Programmation système avancée – projet

1 Modalités

Le projet doit être réalisé en binôme (éventuellement, en monôme). Les soutenances auront lieu au mois de mai, sans doute à partir du 22 mai, la date exacte sera communiquée ultérieurement. Pendant la soutenance, tous les membres d'un binôme devront montrer leur maîtrise de la totalité du code.
Chaque équipe doit créer un dépôt git privé sur le gitlab de l'UFR :

`https://gaufre.informatique.univ-paris-diderot.fr`

dès le début de la phase de codage et y donner accès en tant que *Reporter* à tous les enseignants de cours et TP de Systèmes avancés.
Le dépôt devra contenir un fichier « equipe » donnant la liste des membres de l'équipe (nom, prénom, numéro étudiant et pseudo(s) sur le gitlab). Vous êtes censés utiliser gitlab de manière régulière pour votre développement. Le dépôt doit être créé **le 17 avril au plus tard**. Au moment de la création du dépôt, vous devez envoyer un mail à `zielonka@irif.fr` avec la composition de votre équipe (l'objet du mail doit être « `[syst av] projet` », c'est important si vous ne voulez pas que votre mail se perde).
Le guide de connexion externe et la présentation du réseau de l'UFR se trouvent sur :
`http://www.informatique.univ-paris-diderot.fr/wiki/howto_connect`
et `http://www.informatique.univ-paris-diderot.fr/wiki/linux`
Le projet doit être accompagné d'un `Makefile` utilisable. Les fichiers doivent être compilés avec les options `-Wall -g` sans donner lieu à aucun avertissement (ni erreur bien évidemment). `make clean` devra supprimer tous les fichiers exécutables et les fichiers `*.o` de telle sorte que le `make` suivant permette de recompiler complètement le projet.
La soutenance se fera à partir du code déposé sur le gitlab. Au début de la soutenance vous aurez à cloner votre projet à partir du gitlab et le compiler avec `make`. Préparez-vous pour qu'on ne passe pas 15 minutes à cloner le projet et le compiler. Normalement les deux tâches ne doivent pas prendre plus qu'une minute, si ce n'est pas le cas alors vous aurez moins de temps pour présenter votre travail.
Vous devez fournir un jeu de tests permettant de vérifier que vos fonctions sont capables d'accomplir les tâches demandées, en particulier quand plusieurs processus lancés en parallèle accèdent aux fichier en posant des verrous.
Si vous avez des questions merci de les poser uniquement sur le forum sur moodle.

2 Rappel sur les verrous de fichiers

Les verrous BSD `flock` sont associés aux éléments de la table des ouvertures de fichiers (voir les transparents du cours sur les fichiers). Cela implique que :

(1) quand on duplique un descripteur de fichier avec `dup` ou `dup2`, le verrou est partagé par le nouveau descripteur (puisqu'il référence la même entrée dans la table des ouvertures de fichiers) ;

(2) si un processus pose un verrou sur le fichier et exécute `fork`, le processus enfant partage le verrou avec son parent ; en particulier, si un des deux processus lève ensuite le verrou, l'autre processus le perd également ;

(3) si un processus pose un verrou sur un fichier puis ouvre une deuxième fois *le même fichier*, le deuxième appel à `open` crée une nouvelle entrée dans la table des ouvertures de fichiers sans verrou [1],

(4) les verrous sont préservés par `exec`.

L'inconvénient des verrous BSD réside dans l'impossibilité de limiter le verrouillage à un segment de fichier : pour accéder même à un tout petit segment de fichier, il faut verrouiller le fichier entier.

Les verrous POSIX (`fcntl`) permettent de verrouiller un segment de fichier (*record locking*). Ces verrous sont associés aux couples (processus, inode) (voir les transparents du cours sur les fichiers pour la relation entre les tables de descripteurs, la table des ouvertures de fichiers et la table des inodes, et les transparents du cours sur les verrous de fichiers pour le dessin montrant l'implémentation des verrous POSIX).
Cela implique que :

(A) la duplication de descripteurs préserve les verrous (puisque ni le inode ni le processus ne changent) ;

(B) immédiatement après le `fork`, le processus enfant ne possède aucun verrou : le parent ne peut pas transmettre un verrou à son enfant puisque leurs pids sont différents ;

(C) si un processus pose un verrou sur un fichier puis ouvre le même fichier une deuxième fois, cette deuxième ouverture possède les mêmes verrous (le pid du processus et l'inode du fichier sont les mêmes) ;

(D) les verrous sont préservés par `exec`.

