# Programmation système avancée - projet

## 1 Modalités

Le projet doit être réalisé en binôme (éventuellement, en monôme). La date de soutenance sera communiquée ultérieurement. Pendant la soutenance, les membres d'un binôme devront chacun 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 10 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/doku.php/wiki/howto_connect`
`http://www.informatique.univ-paris-diderot.fr/wiki/doku.php/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).
La soutenance se fera à partir du code déposé sur le gitlab et **sur les machines de l'UFR** (salles 2031 et 2032) : au début de la soutenance vous aurez à cloner votre projet à partir du gitlab et le compiler avec `make`.
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 lisent et écrivent des données.

## 2 Objets `mfifo`

Le but du projet est d'implémenter un système de communication entre des processus qui permet deux types de communication :

1. une communication similaire à celle par tube (`pipe` ou `fifo`),
2. et une communication par messages.

Nous appelons `mfifo` l'objet qui permet cette communication, la structure `mfifo` sera décrite en détail dans la section 4.
Les données seront traitées selon l'algorithme `fifo` : les octets sont lus dans un `mfifo` dans l'ordre d'écriture, et la lecture supprime automatiquement les octets lus.
Vous allez implémenter les `mfifo` en utilisant la mémoire partagée,

– soit en projetant en mémoire des *shared memory objects* pour les `mfifo` nommés,
– soit directement par *mmap* shared et anonymous pour les `mfifo` anonymes.

Il est **essentiel** que les opérations à implémenter fonctionnent correctement quand plusieurs processus accèdent en même temps aux objets `mfifo`. **Si ce point n'est pas respecté votre implémentation est erronée.**
Il faudra donc protéger l'accès parallèle à la mémoire de plusieurs processus, soit par des sémaphores POSIX, soit à l'aide de mutexes/conditions.

# 3 Fonctions à implémenter

## 3.1 mfifo_connect() - `mfifo` nommé

```
mfifo *mfifo_connect( const char *nom, int options, mode_t permission,
                      size_t capacite )
```

La fonction `mfifo_connect()` permet de se connecter à un objet `mfifo`, en le créant éventuellement s'il n'existe pas encore ; `nom` est soit le nom de l'objet `mfifo`, soit `NULL` pour la création d'un `mfifo` anonyme, cf. section 3.2 ; `capacite` est la capacité de stockage du `mfifo` – le nombre maximal d'octets que peut contenir l'objet `mfifo`.
La valeur du paramètre `options` est ignorée pour le `mfifo` anonyme.
Pour un `mfifo` nommé, `options` peut prendre les valeurs suivantes :

– 0 pour demander la connexion à un `mfifo` nommé existant,
– `O_CREAT` pour demander la création de `mfifo` s'il n'existe pas, puis la connexion,
– `O_CREAT|O_EXCL` pour indiquer qu'il faut créer un `mfifo` nommé seulement s'il n'existe pas : si `mfifo` existe `mfifo_connect` doit échouer.

Le paramètre `permission` est ignoré sauf à la création du *shared memory object* associé à un `mfifo` nommé (ouvert en mode `O_CREAT` et indique alors les permissions d'accès attribuées à cet objet (troisième paramètre de `shm_open`).
La fonction `mfifo_connect()` retourne un pointeur vers un objet `mfifo` qui sera utilisé par d'autres opérations, voir section 4 pour la description du type `mfifo`.
En cas d'échec, `mfifo_connect()` retourne `NULL`.

## 3.2 `mfifo` anonyme

Un `mfifo` anonyme est créé par un appel à `mfifo_connect()` dont le premier paramètre est `NULL`. Chaque `mfifo_connect(NULL,...)` crée un nouvel objet `mfifo` anonyme.
Un `mfifo` anonyme est partagé uniquement par le processus qui a effectué le `mfifo_connect()` et ses descendants.

## 3.3 mfifo_disconnect()

```
int mfifo_disconnect(mfifo *fifo)
```

déconnecte le processus de l'objet `mfifo` (l'objet n'est pas détruit, mais `fifo` devient inutilisable).
La fonction `mfifo_disconnect()` retourne 0 si OK et $-1$ en cas d'erreur.

## 3.4 mfifo_unlink()

```
int mfifo_unlink(const char *nom)
```

supprime l'objet `mfifo` nommé ; attention, la suppression ne sera effective que lorsque tous les processus connectés seront déconnectés de l'objet ; en revanche toute nouvelle tentative de `mfifo_connect()` doit échouer.
Valeur de retour : 0 si OK, $-1$ si échec (`mfifo_unlink()` est juste une petite fonction qui fait appel à `shm_unlink()` pour supprimer l'objet mémoire implémentant le `mfifo`).

