Programmation système avancée - projet

Modalités

Le projet doit être réalisé en binôme (éventuellement, en monôme). Les soutenances auront lieu en mai, la date précise 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

`http://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 du cours de Systèmes avancés. Le dépôt devra contenir un fichier « `README` » donnant la liste des membres de l'équipe (nom, prénom, numéro étudiant). Vous êtes censés utiliser gitlab de manière régulière pour votre développement. Le dépôt doit être créé **avant le 15 avril au plus tard**.
Le projet doit être accompagné d'un `Makefile` utilisable. Les fichiers doivent être compilés avec les options `-Wall -g -pedantic` sans donner lieu à aucun avertissement (et sans erreurs bien évidemment).
La soutenance se fera à partir du code déposé sur 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 de 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 envoient et réceptionnent les messages.

Première partie

Sujet de base

1 Files de messages

Le but du projet est d'implémenter les files de messages pour une communication entre processus tournant sur la même machine (pas de communication par réseau). Vous devez faire une implémentation en utilisant la mémoire partagée entre les processus, en particulier une solution qui implémente les files de messages à l'aide de files de messages existantes (POSIX ou System V) ou des sockets UNIX **ne sera pas acceptée**.
L'accès parallèle à la file de messages doit être possible avec une protection appropriée, soit avec des sémaphores POSIX, soit avec des mutexes/conditions.

1.1 Fonctions à implémenter

1.1.1 `m_connexion`

```
MESSAGE *m_connexion( const char *nom, int options,
                      size_t nb_msg, size_t len_max, mode_t mode )
```

`m_connexion()` permet soit de se connecter à une file de message existante, soit de créer une nouvelle file de messages et s'y connecter.
`nom` est le nom de la file ou NULL pour une file anonyme, cf. section 3.
`nb_msg` est la *capacité* de la file, c'est à dire le nombre (minimal) de messages qu'on peut stocker avant que la file soit pleine (si on stocke les messages de façon compacte, on pourra stocker plus que `nb_msg` messages, cf. section 5.1 ; mais vous devez garantir que la file pourra stocker au moins `nb_msg`).
`len_max` est la longueur maximale d'un message.
`mode` donne les permissions accordées pour la nouvelle file de messages (« OR » bit-à-bit des constantes définies pour `chmod`, cf `man 2 chmod`).
`options` est un « OR » bit-à-bit de constantes suivantes :

- `O_RDWR`, `O_RDONLY`, `O_WRONLY`, avec la même signification que pour `open` ; exactement une de ces constantes doit être spécifiée ;
- `O_CREAT` pour demander la création de la file ;
- `O_EXCL`, en combinaison avec `O_CREAT`, indique qu'il faut créer la file seulement si elle n'existe pas ; si la file existe déjà, `m_connexion` doit échouer.

Si `options` ne contient pas `O_CREAT`, alors la fonction `m_connexion` n'aura que les deux paramètres `nom` et `options`, autrement dit `m_connexion` est une fonction à nombre variable d'arguments (soit 2, soit 5).
`m_connexion` retourne un pointeur vers un objet de type `MESSAGE` qui identifie la file de messages et sera utilisé par d'autres fonctions, voir section 2 pour la description du type `MESSAGE`.
En cas d'échec, `m_connexion` retourne `NULL`.

1.1.2 `m_deconnexion`

```
int m_deconnexion(MESSAGE *file)
```

déconnecte le processus de la file de messages ; celle-ci n'est pas détruite, mais `file` devient inutilisable.
`m_deconnexion` retourne 0 si OK et -1 en cas d'erreur.

1.1.3 `m_destruction`

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

demande la suppression de la file de messages ; la suppression effective n'a lieu que quand le dernier processus connecté se déconnecte de la file ; en revanche, une fois `m_destruction` exécutée par un processus, toutes les tentatives de `m_connexion` ultérieures échouent.
Valeur de retour : 0 si OK, -1 si échec.

