Processus concurrents et parallélisme

Chapitre 4 - Programmation parallèle de haut niveau

Gabriel Girard

1er février 2023

Chapitre 4 - Programmation parallèle de haut niveau

Les langages évolués

- Aujourd'hui, les systèmes sont de plus en plus écrits dans des langages évolués
- Avantages ? ? ?
- Inconvénients ? ? ?

Modèles (abstraits) utilisé dans les langages

- Modèle basé sur les réseaux de Pétri
- Modèle par flots de données
- Modèle par acteurs
- Modèle par objets

Modèles (abstraits) utilisé dans les langages

- Processus ou fils
- Procédure ou fonction
- Type abstrait de données

Que sont les régions critiques ?

- Introduites pas Brinch Hansen et Hoare
- Notation structurée pour spécifier la synchronisation (exclusion mutuelle)

Syntaxe 1

var V : **shared** T ;
...
region V **do** S ;

Syntaxe 2

var $v_1, v_2, ..., v_n$: T ;
ressource R : $v_1, v_2, ..., v_n$;
...
region R **do** S ;

Pendant l'exécution de «S» aucun autre processus ne peut accéder la ressource R.

Exemple

region R **do** S1 ;
region R **do** S2 ;

Équivalent à « `S1;S2` » ou « `S2;S1` » .

Implantation

Nécessite un sémaphore **mutex = 1**

region R **do** S ;

Génère

P(mutex) ;
S ;
V(mutex) ;

Problème potentiel ! !

```
ressource X:a; Y:b;
...
parbegin
   P1: region X do region Y do S0;
   P2: region Y do region X do S1;
parend
```

Présentation

- Les régions critiques ne permettent pas d'exprimer la synchronisation conditionnelle
- Pour y arriver on ajoute un énoncé **when**

Syntaxe

var $v_1, v_2, ..., v_n$: T ;
ressource R : $v_1, v_2, ..., v_n$;
...
region R **when** B **do** S ;

Caractéristiques

- L'évaluation de B et l'exécution de S sont ininterruptibles.
- Si B est faux, on bloque le pcs et on relâche l'exclusion mutuelle
- Le mécanisme de délai doit être équitable

Système Batch avec RCC

```
Program OPSYS ;

type    tampon(T) :   record
                        item : array[0..n-1] of T ;
                        tete, queue : 0.. n-1 initial (0,0) ;
                        dimension : 0..n initial (0) ;
                      end ;
var     tampon_in : tampon(entree) ;
        tampon_out : tampon(sortie) ;
resource ib : tampon_in ;
         ob : tampon_out ;
```

Système Batch avec RCC

```
process lecteur ;
    var ligne : entree ;
    loop
        lecture ligne ;
        region ib when tampon_in.dimension < n do ;
            tampon_in.item[tampon_in.queue] := ligne ;
            tampon_in.dimension := tampon_in.dimension + 1 ;
            tampon_in.queue := (tampon_in.queue + 1) mod n ;
        end ;
    end ;
end process ;
```

Système Batch avec RCC

```
Process traitement ;
    var ligne : entree ;
        resultat : sortie ;
    loop
        region ib when tampon_in.dimension > 0 do ;
            ligne := tampon_in.item[tampon_in.tete] ;
            tampon_in.dimension := tampon_in.dimension - 1 ;
            tampon_in.tete := (tampon_in.tete + 1) mod n ;
        end ;
        ..... traitement de ligne et génération de resultat ;
        region ob when tampon_out.dimension < n do ;
            tampon_out.item[tampon_out.queue] := resultat ;
            tampon_out.dimension := tampon_out.dimension + 1 ;
            tampon_out.queue := (tampon_out.queue + 1) mod n ;
        end ;
    end ;
end process ;
```

Système Batch avec RCC

```
Process imprimante ;
    var resultat : sortie ;
    loop
        region ob when tampon_out.dimension > 0 do ;
            resultat := tampon_out.item[tampon_out.tete] ;
            tampon_out.dimension := tampon_out.dimension - 1 ;
            tampon_out.tete := (tampon_out.tete + 1) mod n ;
        end ;

        impression de resultat ;
    end ;
end process ;
```

