Question 1. *Remplacez chaque* **`notifyAll()`** *par* **`notify()`** *et testez le système. Observez-vous un blocage ?*

Les étudiants auront de fortes chances d'observer un blocage, s'ils essaient suffisamment de fois. En pratique, les blocages sont souvent difficiles à détecter car ils peuvent résulter d'un entrelacement très particulier qui survient rarement.

Par ailleurs, le blocage pourra éventuellement ne pas apparaitre avec certaines machines virtuelles.

Question 2. *Ce programme peut conduire à une exécution qui s'affichera sur l'écran comme ceci :*

```
$ java Buffer
[ 34 ]
[ ]
```

bloqué pendant 5 secondes et donc manifestement bloqué. Pouvez-vous expliquer ce bug ?

L'idée principale est qu'après avoir remplacé chaque **`notifyAll()`** par **`notify()`**, on risque de ne pas envoyer le signal sur le thread susceptible d'agir.

Le buffer étant synchronisé et les méthodes déterministes, l'état du buffer au cours d'une exécution de ce programme sera entièrement défini par l'ordre des accès au verrou intrinsèque du buffer résultant des appels aux méthodes de cet objet. L'état de chaque thread en revanche dépend du thread qui obtient le verrou après chaque libération et de celui qui reçoit le signal lors de chaque appel de **`notify()`**.

Une situation de blocage pourrait apparaître si un producteur réalise un dépôt et si à la fin du dépôt, lors de l'envoi du signal, le second producteur est en attente, de même que les deux consommateurs. C'est le cas si les deux consommateurs parviennent à saisir le verrou intrinsèque en premier et trouvent le buffer vide (il y a en gros une chance sur quatre). Si le signal envoyé lors du premier dépôt est reçu par le second producteur, celui-ci attendra à nouveau car le buffer est plein ; en même temps, les deux consommateurs demeureront en attente d'un signal. Ce scenario suppose néanmoins que le second producteur soit en attente lorsque le premier producteur termine le premier dépôt : il n'y a aucune raison que ceci puisse se produire, car le second producteur ne pourra avoir exécuté l'instruction **`wait()`** puisqu'il lui faudrait posséder le verrou pour cela.

Un scenario légèrement plus complexe peut néanmoins être imaginé.

① Initialement, le buffer est vide et les deux consommateurs tentent l'un après l'autre un retrait : ils passent chacun dans l'état WAITING ;

② Un producteur dépose un octet dans le buffer et envoi un signal à l'un des consommateurs, qui passe dans l'état BLOCKED, puis relâche le verrou ;

③ Le verrou est pris successivement par les deux producteurs, qui passent tous les deux dans l'état WAITING, puisque le buffer est plein ;

④ Le consommateur dans l'état BLOCKED prend le verrou pour retirer le premier octet déposé et envoie un signal au second consommateur, qui passe dans l'état BLOCKED ;

⑤ Le consommateur relâche le verrou et le reprend immédiatement pour tenter un second retrait : il passe alors dans l'état WAITING, puisque le buffer est vide ;

⑥ Le second consommateur prend le verrou et passe lui aussi dans l'état WAITING, puisque le buffer est vide.

À l'issue de ce scenario, les deux producteurs et les deux consommateurs sont dans l'état WAITING.

Ce blocage peut ainsi être expliqué plus formellement par le scénario détaillé décrit par le diagramme de séquence de la Table 29 ou le tableau récapitulatif de la Table 30.

