Notes de cours « théorie et pratique de la concurrence »

François Laroussinie
francois.laroussinie@liafa.jussieu.fr
version provisoire – 28 février 2012

Table des matières

1 Introduction

- notion de programmes concurrents
- sémantique d'entrelacement
- adéquation de cette sémantique aux différents types de systèmes concurrents (systèmes multi-tâches, systèmes multi-processeurs, systèmes distribués)
- importance de la notion d'atomicité (de certaines instructions ou groupes d'instructions)
- utilisation de la logique temporelle LTL (de temps linéaire) pour la spécification – l'énoncé – de propriétés sur les exécutions des systèmes

Dans la suite, on va distinguer les algorithmes d'exclusion mutuelle en fonction de la méthode utilisée par les processus pour communiquer. Il y a :

- les algorithmes utilisant des variables *partagées* : tous les processus ont accés en lecture comme en écriture à ces variables partagées (on parle de variables "Multiple-Reader/Multiple-Writers"). C'est le cas des algorithmes de Dekker et Peterson.
- les algorithmes où les processus utilisent des variables *propres* : tous les processus ont accés en lecture à ces variables mais seul un processus (le propriétaire) peut modifier une variable donnée (on parle de variables "Multiple-Reader/Single-Writer"). C'est le cas de l'algorithme de la Boulangerie.
- les algorithmes distribués basés sur la communication par messages.

Dans le cours, nous verrons aussi d'autres approches basées sur l'utilisation de sémaphores, de moniteurs ou d'instructions "Test-and-set".

2 Exemples de programme concurrent

La figure 1 décrit un programme concurrent P composé de deux processus, chacun contient des instructions qui manipulent une variable partagée `x`.

```
            int x := 2      // variable partagée


-- Proc. A                                -- Proc. B
a1: x := x+1                              b1: x := x*3
a2: x := x+2                              b2: end
a3: end
```

FIGURE 1 – Programme P

Quel est le comportement de P ? Quelle est la valeur finale de x après l'exécution de P ? Pour répondre à ces questions, il faut fixer des conventions sur ce que signifie l'exécution *concurrente* de ces processus. Ici on on supposera qu'une telle exécution consiste en un *entrelacement* des actions de chaque processus : chaque processus exécute une ou plusieurs instructions, puis c'est à un autre d'avancer *etc.* Les instructions sont donc exécutées les unes à la suite des autres et non pas en même temps... Cette sémantique repose sur une notion importante : les instructions *atomiques*. Une instruction atomique est une instruction qui ne peut être interrompue : une fois commencée, elle est exécutée complètement. Par exemple, on peut supposer que la lecture du contenu d'une variable ou son affectation sont des opérations

atomiques (on parle de registres atomiques). Mais qu'en est-il d'une instruction comme `x:=x+1` ou `x:=x*3` ? Ici il y a, à la fois lecture de `x`, calcule d'une valeur puis affectation... On peut donc supposer qu'une telle instruction se décompose en (au moins) deux étapes : la lecture de `x` et son affectation dans une variable locale au processus, qui sera utilisée pour le calcul, puis l'affectation du résultat dans `x`. On peut donc le représenter par :

```
temp := x
x := temp + 1
```

Un tel choix est important sur le comportement global du prohramme. A titre d'exemple, on donne à la figure 3 le diagramme d'état du programme précédent lorsque chaque instruction `a1`, `a2` et `b1` est considérée comme étant atomique. Et à la figure 4, on donne le diagramme d'état correspondant à la sémantique utilisant des `temp`, c'est-à-dire au programme P' de la figure 2.

```
int x := 2      // variable partagée

-- Proc. A
a1: temp := x
a1': x := temp +1
a2: temp := x
a2: x := temp+2
a2: end

                                -- Proc. B
                                b1: temp := x
                                b1': x := temp*3
                                b2: end
```

FIGURE 2 – Programme P

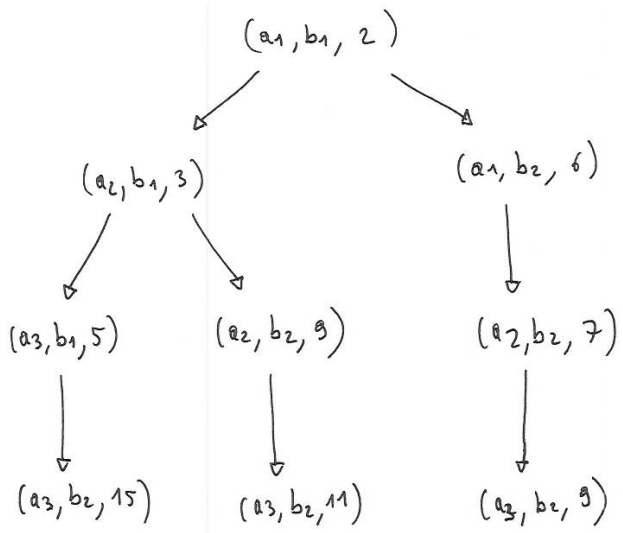

FIGURE 3 – Diagramme d'états de P.

On voit donc que dans le premier cas (P sans variable `temp`), les valeurs finales possibles pour `x` sont 9, 11 ou 15. Mais pour P', elles sont :5, 6, 8, 9, 11 et 15 !

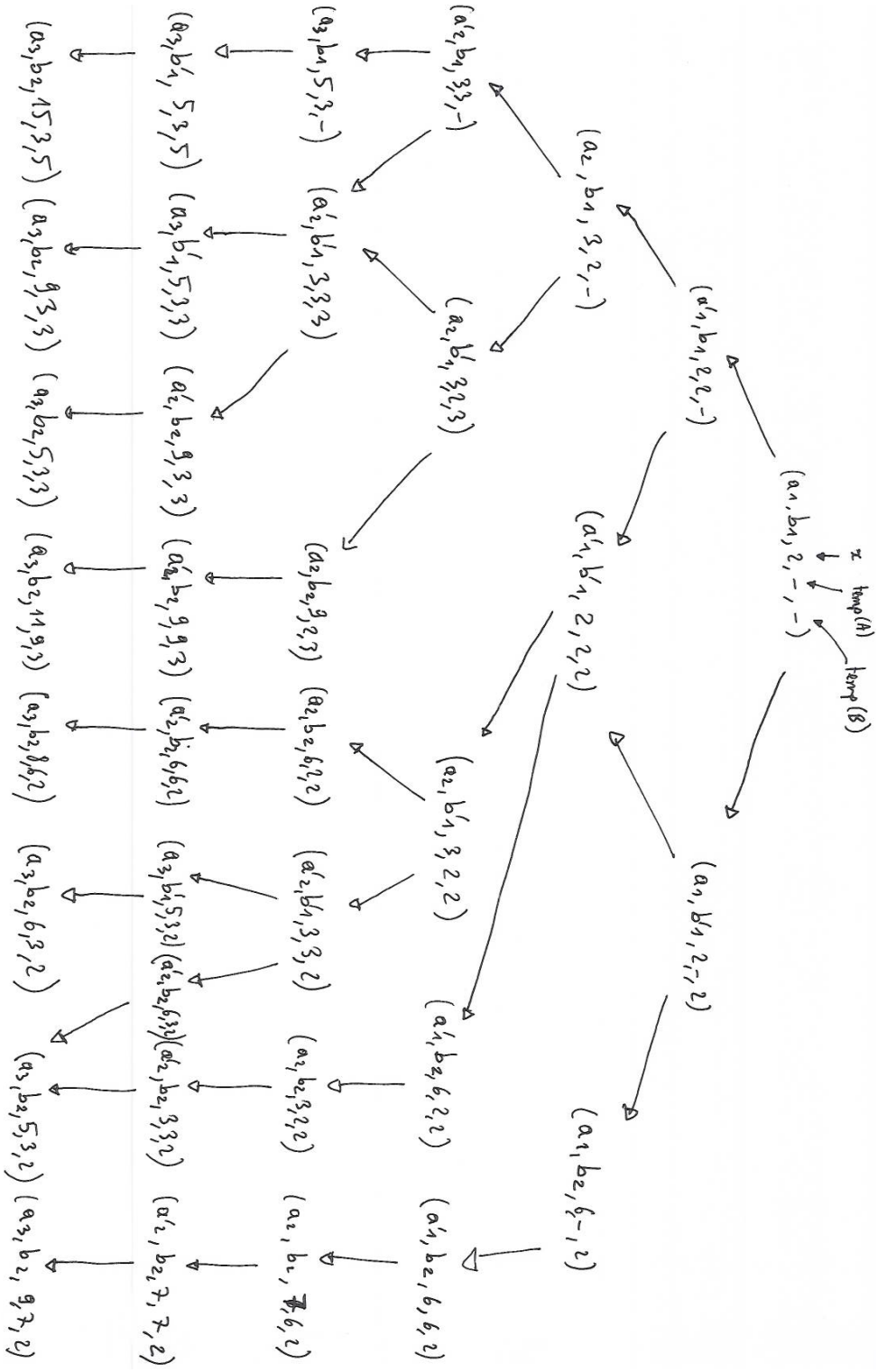

FIGURE 4 – Diagramme d'états de P'.

Le choix des instructions atomiques est donc important, il influe directement sur le com-

portement des programmes concurrents. Il n'y a pas de règle absolue. En fait, cela dépend du langage, du compilateur et de la machine. On peut aussi faire des choix différents pour avoir un programme plus ou moins abstrait. En général il est raisonnable d'éviter les instructions manipulant plusieurs variables (ou occurrence de variables) pouvant modifiées et lues par plusieurs processus (par exemple, dans P, les instructions sont problématiques car `x` apparaît plusieurs fois dans chacune).

Dans la suite, on supposera – sauf indication contraire – que chaque instruction listée dans les processus est atomique.

3 Définitions

On a vu les points suivants :
- Un programme concurrent est un ensemble fini de *processus séquentiels*.
- Chaque processus est décrit par un ensemble fini d'*instructions atomiques*.
- Exécuter un programme concurrent consiste à entrelacer des instructions de chaque processus.
- Un *scénario* (ou un "calcul") est une séquence d'instructions de chaque processus résultant d'un entrelacement des instructions de chaque processus.

Chaque processus dispose d'un *pointeur d'instruction* (ou compteur de programme ou pointeur de contrôle) désignant la **prochaine** instruction devant être exécutée. Au cours d'une exécution d'un programme concurrent, la prochaine instruction est choisie parmi celles pointées par un des pointeurs de contrôleur.

On formalise ici quelques définitions liées au diagramme d'états d'un programme concurrent :

Définition 1
- *L'état d'un programme concurrent $\mathcal{P}$ est un n-uplet contenant : un pointeur d'instruction (indiquant la prochaine instruction devant être exécutée) par processus et une valeur pour chaque variable (locale ou partagée).*
- *Si s_1 et s_2 sont deux états d'un programme $\mathcal{P}$, alors s_2 peut être le successeur de s_1 (noté $s_1 \rightarrow_{\mathcal{P}} s_2$) si s_2 correspond à la transformation de s_1 par l'application d'une instruction pointée dans s_1. On dit aussi qu'il existe une transition entre s_1 et s_2.*
  Par exemple, pour le programme P utilisé précédemment, on avait $(a_2, b_1, 3) \rightarrow_P (a_3, b1, 5)$ (voir la figure 3).
- *Le diagramme d'états de $\mathcal{P}$ est un système de transitions (un graphe avec état initial) $\mathcal{S}_{\mathcal{P}} = (S, A, s_0)$ où $s_0 \in S$ est l'état initial de $\mathcal{P}$ et les ensembles S et A sont les plus petits ensembles vérifiant les propriétés suivantes :*
  - *Si $s \in S$, alors il existe dans S tous les s' tels que $s \rightarrow_{\mathcal{P}} s'$; et*
  - *l'ensemble A contient toutes les arêtes (s, s') telles que $s, s' \in S$ et $s \rightarrow_{\mathcal{P}} s'$.*
- *Un chemin dans $\mathcal{S}$ est appelé un scénario ou une exécution de $\mathcal{P}$.*

Les chemins de $\mathcal{S}_{\mathcal{P}}$ correspondent donc à *tous les entrelacements possibles* d'actions de chaque processus. Ce point est très important : cela signifie que tout entrelacement est possible... En pratique, ce n'est pas le cas (l'ordonnancement des actions des différents processus répond à certaines règles) mais le choix de considérer **tous** les entrelacements possibles est robuste : une fois que nous avons prouvé qu'ils vérifient, tous, les propriétés recherchées, nous savons que le programme réel les vérifiera aussi.

