Partiel – 20 octobre 2014

J.-A. Anglès d'Auriac, R. Bonaque, M. Gleize, N. Sabouret

L'épreuve dure 2h00. Tous les documents sont autorisés. Les exercices sont indépendants.

Exercice 1 – Question de cours (2,5 points)

1. Qu'est-ce qu'un système d'exploitation ? (0,5 point)

   Correction : *L'OS est un programme ou un ensemble de programmes qui fait le lien entre le matériel et les applications.*

2. Qu'est-ce que la multiprogrammation et qu'est-ce que le temps partagé ? (1 point)

   Correction : *Le fait d'utiliser une ressource pendant qu'une autre application utilise le processeur, c'est la multiprogrammmation. Le temps partagé, c'est lorsque plusieurs processus partagent le temps sur l'UC.*

3. Dessinez les principales étapes du cycle de vie d'un processus (1 point)

   Correction : *Schéma avec début/fin et la boucle Prêt, en Exécution, en E/S.*

Exercice 2 – Ordonnancement (5 points)

On considère les processus suivants, définis par leur durée (réelle ou estimée), leur date d'arrivée, et leur priorité (les processus de priorité 0 étant les moins prioritaires) :

P1 durée : 8, date 0, priorité 1

P2 durée : 4, date 1, priorité 0

P3 durée : 3, date 2, priorité 1

P4 durée : 1, date 8, priorité 0

P5 durée : 2, date 12, priorité 1

1. Dessinez un diagramme de Gantt correspondant au résultat d'un ordonnancement par priorité et indiquez le temps d'attente moyen. (1,5 points)

   Correction :

   $$^0P1^8P3^{11}P2^{12}P5^{14}P4^{15}P2^{18}$$

   temps d'attente moyen = (0 + (10+3) + 6 + 6 + 0) / 5 = 5

2. Dessinez un diagramme de Gantt correspondant au résultat d'un ordonnancement « round robin » avec un quantum de temps fixé à 2 et indiquez le temps d'attente moyen. (1,5 points)

   Correction :

   $$^0P1^2P2^4P3^6P1^8P2^{10}P3^{11}P4^{12}P1^{14}P5^{16}P1^{18}$$

   Note : dans la suite, PX(Y) indique "PX, auquel il reste Y de temps d'exécution".

   À t=2, P2(4) et P3(3) sont arrivés dans la file, donc P2 prend la main et P1(6) est placé après P3 dans la file. Puis P3 prend la main jusqu'à t = 6, puis P1 jusqu'à t = 8. La file d'attente avant que P1 y revienne est ainsi : P2(2), P3(1).

   À t=8, P4(1) arrive et se place en fin de file, puis P1(4) rend la main et se place derrière. P2 prend la main et se termine à t=10. P3 prend la main et se termine à t=11. P4 s'exécute entièrement jusqu'à t=12.

   À t=12, P5(2) arrive et se place en fin de file, tandis que P1(4) prend la main. P1 et P5 sont les 2 seuls processus restants. P5 s'exécute entièrement de t=14 à t=16. P1(2) se termine enfin à t=18.

   Temps d'attente moyen = (10 + 5 + 6 + 3 + 2) / 5 = 5,2

3. On définit ainsi un algorithme d'ordonnancement à plusieurs niveaux :
   - Le niveau N correspond à exactement tous les processus de priorité N.
   - Le niveau 1 obéit à un ordonnancement « round robin », quantum 2 : entre eux, les processus de priorité 1 suivent cet ordonnancement.
   - Le niveau 0 obéit à un ordonnancement « plus court d'abord » non préemptif.
   - Le niveau 1 a priorité sur le niveau 0 : tous les processus de priorité 1 sont toujours prioritaires sur ceux de priorité 0, et ce de manière préemptive.

   Dessinez un diagramme de Gantt correspondant au résultat de cet ordonnancement et indiquez le temps d'attente moyen. (1,5 points)

   Correction :

   $$^0P1^2P3^4P1^6P3^7P1^{11}P4^{12}P5^{14}P2^{18}$$

   Note : dans la suite, PX(Y) indique "PX, auquel il reste Y de temps d'exécution".

   De t=0 à t=11, P1 et P3 sont les seuls processus qui s'exécutent, puisque de priorité 1, et alternent selon un round robin de quantum 2, en commençant par P1.

