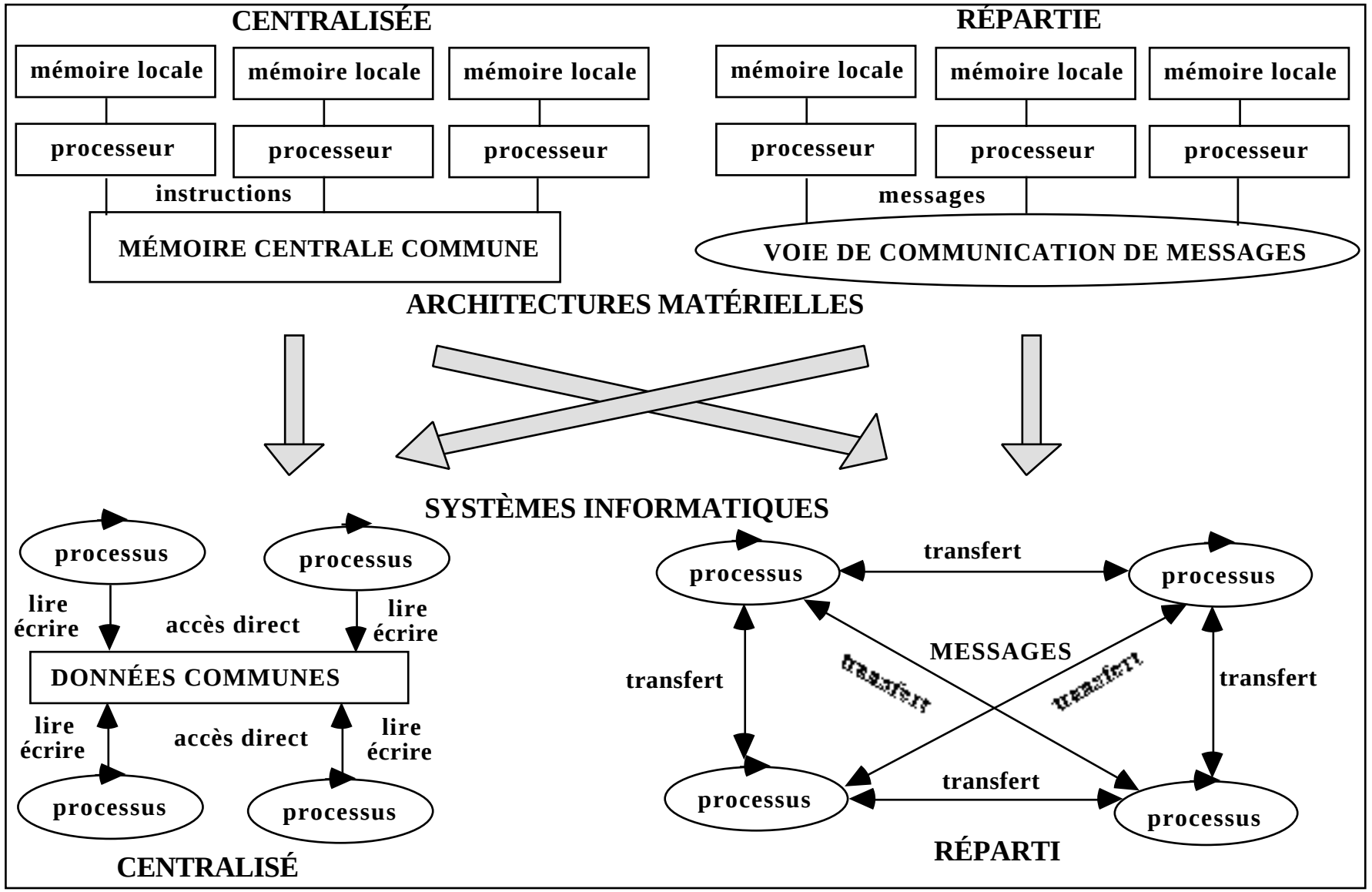

CENTRALISÉE
RÉPARTIE
mémoire locale
mémoire locale
mémoire locale
mémoire locale
mémoire locale
mémoire locale
processeur
processeur
processeur
processeur
processeur
processeur
instructions
messages
MÉMOIRE CENTRALE COMMUNE
VOIE DE COMMUNICATION DE MESSAGES
ARCHITECTURES MATÉRIELLES
SYSTÈMES INFORMATIQUES
processus
processus
lire
écrire
accès direct
lire
écrire
DONNÉES COMMUNES
lire
écrire
accès direct
lire
écrire
processus
processus
CENTRALISÉ
processus
transfert
processus
transfert
MESSAGES
transfert
transfert
transfert
processus
transfert
processus
RÉPARTI

MÉCANISMES DE SYNCHRONISATION POUR DES SYSTÈMES CENTRALISÉS

• SYSTÈMES CENTRALISÉS : les processus ont des données communes accessibles par des instructions élémentaires et des accès direct en mémoire (lire, écrire,...)

• SYSTÈMES RÉPARTIS : les processus ne communiquent que par des messages

MÉCANISME DE BASE POUR LES SYSTÈMES CENTRALISÉS : EXCLUSION MUTUELLE

Exclusion mutuelle par attente active

Cas sans multiprogrammation : un processus par processeur

- **instructions élémentaires de lecture et d'écriture**
- **instruction spéciale du type test and set**

Cas avec multiprogrammation et masquage d'interruption

Exclusion mutuelle avec libération du processeur en cas d'attente

sémaphores

moniteurs

objets protégés

Programmation modulaire avec sémaphores, moniteurs ou objets protégés

EXCLUSION MUTUELLE

HYPOTHÈSES

H1: Les vitesses relatives des processus sont quelconques.
H2: Les priorités ou les droits des processus sont quelconques.
H3: Tout processus sort de sa section critique au bout d'un temps fini; en particulier ni panne ni blocage perpétuel ne sont permis en section critique.

