UGA - IUT2 - Département informatique
BUT Semestre 6 - Programmation Concurrente - 2024/2025

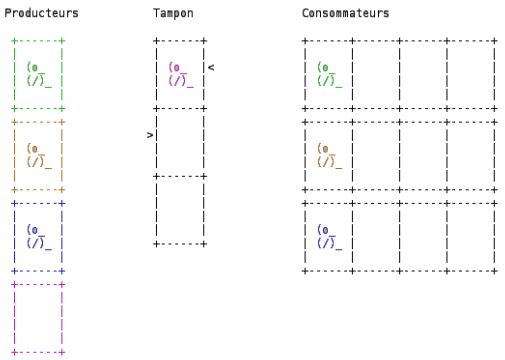
TP03 et TP04 : Moniteurs

1. Section critique à N places

Dans ce premier exercice, placez-vous dans le répertoire `tp03/section_critique/`. Vous pouvez constater en compilant le code sans modifications que la partie de section critique n'est pas assurée, et que le nombre de threads entrant peut-être bien plus grand que le nombre de places disponibles (testez par exemple avec `bin-gcc/section_critique 5 2`). Vous devez dans cet exercice éditer les fichiers pour reproduire la section critique à plusieurs places vue en cours.

Après vos modifications, expérimentez avec plusieurs paramètres pour bien constater que le nombre de threads passant en même temps est toujours au plus le nombre de places disponibles. Par exemple, refaites le test avec 5 threads et 2 places comme avant les modifications. Pour simuler le fonctionnement classique d'un mutex, vous pouvez également tester votre code avec une seule place disponible. Vous constaterez alors à l'exécution que les threads rentrent et sortent un par un de la section critique.

2. Producteurs Consommateurs



Le modèle des producteurs consommateurs a été présenté en cours. Il est constitué d'un tampon au nombre de cases limité, d'une famille de producteurs qui déposent des éléments dans les cases du tampon, et d'une famille de consommateurs qui récupèrent ces éléments. Le modèle de synchronisation doit respecter les contraintes suivantes :

- chaque case ne peut contenir qu'un élément;
- l'accès au tampon doit se faire en exclusion mutuelle;
- le dépôt par un producteur doit être bloqué si le tampon est plein;
- le retrait par un consommateur doit être bloqué si le tampon est vide.

Dans cet exercice, les éléments produits et consommés sont des manchots Tux dont la couleur dépend du numéro du producteur qui les a créés, et correspondent à des instances de la classe **`Element`**. Le tampon est une instance de la classe **`Tampon`**.

Pour pouvoir faire deux versions de cet exercice, la classe **`Tampon`** est en pratique une classe abstraite se dérivant en deux classes d'implémentation :

- **`TamponCond`** : dans cette première version, le tampon est réalisé avec un conteneur de la classe **`std::deque`** et un entier précisant sa taille limite; en revanche le modèle de synchronisation doit être réalisé avec un **`std::mutex`** et deux **`std::condition_variable`**.
- **`TamponSBQ`** : dans cette seconde version, le tampon est réalisé avec un conteneur de la classe **`boost::sync_bounded_queue`** qui inclut par construction le modèle de synchronisation souhaité.

Il s'agit, dans cet exercice, de compléter la définition de ces deux classes, en ajoutant en particulier les outils nécessaires à la bonne synchronisation des threads.

2.1. Variable de condition

Nous vous rappelons la démarche que nous vous conseillons de suivre pour réaliser une synchronisation avec des variables de condition.

- Repérer dans le programme où un thread doit attendre et écrire la condition d'attente avec des variables partagées : ici les propriétés de la **deque** sont suffisantes pour écrire les conditions d'attente.
- Réaliser cette attente sous la forme suivante :

```
while (condition_d_attente)
  var_cond.wait(verrou);
```

  Nous rappelons que cette boucle d'attente doit être incluse dans une section critique établie grâce à la même variable **verrou**, instance de **unique_lock<mutex>**. Cette instance est construite avec un **mutex** qui doit être utilisé pour mettre en section critique toute manipulation des variables utilisées dans la condition d'attente : ici le **mutex** est un attribut de **TamponCond** devant sécuriser toute manipulation de la **deque**.
- Repérer quand un thread peut libérer un de ceux qui sont en attente, et écrire l'envoi du signal de réveil sans condition :

```
var_cond.notify_one();
```

