Technische Informatik 2 | WS 2016/17 | C02
Tutor: Marc/Bingbin

Rene Engel
Dennis Jacob
Jan Schoneberg

Übungsblatt 5

1	2	3	$\sum$
	2	2.5	

Lösungsvorschlag
Abgabe: 05.12.2016

Aufgabe 1

Die Aufgabe wurde auf dem x21-Rechner am 02.12.2016 bearbeitet.

Aufgabe 1a)

Zunächst haben wir mit *gdb* den virtuellen Adressraum ermittelt, die die Funktionen *main* und *factorial* einnehmen. Dazu wurde nachdem gdb mit `gdb factorial` gestartet wurde die Befehle `disassemble main` und `disassemble factorial` ausgeführt. Anhand des Assembler Codes und den entsprechenden Adressen im virt. Adressraum lassen sich folgende Bereiche erkennen:

Funktion	Startadresse	Endadresse
factorial	**0x40054c**	**0x40057c**
main	**0x40057d**	**0x4005e5**
gesamt	**0x40054c**	**0x5005e5**

Die gleichen Startadressen werden auch durch den Befehl `info functions` angezeigt.

Aufgabe 1b)

In Aufgabe 1b haben wir die virtuellen Adressen im Stacksegment untersucht, auf die in der Funktion *factorial* zugegriffen wird. Dazu haben wir zunächst ermittelt, auf welche Adressen zugegriffen wird, wenn der User Stackpointer nicht beachtet wird. Das Vorgehen ist wie folgt:

`break *0x40054c`: Breakpoint 1 am Beginn der Funktion *factorial*
`run 5`: ausführen des Programms; hält bei 0x40054c an
`display/a $ rsp` und `display/a $ sp`: Zeigt in jedem der folgenden Schritte den Inhalt der Register *sp* und *rsp* an (diese stimmen überein)
mehrfach `nexti`: ausführen der einzelnen Befehle bis zum Ende der Funktion *factorial*

Der User Stackpointer enthält während aller Aufrufe nur die Adressen 0x7fffffffe738 und 0x7fffffffe730. 0x7fffffffe738 ist dabei auch die oberste Adresse des Stackframes (der Stack wächst nach unten), was durch die Befehle `info stack, frame 0, info frame` ersichtlich wird. Somit wird in *factorial* nur auf 2 Adressen des Stacks zugegriffen. Dieses Ergebnis tritt auch auf, wenn das Programm mit anderen Werten für n, z.B. `run 10`, gestartet wird.

Wenn der User-Stackpointer beim Eintritt die Funktion den Wert 0x7fffffffdf60 hat, dann muss davon also noch 4_{dez} bzw. $0x4_{hex}$ subtrahiert werden um die zweite Adresse zu erhalten (der Stack wächst nach unten)(an jeder Adresse steht ein Byte, die restlichen 3 Bytes stehen an den folgenden Adressen)
$0x7fffffffdf60 - 0x4 = 0x7fffffffdf5c$

Da immer auf ein 32 Bit Datenwort zugegriffen wird, wird indirekt auch auf die 3 folgenden Adressen zugegriffen. (An jeder Stelle steht 1 Byte. 4 Bytes = 32 Bit)
$0x7fffffffdf5c - 0x3 = 0x7fffffffdf59$

Es wird also in der Funktion *factorial* auf die virtuellen Adressen **0x7fffffffdf60 bis 0x7fffffffdf59** im Stacksegment zugegriffen.

Aufgabe 1c)

Nun haben wir die Funktion *printf* in der Methode *main* untersucht. Dazu haben wir *gdb* zunächst neu gestartet (Befehl `quit`) um sicher zu gehen, dass alle bisherigen Änderungen unwirksam gemacht werden. Zunächst müssen wir analysieren, wie die Methode *printf* arbeitet, um festzustellen wo auf die Zeichenkette im Datensegment zugegriffen wird. Dazu gehen wir wie folgt vor:

`gdb factorial`: *gdb* starten
`disassemble main`: Assembler Code von *main* anzeigen
`break *0x4005da`: Breakpoint 1 an der virt. Adresse von *printf*
`run 5`: *factorial* ausführen; hält bei 0x4005da an
`disassemble printf`: Assembler Code von *printf* anzeigen
`break *0x7ffff7a9e190`: Breakpoint 2 am Beginn von *printf* (Alternativ `break *_ _ printf`
`until 2`: fortfahren bis zum Breakpoint 2 (0x7ffff7a9e190)
`info registers`