1.1.4 m_envoi

Pour envoyer un message, on utilisera la structure

```
struct mon_message{
  long type;
  char mtext[];
};
```

Le champ `type` contient le type du message (nous allons y revenir plus tard). Le champ `mtext` contient le message lui-même ; il ne s'agit pas forcément d'un tableau de caractères, c'est juste une façon d'indiquer que le message commence juste après le champ `type`.

```
int m_envoi(MESSAGE *file, const void *msg, size_t len, int msgflag)
```

`m_envoi` envoie le message dans la file ; `msg` est un pointeur vers le message à envoyer, et `len` la longueur du message en octets (la longueur du champ `mtext` de `struct mon_message`.
Le paramètre `msgflag` peut prendre deux valeurs :

- 0 : le processus appelant est bloqué jusqu'à ce que le message soit envoyé ; cela peut arriver quand il n'y a plus de place dans la file ;
- `O_NONBLOCK` : s'il n'y a pas de place dans la file, alors l'appel retourne tout de suite avec la valeur de retour -1 et `errno` doit prendre la valeur `EAGAIN`.

La fonction retourne 0 quand l'envoi réussit, -1 sinon.
Si la longueur du message est plus grande que la longueur maximale supportée par la file, la fonction retourne immédiatement -1 et met `EMSGSIZE` dans `errno`.
Exemple : pour envoyer un message constitué de deux valeurs `int`, on procède de la façon suivante :

```
int t[2] = {-12, 99}; /*valeurs à envoyer*/

struct mon_message *m = malloc( sizeof( struct mon_message ) + sizeof( t ) );
if( m == NULL ){  /* traiter erreur de malloc */ }

m->type = (long) getpid(); /* comme type de message, on choisit l'identité
                            * de l'expéditeur */
memmove( m -> mtext, t, sizeof( t )) ; /* copier les deux int à envoyer */

/* envoyer en mode non bloquant */
int i = m_envoi( file, m, sizeof( t ), O_NONBLOCK) ;

if( i == 0 ){ /* message envoyé */ }
else if( i == -1 && errno = EAGAIN ){ /* file pleine, essayer plus tard */}
else{  /* erreur de la fonction */ }
```

1.1.5 m_reception

```
ssize_t m_reception(MESSAGE *file, void *msg, size_t len, long type, int flags)
```

Le paramètre `msg` est l'adresse à laquelle la fonction doit copier le message lu ; ce message sera supprimé de la file.

Le paramètre `len` est la longueur (en octets) de mémoire à l'adresse `msg` ; si `len` est inférieur à la longueur du message à lire, `m_reception` échoue et retourne -1 et `errno` prend la valeur `EMSGSIZE`.
Le paramètre `type` précise la demande :

- si `type == 0` : lire le premier message dans la file ;
- si `type > 0` : lire le premier message dont le type est `type` ; grâce au `type`, plusieurs processus peuvent utiliser la même file de messages, par exemple chaque processus peut utiliser son `pid` comme valeur de `type` ;
- si `type < 0` : traiter la file comme une file de priorité ; en l'occurrence, cela signifie : lire le premier message dont le type est inférieur ou égal à la valeur absolue de `type`.

Le paramètre `flags` peut prendre soit la valeur 0, soit `O_NONBLOCK` ; si `flags == 0`, l'appel est bloquant jusqu'à ce que la lecture réussisse ; si `flags == O_NONBLOCK`, et s'il n'y a pas de message du type demandé dans la file, l'appel retourne tout de suite avec la valeur -1 et `errno==EAGAIN`.
`m_reception` lit le premier message convenable sur la file, le copie à l'adresse `msg` et le supprime de la file.
Une file de messages est une file FIFO : si on écrit les messages a, b, c dans cet ordre, et si on lit avec `type==0`, alors les lectures successives retournent a, b, c dans le même ordre.
La fonction retourne le nombre d'octets du message lu, ou 1 en cas d'échec.

