Processus concurrents et parallélisme

Chapitre 3 - Synchronisation

Gabriel Girard

17 octobre 2022

Chapitre 3 - Synchronisation

Chapitre 3 - Synchronisation

Pourquoi la synchronisation ?

- Deux processus peuvent s'exécuter en même temps s'ils sont disjoints
- Pas très commode ! ! ! ! ! ! !
- Les processus partagent souvent des ressources

Principe de base de la synchronisation

- Comportement anormal dû aux interférences incontrôlées
- On élimine ce comportement si on empêche le chevauchement des points non disjoints
- Il suffit de contrôler l'ordonnancement des événements dans le temps
- On appelle cet ordonnancement la synchronisation

Définition et utilité

Définition

La synchronisation est donc un terme général pour toutes les contraintes sur l'ordonnancement des opérations dans le temps

La synchronisation permet à des processus non disjoints de s'exécuter concurremment et de produire de bons résultats

Types de synchronisation

1. Synchronisation conditionnelle
2. Exclusion mutuelle

Qu'est-ce que la communication ?

- La coopération inter-processus implique une certaine forme de communication
- La communication permet à l'exécution d'un processus d'influencer l'exécution d'un autre processus
- Elle se fait par des variables communes ou par messages

Exemples de communication

- Deux processus accédant une même variable x
- Producteur/consommateur

Producteur/consommateur

- On utilise un ensemble de tampons
- Remplis par le producteur
- Vidés par le consommateur
- Doivent se synchroniser pour ne pas consommer un élément non produit ou produire dans un tampon contenant déjà un message non-consommé
- Ne doivent pas accéder simultanément à la structure de donnée dans le but de la modifier

```
type   item = ...
var    tampon : array[0..n-1] of item;
       in, out : 0..n;
       nextp, nextc : item;

in := 0; out := 0;
parbegin
```

// Producteur	// Consommateur

while (true)
  ...
  produire un item
  ...
  while((in+1 %n)=out);
  tampon[in] := nextp;
  in := in + 1 mod n;
	While (true)
	  while(in=out) ;
	  nextc := tampon[out]
	  out := out+1 mod n;
	  ...
	  traiter un item
	  ...

```
parend
```

```
type    item = ...
type    tampon : record     element : item ;
                            suiv : pointer to tampon ;
                 end ;
var     prem, p, c : pointer to tampon ;
nextp, nextc : item ;
prem := nil ;
parbegin
```

```
//      Producteur

while (true)
      ...
      produire item nextp
      ...
      new(p);              // 2
      p.elem := nextp;// 3
      p.suiv := prem; // 4
      prem := p;           // 5
```

```
//      Consommateur

while (true)
      while (prem=nil);
      c := prem;           // 1
      prem:=prem.suiv;// 6
      nextc := c.elem;// 7
      dispose(c);          // 8
      ...
      traiter item nextc
      ...
```

```
parend
```

Chapitre 3 - Synchronisation

Section critique et atomicité

- Pour remédier aux problèmes, on doit regrouper et synchroniser l'exécution des énoncés qui manipulent la liste
- Ces énoncés doivent s'exécuter en exclusion mutuelle

Section critique (SC)

Séquence d'instructions qui doit s'exécuter en exclusion mutuelle

Aussi appelé **atomicité**

Problème !

Comment assurer l'exclusion mutuelle ?

- On ajoute un protocole avant et après chaque section critique
- Ces protocoles assureront l'exclusion mutuelle

```
process Pi(i=1..n)
  loop
     ... section non-critique ...

     protocole d'entrée
     section critique (SC)
     protocole de sortie

  endloop
```

Contraintes

Contraintes à respecter pour que les solutions soient acceptables :

Contraintes

Contraintes à respecter pour que les solutions soient acceptables :

1. aucune supposition sur le matériel (sauf atomicité des instructions)

Contraintes

Contraintes à respecter pour que les solutions soient acceptables :

1. aucune supposition sur le matériel (sauf atomicité des instructions)
2. aucune supposition sur les vitesses relatives des processus

Contraintes

Contraintes à respecter pour que les solutions soient acceptables :

1. aucune supposition sur le matériel (sauf atomicité des instructions)
2. aucune supposition sur les vitesses relatives des processus
3. un processus qui n'est pas en SC ne doit pas pouvoir empêcher les autres processus d'entrer dans leur SC

Contraintes

Contraintes à respecter pour que les solutions soient acceptables :

