Technische Informatik 2
Tutor: Marc Hildebrandt

WS 2016/17

C08

Timo Jasper (Inf, 3.FS.)
Thomas Tannous (Inf, 3.FS.)
Moritz Gerken (Inf, 3.FS.)

Übungsblatt 5

Lösungsvorschlag
Abgabe: 05.11.2016

1	2	3	4	5	$\sum$
	2	2.75			

Aufgabe 1

Mithilfe gdb ermittelter Bereich:
factorial: $0x000000000040054c$ bis $0x000000000040057c$-
main: $0x000000000040057d$ bis $0x00000000004005e5$.
Zusammen: $0x000000000040054c$ bis $0x00000000004005e5$.

1 b)

$-0x18(\%rbp) = 0x7fffffffdf42$
auf $-0x18(\%rbp)$ wird vier mal zugegriffen
$-0x8(\%rbp) = 0x7fffffffdf52$
auf $-0x8(\%rbp)$ wird vier mal zugegriffen

Auf andere Adressen wird nicht zugegriffen, oder sie haben nichts mit dem stack zu tun z.B.: $0x1$
Wir sind von der Funktion factorial selber ausgegeangen und gehen davon aus, dass keine von main generierten Daten im Stack liegen.

1 c)

$Paramter(RDI) : "factorial(\%lu) = \%lu\backslash n"$

0x40069c = 'f'
0x40069d = 'a'
0x40069e = 'c'
0x40069f = 't'
0x4006a0 = 'o'
0x4006a1 = 'r'
0x4006a2 = 'i'
0x4006a3 = 'a'
0x4006a4 = 'l'
0x4006a5 = '('
0x4006a6 = '%'
0x4006a7 = 'l'
0x4006a8 = 'u'

0x4006a9 = ')'
0x4006aa = ' '
0x4006ab = '=')
0x4006ac = ' ')
0x4006ad = '%'
0x4006ae = 'l'
0x4006af = 'u'
0x4006b0 = \n

1 d)

Pagenummer = virtuelle Adresse/Pagegröße (auf ganze Zahl abrunden)

4 KiB = 4096 Byte

für 1a):
0x000000000040054c = 4195660
4195660 / 4096 = 1024,33
Pagenumber = 1024

0x00000000004005e5 = 4195813
4195813 / 4096 = 1024,36
Pagenumber = 1024

für 1b):
0x7fffffffdf42 = 140737488346946
140737488346946 / 4096 = 34359738365,95
Pagenumber = 34359738365

0x7fffffffdf52 = 140737488346962
140737488346962 / 4096 = 34359738365,95
Pagenumber = 34359738365

für 1c):
0x40069c = 4195996
4195996 / 4096 = 1024,41
Pagenumber = 1024

0x4006b0 = 4196016
4196016 / 4096 = 1024,41
Pagenumber = 1024

Die Adressen aus 1a) & 1c) kommen zusammen in Pagenumber 1024.

Die zwei Adressen aus 1b) kommen in Pagenumber 34359738365.

Die Aufgabenstellung gibt keine Strategie vor, welche Frames ausgewählt werden, wir wählen daher für PageNr. 1024 den Frame 0 und für PageNr. 34359738365 den Frame 1, da diese nach der Pagetabelle für den aktiven Prozess, nicht belegt sind.

a) physische Adressen:
Frame 0: 0x1000000 bis 0x1000fff
0x1000000 mit 54c = 0x100054c
0x1000000 mit 54c = 0x10005e5

Der Adressraum aus Aufgabe 1a) entspricht dem physischen Adressraum 0x100054c bis 0x10005e5.

b) physische Adressen:
Frame 1 : 0x1001000 bis 0x1001fff
0x1001000 mit f42 = 0x1001f42
0x1001000 mit f52 = 0x0001f52

vituelle Adresse	physische Adresse
0x7fffffffdf42	0x1001f42
0x7fffffffdf52	0x1001f52

c) physische Adressen:
Frame 0: 0x1000000 bis 0x1000fff
0x1000000 mit 69c = 0x100069c
0x1000000 mit 6b0 = 0x10006b0

Der Adressraum aus Aufgabe 1c) entspricht dem physischen Adressraum 0x1001f42 bis 0x1001f52.