Cependant les verrous POSIX posent le problème suivant qui les rendent peu commodes. Imaginons le scénario suivant :

```
struct flock fl;

fl.l_type = F_WRLCK;
fl.l_whence = SEEK_SET ;
fl.l_start= 0 ;
fl.l_len = 0 ; /* verrou s'étend automatiquement jusqu'à la fin du fichier
                * même dans le cas d'ajout d'octet à la fin du fichier */
int fd1 = open("testfile", O_RDWR);
if( fcntl( fd1, F_SETLK,  &fl) < 0 )
    PANIC_EXIT("fcntl");

int fd2 = open("testfile", O_RDWR);
close(fd2);
```

1. en particulier, une tentative de poser un verrou sur le nouveau descripteur est bloquante

L'appel à `close(fd2)` à la ligne 14 lève tous les verrous posés sur `fd1` puisque pour les deux descripteurs il s'agit du même processus et du même inode.
À l'inverse, si on réécrit le même fragment du code en utilisant les verrous BSD le descripteur `fd2` ne partagerait pas les verrous avec `fd1` (même inode mais deux entrées différentes dans la table des ouvertures de fichers) donc `close(fd2)` n'aurait aucune incidence sur les verrous posés avec `fd1`.

3 Réimplémenter les verrous POSIX

Le but du projet est d'implémenter des verrous qui permettent de verrouiller des segments de fichiers, comme les verrous POSIX, mais en évitant le comportement décrit à la fin de la section précédente.
Le problème, bien évidemment, est que nous n'avons pas d'accès au noyau du système donc nous ne pouvons rien ajouter à la table des ouvertures de fichiers.
L'idée est de mémoriser les structures de données nécessaires dans une mémoire partagée obtenue par la projection (`mmap`) d'un shared memory object (SMO) (`shm_open`).