1. aucune supposition sur le matériel (sauf atomicité des instructions)
2. aucune supposition sur les vitesses relatives des processus
3. un processus qui n'est pas en SC ne doit pas pouvoir empêcher les autres processus d'entrer dans leur SC
4. on ne doit pas remettre indéfiniment la décision qui consiste à admettre un processus, parmi plusieurs, en SC

Attente active

- Un processus boucle sur une condition fausse jusqu'à ce qu'elle soit vraie
- Le processus teste de façon répétitive la condition (attente active)

Problème simplifié

- Le problème général est complexe
- Simplification : solution avec deux processus

```
process Pi(i=1..2)
  loop
     section non critique
     protocole d'entrée
     section critique
     protocole de sortie
  endloop
```

Algorithme 1

```
var  libre: boolean;
begin
  libre := vrai;
  parbegin
   P1: repeat
          while not libre;
          libre := faux;
          section critique
          libre := vrai;
          section non-critique
       forever;
   P2: repeat
          while not libre;
          libre := faux;
          section critique
          libre := vrai;
          section non-critique
       forever;
  parend
end.
```

Algorithme 2

```
var tour: integer;
begin
  tour := 1;  (ou 2)
  parbegin
    P1 : repeat
           while tour = 2 do /* rien */ ;
           section critique
           tour := 2;
           section non-critique
         forever;
    P2 : repeat
           while tour = 1 do /* rien */;
           section critique
           tour := 1;
           section non-critique
         forever;
  parend
end.
```

Algorithme 3

```
var  c1, c2: boolean;
begin
  c1 := c2 := faux;
  parbegin
    P1 : repeat
           while c2 do /*rien*/;
           c1  := vrai;
           section critique
           c1 := false;
           section non-critique
         forever;
    P2 : repeat
           while c1 do /*rien*/;
           c2 := vrai;
           section critique
           c2 := faux;
           section non-critique
         forever;
 parend
end.
```

Algorithme 4

```
var  c1, c2: boolean;
begin
  c1 := c2 := faux;
  parbegin
     P1 : repeat
             c1  := vrai;
             while c2 do /*rien*/;
             section critique
             c1 := false;
             section non-critique
          forever;
     P2 : repeat
             c2 := vrai;
             while c1 do /*rien*/;
             section critique
             c2 := faux;
             section non-critique
          forever;
  parend
end.
```

Algorithme 5

```
var c1, c2: boolean;
c1 := c2 := faux;
parbegin
  P1:while(1) { c1 := vrai;
                while(c2)   { c1 := faux;
                              while (c2) /*rien*/;
                              c1  := vrai;       }
                ...section critique
                c1 := false;
                ... section non-critique ...  }

  P2:while(1) { c2 := vrai;
                while(c1)   { c2 := faux;
                              while (c1) /*rien*/;
                              c2  := vrai;       }
                ... section critique
                c2 := false;
                ... section non-critique... }
 parend
```

Algorithme 6 - Algorithme de Dekker

```
var c1, c2: boolean;  tour : integer;
c1 := c2 := faux; tour := 1;
parbegin
  P1:while(1) { c1:=vrai;
                while (c2) if tour=2 { c1 := faux;
                                       while (tour=2);
                                       c1  := vrai; }
                ... section critique
                c1 := false;  tour := 2;
                ... section non-critique...  }

  P2:while(1) { c2  := vrai;
                while (c1) if tour=1 { c2:=faux;
                                       while (tour=1);
                                       c2 :=vrai;   }
               ...section critique
                c2 := false; tour := 1;
               ... section non-critique... }
parend
```

Preuve du bon fonctionnement

- Exclusion mutuelle garantie
  Vrai car P_j entre seulement si $C_i = \text{faux}$

- Pas d'interblocage (contraintes 3-4)
  1. P_i est le seul à demander l'accès
  2. P_i et P_j demandent l'accès
     Vitesse ou *tour* empêchent l'interblocage

Algorithme 7 (généralisation à n processus)

```
begin /* programme */
  flag := idle; /* pour tous */
  tour := ?;  /* une valeur entre 0 et N-1 */
  parbegin
      P(1); P(2); P(3); P(4); ... P(N-1);
  parend;
end. /* programme */
```

