Softwareentwicklung für eingebettete Systeme

Bachelorstudiengang Technische Informatik
Hochschule Pforzheim

Vorlesung 6
Sommersemester 2014

Dipl.-Ing.(FH) Marc Jüttner

Laufzeitanalyse

- Ermitteln des Laufzeitverhaltens von Anwendungen
  - Auffinden ineffizienter Implementierungen
- Auffinden fehlerhafter Programmierung
  - Speicherfehler
  - Multithreadingfehler
  - Lockingfehler

Optimierung

- Häufig aufgerufene Funktionen können optimiert werden
  - Verbesserung des Laufzeitverhaltens
  - Einhaltung von Echtzeitanforderungen
- Optimierung selten aufgerufener Funktionen ist nicht immer lohnenswert

Speicherfehler

- Fehler bei der Speicherallokation
- Statische Allokation
  - Lokale Variablen auf dem Stack
  - Globale Variablen
- Dynamische Allokation
  - Allokation mittels *malloc()* oder *new*

Speicherfehler

- Speicherfehler durch fehlerhaftes Laufzeitverhalten
- Über- oder Unterschreiten von Arrays
  - Buffer overflow/underflow
- Folgen sind
  - Datenkorruption
  - Codemanipulation

Speicherfehler

- Fehlerhafter Zugriff auf den Stack
  - Overrunning
- Zugriff auf freigegebenen Speicher
  - Datenkorruption möglich
- Verwendung undefinierter Variablen
  - Nichtdeterministisches Programmverhalten

Speicherfehler

- Nichtfreigabe allokierten Speichers
  - Memory leaks
- Doppelte Freigabe allokierten Speichers
- Überlappende Parameter bei memcpy()
  - Quell- und Zielbereiche überlappen
  - Datenkorruption

Allokation und Deallokation

- Funktionspaare zur Allokation und Deallokation müssen immer korrekt verwendet werden
  - malloc() - free()
  - new – delete
  - new[] - delete[]
- Unterschied zwischen malloc()/free() und new/delete?

Multithreadingfehler

- Zugriff auf globale Variablen aus mehreren Threads
  - Erfordert wechselseitigen Ausschluss
- Fehlerhafte Verwendung von Mutexen
  - Nicht freigeben von Mutexen
  - Mutex-Nesting

Allgemeine Analysewerkzeuge

- Debugger
  - Programmablaufverfolgung
  - Speicheruntersuchung
- Profiler
  - Laufzeituntersuchung
  - Untersuchung des Zeitverhaltens
  - Bestimmung der CPU-Zeit pro Funktion

Einfluss der Beobachtung

- Beobachtung des Programmverhaltens beeinflusst den Programmablauf
  - Nicht funktionierende Programme funktionieren „plötzlich“ im Debugger
  - Verlangsamter Programmablauf durch zusätzliche Implementierungen

Profiler

- Fügen zusätzliche Symbole und Code ein
  - Speichern von Zeitstempeln an definierten Stellen
  - Zuordnung von Funktionen und CPU-Zeit
- „Abfangen“ kritischer Systemfunktionen zur Ermittlung der Laufzeit und Aufrufhäufigkeit

Beispiel: gprof

- Profiler zur Untersuchung des Laufzeitverhaltens
  - Besondere Compileroption erforderlich
  - Profildatei erzeugen
  - Gprof-Aufruf nach der Ausführung

```
$gcc -Wall -pg gprof_test.c gprof_ext.c -o gprof_test
$./gprof_test
$gprof gprof_test gmon.out > analysis.txt
```

Programmbeispiel

```c
#include <stdio.h>

void new_func1(void);

void func1(void) {
  printf("\n Inside func1 \n");
  int i = 0;

  for(;i<0xffffffff;i++);
  new_func1();

  return;
}

static void func2(void) {
  printf("\n Inside func2 \n");
  int i = 0;

  for(;i<0xffffffaa;i++);
  return;
}

int main(void) {
  printf("\n Inside main()\n");
  int i = 0;

  for(;i<0xffffff;i++);
  func1();
  func2();

  return 0;
}
```

```c
#include <stdio.h>

void new_func1(void) {
  printf("\n Inside new_func1()\n");
  int i = 0;

  for(;i<0xfffffee;i++);

  return;
}
```