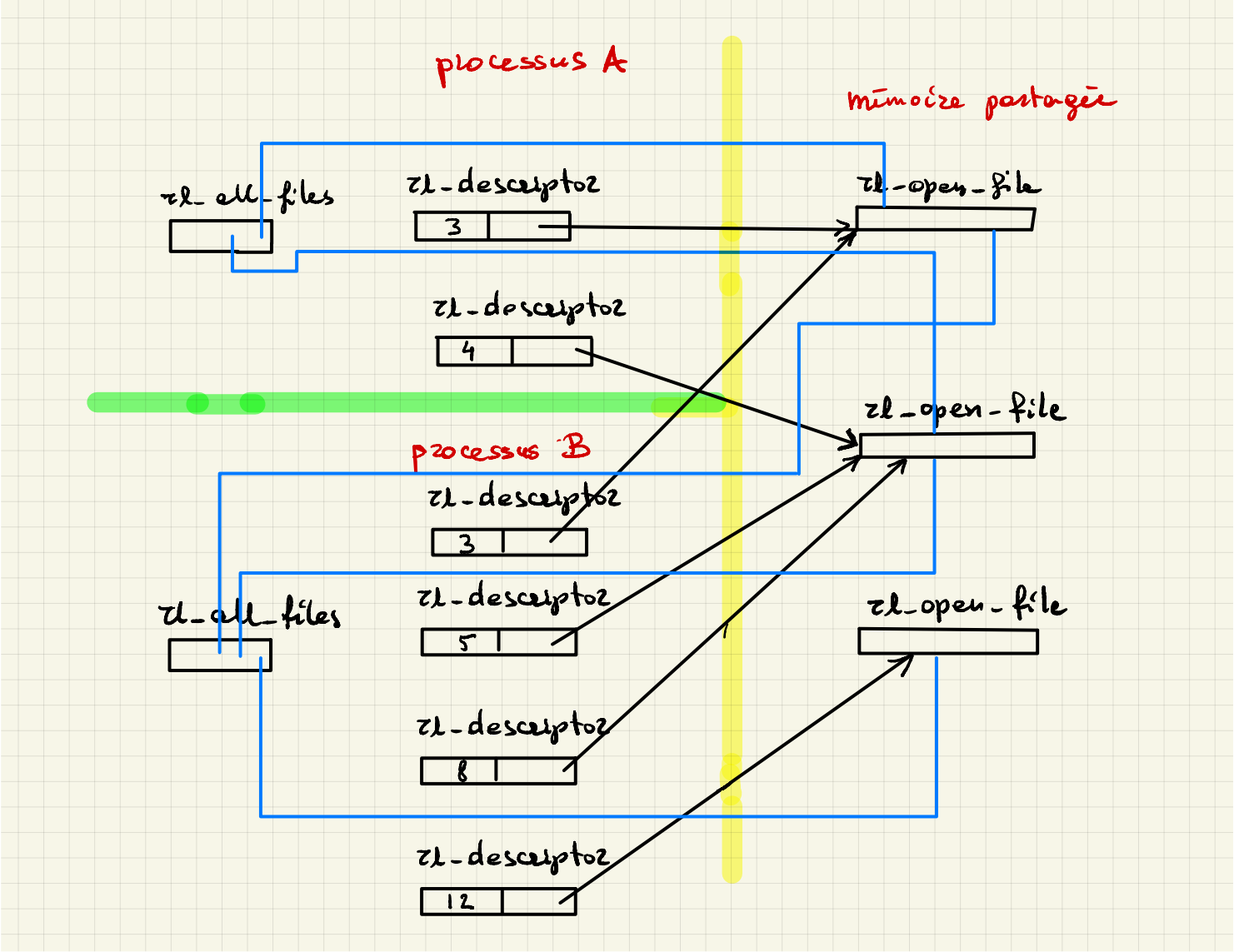
4 Organisation du code

Les définitions de toutes les fonctions de la bibliothèque doivent être regroupées dans un seul fichier `rl_lock_library.c` (`rl` comme l'abréviation de *region lock*). Le nom de ce fichier source ne peut pas être modifié. (Cela permettra de lancer plus facilement le détecteur de plagiat, donc si vous mettez les fonctions dans un autre fichier je vais considérer cela comme une tentative de se soustraire à la procédure de détection de plagiat ce qui aura une influence négative sur la note.) Au début du fichier `rl_lock_library.c` vous mettrez en commentaire les noms des membres de votre binôme.
Les fonctions auxiliaires, qui ne doivent pas être visibles par l'utilisateur de la bibliothèque, doivent être déclarées `static`. La bibliothèque peut (et en fait, elle doit, comme nous verrons plus tard) utiliser des variables globales qui, pour rester invisibles pour l'utilisateur, seront déclarées comme `static`.
Le fichier en-tête `rl_lock_library.h` doit contenir les déclarations de toutes les fonctions utilisables directement pour l'utilisateur de la bibliothèque (les fonctions sans attribut `static`).
Séparément, vous devez fournir dans un ou plusieurs fichiers des programmes permettant de tester le bon fonctionnement de la bibliothèque.
Si cela est commode vous pouvez fournir des scripts (bash ou autre) qui permettent de lancer les programmes de test – cela est pertinent car tester la bibliothèque nécessite de lancer plusieurs processus en parallèle.

5 Structures de données de la bibliothèque `rl_lock_library`

5.1 Présentation générale

Le dessin suivant



présente schématiquement les structures de données utilisées dans le projet. Le dessin présente deux processus, A et B, qui posent des verrous sur 3 fichiers différents. Le processus B possède deux descripteurs (5 et 8) qui désignent le même fichier.

