Projet Task Daemon
Système et programmation système

Licence 2 Informatique
2022 – 2023

Le but de ce projet est de réaliser un clone simplifié de `cron(8)` et `at(8)`, permettant de définir des actions à effectuer à intervalle de temps régulier. Le projet est à implémenter en langage C.

Spécifications

Le projet se compose de trois programmes principaux :
- un lanceur de daemon générique : `launch_daemon` (pouvant éventuellement servir pour lancer d'autres daemons)
- le programme exécuté par le daemon : `taskd`
- un outil en ligne de commande : `taskcli`

L'outil `taskcli` permet, entre autres, d'envoyer au programme `taskd` une commande à exécuter ainsi que sa date de départ et sa période. Une période de 0 indique que la commande ne doit être effectuée qu'une seule fois (à la manière de `at(8)`). La syntaxe est la suivante :

```
./taskcli START PERIOD CMD [ARG]...
```

où `START` est l'instant de départ, `PERIOD` est la période (en secondes) et `CMD` est la commande à exécuter avec ses éventuels arguments (`[ARG]...`). Les trois premiers arguments (`START`, `PERIOD` et `CMD`) sont obligatoires. L'instant de départ peut être spécifié de différentes manières :
- en indiquant le nombre de secondes depuis Epoch (1er janvier 1970) ;
- en indiquant le caractère `+` suivi du nombre de secondes avant de lancer la commande.

Quelques exemples :

```
./taskcli 1681805110 0 echo Bonjour
# On souhaite exécuter 'echo Bonjour' le mardi 18 avril à 10:05:10
# La commande ne sera exécutée qu'une seule fois

./taskcli +10 30 uptime
# On souhaite exécuter 'uptime' toutes les 30 secondes en commençant dans 10 secondes

./taskcli +15 2 date +%H:%M:%S
```

```
# On souhaite exécuter 'date +%H:%M:%S' toutes les 2 secondes en commençant dans 15 secondes

./taskcli +0 60 ls -l /home
# On souhaite exécuter 'ls -l /home' toutes les minutes à partir de maintenant
```

Pour envoyer une commande à exécuter au programme `taskd`, l'outil `taskcli` utilisera un tube nommé `/tmp/tasks.fifo` et le signal `SIGUSR1` pour réveiller `taskd`.

Le programme `taskd` est chargé de l'exécution des différentes commandes. Pour cela, sa tâche principale est de dormir jusqu'à ce qu'une commande doive être exécutée. Il reçoit les commandes à exécuter de l'outil `taskcli`. L'ensemble **courant** des commandes à exécuter est placé dans une structure de données dont la taille évolue dynamiquement. Si vous n'êtes pas à l'aise avec les structures de données dynamiques, vous pouvez utiliser un tableau de 256 commandes (vous serez alors notés sur 17). Dans la suite de cet énoncé, cette struture de données est appelée la liste courante des commandes à exécuter [1].

N.B. : Quand une commande qui ne doit être exécutée qu'une seule fois (c'est à dire dont la période est égale à 0) a été exécutée, elle doit être "supprimée" de la liste courante des commandes à exécuter.

Par ailleurs, le programme `taskd` maintient un fichier **texte** nommé `/tmp/tasks.txt` qui contient une description de la liste courante des commandes à exécuter [2]. Le fichier `/tmp/tasks.txt` contient une ligne par commande à exécuter. Le format du fichier est le suivant :

```
num_cmd;start;period;cmd
```

où :
- le champ `num_cmd` correspond au numéro d'enregistrement de la commande (1 pour la première commande enregistrée, 2 pour la seconde, etc.)
- le champ `start` est la date correspondant à l'instant de départ(`18 avril 10:15:36` par exemple)
- le champ `period` est la période d'exécution de la commande
- le champ `cmd` est la commande à exécuter avec ses arguments

Pour obtenir un "véritable" daemon, il faut lancer le programme `taskd` en utilisant `launch_daemon` :

```
./launch_daemon  ABSOLUTE_PATH_TO_TASKD
```

Outils complémentaires et bibliothèque

Exercice 1 : Préliminaires

Question 1.1 Écrire en C un programme `time` qui affiche le nombre de secondes depuis Epoch pour l'instant présent. Indice : `time(2)`.