   À t=7, P1 s'exécute pendant 4 unités de temps, car il est le seul processus dans la file de niveau 1, et qu'il ne peut pas rendre la main à un processus de priorité inférieure.

   À t=11, P2(4) et P4(1) sont les processus dans la file de priorité 0, celle de la 1 étant vide. Donc on exécute P4, le plus court.

   À t=12, P5 arrive, donc a priorité sur P2.

   À t=14, la file de priorité 1 est de nouveau vide, donc P2 peut s'exécuter entièrement jusqu'à t=18.

   Temps d'attente moyen = (3 + 13 + 2 + 3 + 0) / 5 = 4,2

4. Quel est le meilleur algorithme suivant le critère du temps d'attente moyen ? Du temps d'attente min-max ? (0,5 point)

   Correction : *En temps d'attente moyen, c'est l'algo hybride. En min-max (le min du max), c'est le round robin pur.*

Exercice 3 – Encore de l'ordonnancement (3 points)

Dans l'exercice 2, nous avons considéré que l'ordonnanceur connaissait à l'avance le temps précis nécessaire à chaque processus pour se terminer. Cependant, il n'est pas toujours possible de déterminer cela à l'avance !

Dans cet exercice, nous allons étudier un algorithme d'ordonnancement plus court d'abord préemptif qui se base sur la moyenne des précédents temps d'exécution pour estimer le temps que prendra un processus. Comme vu en cours, le temps d'exécution est le temps passé sur le processeur entre deux entrée/sortie. Le modèle d'ordonnancement que nous utiliserons est un algorithme de type « plus court d'abord avec préemption » et fonctionne de la manière suivante :

- Le *temps estimé* d'un processus est la moyenne des précédents temps d'exécution. Lorsque le processus n'a pas encore effectué d'entrée/sortie, le *temps estimé* est infini.
- Pour un processus dans la file prêt ou en exécution, le *temps estimé restant* est le *temps estimé* moins le temps durant lequel le processus a effectivement été en exécution sur le processeur.
- Lorsqu'un processus passe en attente (requête d'E/S), c'est le processus prêt avec le plus court *temps estimé restant* qui est exécuté (ordonnancement au plus court d'abord)
- Lorsque qu'un processus revient dans la file prêt (par exemple en fin d'E/S), c'est le processus ayant le plus court *temps estimé restant* parmi les processus prêt et le processus actuellement exécuté qui obtient le processeur (algorithme préemptif).

On considérera maintenant le cas d'un processeur devant ordonnancer deux processus P1 et P2 décrits par la table ci-dessous. Chaque processus est une succession de tâches et d'E/S. Le processus P1 arrive au temps 0 mais le processue P2 arrive dans le système seulement au temps 1.

P1	P2
Tâche 1 : 4	Tâche 1 : 3
E/S 1 : 2	E/S 1 : 3
Tâche 2 : 2	Tâche 2 : 4
E/S 2 : 5	...
Tâche 3 : 1	...

1. Représentez le diagramme de Gant montrant à tout moment l'état de chaque processus : prêt, exécuté ou en attente d'entrée/sortie. De plus, vous indiquerez à chaque fois qu'un processus est prêt quel est son temps estimé, et quel est le temps estimé restant du processus en cours d'exécution. On considérera, pour simplifier, que le temps de commutation est nul. (2 points)

Correction :

Date	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
Exécution	*P1*				*P2*		*P1*		*P2*				*P2*				*P1*
Prêt		*P2*			-		*P2*		-					*P1*			
En attente					*P1*		-		*P1*	*P1+P2*							
TE P1	∞				*4*		*4*		*3*					*3*			*3*
TRE P1	∞				-		*4*	*3*	*2*					*3*			*3*
TE P2		∞			∞		∞		∞	*3*			*3*				*3.5*
TRE P2		∞			∞		∞		∞	-			*3*	*2*	*1*	*0*	*-1*

2. Y a-t-il un risque de famine avec cet algorithme lorsqu'il y a plus de deux processus ? Pourquoi ? Si oui, comment peut-on modifier l'algorithme pour l'éviter ? (1 point)

Correction : *Il y a un risque de famine dès qu'il y a plus de deux processus car le*

troisième, qui au départ a un TE infini, peut se faire passer devant alternativement par les deux autres. Concrête, P1 et P2 arrivent en premier et alternent des séquence calcul-E/S de même taille. Lorsqu'ils reviennent, ils sont toujours plus prioritaires que P3 qui n'a jamais eu la main donc reste avec son temps infini. Il y a a priori de nombreuses solutions possibles. Par exemple, ne pas commencer à l'infini (commencer à 0 : tout nouveau processus est prioritaire) et retirer 1 du temps estimé des processus prêts pour chaque autre processus qui les doublent dans la file.