Ausgabe – Seite 1

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 34.38      14.72    14.72        1    14.72    14.72  new_func1
 33.72      29.16    14.44        1    14.44    14.44  func2
 32.90      43.25    14.09        1    14.09    28.81  func1
  0.12      43.30     0.05                             main
```

Gesamtzeit

Laufzeit der Funktion

Anzahl Aufrufe

Laufzeit der Funktion pro Aufruf

Laufzeit der Funktion und Unterfunktionen pro Aufruf

Ausgabe – Seite 2

```
granularity: each sample hit covers 2 byte(s) for 0.02% of 43.30 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.05   43.25                 main [1]
                14.09   14.72       1/1           func1 [2]
                14.44    0.00       1/1           func2 [4]
-----------------------------------------------
                14.09   14.72       1/1           main [1]
[2]     66.5    14.09   14.72       1         func1 [2]
                14.72    0.00       1/1           new_func1 [3]
-----------------------------------------------
                14.72    0.00       1/1           func1 [2]
[3]     34.0    14.72    0.00       1         new_func1 [3]
-----------------------------------------------
                14.44    0.00       1/1           main [1]
[4]     33.3    14.44    0.00       1         func2 [4]
-----------------------------------------------
```

Weitere Funktionen

- Unterdrücken privater Funktionen
- Ausgabe einzelner Analysezweige
  - Flat profile, call graph information
- Untersuchung bestimmter Funktionen
  - Angabe anhand Funktionsnamen

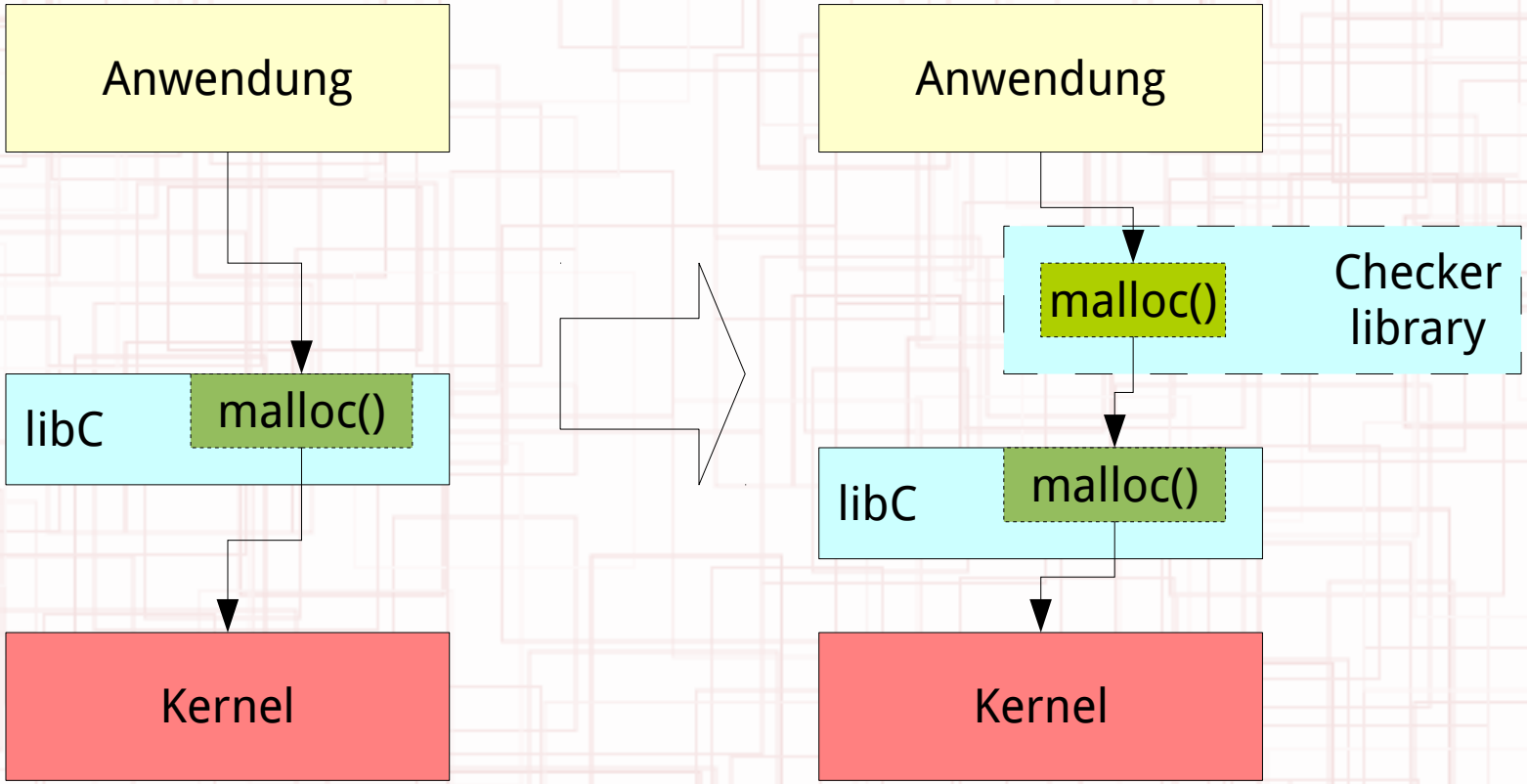
Systemprofiler

- Profilingwerkzeuge zur Analyse des Gesamtsystems
  - Welcher Prozess verbraucht Rechenzeit?
  - Welcher Prozess verbraucht Speicher?
  - In welcher Funktion verbraucht ein Prozess Ressourcen?
  - ...

Beispiel: Valgrind

- Open Source
- Kombiniertes Analysewerkzeug
  - Untersuchung auf Speicherfehler in Anwendungen
  - Thread error detection
  - Heap profiling
  - Call graph generator
  - ...

Memory checker: Funktion



Memory checker

- Valgrind: Memcheck
  - Aufruf ähnlich Debugger
  - Testsoftware als Parameter