1. commande à exécuter : commande qui doit être exécutée dans le futur
2. Le programme `taskd` doit mettre à jour le fichier `/tmp/tasks.txt` à chaque fois que la liste courante des commandes à exécuter est modifiée.

Question 1.2 Écrire en C un programme `when` qui prend en argument un nombre de secondes depuis Epoch et qui affiche la date à laquelle ce nombre correspond. Indice : `ctime(3)`.
N.B. : pour afficher la date en français, vous pouvez utiliser les fonctions `setlocale(3)` et `localtime(3)`.

Exercice 2 : Communication via un tube nommé

Le but de cette partie est de créer une bibliothèque dynamique `libmessage.so` qui contient des fonctions utiles aux programmes `taskd` et `taskcli`. Ces fonctions sont utilisées pour communiquer des chaînes de caractères via un tube nommé (en utilisant un descripteur de fichier pointant sur une de ses extrémités). Il faut faire un fichier d'en-tête `message.h` contenant les prototypes et un fichier source `message.c` contenant les définitions.

Il faut **traiter les erreurs** lors des écritures et des lectures dans les tubes nommés. De plus, il faut détecter les éventuelles **fins de fichiers** lors des lectures.

Question 2.1 Écrire une fonction `send_string` qui envoie une chaîne de caractères via un descripteur de fichier. Pour cela, on enverra d'abord la longueur de la chaîne, telle qu'elle est représentée dans la mémoire du programme [3] , puis son contenu (i.e. les caractères de la chaîne). La fonction a le prototype suivant :

```
int send_string(int fd, char *str);
```

Question 2.2 Écrire une fonction `recv_string` qui reçoit une chaîne de caractères, envoyée par `send_string` via un descripteur de fichier. Pour cela, on recevra d'abord la longueur de la chaîne, puis on allouera l'espace nécessaire (via `malloc(3)` ou `calloc(3)`, attention au caractère '\0' final), puis on lira les caractères de la chaîne. La fonction a le prototype suivant :

```
char *recv_string(int fd);
```

Question 2.3 Écrire une fonction `send_argv` qui envoie un tableau de chaînes de caractères (le dernier pointeur de ce tableau vaut NULL). On enverra d'abord la taille du tableau, puis chacune des chaînes de caractères. La fonction a le prototype suivant.

```
int send_argv(int fd, char *argv[]);
```

Question 2.4 Écrire une fonction `recv_argv` qui reçoit un tableau de chaînes de caractères, envoyé par `send_argv`. On recevra d'abord la taille du tableau puis on allouera l'espace nécessaire (Attention : le tableau doit se terminer par un NULL), puis on lira chacune des chaînes de caractères. La fonction a le prototype suivant :

```
char **recv_argv(int fd);
```

3. Si la longueur est mémorisée dans un `int`, il faut envoyer directement les `sizeof(int)` octets du `int` (Il s'agit du code complément à 2 sur `sizeof(int)` octets de ce `int`). Si la longueur est mémorisée dans un `size_t`, il faut envoyer directement les `sizeof(size_t)` octets du `size_t`.

Question 2.5 Avant de passer à la suite, tester, avec `valgrind` , les 4 fonctions de la bibliothèque en créant 2 programmes qui communiquent via un tube nommé [4]. Cette étape est **absolument fondamentale** : vous ne pourrez pas rendre un projet qui fonctionne si ces fonctions ne sont pas correctement implémentées.

L'outil `taskcli`

L'outil `taskcli` peut être appelé de deux façons :

```
./taskcli START PERIOD CMD [ARG]...
./taskcli
```

La première forme permet d'envoyer à `taskd` une nouvelle commande à exécuter. La seconde forme (sans argument) permet de lire le fichier `/tmp/tasks.txt`, et d'afficher son contenu.

Exercice 3 : Vérification de l'existance d'un processus exécutant `taskd`

Lorsqu'il démarre, le programme `taskd` enregistre son PID dans un fichier **texte** nommé `/tmp/taskd.pid`. L'existance ou la non existance de ce fichier permet de savoir si un processus est en train d'exécuter `taskd`. Ce fichier texte, quand il existe, permet donc de connaître le PID du processus qui exécute `taskd`, ce qui permettra ensuite de le "contacter" en lui envoyant par exemple un signal.

Question 3.1 Écrire une fonction qui permet de tester l'existance d'un processus exécutant `taskd`, et de lire le cas échéant son PID.

