1 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://moule.informatique.univ-paris-diderot.fr:8080/`

dès le début de la phase de codage et y donner accès en tant que Reporter à tous les enseignants de cours de Systèmes avancés : Wieslaw Zielonka, Benjamin Bergougnoux et Ilias Garnier. 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 **avant le 10 avril au plus tard**. Au moment de la création de dépôt vous devez envoyer un mail à `zielonka@irif.fr` avec la composition de votre équipe (l'objet du mail doit être « projet sys2 », 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 (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

2 Files de messages

Le but du projet est d'implémenter les files de messages pour une communications entre les processus 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) **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 les mutexes/conditions.

2.1 Fonctions à implémenter

2.1.1 msg_connect

```
MESSAGE *msg_connect( const char *nom, int options,
                      size_t nb_msg, size_t len_max )
```

`msg_connect()` permet soit de se connecter à une file de message existante soit créer et se connecter à une nouvelle file de messages.

`nom` est le nom de la file ou NULL pour une file anonyme, cf. section 4.

`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 devienne pleine. (Si on stocke les messages de façon compacte on pourra stocker plus que `nb_msg` messages, section 6.2, vous devez garantir que la file pourra stocker au moins `nb_messages`.)

`len_max` est la longueur maximale d'un message.

`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` avec `O_CREAT` indique qu'il faut créer la file seulement si elle n'existe pas, si la file existe `msg_connect` doit échouer.

Si `options` ne contient pas `O_CREAT` alors la fonction `msg_connect` n'aura que deux paramètres `nom` et `options`, `msg_connect` sera implémenté comme une fonction à nombre variable d'arguments.

`msg_connect` retourne un pointeur vers un objet `MESSAGE` qui identifie la file de messages et sera utilisé par d'autres fonctions, voir section 3 pour la description de type `MESSAGE`.

En cas d'échec `msg_connect` retourne `NULL`.

2.1.2 msg_disconnect

```
int msg_disconnect(MESSAGE *file)
```

déconnecte le processus de la file de messages (la file n'est pas détruite, mais `MESSAGE *` devient inutilisable).

`msg_disconnect` retourne 0 si OK et -1 en cas d'erreur.

2.1.3 msg_unlink

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

supprime la file de messages. La suppression a lieu quand le dernier processus se déconnecte de la file. Par contre une fois `msg_unlink` exécutée par un processus, toutes les tentatives de `msg_connect` doivent échouer.

Valeur de retour : 0 si OK, -1 si échec. `msg_unlink` est juste une petite fonction qui fait appel à `shm_unlink` pour supprimer l'objet mémoire implémentant la file de messages.

2.1.4 msg_send et msg_trysend

```
int msg_send(MESSAGE *file, const char *msg, size_t len)
int msg_trysend(MESSAGE *file, const char *msg, size_t len)
```

`msg_send` envoie le message dans la file. `msg` est le pointeur vers le message à envoyer et `len` la longueur du message.

`msg_send` bloque le processus appelant jusqu'à ce que le message soit envoyé. Il y a essentiellement deux situations dans lesquelles le processus appelant peut être bloqué :

(1) la file est pleine,

(2) quand plusieurs processus envoient des messages seulement un seul peut effectivement mettre le message dans la file, les autres doivent attendre (comportement à implémenter avec sémaphores ou mutexes/conditions pour garantir l'intégrité de données).

`msg_trysend()` est une version non bloquante de `msg_send`. Si la file est pleine `msg_trysend` retourne immédiatement -1 et met la valeur `EAGAIN` dans `errno`.

On vous laisse choisir ce que doit faire `msg_trysend` lorsque la file n'est pas vide mais qu'il y a juste au même moment un autre processus qui envoie un message (e.g. bloquage, bloquage mais pendant un temps limité ou bien retour immédiat de -1).

Si la longueur du message est plus grande que la longueur maximale supportée par la file les deux fonctions retournent immédiatement -1 et mettent `EMSGSIZE` dans `errno`.

En général, les deux fonctions retournent 0 si OK et -1 en cas d'erreur.

2.1.5 msg_receive et msg_tryreceive

```
ssize_t msg_receive(MESSAGE *file, char *msg, size_t len)
ssize_t msg_tryreceive(MESSAGE *file, char *msg, size_t len)
```

`msg_receive` lit le premier message sur la file, le met à l'adresse `msg` et le supprime de la file.

La file de messages est une file FIFO, si on écrit les messages a,b,c dans cette ordre alors les lectures successives retournent a,b,c dans le même ordre.

`len` est utilisé pour indiquer la longueur du tampon `msg`. Si cette longueur est inférieure à la taille maximale d'un message les deux fonctions retournent -1 et mettent `EMSGSIZE` dans `errno`.

`msg_receive` bloque le processus appelant si la file est vide ou quand il y a un autre processus qui lit le message. Comme `msg_receive` modifie la file on ne peut pas effectuer deux lectures en même temps : ce comportement doit être assuré en utilisant des sémaphores ou des mutexes/conditions.

`msg_tryreceive` est une version non bloquante de `msg_receive`. Si la file est vide `msg_tryreceive` retourne -1 et met `EAGAIN` dans `errno`.

