Synchronisation Distribuée
Exclusion mutuelle avec horloge

Laurent PHILIPPE

Master 2 Informatique
Ingénierie des Systèmes et Logiciels

2021/2022

Sommaire

Synchronisation distribuée avec horloges logiques

Hypothèses

- Processus ne communiquent que par échanges de messages
- Chaque processus connaît l'ensemble des processus du système
- Chaque processus a ses propres variables, pas de partage de variables
- On ne considère pas les cas de pertes de messages ni les pannes de machines
- Les échanges de messages sont FIFO : les messages ne se doublent pas.

Synchronisation distribuée avec horloges logiques

Principe

- Diffuser la demande
- Il faut l'accord de tous les autres
- Une fois le consensus obtenu on accède à la SC
- Difficulté : demandes concurrentes

Algorithme de Lamport

- Utilise les horloges logiques pour ordonner les évènements
- P_i incrémente son horloge entre deux évènements locaux ou distants
- À la réception d'un message de P_J par P_i d'horloge h_j, l'horloge h devient $max(h, h_j) + 1$

Algorithme de Lamport

Variables pour chaque processus P_i

- Horloge locale h
- Tableau des horloges des autres processus, initialisé à 0
- Chaque processus maintient un tableau de requêtes : état de demande des autres processus (ACK, REL, REQ)
- Initialement, chaque processus connaît tous les autres participants : liste des voisins

Algorithme de Lamport

Principe de l'algorithme(1)

- Quand P_i veut entrer en Section Critique, il incrémente son horloge, envoie le message $REQ(h_{sc} : P_i)$ de demande de SC à tous les autres processus et dépose ce message dans tableau des requêtes
- Quand un processus P_i reçoit le message $REQ(h_j : P_j)$ de demande de SC, il synchronise son horloge, met le message dans son tableau de requêtes et envoie un message ACK daté de bonne réception à P_j

Algorithme de Lamport

Principe de l'algorithme (2)

- P_i accède à la SC quand les deux conditions suivantes sont réalisées :
  - Il existe un message $REQ(h_{sc} : P_i)$ dans son tableau de requêtes qui est ordonné devant toute autre requête selon la relation de précédence
  - Le processus P_i a reçu un message de tous les autres processus ayant une date supérieure à h_{sc}

  Ces 2 conditions sont testées localement par P_i
- Quand P_i quitte la SC, il enlève sa requête $REQ(h_{sc} : P_i)$ de demande de SC de son tableau de requêtes, incrémente son horloge et envoie un message daté $REL(h : P_i)$ pour signaler qu'il quitte la SC à tous les processus
- Quand P_i reçoit le message $REL(h_j : P_j)$ (quitte SC), il synchronise son horloge et enlève le message $REQ(h_j : P_j)$ (demande de SC) de son tableau de requêtes

Algorithme de Lamport

Variables utilisées par chaque processus P_i

h : entier indiquant l'horloge locale. Init à 0
$F\_H[]$: tableau contenant les horloges des différents processus. La taille du tableau correspond au nombre de participants.
$F\_M[]$: tableau des messages, $\in \{REQ, ACK, REL\}$
V : ensemble de tous les voisins de P_i

Messages utilisés

$REQ(H)$: demande d'entrée en SC
$ACK(H)$: acquittement d'une demande d'entrée en SC
$REL(H)$: sortie de SC

Algorithme de Lamport

Règle 1 : P_i demande la Section Critique

Faire

$h \leftarrow h + 1$

Pour $j \in V$: P_i **envoie** $REQ(h)$ à P_j

$F\_H[i] \leftarrow h$

$F\_M[i] \leftarrow REQ$

attendre

$\forall j \in V((F\_H[i] < F\_H[j]) \vee ((F\_H[i] = F\_H[j]) \wedge i < j))$

$< SectionCritique >$

Fait

Algorithme de Lamport

Règle 2 : P_i **reçoit** $REQ(h_j)$ de P_j

Faire

$h \leftarrow \max(h, h_j) + 1$

$F\_H[j] \leftarrow h_j$

$F\_M[j] \leftarrow REQ$

P_i **envoie** $ACK(h)$ à P_j

Fait

Algorithme de Lamport

Règle 3 : P_i **reçoit** $ACK(h_j)$ de P_j

Faire

$h \leftarrow \max(h, h_j) + 1$

Si $F\_M[j] \neq REQ$ **Alors**

$F\_H[j] \leftarrow h_j$

$F\_M[j] \leftarrow ACK$

Fsi

Fait

Algorithme de Lamport

Règle 4 : P_i sort de la Section Critique

Faire

$h \leftarrow h + 1$

Pour $j \in V$: P_i **envoie** $REL(h)$ à P_j

$F\_H[i] \leftarrow h$

$F\_M[i] \leftarrow REL$

Fait

Règle 5 : P_i **reçoit** $REL(h_j)$ de P_j

Faire

$h \leftarrow \max(h, h_j) + 1$

$F\_H[j] \leftarrow h_j$

$F\_M[j] \leftarrow REL$

Fait

Déroulement de l'algorithme de Lamport

Exécution avec 3 processus : P_0, P_1 et P_2

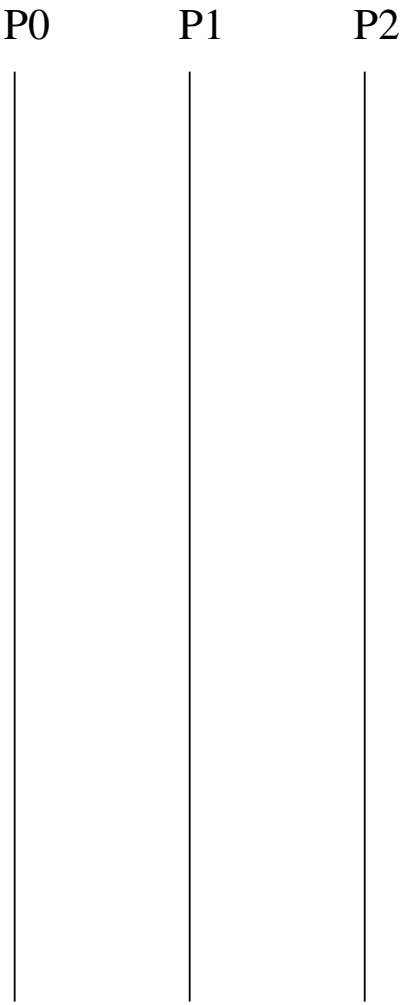
1. P_0 demande à entrer en section critique, y accède puis sort
2. P_1 demande à entrer en section critique, y accède puis sort

Algorithme de Lamport

Initialisation, état des variables

Var	h	F_H	F_M
P_0	0	0,0,0	REL, REL, REL
P_1	0	0,0,0	REL, REL, REL
P_2	0	0,0,0	REL, REL, REL

Chronogramme :