Implantation

- On associe à chaque ressource R :

```
var R_mutex, R_wait : sem;
    R_count, R_temp : integer;
```

- R_mutex → exclusion mutuelle
- R_wait → attente si B est faux
- R_count → nombre de pcs en attente sur R_wait

Implantation

- Un pcs quittant la section critique peut changer la condition booléenne
- Tous les processus en attente sur la condition booléenne doivent pouvoir la re-tester
- R_temp $\rightarrow$ compte le nombre de pcs qui ont re-testé la condition

Code généré

region R **when** B **do** S ;

Génère

```
var     R_mutex, R_wait : semaphore ;
        R_count, R_temp : integer ;

R_count := R_temp := R_wait :=0 ;
R_mutex := 1 ;
```

Code généré

```
P(R_mutex);
if not B then begin
                    R_count := R_count + 1;
                    V(R_mutex);
                    P(R_wait);
                    while not B do
                        begin
                                R_temp := R_temp + 1;
                                if R_temp < R_count
                                        then V(R_wait);
                                        else V(R_mutex);
                                P(R_wait);
                        end;
                    R_count := R_count -1;
                end;
Section critique
```

Code généré

Section critique

```
if R_count > 0 then
                    begin
                        R_temp := 0;
                        V(R_wait);
                    end;
else V(R_mutex);
```

Amélioration

- Ce type de RCC permet d'attendre seulement au début de la région
- Amélioration (Brinch Hansen)

```
region R do
begin
    S1;
    await(B);
    S2;
end
```

Exemple : lecteurs/écrivains

```
type reader-writer = class
        var    v :    shared record
                                  nreaders, nwriters : integer ;
                                  busy : boolean ;
                      end ;
        procedure entry open-read ;
                region v do ;
                                  await(nwriters = 0) ;
                                  nreaders := nreaders + 1 ;
                end ;
        end procedure ;

        procedure entry close-read ;
                region v do ;
                                  nreaders := nreaders - 1 ;
                end ;
        end procedure ;
```

Lecteurs/écrivains

```
procedure entry open-write ;
        region v do ;
                        nwriters := nwriters + 1 ;
                        await((not busy) and (nreaders = 0)) ;
                        busy := vrai ;
        end ;
end procedure ;

procedure entry close-write ;
        region v do ;
                        nwriters := nwriters - 1 ;
                        busy := faux ;
        end ;
end procedure ;
```

Lecteurs/écrivains

- Soit `rw` un objet de type `reader_writer`
- Un lecteur doit faire :

```
rw.open_read()
  lecture
rw.close_read()
```

- Un écrivain doit faire :

```
rw.open_write()
  lecture
rw.close_write()
```

Lecteurs/écrivains

- Problème : un utilisateur peut oublier d'obéir aux règles
- Similaire à l'utilisation des sémaphores
- Solution ...

Lecteurs/écrivains

```
procedure entry read(.....) ;
        begin
            open-read ;
            ...
            lecture du fichier
            ...
            close-read
        end

procedure entry write(.....) ;
        begin
            open-write ;
            ...
            écriture dans le fichier
            ...
            close-write ;
        end
```

Évaluation

- Coût ? ?
- Modularité ? ?
- Puissance d'expression ? ?
- Facilité d'utilisation ? ?

Présentation

- Introduits par Hoare et Brinch Hansen
- Un moniteur est formé en encapsulant la définition de la ressource avec ses opérations
- Similaire à une classe

Syntaxe

Nom-moniteur = **monitor**
var ... : *déclaration des variables permanentes* ;

procedure op1(paramètres) ;
var ... : *déclaration des variables locales à* **op1** ;
begin ;
code pour implanter **op1** ;
end ;
...
procedure opN(paramètres) ;
var ... : *déclaration des variables locales à* **opN** ;
begin ;
code pour implanter **opN** ;
end ;

begin
code pour initialiser les variables permanentes ;
end ;

Utilisation

- Appel : `call` *nom_moniteur*`.opi(paramètres)`
- Par définition du moniteur, toutes les procédures s'exécutent en exclusion mutuelle
- Plusieurs solutions existent pour résoudre le problème de la synchronisation conditionnelle.

