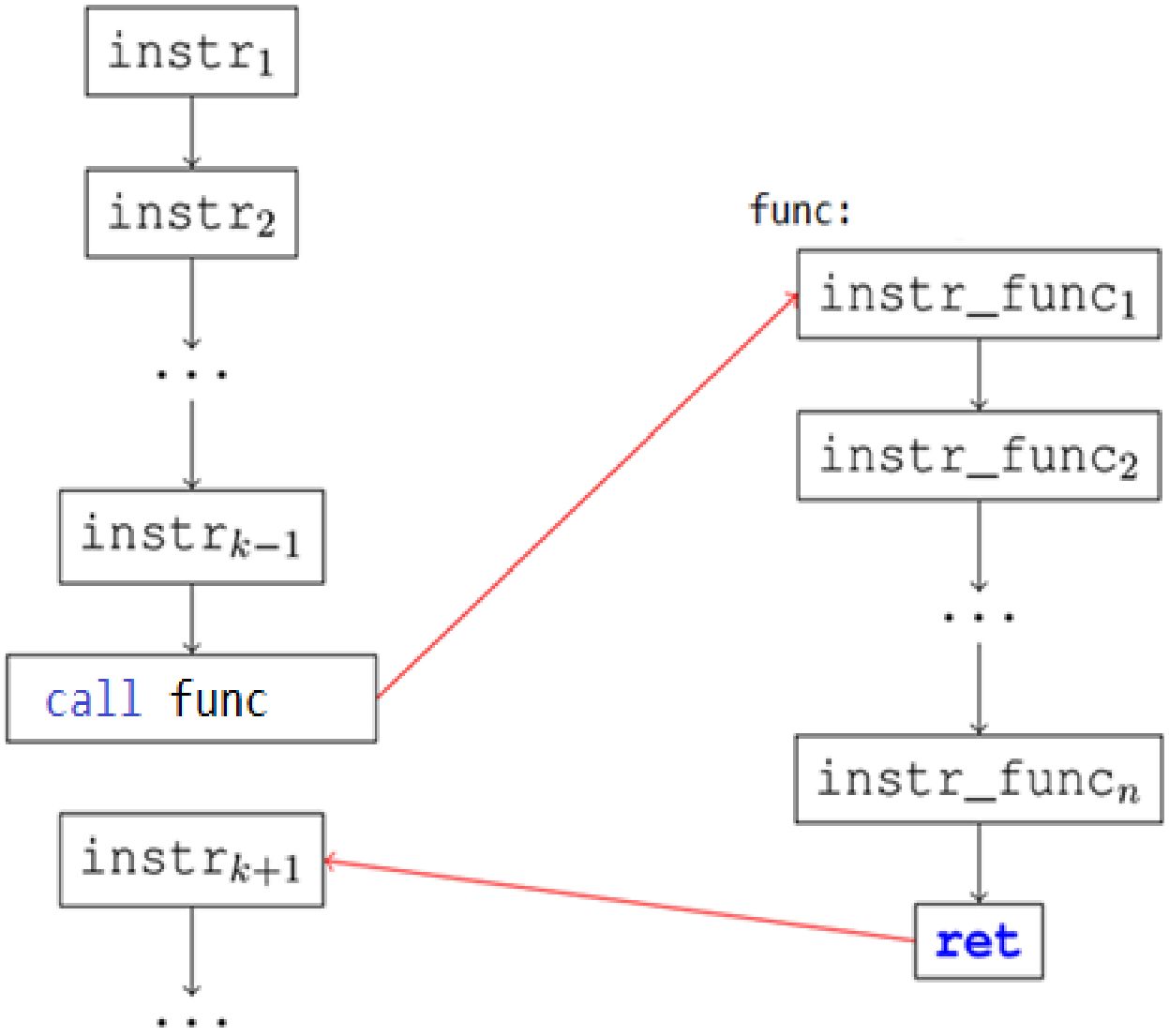
Seminar 5

Rechnerarchitektur

Aufruf von C/C++ - Funktionen

- Beim Aufruf der Funktion wird die Rücksprungadresse auf dem Stack abgelegt
- Bei Speichermodellen mit maximal 64 kB Code ist dies IP (Instruction Pointer Register), bei Speichermodellen mit mehr Code ist es CS:IP
- Für den Aufruf benutzt man **`CALL <function_name>`**
- **CALL** speichert die Adresse des nächsten Befehls nach CALL, die Rücksprungadresse, auf dem Stack und überträgt die Kontrolle an die Funktion (springt zu der angegebenen Startadresse des gerufenen Unterprogramms)
- Am Ende des Unterprogramms/Funktions benutzt man: **RET**
- **RET** führt den Rücksprung aus, d.h. RET holt die Rückgsprungadresse vom Stack und lädt sie in das IP, so dass dort fortgesetzt wird


instr$_1$
instr$_2$
...
instr$_{k-1}$
call func
instr$_{k+1}$
...
func:
instr_func$_1$
instr_func$_2$
...
instr_func$_n$
ret