  S'il convient de libérer tous les threads en attente, exécuter plutôt :

```
var_cond.notify_all();
```

Instructions et Questions :

1. Se positionner dans le répertoire `tp03/prod-cons/` et lire les différents fichiers pour comprendre la structure du programme et des classes fournies.
2. Compléter les fichers `TamponCond.hpp` et `TamponCond.cpp` pour réaliser la synchronisation décrite ci-dessus. En particulier, vous ajouterez aux attributs de la classe **TamponCond** un **mutex** et deux **condition_variable**.
3. Compiler et tester votre programme en variant le nombre des producteurs et des consommateurs. Par exemple :
   - `bin-gcc/prod_cons cond 5 1 1 5 1`
   - `bin-gcc/prod_cons cond 1 5 5 1 1`
   - `bin-gcc/prod_cons cond 5 1 1 5 3`
   - `bin-gcc/prod_cons cond 1 5 5 1 3`
   - `bin-gcc/prod_cons cond 4 3 3 4 7`
   - `bin-gcc/prod_cons cond 4 2 2 1 7`
   - `bin-gcc/prod_cons cond 2 1 4 2 7`

   Remarque : pour terminer le programme, il faut taper sur une touche du clavier.

2.2. Boost : `Synchronized Bounded Queue`

Boost propose des outils dits synchronisés, c'est-à-dire incluant des mécanismes de synchronisation. Parmi ces outils figure la **synchronized bounded queue** qui réalise exactement le modèle de synchronisation des producteurs / consommateurs. Voici un extrait de la définition de cette classe disponible dans le fichier `/usr/include/boost/thread/sync_bounded_queue.hpp`.

Extrait de la classe `sync_bounded_queue`

```
template <typename ValueType>
class sync_bounded_queue
{
public:
  explicit sync_bounded_queue(std::size_t max_elems);
  void push_back(const ValueType& x);
  ValueType pull_front();
}
```

Attention! Cet outil est encore expérimental et peut être modifié dans les versions futures de boost. En particulier, il n'est pas garanti exempt d'éventuels *bugs*.

Instructions et Questions :

1. Compléter les fichers `TamponSBQ.hpp` et `TamponSBQ.cpp` pour réaliser la synchronisation des producteurs / consommateurs. En particulier, vous ajouterez aux attributs de la classe **TamponSBQ** une **sync_bounded_queue**. Aucun **mutex** ni **condition_variable** n'est nécessaire : ces outils sont déjà inclus dans la *synchronized bounded queue* et sont manipulés par les méthodes **push_back()** et **pull_front()** sans que l'utilisateur n'ait à s'en occuper. En revanche, ne pas oublier d'inclure dans les fichiers le *header* contenant la définition de cette classe, par la commande **#include <boost/thread/sync_bounded_queue.hpp>**.
2. Compiler et tester votre programme en variant le nombre des producteurs et des consommateurs comme pour l'exercice précédent.

3. Lecteurs Rédacteurs

Dans ce modèle, la ressource partagée peut être assimilée à un fichier. Deux familles de processus peuvent accéder au fichier :

- **lecteurs** : plusieurs lecteurs peuvent accéder en même temps au fichier
- **rédacteurs** : un seul rédacteur à la fois peut accéder au fichier

Par ailleurs

- un rédacteur ne peut pas accéder au fichier tant que des lectures sont en cours
- un lecteur ne peut pas accéder au fichier tant qu'une écriture est en cours

Il y a donc exclusion mutuelle entre un rédacteur et tout groupe de lecteurs.

Par analogie avec les exercices précédents, il est possible de maintenir en permanence deux compteurs, l'un pour les rédacteurs en train d'écrire dans le fichier, et un associé aux lecteurs en train de lire dans le fichier. Il s'agira alors de vérifier à tout moment :

- Soit le nombre de rédacteurs est nul, soit le nombre de lecteurs est nul (ou les deux sont nuls).
- Il y a au plus un rédacteur en cours d'écriture.

Le fonctionnement d'un tel modèle peut être diversifié par des priorités qu'on peut donner à telle ou telle population de processus. Dans le cadre de cet exercice, nous proposons de mettre en œuvre deux types de priorités grâce à deux outils de synchronisation différents.

