Equité vs Famine

Raymond Devillers
Université Libre de Bruxelles

26 août 2012

L'équité demande que nul acteur n'est (trop) défavorisé par rapport à d'autres. En d'autres termes, lorsqu'un processus demande une ressource, il doit finir par l'avoir. Une ressource est tout ce dont un processus peut avoir besoin pour continuer son travail. On considérera essentiellement des ressources non-préemptibles, c'est-à-dire que l'on ne peut reprendre tant que le processus ne les a pas relâchées ; les ressources préemptibles (processeur, mémoire, ...) posent moins de problème car on connaît bien des techniques (round robin par exemple) qui assurent des formes d'équité.

Lorsqu'un processus est sans cesse repoussé, on parle parfois de "famine". Cela fait référence à un problème classique, introduit par Dijkstra, le problème des philosophes (ou du dîner des philosophes).

1.1 Problème des philosophes (Dijkstra)

Des philosophes (prenons-en quatre), sont autour d'une table (figure 1.1).

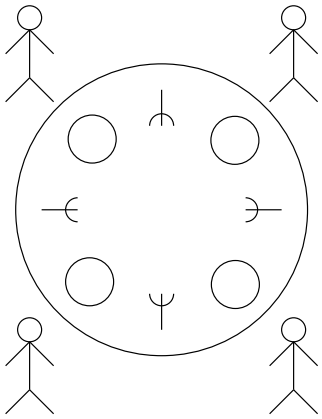

FIGURE 1.1 – Problème des philosophes.

Leur vie n'est qu'une succession de phases où ils mangent et où ils pensent. Evidemment, de temps à autre, ils ont faim. On leur donne du spaghetti à manger et ils le mangent avec deux fourchettes (les grands savants sont souvent excentriques !). Malheureusement pour eux, il n'y a jamais qu'une seule fourchette placée entre deux assiettes. Les philosophes ne pourront donc pas tous se nourrir en même temps (en fait, deux philosophes voisins ne pourront jamais manger simultanément).

Nous allons donc élaborer des protocoles qui vont permettre aux philosophes de se comporter correctement pour manger.

La vie d'un philosophe se résume donc à la boucle infinie suivante :

do
think
protocole d'entrée (PE)
eat
protocole de sortie (PS)
od

Avec cela, nous pouvons voir facilement que ça cache en fait un problème informatique : les philosophes représentent les processus, les fourchettes sont les ressources et les phases *eat* sont les phases au cours desquelles les processus utilisent les ressources.

Protocole 1

Voyons maintenant un premier protocole :

PE :
prendre fg // prendre fourchette gauche
prendre fd // prendre fourchette droite

PS :
reposer fg // reposer fourchette gauche
reposer fd // reposer fourchette droite

Nous voyons tout de suite que nous avons un gros problème avec ce protocole. Si tous les philosophes prennent leur fourchette gauche au même moment, tout le monde va réclamer ensuite sa fourchette droite qui est la fourchette gauche de quelqu'un d'autre. Autrement dit, nous avons un deadlock. Ce protocole n'est donc évidemment pas acceptable.

Hypothèses

Nous allons faire quelques hypothèses avant d'essayer un nouveau protocole.
- Les phases *eat* sont toujours finies (si ce n'est pas le cas, les voisins du philosophe risquent d'avoir faim longtemps !).
- Les philosophes ne peuvent pas disparaître en gardant leurs fourchettes (ils les déposent). Si un philosophe meurt, c'est comme s'il avait une phase *think* infinie. Il faut d'ailleurs remarquer que nous n'avons pas mis de délais, ni pour les phases *think*, ni pour les phases *eat*. Nous n'avons pas non plus de contraintes sur les moments où les philosophes meurent (et si un philosophe meurt, c'est discrètement).

Une stratégie simple consistant à numéroter les philosophes et à leur donner l'accés aux fourchettes (et à la table) à tour de rôle n'est dès lors pas acceptable. En effet, si le premier philosophe meurt tout de suite, les autres vont attendre indéfiniment. même si nous ne mettons pas de borne sur le temps (*deadline*), attendre un temps infini c'est quand même beaucoup ! :-)

Outre cela, nous allons essayer (même si ce ne sera pas toujours possible) d'avoir des solutions symétriques. Tout le monde devra donc se comporter de la même manière, c'est-à-dire que les protocoles d'entrée et de sortie devront étre identiques pour tous.

Une solution avec sémaphores

Le problème du premier protocole vient du fait que l'on prend séparément les deux fourchettes. Le deadlock ne survient pas si on fait des demandes en bloc (les deux fourchettes en même temps). Pour cela, Dijkstra a imaginé une solution utilisant des sémaphores (dont il est l'inventeur). Un sémaphore n'est rien d'autre qu'un entier et une file d'attente, et peut être apparenté à une ressource consommable. Dans notre cas, des sémaphores binaires suffiront.

A chaque philosophe i, on associe un indicateur C_i donnant son état :

$$\begin{cases} 0 & = & état\ think \\ 1 & = & état\ hungry \\ 2 & = & état\ eat \end{cases}$$

Chaque philosophe aura également un sémaphore privé Go_i. Seul ce philosophe peut faire des P sur son sémaphore mais tout le monde peut faire des V (donc tout le monde peut débloquer le processus éventuellement) [1].

Un sémaphore $Mutex$ (*Mutual Exclusion*) permet d'éviter que deux philosophes différents ne puissent entrer en même temps dans leurs sections critiques.

Nous avons au départ : $Mutex = 1$ et $\forall i : C_i = 0 \wedge Go_i = 0$.

1. P pour *Proberen* et V pour *Verhoogen*.

PE :

$P(Mutex)$
$C_i \leftarrow 1$ // j'ai faim ...
if $C_{i\ominus 1} \neq 2 \neq C_{i\oplus 1}$ **then** // opérations circulaires (modulo)
  $C_i \leftarrow 2$ // je réserve les fourchettes
  $V(Go_i)$ // je peux y aller
fi
$V(Mutex)$
$P(Go_i)$ // si un voisin mange, je reste coincé
prendre fg et fd // OK car fourchettes réservées en $C_i \leftarrow 2$

PS :

libérer fg et fd
$P(Mutex)$
$C_i \leftarrow 0$ // je "pense" au lieu de je "panse"...
if $C_{i\ominus 2} \neq 2 \wedge C_{i\ominus 1} = 1$ **then** // réveiller le voisin de gauche ?
  $C_{i\ominus 1} \leftarrow 2$
  $V(Go_{i\ominus 1})$
fi
if $C_{i\oplus 2} \neq 2 \wedge C_{i\oplus 1} = 1$ **then** // réveiller le voisin de droite ?
  $C_{i\oplus 1} \leftarrow 2$
  $V(Go_{i\oplus 1})$
fi
$V(Mutex)$

Voici donc un protocole nettement plus élaboré, mais est-ce qu'il fonctionne ? Pour cela, nous allons voir les différentes causes possibles aux problèmes de famine.

1. Les problèmes de deadlock. Ce problème-là est réglé avec ce protocole.
2. Mauvaise gestion de la file d'attente des sémaphores. Supposons par exemple, que l'on utilise LIFO pour gérer $Mutex$. Le deuxième philosophe pourrait attendre très longtemps avant de pouvoir avoir ses fourchettes (il ne saura même pas dire qu'il a faim !). Il faut donc une gestion intelligente et équitable (*fair*) de la file (même si la gestion ne doit pas être parfaitement équitable). Nous pourrions par exemple utiliser FIFO, ou d'autres systèmes de files à priorités (FIFO avec des degrés de liberté).
3. Les coalitions. Supposons que deux philosophes face à face décident de se coaliser contre les deux autres. Le premier mange et gène donc ses deux voisins. Il attend que son vis-à-vis commence à manger pour libérer les fourchettes.

