Les sémaphores

Orestis Malaspinas, Steven Liatti

1 Les sémaphores

Le contenu de ce chapitre est basé sur l'excellent livre de R. H. Arpaci-Dusseau et A. C. Arpaci-Dusseau [1].

Le sémaphore est une autre primitive de synchronisation inventée par E. Dijkstra. En fait, les sémaphores peuvent être utilisés pour remplacer toutes les autres primitives de synchronisation que nous avons vues.

Ici, nous allons voir comment utiliser les sémaphores pour synchroniser des threads à l'intérieur d'un même processus, mais ces objets sont beaucoup plus génériques et peuvent aussi servir pour synchroniser *différents processus*.

1.1 Définition

Un sémaphore est une structure contenant un nombre entier que nous pouvons manipuler avec deux fonctions. Dans l'API POSIX elles s'appellent `sem_wait()` et `sem_post()` [2]. Comme un sémaphore contient un entier, il faut l'initialiser avec la fonction `sem_init()`. L'initialisation prend trois paramètres, une pointeur vers le sémaphore, de type `sem_t`, un entier, `pshared` spécifiant s'il est partagé entre les threads d'un même processus (dans ce cas il vaut 0) ou entre processus (il vaut une valeur différente de zéro), et la valeur de l'entier non-signé contenu dans le sémaphore, `value`. Comme toujours cette fonction retourne 0 en cas de succès.

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Un sémaphore "à un" sera ainsi initialisé comme

```
#include <semaphore.h>
sem_t s;
int sem_init(&s, 0, 1);
// 0 est pour un sémaphore entre les threads d'un même processus
// 1 pour initialiser le sémaphore à un.
```

Avant de nous intéresser à leur implémentation [3], intéressons-nous aux fonctions

1. R. H. Arpaci-Dusseau et A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, Arpaci-Dusseau Books, ed. 0.91, (2015).
2. Historiquement ces fonctions s'appelaient `P()` et `V()` pour "prolaag" (contraction voulant dire essayer et diminuer) et "verhoog" pour augmenter en néerlandais, la langue de E. Dijkstra.
3. Il est clair que comme plusieurs threads vont appeler ces fonctions elles devront contenir des sections critiques qu'il faudra gérer avec soin. On suppose pour le moment que toutes ces opérations sont atomiques.

`sem_wait()` et `sem_post()` qui servent à interagir avec un sémaphore initialisé. En particulier, voyons comment les utiliser.

Comme décrit dans le pseudo-c ci-dessous, les fonction `sem_wait()` va décrémenter la valeur du sémaphore de un et retourner immédiatement si la valeur du sémaphore est plus grande ou égale à zéro, sinon se mettre en sommeil.

```
int sem_wait(sem_t *s) {
    // décrémente de un la valeur du sémaphore
    // si la valeur du sémaphore est < 0 s'endormir
}
```

Si `sem_wait()` est appelé plusieurs fois à la suite les threads mis en sommeil sont insérés dans une file, en attente de réveil. On constate donc que si le sémaphore est négatif, sa valeur correspond au nombre de fils qui sont en attente. Bien que ce ne soit pas forcément utile en pratique, c'est une bonne chose à se rappeler pour comprendre comment fonctionnent les sémaphores.

A l'inverse, `sem_post()` incrémentera la valeur du sémaphore de un, et s'il y a un thread ou plus en sommeil, en réveiller un.

```
int sem_post(sem_t *s) {
    // incrémente de un la valeur du sémaphore
    // réveiller un thread endormi, si au moins un dort
}
```

1.2 Les sémaphores comme verrous

Notre but ici est de construire un verrou à l'aide d'un sémaphore.

Question 1

Pouvez-vous imaginer comment faire?

En fait, il suffit d'initialiser le sémaphore à un, et d'avoir un appel à `sem_wait()` et à `sem_post()`, avant et après la section critique que nous souhaitons protéger avec le verrou, comme on peut le voir dans le code ci-dessous.

```
sem_t s;
sem_init(&s, 0, 1);

sem_wait(&s);
// Section critique
sem_post(&s);
```

Pour être sûr que j'ai bien compris ce qui se passe ici, je vais expliquer plus en détails ce qui se passe. Le sémaphore est initialisé à un. Puis, le premier thread, T_1, qui appelle `sem_wait()` va décrémenter sa valeur à `0` et retourner (comme la valeur du sémaphore est $\geq$`0`).