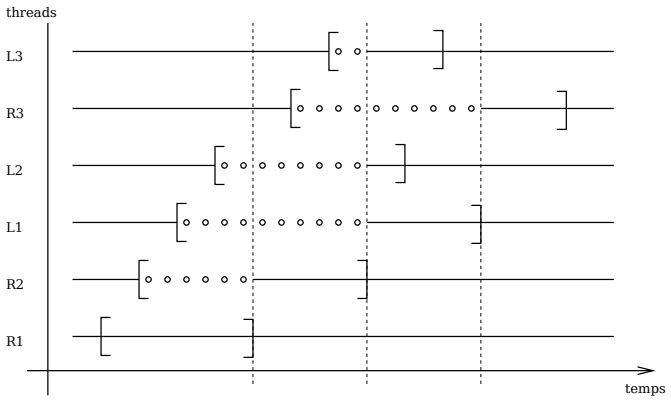
3.1. Priorité simple aux Lecteurs avec des variables condition

La **priorité simple aux lecteurs** est définie comme suit : un lecteur est prioritaire sur un rédacteur si une lecture est déjà en cours. Dans ce cas, tout lecteur peut accéder au fichier quel que soit le nombre de rédacteurs en attente.

Questions :

1. Se positionner dans le répertoire `tp03/lec-red/` et lire les différents fichiers pour comprendre la structure du programme et les classes fournies.
2. Compléter les fichiers `OutilsCond.hpp` et `OutilsCond.cpp` pour réaliser une telle modélisation à l'aide de variables de type condition.
3. Les délais fournis ont été choisis pour réaliser le chronogramme qui vous est fourni ci-dessous. Vérifier que les traces qui s'affichent dans le terminal lors de l'exécution de votre programme par `bin-gcc/lec_red cond` sont donc bien cohérentes avec le chronogramme (attention l'implémentation C++ ne garantit pas le FIFO sur la variable condition).

Chronogramme d'une priorité simple aux lecteurs (ici représenté avec une file FIFO)



3.2. Priorité forte aux Rédacteurs avec des variables condition

Dans cette variante, vous pourrez ajouter un troisième compteur pour identifier le cas où des rédacteurs sont en attente. Les lecteurs arrivant après doivent alors être bloqués tant qu'il reste des rédacteurs en attente. Attention, dans cette variante la priorité est forte, c'est à dire que les rédacteurs arrivant après des lecteurs déjà bloqués en attente passeront avant tous ces lecteurs.

Compléter les fichiers `OutilsCondPrio.hpp` et `OutilsCondPrio.cpp` pour réaliser une telle modélisation à l'aide de variables de type condition.

3.3. Priorité égale et FIFO stricte avec des variables condition

Pour remédier aux problèmes précédents, une troisième version peut vous permettre de gérer finement les priorité en forçant l'exécution FIFO des threads demandant un accès au fichier. Pour ce faire, il faudra ajouter une **`deque`** qui contiendra les numéros de **`TID`** de chaque thread en attente (Indice : utiliser `std::thread::id this_id = std::this_thread::get_id();`). Au réveil, chaque thread peut vérifier s'il est en tête de la liste et si c'est le cas il pourra entrer en section critique en se retirant de la liste d'attente. N'oubliez pas que le principe de base de l'exercice est que plusieurs lecteurs peuvent accéder en même temps à la ressource critique. Il sera donc opportun de réveiller plusieurs lecteurs et de permettre à plusieurs lecteurs de dépiler leur numéro de **`TID`** du sommet de la **`deque`** (*attention, l'ordre de réveil n'est pas garanti, il faudra donc procéder à plusieurs réveils collectifs en cascade*).

Compléter les fichiers `OutilsCondFifo.hpp` et `OutilsCondFifo.cpp` pour réaliser une telle modélisation à l'aide de variables de type condition.

3.4. Priorités égales avec des `shared_timed_mutex`

La modélisation des **priorités égales** est définie comme suit : les lecteurs et les rédacteurs ont même priorité, c'est-à-dire premier arrivé, premier servi. En particulier, si une lecture est en cours, un lecteur passera si aucun rédacteur n'est arrivé avant lui; sinon il devra patienter derrière lecteurs et rédacteurs ayant demandé l'accès au fichier avant lui.