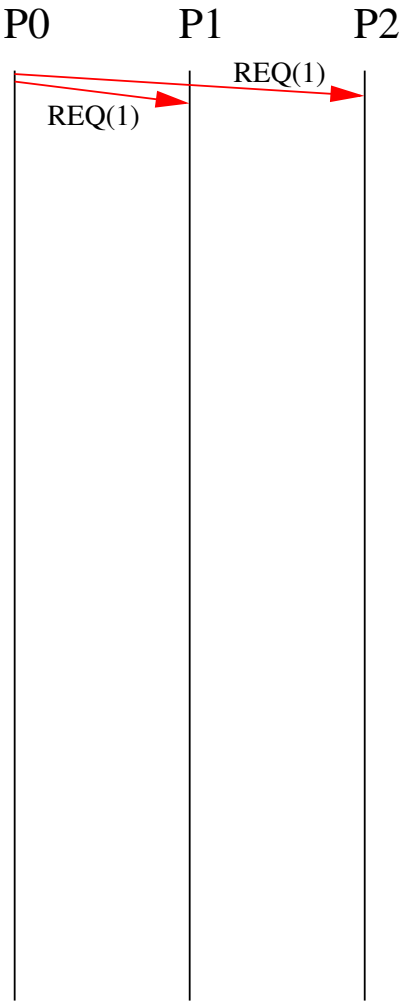


Algorithme de Lamport

P_0 demande à entrer en SC

Var	h	F_H	F_M
P_0	1	1,0,0	REQ, REL, REL
P_1	0	0,0,0	REL, REL, REL
P_2	0	0,0,0	REL, REL, REL

Chronogramme :

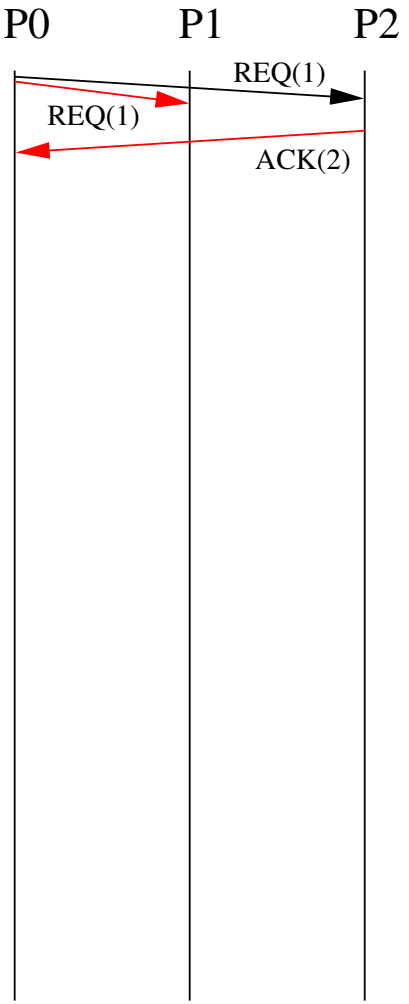


Algorithme de Lamport

P_2 reçoit la requête `REQ(1)` de P_0

Var	h	F_H	F_M
P_0	1	1,0,0	REQ, REL, REL
P_1	0	0,0,0	REL, REL, REL
P_2	2	1,0,0	REQ, REL, REL

Chronogramme :



Algorithme de Lamport

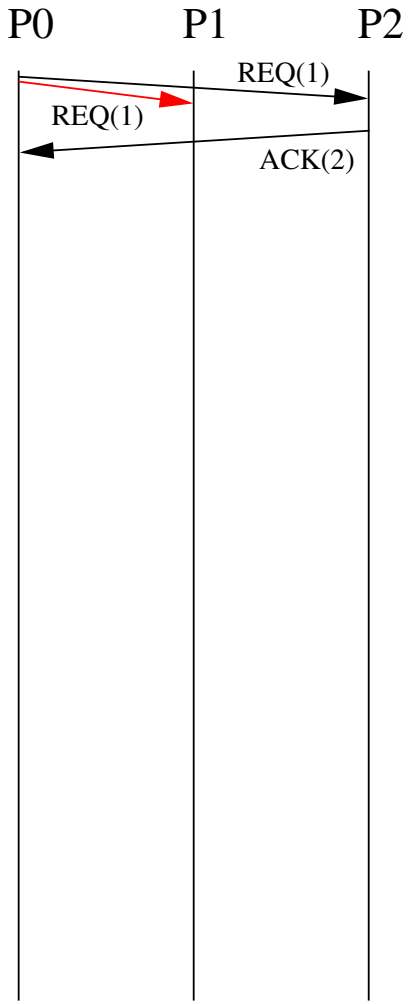
P_0 reçoit la réponse `ACK(2)` de P_2

Var	h	F_H	F_M
P_0	3	1,0,2	REQ, REL, ACK
P_1	0	0,0,0	REL, REL, REL
P_2	2	1,0,0	REQ, REL, REL

Condition : fausse en P_0

$F\_H[0] < F\_H[2]$ mais $F\_H[0] > F\_H[1]$

Chronogramme :

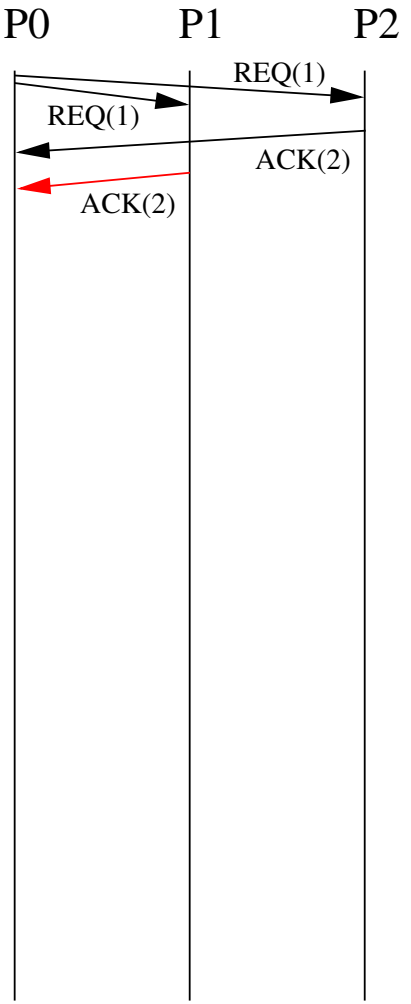


Algorithme de Lamport

P_1 reçoit la requête `REQ(1)` de P_0

Var	h	F_H	F_M
P_0	3	1,0,2	REQ, REL, ACK
P_1	2	1,0,0	REQ, REL, REL
P_2	2	1,0,0	REQ, REL, REL

Chronogramme :



