Accord et coordination
Exclusion mutuelle distribuée

Master1 RID

Accord et coordination

- Une famille de problèmes en algorithmique distribuée.
- **Catégories de problèmes de cette famille :**
  - Accord sur l'accès à une ressource partagée
    - **Exclusion mutuelle distribuée**
  - Accord sur l'ordre d'envoi de messages à tous
    - **Diffusion atomique**
  - Accord sur un processus jouant un rôle particulier
    - **Élection d'un maître**
  - Accord sur une valeur commune
    - **Consensus**
  - Accord sur une action à effectuer par tous ou personne
    - **Transaction**

Accord et coordination

- **Architectures générales de solution :**
  - **Avec un élément coordinateur :** par lequel toutes les communications et tous les messages transitent.
    - Simplifie les algorithmes et les solutions.
    - Mais point faible potentiel.
  - **Mise en œuvre totalement distribuée :** processus identiques dont certains peuvent jouer un rôle particulier dans une interaction.
    - Permet plus de facilité pour tolérance aux fautes.
    - Mais algorithmes plus complexes.

Accord sur l'accès à une ressource partagée

« Exclusion mutuelle distribuée »

Plan

- **Définition**
- **Quelques exemples**
- **Méthodes pour gérer l'exclusion mutuelle distribuée**
  - **Contrôle par un coordinateur**
    - **Algorithme d'exclusion mutuelle centralisé implanté en réparti**
  - **Contrôle par jeton**
    - **Algorithme de *« Le Lann »* en 1977**
      - Anneau sur lequel circule le jeton en permanence
    - **Algorithme de *« Ricart et Agrawala »* en 1983**
      - Jeton affecté à la demande des processus
  - **Contrôle par permission**
    - **Permission individuelle : Algorithme de « Ricart et Agrawala » en 1981**
    - **Permission par arbitre : Algorithme de Maekawa en 1985**
      - *Non traité dans ce cours*

Définition de l'exclusion mutuelle distribuée

- Accès à une **ressource partagée distante** par **un seul processus** à la fois.

- Gestion d'accès via des messages échangés entre les processus.

- Nécessité de mettre en œuvre des algorithmes gérant ces échanges de messages pour assurer l'exclusion mutuelle.

Exemples d'exclusion mutuelle distribuée

- **Imprimer en réseau :**

Exemples d'exclusion mutuelle distribuée

- **Plusieurs trains qui passent par un seul tunnel :**

Exemples d'exclusion mutuelle distribuée

- **Guichet chez la poste, la banque, l'aéroport, … :**

Propriétés de l'exclusion mutuelle

- L'accès en exclusion mutuelle doit respecter deux propriétés :
  - **Sûreté (safety) :** au plus un processus est à la fois dans la section critique.
  - **Vivacité (liveness) :** tout processus demandant à entrer dans la section critique y entre en un temps fini.

Etats du processus par rapport à l'accès à la ressource

- Un processus est dans 3 états possibles, par rapport à l'accès à la ressource :
  - **Demandeur :** demande à utiliser la ressource, à entrer dans la section.
  - **Dedans :** dans la section critique, utilise la ressource partagée.
  - **Dehors :** en dehors de la section et non demandeur d'y entrer.

Exemple de plusieurs trains qui passent par un seul tunnel

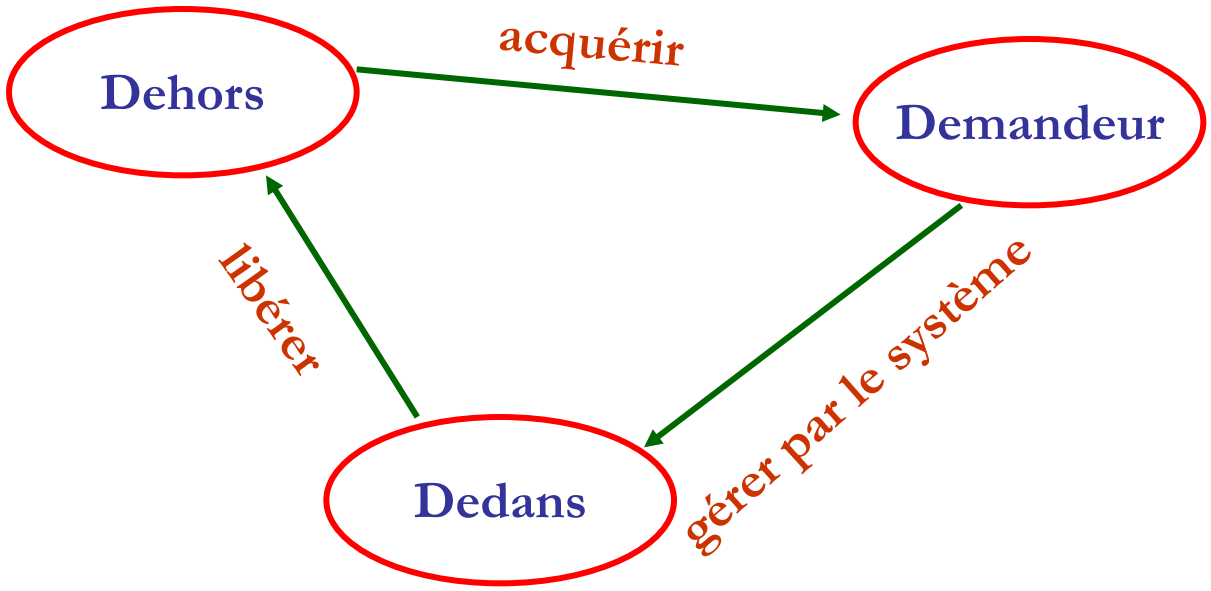
Dehors : libère la ressource

Dedans : utilise la ressource

Demandeur : demande la ressource

Etats du processus par rapport à l'accès à la ressource

- **Diagramme d'états de l'accès en exclusion mutuelle :**



Méthodes pour gérer l'exclusion mutuelle distribuée

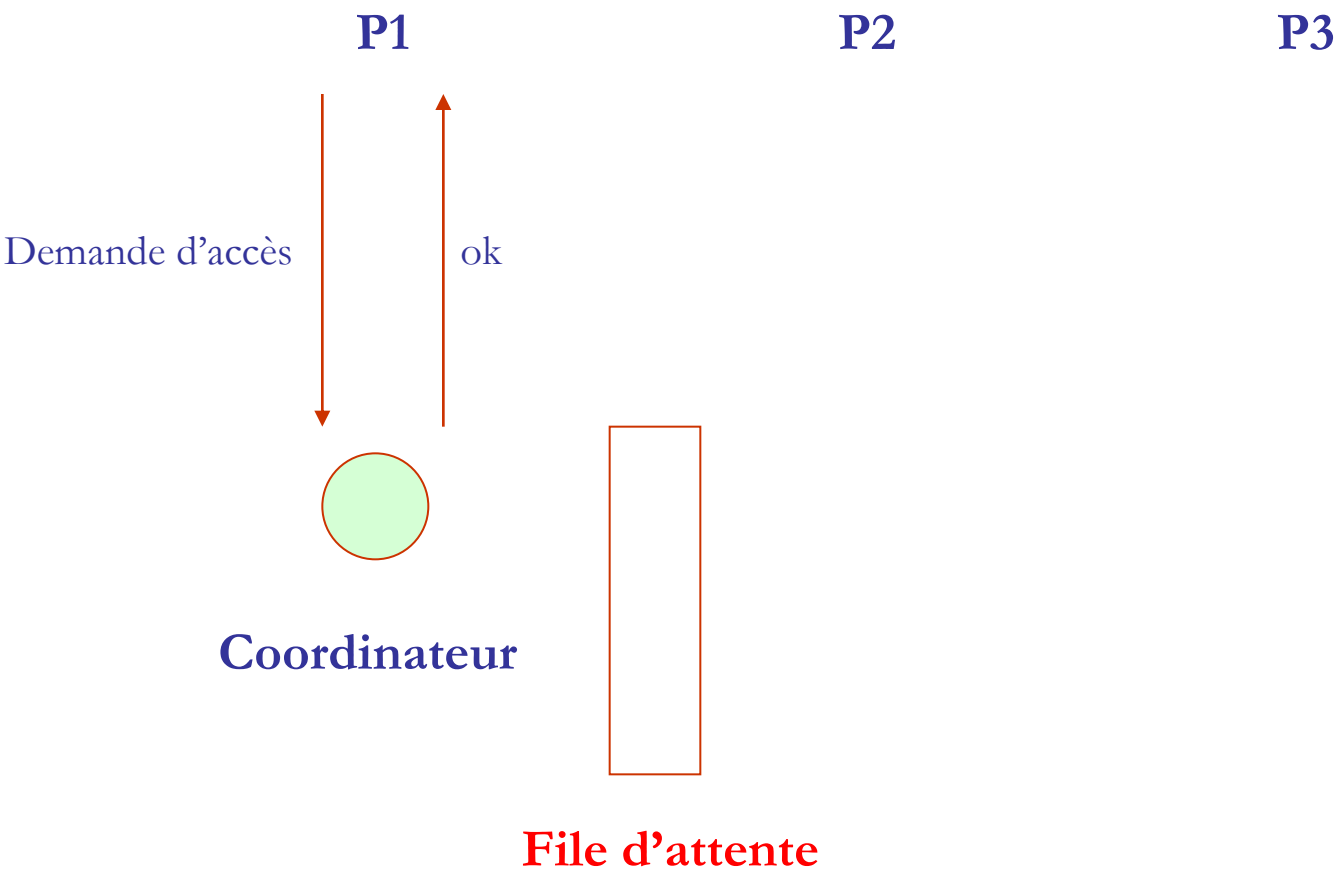
- **Plusieurs grandes familles de méthodes :**
  - **Contrôle par un coordinateur :** qui centralise les demandes d'accès à la ressource partagée.
  - **Contrôle par jeton :**
    - Un jeton circule entre les processus et donne l'accès à la ressource.
      - La gestion et l'affectation du jeton – ***et donc l'accès à la ressource*** – est faite par les processus entre eux.
    - Deux approches : jeton circulant en permanence ou affecté à la demande des processus.
  - **Contrôle par permission :**
    - Les processus s'autorisent mutuellement à accéder à la ressource.

Contrôle par coordinateur

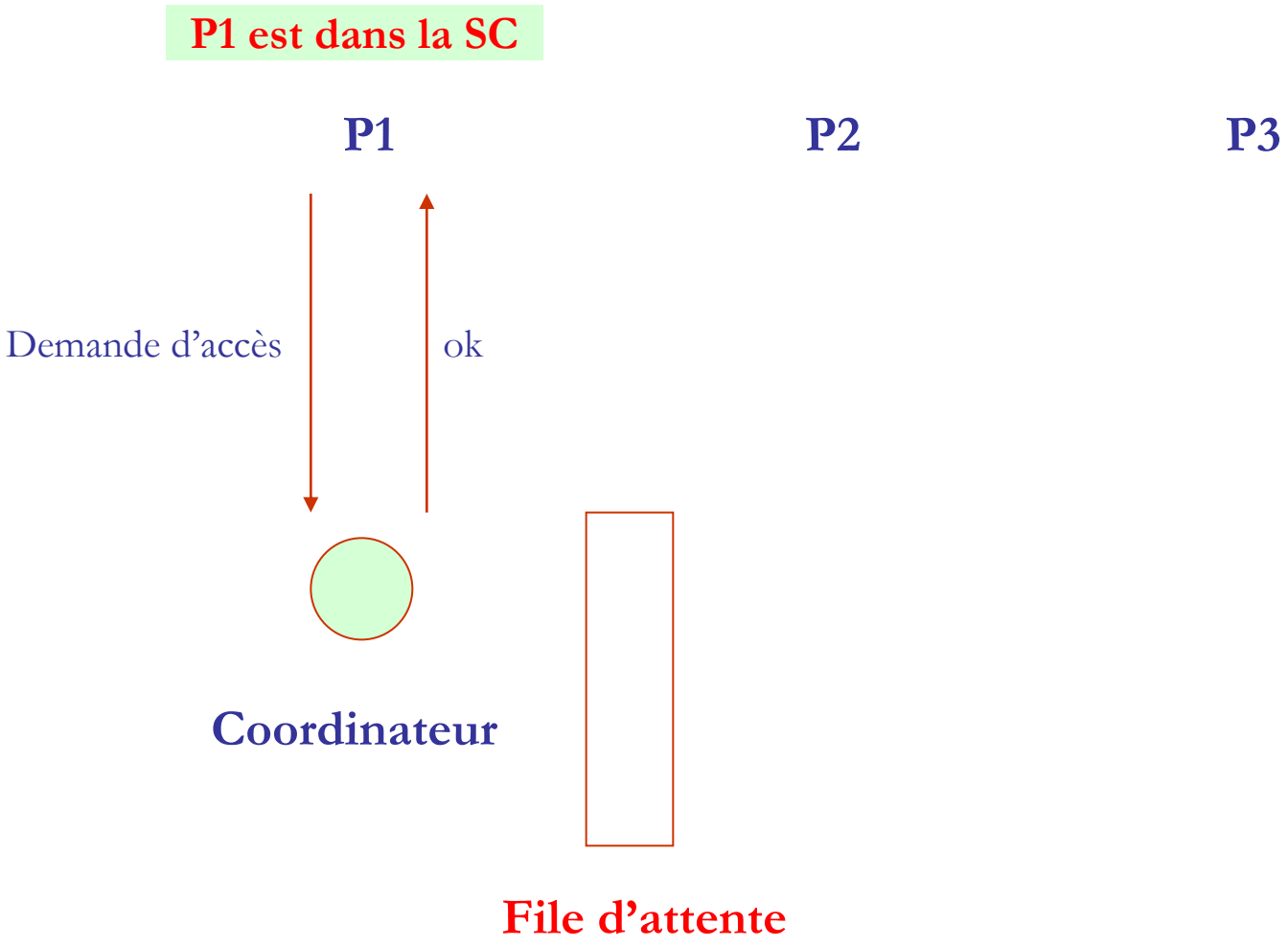
- **Principe général :** un coordinateur centralise et gère l'accès à la ressource.

- **Algorithme :**
  - Un processus voulant accéder à la ressource envoie une requête au coordinateur.
  - Quand le coordinateur lui envoie l'autorisation, il accède à la ressource.
    - Il informe le coordinateur quand il relâche la ressource.
  - Le coordinateur reçoit les demandes d'accès et envoie les autorisations d'accès aux processus demandeurs.
    - Avec une gestion FIFO : premier processus demandeur, premier autorisé à accéder à la ressource.