1.1.6 L'état de la file

```
size_t m_message_len(MESSAGE *)
size_t m_capacite(MESSAGE *)
size_t m_nb(MESSAGE *)
```

Ces fonctions retournent respectivement la taille maximale d'un message, le nombre maximal de messages dans la file, et le nombre de messages actuellement dans la file.

2 Structures de données

2.1 MESSAGE

Le type `MESSAGE` est une structure qui contient au moins les éléments suivants :

1. le type d'ouverture de la file de messages (lecture, écriture, lecture et écriture) ; cette information est nécessaire pour les opérations `m_envoi` et `m_reception` ; il faut vérifier que l'opération est autorisée, par exemple `m_envoi` doit échouer si la file a été ouverte seulement en lecture ;
2. le pointeur vers la mémoire partagée qui contient la file.

Si vous trouvez d'autres informations pertinentes à stocker dans la structure `MESSAGE`, n'hésitez pas à les ajouter.

2.2 Organisation de la mémoire partagée

Une file de message contient deux sortes d'informations.

Au début, on mettra une structure en-tête qui contient des informations générales sur l'état de la file de messages, cf. section 2.2.2.
Cet en-tête est suivi du tableau de messages.
La structure en-tête et le tableau de messages peuvent faire partie d'une seule structure, c'est à vous de fixer les détails de l'implémentation.
Commençons par le tableau de messages.

2.2.1 Tableau de messages

Le plus simple est de stocker les messages dans un tableau, de préférence un tableau circulaire ; dans ce cas il suffit de maintenir dans l'en-tête deux indices `first` et `last` : `first` est l'indice du premier élément de la file, celui qui sera lu par `m_reception` ; `last` est l'indice de premier élément libre de tableau, celui que `m_envoi` utilisera pour placer le nouveau message.
Dans le tableau circulaire

- soit `first < last` et les éléments de la file occupent les cases `first, first+1, ..., last-1`,
- soit `last <= first` et la file occupe les cases `first, first+1, ..., n-1, 0, 1, ..., last-1` où `n` est le nombre d'éléments dans le tableau.

Donc l'arithmétique des indices se fera modulo `n`.
En particulier si `first == last`, alors le tableau est plein.
Pour distinguer un tableau plein d'un tableau vide, on pourra supposer que `first == -1` si le tableau est vide.
Ce n'est pas la seule possibilité pour implémenter un tableau circulaire, faites ce qui vous convient.

Vous pouvez aussi utiliser un tableau non circulaire tel qu'on ait toujours `first <= last` ; les éléments de la file sont dans les cases `first, first+1, ..., last-1` ; dans ce cas, quand on cherche à placer un nouvel élément à la case de l'indice `n-1` (`n` le nombre d'éléments de tableau) alors que `first > 0`, alors il faudra faire un décalage de tous les éléments pour que le premier élément de la file se retrouve à l'indice 0 etc.

Dans chaque message, il faut stocker sa longueur ; chaque message est donc une structure qui contient au moins le nombre d'octets dans le message (nécessaire pour la valeur de retour de `m_reception`), le type de message et le message lui-même.

2.2.2 En-tête

Dans l'en-tête de la file de message, on stockera au moins les informations suivantes :

(1) la longueur maximale d'un message,

(2) la capacité de la file (le nombre minimal de messages que la file peut stocker),

(3) les deux indices `first` et `last` dans le tableau de messages,

(4) les sémaphores ou variables mutex/conditions nécessaires,

(5) tout autre information utile pour votre implémentation et qui m'échappe maintenant...

