Lehrstuhl Connected Mobility
Fakultät für Informatik
Technische Universität München

Grundlagen von Betriebssystemen und Systemsoftware

WS 2018/19

Übung 6: Modellierung

zum 26. November 2018

- Die **Hausaufgaben** für dieses Übungsblatt müssen **spätestens** am **Sonntag, den 02.12.2018 um 23:59** in Moodle hochgeladen worden sein.
- Alle C-Programme müssen mit den folgenden Flags kompiliert werden:
  `gcc -Wall -o <progname> <prog.c>`.
- Hierzu stellen wir für jedes Übungsblatt jeweils ein Makefile bereit, das **nicht** verändert werden darf, um sicherzustellen, dass Ihre Abgabe auch korrekt von uns getestet und bewertet werden kann. **Ihr Programmcode muss zwingend mit dem Makefile kompilierbar sein. Anderenfalls kann die Abgabe nicht bewertet werden.** Wenn Sie zwischenzeitlich Änderungen vornehmen wollen, um etwa bestimmte Teile mittels `#ifdef` einzubinden und zu testen, dann kopieren Sie am besten das Makefile und modifizieren Ihre Kopie. Mit `make -f <Makefile>` können Sie dann eine andere Datei zum Übersetzen verwenden.
- Die Abgabe der Programme erfolgt als Archivdatei, die die verschiedenen Quelldateien (.{c|h}) umfasst und **nicht** in Binärform. Nicht kompilierfähiger Quellcode wird **nicht gewertet**.
- Um die Abgabe zu standardisieren enthält das Makefile ein Target "submit" (`make submit`), was dann eine Datei blatt06.tgz zum Hochladen erzeugt.
- Damit die richtigen Dateien hochgeladen und ausgeführt werden, geben wir bei allen Übungen die jeweils zu verwendenden Dateinamen für den Quellcode und auch für das Executable an.
- Die Tests werden auf diesem Aufgabenblatt mittels Python-Programmen durchgeführt. Zum Ausführen der Tests geben Sie `python xyz_test.pyc` auf der Konsole ein. Ihre ausführbaren Programme müssen sich im selben Verzeichnis wie die Testprogramme befinden.

1 Vorbereitung auf das Tutorium

Was ist die Rolle der folgenden POSIX- (Portable Operating System Interface) bzw. Bibliotheksfunktionen? Erläutern Sie kurz die Parameter und den Rückgabewert jeder Funktion. [1]

- `bash(1)`
- `int strncmp (const char *s1, const char *s2, size_t n)`
- `ctype`
- `int execve (const char *path, char *const argv[], char *const envp[]);`
- `char *getenv (char *name)`
- `void *realloc (void *ptr, size_t size);`
- `char *strchr (const char *s, int c);`
- `char *strcat (char *restrict s1, const char *restrict s2);`
- `char *strncpy (char * dst, const char * src);`
- `char *strdup (const char *s);`
-

2 Tutoraufgaben

2.1 Sequentielle und nebenläufige Ausführung

Gegeben seien die folgenden fünf arithmetischen Ausdrücke:

```
(A) ( a − b )
(B) ( c + d )
(C) ( e − f )
(D) ( a − b ) ∗ ( c + d ) / ( e − f )
(E) ( u / (v + w ) ∗ ( r − t )) ∗ ( q + p ) / h
```

Wir gehen von einem Compiler aus, der die oben gelisteten Berechnungen in Maschinencode übersetzen möchte. Hierfür ist es nötig, geschachtelte Ausdrücke in einzelne Maschinenbefehle herunterzubrechen.

2.1.1 Sequentielle Ausführung

Formulieren Sie aus Sicht des Compilers ein einzelnes Programm P_1 (in Pseudocode-ähnlicher [2] Notation), das die arithmetischen Ausdrücke (A), (B) und (C) berechnet.

Formulieren Sie ein zweites Programm P_2 (in Pseudocode-ähnlicher Notation), das die beiden arithmetischen Ausdrücke (D) und (E) sequentiell berechnet.