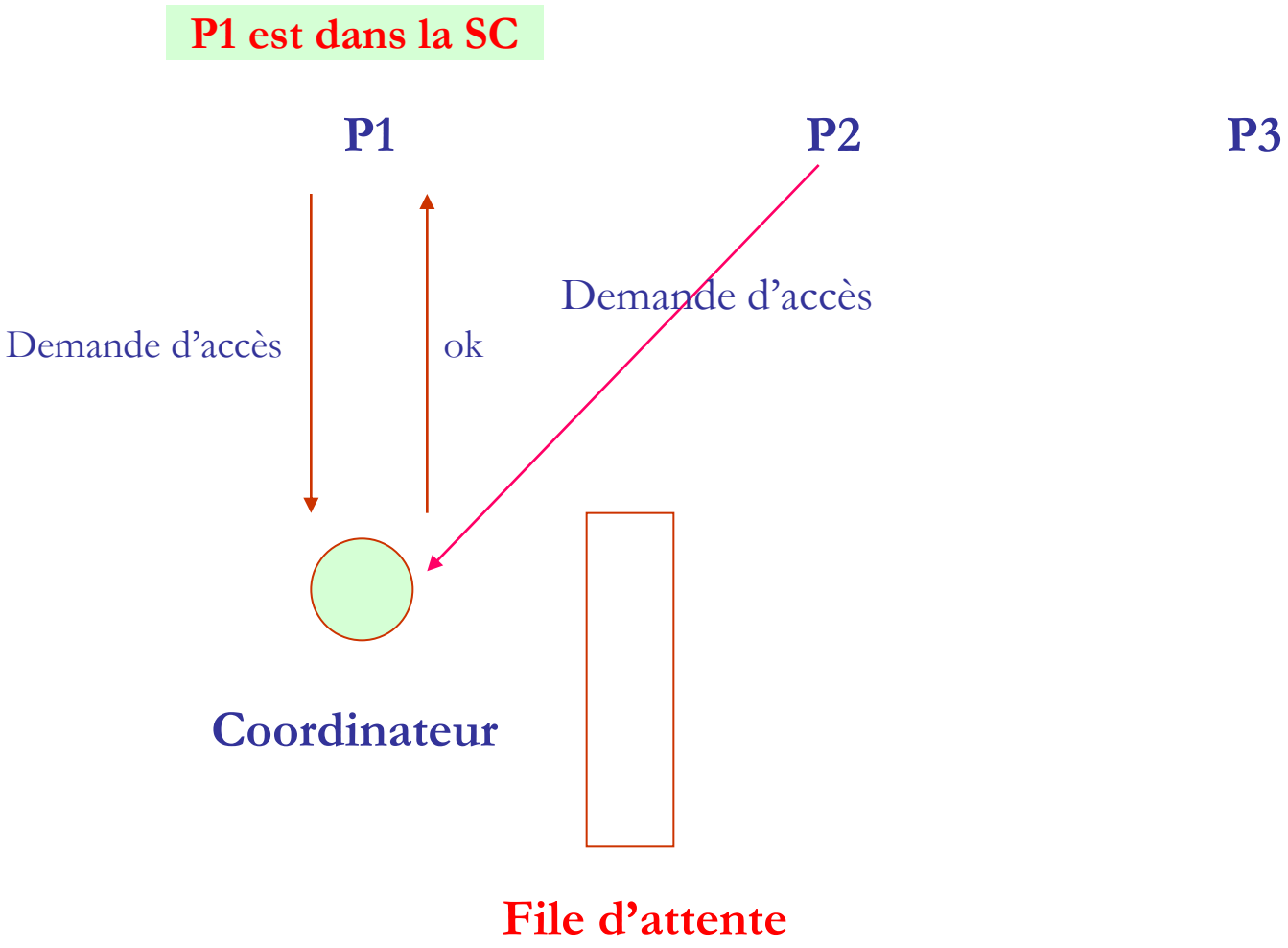
Contrôle par coordinateur



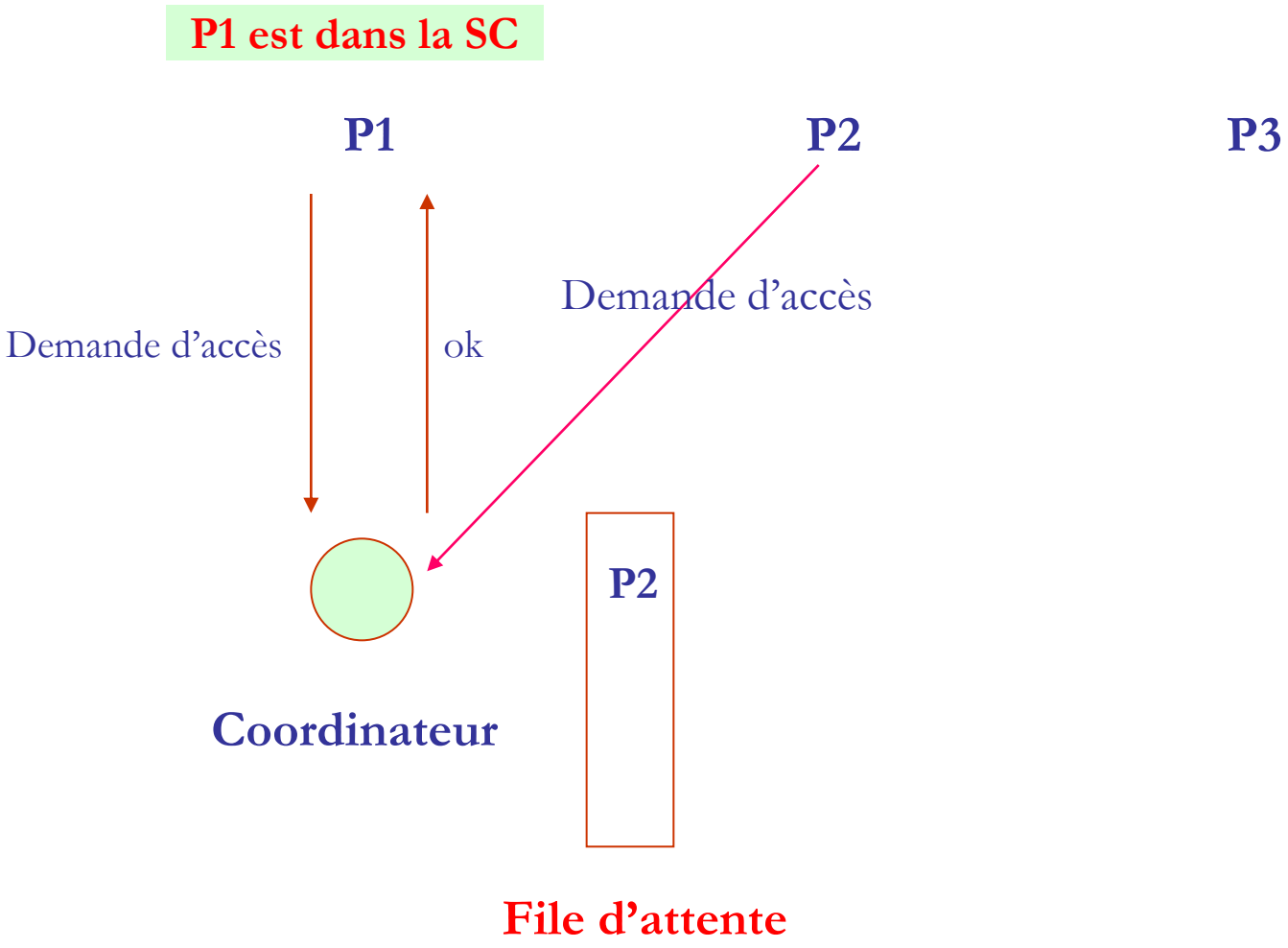
Contrôle par coordinateur



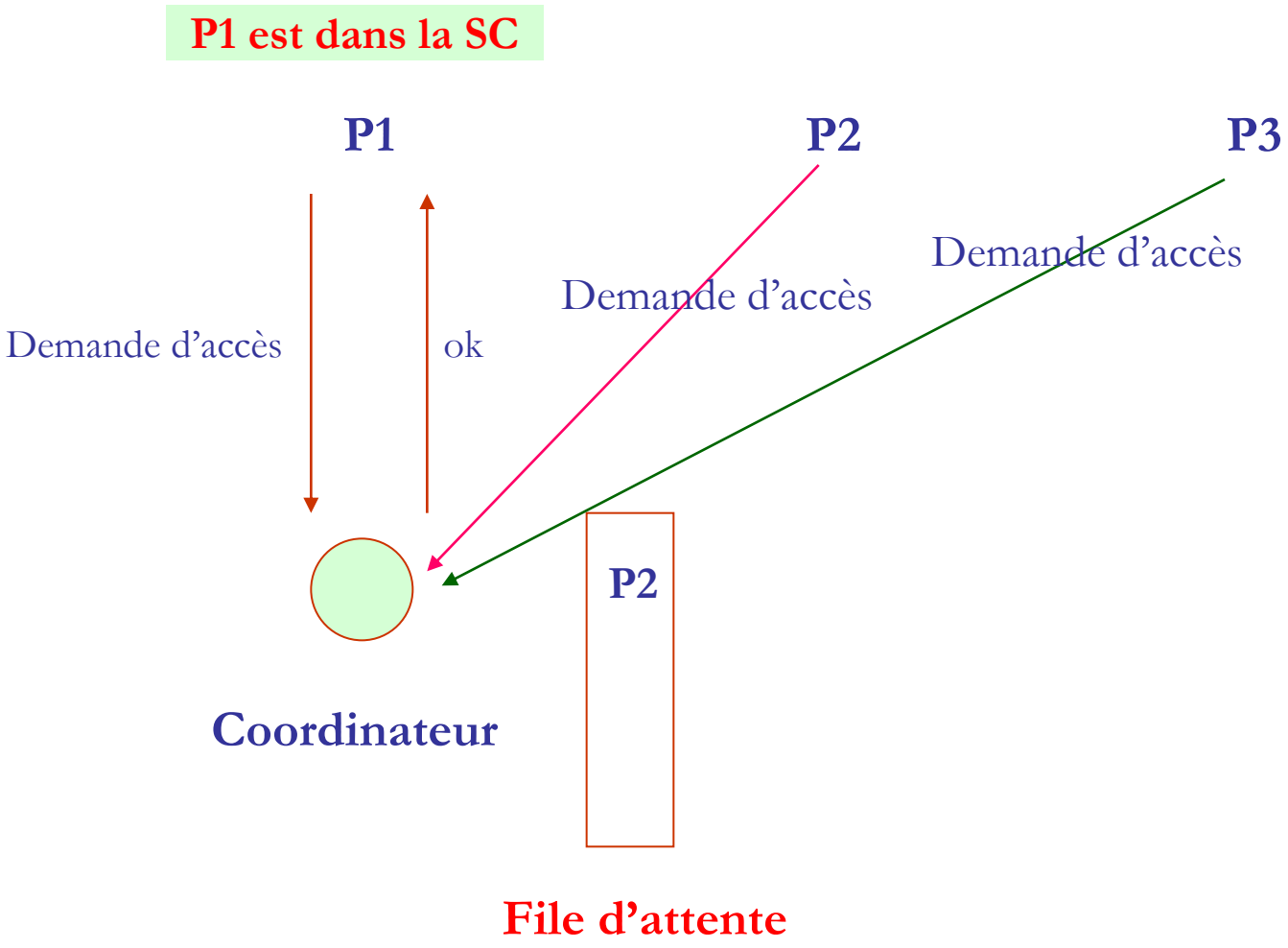
Contrôle par coordinateur



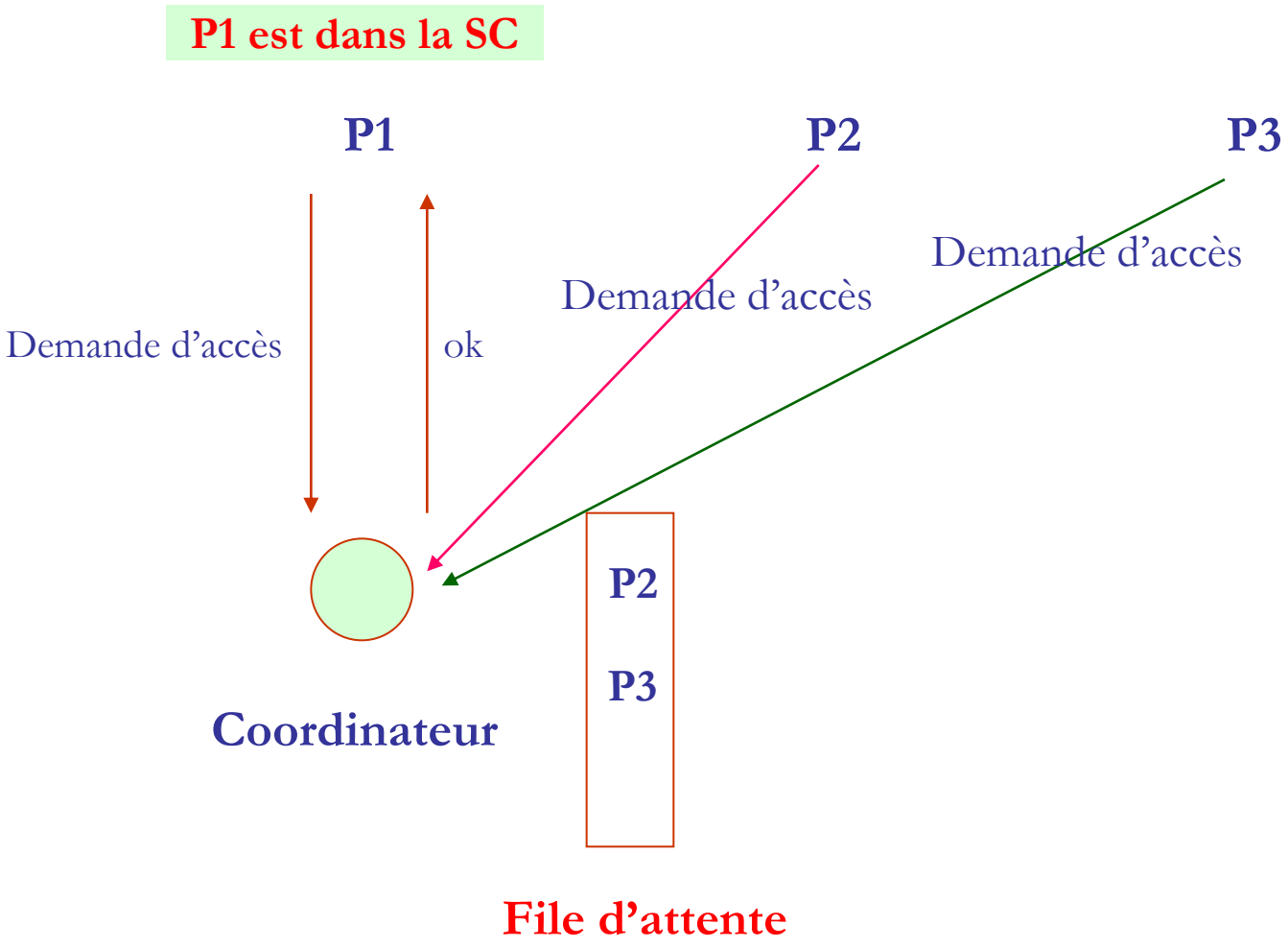
Contrôle par coordinateur



Contrôle par coordinateur

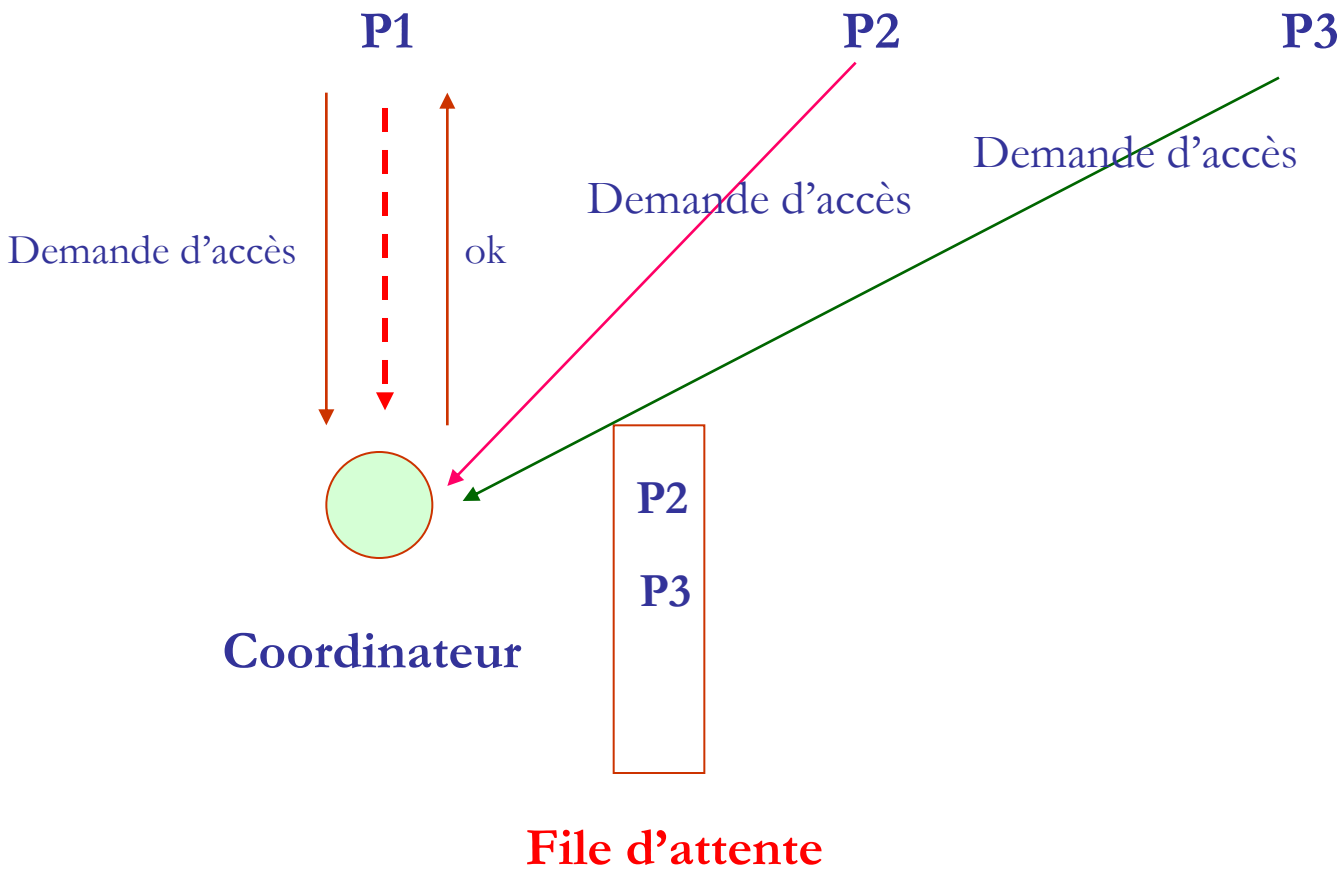


Contrôle par coordinateur

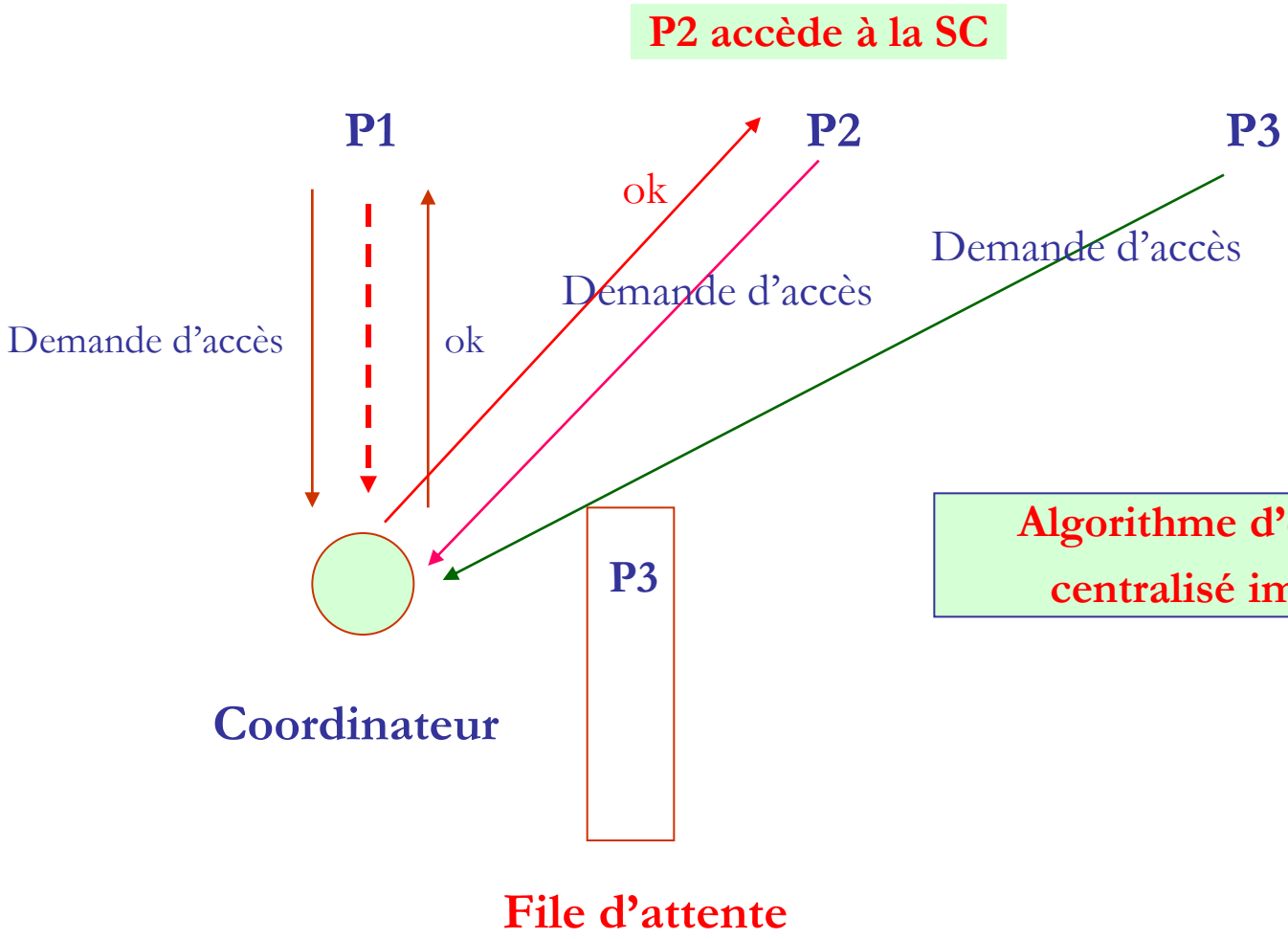


Contrôle par coordinateur

P1 libère la ressource



Contrôle par coordinateur



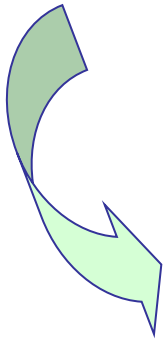
Algorithme d'exclusion mutuelle centralisé implanté en réparti

