Command Launcher - Manuel technique

Rémi DURIEU / Thomas EVRARD

Décembre 2020

Arborescence du projet

```
.
|-- cmdl.c              # Sources du client
|-- cmdld.c             # Sources du daemon
|-- cmdld.conf          # Fichier de configuration du daemon
|-- inc                 # -- Répertoire contenant les en-têtes des modules
|   |-- common.h        # Définitions communes utilisées par le client et le daemon
|   |-- config.h        # En-tête du module de configuration
|   |-- squeue.h        # En-tête du module de file synchronisée
|-- LICENSE             # Licence MIT
|-- Makefile            # Makefile
|-- README.md           # README
|-- src                 # -- Répertoire contenant les sources des modules
|   |-- config.c        # Sources du module de configuration
|   |-- squeue.c        # Sources du module de file synchronisée
|-- test                # -- Répertoire contenant les sources des programmes de test
    |-- test.sh         # Script shell de test global
    |-- test_squeue.c   # Programme de test du module de file synchronisée
```

File synchronisée

Le module `squeue` est un module générique et autonome permettant de gérer des files synchronisées en mémoire partagée. Les files créées acceptent tout type d'objets à la seule condition que la taille de ces derniers soit précisée à la création d'une file.

Le module définit le type opaque `SQueue` destiné à être manipulé uniquement au travers des fonctions fournies.

La création d'une nouvelle file s'effectue avec la fonction `sq_empty()`. Cette fonction prend en paramètre le nom de l'objet mémoire partagée à utiliser, la taille des éléments qui seront stockés dans la file et la taille maximale de cette dernière. Une file déjà existante peut alors être ouverte avec la fonction `sq_open()` en précisant le nom de l'objet mémoire partagée.

Pour garantir l'accès unique à la mémoire partagée, les fonctions du module font appel à un mécanisme d'exclusion mutuelle (mutex `mshm` dans la structure privée `__squeue`).

L'enfilage et le défilage de données est analogue au problème du producteur/consommateur : les fonctions `sq_enqueue()` et `sq_dequeue()` utilisent des sémaphores pour assurer le blocage du processus en cas d'enfilage d'une file pleine (sémaphore `mnfull`) ou de défilage d'une file vide (sémaphore `mnempty`).

Les données enfilées sont entièrement copiées dans la file et la zone mémoire utilisée est un tableau d'octets à taille variable défini en fin de la structure `__squeue` (VLA C99), ce qui permet d'y stocker tout type d'élément.

Les ressources utilisées par une file peuvent être libérées par appel de la fonction `sq_dispose()`.

Pour tester le module, le programme de test `test_squeue` est fourni. Il utilise la fonction `assert()` afin de tester le comportement de chaque fonction une par une. Puis il teste le mécanisme de blocage en enfilant 21 éléments à la suite dans une file de longeur 16, tandis que 5 éléments sont défilés en parallèle afin de permettre l'enfilage des éléments en attente. Le résultat est finalement affiché sur la sortie standard.

Client (`cmdl.c`)

Signaux

Le daemon communique aux clients l'échec ou la réussite des commandes par les signaux `SIGUSR1` et `SIGUSR2` (redéfini en `SIG_FAILURE` et `SIG_SUCCESS`).

Afin de s'assurer du bon affichage du résultat des commandes, `SIG_SUCCESS` est bloqué le temps de la lecture depuis le tube de communication et de l'écriture sur la sortie standard. Le signal est ensuite débloqué et le client placé en attente de ce dernier.

La réception d'un signal `SIG_FAILURE`, quand à elle, peut interrompre à tout moment le client pour indiquer un échec de l'exécution de la commande.

Requêtes

Le client récupère la commande a envoyer au daemon depuis les arguments passés en ligne de commande. Une requête (`struct request`) est ensuite créée. Cette dernière contient la commande à exécuter, le nom du tube de communication et le PID du client.

La file synchronisée préalablement créée par le daemon est ouverte, la requête est enfilée, et le tube de communication est créé et ouvert, ce qui a pour effet de bloquer le processus jusqu'à ce que le daemon prenne en charge la requête et ouvre à son tour le tube, ou qu'un `SIG_FAILURE` interrompe l'attente. Après affichage sur la sortie standard, le client se place en attente d'un `SIG_SUCCESS` avant de se terminer.

Daemon (`cmdld.c`)

Unicité

Pour empêcher l'exécution de plusieurs daemons en même temps, un mécanisme d'unicité a été mis en place. Lors du démarrage de `cmdld`, le programme tente d'abord de verrouiler un mutex (fonction `trylock()`). Si le verrouillage échoue, cela signifie qu'une autre instance du daemon est déjà en cours d'exécution et le programme s'arrête.

Le PID du daemon en cours d'exécution est stocké dans une mémoire partagée. Ceci permet à la commande `cmdld stop` de récupérer le PID du daemon afin de lui envoyer un signal de terminaison `SIGTERM`.

Configuration

Le module de configuration permet de lire le contenu du fichier `cmdld.conf`. Ce dernier contient la longueur maximale de la file synchronisée, ainsi que le nombre de workers.

Dans `cmdld.conf` Les clés et les valeurs sont séparées par une ou plusieurs tabulations et les lignes commençant par le caractère `#` sont ignorées.

La longueur maximale des commandes et du nom des tubes de communication ont été jugés comme relevant de l'ordre de l'implémentation et ne figurent donc pas dans le fichier de configuration. L'en-tête `common.h` définit ces deux constantes dans le cas où elles ne le seraient pas déjà par le système.

Daemonisation

Le processus de daemonisation a été implanté tel que décrit sur Wikipedia (en omettant les étapes optionnelles). C'est la fonction `daemonise()` qui se charge de l'opération.

Afin de confirmer la daemonisation, le processus parent attend un signal `SIG_SUCCESS` en provenance du daemon. Si celui-ci n'arrive pas dans les 5 secondes, la daemonisation est considérée comme échouée et le processus s'arrête.

À la fin de la daemonisation, la fonction `maind()` est exécutée. Elle contient les étapes d'intitialisation du daemon ainsi que sa boucle principale.

Traitement des requêtes

Le daemon est dans une boucle infinie bloquée par `sq_dequeue()` lorsque la file de resquêtes est vide. Lorsqu'une requête est enfilée par un client puis défilé par le daemon, ce dernier recherche le premier worker libre, lui confie la requête et le débloque. Si aucun workers libre n'a été trouvé, il envoie un signal `SIG_FAILURE` au client.

Workers et exécution de la commande

Au démarrage du daemon, les workers sont initialisés et les threads qui leur sont associés sont lancés avec la fonction de démarrage `wkstart()`. Cette fonction lance une boucle infinie bloquante à chaque itération. Lors du traitement des requêtes, le daemon débloque le thread associé à un worker ce qui permet à ce dernier de traiter la commande présente à ce moment là dans la structure `struct worker` qui lui est associée.

Lorsque la commande lancée par un worker a terminé de s'exécuter, ce dernier envoie un signal `SIG_SUCCESS` ou `SIG_FAILURE` au client en fonction du statut de la commande. Le worker en question est alors à nouveau disponible et bloque son thread en attendant une nouvelle requête.

Pistes d'améliorations

- Signaux plus précis dans le cas d'un échec de commande
  - Utilisation des signaux temps rééls ?
- Redémarrage du daemon à la réception d'un `SIGHUP`
  - Recharger le fichier de configuration
  - Envoyer `SIG_FAILURE` aux clients dont les commandes sont en cours de traitement (mise à profit de `sq_apply()`) ?