Exercice 4 – Allocation mémoire (5 points)

On se place dans un système de mémoire de 32Ko de mémoire haute (c'est-à-dire au delà de la partie utilisée par le système) géré en allocation contigüe variable (« swapping » dans le cours). L'état initial de la mémoire représenté sur la Figure 1 ci-après. Certaines régions de la mémoire sont déjà allouées (zone grisées et marquées d'un X) et seront considérées comme constante pour cet exercice.

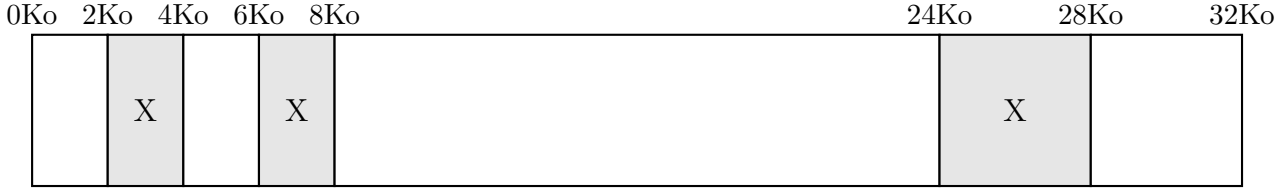


FIGURE 1 – Représentation de la mémoire haute disponible

Deux processus A et B demandent au système de leur allouer des blocs de mémoire (par example suite à des appels à la fonction *malloc*). Ils réalisent l'une après l'autre les opérations suivantes :

