I Section critique

Définition : Section critique ♡

Une section critique est un bloc de code qui vérifie :
- Exclusion mutuelle : Il ne peut y avoir qu'un seul thread à la fois dans la section critique.
- Absence de famine : Un thread ne doit pas attendre indéfiniment pour entrer dans la section critique, en supposant qu'aucun thread ne reste indéfiniment ou déclenche une erreur dans la section critique.

Remarques :

- Une section critique permet d'éviter qu'une même ressource (variable, fichier, etc.) soit modifiée par plusieurs threads en même temps.
- Il est préférable d'utiliser le moins possible de sections critiques, car elles limitent la parallélisation du programme.

II Mutex

Définition : Mutex ♡

Un mutex (*mutual exclusion*) ou verrou est un objet ayant trois opérations :
- `create` : création du mutex.
- `lock` : verrouillage du mutex.
- `unlock` : déverrouillage du mutex.

Tel que :
- Au plus un thread peut verrouiller le mutex à la fois.
- Le bloc de code entre le verrouillage et le déverrouillage est une section critique.

Il existe déjà des implémentations de mutex en C et OCaml :

```c
int counter = 0;
pthread_mutex_t mutex;

void *increment(void *arg){
    for (int i = 1; i <= 1000000; i++) {
        pthread_mutex_lock(&mutex);
        counter++;
        pthread_mutex_unlock(&mutex);
    }
}
```

Le résultat est toujours 2000000 (avec 2 threads).

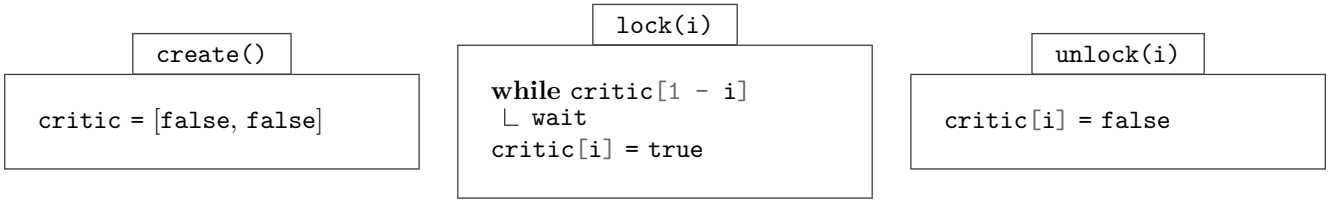
III Implémentation d'un mutex

Pour simplifier, on va implémenter un mutex pour deux threads seulement (alors que les mutex en C ou OCaml fonctionnent avec un nombre arbitraire de threads). On suppose de plus que les threads sont numérotés 0 et 1.

Exercice : Pour chaque tentative d'implémentation suivante, dire si l'exclusion mutuelle et l'absence de famine sont garanties.

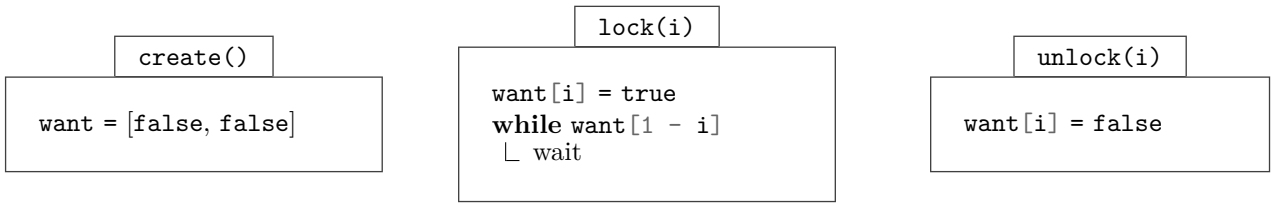
III.1 Tentative d'implémentation 1

On utilise un tableau de booléens `critic` tel que `critic[i]` détermine si le thread `i` est dans la section critique.



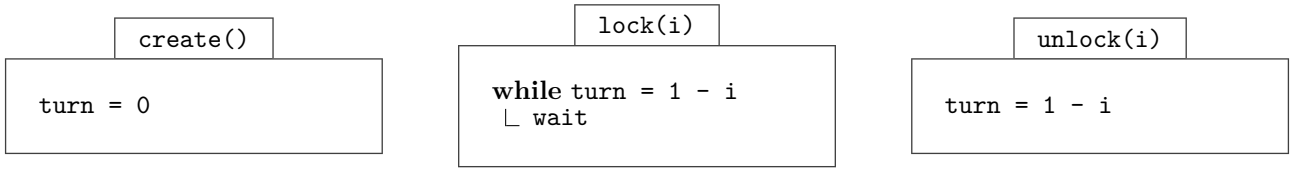
III.2 Tentative d'implémentation 2

On utilise un tableau `want` de booléens tel que `want[i]` détermine si le thread `i` veut entrer dans la section critique.