Exercice 4 : Envoi d'une nouvelle commande à exécuter

Pour envoyer à `taskd` une nouvelle commande à exécuter, il faudra lui transmettre les informations nécessaires via le tube nommé `/tmp/tasks.fifo`, et lui envoyer le signal `SIGUSR1`.

Question 4.1 Déterminer la date de départ et la période en fonction des arguments. On affichera une aide si une erreur est détectée (mauvais nombre d'arguments, arguments erronés). On utilisera `strtol(3)` pour convertir en entier les chaînes numériques passées en argument et détecter s'il y a une erreur.

Exemple de détection d'erreur :

```
$ ./taskcli +10az 5 echo bonjour
Invalid start : +10az
Usage : ./taskcli START PERIOD CMD [ARG]...
Usage : ./taskcli
```

4. Vous pouvez pour cela vous aider des programmes donnés dans l'annexe.

Question 4.2 Envoyer (dans un ordre à choisir) :
- les informations nécessaires (date de départ, période, commande avec ses arguments) via le tube nommé `/tmp/tasks.fifo` en vous servant notamment de la bibliothèque dynamique `libmessage.so` ;
- le signal `SIGUSR1` à `taskd` pour lui indiquer qu'il va devoir lire des données.

Exercice 5 : Affichage de la liste courante des commandes

L'outil `taskcli`, appelé sans argument supplémentaire, permet de lire et d'afficher le contenu du fichier `/tmp/tasks.txt` contenant l'ensemble courant des commandes à exécuter.

Le processus `taskd` modifie le fichier `/tmp/tasks.txt`, alors que les processus `taskcli` le lisent. Le fichier `/tmp/tasks.txt` est donc en quelque sorte **partagé** entre ces processus. Pour éviter que le processus `taskd` ne modifie le fichier partagé `/tmp/tasks.txt` à un instant où un processus `taskcli` est en train de le lire, et réciproquement qu'un processus `taskcli` ne lise le fichier partagé `/tmp/tasks.txt` à un instant où le processus `taskd` est en train de le modifier, vous devez utiliser un verrou consultatif : cf. `flock(2)`.

Exemples :

```
$ ./taskcli
No command to run.

$ ./taskcli
Commands to run :
1;18 avril 10:15:36;20;echo bonjour
3;18 avril 10:44:57;0;ls /tmp/tasks
4;18 avril 10:46:02;10;date +%H:%M:%S
```

N.B. : lors de la deuxième exécution de `taskcli`, on suppose que :
- la commande numéro 2 était une commande qui ne devait être exécutée qu'une seule fois, et qu'elle a déjà été exécutée ;
- l'instant présent est antérieur au timestamp correspondant au `18 avril 10:44:57` : la commande numéro 3 n'a pas encore été exécutée.

Question 5.1 Lire et afficher le contenu du fichier `/tmp/tasks.txt` en utilisant un verrou consultatif.

Le programme `taskd`

Exercice 6 : Initialisations diverses

Question 6.1 Écrire le PID du processus dans le fichier **texte** `/tmp/taskd.pid`. Comme il ne faut pas qu'il y ait plus d'un processus qui exécute ce programme, si ce fichier existe déjà, le programme doit se terminer.

Question 6.2 S'il n'existe pas, créer le tube nommé `/tmp/tasks.fifo`.

Question 6.3 S'il n'existe pas, créer le fichier texte `/tmp/tasks.txt`. Sinon, s'il existe déjà, tronquer le.

Question 6.4 S'il n'existe pas, créer le répertoire `/tmp/tasks`.

Exercice 7 : Définition des strutures de données

Question 7.1 Définir une structure de données pour chaque enregistrement (ou commande) de la liste.

Question 7.2 Définir une structure pour la "liste" de commandes à exécuter (qui, pour rappel, peut être un simple tableau de 256 commandes) ainsi que les opérations associées : ajout, etc.

Exercice 8 : Ajout d'une commande dans la liste

Question 8.1 Quand le signal `SIGUSR1` est reçu, lire la nouvelle commande envoyée par l'outil `taskcli` à travers le tube nommé, et faire les traitements nécessaires (ajout dans la liste, dans le fichier `/tmp/tasks.txt` [5], etc.).

