Nom : Prénom :

IUP MIAGe 3^{eme} année — Année universitaire 96-97

Examen Programmation Parallèle — Alain Griffault

Vendredi 31 janvier, 10 heures - 12 heures
Tous documents autorisés

Exercice 1 (8 points) : Ordonnancement de tâches avec des sémaphores

Un sémaphore S, à attente active, est un entier muni de deux primitives $P(S)$ et $V(S)$ qui permettent de le manipuler de façon atomique, P permet d'acquérir une ressource protégée par un sémaphore (lorsque l'entier est > 0) tout en décrémentant de 1 la valeur du sémaphore, et V permet de la libérer (en augmentant de 1 la valeur du sémaphore). Plus formellement :

P(S)	V(S)
tant que $S \leq 0$ faire rien $S := S - 1$	$S := S + 1$

Les sémaphores peuvent être utilisés pour garantir des ordonnancements entre tâches. Une première solution consiste à utiliser un sémaphore (initialisé à 0) par arête dans le graphe d'ordonnancement des tâches.

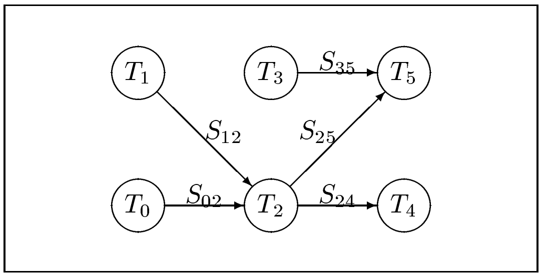

Figure 1: Ordonnancement avec 5 sémaphores.

Avec ces sémaphores, l'ordonnancement est obtenu, si chaque tâche exécute les codes suivants.

T_0	T_1	T_2	T_3	T_4	T_5
code V(S_{02})	code V(S_{12})	P(S_{02}) P(S_{12}) code V(S_{24}) V(S_{25})	code V(S_{35})	P(S_{24}) code	P(S_{25}) P(S_{35}) code

Question 1 *Expliquer en quelques lignes le fonctionnement de la tâche T_2.*

1. T_2 cherche a obtenir la ressource S_{02}. S'il veut l'obtenir trop tôt, il sera bloqué jusqu'à sa libération par T_0. S'il attend suffisemment, il obtiendra directement la ressource, car le sémaphore après l'action V(S_{02}) sera égal à 1.
2. idem pour la ressource S_{12} protégée initialement par T_1
3. T_2 exécute son code.
4. T_2 libère la ressource S_{24}, ce qui peut débloqué T_4.
5. T_2 libère la ressource S_{25}, ce qui peut débloqué T_5.

Une deuxième solution consiste à utiliser un sémaphore pour chaque tâche dépendante dans le graphe.

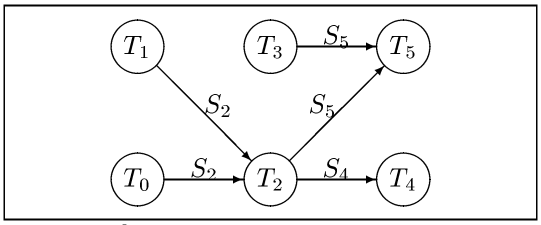

Figure 2: Ordonnancement avec 3 sémaphores.

Avec ces sémaphores, l'ordonnancement est obtenu, si chaque tâche exécute les codes suivants.

T_0	T_1	T_2	T_3	T_4	T_5
code	code	P(S_2)	code	P(S_4)	P(S_5)
V(S_2)	V(S_2)	code	V(S_5)	code	code
		V(S_4)			
		V(S_5)			

La deuxième solution diminue le nombre de sémaphores nécessaires, et diminue la nombre d'actions P et V nécessaires. En contre partie, les initialisations des valeurs des sémaphores ne sont pas toujours 0.