> Et le philosophe d'en face fait de même.... Il faut remarquer que les coalitions peuvent être volontaires ou dues au hasard. Malheureusement, le dernier protocole n'empêche pas ce problème.

A part ça, nous pouvons noter que nous avons de l'expédience avec cette solution : les philosophes ne sont pas mis en attente inutilement. De plus, nous avons confinement du problème aux voisins du philosophe : si quelqu'un meurt pendant qu'il mange et qu'il garde ses fourchettes (relâchement des hypothèses de départ), seuls ses voisins seront ennuyés (mais fameusement !).

Nous allons devoir trouver des techniques permettant d'éviter ces problèmes de coalition. Dijkstra a donc imaginé une autre solution, dans un cas plus général. Plus exactement, il s'est proposé de caractériser toute une famille de solutions au problème.

1.2 Cas général de Dijkstra

Soit $\mathcal{V} \in 2^{2^P}$. $\mathcal{V}$ est l'ensemble des sous-ensembles de processus pouvant entrer simultanément dans leur section critique. Dans le cas du problème des philosophes, ou de tout autre problème à ressources non-interchangeables, il comprend en fait tous les ensembles de processus ne comportant pas deux processus voisins ; dans un cas plus général, il comprendra tous les ensembles de processus dont la somme (vectorielle) des besoins (des phases *eat*) est inférieure ou égale au vecteur $\bar{r}$ des ressources.

$\mathcal{V}$ ne pourra être quelconque. Une première propriété qu'il doit satisfaire est que $V' \subseteq V \in \mathcal{V} \Rightarrow V' \in \mathcal{V}$, càd que les sous-ensembles d'un ensemble de $\mathcal{V}$ appartiennent aussi à $\mathcal{V}$ (la somme des besoins est moindre).

Une deuxième propriété est que $\forall P : \{P\} \in \mathcal{V}$. Le système est donc raisonnable (P peut manger seul).

A chaque processus P_i, nous allons associer un compteur ac_i (*allowance counter*). Pour éviter d'avoir un problème de famine, il ne faut pas qu'un nombre infini de processus puisse passer avant soi-même (alors qu'on a faim). C'est le réle de ac_i. Il va être décrémenté d'une unité chaque fois que quelqu'un mange et quand il arrive à 0, il faut qu'on mange tout de suite. Si le processus pense ou mange, $ac_i = \infty$. Remarquons que lorsque quelqu'un mange, nous pouvons décrémenter tous les compteurs ($\infty - 1 = \infty$) ou seulement ceux des affamés.

Remarquons également que nous n'avons plus besoin de l'indicateur C_i. En effet, si le processus a faim, son ac_i est fini, et s'il mange ou s'il pense, son $ac_i = \infty$. Nous pouvons de plus voir qu'un processus mange car il a les fourchettes (ou plus généralement les ressources dont il a besoin pour manger).

Condition générale d'admissibilité (permission générale)

Pour qu'un processus puisse passer dans l'état *eat*, il faut et il suffit :

1. qu'il ait faim.
2. que $V \cup \{P_i\} \in \mathcal{V}$. V est l'ensemble des processus actuellement dans leur section critique (il faut que les "fourchettes" soient disponibles).
3. que la situation reste sûre et que $\forall i : ac_i \geq 0$.

Le dernier point mérite quand même quelques explications. Si deux ac_i sont à 0, personne ne peut manger. En effet, en mangeant, on décrémente tous les compteurs mais alors, au moins l'un d'eux va devenir négatif. Si l'on n'y prend garde, on remplace donc le problème de famine par un problème d'interblocage. Il nous faudra donc éviter les situations dangereuses. Nous avons donc des situations sûres et non sûres.

Ces trois points forment la condition générale d'admissibilité, ou permission générale. Cela permet en fait à un processus qui a faim de manger, mais il n'est pas toujours obligé de le faire tout de suite : on n'impose pas l'expédience.

Il faut se rendre compte que les compteurs sont une forme de ressource mais que ce sont des ressources consommables, avec un schéma de production et de consommation bien particulier. Il n'y a donc pas de stratégie générale pour éviter les deadlocks. Il faut au contraire étudier les problèmes au cas par cas.

Obligation spécifique

Dijkstra introduit encore une autre condition que doivent vérifier ses stratégies : s'il y a un processus affamé, il existe au moins un processus dans sa phase *eat* ou dans son protocole de sortie. Supposons que cela ne soit pas le cas : tous les processus pensent ou sont affamés (et il y a des affamés). Mais si les penseurs pensent tous à l'infini, alors la situation ne changera plus jamais (car c'est dans les protocoles d'entrée et de sortie qu'on peut décider de faire manger quelqu'un) et des processus restent affamés. Ce n'est donc pas acceptable.

- Si un processus devient affamé et que tous les autres pensent, il doit manger.
- Si un processus quitte la table et qu'il était le dernier à manger, il doit réveiller quelqu'un (s'il y a des processus qui sont affamés). Il faut remarquer qu'il ne doit pas forcément réveiller tout le monde.

Nous appelerons cette condition l'obligation spécifique. Il s'agit donc d'une forme très affaiblie d'expédience (on n'est pas obligé de faire manger sauf si on est le dernier à manger ; on n'est pas obligé de manger, sauf si on est le seul affamé).

1.2.1 Critère de sûreté

Supposons que nous triions les compteurs (ac_i) des processus (par ordre croissant).

0, 0, 1, 2, ...
0, 1, 1, 4, ...
2, 2, 2, 2, 5, ...
1, 2, 3, 3, 3, 8, ...

Nous pouvons voir assez facilement que ces séquences ne sont pas sûres. Nous sommes certain d'avoir des deadlock car plusieurs processus vont avoir, en même temps, leur compteur à 0. Pour que la situation soit sûre, il faut trouver une séquence permettant d'admettre les affamés les uns à la suite des autres de sorte qu'aucun compteur ne devienne négatif.

Condition de sûreté

La condition de sûreté devient donc : $\exists$ séquence $ac_1, ac_2, \ldots, ac_n : \forall k\ ac_k \geq k-1$. Il faut que chaque compteur soit plus grand ou égal à son rang moins un (les rangs commencent à partir de 1). Le premier processus doit avoir un compteur au moins égal à 0, le deuxième doit avoir un compteur au moins égal à 1, etc.

Nous pouvons dire cela autrement : il faut que le compteur d'un processus soit au moins égal au nombre de processus avant lui. Si nous avons trouvé une séquence répondant à cette condition, nous pouvons toujours nous en sortir en prenant les processus dans l'ordre de cette séquence. Il s'agit d'une condition nécessaire et suffisante.

Nous avons en fait trois conditions équivalentes (c'est assez facile à prouver) :

$$\begin{gathered}\exists \text{ séquence } \forall i : ac_i \geq i-1 \\ \Updownarrow \\ \forall K \geq 0 : |\{i : ac_i < K\}| \leq K \\ \Updownarrow \\ \text{dans la séquence triée}, \forall i : ac_i \geq i-1\end{gathered}$$

Le problème c'est maintenant de trouver une séquence valide (forme de séquence de réduction). Comme il suffit de considérer la séquence triée, on peut le faire en $\mathcal{O}(n^2)$ ou en $\mathcal{O}(n \log n)$. En fait, c'est moins que cela s'il ne faut pas construire la séquence mais seulement la maintenir triée.