SPÉCIFICATION DE COMPORTEMENT

C1: Un processus au plus en section critique. Peu importe l'ordre d'accès.
C2: Pas d'interblocage actif ou passif. Si aucun processus n'est en section critique et que plusieurs processus attendent d'entrer dans leur section critique, alors l'un d'eux doit nécessairement y entrer au bout d'un temps fini. (contrexemple: déclaration et politesse)
C3: Un processus bloqué en dehors de section critique, en particulier un processus en panne, ne doit pas empêcher l'entrée d'un autre processus dans sa section critique. (contrexemple: accès à l'alternat)
C4: La solution ne doit faire d'hypothèse ni sur les vitesses, ni sur les priorités relatives des processus. De ce point de vue la solution doit être symétrique.

CONTRAINTE D'IMPLANTATION. Toute solution viable doit supposer le respect permanent de l'hypothèse H3 ou plutôt en reconstituer les conditions, en cas de panne matérielle ou logicielle. Ce rôle est assuré par le noyau du système. Par exemple, les fichiers ouverts par un programme utilisateur qui est arrêté pour cause d'erreur ou de dépassement de temps doivent être fermés. Quand le noyau tombe en panne, le système est aussi en panne.

REMARQUE. Une exclusion mutuelle par ressource avec une section critique par processus

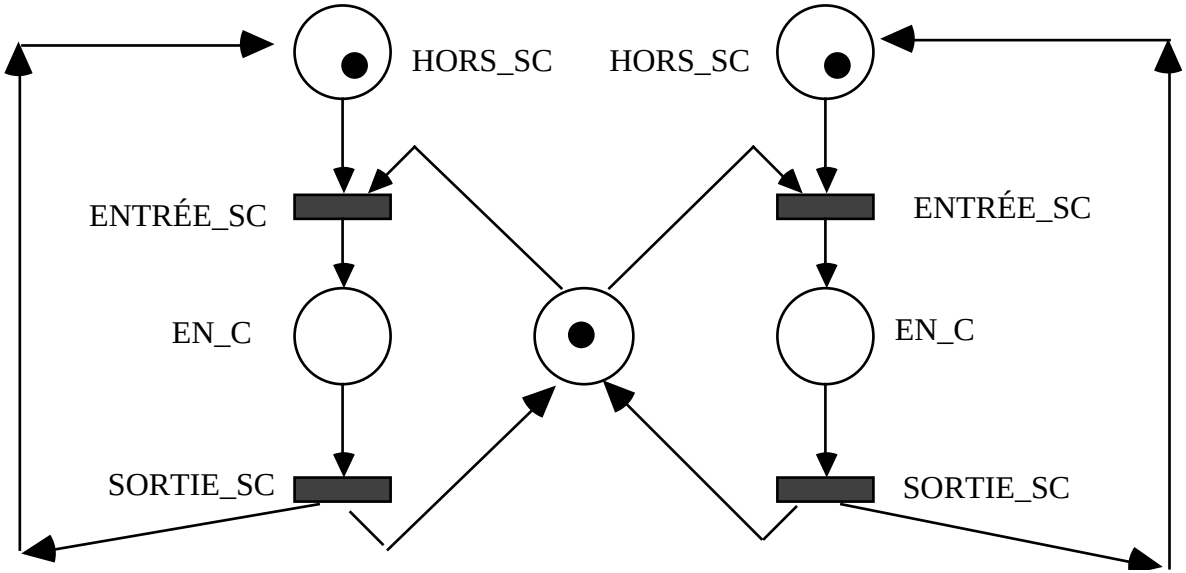
SPÉCIFICATION FORMELLE DE L'EXCLUSION MUTUELLE

1. SCHÉMA D'EXECUTION

Protocole d'initialisation
procédures de contrôle: ENTREE_SCi;
section_critique_i;
SORTIE_SCi;

2. RÉSEAU DE PETRI POUR DEUX PROCESSUS

(attention à la sémantique du modèle : tirage exclusif d'une des transitions franchissables)



PAS DE MÉCANISME D'EXCLUSION MUTUELLE

programmation erronée

```
procedure ACCES_BRUTAL is
                    COMPTE_CLIENT : Integer := 0;     -- variable commune persistante

task P is                                          • task Q is
   X : Integer := 0;  -- variable locale à P       •    Y : Integer := 0;  -- locale à Q
begin                                              • begin
   for I in 1 .. 16 loop                           •    for I in 1 .. 16 loop
      actions_hors _section_critique;              •       actions_hors _section_critique;
      X := COMPTE_CLIENT;        -- P1             •       Y := COMPTE_CLIENT;     -- Q1
      X := X + 1;                   -- P2          •       Y := Y + 1;                -- Q2
      COMPTE_CLIENT := X;        -- P3             •       COMPTE_CLIENT := Y;     -- Q3
   end loop;                                       •    end loop;
end P;                                             • end Q;

-- maintenant sont actifs les deux processus P et Q en plus du programme principal ACCES_BRUTAL
begin
   null;
end ACCES_BRUTAL;
```

Notant [ ]^{x} une boucle de x cycles, on peut tracer quelques ordonnancements significatifs. :

