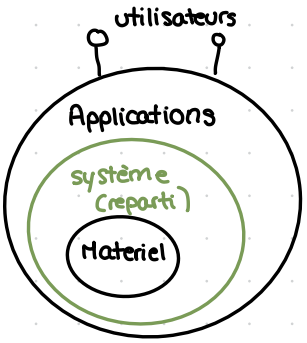
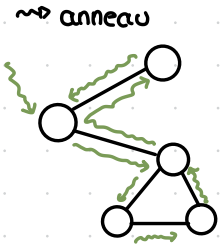
Cours 3 21/01/26



1) Communication

diffusion
récolte (gathering)
construction de structures



2) Contrôle

- Election de leader
- Exclusion mutuelle
- detection de terminaison
- snapshots (instantanés)

3) Temps (séquencement)

- horloges (logiques)

4) Tolérance aux défaillances

- Autostabilisation

Complexité:
- en messages
  - nbr de messages émis / bits émis
- en temps

→ Algorithme diffusion (nouvelle)

type canal = entier
voisins : ensemble de canal;
déjà_vu: booléen initialisé à faux

un processus sur réception de Nouvelle:
si déjà_vu = faux alors
  pour tout c dans voisins (c, Nouvelle);

Diffusion de Nouvelle sur un arbre préalablement construit (fils) avec comme racine le processus recevant Nouvelle:
fils: ensemble de canal;
racine booléen initialisé à faux sauf pour le processus recevant Nouvelle,
le processus racine sur réception de nouvelle →
  si fils ≠ ∅ alors
    pour tout c dans fils envoyer (c, Nouvelle)
un processus sur réception de Nouvelle →
  si fils ≠ ∅ alors
  pour tout c dans fils envoyer (c, nouvelle)

* missing: cours 3

Cours 4 28/01/26

Diffusion sur un arbre avec détéction de terminaison par la racine

constantes: fils = ensemble de canal;
père = canal

variables: racine: booléen initialisé à faux sauf pour un processus;
attendus: ensemble de canal

la racine sur réception de nouvelle →
si fils = ∅ alors <terminer> sinon
attendus := fils;
pour tout c dans fils envoyer(c, Nouvelle);

attendus = {1, 32}
1) racine
1 32
18 fin 22 attendus = {18, 22}

fin = aquittement
un processus sur reception de Nouvelle →
si fils = ∅ alors envoyer (père fin)
sinon attendus := fils; pour tout c dans attendu envoyer(c, fin(Nouvelle));

un processus sur réception de fin (Nouvelle) sur le canal c →
attendus := attendus - {c};
si attendus = ∅ alors
si racine alors <terminer>
sinon envoyer (père, fin (Nouvelle)

nouvelle : 1 par arrête
aquittement : 1 par arrête
) n ⇒ 2n
ne peut pas être executée avec moins de n message (sinon ∃ un processus qui n'a pas reçu de message)

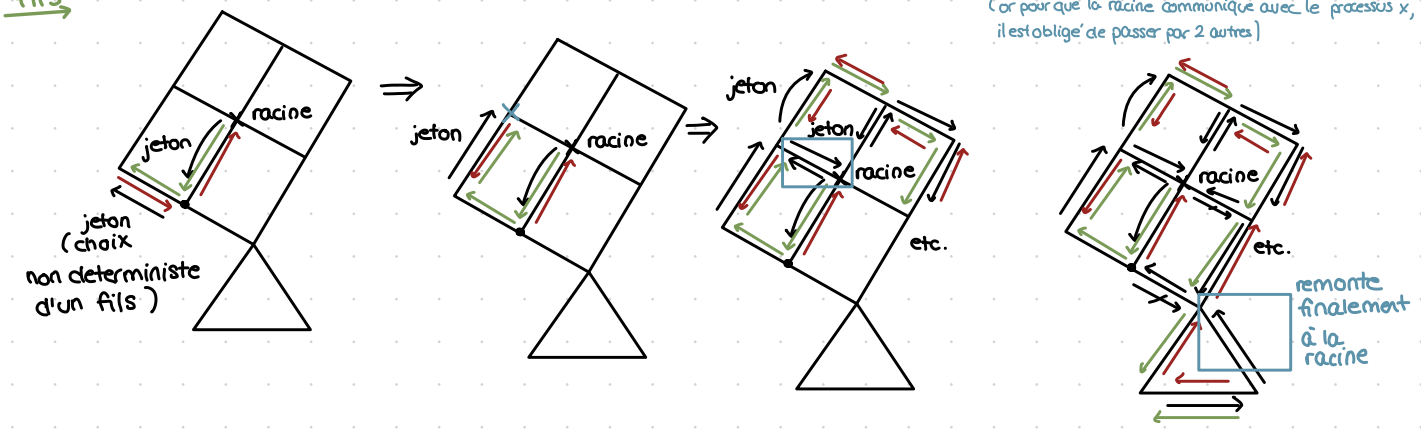
II. Comment construire un arbre recouvrant d'un graphe connexe quelconque?