Initialisation des compteurs

Et que se passe-t-il si un nouveau processus arrive ? Nous devons initialiser le compteur du petit nouveau, mais quelle valeur pouvons-nous lui donner ? Il faut

évidemment toujours respecter les contraintes précédemment citées.

S'il n'y a pas d'affamés, le nouveau processus A peut recevoir n'importe quelle valeur pour son compteur. Si par contre nous avons des affamés, alors nous savons qu'il y a au moins un mangeur (à cause de l'obligation spécifique). Dans le pire des cas, nous avons :

$$\underbrace{M}_{1 \text{ mange}} \; | \underbrace{0, 1, 2, 3, \cdots, n-3}_{n-2 \text{ affamés}} \; A$$

A doit se placer à la fin, et donc il faut que $ac_i \leftarrow V \geq n-2$. Il s'agit d'une condition nécessaire et suffisante (si V est fixée une fois pour toute au début).

Voila donc les conclusions de Dijkstra pour régler les problèmes génériques de famine. Mais nous allons maintenant faire quelques remarques.

1.2.2 Remarques sur la méthode de Dijkstra

1. Tout d'abord, il n'est pas fondamental d'avoir une solution parfaitement symétrique car le probléme n'est pas toujours symétrique (le problème des philosophes oui, mais pas le cas général) ; de plus, des solutions avec une petite asymétrie peuvent aussi être intéressantes, ou même nécessaire comme on le verra, et l'asymétrie peut résulter du résultat d'un tirage aléatoire "symétrique".
2. Malgré l'aspect général de la famille de stratégies définies (ou plutôt caractérisées car il n'y a pas d'implantation précise fournie) par Dijkstra, cela ne recouvre pas toutes les stratégies valides ; en particulier, on ne retrouve pas la stratégie FIFO, rigide mais bien connue ; nous allons voir comment l'obtenir à partir d'une extension des stratégies de Dijkstra.
3. Maintenant, nous pouvons nous demander si toutes les hypothèses, explicites ou implicites, de Dijkstra tiennent la route. Nous avons dit qu'il faut limiter le nombre de processus passant avant soi. C'est vrai pratiquement mais c'est faux théoriquement. En effet, une somme infinie peut être finie ($1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 2$). Cela signifie donc que si un processus mange suffisamment de plus en plus vite, il peut manger une infinité de fois avant nous. En réalité, c'est purement théorique car la durée de la phase *eat* est bornée inférieurement par le tick d'horloge ; la contrainte est donc raisonnable. Cependant, il n'est pas nécessaire de baser directement les stratégies là-dessus. Il suffit que cela soit *impliqué* par les stratégies choisies. Nous pourrions aussi essayer de compter (comme nous le verrons plus tard) autre chose que le nombre de processus passant avant soi, comme par exemple compter le nombre de fois où les voisins passent avant soi (il s'agit là d'une attaque plus indirecte). De plus, il faudrait peut-étre tenir compte du fait que certains processus peuvent manger simultanément.

4. Nous pouvons également affaiblir l'obligation spécifique. En effet, l'OS n'est pas compté parmi les processus, or il peut se charger lui-même de réveiller des processus affamés si les autres sont morts : l'obligation spécifique devient alors que l'OS ne peut pas laisser stable une situation où il n'y a que des penseurs et des affamés.

5. Il faut également regarder comment se passent les réveils. Ils peuvent se dérouler de manière synchrone ou asynchrone. Dans le cas asynchrone, un signal est lancé par le processus qui peut directement sortir de son protocole de sortie. Il faut évidemment être sûr que le signal arrivera bien à destination, mais entretemps le processus réveilleur peut lui-même être redevenu affamé : on ne peut donc plus se baser sur le fait qu'au moins un processus n'est pas affamé et la condition sur la valeur initiale devient $ac_i \leftarrow V \geq n-1$ (ou $ac_i \leftarrow V_i \geq n-1$ si l'on admet une asymétrie). Notons que c'est vrai également si c'est l'OS qui se charge des réveils.

$$\underbrace{0, 1, 2, 3, \cdots, n-2}_{n-1 \text{ affamés}}\ A$$

1.3 Stratégies dynamiques

Les stratégies de Dijkstra sont statiques car la valeur initiale du compteur est choisie a priori. On peut procéder autrement et choisir des valeurs de compteur différentes, du moment que la situation reste sûre.

Nous allons essayer de trouver des solutions dynamiques et symétriques. Dynamiques car les processus pourront choisir la valeur de départ de leur compteur chaque fois qu'ils deviendront affamés, mais symétriques car les processus détermineront tous cette valeur de la même manière (ils vont suivre le même algorithme de choix initial).

Le problème est donc de déterminer, lorsqu'un processus devient affamé, en fonction de l'état du système, c'est-à-dire des valeurs des autres compteurs, quelles valeurs initiales pourra prendre le compteur du nouvel affamé. Au passage, on détermine aussi qui peut commencer à manger tout en gardant une situation sûre, lorsque les ressources demandées sont disponibles. Nous allons repartir du critère ac_i *dans séquence triée* $\geq i-1$. Parmi tous ces compteurs, il y en a qui sont à la limite, c'est-à-dire dont le compteur est égal à leur rang moins un ($ac_k = k-1$) dans la séquence triée. Nous allons regarder le premier et le dernier de ces rangs, soient h et k. Nous avons donc $ac_h = h-1$ et $ac_k = k-1$. Nous pourrions avoir par exemple :

$$1, 2, 3, \underline{3}, \underline{4}, 7, 7, \underline{7}, 10, 12$$

avec $h = 4$ et $k = 8$, $ac_4 = 3$ et $ac_8 = 7$.

Si un processus P devient affamé, il faut que $ac_P \leftarrow V \geq k$. En effet, supposons que $V < k$. Dans ce cas, le k^{e} processus change de rang dans la liste triée (il augmente de un) mais son compteur ne change pas. Comme le processus était juste à la limite, il est en dessous maintenant ($ac_{k+1} = k - 1$) et la situation est non-sûre. Si par contre, $V \geq k$, il n'y a aucun problème pour les processus suivant le k^{e} puisqu'ils n'étaient pas à la limite. Dans le pire des cas, de nouveaux processus vont avoir un compteur égal à leur nouveau rang moins un (il faut alors recalculer le dernier) mais la situation reste sûre.

Quand il faut trouver un processus pouvant manger, on peut choisir n'importe lequel avant le rang h (ou celui de rang h). Supposons que nous en prenions un autre, après le rang h. Les processus avant vont voir leur compteur diminuer de un alors que leur rang ne change pas. Nous aurons donc $ac_h = h - 2$, ce qui n'est pas acceptable. En fait, le nombre de processus qui sont susceptibles d'avoir des ennuis est le nombre de processus avant celui choisi qui sont à la limite. Pour les processus après, il n'y a pas de problème car leur compteur diminue mais leur rang également. Leur situation ne change donc pas. Après avoir choisi un processus, il faut recalculer le premier et le dernier (en fonction du processus choisi).

Remarquons que s'il n'y a pas de processus à la limite, on peut prendre $k = 0$ et $h = n$. Quand un processus arrive, on peut lui donner n'importe quelle valeur pour son compteur, et quand il faut choisir le processus qui va manger, on peut prendre n'importe lequel (parmi les affamés dont les ressources sont disponibles).