Beachten Sie hierbei, dass komplexe (zusammengesetzte) arithmetische Ausdrücke nicht erlaubt sind. Jeder arithmetische Ausdruck darf maximal eine Operation und zwei Operatoren beinhalten. D. h. bei der Berechung solcher arithmetischer Ausdrücke muss die Berechnung von Teilausdrücken und

[1] Erläuterungen und Beispiele der einzelnen Funktionen finden Sie auch in den **man**-Pages auf Ihrer Linux-VM.

[2] Syntaktische Verwandschaft mit Programmiersprachen ist **nicht** notwendig. Der von Ihnen skizzierte Algorithmus muss jedoch nachvollziehbar sein.

die Ablage von Zwischenergebnissen in Hilfsvariablen von Ihnen explizit programmiert werden (z. B. $e_1 = a - b$). Führen Sie an jede der von Ihnen eingeführten Hilfsvariablen maximal eine Zuweisung aus (sog. **Single-Static-Assignment**-Form).

2.1.2 Nebenläufige Ausführung

Unser imaginärer Compiler versucht nun, Berechnungen durch Parallelisierung zu beschleunigen. Notieren Sie nebenläufige Programme P_1' und P_2', welche die entsprechenden arithmetischen Ausdrücke in möglichst wenigen getrennten Schritten berechnen. Zur Formulierung einer Menge von n Anweisungen, die nebenläufig ausgeführt werden sollen, benutzen Sie die **concurrent**-Anweisung in der folgenden Form:

```
concurrent_begin
  <anweisung_1>
  <anweisung_2>
  .....
  <anweisung_n>
concurrent_end;
```

2.2 Petri-Netze

Parallele Prozesse können durch Petri-Netze (S, T, F) modelliert werden. Dabei stehen die Stellen typischerweise für Zustände und die Transitionen für Aktionen. Modellieren Sie unter diesem Gesichtspunkt die folgenden Prozesse mit Petri-Netzen.

2.2.1 Anwendungsfall und Modellierung

Im Folgenden sollen Sie eine stark vereinfachte Version einer Bank modellieren. Nehmen Sie vereinfachend an, dass ein Kunde die Bank betreten kann. In der Bank erwartet ihn ein Angestellter am Schalter, der das Anliegen des Kunden bearbeitet. Sobald das Anlegen erledigt ist, verlässt der Kunde die Bank.

Erstellen Sie ein Petri-Netz, das die obige Situation beschreibt. Achten Sie auf eine aussagekräftige Benennung der Stellen und Transitionen.

2.2.2 Modifizierung des Modells

Modellieren Sie wieder die obige Bank. Nehmen Sie zusätzlich an, dass zwei Angestellte an jeweils einem Schalter (Schalter A und B) Kundenwünsche erfüllen können.

2.2.3 Erreichbarkeit

Zeigen und begründen Sie, dass das Petri-Netz (S, T, F) aus Aufgabe 2.2.2 mit der Startmarkierung M_0 lebendig ist. Erstellen Sie dazu einen Erreichbarkeitsgraphen, wobei die Knoten des Graphen die erreichbaren Markierungen und die Kanten die Transitionen des Petri-Netzes darstellen.

Beispiel: Der Knoten $(s_1\ s_2\ s_{3a}\ s_{3b}\ s_4)$ modelliert den Zustand bzw. die Belegung des Petri-Netzes ($s_i = 1$ falls s_i belegt, sonst 0).

2.3 Modellierung mit Aktionsspuren

Gegeben sei das Szenario aus Aufgabe 2.2.2. Es beschreibt den Ablauf wie Kunden ihr Anliegen an zwei Schaltern von Angestellten einer Bank bearbeiten lassen können. Abbildung 1 zeigt das zugehörige Petrinetz in einem Beispiel mit 4 Kunden.