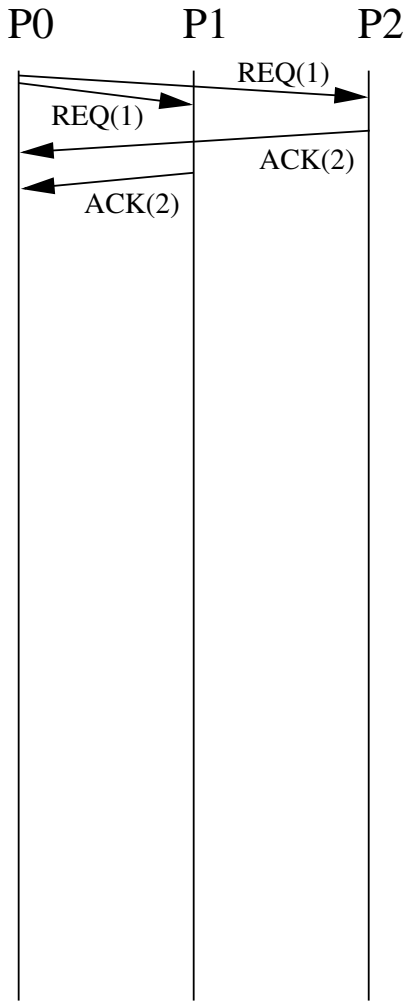
Algorithme de Lamport

P_0 reçoit la réponse `ACK(2)` de P_1

Var	h	F_H	F_M
P_0	4	1,2,2	REQ, ACK, ACK
P_1	2	1,0,0	REQ, REL, REL
P_2	2	1,0,0	REQ, REL, REL

Condition : vraie en P_0
$F\_H[0] < F\_H[1]$ et $F\_H[0] < F\_H[2]$
P_0 accède en SC

Chronogramme :

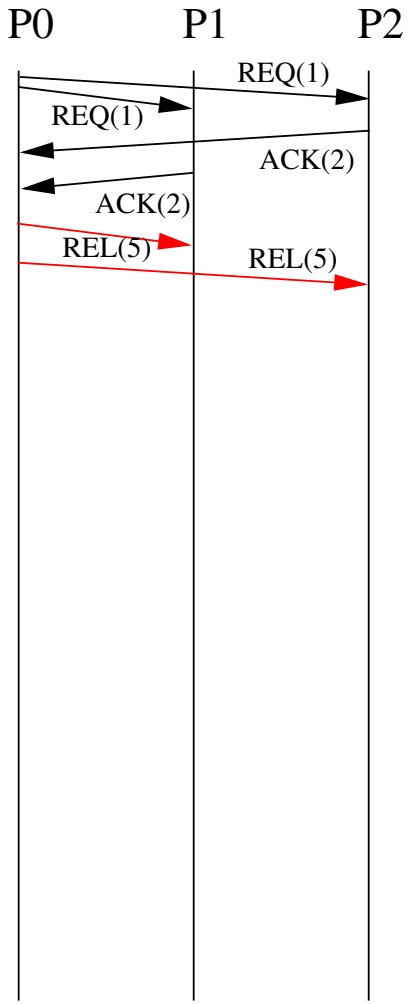


Algorithme de Lamport

P_0 sort de SC

Var	h	F_H	F_M
P_0	5	5,2,2	REL, ACK, ACK
P_1	2	1,0,0	REQ, REL, REL
P_2	2	1,0,0	REQ, REL, REL

Chronogramme :

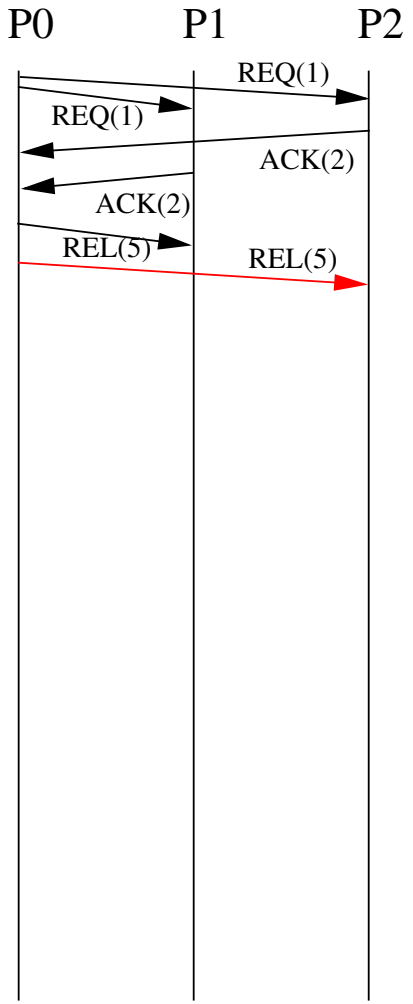


Algorithme de Lamport

P_1 reçoit le message $REL(5)$ de P_0

Var	h	F_H	F_M
P_0	5	5,2,2	REL, ACK, ACK
P_1	6	5,0,0	REL, REL, REL
P_2	2	1,0,0	REQ, REL, REL

Chronogramme :

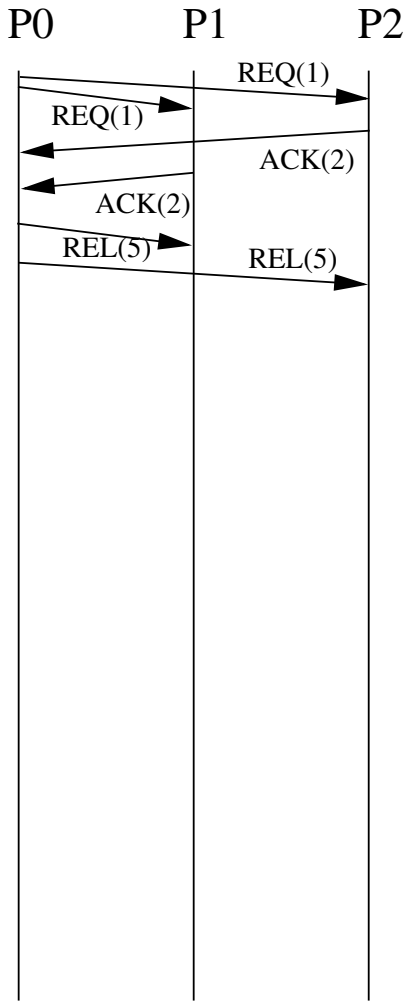


Algorithme de Lamport

P_2 reçoit le message $REL(5)$ de P_0

Var	h	F_H	F_M
P_0	5	5,2,2	REL, ACK, ACK
P_1	6	5,0,0	REL, REL, REL
P_2	6	5,0,0	REL, REL, REL

Chronogramme :

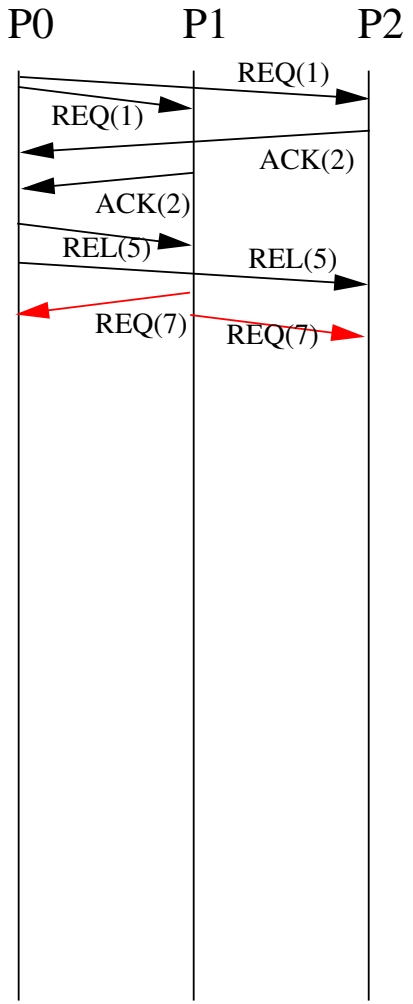


Algorithme de Lamport

P_1 demande à entrer en SC

Var	h	F_H	F_M
P_0	5	5,2,2	REL, ACK, ACK
P_1	7	5,7,0	REL, REQ, REL
P_2	6	5,0,0	REL, REL, REL

Chronogramme :