Question 2 *Donner les valeurs initiales des trois sémaphores, afin de garantir l'ordonnancement des tâches.*

Sémaphore	S_2	S_4	S_5
Valeur initiale	-1	0	-1

Un projet complexe est constitué de 12 tâches élémentaires. Les contraintes d'antériorités sont données par :

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}
après				T_1, T_2	T_2	T_2	T_4, T_5	T_3, T_6	T_3, T_6	T_9	T_7, T_8	T_{10}, T_{11}

On désire contrôler cet ordonnancement avec des sémaphores, utilisés comme dans la seconde solution.

Question 3 *Dessiner le graphe d'ordonnancement des tâches, en indiquant sur chaque arête le sémaphore utilisé.*

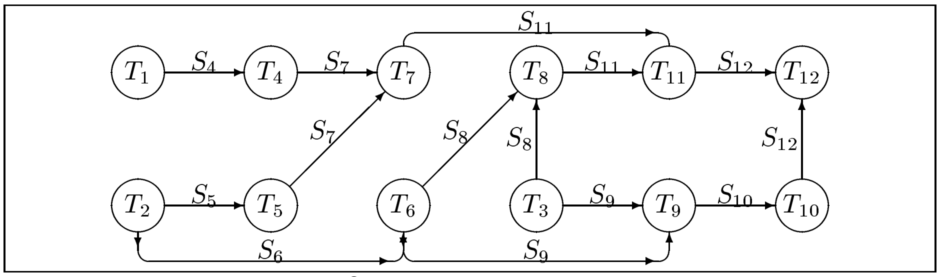

Figure 3: Ordonnancement des 12 tâches.

Question 4 *Placer avant ou après le code de chaque tâche, les primitives P et V nécessaires.*

T_1	T_2	T_3	T_4	T_5	T_6
code	code	code	P(S_4)	P(S_5)	P(S_6)
V(S_4)	V(S_4)	V(S_8)	code	code	code
	V(S_5)	V(S_9)	V(S_7)	V(S_7)	V(S_8)
	V(S_6)				V(S_9)

T_7	T_8	T_9	T_{10}	T_{11}	T_{12}
P(S_7)	P(S_8)	P(S_9)	P(S_{10})	P(S_{11})	P(S_{12})
code	code	code	code	code	code
V(S_{11})	V(S_{11})	V(S_{10})	V(S_{12})	V(S_{12})	

Question 5 *Pour chacun des sémaphores utilisés, donner les valeurs initiales.*

Sémaphore	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}
Valeur initiale	-1	0	0	-1	-1	-1	0	-1	-1

Exercice 2 (12 points): Un ascenseur

L'ascenseur que nous allons étudier dessert N étages. La cabine comporte N boutons lumineux (ie lorsqu'un bouton est allumé, il témoigne de l'existence d'une requête pour cet étage) qui permettent de choisir la (ou les) destination(s). De plus, à chaque étage se trouve un bouton de même type qui permet d'appeler l'ascenseur. Lorsque la cabine s'arrête à un étage, la porte s'ouvre alors automatiquement.
Le comportement de cet ascenseur est contrôlé par logiciel. Les changements d'états du contrôleur sont liés aux actions sur les boutons lumineux. Nous allons écrire un programme CSP et/ou OCCAM (comme un ensemble de tâches) qui nous permettra d'analyser ce comportement.
Une première tâche sert à représenter les actions des usagers sur les boutons.

```
Environnement ::
*[ BoutonEtage[i] ! appuyer --> SKIP
    ◇
   BoutonCabine[i] ! appuyer --> SKIP]
```

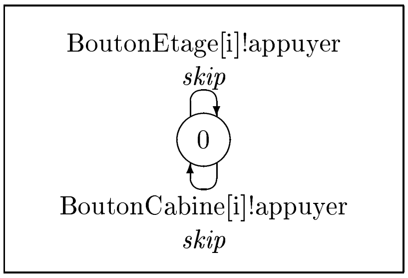

Figure 4: L'environnement.