Algorithme 7 - Algorithme de Dijkstra

```
var  flag: array[0..N-1] of (idle, want-in, in-cs);
     tour : 0..N-1;
Procedure P(i : integer)
   var  j : integer;
begin /* procedure */
  repeat
  {  repeat
     { flag[i]:=want-in;
       while (tour!=i)
       {       if (flag[tour]=idle) tour := i;  }
       flag[i] := in-cs;  j:= 0;
       while (j< N) and (j=i or flag[j] != in-cs)
            j:= j+1;

     } until j>=N;
...section critique
     flag[i]:=idle;
...section non-critique
  } forever;
end;  /* procedure */
```

Preuve du bon fonctionnement

- Exclusion mutuelle garantie
  Vrai car
  1. P_j entre seulement si tous les $flag[i] \neq$ `in-cs`
  2. P_j teste $flag[i]$ après avoir modifié le sien

- Pas d'interblocage (contraintes 3-4)
  1. $flag_i =$ `in-cs` n'implique pas que $tour = i$
  2. si $tour = i$ et $flag_i \neq idle$, $tour$ ne sera plus modifié
  3. au tour suivant, un seul passe

Contrainte supplémentaire

5. Il doit y avoir un nombre fini de processus autorisés à passer en SC après qu'un processus quelconque ait fait une demande d'entrée et avant que cette entrée soit autorisée.

Algorithme 8 - Algorithme de Eisenberg et McGuire

```
Procedure P(i : integer)
  var  j : integer;
begin /* procedure */
  repeat
  {  repeat
     {  flag[i] := want-in;
        j := tour;
        while (j!=i) if flag[j]!=idle  then j:=tour;
                                       else j:=j+1 mod N;
        flag[i] := in-cs;  j:= 0;
        while (j<N) and (j = i or flag[j]!=in-cs) do
                            j:= j+1;
     } until (j>=N) and (tour=i or flag(tour)=idle);
     tour := i;
...section critique ...
     j:=tour+1 mod N;
     while ((j!=tour) and (flag[j]=idle)) j:=j+1 mod N;
     tour := j;  flag[i] := idle;
... section non-critique ...
  } forever;
end;  /* procedure */
```

Algorithme 8 - Algorithme de Eisenberg et McGuire

```
Procedure P(i : integer)
  var  j : integer;
begin /* procedure */
  repeat
  { repeat
    { flag[i] := want-in;
      j := tour;
      while (j!=i) if flag[j]!=idle then j:=tour;
                                    else j :=j+1 mod N;
      flag[i] := in-cs;  j:= 0;
      while (j<N) and (j = i or flag[j]!=in-cs) do
      j := j+1;
    } until (j>=N) and (tour=i or flag(tour)=idle);
    tour := i;
...section critique ...
    j :=tour+1 mod N;
    while ((j!=tour) and (flag[j]=idle)) j:=j+1 mod N;
    tour := j;  flag[i] := idle;
... section non-critique ...
  } forever;
end;  /* procedure */
```

Algorithme de la boulangerie (2 processus)

```
var  c1, c2, n1, n2: boolean;
begin
  c1 := c2 := n1 := n2 := 0;
  parbegin
  P1 : repeat
          c1 := 1; n1 := n2 + 1; c1 := 0;
          while c2!=0 do /* rien */;
          while (n2 != 0) and (n2<n1) do /*rien*/;
          .... section critique
          n1 := 0;
          .... section non-critique
       forever;
  P2 : repeat
          c2:=1; n2 := n1 + 1; c2 :=0;
          while c1!=0  do /*rien*/;
          while (n1!=0) and (n1 <= n2) do /*rien*/;
          .... section critique
          n2 := 0;
          .... section non-critique
       forever;
  parend
end.
```

Algorithme de la boulangerie 2 (2 pcs)

```
var n1, n2:  boolean;
begin
  n1 := n2 := 0;
  parbegin
  P1 :  repeat
           n1 := 1; n1 := n2 + 1;
           while (n2!=0) and (n2<n1) do /*rien*/;
        .... section critique
           n1 := 0;
        .... section non-critique
        forever;
  P2 :  repeat
           n2 := 1; n2 := n1 + 1;
           while (n1!=0) and (n1<=n2)  do /*rien*/;
        .... section critique
           n2 := 0;
        .... section non-critique
        forever;
  parend
end.
```

Algorithme 11 - Algorithme la boulangerie (n processus)

```
var  choosing : array[0..n-1] of boolean;
     number   : array[0..n-1] of integer;
begin
  choosing[0..n-1]:=faux;
  number[0..n-1] := 0;
  process Pi(1..n)
  { repeat
    {  choosing[i] := vrai;
       number[i] := max(number[0], ..., number[n-1])+1;
       choosing[i] := faux;
       for (j:=0 to n-1)
       { while choosing[j] do /*rien*/;
         while (number[j]!=0) and
               ((number[j],j)<(number[i],i)) do /*rien*/;
       }
   .... section critique
       number[i] := 0;
   .... section non-critique
    } forever;
  }
end
```

