Introduction

La gestion des sources du projet s'est faite via `git`, avec un dépôt privé hébergé GitHub. Une intégration continue via Travis a été mise en place, avec l'envoi des informations de couverture de code vers Codecov.io. Ainsi, à chaque commit envoyé, nous avions un retour sur les résultats des tests, ainsi que sur la couverture du code.

Le projet est organisé pour avoir les fichiers sources (fichiers `.c` *et* `.h`) dans `src/`, les scripts de test dans `test/`, la génération des objets dans `obj/`, et les binaires finaux dans `bin/`. Le `Makefile` a donc été modifié en conséquence. Aussi, le `Makefile` contient des règles pour générer le PDF du rapport via `pandoc` à partir du fichier `README.md`, ainsi que la génération de l'archive de rendu.

Structure générale

Le code est structuré sémantiquement en plusieurs fichiers. Les fonctions (en dehors de celles dans le fichier `main.c`) ont été pensées de manière à ce qu'elles puissent éventuellement être utilisées dans un autre contexte, le tout empaqueté dans une bibliothèque.

Ainsi, les appels à `exit(3)`, ainsi que les écritures dans la sortie standard/sortie d'erreur ne se font que dans quelques fonctions. La remontée des erreurs se fait via des codes de retour négatifs (cf. section sur la gestion des erreurs).

`watch.c` – Structure `Watcher`

Toutes les informations relatives au lancement d'une commande sont donc contenues dans une structure `Watcher`.

La logique est la suivante:

- Un objet `Watcher` est créé pour une commande via `create_watcher(const char *commande[])`
- On lance `run_loop(…)` avec en paramètre le *watcher* à lancer, ainsi que quelques paramètres comme l'intervalle de temps entre les lancements, la limite du nombre de lancements, etc.
- `run_loop` va faire plusieurs appels à `run_watcher(Watcher w)`, qu'elle lance une fois la commande associée au *watcher*, en comparant sa sortie standard avec la sortie standard du lancement précédent. `run_watcher` retourne `1` si un changement a été constaté, et `0` sinon.
- `run_loop` s'occupe ainsi de la logique d'affichage, de l'intervalle entre les exécutions, de la limite du nombre d'exécution, de l'affichage du temps si besoin, et de la comparaison du code de sortie si besoin.

- `free_watcher` est ensuite appelé pour désallouer le *watcher*, ainsi que le *buffer* qu'il utilise en interne.

La structure `Watcher` contient les champs suivants :

- `Buffer last_output`: *buffer* contenant la sortie de la dernière exécution
- `int last_status`: code de retour de la dernière exécution
- `int run_count`: nombre d'exécutions de la commande
- `int exec_failure`: `1` si la dernière exécution a échoué (par exemple, si le binaire n'existe pas)
- `char **command`: la commande à exécuter

Cette structure est mutable, et est modifiée par effet de bord par `run_watcher`.

`buffer.c` – Structure `Buffer`

Pour stocker la sortie de chaque exécution, nous avons opté pour une structure de liste chainée de tampons. Ce choix nous évite par exemple d'avoir à constamment `realloc` le *buffer* pour l'agrandir, et limite les copies de gros segments en mémoire.

La structure est ainsi relativement simple, et sa définition parle d'elle même:

```
#define BUF_SIZE 1024

typedef struct s_buffer {
  int size;
  char content[BUF_SIZE];
  struct s_buffer *next;
} *Buffer;
```

`buffer.c` contient ainsi les fonctions relatives à la gestion de ces *buffers* :

- `Buffer create_buffer(void)` alloue un nouveau *buffer*
- `void free_buffer(Buffer)` libère (récursivement) un *buffer*
- `int print_buffer(Buffer)` affiche (récursivement) le contenu d'un *buffer* dans la sortie standard
- `int compare_buffers(Buffer a, Buffer b)` compare le contenu de deux *buffers* (retourne `1` si ces contenus sont identiques, `0` sinon)
- `Buffer read_to_buffer(int fd)` lit depuis un descripteur de fichier, et stocke le résultat dans **un** *buffer*

`spawn.c` – Lancement des commandes

Ce fichier contient uniquement une fonction `int spawn(const char* command[])`, qui lance la commande spécifiée, et retourne un descripteur de fichier permettant de lire la sortie standard de cette commande.

Ce lancement se fait via un `fork` suivi d'un `execvp`, avec la création d'un `pipe` pour lire la sortie standard de la commande.

`main.c` – Analyse des options et lancement

Le fichier `main.c` se limite ainsi à l'analyse des options (avec l'affichage de l'aide si besoin), et d'appels aux fonctions définies dans `watch.c`.

`util.[ch]` - Fonctions utilitaires & macros de gestion d'erreurs

Ce fichier contient une fonction pour afficher le temps courant selon le format `print_time`, ainsi que la logique de gestion des erreurs.