Remarquons que, avec ces stratégies dynamiques, le système peut l'être aussi : on peut ajouter de nouveaux processus en cours de route, ce que ne permettait pas Dijkstra.

Une sous-classe des stratégies dynamiques

C'est donc fort simple à utiliser mais nous pouvons faire encore plus simple ! Nous allons considérer une sous-classe de stratégies dynamiques. Nous allons imposer que tous les compteurs soient différents. Dans ce cas, le pire cas c'est que les compteurs soient $0, 1, 2, 3, \ldots$ Cette situation est évidemment sûre. Nous n'avons donc même plus de test à faire puisque nous savons que la situation est sûre.

Pour déterminer les processus qui peuvent manger, il suffit de regarder le premier dans la liste triée. Si $ac_1 = 0$, c'est ce processus qui peut être le prochain à entrer dans sa section critique, et uniquement lui (pour respecter le critère de sûreté). Sinon, ça peut être n'importe quel processus. Il faut remarquer que, quand un processus se

met à manger, les compteurs des affamés restent différents. La situation reste donc sûre. Il s'agit d'une propriété stable (si les compteurs sont différents, ils le resteront).

Quand un nouveau processus arrive, on lui donne une valeur pour son compteur hors de l'ensemble déjà existant.

Pour avoir FIFO, il suffit de prendre le plus petit compteur possible quand un nouveau processus arrive.

Initialisation

On choisit $d \geq 0$ // d degrés de liberté
$M \leftarrow d$ // M = valeur d'initialisation du compteur
$nozero \leftarrow True$ // aucun compteur nul au début
$\forall i : ac_i \leftarrow \infty$ // au début, tous les compteurs sont à l'∞

i **devient affamé**

$ac_i \leftarrow M$
if $M = 0$ **then** // un processus doit absolument manger
  $nozero \leftarrow False$
fi
$M \leftarrow M + 1$

critère d'admissibilité pour i **(pour pouvoir manger)**

$(ac_i = 0 \vee (nozero \wedge ac_i \neq \infty)) \wedge ressources\ disponibles$

Si i **est admis**

$nozero \leftarrow True$
$ac_i \leftarrow \infty$ // le processus va manger
for $j \in [1 \ldots n]$ **do**
  $ac_j \leftarrow ac_j - 1$
  if $ac_j = 0$ **then** // on décrémente les compteurs
    $nozero \leftarrow False$ // et on vérifie qu'aucun n'arrive à zéro
  fi
od
$M \leftarrow M - 1$

Remarquons que nous n'avons même pas besoin de liste triée. Nous pouvons facilement voir que cela marche. Enonçons pour cela quelques invariants :

1. $M = d$ + *nombre d'affamés*
2. i affamé $\Rightarrow ac_i < M$
3. les compteurs des affamés sont différents (soit le nouveau processus i. Nous avons $ac_j < M$ *pour* j affamé or $ac_i \leftarrow M$)

4. Soit i, j affamés : $ac_i < ac_j$ si i est plus affamé que j (si i est arrivé avant j). En effet, j reçoit un compteur plus grand et la différence reste la même jusqu'à ce qu'un des deux processus mange.

Nous pouvons formuler quelques remarques :
- Il s'agit d'une forme de FIFO avec d degrés de liberté (on peut se faire dépasser au plus d fois). Pour ne jamais se faire dépasser, il faut avoir $d = 0$. Dans ce cas, nous avons un FIFO pur.
- Nous pouvons voir que si $d > 0$, le test **if** $M = 0$ n'arrive jamais.
- Si nous appliquons cela au problème des philosophes, nous pouvons voir qu'il y a quelque chose d'anormal. En effet, nous ne tenons pas du tout compte du voisinage. Nous décomptons tous les ac_i mais on peut plutôt s'attendre à décompter seulement les ac_i des voisins. De plus, avec Dijkstra (et les stratégies dynamiques dérivées), si on rajoute quelqu'un, tout le monde change car tout le monde se connaît.

1.4 Courtois I

Nous allons donc maintenant essayer de trouver des solutions locales, c'est-à-dire des solutions où les protocoles d'entrée et de sortie ne dépendent que des voisins du processus.

Ces solutions ont été imaginées par une équipe de chercheurs belges, dirigée par Courtois. Elles concernent le problème des philosophes et utilisent donc le caractére local de ce problème de famine.

Nous utilisons toujours un compteur ac (*allowance counter*) qui va maintenant nous servir à contrôler le nombre de fois où les voisins d'un philosophe passent devant lui. En fait, c'est même plus fort que cela. On peut ne contrôler qu'un seul voisin. Mais pour cela, il faut introduire de l'équité (*fairness*) ou de l'expédience. En l'occurrence, c'est l'expédience que nous allons utiliser (elle remplace en quelque sorte l'obligation spécifique).

Lorsque notre compteur tombe à zéro, nous bloquons le voisin que nous contrôlons et l'empêchons ainsi de manger. Nous dirons alors que nous avons un contrôle effectif sur ce voisin. Ensuite, dès que l'autre fourchette est libre, nous pouvons (et devons, par expédience) commencer à manger.

On peut avoir des stratégies de deux natures :
- statiques : le voisin que l'on contrôle est fixé a priori.
- dynamiques : quand on devient affamé, on choisit le voisin que l'on va contrôler.

Semblablement, la valeur initiale des compteurs peut être choisie statiquement a priori ou dynamiquement lors de l'arrivée d'un affamé.

Situations sûres et non sûres

Quand on se met à manger, on va décrémenter zéro, un ou deux compteurs (ceux des voisins qui nous contrôlent).

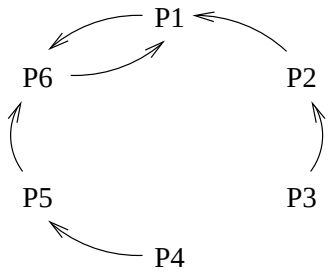

FIGURE 1.2 – Graphe de contrôle.

Nous pouvons voir, sur la figure 1.2 que P_3 et P_4 ne sont pas contrôlés, que P_2 et P_5 sont tous deux contrôlés par un seul voisin alors que P_1 et P_6 sont contrôlés par deux voisins.

De façon générale, on peut définir plusieurs types de graphes pour ces systèmes : le graphe de contrôle potentiel (dans le cas des stratégies statiques) reprend tous les arcs de contrôle, à partir de tous les nœuds ; le graphe de contrôle reprend tous les arcs de contrôle provenant de philosophes affamés (avec des compteurs finis) ; le graphe de contrôle effectif reprend tous les arcs de contrôle provenant de philosophes avec un compteur nul.

Ca va marcher, à condition que l'introduction des compteurs n'introduise pas de deadlock. En effet, sur la figure 1.3, nous voyons un exemple de deadlock causé par un cycle.

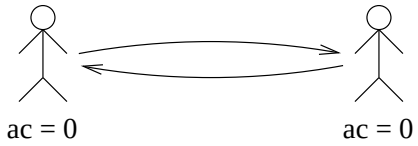

FIGURE 1.3 – Exemple de deadlock.

Il y a donc à nouveau des situations sûres et non sûres (attention : l'expédience ne pourra s'appliquer que si l'état résultant est sûr). La situation sera sûre s'il existe un ordre de débloquage des philosophes, c'est-à-dire une séquence convenable des ac_i : pour que cela marche, on ne peut pas avoir une situation où un processus i ayant un compteur ac_i nul, contrôle un processus avant lui dans la séquence.

