Technische Informatik 2
Tutor: Marc/Bingbin

WS 2016/17

C-01

Habermann, Paul
Köster, Joschka
Rohde, Florian

Übungsblatt 4

1	2	3	4	5	$\sum$

Lösungsvorschlag
Abgabe: 28.11.2016

Aufgabe 1

Aufgabe 1.a virtueller Adressraum des Prozesses, den die Funktionen einnehmen

Um den Adressraum zu extrahieren, haben wir im gegebenen Programm mittels `gdb` breakpoints jeweils an den Anfang und das Ende der beiden Funktionen gesetzt.
Ergebnis:

Anfang main: 0x000000000040057d

```
Breakpoint 1, 0x000000000040057d in main ()

(gdb) info frame
Stack level 0, frame at 0x7fffffffe6f0:
 rip = 0x40057d in main; saved rip 0x7ffff7a6eead
 Arglist at 0x7fffffffe6e0, args:
 Locals at 0x7fffffffe6e0, Previous frame's sp is 0x7fffffffe6f0
 Saved registers:
  rip at 0x7fffffffe6e8
(gdb) info stack
#0 0x000000000040057d in main ()
```

Ende main: 0x00000000004005e5

```
Breakpoint 4, 0x00000000004005e5 in main ()
(gdb) info frame
Stack level 0, frame at 0x7fffffffe6e8:
 rip = 0x4005e5 in main; saved rip 0x7ffff7a6eead
 Arglist at unknown address.
 Locals at unknown address, Previous frame's sp is 0x7fffffffe6f0
 Saved registers:
  rip at 0x7fffffffe6e8
```

Anfang factorial: 0x000000000040054c

```
Breakpoint 3, 0x000000000040054c in factorial ()
(gdb) info frame
Stack level 0, frame at 0x7fffffffe6c0:
 rip = 0x40054c in factorial; saved rip 0x4005c6
 called by frame at 0x7fffffffe6f0
 Arglist at 0x7fffffffe6b0, args:
 Locals at 0x7fffffffe6b0, Previous frame's sp is 0x7fffffffe6c0
 Saved registers:
  rip at 0x7fffffffe6b8
(gdb) info stack
#0 0x000000000040054c in factorial ()
#1 0x00000000004005c6 in main ()
```

Ende factorial: 0x000000000040057c

```
Breakpoint 4, 0x000000000040057c in factorial ()
(gdb) info frame
Stack level 0, frame at 0x7fffffffe6b8:
 rip = 0x40057c in factorial; saved rip 0x4005c6
 called by frame at 0x7fffffffe6f0
 Arglist at 0x7fffffffe6e0, args:
 Locals at 0x7fffffffe6e0, Previous frame's sp is 0x7fffffffe6c0
 Saved registers:
  rip at 0x7fffffffe6b8
(gdb) info stack
#0 0x000000000040057c in factorial ()
#1 0x00000000004005c6 in main ()
```

Auch zu sehen sind diese Adressen, wenn man die Funktionen main und factorial dissasambelt. Eine Ausdehnung findet in strtol und printf (dazu später mehr) statt.

```
  0x00007ffff7a87af0 <+0>: mov 0x34f271(%rip),%rax # 0x7ffff7dd6d68
  0x00007ffff7a87af7 <+7>: xor %ecx,%ecx
  0x00007ffff7a87af9 <+9>: mov %fs:(%rax),%r8
  0x00007ffff7a87afd <+13>: jmpq 0x7ffff7a87b50 <*__GI_____strtol_l_internal>
```

