Exercices sur le contrôle de processus et IPC : Automatisation des tests

Chaque fois que l'on a écrit des tests pour un module `Module`, on les a regroupés dans un module `ModuleTest`, les fonctions ayant toutes la forme `void ModuleTest_function (void)`. Ces fonctions ne font rien lorsque leurs tests passent, et font échouer le programme avec `assert()` lorsque leurs tests ne passent pas. Une fonction du module, de la forme `void ModuleTest_runAll (void)`, les invoque toutes en séquence, et on invoque cette fonction dans le programme principal.

Cette approche a plusieurs problèmes, en particulier :

- à chaque fois qu'on écrit une fonction de test `void ModuleTest_function (void)`, il faut penser à la rajouter son invocation dans le corps de `void ModuleTest_runAll (void)`. Il est facile d'oublier d'invoquer une fonction, et c'est d'ailleurs arrivé.
- les fonctions sont lancées dans le même processus, et une fonction peut échouer parce qu'une autre appelée précédemment a corrompu la mémoire.
- la suite de tests s'arrête au premier test qui échoue, or on voudrait les tester toutes : une autre fonction échoue peut-être sans qu'on le sache parce qu'invoquée après le premier échec.
- hormis lorsqu'un `assert()` échoue et affiche son diagnostic, on ne dispose pas de retour sur les tests lancés : liste des tests invoqués, nombre total de tests, nombre de tests passés.

On souhaite donc améliorer la méthode d'invocation des tests :

- en rendant automatique l'invocation de chaque tests,
- en lançant chaque test dans un processus séparé,
- en lançant tous les tests, sans s'arrêter au premier échec,
- en listant et en comptant les tests invoquée, et en comptant les tests passés et non-passés.

Pour cela, on va exploiter le fait que la commande `nm program` liste toutes les fonctions définies dans l'exécutable `program` par une ligne de la forme `"address flag function"`, où :

- `address` est l'adresse de la fonction sur 16 chiffres hexadécimaux. Ce nombre de 64 bits est représentable par un `unsigned long long`.
- `flag` est l'un des caractères `t` ou `T`, selon que la fonction est déclarée `static` ou non.
- `function` est le nom de la fonction.

Par exemple, dans la sortie de `nm poker`, on peut trouver des lignes analogues à :

```
00000000004041d0 T HandTest_findFlush
00000000004034f0 T HandTest_findQuads
0000000000403c50 T HandTest_findStraight
0000000000404930 T HandTest_findStrFlush
```

Rien n'indique le prototype des fonctions, mais on supposera que les noms de fonctions contenant le motif `Test_` sont des fonctions de test, sans argument et sans retour, correspondant au type défini par : `typedef void TestFunction (void)`.

On peut convertir la représentation texte hexadécimale d'un nombre en `unsigned long long`, via la fonction `strtoull()`. On peut ensuite convertir ce nombre en un pointeur sur fonction de type `TestFunction*` par le cast `(TestFunction*)`, et invoquer la fonction via ce pointeur. Par exemple, le programme `poker` peut lancer la fonction de test `HandTest_findFlush()` à partir de son adresse `00000000004041d0`.

Exercice 1. Écrire une fonction `void Tester_checkSingleTest(int argc, char * argv[])` qui vérifie si le programme a été lancé avec exactement les 3 arguments `-test address function`, où `address` est la représentation d'un nombre sur 16 chiffres hexadicimaux. Si ce n'est pas le cas, on ne fait rien. Si c'est le cas, on affiche sur la sortie d'erreur le message `"Testing"` suivi du nom de la fonction `function`. Celle-ci est ensuite lancée via le pointeur sur fonction `TestFunction*` obtenu à partir de `address`. En supposant que `assert()` n'a pas avorté le programme avec `abort()`, on affiche ensuite "OK" sur la même ligne et on termine le programme avec le status `EXIT_SUCCESS`. La fonction `Tester_checkSingleTest()` a vocation a être invoquée en début de `main()`. Par exemple, on pourrait avoir le source `autotest.c` suivant :

```c
// File: autotest.c
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>

void DummyTest_fail1 (void) { assert (false); }
void DummyTest_fail2 (void) { assert (false); }
void DummyTest_pass1 (void) { assert (true ); }
void DummyTest_pass2 (void) { assert (true ); }
void DummyTest_pass3 (void) { assert (true ); }
void DummyTest_runAll(void) { /* ... */ }

typedef void TestFunction (void);
void Tester_checkSingleTest (int argc, char * argv[]) { /*...*/ }

int main (int argc, char * argv[]) {
  Tester_checkSingleTest (argc, argv);
  fprintf (stdout, "Hello World!\n");
  return 0;
}
```