1) Construction par exploration séquentielle: (Audio 1)

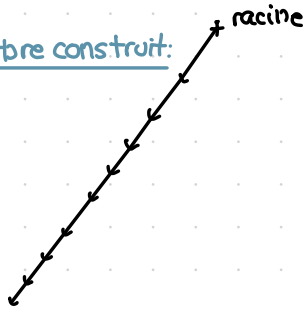
Le jeton visite successivement tous les processus.

"aquittement pour signaler qu'il ne veut pas être fils de la racine"
↳ séquentiellement, il est plus probable de tomber sur un processus sans père)

père → fils



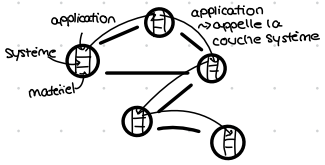
arbre construit:



code associé:

constantes et variables

voisins = ensemble de canal;
initiateur, participant : booléen initialisé à faux;
encore, fils : ensemble de canal;
père : canal;



un processus sur réception de Unit →

```
initiateur := vrai;
participant := vrai;
si voisins = ∅ alors terminer
sinon pourtout c dans voisins envoyer (c, Vu); □
    choisir c dans voisins
    envoyer (c, exploration);
    fils := {c};
    encore := encore - {fils};
```

un processus sur réception de Vu sur reception de Vu sur le canal c →

```
encore = encore - {fils};
```

un processus sur réception de exploration sur le canal c →

```
si non participant alors | participant = vrai;
                         | père = c;
                         | pour tout c dans voisins envoyer(c, Vu);
                         | encore := encore - {père};
                         si encore ≠ 0 alors
                             choisir c' dans encore;
                             envoyer (c', exploration);
                             fils := {c'};
                             encore := encore - fils;
                         sinon envoyer (père, retour);
sinon envoyer (c, annule);
```

un processus sur réception de retour ou d'annule sur le canal c →

```
si annule alors fils := fils - {c}
si encore ≠ ∅ alors
        choisir c dans encore
        envoyer (c, exploration);
        fils := fils ∪ {c};
sinon si initiateur alors <terminer>
      sinon envoyer(père, retour);
```

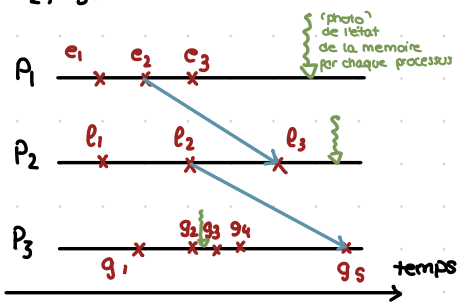
Cours 5 04/02/26

Horloges logiques

horloge: application d'un ensemble d'événements dans un ensemble (partiellement) ordonné, en relation avec une relation de causalité

processus séquentiel:

$a_1, a_2, a_3, \ldots$



événements : entiers | vecteurs d'entiers

$a \longrightarrow \theta(a)$

Une horloge est compatible avec une relation de causalité si $a \rightarrow b \Rightarrow \theta(a) < \theta(b)$

L'horloge est caractéristique de la causalité si $a \rightarrow b \Leftrightarrow \theta(a) < \theta(b)$

Deux événements a et b sont indépendants ssi on n'a ni $a \rightarrow b$, ni $b \rightarrow a$

