LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Arbeitsblatt 8
13.06.2022 - 19.06.2022

T8.1 Valgrind

Früher oder später schleicht sich in so gut wie jedes C Programm ein Fehler bezüglich der Speicherallokation oder -freigabe, oder bezüglich Speicherzugriffen ein. Gerade diese sind allerdings häufig schwer zu debuggen, da der Fehler oftmals nicht im direkten lokalen und/oder temporalen Kontext der eigentlichen Ursache auftritt. Wir möchten uns anhand des Programms `outerprod` nun ansehen, wie uns das Tool Valgrind helfen kann, die Ursache dieser Fehler dennoch aufzuspüren und zu beheben.

Material: `https://gra.caps.in.tum.de/m/outerprod.tar`

1. Besprechen Sie mit Ihrem Tutor zunächst den Aufbau und die grobe Funktionsweise des Programms.

   In der Funktion `test` werden zwei Arrays alloziert, befüllt und ausgegeben. Danach wird in der Funktion `outerprod` Speicher für die Matrix des äußeren Produkts alloziert und dieses berechnet. Schließlich wird das äußere Produkt in der Funktion `test` ausgegeben.

2. Kompilieren Sie das Programm mit `make` und führen Sie es aus. Verhält es sich wie erwartet?

   Das Programm stürzt mit einem* Segmentation Fault *ab.

3. Führen Sie das Programm zum Debuggen nun mit GDB aus (`gdb ./outerprod`; `r`). An welcher Stelle der Datei `outerprod.c` stürzt das Programm ab?

   Das Programm stürzt im Bereich der Zuweisung `mat[k + j] = u[i] * v[j]` ab.

4. Lassen Sie sich mit dem Befehl `p mat` den Pointer auf die Ausgabematrix anzeigen. Können Sie allein hieraus erkennen, warum das Programm abstürzt?

   Der Pointer `mat` ist ungleich `NULL`. Es handelt sich also nicht um einen Absturz aufgrund eines `NULL`-Pointer Zugriffs. Weitere Informationen über den Grund des Absturz liefert GDB nicht.

5. Führen Sie das Programm nun mit Valgrind aus: `valgrind ./outerprod`. Wie interpretieren Sie die Ausgabe? Was ist die *unmittelbare* Ursache für den Absturz?

   Valgrind warnt vor einem „Invalid write of size 8“ und gibt weiterhin als Grund für den Absturz „Access not within mapped region“ an. Es handelt sich also um einen Absturz aufgrund eines Schreibversuchs auf Speicher, auf den das Programm nicht zugreifen darf.

6. Wie groß ist der Speicherbereich, auf den zum Zeitpunkt des Programmabsturz zugegriffen wird? Wie groß *sollte* er sein?

   Valgrind gibt die Größe des Speicherbereichs mit „524,288 alloc'd" Bytes an. Der Speicherbereich selbst wurde in Zeile 24 mittels `malloc(mn * sizeof *mat)` ***alloziert. Da das Programm im Testfall für*** `m == 131074` ***und*** `n == 32768` ***abstürzt, würde man als Größe aber*** `131074 * 32768 * sizeof double == 34360262656` ***Bytes erwarten.***

7. Wie erklären Sie sich diese Diskrepanz?

   Sowohl `m` ***als auch*** `n` ***sind vom Typ*** `unsigned`***, also i.d.R. 32 Bit groß. Im letzten Testfall ist das*** **Ergebnis** ***der Multiplikation*** `mn = m * n` ***allerdings nicht mehr in 32 Bit darstellbar. In diesem Fall wird also nur das eigentliche Ergebnis modulo*** 2^{32} ***gespeichert***[1] ***und es gilt somit insbesondere*** `mn = 65536` ***und*** `mn * sizeof double == 524288`***.***

8. Informieren Sie sich im GCC Manual[2] über die Integer Overflow Built-ins und überlegen Sie, wie Sie diese nutzen können, um dieses und ähnliche Probleme allgemein zu verhindern.