Comment décrire un programme concurrent ? On va utiliser un pseudo-code (comme en algorithmique) avec des primitives spéciales, notamment `await` (voir ci dessous) pour mettre le processus en attente d'une condition. Nous en verrons d'autres lorsque nous nous intéresserons aux algorithmes basés sur des échanges de messages entre processus. En TP, on utilisera Java. Nous verrons aussi le langage Promela utilisé dans l'outil jSPIN (basé sur l'outil SPIN) :

- jSPIN : `http://stwww.weizmann.ac.il/g-cs/benari/jspin/`
- SPIN : `http://spinroot.com/`

Les programmes Promela des algorithmes vus en cours seront disponibles sur la page web du cours : `http://www.liafa.jussieu.fr/~francoisl/m1tpc.html`

Remarque : l'instruction « `await` C » est ici équivalente à « `while` $\neg C$ `: skip;` ». Il s'agit d'une attente active où le processus testera régulièrement la valeur de C tant que celle-ci n'est pas égale à VRAI.

4 Problèmes de section critique

Cette partie s'appuie en grande partie sur [1].

Les algorithmes présentés ne sont pas utilisés « en vrai » (il existe des mécanismes de plus haut niveau pour gérer l'exclusion mutuelle) mais ils illustrent bien les mécanismes généraux de la concurrence.

Le problème. N ($N \geq 2$) processus exécutent continuellement (on utilise une boucle infinie) deux séquences d'instructions :

- la section non-critique (SNC)
- la section critique (SC),

On complète ces deux parties par un pré-protocole (à exécuter avant d'accéder à la SC) et un post-procole (à exécuter après l'accès à la SC).

Les propriétés de *correction* s'énoncent (de manière informelle) comme suit :

- **exclusion mutuelle** : « au plus un seul processus peut être dans sa section critique à un instant donné ».
- **absence d'inter-blocage** : « si plusieurs processus essaient en même temps d'accéder à leur section critique, alors un d'entre eux doit y parvenir » : ils ne peuvent pas se bloquer mutuellement dans la phase de pré-protocole (voir ci-dessous).
- **absence de famine** : « si un processus essaie d'entrer dans sa SC, alors il y parviendra ».
- **attente bornée** : « si un processus P attend pour accéder à sa SC, alors il y arrivera et on peut borner le nombre de fois où d'autres processus pourront accéder à leur SC avant P ».
- **absence de privilège** : il n'y a pas de rôle privilégié pour un processus, chacun est traité à égalité...

On peut définir d'autres propriétés sur ces algorithmes. Toutes ces propriétés ne sont pas de même importance : la première – l'exclusion mutuelle – est la propriété fondamentale de ces algorithmes, et c'est une propriété de sûreté (on veut vérifier qu'une "mauvaise chose" n'arrive pas) . L'absence de famine est également une propriété clé et c'est une propriété de *vivacité* ("quelque chose de bien doit arriver") .

Notons aussi qu'il y a un lien entre l'absence d'interblocage, l'absence de famine et l'attente bornée :
att. bornée $\Rightarrow$ abs. de famine $\Rightarrow$ abs. d'interblocage

Nous allons voir plusieurs algorithmes et pour chacun d'entre eux, on vérifiera quelles propriétés sont satisfaites.

Pour traiter ce problème de section critique, nous allons faire des hypothèses importantes que nous utiliserons par la suite pour prouver la correction de nos algorithmes.

Les hypothèses. Nous faisons les hypothèses suivantes :

1. Nous nous intéressons aux processus structurés de la manière suivante :
   Boucle infinie de :
   - section non-critique
   - **pré-protocole** (le processus **veut** accéder à la SC)
   - section critique (le processus y est arrivé)
   - **post-protocole** (le processus a quitté sa SC)
2. On dispose d'un mécanisme de communication par variables partagées qui vont être utilisées dans le pré-protocole et le post-protocole. Ces variables ne sont pas manipulées en dehors (c'est-à-dire dans la SC et la SNC).
3. La section critique ne bloque jamais (on finit toujours par en sortir et par exécuter le post-protocole).
   NB : cette hypothèse est assez naturelle et facile à garantir si on suppose que la section critique est une petite séquence d'instructions élémentaires (contrairement à la SNC).
4. La section non-critique peut éventuellement bloquer (*i.e.* le processus peut s'arrêter ou boucler) et dans ce cas le pré-protocole ne sera plus exécuté.

5 Algorithmes avec variables partagées et pour deux processus

5.1 Preuve par analyse du graphe d'états

On commence l'algorithme A_1 de la figure 5.

```
            int turn := 1      // variable partagée


-- Processus P1                          -- Processus P2
loop forever :                           loop forever :
p1: section NC                           q1: section NC
p2: await turn==1                        q2: await turn==2
p3: section critique                     q3: section critique
p4: turn:=2                              q4: turn:=1
```

FIGURE 5 – Algorithme A1

L'idée de l'algorithme est très simple : à chaque instant la variable `turn` désigne le numéro du processus dont c'est le *tour* d'accéder à la section critique. Un processus souhaitant accéder à la SC doit attendre son tour.

L'instruction `p2` de P_1 est son pré-protocole et `p4` est son post-protocole (pour P_2, il s'agit de `q2` et `q4`).

Comment prouver que cet algorithme vérifie la propriété d'exclusion mutuelle ? On peut utiliser deux techniques : on peut montrer cette propriété sur le graphe d'états de l'algorithme (une inspection de ce graphe montrera que la propriété est assurée) et on peut aussi utiliser une preuve classique sur l'algorithme. Pour cet algorithme, nous allons utiliser la première méthode. Pour l'algorithme suivant (algorithme de Dekker), nous utiliserons la seconde.

Construction du diagramme d'états. Un état de l'algorithme $A1$ contient :

- un pointeur sur la prochaine instruction de P_1,
- un pointeur sur la prochaine instruction de P_2,
- la valeur de `turn`,
- et la valeur des différentes variables internes de P_1 et P_2 utilisées dans les sections critiques et non-critiques.

Pour représenter le comportement lié au protocole, on peut oublier les variables autres que `turn` (car elles n'ont pas d'effet sur les propriétés étudiées). On va donc représenter un état de l'algorithme par un triplet (p, q, t) où p est un élément de $\{\texttt{p1}, \texttt{p2}, \texttt{p3}, \texttt{p4}\}$, q un élément de $\{\texttt{q1}, \texttt{q2}, \texttt{q3}, \texttt{q4}\}$ et t un élément de $\{1, 2\}$.

Le graphe d'états va contenir tous les états de cette forme et des transitions représentant l'exécution de l'instruction p ou q et conduisant à des états où les pointeurs ont été mis à jour ainsi que la variable `turn`. Pour vérifier la propriété d'exclusion mutuelle, il suffit de vérifier qu'aucun état $(\texttt{p3}, \texttt{q3}, 1)$ ou $(\texttt{p3}, \texttt{q3}, 2)$ n'est atteignable depuis l'état de départ $(\texttt{p1}, \texttt{q1}, 1)$.

Combien peut-il y avoir d'états dans le graphe ? $4 \times 4 \times 2 = 32$. On va le construire de manière incrémentale en partant de l'état de départ. Cela donne le graphe de la figure 6.

Dans ce diagramme d'états, nous avons précisé pour chaque transition, si elle correspondait au processus P_1 ou au processus P_2 en étiquetant la transition avec 1 ou 2.

On peut remarquer que chaque état a toujours deux ou trois transitions sortantes : au moins une correspond à l'exécution pointée dans P_1, et au moins une correspond à celle de P_2. Comme nous ne savons pas si l'exécution d'une section non-critique termine ou boucle, c'est-à-dire si son exécution conduit à l'instruction suivante (`p2` ou `q2`), nous sommes obligés de considérer les deux possibilités : une boucle sur le même état (le pointeur restant en `p1` ou `q1`) ou une transition vers `p2` ou `q2`... Les transitions de boucle en SNC sont indiquées avec des pointillés.

Notons aussi qu'il n'y a que 16 états car les 16 autres ne sont pas accessibles depuis l'état de départ.

Equité entre processus. Une exécution de l'algorithme A1 correspond à un chemin (infini) dans le graphe d'états. Mais l'inverse n'est pas vrai : certains chemins infinis du graphe ne sont pas de « bonnes » exécutions de l'algorithme. Par exemple, les chemins qui bouclent infiniment dans un unique état avec une transition « pleine » correspondent à une exécution infinie soit du pré-protocole de P_1 (pour les états de la forme $(\texttt{p2}, -, 2)$), soit du pré-protocole de P_2 (pour les états de la forme $(-, \texttt{q2}, 1)$). Or dans tous ces cas, cela signifierait que l'un des deux processus est systématiquement empêché de progresser : ce serait toujours l'autre qui aurait la main. Ce genre de situation n'est pas acceptable : on souhaite autoriser tous les entrelacements possibles d'actions des deux processus mais pas l'exclusion *ad vitam aeternam* d'un des deux processus... Les exécutions de A1 seront donc représentées par les chemins

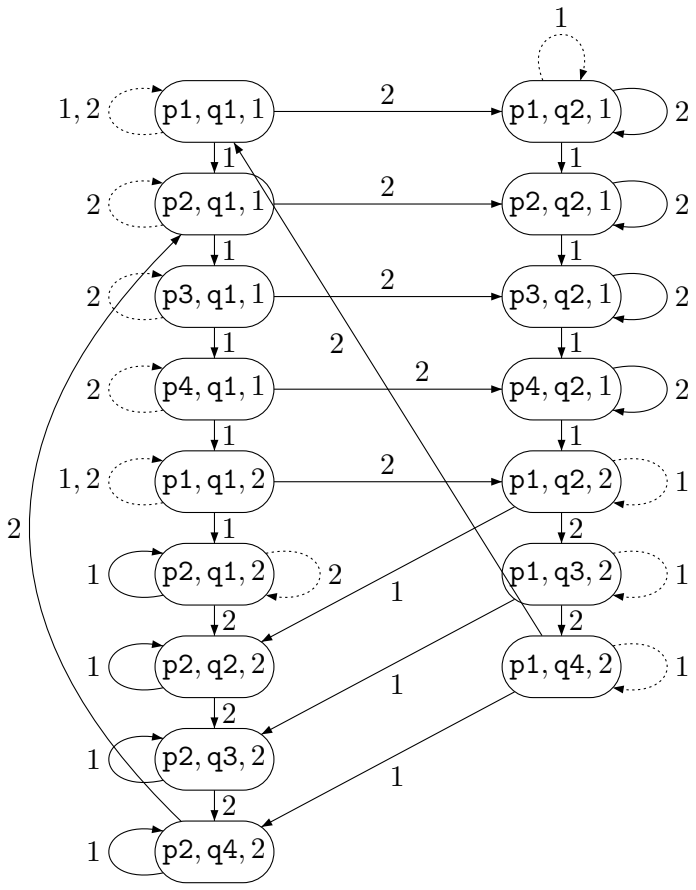

FIGURE 6 – Diagramme d'états de A1.

équitables où « un processus qui peut avancer, avancera un jour ». **Dans le diagramme d'états de la figure 6, les deux processus peuvent toujours exécuter une action, et donc les chemins équitables sont les chemins infinis où il y a une infinité de transitions étiquetées par** 1 **et une infinité de transitions étiquetées par** 2**.**

Etude des propriétés de l'algorithme. L'analyse du graphe d'états de A1 montre clairement que les états $(\texttt{p3}, \texttt{q3}, 1)$ et $(\texttt{p3}, \texttt{q3}, 2)$ ne sont pas accessibles. **La propriété d'exclusion mutuelle est donc vérifiée.**

Qu'en est-il des autres propriétés ?

- **absence d'inter-blocage** : les processus ne peuvent pas se bloquer mutuellement dans la phase de pré-protocole.
  Cette propriété est aussi vérifiée : une analyse du graphe de la figure 6 montre que lorsque les deux processus sont ensemble dans leur phase de Pre-protocole, il y en a toujours un qui peut accéder à la SC : P_1 si `turn` vaut 1, et P_2 si `turn` vaut 2, sachant que `turn` vaut toujours 1 ou 2.
- **absence de famine** : La question est de savoir si à tout moment lorsqu'un processus se trouve dans son pré-protocole (*i.e.* quand il souhaite accéder à sa SC), alors il y arrivera

un jour.