Variables conditions

- Proposée par Hoare
- On définit des variables conditions dans le moniteur

```
var cond : condition
```

- Reconnaissent deux opérations : `wait` et `signal`

```
cond.wait
cond.signal
```

Variables conditions

- `cond.wait`
  Bloque le processus appelant et relâche l'exclusion mutuelle sur le moniteur
- `cond.signal`
  - Si aucun processus bloqué, l'appelant continu et le signal est perdu.
  - S'il y a des processus en attente, le signal réactive un processus et bloque l'émetteur du signal pour donner le contrôle au processus réactivé

Exemple 1 : tampons à N éléments

```
type tampon(T) = monitor

    var     item : array[0..n-1] of T ;
            tete, queue : 0.. n-1 ;
            dimension : 0..n ;
            non-plein, non-vide : condition ;

    procedure  deposer(p : T) ;
            begin ;
                    if dimension = n then non-plein.wait ;
                    item[queue] := p ;
                    dimension := dimension + 1 ;
                    queue := (queue + 1) mod n ;
                    non-vide.signal ;
            end ;
```

Exemple 1 : tampons à N éléments

```
procedure  retirer(var p : T) ;
        begin ;
                if dimension = 0 then non-vide.wait ;
                p := item[tete] ;
                dimension := dimension - 1 ;
                tete := (tete + 1) mod n ;
                non-plein.signal ;
        end ;

begin
        dimension := 0 ; tete := 0 ; queue := 0 ;
end ;
```

Exemple 1 : tampons à N éléments

La solution suivante fonctionne-t-elle ?

```
procedure retirer(var p : T) ;
    begin ;
        non-vide.wait ;
        p := item[tete] ;
        dimension := dimension - 1 ;
        tete := (tete + 1) mod n ;
        non-plein.signal ;
    end ;
```

Exemple 2 : système batch

```
Program OPSYS ;

type        tampon(T) = { ...moniteur précédant} ;
var         tampon_in : tampon(entree) ;
            tampon_out : tampon(sortie) ;


process lecteur ;
        var ligne : entree ;
        loop
                lecture ligne ;
                call tampon_in.deposer(ligne) ;
        end ;
end process ;
```

Exemple 2 : système batch

```
Process traitement ;
        var     ligne : entree ;
                resultat : sortie ;
        loop
                call tampon_in.retirer(ligne) ;
                traitement de ligne et génération de resultat ;
                call tampon_out.deposer(resultat) ;
        end ;
end process ;

Process imprimante ;
        var resultat : sortie ;
        loop
                call tampon_out.retirer(resultat) ;
                impression de resultat ;
        end ;
end process ;
```

Exemple 3 : sémaphores

```
type semaphore = monitor
        var   occupe : boolean ;
              non-occupe : condition ;

        procedure  P() ;
              begin ;
                    if busy then non-occupe.wait ;
                    occupe := vrai ;
              end ;
        procedure  V() ;
              begin ;
                    occupe := faux ;
                    non-occupe.signal ;
              end ;

        begin
              occupe := faux ;
        end ;
```

Exemple 4 : philosophes

```
type philosophes = monitor

    var     etat : array[0..4] of (pense, afaim, mange) ;
            attente : array[0..4] of condition ;

    procedure  prendre(i : 0..4) ;
            begin ;
                    etat[i] := afaim ;
                    teste(i) ;
                    if etat[i] ≠ mange then attente[i].wait ;
            end ;

    procedure  deposer(i : 0..4) ;
            begin ;
                    etat[i] := pense ;
                    teste(i-1 mod 5) ;
                    teste(i+1 mod 5) ;
            end ;
```

Exemple 4 : philosophes

```
procedure teste(k : 0..4);
    begin;
        if (etat[k+1 mod 5] ≠ mange) and
           (etat[k] = afaim) and (etat[k-1 mod 5] ≠ mange)
           then begin
                etat[k] := mange;
                attente[k].signal;
           end;
    end;
begin
    for i :=0 to 4 do etat[i] := pense;
end;
```

Exemple 4 : philosophes

```
var dp:philosophes;


dp.prendre(i);

.....mange

dp.deposer(i);
```

Exemple 4 : philosophes

Qu'est-ce qui ne fonctionne pas avec cette solution ?