Contrôle par coordinateur

- **Avantages :**
  - **Très simple à mettre en œuvre : pour gérer la concurrence d'accès à la ressource.**
- **Inconvénients :**
  - Nécessite un élément particulier (coordinateur) pour gérer l'accès.
    - **Si le coordinateur tombe en panne ?**
    - **Potentiel point faible, goulot d'étranglement.**
    - Si un processus demande l'accès à la SC et ne reçoit pas de réponse : **ressource utilisée ou coordinateur en panne ?**

Contrôle par coordinateur

- **Avantages :**
  - **Très simple à mettre en œuvre : pour gérer la concurrence d'accès à la ressource.**
- **Inconvénients :**
  - Nécessite un élément particulier (coordinateur) pour gérer l'accès.
    - **Si le coordinateur tombe en panne ?**
    - **Potentiel point faible, goulot d'étranglement.**
    - Si un processus demande l'accès à la SC et ne reçoit pas de réponse : **ressource utilisée ou coordinateur en panne ?**



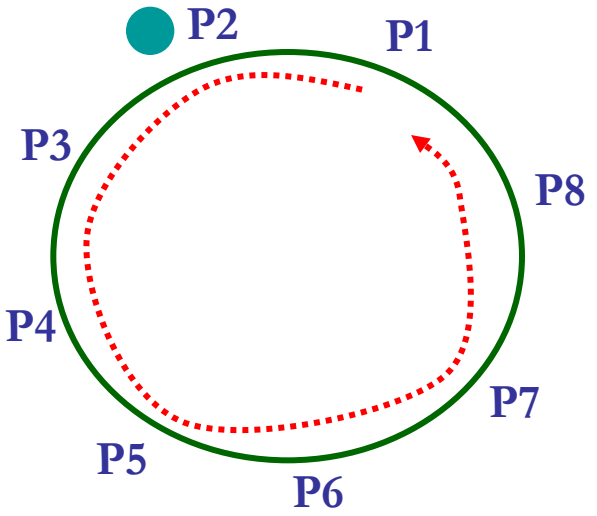
  - **Suppression du coordinateur :**
    - Via par exemple une méthode à jeton : le processus qui a le jeton peut accéder à la ressource.
    - La gestion et l'affectation du jeton est faite par les processus entre eux.
      - Pas de besoin de coordinateur centralisateur.

Méthode par jeton

- **Principe général :**
  - Un jeton unique circule entre tous les processus.
  - Le processus qui a le jeton est le seul qui peut accéder à la section critique.
- **Respect des propriétés :**
  - **Sûreté :** grâce au jeton unique.
  - **Vivacité :** l'algorithme doit assurer que le jeton circule bien entre tous les processus voulant accéder à la ressource.
- **Deux versions :**
  - **Anneau sur lequel circule le jeton en permanence.**
  - **Jeton affecté à la demande des processus.**

Algorithme de « Gérard Le Lann » en 1977

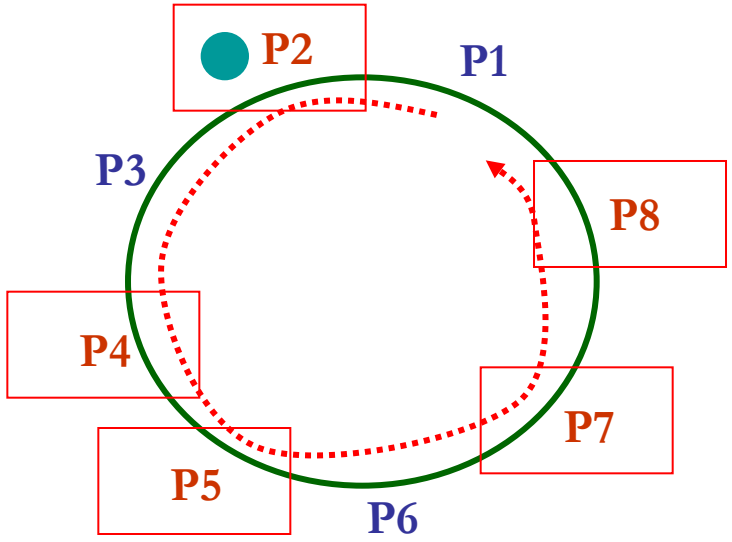
- Un jeton unique circule en permanence entre les processus via une **topologie en anneau.**



- **Quand un processus reçoit le jeton :**
  - S'il est dans l'état *dehors*, **il passe le jeton à son voisin.**
  - S'il est dans l'état *demandeur* : **il passe dans l'état *dedans* et accède à la ressource.**
    - Quand le processus quitte l'état *dedans*, **il passe le jeton à son voisin.**
- **Respect des propriétés :**
  - **Sûreté :** via le jeton unique qui autorise l'accès à la ressource.
  - **Vivacité :** si un processus lâche le jeton (la ressource) en un temps fini et que tous les processus appartiennent à l'anneau.

Algorithme de « Gérard Le Lann » en 1977

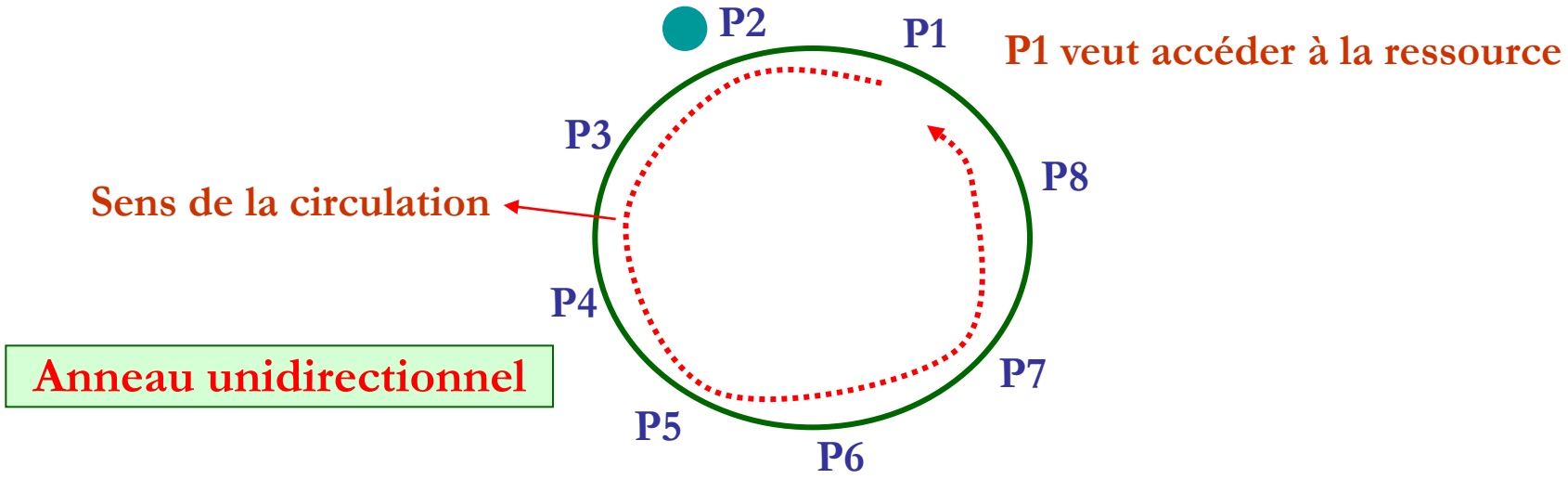
- **Avantages :**
  - Très simple à mettre en œuvre.
  - Intéressant si nombreux processus demandeurs de la ressource.
    - Jeton arrivera rapidement à un processus demandeur.
    - Équitable en terme de nombre d'accès et de temps d'attente.
      - Aucun processus n'est privilégié.



P2, P4, P5, P7 et P8 sont demandeurs de la ressource.

Algorithme de « Gérard Le Lann » en 1977

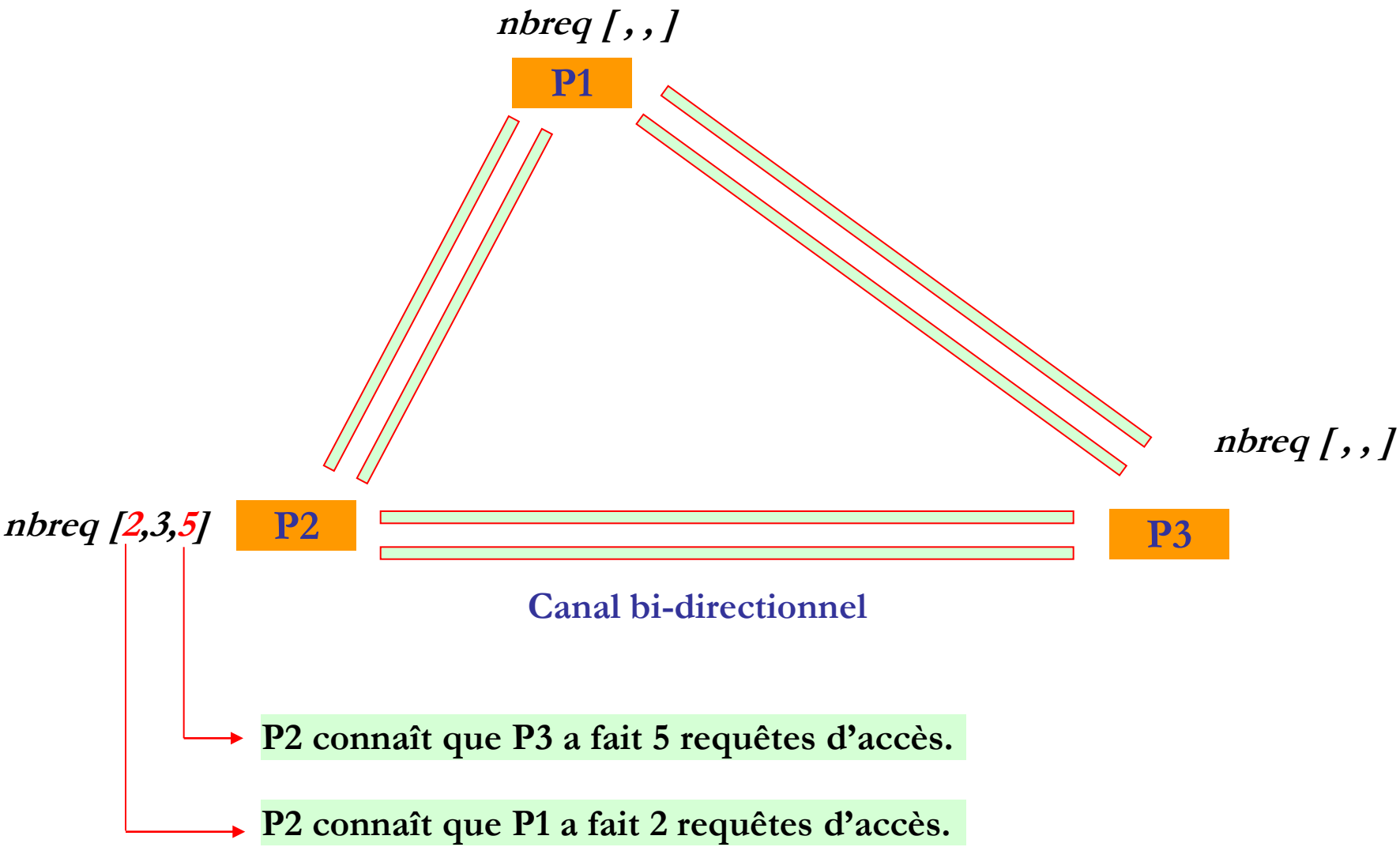
- ***Inconvénients :***
  - Nécessite des échanges de messages (pour faire circuler le jeton) même si aucun processus ne veut accéder à la ressource.
  - Temps d'accès à la ressource peut être potentiellement relativement long.
    - Si le processus *i+1* a le jeton et que le processus *i* veut accéder à la ressource et est le seul à vouloir y accéder, il faut quand même attendre que le jeton fasse tout le tour de l'anneau.



Algorithme de « Ricart et Agrawala » en 1983

- **Variante de la méthode du jeton :** au lieu d'attendre le jeton, un processus diffuse à tous le fait qu'il veut obtenir le jeton.
  - Le processus qui a le jeton sait alors à qui il peut l'envoyer.
  - Évite les attentes et les circulations inutiles du jeton.
- **Algorithme de *« Ricart et Agrawala »* en 1983 :**
  - Soit *N* processus avec un canal bi-directionnel entre chaque processus.
    - Canaux fiables mais pas forcément FIFO.
  - Localement, un processus *Pi* possède un tableau ***nbreq***, de taille *N*.
    - Pour *Pi*, ***nbreq [j]*** est le nombre de requêtes d'accès que le processus *Pj* a fait et que *Pi* connaît (par principe il les connaît toutes).

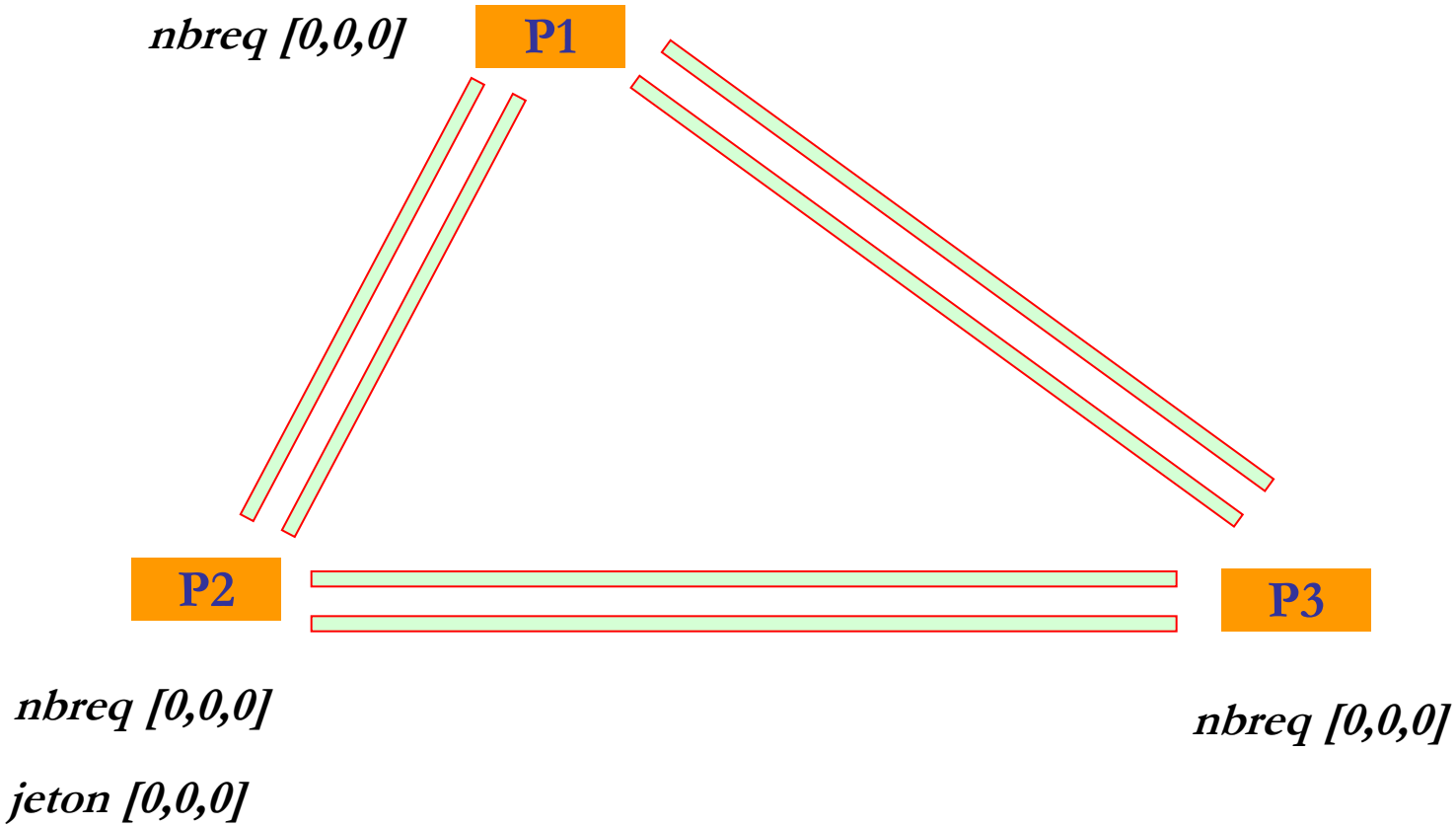
Algorithme de « Ricart et Agrawala » en 1983



Algorithme de « Ricart et Agrawala » en 1983

- **Algorithme de *« Ricart et Agrawala »* :**
  - Le jeton est un tableau de taille *N*.
    - *jeton [i]* est le nombre de fois où le processus *Pi* a accédé à la ressource.
    - La case *i* de *jeton* n'est modifiée que par *Pi* quand celui-ci libère la ressource.
  - **Initialisation :**
    - Pour tous les processus *Pi* : **∀ j ∈ [1 .. N] : nbreq [j] = 0.**
    - Jeton : **∀ j ∈ [1 .. N] : jeton [j] = 0.**
      - Un processus donné possède le jeton au départ
  - Quand un processus veut accéder à la ressource et n'a pas le jeton.
    - Envoie un message de requête à tous les processus.

