Technische Informatik 2 WS 2016/17 C04
Tutor: Marc Hildebrandt/ Bingbin Yu

Michael Schmidt
Stanislav Telis
Dominique Schulz
Norman Lipkow

Übungsblatt 5

1	2	3	4	5	$\sum$
	2	2.5			

Lösungsvorschlag
Abgabe: 05.12.2016

Aufgabe 1

Wir haben das Programm `factorial` mittels `gdb` gestartet.

```
1  do_sc@x14 /home/ti2/ueb/05
2  ->ls
3  Makefile  factorial  factorial.c
4  do_sc@x14 /home/ti2/ueb/05
5  ->gdb factorial
6  GNU gdb (GDB) 7.4.1-debian
7  Copyright (C) 2012 Free Software Foundation, Inc.
8  License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
9  This is free software: you are free to change and redistribute it.
10 There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
11 and "show warranty" for details.
12 This GDB was configured as "x86_64-linux-gnu".
13 For bug reporting instructions, please see:
14 <http://www.gnu.org/software/gdb/bugs/>...
15 Reading symbols from /home/ti2/ueb/05/factorial...(no debugging symbols found)...done.
16 (gdb)
```

a)

Anschließend führen wir in `gdb` den Befehl `disassemble main` aus um uns den virtuellen Adressraum anzeigen zu lassen, welche die Funktion `main` des Programms `factorial` einnimmt.

```
16 (gdb) disassemble main
17 Dump of assembler code for function main:
18    0x000000000040057d <+0>:     push   %rbp
19    0x000000000040057e <+1>:     mov    %rsp,%rbp
20    0x0000000000400581 <+4>:     sub    $0x20,%rsp
21    0x0000000000400585 <+8>:     mov    %edi,-0x14(%rbp)
22    0x0000000000400588 <+11>:    mov    %rsi,-0x20(%rbp)
23    0x000000000040058c <+15>:    cmpl   $0x1,-0x14(%rbp)
24    0x0000000000400590 <+19>:    jle    0x4005b1 <main+52>
25    0x0000000000400592 <+21>:    mov    -0x20(%rbp),%rax
26    0x0000000000400596 <+25>:    add    $0x8,%rax
27    0x000000000040059a <+29>:    mov    (%rax),%rax
28    0x000000000040059d <+32>:    mov    $0xa,%edx
29    0x00000000004005a2 <+37>:    mov    $0x0,%esi
30    0x00000000004005a7 <+42>:    mov    %rax,%rdi
```

```
31      0x00000000004005aa <+45>:    callq  0x400430 <strtol@plt>
32      0x00000000004005af <+50>:    jmp    0x4005b6 <main+57>
33      0x00000000004005b1 <+52>:    mov    $0x0,%eax
34      0x00000000004005b6 <+57>:    mov    %rax,-0x8(%rbp)
35      0x00000000004005ba <+61>:    mov    -0x8(%rbp),%rax
36      0x00000000004005be <+65>:    mov    %rax,%rdi
37      0x00000000004005c1 <+68>:    callq  0x40054c <factorial>
38      0x00000000004005c6 <+73>:    mov    %rax,%rdx
39      0x00000000004005c9 <+76>:    mov    -0x8(%rbp),%rax
40  ---Type <return> to continue, or q <return> to quit---
41      0x00000000004005cd <+80>:    mov    %rax,%rsi
42      0x00000000004005d0 <+83>:    mov    $0x40069c,%edi
43      0x00000000004005d5 <+88>:    mov    $0x0,%eax
44      0x00000000004005da <+93>:    callq  0x400410 <printf@plt>
45      0x00000000004005df <+98>:    mov    $0x0,%eax
46      0x00000000004005e4 <+103>:   leaveq
47      0x00000000004005e5 <+104>:   retq
48  End of assembler dump.
49  (gdb)
```

Wir sehen nun das die Funktion `main` den virtuellen Adressraum von `0x000000000040057d` (Zeile 18) bis `0x00000000004005e5` (Zeile 47) belegt.