{COMPTE_CLIENT = 0} [P1 P2 P3]^{16} [Q1 Q2 Q3]^{16} {COMPTE_CLIENT = 32}

{COMPTE_CLIENT = 0} [P1 P2 P3 Q1 Q2 Q3]^{16} {COMPTE_CLIENT = 32}

{COMPTE_CLIENT = 0} [P1 Q1 P2 P3 Q2 Q3]^{16} {COMPTE_CLIENT = 16}

{COMPTE_CLIENT = 0} P1[Q1 Q2 Q3]^{15} P2 P3 Q1[P1 P2 P3]^{15} Q2 Q3 {COMPTE_CLIENT = 2}

EXCLUSION MUTUELLE : MÉCANISME NAÏF ET FAUX
programmation erronée

```
procedure FAUSSE_EXCLUSION is
                    COMPTE_CLIENT : Integer := 0;          -- variable commune
                    Verrou : Boolean := False;        -- initialisation du protocole d'exclusion mutuelle

task P is
  X : Integer := 0;  -- variable locale à P
begin
  for I in 1 .. 16 loop
    actions_hors _section_critique;
    -- ENTREE_SC1 avec possible attente active
    while Verrou loop null; end loop;
    Verrou := True; -- fin d'ENTREE_SC1
        X := COMPTE_CLIENT;          -- P1
        X := X + 1;                  -- P2
        COMPTE_CLIENT := X;       -- P3
    Verrou := False; -- SORTIE_SC1
  end loop;
end P;

task Q is
  Y : Integer := 0;  -- locale à Q
begin
  for I in 1 .. 16 loop
    actions_hors _section_critique;
    -- ENTREE_SC2 avec possible attente active
    while Verrou loop null; end loop;
    Verrou := True; -- fin d'ENTREE_SC2
        Y := COMPTE_CLIENT;      -- Q1
        Y := Y + 1;                  -- Q2
        COMPTE_CLIENT := Y;      -- Q3
    Verrou := False; -- SORTIE_SC2
  end loop;
end Q;

-- maintenant les deux processus P et Q sont actifs, le programme principal est aussi actif
begin
      null;
end FAUSSE_EXCLUSION ;

-- non respect de l'exclusion mutuelle, programme faux
```

EXCLUSION MUTUELLE : SOLUTIONS POUR DEUX PROCESSUS

PRINCIPE

1. Chacun des deux processus fait connaître (à l'autre) sa candidature
2. En cas de candidatures simultanées (collision), le conflit est règlé en donnant priorité à un processus
3. Pour avoir une solution équitable et respecter l'ancienneté des requêtes, la priorité doit être variable

- **Traitement du conflit dans l'algorithme de DEKKER (1965)**

La priorité indique le premier qui doit passer. Elle reste fixe pendant la résolution du conflit.
Le processus non prioritaire retire sa requête et attend de devenir prioritaire pour la redéposer
L'autre attend ce retrait avant d'entrer en section critique.
Un processus qui sort de section critique donne la priorité à l'autre pour un éventuel futur conflit.
On se ramène donc toujours à un état simple : un seul processus candidat à la section critique
C'est une solution assez compliquée et qui se généralise assez difficilement

- **Traitement du conflit dans l'algorithme de PETERSON (1981)**

La priorité indique le dernier processus qui doit passer.
Elle est changée dynamiquement avant toute entrée en section critique
Après avoir fait connaître sa candidature, chaque processus se propose comme dernier.
En cas de conflit, ceci bloque le demandeur jusqu'à l'arrivée d'un nouveau dernier.
C'est donc celui qui se propose en dernier comme dernier qui est la victime retenue.
C'est équitable car la dernière requête arrivée ne reste pas indéfiniment la dernière servie.
C'est une solution simple et qui se généralise bien (à chaque étape, on élimine un candidat)

EXCLUSION MUTUELLE : ALGORITHME DE DEKKER
cas de deux processus

```
procedure DEKKER_Pour_2 is
                COMPTE_CLIENT : Integer := 0;     -- variable commune
                Premier : Integer := 1-- initialisation du protocole d'exclusion mutuelle
                Candidat_P, Candidat_Q : Boolean := False;

task P is
  X : Integer := 0;  -- variable locale à P
begin
  for I in 1 .. 16 loop
    actions_hors _section_critique;
    -- ENTREE_SC1 avec possible attente active
    Candidat_P := True; -- demande de P
    while Candidat_Q loop   -- si conflit
      while Premier = 2 loop --l'autre d'abord
          Candidat_P := False; -- P se retire
      end loop ;  -- cela évite l'interblocage
      Candidat_P:= True;  -- Q est sorti de SC2
    end loop; --pas de conflit, fin ENTREE_SC1
        X := COMPTE_CLIENT;          -- P1
        X := X + 1;                  -- P2
        COMPTE_CLIENT := X;       -- P3
    Candidat_P:= False; -- SORTIE_SC1
    Premier := 2; -- on passe la priorité
  end loop;
end P;

task Q is
   Y : Integer := 0;  -- locale à Q
begin
  for I in 1 .. 16 loop
     actions_hors _section_critique;
     -- ENTREE_SC2 avec possible attente active
    Candidat_Q := True; -- demande de Q
    while Candidat_P loop   -- si conflit
      while Premier = 1 loop --l'autre d'abord
          Candidat_Q := False; -- Q se retire
      end loop ;  -- cela évite l'interblocage
      Candidat_Q:= True;  -- P est sorti de SC1
    end loop; --pas de conflit, fin ENTREE_SC2
         Y := COMPTE_CLIENT;     -- Q1
         Y := Y + 1;                  -- Q2
         COMPTE_CLIENT := Y;     -- Q3
    Candidat_Q := False; -- SORTIE_SC2
    Premier := 1; -- on passe la priorité
  end loop;
end Q;

begin  null;  end DEKKER_Pour_2 ;
```