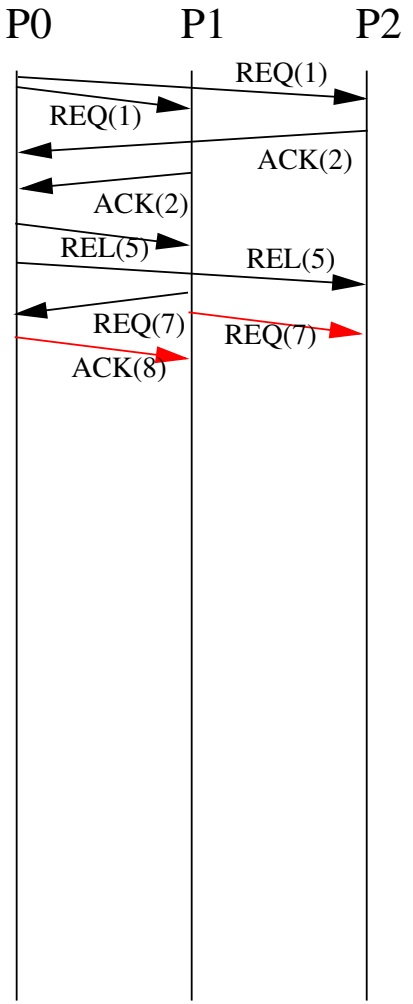


Algorithme de Lamport

P_0 reçoit `REQ(7)` de P_1

Var	h	F_H	F_M
P_0	8	5,7,2	REL, REQ, ACK
P_1	7	5,7,0	REL, REQ, REL
P_2	6	5,0,0	REL, REL, REL

Chronogramme :

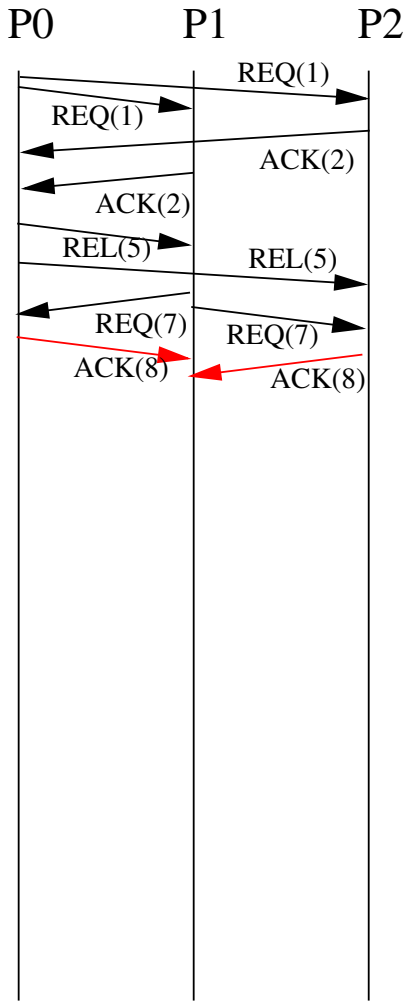


Algorithme de Lamport

P_2 reçoit `REQ(7)` de P_1

Var	h	F_H	F_M
P_0	8	5,7,2	REL, REQ, ACK
P_1	7	5,7,0	REL, REQ, REL
P_2	8	5,7,0	REL, REQ, REL

Chronogramme :



Algorithme de Lamport

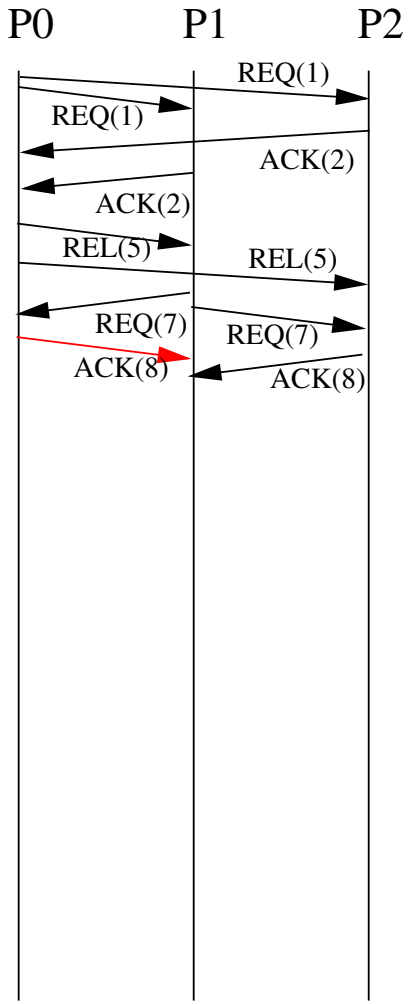
P_1 reçoit `ACK(8)` de P_0

Var	h	F_H	F_M
P_0	8	5,7,2	REL, REQ, ACK
P_1	9	8,7,0	ACK, REQ, REL
P_2	8	5,7,0	REL, REQ, REL

Condition : fausse en P_1

$F\_H[1] < F\_H[0]$ mais $F\_H[1] > F\_H[2]$

Chronogramme :



Algorithme de Lamport

P_1 reçoit ACK(8) de P_2

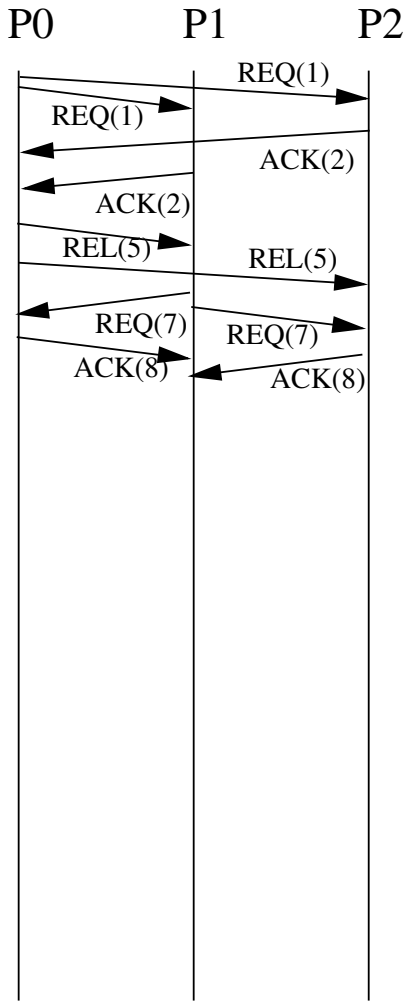
Var	h	F_H	F_M
P_0	8	5,7,2	REL, REQ, ACK
P_1	10	8,7,8	ACK, REQ, ACK
P_2	8	5,7,0	REL, REQ, REL

