Travaux Dirigés et Pratiques n°2

Algorithmes parallèles et répartis

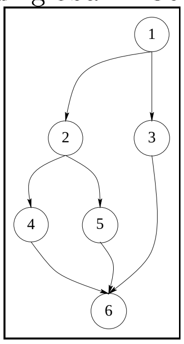
—Élèves-Ingénieurs de 5-ème année—

Concurrence et Systèmes Distribués

Cette feuille porte tout d'abord sur l'expression de la concurrence avec une introduction au problème d'ordonnancement de tâches. Nous mettons ensuite en évidence les problèmes d'exclusion d'accès à des données partagées par plusieurs processus concurrents. Nous voyons en particulier deux types d'algorithme qui correspondent respectivement à des solutions dites centralisées et distribuées. Ceci nous amène à regarder de plus près les problèmes se posant en système (complètement) distribué.

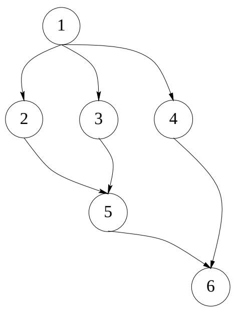
Expression de la concurrence

Un des problèmes classiques rencontré dès lors que l'on cherche à paralléliser un programme concerne l'ordonnancement des processus qui vont intervenir dans le calcul global. Ce problème est connu sous le nom d'ordonnancement de tâches. Le problème de l'ordonnancement de tâches peut se formuler ainsi : ordonnancer en temps minimum un ensemble de tâches soumises à des contraintes de précédence fixées, sur un ensemble de processeurs identiques.



Lorsque l'on cherche à résoudre des problèmes d'ordonnancement, on représente les contraintes de précédence imposées pour les différentes tâches par un graphe de précédence dont les sommets sont les tâches mises en concurrence. Les arêtes orientées du graphe spécifient l'ordre d'exécution des différentes tâches, le graphe étant par nature acyclique.

▸ **Exercice 1.** **(expression de la concurrence)** *On considère le graphe de précédence suivant, qui met en œuvre* 6 *tâches concurrentes :*



Partant du graphe de précédence, on exprime la concurrence à l'aide des primitives `fork` *et* `join` *ou du formalisme* `parbegin` *et* `parend`*.*

1. *Spécifier une exécution parallèle de ce graphe en utilisant les constructions* `parbegin` *et* `parend`*.*
2. *En spécifier une exécution parallèle en utilisant les constructions* `fork` *et* `join`*.*

▸ **Exercice 2.** **(du séquentiel au parallèle)** *On considère la portion de programme suivante :*

1.	`a=b-c+h`	2.	`b=d-e`
3.	`h=c-i`	4.	`a=c+e`
5.	`h=2*h`	6.	`d=d*3`
7.	`i=i-c`	8.	`b=e`
9.	`e=b-d`	10.	`i=h-c+a-e`

et on définit les ensembles $\mathcal{R}(i)$ et $\mathcal{W}(i)$ d'une instruction i comme étant :

① $\mathcal{R}(i)$ *est l'ensemble des variables dont les valeurs sont utilisées par i ;*

② $\mathcal{W}(i)$ *est l'ensemble des variables dont les valeurs sont modifiées par i.*

1. *Quelles sont les conditions (dites de Bernstein) sur les ensembles $\mathcal{R}$ et $\mathcal{W}$ pour que deux instructions i_1 et i_2 soient en concurrence.*
2. *Donner les ensembles $\mathcal{R}$ et $\mathcal{W}$ pour chacune des lignes du programme.*
3. *En déduire le graphe de précédence de cette portion de programme.*
4. *En spécifier une exécution parallèle en utilisant les constructions* `fork` *et* `join`*.*
5. *Pourquoi l'opération de* `join` *doit-elle être atomique ?*
6. *Spécifier une exécution parallèle de ce graphe en utilisant les constructions* `parbegin` *et* `parend`*.*

   On considère maintenant une modification de la portion de programme précédente. On remplace l'instruction `6` *par l'instruction :* `d=i-d`*.*
7. *Peut-on paralléliser cette portion de programme en utilisant uniquement les constructions* `parbegin` *et* `parend` *?*
8. *Spécifier une exécution parallèle en utilisant les constructions* `fork` *et* `join`*.*
9. *Utiliser un sémaphore et la construction* `parbegin` *et* `parend` *pour spécifier une exécution parallèle de ce programme.*

On restreint à deux processus P_0 et P_1 notre étude du problème de la section critique et on considère une exécution concurrente du type `parbegin P0; P1; parend`. On supposera par ailleurs que chaque processus a une vitesse d'exécution non nulle et que l'on utilise une machine ne disposant que des opérations atomiques classiques comme `load` et `store`.