Variables de type queue

- Similaires aux variables conditions mais moins coûteuses

```
var    ress: queue;
```

- `ress.wait`
  Un seul processus peut se bloquer sur une variable de type queue. L'ordonnancement se fait par un tableau de variables de type queue.
- `ress.continue`
  Réactive le processus bloqué et sort l'appelant du moniteur.

Attente conditionnelle

- Introduction par Hoare d'un énoncé d'attente conditionnelle
  `wait(B);`
- B est une expression booléenne
- Aucun signal requis
- Problème ? ? ?

Attente conditionnelle

```
type A = monitor

        var     non-occupe : boolean ;

        procedure  reserve() ;
                begin ;
                        wait(non-occupe) ;
                        non-occupe := faux ;
                end ;

        procedure  libere() ;
                begin ;
                        non-occupe := vrai ;
                end ;

        begin
                non-occupe := vrai ;
        end ;
```

Variables conditions et wait/notify

- On change le signal pour un notify (= `signal & continue`)
- `cond.notify;`
  - débloque le processus signalé
  - l'émetteur du « notify » poursuit son exécution
  - le processus signalé est ordonnancé pour l'entrée dans le moniteur

  Permet donc à un processus bloqué sur une variable condition de poursuivre son exécution à un moment donné dans le futur. Ne bloque pas l'appelant.

Implications

- Le processus signalé est en concurrence avec les autres pour entrer à nouveau dans le moniteur
- Il n'y a aucune garantie que la condition qui a permis de le débloquer soit toujours vraie lors de sa prochaine entrée dans le moniteur
- La condition doit être testée de nouveau

```
while (...) { ... cond.wait()...}
```

- Cela a l'avantage de mieux respecter les priorités si ce concept est implanté dans la file d'entrée

Exemple 1 : tampons à N éléments

```
type tampon(T) = monitor

    var     item : array[0..n-1] of T ;
            tete, queue : 0.. n-1 ;
            dimension : 0..n ;
            non-plein, non-vide : condition ;

    procedure  deposer(p : T) ;
            begin ;
                    while dimension = n then non-plein.wait ;
                    item[queue] := p ;
                    dimension := dimension + 1 ;
                    queue := (queue + 1) mod n ;
                    non-vide.notify ;
            end ;
```

Exemple 1 : tampons à N éléments

```
procedure  retirer(var p : T) ;
        begin ;
                while dimension = 0 then non-vide.wait ;
                p := item[tete] ;
                dimension := dimension - 1 ;
                tete := (tete + 1) mod n ;
                non-plein.notify ;
        end ;

begin
        dimension := 0 ; tete := 0 ; queue := 0 ;
end ;
```

Résumé sur les différentes variables conditions

- Signal & wait
- Signal & continue
- Signal & exit
- Signal & urgent wait
- SignalAll

Moniteur avec variables conditions (wait/signal)

- Pour chaque moniteur, on a un sémaphore `mutex = 1`
- Pour les variables conditions :
  - Un sémaphore `sem (= 0)` pour bloquer les émetteurs de `wait`
  - Une variable `cpt` pour compter le nombre de processus en attente sur `sem`
  - Un sémaphore `next (= 0)` sert à bloquer les émetteurs de `signal`.
  - Une variable `next_count` compte le nombre de processus suspendu sur `next`

Code généré

Chaque procédure est encadré par :

```
P(mutex)
    ...
    code pour OPi
    ...
if (next_count > 0)
    then V(next)
    else V(mutex)
```

