assembly
read and write

TECHNISCHE UNIVERSITÄT DARMSTADT

Rechnerorganisation
Sommersemester 2020

Jan Braun
Simon Gröger

Inhaltsverzeichnis

printf (write) 1

scanf (read) 2

Anwendungsbeispiele 3

call parameter (read) 3

printf (write)

Um primitiv Text auf die Konsole/Terminal (stdout) ausgeben zu können, gibt es Funktionen wie *printf*. *printf* kann zusätzlich zur primitiven Textausgabe auch variable verschiedene Zahlenwerte oder auch weitere Strings in den Hauptstring einsetzen und als Ganzes ausgeben. Daher hat die Signatur dieser C-funktion auch keine feste Anzahl an parameter (mindestens jedoch einen – den Hauptstring). Es ist also (beinahe) beliebig möglich weitere Parameter zu ergänzen. Dies hat außerdem den Vorteil, dass die Zahlen, die eingesetzt werden sollen, noch berechnet werden können. Durch Verzweigungen im Programm kann unter anderem entschieden werden, welche Version (spezifischer Wert) einer Variable eingesetzt.

Was muss man dafür tun?
Dafür gibt es bestimmte Platzhalter, die man im seinem String verwenden kann, zu denen dann in passender Reihenfolge ein Parameter mit diesem Typ folgen muss. Falls Man also %s verwendet und an dieser Stelle eine Zahl steht, dann würde diese als Addresse für den Beginn des Strings gewertet werden. Man erhält also möglicher Weise einen *Segmentation Fault* oder einen sehr unerwarten String, wenn man die Reihenfolge nicht einhält oder den falschen Typ zum Platzhalter. In C wird man vielleicht noch durch Compiler-Warnungen darauf noch hingewiesen, weil C selbst noch gewisse Typen kennt. In Assembler legst du komplett selbst fest, wie deine Bitfolgen ausgewertet bzw. verarbeitet werden sollen.

Platzhalter	Typ
%d	Integer / Dezimal Zahl
%f	Kommazahl
%s	String
%p	Pointer (Addresse)
%o	Octal Zahl
%x	Hexadezimal Zahl
%c	Character (ASCII)

Solange man **nicht selbst** „printf“ als **Label** verwendet hat, wird GCC für einen entweder in einem der anderen Inputfiles oder in der Standardbibliothek nach diesem suchen. Wenn man nun mit entweder „gcc -o printf printf.c“ (Listing 1) oder „gcc -o printf printf.s“ (Listing 2) sein Programm compiliert hat, sollte jeweils mit dem ausführen von „./printf“, folgender output entstehen: „11 Äpfel kosten am Marktplatz 22 Euro.“

Listing 1: printf.c

```c
#include <stdio.h>

char* mystring = "%d %s kosten %s %d %s.\n";
int count = 11;
char* product = "Äpfel";
char* location = "am Marktplatz";
int price = 22;
char* currency = "Euro";

int main() {

    printf(mystring,
           count,
           product,
           location,
           price,
           currency);

    return 0;
}
```

Assembler/Architektur	Bedeutung in der Hochsprache
R0	Adresse des Hauptstringes
R1-R3	die ersten 3 einzusetzenden Werte
Stack	5+. Paramter

Der Stack wächst von hohen Addressen in Richtung niedrigeren Addressen. Die Parameter werden jedoch der Reihe nach ab der niedrigsten (SP = R13) zu den höheren Addressen auf den Stack gelegt. Wichtig ist nur den Stack vorher passend zu erweitern (um das 4fache der Anzahl der Parameter), damit man nicht alte Werte überschreibt. Danach natürlich wieder zurücksetzen. Mehr dazu behandeln wir später noch hier in RO, kommt aber auch ausführlich in EiCb (Einführung in den Compilerbau) dran oder Sie suchen nach der Calling Convention für ARM.

Listing 2: printf.s

```
.data
mystring: .asciz "%d %s kosten %s %d %s.\n"
count: .word 11 // or 0xB or 0b1011
product: .asciz "Äpfel"
location: .asciz "am Marktplatz"
price: .word 22 // or 0x16 or 0b10110
currency: .asciz "Euro"

.text
.global main
main:
    push {lr} // save return address

    ldr r0, =mystring   // mystring
    ldr r1, =count
    ldr r1, [r1]        // %d = count
    ldr r2, =product    // %s = product
    ldr r3, =location   // %s = location
    ldr r4, =price
    ldr r4, [r4]
    ldr r5, =currency

    sub sp, #8          // 2 words
    str r4, [sp]        // %d = price
    str r5, [sp, #4]    // %s = currency

    bl printf   // calls printf
    add sp, #8  // 2 words

    mov r0, #0  // return value
    pop {lr}
    bx lr       // return or exit
```