Si aucun thread n'appelle `sem_wait()`, avant que T_1 appelle `sem_post()`, la valeur du sémaphore sera remise à un et un autre thread pourra appeler `sem_wait()`

et retourner immédiatement.

Si en revanche, pendant que T_1 est dans la section critique, un autre thread, T_2, appelle `sem_wait()`, la valeur du verrou sera décrémentée à -1 et T_2 sera mis en sommeil. Lorsque T_1 sera à nouveau ordonnancé, il appellera `sem_post()`, incrémentera le sémaphore de un (il aura la valeur `0`) et réveillera T_2 qui pourra entrer dans la section critique. Lorsque T_2 sort de la section critique et appelle `sem_post()` à son tour, il fait à nouveau passer le sémaphore à un et n'a pas de thread à réveiller.

Question 2

Que se passe-t-il avec trois threads?

Si maintenant, nous avons T_1, T_2 et T_3, trois threads qui tentent d'acquérir le verrou. T_1 arrive en premier `sem_wait()`, il décrémente le sémaphore à 0 et entre dans la section critique. Si T_2 appelle `sem_wait()` à son tour avant que T_1 soit sorti de la section critique, le sémaphore passe à `-1` et le thread se retrouve en sommeil. Finalement, si T_3 arrive également à appeler `sem_wait()` avant que T_1 soit sorti de la section critique, alors le sémaphore passe à `-2` et T_3 est mis en sommeil. Finalement, lorsque T_1 arrive enfin à sortir de la section critique, appelle `sem_post()`, ce qui fait passer le sémaphore à `-1` et réveille un thread (disons T_2). Celui-ci à son tour appelle `sem_post()`, le sémaphore passe à `0` et T_3 est réveillé. Il appelle également `sem_post()` ce qui incrémente le sémaphore à `1` à nouveau et ne réveille personne car personne ne peut être réveillé. Le sémaphore étant à un de nouveau le "verrou-sémaphore" peut être acquis par un autre thread éventuellement.

1.3 Les sémaphores pour ordonner une séquence

Les sémaphores peuvent aussi être utiles pour ordonner des événements dans un programme concurrent. On peut par exemple souhaiter attendre qu'une liste se remplisse pour récupérer des éléments à l'intérieur. Nous avons déjà utilisé les variables de condition de cette façon lorsque nous avions des threads qui attendaient qu'on leur signale que l'état de l'application avait changé.

On peut utiliser les sémaphores de façon similaire. Imaginons le cas très simple d'un thread thread T_1 créant un thread T_2, et que T_1 attende que T_2 se termine. Un exemple de ce genre de programme pourrait être le suivant.

```
sem_t s;

void *t2(void *arg) {
    printf("Eh bien, tu vas attendre.\n");
    sem_post(&s); // signal here
    return NULL;
}

int main() {
    sem_init(&s, 0, 0);
```

```
    printf("J'aimerais bien attendre...\n");

    pthread_t tid;
    pthread_create(&tid, NULL, t2, NULL);
    sem_wait(&s);

    printf("Merci de m'avoir fait attendre.\n");

    return EXIT_SUCCESS;
}
```

Comme on le voit ici le sémaphore doit être initialisé à zéro. Si `main()` atteint `sem_wait()` avant que `t2()` atteigne `sem_post()`, il décrémente la valeur du sémaphore à `-1` et se met en sommeil. Lorsque `t2()` est ordonnancé et atteint `sem_post()` il remet le sémaphore à `0` et réveille `main()`. Les deux threads continuent ensuite leur vie [4].

L'exécution de ce code afficherait

```
J'aimerais bien attendre...
Eh bien, tu vas attendre.
Merci de m'avoir fait attendre.
```

1.4 Le problème producteurs/consommateurs revisité

Comme nous venons de le voir, on peut exprimer la signalisation avec les sémaphores et pas seulement avec les variables de conditions. Bien que l'effet soit le même le raisonnement est différent.