Algorithme 12 - Algorithme de Peterson (2 processus)

```
begin /* programme */
   flag[0] := flag[1] := faux;
   tour := ?;  /*une valeur entre 0 et 1 */
   parbegin
      P(0); P(1);
   parend;
end. /*programme*/
```

Algorithme 12 - Algorithme de Peterson (2 processus)

```
var flag : array[0..1] of boolean;
    tour : 0..1;
Procedure P(i : integer); /* i=0 ou 1 et j=i+1 mod 2*/
    var j : integer;
begin /* procedure */
   j := i + 1 mod 2;
   repeat
      flag[i] := vrai;
      tour := j;
      while (flag[j] and tour = j) /*rien*/;
   .... section critique
      flag[i] := faux;
   .... section non-critique
   forever;
end; /*procedure*/
```

Preuve du bon fonctionnement

- Exclusion mutuelle garantie
  Supposons que les deux sont en section critique....
  1. $flag_i = flag_j =$ vrai
  2. $tour = i$ ou $tour = j$ avec affectation défavorable
- Pas d'interblocage (contraintes 3-4)
  1. $flag_i =$ faux alors P_j passe
  2. Si $flag_i = flag_j =$ vrai alors $tour$ débloque un processus

Algorithme 13 - Algorithme de Peterson (n processus)

```
begin /* programme */
   flag[0..n-1] := -1;
   tour[0..n-2] := 0;
   parbegin
      P(0); P(1); P(2); ...; P(n-1);
   parend;
end. /* programme */
```

Algorithme 13 - Algorithme de Peterson (n processus)

```
var   flag : array[0..n-1] of -1..n-2;
      tour : array[0..n-2] of 0..n-1;
Procedure P(i : integer);
var j : integer;
begin /*procedure*/
  repeat
  { for (j:=0 to n-2)
    {  flag[i] := j;
       tour[j] := i;
       Repeat  /*rien*/
       until ((forall k!=i : flag[k] < j) or (tour[j] != i));
    }
  .... section critique
     flag[i] := -1;
  .... section non-critique
  } forever;
end;  /* procedure */
```

Preuve du bon fonctionnement

- Exclusion mutuelle garantie
- Pas d'interblocage ni famine (contraintes 3-4)

Algorithme avec instructions machines

- Certaines machines fournissent des instructions spéciales qui permettent à un processus de tester et modifier le contenu de la mémoire ou d'échanger le contenu de deux zones mémoire de façon atomique.
- Exemple : tst, swap, fadd, rmw, ...

tst et swap

```
procedure tst(var a,b : boolean)
begin
    a:=b;
    b:=true;
end

procedure swap(var a,b : boolean)
begin
    var temp : boolean;
    temp := a;
    a := b;
    b := temp;
end
```

Exclusion mutuelle avec tst

```
var active : boolean := false;

process P(i:=1 to n)
  var libre : boolean;

  repeat
     libre := true;
     while libre do tst(libre, active);
  ... section critique
     active := false;
  forever;
```

Exclusion mutuelle avec swap

```
var active : boolean := false;

process P(i := 1 to n)
   var cle : boolean;

   repeat
      cle := true;
      while (cle = TRUE)
         swap(active,cle);
      endwhile
   ..... section critique
      active := false;
  forever;
```

C++

- `atomic_compare_exchange`
- `atomic_fetch_and_add`

Problèmes avec les algorithmes d'attente active

- Difficile à concevoir et à prouver correct
- Interblocage possible à cause des politiques
- Utilisation inutile du temps de la machine
- Synchronisation dépend de l'usager
- Solutions peu lisibles (utilisation des variables, ...)
- Solutions difficiles à généraliser

Chapitre 3 - Synchronisation

Introduction

- Introduits par Dijkstra dans les années 60
- Un sémaphore est une variable entière sur laquelle on définit deux opérations atomiques : P (wait) et V (signal)
- Un sémaphore peut être initialisé à une valeur $n \geq 0$

Introduction

- Soit un sémaphore S
- $P(S)$ bloque le processus appelant jusqu'à ce que $S > 0$
- Une file d'attente est associée à chaque sémaphore pour contenir les processus bloqués.
- $V(S)$ débloque le 1^{er} processus en attente s'il y en a un, sinon les signaux s'accumulent