On ne considère ici que des exécutions équitables entre processus. Par exemple, on exclut une exécution qui bouclerait sur l'état $(\texttt{p2}, \texttt{q2}, 1)$: car cette boucle consiste à exécuter infiniment le test `turn==2` du processus P_2. Dans une exécution équitable, après un certain nombre (arbitrairement grand) de tours de boucle, le processus P_1 finira toujours par avancer en passant « `await turn==1` » et accédera à $(\texttt{p3}, \texttt{q2}, 1)$.

Mais une analyse du graphe de A1 montre qu'il est possible de rester bloquer dans l'état $(\texttt{p1}, \texttt{q2}, 1)$ tout en suivant une exécution équitable (car contenant à la fois des transitions du processus P_1 en pointillées et des transitions du processus $P_2 \ldots$). Ce scénario correspond au cas où le processus P_1 ne sort pas de sa SNC et ne mettra donc plus `turn` à 2, ce qui empêche P_2 d'accéder à sa SC ! On a bien sûr le même phénomène pour P_1 depuis l'état $(\texttt{p2}, \texttt{q1}, 2)$. On voir ici l'importance de l'hypothèse sur la non-terminaison possible de la SNC.

Dans l'algorithme A1, il y a une famine est possible.

- **attente bornée** : Cette propriété est clairement fausse puisqu'il peut y avoir famine...
- **absence de privilège** : On peut considérer que le processus P_1 est avantagé car il est sûr de pouvoir accéder au moins une fois à sa SC contrairement à P_2.

Dans l'annexe A, on discute de plusieurs simplifications possibles pour le diagramme d'états de A1.

5.2 Algorithme de Dekker

On considère maintenant l'algorithme de Dekker (proposé en 1964) qui est décrit à la figure 7. C'est, semble-t-il, le premier algorithme correct pour résoudre le problème de section critique dans ce cadre.

```
            boolean D1 := False      // variables partagées
            boolean D2 := False
            int turn := 1


-- Processus P1                      -- Processus P2
loop forever :                       loop forever :
p1: section NC                       q1: section NC
p2: D1 := True                       q2: D2 := True
p3: while (D2 == True) :             q3: while (D1 == True) :
p4:       if (turn == 2)             q4:       if (turn == 1)
p5:            D1 := False           q5:            D2 := False
p6:            await (turn == 1)     q6:            await (turn == 2)
p7:            D1 := True            q7:            D2 := True
p8: section critique                 q8: section critique
p9: turn := 2                        q9: turn := 1
p10: D1:=False                       q10: D2:=False
```

FIGURE 7 – Algorithme de Dekker

Idée de l'algorithme. Lorsque P_1 veut accéder à sa SC, il commence par mettre `D1` à VRAI (signifiant ainsi à P_2 son souhait d'y aller). Ensuite si P_1 constate que P_2 souhaite aussi accéder à sa SC (*i.e.* `D2` == `True`), alors il regarde qui est prioritaire (valeur de `turn`) : si c'est P_2, il se met en attente de son tour (`turn` == 1) après avoir signifié à P_2 qu'il n'était plus candidat à la SC (en mettant `D1` à FAUX). Après avoir accéder à sa SC, P_1 mettra `turn`à 2 (donnant donc la priorité à P_2) et remettant `D1` à FAUX.

Voici deux exemples de scénario pour l'algorithme de Dekker. Les états successifs du système sont représentés ligne par ligne mais on ne note que les changements (de pointeur d'instruction et/ou de variables) :

	P_1	P_2	D1	D2	turn
1	p1	q1	$\bot$	$\bot$	1
2	p2				
3	p3		$\top$		
4		q2			
5	p8				
6		q3		$\top$	
7		q4			
8		q5			
9		q6		$\bot$	
10	p9				
11	p10				2
12		q7			
13		q3		$\top$	
14	p1		$\bot$		
15		q8			
			...		

	P_1	P_2	D1	D2	turn
1	p1	q1	$\bot$	$\bot$	1
2		q2			
3		q3		$\top$	
4	p2				
5	p3		$\top$		
6		q4			
7	p4				
8	p3				
9		q5			
10		q6		$\bot$	
11	p8				
			...		

Pour cet algorithme, nous allons d'abord prouver un premier lemme qui énonce plusieurs invariants dans lesquels on utilise les étiquettes des instructions (`p-` ou `q-`) pour indiquer la position du pointeur de contrôle du processus P_1 ou P_2.

Lemme 1 *Les trois propriétés suivantes sont des invariants de l'algorithme de Dekker :*
- $turn == 1 \vee turn == 2$
- $(p3 \vee p4 \vee p5 \vee p8 \vee p9 \vee p10) \Leftrightarrow D1 == \top$
- $(q3 \vee q4 \vee q5 \vee q8 \vee q9 \vee q10) \Leftrightarrow D2 == \top$

Preuve : On montre que la propriété est vraie au début et se maintient tout au long de l'exécution de l'algorithme.

Pour la première propriété, le résultat est direct : `turn` est initialisé à 1, puis on ne lui affecte que les valeurs 1 ou 2...

Pour les deux autres propriétés, on peut examiner chaque processus séparément car la première propriété ne porte que sur le processus P_1 et la seconde que sur P_2. En effet, la variable `D1` n'est modifiée que par P_1 (et `D2` que par P_2).

On construit donc le diagramme d'états de la figure 8 qui décrit les comportements possibles de P_1 : chaque état du graphe contient la prochaine instruction de P_1 à exécuter et la

valeur courante de D1. Notons qu'avec un tel graphe, nous ne pouvons pas savoir le résultat des tests sur turn ou D2 puisque leurs valeurs ne sont pas représentées dans les états. Nous sommes donc obligés de représenter les deux possibilités (test vrai ou test faux) et faire partir deux transitions différentes. Nous avons donc une approximation du comportement de P_1 dans la mesure où certaines exécutions de ce graphe ne seraient pas forcément possibles dans le graphe complet. Mais nous allons voir que malgré ces comportements supplémentaires, nous avons bien toujours la propriété recherchée sur la valeur de D1 et les états de P_1. Cela nous permettra de conclure que la propriété est vraie sur le « vrai » comportement de P_1 (qui est plus restreint).

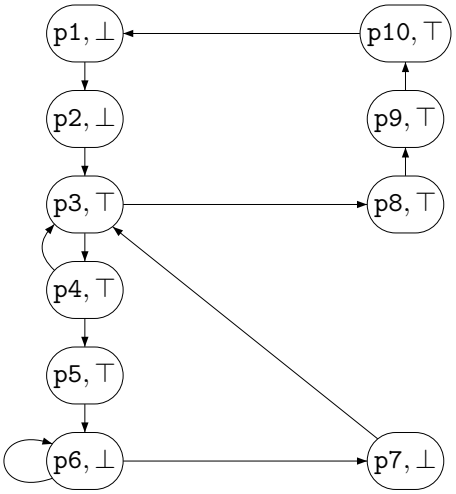

FIGURE 8 – Diagramme d'états simplifié de P_1 (avec la valeur de D1.

On fait la même chose pour D2 et P_2. □

Ce lemme peut aussi s'énoncer en utiliser l'opérateur □ de la logique temporelle (voir l'annexe B) pour énoncer « il est toujours vrai... », on dira ainsi les propriétés suivantes sont vraies pour algorithme :

- $\Box \Big(\texttt{turn} == 1 \ \vee \ \texttt{turn} == 2 \Big)$
- $\Box \Big((\texttt{p3} \vee \texttt{p4} \vee \texttt{p5} \vee \texttt{p8} \vee \texttt{p9} \vee \texttt{p10}) \ \Leftrightarrow \ \texttt{D1} == \top \Big)$
- $\Box \Big((\texttt{q3} \vee \texttt{q4} \vee \texttt{q5} \vee \texttt{q8} \vee \texttt{q9} \vee \texttt{q10}) \ \Leftrightarrow \ \texttt{D2} == \top \Big)$

On en déduit le théorème :

Théorème 1 (Exclusion mutuelle de l'algorithme de Dekker)
L'exclusion mutuelle est assurée : on a $\Box\neg(\texttt{p8} \wedge \texttt{q8})$.

Preuve : Pour que le pointeur de P_1 « passe » en p8 , il faut que le programme soit sorti de la boucle en p3, et donc que la variable D2 soit à False. Ce dernier point implique (par le lemme précédent) que le pointeur de P_2 se trouve en q1, q2, q6 ou q7. Et cela exclut qu'il soit en q8.

On fait le même raisonnement pour le cas où le pointeur de P_2 passe en `q8` : on en déduit alors que celui de P_1 ne peut pas être en `p8`.

Il y a bien exclusion mutuelle dans la section critique. □

Théorème 2 *Il n'y a pas d'interblocage dans l'algorithme de Dekker.*

Preuve : Montrer l'absence d'interblocage consiste à montrer que si les deux processus sont dans leur pré-protocole, alors il ne peuvent pas y rester bloqués tous les deux. On fait une preuve par l'absurde. Supposons qu'il existe un interblocage, alors les deux processus P_1 et P_2 sont bloqués dans leur instructions `p3`,...,`p7` et `q3`...,`q7`. Supposons que la variable `turn` vaille 1 (nous savons que cette valeur ne changera plus puisque les deux processus ne pourront plus la modifier avec leurs instructions `p9` et `q9`). Dans ce cas, P_1 boucle sur le `while` et le `if` : P_1 exécutera pour toujours les instructions `p3`,`p4`,`p3`,... On a donc `D1` à `True` pour tous les états, et la variable `D2` doit être vraie *infiniment souvent* (pour permettre à P_1 de tester correctement le `while`). Que peut faire P_2 ? Il ne peut pas être bloqué en `q6` car sinon cela empêcherait `D2` de prendre la valeur `True` infiniment souvent. P_2 doit donc boucler en `q3`, mais alors il doit exécuter `q4` et comme `turn==1`, alors il se retrouve bloquer en `q6`, ce qui est contradictoire avec la remarque précédente.
Il n'y donc pas d'interblocage : si les deux processus exécutent leur pré-protocole, alors un en sortira... □

Nous allons maintenant montrer l'absence de famine. Il s'agit donc de montrer que tout processus qui entre dans son pré-protocole finira par accéder à sa section critique. Cette propriété s'énonce $\Box(\texttt{p2} \Rightarrow \Diamond \texttt{p8})$ en logique temporelle. On a le résultat suivant :

Théorème 3 *L'algorithme de Dekker garantit l'absence de famine sous hypothèse d'équité entre processus et en supposant que toute section critique termine et conduit à l'exécution du post-protocole. L'algorithme vérifie :*

- $\Box(p2 \Rightarrow \Diamond p8)$, *et*
- $\Box(q2 \Rightarrow \Diamond q8)$

Preuve : Supposons qu'il y ait une famine pour P_1. Alors cela signifie que P_1 va rester bloquer dans son pré-protocole. Que peut-il se passer pour P_2 ?

- Il ne peut pas bloquer dans le pré-protocole (car il n'y a pas d'interblocage, voir le théorème 2).
- Supposons qu'il bloque dans sa section non-critique. Soit t_4 un instant où P_2 est bloqué dans sa SNC et P_1 est bloqué dans son pré-protocole. Alors on a `D2==False` pour tous les instants futurs (grâce au Lemme 1) . Donc si P_1 boucle, ce n'est pas dû au `while`, mais au `await (turn==1)` (instruction `p6`). Donc `turn==2` (pour toujours).
  Soit t_2 le dernier instant où P_1 a exécuté l'instruction `p3` ($t_2 < t_4$) : en t_2, on avait `D2==True` et donc entre t_2 et t_4, le processus P_2 a exécuté `D2:=False` (`q10`), soit t_3 cet instant.
  Maintenant, nous savons qu'avant t_3, P_2 a effectué `q9` mettant ainsi `turn` à 1 (soit $t_0 < t_3$ cet instant) : comme `turn` vaut 2 en t_4, il a fallu qu'après t_0, P_1 exécute `p9` pour mettre `turn` à 2... soit t_1 cet instant : on a $t_0 < t_3$ et aussi $t_1 < t_2$. D'où la figure 9.
  En t_1, le pointeur de programme de P_1 est donc en `p9` et celui de P_2 est en `q10`. On sait de plus que `D1==True` et `D2==True`. Une telle configuration (`p9`, `q10`, 1, `True`, `True`) est

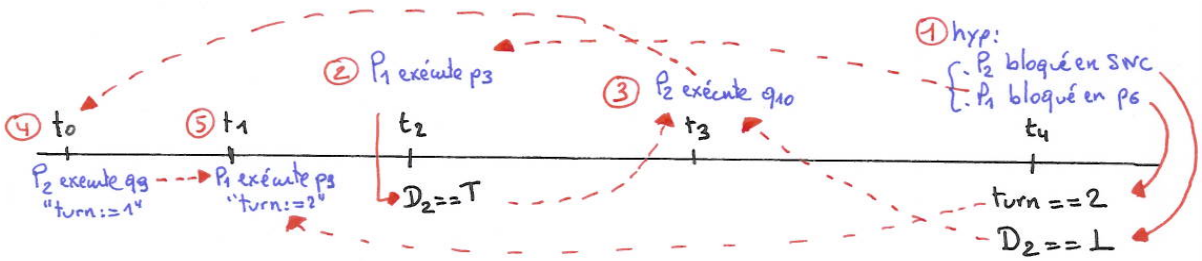

FIGURE 9 – Scénario conduisant au blocage de P_1 dans le pré-protocole avec P_2 en SNC.

inaccessible si l'exclusion mutuelle est assurée (voir le lemme 2 ci dessous qui montre ce point en détail). Ce cas est donc aussi impossible.
Le blocage de P_1 alors que P_2 reste dans sa section non-critique est donc impossible.