   Die Built-ins erlauben es zwei Integer zu verrechnen, wobei der Rückgabewert der Built-in angibt, ob das Ergebnis im Ausgabetyp darstellbar ist. Ist dies der Fall wird `false` ***zurückgegeben, sonst*** `true`***. In der Funktion*** `outerprod` ***können wir so mittels*** `__builtin_umul_overflow` ***ein Ergebnis der Multiplikation*** `mn = m * n`***, das nicht mehr in 32 Bit darstellbar ist, abfangen und in diesem Fall einen*** `NULL`***-Pointer zurückgeben.***

```
23 double* outerprod(double* u, unsigned m, double* v, unsigned n) {
24     unsigned mn;
25     if(__builtin_umul_overflow(m, n, &mn))
26         return NULL;
27
28     double* mat = malloc(mn * sizeof *mat);
29     ...
```

   An dieser Stelle sei allerdings darauf hingewiesen, dass für Größenangaben immer `size_t` ***verwendet werden sollte. Zwar kann auch das Ergebnis einer arithmetischen Operation mit zwei*** `size_t`***-Werten nicht mehr als*** `size_t`***-Wert darstellbar sein; die Overflow Built-ins sind also nach wie vor nützlich. Nichtsdestotrotz erlaubt es*** `size_t` ***– definitionsgemäß – aber, den allozierbaren Speicher bestmöglich auszunutzen, wohingegen sich mit*** `unsigned` ***z.B. nur Arrays mit maximal*** 2^{32} ***Elementen auf einmal allozieren lassen.***

[1] Hier ist dieses Verhalten tatsächlich vom C-Standard garantiert, da es sich um *unsigned* 32-bit Integer handelt. Im Fall von *signed* Integern sind arithmetische Overflows allerdings undefiniertes Verhalten!

[2] `https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html`

S8.1 Leistungsoptimierung

Material: `https://gra.caps.in.tum.de/m/benchmark.tar`

1. Kompilieren Sie das Programm `matr` mit `make` und führen Sie es mit `time ./matr` aus. Welche benötigte Laufzeit gibt das Programm an? Wie groß ist der Unterschied zwischen der real benötigten Zeit und der angezeigten Zeit?

   Der Unterschied ist beachtlich groß:

```
$ time ./matr
Processing element 511,511...Took 0.056654 seconds

real    0m3.655s
user    0m1.000s
sys     0m0.616s
```

2. Um den Grund für den Unterschied herauszufinden, verwenden wir nun den *Profiler* `perf`. Dieser ist auf der `lxhalle` bereits installiert.[3] Führen Sie das Programm nun mit `perf record`[4] aus:

   `perf record ./matr`

   Und lassen Sie sich nach der Ausführung das Resultat anzeigen:

   `perf report`

   In welchen Funktionen wird am meisten Zeit verbracht? Entfernen Sie den (unnötigen) Aufruf in diese Funktion.

   Es wird ein Großteil der Zeit in `printf` und den dadurch aufgerufenen Unterfunktionen verbracht (sowie dem Kernel, der dann die I/O-Operation ausführt). Dieser Aufruf lässt sich einfach entfernen, und danach ist das Programm schon deutlich schneller. Beispielhafte Ausgabe:

```
# Overhead  Command  Shared Object  Symbol
# ........  .......  .............  ..............................
    26.03%  matr     matr           [.] compute
    16.76%  matr     libc-2.28.so   [.] vfprintf
    11.27%  matr     libc-2.28.so   [.] _IO_default_xsputn
    11.21%  matr     libc-2.28.so   [.] __strchrnul_avx2
     6.64%  matr     libc-2.28.so   [.] buffered_vfprintf
```

[3] Wir empfehlen Ihnen, soweit möglich, ein eigenes System für diese Aufgabe zu verwenden, da die `lxhalle` eine sehr restriktive Konfiguration hat.

[4] Dieser Befehl führt Ihr Programm aus, während `perf` im Hintergrund in regelmäßigen Abständen aufzeichnet, an welcher Stelle sich Ihr Programm befindet. Diese Information wird in der Datei `perf.data` gespeichert.

3. Kompilieren Sie das Programm erneut und führen Sie es erneut mit `perf record` aus. Lassen Sie sich wieder anzeigen, wo das Programm die meiste Zeit verbringt. Wie lässt sich das Problem beheben?