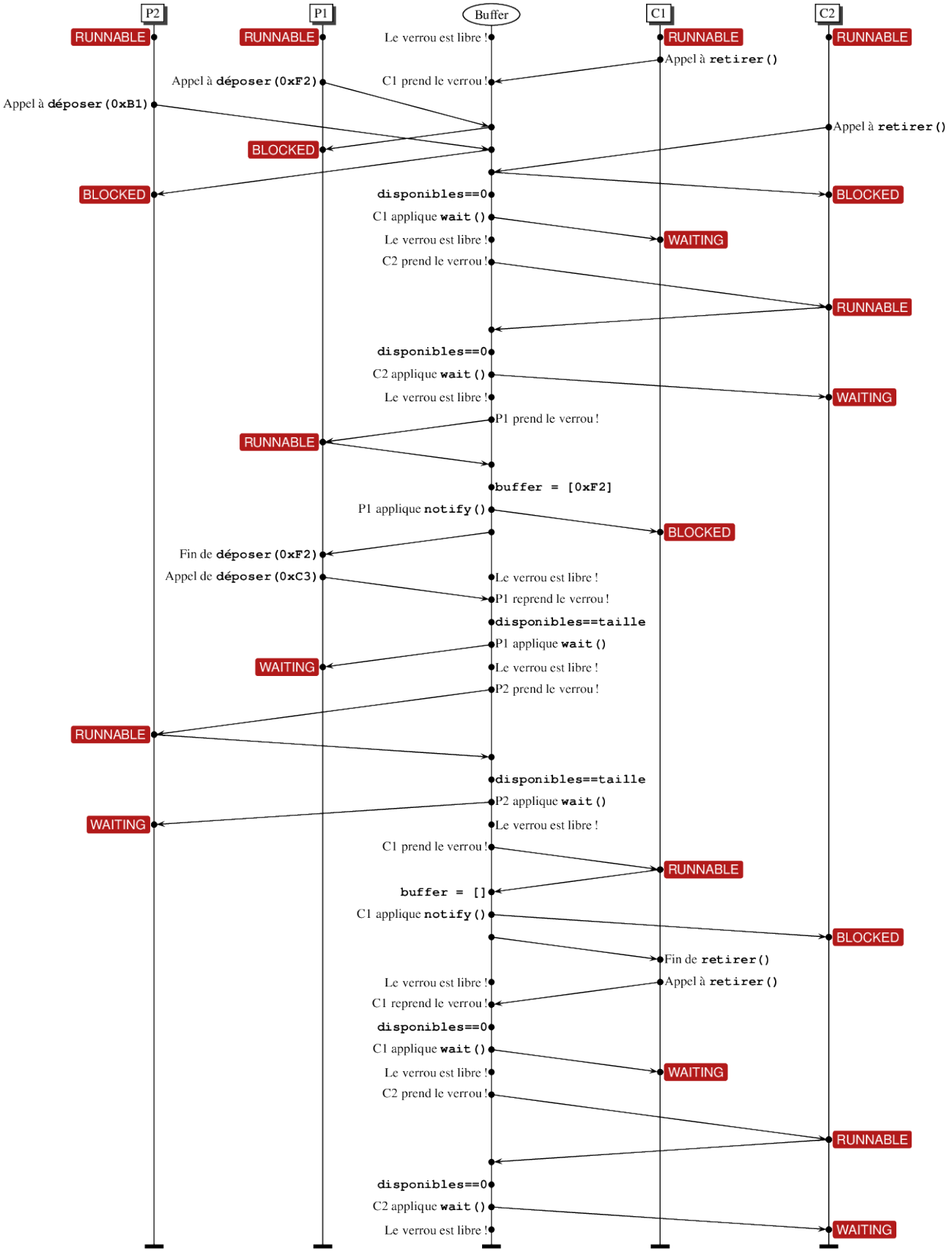

TABLE 29 – Diagramme de séquence décrivant le bug du buffer circulaire

buffer	verrou	action	P2	P1	C1	C2
	Initialement le buffer est vide et tous les threads sont dans l'état RUNNABLE.					
`[]`	libre		RUNNABLE	RUNNABLE	RUNNABLE	RUNNABLE
	Le premier consommateur (C1) passe dans l'état WAITING car le buffer est vide. Les autres threads passent dans l'état BLOCKED (car le verrou est pris).					
`[]`	C1	`wait()`	BLOCKED	BLOCKED	WAITING	BLOCKED
	Le second consommateur passe dans l'état WAITING car le buffer est vide.					
`[]`	C2	`wait()`	BLOCKED	BLOCKED	WAITING	WAITING
	Le premier producteur insère l'octet 34 (0xF2) et envoie un signal à C1.					
`[0xF2]`	P1	`notify()`	BLOCKED	RUNNABLE	BLOCKED	WAITING
	Le premier producteur tente d'insérer un second octet et passe dans l'état WAITING.					
`[0xF2]`	P1	`wait()`	BLOCKED	WAITING	BLOCKED	WAITING
	Le second producteur tente d'insérer un second octet et passe aussi dans l'état WAITING.					
`[0xF2]`	P2	`wait()`	WAITING	WAITING	BLOCKED	WAITING
	Le premier consommateur consomme l'octet 34 placé dans le buffer. Le signal est adressé au second consommateur qui passe dans l'état BLOCKED.					
`[]`	C1	`notify()`	WAITING	WAITING	RUNNABLE	BLOCKED
	Le premier consommateur tente de consommer un octet et repasse dans l'état WAITING.					
`[]`	C1	`wait()`	WAITING	WAITING	WAITING	BLOCKED
	Le second consommateur tente de consommer un octet et repasse dans l'état WAITING.					
`[]`	C2	`wait()`	WAITING	WAITING	WAITING	WAITING
	Tous les threads sont dans l'état WAITING. Aucun signal ne peut arriver...					

