Bug-freies Programmieren

Das Ziel jeder Softwareentwicklung ist, fehlerfreie Programme zu erzeugen. Was die Businesslogik eines Programms betrifft (also die Funktionalität auf der Ebene der Anwendung) ist das noch *relativ* leicht zu verifizieren und sicherzustellen. Die gefährlichen und heimtückischen Fehler schleichen sich auf einer anderen Ebene ein: dem Umgang des Programmierers mit der Programmiersprache.

Die hier erzeugten „Bugs“ entstehen nicht dadurch, dass z.B. ein Algorithmus falsch abgebildet wurde. Sie resultieren vielmehr aus einer nicht ganz sachgerechten Verwendung der Möglichkeiten, die von der jeweiligen Sprache geboten werden. Das kann zum Beispiel dann passieren, wenn der Programmierer über nur ungenügende Kenntnisse der Arbeitsweise der Sprache verfügt, ihre Besonderheiten außer Acht lässt oder einfach nur unkonzentriert arbeitet.

Im Folgenden finden Sie eine Zusammenstellung jener Bereiche, die am anfälligsten für fehlerhaften Code sind. Darüber hinaus werden Hinweise geboten, wie Fehler dort am Besten zu vermeiden sind.

1. Übersicht der kritischen Bereiche

Abgesehen von Fehlern in der Businesslogik treten Programmierfehler vor allem in folgenden Bereichen auf:

Resource Management
Unter Resources wird alles verstanden, was von einem Programm während der Laufzeit belegt und wieder freigegeben wird, also zum Beispiel Speicher, Massenspeicher, Datenbank-Verbindungen, etc...

Exception Handling
Exceptions sind Mechanismen moderner Programmiersprachen, um Fehler abzubilden. Sie funktionieren im Wesentlichen so, dass statt der Rückgabe eines Fehlercodes das Programm zu einer Stelle springt, an der ein Codestück zur Fehlerbehandlung steht.

Multithreading
Multithreading bezeichnet die parallele Verarbeitung von zwei oder mehreren Prozessen, den so genannten Threads, die sich ein Datensegment teilen.

Buffer Overflows
Programmiersprachen bieten für die verschiedensten Zwecke Buffer sowie Funktionen zu deren Beschreibung an. Nicht alle dieser Funktionen schützenautomatisch vor einem Überlaufen der Buffer.

2. Resource Management

Unter Resource Management wird sowohl das Belegen und Freigeben von Resources als auch deren effiziente Verwaltung (im Hinblick auf den Footprint des Programms als auch im Hinblick auf die Laufgeschwindigkeit) verstanden.

Unter einer Resource selbst versteht man unter anderem: Hauptspeicher (private oder process-shared), Festplattenspeicher, Datenbankhandles, Betriebssystem-objekthandles – also generell alles, was ein Programm benötigt, um zu laufen, was es aber erst zur Laufzeit belegen kann. Das Hauptaugenmerk liegt hierbei meist auf dem Hauptspeicher.

Unter Footprint versteht man die Summe aller zu einem Zeitpunkt belegter Resources - je größer dieser ist, umso weniger Platz haben andere Programme, die zur selben Zeit am selben Rechner laufen. Ferner kann ein zu großer Memory Footprint ungünstige Auswirkungen auf die Performance haben: es kann zur schlechten Ausnutzung des Prozessor Cache (langsame Speicherzugriffe) oder gar zur Auslagerung in den virtuellen Speicher auf der Festplatte kommen (sehr langsame Speicherzugriffe).

Generell sollte jede Resource, die belegt wird, irgendwann auch wieder freigegeben werden - und zwar möglichst früh, um den Footprint des Programms klein zu halten. Wenn Resources gar nicht mehr freigegeben werden, spricht man von Resource Leaks. Bei Programmiersprachen, die einen Garbage Collector besitzen (wie etwa Java oder Smalltalk), passiert dieses Freigeben etwa für Hauptspeicher normalerweise automatisch, d.h. der Programmierer sollte damit nicht belastet sein.

Dies ist jedoch nur die halbe Wahrheit. Folgendes muss berücksichtigt werden:

Erstens befassen sich Garbage Collectors meistens nur mit dem Hauptspeicher, nicht jedoch mit anderen Resources, wie etwa Datenbankhandles - solche Resources müssen in jedem Fall manuell wieder freigegeben werden.