## 3.5 Écriture dans un `mfifo`

```
int mfifo_write(mfifo *fifo, const void *buf, size_t len)
int mfifo_trywrite(mfifo *fifo, const void *buf, size_t len)
int mfifo_write_partial(mfifo *fifo, const void *buf, size_t len)
```

Ces fonctions écrivent dans `fifo` les `len` octets situés à l'adresse `buf`.
Si `len` est plus grand que `capacite` (voir Section 3.1), les fonctions `mfifo_write()` et `mfifo_trywrite()` retournent immédiatement $-1$ et mettent `errno` à la valeur `EMSGSIZE` (définie dans `errno.h` comme tous les codes d'erreur).
La fonction `mfifo_write_partial()` en revanche accepte un paramètre `len` supérieur à `capacite`.
`mfifo_write()` bloque le processus appelant jusqu'à ce que `len` octets soient écrits dans `fifo`. Les octets écrits ne doivent pas être mélangés avec les octets écrits par d'autres processus, et le processus appelant reste donc bloqué tant qu'il n'y a pas de place pour `len` octets dans `fifo`.
`mfifo_trywrite()` est une version non-bloquante de `mfifo_write()` ; s'il n'y a pas assez de place pour écrire `len` octets, `mfifo_trywrite()` retourne immédiatement $-1$ et met `errno` à la valeur `EAGAIN`.
`mfifo_write_partial()` écrit `len` octets dans `fifo` mais pas forcément de façon continue, et des octets écrits par d'autres processus peuvent s'intercaler entre ces `len` octets. Par exemple, pour écrire 1000 octets, le processus qui obtient l'accès en écriture sur un `mfifo` avec seulement 100 octets disponibles écrira 100 octets puis se mettra en attente pour écrire les 900 octets qui restent à écrire. Pendant qu'il est en attente d'un nouvel accès au `mfifo`, d'autres processus peuvent obtenir l'accès et écrire.
On préfèrera des implémentations de `mfifo_write()` et `mfifo_write_partial()` qui ne laissent aucun processus en attente infinie : si des processus lecteurs lisent dans le `mfifo`, alors chaque processus qui essaie d'écrire aura à un moment la possibilité de le faire.
Les trois fonctions retournent 0 si OK et $-1$ en cas d'erreur.
**Exemple.** Pour stocker une valeur `int` dans un `mfifo`, on pourra faire

```
int i = 6;
int n = mfifo_write(f, &i, sizeof(i));
```

## 3.6 Lecture dans un `mfifo`

On distingue deux types de lecture : lecture simple et lecture avec verrou.

### 3.6.1 Lecture simple

L'opération de lecture simple est effectuée par :

```
ssize_t mfifo_read(mfifo *fifo, void *buf, size_t len)
```

Supposons qu'un `mfifo` contient `n` octets et `l = min(len, n)`. La fonction `mfifo_read()` copie `l` octets de `mfifo` à l'adresse `buf`. Les octets copiés sont supprimés du `mfifo` et `mfifo_read` retourne le nombre d'octets copiés.
S'il y plusieurs processus lecteurs qui tentent de lire en même temps, tous sauf un seront bloqués en attendant la fin de la lecture du seul processus autorisé à lire. Chaque lecture lit donc un segment contigu d'octets stockés dans le `mfifo`.

### 3.6.2 Verrou de lecture

Il s'avère qu'une protection plus stricte est parfois nécessaire, pour regrouper plusieurs opérations de lecture en une seule opération indivisible. Nous obtenons cet effet par le verrou de lecture.
Examinons la situation où plusieurs processus écrivains et plusieurs processus lecteurs opèrent sur le même objet `mfifo`.
Supposons que les écrivains écrivent des messages de longueurs variables et que les longueurs des messages ne sont pas connues par les lecteurs. Mais pour qu'un processus lecteur puisse lire un message, il doit fournir sa longueur quand il exécute `mfifo_read()`. L'idée est d'écrire un message composé d'un entier `l`, qui donne sa longueur, suivi par le contenu de `l` octets. Il est alors important d'écrire la valeur de `l` et le contenu de message avec un seul appel à `mfifo_write()`, pour éviter qu'un autre écrivain intercale du contenu entre `l` et le message.
Pour écrire un message de longueur variable avec sa longueur on peut procéder de façon suivante : on définit

```
typedef struct{
   size_t l;
   char mes[];
} message;
```