```
$gcc -O0 -g -o memleak memleak.c

$valgrind --tool=memcheck --read-var-info=yes \
    --leak-check=full --undef-value-errors=yes \
    --track-origins=yes ./memleak
```

Programmbeispiel

```c
#include <stdio.h>
#include <malloc.h>

#define ARRAYSIZE       128

void main(void) {
  int *values, *results, sum, *oldsum;
  int count;

  values = (int *)malloc(ARRAYSIZE * sizeof(*values));
  results = (int *)malloc(ARRAYSIZE * sizeof(*values));
  oldsum = (int *)malloc(sizeof(int));

  printf("Populate values...\n");
  for (count = 0; count < ARRAYSIZE; count++) {
    values[count] = count;
  }

  printf("Oldsum is %d.\n", *oldsum);
  printf("Sum is %d.\n", sum);

  for (count = 0; count <= ARRAYSIZE; count++) {
    results[count] = values[count] * values[count + 1];
    sum += results[count];
  }

  free(values);

  printf("Exiting.\n");
}
```

Ausgabe – Seite 1

```
==23196== Memcheck, a memory error detector
==23196== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==23196== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==23196== Command: ./memleak
==23196==
Populate values...
==23196== Conditional jump or move depends on uninitialised value(s)
==23196==    at 0x4E7E030: vfprintf (vfprintf.c:1654)
==23196==    by 0x4E87FF8: printf (printf.c:34)
==23196==    by 0x40068B: main (memleak.c:19)
==23196==  Uninitialised value was created by a heap allocation
==23196==    at 0x4C2A2DB: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==23196==    by 0x40063A: main (memleak.c:12)
==23196==
==23196== Use of uninitialised value of size 8
==23196==    at 0x4E7D5AB: _itoa_word (_itoa.c:179)
==23196==    by 0x4E81AE1: vfprintf (vfprintf.c:1654)
==23196==    by 0x4E87FF8: printf (printf.c:34)
==23196==    by 0x40068B: main (memleak.c:19)
==23196==  Uninitialised value was created by a heap allocation
==23196==    at 0x4C2A2DB: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==23196==    by 0x40063A: main (memleak.c:12)
==23196==