Algorithme de « Ricart et Agrawala » en 1983

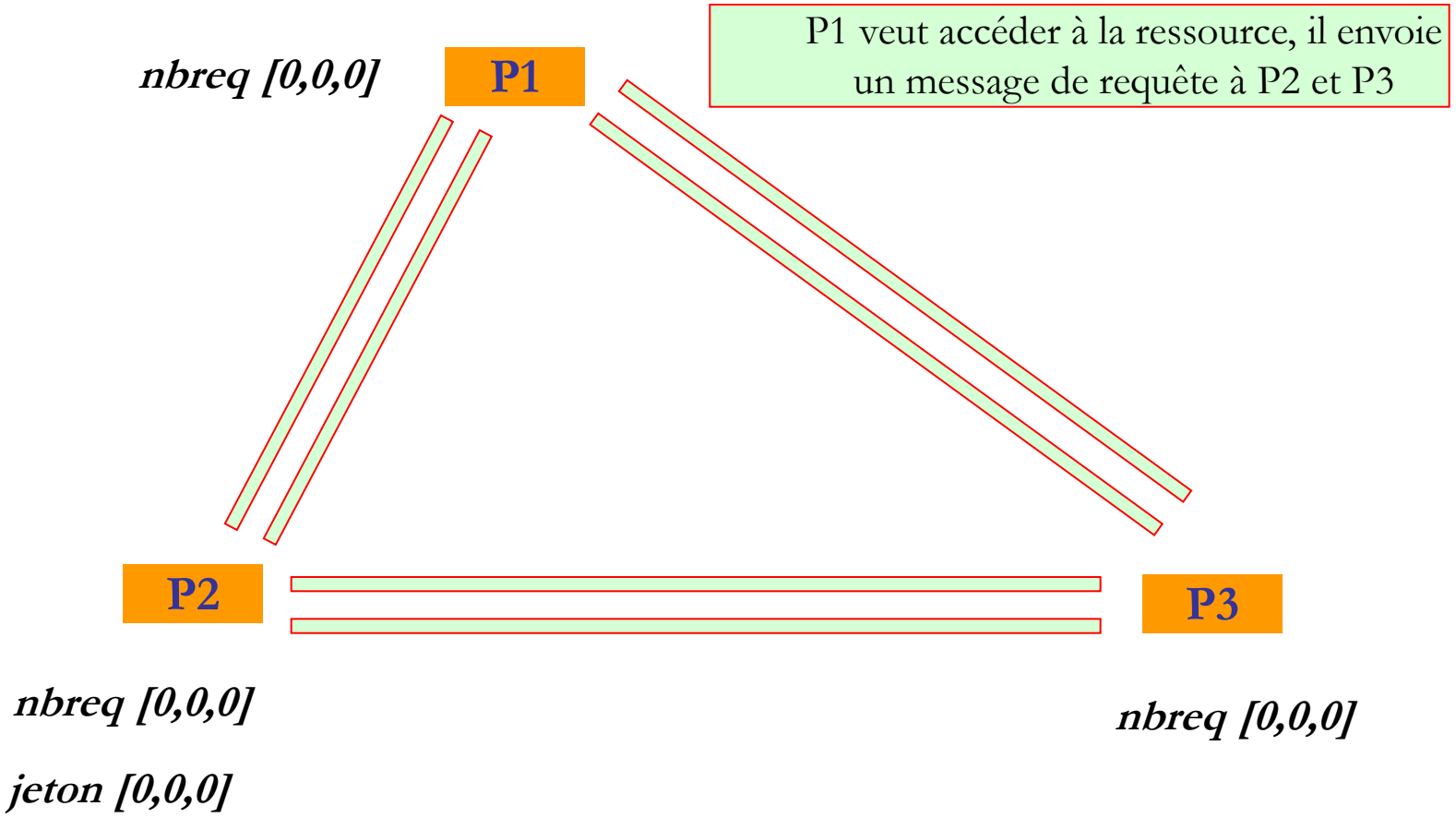


Initialisation

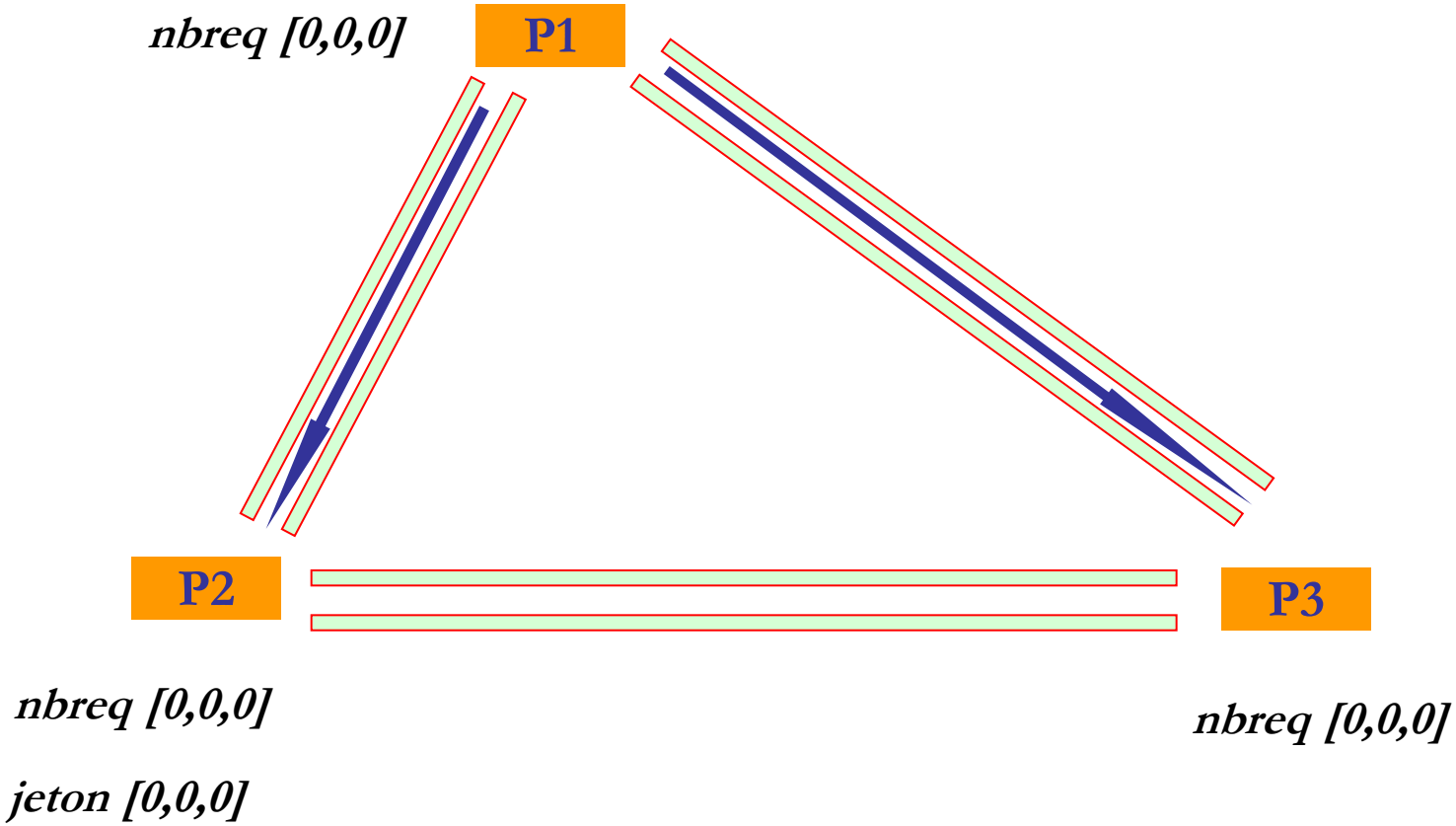
Algorithme de « Ricart et Agrawala » en 1983

- **Algorithme de *« Ricart et Agrawala »* :**
  - **Quand processus *Pj* reçoit un message de requête venant de *Pi* *(demandeur de ressource).***
    - *Pj* modifie son *nbreq* localement : **nbreq [i] = nbreq [i] + 1.**
      - *Pj* mémorise que *Pi* a demandé à avoir la ressource.
    - Si *Pj* possède le jeton et est dans l'état *dehors*.
      - *Pj* envoie le jeton à *Pi*.
  - **Quand processus récupère le jeton.**
    - Il accède à la ressource (passe dans l'état *dedans*).
  - **Quand *Pi* libère la ressource (passe dans l'état *dehors*).**
    - Met à jour le jeton : **jeton [i] = jeton [i] + 1.**
    - **Parcourt *nbreq* pour trouver un *j* tel que : nbreq [j] > jeton [j].**
      - Une demande d'accès à la ressource de *Pj* n'a pas encore été satisfaite : *Pi* envoie le jeton à *Pj*.
    - **Si aucun processus n'attend le jeton : *Pi* le garde.**

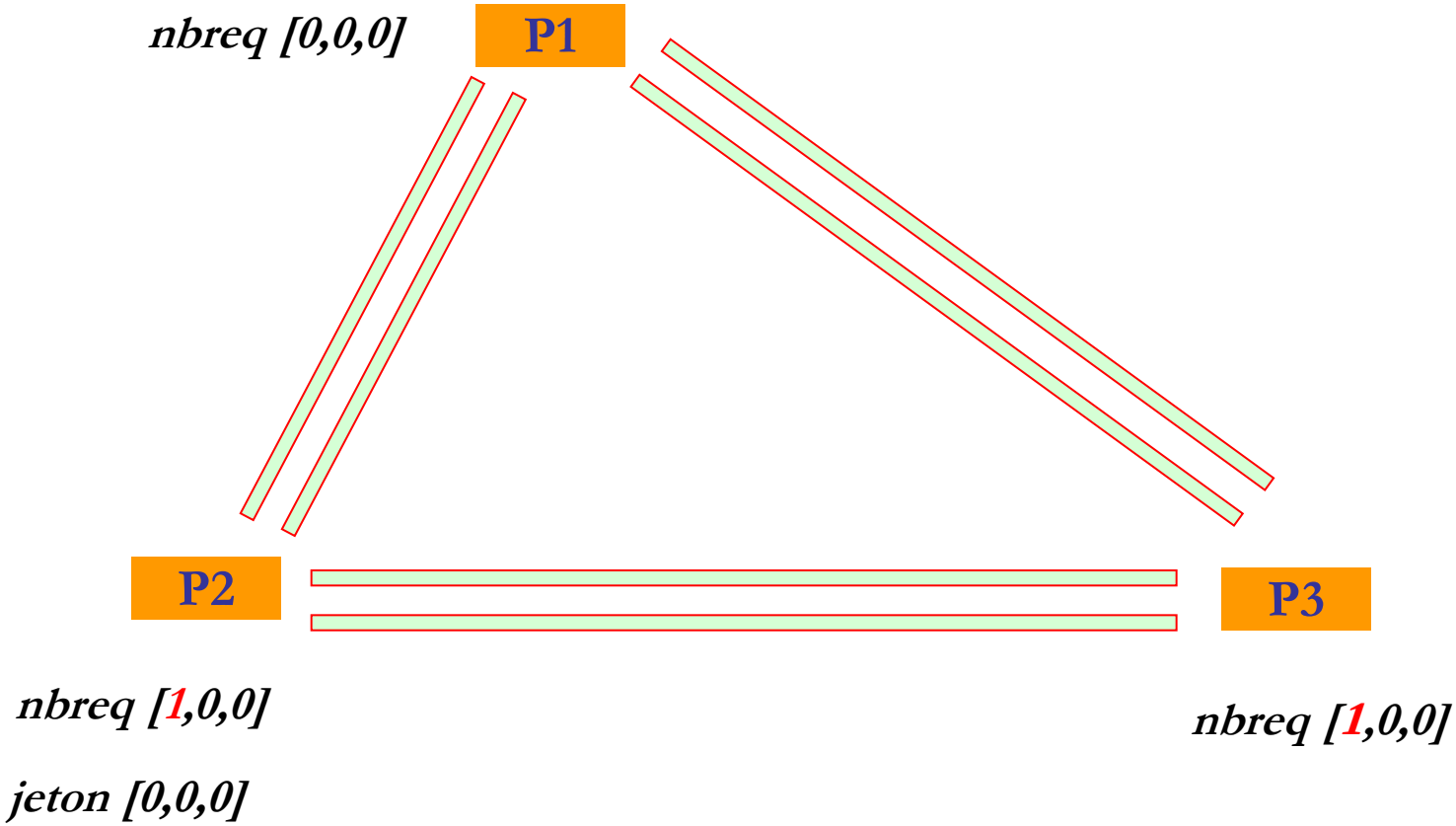
Algorithme de « Ricart et Agrawala » en 1983



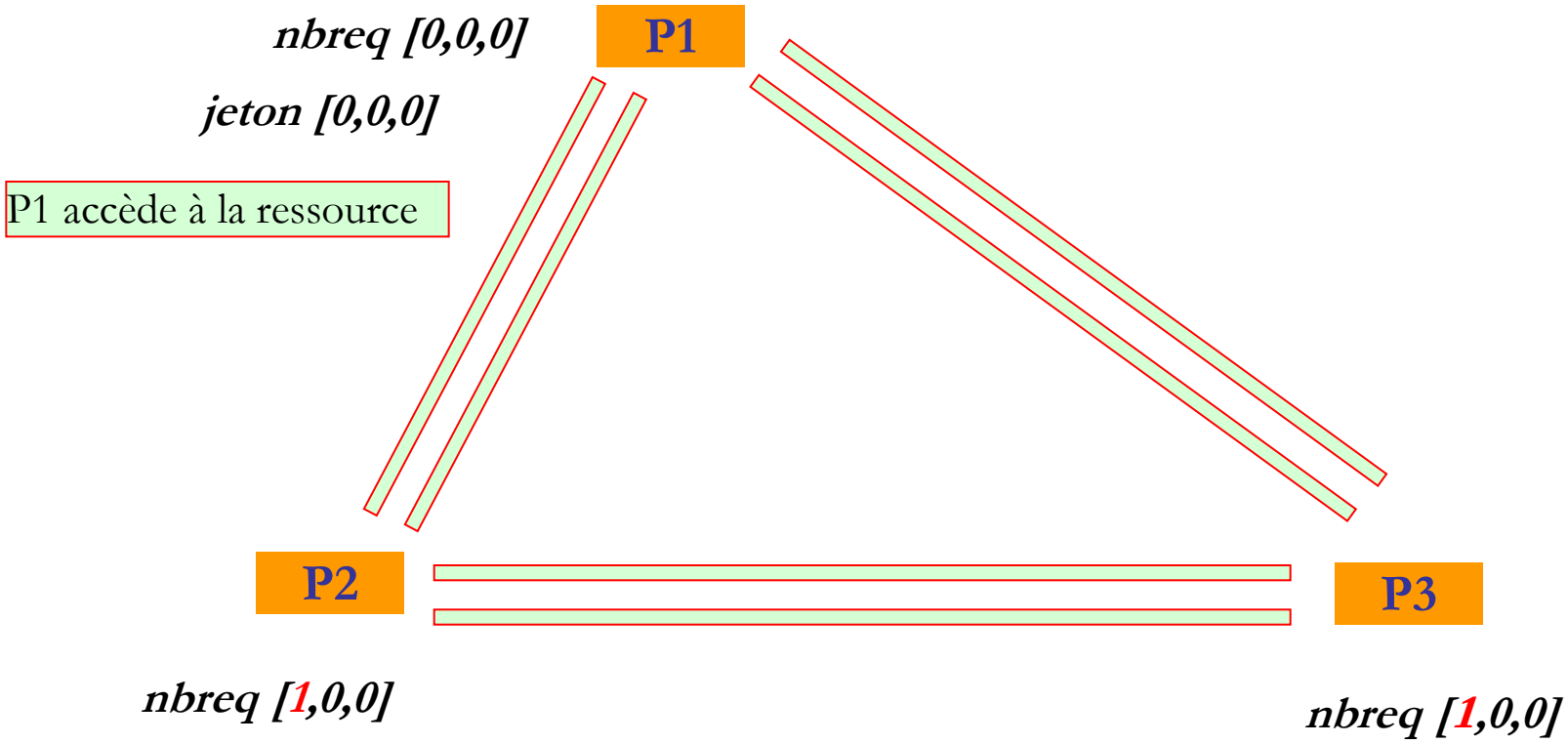
Algorithme de « Ricart et Agrawala » en 1983