Gestion des erreurs

En gardant à l'esprit que les fonctions en dehors du fichier `main.c` pouvaient éventuellement faire partie d'une bibliothèque, nous ne voulions rien écrire, ni `exit` en cas d'erreur. Nous avons donc écrit une macro `TRY(…)` (et son équivalent `TRY_ALLOC(…)` pour les allocations) pour la gestion des erreurs des primitives système.

Cette macro va évaluer ce qui lui est passé en paramètre, et si c'est égal à `-1`, force un `return -1` de la fonction courante. Ainsi, les fonctions comme `spawn`, `read_to_buffer`, etc. vont également retourner `-1` en cas d'erreur. (Dans le cas des allocations, `TRY_ALLOC` fait retourner `NULL`)

On retrouve ainsi dans le code des appels type `TRY(pid = fork())`.

Aussi, cette macro va enregistrer ce qui a été passé en paramètre (cette fois, en non-évalué) dans une variable globale `errsrc`, permettant ainsi de tracer d'où vient l'erreur. La valeur de `errsrc` peut être obtenue via `char *geterr(void)`, définie dans `util.c`.

Cela permet d'avoir, dans `main.c`, le code suivant:

```
if (run_loop(w, opt_format, opt_interval,
             opt_limit, opt_flag_check_status) == -1) {
  // Something failed, show the error
  perror(geterr());
  return EXIT_FAILURE;
}
```

Si à un moment l'appel à une primitive système a échoué, `run_loop` retourne `-1`, et on affiche à l'utilisateur l'appel système qui a échoué.

Par exemple, si le `pipe` dans `spawn.c` échoue, on retrouvera dans la sortie d'erreur:

```
pipe(fildes): Too many open files
```

Cette méthode n'est probablement pas *idéale* (par exemple, les erreurs après le `fork` ne sont probablement pas remontées correctement), et peut éventuellement donner des messages pas forcément très parlant pour l'utilisateur, mais elle a le mérite d'être relativement simple à mettre en place, tout en donnant suffisamment d'informations pour déboguer en cas d'erreur.

Remontée des erreurs depuis le fils

Pour signaler une erreur dans l'exécution dans le fils, le fils envoie un signal `SIGUSR1` au parent. Ce signal est capturé par le parent, et le flag `watcher->exec_failure` est mis à `1` dans le *watcher*.

La fonction capturant le signal est mise en place via `install_signal`, supprimée par `restore_signal`, respectivement au début et à la fin de `run_loop`. Ces fonctions sauvegardent le *watcher* concerné dans la variable statique `installed_watcher`.

Jeux de tests

Des tests supplémentaires ont été écrits, soit pour des raisons de couverture, ou simplement pour des cas qui n'étaient pas testés, et qui posaient problème dans notre implémentation initiale.

Dans `test-110.sh`, des tests couvrent les cas de:

- commande invalide
- option invalide
- affichage de l'aide
- commande n'écrivant rien dans la sortie standard (ex: `true`)
- remontée d'erreur en cas d'échec de primitive système

Pour forcer l'échec d'une primitive système, nous avons utilisé `ulimit -n 4`, qui limite le nombre de descripteur de fichiers ouverts à 4 (ce qui dans nos tests fait échouer l'appel à `pipe` dans `spawn.c`).

Le test `test-130.sh` contient un test similaire à `test-120.sh`, en ayant une commande qui écrit alternativement `toto` et `titi`. Ce test est effectué pour avoir un cas où la sortie n'est pas la même, mais est de la même taille (puisque la taille du *buffer* est d'abord testée dans `compare_buffers`, avant de vérifier le contenu).

Dans la première implémentation de `read_to_buffer`, un seul appel à `read` était fait. Cela ne posait pas problème dans les jeux de tests fournis, puisque les données arrivaient suffisamment "rapidement" pour systématiquement remplir entièrement le *buffer*.

Cependant, cela posait problème dans le cas où l'écriture dans la sortie standard se faisait en deux parties. Par exemple, si lors de deux exécution, la commande écrivait les 15 même caractères, mais lors de la première exécution, elle en écrivait 5 puis les 10 autres, alors que dans la seconde exécution elle écrivait les 8 premiers puis les 7 autres, la comparaison des *buffers* indiquait que la sortie n'étais *pas* la même, alors que c'était le cas.

Nous avons donc écrit un petit programme en C – `src/test-150-script.c` – qui a ce comportement. Le programme est relativement simple, il écrit une chaine de caractère en deux fois, avec un appel à `fsync` (qui force l'écriture des tampons internes du noyau) entre les deux `write`.

Ce programme est ensuite appelé par `test/test-150.sh`.

Avec tous ces tests, nous avons réussi à obtenir 100% de couverture de code, avec – selon Valgrind – aucune fuite de mémoire.