Condition : vraie en P_1

$F\_H[1] < F\_H[0]$ et $F\_H[1] < F\_H[2]$

P_1 entre en SC

Chronogramme :

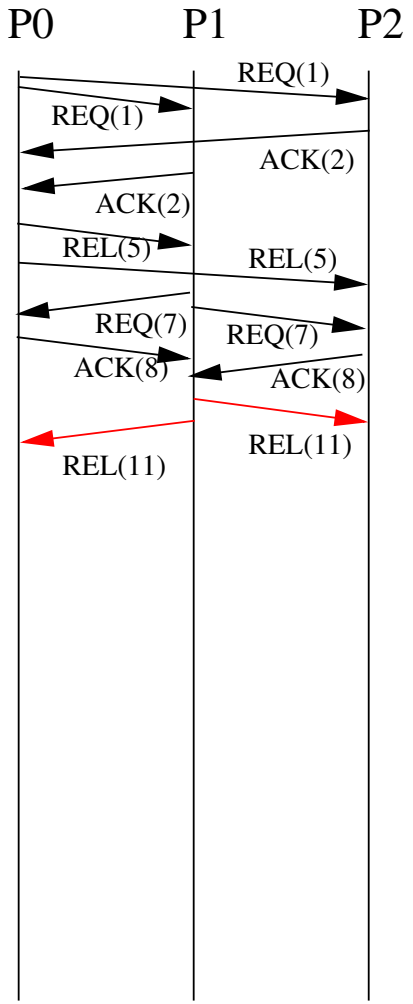


Algorithme de Lamport

P_1 sort de SC

Var	h	F_H	F_M
P_0	8	5,7,2	REL, REQ, ACK
P_1	11	8,11,8	ACK, REL, ACK
P_2	8	5,7,0	REL, REQ, REL

Chronogramme :

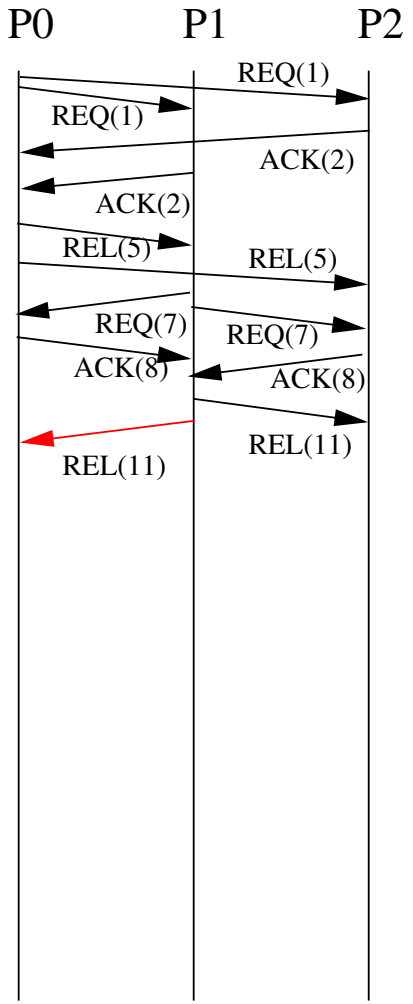


Algorithme de Lamport

P_2 reçoit `REL(11)` de P_1

Var	h	F_H	F_M
P_0	8	5,7,2	REL, REQ, ACK
P_1	11	8,11,8	ACK, REL, ACK
P_2	12	5,11,0	REL, REL, REL

Chronogramme :

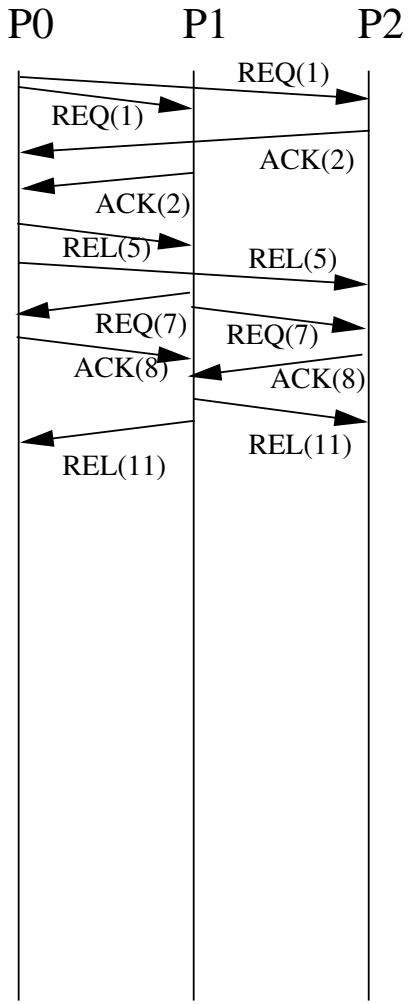


Algorithme de Lamport

P_0 reçoit `REL(11)` de P_1

Var	h	F_H	F_M
P_0	12	5,11,2	REL, REL, ACK
P_1	11	8,11,8	ACK, REL, ACK
P_2	12	5,11,0	REL, REL, REL

Chronogramme :



Déroulement de l'algorithme de Lamport

Exercice avec 3 processus : P_0, P_1 et P_2

1. Depuis l'état obtenu à l'exécution précédente
2. Les processus P_0 et P_2 demandent la SC en même temps. Dérouler l'algorithme jusqu'à ce que le premier des deux sorte de SC.
3. P_1 demande à entrer en SC à ce moment là. Dérouler l'algorithme jusqu'à ce que P_1 sorte de SC.
4. Les processus P_1 et P_2 demandent la SC en même temps. Dérouler l'algorithme jusqu'à ce que P_1 et P_2 sortent de SC.

Exercice

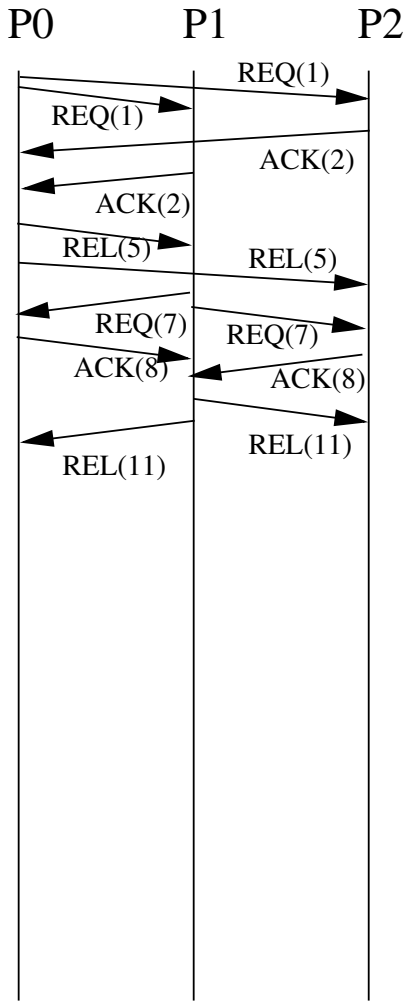
Que se passe-t-il si la propriété FIFO n'est pas respectée ?