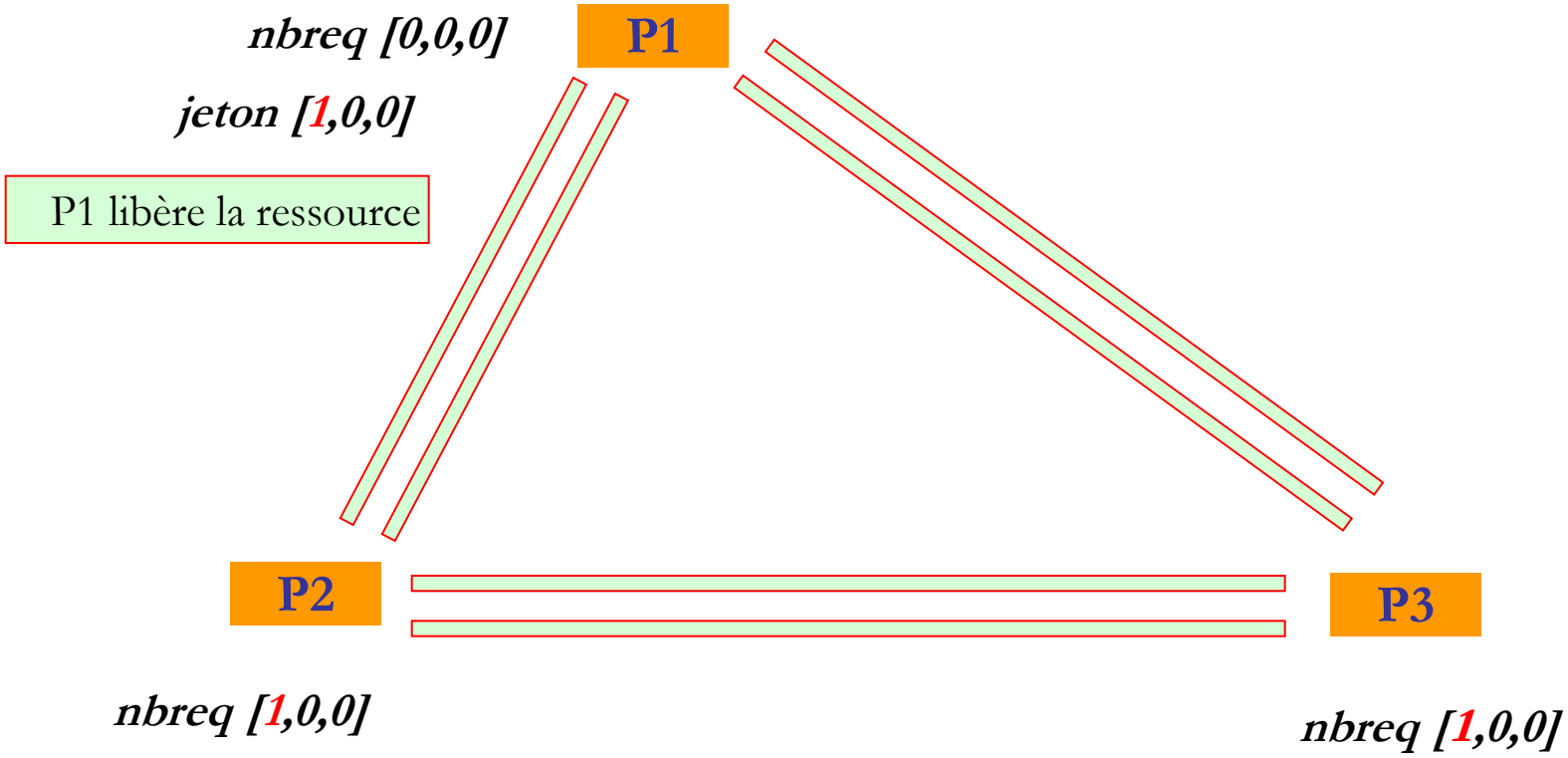
Algorithme de « Ricart et Agrawala » en 1983



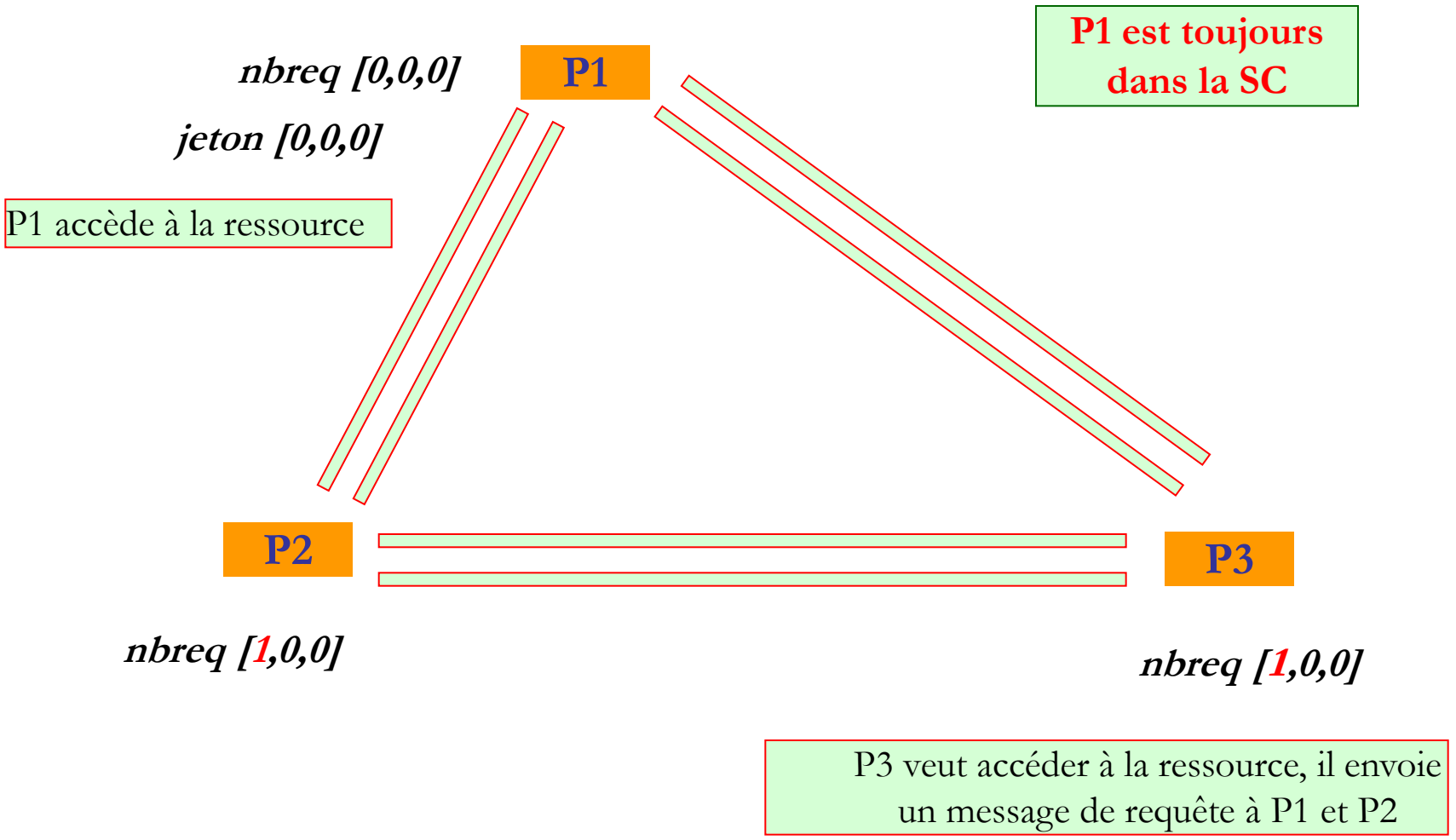
Algorithme de « Ricart et Agrawala » en 1983



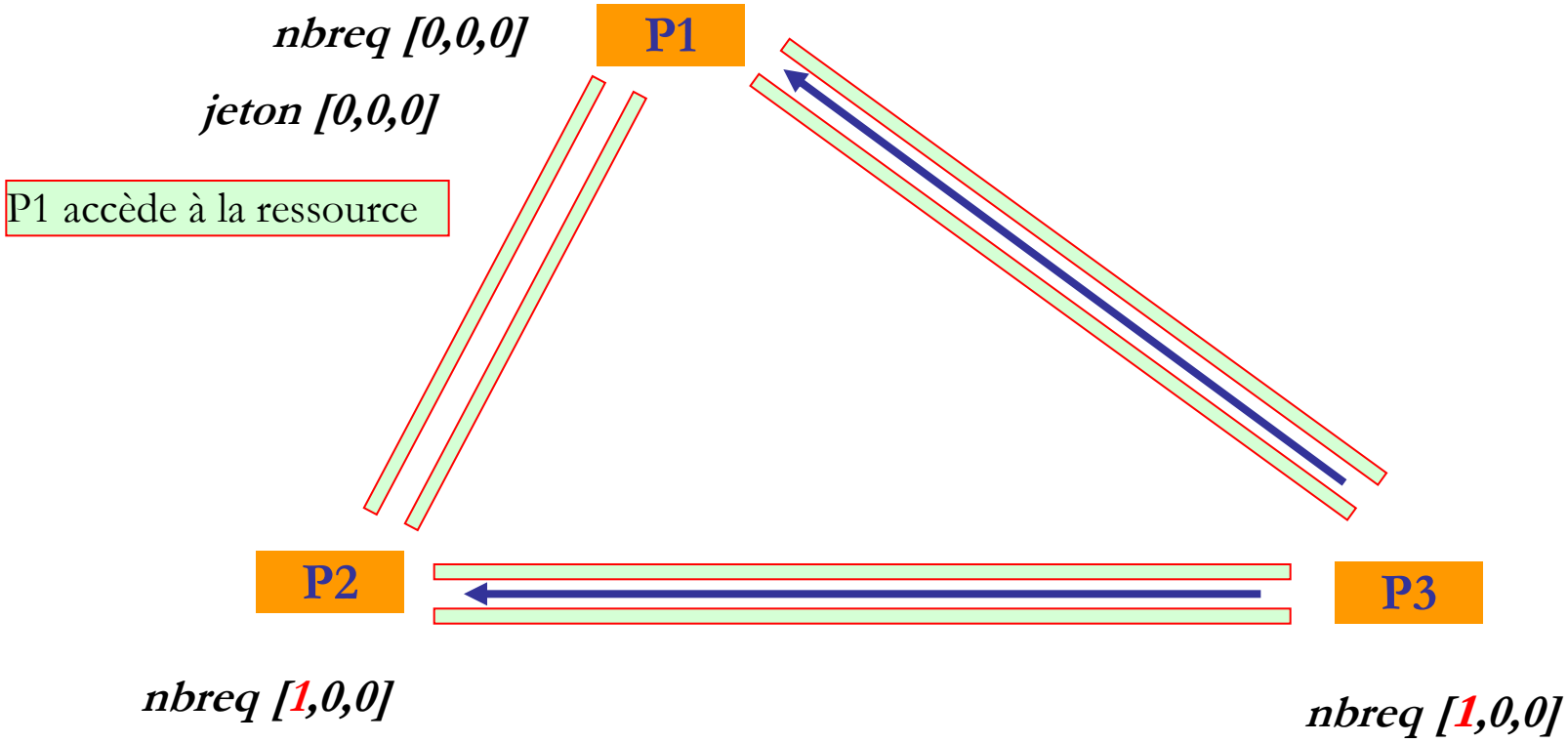
Algorithme de « Ricart et Agrawala » en 1983



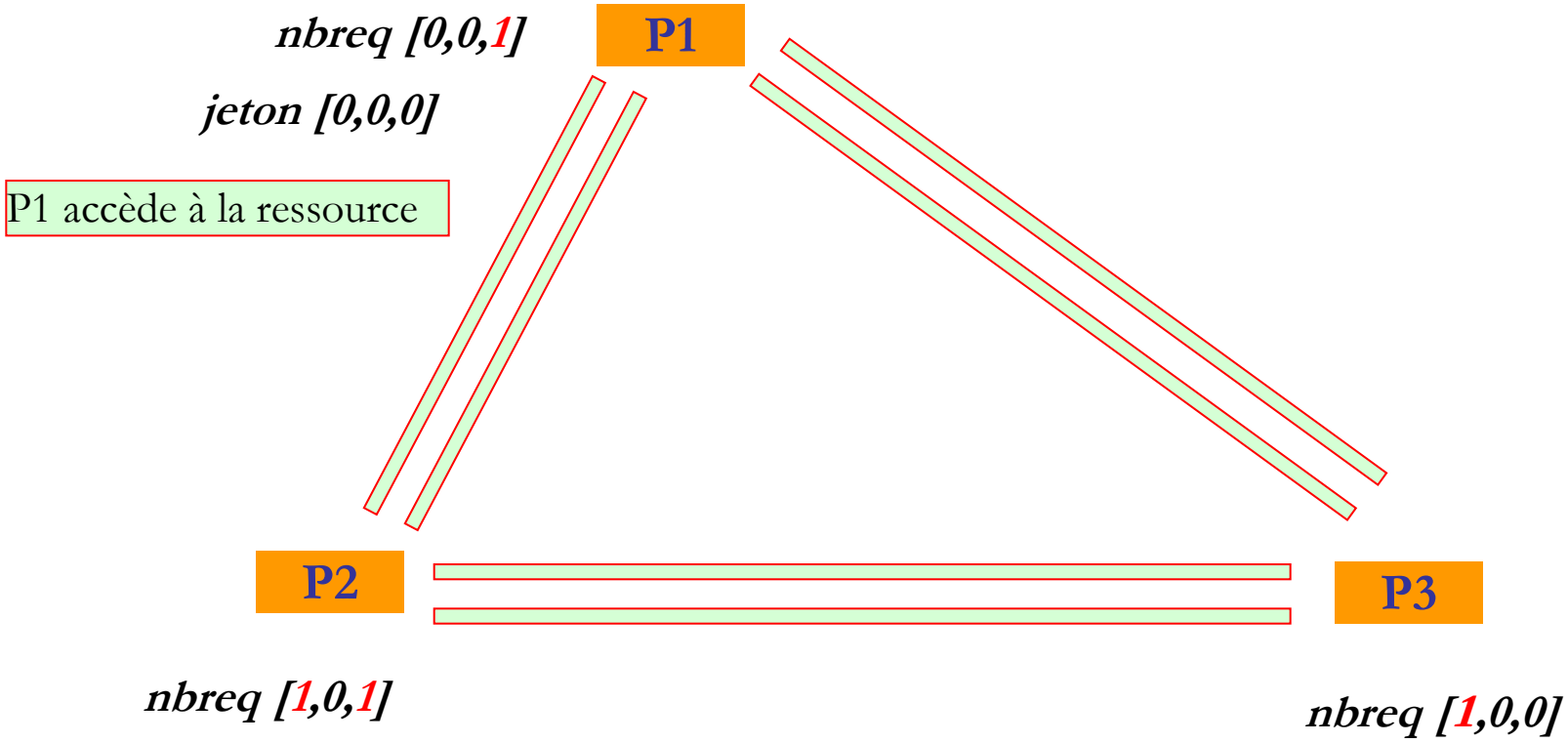
Algorithme de « Ricart et Agrawala » en 1983