TABLE 30 – Tableau explicatif décrivant le bug du buffer circulaire

Question 1. *Observez sur une exécution que plusieurs sauvages peuvent attendre simultanément que le pôt soit rempli par le cuisinier. Selon vous, dans quel état sont ces threads ?*
Les sauvages ont exécuté **`wait()`** : ils sont dans l'état WAITING.

Question 2. *Observez également que si plusieurs sauvages sont en attente du cuisinier, le premier sauvage ayant réveillé le cuisinier n'est pas toujours le premier à se servir lorsque le pôt est plein.*
Sur les machines des salles de TP, en lançant une exécution soit dans un PowerShell sous Windows soit dans un Terminal sous Linux, chaque étudiant doit pouvoir observer la situation indiquée, au bout d'un certain temps. Lorsque plusieurs sauvages attendent devant le pôt vide après avoir réveillé le cuisinier, le premier arrivé est rarement le premier servi (même si, en pratique, certaines machines virtuelles semblent le permettre). On obtiendra par exemple à l'écran :

```
$ java Diner.java
Il y a 10 sauvages.
Le pôt peut contenir 3 portions.
S0: J'ai faim!
    S0: Il y a une part disponible !
S2: J'ai faim!
    S2: Il y a une part disponible !
S9: J'ai faim!
    S9: Il y a une part disponible !
S6: J'ai faim!
    S6: Le pôt est vide!
        S6: Je réveille le cuisinier.
        S6: J'attends que le pôt soit plein!
Cuisinier: Je cuisine...
S7: J'ai faim!
S5: J'ai faim!
S2: J'ai faim!
S8: J'ai faim!
S4: J'ai faim!
Cuisinier: Le pôt est plein!
        S6: Je me réveille! Je me sers.
    S4: Il y a une part disponible !
    S8: Il y a une part disponible !
    S2: Le pôt est vide!
        S2: Je réveille le cuisinier.
        S2: J'attends que le pôt soit plein!
    S5: Le pôt est vide!
        S5: Je réveille le cuisinier.
        S5: J'attends que le pôt soit plein!
    S7: Le pôt est vide!
        S7: Je réveille le cuisinier.
        S7: J'attends que le pôt soit plein!
Cuisinier: Je cuisine...
S4: J'ai faim!
S1: J'ai faim!
S9: J'ai faim!
S0: J'ai faim!
S8: J'ai faim!
S6: J'ai faim!
S3: J'ai faim!
Cuisinier: Le pôt est plein!
        S7: Je me réveille! Je me sers.
        S2: Je me réveille! Je me sers.
        S5: Je me réveille! Je me sers.
    S3: Le pôt est vide!
^C$
```

