# TP2 – Interblocage

Dû le 9 mars à 23h00

## Préambule

**Attention!** Ce TP est le deuxième d'une suite de TPs qui vont se suivre. Il est donc important pour vous de faire le meilleur TP que vous pouvez faire, et de coder le plus clairement possible, puisque les prochains TPs de la série vont demander de réutiliser le code écrit. À chaque TP, vous aurez l'occasion de prendre une version corrigée. Cependant, vous ne serez peut-être pas à l'aise avec la structure du code, ce qui pourrait vous ralentir. Afin de s'assurer de pouvoir terminer correctement, on conseille de comparer votre solution avec le corrigé. Comme ce TP est la suite du premier, vous devez utiliser le première comme base pour compléter celui-ci.
**Attention!** Lisez bien, AU COMPLET, cet énoncé avant de commencer. Plusieurs choses sont décrites plus loin qui vont vous aider à bien structurer votre TP.
**Attention!** Ce TP est plus long (il vaut 15% de votre note finale)! Ne vous y prenez pas à la dernière minute.

## Introduction

Le but de ce TP est d'implémenter la gestion de ressources en C, tel qu'un système d'exploitation le ferait. Pour ce faire, votre shell va devoir être **significativement** modifié. En effet, nous allons imposer une limite au shell. Celui-ci doit contrôler les commandes qu'il exécute et limiter l'exécution de manière à satisfaire les contraintes de ressources. Chaque commande va compter dans une catégorie de ressource spécifique (selon un tableau qui sera défini plus tard, ou selon des arguments passés en entrée du shell). Lors de l'exécution d'une ligne de commandes, les ressources seront demandées au système selon la sémantique de la commande. Par exemple, une commande && entre deux commandes va allouer les ressources pour exécuter la partie de gauche et ensuite la partie de droite (seulement si la partie de gauche retourne 0). L'algorithme du banquier va vous permettre d'exécuter toutes les commandes dans un ordre qui vise à ne jamais obtenir de *deadlock* et d'éviter le *ressource-starvation*. En effet, votre shell va toujours être en train d'exécuter quelque chose si des entrées sont demandées. Afin de tester que l'algorithme du banquier fonctionne bien, nous **bombarderons** votre shell avec des commandes à exécuter en arrière-plan de façon à forcer l'algoritme à fonctionner. Nous vous recommandons de faire de même.

## Catégories de commandes

Les commandes que votre shell va exécuter seront divisées en catégories selon ce tableau (par défaut, on verra que l'initialisation du shell peut ajouter des catégories).
Il vous faut gérer ces catégories. Les commandes qui ne sont ni dans ce tableau, ni dans des configurations supplémentaires (voir le prochain paragraphe) vont aller dans une catégorie spéciale, appelée la catégorie ***other***, catégorie qui sera aussi limitée afin d'offrir un défi supplémentaire.

## 1 Initialisation du shell

Afin d'initialiser le shell, on considérera la première ligne entrée dans celui-ci comme étant spéciale. En effet, celle-ci doit avoir le format suivant:

```
(command_name[,]){n}&(command_cap[,]){n}&(file_system_cap)&(network_cap)
&(system_cap)&(other_cap)
```

| Catégorie système de fichier | Catégorie réseau | Catégorie système |
|---|---|---|
| ls | ping | uname |
| cat | netstat | whoami |
| find | wget | exec |
| grep | curl | |
| tail | dnsmasq | |
| head | route | |
| mkdir | | |
| touch | | |
| rm | | |
| whereis | | |

Table 1: Tableau des catégories de commandes

Les commandes nommées dans `command_name` seront des commandes qui sont leur propre catégorie. Leur limite est décrite par le `command_cap` associé (voir l'exemple). Le `file_system_cap`, le `network_cap`, le `system_cap` ainsi que le `other_cap` correspondent au limites respectives du nombre de commandes dans la catégorie "système de fichier", "réseau", "système" et "other" respectivement. Voici un exemple de ligne d'entrée:

```
echo,sed&5,27&12&11&14&2
```

L'interprétation de cette ligne nous indique qu'on doit limiter la commande `echo` à 5 utilisations concurrentes, la commande `sed` à 27 et ainsi de suite pour les catégories de commandes.
Notez cependant qu'on peut redéfinir le nombre alloué des commandes. Par exemple, si on dit que "`ls`" a seulement une ressource, alors lorsqu'on appelle un `ls`, on regardera seulement cette ressource, et non celle du système de fichier.
Afin de gérer la configuration du shell, vous devez d'abord implémenter les fonctions suivantes:

### 1.1 error_code parse_first_line(char* first_line);

Cette fonction va prendre en paramètre la première ligne d'entrée. Vous devez remplir correctement (après avoir allouée la mémoire requise) la structure globale de configuration et retourner un code d'erreur en cas d'erreur.
Attention! Les numéros de ressource 0, 1, 2 sont réservés aux ressources de type "système de fichier", "réseau" et "système". Les autres ressources sont numérotées selon leur ordre d'apparition dans la ligne de configuration, avec la ressource de type "*other*" étant la dernière.