Horloge de Fidge - Mattern

Histoire de a, $H(a)$ (k premiers événements d'un processus)

$H(a) = \{b \mid b \rightarrow a\} \cup \{a\}$

↑ passé causal de a

Projections $H_1(a), H_2(a), \ldots$,

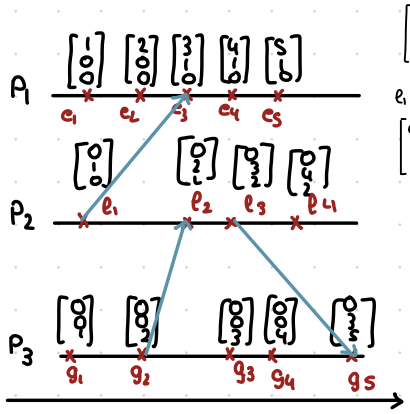
$$\theta(a) = \begin{bmatrix} \text{card } H_1(a) \\ \text{card } H_2(a) \\ | \end{bmatrix}$$

(* pour comparer 2 vecteurs :

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \leq \begin{bmatrix} t \\ u \\ v \end{bmatrix} \Leftrightarrow \begin{bmatrix} x \leq t \\ y \leq u \\ z \leq v \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} < \begin{bmatrix} t \\ u \\ v \end{bmatrix} \Leftrightarrow \begin{bmatrix} x \leq t \\ y \leq u \\ z \leq v \end{bmatrix} \text{ et } \neq \Big)$$

diagramme de causalité:



e_5 et g_5 : $\begin{bmatrix} 5 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 3 \\ 5 \end{bmatrix}$ incomparables

l_1 et g_5 : $\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} < \begin{bmatrix} 0 \\ 3 \\ 5 \end{bmatrix}$

preuve: $\Rightarrow$ transitivité

$$a \rightarrow b \Leftrightarrow H(a) \subsetneq H(b) \text{ et } a \neq b$$

$$\Leftarrow \quad H_1(a) \subseteq H_1(b)$$

$$\Leftrightarrow \quad H_2(a) \subseteq H_2(b) \text{ et } a \neq b$$

$$\Leftrightarrow \quad \text{Card}(H_1(a)) \leq \text{Card}(H_1(b))$$

$$\text{Card}(H_2(a)) \leq \text{Card}(H_2(b)) \text{ et } a \neq b \quad \Leftrightarrow \begin{bmatrix} H_1(a) \\ H_2(a) \\ | \end{bmatrix} \not\leq \begin{bmatrix} H_1(b) \\ H_2(b) \\ | \end{bmatrix}$$

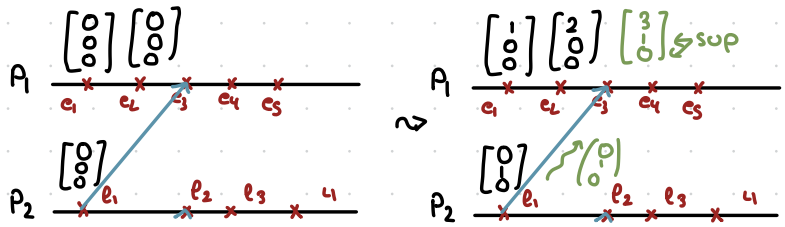
Calcul de H_i par le processus P_i } algo

Initialement $H = \begin{bmatrix} 0 \\ 0 \\ | \\ 0 \end{bmatrix}$

(réception locante)

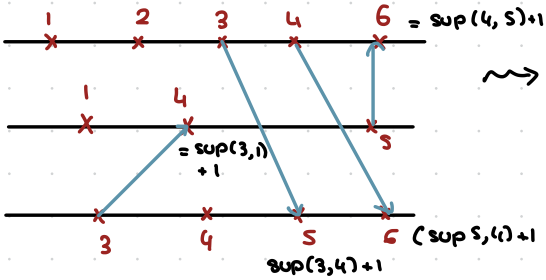
1) Lors d'un événement interne, incrémenter la $i^{ème}$ composante de H
2) Lors d'une émission d'un message m, incrémenter la $i^{ème}$ composante de H et attacher H à m (piggybacking)
3) Lors de la réception d'un message m, incrémenter la $i^{ème}$ composante de H et prendre le sup des autres composantes.