Aufruf von C/C++ - Funktionen

- Der Assemblercode muss vor dem Aufruf die Parameter auf dem Stack hinterlegen
- Der C-Compiler legt die Programme so an, dass Funktionsergebnisse möglichst im Register EAX bzw. Teilen davon zurückgegeben werden

Funktionstyp	**Rückgaberegister**
char	AL
short	AX
int, long	EAX
Strukturen bis zu 64 Bit	EDX:EAX
Strukturen größer als 64 Bit	Zeiger auf Speicherbereiche

Aufruf von C/C++ - Funktionen

- Eine Aufrufkonvention bestimmt, wie die Parameterübergabe an Funktionen gestaltet wird:
  - Die Reihenfolge der Parameter
  - Methode der Übergabe der Parameter (auf dem Stack, in den Registern)
  - Welche Registern müssen in der Funktion/Unteprogramm nicht geändert werden (non-volatile Registern oder Registern gespeichert für die aufrufende Funktion)
  - Wie werden die Aufgaben zwischen Aufrufende Funktion und aufgerufte Funktion aufgeteilt (Vorbereitung des Stacks, Wiederherstellung des Stacks)

Aufruf von C/C++ - Funktionen

- Man benutzt man die CDECL Aufrufkonvention (C declaration):
  - Stack-Parameterübergabe
  - Reihenfolge der Parameter von rechts nach links (der letztgenannte Parameter wird zuerst abgelegt)
  - Register EAX, ECX und EDX werden in der Funktion benutzt, also die Werte können überschrieben werden (wenn man diese Werte braucht speichert man die Werte vor dem Aufruf in Variablen oder auf dem Stack)
  - Aurufende Funktion räumt Stack auf (die Funktion löscht die Parameter vom Stack nicht, das muss die aufrufende Funktion tun)

Input/Output Funktionen

- Drucke auf dem Bildschirm aus & lese eine Tastatureingabe
- Öffne eine Datei:
  - Eine existierende Datei
  - Erstelle eine neue Datei und öffne diese
- Lese Daten aus einer Datei & schreibe in eine Datei
- Schließe eine Datei

Drucke auf dem Bildschirm **int printf(const char * format, variable_1, constant_2, ...);**	
printf("Seminar 6 ASC");	segment data use32 class=data text db "Seminar 6 ASC", 0 segment code use32 class=code push dword text call [printf] add esp, 4 * 1
printf("Seminar %u ASC ", 6);	segment data use32 class=data format db "Seminar %u ASC", 0 segment code use32 class=code push dword 6 push dword format call [printf] add esp, 4 * 2
printf("It's %s and outside are %d degrees ","Monday", -2);	segment data use32 class=data format db " It's %s and outside are %d degrees", 0 day db "Monday", 0 degree dd -2 segment code use32 class=code push dword [degree] push dword day push dword format call [printf] add esp, 4 * 3

Lese von der Tastatur (reading a keyboard input) **int scanf(const char * format, adress_variable_1, ...);**	
scanf("%d", &n);	segment data use32 class=data n dd 0 format db "%d",0 segment code use32 class=code push dword n push dword format call [scanf] add esp, 4 * 2
scanf("%s%d", &day,&degrees);	segment data use32 class=data day times 10 db 0 degrees dd 0 format db "%s%d", 0 segment code use32 class=code push dword degrees push dword day push dword format call [scanf] add esp, 4 * 3

Allgemeine Formatierungscodes für printf()

Code	Base/Type	Description
%c	character	Zeigt ein Charakter als Zeichen.
%d	10	Konvertiert eine Zahl und zeigt sie als Dezimalzahl.
%s	String	Zeigt ein String als String.
%x	16	Konvertiert eine Zahl und zeigt sie als Hexadezimalzahl.
%%	Percent symbol	Zeigt ein Prozentsymbol

Bemerkungen

- Printf() hat **keine festgelegte Anzahl von Parameter**. Es kann ein einziges Parameter haben, aber auch viel mehrere.
- Für Daten von 32-Bit oder 64-Bit speichert man die Daten selber auf dem Stack
- Für Strukturen größer als 64 Bit, d.h. für **Strings und Arrays**, speichert man auf dem Stack ein Zeiger auf Speicherbereiche
- Alle **String** Parameter müssen **den Endcharakter 0** haben (das ist die einzige Möglichkeit wie die glibc Funktionen wissen können, wo die Strings enden)
- Wenn man **ECX als Zähler** benutzen (z.B. für einen Loop) und inzwischen einen printf() aufruft, dann wird **der Wert in ECX verloren gehen**! Man muss ECX vor dem Auruf speichern (z.B. auf dem Stack) und nach der Ausführung der Funktion kann man den Wert wiederherstellen