Pour chaque fichier traité à l'aide de la bibliothèque, il nous faut une structure permettant de mémoriser les verrous posés sur des segments du fichier par différents processus. Cette structure `rl_open_file` est présentée en détail dans la section 5.3.2.
Si un processus pose les verrous sur des régions d'un fichier, tout autre processus qui utilise le même fichier doit être capable de « voir » quelles sont les régions verrouillées. Cela implique que les verrous doivent être placés dans une mémoire partagée par les processus qui utilise la bibliothèque. Pour chaque fichier (ou plus précisément pour chaque inode) il y aura une structure `rl_open_file` stockée dans un shared memory object projeté en mémoire avec `mmap` de type `MAP_SHARED`, cette structure contient un tableau des verrous posés sur des régions du fichier.

Pour accéder aux objets `rl_open_file`, chaque processus doit maintenir l'adresse de chacun de ces objets pour tous les fichiers qu'il traite à l'aide de la bibliothèque. La bibliothèque a besoin de cette référence, tandis que les opérations usuelles sur les fichiers uti-

lisent les descripteurs ; il est donc commode de stocker ensemble l'adresse de `rl_open_file` et le descripteur du fichier concerné. La structure `rl_descriptor`, décrite en détail dans la section 5.4, stocke le couple (descripteur, l'adresse de `rl_open_file`) et sera utilisée pour réimplémenter les fonctions POSIX qui utilisent le descripteur de fichier. Contrairement à `rl_open_file` les structures `rl_descriptor` ne sont pas partagées entre les processus.

Quand un processus exécute `fork()`, il faut que le nouvel enfant « hérite » des verrous de son parent. Pour cela, il faut être capable de retrouver tous les verrous du parent ; cela est possible si la bibliothèque garde les adresses de tous les objets `rl_open_file` utilisés par le processus. La bibliothèque maintient donc une variable globale (et `static`) `rl_all_files` qui contient les références vers tous les objets `rl_open_file` utilisés par le processus. Cette variable est déclarée `static` dans la bibliothèque, donc l'utilisateur n'aura pas d'accès direct à cette variable : elle sera disponible uniquement à l'intérieur de la bibliothèque. La variable `rl_all_files` est présentée dans la section 5.5.

5.2 Structure d'un verrou - `rl_lock`

Chaque verrou sur un segment de fichier possède un ou plusieurs propriétaires. Le propriétaire du verrou est identifié à un couple décrit par la structure `owner` :

```
typedef struct{
   pid_t proc;  /* pid du processus */
   int des;     /* descripteur de fichier */
} owner;
```

La structure `rl_lock` décrit un verrou sur un segment de fichier :

```
typedef struct{
   int next_lock;
   off_t starting_offset;
   off_t len;
   short type; /* F_RDLCK F_WRLCK */
   size_t   nb_owners;
   owner lock_owners[NB_OWNERS];
} rl_lock;
```

- le champ `starting_offset` indique le décalage du début du segment par rapport au début du fichier ;
- le champ `len` donne la longueur du segment ; si `len == 0`, alors le segment protégé par ce verrou s'étend jusqu'à la fin du fichier (il est de longueur variable, et s'étennd automatiquement si on ajoute des octets à la fin du fichier) ;
- `type` donne le type de verrou ; on utilisera les mêmes constantes que pour le champ `l_type` de la structure `flock` ;
- `lock_owners` est le tableau des propriétaires du verrou : un verrou peut avoir plusieurs propriétaires par exemple suite à un `fork` ; c'est à vous de choisir la taille `NB_OWNERS` du tableau ;
- `nb_owners` est le nombre de propriétaires effectifs du verrou ; les propriétaires effectifs occupent les premières `nb_owners` cases du tableau `lock_owners` ;

– les structures `rl_lock` sont stockées dans un tableau [2] mais elles n'en occupent pas des cases successives : elles forment une liste chaînée et `next_lock` est l'indice de l'élément suivant dans la liste ; il vaut `-1` pour le dernier élément de la liste et `-2` si la case correspondante du tableau n'est pas utilisée (case libre).