Zweitens gibt es Resources (wie etwa Filehandles) die freigegeben ("geschlossen") werden, sobald das zugehörige Objekt vom Garbage Collector freigegeben wird. Es ist jedoch generell nicht vorhersehbar, *wann* der Garbage Collector das File Objekt tatsächlich freigibt. Da ein nicht geschlossenes File nun aber schlimmere Auswirkungen als nur einen erhöhten Footprint des Programms hat, wie etwa File Locks (d.h. andere Programme können auf dieses File nicht zugreifen), ist es normalerweise nötig, das Schließen des Filehandles selbst in die Hand zu nehmen anstatt es dem Garbage Collector zu überlassen.

In beiden oben genannten Fällen muss besonders auf Exceptions Rücksicht genommen werden: Exceptions verursachen nichtlokale Programmflüsse, d.h. bestimmte Codestellen werden einfach "überflogen". Um nun zu verhindern, dass der Code "überflogen" wird, der die Handles freigibt, gibt es in den genannten Programmiersprachen so genannte Final Handler. Das sind Codeblöcke, die selbst beim Auftreten von Exceptions ausgeführt werden - der Einsatz dieser Final Handler ist hier also extrem wichtig!

Drittens muss man auch beim Hauptspeicher, der eigentlichen Domäne des Garbage Collectors, aufpassen: Auch hier gibt es die Möglichkeit, dass Memory Leaks auftreten, und zwar in Form nicht genutzter Referenzen auf den fraglichen Speicher. D.h. irgendwo im Programm gibt es noch Variablen, die auf diesen Speicher zeigen, obwohl er nicht mehr benötigt wird. Als bestes Gegenmittel empfiehlt sich hier das möglichst frühzeitige Setzen von Variablen auf den definierten Null-Wert.

Generell kann festgehalten werden, dass in Programmiersprachen mit Garbage Collectors die Versuchung sehr groß ist, auf ein korrektes Resource Management zu vergessen. Es ist eine Frage des Bewusstseins, dass auch in solchen Fällen von Seiten des Programmierers Wert auf korrektes Resource Management zu legen ist.

Programmiersprachen, die von vornherein keinen Garbage Collector besitzen, wie etwa C++, stellen einen anderen Fall dar. Wiederum gilt, und nun in verschärftem Maß, dass auf das frühzeitige Freigeben von Resources zu achten ist. Und wiederum muss man auf nichtlokale Programmflüsse (durch Exceptions verursacht) aufpassen. Eine der Schwierigkeiten hierbei ist, dass es in manchen Sprachen, wie etwa C++, keinen Final Handler gibt - man muss sich hier mit entsprechenden try/catch clauses behelfen. D.h. man muss darauf achten, dass die Resource sowohl bei normalem als auch bei nichtlokalem Programmfluss freigegeben wird.

Eine Variante, mit diesem Thema umzugehen, ist die Verwendung von so genannten Smart Pointers. Dabei handelt es sich um Pointer, die einen Referenzzähler verwalten, der angibt wie oft die Resource (auf die sie zeigen) schon verwendet wird. Wenn dieser Zähler auf 0 geht, wird die Resource automatisch freigegeben. Die weitgehende Verwendung von solchen Smart Pointers kommt fast schon einem Garbage Collector gleich - wiewohl es Fälle gibt, wie etwa zyklische Referenzen, mit denen Smart Pointer nicht umgehen können und die daher zu vermeiden sind. Ein Vorteil der Smart Pointer gegenüber Garbage Collectors ist die Tatsache, dass das Freigeben der Resource mehr der Kontrolle des Programmierers unterliegt. Garbage Collectors haben weitgehende Freiheit, wann und *ob* sie überhaupt laufen - wenn etwa ein Programm beendet wird, kann es sein, dass der Garbage Collector erst gar nicht mehr aufgerufen wird - nicht so bei Smart Pointers.

Eine weitere Variante ist die Verwendung von Wrapper Objects nach dem "Resource allocation is acquisition" Schema, wie von Bjarne Stroustrup definiert. Es handelt sich dabei um Objekte, die auf den Stack gelegt werden, und deren einziger Sinn darin besteht, eine Referenz auf die Resource zu halten. Sobald die Objekte vom Stack wieder entfernt werden (bei Beendigung eines Funktionsaufrufes oder beim Verlassen eines Codeblockes) wird der sog. Object Destructor aufgerufen, der nun die assoziierte Resource freigibt. Mit dieser Technik ist man (ebenso wie mit Smart Pointers), auch auf nichtlokale Programmflüsse entsprechend vorbereitet. Ein Beispiel ist etwa das C++ Objekt "fstream". Wenn ein "fstream" Objekt vom Stack entfernt wird, wird auch das darunter liegende File geschlossen. Ganz im Gegensatz zu Java, wo man nie genau sagen kann, wann das Fileobjekt vom Garbage Collector entfernt wird, passiert das bei C++ an einem wohl definierten Punkt und obliegt somit voll und ganz der Kontrolle des Programmierers - ein explizites Schließen des Files ist somit unnötig.