Öffne eine Datei FILE * fopen(const char* file_name, const char * access_mode)		
Modus	**Bedeutung**	**Beschreibung**
r	read	• Öffne Datei für Lesen • Die Datei muss existieren.
w	write	• Öffne eine Datei für Lesen. • Falls die Datei nicht existiert, dann wird diese erstellt. • Falls die Datei existiert, dann wird der Inhalt der Datei überschrieben.
a	append	• Öffne eine Datei für Schreiben. • Falls die Datei nicht existiert, dann wird diese erstellt. • Falls die Datei existiert, dann wird der Inhalt der Datei nicht überschrieben, sondern man schreibt am Ende der Datei.
r+	read und write für eine existierende Datei	• Öffne Datei für Lesen und Schreiben. • Die Datei muss existieren.

Öffne eine Datei FILE * fopen(const char* file_name, const char * access_mode)		
Modus	**Bedeutung**	**Beschreibung**
w+	read und write	• Öffne eine Datei für Lesen und Schreiben. • Falls die Datei nicht existiert, dann wird diese erstellt. • Falls die Datei existiert, dann wird der Inhalt der Datei überschrieben.
a+	read und append	• Öffne eine Datei für Lesen und Schreiben. • Falls die Datei nicht existiert, dann wird diese erstellt. • Falls die Datei existiert, dann wird der Inhalt der Datei nicht überschrieben, sondern man schreibt am Ende der Datei.
Ergebnis		
Falls die Datei erfolgreich geöffnet wurde, dann enthält EAX den Dateideskriptor (ein Identifikator), der weiter benutzt wird bei der Arbeit mit der Datei (lesen und schreiben). **Falls ein Fehler auftritt, dann wird EAX auf 0 gesetzt.**		

Lese aus einer Datei **int fread(void * str, int size, int count, FILE * stream)**
• **das erste Parameter ist der String, wo die gelesenen Bytes gespeichert werden** • **das zweite Parameter enthält die Größe der Elemente, die aus der Datei gelesen werden** • **das dritte Parameter enthält die maximale Anzahl der Elemente, die gelesen werden** • **das letzte Parameter ist der Dateideskriptor**
Ergebnis
EAX wird die Anzahl der gelesenen Elemente enthalten. Falls diese Zahl kleiner als *count* ist, dann heißt das entweder, dass ein Fehler aufgetreten ist, oder dass man bis am Ende der Datei gelesen hat.

Schreibe in eine Datei int fprintf(FILE * stream, const char * format, <variable_1>, <constant_2>, <...>)
Ergebnis
Falls es eine Fehlermeldung gibt, dann enthält EAX einen Wert < 0

Schließen einer Datei int fclose(FILE * descriptor)

Beispiel CREATE + CLOSE	
segment data use32 class=data	**segment code use32 class=code**

file_name db "ana.txt", 0
access_mode db "w", 0
file_descriptor dd -1
	push dword access_mode
	push dword file_name
	call [fopen]
	add esp, 4*2
	
	mov [file_descriptor], eax
	
	cmp eax, 0
	je final
	
	push dword [file_descriptor]
	call [fclose]
	add esp, 4

Beispiel CREATE + WRITE	
segment data use32 class=data	**segment code use32 class=code**

file_name db "ana.txt", 0
access_mode db "w", 0
text db "Some text.", 0
file_descriptor dd -1
	push dword access_mode
	push dword file_name
	call [fopen]
	add esp, 4*2
	
	mov [file_descriptor], eax
	
	cmp eax, 0
	je final
	
	push dword text
	push dword [file_descriptor]
	call [fprintf]
	add esp, 4*2
	
	push dword [file_descriptor]
	call [fclose]
	add esp, 4

Beispiel CREATE + APPEND	
segment data use32 class=data	**segment code use32 class=code**

file_name db "ana.txt", 0
access_mode db "a", 0
text db "Some text.", 0
file_descriptor dd -1
	push dword access_mode
	push dword file_name
	call [fopen]
	add esp, 4*2
	
	mov [file_descriptor], eax
	
	cmp eax, 0
	je final
	
	push dword text
	push dword [file_descriptor]
	call [fprintf]
	add esp, 4*2
	
	push dword [file_descriptor]
	call [fclose]
	add esp, 4

Beispiel READ

segment data use32 class=data

```
file_name db "ana.txt", 0
access_mode db "r", 0
file_descriptor dd -1
len equ 100
text times len db 0
```

segment code use32 class=code

```
push dword access_mode
push dword file_name
call [fopen]
add esp, 4*2

mov [file_descriptor], eax

cmp eax, 0
je final

push dword [file_descriptor]
push dword len
push dword 1
push dword text
call [fread]
add esp, 4*4

push dword [file_descriptor]
call [fclose]
add esp, 4
```