5.3 Contenu de la mémoire partagée

5.3.1 Shared memory object

Chaque fichier ouvert sera associé à un shared memory object dont le nom doit correspondre à l'identité du fichier. Rappelons encore une fois qu'un fichier est identifié par le couple `(st_dev, st_ino)` de la structure `struct stat`. Le nom du shared memory object correspondant au fichier sera construit de façon suivante :

`"/f_dev_ino"`

où `/f` est un préfixe fixe, et `dev` et `ino` sont respectivement le numéro de la partition (device) et le numéro d'inode du fichier. Par exemple si `dev==4` et `ino==1000564` alors le nom du shared memory object sera `"/f_4_1000564"`. (À la place de `"f"`, vous pouvez utiliser une autre chaîne de caractères, le plus commode étant de ne pas fixer le préfixe en dur dans le programme mais de le spécifier par exemple via une variable d'environnement.)

5.3.2 Le contenu du shared memory object

Chaque shared memory object contient une structure

```
typedef struct{
  int first;
  rl_lock lock_table[NB_LOCKS];
} rl_open_file;
```

où `lock_table` est un tableau de verrous de segments du fichier correspondant. Les verrous sont rangés dans une liste. Le champ `first` est l'indice du premier élément et chaque `rl_lock` contient le champ `next_lock` qui donne l'indice de l'élément suivant (voir la description dans la section 5.2). Il est sans doute commode que la liste de `rl_lock` soit triée dans l'ordre croissant de `starting_offset`. Initialement toutes les cases de `lock_table` sont libres (donc `first == -2` et `next_lock == -2` pour chaque verrou).

5.4 `rl_desc` - les descripteurs

Les fonctions de la bibliothèque ne pourront pas travailler directement avec les descripteurs de fichiers. Pour chaque fichier ouvert le processus aura donc besoin, en plus de descripteurs, du pointeur vers le shared memory object décrit dans la section 5.3.
Les descripteurs habituels seront remplacés par la structure

```
typedef struct{
  int d;
  rl_open_file *f;
} rl_descriptor;
```

2. défini dans la section 5.3.2

Le champ `d` de `rl_descriptor` est un descripteur de fichier habituel. Le champ `f` est un pointeur vers la mémoire partagée contenant la structure `rl_open_file` de la section précédente.

5.5 Vecteur des fichiers ouverts

L'implémentation de la fonction qui remplace `fork()` nécessite des données supplémentaires. Pour chaque processus on définit donc une variable globale `static rl_all_files` :

```
#define NB_FILES 256
static struct {
    int nb_files;
    rl_open_file *tab_open_files[NB_FILES];
}  rl_all_files;
```

Les `nb_files` premières cases du tableau `rl_open_file` contiennent les pointeurs vers toutes les structures `rl_open_file` utilisées par le processus. À chaque création d'un nouvel objet `rl_open_file` (voir la section 6.1), la fonction `rl_open` ajoute l'adresse de cet objet dans le tableau `tab_open_files` et met à jour le champ `nb_files` qui donne le nombre de cases de `tab_open_files` effectivement utilisées.
(À la place d'un tableau `tab_files` vous pouvez aussi utiliser une liste chaînée).

5.6 Compléter les structures de données

Les définitions de structures `rl_open_file` et `rl_lock` *ne sont pas complètes*. Rappelons que `rl_open_file` réside dans la mémoire partagée obtenue par la projection en mémoire d'un shared memory object. Plusieurs processus peuvent accéder ou essayer d'accéder à cette mémoire en même temps. Pour la protection de la mémoire il faudra utiliser soit des variables mutex et conditions soit des sémaphores. C'est à vous d'ajouter ces objets dans les définitions de `rl_open_file` et éventuellement `rl_lock`.
Un projet qui ne prévoit pas de protection de mémoire et qui permet, par exemple, que plusieurs processus modifient la mémoire partagée en même temps (ou plus exactement permet de modifier les mêmes éléments de la mémoire partagée en même temps) ne vaut pas grand-chose.
D'un autre côté, pour assurer le maximum de parallélisme, la possibilité de lecture simultanée de la mémoire partagée par plusieurs processus sera appréciée.
Vous pouvez ajouter d'autres champs dans les structures de données proposées. Il est même possible de redéfinir complètement l'architecture de la bibliothèque à condition de garder toutes les fonctionnalités décrites dans la section 6.