– Supposons enfin que P_2 ne bloque pas dans sa SNC : qu'il exécute son protocole infiniment souvent. Mais alors après avoir accédé à sa SC, il mettra `turn` à 1, donnant ainsi la priorité à P_1. Cette valeur pour `turn` ne changera plus. Du coup, P_1 ne peut plus être bloqué en `p6` et bouclera en `p3p4p3p4`... et `D1` sera à `True` pour toujours. Mais alors P_2 testera son `while` et finira bloqué en `q6`! Ce qui conduirait à un interblocage, ce qui n'est pas possible...

□

On utilise le lemme suivant dans la preuve d'absence de famine :

Lemme 2 *Les configurations* ($p9$, $q10$, $turn = 1$, $D1 = \top$, $D2 = \top$) *ou* ($p9$, $q10$, $turn = 2$, $D1 = \top$, $D2 = \top$) *sont inaccessibles dans le diagramme d'états de l'algorithme de Dekker.*

Preuve : Pour le montrer il suffit de calculer les prédécesseurs possibles pour ces configurations (on omet la valeur de `turn` car elle n'est pas utile ici). Notons d'abord que les dernières instructions effectuées par P_1 sont $\texttt{p3} \cdot \texttt{p8} \cdot \texttt{p9}$, et celles de P_2 sont $\texttt{q3} \cdot \texttt{q8} \cdot \texttt{q9} \cdot \texttt{q10}$. La figure 10 contient ce graphe d'états construits « en arrière » à partir de $(\texttt{p9}, \texttt{q10}, -, \texttt{D1} = \top, \texttt{D2} = \top)$.

A chaque état on associe les deux transitions entrantes correspondant à l'exécution d'une instruction de P_1 ou P_2. Une transition en pointillée est une transition impossible à franchir.

Une analyse du graphe montre bien que de tous les chemins menant à une configuration de la forme $(\texttt{p9}, \texttt{q10}, -, \top, \top)$ passe par un conflit en section critique $(\texttt{p8}, \texttt{q8}, \ldots)$, ce qui est exclu par le théorème 1.

□

Il n'y a donc pas de famine, ni d'interblocage. Le protocole vérifie-t-il l'attente bornée ? Oui car un processus P peut se faire doubler au plus une fois : il se fait doubler si la variable `turn` lui est « défavorable » mais alors l'autre processus Q modifie `turn` de manière à avantager P...

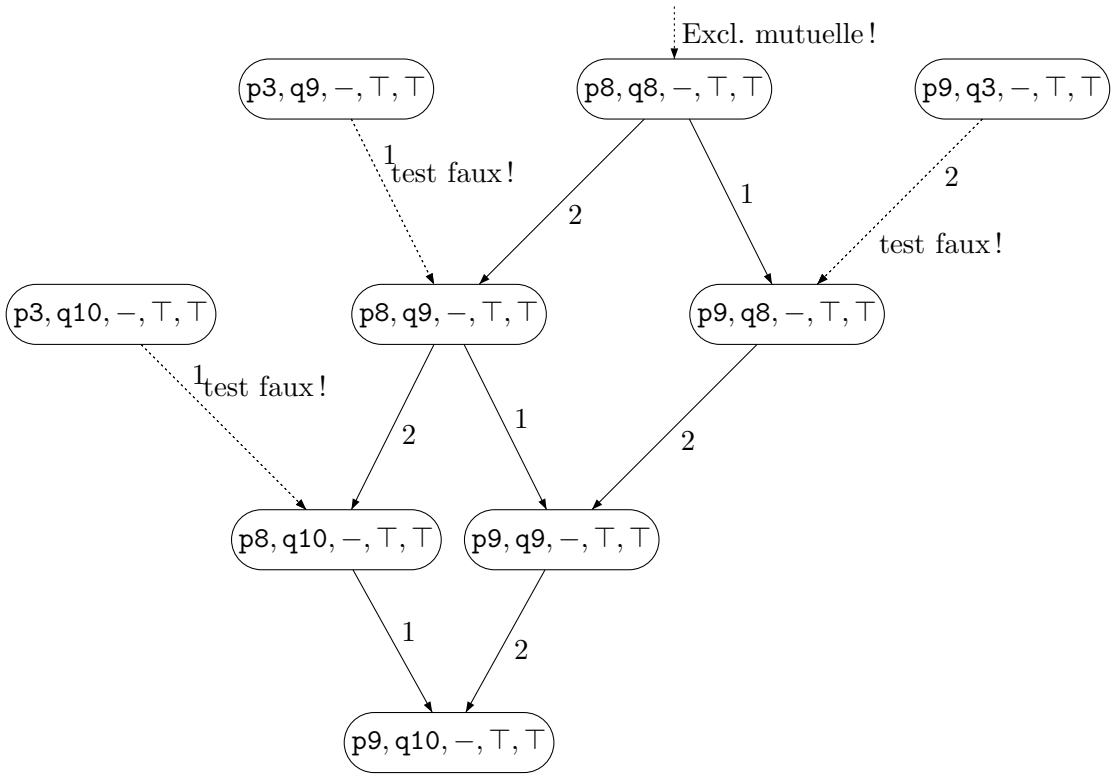

FIGURE 10 – Graphe ≪ arrière ≫ depuis $(\mathtt{p9}, \mathtt{q10}, -, \top, \top)$ – algorithme de Dekker.

5.3 Algorithme de Peterson

L'algorithme de Peterson est plus simple et plus récent (1981) que l'algorithme de Dekker. Il est décrit dans la figure 11.

```
boolean D1 := False      // variables partagées
boolean D2 := False
int turn := 1
```

```
-- Processus P1                          -- Processus P2
loop forever :                           loop forever :
p1: section NC                           q1: section NC
p2: D1 := True                           q2: D2 := True
p3: turn := 2                            q3: turn := 1
p4: await (D2==False  OR turn==1)        q4: await (D1==False  OR turn==2)
p5: section critique                     q5: section critique
p6: D1:=False                            q6: D2:=False
```

FIGURE 11 – Algorithme de Peterson

Idée de l'algorithme. Lorsque le processus P_1 veut accéder à sa SC, il le signale en mettant `D1` à « VRAI » puis donne la priorité à l'autre processus P_2, et ensuite vérifie avant d'accéder à sa SC, que P_2 ne veut pas y aller (`D2` == `False`) ou alors qu'il (P_1) est prioritaire.

On a :

Théorème 4 (Exclusion mutuelle de l'algorithme de Peterson)
L'exclusion mutuelle est garantie : on a $\Box\neg(p5 \wedge q5)$.

Preuve : Supposons que P_1 soit déjà dans sa SC lorsque P_2 accède à sa SC. On a alors `D1==D2==True` (NB : la valeur de `D1` n'est modifiable que par P_1 et celle de `D2` n'est modifiable que par P_2 : on peut donc connaître directement leur valeur en fonction du pointeur d'instruction des deux processus). Donc, si P_2 accède à sa SC, c'est que `turn` vaut 2. Quel test a pu permettre à P_1 d'accéder dans sa SC ?

- `D2 == False` ? Mais alors P_2 a (depuis l'arrivée de P_1 en SC) fait « `D2:=True` » puis « `turn:=1` » et personne n'a pu changer cette valeur et donc lui permettre d'accéder à la SC...
- `turn == 1` ? Là encore, P_2 a modifié `turn` après la modification (« `turn:=2` ») de P_1 et donc `turn` vaut toujours 1 lorsque P_2 veut accéder à sa SC et donc il ne peut y arriver.

Donc P_2 ne peut pas rejoindre P_1... □

Il n'y a pas d'interblocage des processus lors du pré-protocole. En effet, si il y avait un tel blocage, cela signifierait que P_1 serait en attente de « `D2==False OR turn==1` » et P_2 serait en attente de « `D1==False OR turn==2` », on en conclut que dans cette situation on aurait « `D1==True` », « `D2==True` » et surtout « `turn!=1` » et « `turn!=2` », ce qui n'est clairement pas possible !

On en déduit :

Théorème 5 *L'algorithme de Peterson garantit l'absence de famine sous hypothèse d'équité entre processus et en supposant que toute section critique termine et conduit à l'exécution du post-protocole. L'algorithme vérifie :*

- $\Box(p2 \Rightarrow \Diamond p5)$, *et*
- $\Box(q2 \Rightarrow \Diamond q5)$

Preuve : Supposons que P_1 soit en situation de famine. Il a donc exécuté `p2`, puis `p3` (l'équité en processus garantit sa progression) et arrivera en `p4` où il restera bloqué sur le `await`. Si il est bloqué à ce stade, c'est que l'on a (`D2==True` $\wedge$ `turn==2`) lors des tests du `await`. Cela signifie que la valeur de `D1` restera à `True` pour toujours (P_1 est bloqué), tandis que `turn` devra valoir 2 et `D2` `True` infiniment souvent (à chaque fois que P_1 testera la condition de son `await`). Mais si la variable `D2` est à `True`, c'est que le processus P_2 se trouve en `q3`, `q4`, `q5` ou `q6` :

- de `q3`, il irait en `q4` (équité entre processus) et mettrait `turn` à 1 avant de bloquer en `q4` et laisser passer P_1, ce qui est contradictoire avec l'hypothèse de départ... Ce n'est donc pas possible.
- en `q4`, il franchirait le `await` si `turn` vaut 2, et passerait en `q5` ;
- de `q5`, il irait en `q6` (car la SC termine) ; et
- de `q6`, il finirait par mettre `D2` à `False` et arriverait en `q1`. De là, P_2 ne peut pas bloquer en SNC car alors `D2` resterait à `False` pour toujours et cela contredirait l'hypothèse du

blocage de P_1 en `p4`... Donc P_2 exécutera `q2` puis `q3` et on retombe sur le premier cas qui contredit l'hypothèse.

On a donc vu que dans tous les cas, P_2 ne peut pas assurer le fait d'avoir infiniment souvent (`D2==True` $\wedge$ `turn==2`). Il n'y a donc pas de famine pour P_1. □

Le protocole vérifie-t-il l'attente bornée ? Oui : un processus P peut se faire doubler au plus **une** fois (il se fait doubler si il exécute son pré-protocole juste après le processus Q et modifie la variable `turn` pour lui donner l'avantage, mais alors l'autre processus Q, après avoir atteint sa SC, modifiera `turn` de manière à avantager P...).

Dans l'algorithme de Dekker, il n'y a pas vraiment de rôle privilégié pour un processus, même si l'initialisation de la variable `turn` donne un léger avantage à P_1. Et dans l'algorithme de Peterson, il n'y a strictement aucun privilège : l'initialisation de la variable n'influe pas sur le comportement de l'algorithme (pourquoi ?).

6 Algorithmes distribués pour n processus

6.1 De 2 à n : les tournois

Ici nous allons voir une idée proposée par G.L. Peterson et M.J. Fischer en 1977 pour passer d'un algorithme de section critique pour **2** processus à un algorithme pour **n** processus (voir [4]). L'idée repose sur une approche "diviser-pour-régner".