3 Files anonymes

Une file anonyme est créée par un appel à `m_connexion` dont le premier paramètre est NULL. Chaque `m_connexion(NULL,...)` ouvre une nouvelle file anonyme, qui ne peut être partagée que par héritage.
Le paramètre `options` est ignoré pour une file anonyme, qui sera toujours ouverte à la fois en lecture et écriture.
Comme `m_connexion(NULL,...)` ouvre chaque fois une nouvelle file anonyme, les paramètres `nb_msg` et `len_max` sont toujours obligatoires pour `m_connexion(NULL,...)`.
Les files anonymes ne supportent pas `m_destruction` ; une telle file disparaît automatiquement quand le dernier processus connecté se déconnecte avec `m_deconnexion`.
(Évidemment une file anonyme sera implémentée par `mmap` anonyme)

4 Gestion des accès parallèles

Les sémaphores ou variables mutex permettent de gérer les accès parallèles de plusieurs processus à une file de messages. Une solution triviale est bien sûr de bloquer l'accès à la file à tous les autres processus pour chaque opération. Mais cette solution est franchement sans intérêt.
Si la file n'est ni vide ni pleine, un processus qui ajoute un message et un processus qui lit un message n'utilisent pas la même partie de la mémoire partagée qui contient la file. Donc, en principe, il n'y a pas de raison qu'un écrivain bloque un lecteur et vice-versa.
Si plusieurs processus ajoutent des messages dans la file, dès que le premier processus a changé la valeur de `last` pour réserver la place pour son message, il est inutile de faire poireauter le suivant qui peut déjà commencer à exécuter `m_envoi()` en même temps que le premier copie le message dans la mémoire.
Une remarque similaire s'applique à des processus qui lisent les messages.

5 Organisation du code

Toutes les fonctions demandées doivent être regroupées dans un fichier `m_file.c` accompagné de `m_file.h` de telle sorte que chaque programme qui utilise les files de messages puisse faire juste un `#include "m_file.h"` pour être compilé (l'édition de liens ajoutera `m_file.o` obtenu par la compilation de `m_file.c` mais c'est une autre histoire).
Les programmes de test doivent être dans des fichiers séparés.
Les noms de fichiers `m_file.c` et `m_file.h` ne doivent pas être modifiés (si nous nous décidons à écrire nos propres programmes pour tester vos fonctions, alors il faut que `#include "m_file.h"` marche dans nos programmes).

Deuxième partie
Extensions

Le sujet de base est simple. Bien fait, il ne rapportera guère plus que la moyenne. Pour escompter une note plus généreuse, on vous propose des extensions. Faire au moins une des extensions améliorera sensiblement la note.

5.1 Compacter le tableau de messages

Si on déclare que chaque message contient au maximum 200 octets mais qu'en réalité les messages qui arrivent contiennent en moyenne 10 octets, il y aura un gaspillage de place lors du stockage de chaque message dans un tableau. Pour éviter le gaspillage, il suffit de stocker les messages l'un après l'autre sans « trou » entre eux mais en utilisant juste la quantité d'octets nécessaire. Comme chaque message est stocké avec sa longueur ce n'est pas un problème. Juste une modification : `first` et `last` ne sont plus les indices dans le tableau de messages, mais les adresses relatives du premier octet du premier message et du premier octet libre.
« Adresses relatives » signifie que ce ne sont pas des pointeurs – les adresses virtuelles après chaque `mmap` sont différentes même si `mmap` est effectué sur le même objet mémoire. Il faut donc stocker dans `first` le *décalage* entre l'adresse du premier octet du premier message et l'adresse du début de la mémoire partagée. La même remarque s'applique à `last`.

5.2 Notifications

Un processus peut s'enregistrer sur la file de messages pour recevoir un signal quand un message arrive dans la file. Donc au lieu d'appeler `m_reception`, un processus peut s'enregistrer et poursuivre son exécution.
Quand le processus s'enregistre, il doit indiquer quel signal il veut recevoir. Le nombre de processus qui peuvent être enregistrés en même temps est limité. Quand le processus s'enregistre, il doit indiquer le type de message qu'il attend.
Seul le processus enregistré peut annuler son propre enregistrement.
Le signal est envoyé uniquement quand les deux conditions suivantes sont satisfaites :

- un message du type demandé arrive dans la file ;
- il n'y a aucun processus suspendu en attente de ce message.

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