AUFGABE 5: SOFTWARE-ENTWURF UND -TEST

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 einer anderen Gruppe getestet. Im Gegenzug testen Sie die Warteschlangenimplementierung einer weiteren Gruppe. Hierbei soll ein Informationsaustausch zu ihrer Implementierung entstehen. Zusätzlich werden Sie Ihre eigene Implementierung auch noch mit Hilfe feingranularerer Testfälle testen, um mindestens vollständige Codeüberdeckung zu erreichen.

Hinweis: Beachten Sie, dass es für dieses Aufgabenblatt einen Zwischenabgabetermin gibt. Das Versenden Ihrer Warteschlangenimplementierung (siehe Aufgabe 9) muss bis zum **17.06.2019** erfolgen!

Aufgaben

Aufgabe 1 *Kommunikation mit anderen Gruppen*

Für einige Aufgaben dieses Aufgabenblattes (9 11 12) werden Sie mit anderen Gruppen zusammenarbeiten. Hierfür sollten Sie zwei Mails erhalten haben, die Sie jeweils einem Projekt für die Entwicklung und die Qualitätsssicherung zuordnen. Melden Sie sich, falls Sie **keine** Emails erhalten haben sollten! Genaure Hinweise zur Verwendung der Kommunikationsinfrastruktur finden Sie in besagten Emails.

Aufgabe 2 *Arbeitsumgebung*

Initialisieren Sie die Vorgabe wie in den Übungsfolien besprochen. Machen Sie sich mit der Konfiguration, den vorgegebenen Dateien und den generierten Dateien vertraut.

☞ **cd** `build`
☞ cmake ..

Aufgabe 3 *Programmschnittstelle*

Machen Sie sich mit der Schnittstelle und dem gewünschten Verhalten der Warteschlange vertraut. Generieren Sie hierfür die doxygen-Dokumentation der vorgebenen C-Schnittstelle. Anschließend finden Sie unter `doc/html/index.html` die

☞ `make doxy`
☞ `priority_queue.h`

Dokumentation der einzelnen Funktionen sowie eine Spezifikation der Warteschlange in Fließtext.

Aufgabe 4 *Fehlerraumanalyse*

Skizzieren Sie die Fehlerhypothese. *Welche Randfälle müssen Sie berücksichtigen und warum?*

Antwort:

Aufgabe 5

Erstellen Sie nun Funktionsstümpfe, also eine Implementierung in korrektem C ohne irgendeine Funktionalität, für die Funktionen der Schnittstelle, so dass das Projekt ohne Fehler übersetzt werden kann. Sichern Sie nun den aktuellen Stand Ihres Projekts, in dem Sie einen git-Commit erstellen.

Aufgabe 6 *Testfallentwurf*

Entwerfen Sie nun auf Basis Ihrer Analyse und der identifizierten Randbereiche Testfälle, die geeignet sind zu zeigen, dass die Implementierung Ihres Softwareentwurfs den Anforderungen entspricht und auch sonst kein nicht definiertes Verhalten besitzt. *Welche Kategorien von Testfällen gibt es?*

Antwort:

Implementierung

Aufgabe 7 *Aufteilung*

Setzen Sie nun den Softwareentwurf und den Testfallentwurf um. Hierbei sollten die Testfälle für eine Funktion nicht von der gleichen Person geschrieben werden, die

diese Funktion implementiert. Teilen Sie die Implementierungsarbeit innerhalb Ihrer Gruppe in möglichst unabhängige Arbeitspakete auf. Die Verwendung verschiedener Git-Branches kann diese Aufteilung erleichtern. *Wieso ist diese Herangehensweise sinnvoll? Dokumentieren Sie die Arbeitsaufteilung!*

Antwort:

Aufgabe 8 *Testfälle*

Für die Implementierung der Testfälle sollen Sie die Testumgebung von `cmake` verwenden. Legen Sie hierbei pro Testfall eine eigene `.c`-Datei mit `main()`-Funktion im Unterverzeichnis `tests` an. Tragen Sie Ihre Tests in der Konfigurationsdatei `tests/CMakeLists.txt` in die passende der drei Testfalllisten ein. Die Bedeutung der einzelnen Listen (`EZS_PQ_GENERAL_TESTS`, `EZS_PQ_MALLOC_TESTS`, `EZS_PQ_OWN_ONLY_TESTS`) ist in der Konfigurationsdatei dokumentiert.

☞ `make test`

Hinweis: Das von `cmake` erzeugte `test`-Target führt nur die Tests aus, ohne Ihr zu testendes Programm neu zu kompilieren. Es bietet sich daher an, Tests mittels `make && make test` aufzurufen.

Welche Anforderungen sind an gute Testfälle zu stellen?

Antwort:

Aufgabe 9 *Warteschlange*

Implementieren Sie die nötige Funktionalität für die vorgegebene Schnittstelle der Prioritätswarteschlange. **Hinweis:** Es bietet sich an, die Warteschlange als einfach verkettete Liste zu implementieren. Packen Sie Ihre fertige Implementierung in ein `tar`-Archiv und senden Sie dieses zum Testen an Ihre zugeteilte Testgruppe (`qa`, siehe Aufgabe 1).