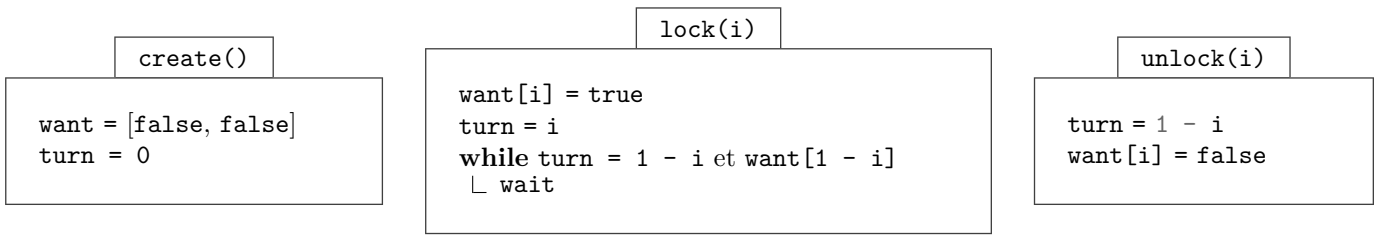
III.3 Tentative d'implémentation 3

On utilise une variable `turn` qui détermine quel thread peut entrer dans la section critique.



III.4 Tentative d'implémentation 4

On combine les deux tentatives précédentes.



III.5 Algorithme de Peterson ♡

```
create()
  want = [false, false]
  turn = 0

lock(i)
  want[i] = true
  turn = 1 - i
  while turn = 1 - i and want[1 - i]
    wait

unlock(i)
  want[i] = false
```

Algorithme de Peterson

Théorème

L'algorithme de Peterson garantit l'absence de famine.

Preuve : Supposons par l'absurde que le thread 0 attende indéfiniment dans `lock`. Alors `turn = 1` et `want[1]` vaut vrai. On distingue les prochaines instructions du thread 1 :

- Si le thread 1 essaie de verrouiller le mutex, alors il écrit 0 dans `turn` et `turn` reste à 0 tant que le thread 0 ne change pas sa valeur. Ainsi, le thread 0 sort du `while`.
- Si le thread 1 a déverrouillé le mutex et n'essaie pas de le verrouiller à nouveau, alors `want[1]` est mis à faux et n'est plus modifié. Le thread 0 peut donc sortir du `while`.
- Si le thread 1 est bloqué dans la boucle `while` alors `turn = 0`, ce qui est absurde par hypothèse.

Remarque : Dans l'algorithme de Peterson, un fil qui essaie de verrouiller le mutex est en état d'attente active (il exécute du code en boucle), ce qui utilise le processeur inutilement. Cet algorithme théorique n'est pas utilisé en pratique.

Théorème

L'algorithme de Peterson garantit l'exclusion mutuelle.

Preuve : Supposons par l'absurde que les deux threads sont simultanément en section critique. Considérons les opérations atomiques suivantes dans le `lock` :

1. thread 0 met `want[0]` à vrai ;
2. thread 0 met `turn` à 1 ;
3. thread 0 lit `turn` ;
4. thread 0 lit `want[1]` ;
5. thread 1 met `want[1]` à vrai ;
6. thread 1 met `turn` à 0 ;
7. thread 1 lit `turn` ;
8. thread 1 lit `want[0]`.

Supposons que 2 soit exécuté avant 6, noté $2 \prec 6$. Comme $6 \prec 7$ et que le thread 1 est entré en section critique, le thread 1 a lu `turn = 0` donc a évalué `want[0]` à faux. D'où $2 \prec 6 \prec 7 \prec 8 \prec 1 \prec 2$: absurde.

Exercice 1.
L'algorithme de Peterson reste t-il correct si on échange `want[i] = true` et `turn = 1 - i` dans `lock` ?

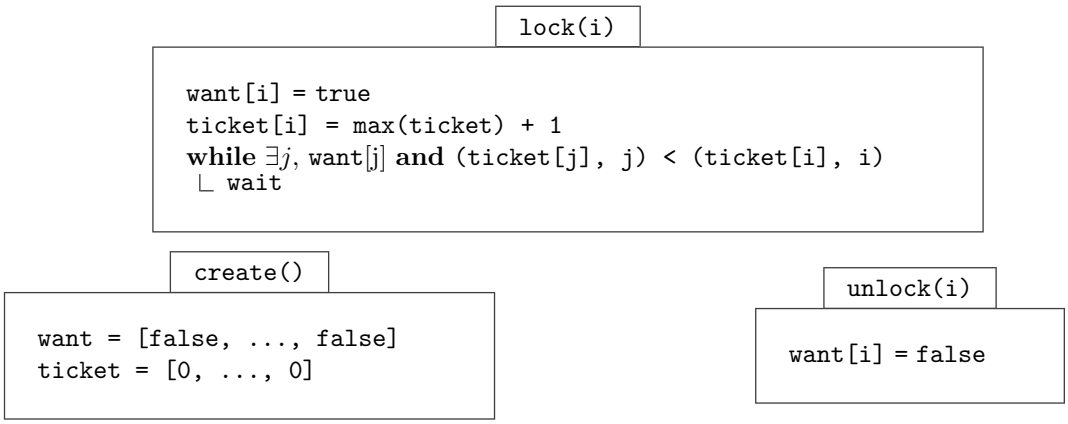
IV Algorithme de la boulangerie de Lamport ♡

L'algorithme de Peterson ne fonctionne que pour deux threads.