Supposons que l'on dispose de $n = 2^k$ processus : P_0, P_1,..., P_{n-1}. On va faire un tournoi à $\log(n)$ (*i.e.* k) tours. Après le premier tour, il ne reste que $\frac{n}{2}$ processus (bien sûr chaque compétition entre deux processus est régie par l'algorithme de section critique pour deux processus dont on dispose : ici on va utiliser l'algorithme de Peterson), puis après le second tour il ne reste que $\frac{n}{4}$ processus, *etc.*

On numérote les tours de 0 à $\log(n) - 1$ selon la figure 12(a).

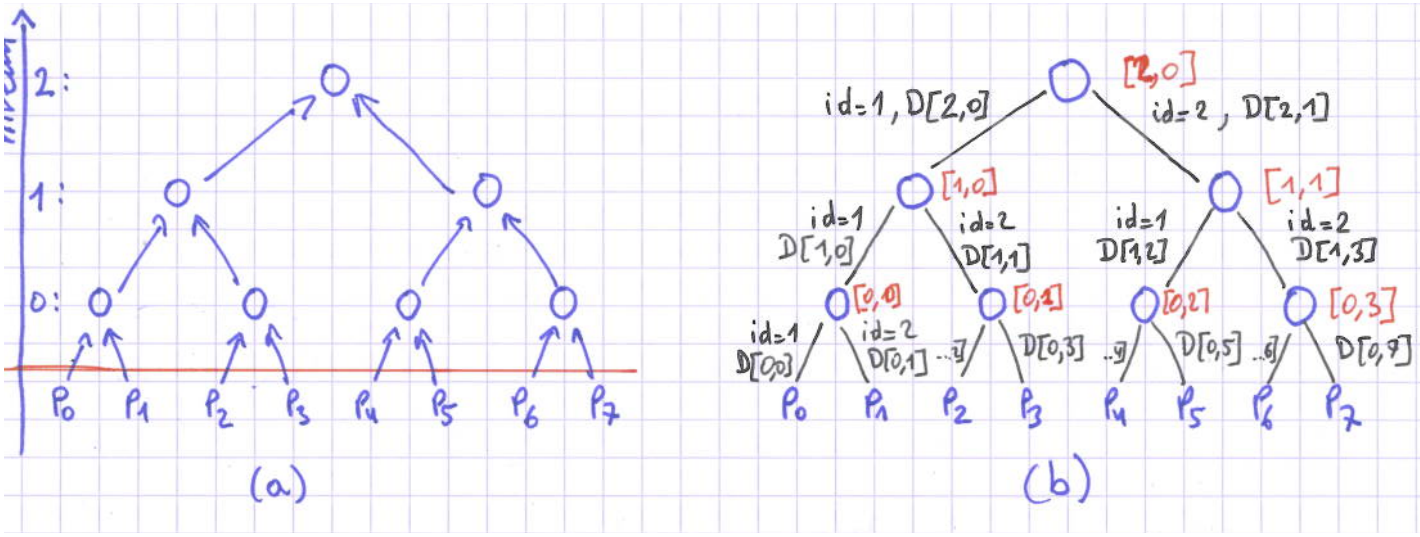

FIGURE 12 – Organisation du tournoi

Chaque compétition est repérée par un niveau ℓ et un numéro k : $\ell \in \{0, \ldots, \log(n)-1\}$ et $k \in \{0, \ldots, 2^{\log(n)-1-\ell} - 1\}$. Pour chaque compétition, on va utiliser l'algorithme de Peterson et on doit donc disposer de l'équivalent des variables partagées `turn`, `D1` et `D2`, pour cela on utilise :

- `turn`$[\ell, k]$: qui est prioritaire ?
- `D`$[\ell, 2k]$: est-ce que le processus de gauche veut accéder à la SC ?
- `D`$[\ell, 2k+1]$: est-ce que le processus de droite veut accéder à la SC ?

De plus, chaque processus P_i doit pouvoir se repérer et savoir à tout moment à quel niveau il se trouve et si il correspond au "processus gauche" ou au "processus droit" du tournoi en cours afin de pouvoir interpréter correctement la valeur de `turn`$[\ell, k]$. Pour chaque processus P_i, on a donc aussi les variables locales suivantes :

- ℓ : le niveau de la compétition en cours ;
- k : le noeud correspondant à la compétition en cours au niveau ℓ ; et
- `id` $\in \{1, 2\}$: indiquant si P_i est le processus de gauche (*i.e.* joue le rôle de P_1 dans l'algorithme de Peterson) ou si c'est le processus de droite (*i.e.* joue le rôle de P_2).

Algorithme $\texttt{P}_i$
loop forever :
1 Section NC
2 $k := i$
3 **pour** $\ell = 0 \dots \log(n) - 1$ **faire**
4 $\quad \texttt{id} := (k\%2) + 1$
5 $\quad k := \lfloor \frac{k}{2} \rfloor$
6 $\quad \texttt{D}[\ell, 2k + (\texttt{id} - 1)] := \texttt{true}$
7 $\quad \texttt{turn}[\ell, k] := 3 - \texttt{id}$
8 $\quad$ **await**$(\neg\, \texttt{D}[\ell, 2k + 2 - \texttt{id}] \ \vee\ \texttt{turn}[\ell, k] == \texttt{id})$
9 Section critique
10 **pour** $\ell = \log(n) - 1 \dots 0$ **faire**
11 $\quad \texttt{D}[\ell, \lfloor \frac{i}{2^\ell} \rfloor] := \texttt{false}$

Algorithme 1 : Processus P_i pour le tournoi

Exemples... Supposons que l'on s'intéresse au processus P_5 dans un tournoi à 8 processus. Alors :

- Dans le pre-protocole, les 3 itérations pour $\ell = 0, 1, 2$ vont donner les instructions suivantes :
  1. $\texttt{id} := 2$, $\texttt{D}[0, 5] := \texttt{true}$ (car $2 \cdot \lfloor \frac{5}{2} \rfloor + (2 - 1) = 5$), $\texttt{turn}[0, 2] := 1$ et **await**$(\neg\, \texttt{D}[0, 4] \vee \texttt{turn}[0, 2])$.
  2. $\texttt{id} := 1$, $\texttt{D}[1, 2] := \texttt{true}$ et $\texttt{turn}[1, 1] := 2$ et **await**$(\neg\, \texttt{D}[1, 3] \vee \texttt{turn}[1, 1])$.
  3. $\texttt{id} := 2$, $\texttt{D}[2, 1] := \texttt{true}$ et $\texttt{turn}[2, 0] := 1$ et **await**$(\neg\, \texttt{D}[2, 0] \vee \texttt{turn}[2, 0])$.
- Ensuite, dans le post-protocole, le processus P_5 mettra à `false` les trois variables $\texttt{D}[2, 1]$, $\texttt{D}[1, 2]$ et $\texttt{D}[0, 5]$.

On peut vérifier sur la figure 12(b), que les variables utilisées par P_5 sont bien celles correspondant aux différentes compétitions aux quelles doit participer P_5 pour accéder à la SC.

Théorème 6 *L'exclusion mutuelle est assurée par le tournoi.*

Preuve : Si deux processus P_i et P_j se trouvent dans leur SC au même moment, c'est qu'ils ont franchi au moins un même noeud (ℓ, k) du tournoi. Soit (ℓ, k) celui de ces noeuds ayant un niveau minimal ℓ. Pour ce niveau, les deux processus viennent de deux branches différentes et donc lors de l'exécution du préprotocole pour le niveau ℓ, ils ont chacun un `id` différent : si on note `id` celui de P_i et $\texttt{id}'$ celui de P_j, on a $\texttt{id} = 3 - \texttt{id}'$.

Supposons que ce soit P_i qui rejoigne P_j en SC. Alors P_i a franchi le noeud (ℓ, k) (et a donc exécuté **await**$(\neg\texttt{D}[\ell, 2k + 2 - \texttt{id}] \ \vee\ \texttt{turn}[\ell, k] == 1)$) alors que P_j avait déjà franchi ce noeud sans avoir depuis, exécuté son post-protocole. Du coup, si P_i franchit le noeud (ℓ, k), c'est que l'algorithme de Peterson pour deux processus est faux... Or nous savons que l'algorithme de Peterson est correct : si P_j a franchi avec succès le noeud (ℓ, k) sans avoir depuis exécuté son post-protocole, P_i ne peut pas franchir le noeud (ℓ, k). □

L'absence de famine est aussi garanti :

Théorème 7 *L'algorithme du tournoi assure l'absence de famine sous hypothèse d'équité entre processus et en supposant que toute section critique termine et conduit à l'exécution du post-protocole.*

Preuve : Si un processus P commence à exécuter son pré-protocole et se retrouve bloqué à un noeud (ℓ, k), alors il deviendra prioritaire dès que le processus responsable du blocage à ce noeud aura exécuté son post-protocole. L'équité entre processus lui assurera alors de franchir le noeud (ℓ, k). □

Mais **l'attente n'est pas bornée** : pendant qu'un processus est bloqué en (ℓ, k), d'autres processus peuvent accéder à leur SC un nombre arbitrairement grand de fois. Sur l'exemple précédent, on peut imaginer que P_0 est bloqué en $(0, 1)$, et pendant ce temps les processus P_2, P_3,..., P_7 peuvent accéder à leur SC un nombre de fois arbitrairement grand (le noeud $(0, 1)$ n'est pas utilisé pendant toute cette période). Bien sûr cela suppose que P_0 est très très lent (arbitrairement lent !).

6.2 Avec des variables propres : l'algorithme de la boulangerie

On s'intéresse à des problèmes faisant intervenir $n \geq 2$ processus.

A présent nous n'utiliserons plus des variables partagées (comme la variable `turn` des algorithmes de Dekker ou Peterson, que chaque processus peut lire et modifier. Ici les différents processus ne pourront communiquer que par des variables *propres* ou « Multiple-Readers/Single-Writer » : une telle variable peut être lue par tous les processus mais elle ne peut être modifiée que par un seul processus.

Notons que ce mécanisme permet à chaque processus P d'obtenir des informations sur l'état d'un autre processus Q (par la lecture des variables propres de Q) mais cela ne lui permet pas d'influer directement sur ces états (impossible de modifier ces variables). On est donc dans une configuration différente des premiers algorithmes présentés où les processus partageait réellement et complètement des variables.

Nous allons étudier l'algorithme de la Boulangerie (Bakery algorithm) proposé par Lamport en 1974. L'idée de cet algorithme est assez classique et on le retrouve dans les gares, les bureaux de poste, l'administration : Il suffit de choisir un numéro supérieur à celui de toutes les personnes déjà en attente, puis d'attendre suffisamment longtemps pour que ce numéro devienne le plus petit des numéros des personnes en attente...

La figure 13 présente l'algorithme de la Boulangerie. Les tableaux `Choosing` et `Nb` sont des variables propres : chaque élément est lisible par tous les processus mais seul le processus i peut modifier les i-ème éléments de ces tableaux (c.-à-d. `Choosing[i]` et `Nb[i]` appartiennent à P_i).

Dans cet algorithme on doit calculer le `Max` de tous les numéros utilisés par les processus. Cette opération n'est pas considérée comme une opération atomique : on peut donc avoir plusieurs max qui se calculent en même temps et obtenir le même numéro pour chacun de ces calculs. Comme l'algorithme repose sur l'idée que le numéro minimal est prioritaire, on résout les « égalités » en considérant les numéros de processus (on suppose que ces numéros sont uniques) dans la relation d'ordre, on définit ainsi la relation $<<$ sur les paires d'entiers :

$$(n, i) << (n', i') \quad \Leftrightarrow \quad \Big((n < n') \vee (n == n' \wedge i < i')\Big)$$

Notons que si $i \neq i'$ on a toujours $(n, i) << (n', i')$ ou $(n', i') << (n, i)$.

On notera $\texttt{pX}^{(i)}$ le fait que le processus P_i se trouve à la X-ème instruction.