[...]
Oldsum is 0.
```

Ausgabe – Seite 2

```
==23196== Conditional jump or move depends on uninitialised value(s)
==23196==    at 0x4E7E030: vfprintf (vfprintf.c:1654)
==23196==    by 0x4E87FF8: printf (printf.c:34)
==23196==    by 0x40069F: main (memleak.c:20)
==23196==  Uninitialised value was created by a stack allocation
==23196==    at 0x40060D: main (memleak.c:6)
==23196==
==23196== Use of uninitialised value of size 8
==23196==    at 0x4E7D5AB: _itoa_word (_itoa.c:179)
==23196==    by 0x4E81AE1: vfprintf (vfprintf.c:1654)
==23196==    by 0x4E87FF8: printf (printf.c:34)
==23196==    by 0x40069F: main (memleak.c:20)
==23196==  Uninitialised value was created by a stack allocation
==23196==    at 0x40060D: main (memleak.c:6)
==23196==
[...]
Sum is 4196144.
[...]
```

Ausgabe – Seite 3

```
==23196== Invalid read of size 4
==23196==    at 0x4006EB: main (memleak.c:23)
==23196==  Address 0x51fc240 is 0 bytes after a block of size 512 alloc'd
==23196==    at 0x4C2A2DB: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==23196==    by 0x40061E: main (memleak.c:10)
==23196==
==23196== Invalid read of size 4
==23196==    at 0x4006D1: main (memleak.c:23)
==23196==  Address 0x51fc240 is 0 bytes after a block of size 512 alloc'd
==23196==    at 0x4C2A2DB: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==23196==    by 0x40061E: main (memleak.c:10)
==23196==
==23196== Invalid write of size 4
==23196==    at 0x4006F0: main (memleak.c:23)
==23196==  Address 0x51fc480 is 0 bytes after a block of size 512 alloc'd
==23196==    at 0x4C2A2DB: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==23196==    by 0x40062C: main (memleak.c:11)
==23196==
==23196== Invalid read of size 4
==23196==    at 0x400706: main (memleak.c:24)
==23196==  Address 0x51fc480 is 0 bytes after a block of size 512 alloc'd
==23196==    at 0x4C2A2DB: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==23196==    by 0x40062C: main (memleak.c:11)
[...]
```

Ausgabe – Seite 4