Implantation

```
class semaphore {
      int valeur;
      listeDePcs liste;
    public:
      void P();
      void V();
}
```

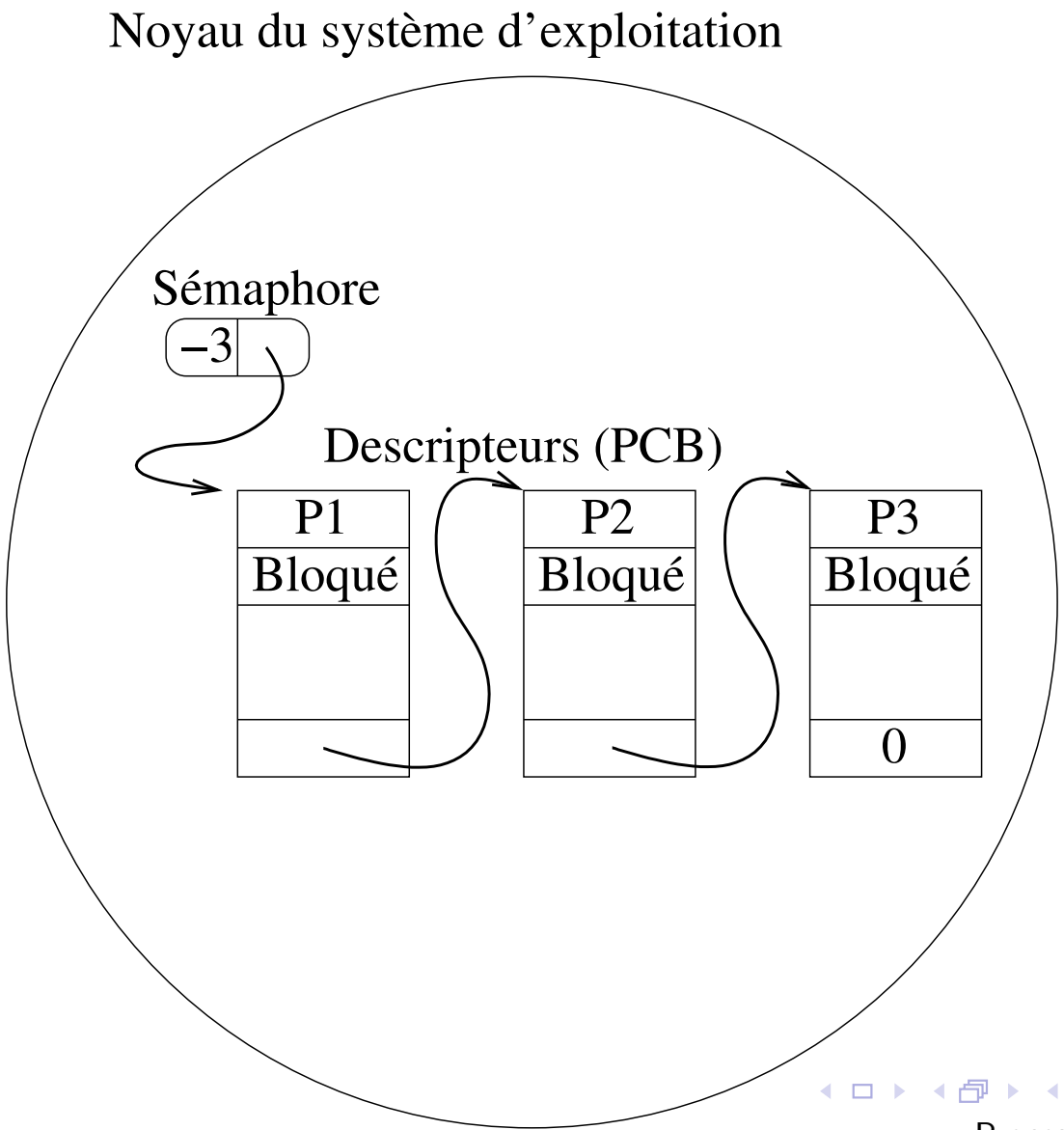
```
semaphore::void P()
{  valeur--;
   if (valeur < 0)
   {    état du processus courant := bloqué;
        liste.ajoute(processus courant);
   }
}
semaphore::void V()
{  process processus;
   valeur++;
   if (valeur <= 0)
   {   processus := liste.retire();
       processus.etat := prêt;
   }
}
```

```
semaphore::void P()
{  if (valeur = 0)
   {    état du processus courant := bloqué;
        liste.ajoute(processus courant);
   }
   else  valeur--;
}
semaphore::void V()
{  process processus;
   if (!liste.vide())
   {   processus := liste.retire();
       processus.etat := prêt;
   }
   else valeur++;
}
```

Problèmes

- File d'attente :
  - on l'implante où et comment ?
  - son implantation assure ou non l'équité...
- Atomicité :
  - les opérations P et V doivent être atomiques (sections critiques)
  - comment y parvenir (mono et multi processeur) ?