où `l` est le nombre d'éléments dans le tableau `mes`. Puis pour envoyer le message « Bonjour » on pourra maintenant procéder de la façon suivante :

```
char *s="Bonjour";
size_t lon = strlen(s);

//long_tot : longueur totale, +1 pour le caractere nul à la fin du message
size_t long_tot = sizeof(message) + lon + 1;

message *m = malloc(long_tot);
m->l = lon + 1;
//copier le message, y compris le caractère nul dans le tableau mes[]
memmove(m->mes, s, m->l) ;

//ecrire la structure avec le message dans mfifo f
mfifo_write(f, m, long_tot);
```

**Rappel : structure avec un tableau sans élément** La structure `message` dans l'exemple ci-dessus contient deux champs : le champ `l` et le tableau `mes` de taille 0 (sans élément). En particulier `sizeof(message)` est la taille de la structure mais **sans compter** la mémoire pour le tableau `mes`.
Pour allouer la mémoire pour la structure `message` avec un tableau `mes` de `x` octets, il faut faire

```
message *m = malloc( sizeof(message) + x );
```

Intuitivement, le résultat de ce `malloc` est un pointeur `m` vers une structure composée des champs :

```
size_t l;
char m[x];
```

**Fin du rappel**

Supposons maintenant qu'un processus $P_1$ souhaite lire le message envoyé dans le `mfifo` dans l'exemple précédent. Il peut le faire avec deux appels à `mfifo_read()` :

– le premier pour lire la valeur de `l` :

```
message ms;
mfifo_read(f, &ms, sizeof(message));
```

– et le deuxième pour lire les `l` octets du message lui-même :

```
char *t = malloc( ms.l );
mfifo_read(f, t, ms.l);
```

Une autre possibilité, aussi avec deux appels à `mfifo_read` :

```
message *ps  = malloc( sizeof( message) );
mfifo_read(f, ps, sizeof(message)); /*  lire sans tableau mes[] */

ps = realloc( ps, ps->l + sizeof(message) ); /* agrandir la mémoire pour le
    tableau */
mfifo_read(f, ps->mes , ps->l );   /* lire le message lui même */
```

Mais si $P_1$ est en concurrence avec d'autres processus lecteurs, un processus $P_2$ pourrait obtenir l'accès en lecture après la lecture de `l` par $P_1$, et lire les caractères appartenant au message. Quand $P_1$ obtiendra à nouveau l'accès en lecture, il ne trouvera plus le message recherché dans le `mfifo`.
La lecture d'un message nécessite donc un mécanisme permettant à un processus

(A) de verrouiller le `mfifo` pour la lecture,

(B) lire dans le `mfifo` avec plusieurs appels à `mfifo_read()`,

(C) déverrouiller l'accès au `mfifo` en lecture.

Pour implémenter (A) et (C) vous devez implémenter les opérations

```
int mfifo_lock(mfifo *fifo)
int mfifo_unlock(mfifo *fifo)
```

On dit qu'un processus possède le verrou s'il a réussi à faire `mfifo_lock()` et qu'il n'a pas encore exécuté `mfifo_unlock()`. L'appel à `mfifo_lock()` est bloquant jusqu'à ce que le processus appelant obtienne le verrou. À tout instant, au plus un processus possède un verrou sur un objet `mfifo` donné. ***Quand un processus possède ce verrou, l'appel `mfifo_read()` bloque pour tout autre processus.***
Les fonctions `mfifo_lock()` et `mfifo_unlock()` renvoient 0 en cas de succès et $-1$ en cas d'échec.
Une fois le verrou obtenu, le processus peut exécuter une suite d'opérations de lecture

```
ssize_t mfifo_read(mfifo *fifo, void *buf, size_t len)
```

Cette suite d'opérations de lecture est alors indivisible : tant qu'un processus détient le verrou, aucun autre processus ne peut lire le contenu du `mfifo`.
Cependant, la possession du verrou modifie fortement le comportement de `mfifo_read()` tel que décrit à la section 3.6, effectué sans obtention de verrou : l'appel à `mfifo_read()` exécuté après avoir posé le verrou ***réussit seulement si le `mfifo` contient au moins `len` caractères***. Sinon, `mfifo_read()` échoue immédiatement et renvoie $-1$. Dans ce cas rien n'est copié à l'adresse `buf`.
Enfin, vous devez également implémenter la version non bloquante

```
int mfifo_trylock(mfifo *fifo)
```

qui retourne immédiatement 0 si le verrou est obtenu, et -1 si la prise du verrou est impossible, soit parce qu'un autre processus possède le verrou, soit parce qu'un autre processus effectue une opération d'écriture. La variable `errno` prend alors la valeur `EAGAIN`.