```
Exiting.
==23196==
==23196== HEAP SUMMARY:
==23196==     in use at exit: 516 bytes in 2 blocks
==23196==   total heap usage: 3 allocs, 1 frees, 1,028 bytes allocated
==23196==
==23196== 4 bytes in 1 blocks are definitely lost in loss record 1 of 2
==23196==    at 0x4C2A2DB: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==23196==    by 0x40063A: main (memleak.c:12)
==23196==
==23196== 512 bytes in 1 blocks are definitely lost in loss record 2 of 2
==23196==    at 0x4C2A2DB: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==23196==    by 0x40062C: main (memleak.c:11)
==23196==
==23196== LEAK SUMMARY:
==23196==    definitely lost: 516 bytes in 2 blocks
==23196==    indirectly lost: 0 bytes in 0 blocks
==23196==      possibly lost: 0 bytes in 0 blocks
==23196==    still reachable: 0 bytes in 0 blocks
==23196==         suppressed: 0 bytes in 0 blocks
==23196==
==23196== For counts of detected and suppressed errors, rerun with: -v
==23196== ERROR SUMMARY: 31 errors from 18 contexts (suppressed: 2 from 2)
```

Heap profiler

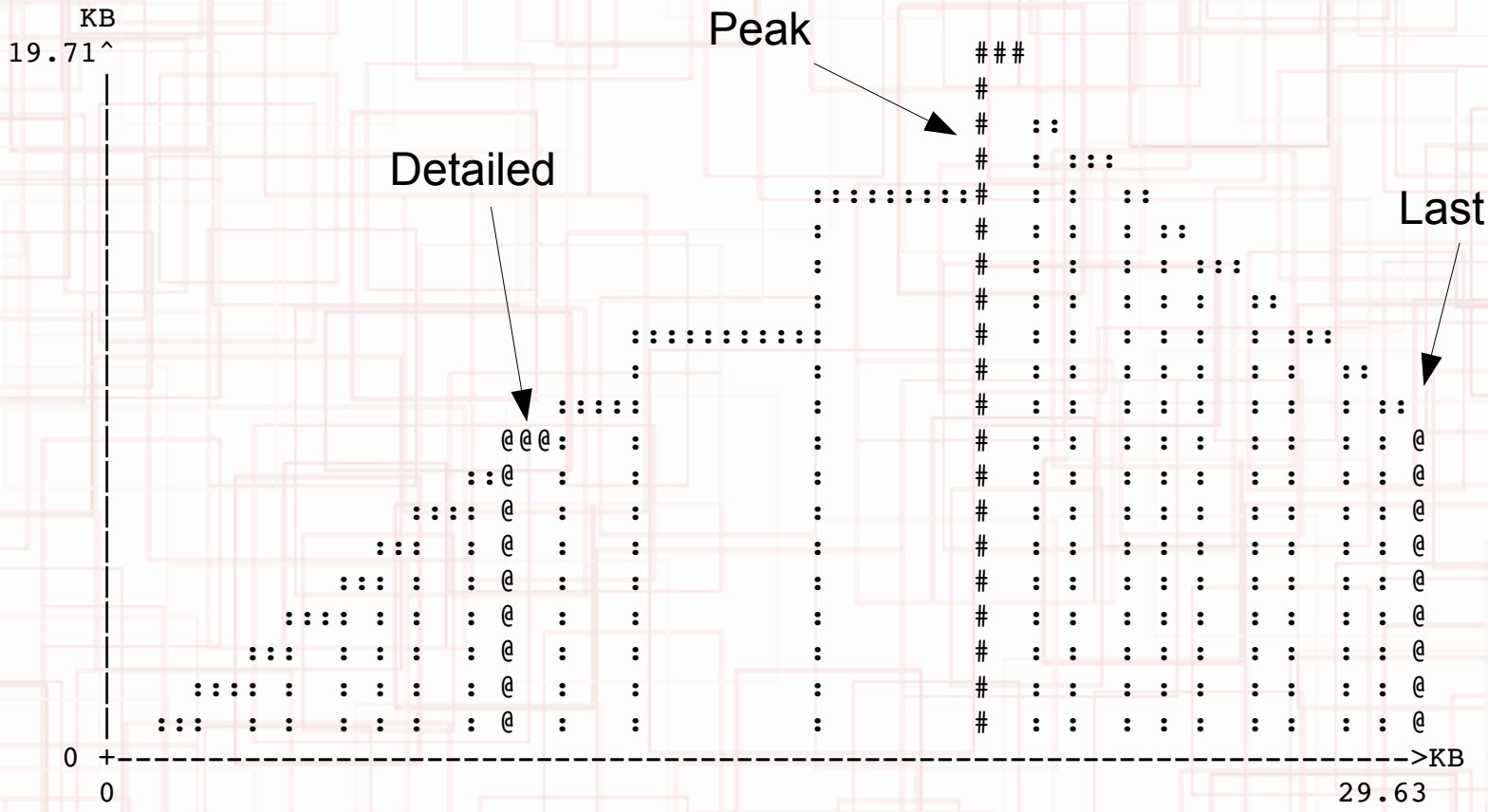
- Valgrind: Massif
  - Erzeugt Profildatei im aktuellen Verzeichnis
  - Auswertung mittels Zusatzprogramm

```
$gcc -O0 -g -o heapusage heapusage.c

$valgrind --tool=massif --time-unit=B \
    --massif-out-file=massif.out ./heapusage
$ms_print massif.out
```

Programmbeispiel