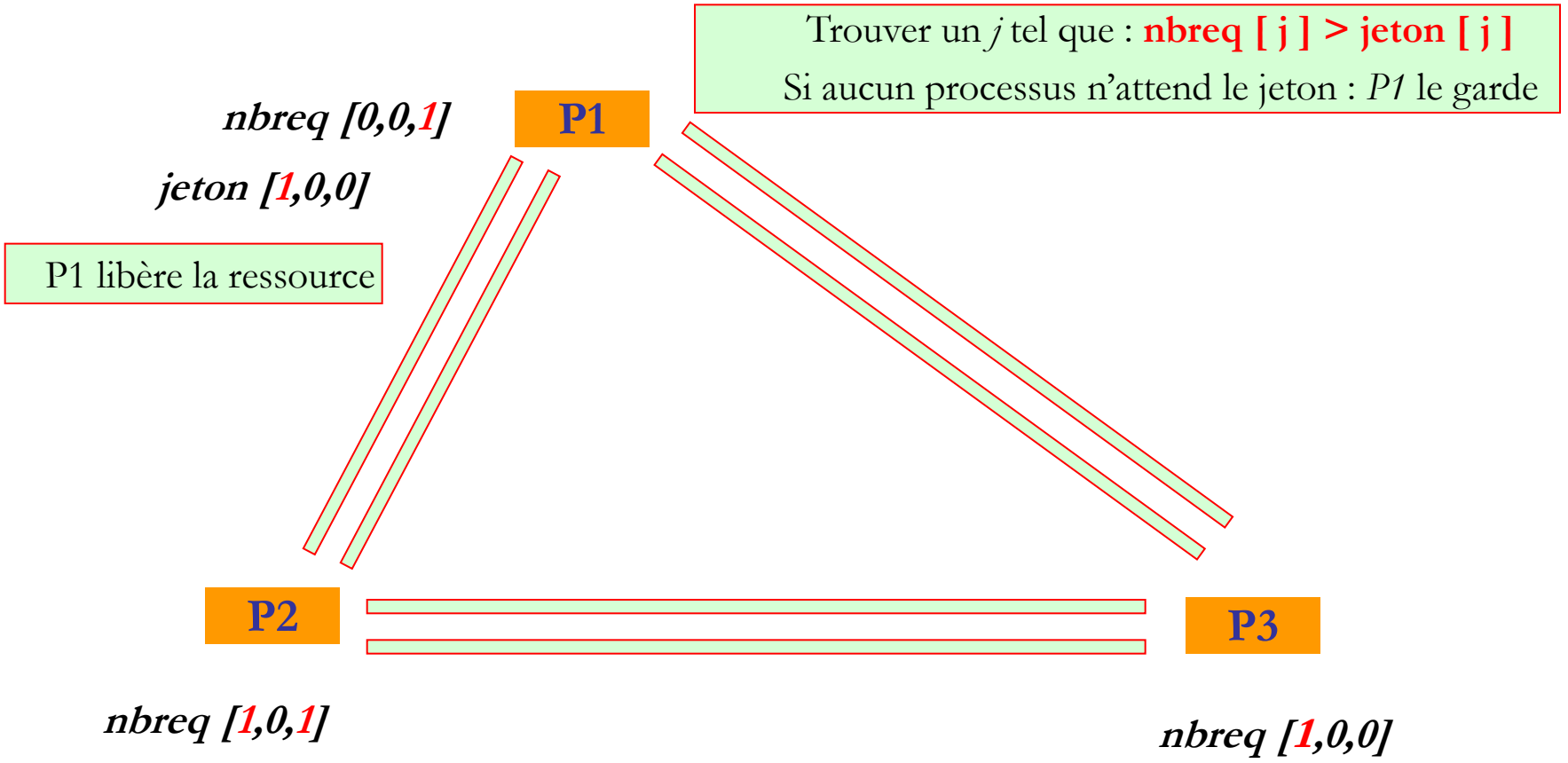
Algorithme de « Ricart et Agrawala » en 1983



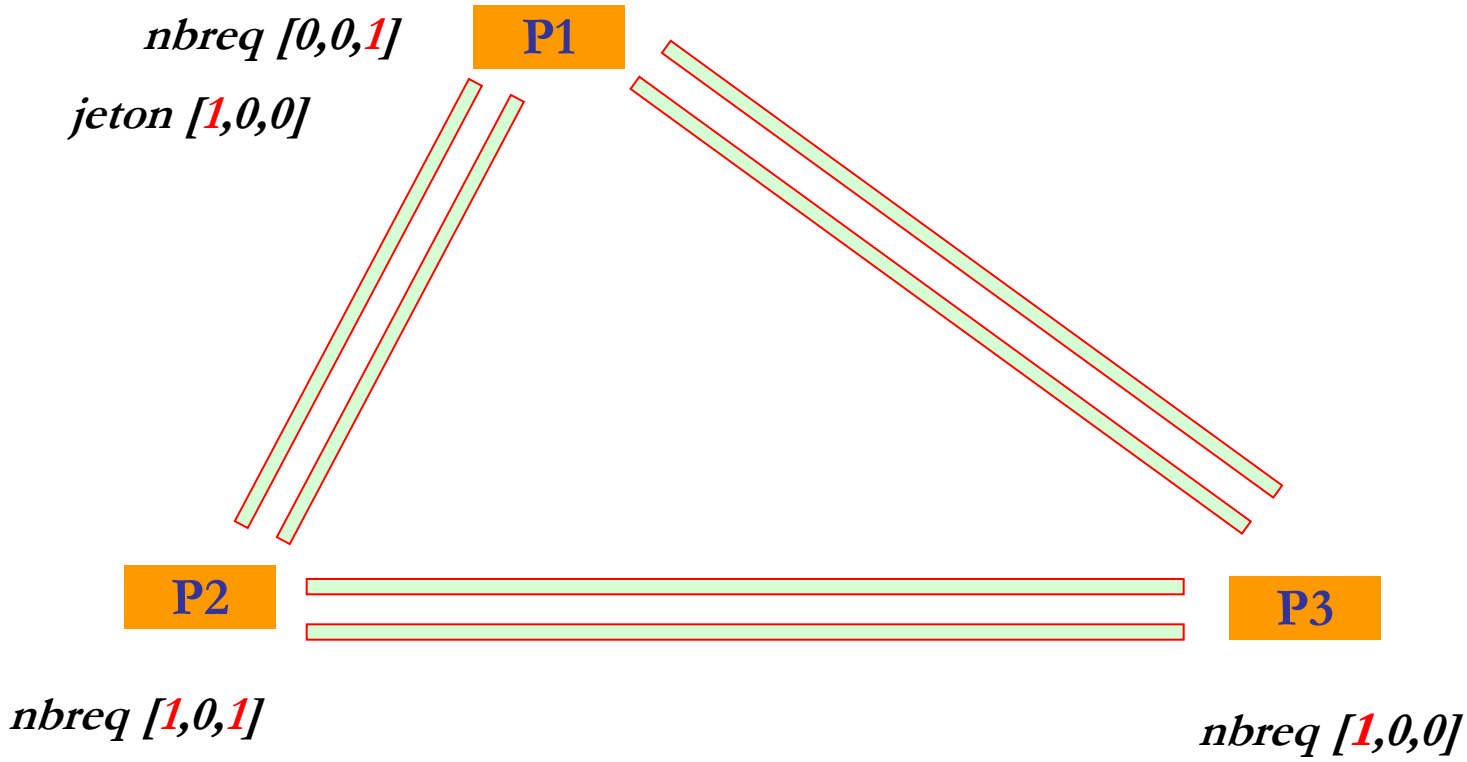
Algorithme de « Ricart et Agrawala » en 1983



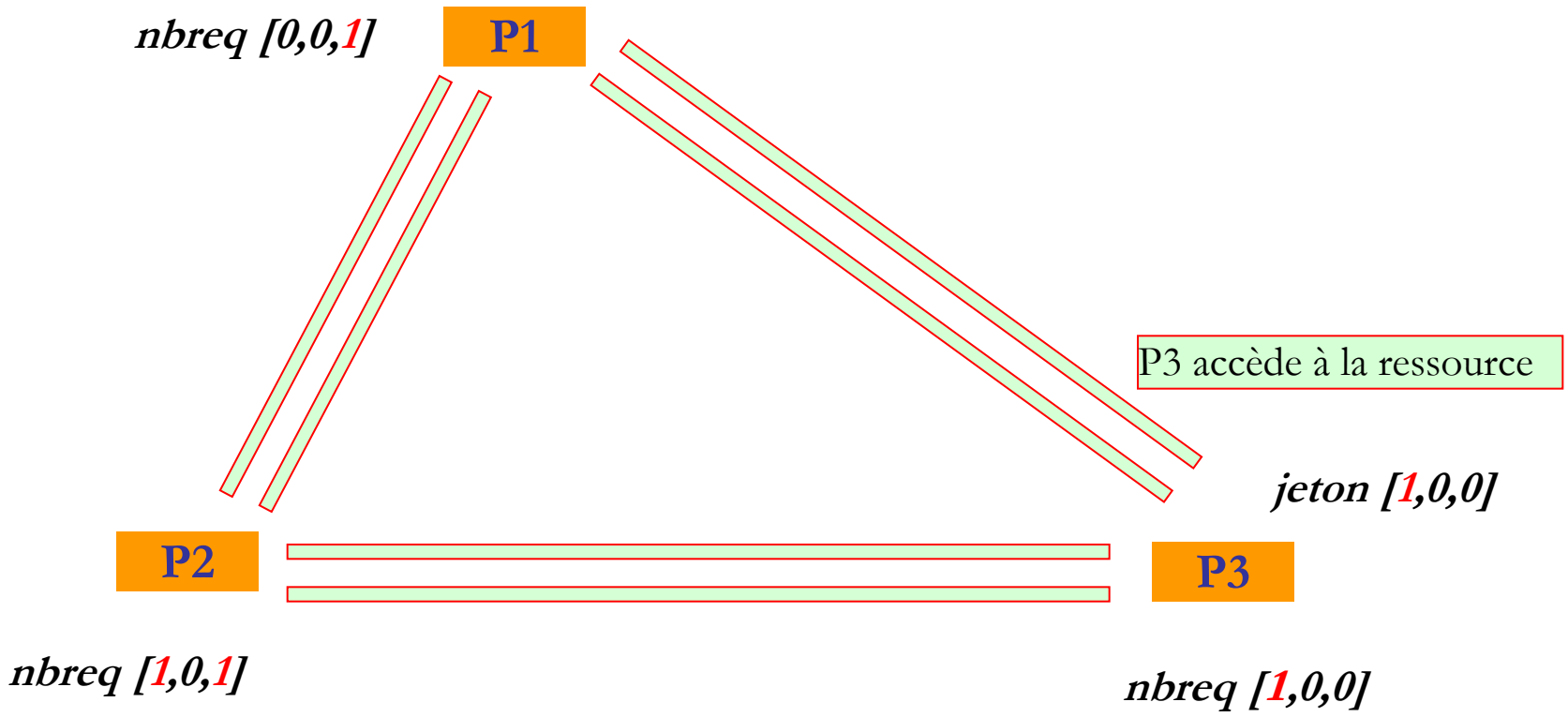
Algorithme de « Ricart et Agrawala » en 1983



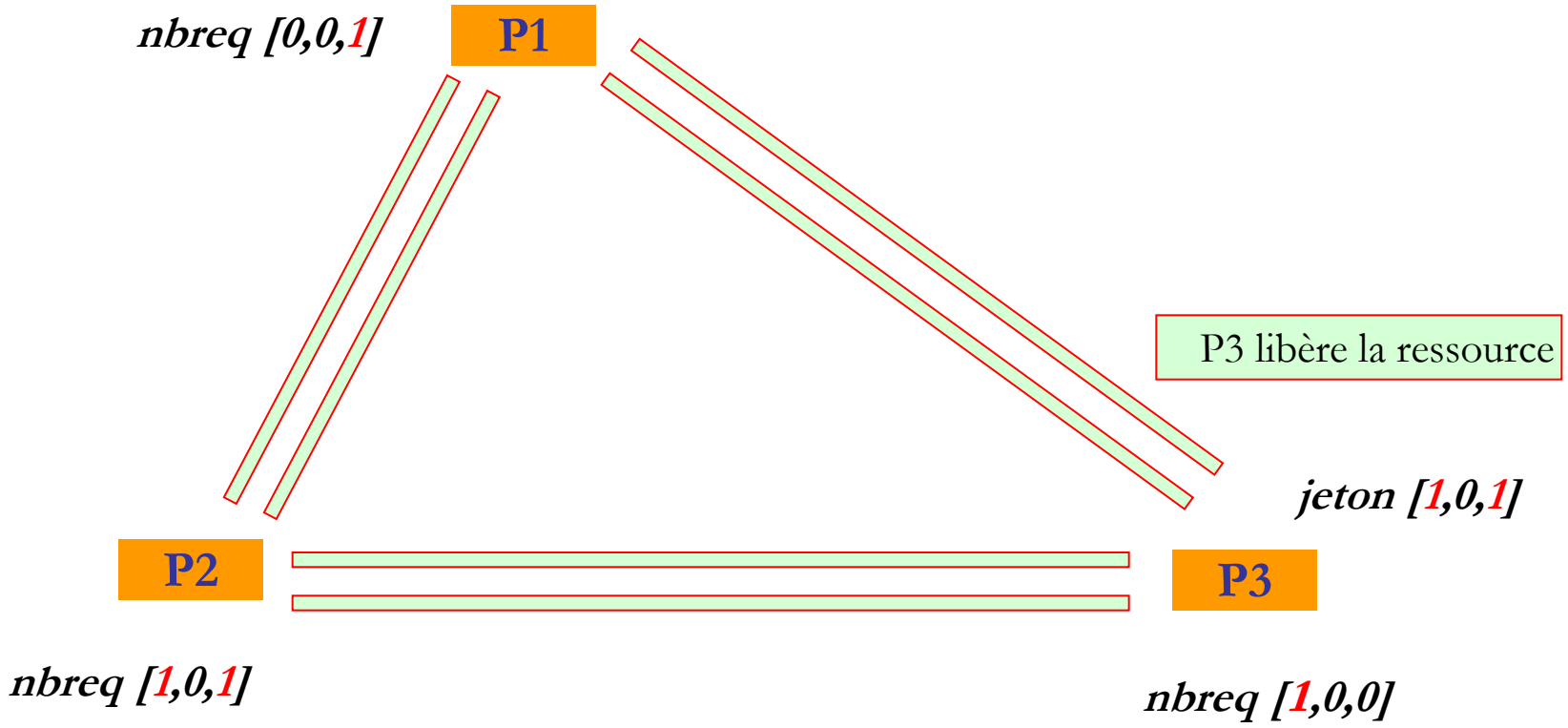
Algorithme de « Ricart et Agrawala » en 1983



Algorithme de « Ricart et Agrawala » en 1983



Algorithme de « Ricart et Agrawala » en 1983



Algorithme de « Ricart et Agrawala » en 1983

- **Respect des propriétés :**
  - **Sûreté :** seul le processus ayant le jeton accède à la ressource.
  - **Vivacité :** assurée **si les processus distribuent équitablement le jeton aux autres processus.**
    - Méthode de choix du processus qui va récupérer le jeton lorsque l'on sort de l'état dedans.
      - ***Pi* parcourt *nbreq* à partir de l'indice i+1 jusqu'à *N* puis continue de 1 à i-1.**
      - Évite que par exemple tous les processus avec un petit identificateur soient servis systématiquement en premier.

Méthodes par permission

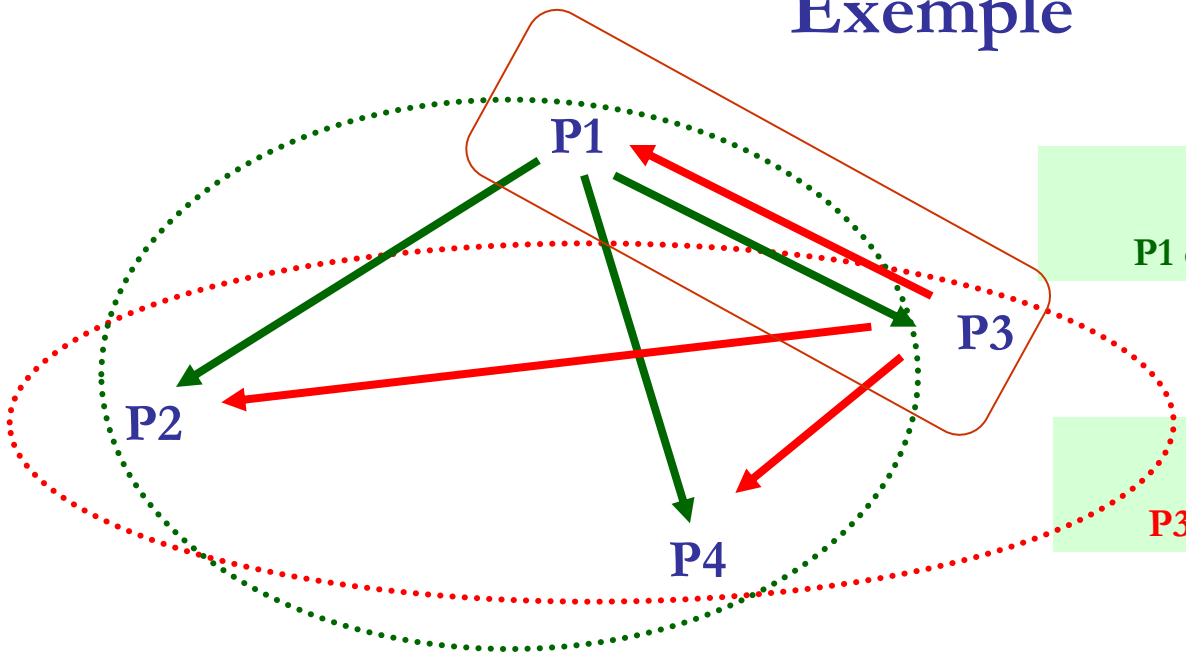
- **Principe :**
  - Un processus doit avoir l'autorisation des autres processus pour accéder à la ressource (un processus demande l'autorisation à un sous-ensemble donné de tous les processus).
- **Deux modes :**
  - **Permission individuelle :** chaque processus demande l'autorisation à tous les autres (sauf lui par principe).
    - Un processus peut donner sa permission à plusieurs autres à la fois.

- ***Permission par arbitre : Algorithme de Maekawa en 1985.***
  - *Non traité dans ce cours*

Permission individuelle : Algorithme de Ricart et Agrawala en 1981

- **Permission individuelle :** chaque processus demande l'autorisation à tous les autres (sauf lui par principe).
  - Liste des processus à interroger par le processus *Pi* pour accéder à la ressource : **Ri = {P1, ... , PN} – {Pi}.**
- Se base sur une horloge logique ***(Lamport)*** pour garantir le bon fonctionnement de l'algorithme.
  - **Ordonnancement** des demandes d'accès à la ressource.
  - Si un processus ayant fait une demande d'accès reçoit une demande d'un autre processus avec une date antérieure à la sienne (<), il donnera son autorisation à l'autre processus.