Aufgabe 2

Um die benötigte Anzahl der Blöcke für die Daten der Datei an sich zu berechnen, rechnen wir:

$$33696325B \div 512 \approx 65814$$

Das Inode selbst ist 128 Byte groß, braucht also noch selbst einen Block. (+1 Block).
Nun ist noch die Anzahl der Verweise eines Indirektblocks unbekannt. Um sie auszurechnen muss man die größe eines Blocks durch Anzahl der Bytes für eine Blocknummer teilen, so weiß man wie viel Verweise ein Indirektblock hat.

$$512B \div 4B = 128Verweise$$

Nun schauen wir, wie viele Indirektblöcke benutzt werden, indem von unseren errechneten Blöcken immer jene abziehen, welche durch direkte bzw. Indirekteblöcke dargestellt werden

können und die benötigten Indirektblöcke mitzählen.

$65814 - 10 = 65804$	kein extra Block	+2P
$65804 - 128 = 65676$	+1 Block für Indir.	
$65676 - (128 * 128) = 49292$	+129 (1 2x-Indir., 128 1x-Indir.)	
$49292 - (128 * 128 * 3) = 140$	+388 (1 3x-indir. 3 2x-Indir. 128 1x-Indir.)	
$140 - (128 * 2) = -116$	+3 (1 2x-Indir. 2 1x-Indir)	

Also lautet die gesamte Rechnung für alle benötigten Blöcke.

$$521 Indirektbloecke + 1 BlockInode + 65814 BloeckeDaten = 66336 Bloecke$$

Aufgabe 3 2.75/3P

`int f = open("/home/ti2/archive/nikolaus.avi", O_RDONLY);`
Der Dateiname / hat den Inode 0 und der zugehörige Block 2 ist bereits im Buffer-Cache.
Der Dateiname home hat den Inode 36 und lädt Block 9 in den HS.
Der Dateiname ti2 hat den Inode 99 und lädt Block 2000 in den HS.
Der Dateiname archive hat den Inode 206 und lädt Block 3101 in den HS.
Der Dateiname nikolaus.avi hat den Inode 12783 und lädt Block 50 in den HS.
`O_RDONLY` ist ein oflag, welcher für Open for reading and writing steht, die Datei wird also nicht verändert sondern nur gelesen.

`int g = open("/home/ti2/meta", O_RDWR);`
Der Dateiname / hat den Inode 0 und der zugehörige Block 2 ist bereits im Buffer-Cache.
Der Dateiname home hat den Inode 36 und lädt Block 9 in den HS.
Der Dateiname ti2 hat den Inode 99 und lädt Block 2000 in den HS.
Der Dateiname meta hat den Inode 99 und lädt Block 8521 in den HS.
`O_RDW` ist ein oflag, welcher für Open for reading only steht, die Datei kann also gelesen und verändert werden.

`lseek(f, -10000, SEEK_END);` ändert das offset in der file descriptor Tabelle für nikloas.avi zu 10000 Byte vor dem Ende der Datei, also: $12.582.912 - 10.000 = 12.572.912$ Bytes.
12 * 1024^2 - 1 - 10000 = 12572911
`count = read(f, buf, .4096);` ließt die Datei f, also nikolaus.avi, ab dem Byte in der file descriptor Tabelle die nächsten 4096 Bytes in buf ein und gibt, wenn dies erfolgreich war, die Anzahl der eingelesenen Bytes in count zurück.

`write(g, buf, count);` schreibt die ersten 4096 Bytes von buf in die leere Datei meta ein.

+1P: das Laden von Inodes wird berücksichtigt

+1P: der Zugriff auf Datenblöcke von Dateien wird berücksichtigt

+ 0.75P: hinreichend korrekte Beschreibung der Auswirkungen von lseek() und read()/write() (insbesondere Zugriff auf Datenblöcke, die über Indirektblöcke erreicht werden)

need more details description of functions, especially in how to reach Indirektblöcke