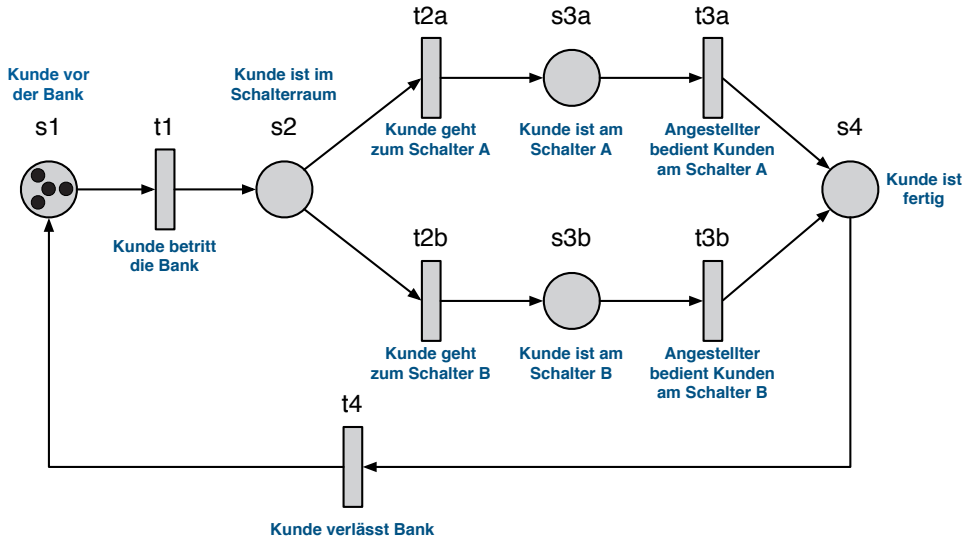

Figure 1: Petrinetz eines parallelen Ablaufs in einer Bank

2.3.1 Aktionsstrukturen

Ein Prozess P wird durch die Aktionsstruktur $p = (E, \leq, \alpha)$ beschrieben. Im Folgenden sollen Sie den Prozess P beschreiben, in dem Laura und Fred gemeinsam eine Bank betreten, bedient werden und die Bank wieder gemeinsam verlassen.

Die Aktionsmenge A sei gegeben durch:

- a_1: Zwei Kunden betreten gemeinsam die Bank
- a_2: Ein Kunde geht zum Schalter
- a_3: Ein Kunde bekommt sein Anliegen bearbeitet
- a_4: Ein Kunde ist fertig
- a_5: Zwei Kunden verlassen die Bank

a) Definieren Sie zunächst die Ereignismenge $E \subseteq E^*$ als Auslöser von Aktionen A. Ordnen Sie anschließend alle Aktionen $a_i \in A$ den zugehörigen Ereignissen $e_j \in E$ mit Hilfe der Aktionsmarkierung $\alpha : E \to A$ zu, so dass alle Ereignisse $e_j \in E$ einer Aktion $a_i \in A$ zugeordnet sind. Beachten Sie, dass α nicht notwendigerweise injektiv sein muss: eine Aktion kann mehreren Ereignissen zugeordnet werden.

b) Geben Sie die Kausalitätsrelation $\leq$ an, die die kausalen Abhängigkeiten der Ereignisse definiert:

$$(e_i, e_j) \in \leq \iff e_i \text{ findet vor } e_j \text{ statt} \wedge e_i, e_j \in E$$

c) Skizzieren Sie die Kausalitätsrelation $\leq$ in Form eines gerichteten Graphen, wobei die Ereignisse $e_j \in E$ als Knoten und ihre kausalen Abhängigkeiten durch gerichtete Kanten des Graphen dargestellt werden sollen.

2.3.2 Nebenläufigkeit

Welche Ereignisse sind in P nebenläufig?

2.3.3 Sequenzialisierung

a) Sequenzialisieren Sie den Prozess P. Geben Sie dabei eine Spur zur Aktionsstruktur p an.

b) Entwerfen Sie einen Algorithmus zur Sequentialisierung eines beliebigen Prozesses.