```c
#include <stdlib.h>

#define ARRAYSIZE       10

int *f_alloc, *g_alloc;

void g(void) {
  g_alloc = (int *)malloc(4000);
}

void f(void) {
  f_alloc = (int *)malloc(2000);
  g();
}

int main(void) {
  int i;
  int *a[ARRAYSIZE];

  for (i = 0; i < ARRAYSIZE; i++) {
    a[i] = (int *)malloc(1000);
  }

  f();

  g();

  for (i = 0; i < ARRAYSIZE; i++) {
    free(a[i]);
  }

  return 0;
}
```

Ausgabe – Seite 1




```
Number of snapshots: 25
 Detailed snapshots: [9, 14 (peak), 24]
```

Ausgabe - Seite 2

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0              0              0            0
  1          1,016            1,016          1,000             16            0
  2          2,032            2,032          2,000             32            0
  3          3,048            3,048          3,000             48            0
  4          4,064            4,064          4,000             64            0
  5          5,080            5,080          5,000             80            0
  6          6,096            6,096          6,000             96            0
  7          7,112            7,112          7,000            112            0
  8          8,128            8,128          8,000            128            0
  9          9,144            9,144          9,000            144            0
98.43% (9,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->98.43% (9,000B) 0x4005C9: main (heapusage.c:21)
```

Ausgabe – Seite 3

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 10         10,160           10,160          10,000           160            0
 11         12,168           12,168          12,000           168            0
 12         16,176           16,176          16,000           176            0
 13         20,184           20,184          20,000           184            0
 14         20,184           20,184          20,000           184            0
99.09% (20,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->49.54% (10,000B) 0x4005C9: main (heapusage.c:21)
|
->39.64% (8,000B) 0x400589: g (heapusage.c:8)
| ->19.82% (4,000B) 0x4005AC: f (heapusage.c:13)
| | ->19.82% (4,000B) 0x4005E5: main (heapusage.c:24)
| |
| ->19.82% (4,000B) 0x4005EA: main (heapusage.c:25)
|
->09.91% (2,000B) 0x4005A0: f (heapusage.c:12)
  ->09.91% (2,000B) 0x4005E5: main (heapusage.c:24)
```

Ausgabe - Seite 4

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 15         21,200           19,168          19,000           168            0
 16         22,216           18,152          18,000           152            0
 17         23,232           17,136          17,000           136            0
 18         24,248           16,120          16,000           120            0
 19         25,264           15,104          15,000           104            0
 20         26,280           14,088          14,000            88            0
 21         27,296           13,072          13,000            72            0
 22         28,312           12,056          12,000            56            0
 23         29,328           11,040          11,000            40            0
 24         30,344           10,024          10,000            24            0
99.76% (10,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->79.81% (8,000B) 0x400589: g (heapusage.c:8)
| ->39.90% (4,000B) 0x4005AC: f (heapusage.c:13)
| | ->39.90% (4,000B) 0x4005E5: main (heapusage.c:24)
| |
| ->39.90% (4,000B) 0x4005EA: main (heapusage.c:25)
|
->19.95% (2,000B) 0x4005A0: f (heapusage.c:12)
| ->19.95% (2,000B) 0x4005E5: main (heapusage.c:24)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)
```

Warum dynamisch?

- Dynamische Untersuchung bringt Vorteile
- Programme ohne Memory leaks können Probleme verursachen
  - Fehlendes free()/delete
  - Verbrauch des gesamten virtuellen Speichers...

Thread error detector

- Valgrind: Helgrind
  - Direkte Ausgabe der Ergebnisse auf der Standardkonsole

```
$gcc -O0 -g -o threads threads.c -lpthread

$valgrind --tool=helgrind ./threads
```