EXCLUSION MUTUELLE : ALGORITHME DE PETERSON
cas de deux processus

```
procedure PETERSON_Pour_2 is
                    COMPTE_CLIENT : Integer := 0;     -- variable commune
                    Dernier : Integer := 1-- initialisation du protocole d'exclusion mutuelle
                    Candidat_P, Candidat_Q : Boolean := False;

task P is
  X : Integer := 0;  -- variable locale à P
begin
  for I in 1 .. 16 loop
    actions_hors _section_critique;
    -- ENTREE_SC1 avec possible attente active
    Candidat_P := True; -- demande de P
    Dernier := 1 ; -- se propose comme victime
    while Candidat_Q and Dernier = 1 loop
      null ;   --  attente en cas de conflit
    end loop ;       -- évite aussi l'interblocage
    -- fin d'ENTREE_SC1
        X := COMPTE_CLIENT;          -- P1
        X := X + 1;                  -- P2
        COMPTE_CLIENT := X;          -- P3
    Candidat_P:= False; -- SORTIE_SC1
  end loop;
end P;

task Q is
  Y : Integer := 0;  -- locale à Q
begin
  for I in 1 .. 16 loop
    actions_hors _section_critique;
    -- ENTREE_SC2 avec possible attente active
    Candidat_Q := True; -- demande de Q
    Dernier := 2; -- se propose comme victime
    while Candidat_P and Dernier = 2 loop
      null;        -- attente en cas de conflit
    end loop ; -- évite aussi l'interblocage
    -- fin d'ENTREE_SC2
        Y := COMPTE_CLIENT;      -- Q1
        Y := Y + 1;              -- Q2
        COMPTE_CLIENT := Y;      -- Q3
    Candidat_Q := False; -- SORTIE_SC2
  end loop;
end Q;

begin  null;  end PETERSON_Pour_2 ;
```

SOLUTION DE PETERSON POUR N PROCESSUS : On construit une suite de N-1 tournois.
A chaque tournoi, on élimine un candidat et on se ramène ainsi in fine au cas de deux processus

EXCLUSION MUTUELLE : SOLUTION DE LAMPORT POUR N PROCESSUS

PRINCIPE DE L'ALGORITHME (1974)

1. Chaque requête reçoit un ticket, de numéro plus grand que celui des autres requêtes en attente

```
Function Max_Ticket return Integer is    M : Integer := 0;
begin
   for I in IdProc loop       if Ticket(I) > M then M := Ticket(I) ; end if;       end loop;
   return M;
end max_Ticket;
```

Un processus sans requête a un ticket de valeur 0.

2. Élection de la requête de plus petit ticket (différent de 0), c'est à dire la plus ancienne
Chaque demandeur compare successivement son numéro d'ordre à celui de chacun des autres
Tant que son numéro d'ordre est plus grand, il attend; il peut attendre N-1 fois
On ne compare que des tickets stables, donc on attend la fin du calcul du ticket avant de l'utiliser

3. En cas de conflit, c'est à dire de même valeur de ticket, on utilise une priorité fixe pour départager
règle de précédence (notée =>)
x => y ⇔ (Ticket(x) < Ticket(y)) ou (Ticket(x) = Ticket(y) et priorité(x) < priorité(y))

```
Function Precede(x, y : in IdProc) return Boolean is
begin
   return (Ticket(x) < Ticket(y)) or (Ticket(x) = Ticket(y) and Priorité(x) < Priorité(y)) ;
end Precede;
```

PROPRIÉTÉS DE LA SOLUTION

Exclusion mutuelle. Pas d'interblocage. Pas de famine.
car accès à la section critique selon un ordre total défini par l'ancienneté des requêtes

LIMITATIONS DE LA SOLUTION

Complexité en temps et attente active
Les ticket() croissent indéfiniment s'il y a toujours au moins une requête en attente ou en S.C.

EXCLUSION MUTUELLE : ALGORITHME DE LAMPORT

```
procedure LAMPORT_Pour_N is            -- cas de N processus
                    COMPTE_CLIENT : Integer := 0;       -- variable commune
                    Priorite: array(1..N) of Positive; --initialisation du protocole d'exclusion mutuelle
                    Ticket : array(1..N) of Natural := (others => 0);
                    Choix : array(1..N) of Boolean := (others := False) ;

task Processus(I) is                                                    • task Processus(J) is
  X : Integer := 0;  -- variable locale à I                             •   Y : Integer := 0;  -- locale à J
begin                                                                   • begin
  Priorite(I) := I;                                                     •   Priorite(J) := J;
  loop                                                                  •   loop
    actions_hors _section_critique;                                     •     actions_hors _section_critique;
    -- ENTREE_SC1 avec possible attente active                          •     -- ENTREE_SC2 avec possible attente active
    Choix(I) := True; -- demande de I                                   •     Choix(J) := True; -- demande de J
    Ticket(I) := Max_Ticket + 1;                                        •     Ticket(J) := Max_Ticket + 1;
    Choix(I) := False; -- I a un numero d'ordre       for               •     Choix(J) := False; -- J a un numero d'ordre      for
K in 1..N loop -- I se compare à tous                                   • K in 1..N loop -- J se compare à tous
      while Choix(K) loop null; end loop;                               •       while Choix(K) loop null; end loop;
      while Ticket(K) /= 0 and Precede(K, I)                            •       while Ticket(K) /= 0 and Precede(K, J)
      loop null; end loop ; --K passe avant I                           •       loop null; end loop ; --K passe avant J
    end loop;                                                           •     end loop;
    -- fin d'ENTREE_SC1 ; I est le plus ancien                          •     -- fin d'ENTREE_SC2 ; J est le plus ancien
        X := COMPTE_CLIENT;          -- P1                              •         Y := COMPTE_CLIENT;      -- Q1
        X := X + 1;                  -- P2                              •         Y := Y + 1;              -- Q2
        COMPTE_CLIENT := X;          -- P3                              •         COMPTE_CLIENT := Y;      -- Q3
    Ticket(I) := 0;  -- SORTIE_SC11                                     •     Ticket(J) := 0; -- SORTIE_SC2
  end loop;                                                             •   end loop;
end Processus;                                                          • end Processus;
begin null; end LAMPORT_Pour_N ;
```