3 Hausaufgaben

3.1 Kommandozeilenparser (7 Punkte)

Wir wollen in dieser Aufgabe mit der Implementierung Ihrer Shell beginnen. Diese soll (der Anfang ist einfach) einen Prompt ausgeben: "$ " und dann auf eine Eingabe des Benutzers warten. Lesen Sie die Eingabe mit `fgets()` ein und analysieren Sie danach die Eingabe. Sie dürfen annehmen, dass die Eingabe nicht länger sein wird als 1023 Zeichen plus Nullterminierung des Strings ("\0").

Schreiben Sie hierzu einen Parser, der einen Eingabestring in einzelne Elemente aufteilt. Als Trennzeichen werden (wie bei der echten Shell auch) Leerzeichen verwendet. Achtung: berücksichtigen Sie, dass zwei Elemente durch mehr als ein Leerzeichen getrennt sein können. Nutzen Sie Ihre Listenverwaltung, um die Parameter in der Reihenfolge ihres Auftreten in diese Liste einzufügen und anschließend auszugeben.

Ein einfaches Beispiel:

```
$ echo MW 0001 abc
1:echo
2:MW
3:0001
4:abc
```

Ihr Kommandozeilnenparser soll im nächsten Schritt weitere Sonderzeichen behandeln können, wie wir sie auch von anderen Shells kennen. Implementieren Sie insbesondere folgende Sonderzeichen:

- ¨ – Doppelte Anführungszeichen müssen immer in Paaren auftreten. Sie dienen dazu, den von ihnen umschlossenen Teil als einen Parameter aufzufassen, auch wenn Leerzeichen auftreten. Außerdem setzen sie andere Funktionen wie etwa Wildcards ("*") und andere speziellen Zeichen außer Kraft.

  So wird dann z.B. ¨`MW 0001`¨ als ein Parameter wahrgenommen. Die Anführungszeichen selbst dienen lediglich der Gruppierung, werden aber nicht Bestandteil des Parameters.

- ' – Einfache Anführungszeichen müssen ebenfalls immer in Paaren auftreten. Auch diese dienen dazu, den von ihnen umschlossenen Teil als einen Parameter aufzufassen, auch wenn Leerzeichen auftreten. Außerdem setzen sie andere Funktionen wie etwa Wildcards ("*") und andere speziellen Zeichen außer Kraft.

  Analog zu doppelten Anführungszeichen wird z.B. `'MW 0001'` als ein Parameter wahrgenommen.

  Wichtig: Werden einfache und doppelte Anführungszeichen geschachtelt, so haben nur die äußeren Zeichen ihre Sonderfunktion: Also ¨'¨ liefert ', und '¨' liefert ¨. Und ¨`Hier steht 'Text'.`¨ ergibt einen Parameter `Hier steht 'Text'.`

- $ – Das Dollar-Zeichen signalisiert, dass der folgende Text bis zum nächsten Zeichen, das kein Buchstabe, keine Zahl und kein Unterstrich ist, als Variablenname wahrgenommen wird, die dann durch ihren Wert ersetzt wird. Erlauben Sie konkret die folgenden Zeichens in Variablennamen: "`ABZDEFGHIJKLMNOPQRSTUVWXYZ0123456789_`"

  Beispielsweise expandiert `$PWD` zu "`/etc/home/user/directory/`".

  Die Namen und Werte der Umgebungsvariablen finden Sie in dem bisher zwar genannten, aber noch nicht weiter diskutierten dritten Parameter der Funktion `main()`. Zur Erinnerung:

```
int main (int argc, char *argv [], char *envp []);
```

Der Pointer `envp` zeigt auf ein null-terminiertes Array von Strings, die jeweils den Aufbau `NAME=WERT` haben, den sogenannten Umgebungsvariablen (engl. Environment).

Für jede Variable, die Sie in der Kommandozeile identifizieren, finden Sie den entsprechenden Wert, und ersetzen Sie die Variable durch diesen Wert.

- `\` – Der Backslash hebt die Sonderbedeutung des nachfolgenden Zeichens auf. Insbesondere wird auch die Sonderfunktion der obigen Sonderzeichen `'`, `¨` und `$` aufgehoben. Auch die trennende Wirkung des nachfolgenden Leerzeichens wird aufgehoben.