### 1.2 unsigned int ressource_no(char* cmd)

Cette commande prend en paramètre une chaîne de caractères qui représente une commande shell. Elle retourne le numéro de la ressource correspondant à la commande reçue en paramètres (trouvé dans la configuration globale).

### 1.3 unsigned int ressource_count(unsigned int cmd_no)

Cette commande prend en paramètre une catégorie de commande et retourne la quantité de ressources disponibles pour la catégorie (trouvée dans la configuration globale). Veuillez porter

attention à la catégorie de la commande. Le numéro peut correspondre à une catégorie *statique* ou une catégorie dynamique, comme une catégorie obtenue par la chaîne de configuration décrite plus haute.

Avec ces fonctions, vous devriez être en mesure d'analyser les commandes selon les ressources qu'elles utilisent.

## Au sujet de `rN(cmd)` et `fN(cmd)`

Comme vous l'aurez peut-être deviné, la commande `rN(cmd)` va exécuter une commande plusieurs fois. Cependant, elle ne compte pas comme une seule commande, mais bien comme $N$ commandes (en terme de ressources utilisées). Lorsqu'il sera le temps d'allouer des ressources pour une commande, il faudra en tenir compte.
La commande `fN(cmd)` fait exactement l'inverse. Celle-ci libère des ressources et compte comme un nombre **négatif** de ressources selon la catégorie de la commande. Il vous suffit donc de compter les ressources comme un compte négatif et de désallouer les ressources correspondantes.

# 2 Lors de l'exécution des commandes

Lorsque vous lisez une ligne en entrée, vous devez maintenant analyser la quantité maximale de ressource qu'une ligne de commandes peut demander. Vous devez implémenter les fonctions suivantes:

### 2.1 error_code* create_command_chain(char* line, command_head **result)

Cette fonction doit créer une liste chaînée dont la structure ce trouve dans le fichier `main.c` donné en annexe. Dans un cas d'erreur, on retourne un code d'erreur. La liste chaînée possède un format particulier. Le premier bloc est un bloc d'entête, qui va servir à définir le mode d'exécution du reste des commandes. Le format est décrit en annexe.
Les blocs de commandes, qui suivent l'entête, ont aussi un format décrit en annexe. Afin d'assigner les bonnes valeurs aux ressources utilisées, vous aurez besoin des fonctions décrite ci-dessous. Le parsing des commandes de votre TP1 doit être déplacé ici.
Attention! Cette fonction requiert que les fonctions suivantes aient été implémentées! En effet, après l'exécution entière de la présente fonction (une fois que la chaîne a été créer et correctement créée), vous devrez appeler `evaluate_whole_chain` pour compléter la liste chaînée.

### 2.2 error_code count_ressources(command_head *head, command* command_block)

L'entête de la chaîne de commande et un bloc de commande. Son objectif est de compter les ressources utilisées par le bloc de commande en paramètre. Le tableau `ressource` du block est alloué selon le nombre de type de ressources configurées (voir configuration) et va être initialisé de façon à contenir la quantité de chaque ressources qui peuvent être allouées. Donc, si on demande un `ls`, il faut demander une ressource de système de fichier (ou de la catégorie "ls", si on a spécifié un nombre de ressources spécifique à `ls`).
Nous ajoutons ici l'exception que les commandes `fN(cmd)` vont avoir un compte négatif. En effet,

ces commandes vont servir à libérer des ressources. Il faut donc en tenir compte. La fonction retourne un code d'erreur s'il y a lieu.

### 2.3 error_code evaluate_whole_chain(command_head *head)

Cette fonction prend l'entête d'une chaîne de commande et évalue la quantité maximale de ressources concurrentes possibles pour la chaîne en faisant la somme des ressources de chaque bloc dans la chaîne.
Pour chaque bloc dans la chaîne, un appel doit être fait à la fonction `count_ressources` (pour initialiser les tableaux de ressources du bloc de commande). Elle affecte ensuite le tableau `max_ressources` dans la structure `command_chain_head` à un tableau qui contient les bonnes valeurs.