Exemple



R1 = {P2, P3, P4}

P1 demande la permission de P2, P3 et P4

R3 = {P1, P2, P4}

P3 demande la permission de P1, P2 et P4

- P2 n'est pas en attente d'accès à la ressource : **envoie permission à P1 et P3.**
- P4 n'est pas en attente d'accès à la ressource : **envoie permission à P1 et P3.**
  - **Permission individuelle :** un processus peut donner sa permission à plusieurs autres à la fois.
- Pour accéder à la ressource : P1 lui manque la permission de P3 et P3 lui manque la permission de P1.
- **Le processus prioritaire est celui qui a fait sa demande en premier.**
  - **Si P1 est prioritaire : P3 lui envoie sa permission.**
  - P1 exécute la section critique (a reçu toute les permissions), envoie ensuite sa permission à P3 qui a son tour entre en section critique.

Fonctionnement de l'algorithme de Ricart et Agrawala

- Chaque processus gère les variables locales suivantes :
  - Une horloge ***Hi.***
  - Une variable ***dernier*** qui contient la date de la dernière demande d'accès à la ressource.
  - L'ensemble ***Ri.***
  - Un ensemble d'identificateurs de processus dont on attend une réponse : ***attendu.***
  - Un ensemble d'identificateurs de processus dont on diffère le renvoi de permission si on est plus prioritaire qu'eux : ***différé.***

- **Initialisation :**
  - Hi = dernier = 0.
  - attendu = Ri.
  - différé = Ø.

Fonctionnement de l'algorithme de Ricart et Agrawala

P1

H1 = 0
dernier = 0
R1 = {P2, P3, P4}
attendu = {P2, P3, P4}
différé = Ø

H2 = 0
dernier = 0
R2 = {P1, P3, P4}
attendu = {P1, P3, P4}
différé = Ø

Initialisation

P2

H3 = 0
dernier = 0
R3 = {P1, P2, P4}
attendu = {P1, P2, P4}
différé = Ø

P3

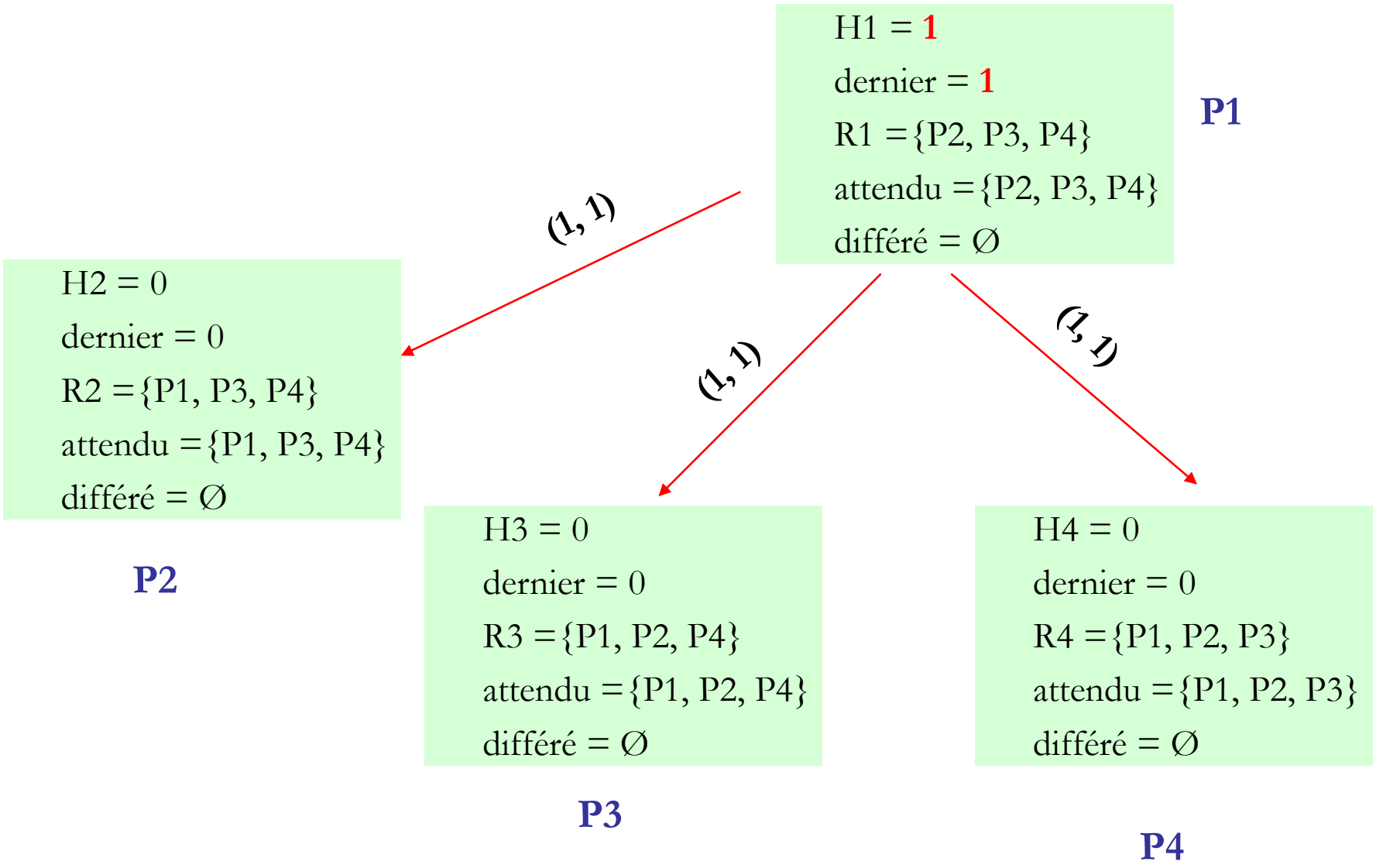
H4 = 0
dernier = 0
R4 = {P1, P2, P3}
attendu = {P1, P2, P3}
différé = Ø

P4

Fonctionnement de l'algorithme de Ricart et Agrawala

- **Si un processus veut accéder à la ressource, il exécute :**
  - Hi = Hi + 1.
  - dernier = Hi.
  - attendu = Ri.
  - Envoie une demande de permission à tous les processus de *Ri* avec estampille *(i, Hi)*.
  - Se met alors en attente de réception de permission de la part de tous les processus dont l'identificateur est contenu dans *attendu*.
- **Quand l'ensemble *attendu* est vide, le processus a reçu la permission de tous les autres processus.**
  - Accède alors à la ressource partagée.
  - Quand accès terminé.
    - Envoie une permission à tous les processus dont l'id est dans *différé*.
    - *différé* est ensuite réinitialisé (différé = Ø).

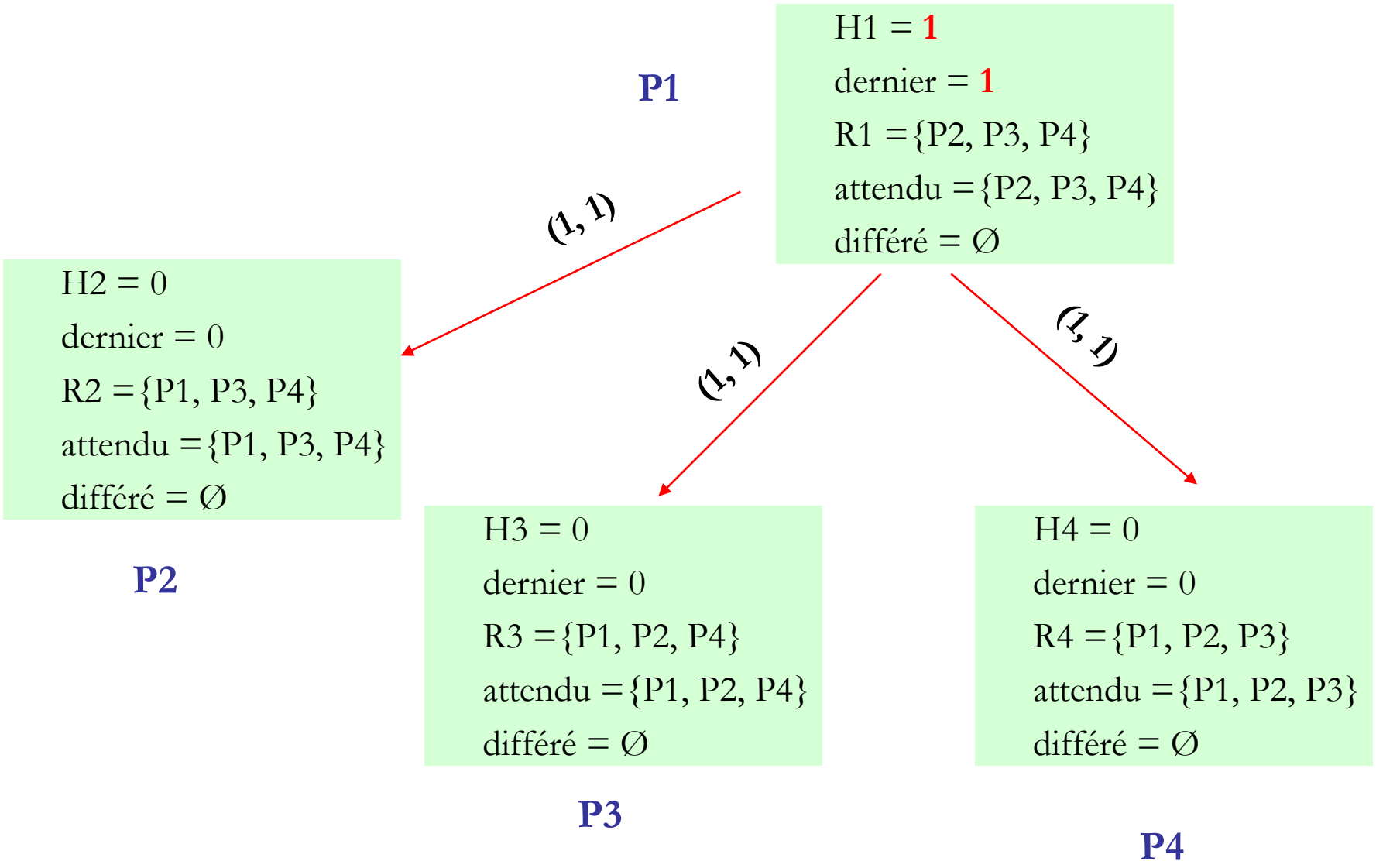
Exemple : P1 souhaiterait accéder à la SC



Fonctionnement de l'algorithme de Ricart et Agrawala

- **Quand un processus *Pi* reçoit une demande de permission de la part du processus *Pj* contenant l'estampille *(j, H).***
  - Met à jour *Hi* : Hi = max (Hi, H).
  - Si *Pi* pas en attente d'accès à la ressource : envoie permission à *Pj*.
  - **Sinon, si *Pi* est en attente d'accès à la ressource :**
    - Si *Pi* est prioritaire : place *j* dans l'ensemble *différé*.
      - On lui enverra la permission quand on aura accédé à la ressource.
    - Si *Pj* est prioritaire : envoi permission à *Pj*.
      - *Pj* doit passer avant moi, je lui envoie ma permission.
    - **La priorité est définie selon la datation des demandes d'accès à la ressource de chaque processus.**
      - Le processus prioritaire est celui qui a fait sa demande en premier.
      - Ordre des dates : l'ordre << de l'horloge de Lamport :
      - **(i, dernier) << (j, H) si ((dernier < H) ou (dernier = H et i < j)).**
- **Quand processus *Pi* reçoit une permission de la part du processus *Pj* :**
  - Supprime l'identificateur de *Pj* de l'ensemble attendu : attendu = attendu – { j }.

Exemple : P1 souhaiterait accéder à la SC



Exemple : P1 souhaiterait accéder à la SC

Mettre à jour les horloges H2, H3 et H4

P1

H1 = 1
dernier = 1
R1 = {P2, P3, P4}
attendu = {P2, P3, P4}
différé = Ø

(1, 1)

(1, 1)

(1, 1)

H2 = 1
dernier = 0
R2 = {P1, P3, P4}
attendu = {P1, P3, P4}
différé = Ø

P2

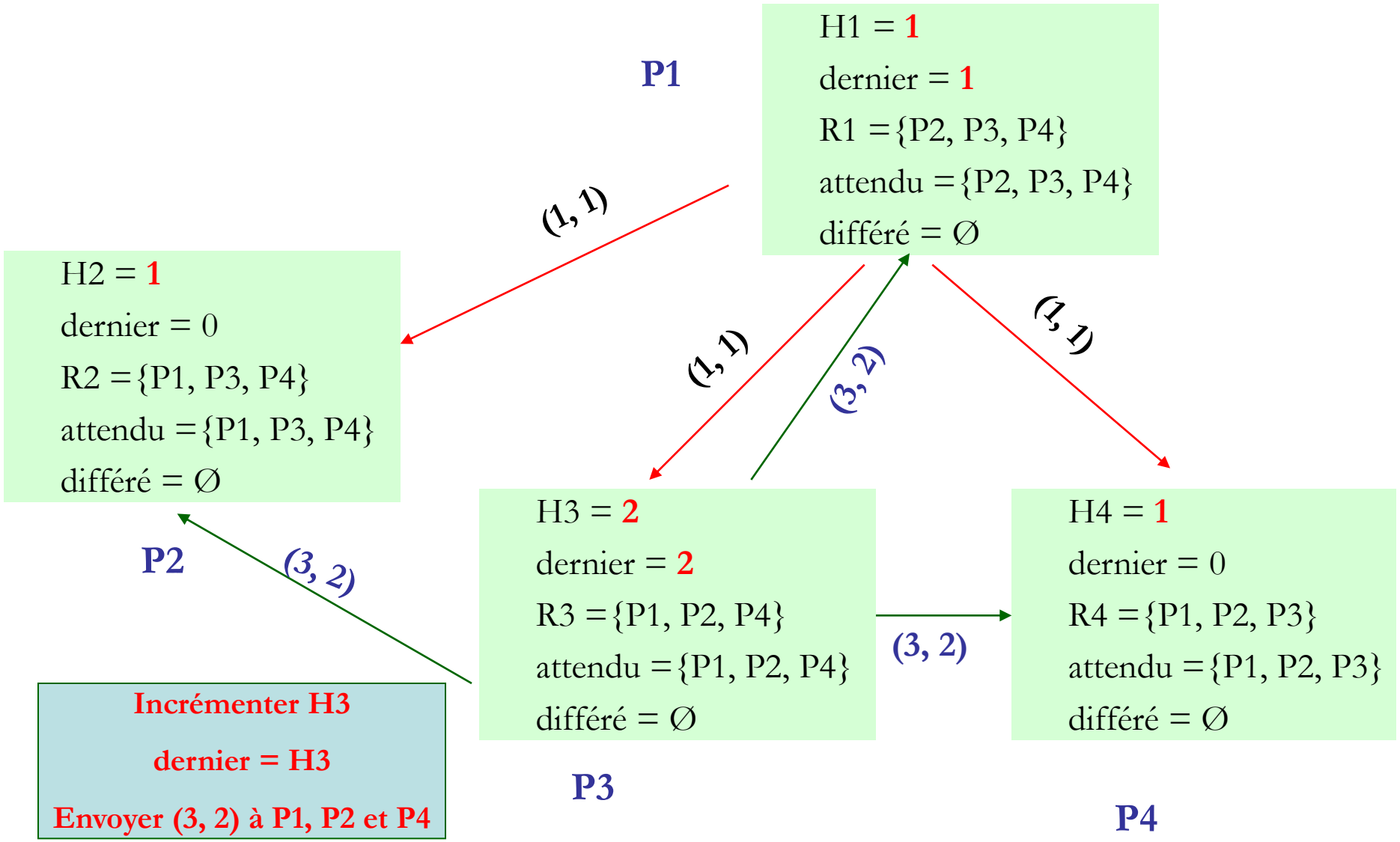
H3 = 1
dernier = 0
R3 = {P1, P2, P4}
attendu = {P1, P2, P4}
différé = Ø