La bibliothèque standard C++ propose dans le *header* **`<shared_mutex>`**, un outil qui réalise presque cette priorité : le **`shared_timed_mutex`**. Il s'agit d'un mutex qui propose deux niveaux d'accès :

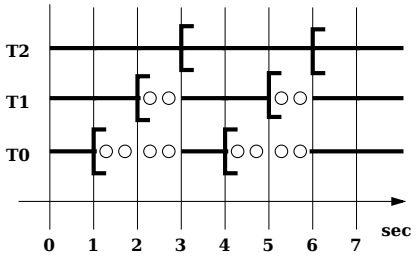
- un niveau exclusif (identique au niveau d'accès habituel des mutex) géré par les méthodes suivantes :
  - **`lock()`** : acquisition exclusive du mutex;
  - **`unlock()`** : libération du mutex.
- un niveau partagé géré par les méthodes suivantes :
  - **`lock_shared()`** : acquisition partagée du mutex;
  - **`unlock_shared()`** : libération du mutex.

Cet outil diffère légèrement du modèle décrit ci-dessus lorsque qu'un thread libère un mutex dont il avait un accès exclusif: tous les threads en attente du mutex sont réveillés et tentent à nouveau d'acquérir le mutex, quel que soit le niveau d'accès demandé. A cette occasion, le strict ordre premier arrivé, premier servi, peut finalement ne pas être respecté.

Questions :

1. Compléter les fichiers `OutilsShared.hpp` et `OutilsShared.cpp` pour réaliser une telle modélisation à l'aide d'un **`shared_timed_mutex`**.
2. Observer les traces qui s'affichent dans le terminal lors de l'exécution de votre programme par `bin-gcc/lec_red shared` et vérifier qu'elles sont bien cohérentes avec les priorités égales.

4. Rendez-vous collectif



Dans ce modèle, on considère **`nb_threads`** tâches qui s'attendent mutuellement à un instant donné de leur exécution en faisant appel à une même méthode parmi les méthodes **`rvReveilleurUnique()`** ou **`rvCascade()`** de la classe **`RendezVous`**. À l'arrivée de la dernière tâche au point de rendez-vous, chacune reprend son exécution indépendamment des autres. Les **`nb_threads`** tâches peuvent se donner **`nb_rendez_vous`** rendez-vous consécutifs. Le chronogramme ci-contre illustre cette situation dans le cas de 3 tâches et 2 rendez-vous (le symbole **[** correspond à l'appel à la méthode de **`RendezVous`**).

Instructions et Questions :

1. Se positionner dans le répertoire `tp03/rendez-vous/` et lire les différents fichiers pour comprendre la structure du programme et les classes fournies.
2. Compléter la définition de la classe `RendezVous` de façon à ce que tous les threads attendant soient réveillés par le dernier thread arrivant au rendez-vous.
3. Vérifier que l'exécution de `bin-gcc/rv_cond 5 2` produit sur la sortie standard une trace semblable à celle donnée dans le fichier `res_rv_cond.txt`.
4. Compléter la méthode **`rvCascade()`** pour que :
   - les **`nb_threads`**-1 premières tâches exécutant **`rvCascade()`** se bloquent;
   - à l'exécution de la même méthode par la tâche suivante, le réveil des tâches s'effectue en cascade : **chaque tâche doit en réveiller une et une seule autre** (sauf la **dernière réveillée** qui n'en réveille aucune).

   Attention : Votre méthode de réveil en cascade doit permettre aux rendez-vous de s'enchainer sans risquer d'avoir un problème quand un thread sortant d'un rendez-vous redemande le rendez-vous suivant avant que tous les threads aient pu sortir du rendez-vous précédent.

   Une solution pour éviter ces problèmes de rendez-vous successif est d'ajouter un compteur **`threads_sortis`** et une variable condition **`sortie_en_cours`** supplémentaires. Tant qu'une sortie est en cours, les threads demandant un nouveau rendez-vous seront bloqués pour attendre la fin du rendez-vous précédent. Les threads sortant se comptent tant que des threads ont encore besoin de sortir. Le dernier à sortir peut alors remettre le compteur à zéro pour préparer la sortie de rendez-vous suivante, et réveiller tous les threads qui attendaient de pouvoir se bloquer sur la barrière du rendez-vous.
5. Vérifier que l'exécution de `bin-gcc/rv_cond 5 2 cascade` produit sur la sortie standard une trace semblable à celle donnée dans le fichier `res_rv_cond.txt`.
6. La solution du second compteur **`threads_sortis`** et de la seconde variable condition **`sortie_en_cours`** peut également être substitué dans le cas du réveilleur unique à la méthode du tableau permettant de compter les threads dans les rendez-vous pairs et impairs, comme vu en cours. Revenez donc sur le code de **`rvReveilleurUnique()`** pour mettre en place cette solution.