Question 6 *Une deuxième tâche sert à représenter les comportements des boutons de la cabine. Dessiner le graphe des comportements correspondant au processus CSP suivant.*

```
BoutonsCabine[i] (i de 1 a N) ::
var Allume : booleen
Allume := false;
*[ Environnement ? appuyer and (not allume) -->
           Allume := true;
           Controleur ! requete;
    ◇
   Environnement ? appuyer and (allume) -->
           SKIP;
    ◇
   Porte[i] ? fermee -->
           Allume := false;]
```

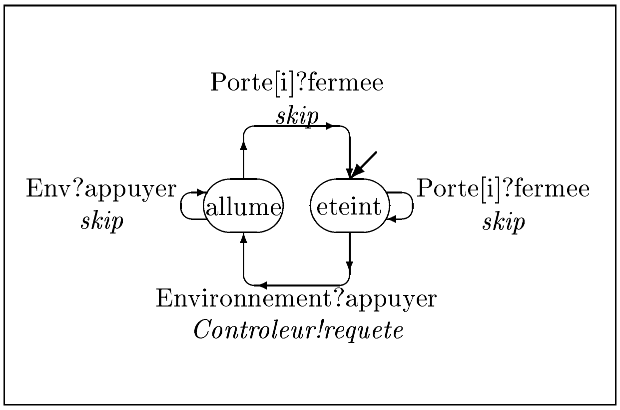

Figure 5: Un bouton de la cabine.

Question 7 *Une troisième tâche sert à représenter les comportements des boutons des étages. Dessiner le graphe des comportements ou donner le processus CSP associé.*

```
BoutonsEtage[i] (i de 1 a N) ::
var Allume : booleen
Allume := false;
*[
```

idem BoutonsCabine

idem bouton de la cabine

Figure 6: Le bouton d'un étage.

Question 8 *Une quatrième tâche sert à représenter les comportements des portes des étages. Donner le processus CSP associé au graphe suivant.*

```
Porte[i] (i de 1 a N) ::
var Ouverte : booleen
Ouverte := false;

*[Controleur ? delai and (not Ouverte) -->
           SKIP
    ◇
  Controleur ? delai and Ouverte -->
           Ouverte := false;
           BoutonCabine[i] ! fermee;
           BoutonEtage[i] ! fermee;
           Controleur ! fermee
    ◇
  Controleur ? ouvrir -->
           Ouverte := true]
```

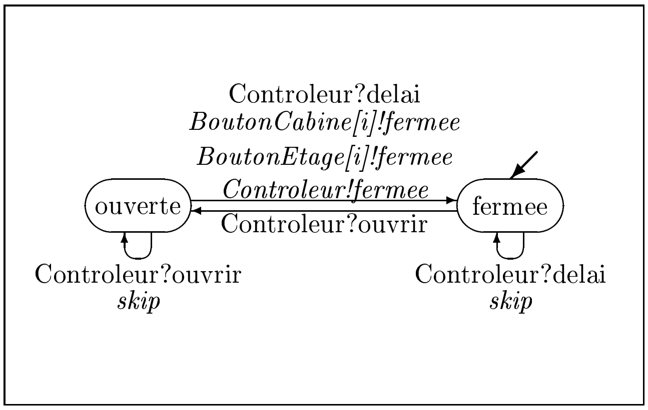

Figure 7: La porte d'un étage.