6 Fonctions à implémenter par la bibliothèque `rl_lock_library`

La bibliothèque à implémenter contient les fonctions qui remplaceront certaines fonctions POSIX. Dans la plupart des cas, la nouvelle fonction porte le nom obtenu par l'ajout du préfixe `rl_` et elle garde les mêmes paramètres que la fonction POSIX correspondante avec une modification : à la place d'un descripteur de fichier de type `int`, la nouvelle fonction utilisera un pointeur vers un `rl_descriptor`. Si la nouvelle fonction est conforme à cette règle nous n'allons pas décrire les paramètres en détail.

6.1 `rl_open`

```
rl_descriptor rl_open(const char *path, int oflag, ...)
```

La fonction `rl_open` ouvre le fichier à l'aide de `open` (et ses paramètres sont identiques à ceux de la fonction `open`).
Le champ `d` de la structure `rl_descriptor` retournée par la fonction est le descripteur de fichier ouvert, ou -1 si `open` échoue.
Le champ `f` de `rl_descriptor` contient l'adresse de la mémoire partagée. Deux possibilités pour obtenir cette adresse :

- si le shared memory object associé au fichier[3] existe déjà, alors `rl_open` l'ouvre et le projette en mémoire
- sinon, `rl_open` crée un shared memory object de taille `sizeof(rl_open_file)`, le projette en mémoire, et initialise les éléments de tableau `lock_table`.

Le cas échéant, il faudra mettre à jour la variable `rl_all_files`.
(Si vous ne savez pas comment découvrir si un shared memory object existe ou non regardez sur moodle le code du premier exemple dans le cours 07).

6.2 Remarque sur les propriétaires des verrous

Dans les fonctions qui suivent, un des paramètres est `rl_descriptor lfd`. Définissons

```
owner lfd_owner = { .proc = getpid(), .des = lfd.d };
```

Intuitivement `lfd_owner` est le propriétaire (`owner`) associé au processus courant et au descripteur de fichier codé dans `lfd`.
Soit `owner ow` un autre objet de type `owner`. On dit que `ow` est égal à `lfd_owner` si

`lfd_owner.proc == ow.proc` et `lfd_owner.des == ow.des`.

On dit que `lfd_owner` appartient au tableau `lock_table` (la section 5.3.2) si `lfd_owner` est égal à un élément de ce tableau.

6.3 `rl_close`

```
int rl_close( rl_descriptor lfd)
```

La fonction ferme le descripteur `lfd.d` en utilisant l'appel à `close`, et supprime tous les verrous associés à ce descripteur et au processus appelant : chaque structure `rl_lock` du tableau `lfd.f->lock_table` contient le tableau `lock_owners` des propriétaires du verrou ; si `lfd_owner`[4] est présent dans ce tableau, on le supprime du tableau ; s'il s'agissait de l'unique propriétaire du verrou, on supprime aussi le verrou lui-même.

3. et dont le nom est obtenu par le procédé expliqué dans la section 5.3.1
4. défini dans la section 6.2

6.4 `rl_fcntl`

Il est temps de réécrire la fonction qui pose et enlève les verrous :

```
int rl_fcntl(rl_descriptor lfd, int cmd, struct flock *lck)
```

La structure `struct flock` est la même que pour `fcntl` :

```
struct flock{
  short rl_type; /* F_RDLCK F_WRLCK  F_UNLCK */
  short rl_whence; /* SEEK_SET SEEK_CUR SEEK_END */
  off_t rl_start; /*offset où le verrou commence*/
  off_t len; /* la longueur de segment*/
  pid_t pid;  /* non utilisé dans le projet */
}
```

La pose de verrou est possible si et seulement si aucun verrou incompatible posé sur une partie du segment n'a de propriétaire différent de `lfd_owner` décrit dans la section 6.2.
Rappelons que `cmd` peut prendre une des trois valeurs suivantes :

- `F_SETLK` pour poser ou lever le verrou, en fonction de la valeur du champ `rl_type` de la structure `struct flock` dont l'adresse est le dernier paramètre de `rl_fcntl`.
  Si la pose de verrou échoue à cause de conflits avec d'autres verrous, `rl_fcntl` retourne immédiatement -1 et `errno==EAGAIN`.
- `F_SETLKW` a le même rôle mais son comportement est bloquant : le processus est suspendu tant qu'il existe des verrous incompatibles sur une partie du segment à verrouiller.
- `F_GETLK` n'est pas utilisé dans le projet.