Programmbeispiel

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>


#define NUMTHREADS      3
#define EVILTHREAD      (NUMTHREADS - 1)

pthread_t threads[NUMTHREADS];

struct threadctx {
  int number;
} threadctx[NUMTHREADS];

pthread_mutex_t mutex;
int globalcounter = 0;
int runflag = 1;

void *threadfn(void *arg) {
  struct threadctx *ctx = (struct threadctx *)(arg);
  int run = 1;
  printf("[%03d] Thread starting.\n", ctx->number);

  while (runflag) {
    pthread_mutex_lock(&mutex);
    printf("[%03d] Thread alive.\n", ctx->number);
    globalcounter++;
    if (EVILTHREAD != ctx->number) pthread_mutex_unlock(&mutex);
    sleep(1);
  }
  printf("[%03d] Thread terminates...\n", ctx->number);

  return arg;
}

```

main()

```
35
36  void main(void) {
37    int count;
38    void *returnvalue;
39
40    if (0 > pthread_mutex_init(&mutex, NULL)) {
41      printf("Failed to initialise mutex.\n");
42      return;
43    }
44
45    for (count = 0; count < NUMTHREADS; count++) {
46      threadctx[count].number = count;
47      if (0 > pthread_create(&threads[count], NULL, threadfn, (void *)&threadctx[count])) {
48        printf("Failed to create thread %d.\n", count);
49        exit(0);
50      }
51    }
52
53    sleep(3);
54    runflag = 0;
55
56    sleep(5); /* wait for threads to terminate */
57    printf("Total count: %d.\n", globalcounter);
58
59    if (0 > pthread_mutex_destroy(&mutex)) {
60      printf("Failed to destroy mutex.\n");
61    }
62
63  }
```

Ausgabe – Seite 1

```
==24788== Helgrind, a thread error detector
==24788== Copyright (C) 2007-2012, and GNU GPL'd, by OpenWorks LLP et al.
==24788== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==24788== Command: ./threads
==24788==
[000] Thread starting.
[000] Thread alive.
[001] Thread starting.
[001] Thread alive.
[002] Thread starting.
[002] Thread alive.
==24788== ---Thread-Announcement------------------------------------------
==24788==
==24788== Thread #4 was created
==24788==    at 0x514D98E: clone (clone.S:76)
==24788==    by 0x4E3CF24: do_clone.constprop.4 (createthread.c:74)
==24788==    by 0x4E3E64D: pthread_create@@GLIBC_2.2.5 (createthread.c:244)
==24788==    by 0x4C2E870: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==24788==    by 0x4009A1: main (threads.c:47)
```

Ausgabe - Seite 2

```
==24788==
==24788== ----------------------------------------------------------------
==24788==
==24788== Thread #4: Attempt to re-lock a non-recursive lock I already hold
==24788==    at 0x4C2FB00: pthread_mutex_lock (in
/usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==24788==    by 0x4008BA: threadfn (threads.c:25)
==24788==    by 0x4C2EA06: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==24788==    by 0x4E3DF6D: start_thread (pthread_create.c:311)
==24788==    by 0x514D9CC: clone (clone.S:113)
==24788==  Lock was previously acquired
==24788==    at 0x4C2FC35: pthread_mutex_lock (in
/usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==24788==    by 0x4008BA: threadfn (threads.c:25)
==24788==    by 0x4C2EA06: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==24788==    by 0x4E3DF6D: start_thread (pthread_create.c:311)
==24788==    by 0x514D9CC: clone (clone.S:113)
==24788==
```

Ausgabe – Seite 3

```
==24788== ---Thread-Announcement------------------------------------------
==24788==
==24788== Thread #1 is the program's root thread
==24788==
==24788== ----------------------------------------------------------------
==24788==
==24788== Possible data race during write of size 4 at 0x601080 by thread #1
==24788== Locks held: none
==24788==    at 0x4009DD: main (threads.c:54)
==24788==
==24788== This conflicts with a previous read of size 4 by thread #4
==24788== Locks held: none
==24788==    at 0x400905: threadfn (threads.c:24)
==24788==    by 0x4C2EA06: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==24788==    by 0x4E3DF6D: start_thread (pthread_create.c:311)
==24788==    by 0x514D9CC: clone (clone.S:113)
==24788==
==24788== ----------------------------------------------------------------
==24788==
```

Ausgabe - Seite 4