Question 9 *Une cinquième tâche sert à représenter les déplacements de la cabine. Donner le graphe associé au processus CSP suivant.*

```
Cabine ::
var Etage : 1..N
Etage := 1;
*[Controleur ? monter and (etage < N) -->
           Etage := Etage + 1;
           Controleur ! Etage
    ◇
  Controleur ? monter and (etage = N) -->
           SKIP
    ◇
  Controleur ? descendre and (etage > 1) -->
           Etage := Etage - 1;
           Controleur ! Etage
    ◇
  Controleur ? descendre and (etage = 1) -->
           SKIP]
```

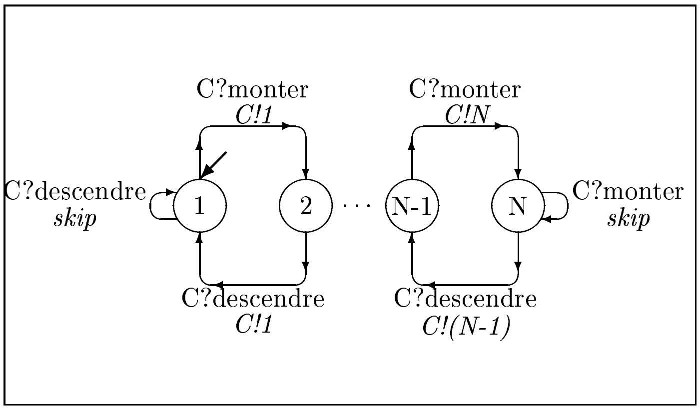

Figure 8: La cabine.

La dernière tâche sert à contrôler l'ascenseur. Le processus CSP suivant décrit une première solution.

```
Controleur ::
var Requete[1..N] : booleen
Requete[1..N] := false;
*[BoutonCabine[i] ? requete --> Requete[i] := true
    ◇
  BoutonEtage[i] ? requete --> Requete[i] := true
    ◇
  Porte[i] ? fermee --> Requete[i] := false
    ◇
  Cabine ? Etage -->
           +[Requete[Etage] --> Porte[Etage] ! ouvrir;
                                Porte[Etage] ! delai
               ◇
             not Requete[Etage] --> Cabine ! Monter
               ◇
             not Requete[Etage] --> Cabine ! Descendre]]
```

Question 10 *Aimeriez vous utiliser un ascenseur contrôlé par le source CSP précédent ? (Justifier votre réponse).*

⊤

Ce modèle d'ascenseur n'est pas très souhaitable, principalement pour les raisons suivantes :

Il présente des risques de blocage Cela signifie que l'ascenseur peut rester indéfiniment immobile.

- A partir de l'état initial, l'environnement peut appuyer sur les boutons, le contrôleur sera averti, mais ensuite rien ne se passe. En effet, le contrôleur attend alors de la cabine, mais celle-ci n'emet des messages vers le contrôleur qu'après en avoir reçu. Il y a donc interblocage.
- Si l'on suppose que l'interblocage précédent est corrigé, un autre apparaît : après l'ouverture, puis la fermeture de la porte, le contrôleur et la cabine attendent de nouveau mutuellement un message l'un de l'autre.

Il n'est pas équitable Cela signifie que certaines propriétés peuvent ne jamais être vraies.

- Il est possible d'attendre indéfiniment la cabine à un pallier.
- On peut, une fois dans la cabine, ne jamais aller à l'étage désiré.

⊥

Question 11 *Proposer un contrôleur qui corrige les défauts listés dans votre réponse à la question précédente.*

⊤

Nous doonons ici des idées (et non des preuves) pour corriger les défauts listés à la question précédente.

Il présente des risques de blocage Si la cabine signale systématiquement l'étage auquel elle se trouve au contrôleur, cela semble suffisant pour éviter les interblocages.

Il n'est pas équitable Une solution pour un contrôle équitable consiste à imposer à la cabine un mouvement de type aller-retour entre les extrémités. Cette première idée semble équitable, mais les utilisateurs n'apprécieront peut-être pas pour, du second étage, passer par le trentième pour aller au premier. Ce défaut semble pouvoir être corrigé en d'une part mémorisant le sens de déplacement, d'autre part en changeant de sens dès qu'il n'est plus nécessaire de continuer dans le même sens.

⊥