Mittels `disassemble factorial` wollen wir auch noch den virtuellen Adressraum der Funktion `factorial` in Erfahrung bringen.

```
49  (gdb) disassemble factorial
50  Dump of assembler code for function factorial:
51      0x000000000040054c <+0>:     push   %rbp
52      0x000000000040054d <+1>:     mov    %rsp,%rbp
53      0x0000000000400550 <+4>:     mov    %rdi,-0x18(%rbp)
54      0x0000000000400554 <+8>:     movq   $0x1,-0x8(%rbp)
55      0x000000000040055c <+16>:    jmp    0x400570 <factorial+36>
56      0x000000000040055e <+18>:    mov    -0x8(%rbp),%rax
57      0x0000000000400562 <+22>:    imul   -0x18(%rbp),%rax
58      0x0000000000400567 <+27>:    mov    %rax,-0x8(%rbp)
59      0x000000000040056b <+31>:    subq   $0x1,-0x18(%rbp)
60      0x0000000000400570 <+36>:    cmpq   $0x0,-0x18(%rbp)
61      0x0000000000400575 <+41>:    jne    0x40055e <factorial+18>
62      0x0000000000400577 <+43>:    mov    -0x8(%rbp),%rax
63      0x000000000040057b <+47>:    pop    %rbp
64      0x000000000040057c <+48>:    retq
65  End of assembler dump.
66  (gdb)
```

Hier sehen wir nun, dass sich die Funktion `factorial` von der virtuellen Adresse `0x000000000040054c` (Zeile 51) bis zur virtuellen Adresse `0x000000000040057c` (Zeile 64) erstreckt.

b)

Wenn der User-Stackpointer `%rsp` bei Eintritt der Funktion `factorial` den Wert `0x7fffffffdf60` hat, dann wird auf die virtuellen Adressen `0x7fffffffdf48` und `0x7fffffffdf58` zugriffen.
Es wird auf die virtuelle Adresse `0x7fffffffdf48` zugegriffen, weil der Wert von `%rbp` auf den

Wert des User-Stackpointer (also `0x7fffffffdf60`) gesetzt wird und dort dann um `-0x18` Bytes verschoben wird. Demnach ist `0x7fffffffdf60` − `0x18` = `0x7fffffffdf48`.
Analog dazu wird auf die virtuelle Adresse `0x7fffffffdf58` zugegriffen, weil der `%rbp` weiterhin auf `0x7fffffffdf60` zeigt und dann um `-0x8` Bytes verschoben wird. Daraus folgt `0x7fffffffdf60` − `0x8` = `0x7fffffffdf58`.

c)

Wir nutzen das Kommando `disassemble printf`, um die virtuellen Adressen der Zeichenketten zu erfahren. Uns wird folgendes auf der Konsole angezeigt:

```
(gdb) disassemble printf
Dump of assembler code for function __printf:
   0x00007ffff7a9e190 <+0>:     sub    $0xd8,%rsp
   0x00007ffff7a9e197 <+7>:     movzbl %al,%eax
   0x00007ffff7a9e19a <+10>:    mov    %rdx,0x30(%rsp)
   0x00007ffff7a9e19f <+15>:    lea    0x0(,%rax,4),%rdx
   0x00007ffff7a9e1a7 <+23>:    lea    0x44(%rip),%rax   # 0x7ffff7a9e1f2 <__printf+98>
   0x00007ffff7a9e1ae <+30>:    mov    %rsi,0x28(%rsp)
   0x00007ffff7a9e1b3 <+35>:    mov    %rcx,0x38(%rsp)
   0x00007ffff7a9e1b8 <+40>:    mov    %rdi,%rsi
   0x00007ffff7a9e1bb <+43>:    sub    %rdx,%rax
   0x00007ffff7a9e1be <+46>:    lea    0xcf(%rsp),%rdx
   0x00007ffff7a9e1c6 <+54>:    mov    %r8,0x40(%rsp)
   0x00007ffff7a9e1cb <+59>:    mov    %r9,0x48(%rsp)
   0x00007ffff7a9e1d0 <+64>:    jmpq   *%rax
   0x00007ffff7a9e1d2 <+66>:    movaps %xmm7,-0xf(%rdx)
   0x00007ffff7a9e1d6 <+70>:    movaps %xmm6,-0x1f(%rdx)
   0x00007ffff7a9e1da <+74>:    movaps %xmm5,-0x2f(%rdx)
   0x00007ffff7a9e1de <+78>:    movaps %xmm4,-0x3f(%rdx)
   0x00007ffff7a9e1e2 <+82>:    movaps %xmm3,-0x4f(%rdx)
   0x00007ffff7a9e1e6 <+86>:    movaps %xmm2,-0x5f(%rdx)
   0x00007ffff7a9e1ea <+90>:    movaps %xmm1,-0x6f(%rdx)
   0x00007ffff7a9e1ee <+94>:    movaps %xmm0,-0x7f(%rdx)
---Type <return> to continue, or q <return> to quit---
   0x00007ffff7a9e1f2 <+98>:    lea    0xe0(%rsp),%rax
   0x00007ffff7a9e1fa <+106>:   mov    %rsp,%rdx
   0x00007ffff7a9e1fd <+109>:   movl   $0x8,(%rsp)
   0x00007ffff7a9e204 <+116>:   movl   $0x30,0x4(%rsp)
   0x00007ffff7a9e20c <+124>:   mov    %rax,0x8(%rsp)
   0x00007ffff7a9e211 <+129>:   lea    0x20(%rsp),%rax
   0x00007ffff7a9e216 <+134>:   mov    %rax,0x10(%rsp)
   0x00007ffff7a9e21b <+139>:   mov    0x338cee(%rip),%rax        # 0x7ffff7dd6f10
   0x00007ffff7a9e222 <+146>:   mov    (%rax),%rdi
   0x00007ffff7a9e225 <+149>:   callq  0x7ffff7a93330 <_IO_vfprintf_internal>
   0x00007ffff7a9e22a <+154>:   add    $0xd8,%rsp
   0x00007ffff7a9e231 <+161>:   retq
End of assembler dump.
```