3. Exception Handling

Exceptions sind ein Mechanismus, um unter außergewöhnlichen Fehlerzuständen einen nichtlokalen Programmfluss zu triggern. D.h. kurz gesagt, es wird ohne weitere Umwege der Errorhandling Code ausgeführt - der "normale" Code wird in großen Zügen übersprungen.

Daraus ergibt sich auch schon das Hauptproblem mit Exceptions: Wenn der Programmierer nicht damit rechnet, dass Teile seines Codes nicht ausgeführt werden, kann es zu Resource Leaks oder zu inkonsistenten Programmzuständen kommen (d.h. Programmzustände, die nicht den Erwartungen oder Annahmen des Programmierers entsprechen). Technisch gesagt heißt das, dass Programminvarianten verletzt werden können. Diese Verletzungen können soweit gehen, dass das Programm in Folge abstürzt bzw. generell undefiniertes Verhalten zeigt.

Um dem entgegenzuwirken, sollte man sich bei jeder Funktion oder Methode überlegen, welche Programminvarianten existieren (müssen). Während nun die Funktion/Methode abläuft, kann oder wird es dazu kommen, dass diese Invarianten vorübergehend verletzt werden. Es sollte nun sichergestellt werden, dass in Codeabschnitten, während deren Ablauf die Invarianten verletzt werden, keineExceptions auftreten. Oft läuft diese Vorgehensweise darauf hinaus, Änderungen an Programmdaten hintan zu halten bis alle Operationen, die eventuell Exceptions generieren können, erledigt sind (alle Invarianten sollten noch erfüllt sein). Danach werden in einem "exception atomischen" Codeabschnitt alle Programmdaten entsprechend geändert (sodass danach wiederum die Invarianten erfüllt sind). "exception atomisch" bedeutet, dass keine Exceptions in diesem Codeabschnitt generiert werden können.

Diese Methode exception safe zu programmieren ist relativ aufwändig - sie erfüllt das so genannte "Strong Promise", was übersetzt heißt: die entsprechenden Codeteile versprechen, dass im Falle von Exceptions das Programm in einem wohl definierten Status ist, in dem alle Invarianten erfüllt sind - es kann also nach Auftreten einer Exception normal weiterlaufen (sofern der Fehler dies zulässt).

Einfacher zu erfüllen ist das sog. "Weak Promise". Es besagt, dass nach Auftreten einer Exception nicht unbedingt alle Invarianten erfüllt sind, mindestens jedoch jene, die sicherstellen, dass alle Objekte im Programmprozessraum erfolgreich gelöscht werden können. Das ist insbesondere in nicht-Garbage-Collector-basierten Programmiersprachen wie C++ besonders wichtig. Wiederum kann die oben beschriebene Technik angewendet werden, jedoch muss nur auf einen Teil der Invarianten Rücksicht genommen werden - nämlich jene, die ein erfolgreiches Ablaufen der Destruktoren ermöglicht.

Bei allen oben genannten Fällen ist natürlich auch wieder besonders auf die korrekte Freigabe von Resources zu achten.

Ein C++ spezifisches Problem im Zusammenhang mit Exception Handling soll auch nicht verschwiegen werden. Wenn man gewisse Grundregeln nicht beherzigt, kann es zur (aus Sicht des Programmierers) unerwarteten Beendigung des Programms kommen, und zwar insbesondere wenn:

• in Destruktoren Exceptions ausgelöst werden - wann immer möglich sollte man so etwas schon im Programmdesign verhindern.

• Exceptions ausgelöst werden, die in einer throws Spezifikation einer Funktion nicht angegeben sind. Das kann in C++ passieren, und um es zu vermeiden haben sich führende Köpfe mehr und mehr der Meinung angeschlossen, throws Spezifikationen in C++ überhaupt nicht zu verwenden. Dieser Empfehlung schließt sich der Autor an.

4. Multithreading (“don´t try this at home”)

Multithreading wurde lange Zeit, völlig zu Unrecht, als relativ leicht angesehen. Inzwischen weiß man, dass korrekte Multithreading Programmierung zu einer derschwierigsten Übungen überhaupt zählt.