exemple:



Horloge de Lamport:

(ajouter 1 à chaque chemin)



valeur de l'horloge de Lamport de l'evenement a

$\rightsquigarrow$ si $a \rightarrow b$ alors $L(a) \not\leq L(b)$

si on considère $(L(a), i)$

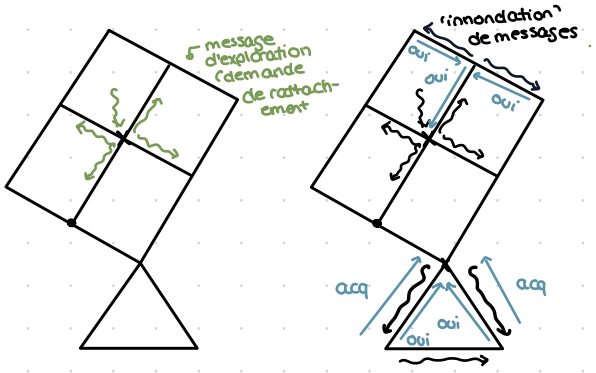
$(L(a), i) \not\leq (L(b), i) \Leftrightarrow L(a) < L(b)$

ou $L(a) = L(b)$ et $i < j$

$\rightsquigarrow$ permet d'avoir un ordre total sur les évenements

Suite cours 4 (Construction d'arbre)

Construction d'arbre par exploration parallèle



code:

variables / constantes:

voisins : ensemble de canal

participant : booléen initialisé à faux

attendus, fils : ensemble de canal

père : canal

racine : booléen initialisé à faux

un processus sur réception de Init →

participant := vrai ; racine := vrai ; fils = ∅ ;

attendus := voisins ;

on supp graphe non-dégénéré

pour tout c dans voisins envoyer (c, explorer) ;

un processus sur réception de Explorer sur le canal c →

si non participant alors | participant := vrai ;
| père := c ;
| fils := ∅ ;
| attendus := voisins - {c} ;
| si attendus ≠ ∅ alors pour tout c' dans attendus
| envoyer (c', explorer)
| sinon envoyer (père, oui)

sinon envoyer (c, Non) ;

un processus sur reception de oui sur le canal c →

fils := fils ∪ {c} ;

attendus := attendus - {c} ;

si attendus = ∅ alors

si racine alors <terminer>

sinon envoyer (père, oui) ;

un processus sur réception de Non sur le canal c →

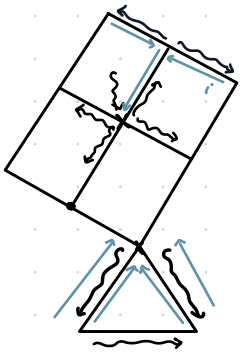
attendus := attendus - {c} ;

si attendus = ∅ alors

si racine alors <terminer>

sinon envoyer (père, oui) ;

Cours 6 11/02/26



attendus: acquittements de vagues
encore: acquittements de terminaux

Bellman-Ford

Algorithme de construction d'arbre recouvrant par vagues synchronisées et détection de terminaison

variables / constantes:
voisins ;
encore, attendus, fils : ensemble de canal;
père : canal ;
niveau : entier;
racine : booléen initialisé à faux ;

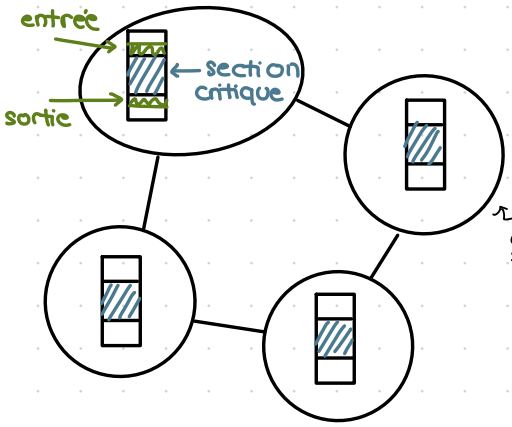
Initiallement, l'environnement contacte un processus.