Hier sehen wir, dass beim Lesen der Zeichenkette auf die virtuellen Adressen `0x7ffff7a9e1f2` und `0x7ffff7dd6f10` zugegriffen wird.

d)

In der Aufgabestellung wurden einige Informationen vorgegeben unter anderem die Page-Tabelle und folgendes:

Page-Größe ist 4 KiB = 4096 B = 0x1000 B
Offset von Page-Frame 0 ist `0x1000000`

1. Physische Adressen zu den virtuellen Adressen von der Funktion `main()`:

Virtuelle Adresse ist `0x40057d`

Page(`0x40057d`) = $floor$(`0x40057d`/`0x1000`) = `0x400` = 1024_{10}
Offset(`0x40057d`) = `0x40057d` mod `0x1000` = `0x57d`

Nachschauen in der Page-Tabelle Page 1024 → Page-Frame 17363 = `0x43d3`

Physische Adresse: `0x43d3` ∗ `0x1000` + `0x57d` + `0x1000000` = `0x53d357d`

Virtuelle Adresse ist `0x4005e5`

Page(`0x4005e5`) = floor(`0x4005e5` / `0x1000`) = `0x400` = 1024_{10}
Offset(`0x4005e5`) = `0x4005e5` mod `0x1000` = `0x5e5`

Nachschauen in der Page-Tabelle Page 1024 → Page-Frame 17363 = `0x43d3`

Physische Adresse: `0x43d3` ∗ `0x1000` + `0x5e5` + `0x1000000` = `0x53d35e5`

2. Physische Adressen zu den virtuellen Adressen der Funktion `factorial()`:

Virtuelle Adresse ist `0x40054c`

Page(`0x40054c`) = floor(`0x40054c` / `0x1000`) = `0x400` = 1024_{10}
Offset(`0x40054c`) = `0x40054c` mod `0x1000` = `0x54c`

Nachschauen in der Page-Tabelle Page 1024 → Page-Frame 17363 = `0x43d3`

Physische Adresse: `0x43d3` ∗ `0x1000` + `0x54c` + `0x1000000` = `0x53d354c`

Virtuelle Adresse ist `0x40057c`

Page(`0x40057c`) = floor(`0x40057c` / `0x1000`) = `0x400` = 1024_{10}
Offset(`0x40057c`) = `0x40057c` mod `0x1000` = `0x57c`