Dans cette section, nous allons revisiter le problème producteurs/consommateurs avec les sémaphores.

Rappel 1

On a des threads qui écrivent dans un buffer d'une taille certaine, à l'aide d'une fonction `put()` et d'autres qui lisent depuis le buffer à l'aide d'une fonction `get()`. Il faut bien synchroniser les threads afin que le buffer ne soit pas vide quand on essaie de lire et à l'inverse qu'on essaie pas d'écrire dans un buffer plein.

Pour simplifier, le buffer est un tableau d'entiers et on essaie de lire et d'écrire un entier à la fois. Les fonctions `put()` et `get()` sont comme ci-dessous

```
#define MAX 1
int buffer[MAX]; // MAX est déclaré avant

int fill = 0;
int use  = 0;
```

4. Le cas où `t2()` atteint `sem_post()` en premier est laissé à faire en exercice au lecteur. Le premier `merge request` gagne 0.1 points sur l'examen.

```
void put(int val) {
    buffer[fill] = val;
    fill = (fill + 1) % MAX;
}

int get() {
    int tmp = buffer[use];
    use = (use + 1) % MAX;
    return tmp;
}
```

1.4.1 Tentative ratée

En nous inspirant de ce que nous avons fait avec les variables de conditions, nous allons utiliser deux sémaphores: `empty` et `full`. Le code pourrait ressembler au code ci-dessous.

```
sem_t empty, full;

void *producer(void *arg) {
    for (int i = 0; i < loops; ++i) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
    return NULL;
}

void *consumer(void *arg) {
    int tmp = 0;
    while (tmp != -1) { // condition d'arrêt
        sem_wait(&full);
        tmp = get();
        sem_post(&empty);
        printf("%d\n", tmp);
    }
}

int main() {
    sem_init(&empty, 0, MAX); // MAX buffers sont vides
    sem_init(&full,  0, 0); // 0 buffers sont pleins
}
```

Imaginons que `MAX=1` et qu'il y a deux threads T_1 et T_2 sur un seul CPU. Dans notre scénario T_1 est le producteur et T_2 le consommateur. Si T_1 est ordonnancé en premier, il va appeler `sem_wait(&full)` en premier. Comme `full` est initialisé à `0`, il sera décrémenté à `-1` et sera mis dans l'état *bloqué*. A présent T_2 est ordonnancé comme il est seul, et va entrer dans sa boucle, décrémenter le sémaphore `empty` à `0` (on l'a initialisé à `MAX=1` rappelez-vous),

appeler `put(i)` , puis appeler `sem_post(&full)` qui va incrémenter la valeur du sémaphore `full` à `0` et réveiller T_1 (le mettre dans l'état *prêt*). Ensuite il peut se passer deux choses:

1. T_2 continue son exécution et appeler `sem_wait(&empty)`. Il verra alors que la valeur de `empty` est nulle et sera bloqué.
2. Si T_2 est interrompu et que T_1 est ordonnancé, il consommera la valeur dans le buffer, incrémentera `empty` à `1`.

Dans les deux cas, le fonctionnement est bien celui qu'on souhaite. En fait même avec plus de deux threads cela fonctionne (même si c'est un peu compliqué à voir...).

Question 3

Que se passe-t-il à présent si `MAX>1`?

Dans le cas où `MAX>1`, nous avons plusieurs producteurs et plusieurs consommateurs. Disons qu'il y a deux producteurs P_1 et P_2 qui arrivent plus ou moins en même temps à l'appel de `put()`. Si P_1 passe en premier dans la fonction `put()` et remplit `buffer[0]` avec la première valeur, si ensuite avant d'incrémenter `fill` P_1 est interrompu et P_2 remplit également `buffer[0]`. Nous avons un **accès concurrent**!!! On perd des données des producteurs et notre modèle ne fonctionne pas.