Un processus sur communication avec l'environnement →
participant := vrai; racine := vrai;
niveau := 0; encore := voisins; ← pour avoir la terminaison
si encore ≠ 0, alors pour tout c dans voisins envoyer(c, aller(1)) ← vague à distance 1 de la racine
attendus := voisins;

un processus sur réception de aller(K) sur le canal c →
si non participant alors | participant = vrai; père := c; niveau := K; fils = ∅;
encore := voisins - {père};
si encore = ∅ alors envoyer(père, retour(fin(K))
sinon envoyer(père, retour(again(K)));
sinon // si participant
si père = c alors | père := voisins - {père};
pour tout c dans encore
envoyer(c, aller(K));
attendus := encore
sinon envoyer(c, retour(non(K)));

un processus sur réception (retour(reponse(K)) sur le canal c →
attendus := attendus - {c};
si(réponse = non) ou (réponse = fin) alors encore := encore - {c};
si(réponse = again) ou (réponse = fin) alors fils := fils ∪ {c};
si encore ≠ 0 et attendus = ∅ alors
si(¬ racine) alors envoyer(père, retour(again(K))
sinon | pour tout c dans encore envoyer(aller(K+1));
attendus := encore;
si encore = 0 alors si(non racine) alors envoyer(père, retour(fin(K));
sinon < terminé >

Exclusion mutuelle des sections critiques.



un processus est dans sa section critique si son compteur de programme pointe vers une instruction de la section critique.

↳ on veut que l'accès à la section critique soit séquentiel

↳ à chaque instant, au plus un processus exécute une instruction dans sa section critique (⇔)

↳ causalement indépendants

Spécification précise d'un problème :

≡ propriété d'exécution

≡ conditions sur les exécutions

Safety (sureté) = satisfiabilité à chaque partie d'exécution (⇔) ↳ ne pas construire un arbre en cycle par exemple

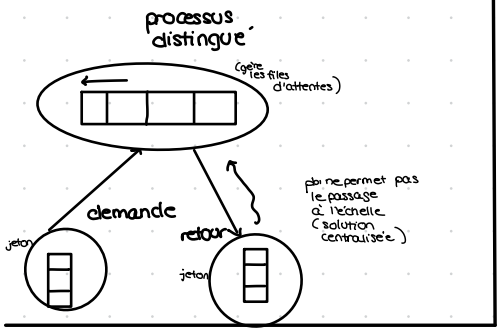
Liveness (vivacité) = satisfaite dans le futur

↳ un processus peut rentrer dans sa section critique au bout d'un temps fini

↳ terminaison par exemple

(ex (pour élection de leader)
S : à aucun moment on a 2 ou plus leaders
E : terminaison

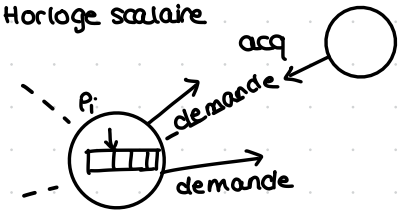
Solution centralisée :



Horloge de Lamport avec l'ordre total :

(∼, i) ↓ ↳ num du processus
↓ valeur de l'horloge de Lamport

Horloge scalaire



canaux FIFO (on conserve l'ordre des messages)

Avant d'entrer en section P_i, un processus attend que l'estampille (↓ le couple) soit la plus petite de toutes les estampilles des messages qu'il a reçu impliquant que la demande est causalement la plus ancienne et unique

---

type_de_message : demande, sortie, acquittement

message : 3 champs : type
{ horloge
{ numéro de processus

file_d_attente : tableau [1, ..., n] de message ; (initialisé à file[i] = (sortie, 0, i);

un processus P_i pour demander à rentrer en S.C ⟶

```
gérer l'horloge;
diffuser(demande, horloge, i);
attendre file[i] contienne le plus petit couple
(estampille, numéro) de toute la file d'attente;
dedans := vrai;
< section critique >
```

un processus qui reçoit un message de demande de p_i →

tant que dedans attendre

gerer l'horloge;

envoyer(acquittement, horloge, i);

un processus A_i qui sort de section critique →

gérer l'horloge; (incrémenter)

diffuser(sortie, horloge, i);

$\forall j$,

à la réception de {acquittement, sortie, j} →

si la cellule j ne contient pas une demande, mettre à joure file(j);

