Softwaretechnik Übung: Debugger und Fehlersuche

Klaus Kusche

1. **Core Dumps & gdb:**

   - Schreib ein kleines Programm mit Rekursion und viel CPU-Verbrauch (z.B. Binomialkoeffizienten oder Fibonacci-Zahlen rekursiv berechnen), compiliere es mit Debug-Information, starte es und brich es mit SIGQUIT ab (SIGQUIT wirkt wie Ctrl/C, aber mit Core Dump. Mit **`stty -a`** kannst du herausfinden, welche Tastenkombination SIGQUIT auslöst).

     Schau dir den Callstack im Coredump an (mit dem Commandline-**`gdb`** und mit dem grafischen Debugger **`nemiver`**).

   - Versuche im **`gdb`** (auf der Commandline), außer dem Callstack auch einzelne Variablen auf verschiedenen Aufrufsebenen anzuzeigen.

   - Teste auch andere Wege, die zu einem Dump führen:

     - Bau eine Nullpointer-Dereferenz in dein Programm ein. Lass ein solches Programm auch mit **`catchsegv`** laufen.

     - Experimentiere mit den C-Funktionen **`abort`** und **`assert`**.

   - Kannst du mit **`gdb`** oder **`nemiver`** ein schon laufendes Programm debuggen?

2. **Debugger:**

   Such dir aus meinen Musterlösungen ein Programm mit Funktionen und Pointern (z.B. unser Tree-Beispiel aus dem 3. Semester), übersetze es mit Debug-Information, und versuche, es mit dem grafischen Debugger **`nemiver`** laufenzulassen.

     - Teste die wichtigsten Debugger-Features, die im Vorlesungsskript erwähnt sind: Steppe zeilenweise durch den Code, setze Breakpoints, ...

     - Kannst du dir bei geschachtelten Funktionsaufrufen die Variablen aller Aufrufsebenen anschauen? Pointer verfolgen und Strukturen ansehen? Auf die Änderung einzelner Variablen warten?

3. **`ltrace` / `strace`:**

   Beschäftige dich mit beidem, nimm als zu testendes Programm **`lsof`**.

   - Welcher System Call braucht bei einem **`lsof`** die meiste Zeit?

   - Trace mit **`strace`** ein Programm mit parallelen Prozessen (**`lsof`** arbeitet unter bestimmten Umständen parallel), leite den Trace in einen File um, und vergleiche den Log bei **`-f`** und **`-ff`**. Was ist der Unterschied?

     - Konkret bei **`lsof`**: Kannst du das Verhalten deuten? Mit welcher Option kommt man eher ans Ziel, wenn man das Zusammenspiel der Prozesse analysieren will? Wie synchronisieren sich die beiden Prozesse bzw. wie kommunizieren sie miteinander?

   - Welches Problem hast du, wenn du mit **`ltrace`** auf ein C++-Programm mit Calls in C++-Libraries losgehst? Wie löst du es?

- Was ist der Unterschied zwischen **`-tt`**, **`-T`** und **`-r`** ?
- Kannst du die Menge des Outputs von **`strace`** auf bestimmte Operationen einschränken?
- Versuche, zu einem Aufruf in der **`ltrace`**-Ausgabe die entsprechende Stelle im Sourcecode zu finden. Du brauchst ein mit Debug-Information übersetztes Programm, die richtige **`ltrace`**-Option, und das Tool **`addr2line`**.

  Funktioniert dasselbe Vorgehen auch bei **`strace`**?
- Trace irgendeinen Commandline-Befehl, der fehlschlägt (z.B. weil die Rechte auf einen File fehlen). Kannst du den Fehler im **`strace`**-Output erkennen?

  Welche generelle Regel gilt für (fast) alle System Calls in Linux bei Fehlern? Wie kann man daher nach fehlgeschlagenen System Calls suchen?
- Angenommen, du siehst im **`strace`** einen verdächtigen Input in einem **`read`**-Aufruf. Wie findest du den dazugehörigen File?
- Lass ein Programm (z.B. aus Punkt 1) abstürzen (mit Nullpointer oder **`abort`**). Wie äußert sich das im **`strace`**?
- Wozu dient ein Großteil der System Calls **`open`**, **`mmap`** usw., die **`strace`** beim Programmstart anzeigt? Wann könnten diese Dinge hilfreich sein?

4. **`lsof`**:
   - Schau dir den Output von **`lsof`** an. Kannst du die gesamte Information deuten?
   - Siehst du einen Zusammenhang zwischen den Files im **`lsof`** und den File-System-Calls im **`strace`**?
   - Wie kannst du mit **`lsof`** die offenen Files eines einzelnen Programms ständig im Auge behalten?
   - Kannst du **`lsof`** auf einen einzigen Prozess oder Benutzer einschränken? Auf Netzwerk-Verbindungen? Auf Files in einem bestimmten Verzeichnisbaum?