▸ **Exercice 3.** (**propositions d'algorithmes**) *Dans la suite, on décrit l'algorithme effectué par le processus $P_{\mathtt{i}}$ et on dénote par* `j` *le numéro de l'autre processus* ($\mathtt{j} = 1 - \mathtt{i}$). *Critiquer chacun des algorithmes suivants sensés résoudre le problème de la section critique :*

1. *Les processus partagent une variable* `tour` *initialisée à* 0 *:*

   ```
   int tour=0;
   for (;;)
   { while (tour!=i);
     section_critique();
     tour=j;
     section_non_critique();
   }
   ```

2. *L'algorithme suivant utilise une variable* `drapeau` *de type tableau de deux booléens :*

   ```
   typedef enum {faux, vrai} Bool;
   Bool drapeau[2]={faux, faux};
   for (;;)
   { while (drapeau[j]==vrai);
     drapeau[i]=vrai;
     section_critique();
     drapeau[i]=faux;
     section_non_critique();
   }
   ```

3. *L'algorithme suivant est une légère modification de la proposition précédente :*

   ```
   for (;;)
   { drapeau[i]=vrai;
     while (drapeau[j]==vrai);
     section_critique();
     drapeau[i]=faux;
     section_non_critique();
   }
   ```

4. *L'algorithme suivant reprend les variables partagées de l'algorithme précédent :*

   ```
   for (;;)
   { drapeau[i]=vrai;
     while (drapeau[j]==vrai)
     { drapeau[i]=faux;
       while (drapeau[j]==vrai);
       drapeau[i]=vrai;
     }
   ```

```
    section_critique();
    drapeau[i]=faux;
    section_non_critique();
  }
```

▸ **Exercice 4.** (**algorithme de Dekker**) *On considère deux processus P_0 et P_1 qui exécutent l'algorithme suivant dû à Dekker :*

```
typedef enum {faux, vrai} Bool;
Bool drapeau[2] = {faux, faux};
int tour=0;
for (;;)
{ j=1-i;
  drapeau[i]=vrai;
  while (drapeau[j]==vrai)
  { if (tour==j)
    { drapeau[i]=faux;
      while (tour==j);
      drapeau[i]=vrai;
    }
  }
  section_critique();
  tour=j;
  drapeau[i]=faux;
  section_non_critique();
}
```

Prouver que l'algorithme de Dekker satisfait bien les trois contraintes du problème de la section critique pour deux processus, c'est-à-dire l'exclusion mutuelle en section critique, l'atteignabilité de la section critique et l'attente bornée.

Les algorithmes présentés ci-dessus concernent le problème de la section critique dans le cas de deux processus. Ces algorithmes étaient tous *centralisés*. Cela signifie que les processus impliqués utilisaient plusieurs variables matérialisant un certain état du système de processus et parmi ces variables, au moins une était accessible à la fois en lecture et écriture par l'ensemble des processus. Ce type de variable est appelé *variable partagée*.

▸ **Exercice 5.** (**algorithme de Eisenberg et McGuire, 1972**) *Dijkstra a généralisé en 1965 la solution de Dekker au cas de n processus. Cette solution assure l'exclusion mutuelle, l'absence d'interblocage mais n'exclut pas la famine. La première solution équitable a été donnée par Knuth en 1966. Elle a subit des modifications pour réduire la borne exponentielle de l'attente maximale qu'elle induisait. Ces modifications conduisent à la version de Eisenberg et McGuire où* `n` *processus partagent les variables suivantes :*

```
typedef enum {passif, demande, en_sc} Bool;
```

```
Bool drapeau[n]={passif, ..., passif};
int tour=0;
```