Par contre si la file n'est pas vide mais qu'un autre processus est en train de lire un message alors on vous laisse choisir ce que fait `msg_tryreceive` (attente, attente limitée en temps, ou retour immédiat comme pour la file vide).

En cas de réussite les deux fonctions retournent le nombre d'octets du message lu. En cas d'échec elles retournent -1.

2.1.6 L'état de la file

```
size_t msg_message_size(MESSAGE *)
size_t msg_capacite(MESSAGE *)
size_t msg_nb(MESSAGE *)
```

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

3 Structures de données

3.1 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 `msg_send` et `msg_receive`. Il faut vérifier que l'opération est autorisée, par exemple `msg_send` 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.

3.2 Organisation de la mémoire partagée

La 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 3.2.2.

L'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.

3.2.1 Tableau de messages

Le plus simple est de stocker les messages dans un tableau. Cela peut être un tableau circulaire. Dans ce cas il suffit 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 `msg_receive`. `last` est l'indice de premier élément libre de tableau, celui que `msg_send` utilisera pour placer le nouveau message.

Dans le tableau circulaire

— soit $first < last$ et les éléments de la file occupent les cases $first, first + 1, \ldots, last - 1$,

— soit $last \leq first$ et la file occupe les cases $first, first + 1, \ldots, n - 1, 0, 1, \ldots, last - 1$ où n est les nombres 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 le tableau plein de tableau vide on pourra supposer que $first == -1$ si le tableau est vide.

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 $first \leq last$. Les éléments de la file sont dans les cases $first, first + 1, \ldots, last - 1$. Dans ce cas quand on place le nouveau élément à la case de l'indice $n - 1$ (n le nombre d'éléments de tableau) et $first > 0$ alors il faudra faire un shift 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 donc chaque message est une structure qui contient au moins le nombre d'octets dans le message (nécessaire pour la valeur de retour de `msg_receive`) et le message lui-même.

3.2.2 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 ...

4 Files anonyme

Une file anonyme est créé par `msg_connect` dont le premier paramètre est NULL. Chaque `msg_connect(NULL,...)` ouvre une nouvelle file anonyme. La file anonyme peut être partagée uniquement par un processus qui a effectué `msg_connect` à ses enfants engendrés après la création de la file.

Le paramètre `options` est ignoré pour une file anonyme, la file anonyme sera toujours ouverte en lecture et écriture.

Comme `msg_connect(NULL,...)` ouvre chaque fois une nouvelle file anonyme les paramètres `nb_msg` et `len_max` sont toujours obligatoires pour `msg_connect(NULL,...)`.

La file anonyme ne supporte pas `msg_unlink`, elle disparaît automatiquement quand le dernier processus se déconnecte avec `msg_disconnect`.

5 Gestion d'accès parallèles

Les sémaphores ou variables mutex permettent 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. Franchement, c'est sans intérêt comme solution.

Si la file 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, pas de raison qu'un écrivain bloque un lecteur et vice-versa.

Si plusieurs processus ajoutent des messages dans la file, quand le premier processus a changé la valeur de `last` pour réserver la place pour son message pas la peine de faire poireauter le suivant qui peut déjà commencer à exécuter `msg_send()` en même temps que le premier copie le message dans la mémoire.

Une remarque similaire s'applique à des processus qui lisent les messages.

6 Organisation du code

Toutes les fonctions demandés doivent être regroupées dans un fichier `msg_file.c` accompagné de `msg_file.h` de telle sorte que chaque programme qui utilise les files de messages puisse faire juste un include de `msg_file.h` pour être compilé (l'édition de liens ajoutera `msg_file.o` obtenu par la compilation de `msg_file.c` mais c'est une autre histoire).

Les programmes de test doivent être dans des fichiers séparés.

Les noms de fichiers `msg_file.c` et `msg_file.h` ne sont pas à modifier (si nous nous décidons à écrire nos propres programmes pour tester vos fonctions alors il faut que include de `msg_file.h` marche dans nos programmes).

Deuxième partie
Extensions

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

6.1 Lecture/écriture par lots

Implémenter la lecture et l'écriture de plusieurs messages d'un seul coup : analogue à `readv` et `writev` mais appliqués aux files de messages et non pas aux fichiers.

Les lectures/écritures par lots doivent être atomiques, écrire 10 messages d'un coup ce n'est par écrire 10 messages dans une boucle un par un. Quand on écrit un lot de n messages alors

— il n'y a pas d'écriture partielle, soit on écrit n messages soit rien,

— les n messages doivent être consécutifs dans la file, il ne faut pas qu'ils soient intercalés avec d'autre messages.

Pour la lecture d'un lot de n messages on suppose qu'une lecture partielle est possible : si la file contient $m < n$ messages alors on lira seulement les m messages disponibles.

Les messages lus par lot doivent être consécutifs dans la file.

6.2 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 messages 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 veut dire que ce ne sont pas des pointeurs, les adresse virtuelles après chaque `mmap` sont différentes même si `mmap` est effectué sur le même objet mémoire. Donc il faut 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`.

6.3 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 de `msg_receive` ou `msg_tryreceive`, un processus s'enregistre et poursuit son exécution.

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 :

— un message arrive dans la file vide et

— il n'y a aucun processus suspendu en attente d'un message.

Quand le signal de notification est envoyé le processus enregistré doit être automatiquement dé-enregistré.