Rückgabewert
Wenn *printf* returned, gibt es einem die **Anzahl an geschriebenen Zeichen** zurück. Falls ein Fehler aufgetreten ist, ist der Wert negativ. In unserem Fall wären das dann 41. In Assembler kann man das wie immer überprüfen in dem man **R0** nach *printf* (nach Zeile 27 – s. Listing 2) ausliest.

scanf (read)

Wenn wir nun interaktiv auf Nutzereingaben reagieren wollen. Natürlich wieder nur vereinfacht über die Konsole/Terminal (stdin) einlesen, gibt es Funktionen wie *scanf*, welche die selben Platzhalter wie *printf* verwenden können, jedoch nun die Eingabe umgewandelt in den passend Typ in eine vorher festgelegte Variable schreibt. Für die Eingabe „123 456“ würde scanf("%d %d", num1, num2) jeweils „123“ in **num1** und „456“ in **num2** speichern. *scanf* liest immer nur bis zum ersten Leerzeichen ein, bedeutet also, dass bei folgendem Code: scanf("%s", s) und folgender Eingabe: „Herzlich Willkommen“ nach Ausführung „Herzlich“ in s steht. Möchte man mehrere Wörter getrennt einlesen, kann man das bei bekannter Anzahl an Wörtern über scanf("%s %s %s ...", s1, s2, s3,...) realisieren, sodass das erste Wort in s1 ist, das zweite in s2, usw..

Rückgabewert: die Anzahl an erfolgreich gelesenen Inputs oder EOF (GCC hat das standardweise als -1 definiert), falls ein Fehler auftritt.

Zu Beachten ist:
In C muss man die Länge eines String dadurch festlegen, dass man eine feste Menge an Speicher für das char-Array reserviert. Wenn man nun dem Nutzer freie Wahl lässt kann dies sehr schnell dazu führen, dass man den Stack beim Überschreiten der Arraygrenze überschreibt und damit potenziell **die Rücksprung addresse manipuliert** wird (**Stack buffer overflow** – Sollte euch hoffentlich in Computersystemsicherheit oder Betriebssysteme ausführlich nochmal gezeigt werden). Einschränken kann man das, in dem man zwischen das % und dem Platzhalter die Anzahl an zulesenden Zeichen definiert (z.B: „%5s“ um 5 Zeichen für einen String ein-

zulesen). Negativer Nebeneffekt alle weiteren Zeichen bleiben erstmal in der Inputqueue stehen. D.h. wenn das nächste Mal eine Funktion vom Eingabefile (stdin) liest, wird bei uns ab dem 6. Eingabezeichen weiter gelesen. Da am Ende unser Zeilenumbruch duch das Enter folgt, ist diese Eingabe meist schon ausgewertet, bevor man sie eintippen durfte. Hierfür muss man selbst Wege finden die Inputqueue zu bereinigen.

Listing 3: scanf.c

```
1  #include <stdio.h>
2
3  char name[20];
4  char* welcome = "Willkommen, %s!\n";
5
6  int main() {
7
8      printf("Nenne deinen Namen: ");
9      scanf("%20s",
10           name);
11
12     printf(welcome,
13            name);
14
15     return 0;
16 }
```

Durch Verwendung der Register ist eigentlich genauso wie bei *printf*. Man muss jedoch jetzt immer den Pointer (Addresse) der Variable verwenden. Deswegen wurde in der 3. Zeile (rechts) mit „.skip“ 20 Bytes reseviert, um diese für den String bereitzustellen.

Listing 4: scanf.s

```
1  .data
2  // ASCI: 1 Byte = 1 Zeichen
3  name: .skip 20 // 20 Byte langes Array
4  welcome: .asciz "Willkommen, %s!\n"
5  message: .asciz "Nenne deinen Namen: "
6  format: .asciz "%20s"
7
8  .text
9  .global main
10 main:
11     push {lr} // bl will change our return address
12
13     ldr r0, =message     // "Nenne ..."
14     bl printf            // calls printf
15
16     ldr r0, =format      // "%20s"
17     ldr r1, =name        // %s = name
18     bl scanf             // calls scanf
19
20     ldr r0, =welcome     // welcome
21     ldr r1, =name        // %s = name
22     bl printf            // calls printf
23
24     mov r0, #0  // return value
25
26     pop {lr}
27     bx lr        // return or exit
```

Anwendungsbeispiele

printf

Variablenbelegung	Code	Ausgabe

int x=5;
int y=2;
char* s1 = “Eine Birne kostet %d Euro”;
int z = 16;
	printf(“Dies ist eine Ausgabe”);
	printf(“Ein Apfel kostet %d Euro”, x);
	printf(s1, y);
	printf(“%x”, z);
		„Dies ist eine Ausgabe“
		„Ein Apfel kostet 5 Euro“
		„Eine Birne kostet 2 Euro“
		„10“

scanf

Code	Eingabe	Variablenbelegung
scanf(”%s”, s);	„Hallo“	char* s = ”Hallo”;
scanf(”%s”, s);	„Hallo Tom“	char* s = ”Hallo”;
scanf(”%s %s”, s1, s2);	„Hallo Tom“	char* s1 = ”Hallo”; char* s2 = ”Tom”;
scanf(”%d”, &x);	„14“	int x = 14;

call parameter (read)