Le `i`*-ème processus exécute alors l'algorithme suivant et utilise une variable locale* `j` *de type entier :*

```
for (;;)
{ do
  { drapeau[i]=demande;
    j=tour;
    while (j!=i) if (drapeau[j]!=passif) j=tour; else j=(j+1)%n;
    drapeau[i]=en_sc;
    j=0;
    while ((j<n) && ((j==i) or (drapeau[j]!=en_sc))) j++;
  }
  while ((j>=n) && ((tour==i) or (drapeau[tour]==passif)));
  tour=i;
  section_critique();
  j=(tour+1)%n;
  while ((j!=tour) && (drapeau[j]==passif)) j=(j+1)%n;
  tour=j;
  drapeau[i]=passif;
}
```

1. *Quelle propriété (exclusion mutuelle, atteignabilité de la section critique, absence de famine, etc.) se déduit du fait que la variable* `tour` *n'est modifiée que par un processus entrant ou quittant sa section critique.*

2. *Prouver l'absence de famine et donner la borne maximale du nombre de tours d'attente pour entrer en section critique.*

3. *Prouver l'exclusion mutuelle de la section critique.*

Face aux algorithmes résolvant le problème de la section critique en centralisé, se trouvent les algorithmes en *distribué*, qui n'utilisent aucune variable partagée.

Lamport a été le premier en 1974 à donner un algorithme ne s'appuyant sur aucun dispositif centralisé. Cet "algorithme de la boulangerie" s'appuie sur le principe très simple (que l'on trouve dans certains magasins) selon lequel chaque client reçoit un ticket en entrant qui permet de servir les clients dans leur ordre d'arrivée. Mais puisque l'on ne veut pas dans cet algorithme se servir d'une information centralisée, chacun des processus choisit son propre numéro en fonction des numéros pris par les autres et si deux processus ont pris des numéros identiques, un protocole (arbitraire) résoud le conflit.

▸ **Exercice 6.** (**l'algorithme de Lamport, dit "de la boulangerie"**) *On considère* `n` *processus partageant en lecture les variables suivantes :*

```
typedef enum {faux, vrai} Bool;
Bool choix[n]={faux, ..., faux};
int num[n]={0, ..., 0};
```

On considère l'algorithme suivant exécuté par chaque processus $P_{\mathtt{i}}$ *pour* $\mathtt{i} = 0, \ldots, \mathtt{n}-1$ *:*

```
choix[i]=vrai;
num[i]=1+max(num[0], ..., num[n-1]);
choix[i]=faux;
for(j=0;j<n;j++)
{ if(j!=i)
  { while (choix[j]==vrai);
    while ((num[j]!=0) && ((num[j],j)<(num[i],i)));
  }
}
section_critique();
num[i]=0;
section_non_critique();
```

L'ordre sur les couples correspond à l'ordre $(a, b) < (c, d)$ *si* $a < c$ *ou si* $a = c$ *et* $b < d$.

1. *Montrer que si la variable* `choix` *n'est pas utilisée, l'algorithme est faux, c'est-à-dire n'interdit pas la situation où deux processus seraient en section critique en même temps.*
2. *Montrer que l'algorithme satisfait bien l'exclusion mutuelle, l'atteignabilité de la section critique et l'attente bornée.*

Systèmes complètement distribués

Le problème de la synchronisation en environnement distribué est équivalent à la résolution du problème de l'exclusion mutuelle. C'est ainsi que résoudre l'un revient à résoudre l'autre. Puisque nous nous intéressons au problème de l'exclusion mutuelle, nous restons dans ce cadre mais cette fois en environnement complètement distribué, c'est-à-dire tel qu'aucune variable ne soit partagée ni en lecture ni en écriture. Nous considérons donc n processus résidant sur n processeurs différents voulant chacun entrer en section critique.

Il existe deux types de solution : l'une qualifiée de centralisée consiste à une prise de décision concernant le prochain processus autorisé à entrer en section critique localisée au niveau d'un processus fixé ; l'autre distribuée et même complètement distribuée n'utilise que des échanges de messages pour la décision.