1.4.2 Tentative ratée bis

Dans cette implémentation on a pas d'exclusion mutuelle: quand on écrit/lit dans le buffer, rien ne nous garantit qu'on ne est le · la seul · e à le faire. On va utiliser le sémaphore binaire (le "mutex-sémaphore") de la section précédente autour de `get()` et `put()` et voir comment on s'en sort.

```
sem_t empty, full, mutex;

void *producer(void *arg) {
    for (int i = 0; i < loops; ++i) {
        sem_wait(&mutex);
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
    return NULL;
}

void *consumer(void *arg) {
    int tmp = 0;
    while (tmp != -1) { // condition d'arrêt
        sem_wait(&mutex);
```

```
        sem_wait(&full);
        tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}

int main() {
    sem_init(&empty, 0, MAX); // MAX buffers sont vides
    sem_init(&full,  0, 0); // 0 buffers sont pleins
    sem_init(&mutex,  0, 1); // It's a lock!
}
```

Question 4

Est-ce que ça marche?

Ici, on pourrait se croire soti · e · s d'affaire. En fait non... Il y a de très grandes chances qu'on ait un interblocage. Reprenons nos deux threads T_1 (producteur) et T_2 (consommateur). T_2 est ordonnancé en premier, il acquière le `mutex`, appelle `sem_wait(&full)` mais comme il n'y a rien à lire dans le buffer il se met en état bloqué et attend que T_1 le réveille. Sauf que T_1 ne pourra jamais le réveiller, car il ne pourra jamais acquérir le verrou: **deadlock**! En quelques mots: T_1 détient le verrou et attend un signal de T_2 qui pourrait envoyer le signal mais il attend la libération du verrou.

1.4.3 Tentative réussie

Après tant de déceptions, il est temps de faire le code qui marche.

Question 5

La solution est pourtant simple. Saurez-vous la trouver?

En fait pour résoudre le problème il suffit d'inverser l'ordre signal-verrou. Le code suivant fonctionne.

```
sem_t empty, full, mutex;

void *producer(void *arg) {
    for (int i = 0; i < loops; ++i) {
        sem_wait(&empty);
        sem_wait(&mutex);
        put(i);
        sem_post(&mutex);
        sem_post(&full);
```

```
    }
    return NULL;
}

void *consumer(void *arg) {
    int tmp = 0;
    while (tmp != -1) { // condition d'arrêt
        sem_wait(&full);
        sem_wait(&mutex);
        tmp = get();
        sem_post(&mutex);
        sem_post(&empty);
        printf("%d\n", tmp);
    }
}

int main() {
    sem_init(&empty, 0, MAX); // MAX buffers sont vides
    sem_init(&full,  0, 0); // 0 buffers sont pleins
    sem_init(&mutex,  0, 1); // It's a lock!
}
```

1.5 Les verrous lecteurs/rédacteurs

Pour certaines classes de problèmes, nous avons besoin de verrous qui soient plus flexibles. En particulier, différentes classes de structures de données peuvent nécessiter différentes sortes de verrous. Par exemple, imaginons un certain nombre d'opérations à effectuer sur des listes: l'insertion et la recherche. Alors que l'insertion change l'état de la liste, la recherche ne fait que lire dans la structure. Ainsi, aussi longtemps que nous pouvons garantir qu'aucune insertion ne se fait, on peut autoriser autant de recherches concurrentes qu'on le souhaite. Nous allons donc écrire ici un verrou **lecteurs/rédacteur** basé sur les sémaphores.

Nous voulons donc créer un verrou, de type `rwlock_t`, qui permettra à autant de lecteurs que nous le voulons de lire des données, mais pas pendant qu'un rédacteur est en train d'écrire. Nous devrons donc verrouiller/déverrouiller de deux façons différentes. Une fois nous verrouillerons en écriture: seul le thread rédacteur peut entrer dans la section critique. L'autre fois, nous verrouillerons en écriture: autant de threads lecteurs qu'on le souhaite pourront entrer dans la section critique, qui ne sera protégée que du/des threads rédacteurs. Nous aurons donc besoin de deux classes de verrouillage/déverrouillage: un `acquire_/release_readlock` et un `acquire_/release_writelock`. Pour le verrou lecteur, il faut l'acquérir lorsque le premier lecteur entre dans la section critique. On le symbolise par un sémaphore `writelock`. Il ne sera libéré qu'au moment où le dernier lecteur aura fini sa lecture. Il faut donc avoir un moyen de garder la trace du nombre de lecteurs simultanés: on introduit une variable entière `num_readers` dans ce but. Finalement, cette variable doit être également protégée par un second verrou (nommé `lock`), pour éviter sa modification par plusieurs lecteurs simultanément. La partie du verrouillage pour le rédacteur est plus simple: il suffit de verrouiller/déverrouiller le sémaphore binaire `writelock`. Le code ci-dessous