Nous allons regarder le graphe de contrôle effectif : pour pouvoir construire une séquence correcte, il faut et il suffit que ce graphe soit acyclique. Il faut qu'il soit acyclique car sinon, nous aurons de toute façon un processus avec un compteur nul contrôlant un autre avant lui dans la séquence. Mais il suffit qu'il soit acyclique car, dans ce cas, nous pouvons séquentialiser le graphe. Nous avons un ordre partiel et on peut créer un ordre total (tri topologique). Nous aurons alors une séquence avec des flèches vers la droite. Remarquons que, lorsqu'un processus commence à manger, on décrémente les compteurs de ceux qui le contrôle ; cela peut faire apparaître de nouvelles flèches de contrôle effectif, mais uniquement vers le nouveau mangeur (puisque c'est leur unique contrôlé), et donc cela ne peut faire apparaître un cycle puisque son compteur est passé à l'infini.

Deux types de cycles

Mais nous pouvons faire encore plus simple ! Nous allons utiliser le fait qu'il ne peut y avoir que deux types de cycles qui apparaissent : des cycles aller-retour et des cycles d'Euler (qui font le tour de la table).

Que faire lorsqu'un philosophe devient affamé ? Il faut lui donner une valeur initiale pour son compteur. Si on donne une valeur supérieure à zéro ($ac_i > 0$), il n'y a pas de problème car ça ne change pas le graphe de contrôle effectif. Mais si les philosophes sont vraiment très impatients et qu'on veut leur donner un compteur nul, il peut y avoir des problèmes (car des cycles peuvent apparaître dans le graphe de contrôle effectif). Cela va alors dépendre de la stratégie utilisée.

- Stratégie statique. La personne que l'on veut contrôler est fixée. Mais si cette dernière nous contrôle avec un compteur nul, un cycle aller-retour va apparaître. Ce n'est donc pas acceptable et il faut choisir un compteur strictement positif.
- Stratégie dynamique. Nous pouvons déterminer le voisin que nous allons contrôler. Le seul cas à problème est si les deux voisins contrôlent le nouveau processus et que leurs compteurs sont nuls. Dans tous les autres cas, nous pouvons toujours décider de contrôler quelqu'un avec un compteur nul sans introduire de cycle dans le graphe de contrôle effectif.

Mais nous n'avons fait que regarder les problèmes causés par les cycles aller-retour ! Reste le problème du cycle d'Euler. En fait, le problème n'existe pas car ce type de cycle ne peut pas apparaître.

Théorème 1 *Un cycle d'Euler de contrôle effectif ne peut se former.*

Démonstration Démontrons cela par l'absurde.

Regardons le cycle d'Euler et surtout, la situation juste avant la fermeture du cycle (figure 1.4).

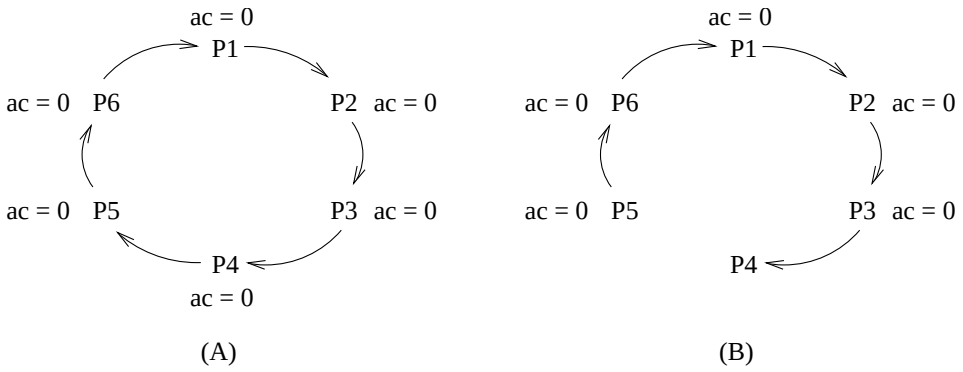

FIGURE 1.4 – Cycle d'Euler (A) et la situation juste avant (B).

P_4 devient donc affamé et il voudrait contrôler (effectivement) P_5. Mais cette situation est impossible. En effet, P_5 est affamé, n'a pas de contrôleur effectif et les fourchettes sont libres (car P_5 contrôle P_6 qui est affamé et P_4 pense). Or le système est expédient. Cela signifie donc que P_5 devrait être en train de manger.

Nous ne pouvons donc pas faire apparaître de cycle d'Euler dans le graphe de contrôle effectif. □

L'hypothèse d'expédience nous assure donc deux choses :

1. il ne faut contrôler qu'un seul voisin.
2. nous ne pourrons jamais fermer un cycle d'Euler.

De plus, nous pouvons constater que l'expédience est un phénomène local qui empêche un cycle d'Euler qui est un cycle global.

Nous pouvons alors utiliser des stratégies statiques et fixer le contrôle potentiel selon le cycle d'Euler. De la sorte, nous sommes sûrs de ne pas avoir de cycles aller-retour. Nous pouvons même toujours choisir un compteur nul quand on est affamé, et donc nous n'en n'avons plus besoin. Que devient alors le critère d'admissibilité ? Tout d'abord, il faut bien sûr avoir faim. Si le contrôle potentiel est fixé vers la gauche, il faut que le voisin de gauche ne mange pas et que le voisin de droite ne soit pas affamé (il doit penser). Si le contrôle potentiel est fixé vers la droite, c'est évidemment l'inverse.

Il s'agit donc finalement d'un critère très simple et local. Il y a cependant plusieurs problèmes qui surviennent.

1.4.1 problèmes liés à Courtois I

– Quand un philosophe arrive (devient affamé), il n'y a pas de problème. Avant de choisir son compteur, il va regarder s'il ne doit pas manger (à cause de

l'expédience). Si les fourchettes sont libres et s'il n'est pas contrôlé effectivement, il va commencer à manger. Mais le problème vient quand il a fini de manger. Il doit regarder si ses voisins ne doivent pas passer à table. Il doit donc vérifier que les fourchettes sont libres, mais aussi que les voisins ne soient pas contrôlés effectivement par leurs sur-voisins. Et ça, ce n'est plus local ! En effet, le processus va donc devoir regarder le voisin de gauche (droite) du voisin de gauche (droite). L'implantation de l'expédience n'est donc pas locale.
Pour résoudre ce problème, il y a deux possibilités : soit on relâche les contraintes et on accepte que les stratégies soient locales au deuxième degré, soit on réveille systématiquement les voisins et c'est à eux de voir s'ils peuvent manger. S'ils ne peuvent pas, ils se rendorment aussitôt. Dans ce dernier cas, cependant, nous n'aurons plus d'attentes passives mais des attentes semi-actives.
Remarquons également que, à cause de l'expédience, nous ne pouvons pas quitter la table avant que les voisins aient bien examiné la situation et pris leur décision. Ce n'est pas drôle car nous devons faire des attentes pour être synchrone.
Une autre manière de faire, c'est que tous les philosophes stockent dans leurs variables les états de leurs voisins. Cela semble local évidemment, mais cela contourne la véritable localité !