On constatera ici que le sauvage S2 est le premier à réveiller le cuisinier, mais qu'il n'est pas le premier à se servir.

```
PS C:\Users\morin.r\Downloads\TP_D\TP_D\3_-_Pôt_des_sauvages> java .\Diner.java
Il y a 10 sauvages.
Le pôt peut contenir 3 portions.
S3: J'ai faim!
        S3: Il y a une part disponible !
S1: J'ai faim!
        S1: Il y a une part disponible !
S9: J'ai faim!
        S9: Il y a une part disponible !
S2: J'ai faim!
        S2: Le pôt est vide!
                S2: Je réveille le cuisinier.
                S2: J'attends que le pôt soit plein!
Cuisinier: Je cuisine...
S1: J'ai faim!
S0: J'ai faim!
S7: J'ai faim!
S3: J'ai faim!
S6: J'ai faim!
S4: J'ai faim!
S8: J'ai faim!
Cuisinier: Le pôt est plein!
                S2: Je me réveille! Je me sers.
        S8: Il y a une part disponible !
        S4: Il y a une part disponible !
        S6: Le pôt est vide!
                S6: Je réveille le cuisinier.
                S6: J'attends que le pôt soit plein!
        S3: Le pôt est vide!
                S3: Je réveille le cuisinier.
                S3: J'attends que le pôt soit plein!
        S7: Le pôt est vide!
                S7: Je réveille le cuisinier.
                S7: J'attends que le pôt soit plein!
        S0: Le pôt est vide!
                S0: Je réveille le cuisinier.
                S0: J'attends que le pôt soit plein!
        S1: Le pôt est vide!
                S1: Je réveille le cuisinier.
                S1: J'attends que le pôt soit plein!
Cuisinier: Je cuisine...
S5: J'ai faim!
S8: J'ai faim!
S9: J'ai faim!
Cuisinier: Le pôt est plein!
                S1: Je me réveille! Je me sers.
                S6: Je me réveille! Je me sers.
                S3: Je me réveille! Je me sers.
        S9: Le pôt est vide!
                S9: Je réveille le cuisinier.
                S9: J'attends que le pôt soit plein!
```

TABLE 31 – Inéquité du pôt sous Windows

```
$ java Diner.java
Il y a 10 sauvages.
Le pôt peut contenir 3 portions.
S6: J'ai faim!
        S6: Il y a une part disponible !
S5: J'ai faim!
        S5: Il y a une part disponible !
S4: J'ai faim!
        S4: Il y a une part disponible !
S9: J'ai faim!
        S9: Le pôt est vide!
                S9: Je réveille le cuisinier.
                S9: J'attends que le pôt soit plein!
Cuisinier: Je cuisine...
S7: J'ai faim!
S2: J'ai faim!
S0: J'ai faim!
S1: J'ai faim!
Cuisinier: Le pôt est plein!
                S9: Je me réveille! Je me sers.
        S1: Il y a une part disponible !
        S0: Il y a une part disponible !
        S2: Le pôt est vide!
                S2: Je réveille le cuisinier.
                S2: J'attends que le pôt soit plein!
        S7: Le pôt est vide!
                S7: Je réveille le cuisinier.
                S7: J'attends que le pôt soit plein!
Cuisinier: Je cuisine...
S8: J'ai faim!
S6: J'ai faim!
S5: J'ai faim!
S3: J'ai faim!
S9: J'ai faim!
Cuisinier: Le pôt est plein!
                S7: Je me réveille! Je me sers.
                S2: Je me réveille! Je me sers.
        S9: Il y a une part disponible !
        S3: Le pôt est vide!
                S3: Je réveille le cuisinier.
                S3: J'attends que le pôt soit plein!
        S5: Le pôt est vide!
                S5: Je réveille le cuisinier.
                S5: J'attends que le pôt soit plein!
        S6: Le pôt est vide!
                S6: Je réveille le cuisinier.
                S6: J'attends que le pôt soit plein!
        S8: Le pôt est vide!
                S8: Je réveille le cuisinier.
                S8: J'attends que le pôt soit plein!
Cuisinier: Je cuisine...
S4: J'ai faim!
^C
```

TABLE 32 – Inéquité du pôt sous Linux

Question 3. *Il s'agit dans cet exercice de garantir que le sauvage qui réveille le cuisinier en premier soit toujours le premier à se servir lorsque le pôt est plein. Pour celà, l'idée proposée est d'ajouter une louche au pôt et d'exiger qu'un sauvage prenne la louche pour se servir et la garde pour lui lorsqu'il réveille le cuisinier. Intuitivement, il faut la louche pour se servir mais aussi pour réveiller le cuisinier ! Décommentez les lignes proposées dans le code fourni afin d'ajouter l'emploi de la louche (vue comme un verrou) dans le comportement des sauvages.*
La louche est utilisée seulement dans la méthode **`seServir()`** : il suffit donc de décommenter les trois premières lignes et les trois dernières lignes de cette méthode (Fig. 26).

Question 4. *De manière abstraite, un interblocage est caractérisé par un ensemble de processus dans lequel chaque processus attend un évènement que seul un autre processus de cet ensemble peut provoquer. Pouvez-vous décrire l'interblocage observé à la question précédente ? Combien de sauvages sont impliqués dans cet interblocage ? Dans quel état sont le cuisinier et le sauvage qui tient la louche ?*
Cette situation d'interblocage est par exemple illustrée par l'exécution reproduite sur la Figure 27. Avec un peu de chance et de patience, chaque étudiant pourra observer cette situation de blocage sur les machines en salle de TP.