Implantation des sémaphores dans le noyau



Exclusion mutuelle

```
semaphore mutex;

mutex.init(1);

    repeat
        P(mutex)
    .....section critique
        V(mutex)
    .....section non-critique
    forever
```

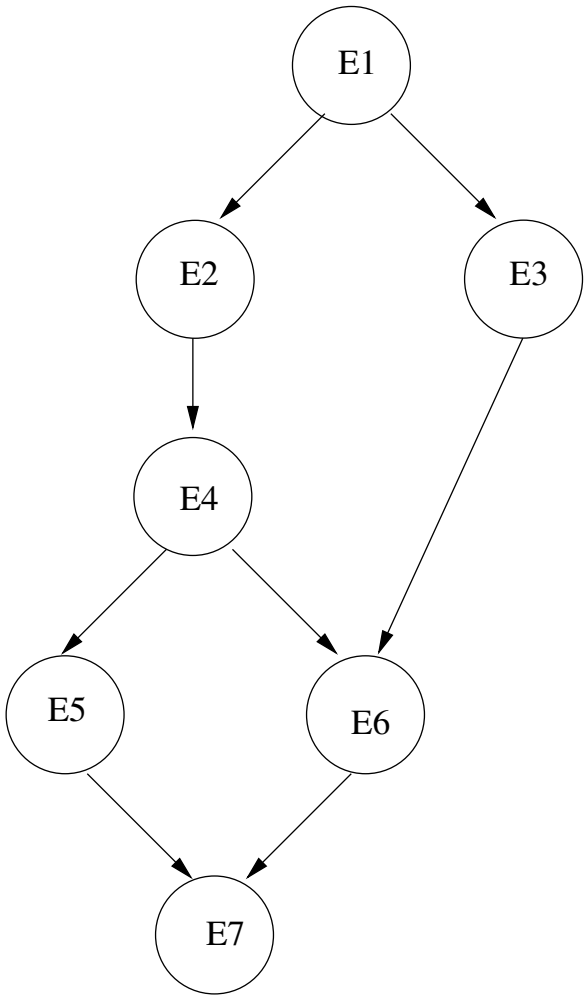
Synchronisation conditionnelle

```
semaphore cond;

cond.init(0);
```

```
P1 : .                    P2 : .
     .                         .
     .                         .
     S1;                       P(cond);
     V(cond);                  S2;
     .                         .
     .                         .
     .                         .
```

Synchronisation conditionnelle




```
Var a,b,c,d,e,f,g : semaphores (=0)
begin
 parbegin
   begin E1; V(a); V(b); end;
   begin P(a); E2; E4; V(c); V(d); end;
   begin P(b); E3; V(e); end;
   begin P(c); E5; V(f); end;
   begin p(d); P(e); E6; V(g); end;
   begin P(f); P(g); E7; end;
 parend;
end;
```

Limiter les accès

- Soit un tampon contenant n espaces
- On peut initialiser un sémaphore à n pour détecter que le tampon est plein
- Utilisé dans le problème des producteurs/consommateurs
- Sémaphore général

Types de sémaphore

- Sémaphore général (counting semaphore)
- Sémaphore binaire
- Mutex

Avantages

- Assurent l'exclusion mutuelle avec facilité
- Évitent l'interblocage
- Sont-ils équitables ?

Inconvénients

- connaissance des compétiteurs
- opérations toujours difficiles à utiliser
- on peut oublier de mettre des éléments en section critique
- les mêmes primitives assurent l'exclusion mutuelle et la synchronisation conditionnelle
- les opérations P et V ne donnent aucune idée sur la ressource visée

Autres problèmes potentiels

- Libération accidentelle d'un sémaphore
- Interblocage récursif
- Inversion de priorité
- Interblocage par la mort d'un processus

Avantage ou inconvénient

Temps pour changement de contexte vs attente active ?