Code généré

L'opération `wait` génère le code suivant

```
cpt = cpt+1
if (next_count > 0)
    then V(next)
    else V(mutex)
P(sem)
cpt = cpt - 1
```

Code généré

L'opération `signal` génère le code suivant

```
if (cpt > 0)
    then begin
            next_count = next_count + 1
            V(sem)
            P(next)
            next_count = next_count - 1
         end
```

Moniteur avec variables conditions (wait/notify)

- Pour chaque moniteur, on a un sémaphore `mutex = 1`
- Pour les variables conditions on a un sémaphore `sem (= 0)` pour bloquer les émetteurs de `wait`

Code généré

Chaque procédure est encadré par :

```
P(mutex)
 ...
 code pour OPi
 ...
V(mutex)
```

Code généré

L'opération `wait` génère le code suivant

```
V(mutex)
P(sem)
P(mutex)
```

Code généré

L'opération `signal` génère le code suivant

```
V(sem)
```

Évaluation

- Moniteurs imbriqués ! !
- Une partie du code de synchronisation demeure visible ! !
- Ordre de sortie sur les variables conditions ! !

Ordre de sortie

- Qu'arrive-t-il si on veut des priorités ?
- Pour régler le problème Hoare a introduit un paramètre `p` :

```
cond.wait(p);
```

- Lors du signal, le processus avec le plus petit `p` est choisi.

Exemple

```
type SJF = monitor
        var     libre : boolean ;
                tour : condition ;

        procedure  demander(time : integer) ;
                begin ;
                        if not  libre then tour.wait(time) ;
                        libre := faux ;
                end ;
        procedure  libere() ;
                begin ;
                        libre := vrai ;
                        tour.signal() ;
                end ;

        begin
                libre := vrai ;
        end ;
```

Moniteurs étendus

- Le problème avec les moniteurs est que toutes les procédures s'exécutent en exclusion mutuelle
- L'exclusion mutuelle peut être longue dans un moniteur (lecteur/écrivain)
- Limite aussi inutilement la concurrence (lectures simultanées doivent se faire à l'extérieur du moniteur)
- Solution : les moniteurs étendus

Moniteurs étendus

- Ce type de moniteur permet de définir
  - des procédures gardées (qui s'exécutent en exclusion mutuelle)
  - des procédures normales qui peuvent être appelées seulement par les processus autorisés
- Les autorisations sont distribuées dans les procédures gardées

Exemple : lecteurs/écrivains

```
crowd monitor readwrite ;

export startread, endread, read, startwrite, endwrite, write ;

var       readers : crowd read ;
          writers : crowd read, write ;

guard procedure  startread ;
          begin ;
                ... (bloque l'appelant jusqu'à autorisation de lire) ;
                enter readers ;
          end ;

guard procedure  endread ;
          begin ;
                leave readers ;
                ... (libération de la ressource si nécessaire) ;
          end ;
```

Lecteurs/écrivains

```
guard procedure startwrite;
        begin;
                ... (bloque l'appelant jusqu'à autorisation d'écrire);
                enter writers;
        end;

guard procedure endwrite;
        begin;
                leave writers;
                ... (libération de la ressource);
        end;
```

Lecteurs/écrivains

```
procedure read;
        begin;
                ... (lecture de la ressource partagée);
        end;

procedure write;
        begin;
                ... (écriture de la ressource partagée);
        end;

end readwrite;
```

Expressions de chemin

- On veut spécifier toute la synchronisation à un seul endroit
- Cette spécification doit être indépendante des opérations
- Les expressions de chemin utilisent le concept de « classe »
- Les expressions de chemin dans l'entête de la classe spécifient tous les besoins en synchronisation

Syntaxe

- Format général : `PATH` *liste* `END;`
- La liste contient les noms des opérations et les opérateurs de chemin
- Les opérateurs de chemin sont :
  1. → « , » : concurrence
  2. → « ; » : séquence
  3. → « n :(liste) » : n exécutions concurrentes de la liste
  4. → « [liste] » : nombre illimité d'exécutions concurrentes de la liste

Exemple 1 : tampons à N éléments

- PATH deposer, retirer END ;
- PATH [deposer],[retirer] END ;
- PATH [deposer,retirer] END ;
- PATH deposer ; retirer END ;
- PATH 1 :(deposer ;retirer) END ;
- Tampon contenant N éléments et chaque opération en exclusion mutuelle ?