EXCLUSION MUTUELLE : MASQUAGE DES INTERRUPTIONS
en monoprocesseur, on empêche le changement de contexte

```
procedure Masquage_D_Interruptions_En_Monoprocesseur is  -- quel que soit le nombre de processus
                    COMPTE_CLIENT : Integer := 0;          -- variable commune persistante
                    package Interruptions is
                       procedure Masquage;       -- opération indivisible
                       procedure Demasquage;     -- opération indivisible
                    end Interruptions ;          -- initialisation du système d'interruptions

task P is
  X : Integer := 0;  -- variable locale à P
begin
  for I in 1 .. 16 loop
    actions_hors _section_critique;
    -- ENTREE_SC1 avec possible attente active
    Interruptions.Masquage;
    -- fin d'ENTREE_SC1
        X := COMPTE_CLIENT;          -- P1
        X := X + 1;                  -- P2
        COMPTE_CLIENT := X;          -- P3
    Interruptions.Demasquage;-- SORTIE_SC1
  end loop;
end P;
```

```
task Q is
  Y : Integer := 0;  -- locale à Q
begin
  for I in 1 .. 16 loop
    actions_hors _section_critique;
    -- ENTREE_SC2 avec possible attente active
    Interruptions.Masquage;
    -- fin d'ENTREE_SC2
        Y := COMPTE_CLIENT;      -- Q1
        Y := Y + 1;              -- Q2
        COMPTE_CLIENT := Y;      -- Q3
    Interruptions.Demasquage; -- SORTIE_SC2
  end loop;
end Q;
```

```
begin null;  end Masquage_D_Interruptions_En_Monoprocesseur ;

-- Au lieu de masquer les interruptions, on peut interdire à l'allocateur UC de changer de processus élu
-- la section critique doit être de courte durée (ex : pas d'entrée-sortie avec polling de périphérique)
```

EXCLUSION MUTUELLE : INSTRUCTIONS SPÉCIALES INDIVISIBLES
ces instructions imposent l'indivisibilité de la suite d'accès mémoire "lire et écrire" par un processeur

```
procedure SWAP (A, B : in out Boolean) is -- permutation cablée indivisible de deux mots de la mémoire
    Copie : Boolean; -- variable interne à l'instruction dans l'unité centrale qui l'exécute
begin Copie := A; A := B; B := Copie; end SWAP;
     -- A et B accessibles à un et un seul processeur pendant ces accès : indivisibilité de l'opération

procedure Avec_Instruction_Speciale is  -- valable quel que soit le nombre de processus
                    COMPTE_CLIENT : Integer := 0;          -- variable commune persistante
                    Verrou : Boolean := False;          -- initialisation du protocole d'exclusion mutuelle

task P is
  X : Integer := 0;      -- variable locale à P
  PasMoi : Boolean;  -- variable locale à P
begin
  for I in 1 .. 16 loop
    actions_hors _section_critique;
    -- ENTREE_SC1 avec possible attente active
    PasMoi := True ; -- à permuter avec Verrou
    loop
       SWAP(Verrou, PasMoi) ;
       exit when not PasMoi; -- sortie de boucle
    end loop;
    -- fin d'ENTREE_SC1
        X := COMPTE_CLIENT;          -- P1
        X := X + 1;                  -- P2
        COMPTE_CLIENT := X;          -- P3
    Verrou := False;-- SORTIE_SC1
  end loop;
end P;

task Q is
   Y : Integer := 0;     -- locale à Q
   PasMoi : Boolean;  -- variable locale à Q
begin
   for I in 1 .. 16 loop
     actions_hors _section_critique;
     -- ENTREE_SC2 avec possible attente active
     PasMoi := True ; -- à permuter avec Verrou
     loop
        SWAP(Verrou, PasMoi) ;
        exit when not PasMoi;
     end loop;
     -- fin d'ENTREE_SC2
         Y := COMPTE_CLIENT;      -- Q1
         Y := Y + 1;              -- Q2
         COMPTE_CLIENT := Y;      -- Q3
     Verrou := False;-- SORTIE_SC2
   end loop;
end Q;

begin null; end Avec_Instruction_Speciale ;
```