## 3.7 État d'un `mfifo`

```
size_t mfifo_capacity(mfifo *fifo)
size_t mfifo_free_memory(mfifo *fifo)
```

retournent respectivement la capacité totale du `mfifo` et le nombre d'octets actuellement disponibles (c'est-à-dire le nombre maximal d'octets que le prochain `mfifo_write()` pourra y écrire).

## 3.8 `mfifo` et `exec`

Un processus qui fait appel à une fonction de la famille `exec` perd les références vers tous les objets `mfifo` (anonymes ou non). Cela pose un grand problème quand ce processus possède les verrous sur des objets `mfifo`.
Hélas, il semble qu'il n'y ait aucun mécanisme simple qui permette d'appeler **automatiquement** [1] `mfifo_unlock` sur tous les objets `mfifo` quand un processus fait appel à `exec`.
Il est possible de le faire de façon compliquée ; si quelqu'un découvre comment cela mérite sans doute de points supplémentaires. Avertissement : les informations données dans ce cours ne suffisent pas.
Pour éviter les problèmes, on suppose donc qu'un processus qui fait appel à `exec` ne doit pas détenir de verrous sur des objets `mfifo`.

1. automatiquement donc de façon transparente pour celui qui utilise `mfifo`

Comme exercice facultatif (mais qui apporte des points si vous le faites) vous pouvez implémenter la fonction suivante :

```
int mfifo_unlock_all(void)
```

qui applique `mfifo_unlock()` à tous les verrous posés par le processus. La seule difficulté (de programmation C et non pas de systèmes) est de concevoir comment le processus peut stocker tous les objets `mfifo` pour lesquels il détient un verrou et que ce soit transparent (vous n'avez pas de droit de modifier les signatures de fonctions que vous devez implémenter). Maintenant il suffit d'indiquer que chaque processus qui utilise un `mfifo` doit faire appel à `mfifo_unlock_all()` juste avant `exec...`.

### 3.9 Gestion des accès parallèles

Les sémaphores ou variables mutex permettent de gérer les accès parallèles de plusieurs processus à un objet `mfifo`. Une solution triviale aux problèmes posés par la concurrence est bien sûr de bloquer l'accès au `mfifo` à tous les autres processus pour chaque opération. Mais c'est ennuyeux comme solution : si un `mfifo` n'est ni vide ni plein, un processus qui écrit et un processus qui lit n'utilisent pas la même partie de la mémoire partagée. Donc, en principe, pas de raison pour qu'un lecteur bloque un écrivain.

## 4 La structure de données `mfifo`

Nous présentons une proposition de structure de données, vous n'êtes pas obligés de la suivre à la lettre. Le type `mfifo` est une structure qui contient plusieurs champs :

```
typedef struct{
     size_t capacity;
     size_t debut;
     size_t fin;
     pid_t pid;
     /* semaphores , mutexes, conditions */
     char memory[];
}mfifo;
```

Le dernier champ de la structure est un tableau `char memory[]` qui contient en fait `capacite` octets [2]. La partie de la structure `mfifo` mise en commentaires contient tous les sémaphores, mutexes, et conditions nécessaires pour assurer l'exclusion mutuelle [3].
Le champ `pid` contient l'identifiant de l'unique processus qui possède le verrou sur le `mfifo`, ou $-1$ si aucun processus n'a de verrou.
Les champs `debut` et `fin` sont des indices dans le tableau `memory`.

Le plus satisfaisant est de voir `memory` comme un tableau circulaire entre les indices 0 et `capacite-1` ; l'indice suivant `i` est `(i+1) % capacite` ; `debut` est l'indice de la première

2. J'écris que `memory` est un tableau d'octets parce qu'on peut y stocker des données autres que des caractères ascii : `memmove` permet d'y copier tout type de données.