L'algorithme de la Boulangerie garantit l'exclusion mutuelle :

```
Algorithme de la Boulangerie
boolean Choosing[1..n] := False       // variables propres
int Nb[1..n] := 0

-- Processus Pi
loop forever :
p1: section NC
p2: Choosing[i] := True
p3: Nb[i] := Max(Nb[1..n])+1
p4: Choosing[i] := False
p5: For j =1..n : if j != i :
p6:   await Choosing[j]==False
p7:   await Nb[j]==0  or  (Nb[i],i) << (Nb[j],j)
p8: section critique
p9: Nb[i] := 0
```

FIGURE 13 – Algorithme de la Boulangerie

Théorème 8 *L'algorithme de la Boulangerie assure l'exclusion mutuelle de la section critique : la propriété* $\Box \bigwedge_{i \neq j} \neg(p8^{(i)} \wedge p8^{(j)})$

Preuve : On va montrer que lorsque P_i arrive en section critique, aucun autre processus ne peut déjà s'y trouver.

Supposons que P_i atteigne sa SC au temps t_2. Soit P_k un autre processus ($k \neq i$). On veut montrer que P_k n'est pas en SC au temps t_2.

Puisque P_i est en SC au temps t_2, c'est qu'il a franchi, lors de l'itération $j = k$ de la boucle `for`, son instruction `p6` (soit t_0 cet instant où `Choosing[k]==False`) et son instruction `p7` (soit t_1 cet instant où l'on a `Nb[k]==0 or ((Nb[i],i) ≪ (Nb[k],k))`). On a bien sûr $t_0 < t_1 < t_2$.

Où peut être P_k à l'instant t_0 ? Il y a deux cas :

- en `p1` ou en `p2` : lorsque P_k essaiera d'entrer en SC, son numéro sera forcément supérieur strictement à `Nb[i]` car la valeur de `Nb[i]` sera prise en compte dans le calcul du max. Et donc P_k ne pourra ni doubler, ni rejoindre rejoindre P_i dans la SC.
- en `p5`, `p6`, `p7`, `p8` ou `p9` : Dans ce cas, on distingue deux autres cas selon la valeur du test `(Nb[k],k]) ≪ (Nb[i],i)` à l'instant t_0 :
  - `(Nb[k],k]) ≪ (Nb[i],i)` : alors `Nb[k]` sera obligatoirement remis à zéro avant l'instant t_1 (où P_i franchit `p7` en examinant P_k) et on se ramène au cas précédent avec P_k en `p1` ou `p2`, ou au cas suivant avec `(Nb[i],i] ≪ (Nb[k],k)`...
  - `(Nb[i],i]) ≪ (Nb[k],k)` : alors dans ce cas, P_k ne peut pas franchir son instruction `p7` lors de son examen de P_i avant que P_i ne remette `Nb[i]` à zéro, c'est à dire en sortant de la SC. Donc P_k ne peut accéder à sa SC entre t_0 et t_2.
    Il reste donc à vérifier que P_k n'a pas pu accéder à sa SC avant t_0 (et y être toujours). Si cela était le cas, alors P_k serait arrivé dans la SC après son choix de `Nb[k]`. Or au moment du choix de `Nb[k]`, le calcul de `Nb[i]` a déjà commencé (ou même terminé)

car `Nb[k]` est supérieur ou égal à `Nb[i]` et donc lorsque P_k examinera P_i lors de son instruction `p7`, il ne pourra pas la franchir avant P_i et donc il ne peut pas accéder à sa SC avant t_0.

Il n'y a donc pas de possibilité où P_i rejoigne un autre processus en SC. □

Il n'y a pas d'interblocage car le processus bloqué avec le plus petit numéro (et, en cas d'égalité, ayant le plus petit indice) passera en SC. L'absence de famine utilise le même argument :

Théorème 9 *L'algorithme de la Boulangerie garantit l'absence de famine sous hypothèse d'équité entre processus et en supposant que toute section critique termine et conduit à l'exécution du post-protocole. L'algorithme vérifie :* $\Box \bigwedge_i (p2^{(i)} \Rightarrow \Diamond p8^{(i)})$.

Preuve : Considérons le cas du processus P_i. Lorsque P_i commence à exécuter son pré-protocole, il exécute `p2`, puis `p3` (ce calcul termine...), puis `p4` et commence les itérations de `p6` et `p7`. Il ne peut bloquer qu'en `p7` (car tous les calculs de `Max` terminent). Supposons que P_i bloque en `p7` à l'itération j, donc sur l'instruction :

```
await( Nb[j]==0 or (Nb[i],i) << (Nb[j],j) )
```

Comme il n'y a pas d'interblocage, les processus ayant des numéros plus prioritaires que P_j finiront par accéder puis quitter leur SC, et P_j accédera un jour à sa SC (avant P_i), puis :

- soit P_j restera en section non-critique et alors `Nb[j]==0`, ce qui libérera P_i de son blocage...
- soit P_j réexécutera son pré-protocole mais alors le numéro `Nb[j]` qu'il choisira sera strictement supérieur à `Nb[i]` et donc là aussi, P_i sera débloqué...

□

L'algorithme de la Boulangerie assure aussi une attente bornée : le nombre de processus qui peuvent passer devant P_i après que P_i ait demandé à accéder à sa SC et calculé son `Nb`, est borné par $n-1$: après un éventuel accès à la SC, tout processus P_j se verra attribuer (si il demande à y retourner) un numéro supérieur à celui de P_i...

Un inconvénient de l'algorithme de la Boulangerie est souligné dans l'exercice ci-dessous : les numéros renvoyés par le `Max` peuvent croître indéfiniment. Il existe des améliorations de l'algorithme de la Boulangerie qui évitent ce problème.

Exercice 1 *Montrer que les numéros utilisés par l'algorithme de la Boulangerie ne sont pas bornés. Donner un exemple.*

7 Algorithmes distribués utilisant des messages

Maintenant nous allons considérer le cas des algorithmes distribués communiquant par envois/réceptions de messages. Nous considérons des communications **asynchrones** : un envoi de message est suivi (plus tard) par une réception.

Les différents processus sont distribués sur différentes machines (on parle de sites), chacun a sa mémoire locale qu'il ne partage pas avec les autres. Chaque processus peut envoyer des messages à n'importe quel autre processus : un message est typé (sa structure est fixée). L'envoi de messages de la part d'un processus P permet de signaler aux autres processus des changements dans l'état de P.

Sur chaque site, on va distinguer deux parties du processus : une sera chargée de représenter l'activité du processus (pour accéder à la SC) et l'autre sera chargée de la réception des messages des autres processus. Ces deux parties évoluent en parallèle (en partageant la mémoire du site).

Nous allons faire les hypothèses suivantes :
- un site ne peut pas s'arrêter complètement : il doit toujours être capable d'envoyer et recevoir des messages ;
- chaque site peut envoyer (et recevoir) des messages à (de) n'importe quel autre site ;
- les canaux sont fiables : aucun message n'est perdu ;
- le temps de communication (le délai séparant l'envoi de la réception d'un message) est arbitraire (mais fini !).

7.1 Algorithme de Lamport (1978)

Hypothèses. Dans cet algorithme (voir par exemple [2]), on ne connaît pas le temps nécessaire à la transmission d'un message (entre son envoi et sa réception), mais on suppose toujours qu'entre deux processus donnés, deux messages ne peuvent pas se doubler : si P_i envoie un message M_1 à P_j, puis un autre message M_2, alors on sait que P_j recevra d'abord M_1 puis M_2.

Idée de l'algorithme. Chaque processus P_i dispose d'une horloge locale c_i (ce mécanisme est expliqué ci-dessous) que l'on va utiliser pour dater les messages. Un processus qui veut accéder à sa section critique va signaler son intention à tous les autres processus en envoyant un message `request` accompagné de la valeur de son horloge locale. Chaque processus souhaitant accéder à la SC dispose donc de toutes les demandes en cours des autres processus et sait si sa propre demande est la plus ancienne ou non : pour accéder à la SC, il suffit à P_i d'attendre que sa demande soit la plus ancienne de toutes les demandes en cours. Et bien-sûr tout processus quittant la SC doit le signaler avec un message `release` adressé à tous les processus.
Comme dans cet algorithme, on ne connaît pas le temps nécessaire à la transmission d'un message, il faut vérifier, avant d'accéder à la SC, qu'aucun message « ancien » n'est en cours de transmission, pour cela on impose au processus accédant à la SC d'avoir reçu un message « récent » de la part de tous les autres processus. Afin de garantir l'émission de messages récents (autres que les demandes d'accès ou les libérations de SC), on demande à chaque processus d'envoyer un accusé de réception `ack` à chaque fois qu'il reçoit une demande `request` d'un autre processus.

Evolution des horloges locales. Un problème courant dans les algorithmes distribués est d'ordonner les différents évènements (envoi ou réception de messages) : chaque processus a une vision locale qu'il se construit à partir de son comportement et des messages reçus. Chaque processus de l'algorithme va disposer d'une horloge locale qui est incrémentée de 1 à chaque envoi de messages. Mais ce mécanisme peut engendrer un décalage très grand entre les valeurs des horloges des différents processus. Pour limiter ce décalage, on impose à chaque processus P_i qui reçoit un message de P_j daté par la valeur t (t est la date locale à P_j au moment de l'envoi du message) de comparer la valeur de son horloge c_i à t : si $c_i < t$ alors P_i met à jour son horloge avec la valeur t. De plus, P_i incrémentera son horloge après chaque envoi (ou diffusion groupée) et réception de messages.

Description de l'algorithme. Tout d'abord, on précise les objets manipulés par les processus. Les horloges sont des entiers positifs. Les messages seront des triplets (type, val. d'horloge, num. de proc.) où le type peut être `request`, `release` ou `ack`. Chaque processus dispose d'un tableau `F[1..n]` pour stocker les derniers messages « importants » (voir ci-dessous) arrivés pour chaque processus. Dans son `F[i]`, le processus P_i stockera son dernier message de type `request` ou `release`.

Lorsque P_i reçoit un message (`request`$, t, j$) ou (`release`$, t, j$), il le stocke en `F[j]`. Lorsqu'il reçoit un message (`ack`$, t, j$), il le stocke en `F[j]` seulement si `F[j]` ne contient pas de message de type `request`.

Notons que les dates des messages contenus dans `F[j]` sont croissantes (pour tout `j`) : un message avec une date t est remplacé par un message avec une date t' telle que $t < t'$.

La description de l'algorithme se trouve dans les figures 14 et 15. On distingue la partie « gestion des réceptions de messages » (figure 15) du reste de l'algorithme (figure 14).

On note `Send(request,c,i) -> j` l'envoi du message `(request,c,i)` au processus `j`. La fonction `type(m,t,k)` retourne le type `m` du message et la fonction `date(m,t,k)` retourne la date `t` associée à un message.

```
Algorithme de Lamport (1978)

-- Processus Pi
Message F[1..n] := nil
int c := 0
loop forever :
p1: section NC
p2: { for all j != i : Send(request,c,i) -> j
p3:   F[i] := (request,c,i)
p4:   c := c+1 }
p5: await [ (date(F[i]) << date(F[j]) for all j != i ]
p6: section critique
p7:  { for all j!=i : Send(release,c,i) -> j
p8:    F[i] := (release,c,i)
p6:    c := c+1  }
```

FIGURE 14 – Algorithme de Lamport 1978

```
Algorithme de Lamport (1978) -- gestion des RECEPTIONS du processus Pi

case (request,t,j)  :
 { if (t > c) : c:=t
   c := c+1
   F[j] := (request,t,j)
   Send(ack,c,i) -> j  }

case (release,t,j)  :
 { if (t > c) : c:=t
   c := c+1
   F[j] := (rel,t,j)  }

case (ack,t,j)  :
 { if (t > c) : c:=t
   c := c+1
   if (type(F[j]) != request) : F[j] := (ack,t,j)  }
```

FIGURE 15 – Algorithme de Lamport 1978 – gestion des réceptions

Remarque : Dans le code des figures 14 et 15, on utilise des « $\{\ldots\}$ » pour désigner des instructions que l'on fait de manière atomique localement : le code de la partie principale et celui de la gestion des réceptions d'un même processus ne peuvent pas alors s'entremêler (pour garantir cette exclusion mutuelle on peut utiliser les algorithmes vus précédemment basé sur le partage de la mémoire comme Peterson). On verra pourquoi cette hypothèse est nécessaire.