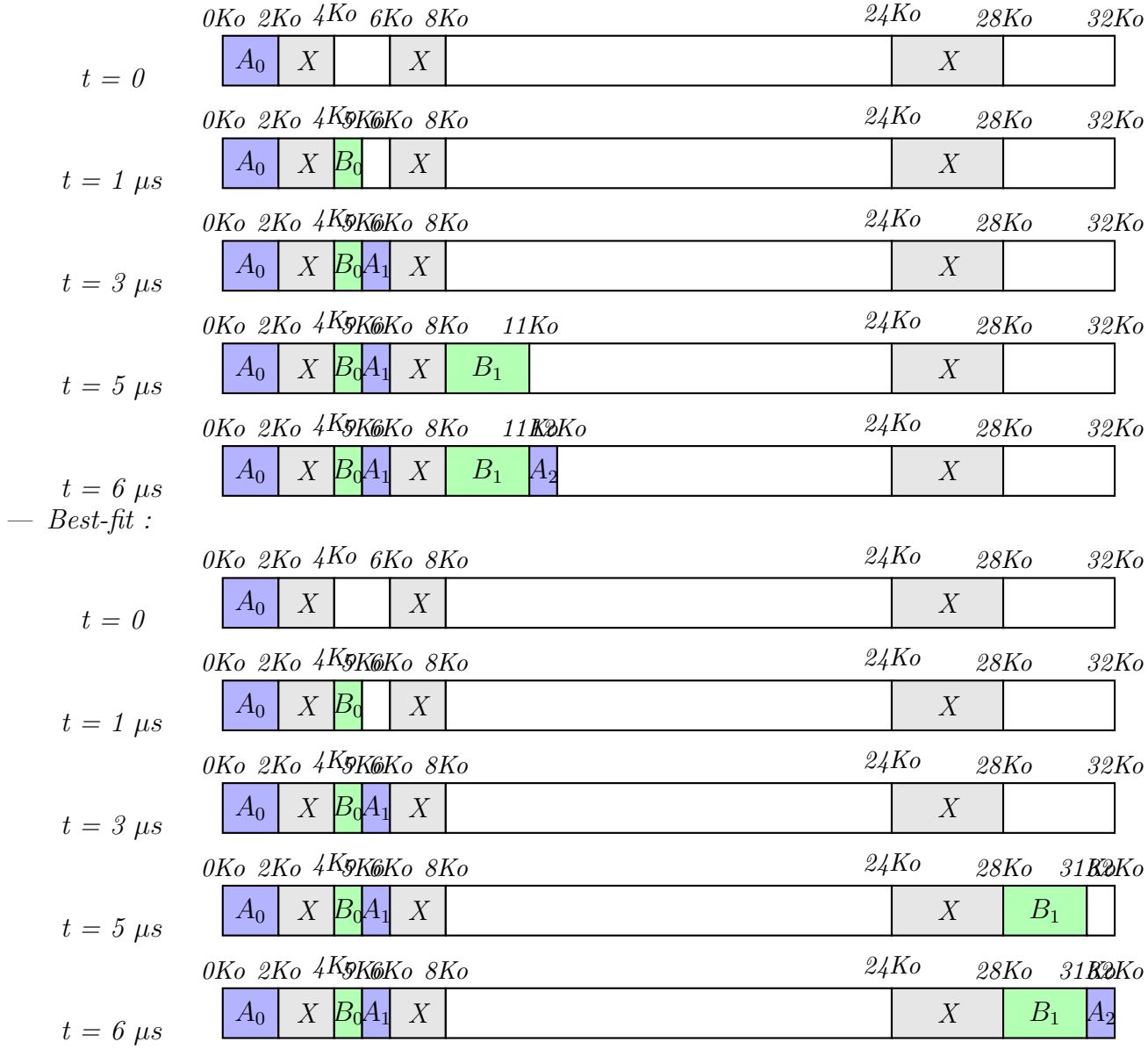
A	B
— demande un bloc de taille 2Ko, qu'on notera A_0 — attend $3\mu s$ — demande un bloc de taille 1Ko, noté A_1 — attend $3\mu s$ — demande un bloc de taille 1Ko, noté A_2 — attend $3\mu s$ — libère le dernier bloc qu'il a demandé (A_2)	— demande un bloc de taille 1Ko, qu'on notera B_0 — attend $4\mu s$ — demande un bloc de taille 3Ko, noté B_1 — attend $2\mu s$ — demande un bloc de taille 13Ko, noté B_2 — attend $3\mu s$ — libère le bloc de 3Ko qu'il a demandé (B_1) — attend $3\mu s$ — demande un bloc de 4Ko, noté B_3

1. On lance A à $t = 0\mu s$ et B à $t = 1\mu s$. Nous nous intéressons tout d'abord à l'état de la mémoire à $t = 6\mu s$, c'est-à-dire après l'allocation du bloc A_2 mais avant celle du bloc B_2.

   Donnez l'allocation mémoire obtenue avec l'algorithme **First-fit** (prochain bloc libre) et avec l'algorithme **Best-fit** (plus petit bloc libre). (2 points)

   Correction :

   — *First-fit :*

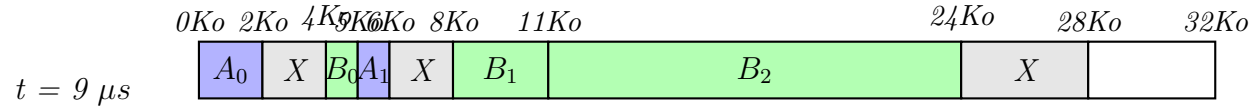


— *Best-fit :*

2. On suppose pour cette question que lorsqu'un processus demande un bloc de mémoire plus grand que que n'importe quel bloc libre il est mis en attente jusqu'au moment où un bloc suffisemment grand est disponible. Donnez et expliquez l'allocation mémoire à $t = 10\mu s$ pour chacun des deux algorithmes First Fit et Best Fit. Quel algorithme utilise le plus efficacement la mémoire sur cet exemple ? (1,5 points)