Exercice 9 : Gestion des commandes à exécuter

Question 9.1 Parcourir la liste pour déterminer la durée à attendre avant la prochaine action à effectuer, et prévoir une alarme.

Question 9.2 A la réception du signal `SIGALRM`, vérifier qu'il y a bien une (ou plusieurs) action(s) à effectuer et les lancer. Pour chaque action, on fera les redirections indiquées ci-dessous :

- l'entrée standard sera redirigée vers le trou noir `/dev/null` ;
- la sortie standard et l'erreur standard seront redirigées respectivement vers les fichiers `/tmp/tasks/X.out` et `/tmp/tasks/X.err` où `X` est le numéro d'enregistrement de la commande (1 pour la première commande enregistrée, 2 pour la seconde, etc.). Les deux fichiers `/tmp/tasks/X.out` et `/tmp/tasks/X.err` seront ouverts en mode ajout en fin de fichier.

Attention à ne pas oublier des actions à exécuter.

Exercice 10 : Gestion des fin de processus

Question 10.1 Gérer le signal `SIGCHLD`, qui est notamment envoyé à un processus quand un de ces processus fils se termine, de manière à éliminer les zombies (bien vérifier que tous les processus zombies sont éliminés). Le processus père devra afficher, sur son erreur standard, comment ses fils se sont terminés.

5. en utilisant un verrou consultatif : cf. `flock(2)`

Exercice 11 : Gestion de la fin du programme `taskd`

Question 11.1 Gérer les signaux `SIGINT`, `SIGQUIT` et `SIGTERM` afin que l'envoi de l'un de ces signaux termine "proprement" le processus qui exécute `taskd` :

- supprimer le fichier `/tmp/taskd.pid`
- libérer l'espace mémoire alloué dynamiquement
- terminer et éliminer tous les processus créés restant

Exercice 12 : Gestion des signaux

Afin de réduire au maximum la taille du code exécuté dans les handlers de signaux, de ne pas utiliser de fonctions qui ne sont pas «async-signal-safe» dans un handler, et de ne pas devoir créer des structures de données globales (comme une liste chaînée) accessibles (notamment en modification) à la fois dans un handler et dans le programme principal, nous vous conseillons d'adopter l'architecture suivante :

```
/* déclaration de variables globales servant de drapeaux */
volatile int usr1_receive = 0;
volatile int alrm_receive = 0;
...

/* handler du signal SIGUSR1 */
void handSIGUSR1(int sig) {
  usr1_receive = 1;
}
/* handler du signal SIGALRM */
void handSIGUSR1(int sig) {
  alrm_receive = 1;
}
...

int main (int argc, char *argv[]){
  /* initialisations diverses */
  /* installation des handlers */

  while(1){
    // attente de réception d'un signal
    ...
    if (usr1_receive){
      // faire ce qu'il faut quand le processus taskd reçoit SIGUSR1
      usr1_receive = 0;
    }
    if (alrm_receive){
      // faire ce qu'il faut quand le processus taskd reçoit SIGALRM
      alrm_receive = 0;
    }
    ...
  }
  ...
}
```

Pour attendre la réception d'un signal, vous pouvez, dans une première version, utiliser `pause(2)`. Mais, cette solution présente une faille : si un signal est délivré au processus à un instant où il n'est pas bloqué sur l'appel de `pause(2)`, il restera bloqué sur l'appel de `pause(2)` suivant. Cela peut se produire si le processus reçoit 2 signaux dans un laps de temps très court, la délivrance du premier signal interrompant l'appel de `pause(2)`, et celle du second ayant lieu avant que le processus ne rappelle `pause(2)`.

Pour annihiler cette faille, on pourrait penser à démasquer (avec `sigprocmask(2)` l'ensemble des signaux attendus dans la boucle juste avant l'appel de `pause(2)`, et à les masquer juste après cet appel :

```
while(1){
  // attente de réception d'un signal
  sigprocmask(...); // démasquage de l'ensemble des signaux attendus
  pause();
  sigprocmask(...); // masquage de l'ensemble des signaux attendus
  ...
}
```

Mais cela ne changerait rien : dans le scénario précédent, le 2ème signal serait délivré au retour du premier appel de `sigprocmask(2)`, juste avant l'appel de `pause(2)`.