   Hinweis: Es empfiehlt sich, die Zahl der Iterationen beim Programmaufruf wie folgt zu erhöhen: `./matr -i 50`

   Nun wird ein großer Teil der Zeit damit verbracht, die Zeit zu messen:

```
# Overhead  Command  Shared Object  Symbol
# ........  .......  .............  ............................
#
    61.70%  matr     matr           [.] compute
    28.58%  matr     [vdso]         [.] __kernel_clock_gettime
     8.25%  matr     libc-2.24.so   [.] __clock_gettime
```

   Das Problem lässt sich beheben, indem die Aufrufe von `clock_gettime` vor die* äußerste *Schleife bewegt werden. Denn der Overhead der Schleife ist um ein Vielfaches geringer als das Messen der Zeit. Außerdem wird somit sicher gestellt, dass zwischen den `clock_gettime` Aufrufen hinreichend Zeit ist (Empfehlung:** > 1 ***Sekunde!), um Schwankungen aufgrund von externen Einflüssen zu vermeiden.

   Nach der Behebung sieht die Ausgabe wie folgt aus:

```
# Overhead  Command  Shared Object      Symbol
# ........  .......  .................  ..........................
#
    90.86%  matr     matr               [.] compute
     9.03%  matr     matr               [.] main
```

4. Nun kann `perf` nicht nur die benötigte Zeit anzeigen, sondern auch viele weitere Informationen, welche aus sog. *Performance Countern* des Prozessors ausgelesen werden können. Diese zählen im Hintergrund verschiedene Events mit, welche dabei helfen können, Performanzprobleme zu analysieren. Führen Sie `perf list` aus, um eine Übersicht über die Events, die von dem jeweiligen System (Prozessor, Betriebssystem) unterstützt werden, zu erhalten.

5. Im Folgenden betrachten wir das Event `cache-misses`, welches die Anzahl der Speicherzugriffe zählt, wo das Resultat nicht im schnellen Cache im Prozessor liegt, sondern erst aus dem verhältnismäßig langsamen Hauptspeicher geholt werden muss.

   Führen Sie das Programm wie folgt aus, um anstelle der Berechnungszeit die Zahl der Cache-Misses zu messen:

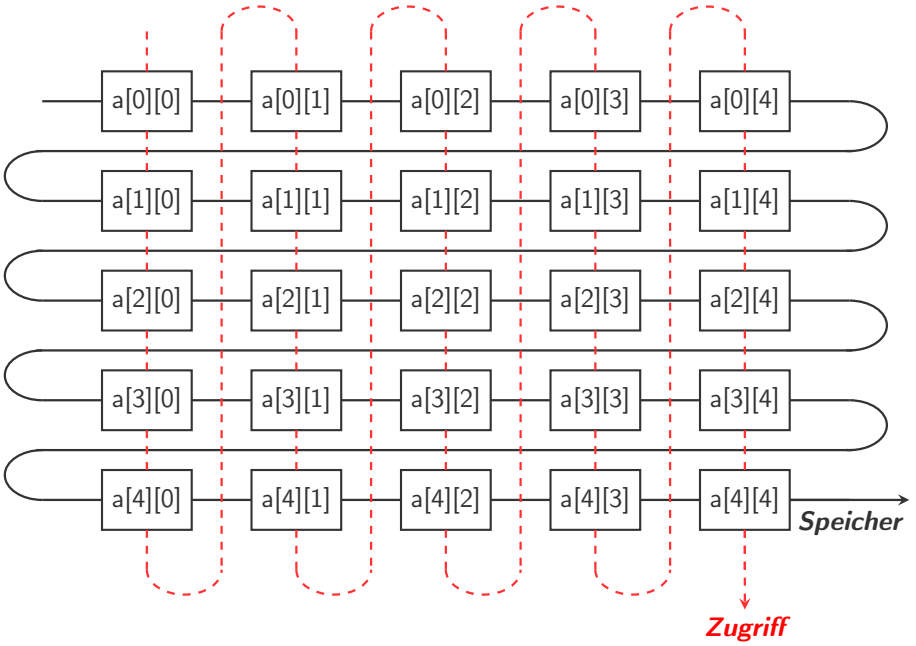
```
perf record -e cache-misses ./matr -i 50

perf report
```

Genauere Informationen können Sie erhalten, indem Sie eine Funktion auswählen (Navigation über Pfeil-Tasten, Auswahl mit *Enter*) und in dem folgenden Menü den Punkt *Annotate* wählen. Sie können die Ansicht mit `q` oder `Ctrl-C` verlassen.

An welchen Stellen verursacht das Programm viele Cache-Misses?

Die meisten Cache-Misses gibt es in der Funktion `compute`. Was in der* Annotate*-Ansicht auffällt ist, dass die Cache-Miss-Events nicht direkt am* Load *stehen, sondern an der folgenden Instruktion. Dies hängt damit zusammen, dass der Program Counter bereits erhöht wurde und der Prozessor darauf wartet, dass das Speichersystem den Wert zurückgibt.