Chapitre 3 - Synchronisation

Tampon fini

- On possède un tampon contenant n éléments chacun contenant un item
- Pour la synchronisation on utilise 3 sémaphores
  - mutex (exclusion mutuelle)
  - plein et vide (synchronisation conditionnelle)

Sémaphores - tampon fini

type item : ...

var plein, vide, mutex : **semaphore** ;
tampon : **array**[0..n-1] **of** item ;
nextp, nextc : item ;

```
begin
    plein := n ; vide := 0 ; mutex := 1 ;

    parbegin
        producteur :        repeat
                                produire un "item" dans nextp ;
                                P(plein) ; P(mutex) ;
                                dépose nextp dans tampon ;
                                V(mutex) ; V(vide) ;
                            forever ;

        consommateur :      repeat
                                P(vide) ; P(mutex) ;
                                lire nextc de tampon ;
                                V(mutex) ; V(plein) ;
                                traite nextc ;
                            forever ;

    parend
end ;
```

Sémaphores - Système "batch"

```
Program OPSYS ;
var     in_mutex, out_mutex : semaphore initial (1,1) ;
        nun_in, num_out : semaphore initial (0,0) ;
        free_in, free_out : semaphore initial (n,n) ;
        tampon_in : array[0..n-1] of entree ;
        tampon_out : array[0..n-1] of sortie ;

process lecteur ;
        var ligne : entree ;
        loop
                lecture ligne ;
                P(free_in) ; P(in_mutex) ;
                dépose ligne dans tampon_in ;
                V(in_mutex) ; V(num_in) ;
        end ;
end process ;
```

```
Process traitement ;
        var    ligne : entree ;    resultat : sortie ;
        loop
                P(num_in) ; P(in_mutex) ;
                lecture ligne de tampon_in ;
                V(in_mutex) ; V(free_in) ;
                traitement de ligne et génération de resultat ;
                P(free_out) ; P(out_mutex) ;
                dépose resultat dans tampon_out ;
                V(out_mutex) ; V(num_out) ;
        end ;
end process ;

Process imprimante ;
        var resultat : sortie ;
        loop
                P(num_out) ; P(out_mutex) ;
                lecture resultat de tampon_out ;
                V(out_mutex) ; V(free_out) ;
                impression de resultat ;
        end ;
end process ;
```

Lecteurs/écrivains

- Un objet peut être partagé par plusieurs processus
- Certains peuvent faire des lectures et d'autres des mises à jour
- Cette distinction est importante :
  - lectures simultanées seulement $\rightarrow$ pas de problème.
  - écritures simultanées $\rightarrow$ risque d'incohérences
  - lectures et écriture simultanées $\rightarrow$ risque d'incohérences

Lecteurs/écrivains

Exemple

- Compte en banque A contient $500
- Transaction B ajoute $10 sur A
- Transaction C ajout $1000 sur A
- Séquence :
  1. B lit A ($500)
  2. C lit A ($500)
  3. C ajoute $1000
  4. C écrit A ($1500)
  5. B ajoute $10
  6. B écrit A ($510) $\Rightarrow$ A contient à la fin $510

Lecteurs/écrivains

Solutions :

- Permet plusieurs lecteurs simultanées
- Accès exclusif aux écrivains
- Variations :
  - on ne fait attendre aucun lecteur
  - on ne fait attendre aucun écrivain
  - autres ? ? ?

Exemple de solution

- 2 sémaphores : mutex (1) et wrt (1)
- 1 entier (nblecteur)