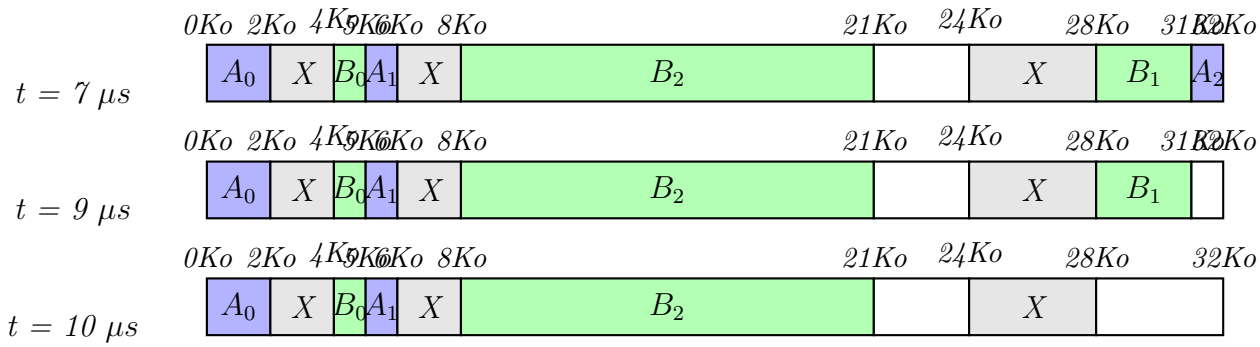
Correction : *Pour First-fit on n'a pas la place d'allouer un bloc de taille 13Ko : il ne reste qu'un bloc de taille 12Ko (entre les adresses 12Ko et 24Ko) et un bloc de taille 4 (entre 28Ko et 32Ko). On va donc devoir attendre qu'un bloc se libère à $t = 9\mu s$ pour pouvoir allouer le bloc qu'on aurait du allouer à $t = 7\mu s$ pour B, cela va donc décaler toutes les actions de B de $2\mu s$.*

First-fit :



Qui reste inchangé à $t = 10\mu s$.

Best-fit :



B a donc prit $2\mu s$ *de retard dans le cas du First-fit par rapport au Best-fit : c'est Best-fit qui a été le plus efficace dans sa gestion de la mémoire dans cette question.*

3. En fait, le système n'est pas obligé d'attendre qu'un bloc se libère : il peut refuser aux processus l'attribution de mémoire. Comment cela se traduit-il pour un programme C faisant un appel à la fonction *malloc* ? (0,5 point)

   Correction : *Si malloc ne permet pas d'allouer un bloc de mémoire la fonction retourne le pointeur null, i.e. l'entier* 0.

4. Les systèmes d'exploitation modernes incluent un mécanisme de swapping qui permet de déplacer des blocs de mémoire sur le disque lorsque la mémoire est saturée. Supposons que l'on puisse enlever de la mémoire haute (et déplacer vers le disque) un des blocs alloué au début (ceux marqués d'un X), quel est le plus petit bloc déjà que l'on peut enlever pour minimiser le temps pris par A et B pour exécuter toutes leurs opérations avec l'algorithme First-fit ? (1 point)

   Correction : *Il faut enlever le bloc situé entre 6 et 8 Ko : il est de plus petite taille et il permet d'effectuer toutes les opérations en* $13\mu s$ *ce qui est minimal car B démarre à* $t = 1\mu s$ *et attend au moins* $12\mu s$.

Exercice 5 – Synchronisation (5 points)

Un mutex fair est un mutex qui est acquis dans l'ordre ou il est demandé : si un thread t appelle *mutex.acquiere* avant le thread u alors *mutex.acquiere* retourne sur le thread t avant de retourner sur le thread u. On l'oppose au mutex classique (unfair) pour lequel *mutex.acquiere* peut retourner sur u d'abord.

1. Le pseudo code de la figure 2 décrit une tentative d'implémenter un mutex fair (FairMutex) à partir d'un mutex classique (Mutex). L'idée de l'algorithme est que lorsqu'un thread fait un appel à *acquiere* il est rajouté à une file d'attente (*file*), puis se met à attendre (sur le mutex *attente*). A chaque fois que le FairMutex est released on réveille tous les threads qui attendent et on les remet tous à attendre, sauf le premier de la file d'attente qui lui acquière le mutex.
   Ce pseudo code implémente t-il effectivement un mutex fair ? Pourquoi ?
   (2 points)