6. Überlegen Sie sich, wie die Matrix im Speicher abgelegt ist und in welcher Reihenfolge darauf zugegriffen wird. Zeichnen Sie beide Reihenfolgen in folgende Grafik ein.



Ein zweidimensionales Array liegt bei C immer zeilenweise (schwarze Linie) im Speicher. Der Zugriff im Programm erfolgt aber immer spaltenweise (rote Linie). Durch den sprunghaften Speicherzugriff ist für den Prozessor nicht vorhersehbar, welches Element als nächstes verwendet wird, und kann daher die Elemente nicht schon vorher im Hintergrund in den Cache laden (Prefetching).

Durch welche kleine Veränderung lässt sich die Cache-Nutzung *deutlich* verbessern und damit die Performanz signifikant steigern?

Durch das vertauschen der beiden `for`-Schleifen über `i` und `j`.

P8.1 Hamming-Distanz [4 Pkt.]

Die Pinguine der Gattung ERA (kurz: ERA-P) verfügen über Augen, die lediglich Helligkeitswerte wahrnehmen[5] und diese pro wahrgenommen Bildpunkt digital im Wertebereich von 0–255 an das Gehirn weitergeben; eine Wahrnehmung ist dabei eine Liste bekannter Länge von Bildpunktdaten. Im Gehirn eines ERA-Pinguin sind Bilder von vergangenen Wahrnehmungen gespeichert. Ein Pinguin orientiert sich, in dem die aktuelle Wahrnehmung mit vormaligen Wahrnehmungen verglichen wird.

Da aber weder die Erinnerung der ERA-Pinguine noch die Übertragung der Wahrnehmung von den Augen an das Gehirn Error-Correcting-Codes benutzt, gehen sporadisch Daten verloren: Bei einigen Wahrnehmungen in der Erinnerung sind einzelne Bildpunkte nicht korrekt und bei der Übertragung der Daten von den Augen kommen manche Bildpunktdaten am Ende nur als zufälliges Rauschen an.

Deswegen realisieren die ERA-Pinguine den Bildvergleich, in dem die Anzahl der abweichenden Bildpunkte gezählt wird (*Hamming Distance*); die Erinnerung mit der geringsten Abweichung wird dann zur Orientierung genutzt. Insbesondere im Wasser ist eine extrem schnelle Orientierung überlebenswichtig, weshalb der Vergleich zwischen Wahrnehmung und Erinnerung so schnell wie möglich geschehen muss.

Aufgabe: Optimieren Sie folgende Funktion, welche die Anzahl der unterschiedlichen Elemente der Arrays `a` und `b` der gleichen Länge `n` berechnet, um die Überlebensfähigkeit der ERA-Pinguine zu erhöhen!

```
size_t hamming_dist(size_t n, const char a[n], const char b[n]) {
  size_t res = 0;
  for (size_t i = 0; i < n; i++)
    res += a[i] != b[i];
  return res;
}
```

Referenzlösung:

```
size_t hamming_dist(size_t n, const char a[n], const char b[n]) {
  size_t ub = n - (n % 16);

  __m128i sums = _mm_setzero_si128();
  __m128i ones = _mm_set1_epi8(1);
  for (size_t i = 0; i < ub; i += 16) {
    __m128i as = _mm_loadu_si128((const __m128i_u*) &a[i]);
    __m128i bs = _mm_loadu_si128((const __m128i_u*) &b[i]);
    __m128i cs = _mm_cmpeq_epi8(as, bs);
    cs = _mm_andnot_si128 (cs, ones); // eq/ff -> 00; noteq/00 -> 01
    cs = _mm_sad_epu8(cs, _mm_setzero_si128());
    sums = _mm_add_epi64(sums, cs);
  }

  size_t res = sums[0] + sums[1];
```

[5] Die Pinguine entstammen der Schwarz-Weiß-ERA.

```
16   for (size_t i = ub; i < n; ++i)
17     res += a[i] != b[i];
18   return res;
19 }
```

X8.1 ToUpper: MemeAssembly-Edition [2 Pkt. Bonus]

In dieser Einheit betrachten wir erneut die exzellente Funktion `toupper`, welche in einem String sämtliche unexzellente Klein-Buchstaben durch die entsprechenden exzellenten Groß-Buchstaben ersetzt. Diese Aufgabe wird Ihnen eine Einführung in die exzellente Programmiersprache MemeAssembly geben.