– De plus, il faut protéger les variables. Pour ce faire, il faut utiliser des sémaphores. Mais les sémaphores, ce sont des ressources ... comme les fourchettes ! Nous risquons donc d'avoir des problèmes de famine liés aux sémaphores. Des solutions sans compteurs existent mais sont très rigides. On pourrait dès lors utiliser des compteurs pour la gestion des fourchettes (cela indique donc un degré d'urgence) et utiliser une solution sans compteurs pour les sémaphores (car on ne garde pas les sémaphores très longtemps).
Par exemple, pour gérer les sémaphores, nous pouvons utiliser la technique hiérarchique de Havender. Nous allons donc ordonner les sémaphores et les prendre en suivant l'ordre. En effet, dans le cas de ressources uniques, Havender élimine les problèmes de deadlock et de famine (si on a une gestion équitable de la file d'attente (comme FIFO par exemple)).
Il faut cependant remarquer autre chose, maintenant. Un philosophe doit prendre son sémaphore et celui de ses deux voisins, mais tout le monde ne pourra pas le faire suivant le même ordre ! Cela brise donc la symétrie. Il faut donc une autorité centrale qui va choisir l'ordre, mais alors ce n'est plus local !

Remarques

Nous pouvons constater que nous aurions pu utiliser cela depuis le début pour les fourchettes (même si c'est assez rigide) car alors, on n'a plus besoin de compteurs (technique de Havender, mais c'est Courtois qui a fait remarquer cela). Chaque processus va indiquer, en entrant dans le système, s'il est gaucher ou droitier (il est gaucher s'il prend d'abord sa fourchette gauche, puis la droite, et s'il est droitier, il fait juste l'inverse). Pour chaque processus, il y a donc un ordre local total, et il faut que la réunion de ces ordres soit un ordre global (partiel éventuellement) ; il y a donc deux cas à éviter : que tout le monde soit, soit gaucher, soit droitier. Autrement dit, il faut au moins un gaucher et au moins un droitier. Dans ce cas, on ne pourra pas fermer de cycle d'Euler. On peut par exemple numéroter les philosophes et dire que les philosophes pairs sont gauchers et que les philosophes impairs sont droitiers.

Nous pouvons remarquer que prendre la fourchette gauche revient à contrôler de manière effective le voisin de gauche. S'il y a au moins un gaucher et un droitier, nous n'aurons donc pas de cycle de contrôle effectif. Et nous n'avons plus vraiment besoin d'expédience pour cette raison (il n'y a plus de cycle d'Euler) car on prend les fourchettes une à une. Notons tout de même que nous avons toujours besoin d'équité ou d'expédience pour ne contrôler qu'un seul voisin.

Remarquons qu'en prenant en alternance un gaucher et un droitier, nous pouvons confiner le problème des "morts-mangeurs". Supposons que nous ayons un nombre pair de philosophes. Regardons la figure 1.5.

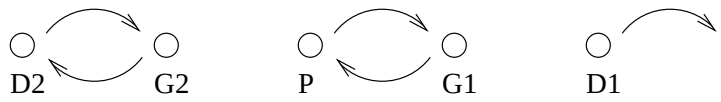

FIGURE 1.5 – Nombre pair de philosophes et alternance des gauchers et des droitiers

Le philosophe P est droitier. Regardons ce qui se passe si P meurt en gardant ses fourchettes. Nous voyons que G_1 ne peut pas prendre sa fourchette gauche, et ne cherche donc pas à prendre sa fourchette droite. D_1 n'est donc pas dérangé et peut continuer son exécution. Pour G_2, par contre, il y a un problème. G_2 peut prendre sa fourchette gauche, puis être bloqué avec sa fourchette droite. Cela signifie alors que D_2 est également bloqué. Mais c'est tout ! Nous avons donc confiné le problème à un philosophe à droite et à deux philosophes à gauche. Si on ne respecte pas l'alternance, il y aura aussi confinement mais dans des domaines plus grands.

Autre solution de Courtois

Voici à présent une autre solution de Courtois restaurant la symétrie. Tous les philosophes sont gauchers (ou tous sont droitiers), et nous avons un nouveau sémaphore ($Table$) initialisé à $n-1$ (n étant le nombre de philosophes dans le système).

do
think
$P(Table)$
$P(fg); P(fd)$
eat
$V(fd); V(fg)$
$V(Table)$
od

Ce qui est intéressant, c'est que nous n'avons plus besoin d'expédience car nous empêchons la fermeture du cycle d'Euler. En effet, à cause du sémaphore $Table$, il ne pourra jamais y avoir tous les philosophes dans leur section critique. Nous n'aurons donc pas plus de $n-1$ philosophes affamés. Il pourra toujours y en avoir un qui mange.

Le seul problème, c'est que ce n'est plus strictement local. $Table$ est en effet global. Ceci dit, le comportement d'un philosophe ne dépend pas vraiment de celui des autres philosophes. Si un nouveau philosophe arrive, on ne va changer que localement, mais il faudra augmenter artificiellement le sémaphore $Table$ (il faut faire $V(Table)$).

A part cela, avec cette méthode, nous n'avons pas de confinement en cas de problème.

Conclusion

Nous voyons que nous avons toujours des petits problèmes qui surviennent. Généralement, nous ne parviendrons pas à garder la symétrie, le caractére local et des attentes passives. Nous devrons toujours un peu relâcher ces contraintes. Nous avons vu également que l'implantation de l'expédience n'est pas anodine.

1.4.2 Généralisation

Nous allons maintenant essayer de généraliser la méthode de Courtois et voir ce que cela donne.

Les fourchettes sont donc des ressources non-préemptibles non-interchangeables et les philosophes sont des processus. Deux processus sont voisins s'ils ont besoin de ressources communes (s'ils sont mutuellement exclusifs) pour leur phase critique (eat). Si un processus a n voisins, il doit en contrôler $n-1$ (il peut les contrôler de manière statique ou dynamique).

Mais le critère de Courtois pose des problèmes. Celui-ci était que l'on pouvait se construire une séquence de réduction valide si et seulement si il n'y avait pas de cycle de contrôle effectif. Nous avions donc un critère nécessaire et suffisant. De plus nous n'avions que deux types de cycles (cycles d'Euler et cycles aller-retour), et ici, ce n'est plus vrai (nous avons un graphe de voisinage quelconque [2]).

Nous ne pourrons plus contrôler les cycles localement, même avec l'expédience mais en plus de cela, même si nous n'avons pas de cycle de contrôle effectif, il se peut que le système soit non-sûr : la condition n'est plus suffisante. Pour s'en convaincre, il suffit de regarder le graphe de la figure 1.6.

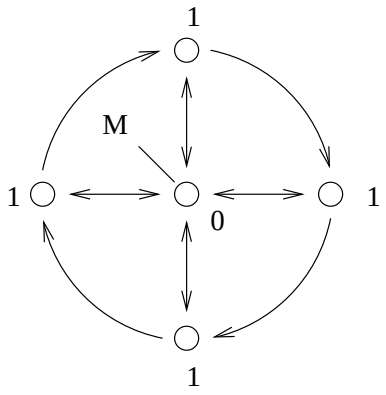

FIGURE 1.6 – Exemple de système sans cycle de contrôle effectif mais menant à un deadlock.

Nous avons N processus sur la roue, donc le processus du centre a $N+1$ voisins. Nous pouvons voir que le système est bien expédient (le processus M mange, le processus du centre ne peut pas manger à cause de M, et les processus de la roue ne peuvent pas manger car ils sont contrôlés effectivement par le processus du centre). Nous pouvons également voir que nous n'avons pas de cycle de contrôle effectif, et chaque affamé contrôle $n_{voisin}-1$ voisins.

