement *e* peut avoir lieu ou pas; à l'initialisation, l'événement n'a pas eu lieu, i.e, e.survenu=faux

Exemple

- Reprendre l'exemple du Clavier/Ecran, sachant que les deux processus sont Cycliques:
- On a deux types d'événements: *tamponVide, tamponPlein*

Processus Clavier	*Processus Ecran*
Var car_lu: char;	*Var car_à_écrire: char;*
Loop	*Loop*
…	…
Lire(car_lu);	*car_à_écrire:=C;*
C:= car_lu;	…
	afficher (car_à_écrire);
…	…
End loop.	*EndLoop.*

Évènements (3)

- L'inconvénient des événements, c'est qu'ils se prêtent mal au pb. de la synchro, en particulier au pb d'E.M.
- Néanmoins, on peut réaliser l'E.M par attente active, en utilisant les événements:

```
Tester_et_déclencher (e:évènement): booléen
{
   tester_et_déclencher:=e.survenu;
   déclencher(e);
   Return(tester_et_déclencher);
}
```


Évènements (4)

```
Protocole d'entrée
{
   Tant que (tester_et_déclencher(e) = vrai) /*attente*/;
}

Protocole de sortie
{
   réinitialiser (e);
}
```


Synchronisation à l'aide des sémaphores

- Fournit deux types de « RdV »:
- **Unilatéral**: un processus attend un événement produit par un autre processus. Le processus qui produit cet événement n'attend pas que l'autre processus (celui qui attend l'événement, prenne en compte cet événement.
- **Bilatéral**: le processus qui produit l'événement doit attendre que l'événement soit pris en compte avant de poursuivre son exécution ou de produire un nouvel événement.

Exemple

- Synchronisation de type « block/wake up »:
  - réalisée par un sémaphore (binaire) init. à 0; la procédure block exécute un P() sur ce sémaphore.
  - Inconvénient: des événements sont perdus si le proc. « wake up » s'exécute plus rapidement que le proc. « block »

Exemple

- Synchronisation de type Lecture/Ecriture: nécessite un « RdV » bilatéral pour empêcher que:
  - L'Ecran perd des caractères lorsque le Clavier va trop vite
  - L'Ecran affiche plusieurs fois le même caractères, car le Clavier va trop lentement.
- L'implémentation du « RdV » bilatéral nécessite 2 sémaphores.

Exemple: Ecran/Clavier

Processus Clavier	*Processus Ecran*
Var car_lu: char;	*Var car_à_écrire: char;*
Loop	*Loop*
Lire (car_lu);	*P(ce);*
P(ec); ;	*car_à_écrire:=C;*
C:= car_lu;	*V(ec);*
V(ce);	*afficher (car_à_écrire);*
End loop.	*EndLoop.*

**/Déclaration*/*
ce: sem. Init à 0;
ec: sem Init à 1;

Synchronisation par les données: la boîte aux lettres

- Une boîte aux lettre est une variable sur laquelle sont définies deux opérations: envoyer et recevoir
  - *Envoyer (b, mess):* dépose le message *mess* dans la boîte aux Lettres *b*;
  - *Recevoir (b, mess):* prélève le message *mess* de la boîte aux lettres *b*;
- Les messages contenus dans la boîte aux lettres sont gérés en queue.

Syncho. par la boîte aux lettres: sémantique

- Un proc. qui exécute recevoir (b, mess) est mis en attente si la boîte est vide.
  - Il est débloqué par le premier proc. exécutant la primitive envoyer (b, mess)
  - Donc un processus « consommateur » est bloqué jusqu'à ce qu'une donnée soit disponible dans la boîte.
- Remarque: du point de vue sémantique, un processus exécutant envoyer sur une boîte pleine sera également bloqué
  - Cette attente n'est pas de même nature que l'attente provoquée par l' exécution de recevoir lorsque la boîte est vide.

Exemple

Program principal
B: boîte aux lettres {C: char;}

Processus Clavier	*Processus Ecran*
Var car_lu: char;	*Var car_à_écrire: char;*
Loop	*Loop*
Lire(car_lu);	***recevoir(B, car_écrire);***
Envoyer(B,car_lu);	*afficher (car_à_écrire);*
End loop.	*EndLoop.*

Synchronisation forte par les données: RdV

- Introduite par [Hoare 78], elle s'exprime à l'aide de deux commandes:
  - La commande d'envoie, notée P!e, où P est un processus et e une expression
  - La commande de réception P?v, où P est un processus et v une variable
- Le processus qui exécute P!e demande l'envoi au processus destinataire P de la valeur de l'expression e.

Synchronisation forte par les données: RdV (2)

- Le processus qui exécute P?v demande de recevoir dans la variable v une valeur envoyée par le processus expéditeur P.
  - La commande de réception bloque le processus jusqu'à réception de « l'information »;
  - Il en est de même pour la commande d'envoi, qui bloque le proc. expéditeur jusqu'à ce que le processus destinataire exécute la commande de réception correspondante.

Exemple

Processus Clavier	*Processus Ecran*
Var car_lu: char;	*Var car_à_écrire: char;*
Loop	*Loop*
Lire(car_lu);	*Clavier?car_écrire;*
Ecran!car-lu;	*afficher (car_à_écrire);*
End loop.	*EndLoop.*

Synchronisation par les données: RdV ADA

- **Déclaration de processus :**
  TASK Body *<id_processus>* ***IS*** *<corps_du_processus>* ***END;***
- **Déclaration de RDV dans la spécification d'un proc. proc1 :**
  TASK *<id_processus>* ***IS***
  ENTRY <id_rendez-vous> (liste de paramètres formels);
  END;

  -- 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

- **Demande 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
  - 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.