3. Comme expliqué dans un exemple dans la section 3.6.2 qui utilise aussi une structure avec un tableau sans éléments comme le dernier champ, la taille de la mémoire nécessaire pour implémenter `mfifo` avec `capacite` de mémoire pour les données est de `sizeof( mfifo ) + capacite`.

case occupée (contenant le premier élément lu par `mfifo_read()`), et `fin` est l'indice de la première case libre (la première utilisée par `mfifo_write()` pour placer un nouvel octet). Dans le tableau circulaire,

- soit `debut < fin`, et les octets de `memory` occupés par les données sont dans les cases $debut, debut + 1, \ldots, fin - 1$.
- soit `fin < debut`, et les octets occupés par les données sont dans les cases $debut$, $debut + 1, \ldots, capacite - 1, 0, 1, \ldots, fin - 1$.

Si `debut == fin`, alors le tableau `memory` est vide. Si `(fin+1)%capacite == debut`, alors `memory` est plein.
Dans le tableau `memory`, l'octet d'indice `fin` ne contient jamais de données, en particulier il y a un octet perdu quand `memory` est plein et `memory` permet de stocker en pratique au plus `capacite-1` octets.
Ce n'est pas la seule possibilité pour implémenter le tableau circulaire, faites ce qui vous convient.

Vous pouvez aussi utiliser un tableau non circulaire tel que on ait toujours `debut <= fin`. Les éléments qui contiennent les données sont alors dans les cases $debut, debut + 1, \ldots, fin - 1$. Quand on place un nouvel octet à la case d'indice `capacite-1` alors que `debut > 0`, il faudra décaler tous les octets pour que le premier octet de données se retrouve à l'indice 0 etc.

# 5 Organisation du code

Toutes les fonctions demandées doivent être regroupées dans un fichier `mfile.c` accompagné de `mfile.h` de telle sorte que chaque programme qui utilise les objets `mfifo` puisse faire juste un include de `mfile.h` pour être compilé (l'édition de liens ajoutera `mfile.o` obtenu par la compilation de `mfile.c`, mais c'est une autre histoire). Il est possible que pour factoriser le code vous ayez besoin de fonctions auxiliaires, elles doivent être déclarées `static` dans `mfile.c` pour qu'elle soit invisibles à l'utilisateur.
Vous devez tester toutes les fonctions que vous écrivez. Fournir les fonctions demandées sans une batterie de tests est pratiquement inutile. Nous aurons beaucoup beaucoup de mal à croire que vos fonctions exécutent les tâches demandées si vous êtes incapable de le prouver. À la rigueur, il vaut mieux implémenter juste certaines opérations de lecture/écriture et faire des tests convainquants et complets qu'« implémenter » toutes les fonctions sans ou avec très peu de tests.
Les programmes de test doivent être dans des fichiers séparés (pas de `main` dans `mfile.c`). Les noms de fichiers `mfile.c` et `mfile.h` ne sont pas modifiables.

# 6 Extensions

## 6.1 Notifications

Nous proposons une extension de projet de base (pour ceux qui ont implémenté le projet principal et ont testé les fonctions).
Un processus peut s'enregistrer sur un `mfifo` pour recevoir un signal quand le `mfifo` reçoit des données.

Quand le processus s'enregistre, il doit indiquer quel signal il veut recevoir. Un seul processus peut être enregistré à un moment donné, une tentative d'enregistrement d'un autre processus doit échouer.
Seul le processus enregistré peut annuler son enregistrement, ce qui permet à un autre processus de s'enregistrer.
Le signal est envoyé uniquement quand les conditions suivantes sont satisfaites :

- les octets sont écrits dans le `mfifo` vide et
- il n'y a aucun processus suspendu en attente sur une opération de lecture.

Quand le signal de notification est envoyé, le processus enregistré doit être automatiquement désenregistré.

## 6.2 Disparition d'un processus qui détient un verrou

Supposons qu'un processus qui a posé un ou plusieurs verrous avec `mfifo_lock` termine (soit en faisant /exit/ soit tué par un signal) sans avoir levé ces verrous.
Il reste toujours enregistré comme le propriétaire de verrou dans le champ `pid` de `mfifo` ce qui empêche à tout autre processus d'effectuer la lecture de `mfifo`. Pour palier à ce problème on pourra modifier les fonctions de lecture et la fonction `mfifo_lock` de telle sorte que quand le champ `pid` de `mfifo` contient un `pid` d'un processus qui n'existe pas alors la fonction pourra réinitialiser ce champ à $-1$.
(Bien sûr faudra s'assurer de l'exclusion mutuelle lors de l'accès à ce champ.)
Pour vérifier si le processus `pid` existe on pourra faire appel

```
pid_t p = getpgid( pid );
if( p == (pid_t) -1 && errno == ESRCH ){
   /* le processus pid n'existe pas */
}
```