effectue cette implémentation.

```
typedef struct __rwlock_t {
    int num_readers;
    sem_t lock;
    sem_t writelock;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->num_readers = 0;
    sem_init(&lock, 0, 1);
    sem_init(&writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->num_readers += 1;
    if (rw->num_readers == 1) { // le premier lecteur acquière le verrou
        sem_wait(&rw->writelock); // le sémaphore passe à 0,
                                  // s'il est libre ou à < 0 s'il l'est pas
    }
    sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->num_readers -= 1;
    if (rw->num_readers == 0) { // le dernier lecteur libère le verrou
        sem_post(&rw->writelock); // le sémaphore passe à 1,
                                  // et réveille un éventuel rédacteur
    }
    sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
    sem_wait(&rw->writelock); // verrouillage classique
}

void rwlock_release_writelock(rwlock_t *rw) {
    sem_post(&rw->writelock); // déverrouillage classique
}
```

Cette implémentation fonctionne. On voit par contre qu'il peut y avoir un problème d'équité. En effet, il est assez facile de voir que les threads rédacteurs peuvent souffrir de la famine à cause des threads lecteurs. Pour éviter ce problème il faudrait imaginer un système qui empêche des nouveaux threads lecteurs d'entrer dans le verrou quand un thread rédacteur attend.

Exercice 1

Réaliser un verrou lecteurs/rédacteurs plus équitable. [5]

```
typedef struct __rwlock_t {
    int num_readers;
    sem_t lock;
    sem_t writelock;
    sem_t waitforwrite;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->num_readers = 0;
    sem_init(&lock, 0, 1);
    sem_init(&writelock, 0, 1);
    sem_init(&waitforwrite, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
    sem_wait(&rw->waitforwrite); // Un seul lecteur à la fois. Vérouillé
                                 // tant qu'un rédacteur attend
    sem_post(&rw->waitforwrite);
    sem_wait(&rw->lock);
    rw->num_readers += 1;
    if (rw->num_readers == 1) { // le premier lecteur acquière le verrou
        sem_wait(&rw->writelock); // le sémaphore passe à 0,
                                  // s'il est libre ou à < 0 s'il l'est pas
    }
    sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->num_readers -= 1;
    if (rw->num_readers == 0) { // le dernier lecteur libère le verrou
        sem_post(&rw->writelock); // le sémaphore passe à 1,
                                  // et réveille un éventuel rédacteur
    }
    sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
    sem_wait(&rw->waitforwrite); // bloque les lecteurs si sur le point d'écrire
    sem_wait(&rw->writelock); // verrouillage classique
}

void rwlock_release_writelock(rwlock_t *rw) {
    sem_post(&rw->writelock); // déverrouillage classique
```

5. Cet exercice est laissé à faire en exercice au lecteur. Le premier `merge request` gagne 0.1 points sur l'examen.

```
    sem_post(&rw->waitforwrite); // libere les lecteurs apres écriture
}
```

1.6 Le dîner des philosophes

Le problème du *dîner des philosophes* a été proposé et résolu par E. Dijkstra. C'est un problème classique de concurrence qui concerne le partage de ressources informatiques: l'ordonnancement et leur allocation.