Dans la version de base du projet on demande d'implémenter uniquement la commande `cmd == F_SETLK`.

6.4.1 Détails de l'acquisition et de la libération de verrou

Soit `lfd_owner` défini comme dans la section 6.2.

- L'opération de lever le verrou ne lève le verrou *que* pour le propriétaire `lfd_owner` ; elle réussit toujours – en particulier si `lfd_owner` ne possède pas de verrou, elle réussit sans rien faire.
- Si `lfd_owner` est déjà propriétaire d'un verrou en lecture sur un segment et demande à poser un verrou en écriture sur le même segment, alors la demande échoue s'il y a d'autres propriétaires de verrou en lecture sur le même segment ; sinon le verrou en lecture est promu en verrou en écriture pour `lfd_owner`.
- Si `lfd_owner` est propriétaire d'un verrou en écriture sur un segment et demande un verrou en lecture sur le même segment, le type de verrou est transformé seulement pour `lfd_owner`, pas pour les co-propriétaires. Donc c'est comme si on levait le verrou pour `lfd_owner` pour ensuite ajouter un nouveau verrou pour `lfd_owner` sur le même segment sauf que l'opération doit être atomique.
- Lever le verrou au milieu d'un segment donne deux segments verrouillés : par exemple si `lfd_owner` possède un verrou sur le segment de 50 à 200 octets et lève le verrou de 100 à 150, il obtient un segement verrouillé de 50 à 100 et un deuxième de 150 à 200.

– Poser le verrou peut échouer parce qu'un processus qui n'existe plus « a oublié » de lever le verrou. Donc si poser le verrou est impossible à cause d'un verrou existant `fcntl` doit vérifier si le processus coupable `p` est toujours vivant [5]. Si le processus `p` n'existe plus alors `fcntl` doit enlever les verrous posés par `p` et réessayer de poser son verrou.

Si des situations ne sont pas couvertes par ce descriptif, n'hésitez pas à poser des questions sur le forum sur moodle, ou réfléchissez vous-même à ce qui convient le mieux. Regardez aussi la page man de `fcntl`, le comportement de `rl_fcntl` doit être calqué autant que possible sur le comportement de `fcntl`.

6.5 `rl_dup` et `rl_dup2`

```
rl_descriptor rl_dup( rl_descriptor lfd )
rl_descriptor rl_dup2( rl_descriptor lfd, int newd )
```

`rl_dup()` est censé faire la même chose que `dup`. Donc on commence par dupliquer le descripteur :

```
int newd = dup( lfd.d );
```

Ensuite on duplique toutes les occurrences de `lfd_owner` (défini comme dans la section 6.2) comme propriétaire de verrou : pour chaque verrou dont un des propriétaires est `lfd_owner` on ajoute un nouveau propriétaire `new_owner` :

```
owner new_owner = {.proc = getpid(), .des = newd } ;
```

Autrement dit, si le processus courant possède un verrou avec le descripteur `lfd.d`, il doit obtenir le même verrou avec le nouveau descripteur `newd`. Et finalement `rl_dup` retourne le nouveau `rl_descriptor new_rl_descriptor` initialisé de la façon suivante :

```
rl_desriptor new_rl_descriptor = { .d = newd, .f = lfd.f };
```

`rl_dup2( rl_descriptor lfd, int newd )` exécute `dup2( lfd.d , newd )`. Ensuite, comme pour `rl_dup`, pour chaque verrou dont un des propriétaires est `lfd_owner` on ajoute le nouveau propriétaire `new_owner` :

```
owner new_owner = {.proc = getpid(), .des = newd } ;
```

et finalement la fonction renvoie

```
rl_desriptor new_rl_descriptor = { .d = newd, .f = lfd.f };
```

6.6 `rl_fork`

```
pid_t rl_fork()
```

La fonction exécute `fork()` ; chez le processus parent elle retourne immédiatement le résultat de `fork()` ; chez l'enfant `rl_fork()` doit copier tous les verrous de son parent avant de retourner. Plus exactement, soit