Notez que pour les deux dernières fonctions, et pour tous les tableaux de ressources de ce TP, le tableau doit mesurer `3+1+nb_other_ressources` de long: 3 pour les catégories prédéfinies, 1 pour "other", et un nombre variable selon la ligne de configuration.
**Notez aussi que dès maintenant, toutes les lignes de commandes "en arrière plan" (donc, se terminant avec un `&`) devront partir leur propre thread, qui se fermera une fois l'exécution de la ligne terminée.**

## 3 Contrôle des ressources

### 3.1 banker_customer *register_command(command_head *head)

Cette fonction doit enregistrer la chaîne de commande dans une liste chaînée de `banker_customer` dont la structure est décrite en annexe. Comme plusieurs threads de client peuvent appeler cette fonction, afin de garder la cohérence dans le tableau vous devez utiliser les mécanismes de synchronisation nécessaires et suffisants.
En cas d'erreur, `NULL` doit être retourné. Sinon, on retourne un pointeur vers la structure.

### 3.2 error_code unregister_command(banker_customer *customer)

Cette fonction fait l'inverse de la fonction précédente.
Étant donné un client du banquier, elle détruit l'enregistrement associé (celui passé en paramètres). Attention à bien mettre à jour la liste chaînée et à bien libérer les ressources: il faut dissocier le client, et remplacer le prev et le next de ses voisins par les bonnes références.

### 3.3 error_code request_ressource(command_head *head, int cmd_depth)

Cette fonction doit demander au banquier de donner l'accès sur les ressources à un client.
Cette fonction va utiliser le mutex dans la structure `command_head` pour se verrouiller et attendre que le banquier déverrouille l'accès afin d'entrer. Vous devrez être astucieux avec le mutex: il faut en effet que le client bloque tant que le banquier ne lui a pas dit "oui, tu as le droit d'accéder aux ressources". Un seul appel à `pthread_mutex_lock` ne suffira probablement pas...
Le paramètre *depth* constitue la profondeur dans la chaîne de commande et constitue donc une

requête précise faite par le client. La profondeur commence à 0.

# 4 Thread banquier

### 4.1 void *banker_thread_run()

Ceci est la fonction que le thread banquier va exécuter sans fin (donc, cette fonction contient une boucle infinie).
À chaque tour de boucle, ce thread banquier doit:

1. Acquérir le mutex d'enregistrement.
2. Parcourir tous les clients enregistrés
3. En trouver un dont le `depth` n'est pas -1 (si on n'en trouve pas, le mutex est déverrouillé et on passe au prochain tour de boucle)
4. Appeler `call_bankers` sur ce client
5. Déverrouiller le mutex d'enregistrement

### 4.2 void call_bankers(banker_customer *customer)

Cette fonction prépare l'appel à `bankers`. On y alloue les tableaux pertinents (`work` et `finish`), on y teste les conditions de deadlock évidentes (qui n'ont pas besoin de l'algorithme du banquier), et on y change provisoirement `available` (le tableau des ressources qu'on a en banque) comme si la requête à la profondeur `customer->depth` avait passé.
Ensuite, on appelle `bankers`.
Toute cette fonction est une section critique. Elle doit acquérir le mutex d'`available` avant de faire quoi que ce soit, et le libérer juste avant de terminer son exécution.
C'est aussi dans cette fonction qu'on libérera (OU PAS) les clients qui attendent sur leur propre mutex (tel que décrit dans `request_ressources`) selon la valeur de retour de `bankers`.
Si `bankers` indique que la requête n'est pas sécuritaire, il faut défaire les changements provisoires qu'on à fait à `available`.

### 4.3 int bankers(int *work, int *finish)

Cette fonction sert à rouler l'algorithme du banquier.
Elle retourne faux si la requête décrite par work et finish n'est pas valide/sécuritaire.
Elle retourne vrai sinon.

# 5 Tout mettre ensemble

Cette question requiert de mettre ensemble toutes les petites fonctions qui ont été implémentées dans les exercices précédents. Pour cela, il va falloir ajouter une `pthread` par exécution de ligne lue en entrée qui doit être exécutée en arrière-plan (afin de pouvoir utiliser le modèle producteur-consommateur que nous avons ici défini). Le thread du banquier doit s'exécuter tout au long de l'exécution du shell.