L'algorithme de Lamport-1978 assure l'exclusion mutuelle :

Théorème 10 *L'algorithme de Lamport 1978 assure l'exclusion mutuelle de la section critique : la propriété* $\Box \bigwedge_{i \neq j} \neg(\texttt{p6}^{(i)} \wedge \texttt{p6}^{(j)})$ *est vérifiée.*

Preuve : Supposons que les processus P_i et P_j soient au même moment en `p6`. Supposons qu'à cet instant, on ait `date(F`^{i}`[i]) << date(F`^{j}`[j])` où `F`^{k} désigne le tableau du processus k. Alors c'est le processus P_j qui n'a pas respecté le protocole. En effet, P_j a envoyé une requête au temps t, et P_i a lui envoyé une requête au temps t' avec $t' < t$ ou $(t = t' \wedge i < j)$ (*i.e.* $(t', i) << (t, j)$). Si P_j accède à sa SC, c'est qu'il a stocké un message « récent » de tous les processus, et donc en particulier de P_i, dans son `F`^{j}`[i]`, ce message est de la forme :

- `(request,`t''`,i)` ou `(release,`t''`,i)` avec $(t, j) << (t'', i)$: alors on a $t' < t''$, le processus P_i a donc envoyé le message `(request,`t''`,i)` ou `(release,`t''`,i)` **après** le message `(request,`t'`,i)` : d'après le code du processus P_i, il n'est pas possible que `(request,`t'`,i)` soit encore stocké dans `date(F`^{i}`[i])`...
- `(ack,t'',i)` avec $(t, j) << (t'', i)$: alors on sait que le message `(request,`t'`,i)` a été reçu par P_j avant `(ack,t'',i)` (car par hypothèse les messages ne se doublent pas). Or

si ce dernier message a été stocké dans `date(F`^{j}`[i])` c'est qu'il n'y avait pas de message de type `request` (voir la gestion des réceptions) dans `F`^{j} mais un message `release` signifiant que P_i avait quitté sa SC.

□

Exercice 2 – *Montrer qu'il n'y a pas d'interblocage.*
– *Montrer qu'il n'y a pas de famine.*
– *Et l'attente bornée ?*
– *Combien de messages sont nécessaires pour l'accès d'un processus à la section critique ?*

Pour quelles propriétés, on utilise le mécanisme de « synchronisation » des horloges (mise à jour avec les valeurs d'horloges reçues dans les messages) ?

7.2 Algorithme de Ricart-Agrawala (1981)

L'idée de cet algorithme est assez proche de l'algorithme précédent mais on va essayer de réduire le nombre de messages nécessaires à l'accès de la section critique : lorsqu'un processus P_i communique à P_j son souhait d'accéder à la SC, P_j ne lui retourne un accusé de réception `ok` que si P_j ne souhaite pas y accéder ou si P_j n'est pas prioritaire (voir ci dessous). Dans le cas où P_j ne répond pas tout de suite, il le fera après avoir accédé à la SC (il utilise un tableau `Waiting` pour stocker la liste des processus à qui il doit envoyer un `ok` plus tard).

La priorité d'un processus est établie par un numéro que chaque processus choisit lorsqu'il souhaite accéder à sa SC. **Plus ce numéro est petit, plus sa demande d'accès à la SC est prioritaire.** Ce numéro est choisi supérieur à tous les autres numéros que le processus a vu pour le moment (on utilise pour cela une variable `maxnb`).

Un processus n'accède alors à sa SC que lorsqu'il a reçu un accord explicite de tous les autres processus ($N-1$ messages).

Cet algorithme utilise deux types de message : `request` (comme précédemment) et `ok` qui sert à donner son accord à un processus pour qu'il accède à la SC. Notons enfin qu'ici nous ne faisons pas l'hypothèse que les messages arrivent dans l'ordre : si P_i envoi un premier message à P_j, puis un second, alors l'ordre d'arrivée n'est pas fixé.

Remarque : Dans le code des figures 16 et 17, on utilise des « $\{\ldots\}$ » pour désigner des instructions que l'on fait de manière atomique : le code de la partie principale et celui de la gestion des réceptions ne peuvent pas alors s'entremêler (pour garantir cette exclusion mutuelle on peut utiliser les algorithmes vus précédemment basé sur le partage de la mémoire comme Peterson). On verra pourquoi cette hypothèse est nécessaire.

Théorème 11 *L'algorithme de Ricart-Agrawala assure l'exclusion mutuelle de la section critique (sous hypothèse de respect des blocs atomiques mentionnés dans l'algorithme) : la propriété* $\Box \bigwedge_{i \neq j} \neg(p\gamma^{(i)} \wedge p\gamma^{(j)})$ *est vérifiée.*

Preuve : Supposons que P_i et P_j se trouvent ensemble dans leur section critique. Il se sont donc chacun envoyés un message `ok`. Ils ont aussi chacun choisit une priorité : pr_i pour P_i et pr_j pour P_j.

Pour P_i on distingue les dates suivantes :
– t_i^0 où P_i choisit son numéro `pr` ;

```
Algorithme de Ricart-Agrawala

-- Processus Pi
boolean Waiting[1..n] := False
boolean reqCS :=False
int pr := 0
int maxpr := 0      // plus grand numero reçu
int nba:=0          // compte le nb de réponses attendues

loop forever :
p1: section NC
p2: { reqCS := True
P3:   nba := N-1
p4:   pr := maxpr +1 }
P5: for all j != i : Send(request,pr,i) -> j
p6: await [ nba == 0 ]
p7: section critique
p8: { reqCS := False
p9:   for all j: if (Waiting[j] == True) :
p10:          Waiting[j] := False
p11:          Send(ok,i) -> j }
```

FIGURE 16 – Algorithme de Ricart-Agrawala

```
case (request,k,j)  :
  { if (k > maxpr) :  maxpr := k
    if (!reqCS or (k,j)<<(pr,i)) : Send(ok,i) -> j
      else : Waiting[j] := True }

case (ok,j)  :
  nba := nba - 1
```

FIGURE 17 – Algorithme de Ricart-Agrawala – gestion des réceptions

- t_i^1 où P_i envoie le message `(request,i,pr)` à P_j ;
- t'_j où P_j reçoit `(request,i,pr)` et retourne `(ok,j)` à P_i (NB : en raison des hypothèses d'atomicité, on sait que cet instant existe : la réception du message de P_i est suivie immédiatement – sans l'exécution d'instructions de la procédure principale de P_j – par l'envoi de `ok` à P_i) ;
- t_i^2 où P_i reçoit le message `(ok,j)` de P_j.

On définit de même t_j^0, t_j^1, t'_i et t_j^2 pour le processus P_j. On a clairement : $t_i^0 < t_i^1 < t'_j < t_i^2$, et $t_j^0 < t_j^1 < t'_i < t_j^2$.

Supposons $t_i^0 < t_j^0$. Alors on distingue deux cas :

– $t'_j < t^0_j$: alors le choix de pr_j sera fait après la mise à jour de `maxpr` dans P_j et donc on aura $(pr_j, j) >> (pr_i, i)$. En conséquence, il n'est pas possible que P_i envoie un message `ok` à P_j avant de quitter sa SC... Il ne peut y avoir de conflit en section critique de cette manière.
– $t^0_j < t'_j$: alors on a soit $(pr_j, j) >> (pr_i, i)$ ou $(pr_i,) >> (pr_j, j)$. Dans le premier cas, P_j ne pourra recevoir de message `ok`, et dans le second c'est P_i qui n'en recevra pas de la part de P_j. Dans les deux cas, un des deux processus devra attendre que l'autre sorte de la section critique.

Le cas $t^0_j < t^0_i$ est similaire.

Il n'y a donc pas de conflit en section critique. □

L'absence de famine est aussi garantie :

Théorème 12 *L'algorithme de Ricart-Agrawala garantit l'absence de famine sous hypothèses...*
– *d'équité entre processus,*
– *de terminaison des sections critiques,*
– *de gestion ininterrompue des réceptions de messages, et*
– *de respect des blocs atomiques mentionnés dans l'algorithme.*

Alors l'algorithme vérifie : $\Box \bigwedge_i (p2^{(i)} \Rightarrow \Diamond p7^{(i)})$.

Preuve : Supposons que le processus P_i soit en situation de famine. Soit pr_i sa priorité. Si P_i reste bloqué, c'est forcément en `p6` : il attend d'avoir une réponse de chaque processus [1].

Il existe donc un processus P_j qui ne donne jamais son accord à P_i. Si tel est le cas, c'est que P_j veut aussi accéder à sa SC (ceci est nécessaire pour différer l'envoi de `ok`). Et si P_j bloque pour toujours P_i, c'est que lui aussi est bloqué : car si P_j accédait à sa SC, il la terminerait et finirait par envoyer `ok` à P_i...

Notons au passage, que les autres processus finiront par être aussi bloqués lors de leur prochain pré-protocole : ils ne recevront plus de message `ok` de P_i et P_j car ils auront des numéros supérieurs à pr_i et pr_j... Tout le monde finira donc bloqué (ou en section non-critique).

Soit P_m le processus bloqué dans son pré-protocole ayant la plus petite priorité pr_m (*i.e.* minimale pour $<<$) . Soit P_k un autre processus bloqué qui n'a pas envoyé son message `ok` à P_m. Pourquoi P_k a-t-il bloqué P_m ? C'est qu'au moment de la réception de `(request,`pr_m`,m)`, P_k voulait accéder à la SC (`reqCS==True`) et que son pr était inférieur (pour $<<$) à pr_m (NB : c'est ici que l'hypothèse d'atomicité du bloc `p2`,`p3`,`p4` est utilisée). Mais depuis P_k a changé de pr (puisque pr_m est le minimum – l'égalité est impossible grâce à la relation $<<$), c'est donc qu'il est passé en SC et donc qu'il a exécuté la boucle `p9` et donc envoyé un message `ok` à P_m. Il y a bien contradiction : P_m ne peut pas être bloqué... □

Dans cet algorithme, une demande d'accès en SC demande $2(n-1)$ messages.

Exercice 3 *Donner une exécution du protocole où le non respect des hypothèses d'atomicité conduit à un blocage et une famine.*

On peut trouver une preuve automatique de cet algorithme dans [3] (accessible sur le web).

1. son compteur `nba` décroît à chaque réception d'un `ok` et il est aisé de voir que l'envoi d'un message `ok` ne se fait qu'à la réception d'un message `request` dans le bloc de gestion des messages, ou après la section critique **sinon** : un processus ne peut pas envoyer plus de `ok`.

A Abstraction et diagramme d'états

La taille du diagramme d'états est un problème majeur de l'analyse des algorithmes concurrents. Il peut être exponentiel dans le nombre de processus évoluant en parallèle et dans le nombre de variables... Il est donc important d'*abstraire* le plus possible ce graphe, en ne conservant que les parties pertinentes vis-à-vis de la propriété à vérifier. Ce travail n'est pas toujours facile à faire. Nous allons l'illustrer de deux manières sur l'exemple de l'algorithme A1 de la section 5.1.

Tout d'abord, nous pouvons simplifier l'algorithme en considérant celui de la figure 18.

```
            int turn := 1      // variable partagée


-- Processus P1                          -- Processus P2
loop forever :                           loop forever :
p1': await turn==1                       q1': await turn==2
p2': turn:=2                             q2': turn:=1
```

FIGURE 18 – Algorithme A1'

En effet, pour la propriété d'exclusion mutuelle, on ne s'intéresse pas à ce qui se passe dans les sections critiques et les sections non-critiques. Par exemple, si ces deux parties du code étaient vides, alors le protocole devrait toujours fonctionner (on sait que ces parties ne peuvent modifier la variable `turn`). On peut donc se limiter aux deux phases du protocole. Le diagramme d'états de A1' est alors décrit à la figure 19.

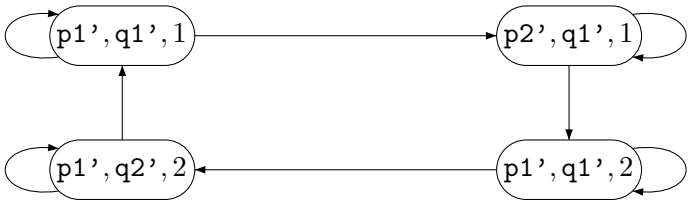

FIGURE 19 – Diagramme d'états de A1'.

Comment vérifier la propriété d'exclusion mutuelle ? En vérifiant que les états (`p2'`, `q2'`, 1) et (`p2'`, `q2'`, 2) ne sont pas accessibles depuis l'état de départ... Et c'est bien le cas !