D'un point de vue formel, si on construit le graphe orienté dont les noeuds sont les processus qui participent à l'interblocage et dont les arrêtes matérialisent l'attente par un processus d'une action espérée d'un autre processus, ce graphe contiendra nécessairement un *cycle* (car le nombre de processus est fini). La technique classique vue en cours consistant à ordonner totalement les verrous et toujours prendre les verrous dans cet ordre, lorsque plusieurs verrous sont requis, interdit qu'un cycle puisse apparaître, puisqu'un ordre total ne contient pas de cycles : il ne peut donc pas y avoir d'interblocage si l'on applique cette technique. C'est anecdotique, mais nous verrons que le code avec la louche ne respecte pas la technique vue en cours : il y a effectivement deux verrous et ces deux verrous ne sont pas toujours pris dans le même ordre.

Dans cet exercice, l'interblocage fait intervenir trois threads : deux sauvages et le cuisinier.
— Le premier sauvage arrive devant le pôt vide : il tient la louche en réveillant le cuisinier qui passe de l'état WAITING à l'état BLOCKED, en tenant la louche.
— Le premier sauvage s'endort ensuite sur le pôt en relâchant le verrou intrinsèque de cet objet, *mais en conservant la louche* : il est alors dans l'état WAITING.
— Un second sauvage souhaite se servir et prend le verrou intrinsèque du pôt pour appliquer la méthode **`seServir()`**. Il ne peut néanmoins pas prendre la louche, déjà saisie par le premier sauvage : il passe alors dans l'état BLOCKED *en conservant le verrou intrinsèque du pôt.*
— Le cuisinier souhaite redémarrer mais ne peut sortir de l'état BLOCKED, car le verrou intrinsèque du pôt est déjà pris par le second sauvage.

Au total,
— Le premier sauvage est dans l'état WAITING : il attend que le pôt soit rempli par le cuisinier ;
— Le cuisinier est dans l'état BLOCKED : il attend que le verrou intrinsèque du pôt soit libéré par le second sauvage ;
— Le second sauvage est dans l'état BLOCKED : il attend que le premier lâche la louche.

On a donc bien un interblocage impliquant essentiellement ces trois threads. Notez que les autres sauvages seront également bloqués puisqu'ils ne pourront pas se servir non plus, car ils attendent, comme le cuisinier, que le verrou intrinsèque du pôt soit libéré par le second sauvage.

Notez qu'il y a deux verrous : la louche et le verrou intrinsèque du pôt. En général, un sauvage prend le verrou du pôt puis celui de la louche. Mais si le pôt est vide, le sauvage relâche le verrou du pôt lors de l'instruction **`wait()`** et doit l'acquérir à nouveau lors de son réveil : il saisit donc le verrou du pôt *après* avoir saisi la louche.

Question 5. *Corrigez le programme faisant de l'attribut* **`louche`** *un simple booléen et en adaptant le reste du code du pôt afin d'utiliser correctement cette louche (c'est-à-dire en supprimant l'interblocage). Il s'agit ici de s'assurer qu'un sauvage qui arrive devant le pôt vide alors que la louche est déjà prise n'empêche pas le cuisinier de remplir le pôt.*

Un code corrigé est présenté sur la Table 33. Les modifications opérées sur le code sont les suivantes :

(a) On remplace

```
private ReentrantLock louche = new ReentrantLock(true) ;
```

par

```
private boolean loucheLibre = true ;
```

(b) On remplace

```
louche.lock();
```

par

```
while( ! loucheLibre ) wait() ;
loucheLibre = false ;
```

Prendre la louche correspond désormais à écrire **`false`** à la place de **`true`** de manière atomique dans la variable booléenne **`loucheLibre`**.

(c) On remplace

```
louche.unlock();
```

par

```
loucheLibre = true ;
notifyAll();
```

Relâcher le louche revient à écrire true dans la variable booléenne **`loucheLibre`**. Puisque des threads sont susceptibles d'attendre la louche, il est naturel d'ajouter **`notifyAll()`** à la suite.

De la sorte, un sauvage qui attend la louche relâchera le verrou intrinsèque du pôt pendant son attente.