▸ **Exercice 7.** (**solution centralisée**) *On s'intéresse dans cet exercice à une solution centralisée au problème de la section critique :*

1. *Donner une solution centralisée au problème de l'exclusion mutuelle en environnement distribué.*

2. *Combien de messages sont nécessaires par entrée en section critique ?*

3. *Quels sont les problèmes posés par cette solution ?*

▸ **Exercice 8.** (**solution complètement distribuée**) *Si on cherche une solution complètement distribuée, le problème est plus difficile à résoudre. Lamport a été le premier a donner une solution en 1978, solution s'appuyant sur son mécanisme permettant d'ordonner totalement les évènements d'un système. La solution demandait $3 \times (n-1)$ messages par entrée en section critique. Par ailleurs, Ricart et Agrawala ont proposé en 1981 une solution ne demandant que $2 \times (n-1)$ messages par entrée en section critique. Cette solution s'appuie aussi sur le même mécanisme d'ordonnancement des évènements. Voici cet algorithme.*

Lorsqu'un processus p souhaite entrer en section critique, il génère un nouveau label de temps, TS, et envoie une demande (p, TS) à tous les autres processus (y compris lui-même). Lorsqu'un processus reçoit un message de demande, il va soit répondre immédiatement soit différer sa réponse (notamment par exemple parce qu'il est lui-même en section critique). Un processus ayant reçu une réponse de tous les autres processus pourra entrer en section critique. Une fois sorti de sa section critique, le processus enverra une réponse à tous les processus dont il a reçu une demande (c'est-à-dire ceux pour qui il avait différé la réponse).

La décision de différer ou non la réponse suit les principes suivants :

- *si le processus est en section critique, différer la réponse ;*
- *si le processus ne souhaite pas entrer en section critique, répondre immédiatement ;*
- *si le processus souhaite entrer en section critique, il consulte la liste de demandes qu'il maintient. Cette liste contient toutes les demandes y compris la sienne auxquelles il n'a pas encore répondu. Recevant un message de demande, (p, TS), il compare TS avec les TS_i de tous les messages de la file de demandes. Si TS est plus petit que tous les TS_i, une réponse est immédiatement envoyée à p. Sinon, la demande est stockée dans la liste.*

1. *Quelle connaissance globale sur le système de processus chaque processus doit-il avoir ?*

2. *Que faire lorsqu'un nouveau processus entre dans le système ?*

3. *Que se passe t-il lorsqu'un des processus meurt ? Comment contrer ce problème ?*

4. *Réaliser une implémentation sous* `PVM` *de cet algorithme.*

Nous venons de voir une solution centralisée demandant l'élection d'un coordinateur parmi différents processus. Ce mécanisme d'élection est classique en système distribué car ce type de système utilise souvent un coordinateur pour réaliser telle ou telle action nécessaire aux autres processus. Lorsque le coordinateur meurt, il faut choisir parmi le groupe de processus restants un nouveau coordinateur. C'est l'objet des algorithmes d'élection.

Les algorithmes d'élection utilisent une priorité associée à chaque processus p_i. Par soucis de simplicité, nous noterons i la priorité du processus p_i.

▸ **Exercice 9. (l'algorithme "brute")** *L'algorithme "brute" ("bully" algorithm) suppose qu'un processus p_i envoie une requète pour laquelle il n'obtient pas de réponse de la part du coordinateur après un temps T. Il considère donc que le processus coordinateur est mort, et il tente alors de s'élire nouveau coordinateur. Pour cela :*

- *p_i envoie un message d'élection à tous les processus ayant une priorité supérieure à i. Il se met alors en attente d'une réponse de l'un des processus contactés ;*
- *si aucune réponse ne lui parvient après un temps T, il considère que tous les processus de priorité supérieure sont morts et de fait, p_i s'élit comme nouveau coordinateur. Il en informe tous les processus de priorité plus faible ;*
- *par contre, si une réponse lui parvient, p_i commence une attente pendant un temps T' d'un message l'informant qu'un processus de priorité plus grande a été élu. Ainsi, un autre processus est en train de s'élire lui-même et d'informer les autres processus de son élection en un temps T'. Si aucun message n'est reçu après le temps T', le processus ayant un numéro de priorité plus grand est considéré comme mort, et p_i recommence l'algorithme.*

1. *Finir la description de l'algorithme en précisant les deux types de situations pouvant se produire à la réception par p_i d'un message de p_j.*
2. *Quel est le processus élu à la fin de l'algorithme ?*