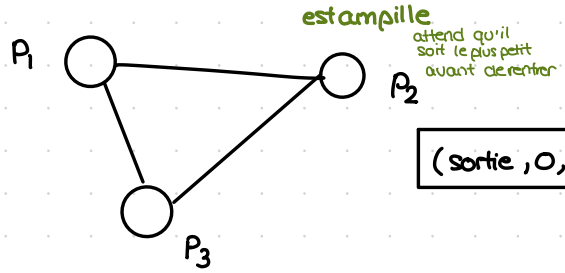
Cours 7 18/02/26

Algorithme d'exclusion mutuelle de Lamport (exemple de déroulement de l'algo)

Tableau demande

P_1	P_2	P_3
(sortie, 0, 1)	(sortie, 0, 2)	(sortie, 0, 3)

estampille



attend qu'il soit le plus petit avant de rentrer

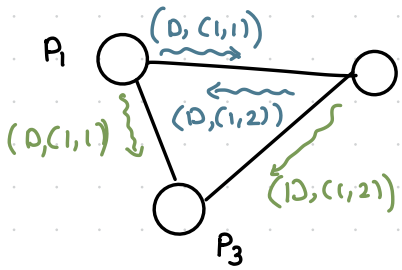
(sortie, 0, 1)	(sortie, 1, 2)	(sortie, 0, 3)

(sortie, 0, 1)	(sortie, 0, 2)	(sortie, 0, 3)

P_1 demande à entrer en S.C.
P_2 demande à entrer en S.C.
Durant toute l'execution, P_3 ne demande pas à entrer en S.C.

⇒

(demande, 1, 1)	(demande, 1, 2)	(sortie, 0, 3)



attend qu'il soit le plus petit avant de rentrer

Δ! canaux FIFO
(l'ordre des demandes est conservé)

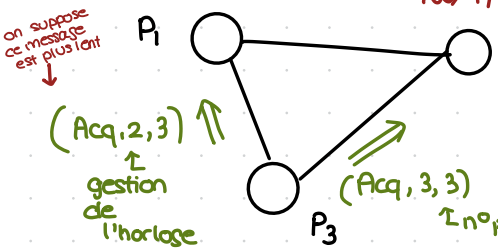
(demande, 1, 1)	(demande, 1, 2)	(sortie, 0, 3)

(demande, 1, 1)	(sortie, 0, 2)	(sortie, 0, 3)

⇒

(demande, 1, 1)	(demande, 1, 2)	(acq, 2, 3)

P_1 en SC
⇝ P_1 entre en section critique



on suppose ce message est plus lent

gestion de l'horloge

n° processus

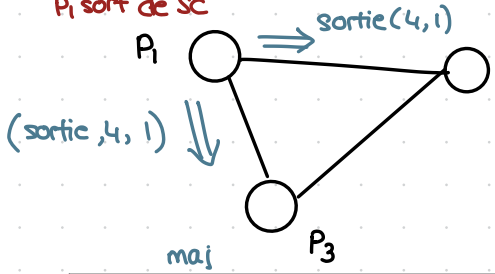
P_2 est tjs bloqué

(demande, 1, 1)	(demande, 1, 2)	(acq, 3, 3)

(demande, 1, 1)	(sortie, 0, 2)	(sortie, 0, 3)

(demande, 1, 1)	(demande, 1, 2)	(acq, 2, 3)

P_1 sort de SC



maj

(sortie, 4, 1)	(demande, 1, 2)	(acq, 3, 3)

⇝ P_2 entre en SC

(demande, 1, 1)	(sortie, 0, 2)	(sortie, 0, 3)

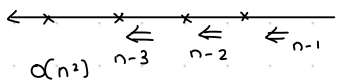
Complexité :

en messages

nb de messages émis pour 1 entrée en S.C :

→ (n-1) (P1 diffuse son message)
+ (n-1) (diffuse son message de sortie) ⇒ 3(n-1)
+ (n-1) (acquittements)

(nb d'arrêtes)
(n-1 si graphe complet)



Algorithme de Ricard et Agrawala :