EXCLUSION MUTUELLE : INSTRUCTIONS SPÉCIALES INDIVISIBLES, CAS MULTIPROCESSEUR

```
procedure Masquage_D_Interruptions_En_Multiprocesseur is
                    -- valable quel que soit le nombre de processus
                    COMPTE_CLIENT : Integer := 0;          -- variable commune persistante
                    Verrou : Boolean := False;          -- initialisation du protocole d'exclusion mutuelle

task P is
  X : Integer := 0;  -- variable locale à P
  PasMoi : Boolean;  -- variable locale à P
begin
  for I in 1 .. 16 loop
   actions_hors _section_critique;
   -- ENTREE_SC1 avec possible attente active
   PasMoi := True ; -- à permuter avec Verrou
   loop
     Interruptions.Masquage;
      SWAP(Verrou, PasMoi) ;
      exit when not PasMoi; -- sortie de boucle
     Interruptions.Demasquage;
   end loop;
   -- fin d'ENTREE_SC1
       X := COMPTE_CLIENT;          -- P1
       X := X + 1;                  -- P2
       COMPTE_CLIENT := X;          -- P3
   Verrou := False;
   Interruptions.Demasquage;-- SORTIE_SC1
  end loop;
end P;

• task Q is
•   Y : Integer := 0;  -- locale à Q
•   PasMoi : Boolean;  -- variable locale à Q
• begin
•   for I in 1 .. 16 loop
•     actions_hors _section_critique;
•     -- ENTREE_SC2 avec possible attente active
•     PasMoi := True ; -- à permuter avec Verrou
•     loop
•       Interruptions.Masquage;
•        SWAP(Verrou, PasMoi) ;
•        exit when not PasMoi;
•       Interruptions.Demasquage;
•     end loop;
•     -- fin d'ENTREE_SC2
•           Y := COMPTE_CLIENT;      -- Q1
            Y := Y + 1;              -- Q2
            COMPTE_CLIENT := Y;      -- Q3
      Verrou := False;
      Interruptions.Demasquage; -- SORTIE_SC2
    end loop;
  end Q;

begin null; end Masquage_D_Interruptions_En_Multiprocesseur;
```

EXCLUSION MUTUELLE : BILAN DE L'ATTENTE ACTIVE

INCONVÉNIENTS DE L'ATTENTE ACTIVE

- **immobilisation du processeur simplement pour attendre**
- **congestion du bus mémoire par des accès simplement pour attendre**
- **ralentissement des autres processeurs à cause de cette congestion**
- **les solutions ne respectent pas toutes l'ancienneté des requêtes (FIFO)**

CAS D'UTILISATION DE L'ATTENTE ACTIVE

- **sections critiques de courte durée dans architectures à multiprocesseurs**
- **pour construire d'autres mécanismes de base qui eux n'auront pas d'attente active**

SOLUTIONS SANS ATTENTE ACTIVE

- **verrous (bases de données et transactions), sémaphores (généralisation)**
- **moniteur(Hoare, Brinch Hansen), méthodes synchronisées en Java, objets protégés en Ada**

LES SÉMAPHORES

• mécanisme de base permettant un auto-contrôle et éventuellement un passage en attente passive

• contrôle de concurrence sur la base d'un compte d'autorisations (positif : disponible; négatif : déficit)

• blocage du processus auto-contrôlé s'il n'y a plus d'autorisations positives lors de la demane

• restitution d'autorisations et réveil d'un demandeur bloqué

INTERFACE ET PRIMITIVES DISPONIBLES

• S sémaphore commun à tous les processus qui l'utilisent par les primitives (ou API) suivantes :

P(S) : autocontrôle : -- Puis-je (Passeren), wait(S),
consommation d'une autorisation de S: franchissement s'il en restait, sinon blocage

V(S) : réveil éventuel : -- Vas-y (Vrygeven), signal(S)
ajout d'une autorisation à S : réveil d'un processus bloqué s'il y en a

E0(S, I)) : valeur initiale de S : -- initial(S,I), S : semaphore(I)
initialisation des autorisations de contrôle à une valeur I positive ou nulle

NP(S) <= NV(S) + I

• primitives indivisibles réalisées avec les mécanismes élémentaires à attente active

• pas de contrainte sur l'ordre d'attente ou de réveil des processus bloqués : on en réveille un seul

REALISATION DES SEMAPHORES

```
type Semaphore is    record    E : Integer;
                               F : File_de_Processus_Bloqués;
                     end Semaphore;
```

• réalisation historique (Dijkstra 1965) E peut être positif, négatif ou nul; P, V et E sont indivisibles

```
procedure P(S : in out Semaphore) is -- API exécutée par le processus élu, connu par son pid = EGO
begin
   S.E := S.E -1;
   if S.E < 0 then bloquer_le_processus_élu_et_mettre_son_pid_dans_S.F;
            -- état(EGO) = bloqué ; EGO ∈ S.F ; appel de l'allocateur d'UC qui élit un autre processus
            -- au réveil de EGO, le processus continuera après P(S)
   end if;
end P;

procedure V(S : in out Semaphore) is -- API exécutée par le processus élu connu par son pid = EGO
   begin
   S.E := S.E + 1;
   if S.E <= 0 then retirer_un_processus_de_S.F_et_le_rendre_actif; -- LUI = pid du processus réveillé
   end if;                                                          -- LUI ∉ S.F ; état(LUI) = prêt
end V;   -- LUI et EGO sont prêts tous deux ; un seul est élu en monoprocesseur ; politiques de choix

procedure E0(S : in out Semaphore; I in Natural) is
begin S.E := I; S.F := ∅; end E0;                    -- I ≥ 0
```