Attente multiple

SELECT
ACCEPT *<id_RV1>(...)* ***DO*** *<liste d'instructions >* ***END;***

<liste d'instructions>;

OR
ACCEPT *<id_RV2>(...)* ***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 du Select (1)

- 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.

Sémantique du Select (2)

- 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.

Sémantique du Select (3)

- 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.
- L'intérêt du select est de permettre de pouvoir attendre des événements dans un ordre non déterminé.

Exemple: Réalisation d'une EM

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


Exemple: producteur consommateur

```
TASK MAGASINIER IS
ENTRY DEPOT (MESS: IN MESSAGE);
ENTRY RETRAIT(MESS: OUT MESSAGE);
END MAGASINIER.

TASK BODY MAGASINIER IS
    TYPE MESSAGE;
    TAMPON ARRAY(0.. N-1) OF MESSAGE;
    NOMBRE : INTEGER := 0;
    TETE, QUEUE : INTEGER RANGE 0 .. N- 1 := 0;
```


Exemple: P/C

```
BEGIN
LOOP
SELECT
   WHEN NOMBRE < N =>
      ACCEPT DEPOT(MESS1: IN MESSAGE)
         DO TAMPON(TETE ) := MESS1 END;
         TETE := (TETE + 1) MOD N;
      NOMBRE := NOMBRE + 1;
OR
   WHEN NOMBRE > 0 =>
      ACCEPT RETRAIT(MESS2: OUT MESSAGE)
         DO MESS2 := TAMPON(QUEUE ) END;
         QUEUE := (QUEUE + 1) MOD N ;
         NOMBRE := NOMBRE - 1;
END SELECT;
END LOOP;
END MAGASINIER;
```


Exemple: P/C

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

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

Synchro à l'aides des moniteurs (1)

- Proposée, par Hoare en 1974, dans le langage Portal.
- Au lieu de définir une structure de données et des primitives, on définit:
  - Une unité (Module) syntaxique, appelé *Moniteur*, assurant l'E.M.
- Un *Moniteur* regroupe des variables, ainsi que les procédures agissant sur ces variables.
- l'E.M est assurée sur les procédures déclarées dans le moniteur par la non réentrance des procédures Internes au Moniteur.

Synchro à l'aides des moniteurs (2)

- Cette notion de *Moniteur* implique que lorsqu'un processus exécute une procédure d'un *Moniteur*:
  - Tout appel d'une procédure quelconque du moniteur par un autre processus, met celui-ci en attente;
  - Le processus est débloqué dès que le processus dans le moniteur le quitte.

Gestion d'un Moniteur (1)

- En plus des procédures du moniteur, il existe deux autres procédures: *Wait* et *Send*:
  - Lorsqu'un processus, qui occupe un moniteur, exécute la procédure *Wait*, il se met en attente, et l'EM sur le moniteur est levée.
  - Lorsqu'un processus, qui occupe un moniteur, exécute la procédure *Send*, c'est le processus débloqué qui devient le proc. active du Moniteur (le processus débloqué est celui qui a exécuté la procédure Wait).
- Plusieurs processus peuvent être en attente d'un même signal: l'exécution du Send sur ce signal, débloque le processus le plus ancien (gérée en file).

Gestion d'un Moniteur (2)

- Le mécanisme de signaux géré par le Moniteur est utile pour résoudre le problème d'interblocage entre les processus, lorsque la synchronisation est gérée par les sémaphores:
- Exemple:

Mutex: sem init; à 1; S: sem init à 0;

Processus 1	**Processus 2**
P(mutex)	*P(mutex)*
Si (C1) alors P(S);	*Si (C2) alors V(S);*
V(mutex)	*V(mutex);*
end processus1;	end processus1;

États d'un processus vis à vis d'un Moniteur

- *Activable*: proc. susceptible d'obtenir le moniteur. Dans chaque Moniteur un processus au maximum peut être dans cet état;
- *Stopé*: processus susceptible de devenir activable dès qu'il n' y a plus de processus activables dans M.
  - *Stoppé_entrée*: proc. qui désire entrer dans un moniteur occupé
  - *Stoppé_send*: proc. ayant exécuté la procédure Send.
- *Attente*: proc. en attente d'un signal (à l'arrivé d'un signal, un seul proc. passe à l'état activable);
- *Sortie*: proc se trouvant en sortie du moniteur.

Sémantique états-transitions d'un processus vis à vis du Moniteur

- (sortie) entrée -> (activable)
- (sortie, activable) entrée -> (stoppé_entrée, activable)
- (activable) sig-wait -> (attente)
- (activable, stoppé_send) sig-wait -> (attente, activable)
- (activable, stoppé_entrée) sig-wait -> (attente, activable)
- (activable, attente) sig-send -> (stoppé_send, activable)
- (activable) sig-send -> (activable)
- (activable, stoppé_send) sortie -> (sortie, activable)
- (activable, stoppé_entrée) sortie -> (sortie, activable)
- (activable) sortie -> (sortie)

Exemple:clavier/Ecran