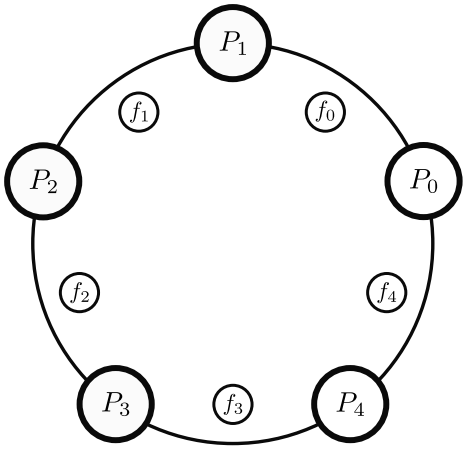

FIGURE 1 – Le dîner des philosophes, Inkscape sur fond blanc, O. Malaspinas, Musée du Louvre, 2019. Cinq philosophes, cinq fourchettes.

Considérons cinq philosophes, qui ont chacun à leur droite une fourchette (cela fait donc cinq fourchettes au total également, voir fig. 1). Ils ont devant eux une assiette avec de la nourriture (par exemple un magnifique et juteux filet de bœuf parfaitement cuit, donc bleu [6]). Les philosophes peuvent faire deux choses: penser et manger. Ils peuvent penser pendant un temps indéfini et n'ont besoin d'aucun matériel pour le faire. En revanche, pour manger chaque philosophe a besoin de deux fourchettes, celle qui se trouve à sa gauche et celle qui se trouve à sa droite, mais chaque fourchette ne peut être tenue que par un seul philosophe à la fois. Il faut donc que les philosophes reposent leurs fourchettes dès qu'ils ont fini de manger. Ils peuvent également se saisir des fourchettes sans que les deux soient libres, mais doivent impérativement avoir les deux en main pour manger.

Le problème est de réussir à faire en sorte que les philosophes ne meurent pas de faim, car chacun va manger ou penser, mais aucun des philosophes ne sait quand les autres vont faire.

Il est plus difficile que prévu d'écrire un algorithme robuste à ce problème. Par exemple, l'algorithme suivant ne fonctionnerait pas. Chaque philosophe (qui représentent en fait des threads) fera ces actions

— Penser jusqu'à ce que la fourchette de droite soit libre, quand elle l'est, la ramasser.

6. Comme vous pouvez le remarquer ils ont que des fourchettes pour manger du filet de bœuf c'est pas super réaliste, mais c'est pas moi qui ai posé le problème...

— Penser jusqu'à ce que la fourchette de gauche soir libre, quand elle l'est la ramasser.
— Manger un certain temps lorsque les deux fourchettes sont en ma possession.
— Poser la fourchette de gauche.
— Poser la fourchette de droite.
— Recommencer dès le début.

On voit assez facilement, que cet algorithme peut produire un interblocage. En effet, si par malheur les cinq philosophes décident de ramasser la fourchette de droite en même temps, ils vont se retrouver bloqués, sans moyen de se débloquer. Un autre problème qui pourrait se produire est qu'un philosophe meure de faim, car il n'arrive pas à se saisir des deux fourchettes, pendant qu'un ou plusieurs autres se gavent de nourriture.

Le problème principal ici, c'est que nous n'utilisons aucune primitive d'exclusion mutuelle permettant de synchroniser l'accès aux ressources. Voyons à comment en pratique résoudre ce problème.

Chaque philosophe va effectuer en boucle la séquence d'opérations suivantes (code en pseudo-c).

```
while(1) {
    think();
    get_forks();
    eat();
    put_forks();
}
```

Alors que `tink()` et `eat()`, sont deux opérations qui peuvent d'effectuer sans synchronisation, les fonctions `get_forks()` et `put_forks()` doivent être écrites avec plus de soin pour éviter famine, interblocages, et tenter d'avoir un maximum de philosophes en train de manger (parce que manger c'est bon).

Avant d'aller plus loin, on introduit deux fonctions qui nous seront utiles.

```
int get_left(int p) {
    return (p + 4) % 5;
}

int get_right(int p) {
    return p;
}
```

Alors que la fourchette de droite de chaque philosophe a le même indice que lui, celle de gauche est obtenue grâce à cette opération modulo. On a par exemple que P_0 a à sa droite f_0 et à sa gauche $f_{(0+4)\%5} = f_4$ comme prévu (ouf).