Aufgabe 1.b

Aufgabe 1.c printf

```
(gdb) disass main
Dump of assembler code for function main:
   0x000000000040057d <+0>: push %rbp
   0x000000000040057e <+1>: mov %rsp,%rbp
   0x0000000000400581 <+4>: sub $0x20,%rsp
   0x0000000000400585 <+8>: mov %edi,-0x14(%rbp)
   0x0000000000400588 <+11>: mov %rsi,-0x20(%rbp)
   0x000000000040058c <+15>: cmpl $0x1,-0x14(%rbp)
   0x0000000000400590 <+19>: jle 0x4005b1 <main+52>
   0x0000000000400592 <+21>: mov -0x20(%rbp),%rax
   0x0000000000400596 <+25>: add $0x8,%rax
   0x000000000040059a <+29>: mov (%rax),%rax
   0x000000000040059d <+32>: mov $0xa,%edx
   0x00000000004005a2 <+37>: mov $0x0,%esi
   0x00000000004005a7 <+42>: mov %rax,%rdi
   0x00000000004005aa <+45>: callq 0x400430 <strtol@plt>
   0x00000000004005af <+50>: jmp 0x4005b6 <main+57>
   0x00000000004005b1 <+52>: mov $0x0,%eax
   0x00000000004005b6 <+57>: mov %rax,-0x8(%rbp)
   0x00000000004005ba <+61>: mov -0x8(%rbp),%rax
   0x00000000004005be <+65>: mov %rax,%rdi
   0x00000000004005c1 <+68>: callq 0x40054c <factorial>
   0x00000000004005c6 <+73>: mov %rax,%rdx
   0x00000000004005c9 <+76>: mov -0x8(%rbp),%rax
   0x00000000004005cd <+80>: mov %rax,%rsi
   0x00000000004005d0 <+83>: mov $0x40069c,%edi
   0x00000000004005d5 <+88>: mov $0x0,%eax
   0x00000000004005da <+93>: callq 0x400410 <printf@plt>
   0x00000000004005df <+98>: mov $0x0,%eax
   0x00000000004005e4 <+103>: leaveq
   0x00000000004005e5 <+104>: retq
End of assembler dump.

(gdb) x/s 0x40069c
0x40069c: "factorial(%lu) = %lu\n"
```

Wir gehen also davon aus, dass die Adresse 0x40069c, die den gesuchten String enthält, die Antwort ist.

Der Anfang/das Ende von printf sind 0x00007ffff7a9e190/0x00007ffff7a9e231. In der Funktion findet eine Ausdehung statt und zwar auf: _IO_vfprintf_internal Diese Funktion befindet sich dem virtuellen Adressbereich von 0x00007ffff7a93330 - 0x00007ffff7a9894d

Aufgabe 1.d Adressbereiche

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

Page(ADRESSE) = $\lfloor \frac{(ADRESSE)}{0x1000} \rfloor$ = ?

Offset(ADRESSE) = (ADRESSE) mod 0x1000 = ?

Dann für die Adresse in der entsprechenden Page den Pageframe heraussuchen,
z.B. Page $0 -> 10342$,

Physikalische Adresse = $(Pageframe) \cdot 0x1000 + (Offset) + 0x1000000$. Berechnung der Adressen:

Einige Adressbereich würden keine korrekte Page-Adresse zurückliefern deswegen, haben wir auf deren Rechungen verzichtet.

Aufgabe 1.d.1 Adressen aus 1a)

Wir haben in 1a) folgende Adressen gefunden:

- $0x000000000040057d$;
  - Page = $\lfloor \frac{0x000000000040057d}{0x1000} \rfloor = 1024 \rightarrow PageFrame = 17363(0x43D3)$;
  - Offset = $0x000000000040057d \bmod 0x1000 = 0x57D$
  - Phys. Adresse = $0x43D3 \cdot 0x1000 + 0x57D + 0x1000000 = 0x53D357D$
- $0x00000000004005e5$;
  - Page = $\lfloor \frac{0x00000000004005e5}{0x1000} \rfloor = 1024 \rightarrow PageFrame = 17363$.
  - Offset = $0x00000000004005e5 \bmod 0x1000 = 0x5E5$
  - Phys. Adresse = $0x43D3 \cdot 0x1000 + 0x5E5 + 0x1000000 = 0x53D35E5$
- $0x000000000040054c$;
  - Page = $\lfloor \frac{0x000000000040054c}{0x1000} \rfloor = 1024 \rightarrow PageFrame = 17363$.
  - Offset = $0x000000000040054c \bmod 0x1000 = 0x54C$
  - Phys. Adresse = $0x43D3 \cdot 0x1000 + 0x54C + 0x1000000 = 0x53D354C$
- $0x000000000040057c$;
  - Page = $\lfloor \frac{0x000000000040057c}{0x1000} \rfloor = 1024 \rightarrow PageFrame = 17363$.