Folgende potentielle Fehlerquellen und Problembereiche existieren unter anderem:

• *Deadlocks*: Mehrere Threads blockieren sich gegenseitig, da sie alle aufeinander warten (über Umwege - vergleichbar der Situation an einer Kreuzung, an der von allen 4 Seiten Autos eingefahren sind, die aber nicht weiterkommen, weil sie jeweils vom Querverkehr behindert werden).
• *Race Conditions*: Je nachdem, welcher Thread zuerst welche Aufgabe erfüllt, kann es sein, dass das Programm manchmal richtig läuft und manchmal nicht. Es kann zu unerklärlichen, nur selten auftretenden Abstürzen und anderen Fehlfunktionen kommen.
• *Concurrency Situtations*: Mehrere Threads versuchen zur selben Zeit dieselbe Resource zu modifizieren - undefiniertes Programmverhalten (wie etwa Abstürze) kann daraus resultieren.
• *Unklarer Programmstatus*: Da zu jedem Zeitpunkt parallel mehrere Threads laufen, die jeweils völlig unterschiedlich die Programmvariablen modifizieren und abfragen, ist der Programmstatus zu einem gegebenen Zeitpunkt oft mehr das Produkt des Zufalls als eines geplanten Vorgangs - die Statusverwaltung ist nicht zentral.
• *Cache Abgleich*: Variablen werden von Threads, die auf verschiedenen CPUs laufen, geshared, wurden aber nicht als Volatile markiert. Jede CPU hat nun ihre eigene Kopie der Variablen im Cache und die beiden "Weltsichten" divergieren; Datenaustausch von Thread 1 an Thread 2 über eine solche Variable funktioniert manchmal schon, manchmal nicht.
• *Sehr schwer zu Testen*: Da ein und dieselbe Situation kaum zweimal auftritt (siehe Unklarer Programmstatus, Race Conditions), sind Fehler nur äußerst schwer nachvollziehbar. Oft läuft ein Programm auf Single CPU Rechnern fehlerfrei, auf Multi CPU Rechnern jedoch nicht.

Wie bringt man nun solche Probleme mit Multithreading (MT) in den Griff?

Lösung 1: Nicht multithreaded programmieren. Das ist zugleich die einfachste und effizienteste Lösung. Wann immer es nicht *unbedingt* nötig ist, sollte man die Finger davon lassen. Für viele Einsatzbereiche gibt es Alternativen. Server etwa können Multiplexingtechniken verwenden. Ein positiver Seiteneffekt hier ist potentiell höhere Performance im Fall, dass nur eine physische CPU vorhanden ist, da Contextswitches wegfallen.

Lösung 2: Autonome Objekte. Man dekomponiert das Problem in eine Menge von Objekten, die jeweils eine Resource verwalten und jeweils einen Thread haben. Zwischen den Objekten gibt es einen asynchronen Kommunikationskanal, über den sie Messages austauschen. Semaphoren, Mutexes, Critical Sections und ähnliche MT Mechanismen werden in den Objekten *nicht* verwendet (nur der Kommunikationskanal benötigt sie - er wird einmal programmiert und getestet und ist unabhängig von der Businesslogik). Diese Technik kann Performanceprobleme verursachen - wenn sie jedoch genügend Performance erzielt, ist sie relativ narrensicher. Sie wurde unter anderem auch unter dem Namen "Quantum Programming" von Miro Samek für Embedded Systems beschrieben.

Lösung 3: Wenn tatsächlich klassische MT Programmierung vonnöten ist, gibt es eine Reihe von Regeln, die einem das Leben erleichtern können. Dazu gehören unter anderem:

• Möglichst wenig Kommunikation über das Datensegment (Variablen) - eher Interprozess Mechanismen wie Named Pipes oder einen unter Lösung 2) beschrieben Kommunikationskanal verwenden.
• Möglichst wenig Berührungspunkte zwischen den Threads vorsehen.
• Variablen, die von mehreren Threads verwendet werden, entsprechend als Volatile markieren; mittels Synchronisationsmechanismen wie Mutex, Critical Section exklusiven Zugriff garantieren.
• Möglichst wenige Objekte teilen - einfacher ist es, Objekte einem Thread zuzuordnen und die Kommunikation vermehrt über integrale Daten und/oder PODS (Plain Old Datatypes i.e. Strukturen) abzuhandeln - das vereinfacht das Verständnis der Vorgänge enorm.
• Prozesse anstatt Threads verwenden. Dieser Ansatz skaliert oft auch besser, da Heap Contingency vermieden wird (das gegenseitige Warten bei Zugriff auf den geteilten dynamischen Speicherbereich, auch Heap oder Free Store genannt).