5. `kill` avec le numéro de signal 0, voir la page `man` de `kill`

```
pid_t parent = getppid();
```

le pid du parent. Supposons qu'un des propriétaires (`owner`) d'un verrou `rl_lock` est le processus parent, c'est-à-dire le tableau `owners` contient le couple

```
{  .proc = parent, .des = i  }
```

Pour ce verrou il faut ajouter un nouveau `owner` :

```
owner new_owner = { .proc = getpid(), .des = i };
```

avec le même descripteur mais avec le pid de l'enfant. Pour parcourir tous les verrous du parent, `rl_fork()` utilise la variable globale `rl_all_files` décrite dans la section 5.5.

6.7 Suppression de shared memory object

Il reste un problème délicat. A quel moment pouvons nous supprimer un shared memory object qui contient la structure `rl_open_file` ?

Il est clair qu'il est inutile de le conserver pour toujours, si le fichier correspondant n'existe plus alors le shared memory object devient inutile et fait juste encombrer la mémoire.
Donc peut-être il faut remplacer `unlink` par une nouvelle nouvelle fonction `rl_unlink` qui supprimera aussi le shared memeory object associé au fichier ?
Mais `unlink` ne supprime pas le fichier mais juste fait diminuer le nombre de liens durs vers le fichier. Et même si le nombre de liens durs devient 0 le système supprime le fichier seulement quand il n'y a plus de descripteurs ouverts sur le fichier. Donc il ne faut pas se précipiter, en particulier ne pas supprimer le shared memeory object s'il y a encore des `rl_descriptor`s qui font la référence vers le `rl_open_file` correspondant.

Peut-être faut-il ajouter la suppression de shared memeory object dans `rl_close` ? Bien sûr encore une fois, tant que il y a des `rl_descriptor`s qui possèdent la référence vers le `rl_open_file` il faut garder le shared memeory object.

Ou peut-être il faut rien faire et le problème se résout soi-même ?
Réfléchissez et, éventuellement, implémenter une solution.

6.8 Initialiser la bibliothèque

Ajouter dans la bibliothèque une fonction

```
int rl_init_library()
```

qui initialise la variable `rl_all_files`.

7 Extensions

Si vous voulez avoir une très bonne note alors vous pouvez (devez ?) implémenter au moins une extension du projet de base. Certaines sont en fait *de facto* nécessaires pour avoir une bibliothèque vraiment utile et utilisable.

– Ajouter dans `rl_fcnl` la commande `F_SETLKW` : un processus qui demande un verrou doit être suspendu (soit sur un mutex soit sur un sémaphore) tant que sa demande ne peut pas être satisfaite. Cela implique bien sûr qu'un processus qui lève un verrou doit « réveiller » un des processus suspendus à cause de ce verrou ; cela implique aussi qu'un processus qui termine (et supprime ses verrous, voir la section précédente) doit réveiller tous les processus suspendus à cause de ces verrous. Cette extension est *prioritaire*.

– Si vous avez implémenté la commande `F_SETLKW` dans `rl_fcntl`, se pose la question des *deadlocks* : supposons qu'un processus p_0 essaie de poser un verrou avec la commande `F_SETLKW` et que cela provoque un cycle de processus $p_0 \to p_1 \to p_2 \ldots\ldots p_n \to p_0$ tel que la demande de verrou sur un segment de fichier par le processus p_i est suspendu à cause du verrou tenu par p_{i+1} ; alors il faut que la demande de p_0 échoue tout de suite avec `errno == EDEADLK`.

– Le projet sous la forme actuelle ne permet pas de traiter convenablement les verrous d'un processus qui exécute *exec*. Les verrous sont bien préservés par *exec* – mais le processus perd la variable `rl_all_files` et tous les objets `rl_descriptor`. Pour que le processus puisse récupérer l'information contenue dans la variable `rl_all_files` après exec *exec* il faudrait stocker le contenu de cette variable, par exemple dans un shared memory object qui pourra être facilement retrouvé après *exec*.