Algorithme de Lamport : sensibilité FIFO

État initial précédent

Var	h	F_H	F_M
P_0	12	5,11,2	REL, REL, ACK
P_1	11	8,11,8	ACK, REL, ACK
P_2	12	5,11,0	REL, REL, REL

Chronogramme :

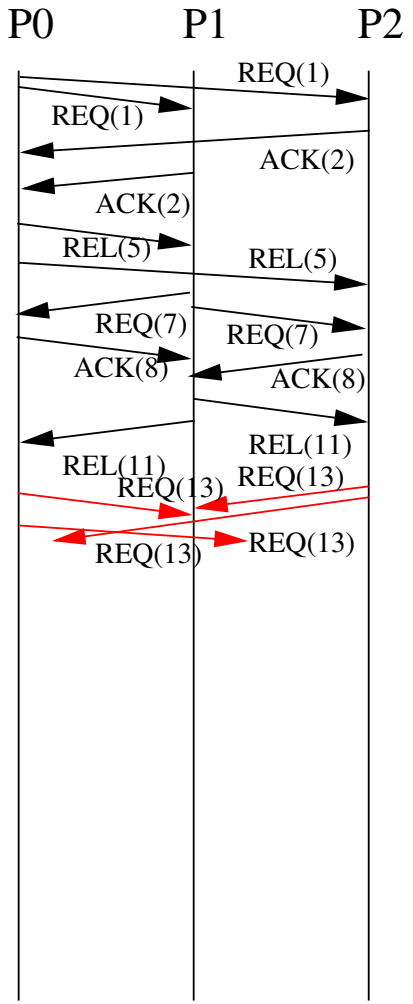


Algorithme de Lamport : sensibilité FIFO

Les processus P_0 et P_2 demandent la SC en même temps

Var	h	F_H	F_M
P_0	13	13,11,2	REQ, REL, ACK
P_1	11	8,11,8	ACK, REL, ACK
P_2	13	5,11,13	REL, REL, REQ

Chronogramme :



Algorithme de Lamport : sensibilité FIFO

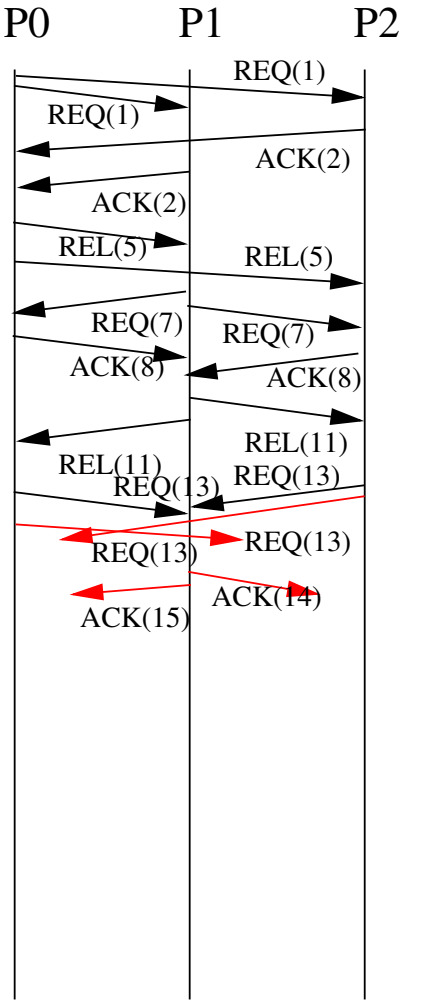
P_1 reçoit le message REQ(13) de P_2

Var	h	F_H	F_M
P_0	13	13,11,2	REQ, REL, ACK
P_1	14	8,11,13	ACK, REL, REQ
P_2	13	5,11,13	REL, REL, REQ

P_1 reçoit le message REQ(13) de P_0

Var	h	F_H	F_M
P_0	13	13,11,2	REQ, REL, ACK
P_1	15	13,11,13	REQ, REL, REQ
P_2	13	5,11,13	REL, REL, REQ

Chronogramme :



Algorithme de Lamport : sensibilité FIFO

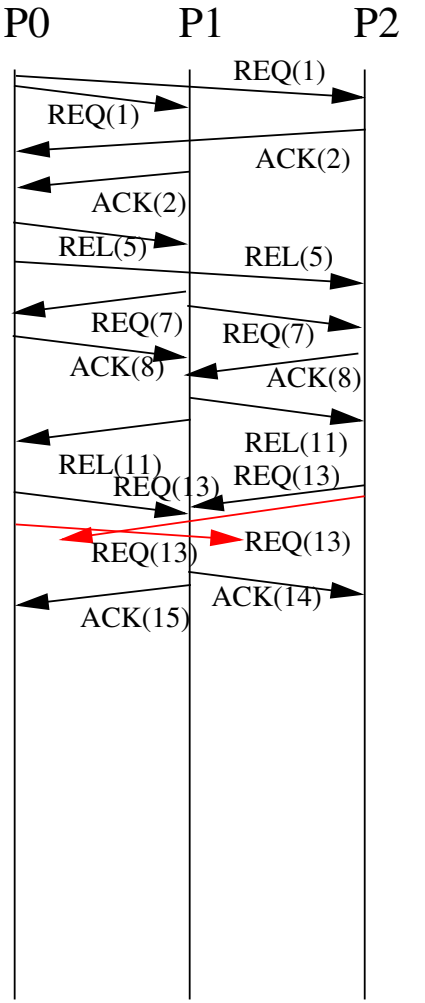
P_2 reçoit le message ACK(14) de P_1

Var	h	F_H	F_M
P_0	13	13,11,2	REQ, REL, ACK
P_1	15	13,11,13	REQ, REL, REQ
P_2	15	5,14,13	REL, ACK, REQ

P_0 reçoit le message ACK(15) de P_1

Var	h	F_H	F_M
P_0	16	13,15,2	REQ, ACK, ACK
P_1	15	13,11,13	REQ, REL, REQ
P_2	15	5,14,13	REL, ACK, REQ

Chronogramme :



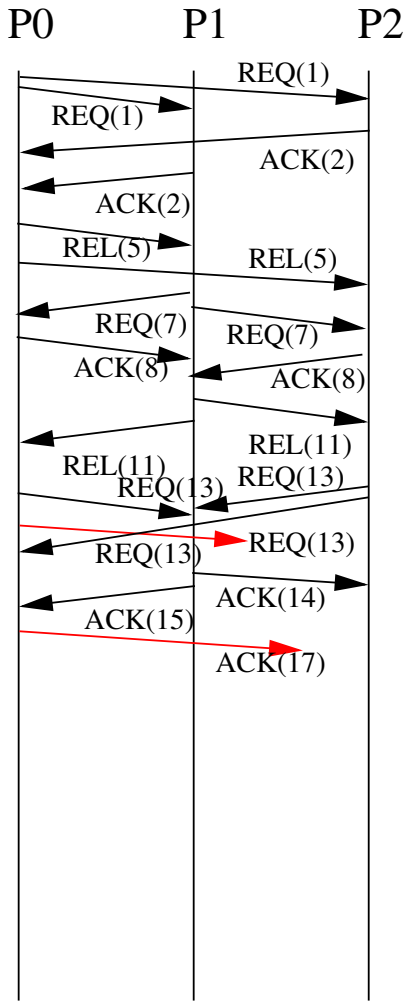
Algorithme de Lamport : sensibilité FIFO