  So wird `MW\ 0001` als ein Parameter "MW 0001" interpretiert. Und `\$` steht dann für `$` (und nicht mehr für eine Variable) usw.

Hinweis: Das Parsieren eines Eingabestrings können Sie sich als Zustandsautomat vorstellen, dessen unterschiedliche Zustände beispielsweise angeben, ob der Parser gerade in einem von Anführungszeichen umschlossenen String befindet oder in einer Variable, ob das vorige Zeichen ein Backslash war, ob Sie gerade Leerzeichen lesen usw.

Außerdem werden Sie nicht umhinkommen, den Speicher für die Inhalte der einzelnen Listenelemente dynamisch zu allozieren, da diese ja durch Variablenexpansion wachsen können.

Teilen Sie Ihren Programmcode so auf, dass Sie die Parserfunktion (parser.c) von der Main-Funktion (parsertest.c) trennen, wobei letztere auch das Prompt ausgibt und auf die Eingabe wartet.

Wenn als Kommando "exit" eingegeben wird, soll das Programm terminieren.

Quellen: parser.c parsertest.c list.c

Executable: parsertest

Weitere Beispiele:

```
$ "Hello, world" 2 3 4
1:Hello, world
2:2
3:3
4:4
$ "Hier kommt das aktuelle Verzeichnis: $PWD"
1:Hier kommt das aktuelle Verzeichnis: /tum/gbs/ws18/06/c-files
$ Leerzeichen\ mit\ Backslash und auch ohne
1:Leerzeichen mit Backslash
2:und
3:auch
4:ohne
$ Hier steht \"ein Text\"
1:Hier
2:steht
3:"ein
4:Text"
$ Hier steht "\"noch ein Text\""
1:Hier
2:steht
3:"noch ein Text"
$ Oder auch '"so ein Text"'
1:Oder
2:auch
3:"so ein Text"
$ Das geht auch "'mit anderen Anfuehrungszeichen'"
1:Das
2:geht
3:auch
```

```
4:'mit anderen Anfuehrungszeichen'
$ Mal tut der \\ nichts: \hier \etwa
1:Mal
2:tut
3:der
4:\
5:nichts:
6:hier
7:etwa
$   "Anfuehrungs"zeichen koennen einfach 'ver'schwin'den'
1:Anfuehrungszeichen
2:koennen
3:einfach
4:verschwinden
$   und    viele Leerzeichen machen    nichts
1:und
2:viele
3:Leerzeichen
4:machen
5:nichts
```

3.2 Ausführung der Kommandos (3 Punkte)

Jetzt kommen wir zur eigentlichen Shell und damit der Programmausführung. Zur Ausführung des Kommandos der Kommandozeile verwenden Sie zunächst das schon bekannte `fork()`, um einen Kindprozess zu erzeugen (den Sie wie schon zuvor in einer Liste verwalten, damit Sie später auf dessen Termination warten können (und auch in einer der nächsten Aufgaben *Job Control* ergänzen können). Verwenden Sie `execve()` zum Ausführen des Kommandos, d.h., des ersten Elementes der Kommandozeile. Die weiteren Elemente werden dem Kommando als Parameter übergeben.

Für die Ausführung müssen Sie folgendes beachten:

- Sie müssen die in einer Liste gesammelten Parameter wieder in ein Array überführen. Schreiben hierzu Sie eine Funktion `list_to_array()`, die eine Liste in ein dynamisch alloziertes Array von Pointern umwandelt. Hierbei kann es praktisch sein, dass Sie ihre Listen-Datenstruktur um einen Zähler ergänzen, der die aktuelle Anzahl der Elemente der Liste enthält. Dann wissen Sie gleich, wieviel Speicher Sie für Ihr Array allozieren müssen. Dabei ist es wichtig, als letztes Element des Arrays einen NULL-Pointer zu ergänzen.
- Der erste Eintrag im Array – `argv [0]` – enthält immer den Kommandonamen, z.B. “ls” oder “make”.
- Die Funktion `execve()` erwartet als ersten Parameter den kompletten Pfadnamen. Eine Shell weiß aufgrund der Umgebungsvariable `$PATH`, wo sie in welcher Reihenfolge nach dem Kommando suchen soll. `$PATH` enthält eine Liste der Verzeichnisse, in denen nach einem Kommando gesucht werden soll, durch Doppelpunkte getrennt. Ihre Shell muss diese Umgebungsvariable parsieren und, beginnend mit dem ersten Verzeichnis, für jedes Verzeichnis den kompletten Pfadnamen durch Anhängen von Slash (“/”) und dem angebenen Kommandonamen testen, ob das Kommando dort gefunden wird. Sie können das in einer Schleife realisieren, in der Sie `execve()` aufrufen und den Rückgabewert prüfen. (Wenn `execve()` erfolgreich ist, kehrt der Aufruf nicht zurück.) Wenn das Kommando nicht gefunden wird, geben Sie eine Fehlermeldung aus.