Pour résoudre ce problème, il faut utiliser `sigsuspend(2)` qui permet de réaliser de façon **atomique** l'installation du masque de signaux qu'on lui passe en paramètre ET la mise en sommeil du `processus`. Et si l'appel de `sigsuspend(2)` retourne suite à la délivrance d'un signal qui ne termine pas le processus (ce que l'on souhaite ici), au retour de cet appel, le masque de signaux du processus est restoré à sa valeur avant l'appel de `sigsuspend(2)`.

Le lanceur de daemon `launch_daemon`

Pour obtenir un "véritable" daemon, il faut lancer le programme `taskd` en utilisant `launch_daemon` :

```
./launch_daemon  ABSOLUTE_PATH_TO_TASKD
```

N.B. : le programme `launch_daemon` est générique : il pourrait être utilisé pour lancer d'autres daemons.

Exercice 13 : Modification de `taskd`

Question 13.1 Rediriger sa sortie standard et son erreur standard respectivement vers les fichiers `/tmp/taskd.out` et `/tmp/taskd.err`. Si un fichier n'existe pas, il faut le créer ; sinon, il ne faut pas le tronquer : toutes les écritures doivent être faites en fin de fichier.

Exercice 14 : Implémentation du lanceur de daemon

Question 14.1 Implémenter la méthode du double `fork(2)` pour lancer le daemon. Pour rappel, voici les actions à effectuer :

- Le processus principal crée un nouveau processus, et attend la fin de son fils.
- Le processus fils devient leader de session via `setsid(2)`.
- Le processus fils crée un nouveau processus, et se termine en appelant `exit(2)`, rendant le petit-fils orphelin.
- Le processus petit-fils n'est pas leader de session, ce qui l'empêche d'être rattaché à un terminal.
- Le processus principal se termine, laissant le petit-fils être le daemon.

→ **Explications supplémentaires**

- Le shell qui lance le processus principal (ou processus père) crée un nouveau groupe de processus, dont le leader est le processus principal. La création d'une nouvelle session passe par la création d'un nouveau groupe de processus ; la nouvelle session et le nouveau groupe prennent l'identifiant du processus appelant (i.e. le processus appelant en est le leader). Or pour créer un nouveau groupe (et donc une nouvelle session), un processus ne doit pas être déjà leader de son groupe : il est indispensable que cet identifiant ne soit pas encore attribué à un groupe, ou à une session, qui pourraient éventuellement contenir d'autres processus.
- À partir du moment où le fils crée une nouvelle session, tous ses descendants (qui ne créent pas eux mêmes une nouvelle session) appartiennent à cette session.
- Tous les processus d'une session partagent (éventuellement) un même terminal de contrôle. Une session sans terminal de contrôle en acquiert un lorsque le leader de session ouvre un terminal pour la première fois. Ici, le leader de session se termine sans ouvrir de terminal. Donc la session, à laquelle appartient le petit fils, ne pourra plus jamais acquérir un terminal de contrôle.

Question 14.2 Changer la configuration du daemon :

- Changer le répertoire courant à `/` via `chdir(2)` ;
- Changer le umask à 0 via `umask(2)` ;
- Fermer tous les descripteurs de fichiers standard.

Question 14.3 Charger le nouveau programme à exécuter : `taskd`

Bonus

Exercice 15 : Retrait d'une commande

Question 15.1 Ajouter la possibilité de retirer une commande de la liste courante des commandes à exécuter avec l'outil `taskcli`.

Annexe : les tubes nommés

Un tube est un mécanisme de type **FIFO** (First In First Out) qui permet à deux processus d'échanger des informations de manière **unidirectionnelle** en mode **flot**. "En mode flot" signifie que les envois successifs de données dans un tube sont vues du point de vue de leur extraction comme une une seule et même émission.

Le schéma (fig. 1) illustre la possibilité de réaliser des opérations de lecture dans un tube sans relation avec les opérations d'écriture.

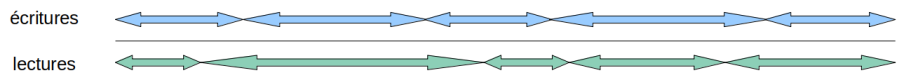

FIGURE 1 – Echange d'informations en mode flot

Ce mode de communication est à opposer à la communication en mode structuré (ou datagramme) dans lequel une lecture correspond exactement à une écriture. "En mode flot", les limites entre les messages émis ne sont pas préservées. Par exemple, si un processus appelle deux fois `write(2)` pour écrire respectivement 100 octets et 200 octets, les 300 octets écrits peuvent être lus par un seul appel de `read(2)`.