P_0 reçoit le message REQ(13) de P_2

Var	h	F_H	F_M
P_0	17	13,15,13	REQ, ACK, REQ
P_1	15	13,11,13	REQ, REL, REQ
P_2	15	5,14,13	REL, ACK, REQ

P_0 envoie le message ACK(17) à P_2
P_0 entre en SC
Son message ACK(17) double REQ(13)

Chronogramme :



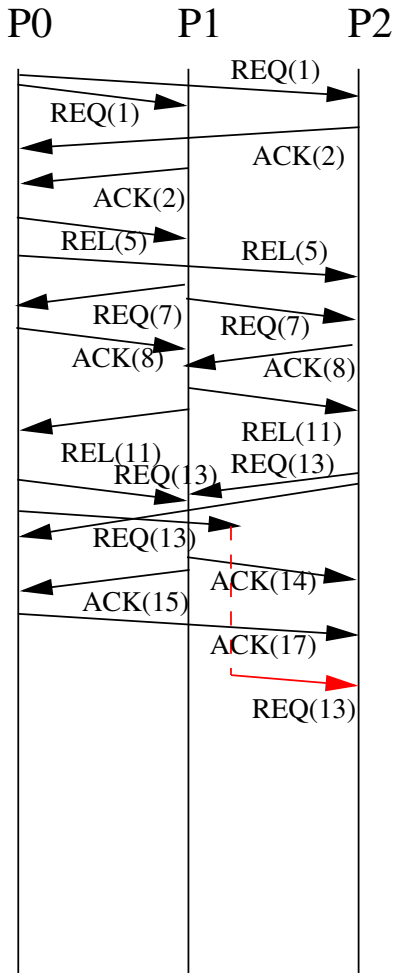
Algorithme de Lamport : sensibilité FIFO

P_2 reçoit le message ACK(17) de P_0

Var	h	F_H	F_M
P_0	17	13,15,13	REQ, ACK, REQ
P_1	15	13,14,13	REQ, ACK, REQ
P_2	18	17,14,13	ACK, ACK, REQ

P_2 ne voit pas la demande de P_0
P_2 entre en SC

Chronogramme :



Algorithme de Lamport

Propriétés de l'algorithme

- Sûreté : exclusion mutuelle garantie
- Vivacité garantie
- Algorithme exempt d'inter-blocage

Algorithme de Lamport : Preuve sûreté

Par contradiction :

- Supposons que P_i et P_j sont en même temps dans la SC à l'instant t,
- Alors la condition d'accès doit être valide sur les deux processus en même temps et ils sont tous les deux dans l'état REQ
- Les messages de requêtes étant datés, on peut supposer sans perte de généralité que , $H(P_i) < H(P_j)$ (ou l'inverse, ce qui revient au même) puisque deux horloges logiques ne sont jamais identiques
- A partir de l'hypothèse de communication FIFO, il est clair que le message de requête de P_i est dans le tableau F_M de P_j au moment où celui-ci entre en SC puisque P_j attend toutes les réponses (ACK) des processus avant d'entrer, donc celui de de P_i.
- Or l'acquittement de P_i a été généré après que P_i ait émis le message REQ puisque $H(P_i) < H(P_j)$
- Donc P_j est entré en SC alors que $H(P_i) < H(P_j) \rightarrow$ contradiction

Algorithme de Lamport

Vivacité

- Toute demande d'entrée en Section Critique sera satisfaite au bout d'un temps fini.
- En effet, après la demande d'un processus P_i, au plus, $N-1$ processus peuvent entrer avant lui

Complexité

$3 * (N-1)$ messages par entrée en Section Critique, où N est le nombre de processus

- $N-1$ Requêtes
- $N-1$ Acquittements
- $N-1$ Libérations

Algorithme de Ricart et Agrawala

Objectif

Algorithme proposé dans le but de diminuer le nombre de messages par rapport à l'algorithme de Lamport

Changements

- Regrouper information de sortie de SC et accord
- Pas de retour systématique de ACK
- N'informer que les processus en attente à la sortie de SC

Algorithme de Ricart et Agrawala

Principe de l'algorithme

- Lorsqu'un processus P_i demande la Section Critique, il diffuse une requête datée à tous les autres processus.
- Lorsqu'un processus P_i reçoit une requête de demande d'entrée en Section Critique de P_j, deux cas sont possibles :
  - Cas 1 : si le processus P_i n'est pas demandeur de la Section Critique, il envoie un accord à P_j.
  - Cas 2 : si le processus P_i est demandeur de la Section Critique et si la date de demande de P_j est plus récente que la sienne, alors la requête de P_j est différée, sinon un message d'accord est envoyé à P_j.
- Lorsqu'un processus P_i sort de la Section Critique, il diffuse à tous les processus dont les requêtes sont différées, un message de libération.

Algorithme de Ricart et Agrawala

Variables utilisées par chaque processus P_i :

h : entier, estampille locale. Init à 0
hsc : entier, estampille de demande de SC. Init à 0
r : booléen, le processus est demandeur de SC. Init à FAUX
X : ensemble, processus dont l'accord est différé. Init à $\emptyset$
$nrel$: entier, nombre d'accords attendus. Init à 0.
V : voisinage de P_i

Messages utilisés :

$REQ(h)$: demande d'entrée en SC
$REL()$: message de permission

Algorithme de Ricart et Agrawala

Règle 1 : P_i demande la SC

Faire

$r \leftarrow VRAI$

$h \leftarrow h + 1$

$hsc \leftarrow h$

$nrel \leftarrow cardinal(V)$

Pour $P_j \in V$: P_i **envoie** $REQ(hsc)$ à P_j

attendre $(nrel = 0)$

$< SC >$

Fait

Algorithme de Ricart et Agrawala

Règle 2 : P_i **reçoit** $REQ(h_j)$ de P_j

Faire

$h \leftarrow \max(h, h_j) + 1$

Si $r \wedge ((hsc < h_j) \vee ((hsc = h_j) \wedge i < j))$ **Alors**

$X \leftarrow X \cup P_j$

Sinon

P_i **envoie** $REL()$ à P_j

FSi

Fait

Algorithme de Ricart et Agrawala

Règle 3 : P_i **reçoit** $REL()$ de P_j

Faire

$nrel \leftarrow nrel - 1$

Fait

Règle 4 : P_i libère de la SC

Faire

$r \leftarrow FAUX$

Pour $P_j \in X$: P_i **envoie** $REL()$ à P_j

$X \leftarrow \emptyset$

Fait

Algorithme de Ricart et Agrawala

Déroulement avec 3 processus : P_0, P_1, P_2

1. Le processus P_1 demandent l'accès à la Section Critique, jusqu'à sa sortie de SC
2. Le processus P_2 demande l'accès à la Section Critique, une fois qu'il est entré, P_1 demande, jusqu'à la sortie des deux.
3. P_0 demande la SC en même temps que P_2, lorsque l'un des deux processus est en SC, P_1 demande, jusqu'à la sortie des trois.