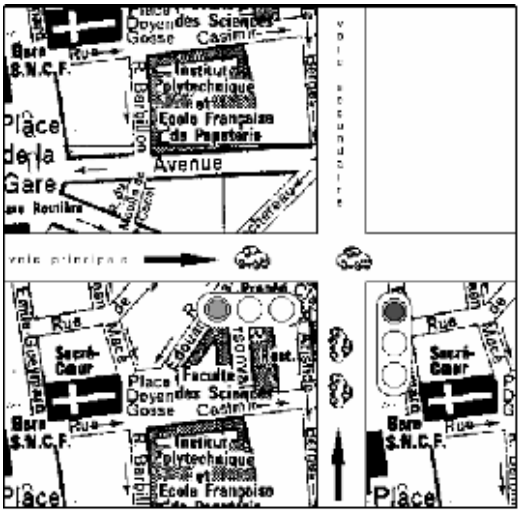
5. Carrefour routier

5.1. Modèle

Le problème du carrefour routier à deux routes a été présenté en cours. Voici un rappel des règles de circulation que l'on veut imposer aux voitures :

1. Une automobile ne peut pas entrer sur le carrefour si **son feu n'est pas à vert** ou si **une automobile est déjà engagée sur le carrefour**;
2. Lorsqu'une voiture quitte le carrefour elle indique que le carrefour est libre et réveille une autre voiture sur la voie ayant le feu vert : cette voiture pourra alors à nouveau tenter de traverser le carrefour;
3. Après avoir changé les couleurs des feux tricolores, la tâche chargée de commuter les feux réveille une voiture qui attendait de rentrer sur le carrefour depuis la voie nouvellement à vert.

5.2. Implémentation



Le programme que vous allez modifier a une sortie texte uniquement, qui peut ensuite être passé par un tube à un second programme qui permet un affichage graphique. Il peut être intéressant de travailler sur la trace textuelle du premier programme plutôt que sur son interprétation, voici donc le sens associé aux messages que vous pouvez lire sur un terminal, et leur relation aux règles de circulation :

- Chaque automobile affiche **`TRAVERSE`** quand il a obtenu l'accès au carrefour : la dernière trace de la commutation concernant sa voie ne doit pas être **`FEU ROUGE`** (idéalement il devrait être **`FEU VERT`**), et toute automobile qui aurait écrit **`TRAVERSE`** pour la même voie doit avoir écrit entre temps **`QUITTE`**.
- La commutation affiche **`FEU ORANGE`** quand elle souhaite commuter les feux mais une automobile peut encore être sur le carrefour.
- Lorsque la commutation affiche **`FEU ROUGE`** ou **`FEU VERT`**, le carrefour doit être vide : la dernière trace écrite par une automobile ne doit pas être **`TRAVERSE`**.

Instructions et Questions :

Se positionner dans le répertoire `tp03/carrefour/` et lire les différents fichiers pour comprendre la structure du programme et les classes fournies. Il s'agit, dans cet exercice et selon les indications données dans les questions suivantes, de compléter les fichiers `Carrefour.hpp` et `Carrefour.cpp` pour réaliser le modèle de synchronisation décrit ci-dessus.

1. Ajouter aux attributs de la classe **`Carrefour`** un **`mutex`** et autant de variables de condition que nécessaire (c'est-à-dire autant que de files d'attentes différentes). Noter que les variables partagées nécessaires à la synchronisation, **`voie_verte`** et **`automobile_engagee`**, sont déjà déclarées.