P3

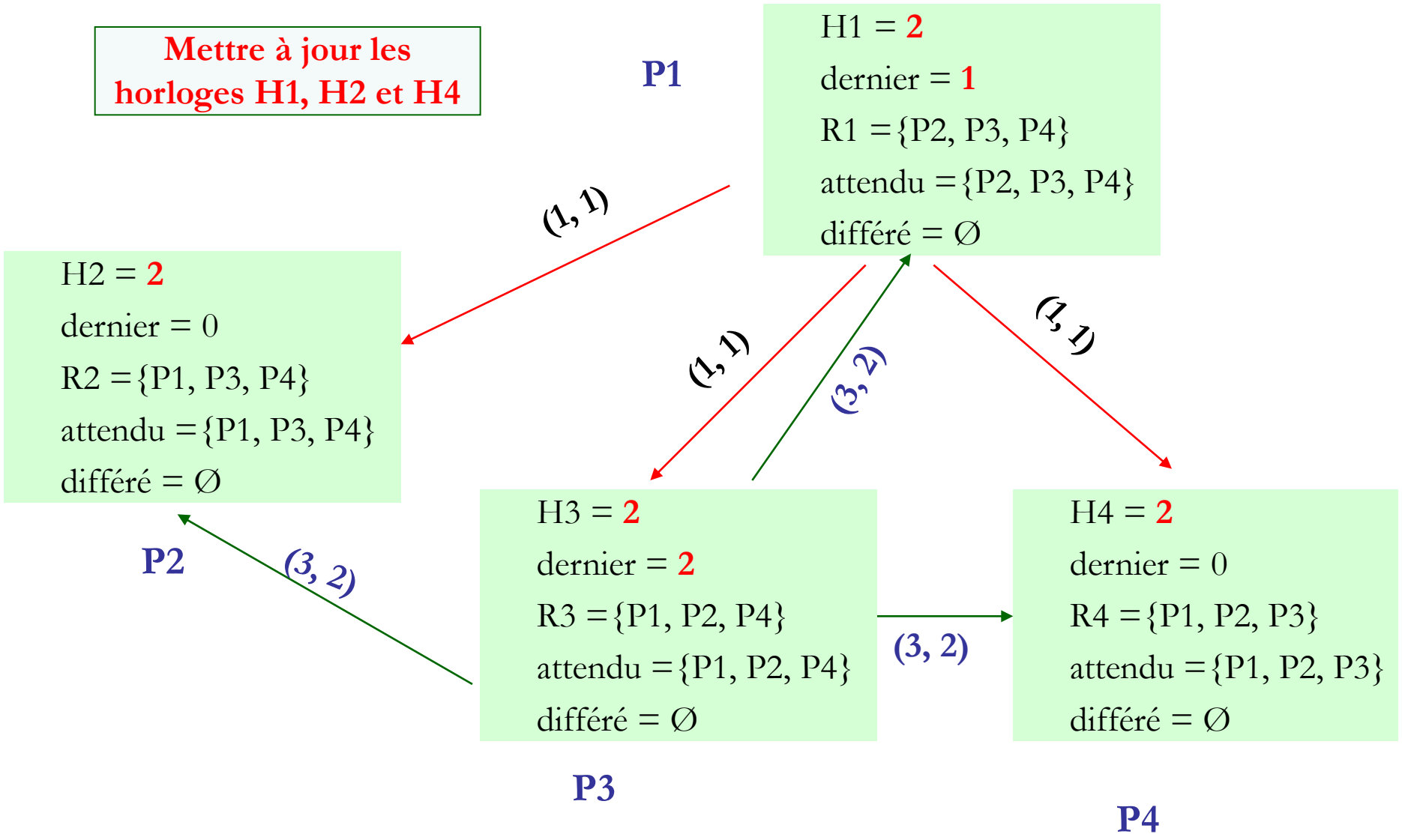
H4 = 1
dernier = 0
R4 = {P1, P2, P3}
attendu = {P1, P2, P3}
différé = Ø

P4

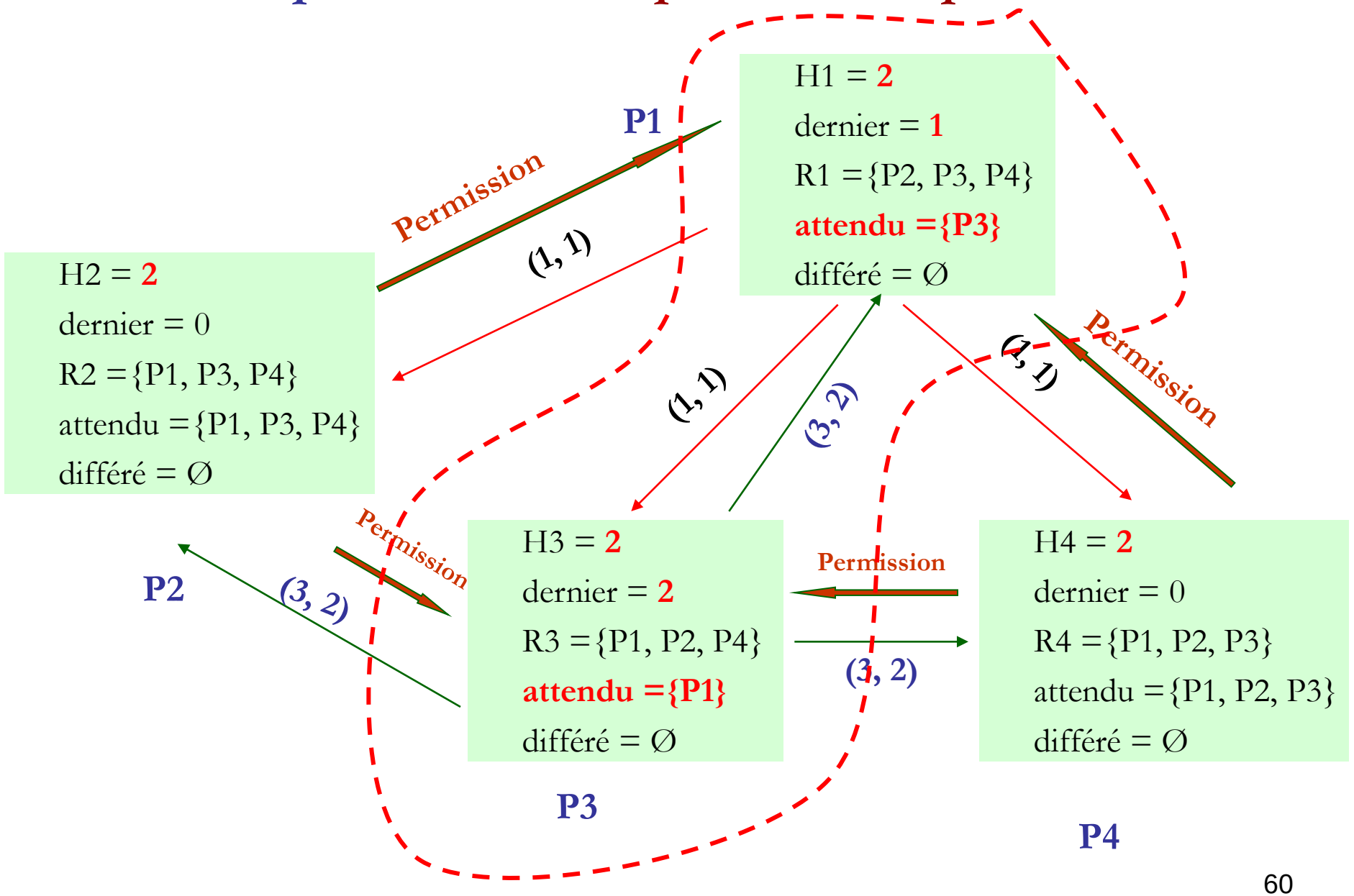
Exemple : P3 souhaiterait accéder à la SC



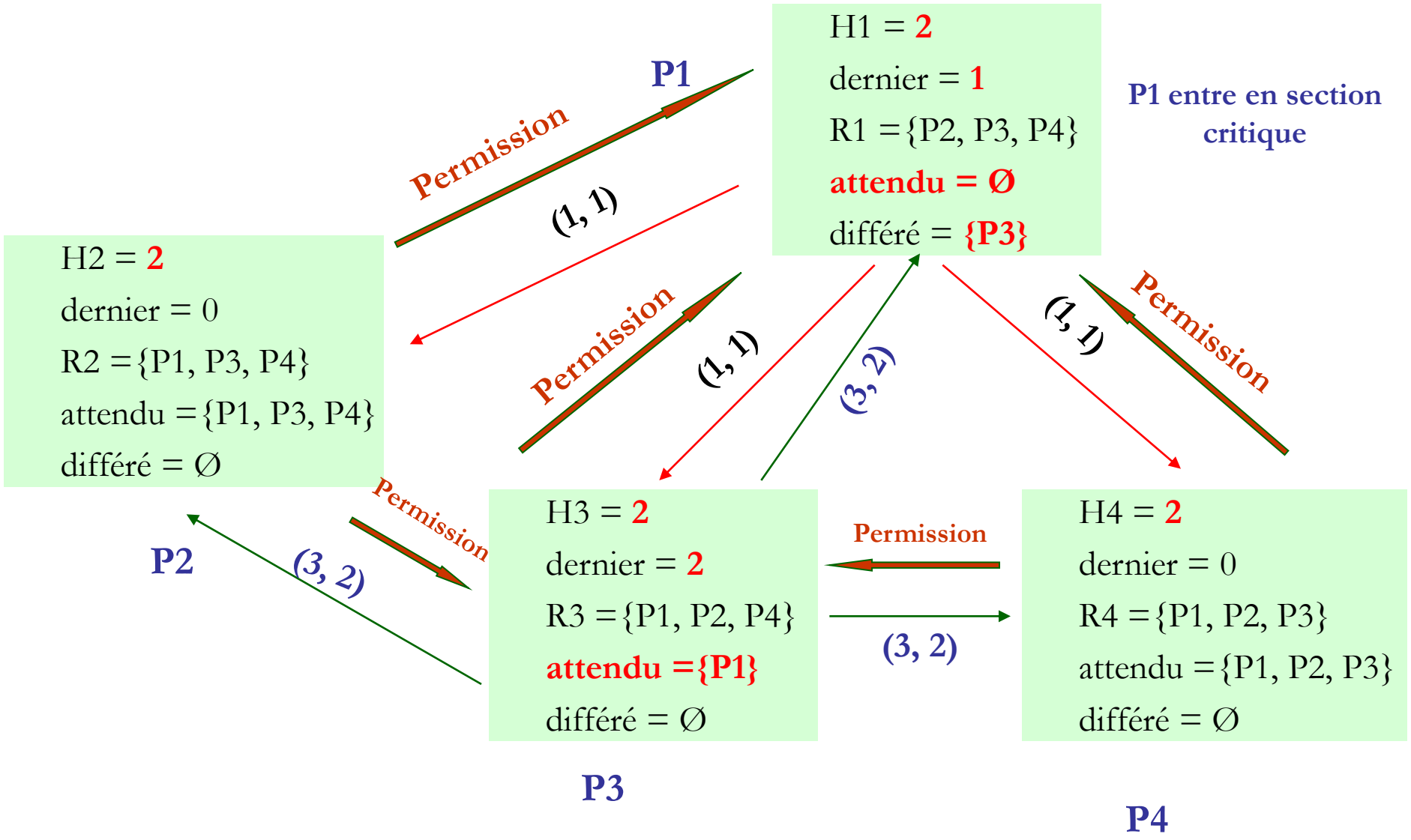
Exemple : P3 souhaiterait accéder à la SC



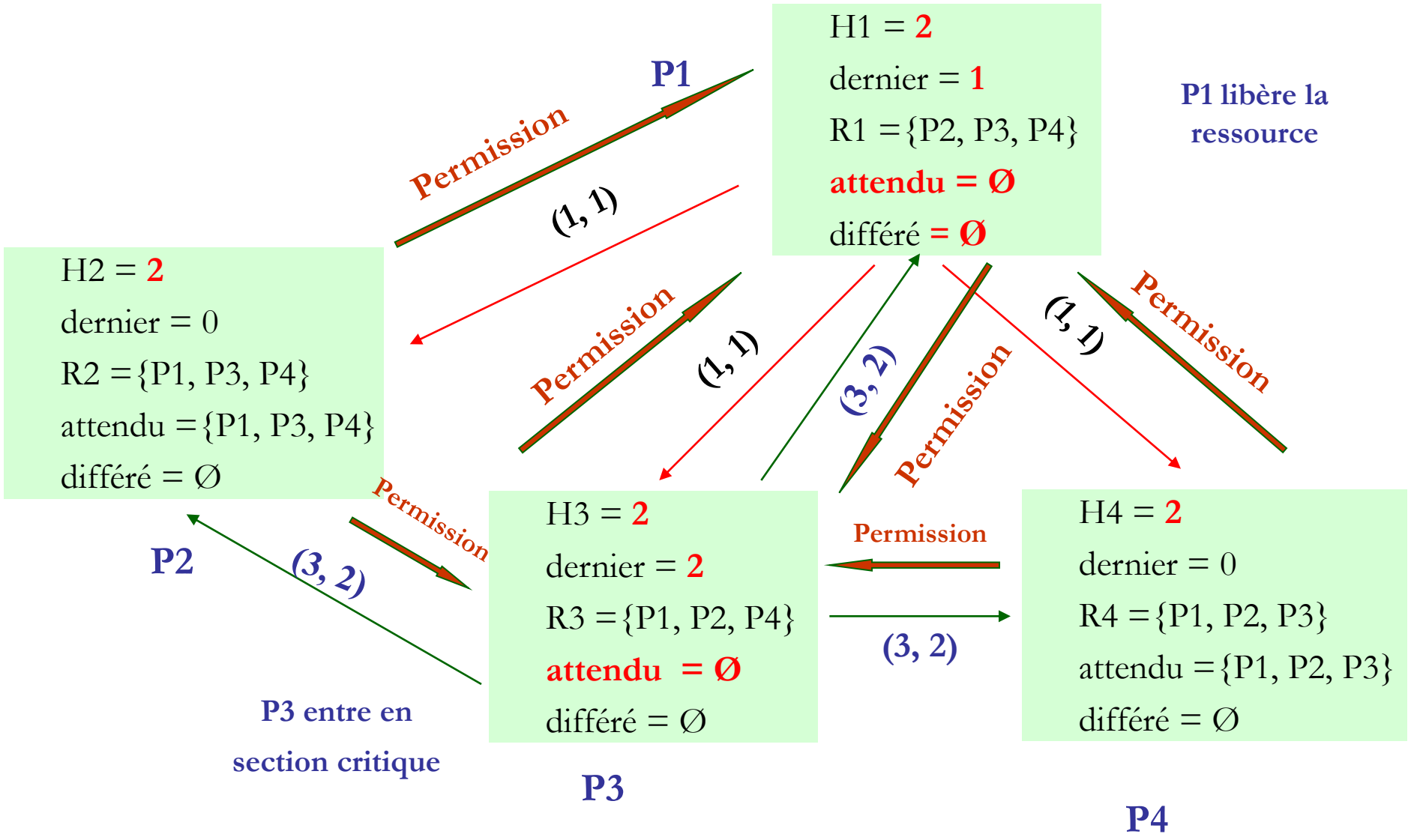
Exemple : envoie des permissions par P2 et P4



Exemple : P1 est prioritaire



Exemple : P1 libère la ressource

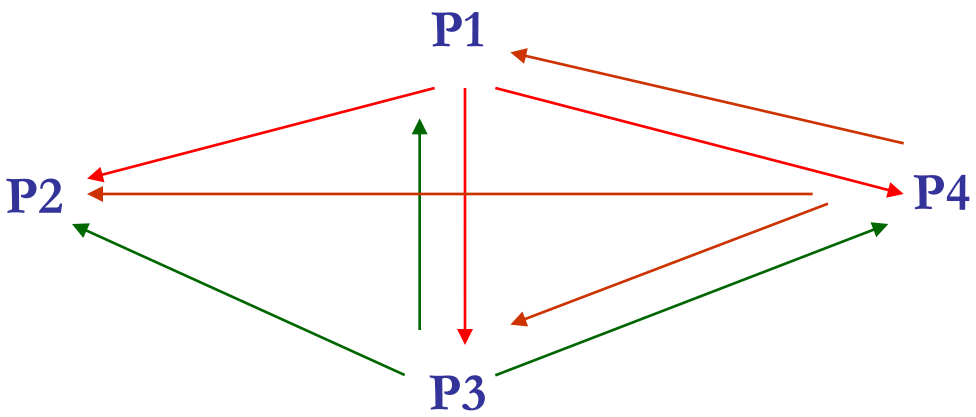


Fonctionnement de l'algorithme de Ricart et Agrawala

- **Respect des propriétés :**
  - **Sûreté :** vérifiée (et prouvable...).
  - **Vivacité :** assurée grâce aux datations et aux priorités associées.
- **Inconvénient principal de cet algorithme :**
  - Nombre relativement important de messages échangés.

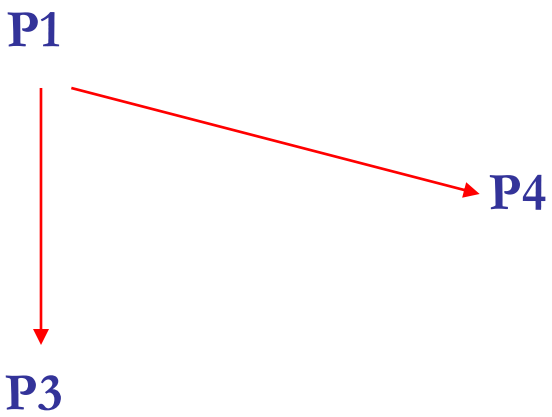
Permission individuelle

- **Amélioration de l'algorithme de Ricart et Agrawala :**
- **Carvalho et Roucairol en 1983 :**
  - Si *Pi* veut accéder plusieurs fois à la ressource partagée et si *Pj* entre 2 accès (ou demandes d'accès) de *Pi* n'a pas demandé à accéder à la ressource.
    - Pas la peine de demander l'autorisation à *Pj* car on sait alors qu'il donnera par principe son autorisation à *Pi*.
    - Limite alors le nombre de messages échangés.



Permission individuelle

- **Amélioration de l'algorithme de Ricart et Agrawala :**
- **Carvalho et Roucairol en 1983 :**
  - Si *Pi* veut accéder plusieurs fois à la ressource partagée et si *Pj* entre 2 accès (ou demandes d'accès) de *Pi* n'a pas demandé à accéder à la ressource.
    - Pas la peine de demander l'autorisation à *Pj* car on sait alors qu'il donnera par principe son autorisation à *Pi*.
    - Limite alors le nombre de messages échangés.



P1 veut accéder à nouveau à la ressource, **P2 entre 2 accès de P1 n'a pas demandé à accéder à la ressource.**

Tolérance aux fautes

- Les algorithmes décrits dans ce cours ne supportent pas des pertes de messages et/ou des crash de processus.
  - Mais adaptation possibles de certains algorithmes pour résister à certains problèmes.
    - **Exemple : pour la méthode par coordinateur, élection d'un nouveau coordinateur en cas de crash.**
  - Peut aussi améliorer la tolérance aux fautes en utilisant des détecteurs de fautes associés aux processus pour détecter les processus morts.