2. Remarquons que pour un graphe donné, il y a toujours un système correspondant. Il suffit de considérer les arêtes du graphe comme étant les ressources et les nœuds comme étant les processus. On peut toujours supposer que le graphe est connexe, sinon on décompose le système en plusieurs sous-systèmes.

Regardons maintenant le déroulement futur du système. M arrête de manger. Le processus du centre doit commencer à manger et tous les compteurs finis sont décrémentés. Mais alors, tous les processus de la roue ont un compteur nul. Nous avons donc un splendide cycle de contrôle effectif après un coup. Cela signifie donc que la situation de départ était non sûre puisqu'elle a mené à un deadlock.

Le critère de Courtois n'est donc plus une condition suffisante (mais elle est toujours nécessaire !) : le critère ne se généralise pas directement.

En fait, il y a bien un critère de sûreté plus général : il faut trouver une séquence des affamés où ça se passe bien. Il faut donc que $ac_i \geq$ nombre de processus qu'il contrôle et qui sont avant lui dans la séquence. Mais, outre le fait que ça n'a plus une représentation graphique simple, trouver un tel ordre n'est clairement pas un problème local en général.

1.5 Lauer & Devillers

Voici à présent une solution générale, assez simple, qui permet également de supprimer l'hypothèse d'expédience.

Dans le cas des philosophes, si on n'avait pas d'expédience, il fallait contrôler les deux voisins. Ici, un processus devra donc contrôler tous ses voisins. Et pour chaque voisin d'un processus, nous aurons un compteur ($\forall P_i : \forall$ voisin $j : \exists ac_i^j$). Cela permet d'être plus souple que Courtois car on peut décider d'accepter de faire passer plus souvent un processus devant soi qu'un autre. Dans le graphe, nous pouvons donc associer les compteurs aux flèches de voisinage plutôt qu'aux processus.

Nous remplaçons l'expédience par une hypothèse d'instabilité : si un processus affamé n'a que des voisins non-mangeurs, cela ne peut pas continuer indéfiniment ; il faut que l'un des affamés commence à manger.

Quand un processus devient affamé, il faut initialiser les compteurs. Quand un processus commence à manger, il faut décrémenter les compteurs le concernant (flèches entrantes) ; nous ne pouvons toujours pas avoir de compteurs négatifs. Comme d'habitude, cela va bien se passer si cela n'introduit pas de deadlock, c'est-à-dire si la situation est sûre, càd s'il existe une séquence de déblocage des affamés qui ne fait pas apparaître de compteur négatif.

Nous avons un critère très simple maintenant.

Théorème 2 *Le système est sûr $\Leftrightarrow$ il n'y a pas de cycle de contrôle effectif.*

Démonstration

⇒

Regardons la contraposée : *le système est non sûr* ⇐ *il existe un cycle de contrôle effectif.*

S'il y a un cycle de contrôle effectif, pour toute séquence de déblocage, le dernier processus du cycle va devenir négatif. Ce n'est donc pas acceptable.

⇐

S'il n'y a pas de cycle, il existe un ordre partiel lié au graphe de contrôle effectif. Nous pouvons trouver au moins un processus P qui n'a pas de prédécesseur (il n'a donc pas de contrôleur effectif) et le débloquer. Cela peut faire apparaître de nouveaux compteurs nuls chez ses voisins (uniquement) et de nouvelles flèches de contrôle effectif, mais uniquement vers P, dont tous les compteurs sortant passent à l'infini. Cela ne peut donc faire apparaître de cycle de contrôle effectif (cependant, il faut reconstruire l'ordre car il existe des nouvelles flèches et l'ancien ordre n'est peut-étre plus valable), et nous pouvons recommencer avec un deuxième, . . ., jusqu'à épuiser tous les affamés. □

Quand on devient affamé, il faut choisir des compteurs pour les arcs sortants mais il faut veiller à ne pas fermer de cycles. Le problème c'est que les cycles ne sont pas locaux. Nous avons deux possibilités :

- on regarde le graphe globalement. Dans ce cas, on peut choisir les valeurs que l'on veut, du moment que l'on ne fait pas apparaître de cycle de contrôle effectif.
- on regarde le graphe localement. Dans ce cas, il suffit de mettre les $ac_i^j > 0$. En réalité, on peut mettre des $ac_i^j = 0$ si tous les arcs arrivant vers soi sont supérieurs à zéro (on ne fait alors pas apparaître de cycle).

De ce point de vue-là, c'est moins souple que Courtois car on ne peut pas toujours être très pressé (Courtois permettait d'avoir toujours des compteurs nuls). Cependant, nous n'avons plus besoin d'expédience et il ne faut plus absolument réveiller les autres processus.

Ceci dit, il faut quand même protéger les compteurs. Nous allons donc utiliser des sémaphores. Il faut un ordre partiel qui est localement total, et nous pouvons alors utiliser Havender (solution sans compteurs) pour prendre ces sémaphores sans deadlock et sans famine.

Le nombre de compteurs est assez important ($n * nombre\ moyen\ de\ voisins$, et donc $n(n-1)$ si le graphe est complet). Mais il y a pas mal de souplesse car on peut être impatient vis-à-vis de certains voisins et pas pour d'autres.

Avec cette méthode, nous n'avons pas de confinement. C'est évidemment lié à la plus longue chaîne de contrôle effectif, et celle-ci peut être presque un cycle d'Euler.

1.6 Courtois II

Nous nous plaçons ici aussi dans le cas général d'un graphe de voisinage quelconque. Cette autre méthode imaginée par Courtois est tout à fait différente des précédentes.

Le principe de base est toujours le même : il faut éviter les cycles de contrôle effectif. Mais voici maintenant l'idée (qui peut sembler bizarre) générale de la méthode : si on ne voit pas qu'on est contrôlé, il n'y a pas de problème ! Ceci dit, il faudra quand même de temps en temps regarder !

Pour ce faire, nous allons faire des photographies de la situation. Nous n'avons de nouveau qu'un seul compteur par processus, mais celui-ci peut maintenant devenir négatif. Nous n'aurons besoin que d'une forme affaiblie d'expédience. Nous avons toujours la même boucle générale :

```
do
   think
   protocole d'entrée (PE)
   eat
   protocole de sortie (PS)
od
```

Commençons par voir le protocole d'entrée :

```
PE :                          // Prendre les sémaphores nécessaires, dans un ordre approprié
   P ← C                      // Photo des contrôleurs
   if P¬vide then
      wait                    // Sortie de PE (relâche les sémaphores)
   fi
Point d'entrée :
   P ← E                      // Photo des mangeurs (Eater)
   if P¬vide then
      wait                    // Sortie de PE (relâche les sémaphores)
      goto Point d'entrée
   fi
   ac ← ∞                     // car on commence à manger
   prendre les ressources     // et relâcher les sémaphores (on sort de PE)
```

Au début du protocole d'entrée, on réalise une photo des contrôleurs. Dans cette photo, on ne garde que nos contrôleurs effectifs (processus dont le compteur est

inférieur ou égal à zéro). Nous ne faisons pas apparaître de cycle car le compteur du processus est encore à l'infini.

Quand des processus sont sur cette photo, on ne peut pas manger car on est contrôlé effectivement (et on le sait !). Il faut donc attendre. Après cela, nous ferons une photo des mangeurs, et à nouveau, on ne peut pas manger si des voisins mangent. Mais cette fois, c'est parce que des voisins sont occupés à manger, et pas parce qu'on a des contrôleurs effectifs ; donc on peut considérer qu'on ne fait jamais apparaître de cycle de contrôle effectif.