EXCLUSION MUTUELLE PAR SÉMAPHORE

```
procedure Exclusion_Mutuelle_Par_Semaphore is  -- quel que soit le nombre de processus
                    COMPTE_CLIENT : Integer := 0;          -- variable commune persistante
                    S : semaphore;  -- E0(S,1) ; -- initialisation avec une seule autorisation E.S = 1

task P is
   X : Integer := 0;  -- variable locale à P
begin
   for I in 1 .. 16 loop
     actions_hors _section_critique;
     P(S) ; -- ENTREE_SC1 avec attente possible
          X := COMPTE_CLIENT;          -- P1
          X := X + 1;                  -- P2
          COMPTE_CLIENT := X;          -- P3
     V(S); -- SORTIE_SC1
   end loop;
end P;

task Q is
   Y : Integer := 0;  -- locale à Q
begin
   for I in 1 .. 16 loop
     actions_hors _section_critique;
     P(S) ; -- ENTREE_SC2 avec attente possible
          Y := COMPTE_CLIENT;      -- Q1
          Y := Y + 1;              -- Q2
          COMPTE_CLIENT := Y;      -- Q3
     V(S); -- SORTIE_SC2
   end loop;
end Q;

begin
    E0(S,1) ;     -- initialisation du sémaphore, avant démarrage de P et de Q
end Exclusion_Mutuelle_Par_Semaphore ;
```

MODULARITÉ

CONTRAINTES D'UTILISATION DE L'EXCLUSION MUTUELLE ET CRITIQUE DES SÉMAPHORES

• respect des spécifications de programmation =>
pas d'entrée incontrôlée (hors protocole) dans le code de la section critique
pas de sortie incontrôlée (hors protocole) du code de la section critique
attention : exceptions, erreurs, trappes, destruction du processus en section critique
• respect des spécifications comportementales (toute section critique est de durée finie) =>
pas de boucle infinie en section critique
pas de blocage en section critique si le réveil implique la ressource critique protégée
pas de destruction du processus en section critique
• parfois besoin de spécifier l'attente : FIFO, priorités, délai maximal, équité (absence de famine)
• le sémaphore est un mécanisme simple et puissant, mais dangereux sans modularité et limite de portée

MODULARITÉ NÉCESSAIRE

• regrouper et encapsuler les données partagées à contrôler et les mécanismes qui les contrôlent
• mécanismes de contrôle associés aux méthodes d'accès (procédures et fonctions) et pas ailleurs
• distinguer l'interface (accessible par l'extérieur) et l'implantation (cachée à l'extérieur)

SÉMAPHORES : encapsuler dans un paquetage les sections critiques d'une même ressource (explicite)
MONITEURS (Hoare, Brinch Hansen) : structure modulaire déclarée avec exclusion mutuelle implicite
JAVA : méthodes déclarées "synchronized" comme sections critiques dans une classe d'objet
ADA : objet protégé : structure modulaire déclarée avec exclusion mutuelle implicite

EXCLUSION MUTUELLE MODULAIRE AVEC SÉMAPHORE

```
procedure Exclusion_Mutuelle_Modulaire_Par_Semaphore is  -- quel que soit le nombre de processus

   package Compte is procedure Section_critique_1;  procedure Section_critique_2; end Compte;

   package body Compte is     -- on encapsule la donnée critique et son sémaphore de contrôle
                        COMPTE_CLIENT : Integer := 0;         -- variable commune persistante
                        S : semaphore; -- à initialiser avec une seule autorisation E.S = 1
      procedure body Section_critique_1 is
         X : Integer := 0;  -- variable locale
      begin
         P(S) ; -- ENTREE_SC1
            X := COMPTE_CLIENT;        -- P1
            X := X + 1;                -- P2
            COMPTE_CLIENT := X;        -- P3
         V(S); -- SORTIE_SC1
      end Section_critique_1 ;
      procedure body Section_critique_2 is
         Y : Integer := 0;  -- variable locale
      begin
         P(S); -- ENTREE_SC2
               Y := COMPTE_CLIENT;      -- Q1
               Y := 2*Y + 3;            -- Q2
               COMPTE_CLIENT := Y;      -- Q3
         V(S); -- SORTIE_SC2
      end Section_critique_2 ;
   begin E0(S,1) ;    -- initialisation du semaphore, avant utilisation du paquetage
   end Compte;        -- fin du module partagé, unité de bibliothèque

task P is
begin
   for I in 1 .. 16 loop
         actions_hors _section_critique;
         Compte.Section_critique_1;
   end loop;
end P;
                                                    •
                                                    •
                                                    •
                                                    •
                                                    •
                                                    •
task Q is
begin
   for I in 1 .. 16 loop
         actions_hors _section_critique;
         Compte.Section_critique_2;
   end loop;
end Q;

begin null; end Exclusion_Mutuelle_Modulaire_Par_Semaphore ;
```

EXCLUSION MUTUELLE MODULAIRE PAR MONITEUR

```
procedure Exclusion_Mutuelle_Modulaire_Par_Moniteur is  -- quel que soit le nombre de processus

   Compte : monitor;
   -- on encapsule la donnée critique et le contrôle d'exclusion mutuelle est implicite
                        var COMPTE_CLIENT : Integer ;        -- variable commune persistante
      procedure Section_critique_1 is                       procedure Section_critique_2 is
         X : Integer := 0;  -- variable locale                 Y : Integer := 0;  -- variable locale
      begin                                                 begin
            X := COMPTE_CLIENT;        -- P1                       Y := COMPTE_CLIENT;      -- Q1
            X := X + 1;                -- P2                       Y := 2*Y + 3;            -- Q2
            COMPTE_CLIENT := X;        -- P3                       COMPTE_CLIENT := Y;      -- Q3
      end Section_critique_1 ;                              end Section_critique_2 ;
   begin COMPTE_CLIENT := 0;  -- initialisation, avant utilisation du moniteur
   end Compte;                          -- fin de déclaration du moniteur partagé
```

```
task P is                                  •  task Q is
begin                                      •  begin
   for I in 1 .. 16 loop                   •     for I in 1 .. 16 loop
         actions_hors _section_critique;   •           actions_hors _section_critique;
         Compte.Section_critique_1;        •           Compte.Section_critique_2;
   end loop;                               •     end loop;
end P;                                     •  end Q;
```

```
begin null; end Exclusion_Mutuelle_Modulaire_Par_Moniteur ;
```