  Wichtig: Die PATH-Variable wird **nicht** evaluiert, wenn ein Pfad explizit im Kommando angegeben wurde, d.h., das Kommando einen Slash (“/”) enthält: beispielsweise `/bin/ls`, `./mysh`,

`src/xyz` oder `../client`.

Wenn als Kommando “exit” eingegeben wird, soll Ihre Shell terminieren.

Quellen: parser.c mysh.c list.c

Executable: mysh

```
$ ./mysh
$ ls
Makefile        list.c~         list.h~         mysh.c          parser.c
parsertest      parsertest.c~   list.c          list.h          mysh
mysh.c~         parser.c~       parsertest.c
$ make
make: Nothing to be done for ‘all’.
$ echo $PWD
/tum/gbs/ws18/06/c-files
$ exit
```

3.3 Häufige Fehler in C (für Interessierte - Keine Abgabe)

Erfahrungsgemäß unterschätzen viele Studierende die Wichtigkeit von korrektem Umgang mit dem Speicher und undefiniertem Verhalten. Wir möchten Sie dabei unterstützen, derartige Fehler von vornherein zu vermeiden, insbesondere, da sie nicht sofort auffallen und oft keinen Compiler- oder Laufzeitfehler verursachen. Im Folgenden stellen wir zwei Konzepte vor, die den Entwickler dabei unterstützen können potenzielle Fehler bei der Speicherallokation schneller zu finden.

Im Folgenden möchten wir (künstliche) Speicherleaks auf dem Heap analysieren. Verwenden Sie dafür z.B. die Lösung Ihrer Listenverwaltung, und modifizieren Sie den Code, sodass die dynamisch allokierten Knoten nicht freigegeben werden. Löschen bzw. kommentieren Sie dafür die folgende Zeile aus dem Source Code.

Kompilieren Sie anschließend den Code wie gewohnt. Verwenden Sie dabei zusätzlich die Option `-g`. Diese Option generiert Debugging-Informationen, die Ihnen durch die folgenden Konzepte u.a. die Zeilennummern mitteilen, in denen ein potenzielles Problem aufgedeckt wurde.

3.3.1 Valgrind

Das Tool Valgrind [3] ist in der Lage, nicht freigegebenen Speicher zu finden, der dynamisch z.B. mit Hilfe der Funktion `malloc` auf dem Heap allokiert wurde. Valgrind ist sowohl in Ihrer VM als auch auf den Systemen der Rechnerhalle installiert und kann für diese Hausaufgabe verwendet werden. Im Folgenden möchten wir dynamische Speicherleaks auf dem Heap des oben vorbereiteten Programms analysieren.