Nous allons également utiliser cinq sémaphores pour synchroniser l'accès aux fourchettes.

```
sem_t forks[5];
```

1.6.1 Une solution ratée

Comme vous commencez à en avoir l'habitude, on commence par écrire une solution qui ne marche pas. On commence à initialiser toues les sémaphores à un.

```
void get_forks(void *arg) {
    int p = *(int *)arg;
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
}

void put_forks(void *arg) {
    int p = *(int *)arg;
    sem_post(&forks[left(p)]);
    sem_post(&forks[right(p)]);
}
```

Ici, on a simplement écrit l'algorithme décrit un peu plus haut. Chaque philosphe commence par attendre que sa fourchette de gauche soit libre, puis que celle de droite soit libre avant de pouvoir manger. Puis, lorsqu'il a fini, il repose la fourchette de gauche, puis celle de droite et se remet à penser.

Exactement comme on l'a discuté tout à l'heure, cette solution peut donner lieu à un *deadlock*. Si chaque philosophe prend en même temps sa fourchette de gauche, tous les philosophes attendront sur leur fourchette de droite et seront bloqués.

1.6.2 Une solution possible

La façon la plus simple de résoudre ce problème est de modifier l'ordre d'acquisition d'un des philosophes. Si P_0 acquière d'abord la fourchette de droite au lieu de celle de gauche il n'y a pas moyen que tous les philosophes se retrouvent en attente.

```
void get_forks(void *arg) {
    int p = *(int *)arg;
    if (p == 0) {
        sem_wait(&forks[right(p)]);
        sem_wait(&forks[left(p)]);
    } else {
        sem_wait(&forks[left(p)]);
        sem_wait(&forks[right(p)]);
    }
}
```

1.6.3 Le mot de la faim

Il existe toute une classe de problèmes similaires pour réfléchir sur les problèmes de concurrence. En particulier, citons le problème du **barbier assoupi**, ou des **fumeurs de cigarettes**[7].

7. Fumer nuit gravement à la santé.

1.7 L'implémentation des sémaphores

Histoire de comprendre un peu mieux le fontionnement des sémaphores, essayons d'en cronstruire un à partir des primitives de synchronisation que nous avons à disposition: les verrous et les variables de condition. Nous allons appeler ces sémaphores maison les *séphamores*.

En plus de la variable de condition et du verrou, il faut utiliser un entier qui stocke la valeur du séphamore.

```
typedef struct __seph_t {
    int value;
    phtread_cond_t cond;
    pthread_mutex_t mutex;
} seph_t;
```

Ensuite, il faut implémenter trois fonctions: l'intialisation, `init()`, la mise en attente `wait()`, et le réveil `post()`. Ces trois fonctions peuvent être implémentées de la façon suivante.

```
// Un seul thread appelle ça.
void seph_init(seph_t &s, int value) {
    s->value = value;
    pthread_cond_init(&s->cond, NULL);
    pthread_mutex_init(&s->mutex, NULL);
}

// Un seul thread appelle ça.
void seph_destroy(seph_t &s, int value) {
    s->value = value;
    pthread_cond_destroy(&s->cond);
    pthread_mutex_destroy(&s->mutex);
}

void seph_wait(seph_t &s) {
    pthread_mutex_lock(&s->mutex);
    while (s->value <= 0) {
        pthread_cond_wait(&s->cond);
    }
    s->value -= 1;
    pthread_mutex_unlock(&s->mutex);
}

void seph_post(seph_t &s) {
    pthread_mutex_lock(&s->mutex);
    s->value += 1;
    pthread_cond_signal(&s->cond);
    pthread_mutex_unlock(&s->mutex);
}
```

Question 6

Cette façon d'implémenter les sémaphores ne correspond pas tout à fait à ce que nous avons vu au début de ce chapitre. Pouvez-vous dire quelle est la différence?

En fait, on constate que le sémaphore tel que nous l'avons implémenté ici n'a pas d'équivalence entre le nombre de threads en attente et sa valeur. En effet, ici la valeur du sémaphore ne sera jamais plus petite que zéro.