EXCLUSION MUTUELLE MODULAIRE EN JAVA

```
public class Compte {
      // on encapsule la donnée critique et le contrôle d'exclusion mutuelle est implicite
   private int COMPTE_CLIENT = 0          // variable persistante, commune pour les objets de la classe

   public synchronized void Section_critique_1(){
         int X = COMPTE_CLIENT;      -- P1
         X := X + 1;                 -- P2
         COMPTE_CLIENT := X;         -- P3
   }     // fin de Section_critique_1 ;

   public synchronized void Section_critique_2(){
         int Y = COMPTE_CLIENT;      -- Q1
         Y := 2*Y + 3;               -- Q2
         COMPTE_CLIENT := Y;         -- Q3
   }     // fin de Section_critique_2 ;
}    // fin de la classe Compte
```

```
public class P extends thread{
   private Compte compte;
   P(Compte compte){this.compte = compte;}
   public void run(){
      for (int I=1; I <16; I++){
         actions_hors _section_critique;
         Compte.Section_critique_1;
   }
}    // fin de P
```

```
public class Q extends thread{
   private Compte compte;
   Q(Compte compte){this.compte = compte;}
   public void run(){
      for (int I=1; I <16; I++){
         actions_hors _section_critique;
         Compte.Section_critique_2;
   }
}    // fin de Q
```

```
public class Exclusion_Mutuelle_Modulaire_En_Java {     // idem quel que soit le nombre de processus
   static Compte c = new Compte();
   public static void main(String[] args) {             // programme principal
      P p = new P(c); Q q = new Q(c);                   // on doit passer en paramètre les objets partagés
      p.start(); q.start();                             // lance les processus
   }                                                    // fin du programme principal main
}                                                       // fin de Exclusion_Mutuelle_Modulaire_En_Java
```

EXCLUSION MUTUELLE MODULAIRE AVEC OBJET PROTÉGÉ EN ADA

```
procedure Exclusion_Mutuelle_Par_Objet_Protege is  -- idem quel que soit le nombre de processus
```

```
   protected Compte is
      procedure Section_critique_1;          -- l'accès se fait uniquement en exclusion mutuelle
      procedure Section_critique_2;          -- l'accès se fait uniquement en exclusion mutuelle
   private-- on encapsule la donnée critique et l'exclusion mutuelle est implicite
      COMPTE_CLIENT : Integer := 0;          -- variable commune persistante
end Compte;                                  -- fin de déclaration de l'interface de l'objet protégé
```

```
   protected body Compte is
      procedure body Section_critique_1 is
         X : Integer := 0;  -- variable locale
      begin
            X := COMPTE_CLIENT;        -- P1
            X := X + 1;                -- P2
            COMPTE_CLIENT := X;        -- P3
      end Section_critique_1 ;
      procedure body Section_critique_2 is
         Y : Integer := 0;  -- variable locale
      begin
            Y := COMPTE_CLIENT;        -- Q1
            Y := 2*Y + 3;              -- Q2
            COMPTE_CLIENT := Y;        -- Q3
      end Section_critique_2 ;
   end Compte;                          -- fin de déclaration du corps de l'objet protégé
```

```
task P; task body P is
begin
   for I in 1 .. 16 loop
         actions_hors _section_critique;
         Compte.Section_critique_1;
   end loop;
end P;
```

•

```
task Q; task body Q is
begin
   for I in 1 .. 16 loop
         actions_hors _section_critique;
         Compte.Section_critique_2;
   end loop;
end Q;
```

```
begin null; end Exclusion_Mutuelle_Modulaire_Par_Semaphore ;
-- on appréciera la clarté et la simplicité de l'écriture
```

CONTRÔLE DE CONCURRENCE ET SYNCHRONISATION DES PROCESSUS

• l'exclusion mutuelle ne suffit pas pour traiter tous les aspects de la synchronisation et le contrôle entre les processus concurrents
• il faut aussi pouvoir bloquer un processus et réveiller un processus

SÉMAPHORES

on peut traiter tous les cas avec le seul mécanisme des sémaphores (voir la suite du cours)

MONITEURS (ET JAVA)

• le moniteur ajoute le type condition et une file d'attente associée à chaque variable de type condition
ainsi que des primitives wait x, signal x, avec x : condition
Un processus se bloque par wait x et libère le moniteur
Un processus réveille un autre processus par signal x, mais un seul processus utilise le moniteur, l'autre doit attendre la libération du moniteur
• En java cela s'appelle wait(), notify(), notifyAll()
et il n'y a qu'une condition anonyme avec une seule file d'attente par moniteur

OBJETS PROTÉGÉS

• les conditions sont remplacées par des gardes sur les appels de procédure, transformées en entrées gardées par des expressions booléennes. Un processus appelant une procédure ou une entrée s'exécute en exclusion mutuelle. La fin d'une exécution d'entrée soit s'accompagne d'un retour du processus vers la procédure appelante, soit bloque à nouveau le processus si l'instruction requeue nom_d_entrée est programmée. L'objet protégé est libéré pour un autre processus. Avec les gardes et le requeue, on peut programmer l'utilisation d'un objet protégé comme un automate synchronisé.