```
==24788==
==24788== Lock at 0x6010E0 was first observed
==24788==    at 0x4C2F8EA: pthread_mutex_init (in
/usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==24788==    by 0x400942: main (threads.c:40)
==24788==
==24788== Possible data race during read of size 4 at 0x6010A4 by thread #1
==24788== Locks held: none
==24788==    at 0x4009F6: main (threads.c:57)
==24788==
==24788== This conflicts with a previous write of size 4 by thread #4
==24788== Locks held: 1, at address 0x6010E0
==24788==    at 0x4008DB: threadfn (threads.c:27)
==24788==    by 0x4C2EA06: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==24788==    by 0x4E3DF6D: start_thread (pthread_create.c:311)
==24788==    by 0x514D9CC: clone (clone.S:113)
==24788==
```

Ausgabe - Seite 5

```
==24788==
Total count: 3.
==24788== ------------------------------------------------------------
==24788==
==24788== Thread #1: pthread_mutex_destroy of a locked mutex
==24788==    at 0x4C2F978: pthread_mutex_destroy (in
/usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==24788==    by 0x400A16: main (threads.c:59)
==24788==
==24788== ------------------------------------------------------------
==24788==
==24788== Thread #1's call to pthread_mutex_destroy failed
==24788==    with error code 16 (EBUSY: Device or resource busy)
==24788==    at 0x4C2FA42: pthread_mutex_destroy (in
/usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==24788==    by 0x400A16: main (threads.c:59)
==24788==
[...]
```

Vermeidung von Fehlern

- Striktes Befolgen von Codierregeln
  - Korrektes Klammern
  - Kommentare nicht zum Deaktivieren von Code verwenden
  - Typsicherheit beachten
- Compilerwarnungen beachten
- Mutexe besonders genau verwenden

MISRA C

- Motor Industry Software Reliability Association
- Programmierstandard für C zur Vermeidung unklarer Definitionen im Standard
- Standard entspricht Verabschiedungsjahr
  - MISRA C:1998, MISRA C:2004, MISRA C:2012
  - MISRA C++:2008

Regelbeispiele

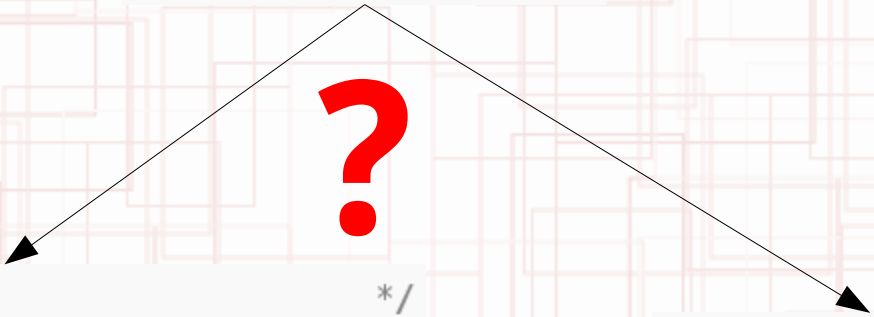
- Keine Zeilenkommentare
  - `// Kommentar`
- Keine verschachtelten Kommentare
  - Undefiniertes Verhalten im C-Standard

```
/* Code testweise deaktivieren
// i um 1 erhöhen
i++
/* Wenn Zähler i gleich a ist, MeineFunktion aufrufen um einen hoch */
if (i==a) { meineFunktion(a) }
*/
```

Quelle: Wikipedia

Zuweisungskonstrukte

```c
if ( i = a )
{
   /* Anweisung */
}
```




```c
/* Zuweisung                    */
i = a;
/* dann ein Vergleich           */
if ( i != 0 )
{
   /* Anweisung */
}
```

```c
if ( i == a )
{
   /* Anweisung */
}
```

Weitere…

- Kein Vergleich von Gleitkommazahlen
- Vorzeichenlose Konstanten kennzeichnen
  - `0xff ==> 0xffu`
- Verhindern einer Division durch Null
  - `if (b!=0) a/=b;`
- Keine Rekursion
- Operatorreihenfolgen
  - `(a && b || c) ==> ((a && b) || c)`