Abschließend muss nochmals betont werden: Multithreading ist ein sehr kompliziertes und "giftiges" Thema. Setzen sie nur erfahrene Programmierer an Codeteile, die Multithreading beinhalten! Es handelt sich hier um einen typischen Fall von: "Kids don't try this at home."

5. Buffer Overflows

Ein Thema, das vor allem Low-level Sprachen wie C (und auch C++) betrifft und in letzter Zeit eine völlig neue Bedeutung durch so genannte Buffer Overflow Exploits erlangt hat, sind Buffer Overflows.

Der Grundtenor hier ist: Wann immer Code Daten in einen Buffer schreibt und nicht auf die Länge des Buffers achtet, besteht die Gefahr, dass Daten, die hinter dem Buffer liegen, überschrieben werden. Das nutzen findige Hacker inzwischen aus um auch das Stacksegment und somit auch die Rücksprungadresse zu überschreiben, womit es ihnen gelingt, eigenen Code zur Ausführung zu bringen und das Programm somit zu übernehmen. Das kann leicht verhindert werden, wenn man *immer* überprüft, ob man nicht über das Ende des Buffers schreibt. Leider gibt es einige C Library Funktionen, die das nicht selbst erledigen, darunter:

- sprintf
- vsprintf
- strcat
- strcpy

Ferner gibt es Funktionen wie etwa "strncpy", welche die Länge Buffers zwar honorieren, jedoch keinen 0 Terminator einfügen - das kann besonders von Anfängern übersehen werden und dazu führen, dass (unbeabsichtigt) Daten, die hinter dem Buffer liegen, ausgelesen und in Folge vielleicht sogar (ebenfalls unbeabsichtigt) verändert werden.

Empfehlung: Moderne Compiler weisen Abarten der oben genannten Funktionen auf, die auf die Länge des Buffers Rücksicht nehmen (etwa "_snprintf" bei Visual C++). Diese sollten *immer* anstelle der originalen Funktionen verwendet werden, falls die Originale keine Länge des Buffers entgegennehmen und honorieren. Die Längen des Buffers selbst sollten immer vermittels “sizeof“ vom Compiler generiert werden und nicht hardcodiert werden. Für “strncpy“ ähnliche Funktionen empfiehlt es sich, einen eigenen Ersatz zu schreiben, der die terminierende 0 in jedem Fall einfügt.

In diesem Zusammenhang empfiehlt sich besonders der Einsatz von Tools zur statischen und dynamischen Analyse des Programms, wie etwa Lint oder Bounds Checker.

Bei der statischen Analyse wird der Sourcecode untersucht und auf offensichtliche Fehler hingewiesen. Das Problem dabei ist oft eine relativ lange und mühselige Konfigurationsphase, in der man alle unerwünschten Warnungen ausschaltet. Dabei muss man besonders aufpassen, das Kind nicht mit dem Bad auszuschütten - schnell hat man unbeabsichtigt die Warnung ausgeschaltet, die auf einen echten Programmfehler hinweisen würde.

Tools wie Bounds Checker untersuchen das Programm zur Laufzeit und weisen sowohl auf Buffer Overruns als auch auf Memory Leaks hin. Ferner kann mit einigen dieser Tools auch die korrekte Verwendung von APIs validiert werden.

Generell kann empfohlen werden, auf die Verwendung von solchen Buffers wann immer möglich (unter Einbeziehung von Performanceüberlegungen) zugunsten von höherwertigen Konstrukten (wie etwa Klassen von Buffers, die automatisch nach Bedarf zusätzlichen Speicher belegen) zu verzichten. Zum Beispiel sollte in C++ die Klasse std::string einem Character Buffer vorgezogen werden.

Ergänzend zu diesem Text steht Ihnen auf der InfraSoft Website ein Glossar zur Verfügung, in dem Sie die meisten der hier verwendeten Begriffe finden. Über das aktuelle Angebot an weiteren, kostenlosen Fachbeiträgen zur Softwareentwicklung informieren Sie sich bitte unter www.infrasoft.at/service.

Harald Nowak
Wien, im April 2003

Der Autor ist Mitarbeiter der InfraSoft, einem Unternehmen, das auf komplexe Softwareentwicklungen spezialisiert ist. Die Experten der InfraSoft haben langjährige Erfahrungen in der Entwicklung und verfügen über fundierte Kenntnisse in Design, Analyse, Realisierung, Test und Projektmanagement.
Für **individuelle Beratungen** zur Entwicklung von Softwarelösungen und die Bereitstellung von **Realisierungsteams** wenden Sie sich bitte an info@infrasoft.at.