Exemple de solution

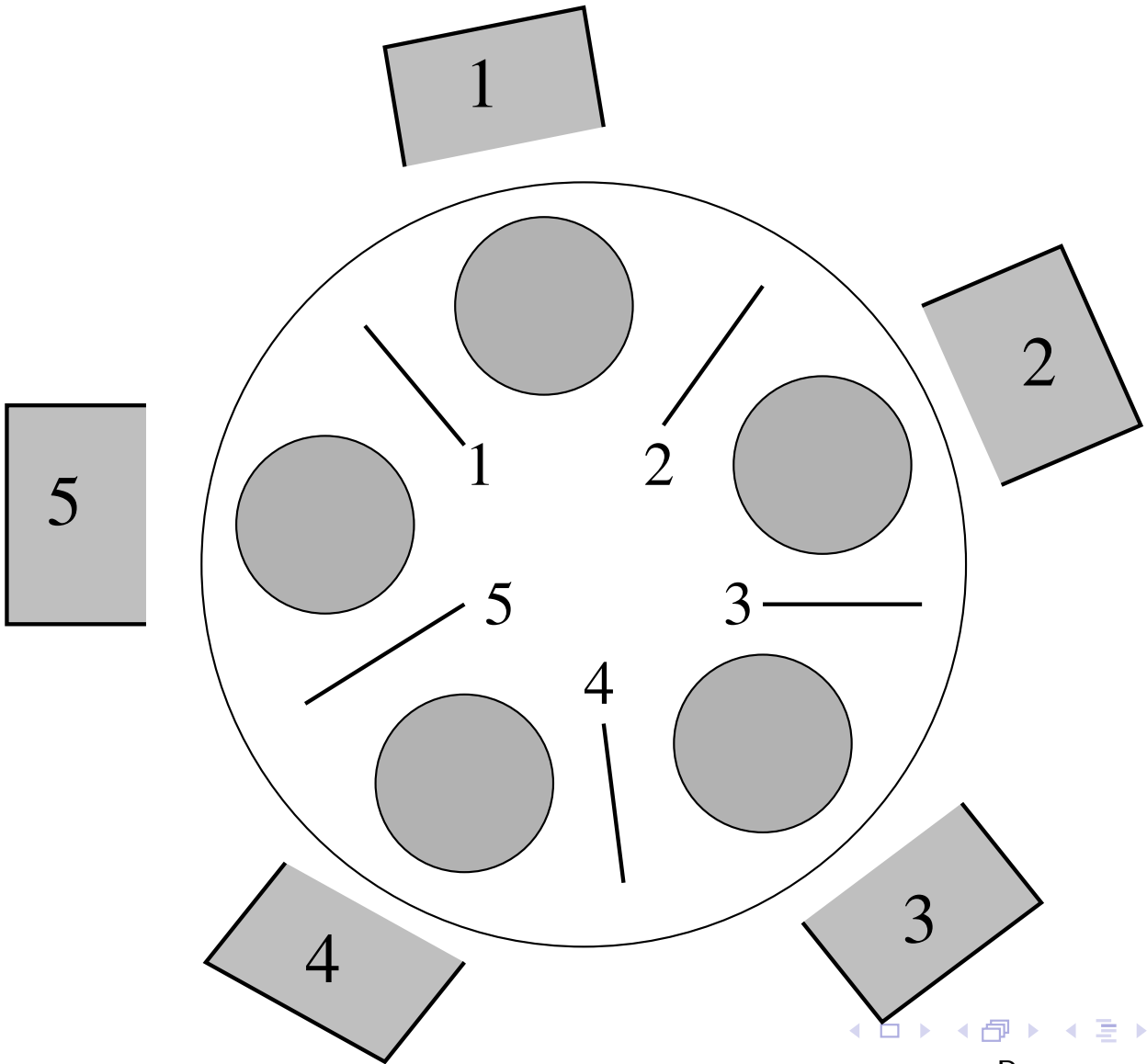
```
Lecteur : P(mutex)
             nblecteur++
             if (nblecteur=1) P(wrt)
          V(mutex)
         .... lecture.....
          P(mutex)
             nblecteur--
             if (nblecteur=0) V(wrt)
          V(mutex)

Écrivain: P(wrt)
         .... écriture .....
          V(wrt)
```

Philosophes

- Introduit par Dijkstra (1965)
- 5 philosophes passent leur vie à penser et manger
- Ils partagent une table circulaire, 5 chaises, 5 plats de riz et 5 baguettes
- Pour manger il doit prendre 2 baguettes (les plus rapprochées)
- Si une des baguettes n'est pas disponible, il attend

Philosophes



```
Var baguette : array[0..4] of semaphore;
Procedure phil(i:integer)
begin
   repeat
      P(baguette[i])
      P(baguette[i+1mod5])
     ...mange ...
      V(baguette[i]
      V(baguette[i+1mod5])
     ... pense ...
  forever
end
begin
   baguette[0..4] :=1
   cobegin
      phil(0); phil(1); phil(2); phil(3); phil(4)
   coend
end
```

Chapitre 3 - Synchronisation

Conclusion

- Tout semble beau... En théorie ! !
- Les algorithmes supposent une certaine cohérence de la mémoire
- Cette cohérence est absente sur la plupart des ordinateurs modernes ! ! !
- Sans compter que les compilateurs ré-ordonnent certains énoncés ! ! !
- Les algorithmes de Dekker et de Peterson ne fonctionnent donc plus ! ! ! !

Conclusion

- Exemples : Dekker et Peterson
- Disponibles sur le site Web...