Un extrait de la table des symboles de l'exécutable `autotest` pourrait être :

```
$ nm ./autotest
...
0000000000400810 T Tester_checkSingleTest
0000000000400780 T DummyTest_fail1
00000000004007b0 T DummyTest_fail2
00000000004007e0 T DummyTest_pass1
00000000004007f0 T DummyTest_pass2
0000000000400800 T DummyTest_pass3
0000000000400930 T main
```

Et trois sorties de l'exécutable `autotest` seraient alors :

```
$ ./autotest
Hello World!
$ echo $?
0
```

```
$ ./autotest -test 00000000004007e0 DummyTest_pass1
Testing DummyTest_pass1: OK
$ echo $?
0
```

```
$ ./autotest -test 0000000000400780 DummyTest_fail1
Testing DummyTest_fail1: autotest: autotest.c: void DummyTest_fail1(): Assertion...
Aborted (core dumped)
$ echo $?
134
```

Rappelons que lorsque `assert()` échoue, il invoque `abort()`, qui envoie le signal `SIGABRT`.
La terminaison est anormale (interrompue par un signal), et il n'y a pas d'exit status.
Cependant, `bash` stocke dans la variable `$?` le numéro du signal augmentée de `128`.
Le signal `SIGABRT` ayant pour numéro `6`, la variable `$?` vaut donc `128+6= 134`.

Exercice 2. Avec un pipe sur `egrep`, comment filtrer la sortie de `nm autotest` pour n'obtenir que les fonctions de test définies (ayant une adresse), qu'elle soient statiques ou non (flag `T` ou `t`).

```
$ nm ./autotest | egrep ...
0000000000400790 T DummyTest_fail1
00000000004007d0 T DummyTest_fail2
0000000000400780 T DummyTest_pass1
00000000004007c0 T DummyTest_pass2
0000000000400800 T DummyTest_pass3
0000000000400810 T DummyTest_runAll
```

Avec `cut`, comment augmenter le pipe précédent pour supprimer le flag de la sortie ?

```
$ nm ./autotest | egrep ... | cut ...
0000000000400790 DummyTest_fail1
00000000004007d0 DummyTest_fail2
0000000000400780 DummyTest_pass1
00000000004007c0 DummyTest_pass2
0000000000400800 DummyTest_pass3
0000000000400810 DummyTest_runAll
```

Comment prolonger le pipe pour éliminer de la sortie les fonctions se terminant en `Test_runAll` ?

```
$ nm ./autotest | egrep ... | cut ... | egrep ...
0000000000400790 DummyTest_fail1
00000000004007d0 DummyTest_fail2
0000000000400780 DummyTest_pass1
00000000004007c0 DummyTest_pass2
0000000000400800 DummyTest_pass3
```

Enfin, prolonger le pipe avec une boucle `while` sur `read` pour lancer tous les tests.

```
$ nm ./autotest | ... | while read ADDR TEST ; do ... ; done
Testing DummyTest_fail1: autotest: autotest.c: void DummyTest_fail1(): Assertion...
Testing DummyTest_fail2: autotest: autotest.c: void DummyTest_fail2(): Assertion...
Testing DummyTest_pass1: OK
Testing DummyTest_pass2: OK
Testing DummyTest_pass3: OK
```

Exercice 3. On ne veut plus écrire ce pipeline. On veut donc que le programme prenne en charge ce travail, lorsqu'il est invoqué avec l'option `-testall`. Écrire une fonction `Tester_checkAllTests (int argc, char * argv[])` qui vérifie si le programme a été lancé à exactement l'argument `-testall`. Si ce n'est pas le cas, on ne fait rien. Si c'est le cas, on utilise `system()` pour lancer le pipeline vu précédemment qui lance tous les tests, et on termine le programme avec le status `EXIT_SUCCESS`. Utiliser `argv[0]` pour le nom de l'exécutable lorsqu'il apparaît dans le pipeline.

```c
void Tester_checkSingleTest (int argc, char * argv[]) { /*...*/ }
void Tester_checkAllTests   (int argc, char * argv[]) { /*...*/ }

int main (int argc, char * argv[]) {
  Tester_checkSingleTest (argc, argv);
  Tester_checkAllTests   (argc, argv);
  fprintf (stdout, "Hello World!\n");
  return 0;
}
```

```
$ ./autotest -testall
Testing DummyTest_fail1: autotest: autotest.c: void DummyTest_fail1(): Assertion....
Aborted (core dumped)
Testing DummyTest_fail2: autotest: autotest.c: void DummyTest_fail2(): Assertion...
Aborted (core dumped)
Testing DummyTest_pass1: OK
Testing DummyTest_pass2: OK
Testing DummyTest_pass3: OK
```

Remarque : les lignes `"Aborted (core dumped)"` sont affichées par le shell `sh -c cmd`.