Exemple 2 : tampons à N éléments

```
type tampon(T) = module
        PATH n :((1 :deposer) ;1 :(retirer)) END

        var     item : array[0..n-1] of T ;
                tete, queue : 0.. n-1 ;
        procedure  deposer(p : T) ;
                begin ;
                        item[queue] := p ;
                        queue := (queue + 1) mod n ;
                end ;
        procedure  retirer(var p : T) ;
                begin ;
                        p := item[tete] ;
                        tete := (tete + 1) mod n ;
                end ;

        begin
                tete := 0 ; queue := 0 ;
        end ;
```

Exemple 2 : sémaphores

PATH [V ;P] **END**

```
procedure V()
    begin
    end.
```

```
procedure P()
    begin
    end.
```

Exemple 3 : lecteurs/écrivains

?????

Exemple 3 : lecteurs/écrivains

PATH 1 :([read],write) **END**

Exemple 3 : lecteurs/écrivains

PATH 1 :(write-attempt) **END**
PATH 1 :([request-read],request-write) **END**
PATH 1 :([read],(open-write ;write)) **END**

write-attempt = **begin** request-write **end** ;
request-write = **begin** open-write **end** ;
request-read = **begin** read **end** ;

READ = **begin** request-read **end** ;
WRITE = **begin** write-attempt ; write **end** ;

Exemple 3 : lecteurs/écrivains

PATH 1 :(read-attempt) **END**
PATH 1 :(request-read,[request-write]) **END**
PATH 1 :([open-read ;read],write) **END**

read-attempt = **begin** request-read **end** ;
request-read = **begin** open-read **end** ;
request-write = **begin** write **end** ;

READ = **begin** read-attempt, read **end** ;
WRITE = **begin** request-write **end** ;

Exemple 3 : lecteurs/écrivains

PATH 1 :(request-read,request-write) **END**
PATH 1 :([open-read ;read], write) **END**

request-read = **begin** open-read **end** ;
request-write = **begin** write **end** ;

READ = **begin** request-read, read **end** ;
WRITE = **begin** request-write **end** ;

Implantation

- Les expressions de chemin se traduisent directement en une séquence de P et V que l'on ajoute en prologue et en épilogue aux opérations.
- La traduction se fait en plusieurs étapes bien définies

Algorithme de génération

- Soit l'expression de chemin

  `PATH <liste> END;`

- Initialisation

  `PATH <liste> END` `devient` `« L M R »`

  où L = null, M= liste et R=null

- À la fin on veut «L op_i R» pour chaque opération i

Algorithme de génération

- Traduction : on examine la « liste »
  - si « L $<E_1>$,$<E_2>$ R »
    $\rightarrow$ « L $<E_1>$ R » et « L $<E_2>$ R »
  - si « L $<E_1>$;$<E_2>$ R »
    $\rightarrow$ « L $<E_1>$ V(s1) » et « P(s1) $<E_2>$ R » (s1 = 0)
  - si « L n :(liste) R »
    $\rightarrow$ « P(s2) L <liste> R V(s2) » (s2=n)

Algorithme de génération

- si « L [liste] R »
  → « PP(c,s,L) <liste> VV(c,s,R) »
- si « L (<liste>) R »
  → « L <liste> R »
- si « L <opération> R »
  → begin
  L ;
  <opération> ;
  R ;
  end.