Um ein kompiliertes Programm auf potenzielle Speicherleaks zu überprüfen können Sie Valgrind wie folgt in Ihrer Konsole ausführen:

```
valgrind --leak-check=full ./syncem ...
```

Sobald das Programm sich beendet hat, teilt Valgrind Ihnen wie folgt eine Zusammenfassung des Leak-Checks mit:

```
==820==
==820== HEAP SUMMARY:
==820==     in use at exit: 120 bytes in 5 blocks
==820==   total heap usage: 6 allocs, 1 frees, 1,144 bytes allocated
==820==
==820== 120 (24 direct, 96 indirect) bytes in 1 blocks are definitely
            lost in loss record 5 of 5
==820==    at 0x4C2BBAF: malloc (vg_replace_malloc.c:299)
==820==    by 0x108824: new_node (bst-solution.c:15)
==820==    by 0x10886F: insert_rec (bst-solution.c:29)
==820==    by 0x108911: make_random_tree (bst-solution.c:46)
==820==    by 0x108B80: main (bst-solution.c:133)
==820==
==820== LEAK SUMMARY:
==820==    definitely lost: 24 bytes in 1 blocks
==820==    indirectly lost: 96 bytes in 4 blocks
==820==      possibly lost: 0 bytes in 0 blocks
==820==    still reachable: 0 bytes in 0 blocks
==820==         suppressed: 0 bytes in 0 blocks
==820==
==820== For counts of detected and suppressed errors, rerun with: -v
==820== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

[3] Valgrind: `http://valgrind.org/docs/manual/quick-start.html`

Die `LEAK SUMMARY` teilt Ihnen mit, dass definitiv Speicher verloren gegangen ist (*definitely/indirectly lost*). Zusätzlich erhält man in `HEAP SUMMARY` die Informationen darüber in welcher Funktion der nicht freigegebene Speicher allokiert wurde (in unserem Fall in der Funktion `new_node` in der Zeile 15). Solche Fehler können leichter gefunden und behoben werden und ersparen dem Entwickler eine Menge Ärger mit der dynamischen Speicherverwaltung im Code.

Beachten Sie: Speicherfehler auf dem Stack werden durch Valgrind nicht gefunden. Um auch solche Fehler aufzudecken, kann der Adress Sanitizer verwendet werden.

3.3.2 Address Sanitizer

Address Sanitizer [4] kann sowohl in Ihrer VM als auch auf den Systemen der Rechnerhalle verwendet werden, indem man den Code mit der Option `-fsanitize=address` kompiliert:

```
gcc -Wall -g -fsanitize=address -o yourprog yourprog.c
```

Sobald das Programm sich beendet hat teilt Ihnen der Address Sanitizer wie folgt eine Zusammenfassung des Leak-Checks mit:

```
=================================================================
==926==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 24 byte(s) in 1 object(s) allocated from:
#0 0x7fe7f6943d28 in malloc (/usr/lib/x86_64-linux-gnu/libasan.so.3
                  +0xc1d28)
#1 0x5638c0215ec4 in new_node /tmp/bst-solution.c:15
#2 0x5638c0215f88 in insert_rec /tmp/bst-solution.c:29
#3 0x5638c02160ec in make_random_tree /tmp/bst-solution.c:46
#4 0x5638c0216544 in main /tmp/bst-solution.c:133
#5 0x7fe7f65032b0 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6
                  +0x202b0)

Indirect leak of 24 byte(s) in 1 object(s) allocated from:
#0 0x7fe7f6943d28 in malloc (/usr/lib/x86_64-linux-gnu/libasan.so.3
                  +0xc1d28)
#1 0x5638c0215ec4 in new_node /tmp/bst-solution.c:15
#2 0x5638c0215f88 in insert_rec /tmp/bst-solution.c:29
#3 0x5638c0216001 in insert_rec /tmp/bst-solution.c:33
#4 0x5638c0216136 in make_random_tree /tmp/bst-solution.c:51
#5 0x5638c0216544 in main /tmp/bst-solution.c:133
#6 0x7fe7f65032b0 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6
                  +0x202b0)

...
```

Verwenden Sie für Ihre Hausaufgaben den Address Sanitizer, um potenzielle Speicherleaks zu vermeiden.

Beachten Sie: Für Performanzmessungen oder Analyse mit `objdump` muss diese Option unbedingt deaktiviert sein, da der Compiler bei aktiviertem `Adress Sanitizer` viel Boilerplatecode generiert, der beim Lesen des Disassemblys hinderlich ist.

[4] Address Sanitizer: `https://github.com/google/sanitizers/wiki/AddressSanitizer`