Un tube **anonyme** est un tube qui n'existe que pendant la durée des processus qui l'utilisent, et disparaît ensuite. Il ne permet la communication qu'entre processus **ayant un lien de parenté**. On le crée en shell avec le symbole `|`.

Les tubes **nommés** ont les mêmes caractéristiques que les tubes anonymes et ont la propriété supplémentaire d'être **référencés** dans le système de fichiers. C'est cette propriété qui permet à tout processus connaissant une référence sur un tube nommé d'obtenir en appelant `open(2)` un descripteur en lecture ou en écriture connecté à ce tube, afin de communiquer des informations à d'autres processus par son intermédiaire. Ils permettent donc la communication entre processus **sans lien de parenté**. Un tube nommé est un tube qui persiste tant qu'il n'est pas explicitement détruit. On le crée en shell avec la commande `mkfifo(1)`.

Pour plus d'informations : `man 7 fifo`.

→ Exemple de création et d'utilisation d'un tube nommé en shell

Synopsis de la commande `mkfifo(1)` :

```
mkfifo [OPTION]... name...
```

Cette commande permet de créer un tube nommé dont le nom est `name`.

Premier exemple :

```
$ mkfifo foo
$ ls -l foo
prw-r--r-- 1 eric eric 0 avril 18 17:36 foo
```

```
$ rm foo
$
```

Les tubes nommés sont identifiables dans les résultats fournis par la commande `ls -l` par le caractère `p` correspondant à leur type.

Deuxième exemple :

```
$ mkfifo baz
$ gzip -9 -c < baz > out.gz
```

On commence par créer un tube nommé `baz` avec la commande `mkfifo(1)`. Ensuite, pour la commande `gzip`, le shell crée un nouveau processus pour exécuter la commande `gzip -9 -c`. Avant que le fils ne se recouvre pour exécuter la commande `gzip`, le shell doit rediriger l'entrée standard du processus fils vers la sortie du tube nommé, et sa sortie standard vers le fichier `out.gz`. Pour cela, il doit ouvrir en lecture seule le tube nommé : cette ouverture est **bloquante** jusqu'à ce qu'un autre processus réalise une ouverture en écriture du même tube nommé (cf. le paragraphe suivant).

On exécute alors dans un second terminal :

```
$ cat file > baz
```

Pour exécuter cette commande, le shell associé au second terminal crée un nouveau processus pour exécuter la commande `cat file`. Avant que le fils ne se recouvre pour exécuter la commande `cat`, le shell doit rediriger la sortie standard du processus fils vers l'entrée du tube nommé `baz`. Pour cela, il ouvre ce tube nommé en écriture seule, ce qui a pour effet de débloquer le premier shell.
Les 2 shells finissent alors leur travail de redirection des descripteurs standard des 2 processus fils, et ces 2 processus fils se recouvrent : ils chargent, en mémoire, respectivement les exécutables des commande `gzip` et `cat`.
Les 2 processus fils s'exécutent alors en concurrence. Quand le processus qui exécute `cat` se termine (plus précisément quand il ferme le descripteur correspondant à sa sortie standard), le processus qui exécute `gzip` détecte une fin de fichier sur la sortie du tube nommé (son entrée standard) une fois qu'il est vide, et se termine alors à son tour.
Le fichier `out.gz` contient alors le résultat de la compression du fichier `file`.

→ Création d'un tube nommé en C

```
int mkfifo(const char *pathname, mode_t mode);
```

`mkfifo(3)` crée le tube nommé dont le nom est `pathname` avec les permissions `mode`.