   Correction : *Cette tentative n'implémente pas un mutex fair, on peut citer plusieurs raisons :*
   — *section_critique est un mutex unfair, en particulier si on considère deux threads t et u tels que t invoque la méthode acquiere d'un FairMutex avant u il peut tout à fait attendre sur section_critique.acquiere() (ligne 10) alors que u acquière section_critique plus rapidement. Cela aboutit à mettre u sera mis dans la file avant t.*

```
1  class FairMutex
2
3      FIFO<int> file = new FIFO()
4      int nb_requetes = 0
5      Mutex section_critique = new Mutex()
6      Mutex attente = new Mutex()
7
8      void acquiere() {
9
10         section_critique.acquiere()
11         Mutex old_attente = attente
12         int identifieur = nb_requetes
13         nb_requetes++
14         file.empiler(identifieur)
15         section_critique.release()
16
17         while (old_attente.acquiere()) {
18             section_critique.acquiere()
19             if (file.tete == identifieur) {
20                 attente = new Mutex()
21                 attente.acquiere()
22                 old_attente.release()
23                 section_critique.release()
24                 return
25             } else {
26                 old_attente.release()
27                 old_attente = attente
28                 section_critique.release()
29             }
30         }
31     }
32
33     void release() {
34
35         section_critique.acquiere()
36         file.depiler()
37         attente.release()
38         section_critique.release()
39     }
40 }
```

FIGURE 2 – Pseudo code du mutex fair pour la question 1

— *le réveil des threads qui attendent sur attente peut ne pas marcher, en effet chaque thread réveillé relâche le mutex puis tente d'acquérir le nouveau mutex d'attente (lignes 26, 27 et 17) mais, à moins que le thread en tête de file ne se réveille, rien ne change le mutex d'attente il est donc possible d'avoir un unique thread qui récupère toujours le mutex en bouclant entre les lignes 17-19 et 25-28.*

2. Le pseudo code de la figure 3 décrit une tentative d'implémenter un sémaphore à partir de mutex fairs (*section_critique* et *attente* sont fair ici).
   Ce pseudo code implémente t-il effectivement un sémaphore ?
   (1.5 points)

   Correction : *Ce pseudo code implémente effectivement un sémaphore : chaque thread qui exécute Semaphore.acquiere va tenter de prendre une ressource du sémaphore, si celle-ci sont épuisées le thread attend (à l'aide de attente) qu'une ressource soit relâchée.*

3. Quel est l'intérêt d'utiliser des mutex fairs ? (0.5 point)

   Correction : *L'intérêt d'utiliser des mutex fairs est de rendre le sémaphore fair lui même.*

4. Ce sémaphore ne peut acquérir ou relâcher qu'une seule ressource à la fois. Peut-on utiliser une boucle for qui exécute *acquiere* ou *release* plusieurs fois pour pouvoir acquérir (ou relâcher) plusieurs ressources ? Pourquoi ? (1 point)

   Correction : *Non : si on se contente d'une simple boucle for rien n'empêche d'avoir un switch vers un autre thread au milieu, ce qui peut créer un interblocage. Par example si il existe deux ressources et que deux threads veulent en acquérir chacun deux, si un premier thread en acquière une puis qu'on switch sur le deuxieme thread qui acquière l'autre il ne pourront jamais acquérir une deuxième ressource et donc relâcher celle qu'ils tiennent. Une manière de corriger cela serait de rentrer en section critique pour faire la boucle for, même si ce n'est pas la méthode la plus efficace.*

```
1  class Semaphore
2
3      int resources_disponibles
4      Mutex section_critique = new Mutex() // fair mutex
5      Mutex attente = new Mutex() // fair mutex
6
7      Semaphore(int resources_initiales) {
8          resources_disponibles = resources_initiales
9          attente.acquire()
10     }
11
12     void acquiere() {
13
14         section_critique.acquiere()
15         resources_disponibles--
16         if (resources_disponibles < 0) {
17             section_critique.release()
18             attente.acquire()
19         } else {
20             section_critique.release()
21         }
22     }
23
24     void release() {
25
26         section_critique.acquiere()
27         resources_disponibles++
28         if (resources_disponibles < 0) {attente.release()}
29         section_critique.release()
30     }
31 }
```

FIGURE 3 – Pseudo code du semaphore pour la question 2