Question

Pourquoi deux horloges logiques ?

Algorithme de Ricart et Agrawala

Preuve

- Par contradiction
- On suppose que P_i et P_j sont les deux en section critique.
- On peut supposer sans perte de généralité que la requête de P_i a une horloge plus petite que celle de P_j
- Puisque l'horloge de P_i est plus petite que celle de P_j cela signifie qu'il a reçu la requête de P_j après avoir fait sa demande
- D'après l'algorithme P_j ne peut entrer en section critique que si P_i lui a envoyé un message *REL*
- Puisque P_i est en section critique alors il a envoyé ce message *REL* avant d'entrer en section critique
- Ce qui est impossible puisque l'horloge de P_j est supérieure à celle de P_i, or d'après l'algorithme P_i n'envoie *REL* que si l'horloge reçue est plus petite

Algorithme de Ricart et Agrawala

Complexité

Une utilisation de la Section Critique nécessite $2 * (N - 1)$ messages :

- $N - 1$ Requêtes
- $N - 1$ Permissions

Synchronisation distribuée utilisant la notion d'horloges logiques

Algorithme de Carvalho et Roucairol

Algorithme proposé dans le but de diminuer le nombre de messages par rapport à l'algorithme de Ricart et Agrawala

Algorithme de Carvalho et Roucairol

Principe de l'algorithme

Soit le cas où P_i demande l'accès à la Section Critique plusieurs fois de suite (état ***demandeur*** plusieurs fois de suite) alors que P_j n'est pas intéressé par celle-ci (état ***dehors***)

- Avec l'algorithme de Ricart et Agrawala, P_i demande la permission de P_j à chaque nouvelle demande d'accès à la Section Critique
- Avec l'algorithme de Carvalho et Roucairol, puisque P_j a donné sa permission à P_i, ce dernier la considère comme acquise jusqu'à ce que P_j demande sa permission à P_i : P_i ne demande qu'une fois la permission à P_j

Algorithme de Carvalho et Roucairol

Variables utilisées par chaque processus P_i

hsc : entier, estampille de demande d'entrée en SC, initialisée à 0
h : entier, estampille locale, initialisée à 0
r : booléen, le processus est demandeur de SC, init à FAUX
sc : booléen, le processus est en SC, init à $sc =$ FAUX
X : ensemble, processus dont l'envoi de l'avis de libération est différé. Init à $\emptyset$
XA : ensemble des processus desquels i attend une autorisation. Init à V
nrel : nombre d'avis de libération attendus, init à 0.
V : voisinage de P_i

Algorithme de Carvalho et Roucairol

Messages utilisés

$REQ(h)$: demande d'entrée en SC
$REL()$: message de permission

Algorithme de Carvalho et Roucairol

Règle 1 : P_i demande la SC

Faire

$r \leftarrow VRAI$

$h \leftarrow h + 1$

$hsc \leftarrow h$

$nrel \leftarrow cardinal(XA)$

Pour $P_j \in XA : P_i$ **envoie** $REQ(hsc)$ à P_j

Attendre $(nrel = 0)$

$sc \leftarrow VRAI$

$<$ SC$>$

Fait

Algorithme de Carvalho et Roucairol

Règle 2 : P_i **reçoit** $REQ(h_j)$ de P_j

Faire
  $h \leftarrow \max(h, h_j) + 1$
  Si $sc \vee (r \wedge ((hsc < h_j) \vee ((hsc = h_j) \wedge i < j)))$ **Alors**
    $X \leftarrow X \cup P_j$
  Sinon
    P_i **envoie** $REL()$ à P_j
    Si $r \wedge (\neg sc) \wedge j \notin XA$ **Alors**
      Envoyer$REQ(hsc)$ à j
      $nrel \leftarrow nrel + 1$
    FSi
    $XA \leftarrow XA \cup P_j$
  FSi
Fait

Algorithme de Carvalho et Roucairol

Règle 3 : P_i **reçoit** $REL()$ de P_j

Faire

$nrel \leftarrow nrel - 1$

Fait

Règle 4 : P_i libère la SC

Faire

$r \leftarrow FAUX$

$sc \leftarrow FAUX$

$XA \leftarrow X$

Pour $P_j \in X$: P_i **envoie** $REL()$ à P_j

$X \leftarrow \emptyset$

Fait

Algorithme de Carvalho et Roucairol

Déroulement de l'algorithme avec 3 processus : P_0, P_1 et P_2

1. Les horloges logiques de chaque processus sont à 0
2. Les processus P_2 et P_1 demandent l'accès à la Section Critique en même temps
3. Déroulement de l'algorithme jusqu'à ce que les deux processus P_1 et P_2 soient entrés et sortis de Section Critique
4. Les processus P_0 et P_1 demandent la section critique
5. Arrêt du déroulement de l'algorithme lorsque les trois processus P_0 et P_1 seront entrés et sortis de Section Critique

Algorithme de Carvalho et Roucairol

Question

Pourquoi un processus qui reçoit une REQ renvoie-t-il parfois une REQ ?

Complexité

Le nombre de messages requis pour une utilisation de la Section Critique est :

- pair car à toute requête d'entrée en SC correspond un envoi de permission
- fonction de la structure du système : varie entre 0 et 2*(N-1)

Sommaire

Synchronisation distribuée utilisant un jeton

Algorithmes

Différentes structures de communication :

- Anneau : Le Lann
- Arbre : Naimi-Trehel
- Diffusion : Suzuki-Kazami

Algorithme de Suzuki-Kasami

Principe de l'algorithme

- Chaque processus connaît l'ensemble des participants
- Quand P_i veut accéder à la Section critique, il envoie un message $REQ(h)$ à tous les participants
- A la réception d'une requête le processus mémorise la date H associée
- Le jeton contient l'estampille de la dernière visite qu'il a effectuée à chacun des processus
- Quand P_i sort de Section Critique, il cherche le premier processus P_k tel que l'estampille de la dernière requête de P_k soit supérieure à celle mémorisée dans le jeton (correspondant à la dernière visite du jeton à P_k)

Algorithme de Suzuki-Kasami

Exercices

- Écrire l'algorithme
- Dérouler l'algorithme avec 4 processus P_0 à P_3 : P_1 demande l'accès à la SC. Lorsque P_1 est en SC, P_0 demande l'accès. Puis, lorsque P_0 est en SC, P_1 et P_3 demandent l'accès simultanément. Dérouler jusqu'à ce que les deux derniers processus soient sortis de SC.
- Quelles sont les propriétés de l'algorithme ?
- Comment améliorer l'équité ?