L'algorithme de la boulangerie de Lamport permet d'implémenter un mutex pour des threads numérotés $0, ..., n-1$, en utilisant deux tableaux `want` et `ticket` :

- `want[i]` détermine si le thread i veut entrer dans la section critique.
- `ticket[i]` est la priorité du thread i : si `ticket[i] < ticket[j]` alors le thread i est prioritaire sur le thread j.

Idée : on fait rentrer le thread dont le ticket est le plus petit. On départage les égalités avec le numéro du thread, en utilisant l'ordre lexicographique sur les couples.



Algorithme de la boulangerie de Lamport

Théorème

Pour tout thread `i`, `ticket[i]` est croissant au cours du temps.

Preuve : `ticket[i]` n'est jamais remis à 0 et son ancienne valeur est prise en compte dans le calcul du `max`.

Théorème

L'algorithme de la boulangerie de Lamport garantit l'absence de famine.

Preuve : Supposons qu'un thread `i` souhaite entrer dans la section critique. Comme le numéro de ticket des autres threads ne peut qu'augmenter strictement, le thread `i` finira par être le thread prioritaire et pourra entrer dans la section critique.

Théorème

L'algorithme de la boulangerie de Lamport garantit l'exclusion mutuelle.

Preuve : Supposons par l'absurde que deux threads `i` et `j` soient simultanément en section critique et que `(ticket[i], i) < (ticket[j], j)`. Il y a deux possibilités lorsque le thread `j` est entré en section critique :

- Il a lu `(ticket[j], j) < (ticket[i], i)`, ce qui est impossible par hypothèse (en utilisant le fait que `ticket[i]` est croissant).
- Il a lu faux dans `want[i]`. Alors le thread `j` était dans le `while` quand le thread `i` a commencé l'appel à `lock`. Mais alors `ticket[i] > ticket[i]`, ce qui est absurde.

V Sémaphore et application au problème producteurs-consommateurs

Définition : Sémaphore ♡

Un sémaphore est un compteur avec trois opérations :
- `init` : initialise le compteur à une valeur donnée.
- `wait` (ou `P`) : attend que le compteur devienne > 0 puis décrémente le compteur.
- `post` (ou `V`) : incrémente le compteur et s'il y au moins un fil en attente, en réveille un.

Remarques :

- Le compteur compte le nombre de ressources disponibles.

- Un sémaphore binaire (dont le compteur ne peut prendre que les valeurs 0 et 1) est un mutex.

Dans le problème producteurs-consommateurs, une file (*buffer*) est partagée pour permettre aux threads de se synchroniser. Les producteurs ajoutent des éléments dans la file et les consommateurs les retirent, avec les contraintes suivantes :

- La file est de taille n.
- L'accès à la file doit être une section critique.
- Si un producteur veut ajouter un élément dans une file pleine, il est mis en attente.
- Si un consommateur veut retirer un élément d'une file vide, il est mis en attente.

Par exemple, plusieurs threads (producteurs) qui veulent écrire dans un même fichier peuvent ajouter leurs modifications dans une file. Le système (consommateur) écrit alors périodiquement les modifications dans le fichier.

On peut résoudre le problème producteurs-consommateurs avec deux sémaphores et un mutex :

- Un sémaphore `empty`, initialisé à n, qui compte le nombre de places libres dans la file.
- Un sémaphore `full`, initialisé à 0, qui compte le nombre de places occupées dans la file.
- Un mutex pour protéger l'accès à la file.

```c
#include <pthread.h>
#include <semaphore.h>

pthread_mutex_t mutex;
sem_t empty, full;

void *producer(void *arg){
    while (1) {
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        // Ajouter un élément dans la file
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
    return NULL;
}
void *consumer(void *arg){
    while (1) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        // Retirer un élément de la file
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
    return NULL;
}
int main(){
    pthread_t prod, cons;
    pthread_mutex_init(&mutex, NULL);
    sem_init(&empty, 0, n);
    sem_init(&full, 0, 0);
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);
    return 0;
}
```

```ocaml
module Sem = Semaphore.Counting

type 'a buffer = {
    q : 'a Queue.t;
    empty : Sem.t;
    full : Sem.t;
    mutex : Mutex.t;
}
let create_buffer () = {
    q = Queue.create ();
    empty = Sem.create n;
    full = Sem.create 0;
    mutex = Mutex.create ();
}
let produce buf item =
    Sem.wait buf.empty;
    Mutex.lock buf.mutex;
    Queue.push item buf.q;
    Mutex.unlock buf.mutex;
    Sem.post buf.full

let consume buf =
    Sem.wait buf.full;
    Mutex.lock buf.mutex;
    let item = Queue.pop buf.q in
    Mutex.unlock buf.mutex;
    Sem.post buf.empty
```