Es lässt sich erkennen, dass nicht alle Register belegt sind. Wenn man sich alle belegten Register mit:
`print/s $REGSITER`
anzeigen lässt, stellt man fest, dass im Register *rdi* an Adresse 0x40069c der String „factorial(%lu) = %lu\n“steht. Allerdings muss auch wieder die Länge ($21_{Dez} = 15_{hex}$ chars bzw. Bytes) des String berücksichtigt werden (das Datensegment wächst nach oben!). Es wird also auch auf die folgenden $20_{Dez} = 14_{Hex}$ Adressen zugegriffen:
0x40069c + 0x14 = 0x4006b0

Aufgabe 1d)

Der Adressbereich der physischen Adressen lässt sich wie folgt errechnen:

$$PA = PF\left(\left\lfloor \frac{VA}{PS} \right\rfloor\right) \cdot PS + PO_0 + PO$$

Wobei:
$PA \Rightarrow$ physische Adresse
$VA \Rightarrow$ virtuelle Adresse
$PS \Rightarrow$ Page-Size (4096 B = 0x1000)
$PO_0 \Rightarrow$ Offset des Page-Frame 0(0x1000000)
$PO \Rightarrow$ Offset der Page

physische Adressbereich der Aufgabe 1a)

virtuelle Adresse	Page	Offset	Page Nummer	Pageframe	physische Adresse
0x40054c	0x400	0x54c	1024	0x43d3	0x53d354c
0x40057c	0x400	0x57c	1024	0x43d3	0x53d357c
0x40057d	0x400	0x57d	1024	0x43d3	0x53d357d
0x4005e5	0x400	0x5e5	1024	0x43d3	0x53d35e5

physische Adressbereich der Aufgabe 1b)

virtuelle Adresse	Page	Offset	Page-Nummer	Pageframe	physische Adresse
0x7fffffffdf60	0x7fffffffd	0xf6	34359738365	?	?
0x7fffffffdf59	0x7fffffffd	0xf53	34359738365	?	?

Da diese Page-Nummer nicht im Hauptspeicher geladen ist, vermuten wir einen Fehler in Aufgabe 1b. Die Berechnung ergibt Sinn, wenn ich der virtuellen Adresse die erste 7 weg gelassen wird.

virtuelle Adresse	Page	Offset	Page-Nummer	Pageframe	physische Adresse
0xffffffdf60	0xffffffd	0xf60	4294967293	0x7d1	0x17d1f60
0xffffffdf59	0xffffffd	0xf59	4294967293	0x7d1	0x17d1f59

physische Adressbereich der Aufgabe 1c)

virtuelle Adresse	Page	Offset	Page-Nummer	Pageframe	physische Adresse
0x4006b0	0x400	0x6b	1024	0x43d3	0x53d306b

Aufgabe 2

> Wieviele Blöcke werden mindestens benötigt, um eine Datei der Größe 33696325 B auf der Platte zu speichern?

Die Anzahl der Blöcke, die benötigt werden, um nur die Datei zu speichern betragt $\frac{33696325\ B}{512\ B} = 65814.$ Hierbei ist zu beachten, dass immer aufgerundet werden muss, da ein Block belegt ist, auch wenn nur ein Bit wirklich Nutzdaten enthält. +0.5P

Verbleibene Datenblöcke	Name	Fassungsvermögen	Overhead
65814	Inode	10	1
65804	einfach Indirektblock	128	1
65676	zweifach Indirekblock	128^2	$1 + 128$
49292	dreifach Indirektblock	128^3	$1 + 128^2$

Für die restlichen 49292 Datenblöcke muss nicht der gesamte dreifach Indirektblock verwendet werden. Daher hier eine kleine Nebenrechnung:
$\left\lceil \frac{49292}{128} \right\rceil = 386 \Rightarrow$ Anzahl der einfach Indirektblöcken "unterhalb"des dreifach Indirektblocks
$\left\lceil \frac{386}{128} \right\rceil = 4 \Rightarrow$ Anzahl der zweifach Indirektblöcken "unterhalb"des dreifach Indirektblocks
Somit ergibt sich für den Overhead:
$1 + 1 + 1 + 128 + 1 + 4 + 386 = 522$ +1P
Und somit für die Gesamtblockanzahl:
$522 + 65814 = 66336$ +0.5P

Aufgabe 3

> Verfolgt die Abarbeitung der Systemaufrufe und beschreibt, welche Inodes und Datenblöcke von der Festplatte in den Hauptspeicher geladen werden müssen.