- Offset = $0x000000000040057c \bmod 0x1000 = 0x57C$
- Phys. Adresse = $0x43D3 \cdot 0x1000 + 0x57C + 0x1000000 = 0x53D357C$

Aufgabe 1.d.2 Adressen aus 1b)

Aufgabe 1.d.3 Adressen aus 1c)

- $0x40069c$;
  - Page = $\lfloor \frac{0x40069c}{0x1000} \rfloor = 1024 \rightarrow PageFrame = 17363$;
  - Offset = $0x40069c \bmod 0x1000 = 0x69C$
  - Phys. Adresse = $0x43D3 \cdot 0x1000 + 0x69C + 0x1000000 = 0x53D369C$

Aufgabe 2

Für die Darstellung von 33.696.325 B brauchen wir mindestens 65.814 Blöcke à 512 B;

Allerdings müssen wir ja auch die Blocknummern speichern, wodurch je Block 4 B wegfallen.

Bei der Datenmenge benötigen wir also 518 weitere Blöcke, da nur je 508 B Nutzdaten zu Verfügung stehen.

Da ein weiterer Block für die Inode verbraucht wird, insgesamt also dann mindestens **66.333 Blöcke**.

Aufgabe 3

```
int f = open(``/home/ti2/archive/nikolaus.avi'', O_RDONLY);
```

open stellt die Verbindung zwischen der Datei `nikolaus.avi` und dem "file descriptor" `f` her.
Es ist also nach unserem Verständnis quasi ein Pointer auf die Datei.
Deshalb wird die Datei geöffnet, aber nicht geladen.

- Die Inode 0 (aus Block 2);
- der Datenblock der Verzeichnisdatei "/", dessen Nummer in Inode 0 steht;
- Inode 36 aus Datenblock 9;
- Datenblock der Verzeichnisdatei "home", extrahiert aus Inode 36;
- Inode 99 aus Datenblock 2000;
- Datenblock der Verzeichnisdatei "ti2", extrahiert aus Inode 99;
- Inote 206 aus Datenblock 3101;
- Datenblock der Verzeichnisdatei "archive", extrahiert aus Inode 206;
- Inode 12783 aus Datenblock 50.

```
int f = open(``/home/ti2/meta'', O_RDWR);
```

Es wird geladen:

- Die Inode 0 (aus Block 2);
- der Datenblock der Verzeichnisdatei "/", dessen Nummer in Inode 0 steht;
- Inode 36 aus Datenblock 9;
- Datenblock der Verzeichnisdatei "home", extrahiert aus Inode 36;
- Inode 99 aus Datenblock 2000.

```
lseek(f,-10000, SEEK_END);
```

`lseek` setzt den Pointer in `f` 10.000 Zeichen vor das Dateiende.

```
count = read(f, buf, 4096);
```

`count` liest 4096 B ab der, durch `lseek` gesetzten, Position und speichert diese in buf.
Dies sind die letzten 9 Blöcke ($\frac{4096}{508} = 8.06$, da 4B je Block durch die Blocknummer belegt sind) der Datei "nikolaus.avi", deren Adressen wir aus (einem der Indirektblöcke) der Inode 12783 (im Datenblock 50) auslesen müssen.

```
write(g, buf, count);
```

Schließlich schreiben wir die gelesenen 4096 B aus buf in g (also in die Datei meta).
Die Inode 99 aus Datenblock 2000 enthält die Informationen über den Datenblock der Datei "meta", welcher leer ist (abgesehen von der Datenblocknummer, die die ersten 4B belegt).
Wir schreiben hier in 9 Blöcke, die in den Direktblock der o.a. Inode passen.