1. Verwenden Sie die Dokumentation[6], um für die folgenden Probleme geeignete Commands zu finden:
   - Laden der Zahl 42 in das Register `rax`

     ```
     rax is brilliant, but I like 42
     ```

   - `rcx` auf den Stack pushen

     ```
     stonks rcx
     ```

   - Sprung, falls `rax` = 0

     ```
     corporate needs you to find the difference between rax and 0
     [...]
     they're the same picture
     ```

   - Ein Byte von der Speicheradresse in `rdi` in das Register `al` laden

     Hinweis: Der Register-Suffix `do you know de wey` führt zu einem Speicherzugriff (äquivalent zu `[Register]`)

     ```
     al is brilliant, but I like rdi do you know de wey
     ```

   - Vergleich zwischen `al` und 97, mit dem Sprung zu einem jeweiligen Marker – je nachdem, welcher Wert größer ist

     ```
     who would win? al or 97?
     [...]
     al wins
     [...]
     97 wins
     ```

[6] `https://kammt.github.io/MemeAssembly`

- Berechnen von 2^3 in `rax`

```
    rax is brilliant, but I like 2
    rax UNLIMITED POWER 3
```

- Programmabsturz durch einen Segmentation Fault

```
    guess I'll die
```

- Durch Zufall während des Kompilierens die Hälfte aller Codezeilen löschen

```
    perfectly balanced as all things should be
```

2. Beginnen Sie mit dem groben Schleifenkonstrukt: Laden Sie einen Buchstaben aus dem Speicher und springen Sie aus der Schleife heraus, falls es sich um ein `NULL`-Byte handelt. Vergessen Sie dabei nicht, den Pointer zu inkrementieren.

```
banana
al is brilliant, but I like rdi do you know de wey
corporate needs you to find the difference between al and 0

    upvote rdi
    where banana

they're the same picture
right back at ya, buckaroo
```

3. Für den Zweck des Vergleichs bietet sich der Command `who would win?` an. Überprüfen Sie nun, ob der Buchstabe ein (unexzellenter) Kleinbuchstabe ist. Beachten Sie hierbei, dass Sie eventuell den (potentiell unexzellenten) Buchstaben in ein anderes Register verschieben müssen, um einen neuen Sprung-Marker erstellen zu können. ***Für den Vergleich 97 < al < 122 benötigen wir zwei*** *`who would win?`-Vergleiche:*

```
who would win? al or 97
al wins
cl is brilliant, but I like al
who would win? cl or 122
122 wins

what the hell happened here? Falls wir hier landen, ist es ein
    unexzellenter Kleinbuchstabe

cl wins
97 wins
```

4. Verwandeln Sie nun den Buchstaben in einen exzellenten Großbuchstaben und schreiben Sie ihn in den Speicher zurück.

   Erinnerung: Der Suffix `do you know de wey` interpretiert einen Register-Parameter als eine Speicheradresse

```
parry 32 you filthy casual cl
rdi do you know de wey is brilliant, but I like cl
```

5. Sie werden nun feststellen, dass Ihr Code zwar (hoffentlich) funktioniert, jedoch noch zu unexzellent ist, um einen Score zu erhalten. Werden Sie kreativ - wie könnte man den Code so weit verlangsamen, ohne dabei einen Timeout auszulösen?

```
What the hell happened here? This program takes a pointer to a string in rdi, the string is modified in-place
I like to have fun, fun, fun, fun, fun, fun, fun, fun, fun, fun
    toupper_memeasm

    banana
    al is brilliant, but I like rdi do you know de wey
    corporate needs you to find the difference between al and 0

        who would win? al or 97
        al wins
        cl is brilliant, but I like al
        who would win? cl or 122
        122 wins

        what the hell happened here? Embrace excellence - convert it to upper case
        parry 32 you filthy casual cl
        rdi do you know de wey is brilliant, but I like cl

        cl wins
        97 wins

        upvote rdi

        What the hell happened here? Calculate rax^20 to slow down the program - just because
        rax UNLIMITED POWER 20

        where banana

    they're the same picture
    right back at ya, buckaroo
```

Q8.1 Quiz [4 Pkt.] (siehe Praktikumswebsite)

Q8.2 Praktikumsordnung [2 Pkt.] (siehe Praktikumswebsite)