Mais on ne peut pas utiliser ce graphe pour vérifier la propriété d'absence de famine pour A1. En effet, dans A1', on ne représente plus le fait que les SNC peuvent ne pas terminer... Or c'est précisément cela qui rend possible la famine des processus avec A1. Et ce n'est plus le cas dans A1' : cet algorithme vérifie bien l'absence de famine (on le constate dans le graphe de la figure 19 où le long de toutes les exécutions équitables, les deux processus accèdent infiniment souvent à la SC). Il faut donc être très prudent lorsqu'on simplifie (abstrait) un algorithme : il faut s'assurer que cette abstraction est compatible avec les propriétés étudiées.

Un autre graphe abstrait. Enfin, nous pouvons aussi obtenir un graphe simplifié représentant le comportement de l'algorithme A1 de la manière suivante. Il suffit de partir d'un autre point de vue : puisque l'objectif est d'analyser les accès à la section critique, nous

pouvons représenter un état par un triplet $(\epsilon_1, \epsilon_2, t)$ où ϵ_1 est un booléen valant vrai ssi P_1 est dans sa SC, et ϵ_2 vaut vrai ssi P_2 est dans sa SC, et t représente la valeur de la variable `turn`. A partir de ces triplets, on peut décrire le comportement du programme A1 avec le graphe étiqueté de la figure 20.

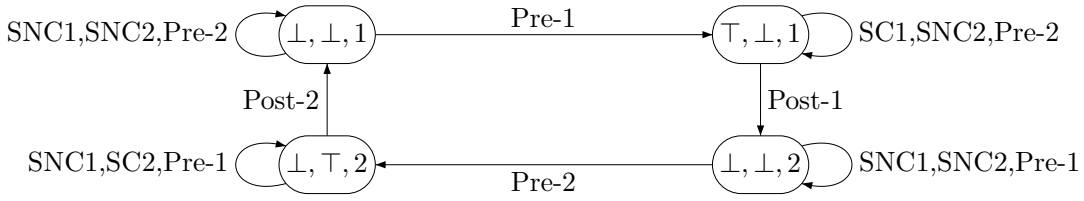

FIGURE 20 – Graphe simplifié de A1.

L'idée des transitions étiquetées est d'indiquer quelles instructions du programme sont effectuées lors de la transition. L'étiquette « Pre-1 » signifie que le processus P_1 effectue son Pre-protocole, c'est-à-dire l'instruction « `await turn==1` » (qui a pour effet de boucler si le test est négatif, ou de passer en section critique sinon).

Ce graphe peut servir pour prouver l'exclusion mlutuelle ainsi que la présence de famine (en considérant le cycle SNC1 / Pre-2 sur le premier état : si le processus P_1 ne sort jamais de sa section non-critique, il ne permettra jamais à P_2 d'accéder à sa SC...).

Notons que ce graphe simplifié n'est pas à proprement parler un diagramme d'états comme nous les avons définis au premier cours. Il s'agit d'un autre type de construction qui décrit aussi le comportement du programme et qui peut servir pour analyser certaines propriétés. Cela montre qu'il existe plusieurs méthodes alternatives possibles mais à chaque fois il faut être sûr que la construction est pertinente pour la propriété étudiée : un graphe trop simplifié ne marchera pas... Ce travail de construction d'un modèle (diagramme d'états abstrait) est un problème délicat !

Exercice 4 *Construisez le graphe simplifié (comme celui de la figure 20) pour l'algorithme A2 de la figure 21. Quelles propriétés vérifie A2 ?*

```
boolean D1 := False      // variables partagées
boolean D2 := False
```

```
-- Processus P1                    -- Processus P2
loop forever :                     loop forever :
p1: section NC                     q1: section NC
p2: D1 := True                     q2: D2 := True
p3: await (D2==False)              q3: await (D1==False)
p4: section critique               q4: section critique
p5: D1:=False                      q5: D2:=False
```

FIGURE 21 – Algorithme A2

B Logique temporelle – mini kit de survie

Ici nous utilisons la logique temporelle LTL pour énoncer des propriétés sur le comportement des processus concurrents.

LTL est une logique temporelle de temps linéaire : les formules de LTL s'interprètent sur des *exécutions étiquetées* : Une exécution ρ est une séquence infinie d'états $\rho : q_0 \to q_1 \to q_2 \to \ldots$ et chaque état q_i est étiqueté par un ensemble de *propositions atomiques* $\ell(q_i)$ (ℓ est appelée la fonction d'étiquetage) qui correspondent à l'ensemble des propriétés « élémentaires » que cet état vérifie.

On note $\rho(i)$ le $(i+1)$-ème état de ρ (dans l'exemple ci-dessus on a $\rho(i) = q_i$) et on note ρ^i le $(i+1)$-ème suffixe de ρ, c'est-à-dire $q_i \to q_{i+1} \to \ldots$

Dans notre cas, on va interpréter les formules de LTL sur des exécutions du diagramme d'états (ou d'un graphe équivalent) associé à un algorithme. On a donc une séquence d'états $q_0 \to q_1 \to q_2 \to \ldots$ où chaque transition correspond à un pas de l'algorithme et chaque état est une structure de la forme $(\texttt{p-}, \texttt{q-}, \texttt{v1}, \texttt{v2}, \ldots)$: on sait pour chaque processus quelle est la prochaine instruction qui sera exécutée, et on connaît la valeur de certaines variables. Ces caractéristiques de chaque état sont vues comme des « propositions atomiques » de la logique temporelle : c'est à dire des propositions qui s'interprètent sur un état (et non sur l'ensemble de l'exécution) : ainsi la proposition p2 sera vraie dans un état de la forme $(\texttt{p2}, \ldots)$.

Un exemple d'exécution du diagramme d'états de l'algorithme A1 est :

$$(\texttt{p1},\texttt{q1},1) \to (\texttt{p1},\texttt{q1},1) \to (\texttt{p1},\texttt{q2},1) \to (\texttt{p2},\texttt{q2},1) \to (\texttt{p3},\texttt{q2},1) \to (\texttt{p3},\texttt{q2},1) \to (\texttt{p4},\texttt{q2},1)\ldots$$

Dans cet exemple, l'ensemble des propositions atomiques sera $\{\texttt{p1}, \texttt{p2}, \texttt{p3}, \texttt{p4}, \texttt{q1}, \texttt{q2}, \texttt{q3}, \texttt{q4}, \texttt{turn}{=}{=}1, \texttt{turn}{=}{=}2\}$. La proposition p3 sera vraie dans tout état de la forme $(\texttt{p3}, \texttt{q-}, -)$, la proposition turn==2 sera vraie dans tout état de la forme $(\texttt{p-}, \texttt{q-}, 2)$, *etc.*

La syntaxe de la logique LTL est définie comme suit :

Définition 2 (syntaxe de LTL)

$$\phi, \psi ::= P \mid \phi \wedge \psi \mid \neg\phi \mid \phi \,\mathsf{U}\, \psi \mid \bigcirc\phi$$

où P est une proposition atomique.

Les opérateurs booléens s'interprètent comme d'habitude. Une exécution ρ munie d'une fonction d'étiquetage ℓ vérifie une proposition atomique P si l'état $\rho(0)$ contient P dans son étiquetage (*i.e.* $P \in \ell(\rho(0))$). L'opérateur U (pour « until ») est utilisé de la manière suivante : $\phi \mathsf{U} \psi$ est vraie pour une exécution ρ ssi il existe un suffixe de ρ vérifiant ψ et tous les suffixes précédents vérifient ϕ (on a bien ϕ jusqu'à avoir ψ !). L'opérateur $\bigcirc$ permet de parler du suffixe obtenu après une transition.

On va maintenant définir formellement cette sémantique. Etant donnée une exécution $\rho : q0 \to q_1 \to q_2 \to \ldots$, on note $\rho \models \phi$ pour signifier que la formule ϕ est vraie pour ρ. On définit cette relation comme suit :

- $\rho \models P$ ssi $P \in \ell(\rho(0))$;
- $\rho \models \phi \wedge \psi$ ssi $\Big(\ \rho \models \phi$ et $\rho \models \phi\ \Big)$;
- $\rho \models \neg\phi$ ssi il n'est pas vrai que l'on a : $\rho \models \phi$;
- $\rho \models \phi \,\mathsf{U}\, \psi$ ssi $\Big(\ \exists i \geq 0$ tq $\rho^i \models \psi$ et $\forall 0 \leq j < i$ on a : $\rho^j \models \phi\ \Big)$

– $\rho \models \bigcirc \phi$ ssi $\rho^1 \models \phi$.

On définit quelques « macros » :
– $\phi \vee \psi$ par $\neg(\neg\phi \wedge \neg\psi)$;
– $\phi \Rightarrow \psi$ par $(\neg\phi) \vee \psi$;
– $\top$ par $P \vee \neg P$ pour n'importe quelle proposition atomique P : $\top$ est toujours vrai (pour n'importe quel état de n'importe quelle exécution) ;
– $\Diamond\phi$ par $\top \mathsf{U} \phi$: « un jour (dans le futur), ϕ sera vraie » (en suivant la définition du Until, on exige qu'il existe un suffixe vérifiant ϕ et que tous les suffixes précédents vérifient $\top$ – ce dernier point étant toujours vrai).
– $\Box\,\phi$ par $\neg\Diamond\neg\phi$: « ϕ est toujours (dans le futur) vraie ».

Si l'on reprend l'exemple d'exécution donnée ci-dessus (pour l'algorithme A1), alors la propriété $\Diamond(\texttt{p3} \wedge \texttt{q2})$ est vérifiée. Il en est de même pour la formule $(\texttt{turn} == 1)\mathsf{U}\texttt{p3}$.

Que signifie qu'un programme vérifie une formule de LTL ? Cela signifie que **toutes** les exécutions de ce programme vérifient la formule.

Dans le cours, nous utiliserons essentiellement les opérateurs $\Diamond$ et $\Box$ (et les opérateurs booléens). Cela permet d'énoncer des propriétés classiques comme :
– des **propriétés de sûreté** qui expriment l'idée qu'une « mauvaise chose » n'arrive jamais.
  Par exemple $\Box\,\neg(\texttt{p8} \wedge \texttt{q8})$: il est toujours vrai que P_1 et P_2 ne peuvent pas être dans leur 8-ème instruction au même instant... C'est équivalent à écrire : $\neg\Diamond(\texttt{p8} \wedge \texttt{q8})$ ou encore $\Box\,(\neg\texttt{p8} \vee \neg\texttt{q8})$.
– des **propriétés de vivacité** qui expriment l'idée qu'une « bonne chose » doit arriver (dans certaines circonstances).
  Par exemple $\Box\,(\texttt{p2} \Rightarrow \Diamond\texttt{p8})$: il est toujours vrai que si P_1 est en p2, alors plus tard il accédera à sa 8-ème instruction (par ex. sa section critique)...
– des **propriétés d'équité** qui expriment une forme d'équité sur l'exécution.
  Par exemple, on pourrait dire que si un processus demande infiniment souvent d'accéder à sa section critique, alors il y parviendra infiniment souvent... Notez que c'est moins fort que la propriété précédente car avec cette nouvelle propriété, on peut imaginer un processus qui demande 100 fois d'accéder à sa SC et qui y renonce après quelque temps et finit par ne plus la demander. Ce type d'exécutions où la SC n'est jamais obtenue vérifierait la propriété d'équité mais pas celle de vivacité ci-dessus.
  Si la demande d'accès à sa SC se fait à l'instruction p15 et que sa SC se trouve en p23, on peut écrire la propriété d'équité décrite ci-dessus par : $(\Box \Diamond\, \texttt{p15}) \Rightarrow (\Box \Diamond\, \texttt{p23})$

Ce ne sont que quelques exemples, la logique temporelle permet d'énoncer des propriétés très fines sur les exécutions d'un programme. Mais nous ne l'utiliserons que de manière très simple dans tout le cours...

Références

[1] Moti Ben-Ari. *Principles of Concurrent and Distributed Programming.* Addison-Wesley, second edition, 2006.

[2] Michel Raynal. *Algorithmique du parallélisme, le problème de l'exclusion mutuelle.* Dunod, 1984.

[3] Ekaterina Sedletsky, Amir Pnueli, and Mordechai Ben-Ari. Formal verification of the ricart-agrawala algorithm. In *Proceedings of the 20th Conference of Foundations of Software Technology and Theoretical Computer Science (FSTTCS'2000)*, volume 1974 of *Lecture Notes in Computer Science*, pages 325–335. Springer, 2000.

[4] Gadi Taubenfeld. *Synchronization Algorithms and Concurrent Programming.* Prentice Hall, 2006.