Quand un thread d'une commande termine (parce que toutes les commandes sont été exécutées), il demande au banquier de l'oublier en appelant la fonction `unregister_command`.

### 5.1 Fonction `exit`

Vous devez aussi implémenter une façon de sortir de votre shell: lorsqu'on entre la ligne `"exit"`, le shell doit terminer son exécution.

## Structure supplémentaire

Afin d'initialiser des ressources globales, un "constructeur" et un "destructeur" sont mis à votre disposition. Vous n'êtes pas obligé de les utiliser, mais vous verrez qu'ils simplifient grandement le travail. Une fonction de lecture de ligne est aussi fournie avec un petit peu de code afin de vous aider à démarrer votre TP.
Ces fonctions seront appelés uniquement une seule fois dans une contexte d'exécution de votre TP (une fois au début, une fois à la fin). Elles peuvent servir à l'initialisation de ressources globales qui doivent être dynamiquement allouées et à leur nettoyage. Si la fonction `init_shell` retourne un code d'erreur, l'initialisation ne sera pas faite. Si vous n'avez pas la mémoire requise pour démarrer le shell, c'est le comportement voulu.

## Modification aux structures existantes

Vous pouvez modifier les structures qui sont données dans le template en ajoutant des champs si vous pensez que c'est nécessaire. Cependant, vous devez laisser les champs existants en place, les remplir correctement et ne pas modifier **AUCUN** champ, nom de structure ou nom de champ. À titre indicatif, le solutionnaire utilise exactement le code fourni.
Vous pouvez ajouter des fonctions pour rendre votre code plus propre, ce qui est **fortement** recommandé et fait dans le solutionnaire.

## Remise

Ce travail est à faire en groupes de 2 étudiants. Le code est à remettre par remise électronique avant la date indiquée.
Chaque jour de retard est -15%, mais après le deuxième jour la remise ne sera pas acceptée.
Indiquez clairement votre nom et votre matricule dans un commentaire en bloc au début de chaque fichier, dans ce format:

```
/*
 *   Prenom1 NomDeFamille1 Matricule1
 *   Prenom2 NomDeFamille2 Matricule2
*/
```

Si un étudiant préfère travailler seul, libre à lui, mais l'évaluation de son travail n'en tiendra pas compte. Nous n'aiderons pas ceux qui ne se trouvent pas de partenaire à s'en trouver un.
Le programme doit être exécutable sur les ordinateurs du DIRO.
Pour la remise, vous devez remettre le fichier main.c en utilisant **ce lien** dans un fichier zip. Assurez-vous que tout fonctionne correctement sur les ordinateurs du DIRO.

# Barème

- Votre note sera divisée équitablement entre chaque question et un test global qui vérifie si tout fonctionne bien ensemble. Donc, avec 5 questions et un test global, on obtient 16.67% par question et pour le test global.
- Tout usage de matériel (code ou texte) emprunté à quelqu'un d'autre (trouvé sur le web, etc.) doit être dûment mentionné, sans quoi cela sera considéré comme du plagiat. Si pour une question votre solution est directement copiée, même si il y a attribution de la source, cette question se verra attribuée la note de zéro. Vous pourrez cependant l'utiliser dans les sections suivantes sans pénalité.
- Vous serez pénalisés pour chaque warning lors de la compilation à raison de 1% par warning, sauf pour les warning reliés à l'assignation à NULL, à la comparaison avec NULL, et aux override des fonctions de librairie.
- Si une fuite mémoire mémoire est identifiée, vous perdrez 15%. Vous ne perdrez pas plus de points pour les fuites si vous en avez une ou trente.
- Les accès mémoire illégaux identifiés par Valgrind entraîneront jusqu'à 5% de perte, à raison de 1% par accès. La répétition d'un même accès sera comptée comme 1% de plus quand même.
- Votre devoir sera corrigé automatiquement en très grande partie. Si vous déviez de ce qui est demandé en output, les points que vous perdrez seront perdus pour de bon. Si vous n'êtes pas certains d'un caractère demandé, demandez sur le forum studium et nous répondrons de façon à ce que chaque étudiant puisse voir la réponse.
- La méthode de développement recommandée est d'utiliser CLion et son intégration avec Valgrind. Si vous voulez utiliser d'autres techniques, vous pouvez le faire, mais nous ne vous aiderons si vous rencontrez des problèmes avec ces techniques.