Exercice 4. En bash, après avoir redirigé la sortie d'erreur de `./autotest -testall` vers la sortie standard, comment obtenir le nombre total de tests ? le nombre de tests passants ? le nombre de tests non passants ?

```
$ ./autotest -testall 2>&1 | ... # we have 5 total testss
5
$ ./autotest -testall 2>&1 | ... # we have 3 passed tests
3
$ ./autotest -testall 2>&1 | ... # we have 2 failed tests
2
```

Exercice 5. Écrire une variante `Tester_checkAllTestsWithPopen()` déclenchée par l'option `-testallwithpopen` en utilisant `popen()`/`pclose()` et une boucle sur `fgets()`/`fprintf()` au lieu de `system()`. Attention, `popen()` lit la sortie standard et non la sortie d'erreur, il faut donc modifier le pipeline pour rediriger sa sortie d'erreur sur la sortie standard. N'afficher que les lignes commençant par `Testing`. Afficher à la fin le ratio du nombre de tests passants sur le nombre total de tests.

```
$ ./autotest -testallwithpopen
Testing DummyTest_fail1: autotest: autotest.c: void DummyTest_fail1(): Assertion...
Testing DummyTest_fail2: autotest: autotest.c: void DummyTest_fail2(): Assertion...
Testing DummyTest_pass1: OK
Testing DummyTest_pass2: OK
Testing DummyTest_pass3: OK
Passed: 3 / 5
```

Exercice 6. Écrire une variante `Tester_checkAllTestsWithFork()` déclenchée par l'option `-testallwithfork` où `popen()` ne prend plus en charge la partie finale `"while read ADDR TEST; do ... ; done"` du pipeline. À la place, chaque test est lancé dans un processus enfant en faisant un `fork()` pour chaque couple `ADDR TEST` lu sur une ligne dans une boucle sur `fgets()`. Lorsque le parent attend la terminaison de l'enfant, utiliser `waitpid()` plutôt que `wait()` afin d'attendre spécifiquement l'enfant du fork, sinon on risque d'attendre par erreur celui créé par `popen()` alors que ce dernier doit être attendu par `pclose()`.

Exercice 7. Écrire une variante `Tester_checkAllTestsWithExec()` déclenchée par l'option `-testallwithexec`, modification de la précédente, où l'enfant n'invoque pas directement la fonction de test via le pointeur sur fonction, mais via l'option `-test` du programme et un recouvrement par `execl()`.

Exercice 8. Écrire une variante `Tester_checkAllTestsWithPipe()` déclenchée par l'option `-testallwithpipe`, modification de la précédente, où un pipe est créé avant le `fork()`. La sortie d'erreur de l'enfant est redirigée dans le pipe, et c'est le parent qui prend en charge l'affichage du diagnostic en lisant l'entrée du pipe. Utiliser `fdopen()` pour obtenir un `FILE*` à partir du descripteur d'entrée du pipe et boucler sur `fgetc()`.

Exercice 9. Modifier `Tester_checkAllTestsWithPipe()` pour tuer les tests lents (durant plus d'une seconde). Le parent utilise `alarm(1)` afin d'être interrompu si sa boucle de lecture du pipe dure plus d'une seconde. Le handler de l'alarme met à `true` une variable globale Booléenne `ALARMED` initialement fausse avant la boucle de lecture du pipe. Si l'alarme arrive pendant `fgetc()`, la fonction retourne `EOF` donc on sort de la boucle comme si l'on a rencontré une fin de flux normale. Mais il faut aussi prévoir que l'alarme peut arriver en dehors de `fgetc()` et donc boucler sur la lecture tant que `ALARMED` n'est pas activé. Si après la boucle de lecture, on détecte que `ALARMED` est activé, on tue le processus fils via `kill()` et le signal `SIGKILL` et on affiche `"too slow, killed"` sur la sortie d'erreur. Dans tous les cas, on désactive l'alarme avec `alarm(0)`. On peut utiliser `sleep(2)` dans des tests pour simuler un test lent.

```
$ ./autotest -testallwithpipe
...
Testing DummyTest_slow1: too slow, killed
Testing DummyTest_slow2: too slow, killed
Passed: 3 / 7
```