Nachschauen in der Page-Tabelle Page 1024 → Page-Frame 17363 = `0x43d3`

Physische Adresse: `0x43d3` ∗ `0x1000` + `0x57c` + `0x1000000` = `0x53d357c`

3. Physische Adressen zu den virtuellen Adressen von Aufgabeteil b):

Virtuelle Adresse ist `0x7fffffffdf48`

Page(`0x7fffffffdf48`) = floor(`0x7fffffffdf48` / `0x1000`) = `0x7fffffffd` = 34359738365_{10}
Offset(`0x7fffffffdf48`) = `0x7fffffffdf48` mod `0x1000` = `0xf48`

Nachschauen in der Page-Tabelle Page 34359738365 → **Page-Fault** → wir wählen freies Page-Frame 8080 = `0x1f90`

Physische Adresse: `0x1f90` ∗ `0x1000` + `0xf48` + `0x1000000` = `0x2f90f48`

Virtuelle Adresse ist `0x7fffffffdf58`

Page(`0x7fffffffdf58`) = floor(`0x7fffffffdf58` / `0x1000`) = `0x7fffffffd` = 34359738365_{10}
Offset(`0x7fffffffdf58`) = `0x7fffffffdf58` mod `0x1000` = `0xf58`

Nachschauen in der Page-Tabelle Page 34359738365 → **Page-Fault** → wir wählen freies Page-Frame 8900 = `0x22c4`

Physische Adresse: `0x22c4` ∗ `0x1000` + `0xf58` + `0x1000000` = `0x32c4f58`

4. Physische Adressen zu den virtuellen Adressen von Aufgabenteil c):

Virtuelle Adresse ist `0x7ffff7a9e1f2`

Page(`0x7ffff7a9e1f2`) = floor(`0x7ffff7a9e1f2`) / `0x1000`) = `0x7ffff7a9e` = 34359704222_{10}
Offset(`0x7ffff7a9e1f2`) = `0x7ffff7a9e1f2` mod `0x1000` = `0x1f2`

Nachschauen in der Page-Tabelle Page 34359704222 → **Page-Fault** → wir wählen freies Page-Frame 13599 = `0x351f`

Physische Adresse: `0x351f` ∗ `0x1000` + `0x1f2` + `0x1000000` = `0x451f1f2`

Virtuelle Adresse ist `0x7ffff7dd6f10`

Page(`0x7ffff7dd6f10`) = floor(`0x7ffff7dd6f10`) / `0x1000`) = `0x7ffff7dd6` = 34359705046_{10}
Offset(`0x7ffff7dd6f10`) = `0x7ffff7dd6f10` mod `0x1000` = `0xf10`

Nachschauen in der Page-Tabelle Page 34359705046 → **Page-Fault** → wir wählen freies Page-Frame 15777 = `0x3da1`

Physische Adresse: `0x3da1` ∗ `0x1000` + `0xf10` + `0x1000000` = `0x4da1f10`

Aufgabe 2

Wieviele Indirekt-Einträge passen in einen Indirekt Block?
Ein indirekter Verweis kann 128 Einträge/Adressen speichern. Da: 512 Byte (ein Datenblock) / 4 Byte (Adresse auf direkten oder indirekten Datenblock = 128

Wieviele Datenblöcke werden benötigt?
33.696.325 B / 512 B = 65813,1349 → 65814 Datenblöcke + 1 (für den Inode-Block) = 65815 Datenblöcke (+ die indirekten Blöcke, s. unten) 65814 Datenblöcke + 1 Inode +0.5

Wieviele Indirekt-Blöcke werden benötigt?
Auf 10 Datenblöcke wird direkt verwiesen. Also 10 * 512 B = 5120 B Ein indirekter Block kann auf 128 * 512 B (65.536 B) Daten verweisen. 1 doppelt indirekter Verweis kann auf 128^2 * 512 B (8.388.608 B) Daten verweisen 1 dreifach indirekter Verweis kann auf 128^3 * 512 B (1.073.741.824 B)

33.696.325 - 8.388.608 - 65.536 - 5120 = 25.237.061 bleiben übrig wenn wir den einfachen und doppelten Indirekt-Blöcke auslasten. Wir wissen, dass wir mit 1 doppelten Verweis 8.388.608 B ansprechen. Wir brauchen noch ca. 25.237.061. Also:

25.237.061 B / 8.388.608 B = 3,00849211. D.h. wir brauchen 4 Verweise auf doppelt indirekte Verweise. Jedoch müssen wir die Verweise nicht komplett auslasten. Wir können mit 3 Verweisen auf doppelt indirekte Verweise 25.165.824 B Daten ansprechen. Uns fehlt nun also 25.237.061 - 25.165.824 = 71.237 B

Mit einem indirekten Verweis können wir auf 65.536 B Daten verweisen. 71.237 - 65.536 = 5701 B

In einer weiteren indirekten Verweis brauchen wir nun 5701 / 512 Verweise = 11,13 → 12 direkte Verweise

einfach+zweifach:130: +0.5

Wir haben nun einen indirekten Block komplett ausgelastet. → 1 Wir haben einen doppelt indirekten Verweis komplett ausgelastet → 128 + 1 = 129 Wir haben 3 weitere doppelt indirekte Verweise aus der dreifach indirekten Verweis → 129 * 3 = 387 + 1 Und wir haben 1 kompletten weiteren indirekten Datenblock und einen nicht komplett gefüllten → 3

dreifach: 391 +0.5

Also: 1+129+388+3 = 521 indirekte Datenblöcke → 521 * 512 B = 270.920 B = 270 MB

Das heißt wir bräuchten mindestens 65.815 + 521 = 66.336 Datenblöcke auf der Platte. Das wären 66.336 * 512 B = 33.964.032 B Speicher der auf der Platte benötigt werden würde.

66336 Datenblöcke +0.5

Aufgabe 3

Zu `open(''/home/ti2/archive/nikolaus.avi'', O_RDONLY)`:

Da der Inode für / sich bereits im Inode-Tabelle befindet wird danach der Datenblock für / in den Cache geladen. Von Block 2 aus wird dann der Inode für das Verzeichnis home geladen. Dem entsprechend wird Block 9 in den Buffer-Cache geladen. Der Block 9 verweist auf das Inode für das Verzeichnis ti2, deshalb wird der Block 2000 in den Buffer-Cache geladen. Nun wird der Inode von archive gelesen und der Block 3101 in den Buffer-Cache geladen, hier ist dann ein Verweis auf die Datei nikolaus.avi mit dem Inode 12783, dann wird der Datenblock 50 gelesen und ein Verweis auf den Inode der Datei nikolaus.avi in den Hauptspeicher abgelegt.

+1P

Zu `open(''/home/ti2/meta'', O_RDWR)`:

Inode für / bereits in Inode-Tabelle und Block 2 im Buffer-Cache. Dann wird das Inode für das Verzeichnis home geladen. Also wieder Block 9 in den Buffer-Cache. Der Datenblock verweist auf das Verzeichnis home mit dem Inode 36, hier wird also Block 9 in den Buffer-Cache geladen. Dann wird der Inode (112) der Datei meta ausgelesen und folglich der Datenblock 8521 in den Buffer-Cache. Nun wird ein Verweis zum Inode der Datei meta in den Hautspeicher gelegt.

+1P

lseek(): Verschieben des Positionszeigers von f auf 12 * 1024^2 - 1 - 10000 = 12572911
Datenblocks 24556 (12572911/512)

Mit dem `lseek()` Systemaufruf wird zunächst der Pointer der Datei nikolaus.avi auf 2783 gesetzt. Beim `read()` Systemaufruf müssen 4096 Bytes, also die Länge von 8 Datenblöcken, gelesen werden. Um die Datei lesen zu können, müssen entsprechende Datenblöcke in den Hauptspeicher gelesen werden. Geht man von von 512 Bytes pro Datenblock aus geht bei der Stelle 2561 ein neuer Datenblock der Datei nikolaus.avi los (da 4* 512 B = 2560 Bytes). Um die Datei von Byte 2783-6879 nun lesen zu können, muss der Datenblock von 2561-3072, so wie 4 weitere Datenblöcke (also insgesamt von 2561-7168) in den Hauptspeicher geladen werden. Beim `write()` Befehl muss nichts mehr von der Platte geladen werden. Hier findet ein "reclaimßtatt und das zuvor gelesene wird in die Datei meta geschrieben und auf der Platte gespeichert.

+0.5P

hinreichend korrekte Beschreibung der Auswirkungen von lseek() und read()/write() (insbesondere Zugriff auf Datenblöcke, die über Indirektblöcke erreicht werden)