Privilège : avoir la plus petite estampille parmi les demandes

↳ exclusion mutuelle est réalisée ⇝ seul un processus avec une plus petite peut entrer en S.C. (algo + FIFO)
la plus petite demande est unique (Lamport'élargi)

Faimess : Le processus avec la plus petite estampille finit par entrer en S.C.
(pas de perte de message) (pas de blocage)
- processus "précédé" par les autres processus
  - i) K < n-1 (algorithme)
  - ii) au bout d'un certain temps, K = K-1

Privilège implicite (condition)
R.A. privilège explicite ⇝ jeton

Le 'jeton' est un tableau qui contient le nb d'entrées en S.C. de chaque processus

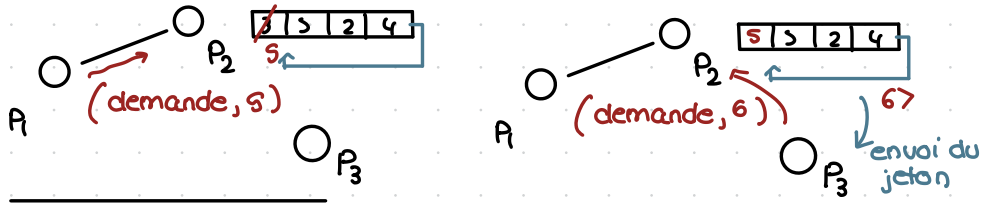
jeton

2	0	5	8

- Lorsqu'un processus qui "veut" entrer en S.C. libère le jeton à la sortie de S.C., il incrémente son compteur (dans le jeton)
- Au départ, le jeton est placé dans un processus particulier, et toutes ses cellules contiennent 0.

Si la possession du jeton garantit l'entrée en S.C et si une entrée en S.C. est impossible sans possession du jeton, l'unicité du jeton garantit l'exclusion mutuelle des S.C.

- Un processus qui veut entrer en S.C. diffuse un message de demande, avec le compteur de ses S.C. passées.



---

algo : (horloge, n processus)

message = (entier, émetteur) ;
jeton = tableau [site] de entier;
procédure attendre_jeton (v : jeton)
{ attend l'arrivée d'un message de type jeton et range sa valeur dans la variable v }
tampon

constantes, variables

{ pour p_i }

const : numéro initialisé à i ;

var : tampon ; demande : jeton ;

estampille : horloge initialisé à 0 ;

jeton_présent : booléen initialisé à (mon_numéro = 1) ;

dedans : booléen initialisé à faux ;

requête : message ;

lors de demande d'entrée en SC →

```
si jeton_présent = faux alors
        début
        | estampille := estampille + 1 ;
        | requête.date := estampille ;
        | requête.émetteur := mon_numéro ;
        | diffuser(requête) ;
        | attendre_jeton(tampon) ;
        fin
    jeton présent := vrai ;
    dedans := vrai ;
```

lors de sortie en S.C →

```
var j : entre site ;
tampon(mon_numéro) := estampille ;
dedans := faux ;
pour j := mon_numéro+1 jusqu'à n, 1 jusqu'à mon_numéro-1
faire :
    si demande(j) > tampon(j) et jeton_présent
    alors | jeton_présent := faux ;
          | envoyer(tampon, j) ;
```

Lors de l'arrivée d'une requête →

```
var h : numéro ;
k := requête_émetteur ;
demande[k] := max( demande[h], requête.date) ;
                  (cas où il reçoit une demande "périmée")
si jeton_présent et dedans = faux alors
    { }
```

→ complexité : en message : $(n-1)+1 = n$

exclusion mutuelle : exclusion du jeton dont la posession est une condition nécessaire et suffisante d'entrer en section critique

Fairness : exploration circulaire

Election sur un anneau

but : désigner un processus particulier, parmis des processus numérotés

spécialisation : Fairness : l'algorithme se termine explicitement

Safety : jamais 2 leaders et 1 seul à la terminaison.

→ graphe en forme d'anneau : - unidirectionnel (successeur)
- bidirectionnel (successeur + prédecesseur)
- orienté ssi étant donnés 2 voisins p_i et p_j, mon_successeur(p_i) = p_j ⟺ mon_prédecesseur(p_j) = p_i

Algorithme de Champ et Roberts