☞ `make pack-review`

Termin: 17.06.2019

Fremdtests

Aufgabe 10 *Testumgebung*

Um gleichzeitig die Fremdbibliothek testen und die weiteren Aufgaben bearbeiten zu können, bietet es sich an, ein separates Build-Verzeichnis zu erstellen. Die Option `BUILD_ALIEN_TEST` muss gesetzt sein, um unsere Testinfrastruktur nutzen zu können. Diese trifft unter anderem gewisse Sicherheitsvorkehrungen zum Ausführen fremden Codes, ist also dringend notwendig.

Setzen Sie eine neue Buildumgebung mittels folgender Befehle auf:

```
mkdir build-other && cd build-other
cmake .. -DBUILD_ALIEN_TEST=ON -DBUILD_WITH_COVERAGE=OFF
```

Innerhalb dieses Verzeichnisses können Sie die Fremdimplementierung nun wie gewohnt mittels `make test` testen.

Aufgabe 11 *Testen*

Sie haben die Implementierung der Warteschlange von einer zufällig zugeteilten Gruppe erhalten. Speichern Sie diese in Ihrem Quellcode-Verzeichnis unter `review/lib_priority_queue_alien.a`. Testen Sie diese Implementierung mit Hilfe Ihrer Testfälle und teilen Sie der anderen Gruppe über die in Aufgabe 1 vorgestellte Mailinfrastruktur die Ergebnisse mit.

☞ `make test`

Sollten Sie von der Gruppe eine überarbeitete Implementierung erhalten, wiederholen Sie bitte das Vorgehen, bis alle Tests erfolgreich verlaufen.

Aufgabe 12 *Implementierung*

Sie haben von der Ihnen zugeteilten Testgruppe Rückmeldung über die Testergebnisse Ihrer Warteschlage erhalten. Falls nötig, beheben Sie diese Fehler und senden Sie Ihre Implementierung erneut an die Testgruppe.

Testen

Hinweis: Achten Sie darauf, die folgenden Tests wieder in Ihrem eigenen Build-Verzeichnis durchzuführen!

Aufgabe 13 *Überdeckung*

Überprüfen Sie nun die erreichte Überdeckung mit Hilfe von `lcov`. Beachten Sie, dass `make lcov` erst erfolgreich aufgerufen werden kann, falls schon mindestens

☞ `make lcov`

ein Testfall ausgeführt wurde, der Ihre Prioritätswarteschlange auch tatsächlich verwendet. *Welches Überdeckungskriterium überprüft* `lcov`*?* Schreiben Sie mindestens so viele Testfälle, dass `lcov` eine 100%ige Überdeckung anzeigt.

Antwort:

Aufgabe 14 `AddressSanitizer`

Auch für den `AddressSanitizer` bietet es sich an, eine eigenen 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?*

Antwort:

Aufgabe 15

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

Antwort:

Aufgabe 16 `Clang Static Analyzer`

Testen Sie Ihr Programm mit dem `Clang Static Analyzer`. Dokumentieren Sie die Ergebnisse. *Wo liegen die Vor- und Nachteile dieses Ansatzes vor allem in Hinblick auf den* `AddressSanitizer`*?*

Hinweis: Aufgrund eines Fehlers in der Paketierung von `scan-view-3.8` unter Debian muss im CIP vor dem Starten von `scan-view` die Variable `PYTHONPATH` angepasst werden:

```
export PYTHONPATH="$PYTHONPATH:/usr/share/clang/scan-view-3.8/share"
scan-view /tmp/scan-build-...
```

Antwort:

Aufgabe 17 *Nachbereitung*

Welche Aspekte Ihres ursprünglichen Entwurfs haben sich im Verlauf der Implementierung als unzureichend erwiesen?

Antwort:

Aufgabe 18

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 *Fuzzing*

In der Übung haben Sie *Fuzzing* als Technik zum Testen von Programmen mit zufällig generiertem Input kennengelernt. In der Vorgabe finden Sie eine Implementierung eines Brainfuck-Interpreters. Der Interpreter enthält einen Bug. Finden Sie diesen mittels Fuzzing unter Verwendung der in der Übung vorgestellten 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.

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

☞ bfi.cpp

☞ libFuzzer

☞ LLVMFuzzerTestOneInput

Beheben Sie den gefundenen Fehler.

Aufgabe 21 *Fuzzing mal anders*

Benutzen Sie den Fuzzer, um sich ein Brainfuck-Programm generieren zu lassen, das die Zeichenkette „VEZS19“ ausgibt. Identifizieren Sie hierfür eine geeignete Stelle im Interpreter, an der Sie die Ausgabe überprüfen können. Erweitern Sie den Interpreter, analog zum in der Tafelübung vorgestellten Vorgehen, um eine Überprüfung, die, falls die Zeichenkette gefunden wurde, die Ausführung mit einem Fehler terminiert.

☞ __builtin_trap()

Hinweise

- Bearbeitung: Gruppe mit je zwei bis drei Teilnehmern.
- Abgabefrist: 25.06.2019
- Fragen bitte an `i4ezs@lists.cs.fau.de`