→ Ouverture d'un tube nommé en C

- Une fois créé, le tube nommé peut être ouvert avec `open(2)` comme n'importe quel fichier.
- Pour être utilisé, un tube nommé doit être ouvert par deux processus, un en lecture et l'autre en écriture.
- L'ouverture d'un tube nommé **peut être bloquante** :
  - une demande d'ouverture en lecture est bloquante en l'absence d'écrivain sur le tube et de processus bloqué sur une ouverture en écriture
  - une demande d'ouverture en écriture est bloquante s'il n'y a aucun lecteur sur le tube et aucun processus bloqué sur une ouverture en lecture

→ Exemple de création et d'utilisation d'un tube nommé en C

```
int main (void)
{
  if (mkfifo("baz", 0644)) {
    perror("mkfifo");
    exit(EXIT_FAILURE);
  }
  fd = open("baz", O_RDONLY);
  char buf;
  while (read(fd, &buf, 1) > 0){
      ...
  }
  close(fd);
  unlink("baz");
  return 0;
}
```

Ce premier programme commence par créer un tube nommé de nom `baz`. Ensuite, il réalise une demande d'ouverture en lecture seule de ce tube nommé. Cette demande d'ouverture est bloquante car, à l'instant où `open(2)` est appelée, il n'y a ni écrivain dans ce tube nommé, ni aucun processus bloqué sur une demande d'ouverture en écriture.

```
int main (void)
{
  int fd = open("baz", O_WRONLY);
  ...

  const char *str = "hello world!";
  size_t lo = strlen(str);
```

```
  ssize_t nb = write(fd, str, lo);
  close(fd);
  return 0;
}
```

L'exécution de ce second programme est alors lancée. Ici, l'appel de `open(2)` retourne immédiatement car le premier processus est bloqué sur une demande d'ouverture en lecture du même tube nommé. En fait, les 2 appels de `open(2)` dans les 2 processus retournent "simultanément". Le noyau assure une **synchronisation** entre les 2 processus : le premier processus qui fait une demande d'ouverture attend que l'autre fasse une demande d'ouverture dans l'autre sens (lecture / écriture) pour poursuivre son exécution.

Remarque : la synchronisation assurée par le noyau lors des ouvertures du tube est nécessaire car :

- si le premier processus appelait `read(2)` avant qu'il y ait un écrivain dans le tube, il détecterait immédiatement une fin de fichier avant que l'écrivain ait eu le temps d'écrire des données dans le tube ;
- si un processus écrivain écrit dans un tube alors qu'il n'y a pas de lecteur dans le tube, le noyau lui envoie le signal `SIGPIPE`, qui par défaut le termine.

Ensuite, les opérations de lecture de d'écriture dans le tube nommé fonctionnent comme dans un tube anonyme : le noyau assure une synchronisation des processus qui écrive et lise dans le tube :

- si le tube est plein, il bloque le processus écrivain
- si le tube est vide, et qu'il reste au moins un écrivain, il bloque le processus lecteur
- si le tube est vide, et qu'il y a plus d'écrivain, le lecteur détecte une fin de fichier (i.e. `read(2)` renvoie 0). C'est pourquoi, le processus lecteur ne sort de la boucle de lecture qu'après la fermeture du tube dans l'écrivain.

L'appel de `unlink(2)` correspond à une demande de suppression du lien physique `baz`.

Évaluation

La note du projet tiendra compte des points suivants (liste non-exhaustive) :
- séparation du projet en plusieurs unités de compilation et bibliothèque(s) [6] ;
- présence d'un Makefile fonctionnel [7] ;
- optimalité du Makefile ;
- qualité du code :
  - découpage en fonctions, **factorisation** ;
  - optimalité des algorithmes ;
  - paramètrage des applications à l'aide de constantes macrodéfinies ;
  - commentaires dans le code source (chaque fonction doit être précédée d'un bloc de commentaires qui indique son rôle et décrit les paramètres attendus et sa valeur de retour) ;
  - contrôle des valeurs de retour des appels système et fonctions ;
  - absence de fuites et d'erreurs mémoire.

Vous devrez déposer, dans le dépôt MOODLE prévu à cet effet avant la date indiquée, une archive compressée, au format `.tar.gz`, contenant vos programmes source, et un fichier texte README. Cette archive aura pour nom les deux noms du binôme, mis bout à bout et séparés par le caractère underscore (exemple : «dupont_durand.tar.gz»).

Le fichier README doit contenir :
- une conclusion/bilan sur le produit final (ce qui est fait, testé, non fait) ;
- un bilan par rapport au travail en binôme.

6. Le projet doit contenir au moins une bibliothèque dynamique : la bibliothèque `libmessage.so`

7. Il ne doit y avoir qu'un seul fichier Makefile dans votre projet : il doit permettre de produire l'ensemble des exécutables et bibliothèque(s) du projet.