Algorithme de génération

- PP(c,s,L) (semaphore s=1, int c = 0)

```
P(s);
c = c + 1;
if (c==1) then L;
V(s);
```

- VV(c,s,L) (semaphore s=1, int c = 0)

```
P(s);
c = c - 1;
if (c==1) then R;
V(s);
```

Exemples de génération

- PATH deposer, retirer END
- PATH deposer ; retirer END
- PATH 1 :(deposer ;retirer) END
- PATH 1 :([lecture] ;écriture) END
- PATH n :(1 :(lecture) ;1 :(écriture)) END

Expressions invariantes

- Ressemble aux expressions de chemin
- On associe à chaque procédure (méthode) une liste d'attente de processus
- Une expression invariante est associée à chaque liste (elle indique qui peut entrer ou non)
- Aucune expression invariante $\rightarrow$ aucune restriction

Expressions invariantes

- Les exp. inv. utilisent 5 compteurs prédéfinis qui enregistrent les événements pour une procédure « proc » :
  1. requestCount(proc) : nombre de demandes
  2. startCount(proc) : nombre de démarrages
  3. finishCount(proc) : nombre de terminaisons
  4. currentCount(proc) : startCount(proc) - finishCount(proc)
  5. waitCount(proc) : requestCount(proc) - startCount(proc)

Syntaxe

- Un expression invariante a la forme :

  expression comparaison constante

- *expression* est composée de sommes ou de soustractions
- *comparaison* est composée de $<, >, \geq, \leq, =, \neq$
- Exemples : « waitCount(proc1) - requestCount(proc2) > 4 »

Exemple 1 : tampon à N éléments

```
type tampon(T) = module

        export : deposer, retirer ;

        var    item : array[0..n-1] of T ;
               tete, queue : 0.. n-1 ;

        invariant deposer
               startcount(deposer) - finishCount(retirer) < n ;
               currentCount(deposer) = 0 ;
        invariant retirer
               startcount(retirer) - finishCount(deposer) < 0 ;
               currentCount(retirer) = 0 ;
```

Exemple 1 : tampon à N éléments

```
procedure deposer(p : T);
        begin;
                item[queue] := p;
                queue := (queue + 1) mod n;
        end;

procedure retirer(var p : T);
        begin;
                p := item[tete];
                tete := (tete + 1) mod n;
        end;

begin
        tete := 0; queue := 0;
end;
```

Exemple 2 : leccteurs/écrivains

invariant read
currentCount(write) = 0 ;
invariant write
currentCount(read) + currentCount(write) = 0 ;

invariant read
waitCount(write) + currentCount(write) = 0 ;
invariant write
currentCount(read) + currentCount(write) = 0 ;

Compteurs d'événements et séquenceur

- Compteur d'événements (E)
  avance(E), lire(E), attendre(E)
- Séquenceur (S)
  ticket(S) (incrémente S après la lecture)
- Serializers

Verrous et variations des sémaphores

- Lock/unlock (Dijkstra 68 - réservé aux noyaux)
  Lock(E)
  Unlock(E)
- Sémaphore (chunk)
  P(S,t)
  V(S,t)
- Sémaphore et disjonction
  $P_{or}(S_1, S_2, ...S_n)$
  $V(S_i)$

Variations des sémaphores

- Sémaphore et conjonction

  $P_{et}(S_1, S_2, ...S_n)$
  $V(S_1, S_2, ...S_n)$

- Sémaphore général

  $P_{et}(S_1, t_1; S_2, t_2; ...; S_n, t_n)$
  $V(S_1, t_1; S_2, t_2; ...; S_n, t_n)$

Évaluation des méthodes de synchronisation

- L'évaluation se base sur trois caractéristiques
  - modularité
  - puissance d'expression
  - facilité d'utilisation

Évaluation

- La modularité est possible grâce à l'abstraction de données et la séparation du contrôle de la concurrence de l'accès à la ressource
- La puissance d'expression est l'habileté à formuler l'exclusion mutuelle, la synchronisation conditionnelle et les besoins de piorité
- La facilité d'utilisation est un critère subjectif....

Conclusion

- Autres techniques dérivées
  - Predicate path expressions
  - Synchronization counters
  - Iteration expressions (Java)
  - ...