Reden wir zum Schluss nochmal darüber, wie man Parameter beim Aufruf des Programmes übergibt. Was mittlerweile klar sein sollte, dass zumindest bei GCC jeder C-Funktion für die **Parameter** zuerst in den ersten 4 Registern **(R0-R3)** genutzt werden und danach der **Stack** verwendet wird. Der **Rückgabewert** (falls vorhanden – nicht void) wird in **R0** gespeichert.
Bei der main-Funktion gibt es die Möglichkeit auch nach den Parameter zu fragen mit denen das Programm aufgerufen wurde. In dem Fall gibt man noch *argc* und *argv* an.

argc (R0) speichert die **Anzahl an Parametern**. Es ist zu beachten, dass der Programmname auch dazuzählt. Falls man keine Parameter übergibt, ist dies in dem Fall trotzdem 1.
argv (R1) ist die **Startaddresse** des Array, in dem die **Parameter als Strings** gespeichert werden. Das erste Element ist in dem Fall auch der Name, mit dem das Programm aufgerufen wurde. Dies ist in dem Fall ein **Doppelpointer**, da C keine Strings kennt und dies char-Arrays sind. Die einzelnen Arrayelemente sind also auch wieder Addressen von den einzelnen Strings der Parameter.
Jeder Parameter ist durch ein Leerzeichen getrennt. Sollte man Anführungszeichen verwenden, wird der Block von Anführungszeichen bis zum Nächsten nicht durch Leerzeichen getrennt.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  /* check parameter count */
  // programname + 2 parameters (factors)
  if(argc != 3) {
    printf("expected 2 parameter\n");
    printf("Syntax: %s [name] [number]\n",
           argv[0]);
    // ERROR CODE: 1
    // ("wrong parameter count")
    return(1);
  }

  /* load parameter */
  char* name = argv[1];
  int   num  = atoi(argv[2]);

  printf("Willkommen, %s!\n",
         name);
  printf("Das Quadrat von %d ist %d.\n",
         num,
         num*num);

  return(0);  // ERROR CODE: 0 ("no error")
}
```

Nach dem Compilieren könnt ihr zum Ausprobieren das Programm einfach mal mit ./main "orname Nachname" 7 starten und solltet folgende Antwort erhalten:
Willkommen, Vorname Nachname!
Das Quadrat von 7 ist 49.

Bei falschem Setzen der Anführungszeichen, erhaltet ihr entweder die Fehlermeldung zu der falschen Parameteranzahl oder es wird das Quadrat von 0 berechnet. Wenn keine Zahl erkannt werden kann, gibt „*atoi*“ gibt nämlich einfach 0 zurück.

Selbst wenn ihr das Programm umbennen solltet oder es über einen symbolischen Link aufruft, wird euch bei der Fehlermeldung immer das angezeigt werden, was ihr zum Aufrufen eingeben habt. „*argv*“ spiegelt im Prinzip den kompletten String, mit dem das Programm aufgerufen wurde, wider. Daher steht auch wie oben schon erwähnt im ersten Element der String des Programmnamens sozusagen.

```
.data
err_1_p1: .asciz "expected 2 parameter\n"
err_1_p2: .asciz "Syntax: %s [name] [number]\n"
welcome: .asciz "Willkommen, %s!\n"
quad: .asciz "Das Quadrat von %d ist %d.\n"

.text
.global main
main:
  push {lr}

  /* check parameter count */
  // programname + 2 parameters (factors)
  cmp r0, #3
  beq START
    push {r1}         // save argv

    ldr r0, =err_1_p1 // err message first line
    bl printf

    ldr r0, =err_1_p2 // err message second line
    pop {r1}        // reload argv
    ldr r1, [r1]    // load argv[0] = programname
    bl printf

    // ERROR CODE: 1
    // ("wrong parameter count")
    mov r0, #1
    b END

START:  /* load parameter */
  ldr r5, [r1, #4]  // get $1 = argv[1]
  ldr r6, [r1, #8]  // get $2 = argv[2]
  push {r6} // save $1 during num = atoi($2)

  mov r1, r5
  ldr r0, =welcome
  bl printf         // call printf

  pop {r0}          // reload $2 = argv[2]
  bl atoi           // string (asci) to int
  mov r1, r0        // r1 = num
  mul r2, r1, r1    // num * num
  ldr r0, =quad
  bl printf

  mov r0, #0  // ERROR CODE: 0 ("no error")
END:    /* end of program */
  pop {lr}
  bx lr       // return or exit
```