1. `int f = open("/home/ti2/archive/nikolaus.avi", O_RDONLY);`
   Mit diesem Befehl wird die Datei *nikolaus.avi* mit *Read-Only*-Berechtgung (nur lesen) geöffnet. Dabei ergibt sich folgende Aufrufkette der **Inodes** und **Datenblöcke**:
   1.1. Der *Inode* des *root*-Verzeichnisses `/` wird beim Booten ermittelt. Von diesem aus wird der Datenblock 2 des *root*-Verzeichnisses geladen. In diesem ist die Pfadkomponente `home` zu finden die auf den *Inode* 36 verweist. Der Datenblock 9 wird geladen. +1P
   1.2. Im Datenblock 9 verweist die Pfadkomponente `ti2` auf den *Inode* 99. Datenblock 2000 wird geladen.
   1.3. Im Datenblock 2000 verweist die Pfadkomponente `archive` auf den *Inode* 206. Datenblock 3101 wird geladen.
   1.4. Im Datenblock 3101 verweist der Dateiname `nikolaus.avi` auf der *Inode* 12783. Der Datenblock 50 wird geladen.
   1.5. Im Datenblock 50 wird auf den Datenblock der Datei `nikolaus.avi` verwiesen.
   1.6. Der Datenblock der `nikolaus.avi` wird geladen.

1.7. `open()` gibt einen `File Descriptor` für die `nikolaus.avi` zurück.

2. `int g = open("/home/ti2/meta", O_RDWR);`
   Mit diesem Befehl wird das Verzeichnis `meta` geöffnet. Die Schritte 1 und 2 des 1. Befehlsaufrufs wiederholen sich.

+1P

   2.3. Im Datenblock 2000 verweist die Pfadkomponente `meta` auf den *Inode* 112. Datenblock 8521 wird geladen.

   2.4. `open()` gibt einen `File Descriptor` für das Verzeichnis `meta` zurück.

3. `lseek(f, -10000, SEEK_END);`
   Verschieben des Positionszeigers von f auf 12 * 1024^2 - 1 - 10000 = 12572911.....
   Mit diesem Befehl wird die aktuelle Position im Datenblock der Datei `nikolaus.avi` auf die mittels `File Descriptor`-Wert `f` verwiesen wird um -10000 vom Dateiende ausgehend verschoben (Ab dieser Position folgen dann 10000 Bytes bis zum Dateiende).

4. `count = read(f, buf, 4096);`
   Zugriff auf Inode von 'nikolaus.avi' zum Ermitteln des Datenblocks 24556 (floor(12572911/512))......
   Mit diesem Befehl werden 4096 Bytes aus dem Datenblock der Datei `nikolaus.avi` auf die mittels `File Descriptor`-Wert `f` verwiesen wird in den Buffer `buf` geladen. Es gilt hierbei zu beachten, dass das Lesen erst bei den letzten 10000 Bytes des Datenblockes beginnt (Nach dem letzten gelesenen Byte folgen noch 5904 Bytes bis zum Datenblockende).

5. `write(g, buf, count);`
   Mit diesem Befehl werden die Daten des Buffers `buf` in das Verzeichnis `meta` auf das mittels `File Descriptor`-Wert `g` verwiesen wird geschrieben. Dabei ist zu beachten, dass `count` die tatsächliche Anzahl der in den Buffer gelesenen Dateien angibt. Diese Variable wird hier verwendet um anzugeben, wie viele Bytes geschrieben werden sollen. Durch die Ausführung des Befehls wird dann ein freier Datenblock mit den zu schreibenden Bytes belegt. Es erfolgt ein neuer Eintrag in der *Inode*-Tabelle der auf diesen Datenblock verweist.

+0.5

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