Remarquons que lors des *wait*, on sort du protocole d'entrée. En effet, nous devons évidemment relâcher les sémaphores pris.

Regardons maintenant le protocole de sortie :

```
PS : // on prend les sémaphores, ...
    pour tout voisin v faire                  // Mettre à jour la photo
        si on est sur la photo alors
            on s'efface de la photo
            si la photo devient vide alors
                si ac_v = ∞ alors
                    ac_v ← I_v
                sinon
                    ac_v ← ac_v − 1
                fsi
                réveiller le voisin           // v revient au Point d'entrée en reprenant les
            fsi                                  sémaphores nécessaires dans l'ordre approprié
        fsi
    fpourtout                                 // on relâche les sémaphores
```

Le choix de I_v est quelconque (≥ 0) ; il peut dépendre des voisins, du temps, des évolutions passées, etc. Quand un voisin est réveillé, il revient dans le protocole d'entrée au *Point d'entrée*. Nous ne comptons donc plus le nombre de fois où les autres mangent avant nous-même, mais le nombre de fois où l'on est réveillé.

Il n'y a pas de blocage possible puisqu'il n'y a pas de cycle de contrôle effectif qui peut apparaître, mais comme les compteurs peuvent devenir négatifs, il faut effectuer une analyse plus fine du temps d'attente pour s'assurer qu'il n'y a pas famine. Nous allons pour cela faire deux hypothèses : les philosophes ont un temps maximum dans leur phase *eat* (soit e_i le temps dans la phase *eat* pour le processus i), et les temps passés dans les protocoles d'entrée et de sortie sont négligeables ; nous allons aussi supposer que I_v est fixé.

Nous allons maintenant pouvoir faire une analyse assez fine de cette méthode.

Calculons le temps maximum d'attente pour un processus i. Pour cela, nous

allons regarder les photos des processus puisque c'est à ces moments-là que les processus peuvent se mettre en attente.

- Pas de contrôleurs et pas de mangeurs (les deux photos sont vides). Le temps d'attente est nul ($T = 0$).
- Pas de contrôleurs mais il y a des voisins mangeurs (première photo vide mais pas la deuxième). Le temps maximum avant d'être réveillé est $\max_{v \in V_i}(e_v)$, où V_i est l'ensemble des voisins de i, si chaque processus a son propre processeur (sinon, il faut remplacer max par $\sum$).
  Après un temps $(I_i+1)\max_{v \in V_i}(e_v)$, le processus i va devenir contrôleur effectif. I_i+1 correspond à l'initialisation (alors que le processus a encore son compteur à l'infini) et au nombre de fois que des voisins peuvent passer devant lui. Après ce temps, de nouveaux processus ne peuvent plus devenir affamé et passer devant lui car il va apparaître sur leur photo des contrôleurs. La seule chose qui peut encore empêcher le processus de manger est qu'il y a encore des voisins mangeurs, qui étaient devenus affamés avant que ac_i ne devienne nul ou négatif. Au pire, chacun peut manger encore une fois, chacun à son tour s'ils ont des ressources communes. Nous allons donc encore rajouter ce temps-là. Nous obtenons finalement :
  $$T \leq (I_i + 1) \max_{v \in V_i}(e_v) + \sum_{v \in V_i} e_v$$
  En réalité, le temps d'attente est moins grand que cela car :
  - nous n'avons que des bornes supérieures de e_v.
  - tous les voisins n'étaient peut-être pas en train de manger lors des photos.
  - de fait, le processus qui nous a rendu nul est mis hors-jeu dans le dernier terme.
  - dans le dernier terme, certains voisins peuvent manger simultanément.
- Il y a des contrôleurs, puis des voisins mangeurs (les deux photos ne sont pas vides). Le processus doit d'abord attendre d'être réveillé. Il faut donc calculer le temps d'attente maximum pour la première fois. Ce temps vaut :
  $$\max_{v \in V_i}(e_v + \sum_{w \in \{V_v \setminus \{i\}\}} e_w)$$
  C'est le temps nécessaire pour que le dernier de la photo puisse finir de manger, mais celui-ci peut être géné par des voisins mangeurs (uniquement, et au plus une fois puisqu'il est lui-même déjà contrôleur effectif).
  Le temps maximum d'attente est comme dans le cas précédent après le premier réveil. Le temps total (maximal) vaut donc :
  $$T = \max_{v \in V_i}(e_v + \sum_{w \in \{V_v \setminus \{i\}\}} e_w) + I_i \max_{v \in V_i}(e_v) + \sum_{v \in V_i} e_v$$

Ce qui est intéressant, c'est que nous avons des bornes dans tous les cas. Il suffit donc de prendre le maximum des deux formules pour avoir le temps d'attente dans le pire des cas. De plus, comme nous avons une borne, nous n'avons pas de famine !

En outre, cette formule ne dépend que des e_v et des e_w, c'est-à-dire des voisins et des voisins des voisins. Cela signifie que si un philosophe meurt en gardant ses fourchettes, il n'embête pas trop de monde : nous avons à nouveau confinement des problèmes.

Regardons un exemple :

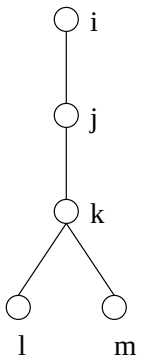

FIGURE 1.7 – Confinement du problème dans Courtois II.

Si i meurt dans sa phase eat, $ac_i = \infty$. Si j a faim, $ac_j < \infty$. Il ne peut pas manger car i est sur la photo des mangeurs de j et y restera. Mais j peut-il être contrôleur effectif ? Oui, si k a fini de regarder la photo des contrôleurs et que j devient contrôleur effectif à ce moment-là. Dans ce cas, k ne peut plus manger car il est contrôlé effectivement par j. Par contre, k ne géne, ni l, ni m, car j reste toujours sur sa photo des contrôleurs et ac_k ne change jamais (et il reste à $ac_k = \infty$). k ne gênera donc personne car il ne sera jamais sur aucune photo. En fait, personne ne saura jamais que k a faim.

Avec Courtois II, nous avons donc un confinement du problème aux voisins, et aux voisins de ceux-ci.

Ici, nous supposons que le temps éventuel d'attente est compris dans e_i. Remarquons que nous avons besoin d'une forme d'expédience car sinon, e_i ne voudrait rien dire ! Autrement dit, si nous n'avons pas un peu d'expédience, nous n'avons pas de borne supérieure !

Il faut cependant remarquer que, comme dans les autres méthodes, il faut protéger les variables (les photos). Nous aurons donc de nouveau besoin de sémaphores, et donc de Havender.

Bibliographie

[1] JJ. Cocu et R. E. Devillers. *On a class of allocation strategies inducing bounded delays only*, The Computer Journal (25), pp 52–55 (1982).

[2] P. J. Courtois et J. Georges. *On starvation prevention*, RAIRO Informatique 11 (2), pp 127–141 (1977).

[3] R. E. Devillers et P. E. Lauer. *A general mechanism for avoiding starvation with distributed control*. Information Processing Letters 7 (3), pp 156–158 (1978).

[4] E.W. Dijkstra. *Cooperating sequential processes*, in Programming Languages, Academic Press (Genuys, ed.), pp 103–110 (1968).

[5] E.W. Dijkstra. *Hierarchical ordering of sequential processes*, Acta Informatica 1(2) pp 115–138 (1971).

[6] E.W. Dijkstra. *A class of allocation strategies inducing bounded delays only*, Spring Joint Computer Conference, pp 933–936 (1972).