Beispiel READ + WRITE

segment data use32 class=data

```
file_name db "ana.txt", 0
access_mode db "r", 0
len equ 100
text times (len+1) db 0
file_descriptor dd -1
format db "We have read %d
characters from the file. The
text is: %s", 0
```

segment code use32 class=code

```
push dword access_mode
push dword file_name
call [fopen]
add esp, 4*2
mov [file_descriptor], eax

cmp eax, 0
je final

push dword [file_descriptor]
push dword len
push dword 1
push dword text
call [fread]
add esp, 4*4
```

```
push dword text
push dword EAX
push dword format
call [printf]
add esp, 4*3


push dword [file_descriptor]
call [fclose]
add esp, 4
```

Beispiel READ + PRINT FULL

segment data use32 class=data

```
file_name db "input.txt", 0
access_mode db "r", 0
file_descriptor dd -1
nr_char_read dd 0
len equ 100
buffer times (len+1) db 0
format db "We have read %d
characters from the file. The
text is: %s", 10,13,0
```

segment code use32 class=code

```
push dword access_mode
push dword file_name
call [fopen]
add esp, 4*2

cmp eax, 0
je final

mov [file_descriptor], eax

repeat:
        push dword [file_descriptor]
        push dword len
        push dword 1
        push dword buffer
        call [fread]
        add esp, 4*4

cmp eax,0
je cleanup

mov [nr_char_read], eax
push dword buffer
push dword EAX
push dword format
call [printf]
add esp, 4*3

jmp repeat

cleanup:
    push dword [file_descriptor]
    call [fclose]
    add esp, 4
```

Aufruf von C/C++ - Funktionen in Assembly Code

- Die benutzte C-Funktionen werden als extern deklariert:

  z.B. `extern printf , scanf`

- Die benutzte Funktionen werden aus der Bibliothek msvcrt.dll importiert:

  z.B. `import printf msvcrt.dll`

  `import scanf msvcrt.dll`

- `Liste anderer C-Funktionen die man aufrufen kann:`

https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/crt-alphabetical-function-reference?view=msvc-160&viewFallbackFrom=vs-2017

Aufgabe 1

Lese eine Hexadezimale Zahl n, die auf ein Wort gespeichert werden kann. Öffne eine Datei *in.txt*, welche genau 16 Bytes enthält und drucke auf dem Bildschirm die Bytes, welche einem Bit 1 in der binäre Darstellung der Zahl n entsprechen.

Beispiel:

- gelesene Zahl: $F2A1_h = 1111001010100001_b$, d.h. die Zahl *n* hat Bits mit dem Wert 1 auf den Positionen: 0, 5, 7, 9, 12, 13, 14, 15
- Die Datei *in.txt* enthält folgende Bytes: 0123456789abcdef
- Auf dem Bildschirm sollte man folgende Bytes ausdrucken: 0, 5, 7, 9, c, d, e, f

Aufgabe 2

Sei *s* ein String, das unterschiedliche Charakter enthält.

Lese den Namen einer Input Datei von der Tastatur.

Erstelle ein String *d* von Bytes, der für jedes Zeichen aus dem String *s* die entsprechende Anzahl der Auftritte in der gegebenen Datei enthält.

Erstelle eine Datei *output.txt* und schreibe das Ergebnis in der folgender Form:

Charakter 1 – Anzahl 1

Charakter 2 – Anzahl 2 (jedes Paar auf eine neue Zeile)

Aufgabe 2 - Beispiel

- s db ‘13579abcd’
- input.txt : aba13a124
- Der String d enthält:
  - auf der ersten Position die Anzahl der Auftritte der Zeichen ‘1’ in der Datei, also 2
  - auf der zweiten Position die Anzahl der Auftritte der Zeichen ‘3’ in der Datei, also 1
  - auf der dritten Position die Anzahl der Auftritte der Zeichen ‘5’ in der Datei, also 0 ....
- d: 2, 1, 0, 0, 0, 3, 1, 0, 0
- Die Datei output.txt enthält folgende Daten:

  1 – 2

  3 – 1

  5 – 0

  7 – 0

  9 – 0

  a – 3

  b – 1

  c – 0

  d – 0