AUFGABE 5.3: SOFTWARE-ENTWURF UND -TEST – TEIL 3

In dieser Aufgabe werden Sie einen abstrakten Datentyp zur Verwaltung einer *Prioritätswarteschlange* implementieren und testen. Sie können hierbei davon ausgehen, dass Sie nicht den Einschränkungen eines eingebetteten Systems unterliegen – d. h. die Verwendung von `malloc` und `free` ist zulässig und Sie können die Details Ihrer Datentypen vollständig vor dem Anwender verbergen. Anschließend wird Ihre Warteschlangenimplementierung versendet und von anderen Gruppen getestet. Im Gegenzug testen Sie die Warteschlangenimplementierung weiterer Gruppen. Hierbei soll ein Informationsaustausch zu Ihrer Implementierung entstehen.

Hinweis: Beachten Sie, dass sich dieses Aufgabenblatt aus mehreren Teilblättern mit eigenen Abgabeterminen zusammensetzt.

1 Aufgabenstellung

Vermerken Sie Ihre Antworten zu den Fragen der einzelnen Aufgaben an den vorgesehenen Stellen in der vorgegebenen `answers.md`*. Bitte erstellen Sie, um die Abgabe durch Mergerequests zu vereinfachen,* ***pro Aufgabe einen eigenen Branch****. Um einen konsistenten Zustand zu gewährleisten, benutzen Sie dazu bitte folgenden Befehl:*

```
git fetch git@gitos.rrze.fau.de:i4/teaching/vezs/vezs24-vorgabe.git
aufgabe5 && git checkout -b aufgabe5 FETCH_HEAD
```

Werkzeuggestütztes Testen

Hinweis: Achten Sie darauf, die folgenden Tests wieder in Ihrem eigenen Build-Verzeichnis (nicht auf den fremden Implementierungen) durchzuführen!

Aufgabe 14 *Überdeckung*

Überprüfen Sie nun die erreichte Überdeckung mit Hilfe von `lcov`. Beachten Sie, dass `make lcov` erst erfolgreich aufgerufen werden kann, falls schon mindestens ein Testfall ausgeführt wurde, der Ihre Prioritätswarteschlange auch tatsächlich verwendet. *Welches Überdeckungskriterium überprüft* `lcov`*?* Versuchen Sie durch Ihre Testfälle eine möglichst hohe Testabdeckung zu erreichen; eine 100%ige Überdeckung ist (auch mittels der Hilfestellungen in `priority_queue_test2.c`) möglich. Achten Sie auf die korrekte Eingruppierung ihrer Tests.

☞ `make lcov`

Antwort:

Aufgabe 15 `AddressSanitizer`

Auch für den `AddressSanitizer` bietet es sich an, eine eigene Build-Umgebung aufzusetzen:

```
mkdir build-asan && cd build-asan && cmake -DCMAKE_BUILD_TYPE=Debug ..
```

Lassen Sie Ihre Testfälle auch mit aktiviertem `AddressSanitizer` laufen und korrigieren Sie ggf. auftretende Fehler. *Welche Fehler haben Sie gefunden, die ohne den* `AddressSanitizer` *nicht zu einem Fehlverhalten führen?*

Hinweis: Falls Ihre Lösung mittlerweile schon zu gut getestet ist, und glücklicherweise der `AddressSanitizer` nicht fündig wird, so möchten wir Sie bitten, den Test nochmals auf Ihrer initial versendeten Lösung des ersten Aufgabenteiles zu wiederholen. Wenn auch in dieser Variante keine Fehler auftreten, haben Sie entweder zu wenig Testfälle oder sehr gründlich gearbeitet.

Bitte fügen Sie die initiale Ausgabe als Commit ihrer Abgabe bei, und beschreiben Sie die zugrundeliegenden Fehler.

Antwort:

Aufgabe 16

Diese Fehler scheinen somit ja keine Auswirkung zu haben. *Wieso ist der Einsatz des* `AddressSanitizer`*s trotzdem sinnvoll?*

Antwort:

Aufgabe 17 `Clang Static Analyzer`

Testen Sie Ihr Programm mit dem `Clang Static Analyzer`. Dokumentieren Sie die Ergebnisse. Auch hierfür bietet es sich an, eine eigene Build-Umgebung zu

erstellen:
`mkdir build-csa && cd build-csa && CC=clang scan-build cmake ..`

Hinweis: Es bietet sich final an, sicherzustellen, dass der Analyzer keine Fehlerquellen übersieht, nur weil `make` auf Zwischenergebnisse zurückgreift, deren Quellcode dann nicht mehr analysiert wird. Zwingen sie dazu `make` mit der Option `-B` dazu, auch sämtliche Zwischenprodukte neu zu kompilieren (Voller Aufruf also `scan-build make -B`).

Wo liegen die Vor- und Nachteile dieses Ansatzes vor allem im Hinblick auf und im Vergleich mit den Ergebnissen des `AddressSanitizer`*s?*

Hinweis: Falls Ihre Lösung mittlerweile schon zu gut getestet ist, und somit der `Clang Static Analyzer` nicht fündig wird, so möchten wir Sie bitten, den Test nochmals auf Ihrer initial versendeten Lösung des ersten Aufgabenteiles zu wiederholen. Wenn auch in dieser Variante keine Fehler auftreten, haben Sie offensichtlich sehr gründlich gearbeitet.

Bitte fügen Sie die initiale Ausgabe als Commit ihrer Abgabe bei, und beschreiben Sie die zugrundeliegenden Fehler.

Antwort:

Aufgabe 18 *Nachbereitung*

Welche Aspekte Ihres ursprünglichen Entwurfs haben sich im Verlauf der Implementierung als unzureichend erwiesen? Wie haben Sie diese Probleme behoben?

Antwort:

Erweiterte Aufgabe

Aufgabe 19 *Iteratorkonsistenz*

Konzeptionell ist es möglich, dass Iteratoren inkonsistente Werte liefern, wenn die Warteschlange, über die Sie iterieren, währenddessen, beispielsweise durch das Hinzufügen eines neuen Elements, verändert wird. Stellen Sie sicher, dass Ihre

Implementierung ein konsistentes Iteratorverhalten gewährleistet, indem Sie bei Veränderungen an der Warteschlange alle aktiven Iteratoren invalidieren.

Aufgabe 20 *Iteratorkonsistenz Testen*

Stellen Sie auch hier, wie auch schon in Aufgabe 14 vollständige Testüberdeckung gemäß lcov her.

Aufgabe 21 *Fuzzing*

Machen Sie sich mit *Fuzzing* als Technik zum Testen von Programmen mit zufällig generiertem Input vertraut. In der Vorgabe finden Sie eine Implementierung eines Brainfuck-Interpreters. Der Interpreter enthält einen Bug. Finden Sie diesen mittels Fuzzing unter Verwendung der Clang-Erweiterung. Implementieren Sie hierfür die vorgegebene Schnittstelle für den Fuzzer, die die vom Fuzzer erzeugte Eingabe als Brainfuck-Programm interpretiert. Sie können den Fuzzer mittels des make targets `make fuzz` starten.

Beheben Sie den gefundenen Fehler.

☞ https://en.wikipedia.org/wiki/Brainfuck

☞ bfi.cpp

☞ libFuzzer

☞ LLVMFuzzerTestOneInput

Hinweise

- Bearbeitung: Gruppenarbeit
- Abgabefrist: 10.01.2025
- Fragen bitte an `i4ezs@lists.cs.fau.de`