   *Indications : vous pourrez utiliser les constantes **`nb_voies`**, **`voie_principale`** et **`voie_secondaire`** définies dans `Carrefour.hpp`. Quand vous manipulez plusieurs variables conditions, il est souvent plus simple de les utiliser sous forme de tableau, en particulier dans cet exercice vous disposez de constantes qui valent 0 et 1 qui permettent d'identifier les cases du tableau.*
2. Compléter la méthode **`entrer_dans_carrefour()`** pour que le premier point du modèle de synchronisation décrit ci-dessus soit réalisé. En particulier, le prédicat testé par chaque automobile, avant d'éventuellement se bloquer, utilisera les deux variables partagées citées dans la question précédente.
3. Compléter la méthode **`sortir_du_carrefour()`** pour que le deuxième point du modèle de synchronisation décrit ci-dessus soit réalisé. En particulier, vous ferez attention à manipuler les variables partagées en section critique en utilisant un mutex unique, attribut de la classe **`Carrefour`**.
4. Compléter la méthode **`basculer_sur()`** pour que le dernier point du modèle de synchronisation décrit ci-dessus soit réalisé. En particulier, et comme pour la question précédente, vous ferez attention à manipuler les variables partagées en section critique en utilisant le même mutex. *Indication: utiliser les traces **`FEU ORANGE`** et **`FEU ROUGE`** comme des traces de demande et d'obtention de la section critique par la tâche de commutation*.
5. Dans un premier temps, exécuter votre programme par la commande `bin-gcc/carrefour_main 5 avec_alea` afin de comprendre la succession des affichages. L'argument `avec_alea` permet de construire les threads automobiles avec un temps de construction aléatoire : le programme doit fonctionner dans toute situation. Dans un deuxième temps, rediriger les traces de ce programme vers l'application `carrefour` grâce à la commande `bin-gcc/carrefour_main 5 avec_alea | python carrefour.py`.

6. Vérifier dans le fichier `Auto.cpp` que la durée de traversée est de 2 secondes, compiler, exécuter `bin-gcc/carrefour_main 5 sans_alea`, et analyser la trace qui s'affiche dans le terminal. Vous devriez constater que lorsque plusieurs automobiles attendent au feu rouge, ce n'est pas nécessairement la première qui s'engage quand le feu passe au vert.
    - une première hypothèse pour expliquer cela pourrait être le fait que la liste associée à une variable **`condition_variable`** n'est pas garantie FIFO.
    - nous allons voir que dans la plupart des cas, le caractère FIFO est en fait respecté et qu'une autre raison explique ce mauvais comportement : un signal de réveil est envoyé à des automobiles qui ne peuvent pas s'engager. Elles sortent alors de la tête de la file d'attente pour se repositionner en queue après l'évaluation à faux de leur prédicat.

   Pour vous en convaincre, comprendre le début de la trace d'exécution correspondant à l'évolution de la file d'attente des automobiles de la voie secondaire (tableau ci-dessous). Compléter cette trace (n'indiquer que les événements modifiant la file secondaire).
7. Pour corriger cela, faire en sorte que le thread de commutation n'envoie un signal que s'il est certain que le destinataire peut traverser. Compiler, tester et vérifier que l'ordre FIFO est alors respecté. Vous pouvez alors exécuter votre programme avec l'interface graphique `bin-gcc/carrefour_main 5 sans_alea | python carrefour.py`.
8. Compléter la classe **`Carrefour`** pour que le feu orange ait la signification suivante : la tâche de commutation des feux souhaite commuter les feux, mais si une automobile occupe le carrefour, la tâche de commutation doit attendre que le carrefour soit libéré pour réellement passer au rouge. Pour cela il vous faudra ajouter de nouveaux attributs à la classe **`Carrefour`**. Procéder comme d'habitude :
    - identifier l'endroit où la tâche de commutation doit attendre;
    - définir le prédicat d'attente et réaliser celle-ci dans une boucle **`while`**;
    - identifier l'endroit où la tâche réveilleur doit envoyer son signal et décider s'il s'agit d'un **`notify_one()`** ou d'un **`notify_all()`**.

   De plus, pour que la tâche de commutation des feux soit prioritaire sur les voitures qui attendent sur les voies, il faudra que la tâche qui envoie le signal de réveil sache qu'une demande de commutation a été formulée : un attribut supplémentaire sera utilisé à cette fin.

Trace d'exécution du carrefour sans respect du FIFO

événements modifiant la file secondaire	**file secondaire**
S0 ARRIVE, FEU S ROUGE	S0
S1 ARRIVE , FEU S ROUGE	S1 | S0
COMMUT FEU S VERT, carrefour occupé	S0 | S1
P1 QUITTE , FEU S VERT -> S1 TRAVERSE	S0
.	
.	
.	
.	
.	
.	
.	
.	